UNIT-1

What is Data Structure?

The data structure name indicates itself that organizing the data in memory. There are many ways of organizing the data in the memory as we have already seen one of the data structures, i.e., array in C language.

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently.

Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artificial intelligence, Graphics and many more.

The data structure is not any programming language like C, C++, java, etc. It is a set of algorithms that we can use in any programming language to structure the data in the memory.

Types of Data Structures

There are two types of data structures:

- Primitive data structure
- Non-primitive data structure

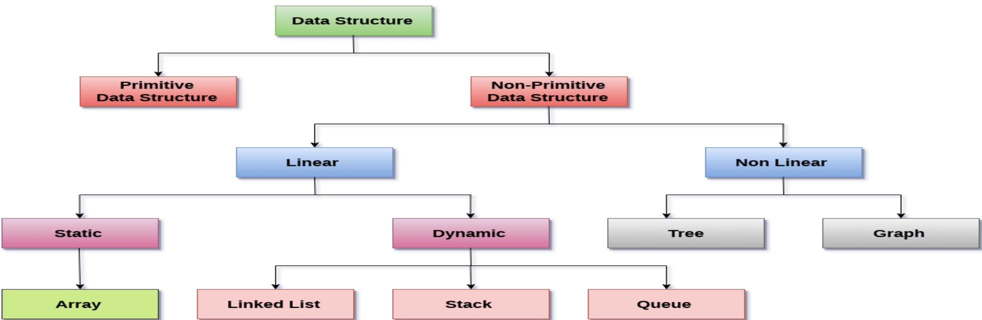
- **Primitive Data structure**

The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

- **Non-Primitive Data structure**

The non-primitive data structure is divided into two types:

- Linear data structure
- Non-linear data structure



Linear Data Structures:

A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

1. Arrays: An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array age are: age[0], age[1], age[2], age[3],......... age[98], age[99].

2. Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

3. Stack: Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.

A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

4. Queue: Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.

It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

Non-Linear Data Structures:

This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non-Linear Data Structures are given below:

1. Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottom most nodes in the hierarchy are called **leaf node** while the topmost node is called **root node**.

Each node contains pointers to point adjacent nodes. Tree data structure is based on the parent-child relationship among the nodes.

Each node in the tree can have more than one child except the leaf nodes whereas each node can have atmost one parent except the root node. Trees can be classified into many categories which will be discussed later in this tutorial.

2. Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree cannot have the one.

Data structures can also be classified as:

- **Static data structure:** It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.
- **Dynamic data structure:** It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

Major Operations

The major or the common operations that can be performed on the data structures are:

- **Searching:** We can search for any element in a data structure.
- **Sorting:** We can sort the elements of a data structure either in an ascending or descending order.
- **Insertion:** We can also insert the new element in a data structure.
- **Updation:** We can also update the element, i.e., we can replace the element with another element.
- **Deletion:** We can also perform the delete operation to remove the element from the data structure.

SEARCHING

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful, and the process returns the location of that element; otherwise, the search is called unsuccessful.

Two popular search methods are

(i) Linear Search and

(ii) Binary Search.

Linear Search

Linear search is also called as **sequential search algorithm.** It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found.

If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted. The worst-case time complexity of linear search is **O(n).**

The steps used in the implementation of Linear Search are listed as follows -

- First, we have to traverse the array elements using a **for** loop.
- In each iteration of **for loop,** compare the search element with the current array element, and -
  - If the element matches, then return the index of the corresponding array element.
  - If the element does not match, then move to the next element.
- If there is no match or the search element is not present in the given array, return **-1.**

Now, let's see the algorithm of linear search.

Algorithm

Linear_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search

Step 1: set pos = -1

Step 2: set i = 1

Step 3: repeat step 4 while i **<**= n

Step 4: if a[i] == val

set pos = i

print pos

go to step 6

[end of if]

set ii = i + 1

[end of loop]

Step 5: if pos = -1

print "value is not present in the array "

[end of if]

Step 6: exit

Working of Linear search

Now, let's see the working of the linear search Algorithm.

To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.

Let the elements of array are -

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

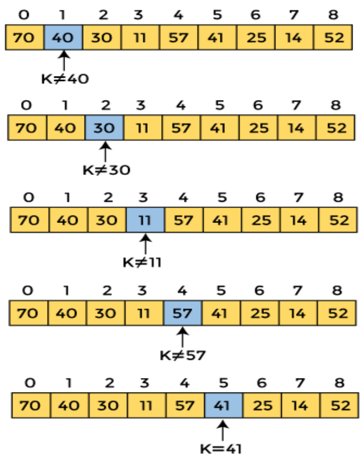
Let the element to be searched is **K = 41**

Now, start from the first element and compare **K** with each element of the array.

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

K≠70

The value of **K,** i.e., **41,** is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.



Now, the element to be searched is found. So algorithm will return the index of the element matched.

Python3 code to linearly search x in arr[].

If x is present then return its location,

```
# otherwise return -1
def search(arr, n, x):

	for i in range(0, n):
		if (arr[i] == x):
			return i
	return -1
# Driver Code
arr = [2, 3, 4, 10, 40]
x = 10
n = len(arr)
# Function call
result = search(arr, n, x)
if(result == -1):
	print("Element is not present in array")
else:
	print("Element is present at index", result)
```

Binary search

Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.

Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list.

If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

Now, let's see the algorithm of Binary Search.

Algorithm

```
Binary_Search(a, lower_bound, upper_bound, val) // 'a' is the given array, 'lower_bound' is the
 index of the first array element, 'upper_bound' is the index of the last array element, 'val' is th
e value to search
Step 1: set beg = lower_bound, end = upper_bound, pos = - 1
Step 2: repeat steps 3 and 4 while beg <=end
Step 3: set mid = (beg + end)/2
Step 4: if a[mid] = val
set pos = mid
print pos
go to step 6
else if a[mid] > val
set end = mid - 1
```

else

set beg = mid + 1

[end of if]

[end of loop]

Step 5: if pos = -1

print "value is not present in the array"

[end of if]

Step 6: exit

Working of Binary search

Now, let's see the working of the Binary Search Algorithm.

To understand the working of the Binary search algorithm, let's take a sorted array. It will be easy to understand the working of Binary search with an example.

There are two methods to implement the binary search algorithm -

- Iterative method
- Recursive method

The recursive method of binary search follows the divide and conquer approach.

Let the elements of array are -

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

Let the element to search is, **K = 56**

We have to use the below formula to calculate the **mid** of the array -

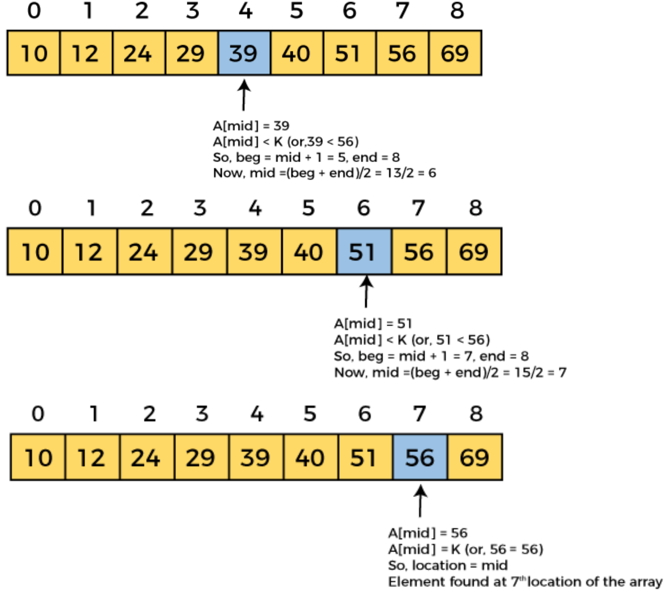
1. mid = (beg + end)/2

So, in the given array -

beg = 0

end = 8

mid = (0 + 8)/2 = 4. So, 4 is the mid of the array.



Now, the element to search is found. So algorithm will return the index of the element matched.

Sorting: Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order.

Consider an array;

int A[10] = { 5, 4, 10, 2, 30, 45, 34, 14, 18, 9)

The Array sorted in ascending order will be given as;

A[] = { 2, 4, 5, 9, 10, 14, 18, 30, 34, 45 }

Different types of Sorting algorithms are

1. Bubble sort
2. Selection sort
3. Quick sort
4. Merge sort
5. Insertion sort

1. Bubble sort Algorithm

Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. The array elements in bubble sort move to the end in each iteration.

Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world. It is not suitable for large data sets. The average and worst-case complexity of Bubble sort is **$O(n^2)$,** where **n** is a number of items.

- Bubble sort is majorly used where -complexity does not matter
- simple and shortcode is preferred

Algorithm

In the algorithm given below, suppose **arr** is an array of **n** elements. The assumed **swap** function in the algorithm will swap the values of given array elements.

```
begin BubbleSort(arr)
 for all array elements
  if arr[i] > arr[i+1]
   swap(arr[i], arr[i+1])
```

```
    end if
  end for
  return arr
end BubbleSort
```

Working of Bubble sort Algorithm

Now, let's see the working of Bubble sort Algorithm.

To understand the working of bubble sort algorithm, let's take an unsorted array. We are taking a short and accurate array, as we know the complexity of bubble sort is **$O(n^2)$.**

Let the elements of array are -

13	32	26	35	10

First Pass

Sorting will start from the initial two elements. Let compare them to check which is greater.

13	32	26	35	10

Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.

13	32	26	35	10

Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -

13	26	32	35	10

Now, compare 32 and 35.

13	26	32	35	10

Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.

Now, the comparison will be in between 35 and 10.

13	26	32	35	10

Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -

13	26	32	10	35

Now, move to the second iteration.

Second Pass

The same process will be followed for second iteration.

13	26	32	10	35

13	26	32	10	35

13	26	32	10	35

Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -

13	26	10	32	35

13	26	10	32	35

Now, move to the third iteration.

Third Pass

The same process will be followed for third iteration.

13	26	10	32	35

13	26	10	32	35

Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -

13	10	26	32	35

13	10	26	32	35

13	10	26	32	35

Now, move to the fourth iteration.

Fourth pass

Similarly, after the fourth iteration, the array will be -

10	13	26	32	35

Hence, there is no swapping required, so the array is completely sorted.

Bubble sort complexity

Now, let's see the time complexity of bubble sort in the best case, average case, and worst case. We will also see the space complexity of bubble sort.

Time Complexity

Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

Space Complexity

Space Complexity	O(1)
Stable	YES

2. Selection Sort Algorithm:

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.

In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

The average and worst-case complexity of selection sort is **$O(n^2)$**, where **n** is the number of items. Due to this, it is not suitable for large data sets.

Selection sort is generally used when -

- A small array is to be sorted
- Swapping cost doesn't matter
- It is compulsory to check all elements

Now, let's see the algorithm of selection sort.

Algorithm

SELECTION SORT(arr, n)

Step 1: Repeat Steps 2 **and** 3 **for** i = 0 to n-1

Step 2: CALL SMALLEST(arr, i, n, pos)

Step 3: SWAP arr[i] with arr[pos]

[END OF LOOP]

Step 4: EXIT

SMALLEST (arr, i, n, pos)

Step 1: [INITIALIZE] SET SMALL = arr[i]

Step 2: [INITIALIZE] SET pos = i

Step 3: Repeat **for** j = i+1 to n

if (SMALL > arr[j])

SET SMALL = arr[j]

SET pos = j

[END OF **if**]

[END OF LOOP]

Step 4: RETURN pos

Working of Selection sort Algorithm

Now, let's see the working of the Selection sort Algorithm.

To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.

Let the elements of array are -

12	29	25	8	32	17	40

Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.

12	29	25	8	32	17	40

So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.

8	29	25	12	32	17	40

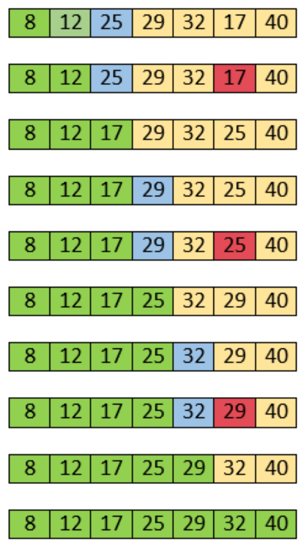
For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

8	29	25	12	32	17	40

Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.

8	12	25	29	32	17	40

The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.



Now, the array is completely sorted.

Selection sort complexity

Now, let's see the time complexity of selection sort in best case, average case, and in worst case. We will also see the space complexity of the selection sort.

Time Complexity

Case	**Time Complexity**
Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of selection sort is O(1). It is because, in selection sort, an extra variable is required for swapping.

3. Quick Sort Algorithm

Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements.

It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach.

Divide and conquer is a technique of breaking down the algorithms into sub problems, then solving the sub problems, and combining the results back together to solve the original problem.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

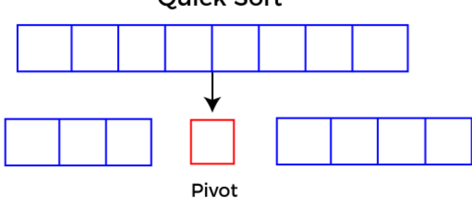
Conquer: Recursively, sort two sub arrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element.

In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element or the leftmost element of the given array.
- Select median as the pivot element.

Algorithm:

```
QUICKSORT (array A, start, end)
{
 if (start < end)
 {
 p = partition(A, start, end)
 QUICKSORT (A, start, p - 1)
 QUICKSORT (A, p + 1, end)
 }
}
```

Partition Algorithm:

The partition algorithm rearranges the sub-arrays in a place.

```
PARTITION (array A, start, end)
{
 pivot = A[end]
 i = start-1
 for j = start to end -1 {
 do if (A[j] < pivot) {
 then i = i + 1
 swap A[i] with A[j]
  }}
 swap A[i+1] with A[end]
 return i+1
}
```

Working of Quick Sort Algorithm

Now, let's see the working of the Quicksort Algorithm.

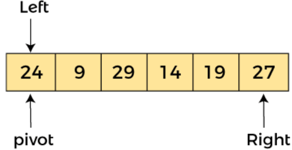
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

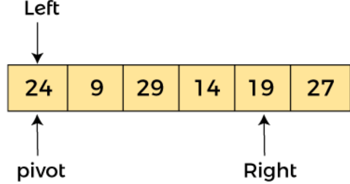
24	9	29	14	19	27

In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

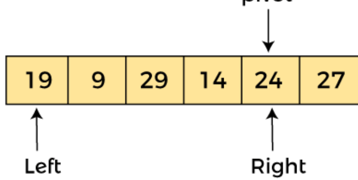


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



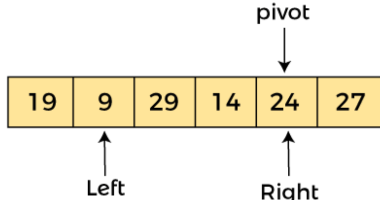
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

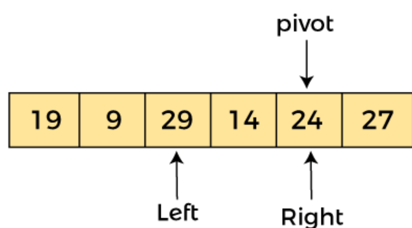


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

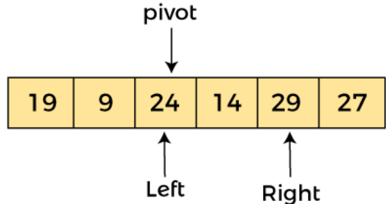
As a[pivot] > a[left], so algorithm moves one position to right as -



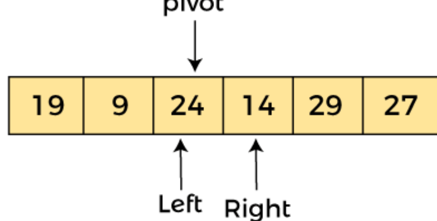
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



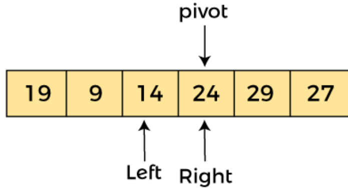
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



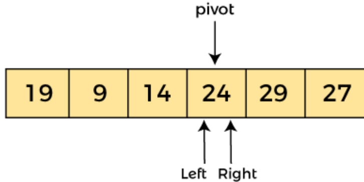
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



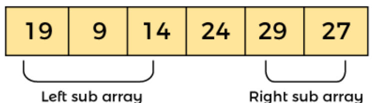
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -

9	14	19	24	27	29

Quicksort complexity

Now, let's see the time complexity of quicksort in best case, average case, and in worst case. We will also see the space complexity of quicksort.

Time Complexity

Case	Time Complexity
Best Case	O(n*logn)

Average Case	O(n*logn)
Worst Case	$O(n^2)$

Space Complexity

Space Complexity	O(n*logn)
Stable	NO

- The space complexity of quicksort is O(n*logn).

4. Merge Sort Algorithm:

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm.

It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process.

The sorted two-element pairs are merged into the four-element lists, and so on until we get the sorted list. Now, let's see the algorithm of merge sort.

Algorithm

In the following algorithm, **arr** is the given array, **beg** is the starting element, and **end** is the last element of the array.

```
MERGE_SORT(arr, beg, end)

 if beg < end

set mid = (beg + end)/2

MERGE_SORT(arr, beg, mid)

MERGE_SORT(arr, mid + 1, end)

MERGE (arr, beg, mid, end)

end of if

 END MERGE_SORT
```

The important part of the merge sort is the **MERGE** function. This function performs the merging of two sorted sub-arrays that are **A[beg…mid]** and **A[mid+1…end]**, to build one sorted array **A[beg…end]**. So, the inputs of the **MERGE** function are **A[], beg, mid,** and **end**.

The implementation of the **MERGE** function is given as follows –

```
/* Function to merge the subarrays of a[] */

void merge(int a[], int beg, int mid, int end)

{

   int i, j, k;

   int n1 = mid - beg + 1;

   int n2 = end - mid;

    int LeftArray[n1], RightArray[n2]; //temporary arrays

       /* copy data to temp arrays */
```

```
    for (int i = 0; i < n1; i++)
    LeftArray[i] = a[beg + i];
    for (int j = 0; j < n2; j++)
    RightArray[j] = a[mid + 1 + j];
    i = 0, /* initial index of first sub-array */
    j = 0; /* initial index of second sub-array */
    k = beg;  /* initial index of merged sub-array */
       while (i < n1 && j < n2)
    {
      if(LeftArray[i] <= RightArray[j])
      {
        a[k] = LeftArray[i];
       i++;
      }
      else
      {
        a[k] = RightArray[j];
        j++;
      }
      k++;
    }
    while (i<n1)
    {
      a[k] = LeftArray[i];
      i++;
      k++;
    }
       while (j<n2)
    {
      a[k] = RightArray[j];
      j++;
      k++;
    }
}
```

Working of Merge sort Algorithm

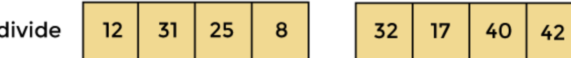
Now, let's see the working of merge sort Algorithm.

To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are -

12 | 31 | 25 | 8 | 32 | 17 | 40 | 42

According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.
As there are eight elements in the given array, so it is divided into two arrays of size 4.



Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.

divide [12 | 31] [25 | 8] [32 | 17] [40 | 42]

Now, again divide these arrays to get the atomic value that cannot be further divided.

divide [12] [31] [25] [8] [32] [17] [40] [42]

Now, combine them in the same manner they were broken.
In combining, first compare the element of each array and then combine them into another array in sorted order.
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.

merge [12 | 31] [8 | 25] [17 | 32] [40 | 42]

In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.

merge [8 | 12 | 25 | 31] [17 | 32 | 40 | 42]

Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -

[8 | 12 | 17 | 25 | 31 | 32 | 40 | 42]

Now, the array is completely sorted.

Merge sort complexity

Now, let's see the time complexity of merge sort in best case, average case, and in worst case. We will also see the space complexity of the merge sort.

Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n*logn)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **O(n*logn)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **O(n*logn)**.

2. Space Complexity

Space Complexity	O(n)
Stable	YES

- The space complexity of merge sort is O(n). It is because, in merge sort, an extra variable is required for swapping.

5. Insertion Sort Algorithm:

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card.

If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array.

Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is **$O(n^2)$**, where n is the number of items.

Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as –

- Simple implementation
- Efficient for small data sets
- Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

Now, let's see the algorithm of insertion sort.

Algorithm

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step2 - Pick the next element, and store it separately in a **key.**

Step3 - Now, compare the **key** with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

Working of Insertion sort Algorithm

Now, let's see the working of the insertion sort Algorithm.

To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.

Let the elements of array are -

12	31	25	8	32	17

Initially, the first two elements are compared in insertion sort.

12	31	25	8	32	17

Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.

12	31	25	8	32	17

Now, move to the next two elements and compare them.

12	31	25	8	32	17

12	31	25	8	32	17

Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

12	25	31	8	32	17

Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

12	25	31	8	32	17

12	25	31	8	32	17

Both 31 and 8 are not sorted. So, swap them.

12	25	8	31	32	17

After swapping, elements 25 and 8 are unsorted.

12	25	8	31	32	17

So, swap them.

12	8	25	31	32	17

Now, elements 12 and 8 are unsorted.

12	8	25	31	32	17

So, swap them too.

8	12	25	31	32	17

Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.

8	12	25	31	32	17

Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

8	12	25	31	32	17

Move to the next elements that are 32 and 17.

8	12	25	31	32	17

17 is smaller than 32. So, swap them.

8	12	25	31	17	32

8	12	25	31	17	32

Swapping makes 31 and 17 unsorted. So, swap them too.

8	12	25	17	31	32

8	12	25	17	31	32

Now, swapping makes 25 and 17 unsorted. So, perform swapping again.

8	12	17	25	31	32

Now, the array is completely sorted.

Insertion sort complexity

Now, let's see the time complexity of insertion sort in best case, average case, and in worst case. We will also see the space complexity of insertion sort.

Time Complexity

Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is **O(n)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is **$O(n^2)$**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is **$O(n^2)$**.

Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of insertion sort is O(1). It is because, in insertion sort, an extra variable is required for swapping.

UNIT-II

STACKS INTRODUCTION:

A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, It contains only one pointer **top pointer** pointing to the topmost element of the stack.

Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

Some key points related to stack

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Stack Operations:

The following are some common operations implemented on the stack:

- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full or not.'
- **peek():** It returns the element at the given position.
- **count():** It returns the total number of elements available in a stack.
- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.

PUSH operation:

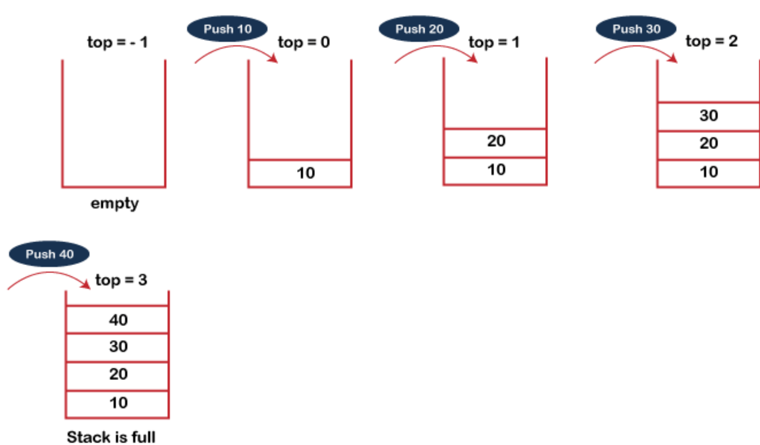
The steps involved in the PUSH operation is given below:

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.

- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the ***max*** size of the stack.

Algorithm for push:

```
begin
 if stack is full
    return
 endif
else
 increment top
 stack[top] assign value
end else
end procedure
```



POP operation:

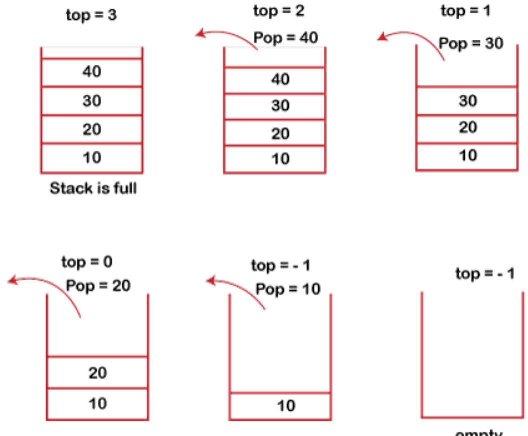
The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the ***top***
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.

Algorithm for pop:

```
begin
 if stack is empty
    return
 endif
else
 store value of stack[top]
```

decrement top
return value
end else
end procedure



Implementation of Stacks using Arrays:

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Let's see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refer to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm:

```
begin
   if top = n then stack full
   top = top + 1
   stack (top) : = item;
end
```

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be decremented by 1 whenever an item is deleted from the stack.

The top most element of the stack is stored in another variable and then the top is decremented by 1. The operation returns the deleted value that was stored in another variable as the result.

The underflow condition occurs when we try to delete an element from an empty stack.

Algorithm :

```
begin
   if top = 0 then stack empty;
   item := stack(top);
   top = top - 1;
end;
```

```
# Stack implementation in python


# Creating a stack
def create_stack():
   stack = []
   return stack


# Creating an empty stack
def check_empty(stack):
   return len(stack) == 0


# Adding items into the stack
def push(stack, item):
   stack.append(item)
   print("pushed item: " + item)


# Removing an element from the stack
def pop(stack):
   if (check_empty(stack)):
      return "stack is empty"

   return stack.pop()


stack = create_stack()
push(stack, str(1))
push(stack, str(2))
push(stack, str(3))
push(stack, str(4))
print("popped item: " + pop(stack))
print("stack after popping an element: " + str(stack))
```

Output :

```
pushed item: 1
pushed item: 2
pushed item: 3
pushed item: 4
popped item: 4
stack after popping an element: ['1', '2', '3']
```

Pros: Easy to implement. Memory is saved as pointers are not involved.
Cons: It is not dynamic. It doesn't grow and shrink depending on needs at runtime.

APPLICATIONS:

Expression

An expression can be defined as a collection of operators and operands that represents a specific value.

In above definition, **operator** is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,

Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types

Based on the operator position, expressions are divided into THREE types. They are as follows...

1. **Infix Expression**
2. **Postfix Expression**
3. **Prefix Expression**

1. Infix Expression

In infix expression, operator is used in between the operands. The general structure of an Infix expression is as follows...

Operand1 Operator Operand2

Ex: a+b

2. Postfix Expression

In postfix expression, operator is used after operands. We can say that "**Operator follows the Operands**". The general structure of Postfix expression is as follows...

Operand1 Operand2 Operator

Ex: ab+

3. Prefix Expression

In prefix expression, operator is used before operands. We can say that "**Operands follows the Operator**". The general structure of Prefix expression is as follows...

Operator Operand1 Operand2

Ex:+ab

Every expression can be represented using all the above three different types of expressions. And we can convert an expression from one form to another form like **Infix to Postfix**, **Infix to Prefix**, **Prefix to Postfix** and vice versa.

Conversion from Infix to Postfix notation:

Problem with infix notation:

To evaluate the infix expression, we should know about the **operator precedence** rules, and if the operators have the same precedence, then we should follow the **associativity** rules.

The use of parenthesis is very important in infix notation to control the order in which the operation to be performed. An infix expression is the most common way of writing expression, but it is not easy to parse and evaluate the infix expression without ambiguity.

So, mathematicians and logicians discovered two other ways of writing expressions which are prefix and postfix. Both expressions do not require any parenthesis and can be parsed without ambiguity. It does not require operator precedence and associativity rules.

Conversion of infix to postfix

Here, we will use the stack data structure for the conversion of infix expression to prefix expression. Whenever an operator will encounter, we push operator into the stack. If we encounter an operand, then we append the operand to the expression.

Rules for the conversion from infix to postfix expression

1. Print the operand as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator on to the stack.
3. If the incoming symbol is '(', push it on to the stack.
4. If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has lower precedence than the top of the stack, pop and print the top of the stack. Then test the incoming operator against the new top of the stack.
7. If the incoming operator has the same precedence with the top of the stack then use the associativity rules. If the associativity is from left to right then pop and print the top of the stack then push the incoming operator. If the associativity is from right to left then push the incoming operator.
8. At the end of the expression, pop and print all the operators of the stack.

Let's understand through an example.

Infix expression: K + L - M*N + (O^P) * W/U/V * T + Q

Input Expression	Stack	Postfix Expression
K		K
+	+	
L	+	K L
-	-	K L+
M	-	K L+ M
*	- *	K L+ M
N	- *	K L + M N
+	+	KL+MN* K L + M N* -
(	+ (	K L + M N *-
O	+ (	K L + M N * - O
^	+ (^	K L + M N* - O
P	+ (^	K L + M N* - O P
)	+	K L + M N* - O P ^
*	+ *	K L + M N* - O P ^
W	+ *	K L + M N* - O P ^ W
/	+ /	K L + M N* - O P ^ W *
U	+ /	K L + M N* - O P ^W*U
/	+ /	K L + M N* - O P ^W*U/
V	+ /	KL + MN*-OP^W*U/V
*	+ *	KL+MN*-OP^W*U/V/
T	+ *	KL+MN*-OP^W*U/V/T
+	+	KL+MN*-OP^W*U/V/T* KL+MN*-OP^W*U/V/T*+
Q	+	KL+MN*-OP^W*U/V/T*Q
		KL+MN*-OP^W*U/V/T*+Q+

The final postfix expression of infix expression **(K + L - M*N + (O^P) * W/U/V * T + Q)** is **KL+MN*-OP^W*U/V/T*+Q+**

#python code to convert from infix to postfix

```
OPERATORS = set(['+', '-', '*', '/', '(', ')', '^'])  # set of operators
PRIORITY = {'+':1, '-':1, '*':2, '/':2, '^':3} # dictionary having priorities
def infix_to_postfix(expression): #input expression
    stack = [] # initially stack empty
    output = '' # initially output empty
      for ch in expression:
       if ch not in OPERATORS:  # if an operand then put it directly in postfix expression
           output+= ch
       elif ch=='(':  # else operators should be put in stack
           stack.append('(')
       elif ch==')':
           while stack and stack[-1]!= '(':
               output+=stack.pop()
           stack.pop()
       else:
           # lesser priority can't be on top on higher or equal priority
            # so pop and put in output
           while stack and stack[-1]!='(' and PRIORITY[ch]<=PRIORITY[stack[-1]]:
               output+=stack.pop()
           stack.append(ch)
    while stack:
        output+=stack.pop()
    return output
expression = input('Enter infix expression')
print('infix expression: ',expression)
print('postfix expression: ',infix_to_postfix(expression))
```

Output:

```
Enter infix expression a+b-c*d^e/f
infix expression:   a+b-c*d^e/f
postfix expression:   ab+cde^*f/-
```

Evaluation of postfix expression:

Postfix expression:

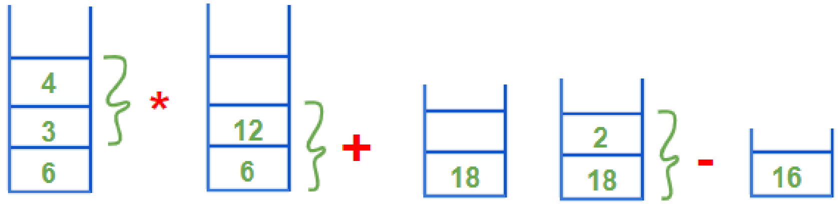
A postfix expression can be represented as: <operand><operand><operator>. Operator is succeeded by operands eg: XY+.

Rules:

1. Scan each value in the expression
2. If character == operand, put in stack
3. If character == operator, pop first 2 elements from stack
4. Use that operator on popped elements and put result in stack
5. Repeat above steps till you get final result

Example:

Consider a postfix expression = 634*+2-



Result = 16

```
OPERATORS=set(['*','-','+','%','/','**']) # set of operators allowed in expression
def evaluate_postfix(expression):
   stack=[] # empty stack for storing numbers
   for i in expression:
      if i not in OPERATORS:
         stack.append(i)  #contains numbers
      else:
         a=stack.pop() # if ch==operator then pop 2 numbers
         b=stack.pop()
         if i=='+':
            res=int(b)+int(a) # old val <operator> recent value
         elif i=='-':
            res=int(b)-int(a)
         elif i=='*':
            res=int(b)*int(a)
         elif i=='%':
            res=int(b)%int(a)
         elif i=='/':
            res=int(b)/int(a)
         elif i=='**':
            res=int(b)**int(a)
         stack.append(res) # final result
   return(''.join(map(str,stack)))
expression = input('Enter the postfix expression ')
print()
print('postfix expression entered: ',expression)
print('Evaluation result: ',evaluate_postfix(expression))
```

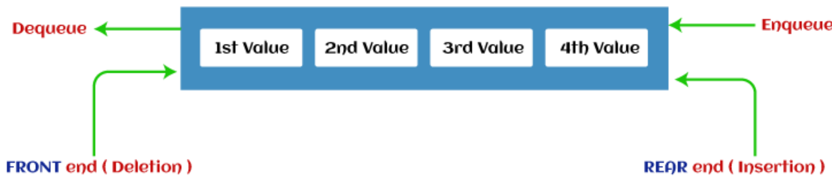
QUEUES:

Queue is the data structure that is similar to the queue in the real world. A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy.

Queue can also be defined as the list or collection in which the insertion is done from one end known as the **rear end** or the **tail** of the queue, whereas the deletion is done from another end known as the **front end** or the **head** of the queue.

The real-world example of a queue is the ticket queue outside a cinema hall, where the person who enters first in the queue gets the ticket first, and the last person enters in the queue gets the ticket at last. Similar approach is followed in the queue in data structure.

The representation of the queue is shown in the below image -



Now, let's move towards the types of queue.

Types of Queue

There are four different types of queue that are listed as follows –

- Simple Queue or Linear Queue
- Circular Queue
- Priority Queue
- Double Ended Queue (or Deque)

Operations performed on queue

The fundamental operations that can be performed on queue are listed as follows -

- **Enqueue:** The Enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **Dequeue:** It performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value.
- **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
- **Queue overflow (isfull):** It shows the overflow condition when the queue is completely full.
- **Queue underflow (isempty):** It shows the underflow condition when the Queue is empty, i.e., no elements are in the Queue.

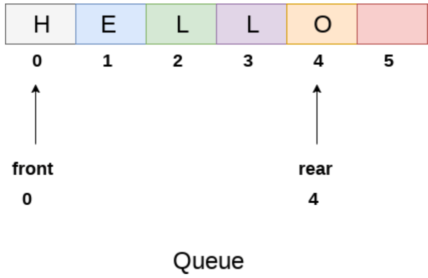
Now, let's see the ways to implement the queue.

Array representation of Queue

We can easily represent queue by using linear arrays. There are two variables i.e. front and rear that are implemented in the case of every queue.

Front and rear variables point to the position from where insertions and deletions are performed in a queue.

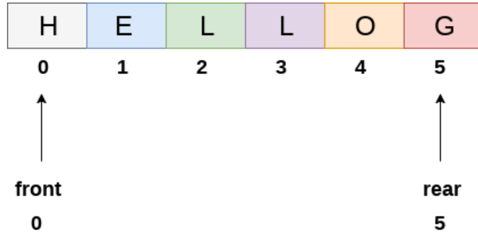
Initially, the value of front and rear is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



Queue

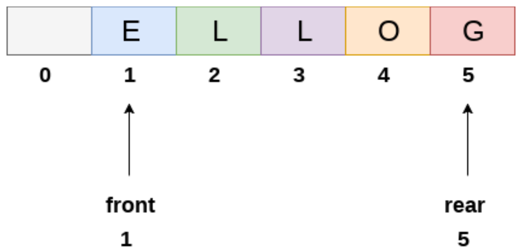
The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 .

However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

- **Step1:** IF REAR = MAX - 1
  Write OVERFLOW
  Go to step
  [END OF IF]
- **Step 2:** IF FRONT = -1 and REAR = -1
  SET FRONT = REAR = 0
  ELSE
  SET REAR = REAR + 1
  [END OF IF]
- **Step 3:** Set QUEUE[REAR] = NUM
- **Step 4:** EXIT

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear, write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm

- **Step 1:** IF FRONT = -1 or FRONT > REAR
  Write UNDERFLOW
  ELSE

SET VAL = QUEUE[FRONT]

SET FRONT = FRONT + 1

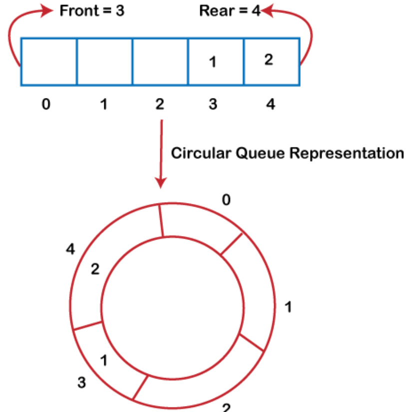
[END OF IF]

- **Step 2:** EXIT

CIRCULAR QUEUE:

There was one limitation in the array implementation of Queue

If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.



As we can see in the above image, the rear is at the last position of the Queue and front is pointing somewhere rather than the 0th position.

In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue.

There is one solution to avoid such wastage of memory space by shifting both the elements at the left and adjust the front and rear end accordingly.

It is not a practically good approach because shifting all the elements will consume lots of time. The efficient approach to avoid the wastage of the memory is to use the circular queue data structure.

What is a Circular Queue?

A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a ***Ring Buffer***.

Operations on Circular Queue

The following are the operations that can be performed on a circular queue:

- **Front:** It is used to get the front element from the Queue.

- **Rear:** It is used to get the rear element from the Queue.
- **enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
- **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.

Applications of Circular Queue

The circular Queue can be used in the following scenarios:

- **Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.
- **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.
- **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every jinterval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Enqueue operation

The steps of enqueue operation are given below:

- First, we will check whether the Queue is full or not.
- Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
- When we insert a new element, the rear gets incremented, i.e., ***rear=rear+1***.

Scenarios for inserting an element

There are two scenarios in which queue is not full:

- **If rear != max - 1**, then rear will be incremented to **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
- **If front != 0 and rear = max - 1**, it means that queue is not full, then set the value of rear to 0 and insert the new element there.

There are two cases in which the element cannot be inserted:

- When **front ==0** && **rear = max-1**, which means that front is at the first position of the Queue and rear is at the last position of the Queue.

- front== rear + 1;

Algorithm to insert an element in a circular queue

Step 1: IF (REAR+1)%MAX = FRONT
Write " OVERFLOW "
Goto step 4
[End OF IF]

Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE IF REAR = MAX - 1 and FRONT ! = 0
SET REAR = 0
ELSE
SET REAR = (REAR + 1) % MAX
[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT

Dequeue Operation

The steps of dequeue operation are given below:

- First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by 1.
- If there is only one element left which is to be deleted, then the front and rear are reset to -1.

Algorithm to delete an element from the circular queue

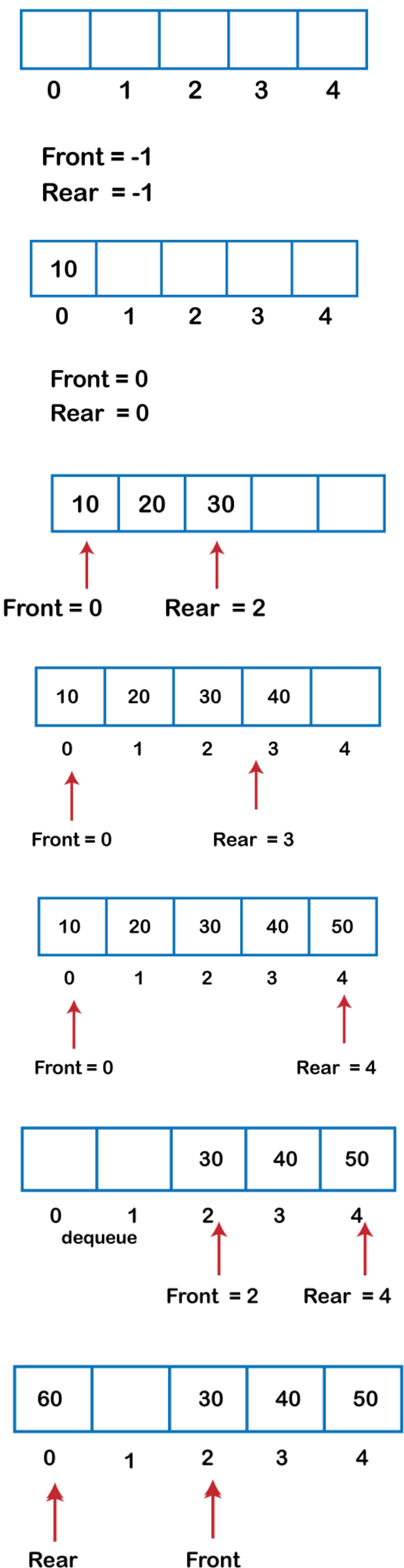
Step 1: IF FRONT = -1
Write " UNDERFLOW "
Goto Step 4
[END of IF]

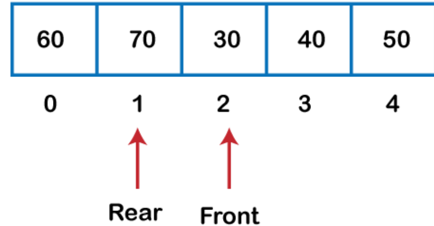
Step 2: SET VAL = QUEUE[FRONT]

Step 3: IF FRONT = REAR
SET FRONT = REAR = -1
ELSE
IF FRONT = MAX -1
SET FRONT = 0
ELSE
SET FRONT = FRONT + 1
[END of IF]
[END OF IF]

Step 4: EXIT

Let's understand the enqueue and dequeue operation through the diagrammatic representation.





DEQUE (OR DOUBLE-ENDED QUEUE)

The deque stands for Double Ended Queue. Deque is a linear data structure where the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule. The representation of a deque is given as follows -

Representation of deque

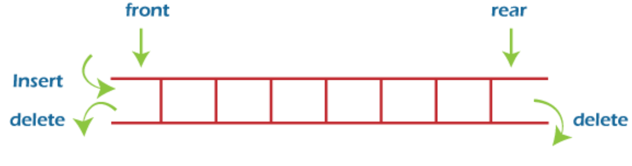
Types of deque

There are two types of deque -

- Input restricted queue
- Output restricted queue

Input restricted Queue

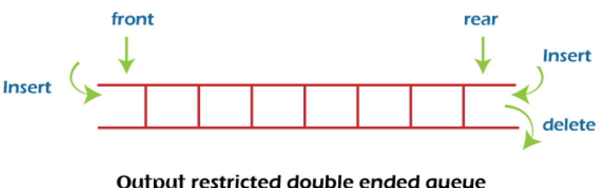
In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

Output restricted Queue

In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Operations performed on deque

There are the following operations that can be applied on a deque -

- Insertion at front
- Insertion at rear
- Deletion at front
- Deletion at rear

We can also perform peek operations in the deque along with the operations listed above. Through peek operation, we can get the deque's front and rear elements of the deque. So, in addition to the above operations, following operations are also supported in deque –

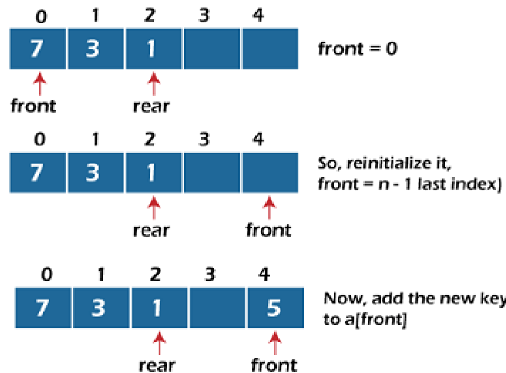
- Get the front item from the deque
- Get the rear item from the deque
- Check whether the deque is full or not
- Checks whether the deque is empty or not

Now, let's understand the operation performed on deque using an example.

Insertion at the front end

In this operation, the element is inserted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is full or not. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

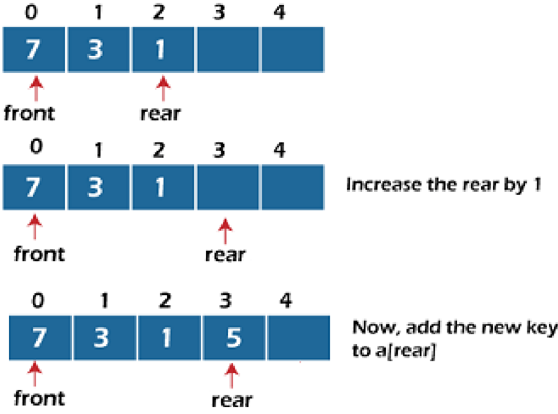
- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by **front = n - 1**, i.e., the last index of the array.



Insertion at the rear end

In this operation, the element is inserted from the rear end of the queue. Before implementing the operation, we first have to check again whether the queue is full or not. If the queue is not full, then the element can be inserted from the rear end by using the below conditions –

- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, increment the rear by 1. If the rear is at last index (or size - 1), then instead of increasing it by 1, we have to make it equal to 0.



Deletion at the front end

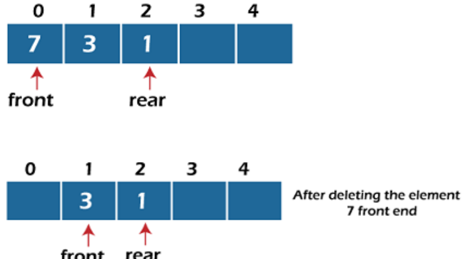
In this operation, the element is deleted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

If the deque has only one element, set rear = -1 and front = -1.

Else if front is at end (that means front = size - 1), set front = 0.

Else increment the front by 1, (i.e., front = front + 1).



Deletion at the rear end

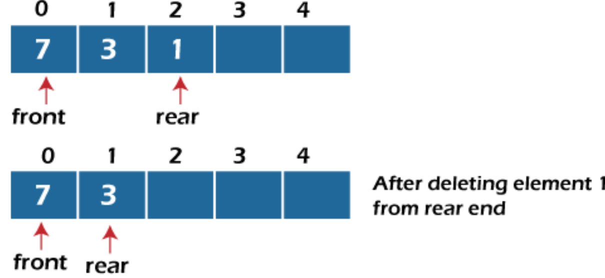
In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion.

If the deque has only one element, set rear = -1 and front = -1.

If rear = 0 (rear is at front), then set rear = n - 1.

Else, decrement the rear by 1 (or, rear = rear -1).



Check empty

This operation is performed to check whether the deque is empty or not. If front = -1, it means that the deque is empty.

Check full

This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n - 1 it means that the deque is full.

The time complexity of all of the above operations of the deque is O(1), i.e., constant.

Applications of deque

- Deque can be used as both stack and queue, as it supports both operations.
- Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.

PRIORITY QUEUE:

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue.

The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Let's understand the priority queue through an example.

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

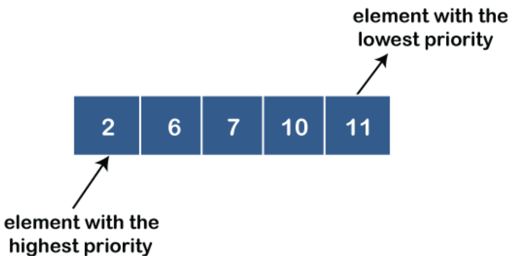
All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

- **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- **add(2):** This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- **poll():** It will remove '2' element from the priority queue as it has the highest priority queue.
- **add(5):** It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

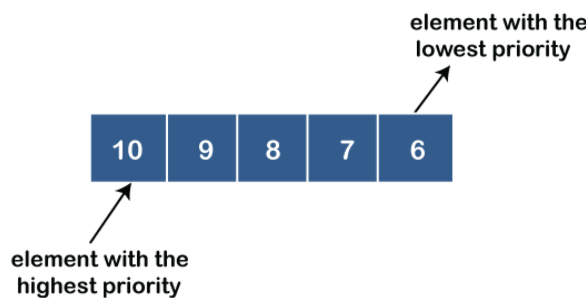
Types of Priority Queue

There are two types of priority queue:

- **Ascending order priority queue:** In ascending order priority queue, a lower number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e.,1 is given as the highest priority in a priority queue.



- **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.



Representation of priority queue

Now, we will see how to represent the priority queue through a one-way list.

We will create the priority queue by using the list given below in which **INFO** list contains the data elements, **PRN** list contains the priority numbers of each data element available in the **INFO** list, and LINK basically contains the address of the next node.

	INFO	PNR	LINK
0	200	2	4
1	400	4	2
2	500	4	6
3	300	1	0
4	100	2	5
5	600	3	1
6	700	4	

Let's create the priority queue step by step.

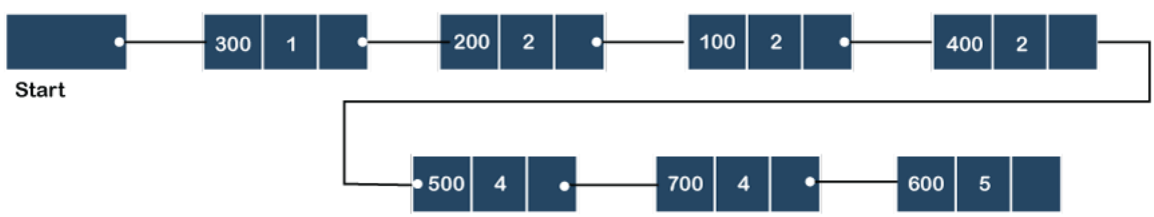
In the case of priority queue, lower value number is considered the higher priority, i.e., lower number = higher priority.

Step 1: In the list, lower priority number is 1, whose data value is 300, so it will be inserted in the list as shown in the below diagram:

Step 2: After inserting 300, priority number 2 is having a higher priority, and data values associated with this priority are 200 and 100. So, this data will be inserted based on the FIFO principle; therefore 200 will be added first and then 100.

Step 3: After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 400, 500, 700. In this case, elements would be inserted based on the FIFO principle; therefore, 400 will be added first, then 500, and then 700.

Step 4: After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 600, so it will be inserted at the end of the queue.



Implementation of Priority Queue

How to Implement Priority Queue?

Priority queue can be implemented using the following data structures:

- Arrays
- Linked list
- Heap data structure
- Binary search tree

Let’s discuss all these in detail.

1) Using Array: A simple implementation is to use an array of the following structure.

```
struct item {
  int item;
  int priority;
}
```

- **enqueue():** This function is used to insert new data into the queue.
- **dequeue():** This function removes the element with the highest priority from the queue.

- **peek()/top():** This function is used to get the highest priority element in the queue without removing it from the queue.

2) Using Linked List:

In a Linked List implementation, the entries are sorted in descending order based on their priority. The highest priority element is always added to the front of the priority queue, which is formed using linked lists.

The functions like **push()**, **pop()**, and **peek()** are used to implement priority queue using a linked list and are explained as follows:

- **push():** This function is used to insert new data into the queue.
- **pop():** This function removes the element with the highest priority from the queue.
- **peek() / top():** This function is used to get the highest priority element in the queue without removing it from the queue.

3) Using Heaps:

Binary Heap is generally preferred for priority queue implementation because heaps provide better performance compared to arrays or LinkedList. Operation on Binary Heap are as follows:

- **insert(p):** Inserts a new element with priority p.
- **extractMax():** Extracts an element with maximum priority.
- **remove(i):** Removes an element pointed by an iterator i.
- **getMax():** Returns an element with maximum priority.
- **changePriority(i, p):** Changes the priority of an element pointed by i to p.

4) Using Binary Search Tree:

A Self-Balancing Binary Search Tree like AVL Tree, Red-Black Tree, etc. can also be used to implement a priority queue. Operations like peek(), insert() and delete() can be performed using BST.

Binary Search Tree	peek()	insert()	delete()
Time Complexity	O(1)	O(log n)	O(log n)

What is the difference between Priority Queue and Normal Queue?

- There is no priority attached to elements in a queue, the rule of first-in-first-out(FIFO) is implemented whereas, in a priority queue, the elements have a priority. The elements with higher priority are served first.

Applications of Priority queue

The following are the applications of the priority queue:

- It is used in the Dijkstra's shortest path algorithm.
- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupts handling.

Application of a linear queue

Figure shows a basic diagram of a time-sharing system. A CPU (processor) endowed with memory resources, is to be shared by n number of computer users.

The sharing of the processor and memory resources is done by allotting a definite time slice of the processors attention on the users and in a round-robin fashion.

In a system such as this, the users are unaware of the presence of other users and are led to believe that their job receives the undivided attention of the CPU.

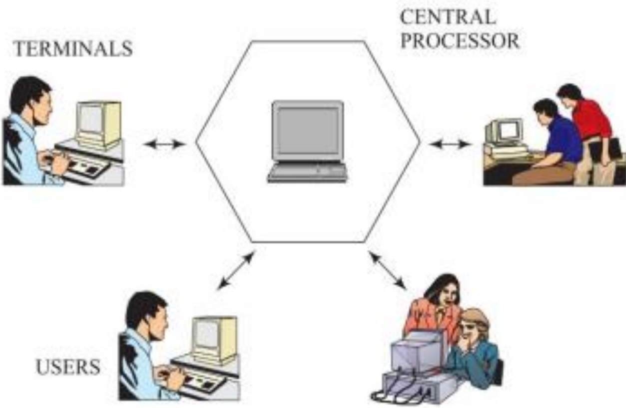


Fig. A basic diagram of a time-sharing system

However, to keep track of the jobs initiated by the users, the processor relies on a queue data structure recording the active user ids.

Example below demonstrates the application of a queue data structure for this job-scheduling problem.

The following is a table of three users A,B,C with their job requests J_i (k) where i is the job number and k is the time required to execute the job.

User	Job requests and the execution time in μ secs
A	J_1 (4), J_2 (3)
B	J_3 (2), J_4 (1), J_5 (1)
C	J_6 (6)

Thus J1 (4), a job request initiated by A needs 4 μ secs for its execution before the user initiates the next request of J2(3).

Throughout the simulation, we assume a uniform user delay period of 5 μ secs between any two sequential job requests initiated by a user. Thus B initiates J4 (1), 5 μ secs after the completion of J3 (2) and so on.

To initiate the simulation, we assume that A logged in at time 0, B at time 1 and C at time 2. Figure below shows a graphical illustration of the simulation.

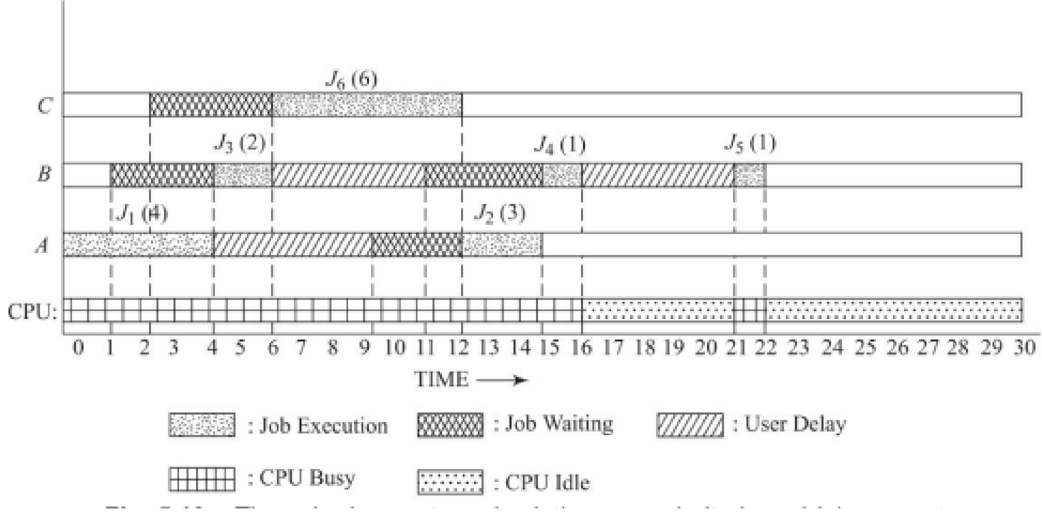


Fig: Time sharing system simulation non-priority based job requests

	Job Queue	
Time 5	B (J_3 (2))	C (J_6 (6))
Time 10	C (J_6 (6))	A (J_2 (3))
Time 14	A (J_2 (3))	C (J_4 (1))

Fig: Snapshot of the queue at times 5, 10 and 14

Application of priority queues:

Assume a time-sharing system in which job requests by users are of different categories. For example, some requests may be real time, the others online and the last may be batch processing requests.

It is known that real time job requests carry the highest priority, followed by online processing and batch processing in that order.

In such a situation the job scheduler needs to maintain a priority queue to execute the job requests based on their priorities.

If the priority queue were to be implemented using a cluster of queues of varying priorities, the scheduler has to maintain one queue for real time jobs (R), one for online processing jobs (O) and the third for batch processing jobs (B). The CPU proceeds to execute a job request in O only when R is empty.

Example: The following is a table of three users A,B,C with their job requests. Ri (k) indicates a real time job Ri whose execution time is k µ secs. Similarly Bi (k) and Oi (k) indicate batch processing and online processing jobs respectively.

User	Job requests and their execution time in μ secs
A	R_1 (4) B_1 (1)
B	O_1(2) O_2(3) B_2 (3)
C	R_2 (1) B_3 (2) O_3(3)

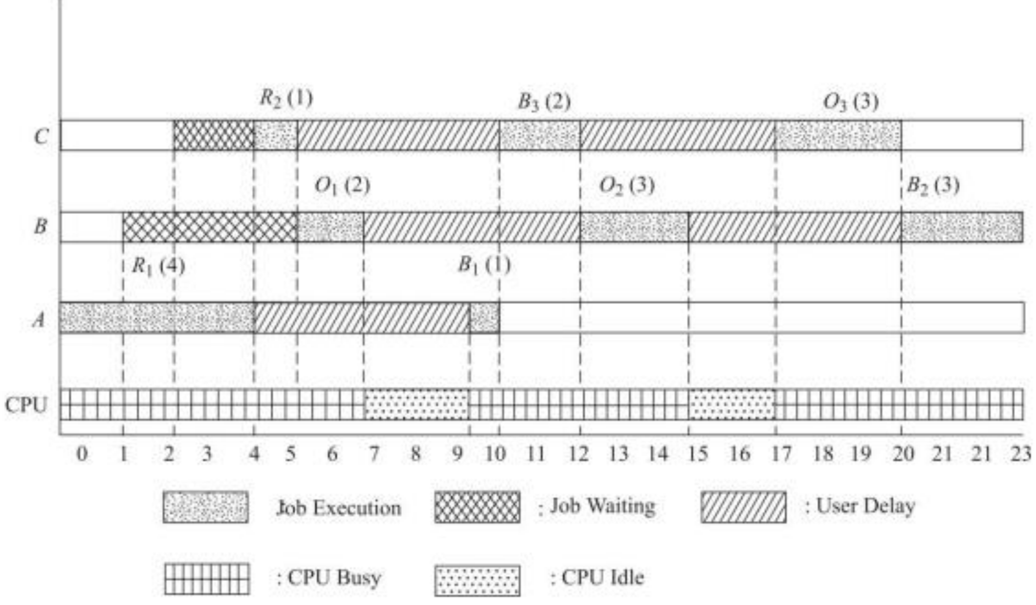


Fig. Simulation of the time sharing system for priority based jobs

At time 4

R
$R_1^{(4)}$ $R_2^{(1)}$
O
$O_1^{(2)}$
B

At time 8

R
O
B

At time 12

R
O
$O_2^{(3)}$
B
$B_3^{(2)}$

B: Batch Processing Queue R: Real Time Queue O: On-line Priority Queue

Fig. Snapshots of the priority queue at time 4, 8 and 12

UNIT-III

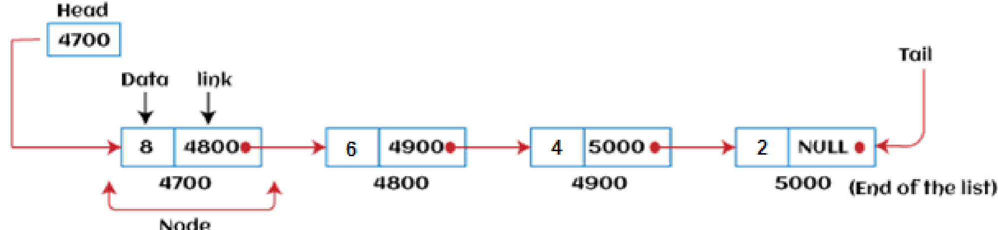
LINKED LIST

Linked list is a linear data structure that includes a series of connected nodes. Linked list can be defined as the nodes that are randomly stored in the memory.

A node in the linked list contains two parts, i.e., first is the data part and second is the address part. The last node of the list contains a pointer to the null. After array, linked list is the second most used data structure. In a linked list, every link contains a connection to another link.

Representation of a Linked list

Linked list can be represented as the connection of nodes in which each node points to the next node of the list. The representation of the linked list is shown below -



Why use linked list over array?

Linked list is a data structure that overcomes the limitations of arrays. Let's first see some of the limitations of arrays -

- The size of the array must be known in advance before using it in the program.
- Increasing the size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
- All the elements in the array need to be contiguously stored in the memory. Inserting an element in the array needs shifting of all its predecessors.

Linked list is useful because -

- It allocates the memory dynamically. All the nodes of the linked list are non-contiguously stored in the memory and linked together with the help of pointers.

- In linked list, size is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

Types of Linked list

Linked list is classified into the following types -

- **Singly-linked list -** Singly linked list can be defined as the collection of an ordered set of elements. A node in the singly linked list consists of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node, while the link part of the node stores the address of its immediate successor.
- **Doubly linked list -** Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly-linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), and pointer to the previous node (previous pointer).
- **Circular singly linked list -** In a circular singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.
- **Circular doubly linked list -** Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the nodes. The last node of the list contains the address of the first node of the list. The first node of the list also contains the address of the last node in its previous pointer.

Now, let's see the benefits and limitations of using the Linked list.

Advantages of Linked list

The advantages of using the Linked list are given as follows -

- **Dynamic data structure -** The size of the linked list may vary according to the requirements. Linked list does not have a fixed size.
- **Insertion and deletion -** Unlike arrays, insertion, and deletion in linked list is easier. Array elements are stored in the consecutive location, whereas the elements in the linked list are stored at a random location. To insert or delete

an element in an array, we have to shift the elements for creating the space. Whereas, in linked list, instead of shifting, we just have to update the address of the pointer of the node.

- **Memory efficient -** The size of a linked list can grow or shrink according to the requirements, so memory consumption in linked list is efficient.
- **Implementation -** We can implement both stacks and queues using linked list.

Disadvantages of Linked list

The limitations of using the Linked list are given as follows -

- **Memory usage -** In linked list, node occupies more memory than array. Each node of the linked list occupies two types of variables, i.e., one is a simple variable, and another one is the pointer variable.
- **Traversal -** Traversal is not easy in the linked list. If we have to access an element in the linked list, we cannot access it randomly, while in case of array we can randomly access it by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.
- **Reverse traversing -** Backtracking or reverse traversing is difficult in a linked list. In a doubly-linked list, it is easier but requires more memory to store the back pointer.

Applications of Linked list

The applications of the Linked list are given as follows -

- With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial.
- A linked list can be used to represent the sparse matrix.
- The various operations like student's details, employee's details, or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
- Using linked list, we can implement stack, queue, tree, and other various data structures.
- The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as

an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.

- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

Operations performed on Linked list

The basic operations that are supported by a list are mentioned as follows -

- **Insertion -** This operation is performed to add an element into the list.
- **Deletion -** It is performed to delete an operation from the list.
- **Display -** It is performed to display the elements of the list.
- **Search -** It is performed to search an element from the list using the given key.

Representation:

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head points to NULL.

Each node in a list consists of at least two parts:

1) data (we can store integer, strings or any type of data).

2) Pointer (Or Reference) to the next node (connects one node to another)

In python, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

```
# Node class

class Node:

    # Function to initialize the node object
    def __init__(self, data):
        self.data = data  # Assign data
        self.next = None  # Initialize
                          # next as null

# Linked List class
class LinkedList:

    # Function to initialize the Linked
    # List object
    def __init__(self):
        self.head = None
```

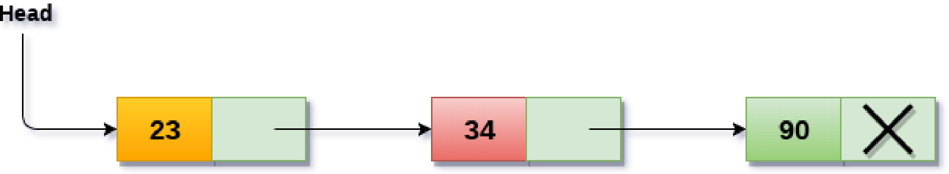
SINGLY LINKED LIST OR ONE WAY CHAIN:

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program.

A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.



In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject.

The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SN	Operation	Description

1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.

5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

<u>Insertion in singly linked list at beginning:</u>

Inserting a new element into a singly linked list at beginning is quite simple. We just need to make a few adjustments in the node links.

- Allocate the space for the new node and store data into the data part of the node.
- Make the link part of the new node pointing to the existing first node of the list.
- At the last, we need to make the new node as the first node of the list this will be done by using the following statement.

Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 6
[END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET NEW_NODE → DATA = VAL

Step 4: SET NEW_NODE → NEXT = HEAD

Step 5: SET HEAD = NEW_NODE

Step 6: EXIT

<u>Algorithm for insertion at end of the list:</u>

In order to insert a node at the last, there are two following scenarios which need to be mentioned.

1. The node is being added to an empty list
2. The node is being added to the end of the linked list

in the first case,

The condition (head == NULL) gets satisfied. Hence, we just need to allocate the space for the node.

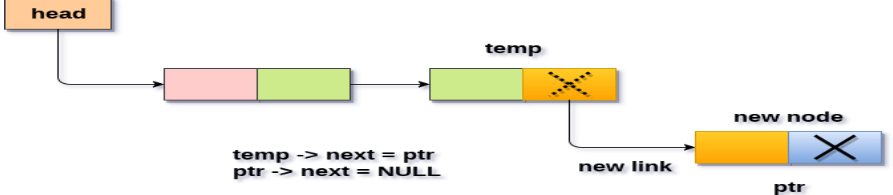
Since, **ptr** is the only node that will be inserted in the list hence, we need to make this node pointed by the head pointer of the list.

In the second case,

- The condition **Head = NULL** would fail, since Head is not null. Now, we need to declare a temporary pointer temp in order to traverse through the list. **temp** is made to point the first node of the list.

- Then, traverse through the entire linked list.
- At the end of the loop, the temp will be pointing to the last node of the list. Now, allocate the space for the new node, and assign the item to its data part.
- Since, the new node is going to be the last node of the list hence, the next part of this node needs to be pointing to the **null**.
- We need to make the next part of the temp node (which is currently the last node of the list) point to the new node (ptr) .

Step 1: IF PTR = NULL Write OVERFLOW
  Go to Step 9
  [END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET NEW_NODE - > DATA = VAL
Step 4: SET NEW_NODE - > NEXT = NULL
Step 5: SET PTR = HEAD
Step 6: Repeat Step 8 while PTR - > NEXT != NULL
Step 7: SET PTR = PTR - > NEXT
[END OF LOOP]
Step 8: SET PTR - > NEXT = NEW_NODE
Step 9: EXIT



Inserting node at the last into a non-empty list

Algorithm for insertion after specified node:

In order to insert an element after the specified number of nodes into the linked list, we need to skip the desired number of elements in the list to move the pointer at the position after which the node will be inserted.

Allocate the space for the new node and add the item to the data part of it.

Now, we just need to make a few more link adjustments and our node at will be inserted at the specified position. Since, at the end of the loop, the loop pointer temp would be pointing to the node after which the new node will be inserted.

Therefore, the next part of the new node ptr must contain the address of the next part of the temp (since, ptr will be in between temp and the next of the temp).

Now, we just need to make the next part of the temp, point to the new node ptr. This will insert the new node ptr, at the specified position.

STEP 1: IF PTR = NULL

WRITE OVERFLOW

GOTO STEP 12

END OF IF

STEP 2: SET NEW_NODE = PTR

STEP 3: NEW_NODE → DATA = VAL

STEP 4: SET TEMP = HEAD

STEP 5: SET I = 0

STEP 6: REPEAT STEP 5 AND 6 UNTIL I<loc< li="">

STEP 7: TEMP = TEMP → NEXT

STEP 8: IF TEMP = NULL

WRITE "DESIRED NODE NOT PRESENT"

GOTO STEP 12

END OF IF

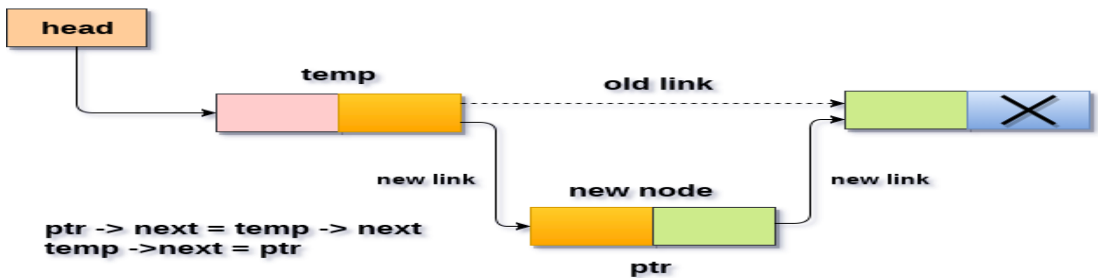
END OF LOOP

STEP 9: PTR → NEXT = TEMP → NEXT

STEP 10: TEMP → NEXT = PTR

STEP 11: SET PTR = NEW_NODE

STEP 12: EXIT




```
# A complete working Python program to demonstrate all
# insertion methods of linked list

# Node class
class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data  # Assign data
        self.next = None  # Initialize next as null


# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None


    # Function to insert a new node at the beginning
    def push(self, new_data):

        # 1 & 2: Allocate the Node &
        #        Put in the data
        new_node = Node(new_data)

        # 3. Make next of new Node as head
        new_node.next = self.head
```

```
    # 4. Move the head to point to new Node
    self.head = new_node


# This function is in LinkedList class. Inserts a
# new node after the given prev_node. This method is
# defined inside LinkedList class shown above */
def insertAfter(self, prev_node, new_data):

    # 1. check if the given prev_node exists
    if prev_node is None:
        print("The given previous node must inLinkedList.")
        return

    #  2. create new node &
    #      Put in the data
    new_node = Node(new_data)

    # 4. Make next of new Node as next of prev_node
    new_node.next = prev_node.next

    # 5. make next of prev_node as new_node
    prev_node.next = new_node


# This function is defined in Linked List class
# Appends a new node at the end.  This method is
# defined inside LinkedList class shown above */
def append(self, new_data):

    # 1. Create a new node
    # 2. Put in the data
    # 3. Set next as None
    new_node = Node(new_data)

    # 4. If the Linked List is empty, then make the
    #    new node as head
    if self.head is None:
        self.head = new_node
        return

    # 5. Else traverse till the last node
    last = self.head
    while (last.next):
        last = last.next
```

```
        # 6. Change the next of last node
        last.next =  new_node


    # Utility function to print the linked list
    def printList(self):
        temp = self.head
        while (temp):
            print(temp.data,end=" ")
            temp = temp.next



# Code execution starts here
if __name__=='__main__':

    # Start with the empty list
    llist = LinkedList()

    # Insert 6.  So linked list becomes 6->None
    llist.append(6)

    # Insert 7 at the beginning. So linked list becomes 7->6->None
    llist.push(7);

    # Insert 1 at the beginning. So linked list becomes 1->7->6->None
    llist.push(1);

    # Insert 4 at the end. So linked list becomes 1->7->6->4->None
    llist.append(4)

    # Insert 8, after 7. So linked list becomes 1 -> 7-> 8-> 6-> 4-> None
    llist.insertAfter(llist.head.next, 8)

    print('Created linked list is: ')
    llist.printList()
```

Output:

Created Linked list is: 1 7 8 6 4

Deletion in singly linked list at beginning:

Deleting a node from the beginning of the list is the simplest operation of all. It just need a few adjustments in the node pointers. Since the first node of the list is

to be deleted, therefore, we just need to make the head, point to the next of the head.

Now, free the pointer ptr which was pointing to the head node of the list.

Algorithm

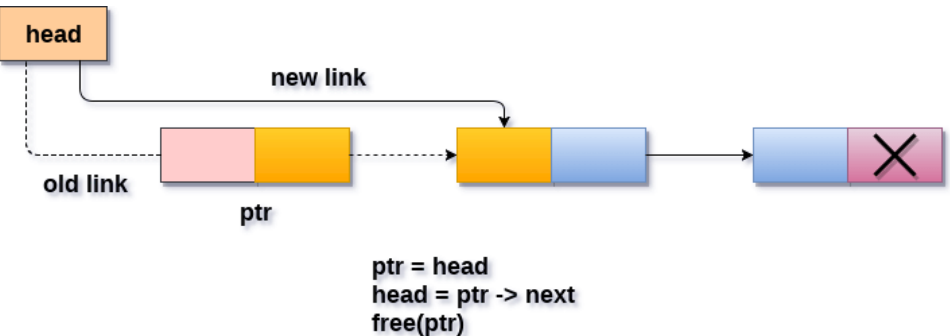
Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 5
[END OF IF]

Step 2: SET PTR = HEAD

Step 3: SET HEAD = HEAD -> NEXT

Step 4: FREE PTR

Step 5: EXIT



Deleting a node from the beginning

Deletion in singly linked list at the end:

There are two scenarios in which, a node is deleted from the end of the linked list.

1. There is only one node in the list and that needs to be deleted.
2. There are more than one node in the list and the last node of the list will be deleted.

In the first scenario,

The condition head → next = NULL will survive and therefore, the only node head of the list will be assigned to null.

In the second scenario,

The condition head → next = NULL would fail and therefore, we have to traverse the node in order to reach the last node of the list.

For this purpose, just declare a temporary pointer temp and assign it to head of the list. We also need to keep track of the second last node of the list. For this purpose, two pointers ptr and ptr1 will be used where ptr will point to the last node and ptr1 will point to the second last node of the list.

Now, we just need to make the pointer ptr1 point to the NULL and the last node of the list that is pointed by ptr will become free.

Algorithm

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 8
[END OF IF]

Step 2: SET PTR = HEAD

Step 3: Repeat Steps 4 and 5 while PTR -> NEXT!= NULL

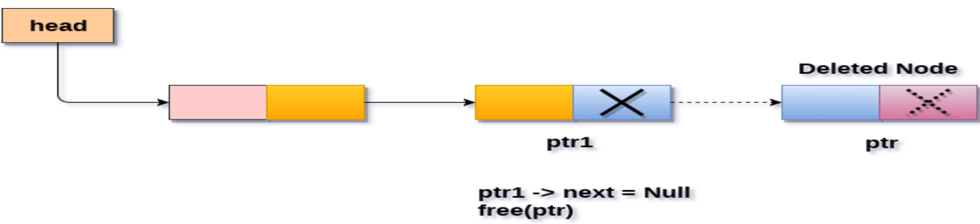
Step 4: SET PREPTR = PTR

Step 5: SET PTR = PTR -> NEXT
[END OF LOOP]

Step 6: SET PREPTR -> NEXT = NULL

Step 7: FREE PTR

Step 8: EXIT



Deleting a node from the last

<u>Deletion in singly linked list after the specified node:</u>

In order to delete the node, which is present after the specified node, we need to skip the desired number of nodes to reach the node after which the node will be deleted. We need to keep track of the two nodes.

The one which is to be deleted the other one if the node which is present before that node. For this purpose, two pointers are used: ptr and ptr1.

Now, our task is almost done, we just need to make a few pointer adjustments. Make the next of ptr1 (points to the specified node) point to the next of ptr (the node which is to be deleted).

Algorithm

STEP 1: IF HEAD = NULL

WRITE UNDERFLOW

GOTO STEP 10

END OF IF

STEP 2: SET TEMP = HEAD

STEP 3: SET I = 0

STEP 4: REPEAT STEP 5 TO 8 UNTIL I<loc< li="">

STEP 5: TEMP1 = TEMP

STEP 6: TEMP = TEMP → NEXT

STEP 7: IF TEMP = NULL

WRITE "DESIRED NODE NOT PRESENT"

GOTO STEP 12

END OF IF

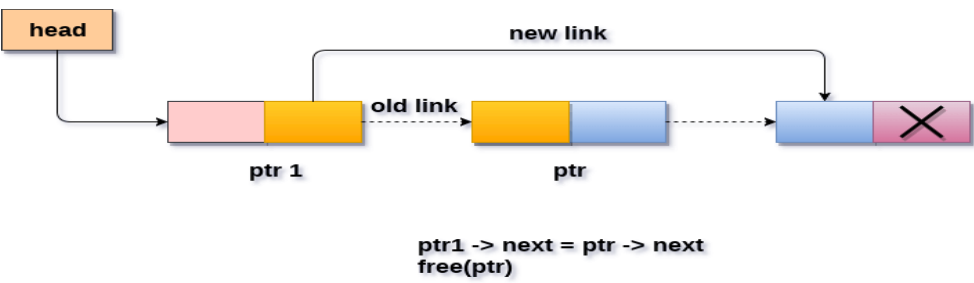
STEP 8: I = I+1

END OF LOOP

STEP 9: TEMP1 → NEXT = TEMP → NEXT

STEP 10: FREE TEMP

STEP 11: EXIT



Deletion a node from specified position

Traversing in singly linked list:

Traversing is the most common operation that is performed in almost every scenario of singly linked list. Traversing means visiting each node of the list once in order to perform some operation on that. This will be done by using the following statements.

Algorithm

STEP 1: SET PTR = HEAD

STEP 2: IF PTR = NULL

WRITE "EMPTY LIST"

GOTO STEP 7

END OF IF

STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR != NULL

STEP 5: PRINT PTR→ DATA

STEP 6: PTR = PTR → NEXT

[END OF LOOP]

STEP 7: EXIT

Searching in singly linked list:

Searching is performed in order to find the location of a particular element in the list. Searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element.

If the element is matched with any of the list element then the location of the element is returned from the function.

Algorithm

Step 1: SET PTR = HEAD

Step 2: Set I = 0

STEP 3: IF PTR = NULL

WRITE "EMPTY LIST"

GOTO STEP 8

END OF IF

STEP 4: REPEAT STEP 5 TO 7 UNTIL PTR != NULL

STEP 5: if ptr → data = item

write i+1

End of IF

STEP 6: I = I + 1

STEP 7: PTR = PTR → NEXT

[END OF LOOP]

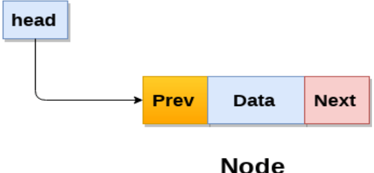
STEP 8: EXIT

Doubly linked list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence.

Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer).

A sample node in a doubly linked list is shown in the figure.



Node

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.

Doubly Linked List

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

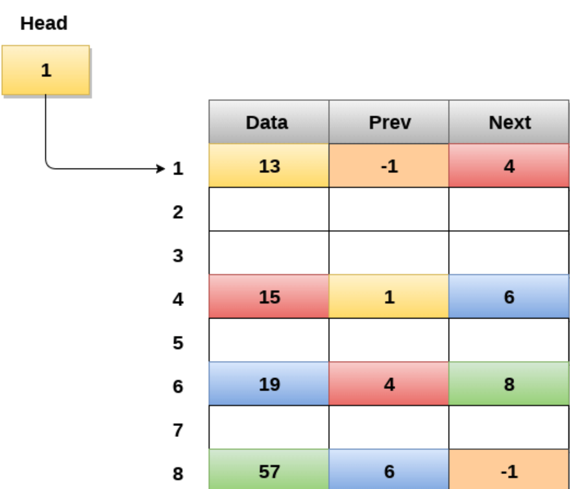
In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes.

However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list:

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion.

However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).



Memory Representation of a Doubly linked list

Operations on doubly linked list:

Node representation of a doubly linked list

```
# Node of a doubly linked list
class Node:
    def __init__(self, next=None, prev=None, data=None):
        self.next = next # reference to next node in DLL
        self.prev = prev # reference to previous node in DLL
        self.data = data
```

All the remaining operations regarding doubly linked list are described in the following table.

SN	Operation	Description

1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Insertion in doubly linked list at beginning:

As in doubly linked list, each node of the list contain double pointers therefore we have to maintain more number of pointers in doubly linked list as compare to singly linked list.

There are two scenarios of inserting any element into doubly linked list. Either the list is empty or it contains at least one element. Perform the following steps to insert a node in doubly linked list at beginning.

- Allocate the space for the new node in the memory.
- Check whether the list is empty or not. The list is empty if the condition head == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

- In the second scenario, the condition **head == NULL** become false and the node will be inserted in beginning. The next pointer of the node will point to the existing head pointer of the node. The prev pointer of the existing head will point to the new node being inserted.

Since, the node being inserted is the first node of the list and therefore it must contain NULL in its prev pointer. Hence assign null to its previous part and make the head point to this node.

Algorithm :

Step 1: IF ptr = NULL

Write OVERFLOW

Go to Step 8

[END OF IF]

Step 2: SET NEW_NODE = ptr

Step 3: SET NEW_NODE -> DATA = VAL

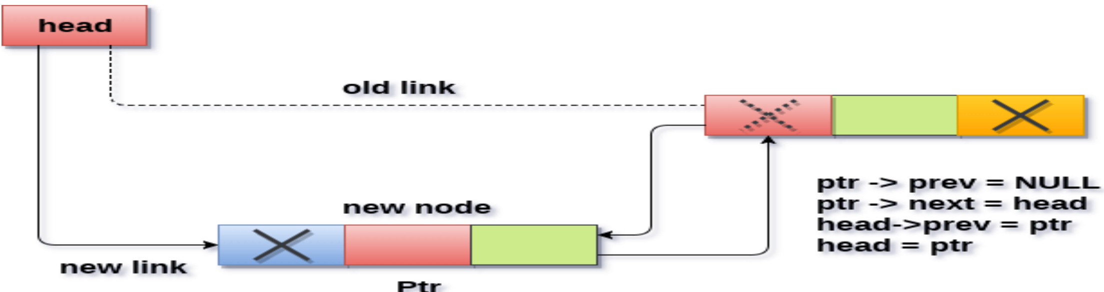
Step 4: SET NEW_NODE -> PREV = NULL

Step 5: SET NEW_NODE -> NEXT = START

Step 6: SET head -> PREV = NEW_NODE

Step 7: SET head = NEW_NODE

Step 8: EXIT



Insertion into doubly linked list at beginning

Insertion in doubly linked list at the end:

In order to insert a node in doubly linked list at the end, we must make sure whether the list is empty or it contains any element. Use the following steps in order to insert the node in doubly linked list at the end.

- Allocate the memory for the new node. Make the pointer **ptr** point to the new node being inserted.
- Check whether the list is empty or not. The list is empty if the condition **head == NULL** holds. In that case, the node will be inserted as the only node of the

list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

- In the second scenario, the condition head == NULL become false. The new node will be inserted as the last node of the list. For this purpose, we have to traverse the whole list in order to reach the last node of the list. Initialize the pointer **temp** to head and traverse the list by using this pointer.

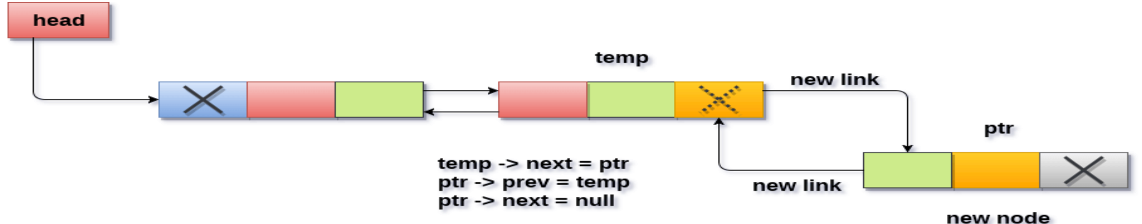
The pointer temp point to the last node at the end of this while loop. Now, we just need to make a few pointer adjustments to insert the new node ptr to the list. First, make the next pointer of temp point to the new node being inserted i.e. ptr.

make the previous pointer of the node ptr point to the existing last node of the list i.e. temp.

make the next pointer of the node ptr point to the null as it will be the new last node of the list.

Algorithm:

Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 10
[END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET NEW_NODE -> DATA = VAL
Step 4: SET NEW_NODE -> NEXT = NULL
Step 5: SET TEMP = HEAD
Step 6: Repeat Step 7 while TEMP -> NEXT != NULL
Step 7: SET TEMP = TEMP -> NEXT
[END OF LOOP]
Step 8: SET TEMP -> NEXT = NEW_NODE
Step 9: SET NEW_NODE -> PREV = TEMP
Step 10: EXIT



Insertion into doubly linked list at the end

Insertion in doubly linked list after Specified node:

In order to insert a node after the specified node in the list, we need to skip the required number of nodes in order to reach the mentioned node and then make the pointer adjustments as required.

Use the following steps for this purpose.

- Allocate the memory for the new node.
- Traverse the list by using the pointer **temp** to skip the required number of nodes in order to reach the specified node.
- The temp would point to the specified node at the end of the **for** loop. The new node needs to be inserted after this node therefore we need to make a few pointer adjustments here. Make the next pointer of **ptr** point to the next node of temp.

make the **next** pointer of temp point to the new node ptr.

Algorithm:

Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 14
[END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET NEW_NODE -> DATA = VAL
Step 4: SET TEMP = head
Step 5: SET I = 0
Step 6: REPEAT 7 to 9 until I
Step 7: SET TEMP = TEMP -> NEXT

STEP 8: IF TEMP = NULL

STEP 9: WRITE "LESS THAN DESIRED NO. OF ELEMENTS"

GOTO STEP 14

[END OF IF]

[END OF LOOP]

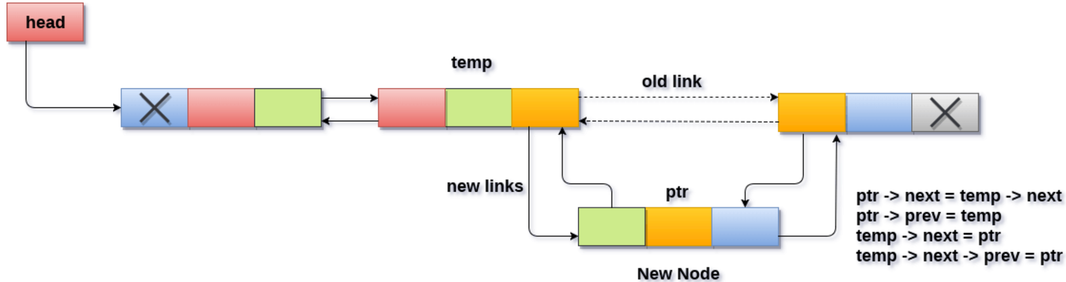
Step 10: SET NEW_NODE -> NEXT = TEMP -> NEXT

Step 11: SET NEW_NODE -> PREV = TEMP

Step 12 : SET TEMP -> NEXT = NEW_NODE

Step 13: SET TEMP -> NEXT -> PREV = NEW_NODE

Step 14: EXIT



Insertion into doubly linked list after specified node

Deletion at beginning:

Deletion in doubly linked list at the beginning is the simplest operation. We just need to copy the head pointer to pointer ptr and shift the head pointer to its next.

Now make the prev of this new head node point to NULL. This will be done by using the following statements.

Now free the pointer ptr by using the **free** function.

Algorithm

STEP 1: IF HEAD = NULL

WRITE UNDERFLOW

GOTO STEP 6

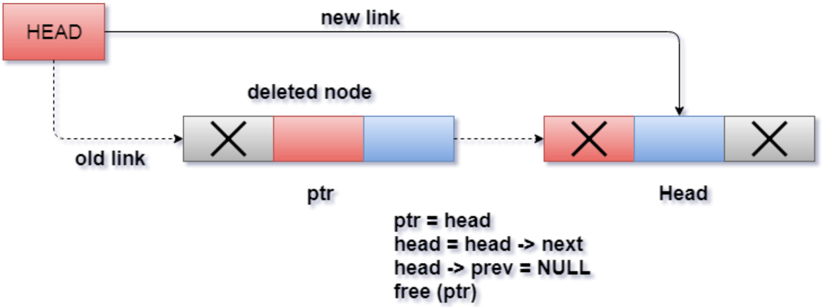
STEP 2: SET PTR = HEAD

STEP 3: SET HEAD = PTR→ NEXT

STEP 4: SET HEAD → PREV = NULL

STEP 5: FREE PTR

STEP 6: EXIT



Deletion in doubly linked list from beginning

Deletion in doubly linked list at the end:

Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position.

In order to delete the last node of the list, we need to follow the following steps.

- If the list is already empty then the condition head == NULL will become true and therefore the operation can not be carried on.
- If there is only one node in the list then the condition head → next == NULL become true. In this case, we just need to assign the head of the list to NULL and free head in order to completely delete the list.
- Otherwise, just traverse the list to reach the last node of the list. This will be done by using the following statements.

- The ptr would point to the last node of the ist at the end of the for loop. Just make the next pointer of the previous node of **ptr** to **NULL**.

free the pointer as this the node which is to be deleted.

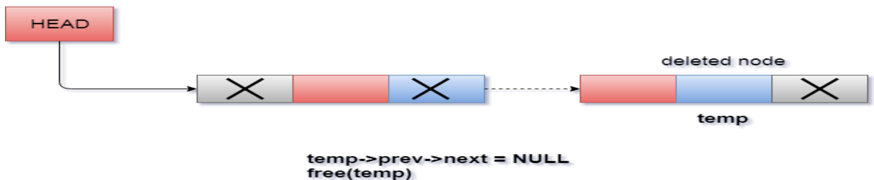
Algorithm

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 7
[END OF IF]
Step 2: SET TEMP = HEAD
Step 3: REPEAT STEP 4 WHILE TEMP->NEXT != NULL
Step 4: SET TEMP = TEMP->NEXT
[END OF LOOP]

Step 5: SET TEMP ->PREV-> NEXT = NULL

Step 6: FREE TEMP

Step 7: EXIT



Deletion in doubly linked list at the end

Deletion in doubly linked list after the specified node:

In order to delete the node after the specified data, we need to perform the following steps.

- Copy the head pointer into a temporary pointer temp.
- Traverse the list until we find the desired data value.
- Check if this is the last node of the list. If it is so then we can't perform deletion.
- Check if the node which is to be deleted, is the last node of the list, if it so then we have to make the next pointer of this node point to null so that it can be the new last node of the list.
- Otherwise, make the pointer ptr point to the node which is to be deleted. Make the next of temp point to the next of ptr. Make the previous of next node of ptr point to temp. free the ptr.

Algorithm:

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 9
[END OF IF]

Step 2: SET TEMP = HEAD

Step 3: Repeat Step 4 while TEMP -> DATA != ITEM

Step 4: SET TEMP = TEMP -> NEXT
[END OF LOOP]

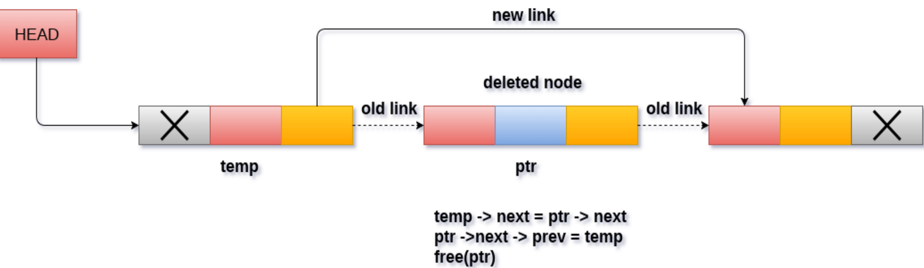
Step 5: SET PTR = TEMP -> NEXT

Step 6: SET TEMP -> NEXT = PTR -> NEXT

Step 7: SET PTR -> NEXT -> PREV = TEMP

Step 8: FREE PTR

Step 9: EXIT



Deletion of a specified node in doubly linked list

Searching for a specific node in Doubly Linked List:

We just need traverse the list in order to search for a specific element in the list. Perform following operations in order to search a specific operation.

- Copy head pointer into a temporary pointer variable ptr.
- declare a local variable I and assign it to 0.
- Traverse the list until the pointer ptr becomes null. Keep shifting pointer to its next and increasing i by +1.
- Compare each element of the list with the item which is to be searched.
- If the item matched with any node value then the location of that value I will be returned from the function else NULL is returned.

Algorithm:

Step 1: IF HEAD == NULL
WRITE "UNDERFLOW"
GOTO STEP 8
[END OF IF]

Step 2: Set PTR = HEAD

Step 3: Set i = 0

Step 4: Repeat step 5 to 7 while PTR != NULL

Step 5: IF PTR → data = item
return i
[END OF IF]

Step 6: i = i + 1

Step 7: PTR = PTR → next

Step 8: Exit

Traversing in doubly linked list:

Traversing is the most common operation in case of each data structure. For this purpose, copy the head pointer in any of the temporary pointer ptr.

Then, traverse through the list by using while loop. Keep shifting value of pointer variable **ptr** until we find the last node. The last node contains **null** in its next part.

Although, traversing means visiting each node of the list once to perform some specific operation. Here, we are printing the data associated with each node of the list.

Algorithm:

Step 1: IF HEAD == NULL
  WRITE "UNDERFLOW"
 GOTO STEP 6
 [END OF IF]

Step 2: Set PTR = HEAD

Step 3: Repeat step 4 and 5 while PTR != NULL

Step 4: Write PTR → data

Step 5: PTR = PTR → next

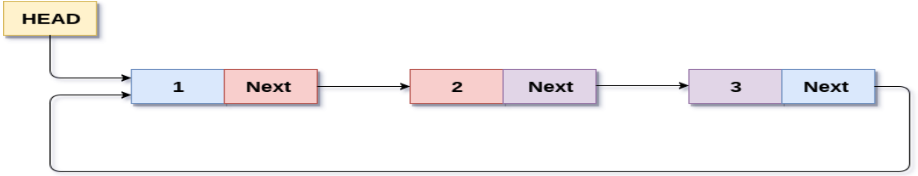
Step 6: Exit

Circular Singly Linked List:

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



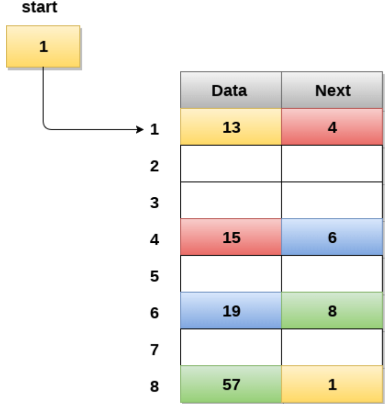
Circular Singly Linked List

Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Memory Representation of circular linked list:

In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects.

However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.



Memory Representation of a circular linked list

Operations on Circular Singly linked list:

Insertion

SN	Operation	Description
1	Insertion at beginning	Adding a node into circular singly linked list at the beginning.
2	Insertion at the end	Adding a node into circular singly linked list at the end.

Deletion & Traversing

SN	Operation	Description
1	Deletion at beginning	Removing the node from circular singly linked list at the beginning.
2	Deletion at the end	Removing the node from circular singly linked list at the end.
3	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4	Traversing	Visiting each element of the list at least once in order to perform some specific operation.

Insertion into circular singly linked list at beginning

There are two scenarios in which a node can be inserted in circular singly linked list at beginning. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

In the first scenario, the condition **head == NULL** will be true. Since, the list in which, we are inserting the node is a circular singly linked list, therefore the only node of the list (which is just inserted into the list) will point to itself only. We also need to make the head pointer point to this node.

In the second scenario, the condition **head == NULL** will become false which means that the list contains at least one node. In this case, we need to traverse the list in order to reach the last node of the list.

At the end of the loop, the pointer temp would point to the last node of the list. Since, in a circular singly linked list, the last node of the list contains a pointer to the first node of the list.

Therefore, we need to make the next pointer of the last node point to the head node of the list and the new node which is being inserted into the list will be

the new head node of the list therefore the next pointer of temp will point to the new node ptr.

The next pointer of temp will point to the existing head node of the list.

Now, make the new node ptr, the new head node of the circular singly linked list.

in this way, the node ptr has been inserted into the circular singly linked list at beginning.

Algorithm

Step 1: IF PTR = NULL

Write OVERFLOW

Go to Step 10

[END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET NEW_NODE -> DATA = VAL

Step 4: SET TEMP = HEAD

Step 5: Repeat Step 8 while TEMP -> NEXT != HEAD

Step 6: SET TEMP = TEMP -> NEXT

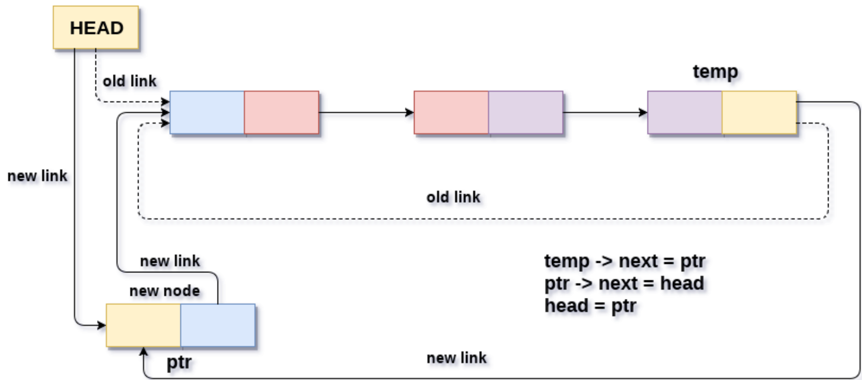
[END OF LOOP]

Step 7: SET NEW_NODE -> NEXT = HEAD

Step 8: SET TEMP → NEXT = NEW_NODE

Step 9: SET HEAD = NEW_NODE

Step 10: EXIT



Insertion into circular singly linked list at beginning

Insertion into circular singly linked list at the end

There are two scenarios in which a node can be inserted in circular singly linked list at beginning. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

In the first scenario, the condition **head == NULL** will be true. Since, the list in which, we are inserting the node is a circular singly linked list, therefore the only node of the list (which is just inserted into the list) will point to itself only.

We also need to make the head pointer point to this node. In the second scenario, the condition **head == NULL** will become false which means that the list contains at least one node.

In this case, we need to traverse the list in order to reach the last node of the list. This will be done by using the following statement.

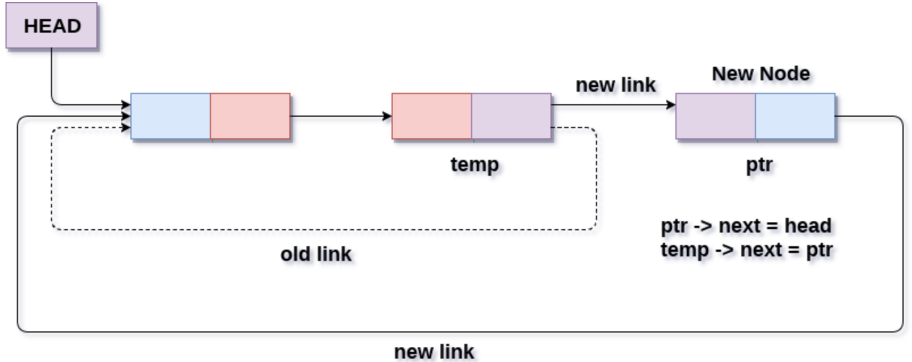
At the end of the loop, the pointer temp would point to the last node of the list. Since, the new node which is being inserted into the list will be the new last node of the list. Therefore the existing last node i.e. **temp** must point to the new node **ptr**.

The new last node of the list i.e. ptr will point to the head node of the list.

In this way, a new node will be inserted in a circular singly linked list at the beginning.

Algorithm

Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 9
[END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET NEW_NODE -> DATA = VAL
Step 4: SET NEW_NODE -> NEXT = HEAD
Step 5: SET TEMP = HEAD
Step 6: Repeat Step 7 while TEMP -> NEXT != HEAD
Step 7: SET TEMP = TEMP -> NEXT
[END OF LOOP]
Step 8: SET TEMP -> NEXT = NEW_NODE
Step 9: EXIT



Insertion into circular singly linked list at end

Deletion in circular singly linked list at beginning:

In order to delete a node in circular singly linked list, we need to make a few pointer adjustments.

There are three scenarios of deleting a node from circular singly linked list at beginning.

Scenario 1: (The list is Empty)

If the list is empty then the condition **head == NULL** will become true, in this case, we just need to print **underflow** on the screen and make exit.

Scenario 2: (The list contains single node)

If the list contains single node then, the condition **head → next == head** will become true. In this case, we need to delete the entire list and make the head pointer free.

Scenario 3: (The list contains more than one node)

If the list contains more than one node then, in that case, we need to traverse the list by using the pointer **ptr** to reach the last node of the list.

At the end of the loop, the pointer ptr point to the last node of the list. Since, the last node of the list points to the head node of the list. Therefore this will be changed as now, the last node of the list will point to the next of the head node.

Now, free the head pointer by using the free() method in C language.

Make the node pointed by the next of the last node, the new head of the list.

In this way, the node will be deleted from the circular singly linked list from the beginning.

Algorithm

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 8
[END OF IF]

Step 2: SET PTR = HEAD

Step 3: Repeat Step 4 while PTR → NEXT != HEAD

Step 4: SET PTR = PTR → next

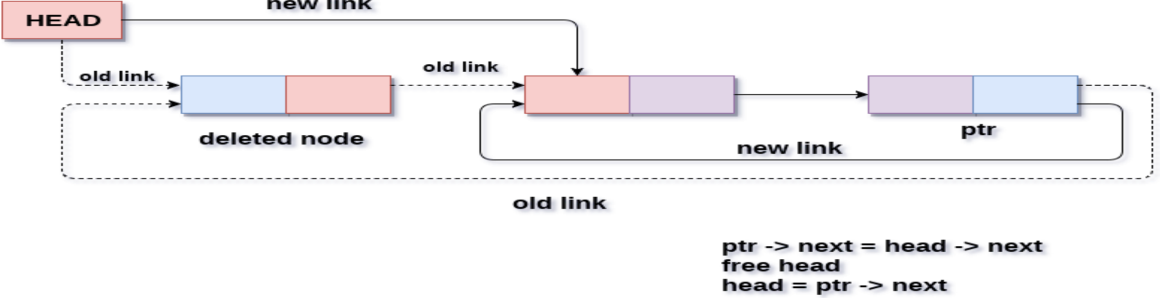
[END OF LOOP]

Step 5: SET PTR → NEXT = HEAD → NEXT

Step 6: FREE HEAD

Step 7: SET HEAD = PTR → NEXT

Step 8: EXIT



Deletion in circular singly linked list at beginning

Deletion in Circular singly linked list at the end

There are three scenarios of deleting a node in circular singly linked list at the end.

Scenario 1 (the list is empty)

If the list is empty then the condition **head == NULL** will become true, in this case, we just need to print **underflow** on the screen and make exit.

Scenario 2(the list contains single element)

If the list contains single node then, the condition **head → next == head** will become true. In this case, we need to delete the entire list and make the head pointer free.
if(head->next == head)

Scenario 3(the list contains more than one element)

If the list contains more than one element, then in order to delete the last element, we need to reach the last node. We also need to keep track of the second last node of the list. For this purpose, the two pointers ptr and preptr are defined.

Now, we need to make just one more pointer adjustment. We need to make the next pointer of preptr point to the next of ptr (i.e. head) and then make pointer ptr free.

Algorithm

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 8
[END OF IF]

Step 2: SET PTR = HEAD

Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != HEAD

Step 4: SET PREPTR = PTR

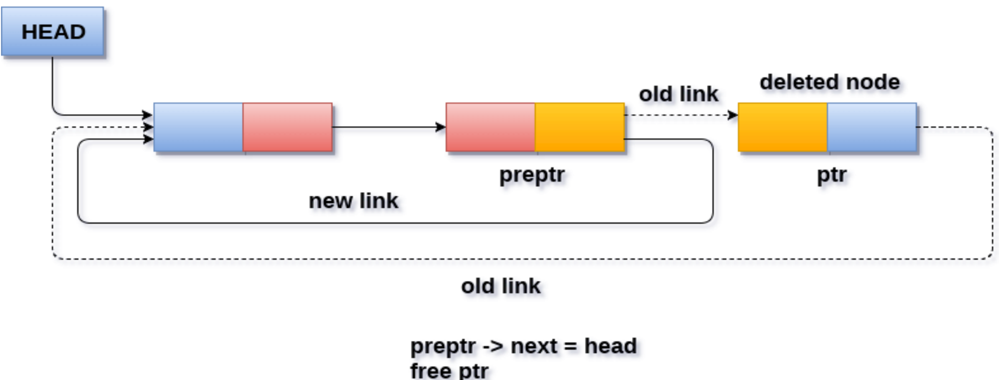
Step 5: SET PTR = PTR -> NEXT

[END OF LOOP]

Step 6: SET PREPTR -> NEXT = HEAD

Step 7: FREE PTR

Step 8: EXIT



Deletion in circular singly linked list at end

<u>Searching in circular singly linked list:</u>

Searching in circular singly linked list needs traversing across the list. The item which is to be searched in the list is matched with each node data of the list once and if the match found then the location of that item is returned otherwise -1 is returned.

Algorithm

Step 1: SET PTR = HEAD

Step 2: Set I = 0

STEP 3: IF PTR = NULL

WRITE "EMPTY LIST"

GOTO STEP 9

END OF IF

STEP 4: IF HEAD → DATA = ITEM

WRITE i+1 RETURN [END OF IF]

STEP 5: REPEAT STEP 5 TO 7 UNTIL PTR->next != head

STEP 6: if ptr → data = item

write i+1

RETURN

End of IF

STEP 7: I = I + 1

STEP 8: PTR = PTR → NEXT

[END OF LOOP]

STEP 9: EXIT

Traversing in Circular Singly linked list:

Traversing in circular singly linked list can be done through a loop. Initialize the temporary pointer variable **temp** to head pointer and run the while loop until the next pointer of temp becomes **head**. The algorithm and the c function implementing the algorithm is described as follows.

Algorithm

STEP 1: SET PTR = HEAD

STEP 2: IF PTR = NULL

WRITE "EMPTY LIST"

GOTO STEP 8

END OF IF

STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR → NEXT != HEAD

STEP 5: PRINT PTR → DATA

STEP 6: PTR = PTR → NEXT

[END OF LOOP]

STEP 7: PRINT PTR→ DATA

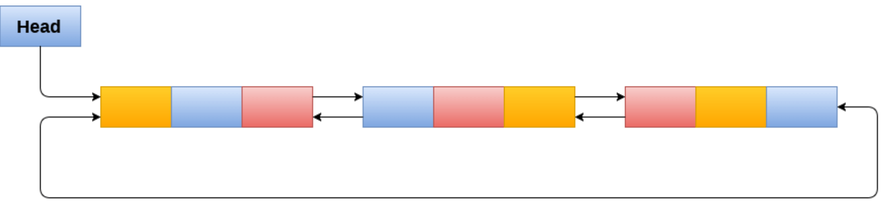
STEP 8: EXIT

CIRCULAR DOUBLY LINKED LIST:

Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node.

Circular doubly linked list doesn't contain NULL in any of the node. The last node of the list contains the address of the first node of the list. The first node of the list also contains address of the last node in its previous pointer.

A circular doubly linked list is shown in the following figure.



Circular Doubly Linked List

Due to the fact that a circular doubly linked list contains three parts in its structure therefore, it demands more space per node and more expensive basic operations. However, a circular doubly linked list provides easy manipulation of the pointers and the searching becomes twice as efficient.

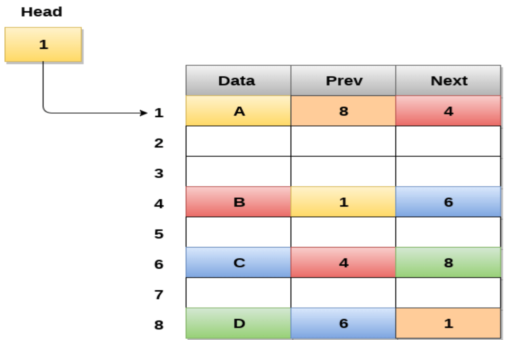
Memory Management of Circular Doubly linked list:

The following figure shows the way in which the memory is allocated for a circular doubly linked list. The variable head contains the address of the first element of the list i.e. 1 hence the starting node of the list contains data A is stored at address 1.

Since, each node of the list is supposed to have three parts therefore; the starting node of the list contains address of the last node i.e. 8 and the next node i.e. 4.

The last node of the list that is stored at address 8 and containing data as 6, contains address of the first node of the list as shown in the image.

In circular doubly linked list, the last node is identified by the address of the first node which is stored in the next part of the last node therefore the node which contains the address of the first node, is actually the last node of the list.



Memory Representation of a Circular Doubly linked list

Operations on circular doubly linked list:

There are various operations which can be performed on circular doubly linked list. The node structure of a circular doubly linked list is similar to doubly linked list. However, the operations on circular doubly linked list are described in the following table.

SN	Operation	Description
1	Insertion at beginning	Adding a node in circular doubly linked list at the beginning.
2	Insertion at end	Adding a node in circular doubly linked list at the end.
3	Deletion at beginning	Removing a node in circular doubly linked list from beginning.
4	Deletion at end	Removing a node in circular doubly linked list at the end.

Traversing and searching in circular doubly linked list is similar to that in the circular singly linked list.

Insertion in circular doubly linked list at beginning

There are two scenario of inserting a node in circular doubly linked list at beginning. Either the list is empty or it contains more than one element in the list.

Allocate the memory space for the new node **ptr** by using the following statement.

In the first case, the condition **head == NULL** becomes true therefore; the node will be added as the first node in the list. The next and the previous pointer of this newly added node will point to itself only.

In the second scenario, the condition **head == NULL** becomes false. In this case, we need to make a few pointer adjustments at the end of the list. For this purpose, we need to reach the last node of the list through traversing the list.

At the end of loop, the pointer temp would point to the last node of the list. Since the node which is to be inserted will be the first node of the list therefore, temp must contain the address of the new node ptr into its next part.

In this way, the new node is inserted into the list at the beginning.

Algorithm:

Step 1: IF PTR = NULL

Write OVERFLOW

Go to Step 12

[END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET NEW_NODE -> DATA = VAL

Step 4: SET TEMP = HEAD

Step 5: Repeat Step 6 while TEMP -> NEXT != HEAD

Step 6: SET TEMP = TEMP -> NEXT

[END OF LOOP]

Step 7: SET TEMP -> NEXT = NEW_NODE

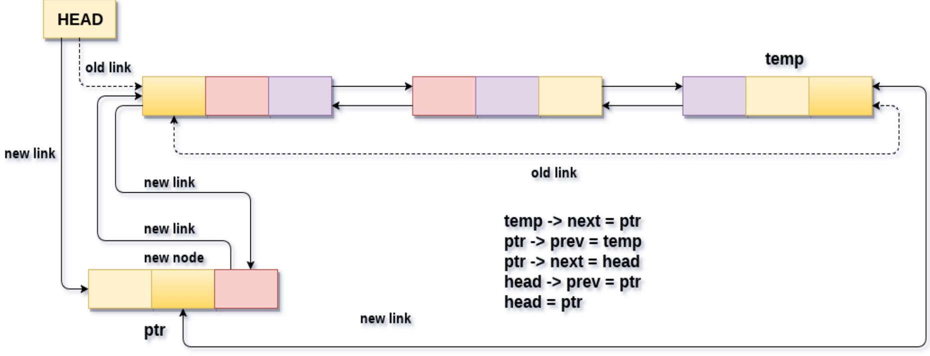
Step 8: SET NEW_NODE -> PREV = TEMP

Step 9 : SET NEW_NODE -> NEXT = HEAD

Step 10: SET HEAD -> PREV = NEW_NODE

Step 11: SET HEAD = NEW_NODE

Step 12: EXIT



Insertion into circular doubly linked list at beginning

Insertion in circular doubly linked list at end:

There are two scenario of inserting a node in circular doubly linked list at the end. Either the list is empty or it contains more than one element in the list.

Allocate the memory space for the new node **ptr**

In the first case, the condition **head == NULL** becomes true therefore, the node will be added as the first node in the list. The next and the previous pointer of this newly added node will point to itself only.

In the second scenario, the condition **head == NULL** become false, therefore node will be added as the last node in the list.

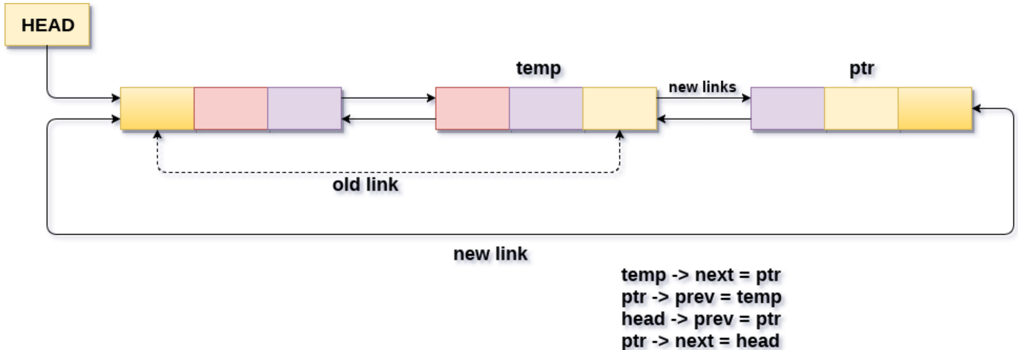
For this purpose, we need to make a few pointer adjustments in the list at the end. Since, the new node will contain the address of the first node of the list therefore we need to make the next pointer of the last node point to the head node of the list.

Similarly, the previous pointer of the head node will also point to the new last node of the list.

Now, we also need to make the next pointer of the existing last node of the list (temp) point to the new last node of the list, similarly, the new last node will also contain the previous pointer to the temp.

Algorithm

Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 11
[END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET NEW_NODE -> DATA = VAL
Step 4: SET NEW_NODE -> NEXT = HEAD
Step 5: SET TEMP = HEAD
Step 6: Repeat Step 7 while TEMP -> NEXT != HEAD
Step 7: SET TEMP = TEMP -> NEXT
[END OF LOOP]
Step 8: SET TEMP -> NEXT = NEW_NODE
Step 9: SET NEW_NODE -> PREV = TEMP
Step 10: SET HEAD -> PREV = NEW_NODE
Step 11: EXIT



Insertion into circular doubly linked list at end

Deletion in Circular doubly linked list at beginning:

There can be two scenario of deleting the first node in a circular doubly linked list.

The node which is to be deleted can be the only node present in the linked list. In this case, the condition head → next == head will become true, therefore the list needs to be completely deleted.

It can be simply done by assigning head pointer of the list to null and free the head pointer.

In the second scenario, the list contains more than one element in the list, therefore the condition **head → next == head** will become false. Now, reach the last node of the list and make a few pointer adjustments there. Run a while loop for this purpose

Now, temp will point to the last node of the list. The first node of the list i.e. pointed by head pointer, will need to be deleted.

Therefore the last node must contain the address of the node that is pointed by the next pointer of the existing head node.

The new head node i.e. next of existing head node must also point to the last node of the list through its previous pointer.

Now, free the head pointer and the make its next pointer, the new head node of the list.

Algorithm

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 9

[END OF IF]

Step 2: SET TEMP = HEAD

Step 3: Repeat Step 4 while TEMP -> NEXT != HEAD

Step 4: SET TEMP = TEMP -> NEXT

[END OF LOOP]

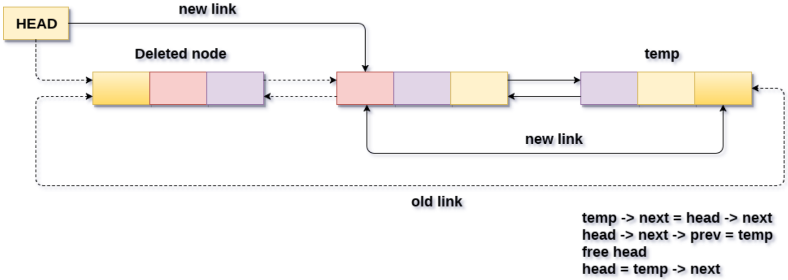
Step 5: SET TEMP -> NEXT = HEAD -> NEXT

Step 6: SET HEAD -> NEXT -> PREV = TEMP

Step 7: FREE HEAD

Step 8: SET HEAD = TEMP -> NEXT

Step 9: Exit



Deletion in circular doubly linked list at beginning

Deletion in circular doubly linked list at end:

There can be two scenario of deleting the first node in a circular doubly linked list.

The node which is to be deleted can be the only node present in the linked list. In this case, the condition **head → next == head** will become true, therefore the list needs to be completely deleted.

In the second scenario, the list contains more than one element in the list, therefore the condition **head → next == head** will become false. Now, reach the last node of the list and make a few pointer adjustments there.

Now, temp will point to the node which is to be deleted from the list. Make the next pointer of previous node of temp, point to the head node of the list.

Make the previous pointer of the head node, point to the previous node of temp.

Now, free the temp pointer to free the memory taken by the node.

Algorithm

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 8

[END OF IF]

Step 2: SET TEMP = HEAD

Step 3: Repeat Step 4 while TEMP -> NEXT != HEAD

Step 4: SET TEMP = TEMP -> NEXT

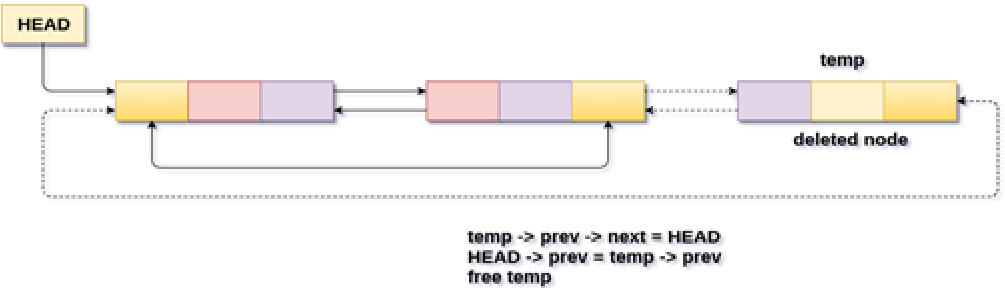
[END OF LOOP]

Step 5: SET TEMP -> PREV -> NEXT = HEAD

Step 6: SET HEAD -> PREV = TEMP -> PREV

Step 7: FREE TEMP

Step 8: EXIT



LINKED LIST IMPLEMENTATION OF STACK:

The major problem with the stack implemented using an array is, it works only for a fixed number of data values.

Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use.

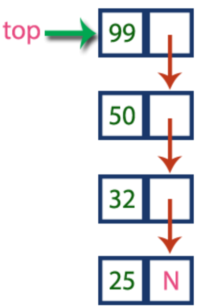
A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values.

The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'.

Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.

Example



Stack Operations using Linked List:

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether stack is **Empty** (**top** == **NULL**)
- **Step 3 -** If it is **Empty**, then set **newNode → next** = **NULL**.
- **Step 4 -** If it is **Not Empty**, then set **newNode → next** = **top**.
- **Step 5 -** Finally, set **top** = **newNode**.

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

- **Step 1 -** Check whether **stack** is **Empty** (**top == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
- **Step 3 -** If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4 -** Then set '**top = top → next**'.
- **Step 5 -** Finally, delete '**temp**'. (**free(temp)**).

display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

- **Step 1 -** Check whether stack is **Empty** (**top** == **NULL**).
- **Step 2 -** If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.
- **Step 4 -** Display '**temp → data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp → next** != **NULL**).
- **Step 5 -** Finally! Display '**temp → data** ---> **NULL**'.

#Python code to implement stack using linked list

```
class Node:

   # Class to create nodes of linked list
   # constructor initializes node automatically
   def __init__(self, data):
      self.data = data
      self.next = None


class Stack:

   # head is default NULL
   def __init__(self):
      self.head = None

   # Checks if stack is empty
   def isempty(self):
      if self.head == None:
         return True
      else:
         return False

   # Method to add data to the stack
   # adds to the start of the stack
   def push(self, data):

      if self.head == None:
         self.head = Node(data)

      else:
         newnode = Node(data)
         newnode.next = self.head
         self.head = newnode

   # Remove element that is the current head (start of the stack)
   def pop(self):

      if self.isempty():
         return None

      else:
         # Removes the head node and makes
         # the preceding one the new head
         poppednode = self.head
         self.head = self.head.next
```

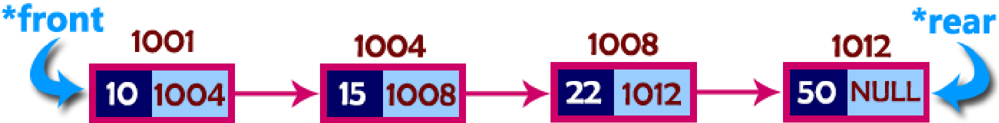
```
            poppednode.next = None
            return poppednode.data

    # Returns the head node data
    def peek(self):

        if self.isempty():
            return None

        else:
            return self.head.data

    # Prints out the stack
    def display(self):

        iternode = self.head
        if self.isempty():
            print("Stack Underflow")

        else:

            while (iternode != None):
                print(iternode.data, "->", end=" ")
                iternode = iternode.next
            return
if __name__ == "__main__":
    s = Stack()

    while(True):
        el = int(input("1 for Push\n2 for Pop\n3 to check if it is Empty\n4 to print
Stack\n5 to peek\n6 to exit\n"))
        if(el == 1):
            item = input("Enter Element to push in stack\n")
            s.push(item)
        if(el == 2):
            print(s.pop())
        if(el == 3):
            print(s.isEmpty())
        if(el == 4):
            s.printStack()
        if(el == 5):
            print(s.peek())
        if(el == 6):
            break
```

Queue implementation Using Linked List:

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. The queue which is implemented using a linked list can work for an unlimited number of values

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

Example



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

Operations:

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

- **Step 1 -** Create a **newNode** with given value and set '**newNode → next**' to **NULL**.
- **Step 2 -** Check whether queue is **Empty** (**rear** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **front** = **newNode** and **rear** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, set **rear →**

  next = **newNode** and **rear** = **newNode**.

deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

- **Step 1 -** Check whether **queue** is **Empty** (**front == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function
- **Step 3 -** If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.
- **Step 4 -** Then set '**front** = **front → next**' and delete '**temp**' (**free(temp)**).

display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

- **Step 1 -** Check whether queue is **Empty** (**front** == **NULL**).
- **Step 2 -** If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.
- **Step 4 -** Display '**temp → data** --->' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp → next** != **NULL**).
- **Step 5 -** Finally! Display '**temp → data** ---> **NULL**'.

#python code to implement queue using linked list

```
class Node:
  def __init__(self,data):
    self.data = data
    self.next = None
class Queue:
  def __init__(self):
    self.head = None
  def display(self):
    temp = self.head
    print('Queue is:[', end='')
    while temp:
      print(temp.data, end=", ")
      temp = temp.next
    print(']')
  def enqueue(self, data):
    if self.head is None:
      self.head = Node(data)
    else:
```

```
            temp = self.head
            while temp.next is not None:
                temp = temp.next
            temp.next = Node(data)
        #self.display()
    def dequeue(self):
        if self.head is None:
            print('Queue is Empty')
            #self.display()
            #return None
        else:
            data=self.head.data
            self.head=self.head.next
            #self.display()
            return data
if __name__ == '__main__':
    Q = Queue()
    print('1.enqueue\n2.dequeue\n3.display\n4.exit')
    while 1:
        choice = input('Enter your option:')
        if choice is '1':
            d = int(input('Enter data:'))
            Q.enqueue(d)
        elif choice is '2':
            print('poped element is:',Q.dequeue())
        elif choice is '3':
            print(Q.display())
        else:
            break
```

<u>**UNIT-IV**</u>

TREES:

Definition of trees:

A tree is defined as a finite set of one or more nodes such that

(i) there is a specially designated node called the root and

(ii) the rest of the nodes could be partitioned into t disjoint sets ($t \geq 0$) each set representing a tree Ti , i = 1, 2, . . . t known as subtree of the tree.

A node in the definition of the tree represents an item of information, and the links between the nodes termed as branches, represent an association between the items of information.

Figure below illustrates a tree. The definition of the tree emphasizes on the aspect of (i) connectedness and (ii) absence of closed loops or what are termed cycles.

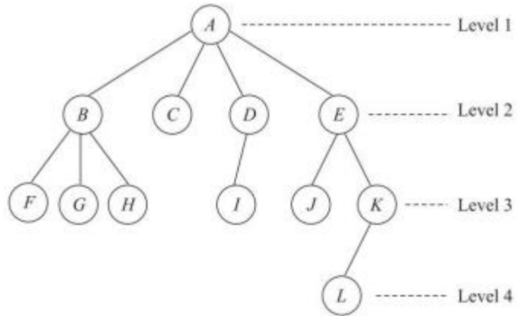


Fig. An example tree

Basic terminologies of trees :

There are several basic terminologies associated with the tree. The specially designated node called **root node**.

The number of subtrees of a node is known as the **degree** of the node. Nodes that have zero degree are called **leaf nodes** or **terminal nodes**.

The rest of them are called as **non-terminal** nodes. These nodes which hang from branches emanating from a node are known a **children** and the node from which the branches emanate is known as the **parent node**.

Children of the same parent node are referred to as **siblings**. The **ancestors** of a given node are those nodes that occur on the path from the root to the given node.

The **degree of a tree** is the maximum degree of the node in the tree. The **level** of a node is defined by letting the root to occupy level 1 . The rest of the nodes occupy various levels depending on their association.

Thus if a parent node occupies level i, its children should occupy level i+1. This renders the tree to have a hierarchical structure with root occupying the top most level of 1.

The **height or depth** of a tree is defined to be the maximum level of any node in the tree.

Depth of a node to be the length of the longest path from the root node to that node, which yields the relation, **depth of the tree = height of the tree – 1** .

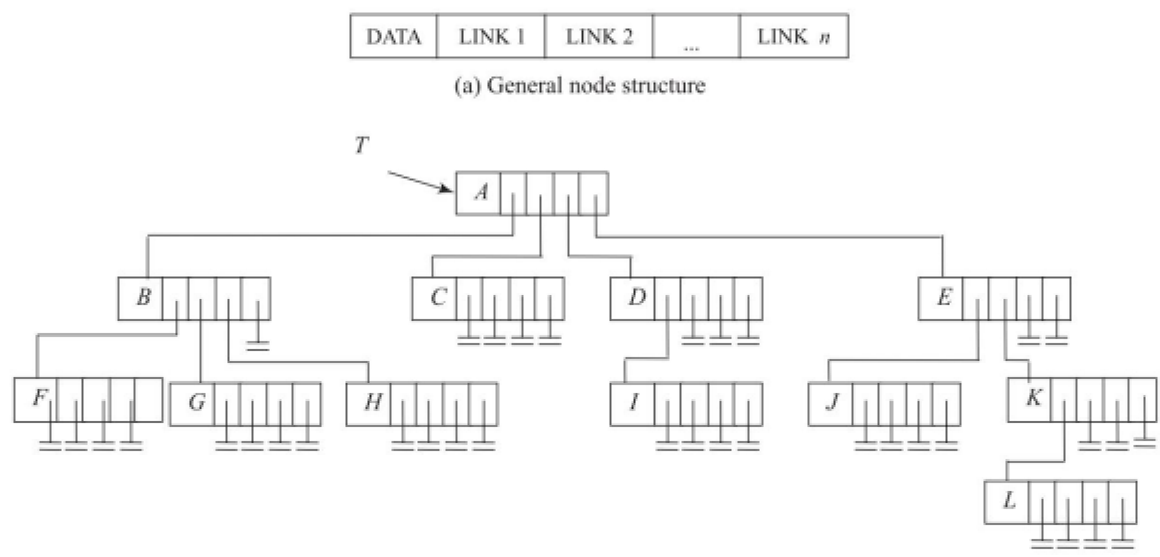
A forest is a set of zero or more disjoint trees. The removal of the root node from a tree results in a forest (of its subtrees!).

Representation of Trees:

A common representation of a tree to suit its storage in the memory of a computer, is a **list**. The tree of above Fig. could be represented in its list form as (A (B(F,G,H), C, D(I), E(J,K(L)))).

The root node comes first followed by the list of subtrees of the node. This is repeated for each subtree in the tree.

This list form of a tree, is a naïve representation of the tree as a linked list. The node structure of the linked list is shown in Fig. (a).



(a) General node structure

(b) Linked list representation of the tree shown in Fig. 8.1

The DATA field of the node stores the information content of the tree node. A fixed set of LINK fields accommodate the pointers to the child nodes of the given node.

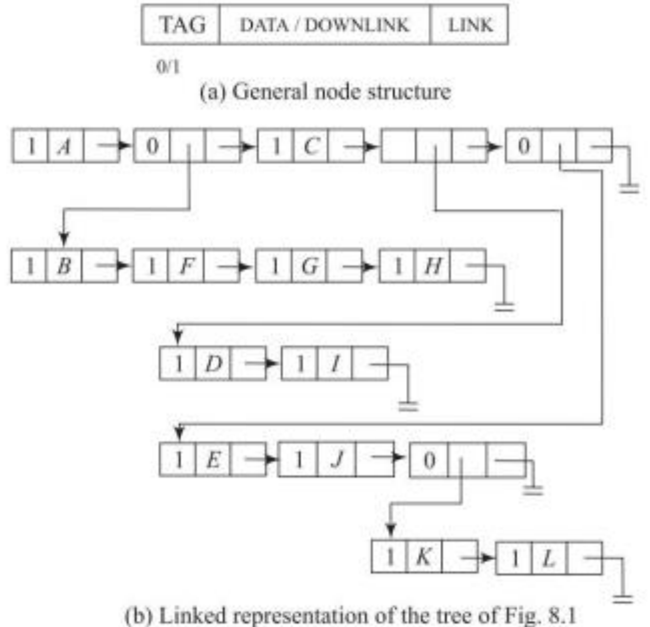
In fact the maximum number of links the node would require is equal to the degree of the tree. The linked representation of the tree illustrated in Fig. (b).

Disadvantage: wastage of space by way of null pointers.

An **alternative representation** would be to use a node structure as shown in Fig.below. Here TAG =1 indicates that the next field (DATA / DOWN LINK) is occupied by data (DATA) and TAG = 0 indicates that the same is used to hold a link (DOWN LINK).

The node structure of the linked list holds a DOWNLINK whenever it encounters a child node which gives rise to a subtree. Thus the root node A has four child nodes, three of which viz., B, D and E give rise to subtrees.

Note the DOWN LINK active fields of the nodes in these cases with TAG set to 0. In contrast, observe the linked list node corresponding to C which has no subtree. The DATA field records C with TAG set to 1.



(a) General node structure

(b) Linked representation of the tree of Fig. 8.1

BINARY TREES:

Basic terminologies:

A binary tree has the characteristic of all nodes having at most two branches, that is, all nodes have a **degree of at most 2**. A binary tree can therefore be empty or consist of a root node and two disjointed binary trees termed **left subtree** and **right subtree**. Figure below illustrates a binary tree.

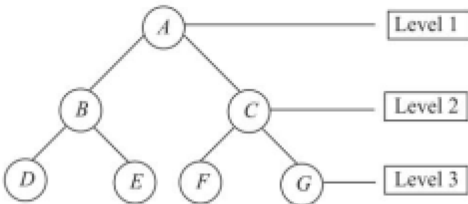


Fig. An example of binary tree

The distinction between trees and binary trees

The distinction between trees and binary trees are while a binary tree can be empty with zero nodes, a tree can never be empty.

Again while the ordering of the subtrees in a tree is immaterial, in a binary tree the distinction of left and right subtrees are very clearly maintained.

Properties:

Some important observations regarding binary trees.

(i) The maximum number of nodes on level i of a binary tree is 2i–1, i ≥ 1.

(ii) The maximum number of nodes in a binary tree of height h is 2h – 1, h ≥ 1.

(iii) For any non-empty binary tree, if to is the number of terminal nodes and t2 is the number of nodes of degree 2, then to = t2 + 1.

These observations could be easily verified on the binary tree shown in above Fig. The maximum number of nodes on level 3 is 232 = 22 = 4.

Also with the height of the binary tree being 3, the maximum number of nodes = 23 1 = 7. Again to = 4 and t2 = 3 which yields to = t2 + 1.4

Types of binary trees:

Full binary tree: A binary tree of height h which has all its permissible maximum number of nodes viz., 2^h-1 intact is known as a full binary tree of height h. Figure (a) illustrates a full binary tree of height 4. Note the specific method of numbering the nodes.

complete binary tree: A binary tree with n' nodes and height h is complete if its nodes correspond to the nodes which are numbered 1 to n (n'≤ n) in a full binary tree of height h.

In other words, a complete binary tree is one in which its nodes follow a sequential numbering that increments from a left-to-right and top-to-bottom fashion. A full binary tree is therefore a special case of a complete binary tree.

Also, the height of a complete binary tree with n elements has a height h given by
$h = \lceil \log 2 (n + 1) \rceil$.

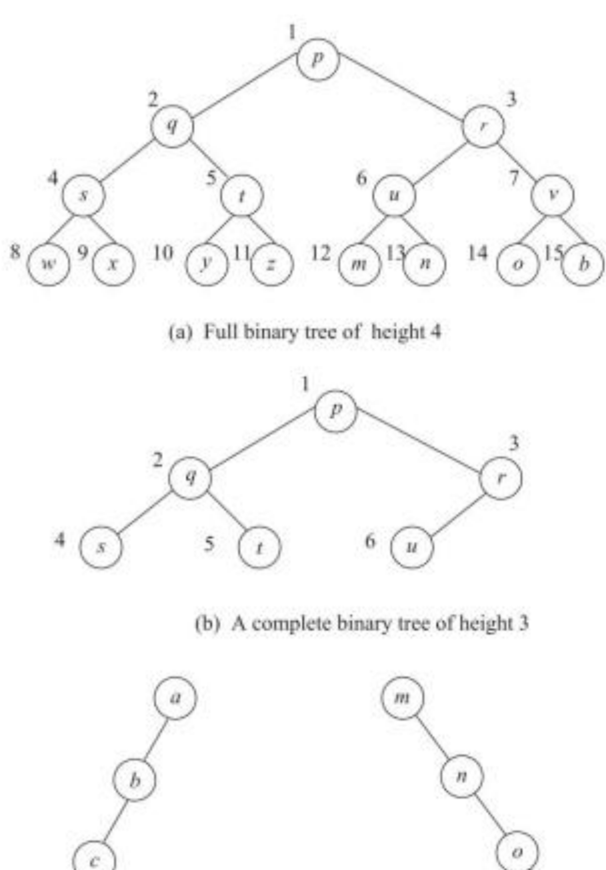
A complete binary tree obeys the following properties with regard to its node numbering: (i) If a parent node has a number i then its left child has the number 2i (2i≤n). If 2i > n then i has no left child.

(ii) If a parent node has a number i, then its right child has the number 2i + 1 (2i + 1 ≤ n). If 2i + 1 > n then i has no right child.

(iii) If a child node (left or right) has a number i then the parent node has the number $\lfloor i/2 \rfloor$ if i ≠ 1. If i = 1 then i is the root and hence has no parent.

Figure (b) illustrates an example complete binary tree.

Skewed binary tree: A binary tree which is dominated solely by left child nodes or right child nodes is called a skewed binary tree or more specifically left skewed binary tree or right skewed binary tree respectively. Figure (c) illustrates examples of skewed binary trees.



(a) Full binary tree of height 4

(b) A complete binary tree of height 3

(c) Skewed binary tree

Representation of Binary Trees:

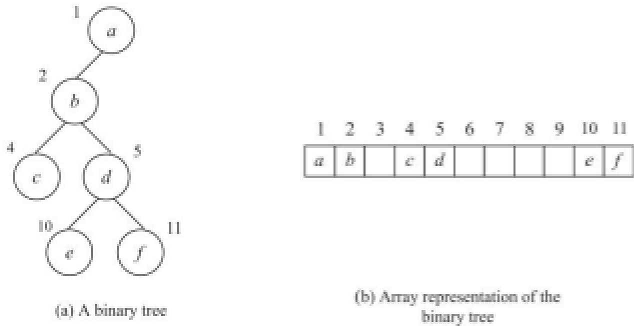
A binary tree could be represented using a **sequential data structure** (arrays) as well as **linked data structure**.

i) Array representation of binary trees:

To represent the binary tree as an array, the sequential numbering system emphasized by a complete binary tree is used. Consider the binary tree shown in Fig. (a) below. The array representation is as shown in Fig. (b) below.

The association of numbers pertaining to parent and left/ right child nodes makes it convenient to access the appropriate cells of the array. However, the missing nodes in the binary tree and hence the corresponding array locations, are left empty in the array.

This obviously leads to a lot of wastage of space. However, the array representation ideally suits a full binary tree due to its non-wastage of space.



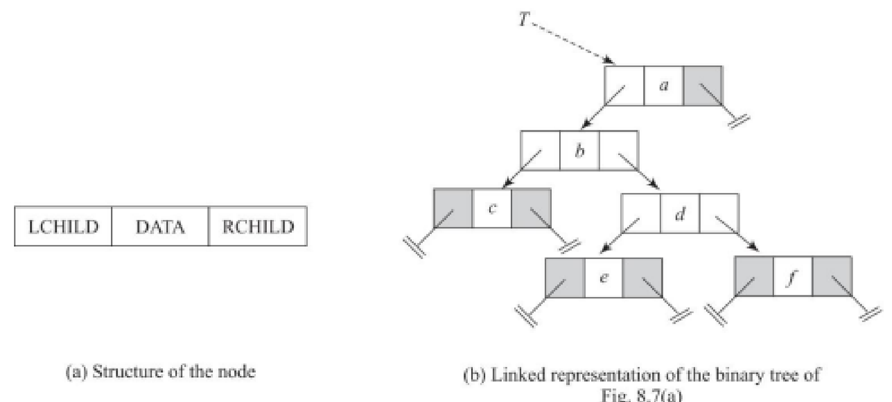
(a) A binary tree

(b) Array representation of the binary tree

Fig. Array representation of a binary tree

ii) Linked representation of binary trees:

The linked representation of a binary tree has the node structure shown in Fig. (a) below. Here, the node, besides the DATA field, needs two pointers LCHILD and RCHILD to point to the left and right child nodes respectively. The tree is accessed by remembering the pointer to the root node of the tree.



(a) Structure of the node

(b) Linked representation of the binary tree of Fig. 8.7(a)

Fig. Linked representation of a binary tree

In the binary tree T shown in Fig. (b), LCHILD (T) refers to the node storing b and RCHILD (LCHILD (T)) refers to the node storing d and so on.

The following are some of the important observations regarding the linked representation of a binary tree:

(i) If a binary tree has n nodes then the number of pointers used in its linked representation is 2 * n.

(ii) The number of null pointers used in the linked representation of a binary tree with n nodes is n + 1.

BINARY TREE TRAVERSALS:

A traversal of a binary tree is where its nodes are visited in a particular but repetitive order, rendering a linear order of the nodes or information represented by them.

A traversal is governed by three actions; viz. **Move left (L), Move Right (R)** and **Process Node (P).** In all, it yields six different combinations of LPR, LRP, PLR, PRL and RLP. Of these, three have emerged significant. They are,

i) LPR - Inorder traversal

ii) LRP - Postorder traversal

iii) PLR - Preorder traversal

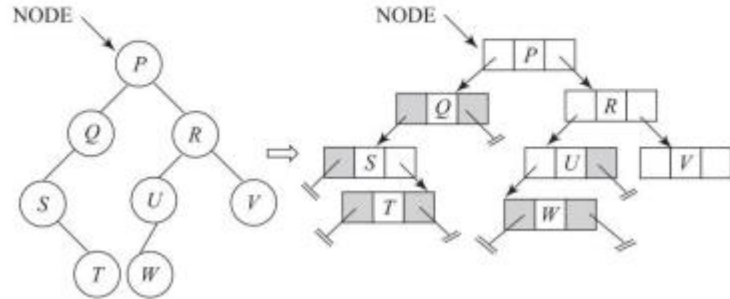


Fig. Binary tree to demonstrate Inorder, Postorder and Preorder traversals

Inorder Traversal:

The traversal keeps moving left in the binary tree until one can move no further, processes the node and moves to the right to continue its traversal again. In the absence of any node to the right, it retracts backwards by a node and continues the traversal.

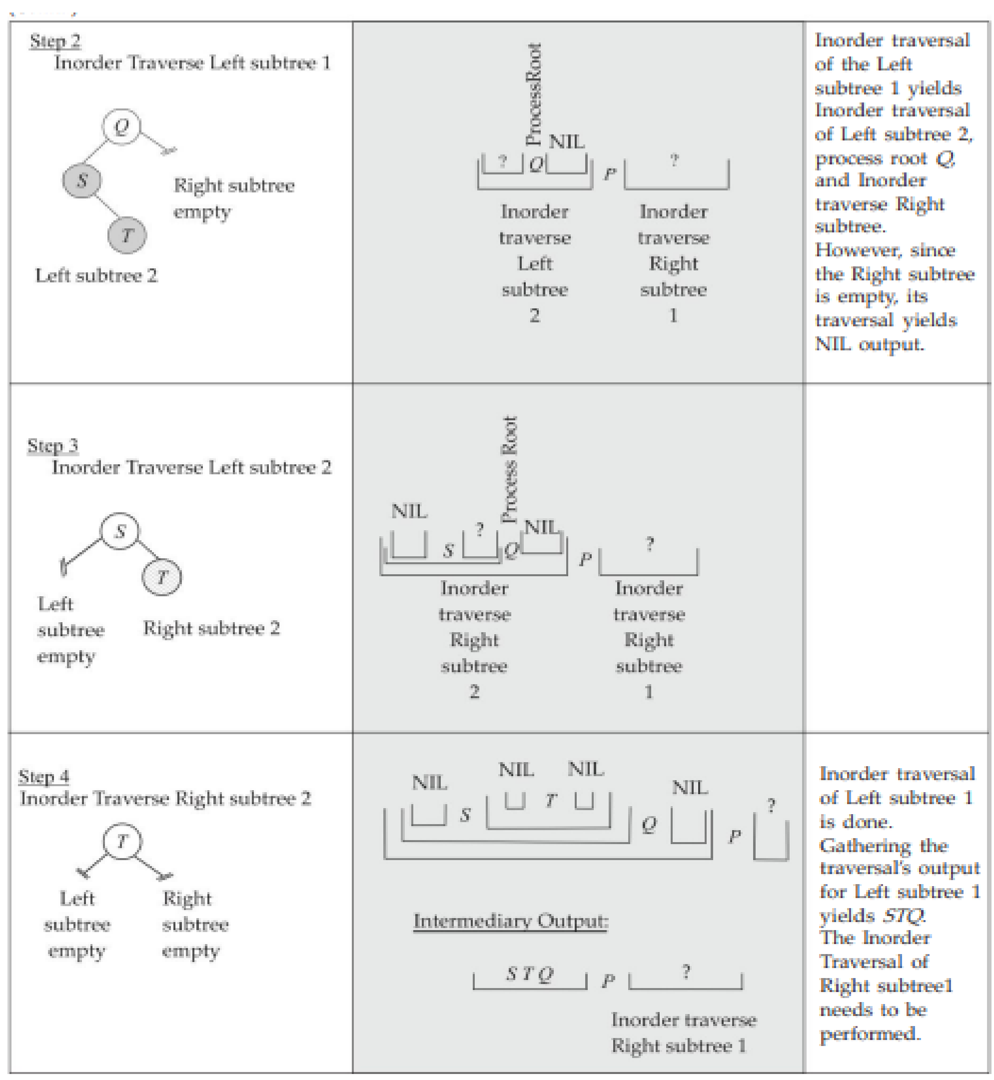
Algorithm : Recursive procedure to perform Inorder traversal of a binary tree

```
procedure INORDER_TRAVERSAL (NODE)
        If NODE ≠ NIL then
        {
        call INORDER_TRAVERSAL (LCHILD(NODE));
        print (DATA (NODE)) ;
        call INORDER_TRAVERSAL (RCHILD(NODE));
        }
end INORDER_TRAVERSAL.
```

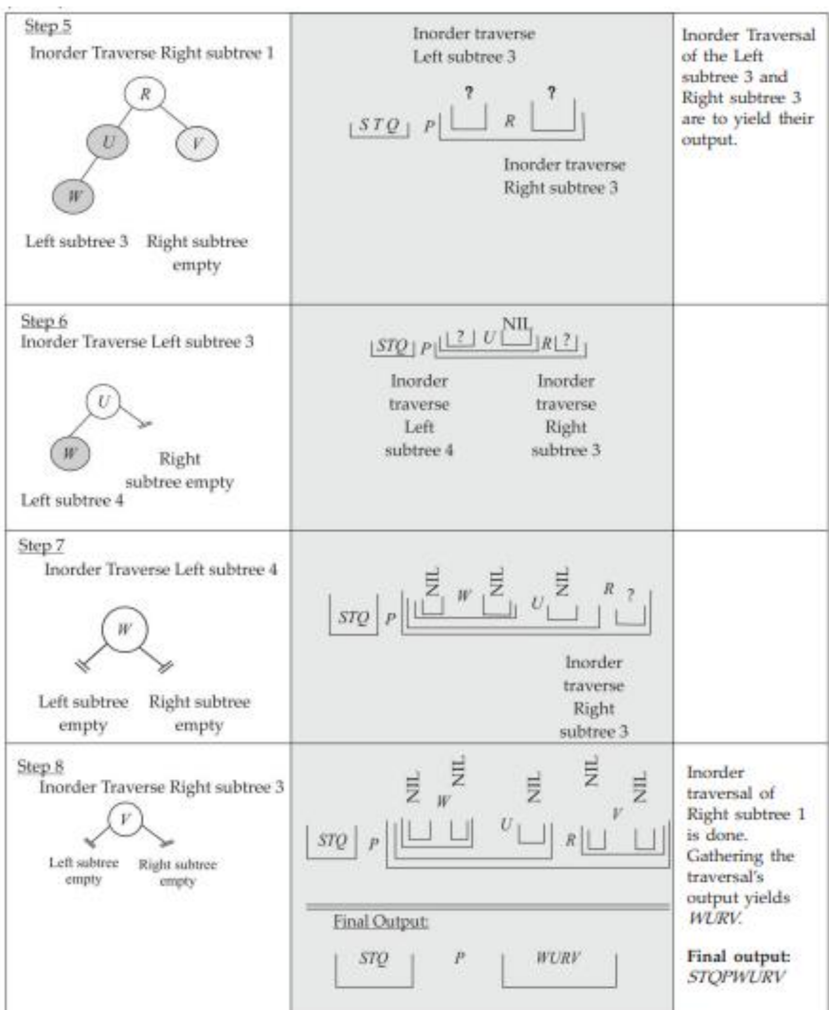
Table : Inorder traversal of binary tree shown in above Fig.

Binary Tree	Inorder Traversal Output	Remarks
Step 1 Inorder Traverse Binary Tree Node → P; Q, R; S, U, V; T, W Left subtree 1 Right subtree 1	Process Root ? P ? Inorder traverse Left subtree 1 / Inorder traverse Right subtree 1	Inorder traversals of the Left and Right subtrees of the root node are to yield their output.

(Contd.)



Step 2 Inorder Traverse Left subtree 1 (Q, S, T; Right subtree empty; Left subtree 2)	? Q NIL P ? — ProcessRoot; Inorder traverse Left subtree 2; Inorder traverse Right subtree 1	Inorder traversal of the Left subtree 1 yields Inorder traversal of Left subtree 2, process root Q, and Inorder traverse Right subtree. However, since the Right subtree is empty, its traversal yields NIL output.
Step 3 Inorder Traverse Left subtree 2 (S, T; Left subtree empty; Right subtree 2)	NIL S ? Q NIL P ? — Process Root; Inorder traverse Right subtree 2; Inorder traverse Right subtree 1	
Step 4 Inorder Traverse Right subtree 2 (T; Left subtree empty; Right subtree empty)	NIL S NIL T NIL Q NIL P ? Intermediary Output: S T Q P ? Inorder traverse Right subtree 1	Inorder traversal of Left subtree 1 is done. Gathering the traversal's output for Left subtree 1 yields *STQ*. The Inorder Traversal of Right subtree1 needs to be performed.



Postorder Traversal:

The traversal proceeds by keeping to the left until it is no further possible, turns right to begin again or if there is no node to the right, processes the node and retraces its direction by one node to continue its traversal.

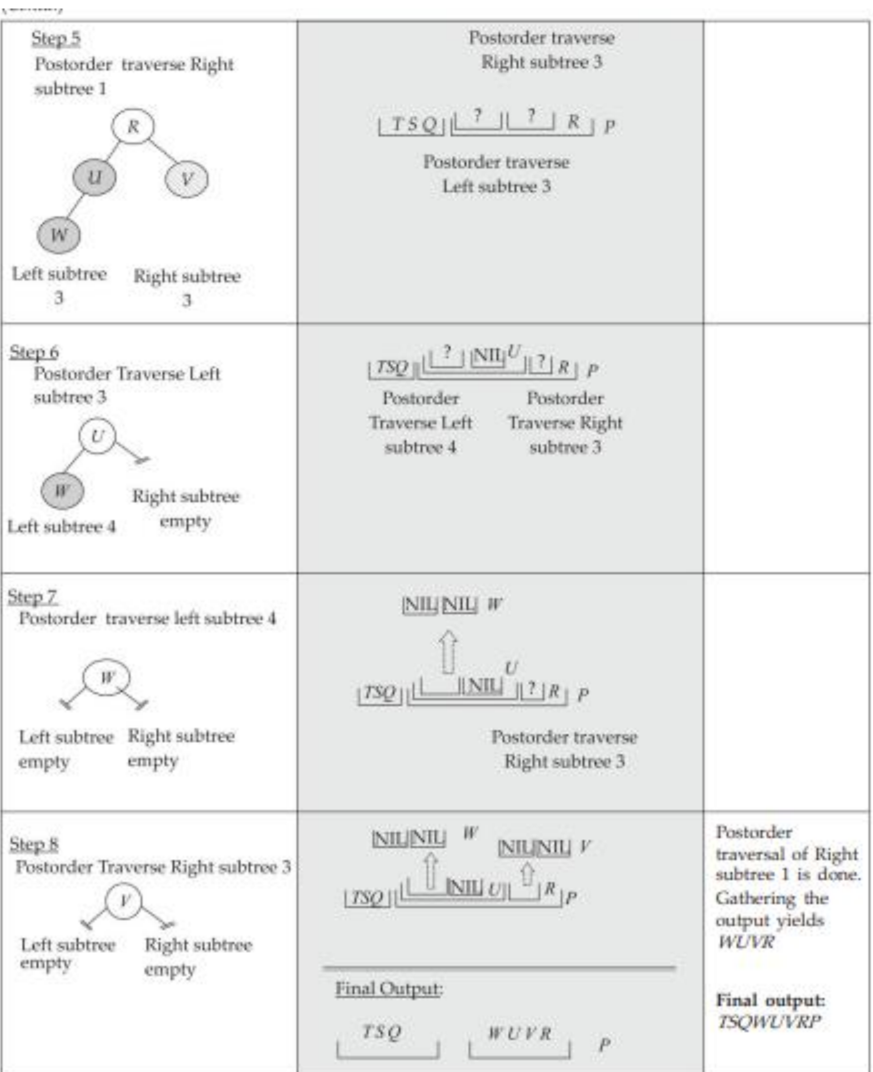
Algorithm: Recursive procedure to perform Postorder traversal of a binary tree

```
procedure POSTORDER_TRAVERSAL (NODE)
            /* NODE refers to the Root node of the binary tree in its first call to the
            procedure. Root node is the starting point of the traversal */
      If NODE ≠ NIL then
      {
      call POSTORDER_TRAVERSAL (LCHILD(NODE));
            / * Postorder traverse the left subtree (L) */
      call POSTORDER_TRAVERSAL (RCHILD(NODE));
            /* Postorder traverse the right subtree (R)*/
       print (DATA (NODE)) ;        /* Process node (P) */
      }
end POSTORDER_TRAVERSAL.
```

Table: Postorder traversal of binary tree shown in above Fig.

Binary Tree	Postorder Traversal Output	Remarks
Step 1 Postorder traverse Binary Tree Node P Q R S U V T W Left subtree 1 Right subtree 1	Process Root ? ? P Postorder traverse Left subtree 1 Postorder traverse Right subtree 1	Postorder Traversal of the Left and Right subtrees of the root is yet to yield their output.
Step 2 Postorder Traverse Left subtree 1 Q S T Right subtree empty Left subtree 2	Postorder traverse Left subtree 2 ? NIL Q ? P Postorder traverse Right subtree 1	
Step 3 Postorder Traverse Left subtree 2 S T Left subtree empty Right subtree 2	NIL ? S NIL Q ? P Postorder traverse Right subtree 2 Postorder traverse Right subtree 1	
Step 4 Postorder traverse Right subtree 2 T Left subtree empty Right subtree empty	NIL NIL NIL T S NIL Q ? P Intermediary Output: T S Q ? P	Postorder traversal of Left subtree 1 is done. Gathering the traversal's output for Left subtree 1 yields TSQ

(Contd.)



Preorder Traversal:

The traversal processes every node as it moves left until it can move no further. Now it turns right to begin again or if there is no node in the right, retracts until it can move right to continue its traversal.

Algorithm: Recursive procedure to perform Preorder traversal of a binary tree procedure

```
PREORDER_TRAVERSAL (NODE)
          /* NODE refers to the Root node of the binary tree in its first call to the
          procedure. Root node is the starting point of the traversal */
  If NODE ≠ NIL then
     {
     print (DATA (NODE)) ; /* Process node (P) */
     call PREORDER_TRAVERSAL (LCHILD(NODE));
          / * Preorder traverse the left subtree (L) */
     call PREORDER_TRAVERSAL (RCHILD(NODE));
          /* Preorder traverse the right subtree (R)*/
          }
```

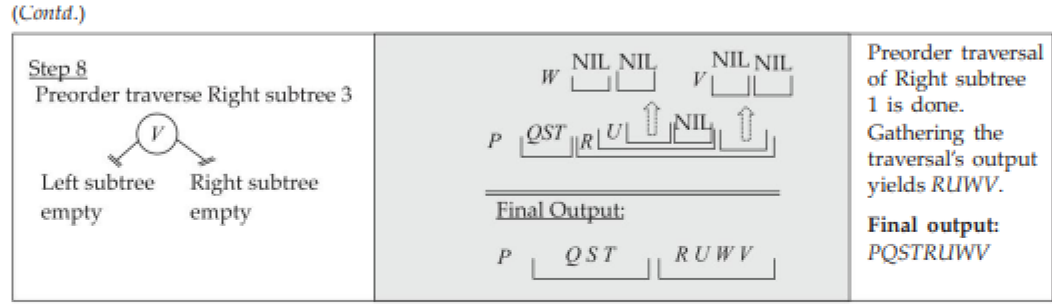
end PREORDER_TRAVERSAL.

Table: Preorder traversal of binary tree shown in above Fig.

Binary Tree	Preorder Traversal Output	Remarks
Step 1 Preorder traversal of Binary Tree Node P, Q, R, S, U, V, T, W Left subtree 1 Right subtree 1	Process Root P ? ? Preorder traverse Left subtree 1 / Preorder traverse Right subtree 1	Preorder traversals of the Left subtree 1 and Right subtree 1 to yield their output.
Step 2 Preorder traverse Left subtree 1 Q, S, T Right subtree empty Left subtree 2	Preorder traverse Left subtree 2 P Q ? NIL ? Preorder traverse Right subtree 1	
Step 3 Preorder traverse Left subtree 2 S, T Left subtree empty Right subtree 2	Preorder traverse Right subtree 2 P Q S NIL ? NIL ? Preorder traverse Right subtree 1	
Step 4 Preorder traverse Right subtree 2 T Left subtree empty Right subtree empty	T NIL NIL P Q S NIL NIL ? Intermediary Output: P QST ?	Preorder traversal of Left subtree 1 is done. Gathering the traversal's output yields *QST*
Step 5 Preorder traverse Right subtree 1 R, U, V, W Left subtree 3 Right subtree 3	Preorder traverse Left subtree 3 P QST R ? ? Preorder traverse Right subtree 3	
Step 6 Preorder traverse Left subtree 3 U, W Right subtree empty Left subtree 4	Preorder traverse Left subtree 4 P QST R U ? NIL ? Preorder traverse Right subtree 3	
Step 7 Preorder traverse left subtree 4 W Left subtree empty Right subtree empty	W NIL NIL P QST R U NIL ? Preorder traverse Right subtree 3	

(Contd.)

(Contd.)



Step 8 Preorder traverse Right subtree 3 V Left subtree empty Right subtree empty	W NIL NIL V NIL NIL P QST R U NIL Final Output: P QST RUWV	Preorder traversal of Right subtree 1 is done. Gathering the traversal's output yields *RUWV*. **Final output:** *PQSTRUWV*

Some significant observations pertaining to the traversals of a binary tree are the following,

(i) Given a preorder traversal of a binary tree, the root node is the first occurring item in the list.

(ii) Given a postorder traversal of a binary tree, the root node is the last occurring item in the list.

(iii) Inorder traversal does not directly reveal the root node of the binary tree.

(iv) An inorder traversal coupled with any one of preorder or post order traversal helps trace back the structure of the binary tree

BINARY SEARCH TREES:

Definition :

A binary search tree T may be an empty binary tree. If non-empty, then for a set S, a binary search tree T satisfies the following norms:

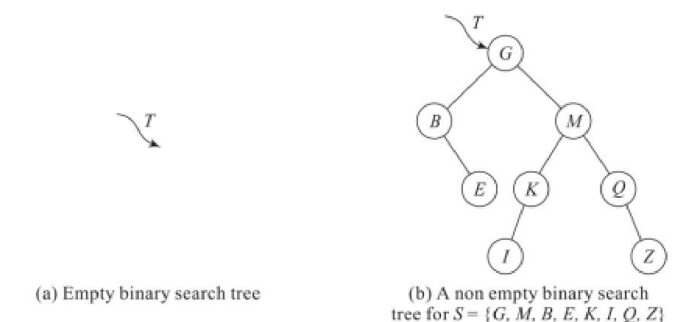
(i) all keys of the binary search tree must be distinct

(ii) all keys in the left subtree of T are less than the root element

(iii) all keys in the right subtree of T are greater than the root element and

(iv) the left and right subtrees of T are also binary search trees.

- The inorder traversal of a binary search tree T yields the elements of the associated set S in the ascending order.



(a) Empty binary search tree

(b) A non empty binary search tree for $S = \{G, M, B, E, K, I, Q, Z\}$

Representation of a binary search tree:

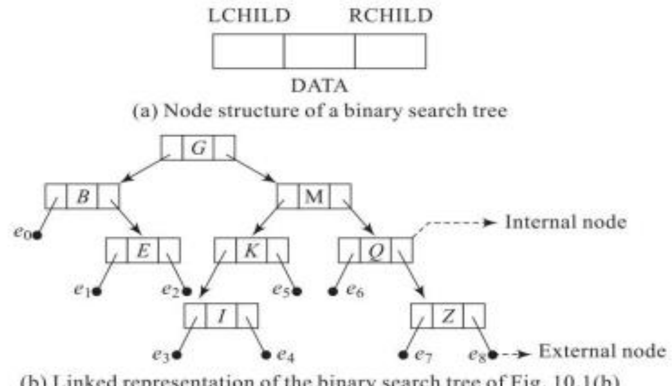
A binary search tree is commonly represented using a linked representation in the same way as that of a binary tree.

The node structure and the linked representation of the binary search tree shown in above Fig. is illustrated in below Fig.

However, the null pointers of the nodes may be represented using fictitious nodes called **external nodes**.

Thus a linked representation of a binary search tree is viewed as a bundle of external nodes which represent the null pointers and **internal nodes** which represent the keys.

Such a binary tree is referred to as an **extended binary tree**. Obviously, the number of external nodes in a binary search tree comprising n internal nodes is n+1. The path from the root to an external node is called as an **external path**.



(a) Node structure of a binary search tree

(b) Linked representation of the binary search tree of Fig. 10.1(b)

Binary Search Trees Operations:

Retrieval from a binary search tree Let T be a binary search tree. To retrieve a key u from T, u is first compared with the root key r of T.

If u = r then the search is done. If u < r then the search begins at the left subtree of T. If u > r then the search begins at the right subtree of T.

The search is repeated recursively in the left and right sub-subtrees with u compared against the respective root keys, until the key u is either found or not found.

If the key is found the search is termed successful and if not found, is termed unsuccessful.

While all successful searches terminate at the appropriate internal nodes in the binary search tree, all unsuccessful searches terminate only at the external nodes in the appropriate portion of the binary search tree. Hence external nodes are also referred to as failure nodes.

Algorithm : Procedure to retrieve ITEM from a binary search tree T

procedure FIND_BST(T, ITEM, LOC)

/* LOC is the address of the node containing ITEM which is to be retrieved from the binary search tree T. In case of unsuccessful search the procedure prints the message ITEM not found and returns LOC as NIL.*/

if T = NIL then

{

```
      print ( binary search tree T is empty);
     exit;
   }                    /* exit procedure*/
  else
   LOC = T;
   while (LOC ≠ NIL) do
   case
           : ITEM = DATA(LOC)
                     return (LOC);              /* ITEM found in node LOC*/
           : ITEM < DATA(LOC)
                     LOC = LCHILD(LOC);   /* search left subtree*/
           : ITEM > DATA(LOC)
                     LOC= RCHILD(LOC);    /* search right subtree*/
    Endcase
  endwhile
  If (LOC=NIL) then
  {
           print(ITEM not found);
         return (LOC)
  }          /* unsuccessful search*/
  end FIND_BST15
```

Example: Consider the set S = {416, 891, 456, 765, 111, 654, 345, 256, 333} whose associated binary search tree T is shown in Fig below.

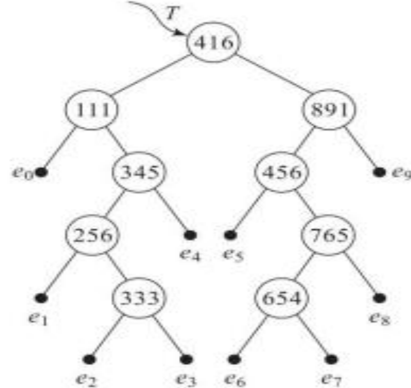


Table: Trace of Algorithm for the retrieval of ITEM=333

LOC	ITEM <, =, > DATA(LOC)?	Updated LOC
Initially LOC = T = #(416)	333 < 416	LOC = LCHILD(#(416)) = #(111)
LOC = #(111)	333 > 111	LOC = RCHILD(#(111)) = #(345)
LOC = #(345)	333 < 345	LOC = LCHILD(#(345)) = #(256)
LOC = #(256)	333 > 256	LOC = RCHILD(#(256)) = #(333)
LOC = #(333)	333 = 333	RETURN(#(333)) Element found and node returned

Table: Trace of Algorithm for the retrieval of ITEM=777

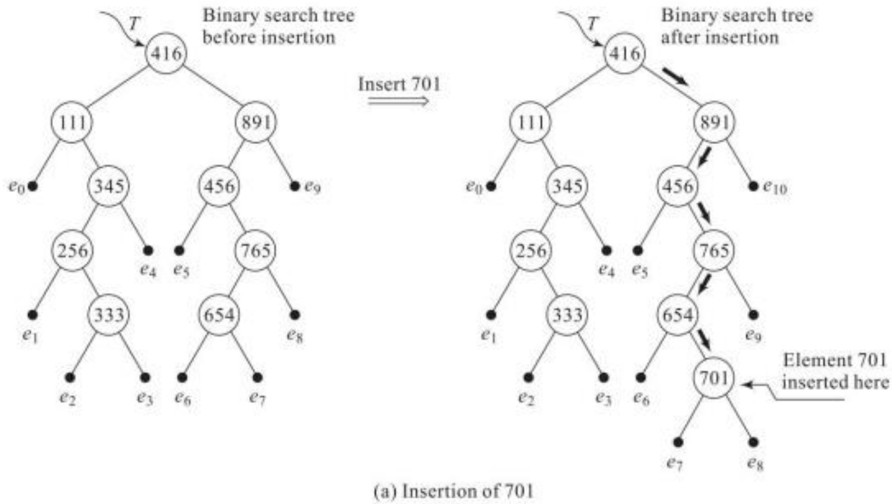
LOC	ITEM <, =, > DATA(LOC)?	Updated LOC
Initially LOC = T = # (416)	777 > 416	LOC = RCHILD(# (416)) = # (891)
LOC = # (891)	777 < 891	LOC = LCHILD(# (891)) = # (456)
LOC = # (456)	777 > 456	LOC = RCHILD(# (456)) = # (765)
LOC = # (765)	777 > 765	LOC = RCHILD(# (765)) = NIL
LOC = NIL		Element not found RETURN (NIL)

<u>Insertion into a binary search tree:</u>

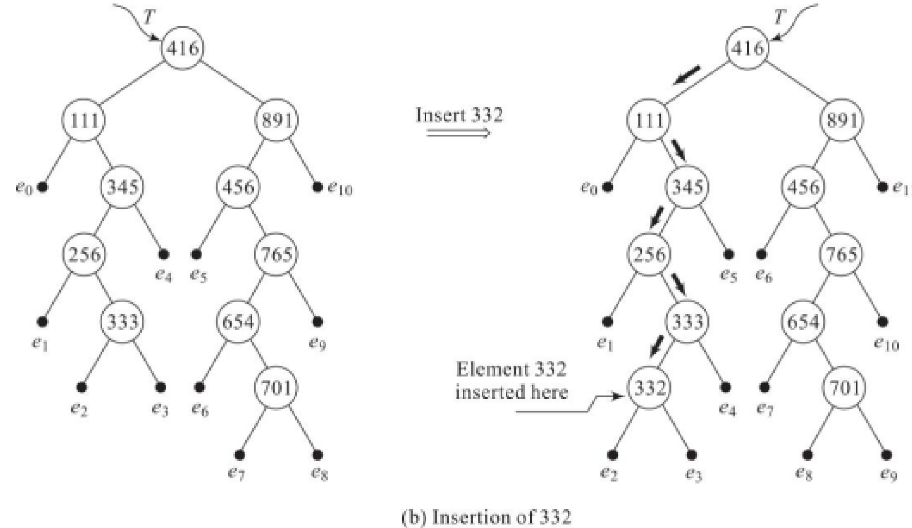
The insertion of a key into a binary search tree is similar to the retrieval operation. The insertion of a key u initially proceeds as if it were trying to retrieve the key from the binary search tree, but on reaching the null pointer (failure node) which it is sure to encounter since key u is not present in the tree, a new node containing the key u is inserted at that position.

Example: Let us insert keys 701 and 332 into the binary search tree T associated with set S ={ 416, 891, 456, 765, 111, 654, 345, 256, 333}, shown in Fig below, Figure (a) shows the insertion of 701.

Note how the operation moves down the tree in the path shown and when it encounters a failure node e7, the key 701 is inserted as the right child of node containing 654. Again the insertion of 332 which follows a similar procedure is illustrated in Fig. (b).



(a) Insertion of 701



(b) Insertion of 332

Deletion from a binary search tree:

For the deletion of a key from the binary search tree we first search for the node containing the key by undertaking a retrieval operation.

But once the node is identified, the following cases are tested before the node containing the key u is appropriately deleted from the binary search tree T:

(i) key u is a leaf node

(ii) key u has a lone subtree (left subtree or right subtree only)

(iii) key u has both left subtree and right subtree

Case (i) If the key u to be deleted is a leaf node then the deletion is trivial since the appropriate link field of the parent node of key u only needs to be set as NIL.

Case (ii) If the key u to be deleted has either a left subtree or a right subtree (but not both) then the link of the parent node of u is set to point to the appropriate subtree.

Case (iii) If the key u to be deleted has both a left subtree and a right subtree, then the problem is complicated. In this case since the right subtree comprises keys that are greater than u, the parent node of key u is now set to point to the right subtree of u.

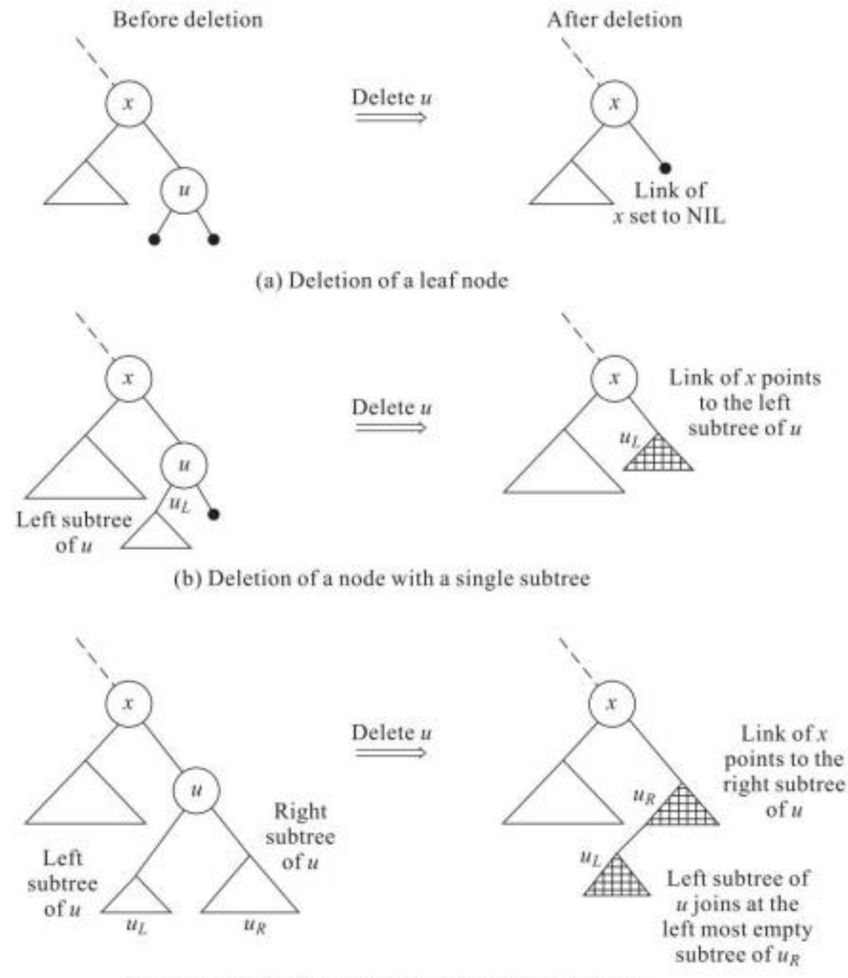
Since all the keys of the left subtree of u are less than that of the right subtree of u, we move as far left as possible in the right subtree of u until an empty left subtree is found and link the left subtree of u at that position.

Algorithm : Procedure to delete a node NODE_U from a binary search tree given its parent node NODE_X

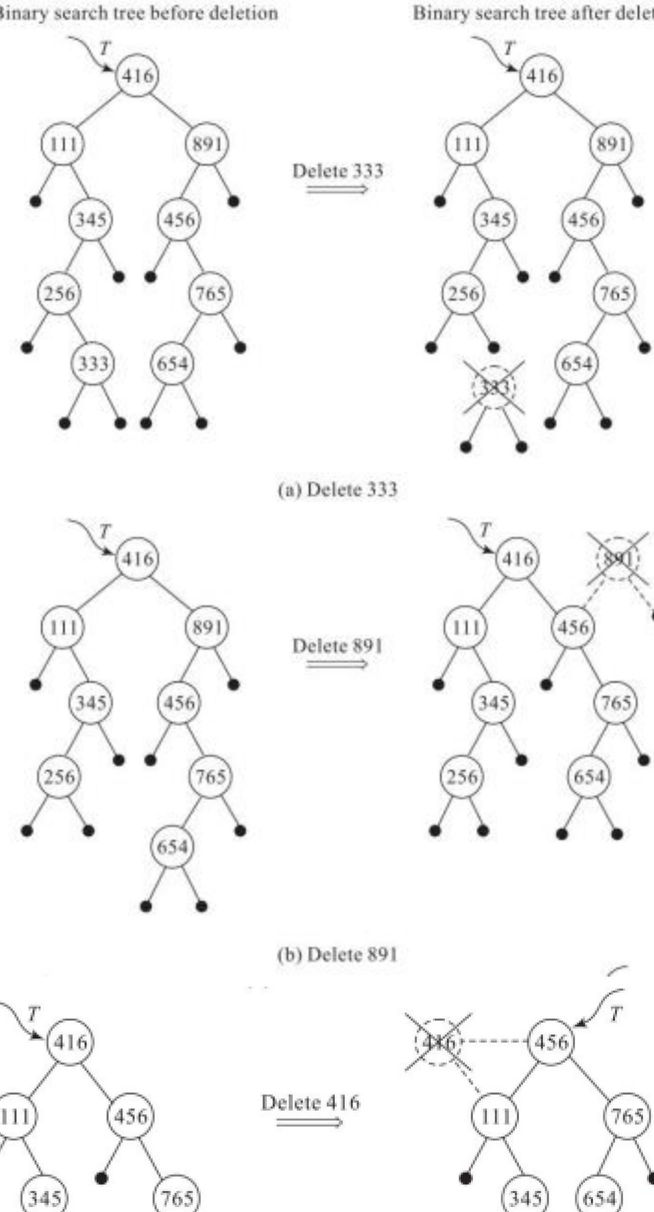
procedure DELETE(NODE_U, NODE_X)

/* NODE_U is the node which is to be deleted from the binary search tree. NODE_X is the parent node for which NODE_U may be the left child or the right child. Procedure DELETE is applicable for deletion of all non-empty nodes other than the root (i.e.) NODE_U ≠ NIL and NODE_X ≠ NIL */

```
case
    :LCHILD(NODE_U )=RCHILD(NODE_U )=NIL:
         Set RCHILD(NODE_X) or LCHILD(NODE_X)to NIL based on whether NODE_U is the right
child or left child of NODE_X respectively;
        call RETURN(NODE_U );

 :LCHILD(NODE_U )< > NIL and RCHILD(NODE_U )<> NIL:

          Set RCHILD(NODE_X) or LCHILD(NODE_X) to RCHILD(NODE_U) based on whether
NODE_U is the right child or left child of NODE_X respectively;

         TEMP= RCHILD(NODE_U);
          while (LCHILD(TEMP) <> NIL) do
         TEMP=LCHILD(TEMP);
          endwhile
         LCHILD(TEMP) = LCHILD(NODE_U);
          call RETURN (NODE_U);

 :LCHILD (NODE_U)< > NIL and RCHILD(NODE_U) = NILTEMP=LCHILD(NODE_U);
         Set RCHILD(NODE_X) or LCHILD(NODE_X) to TEMP based on whether NODE_U is the right
child or left child of NODE_X respectively;
          call RETURN(NODE_U);

  :LCHILD(NODE_U) = NIL and RCHILD(NODE_U) < > NIL:
         TEMP=RCHILD(NODE_U);
         Set RCHILD(NODE_X) or LCHILD(NODE_X )to TEMP based on whether NODE_U is the right
child or left child of NODE_X respectively;
         call RETURN(NODE_U );
 endcase
end DELETE
```



Example: Delete keys 333, 891 and 416 in the order given, from the binary search tree T associated with set S = {416, 891, 456, 765, 111, 654, 345, 256, 333} shown in below Fig.



(a) Delete 333

(b) Delete 891

(c) Delete 416

GRAPHS:

Definitions and Basic Terminologies:

Graph A graph G = (V, E) consists of a finite non empty set of vertices V also called points or nodes and a finite set E of unordered pairs of distinct vertices called edges or arcs or links.

Example Figure below illustrates a graph. Here V = {a, b, c, d} and E = {(a, b), (a, c), (b, c), (c, d)}. However it is convenient to represent edges using labels as shown in the figure.

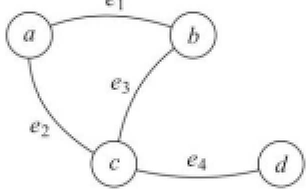


Fig. A graph

V : Vertices : {a, b, c, d}

E : Edges : {e1, e2, e3, e4}

- A graph G = (V, E) where E = ∅, is called as **a null** or **empty graph**.
- A graph with one vertex and no edges is called **a trivial graph**.

Multigraph:

A multigraph G = (V, E) also consists of a set of vertices and edges except that E may contain **multiple edges** (i.e.) edges connecting the same pair of vertices, or may contain **loops or self edges** (i.e.) an edge whose end points are the same vertex.

Example Figure below illustrates a multigraph Observe the multiple edges e1, e2 connecting vertices a, b and e5, e6, e7 connecting vertices c, d respectively. Also note the self edge e4.

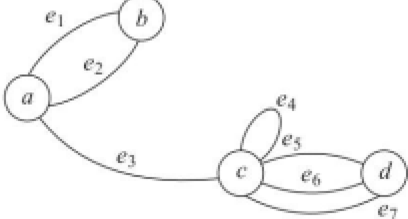


Fig. A multigraph

Directed and undirected graphs:

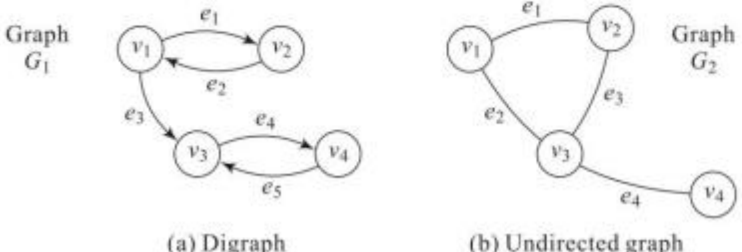
A graph whose definition makes reference to unordered pairs of vertices as edges is known as an undirected graph.

The edge eij of such an undirected graph is represented as (vi , vj) where vi , vj are distinct vertices. Thus an undirected edge (vi , vj) is equivalent to (vj , vi).

On the other hand, **directed graphs or digraphs** make reference to edges which are directed (i.e.) edges which are ordered pairs of vertices.

The edge eij is referred to as which is distinct from < vj , vi > where vi , vj are distinct vertices.

Example: Figure below illustrates a digraph and an undirected graph.



(a) Digraph (b) Undirected graph

In Fig. (a), e1 is a directed edge between v1 and v2, (i.e.) e1 = <v1,v2>, whereas in Fig. (b) e1 is an undirected edge between v1 and v2, (i.e.) e1 = (v1, v2).

The list of vertices and edges of graphs G1 and G2 are:

Vertices (G1) : {v1, v2, v3, v4}

Vertices (G2) : {v1, v2, v3, v4}

Edges (G1) : {<v1,v2>,<v2,v1>,<v1,v3>,<v3,v4>,<v4,v3> } or {e1, e2, e3, e4, e5}

Edges (G2) : {(v1, v2) (v1, v3) (v2, v3) (v3, v4)} or {e1, e2, e3, e4}

- In the case of an undirected edge (vi , vj) in a graph, the vertices vi , vj are said to be **adjacent** or the edge (vi , vj) is said to be **incident** on vertices vi , vj .
- Thus in Fig. (b) vertices v1, v3 are adjacent to vertex v2 and edges e1: (v1, v2), e3: (v2, v3) are incident on vertex v2.
- On the other hand, if is a directed edge, then vi is said to be adjacent to vj and vj is said to be adjacent from vi . The edge is incident to both vi and vj .

Complete graphs:

The number of distinct unordered pairs (vi , vj), vi ≠ vj in a graph with n vertices is $^nC_2$= (n(n-1))/2. An n vertex undirected graph with exactly (n(n-1))/2 edges is said to be complete.

Example Figure below illustrates a complete graph. The undirected graph with 4 vertices has all its $^4C_2$ = 6 edges intact.

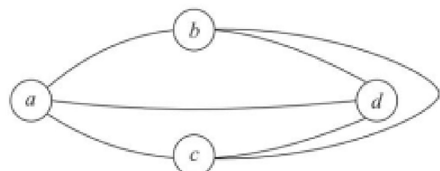
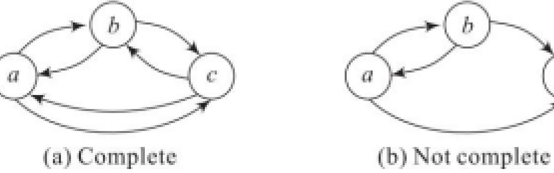


Fig. A complete graph

In the case of a digraph with n vertices, the maximum number of edges is given by $^nP_2$ = n. (n-1).

Such a graph with exactly n.(n-1) edges is said to be a complete digraph.

Example Figure (a) illustrates a digraph which is complete and Fig. (b) a graph which is not complete.

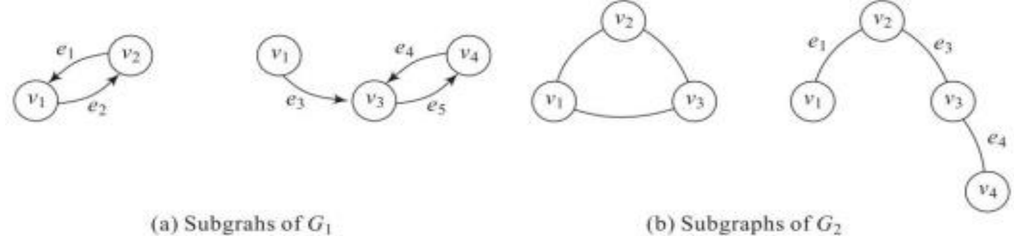


(a) Complete (b) Not complete

Subgraph:

A subgraph $G^{1'} = (V^{1'}, E^{1'})$ of a graph G = (V, E) is such that $V^1 \subseteq V$ and $E^1 \subseteq E$.

Example Figure below illustrates some subgraphs of the directed and undirected graphs shown in Fig. (Graphs G1 and G2)



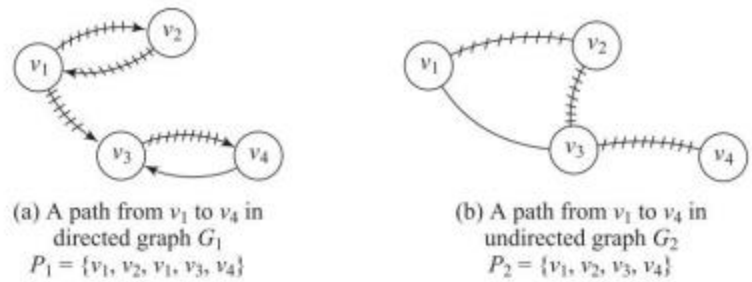
(a) Subgrahs of G_1 (b) Subgraphs of G_2

Path:

A path from a vertex vi to vertex vj in an undirected graph G is a sequence of vertices such that (vi , v_{l1}) (v_{l1} , v_{l2}), ..(v_{lk} , vj) are edges in G.

If G is directed then the path from vi to vj more specially known as a directed path consists of edges $< vi , v_{l1}> < v_{l1} , v_{l2}> .. < v_{lk} , vj >$ in G.

Example Figure (a) illustrates a path P1 from vertex v1 to v4 in graph G1 and Fig (b) illustrates a path P2 from vertex v1 to v4 of graph G2.



(a) A path from v_1 to v_4 in directed graph G_1 $P_1 = \{v_1, v_2, v_1, v_3, v_4\}$

(b) A path from v_1 to v_4 in undirected graph G_2 $P_2 = \{v_1, v_2, v_3, v_4\}$

The length of a path is the number of edges on it. Example In above Fig. the length of path P1 is 4 and the length of path P2 is 3.

A simple path is a path in which all the vertices except possibly the first and last vertices are distinct.

Example In graph G2, the path from v1 to v4 given by {(v1, v2), (v2, v3), (v3, v4)} and written as {v1, v2, v3, v4} is a simple path where as the path from v3 to v4 given by {(v3, v1), (v1, v2), (v2, v3), (v3, v4)} and written as {v3, v1, v2, v3, v4} is not a simple path but a path due to the repetition of vertices.

A cycle is a simple path in which the first and last vertices are the same. A cycle is also known as a **circuit**, **elementary cycle**, **circular path** or **polygon**.

Example In graph G2 the path {v1, v2, v3, v1} is a cycle. Also, in graph G1 the path {v1, v2, v1} is a cycle or more specifically a directed cycle.

Connected graphs:

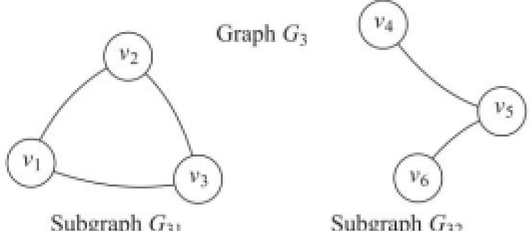
Two vertices vi , vj in a graph G are said to be connected only if there is a path in G between vi and vj .

In an undirected graph if vi and vj are connected then it automatically holds that vj and vi are also connected. An undirected graph is said to be a connected graph if every pair of distinct vertices vi , vj are connected.

Example Graph G2 is connected where as graph G3 shown in below Fig. is not connected.

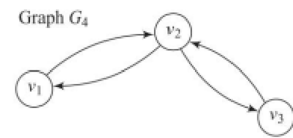
In the case of an undirected graph which is not connected, the **maximal connected subgraph** is called as a **connected component** or simply a component.

Example Graph G3 has two connected components viz., graph G31 and G32.

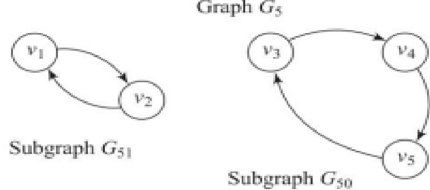


A directed graph is said to be **strongly connected** if every pair of distinct vertices vi , vj are connected (by means of a directed path). Thus if there exists a directed path from vi to vj then there also exists a directed path from vj to vi .

Example Graph G4 shown in Fig. is strongly connected.



However, the digraph shown in Fig. is not strongly connected but is said to possess two strongly connected components. A strongly connected component is a maximal subgraph that is strongly connected.



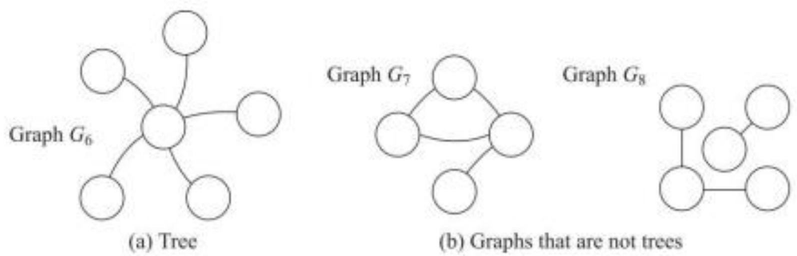
Trees:

A tree is defined to be a connected acyclic graph. The following properties are satisfied by a tree:

(i) There exists a path between any two vertices of the tree, and

(ii) No cycles must be present in the tree. In other words, trees are acyclic.

Example Figure (a) illustrates a tree. Figure (b) illustrates graphs which are not trees due to the violation of the property of acyclicity and connectedness respectively.



Degree:

The degree of a vertex in an undirected graph is the number of edges incident to that vertex.

A vertex with degree one is called as a **pendant vertex** or **end** vertex.

A vertex with degree zero and hence has no incident edges is called an **isolated** vertex. **Example** In graph G2 the degree of vertex v3 is 3 and that of vertex v2 is 2.

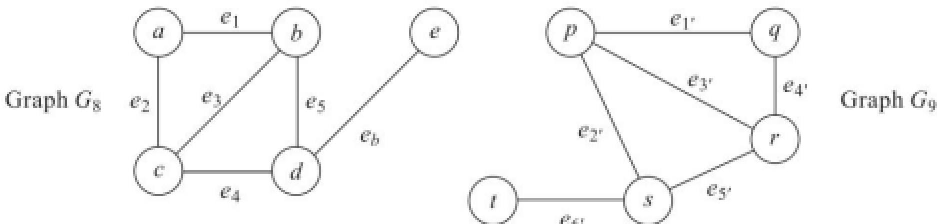
In the case of digraphs, we define the **indegree** of a vertex v to be the number of edges with v as the head and the **outdegree** of a vertex to be number of edges with v as the tail. **Example** In graph G1 the indegree of vertex v3 is 2 and the out degree of vertex v4 is 1.

Isomorphic graphs:

Two graphs are said to be isomorphic if,

(i) they have the same number of vertices
(ii) they have the same number of edges
(iii) they have an equal number of vertices with a given degree

Example Figure illustrates two graphs which are isomorphic.



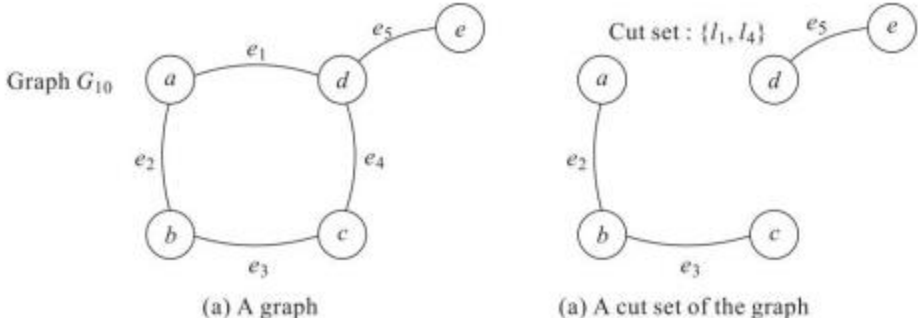
The property of isomorphism can be verified on the lists of vertices and edges of the two graphs G8 and G9 when superimposed as shown below:

Vertices (G_8) :	a	b	c	d	e	
	↕	↕	↕	↕	↕	
Vertices (G_9) :	q	p	r	s	t	
Degree of the vertices :	2	3	3	3	1	
Edges (G_8) :	e_1	e_2	e_3	e_4	e_5	e_6
Edges (G_9) :	e_1'	e_4'	e_3'	e_2'	e_5'	e_6'

Cut set:

A cut set in a connected graph G is the set of edges whose removal from G leaves G disconnected, provided the removal of no proper subset of these edges disconnects the graph G. Cut sets are also known as **proper cut set** or **cocycle** or **minimal cut set**.

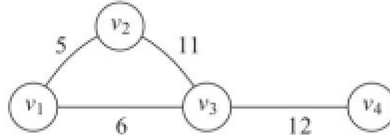
Example Figure below illustrates the cut set of the graph G10. The cut set {e1, e4} disconnects the graph into two components as shown in the figure. {e5} is also another cut set of the graph.



Labeled graphs:

A graph G is called a labeled graph if its edges and / or vertices are assigned some data. In particular if the edge e is assigned a non negative number l(e) then it is called the weight or length of the edge e.

Example Figure below illustrates a labeled graph. A graph with weighted edges is also known as a network.

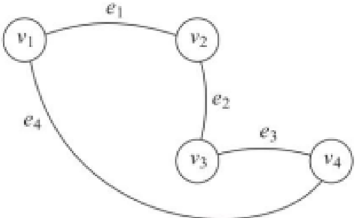


Eulerian graph:

A walk starting at any vertex going through each edge exactly once and terminating at the start vertex is called an Eulerian walk or Euler line.

The Koenigsberg bridge problem was in fact a problem of obtaining an Eulerian walk for the graph concerned. The solution to the problem is, an Eulerian walk is possible only if the degree of each vertex in the graph is even.

Given a connected graph G, G is an Euler graph iff all the vertices are of even degree. Example Figure below illustrates an Euler graph. {e1, e2, e3, e4} shows a Eulerian walk. The even degree of the vertices may be noted.

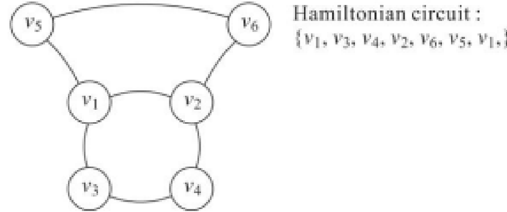


Hamiltonian circuit:

A Hamiltonian circuit in a connected graph is defined as a closed walk that traverses every vertex of G exactly once, except of course the starting vertex at which the walk terminates.

A circuit in a connected graph G is said to be Hamiltonian if it includes every vertex of G. If any edge is removed from a Hamiltonian circuit then what remains is referred to as a Hamiltonian path. Hamiltonian path traverses every vertex of G.

Example Figure below illustrates a Hamiltonian circuit.



Representations of Graphs:

The representation of graphs in a computer can be categorized as

1. sequential representation and
2. linked representation.

Of the two, though sequential representation has several methods, all of them follow a matrix representation thereby calling for their implementation using arrays. The linked representation of a graph makes use of a singly linked list as its fundamental data structure.

1. Sequential representation of graphs

The sequential or the matrix representation of graphs have the following methods:

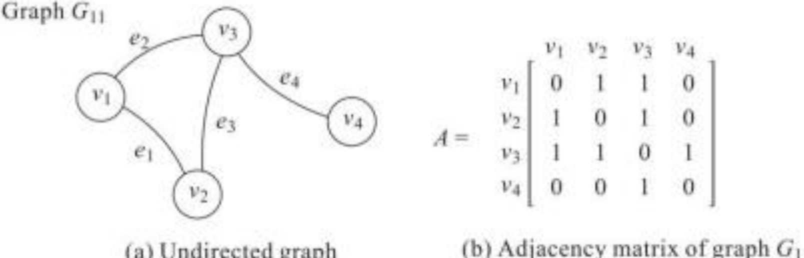
(i) Adjacency matrix representation
(ii) Incidence matrix representation
(iii) Circuit matrix representation
(iv) Cut set matrix representation
(v) Path matrix representation

i) Adjacency matrix representation :

The adjacency matrix of a graph G with n vertices is an n X n symmetric binary matrix given by A = [aij] defined as aij = 1 if the i th and jth vertices are adjacent (i.e.) there is an edge connecting the i th and j th vertices

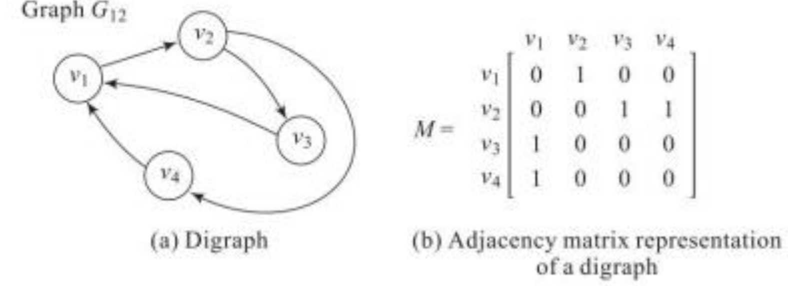
= 0 otherwise, (i.e.) if there is no edge linking the vertices.

Example Figure (a) illustrates an undirected graph whose adjacency matrix is shown in Fig. (b). It can easily be seen that while adjacency matrices of undirected graphs are symmetric, nothing can be said about the symmetricity of the adjacency matrix of digraphs.



(a) Undirected graph (b) Adjacency matrix of graph G_{11}

Example Figure below illustrates a digraph and its adjacency matrix representation.



(a) Digraph (b) Adjacency matrix representation of a digraph

ii) Incidence matrix representation :

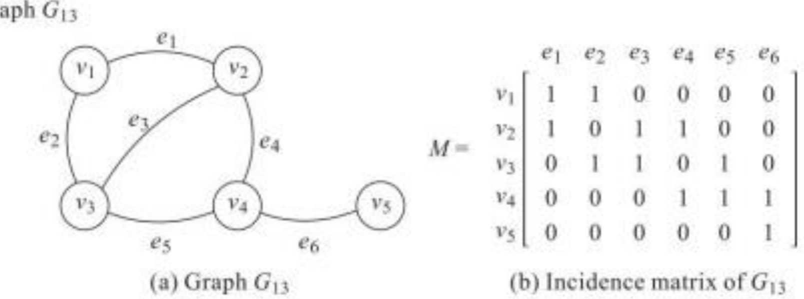
Let G be a graph with n vertices and e edges. Define an n X e matrix M = [mij] whose n rows correspond to n vertices and e columns correspond to e edges, as

mij = 1 if the j th edge ej is incident on the i th vertex vi ,

= 0 otherwise

Matrix M is known as the incidence matrix representation of the graph G.

Example Consider the graph G13 shown in below Fig. (a), the incidence matrix representation for the graph is given in Fig. (b).



(a) Graph G_{13} (b) Incidence matrix of G_{13}

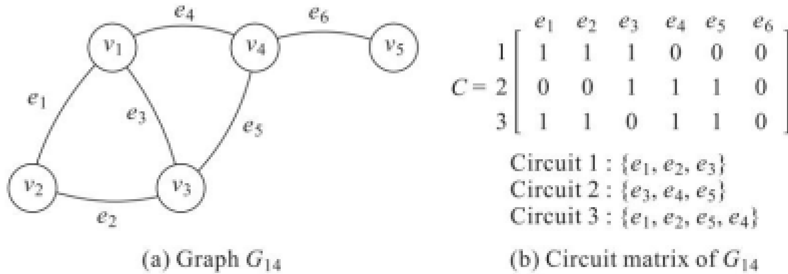
iii) Circuit matrix representation :

For a graph G let the number of different circuits be t and the number of edges be e. Then the circuit matrix C= [Cij] of G is a t X e matrix defined as

Cij = 1 if the i th circuit includes the j th edge,

= 0 otherwise

Example Consider the graph G14 shown in below Fig. (a). The circuits for this graph expressed in terms of their edges are 1: {e1, e2, e3} 2: {e3, e4, e5} 3: {e1, e2, e5, e4}. The circuit matrix C of order 3 X 6 is shown in Fig. (b).



(a) Graph G_{14} (b) Circuit matrix of G_{14}

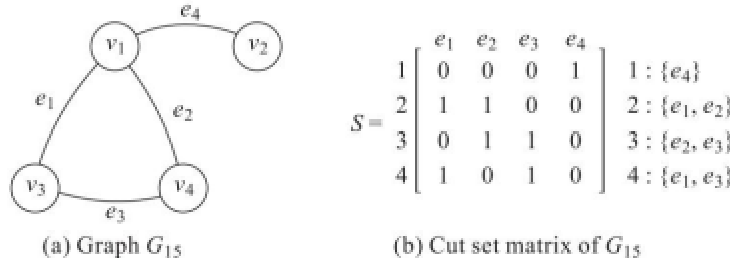
iv) Cut set matrix representation:

For a graph G, a matrix S = [sij] whose rows correspond to cut sets and columns correspond to edges of the graph is defined to be a cut set matrix if

sij = 1 if the i th cut set contains the j th edge,

= 0 otherwise

Example Consider the graph G15 shown in below Fig. (a). The cut sets of the graph are 1:{e4} 2:{e1, e2} 3:{e2, e3} and 4:{e1, e3}. The cut set matrix representation is shown in Fig. (b).



(a) Graph G_{15} (b) Cut set matrix of G_{15}

v) Path matrix representation :

A path matrix is generally defined for a specific pair of vertices in a graph. If (u, v) is a pair of vertices then the path matrix denoted as P(u,v) = [pij] is given by

pij = 1 if the jth edge lies in the i th path between vertices u and v,

= 0 otherwise

Example Consider the graph G16 shown in Fig. (a). The paths between vertices v1 and v4 are 1:{e2, e4} and 2:{e1, e3, e4}. The path matrix representation is shown in Fig (b).

$$P(v_1, v_4) = \begin{array}{c} \\ 1 \\ 2 \end{array} \begin{array}{cccc} e_1 & e_2 & e_3 & e_4 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{array} \quad \begin{array}{l} \text{Paths :} \\ 1 : \{e_2, e_4\} \\ 2 : \{e_1, e_3, e_4\} \end{array}$$

(a) Graph G_{16} (b) Path matrix between v_1 v_4 of G_{16}

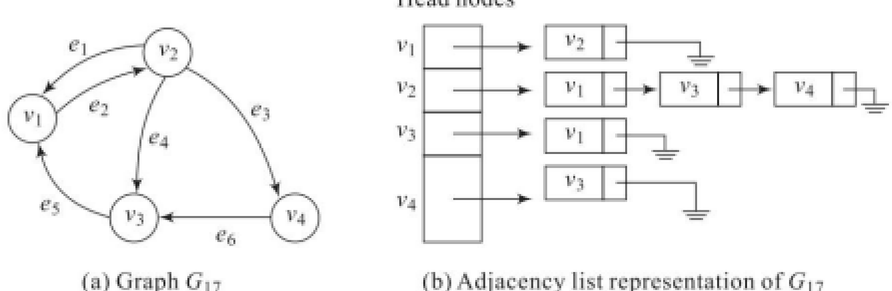
Of all these sequential representations, adjacency matrix representation represents the graph best and is the most widely used representation.

2. Linked representation of graphs :

The linked representation of graphs is referred to as **adjacency list representation** and is comparatively efficient with regard to adjacency matrix representation.

Given a graph G with n vertices and e edges, the adjacency list opens n head nodes corresponding to the n vertices of graph G, each of which points to a singly linked list of nodes, which are adjacent to the vertex representing the head node.

Example Figure below illustrates a graph and its adjacency list representation. It can easily be seen that if the graph is undirected, then the number of nodes in the singly linked lists put together is 2e where as in the case of digraphs the number of nodes is just e, where e is the number of edges in the graph.



(a) Graph G_{17} (b) Adjacency list representation of G_{17}

Graph Traversals:

Graphs support the following traversals:

1. Breadth first Traversal, and
2. Depth first Traversal.

A traversal, to recall, is a systematic walk which visits the nodes comprising the data structure (graphs in this case) in a specific order.

1. Breadth first traversal:

The breadth first traversal starts from a vertex u which is said to be visited. Now all nodes vi , adjacent to u are visited.

The unvisited vertices wij adjacent to each of vi are visited next and so on. The traversal terminates when there are no more nodes to visit.

The process calls for the maintenance of a queue to keep track of the order of nodes whose adjacent nodes are to be visited.

Algorithm: Breadth first traversal

Procedure BFT(s)

/* s is the start vertex of the traversal in an undirected graph G */

/* Q is a queue which keeps track of the vertices whose adjacent nodes are to be visited */

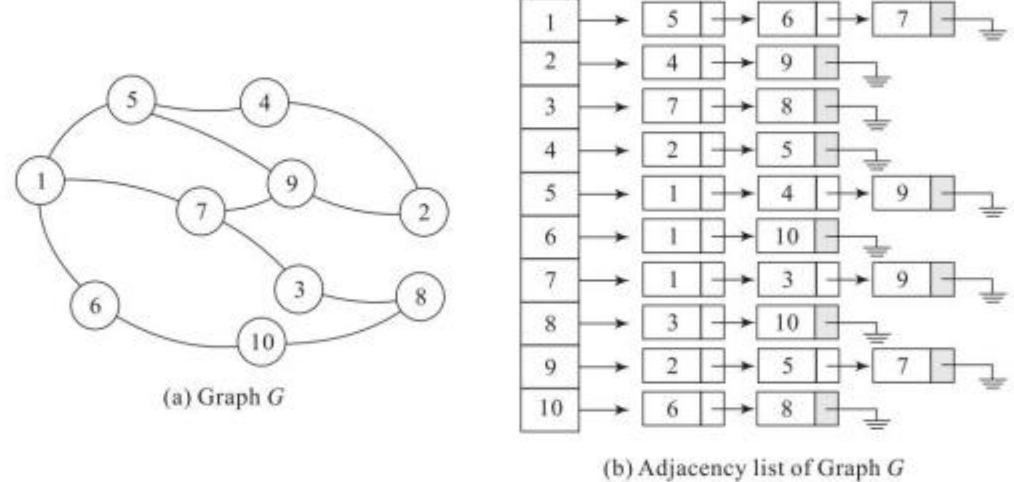
/* Vertices which have been visited have their ' visited' flags set to 1 (i.e.) visited (vertex) = 1. Initially, visited (vertex) = 0 for all vertices of graph G */

Initialize queue Q;
visited(s) = 1;
call ENQUEUE (Q,s); /* insert s into Q */

```
        while not EMPTY_QUEUE(Q) do                    /* process until Q is empty */
                call DEQUEUE (Q,s)                     /* delete s from Q*/
                print (s);                            /* output vertex visited */
                 for all vertices v adjacent to s do
                    if (visited (v) = 0) then
                          {
                           call ENQUEUE (Q, v);
                          visited (v) =1;
                          }
                   end
         endwhile
end BFT.
```

Breadth first traversal as its name indicates traverses the successors of the start node, generation after generation in a horizontal or linear fashion.

Example: Consider the undirected graph G shown in below Fig. (a) and its adjacency list representation shown in Fig.(b). The trace of procedure BFT(1) where the start vertex is 1, is shown in Table below.



(a) Graph *G*

(b) Adjacency list of Graph *G*

Fig. A graph and its adjacency list representation to demonstrate breadth first traversal

Table: Trace of the Breadth first traversal procedure on graph G.

Current Vertex	Queue *Q*	Traversal output	Status of visited flag of vertices {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} of graph *G*									
			1	2	3	4	5	6	7	8	9	10
1 (Start vertex)	1		1	0	0	0	0	0	0	0	0	0
1	5 6 7	1	1	0	0	0	1	1	1	0	0	0
5	5 7 4 9	1 5	1	0	0	1	1	1	1	0	1	0
6	7 4 9 10	1 5 6	1	0	0	1	1	1	1	0	1	1
7	4 9 10 3	1 5 6 7	1	0	1	1	1	1	1	0	1	1
4	9 10 3 2	1 5 6 7 4	1	1	1	1	1	1	1	0	1	1
9	10 3 2	1 5 6 7 4 9	1	0	1	1	1	1	1	0	1	1
10	3 2 8	1 5 6 7 4 9 10	1	1	1	1	1	1	1	1	1	1
3	2 8	1 5 6 7 4 9 10 3	1	1	1	1	1	1	1	1	1	1
2	8	1 5 6 7 4 9 10 3 2	1	1	1	1	1	1	1	1	1	1
8		1 5 6 7 4 9 10 3 2 8	1	1	1	1	1	1	1	1	1	1
		1 5 6 7 4 9 10 3 2 8	**Breadth first traversal ends**									

The breadth first traversal starts from vertex 1 and visits vertices 5,6,7 which are adjacent to it, while enqueuing them into queue Q.

In the next shot, vertex 5 is dequeued and its adjacent, unvisited vertices 4, 9 are visited next and so on. The process continues until the queue Q which keeps track of the adjacent vertices is empty.

Depth first traversal:

The depth first traversal of an undirected graph starts from a vertex u which is said to be visited. Now, all the nodes vi adjacent to vertex u are collected and the first occurring vertex v1 is visited, deferring the visits to other vertices.

The nodes adjacent to v1 viz., w1k are collected and the first occurring adjacent vertex viz., w11 is visited deferring the visit to other adjacent nodes and so on. The traversal progresses until there are no more visits possible.

Algorithm : Depth first traversal

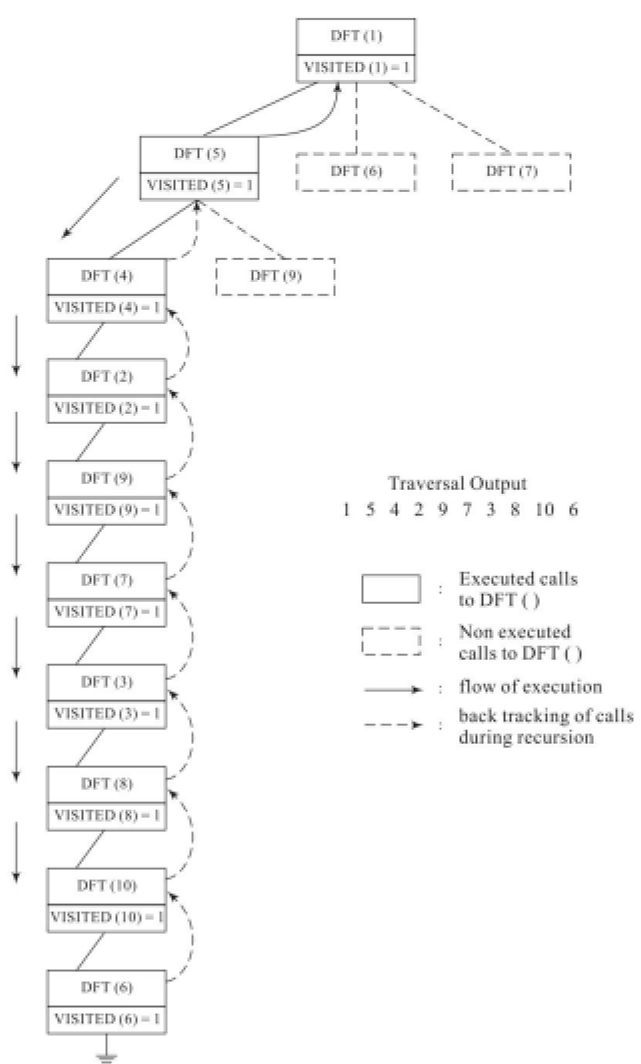
```
Procedure DFT(s)                /* s is the start vertex */
        visited(s) = 1;
        print (s);              /* Output visited vertex */
        for each vertex v adjacent to s do
            if visited(v) = 0 then
             call DFT(v);
         end
        end DFT
```

The depth first traversal as its name indicates visits each node, that is, the first occurring among its adjacent nodes and successively repeats the operation, thus moving deeper and deeper into the graph.

In contrast, breadth first traversal moves sideways or breadth ways in the graph. **Example** illustrates a depth first traversal of a undirected graph.

Consider the undirected graph G and its adjacency list representation shown in above Fig.

Figure shows a tree of recursive calls which represents a trace of the procedure DFT(1) on the graph G with start vertex 1.



The tree of recursive calls illustrates the working of the DFT procedure. The first call DFT(1) visits start vertex 1 and releases 1 as the traversal output.

Vertex 1 has vertices 5, 6, 7 as its adjacent nodes. DFT(1) now invokes DFT(5), visiting vertex 5 and releasing it as the next traversal output.

However DFT(6) and DFT(7) are kept in waiting for DFT(5) to complete its execution. Such procedure calls waiting to be executed are shown in broken line boxes in the tree of recursive calls.

Now DFT(5) invokes DFT(4) releasing vertex 4 as the traversal output while DFT(9) is kept in abeyance. Note that though vertex 1 is an adjacent node of vertex 5, since no DFT( ) calls to vertices already visited are invoked, DFT(1) is not called for.

The process continues until DFT(6) completes its execution with no more nodes left to visit. During recursion the calls made to DFT( ) procedure are indicated using solid arrows in the forward direction.

Once DFT(6) finishes execution, back tracking takes place which is indicated using broken arrows in the reverse direction. Once DFT(1) completes execution the traversal output is gathered to be **1 5 4 2 9 7 3 8 10 6.**

UNIT-VI

AVL Trees: Definition :

Trees whose height in the worst case turns out to be O(log n) are known as balanced trees or height balanced trees. One such balanced tree viz., AVL trees. AVL trees were proposed by Adelson-Velskiiand Landis in 1962.

An empty binary tree is an AVL tree. If non empty, the binary tree T is an AVL tree if

(i) TL and TR, the left and right subtrees of T are also AVL trees and

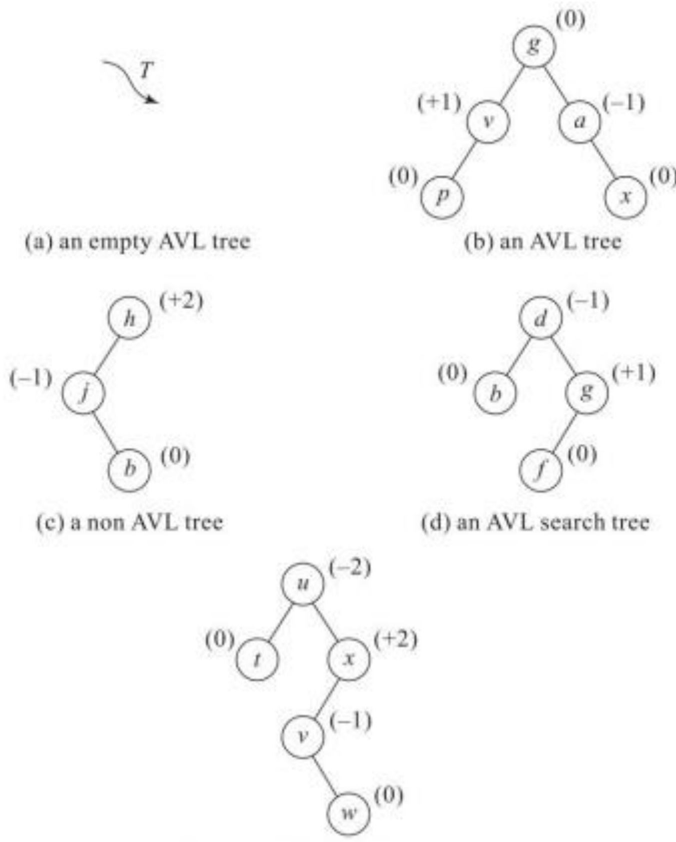
(ii) (ii) |h(TL) – h(TR)| ≤ 1, where h(TL) and h(TR) are the heights of the left subtree and right subtree of T respectively.

For a node u, bf(u) = (h(uL) – h(uR)) where h(uL) and h(uR) are the heights of the left and right subtrees of the node u respectively, is known as the **balance factor (bf)** of the node u.

In an AVL tree therefore, every node u has a balance factor bf(u) which may be either 0 or +1 or –1.

A binary search tree T which is an AVL tree is referred to as an AVL search tree.

Ex:



(a) an empty AVL tree

(b) an AVL tree

(c) a non AVL tree

(d) an AVL search tree

(e) a non AVL search tree

AVL trees and AVL search trees just like binary trees or binary search trees may be represented using a linked representation adopting the same node structure.

Operations:

Retrieval from an AVL search tree:

The retrieval of a key from an AVL search tree is in no way different from the retrieval operation on a binary search tree.

However, since the height of the AVL search tree of n elements is O(log n), the time complexity of the find procedure when applied on AVL search trees does not exceed O(log n).

Insertion into an AVL search tree:

The insertion of an element u into an AVL search tree T proceeds exactly as one would to insert u in a binary search tree.

However, if after insertion the balance factor of any of the nodes turns out to be anything other than 0 or +1 or 1, then the tree is said to be unbalanced.

To balance the tree we undertake what are called rotations. Rotations are mechanisms which shift some of the subtrees of the unbalanced tree to obtain a balanced tree.

For the initiation of rotations, it is required that the balance factors of all nodes in the unbalanced tree are limited to 2, 1, 0, 1, and +2.

Also the rotation is initiated with respect to an ancestor node A that is closest to the newly inserted node u and whose balance factor is either +2 or 2.

If a node w after insertion of node u reports a balance factor of bf(w) = +2 or 2 respectively, then its balance factor before insertion should have been +1 or 1 respectively.

The insertion of a node can only change the balance factors of thosenodes on the path from the root to the inserted node.

If the closest ancestor node A of the inserted node u has a balance factor bf(A) = +2 or –2, then prior to insertion the balance factors of all nodes on the path from A to u must have been 0.

The rotations which are of four different types are listed below. The classification is based on the position of the inserted node u with respect to the ancestor node A which is closest to the node u and reports a balance factor of 2 or +2.

(i) LL rotation - node u is inserted in the left subtree (L) of left subtree (L) of A
(ii) LR rotation - node u is inserted in the right subtree (R) of left subtree (L) of A
(iii) RR rotation - node u is inserted in the right subtree (R) of right subtree (R) of A
(iv) RL rotation - node u is inserted in the left subtree (L) of right subtree (R) of A

Each of the four classes of rotations are illustrated with examples.

LL rotation:

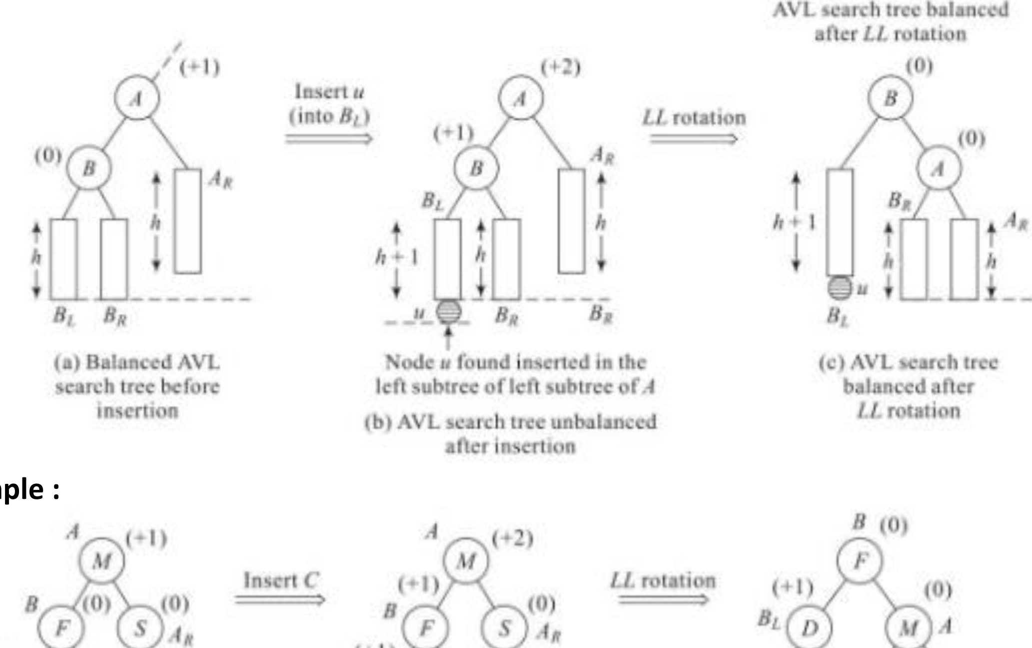
Figure below illustrates a generic representation of LL type imbalance and the corresponding rotation that is undertaken to set right the imbalance.

After insertion of node u, the closest ancestor node of node u, viz., node A, reporting an imbalance (bf(A) = +2) is first found out.

Now with reference to the ancestor node A, we find that the node u has been inserted in the left subtree (L) of left subtree (L) of A.

This implies there is an LL type of imbalance and to balance the tree an LL rotation is to be called for.

The AVL tree before insertion of u (Fig.(a)), the unbalanced tree after insertion of u (Fig.(b)) and the balanced tree after the LL rotation (Fig.(c)) have been illustrated.

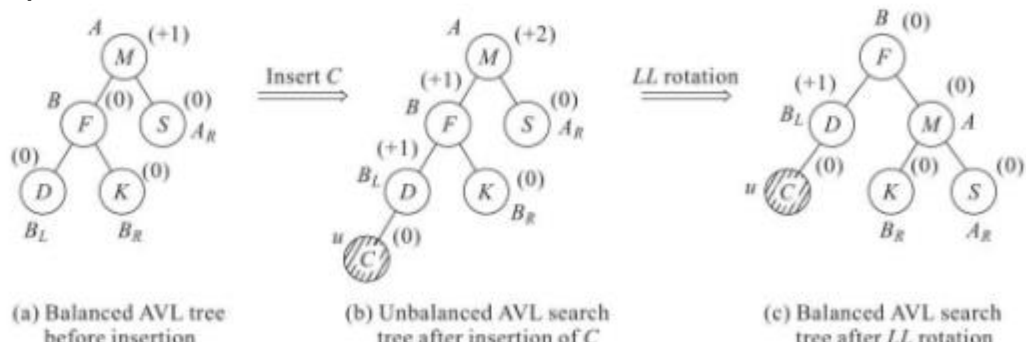


(a) Balanced AVL search tree before insertion

(b) AVL search tree unbalanced after insertion

(c) AVL search tree balanced after LL rotation

Example :



(a) Balanced AVL tree before insertion

(b) Unbalanced AVL search tree after insertion of C

(c) Balanced AVL search tree after LL rotation

<u>LR rotation:</u>

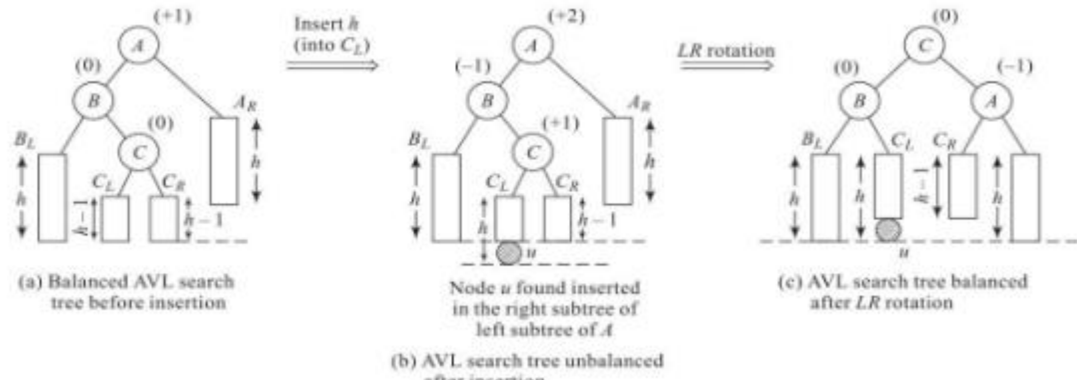
Figure below illustrates the generic representation of an LR type of imbalance and the corresponding rotation that is undertaken to set right the imbalance.

Here the node u on insertion finds A to be its closest ancestor node that is unbalanced and with reference to node A is inserted in the right subtree of left subtree of A.

This therefore is an LR type of imbalance and calls for LR rotation to balance the tree. The AVL tree before insertion of u (Fig.(a)), the unbalanced tree after insertion of u (Fig.(b)) and the balanced tree after the LR rotation (Fig. (c)) have been illustrated.

In the case of LR rotation, the following observations hold:

i) If BF(C) = 0 after insertion of new node then BF(A)=BF(B)=0 after rotation
ii) If BF(C) = 1 after insertion of new node then BF(A)= 0, BF(B)=+1 after rotation
iii) If BF(C) = +1 after insertion of new node then BF(A)=1, BF(B)=0 after rotation3

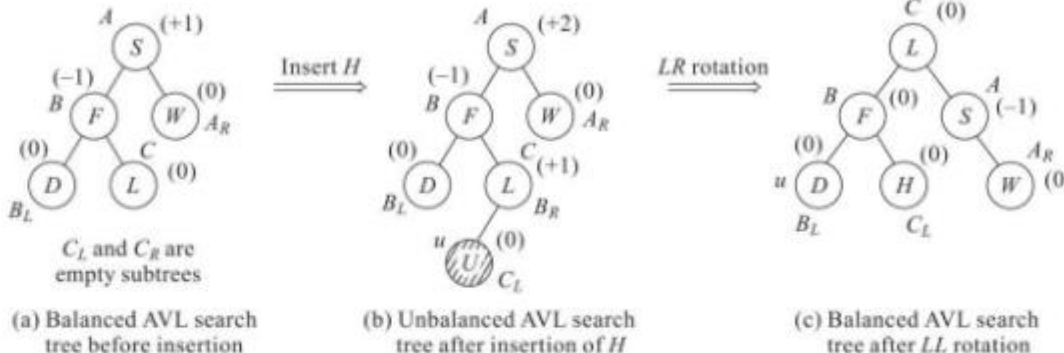


(a) Balanced AVL search tree before insertion

(b) AVL search tree unbalanced after insertion

(c) AVL search tree balanced after LR rotation

Example:

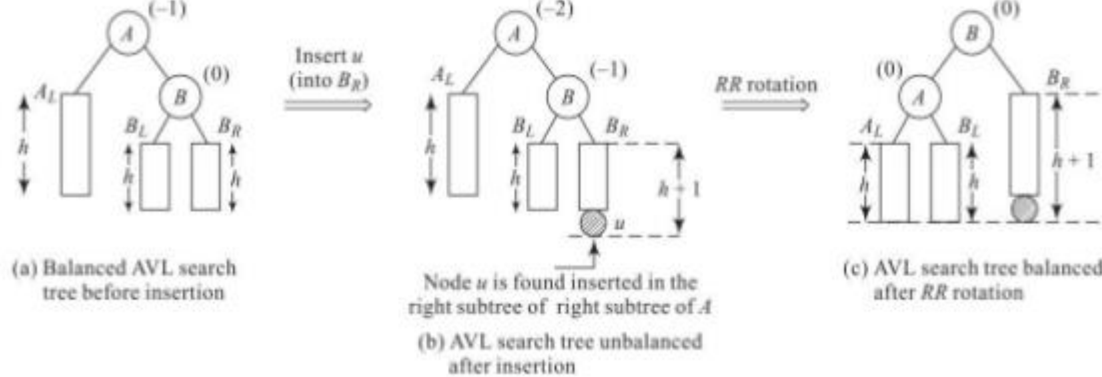


(a) Balanced AVL search tree before insertion (b) Unbalanced AVL search tree after insertion of *H* (c) Balanced AVL search tree after *LL* rotation

RR rotation:

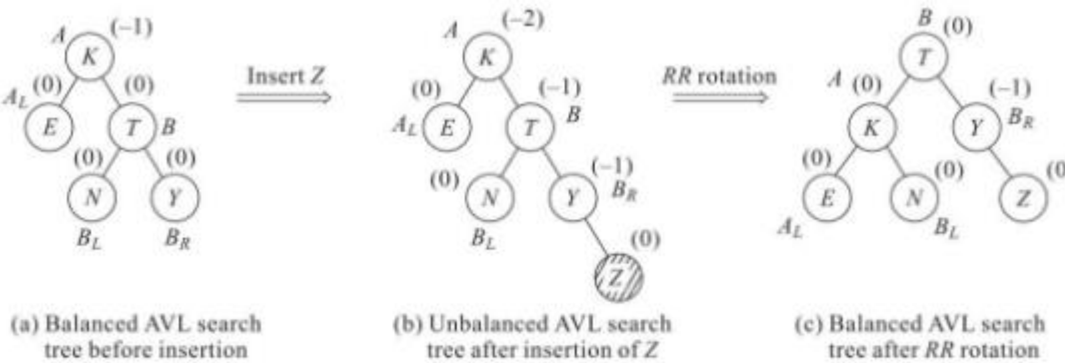
The RR rotation is symmetric to the LL rotation. Figure below illustrates the generic representation of the RR rotation scheme.

Observe how node u finds itself inserted in the right subtree of right subtree of A, the closest ancestor node that is unbalanced and the rotation is merely a mirror image of the LL rotation scheme.



(a) Balanced AVL search tree before insertion (b) AVL search tree unbalanced after insertion (c) AVL search tree balanced after *RR* rotation

Example: Consider the AVL search tree shown in Fig. 10.15. The insertion of Z calls for an RR rotation. The unbalanced AVL search tree and the balanced tree after RR rotation have been shown in Fig below.



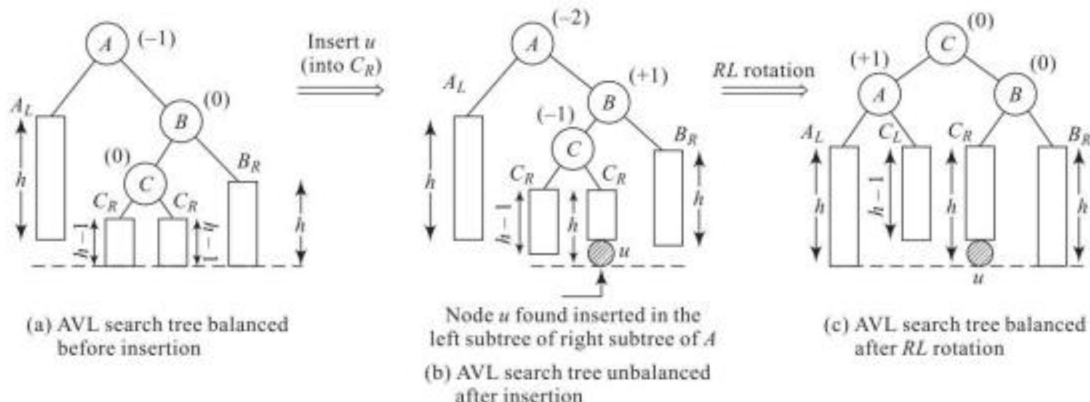
(a) Balanced AVL search tree before insertion (b) Unbalanced AVL search tree after insertion of *Z* (c) Balanced AVL search tree after *RR* rotation

RL rotation:

RL rotation is symmetric to LR rotation. Figure below illustrates the generic representation of the RL rotation scheme.

Here node u finds itself inserted in the left subtree of right subtree of node A which is the closest ancestor node that is unbalanced.

As pointed out for the LR rotation scheme, the rotation procedure for RL remains the same irrespective of u being inserted in CL or CR, the left subtree and right subtree of C respectively.



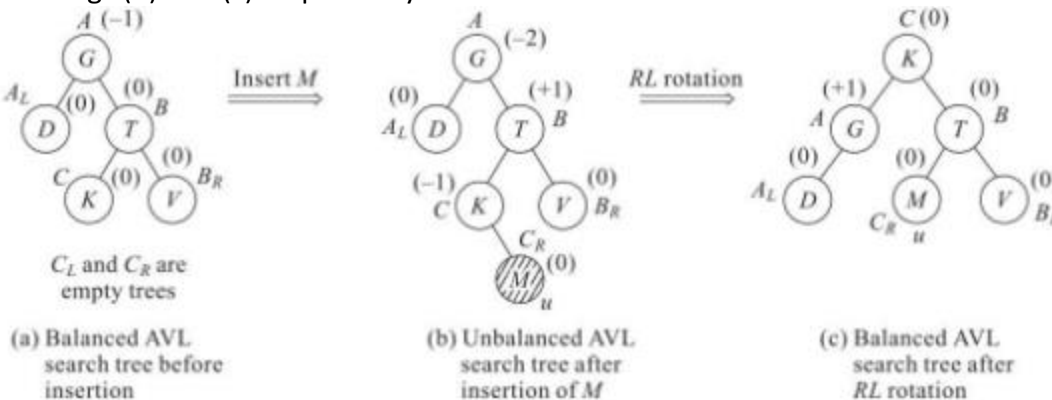
(a) AVL search tree balanced before insertion

Node u found inserted in the left subtree of right subtree of A

(b) AVL search tree unbalanced after insertion

(c) AVL search tree balanced after RL rotation

Example: Consider the AVL search tree shown in Fig. (a). The insertion of M calls for an RL rotation. The unbalanced AVL search tree and the balanced tree after RL rotation have been shown in Figs (b) and (c) respectively.



(a) Balanced AVL search tree before insertion

(b) Unbalanced AVL search tree after insertion of M

(c) Balanced AVL search tree after RL rotation

In the above classes of rotations, LL and RR are called as single rotations and LR and RL are called as double rotations.

An LR rotation is a combination of RR rotation followed by an LL rotation and RL rotation is a combination of LL rotation followed by an RR rotation.

Algorithm : Skeletal procedure to insert an ITEM into an AVL search tree T

procedure INSERT(T, ITEM)
call GETNODE(X);
DATA(X)=ITEM,
LCHILD(X)=RCHILD(X)=NIL and BF(X)=0;
if (T=NIL) then { Set T to X; exit; }

Find node P where ITEM is to be inserted as either the left child or right child of P by following a path from the root onwards. Also, while traversing down the tree in search of the

point of insertion of ITEM, take note of the most recent ancestor node A whose BF(A)= ±1; Insert node X carrying ITEM as the left or right child of node P;

```
if (node A not found) then
{TEMP = T;
while ( TEMP <> X) do
if (DATA(X) > DATA(TEMP))
```

Deletion from an AVL search tree:

To delete an element from an AVL search tree we discuss the operation based on whether thenode t carrying the element to be deleted is either a leaf node or one with a single non empty subtree or with two non emptysubtrees.

A delete operation just like an insert operation may also imbalance an AVL search tree. Just as LL/ LR/ RL/ RR rotations are called for to rebalance the tree after insertion, a delete operation also calls for rotations categorized as L and R. While the Lcategory is further classified as L0,L1 and L-1 rotations, the R category is further classified as R0, R1 and R-1 rotations.

Update balance factors:

Rule 1: With regard to node p, if node t'sdeletion occurred in its right subtree then bf(p) increases by 1 and if it occurred in its left subtree then bf(p) decreases by 1.

Rule 2: If the new bf(p)=0 then the height of the tree is decreased by1 and therefore this calls for updating the balance factors of its parent node and/or its ancestor nodes.

Rule 3: If the new bf(p) = ±1, then the height of the tree is the sameas it was before deletion and therefore the balance factors of the ancestor nodes remains unchanged.

Rule 4: If the new bf(p)= ±2, then the node p is unbalanced and the appropriate rotations need to be called for.

Classify rotations for deletion:

For the R classification, if bf(A)=+2 then it should have been +1 beforedeletion and A should have a left subtree with root B. Based on bf(B) being either 0 or +1 or –1, classify the R rotations further as R0, R1 and R- 1 respectively.

For the L classification, if bf(A)= -2 then it should have been -1 before deletion and A should have a right subtree with root B. Based on bf(B) being either 0 or +1 or -1, classify the L rotations further as L0, L1 and L-1 respectively.

Rotation free deletion of a leaf node:

In the case of deletion of node t which is a leafnode, we physically delete the node and make the child link of its parent, viz., node p null. Now update the balance factor of node p based on whether the deletion occurred to its right or left.

If it had occurred in the right then we increase bf(node p) by 1 or else decrease bf(node p) by 1. The new updated value of bf(node p) is now tested against Rules 1- 4 for updating the balance factors of its ancestor nodes.

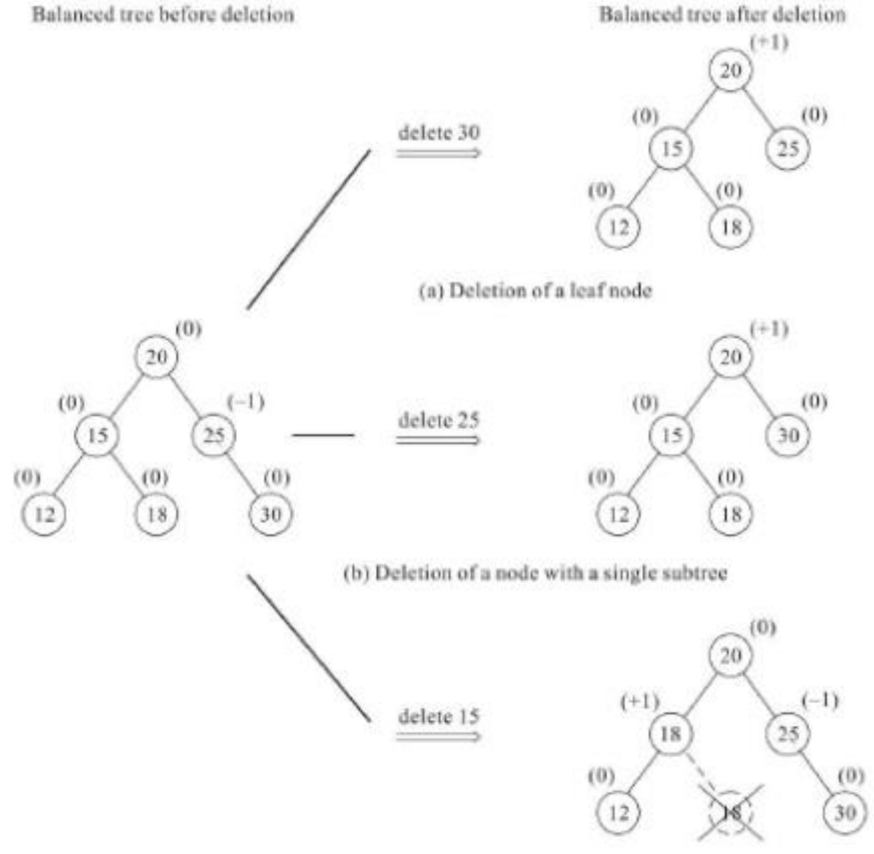
Rotation free deletion of a node having a single subtree:

In the case of deletion ofnode t with a single subtree, just as before, we reset the child link of the parent node, node p, to point to the child node of node t. The balance factors of node p and / or its ancestor nodes are updated using Rules 1-4.

Rotation free deletion of a node having both the subtrees:

In the case of deletionof node t which has both its subtrees to be non empty, the deletion is a little more involved. We first replace DATA (node t) with the smallest key of the right subtreeofnodet or with the largest key of the left subtree of node t.

The smallest key of the right subtree of node t can be obtained by moving right and then moving deep left until the left child link is NIL. Similarly, moving left and then moving deep right until an empty right child link is seen will yield the largest element in the left subtree of node t.



<u>**R category rotations associated with the delete operation:**</u>

<u>**R0 rotation:**</u>

Figure below illustrates the generic representation of an R0 rotation. Node t is to be deleted from a balanced tree with A shown as the root (for simplicity) and with A_R as its right subtree.

B is the root of A's left subtree and has two subtrees B_L and B_R. The heights of the subtrees are as shown in the figure. Now the deletion of node t results in an imbalance with bf(A)=+2.

Since deletion of node t occurred to the right of A and since A is the first node on the path to the root, the situation calls for an R rotation with respect to A. Again since bf(B)=0, the rebalancing needs to be brought about using an R0 rotation only.

Here, B pushes itself up to occupy A's position pushing node A to its right along with A_R. While B retains B_L as its left subtree, B_R is handed over to node A to function as its left subtree.

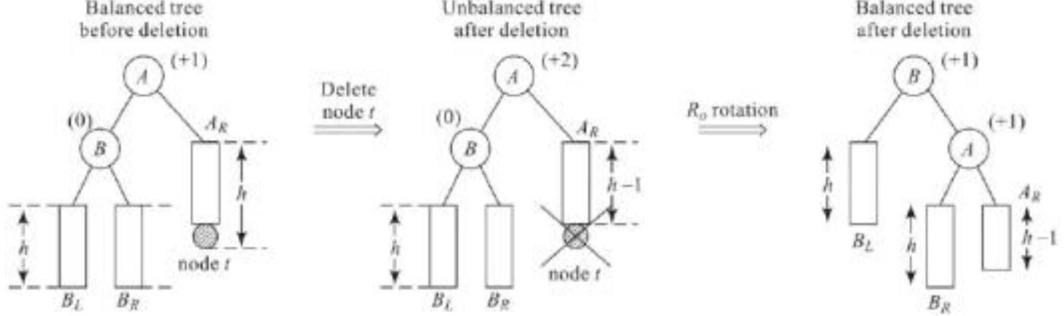


Fig. Generic representation of an R0 rotation

Example:

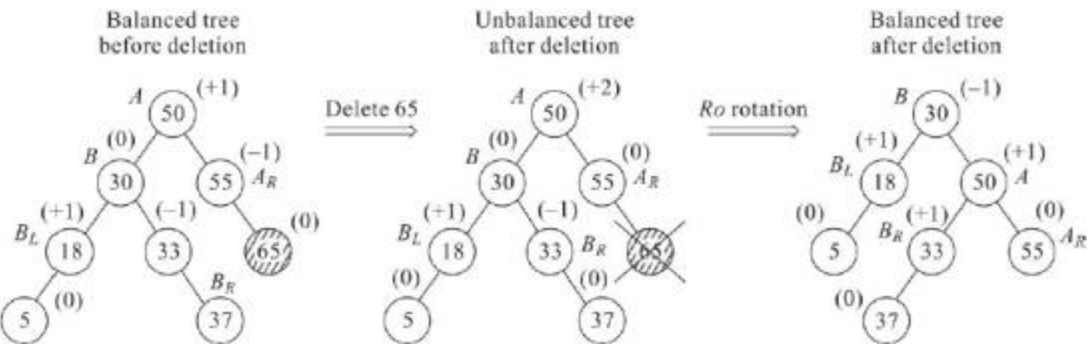


Fig. Deletion of a node calling for R0 rotation

R1 Rotation:

Figure below illustrates the generic representation of an R1 rotation. Deletion of node t occurs to the right of A the first ancestor node whose bf(A)=+2. But bf(B)=+1 classifies it further as R1 rotation. The rotation is similar to the R0 rotation and yields a balanced tree.

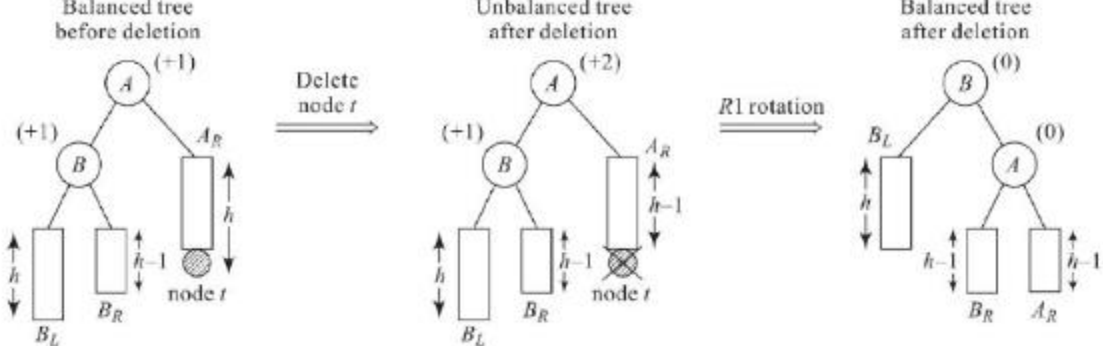


Fig. Generic representation of an R1 rotation

Example:

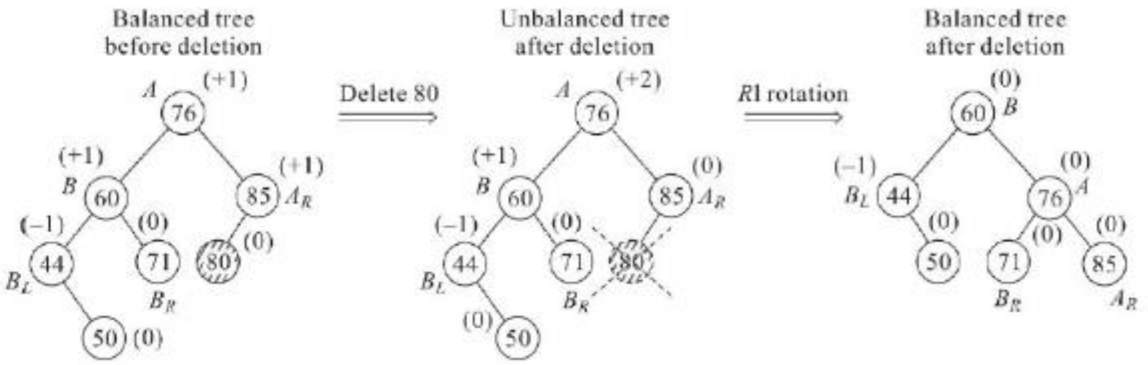


Fig. Deletion of a node calling for R1 rotation

R-1 rotation:

The generic representation of an R-1 rotation is shown in Fig. below. As in the other rotations deletion of node t results in the imbalance of the tree with regard to A and also leaves bf(B)= -1 calling for R-1 rotation.

Here let C be the root of the right subtree of B and C_L and C_R its left and right subtrees respectively. During the rotation C elevates itself to become the root pushing A along with its right subtree A_R to its right.

The tree is now rearranged with C_L as the right subtree of B and C_R as the left subtree of A. The tree automatically gets balanced.

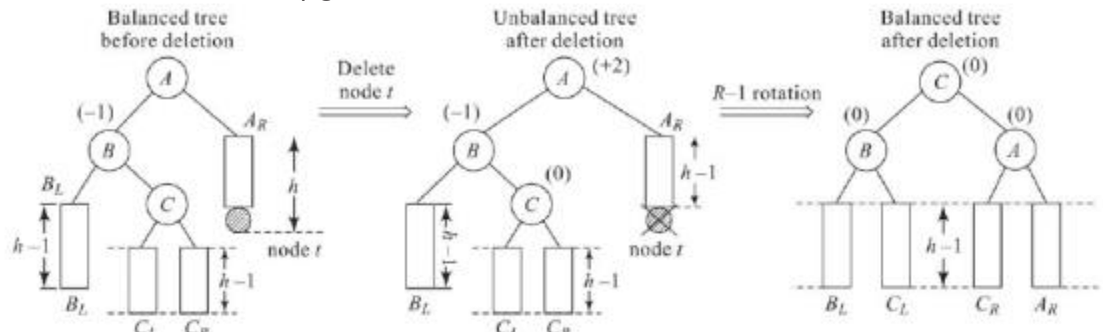


Fig. Generic representation of an R–1 rotation

Example:

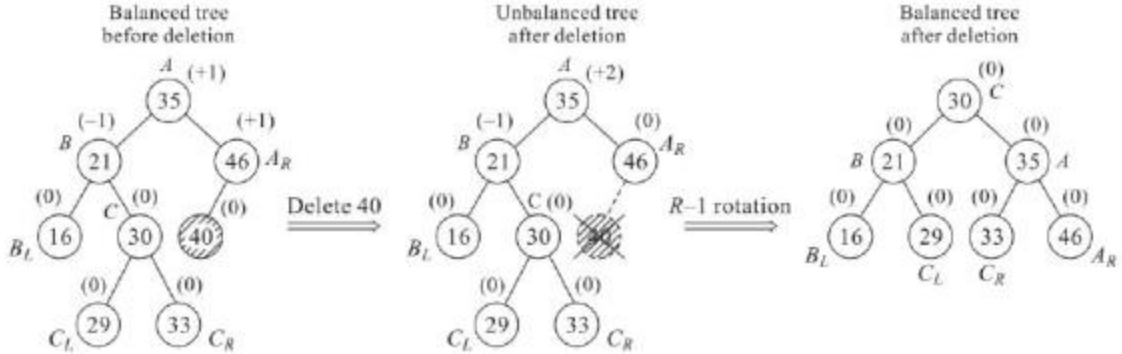


Fig. Deletion of a node calling for R–1 rotation

<u>L category rotations associated with the delete operation:</u>

If the deletion of node t occurs to the left of A, the first ancestor node on the path to the root reporting bf(A)= -2, then the category of rotation to be applied is L.

As in R rotations, based on bf(B)= +1, -1, or 0 the L rotation is further classified as L1, L-1 and L0 respectively.

The generic representations of the L0, L1 and L-1 rotations are shown in Fig.below.

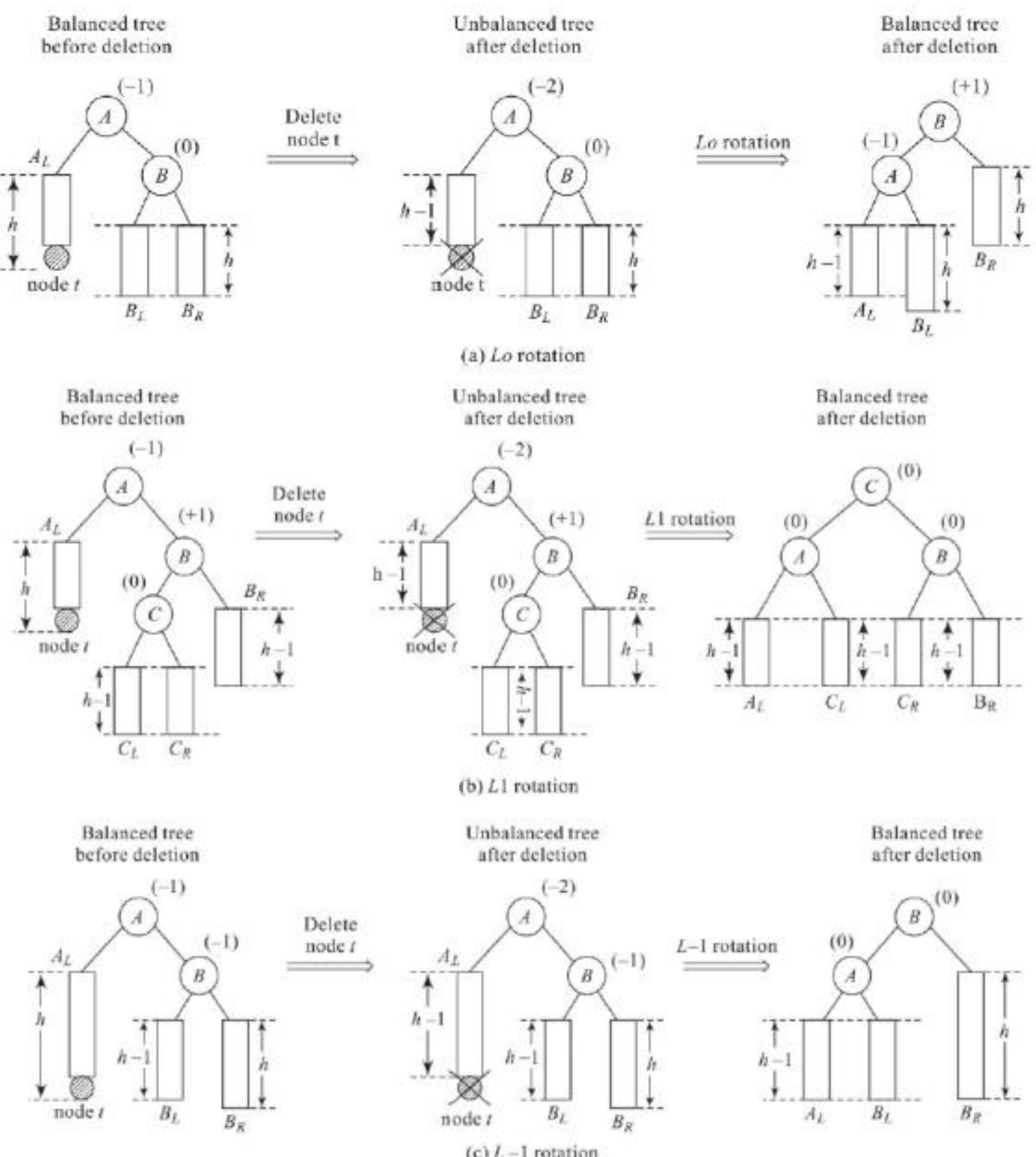


Fig. Generic representations of L0 L1 and L–1 rotations

RED-BLACK TREES:

Definition

A red-black tree is an extended binary search tree in which the nodes and the edges from which these nodes emanate are either red or black and satisfy the following properties:

(i) The root node and the external nodes are always black nodes.

(ii) [Red Condition] No two red nodes can occur consecutively on the path from the root node to an external node.

(iii) [Black Condition] The number of black nodes on the path from the root node to an external node must be the same for all external nodes.

Example

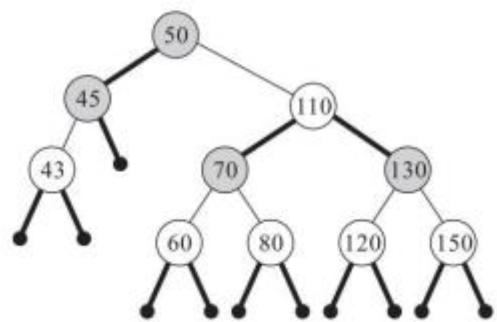


Fig. An example red-black tree

Representation of a red-black tree :

Since a red-black tree is an extended binary search tree, the kind of node representation used for a binary search tree may be employed for the tree as well.

However since the color of a node plays a dominant role in the definition of the red-black tree it is essential that the color is also recorded in the node structure as a field (COLOUR).

Searching a red-black tree :

Searching a red-black tree for a key is in no way different from the procedure used to search for a key in a binary search tree.

Inserting into a red-black tree:

Inserting a key K into a red-black tree follows a procedure exactly similar to the one employed for binary search trees. The only concern now is to determine the colour to which the node must be set to.

If the node is set to black, then the path from the root node to the external node passing through the node would have one more black node. This results in the violation of the Black Condition of a red-black tree.

Hence the other alternative is to set the node to red. Now, if doing so leads to the violation of the Red Condition, then the red-black tree is said to be unbalanced.

To set right the imbalance we need to undertake rotations. Let us suppose u is the newly inserted red node and parent_u its consecutive red node which is also the parent of node u.

Now, u must have a grand parent, grandparent_u which is a black node. Based on the position of node u in relation to parent_u and grandparent_u, and the colour of the other child of grandparent_u, the imbalances are classified as LLb, LLr, RLb, RLr, LRb, LRr, RRb and RRr.

Thus if u is inserted as the Left child of parent_u(L) which in turn is the Left child of grandparent_u (L) and the other child of grandparent_u is black (b) -the child may in fact be an external node which is black- then the rotation undertaken is LLb.

Imbalances of the type LLr, RLr, LRr, and RRr with ‘ r’ as its suffix only call for a colour change of the nodes to set right the imbalance. On the other hand imbalances of the type LLb, LRb, RRb and RLb, with ‘ b’ as its suffix, call for rotations to set right the imbalance.

LLr, LRr, RRr, and RLr imbalances:

Figure below illustrates a generic representation of the LLr, LRr, RRr, and RLr imbalances and the colour changes that need to be undertaken to set right the imbalance.

The notations L, R and r inscribed on the edges of the red-black trees illustrate the classification of the imbalance.

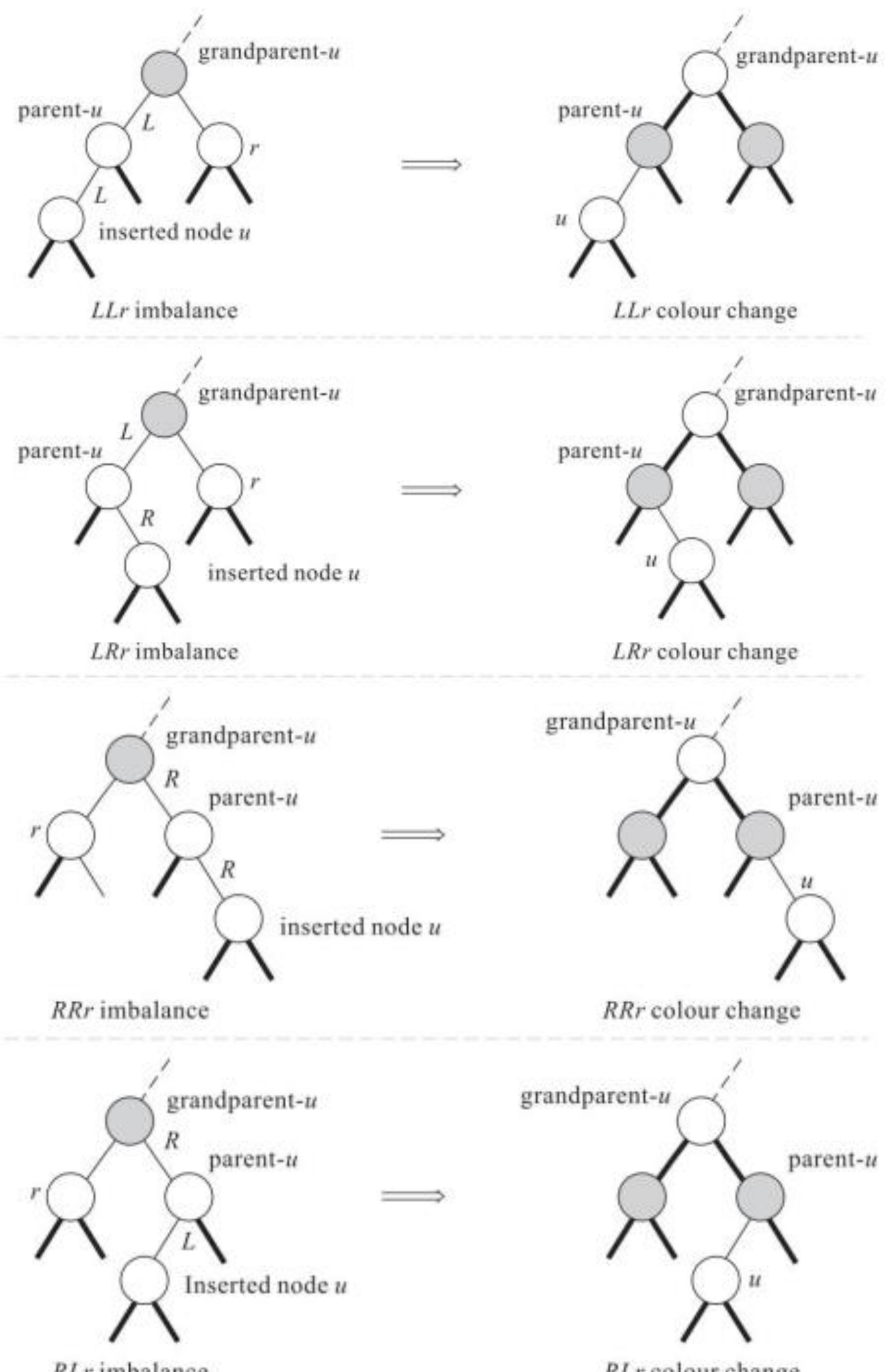


Fig. Generic representations of LLr, LRr, RRr, and RLr imbalances and their colour change

Example:

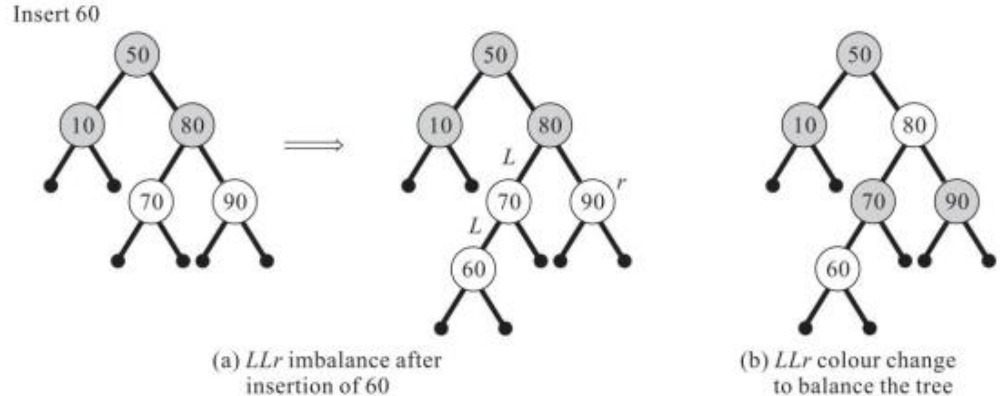


Fig. 12.6 *An example LLr imbalance and its colour change*

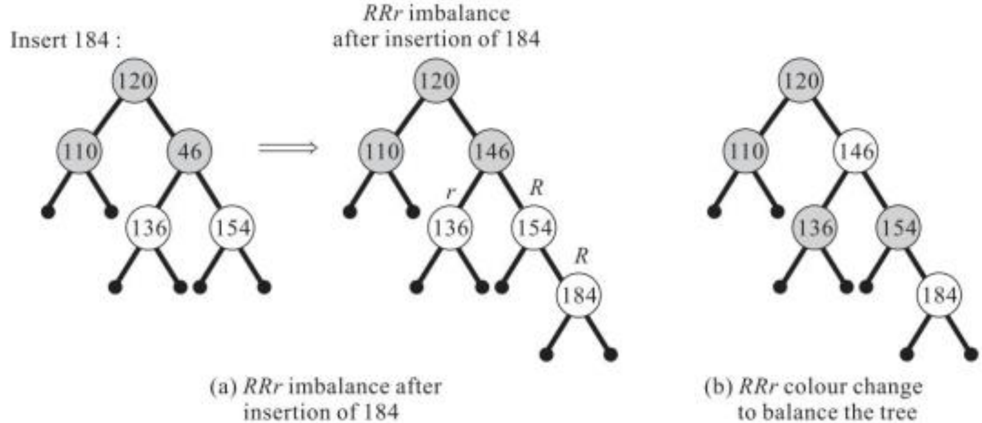


Fig. An example RRr imbalance and its colour change

LLb, LRb, RRb, and RLb imbalances:

Figure illustrates the generic representations of the LLb, LRb, RRb, and RLb imbalances and the respective rotations to rebalance the red-black tree. The notations L, R and b inscribed on the edges of the red-black trees illustrate the classification of the imbalance.

grandparent-u L b grandparent-u^R parent-u L parent-u^R u u^L u^R

LLb imbalance

parent-u grandparent-u u^L u^R parent-u^R grandparent-u^R

After *LLb* rotation

LRb imbalance

After LRb rotation

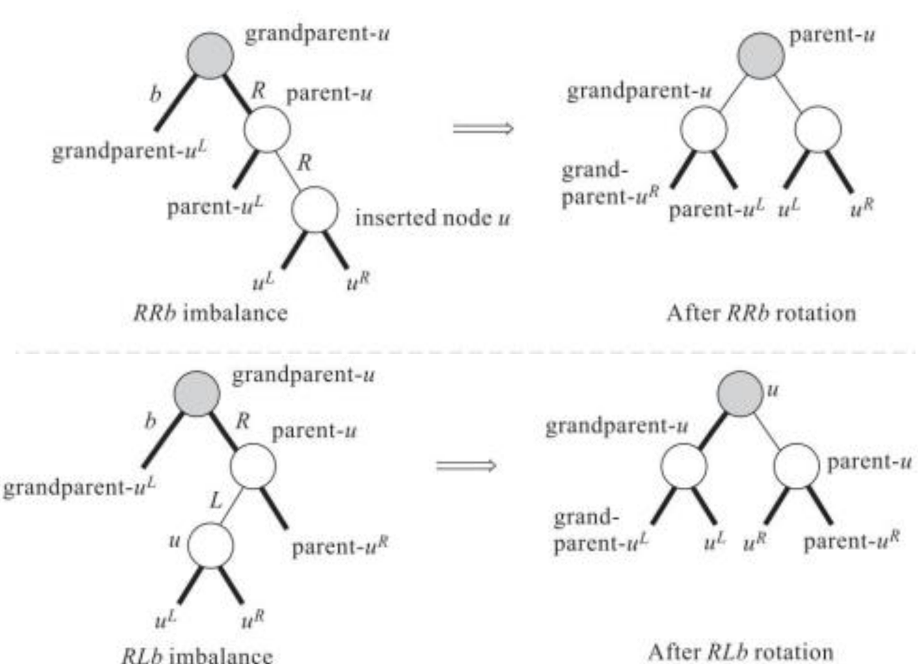
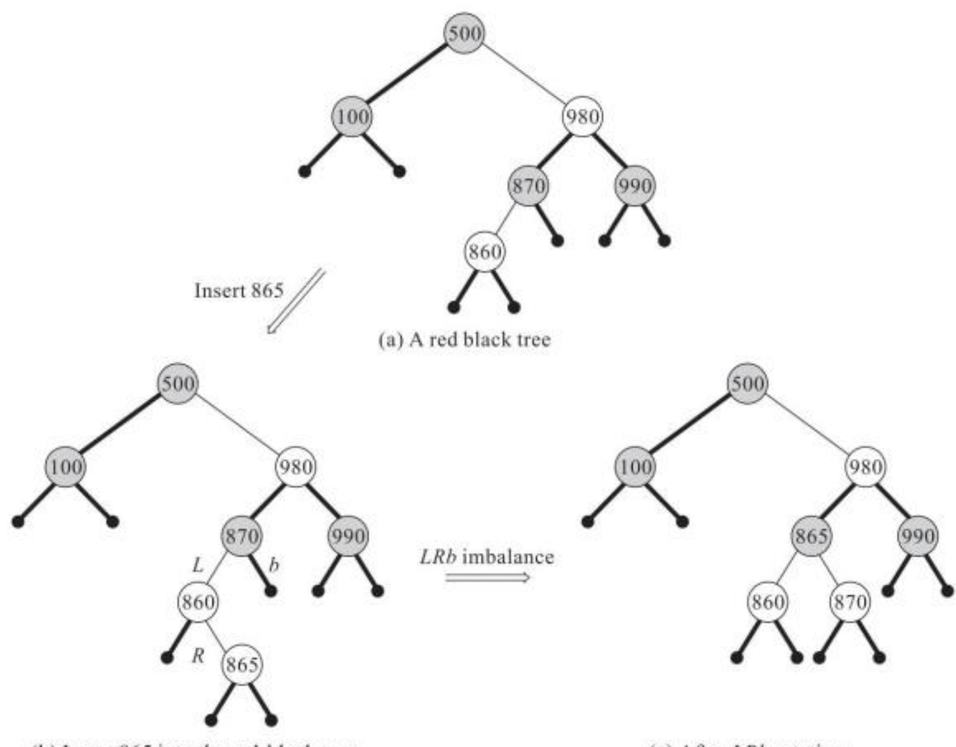


Fig. Generic representations of LLb, LRb, RRb and RLb imbalances and their rotations

Example:



(a) A red black tree

(b) Insert 865 into the red-black tree

(c) After *LRb* rotation

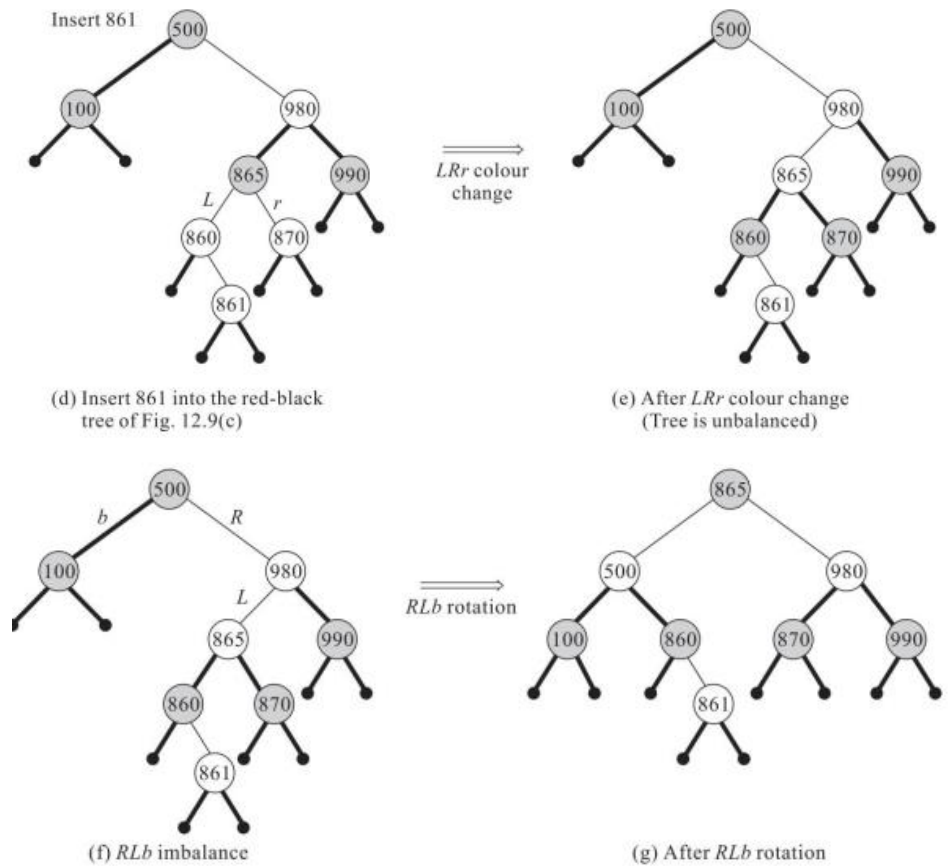


Fig. An example LRb, RLb imbalance

Deleting from a red-black tree:

Deleting a key K from a red-black tree proceeds as one would to delete the same from a binary search tree, when K is a leaf node or K has a lone subtree (left subtree or right subtree only) or K has both left subtree and right subtree.

However, if the deletion results in an imbalance in the tree then this may call for a colour change or a rotation if necessary. If the deleted node were red, then there is no way that the Black Condition would be violated and hence no imbalance in possible.

On the other hand if the deleted node were to be black then there is every possibility of violation of the Black Condition due to the shortage of a black node in a specific root-to-external node path.

In such a case the tree is said to be unbalanced. The imbalance is classified as Left (L) or Right (R) based on whether the deleted node v, occurs to the right or left of its parent node, parent_v.

Again if the sibling of node v, sibling_v is a black node then the imbalance is further classified as Lb or Rb. If sibling_v is a red node, then the imbalance is classified as Lr or Rr. Based on whether sibling_v has 0 or 1 or 2 red children the Lb, Rb imbalances are further sub classified as Lb0, Lb1 and Lb2, and Rb0, Rb1 and Rb2 respectively.

Similarly, the Lr, Rr imbalances are also sub classified as Lr0, Lr1 and Lr2, and Rr0, Rr1 and Rr2 respectively. During rebalancing, v denotes the node that was deleted but physically replaced by another node which takes its place as called for by the delete process.

Rb0, Rb1 and Rb2 imbalances:

Figure below illustrates the generic representations of Rb0, Rb1 and Rb2 imbalances. The

notations R, b and 0/1/2 inscribed on the edges of the red-black trees illustrate the classification of the imbalance. In the case of Rb0 imbalance the rebalancing only calls for a colour change of nodes.

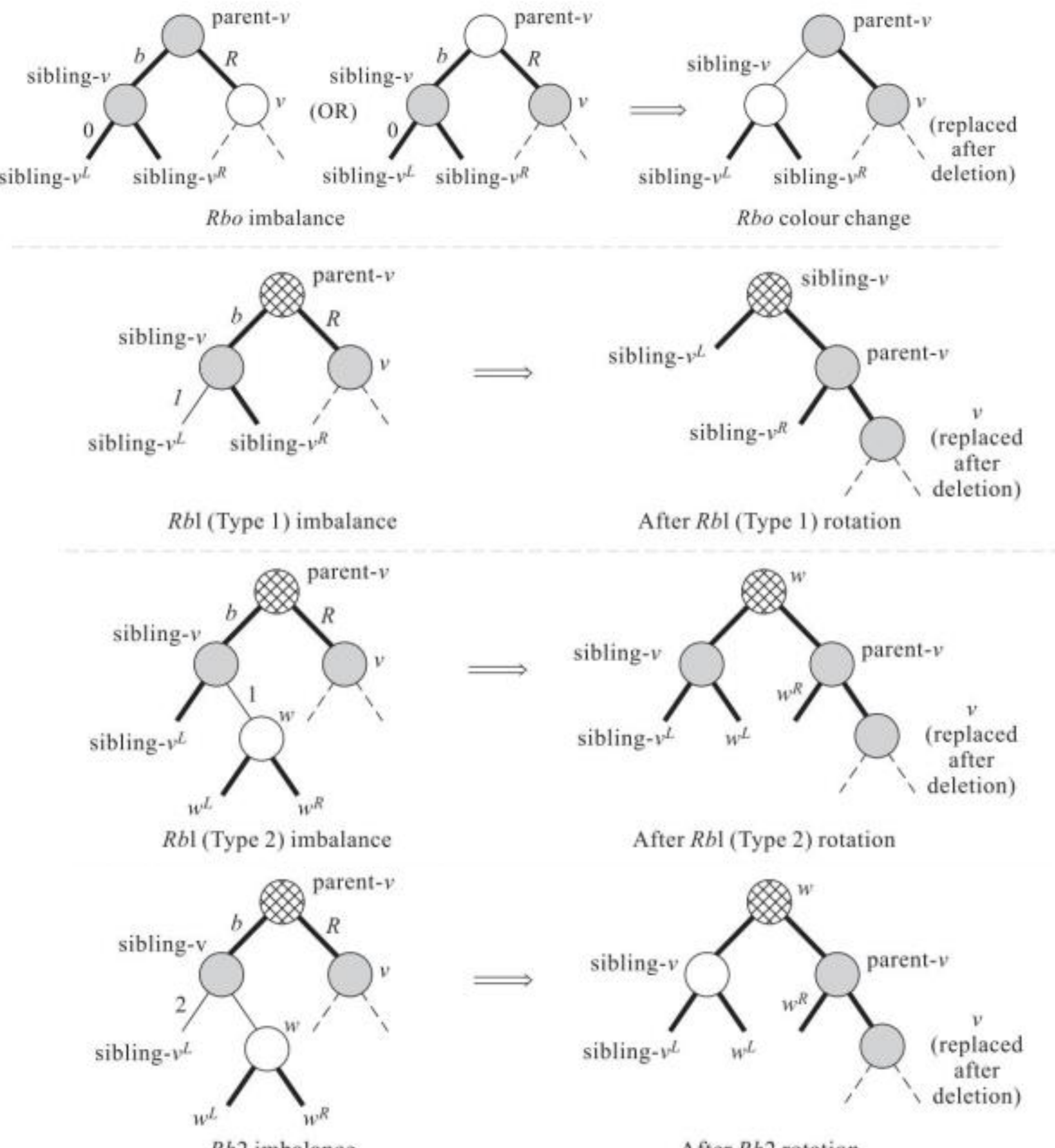
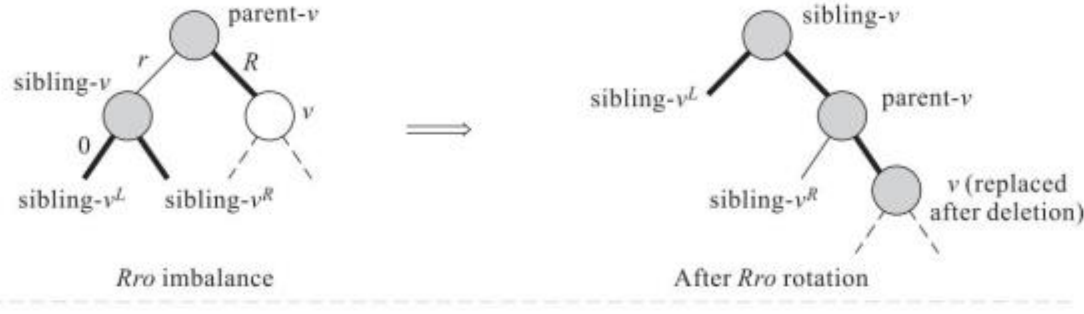


Fig. Generic representations of Rb0, Rb1 and Rb2 imbalances and their rebalancing mechanisms

Rr0, Rr1 and Rr2 imbalances:

Figure below illustrates the generic representations of Rr0, Rr1 and Rr2 imbalances. The notations R, r and 0/1/2 inscribed on the edges of the red-black tree illustrate the classification of imbalance. Rotations are undertaken in all the three cases to rebalance the trees. Rr1 imbalance is of two types indicated as Rr1(type 1) and Rr1(type 2).



Rro imbalance

After *Rro* rotation

*Rr*1 (Type 1) imbalance

After *Rr*1 (Type 1) rotation

*Rr*1(Type 2) imbalance

After *Rr*1 (Type 2) rotation

*Rr*2 imbalance

After *Rr*2 rotation

Fig. Generic representations of Rr0, Rr1 and Rr2 imbalances and their rotations

Example:

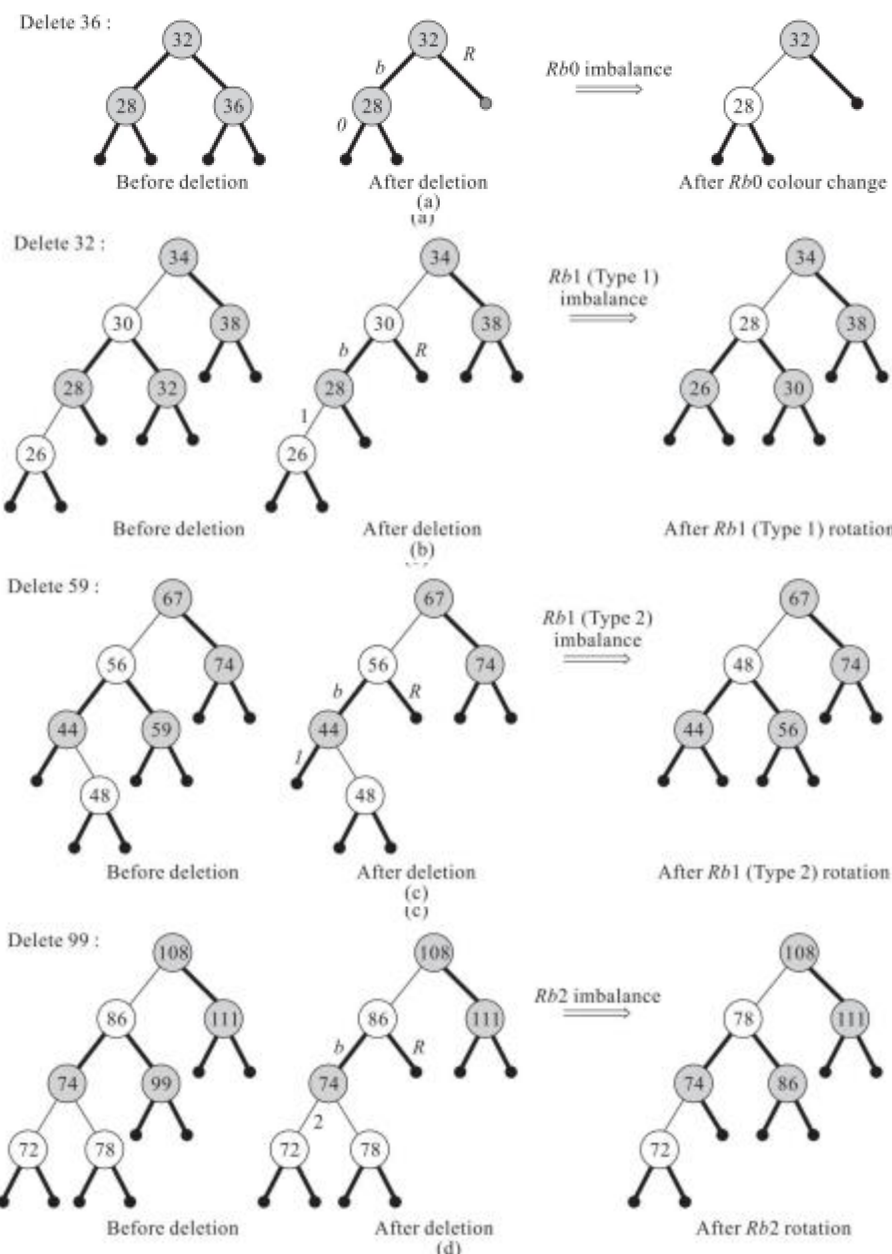


Fig. Example Rb0, Rb1 and Rb2 imbalances

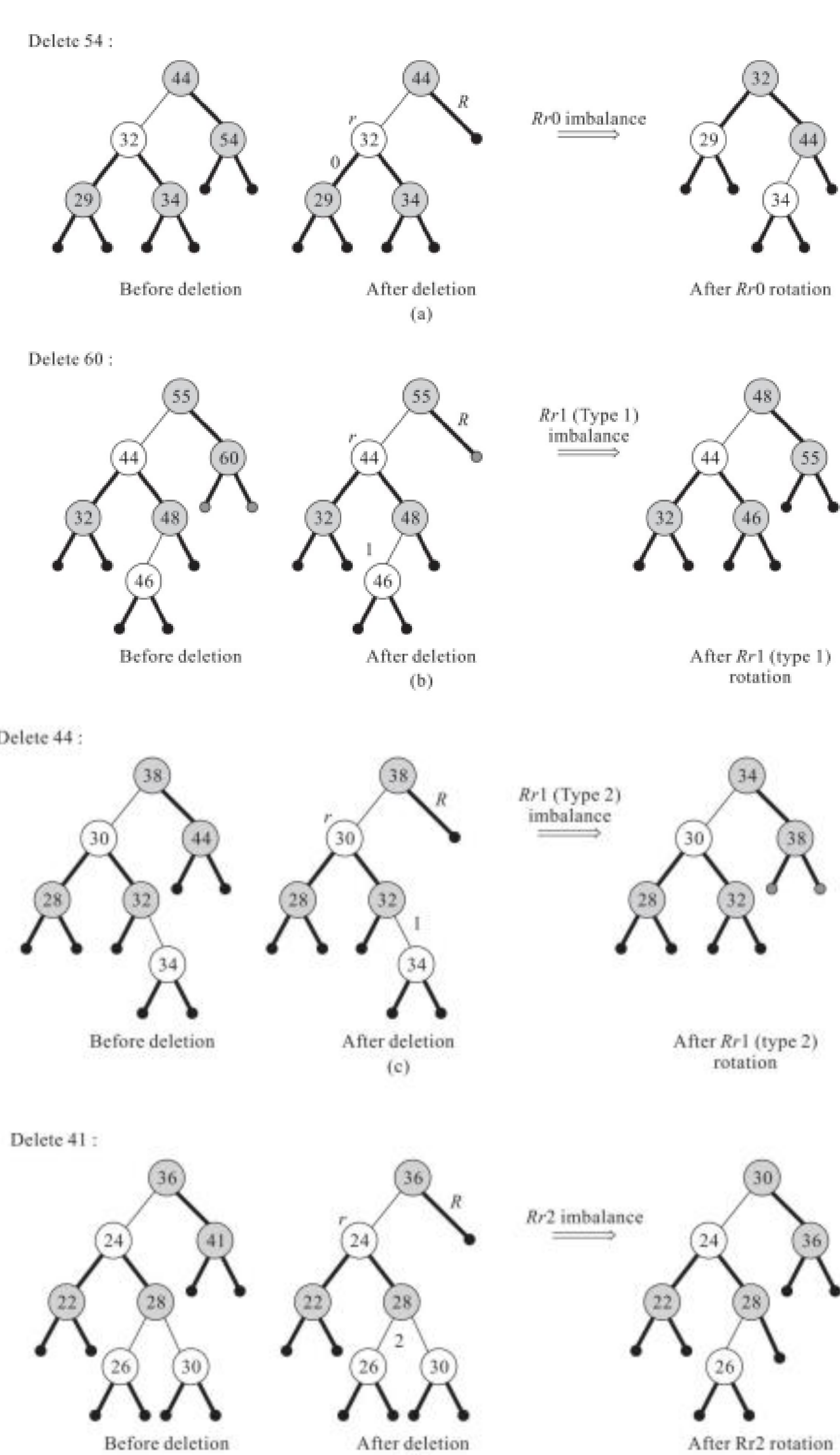


Fig. Example Rr0, Rr1 and Rr2 rotations

Lb0, Lb1 and Lb2 imbalances:

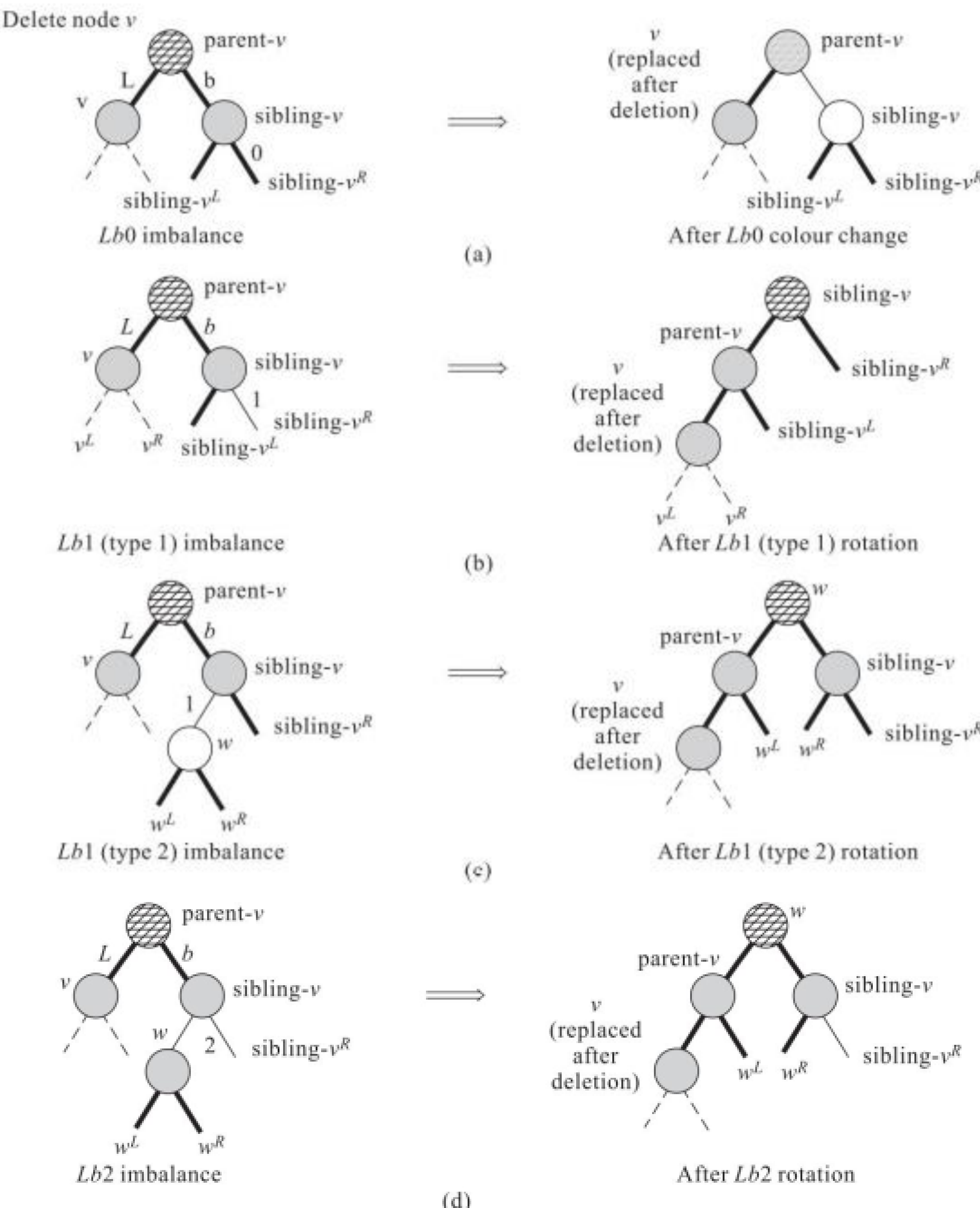


Fig. Generic representations of Lb0, Lb1 and Lb2 imbalances and their rebalancing mechanisms

<u>Lr0, Lr1 and Lr2 imbalances</u>:

*Lr*0 imbalance — After *Lr*0 rotation

(a)

*Lr*1 (type 1) imbalance — After *Lr*1 (type 1) rotation

(b)

*Lr*1 (type 2) imbalance — After *Lr*1 (type 2) rotation

(c)

*Lr*2 imbalance — After *Lr*2 rotation

(d)

Fig. Generic representations of Lr0, Lr1 and Lr2 imbalances and their rotations

SPLAY TREES:

Splay tree is another variant of a binary search tree. In a splay tree, recently accessed element is placed at the root of the tree. A splay tree is defined as follows...

Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree.

In a splay tree, every operation is performed at the root of the tree. All the operations in splay tree are involved with a common operation called **"Splaying"**.

Splaying an element is the process of bringing it to the root position by performing suitable rotation operations.

In a splay tree, splaying an element rearranges all the elements in the tree so that splayed element is placed at the root of the tree.

By splaying elements we bring more frequently used elements closer to the root of the tree so that any operation on those elements is performed quickly. That means the splaying operation automatically brings more frequently used elements closer to the root of the tree.

Every operation on splay tree performs the splaying operation. For example, the insertion operation first inserts the new element using the binary search tree insertion process, then the newly inserted element is splayed so that it is placed at the root of the tree.

The search operation in a splay tree is nothing but searching the element using binary search process and then splaying that searched element so that it is placed at the root of the tree.

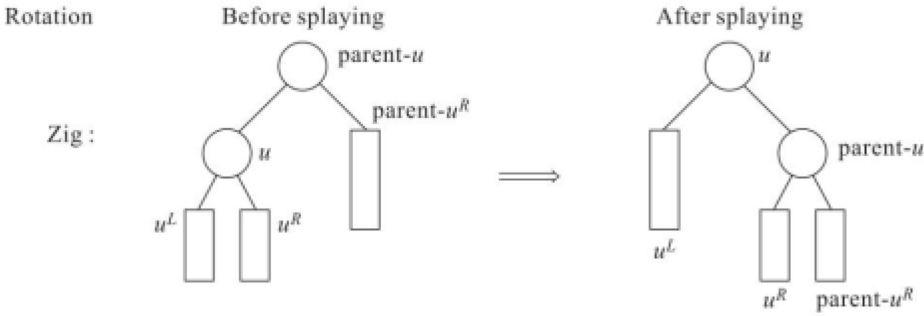
In splay tree, to splay any element we use the following rotation operations...

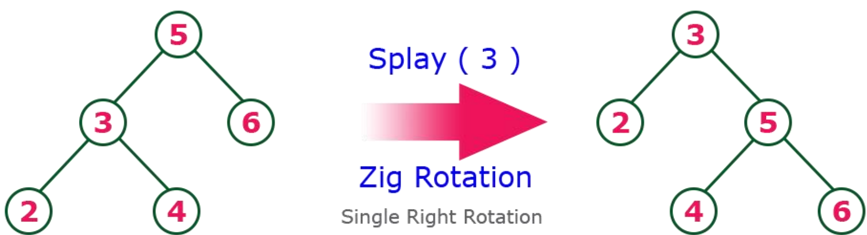
Rotations in Splay Tree

1. Zig Rotation
2. Zag Rotation
3. Zig - Zig Rotation
4. Zag - Zag Rotation
5. Zig - Zag Rotation
6. Zag - Zig Rotation

Zig Rotation

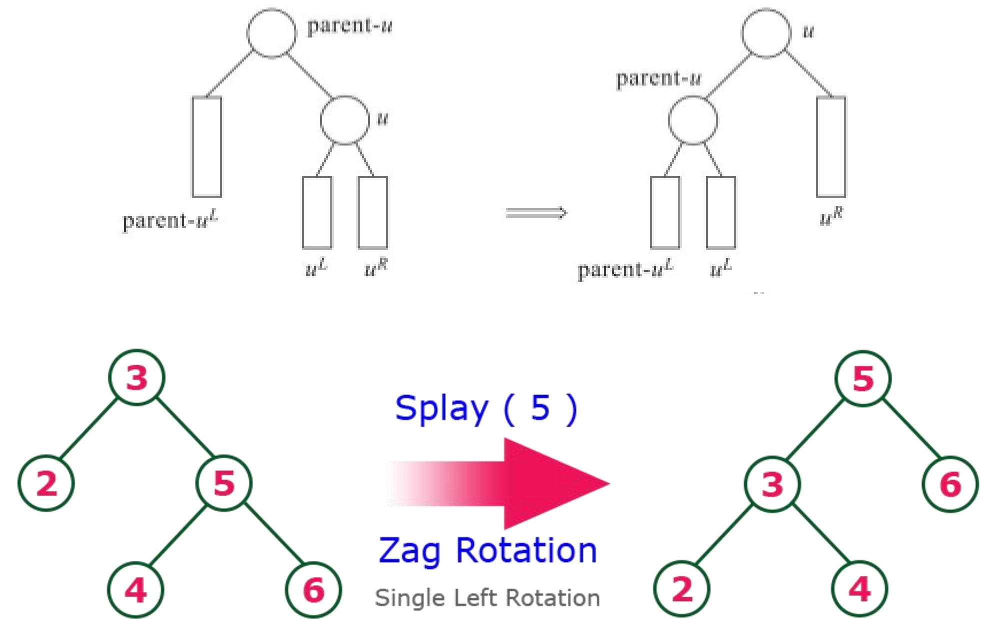
The **Zig Rotation** in splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position. Consider the following example...





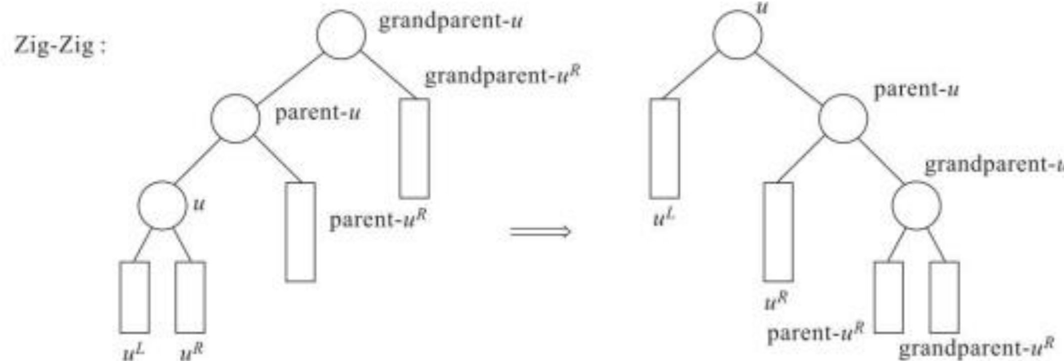
Zag Rotation

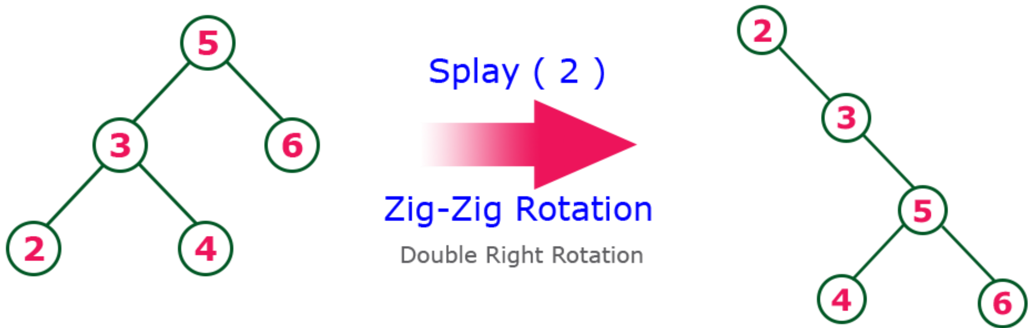
The **Zag Rotation** in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...



Zig-Zig Rotation

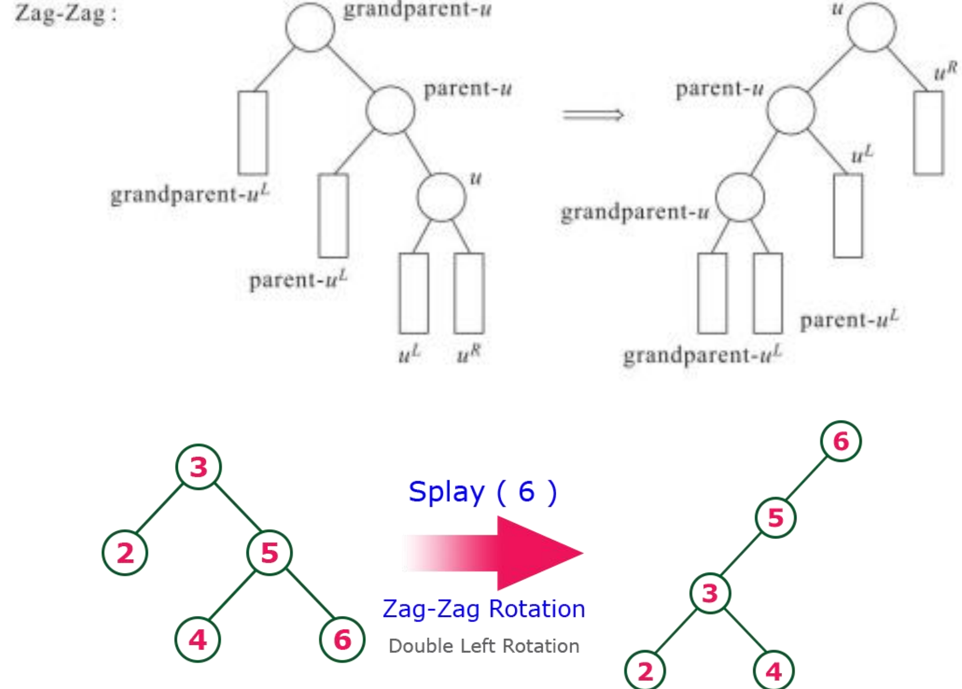
The **Zig-Zig Rotation** in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...





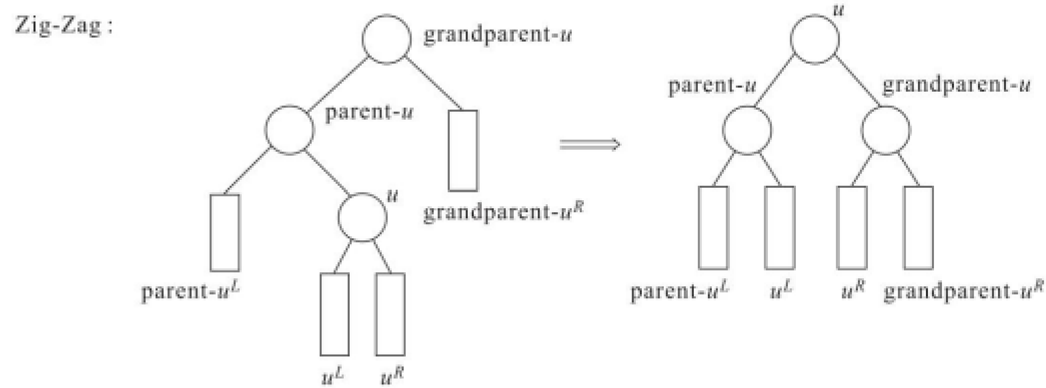
Zag-Zag Rotation

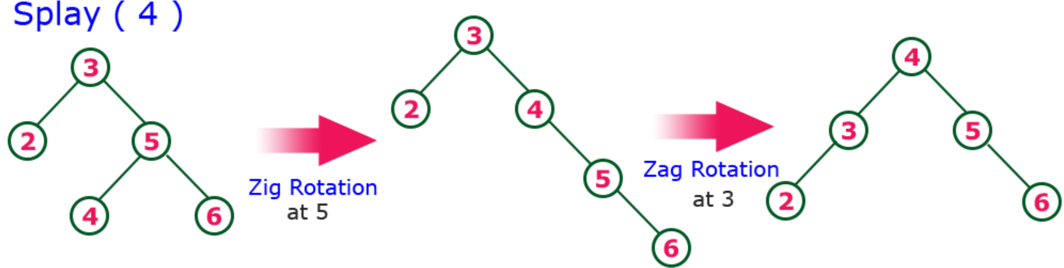
The **Zag-Zag Rotation** in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position. Consider the following example...



Zig-Zag Rotation

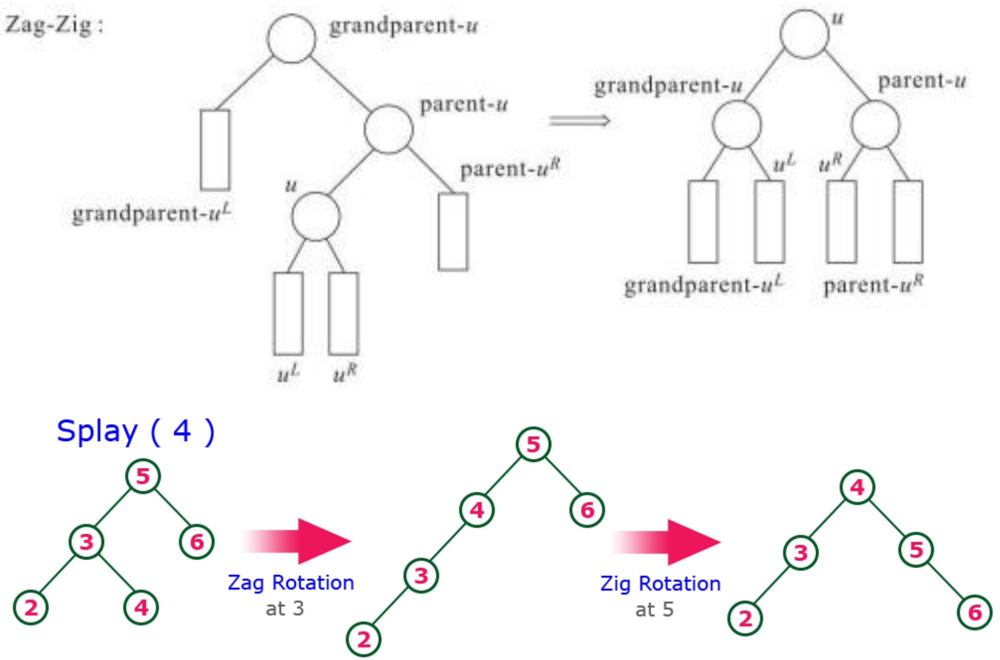
The **Zig-Zag Rotation** in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...





Zag-Zig Rotation

The **Zag-Zig Rotation** in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position. Consider the following example...



- **Every Splay tree must be a binary search tree but it is need not to be balanced tree.**

Insertion Operation in Splay Tree

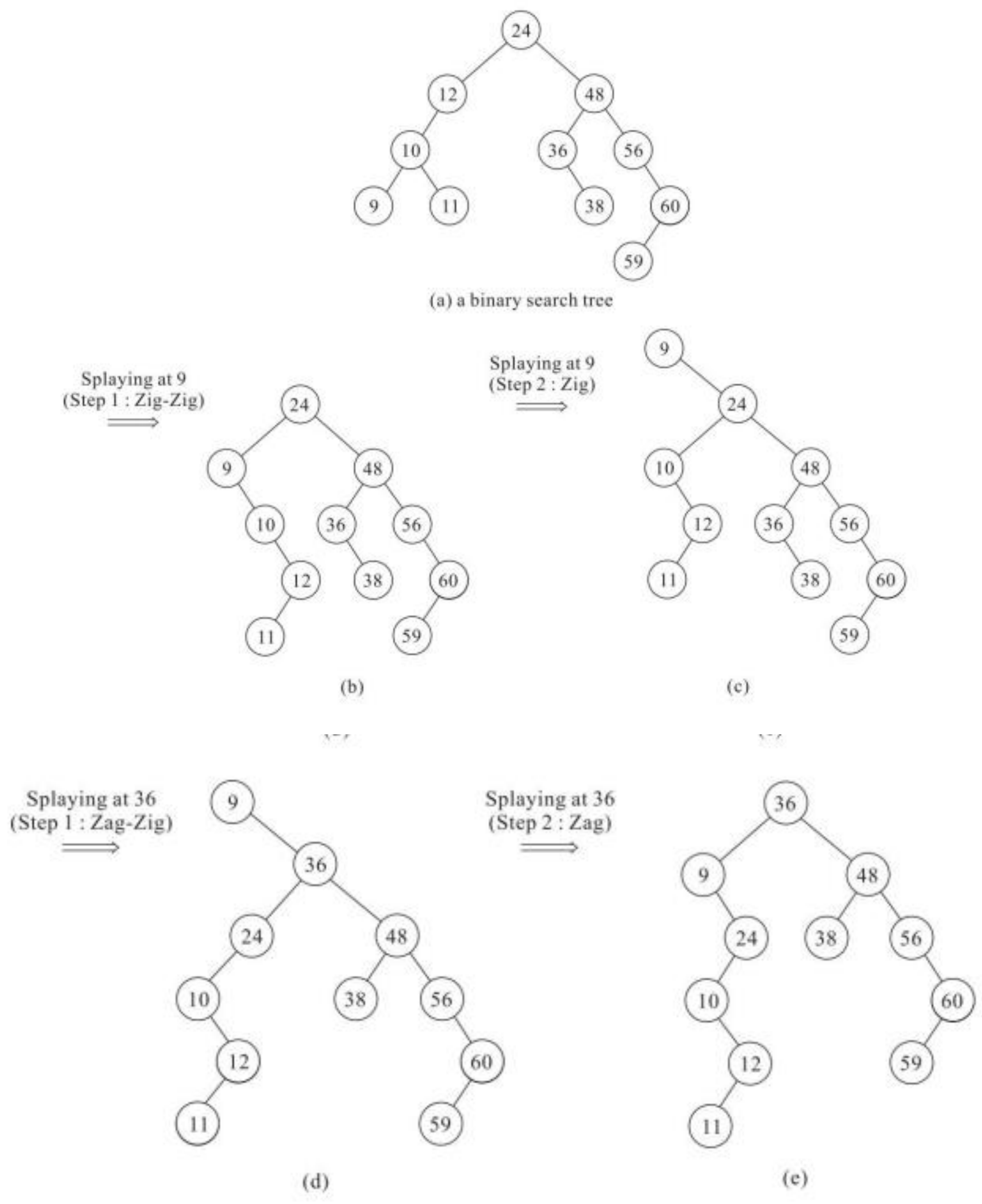
The insertion operation in Splay tree is performed using following steps...

- **Step 1 -** Check whether tree is Empty.
- **Step 2 -** If tree is Empty then insert the **newNode** as Root node and exit from the operation.
- **Step 3 -** If tree is not Empty then insert the newNode as leaf node using Binary Search tree insertion logic.
- **Step 4 -** After insertion, **Splay** the **newNode**

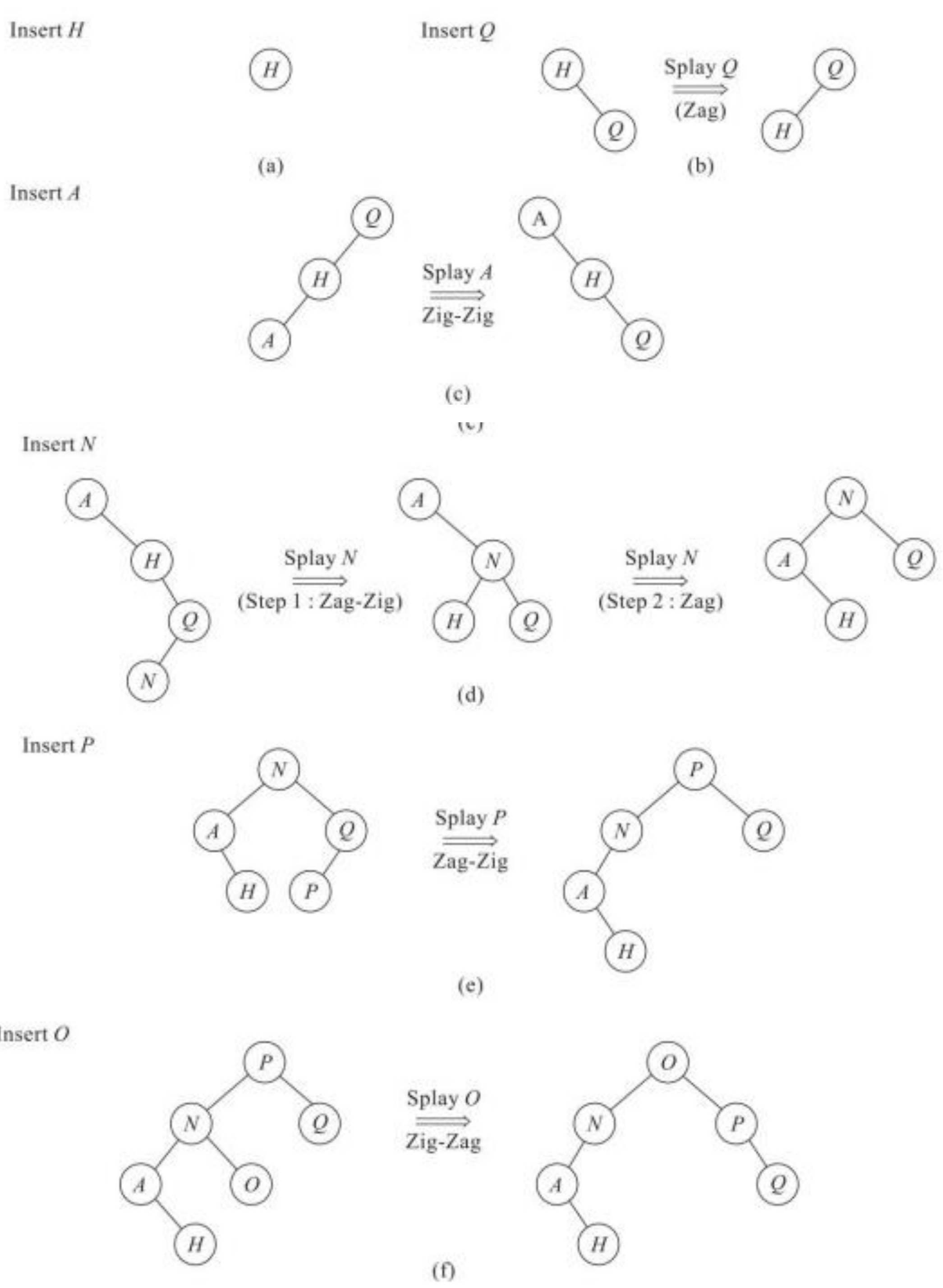
Deletion Operation in Splay Tree

The deletion operation in splay tree is similar to deletion operation in Binary Search Tree. But before deleting the element, we first need to **splay** that element and then delete it from the root position. Finally join the remaining tree using binary search tree logic.

Example: Consider the binary search tree shown in Fig. 12.19(a). Let us attempt splaying the tree at node 9.



Example: Build a splay tree inserting the following elements in the sequence shown: H, Q, A, N, P, O.



UNIT-VI

Indexed Sequential Access Method (ISAM):

Retrieval of records from large files or data bases stored in external memory is time consuming. To promote efficient retrievals, file indexes are maintained.

An index is a **<key, address>** pair. The purpose of indexing is to expedite the search process or retrieval of a record.

Indexed Sequential Access Method (ISAM) based files have been the foremost in using indexing for efficient retrievals. The records of the file are sequentially stored and for each block of records, the largest key and the block address is stored in an index.

To retrieve a record whose key is K, the index is first searched to obtain the address of the block and thereafter a sequential search of the block should yield the desired record.

Figure below illustrates an ISAM file structure. However if the file is too large, then index over indexes may have to be built.

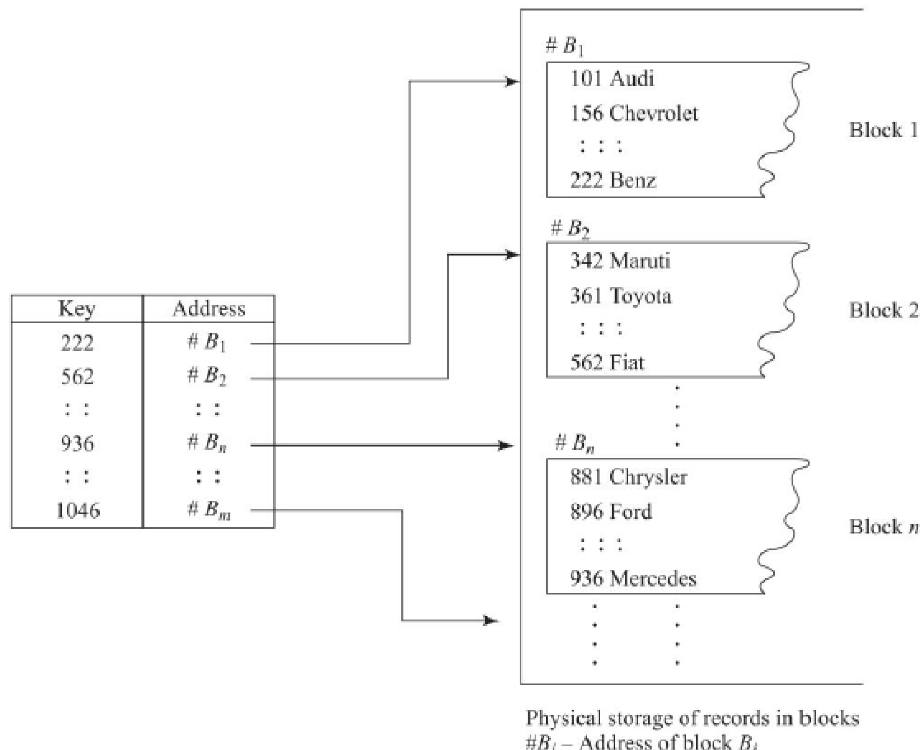


Fig. An ISAM file structure

Though indexes are basically look up tables, it is essential that they are represented using efficient data structures to expedite retrievals.

It is here that one finds the application of multi-way trees such as m-way search trees, B trees and tries.

B trees as file indexes: B trees are ideally suited for storing file indexes. Each internal node of the B tree stores the <key, address> pair.

Their balanced heights call for fewer node accesses during the retrievals. Once the key is found the address of the record is also accessed along with it thereby speeding up the retrieval process.

m-way search trees: Definition and Operations:

Each node of an m-way search tree can hold at most m branches. Thus, m-way search trees adopt multi way branching extending the above mentioned characteristic of binary search trees.

Definition:

An m-way search tree T may be an empty tree. If T is non-empty then the following properties must hold good:

(i) For some integer m, known as the order of the tree, each node has at most m child nodes. In other words, each node in T is of degree at most m. Thus a node of degree m will be represented by C0, (K1, C1), (K2, C2), (K3, C3) …(K_{m-1}, C_{m-1}) where K_i, 1 ≤ i ≤ m-1 are the keys and Cj , 0 ≤ j≤ m-1 are pointers to the root nodes of the m subtrees of the node.

(ii) If a node has k child nodes, k ≤ m, then the node has exactly (k–1) keys K1, K2, K3,…K_{k-1} where Ki < Ki + 1 and each of the keys Ki partitions the keys in the subtrees into k subsets.

(iii) For a node C0, (K1, C1), (K2, C2), (K3, C3)…. (K_{m-1}, C_{m-1}) all key values in the subtree pointed to by Ci are less than the key Ki + 1, 0 ≤ i ≤ m-2 and the key values in the subtree pointed to by C_{m-1} are greater than K_{m-1}.

(iv) The subtrees pointed to by Ci, 0 ≤ i≤ m-1 are also m-way search trees.

Node structure and representation:

An m-way search tree is conceived to be an extended tree with its null pointers represented by external nodes.

Figure below illustrates a general structure of a node in an m-way search tree. The node has (m–1) key elements and hence exactly m child pointers to the root nodes of the m subtrees.

Those pointers to subtrees which are empty are indicated by external nodes represented as circles.

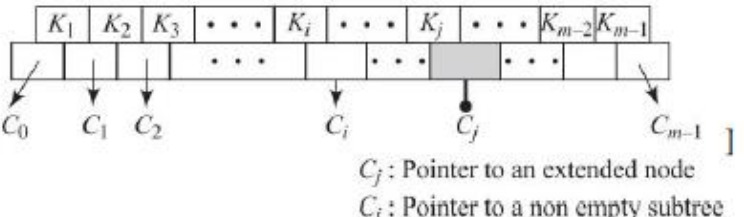


Fig. Structure of a node in an m-way search tree

Example

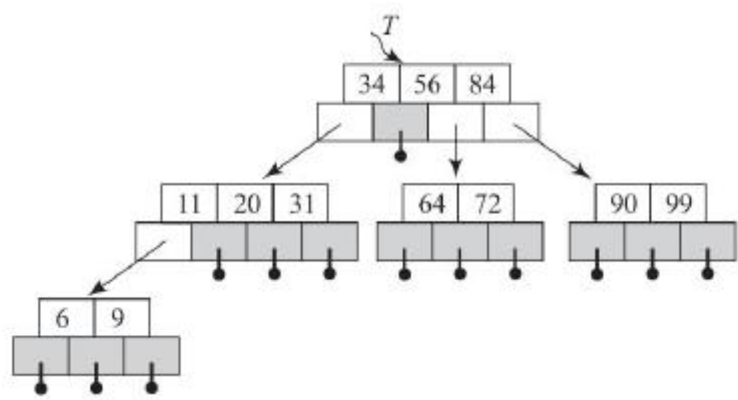


Fig. An example 4-way search tree

Searching an m-way search tree:

K is first sequentially searched against the key elements of the root node [Ki, Ki + 1, … Kt]. If K = Kj then the search is done. If K > Kj and K < Kj + 1 for some j, then the search moves down to the root node of the corresponding subtree Tj.

The search progresses in a similar fashion until the key is obtained in which case the search is termed successful otherwise unsuccessful.

Example:

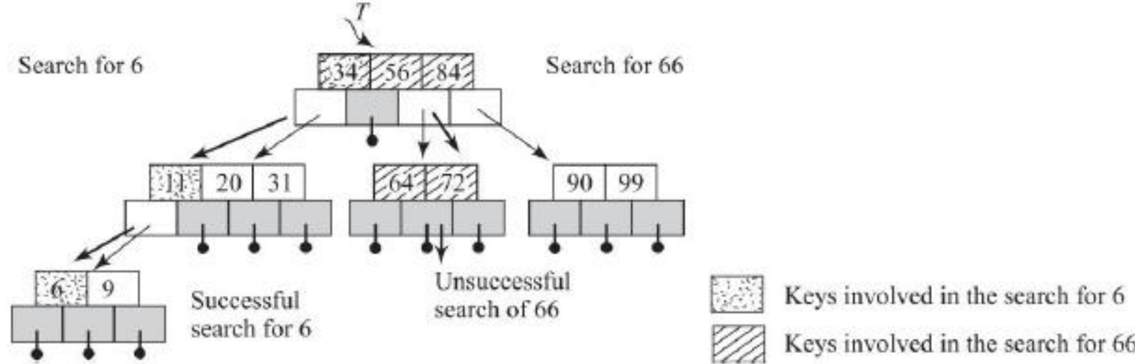


Fig. Search for keys 6 and 66 from the 4-way search tree

<u>Inserting into an m-way search tree:</u>

The insertion of a key into a m-way search tree proceeds as one would search for the key. The search is bound to fall off at some node in the tree. At that position, the key may be either inserted as an element, if the node can accommodate the key or may be inserted as a new node in the next level.

Example:

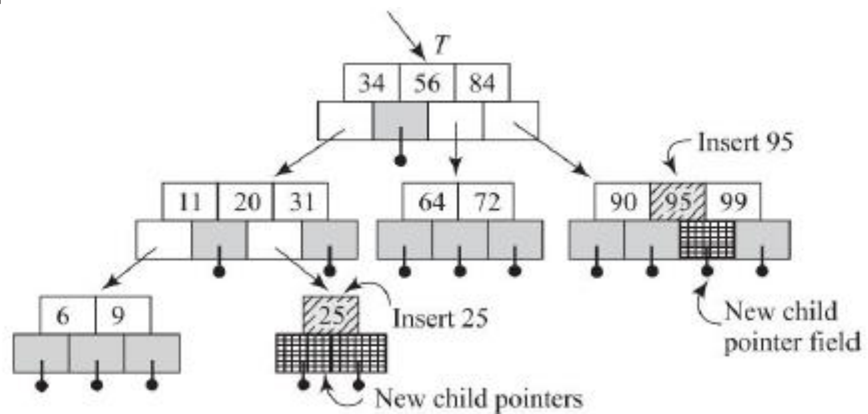


Fig. Insertion of keys 95 and 25 into the 4-way search tree

<u>Deleting from an m-way search tree:</u>

To delete a key we proceed as usual to find the key in the tree. Now let us suppose the key K is found in a node with its left subtree pointer as Ci and its right subtree pointer as Cj. Based on the following cases (Dm in the cases indicates Deletion in an m-way search tree) the deletion of K is undertaken:

<u>Case Dm. 1</u> Ci = Cj = NIL. If the left and right subtrees of key K are NIL, then we simply delete the key K and adjust the number of pointer fields in the node.

Case Dm. 2 Ci = NIL and Cj ≠ NIL. If the left subtree of key K is empty and the right subtree is not, choose the smallest key K' from the right subtree of K and replace K with K'. This in turn may recursively call for the appropriate deletion of K' from the tree following one or more of the four cases.

Case Dm. 3 Ci ≠ NIL and Cj = NIL. If the right subtree of key K is empty and the left subtree is not, choose the largest key K" from the left subtree of K and replace K with K". This in turn may recursively call for the appropriate deletion of K" from the tree following one or more of the four cases.

Case Dm. 4 Ci ≠ NIL and Cj ≠ NIL. If the left subtree and the right subtree of key K are non empty, then choose either the largest key from the left subtree or the smallest key from the right subtree. Call it K"'. Replace key K with the same and as before undertake appropriate steps to delete K"' from the tree.

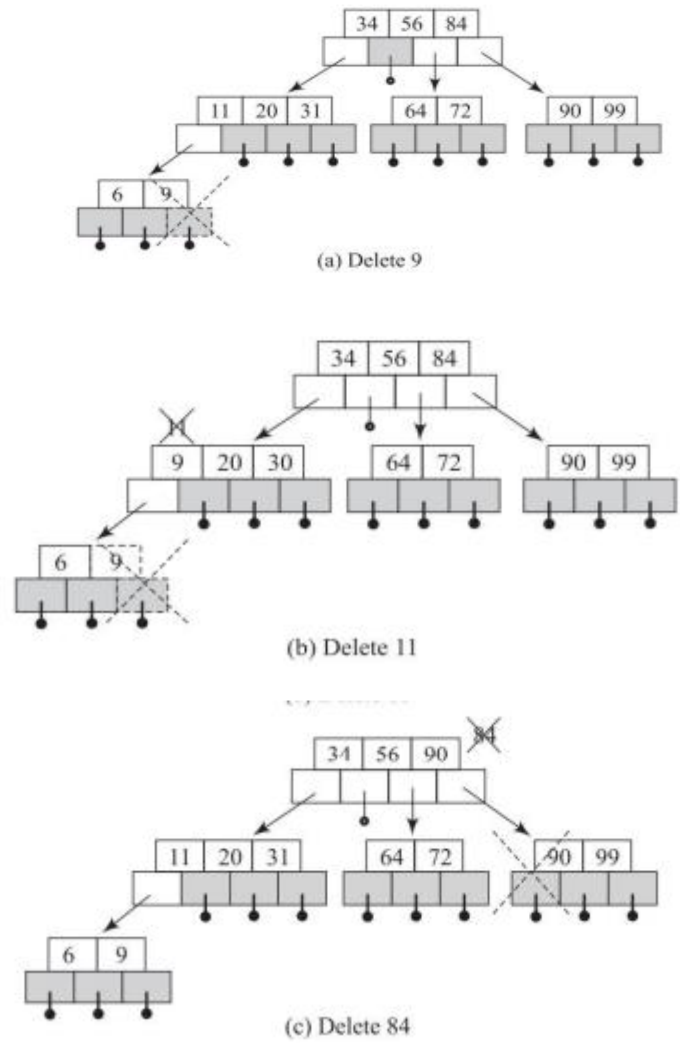


Fig. Deletion (independent) of keys 9, 11 and 84 from the 4-way search tree

B Trees: Definition and Operations:

Balanced m-way search trees which are known as B trees of order m.

Definition :

A B tree of order m is an m-way search tree and hence may be empty. If non empty, then the following properties are satisfied on its extended tree representation:

(i) The root node must have at least two child nodes and at most m child nodes

(ii) All internal nodes other than the root node must have at least $\lceil m/2 \rceil$ non empty child nodes and at most m non empty child nodes

(iii) The number of keys in each internal node is one less than its number of child nodes and

these keys partition the keys of the tree into subtrees in a manner similar to that of m-way

search trees

(iv) All external nodes are at the same level

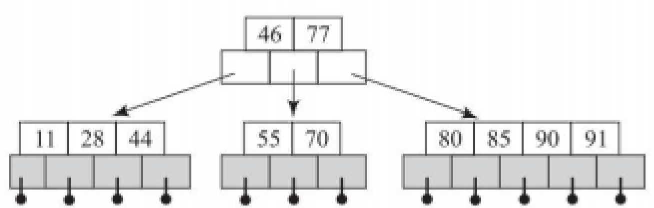


Fig. A B tree of order 5

Searching a B tree of order m :

The search procedure for a B tree of order m is same as the one applied on m-way search trees. The complexity of a search procedure is given by O(h) where h is the height of the B tree of order m.

Inserting into a B tree of order m :

Inserting a key into a B tree of order m proceeds as one would to search for the key. However at the point where the search falls off the tree, the key is inserted based on the following norms (IB in the cases indicates Insertion in a B tree):

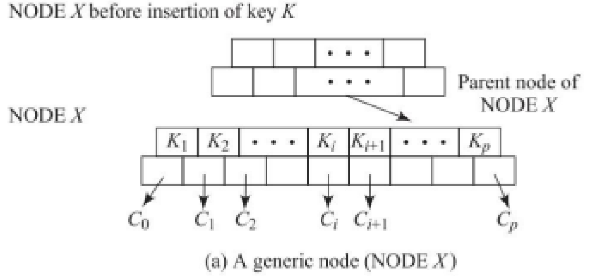
Case IB. 1 If the node X of the B tree of order m, where the key K is to be inserted, can accommodate K, then it is inserted in the node and the number of child pointer fields are appropriately upgraded.

Case IB. 2 If the node X where the key K is to be inserted is full, then we apparently insert K

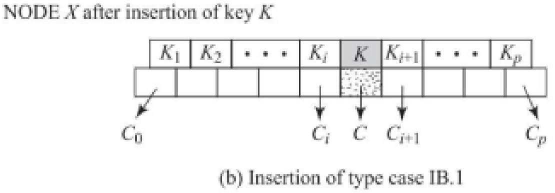
into the list of elements and split the list into two at its median Kmedian.

The keys which are less than Kmedian form a node Xleft and those greater than Kmedian form another node Xright. The median element Kmedian is pulled up to be inserted in the parent node of X.

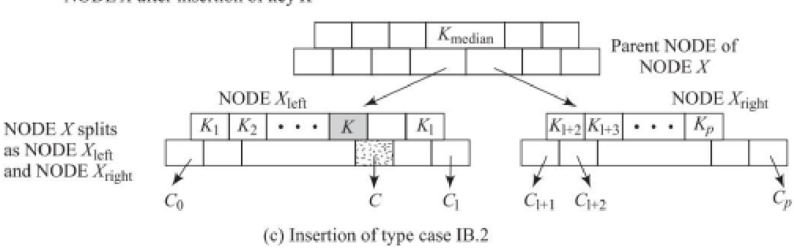
This insertion may in turn call for Case IB. 1 or Case IB. 2 depending on whether the parent node can accommodate Kmedian or not.



(a) A generic node (NODE X)



(b) Insertion of type case IB.1



(c) Insertion of type case IB.2

Fig. Insertion of a key K in a B tree of order m

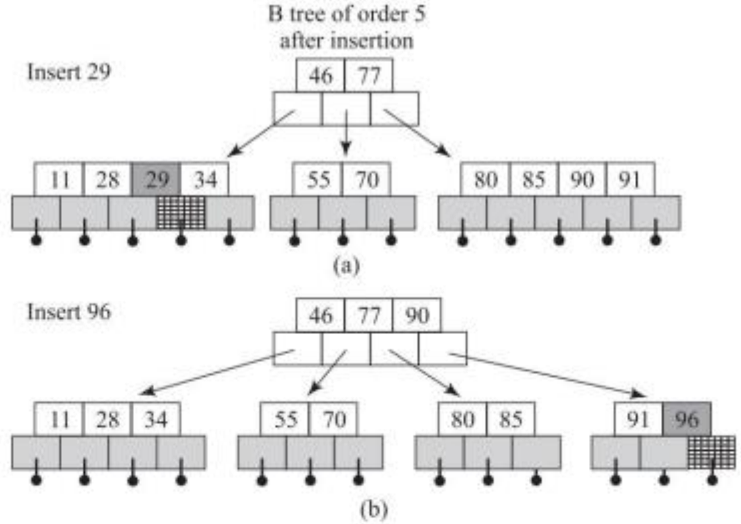


Fig. Insertion of 29 and 96 in the B tree of order 5

Deletion from a B tree of order m :

The deletion of a key K from a B tree of order m may trigger various cases.

Case DB. 1 Key K belongs to a leaf node and its deletion does not result in the node having less than its minimum number of elements. In such a case we merely delete the element from the leaf node and adjust the child pointers accordingly.

Case DB. 2 Key K belongs to a non leaf node. In such a case replace K with the largest key (K') in the left subtree of K or the smallest key (K") from the right subtree of K and follow steps to delete K' or K" from the node. K' or K" is bound to occur in a leaf node and hence triggers Case DB. 1 for their deletion.

Example:

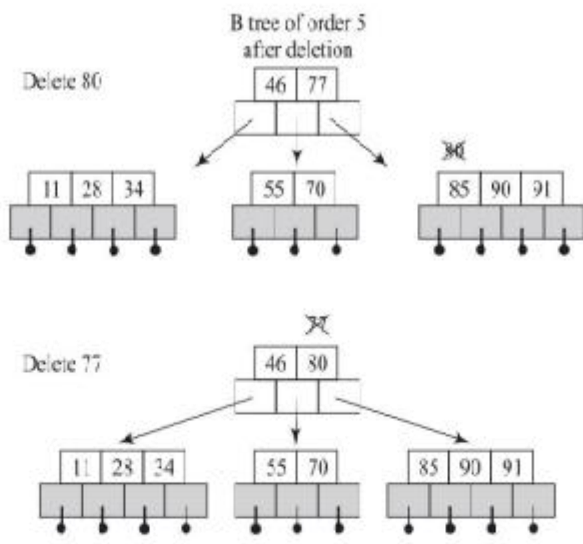


Fig. Deletion of 80 and 77 from the B tree of order 5

Case DB. 3: When the deletion of a key K from a node X leaves it with less than its minimum number of elements, elements are borrowed from one of its left or right sibling nodes.

Thus if the left sibling node has elements to spare, move the largest key K' in the left sibling node to the parent node. The intervening element P in the parent node is moved down to set right the vacancy created by the deletion of K in node X.

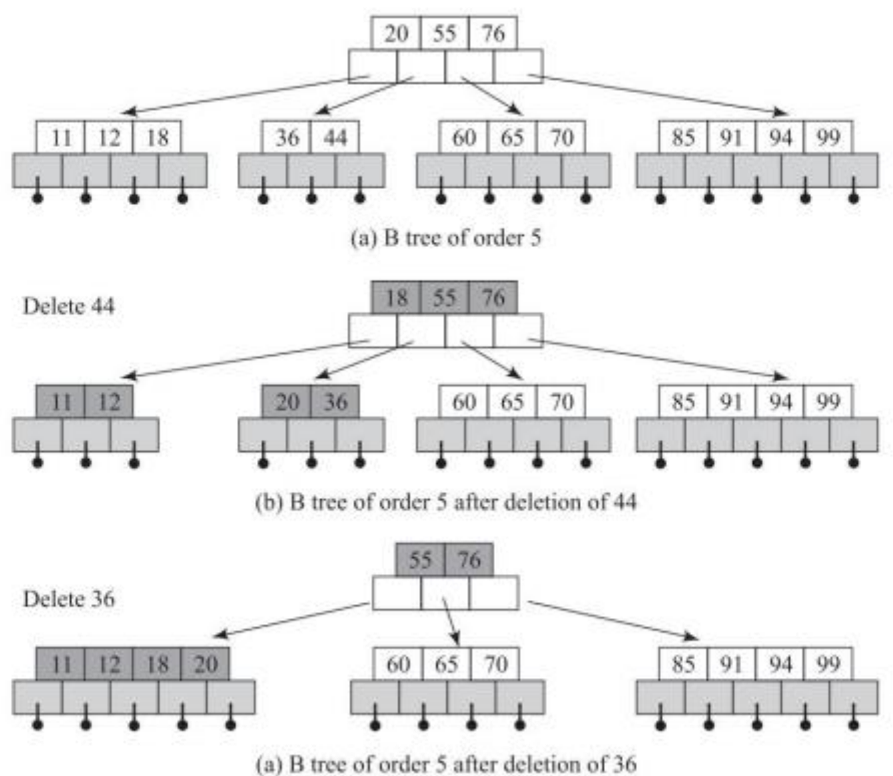
If the left sibling node has no element to spare it would be a waste of time to move to the right sibling node to check if there is an element to spare. In such a case we proceed to Case DB. 4 which covers the case when either of the sibling nodes have no elements to offer.

Case DB. 4 When the deletion of a key K from a node X leaves its elements to be less than the stipulated minimum number and if the first tested sibling node (left or right) or both the

sibling nodes are unable to spare an element, node X is merged with one of the sibling nodes along with the intervening element P in the parent node.

We shall choose to test for the availability of element from the left sibling node first. If there is no element available to be spared, then the elements of the left sibling node are merged with those of node X and the intervening parent element P to create a new node.

This in turn calls for the deletion of element P which may trigger one or more of the cases discussed above.



(a) B tree of order 5

(b) B tree of order 5 after deletion of 44

(a) B tree of order 5 after deletion of 36

Fig. Deletion of 44 and 36 from a B tree of order 5

Height of a B tree of order m:

If a B tree of order m and height h has n elements then n satisfies $n \le m^h - 1$. This is true since a B tree of order m is basically an m-way search tree.

Now having determined the upper bound of n, what is its lower bound? In other words what is the minimum number of elements that a B tree of order m and height h can hold?

To obtain this let us find out what are the minimum number of nodes in levels 1, 2,….(h+1). Here (h+1) is the level at which the external nodes reside.

Since each internal node other than the root have a minimum of $\lceil m/2 \rceil$ child nodes and the root has just one node, the minimum number of nodes in each level beginning from 1 and ending at (h+1) in the sequential order would be 1, 2, $2 . \lceil m/2 \rceil$, $2 . \lceil m/2 \rceil^2$ ….. $2 . \lceil m/2 \rceil^{h-1}$ respectively.

Thus the number of external nodes on level (h + 1) would be $2 . \lceil m/2 \rceil^{h-1}$. Since the number of elements in the B tree is one less than the number of external nodes, the lower bound of n is given by $\mathbf{n \ge 2 . \lceil m/2 \rceil^{h-1} - 1}$.

Hence we have $2 . \lceil m/2 \rceil^{h-1} - 1 \le n \le m^h - 1$ **.**

INTRODUCTION TO B+ Tree:

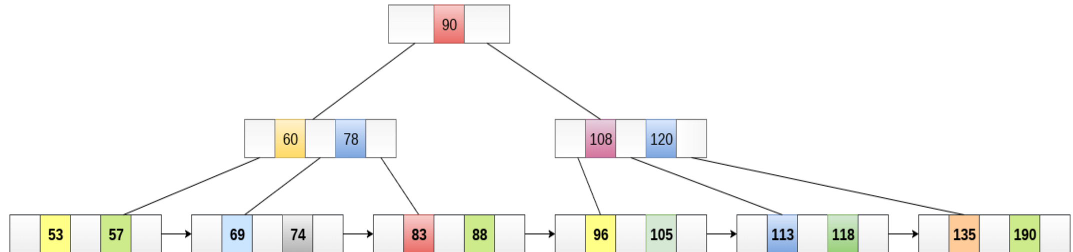
B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

The leaf nodes of a B+ tree are linked together in the form of singly linked lists to make the search queries more efficient.

B+ Tree are used to store the large amount of data which cannot be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in the following figure.



Advantages of B+ Tree

1. Records can be fetched in equal number of disk accesses.
2. Height of the tree remains balanced and less as compare to B tree.
3. We can access the data stored in a B+ tree sequentially as well as directly.
4. Keys are used for indexing.
5. Faster search queries as the data is stored only on the leaf nodes.

B Tree VS B+ Tree

SN	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.

2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

Dictionaries:

Dictionary is a collection of data elements uniquely identified by a field called key. A dictionary supports the operations of search, insert and delete.

The ADT of a dictionary is defined as a set of elements with distinct keys supporting the operations of search, insert, delete and create (which creates an empty dictionary).

A dictionary supports both sequential and random access. A sequential access is one in which the data elements of the dictionary are ordered and accessed according to the order of the keys (ascending or descending, for example).

A random access is one in which the data elements of the dictionary are not accessed according to a particular order. Hash tables are ideal data structures for dictionaries.

Hash Table Structure:

A hash function H(X) is a mathematical function which given a key X of the dictionary D, maps it to a position P in a storage table termed hash table. The process of mapping the keys to their respective positions in the hash table is called hashing. Figure below illustrates a hash function.

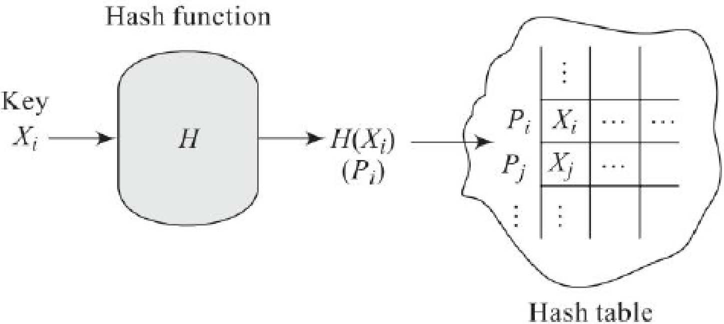


Fig. Hashing a key

When the data elements of the dictionary are to be stored in the hash table, each key X_i is mapped to a position P_i in the hash table as determined by the value of $H(X_i)$, (i.e.) $P_i = H(X_i)$.

To search for a key X in the hash table all that one does is to determine the position P by computing P = H(X) and access the appropriate data element.

In the case of insertion of a key X or its deletion, the position P in the hash table where the data element needs to be inserted or from where it is to be deleted respectively, is determined by computing P = H(X).

If the hash table is implemented using a sequential data structure, for example arrays, then the hash function H(X) may be so chosen to yield a value that corresponds to the index of the array. In such a case, the hash function is a mere mapping of the keys to the array indices.

Example: Consider a set of distinct keys { AB12, VP99, RK32, CG45, KL78, OW31, ST65, EX44 } to be represented as a hash table. Let us suppose the hash function H is defined as below:

H(XYmn) = ord(X) where X, Y are the alphabetical characters, m, n are the numerical characters of the key and ord(X) is the ordinal number of the alphabet X.

The computation of the positions of the keys in the hash table is shown below:

Key *XYmn*	*H(XYmn)*	Position of the key in the hash table
AB12	*ord(A)*	1
VP99	*ord(V)*	22
RK32	*ord(I)*	18
CG45	*ord(C)*	3
KL78	*ord(K)*	11
OW31	*ord(O)*	15
ST65	*ord(S)*	19
EX44	*ord(E)*	5

Let X1, X2, ….Xn be the n keys which are mapped to the same position P in the hash table. Then H(X1) = H(X2) = …H(Xn) = P. In such a case, X1, X2, ….Xn are called as **synonyms.**

The act of two or more synonyms vying for the same position in the hash table is known as **collision**.

Naturally, this entails a modification in the structure of the hash table to accommodate the synonyms. The two important methods are **linear open addressing** and **chaining** to handle synonyms.

The hash table accommodating the data elements appears as shown below:

1	AB12	………
2	…	
3	CG45	
4	…	
5	EX44	………
…	…	
11	KL78	
…		
15	OW31	………
…		
18	RK32	
19	ST65	………
…		
22	VP99	………
…	…	………

Hash Functions:

The choice of the hash function plays a significant role in the structure and performance of the hash table. It is therefore essential that a hash function satisfies the following characteristics:

(i) easy and quick to compute

(ii) even distribution of keys across the hash table. In other words, a hash function must minimize collisions.

Building hash functions:

The following are some of the methods of obtaining hash functions:

(i) Folding: The key is first partitioned into two or three or more parts. Each of the individual parts are combined using any of the basic arithmetic operations such as addition or multiplication. The resultant number could be conveniently manipulated, for example truncated, to finally arrive at the index where the key is to be stored. Folding assures better spread of keys across the hash table.

Example Consider a six digit numerical key: 719532. We choose to partition the key into three parts of two digits each, (i.e.) 71 | 95 | 32, and merely add the numerical equivalent of each of the parts, (i.e.) 71+ 95+ 32 = 198. Truncating the result yields 98 which is chosen as the index of the hash table where the key 719532 is to be accommodated.

(ii) Truncation: In this method the selective digits of the key are extracted to determine the index of the hash table where the key needs to be accommodated. In the case of alphabetical keys their numerical equivalents may be considered. Truncation though quick to compute, does not ensure even distribution of keys.

Example: Consider a group of six digit numerical keys that need to be accommodated in a hash table with 100 locations. We choose to select digits in position 3 and 6 to determine the index where the key is to be stored. Thus key 719532 would be stored in location 92 of the hash table.

(iii) Modular Arithmetic: This is a popular method and the size of the hash table L is involved in the computation of the hash function. The function makes use of modulo arithmetic. Let k be the numerical key or the numerical equivalent if it is an alphabetical key. The hash function is given by

H(k) = k mod L.

The hash function evidently returns a value that lies between 0 and L–1. Choosing L to be a prime number has a proven better performance by way of even distribution of keys.

Example: Consider a group of six digit numerical keys that need to be stored in a hash table of size 111. For a key 145682, H(k) = 145682 mod 111 = 50. Hence the key is stored in location 50 of the hash table.

Linear Open Addressing:

Let us suppose a group of keys are to be inserted into a hash table HT of size L, making use of the modulo arithmetic function H(k) = k mod L.

Since the range of the hash table index is limited to lie between 0 and L-1, for a population of N (N>L) keys collisions are bound to occur.

Hence a provision needs to be made in the hash table to accommodate the data elements that are synonyms. We choose to adopt a sequential data structure to accommodate the hash table.

Let HT[0: L-1] be the hash table. Here the L locations of the hash table are termed as **buckets**. Every bucket provides accommodation for the data elements.

However to accommodate synonyms (i.e.) keys which map to the same bucket, it is essential that a provision be made in the buckets. We therefore partition buckets into what are called **slots** to accommodate synonyms.

Thus if a bucket b has s slots, then s synonyms can be accommodated in the bucket b. In the case of an array implementation of a hash table, the rows of the array indicate buckets and the columns the slots.

In such a case, the hash table is represented as HT[0:L-1, 0:S-1]. The choice of number of slots in a bucket needs to be decided based on the application.

Figure below illustrates a general hash table implemented using a sequential data structure.

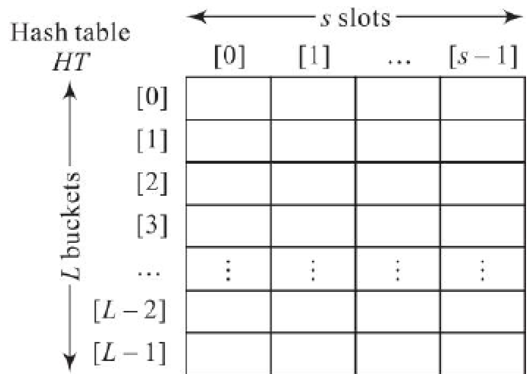


Fig. Hash table implemented using a sequential data structure

Now what happens if a synonym is unable to find a slot in the bucket? In other words, if the bucket is full, then where do we find place for the synonyms? In such a case an overflow is said to have occurred.

All collisions need not result in overflows. But in the case of a hash table with single slot buckets, collisions mean overflows.

The bucket to which the key is mapped by the hash function is known as the home bucket. To tackle overflows we move further down, beginning from the home bucket and look for the closest slot that is empty and place the key in it. Such a method of handling overflows is known as **Linear probing** or **Linear open addressing** or **closed hashing.**

Example : Let us consider a set of keys {45, 98, 12, 55, 46, 89, 65, 88, 36, 21} to be represented as a hash table. Let us suppose the hash function H is defined as H(X) = X mod 11. The hash table therefore has 11 buckets. We propose 3 slots per bucket. Table below shows the hash function values of the keys and Fig. below shows the structure of the hash table.

Observe how keys {45, 12, 89}, {98, 65, 21} and {55, 88} are synonyms mapping to the same bucket 1, 10 and 0 respectively. The provision of 3 slots per bucket makes it possible to accommodate synonyms.

Table 13.1 *Hash function values of the keys (Example 13.2)*

Key X	45	98	12	55	46	89	65	88	36	21
H(X)	1	10	1	0	2	1	10	0	3	10

Hash Table

HT	[0]	[1]	[2]
[0]	55	88	
[1]	45	12	89
[2]	46		
[3]	36		
[4]			
[5]			
[6]			
[7]			
[8]			
[9]			
[10]	98	65	21

Let us proceed to insert the keys { 77, 34, 43} in the hash table

Hash Table

HT	[0]	[1]	[2]
[0]	55	48	77
[1]	45	12	89
[2]	46	34	43
[3]	36		
[4]			
[5]			
[6]			
[7]			
[8]			
[9]			
[10]	98	65	21

Operations on linear open addressed hash tables:

Search: Searching for a key in a linear open addressed hash table proceeds on lines similar to that of insertion. However, if the searched key is available in the home bucket then the search is done.

However, if there had been overflows while inserting the key, then a sequential search has to be called for, which searches through each slot of the buckets following the home bucket, until either (i) the key is found or (ii) an empty slot is encountered in which case the search terminates or (iii) the

search path has curled back to the home bucket. In the case of (i) the search is said to be successful. In the case of (ii) and (iii) it is said to be unsuccessful.

Example: Consider the snapshot of the hash table shown in Fig. below, which represents keys whose first character lies between A and I , both inclusive.

The hash function used is **H(X) = ord(C) mod 10** where C is the first character of the alphabetical key X.

The search for keys F18 and G64 are straightforward since they are present in their home buckets viz., 6 and 7 respectively.

The search for keys A91 and F78 for example, are a trifle involved in the sense that, though they are available in their respective home buckets, they are accessed only after a sequential search for them is done in the slots corresponding to their buckets.

On the other hand, the search for I99 fails to find it in its home bucket viz., 9. This therefore triggers a sequential search of every slot following the home bucket until the key is found, in which case the search is successful or until an empty slot is encountered in which case the search is a failure.

I99 is indeed found in slot 2 of bucket 2. Observe how the search path curls back to the top of the hash table from the home bucket of key I99.

Let us now search for the key G93. The search proceeds to look into its home bucket (7) before a sequential search for the same is undertaken in the slots following the home bucket. The search stops due to its encountering an empty slot and therefore the search is deemed unsuccessful.

Hash Table

HT	[0]	[2]
[0]	I81	I90
[1]	A12	A91
[2]	B47	I99
[3]		
[4]	D36	
[5]		
[6]	F18	F78
[7]	G64	F73
[8]	H11	F99
[9]	I54	I75
	⋮	⋮

Fig. Illustration of search in a hash table

Insert: The insertion of data elements in a linear open addressed hash table is executed as like search operation. The hash function that is quite often modulo arithmetic based, determines the bucket b and thereafter slot s in which the data element is to be inserted. In the case of overflow, we search for the closest empty slot beginning from the home bucket and accommodate the key in the slot.

Delete: When a key is deleted it cannot be merely wiped off from its bucket (slot). A deletion leaves the slot vacant and if an empty slot is chosen as a signal to terminate a search then many of the elements following the empty slot and displaced from their home buckets may go unnoticed.

To tackle this it is essential that the keys following the empty slot are moved up. This can make the whole operation clumsy.

An alternative could be, to write a special element in the slot every time a delete operation is done. This special element not only serves to camouflage the empty space available in the deleted slot when a search is under progress, but also serves to accommodate an insertion when an appropriate element assigned to the slot turns up.

Chaining:

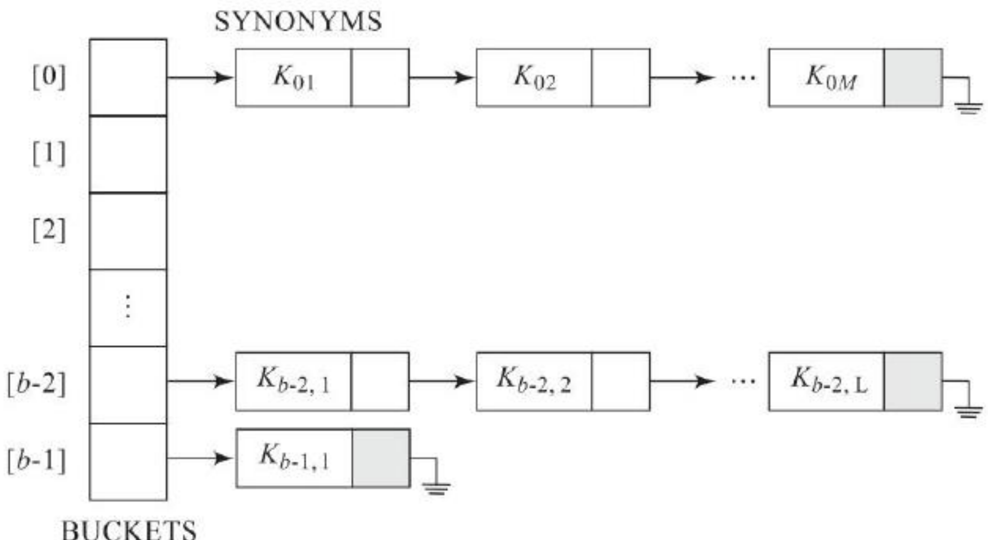
In the case of linear open addressing, the solution of accommodating synonyms in the closest empty slot may contribute to a deterioration in performance.

For example, the search for a synonym key may involve sequentially going through every slot occurring after its home bucket before it is either found or unfound. Also, the implementation of the hash table using a sequential data structure such as arrays, limits its capacity (b × s slots).

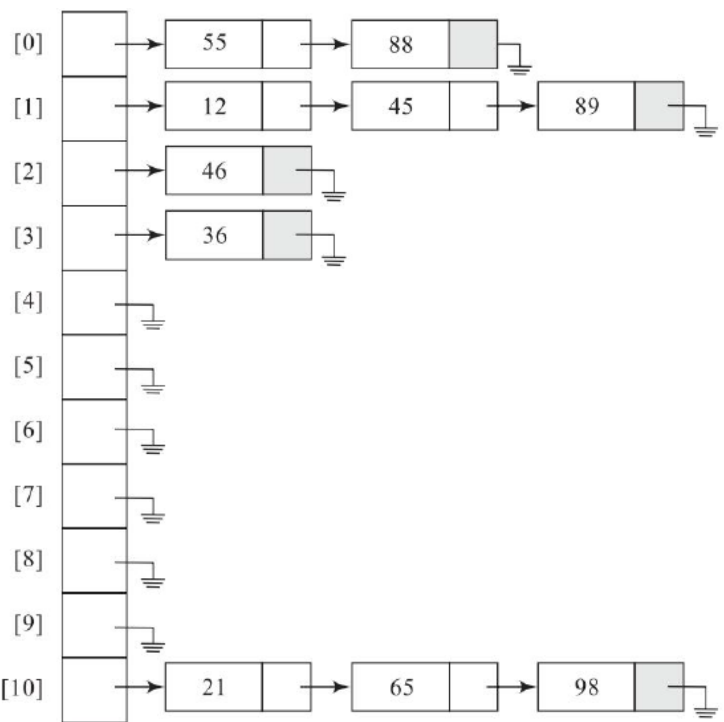
While increasing the number of slots to minimize overflows may lead to wastage of memory, containing the number of slots to the bare minimum may lead to severe overflows hampering the performance of the hash table.

An alternative to overcome this malady is to keep all synonyms that are mapped to the same bucket chained to it. In other words, every bucket is maintained as a singly linked list with synonyms represented as nodes.

The buckets continue to be represented as a sequential data structure as before and to favor the hash function computation. Such a method of handling overflows is called **chaining or open hashing or separate chaining**. Figure below illustrates a chained hash table.



Example : Let us consider the set of keys {45, 98, 12, 55, 46, 89, 65, 88, 36, 21} listed in to be represented as a chained hash table. The hash function H used is H(X) = X mod 11. The hash function values for the keys are as shown in Table. The structure of the chained hash table is as shown in Fig.



Operations on chained hash tables:

Search: The search for a key X in a chained hash table proceeds by computing the hash function value H(X). The bucket corresponding to the value H(X) is accessed and a sequential search along the chain of nodes is undertaken. If the key is found then the search is termed **successful** otherwise **unsuccessful**. If the chain is too long, maintaining the chain in order (ascending or descending) helps in rendering the search efficient.

Insert: To insert a key X into a hash table, we compute the hash function H(X) to determine the bucket. If the key is the first node to be linked to the bucket then all that it calls for, is a mere execution of a function to insert a node in an empty singly linked list.

In the case of keys which are synonyms, the new key could be inserted either in the beginning or at the end of the chain leaving the list unordered. However, it would be prudent and less expensive too, to maintain each of the chains in the ascending or descending order of the keys. This would also render the search for a specific key amongst its synonyms to be efficiently carried out.

Delete: Unlike that of linear open addressed hash tables, the deletion of a key X in a chained hash table is elegantly done. All that it calls for, is a search for X in the corresponding chain and a deletion of the respective node.