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.

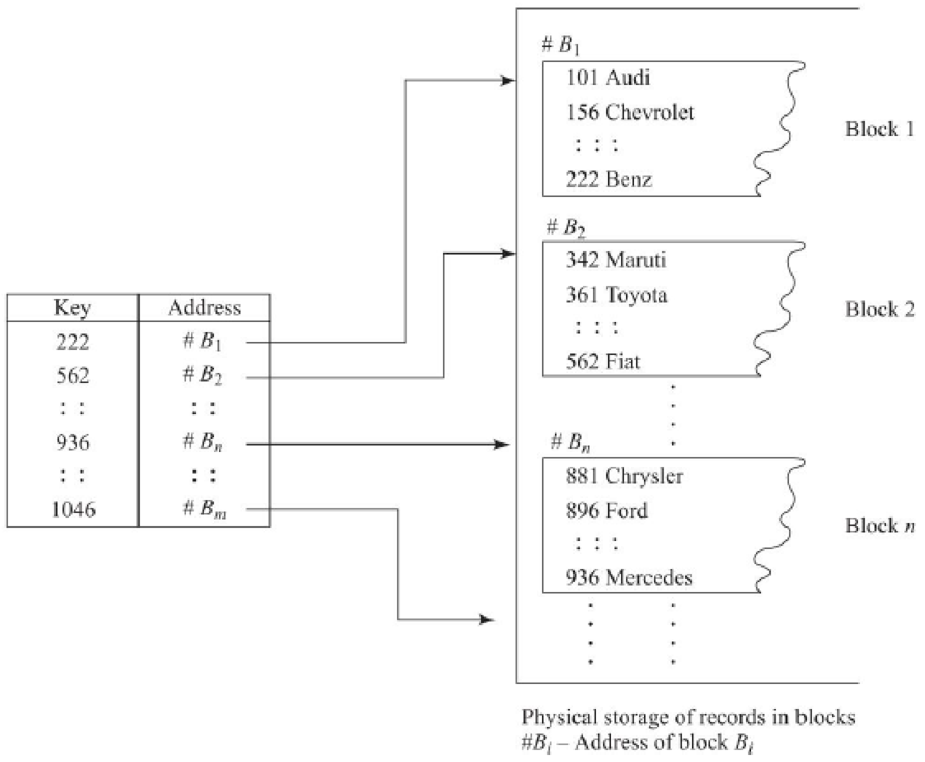


Fig. An ISAM file structure

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.

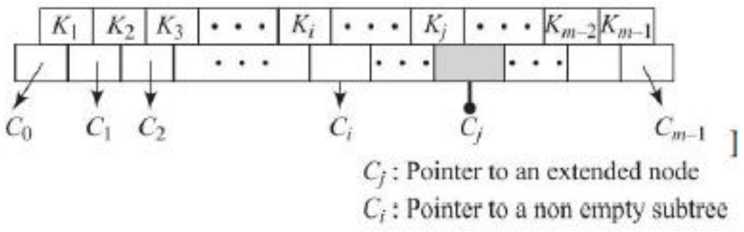


Fig. Structure of a node in an m-way search tree

Example

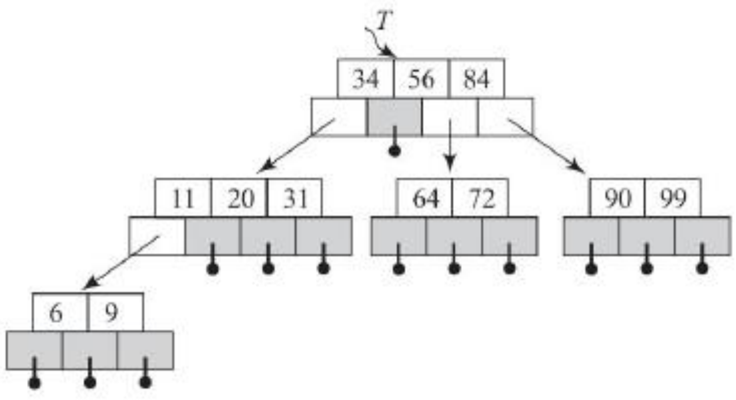


Fig. An example 4-way search tree

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:

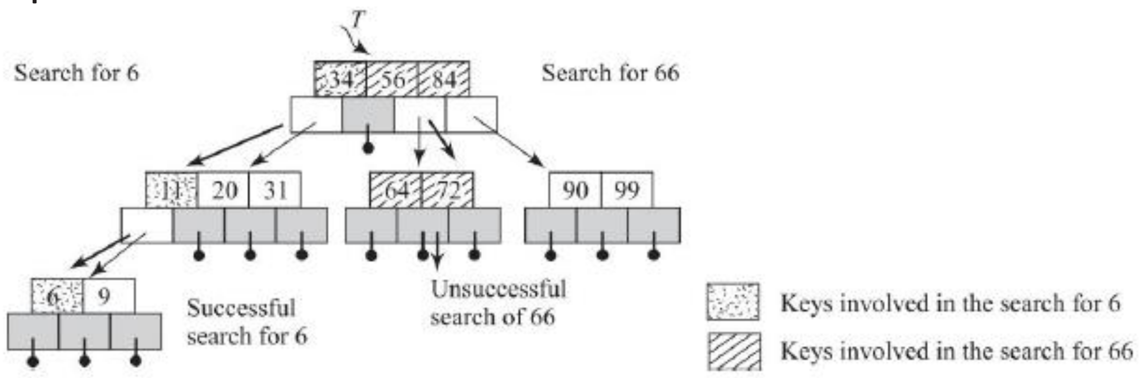

T
Search for 6
Search for 66
34
56
84
11
20
31
64
72
90
99
6
9
Successful search for 6
Unsuccessful search of 66
Keys involved in the search for 6
Keys involved in the search for 66

Fig. Search for keys 6 and 66 from the 4-way search tree

Inserting into an m-way search tree:

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:

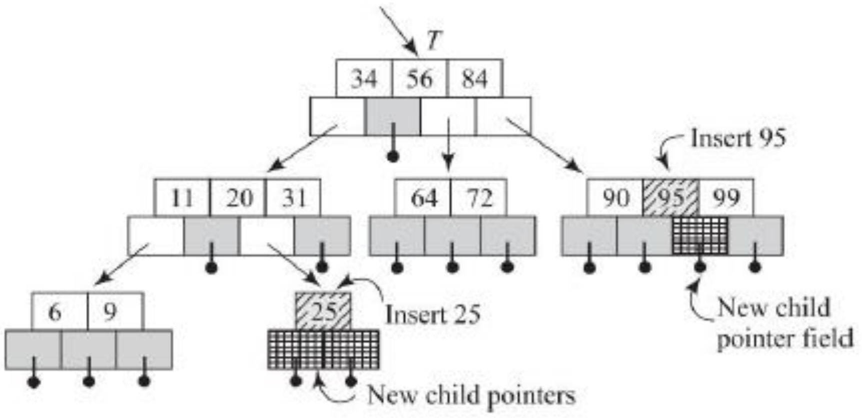


Fig. Insertion of keys 95 and 25 into the 4-way search tree

Deleting from an m-way search tree:

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:

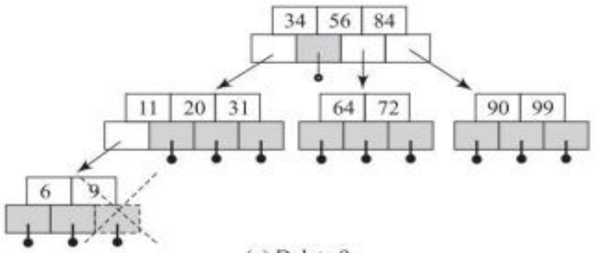
Case Dm. 1 Ci = Cj = NIL. If the left and right subtrees of key K are NIL, then we simply delete

the key K and adjust the number of pointer fields in the node.

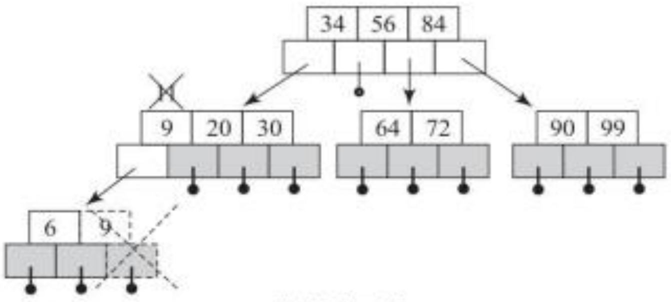
Case Dm. 2 Ci = NIL and Cj ≠ NIL. If the left subtree of key K is empty and the right subtree is not, choose the smallest key K′ from the right subtree of K and replace K with K′. This in turn may recursively call for the appropriate deletion of K′ from the tree following one or more of the four cases.

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.

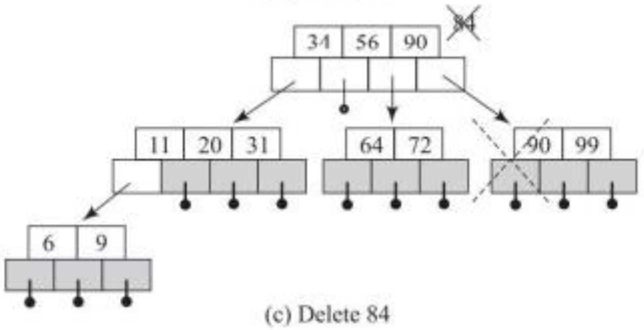
Case Dm. 4 Ci ≠ NIL and Cj ≠ NIL. If the left subtree and the right subtree of key K are non empty, then choose either the largest key from the left subtree or the smallest key from the right subtree. Call it K′′. Replace key K with the same and as before undertake appropriate steps to delete K′′ from the tree.



(a) Delete 9



(b) Delete 11



(c) Delete 84

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

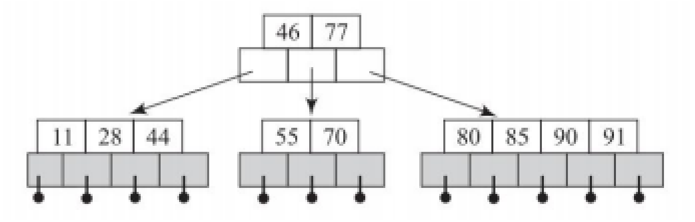


Fig. A B tree of order 5

<u>Searching a B tree of order m</u> :

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.

<u>Inserting into a B tree of order m :</u>

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):

<u>Case IB. 1</u> 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.

<u>Case IB. 2</u> 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.

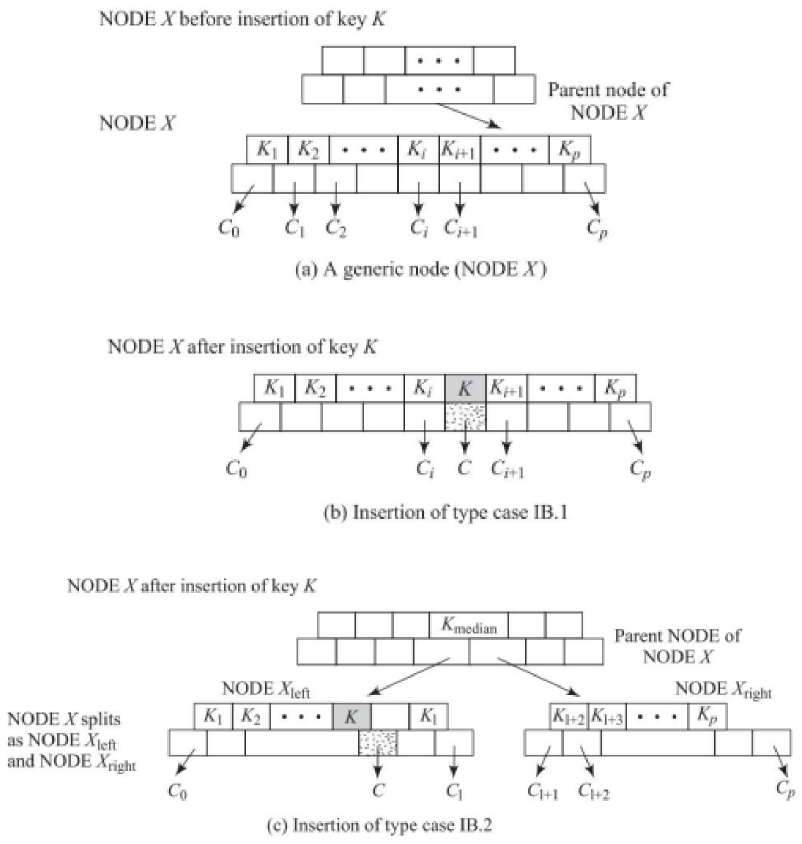

NODE X before insertion of key K
Parent node of NODE X
NODE X
K_1 K_2 K_i K_{i+1} K_p
C_0 C_1 C_2 C_i C_{i+1} C_p
(a) A generic node (NODE X)
NODE X after insertion of key K
K_1 K_2 K_i K K_{i+1} K_p
C_0 C_i C C_{i+1} C_p
(b) Insertion of type case IB.1
NODE X after insertion of key K
K_{median}
Parent NODE of NODE X
NODE X_{left}
NODE X_{right}
NODE X splits as NODE X_{left} and NODE X_{right}
K_1 K_2 K K_l
K_{l+2} K_{l+3} K_p
C_0 C C_l C_{l+1} C_{l+2} C_p
(c) Insertion of type case IB.2

Fig. Insertion of a key K in a B tree of order m

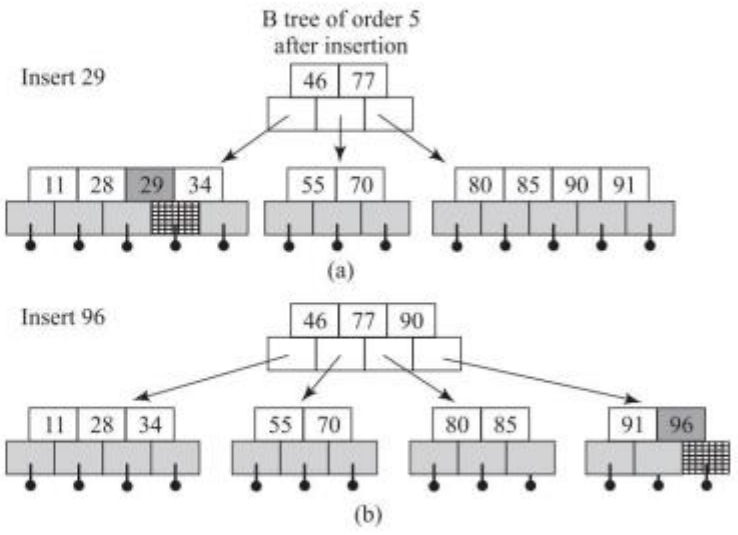


Fig. Insertion of 29 and 96 in the B tree of order 5

<u>Deletion from a B tree of order m :</u>

The deletion of a key K from a B tree of order m may trigger various cases.

<u>Case DB. 1</u> 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.

<u>Case DB. 2</u> 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:

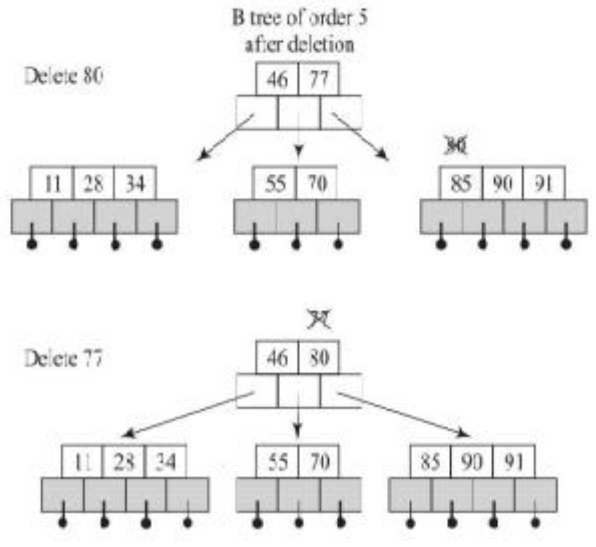


Fig. Deletion of 80 and 77 from the B tree of order 5

<u>Case DB. 3:</u> 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.

<u>Case DB. 4</u> 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.

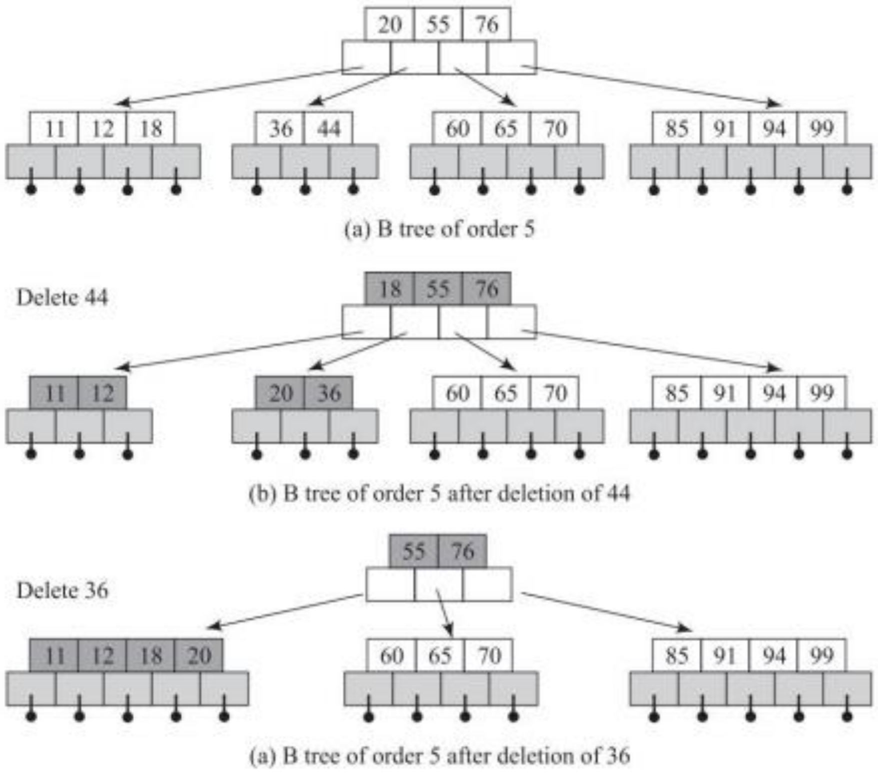


Fig. Deletion of 44 and 36 from a B tree of order 5

<u>Height of a B tree of order m:</u>

If a B tree of order m and height h has n elements then n satisfies $n \leq m^h - 1$. 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 $\mathbf{n \geq 2. \lceil m/2 \rceil^{h-1} - 1}$.

Hence we have $\mathbf{2. \lceil m/2 \rceil^{h-1} - 1 \leq n \leq m^h - 1}$ **.**

INTRODUCTION TO B+ Tree:

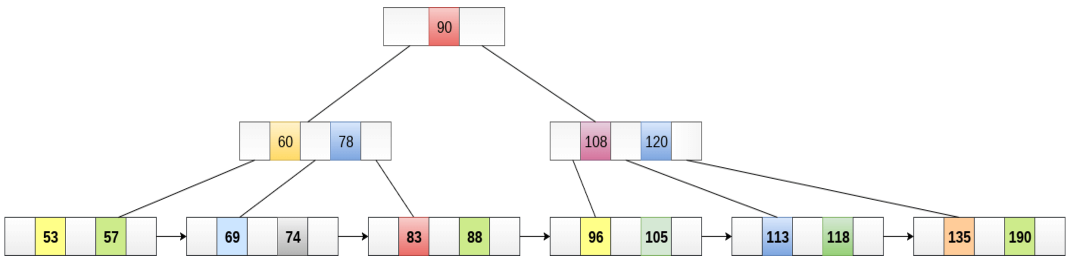
B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

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.

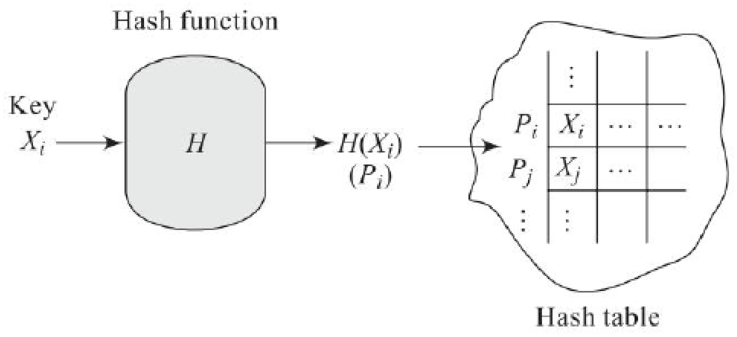


Fig. Hashing a key

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	………
…	…	………

<u>**Hash Functions:**</u>

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.

<u>Building hash functions:</u>
The following are some of the methods of obtaining hash functions:
<u>(i) Folding:</u> 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.

<u>(ii) Truncation:</u> 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.

<u>(iii) Modular Arithmetic:</u> 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.

<u>Linear Open Addressing:</u>

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.

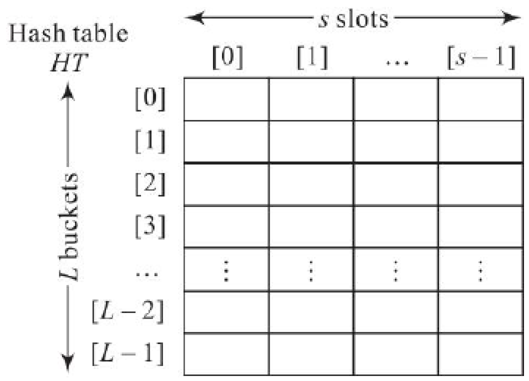


Fig. Hash table implemented using a sequential data structure

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.

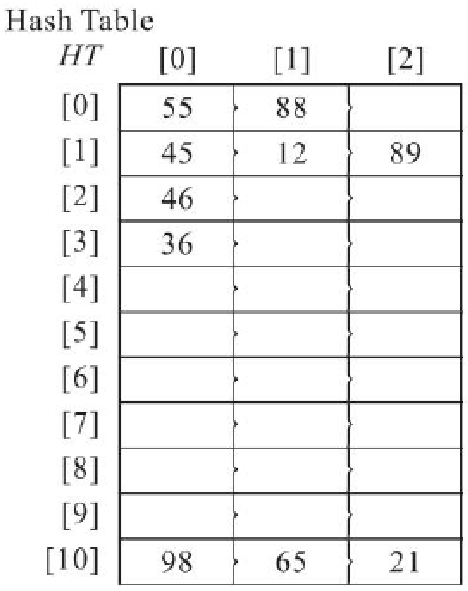
The bucket to which the key is mapped by the hash function is known as the home bucket. To tackle overflows we move further down, beginning from the home bucket and look for the closest slot that is empty and place the key in it. Such a method of handling overflows is known as **Linear probing** or **Linear 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)*

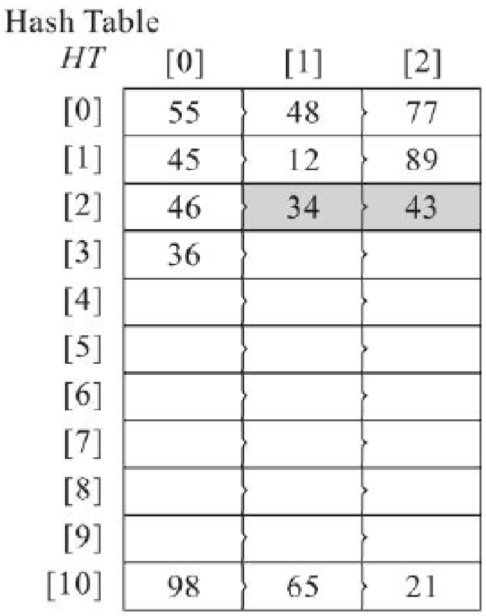
Key X	45	98	12	55	46	89	65	88	36	21
H(X)	1	10	1	0	2	1	10	0	3	10



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.

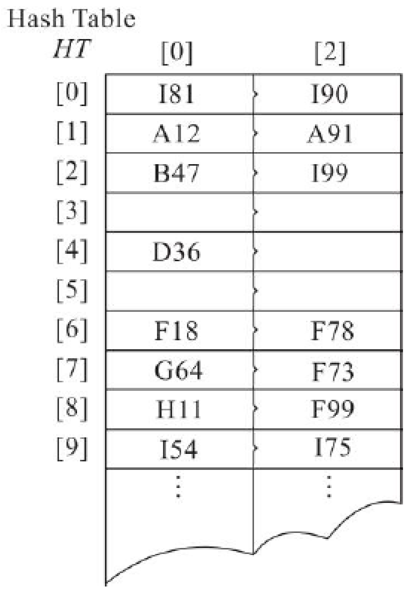
The search for keys F18 and G64 are straightforward since they are present in their home buckets viz., 6 and 7 respectively.

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:

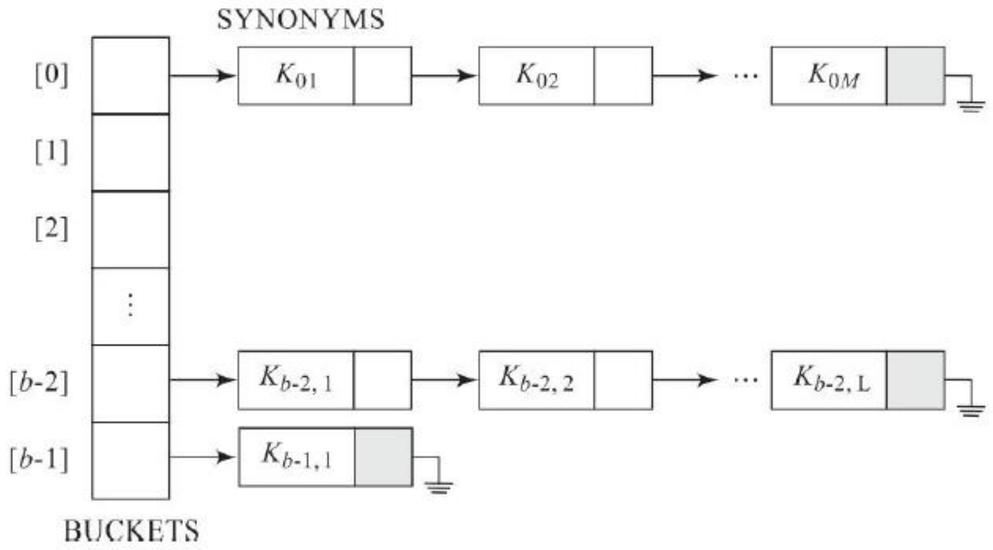
In the case of linear open addressing, the solution of accommodating synonyms in the closest empty slot may contribute to a deterioration in performance.

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).

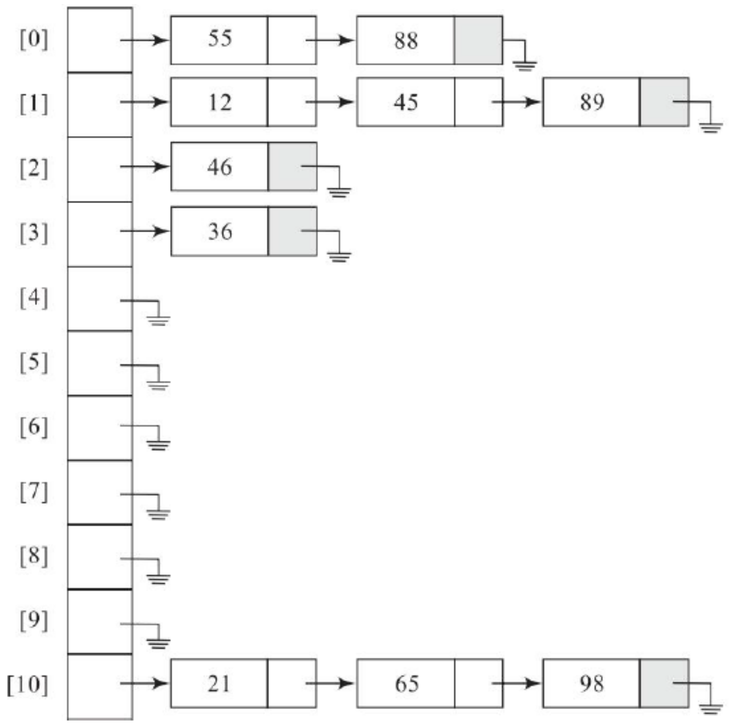
While increasing the number of slots to minimize overflows may lead to wastage of memory, containing the number of slots to the bare minimum may lead to severe overflows hampering the performance of the hash table.

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.



<u>Operations on chained hash tables:</u>

<u>Search:</u> 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.

<u>Insert:</u> 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.

<u>Delete:</u> 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.