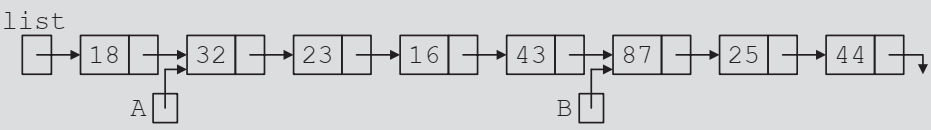
**LINKED LIST: QUICK REVIEW**

1. A linked list is a list of items, called nodes, in which the order of the nodes is determined by the address, called a link, stored in each node.
2. The pointer to a linked list—that is, the pointer to the first node in the list—is stored in a separate location, called the head or first.
3. A linked list is a dynamic data structure.
4. The length of a linked list is the number of nodes in the list.
5. Item insertion and deletion from a linked list does not require data movement; only the pointers are adjusted.
6. A (single) linked list is traversed in only one direction.
7. The search on a linked list is sequential.
8. The first (or head) pointer of a linked list is always fixed, pointing to the first node in the list.
9. To traverse a linked list, the program must use a pointer different than the head pointer of the list, initialized to the first node in the list.
10. In a doubly linked list, every node has two links: one points to the next node, and one points to the previous node.
11. A doubly linked list can be traversed in either direction.
12. In a doubly linked list, item insertion and deletion requires the adjustment of two pointers in a node.
13. A linked list with header and trailer nodes simplifies the insertion and deletion operations.
14. A circular linked list is a list in which, if the list is nonempty, the last node points to the first node.

**EXERCISES**

1. Mark the following statements as true or false.
   1. In a linked list, the order of the elements is determined by the order in which the nodes were created to store the elements.
   2. In a linked list, memory allocated for the nodes is sequential.
   3. A single linked list can be traversed in either direction.
   4. In a linked list, nodes are always inserted either at the beginning or the end because a linked link is not a random access data structure.
   5. The head pointer of a linked list cannot be used to traverse the list.

Consider the linked list shown below. Assume that the nodes are in the usual info-link form. Use this list to answer Exercises 2 through 7. If necessary, declare additional variables. (Assume that list, p, s, A, and B are pointers of type nodeType.



Linked list for Exercises 2–7

1. What is the output of each of the following C++ statements?
   1. cout << list->info;
   2. cout << A->info;
   3. cout << B->link->info;
   4. cout << list->link->link->info
2. What is the value of each of the following relational expressions?
   1. list->info >= 18
   2. list->link == A
   3. A->link->info == 16
   4. B->link == NULL
   5. list->info == 18
3. Mark each of the following statements as valid or invalid. If a statement is invalid, explain why.
   1. A = B;
   2. list->link = A->link;
   3. list->link->info = 45;
   4. \*list = B;
   5. \*A = \*B;
   6. B = A->link->info;
   7. A->info = B->info;
   8. list = B->link->link;
   9. B = B->link->link->link;
4. Write C++ statements to do the following:
   1. Make A point to the node containing info 23.
   2. Make list point to the node containing 16.
   3. Make B point to the last node in the list.
   4. Make list point to an empty list.
   5. Set the value of the node containing 25 to 35.
   6. Create and insert the node with info 10 after the node pointed to by A.
   7. Delete the node with info 23. Also, deallocate the memory occupied by this node.
5. What is the output of the following C++ code?

p = list;

while (p != NULL)

cout << p->info << " ";

p = p->link;

cout << endl;

1. If the following C++ code is valid, show the output. If it is invalid, explain why.

a. s = A;

p = B;

s->info = B;

p = p->link;

cout << s->info << " " << p->info << endl;

b. p = A;

p = p->link;

s = p;

p->link = NULL;

s = s->link;

cout << p->info << " " << s->info << endl;

1. Show what is produced by the following C++ code. Assume the node is in the usual info-link form with the info of type int.

a.

#include <iostream>

using namespace std;

struct nodeType {

int info;

nodeType\* link;

};

int main() {

nodeType\* list;

nodeType\* ptr;

list = new nodeType;

list->info = 10;

ptr = new nodeType;

ptr->info = 13;

ptr->link = NULL;

list->link = ptr;

ptr = new nodeType;

ptr->info = 18;

ptr->link = list->link;

list->link = ptr;

cout << list->info << " " << ptr->info << " ";

ptr = ptr->link;

cout << ptr->info << endl;

return 0;

}

b.

#include <iostream>

using namespace std;

struct nodeType {

int info;

nodeType\* link;

};

int main() {

nodeType\* list;

nodeType\* ptr;

list = new nodeType;

list->info = 20;

ptr = new nodeType;

ptr->info = 28;

ptr->link = NULL;

list->link = ptr;

ptr = new nodeType;

ptr->info = 30;

ptr->link = list;

list = ptr;

ptr = new nodeType;

ptr->info = 42;

ptr->link = list->link;

list->link = ptr;

ptr = list;

while (ptr != NULL)

{

cout << ptr->info << endl;

ptr = ptr->link;

}

return 0;

}

**STACK: QUICK REVIEW**

1. A stack is a data structure in which the items are added and deleted from one end only.
2. A stack is a Last In First Out (LIFO) data structure.
3. The basic operations on a stack are as follows: Push an item onto the stack, pop an item from the stack, retrieve the top element of the stack, initialize the stack, check whether the stack is empty, and check whether the stack is full.
4. A stack can be implemented as an array or a linked list.
5. The middle elements of a stack should not be accessed directly.
6. Stacks are restricted versions of arrays and linked lists.
7. Postfix notation does not require the use of parentheses to enforce operator precedence.
8. In postfix notation, the operators are written after the operands.
9. Postfix expressions are evaluated according to the following rules:

a. Scan the expression from left to right.

b. If an operator is found, back up to get the required number of operands, evaluate the operator, and continue.

**EXERCISES**

1. Show what is output by the following segment of code:

#include <iostream>

using namespace std;

class Stack {

private:

int topIndex;

int size;

int\* stackArray;

public:

Stack(int size) : size(size), topIndex(-1) {

stackArray = new int[size];

}

~Stack() {

delete[] stackArray;

}

void push(int x) {

if (topIndex >= size - 1) {

cout << "Stack Overflow" << endl;

return;

}

stackArray[++topIndex] = x;

}

void pop() {

if (topIndex < 0) {

cout << "Stack Underflow" << endl;

return;

}

--topIndex;

}

int top() {

if (topIndex < 0) {

cout << "Stack is Empty" << endl;

return -1; // Error value, stack is empty

}

return stackArray[topIndex];

}

bool isEmpty() {

return (topIndex < 0);

}

};

void manipulateStack(Stack& stack, int& x, int& y) {

stack.push(7);

stack.push(x);

stack.push(x + 5);

y = stack.top();

stack.pop();

stack.push(x + y);

stack.push(y - 2);

stack.push(3);

x = stack.top();

stack.pop();

}

void printValues(int x, int y) {

cout << "x = " << x << endl;

cout << "y = " << y << endl;

}

void printStack(Stack& stack) {

while (!stack.isEmpty()) {

cout << stack.top() << endl;

stack.pop();

}

}

int main() {

int x = 4;

int y = 0;

// Initialize the Stack with a size of 10

Stack myStack(10);

manipulateStack(myStack, x, y);

printValues(x, y);

printStack(myStack);

return 0;

}

1. Show what is output by the following segment of code:

// Suppose that the input is: 14 45 34 23 10 5 -999

// Show what is output by the following code:

#include <iostream>

using namespace std;

class Stack {

private:

int topIndex;

int size;

int\* stackArray;

public:

Stack(int size) : size(size), topIndex(-1) {

stackArray = new int[size];

}

~Stack() {

delete[] stackArray;

}

void push(int x) {

if (topIndex >= size - 1) {

cout << "Stack Overflow" << endl;

return;

}

stackArray[++topIndex] = x;

}

void pop() {

if (topIndex < 0) {

cout << "Stack Underflow" << endl;

return;

}

--topIndex;

}

int top() {

if (topIndex < 0) {

cout << "Stack is Empty" << endl;

return -1; // Error value, stack is empty

}

return stackArray[topIndex];

}

bool isEmpty() {

return (topIndex < 0);

}

bool isFull() {

return (topIndex >= size - 1);

}

};

int main() {

// Initialize the Stack with a size of 10

Stack myStack(10);

int x;

myStack.push(5);

cin >> x;

while (x != -999)

{

if (x % 2 == 0)

{

if (!myStack.isFull())

myStack.push(x);

}

else

cout << "x = " << x << endl;

cin >> x;

}

cout << "Stack Elements: ";

while (!myStack.isEmpty())

{

cout << " " << myStack.top();

myStack.pop();

}

cout << endl;

return 0;

}

1. Evaluate the following postfix expressions:  
   a. 8 2 + 3 \* 16 4 / - =  
   b. 12 25 5 1 / / \* 8 7 + - =  
   c. 70 14 4 5 15 3 / \* - - / 6 + =  
   d. 3 5 6 \* + 13 - 18 2 / + =
2. Convert the following infix expressions to postfix notations:  
   a. (A + B) \* (C + D) - E  
   b. A - (B + C) \* D + E / F  
   c. ((A + B) / (C - D) + E) \* F - G  
   d. A + B \* (C + D) - E / F \* G + H
3. Write the equivalent infix expression for the following postfix expressions:  
   a. A B \* C +  
   b. A B + C D - \*  
   c. A B – C – D \*

**QUEUE: QUICK REVIEW**

1. A queue is a data structure in which the items are added at one end and removed from the other end.
2. A queue is a First In First Out (FIFO) data structure.
3. The basic operations on a queue are as follows: Add an item to the queue, remove an item from the queue, retrieve the first and last element of the queue, initialize the queue, check whether the queue is empty, and check whether the queue is full.
4. A queue can be implemented as an array or a linked list.
5. The middle elements of a queue should not be accessed directly.
6. If the queue is nonempty, the function front returns the front element of the queue and the function back returns the last element in the queue.
7. Queues are restricted versions of arrays and linked lists.

**EXERCISES**

1. Consider the following statements: Show what is output of the code:  
   #include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

Node\* front = NULL;

Node\* rear = NULL;

bool isEmptyQueue() {

return (front == NULL);

}

void addQueue(int item) {

Node\* newNode = new Node;

newNode->data = item;

newNode->next = NULL;

if (front == NULL) {

front = newNode;

rear = newNode;

} else {

rear->next = newNode;

rear = newNode;

}

}

int frontElement() {

return front->data;

}

void deleteQueue() {

Node\* temp = front;

front = front->next;

delete temp;

}

int main() {

int x, y;

x = 4;

y = 5;

addQueue(x);

addQueue(y);

x = frontElement();

deleteQueue();

addQueue(x + 5);

addQueue(16);

addQueue(x);

addQueue(y - 3);

cout << "Queue Elements: ";

while (!isEmptyQueue()) {

cout << frontElement() << " ";

deleteQueue();

}

cout << endl;

return 0;

}

1. Consider the following code; Suppose the input is: 15 28 14 22 64 35 19 32 7 11 13 30 -999  
   Show what is output by the code:

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

Node\* top = NULL;

Node\* front = NULL;

Node\* rear = NULL;

bool isEmptyStack() {

return (top == NULL);

}

void push(int item) {

Node\* newNode = new Node;

newNode->data = item;

newNode->next = top;

top = newNode;

}

int topElement() {

return top->data;

}

void pop() {

Node\* temp = top;

top = top->next;

delete temp;

}

bool isEmptyQueue() {

return (front == NULL);

}

void addQueue(int item) {

Node\* newNode = new Node;

newNode->data = item;

newNode->next = NULL;

if (front == NULL) {

front = newNode;

rear = newNode;

} else {

rear->next = newNode;

rear = newNode;

}

}

int frontElement() {

return front->data;

}

void deleteQueue() {

Node\* temp = front;

front = front->next;

delete temp;

}

}

int main() {

int x;

push(0);

addQueue(0);

cin >> x;

while (x != -999) {

switch (x % 4) {

case 0:

push(x);

break;

case 1:

if (!isEmptyStack()) {

cout << "Stack Element = " << topElement() << endl;

pop();

} else {

cout << "Sorry, the stack is empty." << endl;

}

break;

case 2:

addQueue(x);

break;

case 3:

if (!isEmptyQueue()) {

cout << "Queue Element = " << frontElement() << endl;

deleteQueue();

} else {

cout << "Sorry, the queue is empty." << endl;

}

break;

}

cin >> x;

}

cout << "Stack Elements: ";

while (!isEmptyStack()) {

cout << topElement() << " ";

pop();

}

cout << endl;

cout << "Queue Elements: ";

while (!isEmptyQueue()) {

cout << frontElement() << " ";

deleteQueue();

}

cout << endl;

mystery();

return 0;

}

1. What is the output of the following program?

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

Node\* front = NULL;

Node\* rear = NULL;

bool isEmptyQueue() {

return (front == NULL);

}

void addQueue(int item) {

Node\* newNode = new Node;

newNode->data = item;

newNode->next = NULL;

if (front == NULL) {

front = newNode;

rear = newNode;

} else {

rear->next = newNode;

rear = newNode;

}

}

int frontElement() {

return front->data;

}

int backElement() {

return rear->data;

}

void deleteQueue() {

Node\* temp = front;

front = front->next;

delete temp;

}

int main() {

addQueue(10);

addQueue(20);

cout << frontElement() << endl;

deleteQueue();

addQueue(2 \* backElement());

addQueue(frontElement());

addQueue(5);

addQueue(backElement() - 2);

Node\* tempFront = front;

while (tempFront != NULL) {

cout << tempFront->data << " ";

tempFront = tempFront->next;

}

cout << endl;

cout << frontElement() << " " << backElement() << endl;

return 0;

}

1. Suppose that queue is an array of size 100, and queueFront and queueRear are two integer variables representing the front and rear of the queue, respectively. The queue is implemented as a circular array. Initially, queueFront is 50 and queueRear is 99.
   1. What are the values of queueFront and queueRear after adding an element to queue?
   2. What are the values of queueFront and queueRear after removing an element from queue?
2. Suppose that queue is an array of size 100, and queueFront and queueRear are two integer variables representing the front and rear of the queue, respectively. The queue is implemented as a circular array. Initially, queueFront is 25 and queueRear is 75.
   1. What are the values of queueFront and queueRear after adding an element to queue?
   2. What are the values of queueFront and queueRear after removing an element from queue?
3. Suppose that queue is an array of size 100, and queueFront and queueRear are two integer variables representing the front and rear of the queue, respectively. The queue is implemented as a circular array. Initially, both queueFront and queueRear are 99.
   1. What are the values of queueFront and queueRear after adding an element to queue?
   2. What are the values of queueFront and queueRear after removing an element from queue?
4. Suppose that queue is implemented as an array with a special reserved slot. Also, suppose that the size of the array implementing queue is 100. If the value of queueFront is 50, what is the position of the first queue element?
5. Suppose that queue is implemented as an array with a special reserved slot. Suppose that the size of the array implementing queue is 100. Also, suppose that the value of queueFront is 74 and the value of queueRear is 99.
   1. What are the values of queueFront and queueRear after adding an element to queue?
   2. What are the values of queueFront and queueRear after removing an element from queue? Also, what is the position of the removed queue element?

**SEARCHING AND HASHING ALGORITHMS: QUICK REVIEW**

1. A list is a set of elements of the same type.
2. The length of a list is the number of elements in the list.
3. A one-dimensional array is a convenient place to store and process lists.
4. The sequential search algorithm searches the list for a given item, starting with the first element in the list. It continues to compare the search item with the elements in the list until either the item is found or no more elements are left in the list with which it can be compared.
5. On average, the sequential search algorithm searches half the list.
6. For a list of length n, in a successful search, on average, the sequential search makes comparisons.
7. A sequential search is not efficient for large lists.
8. A binary search is much faster than a sequential search.
9. A binary search requires the list elements to be in order—that is, sorted.
10. For a list of length n, in a successful search, on average, the binary search makes key comparisons.
11. Let L be a list of size n > 1. Suppose that the elements of L are sorted. If SRH(n) is the minimum number of comparisons needed, in the worst case, by using a comparison-based algorithm to recognize whether an element x is in L, then SRH(n) ≥ .
12. The binary search algorithm is the optimal worst-case algorithm for solving search problems by using the comparison method.
13. To construct a search algorithm of the order less than log2n, it cannot be comparison based.
14. In hashing, the data is organized with the help of a table, called the hash table, denoted by HT. The hash table is stored in an array.
15. To determine whether a particular item with the key, say X, is in the hash table, we apply a function h, called the hash function, to the key X; that is, we compute h(X), read as h of X. The function h is an arithmetic function, and h(X) gives the address of the item in the hash table.
16. In hashing, because the address of an item is computed with the help of a function, it follows that the items are stored in no particular order.
17. Two keys X1 and X2, such that X1 ≠ X2, are called synonyms if h(X1) = h(X2).
18. Let X be a key and h(X) = t. If bucket t is full, we say that an overflow has occurred.
19. Let X1 and X2 be two nonidentical keys. If h(X1) = h(X2), we say that a collision has occurred. If r = 1, that is, the bucket size is 1, an overflow and a collision occurs at the same time.
20. Collision resolution techniques are classified into two categories: open addressing (also called closed hashing) and chaining (also called open hashing).
21. In open addressing, data is stored within the hash table.
22. In chaining, the data is organized in linked lists, and the hash table is an array of pointers to the linked lists.
23. In linear probing, if a collision occurs at location t, then, starting at location t, we search the array sequentially to find the next available array slot.
24. In linear probing, we assume that the array is circular so that if the lower portion of the array is full, we can continue the search in the top portion of the array. If a collision occurs at location t, then starting at t, we check the array locations t, t + 1, t + 2, . . ., (t + j) % HTSize. This is called the probe sequence.
25. Linear probing causes clustering, called primary clustering.
26. In random probing, a random number generator is used to find the next available slot.
27. In rehashing, if a collision occurs with the hash function h, we use a series of hash functions.
28. In quadratic probing, if a collision occurs at position t, then starting at position t we linearly search the array at locations (t + 1) % HTSize, (t + 22 ) % HTSize = (t + 4) % HTSize, (t + 32) % HTSize = (t + 9) % HTSize, . . ., (t + i2) % HTSize. The probe sequence is: t, (t + 1) % HTSize, (t + 22) % HTSize, (t + 32) % HTSize, . . ., (t + i2) % HTSize.
29. Both random and quadratic probing eliminate primary clustering. However, if two nonidentical keys, say X1 and X2, are hashed to the same home position, that is, h(X1) = h(X2), the same probe sequence is followed for both keys. This is because random probing and quadratic probing are functions of the home positions, not the original key. If the hash function causes a cluster at a particular home position, the cluster remains under these probings. This is called secondary clustering.
30. One way to solve secondary clustering is to use linear probing, wherein the increment value is a function of the key. This is called double hashing. In double hashing, if a collision occurs at h(X), the probe sequence is generated by using the rule: (h(X) + i \* g (X)) % HTSize, where g is the second hash function.
31. In open addressing, when an item is deleted, its position in the array cannot be marked as empty.
32. In chaining, for each key X (in the item), first we find h(X) = t, where 0 ≤ t ≤ HTSize – 1. The item with this key is then inserted in the linked list (which might be empty) pointed to by HT [t].
33. In chaining, for nonidentical keys X1 and X2, if h(X1) = h(X2), the items with keys X1 and X2 are inserted in the same linked list.
34. In chaining, to delete an item, say R, from the hash table, first we search the hash table to find where in the linked list R exists. Then we adjust the pointers at the appropriate locations and deallocate the memory occupied by R.
35. Let a = (Number of records in the table / HTSize). The parameter a is called the load factor.
36. In linear probing, the average number of comparisons in a successful search is (1/2){1 + (1 / (1 – a))} and in an unsuccessful search is (1/2){1 + (1 / (1 – a)2)}.
37. In quadratic probing, the average number of comparisons in a successful search is (–log2(1 – a) ) / a and in an unsuccessful search is 1 / (1 – a).
38. In chaining, the average number of comparisons in a successful search is (1 + a / 2) and in an unsuccessful search is a.

**EXERCISES**

1. Mark the following statements as true or false.
   1. A sequential search of a list assumes that the list is in ascending order.
   2. A binary search of a list assumes that the list is sorted.
   3. A binary search is faster on ordered lists and slower on unordered lists.
   4. A binary search is faster on large lists, but a sequential search is faster on small lists.
2. Consider the following list: 63, 45, 32, 98, 46, 57, 28, 100. Using the sequential search algorithm, how many comparisons are required to find whether the following items are in the list? (Recall that by comparisons we mean item comparisons, not index comparisons.)
   1. 90
   2. 57
   3. 63
   4. 120
3. Write the definition of the class orderedArrayListType that implements the search algorithms for array-based lists .
4. Consider the following list: 2, 10, 17, 45, 49, 55, 68, 85, 92, 98, 110. Using the binary search , how many comparisons are required to find whether the following items are in the list? Show the values of first, last, and mid and the number of comparisons after each iteration of the loop.
   1. 15
   2. 49
   3. 98
   4. 99
5. Suppose that the size of the hash table is 150 and the bucket size is 5. How many buckets are in the hash table, and how many items can a bucket hold?
6. Explain how collision is resolved using linear probing.
7. Explain how collision is resolved using quadratic probing.
8. What is double hashing?
9. Suppose that the size of the hash table is 101 and items are inserted in the table using quadratic probing. Also, suppose that a new item is to be inserted in the table and its hash address is 30. If position 30 in the hash table is occupied and the next four positions given by the probe sequence are also occupied, determine where in the table the item will be inserted.
10. Suppose that the size of the hash table is 101. Further suppose that certain keys with the indices 15, 101, 116, 0, and 217 are to be inserted in this order into an initially empty hash table. Using modular arithmetic, find the indices in the hash table if:
    1. Linear probing is used.
    2. Quadratic probing is used.
11. Suppose that 50 keys are to be inserted into an initially empty hash table using quadratic probing. What should be the size of the hash table to guarantee that all the collisions are resolved?
12. Suppose there are eight students with IDs 907354877, 193318608, 132489986, 134052069, 316500320, 106500319, 116510320, and 107354878. Suppose hash table, HT, is of the size 13, indexed 0,1,2, . . ., 12. Show how these students’ IDs, in the order given, are inserted in HT using the hashing function h(k) = k % 13, where k is a student ID.
13. Suppose there are eight teachers with IDs 2733, 1409, 2731, 1541, 2004, 2101, 2168, and 1863. Suppose hash table, HT, is of the size 15, indexed 0, 1, 2, . .., 12. Show how these IDs are inserted in HT using the hashing function h(k) = k % 13, where k is an ID.
14. Suppose there are eight students with IDs 197354883, 933185971, 132489992, 134152075, 216500325, 106500325, 216510325, 197354884. Suppose hash table, HT, is of the size 19, indexed 0, 1, 2, . . ., 18. Show how these students’ IDs, in the order given, are inserted in HT using the hashing function h(k) = k % 19. Use linear probing to resolve collision.
15. Suppose there are six workers, in a workshop, with IDs 147, 169, 580, 216, 974, and 124. Suppose hash table, HT, is of the size 13, indexed 0, 1, 2, . . ., 12. Show how these workers’ IDs, in the order given, are inserted in HT using the hashing function h(k) = k % 13. Use linear probing to resolve collision.
16. Suppose there are five workers, in a shop, with IDs 909, 185, 657, 116, and 150. Suppose hash table, HT, is of the size 7, indexed 0, 1, 2, . . ., 6. Show how these workers’ IDs, in the order given, are inserted in HT using the hashing function h(k) = k % 7. Use linear probing to resolve collision.
17. Suppose there are seven students with IDs 5701, 9302, 4210, 9015, 1553, 9902, and 2104. Suppose hash table, HT, is of the size 19, indexed 0,1,2, . . .,
18. Show how these students’ IDs, in the order given, are inserted in HT using the hashing function h(k) = k % 19. Use double hashing to resolve collision, where the second hash function is given by g(k) = (k+1) % 17.
19. Suppose that an item is to be removed from a hash table that was implemented using linear or quadratic probing. Why wouldn’t you mark the position of the item to be deleted as empty?
20. What are the advantages of open hashing? 20. Give a numerical example to show that collision resolution by quadratic probing is better than chaining.
21. Give a numerical example to show that collision resolution by chaining is better than quadratic probing.
22. Suppose that the size of the hash table is 1001 and the table has 850 items. What is the load factor?
23. Suppose that the size of the hash table is 1001 and the table has 750 items. On average, how many comparisons are made to determine whether an item is in the list if:
    1. Linear probing is used.
    2. Quadratic probing is used.
    3. Chaining is used.
24. Suppose that 550 items are to be stored in a hash table. If, on average, three key comparisons are needed to determine whether an item is in the table, what should be the size of the hash table if:
    1. Linear probing is used.
    2. Quadratic probing is used.
    3. Chaining is used.

**SORTING ALGORITHMS: QUICK REVIEW**

1. Selection sort sorts a list by finding the smallest (or equivalently, the largest) element in the list, and moving it to the beginning (or the end) of the list.
2. For a list of length n, where n > 0, selection sort makes (1/2)n(n – 1) key comparisons and 3(n – 1) item assignments.
3. For a list of length n, where n > 0, on average, insertion sort makes (1/4)n2 + O(n) = O(n2) key comparisons and (1/4)n2 + O(n) = O(n2) item assignments.
4. Empirical studies suggest that for large lists of size n, the number of moves in Shellsort is in the range of n1.25 to 1.6n1.25.
5. Let L be a list of n distinct elements. Any sorting algorithm that sorts L by comparison of the keys only, in its worst case, makes at least key comparisons.
6. Both quicksort and mergesort sort a list by partitioning the list.
7. To partition a list, quicksort first selects an item from the list, called the pivot. The algorithm then rearranges the elements so that the elements in one of the sublists are less than the pivot, and the elements in the second sublist are greater than or equal to the pivot.
8. In a quicksort, the sorting work is done in partitioning the list.
9. On average, the number of key comparisons in quicksort is . In the worst case, the number of key comparisons in quicksort is O(n2).
10. Mergesort partitions the list by dividing it in the middle.
11. In mergesort, the sorting work is done in merging the list.
12. The number of key comparisons in mergesort is .
13. A heap is a list in which each element contains a key, such that the key in the element at position k in the list is at least as large as the key in the element at position 2k + 1 (if it exists) and 2k + 2 (if it exists).
14. The first step in the heapsort algorithm is to convert the list into a heap, called buildHeap. After we convert the array into a heap, the sorting phase begins.
15. Suppose that L is a list of n elements, where n > 0. In the worst case, the number of key comparisons in heapsort to sort L is . Also, in the worst case, the number of item assignments in heapsort to sort L is .

**EXERCISES**

1. Sort the following list using selection sort . Show the list after each iteration of the outer for loop. 26, 45, 17, 65, 33, 55, 12, 18
2. Sort the following list using selection sort . Show the list after each iteration of the outer for loop. 36, 55, 17, 35, 63, 85, 12, 48, 3, 66
3. Assume the following list of keys: 5, 18, 21, 10, 55, 20. The first three keys are in order. To move 10 to its proper position using insertion sort , exactly how many key comparisons are executed?
4. Assume the following list of keys: 7, 28, 31, 40, 5, 20. The first four keys are in order. To move 5 to its proper position using insertion sort , exactly how many key comparisons are executed?
5. Assume the following list of keys: 28, 18, 21, 10, 25, 30, 12, 71, 32, 58, 15. This list is to be sorted using insertion sort algorithm. Show the resulting list after six passes of the sorting phase—that is, after six iterations of the for loop.
6. Recall insertion sort for array-based lists . Assume the following list of keys: 18, 8, 11, 9, 15, 20, 32, 61, 22, 48, 75, 83, 35, 3. Exactly how many key comparisons are executed to sort this list using insertion sort?
7. Explain why the number of item movements in Shellsort is less than the number of item movements in insertion sort.
8. Consider the following list of keys: 80, 57, 65, 30, 45, 77, 27, 4, 90, 54, 45, 2, 63, 38, 81, 28, 62. Suppose that this list is to be sorted using Shellsort. Show the list during each increment.
9. Use the increment sequence 1, 3, 5
10. Use the increment sequence 1, 4, 7.
11. Both mergesort and quicksort sort a list by partitioning the list. Explain how mergesort differs from quicksort in partitioning the list.
12. Assume the following list of keys: 16, 38, 54, 80, 22, 65, 55, 48, 64, 95, 5, 100, 58, 25, 36 This list is to be sorted using quicksort algorithm. Use pivot as the middle element of the list.
13. Give the resulting list after one call to the partition procedure.
14. Give the resulting list after two calls to the partition procedure.
15. Assume the following list of keys: 18, 40, 16, 82, 64, 67, 57, 50, 37, 47, 72, 14, 17, 27, 35. This list is to be sorted using quicksort . Use pivot as the median of the first, last, and middle elements of the list.
16. What is the pivot?
17. Give the resulting list after one call to the partition procedure.
18. Convert the following array into a heap. Show the final form of the array. 47, 78, 81, 52, 50, 82, 58, 42, 65, 80, 92, 53, 63, 87, 95, 59, 34, 37, 7, 20
19. Suppose that the following list was created by the function buildHeap during the heap creation phase of heapsort. 100, 85, 94, 47, 72, 82, 76, 30, 20, 60, 65, 50, 45, 17, 35, 14, 28, 5. Show the resulting array after two passes of heapsort. Exactly how many key comparisons are executed during the first pass?
20. Suppose that L is a list is of length n and it is sorted using insertion sort. If L is already sorted in the reverse order, show that the number of comparisons is (1/2)(n2 – n) and the number of item assignments is (1/2)(n2 +3n) – 2.
21. Suppose that L is a list is of length n and it is sorted using insertion sort. If L is already sorted, show that the number of comparisons is (n – 1) and the number of item assignments is 0.

**BINARY TREES: QUICK REVIEW**

1. A binary tree is either empty or it has a special node called the root node. If the tree is nonempty, the root node has two sets of nodes, called the left and right subtrees, such that the left and right subtrees are also binary trees.
2. The node of a binary tree has two links in it.
3. A node in a binary tree is called a leaf if it has no left and right children.
4. A node U is called the parent of a node V if there is a branch from U to V.
5. A path from a node X to a node Y in a binary tree is a sequence of nodes X0, X1, . . . , Xn such that (a) X = X0, Xn = Y and (b) Xi-1 is the parent of Xi for all i = 1, 2, . . . , n. That is, there is a branch from X0 to X1, X1 to X2, . . . , Xi-1 to Xi, . . . , Xn-1 to Xn.
6. The level of a node in a binary tree is the number of branches on the path from the root to the node.
7. The level of the root node of a binary tree is 0; the level of the children of the root node is 1.
8. The height of a binary tree is the number of nodes on the longest path from the root to a leaf.
9. In an inorder traversal, the binary tree is traversed as follows:

(a) Traverse the left subtree;

(b) visit the node;

(c) traverse the right subtree.

1. In a preorder traversal, the binary tree is traversed as follows:

(a) Visit the node;

(b) traverse the left subtree;

(c) traverse the right subtree.

1. In a postorder traversal, the binary tree is traversed as follows:

(a) Traverse the left subtree;

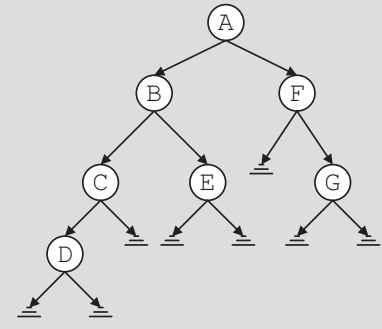
(b) traverse the right subtree;

(c) visit the node.

1. A binary search tree T is either empty or
2. T has a special node called the root node.
3. T has two sets of nodes, LT and RT, called the left subtree and the right subtree of T, respectively.
4. The key in the root node is larger than every key in the left subtree and smaller than every key in the right subtree.
5. LT and RT are binary search trees.
6. To delete a node from a binary search tree that has both left and right nonempty subtrees, first its immediate predecessor is located, then the predecessor’s info is copied into the node, and finally the predecessor is deleted.
7. A perfectly balanced binary tree is a binary tree such that
8. The heights of the left and right subtrees of the root are equal.
9. The left and right subtrees of the root are perfectly balanced binary trees

**EXERCISES**

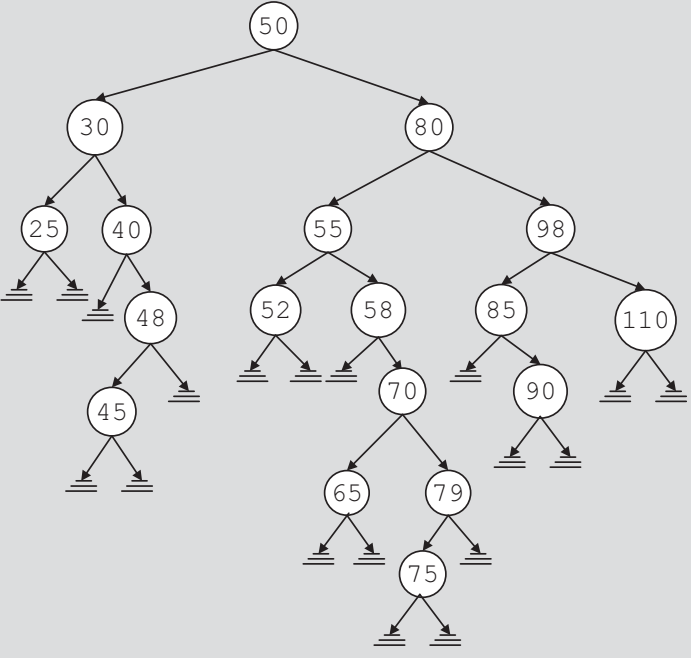
1. Mark the following statements as true or false.
2. A binary tree must be nonempty.
3. The level of the root node is 0.
4. If a tree has only one node, the height of this tree is 0 because the number of levels is 0.
5. The inorder traversal of a binary tree always outputs the data in ascending order.
6. There are 14 different binary trees with four nodes. Draw all of them. The binary tree of Figure 11-34 is to be used for Exercises 3 through 8.



Binary tree for Exercises 3 through 8

1. Find LA, the node in the left subtree of A.
2. Find RA, the node in the right subtree of A.
3. Find RB, the node in the right subtree of B.
4. List the nodes of this binary tree in an inorder sequence.
5. List the nodes of this binary tree in a preorder sequence.
6. List the nodes of this binary tree in a postorder sequence.

The binary tree of Figure 11-35 is to be used for Exercises 9 through 13.



Binary tree for Exercises 9 through 13

1. List the path from the node with info 80 to the node with info 79.
2. A node with info 35 is to be inserted in the tree. List the nodes that are visited by the function insert to insert 35. Redraw the tree after inserting 35.
3. Delete node 52 and redraw the binary tree.
4. Delete node 40 and redraw the binary tree.
5. Delete nodes 80 and 58 in that order. Redraw the binary tree after each deletion.
6. Suppose that you are given two sequences of elements corresponding to the inorder sequence and the preorder sequence. Prove that it is possible to reconstruct a unique binary tree.
7. The nodes in a binary tree in preorder and inorder sequences are as follows:

preorder: ABCDEFGHIJKLM

inorder: CEDFBAHJIKGML

Draw the binary tree.

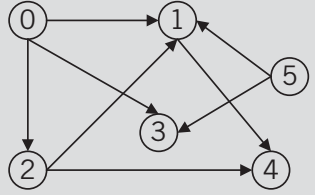
1. Given the preorder sequence and the postorder sequence, show that it might not be possible to reconstruct the binary tree.

**GRAPHS : QUICK REVIEW**

1. A graph G is a pair, G = (V, E), where V is a finite nonempty set, called the set of vertices of G and E ⊆ V x V, called the set of edges.
2. In an undirected graph G = (V, E), the elements of E are unordered pairs.
3. In a directed graph G = (V, E), the elements of E are ordered pairs.
4. Let G be a graph. A graph H is called a subgraph of G if every vertex of H is a vertex of G and every edge in H is an edge in G.
5. Two vertices u and v in an undirected graph are called adjacent if there is an edge from one to the other.
6. An edge incident on a single vertex is called a loop.
7. In an undirected graph, if two edges e1 and e2 are associated with the same pair of vertices {u, v}, then e1 and e2 are called parallel edges.
8. A graph is called a simple graph if it has no loops and no parallel edges.
9. Let e = (u, v) be an edge in an undirected graph G. The edge e is said to be incident on the vertices u and v.
10. A path from a vertex u to a vertex v is a sequence of vertices u1, u2, . . ., un such that u = u1, un = v, and (ui, ui+ 1) is an edge for all i = 1, 2, . . ., n - 1.
11. The vertices u and v are called connected if there is a path from u to v.
12. A simple path is a path in which all the vertices, except possibly the first and last vertices, are distinct.
13. A cycle in G is a simple path in which the first and last vertices are the same.
14. An undirected graph G is called connected if there is a path from any vertex to any other vertex.
15. A maximal subset of connected vertices is called a component of G.
16. Suppose that u and v are vertices in a directed graph G. If there is an edge from u to v, that is, (u, v) ˛ E, we say that u is adjacent to v and v is adjacent from u.
17. A directed graph G is called strongly connected if any two vertices in G are connected.
18. Let G be a graph with n vertices, where n > 0. Let V(G) = {v1, v2, . . ., vn}. The adjacency matrix AG is a two-dimensional n n matrix such that the (i, j)th entry of AG is 1 if there is an edge from vi to vj; otherwise, the (i, j)th entry is 0.
19. In an adjacency list representation, corresponding to each vertex v is a linked list such that each node of the linked list contains the vertex u and (v, u) ˛ E(G).
20. The depth-first traversal of a graph is similar to the preorder traversal of a binary tree.
21. The breadth-first traversal of a graph is similar to the level-by-level traversal of a binary tree.
22. The shortest path algorithm gives the shortest distance for a given node to every other node in the graph.
23. In a weighted graph, every edge has a nonnegative weight.
24. The weight of the path P is the sum of the weights of all the edges on the path P, which is also called the weight of v from u via P.
25. A (free) tree T is a simple graph such that if u and v are two vertices in T, there is a unique path from u to v.
26. A tree in which a particular vertex is designated as a root is called a rooted tree.
27. Suppose T is a tree. If a weight is assigned to the edges in T, T is called a weighted tree.
28. If T is a weighted tree, the weight of T, denoted by W(T), is the sum of the weights of all the edges in T.
29. A tree T is called a spanning tree of graph G if T is a subgraph of G such that V(T) = V(G)—that is, if all the vertices of G are in T.
30. Let G be a graph and V(G) = {v1, v2, . . ., vn}, where n ≥ 0. A topological ordering of V(G) is a linear ordering vi1, vi2, . . ., vin of the vertices such that if vij is a predecessor of vik, j ≠ k, 1 ≤ j, k ≤ n, then vij precedes vik, that is, j < k in this linear ordering.
31. A circuit is a path of nonzero length from a vertex u to u with no repeated edges.
32. A circuit in a graph that includes all the edges of the graph is called an Euler circuit.
33. A graph G is said to be Eulerian if either G is a trivial graph or G has an Euler circuit.

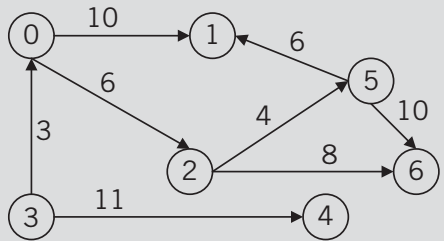
**EXERCISES**

Use the following graph for Exercises 1 through 4.



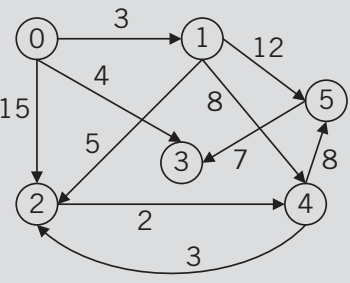
Graph for Exercises 1 through 4

1. Find the adjacency matrix of the graph.
2. Draw the adjacency list of the graph.
3. List the nodes of the graph in a depth-first traversal.
4. List the nodes of the graph in a breadth-first traversal.
5. Find the weight matrix of the following graph.



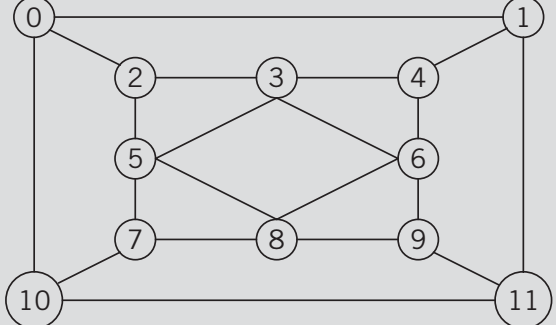
Graph for Exercise 5

1. Consider the following graph in. Find the shortest distance from node 0 to every other node in the graph.



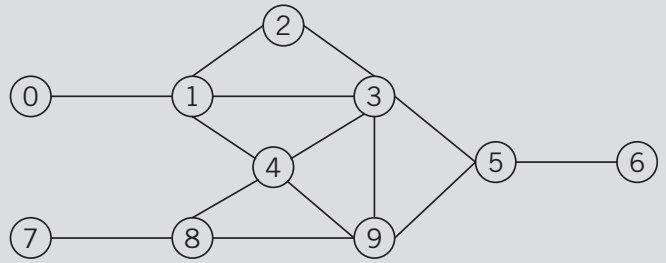
Graph for Exercise 6

1. Find a spanning tree in the following graph



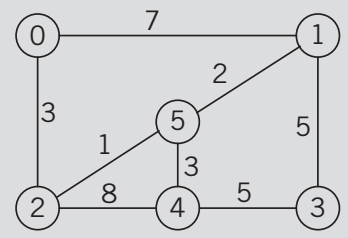
Graph for Exercise 7

1. Find a spanning tree in the following graph.



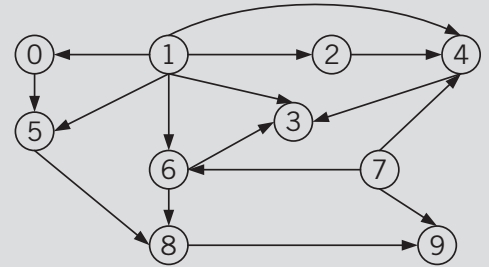
Graph for Exercise 8

1. Find the minimum spanning tree for the following graph using the Prim’s and Kruskal’s algorithms.



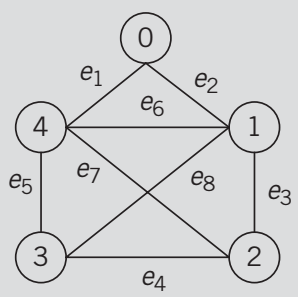
Graph for Exercise 9

1. List the nodes of the following graph in a breadth-first topological ordering.



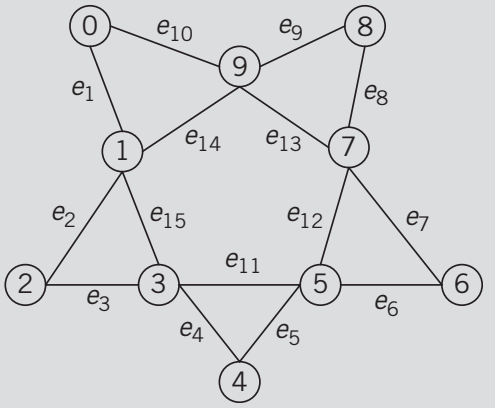
Graph for Exercise 10

1. Describe whether the following graph has an Euler circuit. If the graph has an Euler circuit, find one such circuit.



Graph for Exercise 11

1. Describe whether the following grap has an Euler circuit. If the graph has an Euler circuit, find one such circuit.



Graph for Exercise 12