Chapter 12 : Data Structures

A programming language has various data types in order to store the data used in a program. The language like C allows creating user defined types. However, using just these data types it may become really tricky and complicated to code some real life applications or system programs. The problem is related to storage structure and efficient operations on that data. The solution to this problem Is *Data Structures.* Data structures are concepts and hence are not restricted to any particular programming language. They can be coded using any programming language.

A data structure in brief, is an organized collection of data. It can be seen as the collection of the data along with some relationship among the data elements and some operations that can be used to manipulate the data. An array or a structure can be viewed as examples of basic data structures. They can be said as *fixed size* data structures as generally the size is specified at compile time. Any data structure that uses dynamic memory allocation, like linked–lists can be said as dynamic data structures.

This chapter focuses on the basic data structures like stacks, queues, linked lists and sorting, searching algorithms.

> The chapter mainly focuses on to the concepts of data structure and tries to simplify them. To avoid complexity of the code, the global variables are used heavily even though it is not standard programming practice. Obviously readers can use structures to combine these variables and rewrite the programs to follow standard programming style.

The Stack

A stack is a Last In First out *(LIFO)* data structure i.e. lastly inserted element comes out first. There are many applications requiring the use of stacks. The most striking use of a stack is the stack section of the process that is used for function calls. To achieve *LIFO* technique, the elements in the stack are entered and deleted from the stack from the top of the stack. The following program implements a stack of integer values.

Example 12.1 : Stack of Integers

```
#include <stdio.h>
#define MAX 5

int info[MAX];                                  /*To store the elements*/
int top;                                      /*To maintain TOP of the stack*/
/*function decalarations*/
void init();                                        /*To initialize the queue.*/
int isFull();                                     /*To check if the stack is full.*/
int isEmpty();                                   /*To check if the stack is empty.*/
void push(int);                                /*To enter the elements in the stack.*/
void pop();                                   /*To delete the elements from the stack.*/
int peek();                                 /*To get the top element without deleting.*/

int main()
{
      int choice, value;
```

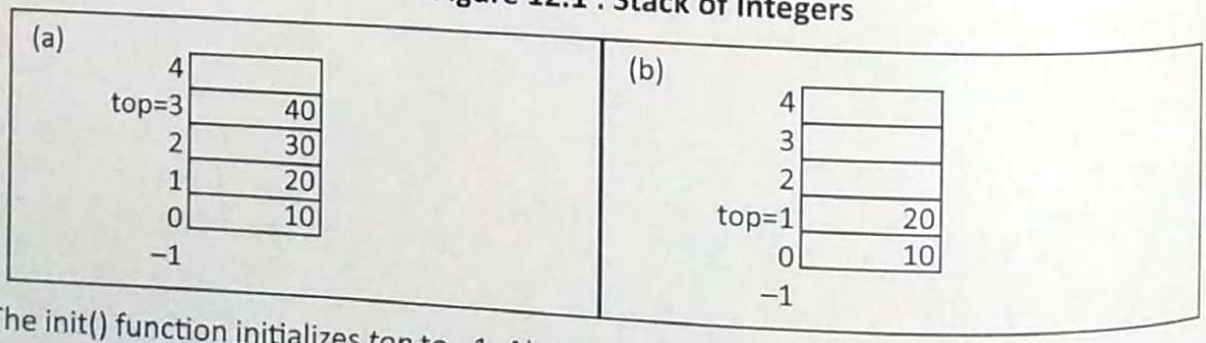
```
    init();
    do
    {
        printf("\n\n0.exit\n1.push\n2.pop\nenter choice : ");
        scanf("%d", &choice);
        switch(choice)
        {
        case 1:/*push*/
            if(isFull())
                printf("stack is full\n");
            else
            {
                printf("enter the value to push : ");
                scanf("%d", &value);
                push(value);
                printf("value pushed : %d\n", value);
            }
            break;
        case 2: /*pop*/
            if(isEmpty())
                printf("stack is empty\n");
            else
            {
                value = peek();
                pop();
                printf("value poped : %d\n", value);
            }
            break;
        }
    }while(choice!=0);
    return 0;
}

/*function implementations*/
/* To initialize the stack.*/
void init()
{
    int i;
    top = -1;                    /*Initialize the top to -1 */
        /*optionally initialize all the elements to -1*/
    for(i=0; i<MAX; i++)
        info[i] = -1;
}
/* To check if the stack is full.*/
int isFull()
{
    if(top == MAX-1)
```

```
        return 1;                          /* stack is full*/
    else
        return 0;                        /* stack is not full*/
}
/* To check if the stack is empty.*/
int isEmpty()
{
    if(top == -1)
        return 1;                          /* stack is empty*/
    else
        return 0;                        /*stack is not empty*/
}
/* To enter the elements in the stack.*/
void push(int data)
{
    top = top + 1;                          /*increment top*/
    info[top] = data;                     /*add data to array*/
}
/* To delete the elements from the queue. */
void pop()
{
    info[top] = -1;     /*optional : initialize array element as -1*/
    top = top - 1;                                  /*decrement  top*/
}
/* To get the top element without deleting.*/
int peek()
{
   return info[top];              /*returns topmost value of stack */
}
```

If values pushed into stack are 10, 20, 30, 40 the stack will look as fig 12.1 (a) and after removing two elements from the stack (40, 30) the stack will look as fig 12.1 (b).

Figure 12.1 : Stack of Integers



The init() function initializes *top* to –1. Also one can initialize all elements to some uniform value (in this code all elements are initialized to –1).

The isFull() function returns 0 if stack is not full and 1 if the stack is full i.e. if *top* is equal to MAX–1.

The isEmpty() function returns 0 if stack is not empty and 1 if the stack is empty i.e. if *top* is equal

to –1.

The push() function increments the value of the top by 1 every time push() is called and places the data at the *top* location in the stack's info array.

The pop() function assign –1 at the *top* of array (giving the effect of deletion of the element) and then decrements the value of *top* by 1.

Note that push() and pop() functions do not check full and empty conditions of the stack. Hence these functions must be called after checking the appropriate code as shown in main().

The peek() function returns the element at *top* from *info* array without removing it from the stack.

The code in main() test all the functions of stack into different cases of switch statement. It is a simple menu driven program.

Applications of Stack

- Stack section of the process is used to call functions.
- Stack is used to convert given infix expression to prefix or postfix notation.
- Stack is used to evaluate prefix or postfix form of expression.
- Stack is used for implementing some complex algorithms into advanced data structures (e.g. depth first search for graphs)

Infix, Prefix and Postfix form of expression

The arithmetic expressions can be expressed into different forms. The *infix* form of expression is written in the C programs. However, compiler cannot solve them in the same form. Usually compiler converts this expression to *prefix* or *infix* form. The machine level code generated by the compiler use the algorithm of solving *prefix* or *postfix* form of expression. These forms can be briefly explained as follows :

Expression Form	Representation	Description
Infix	a + b	Here the *operator* is sandwiched between *operands*.
Postfix	ab +	Here the *operator* follows *operands*.
Prefix	+ ab	Here the *operator* precedes the *operands*.

The Queue

Like stack, queue also is a collection of elements, but unlike stack queue follows First in First Out (FIFO) technique. The elements can be entered in the queue from one end and deleted from other end. The end from which the elements are inserted into queue is known as *rear* end of the queue and the end from which the elements are deleted from the queue is known as *front* end of the queue. This means that the removal of elements from a queue is possible in the same order in which the insertion of elements is made into the queue. The different types of queues are possible given as :

- Linear Queue
- Circular Queue
- Priority based Queues

Linear Queue

In a linear queue the elements are entered from rear end of the queue and deleted from the *front* end of the queue. Because of its linear nature, there is restriction on movement of rear and *front* as they can be only increased. So the maximum value of rear or front can be MAX–1. The following program explains the concept of linear queue.

Example 12.2 : Linear Queue of Integers

```
#include<stdio.h>
#define MAX 5

int info[MAX];        /* To store the elements*/
int rear; /*To maintain rear end of the queue*/
int front;     /* To maintain front end of the queue */

/*function decalarations*/
void init();                          /* To initialize the queue.*/
int isFull();                      /* To check if the queue is full.*/
int isEmpty();                    /* To check if the queue is empty.*/
void push(int);                /* To enter the elements in the queue.*/
void pop();                 /* To delete the elements from the queue.*/
int peek();              /* To get the rear element without deleting.*/

int main()
{
  int choice, value;
  init();
  do
  {
    printf("\n\n0.exit\n1.push\n2.pop\nenter choice : ");
    scanf("%d", &choice);
    switch(choice)
    {
    case 1:/*push*/
      if(isFull())
          printf("queue is full\n");
      else
      {
          printf("enter the value to push : ");
          scanf("%d", &value);
          push(value);
          printf("value pushed : %d\n", value);
      }
      break;
    case 2: /*pop*/
      if(isEmpty())
          printf("queue is empty\n");
      else
      {
          value = peek();
          pop();
          printf("value poped : %d\n", value);
      }
      break;
```

```
    }
  }while(choice!=0);
  return 0;

}
/*function implementations*/
/* To initialize the queue.*/
void init()

{
  int i;
  rear = -1;    /* Initalize the  rear to -1 */
  front = -1;  /* Initialize the front to -1 */
  /*optionally initialize all the elements to -1*/
  for(i=0; i<MAX; i++)
    info[i] = -1;

}
/* To check if the queue is full.*/
int isFull()

{
if (rear == MAX-1)
     return 1;                                    /* queue is full*/
  else
        return 0;                                /*queue is not full*/

}
/* To check if the queue is empty.*/
int isEmpty()
{
    if(rear==front)
    return 1;                                      /*queue is empty*/
     else
        return 0;                                /*queue is not empty*/
}
/* To enter the elements in the queue.*/
void push(int data)
{                                                  /*increment rear*/
    rear = rear +1;                              /*add data to array*/
    info[rear] = data;
}
/*To delete the elements from the queue.*/
void pop()
{                                                 /*decrement  rear*/
    front = front +1;
info[front] =-1;     /*optional : initialize array element as -1*/
}
/* To get the rear element without deleting.*/
int peek()
{
    returns info[front+1] ;     /*returns value at front end of queue*/
}
```

If values pushed into queue are 10, 20, 30, 40 the queue will look as fig 12.2 (a) and after removing two elements from the queue (10,20) the stack will look as fig 12.2 (b).

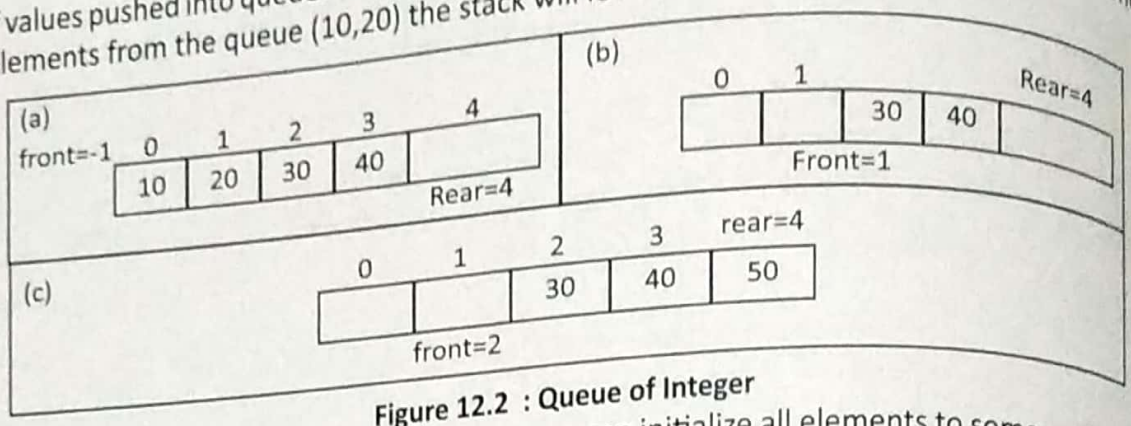


Figure 12.2 : Queue of Integer

The init() function initializes *top* to –1. Also one can initialize all elements to some uniform value (in this code all elements are initialized to –1).

The is Full() function returns 0 if queue is not full and 1 if the queue is full i.e. if *rear* is equal to MAX–1.

The is Empty() function returns 0 if queue is not empty and 1 if the queue is empty i.e. if *rear* equals *front*. :

The push() function increments the value of the *rear* by 1 every time push() is called and places the data at the *rear* location in the queue's info array.

The pop() function increments *front* by 1 and then assign –1 to that locations (giving the effect of deletion of the element) and then decrements the value of *top* by 1.

Note that push() and pop() functions do not check full and empty conditions of the stack. Hence these functions must be called after checking the appropriate code as shown in main().

The peek() function returns the value to be deleted in next pop() call i.e. *info[front+1]*.

Since rear or front can only be incremented, the linear queue has a serious limitation or drawback. Due to linear nature this there might be situations, where queue may have empty places but cannot be used for inserting new data. For example, if the five elements are added In above queue (*rear* will reach to MAX–1) and two elements are deleted from the queue (see figure 12.2 (c)). Now even though two places are empty in the queue (at the start of queue), new values cannot be added as queue full condition is reached i.e. *rear==MAX–1*. This limitation can be overcome by the circular queue.

Circular Queue

Circular queue behaves in same manner as linear queue. However, restriction on movement of front and rear is removed. If rear or front reaches to MAX–1, they can be reinitialized to zero (this is not allowed in case of linear queue). Assuming that front and rear are initialized to zero and queue can have maximum 5 elements, the following figure shows increment of rear. Even front can be incremented in similar manner.

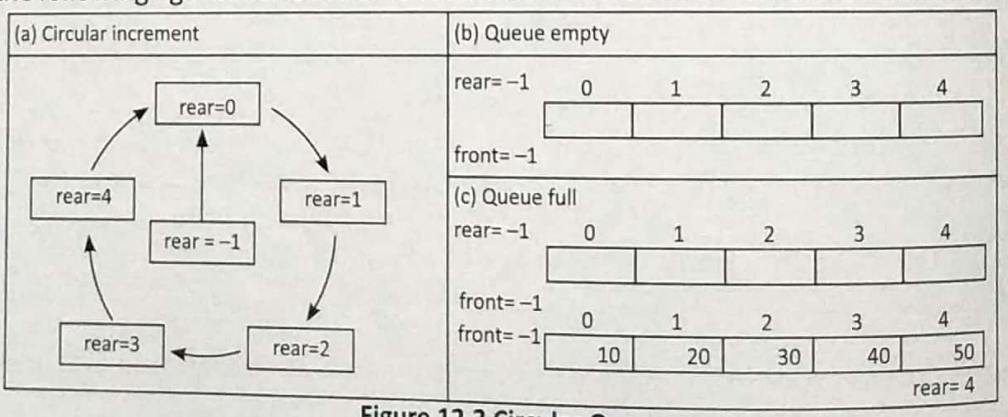


Figure 12.3 Circular Queue

The implementation of circular queue is given in program 12.3.

Example 12.3 : Circular Queue of Integers

```
#include <stdio.h>
#define MAX 5

int info[MAX]; /* To store the elements  */
int rear; /*To maintain rear end of the queue*/
int front;     /* To maintain front end of the queue */

/*function declarations*/
void init();    /* To initialize the queue.*/
int isFull();   /* To check if the queue is full.*/
int isEmpty(); /* To check if the queue is empty.*/
void push(int);     /* To enter the elements in the queue.*/
void pop();     /* To delete the elements from the queue.*/
int peek();     /* To get the rear element without deleting.*/

int main()
{
    int choice, value;
    init();
    do
    {
        printf("\n\n0.exit\n1.push\n2.pop\nenter choice : ");
        scanf("%d", &choice);
        switch(choice)
        {
        case 1:/*push*/
            if(isFull())
                Printf("queue is full\n");
            else
            {
                printf("enter the value to push : ");
                scanf("%d", &value);
                push(value);
                printf("value pushed : %d\n", value);
            }
            break;
        case 2: /*pop*/
            if(isEmpty())
                printf("queue is empty\n");
            else
            {
                value = peek();
                pop();
                printf("value poped : %d\n", value);
            }
            break;
        }
    }while(choice!=0);
```

```
    return 0;
}

/*function implementations*/
/* To initialize the queue.*/
void init()
{
    int i;
    rear = -1;      /*Initialize the  rear to - */
    front = -1;     /*Initialize the front to -1 */
    /*optionally initialize all the elements to -1*/
    for(i=0; i<MAX; i++)
        info[i] = -1;
}
/* To check if the queue is full.*/
int isFull()
{
    if((front==-1 && rear==MAX-1) || (rear==front && rear!=-1))
        return 1; /*queue is full*/
    else
        return 0; /*queue is not full*/
}
/* To check if the queue is empty.*/
int isEmpty()
{
    if(rear==-1 && front==-1)
        return 1; /* queue is empty*/
    else
        return 0; /*queue is not empty*/
}
/* To enter the elements in the queue.*/
void push(int data)
{
    if(rear == MAX-1)
        rear = 0;
    else
        rear = rear +1; /*increment rear*/
    info[rear] = data;  /*add data to array*/
}
/* To delete the elements from the queue. */
void pop()
{
    if(front== MAX-1)
        front = 0;
    else
        front = front +1;   /*decrement rear*/
        info[front] = -1;   /*optional : initialize array element as
-1*/
    if(front==rear)
    {
        front = -1;
        rear = -1;
```

```
    }
}
/* To get the rear element without deleting.*/
int peek()
{   if(front == MAX-1)
        return info[0];
    else
        return info[front+1]; /*returns value at front end of queue*/
}
```

Explanation

The init() function initializes the entire elements to –1 along with *rear* and *front*. The isFull() function returns 1 if queue is full else it returns 0. The queue is full if

- The *rear* is at last position and *front* is at starting position (front==–1 && rear==MAX–1).
- The *front* equals to *rear*. This condition may occur when rear is reinitialized to 0 after reaching up to MAX–1.

The push() function insert the element in queue at the *rear* location after incrementing the *rear*. If *rear* is *MAX–1*, then it is reinitialized to zero.

The pop() function deletes the elements from the *front* location. It increments the value of *front* and if *front* is *MAX–1*, then it is reinitialized to zero. Note that after deletion if queue is found to be empty (*front* equals *rear*), then both are reset to –1.

Finally peek() function returns the element to be deleted by the next pop() operation i.e. *info[front+1]*.

Priority Queue

Queues follow first in first out concept. However, sometimes queues follow some special technique. In this case, some priority value is associated with each element of the queue. Instead of FIFO, the element with highest priority is removed from the queue. These queues may store the elements in the sorted manner. These queues are very frequently used in operating systems.

Applications of Queue

- Printer maintains the queue of the documents to be printed.
- Queues are mainly used in operating systems for CPU scheduling algorithms.
- Windows operating system use the message queue, which is an example of priority queue.
- Stack is used for implementing some complex algorithms in to advanced data structures (e.g. depth first search for graphs)

Central Processing Unit (CPU) Scheduling

The multitasking operating systems allows execution of multiple programs simultaneously. When system has single processor, the same processor executes all the processes. Now the operating system is responsible for allocating CPU cycles for each process, so that it can be executed properly. Different algorithms used by the operating systems for the CPU scheduling. The two examples are :

- **FCFS (First Come First Serve)** : The CPU executes the processes in the same sequence in which they are started executing.
- **SJF (Shortest Job First)** : The CPU executes first the process that needs minimum time for execution and so on.

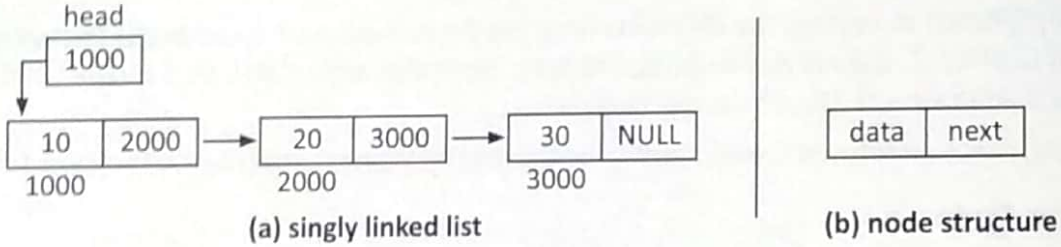
Linked List

Linked list is list of records linked together. Each item in the linked list is called as a *node*. Node has two parts, *data* and *address*. The data part can be data of any type i.e. built–in type or user defined type. Node of *singly linked list* has only one pointer to store address of next node, while node of *doubly linked list* has two pointers to store address of next as well as previous node. Obviousy *singly linked list* allows *unidirectional* access, while doubly linked list allows *bidirectional* access.

The nodes in the linked list are allocated dynamically as and when required and similarly are released after it serves its purpose. So unlike arrays, list can dynamically grow or shrink in size. The starting node of the linked list is known as *header node*. The address of this node is stored in a pointer known as *head pointer*. Optionally the address of last node is stored in a pointer known as *tail pointer*. Maintaining the *tail pointer* can be useful to make certain operations more efficient e.g. adding a node at the end of list. However, it increases the complexity while implementing other algorithms. The programs in this chapter use head pointer only. This pointer is declared globally and is initialized to NULL. NULL address in head pointer means that, list does not contain any node.

Singly linked list

Every node in singly linked list stores some *data* and the *address* of the next node (figure 12.3 (b)). Because there is no node after the last node, the *address* part of the last node has *NULL* address.



(a) singly linked list (b) node structure

Figure 12.4 Linear Singly linked list

Following program 12.4 implement most commonly done operations on the linked list.

Example 12.4 Linear Singly Linked List

```
#include <stdio.h>
#include <stdlib.h>

/*structure declaration*/
typedef struct node
{
    int info; /*data part of node*/
    struct node *next; /*address of the next node*/
}NODE;

/*global pointers to store the starting and ending address*/
NODE *head=NULL;

/*function declrations*/
void displayList();                          /*display all data in list*/
NODE* createNode(int val);      /*allocate a node and initialize it*/
```

```
void addFirst(int val);          /*add the node at the start of list*/
void addLast(int val);           /*add the node at the end of list*/
void insert(int pos, int val);   /*add the node at the given position in list*/
int delFirst();                  /*del the node at first position*/
void del(int pos);               /*del node at given position in list*/
void delAll();                   /*del all elements from the list*/

int main()
{
    int choice, value, pos;
    do
    {
      printf("n\n1.exit\n1.display\n2.add first\n3.add last");
      printf("\n4.insert at pos\n5.del first\n6.del at pos\n7.del all");
      printf("\nenter choice :");
      scanf("%d", &choice);
      switch(choice)
      {
      case 1: /*display*/
          displayList();
          break;
      case 2: /*add first*/
          printf("enter a value : ");
          scanf("%d", &value);
          addFirst(value);
          break;
      case 3: /*add last*/
          printf("enter a value : ");
          scanf("%d", &value);
          addLast(value);
          break;
      case 4: /*insert at pos*/
          printf("enter a value : ");
          scanf("%d", &value);
          printf("ender position : ");
          scanf("%d", &pos);
          insert(pos, value);
          break;
      case 5: /*del first*/
          delFirst();
          break;
      case 6:  /*del at post*/
          printf("enter position : ");
          scanf("%d", &pos);
          del(pos);
          break;
      case 7: /*del all*/
```

```
            delAll();
            break;
        }
    }while(choice!=0);
    delAll();  /*avoid memory leakage*/
    return 0;
}

/*function implementations*/
/*display all data in list*/
void displayList()
{
    NODE *trav;
    printf("LIST :: ");
    trav=head;      /*init trav to start of list*/
    while(trav!=NULL)
    {
        printf("%d ->",trav->info);    /*print node data*/
        trav=trav->next;          /*go to next node*/
    }
    printf("\n");
}
/*allocate a node and initialize it*/
NODE* createNode(int val)
{
  NODE *newnode;
  newnode=(NODE*)malloc(sizeof(NODE)); /*allocate memory for node*/
  newnode->info = val;                 /*initialize node data*/
  newnode->next=NULL;                  /*initialize next pointer to NULL*/
  return newnode;
}
/*add the node at the start of list*/
void addFirst(int val)
{
  NODE *newnode;
  newnode=createNode(val);
  if(head==NULL)    /*if list is empty*/
    head=newnode;   /*newnode is first node*/
  else
  {
    newnode->next=head;  /*add the node before head/first node*/
    head=newnode;  /*consider newnode as head now*/
  }
}
/*add the node at the end of list*/
void addLast(int val)
{
```

```
    NODE *newnode, *trav;
    newnode=createNode(val);
    if(head==NULL)    /*if list is empty*/
      head=newnode;  /*newnode is first node and*/
    else
    {
      trav=head;
      while(trav->next!= NULL)
        trav=trav->next;
    trav->next=newnode;  /*add the node after tail/last node*/
  }
}
/*add the node at the given position in list*/
void insert(int pos, int val)
{
  int i;
  NODE *trav, *newnode;
    if(head==NULL || pos<=1)  /*if pos is 1 or before*/
      addFirst(val); /*add node at the start of list*/
  else
  {
    trav = head;  /*initialize trav to start of list*/
    /*traverse to the node before given pos*/
    for(i=1; i<pos-1; i++)
    {
      trav = trav->next;
      if(trav==NULL)              /*if reach to end of list*/
      {
        printf("invalid position\n");
        return;                   /*cannot insert the element*/
      }
    }
    newnode = createNode(val);  /*create new node*/
    /*add the node after trav node*/
    newnode->next = trav->next;
    trav->next = newnode;
  }
}
/*del the node at first position*/
int delFirst()
{
    NODE *temp;
    int val=0;
    if(head==NULL)                /*if list is empty*/
    {
      printf("list empty\n");    /*print message*/
      return 0;
```

```
    }
    val = head->info;    /*read value of first node*/
    temp = head;    /*store address of first node*/
    head = head->next;    /*take head to the next node*/
    free(temp);    /*free memory of first node*/
    return val;    /*return the value of deleted node*/
}
/*del node at given position in list*/
void del(int pos)
{
    int i;
    NODE *trav, *temp;
    if(head==NULL || pos<=1)    /*if pos is 1 or before*/
      delFirst();               /*delete node at the start of list*/
  else
  {
    trav = head;  /*initialize trav to start of the list*/
    /*traverse to the node before given pos*/
    for(i=1; i<pos-1; i++)
    {
      if(trav==NULL)  /*if reach to end of list*/
      {
        printf("invalid position\n");
          return;                 /*cannot delete the node*/
      }
      trav = trav->next;
    }
    temp=trav->next; /*tempt is the node to be deleted*/
    trav->next=temp->next; /*link trav to the node after temp*/
    free(temp);    /*free memory of temp node*/
  }
}
/*del all elements from the list*/
void delAll()
{
    NODE *temp;
    while(head!=NULL)    /*repeat till all nodes are not deleted*/
    {
      temp = head;              /*store address of first node*/
      head=head->next;          /*take head to next node*/
      free(temp);               /*delete first node*/
    }
}
```

The main() function implement a menu driven program and depending on user choice, corresponding function is called with appropriate arguments. The function *createNode()* is used allocate the memory required for the NODE and then initialize it. This function is called for creating a new node. The program is split into well defined functions, where each function defines an operation on linked list. Each function has number of comments to describe the code written. These operations discussed in brief below.

Display data in list

displayList() displays all the element of the list. Visiting all the nodes in the linked list is also called as *traversing* the linked list. Pointer *trav* is used for traversing here. Note that the last node stores address NULL in *next* pointer as shown in figure 12.3.

Add node at start of list

addFirst() creates the node with given value using function createNode() and add it to the first position in the list. If list do not contain any node, *head* will keep the address of newly created node. In other cases, node will be added at the start of list. Obviously *head* pointer will be updated to keep address of this node, as *head* pointer always keeps address of the first node.

Add node at end of list

addLast() creates the node with given value using function createNode() and add it to the last position in the list. If list do not contain any node, *head* will keep the address of newly created node. In other cases, node will be added at the end of list. The pointer *trav* is used traverse to the last node of the list and then new node is connected to it.

Add node at given position in list

insert() creates the node with given value using function createNode() and add it to the given position in the list. If list do not contain any node or position is given as 1, it simply delegate his job to the addFirst() function. In other cases, *trav* pointer is used to reach to the node before the given position. Now the new node is linked to the node after *trav* node and address of new node is stored in trav. Note that function can add the node at any position from 1 to last position in the list, However, giving any position after that will cause the error message *invalid position*.

Delete node at start of list

delFirst() deletes the node at the first position. If list is empty, the error message will be printed. In other cases, the address of first node is kept in *temp* pointer. After deletion of this node, next node would be the header node, so *head* pointer should now store address of next node. Finally memory of the node pointed by *trav* is released.

Delete node at given position in list

del() deletes the node at the given position in the list. If list do not contain any node or position is given as 1, it simply delegate his job to the delFirst() function. In other cases, *trav* pointer is used to reach to the node before the given position. The pointer *temp* points to the node after *trav*, which is the node to be deleted. Now *temp* node can be linked with the node after that and memory of the node pointed by *trav* is released. Note that function can delete the node at any position from 1 to last position in the list, However, giving any position after that will cause the error message *invalid position*.

Delete all nodes in list

delAll() deletes all the nodes in the list. The same logic of delFirst() is repeated in the loop till, all nodes are not deleted. This function must be called before program is terminated to delete all the nodes from memory, so that there will not be any memory leakage.

Dynamic stack or linked representation of stack

Linear singly linked list can be used to create a dynamic stack also known as linked representation of stack. In fact, addFirst() operation will perform the *push* operation of the stack and delFirst() operation will perform *pop* operation of the stack. Other two main operations isEmpty() and peek() can be implemented as given below.

```
/*check if stack is empty*/
int isEmpty()
{
  if(head==NULL)
    return 1; /*stack i.e. list is empty*/
  else
    return 0; /*stack i.e. list is empty*/
}
```

```
/*read top most element of stack*/
int peek()
{
    /*return data of first node*/
    return head->info;
}
```

Note that, this stack do not need the implementation of isFull() as the memory will be allocated at runtime when required. There will not be the size limitation like array implementation of the stack.

Dynamic queue or linked representation of queue

Linear singly linked list can be used to create a dynamic queue also known as linked representation of stack. In fact, addLast() operation will perform the *push* operation of the queue and delFirst() operation will perform *pop* operation of the queue. Other two main operations isEmpty() and peek() can be implemented as given below.

```
/*check if queue is empty*/
int isEmpty()
{
   if(head==NULL)
     return 1;  /*queue i.e. list is empty*/
   else
     return 0; /*queue i.e. list is empty*/
}
```

```
/*read front element of queue*/
int peek()
{
    /*return data of first node*/
    return head->info;
}
```

Note that, this queue do not need the implementation of isFull() as the memory will be allocated at runtime when required. There will not be the size limitation like array implementation of the queue. Ideally this queue must be implemented using linked having *head* and *tail* pointers in order to make *push* operation i.e. addLast() more efficient than its implementation shown in program 12.4.

Doubly linked list

Every node in dolubly linked list stores some data and the *addresses* of the next and previous nodes (figure 12.4 (b)). Because there is no node after the last node, the *next* part of the last node has *NULL* address. Similarly there is no node before the first node, so the *prev* part of the first node has *NULL* address (figure 12.4 (a)).

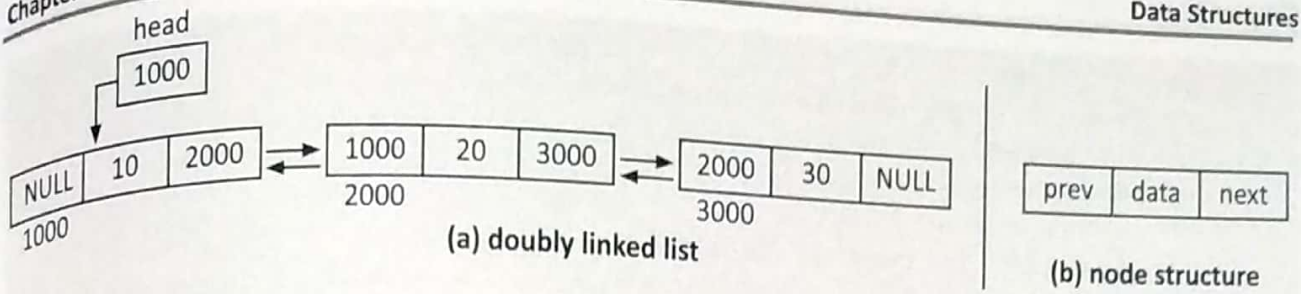


Figure 12.5 Doubly linked list

Following program 12.5 implement most commonly done operations on the linked list. Note that the program contains only those operations, which are different than singly linked list. The remaining functions can be copied from the singly linked list program 12.4. It is expected that one must be comfortable with singly linked list and its implementation before trying implementation of doubly linked list. The program is self explanatory with the help of comments.

Program 12.5 Doubly linked list

```
#include <stdio.h>
#include <stdlib.h>

/*structure declaration*/
typedef struct node
{
    int info; /*data part of node*/
    struct node *next;  /*address of the next node*/
    struct node *prev;  /*address of the next node*/
}NODE;

/*global pointers to store the starting and ending address*/
NODE *head=NULL;
/*function declarations: copy from program 12.4*/

/*main() copy from program 12.4*/

/*function implementations*/
/*display all data in list in forward and reverse order*/
void displayList()
{
    NODE *trav;
    printf("FWD :: ");
    if(head==NULL)/*if list is empty return*/
        return;
    trav=head;      /*init trav to start of list*/
    while(trav!=NULL)
    {
        printf("%d ->",trav->info);    /*print node data*/
        trav=trav->next;     /*go to next node*/
    }
```

```
    printf("\nREV :: ");
    /*traverse up to the last node in list*/
    trav = head;
    while(trav->next!=NULL)
        trav = trav->next;
    /*print list in reverse order*/
    while(trav!=NULL)
    {
    printf("%d ->", trav->info);  /*print node data*/
        trav=trav->prev;     /*go to prev node*/

    }
    printf("\n");
}
/*allocate a node and initialize it*/
NODE* createNode(int val)
{
    NODE *newnode;
    newnode=(NODE*)malloc(sizeof(NODE));  /*allocate memory for node*/
    newnode->info = val;      /*initialize node data*/
    newnode->next=NULL; /*initialize next pointer to NULL*/
    newnode->prev=NULL;/*initialize prev pointer to NULL*/
    return newnode;
}
/*add the node at the start of list*/
void addFirst(int val)
{
    NODE *newnode;
    newnode=createNode(val);
    if(head==NULL) /*if list is empty*/
        head=newnode;  /*newnode is first node*/
    else
    {
        newnode->next=head; /*add the node before head/first node*/
        head->prev=newnode; /*add new node before head node*/
        head=newnode;  /*consider newnode as head now*/
    }
}
/*add the node at the end of list*/
void addLast(int val)
{
    NODE *newnode, *trav;
    newnode=createNode(val);
    if(head==NULL) /*if list is empty*/
        head=newnode;  /*newnode is first node and*/
    else
    {
        trav=head;
```

```
        while(trav->next!=NULL)
            trav=trav->next;
        trav->next=newnode; /*add the node after last node*/
            newnode->prev = trav; /*the last node will be before new node*/
    }
}
/*add the node at the given position in list*/
void insert(int pos, int val)
{
    int i;
    NODE *trav, *newnode, *travnext;
    if(head==NULL || pos<=1) /*if pos is 1 or before*/
        addFirst(val); /*add node at the start of list*/
    else
    {
        trav = head;    /*initialize trav to start of list*/
        /*traverse to the node before given pos*/
        for(i=1; i<pos-1; i++)
        {
            trav = trav->next;
            if(trav==NULL) /*if reach to end of list*/
            {
                printf("invalid position\n");
                    return;    /*cannot insert the elements*/
            }
        }
        newnode = createNode(val);     /*create new node*/
        travnext = trav->next;
            /*add the node between trav and travnext node*/
            newnode->prev = trav;
        newnode->next = travnext;
        trav->next = newnode;
            if(travnext!=NULL)
                travnext->prev = newnode;
    }
}
/*del the node at first position*/
int delFirst()
{
    NODE *temp;
    int val=0;
    if(head==NULL) /*if list is empty*/
    {
        printf("list empty\n");  /*print message*/
        return 0;
    }
    val = head->info;    /*read value of first node*/
```

```
        temp = head;    /*store address of first node*/
        head = head->next;  /*take head to the next node*/
        if(head!=NULL)
                head->prev = NULL;  /*the prev of first node must be
NULL*/
        free(temp);     /*free memory of first node*/
        return val;     /*return the value of deleted node*/
}
/*del node at given position in list*/
void del(int pos)
{
        int i;
        NODE *trav;
        if(head==NULL || pos<=1) /*if pos is 1 or before*/
                delFirst();     /*delete node at the start of list*/
        else
        {
                trav = head;    /*initialize trav to start of the list*/
                /*traverse to the node at given pos*/
                for(i=1; i<pos; i++)
                {
                        if(trav==NULL) /*if reach to end of list*/
                        {
                                printf("invalid position\n");
                                return;    /*cannot delete the node*/
                        }
                        trav = trav->next;
                }
                /*now trav is the node to be deleted*/
                /*link temp's next to trav's prev and vice versa*/
                trav->prev->next = trav->next;
                        if(trav->next!=NULL)
                                trav->next->prev=trav->prev;
                free(trav);     /*free memory of trav node*/
        }
}
/*del all elements from the list*/
void delAll()
{
        NODE *temp;
        while(head!=NULL)    /*repeat till all nodes are not deleted*/
        {
                temp = head;    /*store address of first node*/
                head=head->next;     /*take head to next node*/
                free(temp);     /*delete first node*/
        }
}
```

Singly Circular linked list

If last node of the singly linked list keeps the address of first node, it is called as singly circular linked list (see figure 12.@). The advantage of such a list is that from any node we can traverse to any node because of its circular nature.

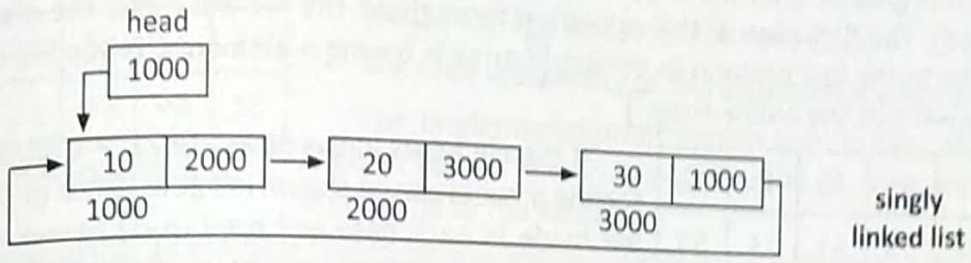


Figure 12.6 Circular

Doubly Circular linked list

If *next* pointer in last node of the doubly linked list keeps the address of first node and *prev* pointer in first node keeps the address of last node, it is called as doubly circular linked list (see figure 12.@). the advantage of such a list is that from any node we can traverse to any node because of its circular nature. Also the last node can be accessed immediately from the first node using head–>*prev*, which is much faster way as compared to traversing til! the last node.

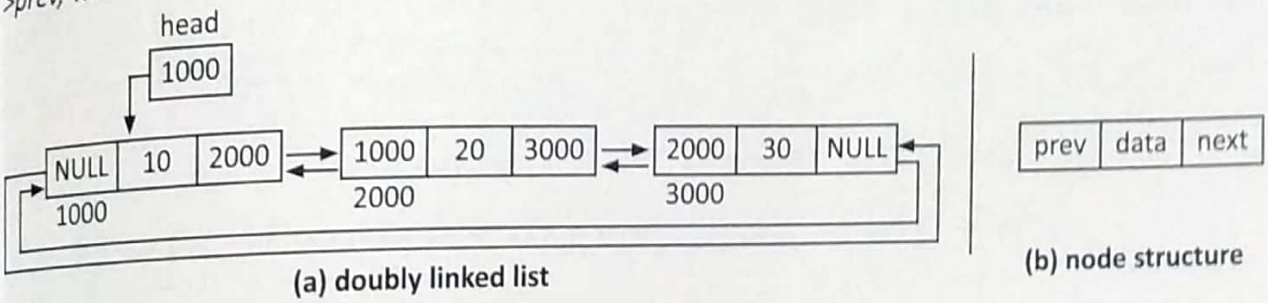


Figure 12.7 Doubly Circular linked list

Sorting Algorithms

Arranging the data in certain order is a very common task. Arranging the data in ascending or descending order is called as *sorting*. There are many techniques available for sorting an array–based list of data elements. These techniques differ in their *time and space complexities*. Some of the simple sorting techniques are discussed here.

Time complexity is the measure of time required to execute a given algorithm. Time required for any sorting and searching algorithm is proportional to the number of comparisons done in the algorithm. Time complexity is expressed in terms of *Big O* notation e.g. $O(n^2)$, O (n log n). $O(n^2)$ means that time complexity is of order of n^2, where n is number of elements in the collection. During some sorting algorithms there is possible case that list contains the element are arranged in such fashion that number of comparisons will be minimum. This case is referred as *best case*. Opposite to this, in the *worst case* maximum number of comparisons will be done. Generally, average case is considered while calculating time complexities.

Space complexity is the measure of space required to execute a given algorithm. Space complexity mainly required two components, the memory required for the code and required for the data. Note that data can be stored in form of local variables, global variables or even in dynamically allocated memory block. All these components must be considered while understanding space complexity of the algorithm.

Bubble Sort

Bubble sorting is a simple sorting technique in which elements are arranged by forming pairs of adjacent elements i.e. each pair contains i^{th} and $(i+1)^{th}$ element. For sorting in ascending order, if i^{th} element is greater than the $(i+1)^{th}$ element then the elements are swapped (i.e. positions are interchanged). The first pass of this operation throughout the list will move the element with the highest value to the last position in the list. If array is having n elements, repeating the process for (n–1) times will sort the entire array.

	22	44	33	55	11
Pass 1	22	33	44	11	55
Pass 2	22	33	11	44	55
Pass 3	22	11	33	44	55
Pass 4	11	22	33	44	55

Figure 12 .8 Bubble Sort

If the elements in the array are 22, 44, 33, 55, 11 then the sorting process can be shown in figure. Since (n–1) comparisons are made in each pass and total (n–1) passes are done, total number of comparisons will be $(n-1)*(n-1) = n^2 - 2n + 1$. The implementation of bubble sort in form of C function is given below. The array and the size of array are passed as arguments to the function. The comments in the snippet should guide for its implementation. The function can be called from any other function according to the requirement, by passing base address of array of integers and its size. With little efforts one can change the code to work for any other built in or user defined data type.

```
int bubble(int a[], int n)
{
    int i, j;
    int temp;
    for(i=1; i<n; i++)  /*repeat for n-1 passes*/
    {
        for(j=0; j<n-1; j++)     /*repeat commparison of consecutive
elements*/
        {
            if(a[j] > a[j+1])    /*compare elements*/
            {    /*swap a[j] and a[j+1]*/
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}
```

Selection Sort

The selection sort compares a selected element with all the remaining elements and then swap it with the smaller element i.e. In an array of *n* elements, i^{th} element is compared with all the elements starting from $(i+1)^{th}$ element till the last element and if j^{th} element is smaller than i^{th} element, both are interchanged.

If the elements in the array are 22, 44, 33, 55, 11 then the sorting process can be shown in figure 12.9. The first pass of selection sort compares 0^{th} element with remaining elements causing

	22	44	33	55	11
Pass 1	11	44	33	55	22
Pass 2	11	22	44	55	33
Pass 3	11	22	33	55	44
Pass 4	11	22	33	44	55

Figure 12 .9 Selection Sort

(n–1) comparisons. In next pass 1st element is compared with remaining elements after that, so the number of comparisons would be (n–2) and so on. Total number of comparisons is calculated as :

$(n-1) + (n-2) + (n-3) + \ldots\ldots + 1 = n(n-1)/2$

The time complexity of this algorithm is also $O(n^2)$.

The implementation of bubble sort in form of C function is given below. The array and the size of array are passed as arguments to the function.

```
int selection(int a[], int n)
{
    int i, j;
    int temp;
    for(i=0; i<n-1; i++)      /*select element from 0 to n-1*/
    {
        for(j=i+1; j<n; j++)      /*compare selected element with re-
maining elements*/
        {
            if(a[i] > a[j])      /*compare elements*/
            {    /*swap a[i] and a[j]*/
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

Time complexities of sorting algorithms

The time complexities of the sorting algorithms are discussed earlier. The worst case and average case time complexities for other well known sorting algorithms are listed in the chart below for reference and comparison.

Table 12.1 Complexity of Sorting Algorithms

sorting algorithm	average case	worst case
selection sort	$O(n^2)$	$O(n^2)$
bubble sort	$O(n^2)$	$O(n^2)$
insertion sort	$O(n^2)$	$O(n^2)$
quick sort	O(n log n)	$O(n^2)$
merge sort	O(n log n)	O(n log n)
heap sort	O(n log n)	O(n log n)

Searching Algorithms

Like sorting, searching is another most commonly used algorithm. Obviously searching in ordered

list would be more efficient as compared to searching in unordered array. There are two most widely used searching algorithms, *linear search* and *binary search*.

Linear Search

If the collection is not sorted, then linear or sequential search is used to search the element in the collection. The linear search proceeds by sequentially comparing the *key* (element to be searched) with each element in the list. If *key* is matched with the element, the index of the element is returned. If *key* do not match with any of the element, some invalid index (–1) can be returned.

In the best case, the search procedure terminates after one comparison only, whereas in the worst case, it will do *n* comparisons. On average, it will do approximately *n/2* comparisons will be performed. The time complexity is expressed as O(n). The implementation of linear search in form of C function is given below. The array, size of array and key to be searched are passed as arguments to the function.

```
int linearSearch(int a[], int n, int key)
{
     int i;
     for(i=0; i<n; i++)  /*go through all elements*/
     {
          if(a[i]==key)  /*if i the element is matchingkey*/
               return i; /*return index i*/
     }
     return -1;      /*return invalid index -1*/
}
```

This function can be called from any other function according to the requirement. If return value of the function is –1, means that element is not found. The typical code for calling the function can be given as follows.

```
int arr[] = {22,11,33,55,44};
int index, key=33;  /*assuming key to search is 33, or can be accepted
from user*/
index = linearSearch(arr, 5, key);
if(index==-1)
     printf("key not found\n");
else
     printf("key found at index %d\n", index);
```

Finally one should remember that linear search can be applied to sorted collection too. However, it will not be that efficient.

Binary Search

If the collection is sorted, then binary search is used to search the element in the collection in much faster way as compared to linear search. The *key* will be first compared with the middle element of the list. If it is matching then index of the element can be written. Otherwise, assuming that array is ascending order, if the *key* is less than middle element, the next attempt to search must be made in left partition and if the *key* is greater than middle element, the next attempt to search must be made in right partition. Now the middle element of decided partition must be located and comparison process continues. If *key* do not match with any of the element, some invalid index (–1)

can be returned.

The number of comparisons in binary search can be maximum *(log n)/(log 2)*. The time complexity is expressed as O(log n). The implementation of linear search in form of C function is given below. The array, size of array and key to be searched are passed as arguments to the function, so function can be called just similar as that of linearSearch() as discussed above.

```
int binarySearch(int a[], int n, int key)
{
  /*left keeps the left min index and right keeps max index of the array*/
  int left=0, right=n-1, mid;
  /*partition of array is valid if and if its min index <= max index*/
  while(left<=right)/*till partition is valid*/
  {
    mid = (left+right)/2;/*find midpoint of array*/
    if(key==a[mid]) /*if key is matching middle element*/
      return mid;   /*return its index*/
    if(key > a[mid])/*if key is greater than middle element*/
      left = mid+1; /*focus right parittion i.e. mid+1 to right index*
    else          /*if key is less than middle element*/
      right = mid-1;/*focus left partition i.e. left to mid-1 index*/
    }
    return -1; /*return invalid index -1*/
}
```

This was fundamentals from the data structure which includes stack, queues, linked lists and sorting, searching algorithms. Obviously more advanced data structures are available like trees, graphs or hash tables but they are out of scope of this text.

Lab Exercise

Q. 1. Write a program to accept a string from user. Push all characters in the string into stack or characters. Print the string in reverse order by using stack operations.

Q. 2. Define a structure student having data members rollno, name, marks. Write a program to implement queue of students.

Q. 3. Write a program to implement singly linked list of structure student.

Q. 4. Write a program to sort array of structure student.

Q. 5. Write a program to search a student by rollno or name according to user choice, in the array of structure student.

Objective Questions

Q.1. A stack is _____ data structure

1) Last in Last out
2) Last in First out
3) First in Last out
4) First in First out

Q.2. A queue is _____ data structure

1) Last in Last out
2) Last in First out
3) First in Last out
4) First in First out

Q.3. A linked list is _____ data structure

1) Static
2) Dynamic
3) Static or Dynamic
4) Global

Introduction to data structures:

Data Structures: It is the way to store data in a organized manner such that operations can be performed on it efficiently.

Algorithm: It is finite set of instructions, which if followed, accomplishes, solution of a given problem.

Performance analysis or Analysis of an algorithm: it is the task of determining how much computer time and space is required for an algorithm to run to completion.

There are two measures to do analysis of an algorithm

1. Time
2. Space

Time Complexity of an algorithm is the amount of computer time it needs to run to completion.

Space Complexity of an algorithm is the amount of memory it needs to run to completion.

Best Case Time Complexity of an algorithm is the minimum time it needs to run to completion.

Worst Case Time Complexity of an algorithm is the maximum time it needs to run to completion.

When time required for an algorithm to run to completion, is neither minimum nor maximum then it is called as **Average Case Time Complexity.**

Time Complexities of some important algorithms:

Name of an Algorithm	Best Case Time Complexity	Worst-Case Time Complexity	Average Case Time Complexity
1. Linear Search	$\Omega(1)$	$O(n)$	$\theta(n)$
2. Binary Search	$\Omega(1)$	$O(\log n)$	$\theta(\log n)$
3. Selection Sort	$\Omega(n^2)$	$O(n^2)$	$\theta(n^2)$
4. Bubble Sort	$\Omega(n)$	$O(n^2)$	$\theta(n^2)$
5. Insertion Sort	$\Omega(n)$	$O(n^2)$	$\theta(n^2)$
6. Quick Sort	$\Omega(n\log n)$	$O(n^2)$	$\theta(n\log n)$
7. Merge Sort	$\Omega(n\log n)$	$O(n\log n)$	$\theta(n\log n)$

TREES:

Introduction to Trees:

Tree is **non-linear advanced data structure** which has **one or more finite set of elements** in which **the first specially designated element/node is called as root node/element,** and all other nodes are connected to it in **heirachical manner**, whereas there is **parent-child relatioship** between them.

The tree with no nodes is called as the **null** or **empty tree.**

Degree of the Node: is the number of subtrees of the node in a given tree. OR **no. of child nodes for any node is called as degree of that node.**

Degree of the Tree: It is the **maximum degree of any node** in a given tree

Leaf Node: a node with **degree 0** is called as **leaf node** or **terminal node.** OR the node which do not have child node is called as leaf node. (or **outer node/external node**)

Non-Leaf Node: a node whose **degree is not zero** OR the node have a child node/s is called non-leaf node or **non-terminal node.(or internal node/inner node).**

Ancestors: all the nodes in the path from the root to that node, are called ancestors of that node

Descendants: all the node accessible from the given node are called as descendants of that node

Siblings: child nodes of the same parent are called as **siblings** or **brothers.**

Level of the Node: the tree is structured in different levels. The entire tree is leveled in such a way that **the root node is at level 0**, then its immidiate children are at level 1 and their children are at level 2.

Level of Child Node = level of its parent node + 1

Depth of the Tree: It is maximum level of any node in a given tree.

Height of the Node : Node is also used to denote Position of node in tree.

Hight Of Node = Max Hight Of Left Subtree And Hight Of Right Subtree +1

Forest: a forest is a set of n >= 0 disjoint trees.

Binary Tree: it is a tree in which every node has maximum two child nodes.

Binary Search Tree: it is a binary tree in which, left child is always smaller than its parent and right child is always greater or equal to its parent.

Complete Binary Tree: binary tree in which all leaf nodes are at same level

Strictly Binary Tree: binary tree in which each non-leaf node has exactly two no. Of child nodes.

Full Binary Tree: binary tree with its full capacity for the given height, i.e. addiing one more node into the tree will increase the height of the tree.

It is always stritly binary tree as well as complete.

no. of elements = 2^{h-1}-1

Left Skewed Binary Tree:binary tree in which only left link is used to keep the address of child node(right link of each node is kept NULL).

Right Skewed Binary tree:binary tree in which only right link is used to keep the address of child node(left link of each node is kept NULL).

Balanced Binary Search Tree:

the binary search tree with minimum possible height(for given no. of elelements) is called as balanced binary search tree.

Balance Factor of a Node = Height of the Left Subtree – Height of Right Subtree.

in a balance binary search tree if balance factor of each node is either **-1 or 0 or 1** then it is called as balanced binary search tree.

AVL tree: self balancing binary search tree

invented by "Greogy Adelson-Velsky" and "E.M. Landis".

Tree Traversal: to visit each node/element in a tree atmost once is called as traversal.

There are three tree traversal methods:

1. Preorder:(visit-left-right)

->start tree traversal from root node

->first visit the current node, if the node having left subtree then goto it left subtree, after complete traversal of its left subtree goto its right subtree.

->root element is always at the first position in this traversal.

2. Inorder:(left-visit-right)

->start tree traversal from root node

->if the node having left subtree then goto it left subtree, after complete the traversal of its left subtree visit the node, and then goto its right subtree.

->this traversal always visits elements in ascending order. for binary search tree.

3. Postorder:(left-right-visit)

->start tree traversal from root node

->if the node having left subtree then goto it left subtree, then if it is having right subtree goto its right subtree.

->after complete traversal of its left subtree as well as right subtree, visit the node.

->root element is always at the last position in this traversal.

GRAPHS :

Graph: graph is non-linear advanced data structure, which has

- finite set of vertices also called as nodes and
- finite set of ordered/unordered pair of vertices called as edges, which may contains weight/ cost/value.

->When edges are ordered pair of vertices graph is called as **directed graph**or **di-graph**, edges are unordered pair of vertices then graph is called as **undirected graph.**

->if there is an direct edge between two vertices then those vertices are called as **adjacent vertices** otherwise they are called as **non-adjacent vertices.**

-> Path : is a set of edges between two vertices

-> Cycle :

Connected Graph: if path exists between any two vertices in a graph then those vertices are called as **connected vertices.** If each vertex is connected with remaining all vertices then the graph is called as **connected graph.**

Complete Graph: if each vertex in a graph is adjacent to remaining all vertices, such a graph is called as complete graph.

In a given graph for any path if the start vertex and end vertex are same, such path is called as **cycle.**

Spanning Tree: it is a sub graph of a graph that can be formed after removing one or more edges so that subgraph remains connected and does not contains a cycle.

->given graph may has multiple spanning trees.

->the sum of weights of all edges is the weight of the graph.

->minimum spanning tree is a spanning tree of a graph having minimum weight.

->for a di-graph **in degree** of vertex is the no. Of vertices incident on it and **outdegree** of vertex is no. of vertices originated from it.

Representation of Graph:

Graph can be represented by two ways:

1. Adjacency Matrix (using 2D array)
2. Adjacency List (using linked list)

1. Adjacency Matrix (using 2D array):

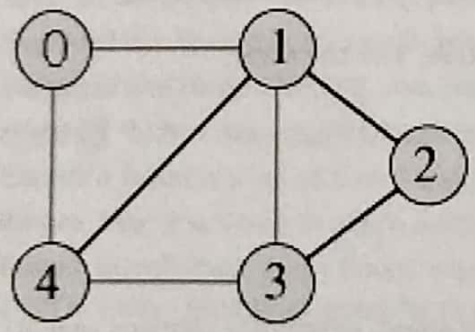
->Adjacency matrix is 2D array of size V*V where V is the number of vertices in a graph.

->Let the 2D array be: adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.

->Adjacency matrix for undirected graph is always symmetric matrix.

->Adjacency matrix is also used to represent weighted graphs. If adj[i][j]=w, then there is an edge from vertex i to vertex j with weight w. Non adjacent vertices are represented as ∞

Example: Un-directed Un-weighted Graph: G V = { 0,1,2,3,4 } and E = { (0,1), (0,4),(1,2), (1,3), (1,4), (2,3), (3,4) }.



MATRIX REPRESENTATION OF ABOVE GRAPH:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

2. Adjacency List (using linked list):

->An array of linked list is used.

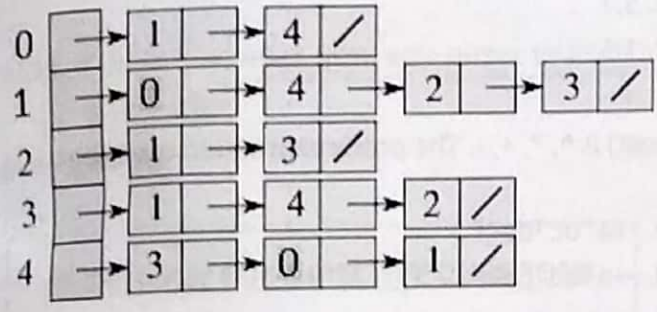
->Size of the array is equal to number of vertices

->Let the array be arr[]. An entry of arr[i] represents ith vertex, and the linked list tree cartains vertices adjacent to the i^{th} vertex.

->This representation can also used to represent a weighted graph.

->The weights of edges can be stored in nodes of linked lists.

->Following is the **adjacency list representation** of the above graph.



FOREST:

->A Forest is an undirected graph, all of whose components are trees.

->In another words, the graph consists of a disjoint union of trees.

->Equivalently, a forest is an undirected cycle free graph.

Objective Questions:

Q. 1. what is the worst case time complexity of quick sort algorithm

a. O(n/2)
b. O(2n)
c. O(n)
d. none of above

$O(n^2)$

Q.2. what is the average time complexity of linear search algorithm

a. O(n)
b. O(nlog n)
c. O(n/2)
d. O(n^2)

Q.3. time complexity of an algorithm/program is

a. only execution time of a program
b. only compilation time of a program
c. execution time + compilation time
d. none of above

Q.4. which sorting algorithm is efficient for array having small size

a. quick sort
b. insertion sort
c. bubble sort
d. merge sort

Q.5. which of the following time complexities are generally same

a. best case & average case
b. worst case & average case
c. best case, average case & worst case
d. none of above

Q.6. The following postfix expression with single digit operands is evaluating using a stack:

8 2 3 ^ / 2 3 * + 5 1 * -

note that ^ is the exponential operator. The top two elements of the stack after the first * is evaluated are:

a. 6,1
b. 5,7
c. 3,2
d. 1,5

Q.7. The order of precedence (from highest to lowest) is ^, *, +, -. The prefix expression corresponding to the infix expression a+b*c-d^e^f is

a. -+a*bc^^def
b. -+a*bc^de^f
c. ^+a*bc^-def
d. -+a^bc*^def

Q.8. which of the following tree traversal method gives the elements in a bst in descending order

a. postorder traversal
b. preorder traversal
c. inorder traversal
d. none of above

Q.9. binary search tree is balanced if

a. each node has exactly two child nodes
b. all leaf nodes are at same level
c. all nodes are balanced
d. root node is balanced

Q.10. what is the time complexity for searching an element in a binary search tree

a. O(n)
b. O(log n)
c. O(n^2)
d. O(2n)

Q.11. Which of the following statement is false

graph consists a cycle

a. every tree is a bipartite graph
b. a tree contains a cycle
c. a tree with n nodes contains n-1 edges
d. a tree is a connected graph

Q.12. Given an undirected graph G with V vertices and E edges the sum of the degrees of all vertices is

a. E
b. 2E
c. V
d. 2V

Q.13.Traversal of a graph is different from tree because

a. there can be a loop in graph so we must maintain a visited flag for every vertex

b. dfs of a graph uses stack, but inorrder traversal of a tree is recursive
c. bfs of a graph uses queue, but a time efficient BFS of a tree is recursive.
d. all of the above

Q.14. spanning tree of a graph contains minimum ____ no. of edges (where V = no. of vertices)

a. V-1
b. V+1
c. V
d. none of above

Q.15. if there is an edge from any vertex to itself it is called as

a. self edge
b. cycle
c. loop
d. circuit

ANSWER KEY FOR OBJECTIVE QUESTIONS

Q.1	D	**Q.6**	A	**Q.11**	B
Q.2	A	**Q.7**	A	**Q.12**	B
Q.3	C	**Q.8**	D	**Q.13**	A
Q.4	B	**Q.9**	C	**Q.14**	A
Q.5	B	**Q.10**	B	**Q.15**	C