Data Structure

Introduction to data structure

Basic concept: - The study of data structures involves two complementary goals. The first goal is to identify and develop useful mathematical entities and operations and to determine what classes of problems can be solved by using these entities and operations. The second goal is to determine representations for those abstract entities and to implement the abstract operations on these concrete representations.

Abstract data types: - A useful tool for specifying the logical properties of a data type is the abstract data type or ADT. Fundamentally, a data type is a collection of values and a set of operations on those values.

The technique of creating new data types that are well suited to an application to be programmed is known as data abstraction. It provides the ability to create user-defined data types for modeling a real world object, having the properties of built-in- data types and a set of permitted operators. The class is a construct in C++ for creating user-defined data types called abstract data types.

Fundamental and derived data types

Primitive data structure

Array:

Address calculation using column and row major ordering: - A two-dimensional array is a logical data structure that consist of rows and columns. Physically in hardware there is no such facility. Any array must be stored in the linear memory of a computer, so it is one-dimensional array. A single address is used to retrieve a particular item from memory. To implement a two-dimensional array, it is necessary to develop a method of ordering its elements in a linear fashion and of transforming a two-dimensional reference to the linear representation.

One method of representing a two-dimensional array in memory is the row-major representation. Under this representation, the first row of the array occupies the first set of memory locations reserved for the array, the second row occupies the next set, and so forth. Let us suppose that a two-dimensional integer array is stored in row-major sequence and let us suppose that, for an array ar, base(ar) is the address of the first element of the array. That is, if ar is declared by

int ar[r][c] where r is no. of row and c is no. of column and base(ar) is the address of ar[0][0]. We also assume that esize is the size of each element in the array. Let us

calculate the address of an arbitrary element, ar[i][j]. Since the element is in row i, its address can be calculated by computing the address of the first element of row i and adding the quantity j*esize. But to reach the first element of row i it is necessary to pass through i complete rows, each of which contains r elements, so that the address of the first element of row i is at base(ar) + i * r * esize. Therefore the address of ar[i][j] is at base(ar) + (i * r + j) * esize.

Various operations on array: -

Matrix multiplication:

Polynomial representation and addition: -

Sparse Matrix: -

Stack and Queues:

Stack: - A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the top of the stack. A single end of the stack is designated as the stack top. New items may be put on top of the stack, or items, which are at the top of the stack, may be removed. It means that which item inserted last that is removed first. This system is called Last In First Out (LIFO).

The operation by which we can insert new item into stack top is called Push operation and by which we can remove items from the top of the stack is called Popup operation. When the stack become full then we cannot apply push operation and when it is empty we cannot apply popup operation.

Program1: Implementation of Stack using array.

```
#include<stdio.h>
#include<conio.h>
#define max 5
struct stack
{
  int top;
  char x[max];
};
main()
{
   struct stack stk;
   void push(struct stack*, char);
   char popup(struct stack*);
   int isempty(struct stack*);
   int isfull(struct stack*);
```

```
    int ans; char c;
    stk.top=-1;
    clrscr();
    do
    {
          printf("\n\n1 : PUSH\n\n");
          printf("2 : POPUP\n\n");
          printf("3 : Exit\n\n");
          printf("Enter your choice : ");
          scanf("%d",&ans);
          fflush(stdin);



          switch(ans)
          {
             case 1 : if(isfull(&stk))
                           printf("\nStack is Full, cannot insert");
                      else
                      {    printf("\nEnter a charater to insert");
                           scanf("%c",&c);
                           push(&stk,c);
                      }
                      break;
             case 2 : if (isempty(&stk))
                           printf("\nStack is empty, cannot delete"):
                      else
                           printf("\nThe deleted character is %c",popup(&stk));
                      break;
          }
    }
    while(ans<3);
}
    int isempty(struct stack *s)
    {     return(s->top<0);
    }
    int isfull(struct stack *s)
    {
      return(s->top==max);
    }
    void push(struct stack *s, char a)
    {
            s->x[++s->top]=a;
    }
    char popup(struct stack *s)
    {
          return(s->x[(s->top--)]);
    }
```

Linked-list implementation of Stack

```
 #include<stdio.h>
#include<conio.h>
struct node
{
   char data;
   struct node *next;
};
main()
{
   struct node *push(struct node *, char);
   char popup(struct node *);
   struct node *top=NULL;
   int a; char c;
   clrscr();
   do
   {
          printf("\n\n 1 : Push\n\n");
          printf("2 : PopUp\n\n");
          printf("Enter your choice : ");
          scanf("%d",&a);
          fflush(stdin);
          switch(a)
          {
            case 1 : printf("\n\nEnter a character : ");
                     scanf("%c", &c);
                     top=push(top,c);
                     break;
            case 2 : if(top == NULL)
                           printf("\n\nStack is empty\n\n");
                     else
                     {       c= popup(top);
                           printf("\n\nRemoved data = %c\n\n",c);
                            top=top->next;
                     }
                     break;
          }
   }while(a<3);
}
struct node *push(struct node *t, char x)
{
  struct node *s=malloc(sizeof(struct node));
  s->data=x;
  (t==NULL) ? t->next=NULL : s->next=t;
  return(s);
}
char popup(struct node *t)
{
```

```
  return(t->data);
}
```

Queue: - A queue is an ordered collection of items. Items may be inserted at one end (called rear) and may be deleted in another end (called front). In queue, which element inserted first that element has to be deleted first. So it is called First In First Out (FIFO). The following operation can be applied on a queue i) check for empty queue ii) check for queue full iii) insertion iv) remove

Queue using Array

```
#include<stdio.h>
#include<conio.h>
#define max 5
struct queue
{
  char data[max];
  int front,rear;
};
main()
{
   void insert(struct queue *, char);
   char del(struct queue *);
   int isempty(struct queue );
   int isfull(struct queue );
   void display(struct queue);
   struct queue q;
   int a;
   q.front=-1; q.rear=-1;
   do
   {
         printf("\n\n 1 : Insert \n\n");
         printf("\n\n 2 : Remove \n\n");
         printf("\n\n 3 : Display \n\n");
         printf("\n\nEnter your choice");
         scanf("%d",&a);
         switch(a)
         {
            case 1 : if(isfull(q))
                        printf("\n\nQueue is Full, cannot insert\n\n");
                     else
                     {
                        printf("Enter a character to insert");
```

```
                    insert(&q,getche());
                }
                break;
        case 2 : if(isempty(q))
                    printf("\n\nQueue is empty, cannot remove data\n\n");
                else
                    printf("\nRemoved data : %c", del(&q));
                break;
        case 3 : if(!isempty(q))
                {  printf("\nThe items in the Queue are \n");
                   display(q);
                }
                else printf("\n\nQueue is empty, cannot display\n\n");
                break;
      }
   }while(a<4);
}
int isempty(struct queue q)
{
   return(((q.rear<0)||(q.front>q.rear))?1:0);
}
int isfull(struct queue q)
{
  return((q.rear==max)?1:0);
}
void insert(struct queue *q, char x)
{
  if(q->front==-1) q->front=0;
  q->data[++(q->rear)]=x;
}
char del(struct queue *q)
{
  return(q->data[(q->front)++]);
}
void display(struct queue q)
{
  int i=q.front;
  while(i<= q.rear)
     printf(" %c ", q.data[i++]);
}
```

Queue using link list

```
#include<stdio.h>
#include<conio.h>
struct queue
{
```

```
    char data;
    struct queue *next;
};
main()
{
    struct queue *insert(struct queue *, char);
    char del (struct queue *);
    void display(struct queue *);
    int a; char c;
    struct queue *front=NULL,*rear=NULL;
    clrscr();
    do
    {
            printf("\n\n1 : Insert Item\n\n");
            printf("2 : Remove Item\n\n");
            printf("3 : Display Item\n\n");
            printf("Enter your choice : ");
            scanf("%d",&a);
            fflush(stdin);
            switch(a)
            {
              case 1 : printf("\n\nEnter a character : ");
                       rear=insert(rear,getche());
                       if(front==NULL)
                             front=rear;
                       break;
              case 2 : if(isempty(rear,front))
                             printf("\n\nQueue is empty\n\n");
                       else
                       {
                             printf("\n\nRemoved data = %c\n\n",del(front));
                             front=front->next;
                       }
                       break;
              case 3 : display(front);
            }
    }while(a<4);
}
void display(struct queue *f)
{
    while(f!=NULL)
    {
          printf(" %c", f->data);
          f=f->next;
    }
}
struct queue *insert(struct queue *r, char x)
{
   struct queue *s=malloc(sizeof(struct queue));
```

```
  s->data=x;
  if(r!=NULL)
    r->next=s;
  s->next=NULL;
  return(s);
}
char del(struct queue *t)
{
  return(t->data);
}
int isempty(struct queue *f, struct queue *r)
{
  return(((r==NULL)||(f==NULL))?1:0);
}
```

Priority Queue :- A priority queue consists of entries, each of which contains a key called the priority of the entry. There are two types of priority queues – ascending priority queue and descending priority queue.

An ascending priority queue is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed. A descending priority queue is similar but allows deletion of only the largest item.

A stack may be viewed as a descending priority queue whose elements are ordered by time of insertion. The element that was inserted last has the greatest insertion-time value and is the only item that can be retrieved. A queue may similarly be viewed as an ascending priority queue whose elements are ordered by time of insertion.

In a time-sharing computer system, a large number of tasks may be waiting for the CPU. Some of these tasks have higher priority than others. Hence the set of tasks waiting for the CPU forms a priority queue.

Circular Queue: - A queue can be considered as circular queue. That is we consider the first element of the queue as immediately its last element. This implies that even if the last element is occupied a new element can be inserted behind it in the first element of the queue if it is empty.

When front == rear, it means that queue is empty. If front = 0 and rear == maximum position than queue is full. In another way if rear+1 = front than also queue is full.

D-Queue: - A deque is an ordered set of items from which items may be deleted at either end and into which items may be inserted at either end.

Example : Reverse a string using Stack

```
#include<stdio.h>
#include<conio.h>
struct stack
{
   char data[20];
   int top;
};
main()
{
  struct stack *s;
  char *x; int i=0;
  void push(struct stack *, char *);
  void popup(struct stack *, char *);
  s->top=-1;
  clrscr();
  printf("Enter a string ");
  gets(x);
  push(s,x);
  popup(s,x);
  printf("The reverse string is : %s ",x);
}
void push(struct stack *s, char *x)
{
   int i=0;
   while(*(x+i)!='\0')
        s->data[++(s->top)]=*(x+i++);
}
void popup(struct stack *s, char *x)
{
   int i=0;
   while(s->top>=0)
        *(x+i++)=s->data[(s->top)--];
}
```

Polish Notation: - The method of writing all operators either before their operands, or after them, is called Polish notation, which is discovered by the Polish mathematician Jan Lukasiewicz. When the operators are written before their operands, it is called the Prefix Form. When the operators come after their operands, it is called the Postfix Form or Reverse Polish Form or Suffix Form. When the operators appear in between the operands it is called Infix Form.

The advantage of postfix and prefix is the without parentheses we can write expression.

Infix form: a+b

Prefix form: +ab

Postfix form: ab+

Application of stacks: -

Conversion from infix to postfix expression: -

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
struct stack
{
  char data[20];
  int top;
};
int isoperand(char);
int isoperator(char);
char push(struct stack *,char);
char popup(struct stack *);

main()
{
  void convert(char *, char *);
  char infix[20], postfix[20];
  clrscr();
  printf("Enter the infix expression \n");
  gets(infix);
  convert(infix,postfix);
  printf("\n\nThe postfix expression is ");
  puts(postfix);
}
void convert(char *inf, char *pfx)
{
  struct stack stk;
  char ope;
  int i,j;
  stk.top=-1;
  for(i=0,j=0;*(inf+i)!='\0';i++)
  {
    if(isoperand(*(inf+i)))
    {
        *(pfx+j)=*(inf+i); j++;
    }
    else
    {
        if(isoperator(*(inf+i)))
         ope=push(&stk, *(inf+i));
         if(ope!='0')
         {
           *(pfx+j)=ope; j++;
```

```
            }
        }
    }
    while(stk.top>=0)
      *(pfx+j++)=stk.data[stk.top--];
    *(pfx+j)='\0';
}

int isoperand(char p)
{   int x=toascii(p);
   return(((x>=49)&&(x<=57)) || ((x>=65)&&(x<=90)) || ((x>=97)&&(x<=122)));
}

int isoperator(char x)
{
   return((x=='/') || (x=='*') || (x=='+') || (x=='-'));
}

char push(struct stack *s, char x)
{  char p;
   if((s->top>=0) && ((x=='+'||x=='-')&&(s->data[s->top]=='*' || s->data[s->top]=='/')))
     {
       p=s->data[s->top--];
       s->data[++s->top]=x ;
       return(p);
     }
   else
   {
    s->data[++s->top]=x;
    return('0');
   }
}
```

Conversion from infix to prefix expression: -

Postfix evaluation programme: -

```
#include<stdio.h>
#include<conio.h>
struct stack
{
   int data[20];
   int top;
};
int isdigit(char);
void push(struct stack *, int);
int popup(struct stack *);
int operation(char, int, int);
```

```
main()
{
  char *p;  int op1, op2, op3, d,i;
  struct stack stk;
  stk.top=-1;
  clrscr();
  printf("Enter the postfix expression\n");
  gets(p);
  for(i=0;*(p+i)!='\0';i++)
  {
    if(isdigit(*(p+i)))
    {
        d=*(p+i)-'0';
        push(&stk,d);
    }
    else
    {
        op2=popup(&stk);
        op1=popup(&stk);
        op3=operation(*(p+i),op1,op2);
        push(&stk,op3);
    }
  }
  printf("Result = %d", popup(&stk));
}

void push(struct stack *s, int x)
{
  s->data[++s->top]=x;
}

int popup(struct stack *s)
{
  return(s->data[s->top--]);
}

int isdigit(char x)
{
  return(x>=48 && x<=89);
}

int operation(char x, int op1, int op2)
{
  switch(x)
  {
    case '+' : return(op1+op2); break;
    case '-' : return(op1-op2); break;
    case '*' : return(op1*op2); break;
    case '/' : return(op1/op2); break;
```

```
    default  : printf("invalid operator\n");return(0);break;
  }
}
```

Linked lists: -
Singly linked list: -

```
#include<iostream.h>
#include<conio.h>
struct list
{
  char data;
  struct list *next;
};
struct list * create(char);
struct list * insertbegin(list *,char);
void insertmiddle(list *, char);
void insertend(list *, char);
void display(list *);
struct list *delbegin(struct list *);
void delmiddle(list *, char);
void delend(struct list *);
int search(struct list *, char);
main()
{
   int ch; char x;
   struct list *first;
   clrscr();
   cout<<"Select the option"<<endl;
   do
   {
     cout<<endl<<"1 = Create"<<endl;
     cout<<"2 = Insert at beginning"<<endl;
     cout<<"3 = Insert at middle"<<endl;
     cout<<"4 = Insert at end"<<endl;
     cout<<"5 = Display"<<endl;
     cout<<"6 = Delete first node"<<endl;
     cout<<"7 = Delete middle node"<<endl;
     cout<<"8 = Delete last node"<<endl;
     cout<<"9 = Search an item"<<endl;
     cout<<"10 = Exit"<<endl<<endl;
     cout<<"Enter your choice"<<endl;
     cin>>ch;
     switch(ch)
     {
          case 1: cout<<endl<<"Enter a character"<<endl;
                  cin>>x;
```

```
                first=create(x); break;
        case 2: cout<<"Enter a character "<<endl;
                cin>>x;
                first=insertbegin(first,x); break;
        case 3: cout<<"Enter a character "<<endl;
                cin>>x;
                insertmiddle(first,x); break;
        case 4: cout<<"enter a character"<<endl;
                cin>>x;
                insertend(first,x); break;
        case 5: cout<<"The characters are : ";
                display(first); break;
        case 6: first=delbegin(first); break;
        case 7: cout<<"Which character to be deleted"<<endl;
                cin>>x;
                delmiddle(first,x); break;
        case 8: delend(first); break;
        case 9: cout<<"Enter character to search"<<endl;
                cin>>x;
                cout<<endl<<"Character appear "<<search(first,x)<<" time"<<endl;break;
     }

   } while(ch<10);
}

int search(struct list *p,char z)
{
  int i=0;
  while(p!=NULL)
  {
    if(p->data==z)
          i++;
    p=p->next;
  }
  return(i);
}
void delend(struct list *p)
{
  while(p->next->next!=NULL)
    p=p->next;
  p->next=NULL;
}
void delmiddle(struct list *p, char z)
{
  struct list *prev=p;
  while(p!=NULL)
  {
     if(p->data==z)
          prev->next=p->next;
```

```
      prev=p;
      p=p->next;
   }
}
struct list *delbegin(struct list *p)
{
  return(p->next);
}
void insertend(struct list *p, char z)
{
  struct list *temp;
  temp=new struct list();
  temp->data=z;
  temp->next=NULL;
  while(p->next!=NULL)
    p=p->next;
  p->next=temp;
}
void insertmiddle(struct list *p, char z)
{
  struct list *temp; int n,i=1;
  temp=new struct list();
  temp->data=z;
  cout<<"After how many nodes ";
  cin>>n;
  while(i<n)
  {
    p=p->next;
    i++;
  }
  temp->next=p->next;
  p->next=temp;
}
struct list *insertbegin(struct list *p, char z)
{
  struct list *temp;
  temp=new struct list();
  temp->data=z;
  temp->next=p;
  return(temp);
}
struct list *create(char z)
{
  struct list *temp;
  temp=new struct list();
  temp->next=NULL;
  temp->data=z;
  return(temp);
}
```

```
void display(struct list *p)
{
  while(p!=NULL)
  {
    cout<<p->data<<" ";
    p=p->next;
  }
  cout<<endl;
}
```

Tree

Definition: - A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the root of the tree. The other two subsets are themselves binary tree called the left and right subtrees of the original tree.

Types of binary tree: -

1. Strictly Binary Tree: - If every nonleaf node in a binary tree has nonempty left and right subtrees, the tree is termed as strictly binary tree. A strictly binary tree with n leaves always contains 2n-1 nodes.
2. Complete Binary Tree: - A complete binary tree of depth d is the strictly binary tree all of whose leaves are at level d.

Properties: -

1. If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l+1.
2. The total number of nodes in a complete binary tree of depth d, than totalnode equals the sum of the number of nodes at each level between 0 and d.

Binary Tree Representation: - A binary tree can be represented in the memory as followes

```
Struct node tree
{
        int data;
        struct node * left;
        struct node * right;
};
```

Binary tree traversal algorithm: - A binary tree can be traversed in three different ways.

Preorder (Left-Root-Right): -

1. Visit the root
2. Traverse the left subtree in preorder
3. Traverse the right subtree in preorder

In-order (Root-Left-Right): -

1. Traverse the left subtree in inorder
2. visit the root
3. traverse the right subtree in inorder

Post-order (Left-Right-Root):-

1. Traverse the left subtree in postorder
2. traverse the right subtree in postorder
3. visit the root

Recursive function for binary tree traversal: -

1. Preorder : -

```
Void pretrav (binarytree tree)
{
        if (tree != NULL)
        {
                printf("%d", tree->info);
                pretrav(tree->left);
                pretrav(tree->right);
        }
}
```

2. Inorder: -

```
Void pretrav (binarytree tree)
{
        if (tree != NULL)
        {
                pretrav(tree->left);
                printf("%d", tree->info);
                pretrav(tree->right);
        }
}
```

2. Postorder: -

```
Void pretrav (binarytree tree)
{
        if (tree != NULL)
        {
                pretrav(tree->left);
                pretrav(tree->right);
                printf("%d", tree->info);
        }
}
```

Non-recursive function for binary tree traversal: -

1. Inorder: - (page no-271 from Tanenbaum book)

Threaded binary tree :-
Binary search tree
Operations
AVL tree
Searching
Sequential search
Binary search

Sorting

Insertion sort
Selection sort
Bubble sort
Quick sort

```
#include<stdio.h>
#include<conio.h>
#define max 10
void qsort(int x[], int);
void partition(int x[],int,int,int*);
void push(struct stack*, struct btype *);
void popup(struct stack*,struct btype *);
int isempty(struct stack*);
struct btype
{
 int lb;
 int ub;
};
struct stack
{
  int top;
  struct btype bound[max];
};
main()
{
 int x[20],i,n;
 clrscr();
 printf("How many numbers");
 scanf("%d",&n);
 for(i=0;i<n;i++)
 {
   printf("Number = ");
   scanf("%d",&x[i]);
 }
 qsort(x,n);
 printf("Sorted numbers are \n");
 for(i=0;i<n;i++)
   printf("%d ",x[i]);
```

```
}

void qsort(int x[],int n)
{
 struct stack st;
 struct btype bnd;
 int j,i;
 st.top=-1;
 bnd.lb=0;
 bnd.ub=n-1;
 push(&st,&bnd);
 while(!isempty(&st))
 {
   popup(&st,&bnd);
   while(bnd.ub>bnd.lb)
   {
        partition(x,bnd.lb, bnd.ub, &j);
        if(j-bnd.lb > bnd.ub-j)
        {
          i=bnd.ub;
          bnd.ub=j-1;
          push(&st,&bnd);
          bnd.lb=j+1;
          bnd.ub=i;
        }
        else
        {
          i=bnd.lb;
          bnd.lb=j+1;
          push(&st,&bnd);
          bnd.lb=i;
          bnd.ub=j-1;
        }
   }
 }
}
void partition(int x[], int lb, int ub, int *j)
{
  int a, down, t, up;
  a=x[lb];
  up=ub;
  down=lb;
  while(down < up)
  {
    while(x[down]<=a && down<ub)
         down++;
    while(x[up] >a)
         up--;
    if (down < up)
```

```
    {
        t=x[down];
        x[down]=x[up];
        x[up]=t;
    }
  }
  x[lb]=x[up];
  x[up]=a;
  *j=up;
}
void push(struct stack *s, struct btype *b)
{
  s->bound[++s->top]=*b;
}
void popup(struct stack *s, struct btype *b)
{
  *b=s->bound[s->top--];
}
int isempty(struct stack *s)
{
  return(s->top<0);
}
```

Merge sort

```
#include<stdio.h>
#include<conio.h>
#define max 10
void msort(int x[], int);
main()
{
  int x[20],i,n;
  clrscr();
  printf("How many numbers");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
    printf("Number = ");
    scanf("%d",&x[i]);
  }
  msort(x,n);
  printf("Sorted elements are : \n");
  for(i=0;i<n;i++)
   printf("%d ",x[i]);
  getch();
}

void msort(int x[],int n)
{
```

```
  int y[20],i,j,k,l1,l2,size, u1,u2;
  size=1;
  while (size<n)
  {
    l1=0;
    k=0;
    while(l1+size < n)
    {
        l2=l1+size;
        u1=l2-1;
        u2=((l2+size-1) < n)?(l2+size-1) : n-1;
        for(i=l1,j=l2; i<=u1 && j<=u2 ; k++)
          if(x[i]<=x[j])
            y[k]=x[i++];
          else
            y[k]=x[j++];
        for(;i<u1;k++)
          y[k]=x[i++];
        for(;j<=u2;k++)
          y[k]=x[j++];
        l1=u2+1;
    }
    for(i=l1;k<n;i++)
        y[k++]=x[i];
    for(i=0;i<n;i++)
        x[i]=y[i];
    size*=2;
  }
}
```

Radix sort

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#define max 10
void rsort(int x[], int);
main()
{
  int x[20],i,n;
  clrscr();
  printf("How many numbers");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
    printf("Number = ");
    scanf("%d",&x[i]);
  }
```

```
  rsort(x,n);
  printf("Sorted elements are : \n");
  for(i=0;i<n;i++)
    printf("%d ",x[i]);
  getch();
}
void rsort(int x[], int n)
{
  int front[10], rear[10];
  struct
  {
    int info;
    int next;
  }  node[max];
  int exp, first,i,j,k,p,q,y;
  for(i=0;i<n-1;i++)
  {
    node[i].info=x[i];
    node[i].next=i+1;
  }
  node[n-1].info=x[n-1];
  node[n-1].next = -1;
  first=0;
  for(k=1;k<5;k++)
  {
    for(i=0;i<10;i++)
    {
        rear[i]=front[i]=-1;
    }
    while(first!=-1)
    {
        p=first;
        first=node[first].next;
        y=node[p].info;
        exp=pow(10,k-1);
        j=(y/exp)%10;
        q=rear[j];
        if(q==-1)
          front[j]=p;
        else
          node[q].next=p;
        rear[j]=p;
    }
    for(j=0;j<10 &&front[j]==-1;j++);
    first=front[j];
    while(j<=9)
    {
        for(i=j+1;i<10 && front[i]==-1;i++);
        if(i<=9)
```

```
        {
          p=i;
          node[rear[j]].next=front[i];
        }
        j=i;
    }
    node[rear[p]].next=-1;
  }
  for(i=0;i<n;i++)
  {
    x[i]=node[first].info;
    first=node[first].next;
  }
}
```

Analysis of Algorithm

Time and Space complexity of algorithms

The *complexity* of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process. Usually there are natural units for the domain and range of this function. There are two main complexity measures of the efficiency of an algorithm:

- Time complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm. "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take
- Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.

Worst Case Analysis (Usually Done)

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one

Average Case Analysis (Sometimes done)

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs.

Best Case Analysis (Bogus)

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location.

average case and worst case analysis,

asymptotic notation as a measure of algorithm complexity

O and notations

Analysis of sorting algorithms

Selection sort,

Bubble sort

Insertion sort

Heap sort

Quick sort

Analysis of searching algorithms –

linear search

binary search.

Heap sort: - (page no:362 from Tanenbaum book);

Hash Function: - A function that transforms a key into a table index is called a hash function. If H is a has function and K is a key, H(K) is called the has of key and is the index at which a record with the K key should be placed.

Hash collision: - Suppose that two keys K1 and K2 are such that H(K1) equals to H(K2). Then when a record with key K1 is entered into the table, it is inserted at position H(K1). But when K2 is hashed, because its hash key is the same as that of K1, an attempt may be made to insert the record into the same position where the record with key K1 is stored. Clearly, two records can not occupy the same position. Such a situation is called a hash collision.

Question Answer:

1. What is the difference between single and double link list. Write the data structure for these two types.

Ans:

i) In single link list we have to maintain address of one node, but in double link list addresses of two nodes have to maintain.

ii) Single link list can be traversed in one direction either right to left or left to right. But double link list can be traversed in both directions.

iii) Each node of a single link list occupies less memory space than that of double link list.

Data structure of single link list

```
Struct node
{
   Data type info
   Struct node *next;
}
```

data structure of double link list

```
struct node
{
    struct node *left;
    data type info;
    struct node *right;
}
```

2.What is the difference between BSF and binary tree?

Ans:

i) Binary Search Tree (BST) is a special type of binary tree.

ii) In BST, value of left child must be smaller than that of root and value of right child must be larger than that of root. But in binary tree it is not mandatory

3. Write a recursive algorithm to determine the number of nodes in a binary tree.

Ans:

1. let c=0;
2. count(struct node *head)
3. c++;
4. if (head->left != NULL) goto step 2 with parameter (head->left) otherwise goto step 6.
5. if (head->right != NULL) goto step 2 with parameter (head->right)otherwise goto step 6.
6. Exit.

4. What are the difference between binary search and linear search?

Ans:

i) Binary search take very less time to search an element from a set, but linear search take more time .

ii) In binary search we have to calculate the middle position to compare the search element, but in linear search comparison start from first element to the last.

iii) In binary search, after each comparison, number of elements to be compared become half of previous, but in linear search. But in linear search, number of elements to be compared become one less than the previous.

What are directed and undirected graphs?

Ans:

Directed Graph:- The graph where each edge has a direction is called directed graph.

Undirected Graph: In undirected graph there is no direction in edge.