DS SEM 2 (2024-2025)

PROGRAM 1: IMPLEMENTATION OF LINEAR SEARCH

AIM: Write a program to implement Linear search and to find a particular element.

THEORY:

- Linear Search is also known as 'Sequential Search'. The element to be searched (key) will be searched sequentially with all the elements of the array till the match is found.
- If there is no match key will be compared till the last element of the array.
- *Time Complexity*: -
  Best Case: O(1) → If the element is at the beginning.
  Worst Case: O(n) → If the element is at the end or not present.
  Average Case: O(n/2) ≈ O(n)

PROCEDURE:

```
#include<stdio.h>

int linearsearch(int arr[],int n,int x)

{for(int i=0;i<n;i++)

{if(arr[i]==x)

{return i;}}

return -1;}

int main()

{int arr[]={1,2,3,4,5,6,7,8,9};

int n=sizeof(arr)/sizeof(arr[0]);

int x=4;

int index=linearsearch(arr,n,x);

if(index==-1)

{printf("Not Found");}

else

{printf("Element found at Index:%d",index);}

return 0;}
```

OUTPUT:

```
Element found at Index:3
```

PROGRAM 2: IMPLEMENTATION OF BINARY SEARCH

AIM: Write a program to implement Binary search to search a particular element.

THEORY:

- It is a searching algorithm which works on sorted array.
- This technique compares the key with the middle element based on the comparison, the key is searched in the first half of the array or in the second half of the array.
- *Time Complexity*:-
  Best Case: O(1) → If the element is at the beginning.
  Worst Case: O(log n)
  Average Case: O (log n)

PROCEDURE:

```
#include<stdio.h>
int binarysearch(int arr[],int x,int low,int high)
{if(high>=low)
{int mid=low+(high-low)/2;
if(arr[mid]==x)
{return mid;}
if(arr[mid]>x)
{return binarysearch(arr,x,low,mid-1);}
else
{return binarysearch(arr,x,mid+1,high);}}
return 0;}
int main(void)
{int arr[]={3,4,5,6,7,8,9};
int n=sizeof(arr)/sizeof(arr[0]);
int x=4;
int result=binarysearch(arr,x,0,n-1);
if(result==0)
{printf("Not Found");}
else
{printf("Found at Index:%d",result);}
```

}//main

OUPUT:

Found at Index:1

PROGRAM 3: IMPLEMENTATION OF FIBONACCI SEARCH

AIM: Write a program to implement Fibonacci search to find a particular element.

THEORY:

- It is an efficient search algorithm used for finding an element in a sorted array.
- It's similar to binary search but uses Fibonacci numbers to divide the search.
- *Time Complexity*:-
  Best Case: O(1)
  Worst Case: O(log n)
  Average Case: O (log n)

PROCEDURE:

```
#include<stdio.h>

int min(int x,int y)

{return (x<y)?x:y;}

int fibnsearch(int arr[],int n,int x)

{int fib2=0;

int fib1=1;

int fib = fib1+fib2;

while(fib<n)

{fib2=fib1;

fib1=fib;

fib=fib1+fib2;}

int offset=-1;

while(fib>1)

{int i=min(offset+fib2,n-1);

if(arr[i]<x)

{fib=fib1;

fib1=fib2;

fib2=fib-fib1;

offset=i;}

else if (arr[i]>x)

{fib=fib2;

fib1=fib1-fib2;
```

```
fib2=fib-fib1;}
else
{return i;}}
if(fib1&&arr[offset+1]==x)
{return (offset+1);}
return -1;}
int main()
{int arr[]={10,20,22,33,45,56,67,87,89,92,97};
int n=sizeof(arr)/sizeof(arr[0]);
int x=40;
int index=fibnsearch(arr,n,x);
if(index>=0)
{printf("Element found at index %d",index);}
else
{printf("Element not Found");}
return 0;}
```

OUTPUT:

Element not Found

PROGRAM 4: IMPLEMENTATION OF HEAP SORT

AIM: Write a program to implement Heap sort to arrange a list of integers either in ascending or descending order.

THEORY:

- A heap is a complete binary tree. Heapify is an important process in the heap sort, where it is the process of rearranging a binary tree into a heaped data structure.
- Heap sort is a sorting algorithm that eliminates the elements one by one from the heap sort of the list and then insert them into the sorted part of the list.

WORKING MODEL:

Heap sort uses a **binary heap** data structure (usually a **max heap**) to sort elements. It works in two phases:

1. **Build a Max Heap** from the input data.
2. **Repeatedly extract the maximum element** from the heap and place it at the end of the array.

Example Array-

[4, 10, 3, 5, 1]

Step 1: Build Max Heap

Convert the array into a max heap.

```
    10
   /  \
  5    3
 / \
4   1
```

Heapified Array: [10, 5, 3, 4, 1]

Step 2: Extract max (root = 10), place at end
[1, 5, 3, 4, **10**] → Max element fixed at end

Re-heapify remaining [1, 5, 3, 4]:

```
    5
   / \
  4   3
 /
1
```

Array: [5, 4, 3, 1, 10]

Repeat...

Final Sorted:

[1, 3, 4, 5, 10]

PROCEDURE:

```
#include<stdio.h>
void heapify(int arr[],int n,int i)
{int temp,maximum,left_index,right_index;
maximum=i;
right_index=2*i+2;
left_index=2*i+1;
if(left_index<n&&arr[left_index]>arr[maximum])
maximum=left_index;
if(right_index<n&&arr[right_index]>arr[maximum])
maximum=right_index;
if(maximum!=i)
{temp=arr[i];
arr[i]=arr[maximum];
arr[maximum]=temp;
heapify(arr,n,maximum);}}
void heapsort(int arr[],int n)
{int i,temp;
for(i=n/2-1;i>=0;i--)
{heapify(arr,n,i);}
for(i=n-1;i>0;i--)
{temp=arr[0];
arr[0]=arr[i];
arr[i]=temp;
heapify(arr,i,0);}}
```

```
int main()
{int arr[]={20,13,45,66,38,95};
int i,n=6;
printf("Original Array: ");
for(i=0;i<n;i++)
{printf("%d ",arr[i]);}
heapsort(arr,n);
printf("\nArray after performing heap sort: ");
for(i=0;i<n;i++)
{printf("%d ",arr[i]);}
return 0;}
```

OUTPUT:

```
Original Array: 20 13 45 66 38 95
Array after performing heap sort: 13 20 38 45 66 95
```

PROGRAM 5: IMPLEMENTATION OF QUICK SORT

AIM: Write a program to implement Quick sort to arrange a list of integers either in ascending or descending order.

THEORY:

- Quick sort follows the Divide & Conquer technique to arrange a given set of items in a systematic manner. We require the usage of a Pivot Element.
- Divide & Conquer technique breaks down the problem into sub-problems, down the algorithm, then solving the problems, combining back the result together to solve the original problem.
- *Time Complexity*:-
  Best Case: O(n log n)
  Worst Case: O(n^2)
  Average Case: O(n log n)

WORKING MODEL:

Quick sort uses a **divide and conquer** strategy:

1. Pick a **pivot**.
2. Partition the array: Elements **<** **pivot** on left, **>** **pivot** on right.
3. Recursively sort the subarrays.

Example Array-

[7, 2, 1, 6, 8, 5, 3, 4]

Step 1: Choose pivot (e.g., 4)

Partition around pivot 4:

Left side < 4: [2, 1, 3]
Right side > 4: [7, 6, 8, 5]

Array becomes: [2, 1, 3, 4, 7, 6, 8, 5]

Step 2: Recursively apply on [2, 1, 3] and [7, 6, 8, 5]

- Sort [2, 1, 3] → [1, 2, 3]
- Sort [7, 6, 8, 5] → [5, 6, 7, 8]

Final Sorted Array:

[1, 2, 3, 4, 5, 6, 7, 8]

PROCEDURE:

```
#include<stdio.h>
int partition(int a[],int start,int end)
{int pivot=a[end];
int i=(start-1);
for(int j=start;j<=end-1;j++)
{if(a[j]<pivot)
{i++;
int t=a[i];
a[i]=a[j];
a[j]=t;}}
int t=a[i+1];
a[i+1]=a[end];
a[end]=t;
return(i+1);}
void quick(int a[],int start,int end)
{if(start<end)
{int p=partition(a,start,end);
quick(a,start,p-1);
quick(a,p+1,end);}}
void printarr(int a[],int n)
{int i;
for(i=0;i<n;i++)
{printf("%d ",a[i]);}}
int main()
{int a[]={25,34,6,8,9,22,32};
int n=sizeof(a)/sizeof(a[0]);
printf("Before Sorting: ");
printarr(a,n);
quick(a,0,n-1);
```

```
printf("\nAfter Sorting: ");
printarr(a,n);
return 0;}
```

OUTPUT:

Before Sorting: 25 34 6 8 9 22 32

After Sorting: 6 8 9 22 25 32 34

PROGRAM 6: IMPLEMENTATION OF MERGE SORT

AIM: Write a program to implement Merge sort to arrange a list of integers either in ascending or descending order.

THEORY:

- Merge Sort is similar to quick sort algorithm, as it uses Divide & Conquer rule to sort out elements. It is one of the most popular and most efficient sorting algorithms.
- It divides the given list into 2 equal halves and then the sub-lists are divided again & again into 2 equal halves until the list cannot be further divided.
- The element pairs are sorted and then are combined. Thus, the merged element list is formed.

WORKING RULE:

Merge sort also uses a **divide and conquer** strategy:

1. **Divide** the array into two halves.
2. **Recursively sort** each half.
3. **Merge** the two sorted halves.

Example Array-

[38, 27, 43, 3, 9, 82, 10]

Step 1: Divide
Left: [38, 27, 43]
Right: [3, 9, 82, 10]

Keep dividing:

- Left → [38] [27] [43]
- Right → [3] [9] [82] [10]

Step 2: Merge sorted pairs

- [38] + [27] → [27, 38]
- [27, 38] + [43] → [27, 38, 43]
- [3] + [9] → [3, 9]
- [82] + [10] → [10, 82]
- [3, 9] + [10, 82] → [3, 9, 10, 82]

Step 3: Merge final halves

[27, 38, 43] + [3, 9, 10, 82] → [3, 9, 10, 27, 38, 43, 82]

PROCEDURE:

```
#include <stdio.h>
void merge(int a[], int beg, int mid, int end)
{int i, j, k;
int n1=mid-beg+1;
int n2=end-mid;
int leftarray[n1],rightarray[n2];
for(i = 0; i < n1; i++)
leftarray[i] = a[beg + i];
for(j = 0; j < n2; j++)
rightarray[j] = a[mid + 1 + j];
i=0;j=0;k=beg;
while (i < n1 && j < n2)
{if (leftarray[i] <= rightarray[j])
{a[k] = leftarray[i];
i++;}
else
{a[k] = rightarray[j];
j++;}
k++;}
while (i < n1)
{a[k]=leftarray[i];
i++;k++;}
while (j < n2)
{a[k]=rightarray[j];
j++;k++;}}
void mergesort(int a[], int beg, int end)
{if (beg < end)
{int mid=(beg + end) / 2;
mergesort(a, beg, mid);
```

```
mergesort(a, mid + 1, end);
merge(a, beg, mid, end);}}
void printarr(int a[], int n)
{for (int i = 0; i < n; i++)
{printf("%d ", a[i]);}
printf("\n");}
int main()
{int a[] = {2, 4, 5, 3, 88, 7, 65, 36, 70, 76};
int n=sizeof(a) / sizeof(a[0]);
printf("Before Sorting: ");
printarr(a,n);
mergesort(a, 0, n - 1);
printf("After Sorting: ");
printarr(a,n);
return 0;}
```

OUTPUT:

Before Sorting: 2 4 5 3 88 7 65 36 70 76

After Sorting: 2 3 4 5 7 36 65 70 76 88

PROGRAM 7: IMPLEMENTATION OF STACK USING ARRAY

AIM: Write a program to implement Stack using Arrays in C Language.

THEORY:

- Stack is a linear data structure that follows LIFO (Last-In-First-Out) Principle, the last element to be inserted will the first element to popped out.
- Insertion & Deletion both occur at the same end only.
- Array: It is a collection of variables of the same datatype with contiguous memory allocation, where all variables are stored one after the other and are called with a common name but are differentiated by their subscript position.

PROCEDURE:

```
#include<stdio.h>
#include<stdlib.h>
void push(int x);int pop();int isfull();int isempty();void display();
int gettop();
int stack[100],maxsize,ch,top=-1,x,i;
int isfull()
{if(top==maxsize-1)
return 1;
else
return 0;}
int isempty()
{if(top==-1)
return 1;
else
return 0;}
void push(int x)
{if(isfull())
{printf("\nSTACK OVERFLOW");}
else
{top++;
stack[top]=x;}}
int pop()
```

```
{if(isempty())
{printf("\nSTACK UNDERFLOW");}
else
{int temp=stack[top];
top--;
return(temp);}}
void display()
{if(isempty())
{printf("\nSTACK IS EMPTY");}
else
{printf("\nELEMENTS IN THE STACK ARE:");printf("\n");
for(i=top;i>=0;i--)
{printf("%d ",stack[i]);}}}
int gettop()
{if(isempty())
{printf("\nSTACK IS EMPTY");}
else
{return stack[top];}}
void main()
{printf("Enter Size of Stack: ");
scanf("%d",&maxsize);
printf("\n\tMENU - STACK OPERATIONS USING ARRAYS");
printf("\n\t1.PUSH\n\t2.POP\n\t3.DISPLAY\n\t4.GET TOP\n\t5.EXIT\n");
do
{printf("\nEnter Choice: ");
scanf("%d",&ch);
switch(ch)
{case 1:{printf("\nENTER VALUE TO BE PUSHED: ");
      scanf("%d",&x);push(x);break;}
 case 2:{printf("\nTHE POPPED ELEMENT IS - %d",pop());break;}
```

```
 case 3:{display();break;}
 case 4:{printf("\nTOP ELEMENT OF THE STACK IS - %d",gettop());
       break;}
 case 5:{printf("\nEXITTING THE PROGRAM");exit(0);}
 default:
 {printf("\nINVALID CHOICE ENTERED");}}}
 while(ch!=5);}
```

OUTPUT:

```
Enter Size of Stack: 5
        MENU - STACK OPERATIONS USING ARRAYS
        1.PUSH
        2.POP
        3.DISPLAY
        4.GET TOP
        5.EXIT
Enter Choice: 1
ENTER VALUE TO BE PUSHED: 22
Enter Choice: 1
ENTER VALUE TO BE PUSHED: 22
Enter Choice: 2
THE POPPED ELEMENT IS - 22
Enter Choice: 1
ENTER VALUE TO BE PUSHED: 77
Enter Choice: 3
ELEMENTS IN THE STACK ARE:
77 22
Enter Choice: 4
TOP ELEMENT OF THE STACK IS - 77
Enter Choice: 6
INVALID CHOICE ENTERED
```

Enter Choice: 5

EXITTING THE PROGRAM

PROGRAM 8: IMPLEMENTATION OF STACK USING LINKED LIST

AIM: Write a program to implement Stack using Linked Lists in C Language.

THEORY:

- Stack implementation using linked list, the nodes are maintained in non-contiguous memory. Each node contains a pointer to the immediate next in line node in the stack (Linked list allocate memory dynamically).
- The Pros of Stack implementation using Linked List –
  Dynamic Data Structure: - Linked list is a dynamic data structure, so it can grow & shrink at runtime by allocating and deallocating memory.
  Insertion & Deletion: - Unlike in an array, we don't have to shift elements after insertion and deletion of stuff. They're relatively easier by updating the address present in the next pointer of a node.

PROCEDURE:

```
#include <stdio.h>
#include <stdlib.h>
struct node
{int data;struct node *next;};
struct node *top = NULL;
void push(int value)
{struct node *newnode = (struct node *)malloc(sizeof(struct node));
if (newnode == NULL)
{printf("\nMEMORY ALLOCATION FAILED");
return;}
newnode->data = value;
newnode->next = top;
top = newnode;
printf("\nNode is Inserted");}
int pop()
{if (top == NULL)
{printf("\nSTACK UNDERFLOW");
return -1;}
else
{struct node *temp = top;
```

```
int temp_data = top->data;
top = top->next;
free(temp);
return temp_data;}}
void display()
{if (top == NULL)
{printf("\nSTACK IS EMPTY");}
else
{printf("\nSTACK IS: ");
struct node *temp = top;
while (temp != NULL)
{printf("%d -> ", temp->data);
temp = temp->next;}printf("NULL\n");}}
int main()
{int ch, value;
printf("\n\tMENU - IMPLEMENTATION OF STACK USING LINKED
LIST");printf("\n\t1.PUSH\n\t2.POP\n\t3.DISPLAY\n\t4.EXIT");
while (1)
{printf("\nEnter Choice: ");
scanf("%d", &ch);
switch (ch)
{case 1:printf("ENTER VALUE TO BE INSERTED: ");
       scanf("%d", &value);push(value);
       break;
case 2:value = pop();
      if (value != -1)
      printf("\nTHE POPPED ELEMENT IS - %d", value);
      break;
case 3:display();break;
case 4:printf("\nEXITING THE PROGRAM\n");exit(0);
default:printf("\nINVALID CHOICE ENTERED");}}
```

return 0;}

OUTPUT:

```
        MENU - IMPLEMENTATION OF STACK USING LINKED LIST
        1.PUSH
        2.POP
        3.DISPLAY
        4.EXIT
Enter Choice: 1
ENTER VALUE TO BE INSERTED: 22
Node is Inserted
Enter Choice: 1
ENTER VALUE TO BE INSERTED: 33
Node is Inserted
Enter Choice: 1
ENTER VALUE TO BE INSERTED: 55
Node is Inserted
Enter Choice: 2
THE POPPED ELEMENT IS - 55
Enter Choice: 3
STACK IS: 33 -> 22 -> NULL
Enter Choice: 5
INVALID CHOICE ENTERED
Enter Choice: 4
EXITING THE PROGRAM
```

PROGRAM 9: IMPLEMENTATION OF QUEUES USING ARRAYS

AIM: Write a program to implement Queues using Array in C Language.

THEORY:

- Queues is a linear data structure that follows the F-I-F-O (First-In-First-Out) Principle.
- The first element to be stored in the queue will be the first element to be removed as well.
- 2 pointers are provided to enqueue or dequeue a queue; 'Front & Rear'
  Front – Tracks the first element of the queue
  Rear – Tracks the last element of the queue
  Initially, the values of Front & Rear are set to -1.
- Arrays are fundamental data structures in computer science, providing a way to store and organize multiple elements of the same data type under a single variable name.

PROCEDURE:

```
#include<stdio.h>
#include<stdlib.h>
int a[100],choice,maxsize=5,f=-1,r=-1,x,i;
int isfull()
{if(r==maxsize-1)
return 1;
else
return 0;}
int isempty()
{if(f==-1)
return 1;
else
return 0;}
void enqueue(int x)
{if(isfull())
printf("\nQUEUE OVERFLOW");
else
{r++;a[r]=x;
```

```
if(f==-1)
f++;}}
int dequeue()
{if(isempty())
return -1;
else
{int temp;temp=a[f];
if(f==r)
f=r=-1;
else
{f++;return temp;}}}
void display()
{if(isempty())
{printf("\nQUEUE IS EMPTY");}
else
for(i=f;i<=r;i++)
{printf("%d\t",a[i]);}}
int size()
{if(isempty())
return -1;
else
return r-f+1;}
int main()
{int maxsize=5;
printf("MENU - QUEUE OPERATIONS");
printf("\n1.ENQUEUE\n2.DEQUEUE\n3.DISPLAY\n4.SIZE\n5.EXIT\n");
while(1)
{printf("\nEnter Choice:");
scanf("%d",&choice);
switch(choice)
```

```
{case 1:{printf("\nEnter Element to be Inserted:");
        scanf("%d",&x);enqueue(x);break;}
 case 2:{printf("\nDeleted Element is %d",dequeue());break;}
 case 3:{display();break;}
 case 4:{printf("\nSize of Queue is %d",size());break;}
 case 5:{ exit(0);}
default:{printf("\nINVALID CHOICE INSERTED");break;}}}
return 0;}
```

OUTPUT:

```
MENU - QUEUE OPERATIONS
1.ENQUEUE
2.DEQUEUE
3.DISPLAY
4.SIZE
5.EXIT
Enter Choice:1
Enter Element to be Inserted:21
Enter Choice:1
Enter Element to be Inserted:34
Enter Choice:1
Enter Element to be Inserted:67
Enter Choice:2
Deleted Element is 21
Enter Choice:3
34        67
Enter Choice:6
INVALID CHOICE INSERTED
Enter Choice:4
Size of Queue is 2
Enter Choice:5
```

PROGRAM 10: IMPLEMENTATION OF QUEUE USING LINKED LISTS

AIM: Write a program to implement Queue using Linked list in C Language.

THEORY:

- In queues, front pointer points to the first element & the rear pointer points to the last element of the queue.
- The problem arising with the linear queues that if some empty cells occur at the beginning of the queue, we cannot insert new elements int the empty spaces as the rear cannot be further implemented. The solution is 'Circular Queue'.
- A Circular Queue is like a normal queue that follows the FIFO principle but does not end the queue, instead connects the last position of the queue to the first and thus, we can insert elements in the empty spaces of the cells.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{ int data;
struct node *link;};
struct node *front=NULL;
struct node *rear=NULL;
void enqueue(int value)
{struct node *newnode=malloc(sizeof(struct node));
newnode->data=value;
newnode->link=NULL;
if(rear==NULL)
{front=rear=newnode;}
else
{rear->link=newnode;
rear=newnode;}}
void dequeue()
{if(front==NULL)
return ;
struct node *temp=front;
```

```
front=front->link;
if(front!=NULL)
{rear=NULL;}
free(temp);}
void display()
{struct node *temp=front;
while(temp!=NULL)
{printf("%d",temp->data);
temp=temp->link;
if(temp!=NULL)
printf("");}printf("\n");}
int main()
{int n,value;
printf("Enter the Number of Elements:");
scanf("%d",&n);
for(int i=0;i<n;i++)
{printf("\nEnter Value: ");
scanf("%d",&value);
enqueue(value);}
dequeue();
display();
return 0;}
```

OUTPUT:

```
Enter the Number of Elements:5
Enter Value: 1
Enter Value: 2
Enter Value: 3
Enter Value: 4
Enter Value: 7
2347
```

PROGRAM 11: IMPLEMENTATION OF INFIX TO POSTFIX EVALUATION

AIM: Write a program to implement Infix to Postfix Evaluation in C Language.

THEORY:

- Infix to Postfix Evaluation in C Language involves the usage of the 'Stack' data structure to transform expressions
- The conversion process prioritizes operands, pushes operators into the stack based on the precedence of the operators and then finally evaluates the resulting postfix expression.

WORKING MODEL:

Infix: (A + B) * (C - D)

Steps**:**

1. Read (→ push
2. Read A → output
3. Read + → push
4. Read B → output
5. Read) → pop + → output
6. Read * → push
7. Read (→ push
8. Read C → output
9. Read - → push
10. Read D → output
11. Read) → pop - → output

→ Final Postfix: A B + C D - *

PROCEDURE:

```
#include<stdio.h>

#include<ctype.h>

char stack[100];

int top=-1,maxsize=30;

void push(char x);char pop();int priority(char x);

void push(char x)

{if(top!=maxsize)

stack[++top]=x;}

char pop()
```

```
{if(top==-1)
{return -1;}
else
{return stack[top--];}}
int priority(char x)
{switch(x)
{case '+':
 case '-':{return 1;break;}
 case '*':
 case '/':{return 2;break;}}
return 0;}
int main()
{char str[100];
char *c,x;
printf("Enter Expression:");
scanf("%s",str);printf("\n");
c=str;
while(*c!='\0')
{if(isalpha(*c)||isdigit(*c))
{printf("%c",*c);}
else if(*c==')')
{while((x=pop())!='('
printf("%c",x);}
else
{while(priority(stack[top])>=priority(*c))
printf("%c",pop());
push(*c);}
c++;}
while(top!=-1)
{printf("%c",pop());}
```

return 0;}

OUTPUT:

Enter Expression:24+44-33/77*659

2444+3377/659*-

PROGRAM 12: IMPLEMENTATION OF POSTFIX EVALUATION

AIM: Write a program to implement Postfix Evaluation using C Language.

THEORY:

- Postfix Evaluation is also called as 'Reverse Polish Notation'.
- Evaluating postfix expressions is a fundamental concept in computer programming, which involves a specific type of notation known as 'Reverse Polish Notation (RPN)'.
- Unlike traditional infix notation where operators are placed in between the operands, postfix notation places operators after their operands.

WORKING MODEL:

Expression: 5 6 2 + *
Meaning: 5 * (6 + 2)

Step	Stack	Action
5	[5]	Push 5
6	[5, 6]	Push 6
2	[5, 6, 2]	Push 2
+	[5, 8]	6 + 2 → push 8
*	[40]	5 * 8 → push 40

Final Result: 40

PROCEDURE:

```
#include<stdio.h>
#include<ctype.h>
void push(int x);int pop();
int stack[20],top=-1,maxsize=25;
void push(int x)
{if(top!=maxsize)
stack[++top]=x;}
int pop()
{if(top!=-1)
return stack[top--];}
int main()
{char str[100],*c;
```

```
int n1,n2,n3,num;
printf("Enter Expression:");
scanf("%s",str);
c=str;
while(*c!='\0')
{if(isdigit(*c))
{num=*c-48;push(num);}
else
{n1=pop();n2=pop();
switch(*c)
{case '+':{n3=n2+n1;break;}
 case '-':{n3=n2-n1;break;}
 case '*':{n3=n2*n1;break;}
 case '/':{n3=n2/n1;break;}}
 push(n3);}c++;}
 printf("\nResult of Expression %s=%d",str,pop());
 return 0;}
```

OUTPUT:

```
Enter Expression:32-19+*
Result of Expression 32-19+*=10
```

PROGRAM 13: IMPLEMENTATION OF SINGLE LINKED LIST

AIM: Write a program to implement Single Linked List in C Language.

THEORY:

- A linked list is a collection of nodes in which one node is linked with another but they are not stored in a sequential manner.
- It consists of a (i) DATA FIELD & (ii) LINK FIELD.
- A linked list data structure uses a data field to store information or the actual data and the link field (reference field) to point to the next node in the list.

PROCEDURE:

```
#include <stdio.h>
#include <stdlib.h>
struct Node
{int data;
struct Node* next;};
void insert(struct Node** head, int value)
{struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = value;
newNode->next = NULL;
if (*head == NULL)
{*head = newNode;}
else
{struct Node* temp = *head;
while (temp->next != NULL)
temp = temp->next;
temp->next = newNode;}}
void display(struct Node* head)
{struct Node* temp = head;
while (temp != NULL)
{printf("%d -> ", temp->data);
temp = temp->next;}
printf("NULL\n");}
```

```
int main()
{struct Node* head = NULL;
int choice, value;
printf("\n1. Insert\n2. Display\n3. Exit");
while (1)
{printf("\nEnter Choice: ");
scanf("%d", &choice);
switch (choice)
{case 1:printf("Enter value to Insert: ");
        scanf("%d", &value);
        insert(&head, value);break;
 case 2:display(head);break;
 case 3:exit(0);
 default:printf("Invalid choice!\n");}}
return 0;}
```

OUTPUT:

```
1. Insert
2. Display
3. Exit
Enter Choice: 1
Enter value to Insert: 22
Enter Choice: 1
Enter value to Insert: 55
Enter Choice: 2
22 -> 55 -> NULL
Enter Choice: 4
Invalid choice!
Enter Choice: 3
```

PROGRAM 14: IMPLEMENTATION OF DOUBLE LINKED LIST

AIM: Write a program to implement Double Linked List in C Language.

THEORY:

- It is a collection of nodes where one node records the link of its previous node and the next node as well.
- This structure allows for traversal in both forward & backward directions, unlike a singly linked list which allows only forward traversal.
- Hence, it solves the issue of the incapability of reverse traversal in a single linked list.

PROCEDURE:

```
#include <stdio.h>
#include <stdlib.h>
struct Node
{int rollno;
struct Node* prev;
struct Node* next;};
struct Node* createNode(int rollno)
{struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->rollno = rollno;
newNode->prev = NULL;
newNode->next = NULL;
return newNode;}
void insertEnd(struct Node** head, int rollno)
{struct Node* newNode = createNode(rollno);
if (*head == NULL)
{*head = newNode;
return;}
struct Node* temp = *head;
while (temp->next != NULL)
temp = temp->next;
temp->next = newNode;
newNode->prev = temp;}
```

```
void displayForward(struct Node* head)
{struct Node* temp = head;
printf("Roll numbers (Forward): ");
while (temp != NULL)
{printf("%d ", temp->rollno);
temp = temp->next;}
printf("\n");}
void displayBackward(struct Node* head)
{if (head == NULL)
return;
struct Node* temp = head;
while (temp->next != NULL)
temp = temp->next;
printf("Roll numbers (Backward): ");
while (temp != NULL)
{printf("%d ", temp->rollno);
temp = temp->prev;}
printf("\n");}
int main()
{struct Node* head = NULL;
int choice, rollno;
 printf("\n1. Insert Roll No\n2. Display Forward\n3. Display Backward\n4. Exit\n");
while (1)
{printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice)
{case 1:printf("Enter roll number to insert: ");
        scanf("%d", &rollno);
        insertEnd(&head, rollno);break;
case 2:displayForward(head);break;
```

```
case 3:displayBackward(head);break;
case 4:printf("Exitting program.\n");exit(0);
default:printf("Invalid choice! Try again.\n");}}
return 0;}
```

OUTPUT:

```
1. Insert Roll No
2. Display Forward
3. Display Backward
4. Exit
Enter your choice: 1
Enter roll number to insert: 101
Enter your choice: 1
Enter roll number to insert: 102
Enter your choice: 1
Enter roll number to insert: 103
Enter your choice: 2
Roll numbers (Forward): 101 102 103
Enter your choice: 3
Roll numbers (Backward): 103 102 101
Enter your choice: 5
Invalid choice! Try again.
Enter your choice: 4
Exitting program.
```

PROGRAM 15: IMPLEMENTATION OF BFS TRAVERSAL

AIM: Write a program to implement BFS in C Language.

THEORY:

- Breadth – First Search (BFS) is a graph traversal algorithm that explores a graph or tree level by level using the 'Queue' data structure. It starts at a designated source node and visits all its immediate neighbors before moving on to their unvisited neighbors and so on.
- This systematic exploration ensures that all nodes at a given depth are visited before moving to the next depth level.

WORKING MODEL:

Example Graph-

Adjacency List-

0: 1, 2

1: 0, 3, 4

2: 0, 5

3: 1

4: 1

5: 2

Steps-

1. **Start at 0** → Enqueue 0 → Mark visited
2. **Dequeue 0** → Visit neighbours 1, 2 → Enqueue them
3. **Dequeue 1** → Visit neighbours 3, 4 → Enqueue them
4. **Dequeue 2** → Visit 5 → Enqueue
5. Continue until queue is empty

Traversal Order: 0 → 1 → 2 → 3 → 4 → 5

PROCEDURE:

```
#include <stdio.h>
#define max 10
int n,adj[max][max],visited[max];
void readmatrix()
{printf("Enter The Number of Vertices in the Graph:");
scanf("%d",&n);
printf("\nEnter The Adjacency Matrix:");
for(int i=1;i<=n;i++)
for(int j=1;j<=n;j++)
scanf("%d",&adj[i][j]);
for(int i=1;i<=n;i++)
visited[i]=0;}
void bfs(int source)
{int queue[max];
int i,front,root,rear;
front=rear=0;
visited[source]=1;
queue[rear++]=source;
printf("%d\t",source);
while(front!=rear)
{root=queue[front];
for(int i=1;i<=n;i++)
if(adj[root][i]&&!visited[i])
{visited[i]=1;queue[rear++]=1;
printf("%d\t",i);}
front++;}}
void main()
{int source;
readmatrix();
printf("\nEnter The Source:");
```

```
scanf("%d",&source);
printf("\nThe Nodes Visited in BFS Order: ");
bfs(source);}
```

OUTPUT:

```
Enter The Number of Vertices in the Graph:4
Enter The Adjacency Matrix:
0 1 1 0
1 0 1 1
1 1 0 1
0 1 1 0
Enter The Source:2
The Nodes Visited in BFS Order: 2  1  3  4
```

PROGRAM 16: IMPLEMENTATION OF DFS TRAVERSAL

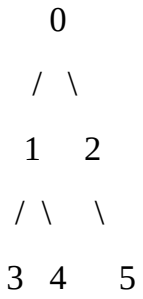
AIM: Write a program to implement DFS in C Language.

THEORY:

- Depth – First Search (DFS) is an algorithm used for traversing or searching tree or graph data structures. Its core principle is to explore as deeply as possible along each branch before backtracking.
- This 'depth-first' approach distinguishes it from BFS, which explores nodes level by level and it uses 'Stack' data structure.

WORKING MODEL:

Example Graph-



Adjacency List-

0 → 1, 2

1 → 3, 4

2 → 5

Steps-

1. Visit 0 → mark visited
2. Go to 1 → mark visited
3. Go to 3 → mark visited
4. Backtrack → go to 4 → mark visited
5. Backtrack to 0 → go to 2 → mark visited
6. Go to 5 → mark visited

Traversal Order- 0 → 1 → 3 → 4 → 2 → 5

PROCEDURE:

```
#include <stdio.h>

#define max 10

int n,adj[max][max],visited[max];

void readmatrix()
```

```
{printf("Enter The Number of Vertices in the Graph:");
scanf("%d",&n);
printf("\nEnter The Adjacency Matrix:");
for(int i=1;i<=n;i++)
for(int j=1;j<=n;j++)
scanf("%d",&adj[i][j]);
for(int i=1;i<=n;i++)
visited[i]=0;}
int dfs(int source)
{int i; visited[source]=1;
printf("%d\t",source);
for(int i=1;i<=n;i++)
if(adj[source][i]&&!visited[i])
dfs(i);}
void main()
{int source;
readmatrix();
printf("\nEnter The Source:");
scanf("%d",&source);
printf("\nThe Nodes Visited in DFS Order: ");
dfs(source);}
```

OUTPUT:

```
Enter The Number of Vertices in the Graph:4
Enter The Adjacency Matrix:
0 1 1 0
1 0 1 1
1 1 0 1
0 1 1 0
Enter The Source:1
The Nodes Visited in DFS Order: 1  2  3  4
```

PROGRAM 17: IMPLEMENTATION OF BST TRAVERSAL

AIM: Write a program to implement BST in C Language.

THEORY:

- BST – Binary Search Tree (Traversal)
- BST Traversal involves a systematic visitation to each node in a tree structure.
- The most common traversal methods are – Preorder, Postorder & Inorder.
- Each traversal type has a unique order for visiting the root, left sub tree and right sub tree, thus making them suitable for different operations

PROCEDURE:

```
#include <stdio.h>
#include <stdlib.h>
struct node
{int element;
struct node* left;
struct node* right;};
struct node* createNode(int val)
{struct node* Node = (struct node*)malloc(sizeof(struct node));
Node->element = val;
Node->left = NULL;
Node->right = NULL;
return (Node);}
void traversePreorder(struct node* root)
{if (root == NULL)
return;
printf(" %d ", root->element);
traversePreorder(root->left);
traversePreorder(root->right);}
void traverseInorder(struct node* root)
{if (root == NULL)
return;
traverseInorder(root->left);
```

```
printf(" %d ", root->element);
traverseInorder(root->right);}
void traversePostorder(struct node* root)
{if (root == NULL)
return;
traversePostorder(root->left);
traversePostorder(root->right);
printf(" %d ", root->element);}
int main()
{struct node* root = createNode(36);
root->left = createNode(26);
root->right = createNode(46);
root->left->left = createNode(21);
root->left->right = createNode(31);
root->left->left->left = createNode(11);
root->left->left->right = createNode(24);
root->right->left = createNode(41);
root->right->right = createNode(56);
root->right->right->left = createNode(51);
root->right->right->right = createNode(66);
printf("\n The Preorder traversal of given binary tree is -\n");
traversePreorder(root);
printf("\n The Inorder traversal of given binary tree is -\n");
traverseInorder(root);
printf("\n The Postorder traversal of given binary tree is -\n");
traversePostorder(root);
return 0;}
```

OUTPUT:

```
The Preorder traversal of given binary tree is -
 36  26  21  11  24  31  46  41  56  51  66
```

The Inorder traversal of given binary tree is -

11 21 24 26 31 36 41 46 51 56 66

The Postorder traversal of given binary tree is -

11 24 21 31 26 41 51 66 56 46 36

PROGRAM 18: IMPLEMENTATION OF HASHING

AIM: Write a program to implement Hashing in C Language.

THEORY:

- Hashing refers to the process of generating a fixed size output from an input of variable size using mathematical formulae known as the 'Hash Function'.
- This technique determines the index or location for the storage of an item in a data structure.
- Hash Function – It returns the index of an element in the array called as 'Hash Table'.
- Hash Table – It is a data structure that maps keys to the values using a hash function.

PROCEDURE:

```
#include <stdio.h>

#define SIZE 13

int ht[SIZE];

void init()

{for (int i = 0; i < SIZE; i++)

ht[i] = -1;}

int hashfun(int x)

{return x % SIZE;}

int collres(int pos)

{int d = pos;

while (ht[d] != -1)

{d++;

if (d >= SIZE)

d = 0;}

return d;}

void insert(int x)

{int pos = hashfun(x);

if (ht[pos] == -1)

ht[pos] = x;

else
```

```
{pos = collres(pos);
ht[pos] = x;}}
void display()
{printf("Hashing TABLE\n\n");
for (int i = 0; i < SIZE; i++)
printf("%d\t%d\n", i, ht[i]);
printf("--------------\n");}
int main() {
int keys[12] = {10, 100, 32, 45, 58, 126, 3, 29, 200, 400, 0, 21};
init();
printf("KEYS: ");
for (int i = 0; i < 12; i++)
printf("%d   ", keys[i]);
printf("\nLOCN: ");
for (int i = 0; i < 12; i++)
printf("%d   ", hashfun(keys[i]));
printf("\n--------------\n");
for (int i = 0; i < 12; i++)
insert(keys[i]);
display();
return 0;}
```

OUTPUT:

```
KEYS: 10   100   32   45   58   126   3   29   200   400   0   21
LOCN: 10   9   6   6   6   9   3   3   5   10   0   8
--------------
Hashing TABLE

0         0
1         21
2         -1
```

3	3
4	29
5	200
6	32
7	45
8	58
9	100
10	10
11	126
12	400

ANNEX

VIVA QUESTIONS & ANSWERS

1. What is a data structure?
   A data structure is a way to store and organize data so that it can be used efficiently.
2. What are the types of data structures?

- Linear: Array, Stack, Queue, Linked List
- Non-linear: Trees, Graphs
- Hash-based: Hash tables

3. What is the difference between array and linked list?

- Array has fixed size, linked list is dynamic
- Array has random access, linked list has sequential access

4. What is searching?
   Searching is the process of finding a specific element in a data structure.
5. Name two types of searching algorithms.

- Linear Search
- Binary Search

6. What is linear search?
   It checks each element one by one until the target is found or the end is reached.
7. What is binary search?
   It searches in a sorted array by repeatedly dividing the search interval in half.
8. Time complexity of binary search?
   O(log n)
9. What is sorting?
   Sorting arranges data in a particular order (ascending/descending).
10. Name some sorting algorithms.
    Bubble sort, Insertion sort, Selection sort, Merge sort, Quick sort

11. What is a linear data structure?
    A structure where elements are arranged sequentially (like array, stack, queue).
12. What is a stack?
    A linear data structure that follows LIFO (Last In First Out) principle.
13. What is a queue?
    A linear structure that follows FIFO (First In First Out) principle.
14. What is the difference between stack and queue?

- Stack: insert/remove from top
- Queue: insert at rear, remove from front

15. What operations are performed on stack?
    Push, Pop, Peek
16. What is a circular queue?
    A queue where the last position is connected to the first to make a circle.
17. What is dequeue?
    A double-ended queue where insertion/deletion is allowed at both ends.
18. Applications of stack?
    Expression evaluation, recursion, undo mechanism
19. Applications of queue?
    CPU scheduling, print spooling, call center queues
20. Difference between static and dynamic data structures?

- Static: size fixed at compile time (arrays)
- Dynamic: size can change at runtime (linked list)

21. What is a linked list?
    A linear data structure where each element points to the next.
22. Types of linked lists?

(i) Singly Linked List
(ii) Doubly Linked List
(iii) Circular Linked List

23. Difference between singly and doubly linked list?

(i) Singly: one pointer (next)
(ii) Doubly: two pointers (next and prev)

24. How do you insert a node at the beginning of a linked list?
    Create a node, point its next to head, update head to new node.
25. How to delete a node from the end of singly linked list?
    Traverse to second last node, set its next to NULL, free last node.
26. Advantages of linked list over array?
    Dynamic size, efficient insertion/deletion
27. Disadvantages of linked list?
    No random access, more memory (extra pointers)
28. What is a circular linked list?
    A list where the last node points to the first node.
29. Applications of linked list?
    Polynomial manipulation, dynamic memory allocation, adjacency lists
30. How to find the middle element in a linked list?
    Use slow and fast pointer approach.

31. What is a non-linear data structure?
    A structure where elements are not in a sequence (e.g., trees, graphs).
32. What is a tree?
    A hierarchical data structure with a root and child nodes.

33. What is a binary tree?
    A tree where each node has at most 2 children.
34. What is a binary search tree (BST)?
    A binary tree where left < root < right for all nodes.
35. What is traversal in trees?
    Visiting all nodes in a specific order: inorder, preorder, postorder
36. What is inorder traversal?
    Left → Root → Right
37. What is preorder traversal?
    Root → Left → Right
38. What is postorder traversal?
    Left → Right → Root
39. What is a complete binary tree?
    A binary tree where all levels are full except possibly the last.
40. What is the height of a tree?
    Number of edges on the longest path from root to a leaf.

41. What is hashing?
    Hashing is a technique to convert a key into an index of an array using a hash function.
42. What is a hash function?
    A function that maps keys to indices in a hash table.
43. What is collision in hashing?
    When two keys map to the same index.
44. How to resolve collisions?

(i) Chaining
(ii) Open Addressing (Linear probing, Quadratic probing, Double hashing)

45. What is the time complexity of searching in a hash table?

(i) Best: O(1)
(ii) Worst: O(n)

46. What is load factor in hashing?
    Ratio of number of elements to size of hash table.
47. What is a balanced binary tree?
    A tree where the height difference of left and right subtree is ≤1.
48. What is AVL tree?
    A self-balancing binary search tree.
49. What is the maximum number of nodes in a binary tree of height h?
    2^(h+1) - 1
50. Applications of hashing?

(i) Symbol tables
(ii) Databases
(iii) Caches

(iv) Password storage