Data Structures Experiments

Ayush Jain | 60004200132 | B1

Academic Year – 2021-22

S.E Computer Engineering

Experiment 1

Aim: Implementation of Insertion sort, Selection sort menu driven program.
Implementation of Quick Sort.

Theory: A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator
on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

SELECTION SORT: - The selection sort algorithm sorts an array by repeatedly finding the minimum element
(considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two
subarrays in a given array. 1) The subarray which is already sorted. 2) Remaining subarray which is
unsorted. In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

INSERTION SORT: - Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in
your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

QUICK SORT: - Like Merge Sort, Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and
partitions the given array around the picked pivot. There are many different versions of quick Sort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

Code:

```
#include<stdio.h>
#include<stdlib.h> void
display(int a[],int n);
voidselection_sort(inta[],
int n); void
insertion_sort(int a[],int
n);
```

```
//-----------------Main Function----------------------
int main()
{
int n,choice,i; char
ch[20];
printf("Enter no. of elements u want to sort : ");
scanf("%d",&n);
int arr[n];
for(i=0;i<n;i++)
{
printf("Enter Element %d: ",i+1);
scanf("%d",&arr[i]);
}
printf("Please select any option Given Below for Sorting : \n");
while(1)
{
printf("\n1. Selection Sort\n2. Insertion Sort\n3. Display Array.\n4. Exit the  Program.\n");
printf("\nEnter your Choice : ");
scanf("%d",&choice);
switch(choice)
{
case 1:
selection_sort(arr,n);
break; case 2:
insertion_sort(arr,n);
break; case 3:
display(arr,n); break;
case 4: return 0;
default:
printf("\nWrong Option.\n");
}
}
return 0;
}
//End of main function
//Display Function void
display(int arr[],int n) {
for(int i=0;i<n;i++)
{
printf(" %d ",arr[i]);
}
}
//Selection Sort Function
void selection_sort(int arr[],int n)
```

```
{
int i,j,temp;
for(i=0;i<n-1;i++)
{
for(j=i+1;j<n;j++)
{
if(arr[i]>arr[j])
{
temp=arr[i];
arr[i]=arr[j];
arr[j]=temp;
}
}
}
printf("Selection Sort\n Elements are : "); display(arr,n);
}
//Insertion Sort Function
void insertion_sort(int arr[],int n)
{ int i,j,min;
for(i=1;i<n;i+
+)
{
min=arr[i];
j=i-1; while(min<arr[j]
&& j>=0)
{
arr[j+1]=arr[j];
j=j-1;
}
arr[j+1]=min;
}
printf("Insertion Sort:\n Elements are : "); display(arr,n);
}
```

Output:

Quick Sort Code:

```
#include<stdio.h>
void quicksort(int number[25],int first,int last){
int i, j, pivot, temp; if(first<last){ pivot=first;
i=first; j=last;
while(i<j){
while(number[i]<=number[pivot]&&i<last)
i++;
while(number[j]>number[pivot]) j--
;
if(i<j){
temp=number[i]; number[i]=number[j];
number[j]=temp;
}
}
temp=number[pivot];
number[pivot]=number[j]; number[j]=temp;
quicksort(number,first,j-1);
quicksort(number,j+1,last);
}
}
int main(){
int i, count, number[25]; printf("Number
of elements: "); scanf("%d",&count);
```

```
printf("Enter %d elements: ", count);
for(i=0;i<count;i++)
scanf("%d",&number[i]);
quicksort(number,0,count-1);
printf("Order of Sorted elements: ");
for(i=0;i<count;i++) printf("
%d",number[i]); return 0;
}
```

Output:

```
Number of elements: 5
Enter 5 elements: 9 6 3 8 5
Order of Sorted elements:  3 5 6 8 9
PS C:\Users\Gautam\Desktop\coding\DS.c>
```

Conclusion: In this experiment we have successfully implemented insertion sort and selection sort menu driven program and also implemented quick sort.

Experiment 2

Aim: Implementing Binary search.

Theory: A Binary Search is a sorting algorithm, that is used to search an element in a sorted array. A binary search technique works only on a sorted array, so an array must be sorted to apply binary search on the array. It is a searching technique that is better than the liner search technique as the number of iterations decreases in the binary search. The logic behind the binary search is that there is a key. This key holds the value to be searched. The highest and the lowest value are added and divided by 2. Highest and lowest and the first and last element in the array. The mid value is then compared with the key. If mid is equal to the key, then we get the output directly. Else if the key is greater than mid then the mid+1 becomes the lowest value and the process is repeated on the shortened array. Else if the key value is less then mid, mid-1 becomes the highest value and the process is repeated on the shortened array. If it is not found anywhere, an error message is displayed.

Code:

```
#include<stdio.h> int
iter_bin(int *arr,int n,int x)
{
int key,low,high,mid;
low=0;  high=n-1;  while
```

```
(low<=high){
mid=(low+high)/2;
if(arr[mid]==x) return
mid;
else if(arr[mid]>x)
{
high=mid-1;
}
else if(arr[mid]<x)
{
low=mid+1;
}
else return -1;
}
}
void main()
{ int n,i,j,x;  int
arr[20];  printf("Enter
size:");
scanf("%d",&n);
printf("Enter array:");
for(i=0;i<n;i++)
{scanf("%d",&arr[i]);
}
for(i=0;i<n-1;i++)
{
for(j=0;j<n-i-1;j++)
{
if(arr[j]>arr[j+1])
{
int t=arr[j];
arr[j]=arr[j+1]
;  arr[j+1]=t;
}
}
}
printf("Sorted  array:");
for          (i=0;i<n;i++)
printf("%d       ",arr[i]);
printf("\nenter       x:");
scanf("%d",&x);      int
bin=iter_bin(arr,n,x);
printf("found at %d",bin);
}
```

Output:

```
Enter size:5
Enter array:9 6 3 8 5
Sorted array:3 5 6 8 9
enter x:5
found at 1
PS C:\Users\Gautam\Desktop\coding\DS.c>
```

Conclusion: In this experiment we have successfully implemented binary search.

Experiment 3

Aim: Implementation of Linked Lists menu driven program

Theory: A Linked List is a linear data structure. Every linked list has two parts, the data section and the address section that holds the address of the next element in the list, which is called a node. The size of the linked list is not fixed, and data items can be added at any locations in the list. The disadvantage is that to get to a node, we must traverse to all the way from the first node to the node that we require. The Linked List is like an array but unlike an array, it is not stored sequentially in the memory.

Code:

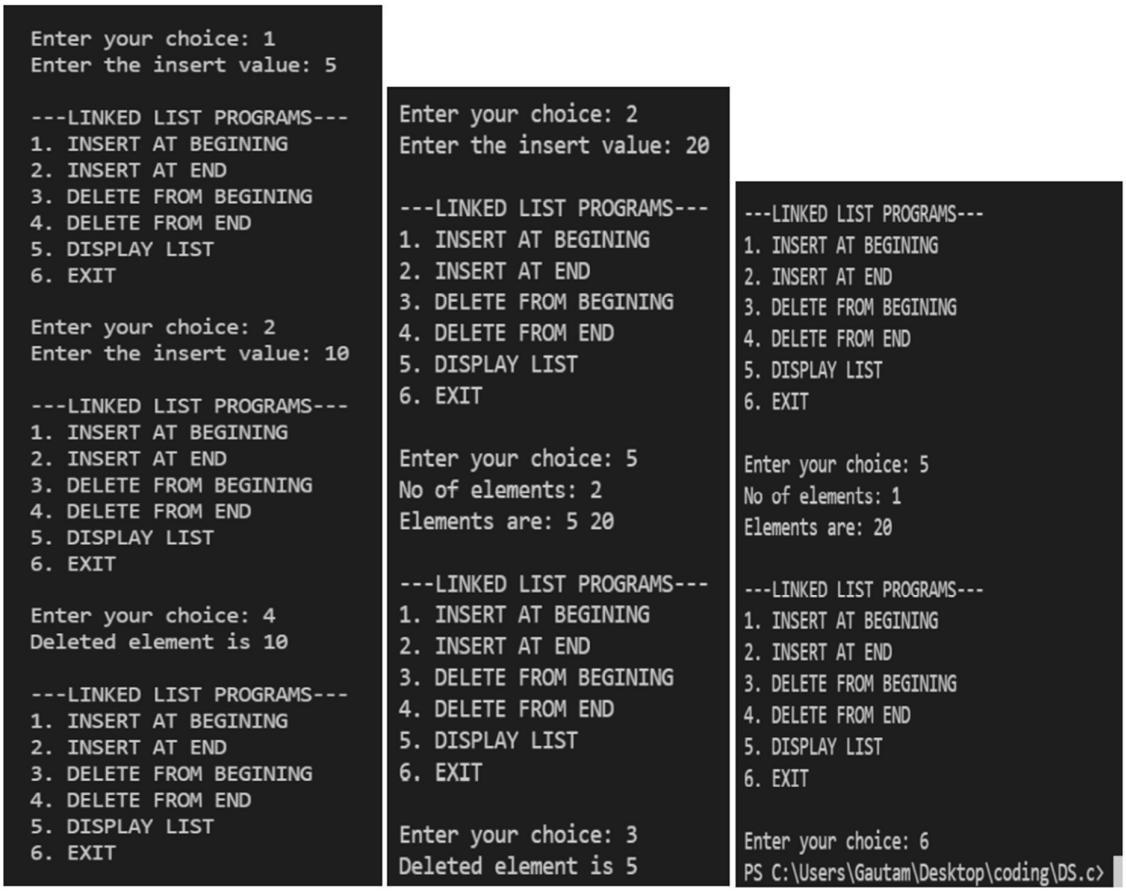
```
#include<stdio.h>
#include<stdlib.h> int
count=0; struct Node
*start=NULL; struct
Node
{
int data;
struct Node *next;
};
void insert_at_begin(int x)
{
struct Node *t;
t=(struct Node*)malloc(sizeof(struct Node)); count++;
if(start==NULL)
{
start=t; start->data=x;
start->next=NULL;
return;
}
t->data=x; t->next=start;
```

```
start=t;
}
void insert_at_end(int x)
{
struct Node *t,*temp; t=(struct
Node*)malloc(sizeof(struct Node));
count++;
if(start==NULL)
{
start=t; start->data=x;
start->next=NULL;
return; }
temp=start;
while(temp->next!=NULL)
{
temp=temp->next;
}
temp->next=t; t->data=x;
t->next=NULL;
}
void delete_from_begin()
{
struct Node *t; int
n;
if(start==NULL)
{
printf("Linked List is empty!!!\n\n"); return;
}
n=start->data; t=start-
>next; free(start); start=t;
count--; printf("Deleted
element is %d\n\n",n);
return; }
void delete_from_end()
{
struct Node *t,*u; int
n;
if(start==NULL)
{
printf("Linked List is
empty!!!\n\n"); return; }
count--;
if(start->next==NULL)
{
```

```
n=start->data;
free(start); start=NULL;
printf("Deleted element is %d\n\n",n);
return; }
t=start;
while(t->next!=NULL)
{
u=t;
t=t->next;
}
n=t->data; u-
>next=NULL;
free(t); printf("Deleted element is
%d\n\n",n); return; }
void display()
{
struct Node *t;
if(start==NULL)
{
printf("Linked List is
empty!!!\n\n"); return; }
printf("No of elements: %d\n",count); printf("Elements are: ");
t=start;
while(t->next!=NULL)
{
printf("%d ",t->data);
t=t->next;
}
printf("%d ",t->data);
printf("\n\n");
}
int main()
{
int ch,data;
while(1)
{
printf("---LINKED LIST PROGRAMS--
-\n"); printf("1. INSERT AT
BEGINING\n"); printf("2. INSERT AT
END\n"); printf("3. DELETE FROM
BEGINING\n"); printf("4. DELETE
FROM END\n"); printf("5. DISPLAY
LIST\n"); printf("6. EXIT\n\n");
```

```
printf("Enter your choice: ");
scanf("%d",&ch);
if(ch==1)
{
printf("Enter the insert value: ");
scanf("%d",&data); printf("\n");
insert_at_begin(data);
}
else if(ch==2)
{
printf("Enter the insert value: ");
scanf("%d",&data); printf("\n");
insert_at_end(data);
}
else if(ch==3)
{
delete_from_begin();
}
else if(ch==4)
{
delete_from_end();
}
else if(ch==5)
{
display();
}
else if(ch==6)
{
break;
}
else
{
printf("Wrong choice!!!\n");
}}}
```

Output:



Conclusion: In this experiment we have successfully implemented Linked Lists menu driven program.

Experiment 4

Aim: Implementations of Linked Lists menu driven program (stack and queue).

Theory: In linked list implementation of a stack, every new element is inserted as 'top' element. That means every newly inserted element is pointed by 'top'. Whenever we want to remove an element from the stack, simply remove
the node which is pointed by 'top' by moving 'top' to its previous node in the list. The next field of the first element
must be always NULL. The storage requirement of linked representation of a queue with n elements is o(n) while the
time requirement for operations is o (1). In a linked queue, each node of the queue consists of two parts i.e., data

part and the link part. Each element of the queue points to its immediate next element in the memory. In the linked queue, there are two pointers maintained in the memory i.e., front pointer and rear pointer. The front pointer
contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

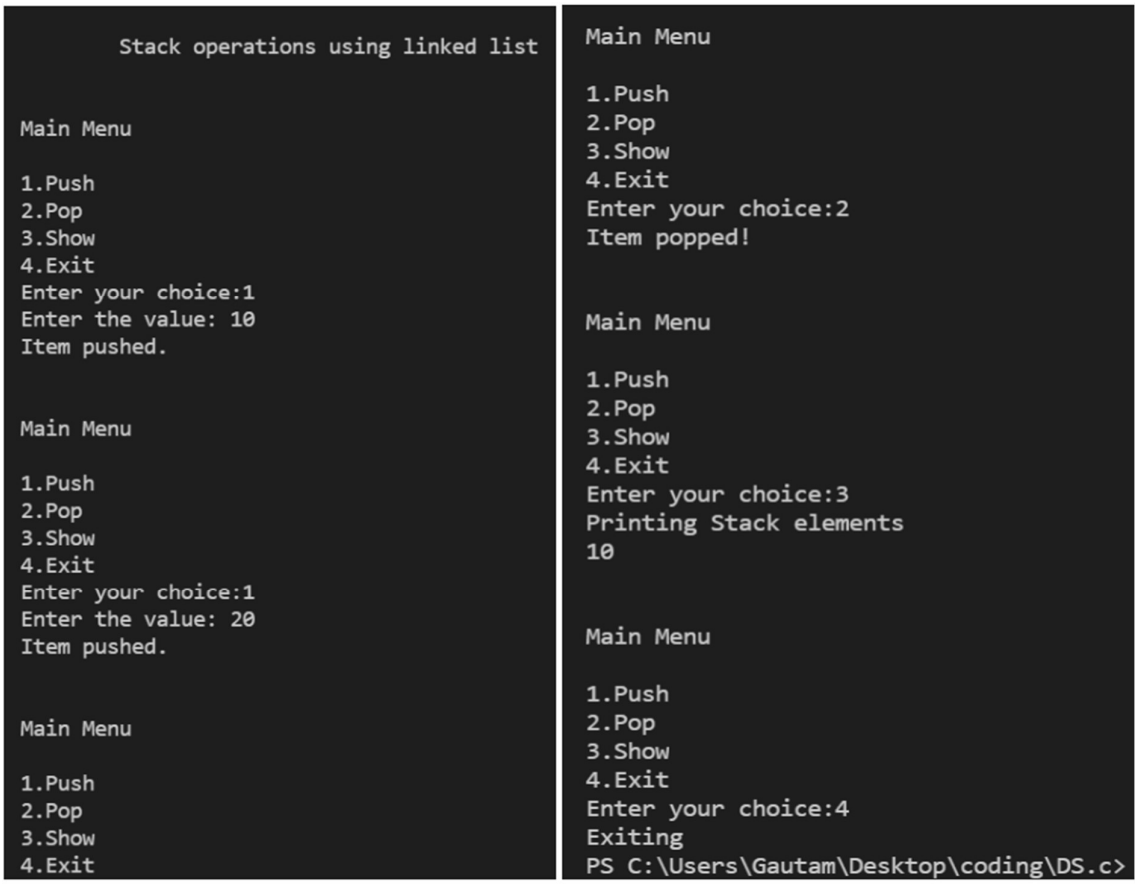
Code:

```
//Stack operations using linked list
#include <stdio.h>
#include <stdlib.h>
struct node
{
int val;
struct node *next;
};
struct node *head;
void push(); void
pop(); void
display();
void main ()
{
int choice=0;
printf("\n\tStack operations using linked list\t\n"); while(choice
!= 4)
{
printf("\n\nMain Menu\n");
printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
printf("\nEnter your choice:");
scanf("%d",&choice);
switch(choice)
{
case 1:
{
push();
break;
}
case 2:
{
pop();
break;
}
case 3:
{
display();
break;
```

```
}
case 4:
{
printf("Exiting");
break;
}
default:
{
printf("Please enter valid choice.");
}
};
}
}
void push ()
{ int
val;
struct node *ptr = (struct node*)malloc(sizeof(struct node)); if(ptr
== NULL)
{
printf("not able to push the element.\n");
}
else
{
printf("Enter the value: ");
scanf("%d",&val);
if(head==NULL)
{
ptr->val = val; ptr
-> next = NULL;
head=ptr;
}
else
{
ptr->val = val; ptr-
>next = head;
head=ptr;
}
printf("Item pushed.\n");
}
}
void pop()
{
int item; struct
node *ptr;
```

```
if (head == NULL)
{
printf("Underflow!\n");
}
else
{
item = head->val;
ptr = head; head =
head->next;
free(ptr);
printf("Item popped!\n");
}
}
void display()
{
int i;
struct node *ptr;
ptr=head; if(ptr
== NULL)
{
printf("Stack is empty\n");
}
else
{
printf("Printing Stack elements \n");
while(ptr!=NULL)
{
printf("%d\n",ptr->val);
ptr = ptr->next;
}
}
}
```

Output:



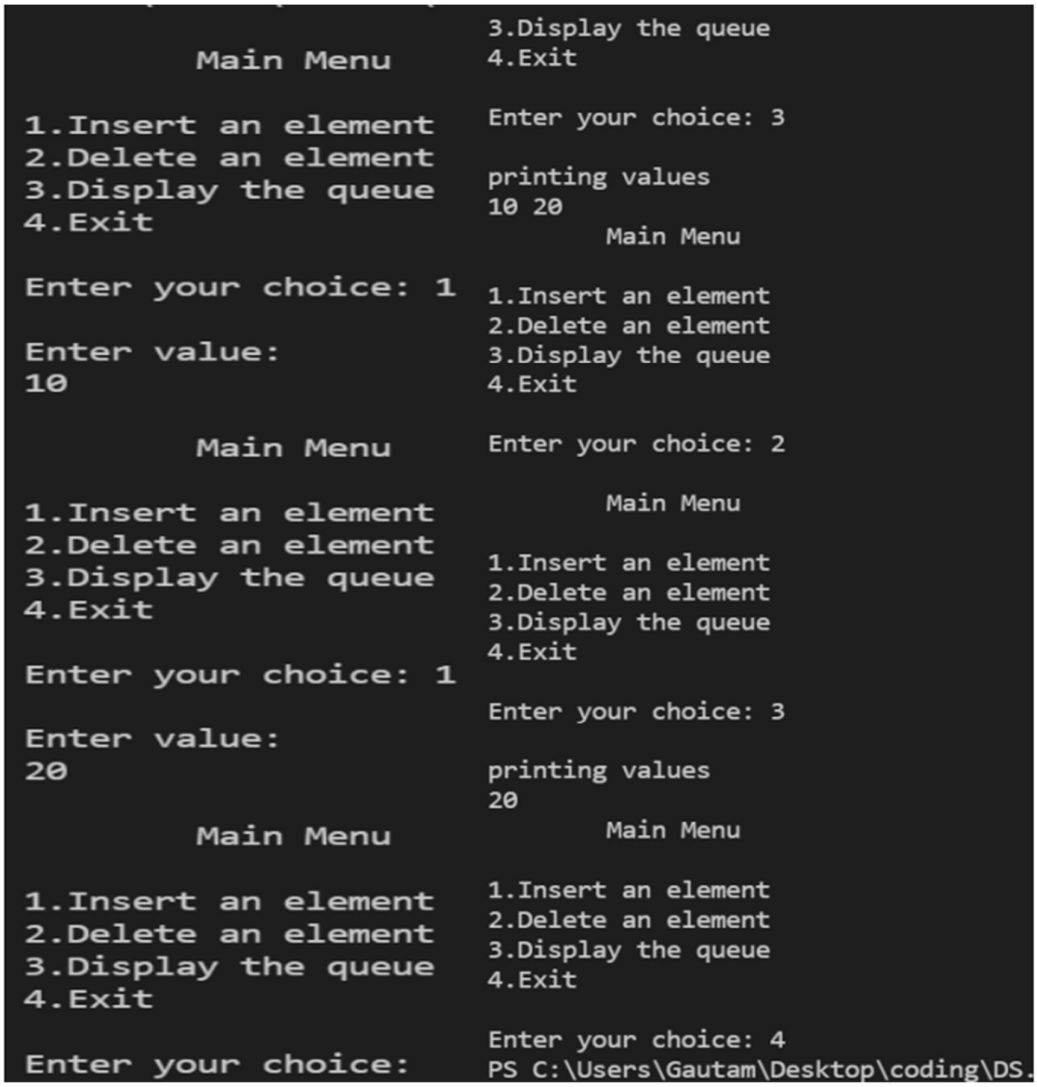
Code:

```
// Linked list implementation of queue
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *front;
struct node *rear;
void insert(); void
delete(); void
display();
void main ()
{
int                choice;
while(choice != 4) {
printf("\n\tMain Menu\t\n");
printf("\n1.Insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
printf("\nEnter your choice: "); scanf("%d",& choice);
```

```
switch(choice)
{
case 1:
insert();
break; case
2: delete();
break; case
3:
display();
break; case
4: exit(0);
break;
default:
printf("\nEnter valid choice.\n");
}
}
}
void insert()
{
struct node *ptr; int
item;
ptr = (struct node *) malloc (sizeof(struct node)); if(ptr
== NULL)
{
printf("\nOVERFLOW\n");
return;
}
else
{
printf("\nEnter value: \n");
scanf("%d",&item); ptr ->
data = item;
if(front == NULL)
{
front = ptr; rear = ptr;
front -> next =
NULL; rear -> next =
NULL;
}
else
{
rear -> next = ptr;
rear = ptr; rear-
>next = NULL;
```

```
}
}
}
void delete ()
{
struct node *ptr;
if(front == NULL)
{
printf("\nUNDERFLOW\n");
return;
}
else
{
ptr = front; front =
front -> next;
free(ptr);
}
}
void display()
{
struct node *ptr;
ptr = front; if(front
== NULL)
{
printf("\nEmpty queue.\n");
}
else { printf("\nprinting
values\n");
while(ptr != NULL)
{
printf("%d ",ptr -> data);
ptr = ptr -> next;
}
}
}
```

Output:



Conclusion: In this experiment we have successfully implemented linked lists menu driven program (stack and queue).

Experiment 5

Aim: Implementation of Infix to Postfix Transformation and its evaluation program. Theory: To convert infix expression to postfix expression, we will use the stack data structure. By scanning the infix
expression from left to right, when we will get any operand, simply add them to the postfix form, and for the operator and parenthesis, add them in the stack maintaining the precedence of them.

Code:

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

```
#define BLANK ' '
#define TAB '\t' #define MAX
50 void infix_to_postfix(); int
priority(char symbol); void
push(long int symbol); long
int pop(); int isEmpty(); int
white_space(char ); long int
eval_post(); char infix[MAX],
postfix[MAX]; long int
stack[MAX]; int top;
int main()
{
long int value; top=-1; printf("Enter infix :
"); gets(infix); infix_to_postfix();
printf("Postfix : %s\n",postfix);
value=eval_post(); printf("Value of
expression : %ld\n",value); return 0;
}/*End of main()*/
void infix_to_postfix()
{
unsigned int i,p=0; char
next; char symbol;
for(i=0;i<strlen(infix);i+
+)
{
symbol=infix[i];
if(!white_space(symbol))
{
switch(symbol)
{ case
'(':
push(symbol);
break; case ')':
while((next=pop())!='(')
postfix[p++] =
next; break; case
'+': case '-': case '*':
case '/': case '%':
case '^':
while( !isEmpty( ) && priority(stack[top])>=
priority(symbol) ) postfix[p++]=pop(); push(symbol); break;
default: /*if an operand comes*/
postfix[p++]=symbol;
}
}
```

```
}
while(!isEmpty( ))
postfix[p++]=pop();
postfix[p]='\0'; /*End postfix with'\0' to make it a string*/
}/*End of infix_to_postfix()*/
/*This function returns the priority of the operator*/ int
priority(char symbol)
{
switch(symbol)
{ case
'(':
return 0;
case '+':
case '-':
return 1;
case '*':
case '/':
case '%':
return 2;
case '^':
return 3;
default :
return 0;
}
}/*End of priority()*/
void push(long int symbol)
{
if(top>MAX)
{
printf("Stack overflow\n");
exit(1);
}
stack[++top]=symbol;
}/*End of push()*/
long int pop()
{
if( isEmpty() )
{
printf("Stack underflow\n");
exit(1);
}
return (stack[top--]); }/*End
of pop()*/
int isEmpty()
```

```
{
if(top==-1) return 1;
else return 0; }/*End
of isEmpty()*/
int white_space(char symbol)
{
if( symbol == BLANK || symbol == TAB )
return 1; else return 0;
}/*End of white_space()*/
long int eval_post()
{
long int a,b,temp,result; unsigned
int i;
for(i=0;i<strlen(postfix);i++)
{
if(postfix[i]<='9' && postfix[i]>='0') push(postfix[i]-
'0');
else
{
a=pop();
b=pop();
switch(postfix[i])
{
case '+':
temp=b+a; break; case
'-':
temp=b-a;break; case
'*':
temp=b*a;break; case
'/':
temp=b/a;break; case
'%':
temp=b%a;break; case
'^':
temp=pow(b,a);
}
push(temp);
}
}
result=pop(); return
result; }/*End of
eval_post */ Output:
```

```
Enter infix : 4 2 + 2 1 + /
Postfix : 4221+/+
Value of expression : 4
PS C:\Users\Gautam\Desktop\coding\DS.c>
```

Conclusion: In this experiment we have successfully implemented a program to convert Infix to Postfix transformation and its evaluation program.

Experiment 6

Aim: Implementation of double ended queue menu driven program

Theory: The dequeue stands for Double Ended Queue. In the queue, the insertion takes place from one end while
the deletion takes place from another end. The end at which the insertion occurs is known as the rear end whereas
the end at which the deletion occurs is known as front end. Deque is a linear data structure in which the insertion
and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

Code:

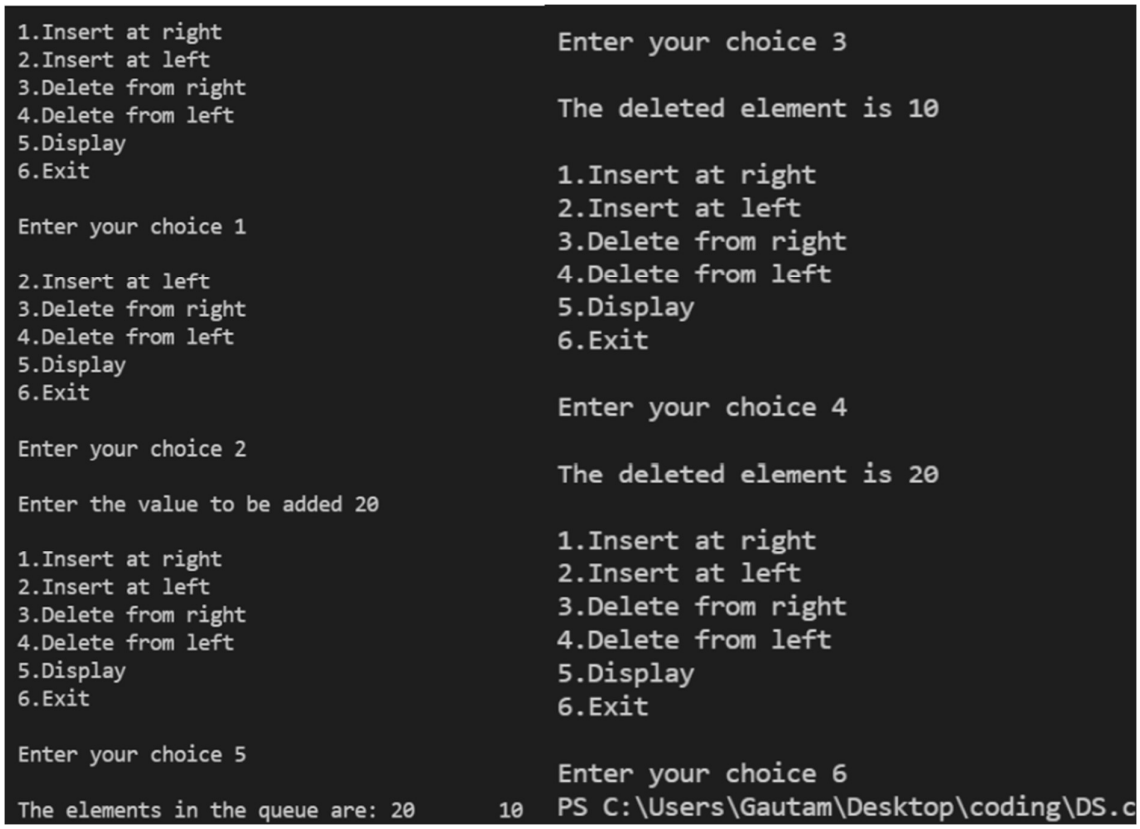
```
#include<stdio.h>
#include<conio.h>
#define MAX 10
int deque[MAX];
int left=-1, right=-1;
void insert_right(void);
void insert_left(void);
void delete_right(void);
void delete_left(void);
void display(void);
int main()
{
int choice;
do {
printf("\n1.Insert at right ");
printf("\n2.Insert at left ");
printf("\n3.Delete from right ");
printf("\n4.Delete from left ");
printf("\n5.Display ");
printf("\n6.Exit"); printf("\n\nEnter
your choice ");
scanf("%d",&choice);
switch(choice)
{
```

```
case 1:
insert_right();
break; case 2:
insert_left();
break; case 3:
delete_right();
break; case 4:
delete_left();
break; case 5:
display();
break;
}
}while(choice!=6);
return 0; }
//-------INSERT AT RIGHT-------
void insert_right()
{
int val;
printf("\nEnter the value to be added "); scanf("%d",&val);
if( (left==0 && right==MAX-1) || (left==right+1) )
{
printf("\nOVERFLOW");
}
if(left==-1) //if queue is initially empty
{
left=0;
right=0;
}
else
{
if(right==MAX-
1) right=0; else
right=right+1;
}
deque[right]=val;
}
//-------INSERT AT LEFT-------
void insert_left()
{
int val;
printf("\nEnter the value to be added "); scanf("%d",&val);
if( (left==0 && right==MAX-1) || (left==right+1) )
{
printf("\nOVERFLOW");
```

```
}
if(left==-1) //if queue is initially empty
{
left=0;
right=0;
}
else
{
if(left==0)
left=MAX-1;
else
left=left-1;
}
deque[left]=val;
}
//-------DELETE FROM RIGHT-------
void delete_right()
{
if(left==-1)
{
printf("\nUNDERFLOW");
return;
}
printf("\nThe deleted element is %d\n", deque[right]);
if(left==right) //Queue has only one element
{
left=-1;
right=-1;
}
else
{
if(right==0)
right=MAX-1;
else
right=right-1;
}
}
//-------DELETE FROM LEFT-------
void delete_left()
{
if(left==-1)
{
printf("\nUNDERFLOW");
return; }
```

```
printf("\nThe deleted element is %d\n", deque[left]);
if(left==right) //Queue has only one element
{
left=-1; right=-
1;
}
else
{
if(left==MAX-
1) left=0; else
left=left+1;
}
}
//-------DISPLAY-------
void display()
{
int front=left, rear=right;
if(front==-1)
{
printf("\nQueue is Empty\n");
return; }
printf("\nThe elements in the queue are: "); if(front<=rear)
{
while(front<=rear)
{
printf("%d\t",deque[front]);
front++;
}
}
else
{
while(front<=MAX-1)
{
printf("%d\t",deque[front]);
front++; }
front=0;
while(front<=rear)
{
printf("%d\t",deque[front]); front++;
}
}
printf("\n");
}
```

Output:



Conclusion: In this experiment we have successfully implemented double ended queue menu driven program.

Experiment 7

Aim: Implementation of queue menu driven program.

Theory: A Queue is a linear structure which follows a particular order in which the operations are performed. The
order is First in First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the
consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we
remove the item the most recently added; in a queue, we remove the item the least recently added.

Code:

```
#include <stdio.h>
#include<stdlib.h
> #define MAX
50 void insert();
void delete();
void display(); int
queue_array[MAX];
```

```
int rear = - 1; int front
= - 1;
int main()
{
int choice;
while (1)
{
printf("1.Insert element to queue \n");
printf("2.Delete element from queue \n");
printf("3.Display all elements of queue \n");
printf("4.Quit \n"); printf("Enter your
choice : "); scanf("%d", &choice);
switch(choice)
{
case 1: insert(); break;
case 2: delete(); break;
case 3: display(); break;
case 4: exit(1); default:
printf("Wrong choice
n");
}
}
}
void insert()
{
int item; if(rear == MAX - 1)
printf("Queue Overflow n");
else
{
if(front== - 1)
front = 0;
printf("Insert the element in queue : ");
scanf("%d", &item); rear = rear + 1;
queue_array[rear] = item;
}
}
void delete()
{
if(front == - 1 || front > rear)
{
printf("Queue Underflow"); printf("Queue
Underflow");
return;
}
```

```
else
{
printf("Element deleted from queue is : %d\n", queue_array[front]); front
= front + 1;
}
}
void display()
{
int i;
if(front == - 1)
printf("Queue is empty");
else
{
printf("Queue is : "); for(i =
front; i <= rear; i++)
printf("%d ",
queue_array[i]);
printf("\n");
}
}
```

Output:

```
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Insert the element in queue : 10
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Insert the element in queue : 20
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3
Queue is : 10 20
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 2
Element deleted from queue is : 10
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 4
PS C:\Users\Gautam\Desktop\coding\DS.c>
```

Conclusion: In this experiment we have successfully implemented program for Queue menu driven program.

Experiment 8

Aim: Implementation of BST program.

Theory: Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers. It is called a
binary tree because each tree node has a maximum of two children. It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time. The properties that separate a binary search tree from a
regular binary tree is: All nodes of left subtree are less than the root node All nodes of right subtree are more than
the root node Both subtrees of each node are also BSTs i.e., they have the above two properties.

Code:

```
#include<stdio.h>
#include<stdlib.h> typedef
struct BST
{
int data; struct
BST *left; struct
BST *right;
}node; node *create(); void
insert(node *,node *); void
preorder(node *);
int main()
{
char ch;
node *root=NULL,*temp;
do
{
temp=create();
if(root==NULL)
root=temp; else
insert(root,temp);
printf("Do you want to enter more(y/n): ");
getchar(); scanf("%c",&ch);
}while(ch=='y'|ch=='Y'); printf("Preorder
Traversal: "); preorder(root);
return 0;
}
node *create()
```

```
{
node *temp; printf("Enter data:");
temp=(node*)malloc(sizeof(node));
scanf("%d",&temp->data); temp-
>left=temp->right=NULL; return
temp;
}
void insert(node *root,node *temp)
{
if(temp->data<root->data)
{
if(root->left!=NULL)
insert(root->left,temp); else
root->left=temp;
}
if(temp->data>root->data)
{
if(root->right!=NULL) insert(root-
>right,temp);
else
root->right=temp;
}
}
void preorder(node *root)
{
if(root!=NULL)
{
printf("%d ",root->data); preorder(root->left);
preorder(root->right);
}
}
```

Output:

```
Enter data:9
Do you want to enter more(y/n): y
Enter data:6
Do you want to enter more(y/n): y
Enter data:3
Do you want to enter more(y/n): y
Enter data:8
Do you want to enter more(y/n): y
Enter data:5
Do you want to enter more(y/n): n
Preorder Traversal: 9 6 3 5 8
PS C:\Users\Gautam\Desktop\coding\DS.c>
```

Conclusion: In this experiment we have successfully implemented Binary Search Tree program.

Experiment 9

Aim: Implementation of Graph menu driven program (DFS & BFS)

Theory: To visit each node or vertex which is a connected component, tree-based algorithms are used. You can do
this easily by iterating through all the vertices of the graph, performing the algorithm on each vertex that is still unvisited when examined. Two algorithms are generally used for the traversal of a graph: Depth first search (DFS)
and Breadth first search (BFS). Depth-first Search (DFS) is an algorithm for searching a graph or tree data structure.
The algorithm starts at the root (top) node of a tree and goes as far as it can down a given branch (path), and then
backtracks until it finds an unexplored path, and then explores it. The algorithm does this until the entire graph has been explored.

Code:

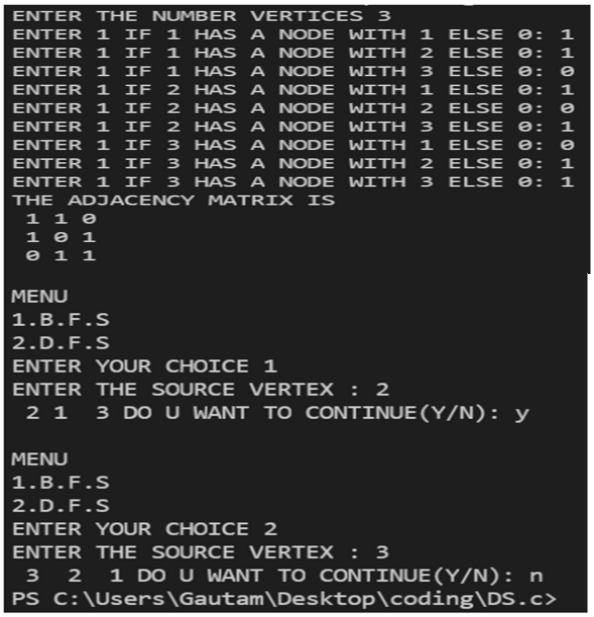
```
#include<stdio.h>
int q[20],top=-1,front=-1,rear=-
1,a[20][20],vis[20],stack[20]; int delete(); void add(int
item); void bfs(int s,int n); void dfs(int s,int n); void push(int
item);
int pop();
void main()
{
int n,i,s,ch,j;
char c,dummy;
printf("ENTER THE NUMBER
VERTICES "); scanf("%d",&n);
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
printf("ENTER 1 IF %d HAS A NODE WITH %d ELSE 0: ",i,j);
scanf("%d",&a[i][j]);
}
}
printf("THE ADJACENCY MATRIX IS\n");
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
```

```
{
printf(" %d",a[i][j]);
}
printf("\n");
}
do
{
for(i=1;i<=n;i++)
vis[i]=0;
printf("\nMENU");
printf("\n1.B.F.S");
printf("\n2.D.F.S"); printf("\nENTER
YOUR CHOICE"); scanf("%d",&ch);
printf("ENTER THE SOURCE
VERTEX :"); scanf("%d",&s);
switch(ch)
{
case 1:bfs(s,n);
break; case 2:
dfs(s,n);
break;
}
printf("DO U WANT TO
CONTINUE(Y/N): ");
scanf("%c",&dummy); scanf("%c",&c);
}while((c=='y')||(c=='Y'));
}
void bfs(int s,int n)
{ int p,i;
add(s);
vis[s]=1;
p=delete();
if(p!=0)
printf("
%d",p);
while(p!=0)
{
for(i=1;i<=n;i++)
if((a[p][i]!=0)&&(vis[i]==0))
{
add(i);
vis[i]=1;
}
p=delete();
```

```
if(p!=0)
printf(" %d ",p);
}
for(i=1;i<=n;i++)
if(vis[i]==0)
bfs(i,n);
}
void add(int item)
{
if(rear==19)
printf("QUEUE FULL");
else
{
if(rear==-1)
{
q[++rear]=item;
front++;
}
else
q[++rear]=item;
}
}
int delete() {
int k; if((front>rear)||(front==-1))
return(0);
else
{
k=q[front++]
; return(k); }
}
void dfs(int s,int n)
{ int i,k;
push(s);
vis[s]=1;
k=pop();
if(k!=0)
printf(" %d
",k);
while(k!=0)
{
for(i=1;i<=n;i++) if((a[k][i]!=0)&&(vis[i]==0))
{
push(i);
```

```
vis[i]=1;
}
k=pop();
if(k!=0)
printf(" %d
",k);
}
for(i=1;i<=n;i++)
if(vis[i]==0)
dfs(i,n);
}
void push(int item)
{
if(top==19) printf("Stack
overflow "); else
stack[++top]=item;
}
int pop()
{ int k;
if(top==-
1)
return(0);
else
{
k=stack[top--];
return(k);
}
}
```

Output:



Conclusion: In this experiment we have successfully implemented Graph menu driven program (breadth-first search and depth-first search)

Experiment 10

Aim: Implementation of hashing functions with different collision resolution techniques

Theory: A hash function is any function that can be used to map data of arbitrary size to fixed-size values. The values returned by a hash function are called hash values, hash codes, digests, or simply hashes. The values are usually used
to index a fixed-size table called a hash table. Use of a hash function to index a hash table is called hashing or scatter
storage addressing. Since a hash function gets us a small number for a key which is a big integer or string, there is a
possibility that two keys result in the same value. The situation where a newly inserted key maps to an already
occupied slot in the hash table is called collision and must be handled using some collision handling technique.

Code:

```
#include <stdio.h>
#include <stdlib.h>
struct set
{
int key;
int data;
};
struct set *array; int
capacity = 10;
int size = 0;
int hashFunction(int key)
{
return (key % capacity);
}
int checkPrime(int n)
{
int i;
if (n == 1 || n == 0)
{
return 0;
}
for (i = 2; i < n / 2; i++)
{
if (n % i == 0)
{
```

```
return 0;
}
}
return 1;
}
int getPrime(int n)
{
if (n % 2 == 0)
{
n++;
}
while (!checkPrime(n))
{
n += 2;
}
return n;
}
void init_array()
{
capacity = getPrime(capacity);
array = (struct set *)malloc(capacity * sizeof(struct set));
for (int i = 0; i < capacity; i++)
{
array[i].key = 0;
array[i].data = 0;
}
}
void insert(int key, int data)
{
int index = hashFunction(key);
if (array[index].data == 0)
{
array[index].key  =  key;
array[index].data = data;
size++;
printf("\n Key (%d) has been inserted \n", key);
}
else if (array[index].key == key)
{
array[index].data = data;
}
else
{
printf("\n Collision occured \n");
```

```
}
}
void remove_element(int key)
{
int index = hashFunction(key);
if (array[index].data == 0)
{
printf("\n This key does not exist \n");
}
else
{
array[index].key = 0; array[index].data
= 0;
size--;
printf("\n Key (%d) has been removed \n", key);
}
}
void display()
{
int i;
for (i = 0; i < capacity; i++)
{
if (array[i].data == 0)
{
printf("\n array[%d]: / ", i);
}
else
{
printf("\n key: %d array[%d]: %d \t", array[i].key, i, array[i].data);
}
}
}
int size_of_hashtable()
{
return size;
}
int main()
{
int choice, key, data, n;
int c = 0;
init_array();
do
{
```

```
printf("1.Insert item in the Hash Table" "\n2.Remove
item from the Hash Table"
"\n3.Check the size of Hash Table"
"\n4.Display a Hash Table" "\n\n
Please enter your choice: ");
scanf("%d", &choice);
switch (choice)
{
case 1:
printf("Enter key -:\t");
scanf("%d", &key);
printf("Enter data -:\t");
scanf("%d", &data);
insert(key, data); break;
case 2:
printf("Enter the key to delete-:");
scanf("%d", &key);
remove_element(key);
break; case
3:
n = size_of_hashtable(); printf("Size of
Hash Table is-:%d\n", n); break; case
4: display(); break; default:
printf("Invalid Input\n");
}
printf("\nDo you want to continue (press 1 for yes): "); scanf("%d",
&c);
} while (c == 1);
}
```

Output:

```
1.Insert item in the Hash Table
2.Remove item from the Hash Table
3.Check the size of Hash Table
4.Display a Hash Table

 Please enter your choice: 1
Enter key -:    3
Enter data -:   12

 Key (3) has been inserted

Do you want to continue (press 1 for yes): 1
1.Insert item in the Hash Table
2.Remove item from the Hash Table
3.Check the size of Hash Table
4.Display a Hash Table

 Please enter your choice: 3
Size of Hash Table is-:1

Do you want to continue (press 1 for yes): 1
1.Insert item in the Hash Table
2.Remove item from the Hash Table
3.Check the size of Hash Table
4.Display a Hash Table

 Please enter your choice: 4
```

```
array[0]: /
array[1]: /
array[2]: /
key: 3 array[3]: 12
array[4]: /
array[5]: /
array[6]: /
array[7]: /
array[8]: /
array[9]: /
array[10]: /
Do you want to continue (press 1 for yes):
```

Conclusion: In this experiment we have successfully implemented hashing functions with different collision resolution techniques.