

**SHARDA SCHOOL OF BASIC SCIENCES AND RESEARCH**

**Department of Mathematics**

**LAB REPORT FILE**

**Course Title: Data structure lab**

**Course Code: BDA 252**

**Program-Bachelor of Science (Hons.)-Data Science**

**Semester: 3rd**

**Session 2022-2023**

**Student Name: - Manish kumar**

**System ID: - 2021509877**

**Roll No. - 2107158014**

**Submitted to**

**Dr. Nair ul Islam**

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| S. No. | Date | Title | Signature |
|  |  |  |  |

**Practical-1**

* **Aim:**

Write a C program using dynamic variables and pointers, to construct a singly linked list:

* **The insertion operation**:

1. At the front of the list
2. At the back of the list
3. At any position of the list

* **The deletion operation:**

1. From the front of the list
2. From the back of the list
3. From any position of the list

* **Programs:**

**based on uses functions to perform the operations on singly linked list.**

**linked list: Creation & Traversal**

#include <stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node\*next;

};

void linkedListTraversal (struct Node\*ptr)

{

while (ptr !=NULL)

{

printf("element :%d\n",ptr-> data);

ptr=ptr->next;

}

}

int main() {

struct Node\*head;

struct Node\*second;

struct Node\*third;

head=(struct Node\*)malloc(sizeof(struct Node));

second=(struct Node\*)malloc(sizeof(struct Node));

third=(struct Node\*)malloc(sizeof(struct Node));

head-> data=7;

head->next=second;

second-> data=11;

second->next=third;

third-> data=41;

third->next=NULL;

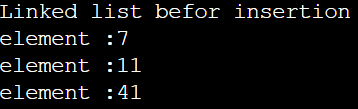
printf("Linked list befor insertion\n");

linkedListTraversal(head);

return 0;

}

* Output:



1. **Insertion at the front of the linked list:**

#include <stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node\*next;

};

void linkedListTraversal (struct Node\*ptr)

{

while (ptr !=NULL)

{

printf("element :%d\n",ptr-> data);

ptr=ptr->next;

}

}

// insertion at first position

struct Node\*insertatfirstNode(struct Node\*head,int data)

{

struct Node\*ptr=(struct Node\*)malloc(sizeof(struct Node));

ptr->data=data;

ptr->next=head;

return ptr;

}

int main() {

struct Node\*head;

struct Node\*second;

struct Node\*third;

head=(struct Node\*)malloc(sizeof(struct Node));

second=(struct Node\*)malloc(sizeof(struct Node));

third=(struct Node\*)malloc(sizeof(struct Node));

head-> data=7;

head->next=second;

second-> data=11;

second->next=third;

third-> data=41;

third->next=NULL;

printf("Linked list befor insertion\n");

linkedListTraversal(head);

head=insertatfirstNode(head,45);

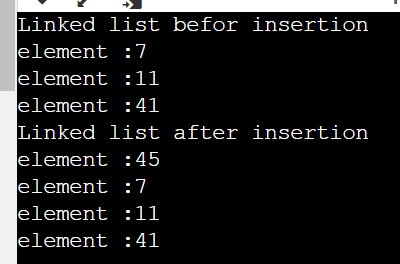
printf("Linked list after insertion\n");

linkedListTraversal(head);

return 0;

}

* **Output:**

****

1. **Insertion at the back of the linked list :**

#include <stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node\*next;

};

void linkedListTraversal (struct Node\*ptr)

{

while (ptr !=NULL)

{

printf("element :%d\n",ptr-> data);

ptr=ptr->next;

}

}

// insert at back position

struct Node\*insertatBackNode(struct Node\*head,int data)

{

struct Node\*ptr=(struct Node\*)malloc(sizeof(struct Node));

ptr->data=data;

struct Node\*p=head;

while(p->next!=NULL){

p=p->next;

}

p->next=ptr;

ptr->next=NULL;

return head;

}

int main() {

struct Node\*head;

struct Node\*second;

struct Node\*third;

head=(struct Node\*)malloc(sizeof(struct Node));

second=(struct Node\*)malloc(sizeof(struct Node));

third=(struct Node\*)malloc(sizeof(struct Node));

head-> data=7;

head->next=second;

second-> data=11;

second->next=third;

third-> data=41;

third->next=NULL;

printf("Linked list befor insertion\n");

linkedListTraversal(head);

head=insertatBackNode(head,45);

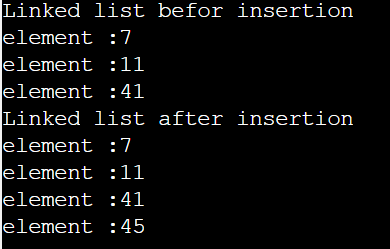
printf("Linked list after insertion\n");

linkedListTraversal(head);

return 0;

}

* **OUTPUT:**

****

1. **Insertion at any position of the linked list:**

#include <stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node\*next;

};

void linkedListTraversal (struct Node\*ptr)

{

while (ptr !=NULL)

{

printf("element :%d\n",ptr-> data);

ptr=ptr->next;

}

}

// insert at any position

struct Node \* insertAtIndex(struct Node \*head, int data, int index){

struct Node \* ptr = (struct Node \*) malloc(sizeof(struct Node));

struct Node \* p = head;

int i = 0;

while (i!=index-1)

{

p = p->next;

i++;

}

ptr->data = data;

ptr->next = p->next;

p->next = ptr;

return head;

}

int main() {

struct Node\*head;

struct Node\*second;

struct Node\*third;

head=(struct Node\*)malloc(sizeof(struct Node));

second=(struct Node\*)malloc(sizeof(struct Node));

third=(struct Node\*)malloc(sizeof(struct Node));

head-> data=7;

head->next=second;

second-> data=11;

second->next=third;

third-> data=41;

third->next=NULL;

printf("Linked list befor insertion\n");

linkedListTraversal(head);

head=insertAtIndex(head,45,1);

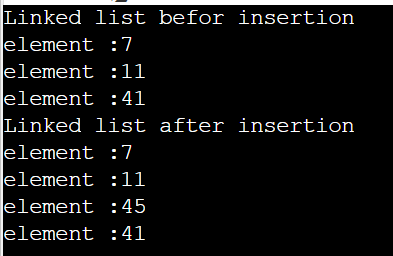
printf("Linked list after insertion\n");

linkedListTraversal(head);

return 0;

}

* **Output:**

****

* **The deletion operation:**
* **From the front of the list:**

#include <stdio.h>

#include <stdlib.h>

struct Node

{

int data;

struct Node \*next;

};

void linkedListTraversal(struct Node \*ptr)

{

while (ptr != NULL)

{

printf("Element: %d\n", ptr->data);

ptr = ptr->next;

}

}

// Deleting the first element from the linked list

struct Node \* deleteFirst(struct Node \* head){

struct Node \* ptr = head;

head = head->next;

free(ptr);

return head;

}

int main()

{

struct Node \*head;

struct Node \*second;

struct Node \*third;

struct Node \*fourth;

head = (struct Node \*)malloc(sizeof(struct Node));

second = (struct Node \*)malloc(sizeof(struct Node));

third = (struct Node \*)malloc(sizeof(struct Node));

fourth = (struct Node \*)malloc(sizeof(struct Node));

head->data = 4;

head->next = second;

second->data = 3;

second->next = third;

third->data = 8;

third->next = fourth;

fourth->data = 1;

fourth->next = NULL;

printf("Linked list before deletion\n");

linkedListTraversal(head);

head = deleteFirst(head);

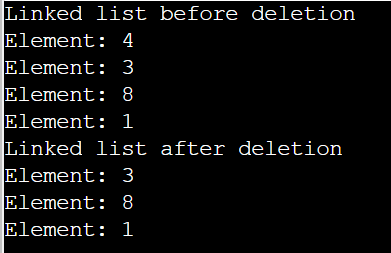
printf("Linked list after deletion\n");

linkedListTraversal(head);

return 0;

}

* **Output:**

****

* **From the back of the list:**

#include <stdio.h>

#include <stdlib.h>

struct Node

{

int data;

struct Node \*next;

};

void linkedListTraversal(struct Node \*ptr)

{

while (ptr != NULL)

{

printf("Element: %d\n", ptr->data);

ptr = ptr->next;

}

}

// Case 3: Deleting the last element

struct Node \* deleteAtLast(struct Node \* head){

struct Node \*p = head;

struct Node \*q = head->next;

while(q->next !=NULL)

{

p = p->next;

q = q->next;

}

p->next = NULL;

free(q);

return head;

}

int main()

{

struct Node \*head;

struct Node \*second;

struct Node \*third;

struct Node \*fourth;

head = (struct Node \*)malloc(sizeof(struct Node));

second = (struct Node \*)malloc(sizeof(struct Node));

third = (struct Node \*)malloc(sizeof(struct Node));

fourth = (struct Node \*)malloc(sizeof(struct Node));

head->data = 4;

head->next = second;

second->data = 3;

second->next = third;

third->data = 8;

third->next = fourth;

fourth->data = 1;

fourth->next = NULL;

printf("Linked list before deletion\n");

linkedListTraversal(head);

head = deleteAtLast(head);

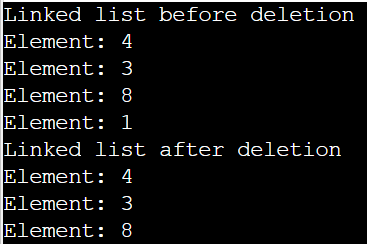
printf("Linked list after deletion\n");

linkedListTraversal(head);

return 0;

}

* **Output:**

****

* **From any position of the list:**

#include <stdio.h>

#include <stdlib.h>

struct Node

{

int data;

struct Node \*next;

};

void linkedListTraversal(struct Node \*ptr)

{

while (ptr != NULL)

{

printf("Element: %d\n", ptr->data);

ptr = ptr->next;

}

}

// Case 2: Deleting the element at a given index from the linked list

struct Node \* deleteAtIndex(struct Node \* head, int index){

struct Node \*p = head;

struct Node \*q = head->next;

for (int i = 0; i < index-1; i++)

{

p = p->next;

q = q->next;

}

p->next = q->next;

free(q);

return head;

}

int main()

{

struct Node \*head;

struct Node \*second;

struct Node \*third;

struct Node \*fourth;

head = (struct Node \*)malloc(sizeof(struct Node));

second = (struct Node \*)malloc(sizeof(struct Node));

third = (struct Node \*)malloc(sizeof(struct Node));

fourth = (struct Node \*)malloc(sizeof(struct Node));

head->data = 4;

head->next = second;

second->data = 3;

second->next = third;

third->data = 8;

third->next = fourth;

fourth->data = 1;

fourth->next = NULL;

printf("Linked list before deletion\n");

linkedListTraversal(head);

head = deleteAtIndex(head,2);

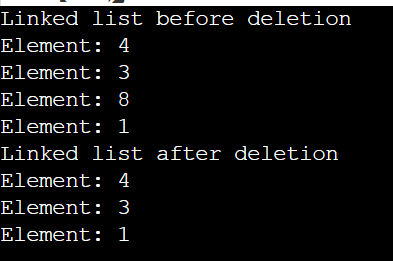
printf("Linked list after deletion\n");

linkedListTraversal(head);

return 0;

}

* **OUTPUT:**

****

**Practical-2**

* **Aim:**

Write a C program using dynamic variables and pointers, to construct a doubly linked list:

a. **The insertion operation:**

1. At the front of the list
2. At the back of the list
3. At any position of the list

b**. The deletion operation:**

1. From the front of the list
2. From the back of the list
3. From any position of the list

c. **Displaying all nodes of a file.**

* **Program:**

**based on uses functions to perform the operations on doubly linked list.**

* **The insertion operation:**

**At the front of the list**

#include<stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node \*next;

};

void linkedListTraversal(struct Node \*head){

struct Node \*ptr = head;

do{

printf("Element is %d\n", ptr->data);

ptr = ptr->next;

}while(ptr!=head);

}

struct Node \* insertAtFirst(struct Node \*head, int data){

struct Node \* ptr = (struct Node \*) malloc(sizeof(struct Node));

ptr->data = data;

struct Node \* p = head->next;

while(p->next != head){

p = p->next;

}

// At this point p points to the last node of this circular linked list

p->next = ptr;

ptr->next = head;

head = ptr;

return head;

}

int main(){

struct Node \*head;

struct Node \*second;

struct Node \*third;

struct Node \*fourth;

// Allocate memory for nodes in the linked list in Heap

head = (struct Node \*)malloc(sizeof(struct Node));

second = (struct Node \*)malloc(sizeof(struct Node));

third = (struct Node \*)malloc(sizeof(struct Node));

fourth = (struct Node \*)malloc(sizeof(struct Node));

// Link first and second nodes

head->data = 4;

head->next = second;

// Link second and third nodes

second->data = 3;

second->next = third;

// Link third and fourth nodes

third->data = 6;

third->next = fourth;

// Terminate the list at the third node

fourth->data = 1;

fourth->next = head;

printf("Circular Linked list before insertion\n");

linkedListTraversal(head);

head = insertAtFirst(head, 54);

head = insertAtFirst(head, 58);

head = insertAtFirst(head, 59);

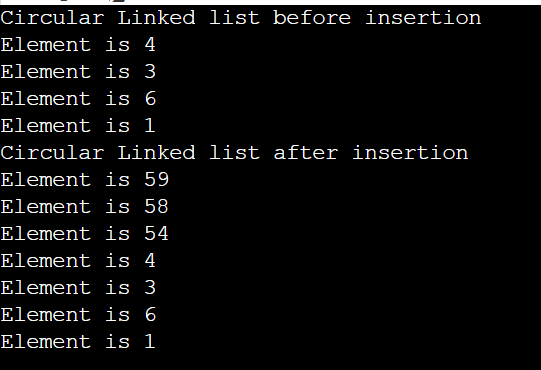
printf("Circular Linked list after insertion\n");

linkedListTraversal(head);

return 0;

}

* **Output:**

****

**The deletion operation:**

* From the front of the list
* From the back of the list
* From any position of the list
* **Code:**

**#include <stdio.h>**

#include <stdlib.h>

struct Node{

int data;

struct Node \*next;

struct Node \*prev;

};

int getLength(struct Node\* node);

void insert(struct Node\*\* head, int data){

struct Node\* freshNode = (struct Node\*) malloc(sizeof(struct Node));

freshNode->data = data;

freshNode->next = \*head;

freshNode->prev = NULL;

if(\*head !=NULL)

(\*head)->prev = freshNode;

\*head = freshNode;

}

void deleteFront(struct Node\*\* head)

{

struct Node\* tempNode = \*head;

if(\*head == NULL){

printf("Linked List Empty, nothing to delete\n");

return;

}

if(tempNode->next == NULL){

printf("%d deleted\n", tempNode->data);

\*head = NULL;

return;

}

\*head = (\*head)->next;

(\*head)->prev = NULL;

printf("%d deleted\n", tempNode->data);

free(tempNode);

}

void deleteEnd(struct Node\*\* head){

struct Node\* tempNode = \*head;

if(\*head == NULL){

printf("Linked List Empty, nothing to delete\n");

return;

}

if(tempNode->next == NULL){

printf("%d deleted\n", tempNode->data);

\*head = NULL;

return;

}

while (tempNode->next != NULL)

tempNode = tempNode->next;

struct Node\* secondLast = tempNode->prev;

secondLast->next = NULL;

printf("%d deleted\n", tempNode->data);

free(tempNode);

}

void deleteNthNode(struct Node\*\* head, int n){

int len = getLength(\*head);

if(n < 1 || n > len){

printf("Enter valid position\n");

return;

}

if(n == 1){

deleteFront(head);

return;

}

if(n == len){

deleteEnd(head);

return;

}

struct Node\* tempNode = \*head;

while (--n){

tempNode = tempNode->next;

}

struct Node\* previousNode = tempNode->prev;

struct Node\* nextNode = tempNode->next;

previousNode->next = tempNode->next;

nextNode->prev = tempNode->prev;

printf("%d deleted \n", tempNode->data);

free(tempNode);

}

int getLength(struct Node\* node){

int len = 0;

while(node!=NULL){

node = node->next;

len++;

}

return len;

}

void display(struct Node\* node)

{

struct Node \*end = NULL;

printf("List in Forward direction: ");

while (node != NULL) {

printf(" %d ", node->data);

end = node;

node = node->next;

}

printf("\nList in backward direction:");

while (end != NULL) {

printf(" %d ", end->data);

end = end->prev;

}

printf("\n\n");

}

int main()

{

struct Node\* head = NULL;

insert(&head,7);

insert(&head,8);

insert(&head,9);

insert(&head,10);

insert(&head,11);

insert(&head,12);

display(head);

deleteFront(&head);

display(head);

deleteEnd(&head);

display(head);

deleteNthNode(&head, 3);

display(head);

deleteNthNode(&head, 1);

display(head);

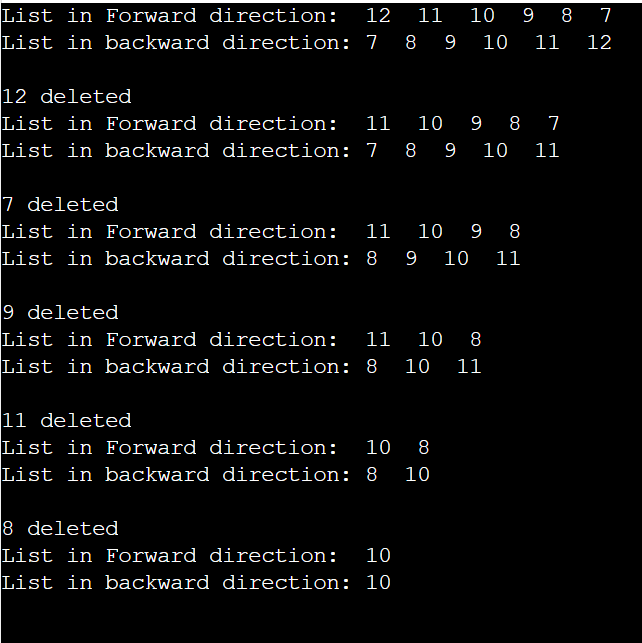
deleteEnd(&head);

display(head);

return 0;

}

* **OUTPUT:**



**Practical-3**

* **Aim:**

Write a C program to construct a stack of integers to perform the following operations:

a. Push

b. Pop

c. Display

It must print appropriate messages for Overflow, Underflow and empty.

* **Program:**

**based on implementation of stack and its operations.**

#include<stdio.h>

#include<stdlib.h>

struct stack{

int size ;

int top;

int \* arr;

};

int isEmpty(struct stack\* ptr){

if(ptr->top == -1){

return 1;

}

else{

return 0;

}

}

int isFull(struct stack\* ptr){

if(ptr->top == ptr->size - 1){

return 1;

}

else{

return 0;

}

}

void push(struct stack\* ptr, int val){

if(isFull(ptr)){

printf("Stack Overflow! Cannot push %d to the stack\n", val);

}

else{

ptr->top++;

ptr->arr[ptr->top] = val;

}

}

int pop(struct stack\* ptr){

if(isEmpty(ptr)){

printf("Stack Underflow! Cannot pop from the stack\n");

return -1;

}

else{

int val = ptr->arr[ptr->top];

ptr->top--;

return val;

}

}

int main(){

struct stack \*sp = (struct stack \*) malloc(sizeof(struct stack));

sp->size = 10;

sp->top = -1;

sp->arr = (int \*) malloc(sp->size \* sizeof(int));

printf("Stack has been created successfully\n");

printf("Before pushing, Full: %d\n", isFull(sp));

printf("Before pushing, Empty: %d\n", isEmpty(sp));

push(sp, 1);

push(sp, 23);

push(sp, 99);

push(sp, 75);

push(sp, 3);

push(sp, 64);

push(sp, 57);

push(sp, 46);

push(sp, 89);

push(sp, 6); // ---> Pushed 10 values

// push(sp, 46); // Stack Overflow since the size of the stack is 10

printf("After pushing, Full: %d\n", isFull(sp));

printf("After pushing, Empty: %d\n", isEmpty(sp));

printf("Popped %d from the stack\n", pop(sp)); // --> Last in first out!

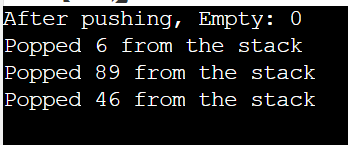
printf("Popped %d from the stack\n", pop(sp));

printf("Popped %d from the stack\n", pop(sp));

return 0;

}

* **OUTPUT:**

****

**Practical-4**

* **Aim:**

Write a C program using dynamic variables and pointers to construct a queue of integers using a singly linked list to perform the following operations:

a. Insert

b. Delete

c. Display

It must print appropriate messages for queue full and empty.

* **Program:**

**based on the implementation of Queue and its operations using linked list.**

#include <stdio.h>

#include <stdlib.h>

struct Node \*f = NULL;

struct Node \*r = NULL;

struct Node

{

int data;

struct Node \*next;

};

void linkedListTraversal(struct Node \*ptr)

{

printf("Printing the elements of this linked list\n");

while (ptr != NULL)

{

printf("Element: %d\n", ptr->data);

ptr = ptr->next;

}

}

void enqueue(int val)

{

struct Node \*n = (struct Node \*) malloc(sizeof(struct Node));

if(n==NULL){

printf("Queue is Full");

}

else{

n->data = val;

n->next = NULL;

if(f==NULL){

f=r=n;

}

else{

r->next = n;

r=n;

}

}

}

int dequeue()

{

int val = -1;

struct Node \*ptr = f;

if(f==NULL){

printf("Queue is Empty\n");

}

else{

f = f->next;

val = ptr->data;

free(ptr);

}

return val;

}

int main()

{

linkedListTraversal(f);

printf("Dequeuing element %d\n", dequeue());

enqueue(34);

enqueue(4);

enqueue(7);

enqueue(17);

printf("Dequeuing element %d\n", dequeue());

printf("Dequeuing element %d\n", dequeue());

printf("Dequeuing element %d\n", dequeue());

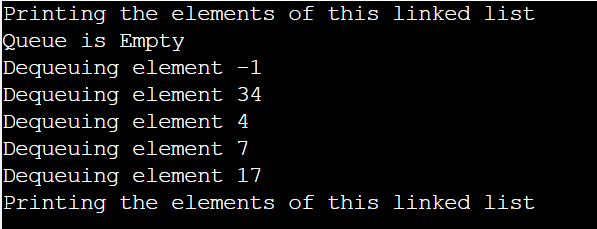
printf("Dequeuing element %d\n", dequeue());

linkedListTraversal(f);

return 0;

}

* **Output:**

****

* **Code:**

**b. Delete:**

#include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node \*next;

} \*front, \*back;

//Create an empty queue

void initialize() {

front = back = NULL;

}

// Returns queue size

int getQueueSize() {

struct node \*temp = front;

int count = 0;

if(front == NULL && back == NULL)

return 0;

while(temp != back){

count++;

temp = temp->next;

}

if(temp == back)

count++;

return count;

}

int getFrontElement() {

return front->data;

}

int getBackElement() {

return back->data;

}

void isEmpty() {

if (front == NULL && back == NULL)

printf("Empty Queue\n");

else

printf("Queue is not Empty\n");

}

//Adding elements in Queue

void enqueue(int num) {

struct node \*temp;

temp = (struct node \*)malloc(sizeof(struct node));

temp->data = num;

temp->next = NULL;

if (back == NULL) {

front = back = temp;

} else {

back->next = temp;

back = temp;

}

}

//Removes an element from front of the queue

void dequeue() {

struct node \*temp;

if (front == NULL) {

printf("\nQueue is Empty \n");

return;

} else {

temp = front;

front = front->next;

if(front == NULL){

back = NULL;

}

printf("Removed Element : %d\n", temp->data);

free(temp);

}

}

//Print's Queue

void printQueue() {

struct node \*temp = front;

if ((front == NULL) && (back == NULL)) {

printf("Queue is Empty\n");

return;

}

while (temp != NULL) {

printf("%d", temp->data);

temp = temp->next;

if(temp != NULL)

printf("-->");

}

}

int main() {

initialize();

// Adding elements in Queue

enqueue(5);

enqueue(6);

enqueue(1);

enqueue(3);

enqueue(9);

printQueue();

printf("\nSize of Queue : %d\n", getQueueSize());

printf("Front Element : %d\n", getFrontElement());

printf("Rear Element : %d\n", getBackElement());

//remove Element for Queue

dequeue();

dequeue();

dequeue();

dequeue();

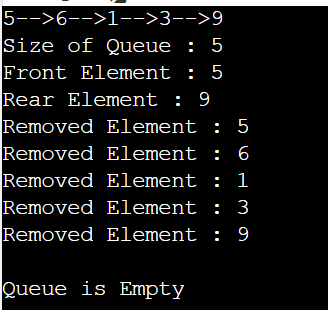
dequeue();

dequeue();

return 0;

}

* **OUTPUT:**

****

**Practical-5**

* **Aim:**

Design, develop, and execute a program in C to simulate the working of a circular queue of integers using an array. Provide the following operations:

a. Insert

b. Delete

c. Display

* **Problem:**

based on the implementation of circuler Queue and its operations using array.

* **Code:**

#include <stdio.h>

# define max 6

int queue[max]; // array declaration

int front=-1;

int rear=-1;

// function to insert an element in a circular queue

void enqueue(int element)

{

if(front==-1 && rear==-1) // condition to check queue is empty

{

front=0;

rear=0;

queue[rear]=element;

}

else if((rear+1)%max==front) // condition to check queue is full

{

printf("Queue is overflow..");

}

else

{

rear=(rear+1)%max; // rear is incremented

queue[rear]=element; // assigning a value to the queue at the rear position.

}

}

// function to delete the element from the queue

int dequeue()

{

if((front==-1) && (rear==-1)) // condition to check queue is empty

{

printf("\nQueue is underflow..");

}

else if(front==rear)

{

printf("\nThe dequeued element is %d", queue[front]);

front=-1;

rear=-1;

}

else

{

printf("\nThe dequeued element is %d", queue[front]);

front=(front+1)%max;

}

}

// function to display the elements of a queue

void display()

{

int i=front;

if(front==-1 && rear==-1)

{

printf("\n Queue is empty..");

}

else

{

printf("\nElements in a Queue are :");

while(i<=rear)

{

printf("%d,", queue[i]);

i=(i+1)%max;

}

}

}

int main()

{

int choice=1,x; // variables declaration

while(choice<4 && choice!=0) // while loop

{

printf("\nPress 1: Insert an element");

printf("\nPress 2: Delete an element");

printf("\nPress 3: Display the element");

printf("\nEnter your choice:");

scanf("%d", &choice);

switch(choice)

{

case 1:

printf("Enter the element which is to be inserted:");

scanf("%d", &x);

enqueue(x);

break;

case 2:

dequeue();

break;

case 3:

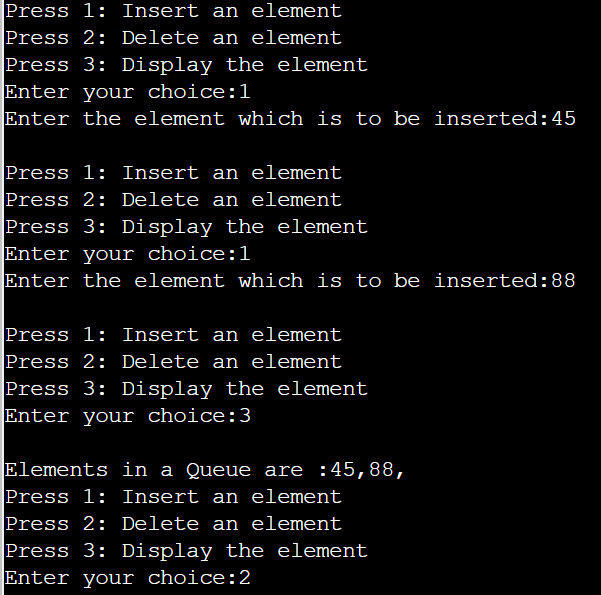
display();

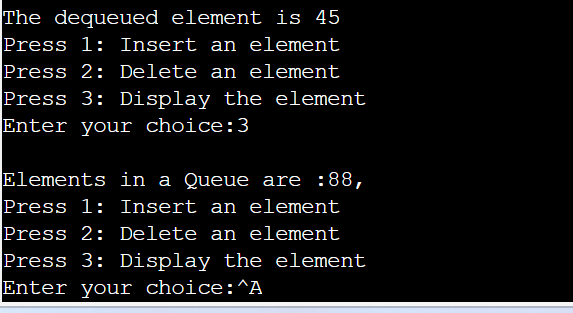
}}

return 0;

}

* **OUTPUT:**

****

****

**Practical-6**

* **Aim:**

Program to implement recursive binary search.

* **Problem:**

based on recursive function to perform binary search.

**Code:**

#include<stdio.h>

int linearSearch(int arr[], int size, int element){

for (int i = 0; i < size; i++)

{

if(arr[i]==element){

return i;

}

}

return -1;

}

int binarySearch(int arr[], int size, int element){

int low, mid, high;

low = 0;

high = size-1;

// Keep searching until low <= high

while(low<=high){

mid = (low + high)/2;

if(arr[mid] == element){

return mid;

}

if(arr[mid]<element){

low = mid+1;

}

else{

high = mid -1;

}

}

return -1;

}

int main(){

// Sorted array for binary search

int arr[] = {1,3,5,56,64,73,123,225,444};

int size = sizeof(arr)/sizeof(int);

int element = 444;

int searchIndex = binarySearch(arr, size, element);

printf("The element %d was found at index %d \n", element, searchIndex);

return 0;

}

* Output:



Practical-7

* **Aim:**

Write a program to sort given set of numbers in ascending/descending order using insertion sort.

* **Problem:**

based on insertion sort**.**

**Code:**

//C Program to sort an array in ascending order using Insertion Sort \*/

#include <stdio.h>

int main(void)

{

int n, i, j, temp;

int arr[64];

printf("Enter number of elements\n");

scanf("%d", &n);

printf("Enter %d integers\n", n);

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

{

scanf("%d", &arr[i]);

}

for (i = 1; i < n; i++)

{

j = i;

while (j > 0 && arr[j - 1] > arr[j])

{

temp = arr[j];

arr[j] = arr[j - 1];

arr[j - 1] = temp;

j--;

}

}

printf("Sorted list in ascending order:\n");

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

{

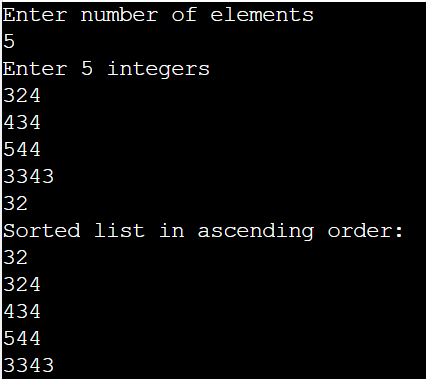
printf("%d\n", arr[i]);

}

return 0;

}

* **Output:**

****

**Practical-8**

* **Aim:**

Write a program to sort given set of numbers in ascending/descending order using Selection sort.

* **Problem:**

based on selection sort.

#include<stdio.h>

void printArray(int\* A, int n){

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

{

printf("%d ", A[i]);

}

printf("\n");

}

void selectionSort(int \*A, int n){

int indexOfMin, temp;

printf("Running Selection sort...\n");

for (int i = 0; i < n-1; i++)

{

indexOfMin = i;

for (int j = i+1; j < n; j++)

{

if(A[j] < A[indexOfMin]){

indexOfMin = j;

}

}

// Swap A[i] and A[indexOfMin]

temp = A[i];

A[i] = A[indexOfMin];

A[indexOfMin] = temp;

}

}

int main(){

int A[] = {3, 5, 2, 13, 12};

int n = 5;

printArray(A, n);

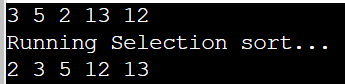
selectionSort(A, n);

printArray(A, n);

return 0;

}

* **OUTPUT:**



**Practical-9**

* **Aim:**

Program to implement Quicksort.

* **Problem:**

based on 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("How many elements are u going to enter?: ");

scanf("%d",&count);

printf("Enter %d elements:\n", count);

for(i=0;i<count;i++)

scanf("%d",&number[i]);

quicksort(number,0,count-1);

printf("Order of Sorted elements: ");

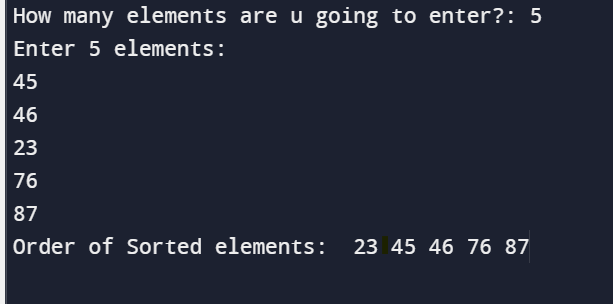
for(i=0;i<count;i++)

printf(" %d",number[i]);

return 0;

}

* **Output:**

****

**Practical:10**

* **Aim:**

Program to implement Merge sort.

* **Problem:**

based on Merge sort.

* **Code:**

#include <stdio.h>

void printArray(int \*A, int n)

{

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

{

printf("%d ", A[i]);

}

printf("\n");

}

void merge(int A[], int mid, int low, int high)

{

int i, j, k, B[100];

i = low;

j = mid + 1;

k = low;

while (i <= mid && j <= high)

{

if (A[i] < A[j])

{

B[k] = A[i];

i++;

k++;

}

else

{

B[k] = A[j];

j++;

k++;

}

}

while (i <= mid)

{

B[k] = A[i];

k++;

i++;

}

while (j <= high)

{

B[k] = A[j];

k++;

j++;

}

for (int i = low; i <= high; i++)

{

A[i] = B[i];

}

}

void mergeSort(int A[], int low, int high){

int mid;

if(low<high){

mid = (low + high) /2;

mergeSort(A, low, mid);

mergeSort(A, mid+1, high);

merge(A, mid, low, high);

}

}

int main()

{

// int A[] = {9, 14, 4, 8, 7, 5, 6};

int A[] = {9, 1, 4, 14, 4, 15, 6};

int n = 7;

printArray(A, n);

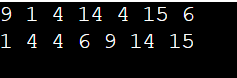
mergeSort(A, 0, 6);

printArray(A, n);

return 0;

}

* **Output:**

****

**Practical-11**

* **Aim:**

Program to implement Heapsort

* **Problem:**

based on Heap sort.

* **Code:**

#include <stdio.h>

// Function to swap the position of two elements

void swap(int\* a, int\* b)

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// To heapify a subtree rooted with node i

// which is an index in arr[].

// n is size of heap

void heapify(int arr[], int N, int i)

{

// Find largest among root, left child and right child

// Initialize largest as root

int largest = i;

// left = 2\*i + 1

int left = 2 \* i + 1;

// right = 2\*i + 2

int right = 2 \* i + 2;

// If left child is larger than root

if (left < N && arr[left] > arr[largest])

largest = left;

// If right child is larger than largest

// so far

if (right < N && arr[right] > arr[largest])

largest = right;

// Swap and continue heapifying if root is not largest

// If largest is not root

if (largest != i) {

swap(&arr[i], &arr[largest]);

// Recursively heapify the affected

// sub-tree

heapify(arr, N, largest);

}

}

// Main function to do heap sort

void heapSort(int arr[], int N)

{

// Build max heap

for (int i = N / 2 - 1; i >= 0; i--)

heapify(arr, N, i);

// Heap sort

for (int i = N - 1; i >= 0; i--) {

swap(&arr[0], &arr[i]);

// Heapify root element to get highest element at

// root again

heapify(arr, i, 0);

}

}

void printArray(int arr[], int N)

{

for (int i = 0; i < N; i++)

printf("%d ", arr[i]);

printf("\n");

}

int main()

{

int arr[] = { 12, 11, 13, 5, 6, 7 };

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

// Function call

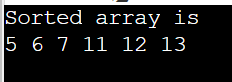
heapSort(arr, N);

printf("Sorted array is\n");

printArray(arr, N);

}

* **Output:**

****

**Practical-12**

* **Aim:**

Write a menu-based program to perform operations for a binary search tree (BST).

* Search an element
* Find minimum
* Find maximum
* Insertion
* Deletion
* **Problem:**

based on binary search tree sort.

* **Code:**

#include<stdio.h>

#include<malloc.h>

struct node{

int data;

struct node\* left;

struct node\* right;

};

struct node\* createNode(int data){

struct node \*n; // creating a node pointer

n = (struct node \*) malloc(sizeof(struct node)); // Allocating memory in the heap

n->data = data; // Setting the data

n->left = NULL; // Setting the left and right children to NULL

n->right = NULL; // Setting the left and right children to NULL

return n; // Finally returning the created node

}

void preOrder(struct node\* root){

if(root!=NULL){

printf("%d ", root->data);

preOrder(root->left);

preOrder(root->right);

}

}

void postOrder(struct node\* root){

if(root!=NULL){

postOrder(root->left);

postOrder(root->right);

printf("%d ", root->data);

}

}

void inOrder(struct node\* root){

if(root!=NULL){

inOrder(root->left);

printf("%d ", root->data);

inOrder(root->right);

}

}

int isBST(struct node\* root){

static struct node \*prev = NULL;

if(root!=NULL){

if(!isBST(root->left)){

return 0;

}

if(prev!=NULL && root->data <= prev->data){

return 0;

}

prev = root;

return isBST(root->right);

}

else{

return 1;

}

}

struct node \* searchIter(struct node\* root, int key){

while(root!=NULL){

if(key == root->data){

return root;

}

else if(key<root->data){

root = root->left;

}

else{

root = root->right;

}

}

return NULL;

}

void insert(struct node \*root, int key){

struct node \*prev = NULL;

while(root!=NULL){

prev = root;

if(key==root->data){

printf("Cannot insert %d, already in BST", key);

return;

}

else if(key<root->data){

root = root->left;

}

else{

root = root->right;

}

}

struct node\* new = createNode(key);

if(key<prev->data){

prev->left = new;

}

else{

prev->right = new;

}

}

struct node\* inOrderPredecessor(struct node \*root){

root = root->left;

while (root->right!=NULL)

{

root = root->right;

}

return root;

}

struct node \*deleteNode(struct node \*root, int value){

struct node\* iPre;

if (root == NULL){

return NULL;

}

if (root->left==NULL&&root->right==NULL){

free(root);

return NULL;

}

//searching for the node to be deleted

if (value < root->data){

root-> left = deleteNode(root->left,value);

}

else if (value > root->data){

root-> right = deleteNode(root->right,value);

}

//deletion strategy when the node is found

else{

iPre = inOrderPredecessor(root);

root->data = iPre->data;

root->left = deleteNode(root->left, iPre->data);

}

return root;

}

int main(){

// Constructing the root node - Using Function (Recommended)

struct node \*p = createNode(5);

struct node \*p1 = createNode(3);

struct node \*p2 = createNode(6);

struct node \*p3 = createNode(1);

struct node \*p4 = createNode(4);

// Finally The tree looks like this:

// 5

// / \

// 3 6

// / \

// 1 4

// Linking the root node with left and right children

p->left = p1;

p->right = p2;

p1->left = p3;

p1->right = p4;

struct node\* n = searchIter(p, 10);

if(n!=NULL){

printf("Found: %d", n->data);

}

else{

printf("Element not found\n");

}

inOrder(p);

printf("\n");

insert(p, 16);

printf("%d", p->right->right->data);

printf("\n");

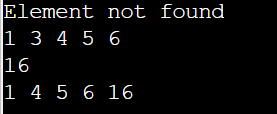
deleteNode(p, 3);

inOrder(p);

return 0;

}

* **Output:**

****