Practical 1

Aim:- To implement stack ADT using array.

Theory:-

Stack is a linear data structure which follows the LIFO (Last In First Out) principle. Elements are inserted at the rear end where the tail pointer points the location of the element to be inserted. Elements are popped from the rear end as well where the tail pointer marks the topmost element.  
For a real time example of a stack, consider a stack of books. One book is on top of the other. For removing a book at a certain position, let’s say the third book from below, we have to take out every single book from the top position till the third book. The book inserted first in the stack will be the last one to be removed. This was a real time example of a stack.

* Operations on stack:-

1. push()
2. pop()
3. peek()
4. isEmpty()
5. isFull()
6. size()

Code:-

// Stack implementation in C

#include <stdio.h>

#include <stdlib.h>

#define MAX 10

int count = 0;

// Creating a stack

struct stack {

int items[MAX];

int top;

};

typedef struct stack st;

void createEmptyStack(st \*s) {

s->top = -1;

}

// Check if the stack is full

int isfull(st \*s) {

if (s->top == MAX - 1)

return 1;

else

return 0;

}

// Check if the stack is empty

int isempty(st \*s) {

if (s->top == -1)

return 1;

else

return 0;

}

// Add elements into stack

void push(st \*s, int newitem) {

if (isfull(s)) {

printf("STACK FULL");

} else {

s->top++;

s->items[s->top] = newitem;

}

count++;

}

// Remove element from stack

void pop(st \*s) {

if (isempty(s)) {

printf("\n STACK EMPTY \n");

} else {

printf("Item popped= %d", s->items[s->top]);

s->top--;

}

count--;

printf("\n");

}

// Print elements of stack

void printStack(st \*s) {

printf("Stack: ");

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

printf("%d ", s->items[i]);

}

printf("\n");

}

// Driver code

int main() {

int ch;

st \*s = (st \*)malloc(sizeof(st));

createEmptyStack(s);

push(s, 1);

push(s, 2);

push(s, 3);

push(s, 4);

printStack(s);

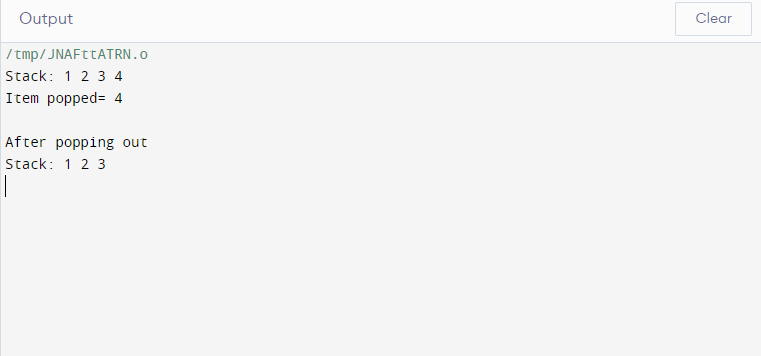
pop(s);

printf("\nAfter popping out\n");

printStack(s);

}

Output:-



Practical 2

Aim:- To convert infix expression to postfix expression using Stack ADT.

Theory:-

Stack is a linear Data Structure which follows the LIFO principle as we saw in Practical 1. Now we use the stack ADT to convert an infix expression to postfix.

* What is an infix expression?

An infix expression is a mathematical expression in which operators are placed between the operands. For example, in the expression “2 + 3”, the “+” operator is placed between the operands “2” and “3”.

* What is a postfix expression?

If we move the operators after the operands then it is known as a postfix expression. In other words, postfix expression can be defined as an expression in which all the operators are present after the operands.

Code:-

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_EXPR\_SIZE 100

// Function to return precedence of operators

int precedence(char operator)

{

switch (operator) {

case '+':

case '-':

return 1;

case '\*':

case '/':

return 2;

case '^':

return 3;

default:

return -1;

}

}

// Function to check if the scanned character

// is an operator

int isOperator(char ch)

{

return (ch == '+' || ch == '-' || ch == '\*' || ch == '/'

|| ch == '^');

}

// Main functio to convert infix expression

// to postfix expression

char\* infixToPostfix(char\* infix)

{

int i, j;

int len = strlen(infix);

char\* postfix = (char\*)malloc(sizeof(char) \* (len + 2));

char stack[MAX\_EXPR\_SIZE];

int top = -1;

for (i = 0, j = 0; i < len; i++) {

if (infix[i] == ' ' || infix[i] == '\t')

continue;

// If the scanned character is operand

// add it to the postfix expression

if (isalnum(infix[i])) {

postfix[j++] = infix[i];

}

// if the scanned character is '('

// push it in the stack

else if (infix[i] == '(') {

stack[++top] = infix[i];

}

// if the scanned character is ')'

// pop the stack and add it to the

// output string until empty or '(' found

else if (infix[i] == ')') {

while (top > -1 && stack[top] != '(')

postfix[j++] = stack[top--];

if (top > -1 && stack[top] != '(')

return "Invalid Expression";

else

top--;

}

// If the scanned character is an operator

// push it in the stack

else if (isOperator(infix[i])) {

while (top > -1

&& precedence(stack[top])

>= precedence(infix[i]))

postfix[j++] = stack[top--];

stack[++top] = infix[i];

}

}

// Pop all remaining elements from the stack

while (top > -1) {

if (stack[top] == '(') {

return "Invalid Expression";

}

postfix[j++] = stack[top--];

}

postfix[j] = '\0';

return postfix;

}

// Driver code

int main()

{

char infix[MAX\_EXPR\_SIZE] = "a+b\*(c^d-e)^(f+g\*h)-i";

// Function call

char\* postfix = infixToPostfix(infix);

printf("%s\n", postfix);

free(postfix);

return 0;

}

Output:-

A black text on a white background

Description automatically generated

Practical 3

Aim:- To evaluate postfix expression using stack ADT

Theory:-

* What is a postfix expression?

If we move the operators after the operands then it is known as a postfix expression. In other words, postfix expression can be defined as an expression in which all the operators are present after the operands.

In this following program, we are going to evaluate a postfix string.

The final output will be an integer and will be printed out to the user.

Code:-

#include <ctype.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Stack type

struct Stack {

int top;

unsigned capacity;

int\* array;

};

// Stack Operations

struct Stack\* createStack(unsigned capacity)

{

struct Stack\* stack

= (struct Stack\*)malloc(sizeof(struct Stack));

if (!stack)

return NULL;

stack->top = -1;

stack->capacity = capacity;

stack->array

= (int\*)malloc(stack->capacity \* sizeof(int));

if (!stack->array)

return NULL;

return stack;

}

int isEmpty(struct Stack\* stack)

{

return stack->top == -1;

}

char peek(struct Stack\* stack)

{

return stack->array[stack->top];

}

char pop(struct Stack\* stack)

{

if (!isEmpty(stack))

return stack->array[stack->top--];

return '$';

}

void push(struct Stack\* stack, char op)

{

stack->array[++stack->top] = op;

}

// The main function that returns value

// of a given postfix expression

int evaluatePostfix(char\* exp)

{

// Create a stack of capacity equal to expression size

struct Stack\* stack = createStack(strlen(exp));

int i;

// See if stack was created successfully

if (!stack)

return -1;

// Scan all characters one by one

for (i = 0; exp[i]; ++i) {

// If the scanned character is an operand

// (number here), push it to the stack.

if (isdigit(exp[i]))

push(stack, exp[i] - '0');

// If the scanned character is an operator,

// pop two elements from stack apply the operator

else {

int val1 = pop(stack);

int val2 = pop(stack);

switch (exp[i]) {

case '+':

push(stack, val2 + val1);

break;

case '-':

push(stack, val2 - val1);

break;

case '\*':

push(stack, val2 \* val1);

break;

case '/':

push(stack, val2 / val1);

break;

}

}

}

return pop(stack);

}

// Driver code

int main()

{

char exp[] = "231\*+9-";

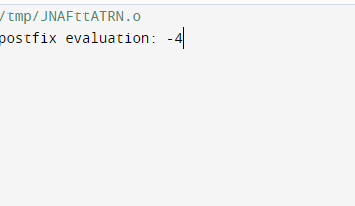
// Function call

printf("postfix evaluation: %d", evaluatePostfix(exp));

return 0;

}

Output:-



Practical 4

Aim:- To implement linear queue ADT using array

Theory:-

A Queue is a linear data structure which follows the FIFO (First In First Out) principle. When an element is inserted into the queue, it is inserted at the rear end of the queue. However unlike stack, when an element is popped, the element at the first position, i.e. the first inserted element is popped.

A queue follows the following operations:-

1. Create()
2. Enqueue()
3. Dequeue()
4. isEmpty()
5. isFull()
6. size()

Code:-

// Queue implementation in C

#include <stdio.h>

#define SIZE 5

void enQueue(int);

void deQueue();

void display();

int items[SIZE], front = -1, rear = -1;

int main() {

//deQueue is not possible on empty queue

deQueue();

//enQueue 5 elements

enQueue(1);

enQueue(2);

enQueue(3);

enQueue(4);

enQueue(5);

// 6th element can't be added to because the queue is full

enQueue(6);

display();

//deQueue removes element entered first i.e. 1

deQueue();

//Now we have just 4 elements

display();

return 0;

}

void enQueue(int value) {

if (rear == SIZE - 1)

printf("\nQueue is Full!!");

else {

if (front == -1)

front = 0;

rear++;

items[rear] = value;

printf("\nInserted -> %d", value);

}

}

void deQueue() {

if (front == -1)

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

else {

printf("\nDeleted : %d", items[front]);

front++;

if (front > rear)

front = rear = -1;

}

}

// Function to print the queue

void display() {

if (rear == -1)

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

else {

int i;

printf("\nQueue elements are:\n");

for (i = front; i <= rear; i++)

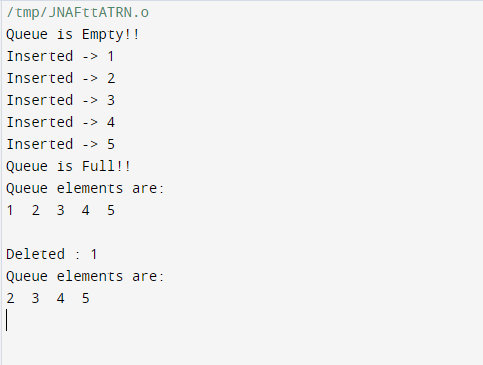
printf("%d ", items[i]);

}

printf("\n");

}

Output:-



Practical 5

Aim:- To implement Circular Queue ADT using Array.

Theory:-

In Practical 4, we saw what is a queue and how it works. A Queue is a linear data structure which follows the FIFO (First In First Out) principle.

In this practical, we will apply and use a circular queue.

The difference between a queue and a circular queue is that a circular queue is an extended version of a normal queue where the last element of the queue is connected to the first element of the queue forming a circle.

We implement this circular queue using arrays.

Code:-

#include <stdio.h>

#define SIZE 5

int items[SIZE];

int front = -1, rear = -1;

// Check if the queue is full

int isFull() {

if ((front == rear + 1) || (front == 0 && rear == SIZE - 1)) return 1;

return 0;

}

// Check if the queue is empty

int isEmpty() {

if (front == -1) return 1;

return 0;

}

// Adding an element

void enQueue(int element) {

if (isFull())

printf("\n Queue is full!! \n");

else {

if (front == -1) front = 0;

rear = (rear + 1) % SIZE;

items[rear] = element;

printf("\n Inserted -> %d", element);

}

}

// Removing an element

int deQueue() {

int element;

if (isEmpty()) {

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

return (-1);

} else {

element = items[front];

if (front == rear) {

front = -1;

rear = -1;

}

// Q has only one element, so we reset the

// queue after dequeing it. ?

else {

front = (front + 1) % SIZE;

}

printf("\n Deleted element -> %d \n", element);

return (element);

}

}

// Display the queue

void display() {

int i;

if (isEmpty())

printf(" \n Empty Queue\n");

else {

printf("\n Front -> %d ", front);

printf("\n Items -> ");

for (i = front; i != rear; i = (i + 1) % SIZE) {

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

}

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

printf("\n Rear -> %d \n", rear);

}

}

int main() {

// Fails because front = -1

deQueue();

enQueue(1);

enQueue(2);

enQueue(3);

enQueue(4);

enQueue(5);

// Fails to enqueue because front == 0 && rear == SIZE - 1

enQueue(6);

display();

deQueue();

display();

enQueue(7);

display();

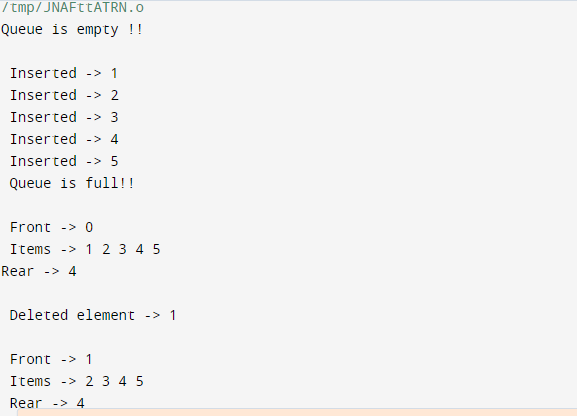
// Fails to enqueue because front == rear + 1

enQueue(8);

return 0;

}

Output:-



Practical 6

Aim: - To implement Singly Linked List ADT

Theory: -

A singly linked list is a type of linked list that is unidirectional, that is, it can be traversed in only one direction from head to the last node (tail). Each element in a linked list is called a node. A single node contains data and a pointer to the next node which helps in maintaining the structure of the list.

Code: -

#include <stdio.h>

#include <stdlib.h>

// Create a node

struct Node {

  int data;

  struct Node\* next;

};

// Insert at the beginning

void insertAtBeginning(struct Node\*\* head\_ref, int new\_data) {

  // Allocate memory to a node

  struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

  // insert the data

  new\_node->data = new\_data;

  new\_node->next = (\*head\_ref);

  // Move head to new node

  (\*head\_ref) = new\_node;

}

// Insert a node after a node

void insertAfter(struct Node\* prev\_node, int new\_data) {

  if (prev\_node == NULL) {

  printf("the given previous node cannot be NULL");

  return;

  }

  struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

  new\_node->data = new\_data;

  new\_node->next = prev\_node->next;

  prev\_node->next = new\_node;

}

// Insert the the end

void insertAtEnd(struct Node\*\* head\_ref, int new\_data) {

  struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

  struct Node\* last = \*head\_ref; /\* used in step 5\*/

  new\_node->data = new\_data;

  new\_node->next = NULL;

  if (\*head\_ref == NULL) {

  \*head\_ref = new\_node;

  return;

  }

  while (last->next != NULL) last = last->next;

  last->next = new\_node;

  return;

}

// Delete a node

void deleteNode(struct Node\*\* head\_ref, int key) {

  struct Node \*temp = \*head\_ref, \*prev;

  if (temp != NULL && temp->data == key) {

  \*head\_ref = temp->next;

  free(temp);

  return;

  }

  // Find the key to be deleted

  while (temp != NULL && temp->data != key) {

  prev = temp;

  temp = temp->next;

  }

  // If the key is not present

  if (temp == NULL) return;

  // Remove the node

  prev->next = temp->next;

  free(temp);

}

// Search a node

int searchNode(struct Node\*\* head\_ref, int key) {

  struct Node\* current = \*head\_ref;

  while (current != NULL) {

  if (current->data == key) return 1;

  current = current->next;

  }

  return 0;

}

// Sort the linked list

void sortLinkedList(struct Node\*\* head\_ref) {

  struct Node \*current = \*head\_ref, \*index = NULL;

  int temp;

  if (head\_ref == NULL) {

  return;

  } else {

  while (current != NULL) {

    // index points to the node next to current

    index = current->next;

    while (index != NULL) {

    if (current->data > index->data) {

      temp = current->data;

      current->data = index->data;

      index->data = temp;

    }

    index = index->next;

    }

    current = current->next;

  }

  }

}

// Print the linked list

void printList(struct Node\* node) {

  while (node != NULL) {

  printf(" %d ", node->data);

  node = node->next;

  }

}

// Driver program

int main() {

  struct Node\* head = NULL;

  insertAtEnd(&head, 1);

  insertAtBeginning(&head, 2);

  insertAtBeginning(&head, 3);

  insertAtEnd(&head, 4);

  insertAfter(head->next, 5);

  printf("Linked list: ");

  printList(head);

  printf("\nAfter deleting an element: ");

  deleteNode(&head, 3);

  printList(head);

  int item\_to\_find = 3;

  if (searchNode(&head, item\_to\_find)) {

  printf("\n%d is found", item\_to\_find);

  } else {

  printf("\n%d is not found", item\_to\_find);

  }

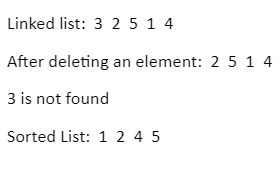
  sortLinkedList(&head);

  printf("\nSorted List: ");

  printList(head);

}

Output: -



Practical 7

Aim: -To implement double linked list

Theory: -

As we saw in the previous example, we implemented a singly linked list which had the following properties:

1. Had a next pointer to point to the next address.
2. Had a data value stored.
3. No reference to the previous node.

Here, in Double Linked List (DLL) we have our original next pointer as well as a previous pointer. This previous pointer will point to the previous node of the given node in the linked list. This way traversal in the tree is easier as each node is linked to both its next AND previous nodes.

Code: -

#include <stdio.h>

#include <stdlib.h>

// node creation

struct Node {

  int data;

  struct Node\* next;

  struct Node\* prev;

};

// insert node at the front

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

  // allocate memory for newNode

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

  // assign data to newNode

  newNode->data = data;

  // make newNode as a head

  newNode->next = (\*head);

  // assign null to prev

  newNode->prev = NULL;

  // previous of head (now head is the second node) is newNode

  if ((\*head) != NULL)

    (\*head)->prev = newNode;

  // head points to newNode

  (\*head) = newNode;

}

// insert a node after a specific node

void insertAfter(struct Node\* prev\_node, int data) {

  // check if previous node is null

  if (prev\_node == NULL) {

    printf("previous node cannot be null");

    return;

  }

  // allocate memory for newNode

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

  // assign data to newNode

  newNode->data = data;

  // set next of newNode to next of prev node

  newNode->next = prev\_node->next;

  // set next of prev node to newNode

  prev\_node->next = newNode;

  // set prev of newNode to the previous node

  newNode->prev = prev\_node;

  // set prev of newNode's next to newNode

  if (newNode->next != NULL)

    newNode->next->prev = newNode;

}

// insert a newNode at the end of the list

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

  // allocate memory for node

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

  // assign data to newNode

  newNode->data = data;

  // assign null to next of newNode

  newNode->next = NULL;

  // store the head node temporarily (for later use)

  struct Node\* temp = \*head;

  // if the linked list is empty, make the newNode as head node

  if (\*head == NULL) {

    newNode->prev = NULL;

    \*head = newNode;

    return;

  }

  // if the linked list is not empty, traverse to the end of the linked list

  while (temp->next != NULL)

    temp = temp->next;

  // now, the last node of the linked list is temp

  // assign next of the last node (temp) to newNode

  temp->next = newNode;

  // assign prev of newNode to temp

  newNode->prev = temp;

}

// delete a node from the doubly linked list

void deleteNode(struct Node\*\* head, struct Node\* del\_node) {

  // if head or del is null, deletion is not possible

  if (\*head == NULL || del\_node == NULL)

    return;

  // if del\_node is the head node, point the head pointer to the next of del\_node

  if (\*head == del\_node)

    \*head = del\_node->next;

  // if del\_node is not at the last node, point the prev of node next to del\_node to the previous of del\_node

  if (del\_node->next != NULL)

    del\_node->next->prev = del\_node->prev;

  // if del\_node is not the first node, point the next of the previous node to the next node of del\_node

  if (del\_node->prev != NULL)

    del\_node->prev->next = del\_node->next;

  // free the memory of del\_node

  free(del\_node);

}

// print the doubly linked list

void displayList(struct Node\* node) {

  struct Node\* last;

  while (node != NULL) {

    printf("%d->", node->data);

    last = node;

    node = node->next;

  }

  if (node == NULL)

    printf("NULL\n");

}

int main() {

  // initialize an empty node

  struct Node\* head = NULL;

  insertEnd(&head, 5);

  insertFront(&head, 1);

  insertFront(&head, 6);

  insertEnd(&head, 9);

  // insert 11 after head

  insertAfter(head, 11);

  // insert 15 after the second node

  insertAfter(head->next, 15);

  displayList(head);

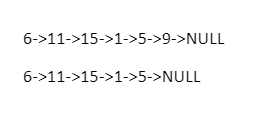
  // delete the last node

  deleteNode(&head, head->next->next->next->next->next);

  displayList(head);

}

Output: -



Practical 8

Aim: -To implement circular linked list

Theory: -

As we saw in the previous example, we implemented a doubly linked list which had the following properties:

1. Had a next pointer to point to the next address.
2. Had a previous pointer to point to the previous address.
3. Had a data value stored.
4. Had an end point of the Linked List.

Here, in Circular Linked List (CLL) we have the last node of the linked list pointing to the first node of the linked this.

This satisfies the name as ‘Circular’ which means that one end is always connected to the beginning. Here the last nodes next pointer will point to the first nodes address.

Code: -

#include <stdio.h>

#include <stdlib.h>

struct Node {

  int data;

  struct Node\* next;

};

struct Node\* addToEmpty(struct Node\* last, int data) {

  if (last != NULL) return last;

  // allocate memory to the new node

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

  // assign data to the new node

  newNode->data = data;

  // assign last to newNode

  last = newNode;

  // create link to iteself

  last->next = last;

  return last;

}

// add node to the front

struct Node\* addFront(struct Node\* last, int data) {

  // check if the list is empty

  if (last == NULL) return addToEmpty(last, data);

  // allocate memory to the new node

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

  // add data to the node

  newNode->data = data;

  // store the address of the current first node in the newNode

  newNode->next = last->next;

  // make newNode as head

  last->next = newNode;

  return last;

}

// add node to the end

struct Node\* addEnd(struct Node\* last, int data) {

  // check if the node is empty

  if (last == NULL) return addToEmpty(last, data);

  // allocate memory to the new node

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

  // add data to the node

  newNode->data = data;

  // store the address of the head node to next of newNode

  newNode->next = last->next;

  // point the current last node to the newNode

  last->next = newNode;

  // make newNode as the last node

  last = newNode;

  return last;

}

// insert node after a specific node

struct Node\* addAfter(struct Node\* last, int data, int item) {

  // check if the list is empty

  if (last == NULL) return NULL;

  struct Node \*newNode, \*p;

  p = last->next;

  do {

  // if the item is found, place newNode after it

  if (p->data == item) {

    // allocate memory to the new node

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

    // add data to the node

    newNode->data = data;

    // make the next of the current node as the next of newNode

    newNode->next = p->next;

    // put newNode to the next of p

    p->next = newNode;

    // if p is the last node, make newNode as the last node

    if (p == last) last = newNode;

    return last;

  }

  p = p->next;

  } while (p != last->next);

  printf("\nThe given node is not present in the list");

  return last;

}

// delete a node

void deleteNode(struct Node\*\* last, int key) {

  // if linked list is empty

  if (\*last == NULL) return;

  // if the list contains only a single node

  if ((\*last)->data == key && (\*last)->next == \*last) {

  free(\*last);

  \*last = NULL;

  return;

  }

  struct Node \*temp = \*last, \*d;

  // if last is to be deleted

  if ((\*last)->data == key) {

  // find the node before the last node

  while (temp->next != \*last) temp = temp->next;

  // point temp node to the next of last i.e. first node

  temp->next = (\*last)->next;

  free(\*last);

  \*last = temp->next;

  }

  // travel to the node to be deleted

  while (temp->next != \*last && temp->next->data != key) {

  temp = temp->next;

  }

  // if node to be deleted was found

  if (temp->next->data == key) {

  d = temp->next;

  temp->next = d->next;

  free(d);

  }

}

void traverse(struct Node\* last) {

  struct Node\* p;

  if (last == NULL) {

  printf("The list is empty");

  return;

  }

  p = last->next;

  do {

  printf("%d ", p->data);

  p = p->next;

  } while (p != last->next);

}

int main() {

  struct Node\* last = NULL;

  last = addToEmpty(last, 6);

  last = addEnd(last, 8);

  last = addFront(last, 2);

  last = addAfter(last, 10, 2);

  traverse(last);

  deleteNode(&last, 8);

  printf("\n");

  traverse(last);

  return 0;

}

Output: -



Practical 9

Aim: -To implement Stack ADT using linked list

Theory: -

So far, we know that Stack is a linear data structure which follows the LIFO (Last-in-first-out) principle.

Linked list is also a linear data structure which has a property which links each and every successive node to each other.

In this experiment, we create a program where we implement the Stack ADT (Abstract Data Type) using a Linked List.

Code: -

// C++ program to Implement a stack

// using singly linked list

#include <bits/stdc++.h>

using namespace std;

// creating a linked list;

class Node {

public:

int data;

Node\* link;

// Constructor

Node(int n)

{

this->data = n;

this->link = NULL;

}

};

class Stack {

Node\* top;

public:

Stack() { top = NULL; }

void push(int data)

{

// Create new node temp and allocate memory in heap

Node\* temp = new Node(data);

// Check if stack (heap) is full.

// Then inserting an element would

// lead to stack overflow

if (!temp) {

cout << "\nStack Overflow";

exit(1);

}

// Initialize data into temp data field

temp->data = data;

// Put top pointer reference into temp link

temp->link = top;

// Make temp as top of Stack

top = temp;

}

// Utility function to check if

// the stack is empty or not

bool isEmpty()

{

// If top is NULL it means that

// there are no elements are in stack

return top == NULL;

}

// Utility function to return top element in a stack

int peek()

{

// If stack is not empty , return the top element

if (!isEmpty())

return top->data;

else

exit(1);

}

// Function to remove

// a key from given queue q

void pop()

{

Node\* temp;

// Check for stack underflow

if (top == NULL) {

cout << "\nStack Underflow" << endl;

exit(1);

}

else {

// Assign top to temp

temp = top;

// Assign second node to top

top = top->link;

// This will automatically destroy

// the link between first node and second node

// Release memory of top node

// i.e delete the node

free(temp);

}

}

// Function to print all the

// elements of the stack

void display()

{

Node\* temp;

// Check for stack underflow

if (top == NULL) {

cout << "\nStack Underflow";

exit(1);

}

else {

temp = top;

while (temp != NULL) {

// Print node data

cout << temp->data;

// Assign temp link to temp

temp = temp->link;

if (temp != NULL)

cout << " -> ";

}

}

}

};

// Driven Program

int main()

{

// Creating a stack

Stack s;

// Push the elements of stack

s.push(11);

s.push(22);

s.push(33);

s.push(44);

// Display stack elements

s.display();

// Print top element of stack

cout << "\nTop element is " << s.peek() << endl;

// Delete top elements of stack

s.pop();

s.pop();

// Display stack elements

s.display();

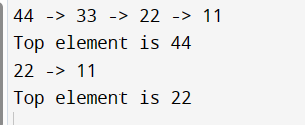
// Print top element of stack

cout << "\nTop element is " << s.peek() << endl;

return 0;

}

Output:-



Practical 10

Aim: -To implement Linear Queue ADT using linked list

Theory: -

So far, we know what the Queue data structure is.

Queue is a linear data structure which follows the FIFO (First-in-first-out) principle.

Linked list is also a linear data structure which has a property which links each and every successive node to each other.

In this experiment, we create a program where we implement the Queue ADT (Abstract Data Type) using a Linked List.

Code: -

// C++ program for the above approach

#include<bits/stdc++.h>;

using namespace std;

struct QNode {

int data;

QNode\* next;

QNode(int d)

{

data = d;

next = NULL;

}

};

struct Queue {

QNode \*front, \*rear;

Queue() { front = rear = NULL; }

void enQueue(int x)

{

// Create a new LL node

QNode\* temp = new QNode(x);

// If queue is empty, then

// new node is front and rear both

if (rear == NULL) {

front = rear = temp;

return;

}

// Add the new node at

// the end of queue and change rear

rear-&gt;next = temp;

rear = temp;

}

// Function to remove

// a key from given queue q

void deQueue()

{

// If queue is empty, return NULL.

if (front == NULL)

return;

// Store previous front and

// move front one node ahead

QNode\* temp = front;

front = front-&gt;next;

// If front becomes NULL, then

// change rear also as NULL

if (front == NULL)

rear = NULL;

delete (temp);

}

};

// Driver code

int main()

{

Queue q;

q.enQueue(10);

q.enQueue(20);

q.deQueue();

q.deQueue();

q.enQueue(30);

q.enQueue(40);

q.enQueue(50);

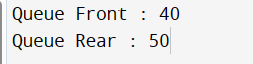
q.deQueue();

cout &lt;&lt; &quot;Queue Front : &quot; &lt;&lt; ((q.front != NULL) ? (q.front)-&gt;data : -1)&lt;&lt; endl;

cout &lt;&lt; &quot;Queue Rear : &quot; &lt;&lt; ((q.rear != NULL) ? (q.rear)-&gt;data : -1);

}

Output: -



Practical 11

Aim: -To implement Binary Search Tree using ADT using linked list

Theory: -

As we know,

Binary Search Tree is a node-based binary tree data structure which has the following properties:

* The left subtree of a node contains only nodes with keys lesser than the node’s key.
* The right subtree of a node contains only nodes with keys greater than the node’s key.
* The left and right subtree each must also be a binary search tree.

Linked list is also a linear data structure which has a property which links each and every successive node to each other.

In this experiment, we create a program where we implement the Binary Search Tree using a Linked List.

Code: -

// Binary Search Tree operations in C

#include <stdio.h>

#include <stdlib.h>

struct node {

int key;

struct node \*left, \*right;

};

// Create a node

struct node \*newNode(int item) {

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

temp->key = item;

temp->left = temp->right = NULL;

return temp;

}

// Inorder Traversal

void inorder(struct node \*root) {

if (root != NULL) {

// Traverse left

inorder(root->left);

// Traverse root

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

// Traverse right

inorder(root->right);

}

}

// Insert a node

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

// Return a new node if the tree is empty

if (node == NULL) return newNode(key);

// Traverse to the right place and insert the node

if (key < node->key)

node->left = insert(node->left, key);

else

node->right = insert(node->right, key);

return node;

}

// Find the inorder successor

struct node \*minValueNode(struct node \*node) {

struct node \*current = node;

// Find the leftmost leaf

while (current && current->left != NULL)

current = current->left;

return current;

}

// Deleting a node

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

// Return if the tree is empty

if (root == NULL) return root;

// Find the node to be deleted

if (key < root->key)

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

else if (key > root->key)

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

else {

// If the node is with only one child or no child

if (root->left == NULL) {

struct node \*temp = root->right;

free(root);

return temp;

} else if (root->right == NULL) {

struct node \*temp = root->left;

free(root);

return temp;

}

// If the node has two children

struct node \*temp = minValueNode(root->right);

// Place the inorder successor in position of the node to be deleted

root->key = temp->key;

// Delete the inorder successor

root->right = deleteNode(root->right, temp->key);

}

return root;

}

// Driver code

int main() {

struct node \*root = NULL;

root = insert(root, 8);

root = insert(root, 3);

root = insert(root, 1);

root = insert(root, 6);

root = insert(root, 7);

root = insert(root, 10);

root = insert(root, 14);

root = insert(root, 4);

printf("Inorder traversal: ");

inorder(root);

printf("\nAfter deleting 10\n");

root = deleteNode(root, 10);

printf("Inorder traversal: ");

inorder(root);

}

Output: -

