Practical 1- Postfix Expression Using Stack

#include <stdio.h>

#include <ctype.h>

#define MAX 100

#define POSTFIX 100

int stack[MAX];

int top = -1;

void push(int item)

{

if (top >= MAX - 1) {

printf("stack over flow");

return;

}

else {

top = top + 1;

stack[top] = item;

}

}

int pop()

{

int item;

if (top < 0)

printf("stack under flow");

else {

item = stack[top];

top = top - 1;

return item;

}

}

void EvalPostfix(char postfix[])

{

int i;

char ch;

int val;

int A, B;

for (i = 0; postfix[i] != ')'; i++) {

ch = postfix[i];

if (isdigit(ch))

push(ch - '0');

else if (ch == '+' || ch == '-' || ch == '\*' || ch == '/') {

A = pop();

B = pop();

switch (ch)

{

case '\*':val = B \* A;

break;

case '/':val = B / A;

break;

case '+':val = B + A;

break;

case '-':val = B - A;

break;

}

push(val);

}

}

printf(" \n Result of expression evaluation : %d \n", pop());

}

int main()

{

int i;

char postfix[POSTFIX];

printf("There are four operators(\*, /, +, -) in an expression with a single digit operand");

printf(" \nEnter postfix expression,\npress right parenthesis ')' for end expression : \n");

for (i = 0; i <= POSTFIX - 1; i++) {

scanf("%c", &postfix[i]);

if (postfix[i] == ')')

{

break;

}

}

EvalPostfix(postfix);

return 0;

}

Practical 2- Infix To Postfix

#include<stdio.h>

#include<stdlib.h>

#include<ctype.h>

#include<string.h>

#define SIZE 100

char stack[SIZE];

int top = -1;

void push(char item)

{

if(top >= SIZE-1)

printf("\nStack Overflow.");

else

{

top = top+1;

stack[top] = item;

}

}

char pop()

{

char item ;

if(top <0)

{

printf("stack under flow: invalid infix expression");

getchar();

exit(1);

}

else

{

item = stack[top];

top = top-1;

return(item);

}

}

int is\_operator(char symbol)

{

if(symbol == '^' || symbol == '\*' || symbol == '/' || symbol == '+' || symbol =='-')

return 1;

else

return 0;

}

int precedence(char symbol)

{

if(symbol == '^')/\* exponent operator, highest precedence\*/

return(3);

else if(symbol == '\*' || symbol == '/')

return(2);

else if(symbol == '+' || symbol == '-') /\* lowest precedence \*/

return(1);

else

return(0);

}

void InfixToPostfix(char infix\_exp[], char postfix\_exp[])

{

int i, j;

char item;

char x;

push('(');

strcat(infix\_exp,")");

i=0;

j=0;

item=infix\_exp[i];

while(item != '\0')

{

if(item == '(')

{

push(item);

}

else if( isdigit(item) || isalpha(item))

{

postfix\_exp[j] = item;

j++;

}

else if(is\_operator(item) == 1)

{

x=pop();

while(is\_operator(x) == 1 && precedence(x)>= precedence(item))

{

postfix\_exp[j] = x;

j++;

x = pop();

}

push(x);

push(item);

}

else if(item == ')')

{

x = pop();

while(x != '(')

{

postfix\_exp[j] = x;

j++;

x = pop();

}

}

else

{ /\* if current symbol is neither operand not '(' nor ')' and nor

operator \*/

printf("\nInvalid infix Expression.\n");

getchar();

exit(1);

}

i++;

item = infix\_exp[i];

}

if(top>0)

{

printf("\nInvalid infix Expression.\n");

getchar();

exit(1);

}

if(top>0)

{

printf("\nInvalid infix Expression.\n");

getchar();

exit(1);

}

postfix\_exp[j] = '\0';

}

int main()

{

char infix[SIZE], postfix[SIZE];

printf("The infix expression should contain single letter variables and single digit constants");

printf("\nEnter Infix expression : ");

InfixToPostfix(infix,postfix);

printf("Postfix Expression: ");

puts(postfix);

return 0;

Practical 3- double ended queue (dequeue) using arrays.

# include<stdio.h>

# define MAX 5

int deque\_arr[MAX];

int left = -1;

int right = -1;

void insert\_right()

{

int added\_item;

if((left == 0 && right == MAX-1) || (left == right+1))

{ printf("Queue Overflow\n");

return;}

if (left == -1) /\* if queue is initially empty \*/

{ left = 0;

right = 0;}

else

if(right == MAX-1) /\*right is at last position of queue \*/

right = 0;

else

right = right+1;

printf("Input the element for adding in queue : ");

scanf("%d", &added\_item);

deque\_arr[right] = added\_item ;

}

void insert\_left()

{ int added\_item;

if((left == 0 && right == MAX-1) || (left == right+1))

{ printf("Queue Overflow \n");

return; }

if (left == -1)/\*If queue is initially empty\*/

{ left = 0;

right = 0; }

else

if(left== 0)

left=MAX-1;

else

left=left-1;

printf("Input the element for adding in queue : ");

scanf("%d", &added\_item);

deque\_arr[left] = added\_item ; }

void delete\_left()

{ if (left == -1)

{ printf("Queue Underflow\n");

return ; }

printf("Element deleted from queue is : %d\n",deque\_arr[left]);

if(left == right) /\*Queue has only one element \*/

{ left = -1;

right=-1; }

else

if(left == MAX-1)

left = 0;

else

left = left+1;

}

void delete\_right()

{if (left == -1)

{printf("Queue Underflow\n");

return ; }

printf("Element deleted from queue is : %d\n",deque\_arr[right]);

if(left == right) /\*queue has only one element\*/

{ left = -1;

right=-1; }

else

if(right == 0)

right=MAX-1;

else

right=right-1; }

void display\_queue()

{ int front\_pos = left,rear\_pos = right;

if(left == -1)

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

return; }

printf("Queue elements :\n");

if( front\_pos <= rear\_pos )

{ while(front\_pos <= rear\_pos)

{ printf("%d ",deque\_arr[front\_pos]);

front\_pos++; } }

else

{ while(front\_pos <= MAX-1)

{ printf("%d ",deque\_arr[front\_pos]);

front\_pos++; }

front\_pos = 0;

while(front\_pos <= rear\_pos)

{ printf("%d ",deque\_arr[front\_pos]);

front\_pos++;

}

}

printf("\n");

}

void input\_que()

{ int choice;

do

{ printf("1.Insert at right\n");

printf("2.Delete from left\n");

printf("3.Delete from right\n");

printf("4.Display\n");

printf("5.Quit\n");

printf("Enter your choice : ");

scanf("%d",&choice);

switch(choice)

{ case 1:insert\_right();

break;

case 2:delete\_left();

break;

case 3:delete\_right();

break;

case 4:display\_queue();

break;

case 5:break;

default:printf("Wrong choice\n");

}

}while(choice!=5);

}

void output\_que()

{ int choice;

do

{ printf("1.Insert at right\n");

printf("2.Insert at left\n");

printf("3.Delete from left\n");

printf("4.Display\n");

printf("5.Quit\n");

printf("Enter your choice : ");

scanf("%d",&choice);

switch(choice)

{

case 1:insert\_right();

break;

case 2:insert\_left();

break;

case 3:delete\_left();

break;

case 4:display\_queue();

break;

case 5:break;

default:printf("Wrong choice\n");

}

}while(choice!=5);

}

int main()

{ int choice;

printf("1.Input restricted dequeue\n");

printf("2.Output restricted dequeue\n");

printf("Enter your choice : ");

scanf("%d",&choice);

switch(choice)

{

case 1 :input\_que();

break;

case 2:output\_que();

break;

default:printf("Wrong choice\n");

}

Practical 4- Stack using two queues

#include <stdio.h>

#define MAX 3

int q1[MAX], q2[MAX];

int front1 = -1, rear1 = -1;

int front2 = -1, rear2 = -1;

int isEmpty(int front, int rear) {

return (front == -1);

}

void enqueue(int q[], int \*front, int \*rear, int value) {

if (\*rear == MAX - 1) {

printf("Stack Overflow\n");

return;

}

if (\*front == -1) {

\*front = 0;

}

(\*rear)++;

q[\*rear] = value;

}

int dequeue(int q[], int \*front, int \*rear) {

if (isEmpty(\*front, \*rear)) {

printf("Stack Underflow\n");

return -1;

}

int value = q[\*front];

if (\*front == \*rear) {

\*front = \*rear = -1;

} else {

(\*front)++;

}

return value;

}

void push(int x) {

enqueue(q1, &front1, &rear1, x);

}

void pop() {

if (isEmpty(front1, rear1)) {

printf("Stack Underflow\n");

return;

}

while (front1 != rear1) {

enqueue(q2, &front2, &rear2, dequeue(q1, &front1, &rear1));

}

int poppedElement = dequeue(q1, &front1, &rear1);

printf("Popped element: %d\n", poppedElement);

for (int i = 0; i <= rear2; i++) {

q1[i] = q2[i];

}

front1 = front2;

rear1 = rear2;

front2 = rear2 = -1;

}

void display() {

if (isEmpty(front1, rear1)) {

printf("Stack is empty\n");

return;

}

printf("Stack elements: ");

for (int i = front1; i <= rear1; i++) {

printf("%d ", q1[i]);

}

printf("\n");

}

int main() {

int choice, value;

while (1) {

printf("\nStack Operations:\n");

printf("1. Push\n");

printf("2. Pop\n");

printf("3. Display\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to push: ");

scanf("%d", &value);

push(value);

break;

case 2:

pop();

break;

case 3:

display();

break;

case 4:

printf("Exiting...\n");

return 0;

default:

printf("Invalid choice! Please try again.\n");

}

}

return 0;

**Practical 5- Queue using two Stacks**

#include<stdio.h>

#define N 5

int stack1[N], stack2[N];

int top1=-1, top2=-1;

int count=0;

void push1(int data)

{

if(top1 == N-1)

{

printf("\nStack is overflow...");

}

else

{

top1++;

stack1[top1] = data;

}

}

int pop1()

{

if(top1 == -1)

{

printf("\nStack is empty..");

return -1;

}

else

{

int a = stack1[top1];

top1--;

return a;

}

}

void push2(int x)

{

if(top2 == N-1)

{

printf("\nStack is full..");

}

else

{

top2++;

stack2[top2] = x;

}

}

int pop2()

{

if(top2 == -1)

{

printf("\nStack2 is empty..");

return -1;

}

else

{

int element = stack2[top2];

top2--;

return element;

}

}

void enqueue(int x)

{

push1(x);

count++;

}

void dequeue()

{

if((top1 == -1) && (top2 == -1))

{

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

}

else

{

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

{

int element = pop1();

push2(element);

}

int b = pop2();

printf("\nThe dequeued element is %d\n", b);

count--;

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

{

int a = pop2();

push1(a);

}

}

}

void display()

{

if(top1 == -1)

{

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

}

else

{

printf("\nQueue elements: ");

for(int i = 0; i <= top1; i++)

{

printf("%d, ", stack1[i]);

}

printf("\n");

}

}

void main()

{

int choice, value;

while(1)

{

printf("\n\nQueue using Stacks");

printf("\n1. Enqueue");

printf("\n2. Dequeue");

printf("\n3. Display");

printf("\n4. Exit");

printf("\nEnter your choice: ");

scanf("%d", &choice);

switch(choice)

{

case 1:

if(top1 == N-1)

{

printf("\nQueue is full!");

}

else

{

printf("\nEnter the value to enqueue: ");

scanf("%d", &value);

enqueue(value);

}

break;

case 2:

dequeue();

break;

case 3:

display();

break;

case 4:

printf("\nExiting");

return;

default:

printf("\nInvalid choice, please try again.");

}

}

}

Practical 6-Single Linked List

#include <stdio.h>

#include <stdlib.h>

struct Node { // Create a node

int data;

struct Node\* next;

};

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

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

new\_node->data = new\_data;

new\_node->next = (\*head\_ref);

(\*head\_ref) = new\_node;

}

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

if (prev\_node == NULL) {

printf("The given previous node cannot be NULL\n");

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;

}

void insertAtEnd(struct Node\*\* head\_ref, int new\_data) { // Insert at the end

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

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

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;

}

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;

}

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

prev = temp;

temp = temp->next;

}

if (temp == NULL) return;

prev->next = temp->next;

free(temp);

}

int searchNode(struct Node\*\* head\_ref, int key) { // Search a node

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

while (current != NULL) {

if (current->data == key)

return 1;

current = current->next;

}

return 0;

}

void sortLinkedList(struct Node\*\* head\_ref) { // Sort the linked list

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

int temp;

if (\*head\_ref == NULL) {

return;

} else {

while (current != NULL) {

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;

}

}

}

void printList(struct Node\* node) { // Print the linked list

while (node != NULL) {

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

node = node->next;

}

}

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);

return 0;

}

Practical 7- Stack using Linked List

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

void push(struct Node\*\* top, int value) {

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

newNode->data = value;

newNode->next = \*top;

\*top = newNode;

printf("Pushed %d\n", value);

}

void pop(struct Node\*\* top) {

if (\*top == NULL) {

printf("Stack Underflow!\n");

return;

}

struct Node\* temp = \*top;

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

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

free(temp);

}

void display(struct Node\* top) {

if (!top) {

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

return;

}

printf("Stack: ");

while (top) {

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

top = top->next;

}

printf("\n");

}

int main() {

struct Node\* top = NULL;

push(&top, 10);

push(&top, 20);

push(&top, 30);

display(top);

pop(&top);

display(top);

return 0;

}

Practical 8-BST

#include <stdio.h>

#include <stdlib.h>

struct Node {

int key;

struct Node\* left;

struct Node\* right;

};

struct Node\* createNode(int key) {

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

newNode->key = key;

newNode->left = newNode->right = NULL;

return newNode;

}

struct Node\* insert(struct Node\* root, int key) {

if (root == NULL) return createNode(key);

if (key < root->key) root->left = insert(root->left, key);

else if (key > root->key) root->right = insert(root->right, key);

return root;

}

int findMin(struct Node\* root) {

while (root->left) root = root->left;

return root->key;

}

int findMax(struct Node\* root) {

while (root->right) root = root->right;

return root->key;

}

int search(struct Node\* root, int key) {

if (root == NULL) return 0;

if (root->key == key) return 1;

if (key < root->key) return search(root->left, key);

return search(root->right, key);

}

int main() {

struct Node\* root = NULL;

int keys[] = {50, 30, 70, 20, 40, 60, 80};

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

for (int i = 0; i < n; i++) root = insert(root, keys[i]);

printf("Minimum key: %d\n", findMin(root));

printf("Maximum key: %d\n", findMax(root));

int searchKey = 40;

if (search(root, searchKey))

printf("Key %d is found in the BST.\n", searchKey);

else

printf("Key %d is not found in the BST.\n", searchKey);

return 0;

}

Peactical 9- HASHING

#include <stdio.h>

#include <stdlib.h>

#define SIZE 10

struct Node {

int key;

struct Node\* next;

};

struct Node\* hashTable[SIZE] = {NULL};

int hash(int key) {

return key % SIZE;

}

void insert(int key) {

int index = hash(key);

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

newNode->key = key;

newNode->next = hashTable[index];

hashTable[index] = newNode;

}

int search(int key) {

int index = hash(key);

struct Node\* temp = hashTable[index];

while (temp) {

if (temp->key == key) return 1;

temp = temp->next;

}

return 0;

}

void display() {

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

printf("%d: ", i);

struct Node\* temp = hashTable[i];

while (temp) {

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

temp = temp->next;

}

printf("NULL\n");

}

}

int main() {

insert(10);

insert(20);

insert(15);

insert(25);

display();

printf("Search 15: %s\n", search(15) ? "Found" : "Not Found");

return 0;

}

**Practical 10-SORTING**

**Insertion Sort**

#include <stdio.h>

void insertionSort(int arr[], int n) {

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

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

void display(int arr[], int n) {

for (int i = 0; i < n; i++) printf("%d ", arr[i]);

printf("\n");

}

int main() {

int arr[] = {5, 2, 9, 1, 5, 6};

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

insertionSort(arr, n);

display(arr, n);

return 0;

}

**Merge Sort:**

#include <stdio.h>

void merge(int arr[], int left, int mid, int right) {

int n1 = mid - left + 1, n2 = right - mid;

int L[n1], R[n2];

for (int i = 0; i < n1; i++) L[i] = arr[left + i];

for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];

while (i < n1) arr[k++] = L[i++];

while (j < n2) arr[k++] = R[j++];

}

void mergeSort(int arr[], int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

void display(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]);

mergeSort(arr, 0, n - 1);

display(arr, n);

return 0;

}

**Quick Sort:**

#include <stdio.h>

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = low - 1;

for (int j = low; j < high; j++) {

if (arr[j] < pivot) {

i++;

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return i + 1;

}

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

void display(int arr[], int n) {

for (int i = 0; i < n; i++) printf("%d ", arr[i]);

printf("\n");

}

int main() {

int arr[] = {10, 7, 8, 9, 1, 5};

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

quickSort(arr, 0, n - 1);

display(arr, n);

return 0;

}