1)program to copy queue into another queue when the queue is iimplemented as linked list.

#include<stdio.h>

#include<stdlib.h>

struct queue

{

int data;

struct queue \*link;

};

typedef struct queue \*QUEUE;

QUEUE createnode();

QUEUE enqueue(QUEUE top);

QUEUE dequeue(QUEUE top);

QUEUE copy(QUEUE top,QUEUE another);

void display(QUEUE top);

QUEUE enqueue2(QUEUE cur,QUEUE another);

int main()

{

QUEUE top,another=NULL;

top=NULL;

int m;

while(1)

{

printf("1:Enqueue 2:Dequeue 3:display 4:copy 5:Displayed copied 6:Exit\n");

printf("Enter your choice\n");

scanf("%d",&m);

if(m==1)

{

top=enqueue(top);

}

else if(m==2)

{

top=dequeue(top);

}

else if(m==3)

{

display(top);

}

else if(m==4)

{

another=copy(top,another);

}

else if(m==5)

{

display(another);

another=NULL;

}

else if(m==6)

{

exit(0);

}

else

{

printf("invalid choice\n");

}

}

}

QUEUE createnode()

{

QUEUE newnode;

newnode=malloc(sizeof(struct queue));

printf("Enter the data that you want to enqueue\n");

scanf("%d",&newnode->data);

newnode->link=NULL;

return newnode;

}

QUEUE enqueue(QUEUE top)

{

QUEUE newnode,cur;

newnode=createnode();

if(top==NULL)

{

top=newnode;

top->link=NULL;

}

else

{

cur=top;

while(cur->link!=NULL)

{

cur=cur->link;

}

cur->link=newnode;

newnode->link=NULL;

}

return top;

}

QUEUE dequeue(QUEUE top)

{

QUEUE cur;

if(top==NULL)

{

printf("The queue is empty\n");

}

else if(top->link==NULL)

{

free(top);

top=NULL;

}

else

{

cur=top->link;

free(top);

top=cur;

}

return top;

}

QUEUE enqueue2(QUEUE cur,QUEUE another)

{

QUEUE newnode,tp;

newnode=malloc(sizeof(struct queue));

newnode->data=cur->data;

if(another==NULL)

{

another=newnode;

another->link=NULL;

}

else

{

tp=another;

while(tp->link!=NULL)

{

tp=tp->link;

}

tp->link=newnode;

newnode->link=NULL;

}

return another;

}

QUEUE copy(QUEUE top,QUEUE another)

{

QUEUE cur;

cur=top;

if(top==NULL)

{

printf("The queue is empty\n");

}

else

{

cur=top;

while(cur!=NULL)

{

another=enqueue2(cur,another);

cur=cur->link;

}

}

return another;

}

void display(QUEUE top)

{

QUEUE cur;

if(top==NULL)

{

printf("THe queue is empty");

}

else

{

cur=top;

while(cur!=NULL)

{

printf("\n");

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

printf(" %xf",cur->link);

cur=cur->link;

}

printf("\n");

}

}

2)write a program to represert a dequeue using a linked list also write functions to read and delete from the deque.

#include<stdio.h>

#include<stdlib.h>

struct queue

{

int data;

struct queue \*link;

};

typedef struct queue \*QUEUE;

QUEUE createnode();

QUEUE enqueue(QUEUE top);

QUEUE dequeue(QUEUE top);

QUEUE dequeue2(QUEUE top);

QUEUE copy(QUEUE top,QUEUE another);

void display(QUEUE top);

QUEUE enqueue2(QUEUE top);

int main()

{

QUEUE top;

top=NULL;

int m;

while(1)

{

printf("1:Enqueue from top 2:enqueue from bottom 3:Remove from bottom 4:remove from top 5:display 6:Exit\n");

printf("Enter your choice\n");

scanf("%d",&m);

if(m==1)

{

top=enqueue(top);

}

else if(m==2)

{

top=enqueue2(top);

}

else if(m==3)

{

top=dequeue(top);

}

else if(m==4)

{

top=dequeue2(top);

}

else if(m==5)

{

display(top);

}

else if(m==6)

{

exit(0);

}

else

{

printf("invalid choice\n");

}

}

}

QUEUE createnode()

{

QUEUE newnode;

newnode=malloc(sizeof(struct queue));

printf("Enter the data that you want to enqueue\n");

scanf("%d",&newnode->data);

newnode->link=NULL;

return newnode;

}

QUEUE enqueue(QUEUE top)

{

QUEUE newnode,cur;

newnode=createnode();

if(top==NULL)

{

top=newnode;

top->link=NULL;

}

else

{

cur=top;

while(cur->link!=NULL)

{

cur=cur->link;

}

cur->link=newnode;

newnode->link=NULL;

}

return top;

}

QUEUE enqueue2(QUEUE top)

{

QUEUE cur,newnode;

newnode=createnode();

if(top==NULL)

{

top=newnode;

top->link=NULL;

}

else

{

newnode->link=top;

top=newnode;

}

return top;

}

QUEUE dequeue(QUEUE top)

{

QUEUE cur;

if(top==NULL)

{

printf("The queue is empty\n");

}

else if(top->link==NULL)

{

free(top);

top=NULL;

}

else

{

cur=top->link;

free(top);

top=cur;

}

return top;

}

QUEUE dequeue2(QUEUE top)

{

QUEUE cur,prev;

if(top==NULL)

{

printf("The queue is empty\n");

}

else if(top->link==NULL)

{

free(top);

top=NULL;

}

else

{

cur=top;

prev=cur;

while(cur->link!=NULL)

{

prev=cur;

cur=cur->link;

}

prev->link=NULL;

free(cur);

cur=prev;

}

return top;

}

void display(QUEUE top)

{

QUEUE cur;

if(top==NULL)

{

printf("THe queue is empty");

}

else

{

cur=top;

while(cur!=NULL)

{

printf("\n");

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

printf("^\n");

cur=cur->link;

}

printf("\n");

}

}

Graphs\_dfs\_bfs

#include <stdio.h>

#include <stdlib.h>

#define MAX\_VERTICES 100

// Stack data structure

struct Stack {

int items[MAX\_VERTICES];

int top;

};

// Queue data structure

struct Queue {

int items[MAX\_VERTICES];

int front;

int rear;

};

// Graph data structure

struct Graph {

int numVertices;

int\*\* adjMatrix;

};

// Function prototypes

struct Stack\* createStack();

int isEmptyStack(struct Stack\* stack);

void push(struct Stack\* stack, int value);

int pop(struct Stack\* stack);

struct Queue\* createQueue();

int isEmptyQueue(struct Queue\* queue);

void enqueue(struct Queue\* queue, int value);

int dequeue(struct Queue\* queue);

struct Graph\* createGraph(int numVertices);

void addEdge(struct Graph\* graph, int src, int dest);

void dfs(struct Graph\* graph, int startVertex);

void bfs(struct Graph\* graph, int startVertex);

// Create an empty stack

struct Stack\* createStack() {

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

stack->top = -1;

return stack;

}

// Check if the stack is empty

int isEmptyStack(struct Stack\* stack) {

return stack->top == -1;

}

// Push an item onto the stack

void push(struct Stack\* stack, int value) {

stack->items[++stack->top] = value;

}

// Pop an item from the stack

int pop(struct Stack\* stack) {

if (isEmptyStack(stack)) {

printf("Stack is empty");

return -1;

} else {

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

}

}

// Create an empty queue

struct Queue\* createQueue() {

struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

queue->front = -1;

queue->rear = -1;

return queue;

}

// Check if the queue is empty

int isEmptyQueue(struct Queue\* queue) {

return queue->rear == -1;

}

// Add an item to the queue

void enqueue(struct Queue\* queue, int value) {

if (queue->rear == MAX\_VERTICES - 1) {

printf("Queue is full");

} else {

if (queue->front == -1) {

queue->front = 0;

}

queue->rear++;

queue->items[queue->rear] = value;

}

}

// Remove an item from the queue

int dequeue(struct Queue\* queue) {

int item;

if (isEmptyQueue(queue)) {

printf("Queue is empty");

item = -1;

} else {

item = queue->items[queue->front];

queue->front++;

if (queue->front > queue->rear) {

queue->front = -1;

queue->rear = -1;

}

}

return item;

}

// Create a graph with a given number of vertices

struct Graph\* createGraph(int numVertices) {

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->numVertices = numVertices;

graph->adjMatrix = (int\*)malloc(numVertices \* sizeof(int));

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

graph->adjMatrix[i] = (int\*)malloc(numVertices \* sizeof(int));

for (int j = 0; j < numVertices; j++) {

graph->adjMatrix[i][j] = 0;

}

}

return graph;

}

// Add an edge between two vertices

void addEdge(struct Graph\* graph, int src, int dest) {

graph->adjMatrix[src][dest] = 1;

graph->adjMatrix[dest][src] = 1;

}

// Depth-First Search (DFS)

void dfs(struct Graph\* graph, int startVertex) {

struct Stack\* stack = createStack();

int visited[MAX\_VERTICES] = {0};

push(stack, startVertex);

visited[startVertex] = 1;

while (!isEmptyStack(stack)) {

int currentVertex = pop(stack);

printf("%d ", currentVertex);

for (int i = 0; i < graph->numVertices; i++) {

if (graph->adjMatrix[currentVertex][i] == 1 && visited[i] == 0) {

push(stack, i);

visited[i] = 1;

}

}

}

}

// Breadth-First Search (BFS)

void bfs(struct Graph\* graph, int startVertex) {

struct Queue\* queue = createQueue();

int visited[MAX\_VERTICES] = {0};

visited[startVertex] = 1;

printf("%d ", startVertex);

enqueue(queue, startVertex);

while (!isEmptyQueue(queue)) {

int currentVertex = dequeue(queue);

for (int i = 0; i < graph->numVertices; i++) {

if (graph->adjMatrix[currentVertex][i] == 1 && visited[i] == 0) {

visited[i] = 1;

printf("%d ", i);

enqueue(queue, i);

}

}

}

}

int main() {

int vertices = 5; // Number of vertices

int edges = 2; // Number of edges

struct Graph\* graph = createGraph(vertices);

// Adding edges

addEdge(graph, 2, 3);

printf("DFS traversal: ");

dfs(graph, 0);

printf("\n");

printf("BFS traversal: ");

bfs(graph, 0);

printf("\n");

    return 0;

}

Array\_reversal

#include<stdio.h>

#include<stdlib.h>

void reverse\_array(int A[100],int n)

{

int i,temp[100],c=0;

printf("Enter array elements\n");

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

{

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

}

for(i=n-1;i>=0;i--)

{

temp[c]=A[i];

c++;

}

printf("Array elements \n");

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

{

A[i]=temp[i];

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

}

}

int main()

{

int A[100],n;

printf("Enter the size of array\n");

scanf("%d",&n);

reverse\_array(A,n);

return 0;

}

Array repeates

#include <stdio.h>

int main() {

int rows, cols;

printf("Enter the number of rows: ");

scanf("%d", &rows);

printf("Enter the number of columns: ");

scanf("%d", &cols);

if (rows != cols) {

printf("Rows and columns are not equal.\n");

return 0;

}

int matrix[rows][cols];

printf("Enter the elements of the matrix:\n");

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

for (int j = 0; j < cols; j++) {

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

}

}

int isSymmetric = 1;

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

for (int j = 0; j < cols; j++) {

if (matrix[i][j] != matrix[j][i]) {

isSymmetric = 0;

break;

}

}

if (!isSymmetric) {

break;

}

}

if (isSymmetric) {

printf("The matrix is symmetric.\n");

} else {

printf("The matrix is not symmetric.\n");

}

return 0;

}

Avl tree

#include <stdio.h>

#include <stdlib.h>

struct AVLNode {

int data;

struct AVLNode\* left;

struct AVLNode\* right;

int height;

};

int max(int a, int b) {

return (a > b) ? a : b;

}

int getHeight(struct AVLNode\* node) {

if (node == NULL)

return 0;

return node->height;

}

struct AVLNode\* createNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

newNode->height = 1;

return newNode;

}

struct AVLNode\* rotateRight(struct AVLNode\* y, FILE\* output) {

struct AVLNode\* x = y->left;

struct AVLNode\* T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

fprintf(output, "Performing RR Rotation at node %d\n", y->data);

fprintf(output, "Old Root: %d\n", y->data);

fprintf(output, "New Root after RR Rotation: %d\n", x->data);

return x;

}

struct AVLNode\* rotateLeft(struct AVLNode\* x, FILE\* output) {

struct AVLNode\* y = x->right;

struct AVLNode\* T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

fprintf(output, "Performing LL Rotation at node %d\n", x->data);

fprintf(output, "Old Root: %d\n", x->data);

fprintf(output, "New Root after LL Rotation: %d\n", y->data);

return y;

}

int getBalanceFactor(struct AVLNode\* node) {

if (node == NULL)

return 0;

return getHeight(node->left) - getHeight(node->right);

}

struct AVLNode\* insertNode(struct AVLNode\* node, int data, FILE\* output) {

if (node == NULL)

return createNode(data);

if (data < node->data)

node->left = insertNode(node->left, data, output);

else if (data > node->data)

node->right = insertNode(node->right, data, output);

else

return node; // Duplicate keys are not allowed in AVL tree

node->height = 1 + max(getHeight(node->left), getHeight(node->right));

int balanceFactor = getBalanceFactor(node);

if (balanceFactor > 1 && data < node->left->data)

return rotateRight(node, output);

if (balanceFactor < -1 && data > node->right->data)

return rotateLeft(node, output);

if (balanceFactor > 1 && data > node->left->data) {

node->left = rotateLeft(node->left, output);

return rotateRight(node, output);

}

if (balanceFactor < -1 && data < node->right->data) {

node->right = rotateRight(node->right, output);

return rotateLeft(node, output);

}

return node;

}

struct AVLNode\* findMinValueNode(struct AVLNode\* node) {

struct AVLNode\* current = node;

while (current->left != NULL)

current = current->left;

return current;

}

struct AVLNode\* deleteNode(struct AVLNode\* root, int data, FILE\* output) {

if (root == NULL)

return root;

if (data < root->data)

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

else if (data > root->data)

root->right = deleteNode(root->right, data, output);

else {

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

struct AVLNode\* temp = root->left ? root->left : root->right;

if (temp == NULL) {

temp = root;

root = NULL;

} else

\*root = \*temp;

free(temp);

} else {

struct AVLNode\* temp = findMinValueNode(root->right);

root->data = temp->data;

root->right = deleteNode(root->right, temp->data, output);

}

}

if (root == NULL)

return root;

root->height = 1 + max(getHeight(root->left), getHeight(root->right));

int balanceFactor = getBalanceFactor(root);

if (balanceFactor > 1 && getBalanceFactor(root->left) >= 0)

return rotateRight(root, output);

if (balanceFactor > 1 && getBalanceFactor(root->left) < 0) {

root->left = rotateLeft(root->left, output);

return rotateRight(root, output);

}

if (balanceFactor < -1 && getBalanceFactor(root->right) <= 0)

return rotateLeft(root, output);

if (balanceFactor < -1 && getBalanceFactor(root->right) > 0) {

root->right = rotateRight(root->right, output);

return rotateLeft(root, output);

}

return root;

}

void inorderTraversal(struct AVLNode\* root, FILE\* output) {

if (root != NULL) {

inorderTraversal(root->left, output);

fprintf(output, "%d ", root->data);

inorderTraversal(root->right, output);

}

}

void preorderTraversal(struct AVLNode\* root, FILE\* output) {

if (root != NULL) {

fprintf(output, "%d ", root->data);

preorderTraversal(root->left, output);

preorderTraversal(root->right, output);

}

}

void postorderTraversal(struct AVLNode\* root, FILE\* output) {

if (root != NULL) {

postorderTraversal(root->left, output);

postorderTraversal(root->right, output);

fprintf(output, "%d ", root->data);

}

}

struct AVLNode\* searchAndDelete(struct AVLNode\* root, int data, FILE\* output) {

if (root == NULL) {

fprintf(output, "%d not found in the AVL tree!\n", data);

return root;

}

if (data < root->data)

root->left = searchAndDelete(root->left, data, output);

else if (data > root->data)

root->right = searchAndDelete(root->right, data, output);

else {

fprintf(output, "%d found and deleted from the AVL tree!\n", data);

root = deleteNode(root, data, output);

}

return root;

}

int getNodeDepth(struct AVLNode\* root, int data, int depth) {

if (root == NULL)

return 0;

if (data < root->data)

return getNodeDepth(root->left, data, depth + 1);

else if (data > root->data)

return getNodeDepth(root->right, data, depth + 1);

else

return depth;

}

int main() {

int minRange, maxRange;

printf("Enter the minimum range for random number generation: ");

scanf("%d", &minRange);

printf("Enter the maximum range for random number generation: ");

scanf("%d", &maxRange);

FILE\* inputFile = fopen("input.txt", "w");

if (inputFile == NULL) {

printf("Error opening input file!\n");

return 1;

}

int i, num;

for (i = 0; i < 4; i++) {

num = minRange + rand() % (maxRange - minRange + 1);

fprintf(inputFile, "%d ", num);

}

fclose(inputFile);

FILE\* inputFileRead = fopen("input.txt", "r");

if (inputFileRead == NULL) {

printf("Error opening input file!\n");

return 1;

}

struct AVLNode\* root = NULL;

while (fscanf(inputFileRead, "%d", &num) != EOF) {

root = insertNode(root, num, NULL);

}

fclose(inputFileRead);

FILE\* outputFile = fopen("avl\_eval\_final.txt", "w");

if (outputFile == NULL) {

printf("Error opening output file!\n");

return 1;

}

printf("Generated Random Numbers: ");

inputFileRead = fopen("input.txt", "r");

while (fscanf(inputFileRead, "%d", &num) != EOF) {

printf("%d ", num);

}

printf("\n\n");

fclose(inputFileRead);

// Perform AVL tree operations and write the output to the file

fprintf(outputFile, "Inorder Traversal: ");

inorderTraversal(root, outputFile);

fprintf(outputFile, "\n");

fprintf(outputFile, "Preorder Traversal: ");

preorderTraversal(root, outputFile);

fprintf(outputFile, "\n");

fprintf(outputFile, "Postorder Traversal: ");

postorderTraversal(root, outputFile);

fprintf(outputFile, "\n");

int searchData;

printf("Enter a number to search and delete from the AVL tree: ");

scanf("%d", &searchData);

struct AVLNode\* oldRoot = root;

root = searchAndDelete(root, searchData, outputFile);

// Print height, depth, balancing factor, old root, and alternate solutions

int height = getHeight(root);

int depth = getNodeDepth(root, root->data, 1);

int balanceFactor = getBalanceFactor(root);

fprintf(outputFile, "\nHeight of the AVL tree: %d\n", height);

fprintf(outputFile, "Depth of the root node: %d\n", depth);

fprintf(outputFile, "Balancing Factor of the root node: %d\n", balanceFactor);

fprintf(outputFile, "Old Root: %d\n", oldRoot->data);

fprintf(outputFile, "Number after RR rotation: %d\n", oldRoot->right ? oldRoot->right->data : -1);

fprintf(outputFile, "Number after RL rotation: %d\n", oldRoot->left ? oldRoot->left->data : -1);

fprintf(outputFile, "Number after LL rotation: %d\n", root->left ? root->left->data : -1);

fprintf(outputFile, "Number after LR rotation: %d\n", root->right ? root->right->data : -1);

fclose(outputFile);

printf("AVL tree operations completed successfully. Check 'avl\_eval.txt' for the output.\n");

return 0;

}

Doubly linked list

#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node \*rightlink;

struct node \*leftlink;

};

struct node \*head,\*temp,\*temp1,\*newnode;

struct node \* insert\_at\_beg(struct node \*);

struct node \* insert\_at\_end(struct node \*);

struct node \* delete\_at\_beg(struct node \*);

struct node \* delete\_at\_end(struct node \*);

void display(struct node \*);

int count\_nodes(struct node \*);

struct node\* insert\_at\_pos(struct node \*newnode,int pos);

struct node\* delete\_at\_pos(struct node \*newnode,int pos);

void search\_node(struct node \*, int);

main()

{

head=NULL;

int ch,c,pos,d,count,key;

while(1)

{

printf("1-insert at begining\n");

printf("2--insert at end\n");

printf("3--delete at begining\n");

printf("4--delete at end\n");

printf("5--display\n");

printf("6--exit\n");

printf("7--insert a node at a given pos\n");

printf("8--delete a node at a given pos\n");

printf("9--count the nodes\n");

printf("10--search for a node\n");

printf("enter the choice\n");

scanf("%d",&ch);

switch(ch)

{

case 1: newnode=(struct node \*)malloc(sizeof(struct node));

printf("enter the data\n");

scanf("%d",&newnode->data);

head=insert\_at\_beg(newnode);

break;

case 2: newnode=(struct node \*)malloc(sizeof(struct node));

printf("enter the data\n");

scanf("%d",&newnode->data);

head=insert\_at\_end(newnode);

break;

case 3: head=delete\_at\_beg(head);

break;

case 4:head=delete\_at\_end(head);

break;

case 5: display(head);

break;

case 6:exit(0);break;

default:printf("Invalid choice\n");break;

case 7: newnode=(struct node \*)malloc(sizeof(struct node));

printf("enter the data\n");

scanf("%d",&newnode->data);

printf("enter the position\n");

scanf("%d",&pos);

count=count\_nodes(head);

if(pos<=count+1)

head=insert\_at\_pos(newnode,pos);

else

printf("Invalid position to insert\n");

break;

case 8: printf("enter the position\n");

scanf("%d",&pos);

count=count\_nodes(head);

if(pos<=count)

head=delete\_at\_pos(head,pos);

else

printf("Invalid position to insert\n");

break;

case 9: count=count\_nodes(head);

printf("number of nodes=%d\n",count);

break;

case 10: if(head==NULL)

printf("DLL is empty----cant search\n");

else

{

printf("enter the data to search\n");

scanf("%d",&key);

search\_node(head,key);

}

break;

}

}

}

struct node\* insert\_at\_beg(struct node \*newnode)

{

if(head==NULL)

{

head=newnode;

newnode->rightlink=NULL;

newnode->leftlink=NULL;

}

else

{

newnode->leftlink=NULL;

newnode->rightlink=head;

head=newnode;

}

return(head);

}

struct node\* insert\_at\_end(struct node \*newnode)

{

if(head==NULL)

{

head=newnode;

newnode->rightlink=NULL;

newnode->leftlink=NULL;

}

else

{

temp=head;

while(temp->rightlink!=NULL)

{

temp=temp->rightlink;

}

temp->rightlink=newnode;

newnode->leftlink=temp;

newnode->rightlink=NULL;

}

return(head);

}

struct node\* delete\_at\_beg(struct node \*head)

{

if(head==NULL)

printf("DLL is empty--cant delete\n");

else

{

if(head->rightlink==NULL)

{

printf("node deleted=%d\n",head->data);

free(head);

head=NULL;

}

else

{

temp=head;

printf("node deleted=%d\n",temp->data);

head=head->rightlink;

head->leftlink=NULL;

free(temp);

}

}

return(head);

}

struct node\* delete\_at\_end(struct node \*head)

{

if(head==NULL)

printf("DLL is empty--cant delete\n");

else

{

if(head->rightlink==NULL)

{

printf("node deleted=%d\n",head->data);

free(head);

head=NULL;

}

else

{

temp=head;

while(temp->rightlink!=NULL)

{

temp1=temp;

temp=temp->rightlink;

}

printf("node deleted=%d\n",temp->data);

free(temp);

temp1->rightlink=NULL;

}

}

return(head);

}

void display(struct node \*head)

{

if(head==NULL)

{

printf("DLL is empty\n");

}

else

{

temp=head;

while(temp!=NULL)

{

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

temp=temp->rightlink;

}

}

}

int count\_nodes(struct node \*head)

{

int c=0;

if(head==NULL)

return(0);

else

{

temp=head;

while(temp!=NULL)

{

c++;

temp=temp->rightlink;

}

return(c);

}

}

//count=count\_nodes(head);

struct node\* insert\_at\_pos(struct node \*newnode,int pos)

{

int n=count\_nodes(head);

if(pos==1)

head=insert\_at\_beg(newnode);

else if(pos==count\_nodes(head)+1)

head=insert\_at\_end(newnode);

else

{

temp=head;

for(int i=0;i<pos-2;i++)

{

temp=temp->rightlink;

}

newnode->rightlink=temp->rightlink;

newnode->leftlink=temp;

temp->rightlink=newnode;

(newnode->rightlink)->leftlink=newnode;

}

return(head);

}

struct node\* delete\_at\_pos(struct node\* head,int pos)

{

temp=head;

int n=count\_nodes(n);

if(pos==1)

head=delete\_at\_beg(head);

else if(pos==count\_nodes(head))

head=delete\_at\_end(head);

else

{

for(int i=0;i<pos-2;i++)

{

temp=temp->rightlink;

}

temp=head;

temp1=temp->leftlink;

struct node \*temp2=temp->rightlink;

printf("node deleted=%d\n",temp->data);

temp1->rightlink=temp2;

temp2->leftlink=temp1;

free(temp);

}

return(head);

}

void search\_node(struct node\* head, int key)

{

int status=0;

temp=head;

while(temp->rightlink!=NULL)

{

if(temp->data==key)

{

printf("node found\n");

status=1;

break;

}

temp=temp->rightlink;

}

if(status==0)

printf("node not found in DLL\n");

}

Array multiplication

#include<stdio.h>

#include<stdlib.h>

void product(int a[10][10],int b[10][10],int c[10][10],int m,int n,int p)

{

int i,j,k;

int sum;

printf("enter array elements\n");

printf("please enter array elements\n");

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

{

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

{

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

}

}

printf("Enter array elements\n");

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

{

for(j=0;j<p;j++)

{

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

}

}

printf("Array elements are\n");

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

{

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

{

printf("%d\t",a[i][j]);

}

printf("\n");

}

printf("Array elements are\n");

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

{

for(j=0;j<p;j++)

{

printf("%d\t",b[i][j]);

}

printf("\n");

}

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

{

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

{

sum=0;

for(k=0;k<p;k++)

{

sum=sum+a[i][k]\*b[k][j];

c[i][k]=sum;

}

}

}

printf("product of two arrays are :\n");

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

{

for(j=0;j<p;j++)

{

printf("%d\t",c[i][j]);

}

printf("\n");

}

}

int main()

{

int a[10][10],b[10][10],c[10][10],m,n,p;

printf("enter m,nand p values\n");

scanf("%d%d%d",&m,&n,&p);

product(a,b,c,m,n,p);

return 0;

}

Queue

#include<stdio.h>

struct queue

{

int array[10];

int rear;

int front;

};

void enqueue(struct queue \*aptr,int data);

int dequeue(struct queue \*aptr);

void display(struct queue \*aptr);

int isfull(struct queue \*aptr);

int isempty(struct queue \*aptr);

int main()

{

int k,data;

struct queue a,\*aptr;

aptr=&a;

aptr->front=-1;

aptr->rear=-1;

while(1)

{

printf("1:enqueue 2:dequeue 3:display\n");

printf("Enter your choice\n");

scanf("%d",&k);

if(k==1)

{

printf("Enter the data to be enqueed\n");

scanf("%d",&data);

enqueue(aptr,data);

}

else if(k==2)

{

data=dequeue(aptr);

printf("The dequeed data is %d",data);

}

else if(k==3)

{

display(aptr);

}

else

{

printf("Invalid choice\n");

}

}

}

int isfull(struct queue \*aptr)

{

if(aptr->rear==9)

{

return 1;

}

else

{

return 0;

}

}

int isempty(struct queue \*aptr)

{

if(aptr->rear==aptr->front)

{

if(aptr->array[aptr->front]==0)

{

return 1;

}

else

{

return 0;

}

}

else

{

return 0;

}

}

void enqueue(struct queue \*aptr,int data)

{

int k;

k=isfull(aptr);

if(k==1)

{

printf("Queue overflow\n");

}

else

{

aptr->rear++;

aptr->array[aptr->rear]=data;

}

}

int dequeue(struct queue \*aptr)

{

int k,m;

k=isempty(aptr);

if(k==1)

{

printf("QUeue underflow\n");

}

else

{

m=aptr->array[aptr->front];

aptr->front++;

}

return m;

}

void display(struct queue \*aptr)

{

int k;

k=isempty(aptr);

if(k==1)

{

printf("Queue is empty\n");

}

else

{

for(int i=aptr->front+1;i<=aptr->rear;i++)

{

printf("%d ",aptr->array[i]);

}

}

}

Random numbers

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<time.h>

#define size 10000

void print\_array(int a[size],int n)

{

int i;

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

{

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

}

}

int main()

{

int a[size],n,rnum;

float duration;

clock\_t start,end;

int p,q,i;

FILE \*fp;

fp=fopen("input.txt","w");

printf("Enter the interval\n");

scanf("%d%d",&p,&q);

printf("Enter number of random numbers to be generated\n ");

scanf("%d",&n);

//read\_array(a,n);

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

{

a[i]=rand()%(q+1-p)+p;

}

print\_array(a,n);

if (fp == NULL)

{

printf("Error in opening file\n");

exit(0);

}

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

{

fprintf(fp,"%d",a[1]);

}

fclose(fp);

return 0;

}

Search\_replace

#include<stdio.h>

#include<Stdlib.h>

#include<string.h>

int Length(char \*strings)

{

int i;

for(i=0;strings[i]!='\0';++i);

return i;

}

void read\_linesFromFile(char lines[10][100], int \*num\_lines)

{

FILE \*file = fopen("input.txt","r");

if (file == NULL)

{

printf("Could not open file input.txt");

return;

}

char line[100];

while (\*num\_lines < 10 && fgets(line, 100, file) != NULL)

{

//fprintf(file,"%s ",\*num\_lines);

strncpy(lines[\*num\_lines], line, 100);

(\*num\_lines)++;

}

fclose(file);

}

void search(char \*pat,char \*txt)

{

int n=Length(txt);

int m=Length(pat);

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

{

int j;

for( j=0;j<m;j++)

{

if(txt[i+j]!=pat[j])

break;

}

if(j==m)

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

}

}

int main()

{

char lines[10][100];

int num\_lines = 0;

char pat[10];//="ould";

printf("Reading from the file\n");

read\_linesFromFile(lines, &num\_lines);

printf("Enter the word you want to search in the file:");

printf("%s",pat);

scanf("%s",pat);

for(int i=0;i<10;i++)

{

search(pat,lines[i]);

}

printf("Enter the word you want to replace:");

scanf("%s",pat);

//write\_linesFromFile(lines, &num\_lines);

}

Sort\_numbers

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<time.h>

#define size 10000

void print\_array(int a[size],int n)

{

int i;

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

{

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

}

}

void bubblesort(int a[size],int n)

{

int i,j,temp;

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

{

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

{

if(a[j]>a[j+1])

{

temp=a[j];

a[j]=a[j+1];

a[j+1]=temp;

}

}

}

printf("After sorting...\n");

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

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

}

void selectionsort(int a[size],int n)

{

int indexofmin,temp,j;

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

{

indexofmin=i;

{

if(a[j]<a[indexofmin])

{

indexofmin=j;

}

}

temp=a[i];

a[i]=a[indexofmin];

a[indexofmin]=temp;

}

}

void insertionsort(int a[size],int n)

{

int key,i,j;

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

{

j=i-1;

key=a[i];

while(j>=0 && a[j]<key )

{

a[j+1]=a[j];

j--;

}

a[j+1]=key;

}

}

int main()

{

int a[size],n,rnum;

float duration;

clock\_t start,end;

int p,q,i;

printf("Enter the interval\n");

scanf("%d%d",&p,&q);

printf("Enter number of random numbers to be generated\n ");

scanf("%d",&n);

//read\_array(a,n);

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

{

a[i]=rand()%(q+1-p)+p;

}

FILE \*fp;

fp=fopen("input.txt","w");

if(fp==NULL)

{

printf("File not present\n");

}

else

{

for(i>p;i<q;i++)

{

int k=0;

fprintf(fp,"%d",a[k]);

k++;

}

}

start=clock();

bubblesort(a,n);

end=clock();

duration=(float)(end-start);//(CLOCKS\_PER\_SEC);

printf("\nduration is for bubble sort %f\n",duration);

start=clock();

selectionsort(a,n);

end=clock();

duration=(float)(end-start);//(CLOCKS\_PER\_SEC);

printf("\nduration is for selection sort %f\n",duration);

insertionsort(a,n);

end=clock();

duration=(float)(end-start);//(CLOCKS\_PER\_SEC);

printf("\nduration is for insertion sort %f\n",duration);

print\_array(a,n);

fclose(fp);

}

Sunjgly linked list

#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node \*link;

};

typedef struct node \*NODE;

NODE create\_node();

int length(NODE head);

NODE insert\_front(NODE head);

NODE insert\_end(NODE head);

void display(NODE head);

NODE insert\_position(NODE head);

NODE delete\_end(NODE head);

NODE delete\_front(NODE head);

NODE delete\_position(NODE head);

int main()

{

NODE head=NULL;

int n;

while(1)

{

printf("1:insert front 2:insert end 3:insert position 4:delete end 5:delete front 6:delete position 7:display\n");

printf("Enter the choice \n");

scanf("%d",&n);

if(n==1)

{

head=insert\_front(head);

}

else if(n==2)

{

head=insert\_end(head);

}

else if(n==3)

{

head=insert\_position(head);

}

else if(n==4)

{

head=delete\_end(head);

}

else if(n==5)

{

head=delete\_front(head);

}

else if(n==6)

{

head=delete\_position(head);

}

else if(n==8)

{

display(head);

}

else

{

printf("Invalid choice\n");

exit(0);

}

}

}

NODE create\_node()

{

NODE newnode;

newnode=malloc(sizeof(struct node));

printf("Enter the data\n");

scanf("%d",&newnode->data);

newnode->link=NULL;

return newnode;

}

NODE insert\_front(NODE head)

{

NODE cur,newnode;

newnode=create\_node();

if(head==NULL)

{

head=newnode;

head->link=NULL;

}

else

{

newnode->link=head;

head=newnode;

}

return head;

}

NODE insert\_end(NODE head)

{

NODE cur,newnode;

newnode=create\_node();

if(head==NULL)

{

head=newnode;

head->link=NULL;

}

else

{

cur=head;

while(cur->link!=NULL)

{

cur=cur->link;

}

cur->link=newnode;

newnode->link=NULL;

}

return head;

}

NODE insert\_position(NODE head)

{

NODE newnode,prev,cur;

int k,m,count=0;

m=length(head);

printf("Enter the position you want to add the node\n");

scanf("%d",&k);

if(k==1)

{

head=insert\_front(head);

}

else if(k==m)

{

head=insert\_end(head);

}

else

{

cur=head;

prev=head;

newnode=create\_node();

while(cur!=NULL)

{

prev=cur;

cur=cur->link;

count++;

if(count==k)

{

prev->link=newnode;

newnode->link=cur;

cur=newnode->link;

}

}

}

return head;

}

NODE delete\_end(NODE head)

{

NODE cur=head,prev=head;

if(head==NULL)

{

printf("THe linked list is empty\n");

}

else

{

cur=head;

while(cur->link!=NULL)

{

prev=cur;

cur=cur->link;

}

prev->link=NULL;

free(cur);

cur=prev->link;

}

return head;

}

NODE delete\_front(NODE head)

{

NODE cur;

if(head==NULL)

{

printf("The linked list is empty\n");

}

else

{

cur=head;

head=head->link;

free(cur);

cur=head;

}

return head;

}

NODE delete\_position(NODE head)

{

NODE cur,prev;

int k,l;

l=length(head);

printf("ENter the position that you want to delete\n");

scanf("%d",&k);

int count=0;

if(k==1)

{

head=delete\_front(head);

}

else if(k==l)

{

head=delete\_end(head);

}

else

{

prev=head;;

cur=head;

while(cur!=NULL)

{

prev=cur;

cur=cur->link;

count++;

if(count==k-1)

{

prev->link=cur->link;

free(cur);

cur=prev->link;

}

}

}

return head;

}

int length(NODE head)

{

int m=0;

NODE cur;

if(head==NULL)

{

printf("The list is empty\n");

}

else

{

cur=head;

while(cur!=NULL)

{

m++;

cur=cur->link;

}

}

return m;

}

void display(NODE head)

{

NODE cur;

if(head==NULL)

{

printf("The linked list is empty\n");

}

else

{

cur=head;

while(cur!=NULL)

{

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

cur=cur->link;

}

printf("\n");

}

}

Stack

#include<stdio.h>

#include<stdlib.h>

#define size 3

struct stack

{

int data[size];

int top;

};

void push(struct stack \*sptr,int element);

int pop(struct stack \*sptr);

int peek(struct stack \*sptr);

void display(struct stack \*sptr);

int main()

{

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

sptr->top=-1;

int ch,element;

while(1)

{

printf("stack menu\n1)push 2)pop 3)peek 4)display 5)exit\n");

scanf("%d",&ch);

switch(ch)

{

case 1:printf("enter the data\n");

scanf("%d",&element);

push(sptr,element);

break;

case 2:element=pop(sptr);

if(element==-1)

printf("stack underflow");

else

printf("popped data is %d",element);

break;

case 3:element=peek(sptr);

printf("the top data is %d",element);

break;

case 4:display(sptr);

break;

case 5:exit(0);

}

}

}

void push(struct stack \*sptr,int element)

{

sptr->top++;

if(sptr->top==size)

{

printf("stack overflow\n");

}

else{

sptr->data[sptr->top]=element;

}

}

int pop(struct stack \*sptr)

{

if(sptr->top==-1)

{

return -1;

}

else{

sptr->top--;

return sptr->data[sptr->top];

}

}

int peek(struct stack \*sptr)

{

int element;

if(sptr->top==-1)

{

printf("stack is empty\n");

return -1;

}

else{

return sptr->data[sptr->top];

}

}

void display(struct stack \*sptr)

{

int i;

for(i=sptr->top;i>=0;i--)

{

printf("%d\n",sptr->data[i]);

}

}

Tree traversals

#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

int level;

struct node \*left;

struct node \*right;

};

typedef struct node \*NODE;

NODE create\_node(int key);

NODE insertion(NODE root,int key);

void level\_order(NODE root);

int max(int a,int b);

int height(NODE root);

NODE postorder(NODE root);

void printlevel\_order(NODE root , int i);

int main()

{

NODE root=NULL;

int m,key;

while(1)

{

printf("1:Insertion 2:Level order display 3:exit\n");

scanf("%d",&m);

if(m==1)

{

printf("Enter the data \n");

scanf("%d",&key);

root=insertion(root,key);

}

else if(m==2)

{

level\_order(root);

}

else if(m==3)

{

exit(0);

}

else

{

printf("Invalid choice\n");

}

}

}

NODE create\_node(int key)

{

NODE newnode=(struct node\*)malloc(sizeof(struct node));

newnode->data=key;

newnode->right=newnode->left=NULL;

newnode->level=1;

return newnode;

}

NODE insertion(NODE root,int key)

{

NODE newnode;

if(root==NULL)

{

newnode=create\_node(key);

return newnode;

}

else

{

if(key>=root->data)

{

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

}

else

{

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

}

}

root->level=1+max(height(root->right),height(root->left));

return root;

}

int max(int a,int b)

{

if(a>b)

return a;

else

return b;

}

int height(NODE root)

{

if(root==NULL)

return 0;

else

return root->level;

}

void level\_order(NODE root)

{

int ht;

ht=height(root);

for(int i=1;i<=ht;i++)

{

printlevel\_order(root,i);

}

}

void printlevel\_order(NODE root , int i)

{

NODE k;

k=root;

int m;

m=i;

if(k==NULL)

{

return;

}

else if(i==1)

{

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

if(k->level==root->level)

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

else

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

}

else

{

printlevel\_order(root->left,i-1);

printlevel\_order(k->right,m-1);

}

}

NODE postorder(NODE root)

{

if(root!=NULL)

{

return postorder(root->left);

return postorder(root->right);

}

}