**Termwork 1**

// TW 1 Simpler- without copying files into arrays of strings

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main()

{

FILE \*f1, \*f2, \*f3;

char m[15], n[15], comm[15][15];

int i, cnt=0;

f1=fopen("input1.txt", "r");

if(f1==NULL)

{

printf("\nInput1 file open error for reading");

exit(1);

}

f2=fopen("input2.txt", "r");

if(f2==NULL)

{

printf("\nInput2 file open error for reading");

exit(2);

}

f3=fopen("output", "w");

if(f3==NULL)

{

printf("\nOutput file open error for writing");

exit(3);

}

fscanf(f1,"%s",m);

fscanf(f2,"%s",n);

while(!feof(f1) && !feof(f2))

{

if(strcmp(m,n)==0)

{

fprintf(f3, "%s\n", m);

strcpy(comm[cnt++], m);

fscanf(f1,"%s",m);

fscanf(f2,"%s",n);

}

else

if(strcmp(m,n)<0)

{

fprintf(f3, "%s\n", m);

fscanf(f1,"%s",m);

}

else

{

fprintf(f3, "%s\n", n);

fscanf(f2,"%s",n);

}

}

if(strcmp(m,n)==0)

{

fprintf(f3, "%s\n", m);

strcpy(comm[cnt++], m);

fscanf(f1,"%s",m);

fscanf(f2,"%s",n);

}

if(!feof(f1))

fprintf(f3,"%s\n",m);

if(!feof(f2))

fprintf(f3,"%s\n",n);

while(!feof(f1))

{

fscanf(f1,"%s",m);

fprintf(f3, "%s\n", m);

if(strcmp(m,n)==0)

strcpy(comm[cnt++], m);

}

while(!feof(f2))

{

fscanf(f2,"%s",n);

fprintf(f3, "%s\n", n);

if(strcmp(m,n)==0)

strcpy(comm[cnt++], m);

}

printf("\nThe common USNs are:\n");

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

printf("%s\n", comm[i]);

return 0;

}

**Termwork 2**

//TW 2 Conversion of infix exp to postfix exp

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#define MAX 20

char stack[MAX];

int top=-1;

void push(char x)

{

if(top==MAX-1)

{

printf("\nStack Overflow");

return;

}

stack[++top]=x;

}

char pop()

{

if(top==-1)

return -1;

else

return stack[top--];

}

int priority(char x)

{

if(x=='(')

return 0;

if(x=='+' || x== '-')

return 1;

if(x=='\*' || x== '/')

return 2;

}

void valid(char infix[40])

{

int i;

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

{

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

push('(');

else

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

pop();

}

if (top==-1)

{

printf("expression is valid\n");

return;

}

else

{

printf("Invalid expression");

exit(1);

}

}

int main()

{

char exp[20], \*e, x;

printf("\nEnter the exp: ");

scanf("%s", exp);

valid(exp);

e=exp;

while(\*e!='\0')

{

if(isalnum(\*e))

printf("%c", \*e);

else

if(\*e=='(')

push(\*e);

else

if(\*e==')')

{

while((x=pop())!='(')

printf("%c", x);

}

else

{

while(priority(stack[top])>=priority(\*e))

printf("%c", pop());

push(\*e);

}

e++;

}

while(top!=-1)

printf("%c", pop());

return 0;

}

**Termwork 3**

//TW 3 Evaluation of Postfix Exp

#include <stdio.h>

#include <stdlib.h>

#define MAX 4

int stack[MAX], top;

void push(int item);

int pop();

int postfixeval(char []);

int main()

{

char exp[20];

top=-1;

printf("\nEnter a valid postfix expression: ");

gets(exp);

printf("\nEvaluated result is: %d", postfixeval(exp));

return 0;

}

void push(int item)

{

if(top==MAX-1)

{

printf("\nStack Overflow");

return;

}

stack[++top]=item;

}

int pop()

{

if(top==-1)

{

printf("\nStack Underflow");

return(-1);

}

return(stack[top--]);

}

int postfixeval(char exp[])

{

int i, opd1, opd2;

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

{

if(exp[i]>='0' && exp[i]<='9')

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

else

{

opd2 = pop();

opd1 = pop();

switch(exp[i])

{

case '+': push(opd1+opd2); break;

case '-': push(opd1-opd2); break;

case '\*': push(opd1\*opd2); break;

case '/': if(opd2==0)

{

printf("\nDivide by zero error");

exit(1);

}

push(opd1/opd2);

}

}

}

return(pop());

}

**Termwork 4**

//TW4 Message Queue

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct node

{

char \*msg;

struct node \*next;

} NODE;

//Prototypes

void Sender(NODE \*\*, NODE \*\*, char \*);

char \* Receiver(NODE \*\*, NODE \*\*);

void DispQ(NODE \*);

//Driver

int main()

{

NODE \*front, \*rear;

int opt;

char \*msg;

msg=(char \*)malloc(50);

if(msg==NULL)

{

printf("\nMalloc failure");

exit(4);

}

//Create an empty Queue

front=rear=NULL;

while(1)

{

printf("\n1: Send a msg\t2: Receive a msg\t3: Disp\t4: Exit\n");

printf("Enter your option: ");

scanf("%d", &opt);

switch(opt)

{

case 1: printf("\nEnter the msg: ");

scanf("%s", msg);

Sender(&front, &rear, msg);

break;

case 2: strcpy(msg,Receiver(&front, &rear));

if(strcmp(msg, "Queue underflow")==0)

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

else

printf("\nReceived msg is: %s", msg);

break;

case 3: DispQ(front); break;

case 4: exit(0);

}

}

return 0;

}

void Sender(NODE \*\*front, NODE \*\*rear, char \* item)

{

NODE \*tmp=(NODE \*)malloc(sizeof(NODE));

if(tmp==NULL)

{

printf("\nMemory allocation failure");

exit(1);

}

tmp->msg=(char \*)malloc(sizeof(item));

if(tmp->msg==NULL)

{

printf("\nMalloc failure");

exit(2);

}

strcpy(tmp->msg, item);

tmp->next=NULL;

// Empty Queue

if(\*rear==NULL)

\*front=\*rear=tmp;

else

{

(\*rear)->next=tmp;

\*rear=tmp;

}

}

char \* Receiver(NODE \*\*front, NODE \*\*rear)

{

char \* item;

NODE \*tmp=\*front;

if(\*front==NULL)

{

\*rear=NULL;

return ("Queue underflow");

}

item=(char \*)malloc(sizeof((\*front)->msg));

if(item==NULL)

{

printf("\nMalloc failure");

exit(3);

}

strcpy(item, (\*front)->msg);

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

if(\*front==NULL) // Q has exactly one Datanode

rear=NULL;

free(tmp);

return item;

}

void DispQ(NODE \*front)

{

NODE \*tmp=front;

if(tmp==NULL)

{

printf("\nEmpty Queue");

return;

}

printf("\nThe Queue contents are:\n");

while(tmp)

{

printf("%s\t", tmp->msg);

tmp=tmp->next;

}

}

**Termwork 5**

// TW 5 Hashtable

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#define SIZE 10

typedef struct

{

int custid;

char custname[30];

int custphno;

int empty;

}CUST;

int HashFn(int key)

{

return (key % SIZE);

}

int Search(int key, CUST custs[])

{

int count, pos;

pos = HashFn(key);

for(count = 1; count <= SIZE ; count++)

{

if(custs[pos].empty == 1)

return -1;

if(custs[pos].custid == key)

return pos;

pos = (pos + 1) % SIZE;

}

return -1;

}

void InsHT\_LP(CUST cust, CUST custs[])

{

int count, pos, flag=0;

int key = cust.custid;

pos = HashFn(key);

for(count = 1; count <= SIZE; count++)

{

if(custs[pos].empty == 1)

{

if(flag==1)

printf("\nCollision resolved, new pos = %d", pos);

custs[pos].custid = cust.custid;

strcpy(custs[pos].custname,cust.custname);

custs[pos].custphno = cust.custphno;

custs[pos].empty = -1;

printf("\nRecord Inserted into Hash Table\n");

return;

}

if(custs[pos].empty==-1)

{

printf("\nCollision Detected\n");

flag=1;

}

pos = (pos + 1) % SIZE;

}

printf("\nHash Table is full\n");

}

void Display(CUST custs[])

{

int count;

printf("\nHash Table");

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

{

printf("\n[%d]:\t", count);

if(custs[count].empty == -1)

printf("\nCustomer - ID: %d Name: %s Phone: %d",custs[count].custid, custs[count].custname, custs[count].custphno);

else

printf("No Hash Entry\n");

}

}

int main()

{

int count, key, option;

CUST custs[SIZE], cust;

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

custs[count].empty=1;

while(1)

{

printf("\n\n1. Insert a Record\n");

printf("2. Search a Record\n");

printf("3. Display All Records\n");

printf("4. Exit\n");

printf("\n Enter Your Option:");

scanf("%d", &option);

switch(option)

{

case 1: printf("\nEnter Customer id, name, ph:");

scanf("%d%s%d", &cust.custid,cust.custname,&cust.custphno);

InsHT\_LP(cust, custs);

break;

case 2: printf("\nEnter the Key to Search:");

scanf("%d", &key);

count = Search(key, custs);

if(count == -1)

printf("\nRecord Not Found\n");

else

printf("\nRecord Found at Index pos:%d\n", count);

break;

case 3: Display(custs);

break;

case 4: exit(1);

}

}

return 0;

}

**Termwork 6**

// TW 6 Warehouse as an Ordered List

#include <stdio.h>

#include <stdlib.h>

typedef struct node

{

int data;

struct node \*next;

}NODE;

NODE\* add(NODE\*, int);

void disp(NODE\*);

int search(NODE\*, int);

int main()

{

//Create an empty warehouse

NODE \*head=NULL;

int opt, item;

while(1)

{

printf("\n1: Add item 2: Disp Warehouse ");

printf("3: Search item 4: exit");

printf("\nEnter your option: ");

scanf("%d", &opt);

switch(opt)

{

case 1: printf("\nEnter item to add to warehouse: ");

scanf("%d", &item);

head=add(head, item);

break;

case 2: disp(head); break;

case 3: printf("\nEnter the item to search: ");

scanf("%d", &item);

if(search(head, item))

printf("\nItem %d is present in the warehouse", item);

else

printf("\nItem %d is NOT present in the warehouse", item);

break;

case 4: exit(0);

}

}

return 0;

}

NODE\* add(NODE\* head, int item)

{

NODE \*prev, \*curr;

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

if(newnode==NULL)

{

printf("\nMalloc failure");

exit(1);

}

newnode->data=item;

newnode->next=NULL;

// Case i - List is empty

if(head==NULL)

head=newnode;

else // Case ii - adding the smallest item

if(item < head->data)

{

newnode->next = head;

head = newnode;

}

else // Case iii

{

prev=head;

curr=head->next;

while(curr && item > curr->data)

{

prev=prev->next;

curr=curr->next;

}//end of while

newnode->next=curr;

prev->next=newnode;

}//end of else

return head;

}

void disp(NODE \*head)

{

if(head==NULL)

{

printf("\nWarehouse is empty");

return;

}

printf("\nThe warehouse items are: ");

while(head)

{

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

head=head->next;

}

}

int search(NODE\* head, int item)

{

if(head==NULL)

{

printf("\nWarehouse is empty");

return 0;

}

while(head && (item > head->data))

head=head->next;

if(head==NULL)

return 0;

if(item==head->data)

return 1;

else

return 0;

}

**Termwork 7**

/\* TW 7 Program for addition of two polynomials

\* polynomial are stored using structure

\* and program uses array of structures

\*/

#include<stdio.h>

/\* declare structure for polynomial \*/

struct poly

{

int coeff;

int expo;

};

/\* declare three arrays p1, p2, p3 of type structure poly.

\* each polynomial can have maximum of ten terms

\* addition result of p1 and p2 is stored in p3 \*/

/\* function prototypes \*/

int readPoly(struct poly []);

int addPoly(struct poly [],struct poly [],int ,

int ,struct poly []);

void displayPoly( struct poly [],int terms);

int main()

{

int t1,t2,t3;

struct poly p1[10],p2[10],p3[10];

/\* read and display first polynomial \*/

t1=readPoly(p1);

printf(" \n First polynomial : ");

displayPoly(p1,t1);

/\* read and display second polynomial \*/

t2=readPoly(p2);

printf(" \n Second polynomial : ");

displayPoly(p2,t2);

/\* add two polynomials and display resultant polynomial \*/

t3=addPoly(p1,p2,t1,t2,p3);

printf(" \n\n Resultant polynomial after addition : ");

displayPoly(p3,t3);

printf("\n");

return 0;

}

int readPoly(struct poly p[10])

{

int t1,i;

printf("\n\n Enter the total number of terms in the

polynomial:");

scanf("%d",&t1);

printf("\n Enter the COEFFICIENT and EXPONENT in

DESCENDING ORDER\n");

for(i=0;i<t1;i++){

printf(" Enter the Coefficient(%d): ",i+1);

scanf("%d",&p[i].coeff);

printf(" Enter the exponent(%d): ",i+1);

scanf("%d",&p[i].expo); /\* only statement in loop \*/

}

return(t1);

}

int addPoly(struct poly p1[10],struct poly p2[10],int t1,int t2,struct poly p3[10])

{

int i,j,k;

i=j=k=0;

while(i<t1 && j<t2)

{

if(p1[i].expo==p2[j].expo)

{

p3[k].coeff=p1[i].coeff + p2[j].coeff;

p3[k].expo=p1[i].expo;

i++;

j++;

k++;

}

else if(p1[i].expo>p2[j].expo)

{

p3[k].coeff=p1[i].coeff;

p3[k].expo=p1[i].expo;

i++;

k++;

}

else

{

p3[k].coeff=p2[j].coeff;

p3[k].expo=p2[j].expo;

j++;

k++;

}

}

/\* for rest over terms of polynomial 1 \*/

while(i<t1)

{

p3[k].coeff=p1[i].coeff;

p3[k].expo=p1[i].expo;

i++;

k++;

}

/\* for rest over terms of polynomial 2 \*/

while(j<t2){

p3[k].coeff=p2[j].coeff;

p3[k].expo=p2[j].expo;

j++;

k++;

}

return(k); /\* k is number of terms in resultant polynomial\*/

}

void displayPoly(struct poly p[10],int term)

{

int k;

for(k=0;k<term-1;k++)

printf("%d(x^%d)+",p[k].coeff,p[k].expo);

printf("%d(x^%d)",p[term-1].coeff,p[term-1].expo);

}

**Termwork 8**

// TW 8 BST

#include <stdio.h>

#include <stdlib.h>

typedef struct node

{

int data;

struct node \*left, \*right;

}NODE;

NODE\* newNode(int item)

{

NODE \* tmp=(NODE \*)malloc(sizeof(NODE));

tmp->data=item;

tmp->left=tmp->right=NULL;

return tmp;

}

NODE\* addBST(NODE\* root, int key)

{

if(root==NULL)

return newNode(key);

if(key<root->data)

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

else

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

return root;

}

int leaf(NODE\* root)

{

if(root==NULL)

return 0;

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

return 1;

return leaf(root->left)+leaf(root->right);

}

int nonleaf(NODE\* root)

{

if(root==NULL)

return 0;

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

return 0;

return 1+nonleaf(root->left)+nonleaf(root->right);

}

int two(NODE\* root)

{

if(root==NULL)

return 0;

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

return 1+two(root->left)+two(root->right);

if(root->left==NULL)

return two(root->right);

if(root->right==NULL)

return two(root->left);

return 0;

}

int total(NODE\* root)

{

if(root==NULL)

return 0;

return 1+total(root->left)+total(root->right);

}

int main()

{

int i,n,key;

NODE \*root=NULL;

printf("\nEnter n: ");

scanf("%d", &n);

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

{

printf("\nEnter int %d: ", i+1);

scanf("%d", &key);

root=addBST(root, key);

}

printf("\nNo of leaf nodes: %d", leaf(root));

printf("\nNo of non-leaf nodes: %d", nonleaf(root));

printf("\nNo of nodes with degree two: %d", two(root));

printf("\nTotal no of nodes: %d", total(root));

return 0;

}

**Termwork 9**

// TW 9 Expression Tree

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

#include <ctype.h>

/\* A binary tree node has data, pointer to left child

and a pointer to right child \*/

struct node {

char data;

struct node\* left;

struct node\* right;

struct node\* next;

};

struct node \*head=NULL;

/\* Helper function that allocates a new node with the

given data and NULL left and right pointers. \*/

struct node\* newNode(char data)

{

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

node->data = data;

node->left = NULL;

node->right = NULL;

node->next = NULL;

return (node);

}

bool isOperator(char c)

{

return (c == '+' || c == '-' || c == '\*' || c == '/' || c == '^');

}

void inorder(struct node\* node)

{

if(node)

{

if(isalnum(node->data))

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

else

{

printf("(");

inorder(node->left);

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

inorder(node->right);

printf(")");

}

}

}

void postorder(struct node \*root)

{

if(root!=NULL)

{

postorder(root->left);

postorder(root->right);

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

}

}

void preorder(struct node \*root)

{

if(root!=NULL)

{

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

preorder(root->left);

preorder(root->right);

}

}

void push(struct node\* x)

{

if(head==NULL)

head = x;

else

{

(x)->next = head;

head = x;

}

}

struct node\* pop()

{

// Poping out the top most[ pointed with head] element

struct node\* p = head;

head = head->next;

return p;

}

int main()

{

char s[30];

printf("\nEnter the exp: ");

gets(s);

int l = strlen(s);

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

printf("%c ", s[i]);

printf("\nl = %d\n", l);

struct node \*x, \*y, \*z;

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

// if read character is operator then poping two

// other elements from stack and making a binary

// tree

if (s[i] == '+' || s[i] == '-' || s[i] == '\*'

|| s[i] == '/')

{

z = newNode(s[i]);

x = pop();

y = pop();

z->left = y;

z->right = x;

push(z);

}

else

{

z = newNode(s[i]);

push(z);

}

}

printf("The given expression is\n");

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

{

printf("%c",s[i]);

}

printf(" \n The Inorder Traversal of Expression Tree:\t ");

inorder(z);

printf(" \n The Preorder Traversal of Expression Tree:\t ");

preorder(z);

printf(" \n The Postorder Traversal of Expression Tree:\t ");

postorder(z);

return 0;

}

**Termwork 10**

// TW 10 Doubly Linked List

#include <stdio.h>

#include <stdlib.h>

typedef struct node

{

int data;

struct node \*next, \*prev;

}NODE;

//Prototypes

NODE\* addFront(NODE\*);

NODE\* addRear(NODE\*);

void disp(NODE\*);

void search(NODE\*);

//Driver

int main()

{

//Create an empty DLL

NODE \*head=NULL;

int opt;

while(1)

{

printf("\n1: FrontAdd\t2: RearAdd\t3: Disp\t");

printf("4: Search\t5: Exit");

printf("\nEnter an option: ");

scanf("%d", &opt);

switch(opt)

{

case 1: head=addFront(head); break;

case 2: head=addRear(head); break;

case 3: disp(head); break;

case 4: search(head); break;

case 5: exit(0);

}

}

return 0;

}

//Adding a node at the front of the DLL

NODE\* addFront(NODE \*head)

{

int data;

NODE \*tmp=(NODE \*)malloc(sizeof(NODE));

if(tmp==NULL)

{

printf("\nMalloc failure");

exit(1);

}

printf("\nEnter data to add: ");

scanf("%d", &data);

tmp->data=data;

tmp->prev=tmp->next=NULL;

if(head==NULL) //Empty List

head=tmp;

else //List has atleast one datanode

{

tmp->next=head;

head->prev=tmp;

head=tmp;

}

return(head);

}

void disp(NODE \*head)

{

if(head==NULL)

{

printf("\nList is empty");

return;

}

printf("\nThe List contents are: ");

while(head)

{

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

head=head->next;

}

}

//Adding a node at the end of the DLL

NODE\* addRear(NODE \*head)

{

int data;

NODE \*trav;

NODE \*tmp=(NODE \*)malloc(sizeof(NODE));

if(tmp==NULL)

{

printf("\nMalloc failure");

exit(1);

}

printf("\nEnter data to add: ");

scanf("%d", &data);

tmp->data=data;

tmp->prev=tmp->next=NULL;

if(head==NULL) //Empty List

head=tmp;

else //List has atleast one datanode

{

trav=head;

while(trav->next)

trav=trav->next;

trav->next=tmp;

tmp->prev=trav;

}

return(head);

}

//Search data in DLL

void search(NODE \*head)

{

int item, count=0;

if(head==NULL)

{

printf("\nList is empty");

return;

}

printf("\nEnter data to search: ");

scanf("%d", &item);

while(head)

{

if(item==head->data)

{

printf("\nData %d present at position %d", item, count+1);

return;

}

else

{

count++;

head=head->next;

}

}

printf("\nData %d not present", item);

}