**Write a program to insert and delete the nodes in link list at different locations.**

#include<stdio.h>

#include<stdlib.h>

typedef struct node

{

int data;

struct node \*back;

struct node \*next;

}node;

node\* creat(node \*start)

{

node \*temp,\*last;

int i;

do

{

printf("enter 1/0\t");

scanf("%d",&i);

if(i==1)

{

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

if(temp==NULL)

exit(1);

printf("enter data\t");

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

if(start==NULL)

{

start=temp;

start->back=NULL;

last=temp;

}

else

{

last->next=temp;

temp->back=last;

last=temp;

}

last->next=NULL;

}

}while(i!=0);

return start;

}

void show\_f(node \*start)

{

node \*temp;

temp=start;

while(temp!=NULL)

{

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

temp=temp->next;

}

}

void show\_b(node \*start)

{

node \*temp;

temp=start;

while(temp->next!=NULL)

{

temp=temp->next;

}

while(temp!=NULL)

{

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

temp=temp->back;

}

}

node\* insert(node \*start)

{

node \*t,\*temp;

t=start;

int i=0,n;

while(t!=NULL)

{

i++;

t=t->next;

}

printf("enter at what position want to insert\t");

scanf("%d",&n);

if(n>i+1)

{

printf("not possible");

}

else

{

int j;

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

printf("enter data\t");

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

if(n==1)

{

temp->back=NULL;

temp->next=start;

start->back=temp;

start=temp;

}

else

{

t=start;

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

{

t=t->next;

}

temp->next=t->next;

temp->back=t;

t->next=temp;

if(n<=i)

(temp->next)->back=temp;

}

}

return start;

}

node\* delet(node \*start)

{

node \*t;

int n,j,i=0;

t=start;

printf("enter position want to delete");

scanf("%d",&n);

while(t!=NULL)

{

i++;

t=t->next;

}

if((n>1)&&(n<=i))

{

t=start;

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

{

t=t->next;

}

t->next=(t->next)->next;

if(n!=i)

(t->next)->back=t;

}

else if((n==1)&&(i>1))

{

t=start;

(t->next)->back=NULL;

start=t->next;

}

else if((n==1)&&(i==1))

{

start=NULL;

}

else if(i==0)

printf("list is empty\n");

else

printf("invalid choice");

return start;

}

void main()

{

int h;

node \*start;

start=NULL;

do

{

printf("\n1 for creat , 2 to insert , 3 to delete \n 4 to show forward , 5 to backward show , 6 to exit\t");

scanf("%d",&h);

switch(h)

{

case 1:

{

start=creat(start);

break;

}

case 2:

{

start=insert(start);

break;

}

case 3:

{

start=delet(start);

break;

}

case 4:

{

show\_f(start);

break;

}

case 5:

{

show\_b(start);

break;

}

case 6:

break;

default:

printf("wrong choice\n");

}

}while(h!=6);

}

**Write a program to implement the concept of binary search tree.**

#include<bits/stdc++.h>

using namespace std;

typedef struct node

{

int data;

struct node \*left;

struct node \*right;

}node;

node\* creat()

{

int ch;

node \*temp;

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

printf("enter data");

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

printf("does %d has left child\t",(temp->data));

scanf("%d",&ch);

if(ch)

{

temp->left=creat();

}

else

temp->left=NULL;

printf("does %d has right child\t",(temp->data));

scanf("%d",&ch);

if(ch)

{

temp->right=creat();

}

else

temp->right=NULL;

return temp;

}

node \* minValueNode(struct node\* node)

{

struct node\* current = node;

while (current->left != NULL)

current = current->left;

return current;

}

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

{

if (root == NULL) return root;

if (key < root->data)

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

else if (key > root->data)

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

else

{

if (root->left == NULL)

{

node \*temp = root->right;

free(root);

return temp;

}

else if (root->right == NULL)

{

node \*temp = root->left;

free(root);

return temp;

}

node\* temp;

temp = minValueNode(root->right);

root->data = temp->data;

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

}

return root;

}

node\* insert(node \*root,int data)

{

if(root == NULL)

{

node \*temp;

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

temp->data = data;

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

return temp;

}

if(root->data > data)

{

root->left = insert(root->left,data);

}

else

{

root->right = insert(root->right,data);

}

}

node\* search(node \*a,int value)

{

node \*temp;

if(a==NULL)

return NULL;

if(a->data==value)

{

return a;

}

else if((a->data)>value)

{

temp=search(a->left,value);

return temp;

}

else

{

temp=search(a->right,value);

return temp;

}

}

int main()

{

node \*root;

root = NULL;

int value,data;

while(1)

{

cout<<"enter 0 to quit & 1 to insert, 2 to search, 3 to delete\n";

cin>>value;

if(value == 0)

break;

if(value == 1)

{

cout<<"enter data to be inserted\n";

cin>>data;

root = insert(root,data);

}

else if(value == 2)

{

node \*temp;

cout<<"enter data to be searhed\n";

cin>>data;

temp = search(root,data);

if(temp == NULL)

cout<<"Data not found\n";

else

cout<<"found\n";

}

else if(value == 3)

{

cout<<"enter data to be delete\n";

cin>>data;

node \*temp;

temp = search(root,data);

if(temp == NULL)

cout<<"Data not found\n";

else{

root = deleteNode(root,data);

cout<<"Removed\n";

}

}

}

return 0;

}

**Write a program to implement the concept of AVL tree.**

#include<bits/stdc++.h>

using namespace std;

struct node

{

int key;

struct node \*left;

struct node \*right;

int height;

};

int max(int a, int b);

int height(struct node \*N)

{

if (N == NULL)

return 0;

return N->height;

}

int max(int a, int b)

{

return (a > b)? a : b;

}

struct node\* newNode(int key)

{

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

node->key = key;

node->left = NULL;

node->right = NULL;

node->height = 1;

return(node);

}

struct node \*rightRotate(struct node \*y)

{

struct node \*x = y->left;

struct node \*T2 = x->right;

x->right = y;

y->left = T2;

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

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

return x;

}

struct node \*leftRotate(struct node \*x)

{

struct node \*y = x->right;

struct node \*T2 = y->left;

y->left = x;

x->right = T2;

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

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

return y;

}

int getBalance(struct node \*N)

{

if (N == NULL)

return 0;

return height(N->left) - height(N->right);

}

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

{

if (node == NULL)

return(newNode(key));

if (key < node->key)

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

else

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

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

int balance = getBalance(node);

if (balance > 1 && key < node->left->key)

return rightRotate(node);

if (balance < -1 && key > node->right->key)

return leftRotate(node);

if (balance > 1 && key > node->left->key)

{

node->left = leftRotate(node->left);

return rightRotate(node);

}

if (balance < -1 && key < node->right->key)

{

node->right = rightRotate(node->right);

return leftRotate(node);

}

return node;

}

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

{

struct node\* current = node;

while (current->left != NULL)

current = current->left;

return current;

}

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

{

if (root == NULL)

return root;

if ( key < root->key )

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

else if( key > root->key )

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

else

{

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

{

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

if(temp == NULL)

{

temp = root;

root = NULL;

}

else

\*root = \*temp;

free(temp);

}

else

{

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

root->key = temp->key;

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

}

}

if (root == NULL)

return root;

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

int balance = getBalance(root);

if (balance > 1 && getBalance(root->left) >= 0)

return rightRotate(root);

if (balance > 1 && getBalance(root->left) < 0)

{

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

return rightRotate(root);

}

if (balance < -1 && getBalance(root->right) <= 0)

return leftRotate(root);

if (balance < -1 && getBalance(root->right) > 0)

{

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

return leftRotate(root);

}

return root;

}

void preOrder(struct node \*root)

{

if(root != NULL)

{

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

preOrder(root->left);

preOrder(root->right);

}

}

int main()

{

struct node \*root = NULL;

int value,data;

while(1)

{

cout<<"enter 0 to quit & 1 to insert, 2 to preorder, 3 to delete\n";

cin>>value;

if(value == 0)

break;

if(value == 1)

{

cout<<"enter data to be inserted\n";

cin>>data;

root = insert(root,data);

}

else if(value == 2)

{

preOrder(root);

}

else if(value == 3)

{

cout<<"enter data to be delete\n";

cin>>data;

root = deleteNode(root,data);

cout<<"Removed\n";

}

}

return 0;

}

**Write a program to implement the heap sort.**

#include <bits/stdc++.h>

using namespace std;

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

{

int largest = i;

int l = 2\*i + 1;

int r = 2\*i + 2;

if (l < n && arr[l] > arr[largest])

largest = l;

if (r < n && arr[r] > arr[largest])

largest = r;

if (largest != i)

{

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

heapify(arr, n, largest);

}

}

void heapSort(int arr[], int n)

{

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

heapify(arr, n, i);

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

{

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

heapify(arr, i, 0);

}

}

void printArray(int arr[], int n)

{

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

cout << arr[i] << " ";

cout << "\n";

}

int main()

{

int n,i;

cout<<"enter size of array\n";

cin>>n;

int arr[n];

cout<<"enter array elements\n";

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

cin>>arr[i];

heapSort(arr, n);

cout << "Sorted array is \n";

printArray(arr, n);

}

**Write a program to implement the all pair shortest path problem.**

#include<bits/stdc++.h>

using namespace std;

int main()

{

int n,i,j,k;

cin>>n;

int dp[n][n];

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

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

cin>>dp[i][j];

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

{

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

{

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

dp[i][j]=min(dp[i][j],dp[i][k]+dp[k][j]);

}

}

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

{

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

cout<<dp[i][j]<<" ";

cout<<"\n";

}

return 0;

}

**Write a program to implement the concept of Binomial heaps.**

#include<bits/stdc++.h>

using namespace std;

struct node{

int data;

int degree;

node \*child;

node \*sibling;

node \*parent;

};

node\* newNode(int key)

{

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

temp->data = key;

temp->degree = 0;

temp->child = temp->parent = temp->sibling = NULL;

return temp;

}

node\* merge\_btree(node \*head, node \*temp)

{

if(head->data > temp->data)

swap(head,temp);

temp->parent = head;

temp->sibling = head->child;

head->child = temp;

head->degree++;

return head;

}

list<node\*> union\_bionomial\_heap(list<node\*> l1, list<node\*> l2)

{

list<node\*> \_new;

list<node\*>::iterator it,ot;

it = l1.begin();

ot = l2.begin();

while(it!=l1.end() && ot!=l2.end())

{

if((\*it)->degree <= (\*ot)->degree)

{

\_new.push\_back(\*it);

it++;

}

else

{

\_new.push\_back(\*ot);

ot++;

}

}

while(it!=l1.end())

{

\_new.push\_back(\*it);

it++;

}

while(ot!=l2.end())

{

\_new.push\_back(\*ot);

ot++;

}

return \_new;

}

list<node\*> adjust(list<node\*> \_heap)

{

if(\_heap.size() <= 1)

return \_heap;

list<node\*> new\_heap;

list<node\*>::iterator it1,it2,it3;

it1=it2=it3 = \_heap.begin();

if(\_heap.size() == 2)

{

it2 = it1; it2++;

it3 = \_heap.end();

}

else

{

it2++; it3=it2; it3++;

}

while(it1 != \_heap.end())

{

if(it2 == \_heap.end())

it1++;

else if((\*it1)->degree < (\*it2)->degree)

{

it1++; it2++;

if(it3!=\_heap.end())

it3++;

}

else if(it3!=\_heap.end() && (\*it1)->degree == (\*it2)->degree && (\*it1)->degree == (\*it3)->degree)

{

it1++; it2++; it3++;

}

else if((\*it1)->degree == (\*it2)->degree)

{

node \*temp;

\*it1 = merge\_btree(\*it1,\*it2);

it2 = \_heap.erase(it2);

if(it3 != \_heap.end())

it3++;

}

}

return \_heap;

}

list<node\*> insert\_a\_tree\_in\_heap(list<node\*> \_heap,node \*tree)

{

list<node\*> temp;

temp.push\_back(tree);

temp = union\_bionomial\_heap(\_heap,temp);

return adjust(temp);

}

list<node\*> insert\_a\_key\_in\_heap(list<node\*> \_head,int key)

{

node \*temp;

temp = newNode(key);

return insert\_a\_tree\_in\_heap(\_head,temp);

}

void print\_tree(node \*h)

{

while(h)

{

cout<<h->data<<" ";

print\_tree(h->child);

h = h->sibling;

}

}

void print\_heap(list<node\*> \_heap)

{

list<node\*> ::iterator it;

it = \_heap.begin();

while(it != \_heap.end())

{

print\_tree(\*it);

cout<<"\n";

it++;

}

}

node\* get\_min(list<node\*> \_heap)

{

node \*temp;

list<node\*>::iterator it;

it = \_heap.begin();

temp = \*it;

while(it != \_heap.end())

{

if((\*it)->data < temp->data)

{

temp = \*it;

}

it++;

}

return temp;

}

list<node\*> remove\_min\_from\_tree\_return\_bheap(node \*tree)

{

list<node\*> heap;

node \*temp = tree->child;

node \*lo;

while(temp)

{

lo = temp;

temp = temp->sibling;

lo->sibling = NULL;

heap.push\_front(lo);

}

return heap;

}

list<node\*> extract\_min(list<node\*> \_heap)

{

list<node\*> new\_heap,lo;

node \*temp;

temp = get\_min(\_heap);

list<node\*>::iterator it;

it = \_heap.begin();

while(it != \_heap.end())

{

if(\*it != temp)

{

new\_heap.push\_back(\*it);

}

it++;

}

lo = remove\_min\_from\_tree\_return\_bheap(temp);

new\_heap = union\_bionomial\_heap(new\_heap,lo);

new\_heap = adjust(new\_heap);

return new\_heap;

}

int main()

{

int ch,key;

list<node\*> \_heap;

while(1)

{

cout<<"enter 0 to break, 1 to insert element, 2 to print min,3 to extract min and print heap, 4 to print heap\n";

cin>>ch;

if(ch == 0)

break;

else if(ch == 1)

{

cout<<"enter key to be inserted\n";

cin>>key;

\_heap = insert\_a\_key\_in\_heap(\_heap,key);

}

else if(ch == 2)

{

node \*temp;

temp = get\_min(\_heap);

cout<<temp->data<<"\n";

}

else if(ch == 3)

{

\_heap = extract\_min(\_heap);

print\_heap(\_heap);

}

else if(ch == 4)

{

print\_heap(\_heap);

}

else

cout<<"Invalid choice\n";

}

return 0;

}