**Practical-4**: Implementation of operations on Fibonacci heaps.

#include <cmath>

#include <cstdlib>

#include <iostream>

struct node {

node\* parent;

node\* child;

node\* left;

node\* right;

int key;

int degree;

char mark;

char c;

};

struct node\* mini = NULL;

int no\_of\_nodes = 0;

void insertion(int val)

{

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

new\_node->key = val;

new\_node->degree = 0;

new\_node->mark = 'W';

new\_node->c = 'N';

new\_node->parent = NULL;

new\_node->child = NULL;

new\_node->left = new\_node;

new\_node->right = new\_node;

if (mini != NULL) {

(mini->left)->right = new\_node;

new\_node->right = mini;

new\_node->left = mini->left;

mini->left = new\_node;

if (new\_node->key < mini->key)

mini = new\_node;

}

else {

mini = new\_node;

}

no\_of\_nodes++;

}

void Fibonnaci\_link(struct node\* ptr2, struct node\* ptr1)

{

(ptr2->left)->right = ptr2->right;

(ptr2->right)->left = ptr2->left;

if (ptr1->right == ptr1)

mini = ptr1;

ptr2->left = ptr2;

ptr2->right = ptr2;

ptr2->parent = ptr1;

if (ptr1->child == NULL)

ptr1->child = ptr2;

ptr2->right = ptr1->child;

ptr2->left = (ptr1->child)->left;

((ptr1->child)->left)->right = ptr2;

(ptr1->child)->left = ptr2;

if (ptr2->key < (ptr1->child)->key)

ptr1->child = ptr2;

ptr1->degree++;

}

void Consolidate()

{

int temp1;

float temp2 = (log(no\_of\_nodes)) / (log(2));

int temp3 = temp2;

struct node\* arr[temp3];

for (int i = 0; i <= temp3; i++)

arr[i] = NULL;

node\* ptr1 = mini;

node\* ptr2;

node\* ptr3;

node\* ptr4 = ptr1;

do {

ptr4 = ptr4->right;

temp1 = ptr1->degree;

while (arr[temp1] != NULL) {

ptr2 = arr[temp1];

if (ptr1->key > ptr2->key) {

ptr3 = ptr1;

ptr1 = ptr2;

ptr2 = ptr3;

}

if (ptr2 == mini)

mini = ptr1;

Fibonnaci\_link(ptr2, ptr1);

if (ptr1->right == ptr1)

mini = ptr1;

arr[temp1] = NULL;

temp1++;

}

arr[temp1] = ptr1;

ptr1 = ptr1->right;

} while (ptr1 != mini);

mini = NULL;

for (int j = 0; j <= temp3; j++) {

if (arr[j] != NULL) {

arr[j]->left = arr[j];

arr[j]->right = arr[j];

if (mini != NULL) {

(mini->left)->right = arr[j];

arr[j]->right = mini;

arr[j]->left = mini->left;

mini->left = arr[j];

if (arr[j]->key < mini->key)

mini = arr[j];

}

else {

mini = arr[j];

}

if (mini == NULL)

mini = arr[j];

else if (arr[j]->key < mini->key)

mini = arr[j];

}

}

}

void Extract\_min()

{

if (mini == NULL)

cout << "The heap is empty" << endl;

else {

node\* temp = mini;

node\* pntr;

pntr = temp;

node\* x = NULL;

if (temp->child != NULL) {

x = temp->child;

do {

pntr = x->right;

(mini->left)->right = x;

x->right = mini;

x->left = mini->left;

mini->left = x;

if (x->key < mini->key)

mini = x;

x->parent = NULL;

x = pntr;

} while (pntr != temp->child);

}

(temp->left)->right = temp->right;

(temp->right)->left = temp->left;

mini = temp->right;

if (temp == temp->right && temp->child == NULL)

mini = NULL;

else {

mini = temp->right;

Consolidate();

}

no\_of\_nodes--;

}

}

void Cut(struct node\* found, struct node\* temp)

{

if (found == found->right)

temp->child = NULL;

(found->left)->right = found->right;

(found->right)->left = found->left;

if (found == temp->child)

temp->child = found->right;

temp->degree = temp->degree - 1;

found->right = found;

found->left = found;

(mini->left)->right = found;

found->right = mini;

found->left = mini->left;

mini->left = found;

found->parent = NULL;

found->mark = 'B';

}

void Cascase\_cut(struct node\* temp)

{

node\* ptr5 = temp->parent;

if (ptr5 != NULL) {

if (temp->mark == 'W') {

temp->mark = 'B';

}

else {

Cut(temp, ptr5);

Cascase\_cut(ptr5);

}

}

}

void Decrease\_key(struct node\* found, int val)

{

if (mini == NULL)

cout << "The Heap is Empty" << endl;

if (found == NULL)

cout << "Node not found in the Heap" << endl;

found->key = val;

struct node\* temp = found->parent;

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

Cut(found, temp);

Cascase\_cut(temp);

}

if (found->key < mini->key)

mini = found;

}

void Find(struct node\* mini, int old\_val, int val)

{

struct node\* found = NULL;

node\* temp5 = mini;

temp5->c = 'Y';

node\* found\_ptr = NULL;

if (temp5->key == old\_val) {

found\_ptr = temp5;

temp5->c = 'N';

found = found\_ptr;

Decrease\_key(found, val);

}

if (found\_ptr == NULL) {

if (temp5->child != NULL)

Find(temp5->child, old\_val, val);

if ((temp5->right)->c != 'Y')

Find(temp5->right, old\_val, val);

}

temp5->c = 'N';

found = found\_ptr;

}

void Deletion(int val)

{

if (mini == NULL)

cout << "The heap is empty" << endl;

else {

Find(mini, val, 0);

Extract\_min();

cout << "Key Deleted" << endl;

}

}

void display()

{

node\* ptr = mini;

if (ptr == NULL)

cout << "The Heap is Empty" << endl;

else {

cout << "The root nodes of Heap are: " << endl;

do {

cout << ptr->key;

ptr = ptr->right;

if (ptr != mini) {

cout << "-->";

}

} while (ptr != mini && ptr->right != NULL);

cout << endl

<< "The heap has " << no\_of\_nodes << " nodes" << endl

<< endl;

}

}

int main()

{

cout << "Creating an initial heap" << endl;

insertion(5);

insertion(2);

insertion(8);

display();

cout << "Extracting min" << endl;

Extract\_min();

display();

cout << "Decrease value of 8 to 7" << endl;

Find(mini, 8, 7);

display();

cout << "Delete the node 7" << endl;

Deletion(7);

display();

return 0;

}

**Output:**

