\*Write a program using advanced data structure like Binomial heap or fibbonacci heap

#include <stdio.h>

#include <stdlib.h>

// Node structure for the Fibonacci heap

typedef struct Node {

int key;

int degree;

struct Node\* parent;

struct Node\* child;

struct Node\* left;

struct Node\* right;

int mark;

} Node;

// Fibonacci heap structure

typedef struct FibonacciHeap {

Node\* min;

int num\_nodes;

} FibonacciHeap;

// Function to create a new node

Node\* createNode(int key) {

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

newNode->key = key;

newNode->degree = 0;

newNode->parent = NULL;

newNode->child = NULL;

newNode->left = newNode;

newNode->right = newNode;

newNode->mark = 0;

return newNode;

}

// Function to create an empty Fibonacci heap

FibonacciHeap\* createFibonacciHeap() {

FibonacciHeap\* newHeap = (FibonacciHeap\*)malloc(sizeof(FibonacciHeap));

newHeap->min = NULL;

newHeap->num\_nodes = 0;

return newHeap;

}

// Function to link two Fibonacci heap trees of the same degree

void linkTrees(Node\* y, Node\* x) {

y->left->right = y->right;

y->right->left = y->left;

y->parent = x;

if (x->child == NULL) {

x->child = y;

y->left = y;

y->right = y;

} else {

y->left = x->child;

y->right = x->child->right;

x->child->right = y;

y->right->left = y;

}

x->degree++;

y->mark = 0;

}

// Function to consolidate the trees in the Fibonacci heap

void consolidate(FibonacciHeap\* heap) {

int maxDegree = heap->num\_nodes;

Node\* arr[maxDegree];

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

arr[i] = NULL;

}

Node\* curr = heap->min;

Node\* end = heap->min->left;

while (curr != end) {

Node\* x = curr;

int d = x->degree;

while (arr[d] != NULL) {

Node\* y = arr[d];

if (x->key > y->key) {

Node\* temp = x;

x = y;

y = temp;

}

linkTrees(y, x);

arr[d] = NULL;

d++;

}

arr[d] = x;

curr = curr->right;

}

heap->min = NULL;

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

if (arr[i] != NULL) {

if (heap->min == NULL) {

heap->min = arr[i];

} else {

arr[i]->left->right = arr[i]->right;

arr[i]->right->left = arr[i]->left;

arr[i]->left = heap->min;

arr[i]->right = heap->min->right;

heap->min->right = arr[i];

arr[i]->right->left = arr[i];

if (arr[i]->key < heap->min->key) {

heap->min = arr[i];

}

}

}

}

}

// Function to extract the minimum key from the Fibonacci heap

int extractMin(FibonacciHeap\* heap) {

Node\* minNode = heap->min;

if (minNode != NULL) {

Node\* child = minNode->child;

Node\* temp = child;

while (child != NULL) {

child->parent = NULL;

child = child->right;

if (child == temp) {

break;

}

}

minNode->left->right = minNode->right;

minNode->right->left = minNode->left;

if (minNode == minNode->right) {

heap->min = NULL;

} else {

heap->min = minNode->right;

consolidate(heap);

}

int minKey = minNode->key;

free(minNode);

heap->num\_nodes--;

return minKey;

}

return -1;

}

// Function to insert a new key into the Fibonacci heap

void insertKey(FibonacciHeap\* heap, int key) {

Node\* newNode = createNode(key);

if (heap->min == NULL) {

heap->min = newNode;

} else {

newNode->left = heap->min;

newNode->right = heap->min->right;

heap->min->right = newNode;

newNode->right->left = newNode;

if (newNode->key < heap->min->key) {

heap->min = newNode;

}

}

heap->num\_nodes++;

}

// Function to decrease the key of a node in the Fibonacci heap

void decreaseKey(FibonacciHeap\* heap, Node\* x, int newKey) {

if (newKey > x->key) {

return;

}

x->key = newKey;

Node\* y = x->parent;

if (y != NULL && x->key < y->key) {

cut(heap, x, y);

cascadingCut(heap, y);

}

if (x->key < heap->min->key) {

heap->min = x;

}

}

// Function to cut a node from its parent in the Fibonacci heap

void cut(FibonacciHeap\* heap, Node\* x, Node\* y) {

if (x->right == x) {

y->child = NULL;

} else {

x->left->right = x->right;

x->right->left = x->left;

if (y->child == x) {

y->child = x->right;

}

}

y->degree--;

x->left = heap->min;

x->right = heap->min->right;

heap->min->right = x;

x->right->left = x;

x->parent = NULL;

x->mark = 0;

}

// Function to perform cascading cut in the Fibonacci heap

void cascadingCut(FibonacciHeap\* heap, Node\* y) {

Node\* z = y->parent;

if (z != NULL) {

if (y->mark == 0) {

y->mark = 1;

} else {

cut(heap, y, z);

cascadingCut(heap, z);

}

}

}

// Function to delete a node from the Fibonacci heap

void deleteNode(FibonacciHeap\* heap, Node\* x) {

decreaseKey(heap, x, INT\_MIN);

extractMin(heap);

}

// Function to print the Fibonacci heap

void printHeap(Node\* node, Node\* start) {

Node\* temp = node;

if (temp != NULL) {

do {

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

if (temp->child != NULL) {

printHeap(temp->child, temp->child);

}

temp = temp->right;

} while (temp != start);

}

}

int main() {

FibonacciHeap\* heap = createFibonacciHeap();

insertKey(heap, 7);

insertKey(heap, 2);

insertKey(heap, 9);

insertKey(heap, 1);

insertKey(heap, 5);

printf("Original Fibonacci Heap: ");

printHeap(heap->min, heap->min);

printf("\n");

int minKey = extractMin(heap);

printf("Extracted Minimum Key: %d\n", minKey);

printf("Updated Fibonacci Heap: ");

printHeap(heap->min, heap->min);

printf("\n");

decreaseKey(heap, heap->min->right->child, 3);

printf("Decreased Key Fibonacci Heap: ");

printHeap(heap->min, heap->min);

printf("\n");

deleteNode(heap, heap->min);

printf("Deleted Key Fibonacci Heap: ");

printHeap(heap->min, heap->min);

printf("\n");

return 0;

}