2.1 Создание бинарной кучи

#include <vector>

#include <algorithm>

#include <stdexcept>

class BinaryHeap {

private:

std::vector<int> heap;

int parent(int i) { return (i-1)/2; }

int left(int i) { return 2\*i + 1; }

int right(int i) { return 2\*i + 2; }

void siftUp(int i) {

while (i > 0 && heap[parent(i)] > heap[i]) {

std::swap(heap[parent(i)], heap[i]);

i = parent(i);

}

}

void siftDown(int i) {

int minIndex = i;

int l = left(i);

int r = right(i);

if (l < heap.size() && heap[l] < heap[minIndex])

minIndex = l;

if (r < heap.size() && heap[r] < heap[minIndex])

minIndex = r;

if (i != minIndex) {

std::swap(heap[i], heap[minIndex]);

siftDown(minIndex);

}

}

public:

void insert(int key) {

heap.push\_back(key);

siftUp(heap.size()-1);

}

int extractMin() {

if (heap.empty())

throw std::runtime\_error("Heap is empty");

int minVal = heap[0];

heap[0] = heap.back();

heap.pop\_back();

if (!heap.empty())

siftDown(0);

return minVal;

}

bool empty() const {

return heap.empty();

}

};

2.2) Создание Биноминальной кучи на С++  
  
#include <memory>

#include <vector>

struct BinomialNode {

int key;

int degree;

std::shared\_ptr<BinomialNode> parent;

std::shared\_ptr<BinomialNode> child;

std::shared\_ptr<BinomialNode> sibling;

BinomialNode(int k) : key(k), degree(0), parent(nullptr),

child(nullptr), sibling(nullptr) {}

};

class BinomialHeap {

private:

std::shared\_ptr<BinomialNode> head;

std::shared\_ptr<BinomialNode> mergeLists(std::shared\_ptr<BinomialNode> h1,

std::shared\_ptr<BinomialNode> h2) {

if (!h1) return h2;

if (!h2) return h1;

std::shared\_ptr<BinomialNode> newHead;

std::shared\_ptr<BinomialNode> \*current = &newHead;

while (h1 && h2) {

if (h1->degree <= h2->degree) {

\*current = h1;

h1 = h1->sibling;

} else {

\*current = h2;

h2 = h2->sibling;

}

current = &(\*current)->sibling;

}

if (h1) \*current = h1;

else \*current = h2;

return newHead;

}

void linkTrees(std::shared\_ptr<BinomialNode> child,

std::shared\_ptr<BinomialNode> parent) {

child->parent = parent;

child->sibling = parent->child;

parent->child = child;

parent->degree++;

}

public:

void insert(int key) {

BinomialHeap newHeap;

newHeap.head = std::make\_shared<BinomialNode>(key);

merge(newHeap);

}

void merge(BinomialHeap& other) {

head = mergeLists(head, other.head);

other.head = nullptr;

if (!head) return;

std::shared\_ptr<BinomialNode> prev = nullptr;

auto curr = head;

auto next = curr->sibling;

while (next) {

if (curr->degree != next->degree ||

(next->sibling && next->sibling->degree == curr->degree)) {

prev = curr;

curr = next;

} else if (curr->key <= next->key) {

curr->sibling = next->sibling;

linkTrees(next, curr);

} else {

if (!prev) head = next;

else prev->sibling = next;

linkTrees(curr, next);

curr = next;

}

next = curr->sibling;

}

}

};

2.3) Создание кучи Фибоначчи

#include <memory>

#include <unordered\_map>

#include <cmath>

struct FibonacciNode {

int key;

int degree;

bool marked;

std::shared\_ptr<FibonacciNode> parent;

std::shared\_ptr<FibonacciNode> child;

std::shared\_ptr<FibonacciNode> left;

std::shared\_ptr<FibonacciNode> right;

FibonacciNode(int k) : key(k), degree(0), marked(false),

parent(nullptr), child(nullptr) {

left = right = shared\_from\_this();

}

};

class FibonacciHeap {

private:

std::shared\_ptr<FibonacciNode> minNode;

int nodeCount;

void addToRootList(std::shared\_ptr<FibonacciNode> node) {

node->left = minNode;

node->right = minNode->right;

minNode->right->left = node;

minNode->right = node;

}

void link(std::shared\_ptr<FibonacciNode> child,

std::shared\_ptr<FibonacciNode> parent) {

// Удаляем child из корневого списка

child->left->right = child->right;

child->right->left = child->left;

// Добавляем child как дочерний для parent

child->parent = parent;

if (!parent->child) {

parent->child = child;

child->left = child->right = child;

} else {

child->left = parent->child;

child->right = parent->child->right;

parent->child->right->left = child;

parent->child->right = child;

}

parent->degree++;

child->marked = false;

}

void consolidate() {

std::unordered\_map<int, std::shared\_ptr<FibonacciNode>> degreeTable;

auto current = minNode;

std::vector<std::shared\_ptr<FibonacciNode>> nodes;

// Собираем все корневые узлы

do {

nodes.push\_back(current);

current = current->right;

} while (current != minNode);

for (auto node : nodes) {

int degree = node->degree;

while (degreeTable.find(degree) != degreeTable.end()) {

auto other = degreeTable[degree];

if (node->key > other->key) {

std::swap(node, other);

}

link(other, node);

degreeTable.erase(degree);

degree++;

}

degreeTable[degree] = node;

}

// Восстанавливаем minNode

minNode = nullptr;

for (auto& pair : degreeTable) {

if (!minNode || pair.second->key < minNode->key) {

minNode = pair.second;

}

}

}

public:

FibonacciHeap() : minNode(nullptr), nodeCount(0) {}

void insert(int key) {

auto node = std::make\_shared<FibonacciNode>(key);

if (!minNode) {

minNode = node;

} else {

addToRootList(node);

if (key < minNode->key) {

minNode = node;

}

}

nodeCount++;

}

int extractMin() {

if (!minNode) return -1;

auto z = minNode;

// Добавляем детей в корневой список

if (z->child) {

auto child = z->child;

do {

auto nextChild = child->right;

addToRootList(child);

child->parent = nullptr;

child = nextChild;

} while (child != z->child);

}

// Удаляем z из корневого списка

z->left->right = z->right;

z->right->left = z->left;

if (z == z->right) {

minNode = nullptr;

} else {

minNode = z->right;

consolidate();

}

nodeCount--;

return z->key;

}

};

2.4) Создание Хеш-таблица на С++

#include <vector>

#include <list>

#include <functional>

template<typename K, typename V>

class HashTable {

private:

std::vector<std::list<std::pair<K, V>>> buckets;

size\_t capacity;

size\_t size;

double loadFactor;

size\_t hash(const K& key) const {

return std::hash<K>{}(key) % capacity;

}

void resize() {

size\_t newCapacity = capacity \* 2;

std::vector<std::list<std::pair<K, V>>> newBuckets(newCapacity);

for (const auto& bucket : buckets) {

for (const auto& pair : bucket) {

size\_t index = std::hash<K>{}(pair.first) % newCapacity;

newBuckets[index].push\_back(pair);

}

}

buckets = std::move(newBuckets);

capacity = newCapacity;

}

public:

HashTable(size\_t cap = 8, double lf = 0.75)

: capacity(cap), size(0), loadFactor(lf) {

buckets.resize(capacity);

}

void put(const K& key, const V& value) {

if (static\_cast<double>(size) / capacity >= loadFactor) {

resize();

}

size\_t index = hash(key);

auto& bucket = buckets[index];

for (auto& pair : bucket) {

if (pair.first == key) {

pair.second = value;

return;

}

}

bucket.emplace\_back(key, value);

size++;

}

V get(const K& key) const {

size\_t index = hash(key);

const auto& bucket = buckets[index];

for (const auto& pair : bucket) {

if (pair.first == key) {

return pair.second;

}

}

throw std::runtime\_error("Key not found");

}

bool contains(const K& key) const {

size\_t index = hash(key);

const auto& bucket = buckets[index];

for (const auto& pair : bucket) {

if (pair.first == key) {

return true;

}

}

return false;

}

};