Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа машиностроения

КУРСОВАЯ РАБОТА

по дисциплине «Объектно-ориентированное программирование»

Тема: Реализация структур данных (Max heap, Fibonacci heap, Binomial heap) на языке C++

Выполнил студент
Группы 3331506/00401: Ленский А. В.

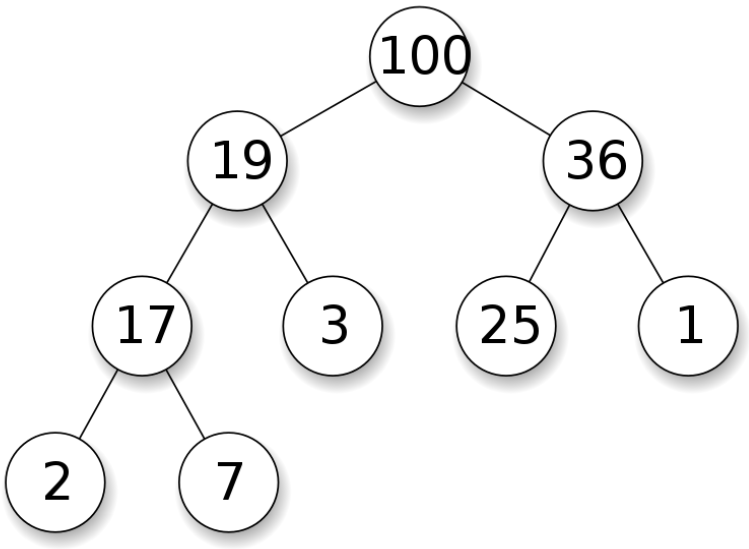
Преподаватель: Ананьевский М.С.

Санкт-Петербург
2023 г.

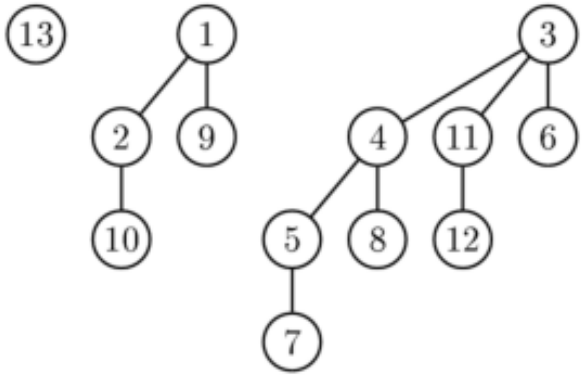
Введение

Куча — это структура данных, представляющая собой упорядоченное дерево, где каждый узел содержит значение, соответствующее определенному правилу порядка. Она обеспечивает эффективные операции вставки, извлечения минимального или максимального элемента и обновления значений.

Максимальная куча — это тип кучи, где каждый узел имеет значение, которое больше или равно значения его потомков. Максимальный элемент находится в корне дерева, и каждый узел подчиняется правилу, что его значение больше или равно значению потомков. Дерево представлено в виде массива, где значения элементов соответствуют узлам дерева.

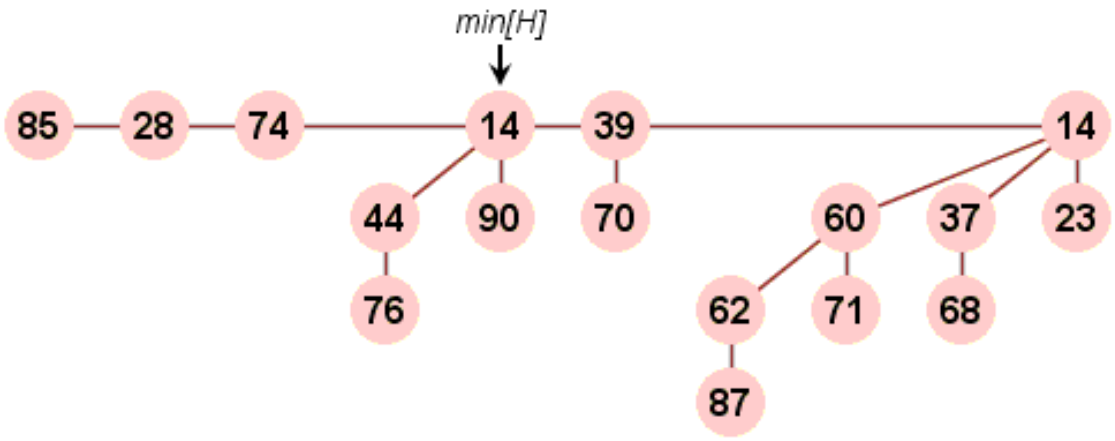


Биномиальная куча — структура данных, реализующая абстрактный тип данных «очередь с приоритетом», которая представляет собой набор биномиальных деревьев с двумя свойствами: Из этих свойств вытекают два следствия. Во-первых, корень каждого из деревьев имеет наименьший ключ среди его вершин. Во-вторых, суммарное количество вершин в биномиальной куче однозначно определяет размеры входящих в неё деревьев.



Фибоначчиева куча — структура данных, представляющая собой набор деревьев, упорядоченных в соответствии со свойством неубывающей пирамиды. Структура является реализацией абстрактного типа данных «Очередь с приоритетом», и замечательна тем, что операции, в которых не требуется удаление, имеют амортизированное время работы, равное O(1) (для двоичной кучи и биномиальной кучи амортизационное время работы равно O(log n)). Кроме стандартных операций INSERT, MIN, фибоначчиева куча позволяет за время O(1) выполнять операцию UNION слияния двух куч.

Фибоначчиевое дерево - это нерегулярное дерево, где каждый узел может иметь произвольное количество потомков.



Описание проделанной работы

В ходе работы были реализованы следующие структуры данных:

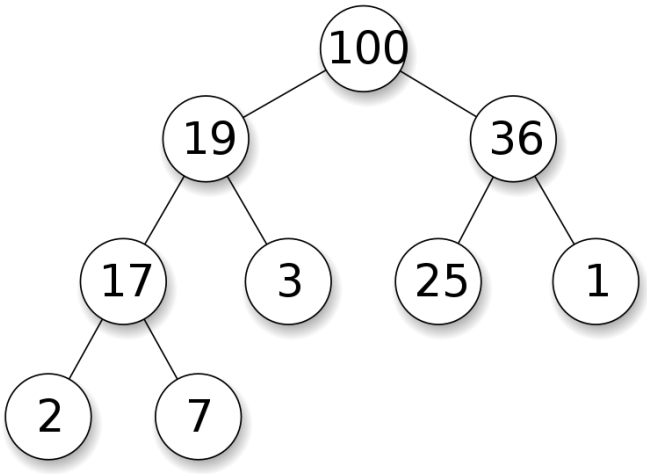
- Max Heap
- Binomial Heap
- Fibonacci Heap

Max Heap

Основные поля:

- *DynamicArray<T> heap;*

  куча представлена в виде массива, где 0 – корневой элемент, i*2 +1 – индекс левого ребёнка элемента с индексом i, i*2 +2 – индекс правого ребёнка элемента с индексом i;



индекс	значение
0	100
1	19
2	36
3	17
4	3
5	25
6	1
7	2
8	7

- *unsigned int size;*

  также хранится неотрицательное число – количество элементов в куче

Разработанные методы:

1. *heapify_down()*

   операция «просеивание вниз»

   пока есть возможность, меняет местами элемент по переданному индексу с ребёнком, хранящим наибольшее значение из 2-ух, если хранимое значение меньше хранимого значения в дочернем элементе.

2. *heapify_up()*

   операция «просеивание вверх»

   пока есть возможность, меняет местами элемент по переданному индексу с его родителем, если хранимое значение больше хранимого значения в родительском элементе .

3. *get_max()*

   операция «получить максимум» должна выполняться за O(1);

   максимальный элемент находится всегда в корне Max Heap, поэтому достаточно обратиться к корневому узлу, чтобы получить максимальное значение в куче.

4. *extract_max()*

   операция «извлечь максимум» должна выполняться за O(log n);

   извлекая из кучи максимальный элемент, необходимо поддерживать основные отличительные черты данной кучи: новый максимум должен оказаться в корне, а сама куча должна сохранить отношение между родителями и детьми (каждый ребёнок должен содержать значение меньшее, чем его родитель).

   Алгоритм:

   1. Запоминаем максимум из корня.
   2. Присваиваем корню значение последнего элемента массива.
   3. Удаляем последний элемент из кучи.

4. Применяем функцию «просеивания вниз» для корневого элемента.
5. Возвращаем сохранённое значение максимума.

5. *insert()*

   операция «вставить» должна выполняться за O(log n);
   добавляя новый элемент также необходимо поддерживать основные отличительные черты данной кучи.

   Алгоритм:

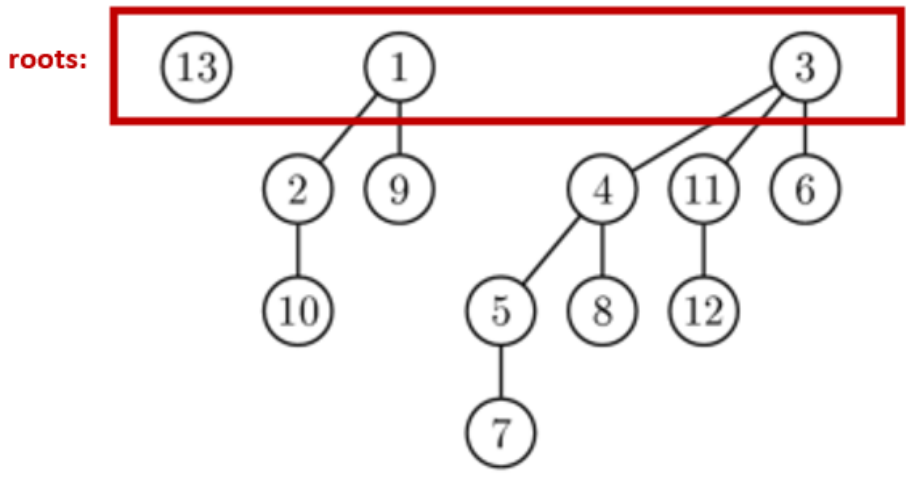
   1. Добавляем новый элемент в конец массива.
   2. Применяем функцию «просеивания вверх» для последнего элемента.

Binomial Heap

Основные поля:

- DynamicArray<Node<T> *> roots

  массив, хранящий указатели на корни деревьев



- Node<T> *minimum

  указатель на корень, содержащий минимальный элемент

- unsigned int size

  также хранится неотрицательное число – количество элементов в куче

Разработанные методы:

1. *collapse()*

   операция, получающая массив корней всех поддеревьев биноминального дерева с корнем, переданным в качестве параметра.
2. *get_min()*

   операция «получить минимум» должна выполняться за O(1);

   на минимальный элемент всегда хранится указатель, поэтому достаточно обратиться по нему, чтобы получить минимальное значение в куче.
3. *extract_min()*

   операция «извлечь минимум» должна выполняться за O(log n);

   извлекая из кучи минимальный элемент, необходимо поддерживать основные отличительные черты данной кучи: новый минимум должен оказаться в корне одного из биноминальных деревьев, а сами кучи должны быть биноминальными и не должно быть деревьев одного ранга; кроме того, каждое дерево должно сохранять отношения между родительскими и дочерними элементами (родительский < любой дочерний).

   Алгоритм:

   1. Запоминаем минимум из хранимого указателя.
   2. Используем метод *collapse()* для дерева, в котором. хранился минимальный элемент, по исполнению которого получаем массив корней деревьев, на которое «распалось» первоначальное.

3. Полученные корни поочерёдно добавляем в *roots*, соединяя деревья, если ранее дерево с подобным рангом уже было в *roots*.
4. Ищем минимум среди корней в *roots*.
5. Возвращаем сохранённое значение минимума.

4. *insert()*

   операция «вставить» должна выполняться за O(log n);
   добавляя новый элемент также необходимо поддерживать основные отличительные черты данной кучи.

   Алгоритм:

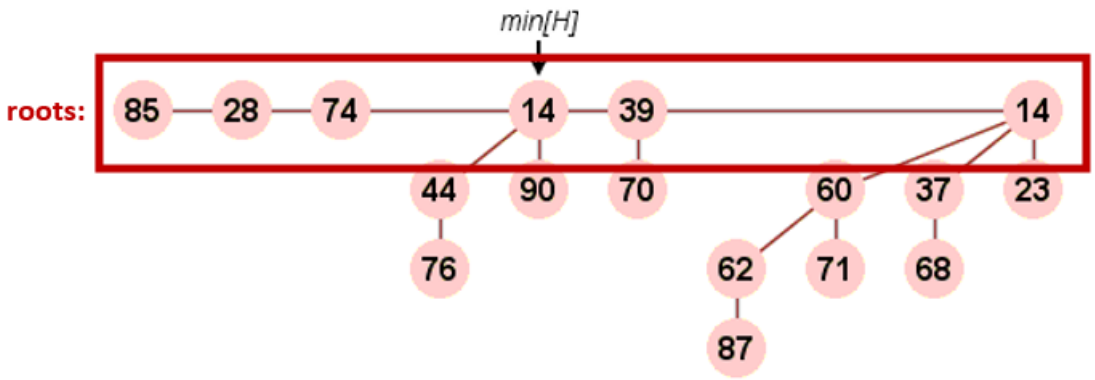
   1. Новый элемент – корень биноминального дерева с рангом равным 0, поэтому добавляем его в *roots*, соединяя деревья, если ранее дерево с подобным рангом уже было.
   2. Если новый элемент меньше того, что в минимуме, то сохраняем указатель на новый элемент в *minimum*.

Fibonacci Heap

Основные поля:

- DynamicArray<Node<T> *> roots

  массив, хранящий указатели на корни деревьев



- Node<T> *minimum

  указатель на корень, содержащий минимальный элемент
- unsigned int size

  также хранится неотрицательное число – количество элементов в куче

Разработанные методы:

1. *consolidate()*

   операция, объединяющая все деревья, корни которых хранятся в roots, оставляя только биноминальные деревья, с уникальными значениями рангов.
2. *get_min()*

   операция «получить минимум» должна выполняться за O(1);

   на минимальный элемент всегда хранится указатель, поэтому достаточно обратиться по нему, чтобы получить минимальное значение в куче.
3. *extract_min()*

   операция «извлечь минимум» должна выполняться за O(log n);

   извлекая из кучи минимальный элемент, для начала нужно переместить все поддеревья в *roots*, а затем необходимо применить функцию *consolidate()* для того, чтобы куча не «размазывалась», вырождаясь в массив корней биноминальных деревьев с рангом ноль.
4. *insert()*

   операция «вставить» должна выполняться за O(1);

   добавляя новый элемент достаточно вставить его в массив *roots*, считая новый элемент корнем биноминального дерева с рангом равным 0, при этом, если новый элемент меньше того, что в минимуме, то сохраняем указатель на новый элемент в *minimum*.
5. *merge()*

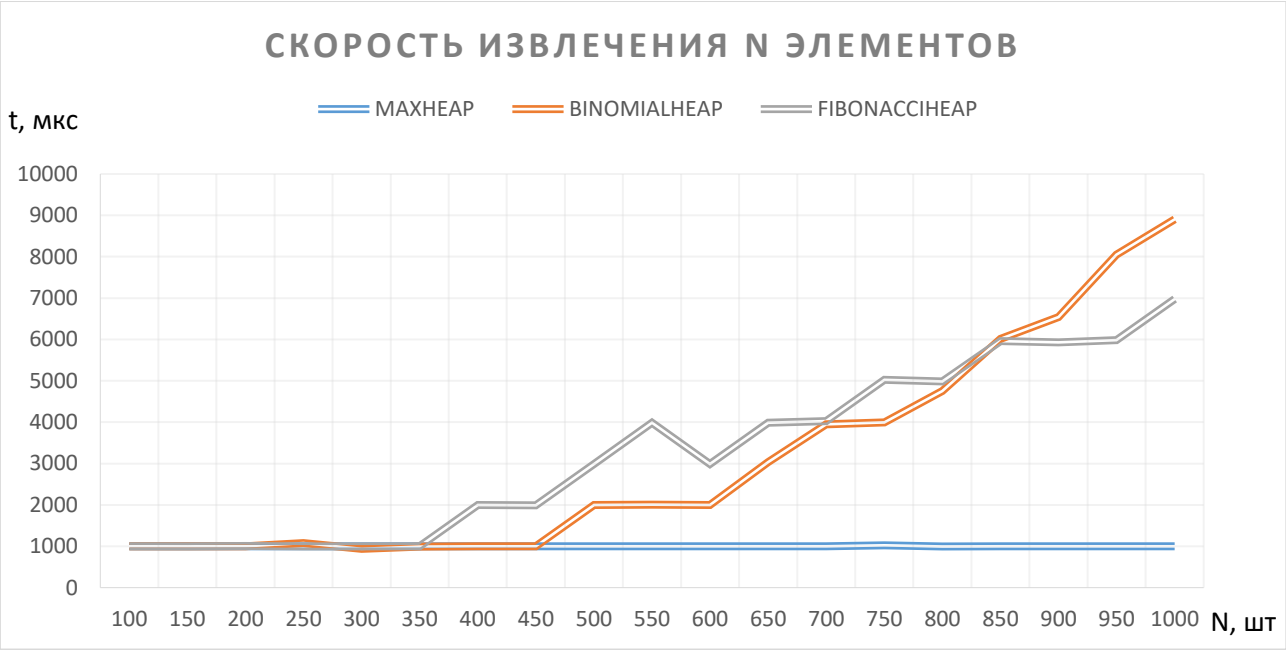
   операция «соединить деревья» должна выполняться за O(n), где n –

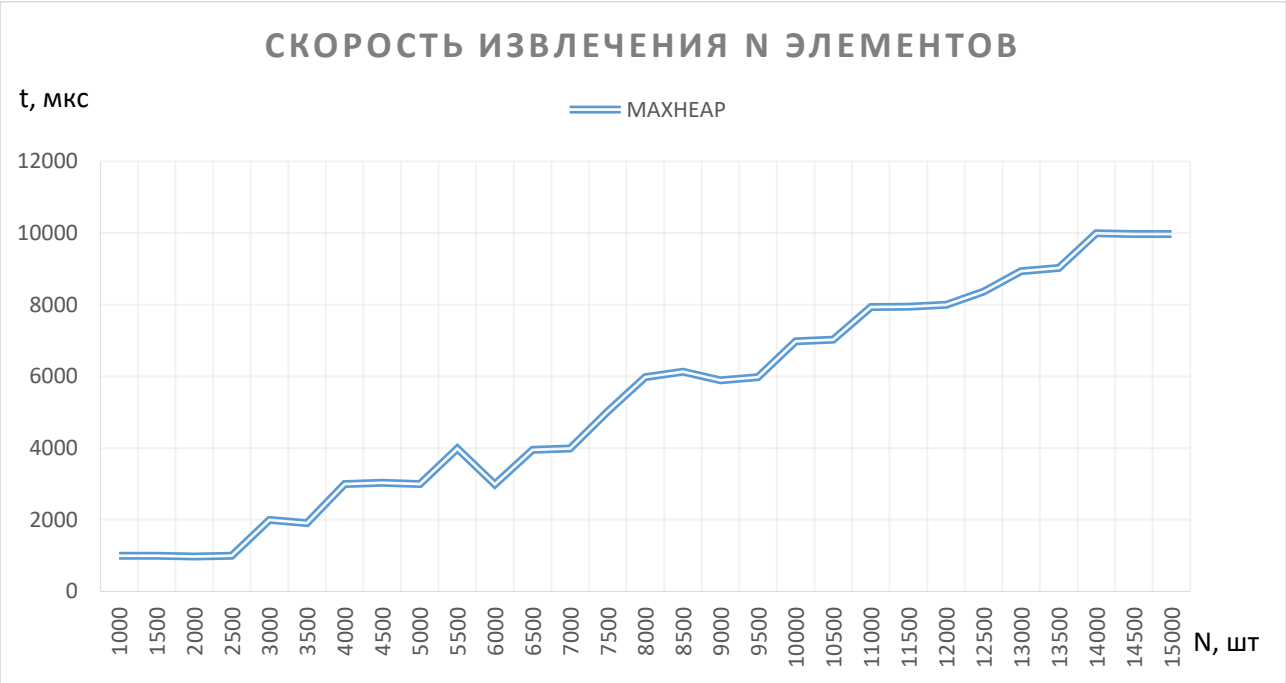
количество деревьев во второй куче;

Все корни деревьев второй кучи переносятся в *roots* первой. После объединения во второй куче не остаётся элементов.

Исследование разработанных структур данных

Были исследованы временные затраты на извлечение всех элементов для каждой из разработанных структур данных при различных размерах.





Заключение

В работе был разработаны Max Heap, Binomial Heap и Fibonacci Heap, а также рассмотрены их особенности. Код содержится в приложении №1.

Max Heap используется для поиска максимального элемента в наборе данных и для сортировки элементов. Binomial Heap используется для эффективного объединения нескольких Binomial Tree в одно дерево. Fibonacci Heap, как и Binomial Heap, используется для эффективной реализации операций вставки, удаления и поиска элементов в наборе данных, а также для решения некоторых задач, таких как нахождение минимального пути в графе, а также реализации очередей с приоритетом.

Список литературы

1. https://en.m.wikipedia.org/wiki/Heap_(data_structure)#
2. https://ru.wikipedia.org/wiki/Биномиальная_куча
3. https://ru.wikipedia.org/wiki/Фибоначчиева_куча
4. https://habr.com/ru/articles/135232/
5. " Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. "Алгоритмы. Вводный курс". – 2-е изд. – Москва : Флинта : Наука, 2010. – 1312 с. – ISBN 978-5-89349-693-2.
6. " Бхаргава, А. "Алгоритмы. Разработка и применение". – 3-е изд., испр. и доп. – Санкт-Петербург : Б. и., 2022. – 512 с.

Приложение №1

MaxHeap.h

```
#ifndef MAXHEAP_H
#define MAXHEAP_H

#include <iostream>
#include "DynamicArray.h"

template<typename T>
class MaxHeap {
private:
    DynamicArray<T> heap;
    unsigned int size;

    void heapify_up(unsigned int index);
    void heapify_down(unsigned int index);

public:
    MaxHeap() : size(0) {};
    MaxHeap(DynamicArray<T> array);
    ~MaxHeap() = default;

    void insert(T value);
    int get_max();
    int extract_max();
    bool is_empty() { return size == 0; }
    unsigned int get_size() { return size; }
    void print();
};

template<typename T>
MaxHeap<T>::MaxHeap(DynamicArray<T> array) {
    heap = array;
    size = array.getSize();
    for (int i = size / 2 - 1; i >= 0; --i) {
        heapify_down(i);
    }
}

template<typename T>
void MaxHeap<T>::heapify_up(unsigned int index) {
    DynamicArray<unsigned int> dop_index;
    bool is_array_empty;
    do {
        while (index > 0) {
            unsigned int parent_index = (index - 1) / 2;
            if (heap[index] > heap[parent_index]) {
                dop_index.push_tail(index);
                std::swap(heap[index], heap[parent_index]);
                index = parent_index;
            } else {
                break;
            }
        }

        is_array_empty = dop_index.getSize() == 0;

        if (!is_array_empty) {
            index = dop_index[dop_index.getSize() - 1];
            dop_index.pop_tail();
        }
```

```
    } while (!is_array_empty);
}

template<typename T>
void MaxHeap<T>::heapify_down(unsigned int index) {
    unsigned int left_child_index;
    unsigned int right_child_index;
    unsigned int largest_index;

    while (true) {
        left_child_index = 2 * index + 1;
        right_child_index = 2 * index + 2;
        largest_index = index;

        if (left_child_index < get_size() && heap[left_child_index] > heap[largest_index]) {
            largest_index = left_child_index;
        }

        if (right_child_index < get_size() && heap[right_child_index] > heap[largest_index]) {
            largest_index = right_child_index;
        }

        if (largest_index != index) {
            std::swap(heap[index], heap[largest_index]);
            index = largest_index;
        } else {
            break;
        }
    }
}


template<typename T>
void MaxHeap<T>::insert(T value) {
    size++;
    heap.push_tail(value);
    heapify_up(get_size() - 1);
}

template<typename T>
int MaxHeap<T>::get_max() {
    if (is_empty()) throw std::runtime_error("Heap is empty!");

    return heap[0];
}

template<typename T>
int MaxHeap<T>::extract_max() {
    if (is_empty()) throw std::runtime_error("Heap is empty!");

    int max_value = heap[0];
    heap[0] = heap[get_size() - 1];
    heap.pop_tail();
    size--;
    heapify_down(0);
    return max_value;
}

template<typename T>
void MaxHeap<T>::print() {
    std::cout << "\nheap: ";
    for (int i = 0; i < get_size(); ++i) {
```

```
        std::cout << heap[i] << ' ';
    }
    std::cout << '\n';
}

#endif
```

BinomialHeap.h

```
#ifndef BINOMIALHEAP_H
#define BINOMIALHEAP_H

#include <iostream>
#include <cmath>

#include "DynamicArray.h"
#include "Node.h"

template<typename T>
class BinomialHeap {
private:
    DynamicArray<Node<T> *> roots;
    Node<T> *minimum;
    unsigned int size;

    Node<T> *merge_trees(Node<T> *a, Node<T> *b);
    void delete_tree(Node<T> *node);
    DynamicArray<Node<T> *> collapse(Node<T> *node);

public:
    BinomialHeap() : roots(), minimum(nullptr), size(0) {}
    ~BinomialHeap();

    void insert(T data);
    T get_min();
    T extract_min();
    void print_roots();
    unsigned int get_size() { return size; };
};

template<typename T>
Node<T> *BinomialHeap<T>::merge_trees(Node<T> *a, Node<T> *b) {
    if (a == nullptr && b == nullptr) throw std::invalid_argument("All nodes are nullptr!");
    if (a == nullptr) return b;
    if (b == nullptr) return a;

    if (a->data > b->data) {
        std::swap(a, b);
    }

    int c_size = a->children.get_size();

    if (c_size == 0) {
        a->children.push_head(b);
    } else {
        a->children.push_next(b, c_size - 1);
    }

    return a;
}

template<typename T>
void BinomialHeap<T>::delete_tree(Node<T> *node) {
    ListNode<Node<T> *> *child = (node->children).get_head();
```

```
    for (int i = 0; i < (node->children).get_size(); ++i) {
        delete_tree(child->data);
        child = child->get_next();
    }
    delete node;
}

template<typename T>
BinomialHeap<T>::~BinomialHeap() {
    for (int i = 0; i < roots.getSize(); ++i) {
        if (roots[i] != nullptr) {
            delete_tree(roots[i]);
        }
    }
}

template<typename T>
void BinomialHeap<T>::insert(T data) {
    auto *new_root = new Node<T>(data);

    if (size == 0) {
        minimum = new_root;
        roots.push_tail(new_root);
        size++;
        return;
    }

    if (minimum->data >= data)
        minimum = new_root;

    size++;
    int temp = ((int) (10 * std::log2(size))) % 10;
    if (temp == 0) roots.push_tail(nullptr);

    for (int i = 0; i < roots.getSize(); ++i) {
        if (roots[i] == nullptr) {
            roots[i] = new_root;
            break;
        }
        new_root = merge_trees(new_root, roots[i]);
        roots[i] = nullptr;
    }
}

template<typename T>
T BinomialHeap<T>::get_min() {
    if (minimum == nullptr) throw std::runtime_error("Heap is empty!");

    return minimum->data;
}

template<typename T>
DynamicArray<Node<T> *> BinomialHeap<T>::collapse(Node<T> *node) {
    DynamicArray<Node<T> *> nodes;
    int c_size = (node->children).get_size();

    if (c_size == 0) {
        nodes.push_tail(nullptr);
        return nodes;
    }

    for (int i = 0; i < c_size; ++i) {
```

```
        nodes.push_tail((node->children).pop_head()->data);
    }
    return nodes;
}

template<typename T>
T BinomialHeap<T>::extract_min() {
    if (minimum == nullptr) throw std::runtime_error("Heap is empty!");

    T min_value = minimum->data;

    DynamicArray<Node<T> *> new_roots = collapse(minimum);

    if (new_roots[0] == nullptr) {
        roots[0] = nullptr;
    } else {
        roots[new_roots.getSize()] = nullptr;
        int temp_index;
        Node<T> *temp_root = nullptr;
        for (int j = 0; j < new_roots.getSize(); ++j) {
            if (roots[j] == nullptr) {
                roots[j] = new_roots[j];
            } else {
                temp_index = j;
                temp_root = roots[j];
                for (; j < new_roots.getSize(); ++j) {
                    temp_root = merge_trees(temp_root, new_roots[j]);
                    new_roots[j] = nullptr;
                }
                break;
            }
        }

        if (temp_root != nullptr) {
            roots[temp_index] = nullptr;
            roots[new_roots.getSize()] = temp_root;
        }
    }

    unsigned int i = 0;
    while (i < roots.getSize() && roots[i] == nullptr) {
        i++;
    }

    size--;
    if (i == roots.getSize()) {
        minimum = nullptr;
        return min_value;
    }

    minimum = roots[i];
    for (; i < roots.getSize(); i++) {
        if (roots[i] != nullptr && roots[i]->data <= minimum->data) {
            minimum = roots[i];
        }
    }

    return min_value;
}

template<typename T>
void BinomialHeap<T>::print_roots() {
```

```
    for (unsigned int i = 0; i < roots.getSize(); ++i) {
        std::cout << "\nroot with rank = " << i << ": ";
        if (roots[i] != nullptr) std::cout << roots[i]->data;
    }
    std::cout << '\n';
}

#endif
```

FibonacciHeap.h

```
#ifndef FIBONACCIHEAP_H
#define FIBONACCIHEAP_H

#include <iostream>

#include "Node.h"
#include "DynamicArray.h"

template<typename T>
class FibonacciHeap {
private:
    DynamicArray<Node<T> *> roots;
    unsigned int size;
    Node<T> *minimum;

    Node<T> *merge_trees(Node<T> *a, Node<T> *b);
    void delete_tree(Node<T> *node);

public:
    FibonacciHeap() : roots(), size(0), minimum(nullptr) {};
    ~FibonacciHeap();

    void merge(FibonacciHeap<T> &heap);
    void insert(T data);
    T get_min() { return minimum->data; }
    T extract_min();
    void print_roots();
    unsigned int get_size() { return size; }
    void roots_sort();

    void consolidate();
};

template<typename T>
Node<T> *FibonacciHeap<T>::merge_trees(Node<T> *a, Node<T> *b) {
    if (a == nullptr && b == nullptr) throw std::invalid_argument("All nodes are nullptr!");
    if (a == nullptr) return b;
    if (b == nullptr) return a;

    if (a->data > b->data) {
        std::swap(a, b);
    }
    int c_size = a->children.get_size();

    if (c_size == 0) {
        a->children.push_head(b);
    } else {
        a->children.push_next(b, c_size - 1);
    }

    return a;
}
```

```
template<typename T>
void FibonacciHeap<T>::delete_tree(Node<T> *node) {
    ListNode<Node<T> *> *child = (node->children).get_head();
    for (int i = 0; i < (node->children).get_size(); ++i) {
        delete_tree(child->data);
        child = child->get_next();
    }
    delete node;
}

template<typename T>
FibonacciHeap<T>::~FibonacciHeap() {
    for (int i = 0; i < roots.getSize(); ++i) {
        if (roots[i] != nullptr) {
            delete_tree(roots[i]);
        }
    }
}

template<typename T>
void FibonacciHeap<T>::merge(FibonacciHeap<T> &heap) {
    for (int i = 0; i < heap.roots.getSize(); ++i) {
        roots.push_tail(heap.roots[i]);
    }

    if (minimum->data > heap.minimum->data) minimum = heap.minimum;
    size += heap.size;

    heap.size = 0;
    heap.roots = DynamicArray<Node<T> *>();
    heap.minimum = nullptr;
}

template<typename T>
void FibonacciHeap<T>::insert(T data) {
    auto *temp = new Node<T>(data);

    if (roots.getSize() == 0) minimum = temp;

    roots.push_tail(temp);

    if (minimum->data > data) minimum = temp;
    size++;
}

template<typename T>
T FibonacciHeap<T>::extract_min() {
    if (minimum == nullptr) throw std::runtime_error("Heap is empty!");

    T min_value = minimum->data;

    if ((minimum->children).get_size() != 0) {
        LinkedList<Node<T> *> children_nodes = minimum->children;

        do {
            Node<T> *child_node = children_nodes.pop_head()->data;
            roots.push_tail(child_node);
        } while (!children_nodes.is_empty());
    }

    std::swap(minimum->data, (roots[roots.getSize() - 1])->data);
    std::swap(minimum->children, (roots[roots.getSize() - 1])->children);
```

```
    roots.pop_tail();
    size--;

    roots_sort();

    if (size != 0) {
        consolidate();

        minimum = roots[0];
        for (int i = 0; i < roots.getSize(); ++i) {
            if (roots[i]->data < minimum->data) {
                minimum = roots[i];
            }
        }
    } else {
        minimum = nullptr;
    }

    return min_value;
}

template<typename T>
void FibonacciHeap<T>::consolidate() {
    DynamicArray<Node<T> *> res_arr;
    Node<T> *temp;

    for (int i = roots.getSize() - 1; i > 0; i--) {
        if (roots[i - 1]->children.get_size() == roots[i]->children.get_size()) {
            temp = merge_trees(roots[i - 1], roots[i]);
            roots[i - 1] = temp;
            roots.pop_tail();
            roots_sort();
        } else {
            res_arr.push_tail(roots[i]);
            roots.pop_tail();
        }
    }
    res_arr.push_tail(roots[0]);
    roots = res_arr;
}

template<typename T>
void FibonacciHeap<T>::print_roots() {
    roots_sort();
    int c_size;
    for (unsigned int i = 0; i < roots.getSize(); ++i) {
        c_size = roots[i]->children.get_size();
        std::cout << "\nroots with rank = " << c_size << ": ";
        std::cout << roots[i]->data;
        while (i + 1 < roots.getSize() && c_size == roots[i + 1]->children.get_size()) {
            i++;
            std::cout << ", " << roots[i]->data;
        }
    }
    std::cout << '\n';
}

template<typename T>
void FibonacciHeap<T>::roots_sort() {
    Node<T> *temp;
    for (int i = 0; i < roots.getSize(); ++i) {
        bool flag = true;
```

```
        for (int j = 0; j < roots.getSize() - (i + 1); j++) {
            if (roots[j]->children.get_size() < roots[j + 1]->children.get_size()) {
                flag = false;
                temp = roots[j];
                roots[j] = roots[j + 1];
                roots[j + 1] = temp;
            }
        }
        if (flag) {
            break;
        }
    }
}


#endif
```

Node.h

```
#ifndef NODE_H
#define NODE_H

#include "LinkedList.h"

template<typename T>
struct Node {
    T data;
    LinkedList<Node<T> *> children;

    Node(const T &data) : data(data), children() {};
};

#endif
```

DynamicArray.h

```
#ifndef DYNAMIC_ARRAY_H
#define DYNAMIC_ARRAY_H

template<class T>
class DynamicArray {
private:
    T *data;
    size_t size;
    size_t capacity;

public:
    DynamicArray();;
    ~DynamicArray();
    T &operator[](size_t index);
    DynamicArray<T> &operator=(const DynamicArray &newArr);

    size_t getSize();
    void expand(int k);
    void push_tail(const T &item);
    void print(std::ostream &out) const;
    void pop_tail();
};

template<typename T>
DynamicArray<T>::DynamicArray() {
    data = nullptr;
    size = 0;
    capacity = 0;
}
```

```
template<typename T>
DynamicArray<T>::~DynamicArray() {
    if (data != nullptr) delete[] data;
}

template<typename T>
T &DynamicArray<T>::operator[](const size_t index) {
    try {
        if (index >= size) throw std::out_of_range("Invalid index!1");
    }
    catch (std::exception &e) {
        std::cerr << e.what();
        exit(-1);
    }
    return data[index];
}

template<typename T>
DynamicArray<T> &DynamicArray<T>::operator=(const DynamicArray &newArr) {
    if (this == &newArr) return *this;

    if (data != nullptr) delete[] data;

    size = newArr.size;
    capacity = newArr.capacity;
    data = new T[capacity];
    for (int i = 0; i < size; ++i) {
        data[i] = newArr.data[i];
    }
    return *this;
}

template<class T>
size_t DynamicArray<T>::getSize() {
    return size;
}

template<class T>
void DynamicArray<T>::expand(int k) {
    T *temp = new T[size];
    for (int i = 0; i < size; ++i)
        temp[i] = data[i];
    capacity += k;
    data = new T[capacity];
    for (int i = 0; i < size; ++i) {
        data[i] = temp[i];
    }
    delete[] temp;
}

template<class T>
void DynamicArray<T>::push_tail(const T &item) {
    if (size == capacity) {
        expand(capacity + 1);
    }
    data[size] = item;
    size++;
}

template<class T>
void DynamicArray<T>::print(std::ostream &out) const {
```

```
    for (size_t i = 0; i < size; i++)
        out << data[i] << ' ';
}

template<class T>
void DynamicArray<T>::pop_tail() {
    if (size != 0) size--;
    else throw std::out_of_range("Invalid index!2");
}

#endif
```

LinkedList.h

```
#ifndef LINKEDLIST_H
#define LINKEDLIST_H

template<typename T>
class ListNode {
private:
    ListNode *next;
public:
    T data;

    ListNode(T data) : data(data), next(nullptr) {}
    ~ListNode() = default;

    ListNode *get_next() { return next; }
    void set_next(ListNode *new_next) { next = new_next; }
    void push_next(T item);
};

template<typename T>
void ListNode<T>::push_next(T item) {
    auto *temp = new ListNode(item);
    if (next == nullptr) {
        next = temp;
    } else {
        ListNode *temp2 = next;
        temp->next = temp2;
        next = temp;
    }
}

template<typename T>
class LinkedList {

private:
    unsigned int size;

public:
    ListNode<T> *head;

    LinkedList() : head(nullptr), size(0) {}
    ~LinkedList() { clear(); }

    void push_head(T data);
    ListNode<T> *pop_head();
    int get_size() const { return size; }
    bool is_empty() const { return size == 0; }
    void clear();
    ListNode<T> *get_head() { return head; }
    void push_next(T data, unsigned int index);
};
```

```
template<typename T>
void LinkedList<T>::push_head(T data) {
    auto *temp = new ListNode<T>(data);
    if (head == nullptr) {
        head = temp;
    } else {
        temp->set_next(head);
        head = temp;
    }
    size++;
}

template<typename T>
ListNode<T> *LinkedList<T>::pop_head() {
    if (head == nullptr) throw std::underflow_error("List is empty!");

    ListNode<T> *temp = head;
    if (head->get_next() == nullptr) {
        head = nullptr;
    } else {
        head = head->get_next();
    }
    size--;
    return temp;
}

template<typename T>
void LinkedList<T>::clear() {
    while (head != nullptr) {
        ListNode<T> *temp = head;
        head = head->get_next();
        delete temp;
    }
    size = 0;
}

template<typename T>
void LinkedList<T>::push_next(T data, unsigned int index) {
    if (index >= size) throw std::out_of_range("Invalid index!");

    ListNode<T> *temp = head;
    for (int i = 0; i < index; ++i) {
        temp = temp->get_next();
    }
    size++;
    temp->push_next(data);
}

#endif
```