Оглавление

Глава 1. Структуры данных, отвечающие интерфейсу очереди с приоритетами …..……..3

1.1 Кучеобразные структуры данных……..…….4

1.1.2 Двоичная куча……..……..5

1.1.3 Биномиальная куча……..……..6

1.1.4 Фибоначчиева куча……7

1.1.5 Тонкая куча……8

1.2 Вывод к главе 1……9

Глава 2. Реализация структуры данных «Тонкая куча»……...10

Заключение……...13

Список литературы……..14

Приложения……..15

Глава 1. Структуры данных, отвечающие интерфейсу очереди с приоритетами

Очередь с приоритетами – это структура данных, хранящая элементы с определенным приоритетом и обеспечивающая быстрый доступ к элементам с наименьшим или наибольшим приоритетом.

Очередь с приоритетами имеет две основные операции: *Insert* и *Delete*. *Insert* добавляет в очередь новый элемент, *Delete* удаляет из очереди элемент с наименьшим или наибольшим приоритетом.

Очереди с приоритетами используются для планирования заданий, в таких алгоритмах как: алгоритм Дейкстры. алгоритм Прима, алгоритм Хаффмана, дискретно-событийное моделирование и так далее.

1.1 Кучеобразные структуры данных

Чаще всего очереди с приоритетами представляются при помощи корневого дерева или набора корневых деревьев. Кучеобразными называют структуры данных типа “дерево”, каждому узлу которого ставят во взаимно однозначное соответствие элемент рассматриваемого множества. При этом приоритет, приписанный каждому узлу, должен быть больше(меньше) приоритета, приписанного его потомкам.

Чаще всего очереди с приоритетами представляются при помощи корневого дерева, каждому узлу которого ставятся во взаимно однозначное соответствие элементы рассматриваемого множества. При этом приоритет, приписанный каждому узлу, должен быть больше(меньше) приоритета, приписанного его потомкам. Если корню приписан элемент с наибольшим приоритетом, такое дерево называют максимальной (минимальной) кучей.

Основными операциями над минимальными кучами являются:

- Поиск элемента с максимальным приоритетом
- Удаление элемента с максимальным приоритетом
- Добавление нового элемента
- Слияние двух куч
- Увеличение или уменьшение приоритета элемента

1.1.2 Двоичная куча

Максимальная двоичная куча представляет собой кучеобразное полное бинарное дерево. То есть дерево, удовлетворяющее следующим условиям:

- Приоритет каждого узла больше приоритета его потомков
- Каждая вершина имеет не более двух потомков
- Заполнение слоев идет строго сверху вниз и слева направо

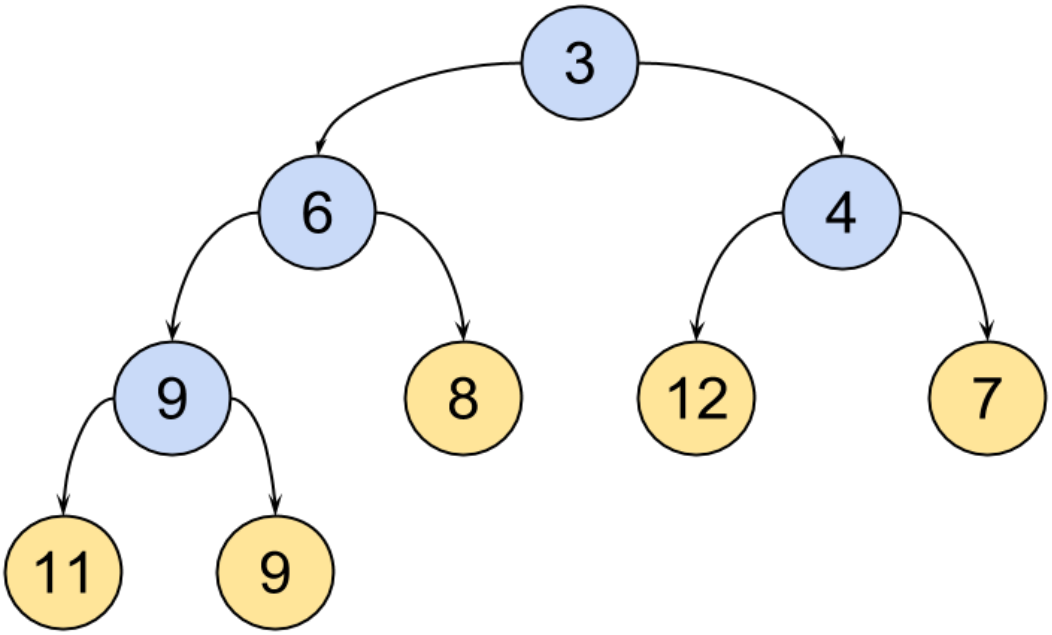

рис. 1. Минимальная двоичная куча

Двоичная куча обычно хранится при помощи одномерного массива, в котором левый потомок узла с индексом i имеет индекс 2*i + 1, а правый индекс 2*i + 2. А корнем является элемент с индексом 0.

Время работы операций *findmax* – $O(1)$, *delete-max*, *insert*, *decreaseKey* – $O(\log n)$, *merge* – $O(n + m)$ где n и m количество элементов двух сливаемых куч.

1.1.3 Биномиальная куча

Биномиальное дерево ранга 0 – дерево, состоящее из одного узла. Биномиальное дерево ранга k – это дерево, состоящее из двух биномиальных деревьев с рангом k – 1, связанных таким образом, что корень одного из них является предком узла второго дерева.

Биномиальная куча является множеством кучеобразных биномиальных деревьев, в котором для любого неотрицательного k может существовать не более одного биномиального дерева ранга k. Корни биномиальных деревьев, из которых состоит биномиальная куча, составляют так называемый корневой односвязный список. Каждый узел биномиального дерева представляется набором полей: *key, parent, child, sibling, degree*. Где *key* – ключ (приоритет элемента, приписанного этому узлу), *parent* – указатель на родителя, *child* – указатель на самого левого предка, *sibling* – указатель на правого брата, *degree* – ранг узла. Доступ к куче осуществляется при помощи указателя на самый левый корень в корневом списке.

Поскольку количество потомков у узла определено неоднозначно, потомки образуют односвязный список, началом которого является самый левый потомок.

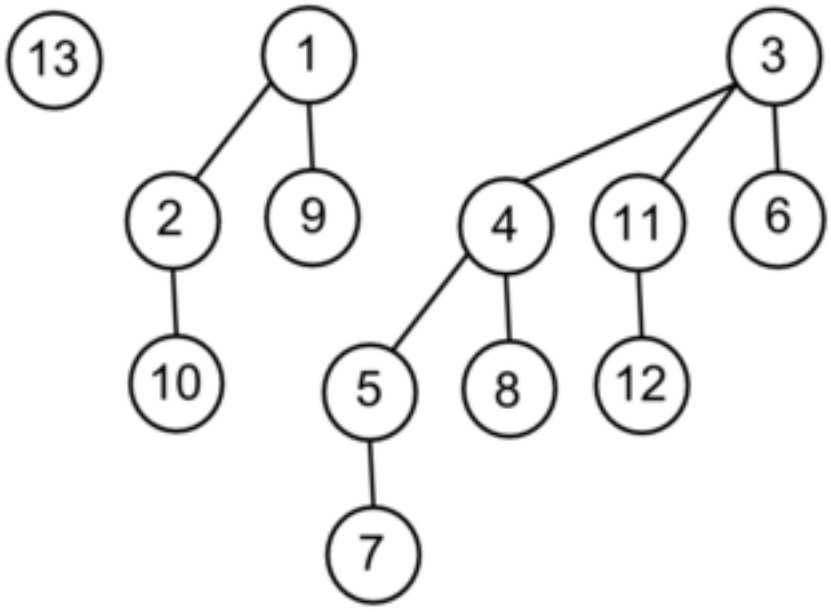

рис. 2. Биномиальные деревья рангов 0, 2, 3 с минимумом в корне

Время работы операций *insert, find-min, merge, decreaseKey, delete, delete-min* в худшем случае $O(\log n)$.

1.1.4 Фибоначчиева куча

Фибоначчиева куча представляет собой набор кучеобразных деревьев, корни которых составляют двусвязный корневой список. В отличии от биномиальной кучи, на деревья, из которых состоит фибоначчиева куча, не наложены никакие строгие ограничения по форме. Каждый узел деревьев, составляющих фибоначчиеву кучу, представляется набором полей: *key, left, right, parent, child, degree, mark*. Где *key* – ключ (приоритет), *left* – указатель на левого соседа в списке, *right* – указатель на правого соседа в списке, *parent* – указатель на родителя, *child* – указатель на одного из предков, *degree* – ранг узла (количество потомков), *mark* – булево значение, истинна если узел потерял ребенка после того, как он в последний раз сделался чьим-либо потомком. Доступ к куче осуществляется при помощи указателя на узел с максимальным (минимальным) приоритетом.

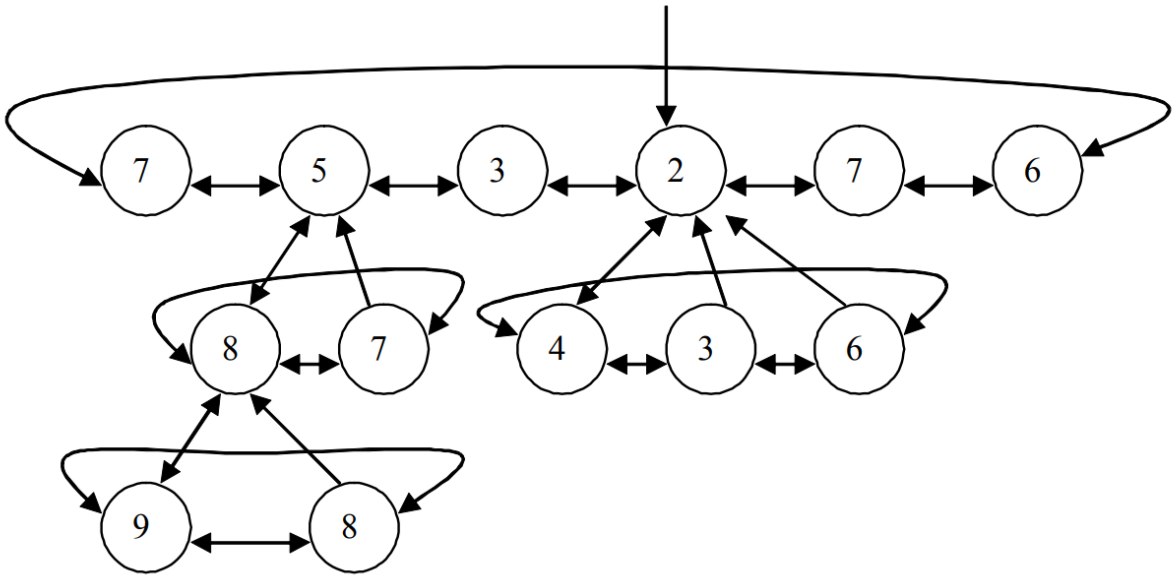

рис. 3. Фибоначчиева куча

Время работы операций *create, insert, find-min, merge, decreaseKey* - *O*(1), а операций *delete, delete-min* - *O*(log n).

1.1.5 Тонкая куча

Тонкое дерево ранга k – это дерево, которое может быть получено из биномиального дерева ранга k, путем удаления самого левого предка у узлов, не являющихся корнем или листом.

Более формально тонкое дерево – это упорядоченное дерево, каждый узел которого имеет положительный или равный 0 ранг, и удовлетворяет следующим свойствам:

- узел с рангом r, либо имеет r потомков с рангами r - 1, r - 2, …, 0 (такой узел называют толстым), либо имеет r - 1 потомков с рангами r -2, r - 3, …, 0 (такой узел называют тонким)
- корень является толстым узлом

Заметим, что если соединить два тонких дерева, корни которых имеют одинаковый ранг r, сделав один из корней левым потомком другого, то получится тонкое дерево с корнем ранга r + 1.

Тонкая куча – это набор кучеобразных тонких деревьев, корни которых составляют односвязный корневой список. Каждый узел тонкого дерева представляется набором полей: *key*, *rank*, *child*, *left*, *right*. Где *key* – ключ (приоритет), *rank* – ранг узла, *child* – указатель на самого левого предка, *left* - указатель на левого брата, либо на родителя, если текущий узел самый левый, либо *null*, если текущий узел является корнем, *right* - указатель на правого брата, либо на следующий корень, если текущий узел является корнем. Доступ к куче осуществляется при помощи указателей *first* и *last*. Где *first* – указатель на самый левый корень в корневом списке, который также является корнем с максимальным (минимальным) приоритетом, *last* – указатель на последний корень в корневом списке.

Рассматриваемые в данной работе тонкие кучи впервые были предложены М.Фредманом и Х.Капланом как альтернатива фибоначчиевым кучам. По структуре тонкая куча во многом похожа на фибоначчиеву кучу и имеет такие же асимптотические оценки. Время работы операций *create*, *insert*, *find-min*, *merge*, *decreaseKey* – $O(1)$, а операций *delete*, *delete-min* – $O(\log n)$. Тонкая куча, при одинаковых асимптотических оценках, имеет меньшее количество констант по сравнению с фибоначчиевой кучей, что на практике должно ускорить работу и уменьшить объем затраченной памяти.

1.2 Вывод к главе 1

Долгое время фибоначчиевы кучи считались лучшими по производительности, но, в связи с большим количеством констант, фибоначчиева куча имеет преимущество над другими структурами данных только на данных огромных размеров. Тонкая куча выигрывает за счёт меньшего количества констант. Также тонкие кучи имеют преимущество над биномиальными и двоичными кучами, если число операций удаления значительно меньше остальных операций, например, при работе с графами, имеющими много ребер. В связи с вышесказанным было принято решение реализовать структуру данных тонкая куча.

Глава 2. Реализация структуры данных «Тонкая куча» (с максимумом в корне)

```
class Node
{
private:
   T priority; //приоритет
   int rank; //ранг
   Node* child; //указатель на самого левого ребенка
   Node* right; //указатель на правого брата, либо на следующий корень, если текущий узел является корнем
   Node* left; // указатель на левого брата , либо на родителя, если текущий узел самый левый, либо null, если текущий узел корень
```

Листинг 1. Структура узла

Набор полей: приоритет, ранг (ранг соответствующего узла в биномиальном дереве) и указатели на самого левого ребенка, правого брата (либо на следующий корень в корневом списке, если текущий узел - корень) и левого брата (либо родителя, если узел самый левый из детей).

Функция *isThin* возвращает булево значение, 1 если корень тонкий, 0 если толстый. Также написаны конструктор по умолчанию и конструктор принимающий приоритет.

```
class thinheap
{
public:
   Node<T>* first; //указатель на корень с максимальным приоритетом
   Node<T>* last; //указатель на последний корень

   thinheap() { first = last = NULL; }
   ~thinheap() { first = last = NULL; }

   Node<T>* getMax() { return first; }
```

Листинг 2. Структура тонкой кучи

Тонкая куча состоит из указателей на первый и последний узлы в корневом списке. Функция *getMax* возвращает указатель на первый узел в корневом списке.

Функция *insert*, принимающая указатель на узел *newnode_*, добавляет в корневой список тонкое дерево, корнем которого является узел *newnode_*. Если приоритет добавляемого узла больше максимального, узел добавляется в начало и указатель на максимальный элемент обновляется. Если же приоритет меньше, узел добавляется в конец.

Функция insert, принимающая приоритет, создает новый узел с этим приоритетом и так же добавляет его в корневой список.

Функция merge принимает тонкую кучу *heap_* и сливает корневые списки двух тонких куч, после чего обновляет указатель на первый или последний элемент корневого списка.

Функция *extractMax* удаляет из тонкой кучи узел с максимальным приоритетом, и возвращает указатель на этот узел. Сначала узел с максимальным приоритетом удаляется из корневого списка, после чего понижается ранг всех его “тонких” детей. Затем все дети добавляются в корневой список, и создается массив из указателей на узлы, на i-том месте которого будет хранится тонкое дерево ранга i. После чего массив заполняется, а деревья одного ранга сливаются, пока не в массиве не останутся тонкие деревья различных рангов. Переменная *Int max_range* соответствует максимальному рангу тонких деревьев из массива. Переменная *max_range* нужна для того, чтобы ограничить перебор массива. Далее обнуляются указатели на первый и последний элементы корневого списка, и затем все элементы массива добавляются в кучу, посредством функции *insert*, в связи с этим, после выполнения функции указатель *first* будет соответствовать узлу с максимальным приоритетом.

```
bool sibling_violation(Node<T>* node_)
  {
    if (node_->getRight() == NULL && node_->getRank() == 1)return 1;
    if (node_->getRight() != NULL && node_->getRight()->getRank() + 2 == node_-
>getRank())return 1;
    return 0;
  }

  bool child_violation(Node<T>* node_)
  {
    if (node_->getRank() == 2 && node_->getChild() == NULL)return 1;
    if (node_->getChild() != NULL && node_->getChild()->getRank() + 3 == node_-
>getRank())return 1;
    return 0;
  }
```

Листинг 3. Вспомогательные функции *sibling_violation* и *child_violation*

Функции *sibling_violation* и *child_violation* возвращают булевы значения. При изменении приоритета могут возникнуть два типа нарушений целостности тонкой кучи. Так называемые “Братские” и “Родительские” нарушения. Функция *sibling_violation* возвращает 1 если узел ранг узла *node_* на 2 больше ранга правого брата, или если *node_* не имеет правого брата, и ранг *node_* равен 1. 0 возвращается в том случае, если узел не является “узлом локализации нарушения”, то есть не нарушает целостность тонкой кучи.

Функция *child_violation* возвращает 1, если узел *node_* не имеет детей, и его ранг равен 2, или если ранг узла *node_* на 3 больше ранга его самого левого ребенка. Аналогично функции *sibling_violation*, 0 возвращается если узел не нарушает целостность тонкой кучи.

Функция *changePriority* получает на вход указатель на узел, новый приоритет для этого узла, и возвращает указатель на этот узел после восстановления (если нужно) целостности тонкой кучи. Указатель *left* отвечает возможному, после смены приоритета, узлу локализации нарушения. Функция работает до тех пор, пока не будут устранены все “Братские” и “Родительские” нарушения. При “Братском” нарушении, если узел локализации нарушения является тонким, его ранг понижается на 1, после чего узлом локализации нарушения будет его родитель, либо левый брат. При “Родительском” нарушении тонкое поддерево, корнем которого является узел локализации нарушения, добавляется в корневой список, и, если узел локализации нарушения не был самым левым из братьев, новым узлом локализации нарушения будет его левый брат. Если же узел локализации нарушения был самым старшим из братьев и его родитель не был тонким узлом, родитель становится тонким узлом, но, если родитель был тонким, он становится новым узлом локализации нарушения.

Вспомогательная функция *heaptovector* помещает все значения из кучи в вектор, используя функцию *extractMax*.

Функция *QuantileSearch* получает на вход значение val, и возвращает val-квантиль, то есть такое значение, что примерно val*100 процентов всех элементов в куче будет меньше этого значения.

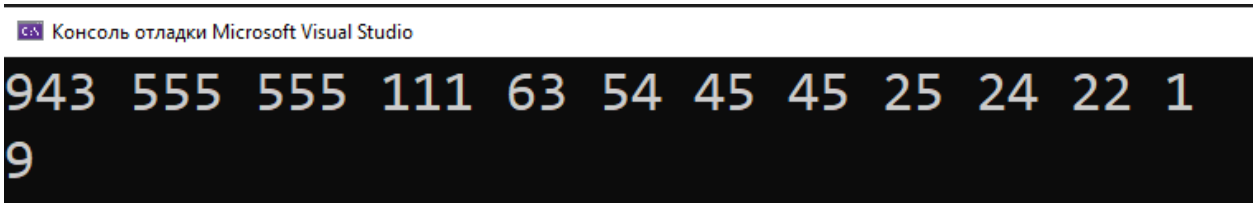

рис. 4. Результат слияния двух куч и найденный 0.4 квантиль

Заключение

Имея константную сложность для всех операций, кроме удаления элементов, тонкие кучи могут относительно эффективно использоваться для работы с большими наборами данных, когда удаление элемента не является частой операцией, например, при работе с достаточно крупными графами. Имея меньшее количества констант, тонкая куча по сути является улучшенной версией фибоначчиевой кучи. Учитывая все вышесказанное, тонкая куча несомненно является структурой данных, отвечающей интерфейсу очереди с приоритетами, на которую следует обратить внимание.

Список литературы

1. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт; пер. с англ. –Санкт-Петербург, «Невский диалект», 2001 – 352 с.

2. Шилдт, Г. Полный справочник по C++, 4-е издание / Г. Шилдт; пер. сангл. – Москва: Вильямс, 2006 – 800 с.

3. Топп, У. Структуры данных в C++ / У. Топп, У. Форд; пер. с англ. –Москва: ЗАО «Издательство Бином», 1999 – 816 с.

4. Каплан Х., Тарьян А. Р.., Thin Heaps, Thick Heaps // ACM Transactions on Algorithms. — 2008. — Т.4. — №1. — С. 1—14. — ISSN: 1549-6325

5. Гербер Р. Оптимизация ПО. Сборник рецептов / Р. Гербер, К. Тиан, К. Смит, А. Бик; пер. с англ. – Санкт-Петербург: Питер, 2010 – 352 с.

Приложения

```
using namespace std;
#define NULL nullptr
#include <iostream>
template <class T>
class Node
{
private:

    T priority; //приоритет
    int rank; //ранг
    Node* child; //указатель на самого левого ребенка
    Node* right; //указатель на правого брата, либо на следующий корень, если текущий узел является корнем
    Node* left; // указатель на левого брата , либо на родителя, если текущий узел самый левый, либо null, если текущий узел корень

public:

    Node() { rank = 0; child = right = left = NULL; }
    Node(T priority_) { priority = priority_; rank = 0; child = right = left = NULL; }
    ~Node() { child = right = left = NULL; rank = -1; }
    T getPriority() { return priority; }
    int getRank() { return rank; }
    Node* getChild() { return child; }
    Node* getRight() { return right; }
    Node* getLeft() { return left; }

    T setPriority(T priority_) { priority = priority_; return priority_; }
    int setRank(int rank_) { rank = rank_; return rank_; }
    Node* setChild(Node* child_) { child = child_; return child; }
    Node* setRight(Node* right_) { right = right_; return right; }
    Node* setLeft(Node* left_) { left = left_; return left; }

    Node* operator =(Node* const n)
    {
        *this->child = n->child();
        *this->right = n->right();
        *this->left = n->left();
        *this->priority = n->priority();
        *this->rank = n->rank();
    }

    bool isThin()
    {
        if (getRank() == 0)return 0;
        if (getChild() == NULL)return 1;
        return (getRank() - getChild()->getRank() == 2 ? 1 : 0);
    }
};
```

```
template <class T>
class thinheap
{
public:
   Node<T>* first; //указатель на корень с максимальным приоритетом
   Node<T>* last; //указатель на последний корень

   thinheap() { first = last = NULL; }
   ~thinheap() { first = last = NULL; }

   Node<T>* getMax() { return first; }

   Node<T>* insert(Node<T>* newnode_)
   {
      if (newnode_ == NULL)return NULL;

      if (first == NULL)
         first = last = newnode_;
      else
      {
         if (newnode_->getPriority() > first->getPriority())
         {
            newnode_->setRight(first);
            first = newnode_;
         }
         else
         {
            last->setRight(newnode_);
            last = newnode_;
            newnode_->setRight(NULL);
         }
      }
      return newnode_;
   }

   Node<T>* insert(T priority_)
   {
      Node<T>* newnode = new Node<T>;
      newnode->setPriority(priority_);
      return insert(newnode);
   }

   thinheap merge(thinheap heap_)
   {
      thinheap tmp;
      if (first->getPriority() > heap_.first->getPriority())
      {
         tmp.first = first;
         tmp.last = last;
         tmp.last->setRight(heap_.first);
         tmp.last = heap_.last;
      }
```

```
        else
        {
            tmp.first = heap_.first;
            tmp.last = heap_.last;
            tmp.last->setRight(first);
            tmp.last = last;
        }

        return tmp;
    }

    Node<T>* extractMax()
    {
        //удаляем из корневого списка узел с максимальным приоритетом, после чего добавляем в корневой список всех его детей, и понижаем ранг всех "тонких" детей
        Node<T>* res = first;
        Node<T>* cur = res->getChild();
        while (cur != NULL)
        {
            Node<T>* next = cur->getRight();
            if (cur->isThin())
                cur->setRank(cur->getRank() - 1);
            cur->setLeft(NULL);
            cur->setRight(NULL);
            insert(cur);
            cur = next;
        }
        cur = res->getRight();
        //создаем массив из указателей на узлы i-й элемент которого будет хранить тонкое дерево ранга i
        Node<T>* nodes[64] = { NULL };
        int max_rank = -1;
        //заполняем массив узлами из корневого списка, сливая деревья одинакового ранга. int max_rank нужен для ограничения перебора всех элементов массива
        while (cur != NULL)
        {
            Node<T>* next = cur->getRight();
            while (nodes[cur->getRank()] != NULL)
            {
                if (cur->getPriority() < nodes[cur->getRank()]->getPriority()) std::swap(cur, nodes[cur->getRank()]);
                nodes[cur->getRank()]->setLeft(cur);
                if (cur->getChild() != NULL)
                {
                    nodes[cur->getRank()]->setRight(cur->getChild());
                    cur->getChild()->setLeft(nodes[cur->getRank()]);
                }
                else
                    nodes[cur->getRank()]->setRight(NULL);
                cur->setChild(nodes[cur->getRank()]);
                nodes[cur->getRank()] = NULL;
```

```
            cur->setRank(cur->getRank() + 1);
            cur->setRight(NULL);
        }
        if (cur->getRank() > max_rank)
            max_rank = cur->getRank();
        nodes[cur->getRank()] = cur;
        cur = next;
    }
    //обнуляем указатели на первый и последний узлы корневого списка, после
чего добавляем в него все тонкие деревья из массива nodes
    first = last = NULL;
    for (int i = 0; i <= max_rank; i++)
    {
        insert(nodes[i]);
    }
    return res;
}


Node<T>* changePriority(Node<T>* node_, T newpriority_)
{
    node_->setPriority(newpriority_);
    if (node_->getLeft() != NULL)
    {
        Node<T>* left = node_->getLeft();
        if (node_->getLeft()->getChild() == node_)
        {
            if (node_->getRight() == NULL)
                node_->getLeft()->setChild(NULL);
            else
            {
                node_->getLeft()->setChild(node_->getRight());
                node_->getRight()->setLeft(node_->getLeft());
            }
        }
        else
        {
            if (node_->getRight() == NULL)
                node_->getLeft()->setRight(NULL);
            else
            {
                node_->getLeft()->setRight(node_->getRight());
                node_->getRight()->setLeft(node_->getLeft());
            }
        }
        while (sibling_violation(left) && child_violation(left))
        {
            if (sibling_violation(left))
            {
                if (left->isThin())
```

```
                {
                    left->setRank(left->getRank() - 1);
                    left = left->getLeft();
                }
                else
                {
                    if (left->getRight()->getRight() != NULL)
                        left->getRight()->getRight()->setLeft(left);
                    left->getLeft()->setChild(left->getRight());
                    left->getRight()->setLeft(left->getLeft());
                    left->setLeft(left->getRight());
                    left->setRight(left->getLeft()->getRight());
                    left->getLeft()->setRight(left);
                }
            }
            else
            {
                Node<T>* newviolation = left->getLeft();
                left->setRank(left->getRank() - 2);
                if (left->getLeft()->getChild() != left)
                {
                    left->getLeft()->setRight(left->getRight());
                    left->getRight()->setLeft(left->getLeft());
                }
                else
                {
                    if (left->getRight() != NULL)
                        left->getRight()->setLeft(left->getLeft());
                    left->getLeft()->setChild(left->getRight());
                }
                left->setLeft(NULL);
                left->setRight(NULL);
                insert(left);
                left = newviolation;
            }
        }
    }
    node_->setLeft(NULL);
    node_->setRight(NULL);
    insert(node_);
    if (node_->isThin())
        node_->setRank(node_->getRank() - 1);
    return node_;
}

bool sibling_violation(Node<T>* node_)
{
    if (node_->getRight() == NULL && node_->getRank() == 1)return 1;
    if (node_->getRight() != NULL && node_->getRight()->getRank() + 2 == node_-
>getRank())return 1;
    return 0;
}
```

```
bool child_violation(Node<T>* node_)
   {
      if (node_->getRank() == 2 && node_->getChild() == NULL)return 1;
      if (node_->getChild() != NULL && node_->getChild()->getRank() + 3 ==
node_->getRank())return 1;
      return 0;
   }

   vector<T> heaptovector()
   {
      vector<T> res;

      while (getMax() != NULL)
      {
         res.push_back(getMax()->getPriority());
         extractMax();
      }
      return res;
   }

   T QuantileSearch(double val)
   {
      vector<T> v = heaptovector();
      int approx = val * v.size();
      approx = v.size() - approx;
      return v[approx];
   }


};

int main()
{
   thinheap<int> h;
   h.insert(25);
   h.insert(24);
   h.insert(22);
   Node<int>* ptr = h.insert(22);
   h.insert(45);
   h.insert(8);
   h.insert(555);

   h.changePriority(ptr, 1);

   thinheap<int> b;
   b.insert(45);
   b.insert(63);
   b.insert(111);
   b.insert(555);
```

```
    b.insert(943);
    b.insert(54);

    thinheap<int> c;
    c.insert(12);
    c.insert(45);
    c.insert(67);
    c.insert(2);
    c.insert(24);
    c.insert(91);
    c.insert(66);
    c.insert(8);
    c.insert(1);
    c.insert(9);

    thinheap<int> a = h.merge(b);


    for (int i = 1; a.getMax() != NULL; i++, a.extractMax())
       cout << a.getMax()->getPriority() << " ";
    cout << "\n";

    cout << c.QuantileSearch(0.4);

}
```