МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по лабораторной работе №5
по дисциплине «Алгоритмы и структуры данных»
Тема: Рандомизированная дерамида поиска

Студент гр. 9304 _____ Арутюнян В.В.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург
2020

Цель работы.

Ознакомиться с понятием рандомизированной дерамиды поиска, реализовать её, решить поставленную задачу на её основе.

Постановка задачи.

Вариант *12.*

Реализовать БДП (Рандомизированная дерамида поиска (treap))

1) По заданной последовательности элементов Elem построить структуру данных определённого типа – БДП или хеш-таблицу;

2) Выполнить следующее:

а) Для построенной структуры данных проверить, входит ли в неё элемент e типа Elem, и если входит, то в скольких экземплярах. Добавить элемент e в структуру данных.

Выполнение работы.

Программа на вход ожидает строку для анализа сразу после флага “*-s*”, иначе ожидается путь до файла со строкой для обработки. Возможно совместное использование:

./lab5 some/path/1 -s “some_string_1” -s “some_string_2” some/path/2

При таком вызове программа обработает строку из файла *some/path/1*, затем строку *some_string_1*, после строку *some_string_2*, далее строку из файла *some/path/2*.

Далее полученная строка анализируется, в строке могут быть как команды управления, так и сами элементы, над которыми производятся операции. Доступны следующие команды управления:

1. «#Insert» – вставка элемента в дерево;
2. «#Remove» – удаление элемента из дерева, если он существует, иначе команда просто игнорируется;
3. «#Count» – узнать, в каком количестве присутствует элемент e типа Elem в дереве;

4. «#Task» – узнать, в каком количестве присутствует элемент e типа Elem в дереве, а затем добавить его в дерево.

Если на вход не подать команды, то по умолчанию используется команда «#Insert», то есть все, введенные элементы далее, будут добавляться в дерево.

Пример ввода:

«2 3 1 2 3 5 6 #Remove 3 5 6 #Task 3 3 #Count 3 3 #Insert 0»

Рассмотрим, как будет анализироваться данная строка:

1. Так как команды не поступило, все полученные далее элементы просто будут добавляться в дерево. То есть элементы «2», «3», «1», «2», «3», «5», «6». На рисунке 1 продемонстрировано, как выглядит дерево послед данного этапа.

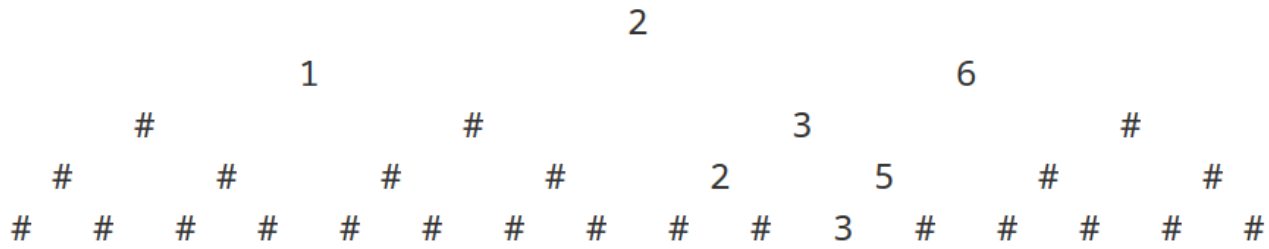


Рисунок 1 – Представление дерева на первом шаге

2. Далее идёт команда «#Remove», а элементы дальше будут удаляться из дерева, если это возможно, либо команда проигнорируется. То есть будет попытка удалить элементы «3», «5», «6». Все три попытки успешно выполнятся, так как такие элементы в дереве уже присутствовали. На рисунке 2 продемонстрировано, как выглядит дерево послед данного этапа.

Рисунок 2 – Представление дерева на втором шаге

3. Поступает команда «#Task», далее для каждого элемента e типа Elem выполнится проверка, которая покажет, в каком количестве присутствует данный элемент в дереве. А затем добавится элемент e в дерево. В ходе работы были получены две строки: «Count = 1 (for the "3" element)» и «Count = 2 (for the "3" element)». Первая говорит о том, что в дереве есть элемент «3» в количестве 1 шт., а вторая – существует элемент «3» в двух экземплярах. На рисунке 3 продемонстрировано, как выглядит дерево послед данного этапа.

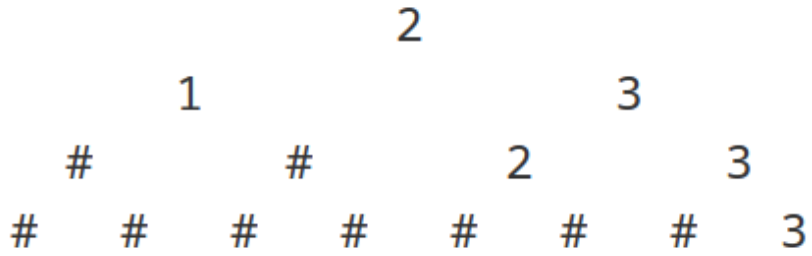


Рисунок 3 – Представление дерева на третьем шаге

4. После следует команда «#Count». Далее для соответствующих элементов вернется, в каком количестве они присутствуют. В ходе работы были получены следующие строки: «Count = 3 (for the "3" element)» и «Count = 3 (for the "3" element)». Первая сообщает, что элемент «3» присутствует в дереве в количестве 3 шт., вторая – существует элемент «3» в трёх экземлярах. Дерево на данном шаге никак не изменяется.

5. Далее идёт команда «#Insert», после которой все элементы будут добавляться в дерево. А именно: «0». На рисунке 4 продемонстрировано, как выглядит дерево послед данного этапа.

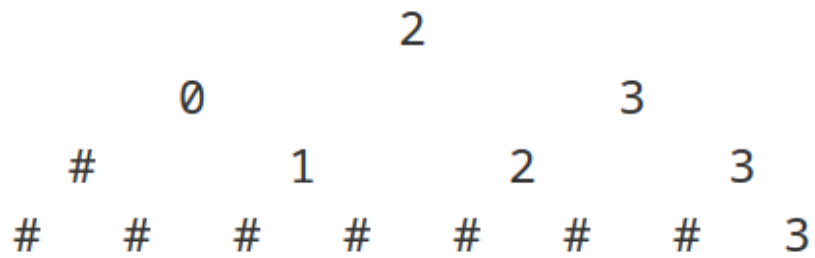


Рисунок 4 – Представление дерева на пятом шаге

Было реализовано два класса: Tree и Treap. Оба являются шаблонными.

Класс Tree представляет из себя элемент, в котором хранятся следующие поля:

1. data_ – основные данные, которые были введены пользователем, с помощью данного поля поддерживаются свойства бинарного дерева в дерамиде поиска;
2. priority_ – случайно сгенерированный приоритет в виде целого числа, необходим для поддержания свойств кучи в дерамиде поиска;
3. left_ – указатель на левого потомка;
4. right_ – указатель на правого потомка.

В классе Treap представляет собой рандомизированную дерамиду поиска. В данном классе находится лишь одно поле:

1. head_ – указатель на элемент типа Tree, является головным элементом дерамиды поиска.

Для данного класса были реализованы следующие методы:

1. Insert() – производит вставку элемента из дерева;
2. Remove() – производит удаление элемента из дерева;
3. Count() – узнает, в каком количестве присутствует определенный элемент в дереве;
4. GetHeight() – узнает высоту дерева;
5. CountAndThenInsert() – аналогично Count(), только после еще вставляет элемент в дерево, по которому изначально производился поиск;
6. operator<< – вывод дерамиды поиска в удобном виде;
7. InsertPrivate() – необходим для работы Insert();
8. RemovePrivate() – необходим для работы Remove();
9. Split() – метод, выполняющий разделение дерева на два, по определенному значению data. Метод возвращает две дерамиды поиска, где в левой находятся все элементы меньше data, а в правом – все остальные;

10. Merge() – метод, выполняющий слияние двух дерамид поиска в одну. Возвращает одну дерамиду поиска;

11. GetHeightPrivate() – необходим для работы GetHeight();

12. DumpFullBinaryTree() – необходим для работы более удобного вывода всего дерева через перегруженный оператор operator<<;

13. RecursiveDumpFullBinaryTree() – необходим для работы DumpFullBinaryTree();

14. RecursiveCount() – необходим для работы Count().

Кратко о том, как работают основные методы:

Split()

Метод Split() разрезает исходное дерево tree по значению data. Возвращает указатель на элемент типа Tree, где потомками являются две дерамиды поиска, которые и необходимо было получить.

Пусть необходимо разрезать дерево по значению data, которое больше, чем значение в корне. Назовем исходное дерево tree, а возвращаемые деревья – left и right, где в left находятся все элементы со значениями меньше, чем data, а в right – все остальные.

Левое поддерево left будет совпадать с левым поддеревом tree. Поэтому для нахождения правого поддерева left и всего right, необходимо разрезать правое поддерево tree на left“ и right“ по значению data. Тогда left“ будет правым поддеревом left, a right совпадет с right“.

Если data меньше или равно значению в корне, то ситуация рассматривается симметрично.

Merge()

Метод Merge() сливает два дерева left и right в одно tree.

Корнем tree должна стать вершина из left или right с наибольшим приоритетом priority. Так как наибольший приоритет в дереве находится в корне, то корнем tree станет либо корень left, либо корень right.

Пусть priority больше у left. Тогда левое поддерево tree совпадет с левым поддеревом left. Правым поддеревом будет Merge() правого поддерева left и дерева right.

Ситуация, когда priority больше у right рассматривается симметрично.

Insert()

Метод Insert() добавляет в дерево tree элемент elem, elem.priority – приоритет элемента, elem.data – данные элемента.

Сначала необходимо, спускаясь по дереву, найти первый элемент, в котором значение приоритета priority окажется меньше, чем elem.priority. После достаточно разбить дерево от найденного элемента с помощью Split(), полученные left и right подвязать как потомков к elem, а elem вставить на место найденного элемента в дереве.

Remove()

Метод Remove() удаляет из дерева tree элемент со значением data.

Сначала необходимо, спускаясь по дереву, найти удаляемый элемент del. Далее нужно с помощью Merge() слить потомков del, а результат слияния вставить вместо удаляемого элемента del.

Count()

Метод Count() возвращает целое число – количество элементов е типа Elem в дереве tree. Подсчёт элементов в дереве ведется простым рекурсивным обходом с помощью метода RecursiveCount().

CountAndThenInsert()

В данном методе используются уже готовые методы Count() и Insert(). Сначала сохраняется значение, которое вернет Count() для элемента e типа Elem, затем с помощью Insert() элемент e добавляется в дерево. После возвращается сохраненное значение.

GetHeight()

GetHeight() возвращает высоту дерева tree. Подсчёт ведется с помощью рекурсивного обхода в методе GetHeightPrivate().

Программа выводит результаты запросов для методов Count и CountAndThenInsert(). В конце выводится полученное дерево после всех выполненных шагов.

Исходный код находится в приложении А.

Тестирование.

Программу можно собрать через *Makefile* командой *make*, после этого создается исполняемый файл *lab5*. Для проведения тестирования необходимо:

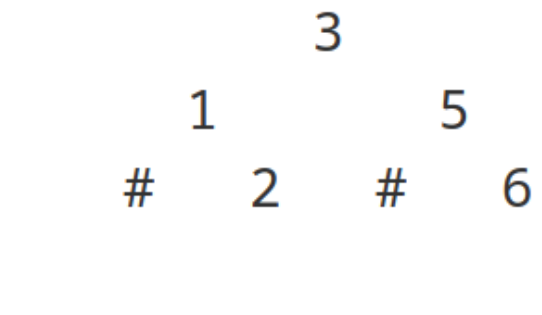
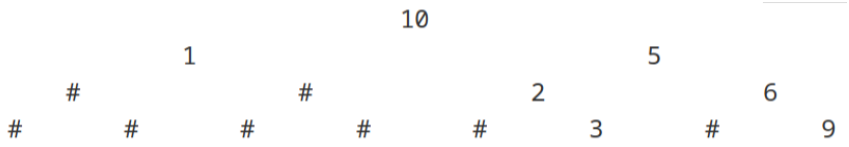
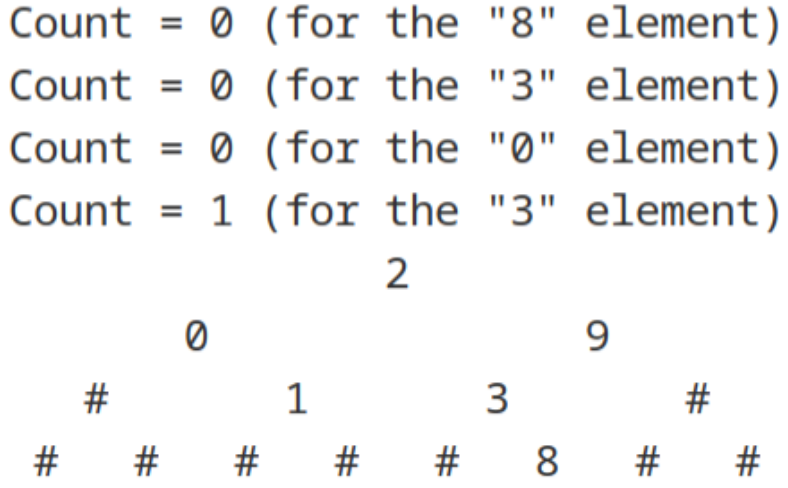
1. Вызвать *lab5*, указав путь до файла с тестом, либо передать флаг "-*s*", а затем строку, которую требуется проанализировать, в кавычках;

Далее будет представлена таблица тестирования с несколькими тестами.

Таблица 1. Примеры входных и выходных данных

№	Входные данные	Выходные данные
1	2 3 1 2 3 5 6 #Remove 3 5 6 #Task 3 3 #Count 3 3 #Insert 0	Count = 1 (for the "3" element) Count = 2 (for the "3" element) Count = 3 (for the "3" element) Count = 3 (for the "3" element) 3 2 3 0 2 # 3 # 1 # # # # # #
2	#Insert 3 8 0 #Insert 1 0 #Remove 7 3	0 # 8 # # 0 # # # # # # 1 # #
3	#Insert 3 8 0 #Task 3 3 #Count 3 #Remove 7 3 #Count 3	Count = 1 (for the "3" element) Count = 2 (for the "3" element) Count = 1 (for the "3" element) 0 # 8 # # 3 #
4	#Count 1 #Insert 1 #Count 1 #Remove 1 #Count 1	Count = 0 (for the "1" element) Count = 1 (for the "1" element) Count = 0 (for the "1" element)

5	#Remove 0 1 2 #Insert 2 0 1 # #Task 0	Count = 1 (for the "0" element) 0 # 1 # # 0 2
6	#Task 0 0 0 #Insert 1 #Remove 0 0 0 #Task 0	Count = 0 (for the "0" element) Count = 1 (for the "0" element) Count = 2 (for the "0" element) Count = 0 (for the "0" element) 1 0 #
7	#Task 1 0 2 5 4 #Insert 4 #Count 4	Count = 0 (for the "1" element) Count = 0 (for the "0" element) Count = 0 (for the "2" element) Count = 0 (for the "5" element) Count = 0 (for the "4" element) Count = 2 (for the "4" element) 4 0 4 # 1 # 5 # # # 2 # # # #

8	3 2 5 6 1	
9	3 2 5 6 1 #Insert 10 9	
10	#Task 8 3 #Insert 1 2 #Task 0 #Insert 9 #Count 3	

Выводы.

В ходе выполнения лабораторной работы был реализован класс рандомизированной дерамиды поиска Treap, который объединяет в себе бинарное дерево и бинарную кучу. Рандомизирования дерамида поиска позволяет в среднем строить невырожденные деревья за счет выбора случайных приоритетов, что обеспечивает асимптотику O(logN) в среднем, вставка, удаление, поиск количества элемента в среднем выполняется за O(logN), где N – количество элементов.

ПРИЛОЖЕНИЕ А

main.cc

```
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>

#include "../lib/treap.h"

enum class Operation {
  kInsert = 0,
  kRemove,
  kCountAndThenInsert,
  kCount,
};

template <typename T>
void Analyze(const std::string& str, Treap<T> treap) {
  Operation current_operation = Operation::kInsert;
  std::stringstream ss;
  std::string in;
  ss << str;

  while (ss >> in) {
    if (in == "#Insert") {
      current_operation = Operation::kInsert;
    } else if (in == "#Remove") {
      current_operation = Operation::kRemove;
    } else if (in == "#Task") {
      current_operation = Operation::kCountAndThenInsert;
    } else if (in == "#Count") {
      current_operation = Operation::kCount;
    } else {
      if (current_operation == Operation::kInsert) {
        treap.Insert(in);
      } else if (current_operation == Operation::kRemove) {
        treap.Remove(in);
      } else if (current_operation == Operation::kCount) {
        std::cout << "Count = " << treap.Count(in) << " (for
the \"" << in
                  << "\" element)" << '\n';
      } else if (current_operation ==
Operation::kCountAndThenInsert) {
        std::cout << "Count = " << treap.CountAndThenInsert(in)
```

```
                  << " (for the \"" << in << "\" element)" << '\
n';
      }
    }
  }
  std::cout << treap << '\n';
}

int main(int argc, char** argv) {
  if (argc == 1) {
    std::cout << "Too small arguments.\n"
              << "A expression is expected after \"-s\" flag, "
              << "otherwise a file path is expected.\n\n"
              << "example: ./lab5 -s \"2 3 1 2 3 5 6 Remove 3 5 6
Task 2\" "
                 "Tests/test/test1.txt\n";
  } else {
    bool is_string = false;

    for (int i = 1; i < argc; ++i) {
      bool is_open = false;
      std::string arg = argv[i];
      Treap<std::string> treap;

      if (arg == "-s" && !is_string) {
        is_string = true;
      } else {
        std::string str = arg;
        if (!is_string) {
          std::ifstream file_in(arg);
          if (file_in.is_open()) {
            is_open = true;
            std::getline(file_in, str);
            file_in.close();
          } else {
            std::cout << "Couldn't open the file.\n";
          }
        }
        if (!(!is_string && !is_open)) {
          Analyze<std::string>(str, treap);
        }
      }
    }  // for
  }    // else
  return 0;
}  // main
```

treap.h

```
#ifndef TREAP_H_
#define TREAP_H_

#include <iomanip>
#include <memory>
#include <queue>
#include <random>
#include <sstream>
#include <vector>

#include "tree.h"

template <typename T>
class Treap {
 public:
  Treap() = default;
  void Insert(const T& data);
  void Remove(const T& data);
  size_t Count(const T& data) const;
  size_t GetHeight() const;
  size_t CountAndThenInsert(const T& data);

  template <typename TT>
  friend std::ostream& operator<<(std::ostream& out, const
Treap<TT>& tree);

  ~Treap() = default;

 private:
  std::shared_ptr<Tree<T>> head_;
  std::shared_ptr<Tree<T>> InsertPrivate(std::shared_ptr<Tree<T>>
tree,
                                         std::shared_ptr<Tree<T>>
elem);
  void RemovePrivate(std::shared_ptr<Tree<T>> tree, const T& key);
  std::shared_ptr<Tree<T>> Split(std::shared_ptr<Tree<T>> tree,
const T& data);
  std::shared_ptr<Tree<T>> Merge(std::shared_ptr<Tree<T>> left,
std::shared_ptr<Tree<T>> right);
  size_t GetHeightPrivate(const std::shared_ptr<Tree<T>> tree)
const;
  std::vector<std::vector<std::shared_ptr<Tree<T>>>>
DumpFullBinaryTree() const;
```

```
  void
RecursiveDumpFullBinaryTree(std::vector<std::vector<std::shared_pt
r<Tree<T>>>>& vec, std::shared_ptr<Tree<T>> elem, long long
current_level = 0) const;
  size_t RecursiveCount(std::shared_ptr<Tree<T>> tree, const T&
data) const;
};

#include "treap.inl"

#endif  // TREAP_H_
```

treap.inl

```
#include "treap.h"

template <typename T>
void Treap<T>::Insert(const T& data) {
  std::random_device rd;
  std::mt19937 mersenne_twister(rd());
  std::shared_ptr<Tree<T>> elem(new Tree<T>(data,
mersenne_twister()));
  head_ = InsertPrivate(head_, elem);
}

template <typename T>
void Treap<T>::Remove(const T& data) {
  RemovePrivate(head_, data);
}

template <typename T>
size_t Treap<T>::Count(const T& data) const {
  return RecursiveCount(head_, data);
}

template <typename T>
size_t Treap<T>::GetHeight() const {
  return GetHeightPrivate(head_);
}

template <typename T>
size_t Treap<T>::CountAndThenInsert(const T& data) {
  size_t save_count = Count(data);
  Insert(data);
  return save_count;
```

```
}

template <typename T>
std::ostream& operator<<(std::ostream& out, const Treap<T>& tree)
{
  std::queue<std::shared_ptr<Tree<T>>> queue;
  const size_t height = tree.GetHeight();
  size_t element_length = 0;

  if (tree.head_ != nullptr) {
    queue.push(tree.head_);
  }

  while (!queue.empty()) {
    std::shared_ptr<Tree<T>> elem = queue.front();
    queue.pop();

    if (elem != nullptr) {
      std ::stringstream ss;
      ss << elem->data_;
      element_length = std::max(element_length, ss.str().size());

      queue.push(elem->left_);
      queue.push(elem->right_);
    }
  }

  auto vec = tree.DumpFullBinaryTree();

  for (int current_level = 0; current_level < height; +
+current_level) {
    if (current_level) {
      out << '\n';
    }
    for (int i = 0; i < vec[current_level].size(); ++i) {
      size_t cell_width = height - current_level + 1;
      if (!i) {
        --cell_width;
      }
      out << std::setw((1 << cell_width) * element_length);
      if (vec[current_level][i] == nullptr) {
        out << '#';
      } else {
        out << vec[current_level][i]->data_;
      }
    }
```

```
  }
  return out;
}

template <typename T>
std::shared_ptr<Tree<T>> Treap<T>::InsertPrivate(
    std::shared_ptr<Tree<T>> tree, std::shared_ptr<Tree<T>> elem)
{
  std::shared_ptr<Tree<T>> new_element(nullptr);
  if (tree == nullptr) {
    new_element = elem;
  } else if (elem->priority_ > tree->priority_) {
    new_element = Split(tree, elem->data_);
    elem->left_ = new_element->left_;
    elem->right_ = new_element->right_;
    new_element = elem;
  } else {
    if (elem->data_ < tree->data_) {
      tree->left_ = InsertPrivate(tree->left_, elem);
    } else {
      tree->right_ = InsertPrivate(tree->right_, elem);
    }
    new_element = tree;
  }
  return new_element;
}

template <typename T>
void Treap<T>::RemovePrivate(std::shared_ptr<Tree<T>> tree, const
T& data) {
  if (tree == nullptr) {
    return;
  } else if (tree->data_ != data) {
    if (data < tree->data_) {
      if (tree->left_ != nullptr) {
        if (tree->left_->data_ != data) {
          RemovePrivate(tree->left_, data);
        } else {
          tree->left_ = Merge(tree->left_->left_, tree->left_-
>right_);
        }
      }

    } else {
      if (tree->right_ != nullptr) {
        if (tree->right_->data_ != data) {
```

```
          RemovePrivate(tree->right_, data);
        } else {
          tree->right_ = Merge(tree->right_->left_, tree->right_-
>right_);
        }
      }
    }
  } else {
    head_ = Merge(head_->left_, head_->right_);
  }
}

template <typename T>
std::shared_ptr<Tree<T>> Treap<T>::Split(std::shared_ptr<Tree<T>>
tree,
                                         const T& data) {
  std::shared_ptr<Tree<T>> new_element(nullptr);
  if (tree == nullptr) {
    new_element.reset(new Tree<T>(nullptr, nullptr));
  } else if (data > tree->data_) {
    new_element = Split(tree->right_, data);
    tree->right_ = new_element->left_;
    new_element->left_ = tree;
  } else {
    new_element = Split(tree->left_, data);
    tree->left_ = new_element->right_;
    new_element->right_ = tree;
  }
  return new_element;
}

template <typename T>
std::shared_ptr<Tree<T>> Treap<T>::Merge(std::shared_ptr<Tree<T>>
left,
                                         std::shared_ptr<Tree<T>>
right) {
  std::shared_ptr<Tree<T>> new_element(nullptr);
  if (left == nullptr) {
    new_element = right;
  } else if (right == nullptr) {
    new_element = left;
  } else if (left->priority_ > right->priority_) {
    left->right_ = Merge(left->right_, right);
    new_element = left;
  } else {
    right->left_ = Merge(left, right->left_);
```

```
    new_element = right;
  }
  return new_element;
}

template <typename T>
size_t Treap<T>::GetHeightPrivate(const std::shared_ptr<Tree<T>>
tree) const {
  if (tree == nullptr) {
    return 0;
  }
  return 1 + std::max(GetHeightPrivate(tree->left_),
                      GetHeightPrivate(tree->right_));
}

template <typename T>
std::vector<std::vector<std::shared_ptr<Tree<T>>>>
Treap<T>::DumpFullBinaryTree() const {
  std::vector<std::vector<std::shared_ptr<Tree<T>>>>
vec(GetHeight());
  RecursiveDumpFullBinaryTree(vec, head_);
  return vec;
}

template <typename T>
void Treap<T>::RecursiveDumpFullBinaryTree(
    std::vector<std::vector<std::shared_ptr<Tree<T>>>>& vec,
    std::shared_ptr<Tree<T>> elem, long long current_level) const
{
  if (current_level == vec.size()) {
    return;
  }

  vec[current_level].push_back(elem);

  RecursiveDumpFullBinaryTree(vec, elem == nullptr ? nullptr :
elem->left_,
                              current_level + 1);
  RecursiveDumpFullBinaryTree(vec, elem == nullptr ? nullptr :
elem->right_,
                              current_level + 1);
}

template <typename T>
size_t Treap<T>::RecursiveCount(std::shared_ptr<Tree<T>> tree,
                                const T& data) const {
```

```
  if (tree == nullptr) {
    return 0;
  }

  return (tree->data_ == data) + (tree->data_ > data
                                      ? RecursiveCount(tree->left_,
data)
                                      : RecursiveCount(tree-
>right_, data));
}
```

tree.h

```
#ifndef TREE_H_
#define TREE_H_

#include <memory>

template <typename T>
class Treap;

template <typename T>
class Tree {
 public:
  Tree();
  Tree(const T& data, long long priority = 0,
       std::shared_ptr<Tree<T>> left = nullptr,
       std::shared_ptr<Tree<T>> right = nullptr);
  Tree(std::shared_ptr<Tree<T>> left, std::shared_ptr<Tree<T>>
right);

 private:
  friend class Treap<T>;
  template <typename TT>
  friend std::ostream& operator<<(std::ostream& out, const
Treap<TT>& tree);

  T data_;
  long long priority_;
  std::shared_ptr<Tree<T>> left_;
  std::shared_ptr<Tree<T>> right_;
};

#include "tree.inl"

#endif  // TREE_H_
```

tree.inl

```
#include "tree.h"

template <typename T>
Tree<T>::Tree() : priority_(0), right_(nullptr), left_(nullptr) {}

template <typename T>
Tree<T>::Tree(const T& data, long long priority,
std::shared_ptr<Tree<T>> left,
              std::shared_ptr<Tree<T>> right)
    : data_(data), priority_(priority), left_(left), right_(right)
{}

template <typename T>
Tree<T>::Tree(std::shared_ptr<Tree<T>> left,
std::shared_ptr<Tree<T>> right)
    : priority_(0), left_(left), right_(right) {}
```

Makefile

```
.PHONY: all clean

CXX = g++
TARGET = lab5
CXXFLAGS = -g -c -std=c++17
CXXOBJFLAGS = -g -std=c++17
LIBDIR = source/lib
SRCDIR = source/src
SRCS = $(wildcard $(SRCDIR)/*.cc)
OBJS = $(SRCS:.cpp=.o)

all: $(TARGET)

$(TARGET): $(OBJS)
     @echo "Compiling:"
     $(CXX) $(CXXOBJFLAGS) $(OBJS) -o $(TARGET)

%.o: $(SRCDIR)/%.cpp $(LIBDIR)/*.h
     $(CXX) $(CXXFLAGS) $<

clean:
     @echo "Cleaning build files:"
     rm -rf $(SRCDIR)/*.o $(TARGET)
```