

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

КУРСОВАЯ РАБОТА
по дисциплине «Алгоритмы и структуры данных»
Тема: Рандомизированная пирамида поиска - вставка и исключение.
Демонстрация.

Студент гр. 7381

Павлов А.П.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2018

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студент Павлов А.П.

Группа 7381

Тема работы: Рандомизированная пирамида поиска - вставка и исключение.
Демонстрация.

Исходные данные:

На вход программе подаются следующие данные: элементы для построения рандомизированной пирамиды поиска, номер действия, необходимого для работы с программой, и элемент для работы с пирамидой.

Содержание пояснительной записки:

Содержание, Аннотация, Введение, Постановка задачи, Алгоритм работы программы, Функции и структуры данных, Пользовательский интерфейс, Тестирование, Заключение.

Предполагаемый объем пояснительной записки:

Не менее 20 страниц.

Дата сдачи реферата:

Дата защиты реферата:

Студент

Павлов А.П.

Преподаватель

Фирсов М.А.

Содержание

Содержание	3
Аннотация.....	4
Введение	5
Постановка задачи.....	6
Алгоритм работы программы	7
Функции и структуры данных	9
Пользовательский интерфейс	12
Тестирование	15
Заключение	24

Аннотация

В данной курсовой работе реализованы рандомизированная пирамида поиска и основные функции для работы с ней: вставка и удаление элемента из пирамиды, поиск заданного элемента, слияние двух пирамид и разрезание одной пирамиды на две по значению ключа.

В работе рандомизированная пирамида поиска создавалось с целью, чтобы программу можно было использовать в обучении для объяснения используемой структуры данных и выполняемых с нею действий. Реализован удобный и понятный для изучения структуры данных БДП интерфейс. В качестве языка программирования для создания программы выбран язык программирования C++.

Введение

Данная курсовая работа основана на работе с рандомизированной пирамидой поиска.

Рандомизированная пирамида поиска – это структура данных, объединяющая в себе бинарное дерево поиска и бинарную кучу (отсюда и второе её название: treap (tree+heap) и дерамида (дерево+пирамида). Рандомизированная пирамида поиска представлена на рис. 1.

Более строго, это структура данных, которая хранит пары (X, Y) в виде бинарного дерева таким образом, что она является бинарным деревом поиска по x и бинарной пирамидой по y . Предполагая, что все X и все Y являются различными, получаем, что если некоторый элемент дерева содержит (X_0, Y_0) , то у всех элементов в левом поддереве $X < X_0$, у всех элементов в правом поддереве $X > X_0$, а также и в левом, и в правом поддереве имеем: $Y < Y_0$.

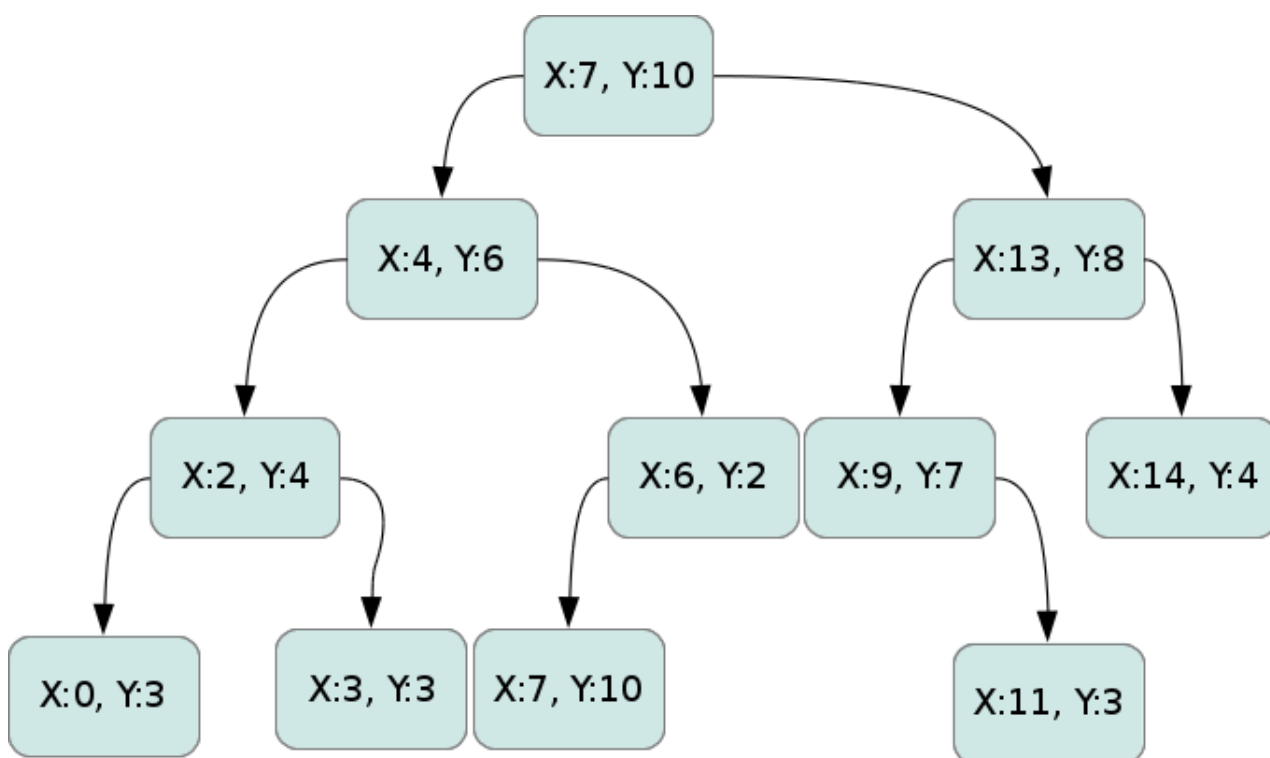


Рисунок 1 – рандомизированная пирамида поиска

Постановка задачи

Реализация демонстрации по рандомизированным пирамидам поиска. "Демонстрация" - визуализация структур данных, алгоритмов, действий. Демонстрация должна быть подробной и понятной (в том числе сопровождаться пояснениями), чтобы программу можно было использовать в обучении для объяснения используемой структуры данных и выполняемых с нею действий. Особенно подробно должна быть продемонстрирована работа функций вставки исключения и построение рандомизированной пирамиды поиска.

Алгоритм работы программы

На вход программе подаются элементы, которые в последующем станут узлами пирамиды. Далее происходит проверка введенных элементов на корректность и построение пирамиды, сопровождающееся подробным выводом на экран. После построения пирамиды в программу подаётся номер действия: 1 - вставка элемента, 2 – исключение элемента, 3 – завершение программы. В случае введения некорректного действия – программа выводит сообщение об ошибке и предлагает ввести данные ещё раз. Когда пользователь введёт номер действия – ему необходимо будет ввести с клавиатуры элемент для работы с пирамидой.

Описание алгоритма вставки:

Метод `insert` получает на вход указатель на корень пирамиды и значение добавляемого элемента. Сначала проверяется, существует ли данный элемент в пирамиде, если да, то метод завершает работу и возвращает указатель на корень. В ином случае происходит разрезание пирамиды на две пирамиды по значению переданного элемента таким образом, что в первой пирамиде находятся элементы меньшие или равные значению, а во второй – большие. Затем выделяется память под переданный элемент, то есть создается дерево, состоящее из одного узла, которое потом объединяется с первым деревом. Созданное дерево с учетом приоритетов объединяется с вторым деревом, таким образом переданный элемент вставляется в дерево.

Описание алгоритма исключения:

Метод `erase` получает на вход указатель на корень пирамиды и значение удаляемого элемента. Сначала проверяется, существует ли данный элемент в пирамиде, если нет, то метод завершает работу и возвращает указатель на корень. В ином случае пирамида разрезается по значению удаляемого элемента. Получается две пирамиды: в первой пирамиде находятся элементы меньшие

значения удаляемого элемента, и он сам, во второй – элементы большие. Затем происходит разрезание первой пирамиды по значению, на единицу меньшему, чем значение удаляемого элемента. Исключаемый элемент удаляется, а две оставшиеся пирамиды сливаются в одну.

Функции и структуры данных

Для выполнения лабораторной работы был написан класс Treap.

`struct Node` – структура, представляющая собой узел дерева.

Содержит в себе:

`Type key` – значение узла типа `Type`;

`int priority` – приоритет узла;

`Node* left` – указатель на левого сына типа `Node`;

`Node* right` – указатель на правого сына типа `Node`.

`struct Trunk` – структура, представляющая собой одно ответвление от узла.

Содержит в себе:

`Trunk *prev` – указатель на предыдущую структуры типа `Trunk`;

`string branch` – строка, представляющая из себя внешний вид ответвления.

`void remove(node * root)` – метод, удаляющий дерево.

Принимаемые аргументы:

`node * t` – корень пирамиды.

Возвращаемое значение: метод ничего не возвращает.

`node* insert(node* root, Type val)` – метод, вставляющий элемент в пирамиду.

Принимаемые аргументы:

`Type val` – элемент для вставки типа `Type`;

`node* root` – корень пирамиды.

Возвращаемое значение: указатель на пирамиду.

`bool exists(node * root, Type val)` – метод нахождения узла в пирамиде.

Принимаемые аргументы:

`node* root` – корень пирамиды;

`Type val` – элемент для поиска.

Возвращаемое значение: `true` – если элемент существует в пирамиде, иначе – `false`.

`std::pair <node *, node *> split(node * root, Type val)` – метод разрезает исходную пирамиду по передаваемому ключу `val` на две: в первой – элементы, меньшие и равные значению `val`.

Принимаемые аргументы:

`node* root` – корень пирамиды;

`Type val` – значение элемента.

Возвращаемое значение: указатели на две новые пирамиды

`node* erase(node * root, Type val)` – метод удаления узла дерева.

Принимаемые аргументы:

`Type val` – элемент для удаления;

`node * root` – корень пирамиды.

Возвращаемое значение: возвращает корень пирамиды.

`void showTrunks(Trunk *p)` – вспомогательный метод для печати рандомизированной пирамиды поиска.

Принимаемые аргументы:

`Trunk *p` – указатель на структуры типа `Trunk`.

Возвращаемое значение: функция ничего не возвращает.

`node * merge(node * root1, node * root2)` – метод объединяет два декартовых дерева в одно с учетом их приоритетов.

Принимаемые аргументы:

`root1` – указатель на узел первой пирамиды;

`root2` – указатель на узел второй пирамиды;

Возвращаемое значение: возвращает указатель на новое дерево.

`void inorder(Node* t, Trunk* prev, bool isRight)` – метод, печатающий рандомизированную пирамиду поиска.

Принимаемые аргументы:

`Node* t` – узел дерева типа `Node`.

`Trunk* prev` – указатель на структуру типа `prev`, предыдущее ответвление дерева.

`bool isRight` – булева переменная, отвечающая за то, является ли элемент правым сыном.

Возвращаемое значение: метод ничего не возвращает.

`Treap()` – конструктор, присваивающий корню пирамиды `NULL`.

`~Treap()` – деструктор, удаляющий пирамиду.

`void display(node * root, Trunk * prev, bool isRight)` – метод, выводящий пирамиду в древовидной форме в стандартный поток вывода.

Принимаемые аргументы:

`node * root` – указатель на корень пирамиды;

`Trunk * prev` – указатель на структуру типа `prev`, предыдущее ответвление пирамиды;

`bool isRight` – булева переменная, отвечающая за то, является ли элемент правым сыном.

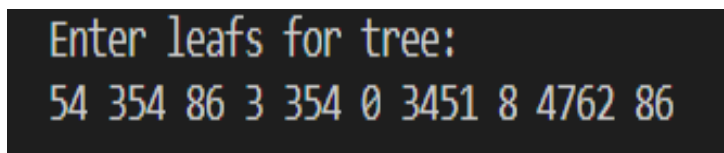
Возвращаемое значение: метод ничего не возвращает.

Пользовательский интерфейс

Работа пользователя с интерфейсом начинается с ввода элементов.

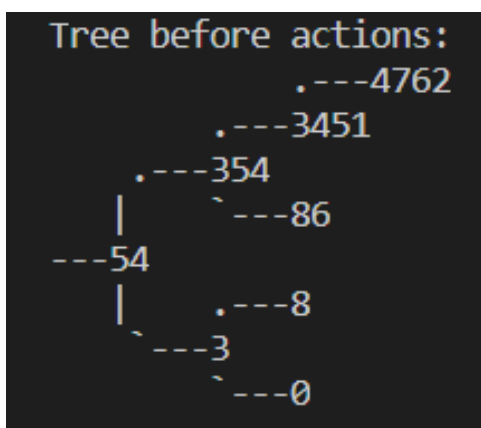
Enter leafs for tree: (ответ пользователя)

После ввода пользователем элементов произойдёт демонстрация построения пирамиды. Ввод элементов представлен на рисунке 2, вывод пирамиды представлен на рисунке 3.



```
Enter leafs for tree:
54 354 86 3 354 0 3451 8 4762 86
```

Рисунок 2- ввод элементов



```
Tree before actions:
      .---4762
       .---3451
        .---354
         |   `---86
        ---54
         |   .---8
         |   `---3
          `---0
```

Рисунок 3 - вывод дерева

Далее пользователю предлагается выбрать действие, которое он хочет выполнить.

Entry rules:

- 1 - Insert element;
- 2 - Delete element;
- 3 - Exit.

Select an action for the item: (ответ пользователя: 1, 2 или 3).

Enter an item to work with: (ответ пользователя)

В случае выбора вставки элемента происходит демонстрация работы функции вставки элемента, представлено на рисунке 3.

```

Enter leafs for tree:
2 47 7 1

                                Work with tree...
The built treap:
    .---(47; 18467)
    .---(7; 6334)
    ---(2; 41)
    `---(1; 26500)

Entry rules:
    1 - Insert element;
    2 - Delete element;
    3 - Exit.
Select an action for the item:
1
Enter an item to work with:
4
<----Inserting element---->
Built treap
    .---(47; 18467)
    .---(7; 6334)
    |   `---(4; 19169)
    ---(2; 41)
    `---(1; 26500)

```

Рисунок 4 - вставка элемента

В случае выбора исключения элемента происходит демонстрация работы функции исключения элемента, представлено на рисунке 4.

```

Built treap
    .---(47; 18467)
    .---(7; 6334)
    |   `---(4; 19169)
    ---(2; 41)
    `---(1; 26500)

Select an action for the item:
2
Enter an item to work with:
7
<----Deleting element---->
Built treap
    .---(47; 18467)
    |   `---(4; 19169)
    ---(2; 41)
    `---(1; 26500)

```

Рисунок 5 - исключение элемента

При попытке ввода некорректных данных программа выводит соответствующее сообщение об ошибке. Если некорректные данные были обнаружены при вводе элементов дерева, то считывание элементов останавливается на некорректном, если же ошибка обнаружена при введении действия, то пользователю предлагается ввести номер заново, если некорректные данные обнаружены при вводе элемента для работы с деревом, то работа программы останавливается.

Тестирование

Для тестирования был использован bat-скрипт, использованный в первых 5 лабораторных работах с небольшими доработками. Данные тестирования представлены в таблице ниже. Ввиду большого объёма выводимой информации большая часть тестов искусственно урезана.

№	Входные данные	Выходные данные
1	4 8 6 14 1 7 3	Enter leafs for tree: 4 8 6 14 The built treap: .---(14; 26500) .---(8; 18467) .---(6; 6334) ---(4; 41) Entry rules: 1 - Insert element; 2 - Delete element; 3 - Exit. Select an action for the item: 1 Enter an item to work with: 7 Priority 41 <= 19169 , go to the right subtree Priority 6334 <= 19169 , go to the right subtree Treap after first merge: .---(7; 19169) .---(6; 6334) ---(4; 41) Priority 41 <= 18467 , go to the right subtree Priority 6334 <= 18467 , go to the right subtree Priority 19169 > 18467 , go to the left subtree Treap after second merge:

		<pre> .---(14; 26500) .---(8; 18467) `---(7; 19169) .---(6; 6334) ---(4; 41) Built treap .---(14; 26500) .---(8; 18467) `---(7; 19169) .---(6; 6334) ---(4; 41) Select an action for the item: 3 The end of program... </pre>
2	<pre> 74 354 596 123 695 2 596 3 </pre>	<pre> Enter leafs for tree: 23 17 41 42 77 97 The built treap: .---(695; 19169) .---(596; 6334) `---(354; 18467) `---(123; 26500) ---(74; 41) Entry rules: 1 - Insert element; 2 - Delete element; 3 - Exit. Select an action for the item: 2 Enter an item to work with: 696 <----Deleting element----> Splitting key - 596 Treaps after first split Treap with key <= 596 </pre>

		<p>.---(596; 6334)</p> <p> `---(354; 18467)</p> <p> `---(123; 26500)</p> <p>---(74; 41)</p> <p>Treap with key > 596</p> <p>---(695; 19169)</p> <p>Treaps after second split</p> <p>Treap with key < 596</p> <p>.---(354; 18467)</p> <p> `---(123; 26500)</p> <p>---(74; 41)</p> <p>Treap with 596 to be deleted</p> <p>---(596; 6334)</p> <p>Priority 41 ≤ 19169 , go to the right subtree</p> <p>Priority 18467 ≤ 19169 , go to the right subtree</p> <p>.---(695; 19169)</p> <p>.---(354; 18467)</p> <p> `---(123; 26500)</p> <p>---(74; 41)</p> <p>Built treap</p> <p>.---(695; 19169)</p> <p>.---(354; 18467)</p> <p> `---(123; 26500)</p> <p>---(74; 41)</p> <p>Select an action for the item: 3</p> <p>The end of program...</p>
3	1 2 3 4 5 6 7 8 9 10 2	<p>Enter leafs for tree:</p> <p>1 2 3 4 5 6 7 8 9 10</p>

<div data-bbox="300 159 325 338"> 8 1 8 3 </div>	<p>The built treap:</p> <pre> .---(10; 24464) `---(9; 26962) `---(8; 29358) .---(7; 11478) `---(6; 15724) `---(5; 19169) `---(4; 26500) .---(3; 6334) `---(2; 18467) ---(1; 41) </pre> <p>Entry rules:</p> <ul style="list-style-type: none"> 1 - Insert element; 2 - Delete element; 3 - Exit. <p>Select an action for the item: Enter an item to work with: <----Deleting element----> Splitting key - 8</p> <p>Treaps after first split</p> <p>Treap with key ≤ 8</p> <pre> .---(8; 29358) .---(7; 11478) `---(6; 15724) `---(5; 19169) `---(4; 26500) .---(3; 6334) `---(2; 18467) ---(1; 41) </pre> <p>Treap with key > 8</p> <pre> ---(10; 24464) `---(9; 26962) </pre>
--	---

		<p>Treaps after second split</p> <p>Treap with key < 8</p> <pre> .---(7; 11478) `---(6; 15724) `---(5; 19169) `---(4; 26500) .---(3; 6334) `---(2; 18467) ---(1; 41) </pre> <p>Treap with 8 to be deleted</p> <pre> ---(8; 29358) </pre> <p>Priority 41 <= 24464 , go to the right subtree</p> <p>Priority 6334 <= 24464 , go to the right subtree</p> <p>Priority 11478 <= 24464 , go to the right subtree</p> <pre> .---(10; 24464) `---(9; 26962) .---(7; 11478) `---(6; 15724) `---(5; 19169) `---(4; 26500) .---(3; 6334) `---(2; 18467) ---(1; 41) </pre> <p>Built treap</p> <pre> .---(10; 24464) `---(9; 26962) .---(7; 11478) `---(6; 15724) `---(5; 19169) `---(4; 26500) .---(3; 6334) </pre>
--	--	---

		<pre> `---(2; 18467) ---(1; 41) Select an action for the item: Enter an item to work with: <----Inserting element----> Adding: 8 Two treaps after split: Treap with key <= 8 .---(7; 11478) `---(6; 15724) `---(5; 19169) `---(4; 26500) .---(3; 6334) `---(2; 18467) ---(1; 41) Treap with key > 8 ---(10; 24464) `---(9; 26962) Priority 41 <= 5705 , go to the right subtree Priority 6334 > 5705 , go to the left subtree Treap after first merge: .---(8; 5705) .---(7; 11478) `---(6; 15724) `---(5; 19169) `---(4; 26500) `---(3; 6334) `---(2; 18467) ---(1; 41) Priority 41 <= 24464 , go to the right subtree Priority 5705 <= 24464 , go to the right sub- tree Treap after second merge: </pre>
--	--	---

		<pre> .---(10; 24464) `---(9; 26962) .---(8; 5705) .---(7; 11478) `---(6; 15724) `---(5; 19169) `---(4; 26500) `---(3; 6334) `---(2; 18467) ---(1; 41) Built treap .---(10; 24464) `---(9; 26962) .---(8; 5705) .---(7; 11478) `---(6; 15724) `---(5; 19169) `---(4; 26500) `---(3; 6334) `---(2; 18467) ---(1; 41) Select an action for the item: The end of program... </pre>
4	84 473 13 58 1 13 1 12 3	<pre> The built treap: .---(473; 18467) ---(84; 41) .---(58; 26500) `---(13; 6334) Entry rules: 1 - Insert element; 2 - Delete element; 3 - Exit. Select an action for the item: Enter an item to work with: <----Inserting element----> </pre>

		<p>Entered element has already inserted in treap</p> <p>Built treap</p> <pre> .---(473; 18467) ---(84; 41) .---(58; 26500) `---(13; 6334) </pre> <p>Select an action for the item: Enter an item to work with: <----Inserting element----> Adding: 12 Two treaps after split: Treap with key ≤ 12 Treap is empty!</p> <p>Treap with key > 12</p> <pre> .---(473; 18467) ---(84; 41) .---(58; 26500) `---(13; 6334) </pre> <p>Treap after first merge: ---(12; 19169)</p> <p>Priority 19169 > 41 , go to the left subtree Priority 19169 > 6334 , go to the left subtree Treap after second merge:</p> <pre> .---(473; 18467) ---(84; 41) .---(58; 26500) `---(13; 6334) `---(12; 19169) </pre> <p>Built treap</p> <pre> .---(473; 18467) ---(84; 41) .---(58; 26500) </pre>
--	--	--

		<p>`---(13; 6334)</p> <p>`---(12; 19169)</p> <p>Select an action for the item:</p> <p>The end of program...</p>
--	--	---

Заключение

В процессе работы были освоены и закреплены навыки работы с основными алгоритмами и структурами данных, используемыми при работе, рандомизированными пирамидами поиска. При разрешении возникающих проблем в приоритете были поиск и использование наиболее оптимальных подходов. В процессе работы были изучены дополнительные источники, что позволило повысить знания по изучаемой и сопутствующим темам. Программа написана на языке C++.

Приложение

Приложение А. Код основной программы.

```
#include <iostream>
#include <cstring>
#include <sstream>

#include "treap.hpp"

int menu(Treap<int> tree) {           // Function, which is a program
menu
    int value, action = 0;
    std::cout << "Entry rules: " << std::endl;
    std::cout << "\t\t1 - Insert element;\n\t\t2 - Delete element;\n\t\t3
- Exit." << std::endl;
    while (action != 3) {
        std::cout << "Select an action for the item: " << std::endl;
        std::cin >> action;
        if(std::cin.fail()) {         // Check item for correctness
            std::cout << "Error! Items for search can only be of one
type(digits)!" << std::endl;
            return 0;
        }
        switch(action) {
            case 1:
                std::cout << "Enter an item to work with:" << std::endl;
                std::cin >> value;
                if(std::cin.fail()) {
                    std::cout << "Error! Items for search can only be of
one type(digits)!" << std::endl;
                    return 0;
                }
                std::cout << "<----Inserting element---->" << std::endl;
                tree.top = tree.insert(tree.top, value);
                std::cout << "Built treap" << std::endl;
                tree.display(tree.top, nullptr, false);
                break;

            case 2:
                std::cout << "Enter an item to work with:" << std::endl;
                std::cin >> value;
                if(std::cin.fail()) {
                    std::cout << "Error! Items for search can only be of
one type(digits)!" << std::endl;
                    return 0;
                }
                std::cout << "<----Deleting element---->" << std::endl;
                tree.top = tree.erase(tree.top, value);
                std::cout << "Built treap" << std::endl;
```

```

        tree.display(tree.top, nullptr, false);
        break;

    case 3:
        std::cout << "The end of program..." << std::endl;
        break;

    default:
        std::cout << "Error! Enter correct number of action." <<
std::endl;
    }
}
return 0;
}

int main(){
    std::string list;
    std::cout << "Enter leafs for tree:" << std::endl;
    getline(std::cin, list);
    std::cout << std::endl;
    std::stringstream ss;
    for(size_t i = 0; i < list.size(); i++)    //validation of data for
correctness
        if(isalpha(list[i]))
            std::cout << "Error: " << list[i] << " - was not digit! Stop
building the tree..." << std::endl;
    ss.str(list);
    Treap<int> tree;
    int value;
    std::cout << "\t\tWork with tree..." << std::endl;
    while(ss >> value) {
        tree.top = tree.insert(tree.top, value);
    }

    std::cout << "The built treap:" << std::endl;
    tree.display(tree.top, nullptr, false);
    menu(tree);
}

```

Приложение Б. Код заголовочного файла – Рандомизированной пирамиды поиска.

```
#include <iostream>
#include <cstdlib>
#include <utility> //for pair

#define TEST

template <typename Type>
class Treap{
    struct node{
        Type key;
        int priority;
        node * left;
        node * right;
    };
public:
    node * top; //pointer to the root of treap

    Treap(){
        top = nullptr;
    }

    bool exists(node * root, Type val){
        if(root == nullptr)
            return false;
        if(val == root->key)
            return true;
        if(val > root->key)
            return exists(root->right, val);
        return exists(root->left, val);
    }

    std::pair <node *, node *> split(node * root, Type val){ //cutting two
trees by value
        if(root == nullptr)
            return {nullptr, nullptr};
        if(root->key <= val){
            auto res = split(root->right, val);
            root->right = res.first;
            return {root, res.second};
        }
        else{
            auto res = split(root->left, val);
            root->left = res.second;
            return {res.first, root};
        }
    }
}
```

```

    node * merge(node * root1, node * root2){//merging two trees with
priority
    if(root1 == nullptr)
        return root2;
    if(root2 == nullptr)
        return root1;
    if(root1->priority <= root2->priority){
#ifdef TEST
        std::cout << "Priority " << root1->priority << " <= " <<
root2->priority << " , go to the right subtree" << std::endl;
#endif
        root1->right = merge(root1->right, root2);
        return root1;
    }
    else{
#ifdef TEST
        std::cout << "Priority " << root1->priority << " > " <<
root2->priority << " , go to the left subtree" << std::endl;
#endif
        root2->left = merge(root1, root2->left);
        return root2;
    }
}

    node * insert(node * root, Type val){
        if(exists(root, val)){
            std::cout << "Entered element has already inserted in treap"
<< std::endl;
            return root;
        }
#ifdef TEST
        std::cout << "Adding:\t" << val << std::endl;
#endif
        auto res = split(root, val);
#ifdef TEST
        std::cout << "Two treaps after split:" << std::endl;
        std::cout << "Treap with key <= " << val << std::endl;
        display(res.first, nullptr, false);
        std::cout << std::endl << "Treap with key > " << val <<
std::endl;
        display(res.second, nullptr, false);
#endif
        node * newnode = new node;
        newnode->key = val;
        newnode->priority = rand();
        newnode->left = nullptr;
        newnode->right = nullptr;

        node * tmp1 = merge(res.first, newnode);

```

```

#ifdef TEST
    std::cout << "Treap after first merge: " << std::endl;
    display(tmp1, nullptr, false);
#endif
    node * tmp2 = merge(tmp1, res.second);
#ifdef TEST
    std::cout << "Treap after second merge: " << std::endl;
    display(tmp2, nullptr, false);
#endif
    return tmp2;
    // return merge(merge(res.first, newnode), res.second);
}

node * erase(node * root, int val){//
    if(!exists(root, val)){
        std::cout << "Entered element has not inserted in treap"
<< std::endl;
        return root;
    }
#ifdef TEST
    std::cout << "Splitting key - " << val << std::endl <<
std::endl;
#endif
    auto res1 = split(root, val);
#ifdef TEST
    std::cout << "Treaps after first split" <<std::endl <<
std::endl;
    std::cout << "Treap with key <= " << val << std::endl;
    display(res1.first, nullptr, false);
    std::cout << std::endl << "Treap with key > " << val <<
std::endl;
    display(res1.second, nullptr, false);
#endif
    auto res2 = split(res1.first, val - 1);
#ifdef TEST
    std::cout << std::endl << "Treaps after second split"
<<std::endl << std::endl;
    std::cout << "Treap with key < " << val << std::endl;
    display(res2.first, nullptr, false);
    std::cout << "Treap with " << val << " to be deleted" <<
std::endl;
    display(res2.second, nullptr, false);
#endif
    delete res2.second;
    node * tmp = merge(res2.first, res1.second);
#ifdef TEST
    display(tmp, nullptr, false);
#endif
    return tmp;
    //return merge(res2.first, res1.second);
}

```

```

    }

~Treap(){//destructor
    if(top)
        remove(top);
}

void remove(node * root){//remove tree
    if(root == nullptr)
        return;
    remove(root->left);
    remove(root->right);
    delete root;
}

struct Trunk {//structure for printing treap
    Trunk *prev;
    std::string branch;
    Trunk(Trunk *prev, std::string branch) {
        this->prev = prev;
        this->branch = branch;
    }
};

void showTrunks(Trunk *p) {//An auxiliary method for printing
branches of a treap
    if(p == nullptr)
        return;
    showTrunks(p->prev);
    std::cout << p->branch;
}

void inorder(node* t, Trunk* prev, bool isRight) {      //method for
print treap
    if(t == nullptr)
        return;
    std::string prev_str = "    ";
    Trunk *trunk = new Trunk(prev, prev_str);
    inorder(t->right, trunk, true);
    if(!prev)
        trunk->branch = "---";
    else if(isRight) {
        trunk->branch = ".---";
        prev_str = "    |";
    }
    else {
        trunk->branch = "`---";
        prev->branch = prev_str;
    }
    showTrunks(trunk);
}

```

```

        std::cout << "(" <<t->key << "; " << t->priority << ")" <<
std::endl;
        if(prev)
            prev->branch = prev_str;
        trunk->branch = "    |";
        inorder(t->left, trunk, false);
        delete trunk;
    }

    void display(node * root, Trunk * prev, bool isRight) {
//method for print treap
        if(!root) {
            std::cout << "Treap is empty!" << std::endl << std::endl;
            return;
        }
        inorder(root, nullptr, false);
        std::cout << std::endl;
    }
};

```