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

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Алгоритмы и структуры данных»

Тема: Рандомизированное бинарное дерево поиска (RBST)

Студентка гр. 7382 _____ Дерябина П.С.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2018

Задание.

Вариант 11. По заданному файлу F (типа file of Elem), все элементы которого различны, построить RBST неопределенного типа. Для построенного RBST проверить, входит ли в него элемент e типа Elem, и если входит, то удалить элемент e из дерева поиска.

Пояснение к заданию.

Возможные типы входного файла F: float, int, char. В случае несовпадения типа элемента для удаления и типа входного файла, удаление не происходит. В случае несовпадения типов каких-либо элементов входного файла, построение RBST не происходит.

Описание алгоритма.

1. Алгоритм создания RBST.

Последовательно перебираются все ключи файла F, и для каждого создается узел, отвечающий условиям: правый сын больше родителя, а левый сын — меньше. При этом каждая вставка элемента может происходить в корень с вероятностью 1/(количество узлов дерева + 1).

2. Алгоритм удаления элемента

Происходит обход дерева КЛП и при совпадении значений текущего ключа и ключа для удаления — удаляется текущий узел, а два сына этого узла сливаются. Для сохранения свойств рандомизированного дерева при слиянии левый сын будет корневым с вероятностью (количество узлов левого сына)/(количество узлов и правого, и левого сыновей).

Описание функций и структур данных.

Код программы приведен в приложении А.

1. Структура данных Node.

Содержит 4 поля: поле void* key, хранящее значение ключа, поле int size, хранящее размер поддерева, корнем которого является ключ key,

поля left и right, указывающие на левого и правого сыновей.

2. Функция void Space(int level).

Принимает на вход целое число и выводит на консоль количество пробелов, равное 3*level. Используется для вывода рекурсивных вызовов функций.

3. Функция int GetElements(float *keys, char* keys_ch, char *str, size_t fixtype).

Принимает указатель keys на float и указатель keys_ch на char, в один из которых потом поместятся значения ключей файла F; строку str в которой хранятся значения ключей файла F; переменную size, хранящую размер типа char или float. В зависимости от значения переменной size функция будет осуществлять считываний символов в массив keys_ch или в массив keys.

4. Функция int fixsize(Node* tree).

Принимает указатель на структуру Node. Функция присваивает полю size каждого узла дерева tree значение, равное весам узлов в дереве tree.

5. Функции Node* RotateRight(Node *tree, size_t size, int level) и Node* RotateLeft(Node* tree, size_t size, int level).

RotateRight Принимает указатель на Node, который хранит дерево tree; переменную size, хранящую размер типа char или float; переменную level со значение глубины рекурсии функции, которая вызвала RotateRight. Функция осуществляет поворот вправо, благодаря чему левый сын корневого дерева становится корнем дерева tree.

Функция RotateLeft аналогична функции RotateRight.

6. Функция Node* Insert(Node* tree, void* key, size_t size, int level).

Принимает указатель на структуру Node, которая хранит дерево tree, переменную key со значением ключа, который нужно вставить в дерево; переменную size со значением размера char или float; переменную level со значением глубины рекурсии. Функция производит обход КЛП дерева tree, и, если текущий узел пуст, то вставляем в него ключ key.

7. Функция Node* InsertRoot(Node* tree, void* key, size_t size, int level).

Функция InsertRoot аналогична функции Insert, но после вставки в первый пустой узел функции InsertRoot вызывает фукнции RotateRight и/или RotateLeft для того, чтобы поднять узел key из положения листа в положения корня.

8. Функция void showtree(Node* tree, size_t size).

Принимает указатель на структуру Node, которая хранит дерево tree и переменную size, хранящую размер типа char или float. С помощью обхода КЛП дерева tree функция выводит на экран значения текущего корня и его детей.

9. Функция Node* Join(Node* left, Node* right).

Принимает 2 указателя на структуру Node, которые хранят левого и правого сына одного узла. Функция склеивает левого и правого сына, причем корнем результирующего дерева будет будет корень левого сына с вероятностью (количество узлов левого сына)/(количество узлов правого и левого сыновей).

10. Функция Node* Remove(Node* tree, size_t size, char* to_delete, int* flag, int level).

Принимает указатель на структуру Node, которая хранит дерево tree и переменную size, хранящую размер типа char или float, строку to_delete со значением ключа, который нужно удалить; переменную flag, которая будет равна 0, если ключа для удаления нет в дереве и 2 — если типы ключа для удаления и дерева не совпадают. Функция совершает обход КЛП дерева tree и удаляет если ключ найден, то происходит его удаление, а левые и правые сыновья склеиваются функцией Join.

11. Функция void delete_BT(Node* tree).

Принимает указатель на структуру Node, которая хранит дерево tree и удаляет все его узлы, совершая обход ЛПК.

Тестирование

Для более наглядной демонстрации работы программы был создан ряд тестов и bash-скрипт, последовательно выводящий содержимое очередного теста и результат работы программы для этого теста. Код bashскрипта и его работа представлены в приложении Б.

Рассмотрим тест 1:

Входные данные:

```
2
a
a b c d e
```

Выходные данные:

```
Insert[level 0]: push [a]
Insert[level 0]: b > a [1]
  Insert[level 1]: push [b]
Insert[level 0]: in root
InsertRoot[level 0]: c > a
  InsertRoot[level 1]: c > b
    InsertRoot[level 2]: push [c]
    RotateLeft
  RotateLeft
Insert[level 0]: d > c [1]
  Insert[level 1]: push [d]
Insert[level 0]: e > c [2]
  Insert[level 1]: e > d [1]
    Insert[level 2]: push [e]


[c]  left: [a]  right: [d]
[a]  left: [null]  right: [b]
[b]  left: [null]  right: [null]
```

[d] left: [null] right: [e]

[e] left: [null] right: [null]

Remove[level 0]:

  Remove[level 1]: delete [a]

RBST after deletion :

[c] left: [b] right: [d]

[b] left: [null] right: [null]

[d] left: [null] right: [e]

[e] left: [null] right: [null]

В ходе работы программы для каждого символа из последовательности символов создаются узлы RBST, причем для некоторых с известной вероятностью происходит вставка в корень, такими узлом оказался узел «c». Сначала его вставили обычным образом, а после двух поворотов налево (так как в обоих случаях узел «c» спускался по правым сыновьям) узел стал корнем дерева.

Далее произошло удаление узла «a».

Выводы.

В ходе работы были изучены рандомизированные деревья бинарного поиска и метода удаления элементов из них.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД

```
     #include <stdio.h>
     #include <stdlib.h>
     #include <ctype.h>
     #include <string.h>
     #include <time.h>
     #include "API.h"
     #define  N 1000

     // function to show recursion's calls
     void Space(int level){

          for(int i = 0; i<level; i++)
               printf("   ");
          return;
     }

     // function to get char or float elements from srint str
     int GetElements(float *keys, char* keys_ch, char *str,
size_t fixtype){

          int size = 0;

          if (str[strlen(str)-1] == '\n')   // remove '\n' if
it exists
               str[strlen(str)-1] = '\0';

             if (fixtype == sizeof(char)){ // char case

                    for(int i = 0; i<strlen(str); i++){

                         if (isalpha(str[i])){  // save
char lelemte
                                 keys_ch[size] = str[i];
                                 size += 1;
                        if (size >= N){
                             printf("Max number of elements:
%d!\n", N);
                             return 0;
                        }
                         }

                         else if (str[i] == ' ' || str[i]
== '\n')  // skip spaces
                                 continue;

                         else{  // if string contains
wrong symbols
                                 printf("Wrong input!\n");
```

```
                                        return 0;
                                }
                        }
                }
                if (fixtype == sizeof(float)){ // float case

                        char* end = str;

                        while(*str){

                                keys[size] = strtod(str,
&end); // save float element
                                size += 1;
                        str = end;

                        if (size >= N){
                                        printf("Max number of
elements: %d!\n", N);
                                        return 0;
                                }

                        while(!isdigit(*str) && *str){  // skip
symbols that are not digits
                                str++;
                        }
                       }
          }
          return size;
     }

     // fixing number of elements in tree
     int fixsize(Node* tree){

          if (tree == NULL){
               return 0;
          }

          tree->size = fixsize(tree->left)+fixsize(tree-
>right) + 1; // number elements of subtrees + 1 = number
elements of tree
          return tree->size;
     }

     Node* RotateRight(Node *tree, size_t size, int level){
          Space(level);
          printf("RotateRight\n");

          Node* tmp = tree->left;   // head of rotated tree is
head of left subtree of tree

          if (tmp == NULL)
```

```
                return tree;

          tree->left = tmp->right;  // replace left subtree
of tree with right subtree of rotated tree
          tmp->right = tree;        // replace right subtree
of rotated tree with tree
          tmp->size = tree->size + 1;
          fixsize(tree);

          return tmp;
     }

     Node* RotateLeft(Node* tree, size_t size, int level){
          Space(level);
          printf("RotateLeft\n");

          Node* tmp = tree->right;

          if (tmp == NULL)
               return tree;

          if (size == sizeof(char) && *(char*)(tree->right-
>key) == *(char*)(tree->key)) // if keys of element and its
right son is equal (char)
               return tree;

          if (size == sizeof(float) && *(float*)(tree->right-
>key) == *(float*)(tree->key)) // if keys of element and its
right son is equal (float)
               return tree;

          tree->right = tmp->left;
          tmp->left = tree;
          tmp->size = tree->size;
          fixsize(tree);

          return tmp;
     }

     Node* InsertRoot(Node* tree, void* key, size_t size, int
level){

          Space(level);
          printf("InsertRoot[level %d]: ", level);

          if (tree == NULL){  // make new elem if current node
is empty

               tree = malloc(sizeof(Node));
               tree->key = key;
               tree->size = 1;
```

```
                tree->left = tree->right = NULL;
                fixsize(tree);

                if (size == sizeof(char))
                                printf("push [%c]\n",
*(char*)key);

                      if (size == sizeof(float))
                              printf("push [%g]\n",
*(float*)key);

                return tree;
          }
          if (size == sizeof(float)){ // float case

                float num_key = *(float*)key; // saving float
value of key

                if (num_key < *(float*)(tree->key)){  // if key
< key of current node make recursion call for left subtree
                      printf("%g < %g\n", num_key, *(float*)
(tree->key));
                      tree->left = InsertRoot(tree->left, key,
size, level+1);
                      tree = RotateRight(tree, size,
level+1);  // rotation right to move key in root

                      return tree;
                }

                else {  // else - make recursion call for right
subtree
                      printf("%g >= %g\n", num_key, *(float*)
(tree->key));
                      tree->right = InsertRoot(tree->right, key,
size, level+1);
                      tree = RotateLeft(tree, size,
level+1);  // rotation left to move key in root

                      return tree;
                }
          }
          if (size == sizeof(char)){ // char case
                char ch_key = *(char*)key;  // saving char
value of key
                char tree_key = *(char*)(tree->key);

                      if ((int)ch_key - (int)tree_key < 0){

                      printf("%c < %c\n", ch_key, tree_key);
```

```
                              tree->left = InsertRoot(tree-
>left, key, size, level+1);
                              tree = RotateRight(tree, size,
level+1);

                    return tree;
                     }
                     else if ((int)ch_key - (int)tree_key >=
0){

                    printf("%c >= %c\n", ch_key, tree_key);
                              tree->right = InsertRoot(tree-
>right, key, size, level+1);
                              tree = RotateLeft(tree, size,
level+1);

                    return tree;
                     }

          }
     }

     Node* Insert(Node* tree, void* key, size_t size, int
level){

          Space(level);
             printf("Insert[level %d]: ", level);

          if (tree == NULL){

               tree = malloc(sizeof(Node));
               tree->key = key;
               tree->size = 1;
               tree->left = tree->right = NULL;
               fixsize(tree);

               if (size == sizeof(char))
                    printf("push [%c]\n", *(char*)key);

               if (size == sizeof(float))
                    printf("push [%g]\n", *(float*)key);

               return tree;
          }

          if (rand()%(tree->size+1) == 0 && level == 0){  //
if got chance for root inserting (1 out of tree->size + 1)

               printf("in root\n");
               return InsertRoot(tree, key, size, 0); //insert
in root
```

```
          }

          else if (size == sizeof(float)){  // float case
               float num_key = *(float*)key;

               if (num_key < *(float*)(tree->key)){
                    printf("%g < %g\n", num_key, *(float*)
(tree->key));
                    tree->left = Insert(tree->left, key, size,
level+1);
               }

               else{
                    printf("%g >= %g\n", num_key, *(float*)
(tree->key));
                    tree->right = Insert(tree->right, key,
size, level+1);
               }
          }

          else if (size == sizeof(char)){   // char case

               char ch_key = *(char*)key;
                     char tree_key = *(char*)(tree->key);

                     if ((int)ch_key - (int)tree_key < 0){

                               printf("%c < %c  [%d]\n", ch_key,
tree_key, (int)ch_key - (int)tree_key);
                               tree->left = Insert(tree->left,
key, size, level+1);
                     }

                     else if ((int)ch_key - (int)tree_key >=
0){

                               printf("%c >= %c [%d]\n", ch_key,
tree_key, (int)ch_key - (int)tree_key);
                               tree->right = Insert(tree->right,
key, size, level+1);
                     }
             }

          fixsize(tree);
          return tree;

     }

     void showtree(Node* tree, size_t size){
          if (tree == NULL){
               printf("Tree is empty!\n");
```

```
                return;
          }

          if (size == sizeof(float)){
                printf("[%g]  ", *(float*)(tree->key));

                if (tree->left != NULL)
                     printf("left: [%g]  ", *(float*)(tree-
>left->key));
                else
                     printf("left: [null]  ");

                if (tree->right != NULL)
                     printf("right: [%g]\n", *(float*)(tree-
>right->key));
                else
                     printf("right: [null]\n");

                if (tree->left != NULL)
                     showtree(tree->left, size);  // recursion
call for left subtree

                if (tree->right != NULL)
                     showtree(tree->right, size); // recursion
call for right subtree

                return;
          }

          if (size == sizeof(char)){
                      printf("[%c]  ", *(char*)(tree->key));

                      if (tree->left != NULL)
                              printf("left: [%c]  ", *(char*)
(tree->left->key));
                      else
                              printf("left: [null]  ");

                      if (tree->right != NULL)
                              printf("right: [%c]\n", *(char*)
(tree->right->key));
                      else
                              printf("right: [null]\n");

                      if (tree->left != NULL)
                              showtree(tree->left, size);

                      if (tree->right != NULL)
                              showtree(tree->right, size);

                return;
```

```
          }
     }


     Node* Join(Node* left, Node* right){
          if (left == NULL)
               return right;

          if (right == NULL)
               return left;

          if (rand()%(left->size + right->size) < left->size){
// if left tree got chanse to be the root
               left->right = Join(left->right, right);
                     fixsize(left);
                     return left;
          }

          else{ // else - right tree is root
               right->left = Join(left, right->left);
               fixsize(right);
               return right;
          }
     }

     Node* Remove(Node* tree,  size_t size, char* to_delete,
int* flag, int level)
     {
          if(tree == NULL )
                     return NULL;

          void* key;

          if(isalpha(to_delete[0]) && size == sizeof(char))
{ // if type of RBST and element for deletion are char
                     key = &to_delete[0];
             }

             else if (isdigit(to_delete[0]) && size ==
sizeof(float)){  // if type of RBST and element for deletion
are float
                     float num = strtod(to_delete, NULL);
                     key = &num;
             }
             else{ //  if type of RBST and element for
deletion aren't same
                     printf("Type of element for deletion
doesn't match with type of BT! [%c]\n", *(char*)(tree->key));
               *flag = 2;
                     return tree;
             }
```

```
          Space(level);
          printf("Remove[level %d]:  ", level);

          if (size == sizeof(float)){
               float num_key = *(float*)(key);

               if(*(float*)(tree->key) == num_key){ // if key
is found
                    *flag = 1;
                    Node* new_tree = Join(tree->left, tree-
>right); // join left and right subtree
                    printf("delete [%g]\n", *(float*)(tree-
>key));
                    free(tree);
                    fixsize(new_tree);

                    return new_tree;
               }

               printf("\n");

               if (num_key < *(float*)(tree->key))  // if key
< current key, then search in left subtree
                    tree->left = Remove(tree->left, size,
to_delete, flag, level+1);

               else // else - search in right subtree
                    tree->right = Remove(tree->right, size,
to_delete, flag, level+1);
          }
          if (size == sizeof(char)){

                     char ch_key = *(char*)(key);

                     if(*(char*)(tree->key) == ch_key){
                    *flag = 1;
                              Node* new_tree = Join(tree->left,
tree->right);
                    printf("delete [%c]\n", *(char*)(tree-
>key));
                    free(tree);
                    fixsize(new_tree);

                              return new_tree;
                     }

               printf("\n");

                     if (ch_key < *(char*)(tree->key))
```

```
                              tree->left = Remove(tree->left,
size, to_delete, flag, level+1);

                    else
                              tree->right = Remove(tree->right,
size, to_delete, flag, level+1);
          }

          return tree;

     }

     void delete_BT(Node* tree){

          if (tree == NULL)
                    return;

          if (tree->left != NULL)
               delete_BT(tree->left);
          if (tree->right != NULL)
               delete_BT(tree->right);

          free(tree);
          return;
     }



     #include <stdio.h>
     #include <stdlib.h>
     #include <ctype.h>
     #include <string.h>
     #include <time.h>
     #include "API.h"

     #define N 1000


     int main(){
          srand(time(NULL)); // new settings for random number

          char *str = malloc(sizeof(char)*N*2);  // string for
keys of RBST
          char *keys_ch = NULL;  // for char keys
          float *keys = NULL;    // for float keys
          int size = 0;          // number of keys
          char to_delete[20];    // key for deletion
          void* key;
          int flag=0;
          int format;            // format of input
          size_t fixtype = 0;    // type of keys
```

```
          printf("Hello! This is randomized binary search tree
maker (RBST)!\nAvailable type of elements: char, float,
int\n");
          printf("All elements of one RBST must be the
same.\n");
          printf("Type '1', if you want input data from file
'input.txt', '2' - from the keyboard: ");
             scanf("%d", &format);
             char c = getchar(); // takes '\n'

             if (format != 1 && format != 2) {
                printf("Wrong input format!\n");
                return 0;
          }

          else if (format == 1) { // input from file

                  FILE* f;

               if ((f = fopen("input.txt", "r")) == NULL) {
                         printf("Couldn't open input.txt\n");
                         return 0;
               }

               fgets(to_delete, 18, f);
               fgets(str, N-1, f);
               fclose(f);
          }
          else if (format == 2){

               printf("Input elemet for deletion: ");
               fgets(to_delete, 18, stdin);
               printf("Input elements of RBST: ");
               fgets(str, N-1, stdin);
          }

          printf("You entered: %s\n", str);

          if (isalpha(str[0])){
                    fixtype = (sizeof(char));
                    keys_ch = malloc(N*sizeof(*keys_ch));
             }

             else if (isdigit(str[0])){
                    fixtype = (sizeof(float));
                    keys = calloc(N, sizeof(float));
             }

          else{
               printf("wrong input!\n");
```

```
            return 0;
        }

        Node *tree = NULL;

        size = GetElements(keys, keys_ch, str, fixtype);

        for(int i = 0; i < size; i++){

            if (fixtype == sizeof(float)){
                key = &keys[i]; // turn to void*
            }

            if (fixtype == sizeof(char)){
                key = &keys_ch[i]; // turn to void*
            }

            tree = Insert(tree, key, fixtype, 0);
        }

        printf("\n\n");
        showtree(tree, fixtype);
        printf("\n\n");

        tree = Remove(tree, fixtype, to_delete, &flag, 0);

        if (flag != 0 && flag != 2){  // if element for
deletion is deleted
            printf("\nRBST after deletion :\n\n");
                showtree(tree, fixtype);
        }

        if (flag == 0){
            printf("Element for deletion isn't in
RBST!\n");
        }

        free(str);
        free(keys);
        free(keys_ch);
        delete_BT(tree);

        return 0;
    }
```

ПРИЛОЖЕНИЕ Б

КОД СКРИПТА

```
#!/bin/bash

make

echo -e '\n\n\033[7mTest1:\033[0m'
cat ./Tests/test1.txt
echo -e 'Result:'
./lab5.exe < ./Tests/test1.txt

echo -e '\n\n\033[7mTest2:\033[0m'
cat ./Tests/test2.txt
echo -e 'Result:'
./lab5.exe < ./Tests/test2.txt

echo -e '\n\n\033[7mTest3:\033[0m'
cat ./Tests/test3.txt
echo -e 'Result:'
./lab5.exe < ./Tests/test3.txt

echo -e '\n\n\033[7mTest4:\033[0m'
cat ./Tests/test4.txt
echo -e 'Result:'
./lab5.exe < ./Tests/test4.txt

echo -e '\n\n\033[7mTest5:\033[0m'
cat ./Tests/test5.txt
echo -e 'Result:'
./lab5.exe < ./Tests/test5.txt

echo -e '\n\n\033[7mTest6:\033[0m'
cat ./Tests/test6.txt
echo -e 'Result:'
./lab5.exe < ./Tests/test6.txt

echo -e '\n\n\033[7mTest7:\033[0m'
cat ./Tests/test7.txt
echo -e 'Result:'
./lab5.exe < ./Tests/test7.txt

echo -e '\n\n\033[7mTest8:\033[0m'
cat ./Tests/test8.txt
echo -e 'Result:'
./lab5.exe < ./Tests/test8.txt

echo -e '\n\n\033[7mTest9:\033[0m'
cat ./Tests/test9.txt
echo -e 'Result:'
./lab5.exe < ./Tests/test9.txt
```

```
echo -e '\n\n\033[7mTest10:\033[0m'
cat ./Tests/test10.txt
echo -e 'Result:'
./lab5.exe < ./Tests/test10.txt
```

РЕЗУЛЬТАТ РАБОТЫ СКРИПТА

```
Test1:
2
a
a b c d e
Result:
Hello! This is randomized binary search tree maker (RBST)!
Available type of elements: char, float, int
All elements of one RBST must be the same.
Type '1', if you want input data from file 'input.txt', '2' - from the keyboard: Input elemet for deletion: Input elements of RBST: You entered: a b c d e

Insert[level 0]: push [a]
Insert[level 0]: b >= a [1]
   Insert[level 1]: push [b]
Insert[level 0]: c >= a [2]
   Insert[level 1]: c >= b [1]
      Insert[level 2]: push [c]
Insert[level 0]: d >= a [3]
   Insert[level 1]: d >= b [2]
      Insert[level 2]: d >= c [1]
         Insert[level 3]: push [d]
Insert[level 0]: e >= a [4]
   Insert[level 1]: e >= b [3]
      Insert[level 2]: e >= c [2]
         Insert[level 3]: e >= d [1]
            Insert[level 4]: push [e]


[a]  left: [null]  right: [b]
[b]  left: [null]  right: [c]
[c]  left: [null]  right: [d]
[d]  left: [null]  right: [e]
[e]  left: [null]  right: [null]


Remove[level 0]:  delete [a]

RBST after deletion :

[b]  left: [null]  right: [c]
[c]  left: [null]  right: [d]
[d]  left: [null]  right: [e]
[e]  left: [null]  right: [null]


Test2:
2
```

```
     d
     a b c d e f
     Result:
     Hello!  This  is  randomized  binary  search  tree  maker
(RBST)!
     Available type of elements: char, float, int
     All elements of one RBST must be the same.
     Type '1', if you want input data from file 'input.txt',
'2'  -  from  the  keyboard:  Input  elemet  for  deletion:  Input
elements of RBST: You entered: a b c d e f

     Insert[level 0]: push [a]
     Insert[level 0]: b >= a [1]
        Insert[level 1]: push [b]
     Insert[level 0]: c >= a [2]
        Insert[level 1]: c >= b [1]
           Insert[level 2]: push [c]
     Insert[level 0]: d >= a [3]
        Insert[level 1]: d >= b [2]
           Insert[level 2]: d >= c [1]
              Insert[level 3]: push [d]
     Insert[level 0]: e >= a [4]
        Insert[level 1]: e >= b [3]
           Insert[level 2]: e >= c [2]
              Insert[level 3]: e >= d [1]
                 Insert[level 4]: push [e]
     Insert[level 0]: f >= a [5]
        Insert[level 1]: f >= b [4]
           Insert[level 2]: f >= c [3]
              Insert[level 3]: f >= d [2]
                 Insert[level 4]: f >= e [1]
                    Insert[level 5]: push [f]


     [a]  left: [null]  right: [b]
     [b]  left: [null]  right: [c]
     [c]  left: [null]  right: [d]
     [d]  left: [null]  right: [e]
     [e]  left: [null]  right: [f]
     [f]  left: [null]  right: [null]


     Remove[level 0]:
        Remove[level 1]:
           Remove[level 2]:
              Remove[level 3]:  delete [d]

     RBST after deletion :

     [a]  left: [null]  right: [b]
     [b]  left: [null]  right: [c]
```

```
[c]  left: [null]  right: [e]
[e]  left: [null]  right: [f]
[f]  left: [null]  right: [null]


Test3:
2
100
2 3.3 3.33 11 100 5
Result:
Hello! This is randomized binary search tree maker (RBST)!
Available type of elements: char, float, int
All elements of one RBST must be the same.
Type '1', if you want input data from file 'input.txt', '2' - from the keyboard: Input elemet for deletion: Input elements of RBST: You entered: 2 3.3 3.33 11 100 5

Insert[level 0]: push [2]
Insert[level 0]: 3.3 >= 2
   Insert[level 1]: push [3.3]
Insert[level 0]: 3.33 >= 2
   Insert[level 1]: 3.33 >= 3.3
      Insert[level 2]: push [3.33]
Insert[level 0]: 11 >= 2
   Insert[level 1]: 11 >= 3.3
      Insert[level 2]: 11 >= 3.33
         Insert[level 3]: push [11]
Insert[level 0]: 100 >= 2
   Insert[level 1]: 100 >= 3.3
      Insert[level 2]: 100 >= 3.33
         Insert[level 3]: 100 >= 11
            Insert[level 4]: push [100]
Insert[level 0]: 5 >= 2
   Insert[level 1]: 5 >= 3.3
      Insert[level 2]: 5 >= 3.33
         Insert[level 3]: 5 < 11
            Insert[level 4]: push [5]


[2]  left: [null]  right: [3.3]
[3.3]  left: [null]  right: [3.33]
[3.33]  left: [null]  right: [11]
[11]  left: [5]  right: [100]
[5]  left: [null]  right: [null]
[100]  left: [null]  right: [null]


Remove[level 0]:
   Remove[level 1]:
      Remove[level 2]:
```

```
            Remove[level 3]:
               Remove[level 4]:  delete [100]

     RBST after deletion :

     [2]  left: [null]  right: [3.3]
     [3.3]  left: [null]  right: [3.33]
     [3.33]  left: [null]  right: [11]
     [11]  left: [5]  right: [null]
     [5]  left: [null]  right: [null]


     Test4:
     2
     5.5
     6.7 6.1 7.2 5.5 3 100 200
     Result:
     Hello!  This  is  randomized  binary  search  tree  maker
(RBST)!
     Available type of elements: char, float, int
     All elements of one RBST must be the same.
     Type '1', if you want input data from file 'input.txt',
'2'  -  from  the  keyboard:  Input  elemet  for  deletion:  Input
elements of RBST: You entered: 6.7 6.1 7.2 5.5 3 100 200

     Insert[level 0]: push [6.7]
     Insert[level 0]: 6.1 < 6.7
        Insert[level 1]: push [6.1]
     Insert[level 0]: 7.2 >= 6.7
        Insert[level 1]: push [7.2]
     Insert[level 0]: in root
     InsertRoot[level 0]: 5.5 < 6.7
        InsertRoot[level 1]: 5.5 < 6.1
           InsertRoot[level 2]: push [5.5]
           RotateRight
        RotateRight
     Insert[level 0]: 3 < 5.5
        Insert[level 1]: push [3]
     Insert[level 0]: 100 >= 5.5
        Insert[level 1]: 100 >= 6.7
           Insert[level 2]: 100 >= 7.2
              Insert[level 3]: push [100]
     Insert[level 0]: 200 >= 5.5
        Insert[level 1]: 200 >= 6.7
           Insert[level 2]: 200 >= 7.2
              Insert[level 3]: 200 >= 100
                 Insert[level 4]: push [200]


     [5.5]  left: [3]  right: [6.7]
     [3]  left: [null]  right: [null]
```

```
[6.7]  left: [6.1]  right: [7.2]
[6.1]  left: [null]  right: [null]
[7.2]  left: [null]  right: [100]
[100]  left: [null]  right: [200]
[200]  left: [null]  right: [null]


Remove[level 0]:  delete [5.5]

RBST after deletion :

[6.7]  left: [6.1]  right: [7.2]
[6.1]  left: [3]  right: [null]
[3]  left: [null]  right: [null]
[7.2]  left: [null]  right: [100]
[100]  left: [null]  right: [200]
[200]  left: [null]  right: [null]


Test5:
2
g
h g f e d c b a
Result:
Hello!  This  is  randomized  binary  search  tree  maker
(RBST)!
Available type of elements: char, float, int
All elements of one RBST must be the same.
Type '1', if you want input data from file 'input.txt',
'2'  -  from  the  keyboard:  Input  elemet  for  deletion:  Input
elements of RBST: You entered: h g f e d c b a

Insert[level 0]: push [h]
Insert[level 0]: g < h  [-1]
   Insert[level 1]: push [g]
Insert[level 0]: f < h  [-2]
   Insert[level 1]: f < g  [-1]
      Insert[level 2]: push [f]
Insert[level 0]: e < h  [-3]
   Insert[level 1]: e < g  [-2]
      Insert[level 2]: e < f  [-1]
         Insert[level 3]: push [e]
Insert[level 0]: d < h  [-4]
   Insert[level 1]: d < g  [-3]
      Insert[level 2]: d < f  [-2]
         Insert[level 3]: d < e  [-1]
            Insert[level 4]: push [d]
Insert[level 0]: c < h  [-5]
   Insert[level 1]: c < g  [-4]
      Insert[level 2]: c < f  [-3]
         Insert[level 3]: c < e  [-2]
```

```
            Insert[level 4]: c < d  [-1]
               Insert[level 5]: push [c]
Insert[level 0]: b < h  [-6]
   Insert[level 1]: b < g  [-5]
      Insert[level 2]: b < f  [-4]
         Insert[level 3]: b < e  [-3]
            Insert[level 4]: b < d  [-2]
               Insert[level 5]: b < c  [-1]
                  Insert[level 6]: push [b]
Insert[level 0]: in root
InsertRoot[level 0]: a < h
   InsertRoot[level 1]: a < g
      InsertRoot[level 2]: a < f
         InsertRoot[level 3]: a < e
            InsertRoot[level 4]: a < d
               InsertRoot[level 5]: a < c
                  InsertRoot[level 6]: a < b
                     InsertRoot[level 7]: push [a]
                     RotateRight
                  RotateRight
               RotateRight
            RotateRight
         RotateRight
      RotateRight
   RotateRight


[a]  left: [null]  right: [h]
[h]  left: [g]  right: [null]
[g]  left: [f]  right: [null]
[f]  left: [e]  right: [null]
[e]  left: [d]  right: [null]
[d]  left: [c]  right: [null]
[c]  left: [b]  right: [null]
[b]  left: [null]  right: [null]


Remove[level 0]:
   Remove[level 1]:
      Remove[level 2]:  delete [g]

RBST after deletion :

[a]  left: [null]  right: [h]
[h]  left: [f]  right: [null]
[f]  left: [e]  right: [null]
[e]  left: [d]  right: [null]
[d]  left: [c]  right: [null]
[c]  left: [b]  right: [null]
[b]  left: [null]  right: [null]
```

```
Test6:
2
5
a z h g b o
Result:
Hello! This is randomized binary search tree maker (RBST)!
Available type of elements: char, float, int
All elements of one RBST must be the same.
Type '1', if you want input data from file 'input.txt', '2' - from the keyboard: Input elemet for deletion: Input elements of RBST: You entered: a z h g b o

Insert[level 0]: push [a]
Insert[level 0]: z >= a [25]
   Insert[level 1]: push [z]
Insert[level 0]: h >= a [7]
   Insert[level 1]: h < z  [-18]
      Insert[level 2]: push [h]
Insert[level 0]: g >= a [6]
   Insert[level 1]: g < z  [-19]
      Insert[level 2]: g < h  [-1]
         Insert[level 3]: push [g]
Insert[level 0]: b >= a [1]
   Insert[level 1]: b < z  [-24]
      Insert[level 2]: b < h  [-6]
         Insert[level 3]: b < g  [-5]
            Insert[level 4]: push [b]
Insert[level 0]: o >= a [14]
   Insert[level 1]: o < z  [-11]
      Insert[level 2]: o >= h [7]
         Insert[level 3]: push [o]


[a]  left: [null]  right: [z]
[z]  left: [h]  right: [null]
[h]  left: [g]  right: [o]
[g]  left: [b]  right: [null]
[b]  left: [null]  right: [null]
[o]  left: [null]  right: [null]


Type of element for deletion doesn't match with type of BT! [a]


Test7:
2
a
2 3 105 1 39
```

```
     Result:
     Hello!  This  is  randomized  binary  search  tree  maker
(RBST)!
     Available type of elements: char, float, int
     All elements of one RBST must be the same.
     Type '1', if you want input data from file 'input.txt',
'2'  -  from  the  keyboard:  Input  elemet  for  deletion:  Input
elements of RBST: You entered: 2 3 105 1 39

     Insert[level 0]: push [2]
     Insert[level 0]: 3 >= 2
        Insert[level 1]: push [3]
     Insert[level 0]: 105 >= 2
        Insert[level 1]: 105 >= 3
           Insert[level 2]: push [105]
     Insert[level 0]: 1 < 2
        Insert[level 1]: push [1]
     Insert[level 0]: 39 >= 2
        Insert[level 1]: 39 >= 3
           Insert[level 2]: 39 < 105
              Insert[level 3]: push [39]


     [2]  left: [1]  right: [3]
     [1]  left: [null]  right: [null]
     [3]  left: [null]  right: [105]
     [105]  left: [39]  right: [null]
     [39]  left: [null]  right: [null]


     Type  of  element  for  deletion  doesn't  match  with  type  of
BT! []


     Test8:
     2
     a
     b c d 3 e f
     Result:
     Hello!  This  is  randomized  binary  search  tree  maker
(RBST)!
     Available type of elements: char, float, int
     All elements of one RBST must be the same.
     Type '1', if you want input data from file 'input.txt',
'2'  -  from  the  keyboard:  Input  elemet  for  deletion:  Input
elements of RBST: You entered: b c d 3 e f

     Wrong input!


     Tree is empty!
```

```
     Element for deletion isn't in RBST!


     Test9:
     2
     a
     a
     Result:
     Hello!  This  is  randomized  binary  search  tree  maker
(RBST)!
     Available type of elements: char, float, int
     All elements of one RBST must be the same.
     Type '1', if you want input data from file 'input.txt',
'2'  -  from  the  keyboard:  Input  elemet  for  deletion:  Input
elements of RBST: You entered: a

     Insert[level 0]: push [a]


     [a]  left: [null]  right: [null]


     Remove[level 0]:  delete [a]

     RBST after deletion :

     Tree is empty!


     Test10:
     2
     Result:
     Hello!  This  is  randomized  binary  search  tree  maker
(RBST)!
     Available type of elements: char, float, int
     All elements of one RBST must be the same.
     Type '1', if you want input data from file 'input.txt',
'2'  -  from  the  keyboard:  Input  elemet  for  deletion:  Input
elements of RBST: You entered:
     wrong input!
```