**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

**отчет**

**по лабораторной работе №3**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «Двоичные деревья»**

**Вариант 1**

|  |  |  |
| --- | --- | --- |
| Студент гр. 0302 |  | Чёрный Я.И. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2021

**Оглавление**

[1 Постановка задачи 3](#_Toc28012475)

[1.1 Реализуемые классы и методы: 3](#_Toc28012476)

[1.1.1 TreeNode: 3](#_Toc28012477)

[1.1.2 Iterator: 3](#_Toc28012478)

[1.1.3 TreeDFTIterator: 3](#_Toc28012479)

[1.1.4 TreeBFTIterator: 4](#_Toc28012480)

[1.1.5 Tree: 4](#_Toc28012481)

[2 Unit-тесты 5](#_Toc28012482)

[3 Пример работы 6](#_Toc28012483)

[3.1 Функция main, представляющая пример использования написанной структуры данных: 6](#_Toc28012484)

[3.2 Результат выполнения: 6](#_Toc28012485)

[4 Текст программы: 7](#_Toc28012486)

[5 Вывод 11](#_Toc28012487)

# Постановка задачи

Целью данной задачи является реализация структуры двоичного дерева поиска с возможностью поиска элемента по ключу, вставки и удаления произвольных ключей. Дополнительно необходимо реализовать для структуры итераторы обхода дерева в ширину и в глубину.

## Реализуемые классы и методы:

### TreeNode:

|  |  |
| --- | --- |
| Поле | Назначение |
| TreeNode\* right, \* left | Указатели на левое и правое поддерево |
| int key | Ключ узла |

|  |  |  |
| --- | --- | --- |
| Функция | Назначение | Временная сложность |
| void SetRight(TreeNode\* rightElement) | Записывает указатель правого поддерева | O(1) |
| void SetLeft(TreeNode\* leftElement) | Записывает указатель левого поддерева | O(1) |
| void SetKey(int key) | Записывает значение ключа | O(1) |
| TreeNode\* GetRight() | Возвращает указатель правого поддерева | O(1) |
| TreeNode\* GetLeft() | Возвращает указатель левого поддерева | O(1) |
| int GetKey() | Возвращает ключ узла | O(1) |

### Iterator:

|  |  |  |
| --- | --- | --- |
| Функция | Назначение | Временная сложность |
| virtual int Next() | Возвращает следующий элемент | - |
| virtual bool HasNext() | Проверяет, существует ли следующий элемент | - |

### TreeDFTIterator:

|  |  |
| --- | --- |
| Поле | Назначение |
| Stack<TreeNode\*> stack | Стек для хранения элементов дерева |

|  |  |  |
| --- | --- | --- |
| Функция | Назначение | Временная сложность |
| void pushToLeft(TreeNode\* node) | Записывает в стек все левые узлы данного поддерева | O(N) |
| int Next() override | Возвращает следующий элемент | O(N) |
| bool HasNext() override | Проверяет, существует ли следующий элемент | O(1) |

### TreeBFTIterator:

|  |  |
| --- | --- |
| Поле | Назначение |
| Queue<TreeNode\*> queue | Очередь для хранения элементов дерева |

|  |  |  |
| --- | --- | --- |
| Функция | Назначение | Временная сложность |
| int Next() override | Возвращает следующий элемент | O(1) |
| bool HasNext() override | Проверяет, существует ли следующий элемент | O(1) |

### Tree:

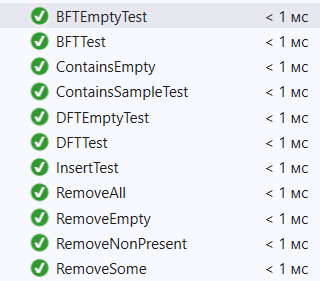
|  |  |
| --- | --- |
| Поле | Назначение |
| TreeNode\* root | Указатель на корень дерева |
| int size; | Кол-во узлов дерева |

|  |  |  |
| --- | --- | --- |
| Функция | Назначение | Временная сложность |
| TreeNode\* deleteRecursive(TreeNode\* root, int key) | Рекурсивный поиск и удаление узла | O(N) |
| int minValue(TreeNode\* root) | Возвращает узел с наименьшим ключом поддерева | O(N) |
| void print(TreeNode\* root, int depth = 0, int pos = 0) | Рекурсивный вывод дерева в консоль | O(N) |
| Iterator\* CreateDFTIterator() | Возвращает новый итератор обхода в глубину | O(1) |
| Iterator\* CreateBFTIterator() | Возвращает новый итератор обхода в ширину | O(1) |
| int GetSize() | Возвращает кол-во узлов дерева | O(1) |
| TreeNode\* GetRoot() | Возвращает указатель на корень дерева | O(1) |
| bool Contains(int key) | Определяет, существует ли узел с заданным узлом | O(N) |
| void Insert(int key) | Вставляет новый узел в дерево по ключу | O(N) |

# Unit-тесты

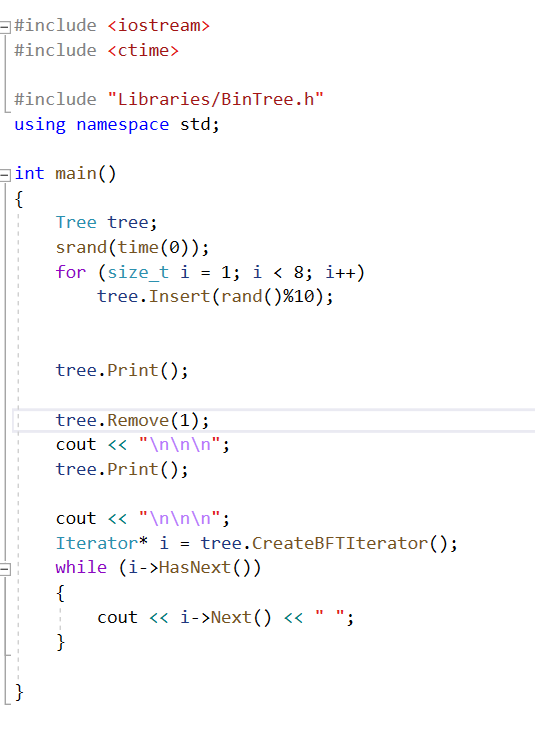
Для проверки работоспособности программы были реализованы юнит-тесты:

|  |  |
| --- | --- |
| Название теста | Назначение |
| InsertTest | Проверка дерева на бинарность после создания дерева путём вставки всех узлов |
| ContainsEmpty | Проверка функции Contains() на пустом дереве |
| ContainsSampleTest | Проверка функции Contains() в заранее заданном дереве |
| RemoveEmpty | Проверка функции Remove() на пустом дереве |
| RemoveNonPresent | Проверка функции Remove() на дереве, в котором нет искомого эл-та |
| RemoveAll | Проверка функции Remove() на дереве, в котором удаляются все узлы |
| RemoveSome | Проверка функции Remove() в заранее заданном дереве |
| DFTEmptyTest | Проверка работы итератора на пустом дереве |
| DFTTest | Проверка работы итератора на заранее заданном дереве |
| BFTEmptyTest | Проверка работы итератора на пустом дереве |
| BFTTest | Проверка работы итератора на заранее заданном дереве |

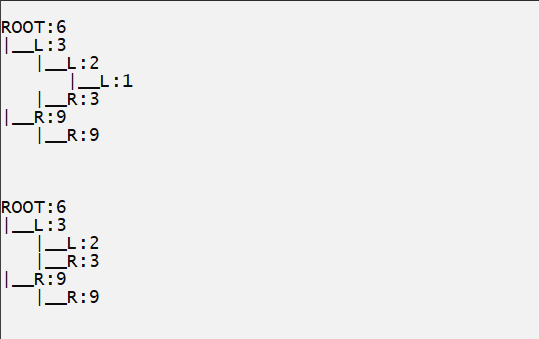


# Пример работы

## Функция main, представляющая пример использования написанной структуры данных:



## Результат выполнения:



# Текст программы:

|  |
| --- |
| #include <stdexcept>  #include <iostream>  #include <string>  #include "Queue.h";  #include "Stack.h";  using namespace std;  class TreeNode  {  TreeNode\* right, \* left;  int key;  public:  ~TreeNode() { delete right, left; }  TreeNode() : right(nullptr), left(nullptr), key(-1) {}  TreeNode(int value) : key(value), right(nullptr), left(nullptr) {}  TreeNode(int value, TreeNode\* right, TreeNode\* left, TreeNode\* parent) : key(value), right(right), left(left) {}  void SetRight(TreeNode\* rightElement) { right = rightElement;}  void SetLeft(TreeNode\* leftElement) { left = leftElement;}  void SetKey(int key) { this->key = key; }  TreeNode\* GetRight() { return right; }  TreeNode\* GetLeft() { return left; }  int GetKey() { return key; }  };  class Iterator  {  public:  virtual int Next() = 0;  virtual bool HasNext() = 0;  };  class TreeDFTIterator : public Iterator  {  private:  Stack<TreeNode\*> stack;  void pushToLeft(TreeNode\* node)  {  if (node == nullptr)  return;  stack.Push(node);  pushToLeft(node->GetLeft());  }  public:  TreeDFTIterator(TreeNode\* start)  {  pushToLeft(start);  }  int Next() override  {  TreeNode\* node = stack.Pop();  pushToLeft(node->GetRight());  return node->GetKey();  }  bool HasNext() override  {  return stack.GetSize() != 0;  }  };  class TreeBFTIterator : public Iterator  {  private:  Queue<TreeNode\*> queue;  public:  TreeBFTIterator(TreeNode\* start)  {  if (start == nullptr)  return;  queue.Enqueue(start);  }  int Next() override  {  TreeNode\* next = queue.Dequeue();  if (next->GetLeft() != nullptr)  queue.Enqueue(next->GetLeft());  if (next->GetRight() != nullptr)  queue.Enqueue(next->GetRight());  return next->GetKey();  }  bool HasNext() override  {  return queue.GetSize() != 0;  }  };  class Tree  {  private:  TreeNode\* root;  int size = 0;  TreeNode\* deleteRecursive(TreeNode\* root, int key)  {  if (root == nullptr)  return root;  //move down  if (key < root->GetKey())  root->SetLeft(deleteRecursive(root->GetLeft(), key));  else if (key > root->GetKey())  root->SetRight(deleteRecursive(root->GetRight(), key));  else  {  //we found target node  // node with only one child or no child  if (root->GetLeft() == nullptr)  {  return root->GetRight();  }  else if (root->GetRight() == nullptr)  {  return root->GetLeft();  }  //set smallest key from right subtree  root->SetKey(minValue(root->GetRight()));  // Delete smallest key in right subtree  TreeNode\* newNode = deleteRecursive(root->GetRight(), root->GetKey());  delete root->GetRight();  root->SetRight(newNode);  }  return root;  }  int minValue(TreeNode\* root)  {  int minv = root->GetKey();  while (root->GetLeft() != nullptr)  {  minv = root->GetLeft()->GetKey();  root = root->GetLeft();  }  return minv;  }  void print(TreeNode\* root, int depth = 0, int pos = 0)  {  //print itself  if (root == nullptr)  return;  cout << endl;  for (size\_t i = 0; i < depth; i++)  {  if (i != depth - 1)  cout << " ";  else  cout << "|\_\_";  }  cout << "";  if (pos == 0)  cout << "ROOT:";  if (pos == 1)  cout << "R:";  if (pos == -1)  cout << "L:";  cout << root->GetKey();  //print left  print(root->GetLeft(), depth + 1, -1);  //print right  print(root->GetRight(), depth + 1, 1);  }  public:  ~Tree() { delete root; }  Tree(TreeNode\* root) : root(root) { }  Tree() : root(nullptr) { }  Iterator\* CreateDFTIterator() { return new TreeDFTIterator(root); }  Iterator\* CreateBFTIterator() { return new TreeBFTIterator(root); }  int GetSize()  {  return size;  }  TreeNode\* GetRoot()  {  return root;  }  //template <class T>  void InsertKeys(std::initializer\_list<int> list)  {  for (auto elem : list)  {  Insert(elem);  }  }  void RemoveKeys(std::initializer\_list<int> list)  {  for (auto elem : list)  {  Remove(elem);  }  }  bool Contains(int key)  {  TreeNode\* current = root;  while (current != nullptr) {  if (key == current->GetKey())  return true;  else  {  if (key < current->GetKey())  current = current->GetLeft();  else  current = current->GetRight();  }  }  return false;  }  void Insert(int key)  {  size++;  TreeNode\* currentNode = root;  if (root == nullptr)  {  root = new TreeNode(key);  return;  }  else  while (true)  {  if (key < currentNode->GetKey())  {  if (currentNode->GetLeft() == nullptr)  {  currentNode->SetLeft(new TreeNode(key));  break;  }  else  currentNode = currentNode->GetLeft();  }  else  {  if (currentNode->GetRight() == nullptr)  {  currentNode->SetRight(new TreeNode(key));  break;  }  else  currentNode = currentNode->GetRight();  }  }  }  void Remove(int key)  {  if (!Contains(key))  throw exception("key is not present in collection");  root = deleteRecursive(root, key);  size--;  }  void Print() {print(root);}  }; |

# Вывод

В результате выполнения данной работы была реализована структура данных “двоичное дерево поиска”, была выполнена оценка временной сложности операций. Линейная сложность вставки, поиска и других функций, позволяет сделать вывод о удобности применения данной структуры для сортировки и хранения данных.