ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Департамент прикладной математики

ОТЧЕТ
К ЛАБОРАТОРНОЙ РАБОТЕ 1
по дисциплине «Алгоритмизация и программирование»

Работу выполнила
студентка группы БПМ 173 _____ М.В. Самоделкина
дата, подпись

Работу проверил _____ С.А. Булгаков
дата, подпись

Москва 2019

Содержание

Постановка задачи

Используя механизм шаблонных классов (*template*) написать классы для динамических структур данных и итераторов для их обхода в соответствии с вариантом 17:

1. Бинарное дерево - *ForwardIterator* с условием
2. Очередь - *ForwardIterator*

1 Основная часть

1.1 Общая идея решения задачи

Для решения задачи были созданы шаблонные классы *Queue* и *BinaryTree* с помощью агрегирования классов итераторов (*Iterator* и *ConstIterator*, и *TreeIterator* и *ConstTreeIterator* соответственно), классы итераторов в свою очередь содержат структуру узла *Node* или *TreeNode* соответственно.

1.2 Структура и принципы действия

Классы *Queue*, *Iterator* и *ConstIterator* содержат в себе структуру узла (*Node*), который состоит из элемента очереди и указателя на следующий элемент.

Классы *BinaryTree*, *TreeIterator* и *ConstTreeIterator* содержат в себе структуру узла (*TreeNode*), который состоит из элемента бинарного дерева и указателей на родительский, левый и правый элементы.

Класс *Queue* содержит в себе указатель на начало очереди и конец очереди (*Node**) и размер, определены конструктор умолчания, конструктор копирования, деструктор, а также такие стандартные методы, как *begin* (возвращает указатель на первый элемент),*end* (возвращает указатель на последний элемент), *len* (возвращает размер очереди), *push* (добавляет элемент в конец очереди), *pop* (удаляет элемент из начала очереди и возвращает его).

Класс *BinaryTree* содержит в себе указатель на начальную вершину бинарного дерева (*TreeNode**), указатель на функцию, которая задает условие добавления элемента в бинарное дерево, определены конструктор с параметром (принимает указатель на функцию), конструктор копирования, деструктор, а также такие стандартные методы, как *begin* (возвращает указатель на вершину дерева),*end* (возвращает нулевой указатель), *push* (добавляет элемент в бинарное дерево в соответствии с условием).

Каждый класс содержит классы *Iterator* и *ConstIterator*, или *TreeIterator* и *ConstTreeIterator*, которые содержат указатель на элемент структуры *Node* или *TreeNode*, конструктор с параметром, для которых определены операторы ++, *, ==, *!=*.

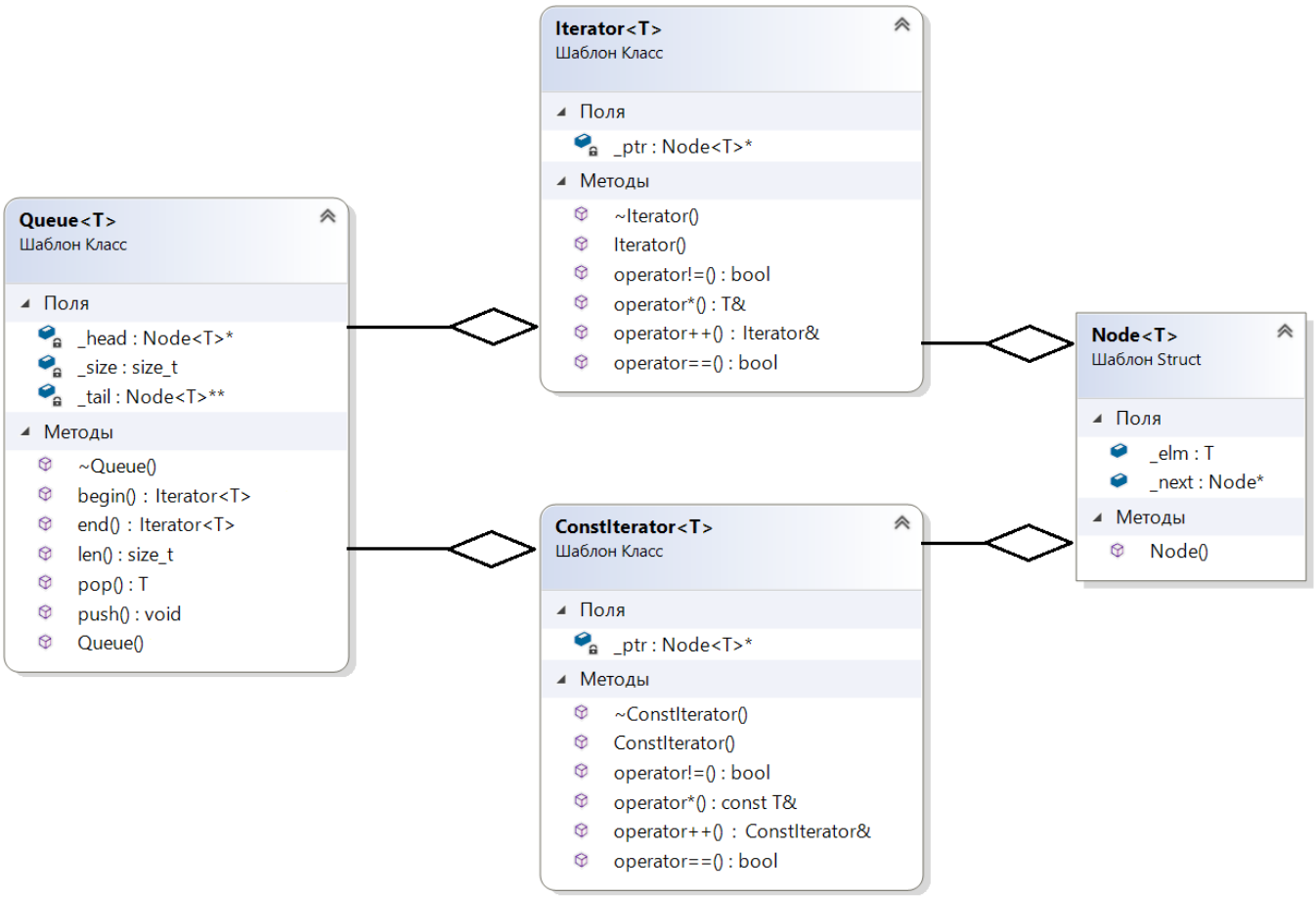

Рис. 1: Диграма классов *Queue*

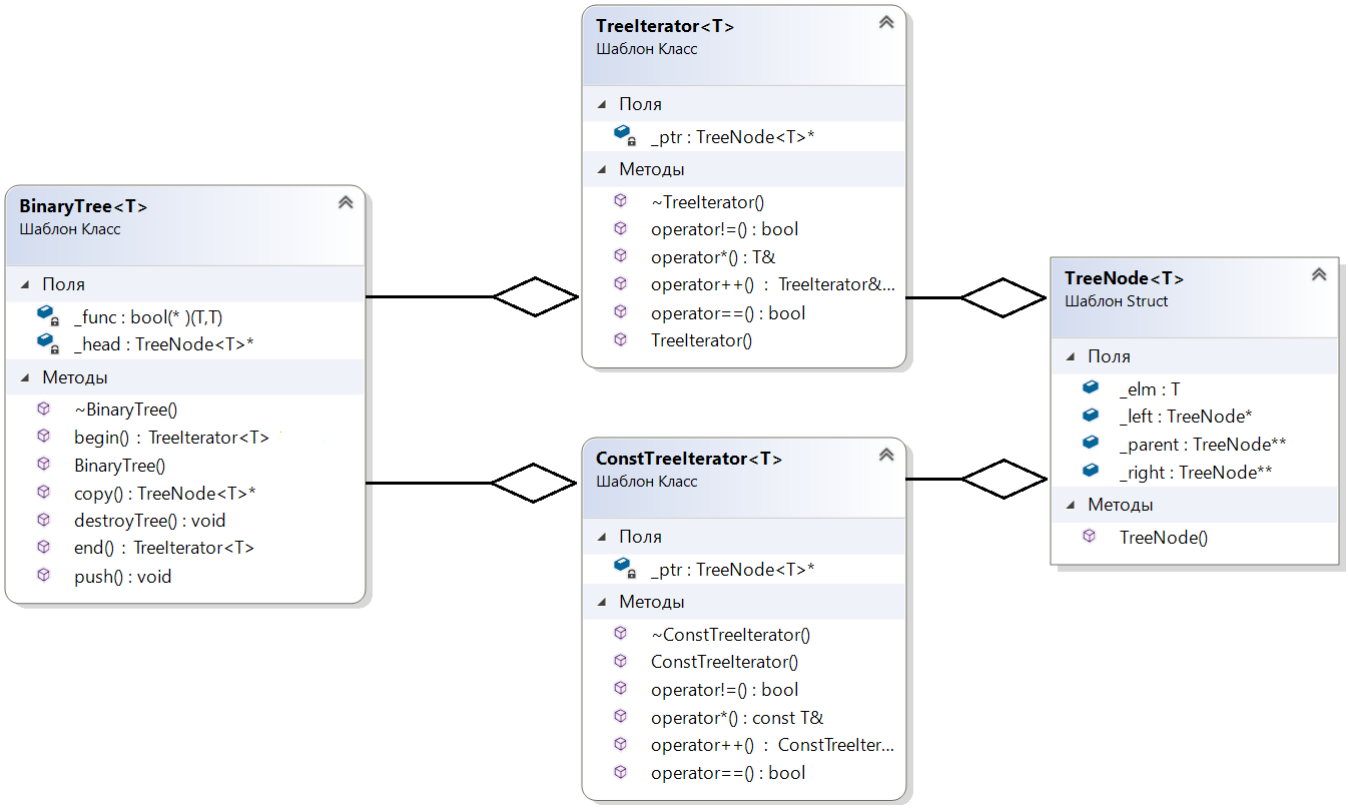

Рис. 2: Диграма классов *BinaryTree*

Также были реализованы шаблонные функции, перегружающие оператор вывода в поток для отображения коллекций *Queue* и *BinaryTree* на экране.

1.3 Процедура получения исполняемых программных модулей

Программный код был скомпилирован с среде *Visual Studio 2017*. Компиляция раздельная: исходный код программы разделён на несколько файлов. Для ускоренной компиляции программы используются предварительно откомпилированные заголовки *"pch.h"*. Помимо этого никаких дополнительных ключей не добавлялось, использовались ключи, которые добавляются по умолчанию.

1.4 Результаты тестирования

Тестирование программы представлено в файле *"ContainerIterator.cpp"* в функции *Main()*. Ожидаемый вывод функции:

```
1 2 -2 4 -4 5 -5 6 6 -6 123 234
1 2 -2 4 -4 5 -5 6 6 -6 123 234

1.7 0.333
0.333
1
```

Приложение А

полный код программы

A.1 - BinaryTree.h

```
1  #pragma once
2  #include ”pch.h”
3  #include <iostream>
4  #include ”Iterator.h”
5
6  template <typename T>
7  class BinaryTree
8  {
9  private:
10   TreeNode<T>* _head;
11   bool(*_func)(T, T);
12 public:
13   BinaryTree(bool(*func)(T, T)) {
14     _head = nullptr;
15     _func = func;
16   }
17   TreeNode<T>* copy(TreeNode<T>* node)
18   {
19     if (!node)
20       return nullptr;
21     TreeNode<T> * next = new TreeNode<T>(node->_elm);
22     next->_left = copy(node->_left);
23     if (next->_left)
24       next->_left->_parent = next;
25     next->_right = copy(node->_right);
26     if (next->_right)
27       next->_right->_parent = next;
28     return next;
29   }
30   BinaryTree(const BinaryTree&tree) {
```

```
31      _head = copy(tree._head);
32    }
33    void destroyTree(TreeNode<T> *t) {
34      if (t->_left)
35        destroyTree(t->_left);
36      if (t->_right)
37        destroyTree(t->_right);
38      delete t;
39    }
40    ~BinaryTree() {
41      destroyTree(_head);
42    }
43    void push(const T & elm) {
44      TreeNode<T>* node = new TreeNode<T>(elm);
45      if (_head) {
46        TreeNode<T>* next = _head;
47        while (next) {
48          if (_func(next->_elm, elm)) {
49            if (next->_left)
50              next = next->_left;
51            else {
52              next->_left = node;
53              node->_parent = next;
54              next = nullptr;
55            }
56          }
57          else {
58            if (next->_right)
59              next = next->_right;
60            else {
61              next->_right = node;
62              node->_parent = next;
63              next = nullptr;
64            }
```

```
65          }
66        }
67      }
68      else {
69        _head = node;
70      }
71    }
72    TreeIterator<T> begin() {
73      return TreeIterator<T>(_head);
74    }
75    TreeIterator<T> end() {
76      return TreeIterator<T>(nullptr);
77    }
78    ConstTreeIterator<T> begin() const {
79      return ConstTreeIterator<T>(_head);
80    }
81    ConstTreeIterator<T> end() const {
82      return ConstTreeIterator<T>(nullptr);
83    }
84  };
```

A.2 - Queue.h

```
1  #pragma once
2  #include <iostream>
3  #include "Iterator.h"
4
5  template <typename T>
6  class Queue
7  {
8  private:
9    Node<T> *_head, *_tail;
10   size_t _size;
11 public:
12   Queue() {
13     _head = _tail = nullptr;
```

```
14      _size = 0;
15    }
16    Queue(const Queue&queue) {
17      _head = _tail = nullptr;
18      _size = 0;
19      Node<T>* next(nullptr);
20      for (auto i = queue._tail;
21        i != nullptr; i = i->_next) {
22        push(T(i->_elm));
23      }
24    }
25    ~Queue() {
26      while (_head != nullptr) {
27        Node<T>* old_head = _head;
28        _head = _head->_next;
29        delete old_head;
30      }
31      _size = 0;
32    }
33    void push(const T & elm) {
34      if (_tail == nullptr) {
35        _tail = _head = new Node<T>(elm);
36        _size++;
37      }
38      else {
39        Node<T>* node = new Node<T>(elm);
40        _head->_next = node;
41        _head = node;
42        _size++;
43      }
44    }
45    T pop() {
46      if (_size) {
47        Node<T>* node = _tail->_next;
```

```
48        T elm = _tail->_elm;
49        delete _tail;
50        _tail = node;
51        _size--;
52        return elm;
53      }
54      else
55        throw std::exception("Queue is empty");
56    }
57    Iterator<T> begin() {
58      return Iterator<T>(_tail);
59    }
60    Iterator<T> end() {
61      return Iterator<T>(nullptr);
62    }
63    ConstIterator<T> begin() const {
64      return ConstIterator<T>(_tail);
65    }
66    ConstIterator<T> end() const {
67      return ConstIterator<T>(nullptr);
68    }
69    size_t len() {
70      return _size;
71    }
72  };
```

A.3 - Node.h

```
1  #pragma once
2  template <typename T>
3  struct Node {
4    T _elm;
5    Node* _next;
6    Node(const T & elm) : _elm(elm), _next(nullptr) {}
7  };
8
```

```
9  template <typename T>
10 struct TreeNode {
11   T _elm;
12   TreeNode *_left, *_right, *_parent;
13   TreeNode(const T & elm) : _elm(elm), _left(nullptr),
       _right(nullptr), _parent(nullptr) {}
14 };
```

A.4 - Iterator.h

```
1  #pragma once
2  #include "Node.h"
3
4  template <typename T>
5  class TreeIterator {
6  private:
7    TreeNode<T>* _ptr;
8  public:
9    TreeIterator(TreeNode<T>* ptr) : _ptr(ptr) {
10     //leftmost node is the begining
11     while (_ptr && _ptr->_left)
12     {
13       _ptr = _ptr->_left;
14     }
15   }
16   ~TreeIterator() {}
17   TreeIterator& operator++() {
18     //find the right leftmost node
19     if (_ptr->_right) {
20       _ptr = _ptr->_right;
21       while (_ptr->_left)
22         _ptr = _ptr->_left;
23     }
24     else {
25       TreeNode<T>* _tmp = _ptr->_parent;
26       //if we are at the right node; go by right
```

```
27        while (_tmp && _ptr == _tmp->_right) {
28          _ptr = _tmp;
29          _tmp = _tmp->_parent;
30        }
31        //go by left side
32        if (_ptr->_right != _tmp)
33          _ptr = _tmp;
34        if (!_tmp) {
35          _ptr = nullptr;
36        }
37      }
38      return *this;
39    }
40    TreeIterator operator++(int) {
41      TreeIterator i = *this;
42      operator++();
43      return i;
44    }
45    T& operator*() const {
46      return _ptr->_elm;
47    }
48    bool operator==(const TreeIterator& rhs) const {
49      return _ptr == rhs._ptr;
50    }
51    bool operator!=(const TreeIterator& rhs) const {
52      return !(_ptr == rhs._ptr);
53    }
54  };
55
56  template <typename T>
57  class ConstTreeIterator {
58  private:
59    TreeNode<T>* _ptr;
60  public:
```

```
61  ConstTreeIterator(TreeNode<T>* ptr) : _ptr(ptr) {
62    //leftmost node is the begining
63    while (_ptr && _ptr->_left)
64    {
65      _ptr = _ptr->_left;
66    }
67  }
68  ~ConstTreeIterator() {}
69  ConstTreeIterator& operator++() {
70    //find the right leftmost node
71    if (_ptr->_right) {
72      _ptr = _ptr->_right;
73      while (_ptr->_left)
74        _ptr = _ptr->_left;
75    }
76    else {
77      TreeNode<T>* _tmp = _ptr->_parent;
78      //if we are at the right node; go by right
79      while (_tmp && _ptr == _tmp->_right) {
80        _ptr = _tmp;
81        _tmp = _tmp->_parent;
82      }
83      //go by left side
84      if (_ptr->_right != _tmp)
85        _ptr = _tmp;
86      if (!_tmp) {
87        _ptr = nullptr;
88      }
89    }
90    return *this;
91  }
92  ConstTreeIterator operator++(int) {
93    TreeIterator i = *this;
94    operator++();
```

```
95      return i;
96    }
97    const T& operator*() const {
98      return _ptr->_elm;
99    }
100   bool operator==(const ConstTreeIterator& rhs) const {
101     return _ptr == rhs._ptr;
102   }
103   bool operator!=(const ConstTreeIterator& rhs) const {
104     return !(_ptr == rhs._ptr);
105   }
106 };
107
108 template <typename T>
109 class Iterator {
110 private:
111   Node<T>* _ptr;
112 public:
113   Iterator(Node<T>* ptr) : _ptr(ptr) {}
114   ~Iterator() {}
115   Iterator& operator++() {
116     _ptr = _ptr->_next;
117     return *this;
118   }
119   Iterator operator++(int) {
120     Iterator i = *this;
121     _ptr = _ptr->_next;
122     return i;
123   }
124   T& operator*() const {
125     return _ptr->_elm;
126   }
127   bool operator==(const Iterator& rhs) const {
128     return _ptr == rhs._ptr;
```

```
129   }
130   bool operator!=(const Iterator& rhs) const {
131     return !(_ptr == rhs._ptr);
132   }
133 };
134
135 template <typename T>
136 class ConstIterator {
137 private:
138   Node<T>* _ptr;
139 public:
140   ConstIterator(Node<T>* ptr) : _ptr(ptr) {}
141   ~ConstIterator() {}
142   ConstIterator& operator++() {
143     _ptr = _ptr->_next;
144     return *this;
145   }
146   ConstIterator operator++(int) {
147     ConstIterator i = *this;
148     _ptr = _ptr->_next;
149     return i;
150   }
151   const T& operator*() const {
152     return _ptr->_elm;
153   }
154   bool operator==(const ConstIterator& rhs) const {
155     return _ptr == rhs._ptr;
156   }
157   bool operator!=(const ConstIterator& rhs) const {
158     return !(_ptr == rhs._ptr);
159   }
160 };
```

A.5 - ContainerIterator.cpp

```
1 #include "pch.h"
```

```
2  #include "List.h"
3  #include "Queue.h"
4  #include "BinaryTree.h"
5
6
7  template<typename T >
8  std::ostream & operator<<(std::ostream & os,
9    const Queue <T> & list) {
10   for (auto i : list)
11   {
12     os << i;
13     os << "␣";
14   }
15   os << std::endl;
16   return os;
17 }
18
19 template<typename T >
20 std::ostream & operator<<(std::ostream & os,
21   const BinaryTree <T> & list) {
22   for (auto i : list)
23   {
24     os << i;
25     os << "␣";
26   }
27   os << std::endl;
28   return os;
29 }
30
31 template<typename T>
32 bool compare(T node, T newnode) {
33   return abs(node) > abs(newnode);
34 }
35
```

```
36 int main()
37 {
38   BinaryTree<int> tree(&compare);
39   tree.push(123);
40   tree.push(1);
41   tree.push(2);
42   tree.push(5);
43   tree.push(6);
44   tree.push(4);
45   tree.push(234);
46   tree.push(6);
47   tree.push(-2);
48   tree.push(-5);
49   tree.push(-6);
50   tree.push(-4);
51   std::cout << tree;
52   BinaryTree<int> tree1(tree);
53   tree.push(-100);
54   std::cout << tree1;
55   Queue<double> queue;
56   queue.push(1.7);
57   queue.push(0.333);
58   Queue<double> queue2(queue);
59   std::cout << queue;
60   queue2.pop();
61   std::cout << queue2;
62   std::cout << queue2.len();
63   return 0;
64 }
```