Министерство образования и науки РФ
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа программной инженерии

Отчёт по дисциплине «Алгоритмы и структуры данных»

Реализация структуры данных Tree (дерево)

Работу выполнил:
студент группы 3530904/00006
Смирнов Е. А.

Работу принял:
преподаватель Павлов Е. А.

Санкт-Петербург

2021

1. Постановка задачи

Реализовать шаблонный класс BinarySearchTree, описывающий интерфейс двоичных деревьев поиска

2. Реализация

Для описания узла дерева используется тип Node, в котором
key_ - значение ключа узла,
left_ - указатель на левое поддерево,
right_ - указатель на правое поддерево,
p_ - указатель на родителя.

Тип Node может использоваться только в классе BinarySearchTree

BinarySearchTree.hpp

```
#ifndef BINARY_SEARCH_TREE_H
#define BINARY_SEARCH_TREE_H

#include <iostream>
#include <stack> // Для итеративного обхода дерева
#include <vector>// Для сравнениЯ деревьев

template <class T>
class BinarySearchTree
{
public:
	BinarySearchTree();
	BinarySearchTree(const BinarySearchTree<T> & src) = delete;
	BinarySearchTree(BinarySearchTree<T>&& src);
	BinarySearchTree<T>& operator= (const BinarySearchTree<T>& src) = delete;
	BinarySearchTree<T>& operator= (BinarySearchTree<T>&& src);

	// true, если все узлы деревьев одинаковые.
	bool operator== (const BinarySearchTree<T>& src);
	bool operator> (const BinarySearchTree<T>& src) = delete;
	bool operator< (const BinarySearchTree<T>& src) = delete;

	virtual ~BinarySearchTree();

	// 1.1 Функция поиска по ключу в бинарном дереве поиска.
	bool iterativeSearch(const T& key) const;

  // 2.1 Вставка нового элемента в дерево:
  // true,если элемент добавлен
	// false, если элемент уже был
	bool insert(const T& key);

	// Удаление элемента из дерева,
	// не нарушающее порядка элементов
	// true,если элемент удален
	// false, если элемента не было.
	bool deleteKey(const T& key);

	// 4.1 Печать строкового изображения дерева в выходной поток out.
	void print(std::ostream& out) const;

	// 5.1 Определение количества узлов дерева.
	int getCount() const;
```

```
        // 6.1 Определение высоты дерева.
        int getHeight() const;

        // 7 Инфиксный обход дерева (итеративный).
        void iterativeInorderWalk () const;

        // 8.1 Инфиксный обход дерева (рекурсивный).
        void inorderWalk() const;

private:
        template <class T>
        struct Node {
                T key_;             // Значение ключа, содержащееся в узле.
                Node<T>* left_;     // Указатель на левое поддерево.
                Node<T>* right_;    // Указатель на правое поддерево.
                Node<T>* p_;        // Указатель на  родителя.

                // Конструктор узла.
                Node(T key, Node<T>* left = nullptr,
                        Node<T>* right = nullptr, Node<T>* p = nullptr):
                        key_(key),
                        left_(left),
                        right_(right),
                        p_(p)
                {}
        };

        Node<T>* root_; // Дерево реализовано в виде указателя на корневой узел.

        // Метод свап.
        void swap(BinarySearchTree<T>& left, BinarySearchTree<T>& right);

        // Рекурсивная функция для освобождения памяти -- используется в деструктуре
        void deleteSubtree(Node<T>* node);

// Рекурсивная функция для вывода изображения дерева в выходной поток в скобочной форме
        void printNode(std::ostream& out, Node<T>* root) const;

        // Рекурсивная функция определения количества узлов дерева.
        int getCountSubTree(const Node<T>* node) const;

        // Рекурсивная функция определения высоты дерева.
       int getHeightSubTree(Node<T>* node) const;

        // Рекурсивная функция сравнение узлов -- используется в операторе ==
        bool equalNode(const Node<T>* lhs, const Node<T>* rhs);

        // Рекурсивная функция для организации обхода узлов дерева. ?
        void inorderWalk(Node<T>* node) const;

        // Функция поиска адреса узла по ключу в бинарном дереве поиска. ?
        Node<T>* iterativeSearchNode(const T& key) const;

};
```

Реализация методов

```
template <class T>
BinarySearchTree<T>::BinarySearchTree():
	root_(nullptr)
{}

template <class T>
BinarySearchTree<T>::BinarySearchTree(BinarySearchTree<T>&& src)
{
	root_ = src.root_;
	src.root_ = nullptr;
}

template <class T>
BinarySearchTree<T>& BinarySearchTree<T>::operator= (BinarySearchTree <T>&& src)
{
	deleteSubtree(root_); // очистка l-value
	root_ = src.root_;    // пермещение
	src.root_ = nullptr;
}

// Память, занимается узлами дерева.
template <class T>
BinarySearchTree<T>::~BinarySearchTree()
{
	deleteSubtree(root_);
}

// Рекурсивная функция для освобождения памяти.
template <class T>
void BinarySearchTree<T>::deleteSubtree(Node<T>* node)
{
	if (node != nullptr)
	{
		if (node->left_ != nullptr) // Очищаем левое поддерево.
		{
			deleteSubtree(node->left_);
		}
		if (node->right_ != nullptr) // Очищаем правое поддерево.
		{
			deleteSubtree(node->right_);
		}
		// Далее, мы уверены, что все поддеревья удалены.

		if (node->p_ == nullptr) // Если node - корень
		{
			root_ = nullptr;
			delete node;
		}
		else // Если мы удаляем какое-то поддерево
		{
			if ((node->p_->left_ != nullptr)
&& (node->p_->left_->key_ == node->key_)) // Если node слева от своего родителя
			{
				node->p_->left_ = nullptr;
				delete node;
			}
			else if ((node->p_->right_ != nullptr)
&& (node->p_->right_->key_ == node->key_)) // Если node справа от своего родителя
			{
				node->p_->right_ = nullptr;
				delete node;
			}
```

```
                }
            }
        }

        // Сравнение деревьев
        template<class T>
        bool BinarySearchTree<T>::operator==(const BinarySearchTree<T>& src)
        {
            return BinarySearchTree<T>::equalNode(root_, src.root);
        }

        // Вставка нового элемента в дерево:
    // true,если элемент добавлен; false, если элемент уже был.
        template <class T>
        bool BinarySearchTree<T>::insert(const T& key)
        {
            Node<T>* x = root_; // Проверяемый узел.
            Node<T>* y = nullptr; // для родителя (изначально nullptr).

            if (root_ == nullptr) // Если дерево изначально пустое, то просто делаем
корень.
            {
                Node<T>* newRoot = new Node<T>(key, nullptr, nullptr, nullptr);
                root_ = newRoot;
                return true;
            }

            while (x != nullptr) // Пока не нашли нужную пустую ячейку...
            {
                if (x->key_ == key) // Не добавляем элемент с ключем, который есть в
дереве.
                {
                    return false;
                }
                // Родителем становится проверяемый элемент,
                // так как x пойдет глубже- он будет на 1 узел впереди.
                y = x;
                if (key < x->key_)
                {
                    x = x->left_;
                }
                else
                {
                    x = x->right_;
                }
            }
            // Смотрим, в какую сторону от родителя вставлять элемент.
            if (key < y->key_)
            {
                y->left_ = new Node<T>(key, nullptr, nullptr, y);
                return true;
            }
            else
            {
                y->right_ = new Node<T>(key, nullptr, nullptr, y);
                return true;
            }
        }

        // Рекурсивное сравнение узлов
        template <class T>
        bool BinarySearchTree<T>::equalNode(const Node<T>* lhs, const Node<T>* rhs)
        {
            if (lhs == nullptr && rhs == nullptr) {
                return true;
```

```
		}
		else if (rhs == nullptr || rhs == nullptr) {
			return false;
		}

		bool isLeftSame = equalNode(lhs->left, rhs->left);
		bool isRightSame = equalNode(rhs->right, rhs->right);

		return (lhs->key == rhs->key && isLeftSame && isRightSame);
	}

	// Метод свап.
	template <class T>
	void BinarySearchTree<T>::swap(BinarySearchTree<T>& left, BinarySearchTree<T>& right)
	{
		std::swap(left.root_, right.root_); // Просто меняем указатели на корень.
	}

	// Печать дерева
	template <class T>
	void BinarySearchTree<T>::print(std::ostream& out) const
	{
		printNode(out, root_);
		out << std::endl;
	}

		// Определение высоты дерева
	template <class T>
	int BinarySearchTree<T>::getHeight() const
	{
		// В случае, если в дереве только корень - высота будет = 0.
		// В случае, если дерево пустое - высота будет = -1.
		return getHeightSubTree(root_) - 1;
	}

	// Рекурсивная функция определения высоты дерева
	template <class T>
	int BinarySearchTree<T>::getHeightSubTree(Node<T>* node) const
	{
		// Если узел пустой - возвращаем 0.
		if (node == nullptr)
		{
			return 0;
		}

		// Если узел непустой - возвращаем 1 + высота самого высокого ребенка.
		// 1 складываются рекурсивно
		return std::max(1 + getHeightSubTree(node->left_), 1 + getHeightSubTree(node-
>right_));
	}

	// Определение количества узлов дерева
	template <class T>
	int BinarySearchTree<T>::getCount() const
	{
		return getCountSubTree(root_);
	}

	// Рекурсивная функция определения количества узлов дерева
	template <class T>
	int BinarySearchTree<T>::getCountSubTree(const Node<T>* node) const
	{

		if (node == nullptr) {
			return 0;
```

```
		}

		// Если узел не пустой - возвращаем 1 + количество узлов снизу.
		return (1 + getCountSubTree(node->left_) + getCountSubTree(node->right_));
	}

	// Функция поиска по ключу в бинарном дереве поиска.
	template <class T>
	bool BinarySearchTree<T>::iterativeSearch(const T& key)  const
	{
		return (iterativeSearchNode(key) != nullptr);
	}

	// Функция поиска адреса узла по ключу в бинарном дереве поиска
	template <class T>
	BinarySearchTree<T>::Node<T>* BinarySearchTree<T>::iterativeSearchNode(const T& key)
const
	{
		Node<T>* x = root_;
		while ((x != nullptr) && (x->key_ != key)) // Пока может происходить поиск :)
		{
			if (key < x->key_) // идем влево
			{
				x = x->left_;
			}
			else  // идем вправо
			{
				x = x->right_;
			}
		}
		return x;
	}

	// Инфиксный обход дерева (рекурсивный)
	template <class T>
	void BinarySearchTree<T>::inorderWalk() const
	{
		inorderWalk(root_);
		std::cout << "\n";
	}

	// Рекурсивная функция для организации обхода узлов дерева.
	template <class T>
	void BinarySearchTree<T>::inorderWalk(Node<T>* node) const
	{
		if (node != nullptr) // Если узел ненулевой..
		{
			inorderWalk(node->left_); // Обходим его левых детей
			// -- мы пишем сначала листья( детей),
			//затем только внутренний узлы ( когда "встреться 2 раз")
			std::cout << node->key_; // выводим его значение.
			std::cout << ", ";
			inorderWalk(node->right_); // Наконец, обходим его правых детей.
		}
	}

	// Инфиксный обход дерева (итеративный)
	template <class T>
	void BinarySearchTree<T>::iterativeInorderWalk() const
	{
		std::stack<Node<T>*> nodeStack; // Стек, хранящий указатели на узлы.
		Node<T>* current = root_; // Узел, в который смотрим.
		while ((nodeStack.size() != 0) || (current != nullptr)) // Пока есть узлы в
стеке или узел, ненулевой.
		{
```

```
                        if (current != nullptr)
                        {
                                nodeStack.push(current); // Заносим в очередь на просмотр.
                                current = current->left_; // Смотрим левого ребенка.
                        }
                        else // Если просматриваемый элемент нулевой (например, по левым детям
просматриваемого узла дошли до nullptr)...
                        {
                                current = nodeStack.top(); // Достаем элемент со стека на
просмотр и по нему идем вправо  // здесь можно использовать родителя вместо стека
                                nodeStack.pop();
                                std::cout << current->key_; // Смотрим его значение
                                std::cout << ", ";
                                current = current->right_; // Переходим в правое поддерево
                        }
                }
                std::cout << "\n";
        }

        // Удаление элемента из дерева, не нарушающее порядка элементов.
        template <class T>
        bool BinarySearchTree<T>::deleteKey(const T& key)
        {
                Node<T>* node = iterativeSearchNode(key); // Ищем узел, который собираемся
удалить.
                if (node == nullptr)
                {
                        return false;
                }

                // Удаляем лист.
                if ((node->left_ == nullptr) && (node->right_ == nullptr))
                {
                        if (node->p_ == nullptr) // Если этот лист - корень...
                        {
                                root_ = nullptr;
                                delete node;
                                return true;
                        }
                        else
                        {
                                if ((node->p_->left_ != nullptr) && (node->p_->left_->key_ ==
node->key_)) // Если этот лист слева от родителя...
                                {
                                        node->p_->left_= nullptr;
                                        node->key_ = 0;
                                        delete node;
                                        return true;
                                }
                                if ((node->p_->right_ != nullptr) && (node->p_->right_->key_ ==
node->key_)) // Если этот лист справа от родителя...
                                {
                                        node->p_->right_ = nullptr;
                                        node->key_ = 0;
                                        delete node;
                                        return true;
                                }
                        }
                }

                else if (((node->left_ != nullptr) && (node->right_ == nullptr)))
                { // Удаляем узел только с левым ребенком.
                        if (node->p_ == nullptr) // Если этот узел - корень...
                        {
                                node->left_->p_ = nullptr;
```

```
                        root_ = node->left_;
                        delete node;
                        return true;
                }
                else
                {
                        if ((node->p_->left_ != nullptr) && (node->p_->left_->key_ ==
node->key_)) // Если этот узел слева от родителя...
                        {
                                node->left_->p_ = node->p_;
                                node->p_->left_ = node->left_;
                                delete node;
                                return true;
                        }
                        if ((node->p_->right_ != nullptr) && (node->p_->right_->key_ ==
node->key_)) // Если этот узел справа от родителя...
                        {
                        node->left_->p_ = node->p_;
                        node->p_->right_ = node->left_;
                        delete node;
                        return true;
                        }
                }
        }
        else if (((node->left_ == nullptr) && (node->right_ != nullptr)))
        { // Удаляем узел только с правым ребенком.
                if (node->p_ == nullptr) // Если этот узел - корень...
                {
                        node->right_->p_ = nullptr;
                        root_ = node->right_;
                        delete node;
                        return true;
                }
                else
                {
                        if ((node->p_->left_ != nullptr) && (node->p_->left_->key_ ==
node->key_)) // Если этот узел слева от родителя...
                        {
                                node->right_->p_ = node->p_;
                                node->p_->left_ = node->right_;
                                delete node;
                                return true;
                        }
                        if ((node->p_->right_ != nullptr) && (node->p_->right_->key_ ==
node->key_)) // Если этот узел справа от родителя...
                        {
                                node->right_->p_ = node->p_;
                                node->p_->right_ = node->right_;
                                delete node;
                                return true;
                        }
                }
        }
        else if (((node->left_ != nullptr) && (node->right_ != nullptr)))
        { // Удаляем узел с обоими детьми. Будем искать минимальный(левый) элемент из
правого поддерева.
                Node<T>* x  = node->right_; // Смотрим правое поддерево.
                while (x->left_ != nullptr) // Пока есть левые дети - будем идти по
ним.
                {
                        x = x->left_;
                } // Теперь x - минимальный элемент из правого поддерева.
                if ((x->p_->left_ != nullptr) && (x->p_->left_->key_ == x->key_)) //
Если x - слева от своего родителя.
                {
```

```
                        x->p_->left_ = x->right_; // Добавляем к родителю возможную
правую ветку x (левой точно нет).
                        if (x->right_ != nullptr) // Устанавливаем родителя возможной
правой ветки x как родителя самого x.
                        {
                                x->right_->p_ = x->p_;
                        }
                }
                if ((x->p_->right_ != nullptr) && (x->p_->right_->key_ == x->key_)) //
Если x - справа от своего родителя.
                {
                        x->p_->right_ = x->right_; // Добавляем к родителю возможную
правую ветку x (левой точно нет).
                        if (x->right_ != nullptr) // Устанавливаем родителя возможной
правой ветки x как родителя самого x.
                        {
                                x->right_->p_ = x->p_;
                        }
                }
                node->key_ = x->key_;
                delete x;
                return true;
        }
        return false;
}
```

```
// Сравнение деревьев: true, если все узлы деревьев одинаковые.
template <class T>
bool BinarySearchTree<T>::operator== (const BinarySearchTree <T>& src)
{
```

```
		// Вектора, которые будут хранить последовательность элементов дерева после инфиксного обхода.
		std::vector<T> vec1;
		std::vector<T> vec2;
		// Указатели, которые ходят по дереву.
		Node<T>* node1 = root_;
		Node<T>* node2 = src.root_;
		// Стеки указателей для итеративных обходов деревьев.
		std::stack<Node<T>*> nodeStack1;
		std::stack<Node<T>*> nodeStack2;

		// Итеративно обходим оба дерева и заносим их в вектор.
		while ((nodeStack1.size() != 0) || (node1 != nullptr))
		{
			if (node1 != nullptr)
			{
				nodeStack1.push(node1); // Заносим в очередь на просмотр.
				node1 = node1->left_; // Смотрим левого ребенка.
			}
			else // Если просматриваемый элемент нулевой (например, по левым детям просматриваемого узла дошли до nullptr)...
			{
				node1 = nodeStack1.top(); // Достаем элемент со стека на просмотр и по нему идем вправо
				nodeStack1.pop();
				vec1.push_back(node1->key_);
				node1 = node1->right_; // Переходим в правое поддерево
			}
		}
		while ((nodeStack2.size() != 0) || (node2 != nullptr))
		{
			if (node2 != nullptr)
			{
				nodeStack2.push(node2);
				node2 = node2->left_;
			}
			else
			{
				node2 = nodeStack2.top();
				nodeStack2.pop();
				vec2.push_back(node2->key_);
				node2 = node2->right_;
			}
		}

		return vec1 == vec2;
	}

#endif
```

3. Тестовые функции и выходные значения

MAIN.CPP

```
#include "BinarySearchTree.hpp"
#include <iostream>

void testBaseMethods();
void testExtraMethods();

int main()
{
  setlocale(LC_ALL, "rus");
  testBaseMethods();
  testExtraMethods();
  return 0;
}

void testExtraMethods()
{
  std::cout << "\nTest 2 begin " << std::endl;

  BinarySearchTree<int> tree1;
  tree1.insert(0);
  tree1.insert(-10);
  tree1.insert(-11);
  tree1.insert(-9);
  tree1.insert(2);
  tree1.insert(1);
  tree1.insert(3);

  std::cout << "First tree: ";
  tree1.print(std::cout);
  std::cout << "Count: " << tree1.getCount() << std::endl;
  std::cout << "Height: " << tree1.getHeight() << std::endl;

  BinarySearchTree<int> tree2;
  tree2.insert(0);
  tree2.insert(-10);
  tree2.insert(-11);
  tree2.insert(-9);
  tree2.insert(2);
  tree2.insert(1);
  tree2.insert(3);
  std::cout << "Second tree: ";
  tree2.print(std::cout);
  std::cout << "Count: " << tree2.getCount() << std::endl;
  std::cout << "Height: " << tree2.getHeight() << std::endl;

  std::cout << ((tree1 == tree2) ? "Trees are equal\n" : "Trees are not equal\n");

  std::cout << "Delete list 3 :\n " << std::endl;
  tree1.deleteKey(3);
  tree1.print(std::cout);
  tree2.print(std::cout);
  std::cout << ((tree1 == tree2) ? "Trees are equal\n" : "Trees are not equal\n");

}
```

```
void testBaseMethods()
{
  std::cout << "Test begin " << std::endl;
  BinarySearchTree<int> tree;

  std::cout << "Add: " << 20 << std::endl;
  tree.insert(20);
  std::cout << "Add: " << 5 << std::endl;
  tree.insert(5);
  std::cout << "Add: " << 4 << std::endl;
  tree.insert(4);
  std::cout << "Add: " << 6 << std::endl;
  tree.insert(6);

  std::cout << "Add: " << 50 << std::endl;
  tree.insert(50);
  std::cout << "Add: " << 60 << std::endl;
  tree.insert(60);
  std::cout << "Add: " << 40 << std::endl;
  tree.insert(40);
  std::cout << "Add: " << 39 << std::endl;
  tree.insert(39);
  std::cout << "Add: " << 41 << std::endl;
  tree.insert(41);

  std::cout << "\nPrint tree:" << std::endl;
  tree.print(std::cout);

  std::cout << " Инфиксный обход дерева (итеративный):";
  tree.iterativeInorderWalk();

  std::cout << " Инфиксный обход дерева (рекурсивный)";
  tree.inorderWalk();

  std::cout << "\nCount: " << tree.getCount() << std::endl;
  std::cout << "Height: " << tree.getHeight() << std::endl;

  std::cout << "Delete list 39: " << std::endl;
  tree.deleteKey(39);
  tree.print(std::cout);

  std::cout << "Delete root: " << std::endl;
  tree.deleteKey(20);
  tree.print(std::cout);

  std::cout << "Delete inner node: " << 50 << std::endl;
  tree.deleteKey(50);
  tree.print(std::cout);

  std::cout << "Move trees: \n";
  BinarySearchTree<int> treeMove = std::move(tree);
  tree.print(std::cout);
  treeMove.print(std::cout);
}
```

Консоль отладки Microsoft Visual Studio

```
Test begin
Add: 20
Add: 5
Add: 4
Add: 6
Add: 50
Add: 60
Add: 40
Add: 39
Add: 41

Print tree:
(20(5(4()())(6()()))(50(40(39()())(41()()))(60()())))
 Инфиксный обход дерева (итеративный):4, 5, 6, 20, 39, 40, 41, 50, 60,
 Инфиксный обход дерева (рекурсивный)4, 5, 6, 20, 39, 40, 41, 50, 60,

Count: 9
Height: 3
Delete list 39:
(20(5(4()())(6()()))(50(40()(41()()))(60()())))
Delete root:
(40(5(4()())(6()()))(50(41()())(60()())))
Delete inner node: 50
(40(5(4()())(6()()))(60(41()())()))
Move trees:
()
(40(5(4()())(6()()))(60(41()())()))

Test 2 begin
First tree: (0(-10(-11()())(-9()()))(2(1()())(3()())))
Count: 7
Height: 2
Second tree: (0(-10(-11()())(-9()()))(2(1()())(3()())))
Count: 7
Height: 2
Trees are equal
Delete list 3 :

(0(-10(-11()())(-9()()))(2(1()())()))
(0(-10(-11()())(-9()()))(2(1()())(3()())))
Trees are not equal

C:\Users\evgen\Desktop\проекты\algorithms\4 tree\tree\Debug\tree.exe (процесс 8244) завершил работу с кодом 0.
```