Санкт-Петербургский политехнический университет Петра Великого

Институт машиностроения, материалов и транспорта

**КУРСОВАЯ РАБОТА**

по дисциплине Объектно-ориентрованное программирование

Выполнил

студент группы 3331506/80401 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ К. В. Зарубин

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ М. С. Ананьевский

«\_\_»\_\_\_\_\_\_\_\_\_2021 г

Санкт-Петербург

2021

Оглавление

[**1** **ВВЕДЕНИЕ** 3](#_Toc69416955)

[**2** **ИДЕЯ АЛГОРИТМА** 4](#_Toc69416956)

[**3** **ЭФФЕКТИВНОСТЬ** 5](#_Toc69416957)

[**4** **РЕЗУЛЬТАТЫ РАБОТЫ АЛГОРИТМА** 6](#_Toc69416958)

[**5** **СПИСОК ЛИТЕРАТУРЫ** 8](#_Toc69416959)

[**6** **ПРИЛОЖЕНИЕ** 9](#_Toc69416960)

1. **ВВЕДЕНИЕ**

В работе будет рассмотрен алгоритм сортировки с помощью бинарного дерева. Это универсальный алгоритм сортировки, заключающийся в построении двоичного дерева поиска с последующей сборкой результирующего массива путём обхода узлов построенного дерева в необходимом порядке.

Данный вид сортировки относится к классу сортировок вставками.

Данная сортировка является оптимальной в случаях, когда:

* данные получаются путём непосредственного чтения из потока (файла, сокета или консоли);
* данные уже построены в дерево;
* данные можно считать непосредственно в дерево.

1. **ИДЕЯ АЛГОРИТМА**

Из элементов массива формируется бинарное дерево поиска, обладающее следующими свойствами:

* оба поддерева – левое и правое – являются двоичными деревьями поиска;
* у всех узлов левого поддерева произвольного узла Х значения ключей данных меньше, нежели значение ключа данных самого узла Х;
* у всех узлов правого поддерева произвольного узла Х значения ключей данных не меньше, нежели значение ключа данных самого узла Х.

Первый элемент – корень дерева, остальные добавляются по следующему методу. Начиная с корня дерева, элемент сравнивается с узлами: если элемент меньше, чем узел, то спускаемся по левой ветке, иначе – по правой; спустившись до конца элемент сам становится узлом.

Построенное таким образом дерево можно легко обойти так, чтобы двигаться от узлов с меньшими значениями к узлам с большими. При этом получаем все элементы в возрастающем порядке. Такой обход называют центрированным (in-order traversal). При таком обходе корень дерева занимает место между результатами соответствующих обходов левого и правого поддерева. Вместе со свойствами бинарного дерева поиска центрированный обход даст отсортированный список узлов. Схема обхода представлена на рисунке 2.1.

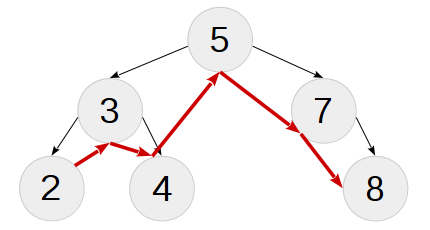


Рисунок 2.1 – Центрированный обход бинарного дерева

1. **ЭФФЕКТИВНОСТЬ**

Опишем скорость работы алгоритма в зависимости от количества входных данных.

Процедура добавления объекта в бинарное дерево имеет среднюю алгоритмическую сложность порядка , так как для каждого элемента требуется сравнений. Соответственно, для объектов сложность будет составлять , что относит сортировку бинарным деревом к группе быстрых сортировок.

Однако сложность добавления объекта в разбалансированное дерево может достигать , что может привести к общей сложности порядка .

Данные собраны в таблицу 1.

Таблица 1 – Алгоритмическая сложность сортировки бинарным деревом

|  |  |
| --- | --- |
| Худшая |  |
| Средняя |  |
| Лучшая |  |

1. **РЕЗУЛЬТАТЫ РАБОТЫ АЛГОРИТМА**

В качестве входных данных будем подавать массивы разной длины, замеряем количество итераций и время выполнения для каждого случая (результаты представлены в таблице 2).

Таблица 2 – Результаты измерения

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Элементы | Итерации | Время, нс | |  |  |  | | --- | --- | --- | | Элементы | Итерации | Время, нс | | 2100 | 2206049 | 1300 | | 2200 | 2421099 | 1920 | | 2300 | 2646149 | 1694 | | 2400 | 2881199 | 1581 | | 2500 | 3126249 | 3678 | | 2600 | 3381299 | 1886 | | 2700 | 3646349 | 3567 | | 2800 | 3921399 | 2737 | | 2900 | 4206449 | 2492 | | 3000 | 4501499 | 2380 | | 3100 | 4806549 | 2874 | | 3200 | 5121599 | 3276 | | 3300 | 5446649 | 2746 | | 3400 | 5781699 | 3144 | | 3500 | 6126749 | 3644 | | 3600 | 6481799 | 3844 | | 3700 | 6846849 | 3675 | | 3800 | 7221899 | 4086 | | 3900 | 7606949 | 3987 | | 4000 | 8001999 | 4979 | |
| 100 | 5049 | 1805 |
| 200 | 20099 | 982 |
| 300 | 45149 | 662 |
| 400 | 80199 | 839 |
| 500 | 125249 | 1292 |
| 600 | 180299 | 582 |
| 700 | 245349 | 593 |
| 800 | 320399 | 780 |
| 900 | 405449 | 778 |
| 1000 | 500499 | 909 |
| 1100 | 605549 | 1116 |
| 1200 | 720599 | 981 |
| 1300 | 845649 | 1467 |
| 1400 | 980699 | 1327 |
| 1500 | 1125749 | 1580 |
| 1600 | 1280799 | 1882 |
| 1700 | 1445849 | 2100 |
| 1800 | 1620899 | 1762 |
| 1900 | 1805949 | 1819 |
| 2000 | 2000999 | 1483 |

По таблице 2 построены графики зависимости времени выполнения от количества элементов (рисунок 4.1) и зависимости количества итераций от количества элементов (рисунок 4.2).

Рисунок 4.1 – График зависимости времени от количества элементов

Рисунок 4.2 – График зависимости количества итераций от количества элементов

1. **СПИСОК ЛИТЕРАТУРЫ**
2. Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Алгоритмы: построение и анализ = Introduction to Algorithms. — 2-е. — М.: Вильямс, 2005. — 1296 с.
3. Готтшлинг П. Современный C++ для программистов, инженеров и ученых. Серия «C++ In-Depth» = Discovering Modern C++: A Concise Introduction for Scientists and Engineers (C++ In-Depth). — М.: Вильямс, 2016. — 512 с.
4. Вирт Н. Алгоритмы и структуры данных — М.: Мир, 1989. — 360 с.
5. **ПРИЛОЖЕНИЕ**

Приложение 1

#include <algorithm>  
#include <vector>  
#include <iostream>  
#include <memory>  
#include <functional>  
  
using namespace std;  
  
// узел бинарного дерева  
struct BinaryTreeNode  
{  
 shared\_ptr<BinaryTreeNode> left, right; // левое и правое поддерево  
 int key; // ключ  
};  
  
// класс, представляющий бинарное дерево  
class BinaryTree  
{  
public:  
  
 void insert(int key);  
 typedef function<void(int key)> Visitor;  
 void visit(const Visitor& visitor);  
  
private:  
  
 void visit\_recursive(const shared\_ptr<BinaryTreeNode> cur\_node, const Visitor visitor);  
 shared\_ptr<BinaryTreeNode> node\_recursive; //корень для рекурсивной вставки  
 shared\_ptr<BinaryTreeNode> m\_root; // корень дерева  
 void insert\_recursive(const shared\_ptr<BinaryTreeNode> cur\_node, const shared\_ptr<BinaryTreeNode> node\_to\_insert);  
  
};  
  
// рекурсивная процедура вставки ключа  
// cur\_node - текущий узел дерева, с которым сравнивается вставляемый узел  
// node\_to\_insert - вставляемый узел  
void BinaryTree::insert\_recursive(const shared\_ptr<BinaryTreeNode> cur\_node, const shared\_ptr<BinaryTreeNode> node\_to\_insert)  
{  
 // сравнение  
 if(node\_to\_insert->key < cur\_node->key)  
 {  
 // вставка в левое поддерево  
 if(cur\_node->left == nullptr)  
 {  
 cur\_node->left = node\_to\_insert;  
 return;  
 }  
 node\_recursive = cur\_node->left;  
 }  
 else  
 {  
 // вставка в правое поддерево  
 if(cur\_node->right == nullptr)  
 {  
 cur\_node->right = node\_to\_insert;  
 return;  
 }  
 node\_recursive = cur\_node->right;  
 }  
 insert\_recursive(node\_recursive, node\_to\_insert);  
}  
  
void BinaryTree::insert(int key)  
{  
 shared\_ptr<BinaryTreeNode> node\_to\_insert(new BinaryTreeNode);  
 node\_to\_insert->key = key;  
  
 if(m\_root == nullptr)  
 {  
 m\_root = node\_to\_insert;  
 return;  
 }  
  
 insert\_recursive(m\_root, node\_to\_insert);  
}  
  
// рекурсивная процедура обхода дерева  
// cur\_node - посещаемый в данный момент узел  
void BinaryTree::visit\_recursive(const shared\_ptr<BinaryTreeNode> cur\_node, const Visitor visitor)  
{  
 // сначала посещаем левое поддерево  
 if(cur\_node->left != nullptr)  
 visit\_recursive(cur\_node->left, visitor);  
  
 // посещаем текущий элемент  
 visitor(cur\_node->key);  
  
 // посещаем правое поддерево  
 if(cur\_node->right != nullptr)  
 visit\_recursive(cur\_node->right, visitor);  
}  
  
void BinaryTree::visit(const Visitor& visitor)  
{  
 if(m\_root == nullptr)  
 return;  
 visit\_recursive(m\_root, visitor);  
}  
  
int main()  
{  
 BinaryTree tree;  
 // добавление элементов в дерево  
 vector<int> data\_to\_sort = {0, -1, 2, -3, 4, -5, 6, -7, 8, -9};  
 for(int value : data\_to\_sort)  
 {  
 tree.insert(value);  
 }  
 // обход дерева  
 tree.visit([](int visited\_key)  
 {  
 cout<<visited\_key<<" ";  
 });  
 cout<<endl;  
  
 return 0;  
}