|  |  |
| --- | --- |
|  | МИНОБРНАУКИ РОССИИ    федеральное государственное бюджетное образовательное учреждение  высшего образования  **«Балтийский государственный технический университет «ВОЕНМЕХ» им. Д.Ф. Устинова»**  **(БГТУ «ВОЕНМЕХ» им. Д.Ф. Устинова»)** |
| БГТУ.СМК-Ф-4.2-К5-01 |

Факультет И Информационные и управляющие системы

шифр наименование

Кафедра И5 Информационные системы и программные технологии

шифр наименование

Дисциплина Структуры и организация данных

ПРАКТИЧЕСКАЯ РАБОТА №3

на тему

Оценка эффективности алгоритмов

Выполнил студент группы И582

Дубровский В.И.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | Фамилия И.О. | | | | |
|  |  |  |  |  | **РУКОВОДИТЕЛЬ** | | |
| Палехова |  |  | О.А. |  |  |  |  |
| Фамилия И.О. | |  | |  | Подпись | |  |
| Оценка | |  | |  |  |  |  |
| «\_\_\_\_\_» |  |  |  |  |  | 20\_\_\_\_ г. | |

САНКТ-ПЕТЕРБУРГ

2019 г.

**8 вариант.**

**Цель работы** – познакомиться с различными алгоритмами сортировки и поиска, оценить их эффективность по памяти и количеству операций.

**Постановка задачи -** написать две программы согласно номеру индивидуального варианта.Перед написанием программ создать файл, содержащий N4 неупорядоченных

восьмизначных натуральных чисел (значение N4 указано в вариативной части в первом задании). В первой программе провести сравнение указанных алгоритмов сортировки массивов, содержащих N1, N2, N3 и N4 элементов. Каждую функцию сортировки вызывать трижды: для сортировки упорядоченного массива, массива, упорядоченного в обратном порядке и неупорядоченного массива. Сортируемая последовательность для всех методов должна быть одинаковой (считывать необходимое количество элементов из созданного файла). При задании начальных последовательностей учесть, что при сортировке по возрастанию или по убыванию все элементы последовательности должны быть различными, а при сортировке по невозрастанию или неубыванию должно быть совпадение части ключей. Оценить эффективность алгоритмов сортировки по заданному критерию и объему требуемой дополнительно памяти. Во второй программе реализовать две указанные структуры данных, заполнив их неповторяющимися значениями из созданного файла, выполнить поиск 100 ключей в указанных структурах данных. Оценить количество требуемой памяти для реализации каждой структуры и количество сравнений при поиске.

*Задача 1.*

Порядок: по неубыванию элементов. Методы: простых вставок, бинарных вставок, пирамидальная сортировка, метод Хоара. N1=15000, N2=40000, N3=80000, N4=150000. Критерий – количество сравнений.

Расчёты:

1. *Сортировка методом простых вставок.*

Сортировка методом простых ставок является одним из квадратичных методов сортировки. Его идея состоит в том: на каждом шаге алгоритма мы выбираем один из элементов входных данных и вставляем его на нужную позицию в уже отсортированной части массива до тех пор, пока весь набор входных данных не будет отсортирован.

Сложность в худшем случае, когда данные отсортированы в обратном порядке(O):

для сравнения ключей:

для вспомогательных сравнений:

для всех операций:

= ==> O(n2)

Сложность в среднем (Ɵ):

для сравнения ключей:

для вспомогательных сравнений:

Сложность в лучшем случае, когда данные отсортированы(Ω):

для сравнения ключей:

для вспомогательных сравнений:

для всех операций:

Пространственная сложность:

Текст функции:

void StraightInsertion(int \*array, int N)

{

for(int i=1; i < N ; i++)

{

int cur=array[i];

int j;

for(j=i; j && cur < array[j-1] ; j--)

{

array[j]=array[j-1];

}

array[j]=cur;

}

}

1. *Сортировка бинарными вставками:*

Сортировка бинарными вставками является улучшением сортировки простыми вставками, из-за того, что мы делим интервал поиска места вставки в упорядоченной последовательности пополам. В среднем при поиске места вставки вложенный цикл будет проходить четверть упорядоченной части. То есть при выигрыше в количестве сравнений (n\*log n), асимптотическая сложность O() останется.

Сложность в худшем случае, когда данные отсортированы в обратном порядке(O):

для сравнения ключей:

для вспомогательных сравнений:

для всех операций:

Сложность в среднем(Ɵ):

для сравнения ключей:

для вспомогательных сравнений:

Сложность в лучшем случае, когда данные уже отсортированы(Ω):

для сравнения ключей:

//вычитаем (n-1) т.к. именно настолько быстрее

левая граница меняет свою позицию

для вспомогательных сравнений:

для всех операций:

Пространственная сложность:

*Текст функции:*

void BinaryInsertion(int \*array, int N){

for(int i=1; (counter\_index++,i < N); i++)

{

int cur = array[i];

int left = 0, right = i;

while((counter\_index++,left < right))

{

int mid = (right+left)/2;

if(array[mid] <= cur)

left = mid + 1;

else

right=mid;

counter\_key++;

}

for (int j=i; (counter\_index++,j>right); array[j--]=array[j-1]);

array[right]=cur;

}

}

1. *Пирамидальная сортировка:*

В пирамидальной сортировке мы сначала преобразуем массив в пирамиду, а затем из пирамиды последовательно извлекаем максимум. Это неустойчивый алгоритм сортировки с временем работы O(n\*log(n)) и использующий O(1) дополнительной памяти.

Сложность в худшем случае, когда данные отсортированы(O):

для сравнения ключей:

//нотация по Седжвику

для вспомогательных сравнений:

для всех операций:

Сложность в среднем(Ɵ):

для всех операций:

Сложность в среднем(Ɵ):

для сравнения ключей:

для вспомогательных сравнений:

для всех операций:

Сложность в лучшем случае, когда данные отсортированы в обратном порядке(уже построенная пирамида)(Ω):

для сравнения ключей:

//нотация по Седжвику

для вспомогательных сравнений:

для всех операций:

Из представленных нотаций видно, что сортировка пирамидой будет выполняться за n\*log(n) независимо от природы входных данных. Значит не существует таких данных, которые могли бы изменить трудоёмкость сортировки

Пространственная сложность:

Текст функций:

void Sift(int \*array, int Left, int N)

{

int parent = Left;

int son = Left\*2+1;

int cur=array[Left];

if(son < N && array[son+1] > array[son]){

son++;

}

while(son < N && array[son] > cur)

{

array[parent] = array[son];

parent=son;

son=son\*2+1;

if(son < N && array[son+1] > array[son]){

son++;

}

}

array[parent]=cur;

}

void HeapSort(int \*array, int N)

{

int Begin = N/2;

int End = N-1;

while(Begin>0){

Sift(array,--Begin,End);

}

while(End>0)

{

std::swap(array[0], array[End]);

Sift(array,Begin,--End);

}

}

1. *Быстрая сортировка:*

Один из самых известных и широко используемых алгоритмов сортировки. Алгоритм состоит в выборе опорного элемента, разделении массива на 2 части относительно опорного (одна — все элементы, меньшие опорного элемента, вторая — большие), и в сортировке полученных частей рекурсивным вызовом себя от них.

Сложность в худшем случае, возникает на отсортированных данных в обратном порядке, когда опорным элементом выбирается крайний(O):

для всех операций:

для всех операций:

Сложность в лучшем случае(Ω):

Пространственная сложность:

Подсчёт раздельно вспомогательных сравнений и сравнений ключей слишком сложен. Так как в нашем тестировании не было выявлено выраженного случая, в котором бы количество сравнений суммарно бы приближалась к , то просто воспользуемся упрощённым тестированием результатов, приняв следующие уравнения:

– для упорядоченного массива

– в обратную сторону

– как средний показатель, т.е. для случайного набора данных (по Седжвику)

Текст функции:

void SORT::Sort\_test::QuickSort(int \*array, int N)

{

int piv = array[N/2];

int begin=0, end=N-1;

do{

while((counter\_key++,array[begin]<piv)) begin++;

while((counter\_key++,piv<array[end])) end--;

if((counter\_index++,begin<=end))

{

std::swap(array[begin], array[end]);

begin++; end--;

}

}while((counter\_index++,begin<end));

if((counter\_index++,end>0))

QuickSort(array, end+1);

if((counter\_index++,begin<N-1))

QuickSort(array+begin, N-begin);

}

*Результаты тестирования:*

1. **Сортировка прямыми вставками**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **15000** | | | | |
| **Исходные данные** | **Ожидаемый результат** | | **Результат работы программы** | |
| **Сравнения** | **Вспомогательные** | **Ключей** | **Вспомогательные** | **Ключей** |
| Не упорядочены | 56265000 | 56253750 | 56171882 | 56156879 |
| Упорядочены по не убыванию | 29999 | 14999 | 29999 | 14999 |
| Упорядочены по не возрастанию | 112500000 | 112492500 | 112490237 | 112517259 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **40000** | | | | |
| **Исходные данные** | **Ожидаемый результат** | | **Результат работы программы** | |
| **Сравнения** | **Вспомогательные** | **Ключей** | **Вспомогательные** | **Ключей** |
| Не упорядочены | 400040000 | 400010000 | 396695910 | 396655905 |
| Упорядочены по не убыванию | 79999 | 39999 | 79999 | 39999 |
| Упорядочены по не возрастанию | 800000000 | 799980000 | 800046515 | 799974464 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **80000** | | | | |
| **Исходные данные** | **Ожидаемый результат** | | **Результат работы программы** | |
| **Сравнения** | **Вспомогательные** | **Ключей** | **Вспомогательные** | **Ключей** |
| Не упорядочены | 1600079999 | 1600019999 | 1591902914 | 1591822909 |
| Упорядочены по не убыванию | 159999 | 79999 | 159999 | 79999 |
| Упорядочены по не возрастанию | 3200000000 | 3199960000 | 3200093495 | 3199949412 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **150000** | | | | |
| **Исходные данные** | **Ожидаемый результат** | | **Результат работы программы** | |
| **Сравнения** | **Вспомогательные** | **Ключей** | **Вспомогательные** | **Ключей** |
| Не упорядочены | 5625149999 | 5625037499 | 5617749545 | 5617599539 |
| Упорядочены по не убыванию | 299999 | 149999 | 299999 | 149999 |
| Упорядочены по не возрастанию | 11250000000 | 11249925000 | 11250174906 | 11249904685 |

1. **Сортировка бинарными вставками**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **15000** | | | | |
| **Исходные данные** | **Ожидаемый результат** | | **Результат работы программы** | |
| **Сравнения** | **Вспомогательные** | **Ключей** | **Вспомогательные** | **Ключей** |
| Не упорядочены | 56413557 | 185577 | 56374237 | 187356 |
| Упорядочены по не убыванию | 223076 | 178078 | 223629 | 178631 |
| Упорядочены по не возрастанию | 112604039 | 193077 | 112725455 | 193197 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **40000** | | | | |
| **Исходные данные** | **Ожидаемый результат** | | **Результат работы программы** | |
| **Сравнения** | **Вспомогательные** | **Ключей** | **Вспомогательные** | **Ключей** |
| Не упорядочены | 400478620 | 551494 | 397291910 | 556001 |
| Упорядочены по не убыванию | 651493 | 571494 | 654479 | 534481 |
| Упорядочены по не возрастанию | 800659700 | 571494 | 800659700 | 573186 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **80000** | | | | |
| **Исходные данные** | **Ожидаемый результат** | | **Результат работы программы** | |
| **Сравнения** | **Вспомогательные** | **Ключей** | **Вспомогательные** | **Ключей** |
| Не упорядочены | 1601017251 | 1183002 | 1593174906 | 1191993 |
| Упорядочены по не убыванию | 1383001 | 1143003 | 1388944 | 1148946 |
| Упорядочены по не возрастанию | 3200651501 | 1223002 | 3201399753 | 1226259 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **150000** | | | | |
| **Исходные данные** | **Ожидаемый результат** | | **Результат работы программы** | |
| **Сравнения** | **Вспомогательные** | **Ключей** | **Вспомогательные** | **Ключей** |
| Не упорядочены | 5627009380 | 2354175 | 5620270366 | 2370822 |
| Упорядочены по не убыванию | 2729173 | 2279175 | 2737873 | 2287875 |
| Упорядочены по не возрастанию | 11251289587 | 2429174 | 11252757981 | 2433076 |

1. **Сортировка пирамидой**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **15000** | | | | |
| **Исходные данные** | **Ожидаемый результат** | | **Результат работы программы** | |
| **Сравнения** | **Вспомогательные** | **Ключей** | **Вспомогательные** | **Ключей** |
| Не упорядочены | 453665 | 411925 | 427437 | 370353 |
| Упорядочены по не убыванию | 468650 | 416152 | 447419 | 382873 |
| Упорядочены по не возрастанию | 438680 | 407698 | 405571 | 357668 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **40000** | | | | |
| **Исходные данные** | **Ожидаемый результат** | | **Результат работы программы** | |
| **Сравнения** | **Вспомогательные** | **Ключей** | **Вспомогательные** | **Ключей** |
| Не упорядочены | 1323000 | 1225863 | 1253433 | 1101333 |
| Упорядочены по не убыванию | 1362984 | 1222986 | 1311593 | 1138226 |
| Упорядочены по не возрастанию | 1283016 | 1228740 | 1194867 | 1067529 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **80000** | | | | |
| **Исходные данные** | **Ожидаемый результат** | | **Результат работы программы** | |
| **Сравнения** | **Вспомогательные** | **Ключей** | **Вспомогательные** | **Ключей** |
| Не упорядочены | 2806016 | 2631755 | 2666949 | 2362679 |
| Упорядочены по не убыванию | 2885999 | 2606001 | 2785475 | 2437913 |
| Упорядочены по не возрастанию | 2726033 | 2657509 | 2547613 | 2292631 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **150000** | | | | |
| **Исходные данные** | **Ожидаемый результат** | | **Результат работы программы** | |
| **Сравнения** | **Вспомогательные** | **Ключей** | **Вспомогательные** | **Ключей** |
| Не упорядочены | 5533362 | 5240643 | 5271349 | 4700654 |
| Упорядочены по не убыванию | 5683344 | 5158346 | 5487315 | 4839326 |
| Упорядочены по не возрастанию | 5383380 | 5322940 | 5046071 | 4570688 |

1. **Быстрая сортировка**

|  |  |  |
| --- | --- | --- |
| **15000** | | |
| **Исходные данные** | **Ожидаемый результат** | **Результирующие данные** |
| Упорядочены по возрастанию | 416180 | 427937 |
| Упорядочены по убыванию | 208090 | 244698 |
| Не упорядочены | 208090 | 259652 |

|  |  |  |
| --- | --- | --- |
| **40000** | | |
| **Исходные данные** | **Ожидаемый результат** | **Результирующие данные** |
| Упорядочены по возрастанию | 1223016 | 1236869 |
| Упорядочены по убыванию | 611508 | 717369 |
| Не упорядочены | 611508 | 757489 |

|  |  |  |
| --- | --- | --- |
| **80000** | | |
| **Исходные данные** | **Ожидаемый результат** | **Результирующие данные** |
| Упорядочены по возрастанию | 2606033 | 2775334 |
| Упорядочены по убыванию | 1303016 | 1514683 |
| Не упорядочены | 1303016 | 1594420 |

|  |  |  |
| --- | --- | --- |
| **150000** | | |
| **Исходные данные** | **Ожидаемый результат** | **Результирующие данные** |
| Упорядочены по возрастанию | 5158380 | 5287664 |
| Упорядочены по убыванию | 2579190 | 2964338 |
| Не упорядочены | 2579190 | 3114237 |

Вывод:

Теоретические расчеты выполнены верно, их значения либо полностью, либо примерно совпадают с результатом работы программы. (Погрешность возникает, т.к. формулы в некоторых случаях описывают приближенное кол-во присваиваний).

* Анализируя результат, заметим, что у сортировки методом простых вставок суммарно самое большее количество сравнений из-за квадратичной трудоёмкости, но этот метод эффективнее остальных по критерию дополнительной памяти.
* Его улучшенная версия с бинарным поиском показала себя чуть лучше, количество сравнений ключей линейно-логарифмическое, но вспомогательные сравнений всё так же имеют квадратичную сложность.
* Пирамидальная сортировка предлагает линейно-логарифмическую трудоёмкость за константные затраты памяти и показывает себя стабильно, вне зависимости от входных данных.
* Самой быстрой сортировкой оказалась QuickSort, но в тоже время она имеет линейную пространственную сложность и редкие вырожденные случаи, в которых сортировка деградирует до квадратичной трудоёмкости.

*Задача 2.*

Несбалансированное дерево бинарного поиска, splay-дерево. Учесть, что вероятности

поиска разных ключей должны различаться.

|  |
| --- |
| **Несбалансированное дерево бинарного поиска** |
| Сложность поиска в среднем:    В лучшем случае искомый элемент находится в корне:  Ω(1)  В худшем случае, когда дерево вырождается в список :  Пространственная сложность:  g(n) = 3\*4\*n = 12n = 1800000 байт. |
| **Splay-дерево** |
| Сложность поиска в среднем:    В лучшем случае искомый элемент находится в корне:  Ω(1)  В худшем случае, когда дерево вырождается в список:  Пространственная сложность:  g(n) = 3\*4\*n = 12n = 1800000 байт. |

Результаты тестирования:

….

38. Key:56432424 - Not found

Count of comparing keys in SplayTree: 25

Count of comparing keys in non Balanced Binary Tree: 21

39. Key:47594519(REP^6) - Found

Count of comparing keys in SplayTree: 5

Count of comparing keys in non Balanced Binary Tree: 19

40. Key:65513297(REP^4) - Found

Count of comparing keys in SplayTree: 5

Count of comparing keys in non Balanced Binary Tree: 22

41. Key:42787934(REP^24) - Not found

Count of comparing keys in SplayTree: 16

Count of comparing keys in non Balanced Binary Tree: 26

42. Key:31648967 - Found

Count of comparing keys in SplayTree: 29

Count of comparing keys in non Balanced Binary Tree: 20

43. Key:42122844(REP^13) - Found

Count of comparing keys in SplayTree: 5

Count of comparing keys in non Balanced Binary Tree: 25

44. Key:47594519(REP^39) - Found

Count of comparing keys in SplayTree: 3

Count of comparing keys int non Balanced Binary Tree: 19

….

54. Key:84135156 - Found

Count of comparing keys in SplayTree: 24

Count of comparing keys in non Balanced Binary Tree: 27

55. Key:78149836 - Found

Count of comparing keys in SplayTree: 17

Count of comparing keys in non Balanced Binary Tree: 28

56. Key:78149836(REP^55) - Found

Count of comparing keys in SplayTree: 1

Count of comparing keys in non Balanced Binary Tree: 28

57. Key:99999563 - Not found

Count of comparing keys in SplayTree: 17

Count of comparing keys in non Balanced Binary Tree: 10

58. Key:91919406(REP^11) - Found

Count of comparing keys in SplayTree: 6

Count of comparing keys in non Balanced Binary Tree: 26

Вывод:

Splay-дерево показывает более быстрый поиск, если поиск некоторых ключей повторяется. При уникальном наборе искомых данных, т.е. все данные для поиска различны, оба дерева осуществляют поиск за примерно равное количество сравнений.

Текст программы:

*Реализация несбалансированного дерева бинарного поиска:*

*(binary\_search\_tree.h)*

#pragma once

#include <iostream>

struct node{

int key;

struct node \*left;

struct node \*right;

explicit node(int key) : key(key)

{

left=right=nullptr;

}

node() = default;

};

class BinarySearchTree

{

private:

node \*root;

void clean\_help(node \*tree);

void add\_help(node \*tree,int key);

node\* search\_help(node \*tree, int key);

void min\_del(node \*min, node \*cur);

void inorder\_help(node \*tree);

void print\_vis(node \*treem, int count);

public:

static size\_t counter\_cmp;

static size\_t counter\_help;

BinarySearchTree() : root(nullptr) {};

~BinarySearchTree()

{

clean\_help(root);

};

void add\_node(int key)

{

if(root == nullptr)

root = new node(key);

else add\_help(root, key);

};

node\* search(int key)

{

return search\_help(root,key);

};

bool del\_node(int key);

void PrintInorder()

{

inorder\_help(root);

std::cout << std::endl;

}

void Print()

{

int counter=0;

print\_vis(root,counter);

}

};

*(binary\_search\_tree.cpp)*

#include "binary\_search\_tree.h"

size\_t BinarySearchTree::counter\_cmp;

void BinarySearchTree::clean\_help(node \*tree)

{

if(tree==nullptr) return;

clean\_help(tree->left);

clean\_help(tree->right);

delete tree;

}

void BinarySearchTree::add\_help(node \*tree, int key)

{

if(tree->key > key)

{

if(tree->left)

add\_help(tree->left, key);

else tree->left = new node(key);

}

else if(tree->key < key)

{

if(tree->right)

add\_help(tree->right, key);

else tree->right = new node(key);

}

else {

//do smth

return;

}

}

node \*BinarySearchTree::search\_help(node \*tree, int key)

{

counter\_cmp++;

if(tree == nullptr || tree->key==key)

return tree;

else if(tree->key>key)

return search\_help(tree->left,key);

else

return search\_help(tree->right,key);

}

void BinarySearchTree::min\_del(node \*min, node \*cur)

{

if(min->left)

min\_del(min->left, cur);

else

{

cur->key=min->key; //swap

cur=min;

min = min->right;

}

}

bool BinarySearchTree::del\_node(int key)

{

node\* del=search\_help(root,key);

if(del)

{

node\* temp=del;

if(del->left==nullptr)

del=del->right;

else if(del->right==nullptr)

del=del->left;

else

min\_del(del->right, temp);

delete temp;

return true;

}

return false;

}

void BinarySearchTree::inorder\_help(node \*tree)

{

if(tree == nullptr) return;

inorder\_help(tree->left);

std::cout << tree->key << '\t';

inorder\_help(tree->right);

}

void BinarySearchTree::print\_vis(node \*tree, int count)

{

if(tree==nullptr) return;

print\_vis(tree->right, count+1);

for(size\_t i=0; i<=count; i++) std::cout << " ";

std::cout << tree->key << std::endl;

print\_vis(tree->left, count+1);

}

*Реализация splay-дерева:*

*(splay\_tree.h)*

#pragma once

#include <iostream>

namespace SPLAY{

struct node{

int key;

//int size;

struct node \*left;

struct node \*right;

struct node \*parent;

node () = default;

explicit node(node \*parent,int key) : key(key)

{

left=right=nullptr;

this->parent=parent;

//size=1;

};

};

class SplayTree

{

private:

node \*root;

////bin tree

void clean\_help(node\* tree);

node\* add\_help(node \*tree, int key);

void inorder\_help(node \*tree);

void print\_vis(node \*tree, int count);

node\* search\_help(node\* cur, node\* prev, int key);

node\* del\_help(node \*cur);

void chainGrandParent(node \*gp, node \*p, node \*ch);

node\* rightRotate(node \*p);

node\* leftRotate(node \*p);

/////splay

bool isZigZag(node \*cur);

void zigZag(node\* cur);

void zigZig(node\* cur);

void zig(node\* cur);

void splay(node\* cur);

public:

static size\_t counter\_cmp;

SplayTree() : root(nullptr) {};

~SplayTree()

{

clean\_help(root);

}

void PrintInorder()

{

inorder\_help(root);

std::cout << std::endl;

}

void add\_node(int key)

{

if(root==nullptr){

node\* parent=nullptr;

root=new node(parent,key);

}

splay(add\_help(root,key));

}

node \*search(int key)

{

node\* prev = nullptr;

node\* res= search\_help(root, prev, key);

if(res){

splay(res);

return root;

}

if(prev)

splay(prev);

return nullptr;

}

bool del\_node(int key)

{

node\* prev=nullptr;

node\* res = search\_help(root, prev, key);

if(res)

splay(del\_help(res));

else if(prev)

splay(prev);

return res ? true : false;

}

void Print()

{

int counter=0;

print\_vis(root, counter);

}

};

}

*(splay\_tree.cpp)*

#include "splay\_tree.h"

size\_t SPLAY::SplayTree::counter\_cmp;

void SPLAY::SplayTree::clean\_help(node \*tree)

{

if(tree==nullptr) return;

clean\_help(tree->left);

clean\_help(tree->right);

delete tree;

}

SPLAY::node\* SPLAY::SplayTree::add\_help(node \*tree, int key)

{

if(tree->key == key)

return tree;

node\* tmp;

if(tree->key > key){

if(tree->left)

tmp = add\_help(tree->left, key);

else

tmp = tree->left = new node(tree,key);

}

else if(tree-> key < key){

if(tree->right)

tmp = add\_help(tree->right, key);

else

tmp = tree->right=new node(tree, key);

}

else{

tmp=tree;

}

return tmp;

}

void SPLAY::SplayTree::inorder\_help(node \*tree)

{

if(tree == nullptr) return;

inorder\_help(tree->left);

std::cout << tree->key << '\t';

inorder\_help(tree->right);

}

void SPLAY::SplayTree::print\_vis(node \*tree, int count)

{

if(tree==nullptr) return;

print\_vis(tree->right, count+1);

for(size\_t i=0; i<=count; i++) std::cout << " ";

std::cout << tree->key << std::endl;

print\_vis(tree->left, count+1);

}

SPLAY::node\* SPLAY::SplayTree::search\_help(node\* cur, node\* prev, int key)

{

while(cur)

{

counter\_cmp++;

if(cur->key == key)

return cur;

else{

prev=cur;

cur=(cur->key > key ? cur->left : cur->right);

}

}

return nullptr;

}

SPLAY::node\* SPLAY::SplayTree::del\_help(node\* cur)

{

node \* removedNode = nullptr, \*p=nullptr;;

if(cur->left && cur->right)

{

node \*prev=nullptr;

node\* higherClosest = search\_help(cur,prev, cur->key+1);

if(higherClosest==nullptr){

higherClosest=prev;

p=prev->parent;

}

else{

p=prev;

}

cur->key = higherClosest->key; // swap

if(higherClosest == p->left)

p->left = higherClosest->right; //higherclosest always is have't left son, here just up right son;

else

p->right = higherClosest->right;

if(higherClosest->right)

higherClosest->right->parent = p;

removedNode = higherClosest;

} else{

p = cur->parent;

node \*ch = cur->left ? cur->left : cur->right;

if(p==nullptr)

root = ch;

else {

if(p->left == cur)

p->left = ch;

else

p->right =ch;

}

if(ch)

ch->parent = p;

removedNode = cur;

}

delete removedNode;

return p;

}

void SPLAY::SplayTree::chainGrandParent(node\* gp, node\* p, node\* ch)

{

if(gp->left == p)

gp->left = ch;

else

gp->right = ch;

}

SPLAY::node\* SPLAY::SplayTree::rightRotate(node \*p)

{

node\* ch = p->left;

p->left=ch->right;

if(ch->right)

ch->right->parent = p;

ch->parent = p->parent;

if(p->parent)

chainGrandParent(p->parent, p, ch);

else

root = ch;

ch->right=p; p->parent = ch;

return ch;

}

SPLAY::node\* SPLAY::SplayTree::leftRotate(node \*p)

{

node \*ch = p->right;

p->right = ch->left;

if(ch->left)

ch->left->parent=p;

ch->parent= p->parent;

if(p->parent)

chainGrandParent(p->parent, p, ch);

else

root = ch;

ch->left = p; p->parent = ch;

return ch;

}

bool SPLAY::SplayTree::isZigZag(node \*cur)

{

node\* p = cur->parent, \*gp=p->parent;

return (p->right == cur && gp->left == p) ||

(p->left == cur && gp->right == p);

}

void SPLAY::SplayTree::zigZag(node \*cur)

{

node\* p= cur->parent;

if(p->right == cur && p->parent->left == p)

rightRotate(leftRotate(p)->parent);

else

leftRotate(rightRotate(p)->parent);

}

void SPLAY::SplayTree::zigZig(node \*cur)

{

node\* p = cur->parent;

if(p->left == cur && p->parent->left == p)

rightRotate(rightRotate(p->parent));

else

leftRotate(leftRotate(p->parent));

}

void SPLAY::SplayTree::zig(node \*cur)

{

node\* p = cur->parent;

if(p->left == cur)

rightRotate(p);

else

leftRotate(p);

}

void SPLAY::SplayTree::splay(node\* cur)

{

if(cur == nullptr)

return;

while(cur->parent)

{

if(cur->parent->parent)

if(isZigZag(cur))

zigZag(cur);

else

zigZig(cur);

else

zig(cur);

}

}

*main.cpp*

#include <iostream>

#include <fstream>

#include "binary\_search\_tree.h"

#include "splay\_tree.h"

#include <vector>

#include <random>

#include <set>

#include <algorithm>

#include <map>

int main() {

SPLAY::SplayTree::counter\_cmp = 0;

BinarySearchTree::counter\_cmp = 0;

auto splay\_tree = new SPLAY::SplayTree();

auto bin\_tree = new BinarySearchTree();

std::ifstream file("unsorted.dat", std::ios::binary);

if(file.bad()){

std::cerr << "Couldn't open file!" << std::endl;

std::cin.get();

return 1;

}

std::vector<int> keys\_search;

std::cout << "Init data of keys" << std::endl;

{

std::uniform\_int\_distribution uid\_chance(1,4);

std::uniform\_int\_distribution uid\_repeat(1,7);

int min=100000000;

int max=0;

std::set<int> existing\_keys;

int exist,smaller,bigger,inside\_range;

std::mt19937 mt\_rand(time(NULL));

{

{

std::uniform\_int\_distribution uid\_exist(30,50);

exist = uid\_exist(mt\_rand);

std::uniform\_int\_distribution uid\_smaller(exist+10, exist+20);

smaller = uid\_smaller(mt\_rand);

std::uniform\_int\_distribution uid\_bigger(smaller+10, smaller+20);

bigger = uid\_bigger(mt\_rand);

inside\_range=99;

}

/\*std::cout << inside\_range-bigger << std::endl << bigger-smaller << std::endl

<< smaller-exist << std::endl << exist << std::endl;

std:: cout << "OAP\n" << exist << '\t' << smaller << '\t' << bigger << std::endl;\*/

}

for(size\_t i=0; !file.eof(); i++)

{

int key;

file.read((char\*) &key, sizeof(int));

if(min>key)

min=key;

if(max<key)

max=key;

existing\_keys.insert(key);

splay\_tree->add\_node(key);

bin\_tree->add\_node(key);

}

//std::cout << min << '\t' << max << std::endl;

{

int rep;

std::uniform\_int\_distribution uid\_inside(min,max);

while(keys\_search.size()<exist)

{

int key = uid\_inside(mt\_rand);

if(existing\_keys.find(key)!=existing\_keys.end()){

if(uid\_chance(mt\_rand)==1){

rep = uid\_repeat(mt\_rand);

for(int i=0; i < rep && keys\_search.size()<exist; i++)

keys\_search.push\_back(key);

}

else keys\_search.push\_back(key);

}

}

std::uniform\_int\_distribution uid\_smaller(10000000, min);

while(keys\_search.size()<smaller)

{

if(uid\_chance(mt\_rand)==1){

rep=uid\_repeat(mt\_rand);

for(int i=0; i<rep && keys\_search.size()<smaller; i++)

keys\_search.push\_back(uid\_smaller(mt\_rand));

}

else keys\_search.push\_back(uid\_smaller(mt\_rand));

}

std::uniform\_int\_distribution uid\_bigger(max,99999999);

while(keys\_search.size()<bigger)

{

if(uid\_chance(mt\_rand)==1){

rep=uid\_repeat(mt\_rand);

for(int i=0; i<rep && keys\_search.size()<bigger; i++)

keys\_search.push\_back(uid\_bigger(mt\_rand));

}

else keys\_search.push\_back(uid\_bigger(mt\_rand));

}

while(keys\_search.size() < 100)

{

int key = uid\_inside(mt\_rand);

if(existing\_keys.find(key)==existing\_keys.end()){

if(uid\_chance(mt\_rand)==1){

rep=uid\_repeat(mt\_rand);

for(int i=0; i<rep && keys\_search.size()<100; i++)

keys\_search.push\_back(key);

}

else keys\_search.push\_back(key);

}

}

}

}

std::cout << "KEYS FOR SEARCH" << std::endl;

for(size\_t i=0; i<keys\_search.size(); i++)

std::cout << keys\_search[i] << std::endl;

{

std::mt19937 mt\_rand(time(NULL));

std::shuffle(keys\_search.begin(), keys\_search.end(),mt\_rand);

}

int found=0;

SPLAY::SplayTree::counter\_cmp = 0;

BinarySearchTree::counter\_cmp = 0;

size\_t sum1=0;

size\_t sum2=0;

std::cout << "\tSearching 100 keys:" << std::endl;

std::map<int,size\_t> repeated;

for(size\_t i=0; i<keys\_search.size(); i++)

{

int key = keys\_search[i];

bool flag = false;

if(splay\_tree->search(key)) {

found++;

flag = true;

}

bin\_tree->search(key);

std::cout << i << ". Key:" << key;

auto rep = repeated.find(key);

if(rep != repeated.end())

{

std::cout << "(REP^" << rep->second << ')';

rep->second = i;

}

else repeated.insert(std::make\_pair(key,i));

std::cout << " - ";

if(flag) std::cout << "Found" << std::endl;

else std::cout << "Not found" << std::endl;

std::cout << "Count of comparing keys in SplayTree: " << SPLAY::SplayTree::counter\_cmp << std::endl;

std::cout << "Count of comparing keys int non Balanced Binary Tree: " << BinarySearchTree::counter\_cmp << std::endl;

sum1+=SPLAY::SplayTree::counter\_cmp;

sum2+=BinarySearchTree::counter\_cmp;

std::cout << '\n';

std::cout << "////////////////////////" << std::endl;

std::cout << '\n';

SPLAY::SplayTree::counter\_cmp=0;

BinarySearchTree::counter\_cmp=0;

}

std::cout << "////////////////////////" << std::endl;

std::cout << "//////////////////" << std::endl;

std::cout << "////////////////////////" << std::endl;

std::cout << "Count of found keys: " << found << std::endl;

std::cout << "Wasn't found:" << keys\_search.size() - found << std::endl;

std::cout << "Sum of cmp in SplayTree: " << sum1 << std::endl;

std::cout << "Sum of cmp in non Balanced Binary Tree: " << sum2 << std::endl;

file.close();

std::cin.get();

return 0;

}