|  |
| --- |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **"МИРЭА – Российский технологический университет"**  **РТУ МИРЭА** |

Институт кибербезопасности и цифровых технологий

Кафедра КБ-2 «Институт кибербезопасности и цифровых технологий»

**ОТЧЕТ   
о выполнении домашней работе**

**«Деревья двоичного поиска»**

**по дисциплине   
«Технологии и методы программирования»**

**Вариант № 46**

Выполнил: студент 2 курса

группы БББО-01-21

Мысливец Л.В.

шифр 21Б0746

Москва 2022 г.

**Задание на № 1.**

В рамках домашней работы №1 требуется программно Реализовать в виде программы абстрактный тип данных «Дерево» согласно варианту.

Пусть А, В, С – деревья соответствующего типа, узлы которых могут содержать целочисленные значения. Требуется реализовать начальное формирование деревьев А и В, путем добавления некоторой последовательности значений (узлов) в пустое дерево. После чего требуется по варианту реализовать заданную операцию над деревьями без использования каких-либо вспомогательных структур работая только с узлами деревьев А и В.

Операция А=A ⋃прB означает, что элементы дерева В будут добавлены в дерево А в прямом порядке обхода дерева В, соответственно А=A ⋃обрB – в обратном, а А=A ⋃симB – симметричном обходе дерева В.

Операция А = A ⋂ B означает, что из дерева А исключаются узлы, отсутствующие в дереве В.

Операция А = A \ B означает, что из дерева А исключаются узлы, присутствующие в дереве В.

**Вариант № 46.**

**Тип дерева: Оптимальное дерево двоичного поиска,** **А–обратный, В – симметричный**

**Вывод деревьев на экран: Левый сын, правый брат (таблица, массив)**

**Алгоритм: С=A ⋃прB**

**Теория о графах.**

**Дерево —** это совокупность элементов, называемых узлами (один из которых определен как корень), и отношений ("родительских"), образующих иерархическую структуру узлов.

Узлы, так же, как и элементы списков, могут быть элементами любого типа.

**Обход деревьев.**

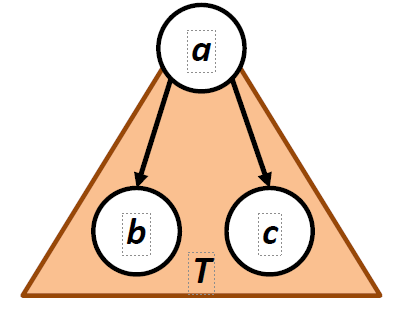
Существует несколько способов обхода (прохождения) всех узлов дерева.

Обход узлов дерева равнозначен упорядочиванию по какому-либо правилу этих узлов.

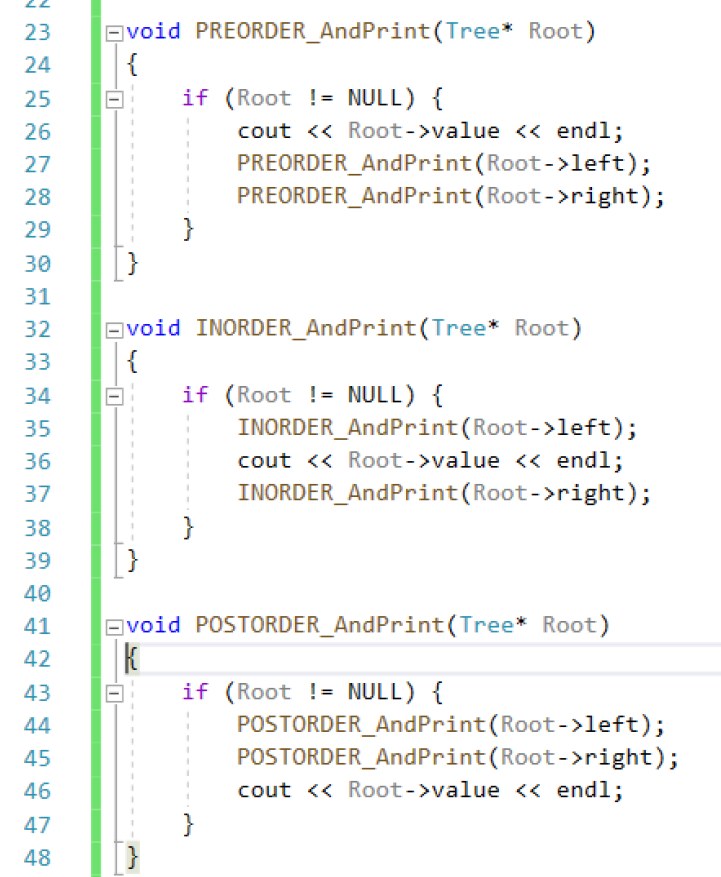
Три наиболее часто используемых способа обхода дерева называются

* обход в прямом порядке,
* обход в обратном порядке;
* обход во внутреннем порядке (или симметричным обходом).

Ниже приведен пример дерева:



Все три способа обхода можно рекурсивно определить следующим образом:



**Бинарное древо.**

**Двоичное дерево поиска (ДДП)** организуется по принципу **–** для произвольного внутреннего узла в левое поддерево будем помещать меньшие значения, а в правые – большие**.**

**Оптимальное ДДП.**

Дерево поиска называется оптимальным, если его цена минимальна. То есть оптимальное бинарное дерево поиска –это бинарное дерево поиска, построенное в расчете на обеспечение максимальной производительности при заданном распределении вероятностей поиска требуемых данных.

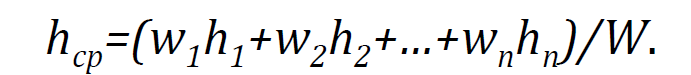
Существует подход построения оптимальных деревьев поиска, при котором элементы вставляются в порядке уменьшения частот, что дает в среднем неплохие деревья поиска. Однако этот подход может дать вырожденное дерево поиска, которое будет далеко от оптимального. Еще один подход состоит в выборе корня k таким образом, чтобы максимальная сумма вероятностей для вершин левого поддерева или правого поддерева была настолько мала, насколько это возможно. Такой подход также может оказаться плохим в случае выбора в качестве корня элемента с малым значением pk.

Припишем каждой вершине дерева Vi вес wi,

пропорциональный частоте поиска этой вершины. Сумма

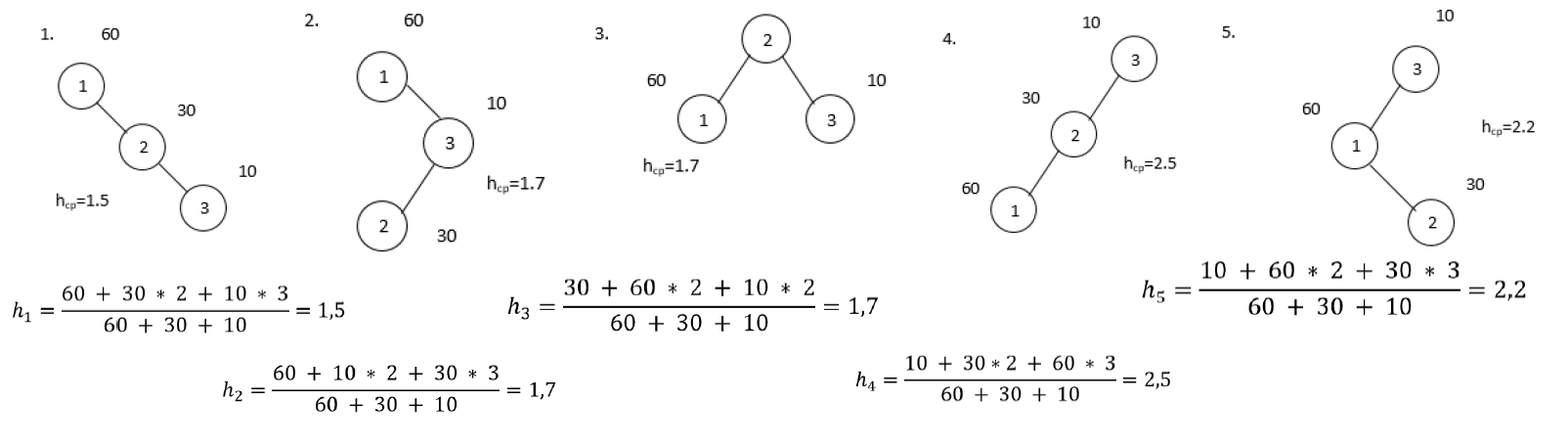
весов всех вершин дает вес дерева W. Каждая вершина Vi расположена на высоте hi, корень расположен на высоте 1.

Высота вершины равна количеству операций сравнения, необходимых для поиска этой вершины. Определим средневзвешенную высоту дерева с n вершинами следующим образом:



Дерево поиска, имеющее минимальную средневзвешенную высоту, называется деревом оптимального поиска.

***Пример***. Рассмотрим множество из трех ключей V1=1, V2=2, V3=3 со следующими весами: w1=60, w2=30, w3=10, W=100. Эти три ключа можно расставить в дереве поиска пятью различными способами.



**Листинг программы с реализацией АТД «Оптимальное ДДП».**

**class** **OBST{**

private static final **int** NMAX **=** 20**;**

static **class** **Node{**

**int** KEY**;**

Node left**,** right**;**

**}**

**//**static Node ROOT**=**new Node**();**

**int[][]** C **=**new **int[**NMAX**][**NMAX**];** **//**cost matrix

**int[][]** W **=**new **int[**NMAX**][**NMAX**];** **//**weight matrix

**int[][]** R **=**new **int[**NMAX**][**NMAX**];** **//**root matrix

**//**static **int[]** q **=**new **int[**NMAX**];** **//**unsuccesful searches

**int[]** p **=**new **int[**NMAX**];** **//**frequencies

**//**static **int** NUMBER\_OF\_KEYS**;** **//**number of keys **in** the tree

public Node ROOT**=**new Node**();**

**int[]** KEYS **=**new **int[**NMAX**];**

void COMPUTE\_W\_C\_R**(int** NUMBER\_OF\_KEYS**){**

**int** x**,** **min;**

**int** i**,** j**,** k**,** h**,** m**;**

**//**Construct weight matrix W

**for(**i **=** 0**;** i **<=** NUMBER\_OF\_KEYS**;** i**++)** **{**

W**[**i**][**i**]** **=** 0**;**

**for(**j **=** i **+** 1**;** j **<=** NUMBER\_OF\_KEYS**;** j**++)**

W**[**i**][**j**]** **=** W**[**i**][**j**-**1**]** **+** p**[**j**];**

**}**

**//**Construct cost matrix C **and** root matrix R

**for(**i **=** 0**;** i **<=** NUMBER\_OF\_KEYS**;** i**++)**

C**[**i**][**i**]** **=** W**[**i**][**i**];**

**for(**i **=** 0**;** i **<=** NUMBER\_OF\_KEYS **-** 1**;** i**++)** **{**

j **=** i **+** 1**;**

C**[**i**][**j**]** **=** C**[**i**][**i**]** **+** C**[**j**][**j**]** **+** W**[**i**][**j**];**

R**[**i**][**j**]** **=** j**;**

**}**

**for(**h **=** 2**;** h **<=** NUMBER\_OF\_KEYS**;** h**++)**

**for(**i **=** 0**;** i **<=** NUMBER\_OF\_KEYS **-** h**;** i**++)** **{**

j **=** i **+** h**;**

m **=** R**[**i**][**j**-**1**];**

**min** **=** C**[**i**][**m**-**1**]** **+** C**[**m**][**j**];**

**for(**k **=** m**+**1**;** k **<=** R**[**i**+**1**][**j**];** k**++){**

x **=** C**[**i**][**k**-**1**]** **+** C**[**k**][**j**];**

**if(**x **<** **min)** **{**

m **=** k**;**

**min** **=** x**;**

**}**

**}**

C**[**i**][**j**]** **=** W**[**i**][**j**]** **+** **min;**

R**[**i**][**j**]** **=** m**;**

**}**

**//**Display weight matrix W

System**.**out**.**println**(**"\nThe weight matrix W:"**);**

**for(**i **=** 0**;** i **<=** NUMBER\_OF\_KEYS**;** i**++){**

**for(**j **=** i**;** j **<=** NUMBER\_OF\_KEYS**;** j**++)**

System**.**out**.print(**W**[**i**][**j**]+**" "**);**

System**.**out**.print(**"\n"**);**

**}**

**//**Display Cost matrix C

System**.**out**.print(**"\nThe cost matrix C:\n"**);**

**for(**i **=** 0**;** i **<=** NUMBER\_OF\_KEYS**;** i**++)** **{**

**for(**j **=** i**;** j **<=** NUMBER\_OF\_KEYS**;** j**++)**

System**.**out**.print(**C**[**i**][**j**]+**" "**);**

System**.**out**.print(**"\n"**);**

**}**

**//**Display root matrix R 8

System**.**out**.print(**"\nThe root matrix R:\n"**);**

**for(**i **=** 0**;** i **<=** NUMBER\_OF\_KEYS**;** i**++)** **{**

**for(**j **=** i**;** j **<=** NUMBER\_OF\_KEYS**;** j**++)**

System**.**out**.print(**R**[**i**][**j**]+**" "**);**

System**.**out**.print(**"\n"**);**

**}**

**}**

**//**Construct the optimal binary search tree

public Node CONSTRUCT\_OBST**(int** i**,** **int** j**){**

Node p**=**new Node**();**

**if(**i **==** j**)**

p **=** null**;**

**else{**

p**.**KEY **=** KEYS**[**R**[**i**][**j**]];**

p**.**left **=** CONSTRUCT\_OBST**(**i**,** R**[**i**][**j**]** **-** 1**);** **//**left subtree

p**.**right **=** CONSTRUCT\_OBST**(**R**[**i**][**j**],** j**);** **//**right subtree

**}**

**return** p**;**

**}**

**//**Display the optimal binary search tree

void preOrder**(**Node ROOT**){**

**if(**ROOT **!=** null**)** **{**

System**.**out**.print(**ROOT**.**KEY**+**" "**);**

preOrder**(**ROOT**.**left**);**

preOrder**(**ROOT**.**right**);**

**}**

**}**

ArrayList**<**Integer**>** B **=** new ArrayList**<>();**

void inOrder**(**Node ROOT**){**

**if(**ROOT **!=** null**)** **{**

inOrder**(**ROOT**.**left**);**

B**.**add**(**ROOT**.**KEY**);**

inOrder**(**ROOT**.**right**);**

**}**

**}**

ArrayList**<**Integer**>** A **=** new ArrayList**<>();**

void postOrder**(**Node ROOT**){**

**if(**ROOT **!=** null**)** **{**

postOrder**(**ROOT**.**left**);**

postOrder**(**ROOT**.**right**);**

A**.**add**(**ROOT**.**KEY**);**

**}**

**}**

static void DISPLAY**(**Node ROOT**,** **int** nivel**){**

**int** i**;**

**if(**ROOT **!=** null**)** **{**

DISPLAY**(**ROOT**.**left**,** nivel **+** 1**);**

**for(**i **=** 0**;** i **<=** nivel**;** i**++)**

System**.**out**.print(**" "**);**

System**.**out**.**println**(**ROOT**.**KEY**);**

DISPLAY**(**ROOT**.**right**,** nivel**+**1**);**

**}**

**}**

void OPTIMAL\_BINARY\_SEARCH\_TREE**(int** NUMBER\_OF\_KEYS**)**

**{**

**float** average\_cost\_per\_weight**;**

COMPUTE\_W\_C\_R**(**NUMBER\_OF\_KEYS**);**

System**.**out**.**println**(**"C[0]="**+**C**[**0**][**NUMBER\_OF\_KEYS**]+**" "**+**"W[0]="**+**W**[**0**][**NUMBER\_OF\_KEYS**]);**

average\_cost\_per\_weight **=** C**[**0**][**NUMBER\_OF\_KEYS**]/(float)**W**[**0**][**NUMBER\_OF\_KEYS**];**

System**.**out**.**println**(**"The cost per weight ratio is:\n" **+** average\_cost\_per\_weight**);**

ROOT **=** CONSTRUCT\_OBST**(**0**,** NUMBER\_OF\_KEYS**);**

inOrder**(**ROOT**);**

postOrder**(**ROOT**);**

System**.**out**.**println**(**"\nTree"**);**

DISPLAY**(**ROOT**,**0**);**

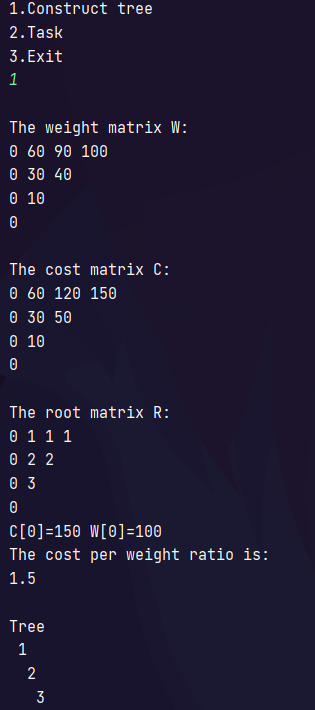
**}**

**Скриншот работы программы:**

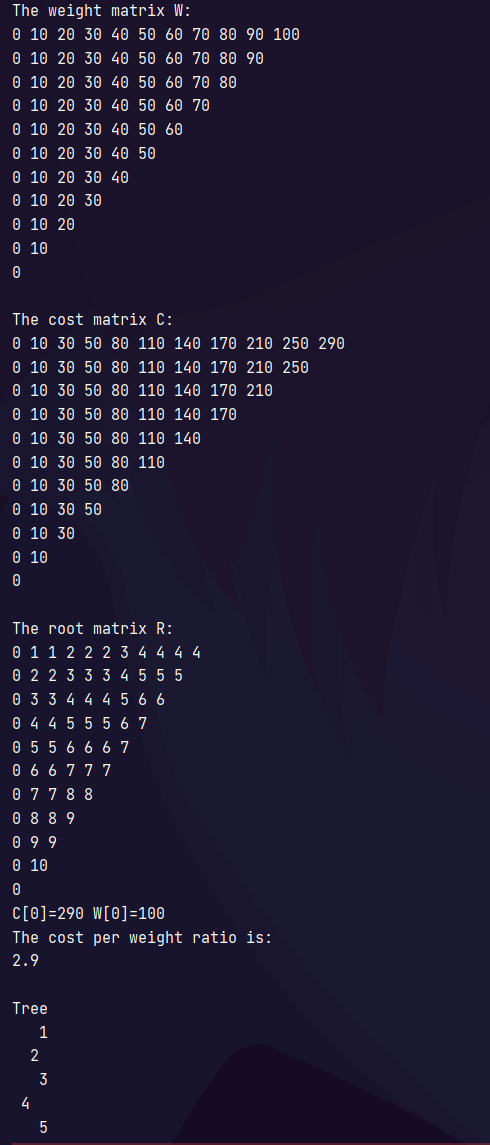
Скриншот 1(ввод данных производится с клавиатуры):



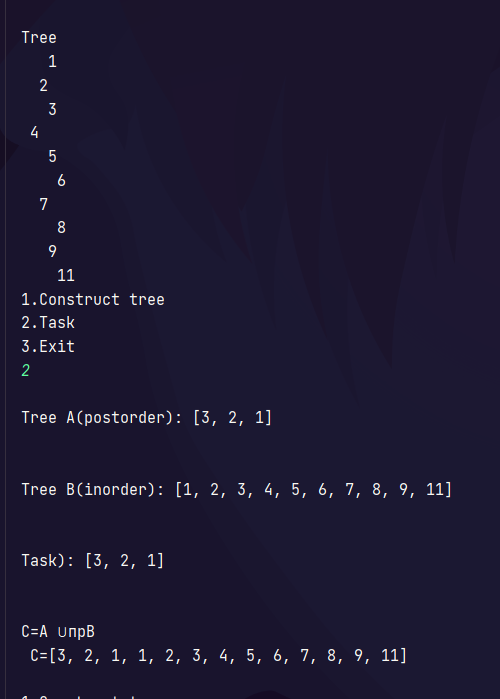
Скриншот 2(построение матриц весов и корней для ОДДП А и и построение дерева):



Скриншот 3(построение матриц весов и корней для ОДДП Б и и построение):



Скриншот 4(перебор деревьев и выполнение операции над ними):



**Выводы.**

По результатам экспериментов было установлено, структура ДДП очень эффективна для реализации абстрактных типов данных, таких как динамические наборы, таблицы поиска и очереди приоритетов, и использоваться в алгоритмах сортировки, таких как сортировка по дереву. Производительность бинарного дерева поиска зависит от порядка вставки узлов в дерево, поскольку произвольные вставки могут привести к вырождению; можно построить несколько вариантов бинарного дерева поиска с гарантированной наихудшей производительностью. BST с гарантированной сложностью в наихудшем случае работают лучше, чем несортированный массив, который потребовал бы линейного времени поиска поэтому.

**Литература:**

1. Ахо А. В. Структуры данных и алгоритмы. – Издательский дом Вильямс, 2000.
2. Белов В. В., Воробьев Е. М., Шаталов В. Е. - Теория графов. — М.: Высш. школа, 1976. — С. 392.
3. Берж К. - Теория графов и ее приложения. М.: ИЛ, 1962. 320c.

**Приложение 1. Листинг программы**

**import** java**.**io**.**Console**;**

**import** java**.**io**.**IOException**;**

**import** java**.**util**.\*;**

**class** **OBST{**

private static final **int** NMAX **=** 20**;**

static **class** **Node{**

**int** KEY**;**

Node left**,** right**;**

**}**

**//**static Node ROOT**=**new Node**();**

**int[][]** C **=**new **int[**NMAX**][**NMAX**];** **//**cost matrix

**int[][]** W **=**new **int[**NMAX**][**NMAX**];** **//**weight matrix

**int[][]** R **=**new **int[**NMAX**][**NMAX**];** **//**root matrix

**//**static **int[]** q **=**new **int[**NMAX**];** **//**unsuccesful searches

**int[]** p **=**new **int[**NMAX**];** **//**frequencies

**//**static **int** NUMBER\_OF\_KEYS**;** **//**number of keys **in** the tree

public Node ROOT**=**new Node**();**

**int[]** KEYS **=**new **int[**NMAX**];**

void COMPUTE\_W\_C\_R**(int** NUMBER\_OF\_KEYS**){**

**int** x**,** **min;**

**int** i**,** j**,** k**,** h**,** m**;**

**//**Construct weight matrix W

**for(**i **=** 0**;** i **<=** NUMBER\_OF\_KEYS**;** i**++)** **{**

W**[**i**][**i**]** **=** 0**;**

**for(**j **=** i **+** 1**;** j **<=** NUMBER\_OF\_KEYS**;** j**++)**

W**[**i**][**j**]** **=** W**[**i**][**j**-**1**]** **+** p**[**j**];**

**}**

**//**Construct cost matrix C **and** root matrix R

**for(**i **=** 0**;** i **<=** NUMBER\_OF\_KEYS**;** i**++)**

C**[**i**][**i**]** **=** W**[**i**][**i**];**

**for(**i **=** 0**;** i **<=** NUMBER\_OF\_KEYS **-** 1**;** i**++)** **{**

j **=** i **+** 1**;**

C**[**i**][**j**]** **=** C**[**i**][**i**]** **+** C**[**j**][**j**]** **+** W**[**i**][**j**];**

R**[**i**][**j**]** **=** j**;**

**}**

**for(**h **=** 2**;** h **<=** NUMBER\_OF\_KEYS**;** h**++)**

**for(**i **=** 0**;** i **<=** NUMBER\_OF\_KEYS **-** h**;** i**++)** **{**

j **=** i **+** h**;**

m **=** R**[**i**][**j**-**1**];**

**min** **=** C**[**i**][**m**-**1**]** **+** C**[**m**][**j**];**

**for(**k **=** m**+**1**;** k **<=** R**[**i**+**1**][**j**];** k**++){**

x **=** C**[**i**][**k**-**1**]** **+** C**[**k**][**j**];**

**if(**x **<** **min)** **{**

m **=** k**;**

**min** **=** x**;**

**}**

**}**

C**[**i**][**j**]** **=** W**[**i**][**j**]** **+** **min;**

R**[**i**][**j**]** **=** m**;**

**}**

**//**Display weight matrix W

System**.**out**.**println**(**"\nThe weight matrix W:"**);**

**for(**i **=** 0**;** i **<=** NUMBER\_OF\_KEYS**;** i**++){**

**for(**j **=** i**;** j **<=** NUMBER\_OF\_KEYS**;** j**++)**

System**.**out**.print(**W**[**i**][**j**]+**" "**);**

System**.**out**.print(**"\n"**);**

**}**

**//**Display Cost matrix C

System**.**out**.print(**"\nThe cost matrix C:\n"**);**

**for(**i **=** 0**;** i **<=** NUMBER\_OF\_KEYS**;** i**++)** **{**

**for(**j **=** i**;** j **<=** NUMBER\_OF\_KEYS**;** j**++)**

System**.**out**.print(**C**[**i**][**j**]+**" "**);**

System**.**out**.print(**"\n"**);**

**}**

**//**Display root matrix R 8

System**.**out**.print(**"\nThe root matrix R:\n"**);**

**for(**i **=** 0**;** i **<=** NUMBER\_OF\_KEYS**;** i**++)** **{**

**for(**j **=** i**;** j **<=** NUMBER\_OF\_KEYS**;** j**++)**

System**.**out**.print(**R**[**i**][**j**]+**" "**);**

System**.**out**.print(**"\n"**);**

**}**

**}**

**//**Construct the optimal binary search tree

public Node CONSTRUCT\_OBST**(int** i**,** **int** j**){**

Node p**=**new Node**();**

**if(**i **==** j**)**

p **=** null**;**

**else{**

p**.**KEY **=** KEYS**[**R**[**i**][**j**]];**

p**.**left **=** CONSTRUCT\_OBST**(**i**,** R**[**i**][**j**]** **-** 1**);** **//**left subtree

p**.**right **=** CONSTRUCT\_OBST**(**R**[**i**][**j**],** j**);** **//**right subtree

**}**

**return** p**;**

**}**

**//**Display the optimal binary search tree

void preOrder**(**Node ROOT**){**

**if(**ROOT **!=** null**)** **{**

System**.**out**.print(**ROOT**.**KEY**+**" "**);**

preOrder**(**ROOT**.**left**);**

preOrder**(**ROOT**.**right**);**

**}**

**}**

ArrayList**<**Integer**>** B **=** new ArrayList**<>();**

void inOrder**(**Node ROOT**){**

**if(**ROOT **!=** null**)** **{**

inOrder**(**ROOT**.**left**);**

B**.**add**(**ROOT**.**KEY**);**

inOrder**(**ROOT**.**right**);**

**}**

**}**

ArrayList**<**Integer**>** A **=** new ArrayList**<>();**

void postOrder**(**Node ROOT**){**

**if(**ROOT **!=** null**)** **{**

postOrder**(**ROOT**.**left**);**

postOrder**(**ROOT**.**right**);**

A**.**add**(**ROOT**.**KEY**);**

**}**

**}**

static void DISPLAY**(**Node ROOT**,** **int** nivel**){**

**int** i**;**

**if(**ROOT **!=** null**)** **{**

DISPLAY**(**ROOT**.**left**,** nivel **+** 1**);**

**for(**i **=** 0**;** i **<=** nivel**;** i**++)**

System**.**out**.print(**" "**);**

System**.**out**.**println**(**ROOT**.**KEY**);**

DISPLAY**(**ROOT**.**right**,** nivel**+**1**);**

**}**

**}**

void OPTIMAL\_BINARY\_SEARCH\_TREE**(int** NUMBER\_OF\_KEYS**)**

**{**

**float** average\_cost\_per\_weight**;**

COMPUTE\_W\_C\_R**(**NUMBER\_OF\_KEYS**);**

System**.**out**.**println**(**"C[0]="**+**C**[**0**][**NUMBER\_OF\_KEYS**]+**" "**+**"W[0]="**+**W**[**0**][**NUMBER\_OF\_KEYS**]);**

average\_cost\_per\_weight **=** C**[**0**][**NUMBER\_OF\_KEYS**]/(float)**W**[**0**][**NUMBER\_OF\_KEYS**];**

System**.**out**.**println**(**"The cost per weight ratio is:\n" **+** average\_cost\_per\_weight**);**

ROOT **=** CONSTRUCT\_OBST**(**0**,** NUMBER\_OF\_KEYS**);**

inOrder**(**ROOT**);**

postOrder**(**ROOT**);**

System**.**out**.**println**(**"\nTree"**);**

DISPLAY**(**ROOT**,**0**);**

**}**

**}**

**class** **Main{**

static Scanner sc **=** new Scanner**(**System**.in);**

static OBST treeA **=** new OBST**();**

**//**static OBST**.**Node nodeA **=** new OBST**.**Node**();**

static OBST treeB **=** new OBST**();**

**//**static OBST**.**Node nodeB **=** new OBST**.**Node**();**

public static void MENU**(**OBST treeA**,**OBST treeB**,** **int** NUMBER\_OF\_KEYSA**,** **int** NUMBER\_OF\_KEYSB**)** **{**

**int** k **=** 0**;**

do **{**

System**.**out**.print(**"1.Construct tree\n2.Task\n3.Exit\n"**);**

k**=**sc**.**nextInt**();**

switch**(**k**)** **{**

case 1**:**

treeA**.**OPTIMAL\_BINARY\_SEARCH\_TREE**(**NUMBER\_OF\_KEYSA**);**

treeB**.**OPTIMAL\_BINARY\_SEARCH\_TREE**(**NUMBER\_OF\_KEYSB**);**

**break;**

case 2**:**

System**.**out**.**println**(**"\nTree A(postorder): " **+**treeA**.**A**+**"\n"**);**

System**.**out**.**println**(**"\nTree B(inorder): " **+**treeB**.**B**+**"\n"**);**

System**.**out**.**println**(**"\nTask): " **+**treeA**.**A**+**"\n"**);**

treeA**.**A**.**addAll**(**treeB**.**B**);**

System**.**out**.**println**(**"\nС=A ⋃прB\n C="**+**treeA**.**A**+**"\n"**);**

**break;**

case 3**:**

k **=** **-**1**;**

**break;**

**}**

**}** **while** **(**k **!=** **-**1**);**

**}**

public static void PRINT\_NUMBER\_OF\_KEYS**(int** NUMBER\_OF\_KEYS**,**OBST tree**){**

**for(int** i **=** 1**;** i **<=** NUMBER\_OF\_KEYS**;** i**++)** **{**

System**.**out**.print(**"key["**+**i**+**"]="**);**

tree**.**KEYS**[**i**]=**sc**.**nextInt**();**

System**.**out**.print(**" frequency = "**);**

tree**.**p**[**i**]=**sc**.**nextInt**();**

**}**

**}**

public static void main**(**String**[]** args**)** **{**

System**.**out**.print(**"Input number of keys A: "**);**

**int** NUMBER\_OF\_KEYSA**=**sc**.**nextInt**();**

PRINT\_NUMBER\_OF\_KEYS**(**NUMBER\_OF\_KEYSA**,**treeA**);**

System**.**out**.print(**"Input number of keys B: "**);**

**int** NUMBER\_OF\_KEYSB**=**sc**.**nextInt**();**

PRINT\_NUMBER\_OF\_KEYS**(**NUMBER\_OF\_KEYSB**,**treeB**);**

MENU**(**treeA**,**treeB**,**NUMBER\_OF\_KEYSA**,**NUMBER\_OF\_KEYSB**);**

**}**

**}**