|  |
| --- |
| C:\Users\mabarkat\Documents\KSU_LOGO.BMPKing Saud University  College of Computer and Information Sciences  Department of Computer Science  CSC 212 Data Structures Project Report – 1st Semester 2016-2017  Developing a Ratings Query Application |

**Authors**

|  |  |  |
| --- | --- | --- |
| Name | ID | Section |
| Areej Almalki | **435202033** | **41196** |
| Haya Alwahhabi | **435202510** | **41196** |
| Tarfah Alghofili | **435200281** | **41196** |

# Introduction

The hole idea of this project is how to store data using the right data structure in an appropriate manner and in a way to avoid memory consumption ,also to retrieve this data in an efficient way .

In phase one we used ADTs in the implementation

And for phase two we used ADT+BST+Heap and added some functions for more efficient program

this report will contain the ADTs ,BSTs,HEAPs specification format and how we designed this project ,also we will clarify some  of the methods that need to be clarified

a summary of the project and the source code

# Specification

**Elements:** The elements are of generic type <Type> (The elements are placed in nodes for linked list implementation).

**Structure:** the elements are linearly arranged. The first element is called head, there is a element called current.

**Domain:** the number of elements in the list is bounded therefore the domain is finite. Type name of elements in the domain: List

**Operations:** We assume all operations operate on a list L.

**Method** FindFirst ( )

**requires:** list L is not empty. **input:** none

**results:** first element set as the current element. **output:** none.

**Method** FindNext ( )

**requires**: list L is not empty. Cur is not last. **input**: none

**results**: element following the current element is made the current element.

**output**: none.

**Method** Retrieve (Type e)

**requires**: list L is not empty. **input**: none

**results**: current element is copied into e. **output**: element e.

**Method** Update (Type e).

**requires**: list L is not empty. **input**: e.

**results**: the element e is copied into the current node.

**output**: none.

**Method** Insert (Type e).

**requires**: list L is not full. **input**: e.

**results**: a new node containing element e is created and inserted after the current element in the list. The new element e is made the current element. If the list is empty e is also made the head element. **output**: none.

**Method** Remove ( )

**requires**: list L is not empty. **input**: none

**results**: the current element is removed from the list. If the resulting list is empty current is set to NULL. If successor of the deleted element exists it is made the new current element otherwise first element is made the new current element. **output**: none.

**Method** Full (boolean flag)

**input**: none. **returns**: if the number of elements in L has reached the maximum number allowed then flag is set to true otherwise false. **output**: flag.

**Method** Empty (boolean flag).

**input**: none. **results**: if the number of elements in L is zero, then flag is set to true otherwise false.

**Output**: flag.

**Method** Last (boolean flag).

**input**: none. **requires**: L ist not empty. **Results**: if the last element is the current element then flag is set to true otherwise false. **Output**: flag

# ADT Heap:

**Elements:** The elements are called HeapElements.

**Structure**: The elements of the heap satisfy the heap conditions.

**Domain:** Bounded. Type name: Heap**.**

**Operations:**

**Method** SiftUp ()  
**Input**: none. **requires**: Elements H[1],H[2],…,H[n-1] satisfy heap conditions.  
**results:** Elements H[1],H[2],…,H[n] satisfy heap conditions. **Output**: none.

**Method** SiftDown (int i)  
**Input**: i. **requires**: Elements H[i+1],H[i+2],…,H[n] satisfy the heap conditions.  
**results**: Elements H[i],H[i+1],…,H[n] satisfy the heap conditions. **Output**: none.

**Method** Insert(int key, T data)  
**input**: key, data. **requires**: Elements H[1],H[2],…,H[n] satisfy heap conditions.  
**results**: The key and data are inserted in H[n+1]. Elements H[1],H[2],….H[n+1] must satisfy the heap conditions. **Output**: none

**Method** RemoveRoot(HeapElement<T> result)   
**input**: none. **requires**: Elements H[1],H[2],…,H[n] satisfy heap condition.  
**results**: The HeapElement in H[1] is removed, and it is value is assigned to result. Elements H[1],H[2],….H[n-1] must satisfy the heap conditions. **output**: none.

**Method** Full(boolean result).

**Method** Size(int result).

# ADT Binary Search Tree:

**Elements:** the elements are nodes (BSTNode), each node contains the following data type: Type,Key and has LeftChild and RightChild references. .

**Structure:** hierarchical structure; each node can have two children: left or right child; there is a root node and a current node. If N is any node in the tree, nodes in the left subtree < N and nodes in the right subtree > N.

**Domain:** the number of nodes in a BST is bounded; type/class name is BST.

**Operations:**

**Method** FindKey (int tkey, boolean found).  
**requires:** none.  
**input:** tkey.

**results:** If bst contains a node whose key value is tkey, then that node is made the current node and found is set to true; otherwise found is set to false and either the tree is empty or the current node is the node to which the node with key = tkey would be attached as a child if it were added to the BST.  
**output:** found.

**Method** Insert (int k, Type e, boolean inserted)

**requires**: Full (bst) is false. **input**: key, e.

**results**: if bst does not contain k then inserted is set to true and node with k and e is inserted and made the current element; otherwise inserted is set to false and current value does not change. **output**: inserted.

**Method** Remove\_Key (int tkey, boolean removed) **input**: tkey

**results:** Node with key value tkey is removed from the bst and removed set to true. If BST is not empty then root is made the current.

**output**: removed.

**Method** Update(int key, Type e, boolean updated)

**requires:** Empty(bst) is false. **input:** key, e. **results:** current node’s element is replaced with e.

**Output:** updated.

**Method** Traverse (Order ord)

**Method** DeleteSub ( ).

**Method** Retrieve (Type e).

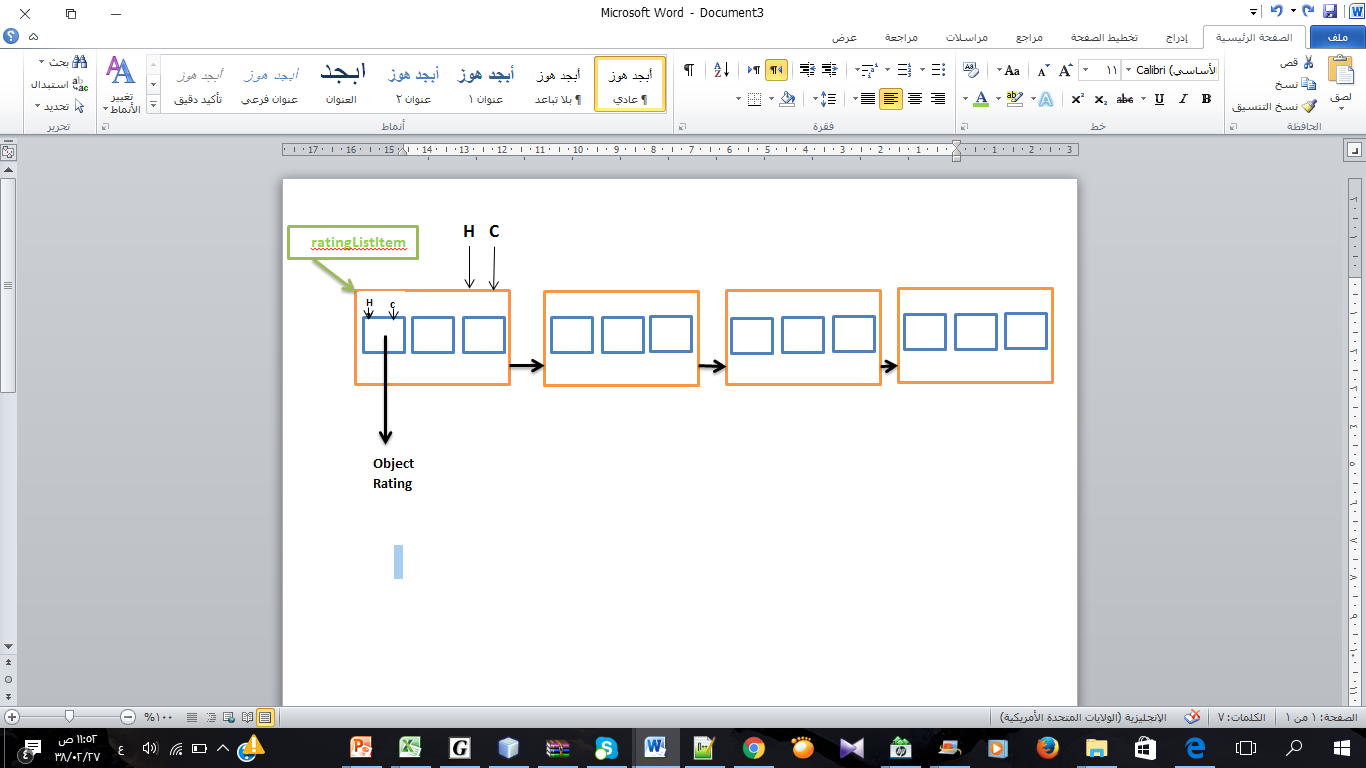
**Method** Empty ( boolean empty ).

**Method** Full (boolean full).

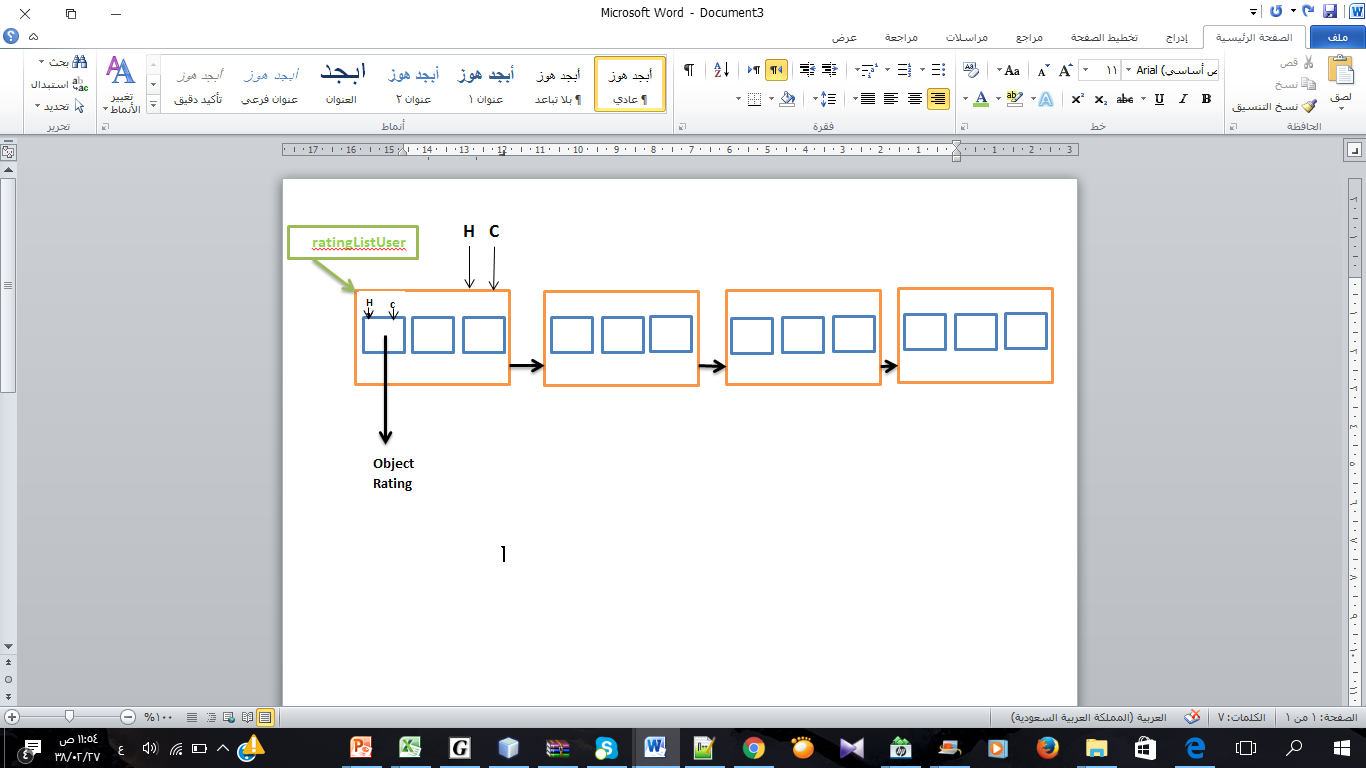
# Design

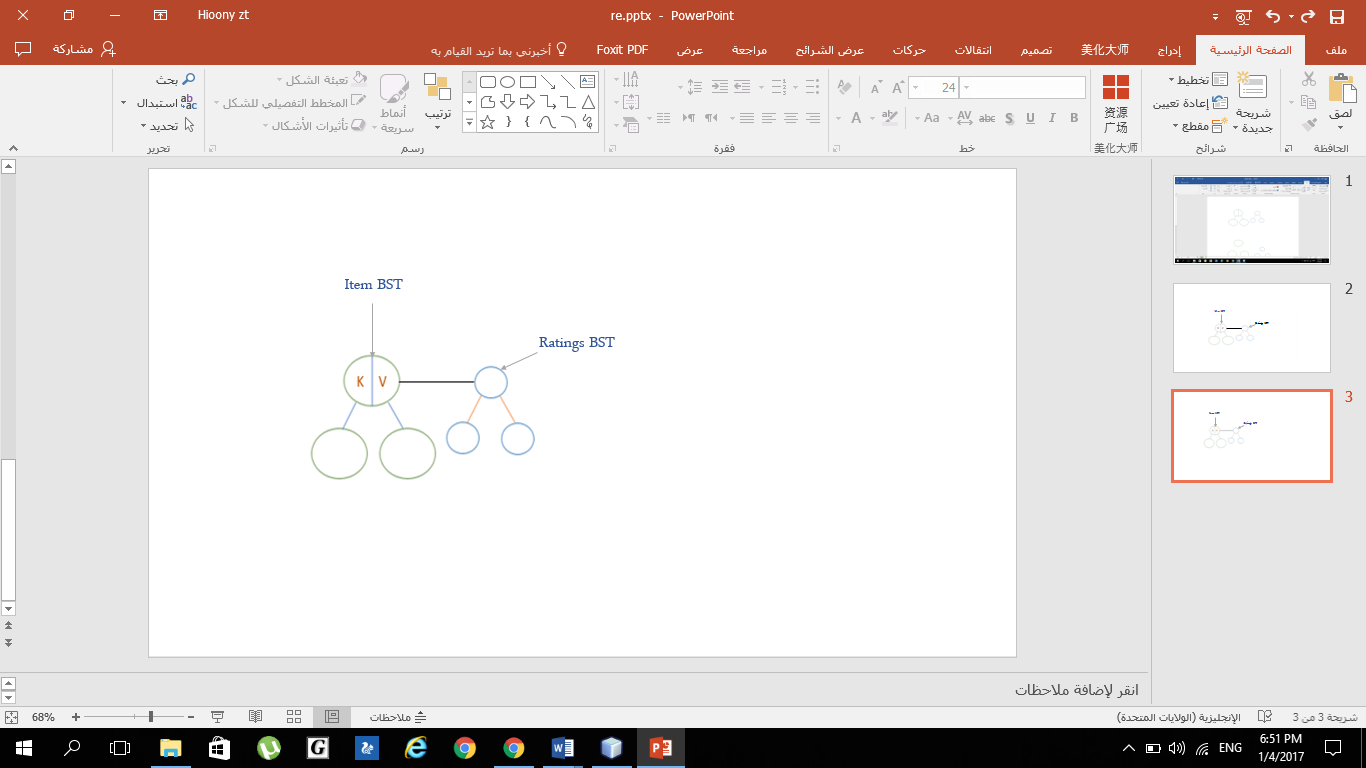
In phase one we have two linked lists ,one for the user and one for the item ,each one has a linked list of ratings ,for Item linked list the the item id in each rating is the same ,for user linked list the user id in each rating is the same .

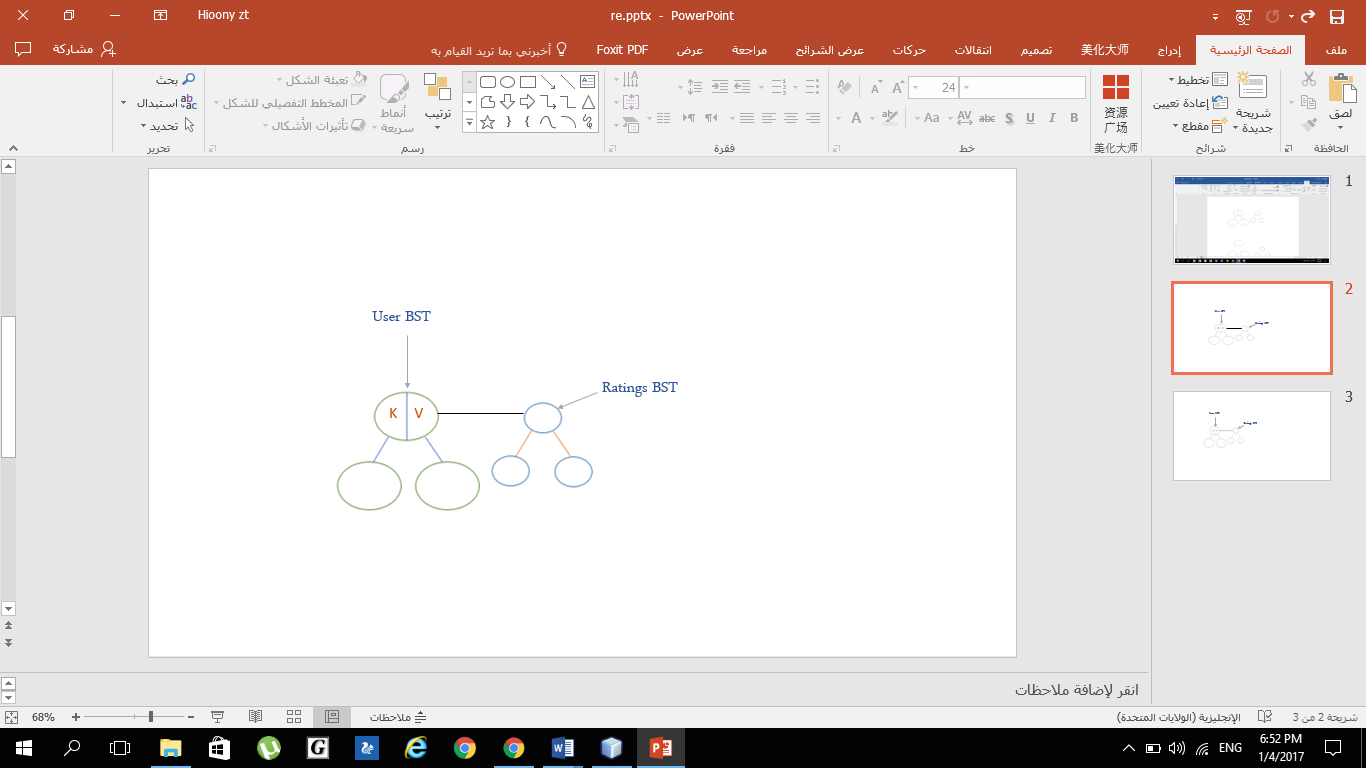
* **Item linked list**



* **User linked list**

In phase two we we have two BSTs ,one for the user and one for the item ,each one has a key (witch is the ID ) and the value is a BST of ratings ,for Item BST the item id in each rating is the same ,for user BST the user id in each rating is the same .





# Implementation for phase 1

Public RatingManager ()

We declared 2 linked lists of ratings one for the user and one for the item

And we declared two counters for the user and item .

Public static RatingManger read (String fileName)

We read the rating from the file using scanner and we saved the rating in an object of type Rating ,then we added this object to the item llist and user list

Public void addRating (Rating rating)

1-we devided the method into two sections …add rating item and add rating user

2-in add rating item the ratings in each node has the same item id

3-in add rating user the ratings in each node hase the same user is

private void addRatingUser(Rating rating)

1-In this method I receive a user number

2- we make a new list that should contain the rating of the user 3-if the user is found thin I add its ratings to the list and return it ,or I return null.

Public Linkedlist<Rating> getUserRatings(int i)

1-In this method I receive a user number

2- we make a new list that should contain the rating of the user 3-if the user is found thin I add its ratings to the list and return it ,or I return null.

Public Linkedlist<Rating> getItemRatings(int i)

1-In this method I receive an Item number

2- we make a new list that should contain the rating of the Item 3-if the Item is found thin I add its ratings to the list and return it ,or I return null.

public double getAverageItemRating(int i)

1- In this method we receive an item number

2- We make a new list that will contain all item ratings

3- If the item is found it will be inserted in the list and increase the count by 1 and add to the sum

4- Finally if the count is zero then -1 is returned ,else,the average will be calculated

Average =sum/count

public double getAverageUserRating(int i)\*\*\*\*

1- In this method we receive an user number

2- We make a new list that will contain all user ratings

3- If the user is found it will be inserted in the list and increase the count by 1 and add to the sum

4- Finally if the count is zero then -1 is returned ,else,the average will be calculated

Average =sum/count

Public linkedList<Integer> getHighestRatingItem()

In this method we will return a linked list that contain the item that have the highest average rating

* **Implementation for phase 2**

Public RatingManager ()

We declared 2 BSTs for the user and item

And we declared two counters for the user and item .

Public static RatingManger read (String fileName)

We read the rating from the file using scanner and we saved the rating in an object of type Rating .

Public void addRating (Rating rating)

We added the rating into the user and item BSTs

Public Linkedlist<Rating> getUserRatings(int i)

1-In this method I receive a user number

2- we make a new list that should contain the rating of the user

3-if the user is found thin I add its ratings to the list and return it ,or I return null.

Public Linkedlist<Rating> getItemRatings(int i)

1-In this method I receive an Item number

2- we make a new list that should contain the rating of the Item

3-if the Item is found thin I add its ratings to the list and return it ,or I return null.

public double getAverageItemRating(int i)

1- In this method we receive an item number

2- We make a new list that will contain all the item ratings

3- If the item is found it will be inserted in the list and increase the count by 1 and add to the sum

4- Finally if the count is zero then -1 is returned ,else,the average will be calculated

Average =sum/count

public double getAverageUserRating(int i)

1- In this method we receive a user number

2- We make a new list that will contain all the user ratings

3- If the user is found its will be inserted in the list and increase the count by 1 and add to the sum

4- Finally if the count is zero then -1 is returned ,else,the average will be calculated

Average =sum/count

Public linkedList<Integer> getHighestRatedItem()

In this method we will return a linked list that contain the items that have the highest average rating

getRating(int i,int j)

this method should return the value that the user I gave to item j

**nbComp(int I,int j)**

**we look for the rating that witch user I gave to item j**

**and in each time we pass a key we raise the count by one,**

**finally the counter is returned.**

**getDist(int ui,int uj )**

**we check if those two users have rated common items if found we calculate the distance ,else we return double infinite .**

**kNNUsers (int I ,LinkedList<Integer> users, int k)**

**after calculating the distance we take the smallest k distance and add it to the list**

**getAverageRating(int j, LinkedList<Integer> users)**

**for item j,we check that the users in the linked List has rated the item**

**if none of the users has rated the item or the list is empty we return the getAverageItemRating for the item j**

# Source Code for phase 1

///////////////////////////\*\* LinkedList and Node \*\*//////////////////////////////////////

1   
 2 class Node<T> {  
 3   
 4 public T data;  
 5 public Node<T> next;  
 6   
 7 public Node(T val) {  
 8 data = val;  
 9 next = null;  
10 }  
11 }  
12   
13 public class LinkedList<T> {  
14   
15 private Node<T> head;  
16 private Node<T> current;  
17   
18 public LinkedList() {  
19 head = current = null;  
20 }  
21   
22 public boolean empty() {  
23 return head == null;  
24 }  
25   
26 public boolean last() {  
27 return current.next == null;  
28 }  
29   
30 public boolean full() {  
31 return false;  
32 }  
33   
34 public void findFirst() {  
35 current = head;  
36 }  
37   
38 public void findNext() {  
39 current = current.next;  
40 }  
41   
42 public T retrieve() {  
43 return current.data;  
44 }  
45   
46 public void update(T val) {  
47 current.data = val;  
48 }  
49   
50 public void insert(T val) {  
51 Node<T> tmp;  
52 if (empty()) {  
53 current = head = new Node<T>(val);  
54 } else {  
55 tmp = current.next;  
56 current.next = new Node<T>(val);  
57 current = current.next;  
58 current.next = tmp;  
59 }  
60 }  
61   
62 public void remove() {  
63 if (current == head) {  
64 head = head.next;  
65 } else {  
66 Node<T> tmp = head;  
67 while (tmp.next != current) {  
68 tmp = tmp.next;  
69 }  
70 tmp.next = current.next;  
71 }  
72 if (current.next == null) {  
73 current = head;  
74 } else {  
75 current = current.next;  
76 }  
77 }  
78   
79 public void display() {  
80 Node<T> tmp = head;  
81 while (tmp != null) {  
82 System.out.print(tmp.data);  
83 tmp = tmp.next;  
84   
85 }  
86 System.out.println();  
87   
88 }  
89   
90 }  
///////////////////////\*\*class Rating\*\*///////////////////////

1 public class Rating {  
 2 private int userId;  
 3 private int itemId;  
 4 private int value; // The value of the rating  
 5   
 6 // Constructor  
 7 public Rating(int userId, int itemId, int value){  
 8 this.itemId=itemId;  
 9 this.userId=userId;  
10 this.value=value;  
11 }  
12 public int getUserId(){  
13 return userId;  
14 }  
15 public int getItemId() {  
16 return itemId;   
17 }  
18 public int getValue(){  
19 return value;   
20 }  
21 public String toString(){  
22 return userId+" "+itemId+" "+value;  
23 }  
24 }  
/////////////////////\*\*class RatingManager\*\*/////////////////

1 import java.io.File;  
 2 import java.io.FileNotFoundException;  
 3 import java.util.NoSuchElementException;  
 4 import java.util.Scanner;  
 5   
 6 public class RatingManager {  
 7   
 8 private int sizeItem;  
 9 private int sizeUser;  
 10 private LinkedList<LinkedList<Rating>> ratingListItem;  
 11 private LinkedList<LinkedList<Rating>> ratingListUser;  
 12 // Constructor  
 13   
 14 public RatingManager() {  
 15 sizeItem = 0;  
 16 sizeUser = 0;  
 17 ratingListItem = new LinkedList<>();  
 18 ratingListUser = new LinkedList<>();  
 19 }  
 20   
 21 // Read ratings from a file and create a RatingManager object that stores these ratings  
 22 public static RatingManager read(String fileName) {  
 23 RatingManager result = new RatingManager();  
 24 Scanner scan=null;  
 25 try {  
 26 scan = new Scanner(new File(fileName));  
 27 while (scan.hasNextLine()) {  
 28   
 29 int userID = scan.nextInt();  
 30 int itemID = scan.nextInt();  
 31 int value = scan.nextInt();  
 32 scan.next();  
 33 Rating rating = new Rating(userID, itemID, value);  
 34 result.addRating(rating);  
 35   
 36 }  
 37 scan.close();  
 38 } catch (FileNotFoundException ex) {  
 39 ex.printStackTrace();  
 40 return null;  
 41 }  
 42 catch (NoSuchElementException ex) {  
 43 // ex.printStackTrace();  
 44 scan.close();  
 45 return result;  
 46 }  
 47 return result;  
 48 }  
 49   
 50 // Add a rating  
 51 public void addRating(Rating rating) {  
 52 addRatingItem(rating);  
 53 addRatingUser(rating);  
 54   
 55 }  
 56   
 57 private void addRatingItem(Rating rating) {  
 58 LinkedList<Rating> list = null;  
 59 if (ratingListItem.empty()) {  
 60 list = new LinkedList<>();  
 61 ratingListItem.insert(list);  
 62 sizeItem++;  
 63 } else {  
 64 ratingListItem.findFirst();  
 65 while (!ratingListItem.last()) {  
 66   
 67 list = ratingListItem.retrieve();  
 68 if (list.retrieve().getItemId() == rating.getItemId()) {  
 69 break;  
 70 } else {  
 71 list = null;  
 72 }  
 73 ratingListItem.findNext();  
 74 }//end while   
 75   
 76 if (list == null) {  
 77 list = ratingListItem.retrieve();  
 78 if (list.retrieve().getItemId() != rating.getItemId()) {  
 79 list = null;  
 80 }  
 81 }  
 82   
 83 }  
 84 if (list == null) {  
 85 list = new LinkedList<>();  
 86 ratingListItem.insert(list);  
 87 sizeItem++;  
 88 }  
 89 list.insert(rating);  
 90   
 91   
 92 }  
 93   
 94 private void addRatingUser(Rating rating) {  
 95 LinkedList<Rating> list = null;  
 96 if (ratingListUser.empty()) {  
 97 list = new LinkedList<>();  
 98 ratingListUser.insert(list);  
 99 sizeUser++;  
100 } else {  
101 ratingListUser.findFirst();  
102 while (!ratingListUser.last()) {  
103   
104 list = ratingListUser.retrieve();  
105 if (list.retrieve().getUserId() == rating.getUserId()) {  
106 break;  
107 } else {  
108 list = null;  
109 }  
110 ratingListUser.findNext();  
111 }  
112 if (list == null) {  
113 list = ratingListUser.retrieve();  
114 if (list.retrieve().getUserId() != rating.getUserId()) {  
115 list = null;  
116 }  
117 }  
118   
119 }  
120 if (list == null) {  
121 list = new LinkedList<>();  
122 ratingListUser.insert(list);  
123 sizeUser++;  
124 }  
125 list.insert(rating);  
126   
127   
128 }  
129   
130 // Return all ratings given by user i. Search should be efficient.   
131 public LinkedList<Rating> getUserRatings(int i) {  
132 LinkedList<Rating> result = null;  
133 ratingListUser.findFirst();  
134 for (int j = 0; j < sizeUser; j++) {  
135 Rating r = ratingListUser.retrieve().retrieve();  
136 if (r.getUserId() == i) {  
137 result = ratingListUser.retrieve();  
138 return result;  
139 }  
140 ratingListUser.findNext();  
141   
142 }  
143 return null;  
144 }  
145   
146 // Return all ratings given to item j. Search should be efficient.  
147 public LinkedList<Rating> getItemRatings(int i) {  
148 LinkedList<Rating> result = null;  
149 ratingListItem.findFirst();  
150 for (int j = 0; j < sizeItem; j++) {  
151 Rating r = ratingListItem.retrieve().retrieve();  
152 if (r.getItemId() == i) {  
153 result = ratingListItem.retrieve();  
154 return result;  
155 }  
156 ratingListItem.findNext();  
157   
158 }  
159 return null;  
160 }  
161   
162 // Return the average rating of item j. If j has no ratings, -1 is returned  
163 public double getAverageItemRating(int i) {  
164 double avg = 0;  
165 double sum = 0;  
166 int count = 0;  
167 LinkedList<Rating> list = this.getItemRatings(i);  
168 if(list==null)  
169 return -1;  
170 list.findFirst();  
171 while (!list.last()) {  
172 Rating r = list.retrieve();  
173   
174 if (r.getItemId() == i) {  
175 count++;  
176 sum += r.getValue();  
177 }  
178 list.findNext();  
179 }  
180 Rating r = list.retrieve();  
181   
182 if (r.getItemId() == i) {  
183 count++;  
184 sum += r.getValue();  
185 }  
186   
187 if (count == 0) {  
188 return -1;  
189 }  
190 avg = sum / count;  
191 return avg;  
192 }  
193   
194 // Return the average rating given by user i. If i has no ratings, -1 is returned  
195 public double getAverageUserRating(int i) {  
196 double avg = 0;  
197 double sum = 0;  
198 int count = 0;  
199 LinkedList<Rating> list = this.getUserRatings(i);  
200 if(list==null)  
201 return -1;  
202 list.findFirst();  
203 while (!list.last()) {  
204 Rating r = list.retrieve();  
205   
206 if (r.getUserId() == i) {  
207 count++;  
208 sum += r.getValue();  
209 }  
210 list.findNext();  
211 }  
212 Rating r = list.retrieve();  
213   
214 if (r.getUserId() == i) {  
215 count++;  
216 sum += r.getValue();  
217 }  
218   
219 if (count == 0) {  
220 return -1;  
221 }  
222 avg = sum / count;  
223 return avg;  
224 }  
225   
226 // Return the list of all items having the highest average rating (for example if the highest average rating is 4.9, the method should return all items with average rating 4.9)  
227 public LinkedList<Integer> getHighestRatedItems() {  
228 // this.ratingListItem.display();  
229 LinkedList<Integer> result = new LinkedList<>();  
230 double[] avg = new double[sizeItem];  
231 double maxAvg = 0;  
232 ratingListItem.findFirst();  
233 for (int i = 0; i < sizeItem; i++) {  
234 Rating r = ratingListItem.retrieve().retrieve();  
235 avg[i] = this.getAverageItemRating(r.getItemId());  
236 if (maxAvg < avg[i]) {  
237 maxAvg = avg[i];  
238 result = new LinkedList<>();  
239 result.insert(r.getItemId());  
240 } else if (maxAvg == avg[i]) {  
241 result.insert(r.getItemId());  
242 }  
243   
244 ratingListItem.findNext();  
245 }//  
246 System.out.println("max avg: "+ maxAvg);  
247 return result;  
248   
249 }  
250   
251 }  
252

# Source Code for phase 2

import java.io.File;  
import java.io.FileNotFoundException;  
import static java.lang.Math.\*;  
import java.util.NoSuchElementException;  
import java.util.Scanner;  
  
public class RatingManager {  
  
 private int sizeItem;  
 private int sizeUser;  
 private BST<BST<Rating>> ratingItem;  
 private BST<BST<Rating>> ratingUser;  
// Constructor  
  
 public RatingManager() {  
 sizeItem = 0;  
 sizeUser = 0;  
 ratingItem = new BST<>();  
 ratingUser = new BST<>();  
 }  
 // Read ratings from a file and create a RatingManager object that stores these ratings. The ratings must be inserted in their order of appearance in the file.  
  
 public static RatingManager read(String fileName) {  
  
 RatingManager result = new RatingManager();  
 Scanner scan = null;  
 try {  
 scan = new Scanner(new File(fileName));  
 while (scan.hasNextLine()) {  
  
 int userID = scan.nextInt();  
 int itemID = scan.nextInt();  
 int value = scan.nextInt();  
 scan.next();  
 Rating rating = new Rating(userID, itemID, value);  
 result.addRating(rating);  
 }  
 scan.close();  
 } catch (FileNotFoundException ex) {  
 ex.printStackTrace();  
 return null;  
 } catch (NoSuchElementException ex) {  
 // ex.printStackTrace();  
 scan.close();  
 return result;  
 }  
 return result;  
 }  
  
 // Add a rating  
 public void addRating(Rating rating) {  
 addRatingItem(rating);  
 addRatingUser(rating);  
  
 }  
  
 private void addRatingItem(Rating rating) {  
 // LinkedList<Rating> list = null;  
 BST<Rating> result = null;  
  
 if (ratingItem.empty()) {  
 result = new BST<>();  
 result.insert(rating.getUserId(), rating);  
 ratingItem.insert(rating.getItemId(), result);  
 sizeItem++;  
 } else if (ratingItem.findKey(rating.getItemId())) {  
  
 result = ratingItem.retrieve();  
 result.insert(rating.getUserId(), rating);  
 } else {  
 result = new BST<>();  
 result.insert(rating.getUserId(), rating);  
 ratingItem.insert(rating.getItemId(), result);  
 sizeItem++;  
 }  
  
 }  
  
 private void addRatingUser(Rating rating) {  
 BST<Rating> result = null;  
  
 if (ratingUser.empty()) {  
 result = new BST<>();  
 result.insert(rating.getItemId(), rating);  
 ratingUser.insert(rating.getUserId(), result);  
 sizeUser++;  
 } else if (ratingUser.findKey(rating.getUserId())) {  
  
 result = ratingUser.retrieve();  
 result.insert(rating.getItemId(), rating);  
 } else {  
 result = new BST<>();  
 result.insert(rating.getItemId(), rating);  
 ratingUser.insert(rating.getUserId(), result);  
 sizeUser++;  
 }  
  
 }  
  
// Return all ratings given by user i.   
 public LinkedList<Rating> getUserRatings(int i) {  
 LinkedList<Rating> result = null;  
 BST<Rating> Items = null;  
 if (ratingUser.findKey(i)) {  
 Items = ratingUser.retrieve();  
 result = Items.getAllData();  
 }  
  
 return result;  
  
 }  
  
// Return all ratings given to item j  
 public LinkedList<Rating> getItemRatings(int j) {  
 LinkedList<Rating> result = null;  
 BST<Rating> users = null;  
 if (ratingItem.findKey(j)) {  
 users = ratingItem.retrieve();  
 result = users.getAllData();  
  
 }  
  
 return result;  
 }  
  
 // Return the list of highest rated items  
 public LinkedList<Integer> getHighestRatedItems() {  
 LinkedList<Integer> result = new LinkedList<>();  
 double[] avg = new double[sizeUser];  
 double maxAvg = 0;  
 Rating r;  
 int i;  
 LinkedList<BST<Rating>> ratingListItem = ratingItem.getAllData();  
 ratingListItem.findFirst();  
 for (i = 0; !ratingListItem.last(); i++) {  
 r = ratingListItem.retrieve().retrieve();  
 avg[i] = this.getAverageItemRating(r.getItemId());  
 if (maxAvg < avg[i]) {  
 maxAvg = avg[i];  
 result = new LinkedList<>();  
 result.insert(r.getItemId());  
 } else if (maxAvg == avg[i]) {  
 result.insert(r.getItemId());  
 }  
 ratingListItem.findNext();  
 }//end for   
 r = ratingListItem.retrieve().retrieve();  
 avg[i] = this.getAverageItemRating(r.getItemId());  
 if (maxAvg < avg[i]) {  
 maxAvg = avg[i];  
 result = new LinkedList<>();  
 result.insert(r.getItemId());  
 } else if (maxAvg == avg[i]) {  
 result.insert(r.getItemId());  
 }  
 return result;  
  
 }  
  
 // Return the average rating of item j. If i has no ratings, -1 is returned  
 public double getAverageItemRating(int j) {  
 LinkedList<Rating> result = this.getItemRatings(j);  
 int count = 0;  
 double avg = 0;  
 double sum = 0;  
 if (result != null) {  
 result.findFirst();  
 while (!result.last()) {  
 sum += result.retrieve().getValue();  
 count++;  
 result.findNext();  
 }  
 sum += result.retrieve().getValue();  
 count++;  
 avg = sum / count;  
 return avg;  
 }  
  
 return -1;  
 }  
  
 // Return the average rating given by user i. If i has no ratings, -1 is returned  
 public double getAverageUserRating(int i) {  
 LinkedList<Rating> result = this.getUserRatings(i);  
 int count = 0;  
 double avg = 0;  
 double sum = 0;  
 if (result != null) {  
 result.findFirst();  
 while (!result.last()) {  
 sum += result.retrieve().getValue();  
 count++;  
 result.findNext();  
 }  
 sum += result.retrieve().getValue();  
 count++;  
 avg = sum / count;  
 return avg;  
 }  
  
 return -1;  
 }  
 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
 // Return the rating of user i for item j. If there is no rating, -1 is returned.  
 public int getRating(int i, int j) {  
 BST<Rating> result = null;  
 if (ratingUser.findKey(i)) {  
 result = ratingUser.retrieve();  
 if (result.findKey(j)) {  
 return result.retrieve().getValue();  
 }  
 }  
 return -1;  
 }  
  
 // Return the number of keys to compare with in order to find the rating of user i for item j.  
 public int nbComp(int i, int j) {  
 BST<Rating> result = null;  
 int count = 0;  
 count = ratingUser.nbComp(i);  
 if (ratingUser.findKey(i)) {  
 result = ratingUser.retrieve();  
 count = count + result.nbComp(j);  
 }  
 return count;  
 }  
  
 // Compute the distance between the two users ui and uj   
 //. If ui and uj have no common item in their ratings, then Double.POSITIVE\_INFINITY is returned.  
 public double getDist(int ui, int uj) {  
 int countI = 0, countJ = 0, countIJ = 0;  
 BST<Rating> BST\_UI = null;  
 BST<Rating> BST\_UJ = null;  
  
 LinkedList<Rating> uiListItems = new LinkedList<>();  
 LinkedList<Rating> ujListItems = new LinkedList<>();  
  
 if (ratingUser.findKey(ui)) {  
 BST\_UI = ratingUser.retrieve();  
 uiListItems = BST\_UI.getAllData();  
 if (ratingUser.findKey(uj)) {  
 BST\_UJ = ratingUser.retrieve();  
 ujListItems = BST\_UJ.getAllData();  
  
 if (uiListItems.empty() || ujListItems.empty()) {  
 return Double.POSITIVE\_INFINITY;  
 }  
  
 LinkedList<Integer> Uij\_ListItem = new LinkedList<>();  
 uiListItems.findFirst();  
 while (!uiListItems.last()) {  
 ujListItems.findFirst();  
 while (!ujListItems.last()) {  
 if (uiListItems.retrieve().getItemId() == ujListItems.retrieve().getItemId()) {  
 Uij\_ListItem.insert(ujListItems.retrieve().getItemId());  
 countIJ++;  
 }  
 ujListItems.findNext();  
 }  
 if (uiListItems.retrieve().getItemId() == ujListItems.retrieve().getItemId()) {  
 Uij\_ListItem.insert(ujListItems.retrieve().getItemId());  
 countIJ++;  
  
 }  
 uiListItems.findNext();  
 }  
 ujListItems.findFirst();  
 while (!ujListItems.last()) {  
 if (uiListItems.retrieve().getItemId() == ujListItems.retrieve().getItemId()) {  
 Uij\_ListItem.insert(ujListItems.retrieve().getItemId());  
 countIJ++;  
 }  
 ujListItems.findNext();  
 }  
 if (uiListItems.retrieve().getItemId() == ujListItems.retrieve().getItemId()) {  
 Uij\_ListItem.insert(ujListItems.retrieve().getItemId());  
 countIJ++;  
  
 }  
  
 if (countIJ != 0) {  
 Uij\_ListItem.findFirst();  
 double distance = 0;  
 double d = 0;  
 int Item;  
 for (int i = 0; i < countIJ; i++) {  
 Item = Uij\_ListItem.retrieve();  
 BST\_UI.findKey(Item);  
 BST\_UJ.findKey(Item);  
 d += Math.pow((BST\_UI.retrieve().getValue() - BST\_UJ.retrieve().getValue()), 2);  
 // distance += Math.sqrt(Math.pow((BST\_UI.retrieve().getValue() - BST\_UJ.retrieve().getValue()), 2)) / countIJ;  
 Uij\_ListItem.findNext();  
 }  
 distance = Math.sqrt(d) / countIJ;  
  
 return distance;  
 }  
 }  
 }  
  
 return Double.POSITIVE\_INFINITY;  
 }  
  
 // Return a list of at most k nearest neighbors to user i from a list of users. User i and users at infinite distance should not be included (the number of users returned can therefore be less than k).  
 public LinkedList<Integer> kNNUsers(int i, LinkedList<Integer> users, int k) {  
 if (users == null || users.empty() || k == 0 || k < 0 || ratingUser == null || !ratingUser.findKey(i)) {  
 return null;  
 }  
 int Qsize = 0;  
 LinkedList l = users;  
  
 l.findFirst();  
 if (!l.empty()) {  
 while (!l.last()) {  
 Qsize++;  
 l.findNext();  
 }  
 Qsize++;  
 }  
  
 LinkedList<Integer> nearest = new LinkedList<>();  
 PQ<Integer> neighbors = new PQ<>(Qsize);  
 double dist, in = Double.POSITIVE\_INFINITY;  
 users.findFirst();  
 while (!users.last()) {  
 dist = getDist(i, users.retrieve());  
 if (dist != in) {  
 neighbors.enqueue(dist, users.retrieve());  
 }  
 users.findNext();  
  
 }  
 dist = getDist(i, users.retrieve());  
 if (dist != in) {  
 neighbors.enqueue(dist, users.retrieve());  
 }  
 if (neighbors.length() == 0) {  
 return null;  
 }  
 if (neighbors.length() > k) {  
 while (neighbors.length() != k) {  
 neighbors.serve();  
 }  
 }  
 while (neighbors.length() != 0) {  
 nearest.insert(neighbors.serve().data);  
 }  
  
 return nearest;  
  
 // return null;  
 }  
  
// Return the average rating given to item j by a list of users. If the list users is empty or non of the users it contains rated item j, then the global average rating of item j (as computed by getAverageItemRating(j)) is returned.  
 public double getAverageRating(int j, LinkedList<Integer> users) {  
 LinkedList<Rating> result = this.getItemRatings(j);  
 if (users == null || result == null) {  
 return getAverageItemRating(j);  
 }  
  
 double avg = 0;  
 int count = 0;  
 boolean flag = false;  
 if (!result.empty() && !users.empty()) {  
 users.findFirst();  
 while (!users.last()) {  
 result.findFirst();  
 while (!result.last()) {  
 if (result.retrieve().getUserId() == users.retrieve()) {  
 avg += result.retrieve().getValue();  
 count++;  
 flag = true;  
 }  
  
 result.findNext();  
 }  
  
 if (result.retrieve().getUserId() == users.retrieve()) {  
 avg += result.retrieve().getValue();  
 count++;  
 flag = true;  
 }  
 if (flag == false) {  
 return getAverageItemRating(j);  
 }  
 users.findNext();  
 flag = false;  
  
 }  
 result.findFirst();  
 while (!result.last()) {  
 if (result.retrieve().getUserId() == users.retrieve()) {  
 avg += result.retrieve().getValue();  
 count++;  
 flag = true;  
  
 }  
 result.findNext();  
 }  
 if (result.retrieve().getUserId() == users.retrieve()) {  
 avg += result.retrieve().getValue();  
 count++;  
 flag = true;  
  
 }  
 if (flag == false) {  
 return getAverageItemRating(j);  
 }  
 if (count != 0) {  
 return avg / count;  
 }  
 return getAverageItemRating(j);  
 }  
 return getAverageItemRating(j);  
 }  
  
 // Return an estimation of the rating given by user i for item j using k nearest neighbor users.  
 public double getEstimatedRating(int i, int j, int k) {  
 int r = getRating(i, j);  
  
 if (r != -1) {  
 return r;  
 }  
 LinkedList<Rating> ratings = getItemRatings(j);  
 LinkedList<Integer> users = new LinkedList<Integer>();  
 if ((ratings  
 != null) && !ratings.empty()) {  
 ratings.findFirst();  
 while (!ratings.last()) {  
 users.insert(ratings.retrieve().getUserId());  
 ratings.findNext();  
 }  
 users.insert(ratings.retrieve().getUserId());  
 }  
 LinkedList<Integer> knn = kNNUsers(i, users, k);  
  
 return getAverageRating(j, knn);  
 }  
  
}

public class Rating {  
 private int userId;  
 private int itemId;  
 private int value; // The value of the rating  
   
 // Constructor  
 public Rating(int userId, int itemId, int value){  
 this.itemId=itemId;  
 this.userId=userId;  
 this.value=value;  
 }  
 public int getUserId(){  
 return userId;  
 }  
 public int getItemId() {  
 return itemId;   
 }  
 public int getValue(){  
 return value;   
 }  
 public String toString(){  
 return userId+" "+itemId+" "+value;  
 }  
   
}

public class PQElem<T> {  
 public double priority;  
 public T data;  
  
 public PQElem(double \_priority, T \_data) {  
 priority = \_priority;  
 data = \_data;  
 }  
}

class HeapElem<T> {  
 public double key;  
 public T data;  
  
 public HeapElem(double \_key, T \_data) {  
 key = \_key;  
 data = \_data;  
 }  
}  
  
class Heap<T> {  
  
 private int maxSize;  
 private int size;  
 private double[] keys;  
 private T[] data;  
  
 public Heap(int \_maxSize) {  
  
 maxSize = \_maxSize;  
 size = 0;  
 keys = new double[maxSize + 1];  
 data = (T[]) new Object[maxSize + 1];  
 }  
  
 public int size() {  
 return size;  
 }  
  
 public boolean full() {  
 return size == maxSize;  
 }  
  
 private void siftUp() {  
 int k = size;  
 while ((k > 1) && (keys[k] > keys[k / 2])) {  
  
 // System.out.println(k);  
  
 double tmpKey = keys[k];  
 keys[k] = keys[k / 2];  
 keys[k / 2] = tmpKey;  
  
 T tmpData = data[k];  
 data[k] = data[k / 2];  
 data[k / 2] = tmpData;  
  
 k = k / 2;  
 }  
 }  
  
 public void insert(double key, T val) {  
  
 size++;  
 keys[size] = key;  
 data[size] = val;  
 siftUp();  
 }  
  
 private void siftDown(int i) {  
 int k = i;  
 while (((2 \* k <= size) && (keys[k] < keys[2 \* k])) || ((2 \* k + 1 <= size) && (keys[k] < keys[2 \* k + 1]))) {  
 if ((2 \* k + 1 <= size) && (keys[2 \* k + 1] > keys[2 \* k])) {  
  
 double tmpKey = keys[k];  
 keys[k] = keys[2 \* k + 1];  
 keys[2 \* k + 1] = tmpKey;  
  
 T tmpData = data[k];  
 data[k] = data[2 \* k + 1];  
 data[2 \* k + 1] = tmpData;  
  
 k = 2 \* k + 1;  
 } else {  
  
 double tmpKey = keys[k];  
 keys[k] = keys[2 \* k];  
 keys[2 \* k] = tmpKey;  
  
 T tmpData = data[k];  
 data[k] = data[2 \* k];  
 data[2 \* k] = tmpData;  
  
 k = 2 \* k;  
 }  
 }  
 }  
  
 public HeapElem<T> removeRoot() {  
  
 HeapElem<T> tmp = new HeapElem<T>(keys[1], data[1]);  
 keys[1] = keys[size];  
 data[1] = data[size];  
 size--;  
 siftDown(1);  
 return tmp;  
 }  
}  
  
public class PQ<T> {  
  
 private Heap<T> heap;  
  
 public PQ(int \_maxSize) {  
 heap = new Heap<T>(\_maxSize);  
 }  
  
 public int length() {  
 return heap.size();  
 }  
  
 public boolean full() {  
 return heap.full();  
 }  
  
 public void enqueue(double pr, T val) {  
 heap.insert(pr, val);  
 }  
  
 public PQElem<T> serve() {  
 HeapElem<T> heapEl = heap.removeRoot();  
 PQElem<T> pqEl = new PQElem<T>(heapEl.key, heapEl.data);  
 return pqEl;  
 }  
}

class Node<T> {  
  
 public T data;  
 public Node<T> next;  
  
 public Node(T val) {  
 data = val;  
 next = null;  
 }  
}  
  
public class LinkedList<T> {  
  
 private Node<T> head;  
 private Node<T> current;  
  
 public LinkedList() {  
 head = current = null;  
 }  
  
 public boolean empty() {  
 return head == null;  
 }  
  
 public boolean last() {  
 return current.next == null;  
 }  
  
 public boolean full() {  
 return false;  
 }  
  
 public void findFirst() {  
 current = head;  
 }  
  
 public void findNext() {  
 current = current.next;  
 }  
  
 public T retrieve() {  
 return current.data;  
 }  
  
 public void update(T val) {  
 current.data = val;  
 }  
  
 public void insert(T val) {  
 Node<T> tmp;  
 if (empty()) {  
 current = head = new Node<T>(val);  
 } else {  
 tmp = current.next;  
 current.next = new Node<T>(val);  
 current = current.next;  
 current.next = tmp;  
 }  
 }  
  
 public void remove() {  
 if (current == head) {  
 head = head.next;  
 } else {  
 Node<T> tmp = head;  
 while (tmp.next != current) {  
 tmp = tmp.next;  
 }  
 tmp.next = current.next;  
 }  
 if (current.next == null) {  
 current = head;  
 } else {  
 current = current.next;  
 }  
 }  
  
 public void display() {  
 Node<T> tmp = head;  
 while (tmp != null) {  
 System.out.print(tmp.data);  
 tmp = tmp.next;  
   
 }  
 System.out.println();  
  
 }  
   
}

class BSTNode<T> {  
  
 public int key;  
 public T data;  
 public BSTNode<T> left, right;  
  
 public BSTNode(int key, T data) {  
 this.key = key;  
 this.data = data;  
 left = right = null;  
 }  
}  
  
public class BST<T> {  
  
 private BSTNode<T> root, current;  
  
 public BST() {  
 current = root = null;  
 }  
  
 public void clear() {  
 current = root = null;  
 }  
  
 public boolean empty() {  
 return root == null;  
 }  
  
 public boolean full() {  
 return false;  
 }  
  
 public T retrieve() {  
 return current.data;  
 }  
  
 public boolean findKey(int k) {  
  
 BSTNode<T> p = root;  
 while (p != null) {  
 current = p;  
 if (k == p.key) {  
 return true;  
 } else if (k < p.key) {  
 p = p.left;  
 } else {  
 p = p.right;  
 }  
 }  
 return false;  
 }  
  
 public int nbComp(int k) { // ظ‡ط°ظ‡ ط§ظ„ظ…ط«ط¯ طھط¹ط¯ ط§ظ„ط¹ظ†ط§طµط±طں  
  
 int nb = 0;  
 BSTNode<T> p = root;  
 while (p != null) {  
 nb++;  
 if (k == p.key) {  
 break;  
 } else if (k < p.key) {  
 p = p.left;  
 } else {  
 p = p.right;  
 }  
 }  
 return nb;  
 }  
  
 public boolean insert(int k, T val) {  
 if (root == null) {  
 current = root = new BSTNode<T>(k, val);  
 return true;  
 }  
  
 BSTNode<T> p = current;  
 if (findKey(k)) {  
 current = p;  
 return false;  
 }  
  
 BSTNode<T> tmp = new BSTNode<T>(k, val);  
 if (k < current.key) {  
 current.left = tmp;  
 } else {  
 current.right = tmp;  
 }  
 current = tmp;  
 return true;  
 }  
  
 public boolean removeKey(int k) {  
  
 // Search for k  
 int k1 = k;  
 BSTNode<T> p = root;  
 BSTNode<T> q = null; // Parent of p  
 while (p != null) {  
  
 if (k1 < p.key) {  
 q = p;  
 p = p.left;  
 } else if (k1 > p.key) {  
 q = p;  
 p = p.right;  
 } else { // Found the key  
  
 // Check the three cases  
 if ((p.left != null) && (p.right != null)) { // Case 3: two  
 // children  
  
 // Search for the min in the right subtree  
 BSTNode<T> min = p.right;  
 q = p;  
 while (min.left != null) {  
 q = min;  
 min = min.left;  
 }  
 p.key = min.key;  
 p.data = min.data;  
 k1 = min.key;  
 p = min;  
 // Now fall back to either case 1 or 2  
 }  
  
 // The subtree rooted at p will change here  
 if (p.left != null) { // One child  
 p = p.left;  
 } else { // One or no children  
 p = p.right;  
 }  
  
 if (q == null) { // No parent for p, root must change  
 root = p;  
 } else if (k1 < q.key) {  
 q.left = p;  
 } else {  
 q.right = p;  
 }  
  
 current = root;  
 return true;  
 }  
 }  
  
 return false; // Not found  
 }  
  
 // Return a list containing all data (result is in increasing order  
 // of the key)  
 public LinkedList<T> getAllData() {  
 LinkedList<T> l = new LinkedList<T>();  
 recCollectInorderData(root, l);  
 return l;  
 }  
  
 private void recCollectInorderData(BSTNode<T> t, LinkedList<T> l) {  
 if (t == null) {  
 return;  
 }  
 recCollectInorderData(t.left, l);  
 l.insert(t.data);  
 recCollectInorderData(t.right, l);  
 }  
  
 public void print() {  
 recprint(root);  
 }//end  
  
 private void recprint(BSTNode<T> p) {  
 if (p == null) {  
 return;  
 }  
  
 recprint(p.right);  
 System.out.println(p.data.toString());  
 recprint(p.left);  
 }//end  
}

# Conclusion

This program adds ratings and each rating have an Item ID,User ID,and the value of the rating.

Also this program calculate the average ratings for each items and users ,also calculate the highest average ratings and return all items that have the highest average ,and calculate the distance ,and find the nearest neighbors to a user .