**Problem Set #1**

1) Exercise 1

1. The basic operations performed by the algorithm that counts toward the running time is: printing a star, printing a space, and printing a newline.
2. The following algorithm will print 15 stars, 10 blank spaces, and 5 newlines.
3. As a function of n, there are n\*(n+1)/2 operations to print stars, n\*(n-1)/2 operations to print spaces, and n operations to print new lines.
4. The Big O runtime will be O(n^2).

2) Exercise 2

1. The basic operations performed by the algorithm that counts toward the running time is: adding the scores and dividing them to find the average class score.
2. The worst-case running time as a total count of these basic operations will be the following: all scores for each student is added c times. Because there are r student cumulative scores, there are (c-1)\*r additions for the totals. Then, these totals are added up (r-1) times to find the total score in the class. The final average will be the result of dividing exactly once by the number of students.
3. The Big O running time is O(rc).
4. This algorithm is linear. (because the runtime is proportionally linear to the input scores)

3) Exercise 3

1. By unshuffling a deck, you would need to create 4 new arrays with length 13 each, representing the suit and rank of each card. Next, you would have to iterate through the cards and place them in their respective spot within these arrays. This always has a Big O runtime of O(1) as the function’s inputs never change, which should known as the worst-case runtime.
2. The basic operations are: finding the card in the deck array and putting into its place in the arrays.
3. Since input sizes never change, the worst-case runtime is the same as the average runtime: O(1).

4) Exercise 4

1. It should be an algorithm that involves no sorting that would find common songs between the lists would compare every song in the first list to every song in the second.
   1. In the worst case, there would be no matching songs from either lists so Big O runtime would be O(n\*m). The basic operations that contributes to the runtime are all the comparisons (each song from the first playlist with the second) that would have to be done.
   2. In the best case, the songs from each list would match immediately that causes a runtime of O(min(m,n)^2). The bookkeeping that would be necessary is storing each of these songs that are present in both lists.
2. Using mergesort to sort each array and then combine the two arrays into a singular array.
   1. The worst case is O(mlog m + nlog n) which accounts for not finding a single matched pair from the entire array.
   2. The best case is O(mlog m + nlog n) which accounts for the runtime and finding only the matched pairs from the array.

5) Exercise 5

1. My algorithm creates a new array that has the cumulative distances for each exit from the starting point. To do this, I would create a new array. For each exit, I would add its own distance from the previous exit and the accumulated distance from that previous exit to the beginning. For this part, the runtime becomes O(n).

After that, when the algorithm is called to find the distance between two exits, it’s necessary to subtract the associated respective cumulative distances between the two input exit points to find the distance between those exits. This would result the runtime of O(1) since you are only calculating the difference between two exits.

**Problem Set #2**

1)

public static IntNode addBefore(IntNode front, int target, int newItem) {

//creates current & previous nodes

IntNode current = front;

IntNode prev = null; //target node is first item in linked list

//when target isn’t reached, moves on to next node in linked list

while(current != null && current.data !=target) {

prev = current;

current = current.next;

}

//target hasn’t been found (so it returns original front)

if (current == null) {

return front;

}

//creates new node inserted in linked list before target

IntNode temp = new IntNode(newItem, current);

//initialized “newItem” data that points current

if (prev == null) {

return temp; //target node is first item in linked list & temp becomes new front

}

prev.next = temp; //inserts before target

return front; //front unchanged

}

2)

public static IntNode addBeforeLast(IntNode front, int item) {

//this class adds an integer before the last integer (aka second to last integer)

//List is empty, so there is no last node to add before

if (front == null)

return front;

IntNode current = front;

IntNode prev = null;

//when target isn’t reached, moves on to next node in linked list

while(current.next != null) {

prev = current;

current = current.next;

}

//list has only one node, creating a new node as the new front (head)

if (prev == null)

return new IntNode (item,current);

//creates new node (2nd to last) before previous (which is last integer)

prev.next = new IntNode(item,current);

return front; //returns original front

}

3)

public static int numberOfOccurrences(StringNode front, String target) {

//method counts how many times current’s Strings equals to target’s Strings

int count = 0;

StringNode current = front;

while(current != null) {

if (current.data.equals(target)) {

count++;

}

current = current.next;

}

return count; //returns & shows how many times current’s String equals to target

}

4)

public static void deleteEveryOther(IntNode front) {

if (front == null)

return;

int dice = 0;

IntNode current = front;

IntNode prev = null;

while(current != null) {

if (dice == 0) {

prev = current;

current = current.next;

dice = 1;

}

else if (dice == 1) {

prev.next = current.next;

current = prev.next;

dice = 0;

}

}

}

5)

public static StringNode deleteAllOccurrences(StringNode front, String target) {

//method removes multiple occurrences of strings

if (front == null)

return null;

while (front != null && front.data.equals(target)) {

front = front.next;

}

StringNode current = front;

StringNode prev = null;

while (current != null) {

if (current.data.equals(target)) {

prev.next = current.next;

current = prev.next; //skips references in linked list when current = target

}

else {

prev = current;

current = current.next; //moves onto next node in linked list

}

}

return front;

}

6)

public static IntNode commonElements(IntNode frontL1, IntNode frontL2) {

IntNode currentL1 = frontL1;

IntNode currentL2 = frontL2;

IntNode currentL3 = null;

IntNode bookmark = null;

while(currentL1 != null) {

while(currentL2 != null) {

if (currentL1.data == currentL2.data) { //find common elements from 2 linked lists

if (currentL3 == null) {

currentL3 = new IntNode(currentL1.data, null); //list with common element data

bookmark = currentL3;

} else {

bookmark.next = new IntNode(currentL1.data, null);

bookmark = bookmark.next;

}

}

currentL2 = currentL2.next;

}

currentL2 = frontL2;

currentL1 = currentL1.next; //compares one element from 1st list with other list elements

}

return currentL3; //returns new list with the common elements

}

**Problem Set #3**

1. public boolean delete(String target) {

Node prev = rear;

Node current = rear.next;

while (rear != current) {

if (current.data.equals(target)) {

prev.next = current.next; //deletes a node

return true; //deletion successful

}

prev = current;

current = current.next; //moves on to next nodes

}

if (rear == current) { //last node is being checked

if (current.data.equals(target)) {

rear = prev;

prev.next = current.next; //deletes last node

return true; //deletion successful

}

}

return false; //no target was found

}

2. public boolean addAfter(String newItem, String afterItem) {

Node current = rear.next;

while (rear != current) { //not at last node

if (current.data.equals( afterItem)) {

Node newNode = new Node(newItem, current.next); //creates new node with newItem

current.next = newNode; //inserts new Node after current node

if (current == rear) { //at last node

this.rear = newNode; //new node becomes last node

}

return true; //insertion successful

}

current = current.next; //moves on to next node

}

return false; //no targeted item to add after from list (insertion unsuccessful)

}

3. public static DLLNode moveToFront(DLLNode front, DLLNode target) {

if (target == null)

return front; //no nodes in list or when target isn’t found

DLLNode current = target;

DLLNode prev = null;

while (current != null) { //loops until last node is reached

prev = current;

current = current.next;

} //prev will become last node after loop

prev.next = front; //last (prev) node will point to first node (last node become first node)

front.prev = prev; //first (prev) node will point back to last node (due to DLL)

return target; //new node is front of the list

}

4. public static DLLNode reverse(DLLNode front) {

DLLNode temp = null;

DLLNode current = front;

DLLNode check= null;

while (current != null) {

check = current;

temp = current.next;

current.next = current.prev; //current.setNext(previous)

current.prev = temp; //current.setPrevious(next)

current = current.prev; //current.getPrevious()

}

return check;

}

5. public static Node deleteAll(Node front, String target) {

if (front == null) return null; //if it’s empty list

if (target == null) return front; //nothing to delete

if (front.data == target) {

return (deleteAll(front.next,target)); //skips current node (not saved)

}

else {

return new Node(front.data, deleteAll(front.next,target));

} //else: creates new nodes that saves data not same value as target

}

6. public static Node merge(Node frontL1, Node frontL2) {

if(frontL1== null) return frontL2; //if it’s empty list, moves on to merge 2nd list

if(frontL2 == null) return frontL1; //if it’s empty list, moves on to merge 1st list

if(frontL1.data<frontL2.data) return new Node(frontL1.data,merge(frontL1.next,frontL2));

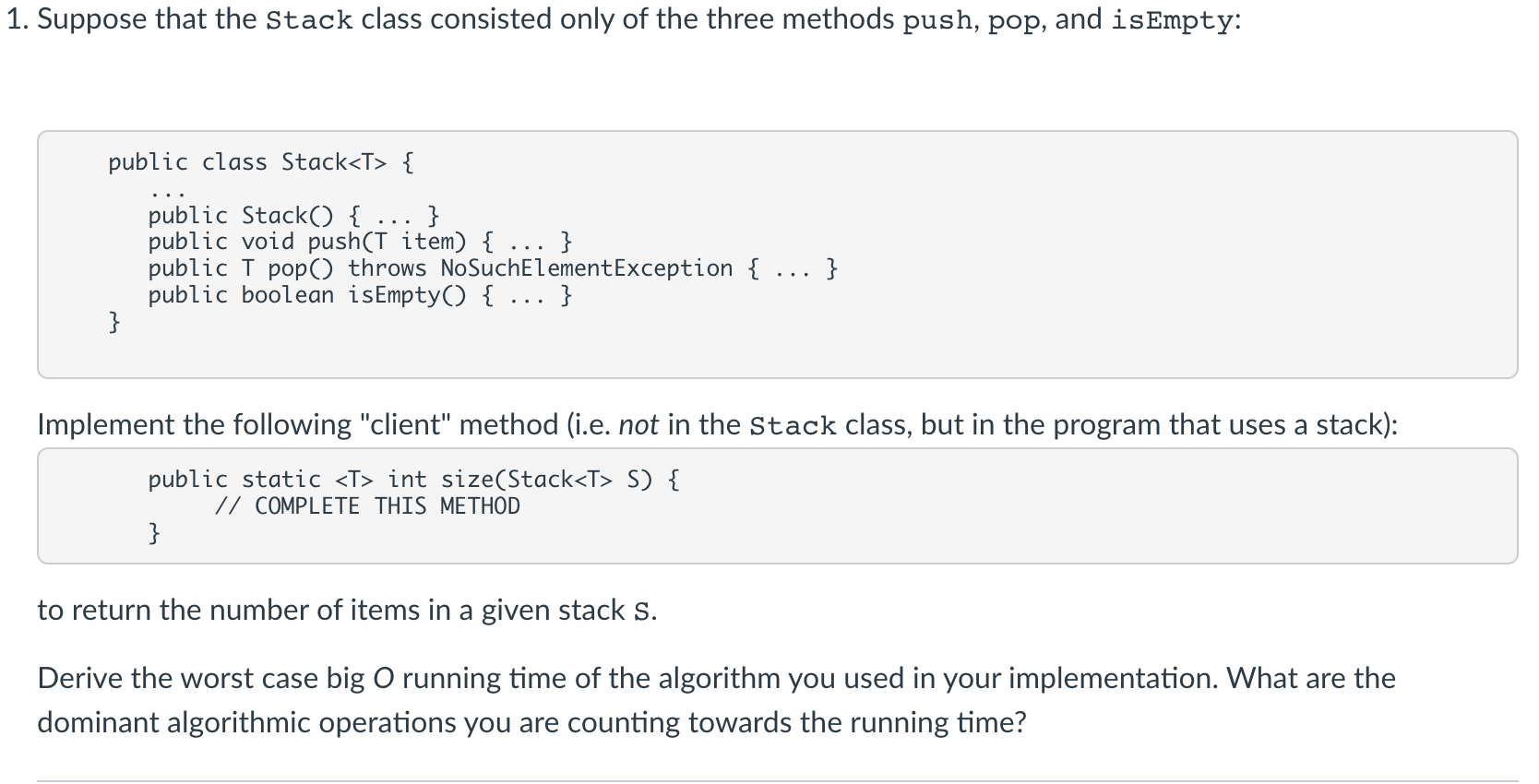
else if(frontL1.data > frontL2.data) return new Node(frontL2.data,merge(frontL1,frontL2.next));

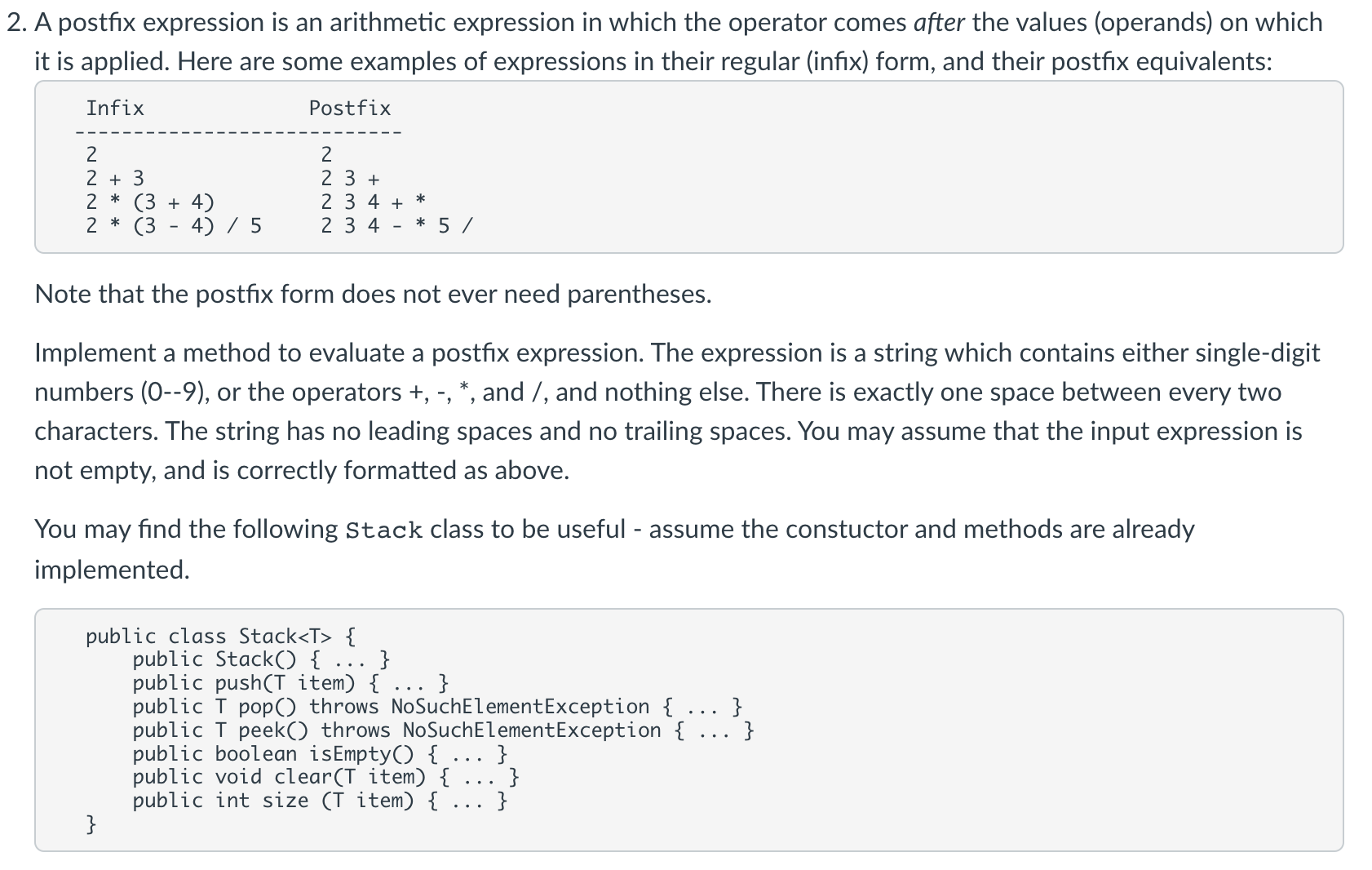
else if(frontL1.data == frontL2.data) return new Node(frontL2.data,merge(frontL1.next,frontL2.next));

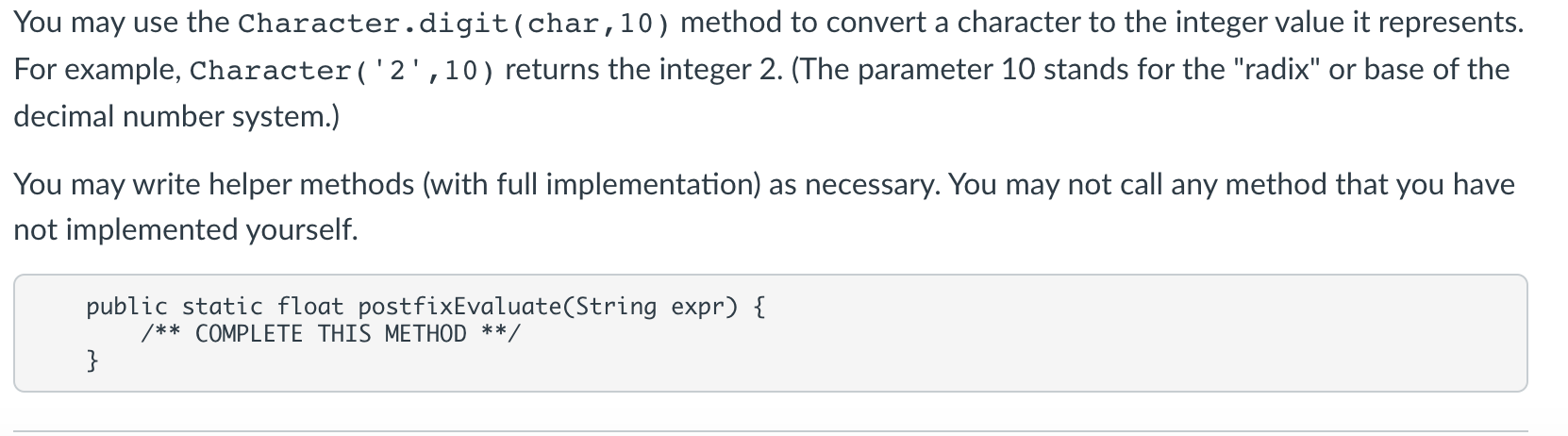
return null;

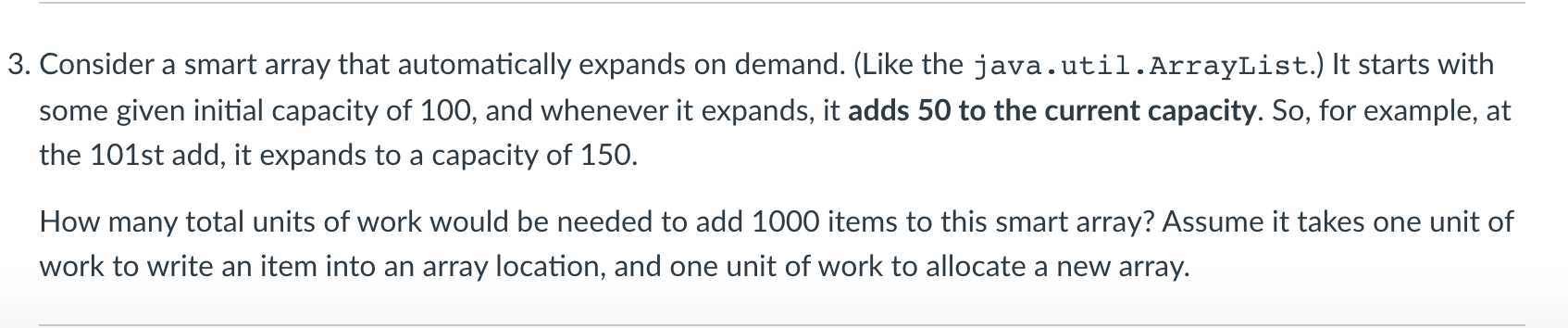
}

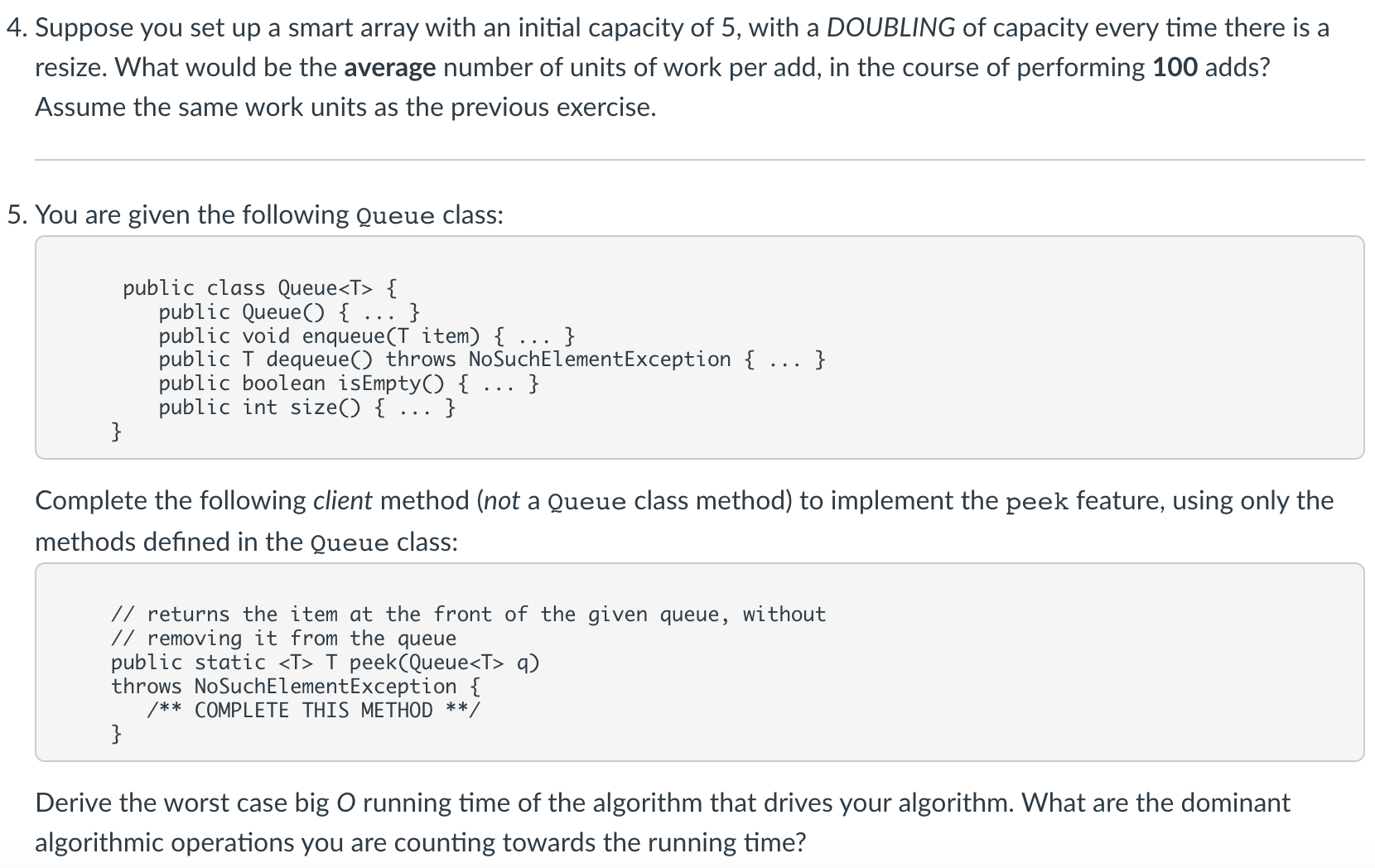
**Problem Set #4**

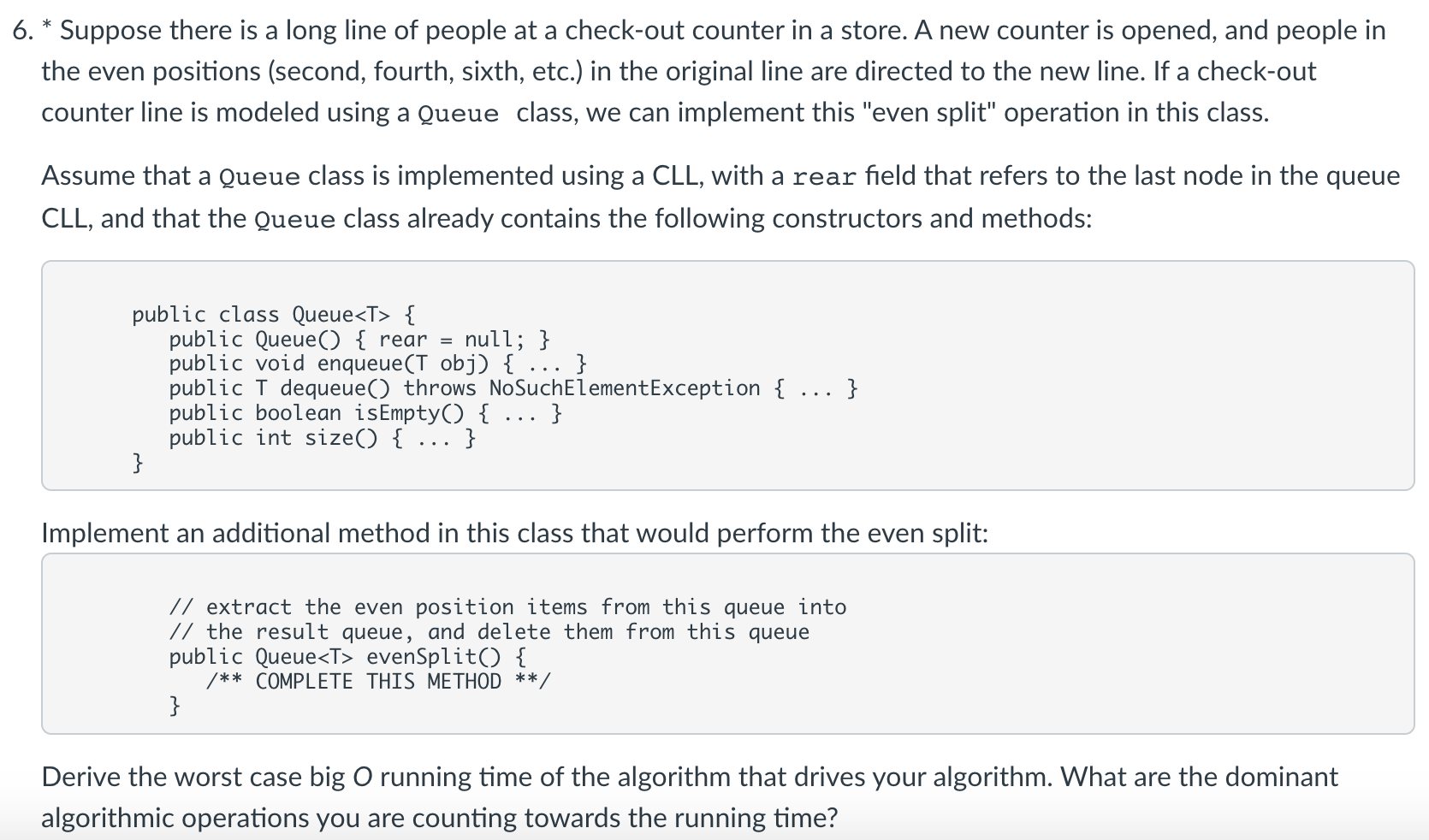
****

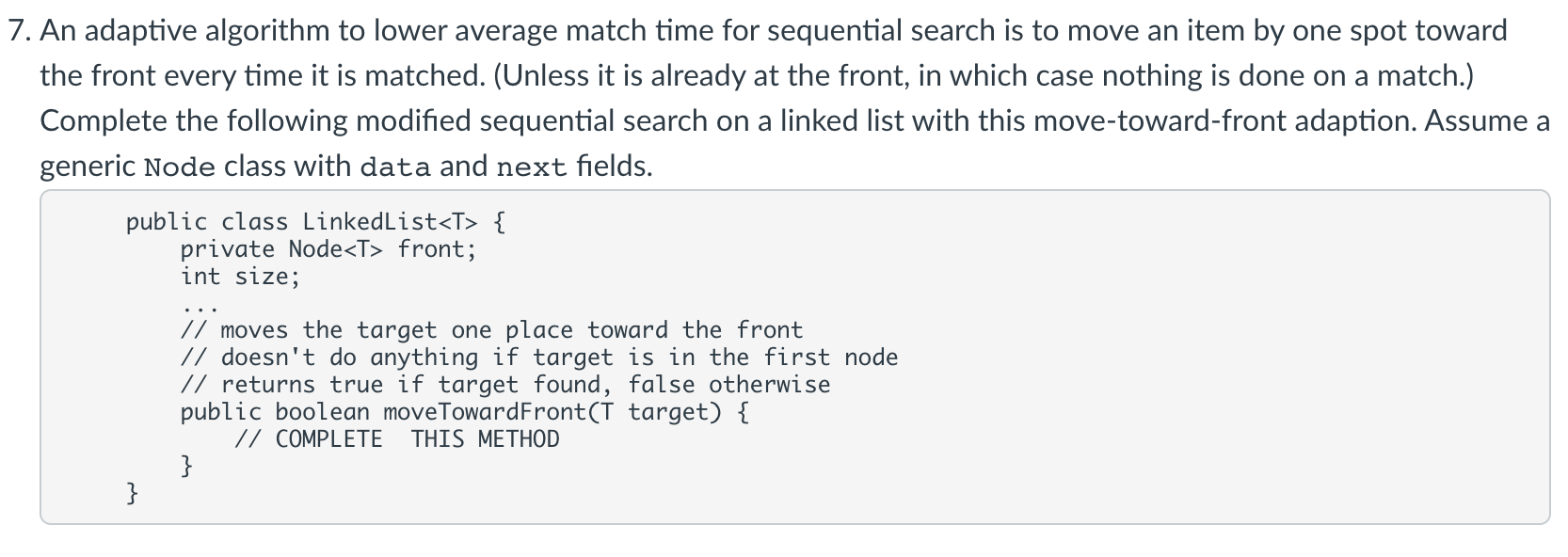
****

****

****

****

****

****

1)

public static <T> int size(Stack<T> S) {

Stack<T> temp = new Stack<T>();

int count = 0;

while (!S.isEmpty()) {

temp.push(S.pop());

count++;

}

while (!temp.isEmpty()) {

S.push(temp.pop());

}

return count;

}

The dominant algorithmic operations for the stack are the push and pop. Elements of the stack are popped and pushed twice each for 4n total pushes and pops, meaning the Big O is O(n).

2.

public static float postfixEvaluate(String expr) {

Stack<Float> S = new Stack<Float>();

char chac=0;

for (int i=0; i < expr.length(); i++) {

chac = expr.charAt(i);

if (chac == '+' || chac == '-' || chac == '\*' || chac == '/') {

float second = S.pop();

float first = S.pop();

switch (chac) {

case '+': S.push(first + second);

case '-': S.push(first - second);

case '\*': S.push(first \* second);

case '/': S.push(first / second);

}

continue;

} else if (chac == ' ') {

continue;

}

S.push((float)Character.digit(chac,10));

}

return S.pop();

}

3.

Achieving 1000 elements would need 18 expansions of 50 units. 100(initial)+18\*50=1000

Calculating total units of work: 100(initial) + (1 (memory allocation) + 100 (writing old elements into new elements) + 50 (new units) + (1 + 150 + 50) ... (1+950+50).

4.

The array needs to be expanded 5 times in order to achieve the desired length. For every expansion, the current length is copied so we need 5, 10, 20, 40, and 80 expansions respectively.

To input 100 new values, it will take 100 units of work.

5 (initial) + 155 (added) + 100 (values written in) = 260. 260/100(total units) = 2.6 average.

5.

public static <T> T peek(Queue<T> q) {

if (q.isEmpty()) {

throw new NoSuchElementException();

}

T first = q.dequeue();

Queue<T> q2 = new Queue<T>();

q2.enqueue(first);

while (!q.isEmpty()) {

q2.enqueue(q.dequeue());

}

while (!q2.isEmpty()) {

q.enqueue(q2.dequeue());

}

return first;

}

The dominant operations are enqueue and dequeue. Everything in the queue is enqueued and dequeued twice - there are 4n operations, for a Big O time of O(n) time.

6. public Queue<T> evenSplit() {

Queue<T> evenQueue = new Queue<T>();

int qSize = size();

for (int pos=0; pos < qSize; pos += 2) {

enqueue(dequeue());

evenQueue.enqueue(dequeue());

qSize-=2;

}

if ((qSize % 2) == 1) {

enqueue(dequeue());

}

return evenQueue;

}

Enqueue and dequeue are the dominant operations. In each loop there is an enqueue and dequeue, for total 2n total operations. The Big O runtime is O(n).

7.

public boolean moveTowardFront(T target) {

Node<T> current = this.front;

Node<T> prev = null;

while (current != null) {

if (current.data.equals(target)) {

break;

} else {

prev = current;

current = current.next;

}

}

if (current == null) {

return false;

}

if (prev == null) {

T temp = prev.data;

prev.data = current.data;

current.data = temp;

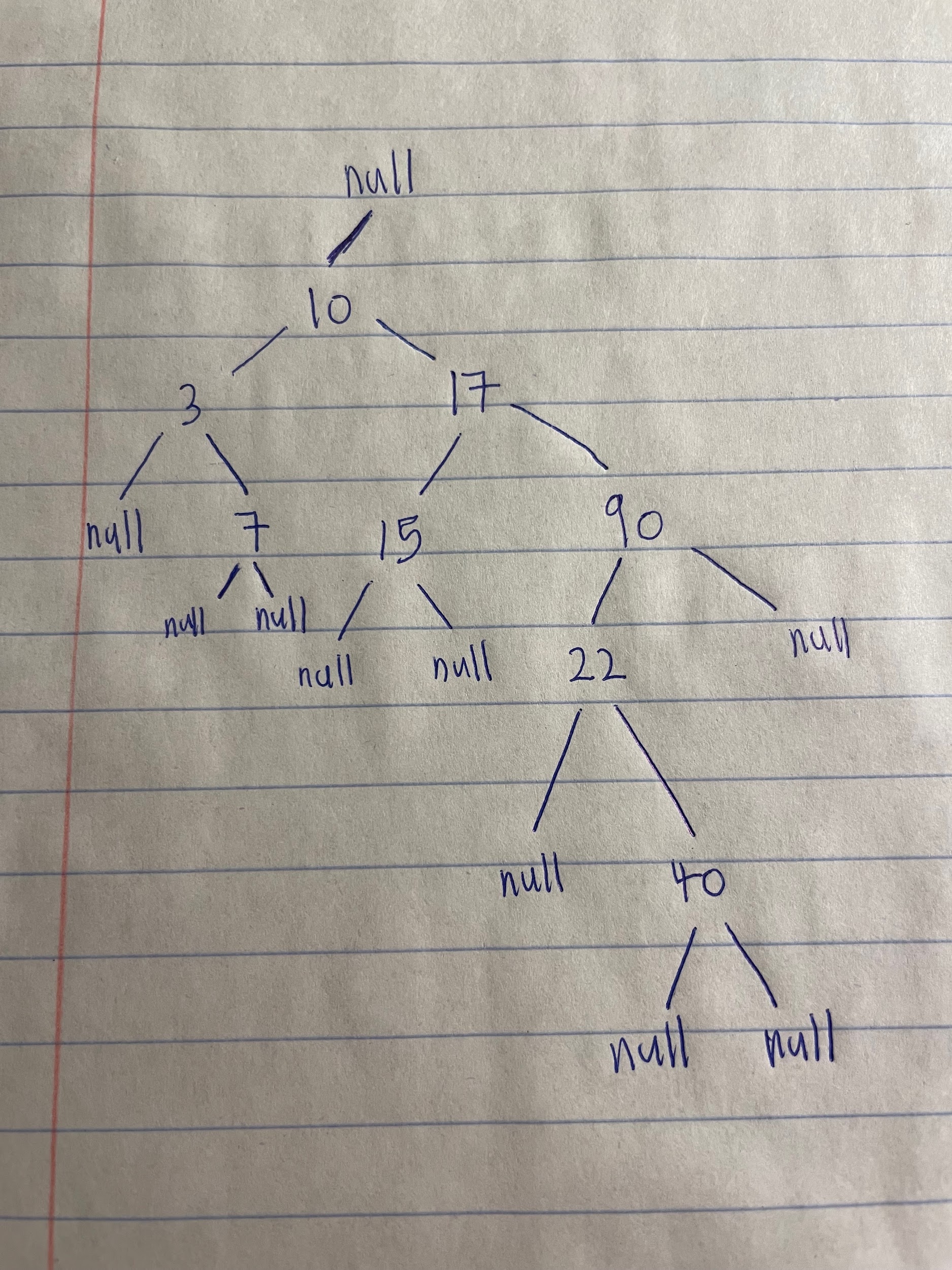
return true;

}

}

**Problem Set #5**

1)



| **Number Inserted** | **# of Comparisons** |
| --- | --- |
| 10 | 1 (branch out from null) |
| 17 | 2 |
| 3 | 2 |
| 90 | 4 |
| 22 | 6 |
| 7 | 4 |
| 40 | 8 |
| 15 | 4 |

**31 comparisons total**

2)

* For the worst case successful search, 9 comparisons is done to get to the number that is deepest in the binary tree, which is **“40.”**
* For the worst case unsuccessful (failed) search, 10 comparisons should be made and **find a number slightly above or below “40”** that will ultimately end at a null node.
* To find the average successful search, find the mean of the values of comparisons for each individual value in the tree. 1+3+3+5+5+5+7+9=38 -> 38/8 = **4.75** comparisons in an average successful search.
* To find 17:
  + Finding the top of the right subtree (“17”) requires **3** units of work.
  + Moving the pointers (previous and current) down through the tree that puts:
    - Previous and current to initially(requires **2** units of work)
    - Move previous and current to 10 & 17 (requires **2** units of work)
* To locate y:
  + Locate smallest value of “22” of right subtree from number “17”:
    - Previous and current moved from 17 to 90, then 90 to 22 (requires **4** units of work)
* Replace 17 with 22 value:
  + No comparison or pointer assignment made (0 unit of work)
* Deletes y of “22”:
  + Pointer assignment made from 90 to 40 (**1** unit of work)

Total units of work:

2 + 2 + 4 + 1 = 9

3 + 9 = **12 units of work**

3)

public void insert(T target) {

if (this.root == null) {

this.root = new BSTNode<T>(target);

} else {

recursiveInsert(this.root, target);

}

}

public void recursiveInsert(BSTNode<T> root, T target) {

int c = 0;

c = target.compareTo(root.data);

if (c == 0) {

throw new IllegalArgumentException("Duplicate key");

} else if (c < 0) {

if (root.left == null) {

BSTNode<T> newNode = new BSTNode<T>(target);

root.left = newNode;

size++;

}

else {

recursiveInsert(root.left,target);

}

}

else {

if (root.right == null) {

BSTNode<T> newNode = new BSTNode<T>(target);

root.right = newNode;

size++;

}

else {

recursiveInsert(root.right,target);

}

}

}

4)

public static <T extends Comparable<T>> void keysInRange(BSTNode<T> root, T min, T max, ArrayList<T> result) {

if (root!=null) {

if (root.data.compareTo(max)<=0 && root.data.compareTo(min)>=0) {

result.add(root.data);

}

keysInRange(root.left,min,max,result);

keysInRange(root.right,min,max,result);

}

}

5)

public static <T extends Comparable<T>> void reverseKeys(BSTNode<T> root) {

if (root == null)

return;

BSTNode<T> temp = root.left;

root.left = root.right;

root.right = temp;

reverseKeys(root.right);

reverseKeys(root.left);

}

6)

public static <T extends Comparable<T>> T kthLargest(BSTNode<T> root, int k) {

if (root.rightSize + 1 == k) {

return root.data;

} else if (root.rightSize + 1 < k) {

return kthLargest(root.left, k - (root.rightSize + 1));

} else {

return kthLargest(root.right, k);

}

}

**Problem Set #6**

1.

* (h+1) nodes
* (2h+1) nodes
* (2^h) nodes

2.

public static <T> boolean isomorphic(BTNode<T> T1, BTNode<T> T2) {

if ((T1!=null && T2==null) || (T1==null && T2!=null) {

return false;

}

if (T1==null && T2==null) {

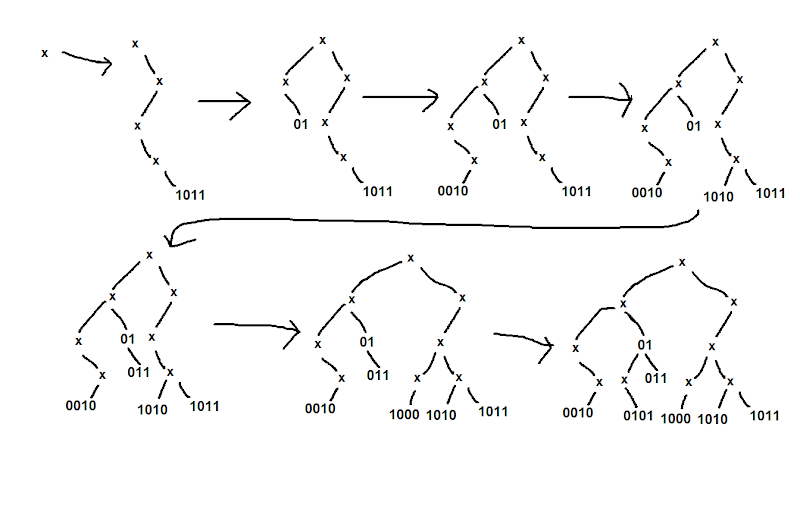
return true;

}

return (isomorphic(T1.left,T2.left) && isomorphic(T1.right, T2.right));

}

3.



* 1011: 4 units of time

01: 2 units of time

0010: 4 units of time

1010: 4 units of time

011: 3 units of time

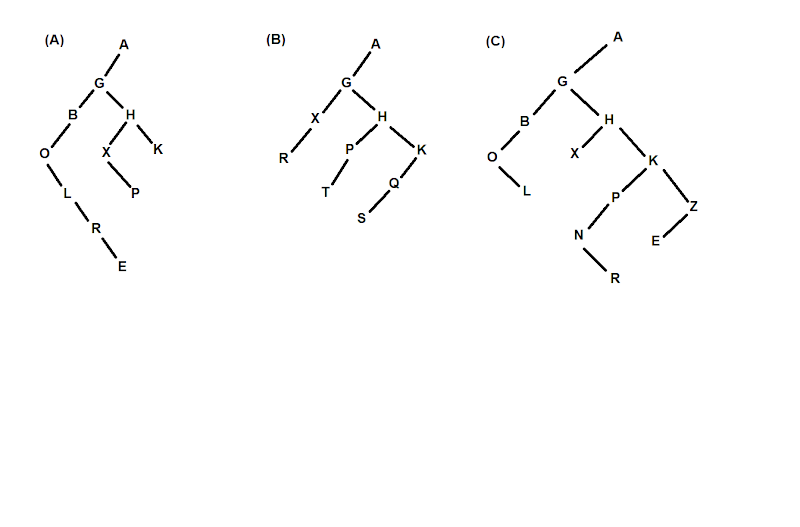
1000: 4 units of time

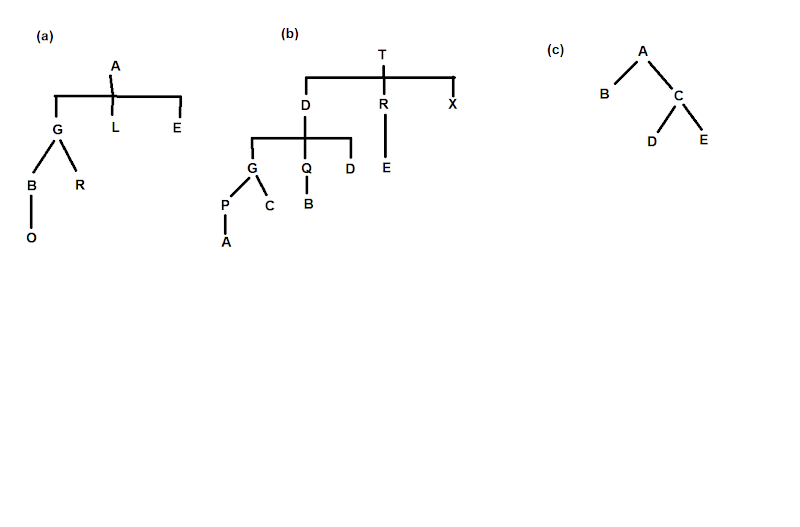
0101: 4 units of time

It would take 25 total units of time to build the tree.

* 15 units of time
* k units of time
* n units of time

4.





5.

public static <T> BTNode<T> genTreeParent(BTNode<T> x) {

if (x.parent.left == x) {

return x.parent;

} else {

return genTreeParent(x.parent);

}

}

public static <T> BTNode<T> genTreekthChild(BTNode<T> x, int k) {

if (x.left == null || k == 0) {

throw new NoSuchElementException();

}

int count = 0;

BTNode<T> kid = x.left;

while (kid != null) {

count++;

if (count == k)

return kid;

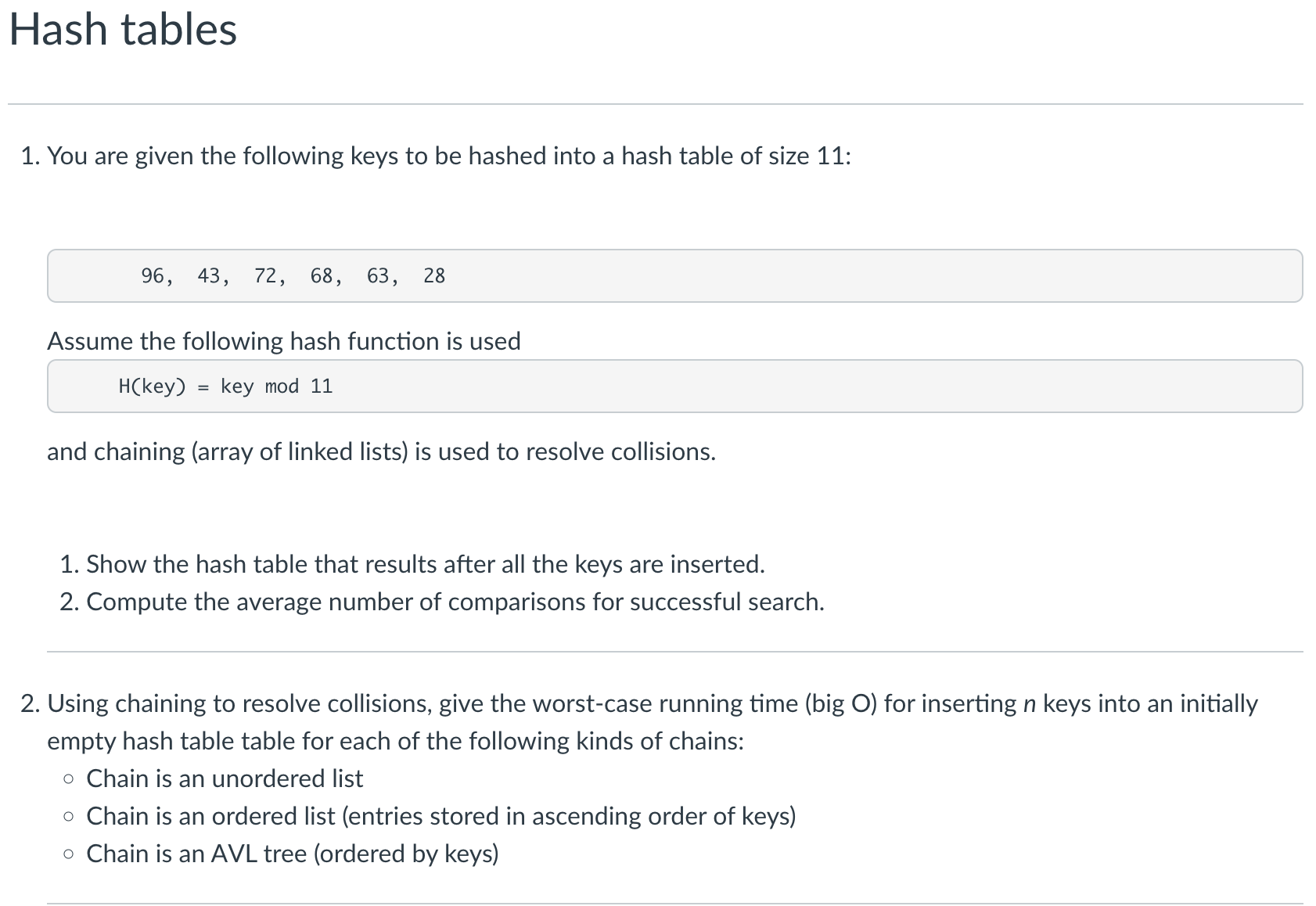
kid = kid.right;

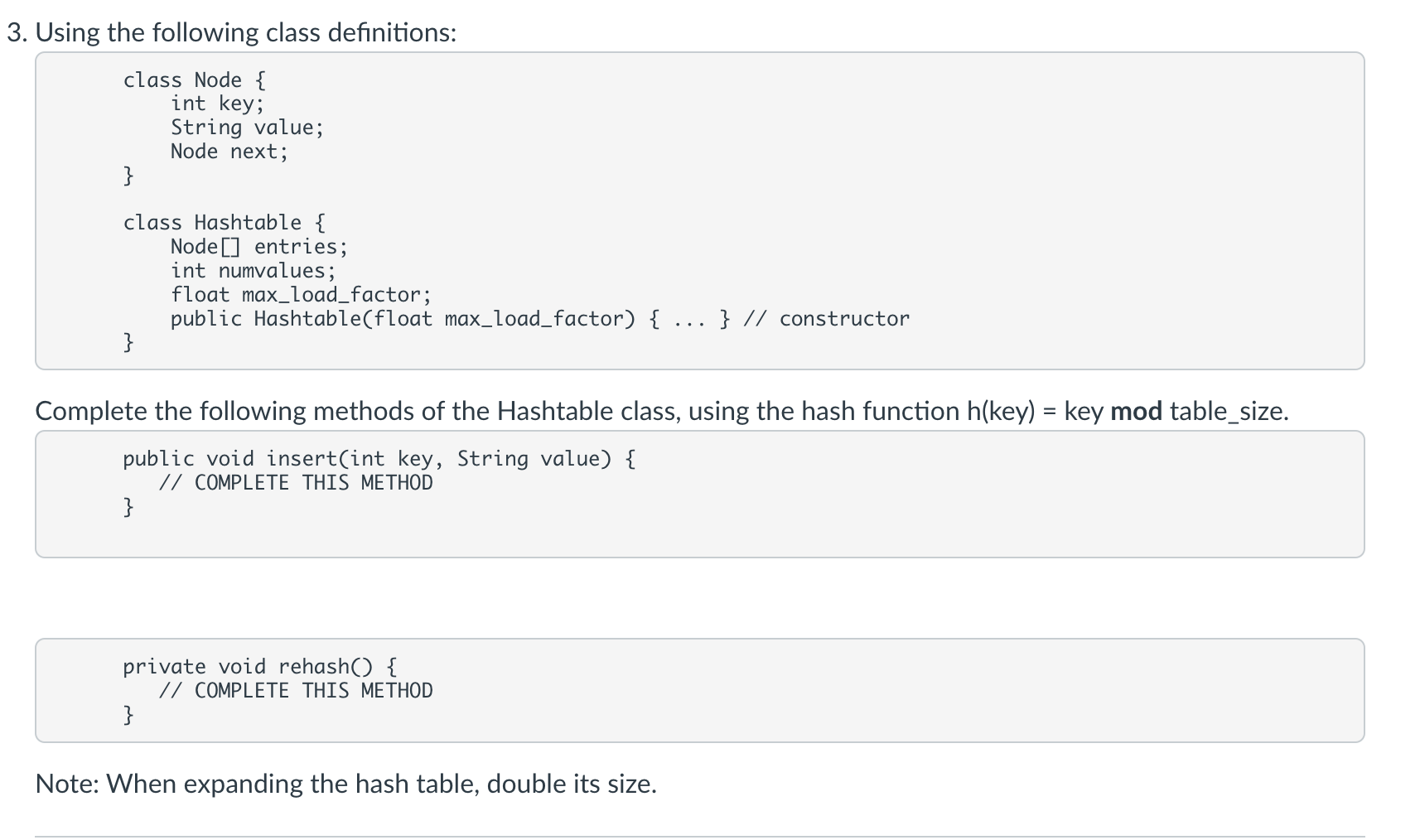
}

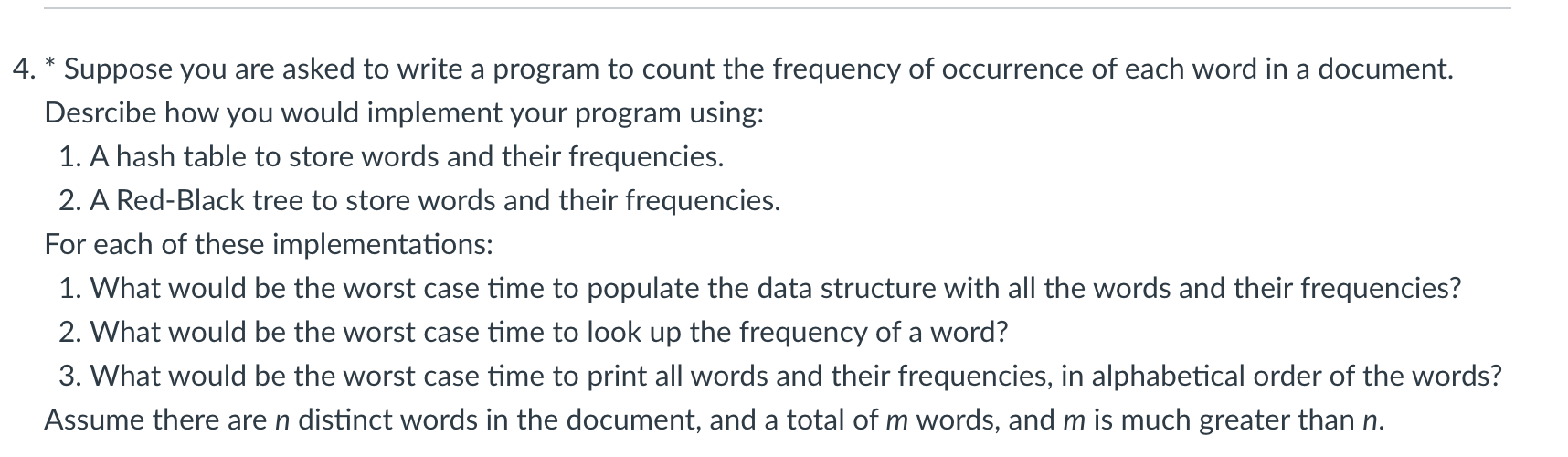
throw new NoSuchElementException();

}

**Problem Set #7**

****

****

****

1)

a)

| 0 |  |  |
| --- | --- | --- |
| 1 |  |  |
| 2 | 68 |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 | 28 | 72 |
| 7 |  |  |
| 8 | 63 | 96 |
| 9 |  |  |
| 10 | 43 |  |
| 11 |  |  |

b)

68 - 1 comparisons

72 - 1 comps

28 - 2 comps

96 - 1 comps

63 - 2 comps

43 - 1 comps

(1+1+2+1+2+1) / 6 = 1 1/3

Average of 1 ⅓ comparisons per successful search.

2) When Chain is:

* Unordered List insertion time: **O(n)**
  + Entry is added to front of list in its index.
* Ordered Linked List insertion time: **O(n^2)** 
  + Entry has to iterate through the list in its index to get to its position.
* AVL (balanced) Tree insertion time: **O(n lg(n))** 
  + Entry has to iterate through the height of the tree in its index to get to its position.

3)

public void insert(int key, String value) {

int hash = key % entries.length;

int i = hash & (entries.length - 1);

Node n = entries[i];

while (n != null) {

if (key.equals(n.key)) {

n.value = value;

return;

}

n = n.next;

}

n = entries[i];

Node newNode = new Node();

newNode.key = key;

newNode.value = value;

newNode.next = n;

entries[i] = newNode;

numvalues++;

if (numvalues > entries.length) {

rehash();

}

return;

}

private void rehash() {

Node[] prevEntries = entries;

Node[] newEntries = new Node[entries.length \* 2];

for (int x = 0; x < entries.length; x++) {

Node n = prevEntries[x];

while (n != null) {

Node next = n.next;

int hash = n.key % newEntries.length;

int i = hash & (newEntries.length - 1);

n.next = newEntries[i];

newEntries[i] = n;

n = next;

}

}

this.entries = newEntries;

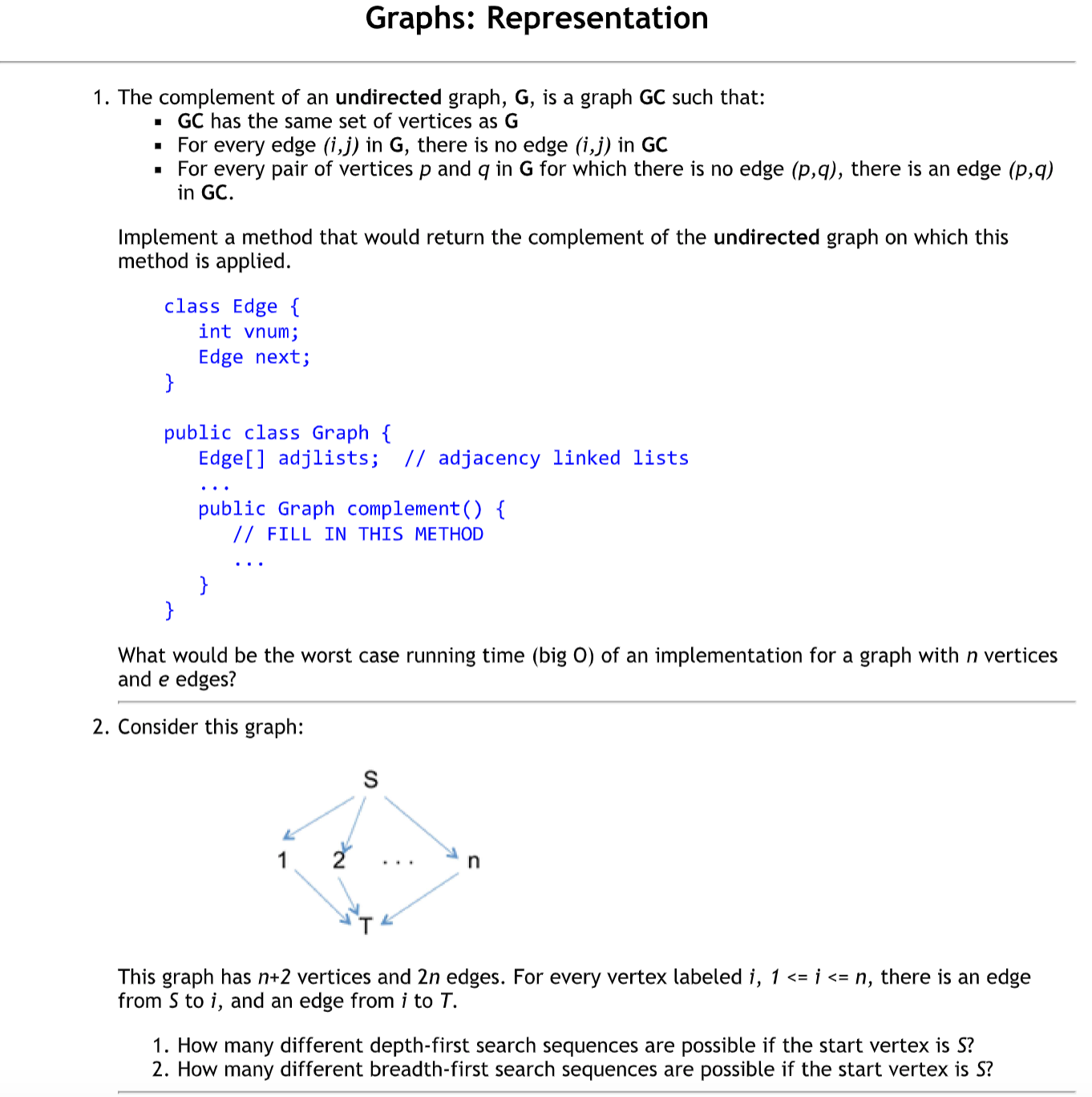
return;

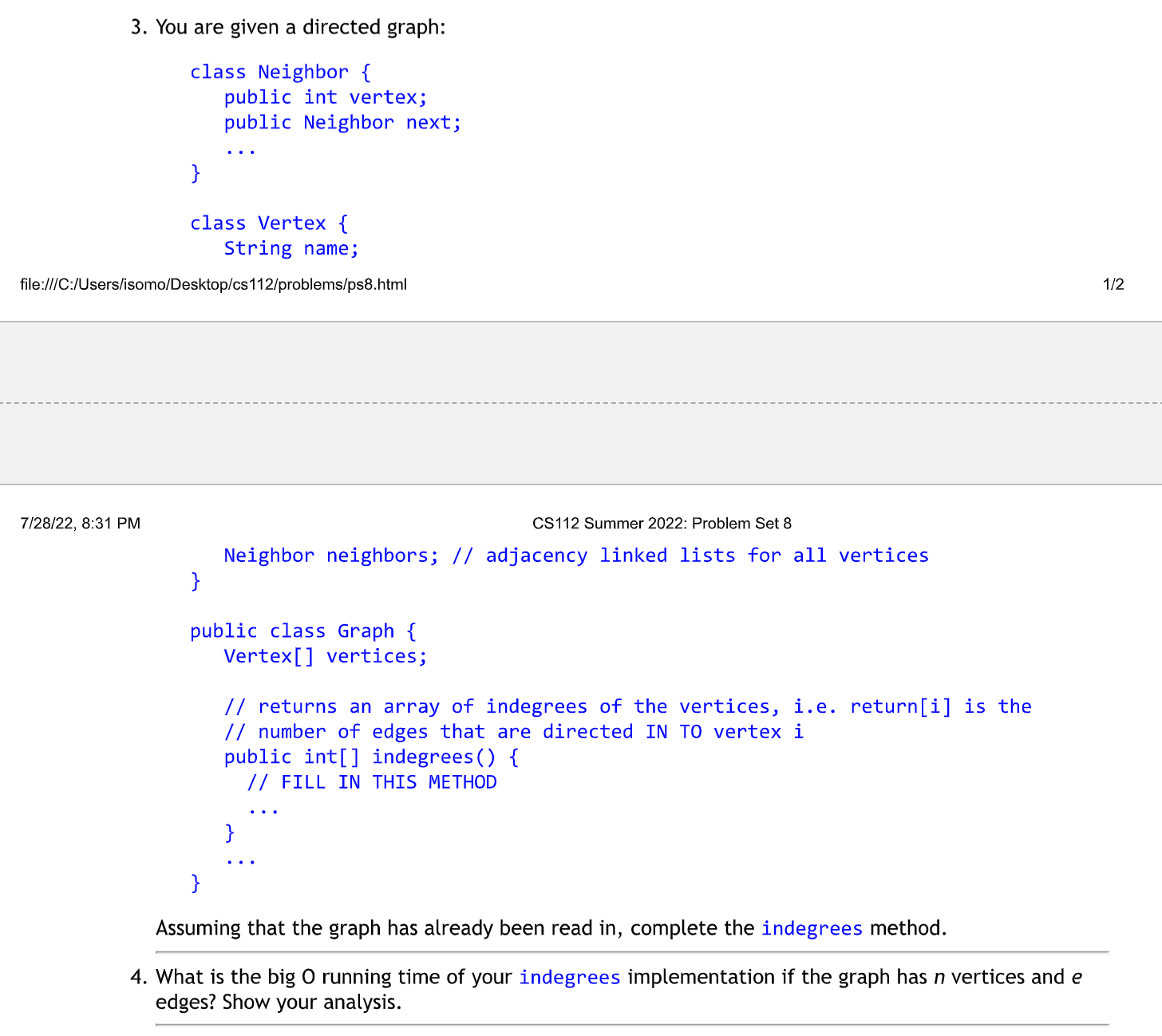
}

4)

1. Hash Table: Use the characters in the words as the keys for where they go in the table. The frequencies are stored inside each word’s node.
2. RB Tree: Every word would be added alphabetically to the tree. For each non-unique word, do not add the same word, but increase the frequency by 1 for the existing word in the tree.
3. Hash Table
   1. Populate time: O(m\*n) - Iterates through total words and table.
   2. Lookup time: O(n) - Iterates through table.
   3. Print time: O(n) - Iterates through table.
4. RB Tree:
   1. Populate time: O(m\*lg(n)) - Iterates through total words and height of tree.
   2. Lookup time: O(lg(n)) - Iterates through single branch of the tree.
   3. Print time: O(n) - Iterates through every element of RB Tree.

**Problem Set #8**

****

****

1.

public Graph complement() {

Graph complement = new Graph();

complement.adjlists = new Edge[this.adjlists.length];

boolean exists = false;

for (int i = 0; i < this.adjlists.length; i++) {

Edge current = this.adjlists[i];

while (current != null) {

exists = false;

for (int j = 0; j < adjlists.length; j++) {

if (this.adjlists[j].vnum == current.vnum) {

exists = true;

}

}

if (exists == false) {

Edge newEdge = new Edge();

newEdge.vnum = current.vnum;

Edge temp = complement.adjlists[i];

complement.adjlists[i] = newEdge;

newEdge.next = temp;

}

current = current.next;

}

}

return complement;

}

Worst case: O(n^2) as you would have to iterate through the vertices and the edges of each vertex.

2.

* Depth-first search sequences: n possible.
* Breadth-first search sequences: 1 possible.

3.

public int[] indegrees() {

int[] indeg = new int[this.vertices.length];

for (int i = 0; i < this.vertices.length; i++) {

Neighbor current = this.vertices[i].neighbors;

while (current != null) {

indeg[current.vertex]++;

current = current.next;

}

}

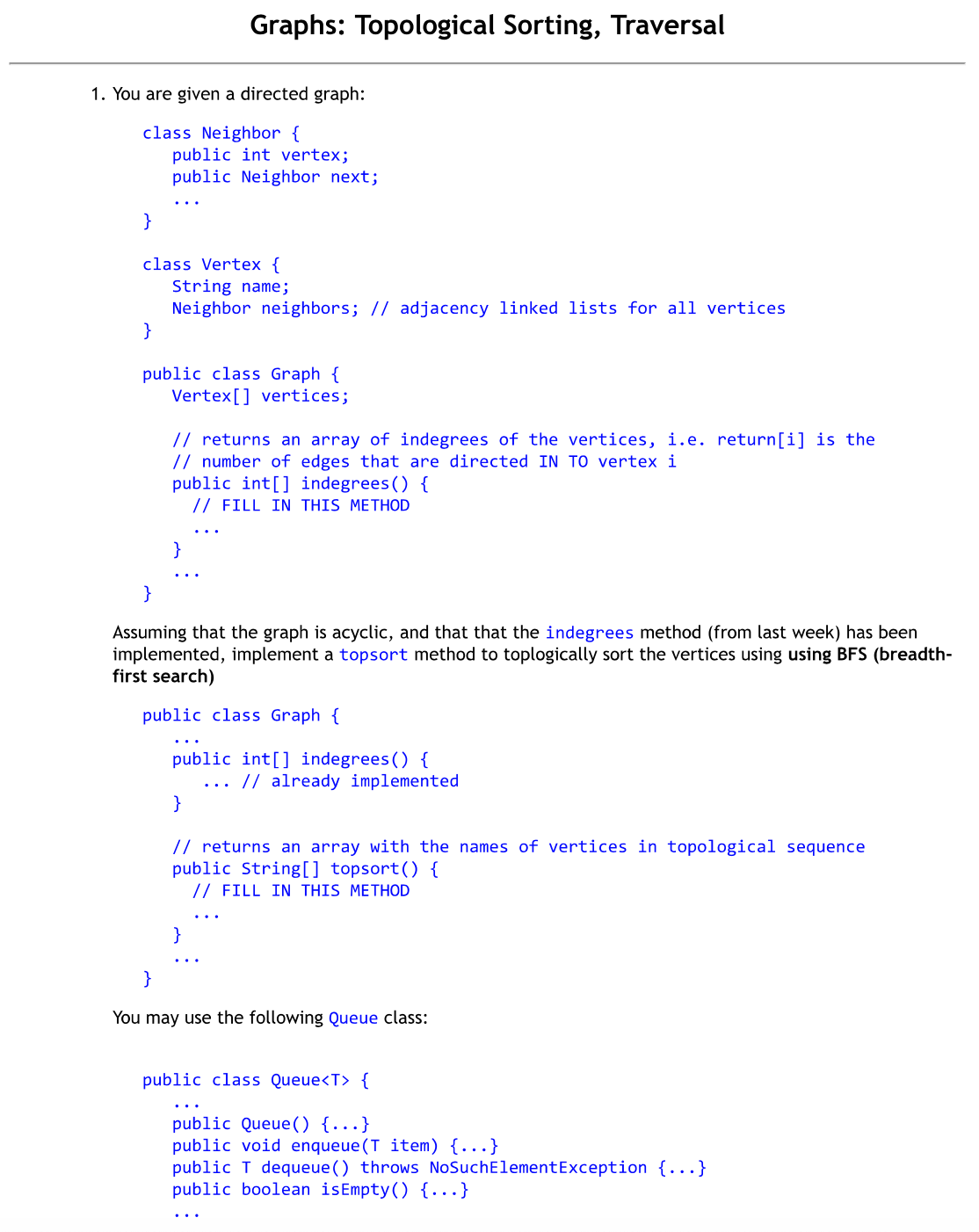
return indeg;

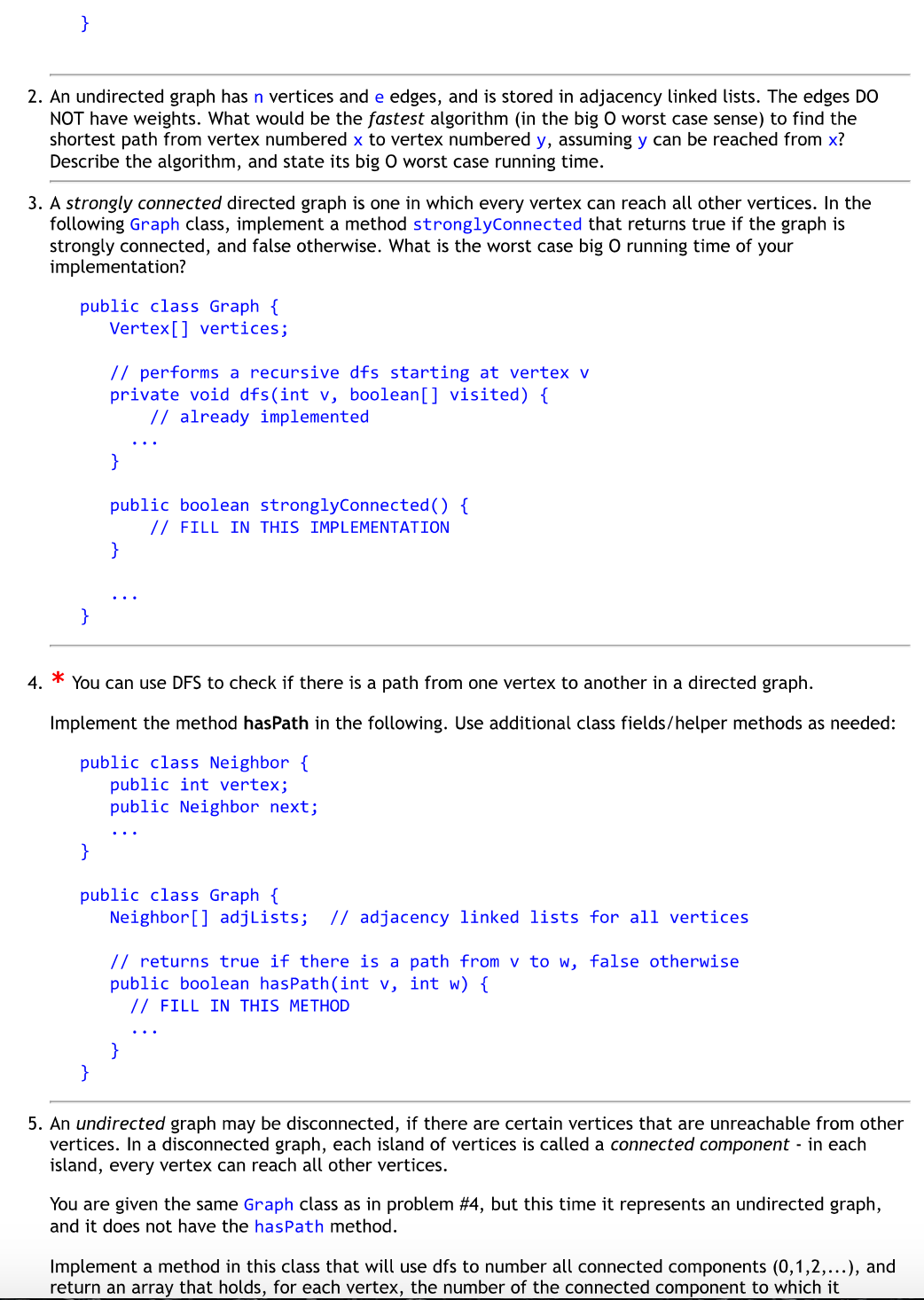
}

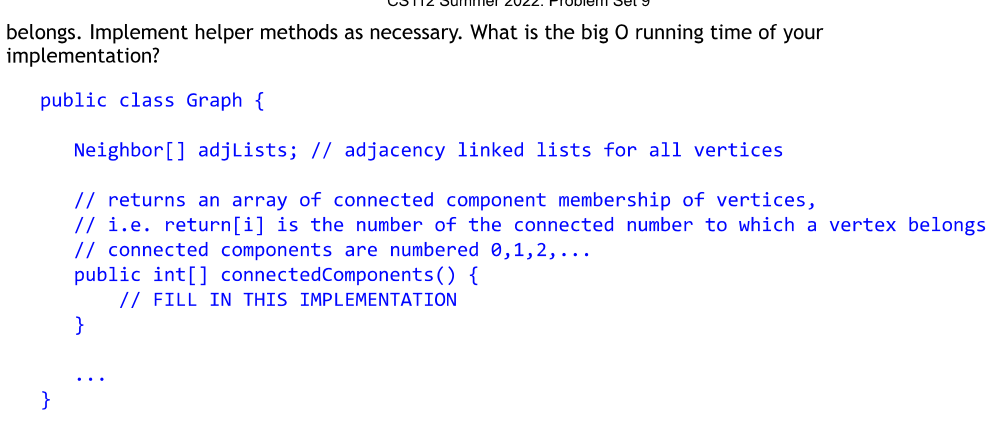
4.

The Big O of indegrees() would be O(n+e). While going through the front of a vertex's neighbors list, update the vertex’s indegree, and then access the neighbor of a vertex are counted as each unit time operation. Since there are “e” neighbors for all, including for all vertices, the neighbor access contributes to “e” units of time. By accessing the front of a vertex's neighbors list, this is done n times in total (once per vertex). As there are e indegree updates, one per edge exists. The total is e + n + e = n + 2e, which should result to O(n + e).

**Problem Set #9**



****

****

1)

public int[] indegrees() {

int[] indeg = new int[this.vertices.length];

for (int i = 0; i < this.vertices.length; i++) {

Neighbor current = this.vertices[i].neighbors;

while (current != null) {

indeg[current.vertex]++;

current = current.next;

}

}

return indeg;

}

public String[] topsort() {

int[] pathsRemaining = indegrees();

String[] sortedTable = new String[vertices.length];

int tableCounter = 0;

Queue myQueue = new Queue();

for (int i = 0; i < vertices.length; i++) {

if (pathsRemaining[i] == 0) {

myQueue.enqueue(i);

sortedTable[tableCounter] = vertices[i].name;

tableCounter++;

}

}

while (!myQueue.isEmpty) {

int v = myQueue.dequeue;

pathsRemaining[v]--;

if (pathsRemaining[v] == 0) {

sortedTable[tableCounter] = vertices[v].name;

tableCounter++;

}

Neighbor current = vertices[v].neighbors;

while (current != null) {

myQueue.enqueue(current.vertex);

current = current.next;

}

}

return sortedTable;

}

2)

We would use a breadth-first search algorithm to find the shortest path. We will start by creating two queues and enqueueing the initial element x in the first queue. Next, we will dequeue x and enqueue every vertex on its adjacency list, including the information about the previous vertex into the second queue. Essentially, every time we dequeue from queue 1, our dequeue from queue 2 will give us the information about the shortest path to get to queue 1’s dequeued element from x. We will continuously dequeue elements from both queues and enqueue elements and their path information into queues 1 and 2 until we reach y in queue 1. After that, we will dequeue from queue 2 to obtain the shortest path to get to y from x.

The Big O would be O(n+e) as you would have to navigate through all the vertices and edges to get from x to y.

3)

public boolean stronglyConnected() {

for (int i = 0; i < vertices.length; i++) {

boolean[] checkArray = new boolean[vertices.length];

dfs(vertices[i], checkArray);

for (int j = 0; j < vertices.length; j++) {

if (checkArray[j] == false)

return false;

}

return true;

}

}

Worst case Big O runtime is **O(n^2+ne)** as you would have to check that every vertex has an edge to every other vertex.

4)

public boolean hasPath(int v, int w) {

boolean[] myMarked=new boolean[adjLists.length];

return hasPath(v, w, myMarked);

}

public boolean hasPath(int v, int w, boolean[] marked) {

marked[v] = true;

Neighbor current = adjLists[v];

while (current != null) {

if (current.vertex == w) {

return true;

}

if (!marked[current.vertex]) {

hasPath(current.vertex, w, marked);

}

current = current.next;

}

return false;

}

5)

public void explore(int v,int compValue, int[] connectedArray) {

if(connectedArray[v]>-1) return;

connectedArray[v]=compValue;

Neighbor current=adjLists[v];

while (current!=null) {

if (connectedArray[current.vertex]==-1) {

explore(current.vertex, compValue, connectedArray);

}

current=current.next;

}

}

public int[] connectedComponents() {

int compCount=-1;

int[] connectedArray=new int[adjLists.length];

for (int i=0;v<adjLists.length;i++) {

connectedArray[i]=-1;

}

for (int v=0;v<adjLists.length;v++) {

if (connectedArray[v]==-1) {

compCount++;

explore(v,compCount,connectedArray);

}

}

}

Big O: **O(n+e)** as it traverses all vertices and edges.

**REFERENCES**

* Problem Sets (Instructions) in GitHub:
  + <https://github.com/USMC1941/CS112-Rutgers/tree/master/Problem_Sets>