CS 61BL Data Structures & Programming Methodology

Summer 2019

FINAL EXAM SOLUTION

This exam has 9 questions worth a total of 60 points and is to be completed in 170 minutes. The exam is closed book except for three double-sided, handwritten cheat sheets. No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided.

Write the statement below in the blank provided and sign. You may do this before the exam begins.

"I have neither given nor received any assistance in the taking of this exam."

I have neither given nor received any assistance in the taking of this exam.

Signature: Matthew Sit

Question	Points
1	5
2	7
3	7
4	8
5	0
6	9
7	9
8	9
9	6
Total	60

Name	Matthew Sit
Student ID	1234567890
GitHub account #	su19 - s1
Lab Section #	0
Name of person to left	Jackson Leisure
Name of person to right	Christine Zhou

- There may be partial credit for incomplete answers. Write as much of the solution as you can, but we may deduct points if your answers are much more complicated than necessary.
- **Work through the problems with which you are comfortable first.** Do not get overly captivated by interesting design issues or complex corner cases you're not sure about.
- Not all information provided in a problem may be useful, and **you may not need all lines**. For code-writing questions, **write only one statement per line** and **do not write outside the lines.**
- Unless otherwise stated, all given code on this exam should compile. All code has been compiled and executed, but in the event that we do catch any bugs in the exam, we'll announce a fix. **Unless we specifically give you the option, the correct answer is not, 'does not compile.'**

1. (5 pts) **Regex**

For each pattern shown below, assume that we are in a Java context (so all escape characters are double escaped in our pattern), that ␣ is equivalent to a space character, and that `java.util.regex.*` is imported as needed. When marking an answer, you must fully fill the corresponding box. If you change your answer, erase any undesired markings completely.

(a) **Repetition**

Mark all of the options below that fully match the pattern: `a?`

[✓ *(empty string)*] [✓ `a`] ☐ `aa` ☐ `aaa`

Mark all of the options below that fully match the pattern: `b*`

[✓ *(empty string)*] [✓ `b`] [✓ `bb`] [✓ `bbb`]

Mark all of the options below that fully match the pattern: `c+`

☐ *(empty string)* [✓ `c`] [✓ `cc`] [✓ `ccc`]

Mark all of the options below that fully match the pattern: `d{1}`

☐ *(empty string)* [✓ `d`] ☐ `dd` ☐ `ddd`

Mark all of the options below that fully match the pattern: `e{1,2}`

☐ *(empty string)* [✓ `e`] [✓ `ee`] ☐ `eee`

(b) **Character Classes**

Mark all of the options below that fully match the pattern: `[ˆchicken]`

☐ `[ˆchicken]` ☐ `hicken` [✓ `z`]
☐ `chicken` ☐ `h` ☐ `zzz`

Mark all of the options below that fully match the pattern: `\\d\\D\\s\\S\\w\\W`

☐ `12␣␣ab` ☐ `\1\2\␣\␣\a\b` [✓ `1a␣bc#`] ☐ `\1\a\␣\b\c\#`

Mark all of the options below that fully match the pattern: `[a+?]`

☐ `(empty string)` [✓ `a`] ☐ `aa` ☐ `aaa`

Name: Matthew Sit

(c) Fill in the comments in the main method to indicate what is outputted by each `System.out.println` command. All of this code compiles and no runtime errors occur. If the answer is a boolean, write either `true` or `false`; do not abbreviate.

```
public static void main(String[] args) {
    Pattern p1 = Pattern.compile("(.+ne)");
    Matcher m1 = p1.matcher("christine");
    System.out.println(m1.matches()); // true

    System.out.println(m1.group(0)); // christine

    System.out.println(m1.group(1)); // christine

    Matcher m1b = p1.matcher("christine's fine dine & brine mine");
    int counter1 = 0;
    while (m1b.find()) { counter1++; }
    System.out.println(counter1); // 1


    Pattern p2 = Pattern.compile("(.?att)");
    System.out.println(p2.matcher("").matches()); // false

    System.out.println(p2.matcher("att").matches()); // true

    System.out.println(p2.matcher("attmatt").matches()); // false
    Matcher m2 = p2.matcher("mattsatt");
    int counter2 = 0;
    while (m2.find()) { counter2++; }
    System.out.println(counter2); // 2


    Pattern p3 = Pattern.compile("^onjack$");
    System.out.println(p3.matcher("^onjack$").matches()); // false

    System.out.println(p3.matcher("onjack").matches()); // true

    Matcher m3 = p3.matcher("^onjack$ ^onjack$ onjack onjack");
    int counter3 = 0;
    while (m3.find()) { counter3++; }
    System.out.println(counter3); // 0
}
```

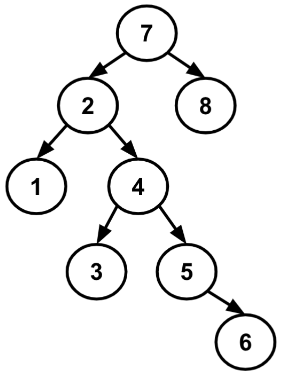
2. (7 pts) **Splay Trees**

This summer, we've learned about B-Trees and Red Black Trees as examples of balanced search trees. Luckily, we have time to learn about one more – splay trees!

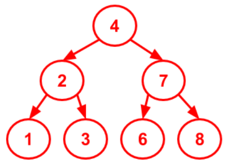
Splay trees have an `insert` and `delete` that behave exactly like those of a binary search tree, except right before these functions return, they perform a *splay* operation. *Splaying* means to perform rotations (like those you learned for Red Black Trees), until a certain node becomes the new root of the tree **(for `insert` that would be the node newly inserted, for `delete` that would be the parent of the deleted node)**. Just like for Red Black Trees, rotations should not violate the invariants of a binary search tree. When applicable, use the **inorder successor** as the replacement for deleted nodes. Please check your answers carefully. We may not be able to award partial credit for this question.

(a) In the splay tree below, perform the **`delete(5)`** operation. This means removing the argument node as if it were a regular binary search tree, and then splaying the parent of the argument node so that it becomes the new root of the entire tree. If you do this correctly, you should see the parent of the `5` move upwards in a zig-zag fashion.

Draw your final answer in the box in the bottom right-hand corner. Perform any scratch work (which will not be graded) outside of this answer box.



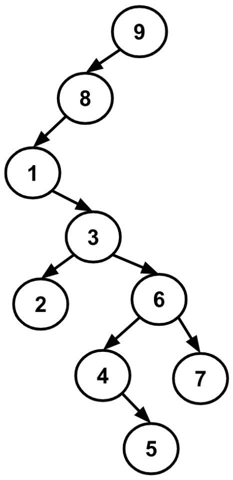
Answer Box



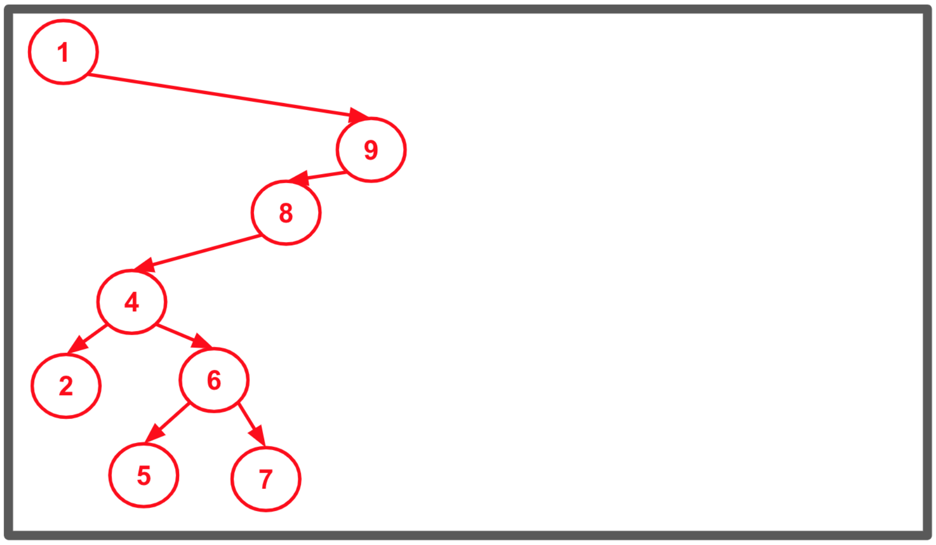
Name: Matthew Sit

(b) In the splay tree below, perform the **`delete(3)`** operation. This means removing the argument node as if it were a regular binary search tree, and then splaying the parent of the argument node so that it is the new root of the entire tree. If you did this correctly, you should see the parent of the `3` move upwards in a zig-zig fashion (as opposed to zig-zag as in the previous question).

Draw your final answer in the box in the bottom right-hand corner. Perform any scratch work (which will not be graded) outside of this answer box.



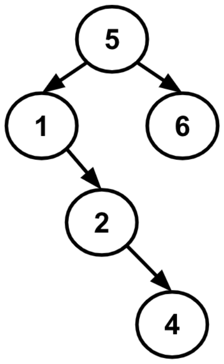
Answer Box

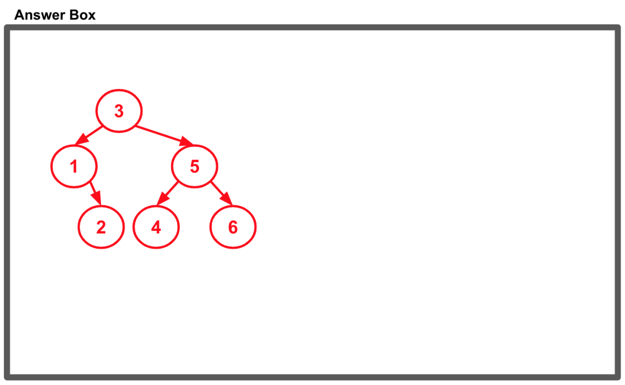


(c) Finally, let's perform a splay operation which consists of multiple zigs and zags.

In the splay tree below, perform the **`insert(3)`** operation. This means inserting the argument node as if it were a regular binary search tree, and then splaying the newly inserted argument node so that it is the new root of the entire tree.

Draw your final answer in the box in the bottom right-hand corner. Perform any scratch work (which will not be graded) outside of this answer box.





(d) Answer the following runtime questions in terms of the following variables:

- `N`, the number of nodes in the tree.

Make sure to use the correct asymptotic bound notations (O, Ω, Θ), and to write it clearly (if you mean to write O, do not put a little loop at the top, because it may be interpreted as a Θ). Simplify your answer if applicable.

What is best case runtime of the *splay* operation?

$\Theta(1)$

What is worst case runtime of the *splay* operation?

$\Theta(N)$

What is runtime of searching for a node that was just inserted and splayed?

$\Theta(1)$

(Splay trees are fantastic in providing fast access to recently accessed nodes. This is called **locality of reference** and in practice can benefit runtime.)

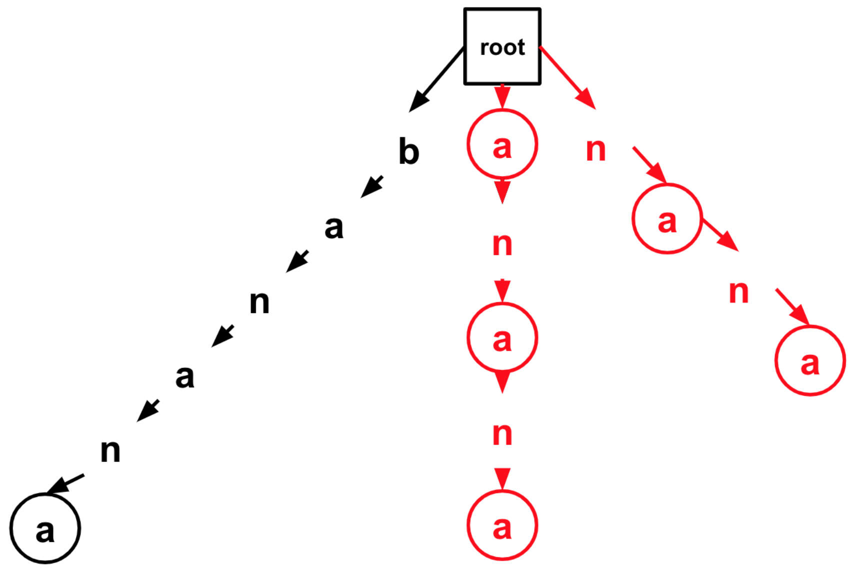
3. (7 pts) **Suffix Tree**

In this question, you will implement a data structure useful for quickly determining whether a query string is contained within another string. This can be useful in contexts such as genome alignment in computational biology. A few simplifications have been made here from the full suffix tree data structure typically used.

A *suffix* is the last some number of letters in a word. For example, all of the suffixes of the word `giraffe` are `giraffe`, `iraffe`, `raffe`, `affe`, `ffe`, `fe`, and `e`. A *suffix tree* is a trie that contains all of the suffixes of a single word.

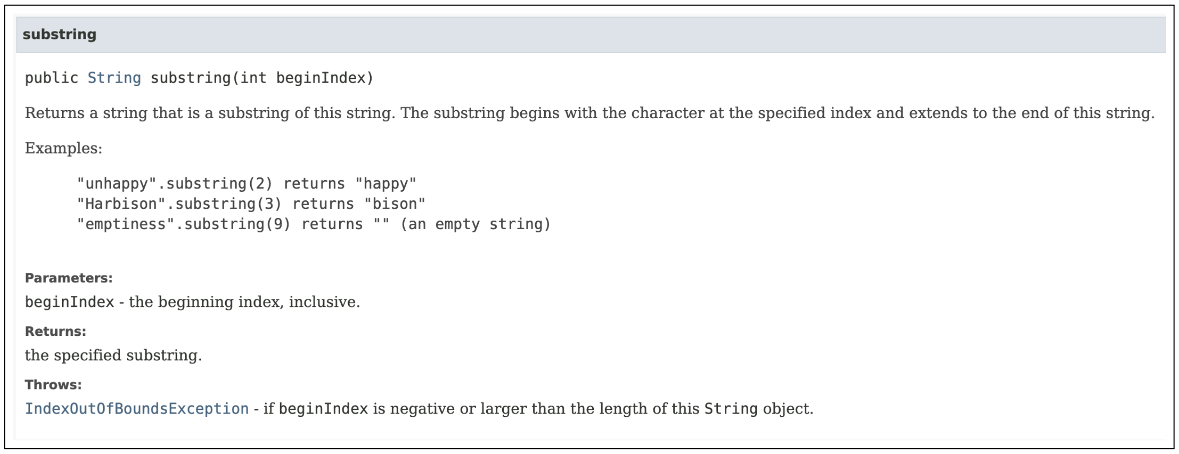
(a) Draw the suffix tree for the word, `banana`. **This is exactly the same as inserting all of the suffixes of `banana`, which are `banana`, `anana`, `nana`, `ana`, `na`, and `a`, into a trie as you learned in lab.**

The root node is provided for you below, as well as the first insertion. **Nodes representing the last letter in a word should be circled, while nodes representing letters that do not mark the end of a word should not be surrounded by a circle.** (Recall that in code this would be a `boolean` instance variable indicator).



Name: Matthew Sit

(b) We will make a class called `Suffixes` that will allow us to loop over all of the suffixes of a `String word` (we will use this in part c in the `SuffixTree` constructor). Fill in the blanks so that this class behaves properly. You may not need to use all blanks provided. The following method from the `String` class will be useful.



substring

```
public String substring(int beginIndex)
```

Returns a string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.

Examples:

```
     "unhappy".substring(2) returns "happy"
     "Harbison".substring(3) returns "bison"
     "emptiness".substring(9) returns "" (an empty string)
```

Parameters:
`beginIndex` - the beginning index, inclusive.

Returns:
the specified substring.

Throws:
`IndexOutOfBoundsException` - if `beginIndex` is negative or larger than the length of this `String` object.

```
import java.util.Iterator; import java.util.NoSuchElementException;

public class Suffixes implements Iterable______________________<String> {
    private String word;
    public Suffixes(String word) { this.word = word; }

    public Iterator______________________<String> iterator______________________() {

        return new SuffixHelper();________________________
    }
    private class SuffixHelper implements Iterator______________________<String> {

        private int index = 0;____________________________

        public boolean______________________ hasNext() {

            return index < word.length();______________________
        }
        public String next() {
            if (!hasNext()) { throw new NoSuchElementException(); }

            index++;______________________________________________

            return word.substring(index - 1);_________________

            ______________________________________________

}}}
```

(c) Here is an outline of the `Trie` class we'll be using.

```
public class Trie {
    /* Implementation not shown. You may assume it works as intended. */
    public void insert(String s) { ... }

    // There are also other methods, but we won't be working with them here.
}
```

We will now implement the `SuffixTree` class. The constructor should take its `String word` argument and `insert` all of the suffixes. **You must use an enhanced for-loop here.** Notice that the `insert` method is disabled for the `SuffixTree`; this is because we only want the suffixes of our specified `word` in the `SuffixTree` – other insertions after this are not allowed.

```
import java.util.Iterator;
public class SuffixTree extends Trie {
    public SuffixTree(String word) {

        for (String suffix :  new Suffixes(word)______________________) {

            super.insert(suffix);_________________________________
        }
    }

    @Override
    public void insert(String key) {
        throw new UnsupportedOperationException();
    }
}
```

Now implement the exact same thing, but **do not use an enhanced for-loop**.

```
public SuffixTree(String word) {

    Iterator<String> si = new Suffixes(word).iterator();____________________

    while (si.hasNext()________________________) {

       super.insert(si.next());_________________________________
    }
}
```

(d) Assuming that we are working with a constant size alphabet, answer the following runtime questions in terms of the following variables:

- `N`, the length of the argument `String word`.
- `Q`, the length of a query `String`.

For example, if we wanted to know whether the query word `nan` is contained within the argument word `banana`, `N` would be 6 and `Q` would be 3.

Make sure to use the correct asymptotic bound notations (O, Ω, Θ), and to write it clearly (if you mean to write O, do not put a little loop at the top because it may be interpreted as a Θ). Simplify your answer if applicable.

What is the runtime of the `SuffixTree` constructor (in other words, how long does it take to insert all the suffixes of the argument word)? Note that `String`'s `substring` method runs in time linear to the length of the substring being made.

$\Theta(N^2)$

What is the worst case runtime of `find`ing the query word in a `SuffixTree`?

$\Theta(Q)$

4. (8 pts) **Potpourri**

Unless otherwise noted, mark one answer per question. When marking an answer, you must fully fill the corresponding circle or box. If you change your answer, erase any undesired markings completely. You may assume that `hashcode()` and `equals()` take constant time unless otherwise stated.

(a) Assuming that the hash table has just resized, what is the **best case** runtime for a single add operation to a hash table which doubles in size when the load factor exceeds $\frac{N}{2^N}$, where N is the number of elements in the hash table?

○ $\Theta(1)$ [✓ ● $\Theta(N)$] ○ $\Theta(N^2)$ ○ $\Theta(N^3)$

(b) Suppose that the `.equals()` method takes constant time and `.hashcode()` method takes N time to run. What is the **worst case** runtime for a single add operation to a hash table which doubles in size when the load factor exceeds 1, where N is the number of elements in the hash table?

○ $\Theta(1)$ ○ $\Theta(N)$ [✓ ● $\Theta(N^2)$] ○ $\Theta(N^3)$

(c) Now suppose that the `.equals()` method takes N time and `.hashcode()` method takes constant time to run. What is the **worst case** runtime for a single add operation to a hash table which doubles in size when the load factor exceeds 1, where N is the number of elements in the hash table? (Hint: when rehashing items, are equality checks necessary?)

○ $\Theta(1)$ ○ $\Theta(N)$ [✓ ● $\Theta(N^2)$] ○ $\Theta(N^3)$

(d) Given a weighted, directed Graph $G(V, E)$, what is the **maximum possible** number of minimum weight crossing edges in a single cut?

○ 1 ○ $V - 1$ [✓ ● $\frac{V^2}{2}$] ○ $V^2 - V$

(e) Given a weighted, undirected, acyclic Graph $G(V, E)$, what is the **maximum possible** number of minimum weight crossing edges in a single cut?

○ 1 [✓ ● $V - 1$] ○ $\frac{V^2}{2}$ ○ V^2

(f) Using heuristic h on a Graph $G(V, E)$ with non-negative edge weights, the $A*$ algorithm succeeds in finding the shortest path from any vertex s to any other vertex t. Suppose h is replaced with another heuristic h'. **Select all possible options** for which using h' is *guaranteed* to still find the shortest path.

[✓ $h' = 0$] [✓ $h' = -1$] [✓ $h' = \frac{h}{2}$] ☐ $h' = \sqrt{h}$

(g) One way to provide sorted edges for Kruskal's algorithm is to use a Priority Queue, which would provide the next lightest edge during each iteration of Kruskal's, one at a time as needed. Given a Graph $G(V, E)$, in the **worst case**, how many edges will be removed from the Priority Queue by the end of Kruskal's algorithm?

○ $V - 1$ ○ V^2 [✓ E] ○ $V + E$

(h) Given a Graph $G(V, E)$, in the **worst case**, how many edges will be considered when determining whether to update priority values after a single removal from the Priority Queue in Prim's algorithm?

[✓ $V - 1$] ○ V^2 ○ E ○ $V + E$

5. (0 pts) **PNH**

What is the Trinomial name for the western lowland gorilla? Alternatively, what is the Trinomial name for the Plains bison?

Gorilla Gorilla Gorilla. Alternatively, bison bison bison.

6. (9 pts) **Oak, Elm, Birch, . . .**

Given a naked (non-encapsulated) linked list of **unique** integers, we want to build a binary search tree (BST). There are a lot of valid BST's that could result from a given list, but we will choose an item from the linked list to be the root of our tree, which we will call the pivot. Then, we will recursively build the left and right subtrees of our tree using all of the elements of the linked list that are less than and greater than the pivot, respectively.

Pivots may be chosen three different ways:

```
public enum Pivot {
    FIRST, RANDOM, MEDIAN
}
```

Linked list nodes are defined as follows:

```
public class LLNode {
    public int item;
    public LLNode next;

    public LLNode(int item, LLNode next) {
        this.item = item;
        this.next = next;
    }
}
```

Binary search tree nodes are defined as follows:

```
public class TreeNode {
    public TreeNode left, right;
    public int item;

    public TreeNode(int item, TreeNode left, TreeNode right) {
        this.item = item;
        this.left = left;
        this.right = right;
    }
}
```

Name: Matthew Sit

(a) To start, let's define a helper method (`chooseRandomPivot`) to help us choose a random pivot out of the items in the list starting at `LLNode node`.

Use `Math.random())` in your solution, which returns a different random double between 0 (inclusive) and 1 (exclusive) each time.

```
private static int chooseRandomPivot(LLNode node) {
    int size = 0;

    for (LLNode temp = node; temp != null; temp = temp.next) {

        size++;
    }

    int index = (int)(size * Math.random());

    while (index > 0) {

        node = node.next;

        index--;
    }
    return node.item;
}
```

(b) Now write the constructor's helper function, which assembles a BST from `LLNode node` based upon `Pivot p`.

```
public class BST {
    TreeNode root;
    public BST(LLNode node, Pivot p) { root = build(node, p); }
    private static int chooseRandomPivot(LLNode n) { /* Implemented in (a). */ }
    private static int chooseMedianPivot(LLNode n) { /* Not shown. */ }

    private TreeNode____________ build(LLNode node, Pivot p) {
        if (node == null) { return null; }
        int pivot;

        if (p == Pivot.RANDOM) {

            pivot = chooseRandomPivot(node);______
        } else if (p == Pivot.FIRST) {

            pivot = node.item;_______________________
        } else {
            pivot = chooseMedianPivot(node);
        }

        LLNode left = null;
        LLNode right = null;

        while (node != null______________________________) {

            if (node.item < pivot______________________________) {

                left = new LLNode(node.item, left)________________;

            } else if (node.item > pivot______________________________) {

                right = new LLNode(node.item, right)_______________;
            }

            node = node.next;___________________________
        }

        return new TreeNode(pivot, build(left, p), build(right, p));__________
    }
}
```

Name: Matthew Sit

(c) Answer the following runtime questions in terms of the following variables:

- `N`, the number of `LLNodes` in the original input linked list.

Make sure to use the correct asymptotic bound notations (O, Ω, Θ) and to write them clearly (if you mean to write O, do not put a little loop at the top, because it may be interpreted as a Θ). Simplify your answers.

What is the average runtime of a single call to `chooseRandomPivot`?

$\Theta(N)$

What is the best case runtime of a single call to `chooseRandomPivot`?

$\Theta(N)$

For the remaining runtime questions, answer in terms of the following variables:

- `N`, the number of `LLNodes` in the original input linked list.
- `A`, your answer for the average runtime of `chooseRandomPivot`. (For example, if the answer was constant time, and `chooseRandomPivot` is run `N` times, then the overall work performed is `AN`, not `N`.) (This is to protect you from a possible cascading error).
- `B`, the average runtime for a single call to `chooseMedianPivot`.

What is the runtime of the `BST` constructor, including the work performed by its helper function? Keep in mind that there are *three* pivot strategies written in to the code. Provide a Big O and a Big Omega bound.

$O($ $(A + B + N)N$ `max`(A, B, N) work$)$ per recursive call; at most N recursive calls possible with bad pivots.

$\Omega($ $N \log N$ Bottle-neck `while` costs$)$ N, at least $\log N$ recursive calls possible with good pivots.

7. (9 pts) **Kruskal Crazy**

   (a) Kruskal's algorithm for constructing a minimum spanning tree from a graph begins by sorting all of the edges. Implement `sortEdgesByWeight` to do this for our graph, as constrained by the comment in the `Graph` constructor. If there are edges with the same weight, tiebreak by listing the edge with the lower `from` vertex number first. If there is still a tie after doing that, then tiebreak by listing the edge with the lower `to` vertex number first.

   You should use `ArrayList`'s `sort` method, which in this case accepts a `Comparator<Edge>` as its parameter, and destructively sorts it in a stable fashion.

```
import java.util.*;
public class Graph {
    class Edge { int from, to, weight; }

    private Edge[][] adjMatrix;

    public Graph() {
        /*  Constructor implementation not shown.
         *  Initializes the matrix and addes Edges to it.
         *  You may assume that the graph is undirected and connected.
         *  (If there's an edge from i to j,
         *    the same edge is also stored in the matrix for j to i.)
         *  Self-edges (from i to i) are never present. */
    }

    public List<Edge> sortEdgesByWeight() {
        ArrayList<Edge> edges = new ArrayList<>();

        for (int i = 0___; i < adjMatrix.length__________; i++) {

            for (int j = i___; j < adjMatrix.length__________; j++) {
                if (adjMatrix[i][j] != null) {
                    edges.add(adjMatrix[i][j]);
                }
            }
        }
        edges.sort((e1, e2) -> e1.weight - e2.weight__________________________);
        return edges;
    }
}
```

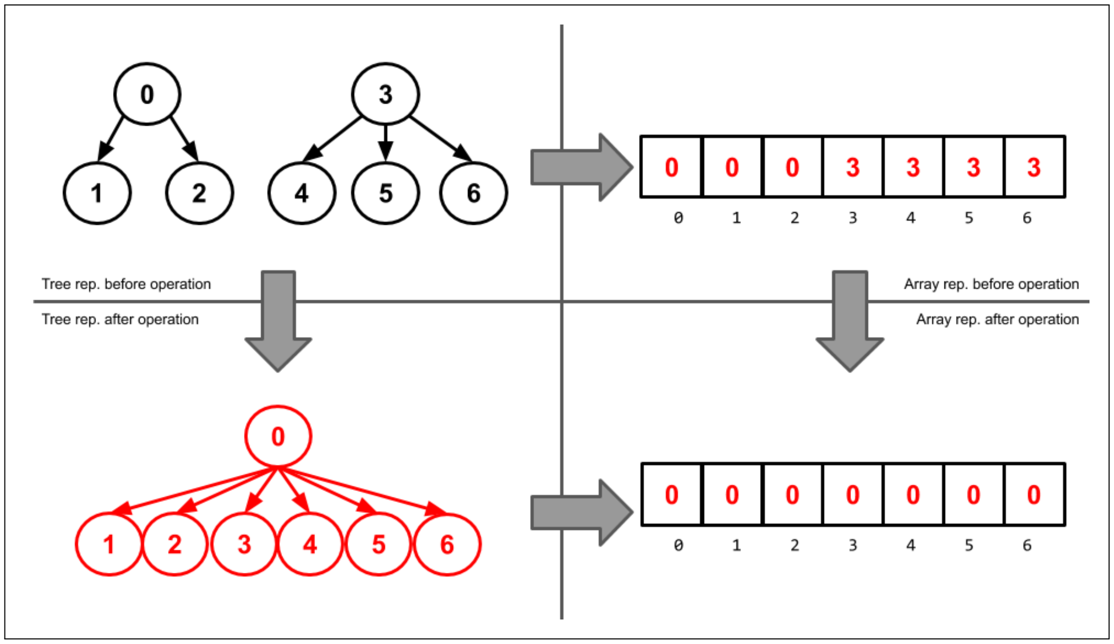
Name: Matthew Sit

(b) Another critical piece of Kruskal's algorithm is the necessity to verify whether two vertices are connected already (`find`), and the ability to connect two vertices so we remember they're connected in future queries (`union`). We learned about several variations of Disjoint Sets to provide such functionality.

For each subpart in this question, convert the tree representation given to you in the top left box to the array representation, which you should put in the top right box. Then in the bottom left box, perform the indicated operation according to the stated Disjoint Set implementation and draw the tree representation result. In the bottom right box, do the same but for the array representation.

You must use the conventions and procedures we taught in lab this summer. Keep in mind that `union` calls upon `find` on its arguments prior to connecting them. If there is a tie, break it arbitrarily. Only your final answer within the boxes will be graded; perform any scratch work (which will not be graded) outside of the boxes.

i. Perform `union(0, 6)` on this **Quick Find** data structure.

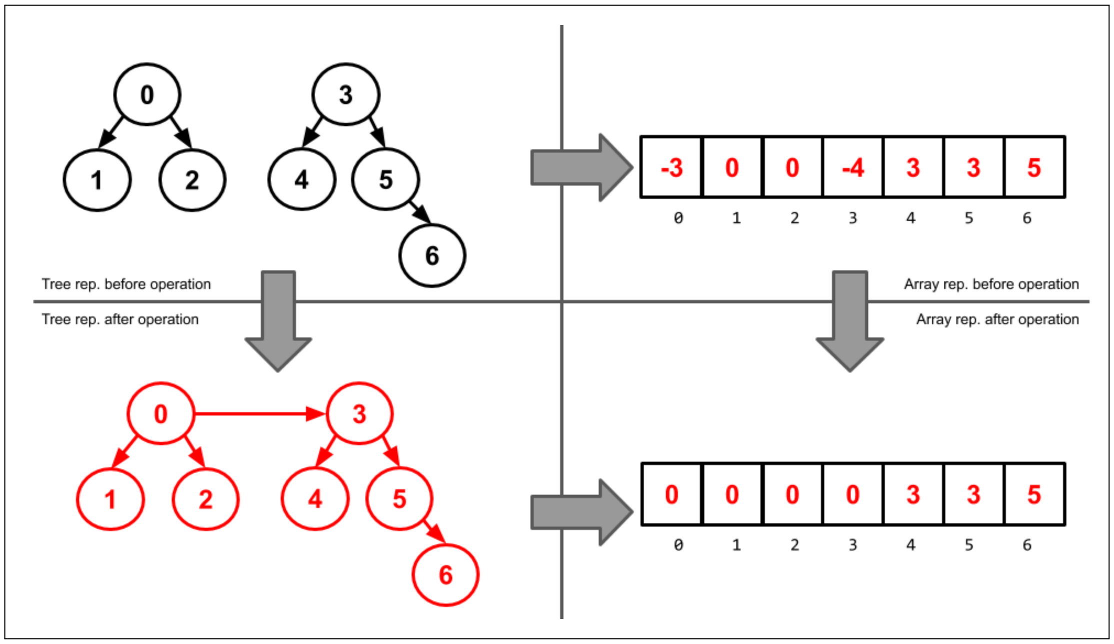


0
1
2
3
4
5
6
Tree rep. before operation
Tree rep. after operation
Array rep. before operation
Array rep. after operation

0	0	0	3	3	3	3
0	1	2	3	4	5	6

0	0	0	0	0	0	0
0	1	2	3	4	5	6

ii. Perform `union(0, 6)` on this **Quick Union** data structure (non-weighted, and without path compression).


Tree rep. before operation
Tree rep. after operation
Array rep. before operation
Array rep. after operation

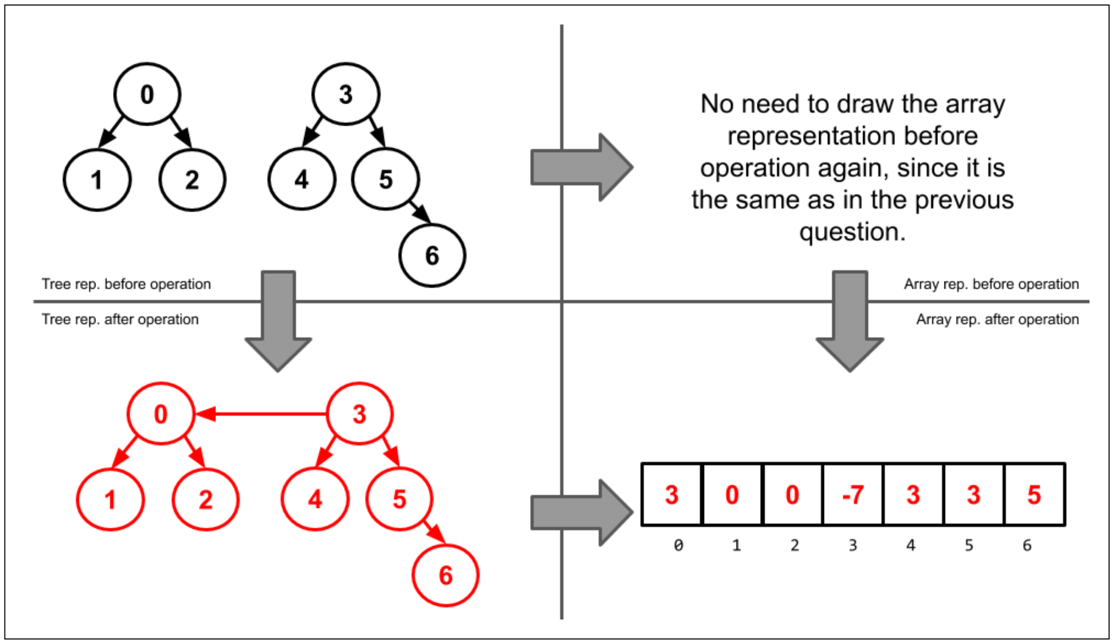
Array rep. before operation:

0	1	2	3	4	5	6
-3	0	0	-4	3	3	5

Array rep. after operation:

0	1	2	3	4	5	6
0	0	0	0	3	3	5

iii. Perform `union(0, 6)` on this **Weighted Quick Union** data structure (without path compression).

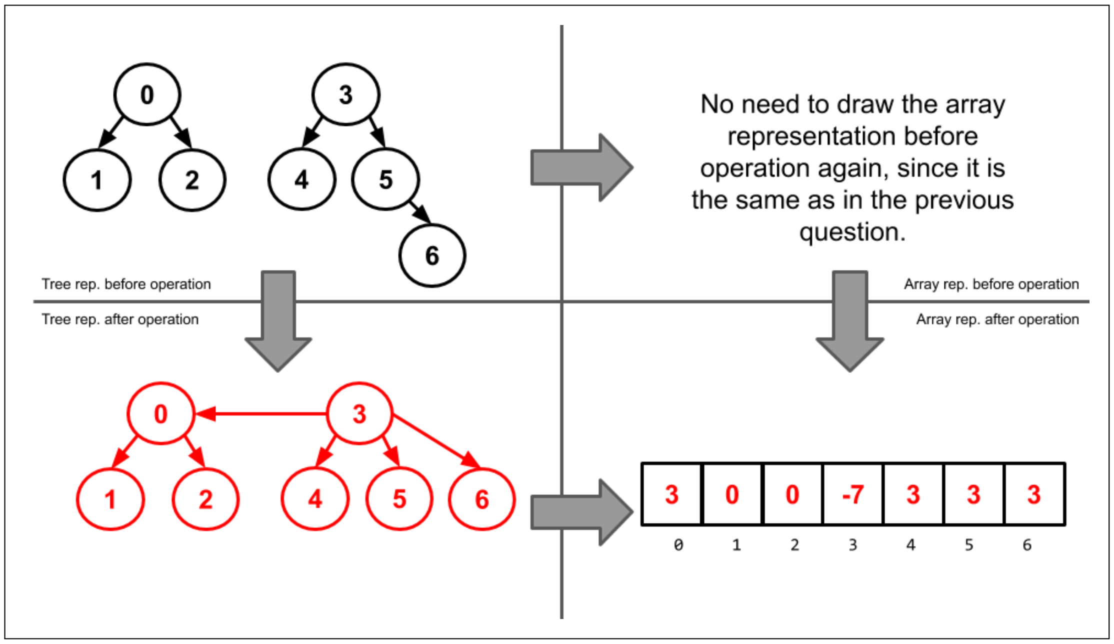

No need to draw the array representation before operation again, since it is the same as in the previous question.
Tree rep. before operation
Tree rep. after operation
Array rep. before operation
Array rep. after operation

Array rep. after operation:

0	1	2	3	4	5	6
3	0	0	-7	3	3	5

Name: Matthew Sit

iv. Perform `union(0, 6)` on this **Weighted Quick Union with Path Compression** data structure.



0
1
2
3
4
5
6
No need to draw the array representation before operation again, since it is the same as in the previous question.
Tree rep. before operation
Tree rep. after operation
Array rep. before operation
Array rep. after operation
0
1
2
3
4
5
6

3	0	0	-7	3	3	3
0	1	2	3	4	5	6

(c) Answer the following runtime questions in terms of the following variables:

- `V`, the number of vertices in our graph.
- `E`, the number of edges in our graph. Note that $\mathtt{E} \leq \mathtt{V}(\mathtt{V}-1)/2$.

Make sure to use the correct asymptotic bound notations (O, Ω, Θ) and to write them clearly (if you mean to write O, do not put a little loop at the top, because it may be interpreted as a Θ). Simplify your answers.

What is the runtime of our `sortEdgesByWeight`, including its call to `ArrayList`'s `sort` method (which runs in $O(N \log N), \Omega(N)$, where N is the number of items being sorted)?

$O(V^2 + E \log E), \Omega(V^2 + E)$; also $V^2 \geq E \implies 2 \log V = \log E$

Given a list of sorted edges (so not including that work in this calculation), what is the worst case runtime of Kruskal's algorithm using **Quick Find** to connect and verify connectedness of vertices?

$E(1) + V(V) + V \in \Theta(E + V^2) = \Theta(V^2)$

Given a list of sorted edges (so not including that work in this calculation), what is the worst case runtime of Kruskal's algorithm using **Quick Union** (unweighted and without path compression) to connect and verify connectedness of vertices?

$E(V) + V(V) + V \in \Theta(V(E + V)) = \Theta(EV)$

Given a list of sorted edges (so not including that work in this calculation), what is the worst case runtime of Kruskal's algorithm using **Weighted Quick Union** (without path compression) to connect and verify connectedness of vertices?

$E(\log V) + V(\log V) + V \in \Theta((\log V)(E + V)) = \Theta(E \log V)$

Given a list of sorted edges (so not including that work in this calculation), what is the worst case runtime of Kruskal's algorithm using **Weighted Quick Union with Path Compression** to connect and verify connectedness of vertices? Disregard the inverse Ackermann function as a constant factor.

$E(1) + V(1) + V \in \Theta(E + V) = \Theta(E)$

Name: Matthew Sit

8. (9 pts) **Proudly Serving The Widest Selection of Data Structures**

For each scenario presented, devise a solution and write down any data structures YOU used in YOUR solution (you may write multiple), as well as the data types of any generics involved. Then fully explain how the data structures you've selected solve the problem. Provide the requested runtime(s) for YOUR solution using the appropriate notation(s). Simplify your runtime answers. Full credit will be awarded to optimal solutions, and partial credit for suboptimal solutions. You may assume that `hashcode()` and `equals()` take constant time.

(a) You've been hired to design a security system for a workplace that tracks where `Employees` are at all times. An `Employee` is either on a `Floor` of a `Building` or outside of any campus `Buildings`. `Employees` must always use their badge to log changes in their location and to unlock the doors, whether it is to enter the ground `Floor` of a `Building`, to move from one `Floor` of a `Building` to another, or to exit a `Building`. Given this data, you should be able to say which `Building` and `Floor` any `Employee` is in, or that they are not inside any `Building` at all. Optimize runtime for both `updates` (people badging in or out) and `lookups`. You only need to store current locations; there is no need to store a log or history of where an `Employee` has been in the past. The amount of space you use should be linear to the number of `Employees`.

Data Structure(s): HashMap<Employee, Building> and HashMap<Employee, Floor>

Usage: If an Employee leaves a Building, remove them from the building map.

If an Employee enters a Building, add them to the building map, and add them and their floor to the floor map.

If an Employee changes floors within a building, update their floor in the floor map.

Lookup an Employee's location by looking in both maps.

If an Employee is not found in the building map, they are not inside, even if they are in the floor map.

Answer in terms of N, the number of `Employees`*; B, the number of* `Buildings`*; and F, the number of floors in the tallest* `Building`*.*

Average case runtime (`update`): $\Theta(1)$ Average case runtime (`lookup`): $\Theta(1)$

Worst case runtime (`update`): $\Theta(N)$ Worst case runtime (`lookup`): $\Theta(N)$

(b) For this question, we're going to be considering a simplified version of html tags, called 61bml. A `Tag` in 61bml consists of 3 parts: an opening symbol `<`, then a word, then a closing symbol `>`.

`Tags` can be either opening `Tags` or closing `Tags`. Closing `Tags` begin with `</` instead of just `<`. `Tags` in 61bml should come in pairs, which means for every opening `Tag`, there should be a closing `Tag` with the matching word. Closing `Tags` should come after opening `Tags`. For example, `<h1>` Hello `</h1>` is how we would make a large header that displays hello.

`Tags` can also be nested - for example, `<html><body>` Content `</body></html>`. One rule is that the most recently opened `Tag` must match the next closed tag - for example, we could not do `<html><body>` Content `</html></body>`. Provide a solution that returns `true` or `false` depending on whether a given list consisting of `N` `Tags` is properly formatted (the input list given to you consists only of the `Tags` themselves). Be sure to specify how your solution determines if it should return `true` or `false`. The amount of space you use should be linear to `N`, the number of `Tags`.

Data Structure(s): Stack<Tag>

Usage: When we see an opening Tag, we push it onto our Stack.

If we see a closing Tag, we pop from our Stack and the popped tag must correspond with this closing Tag.

If not, return false immediately.

If we ever try to pop from the stack but the stack is empty, return false immediately.

At the end, the answer is if the stack is empty.

Answer in terms of N, the number of `Tags`.

Runtime (solution): $\Theta(N)$

Name: Matthew Sit

(c) DSA Music Inc. has noticed that customers tend to listen to the same music genre within one sitting. Their new idea is to create playlists from your most recently listened to `Songs`, where each playlist corresponds to a certain `Genre`. Each playlist should be organized from most recently listened to `Songs` to less recently listened to `Songs`. Also, each playlist can only contain a maximum of `k` `Songs`. You may assume `Songs` store their `Genre`, and that each `Song` only belongs to one `Genre`.

The `update` operation is called whenever a `Song` is being listened to. The `fetch` operation should return a `List` of the ordered Songs of a given `Genre`.

Optimize your solution so that `fetch` runs as quickly as possible, even at the cost of slower `updates`. The amount of space you use should be linear to the number of `Songs`.

Data Structure(s): HashMap<Genre, LinkedList<Song>>

Usage: Maintain a mapping of Genre to Songs of that Genre in a LinkedList<Song>.

Upon update, fetch the list according to the Song's Genre, and search the list for the song, removing it if found. Then, add it to the front of the list. If the list is longer than size k, remove last so that it is size k. Upon fetch, get the list from the map according to the Song's Genre and return it.

Answer in terms of N, the number of unique `Songs` *encountered; G, the number of* `Genres`*; and k, the maximum number of* `Songs` *a single playlist can contain.*

Average case runtime (`update`): $\Theta(k)$ Average case runtime (`fetch`): $\Theta(1)$

Worst case runtime (`update`): $\Theta(G+k)$ Worst case runtime (`fetch`): $\Theta(G)$

9. (6 pts) **The Wiggler™**

The Wiggler™ is a carnival ride with an automated admissions system. Riders purchase tickets from a ticket booth and then walk over to the ride. A gate machine at the ride scans the ticket and allows the rider through if the ticket hasn't been used before. Each ticket has an ID number that starts at 1 and is incremented for each new ticket that's sold. The machine has to keep track of which tickets have been used, and reject those which have already been used before.

Unfortunately, the gate machine has a very limited amount of memory. However, we can exploit the fact that in practice, most riders purchase their tickets and then immediately walk over to the ride: thus, the tickets are roughly redeemed in sequential order. It's also possible that a rider loses their ticket before they get to use it. For example, tickets might be redeemed in the order:
`1, 2, 3, 5, 6, 7, 8, 4, 9, 10`

Your tasks:

1. Design a data structure that allows the machine to efficiently keep track of which tickets have been redeemed.
2. Describe how the data structure is modified when a new ticket is used ("redeem ticket").
3. Describe how the data structure is used to determine if a given ticket has been used before ("check ticket").

Correctness requirements:

- Your system MUST NEVER say a ticket has been used when it hasn't been, or hasn't been used when it has been.

Space requirements:

- Let N be the number of tickets which have been redeemed.
- Your data structure must have a worst-case memory usage of $\Theta(N)$.
- Based on the usage pattern described above, however, your data structure MUST use significantly less than linear memory.

Runtime requirements:

- The runtime of the 'redeem ticket' and 'check ticket' operations should be linearly proportional to memory usage.
- Note that these operations do not need to be constant time. We're saving memory at the expense of doing more computation.
- A naive solution would be to use a HashSet that keeps track of which tickets have been redeemed. However, the memory usage of this system is $\Theta(N)$ even with the usage pattern above, which is unacceptable.
- You may assume that `hashcode()` and `equals()` take constant time.

Name: Matthew Sit

Provide your solution on this page. **Indicate clearly which sections of your response correspond to which task number. Neatly cross out or erase anything that you do not wish to be graded.** You may write your answer in plain English and/or pseudocode.

Solution A:
Keeping track of highest ticket redeemed, and set/list of tickets which have been skipped:
e.g. tickets up to 40 have been redeemed, except 11 and 36.

To check a ticket, see if it is smaller than the highest ticket redeemed and isn't in the set/list of unredeemed tickets.

To redeem a new ticket:

- If it is in the set/list of unredeemed ticket, remove it.
- Otherwise, increase the highest ticket redeemed counter, and add all tickets in-between to the set/list of unredeemed tickets.

Solution B:
Keep a list of which intervals of tickets that have been redeemed:
e.g. [1-10, 12-35, 37-40] means all tickets up to 40 have been redeemed, except 11 and 36.

To check a ticket, iterate through the intervals to see if the ticket is contained in one of the intervals.

To redeem a new ticket, iterate through the intervals, extending an interval, creating a new interval, or merging two intervals as necessary.

Common incorrect solutions involve:

- Trying to track which tickets have been sold (the ticket checker does not have this information, since having this information would already use linear space before we even begin solving the problem).
- Naive solution tracking which tickets have been redeemed.
- Trying to keep track of last X tickets (e.g. 10) – fails if tickets get lost or skipped.
- Keeping track of the highest contiguous number used and a set of all of the tickets used after that – if ticket #2 is lost (for example), this degrades to $\Theta(N)$.