UC Berkeley – Computer Science
CS61B: Data Structures

Final, Spring 2019

This test has 12 questions across 16 pages worth a total of 800 points, and is to be completed in 170 minutes. The exam is closed book, except that you are allowed to use three double sided written cheat sheets (front and back). No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. **Write the statement out 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."

Signature: _____

#	Points	#	Points
0	1	7	84
1	98	8	68
2	43	9	58
3	62	10	73
4	51	11	83
5	50	12	130
6	0		
		TOTAL	800

Name: _____

SID: _____

GitHub Account # : sp19-s_____

Person to Left's # : sp19-s_____

Person to Right's #: sp19-s_____

Exam Room: _____

Tips:

- There may be partial credit for incomplete answers. Write as much of the solution as you can, but bear in mind that we may deduct points if your answers are much more complicated than necessary.
- There are a lot of problems on this exam. **Work through the ones 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.**
- Unless otherwise stated, all given code on this exam should compile. All code has been compiled and executed before printing, but in the unlikely event that we do happen to 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.'**
- ○ indicates that only one circle should be filled in.
- ☐ indicates that more than one box may be filled in.
- For answers which involve filling in a ○ or ☐, please fill in the shape completely.

Optional. Mark along the line to show your feelings on the spectrum between ☹ and ☺.

Before exam: [☹_____☺].
After exam: [☹_____☺].

- There is a separate reference sheet with data structures you can use throughout the test.

0. So it begins (1 point). Write your name and ID on the front page. Write the exam room. Write the IDs of your neighbors. Write the given statement and sign. Write your login in the corner of every page. Enjoy your free point ☺.

1. Sorting Cornucopia.

a) Suppose we want to in-place heap sort the array `[9, 1, 1, 3, 5, 5, 6, 8]` so that it is in ascending order.

i) **(14 points).** Give the result after **bottom-up heapifying** the array so that it is a **max heap**. If you don't remember what bottom up heapification is, you may give any valid max-heap ordering for this array for partial credit.

9	8	6	3	5	5	1	1

ii) **(12 points).** Show the array after one "remove largest" operation (i.e. after one item has been deleted during the in-place heapsort process). Your answer will be graded based on your answer to part i above.

8	5	6	3	1	5	1	9

b) **(9 points).** Suppose we use merge sort instead to sort `[9, 1, 1, 3, 5, 5, 6, 8]`. Give the two arrays that will be merged by the final step of merge sort. In case of a tie, assume the merge operation uses the left item.

1	1	3	9

5	5	6	8

c) **(10 points).** One approach to sorting is to insert all of our items into a `TreeSet<Integer>`, then to iterate over the `TreeSet`, taking advantage of the fact that iteration is an in-order traversal of the tree. Would this approach yield the correct answer for `[9, 1, 1, 3, 5, 5, 6, 8]`? If not, explain why not.

○ Yes, it will work. ● No it will not work because: Duplicates will disappear in the set

d) **(9 points).** For this specific array `[9, 1, 1, 3, 5, 5, 6, 8]`, which sort do you think will be fastest in nanoseconds? Why? Assume all sorts are as described in lecture. Assume Quicksort uses Tony Hoare's partitioning scheme and shuffling.

● Insertion sort ○ Selection sort ○ Merge sort ○ Quick sort ○ Heap sort

Why: This is a small array and also has linear number of inversions

e) **(14 points).** Suppose we partition the array `[4, 5, 1, 3, 7, 2, 6]`, using 4 as our pivot. Which of the following are possible outcomes after a single partitioning operation has completed, i.e. which of the following are valid partitions on the value 4? For each row, state whether the result is possible for Tony Hoare's partitioning scheme, possible for some other partitioning algorithm (and not for Tony Hoare's), or impossible for all partitioning schemes.

	Possible using Tony Hoare's partitioning algorithm.	Not possible using Tony Hoare's partitioning algorithm, but possible with some other partitioning algorithm.	Impossible for all partitioning algorithms.
[4 5 1 3 7 2 6]	○	○	●
[1 2 3 4 5 6 7]	○	●	○
[1 7 5 3 4 2 6]	○	○	●
[1 3 4 2 5 7 6]	○	○	●
[1 3 4 2 5 6 7]	○	○	●
[2 1 3 4 5 7 6]	○	●	○
[3 2 1 4 7 5 6]	●	○	○

f) **(9 points).** Suppose we merge the arrays [1, 7, 8, 9] and [2, 3, 4, 5] using the procedure from class. Which of the following are true? State True or False for each.

● True ○ False	The result will be in sorted order.
● True ○ False	1 will be compared to another element exactly once.
○ True ● False	2 will be compared to 3.
○ True ● False	7 total comparisons will be made.

g) **(21 points).** Now suppose we merge the arrays [1 7 8 9] and [A B C D] using the procedure from class, where A, B, C, and D are integers, **not necessarily in order relative to each other.** Which of the following are true? State True, False, or CBD for "Cannot Be Determined".

○ True ○ False ● CBD	1 will be compared to another element exactly once.
○ True ● False ○ CBD	A will be compared to B.
○ True ○ False ● CBD	7 total comparisons will be made.
● True ○ False ○ CBD	In the result, 1 will be in the correct order relative to A.
○ True ○ False ● CBD	In the result, C will be in the correct order relative to 1.
● True ○ False ○ CBD	In the result, 8 will be in the correct order relative to A.

2. Radix Sorting. Consider the array of integers below:

123	723	175	394	763

a) **(9 points).** Show the output after using counting sort on the rightmost digit only. **Please check your numbers carefully. We will not give partial credit for answers where you accidentally wrote the wrong number**, e.g. if you write "129" you will receive no credit since 129 is not in the list above.

123	723	763	394	175

b) **(9 points).** Show the output after two passes of counting sort in LSD radix sorting. In other words, after we have used counting sort on the rightmost digit, then the middle digit.

123	723	763	175	384

c) **(7 points).** In class, we used counting sort as our subroutine for radix sorting. However, we could also have used another sort, for example, insertion sorting. For example, give the result after insertion sorting ONLY by the leftmost digit of the original array at the top of the page. That is, when we compare two numbers in insertion sort, use only the leftmost digit when comparing the two numbers. **You should start from the array at the top of the page [123, 723, 175, 394, 763].** Assume insertion sort is the same as taught in lecture, e.g. sorting an array of all duplicate items completes in Θ(N) time.

123	175	394	723	763

d) **(18 points).** Suppose we use heapsort as our subroutine for radix sort, yielding two new algorithms LSD-Radix-Heap-Sort and MSD-Radix-Heap-Sort.

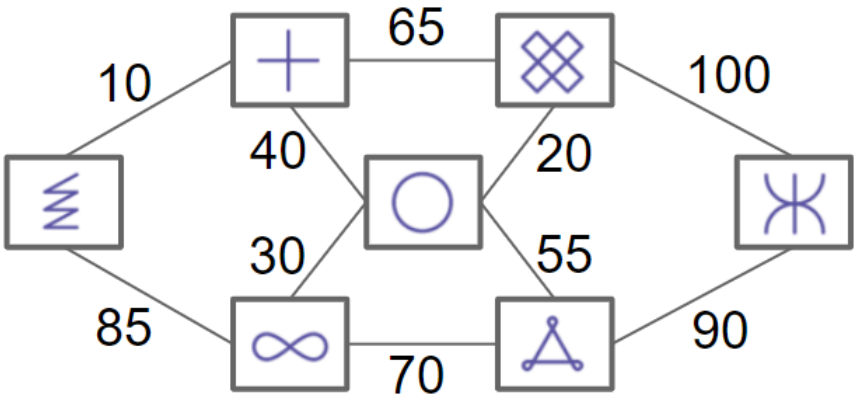
Are these algorithms guaranteed to yield correct results on all inputs, not just the one at the top of this page? If not, briefly explain why not.

○ Yes, LSD-Radix-Heap-Sort will work correctly on all outputs.
● No, LSD-Radix-Heap-Sort can fail because: Heap sort is not stable

● Yes, MSD-Radix-Heap-Sort will work correctly on all outputs.
○ No, MSD-Radix-Heap-Sort can fail because: _____

3. MSTs, SPTs.

Consider the graph below, which we will call G. Throughout this problem, **refer to each edge by its weight**, e.g. 100 refers to the edge at the top right of the image.



a) **(14 points).** Give the edges of the minimum spanning tree for G in the order that they'd be added by **Prim's algorithm**, starting from the vertex with a circle (in the middle). As noted above, **refer to each edge by its weight**, e.g. 100 refers to the edge at the top right of the image. You may not need all blanks.

20 30 40 10 55 90

b) **(14 points).** Suppose we find the shortest paths (SPT) tree starting from the circle vertex in graph G. Is this SPT also the MST of G? If yes, select yes. If no, give an example of an edge that is in the circle vertex's SPT but not in the minimum spanning tree.

○ Yes ● No, edge _100_ is in the SPT for the circle vertex, but not in the MST of G.

c) **(14 points).** Suppose we subtract 1000 from every edge weight in the graph above, yielding a new graph G'. Do the MSTs for G and G' contain the exact same set of edges? Do not worry about edge order. Note that an edge is defined by the vertices it connects, not its weight, so you can't just say "no" simply because the weights are all different.

● Yes, they have the same MST
○ No, edge _____ is in the MST for the original graph G, but the corresponding edge with 1000 subtracted is not in the MST for G'.

d) **(10 points).** True or False: In a weighted undirected graph with non-negative weights, the smallest edge touching a given vertex is always in the MST.

● True ○ False

e) **(10 points).** True or False: In a weighted directed graph with non-negative weights, the smallest edge leaving a given vertex is always in the SPT for that vertex.

● True ○ False

4. Binary Search Tree Removal Redux. On midterm 2, you wrote a `deleteMin` method for a binary search tree, shown below. Also shown is the definition for the **`TreeNode`** class.

```
/** Deletes the smallest item from a BST rooted at x. Returns the
  * root of the BST after deletion. */
private TreeNode deleteMin(TreeNode x) {
    if (x.left == null) { return x.right; }
    x.left = deleteMin(x.left);
    return x;
}
public class TreeNode {
    public TreeNode left;
    public TreeNode right;
    public int item;
    ...
}
```

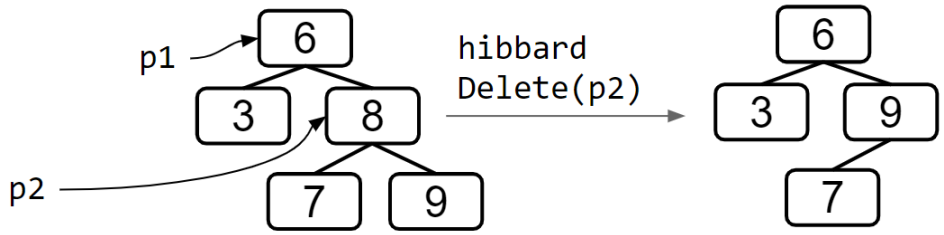
a) **(18 points).** Complete the `min` method for a binary search tree as described below.

```
/* Returns the TreeNode with smallest item in a BST rooted at x. */
public TreeNode min(TreeNode x) {
    if (x == null) { return null; }
    if (x.left == null) { return x; }
    return min(x.left);
}
```

b) **(33 points).** Complete the `hibbardDelete` method below. You may use (but might not need) `min` and `deleteMin` from above. You can assume `min` and `deleteMin` are implemented correctly, regardless of your answer from previous parts.

```
/** This method deletes the root item in a BST rooted at x.
 * Assumes x has two non-null children.
 * Uses the Hibbard Deletion process from class, picking the successor. */
public void hibbardDelete(TreeNode x) {
    TreeNode successor = min(x.right);
    x.right = deleteMin(x.right);
    x.item = successor.item;
}
```

For example, if we call `hibbardDelete(p2)` on the figure below, we'd get the result shown. Note the values of the `p1` and `p2` variables are not shown after the operation. Hint: What happens if we delete p1?


p1
6
3
8
p2
7
9
hibbard
Delete(p2)
6
3
9
7

5. (50 points). Implementation Details.

For each of the pairs of choices below, give the **primary reason** that we prefer the first choice over the second. **There may be more than one answer that is correct – if so, choose the best one**.

When storing a heap in an array (tree representation 3), placing the first item at **index 1** instead of **index 0**.	○ Guarantees correctness ○ Improves runtime ○ Reduce memory use ● Simplifies code
In Dijkstra's algorithm: visiting vertices in order of **increasing distance** from source instead of **decreasing distance**.	● Guarantees correctness ○ Improves runtime ○ Reduce memory use ○ Simplifies code
Considering the **left child** before the **right child** if the query point is to the left of the current node during a nearest query on a KdTree implementation that has implemented all of our pruning rules.	● Guarantees correctness ● Improves runtime ○ Reduce memory use ○ Simplifies code An argument can be made for both so either was accepted. Correctness was original intention.
Shuffling the input before beginning Quicksort instead of **not shuffling**.	○ Guarantees correctness ● Improves runtime ○ Reduce memory use ○ Simplifies code
Rotating after inserting a new item into a BST, instead of **not rotating**.	○ Guarantees correctness ● Improves runtime ○ Reduce memory use ○ Simplifies code
Resizing a hash table after a removal operation, instead of **not resizing**.	○ Guarantees correctness ○ Improves runtime ● Reduce memory use ○ Simplifies code
Using **A*** instead of **Dijkstra's algorithm** to find the shortest path from s to t.	○ Guarantees correctness ● Improves runtime ○ Reduce memory use ○ Simplifies code
Implementing a priority queue using a **heap** instead of an **ordered array**.	○ Guarantees correctness ● Improves runtime ○ Reduce memory use ○ Simplifies code
Using double links (`next` and `prev`) in every node instead of only **single links** (`next`) in a `LinkedListDeque` implementation.	○ Guarantees correctness ● Improves runtime ○ Reduce memory use ○ Simplifies code
Using **sentinel nodes** in a `LinkedListDeque` implementation instead of **using `first` and `last` pointers that point directly to the nodes containing the first and last items in the deque**.	○ Guarantees correctness ○ Improves runtime ○ Reduce memory use ● Simplifies code

6. (0 points). PNH. The standard musical scale is known as twelve-tone equal temperament. In this system, there are 12 evenly spaced musical notes that divide up an octave. What spooky sounding musical scale has instead 13 evenly spaced musical notes that divide up a tritave?

The Bohlen-Pierce Scale. You could have found the answer somewhere on this exam!

Note: A tritave is like an octave, but it's 3x the frequency instead of 2x. For example, 440 hz and 1320 hz are one tritave apart. Note #2: If you want to hear a listenable song using this scale, check out "Orbital" by Sevish.

7. **DisjointSets**. The version of `DisjointSets` that we created in lecture assumed that the items were numbered from 0 to N – 1. Consider the alternate definition for `DisjointSets` below, where items can have any arbitrary integer value, and items can be added at any time.

```
public interface DisjointSetsSp19F {
    public void add(int p);         // creates a new, disconnected item
    public Set<Integer> vertices();
    public void connect(int p, int q);
    public boolean isConnected(int p, int q);
}
```

a) **(38 points).** Complete the implementation of `QuickUnionDisjointSets` below, not including `isConnected`. For full credit, your implementation must be of `QuickUnion`, not `QuickFind`! That is, asymptotic runtimes of all methods should be at least as good as in unweighted `QuickUnion`.

```
public class QuickUnionSp19F implements DisjointSetsSp19F {
    HashMap<Integer, Integer> itemToParent;
    // ^^^ note you may have ONLY ONE INSTANCE VARIABLE! ^^^
    public QuickUnionSp19F() {
        itemToParent = new HashMap<>();
    }
    public void add(int p) { // throws exception if p already exists
        if (vertexExists(p)) { throw new IllegalArgumentException(); }
        itemToParent.put(p, p);
    }
    private boolean vertexExists(int p) { // true if p has been added
        return itemToParent.containsKey(p);
    }
    public Set<Integer> vertices() {
        return itemToParent.keySet();
    }
    public void connect(int p, int q) {
        if (!vertexExists(p) || !vertexExists(q)) {throw new IllegalArgumentException();}
        int pRoot = root(p);
        int qRoot = root(q);
        itemToParent.put(qRoot, pRoot);
    }
    private int root(int p) {
        if (!vertexExists(p)) { throw new IllegalArgumentException(); }
        int parent = itemToParent.get(p);
        if (parent == p) { return p; }
        return root(parent);
    }
    public boolean isConnected(int p, int q) { // implementation not shown }
}
```

b) **(46 points).** Suppose we create a class called **`ConnectedComponentIter`** that lets us iterate over all items connected to a given item. For example, the code below would print out the numbers 30, 50, 60, and 80, **not necessarily in that order.**

```
    public static void main(String[] args) {
        DisjointSetsSp19F ds = new QuickUnionSp19F();
        for (int i = 0; i < 100; i += 1) {
            ds.add(i * 10);
        }
        ds.connect(50, 30); ds.connect(50, 60);
        ds.connect(60, 80); ds.connect(10, 20);
        ConnectedComponentIter cci = new ConnectedComponentIter(ds, 50);
        for (int i : cci) {
            System.out.println(i);
        }
    }
```

Fill in the **`ConnectedComponentIter`** class so that it behaves as expected on all possible inputs, including the example above. There is no runtime requirement. **Your code should work on any `DisjointSetsSp19F` implementation, not just a `QuickUnionSp19F`.**

```
public class ConnectedComponentIter implements Iterable<Integer> {
    public List<Integer> items;

    public ConnectedComponentIter(DisjointSetsSp19F ds, int p) {
        items = new ArrayList<Integer>();
        for (int q : ds.vertices()) {
           if (ds.isConnected(p, q)) {
                items.add(q);
           }
        }
    }

    public Iterator<Integer> iterator() {
        return items.iterator();
    }
}
```

8. (68 points). Asymptotics. Give the **worst case** runtime of the following functions in Θ notation in terms of N.

Θ(NlogN)
```
public int f1(int N) {
    if (N <= 1) { return N; }
    int sum = 0;
    sum += f1(N/2);
    for (int i = 0; i < N/2; i += 1) {
        System.out.println("Bohlen");
    }
    sum += f1(N/3);
    for (int i = 0; i < N/2; i += 1) {
        System.out.println("Pierce");
    }
    sum += f1(N/6);
    return sum;
}
```

Θ(N)
```
public void f2(ArrayList<String> A, String s) {
    int N = A.length();
    int sLocation = A.indexOf(s); // returns -1 if not in A
    if (sLocation > -1) {
       String temp = A.get(0);
       A.get(0) = A.get(sLocation);
       A.set(sLocation, temp);
    }
} // Note: ArrayLists are similar to our AList class,
  // They have one instance variable: An array.
```

For `f3` below, do not assume anything about the distribution of items in the hash table! Assume buckets are stored as linked lists of items. Assume that `keySet` creation is O(N), and that `next()` and `hasNext()` are constant time. Assume that a `TreeSet` is implemented using an LLRB.

Θ(N^2)
```
public int f3(TreeSet<Bloop> A, HashSet<Bloop> B) {
    int N = a.size() + b.size();
    int inBoth = 0;
    for (Bloop a : A.keySet()) {
        if (B.containsKey(a)) {
            inBoth += 1;
        }
    }
    return inBoth;
}
```

Θ(logN * logN)

```
public void f4(int N) {
    int M = 0;
    for (int i = 1; i < N; i *= 2) {
        M += 1;
    }
    f4helper(M);
}

public void f4helper(int Q) {
    if (Q <= 0) { return; }
    for (int i = 0; i < Q; i += 1) {
        System.out.println(i);
    }
    f4helper(Q - 1);
}
```

9. Reductions and Sorting.

a) A 61B student proposes a new algorithm "Graph Path Sort" for sorting an array of N **unique** integers. It works by reducing sorting to the Directed Acyclic Graphs Shortest Paths Tree (DAGSPT) algorithm. Recall that DAGSPT first computes a topological sort, then visits vertices in topological order.

Specifically, in this algorithm, the student creates a graph corresponding to the array as follows:

- Create a vertex for each number.
- For each pair of integers `v` and w, if `v` < `w`, create a directed edge from `v` to `w` with weight `1`.

The student then uses the DAGSPT algorithm with the smallest number as the source. The sorted array is simply the path returned by the DAGSPT from the smallest number to the largest.

i) **(14 points).** What is the overall runtime of Graph Path Sort in terms of N?

$\Theta(N^2)$

ii. **(16 points).** Unfortunately, the student's algorithm is broken. Give the output of the student algorithm when run on the input [5, 1, 7, 9, 3].

[1, 9]

iii. **(28 points).** Describe a simple fix to the student's algorithm that would make it work correctly on all arrays of unique integers. Your answer will be graded on correctness and conciseness.

Set all the edge weights to -1. DAGSPT will find the path with the most edges which will be the full topological ordering of our vertices. This ends up being the sorted order.

10. A New Way. The hash table we described in class computes bucket numbers for an object x as follows: First we compute h, the hash code of the object. We then reduce h by modding by the number of buckets M, yielding a bucket number b. That is, b = x.hashCode() mod M.

Naturally, multiple items may have the same bucket number b, which we called a collision. In class we resolved collisions with a technique we called "separate chaining". That is, if multiple items have the same bucket number b, they are all stored in a linked list together.

One alternate approach to collision that we did not discuss in class is "linear probing". In this approach, we do not use linked lists. Instead, if bucket b is already used, we use bucket (h + 1) mod M. If that bucket is already used, we use bucket (h + 2) mod M. If that is also used, we try bucket (h + 3) mod M, and so forth.

a) **(18 points).** Suppose we add the integers 93, 21, 5, 3, 13, 25, 83, where **the hash code of a number is given by its least significant digit, i.e. h(93) = 3**. Draw the contents of the hash table below, where M = 8. The first one has been done for you. **Represent null entries with the string `null`**. Assume we start from an empty hash table, i.e. **where all entries are initially `null`**.

Index	0	1	2	3	4	5	6	7
Key	83	21	null	93	3	5	13	25

b) **(22 points).** In a linear probing hash table, the `contains(x)` method works by calling `look(b)`, where `b` is the bucket number of `x`. `look(b)` is defined in pseudocode as follows:

- `if bucket[b] == null, return false`
- `if bucket[b].equals(x), return true`
- `return look(b + 1)`

Note that this procedure results in an infinite loop if the hash table is full and we call contains on an item that is not in the hash table. Thus, our linear probing hash table is never allowed to be totally full. In other words, its load factor is always less than 1.

Provide an ordered sequence of 4 add operations such that `contains(0)` requires **two** calls to `equals` and `contains(1)` requires **three** calls to `equals`. Assume we start from an **empty hash table**, M = 8 and we have the same hash code function as in part a.

`add(10)` `add(0)` `add(21)` `add(1)`

c) **(22 points).** We have not described how deletion in a linear probing hash table would work, so here's your chance to figure it out. Suppose we delete 5 from the linear probing hash table from part a. Draw the contents of the hash table below after deletion has been completed. Your answer must be such that future calls to add, contains, and delete will also be correct. You cannot get points for this if your answer to part a was wrong. **Represent null entries with the string `null`**.

Index	0	1	2	3	4	5	6	7
Key	null	21	null	93	3	13	25	83

d) **(11 points).** Give the best and worst case runtimes for the deletion method in theta notation in terms of N, the number of items, and M the number of buckets.

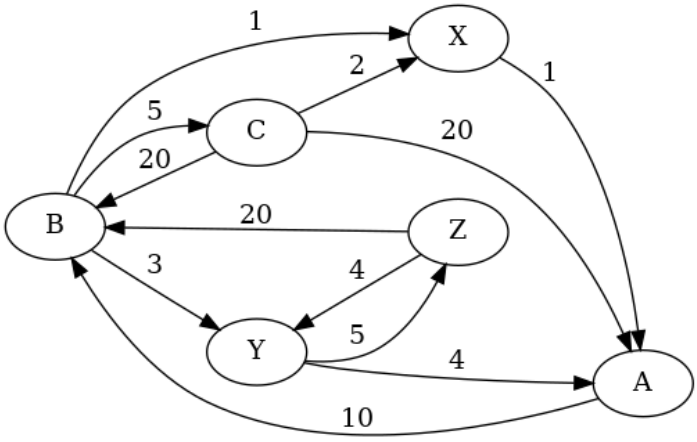
Best: $\Theta(1)$, Worst: $\Theta(N)$

11. Ψcle. First, some terminology: A simple cycle is a path where no vertex is used more than once, other than the first and last vertex. For example, in the graph shown B → C → A→ B would be a simple cycle, but B → Y → Z → B → C → B would not.

Given a weighted directed graph G, suppose we want to find WSSPsiCC(G, Ψ), the **weight** of the shortest simple "psi compliant" cycle in G.

We say that a cycle is "psi compliant" if it includes **at most one** vertex from a set of vertices Ψ.

We say that a cycle is the "shortest psi compliant" cycle if it has the minimum total weight among all psi-compliant cycles.



a) (14 points). Give the WSSPsiCC(G, Ψ) for the graph shown if Ψ = {A, B, X}. That is, the cycle represented by your answer may include A, may include B, may include X, or may not include any of them. However, it cannot include both A and B, both A and X, or both B and X. It also cannot include all three. **Your answer should be a single integer.**

WSSPsiCC(G, Ψ = {A, B, X}) = 9 (cycle is Y->Z->Y)

b) (14 points). Give the WSSPsiCC(G, Ψ) for the graph shown if Ψ = {X, Y, Z}.

WSSPsiCC(G, Ψ = {X, Y, Z}) = 12 (cycle is B->X->A->B)

c) (55 points). Give an algorithm for WSSPsiCC(G, Ψ) that runs in O(P V E log V) time where P is the number of vertices in Ψ, E is the number of edges in the graph, and V is the number of vertices in the graph. For example, in the subproblems above (part a and b), V = 6, E = 12, P = 3.

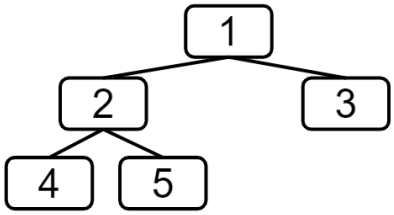
First we define a procedure WSSC(G, v) which finds the weight of the shortest simple cycle in G which starts with v. We can do this by running Dijkstra's from v to find the shortest path to every other vertex. Then we iterate over every other vertex and check if there is an edge between that vertex and v. If there is an edge, then there is a cycle with weight equal to the shortest path length plus the edge weight between the vertex and v. We can then return the min of all the cycles found. This takes a total of O(E*logV) time since Dijkstra's takes that long.

Then, we define a procedure WSSCinG(G) which finds the weight of the shortest simple cycle in G starting anywhere. We can do this just by running WSSC(G, v) for every vertex v in G and returning the min of these calls. This takes a total of O(V*E*logV) since we are running WSSC(G, v) a total of V times.

Then, we can create P + 1 copies of our graph. The first of these copies has every vertex in Ψ removed. The other P copies has all but one vertex from Ψ removed. This takes O(P(E + V)) time. By definition, any cycles in these copies are psi-compliant.

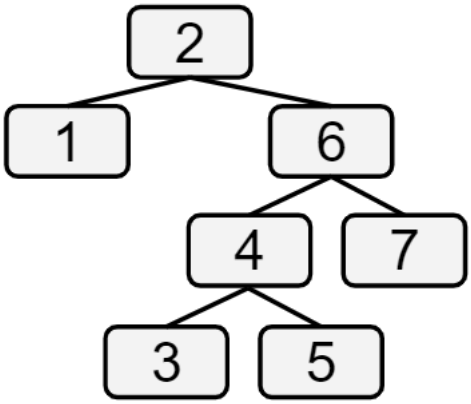
Finally, we can just run WSSCinG(G) on every copy of our graph and return the min of these calls. This takes a total of O(P*V*E*logV + P*(E+V)) = O(P*V*E*logV) time since we are running WSSCinG(G) P times.

12. Tree Representation 4. When studying heaps, we saw that a complete tree can be very simply represented by a sequence of values corresponding to the level order of the tree, which we called representation 3. For example `1, 2, 3, 4, 5` represents the tree below. Since we know the tree is complete, there is no ambiguity about where the items go.



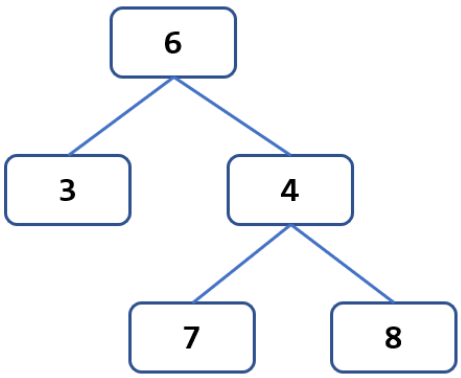
This idea can also work for trees that are not complete. For example, we can also represent trees where every node has either 0 or 2 children in a similar manner. We will call **this tree representation 4**. In this approach, we again represent the tree by its level order traversal, but also precede each item with a boolean, indicating whether it is a leaf or not. E.g., the tree above would be `F, 1, F, 2, T, 3, T, 4, T, 5`.

a) **(12 points).** For the tree to the right, give tree representation 4.



F	2	T	1	F	6	F	4	T	7	T	3	T	5

b) **(12 points).** For the sequence `F, 6, T, 3, F, 4, T, 7, T, 8` draw the corresponding tree below.



c) **(18 points).** For a tree representation 4 sequence that is 18 items long (9 numbers, 9 booleans), what is the minimum and maximum number of possible True values?

Minimum: 5 Maximum: 5

d) **(55 points).** Suppose that we have a tree stored as a recursive data structure of **`Node`** objects (defined below). Fill in `writeTree` so that it writes the tree to a file in representation 4. **Assume that all nodes**

have either zero or 2 non-null children. You may assume the tree is well formed (no loops or other errors).

To write to a file you'll use the provided `Out` object. `out.println(boolean b)` writes the given boolean to the file, and `out.println(int i)`. You should not use any other methods from the `Out` class.

As with any problem on this exam, you are welcome to instantiate whatever data structures that you want from the reference sheet. **You may not create any helper methods.**

Note: This problem is pretty hard. If you're stuck, come back later!

```
public class Rep4Node {
    public Rep4Node left, right;
    public int item;
    public Rep4Node(int i, Rep4Node l,Rep4Node r) {
        left = l; right = r; item = i;             }

    /** Writes this tree to the given file in representation 4. */
    public void writeTree(Out out) {
        Queue<Rep4Node> q = new Queue<>();
        q.enqueue(this);
        while (!q.isEmpty()) {
           Rep4Node p = q.dequeue();
           if (p.left == null && p.right == null) {
                out.println(true);
                out.println(p.item);
           } else {
                out.println(false);
                out.println(p.item);
                q.enqueue(p.left);
                q.enqueue(p.right);
           }
         }
    }
```

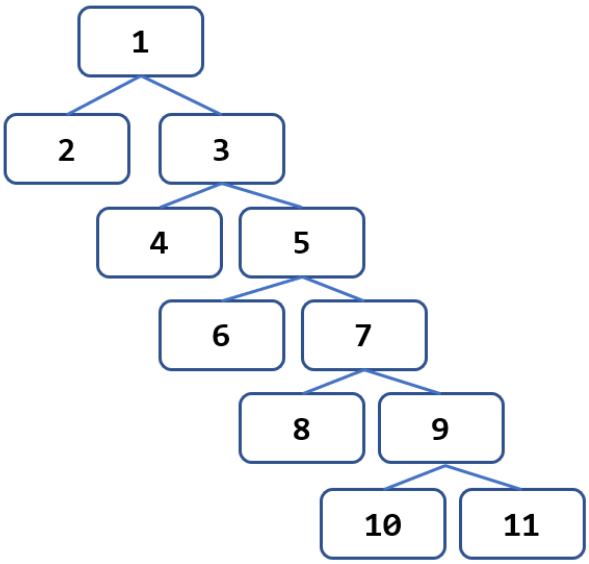
Reminder: for booleans `a` and `b`, `a && b` returns the result of `a` AND `b`

e) (**33 points).** An older version of the exam also had you write the `readTree` method, assuming the `in.readInt()` method gets the next integer and `in.readBoolean()` gets the next boolean. It was a cool problem, and the solution one of us came up with was really elegant, shown below.

```
public static TreeNode readTree(In in) {
    boolean isLeaf = in.readBoolean();
    if (isLeaf) {
        return new TreeNode(in.readInt(), null, null);
    } else {
        return new TreeNode(in.readInt(), readTree(in), readTree(in));
    }
}
```

To test our solution, we wrote a big nice JUnit test that used `writeTree` from the previous page, and then read the tree back in using `readTree` above. The test passed and we were feeling good. Unfortunately, the `readTree` method above is fatally flawed and will never be able to read in a tree representation 4 without starting over completely. Our test just happened to miss the bug. Your job in this problem is to recreate our broken test.

Specifically, **draw a tree with 11 nodes with the following property: If you write this tree out to a file using `writeTree`, then read the file using the buggy `readTree` method above, you get back the correct tree.** You do not need to have completed part d to do this problem. You should assume that `writeTree` correctly outputs the appropriate Representation 4 sequence for your tree.



… and just like that, you've earned your wolf shirt. It's over.

Thanks for being in 61B. Have a fruitful and cool future.