Final Exam Solutions

1 Sometimes Sort of Sorted (8 points)

Let n be some large integer, $k < \log_2 n$. For each of the input arrays described below, explain which sorting algorithm you would use to sort the input array the fastest and why you chose this sorting algorithm. Make sure to state any assumptions you make about the implementation of your chosen sorting algorithm. Also specify the big-Oh running time for each of your algorithms in terms of k and n. Your big-Oh bounds should be simplified and as tight as possible.

(a) Input: An array of n `Comparable` objects in completely random order

Sorts: Quicksort, merge sort, heap sort, tree sort
Runtime: $O(n \log n)$
Explanation: Merge sort and heap sort are always $O(n \log n)$. For quicksort, we can easily choose a good pivot for randomly ordered inputs. For tree sort, the resulting tree will be fairly balanced in the average case.

Comments: You needed to use a comparison-based sort because input contains `Comparable` objects. Bubble, insertion, and insertion sort are inefficient compared to the four listed above. Runtime should not include k.

(b) Input: An array of n `Comparable` objects that is sorted except for k randomly located elements that are out of place (that is, the list without these k elements would be completely sorted)

Sort: Insertion sort
Runtime: $O(nk)$
Explanation: For the $n-k$ sorted elements, insertion sort only needs 1 comparison to check that it is in the correct location (larger than the last element in the sorted section). The remaining k out-of-place elements could be located anywhere in the sorted section. In the worst case, they would be inserted at the beginning of the sorted section, which means there are $O(n)$ comparisons in the worst-case for these k elements. This leads to an overall runtime of $O(nk+n)$, which simplifies to $O(nk)$.

Comments: It was a common error to say the runtime was $O(n)$ or $O(n^2)$. We gave partial credit for both of these answers. $O(n)$ was incorrect because it underestimated the number of comparisons required for the out-of-place elements. $O(n^2)$ was incorrect because it ignored the fact that only 1 comparison is needed for already sorted elements.
Also, it is incorrect to equate k with $\log_2 n$. It was only given that $k < \log_2 n$.

(c) Input: An array of n `Comparable` objects that is sorted except for k randomly located pairs of adjacent elements that have been swapped (each element is part of at most one pair).

Sort Option 1: Bubble sort with optimization
Runtime 1: $O(n)$ or $O(n+k)$
Explanation 1: It was necessary to mention the optimization for bubble sort, which involves stopping after a complete iteration of no swaps occurring. In order words, optimized bubble sort stops once the list is sorted, not after it has run through n iterations for a runtime of $O(n^2)$.

It takes one iteration of bubble sort to swap the k randomly reversed pairs of adjacent elements then another iteration before optimized bubble sort stops due to no swaps. If you count each swap as taking $O(1)$ time, the total runtime is $O(2n+k)$, which simplifies to $O(n)$.

Sort Option 2: Insertion sort
Runtime 2: $O(n)$ or $O(n+k)$
Explanation 2: Insertion sort requires 1 comparison for the $n-k$ sorted elements then requires 2 comparisons for the second element in each of the k pairs. This leads to a runtime of $O(n+k)$, which simplifies to $O(n)$.

(d) Input: An array of n elements where all of the elements are random `int`s between 0 and k

Sort Option 1: Counting sort
Runtime 1: $O(n)$ or $O(n+k)$
Explanation 1: Counting sort involves initializing an array of size k, then going through n elements while incrementing numbers in the array. Recovering the sorted list requires going through k buckets and outputting n numbers. This is a total runtime of $O(2n+2k)$, which simplifies to $O(n+k)$ or $O(n)$.

Comments: Counting sort was the easier sort to explain. Radix sort, explained below, required a more intricate explanation to prove that it had as efficient a runtime as counting sort.

Sort Option 2: Radix sort
Runtime 2: $O(n)$
Explanation 2: Radix sort is $O((n+b)d)$, where b is the number of buckets used and d is the number of digits representing the largest element. (In this case, the largest element was k.) Because the input array is composed of Java `int`s, we can say that b is equal to 2 and d is equal to 32 because Java `int`s are 32-bits long, and each bit can be a 0 or 1. Thus, because b and d are constants, the runtime for radix sort on Java `int`s is $O(n)$.

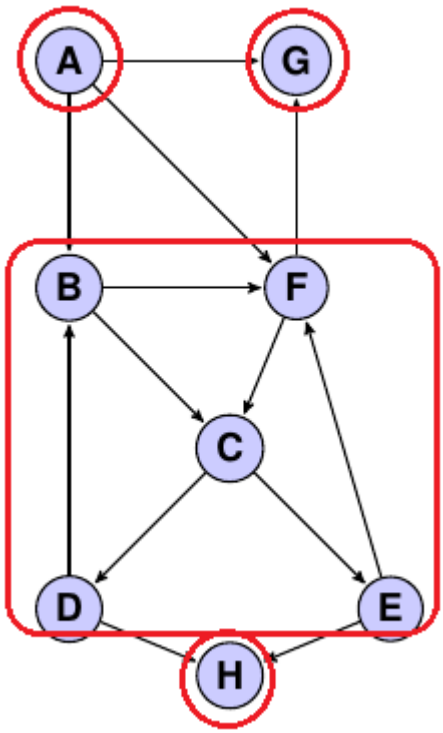
Comments: We also accepted the answer that b was 10 and d was a constant when simplifying the runtime to $O(n)$ based on Java `int`s.
While it is true that radix sort runs in $O(n \log k)$ because the number k can be represented in $\log k$ bits and b would then be a constant, this runtime did not prove that radix sort was as efficient of a sort as counting sort. Thus, we only gave partial credit for this runtime.

Sort Option 3: Bucket sort
Runtime 3: $O(n)$ or $O(n+k)$
Explanation 3: The correctness of bucket sort as an answer depended heavily on the choice of buckets (and how many buckets were used, in particular). Solutions that chose k buckets were quite similar to counting sort and thus had the same runtime as counting sort.

Comments: Solutions that did not explain the choice or number of buckets were docked points because bucket sort actually refers to any sorting algorithm that involves placing elements in buckets. Thus, counting sort, radix sort, and even quicksort and merge sort are all categorized as bucket sorts. Only explanations of bucket sort that were specific about the implementation received full points.

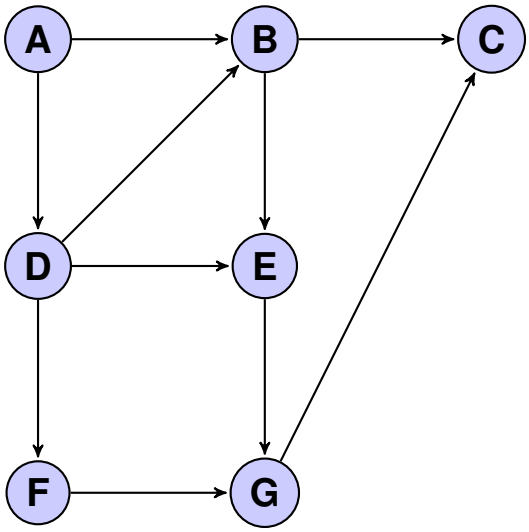
2 The Unsocial Network (4 points)

(a) For each strongly connected component in the directed graph below, draw a circle around all of the vertices in that strongly connected component.



Comments: Many students didn't circle the strongly connected components that consisted of only a single vertex.

(b) Give a topological sort ordering of the vertices in the following directed acyclic graph (in other words, give a linearized ordering of the vertices in the graph).



Potential solution: [A, D, B, F, E, G, C]

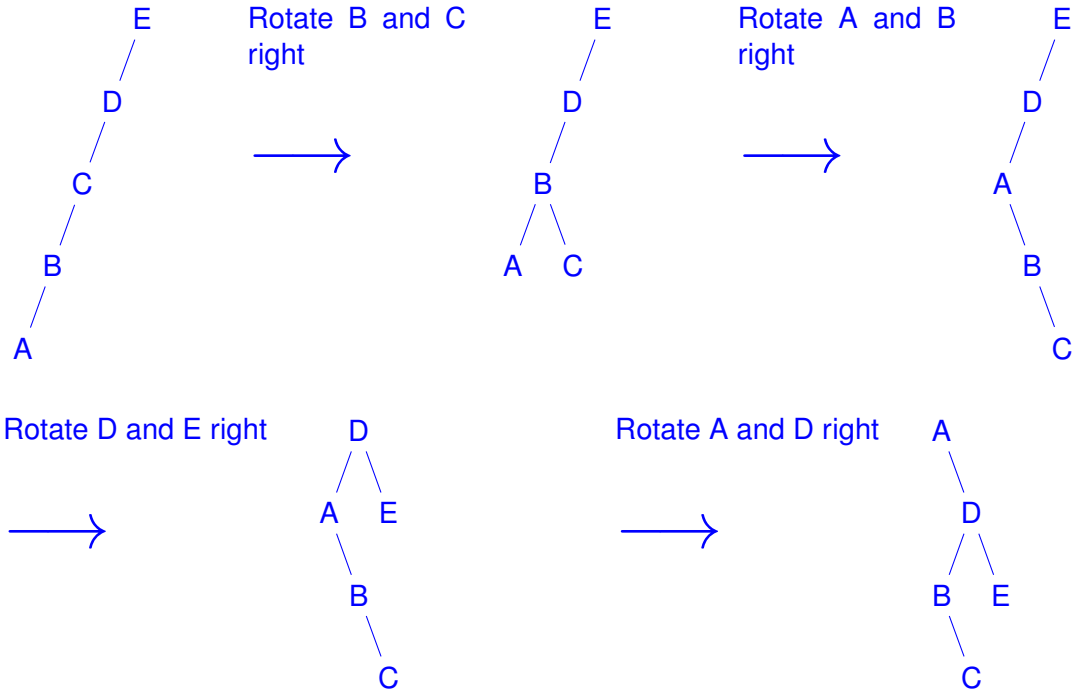
Comments: There were many possible solutions to this problem.

3 Balanced Search Trees (7 points)

For each part below, assume that nodes are ordered alphabetically by their letter (i.e. the value of "A" is less than the value of "B", which is less than the value of "C", etc...)

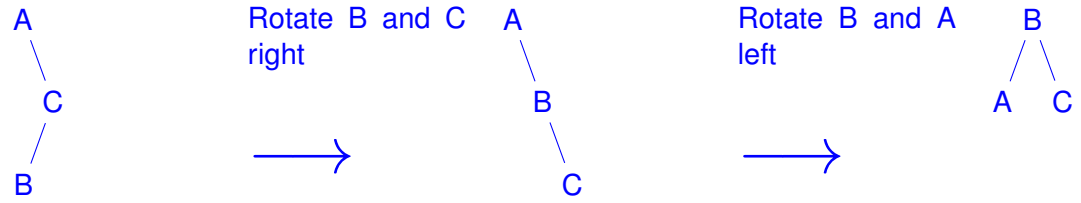
(a) Draw the splay tree that results after calling `find("A")` on the splay tree below.

Solution:



(b) Add a "B" node to the AVL tree below by drawing it below. Then draw the tree that results after the AVL tree balances itself. Show all intermediary steps, if any.

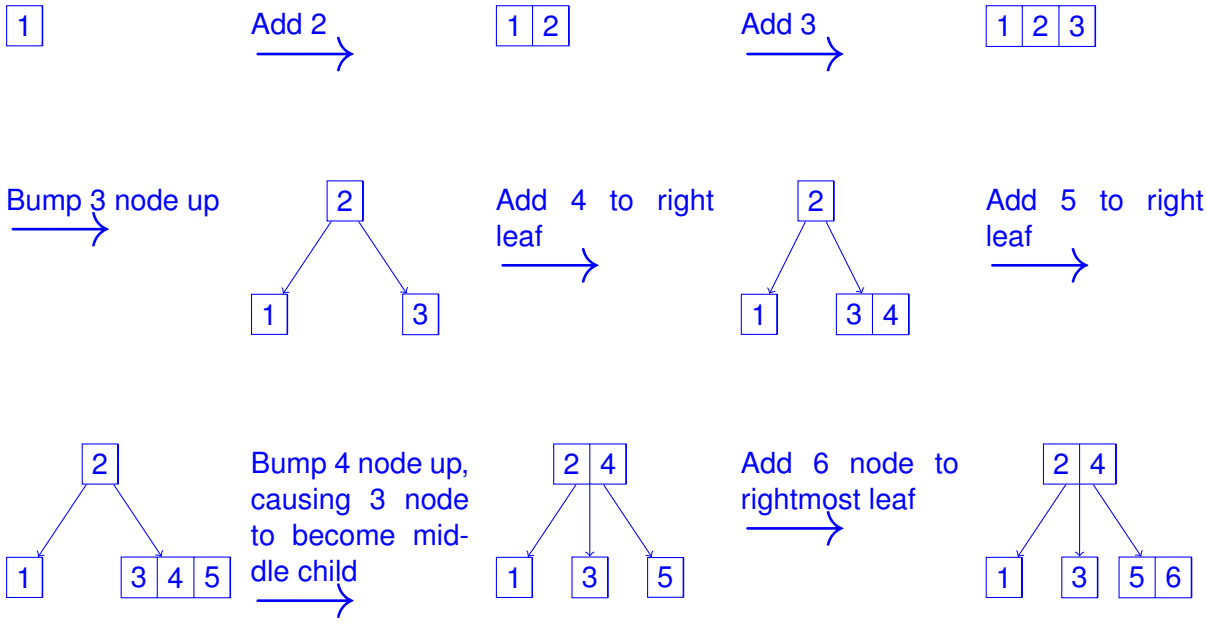
Solution:



(c) Draw the 2-3 tree that results after inserting the following elements in the given order:

1 2 3 4 5 6

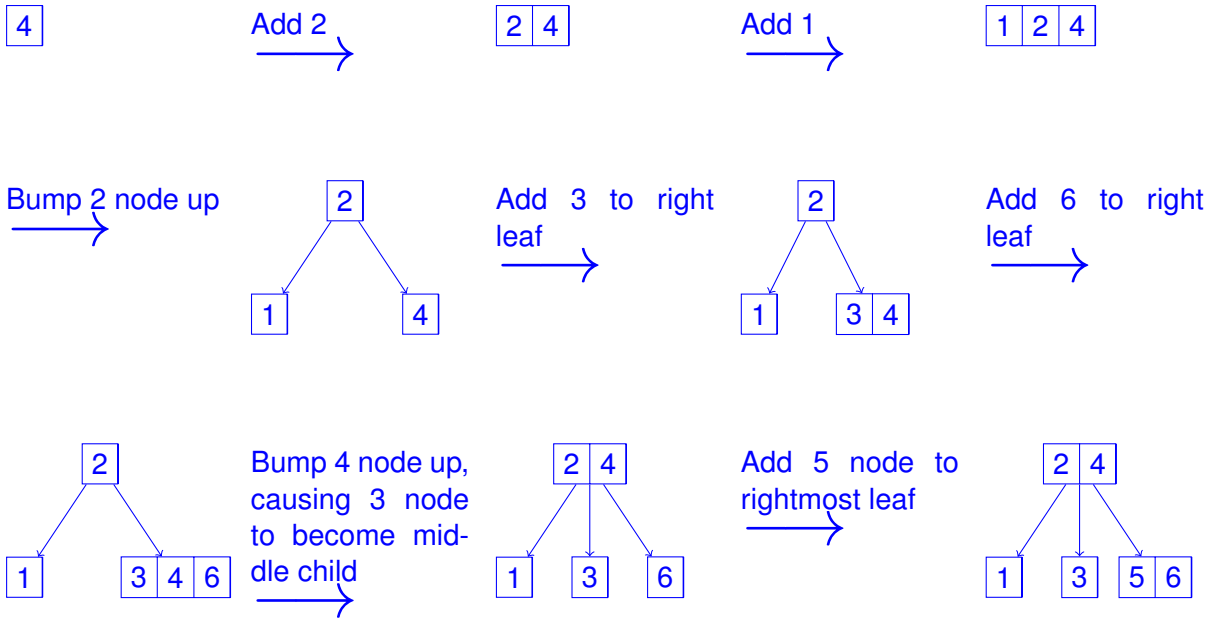
Solution:



(d) Draw the 2-3 tree that results after inserting the following elements in the given order:

4 2 1 3 6 5

Solution:

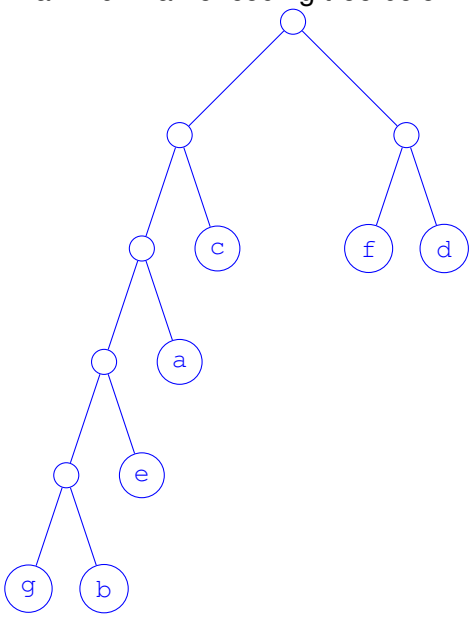


4 Huffman Tree (6 points)

A table of characters and corresponding frequencies is provided below. Next to the table, draw a valid Huffman encoding tree. Then fill out the codemap table with each character's encoding according to your Huffman encoding tree.

Character	Frequency
a	5
b	2
c	8
d	7
e	3
f	7
g	1

Draw Huffman encoding tree below:



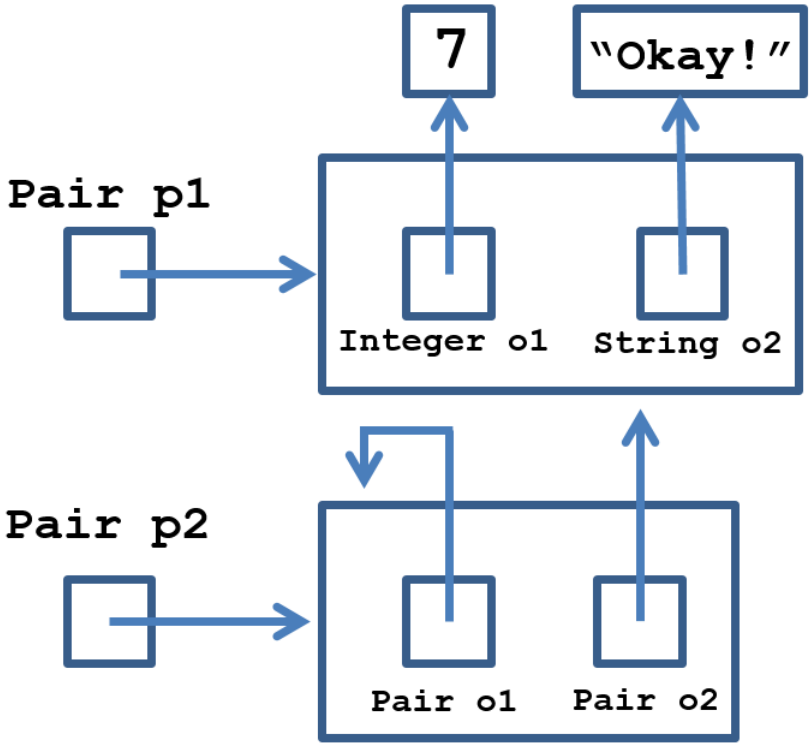
Comments: There were many possible solution trees (one is shown above), but only one way to combine nodes at every step. Thus, any correct solution can be derived from the tree above by switching any node's left and right subtrees.

Fill out the codemap below:

Character	Encoding
a	001
b	00001
c	01
d	11
e	0001
f	10
g	00000

5 `Pair` (5 points)

Write a program, `Pair.java`, that generates the box-and-pointer diagram shown below when run. Your program should include a class with a `main` method. In the diagram below, each object's static type is labeled next to the corresponding variable name. Each object's dynamic type is not shown.



Solution:

```
public class Pair<T1, T2> {
    T1 o1;
    T2 o2;

    public static void main(String[] args) {
        Pair<Integer, String> p1 = new Pair<Integer, String>();
        Pair<Pair, Pair> p2 = new Pair<Pair, Pair>();
        p1.o1 = new Integer(7);
        p1.o2 = "Okay!";
        p2.o1 = p2;
        p2.o2 = p1;
    }
}
```

Comments: Many students didn't use generics, and instead made instance variables have static type `Object` (we only took off one point for this). Another common mistake was to use `p2` before it was initialized (for example, `Pair p2 = new Pair(p1, p2);`)

6 Poorly Named Variables (6 points)

```
1  public class V {
2      private int VVVVV;
3
4      public V(int VVVV) {
5          VVVVV = VVVV;
6      }
7
8      private int VV() {
9          return 0;
10     }
11
12     public double OOOO() {
13         return 0;
14     }
15 }
```

```
1  public class O extends V {
2      private int OOO;
3  }
```

For each of the following methods, explain whether the `O` class would still compile if the method is added to it. If the `O` class wouldn't compile, explain why. The `O` class may have additional methods / constructors. The `V` class does not.

(a) `public O() { }`

This would not compile. In a constructor, if there is no explicit call given to a super constructor, an implicit one is made. Since the implicit call would have been `super()` and class `V` does not have a no-arg constructor, you will get a compile-time error.

Comments: A common error was assuming `V` had a no-arg constructor. However, the default no-arg constructor ceases to exist once you define an explicit constructor.

(b) `void OO() { OOO = VV(); }`

This would not compile. `VV` is a private method and cannot be called in any class outside of `V`.

Comments: Some people thought the error was the lack of a privacy keyword (i.e. `public`, `private`, `protected`). However, lack of a privacy keyword simply means the method has `default` privacy protections. `default` protection means that the method or variable can only be accessed by code within the same class or package.

(c) `public boolean OOOO() { return false; }`

This would not compile. Since the method signature was the same (i.e. same name and same parameters) but the return types were different, Java would be unable to differentiate between the two. One could have defended that this was either a failed override or a failed overload depending on how you explained your answer.

Comments: Many people were confused on the terminology of method signature and method header. A method's signature includes its name and parameters. A method's header includes the privacy modifier and return type in addition to the method signature.

7 Dijkstra Analysis (8 points)

For each part below, all solutions given in terms of big-Oh must be simplified and as tight of an upper bound as possible. $|E|$ is the number of edges in the graph and $|V|$ is the number of vertices in the graph.

(a) Recall that Dijkstra's algorithm finds the shortest paths from some starting vertex s to all other vertices in the graph. In terms of big-Oh, $|E|$, and $|V|$, **how many times** does each of the following priority queue operations get called in one complete run of Dijkstra's algorithm?

i) `enqueue`: $O(|V|)$

ii) `dequeue`: $O(|V|)$

iii) `isEmpty`: $O(|V|)$

iv) `containsKey`: $O(|E|)$

v) `update` (updates a vertex's priority value in the priority queue): $O(|E|)$

(b) In lab, we analyzed the running time of Dijkstra's algorithm using a priority queue implemented with a binary min-heap. Now let's use a priority queue implemented with Java's `HashMap` that maps vertices to their priority values. Assuming that the hash map operations `put`, `get`, and `remove` take constant time, what are the new big-Oh running times (in terms of $|E|$ and $|V|$) of a single call to each of the following operations:

i) `enqueue`: $O(1)$

ii) `dequeue`: $O(|V|)$

iii) `isEmpty`: $O(1)$

iv) `containsKey`: $O(1)$

v) `update`: $O(1)$

Comments: Some students didn't realize that these were the running times of a single call to each operation (as opposed to part (a), which asked for how many times each operation was called). One common mistake was to give the running time of `dequeue` as $O(1)$. This was incorrect because in order to dequeue, the entire `HashMap` needs to be traversed to find the vertex with the lowest priority value. Another common mistake was to give the running time of `isEmpty` as $O(|V|)$. This was incorrect because Java's implementation of `HashMap` (like most implementations) has a variable to keep track of the size of the `HashMap`.

(c) With our changes above, what is the new big-Oh running time of Dijkstra's algorithm (in terms of $|V|$ and $|E|$)? Show your work.

Solution: $|V|*1+|V|*|V|+|V|*1+|E|*1+|E|*1 = 2*|V|+|V|^2+2*|E| \in O(|V|^2+|E|) \in O(|V|^2)$ Multiplying corresponding answers from parts (a) and (b) and summing them gives us an upper bound on the running time. In this case, this is also a tight bound since `dequeue` takes time proportional to the number of keys in the `HashMap`. The first time we call `dequeue` there are $|V|$ keys, then $|V|-1$ keys, and so on. This gives up a running time of $O(|V|^2)$. Finally, since $|E| \leq |V|^2$ for any graph, $O(|V|^2+|E|) \in O(|V|^2)$.

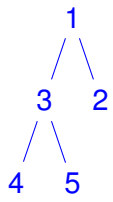
8 Buggy Priority Queue (10 points)

Joe Cool is implementing a priority queue with a binary min-heap using an array list (his code is provided below). However, his pair programming partner tells him that, with this implementation, calls to `dequeue` will not always work as intended.

```
1  import java.util.ArrayList;
2
3  public class MyPriorityQueue {
4
5      private ArrayList<Integer> binMinHeap;
6
7      public MyPriorityQueue() {
8          binMinHeap = new ArrayList<Integer>();
9          binMinHeap.add(null);
10     }
11
12     // Removes and returns item in priority queue with smallest priority
13     public Integer dequeue() {
14         Integer toReturn = binMinHeap.get(1);
15         binMinHeap.set(1, binMinHeap.remove(binMinHeap.size() - 1));
16         bubbleDown(1);
17         return toReturn;
18     }
19
20     // Adds item to priority queue
21     public void enqueue(Integer item) {
22         binMinHeap.add(item);
23         bubbleUp(binMinHeap.size() - 1);
24     }
25
26     // Swaps the elements at index1 and index2 of the binary min heap
27     private void swap(int index1, int index2) {
28         int temp = binMinHeap.get(index1);
29         binMinHeap.set(index1, binMinHeap.get(index2));
30         binMinHeap.set(index2, temp);
31     }
32
33     // Bubbles up the element in the binary min heap array list at given index
34     private void bubbleUp(int index) {
35         while (index / 2 > 0 && binMinHeap.get(index) < binMinHeap.get(index / 2)) {
36             swap(index, index / 2);
37             index = index / 2;
38         }
39     }
40
41     // Bubbles down the element in the binary min heap array list at given index
42     private void bubbleDown(int index) {
43         int n = binMinHeap.size();
44         while (index * 2 < n && binMinHeap.get(index) > binMinHeap.get(index * 2)) {
45             swap(index, index * 2);
46             index = index * 2;
47         }
48     }
49 }
```

(a) Draw an example of a binary min-heap with exactly 5 nodes (either tree or array list form is fine) such that calling Joe's `dequeue` method on your binary min-heap produces an invalid binary min-heap.

Solution:



(b) Which lines of code are buggy? `lines 44-46` Explain the bug.

When we bubble down, we have to check whether or not our current node is bigger than either of its children. If it is, then we switch it with the smaller of the two. This code only checks if the left node is smaller.

(c) Rewrite the lines of code you specified in part b so that his min-heap will work as intended.

Solution: Example 1

```
while (index * 2 < n) {
    int swapIndex = index * 2;
    if (index * 2 + 1 < n) {
        if (binMinHeap.get(index * 2) > binMinHeap.get(index * 2 + 1)) {
            swapIndex = index * 2 + 1;
        }
    }
    if (binMinHeap.get(index) > binMinHeap.get(swapIndex)) {
        swap(index, swapIndex);
        index = swapIndex;
    } else {
        break;
    }
}
```

Solution: Example 2

```
if (index >= n || index * 2 >= n) {
    return;
}
int curr = binMinHeap.get(index);
int left = binMinHeap.get(index * 2);
if (index * 2 + 1 == n) { // no right children
    if (curr > left) {
        swap(index, index * 2);
        bubbleDown(index * 2);
    }
} else { // two children
    int right = binMinHeap.get(index * 2 + 1);
    if (left < right) {
        if (curr > left) {
            swap(index, index * 2);
            bubbleDown(index * 2);
        }
    } else {
        if (curr > right) {
            swap(index, index * 2 + 1);
            bubbleDown(index * 2 + 1);
        }
    }
}
```

Comments: The following are common errors:

1. Forgetting to check that `index * 2 + 1 < n` before `binMinHeap.get(index * 2 + 1)`: There will be an `IndexOutOfBoundsException` in cases without a right child.
2. Not finding the smaller of the children before comparing with the current element: Many solutions fell into the same mistake as the prompt's `bubbleDown` method.
3. Terminating incorrectly: Solution should stop bubbling down children if there was no swap with the current element. Some solutions also causes infinite loops.
4. Not dealing with all cases: Solution should deal with cases where there are only one or both children and when a swap is or is not needed.

9 Every Other (8 points)

```
1  public class MyLinkedList {
2
3      private ListNode head;
4
5      public MyLinkedList(ListNode inputHead) {
6          head = inputHead;
7      }
8
9      public MyLinkedList(Object item) {
10         head = new ListNode(item);
11     }
12
13     private class ListNode {
14         private Object item;
15         private ListNode next;
16
17         public ListNode(Object inputItem) {
18             this(inputItem, null);
19         }
20
21         public ListNode(Object inputItem, ListNode next) {
22             this.item = inputItem;
23             this.next = next;
24         }
25         // There may be other methods not shown here
26     }
27     // There may be other methods not shown here
28 }
```

On the **next page**, write an `evenOdd` method in the `MyLinkedList` class above that destructively sets the linked list to contain every other linked list node of the original linked list, starting with the **first** node. Your method must **also** return a linked list that contains every other linked list node of the original linked list, starting with the **second** node.

Your method should work destructively and should not create any new `ListNode` objects. If a `MyLinkedList` contains zero elements or only one element, a call to `evenOdd` should return `null`. The last `ListNode` of each `MyLinkedList` has it's `next` instance variable set to `null`.

Example: If a `MyLinkedList` initially contains the elements [5, 2, 3, 1, 4], then a call to `evenOdd` should return a `MyLinkedList` with the elements [2, 1], and after the call, the original `MyLinkedList` should contain the elements [5, 3, 4]

Solution:

```
public MyLinkedList evenOdd() {
    if (head == null || head.next == null)
        return null;
    ListNode curr = head;
    ListNode ret = head.next;
    MyLinkedList retList = new MyLinkedList(ret);
    while (curr != null && ret && null) {
        curr.next = ret.next;
        curr = curr.next;
        if (curr != null) {
            ret.next = curr.next;
            ret = curr.next;
        }
    }
    return retList;
}
```

Comments: Many students tried to use methods that were not provided in the MyListNode class (like add and remove). Non-destructive solutions, solutions resulting in NullPointerExceptions, and creation of cyclical lists were also fairly common.

10 MemoryMap (12 points)

You have been tasked with designing a `MemoryMap` class that implements the `Map<K, V>` interface and supports the following operations:

- `V put(K key, V value)`: Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced.
- `V get(K key)`: Returns the value associated with the input `key`, or `null` if there is no mapping for the key.
- `V remove(Object key)`: Removes the mapping for the specified key from this map if present. Returns the previous value associated with the input `key`, or `null` if there was no mapping for `key`.
- `ArrayList<K> recent(int m)`: Returns an `ArrayList` of the `m` unique most recently accessed keys (sorted from most recently accessed to least recently accessed) that still have associated values in the map. A key `k` is considered accessed whenever `put` or `get` is called with `k` as the key. If there are fewer than `m` elements in the map, returns an `ArrayList` with all of the map's keys. Calls to `recent` should not modify the state of the `MemoryMap` in any way (so calling `recent` multiple times in a row without any other method calls in between should result in `recent` returning identical `ArrayList`s).

Example:

```
MemoryMap<String, String> map = new MemoryMap<String, String>();
map.put("A", "1");
map.put("B", "2");
map.recent(2); // Returned list contains: ["B", "A"]
map.get("A"); // Returns "1"
map.recent(2); // Returned list contains: ["A", "B"]
map.put("C", "3");
map.recent(3); // Returned list contains: ["C", "A", "B"]
map.remove("A");
map.recent(2); // Returned list contains: ["C", "B"]
```

(a) Explain in words how you would implement `MemoryMap` (including which data structures you would use) so that the operations listed on the previous page are time efficient. Space efficiency is not a concern. **Do not write any code.** Solutions that are as efficient as possible will receive full credit. Less efficient solutions may receive partial credit.

Solution: For part (a), multiple solutions were accepted as long as the student demonstrated usage of data structures that could potentially solve the MemoryMap question (without consideration for efficiency). Some examples of accepted solutions were:

- HashMap with some list to track the most recent.
- HashMap with either priority queue or second hash map with associated integers to track recency.
- HashMap, storing pointers to list nodes in a list that tracks most recent.

Comments: Some common errors were:

- Using some kind of binary search tree (keys are not necessarily comparable).
- Not specifying the usage of a `HashMap`, or assuming that the `Map` interface was actually a `HashMap`.

(b) For each of the methods listed below, explain how you would implement the method for `MemoryMap` and state your planned implementation's average case running time. State any assumptions you make about the average case running times of any data structures you would use in your implementation.

Solution: Optimal solution: extend `HashMap<K, V>`. Add a linked list of `ListNode<K>` with a `HashMap<K, ListNode<K>`. Start put, get, and remove with a call to super. This received 10 points.

- `put`: Look `K` up in the list node hash map; if it exists, remove it from the linked list and the hash map. Add K to the front of the linked list. O(1)
- `get`: Look `K` up in list node hash map; if it exists, remove it from the linked list and the hash map. Add `K` to the front of the linked list. $O(1)$
- `remove`: Look K up in list node hash map; if it exists, remove it from the linked list and the hash map. $O(1)$
- `recent`: Retrieve the first m elements of the linked list. $O(m)$.

Comments: Some examples of suboptimal solutions were (where N is the number of entries in the hash map):

- `HashMap` and a `ArrayList` or `LinkedList` or `Stack` as your recently accessed list. Either you have to move items around in the list to get rid of duplicates (inefficient put/get/remove running in $O(N)$ time but recent in $O(m)$) or you have to purge duplicates during the recent call (at best, $O(N)$ time using a hash set to check for duplicates, or $O(N^2)$ naively. Correctly implemented and analyzed, this received 7 points.
- `HashMap`, and a second `HashMap<K, Integer>` for tracking priorities with a counter. Every time a get or put is done, we increment the counter and update (or `put`) into the second hash map in $O(1)$. When we remove, we remove from both hash maps in O(1). When `recent` is called, we sort the `entrySet` on value and return the first m in $O(n \log n)$ time. This received 7 points.
- The same as above, but using a priority queue instead of a hash map. `get` and `put` took $O(\log n)$ and `remove` took $O(n)$. `recent` took $O(n \log k)$, or $O(n \log n)$ depending on whether you limited the size of the priority queue or not. This usually received around 4-7 points depending on implementation.
- Using a splay tree. There's almost no way to get this to work; while splay trees do give the most recent item in constant time at the top of the tree, there are no guarantees as to where the m most recent items will be in the tree. Additionally, this requires that the keys be comparable, which is not guaranteed.

Reference Sheet: `ArrayList`

Here are some methods and descriptions from Java's `ArrayList<E>` class API.

Return type and signature	Method description
`boolean add(E e)`	Append the specified element to the end of the list
`boolean contains(Object o)`	Returns `true` if this list contains the specified element
`E get(int index)`	Returns the element at the specified position in this list
`Iterator<E> iterator()`	Returns an iterator over the elements in this list in proper sequence
`E remove(int index)`	Removes the element at the specified position in this list
`boolean remove(Object o)`	Removes the first occurrence of the specified element from this list, if it is present
`E set(int index, E element)`	Replaces the element at the specified position in this list with the specified element (returns the previous element at this position)
`int size()`	Returns the number of elements in this list

Reference Sheet: `Map<K, V>`

Here are some methods and descriptions from Java's `Map<K, V>` interface API.

Return type and signature	Method description
`V get(Object key)`	Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key
`boolean isEmpty()`	Returns `true` if this map contains no key-value mappings
`V put(K key, V value)`	Associates the specified value with the specified key in this map (returns the previous value associated with `key`, or `null` if there was no mapping for `key`)
`V remove(Object key)`	Removes the mapping for a key from this map if it is present
`Set<K> keySet()`	Returns a `Set<K>` of the keys contained in this map (the `Set<K>` class implements `Iterable<K>`)