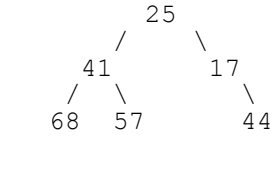
CSE 373 Section Handout #6

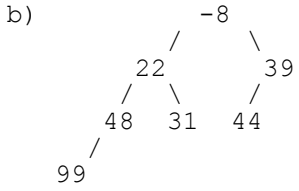
1. Identifying heaps

Which of the following trees are valid binary min-heaps?

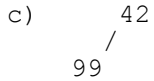
a)



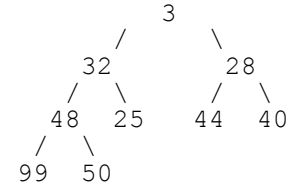
b)



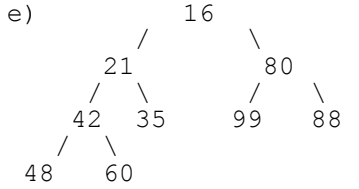
c)



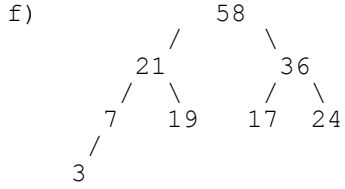
d)



e)



f)



2. Heap simulation 1

Simulate these *adds* on an initially empty binary min-heap. Draw the heap tree after each/all adds. After adding all the elements, perform *three remove (minimum) operations* on the tree and draw its state. Also write the state that would be in the array representation of the same heap after the adds and removes.

- add 35, 42, 28, 17, 61, 12, 21, 15, 30, 9, 25
- remove, remove, remove

3. Heap simulation 2

Simulate these *adds* on an initially empty binary min-heap. Draw the heap tree after each/all adds. After adding all the elements, perform *three remove (minimum) operations* on the tree and draw its state. Also write the state that would be in the array representation of the same heap after the adds and removes.

- add 6, 4, 8, 2, 9, 1, 5, 7, -1, 0
- remove, remove, remove

4. Heap simulation 3

Simulate these *adds* on an initially empty binary min-heap. Draw the heap tree after each/all adds. After adding all the elements, perform *three remove operations* (*not* min) on the tree and draw its state.

- add 15, -1, 12, 30, -5, 26, 4, 19, -10, 42
- remove 26, 15, 4

5. Heap simulation 4

Simulate these *adds* on an initially empty binary min-heap. Draw the heap tree after each/all adds. After adding all the elements, perform *three remove operations* (*not* min) on the tree and draw its state.

- add 35, 99, 80, 42, 60, 21, 57, 88, 16
- remove 42, 35, 21

CSE 373 Section Handout #6

For these problems, recall the `Comparator` *interface from* `java.util` *to perform external ordering:*

```
public interface Comparator<T> {
    public int compare(T object1, T object2);
}
```

6. `StringCaseInsensitiveComparator`

Write a class named `StringCaseInsensitiveComparator` that implements the `Comparator` interface to compare `String` objects. Your comparator should use the normal ABC ordering except it should ignore capitalization during the comparison. So for example, your comparator would indicate the following strings should be in this order:

- `["any", "AT", "HEAVEN", "hElLo", "hugs", "Pineapple", "yogurt"]`

7. `PointXYComparator`

Write a class named `PointXYComparator` that implements the `Comparator` interface to compare `Point` objects from `java.awt`. Your comparator should order the points by ascending *x*, breaking ties by ascending *y*. So for example, your comparator would indicate the following points should be in this order:

- `[(4,2), (4,5), (7,-1), (7,5), (7,5), (14,1), (25,19)]`

8. `PointManhattanDistanceComparator`

Write a class named `PointManhattanDistanceComparator` that implements the `Comparator` interface to compare `Point` objects from `java.awt`. Your comparator should order the points by ascending "Manhattan distance" from the origin, (0, 0). A "Manhattan distance" between two points (also sometimes called the "taxicab distance" or "rectangular distance") is the absolute sum of the differences in *x* and *y* between the two points. It gets its name from the idea that in Manhattan, a car must drive in straight horizontal/vertical lines to reach its destination. For example, the point (3, -5) has a distance of 3+5=8 from the origin. So your comparator would indicate the following points should be in this order:

- `[(0,1), (-2,0), (1,2), (4,-1), (3,3), (5,-2), (4,4)]`

(Please note that "Manhattan distance" is not the kind of distance you should use on your homework.)

CSE 373 Section Handout #6

For these problems, recall the `HeapPriorityQueue` *class written in lecture using a binary min-heap. Assume that methods called* `bubbleUp` *and* `bubbleDown` *have been added to the class to move an element at a given index up/down the heap until it is in its proper position. (The* `bubbleUp` *method is used as a helper by the* `add` *method, and* `bubbleDown` *is used as a helper by* `remove`*.)*

```
public class HeapPriorityQueue<E> implements PriorityQueue<E> {
    private E[] elements;
    private int size;

    public HeapPriorityQueue() {...}
    public void add(E value) {...}
    public boolean isEmpty() {...}
    public E peek() {...}
    public E remove() {...}
    public int size() {...}
    public String toString() {...}

    private void bubbleUp(int index) {...}
    private void bubbleDown(int index) {...}
    private int parent(int index) {...}
    private int leftChild(int index) {...}
    private int rightChild(int index) {...}
    private boolean hasParent(int index) {...}
    private boolean hasLeftChild(int index) {...}
    private boolean hasLeftRightChild(int index) {...}
    private void swap(E[] array, int index1, int index2) {...}
}
```

9. HeapPriorityQueue replace

Write a method named `replace` that could be added to the `HeapPriorityQueue` class from lecture. This method accepts an element value *value1* and a replacement value *value2*, and finds and replaces one occurrence of *value1* with *value2* if *value1* is present in the heap. You must maintain the heap's ordering after your method's work is done. For example, if a heap priority queue `pq` contains `[12, 41, 35, 56, 71, 40, 52, 84, 60, 78, 99, 66]`, the call of `pq.replace(56, 30);` would change `pq` to store `[12, 30, 35, 41, 71, 40, 52, 84, 60, 78, 99, 66]`. A subsequent call of `pq.replace(35, 88);` would change `pq` to store `[12, 30, 40, 41, 71, 52, 88, 84, 60, 78, 99, 66]`. If the *value1* is not found in the heap, no change occurs to the heap.

You are allowed to call methods on your priority queue. This method should run in O(*N*) time where *N* is the number of elements in your queue.

10. HeapPriorityQueue constructor

Write a second constructor that could be added to the `HeapPriorityQueue` class from lecture. This constructor accepts an array of elements as a parameter and uses that array as the heap rather than creating a new array. Of course, the array passed in is probably not in proper heap ordering, so you must rearrange it until it is. There's a neat trick for achieving this: If you just "bubble down" all of the non-leaf nodes of the heap, starting from the last non-leaf node and ending at the root, when you are done the array will be rearranged into proper heap order. *(Why does this work?)* For example, if you are passed the array `[/, 46, 21, 15, 37, 51, 9, 12, 78, 31, 40, 8, 24]`, your constructor would rearrange it to be `[/, 8, 21, 9, 31, 40, 15, 12, 78, 37, 46, 51, 24]`. Assume that element 0 is empty.

You are allowed to call methods on your priority queue. This constructor should run in O(*N*) time where *N* is the number of elements in the array. (If you follow the algorithm described, it turns out to be O(*N*) because the sum of the swap heights of the internal heap tree nodes is proportional to *N*.)

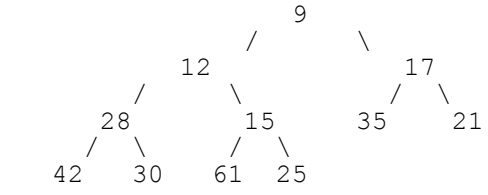
CSE 373 Section Handout #6
Solutions

1.

Examples c) and e) are valid binary min-heaps.
Example a) is not because 44 should be left of 17, and because 25 and 17 are out of order vertically.
Example b) is not because 99 is in the wrong spot (should be right of 39).
Example d) is not because 32 and 25 are out of order vertically.
Example f) is not because the entire tree is in backwards vertical order (it is a max-heap, not a min-heap).

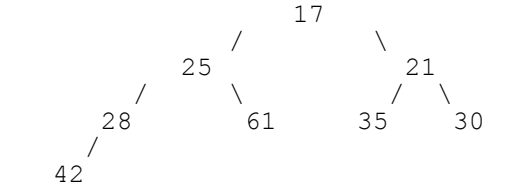
2.

after all adds:



[9, 12, 17, 28, 15, 35, 21, 42, 30, 61, 25]

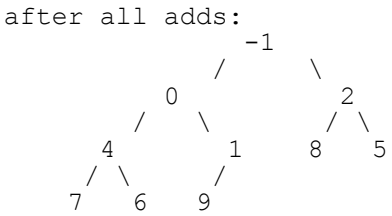
after three remove-mins:



[17, 25, 21, 28, 61, 35, 30, 42]

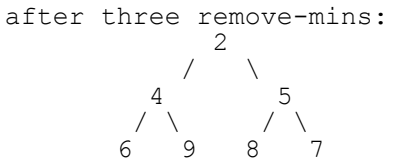
3.

after all adds:



[-1, 0, 2, 4, 1, 8, 5, 7, 6, 9]

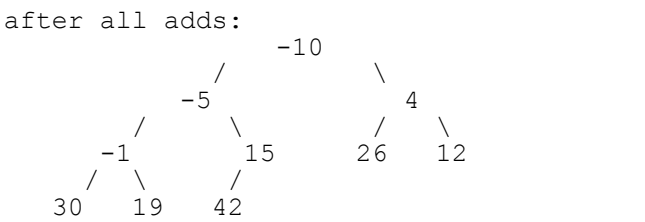
after three remove-mins:



[2, 4, 5, 6, 9, 8, 7]

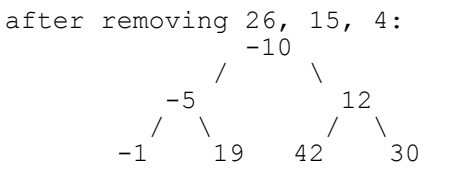
4.

after all adds:



[-10, -5, 4, -1, 15, 26, 12, 30, 19, 42]

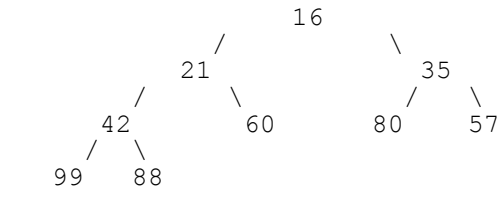
after removing 26, 15, 4:



[-10, -5, 12, -1, 19, 42, 30]

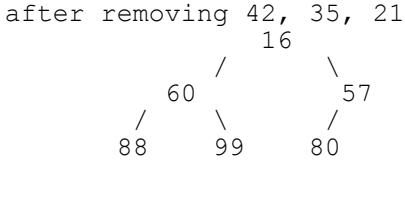
5.

after all adds:



[16, 21, 35, 42, 60, 80, 57, 99, 88]

after removing 42, 35, 21:



[16, 60, 57, 88, 99, 80]

CSE 373 Section Handout #6
Solutions, continued

6.
```
public class StringCaseInsensitiveComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
        return s1.toLowerCase().compareTo(s2.toLowerCase());
    }
}
```

7.
```
public class PointXYComparator implements Comparator<Point> {
    public int compare(Point p1, Point p2) {
        if (p1.x != p2.x) {
            return p1.x - p2.x;
        } else {
            return p1.y - p2.y;
        }
    }
}
```

8.
```
public class PointManhattanDistanceComparator implements Comparator<Point> {
    public int compare(Point p1, Point p2) {
        return (Math.abs(p1.x) + Math.abs(p1.y)) -
               (Math.abs(p2.x) + Math.abs(p2.y));
    }
}
```

9.
```
public void replace(E value1, E value2) {
    for (int i = 1; i <= size; i++) {
        if (elements[i].equals(value1)) {
            elements[i] = value2;
            if (hasParent(i) && elements[i].compareTo(elements[parent(i)]) < 0) {
                bubbleUp(i);
            } else {
                bubbleDown(i);
            }
            return;
        }
    }
}
```

10.
```
public HeapPriorityQueue(E[] elements) {
    this.elements = elements;
    this.size = elements.length - 1;
    for (int i = size / 2; i >= 1; i--) {
        bubbleDown(i);
    }
}
```