CS 15 Heap Review

Introduction

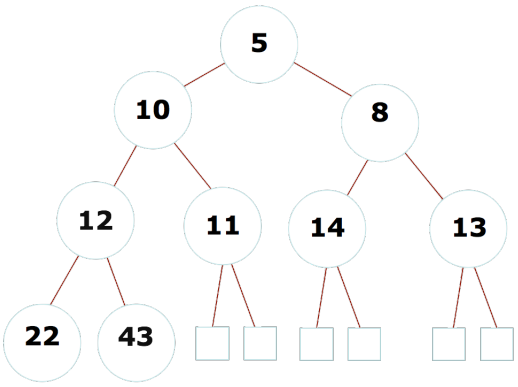
Recall that a *Heap* is a binary tree that maintains two invariants:

- The **shape** property, which requires the tree to be complete, i. e., each level except, perhaps the last is full, and the lowest level has all it's values as far left as possible. Another way to say this is that heaps must fill up level by level, left to right.
- The **heap** property, which requires that each node is more important than its children. For a min-heap, this means that a parent's value will be less than either child's value.

A heap is not a binary search tree and therefore the BST invariants do not apply.

Heap in an Array

Let's consider how a heap is laid out in an array. Take the min-heap of integer keys shown in tree form below.



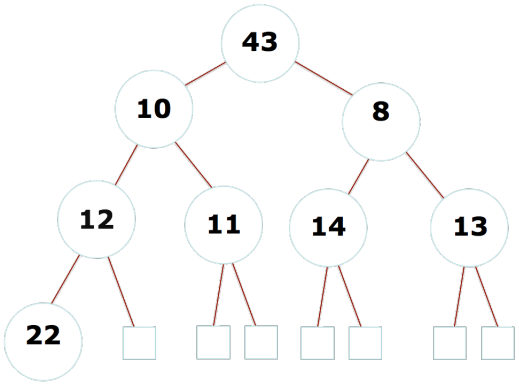
In an array, the heap is arranged in level order, from left-to-right (skipping 0), as shown below. Note that `heapSize == 9` The next element to be added to the heap would populate the slot at index 10.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
	5	10	8	12	11	14	13	22	43						

removeMin()

`removeMin()` removes the element at the root, which is stored at index `[1]`. The element at index `[1]` is always the minimum in a min-heap. We need to temporarily keep a copy of the value at index `[1]`, so that we can return it at the end of the function.

Once we have a temporary copy of the minimum (to return later), we replace the element at index `[1]` with the element at index `[heapSize]` (and we also need to decrement `heapSize`!) to restore the shape property. Because this almost certainly destroys the heap property (where all children are greater than their parents), we `downHeap()` from index `[1]`:



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
	43	10	8	12	11	14	13	22							

Now `heapSize == 8`.

downHeap()

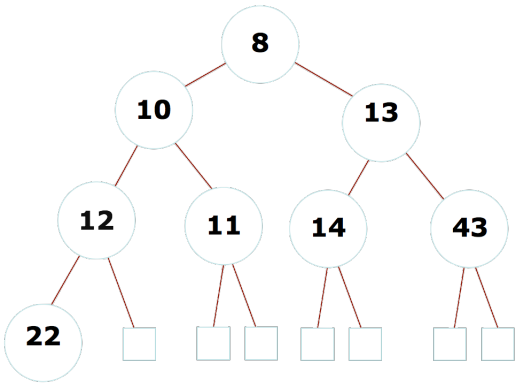
The `downHeap()` function takes a bit of thinking. Conceptually, it is easy: swap the element with its smallest child, and repeat down the heap. Practi-

cally, we need to keep in mind that a node might not have any children, or it could have a left only and not a right (but not the other way around, because a heap must be a complete tree). Here is a pseudocode down heap algorithm:

While not done:

1. Set a variable `smallest` to the index of the value we want to down-heap
2. if left child's index `<= heapsize` and left child's value `<` value at `smallest`
   - set `smallest` to index of left child
3. if right child's index `<= heapsize` and right child's value `<` value at `smallest`
4. set `smallest` to index of right child
5. if `smallest` is still equal to the index we want to down-heap:
   - then done
   - otherwise, swap the value at `smallest` with the value at the index of the original value we want to down-heap
   - The index of the value to down-heap becomes `smallest`

This produces the heap shown below.



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
	8	10	13	12	11	14	43	22							

buildHeap()

Finally, let's discuss `buildHeap()`. We use the `buildHeap()` function when we want to create a heap from a nonheap array of values. The basic idea is to `downHeap()` enough elements to ensure that we have a heap. The trick to `buildHeap()` is knowing that you can start down-heaping at `heapSize / 2`, because you will eventually down-heap enough values to create a true heap. In pseudocode:

```
for (i = heapsize / 2) down to 1:
    downHeap(i)
```

That's it!

Questions

1. Why do we lay out the heap starting from position 1 (think about parents and children)?
2. Why do we have to check if the left child's index is less than or equal to `heapSize` in the `downHeap()` function?
3. How do we know that we are done down-heaping in the `downHeap()` function?
4. Would the `buildHeap()` function work if we started our loop from `heapSize` instead of `heapSize / 2` ? Why are we able to start at `heapSize / 2` ?