CS61B Lecture 23

Monday, March 16, 2020

Announcements

- We will move to Zoom lectures on Wednesday, which will have a maximum of 300 live participants. They will still be recorded for later playback.
- There will be adjustments to deadlines and tests over the next couple of weeks. Please stay tuned.

Searching

The theme so far has been collections of data and how to search through them. We have seen trees and the use of trees for searching. We have done game trees and their use in searching for a good move. Now we will see some specialized search structure.

Priority Queues and Heaps

You may be a little confused about the distinction between these two concepts. In this class, we will define the priority queue as the abstract algorithmic idea, and the heap is an implementation of the idea.

A priority queue is simply a collection of items which has the operations "add", "find largest", and "remove largest". We can stick things in in any order, but we can pull the largest item at any time.

How can this be useful? Well, we could schedule a long stream of actions that have to occur at certain times, where the earliest time is defined as the largest. At any given time, we can execute the next action by finding the largest, performing it and then removing it from the queue.

We could also use it to sort by repeatedly removing the largest item from the queue.

For this concept, the standard implementation is the heap, which is a kind of tree. There are multiple data structures that can give the same effect of a priority queue, including some examples you will see later in this structure. The heap is however, particularly good at this set of operations.

Confusingly, the heap is also used to describe the pool of storage that your `new` requests go to. Don't mix them up!

Heap

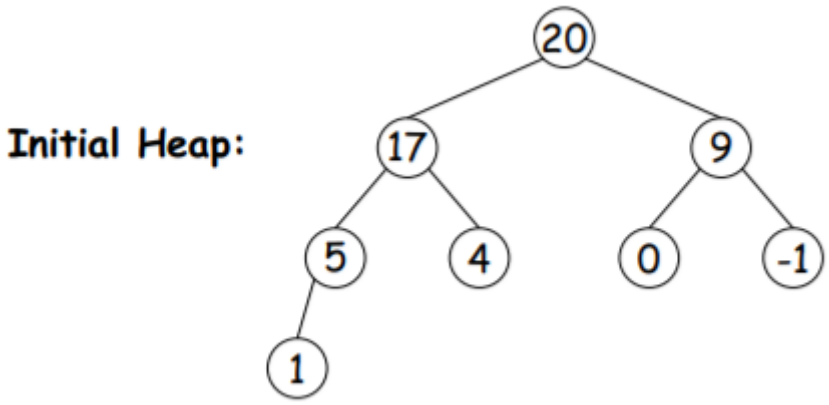
A **max-heap** is a binary tree that enforces the **heap property**, where the labels of both children are less than the node's label, such that the node at the top has the largest label.

The definition is looser than the binary search property, which allows us to keep the tree "bushy". It is always valid to put the smallest nodes anywhere at the bottom of the tree. Because of this, heaps can be made nearly complete, where all but possibly the last row have as many keys as possible. As a result, the worst case insertion and deletion time always takes logarithmic time.

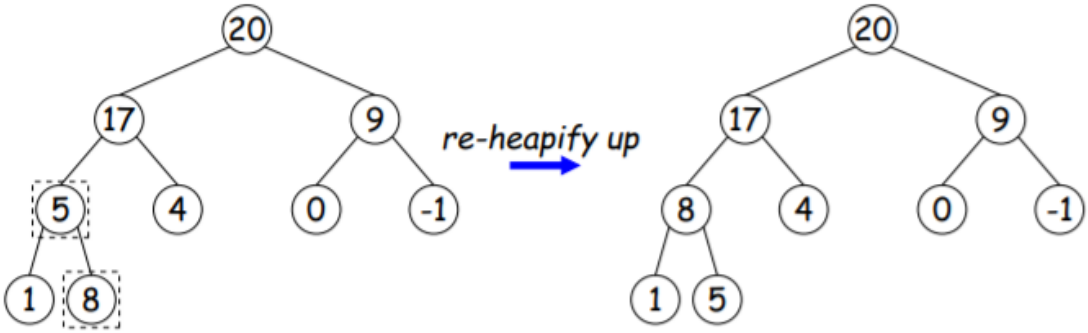
A **min-heap** is basically the same thing, but with the minimum value at the root and children have larger values than their parents.

Adding to heaps

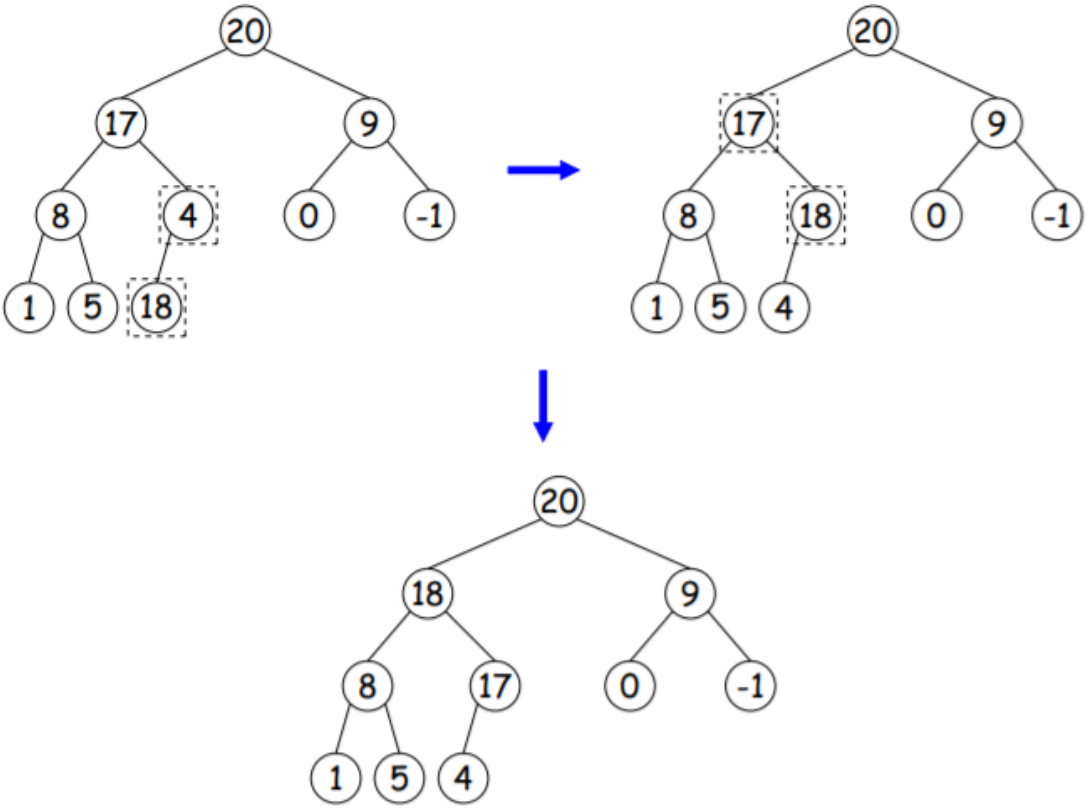
For the data `1 17 4 5 9 0 -1 20`, we can heapify it into this form: (To start, we can either start with an empty heap and continually use `add`, or another technique we will see later)



To add 8, we will first add it to the bottom, where the heap property is violated, so we will re-heapify the tree:

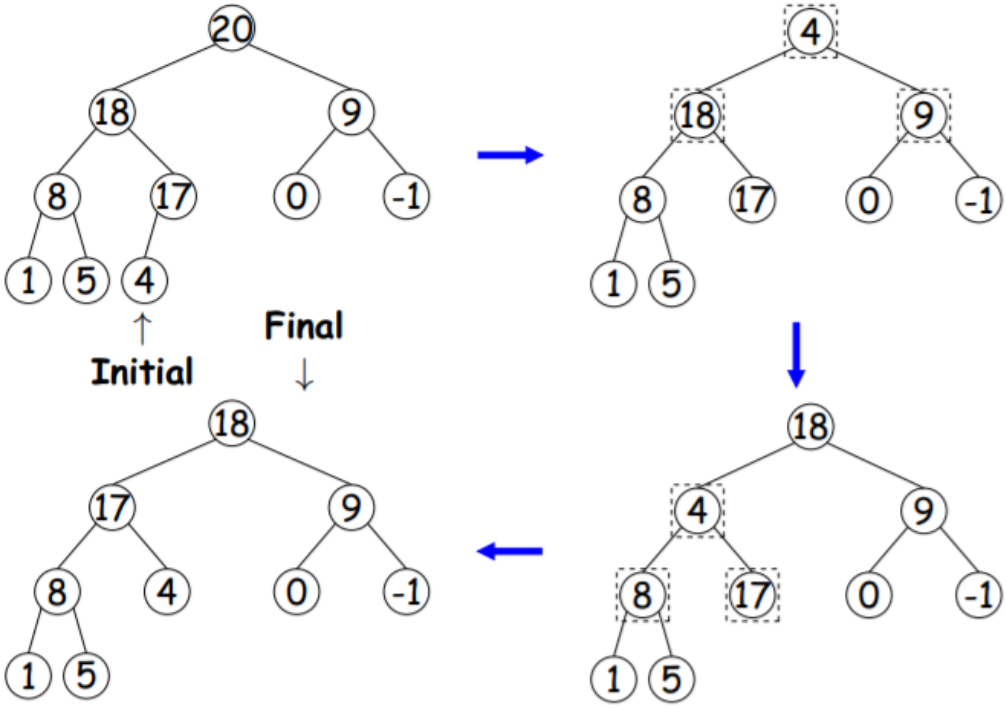


And adding 18 has a similar effect:



Removing from heaps

To remove the largest node, we move the bottommost, rightmost node to the top, then re-heapify down as needed (swap offending node with larger child) to re-establish heap property.

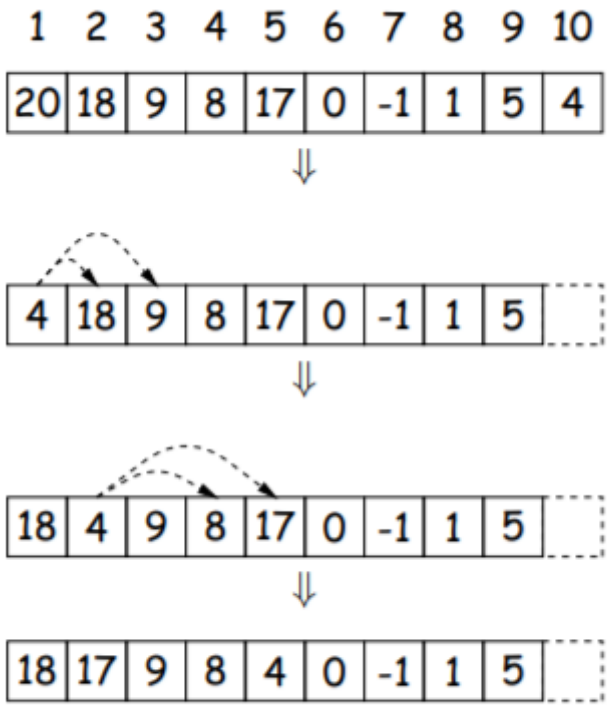


The reason we do this is to preserve as much of the original shape of the heap as possible.

Heaps as Arrays

Since heaps are nearly complete, we can use arrays for compact representation, where the nodes are stored in level order. The children of the node at index `k` are in `2k` and `2k + 1` if numbering from 1, or `2k + 1` and `2k + 2` if numbering from 0.

We can see the removal process in array from as such:



Ranges

So far, we have always looked for specific items in collections. Consider instead the scenario where we want all values in a particular range, such as all labels in a tree T that are between two values L and U. Here is some code that does that:

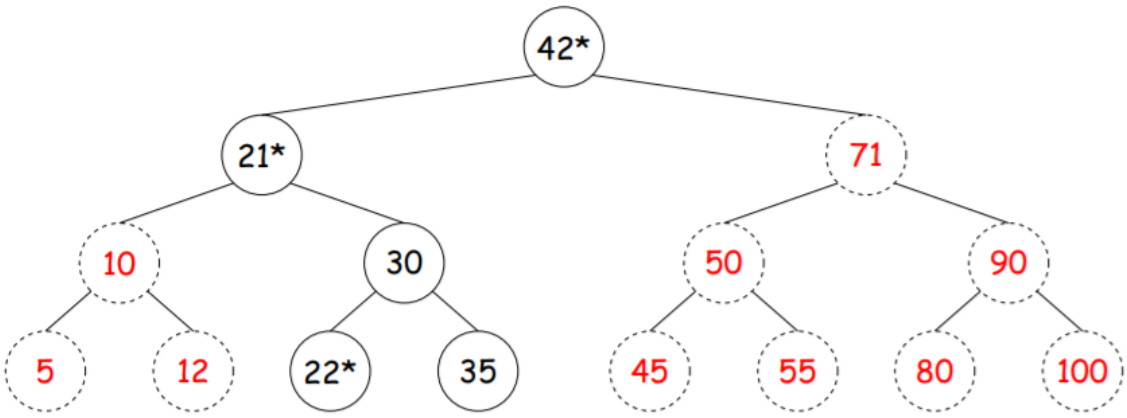
```
/** Apply WHATTODO to all labels in T that are >= L and < U,
 *  in ascending natural order. */
static void visitRange(BST<String> T, String L, String U, Consumer<BST<String>>
whatToDo) {
    if (T != null) {
        int compLeft = L.compareTo(T.label()),
            compRight = U.compareTo(T.label());
        if (compLeft < 0) { /* L < label */
            visitRange(T.left(), L, U, whatToDo);
        }
        if (compLeft <= 0 && compRight > 0) { /* L <= label < U */
            whatToDo.accept(T);
        }
        if (compRight > 0) {/* label < U */
            visitRange(T.right(), L, U, whatToDo);
        }
    }
}
```

Time for Range Queries

The time for the range query is in

where h is the height of the tree and M is the number of data items that turn out to be in the range.

Consider the situation of searching the tree below for all values between 25 inclusive and 40 exclusive.



The dashed nodes are never looked at. The starred nodes are looked at but not output, and the h comes from the starred nodes, while M comes from un-starred non-dashed nodes. This means the function is faster when our range is narrower.

Ordered Sets and Range Queries

A good engineer, instead of implementing this data structure him or herself, will use an already-completed implementation. Don't repeat work that's already been done!

In Java, this is represented by the `SortedSet` that supports range queries with *views* of a set:

- `S.headSet(U)`: subset of `S` that is less than `U`.
- `S.tailSet(L)`: subset of S that is greater than or equal to `L`.
- `S.subSet(L, U)`: subset that is greater than or equal to `L` and less than `U`.

Any changes to views will modify the original `S`. Attempts to, for example, add to a `headSet` beyond U are disallowed.

You can iterate through a view to process a range. For example:

```
SortedSet<String> fauna = new TreeSet<String (Arrays.asList ("axolotl", "elk",
"dog", "hartebeest", "duck"));
for (String item : fauna.subSet ("bison", "gnu")) {
    System.out.printf ("%s, ", item);
}
```

`TreeSet`

The Java library type `TreeSet<T>` is the concrete implementation of `SortedSet` in Java. It requires that `T` either be `Comparable`, or that you provide a `Comparator`, like this:

```
SortedSet<String> rev.fauna = new TreeSet<String>(Collections.reverseOrder());
```

`Comparator` is a type of function object:

```
interface Comparator<T> {
    /** Return <0 is LEFT<RIGHT, >0 if LEFT>RIGHT, else 0 */
    int compare(T left, T right);
}
```

(We will deal with what `Comparator<T extends Comparable<T>>` at some later point.)

For example, the `reverseOrder` comparator is implemented as such:

```
static <T extends Comparable<T>> Comparator<T> reverseOrder() {
    // Java figures out this lambda expression is a Comparable<T>
    return (x, y) -> y.compareTo(x);
}
```

`BSTSet` representation

How do we implement this view thing? Below is a rather complicated example you can sit on and consider.

Each object has 3 fields, a pointer to a tree, and its lower and upper limits. When we create subsets, we use exactly the same tree, but put in the limits, such that the object will only show things within the limits.

(One small note is that `.size()` is very expensive!)

```
SortedSet<String>
    fauna = new BSTSet<String>(stuff);
    subset1 = fauna.subSet("bison","gnu");
    subset2 = subset1.subSet("axolotl","dog");
```