# Data Structures
# Heaps

Cain Susko

Queen's University
School of Computing

March 2, 2022

## Heaps

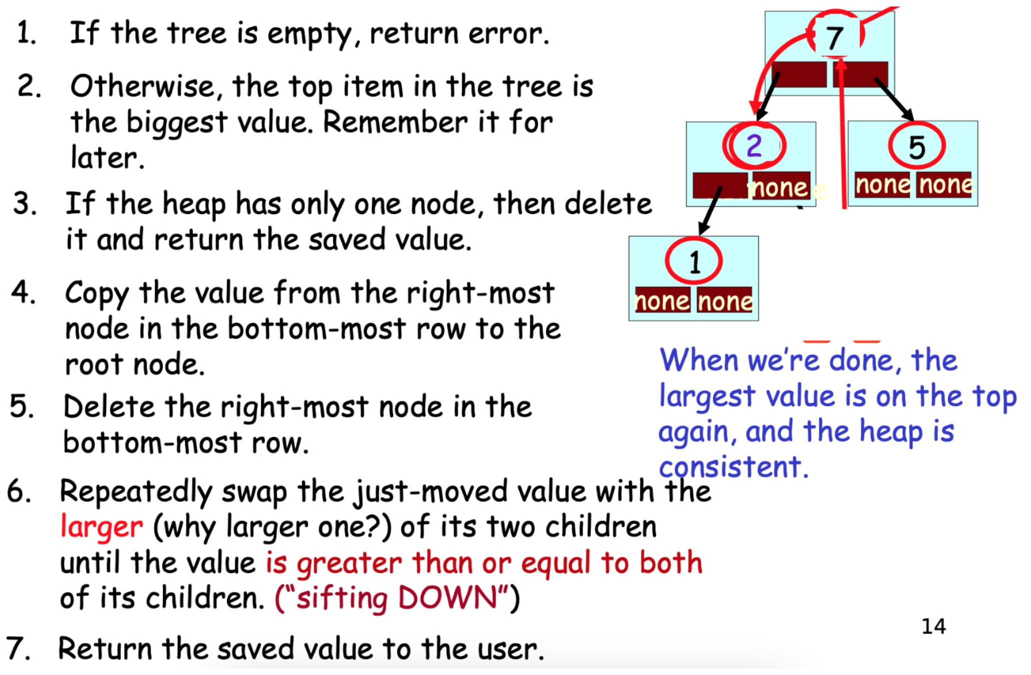
heaps are a type of data structure that is based on the binary tree and a priority queue. there are 2 types of a heap: Maxheap and Minheap

**Maxheap** a max heap can quickly insert values and quickly get the largest value inthe heap.
within a maxheap, data is contained such that a nodes children are all less then the value in the node. it is also a complete binary tree.

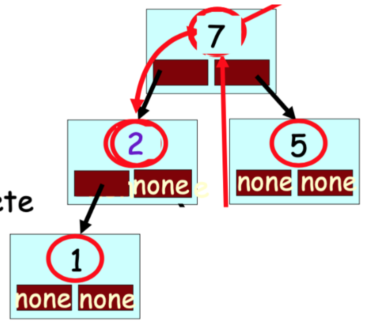
**Minheap** a min heap can quickly insert into heap and get the smallest value from the heap,
the structure is the same as maxheap but with the min at the root with all children are lagrer.

### Operations

the main operation with a heap is **extraction**. The algorithm for extraction is as follows:



1. If the tree is empty, return error.
2. Otherwise, the top item in the tree is the biggest value. Remember it for later.
3. If the heap has only one node, then delete it and return the saved value.
4. Copy the value from the right-most node in the bottom-most row to the root node.
5. Delete the right-most node in the bottom-most row.
6. Repeatedly swap the just-moved value with the larger (why larger one?) of its two children until the value is greater than or equal to both of its children. ("sifting DOWN")
7. Return the saved value to the user.



When we're done, the largest value is on the top again, and the heap is consistent.

14

## Implementation

we can use arrays in order to implement a heap.

we can map the complete binray tree of the heap using level order traverse and entering the values by row. the memory for each row increses eponentially (accept for possibly the final row).

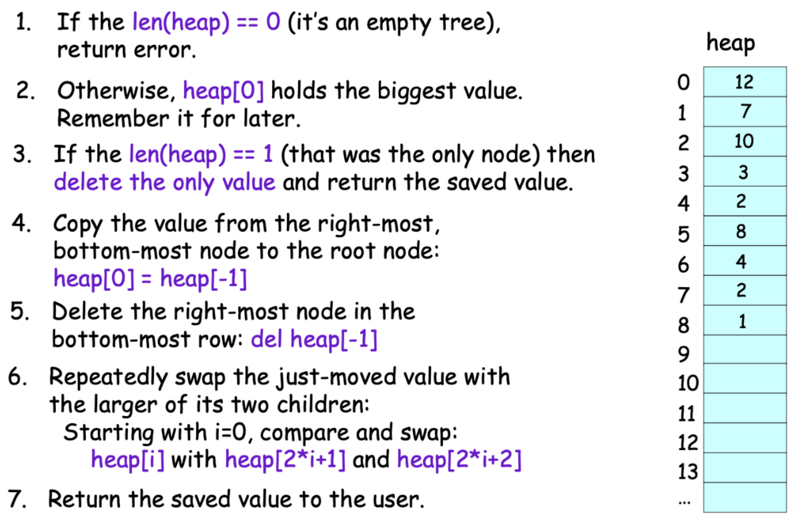
with this implementation, we can always find the largest value at `HEAP[0]` and the bottom-right-most mode at `HEAP[-1]`. finally, we can also always find the bottom-left-most empty spot (for adding a new value) at `HEAP[len(HEAP)]`.

but how do we maintain the tree structure within the array?specifically, can we find the children of a node using an algorithm like so:

```
leftChild(parent) = 2 * parent + 1
```

```
rightChild(parent) = 2 * parent + 2
```

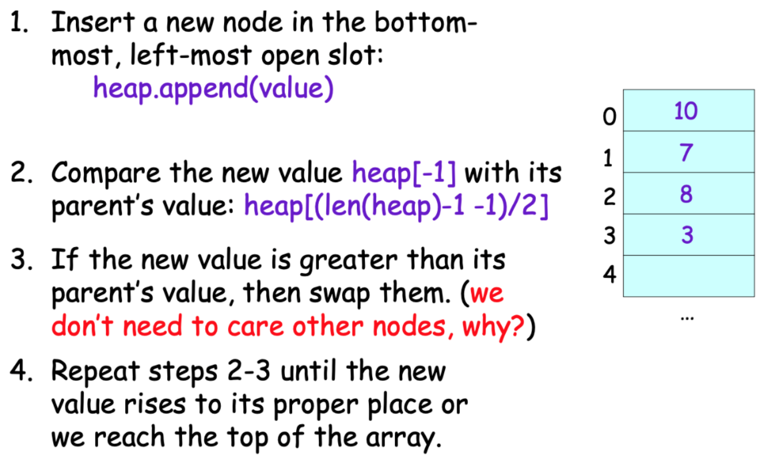
So, the code implementation may use the following logic:



1. If the len(heap) == 0 (it's an empty tree), return error.
2. Otherwise, heap[0] holds the biggest value. Remember it for later.
3. If the len(heap) == 1 (that was the only node) then delete the only value and return the saved value.
4. Copy the value from the right-most, bottom-most node to the root node: heap[0] = heap[-1]
5. Delete the right-most node in the bottom-most row: del heap[-1]
6. Repeatedly swap the just-moved value with the larger of its two children:
   Starting with i=0, compare and swap:
   heap[i] with heap[2*i+1] and heap[2*i+2]
7. Return the saved value to the user.

| | heap |
|---|---|
| 0 | 12 |
| 1 | 7 |
| 2 | 10 |
| 3 | 3 |
| 4 | 2 |
| 5 | 8 |
| 6 | 4 |
| 7 | 2 |
| 8 | 1 |
| 9 | |
| 10 | |
| 11 | |
| 12 | |
| 13 | |
| … | |

Additionally, the implementation for inserting a node to the heap would be:

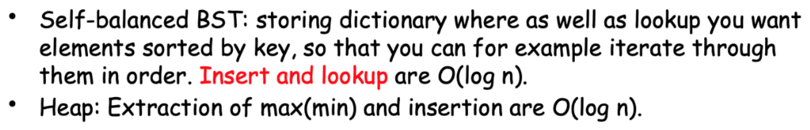


1. Insert a new node in the bottom-most, left-most open slot:
   heap.append(value)
2. Compare the new value heap[-1] with its parent's value: heap[(len(heap)-1 -1)/2]
3. If the new value is greater than its parent's value, then swap them. (we don't need to care other nodes, why?)
4. Repeat steps 2-3 until the new value rises to its proper place or we reach the top of the array.

0
10
1
7
2
8
3
3
4
...

## Complexity

since our tree is a *complete* binary tree, if it has $N$ entries, it is guarunteed to be exactly $log_2(N)$ Levels deep.
extraction, takes, for the max/min: $O(1)$ and for any other value the complexity is $O(\log n)$
the complexity for searching for an element in a heap, however, is $O(n)$



- Self-balanced BST: storing dictionary where as well as lookup you want elements sorted by key, so that you can for example iterate through them in order. Insert and lookup are O(log n).
- Heap: Extraction of max(min) and insertion are O(log n).