Heaps and Priority Queues

CS 580U Fall 2017

Priority Queue

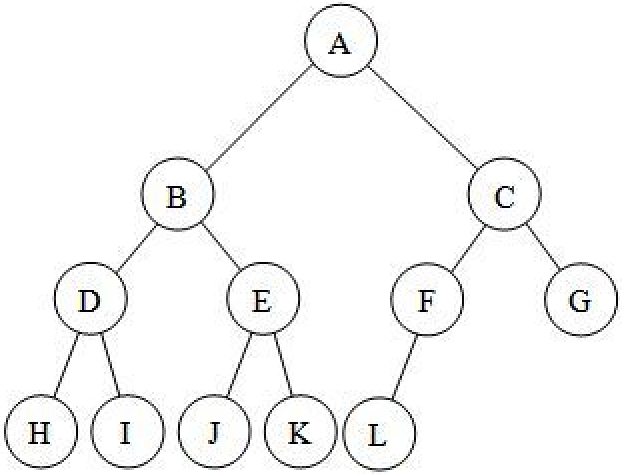
- When a collection of objects is organized by importance or priority, we call this a priority queue.
- Instead of being a "First In First Out" or “Last In First Out” data structure, values come out in order of priority.
    - Every value is assigned a priority

Problem

- Using data structures we've seen so far as a Priority Queue
  - Using an Array
    - *sorted order makes insert slow, removePriority fast*
    - *arbitrary order makes insert fast, removePriority slow*
  - Using a Linked List
    - *Sorted makes insert slow*
    - *Unsorted makes removePriority slow*
  - Using a BST
    - *removePriority is* $\log_2$ *of size IF the tree stays balanced*
    - *If the tree is not balanced, insert and removeMax can be slow.*

Balanced and Complete Trees

- Balanced Tree
  - The tree's height is as small as possible at all times
    - *No leaf is more than one level away from any other leaf in the tree*
- Complete Tree
  - A tree in which every level, except the last, is filled, and all nodes are as far left as possible.



Solution

- A balanced BST will give the best performance for a priority queue
- More Problems:
    - Read operation only accesses a single value
        - *Make read constant time by making the priority value the root*
    - BST structure uses strict ordering
        - *All values can only go one place which makes keeping the tree balanced difficult*

Heap

- A data structure that allows
  - partial ordering
  - always has the priority value as the root
- A heap is:
  - a complete binary tree
    - *nearly always implemented using the array representation*
  - The values in the tree maintain a parent/child relationship only.
    - *No defined relationship with the tree as a whole*
    - *This is called partial order*

Array Based Tree

- Mapping elements of a tree into an array
  - if a node is stored at index k
    - *the left child is at index 2k+1*
    - *The right child is at index 2k+2*
    - *The parent is (K-1)/2*
- **Assertion:** Maintaining a balanced tree is easier with an array based implementation.
    - *Agree?*
    - *Don't confuse the logical representation of a heap (tree) with its implementation (array).*

Heap Variants

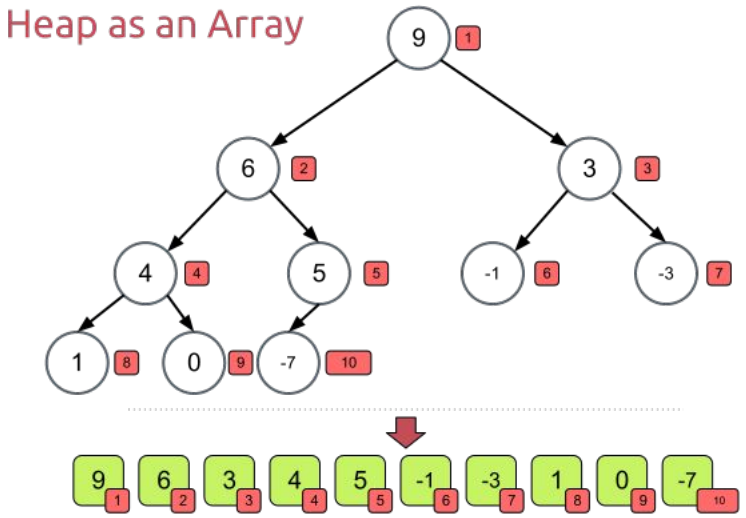
- There are two variants of the heap
  - In a **max heap**, every node stores a value that is greater than or equal to the value its children.
    - *Because the root has a value greater than or equal to its children, which have values greater than or equal to their children, the root stores the maximum of all values in the tree.*
  - In a **min heap**, every node stores a value that is less than or equal to that of its children.

Node Relationships

- There is no relationship between the value of a node and that of its sibling in a heap
    - For example, the values for all nodes in the left subtree of the root could be greater than the values for every node of the right subtree.
        - *This is a feature, not a bug*
- Contrast with a BST which has a strict ordering relationship

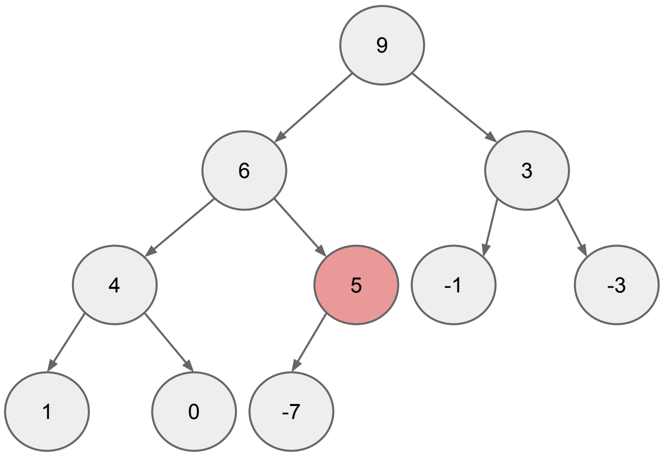
Creating A Heap

- Given an array of N values, a heap can be built by “sifting” each node down to its proper place
    - Any array can be made into a heap using the ‘Heapify’ sorting algorithm



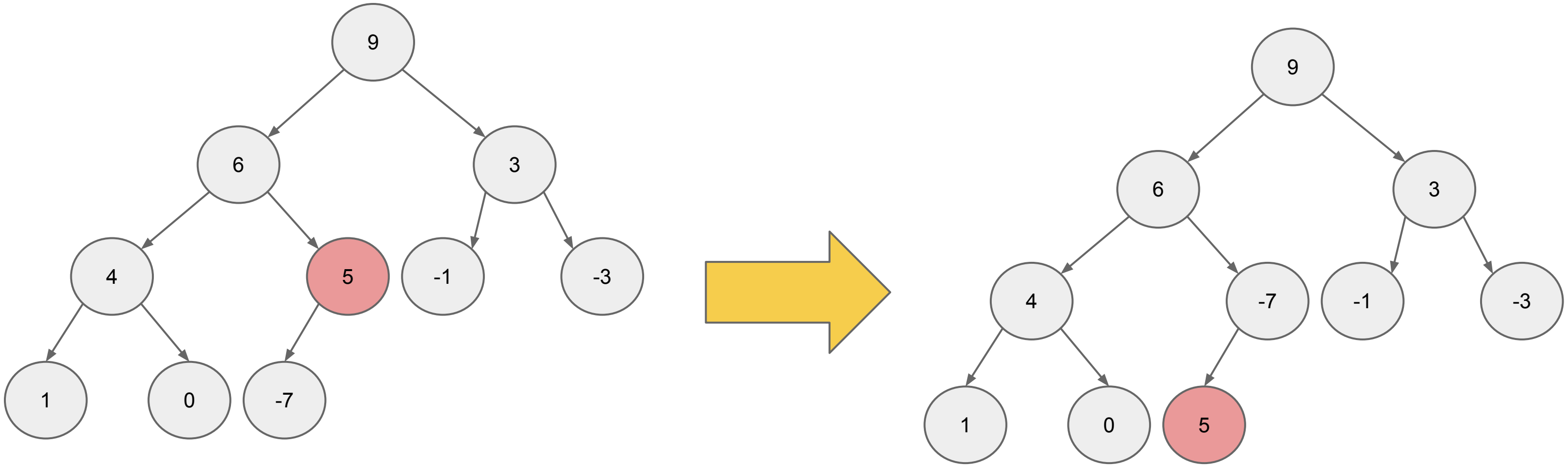
Where to Start?

- Start with the last internal node
    - How do we find the last internal node (non-leaf)?
        - *Take the last index, then find parent: (i-1)/2*
- Swap the current internal node with its smaller child, if necessary



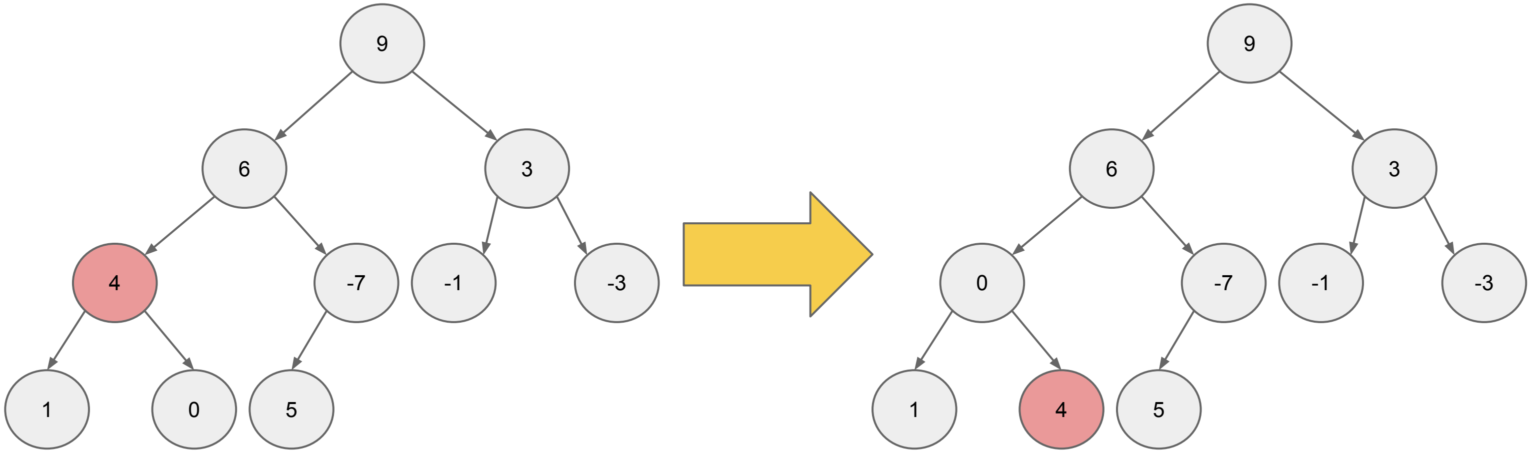
Sift Down

- Follow the swapped node down the tree until both children are larger



Sift Down

- Go to the next internal node. Repeat until all internal nodes are done
  - Check both children for the smaller value



The Heap Instance Variables

```
// Min-heap implementation
    struct MinHeap{
        Data  * heap; //Pointer to an array of comparables
        int n;          // Number of things now in heap
        int max; //maximum size of the heap
        …
```

Constructing the Heap

- Constructor supporting preloading of heap contents
  - ```
    Heap * initMinHeap(Data * h, int num, int max){
        Heap * heap = malloc(sizeof(Heap));
        heap->n = num;
        heap->size = max; heap->heap=h;
        buildheap(heap); //creates the heap data structure
        return heap;
    }
    ```
    ```

Constructing an Array Based heap

- Though not required, you should have functions that return pointers to parents and children
    - ```
      int left(i){
          if(2i + 1 > n)
              return -1;
          else
              return 2i + 1;
      }
      ```
      ```

Heapify

- //Heapify the array elements
  - ```
    void buildheap(Heap * h){
        for (int i=(h->size-2)/2; i>=0; i--)
            siftdown(h->heap, i);
    }
    ```
    - *Why is ‘i’ initialized like this?*
      - Because size is 1 more than the last index of the array
- Buildheap will run through (almost)every element of the array
    ```

SiftDown

- // Put element in its correct place
  - ```
    void siftdown(Data * heap, int pos) {
        if ((pos < 0) || (pos >= n)) return; // Illegal position
        while (!isLeaf(pos)){ //Keep swapping until you get to a leaf
            int j = left(pos); //Get left child
            if ((j+1 < n && (heap[j] > heap[j+1]))
                j++; // j is now index of child with greater value
            if (heap[pos] < heap[j]) return;
            else swap(heap[pos], heap[j]);
            pos = j;  // Move down
        }
    }
    ```
    ```

Removing the Priority Value

- What happens when we remove the priority value?
  - The priority value is stored at the root
- Choose the last leaf to replace the root, then sift down
  - Why choose the last leaf?

```
Comparable removePriority(Heap * h){
    //Check for empty heap
    if (numVertices == 0)
        return ;
    //Swap the root with last leaf
    Comparable priority = heap[0];
    h->heap[0] = h->heap[n - 1];
    h->heap[n - 1] = priority;
    //shrink heap by one node
    n--;
    //sift new root down
    siftDown(0);
    return priority;
}
```

Classwork

From BST to Heaps