Announcements

Program#1 ⇒ Review

- Mean 85; median 100; high 100
- Tested with no input, some lines of input, and 1 million+ lines

Priority Queue

Operations

- heapify (O(lgn)) ⇒ Preserve heap property
- build_heap(O(n)) ⇒ Array to heap
- heap_sort (O(nlgn)) ⇒ Heap to sorted array
- heap_insert (O(lgn)) ⇒ Add new value
- extract_max (O(lgn)) ⇒ Remove top/front value (max heap)
- increase_key (O(lgn)) ⇒ Promote

Support functions

- swap(i,j) ⇒ Swap A[i] and A[j]
- parent(i)
- left(i)
- right(i)
- index_of_max(i, j, k) ⇒ Return index of max A[i], A[j], A[k]

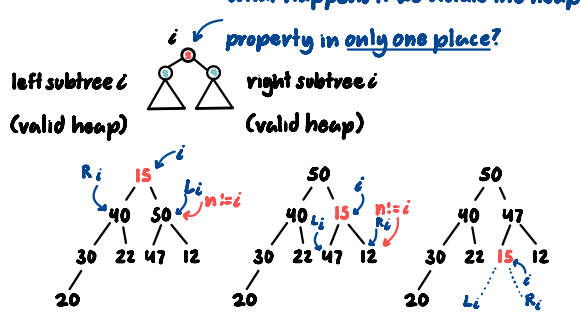
always valid ← i, j → Maybe invalid

Heapify ⇒ Take a value which is out of place and move it

- Heap property ⇒ A[i] ≤ A[parent(i)]
- Pseudo code

```
heapify(i) {
    n = index_of_max(i, left(i), right(i));
    if (n != i) {
        swap(A[i], A[n])   //swap elements
        heapify(n);        //call recursively
    }
}
```

What happens if we violate the heap property in only one place?

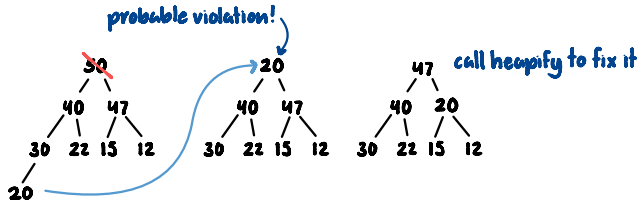


Extract max ⇒ Remove front element from heap

- Remember to check the size, dont crash when empty
- The max will be the first element
- Return and remove first element, fix heap with heapify()

- Pseudo code

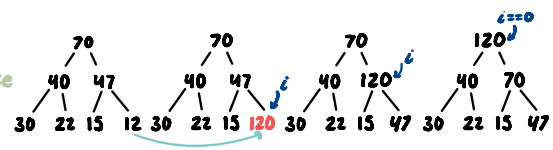
```
extract_max(){
    //check for valid size
    int max = A[0]          //O(1)
    A[0]=A[size-1]          //O(1)
    size--;                 //O(1)
    heapify(0);             //O(lgn), fix the the possible violation
    return max;             //O(1)
}
```



Increase key ⇒ Promote a value

- Pseudo code (index, key)

```
increase_key(i, k) {
    // Bounds check, check index, check k is an increase
    if (A[i]>= k) return;
    A[i] = k;    //promote A[i] to key (k) value
    while (i >0 && A[parent(i)] < A[i]) {    // Move value up until its in the right spot
        swap(i, parent(i))
        i=parent(i);
    }
}
```



Heap insert

- Assumes we have a heap
- The new element goes at the end, calls `increase_key()` to move it to correct location
- Pesudo code

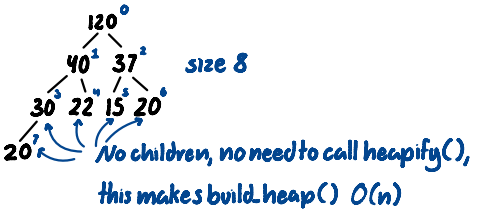
```
heap_insert(val) {
    // Bounds check
    size++;                     //O(1)
    A[size-1] = val-1           //O(1)
    increase_key(size-1,val);   //O(lg(n))
}
```

Build heap ⇒ Takes an array and converts it into a heap

- Need at least two elements to be a heap
- Required to code, not to use
- Takes any array (does not need to be sorted)

- Pesudo code

```
build_heap(){
    for(int i = size/2 -1; i >=0; i--) {
        heapify(i);     //O(lgn)
    }
}
```



Heap Sort ⇒ Sorts a heap into a sorted array

- Requires a heap
- Inverts the order, if a max heap, sorts in accending order
- Pesudo code

```
heap_sort(){
    for(int i=size-1; i>0; i--){
        swap(0, i)
        size--;
        heapify(0);
    }
    //Gives a sorted array and the origional heap is intact
    //Should use two arrays. Take the heap array and copy it (only once), run heapify on that one
    //Sorted array must be the exact size of the sorted values
    return pair(sorted_array, size);
}
```