Topics for Today

- This week's lab
  - Majority Element
  - Priority Queue
  - kth Largest Element
  - Min Heap

Upcoming Dates

Thursday 18 - No class - Stuvac
Friday 19 - Assignment 1 Due

Thursday 25 - No class - Anzac Day

Monday 29 - Assignment 2 Released
…
Week 13 - We make up our class lost on Anzac Day

Majority Element

We are doing another hashmap example, to help you remember this handy data structure.
Given a vector of length *n*, there is a number which appears in more than half its elements, return this number.

```
std::unordered_map<int, int> map {};
```
<key, value>

Find a way to count how many times each element appears, but you can't use a vector because you don't know what range to make the vector.

And you can write a for-each loop like so:
```
for (int i : vec){}
```
Each iteration it will give you `i`, the next item in `vec`.
Equivalent to `for (i=0; …) vec[i]`

std Priority Queue

A priority queue is a container for any number of elements, that is self-sorting. Each time you insert/remove an element, it sorts itself in O(logn) to have the priority element at the top. Priorities are usually largest/smallest.
The standard library includes an implementation of a *priority queue*, you can play around with it in this exercise. Next activity uses it for a problem.

```
std::priority_queue<int> queue {};
```

Priority Queues have the following functions:

- `void push(T val)` – Add `val` to the top of the stack.
- `void pop()` – Remove the top value on the stack.
- `T& top()` – Return a reference to the top value on the stack.
- `int size()` – Return the number of elements in the stack.
- `bool empty()` – Return whether or not the stack is empty.

kth Largest Element

We want to find the kth largest element in a vector.
ie in [0,5,3,6,1] the 1st largest element is 6, 2nd largest element is 5, and so on.

It is more efficient to do this with a *min priority queue* O(n*log(k+1)) than by sorting a vector/array O(n*log(n)).

Standard is a *max* priority queue:

```
std::priority_queue<int, int> prio_queue {};
```

Min priority queue:

```
std::priority_queue<int, int, std::greater<int> > pq {};
```

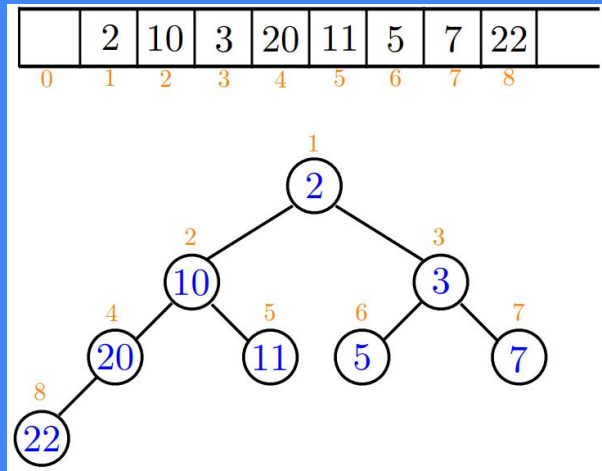
How can you use the queue to get the kth largest element, instead of sorting?
People usually have different solutions to this problem, let's compare

Min Heap

A min-heap (low-priority queue) works by ordering a binary tree on each insertion/deletion.
The tree follows three simple rules:
1- Each node is larger than its parent node.
Then, to maintain this ordering,

2- On insertion, add the new element to the last position in the tree, and "*swim*" upwards until rule 1 is met.

3- On deletion, swap the root node with the last node in the tree, then "*sink*" down as far as necessary to maintain the ordering of rule 1.
We also need to maintain the tight completeness of the graph while doing this.



Min Heap

We are going to use a vector to implement our own min-heap (low-priority queue).

This involves writing both internal functions to aid us, and external functions that are used by the grader to mark us.

Troy likes to 1-index his binary trees so they are a little easier to index, you can use the following notation:

Start off by writing the helper functions,
then the external functions,
so you know what you want from your internal functions.

Remember Sink and Swim from the lecture. pp. 40, pp. 47.

```
unsigned left(unsigned i){
    return 2*i;
}

unsigned right(unsigned i){
    return 2*i+1;
}

unsigned parent(unsigned i){
    // floor(i/2)
    return i >> 1;
}
```