Data structures & algorithms

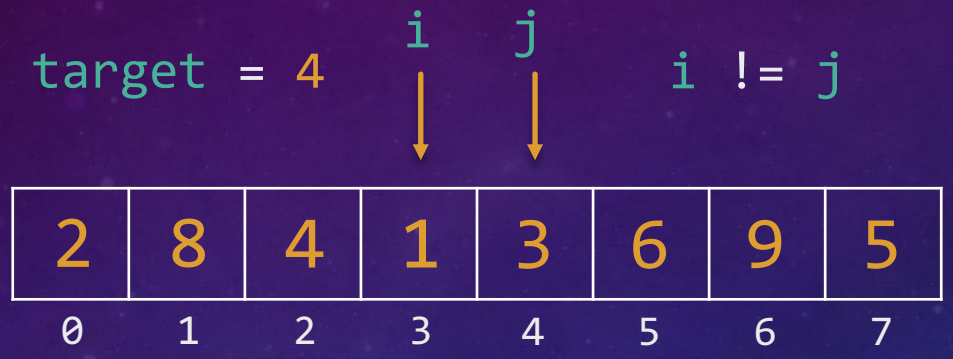
Tutorial 7

Lesson overview

- Recap of important topics from last week
- std::unordered_map
- Majority Element
- Priority Queue
- Kth Largest Element
- Min Heap

A quick recap of important things from last week

Two Sum



- Two Sum problem is one of the most popular easy Leetcode questions
- In the problem, you are given an un-ordered vector and a target. The goal is to find two numbers in the vector that when added together equal the target. These two numbers must also be in differing indexes within the array
- Once two numbers have been found that add up to the target, output the pair of indices
- In the example above, the pair of indices {3, 4} add up to our target, so we return that

Two Sum

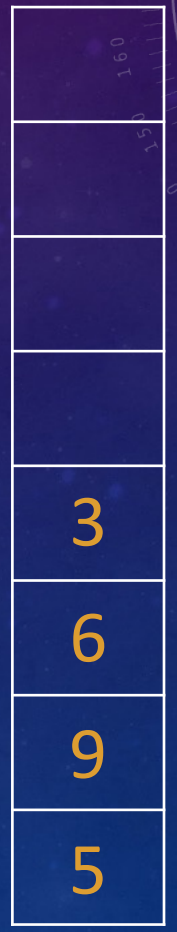
```
for(int i = 0; i < vec.size(); ++i){
    for(int j = 0; j < vec.size(); ++j){

    }
}
```

```
std::unordered_map<int, int>
```

- There are a couple of ways to find the solution to this problem, you could try:
  - Brute force with a nested for-loop with a time complexity of $O(n^2)$
  - Using a std::unordered_map and do it within a single loop, with a time complexity of O(n) and a space complexity of O(n)

Implement a Stack with a Vector

- A stack is essentially a vector with limited operations, it follows the last in, first out (LIFO) principle. Meaning that the last element put on the stack, will be the first element to be removed from it
- You can think the idea of a stack like a stack of pancakes or plates, where to add a pancake/plate it would be to the top of the stack… and to remove one it would also be at the top
- So it's as if someone took a vector and flipped it to stand up vertically



std::stack

Main operations for `std::stack`

```
std::stack<int> myStack{4, 5, 9, 10}; // Initialize with values
```

- `myStack.top()`: accesses element at the top of the stack
  - `myStack.top() // returns 10`
- `myStack.push(value)`: pushes `value` to the top of the stack
  - `myStack.push(13) // stack now contains {4, 5, 9, 10, 13}`
- `myStack.pop()`: removes element at the top of the stack
  - `myStack.pop() // vector now contains {4, 5, 9}`
- `myStack.size()`: returns the size of the stack
  - `myStack.size() // returns 4`
- `myStack.empty()`: empties the stack
  - `myStack.empty() // stack now contains {} (0 elements)`

- As you can see, it's like someone removed half of the operations from a vector.
- The goal of the exercise is to implement these operations of a stack using a vector

Stack Calculator

{‘4’, ‘8’, ‘3’, ‘*’, ‘+’}

sumStack

- In the evalRPN function, it is passed a vector containing list of tokens. This vector is our expression in postfix notation
- You will need to initialize a stack that contains ints (you can honestly just use a vector), this stack will store the results produced throughout the algorithm, and the result of the expression
- You will need to loop through the tokens and check whether it is a number, or not. If it is you will have to push it to the top of the stack (don’t forget to cast it to an int).

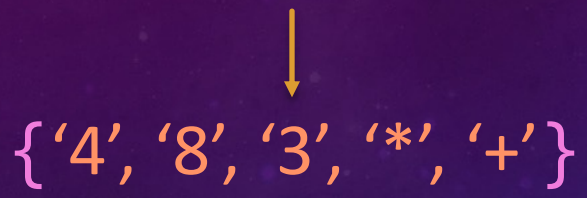
Stack Calculator

{‘4’, ‘8’, ‘3’, ‘*’, ‘+’}

sumStack

- In the evalRPN function, it is passed a vector containing list of tokens. This vector is our expression in postfix notation
- You will need to initialize a stack that contains ints (you can honestly just use a vector), this stack will store the results produced throughout the algorithm, and the result of the expression
- You will need to loop through the tokens and check whether it is a number, or not. If it is you will have to push it to the top of the stack (don’t forget to cast it to an int).

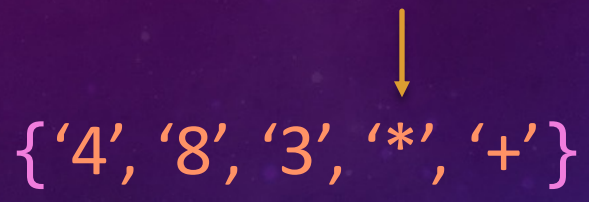
Stack Calculator



sumStack

- In the evalRPN function, it is passed a vector containing list of tokens. This vector is our expression in postfix notation
- You will need to initialize a stack that contains ints (you can honestly just use a vector), this stack will store the results produced throughout the algorithm, and the result of the expression
- You will need to loop through the tokens and check whether it is a number, or not. If it is you will have to push it to the top of the stack (don’t forget to cast it to an int).

Stack Calculator

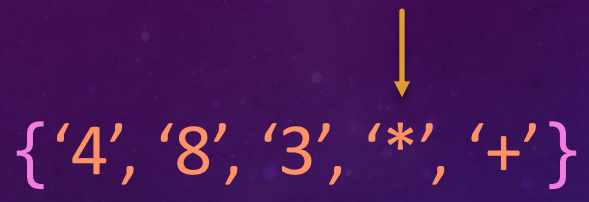



```
int value1 = sumStack.back();
int value2 = *(sumStack.end()-2);
```

sumStack

- In the evalRPN function, it is passed a vector containing list of tokens. This vector is our expression in postfix notation
- You will need to initialize a stack that contains ints (you can honestly just use a vector), this stack will store the results produced throughout the algorithm, and the result of the expression
- You will need to loop through the tokens and check whether it is a number, or not. If it is you will have to push it to the top of the stack (don’t forget to cast it to an int). If it is an operator rather than a number, you take the two last numbers

Stack Calculator

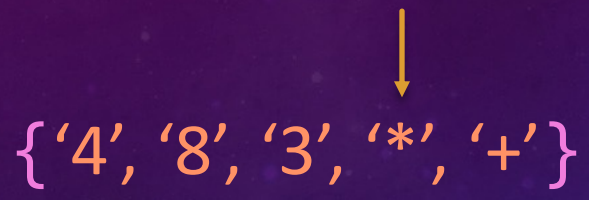



```
int value1 = sumStack.back();
int value2 = *(sumStack.end()-2);
int result = value1 * value2;
```

sumStack

- In the evalRPN function, it is passed a vector containing list of tokens. This vector is our expression in postfix notation
- You will need to initialize a stack that contains ints (you can honestly just use a vector), this stack will store the results produced throughout the algorithm, and the result of the expression
- You will need to loop through the tokens and check whether it is a number, or not. If it is you will have to push it to the top of the stack (don’t forget to cast it to an int). If it is an operator rather than a number, you take the two last numbers, and perform the necessary operation.

Stack Calculator

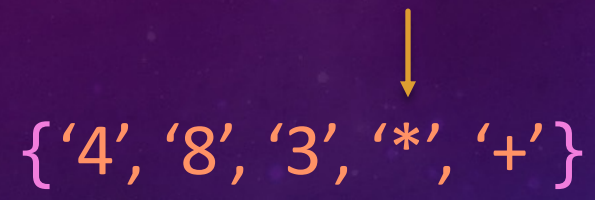



```
int value1 = sumStack.back();
int value2 = *(sumStack.end()-2);
int result = value1 * value2;
```

sumStack

- In the evalRPN function, it is passed a vector containing list of tokens. This vector is our expression in postfix notation
- You will need to initialize a stack that contains ints (you can honestly just use a vector), this stack will store the results produced throughout the algorithm, and the result of the expression
- You will need to loop through the tokens and check whether it is a number, or not. If it is you will have to push it to the top of the stack (don’t forget to cast it to an int). If it is an operator rather than a number, you take the two last numbers, and perform the necessary operation. Remove the last two elements from the sumStack…

Stack Calculator

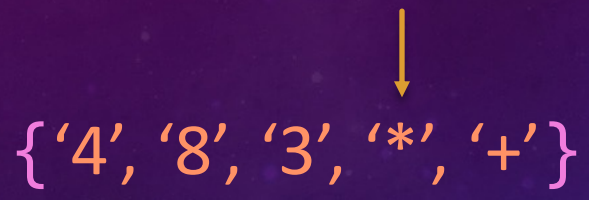



```
int value1 = sumStack.back();
int value2 = *(sumStack.end()-2);
int result = value1 * value2;
```

sumStack

- In the evalRPN function, it is passed a vector containing list of tokens. This vector is our expression in postfix notation
- You will need to initialize a stack that contains ints (you can honestly just use a vector), this stack will store the results produced throughout the algorithm, and the result of the expression
- You will need to loop through the tokens and check whether it is a number, or not. If it is you will have to push it to the top of the stack (don’t forget to cast it to an int). If it is an operator rather than a number, you take the two last numbers, and perform the necessary operation. Remove the last two elements from the sumStack… and push the result on

Stack Calculator




```
int value1 = sumStack.back();
int value2 = *(sumStack.end()-2);
int result = value1 * value2;
```

- In the evalRPN function, it is passed a vector containing list of tokens. This vector is our expression in postfix notation
- You will need to initialize a stack that contains ints (you can honestly just use a vector), this stack will store the results produced throughout the algorithm, and the result of the expression
- You will need to loop through the tokens and check whether it is a number, or not. If it is you will have to push it to the top of the stack (don’t forget to cast it to an int). If it is an operator rather than a number, you take the two last numbers, and perform the necessary operation. Remove the last two elements from the sumStack… and push the result on. Then you just repeat until you have looped through the whole expression…

Stack Calculator


```
int value1 = sumStack.back();
int value2 = *(sumStack.end()-2);
int result = value1 + value2;
```

sumStack

- In the evalRPN function, it is passed a vector containing list of tokens. This vector is our expression in postfix notation
- You will need to initialize a stack that contains ints (you can honestly just use a vector), this stack will store the results produced throughout the algorithm, and the result of the expression
- You will need to loop through the tokens and check whether it is a number, or not. If it is you will have to push it to the top of the stack (don’t forget to cast it to an int). If it is an operator rather than a number, you take the two last numbers, and perform the necessary operation. Remove the last two elements from the sumStack… and push the result on. Then you just repeat until you have looped through the whole expression…

Stack Calculator

{‘4’, ‘8’, ‘3’, ‘*’, ‘+’}

```
int value1 = sumStack.back();
int value2 = *(sumStack.end()-2);
int result = value1 + value2;
```

sumStack

- In the evalRPN function, it is passed a vector containing list of tokens. This vector is our expression in postfix notation
- You will need to initialize a stack that contains ints (you can honestly just use a vector), this stack will store the results produced throughout the algorithm, and the result of the expression
- You will need to loop through the tokens and check whether it is a number, or not. If it is you will have to push it to the top of the stack (don’t forget to cast it to an int). If it is an operator rather than a number, you take the two last numbers, and perform the necessary operation. Remove the last two elements from the sumStack… and push the result on. Then you just repeat until you have looped through the whole expression… the final result is now at the top of the stack

Lab time!

std::unordered_map

```
std::unordered_map<keyType, valueType> myMap{}
```

- An `unordered_map` is an extremely useful data structure when it comes to lowering the time complexity of your algorithms. It's C++'s version of the HashMap from Java, and can be included in your cpp file with `#include <unordered_map>`

std::unordered_map

```
std::unordered_map<keyType, valueType> myMap{}
```

- An unordered_map is an extremely useful data structure when it comes to lowering the time complexity of your algorithms. It's C++'s version of the HashMap from Java, and can be included in your cpp file with #include <unordered_map>
- The reason it is so useful is that it offers on average a time complexity of O(1) to search for values within the unordered_map

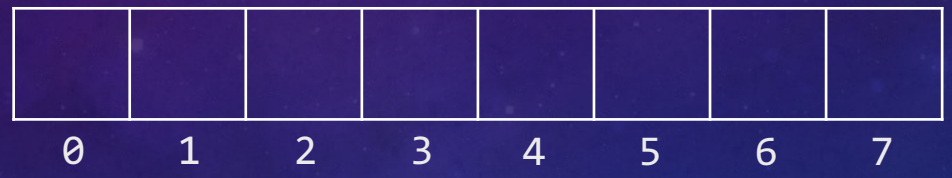
std::unordered_map

```
std::unordered_map<keyType, valueType> myMap{}
```

- An unordered_map is an extremely useful data structure when it comes to lowering the time complexity of your algorithms. It's C++'s version of the HashMap from Java, and can be included in your cpp file with `#include <unordered_map>`
- The reason it is so useful is that it offers on average a time complexity of O(1) to search for values within the unordered_map
- How is it so quick? Well , the underlying data structure for an unordered_map is really just a C++ array, and when we insert a `{key, value}` pair, the key is passed through a hash function to create its index within the array

std::unordered_map

```
std::unordered_map<std::string, int> myMap{}
```



- Here is a quick example, say we wanted our to `unordered_map` contain `{std::string, int}` pairs.

std::unordered_map

```
std::unordered_map<std::string, int> myMap{}
```

```
"Tom's age" -> int hashFunction()
```

- Here is a quick example, say we wanted our to `unordered_map` contain `{std::string, int}` pairs. Well when we insert `{"Tom's age", 25}`, the key `"Tom's age"` gets passed into a hashing function and will output some index, in our case lets say 4.

std::unordered_map

```
std::unordered_map<std::string, int> myMap{}
```

```
"Tom's age" -> int hashFunction()
```

- Here is a quick example, say we wanted our to `unordered_map` contain `{std::string, int}` pairs. Well when we insert `{"Tom's age", 25}`, the key `"Tom's age"` gets passed into a hashing function and will output some index, in our case lets say 4. Then the value gets normally inserted into the array!

std::unordered_map

```
std::unordered_map<std::string, int> myMap{}
```

```
"Tom's age" -> int hashFunction()
```

- Here is a quick example, say we wanted our to `unordered_map` contain `{std::string, int}` pairs. Well when we insert `{"Tom's age", 25}`, the key `"Tom's age"` gets passed into a hashing function and will output some index, in our case lets say 4. Then the value gets normally inserted into the array!
- Similarly, to retrieve value from the `unordered_map`, it will go through the same process, except rather than inserting something it will retrieve the value associated with its hash that is contained within the `unordered_map`

std::unordered_map

Main operations for std::unordered_map

```
std::unordered_map<std::string, int> myMap{}; // Initialize an unordered map
```

- myMap[key]: accesses element with key, also can be used to insert elements into myMap, i.e., myMap[key] = value
- myMap.at(key): accesses element with key
- myMap.contains(): removes the max/min element from the priority queue
- maxPQ.insert({key, value}): inserts key with associated value into myMap
- myMap.contains(key): checks if myMap contains the specified key
- myMap.clear(): clears all contents of myMap
- maxPQ.size(): checks the size of myMap
- myMap.empty(key): checks if is myMap empty or not

Majority Element

4	2	4	1	4	4	9	4

- In this problem, you are given a vector that contains ints with a size n, and you are guaranteed that there is one element that appears more than n / 2 times.

Majority Element

- In this problem, you are given a vector that contains ints with a size n, and you are guaranteed that there is one element that appears more than n / 2 times. So in the above vector, with a size n of 8, there should be an element that appears more than 4 times… This element would be 4.

Majority Element

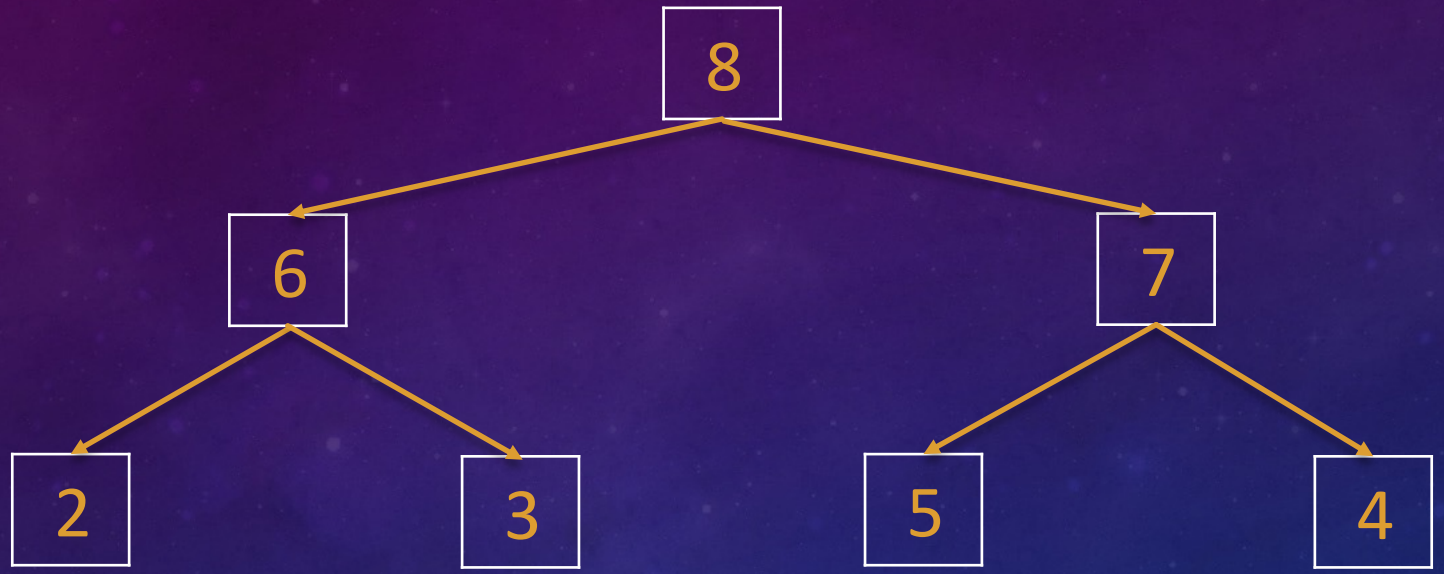
- In this problem, you are given a vector that contains ints with a size n, and you are guaranteed that there is one element that appears more than n / 2 times. So in the above vector, with a size n of 8, there should be an element that appears more than 4 times… This element would be 4.
- Now the goal of this problem is to implement a solution that finds this majority element, but it has to be in time O(n) on an average case. This means that for a size n vector, it should look at each element at most once, i.e., through one iteration…

Majority Element

- In this problem, you are given a vector that contains ints with a size n, and you are guaranteed that there is one element that appears more than n / 2 times. So in the above vector, with a size n of 8, there should be an element that appears more than 4 times… This element would be 4.
- Now the goal of this problem is to implement a solution that finds this majority element, but it has to be in time O(n) on an average case. This means that for a size n vector, it should look at each element at most once, i.e., through one iteration… try implementing a solution with an `unordered_map`, or try looking at the Boyer-Moore Majority Vote algorithm to see if you can implement that!

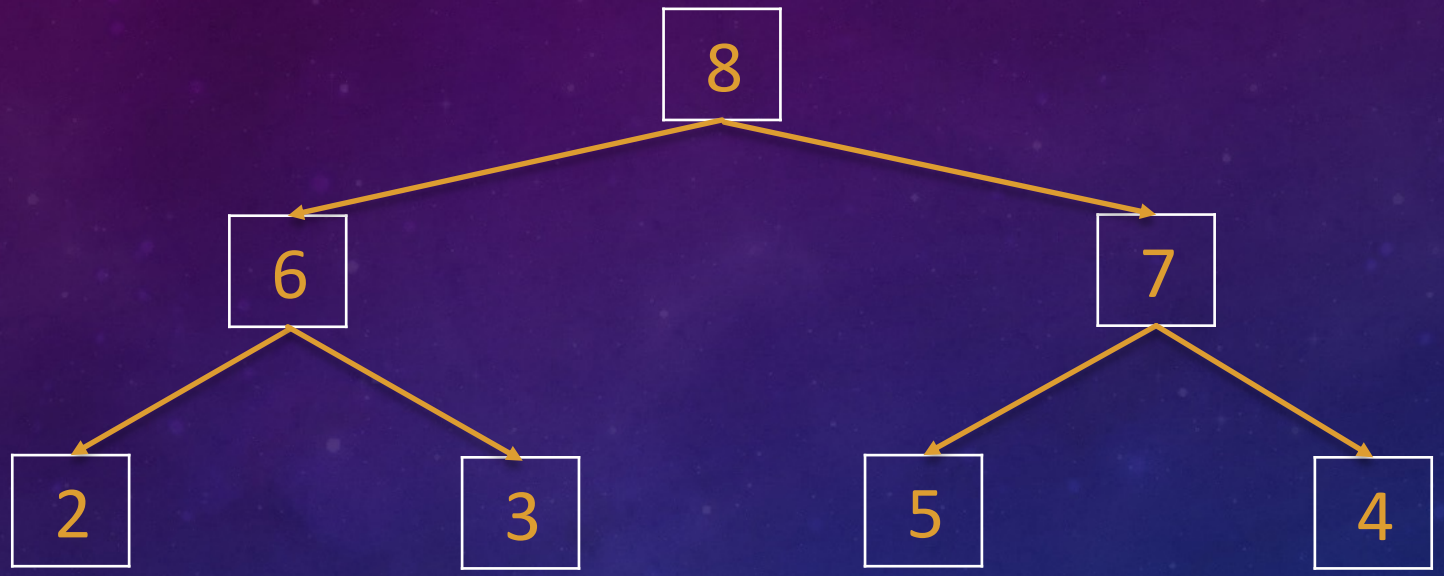
Give “Majority Element” and try and implement a solution to the problem!

Priority Queue



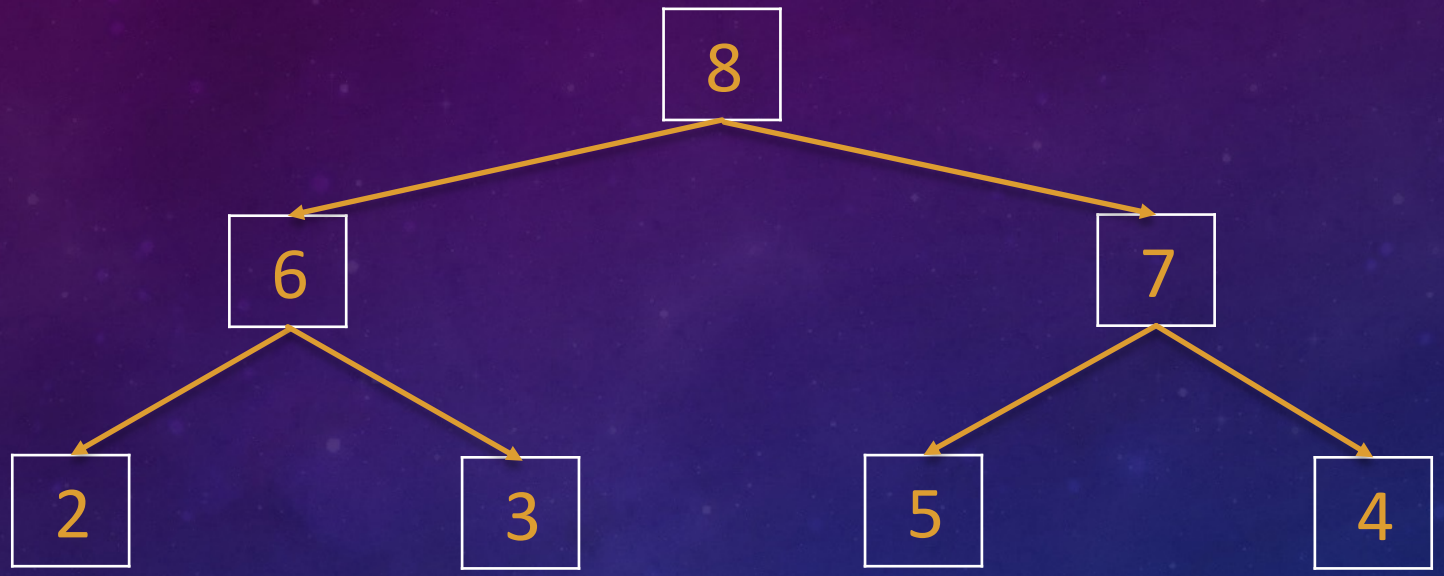
- A priority queue is a data structure that stores values based on a priority associated with each element

Priority Queue



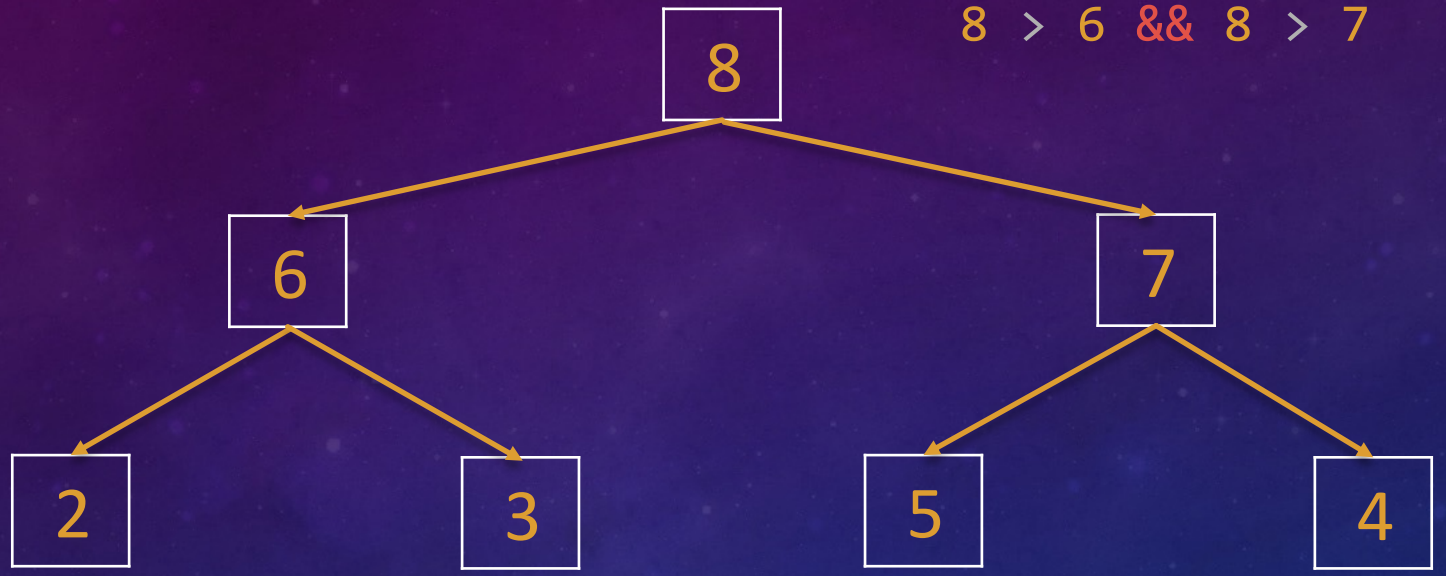
- A priority queue is a data structure that stores values based on a priority associated with each element
- Generally, this priority is based on the comparison between two int's, either prioritising small values or large values.

Priority Queue



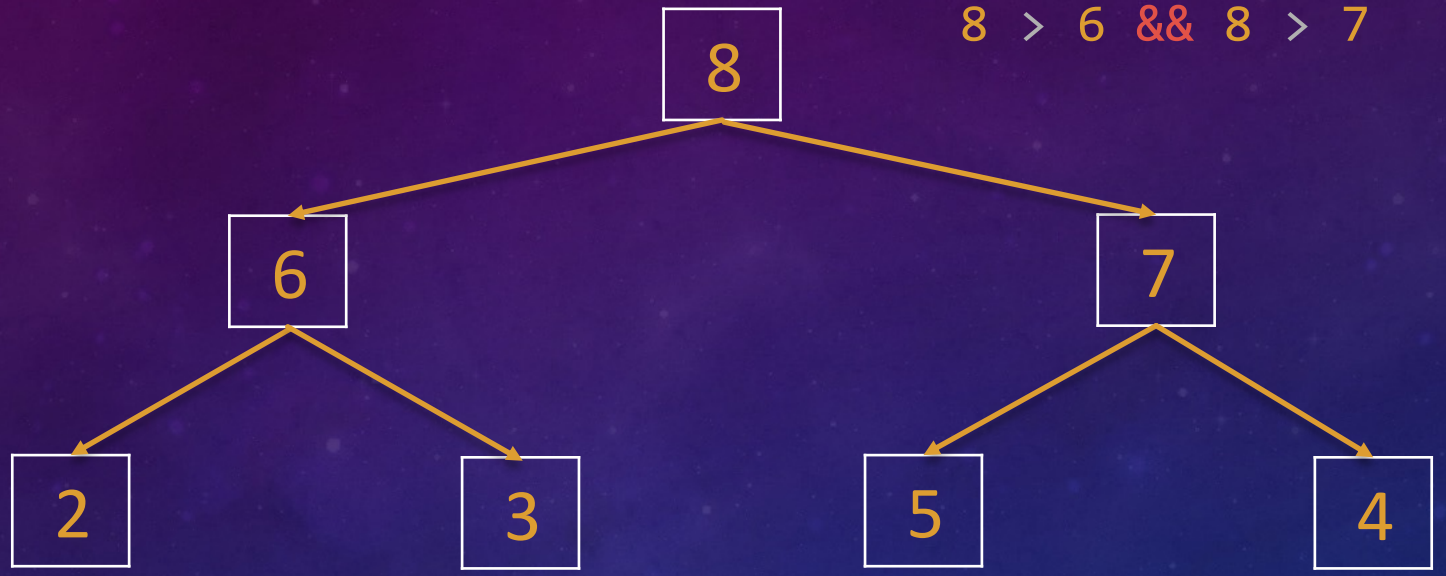
- A priority queue is a data structure that stores values based on a priority associated with each element
- Generally, this priority is based on the comparison between two int's, either prioritising small values or large values.
- Above is an example of a max priority queue, meaning it will prioritise storing larger values closer to the root. So the largest value in the priority queue is 8, and as you step down each layer, the values will always be smaller than the parent

Priority Queue



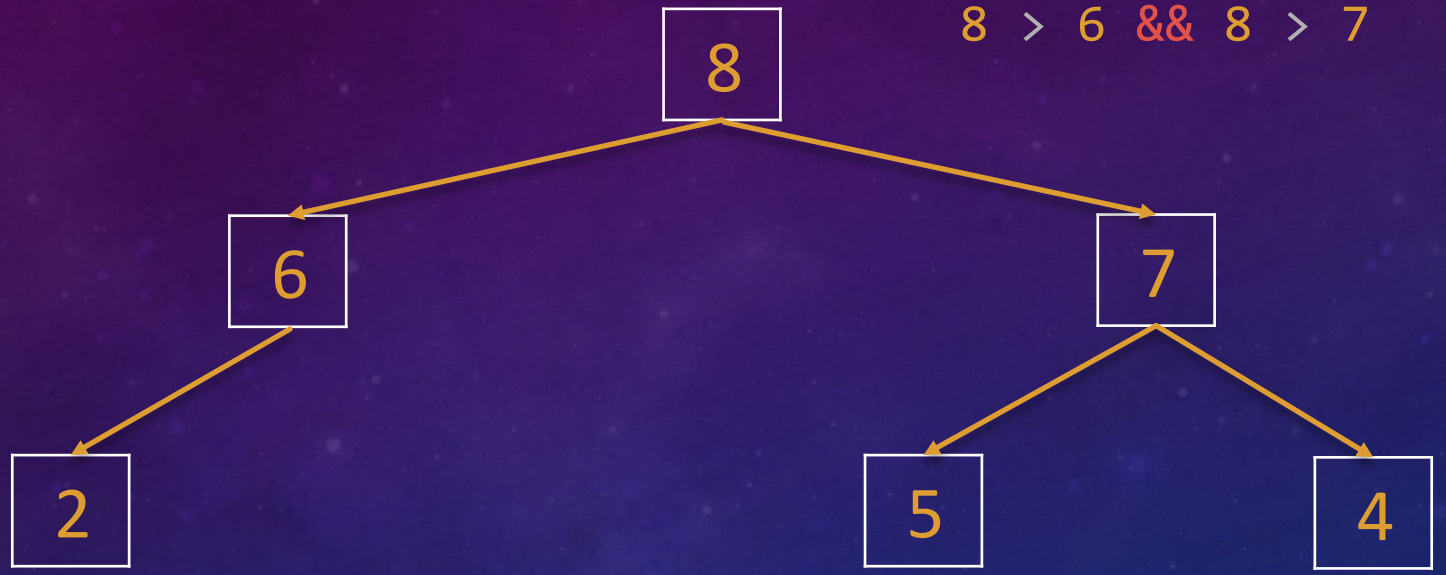
- A priority queue is a data structure that stores values based on a priority associated with each element
- Generally, this priority is based on the comparison between two int's, either prioritising small values or large values.
- Above is an example of a max priority queue, meaning it will prioritise storing larger values closer to the root. So the largest value in the priority queue is 8, and as you step down each layer, the values will always be smaller than the parent

Priority Queue



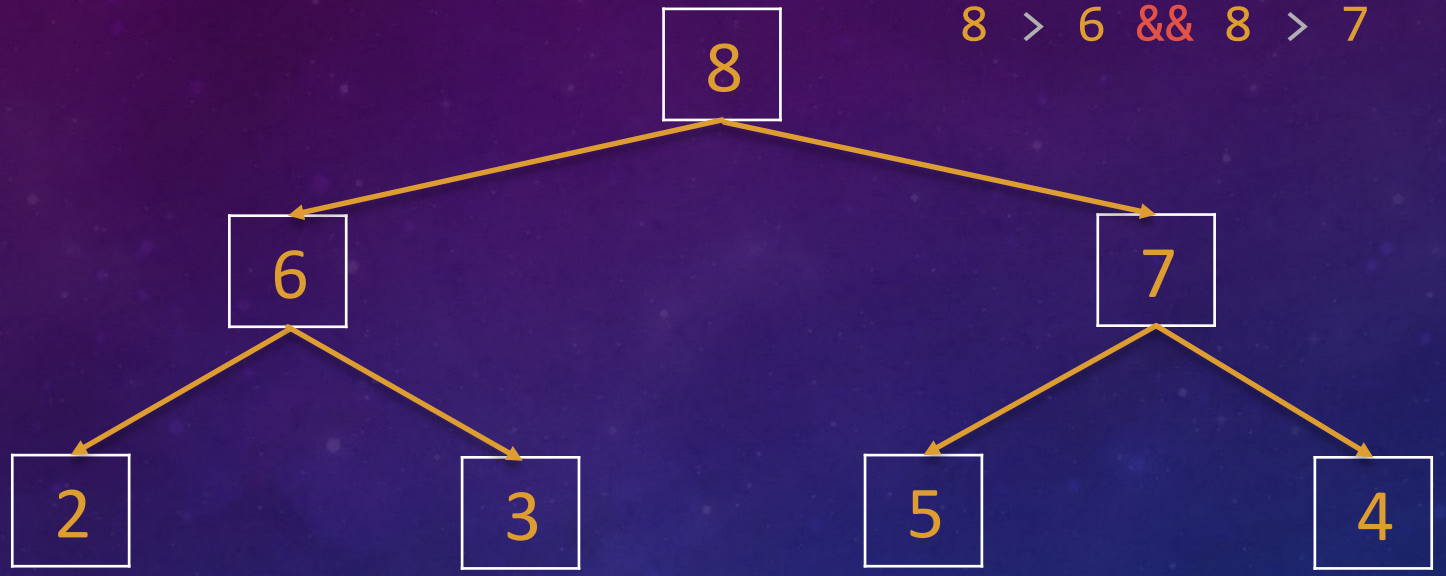
- A priority queue is a data structure that stores values based on a priority associated with each element
- Generally, this priority is based on the comparison between two int's, either prioritising small values or large values.
- Above is an example of a max priority queue, meaning it will prioritise storing larger values closer to the root. So the largest value in the priority queue is 8, and as you step down each layer, the values will always be smaller than the parent
- Priority queues must also satisfy two properties, the first is that it maintains a complete binary tree. This means a row is completely filled from left to right before a new row is started.

Priority Queue



- A priority queue is a data structure that stores values based on a priority associated with each element
- Generally, this priority is based on the comparison between two int's, either prioritising small values or large values.
- Above is an example of a max priority queue, meaning it will prioritise storing larger values closer to the root. So the largest value in the priority queue is 8, and as you step down each layer, the values will always be smaller than the parent
- Priority queues must also satisfy two properties, the first is that it maintains a complete binary tree. This means a row is completely filled from left to right before a new row is started.

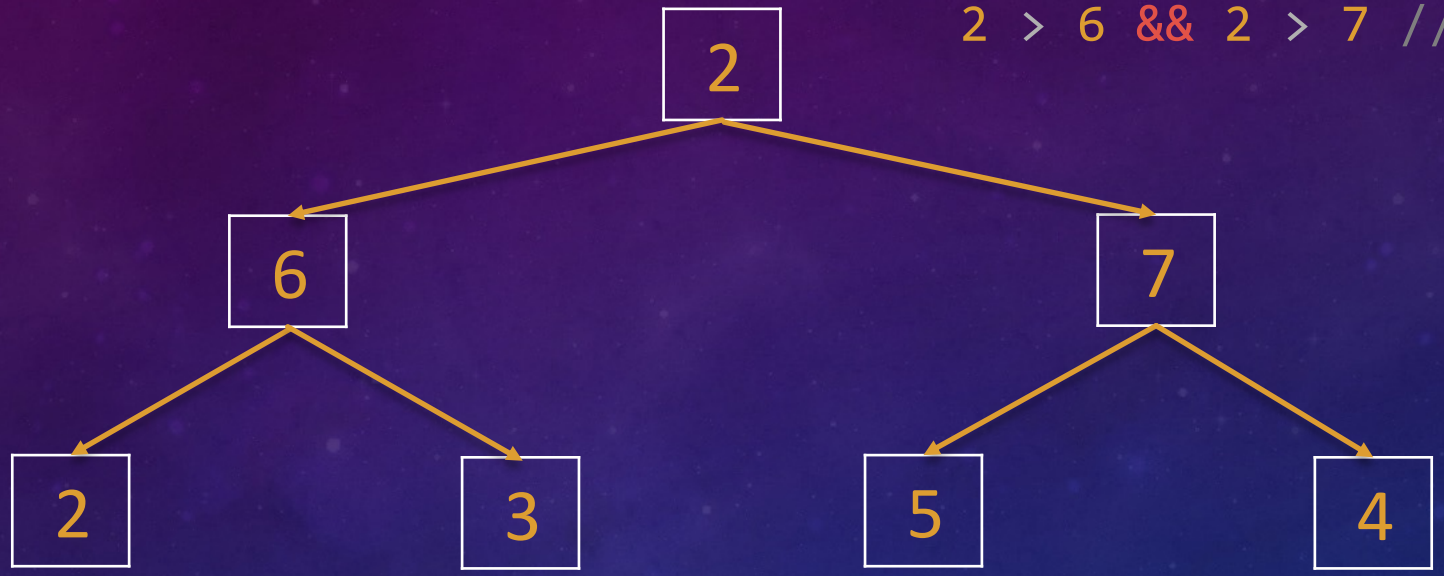
Priority Queue



- A priority queue is a data structure that stores values based on a priority associated with each element
- Generally, this priority is based on the comparison between two int's, either prioritising small values or large values.
- Above is an example of a max priority queue, meaning it will prioritise storing larger values closer to the root. So the largest value in the priority queue is 8, and as you step down each layer, the values will always be smaller than the parent
- Priority queues must also satisfy two properties, the first is that it maintains a complete binary tree. This means a row is completely filled from left to right before a new row is started. It must also satisfy the heap property, which means if it is a max priority queue, the parent should always be bigger than its children, similar to a min priority queue, except it must be smaller

Priority Queue

```
2 > 6 && 2 > 7 //false!
```



- A priority queue is a data structure that stores values based on a priority associated with each element
- Generally, this priority is based on the comparison between two int's, either prioritising small values or large values.
- Above is an example of a max priority queue, meaning it will prioritise storing larger values closer to the root. So the largest value in the priority queue is 8, and as you step down each layer, the values will always be smaller than the parent
- Priority queues must also satisfy two properties, the first is that it maintains a complete binary tree. This means a row is completely filled from left to right before a new row is started. It must also satisfy the heap property, which means if it is a max priority queue, the parent should always be bigger than its children, similar to a min priority queue, except it must be smaller

std::priority_queue

Main operations for `std::priority_queue`

```
std::priority_queue<int> maxPQ{}; // Initialize a max priority queue

std::priority_queue<int, std::vector<int>, std::greater<int>> minPQ{};
// Initialize a  min priority queue
```

- `maxPQ.top()`: accesses element at the top of the priority queue
- `maxPQ.push(value)`: pushes `value` into the priority queue
- `maxPQ.pop()`: removes the max/min element from the priority queue
- `maxPQ.size()`: returns the size of the priority queue
- `maxPQ.empty()`: empties the priority queue

Go noodle around in the “Priority Queue” exercise and see how they work !

Kth Largest Element

- In this next exercise, we will be using a priority queue to solve the kth largest element problem. In this problem, you are given a vector containing ints, and an integer k which represents the kth largest element in the vector.

Kth Largest Element

- In this next exercise, we will be using a priority queue to solve the kth largest element problem. In this problem, you are given a vector containing ints, and an integer k which represents the kth largest element in the vector.
- For example, if k = 2 then we are trying to find the 2nd largest element in the vector.

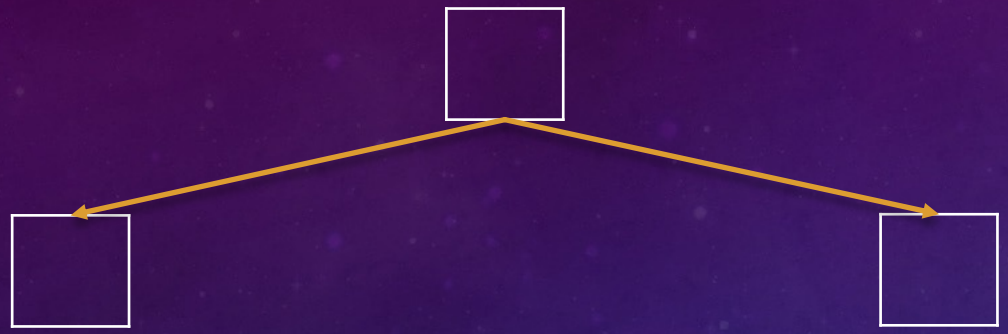
Kth Largest Element

- In this next exercise, we will be using a priority queue to solve the kth largest element problem. In this problem, you are given a vector containing ints, and an integer k which represents the kth largest element in the vector.
- For example, if k = 2 then we are trying to find the 2nd largest element in the vector. A way to find this is by sorting the vector, when sorted we can see that the 2nd largest element will be 8

Kth Largest Element

- In this next exercise, we will be using a priority queue to solve the kth largest element problem. In this problem, you are given a vector containing ints, and an integer k which represents the kth largest element in the vector.
- For example, if k = 2 then we are trying to find the 2nd largest element in the vector. A way to find this is by sorting the vector, when sorted we can see that the 2nd largest element will be 8
- But, we want to solve this using a priority queue, and not by sorting. How do we do this?

Kth Largest Element



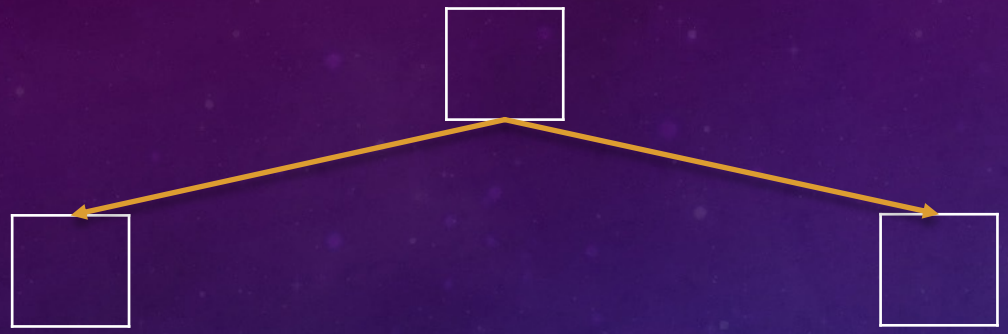
2	8	4	1	3	6	9	5
0	1	2	3	4	5	6	7

k = 2

```
myPQ.size() > k ? myPQ.pop()
```

- Well if we create a minimum priority queue, and make the size of the priority queue doesn't exceed our int k. The value at the top of our priority queue will be our kth largest value.

Kth Largest Element



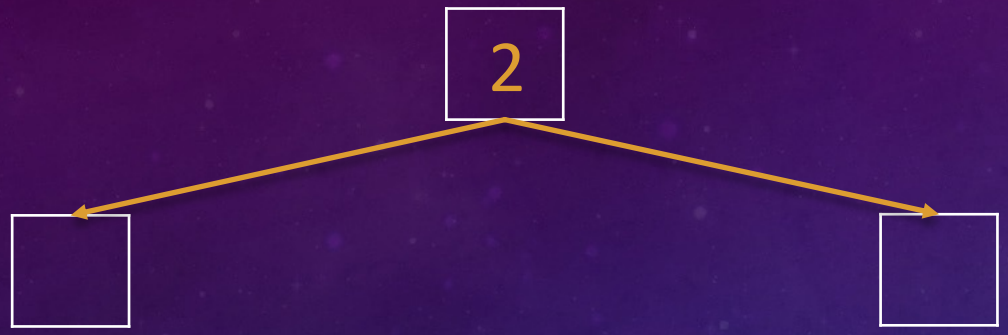
2	8	4	1	3	6	9	5
0	1	2	3	4	5	6	7

k = 2

```
myPQ.size() > k ? myPQ.pop()
```

- Well if we create a minimum priority queue, and make the size of the priority queue doesn't exceed our int k. The value at the top of our priority queue will be our kth largest value.
- Lets see this in action…

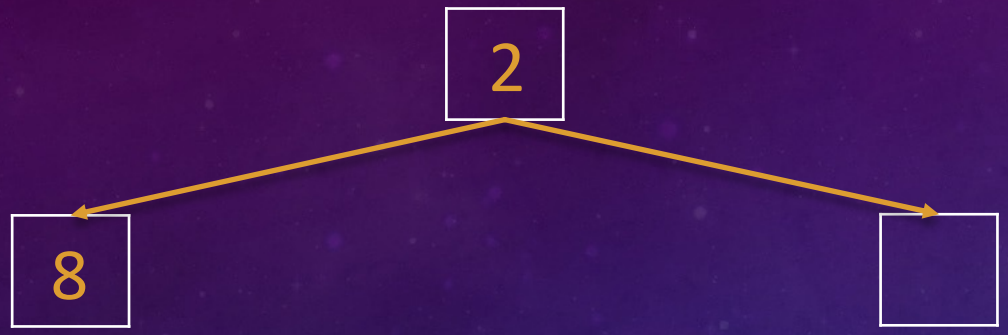
Kth Largest Element




```
myPQ.size() > k ? myPQ.pop()
```

- Well if we create a minimum priority queue, and make the size of the priority queue doesn't exceed our int k. The value at the top of our priority queue will be our kth largest value.
- Lets see this in action…

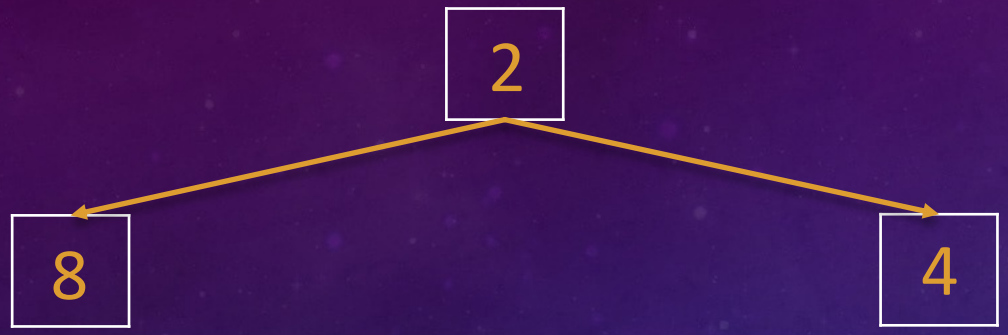
Kth Largest Element




```
myPQ.size() > k ? myPQ.pop()
```

- Well if we create a minimum priority queue, and make the size of the priority queue doesn't exceed our int k. The value at the top of our priority queue will be our kth largest value.
- Lets see this in action…

Kth Largest Element




```
myPQ.size() > k ? myPQ.pop()
```

- Well if we create a minimum priority queue, and make the size of the priority queue doesn't exceed our int k. The value at the top of our priority queue will be our kth largest value.
- Lets see this in action…

Kth Largest Element


```
myPQ.size() > k ? myPQ.pop()
```

- Well if we create a minimum priority queue, and make the size of the priority queue doesn't exceed our int k. The value at the top of our priority queue will be our kth largest value.
- Lets see this in action…

Kth Largest Element


```
myPQ.size() > k ? myPQ.pop()
```

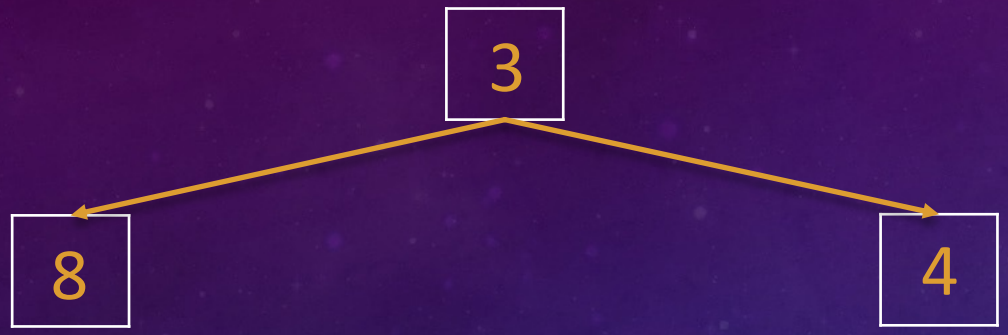
- Well if we create a minimum priority queue, and make the size of the priority queue doesn't exceed our int k. The value at the top of our priority queue will be our kth largest value.
- Lets see this in action…

Kth Largest Element


```
myPQ.size() > k ? myPQ.pop()
```

- Well if we create a minimum priority queue, and make the size of the priority queue doesn't exceed our int k. The value at the top of our priority queue will be our kth largest value.
- Lets see this in action…

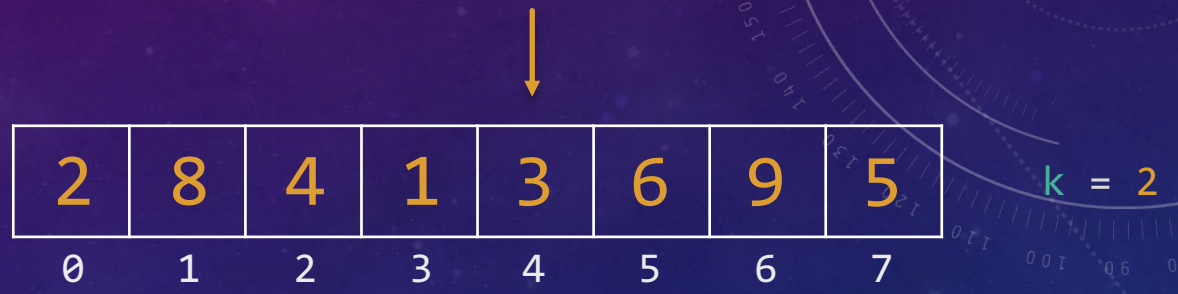
Kth Largest Element




```
myPQ.size() > k ? myPQ.pop()
```

- Well if we create a minimum priority queue, and make the size of the priority queue doesn't exceed our int k. The value at the top of our priority queue will be our kth largest value.
- Lets see this in action…

Kth Largest Element




```
myPQ.size() > k ? myPQ.pop()
```

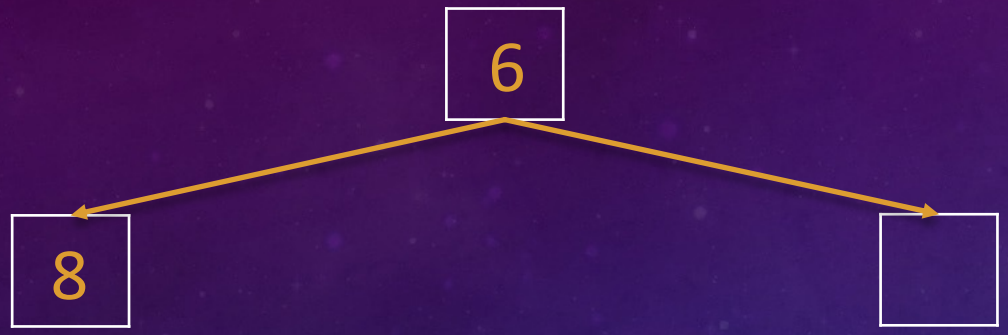
- Well if we create a minimum priority queue, and make the size of the priority queue doesn't exceed our int k. The value at the top of our priority queue will be our kth largest value.
- Lets see this in action…

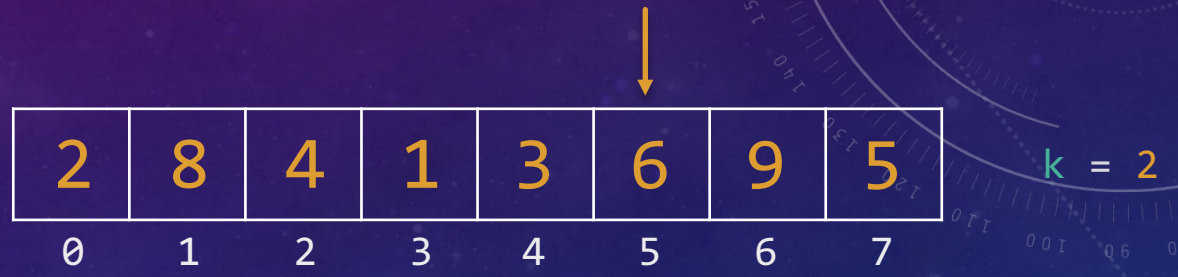
Kth Largest Element


```
myPQ.size() > k ? myPQ.pop()
```

- Well if we create a minimum priority queue, and make the size of the priority queue doesn't exceed our int k. The value at the top of our priority queue will be our kth largest value.
- Lets see this in action…

Kth Largest Element

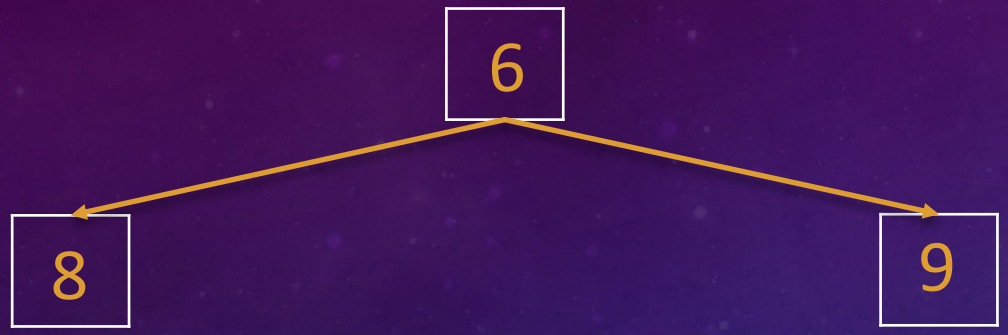





```
myPQ.size() > k ? myPQ.pop()
```

- Well if we create a minimum priority queue, and make the size of the priority queue doesn't exceed our int k. The value at the top of our priority queue will be our kth largest value.
- Lets see this in action…

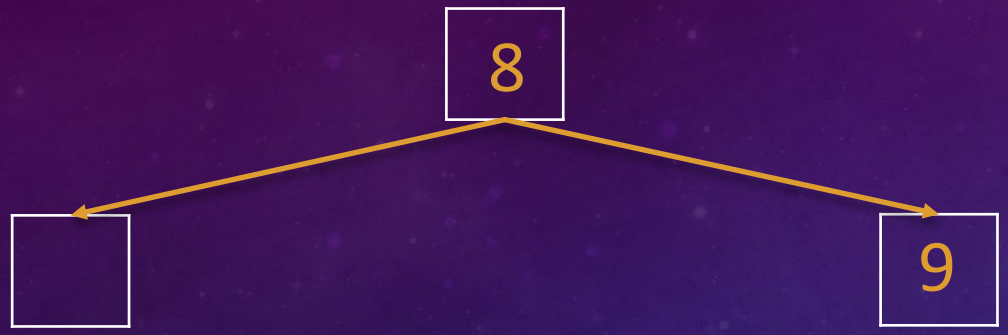
Kth Largest Element




```
myPQ.size() > k ? myPQ.pop()
```

- Well if we create a minimum priority queue, and make the size of the priority queue doesn't exceed our int k. The value at the top of our priority queue will be our kth largest value.
- Lets see this in action…

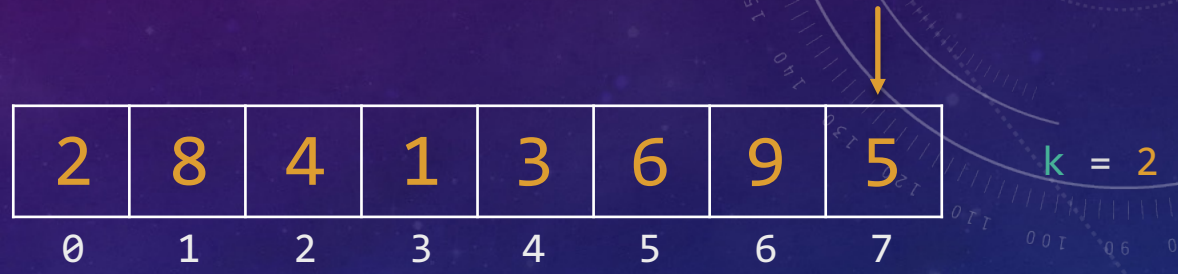
Kth Largest Element




```
myPQ.size() > k ? myPQ.pop()
```

- Well if we create a minimum priority queue, and make the size of the priority queue doesn't exceed our int k. The value at the top of our priority queue will be our kth largest value.
- Lets see this in action…

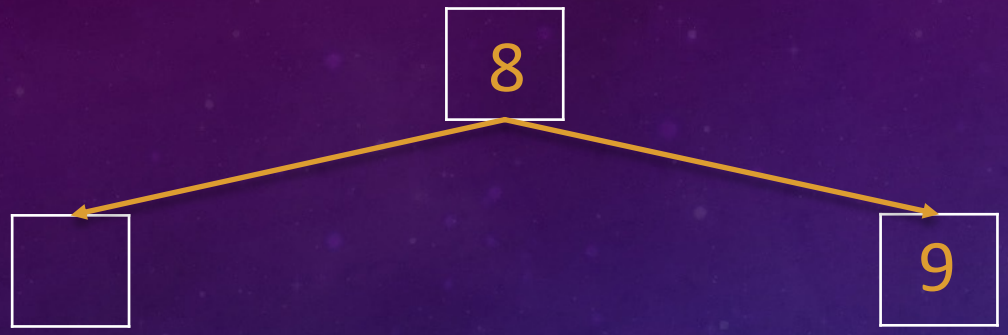
Kth Largest Element




```
myPQ.size() > k ? myPQ.pop()
```

- Well if we create a minimum priority queue, and make the size of the priority queue doesn't exceed our int k. The value at the top of our priority queue will be our kth largest value.
- Lets see this in action…

Kth Largest Element




```
myPQ.size() > k ? myPQ.pop()
```

- Well if we create a minimum priority queue, and make the size of the priority queue doesn't exceed our int k. The value at the top of our priority queue will be our kth largest value.
- Lets see this in action… now that the entire vector has passed through the priority queue, and the size of it has been maintained to only contain k elements. The kth largest element is now at the top of the priority queue

Give “Kth Largest Element” a go to try and implement your solution!

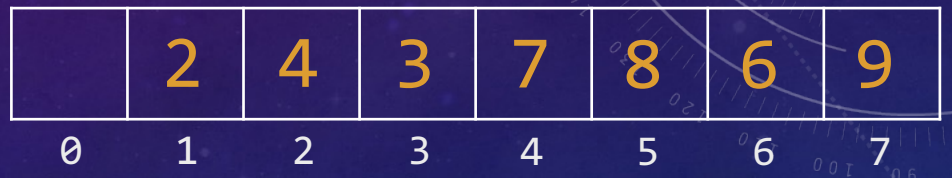
Min Heap

- In this next exercise, we'll be implementing a min heap. Rather than implementing it as a tree like above…

Min Heap

- In this next exercise, we'll be implementing a min heap. Rather than implementing it as a tree like above… we will implement it using a vector!

Min Heap



- In this next exercise, we'll be implementing a min heap. Rather than implementing it as a tree like above… we will implement it using a vector!
- Using an array structure allows for a very memory-efficient approach to implementing many tree-like data structures. Where each row of the structure on the left, is inserted one after another in the array

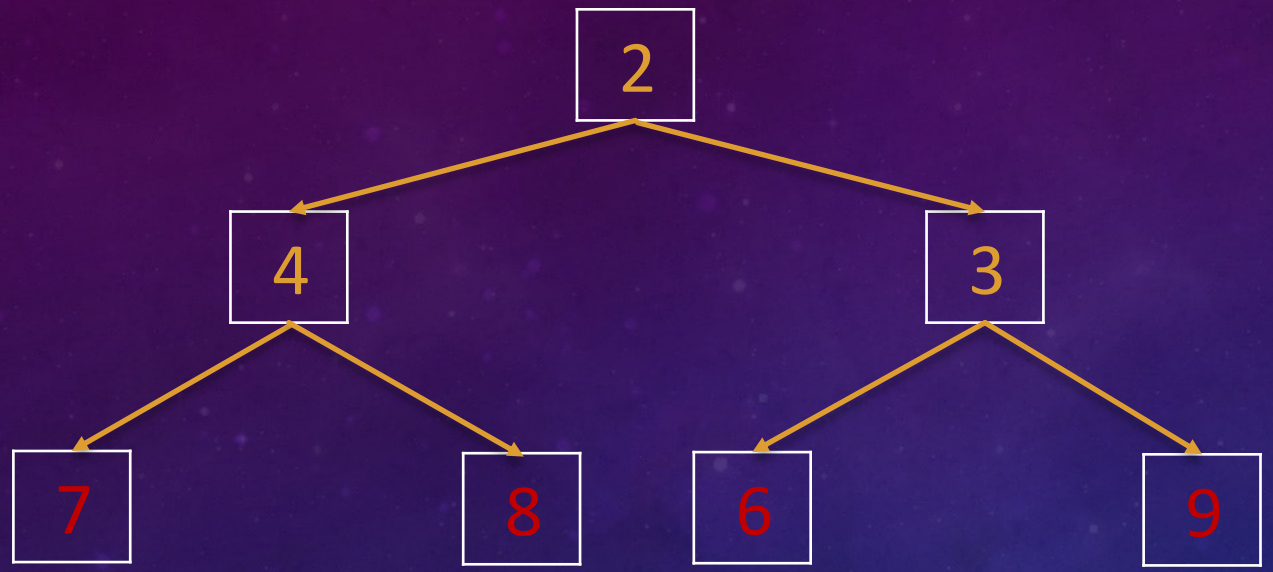
Min Heap

- In this next exercise, we'll be implementing a min heap. Rather than implementing it as a tree like above… we will implement it using a vector!
- Using an array structure allows for a very memory-efficient approach to implementing many tree-like data structures. Where each row of the structure on the left, is inserted one after another in the array

Min Heap

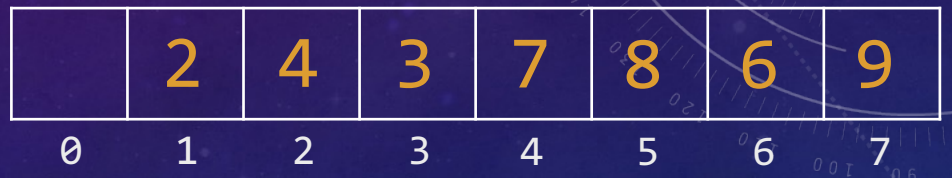
- In this next exercise, we'll be implementing a min heap. Rather than implementing it as a tree like above… we will implement it using a vector!
- Using an array structure allows for a very memory-efficient approach to implementing many tree-like data structures. Where each row of the structure on the left, is inserted one after another in the array

Min Heap



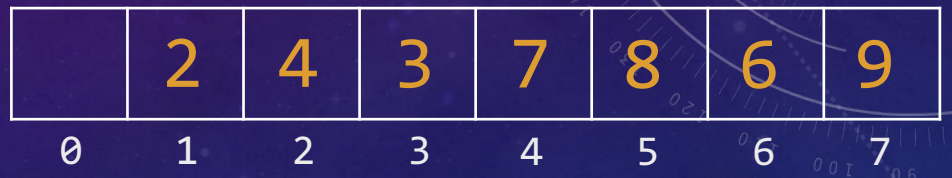
- In this next exercise, we'll be implementing a min heap. Rather than implementing it as a tree like above… we will implement it using a vector!
- Using an array structure allows for a very memory-efficient approach to implementing many tree-like data structures. Where each row of the structure on the left, is inserted one after another in the array

Min Heap



- In this next exercise, we'll be implementing a min heap. Rather than implementing it as a tree like above… we will implement it using a vector!
- Using an array structure allows for a very memory-efficient approach to implementing many tree-like data structures. Where each row of the structure on the left, is inserted one after another in the array
- Some useful helper functions have already been given to you. Like getting the index of the leftChild, rightChild and parent. Which will make implementing the more complex sink and swim functions easier.

Min Heap: sink



- Whenever you pop an element from the priority queue, you have to maintain the heap property by swapping the first and last element, and then sinking that element to the bottom.

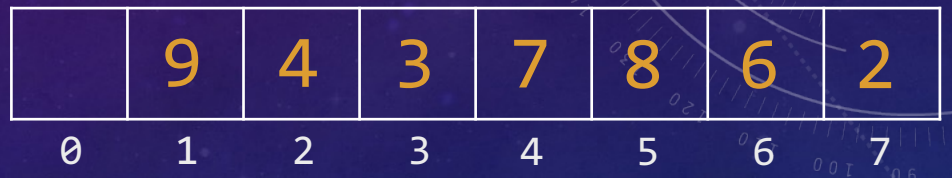
Min Heap: sink

- Whenever you pop an element from the priority queue, you have to maintain the heap property by swapping the first and last element, and then sinking that element to the bottom. Heres an example…

Min Heap: sink

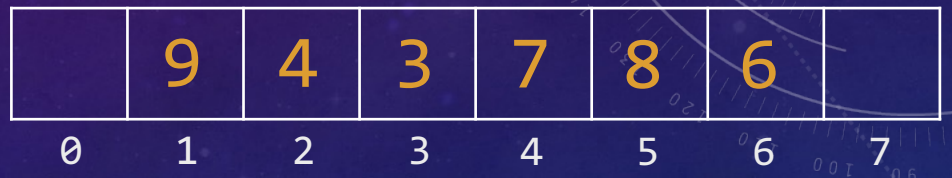
- Whenever you pop an element from the priority queue, you have to maintain the heap property by swapping the first and last element, and then sinking that element to the bottom. Heres an example… when I pop 2, I first swap 2 with the last element in the vector…

Min Heap: sink



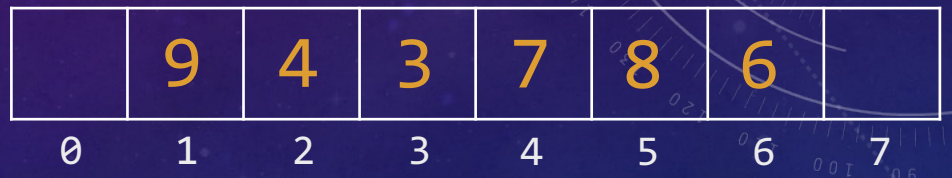
- Whenever you pop an element from the priority queue, you have to maintain the heap property by swapping the first and last element, and then sinking that element to the bottom. Heres an example… when I pop 2, I first swap 2 with the last element in the vector… after this is done, I can then remove 2 from the vector…

Min Heap: sink



- Whenever you pop an element from the priority queue, you have to maintain the heap property by swapping the first and last element, and then sinking that element to the bottom. Heres an example… when I pop 2, I first swap 2 with the last element in the vector… after this is done, I can then remove 2 from the vector…

Min Heap: sink



- Whenever you pop an element from the priority queue, you have to maintain the heap property by swapping the first and last element, and then sinking that element to the bottom. Heres an example… when I pop 2, I first swap 2 with the last element in the vector… after this is done, I can then remove 2 from the vector… but wait, now the heap property isn't maintained as 9 is bigger than 4 and 3!. This is when you sink 9 back into place, by swapping it with the smallest value of its children…

Min Heap: sink

- Whenever you pop an element from the priority queue, you have to maintain the heap property by swapping the first and last element, and then sinking that element to the bottom. Heres an example… when I pop 2, I first swap 2 with the last element in the vector… after this is done, I can then remove 2 from the vector… but wait, now the heap property isn't maintained as 9 is bigger than 4 and 3!. This is when you sink 9 back into place, by swapping it with the smallest value of its children…

Min Heap: sink

- Whenever you pop an element from the priority queue, you have to maintain the heap property by swapping the first and last element, and then sinking that element to the bottom. Heres an example… when I pop 2, I first swap 2 with the last element in the vector… after this is done, I can then remove 2 from the vector… but wait, now the heap property isn't maintained as 9 is bigger than 4 and 3!. This is when you sink 9 back into place, by swapping it with the smallest value of its children…

Min Heap: swim

- To push an element to the min heap, it follows a similar approach. When we push an element to the back of the vector, we call swim with its new size, what swim will do is ”swims” that new element up the min heap, until it is at its correct spot.

Min Heap: swim

- To push an element to the min heap, it follows a similar approach. When we push an element to the back of the vector, we call swim with its new size, what swim will do is ”swims” that new element up the min heap, until it is at its correct spot. So, for example, if we push 1 to the min heap…

Min Heap: swim

- To push an element to the min heap, it follows a similar approach. When we push an element to the back of the vector, we call swim with its new size, what swim will do is ”swims” that new element up the min heap, until it is at its correct spot. So, for example, if we push 1 to the min heap… we push it to the back of the vector…

Min Heap: swim

- To push an element to the min heap, it follows a similar approach. When we push an element to the back of the vector, we call swim with its new size, what swim will do is ”swims” that new element up the min heap, until it is at its correct spot. So, for example, if we push 1 to the min heap… we push it to the back of the vector… and now we call swim with the new size of the priority queue which is now 7…

Min Heap: swim

- To push an element to the min heap, it follows a similar approach. When we push an element to the back of the vector, we call swim with its new size, what swim will do is ”swims” that new element up the min heap, until it is at its correct spot. So, for example, if we push 1 to the min heap… we push it to the back of the vector… and now we call swim with the new size of the priority queue which is now 7… and all the swim function does, is compares whether the new value is smaller than its parent, if it is then it simply swaps the values and continues on that parents index.

Min Heap: swim

- To push an element to the min heap, it follows a similar approach. When we push an element to the back of the vector, we call swim with its new size, what swim will do is ”swims” that new element up the min heap, until it is at its correct spot. So, for example, if we push 1 to the min heap… we push it to the back of the vector… and now we call swim with the new size of the priority queue which is now 7… and all the swim function does, is compares whether the new value is smaller than its parent, if it is then it simply swaps the values and continues on that parents index. Lets see this in action!

Min Heap: swim

Each iteration, we check if the current value is smaller than its parent. If it is we swap, else we return.

- To push an element to the min heap, it follows a similar approach. When we push an element to the back of the vector, we call swim with its new size, what swim will do is ”swims” that new element up the min heap, until it is at its correct spot. So, for example, if we push 1 to the min heap… we push it to the back of the vector… and now we call swim with the new size of the priority queue which is now 7… and all the swim function does, is compares whether the new value is smaller than its parent, if it is then it simply swaps the values and continues on that parents index. Lets see this in action!

Min Heap: swim

Each iteration, we check if the current value is smaller than its parent. If it is we swap, else we return.

`1 < 6`?

- To push an element to the min heap, it follows a similar approach. When we push an element to the back of the vector, we call swim with its new size, what swim will do is ”swims” that new element up the min heap, until it is at its correct spot. So, for example, if we push 1 to the min heap… we push it to the back of the vector… and now we call swim with the new size of the priority queue which is now 7… and all the swim function does, is compares whether the new value is smaller than its parent, if it is then it simply swaps the values and continues on that parents index. Lets see this in action!

Min Heap: swim

Each iteration, we check if the current value is smaller than its parent. If it is we swap, else we return.

`1 < 6`? Swap!

- To push an element to the min heap, it follows a similar approach. When we push an element to the back of the vector, we call swim with its new size, what swim will do is ”swims” that new element up the min heap, until it is at its correct spot. So, for example, if we push 1 to the min heap… we push it to the back of the vector… and now we call swim with the new size of the priority queue which is now 7… and all the swim function does, is compares whether the new value is smaller than its parent, if it is then it simply swaps the values and continues on that parents index. Lets see this in action!

Min Heap: swim

Each iteration, we check if the current value is smaller than its parent. If it is we swap, else we return.

`1 < 6`? Swap!

- To push an element to the min heap, it follows a similar approach. When we push an element to the back of the vector, we call swim with its new size, what swim will do is ”swims” that new element up the min heap, until it is at its correct spot. So, for example, if we push 1 to the min heap… we push it to the back of the vector… and now we call swim with the new size of the priority queue which is now 7… and all the swim function does, is compares whether the new value is smaller than its parent, if it is then it simply swaps the values and continues on that parents index. Lets see this in action!

Min Heap: swim

Each iteration, we check if the current value is smaller than its parent. If it is we swap, else we return.

`1 < 3`?

- To push an element to the min heap, it follows a similar approach. When we push an element to the back of the vector, we call swim with its new size, what swim will do is ”swims” that new element up the min heap, until it is at its correct spot. So, for example, if we push 1 to the min heap… we push it to the back of the vector… and now we call swim with the new size of the priority queue which is now 7… and all the swim function does, is compares whether the new value is smaller than its parent, if it is then it simply swaps the values and continues on that parents index. Lets see this in action!

Min Heap: swim

Each iteration, we check if the current value is smaller than its parent. If it is we swap, else we return.

`1 < 3`? Swap!

- To push an element to the min heap, it follows a similar approach. When we push an element to the back of the vector, we call swim with its new size, what swim will do is ”swims” that new element up the min heap, until it is at its correct spot. So, for example, if we push 1 to the min heap… we push it to the back of the vector… and now we call swim with the new size of the priority queue which is now 7… and all the swim function does, is compares whether the new value is smaller than its parent, if it is then it simply swaps the values and continues on that parents index. Lets see this in action!

Min Heap: swim

Each iteration, we check if the current value is smaller than its parent. If it is we swap, else we return.
`1 < ?`? Remember, you need to check if the parent exists, since it doesn't, we have reached the root and can now return!

- To push an element to the min heap, it follows a similar approach. When we push an element to the back of the vector, we call swim with its new size, what swim will do is ”swims” that new element up the min heap, until it is at its correct spot. So, for example, if we push 1 to the min heap… we push it to the back of the vector… and now we call swim with the new size of the priority queue which is now 7… and all the swim function does, is compares whether the new value is smaller than its parent, if it is then it simply swaps the values and continues on that parents index. Lets see this in action!

Min Heap

- The functions you will be implementing are as follows:
  - `void push(const T& key);`
  - `void pop();`
  - `const T& top();`
  - `int size();`
  - `bool empty();`
  - `void sink();`
  - `void swim(int i);`

- Remember when implementing these functions (especially with sink and swim). You want to check all your edge cases!
- Like what if there is no parent when you're swimming a value up the heap? What if there is no left or right child when sinking a value?
- Make sure to check for these!

Give “Min Heap” a go to try and implement your very own Min Heap!!

Access to google drive

- I will upload slides to the Google Drive after every class
- https://drive.google.com/drive/folders/1H5psebndM_YVyoJE-BJ_ODNJOfgq9-uI

Contact: Thomas.golding@uts.edu.au