Abstract & Concrete Data Types

Gabe Johnson

Applied Data Structures & Algorithms

University of Colorado-Boulder

This sequence is about the different between _abstract_ data types and _concrete_ data types. With abstract types, you only care about how it is used. With concrete types, you also care about the implementation details.

In other words, with an abstract data type, it's not how it works, it's how you use it.

Episode 1

Abstract vs Concrete

Interface vs Implementation

Interface
Abstract definition of how thing is used and how it behaves

Implementation
Practical (e.g. memory, speed) considerations: concrete data type.

If you only care about how you can use it, you're working with an abstract data type.

If you care about how the data is arranged in memory, you're working with a concrete data type.

Abstract/Concrete List

Abstract List:

Get element at i
Set element at i
Get number items in list
Get index of last value
Query if list contains X

Concrete List:

C++ functions
Specific data structures
(e.g. linked list, array)
Memory management
Must implement Abstract List

ADTs are (uhmmm) abstract! meaning they exist only in theory. For example, a list is an abstract data type because we reason about it only in terms of what it does for us, and we don't care how it is implemented.

The abstract behavior might be:

Get element at index i
Set ""
Get the number of items in the list
Get the index of the last value in the list
Query to see if the list contains a given value
... this lets us implement the list however we like.

A concrete list might be the linked list you implemented earlier. Or it might be with a block of sequential memory. Or you might use magic. As long as the implementation matches the abstraction, you're good.

Abstract/Concrete Numbers

Abstract Numbers:	**Concrete Numbers:**
0, 1, 2.523, -4	Tally marks on chalkboard
1/3	Fingers on your hand
π	IEEE-754 floating points
√2	Electrical values of transistors
i	

Abstractly, a number is an abstraction, because mathematically we don't care how it is implemented: tally marks on a blackboard, pencil marks on paper, transistors with high or low values in a digital computer.

We don't care that an 8-bit processor can only directly store 2^8 unique integers, or that there's such a thing as the IEE-754 standard that defines how floating point numbers are defined, and that there are more of them between zero and one than there are between 100 and 101.

So we can talk about the number pi abstractly, though a computer can't represent that value exactly as a single number. Sure, it can represent an approximate value, or represent is symbolically, but not concretely as an exact number.

ADTs in Prog. Langs.

collection: Generic word for any grouping of data
list: Sequence supporting random access
map: Associates keys with values, access via key
set: Unordered group of unique objects
bag/multiset: Like Set but duplicates allowed
queue: List w/ First In, First Out (FIFO) semantics
priority queue: List w/ Priority First Out semantics
stack: List w/ Last In, First Out (LIFO) semantics
graph: Models relationships among data

In many languages, an ADT is presented as an interface or abstract type. Here are some abstract data types:

collection: Generic word for any grouping of data
list: Sequence supporting random access
map: Associates keys with values, access via key
set: Unordered group of unique objects
bag/multiset: Like Set but duplicates allowed
queue: List w/ First In, First Out (FIFO) semantics
priority queue: List w/ Priority First Out semantics
stack: List w/ Last In, First Out (LIFO) semantics
graph: Models relationships among data

Collection

Groupings of data:

lists - vectors - sets - queues

these are all kinds of collections

A ‘Collection’ is a very generic word for any grouping of data. There are many kinds of collections, like lists, vectors, sets, queues, etc. You could even consider a struct or an array to be a collection.

Typed Collections?

C++	collection **must** be typed, e.g. vector<int>
Java	collection **might** be typed, e.g. ArrayList or ArrayList<Integer>
Python	collection **can't** be typed due to nature of the language

We may be able to restrict the collection to only allow items with a given data type (depends on language).

In C++ we must create a vector of items with the same type: e.g. integers, or strings.

In Java we can be specific, or allow a generic group.

In Python we can't create a list of items whose types are specified.

Common Collection Ops

There is no central committee that decides which operations all collections must have. It varies by language. But in general, you can at least iterate over collections. Even in collection types that do not enforce ordering, you can access them in an order (that order may not be dependable among incarnations).

Common Collection Ops

Iterate over elements of the container
Get the **size** of the container
Get or set the **first item**
Get or set the **last item**
Is the container **empty?**
Copy the container
Does the container represent the **same data** as another container?
Does it currently **contain** a given value?

Operations that are usually (not always) available:

- **Iterate** over elements of the container
- Get the **size** of the container
- Get or set the **first item**?
- Get or set the **last item**?
- Is the container **empty**?
- **Copy** the container
- Does the container represent the **same data** as another container?
- Does it currently **contain** a given value?

In the next episode I'm going to introduce a bunch of abstract data types, what behaviors they have and how you might use them.

Episode 2

ADT Examples

In this episode I'm going to introduce a slew of abstract data types. Remember that ADTs are defined in terms of their behavior, rather than their implementation.

This isn't really a difficult distinction, but it is easy to forget about it. Like, List is an abstract data structure, but Python gives you a concrete type called List. And since concrete types also have behavior attached to them, so it is really easy to forget that you might be dealing with an abstract thing.

So, all of the following examples are abstractions, and could be implemented in any way you want as long as they obey the semantics and behaviors the ADT specifies.

List

- **Sequential Data**
- **Random Access via numeric index**
- **May support sorting**
  - **On insert**
  - **Or on demand**

Example Implementations:

C++ vector<int>
Java ArrayList
Python Lists
Go slices

Lists are abstract sequences of information that you access with a numeric index. They may or may not support sorting.

Lists are sometimes called dynamic arrays which is a very fancy term. It means you don’t have to worry about how the memory is laid out, but you also can't depend on it working exactly like an array either.

Map

- Contains key / value pairs
- Access values using key rather than numeric index
- May support sorting
- By key
- By value
- Size refers to number of key / value pairs
- Keys are unique; pairs are not.

A Map can be called a hash, an associative array, or a dictionary. There are probably other names for it out there.

Maps are sort of like lists in that you index into them, but instead of modeling sequential data with a numeric index, maps model dictionary-like data with arbitrary key and value types.

They may or may not support sorting, either by key or by value.

Size refers to number of key / value pairs

And since we’re indexing using keys, they are unique; values are not.

Map Uses

“Phone Book”

Given some piece of info (e.g. someone’s name),
look up some other piece of info (e.g. someone’s phone number)

- IP address to domain names;
- Usernames to password hashes;
- Colleges to mascots;
- Previously computed function results;

You can use a Map whenever you have a dynamic set of things (e.g. you don’t know what they will be) but you need to relate them to values. Examples include:

- name to phone number
- IP address to domain names;
- Usernames to password hashes;
- Colleges to mascots;
- Previously computed function results;

Sets

A List: [5, 8, 8, 5, 10]

… can't be a set because duplicates are not OK

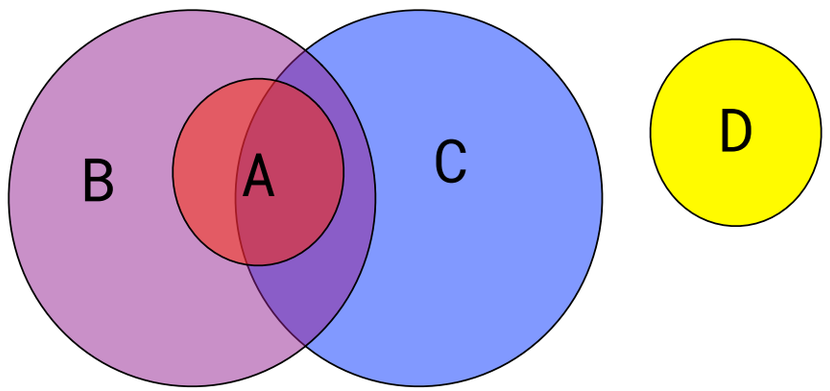
A Set: { 5, 10, 8 }

A Set contains unique items. While a List may contain [5, 8, 8, 5, 10], a set may not have redundant values. So if we convert that list to a set we might be left with (5, 10, 8).

Sets are egalitarian: they are typically unordered. There is no dependable way to determine which one is first or last.

Some variants allow you to enforce a sort order when you iterate over the values.

Useful Set Math



Sets let us add, subtract, intersect, and exclude using set semantics. What’s the intersection of A and C? What’s the combined set of B and C?

Bag (Multiset)

Bags AKA Multisets are sets that allow duplicates.

A List: [5, 8, 8, 5, 10]

… can be a bag because duplicates are perfectly fine

A Bag: { 5, 10, 8, 8, 5 }

A Bag is an extended form of Set that removes the uniqueness constraint, so they can contain multiple entries. They're still not ordered.

Removing from Bags

A Bag: { 5, 10, 8, 8, 5 }

Remove 5!

Possible results:

Remove all instances: { 10, 8, 8 }

Remove one instance: { 5, 10, 8, 8 }

If I have a Bag with [5, 8, 8, 10, 5], and I want to remove 5, what should the container look like afterwards?

Two possible semantics:
- Remove all the fives
- Remove one five.

Queue

FIFO

“First in, first out”

Queues are First In, First Out. The first person in line is the first person to leave the line. This works just like waiting in line at the bank.

Remember that phrase: FIFO = First In First Out. You’ll see it everywhere.

Queue Quirks

bare bones operations:

enqueue (add to the back of the queue)

dequeue (remove from front of the queue)

A bare-bones queue doesn’t need to give operations like random access by key or index. It doesn’t even need to give the size.

All a Queue is required to do is remember which order things were put in so you can remove them in that order.

Queue Quirks 2

Fancy words

Enqueue: add to the queue

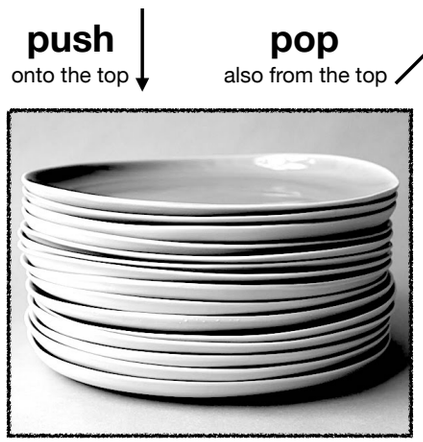
Dequeue: remove from the queue

why ‘add’ and ‘remove’ aren’t sufficient
i do not know

There are special words for adding and removing from queues. I don’t know why.
Enqueue is the fancy word for adding to a queue. Dequeue is the fancy word for removing from a queue.

This was probably decided by the same person who thought it would be a neat idea to have nineteen different words for ‘loan’.

Stack



LIFO

“Last in, first out”

The inverse of a Queue (if there is such a thing) is a Stack. The canonical example is a stack of plates. You push a new plate onto the stack. When you need a plate you pop it off the stack.

This means that a Stack has Last In, First Out semantics, often abbreviated LIFO.
I don’t know why it isn’t First In, Last Out (FILO). That would be more fun.

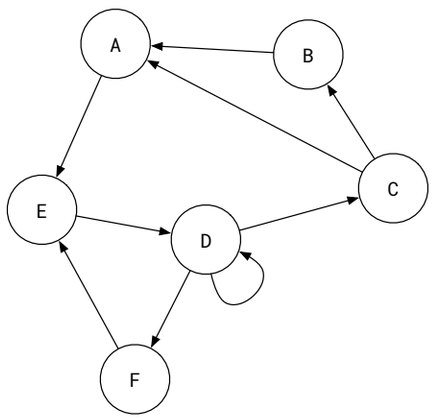
Stacks = Super Useful

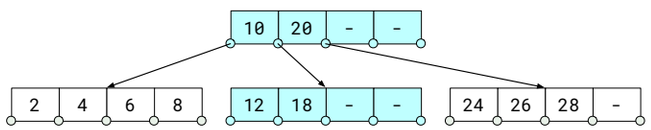
- Parsing program text
- Store/save state
- Many many algorithms
  - Graph algorithms
  - Dynamic programming

Stacks have many, many applications. Your C++ compiler uses stacks to parse your source files. When a program is running, it uses a stack to record what it was doing when you call a function (e.g. so it can return to the right place).

If you issue a function call, the machine pushes the current state onto a stack. When that function call returns, it pops the stack, which restores the variables at play when the function call was made.

Stacks/Queues





B-Tree seek path stack:

pointer to 12/18
pointer to 10/20

Even though stacks and queues are linear (they record an ordering based on when something showed up) they are very useful in traversing nonlinear structures like this.

<next build>

Stacks/queues give us a way to record a path through some complex structure. E.g. recording the path of a B-Tree insert operation.

Episode 3

Common ADT Operations

A while ago when I was introducing ADTs I gave this big list of common operations on a container or collection, like what's the size, or does it contain something, and such.

In this episode I'm going to pick just a few of these ideas and do a little compare-and-contrasting to give you a little better feel for how the ADTs can be used.

Semantics

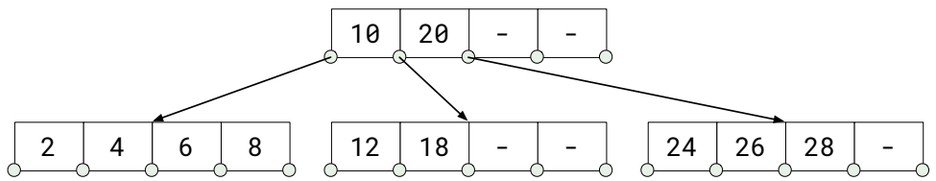
(that means “meaning”)

I was having an argument with someone once about the meaning a certain word in, I think, some marketing thing, because it was important to not confuse people. And my sparring partner was like "oh, you're just arguing semantics". And I was all, "semantics means 'meaning'".

The pop-culture phrase "semantics" seems to mean "something that has no practical difference."

But in software, math, computer science, all things nerdy, semantics means meaning, which correlates strongly with behavior.

Size



size… in memory: 704 bytes

size… num nodes: 4 nodes

size… num keys: 11 keys

there are probably other senses of ‘size’ as well.

For example, what's the semantics of 'size' for a data type?

It might mean 'amount of memory allocated'. This one is actually more about the implementation than the abstraction.

Might mean 'number of nodes'.

Might mean 'number of keys'.

Might mean 'number of unique items'.

So sorry if I'm just arguing semantics here, but these things matter if you want to understand how to use your ADT.

First and Last

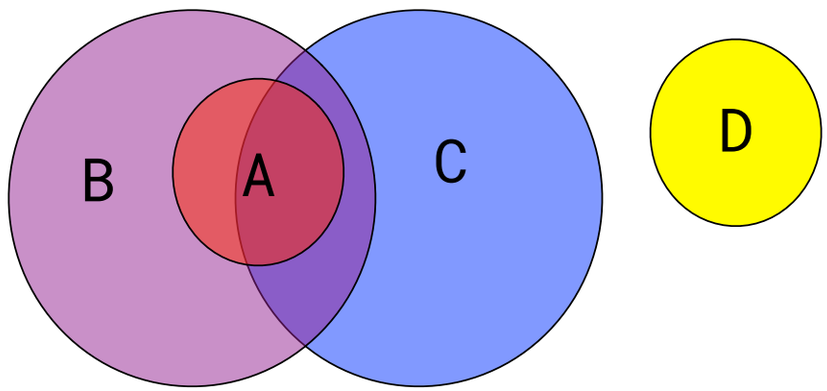
Often this is super clear.

The meaning of 'first' and 'last' might also be not obvious. In a Queue, the first item is the one that has been in the queue the longest. The last item is the one you just pushed onto it.

In a priority queue, the first item is the one with the highest priority, the last item is the one with the lowest.

In both cases, the dequeue operation gives you the first item, but because there are different semantics between these two ADTs, what you get will be different.

Does Not Compute!



Sometimes ‘first’ or ‘last’ makes no sense at all. What’s the first item in a set?

And some ADTs don't even have the concept of first and last. For example, a set is unordered, so what does it mean to ask for the first item? Sure, some implementation might have some weirdo concept of 'first', but for the abstract set type, there's no concept of first or last.

Same Data?

Bag A: { 5, 10, 8, 8, 5 }

Bag B: { 8, 5, 8, 5, 10 }

Bags A and B have same data because order does not matter.

List A: [5, 10, 8, 8, 5]

List B: [8, 5, 8, 5, 10]

Lists A and B don't have same data because the order is different.

You often need to know if two collections contain the same data. But, what if one collection is a list, the other is a set? Often, a programming language gives you an abstract data type that has 'same data' behavior, and it is up to the implementation to define exactly how that should work.

You'll find that in many cases, an ADT has kind of ambiguous meaning, or that it kicks the meaning down to the implementation level. That probably isn't super good engineering practice, but you'll still see it come up in practice.

Equivalence

```
public boolean equals(Thingy other) {
    return this.fooVal == other.fooVal &&
           this.barVal == other.barVal &&
           this.sillyFunction() == other.sillyFunction();
}
```

Here's another way of thinking about the 'same data' problem.

How do we know if two things are equivalent? E.g. if I create two strings with the same character data and throw them into a set, should the set

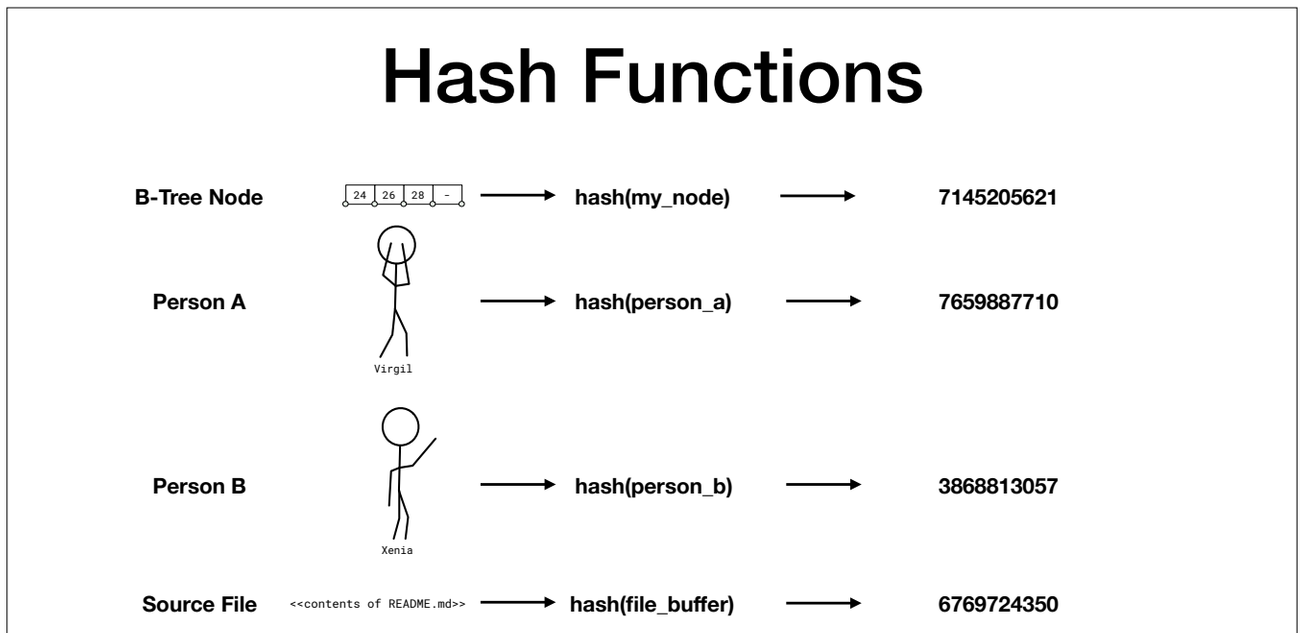
A. Say yes, they are the same since they have the same character data, or
B. Say no, they are not the same since they are unique objects in memory?

Answer: Depends on what you're doing. (Sorry)

If we want two different objects in memory to be treated as 'equal', we need to write code that determines equivalence. One way to do this is with an 'equals' function, or to override the '==' operator.

In Java: <code>

This can be pretty involved. Or not. Depends on how it is done.



Hash Function

A very common method for testing equivalence is to use a hash function. A hash function is a bit of math that turns an arbitrary object into a number, called the hash. For any equivalent object, the same hash function will produce the same hash.

We will cover hash functions in detail in a future sequence.

And More.

The interface is probably not the whole story.

There are more operations that I could show you, but the important thing to get out of this is that the name of the operation doesn't necessarily communicate the whole story. The ADT might specify the behavior directly, or it might be up to the implementation to define how it will behave.

Episode 4

Priority Queues

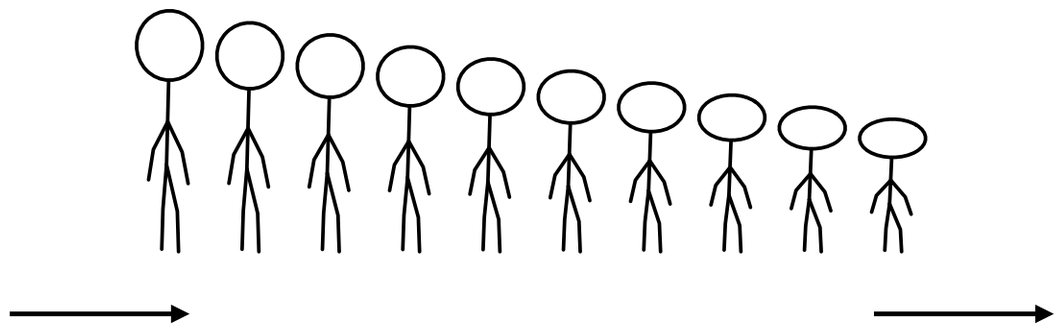
Now we'll talk about the ADT you're doing for the homework assignment.

Queues

In a normal queue, you put stuff in on one end, and remove from the other.

Like waiting in line for whatever these people are doing. You start by going to the end of the line, then you wait until the people in front of you are taken care of. You leave the line (the queue) only when you are the person in front.

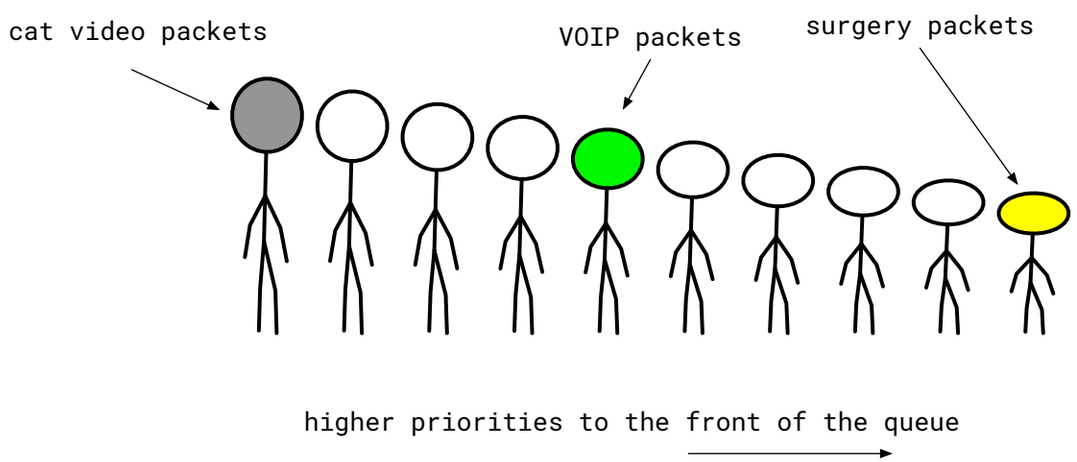
Priority Queues



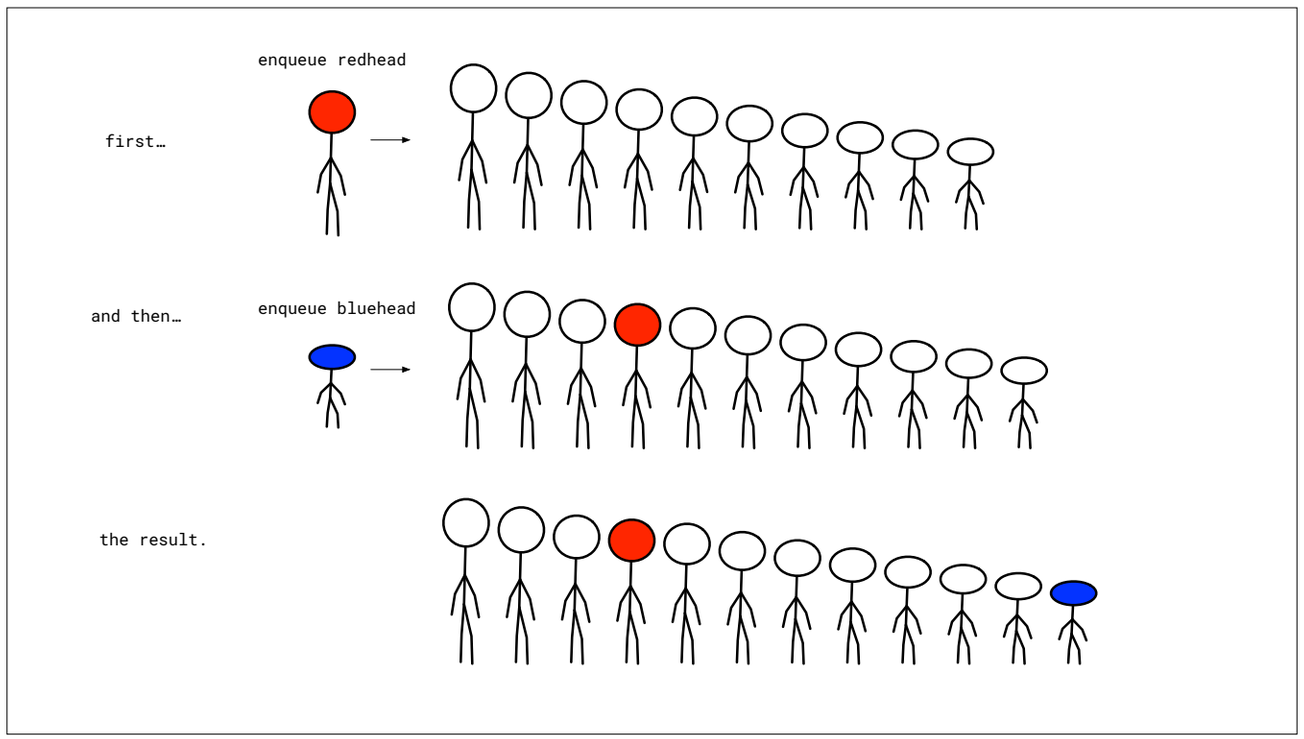
A Priority queue is an abstract structure where the item with the highest priority is chosen next.

Here is a very unfair line at the ticket booth. It is a priority queue that is based on the person's height. Short people start out closer to the front of the line than taller people.

Why We Do This



A priority queue is often used to arrange activities so that we always handle the most important one next. Say we're writing packet routing software for a hospital and we want to discriminate between important actions (e.g. packets for tele-surgery) and relatively less important packets (e.g. awesome cat videos). Deal with the important stuff before the cat videos.



When you add something to the priority queue, it has a priority. Here we're being super unfair and prioritizing by height, where the shorter you are, the more priority you have.

If we start at the top, and add redhead, she'll go towards the back because she's tall. And when we add bluehead, he goes to the front because he's shorter than everybody else.

Priority Queue is an ADT

Implement however you like:

- Unsorted list?
- Sorted list?
- A Heap?
- Magic?
- Other?

You don't have to implement this using a list.

You could use:

* An unsorted list, and scan through the entire thing when you want to find the next item.
* A sorted list, where you insert items in the right place.
* A heap, where items are stored in a heap data structure (binary trees and arrays can be used), and the one on top is always the next to be used.
* Magic. Hey, as long as it does what a priority queue entails, it's fair game!

The homework this time around is to implement a priority queue. You could implement it using heapsort. Or you could implement a double-ended queue with a special insertion strategy that uses priority. How you do it is up to you.

Implementation is Concrete

Example:

ADT: List

Possible Implementations:
- Array
- Linked List
- Red-Black Tree

The concept of a priority is defined by the interface that it exposes: we put stuff in with a given priority, and we take a thing out that has the highest priority in the structure.

The red-black tree tree we looked at a while back is a concrete data structure, because it specifies how the thing works. It could be considered an implementation of an abstract data type like an ordered list.

A priority queue only specifies the interface, but not how we implement that interface.

So: your implementation will be a concrete version of the abstract data type.

Priority Queue Interface

operation	what it does	synonyms
insert(pq, thing, priority)	adds thing to queue with given priority	enqueue; push
remove(pq)	removes thing from queue with highest priority	dequeue; pop
peek(pq)	reads (doesn't remove) thing with highest priority	front; head

For our homework, 'thing' will be a string. Because using ints for everything is starting to be uncool.

So speaking of interfaces. Here it is!

insert(pq, thing, priority)
  Insert the given thing into the priority queue pq with the specified priority. Sometimes called 'enqueue' or 'push'.

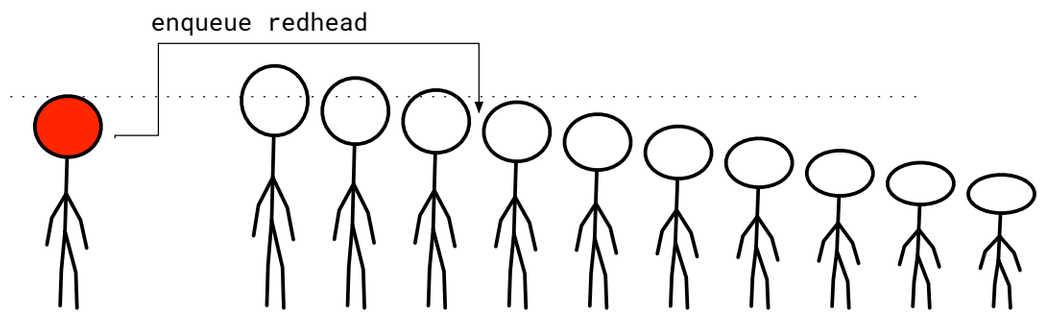
remove(pq)
  Remove and return the highest priority item in the priority queue pq. Sometimes called 'dequeue' or 'pop'.

peek(pq)
  Like 'remove' but do not remove the item. This is common to have with a queue but isn't considered canonical. (But it is considered something you'll do for the homework.) This is sometimes called 'front' or 'head'.

Now I'll go over the concrete C++ functions you'll implement. In the homework. We'll use strings as the data payload, because aren't you getting sick of using ints for everything?

Insert

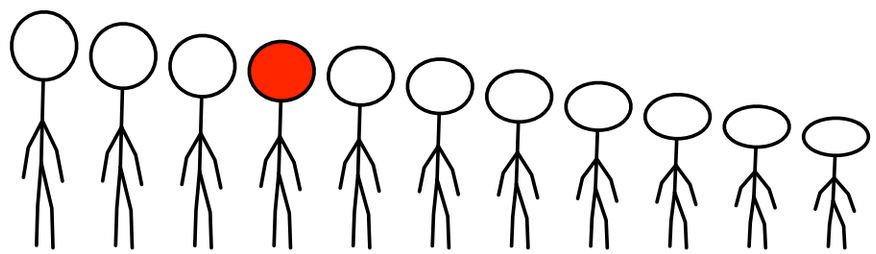
```
void insert(pq*& queue, string& data, float priority);
```



Let's say you implement this with a linked list, but instead of just containing integers like we did for the earlier homework, the nodes contain a priority and a payload. You could then scroll through the list, looking at the priorities to know where to insert.

Insert

```
void insert(pq*& queue, string& data, float priority);
```



And this is where redhead ends up.

Tie?

Options:

1. **Do nothing, don't insert.**
2. **Replace current item at that priority level.**
3. **Place the new item at end of pack of items with that priority level, so it is the least among equals.**
4. **Place the new item randomly among the items with that priority level.**

For homework, do #3.

What about a tie? Good question! It depends on what you're doing.

Option one is to do nothing, don't insert.

Option two is to replace the current item at that priority level.

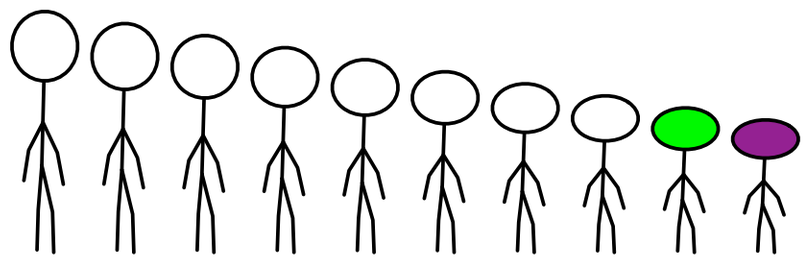
Option three is the put the new item at the end of the pack of iterms with the same priority.

Option four is to put the the new item somewhere among the items with the same priority.

For our homework, we'll use option 3.

Remove

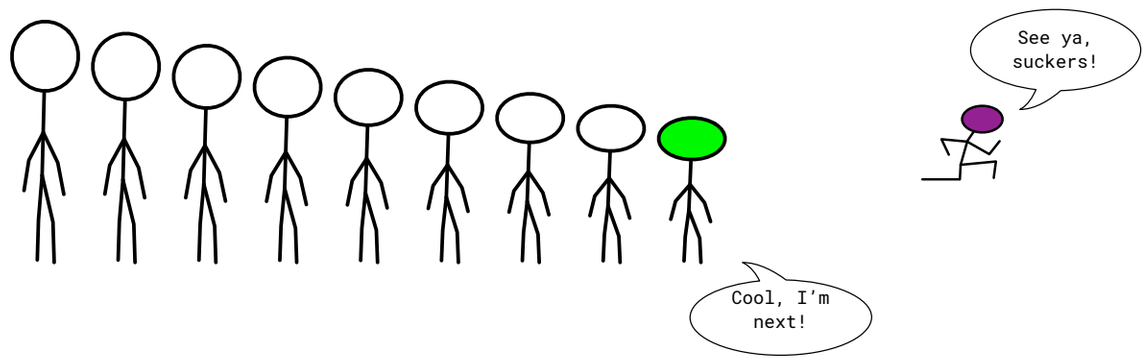
```
string remove(pq* &queue);
```



Removing can be constant time (e.g. if all you need to do is move a pointer), or O(log n) if you need to clean up a heap after a remove, or worse, if you're using a silly implementing data structure.

Remove

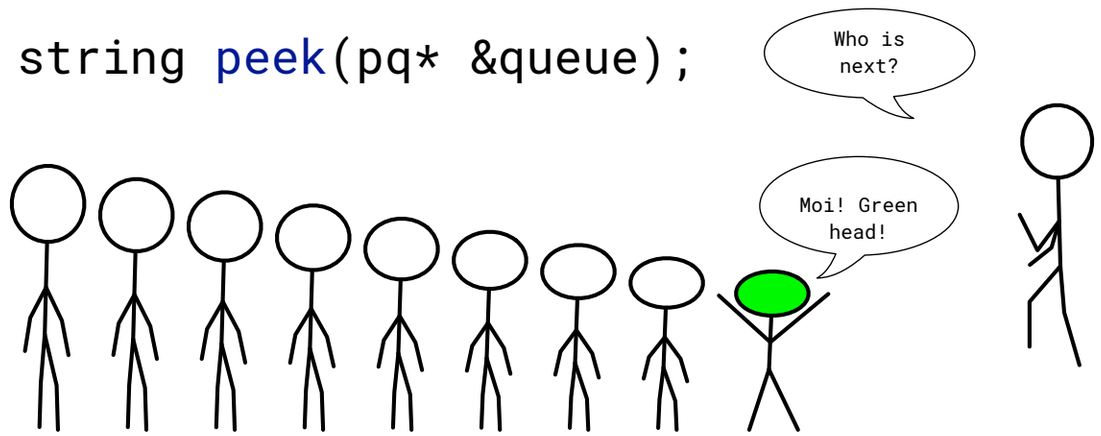
```
string remove(pq* &queue);
```



Now if you remove, it will be the green guy.

Peek

```
string peek(pq* &queue);
```



Peeking is easy. Just return the value that is currently at the highest priority position. Don't modify the underlying data structure. This is like asking the high priority item to report its name without leaving the line.

Homework

How you implement this is up to you!

hint: a **heap** is probably easiest, but you've already got this linked list sitting around…

The operations are straightforward.

The head trip with this assignment is that you will need to decide how you want to implement it. This means: do you use a linked list? a heap? the vector class in the standard template library? something else? However you do it, you'll need to specify the details in your own header file.

Episode 5

Heaps

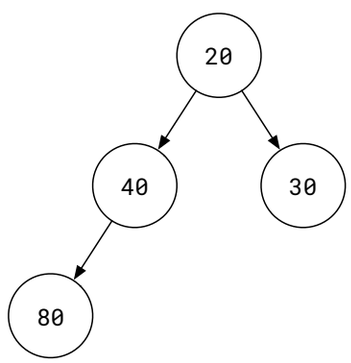
We've been talking about the priority queue abstract data type, and I've been hinting, not very subtly, that you can implement abstract data types using any concrete implementation that you like, as long as it does what the ADT prescribes.

One of the most common data structures for implementing a priority queue is called a _heap_. And that's not to be confused with "the heap", the place you get dynamic memory from. That's a different thing with the same name, unfortunately.

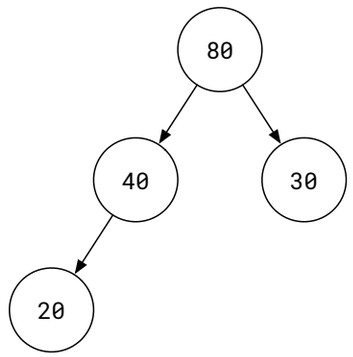
Anyhow, since we're talking about heaps, the data structure kind, we'll also cover heapsort, which is a sorting algorithm like the ones from the sorting sequence.

And that tees up another episode on a cool data structure called _treaps_, which I'll cover partly because it has a super fun name to say.

Min and Max Heaps




```
// min heap
// smaller values at top
```




```
// max heap
// large values at top
```

Heap Sort is an algo that relies on a specific data structure called a heap.

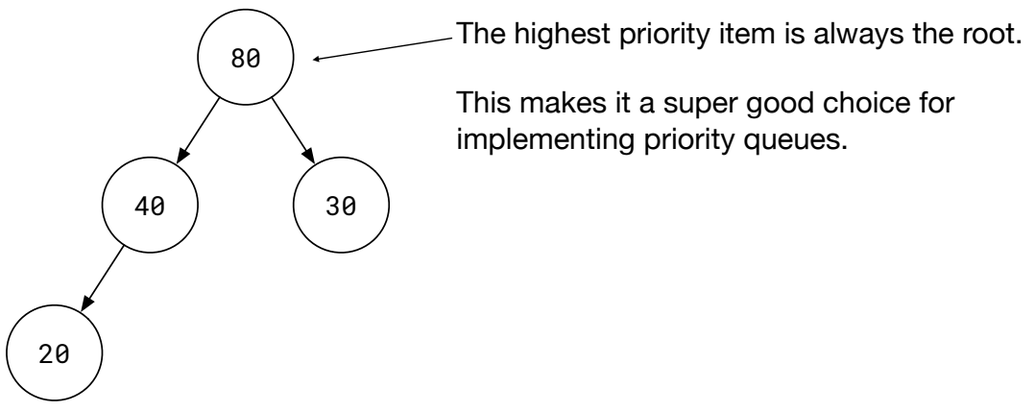
Heaps can be top-heavy, called a max heap, or bottom-heavy, called a min heap. They're the same thing except the direction is different.

With a min heap, parents have smaller values than their children. This means the minimum value is at the top.

With a max heap, parents have larger values, meaning the maximum value is at the top.

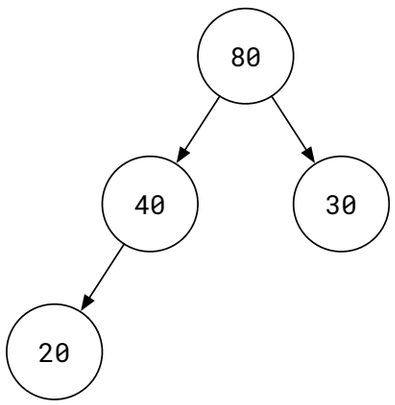
The rest of the examples given here are with a max heap.

Heaps for Priority Queues



A heap is a really good data structure for implementing a priority queue, because the item with the highest priority is always at the top, so it is a constant time operation to access it.

Drawn as Binary Trees

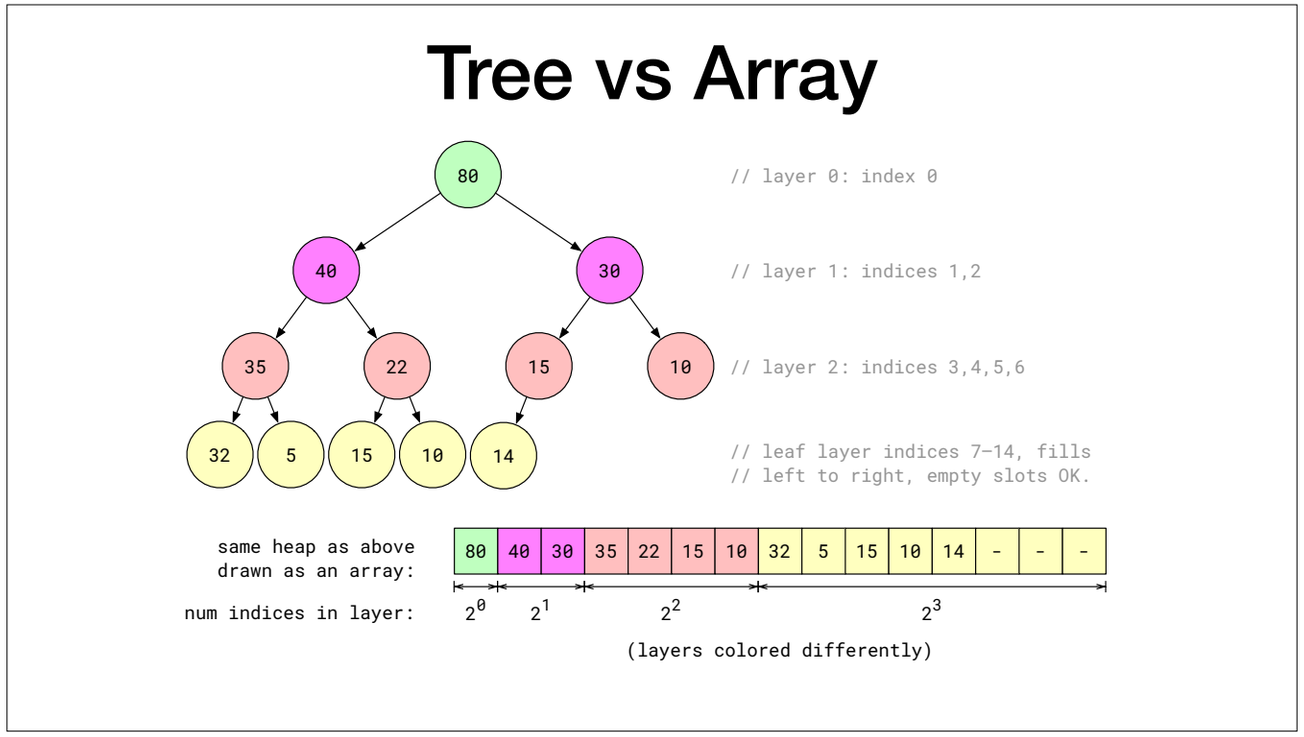


Often *represented* as binary trees, but can (and usually do) implement with an array.

We draw them this way to make it easier for us to see the relationships.

Heaps are often drawn as binary trees. In this case, we're really dealing with a _binary heap_. Even though we draw it as a binary tree, it is way more efficient to use an array since the nodes are all in predictable spots.

There is no distinction made between left and right branches. The thing that matters is that a parent's value is larger than the children.



To make this a bit more obvious, here's a heap of color! Each layer is a different color. Since it is a binary tree, each layer has two to the something number of elements, so layer zero is two to the zero, layer one is two to the one, and so on.

Because of this, we don't actually need to use a tree datastructure, but instead can just use an array. This works only if we also obey the rule that the tree fills top to bottom, left to right. This means the bottom layer might be missing some spots, but if it is, we know the missing ones are all together.

Recap: Heap Invariants

Max heap: parent nodes have larger values than child nodes.
(for min heap it is the opposite)

Tree fills top to bottom, left to right.

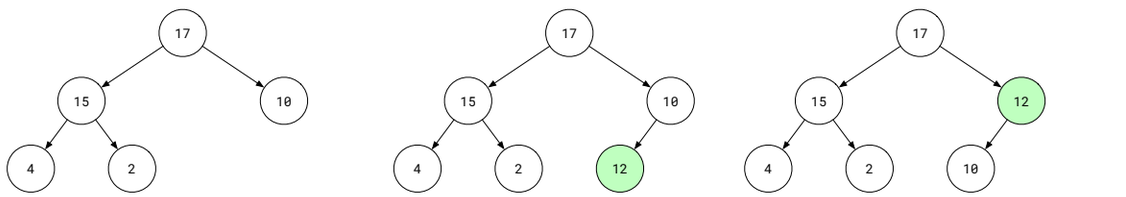
Only bottom-most layer can have empty slots.

Max heap: parents have larger values than children. It's the opposite for min heaps, but you knew that.

The tree fills from top to bottom, left to right.

Only the bottom-most layer can have empty slots.

Inserting into a Heap



starting heap

add 12 to first empty slot

swap with parents until invariant OK (and we're now OK)

Inserting into a heap is easy. Say we start with this heap. We're going to insert 12. Where do you think it should go?

<build>

Take your new value and put it in the first available slot, remembering we fill top to bottom, left to right.

Doing that can break the parent/child ordering, so take a look. In this case we broke that invariant, so ...

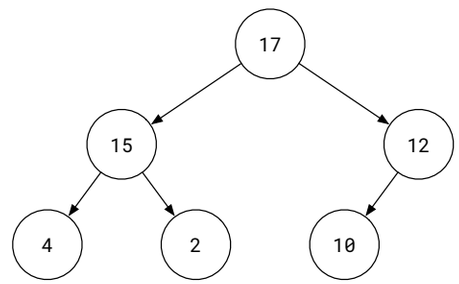
<build>

swap the parent/child, and continue.

In this case we're in a good state, so we can stop. But this process could have bubbled up all the way to the top.

Heap Sort

Goal of heap sort is the same as any other sorting algorithm: produce an ordered sequence of elements based on some input.




```
// First step in heap sort is to build a heap out of
// your data. E.g. given the list [10, 15, 2, 4, 17, 12],
// this is one possible resulting heap.
```

The goal with heap sort is the same as other sorting algorithms: given some input, produce a sequence of ordered elements.

The first step is to build a heap out of the data, just inserting values one at a time.

Heap Sort Algorithm

1. Swap the top of the heap with the last element.
2. Remove the last element, add it to the sorted list.
3. Repair the heap: large numbers above smaller ones.
4. Repeat until heap has nothing left in it.

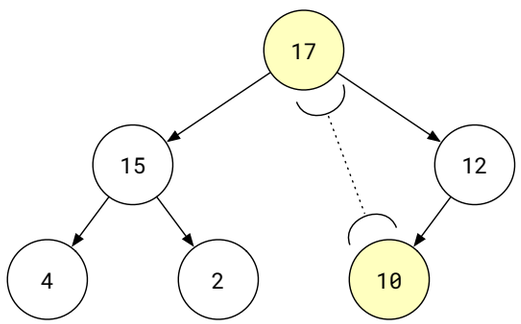
The algorithm is actually quite easy to explain.

Swap the top of the heap with the last element.

Then remove the last element, the one you just swapped in, and add it to the end of the sorted list.

Repair the heap. This is the trickiest step. Large numbers above smaller ones.

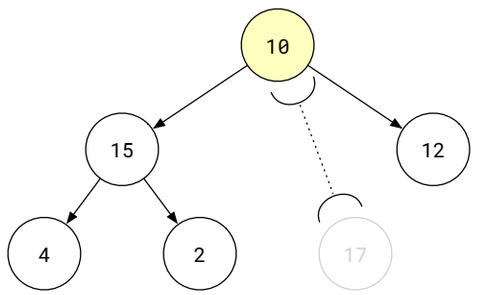
Then repeat those steps until the heap is empty!




```
// Swap top node (17) with last (10).
```

```
// Return list: [ ] (empty)
```

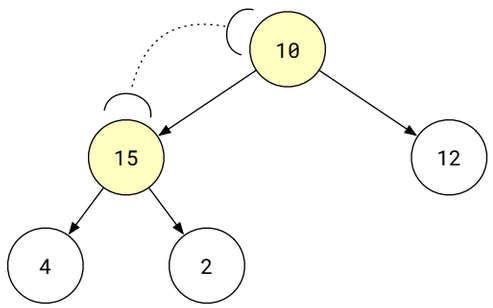
Here we go. This is the heap at the start of the process. We're going to swap the max slot, 17, with the last slot, 10.




```
// Now remove the 17, add it to the return list.
```

```
// Return list: [ 17 ]
```

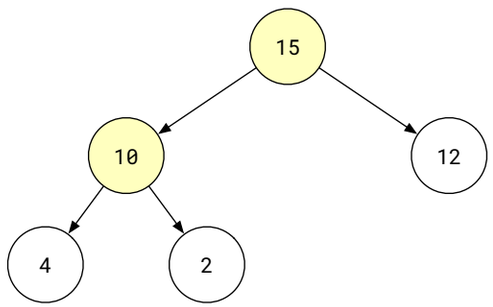
Then remove the 17, and add it to the return list. Notice the 10 is now at the top, and that breaks the max heap invariant because it has children that are larger than it is. So now we fix the heap.




```
// Return list: [ 17 ]
```

```
// Fix broken max heap invariant. Swap 10 with
// the larger of the two children.
```

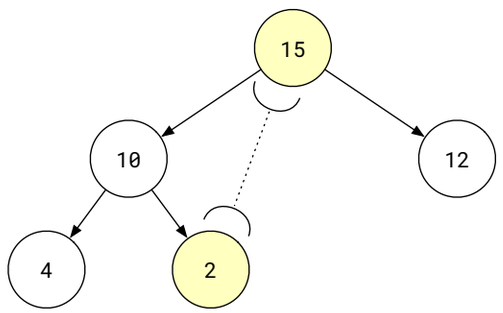
Swap the 10 with one of the children. In a max heap, use the larger of the two; in a min heap use the smaller.




```
// Heap is repaired. Can now continue sorting.
```

```
// Return list: [ 17 ]
```

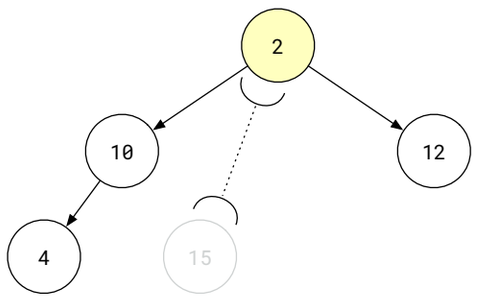
Now the max heap is fixed. 15 is the largest value now, so it will be the next one to get popped off into the sorted list.




```
// Step 1 again. Swap root with last element.
```

```
// Return list: [ 17 ]
```

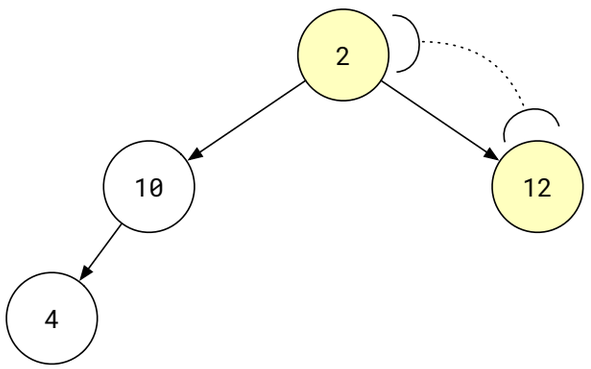
I'll run through the remove/fix process one more time. Now we're removing the 15, and swapping it with the element in the last slot, which happens to be 2.




```
// 15 added to sorted list, but heap needs fixing.
```

```
// Return list: [ 17, 15 ]
```

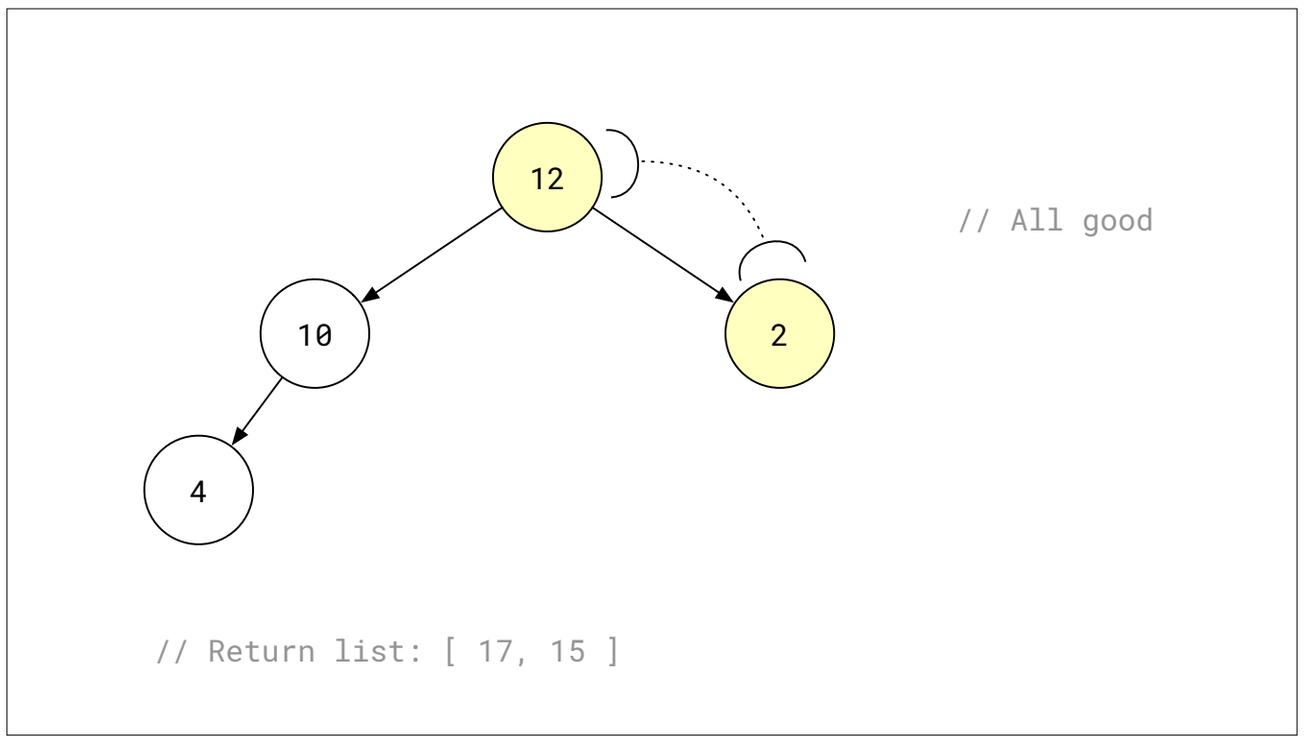
Now the max heap is broken because two is smaller than either child.




```
// Swap 2 with the larger child
```

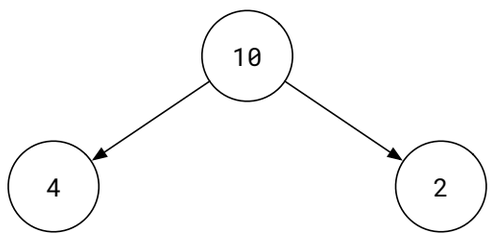
```
// Return list: [ 17, 15 ]
```

Swap 2 with the larger of its children, the 12.



Now the heap is fixed again, so we can continue this util the heap is empty. I'll just scroll through it.

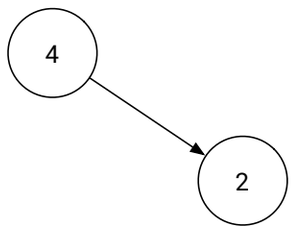
<several more builds>




```
// After removing 12 and fixing

// Return list: [ 17, 15, 12 ]
```

Removed the 12




```
                                                // After removing 10 and fixing
```

```
// Return list: [ 17, 15, 12, 10 ]
```

and the 10

and 4

```
                                        // After removing 2 - heap empty. All done!



// Return list: [ 17, 15, 12, 10, 4 ]
```

and lastly the 2.

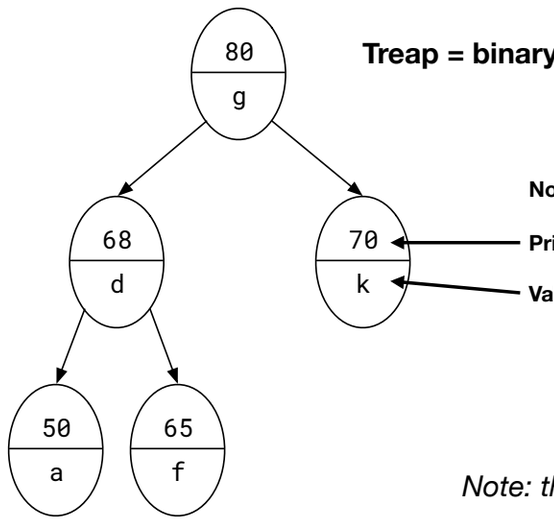
We're finally at the end. Notice that the return list is actually in decending order, which is the opposite of what we did for the homeworks. If you wanted ascending order, you could use a min heap instead of a max heap.

Episode 6

Treaps

A treap is a nifty binary search tree with (some) heap semantics.

A What?



Treap = binary search tree with (some) heap semantics.

Nodes have:

Priority: keeps the tree balanced

Value (key): determines child path order.

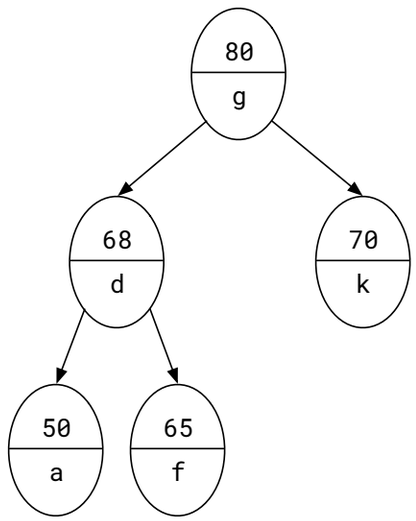
Note: they should have called it a “beap”

Treaps are binary trees. They use two data to determine where a node should be.

A **priority** is a randomly chosen number that is used to keep the tree more balanced than it would be otherwise.

A **value**, also called a **key**, is used to determine which child path the node should be placed. Here the keys are characters.

Priority



This is why we say it has heap semantics.

Parents have higher priority than child nodes.

The priority is why we say this has heap semantics. If we are using a max heap, the large priorities are on the top, and all child nodes must have smaller priorities. If we were using a min heap, then large priorities are on the bottom. Throughout this bit, we're using a max heap, but we could use a min heap instead, just flip the priorities.

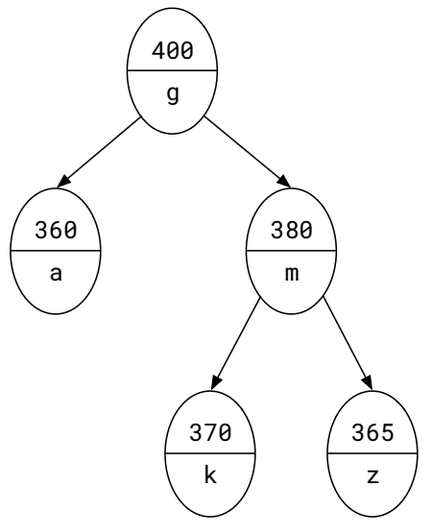
The priority is chosen randomly. This is important, because it has the practical effect of keeping trees __mostly__ balanced. It is immune to pathological conditions, like inserting sorted data and ending up with a tree that looks like a list.

Not Strict Heap

Proper heaps require leaf row fills from left to right.

Treaps can break that, e.g., be filled from the right, instead of strictly from the left.

This Treap is valid.



Note that a proper heap is usually represented as a full tree with the leaf layer filling from the left. A treap does not have to obey this constraint. We can have a somewhat unbalanced treap, as we'll see in the mondo example in a few slides.

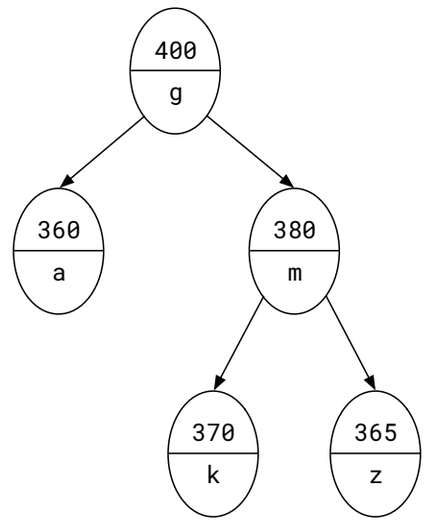
BST Property

Binary search tree order is determined by the value (here we use letters).

smaller values to the left

other values to the right

Using alphabetical ordering in these examples.



The binary search tree property is in effect using the keys. In all of these diagrams the keys are represented with characters, to easily tell them apart from the priority.

Keys are ordered in the same way a standard binary search tree are: lower values are found in the left child, other values are in the right child.

Treap Example

Insert these letters from top to bottom.

Assigning random one-digit priority to make the example understandable.

d	7
b	2
e	8
a	6
g	3
i	5
c	4
h	9

Lets say we want to insert some letters into a treap. When we do this we have to assign each one a random priority. In a real-world implementation we would probably choose numbers from the whole range of integers, but for this example I'll use single digit priorities.

We'll insert from top to bottom with these randomly chosen priorities.

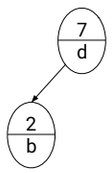
Treap Example

d	7
b	2
e	8
a	6
g	3
i	5
c	4
h	9

Insert d, priority 7

Inserting into an empty treap is easy. Just make an initial node.

Treap Example



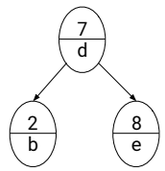
Insert b, priority 2

d	7
b	2
e	8
a	6
g	3
i	5
c	4
h	9

Insert b with a priority of 2. b is lexically less than d, so we know to make a left child. That's always the first step. We only fix the priority situation later, using rotations.

In this case the priorities are already in the correct order, so no action is needed.

Treap Example



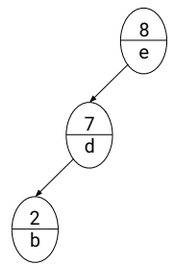
Insert e, priority 8

d	7
b	2
e	8
a	6
g	3
i	5
c	4
h	9

After inserting the 'e' node in the correct location, we see that the max heap property is broken.

We fix this with the cunning use of rotations. Keep the keys sorted in the correct order, but arrange the nodes so higher priorities are on top.

Treap Example

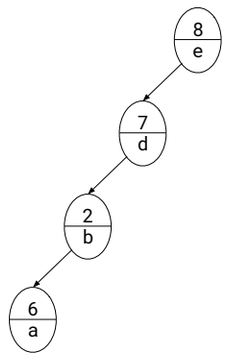


After rotating to maintain heap invariant

d	7
b	2
e	8
a	6
g	3
i	5
c	4
h	9

This is what it looks like after rotating nodes to satisfy both the max heap and binary search tree invariants.

Treap Example



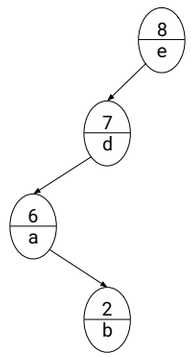
d	7
b	2
e	8
a	6
g	3
i	5
c	4
h	9

Insert a, priority 6. Needs rotation to fix.

After inserting 'a', the priorities are messed up again.

Rotate.

Treap Example

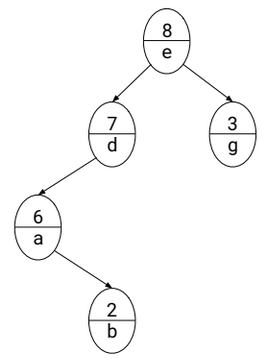


d	7
b	2
e	8
a	6
g	3
i	5
c	4
h	9

After rotation, treap invariants ok now.

Now the value sort order and max heap invariants are ok. Moving on…

Treap Example

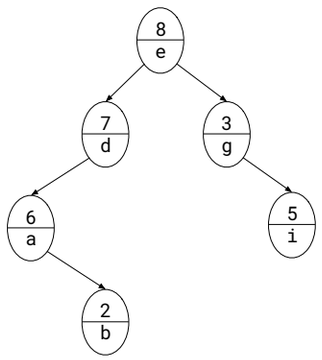


d	7
b	2
e	8
a	6
g	3
i	5
c	4
h	9

Insert g, priority 3. No problems here.

Inserted g with priority 3. All good.

Treap Example

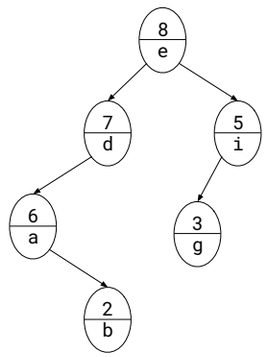


d	7
b	2
e	8
a	6
g	3
i	5
c	4
h	9

Insert i, priority 5. Rotate to fix.

Inserted i, priority 5 on the bottom right there. The heap invariant is broken, since 5 is bigger than 3, so we'll rotate to fix.

Treap Example

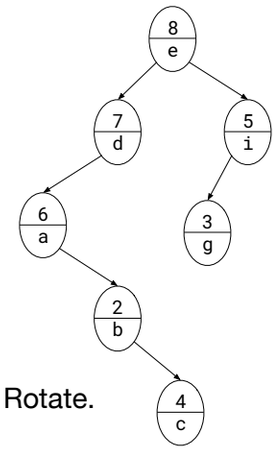


d	7
b	2
e	8
a	6
g	3
i	5
c	4
h	9

After rotating again. Invariants happy.

After rotating, invariants are happy.

Treap Example

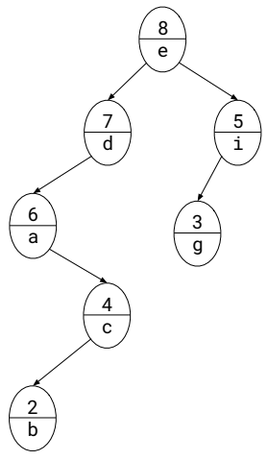


d	7
b	2
e	8
a	6
g	3
i	5
c	4
h	9

Insert c, priority 4, breaks heap. Rotate.

Inserted c, priority 4, and that breaks the heap invariant. Rotate.

Treap Example

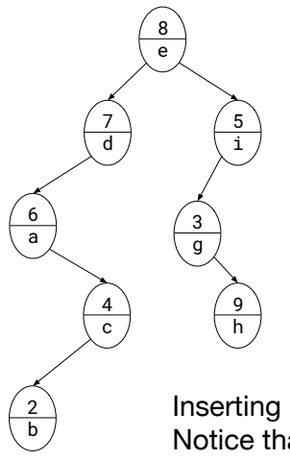


d	7
b	2
e	8
a	6
g	3
i	5
c	4
h	9

Cleaned up.

Now everything is good, continue…

Treap Example

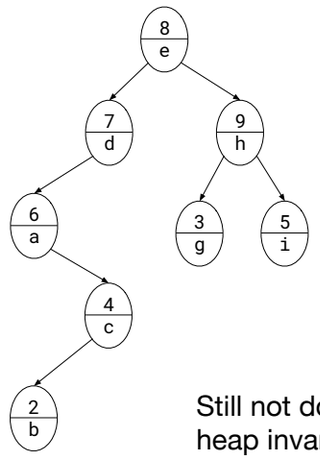


d	7
b	2
e	8
a	6
g	3
i	5
c	4
h	9

Inserting h, priority 9 breaks heap. Notice that's the highest priority so it should bubble up to the top of the tree.

Inserted h, priority 9. That's the highest priority in the treap, so now we'll see how there can be multiple rotations that cause nodes to bubble up to the top.

Treap Example

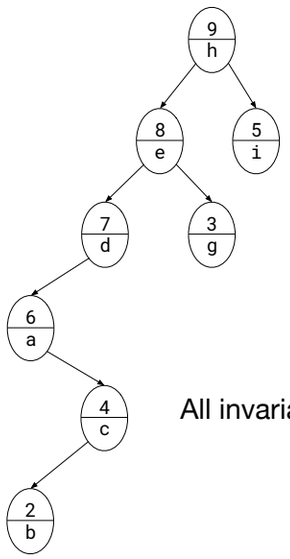


d	7
b	2
e	8
a	6
g	3
i	5
c	4
h	9

Still not done. BST invariant ok but heap invariant is not.

We rotated one time, but the max heap invariant is still broken, since that priority 9 node is below the root with priority 8. Rotate again.

Treap Example



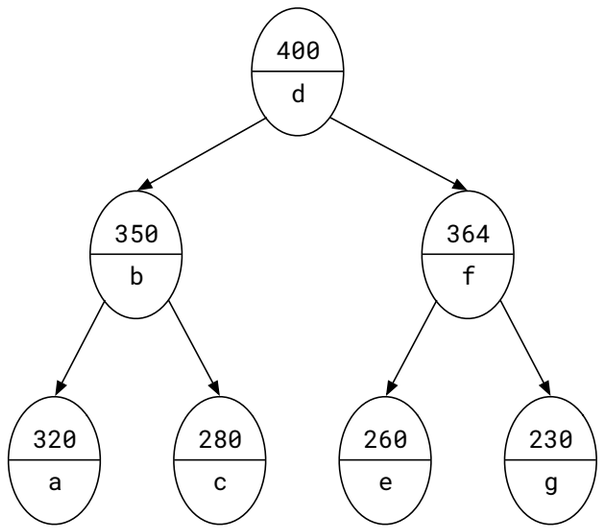
All invariants are OK.

d	7
b	2
e	8
a	6
g	3
i	5
c	4
h	9

And now all the invariants are OK.

This example led to a pretty unbalanced tree, but that was really just dumb luck. In a real setting, over time you'd end up with a pretty balanced tree. No guarantees, since you're using random numbers, but practically it will be good.

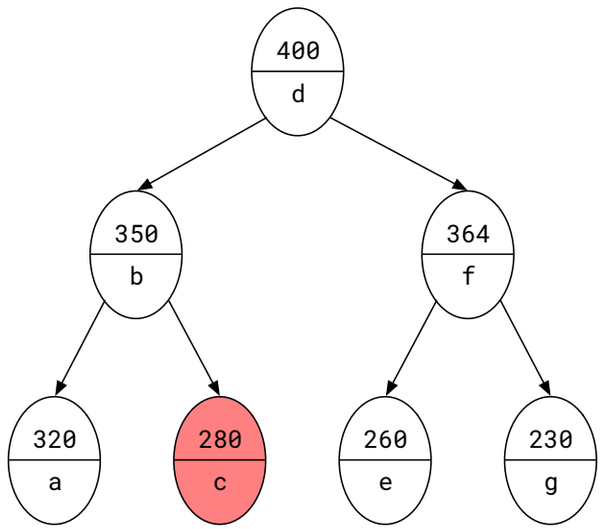
Treap: Remove



Remove: like BST, but post-remove cleanup for priority invariant.

Now, how does the remove operation work? Similar to deleting from a BST but with an additional step of rotations to fix broken heap invariants.

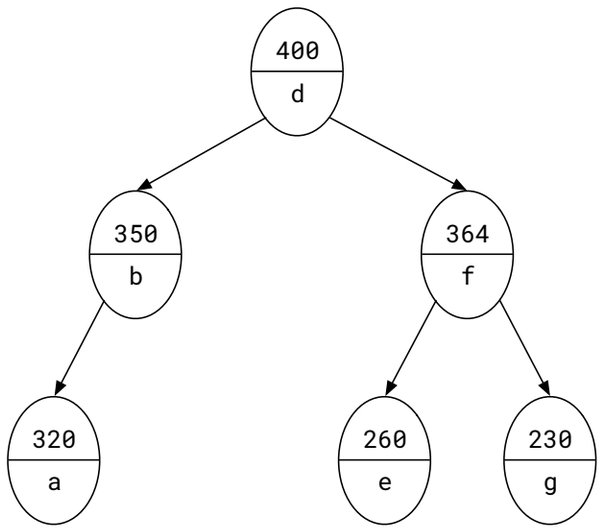
Treap: Remove



Leaf nodes are easy to remove.

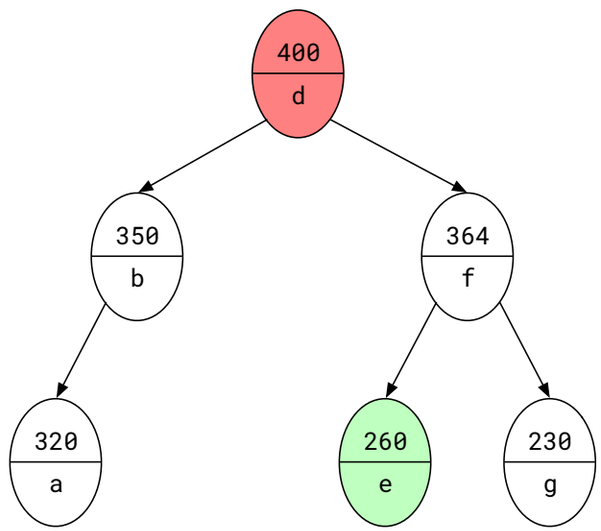
Removing a leaf node is easy. Just remove the link from its parent, and delete the node.

Treap: Remove



And it’s gone.

Treap: Remove

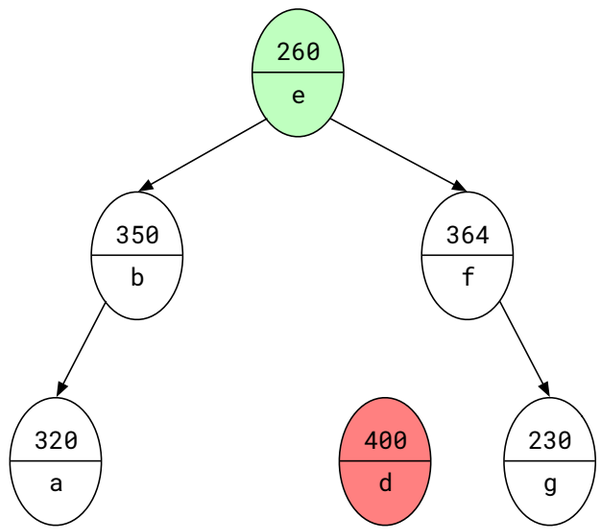


Remove non-leaf as a normal BST, followed by rotations to fix sort and priority invariants.

When swapping the doomed node d with its successor, swap the priority as well.

Removing an internal node is just like with a binary search tree remove operation. Say we remove the root node. It isn't a leaf, so find its sort-order successor, swap them, and remove it as you normally would.

Treap: Remove

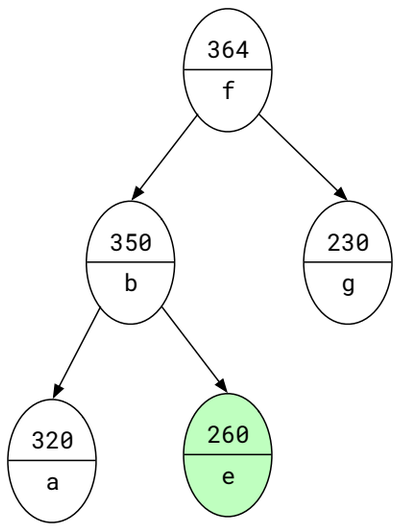


Intermediate step; we've swapped the doomed node with its successor, and severed the reference.

Now fix the broken invariants with rotations.

This is halfway through. We've swapped the doomed node d with its sort order successor e, and deleted the d from its new leaf location. Now the root is e, but the heap property is messed up because 260 is smaller than both of the child node priorities.

Treap: Remove



Remove complete.

So we do a rotation to maintain the sort order, and placate the priority rule as well.

Bonus Round!

Node used frequently? Increase its priority.

Node hardly used? Decrease its priority.

==> fewer traversals.

You can re-assign the priority of a node over time. As the treap is used to find nodes, you may notice some nodes are more popular than others. Other nodes may never be accessed at all.

You may promote frequently used nodes by increasing their priority; demote unused nodes by reducing their priority, and then performing rotations to promote nodes as it makes sense.

This way, the most commonly used items are at the top of the tree. This doesn't change the big-oh notation, but it does reduce the constant number of operations to reach a node.