Contents

List of Algorithms

Chapter 2

Standard Library Data Structures

The purpose of this chapter is to provide an overview on how the most basic and useful data structures work. The implementations of most higher-level languages already coded these for us, but it is important to know how each data structure works rather than blindly use the standard library.

More technical explanations of all of these can be found in a language's API. For Java, this is mostly under the package `java.util`, in the Java API.

I strongly believe that Java is better than C++ for beginning programmers. It forces people into good coding habits, and though the lack of pointers initially frustrated me, it really does make learning general concepts liked LinkedLists much easier, as the intricacies of the C++ pointer no longer distract from the larger idea.

2.1 Generics

In general, a data structure can store any kind of data, ranging from integers to strings to other data structures. We therefore want to implement data structures that can hold any and all kinds of information. When we use a data structure, however, we might want our structure to store only one kind of information: only strings, for example, or only integers. We use *generics* to specify to an external structure that we only want it to store a particular kind of information.

```
1 ArrayList<Integer> al = new ArrayList<Integer>();
```

This means that `al` is an ArrayList of Integers. We can only add Integers into the ArrayList, and anything removed from the ArrayList is guaranteed to be an Integer. We can write `Integer i = al.get(0)` without any need to cast to an Integer.

I don't think the beginning programmer needs to know how to necessarily code a class that supports generics, since each language has its own complex set of rules governing generics. However, we use the standard library extensively in any coding environment, so it is necessary to use a class that does support generics. I think standard classes are relatively straightforward to use but can be annoying to actually implement.

When examining Java API or explanations of implemented functions in this chapter, the characters `E`, `V`, and `K` all can represent generics. For C++, generics are denoted by strings like

`value_type`. For example, in Java, when we set `al = new ArrayList<Integer>()`, `E` represents `Integer`. Otherwise, `E` simply means any object.

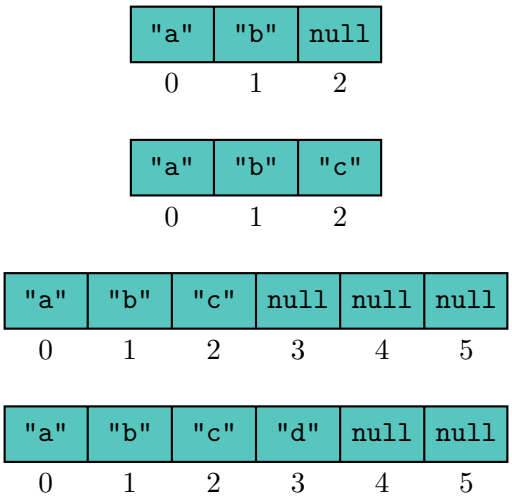
2.2 List

A list is a collection of objects with an ordering. The objects are ordered in the sense that each element is associated with an index that represents its placement in the list. Users of a list have control over where in the list each object is and can access a specific element by its index, like in an array.

2.2.1 Dynamic Array

What is nice about an array? We can access or change any element we want in the array in $O(1)$ time. The problem is that an array has fixed length. It's not easy to append an element to the end of an array.

The fix to this is pretty simple. Why not just make a bigger array, and copy everything over to the new array? Then there's more room at the end to add a new element. If the backbone array runs out of space, we create a new array with double the size and keep going as if nothing happened. Therefore we now have an array of extendable size – a *dynamic array*.



We see that there is still room in the array to add `"c"`, but to add more elements to the list, we must use a new array with double the length.

It's important to note that any given insertion to the structure is either $O(n)$ or $O(1)$, but there is only one $O(n)$ insertion for every $O(n)$ $O(1)$ insertions, so we still average out to constant time.

The Java implementation of a dynamic array is the `ArrayList`. The C++ implementation is the `vector`.

For the following operations, think about how you would implement each and analyze its time complexity.

Function	Java, `ArrayList`	C++, `vector`
add an element to the end of the list	`boolean add(E e)`	`void push_back(const value_type& val)`[1]
insert an element to a particular index in the list, shifting all subsequent elements down one index	`void add(int index, E element)`	`iterator insert(iterator position, const value_type& val)`[2]
access the element stored at a particular index	`E get(int index)`	`reference operator[] (size_type n)`[3]
update the value of the element stored at a particular index to a new element	`E set(int index, E element)`	`reference operator[] (size_type n)`
search whether the list contains a particular element	`boolean contains(Object o)`	`template <class InputIterator, class T> InputIterator find (InputIterator first, InputIterator last, const T& val)`[4]
remove the element at a particular index from the list	`E remove(int index)`	`iterator erase (iterator position)`
search for and *remove* a given element from the list	`boolean remove(Object o)`	use iterators
return the *size* of the list[5]	`int size()`	`size_type size() const`

Accessing and updating elements at particular indices are very nice. They are easy to code and run in constant time. These are the bread and butter of any array. Adding at the end of the list is nice as well. Checking whether some element is contained in the list is a pain, as it is $O(n)$, and adding to and removing from early in the list are more annoying.

2.2.2 Linked List

Arrays are nice for accessing, say, the seventh element in the list. We extend this to the dynamic array to implement adding and removing elements to and from the end of the list nicely. Removing elements from the beginning of the list, however, is cumbersome.

The *linked list* attempts to remedy this. It trades $O(1)$ access to any element in the list for an easier way to remove elements from either end of the list easily. Consider a chain of paper clips:

[1] `&` is a C++ reference

[2] An *iterator* is like a pointer that allows for traversal of a data structure in a particular order. For example, we can increment the iterator to access the next element in the list. An iterator is NOT merely an integer representing the relative position in our list, as the in the Java implementation of `add`

[3] `size_type` is, for our purposes, `unsigned int`, and `reference operator[]` makes this structure's syntax more like a normal array. In particular, to access the element at index `i` of a vector `v`, we simply use `v[i]`, and to update an element, we use `v[i] = val`.

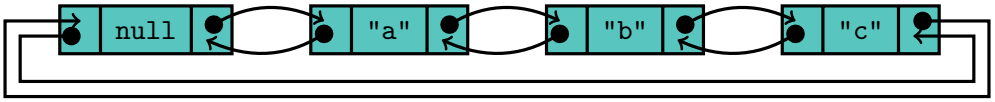
[4] This function is in `<algorithm>`.

[5] Note that the length of the list is not simply the size of the backing array.

[6]

It's easy to add or remove more paper clips from either end of the chain, and from any given paper clip, it's easy to access the paper clip directly previous or next to it in the chain. If we needed the seventh paper clip in the chain, we'd need to manually count, an $O(n)$ operation. However, if we then needed to remove that paper clip from the chain, it wouldn't be that hard, assuming we kept a finger, or pointer, on the seventh paper clip.

The best way to think about and implement a linked list is through a cyclical doubly-linked list, with a dummy head. This means each element has its own node container, while the head of the list is simply a node without an element. Such a data structure looks something like this:



We see that each node maintains a pointer to its next neighbor and its previous neighbor, in addition to containing the String it stores. We can store this data in a class like the following:

```
class ListNode<E> {
    ListNode prev, next;
    E s;
}
```

If we were to insert an element after a ListNode `a`, it is necessary to update all pointers:

```
ListNode<String> b = new ListNode<String>();
b.prev = a;
b.next = a.next;
b.next.prev = b;
a.next = b;
```

[6] http://img.thrfun.com/img/078/156/paper_clip_chain_s1.jpg

Since the linked list is symmetric, inserting an element before a node is also easy. To add something to the end of the list, simply add it before the dummy head. From here it should not be too hard to implement all the important functions of a linked list.

The Java implementation of a linked list is `LinkedList`, and the C++ implementation is `list`. A second C++ class that performs the same tasks but uses a backing array instead of a linked list structure is the `deque`.

Function	Java, `LinkedList`	C++, `list`	C++, `deque`
add an element to the end	`boolean add(E e)`	`void push_back(const value_type& val)`	
insert[7]	`void add(int index, E element)`	`iterator insert(iterator position, const value_type& val)`	
access	`E get(int index)`	use iterators	`reference operator[] (size_type n)`
update	`E set(int index, E element)`	use iterators	`reference operator[] (size_type n)`
search	`boolean contains(Object o)`	InputIterator find (InputIterator first, InputIterator last, const T& val)[8]	
remove the element at a particular index	`E remove(int index)`	`iterator erase (iterator position)`	
search for and *remove* a given element	`boolean remove(Object o)`	`void remove (const value_type& val)`	use iterators
size	`int size()`	`size_type size() const`	
end operations	`addFirst`, `addLast`, `getFirst`, `getLast`, `removeFirst`, `removeLast`	`push_front`, `push_back`, `pop_front`, `pop_back`	

With a linked list implemented, two other data structures immediately follow.

2.3 Stack

A *stack* gets its name from being exactly that: a stack. If we have a stack of papers, we can push things on the top and pop things off the top. Sometimes we peek at to access the element on top but don't actually remove anything. We never do anything with what's on the bottom. This is called *LIFO*: Last In, First Out.

Java implements the stack with `Stack`, C++ with `stack`.

[7] The index in a linked list is implicit.

[8] This function is in `<algorithm>`.

Function	Java, `Stack`	C++, `stack`
push	`E push(E item)`	`void push (const value_type& val)`
pop	`E poll()`	`void pop()`
top	`E peek()`	`value_type& top()`
size	`int size()`	`size_type size() const`

Java implements `Stack` using an array-like structure. This works just as well, and is faster in practice, but I prefer the linked-list structure as a mathematical concept as it is more elegant in its relationship with the queue and more easily customizable.

2.4 Queue

A *queue* is like a queue waiting in line for lunch. We push to the end and pop from the front. Sometimes we peek at the front but don't actually remove anything. The first person in line gets served first. This is called *FIFO*: First In, First Out.

In Java, `Queue` is an interface, and in C++, the implementation of the queue is `queue`.

Function	Java, `Queue`	C++, `queue`
push	`boolean offer(E e)`	`void push (const value_type& val)`
pop	`E poll()`	`void pop()`
top	`E peek()`	`value_type& front()`
size	`int size()`	`size_type size() const`

Since `Queue` is an interface in Java, we cannot instantiate a `Queue`, so the following statement is illegal.

```
Queue<String> q = new Queue<String>();
```

Instead, we must use `LinkedList`, so we do something like this:

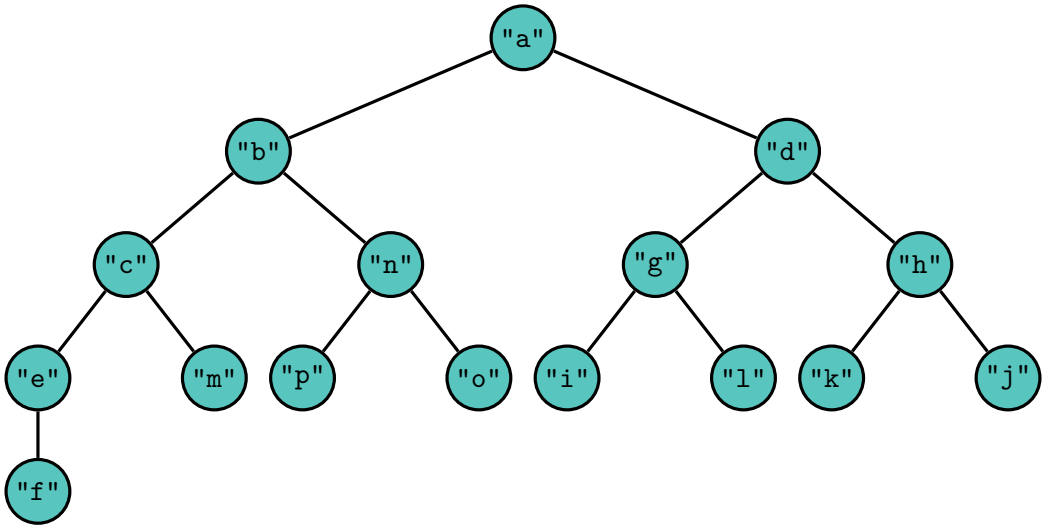
```
Queue<String> q = new LinkedList<String>();
```

This is legal because `LinkedList` implements `Queue`, making it the standard implementation of the FIFO queue.

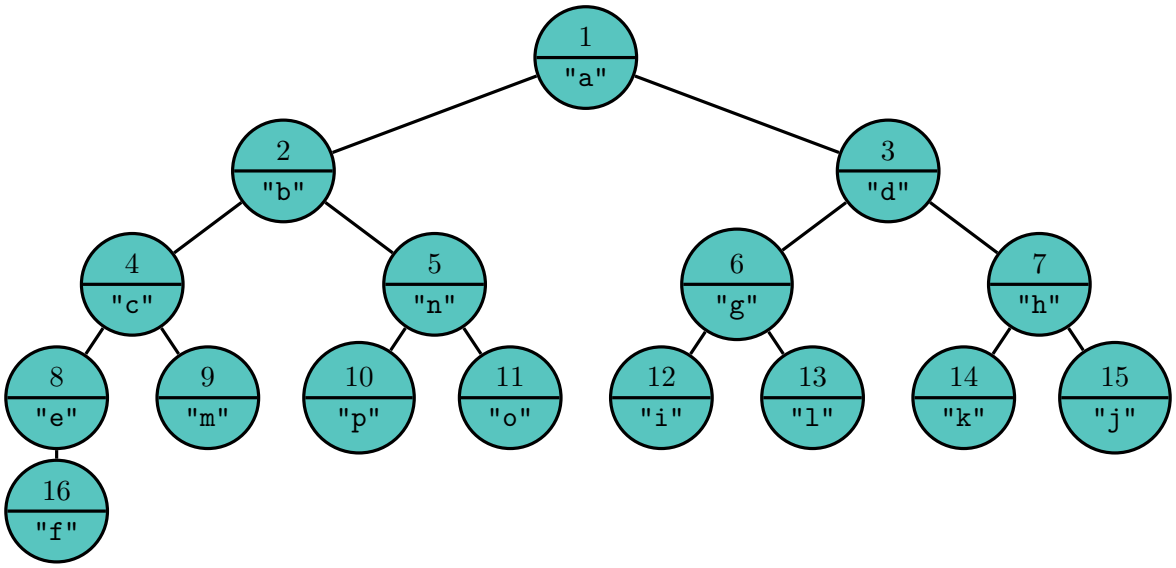
2.5 Heap

Quite often a FIFO queue is not always desirable. For example, perhaps the string I want to remove at every given point is the one that is lexicographically least.

A min heap is a tree such that every node is smaller than or equal to all of its children. A max heap is a tree such that every node is larger than or equal to all of its children. Pictured is a complete binary min heap, which will be of use to us.



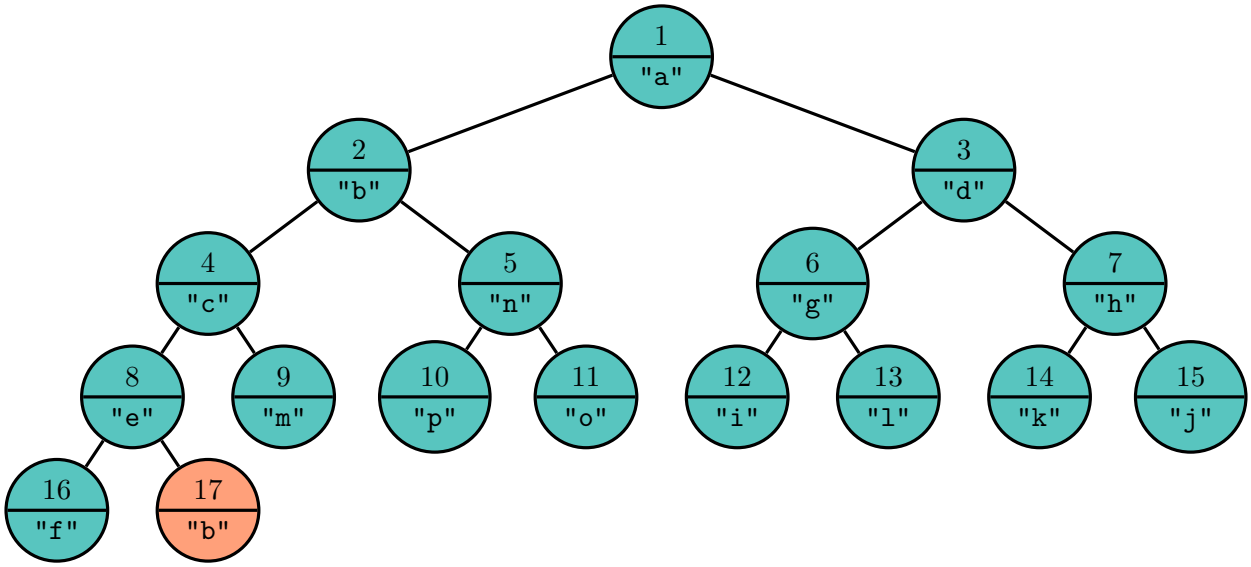
We see that the root of the tree will always be the smallest element. It is tempting to use a container class with a pointer to its left and its right child. However, we have a much nicer way to store *complete* binary trees with an array. Consider the following numbering of the nodes:



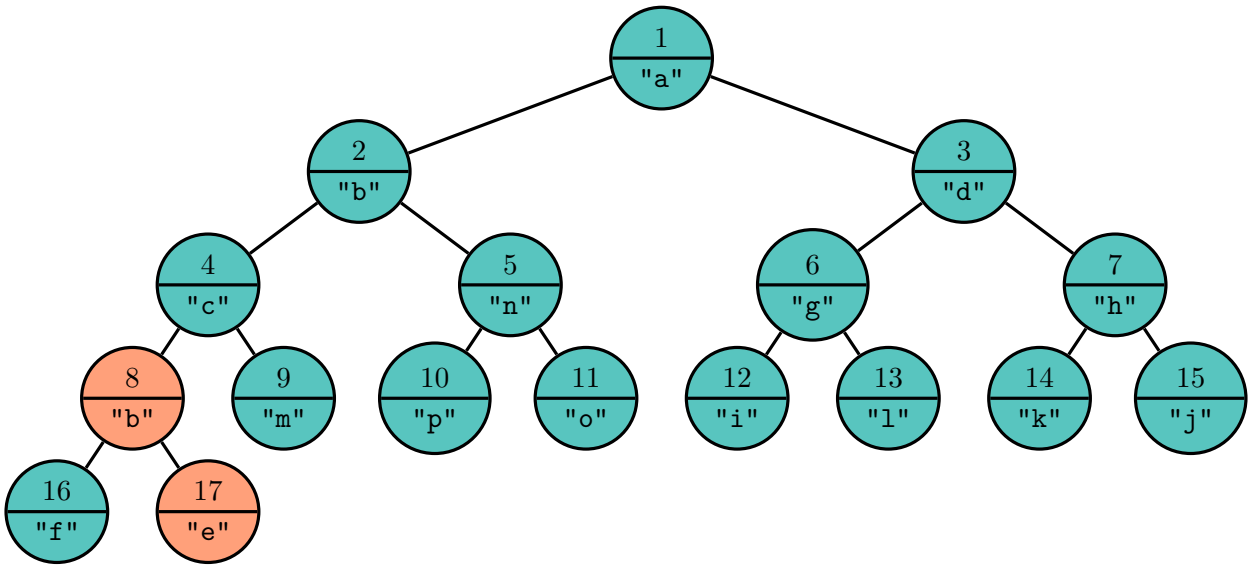
We see that every number from 1 to 16 is used, and for every node, if the index associated with it is i, the left child is $2i$, and the right child is $2i + 1$. This leads to a very natural implementation of the tree in an array:

null	"a"	"b"	"d"	"c"	"n"	"g"	"h"	"e"	"m"	"p"	"o"	"i"	"l"	"k"	"j"	"f"
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

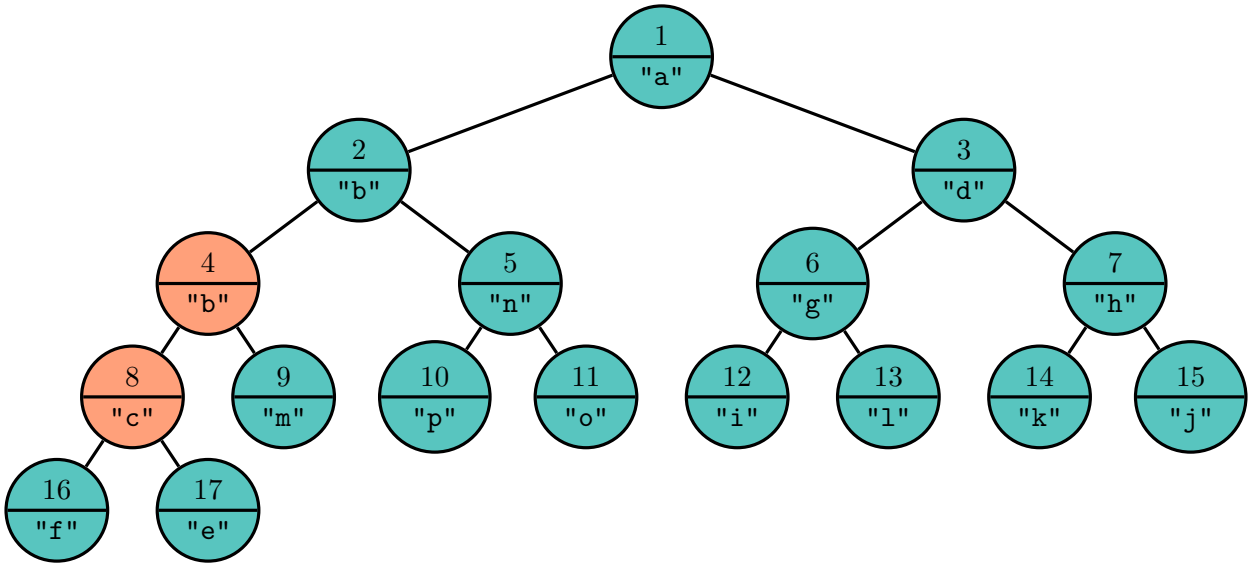
How do we add elements to our heap, while maintaining the heap qualities? Well, let's just add it to the very end and see what we get. Suppose we are to add `"b"` to the tree.



Well, `"b"` comes before `"e"` in the alphabet, so let's swap the nodes. We are guaranteed that `"b"` should come before the other child (in this case, `"f"`) by the transitive property.

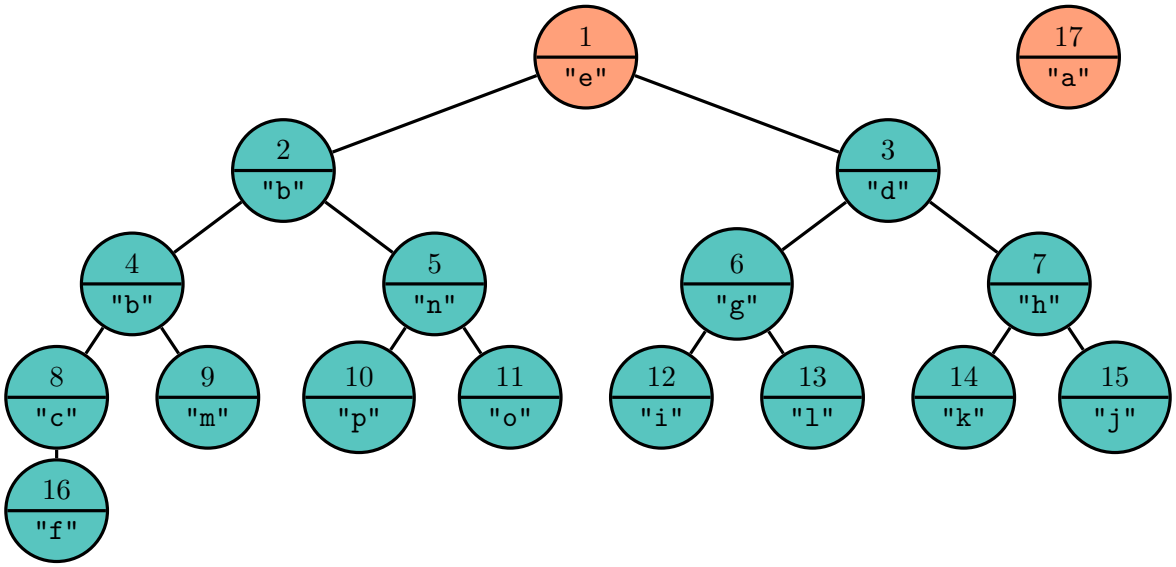


One more swap...

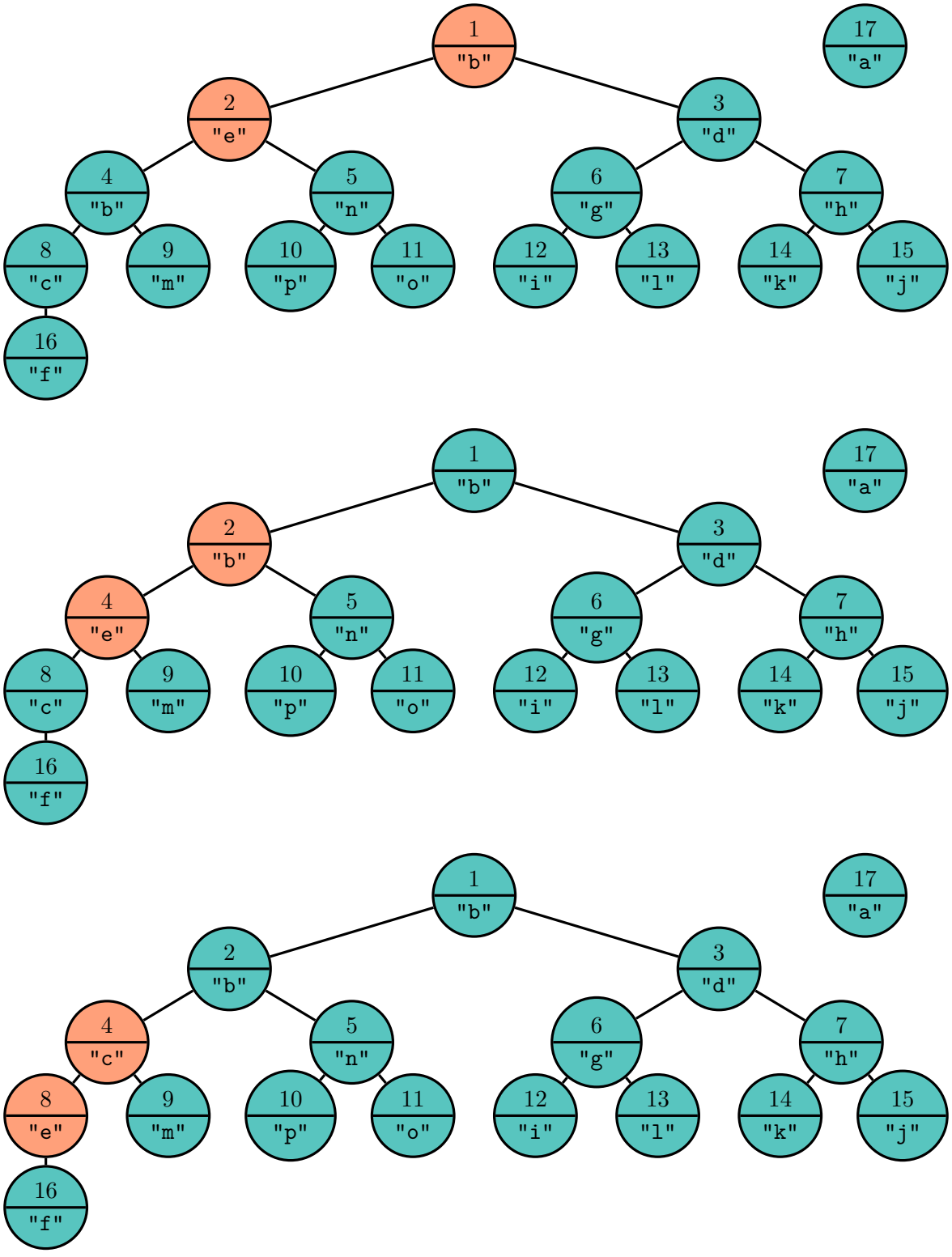


And now we have the heap property restored. As the tree has depth at most $\log n$, this process is $O(\log n)$.

To remove the root from the heap, we replace the root with the last leaf:



We perform a series of swaps to restore the heap property. We always want to choose the smaller child to swap until the heap property is satisfied.



And we are done. Once again, this takes at most $\log(N)$ swaps. This idea can be extended to removing or changing the value of any node we'd like from a tree – this is particularly useful for Dijkstra later.

Remember to implement your heap in an array-like structure!

Java implements a min heap with the `PriorityQueue`. This class, like `LinkedList`, also implements `Queue`. C++ implements a *max*[9] heap with the `priority_queue`. The functions for heaps in both languages are nearly identical to those for queues.

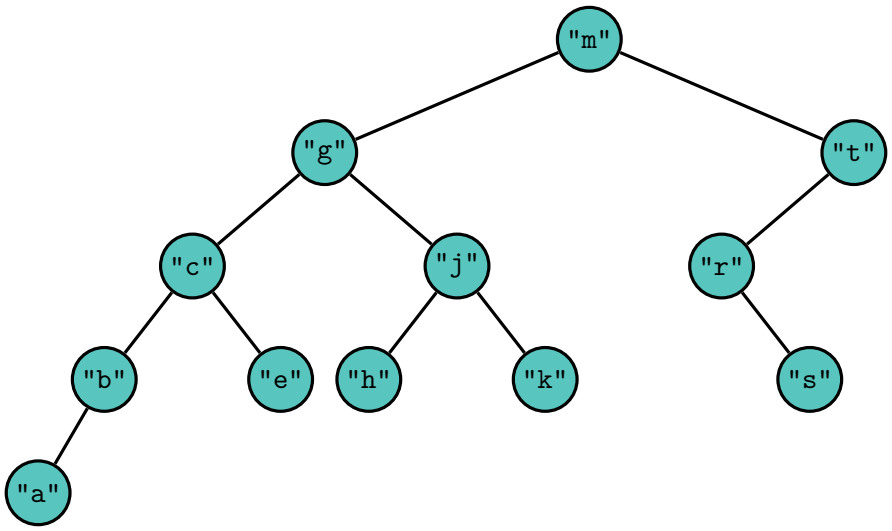
Function	Java, `PriorityQueue`	C++, `priority_queue`
push	`boolean offer(E e)`	`void push (const value_type& val)`
pop	`E poll()`	`void pop()`
top	`E peek()`	`value_type& top()`
size	`int size()`	`size_type size() const`

2.6 Set

A *set* is a collection of objects with no duplicate elements. Note that the data structures discussed in this section can be extended to become multisets, but Java and C++ implementations of these explicitly disallow multiplicity.

2.6.1 Binary Search Tree

A *binary search tree (BST)* is a tree where every node is greater than every node in its left subtree and less than every node in its right subtree. As with a heap, to use a BST, we need to impose some kind of ordering on the elements stored.



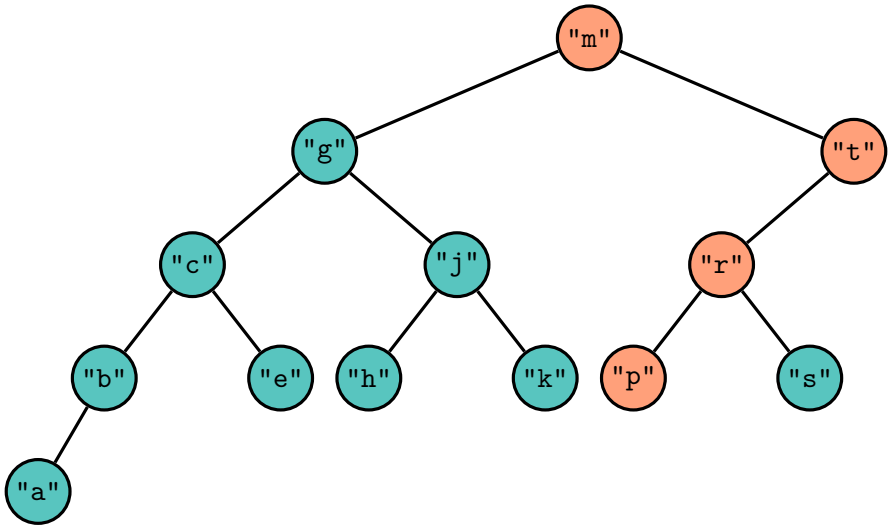
The tree need not be complete, unlike the heap. Because it is not guaranteed to be complete, there is no way to nicely bound the size of the array we would need if we were to use the same storage method as with the heap. Thus, we are forced to use a `TreeNode`, with left and right

[9] Don't forget that C++ implements a max heap, ever.

pointers. This is also problematic when determining guarantees on time complexities later, but the ways to solve this problem are pretty complicated so we'll ignore them for now.

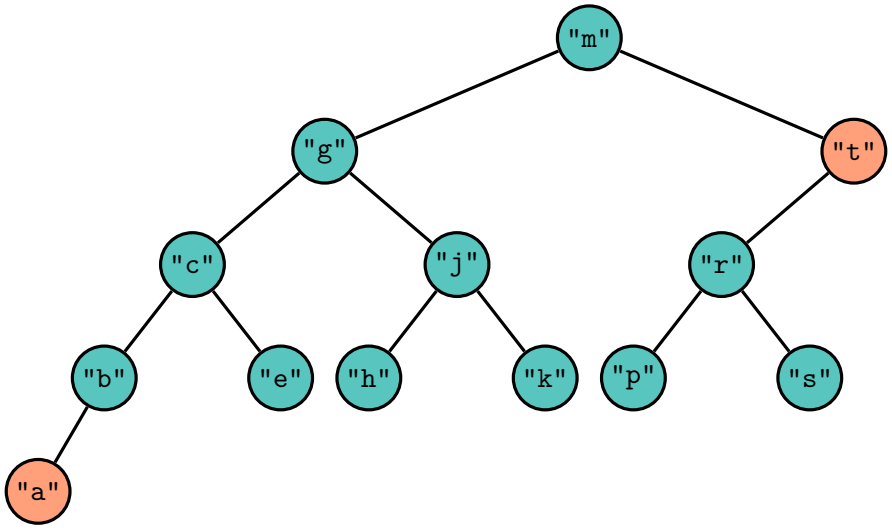
Given the name of the tree, searching for an element within the tree is quite natural, and similar to a binary search. Compare the element to be searched for with the current node. If they are equal, we are done; otherwise, search the appropriate left or right subtree. As with most structures and algorithms with a binary search structure, this operation lends itself nicely to recursion. If the tree is reasonably nice, we expect to complete this in $O(\log n)$ time, but searching can be as bad as linear if the tree looks like a linked list.

Adding an element is also natural. As our tree represents a set, it will not contain the same element twice. We trace down until we hit a null pointer, and add the element in the appropriate spot. Let's add a `"p"` to the BST:

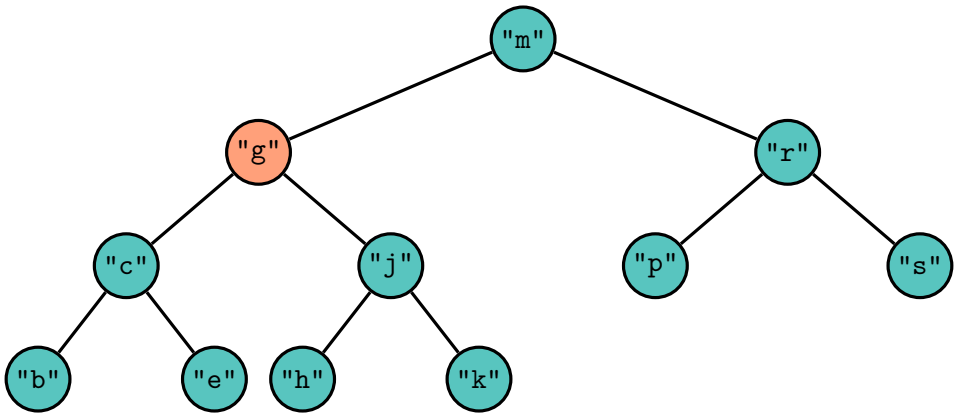


Deleting an element is the annoying part. Unfortunately, there's not much we can do besides casework.

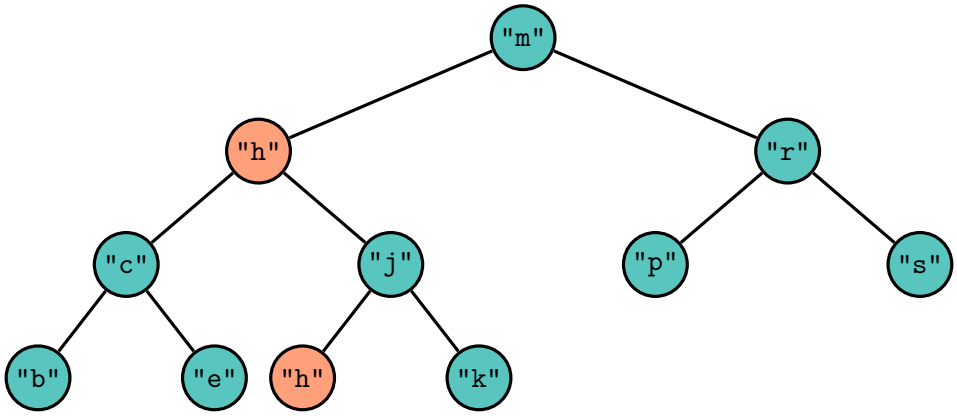
Removing a leaf, like `"a"`, from the tree is very easy. Removing a node with only once child, like `"t"`, is also relatively straightforward.

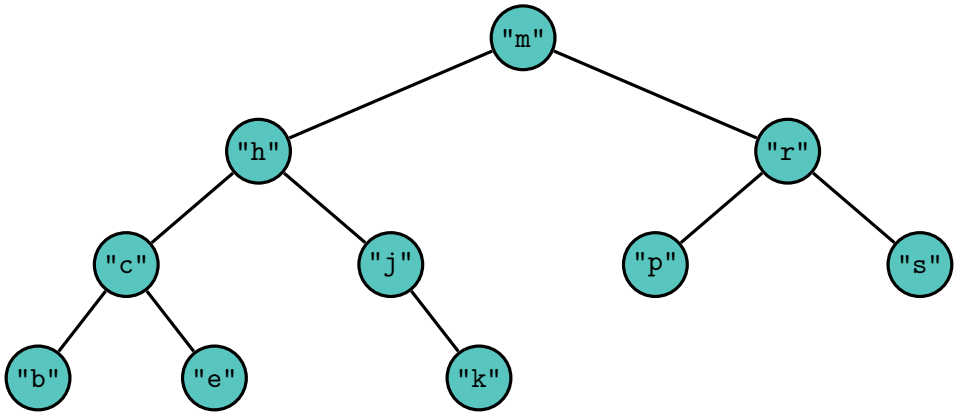


Now, removing an element with two children is tricky. We'll try to remove `"g"`. Consider the least element in the right subtree of `"g"`, which in this case is `"h"`. We find `"h"` by always choosing the left child on the right subtree until we cannot go any further. This must be the least element.



Note that `"h"` has either no children or only one child, and that nodes like these are easy to remove. We then change the value of the node containing `"g"` to `"h"`, which is legal since `"h"` is the least element, and remove `"h"` from the right subtree, and we are done.





Since a BST is ordered, iterating over it from left to right will pass over every element in sorted order.

A standard BST has $O(\log n)$ operations if the tree is "nice," or sufficiently randomized, but each operation can be $O(n)$ in the worst case. We need to find a way to automatically balance the BST such that we avoid linear time complexities.

A red-black tree is a self-balancing BST that guarantees $O(\log n)$ operations by making sure the height of the tree grows logarithmically. It is implemented in Java's `TreeSet` and is usually[10] implemented in the C++ `set`, so while the simple BST I described above does not guarantee nice time bounds, Java's implementation does.

I don't think learning exactly how a red-black tree works is particularly useful for the beginning programmer or a competitive programmer. How exactly a red-black tree works, together with some more balanced binary search trees which are useful on the competitive scene, are covered in a later chapter.

Function	Java, `TreeSet`	C++, `set`
insert	`boolean add(E e)`	`void insert (const value_type& val)`
search	`boolean contains(Object o)`	`size_type count (const value_type& val) const`[11]
delete	`boolean remove(Object o)`	`size_type erase (const value_type& val)`, `void erase (iterator position)`
size	`int size()`	`size_type size() const`
other useful *search* functions	`first`, `last`, `ceiling`, `floor`, `higher`, `lower`	`begin`, `end`, `lower_bound`[12], `upper_bound`[13]

[10] Implementations of `set` must be some kind of balanced binary search tree, but need not be red-black.
[11] 1 if in the `set`, 0 otherwise.
[12] `lower_bound` is similar to Java's `ceiling`
[13] `upper_bound` is similar to Java's `higher`

2.6.2 Hash Table

The fastest way to store values in a collection is through an array structure. Unfortunately, searching an array for a specific value is very costly. The binary search tree provided a quick way to search for a value by imposing constraints on its placement based on characteristics of the value. Similarly, to find a specific value in an array, we need to impose constraints on what index the value is placed in based on the value itself.

Suppose, for instance, I wanted to store some set of numbers from 0 to 999. I want to quickly add a new number to our set, remove a number, or check if a number is in our set. The easiest way to do this is to maintain a size-1000 array of boolean values. If a number i is in our set, we set the ith index in our array to true. We then have $O(1)$ updates and queries for our set.

To extend this to other values, we define the *hash function*. The hash function operates on the object and returns something that characterizes that object. For example, for a string, a possible hash could be the length of the string or the sum of the characters. We want to map an object with an integer hash, so that we can store the values by their hashes in an array. The resulting structure is a *hash table*.

What characterizes a good hash function?

1. If two objects are considered equal, like the strings `"Hello"` and `"Hello"`, their hashes must be equal.
2. If two objects are not equal, like the strings `"Hello"` and `"Bye"`, their hashes are only equal with very low probability. A *collision* is when two different objects have the same hash. We want to minimize the probability of this happening. As a result, hashes like the length of the string are not very good hashes.
3. A good hash should be reasonably fast to compute. One main purpose of hashing is to make equality checks between objects fast. A hash function that is hard to compute defeats the purpose of this.

Every Java `Object` supports the `hashCode()` function. By default, `hashCode()` stores information about the memory address of the `Object`. When we implement a new class, we can override this function. For example, let us define the following *polynomial hash* for strings:

```
public int hashCode() {
    int hash = 0;
    for(int k = 0; k < length(); k++) {
        hash *= 31;
        hash += (int) (charAt(k));
    }
    return hash;
}
```

In Java, `a.equals(b)` should imply `a.hashCode() == b.hashCode()`. This function produces the same result as the actual `hashCode()` function in the String class. However, this is not quite what we want for our hash set implementation, because in the end we wish to be able to store the objects in some kind of array. Since `hashCode()` can be any `int`, this hash not only returns integers

that can be very large, they can also be negative, and thus are not suitable as array indices. The natural way to fix this is to take the hash modulo the size of the array we want.

```
String[] table = new String[10007];
int index(E o) {
    int i = o.hashCode() % table.length;
    if(i >= 0)
        return i;
    return i + table.length;
}
```

The length of our array is somewhat arbitrary. We chose the number 100007 because it is a prime number, and primes are generally nice since integers modulo a prime form a field. Remember that a negative number `%` another number is not necessarily positive, so we need to be a little careful.

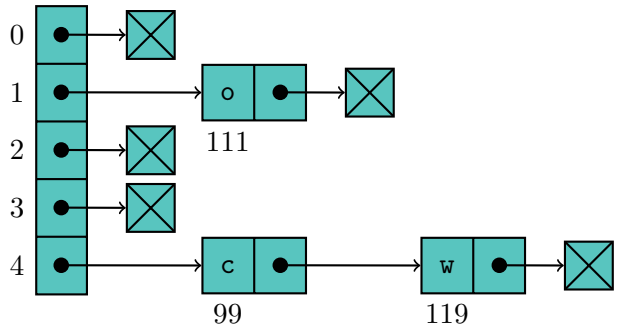
From here, adding an element to the table and checking if an element is contained both seem straightforward:

```
boolean add(E o) {
    table[index(o)] = o;
    return true;
}
boolean contains(Object o) {
    int i = index((E) o);
    return table[i] != null && table[i].equals(o);
}
```

`null` is always annoying to deal with, and will always have to be handled separately.

However, a problem quickly arises in the (hopefully unlikely) instance of a collision. If two strings have the same hash, we can't add both to the table since there isn't enough space in the array. The easiest way to handle a collision is by *chaining*. We change the hash table to store a linked list instead of a single element in the event of a collision. The hope is that not too many objects map to a single index in the array, as searching a linked list for a particular element is $O(n)$. Java once implemented this method of resolving collisions, but recently changed it to a BST in Java 8.

Here's an example of chaining on a small array of size 5 with the characters for "cow". The numbers below the letters represent their hashes. `c` and `w` collide.



If we use a good hash function and a reasonable array size, collisions will almost always be

pretty evenly spread across the array. Then, since we store everything using an array, the hash table provides probabilistic $O(1)$ time complexities for insertion, deletion, and search.

The Java set implementation of a hash table is the `HashSet`. The C++11[14] set implementation of a hash table is the `unordered_set`.

Function	Java, `HashSet`	C++11, `unordered_set`
insert	`boolean add(E e)`	`void insert (const value_type& val)`
search	`boolean contains(Object o)`	`size_type count (const value_type& val) const`[15]
delete	`boolean remove(Object o)`	`size_type erase (const value_type& val)`, `void erase (iterator position)`
size	`int size()`	`size_type size() const`

2.7 Map

A map is very similar to a set. A map is simply a function that takes a key to a value. Generics for maps therefore have two arguments: one for the key and one for the value. Consider the following Java `Map` from `String`s to `String`s.

```
Map<String, String> email = new TreeMap<String, String>();
email.put("Samuel Hsiang", "samuel.c.hsiang@gmail.com");
```

As a map is a function, its domain, or the keys of the map, form a set, though the values need not be unique.

`Map` is a Java interface. The `TreeMap` is the `Map` variant of the `TreeSet`; similarly, the `HashMap` is the `Map` variant of the `HashSet`. `map` is the C++ implementation of a balanced binary search tree map, while `unordered_map` is the C++11 implementation of a hash table.

Function	Java, `TreeMap`	C++, `map`
insert and *update* value	`V put(K key, V value)`	`mapped_type& operator[] (const key_type& k)`[16]
access	`V get(Object key)`	`mapped_type& operator[] (const key_type& k)`
delete	`V remove(Object key)`	`size_type erase (const key_type& val)`, `void erase (iterator position)`
size	`int size()`	`size_type size() const`
other useful *search* functions		just look in the API >.<

[14] added recently; not included in C++

[15] 1 if in the `set`, 0 otherwise.

[16] Works like an array or `vector`.

Function	Java, `HashMap`	C++11, `unordered_map`
insert and *update* value	`V put(K key, V value)`	`mapped_type& operator[] (const key_type& k)`[17]
access	`V get(Object key)`	`mapped_type& operator[] (const key_type& k)`
delete	`V remove(Object key)`	`size_type erase (const key_type& val)`, `void erase (iterator position)`
size	`int size()`	`size_type size() const`

2.8 Big Integer

A big int is for times when int and long just aren't large enough. The way it works is it stores a number as an array of ints. Each value in the array represents a digit in some very large base. Addition and subtraction can be done in the standard way. Generally multiplying two big ints is not necessary on contests, but it can be sped up using multiplication algorithms like Karatsuba.

`BigInteger` is in `java.math`, but C++ does not have an implementation.

[17] Works like an array or `vector`.