CS161 Programming Section

August 1, 2013

Hi everyone. Today we'll be covering just one problem, the minimum spanning tree problem, but we'll be writing up three different solutions to it. The first two solutions will be Kruskal's and Prim's algorithms exactly as described in lecture, while the third will be a variation on Prim's that uses less space but is far more involved to code.

Minimum Spanning Tree

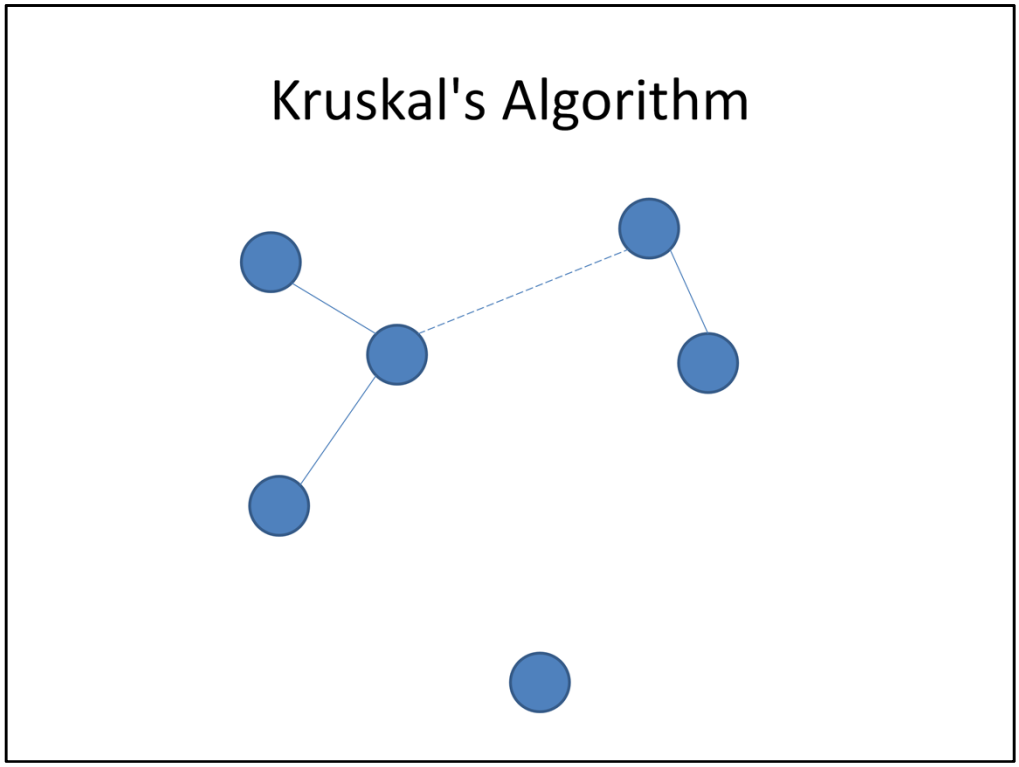
- Input
  - A weighted connected undirected graph with n nodes and m edges
  - $2 \le n \le 10000$, $1 \le m \le 50000$
  - all weights between 0 and 100000
  - nodes are 0-indexed, edges given as pairs (u, v) with weight w
- Input Format Per Case
  - Line 1: 2 integers n m
  - Lines 2 to m+1: 3 integers u_i v_i w_i
- Sentinel: n = m = 0
- Sample

```
3 3
0 1 1
1 2 2
0 2 3
0 0
```

- Output
  - The weight of the minimum spanning tree
- Output Format Per Case
  - Line 1: 1 integer
- Sample

```
3
```

Since we've covered the problem in detail in class, we can start with the specification slide. Notice that the largest graphs we're considering as input are quite sparse. Dense graphs tend to be less interesting, because if the graph is sufficiently dense, then just using an adjacency matrix and running Prim's without any special data structures gives you an optimal algorithm.



First, let's review how Kruskal's algorithm works. First, we take all the edges in the graph and sort them in increasing order of weight. Then, we go over the edges one at a time and add them to our set of edges if and only if the two endpoints are in different connected components. Notice that we don't need any of the standard representations of a graph; instead, we can get away with a simple array of edges, and a specialized data structure for keeping track of what component each node is in.

Union-Find

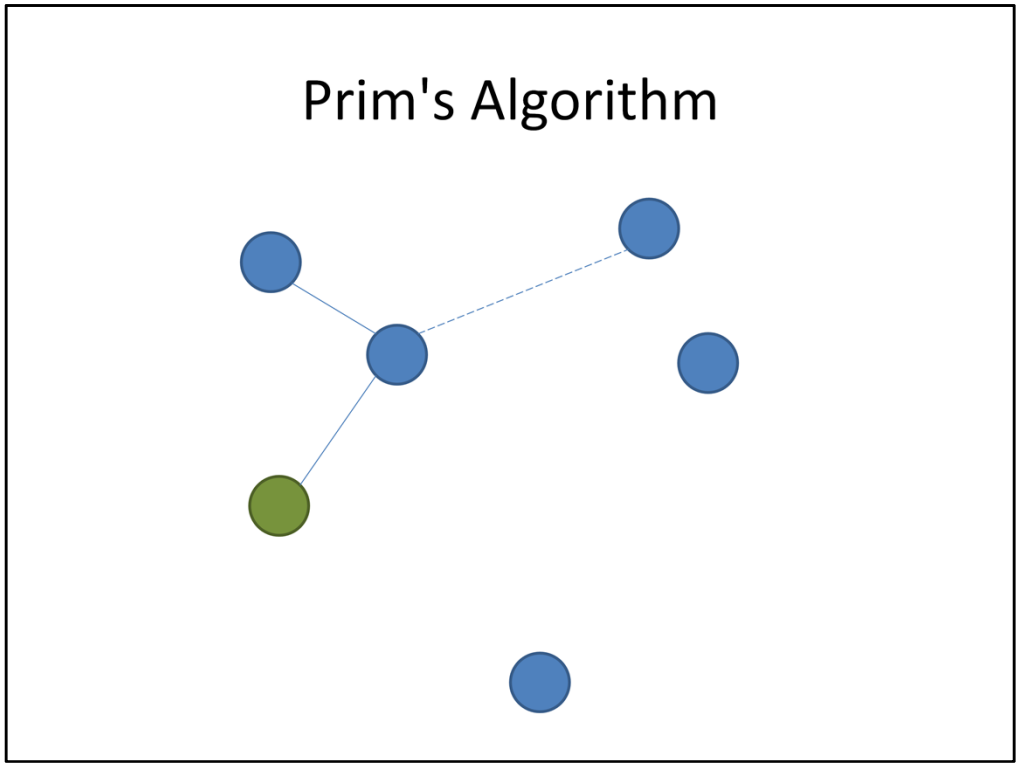
- Find(x): return the component of x
- Union(x, y): merge the components of x and y

This specialized data structure needs to support two operations. First, it needs to be able to find the component that a given node belongs to. Second, it needs to be able to merge two components together. The data structure we'll use to back these operations is the one we learned about in lecture, namely, the disjoint set forest.

Disjoint Set Forest

- Find(x): return the component of x
  - component has a representative called a root
  - each node has a parent; a root's parent is itself
  - if parent isn't root, set parent to be root
- Union(x, y): merge the components of x and y
  - each root has a rank, starting at 0
  - the higher rank root becomes parent of the lower
  - if equal rank, use either and increase rank by 1
  - non-root nodes have their rank ignored

The lecture notes provide an inspiration for this data structure; today, we're going to implement it with all the bells and whistles. Proving its runtime complexity is painful, but coding it up is not. All you need to do is maintain a parent and a rank for every node. You can do this either with a struct or with a couple arrays. The Wikipedia entry on Disjoint Set Forests provides very readable pseudocode if you want a more detailed starting point.



Now let's move on to Prim's algorithm. In this algorithm, we pick an arbitrary start node, say the green one here, and add all its adjacent edges to a priority queue. Then we extract the minimum weight edge from the priority queue, and each time we come across an edge that leads to a new node, we add it to our set, and then we add all of that node's adjacent edges that we haven't seen yet to the queue. For Prim's algorithm, the adjacency list representation will come in handy, since we need to be able to find all the edges incident to a node in order to add them to a queue.

Priority Queue (C++)

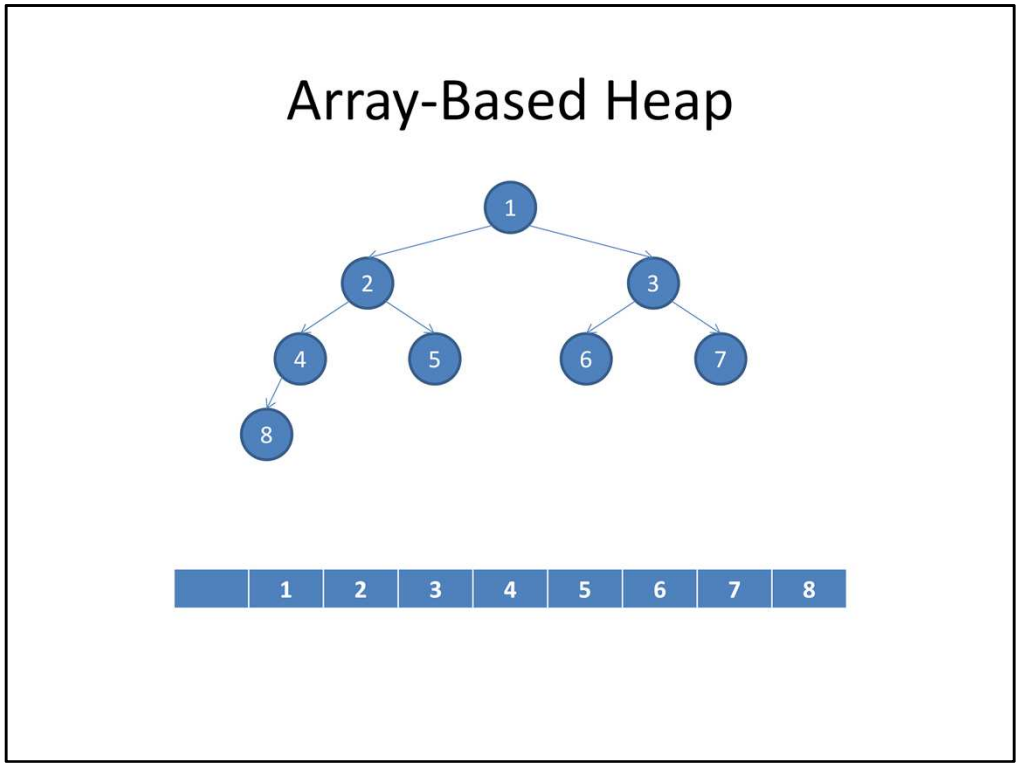
```
#include <queue>
...
struct edge {
  ...
  bool operator<(const edge &o) const {
    return weight > o.weight;
  } // use > for min heap, < for max heap
};
...
priority_queue<edge> pqueue;
pqueue.push(e);
edge min = pqueue.top();
pqueue.pop();
```

Both C++ and Java have an implementation of a priority queue backed by an array-based heap in their standard libraries. All you need to do in order to use them is to add a comparator to your struct. Notice that even though we're implementing operator-less-than, we're using a greater-than sign. This is because C++ priority queues are max-heaps, so if we want a min-heap, our comparator needs to go in the opposite direction.

Priority Queue (Java)

```
import java.util.*;
...
static class Edge implements Comparable<Edge> {
  ...
  public int compareTo(Edge e) {
    return this.weight - e.weight;
  } // use this - e for min heap, e - this for max heap
}
...
PriorityQueue<Edge> pqueue = new PriorityQueue<Edge>();
pqueue.add(e);
Edge min = pqueue.remove();
```

This is the way we make classes comparable in Java. Notice that here we return an int, not a boolean. A negative number indicates that we're smaller than the function argument, a positive number indicates that we're larger, and 0 indicates we're equal. A convenient shorthand for doing this is to take our weight and subtract the argument's weight. Be warned, though, that this shorthand is susceptible to overflow, so make sure the numbers you're comparing aren't too far apart.

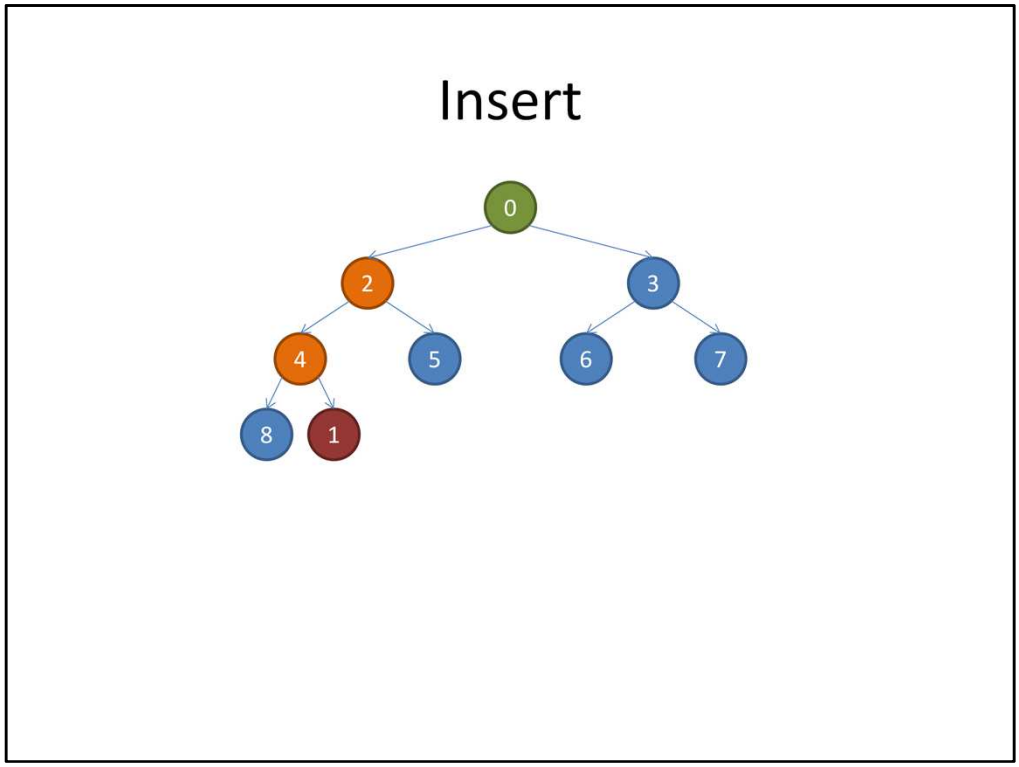


Now we're going to make a modification to Prim's algorithm to make it more space-efficient. Instead of inserting edges into our priority queue, we can insert nodes, and have the priority be equal to the minimum weight of an edge that connects that node to our tree so far. In order to make this work, though, we need to be able to decrease the priority of a node that's already in the queue. In theory, this can be done in logarithmic time, but it turns out this function isn't supported by the language default containers. The reason for this is it's a pain in the neck. Let's try it for ourselves by implementing our own heap that supports insert, extract-min, and decrease-key. As a reminder, a min heap is a dense binary tree that we fill layer by layer from left to right, where every node satisfies the heap property that it's smaller than all of its children. We can represent the heap implicitly by using a one-indexed array.

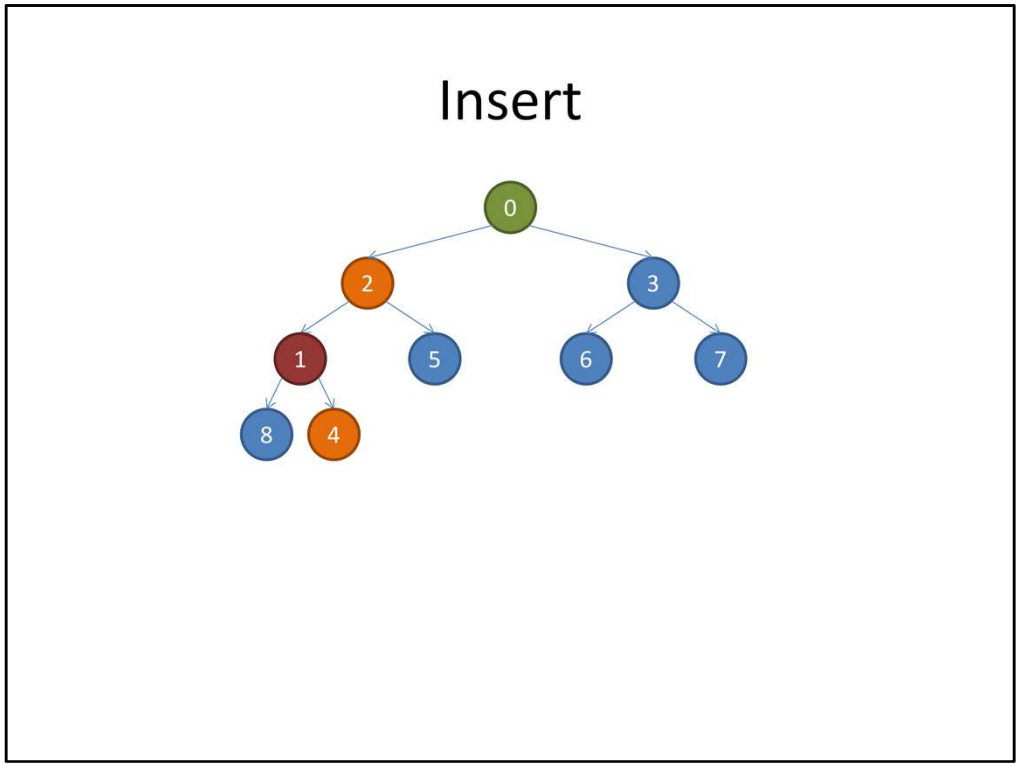
Heap Indexing

- valid indices are 1 through n inclusive
- parent of i = i / 2 (integer division)
  - alternatively, i >> 1
- children of i are 2 * i and 2 * i + 1
  - alternatively, i << 1 and (i << 1) + 1

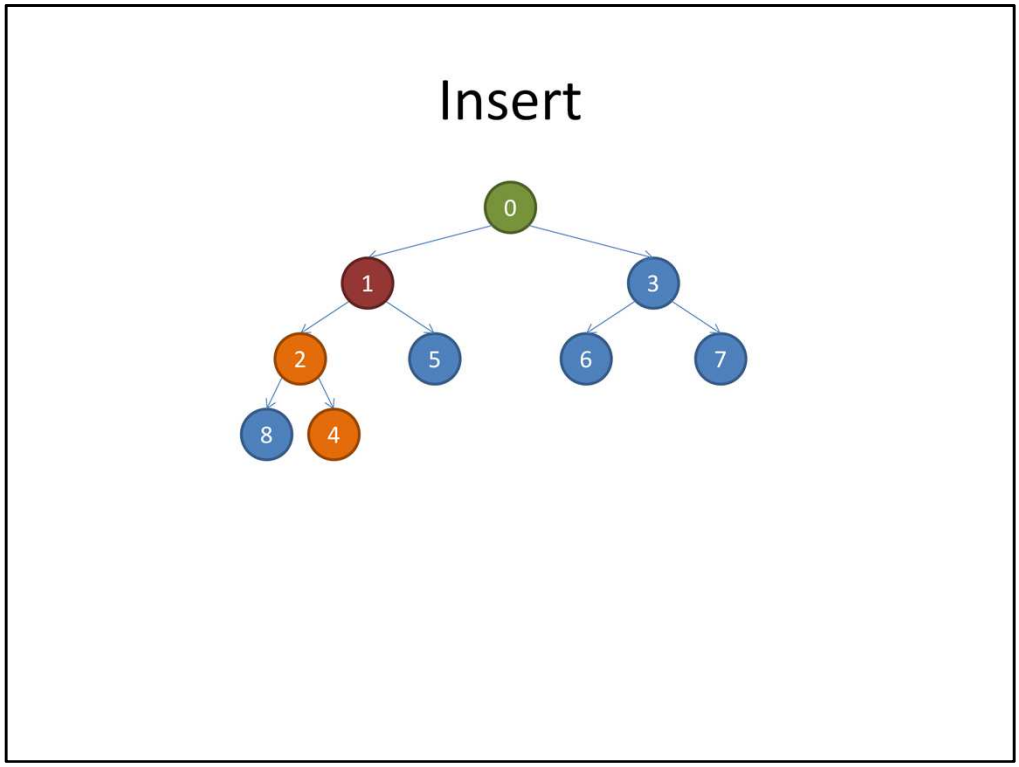
If we do that, then we have convenient ways of accessing the parent and children of a given node, namely by using division and multiplication respectively. Some of you may know from systems courses that division and multiplication are relatively expensive operations, so since we're always dividing or multiplying by two, we can use bit shifts instead for a little extra performance. On the other hand, modern compilers can do this replacement for you, so if you're not comfortable reading bit arithmetic, feel free to use the more common arithmetic operators.



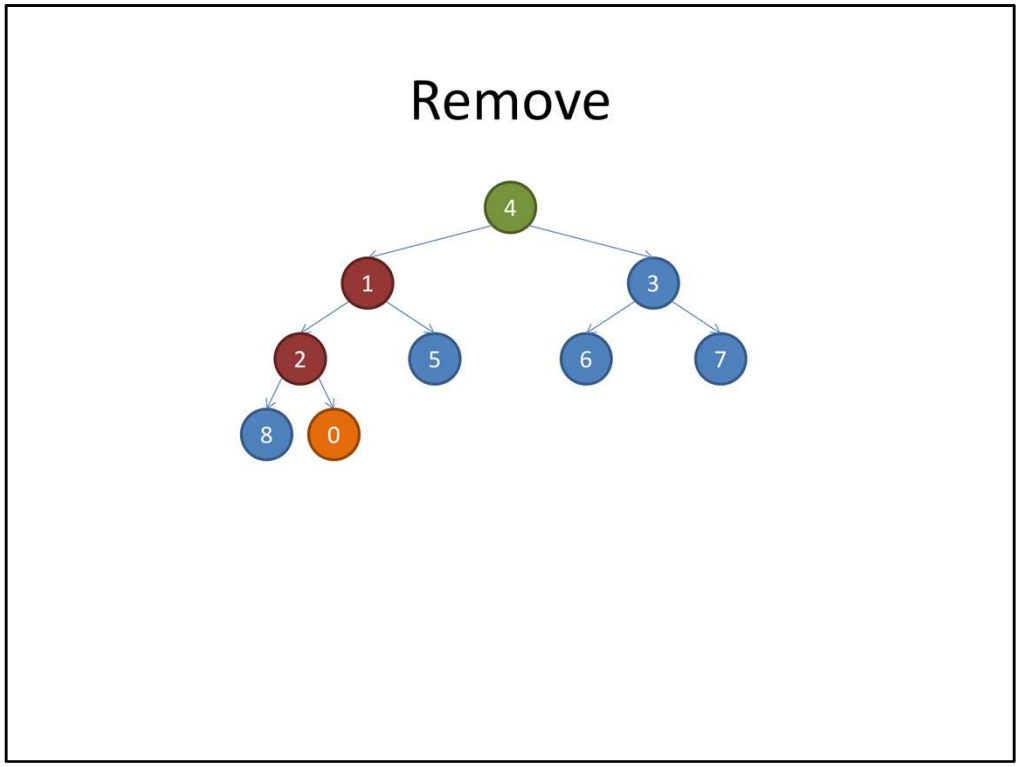
Now, how do we insert a node into the heap? The first thing we do is we add it to the next spot, which is at the end of the array. We then compare it to its parent, and if it's smaller than its parent, we swap it upward.



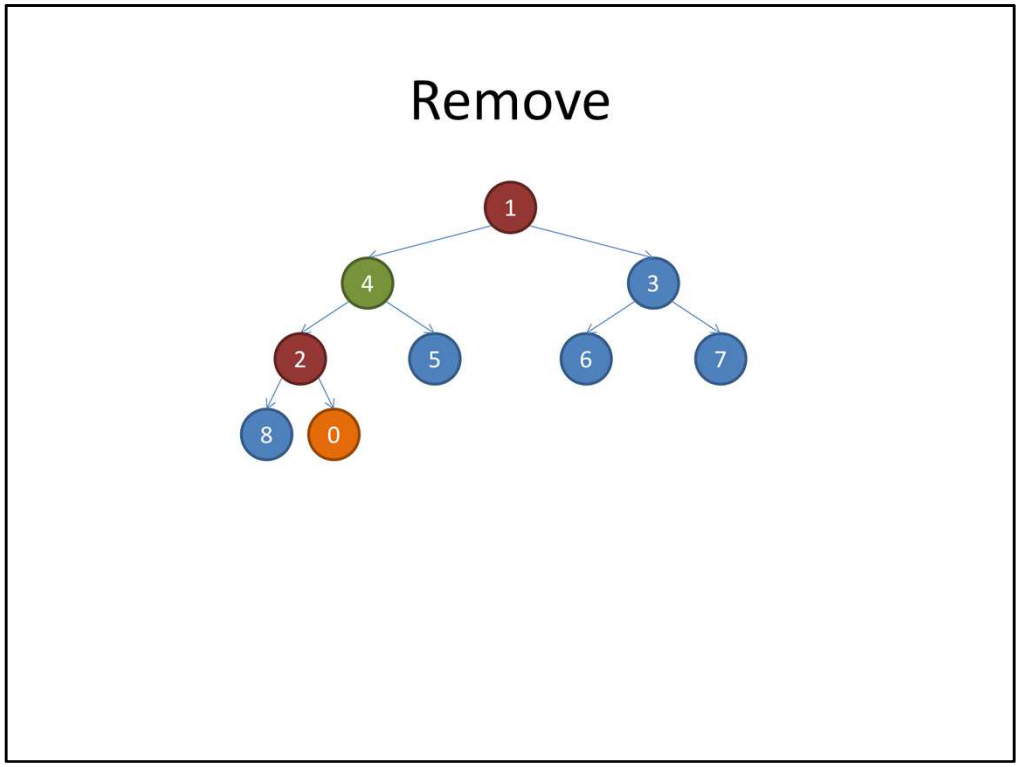
We repeat the process



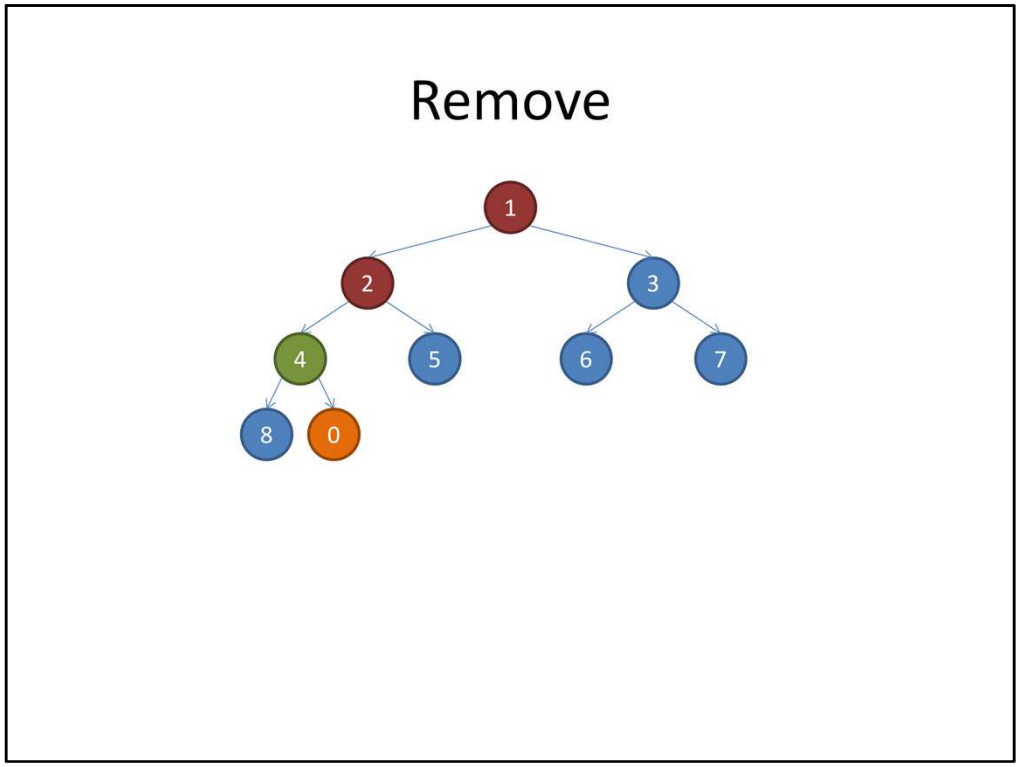
until its parent is smaller than it is, at which point we stop, knowing that we've preserved the heap property everywhere.



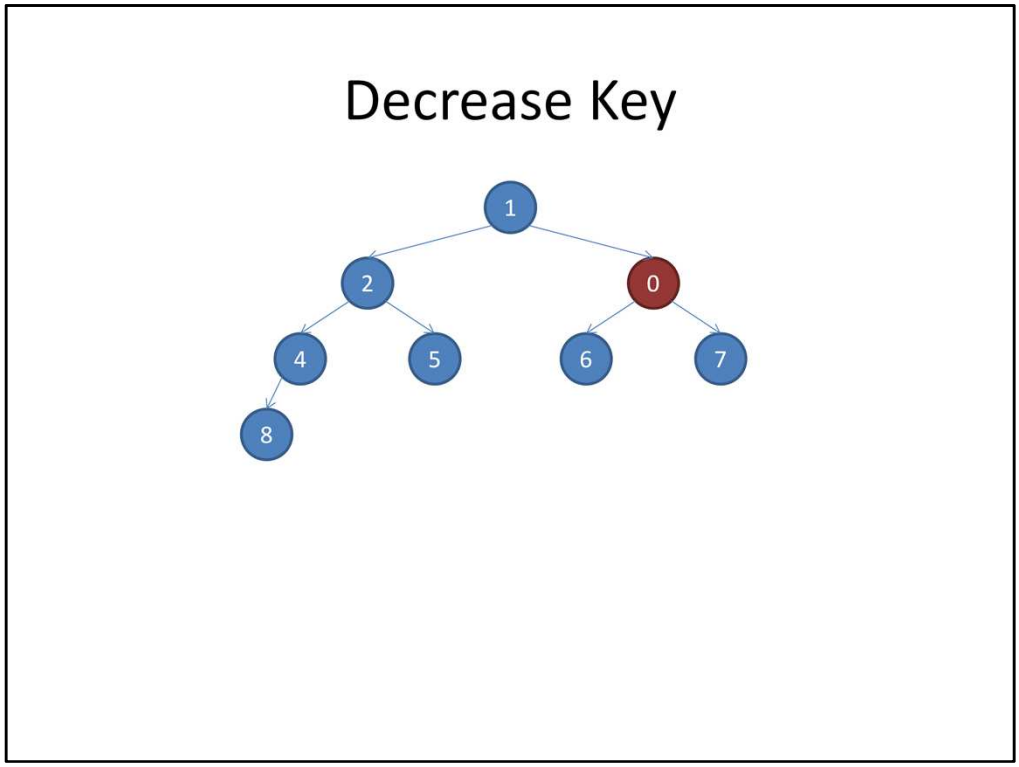
Now, to extract the minimum node, which is the 0 at the top of the heap, we first swap it with the last element in the heap. This new last element is going to be our return value, and we're not going to consider it as part of our heap anymore. Now, we have to fix the new root of the tree, as it may be larger than its children. We compare it with its children, and if it's larger than either of them, then we swap it with its smaller child.



We continue the process



until it is larger than all of its children. Notice that at this position, the 4 only has one child, namely, the 8; the 0 is no longer a part of our heap, and is going to be cut off and returned as the minimum, so the 4 is in the right place.



Now, let's look at what happens when we try to change the priority of an element in the heap, say the 3 from before to a 0. Notice that for our application, we only want to be able to decrease the priority, since we're using it to store the minimum weight edge that leads us to each new node. Now, if we lower a node's priority, we can be sure that it will stay smaller than its children. However, we run the risk of making it smaller than its parent. That means in order to preserve the heap property, we have to swap it upwards until it finds a suitable parent, much like we did for insertion. At first glance, it looks like getting log-time performance is straightforward. But we've neglected to mention one detail, which is we need to FIND the entry in the heap whose priority we want to decrease. If we don't store any additional information, we have no choice but to iterate over the whole heap in order to find the entry. So in order to get log-time decrease key, we need to maintain a map from elements to their locations in the heap. For our application, the things we'll be putting in the heap are going to be nodes that are 0-indexed up to n (exclusive), so we can use an array of size n to store the locations of the n nodes. If we do this, though, we need to make sure that we update this map whenever we swap entries in the heap.