Lecture Notes, CSE 232, Fall 2014 Semester

Dr. Brett Olsen

Week 2 - Data Structures

Alright, today we're going to be talking about data structures, what they are, why they're important for programming competitions, and how to use them effectively. I'll be covering mostly data structures that are useful for a wide range of problems; for those that are only useful for a particular subclass of problems, I'll get to them later when I talk about that subject.

A data structure is largely defined by two things: its interface and its implementation. The interface to a data structure tells you what kinds of things you can do with it - add or remove elements, choose a particular kind of element, etc. - while the implementation contains the details of how those methods and functions are actually performed.

For programming competitions, usually the interface is going to be what's most important to you. That's because often the implementation is tricky, both in terms of handling edge cases and in choosing the most efficient way to implement the interface. So generally, I strongly advise using a library for most common data structures rather than trying to implement one during the contest. You can either use pre-existing libraries like the `C++ STL` or preprepare a library of structures you think will be useful before the contest when you can debug them without time pressure. So for the purposes of this class, what I'll focus on is what the data structure supports and when you should use this structure rather than an alternative.

The only real exceptions to this advice are first, when you don't a good library for a particular structure you want to use and second, when you need to augment a data structure to use in your algorithm. The first can be rectified again, by preparing your own library outside of the competition, and we'll talk about the second, augmented data structures, later.

So, why should you care about data structures at all? Why can't we just store all the data we need in a fixed array or something like that and then write the algorithm using that? Well, usually you can, but choosing the right way to structure your data can often give you access to an interface with efficient methods that make the actual algorithm design completely trivial.

Here's a good example. Sorting is the canonical algorithm problem, and if you've taken an algorithms class you've probably learned a number of different ways to sort an array of n integers. Quicksort is the standard, and it works on a fixed array. But it is not at all trivial to implement correctly under a time constraint; the algorithm itself is complex with some important edge conditions. Suppose instead we had a data structure that gave us an efficient method called `find_min` that identified the minimum value in the data structure along with efficient `insert` and `remove` methods. Then here's an obvious way to write a sort algorithm: add all the numbers in our array to the data structure, then repeatedly find the minimum, put it in the next position in the sorted array, and delete it from our structure until we've gone through all n integers. This takes n calls to each of `find_min`, `insert`, and `remove`. If those are all $log(n)$ operations, our sort is done in $O(nlogn)$ time, just as fast asymptotically as quicksort, but with a much easier implementation.

```
In [11]: #Sample (unimplemented!) data structure sort
         class DataStructure(object):
             def __init__():
                 pass
             def find_min():
                 pass
             def insert(num):
                 pass
             def remove(num):
```

```
            pass

    def structure_sort(array):
        S = DataStructure()

        for item in array:
            S.insert(item)

        for i in range(len(array)):
            array[i] = S.find_min()
            S.remove(array[i])
```

Does anyone recognize this sort? In fact this data structure is called a heap or a priority queue and this sort is a well-known sort called a heapsort. The important thing to note here is that in the quicksort all the complex code is in the algorithm itself. But in the heapsort, all of the complicated stuff is in the data structure, which we won't have to deal with during the contest. We just use a heap. The sort itself is very simple - only 6 lines of code - and thus much less likely to have errors or bugs. So the way to use data structures is to look at your problem, see if you can identify methods that would make algorithm implementation simple, and then find a data structure that makes those methods simple.

Alright, let's get started on specifics.

```
In [12]: #Loading in some visualization libraries for later use
         import networkx as nx
         import matplotlib.pyplot as plt
```

Stacks and Queues Stacks and queues are one of the more basic data structures you'll use. The only thing that would be simpler would be an indexed static array. Just as an aside, one thing you'll discuss a lot in algorithms classes because they're useful for analysis are linked lists, arrays connected by pointers between elements. You should basically never use a linked list in a programming competition unless they are specifically required. You can get almost all of the benefit from a linked list with an indexed array and stored indices of the head and tail and linked lists are notoriously difficult to implement correctly. OK, aside finished.

Stacks and queues are very similar - they have only two methods, one which inserts elements and another which removes them, both of which are implemented so that they run in constant time, independent of the size of the structure. For historical reasons, these methods are called `push` and `pop` for stacks and `queue` and `dequeue` for queues. The only difference between them is in which element gets removed when you pop off of a stack or dequeue from a queue. In a stack, the last element to be pushed onto the stack is the first element to be popped off, just as if you had a stack of elements that you piled new ones onto and removed off the top. This is also known as Last In First Out (LIFO) order. In a queue, the first element to be queued is also the first element to be dequeued, just like a queue of people waiting in line - if you arrive in line first, you will be dequeued before anyone who arrives after you, also known as First In First Out (FIFO) order.

Stacks are often seen in recursive algorithms, even if only implicitly. If each task to be performed requires subtasks to be finished, we can push new tasks onto a common stack as they get generated and then pop each task off to be evaluated. Imagine, say, a program to evaluate the Fibonacci numbers, where $F_0 = 0, F_1 = 1$, and $F_n = F_{n-2} + F_{n-1}$. We could calculate this with explicit recursive calls, or we could do so with a stack of Fibonacci evaluations where we repeatedly remove tasks and replace them with their recursive sums if needed or the base values if they're small enough. In fact the explicitly recursive call contains a stack under the hood - the program holds a stack of function calls to be evaluated that will expand and collapse in exactly the same way.

```
In [13]: def fib_recursive(n):
             if n > 1:
                 return fib_recursive(n - 1) + fib_recursive(n - 2)
             else:
                 return n
```

```
In [14]: def fib_stack(n):
             stack = []
             stack.append(n) #append == push
             running_sum = 0
             while stack:
                 num = stack.pop()
                 if num > 1:
                     stack.append(num - 1)
                     stack.append(num - 2)
                 else:
                     running_sum += num
             return running_sum

In [15]: print [fib_recursive(i) for i in range(10)]

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

In [16]: print [fib_stack(i) for i in range(10)]

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Queues, on the other hand, are often used where the processing order is important - these steps have to be performed in a particular order the same as we put them in. For competitions, look to see whether the order is important to see whether to use a queue or not. We'll also see these later when we do graph theory as a way to structure breadth-first searches.

If you need access to *both* the oldest and newest element, there's another structure called a double-ended queue, or dequeue.

Takeaway:

- Both insert and remove operations should be constant time ($O(1)$) for both queues and stacks.
- Use when you want to process elements in order of newest (stacks) or oldest (queues)
- Use a stack to explicitly represent a recursive algorithm

Data structure implementations for stacks, queues, and dequeues:

- C++ STL libraries:
  - `stack`
  - `queue`
  - `deque`
- Java libraries:
  - `Stack`
  - `Queue`
  - `Deque`
- Python libraries:
  - `[]` (native lists)
  - `collections.deque`
  - `collections.deque`

Dictionaries, Hash Tables, and Binary Search Trees Unlike a fixed array, where elements are accessed by index, or a stack or a queue, where elements are accessed in some particular order, dictionary structures allow access of elements by content, usually with a key k.

A dictionary interface will implement fast ($O(1)$ or $O(logn)$) methods for:

- `search`, which, given a search key k, returns the element in the dictionary whose key value is k, if it exists
- `insert`, which, given a key k and item x, adds it to the dictionary so that `search(k) = x`
- `delete`, which, given a key k, removes the element associated with that key from the dictionary

The classic use case of a dictionary data structure is as an actual dictionary - that is, valid words for some purpose, where the words are keys and the elements associated with them are trivial - say 1. This is better than say, simply a separated list of strings, because searching for a particular key is much faster using a dictionary structure.

Here's an example: suppose we have a string "peanutbutter" that we want to split into two words, where each word must be one of those in a long text file provided as `dictionary.txt`:

```
pea
nut
falafel
peanut
butt
butter
```

For a short input string, we might simply iterate through each way of splitting the string: first "p" and "eanutbutter", then "pe" and "anutbutter", then "pea" and "nutbutter", and so on. We'll then take each of these string pairs and look for them in the text file, by going through each line of the file and seeing if they're present. This works and you'll get the right answer. Unfortunately, this will take longer the more words are in the input file. If the string isn't a valid word, we'll have to iterate through the whole input file to test that. If we've got 10 million possible words, this quickly gets infeasible. So instead, we'll preprocess the text file - loop through each line and insert it into a dictionary structure, where the key is the word and the element is 1, say. Then we can use dictionary operations to quickly test whether or not a particular word is there. Here's an implementation in Python:

```
In [17]: wordlist = ['pea', 'nut', 'falafel', 'peanut', 'butt', 'butter']
         #build a dictionary
         D = {}
         for word in wordlist:
             D[word] = 1

         def splitwords(S, D):
             for i in range(1, len(S) - 1):
                 a, b = S[:i], S[i:]
                 if a in D and b in D:
                     return a, b
             return False

In [18]: print splitwords("peanutbutter", D)
         print splitwords("peanutsbutter", D)
         print splitwords("peanut", D)

('peanut', 'butter')
False
('pea', 'nut')
```

Other possibilities for dictionaries are counts associated with particular characters or words or any other kind of key - any situation where you would want to keep track of values or numbers associated with keys that aren't simply indexes.

As usual, I'm not going to go into great detail on implementation because there are libraries with good implementations of dictionaries available. Usually, a dictionary is going to be implemented with either a hash table or a balanced binary search tree.

Briefly, hash tables allocate a static array of size m, where $m >> n$, the expected number of elements in the dictionary. It then uses a mapping function to map key k to an integer i and then a hash function to map i to an index j, where $j \in [0, m)$. That is, it assigns each key to a particular slot in the static array. If the array size is large enough, collisions where two keys map to the same slot should be fairly rare, and access to a particular key is constant time depending only on the speed of the mapping and hash functions. Be wary - hash functions are easy to get wrong so that the mapping isn't uniform, with lots of keys mapping to the same slots, so performance can degrade.

Balanced binary search trees will work well for keys that are easily comparible to each other, like strings or integers. Each node in the tree has a key associated with it, and the left child of a node always has a key less than the parent, while the right child always has a key greater than the parent. That means you can find a particular key by looking at the root, seeing if the key you're searching for is greater than or less than the key of the root, and then recursing on the left or right subtree. This takes time dependent on the depth of the tree, which if the tree is balanced, should be $O(logn)$. Be careful because *balanced* search trees are hard to guarantee. You'll almost always want to use an existing implementation like a red-black tree or an AVL tree.

Takeaway

- Dictionaries should give $O(1)$ (hash-tables) or $O(logn)$ (BSTs) access to insert, remove, and location methods
- Use where keys of your elements aren't just indices from 0 to n (where an array would work fine)
- Allows very fast testing of element presence scaling well with size of the dictionary
- Think very very hard before deciding to implement your own dictionary during a contest. Both hash tables and balanced binary search trees are very easy to get wrong.

Data structure implementations for dictionaries:

- C++ STL libraries:
  - `map` (binary search tree)
  - `unordered_map` (hash table, may not always be supported)
- Java libraries:
  - `TreeMap` (binary search tree
  - `HashMap` (hash table)
- Python libraries:
  - {} (native dictionary type)

Heaps and Priority Queues Alright, on to the next structure - priority queues and heaps. Here priority queue describes the *interface* and heap describes the *implementation*. Just as I described it earlier, priority queues support a method that can extract either the minimum or the maximum value contained in the structure, depending on implementation. The implementation is usually done as a *complete binary tree* with the particular property that each node is always greater than (for a max-heap) or less than (for a min-heap) both of its children. This means that the root of the tree is always the largest (or smallest) value in the heap.

```
In [37]: def draw_heap(heap, outfile=None):
             """Draw the heap using matplotlib and networkx"""
             G = nx.DiGraph()
             G.add_node(0)
             for i in xrange(1, len(heap)):
```

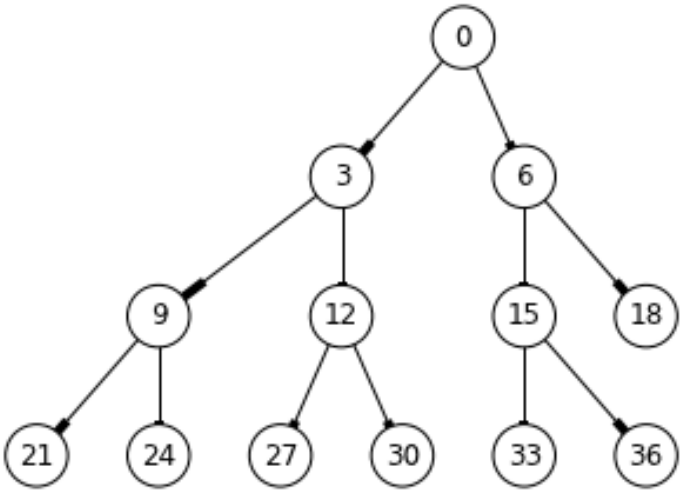
```
            G.add_node(heap[i])
            G.add_edge(heap[(i - 1) / 2], heap[i])

        labels = dict((u, "%d" % (u)) for u in G.nodes())

        nx.draw_graphviz(G, prog='dot', labels=labels, node_size=700, node_color='white')
        return G
```

```
In [38]: import heapq
         heap = (np.arange(13) * 3).tolist()
         heapq.heapify(heap)
         draw_heap(heap)
         print heap
```

```
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36]
```



When you remove the minimum, you simply remove the root, promote its smallest descendant, and continue on down the line, while maintaining the tree's completeness by a series of tree rotations. Similarly, to add a new element insert it as a leaf at the next position, then promoting it and rotating until the heap property is conserved. I'm being deliberately vague here because I don't want to focus on the details - as I said at the beginning, you shouldn't be focused on the implementation details during the contest. Instead what I want to focus on is when you would want to use a priority queue. I mentioned earlier that heaps are quite useful for sorting static data - I have an array of n elements, and a heap lets me sort those in $O(nlogn)$ time just by trivial use of the interface.

But suppose that you instead have a dynamic list of elements, say tasks to be performed, where new tasks can be added during execution (say by recursive calls). Different data structures allow fast identification of the next task to be done in different ways:

- Stacks let you quickly find the *newest* task

- Queues let you quickly find the *oldest* task
- Priority queues let you choose the *best* task based on a numerical priority you assign

As such, priority queues are ideal for any algorithm that requires you to process items in a particular order that's not the same as the order we insert them into the queue. We'll just attach a priority to the item that describes how we want to process it and then remove items from the queue based on its priority. Priorities won't be seen often in isolation, but they're key components to lots of more complex algorithms, in particular graph theory problems where you need to repeatedly find, say, the shortest edge in a graph.

Takeaway

- Priority queues give $O(logn)$ access to insert and remove-max/remove-min methods
- Use when you want to process items based on some numerical priority (path length, task cost, etc.)

Data structure implementations for priority queues:

- C++ STL libraries:
  - `priority_queue`
- Java libraries:
  - `PriorityQueue`
- Python libraries:
  - `heapq` (package, use methods on lists to maintain heap properties)

Union-Find Union-find structures are used when you want to keep track of a number of disjoint sets on a common group of elements. Say these elements are all in group 1, while these elements are all in group 2 and I need to be able to quickly tell what set a particular element is in. It's called a union-find because those are the two interface methods it supports:

- `union`, which given two elements, merges the sets containing them
- `find`, which given an element, returns the set to which that element belongs

Let's take an example array

```
A = [1, 2, 3, 4, 5, 6, 7]
```

where the disjoint sets are (1, 4, 5), (2, 6), (3), and (7). One way you could implement this interface would be to number the sets and create a supplementary array marking which group each element belongs to:

```
S = [1, 2, 3, 1, 1, 2, 4]
```

Then `find` would clearly run in constant time - all we have to do is look it up in S - but how long would a `union` operation take. Suppose, for example, we want to merge groups 3 and 4. We can do that by going through each element of S, find elements belonging to group 4, and changing them to group 3 instead, which will take linear time. Unfortunately, this isn't good enough. A common use case for this kind of structure is starting with each element in its own disjoint set and serially merging sets until they're all in the same set. In the worst case, this can take as many as n `union` operations, leaving us with an $O(n^2)$ algorithm.

To resolve this, instead of marking groups arbitrarily, let's think of another way to mark groups. We'll pick an arbitrary member of each group to be its representative. Then instead of using our supplementary array to mark group numbers, we'll use it to indicate the representative of the group that element is associated with. So if we pick the representatives 4, 6, 3, and 7, we'd have an array like:

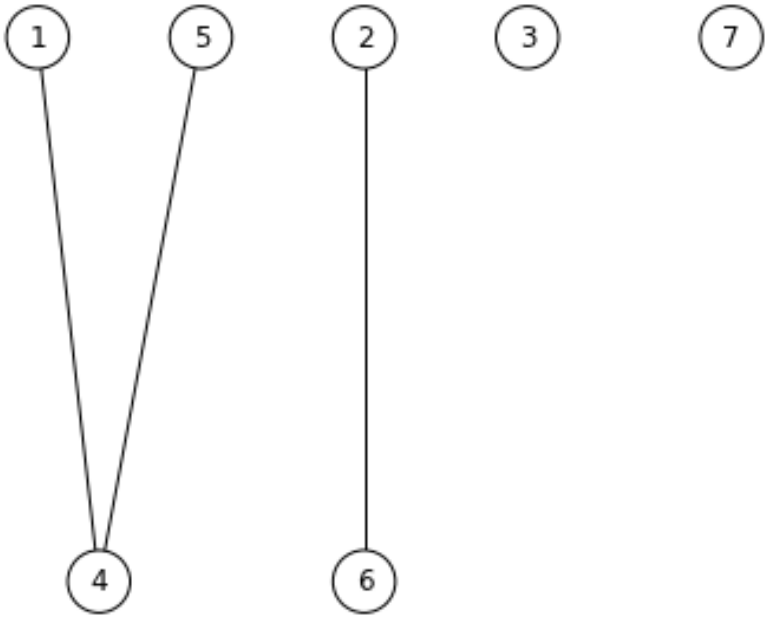
```
S = [4, 6, 3, 4, 4, 6, 7]
```

We can think of this as a forest of trees, with each tree representing a set and root of each tree being that set's representative:

```
In [46]: G = nx.DiGraph()
         A = [1, 2, 3, 4, 5, 6, 7]
         S = [4, 6, 3, 4, 4, 6, 7]
         for element in A:
             G.add_node(element)
         for i, root in enumerate(S):
             G.add_edge(i+1, root)
         labels = dict((u, "%d" % (u)) for u in G.nodes())

         nx.draw_graphviz(G, prog='dot', node_color='white', node_size=700, arrows=False)
         print S
```

```
[4, 6, 3, 4, 4, 6, 7]
```



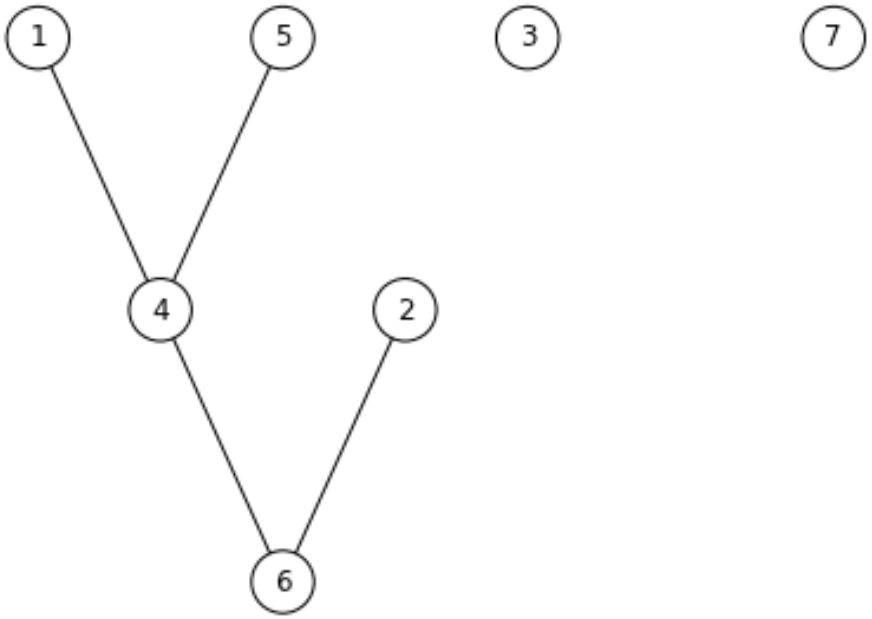
So for a `find` operation, we start at our element of interest. If it points to itself, then we return it. Otherwise, we call `find` on the element it points to. This will give us the representative of the set for every element in the set, and will run in time proportional to the depth of the trees. `union` operations can be done by first finding the representatives of the two elements to be merged, and then changing the representative of one to point to the other. For example, if we called `union(1, 2)`, the representatives of those sets are 4 and 6 respectively. We would then change 4 to point not to itself but instead to 6, or vice versa:

```
In [47]: G = nx.DiGraph()
         A = [1, 2, 3, 4, 5, 6, 7]
         S = [4, 6, 3, 6, 4, 6, 7]
         for element in A:
```

```
            G.add_node(element)
        for i, root in enumerate(S):
            G.add_edge(i+1, root)
        labels = dict((u, "%d" % (u)) for u in G.nodes())

        nx.draw_graphviz(G, prog='dot', node_color='white', node_size=700, arrows=False)
        print S
```

```
[4, 6, 3, 6, 4, 6, 7]
```



Now 6 is the representative of the merged group. With a bad choice of `union` operations, this procedure can give you trees of depth $O(n)$, which as both `union` and `find` run in time based on the maximum depth of the trees, is bad. To fix this, there are usually two supplemental tricks used: first, when merging two sets, keep track of the depth of each tree and make sure we make the deeper tree the represenative of the merged set. Second, we can do path compression during `find` operations - every time we run a `find` we update S to point directly to its representative rather than going through any intermediates.

Takeaway

- Supports *constant* time ($O(1)$) testing of set composition
- No good way to split sets - think if that's something you'll need to do
- Simple implementation, no standard library support, so think about writing your own ahead of time
- If you do, use some tricks (path compression and height-dependent root merging) to ensure good performance
- Use when you need to quickly answer whether two items are in the same set, or ask what set a particular item is in

Segment Trees Segment trees are optimized for doing calculations on intervals. For this one, the implementation is actually going to be important as there aren't great libraries out there, so I'm going to start with that.

Suppose we have an array A with n elements that we know is going to be dynamically changing through the course of our problem. We're going to want to repeatedly query the array to know what is the minimum value over some interval $[i, j]$. The brute force way to solve this problem is to simply iterate over each element of the array from i to j and find the minimum in that interval. Unfortunately, this solution is often too slow - it takes $O(n)$ time for each query. What kind of data structure can let us answer this query more efficiently?

A segment tree holds a subset of the intervals on A so that every possible interval we might query is representable as the union of a small number of intervals present in the tree. We'll build it as a complete binary tree where the root of the tree is $[0, L]$ - the interval covering the whole range of A. Then the children of each node will split the interval of their parent into two disjoint equal-sized pieces. So if our root node is $[0, 6]$, it will have children $[0, 3]$ and $[4, 6]$. We then keep splitting the intervals until the leaves of the tree are intervals that contain only a single element. Then we calculate minimum of our array over each interval in the tree. The leaves of the tree are trivial - the minimum is simply the value of the element at each position. The minimum at larger intervals are then calculated by taking the minimum of the value at each of its children.

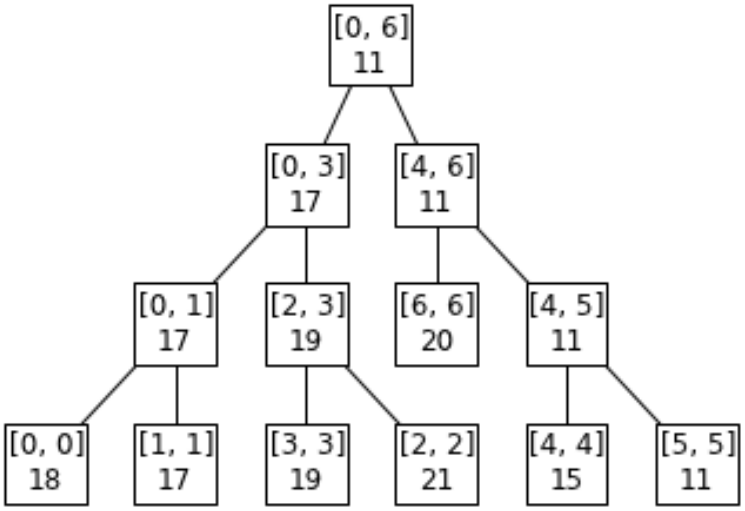
Once we've built this structure in $O(nlogn)$ time, we can now do two operations very quickly: update the values in A and find the minimum over an arbitrary interval. Updates are done by changing the value of the leaf associated with the element being updated and then cascading upward as we did originally. Interval queries are done by interval splitting. Let's look at an example.

```
In [98]: A = [18, 17, 21, 19, 15, 11, 20]
         idx = [0, 1, 2, 3, 4, 5, 6]

         #This is just visualization code!  Don't try to take this as actually building a segment tree!
         def build_segment_tree(i, j, G):
             G.add_node((i,j))
             if i == j:
                 return
             mid = (i + j) // 2
             build_segment_tree(i, mid, G)
             build_segment_tree(mid+1, j, G)
             G.add_edge((i,j), (i,mid))
             G.add_edge((i,j), (mid+1,j))

         G = nx.DiGraph()
         build_segment_tree(idx[0], idx[-1], G)
         #TODO - fix layout so L/R paths work correctly
         labels = dict((u, "[%i, %i]\n%i" % (u[0], u[1], min(A[u[0]:u[1]+1]))) for u in G.nodes())
         print A
         nx.draw_graphviz(G, prog='dot', labels=labels, node_color='white', node_size=1200, arrows=Fals

[18, 17, 21, 19, 15, 11, 20]
```



Now suppose we want to do a query on this tree using the interval $[1, 4]$ - that is, what's the minimum value in the array between those indexes, inclusive? We start at the root of the tree, and at each node there are three possibilities:

- The interval we're querying is identical to that of the node we're at, in which case we simply return the minimum associated with this node
- The query interval is larger than the node we're at and the midpoint of the node's interval is within the query interval. In this case, we split the query interval at this midpoint and recurse the resulting left and right query intervals down to the left and right children of this node, returning the minimum returned by both of these recursions.
- The query interval is larger than the node we're at at the midpoint is outside the query interval. In this case, we simply recurse the query interval down to the appropriate node.

So for the query interval $[1, 4]$, this gets split into the disjoint intervals $[1, 1]$, $[2, 3]$, and $[4, 4]$ by recursing down the tree, with the minimum over the query returned as the minimum over these ending nodes.

Anytime you see a problem where rapid calculation over intervals would be useful, think about whether a segment tree would solve the problem. It's trivial to implement this using other queries - maximum over a range, sums over a range, products over a range - are all easily manageable.

Takeaway

- Efficient performance of range minimum query (RMQ) operations on dynamic arrays
- Can be extended to other queries - maxima, sums, products over an interval range
- Complex to implement - do so ahead of time and include extensive bug-testing!
- Applicable to lots of different interval problems!

Lab Work

- TOJ 1067 - Disk Tree (an implementation problem)

- TOJ 1126 - Magnetic Storms (an algorithm problem)