Comp 250 Study guide

Francis Piche

November 10, 2017

Contents

Part I
Mathematical Tools for Algorithm Analysis

1 Solving and Understanding Recurrences

How can we know the time it takes for a recursive algorithm to run?
Recurrence relations!!

1.1 Basic Idea

The idea behind these problems is to look at the algorithm at every step and put down the time it would take as an equation. We write the relation like this:

- $t(n)$ represents the time taken.
- Every operation has constant, unit-less time. Say, 1.
- Recursive calls are represented as $t(n')$ where n' is the input of the recursive call.

To find $t(n)$ as a constant expression, we use the method of back substitution, and try to see a pattern. Once the pattern is established, we can follow it down all the way until the base case.

Honestly the best way to understand this is to do lot's of examples.

1.2 Simple example

Problem 1. *How long does it take to reverse a list of n elements recursively?*

Solution. It seems like a big question, but lets break it down.

- Remove the first element
- Reverse the rest of the list recursively

So this will take time:

$$t(n) = 1 + t(n-1)$$

So, we can back-substitute like this:

$$t(n) = 1 + (1 + t(n-2))$$
$$t(n) = 1 + (1 + (1 + t(n-3)))$$

we can already notice a pattern developing, it will take n substitutions to get down to the base case, $t(1)$, so there will be n constant 1's. so we have:

$$t(n) = n + t(1)$$

In this course, we always assume t(1) is 1.

$$t(n) = n$$

So this takes time proportional to $O(n)$

The idea is always the same, we just get to more and more complicated examples.

1.3 Useful tools, identities and tips

Here are some useful tools and tricks you might need:

- In comp 250, we can always assume n is a power of 2. This makes certain identities easier to use.
- If the recursive call is $t(n/2)$, it will take $log_2(n)$ iterations to reach the base case. (This doesn't mean it will take $log_2(n)$ time!!)
- If you see something that looks like the sum of powers of a constant, use the geometric series:

$$\sum_{i=0}^{N-1} a^i = \frac{a^N - 1}{a - 1}$$

- Once you've gotten something with all constant terms, look at the largest term involving n to find O().
- $log_b(n) = log_a(n) log_b(a)$
- $a^{log_b(c)} = c^{log_b(a)}$
- The other, more common laws of logs may come in handy as well.
- Do lot's of practice, it's the only way to master this.

1.4 Important Examples: MergeSort + QuickSort

Recall the pseudocode for mergesort:

```
mergesort(List list){
   if {list.length==1}
      return list;
   else{
      mid = (list.size-1)/2
      list1 = list.getElements(0,mid);
      list2 = list.getElements(mid+1,list.size-1);
      list1 = mergesort(list1);
      list2 = mergesort(list2);
      return merge(list1,list2);
   }
}
```

We can see that we call mergesort twice, and we're calling it on a list that is now roughly $\frac{n}{2}$ long. So our recurrence relation will have a $2t(\frac{n}{2})$ term in it.

Notice also that we have a constant amount of work acting on n elements in order to merge them. So we will have a cn term. Where c is a constant.

So our relation is:

$$t(n) = cn + 2t(\frac{n}{2})$$

Back substituting:

$$t(n) = cn + 2(c\frac{n}{2} + 2t(\frac{n}{4}))$$

$$t(n) = cn + cn + 4t(\frac{n}{4})$$

$$t(n) = cn + cn + 4(c\frac{n}{4} + 2t(\frac{n}{8}))$$

We see a pattern begin to emerge:

It will take $log_2(n)$ iterations, so there wil be $log_2(n)cn$ terms, and a power of 2 multiplying the $t(1)$ term, and since n is always a power of 2, we have an n term.

$$t(n) = cnlog_2(n) + n$$

Which is $O(nlog_2(n))$

Prof. Langer makes a point in his notes to pay attention to the fact that if the base case is difficult to compute, we may use a simpler, slower algorithm (like bubblesort) to solve it, since this still takes a constant amount of time, and doesn't introduce an n^2 dependence (since bubble sort takes $O(n^2)$.

Recall the pseudocode for Quicksort:

```
quicksort(List list){
   if (list.length <=1){
      return list;
   }else{
      pivot = list.removeFirst(); //or some other element
      list1 = list.getElementsLessThan(pivot);
      list2=list.getElementsNotLessThan(pivot);
      list1 = quicksort(list1);
      list2 = quicksort(list2);
      return concatenate(list1,pivot,list2)
   }
}
```

Recall that depending on our choice of pivot, this algorithm can be very quick, or very slow. In the best case, it divides the list in two almost evenly, in which case it behaves much like mergesort. In the worst case, the pivot is the max or min value of the list, and divides the list into itself, and the rest of the list.

If this bad split happens at every level of the recursion, it takes time $O(n^2)$.

Note that the bad split causes the lists to be of size 1 and $n - 1$.

Also note that comparing the pivot to each element in the list takes n operations.

Proof that worst case is $O(n^2)$:

$$t(n) = cn + t(n-1)$$

$$t(n) = cn + c(n-1) + t(n-2)$$
$$t(n) = cn + c(n-1) + c(n-2) + t(n-3)$$
$$t(n) = cn + c(n-1) + c(n-2) + c(n-3) + t(n-4)$$
$$...$$
$$t(n) = c\frac{n(n+1)}{2} + t(1)$$

Which is $O(n^2)$

So why is quicksort "quick"?

- Choose the pivot by taking elements first, mid, last, and finding the median. This makes the worst case extremely improbable.
- It can be done "in-place", takes up MUCH less memory than mergesort.

2 Big O, Big Ω, and Big θ

2.1 Semi-formal Definition

For two functions, $t(n)$, $g(n)$ we say that $t(n)$ is $O(g(n))$ if there exists an n_0 such that for all $n \geq n_0$, $g(n) \geq t(n)$

This basically means that beyond a certain point, n_0, then the function $g(n)$ is "bigger" than $t(n)$

Problem 2. *Prove that* $5n + 70$ *is asymptotically boudned above by* $6n$

Solution. We have:

$$5n + 70 \leq 6n \text{ for suffiently large n.}$$

$$\Leftrightarrow 70 \leq n$$

So, for $n \geq 70$, $5n + 70 \leq 6n$, simple right?

2.2 Formal Definition of Big O

Let $t(n)$ and $g(n)$ be functions and $n \geq 0$. Then we say $t(n)$ is $O(g(n))$ if there exist two positive constants n_0 and c such that:

$$t(n) \leq cg(n) \text{ for } n \geq n_0$$

Problem 3. *Prove that* $5n + 70$ *is* $O(n)$

Solution. The idea here is to come up with something that is larger than $5n + 70$

$$5n + 70 \leq ?$$

Well we can see that $5n + 70n$ is always larger, for $n \geq 1$.

$$5n + 70 \leq 5n + 70n \text{ for n} \geq 1$$

$$\Leftrightarrow 5n + 70 \leq 75n \text{ for n} \geq 1$$

so we can take $c = 75$, $n_0 = 1$ and the definition of Big O is satisfied.

Note that there is nothing special about these particular values, other than they satisfy the inequality and definition. Any value of c and n_0 that works is valid.

2.3 Tips and extra notes on Big O proofs.

- $O(1)$ just means that it takes a constant amount of time.
- Be sure to be clear which statement implies which, and that your proof is 100
- Start by looking for some expression that will make the inequality true. Try to make it so that it only includes the type of term you want (n^2, $nlog_2(n)$, etc).
- Like the recurrences, the only way to get better is to practice.

2.4 Big O properties

Constant rule

If $f(n)$ is $O(g(n))$ then $af(n)$ is $O(g(n))$, for some constant a.

Proof

Take the definition of Big O, and multiply it through by a:

There exists a c such that
$$f(n) \leq cg(n)$$
for all $n \geq n_0$, and so
$$af(n) \leq acg(n)$$
for all $n \geq n_0$.

Now c is ac.

Sum Rule

If $f_1(n)$ is $O(g(n))$ and $f_2(n)$ is $O(g(n)$, then $f_1(n) + f_2(n)$ is $O(g(n))$.

Proof

We just extend the definition of Big O to now have two of each constant, and two functions f.

There exists constants c_1, c_2, n_0, n_1 such that
$$f_1(n) \leq c_1 g(n)$$
for all $n \geq n_0$, and
$$f_2(n) \leq c_2 g(n)$$
for all $n \geq n_1$.

Thus,
$$f_1(n) + f_2(n) \leq c_1 g(n) + c_2 g(n)$$

for all $n \geq max(n_0, n_1)$. So we can take $c_1 + c_2$ and $max(n_0, n_1)$ as our two constants.

Product Rule

If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n)$, then $f_1(n)f_2(n)$ is $O(g_1(n)g_2(n))$.

Proof We can use similar constants as in the sum rule, except that now we have two g functions. So, there exists constants c_1, c_2, n_0, n_1 such that

$$f_1(n) \leq c_1 g_1(n)$$

for all $n \geq n_0$, and

$$f_2(n) \leq c_2 g_2(n)$$

for all $n \geq n_1$. Thus,

$$f_1(n)f_2(n) \leq c_1 g_1(n) c_2 g_2(n)$$

for all $n \geq max(n_0, n_1)$. So we can take $c_1 c_2$ and $max(n_0, n_1)$ as our two constants.

Transitivity Rule If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.

Proof

Similar idea as before:

There exists constants c_1, c_2, n_0, n_1 such that

$$f(n) \leq c_1 g(n)$$

for all $n \geq n_0$, and

$$g(n) \leq c_2 h(n)$$

for all $n \geq n_1$. Plugging $g(n)$ from the second inequality into $g(n)$ in the first inequality gives that

$$f(n) \leq c_1 c_2 h(n)$$

for all $n \geq max(n_0, n_1)$.

- The main idea behind all these proofs is just applying the definition of Big O to several functions at once.
- We've been using these properties unknowingly, and now we can justify saying something is $O()$ by looking at the "largest" term.
- We can now say that if, say, $f(n)$ is $O(n^2)$ that $f(n) \in O(n^2)$
- $O(1) \subset O(log_x(n)) \subset O(n) \subset O(nlog_x(n)) \subset O(n^2)... \subset O(2^n) \subset O(n!)$

2.5 Formal definition of Big Ω (Omega)

In a way, this is the opposite of Big O. Instead of saying $f(n)$ is bounded **above**, we saying $f(n)$ is bounded **below** by $g(n)$

The definition follows the same idea:

Let $t(n)$, $g(n)$ be functions, and $n \geq 0$. Then we say $t(n)$ is $O(g(n))$ if there exists a c and n_0 such that for all $n \geq n_0$:

$$t(n) \geq cg(n)$$

Problem 4. *Prove that* $\frac{n(n-1)}{2}$ *is* $\Omega(n^2)$

Solution. We start in a similar way than the problems for Big O. We write down the definition, and look for a relation that is true, and contains only terms of n^2. We'll try different values of c.

$$\frac{n(n-1)}{2} \geq ?$$

Try $c = \frac{1}{4}$

$$\frac{n(n-1)}{2} \geq \frac{n^2}{4} \text{for sufficiently large n}$$

$$\Leftrightarrow 2n(n-1) \geq n^2$$
$$\Leftrightarrow 2n^2 - 2n \geq n^2$$
$$\Leftrightarrow n^2 \geq 2n$$
$$\Leftrightarrow n \geq 2$$

We can see that this holds for $n \geq 2$, $c = \frac{1}{4}$

2.6 Tips and extra notes on big Ω proofs

- These are the same as Big O proofs, but with a flipped inequality.
- Be sure to specify what implies what.
- Do not assume what you're trying to prove!
- Some creativity, pattern matching and testing is involved, so practice!

2.7 Formal Definition of Big θ

We say that $t(n)$ is $\theta(g(n))$ if $t(n)$ is both $O(g(n))$ and $\Omega(g(n))$ for some $g(n)$.

An equivalent definition is that there exists three positive constants n_0 and c_1 and c_2 such that, for all $n \geq n_0$,

$$c_1 g(n) \leq t(n) \leq c_2 g(n)$$

. Obviously, we would need $c_1 \leq c_2$ for this to be possible.

Its possible for a function to not be Big θ of anything, but these examples are weird and don't show up often in practice.

2.8 Best and Worst Cases

The following is a table of examples of algorithms seen in the course, and their best and worst cases.

List Algorithms	$t_{best}(n)$	$t_{worst}(n)$
add, remove element (array list)	$\Theta(1)$	$\Theta(n)$
add, remove an element (doubly linked list)	$\Theta(1)$	$\Theta(n)$
insertion sort	$\Theta(n)$	$\Theta(n^2)$
selection sort	$\Theta(n^2)$	$\Theta(n^2)$
binary search (sorted array)	$\Theta(\log n)$	$\Theta(\log n)$
mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$
quick sort	$\Theta(n \log n)$	$\Theta(n^2)$

Figure 1: Table by Prof. Michael Langer, 2017

2.9 Limits and Big O

There is a rule for determining whether $f(n)$ is $O(g(n))$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \text{ is } O(g(n))$$

Note that this doesn't go the other way around.

Also note that this is weak, since we might get the result of, say $f(n)$ being $O(g(n^2))$ when it's really $O(g(n))$, which is a stronger statement.

Specifically, if we can say that this means it's *not* $\Omega(g(n))$ then this is a stronger statement.

We have the following rule:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) \text{ is } \Omega(g(n))$$

$$\Rightarrow f(n) \text{ is not } O(g(n))$$

and similarly:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c\,, 0 < c < \infty \Rightarrow f(n) \text{ is } \theta(g(n))$$

Part II
Non-Linear Data Structures

3 Rooted Trees

3.1 Basic Idea

Trees are good for organizing hierarchal structures like directory listings and rankings. Trees are best explained through pictures and learning the key terminology, so see the lecture slides for pictures, I'll list the terminology here.

3.2 Tree Vocabulary

Node: The dots on the tree. Sometimes called the vertex.

Child: Each node (except those at the bottom) have child nodes that branch off of them.

Parent: The node that the child node directly comes from. (The parents parent is not the same parent)

Root: The node at the very top of the tree. It has no parent.(it's the only node without a parent). All other nodes originate from this root.

Siblings: The nodes that share the same parent.

Leaf: Nodes with no children.

Internal Node: Node with a child

Path: A sequence of nodes that are connected by edges (edges being a connection between two nodes)

Length of a path: How many edges are between the first node in the path and the last one.

Depth: The length of the path from the root to the node you're interested in finding the depth of.

Height: As you'd expect, the opposite of depth. The length of the path from the "lowest" leaf.

Ancestor: A node that's on the same path from the root as the node you're interested in finding the ancestor of.

Subtree: Take a node, call it the root of your new sub-tree, everything below this node is part of the sub-tree. Trees are subtrees of themselves.

3.3 Notes and Facts

- If a tree has n nodes, it will have $n - 1$ edges. Since every node has an edge with its parent, except for the root.
- If you haven't already, go look at the slides. It's important to grasp the visuals here so you can picture what's happening.
- The length of a path is the number of nodes in the path minus 1.
- An easy algorithm for finding depth would be: If you don't have a parent, your depth is 0 (you are the root), return 0. If you do have a parent, return 1 + the depth of your parent (recursion).
- In a similar way, we find the height by: If you don't have a child, your depth is 0(you are a leaf), return 0. If you do have a child,(or more) for each of your children, take the maximum of their heights, return 1 + that.
- Non-rooted trees are when there's no clear root. Much more complex, more for COMP 251

4 Tree Traversal

4.1 Depth-First Traversal

As the name implies, this is when you go all the way to the bottom of your list, and work your way to the top. There are two ways to do this, pre-, and post- order traversal. Both are done recursively, and it's really elegant when you see how it works.

Here's the pseudocode:

```
depthfirst(root){
  if (root is not empty){
     visit root;
     for each child{
       depthfirst(child);
     }
  }
}
```

The most confusing part about this is the meaning of root. At first its the actual root, but then its actually the root of the sub-tree that's created by looking at the children.

This one is called a "pre-order" tree traversal, since we're looking at the what's in the nodes before moving on to the next one. (Note that "looking" is a loose term, you can actually do anything you want at this stage, like change values, etc.)

It really helps to look at the numbering of the nodes from the slides, and make sure you could do it yourself if asked.

Now we'll look at "post-order" it's exactly the same except that you visit the nodes AFTER the recursive call.

```
depthfirst(root){
  if (root is not empty){
     for each child{
       depthfirst(child);
     }
     visit root;
  }
}
```

The effect this has, is that you'll plunge to the very bottom of the list before visiting your first node, and the root will be the last one visited.

4.2 Iterative Depth First Traversal

Remember how recursion works using a call-stack? Well, seems like maybe we could use a stack then! Turns out you can!

```
stackTraverse(root){
  stack.push(root);
  while(stack not empty){
   current = stack.pop();
   visit current;
   for each child{
      stack.push(child)
     }
  }
}
```

So you can see its really the same as with recursion, since the call-stack IS a stack.

4.3 Notes on Depth-First

- You can't do a post-order traversal with a stack (try it)
- Using a stack has a slightly different order than with recursion
- Recursion goes left to right, while stack goes right to left.

- This might be something to keep in mind in situations where you want to specifically traverse in a particular order. (Post or pre? Stack or recursion?)

4.4 Breath-First Traversal

This is when you read the tree left to right, level by level (like a book). You can do this by using a queue instead of a stack! Check it out:

```
breadthfirst(root){
  queue.enqueue(root);
  while(queue not empty){
    current = queue.dequeue;
    visit current;
    for each child of current{
      queue.enqueue(child)
    }
  }
}
```

Notice this is the exact same as with the stack, but using a queue instead. This simply changes the order of what's being visited! I really recommend going through step by step with the slides and seeing how this happens.

It's also useful when looking at the order of what get's visited, to write out what the queue looks like at each step.

What about if the visit was after the for loop? Well, same as with the stack, nothing changes, since current is still the only thing dequeued.

5 Binary Trees

5.1 Basic Idea

Basically, as the name implies, a binary tree is a tree where each node has at most two children. It might have 0, 1, but at most 2.

How many nodes in a binary tree? Spell it out!

Level	Max Nodes
0	1
1	2
2	4
3	8
4	16

You can see now that we have the powers of 2 emerging!

$$\sum_{i=0}^{n} 2^i, \text{ where n is the number of levels}$$

This is a geometric series, and if you haven't memorized it already, DO IT. Seriously, it comes up over and over.

$$= \frac{2^{n+1} - 1}{2 - 1}$$
$$= 2^{n+1} - 1$$

This result makes sense, since we should have an odd number of nodes (because of the root).

Note that we also have a lower bound, since the height (if you remember from last section) is the number of nodes in the longest path -1. So the number of nodes in the longest path is the height +1. The least number of nodes we can have in a tree of height n is $n + 1$, since each node must have exactly 1 child, since if it has 2 it's not the minimum, and if it has 0 then the height will be determined by that node.

If you're ever asked to find the height from the number of nodes, just rearrange this inequality:

$$n + 1 \leq Nodes \leq 2^{n+1} - 1$$

to

$$log_2(Nodes + 1) \leq n \leq Nodes - 1$$

5.2 Binary Tree Traversal

Tree traversal in binary trees is exactly the same as with general trees, but it's even simpler, since we know exactly how many children we might have. So we can do pre and post order as before, but with an added, third option.

Here's the old pre-and-post-order:

```
preorderBin(root){
  if root not null{
    visit root;
    preorderBin(leftChild);
    preorderBin(rightChild);
  }
}

postorderBin(root){
  if root not null{
    postorderBin(leftChild);
    postorderBin(rightChild);
    visit root;
  }
}
```

Now, the new, third option, since we know exactly how many children there are, is "in-order" traversal:

```
inOrder(root){
  if root not null{
    inOrder(leftChild);
    visit root;
    inOrder(rightChild);
  }
}
```

5.3 Expression Trees

Again, the slides are very helpful here, since it's hard to visualize these things if you haven't seen them. If you make a binary tree with operators $(/, +, -, *,^)$ at the inner nodes, and values at the leaves, you can define order of operations uniquely by how the tree is set up, and how it's read.

See the slides for examples of such trees, but all you need to know is which order of operations each type of traversal gives (preorder, in-order, post order)

Note that this isn't the only way to do expressions, you could have pre and post-fix expression such as - 5 3 (meaning 5-3)(pre-fix) or 5 3 -(post-fix). The advantage to this, is that we don't need to define order of operations, since it's intrinsic to the structure. Using a stack to implement this:

```
expressionEvaluator(){
  stack = empty stack;
  current = head;
  while (current not null){
    if (current is a value){
      stack.push(current);
    }
    else{
      operand2 = stack.pop;
      operand1 = stack.pop;
      operator = current;
      stack.push (evaluate (operand1 operator operand2));
    }
    current = current.next;
  }
}
```

Notice this is exactly what was done in assignment 2. Notice also that pre-order traversal gives a pre-order expression, post-order traversal gives post-order expression, and in-order traversal gives in-fix expression.

6 Binary Search Trees

6.1 Definition

A binary search tree is one that has elements (called keys) that are comparable (ie can be compared with ¿, ¡ =), has no duplicates, and all keys in the left subtree are less than the node, and all keys in the right subtree are greater than the node. (see slides for examples)

Important property: If you do an in-order traversal of a binary search tree, you'll get the elements IN-ORDER. (Recall that the binary search algorithm requires a sorted list, so in a way, this tree is sorted).

6.2 Common Operations

There are a few common things you might want to do with a Binary Search tree:

- find a key
- find minimum or maximum
- add a key
- remove a key

Pseudocode:

```
find(root,key){
  if(root==null){
    return null;
  }
  else if(root.key == key){
    return root;
  }
  else if(key<root.key){
    return find(root.leftchild, key);
  }
  else{
    return find(root.rightChild, key);
  }
```

```
}
```

Makes sense, since the left half of the tree is less than the node, and right half is greater.

```
findMin(root){
  if (root == null){
    return null;
  }
  else if(root.leftChild ==null){
    return root;
  }
  else{
     return findMin(root.leftChild){
  }
}
```

Basically, keep calling findMin on the left child until you get to a leaf, then you know you're at the minimum. Find max is the same reasoning but on the right child.

```
add(root,key){
  if (root==null){
    root = new BSTnode(key);
  }
  else if(key<root.key){
    root.leftChild = add(root.leftChild, key);
  }
  else if(key>root.key){
    root.rightChild = add(root.rightChild, key);
  }
  return root;
}
```

Analyse this carefully, its a bit subtle. If we're at a leaf, then we can add the new Node, but we want to make sure we went down the tree correctly, as to be in the correct position to add a node, (hence the else if's).We also need to reassign the references, or else the new node won't be referenced by it's parent!

Notice that root.leftChild or root.rightChild point to an entire subtree of stuff, so we can call add on that entire subtree recursively.

Also, notice that it does nothing in the case where the root contains the key, since we can't have duplicates.

```
remove(root,key){
  if(root == null){
   return null;
  }
  else if(key<root.key){
   root.leftChild = remove(root.leftChild, key);
  }
  else if(key>root.key){
   root.rightChild = remove(root.rightChild, key);
  else if(root.leftChild ==null){//can just put the right or left
      subtree if one of the children is empty
   root=root.right;
  else if(root.rightChild == null){
   root=root.leftChild;
  }
  else{
   root.key=findMin(root.rightChild).key; //everything in the right
       subtree is greater than the root, copy that key into the
       root.
   root.rightChild= remove(root.rightChild, rightChild.key) //now
       remove the key that is now in the root (still minimum key in
       the right subtree
  }
  return root;
```

This one's fairly hard to wrap your head around, so really take some time to understand it. I commented the code to help a little, but here's step by step this is what it's doing:

- Base case, if the root is empty, you're done.
- If the thing we want to remove is less than the thing at the root,

recursive call on the left subtree

- Similarly, if the thing we want to remove is greater than the thing at the root, recursive call on the right subtree
- If the left subtree is empty, then you can just put the right subtree into the root, effectively removing the root.
- Similarly if the right subtree is empty, you can put the left subtree into the root, which will remove the root.
- Now we're at the case where the root has two children, so we take the smallest thing in the right subtree(which is larger than the root, and less than everything else in the right subtree) and replace the root's key with it (getting rid of the root).But now you have a duplicate of the smallest thing from the right subtree. So, you recursively remove the smallest thing in the right subtree!

DON'T MEMORIZE THE CODE, but understand the algorithm. Take some time on it.

6.3 Best and Worst Cases

All of these operations have the same best and worst cases, best being O(1) and worst being O(n).

The best case of finding the minimum,maximum, and finding a key, is when it's at the root.Worst case is when you need n steps (imagine a line or zig zag where every node has only one child, and the thing is at the bottom)

The best case of adding is when you can simply add to a leaf, or removing a leaf. The worst case is when you need to add to the root, or remove the root.

7 Priority Queues and Heaps

A priority queue is used when you want to make sure that the highest priority thing gets attention first, instead of just first-come-first-serve. They're best implemented using a heap.

7.1 Heaps

A heap is a binary tree whose elements are comparable, every level is full,and if not(the lowest level) the elements are as far left as possible. Also, each node is less than it's children. So the root is the least, and the leaves are the greatest.

7.2 Operations on Heaps

Similar to queues, there are two main operations we care about: add and remove.

```
add(element){
  current = new node; //creating the new node at a leaf position
  current.element = some element; //setting the element of the new
     node
  while(current != root and current.element <
     current.parent.element){
   swap(current,parent);
   current = current.parent; //making sure the new node is in the
      right place
  }
}
```

Basically the idea is to add your node to the bottom, and swap it up until it's either the root, or it's parent is less than it.

```
removeMin(){
  temp = root.element;
  remove last leaf node and put its element into the root
  current = root;
  while ((current has a left child) and
     (current.element)>current.left.element) or (current has a
     right child and current.element > current.right.element)){
  minChild = child with smaller element;
  swapElement(current, minChild)
  current = minchild;
  }
  return temp; //return the smallest element
}
```

The basic idea here is to take the smallest node, replace it with the root, and then remove the root. But the thing is, you can't just do that. You need to first take the leaf off to the side (this is the largest element), put it's element into the root, swap the root down to where it should now be.

Note that the while loop has a complicated condition, but look at it carefully. It considers all of the many cases that can occur.

This is Dr.Langers way of doing it, but I propose a more intuitive method (it's almost the same)

My way is to swap the root node all the way down until it's a leaf, and then simply remove it by cutting ties to the tree.

The way in the slides is "remove first, swap later", mine is "swap first remove later"

7.3 Heaps as Arrays

You could implement a heap as an array if we number the nodes in the tree as indexes of an array (breadthwise). See the lecture slides for how this would look. Note that the array starts indexing at 1. (0 is not used at all)

The relationship between the indexes of a child and it's parent is very convenient.

$$parent = child/2$$

$$leftChild = 2 * parent$$

$$rightChild = 2 * parent + 1$$

If this is confusing, see the lecture slides for numbering, and you'll notice that there's a pattern in the numbering of the tree that we're exploiting.

Note that you could have any binary tree represented as an array, the problem is though, that you'll get lots of gaps in the array if the tree is not complete like in heaps.

Lets look at the add method with arrays:

```
add(element){
   size = size +1; //size is incremented since we're adding
      something
   heap[size] = element; //add to the back of the array
   i = size;

   while(i>1 and heap[i] < heap[i/2]){ //while not at front of
      array (root) 1, and the element is less than its parent,
      swap with parent
     swapElements(i,i/2);
     i=i/2; //update index of element to be the index of the parent
   }
}
```

It truly is the same concept as before.

7.4 Building a Heap with Arrays

What's the time complexity of building a heap from scratch with an array?

Well, the best case is that we can add n elements to the array without needing to do any swaps. So, $\theta(n)$

The worst case is a little trickier. Say we want to add element i. Element i will be somewhere between $2^{level} and 2^{level+1}$ since each level starts with a power of 2.

If we take the log_2 of that, we get this inequality

$$level \leq log_2(i) < level + 1$$

So the level is the floor function of $log_2(i)$ (since i cannot equal level +1, and level can't be a decimal)

Note that the level is the number of swaps from the top, so the worst case is when every time we add an element, we need to do "level" number of swaps.

$$t(n) = \sum_{i=1}^{n} floor(log_2(i))$$

since there are n things to be added.

In the slides, it shows a graph of $log_2(n)$, and shows that the area under the curve is roughly $nlog_2(n)$. And it also shows a diagonal line (which is always below our function), which is $\frac{1}{2}nlog_2(n)$.

So we have:

$$\frac{1}{2}nlog_2(n) < t(n) < nlog_2(n)$$

So worse case is θ(n)

7.5 Removing an Element With Arrays

Here's the code, the idea is exactly the same:

```
removeMin(){
   tmpElement = heap[1]; //holding the root value
   heap[1]=heap[size]; //swap elements of root and last thing
   heap[size]=null; //clear the value of last thing
   size=size-1; //update size
   downHeap(1,size);
   return tmpElement;
}

downHeap(startIndex, maxIndex){
   i=startIndex;
   while(2*i<=maxIndex){ //while child of i is less than size
      child=2*i;
      if (child<size){
         if(heap[child+1] < heap[child]){ //if the right child is
             less than the left
```

```
            child = child +1;
        }

        if(heap[child] < heap[i]){ //if a swap is needed, swap.
            swapElements(i, child)
            i = child;
        }
      else break;
   }
}
```

The idea here is exactly the same as before, but with arrays.

7.6 A Faster Way to Build a Heap

Recall that building a heap using our previous method is worst case $\theta n log_2(n)$.

Look back at the structure of a binary tree (complete) notice that there's always half (or half +1) of the nodes that are leaves! And the index of the last leaf (the one at index size) has a parent who's index is size/2 .

So we can use this to have a faster way of building a heap:

```
buildHeapFast(){

   for(size/2; k>=1; k--){ //starting at the parent of the largest
       node, downheap to the end!
      downHeap(k,size);
   }
}
```

It's really just a way of turning a plain old binary tree (complete) and organizing it into a heap.

How is it so fast? The intuition is that, half our nodes don't need to be downheaped, then, quarter might need to be swapped once, then one eighth might need to be swapped twice, etc. Compared to our other method, where we had half the elements needing to go all the way to the top. This is way faster!!

The actual derivation is quite complicated, and we don't need to be able to reproduce it for COMP250. However, we should know that:

$$t(n) = \sum_{i=1}^{n} \text{height of node i}$$
$$= \sum_{level=0}^{height} (height - level)2^{level}$$

This makes sense, since any given level has 2^{level} elements,and there are at most height number of levels. The height of a node at a given level is height-level.

7.7 Heapsort

Steps: Build a heap, then repeatedly call removeMin() and put the removed elements into a list.

```
heapsort(){
   buildheap()
   for(i=1 to size){
      swapElements(heap[1],heap[size+1-i])
      downHeap(1,size-i);
   }
   return reverse(heap);
}
```

This works by swapping the first and last values, downheap to fix it, but now your list is backwards, so reverse and return the reversed list.

If this is confusing look at the lecture slides for a pictorial example.

Note that you could skip the last reversing step by using a max heap instead of a min heap.