Pointer Review (0:00-16:00)

- Let's come up with a program to illustrate the fundamentals
- We'll keep it simple: just a main method and a declaration/initialization `int x = 6;`
- Recall that when we run this program, all of main() will get a chunk of memory at the bottom of the stack. All of main()'s local variables (x, y) and parameters (argc, argv) will go in here.
- The int is a 32-bit chunk of memory containing the value 6
- If we declare a pointer `int * p`, we get another 32-bit chunk of memory in main() which initially contains who-knows-what
- Then when we say `p = &x`, the address of x gets put in p, which we can represent with an arrow from p to x
- So if, say, x was in byte 234 and p was in byte 238 (they are next to each other, each occupying 4 bytes), then we would put the number 234 in p
- Note that we can set p to point to x by saying

```
int * p = x;
p = &x;
```

- Or, equivalently,

```
int * p = &x;
```

- At the end of our main method, we print out the values of p and x
- We can print out the address of x (in hex) by saying

```
printf("%x", &x);
```

- Or equivalently,

```
printf("%x", p);
```

- And we can print out the value of x (again using hex) by saying

```
printf("%x", x);
```

- Or, equivalently,

```
printf("%x", *p);
```

- Upon compiling and running, we find that x is 6 and p is bfd940c (it's in hex)
- Note that we could have used %d in the format strings, but we (somewhat arbitrarily) chose to print in the hexadecimal base and so used %x
- Let's make this a bit more complicated
- We now declare `int **pp` and set `pp = &p`. pp is a pointer to the pointer p.

- At the end of our program we list the following print statements:

```
printf("%x\n", x);
printf("%x\n", (unsigned int) &x);
printf("%x\n", (unsigned int) p);
printf("%x\n", *p);
printf("%x\n", (unsigned int) pp);
printf("%x\n", (unsigned int) *pp);
printf("%x\n", (unsigned int) *(*pp));
```

- We cast pointers to unsigned ints to avoid complaints from the compiler about printing pointers as ints
- Here's what we get:

```
6
bfc2650c
bfc2650c
6
bfc26508
bfc2650c
6
```

- OK, let's break this down.
- x and *p are the same because p is pointing to x and *p "dereferences" p. That is, it follows the address in p and fetches the value at that address.
- &x and p are the same because &x is the address of x and p points to x (contains its address)
- pp is different than p because pp contains the address of p
- *pp is the same as p (and &x) because pp contains the address of p. *pp follows that address to p and prints out the value of p (which happens to be the address of x)
- *(*pp)) is the same as *p (and x) because *(*pp) dereferences *pp, and *pp was the address of x. That is, *(*pp) "undoes" all of the work we just did by following the address in pp to get to p and then following the address there to get to x…right back where we started.

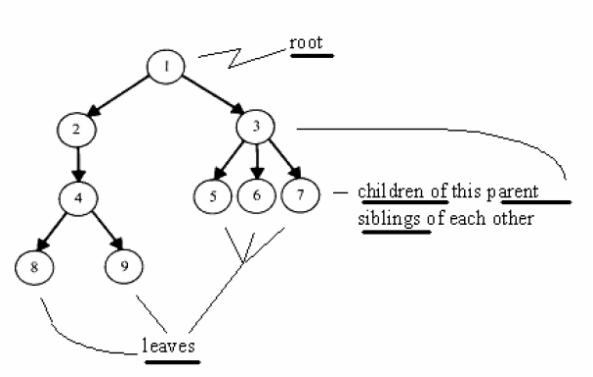
Pointers to Pointers (16:00-19:30)

- Wait, what? You can have a pointer to a pointer?
- Of course. A pointer itself is a little chunk of memory (32 bits to be exact), so you can simply put its address in another pointer using the usual & operator.
- In fact, you guys have been using pointers to pointers for quite a while now.
- Remember argv? This is the second argument of main, an array of all the command-line arguments that were passed in to the program.
- But these arguments are stored as strings, which are under actually char *'s. So argv is an array of pointers to characters.

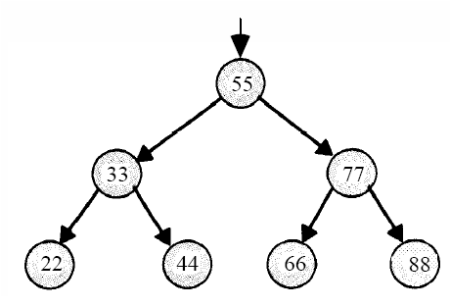
- The name of an array, however, is just a pointer to the first element of the array. So argv is actually a pointer to a pointer.
- We can therefore write it as `char ** argv` or `char * argv[]`
- In fact, any array of strings is just an array of char *'s, so the name of the array is a pointer to a pointer.

Defining the Binary Search Tree (19:30-25:30)

- Trees are a data structure that consist of a root with zero or more children, each of which has zero or more children:



- Trees are defined by their root, in the same way that a linked list is defined by a pointer to its first element
- For our purposes, we will be dealing with binary search trees, in which each node has no more than 2 children:



- To represent each node of the tree, we will need a new data type.
- What should each node contain? Two children nodes and some value.
- We define the following struct to be a new data type using typedef:

```
typedef struct _node{
    int n;
    _node * left;
    _node * right;
} node_t;
```

- Notice that we don't actually store the two children nodes inside the node because having nodes within nodes within nodes would make for a very complicated and large data structure
- Instead, each node simply contains *pointers* to its children, which keeps the size of any node standardized and wieldy.
- Notice also that when we define data types which have members of the same data type (they are "self-referential") we have a second name for the datatype, _node, that is used within the curly braces and never again.
- This type definition is awfully familiar, isn't it? It's a lot like the linked list node definition, but instead of having one "next" pointer it has "left" and "right" pointers
- What if a node has only one child or no children at all? Just set the right pointer to NULL if it has one child and set both pointers to NULL if it has no children.
- How do we declare a new tree?

```
node * root;
```

- This is the same thing we did with the linked list. In order to keep track of our tree, all we need to do is remember a pointer to its root, and from that we can find all the other nodes.

Searching the Binary Search Tree (25:30-39:30)

- We're going to define a function to find a value in a binary search tree.
- The nice thing about a binary search tree is that its nodes are "sorted"
- What does it mean to be sorted in a tree? It means that for any node, the node's value is greater than the value of its left child and less than the value of its right child. (Look back at the diagram above to see how this is true for the sample tree shown.)
- So how do we implement find(int n)?
- Well, one thing's for sure: we'll start at the root, just as we started with the first element of a linked list.
- So we look at the value of our root node. Then what? If n is less than that value, go down the left branch. If n is greater than that value, go down the right branch. And repeat.
- This is shaping up to be a pretty efficient algorithm because, each time we follow a left or right pointer, we eliminate half of the remaining tree from what we need to search
- What does this sound like? Binary search! (surprise, surprise)
- In fact, finding a value in a binary search tree will almost exactly mirror the binary search algorithm, and will have the same runtime and recursive framework.
- We set up the find function as follows because we know that it is going to be recursive:

```
find(int n) {
   return recurse (root,  n);
}
```

- Now, to implement the recusive function, here called recurse(). The job of recurse will be to go left or right and repeat.
- The function recurse() will also return a bool and will take a node * and an int as arguments
- We have two base cases:
  - We've got a NULL pointer (we got to a childless node without every finding a vaue)
  - The value at this node is the number we're looking for
- We've go two recursive cases:
  - The value we're looking for is bigger than the value of the current node, in which case we go right
  - The value we're looking for is smaller than the value of the current node, in which case we go left
- Here's the implementation of recurse():

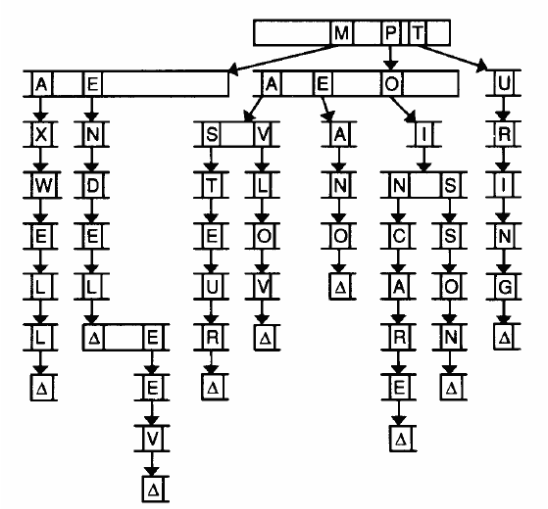
```
bool recurse(node * ptr, int n) {
   if (ptr == NULL)
      return FALSE;
   else if (n == ptr -> n)
      return TRUE;
   else if (n < ptr->n)
      return recurse(ptr->left, n);
   else if (n > ptr->n)
      return recurse(ptr->right, n);
}
```

- Notice that we not only call recurse() within recurse(), but we return whatever we get from it. Each time we recurse, we are "passing the buck."
- We are basically saying that we don't have an answer to the question that was asked, but the answer to that question is the same as the answer to this other, simpler question.
- So we ask the simpler question by calling recurse() again with easier to handle arguments.
- Eventually we get down to a base case, and that call to recurse() will return a TRUE or FALSE. This value will get passed back through the chain of function calls and up to the first call, which came from find().
- This function should help you to see how recursion and pointers can actually be pretty useful.

Tries (39:30-41:30)

- As you remember from last time, the purpose of tries is to look up strings more efficiently than hash tables
- Recall that the runtime of a hash operation and lookup is O(n). In practice, it's more like O(n/m) if we have a good, randomly distributed hash function. But if we're talking about complexity, this is still considered linear in the number of strings
- Tries, on the other hand, provide a runtime that is linear in the length of the string, which, if we have hundreds, thousands, hundreds of thousands of strings, is a huge improvement!

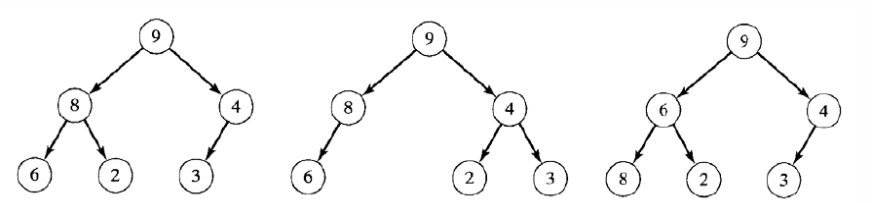
- How did we do this? We have a tree, each of whose nodes is an array of 26 (one slot for each letter of the alphabet).
- Say we want to store the word DOG. We have an array of size 26 and in position D put a pointer to another array of size 26. In position O of that array we put a pointer to another array of 26. In position G of that array we put some value that indicates the end of the word.
- So if we want to look up the word DOG later on, we just use three steps to jump between the arrays. Even if we have thousands of words, the lookup time is just linear in the length of the string.
- But what is the price we pay? Memory! This is pretty clear from the following diagram:



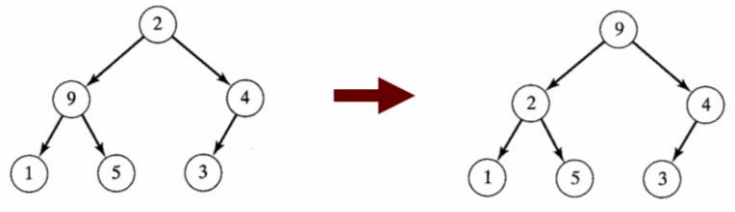
- Each of those little boxes is a 26-member array.
- The data structure becomes very "wide" in memory, even though a lot of the space it is occupying probably ends up being empty.

Heaps (41:30-52:00)

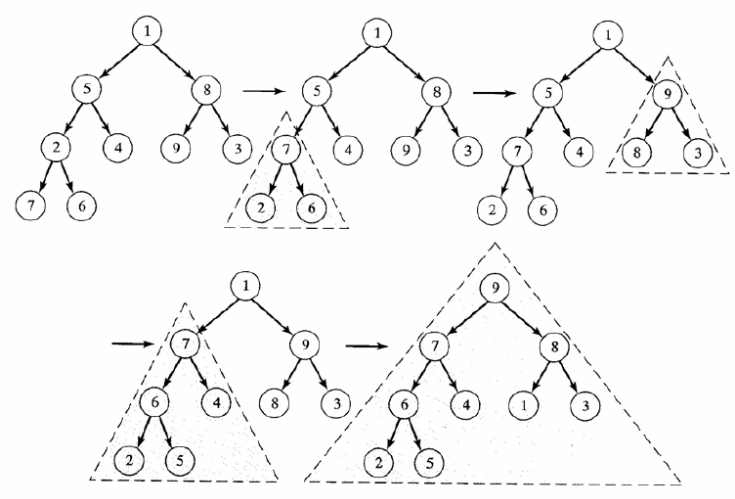
- The last data structure we will look at is the heap.
- A heap is a binary search tree that
  - Is complete. Every level of the tree is completely filled with nodes except for, perhaps, the bottommost level, whose nodes are in the leftmost positions
  - Satisfies the heap order property. Each node's value is greater than or equal to that of its children, if any.
- Which of the following would be considered heaps?



- Only the first one. The second one is not complete because its nodes are not as far left as possible. We have a hole in the left branch. The third is not a heap because six is not greater than eight.
- To heapify an almost heap means to take something that is almost a heap and turn it into a heap:



- The tree on the left is not a heap because nine is greater than two. We fix the problem first by swapping these nodes.
- Have we got a heap yet? No, because 5 > 2. But a series of swaps like the first one would eventually make this tree into a heap.
- Consider the following example:



- Initially, it's not a heap. It's got all sorts of problems, not the least of which is that 1 is the root node and the smallest element.
- To heapify, we start by considering the bottommost, rightmost subtree. We're looking at 2-7-6. We heapify this subtree by swapping 7 and 2.

- Then we look at the next bottommost, rightmost subtree. We're looking at 8-9-3. We heapify this subtree by swapping 9 and 8.
- Then we look at the next bottommost, rightmost subtree which will now consist of five nodes. First we consider 7-5-4 and swap 7 with 5. Then we have to swap 6 and 5. Now this subtree is set.
- The last subtree we look at is the whole tree. We swap 9 and 1, then 1 and 8.
- At the end, we have a binary tree which is still complete (because we didn't add any arrows, just swapped values), but which now satisfies the heap-order property.
- Home exercise: implement this algorithm in C!
- OK, so who cares?
- Well, how many steps does heapifying take?
- First let's think about how many subtrees there are. Well, if we start with subtrees of size 1, then of size 3, and so on, then each node in the tree is a potential root of one and only one subtree. So there are n subtrees.
- Now let's figure out how long it takes to heapify each subtree. The heapification process consists of moving elements upward, so the number of steps to heapify a subtree can only be, at most, log k, where k is the height of the subtree.
- How tall is each subtree? The height of an n-node tree is $\log_2 n$, so we'll be generous and say each subtree has height at most log n, too. This means that each subtree heapification is at most log n steps.
- So we have n heapifications of log n steps each. The whole process, then, is just O(n log n)!
- Why do we care? This motivates a new kind of search called heapsort.
- Throw n elements into one of these structures and heapify it. You can then pluck out each element in sorted order in just n log n more steps.
- This gives a total running time for so-callead heapsort of O(n log n) which is the same as merge sort but with less memory cost!