Recursion, pt. 2:

Thinking it Through

Sorting

- One classic application for recursion is for use in sorting.
  - What might be some strategies we could use for recursively sorting?
  - During the course introduction, a rough overview of quicksort was mentioned.
  - There are other divide-and-conquer type strategies.

Merge Sort

- One technique, called merge sort, operates on the idea of merging two pre-sorted arrays.
  - How could such a technique help us?

Merge Sort

- One technique, called merge sort, operates on the idea of merging two pre-sorted arrays.
  - If we have two presorted arrays, then the smallest overall element *must* be in the first slot of one of the two arrays.
  - It dramatically reduces the effort needed to find each element in its proper order.

Merge Sort

- Problem: when presented with a fresh array, with items in random order within it… how can we use this idea of “merging” to sort the array?
  - Hint: think with recursion!
  - Base case: n = 1 – a single item is automatically a “sorted” list.

Merge Sort

- We've found our base case, but what would be our recursive step?
  - Given the following two sorted arrays, how would we *merge* them into a single sorted array?

```
[13]
=> [13 42]
[42]
```

Merge Sort

- Given the following two sorted arrays, how would we *merge* them into a single sorted array?

```
[-2, 47]
           => [-2, 47, 57, 101]
[57, 101]
```

Merge Sort

- Given the following two sorted arrays, how would we *merge* them into a single sorted array?

```
[13, 42]
            [ 7, 13, 42, 101]
     =>
[7, 101]
```

Merge Sort

- Can we find a pattern that we can use to make a complete recursive step?

Merge Sort

- Given the following two sorted arrays, how would we *merge* them into a single sorted array?

```
[13, 42]
                  [ 7, 13, 42, 101]
        =>
[ 7, 101]
```

Merge Sort

- In words:
  - If both lists still have remaining elements, pick the smaller of the first elements and consider that element removed.
  - If only one list has remaining elements, copy the remaining elements into place.
- This is the recursive step of merge sort.

Merge Sort

13 32 77 55 43 1 42 88

[13 32 77 55],[43 1 42 88]

[13 32],[77 55],[43 1],[42 88]

| | |

[13 32],[55 77],[1 43],[42 88]

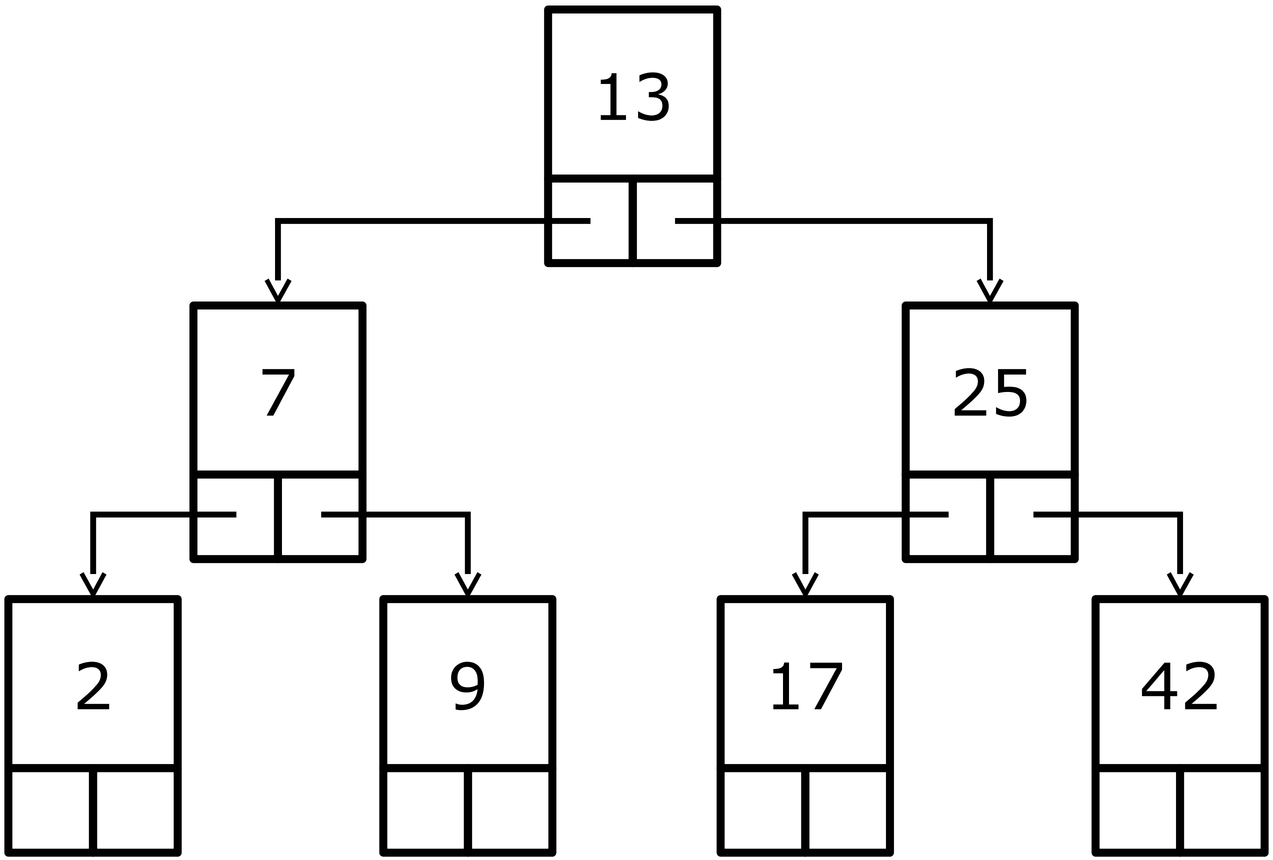
[13 32 55 77],[1 42 43 88]

[1 13 32 42 43 55 77 88]

Merge Sort

- Note that for each element insertion into the new array, only one element needs to be examined from each of the two old arrays.
  - It's possible because the two arrays are presorted.
  - The "merge" operation thus takes O(N) time.

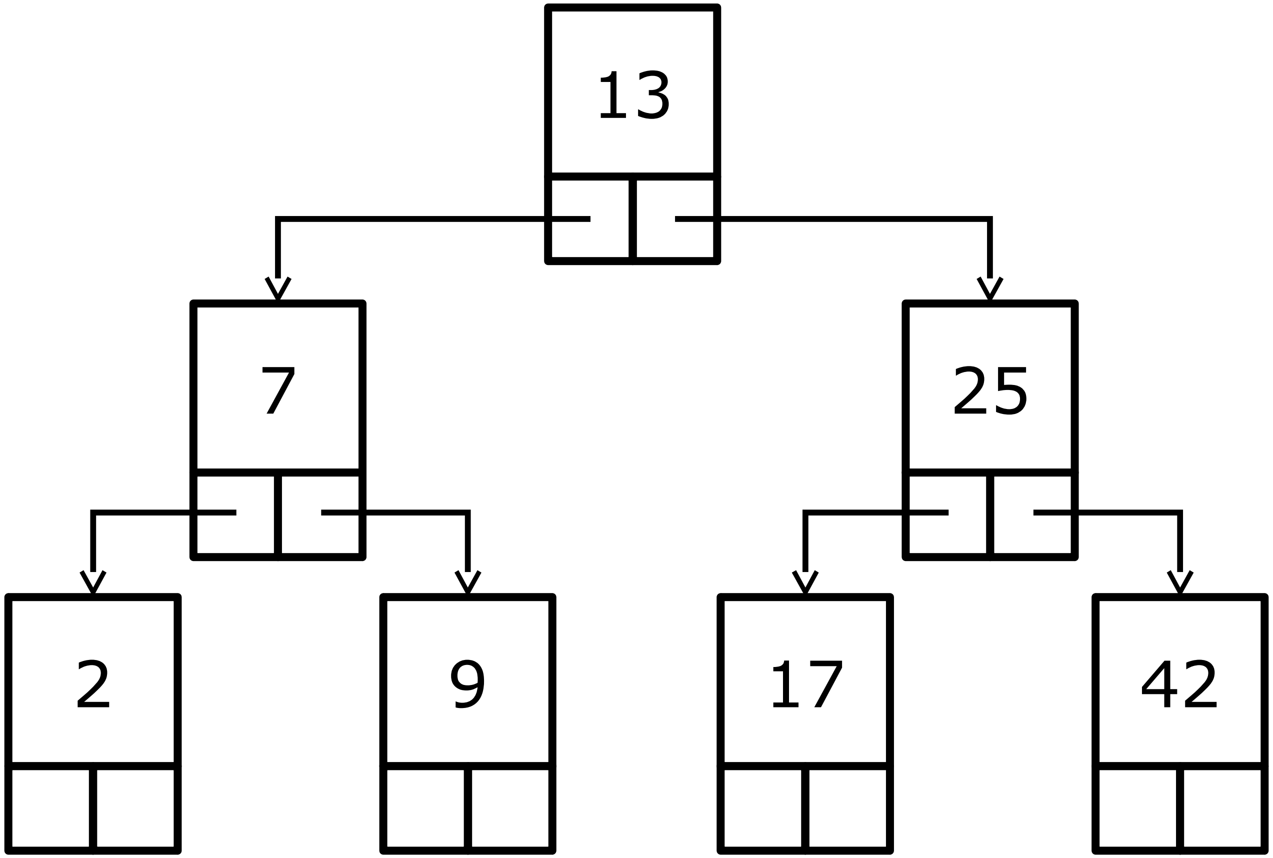
Recursive Structure:
Binary Tree



Recursive Structures

- One data structure we've have yet to examine is that of the binary tree.
- How could we create a method that prints out the values within a binary tree in sorted order?

Binary Tree



Binary Tree Code

```
template <typename K, typename V>
class TreeNode<K, V>
{
  public:
  K key;
  V value;
  TreeNode<K, V>* left;
  TreeNode<K, V>* right;
}
```

Binary Tree

- What can we note about binary trees that can help us print them in sorted order?

Binary Tree

- Note that for a given node, anything in the left subtree comes (in sorted order) before the node.
- On the other hand, anything in the right subtree comes after the node.

Binary Tree Recursion

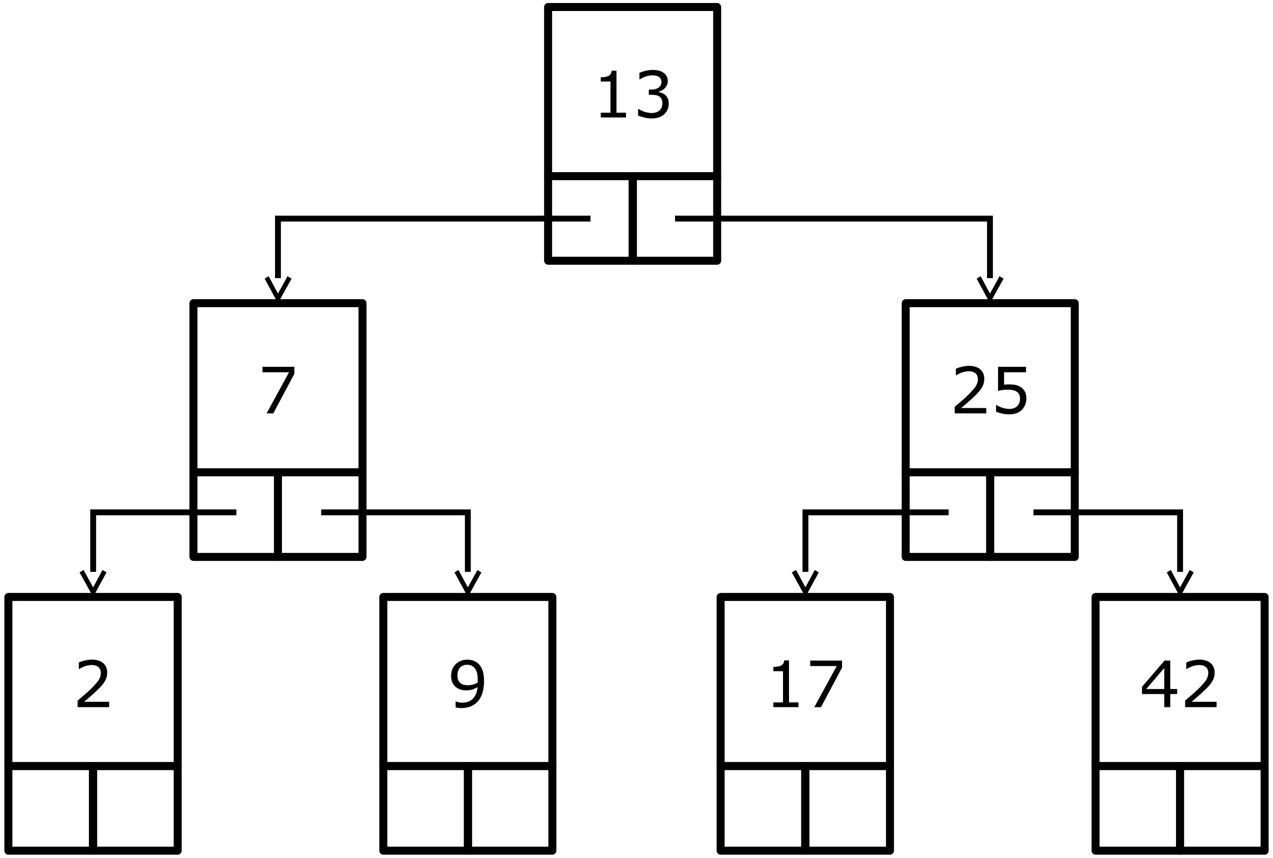
- So, to print out the binary tree…

```
void print(TreeNode<K, V>* node)
{
   if(node == 0) return;
   print(node.left);
   cout << node.value << “ ”;
   print(node.right);
}
```

Binary Tree Recursion

- Calling print() with the tree's root node will then print out the entire tree.
  - Note that we're dumping the values to the console because it's simpler for teaching purposes.
  - We *should* instead set up either an iterator which returns each item in the tree, one at a time, in proper order… or a custom `operator<<`.

Binary Tree



Analysis

1. We started at n and reduced the problem to 1!.
   - Is there a reason we couldn't start from 1 and move up to n?
2. The actual computation was done entirely at the end of the method, after it returned from recursion.
   - Could we do some of this calculation on the way, before the return?

Using Recursion

- Note that the core reduction of the problem is still the same, no matter how we handle the issues raised by #1 and #2.
- How we choose to *code* this reduction, however, can vary greatly, and can even make a difference in efficiency.

Using Recursion

- Let’s examine the issue raised by #1: that of starting from the reduced form and moving to the actual answer we want.

Coding Recursion

```
int factorial(int n)
{
  if(n<0) throw Exception();
  if(???)
  return ?;
  else return n * factorial(n+1);
}
```

- Hmm. We're missing something.

Coding Recursion

- How will we know when we reach the desired value of n?
  - Also, isn't this method modifying the actual value of n? Maybe we need… another parameter.

Coding Recursion

```
int factorial(int i, int n)
{
  if(i<0) throw exception();
  if(???)
    return ?;
  else return i * factorial(i+1, n);
}
```

- That looks better. When do we stop?

Coding Recursion

```
int factorial(int i, int n)
{
   if(i<0) throw exception();
   if(i >= n)
   return n;
   else return i * factorial(i+1, n);
}
```

- Well, almost. It might need cleaning up.

Helper Methods

- Unfortunately, writing the methods in this way does leave a certain design flaw in place.
  - We're *expecting* the caller of these methods to know the correct initial values to place in the parameters.
  - We've left part of our overall method implementation exposed. This could cause issues.

Helper Methods

- A better solution would be to write a *helper method* solution.
  - It’s so named because its sole reason to exist is to *help* setup the needed parameters and such for the true, underlying method.

Coding Recursion

```
int factorial(int i, int n)
{
  if(i >= n)
    return n;
  else return i * factorial(i+1, n);
}
```

Helper Methods

```
int factorialStarter(int n)
{
    if(n < 0) throw exception();
    else if(n==0) return 1;
    else return factorial(1, n);
}
```

Helper Methods

- Note how `factorialStarter` performs the error-checking and sets up the special recursive parameter.
  - This would be the method that should be called for true factorial functionality.
  - The original `factorial` method would then be its *helper method*, aiding the originally-called method perform its tasks.

Helper Methods

- Note that we wish for only `factorialStarter` to be generally accessible – to be `public`.
  - Assuming, of course, that these methods are class methods.
  - The basic `factorial` method is somewhat exposed.
  - The solution? Make `factorial` `private`!

Using Recursion

- Let’s now turn our attention to the issue raised by #2: that of when the main efforts of computation occur.
  - For this version, we’ll return to starting at “n” and counting down to 1.

Using Recursion

- How can we perform most of the computation before reaching the base case of our problem?
  - 5! = 5 * 4!
    = 5 * 4 * 3!
    = …
    = 5 * 4 * 3 * 2 * 1!

Using Recursion

- How can we perform most of the computation before reaching the base case of our problem?
  - 5! = **5** * 4!
    = **5 * 4** * 3!
    = …
    = **5 * 4 * 3 * 2** * 1!
  - We could keep track of this multiplier across our recursive calls.

Coding Recursion

```
int factorial(int part, int n)
{
  if(n == 0 || n == 1)
  return part;
  else
  return factorial
      (part * n, n-1);
}
```

Coding Recursion

```
int factorial(int n)
{
   return factorial(1, n);
}
```

- Using a separate method to start our computation allows us to hide the additional internal parameter.

Tail Recursion

- A *tail-recursive* method is one in which all of the computation is done during the initial method calls.
  - When a method is tail-recursive, the final, full desired answer may be obtained once the base case is reached.
  - In such conditions, the answer is merely forward back through the chain of remaining “return” statements.

Tail Recursion

```
int factorial(int part, int n)
{
   if(n == 0 || n == 1)
   return part;
   else
   return factorial(part * n, n-1);
}
```

- Note how in the recursive step, the “reduced” problem’s return value is instantly returned.

Tail Recursion

- For methods which are tail-recursive, compilers can shortcut the entire return statement chain and instead return directly from the original, first call of the method.
  - In essence, a well-written compiler can reduce tail-recursive methods to mere iteration.