Lecture 3

Dictionaries, Binary Trees, Recurrences

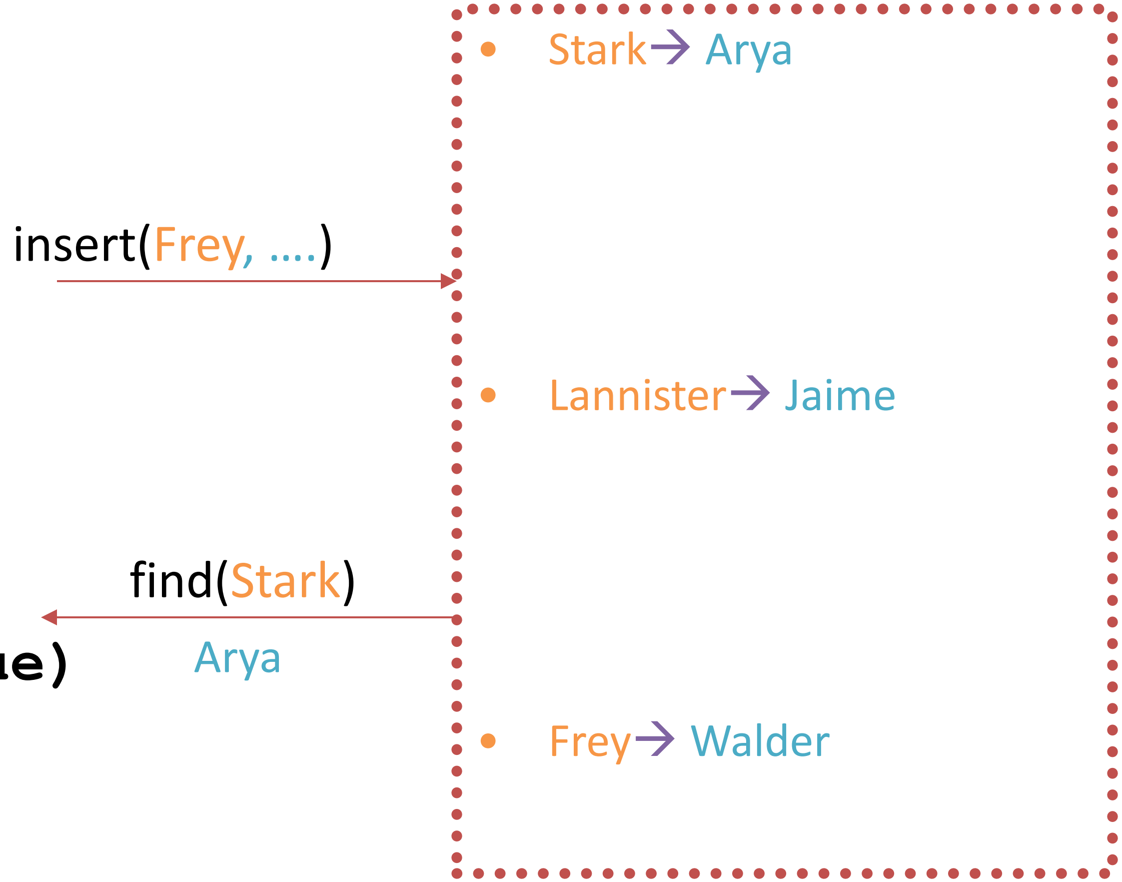
This lecture

More ADTs and Data Structures:

- Dictionaries/Maps, Sets
- Binary Trees

The Dictionary (a.k.a. Map) ADT

- Data:
  - *set* of (key, value) *pairs*
  - keys must be *comparable*
- Operations:
  - **`insert(key,value)`**
  - **`find(key)`**
  - **`delete(key)`**
  - …



Comparison: The Set ADT

The *Set* ADT is like a Dictionary without any values

- A key is *present* or not (no duplicates)

For **`find`**, **`insert`**, **`delete`**, there is little difference

- In dictionary, values are "just along for the ride"
- So *same data-structure ideas* work for dictionaries and sets

But if your Set ADT has other important operations this may not hold

- **`union`**, **`intersection`**, **`is_subset`**
- Notice these are binary operators on sets

Applications

Any time you want to store information according to some key and be able to retrieve it efficiently. Lots of programs do that!

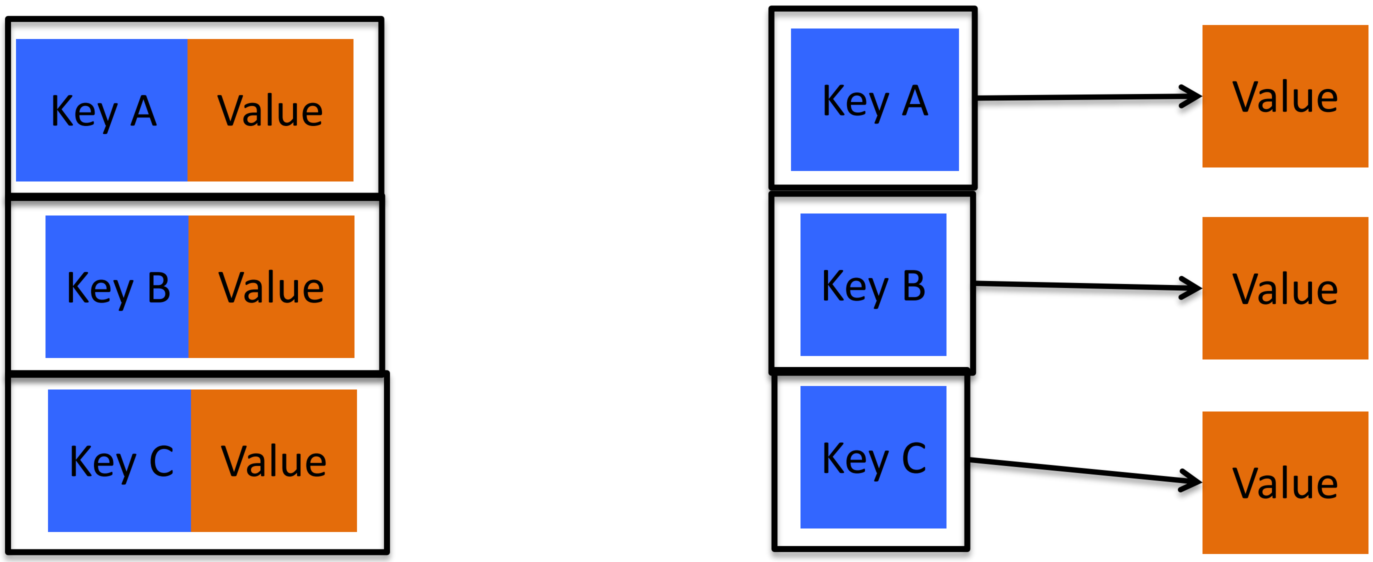
- Lots of fast look-up uses in search: inverted indexes, storing a phone directory, etc
- Routing information through a Network
- Operating systems looking up information in page tables
- Compilers looking up information in symbol tables
- Databases storing data in fast searchable indexes
- Biology genome maps

Dictionary Implementation Intuition

We store the keys with their values so all we really care about is how the keys are stored.

– want fast operations for iterating over the keys

You could think about this in a couple ways:



Simple implementations

For dictionary with *n* key/value pairs

	`insert`	`find`	`delete`
Unsorted linked-list	$O(1)$*	$O(n)$	$O(n)$
Unsorted array	$O(1)$*	$O(n)$	$O(n)$
Sorted linked list	$O(n)$	$O(n)$	$O(n)$
Sorted array	$O(n)$	$O(\log n)$	$O(n)$

* Unless we need to check for duplicates

Implementations

There are many good data structures for (large) dictionaries

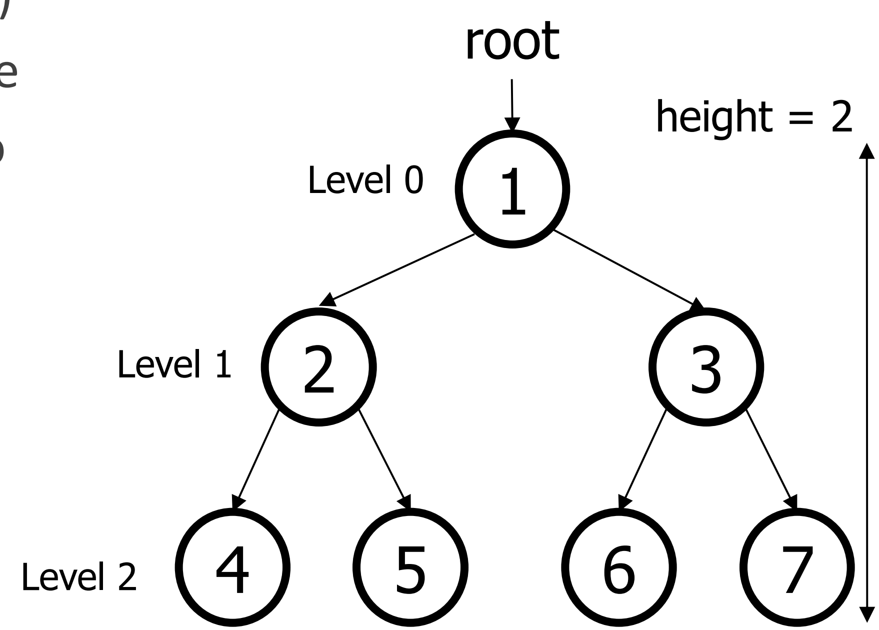
1. Binary Search Trees (next week)
   - Simple and efficient on average
2. AVL trees (next week)
   - Binary search trees with *guaranteed balance*
   - Involves reshaping of the tree on the fly

Other, really cool, data structures (e.g., red-black, splay tress, hastable)

Tree Terminology

- **node**: an object containing a data value and left/right children
  - **root**: topmost node of a tree
  - **leaf**: a node that has no children
  - **branch**: any internal node (non-root)
  - **parent**: a node that refers to this one
  - **child**: a node that this node refers to
  - **sibling**: a node with a common

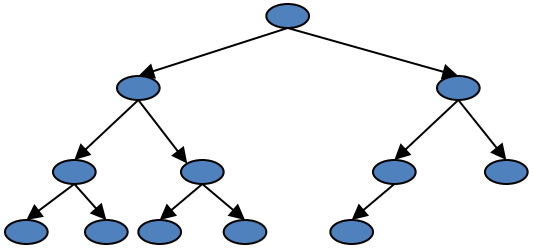
- **subtree**: the smaller tree of nodes on the left or right of the current node
- **height**: length of the longest path from the root to any node (count edges)
- **level** or **depth**: length of the path from a root to a given node

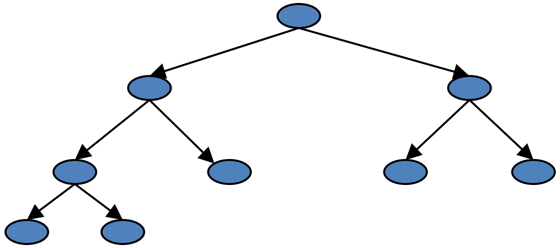


kinds of trees

Certain terms define trees with specific structure

- Binary tree: Each node has at most 2 children (branching factor 2)
- *n*-ary tree: Each node has at most *n* children (branching factor *n*)
- Perfect tree: Each row completely full
- Full tree: Each node has 0 or 2 children
- Complete tree: Each row completely full except maybe the bottom row, which is filled from left to right (binary heap)

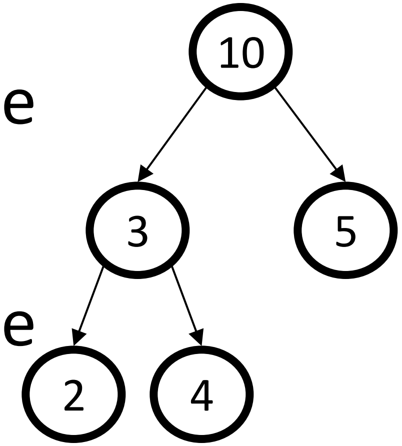




Tree Traversals

A *traversal* is an order for visiting all the nodes of a tree

- *Pre-order*: root, left subtree, right subtree
- *In-order*: left subtree, root, right subtree
- *Post-order*: left subtree, right subtree, root



Tree Traversals

A *traversal* is an order for visiting all the nodes of a tree

- ***Pre-order***: root, left subtree, right subtree

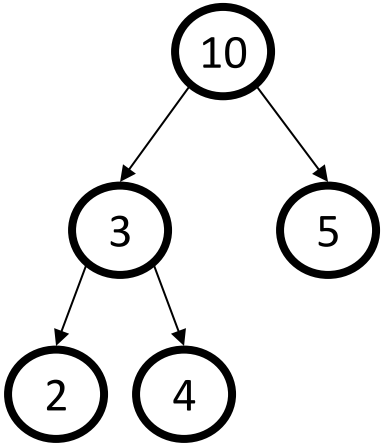
  10 3 2 4 5

- ***In-order***: left subtree, root, right subtree

  2 3 4 10 5

- ***Post-order***: left subtree, right subtree, root

  2 4 3 5 10



Tree Traversals

A *traversal* is an order for visiting all the nodes of a tree

- *Pre-order*: root, left subtree, right subtree

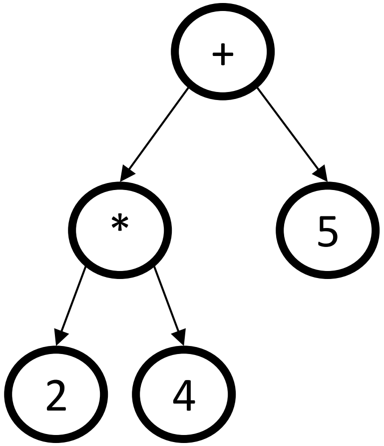
  \+ * 2 4 5

- *In-order*: left subtree, root, right subtree

  2 * 4 + 5

- *Post-order*: left subtree, right subtree, root

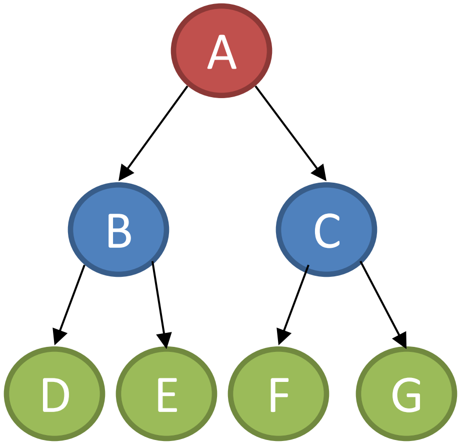
  2 4 * 5 +



(an expression tree)

More on traversals

```
void inOrderTraversal(Node t){
  if(t != null) {
    inOrderTraversal(t.left);
    process(t.element);
    inOrderTraversal(t.right);
  }
}
```



Sometimes order doesn't matter

- Example: sum all elements

Sometimes order matters

- Example: print tree with parent above indented children (pre-order)
- Example: evaluate an expression tree (post-order)

```
A
  B
    D
    E
  C
    F
    G
```

Computable data for Binary Trees

Recall: height of a tree = longest path from root to leaf (count edges)

For binary tree of height *h*:

- max # of leaves: 2^h
- max # of nodes: $2^{(h+1)} - 1$
- min # of leaves: 1
- min # of nodes: $h + 1$

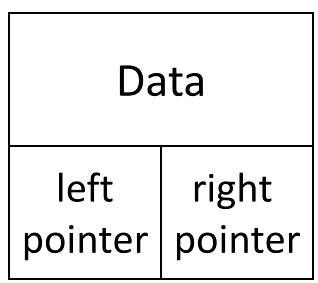
For n nodes:

- *best case is O(log n) height*
- *worst case is O(n) height*

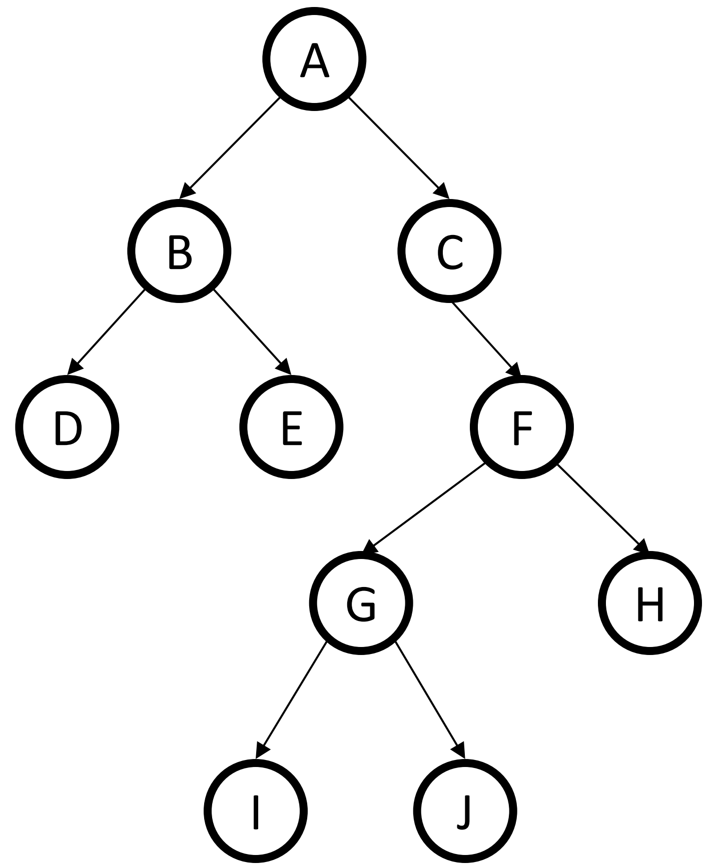
Binary Trees

- Binary tree: Each node has at most 2 children (branching factor 2)

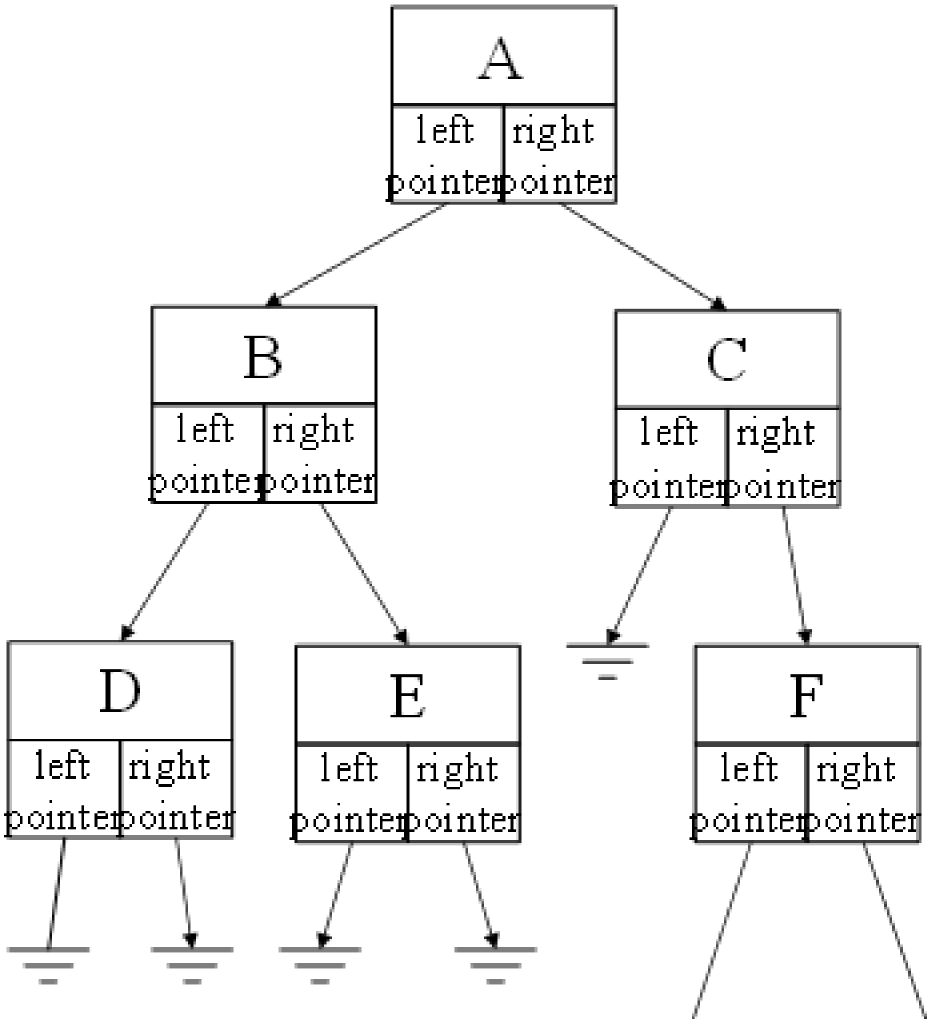
- Binary tree is
  - A root *(with data)*
  - A left subtree that's a binary tree
  - A right subtree that's a binary tree
- *These subtrees may be empty.*
- Representation:

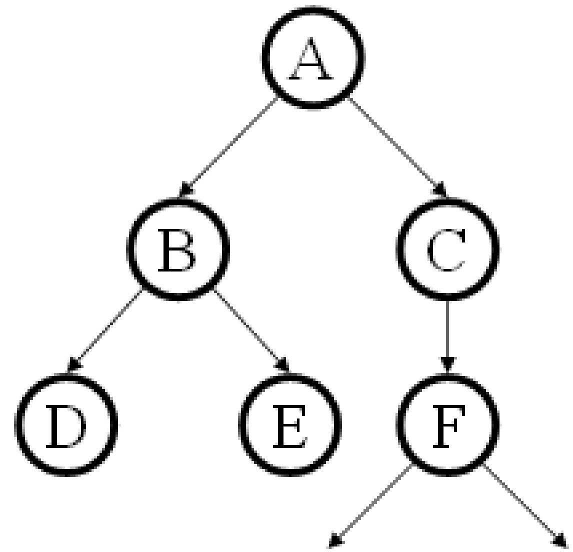


- For a dictionary, data will include a key and a value



Binary Tree Representation





Calculating height

What is the height of a tree with root **`root`**?

```
int treeHeight(Node root) {

      ???


}
```

Calculating height

What is the height of a tree with root **`root`**?

```
int treeHeight(Node root) {
  if(root == null)
    return -1;
  return 1 + max(treeHeight(root.left),
                 treeHeight(root.right));
}
```

Running time for tree with *n* nodes: $O(n)$ – single pass over tree

Note: non-recursive is painful – need your own stack of pending nodes; much easier to use recursion's call stack

Asymptotic Runtime of Recursion

Recurrence Definition:

A recurrence is a recursive definition of a function in terms of smaller values.

Example: Fibonacci numbers.

To analyze the runtime of recursive code, we use a recurrence by splitting the work into two pieces:

- Non-Recursive Work
- Recursive Work

Recursive version of sum:

```
int sum(int[] arr){
   return help(arr,0,arr.length);
}
int help(int[] arr, int lo, int hi) {
   if(lo==hi)   return 0;
   if(lo==hi-1) return arr[lo];
   int mid = (hi+lo)/2;
   return help(arr,lo,mid) + help(arr,mid,hi);
}
```

What's the recurrence T(n)?

- Non-Recursive Work: O(1)
- Recursive Work: T(n/2) * 2 halves

$T(n) = O(1) + 2*T(n/2)$

Solving That Recurrence Relation

1. Determine the recurrence relation. What is the base case?
   - *If* $T(1) = 1$, then $T(n) = 1 + 2*T(n/2)$

2. "Expand" the original relation to find an equivalent general expression *in terms of the number of expansions*.
   - $T(n) = 1 + 2 * T(n / 2)$
     $= 1 + 2 + 2 * T(n / 4)$
     $= 1 + 2 + 4 + \ldots$ for log(n) times
     $= \ldots$
     $= 2^{(\log n)} - 1$

3. Find a closed-form expression by setting *the number of expansions* to a value which reduces the problem to a base case
   - So $T(n)$ is $O(n)$

- Explanation: it adds each number once while doing little else

Solving Recurrence Relations Example 2

1. Determine the recurrence relation. What is the base case?
   – *If* $T(n) = 10 + T(n/2)$ and $T(1) = 10$

2. "Expand" the original relation to find an equivalent general expression *in terms of the number of expansions*.
   – $T(n) = 10 + 10 + T(n/4)$
     $= 10 + 10 + 10 + T(n/8)$
     $= \ldots$
     $= 10k + T(n/(2^k))$

3. Find a closed-form expression by setting *the number of expansions* to a value which reduces the problem to a base case
   – $n/(2^k) = 1$ means $n = 2^k$ means $k = \log_2 n$
   – So $T(n) = 10 \log_2 n + 8$ (get to base case and do it)
   – So $T(n)$ is $O(\log n)$

Really common recurrences

You can recognize some really common recurrences:

$T(n) = O(1) + T(n\text{-}1)$	linear
$T(n) = O(1) + 2T(n/2)$	linear
$T(n) = O(1) + T(n/2)$	logarithmic $O($`log` n)
$T(n) = O(1) + 2T(n\text{-}1)$	exponential
$T(n) = O(n) + T(n\text{-}1)$	quadratic
$T(n) = O(n) + T(n/2)$	linear
$T(n) = O(n) + 2T(n/2)$	$O($n `log` n) (divide and conquer sort)

Note big-Oh can also use more than one variable

- Example: can sum all elements of an n-by-m matrix in $O(nm)$