Final Exam Review
Part II

CS106B

Lecture 30

Anand Shankar

Summer 2018

Based on content by Ashley Taylor, Keith Schwarz, Marty Stepp, Anton Apostolatos, and others.
Thanks to Ashley Taylor for her feedback.

Yesterday's Lecture

- Backtracking
- Pointers
- Linked Lists
- Sorting

Today's Lecture

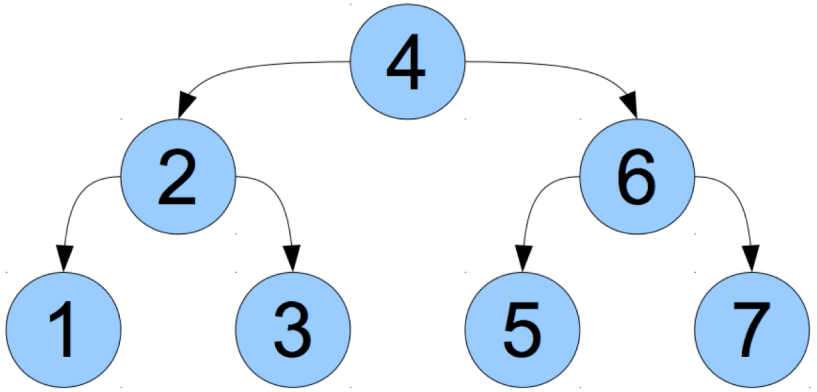
- Trees
- Graphs
- Hashing
- Testing Strategies (if we have time)

Trees

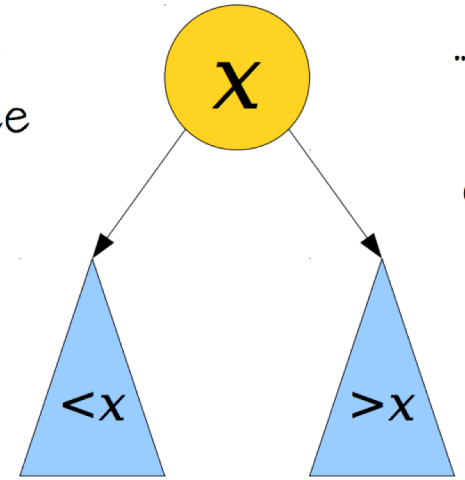
Binary Search Trees

A Binary Search Tree Is Either…

an empty tree, represented by `nullptr`, or…



… a single node, whose left subtree is a BST of smaller values …



… and whose right subtree is a BST of larger values.

```
struct BSTNode {
    BSTNode* left;
    BSTNode* right;
    int val;
};
```

Binary Search Trees

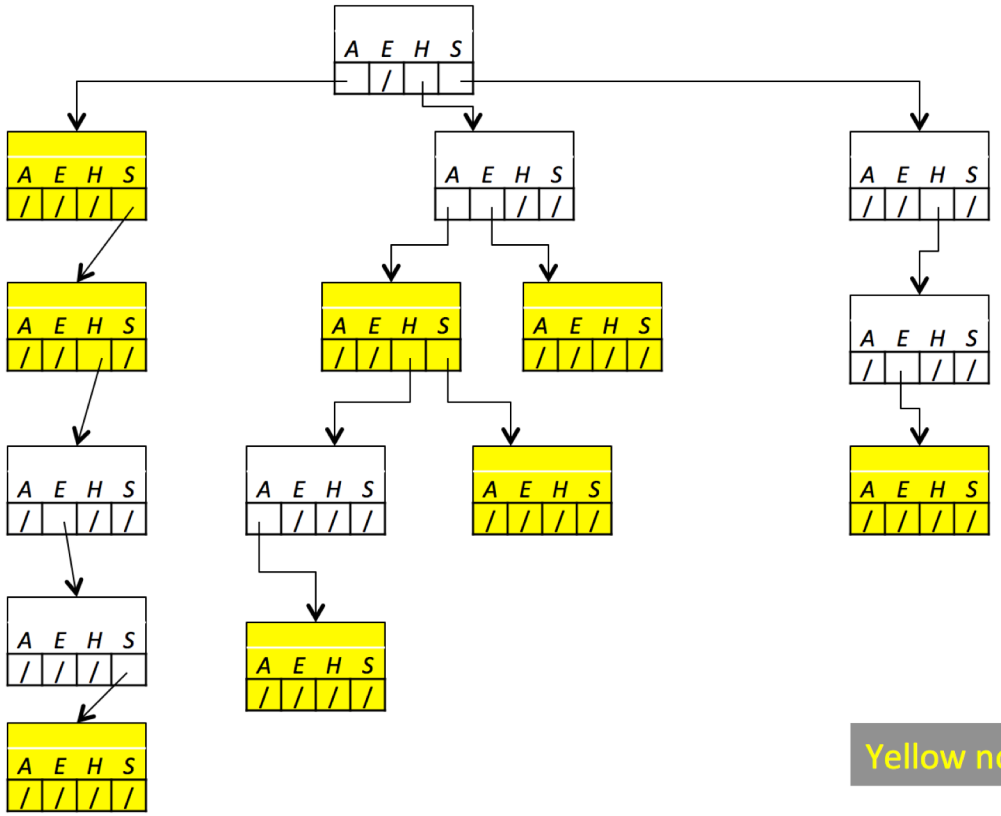
- Traversals:
  - *Inorder*: visit left subtree, then self, then right subtree
  - *Preorder*: visit self, then left subtree, then right subtree
  - *Postorder*: visit left subtree, then right subtree, then self
    - Typically used to free tree
  - There's also *level order*, which uses a Queue (very similar to BFS)
    - Fun fact: I used this in a job interview
- These questions generally involve figuring out which traversal you want to do, then adapting that traversal to the problem
- Solutions are generally recursive
- See: LineManager

Tries

- A ***trie*** is a tree structure optimized for “prefix” searches
- Instead of left/right child pointers, store a pointer to a subtree (child node) for each letter of the alphabet
- See: Autocomplete

```
struct TrieNode {
    bool isWord;
    TrieNode* children[26];
    // storing children
    // depends on alphabet
};
```

(Simplified four-letter alphabet for the sake of this trie)

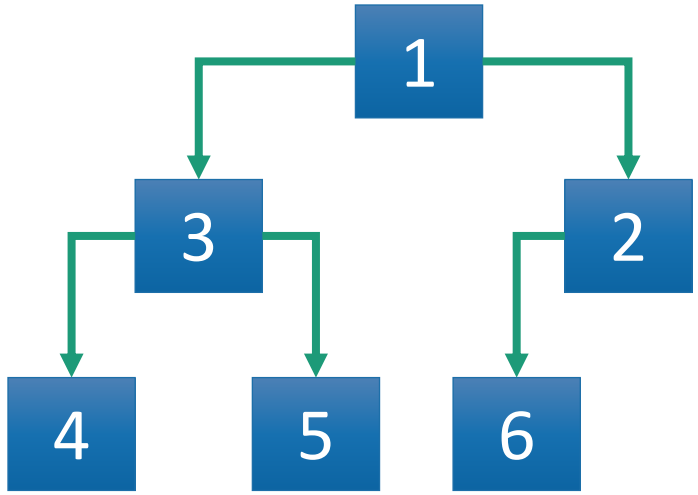


Yellow nodes are words!

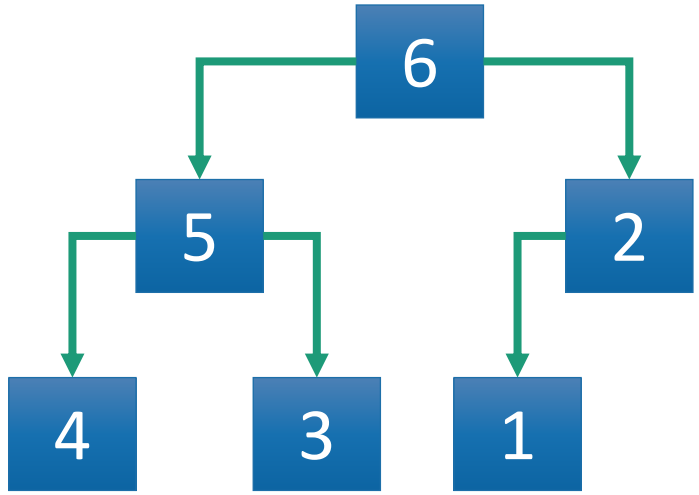
Binary Heaps

- *Complete tree:* every row of the heap, except for the last, must be full.
- Equivalently: each node, except leaf nodes, must have two children.
- Adding and removing involves *bubbling up* or *bubbling down*
- See: lecture 20 slides, textbook

Min-heap: each node ≤ its children
(root is smallest node)

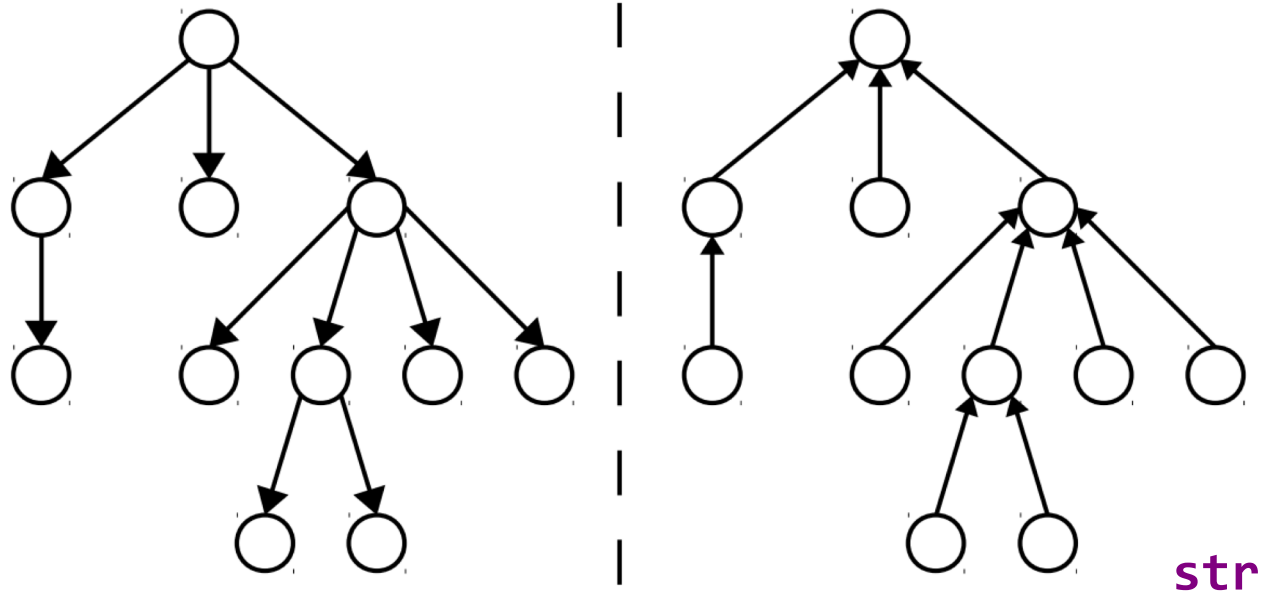


Max-heap: each node ≥ its children
(root is largest node)



Trees Problem, Part 1: Spaghetti!

- We've seen trees where each node has pointers to its children
- You can also have the opposite: each child has a pointer to its parent. These are called ***Spaghetti Stacks***

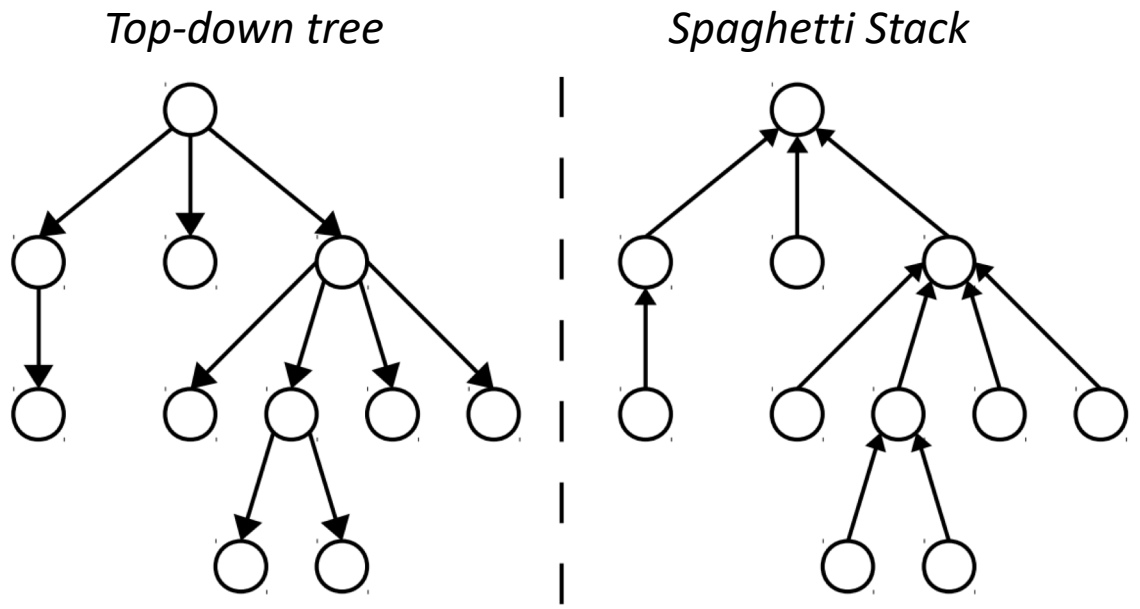


Write struct definitions on board

```
struct TreeNode {
    string val;
    Vector<TreeNode*> children;
};
```

```
struct SpaghettiNode {
    string val;
    SpaghettiNode* parent;
};
```

Let's Make Spaghetti



- With top-down trees, we usually store a pointer to the ***root node***
  - Can find any node from the root
- With spaghetti stacks, we must store a ***set of leaf nodes***
  - Can find any node from the appropriate leaf node
- ***Task:*** given a pointer to root of a normal, top-down tree, construct a Spaghetti stack for that tree. Return a Set of pointers to leaf nodes

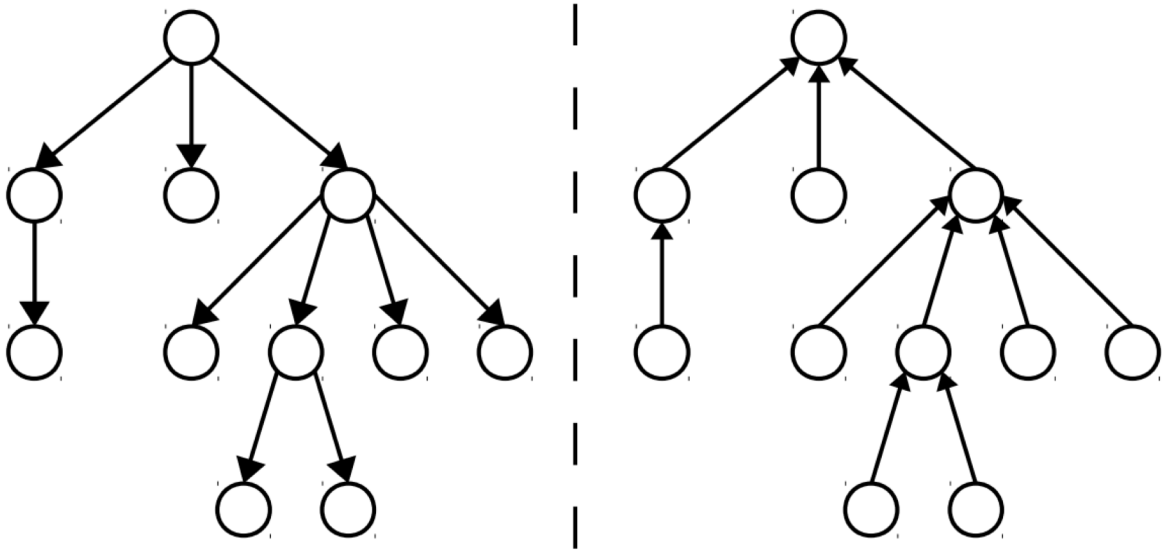
```
Set<SpaghettiNode*> spaghettify(TreeNode* root);
```

- **What questions do you have?**

Let's Make Spaghetti: Strategy

```
Set<SpaghettiNode*> spaghettify(TreeNode* root);
```

- Before writing code, think about ***strategy***
- ***Key insight:*** recursively convert each tree to a spaghetti stack by constructing each tree with knowledge of its parent

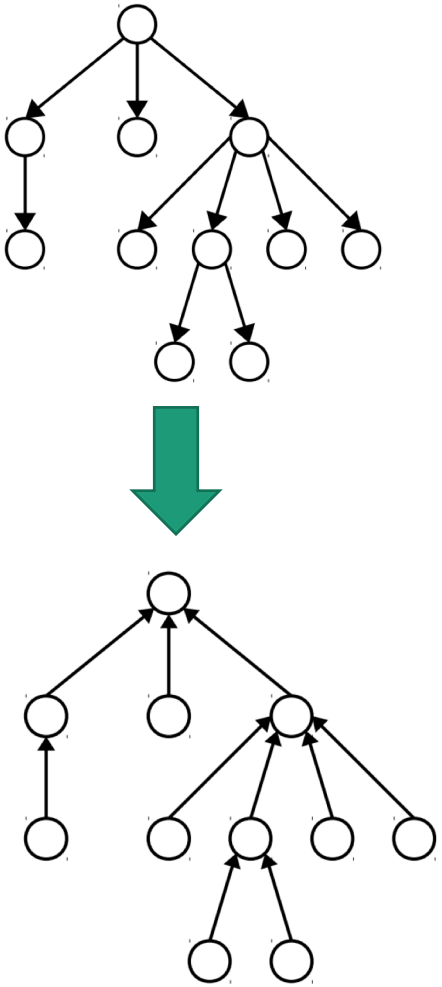


- We must keep track of the parent at every step
- We also need to keep track of the `Set<SpaghettiNode*>`
- Use a helper function with these two additional parameters

Let's Make Spaghetti: Wrapper Function

```
Set<SpaghettiNode*> spaghettify(TreeNode* root) {
    Set<SpaghettiNode*> result;
    spaghettifyRec(root, nullptr, result);
    return result;
}
```

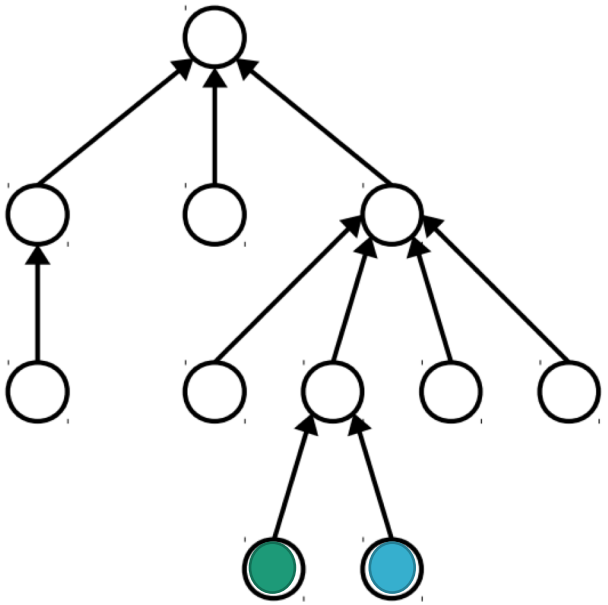
Let's Make Spaghetti: Helper Fn.



```
/* Builds a spaghetti stack from root whose parent in the spaghetti
 * stack is the node parent.
 */
void spaghettifyRec(TreeNode* root, SpaghettiNode* parent,
                    Set<SpaghettiNode>& result) {
    /* If there is nothing to build, we're done. */
    if (root == nullptr) return;

    /* Construct a new spaghetti node wired into the parent. */
    SpaghettiNode* sNode = new SpaghettiNode;
    sNode->value = root->value;
    sNode->parent = parent;

    /* If this is a leaf node, add it to the result set. */
    if (root->children.isEmpty()) {
        result += sNode;
    }

   /* Otherwise, build up all the children of this node as spaghetti stacks
    * that use the current node as a parent.
    */
    else {
        for (TreeNode* child: root->children) {
            spaghettifyRec(child, sNode, result);
        }
    }
}
```

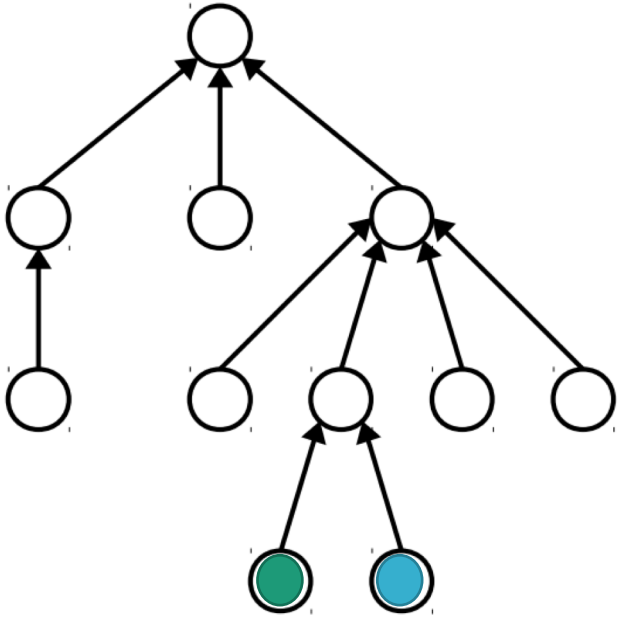
Trees Problem, Part 2: Cleaning Up Our Spaghetti

- In the last problem, we created a spaghetti stack
- Now, we need to free the memory associated with the spaghetti stack: `void freeSpaghettiStack(Set<SpaghettiNode*> leaves);`
- An initial attempt might be to walk from each leaf node to the root and free every node on the way
- Would that approach work on a spaghetti stack?



Be Free, Little Spaghetti

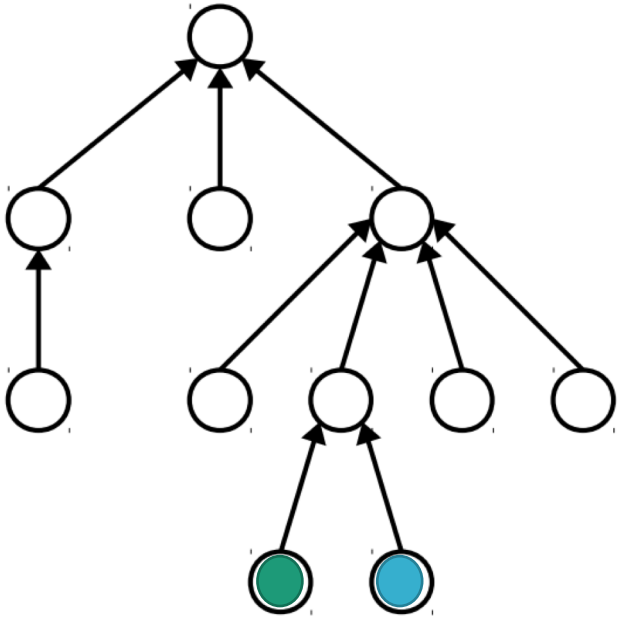
- If the green node frees its parent, the blue node should *not* free its parent
- How can we avoid freeing the same node multiple times?
  - Walk the tree deleting nodes and keep track of what we've deleted. If we encounter a node that was already deleted, stop exploring that path.
  - Or, make a `Set<SpaghettiNode*>` with all the nodes in the tree, then free each one



Be Free, Little Spaghetti: Solution 1

- Walk the tree deleting nodes and keep track of what we've deleted.
- If we encounter a node that was already deleted, stop exploring that path.

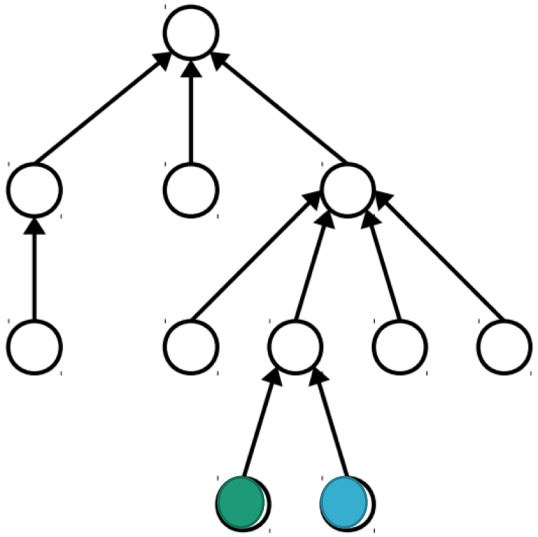
```
void freeSpaghettiStack(Set<SpaghettiNode*> leaves) {
    Set<SpaghettiNode*> deletedNodes;
    for (SpaghettiNode* leaf: leaves) {
        while (leaf != nullptr) {
            if (deletedNodes.contains(leaf)) break;
            deletedNodes += leaf;
            SpaghettiNode* next = leaf->parent;
            delete leaf;
            leaf = next;
        }
    }
}
```



Be Free, Little Spaghetti: Solution 2

- Make a `Set<SpaghettiNode*>` with all the nodes in the tree, then free each one

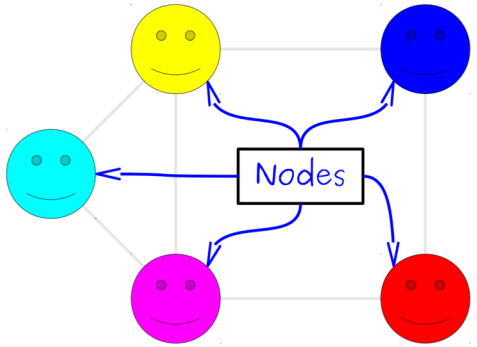
```
void freeSpaghettiStack(Set<SpaghettiNode*> leaves) {
    Set<SpaghettiNode*> nodes;
    for (SpaghettiNode* leaf: leaves) {
        for (SpaghettiNode* curr = leaf; curr != nullptr; curr = curr->parent) {
            nodes += curr;
        }
    }
    for (SpaghettiNode* node: nodes) {
        delete node;
    }
}
```

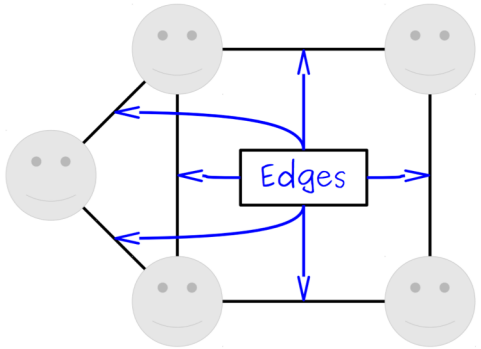


Graphs

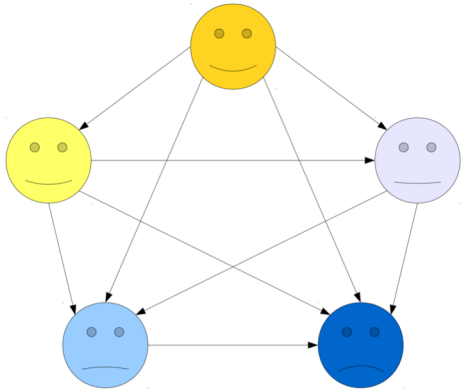
Graphs

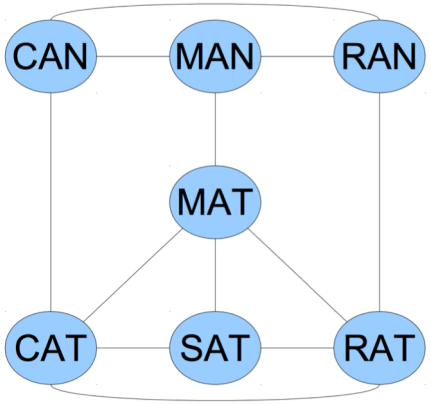
- *Graph:* mathematical structure for representing relationships
- Consists of a set of *nodes* connected by *edges*





- Can be *directed* or *undirected*





Graphs: Terminology

- *Connected:* every node is reachable from every other node
- *Cyclic:* there exists a path from a node back to the same node
- *Complete:* there exists an edge between every pair of nodes

- Representing a graph:
  - *Edge list*
  - *Adjacency list*
  - *Adjacency matrix*

Graphs: Breadth First Search

- *Breadth-first search* will find all nodes reachable from the starting node
- It will visit them in increasing order of hops/distance
- Runtime: O(V + E)
- See: word ladder assignment, lecture slides for animated examples

```
function bfs(v):
    add v to the queue.
    while queue is not empty:
        dequeue a node n.
        enqueue n's unseen neighbors.
```

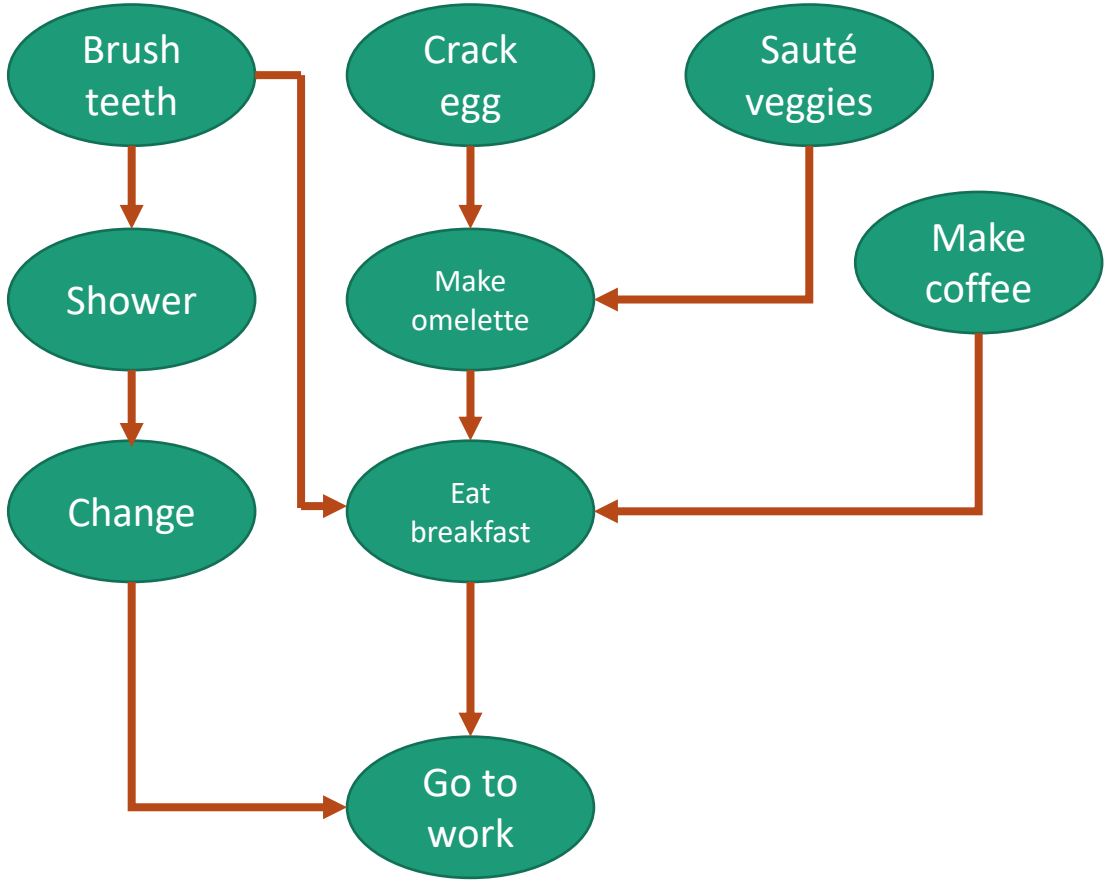
Graphs: Depth First Search (DFS)

- Starting a ***depth-first search*** at a given node will find all nodes reachable from that node
- Runtime: O(V + E)
- Usually implemented using recursion or a Stack
- See: assignment 6, lecture slides for animated examples

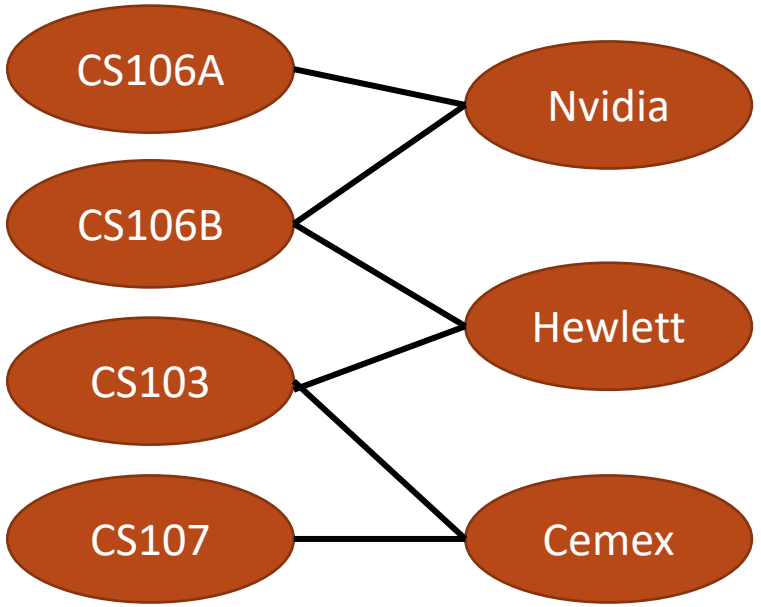
```
function dfs(v):
    mark v as seen.
    for each of v's unvisited neighbors n:
        dfs(n)
```

Graphs: Topological Sort

- Want to order tasks such that every task's prerequisites appear *before* the task itself
- If you need to make coffee before eating breakfast, then coffee should appear before breakfast in the *topological ordering*
- Works only on *directed, acyclic graphs (DAGs)*
- Runtime: O(V + E)
- Pseudocode omitted; you implemented it on assignment 6!



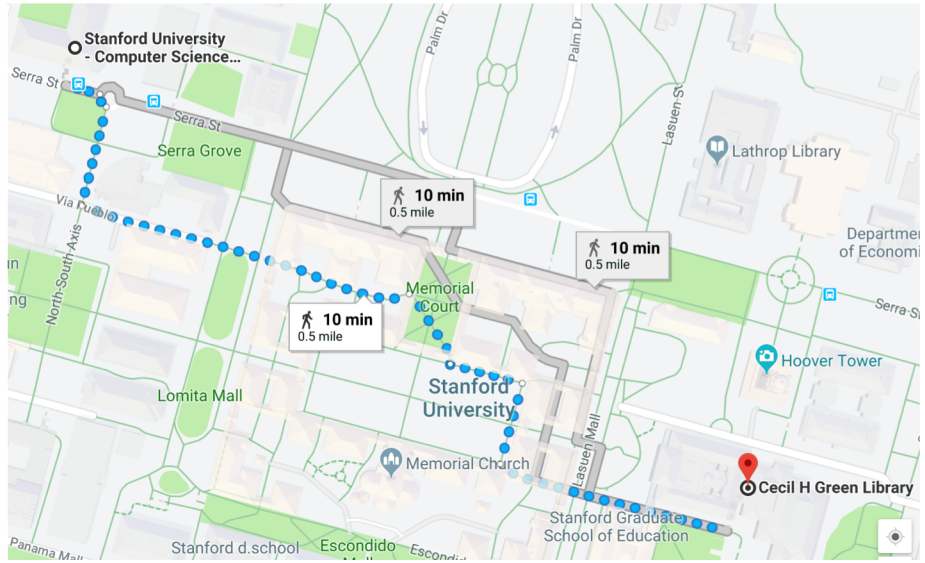
Graphs: Bipartite Graph Matching



- A *bipartite graph* is a graph with two types of nodes (left-hand side and right-hand side), where all the undirected edges go from LHS to the RHS
- A *matching* is a set of edges such that each node is connected to at most one edge
  - *Maximum matching*: largest such set of edges
- Pseudocode omitted; see: HW6, animated examples from lecture
- Side note: you should take the classes on the LHS – they're cool!

Graphs: Dijkstra's Algorithm

- Find *least-cost path* (only applies to graphs with weighted edges)
- Based on BFS but uses a *Priority Queue* instead of a Queue to visit nodes
- Google Maps uses a modified version of this algorithm
- Runtime: O(E log V)
- See: lecture 24 (includes animations), textbook



Graphs: Dijkstra's Algorithm Pseudocode

function dijkstra(v_1, v_2):

consider every vertex to have a cost of infinity, except v_1 which has a cost of 0.

create a *priority queue* of vertexes, ordered by cost, storing only v_1.

while the *priority queue* is not empty:

dequeue a vertex *v* from the *priority queue*, and mark it as **visited**.

for each unvisited neighbor, *n*, of *v*, we can reach *n*

with a total **cost** of (*v*'s cost + the weight of the edge from *v* to *n*).

if this cost is cheaper than *n*'s current cost,

we should **enqueue** the neighbor *n* to the *priority queue* with this new cost,

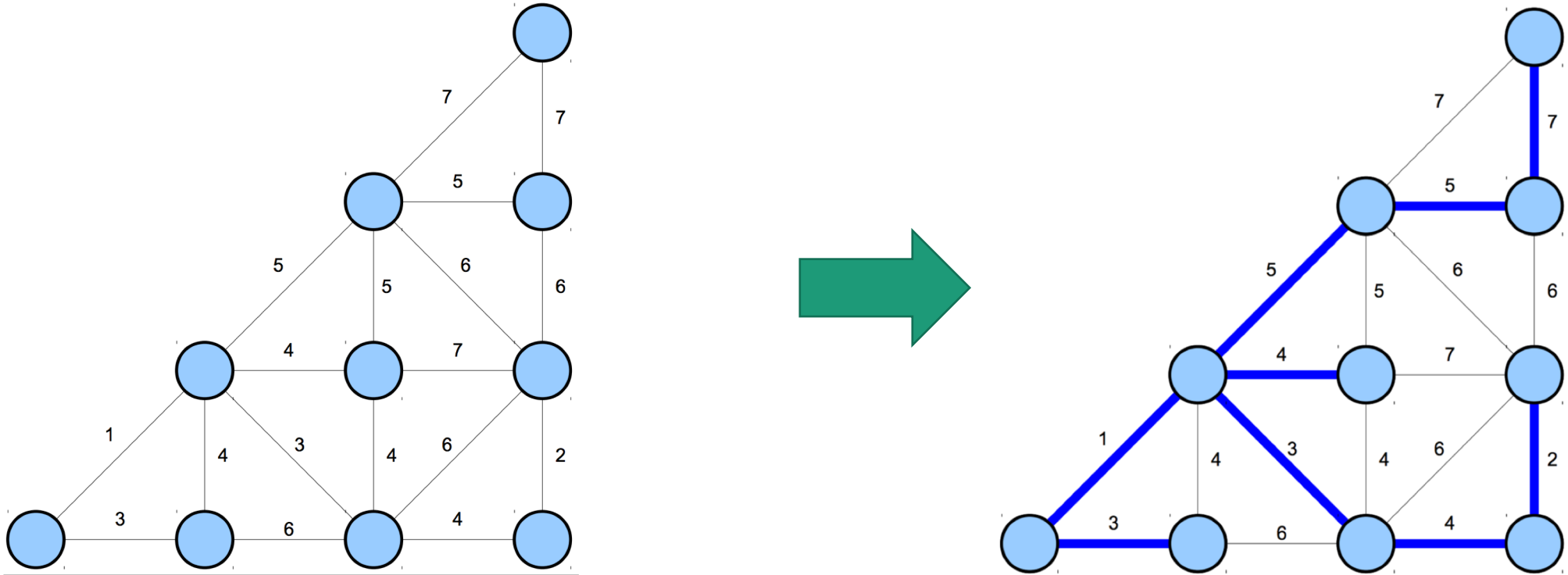
and remember *v* was its previous vertex.

when we are done, we can **reconstruct the path** from v_2 back to v_1

by following the path of previous vertices.

Graphs: Kruskal's Algorithm

- A *spanning tree* in an undirected graph is a set of edges with no cycles that connects all nodes
- A *minimum spanning tree* (or *MST*) is a spanning tree with the least total cost. Kruskal's Algorithm finds an MST.



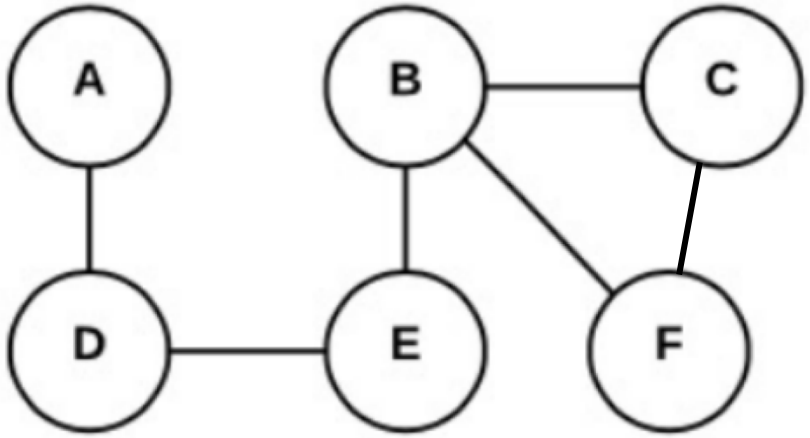
Graphs: Kruskal's Algorithm Pseudocode

```
function kruskal(graph):
    MST = empty set
    Place all edges into a priority queue based on their weight (cost).
    While the priority queue is not empty:
        Dequeue an edge e from the priority queue.
            If e's endpoints aren't already connected, add that edge into the MST.
            Otherwise, skip the edge.
     return MST
```

- Using a priority queue ensures that we add cheap edges before adding expensive edges
- Runtime: O(E log V)
- See: lecture 24 (includes animations)

Graphs Problem: Eccentricity

- The *distance* between two nodes in a graph is the length of the shortest path between them.
- The *eccentricity* of a node in a graph is the maximum distance between that node and any other node in the graph (of the nodes that it can actually reach)



- Eccentricity of A is 4
- Eccentricity of F is 4
- Eccentricity of D is 3

How To Be Eccentric: Strategy

- *Task:* write a function that, given an adjacency list, returns the eccentricity of the given node in the given graph

```
int eccentricityOf(const Map<string, Set<string>>& graph, const string& node);
```

- *Strategy:* Which graph algorithm should we use? Why?
- Imagine that we run a BFS starting at a given node
- The very last node we dequeue has to be as far away as possible from the source node, since
  (1) BFS visits nodes in increasing order of distance, and
  (2) no other nodes will be visited after it.
- We'll run BFS from the given node; the last node we see is the furthest away

How To Be Eccentric: Solution (1/2)

```
int eccentricityOf(const Map<string, Set<string>>& graph, const string& node) {
    Queue<string> worklist;
    worklist.enqueue(node);

    /* Associate each element with a parent node. The parent node is the node that added it
     * into the queue, which means it'll be one step closer to the start node.
     */
    HashMap<string, string> parents;

    /* Track the last node we've seen. */
    string last = node;

    /* Do the BFS! */
    while (!worklist.isEmpty()) {
        string curr = worklist.dequeue();

        for (string next: graph[curr]) {
            /* Don't revisit something we've already enqueued. */
            if (!parents.containsKey(next)) {
                parents[next] = curr; // We discovered this node.
                worklist.enqueue(next);
            }
        }
        /* Remember the last node we've seen. */
        last = curr;
    }
```

Continued on next slide

```
    /* Track back from this node to the start, counting how many steps were
     * needed.
     */
    int result = 0;
    while (parents.containsKey(last)) {
        last = parents[last];
        result++;
    }
    return result;
}
```

How To Be Eccentric: Solution (2/2)

Hashing

Hashing: Hash Functions

- Basic definition: a ***hash function*** maps something (like an `int` or `string`) to a number
- A **valid** hash function is ***deterministic:*** always returns the same number given two inputs that are considered equal
- A **good** hash function distributes the values uniformly over all the numbers

Hashing: Data Structures

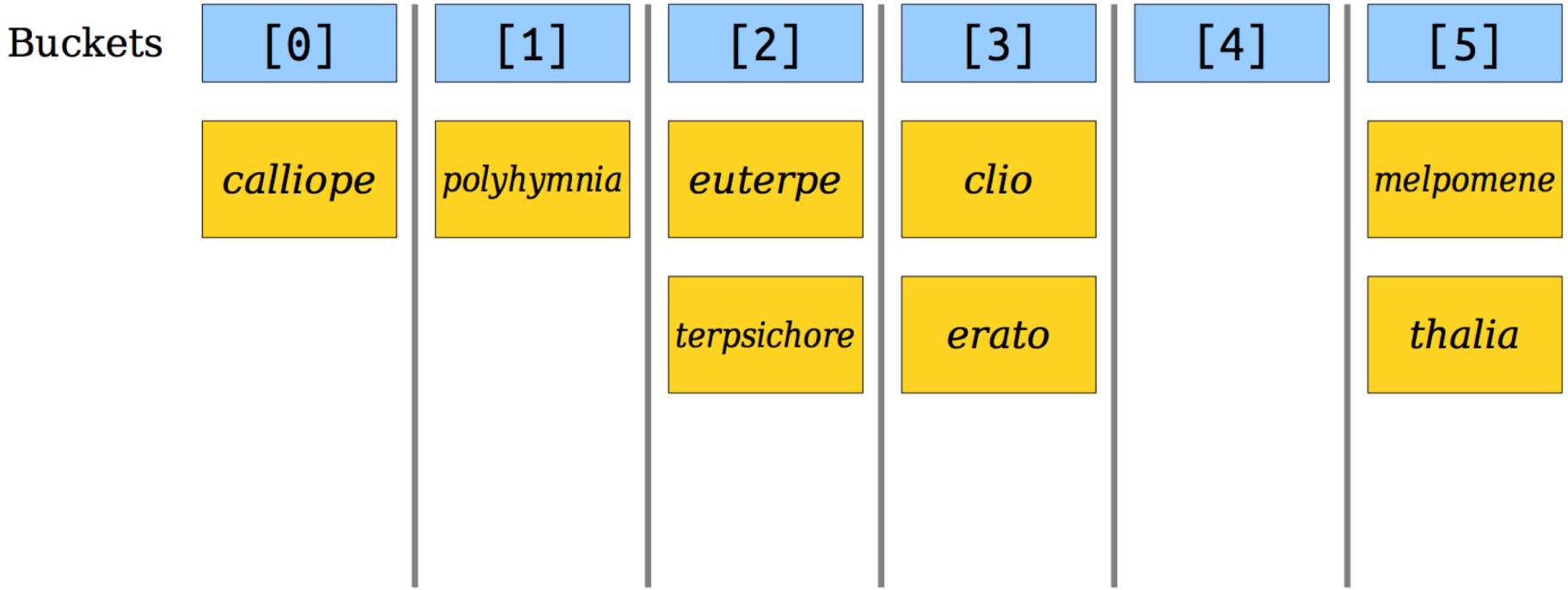
- `HashMap` and `HashSet` use hash functions to achieve amortized O(1) addition, removal, and lookup. To do so, they:

1. Maintain a large number of small collections called ***buckets***
2. Find a ***rule*** that lets us tell where each object should go
3. To find something, ***only look*** in the bucket assigned to it

- See: lectures 26 and 27

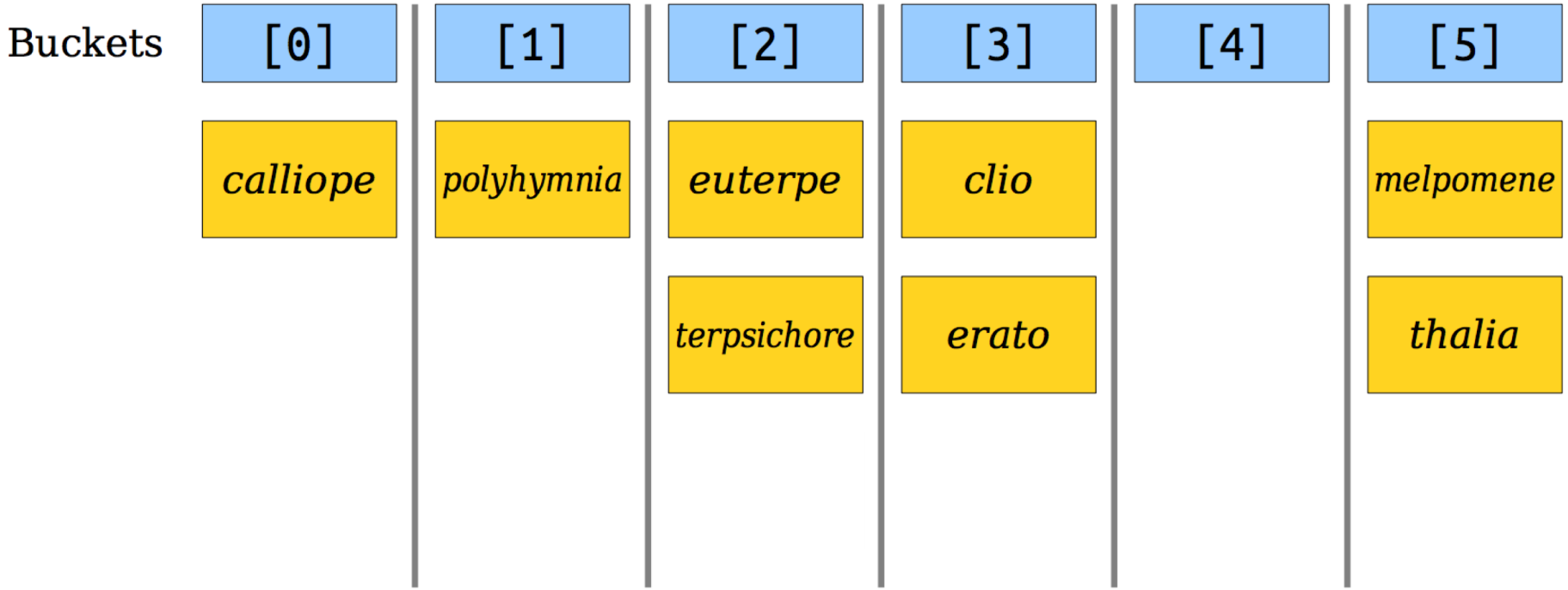
- Side note: hashing is super useful for interview questions.

Hashing: Data Structures Example



- We want to find erato
- Suppose the hash function tells us it's in bucket 3
- We only need to look in bucket 3 and see if erato is there

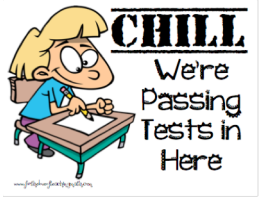
Hashing: Data Structures Example



- We want to add urania to the HashSet
- Suppose the hash function tells us it's in bucket 2
- We add urania to bucket 2

Testing Strategies

Testing Strategies



- Before Saturday, take the practice test *under realistic conditions*
- Read all questions before answering any
- Don't write code until you silently *tell yourself (in words)* what you're going to do
- *Draw pictures* (especially for trees/pointer questions)
- *Break the problem down* into smaller parts (especially for recursion)
- If you get stuck, try a different question and *come back later*
- *Pace yourself*
- Get lots of *sleep* before the exam (trust me…taking exams while sleepy is not fun)
- Your lecturer, head TA and SLs want you to do well (seriously!)
- *You can do it!*

Congratulations, you're done with your last lecture of CS106B.

You have worked hard this quarter, and the entire course staff is very proud of your efforts.

We hope you'll continue to explore CS, be it here or elsewhere.