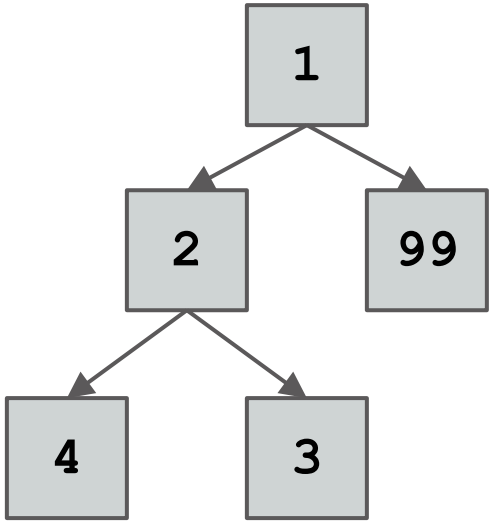
Recitation 9

Prelim Review

Heaps

Review: Binary heap

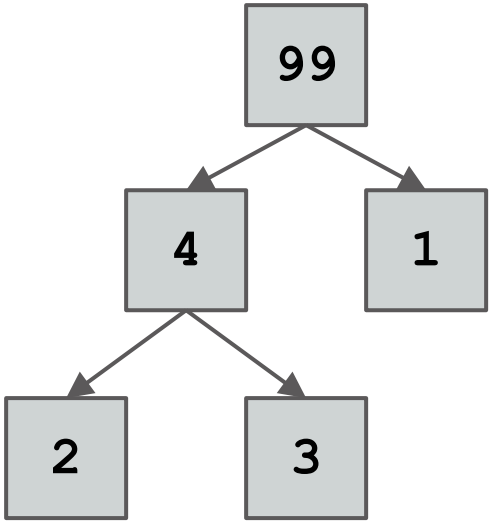
min heap



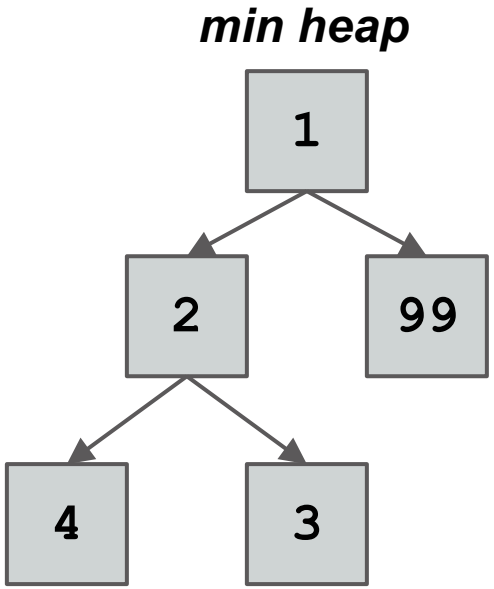
`PriorityQueue`

- Maintains max or min of collection (no duplicates)
- Follows *heap order invariant* at every level
- Always balanced!
- **worst case**:
  O(log n) insert
  O(log n) update
  O(1) peek
  O(log n) removal

max heap



Review: Binary heap



How do we insert element 0 into the min heap?

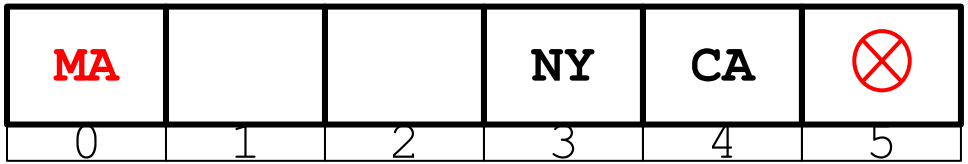
After we remove the root node, what is the resulting heap?

How are heaps usually represented? If we want the right child of index i, how do we access it?

Hashing

Review: Hashing

`HashSet<String>`



Method	Expected Runtime	Worst Case
`add`	*O(1)*	*O(n)*
`contains`	*O(1)*	*O(n)*
`remove`	*O(1)*	*O(n)*

load factor, for open addressing:

$$\frac{\text{number of non-null entries}}{\text{size of array}}$$

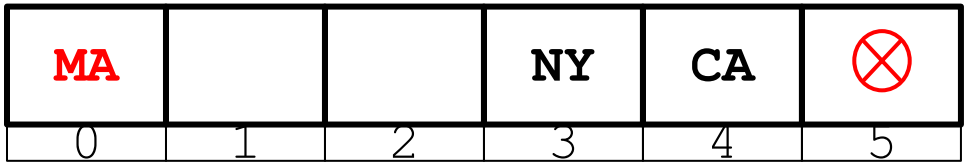
load factor, for chaining:

$$\frac{\text{size of set}}{\text{size of array}}$$

If load factor becomes > 1/2, create an array twice the size and rehash every element of the set into it, use new array

Review: Hashing

`HashSet<String>`



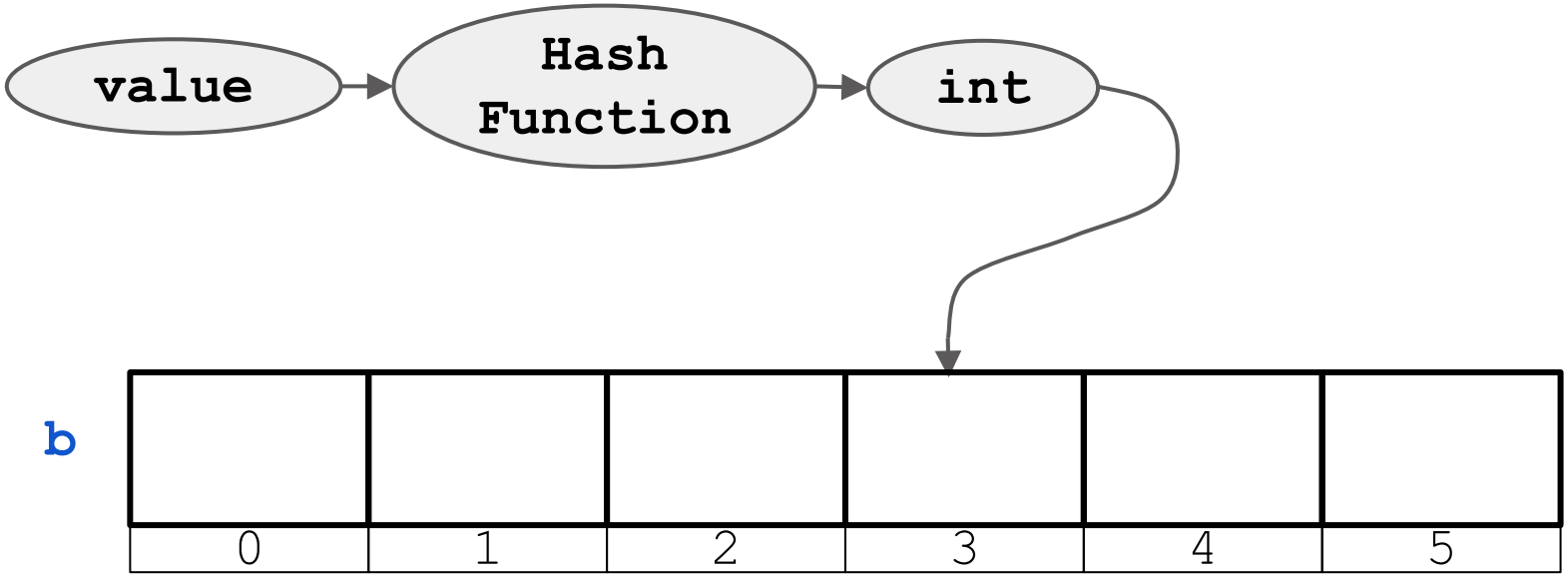
Method	Expected Runtime	Worst Case
`add`	O(1)	O(n)
`contains`	O(1)	O(n)
`remove`	O(1)	O(n)

`HashMap<String,Integer>`

to	2
be	2
or	1
not	1
that	1
is	1
the	1
question	1

Review: Hashing

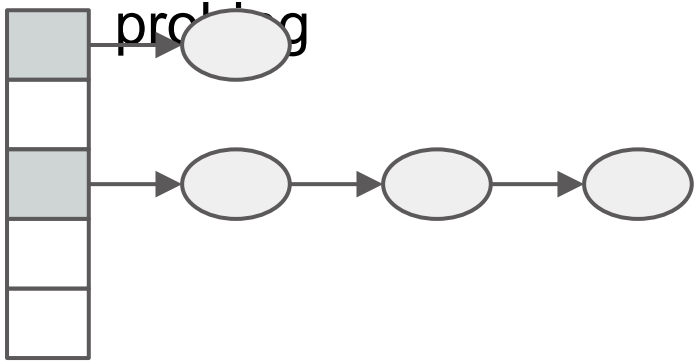
Idea: finding an element in an array takes constant time when you know which index it is stored in



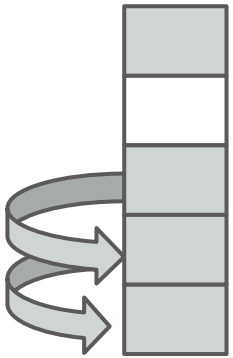
Collision resolution

Two ways of handling collisions:

1. Chaining

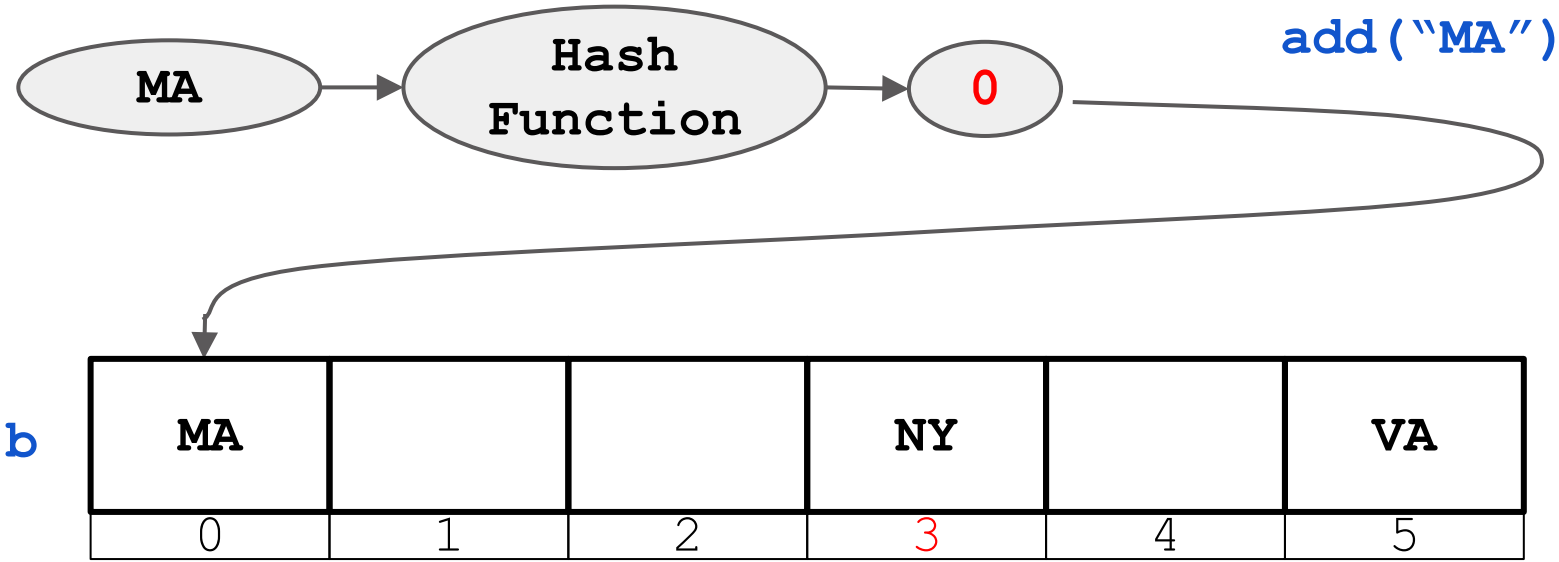


2. Open Addressing with linear



Load factor: b's saturation

Load factor: $\lambda = \frac{\#\ of\ entries}{length\ of\ array} = \frac{3}{6}$

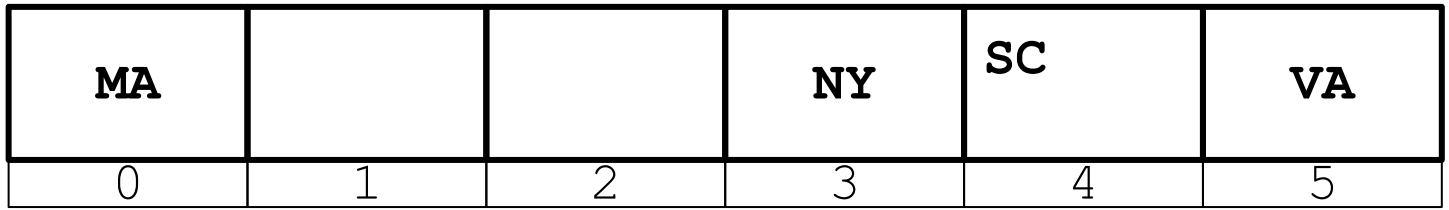


Question: Hashing

Using linear probing to resolve collisions,

1. Add element SC (hashes to 3).
2. Remove VA (hashes to 5).
3. Check to see if MA (hashes to 0) is in the set.
4. What should we do if we override equals()?

b



Trees

Definition

Data structure with nodes

Each node has:

- 0 or more children
- Exactly 1 parent (except the root which has none)
- All nodes are reachable from root

Binary tree - each node has at most 2 children

Use in Recursion

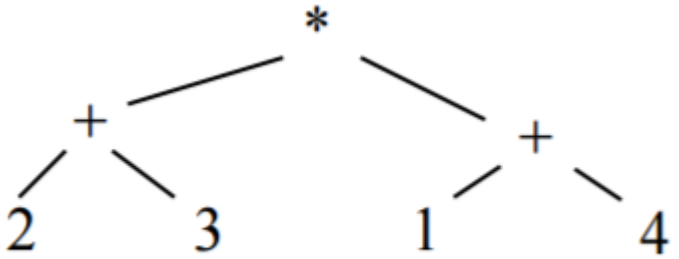
- Data Structure well suited for recursion
  - Binary tree is either
    - Empty
    - A value (root), left binary tree, right binary tree
  - The first becomes the base case and the second becomes the recursive case

Traversals

Pre-Order	(Root, Left ST, Right ST)
In-Order	(Left ST, Root, Right ST)
Post-Order	(you get the pattern :))

Expression Trees

Nodes represent operations or values



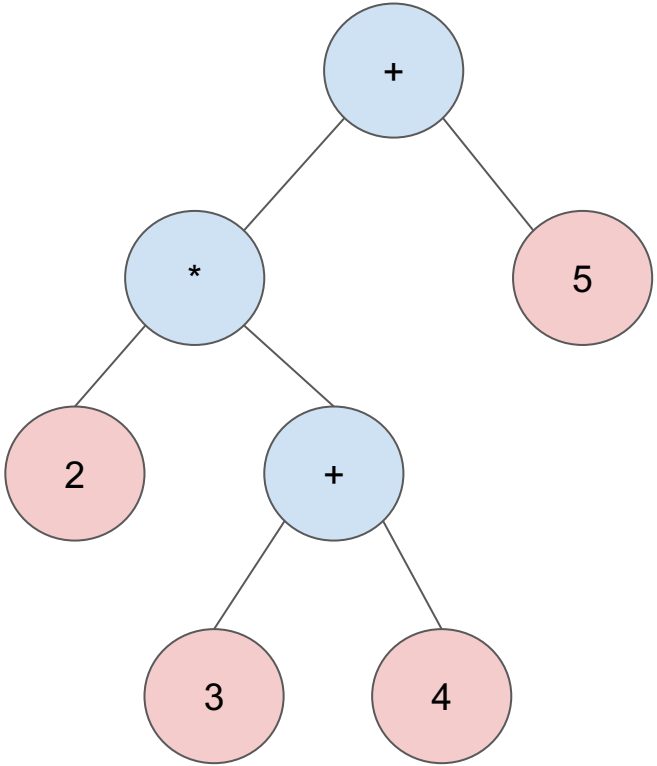
Leaf nodes are values. Non-leaf nodes are operations that work on their children.

Post-Order

Post-order traversal output is code for a stack machine. (recursive descent parsing)

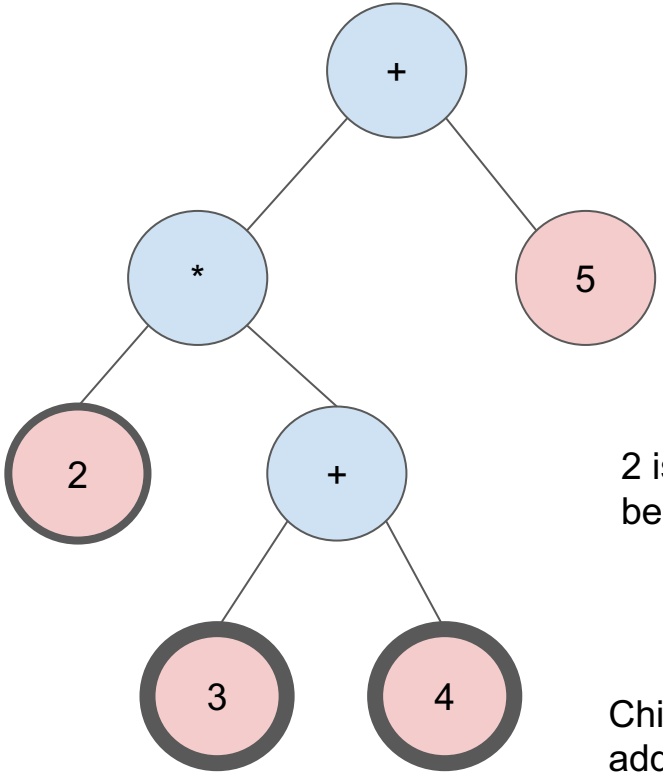
1. Visit nodes in tree in post-order.
2. Push results from a node onto a stack.
3. When you hit a node that is an operation, pop off the required number of elements from the stack and then push the result.

Stack machine as postorder traversal



- Each node evaluates its children (arguments) before itself (operator)
- Result becomes the argument for the next highest level

Stack machine as postorder traversal

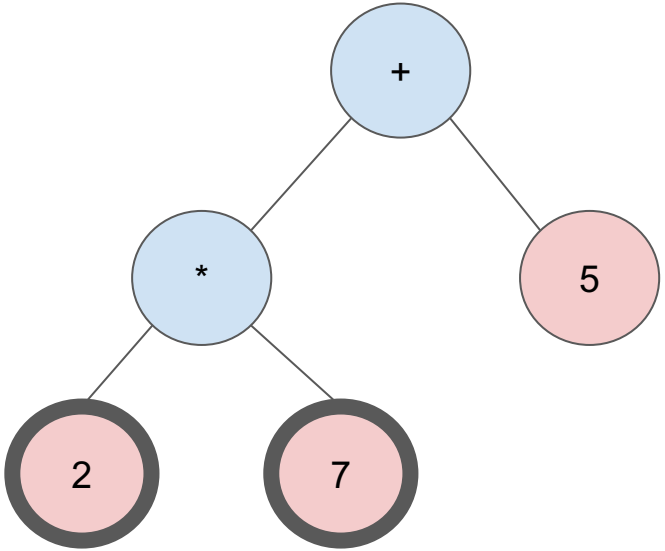


- Each node evaluates its children (arguments) before itself (operator)
- Result becomes the argument for the next highest level

2 is evaluated first. However, the other child subtree needs to be evaluated before the product operator can proceed.

Children are evaluated to 3 and 4, add operator yields 7.

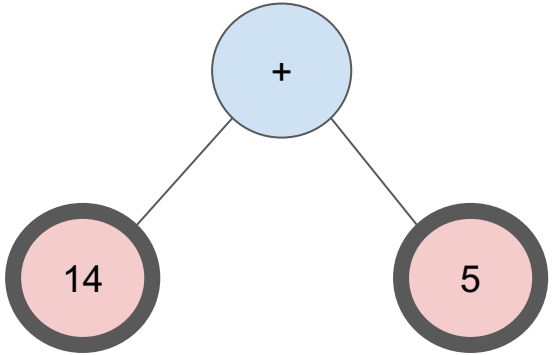
Stack machine as postorder traversal



Children are evaluated to 2 and 7, multiply operator yields 14.

- Each node evaluates its children (arguments) before itself (operator)
- Result becomes the argument for the next highest level

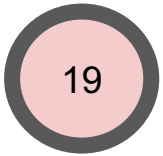
Stack machine as postorder traversal



Children are evaluated to 14 and 5, add operator yields 19.

- Each node evaluates its children (arguments) before itself (operator)
- Result becomes the argument for the next highest level

Stack machine as postorder traversal

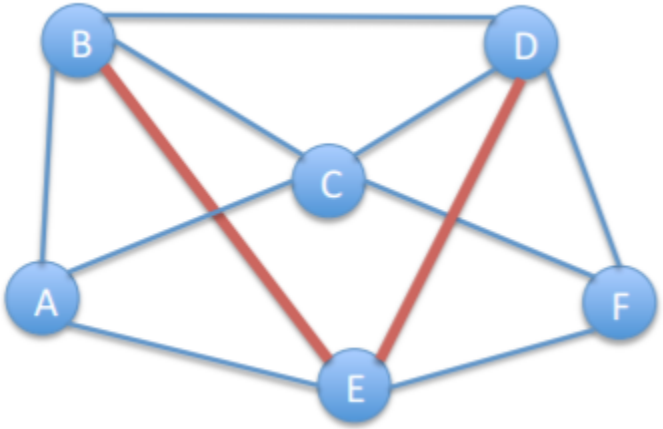


- Each node evaluates its children (arguments) before itself (operator)
- Result becomes the argument for the next highest level

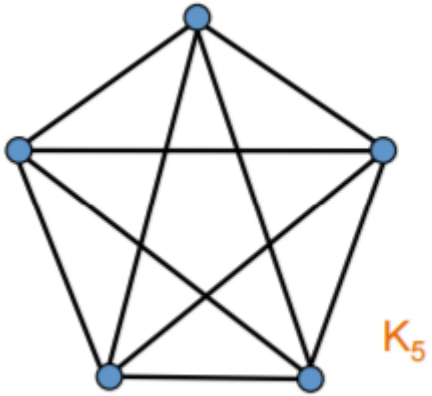
Graphs

Planar Graphs

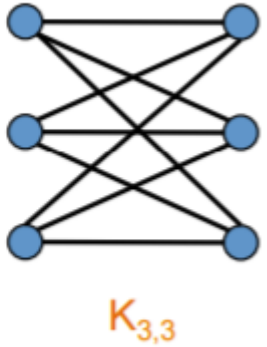
Graph is planar if it can be drawn in the plane without edges crossing. This allows you to 4-color a graph



Yes

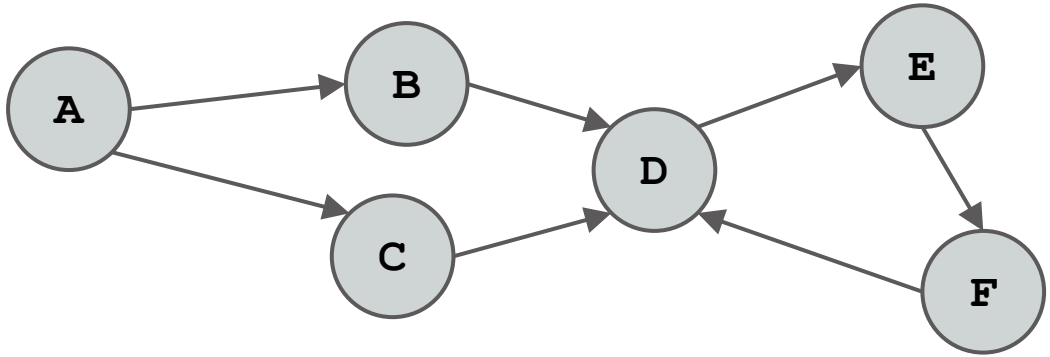


NO



NO

Question: What is BFS and DFS?



1. Starting from node A, run BFS and DFS to find node Z. What is the order in which nodes were processed? Visit neighbors in alphabetical order.
2. What is the difference between DFS and BFS?
3. What algorithm would be better to use if our graph were near infinite and a node was nearby?
4. Is Dijkstra's more like DFS or BFS? Why?
5. Can you run topological sort on this graph?

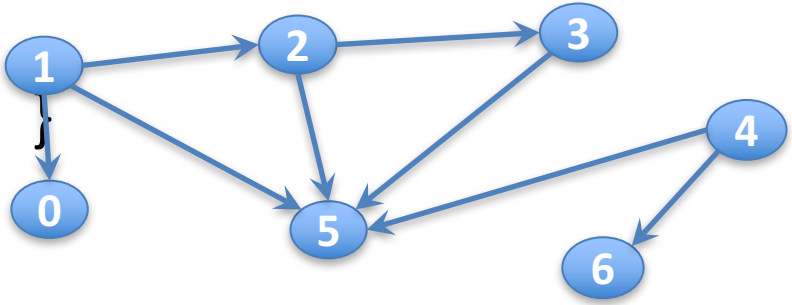
Depth-First Search

```
/** Visit all nodes that are REACHABLE from u.
    Precondition: u is not visited*/
public static void dfs(int u) {
```

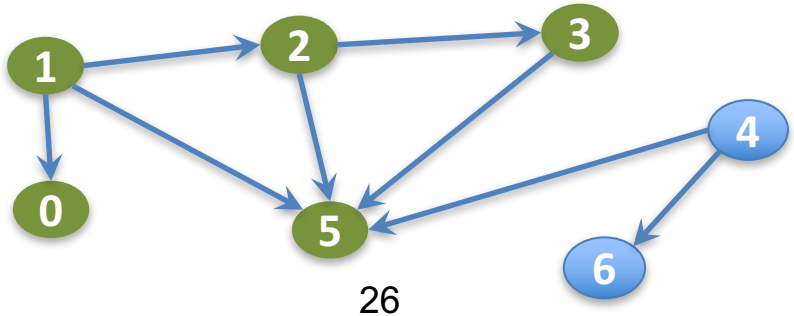
Let u be **1**

The nodes REACHABLE from 1 are 1, 0, 2, 3, 5

Start



End

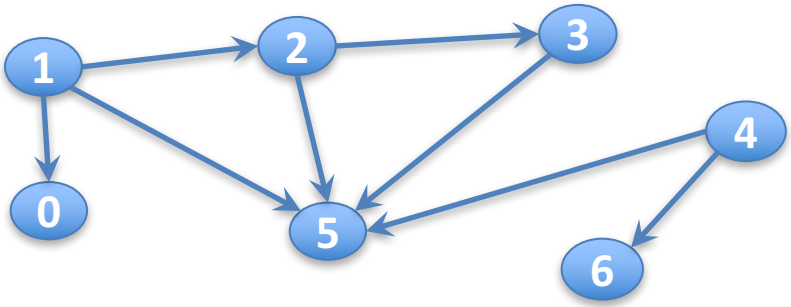


Depth-First Search

```
/** Visit all nodes REACHABLE from u.
    Precond: Node u is unvisited. */
public static void dfs(int u) {



}
```



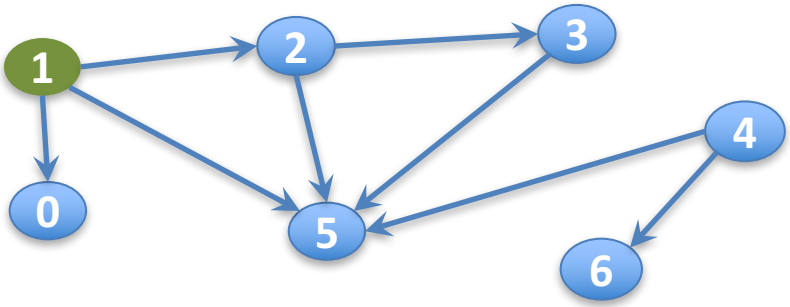
Let u be **1**

The nodes REACHABLE from 1 are 1, 0, 2, 3, 5

Depth-First Search

```
/** Visit all nodes REACHABLE from u.
    Precond: Node u is unvisited. */
public static void dfs(int u) {
   visited[u] = true;


}
```



Let u be **1**

The nodes REACHABLE from 1 are 1, 0, 2, 3, 5

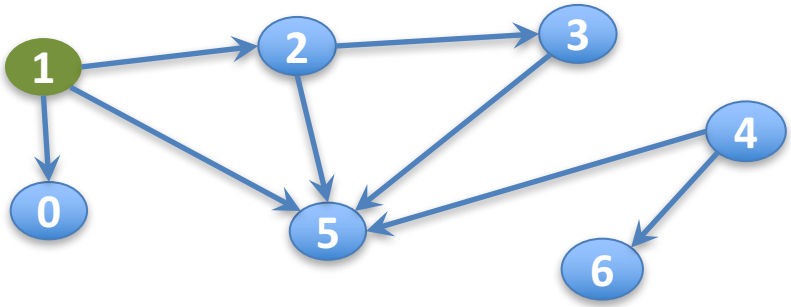
Depth-First Search

```
/** Visit all nodes REACHABLE from u.
    Precond: Node u is unvisited. */
public static void dfs(int u) {
   visited[u] = true;


}
```

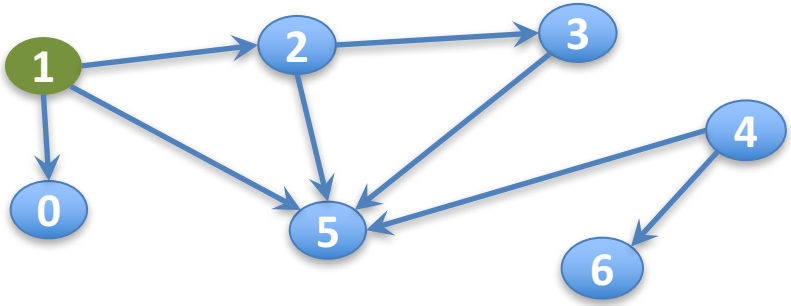
Let u be **1** (visited)

The nodes to be visited are 0, 2, 3, 5



Depth-First Search

```
/** Visit all nodes REACHABLE from u.
    Precond: Node u is unvisited. */
public static void dfs(int u) {
  visited[u] = true;
  for all edges (u, v) leaving u:
    if v is unvisited then dfs(v);
}
```



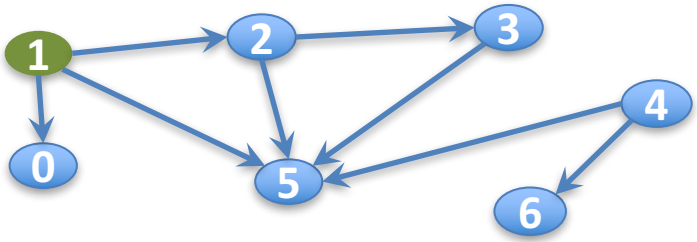
Let u be **1** (visited)

The nodes to be visited are 0, 2, 3, 5

Have to do DFS on all unvisited neighbors of u!

Depth-First Search

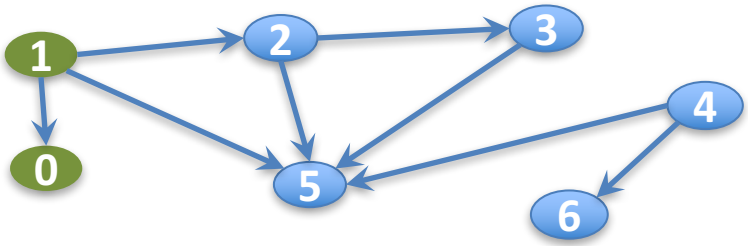
```
/** Visit all nodes REACHABLE from u.
    Precond: Node u is unvisited. */
public static void dfs(int u) {
  visited[u] = true;
  for all edges (u, v) leaving u:
    if v is unvisited then dfs(v);
}
```



Suppose the **for** loop visits neighbors in numerical order. Then **dfs(1)** visits the nodes in this order: 1 …

Depth-First Search

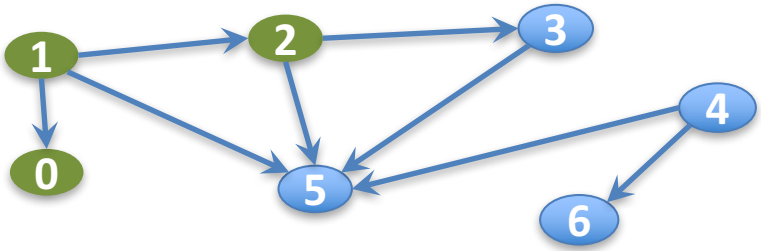
```
/** Visit all nodes REACHABLE from u.
    Precond: Node u is unvisited. */
public static void dfs(int u) {
  visited[u] = true;
  for all edges (u, v) leaving u:
    if v is unvisited then dfs(v);
}
```



Suppose the **for** loop visits neighbors in numerical order. Then **dfs(1)** visits the nodes in this order: 1, 0 …

Depth-First Search

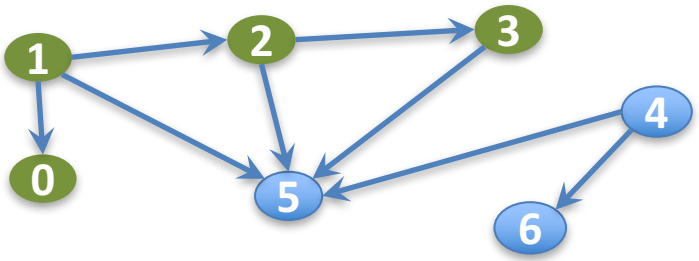
```
/** Visit all nodes REACHABLE from u.
    Precond: Node u is unvisited. */
public static void dfs(int u) {
  visited[u] = true;
  for all edges (u, v) leaving u:
    if v is unvisited then dfs(v);
}
```



Suppose the **for** loop visits neighbors in numerical order. Then **dfs(1)** visits the nodes in this order: 1, 0, 2 …

Depth-First Search

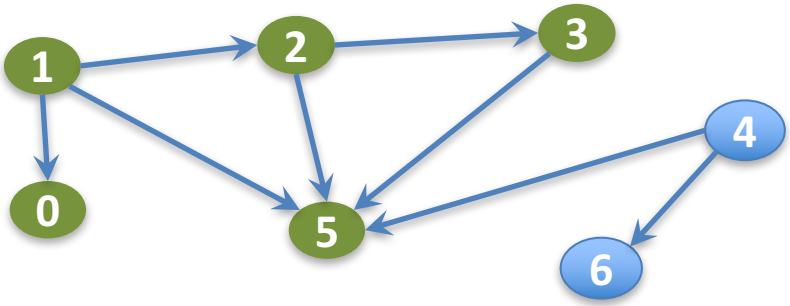
```
/** Visit all nodes REACHABLE from u.
    Precond: Node u is unvisited. */
public static void dfs(int u) {
  visited[u] = true;
  for all edges (u, v) leaving u:
    if v is unvisited then dfs(v);
}
```



Suppose the **for** loop visits neighbors in numerical order. Then **dfs(1)** visits the nodes in this order: 1, 0, 2, 3 …

Depth-First Search

```
/** Visit all nodes REACHABLE from u.
      Precond: Node u is unvisited. */
public static void dfs(int u) {
   visited[u] = true;
   for all edges (u, v) leaving u:
     if v is unvisited then dfs(v);
}
```



Suppose the **for** loop visits neighbors in numerical order. Then **dfs(1)** visits the nodes in this order: 1, 0, 2, 3, 5

Depth-First Search

```
/** Visit all nodes REACHABLE from u.
    Precond: Node u is unvisited. */
public static void dfs(int u) {
  visited[u] = true;
  for all edges (u, v) leaving u:
    if v is unvisited then dfs(v);
}
```

Suppose n nodes are REACHABLE along e edges (in total). What is

- Worst-case execution?
- Worst-case space?

Depth-First Search

```
/** Visit all nodes REACHABLE from u.
    Precond: Node u is unvisited. */
public static void dfs(int u) {
   visited[u] = true;
   for all edges (u, v) leaving u:
     if v is unvisited then dfs(v);
}
```

That's all there is to basic DFS. You may have to change it to fit a particular situation.

If you don't have this spec and you do something different, it's probably wrong.

Example: Use different way (other than array visited) to know whether a node has been visited

Example: We really haven't said what data structures are used to implement the graph

Recommended Practice for Dijkstra

1. Create a graph with labeled nodes and pick a starting node, x
2. Draw a table to keep track of the Frontier, Settled, and Far Off sets for each iteration
3. Include in the table, the distance from x to each node in the graph for each iteration
4. Fill in the table by working through Dijkstra's

Big O

See the Study Habits Note on the course Piazza. There is a 2-page pdf file that says how to learn what you need to know for O-notation.

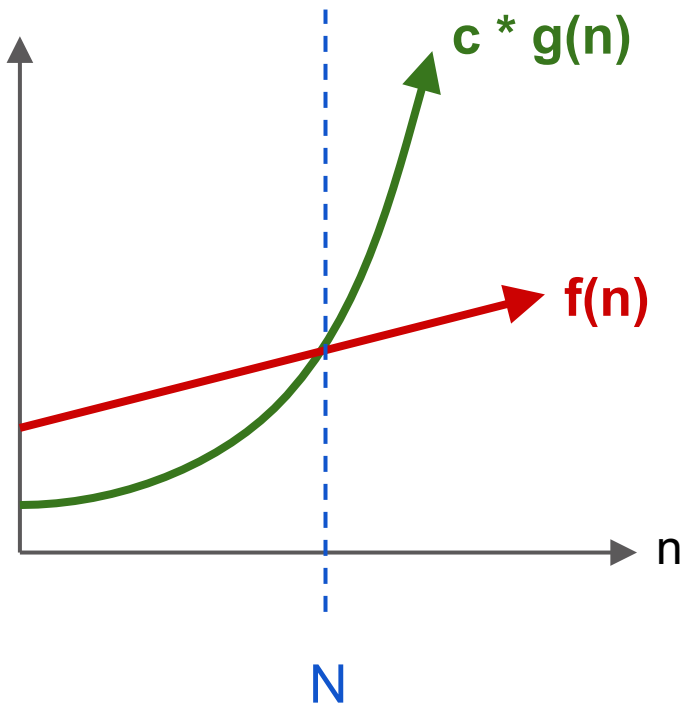
Big O definition

f(n) is O(g(n))

iff

There is a positive constant c
and a real number N such that:

f(n) ≤ c * g(n) for n ≥ N



Is merge sort $O(n^3)$?

Yes, but not tightest upper bound

Review: Big O

Is used to classify algorithms by how they respond to changes in input size n.

Important vocabulary:

- Constant time: O(1)
- Logarithmic time: O(log n)
- Linear time: O(n)
- Quadratic time: $O(n^2)$

Let f(n) and g(n) be two functions that tell how many statements two algorithms execute when running on input of size n.

f(n) >= 0 and g(n) >= 0.

Review: Informal Big O rules

1. Usually: $O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$
   – Such as if something that takes g(n) time for each of f(n) repetitions . . .
   (loop within a loop)

2. Usually: $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$
   – "max" is whatever's dominant as n approaches infinity
   – Example: $O((n^2-n)/2) = O((1/2)n^2 + (-1/2)n) = O((1/2)n^2)$
   $= O(n^2)$

3. Why don't logarithm bases matter?
   –For constants x, y: $O(\log_x n) = O((\log_x y)(\log_y n))$
   –Since $(\log_x y)$ is a constant, $O(\log_x n) = O(\log_y n)$

Test will not require understanding such rules for logarithms

Review: Big O

1. $\log(n) + 20$ is $O(\log(n))$ (logarithmic)
2. $n + \log(n)$ is $O(n)$ (linear)
3. $n/2$ and $3*n$ are $O(n)$
4. $n * \log(n) + n$ is $n * \log(n)$
5. $n^2 + 2*n + 6$ is $O(n^2)$ (quadratic)
6. $n^3 + n^2$ is $O(n^3)$ (cubic)
7. $2^n + n5$ is $O(2^n)$ (exponential)

Review: Big O examples

1. What is the runtime of an algorithm that runs insertion sort on an array $O(n^2)$ and then runs binary search O(log n) on that now sorted array?

1. What is the runtime of finding and removing the fifth element from a linked list? What if in the middle of that remove operation we swapped two integers exactly 100000 times, what is the runtime now?

1. What is the runtime of running merge sort 4 times? *n* times?

Other topics to know

Refer to study guide :)

Spanning Trees NOT ON TEST

Two approaches. A spanning tree is a:

- max set of edges with no cycles

  Algorithm:

  Repeat until no longer possible:

  Find a cycle and delete one of its edges

- minimal set of edges that connect all nodes

  Algorithm:

  Start with all nodes (each one is a component), no edges.

  Repeat until no longer possible:

  Add an edge that connects two unconnected components.

Kruskal's minimum spanning tree algorithm

Definition: minimal set of edges that connect all nodes

Algorithm:

Start with all nodes (each one is a component), no edges.
Repeat until no longer possible:
Add an edge that connects two unconnected components.

Kruskal says to choose an edge with minimum weight

Prim's minimum spanning tree algorithm

Definition: minimal set of edges that connect all nodes

Algorithm:

Start with all nodes (each one is a component), no edges.
Repeat until no longer possible:
Add an edge that connects two unconnected components.

Prim says

(1) In the first step, choose one node n, arbitrarily.
(2) When adding an edge, choose one with minimum weight that is connected to n.