CSE 250

Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lec 29: Midterm #2 Review

Announcements

- Midterm Friday! (see Piazza post)
- No recitations next week

Course Roadmap

Analysis Tools/Techniques	ADTs	Data Structures
Asymptotic Analysis, (Unqualified) Runtime Bounds		
	Sequence	Array, LinkedList
Amortized Runtime	List	ArrayList, LinkedList
Recursive analysis, divide and conquer, Average/Expected Runtime		
	Stack, Queue	ArrayList, LinkedList
Midterm #1		

Course Roadmap

Analysis Tools/Techniques	ADTs	Data Structures
Review recursive analysis	Graphs, PriorityQueue	EdgeList, AdjacencyList, AdjacencyMatrix
	Trees	BST, AVL Tree, Red-Black Tree, Heaps
Midterm #2		
Review expected runtime	HashTables	
Miscellaneous		

Major Topics

- Graphs
  - What can they represent? How can we implement them? Runtimes?
  - What can we use them for? How do we search them?
- PriorityQueues
  - What can they do? How do we implement? What are the runtimes?
  - What can we use them for and how?
- Trees
  - Heaps
  - BSTs (General BSTs, Balanced BSTs – AVL and Red-Black)

Graphs

A (Directed) Graph ADT

Two type parameters (`Graph[V,E]`)

  `V:` The vertex label type
  `E:` The edge label type

Vertices

  …are elements
  …store a value of type **`V`**

Edges

  …are also elements
  …store a value of type **`E`**

A (Directed) Graph ADT

What can we do with a Graph?

- Iterate through the vertices
- Iterate through the edges
- Add a vertex
- Add an edge
- Remove a vertex
- Remove an edge

A (Directed) Graph ADT

```java
public interface Graph<V, E> {
    public Iterator<Vertex> vertices();
    public Iterator<Edge> edges();
    public Vertex addVertex(V label);
    public Edge addEdge(Vertex orig, Vertex dest, E label);
    public void removeVertex(Vertex vertex);
    public void removeEdge(Edge edge);
}
```

A (Directed) Graph ADT

What can we do with a Vertex?

- Get it's label
- Get the outgoing edges
- Get the incoming edges
- Get all incident edges
- Check if it's adjacent to another Vertex

A (Directed) Graph ADT

What can we do with an Edge?

- Get it's label
- Get the incident vertices

A (Directed) Graph ADT

```
public interface Vertex<V,E> {
  public V getLabel();
  public Iterator<Edge> getOutEdges();
  public Iterator<Edge> getInEdges();
  public Iterator<Edge> getIncidentEdges();
  public boolean hasEdgeTo(Vertex v);
}

public interface Edge<V,E> {
  public Vertex getOrigin();
  public Vertex getDestination();
  public E getLabel();
}
```

Implementation Attempt 1: Edge List

Data Model:

A List of Edges

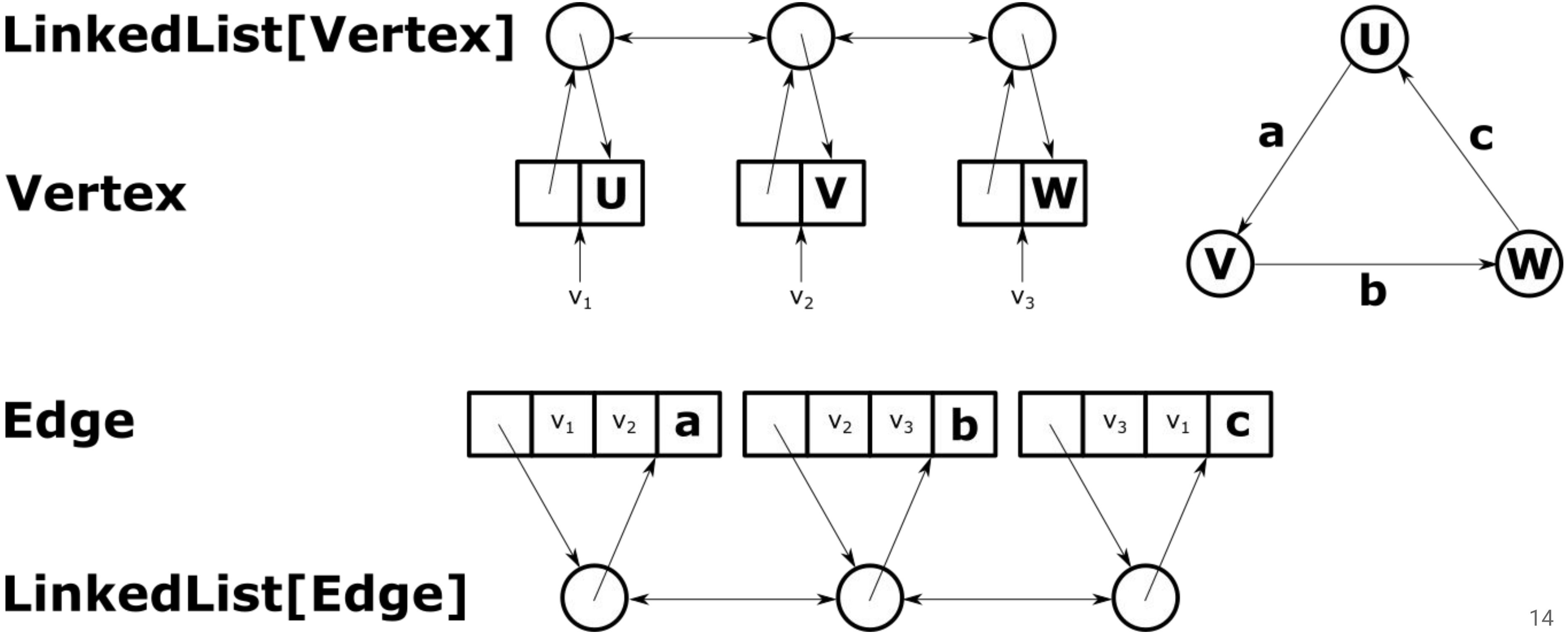
(LinkedList)

A List of Vertices

(LinkedList)

An EdgeList is exactly what it sounds like, a single big list of edges (with a list of vertices as well)

Edge List Summary



Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(m)$
- `vertex.incidentEdges`: $O(m)$
- `vertex.edgeTo`: $O(m)$
- **Space Used:** $O(n) + O(m)$

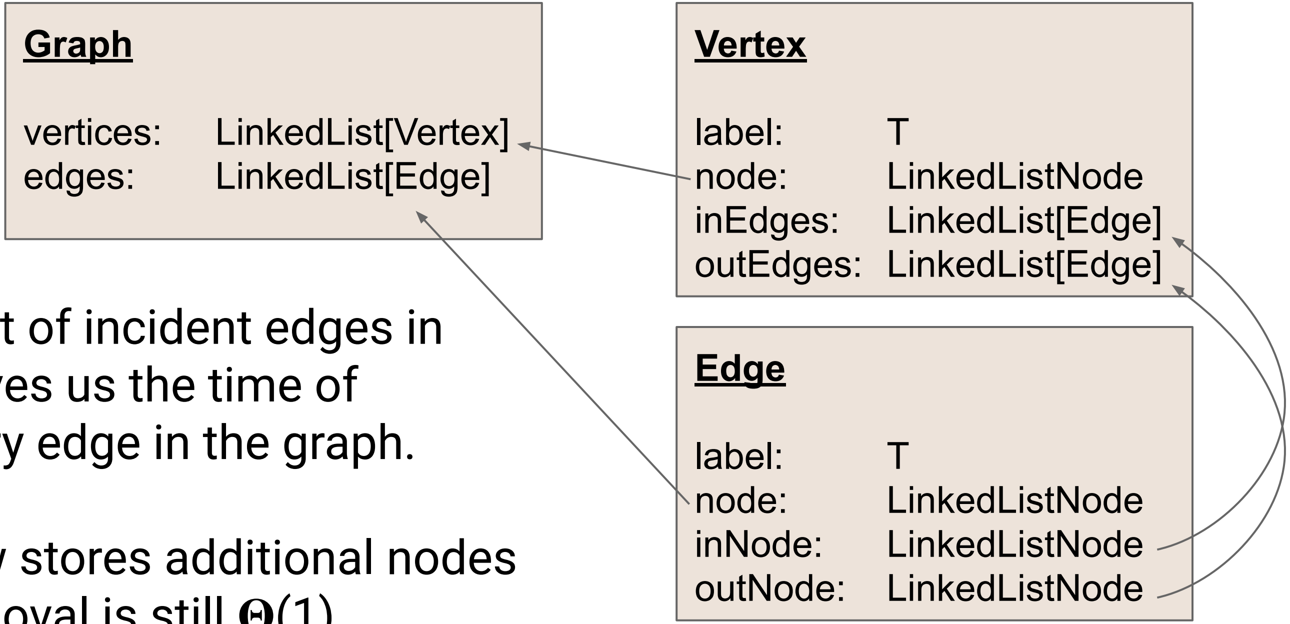
← Involves checking every edge in the graph

How can we improve?

Idea: Store the in/out edges for each vertex!

(Called an adjacency list)

Adjacency List Summary



Graph

vertices: LinkedList[Vertex]
edges: LinkedList[Edge]

Vertex

label: T
node: LinkedListNode
inEdges: LinkedList[Edge]
outEdges: LinkedList[Edge]

Edge

label: T
node: LinkedListNode
inNode: LinkedListNode
outNode: LinkedListNode

Storing the list of incident edges in the vertex saves us the time of checking every edge in the graph.

The edge now stores additional nodes to ensure removal is still $\Theta(1)$

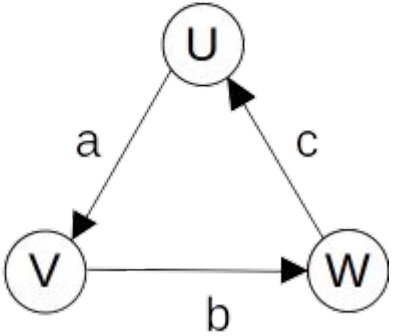
Adjacency List Summary

- `addEdge, addVertex`: $\Theta(1)$
- `removeEdge`: $\Theta(1)$
- `removeVertex`: $\Theta(\text{deg}(\text{vertex}))$
- `vertex.incidentEdges`: $\Theta(\text{deg}(\text{vertex}))$
- `vertex.edgeTo`: $\Theta(\text{deg}(\text{vertex}))$
- **Space Used:** $\Theta(n) + \Theta(m)$

Now we already know what edges are incident without having to check them all

Adjacency Matrix

		Destination		
		U	V	W
Origin	U	***-***	***a***	***-***
	V	***-***	***-***	***b***
	W	***c***	***-***	***-***



Adjacency Matrix Summary

- `addEdge, removeEdge`: $\Theta(1)$
- `addVertex, removeVertex`: $\Theta(n^2)$
- `vertex.incidentEdges`: $\Theta(n)$
- `vertex.edgeTo`: $\Theta(1)$
- **Space Used:** $\Theta(n^2)$

Depth-First Search

Primary Goals

- Visit every vertex in graph ***G* = (*V*,*E*)**
- Construct a spanning tree for every connected component
  - **Side Effect:** Compute connected components
  - **Side Effect:** Compute a path between all connected vertices
  - **Side Effect:** Determine if the graph is connected
  - **Side Effect:** Identify cycles
- Complete in time ***O*(|*V*| + |*E*|)**

DFSOne

```java
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED);
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
}}
```

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED); ← Mark the vertex as VISITED (so we'll never try to visit it again)
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
}}
```

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED);
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
}}
```

Check every outgoing edge (every possible way we could leave the current vertex)

DFSOne

```java
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED);
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
}}
```

Follow the unexplored edges

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED);
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
}}
```

If it leads to an unexplored vertex, then it is a spanning edge. Recursively explore that vertex.

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED);
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
}}
```

Otherwise, we just found a cycle

Depth-First Search Complexity

In summary…

1.	Mark the vertices **`UNVISITED`**	$O(\lvert V \rvert)$
2.	Mark the edges **`UNVISITED`**	$O(\lvert E \rvert)$
3.	**`DFS`** vertex loop	$O(\lvert V \rvert)$ **iterations**
4.	All calls to **`DFSOne`**	$O(\lvert E \rvert)$ **total**

$$O(|V| + |E|)$$

We can also implement DFS without recursion by using a Stack!

Breadth-First Search

Primary Goals

- Visit every vertex in graph ***G* = (*V*,*E*)** **in increasing order of distance from the start**
- Construct a spanning tree for every connected component
  - **Side Effect:** Compute connected components
  - **Side Effect:** Compute a path between all connected vertices
  - **Side Effect:** Determine if the graph is connected
  - **Side Effect:** Identify cycles
  - **Side Effect: Identify shortest paths to the starting vertex**
- Complete in time ***O*(|*V*| + |*E*|)**, with memory overhead ***O*(|*V*|)**

```
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.dequeue();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(w);
        } else {
          e.setLabel(CROSS);
        }
}}}}
```

```
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.dequeue();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(w);
        } else {
          e.setLabel(CROSS);
        }
}}}}
```

Use a queue to keep track of what vertices we want to visit (basically a running TODO list)

```
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.dequeue();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(w);
        } else {
          e.setLabel(CROSS);
        }
}}}}
```

Dequeue a vertex from the Queue and check all of it's outgoing edges

```java
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.dequeue();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(w);
        } else {
          e.setLabel(CROSS);
        }
}}}}
```

When we find a new vertex, mark it as VISITED, and add it to our TODO list.

Remember, our TODO list is a Queue (FIFO) so whatever we enqueud first will be the next thing we dequeue (and explore)

```java
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.dequeue();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(w);
        } else {
          e.setLabel(CROSS);
        }
}}}}
```

When doing BFS we label edges that return to visited vertices as CROSS edges

Breadth-First Search Complexity

In summary…

1. Mark the vertices **`UNVISITED`** — ***O*(|*V*|)**
2. Mark the edges **`UNVISITED`** — ***O*(|*E*|)**
3. Add each vertex to the work queue — ***O*(|*V*|)**
4. Process each vertex — ***O*(|*E*|) total**

***O*(|*V*| + |*E*|)**

Djikstra's Algorithm

- Both BFS and DFS search the whole graph
    - DFS – Exploration order based on a Stack (LIFO)
    - BDS – Exploration order based on a Queue (FIFO)
    - The paths BFS finds are the shortest paths **in terms of # of edges**
- Djikstra's Algorithm finds the shortest path in terms of total distance
    - Can't rely on Stack or Queue – need an ADT that orders the vertices

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    if (curr.vertex.label == UNEXPLORED) {
      curr.vertex.setLabel(VISITED);
      for (Edge e : curr.vertex.outEdges) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          todo.add(new TodoEntry(w, curr.weight + e.weight));
        }
      }
    }
  }
}
```

Create a new PriorityQueue and insert the starting point with a distance of 0

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    if (curr.vertex.label == UNEXPLORED) {
      curr.vertex.setLabel(VISITED);
      for (Edge e : curr.vertex.outEdges) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          todo.add(new TodoEntry(w, curr.weight + e.weight));
        }
      }
    }
  }
}
```

When we pull something out of the PriorityQueue, if it is still UNEXPLORED then we just found the shortest path to that vertex, and we can mark it as VISITED

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    if (curr.vertex.label == UNEXPLORED) {
      curr.vertex.setLabel(VISITED);
      for (Edge e : curr.vertex.outEdges) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          todo.add(new TodoEntry(w, curr.weight + e.weight));
        }
      }
    }
  }
}
```

Add each unexplored neighbor to the PriorityQueue.
Set it's distance equal to our current distance plus the weight of the edge to get to the neighbor.

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    if (curr.vertex.label == UNEXPLORED) {
      curr.vertex.setLabel(VISITED);
      for (Edge e : curr.vertex.outEdges) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          todo.add(new TodoEntry(w, curr.weight + e.weight));
        }
      }
    }
  }
}
```

What is the complexity?

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    if (curr.vertex.label == UNEXPLORED) {
      curr.vertex.setLabel(VISITED);
      for (Edge e : curr.vertex.outEdges) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          todo.add(new TodoEntry(w, curr.weight + e.weight));
        }
      }
    }
  }
}
```

We know removal from a PriorityQueue is ***O*(log(`todo.size()`))**

How big can **`todo`** get?

What is the complexity?

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    if (curr.vertex.label == UNEXPLORED) {
      curr.vertex.setLabel(VISITED);
      for (Edge e : curr.vertex.outEdges) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          todo.add(new TodoEntry(w, curr.weight + e.weight));
        }
      }
    }
  }
}
```

We know removal from a PriorityQueue is ***O*(log(`todo.size()`))**

How big can **todo** get? |E|

Each vertex may be added once per incoming edge. So the size of the PriorityQueue can get as large as |E|

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    if (curr.vertex.label == UNEXPLORED) {
      curr.vertex.setLabel(VISITED);
      for (Edge e : curr.vertex.outEdges) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          todo.add(new TodoEntry(w, curr.weight + e.weight));
        }
      }
    }
  }
}
```

We know removal from a PriorityQueue is ***O*****(log(`todo.size()`))**

How big can **`todo`** get? |E|

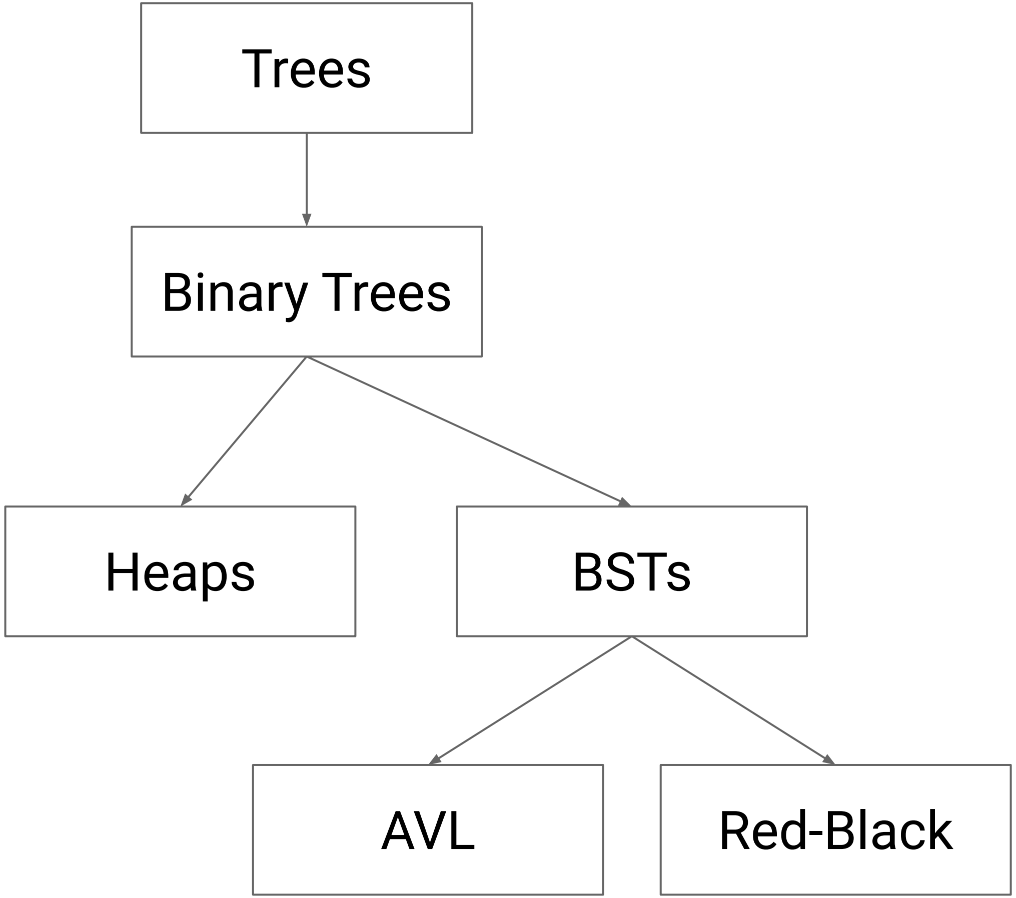
Label the |V| vertices

|E| adds/removes to the PriorityQueue

What is the complexity? O(|V| + |E| log(|E|))

Trees

Types of Trees Covered



Binary Min Heaps

Organize our priority queue as a directed tree

Directed: A directed edge from ***a*** to ***b*** means that $\boxed{\boldsymbol{a \leq b}}$

A max heap would reverse this ordering

Binary: Max out-degree of 2 (easy to reason about)

Complete: Every "level" except the last is full (from left to right)

Balanced: TBD (basically, all leaves are roughly at the same level)

This makes it easy to encode into an array (later today)

Binary Min Heaps

Organize our priority queue as a directed tree

Directed: A directed edge from ***a*** to ***b*** means that ***a ≤ b***

A max heap would reverse this ordering

Binary: Max c out)

Complete: Ev om left to right)

Balanced: TBD (basically, all leaves are roughly at the same level)

If what we are storing in the Heap does not have a default ordering, we must tell Java how to order the items!!

This makes it easy to encode into an array (later today)

The `MinHeap` ADT

`void pushHeap(T value)`
Place an item into the heap

`T popHeap()`
Remove and return the minimal element from the heap

`T peek()`
Peek at the minimal element in the heap

`int size()`
The number of elements in the heap

`pushHeap`

Idea: Insert the element at the next available spot, then fix the heap.

1. Call the insertion point **`current`**
2. While **`current != root`** and **`current < parent`**
   a. Swap **`current`** with **`parent`**
   b. Set **`current = parent`**

What is the complexity (or how many swaps occur)? **O(log(*n*))**

popHeap

Idea: Replace root with the last element then fix the heap

1. Start with **`current = root`**
2. While **`current`** has a **`child < current`**
    a. Swap **`current`** with its smallest **`child`**
    b. Set **`current = child`**

What is the complexity (or how many swaps occur)? ***O*(log(*n*))**

Priority Queues

Operation	**Lazy**	**Proactive**	**Heap**
add	$O(1)$	$O(n)$	$O(\log(n))$
poll	$O(n)$	$O(1)$	$O(\log(n))$
peek	$O(n)$	$O(1)$	$O(1)$

Storing heaps

Notice that:

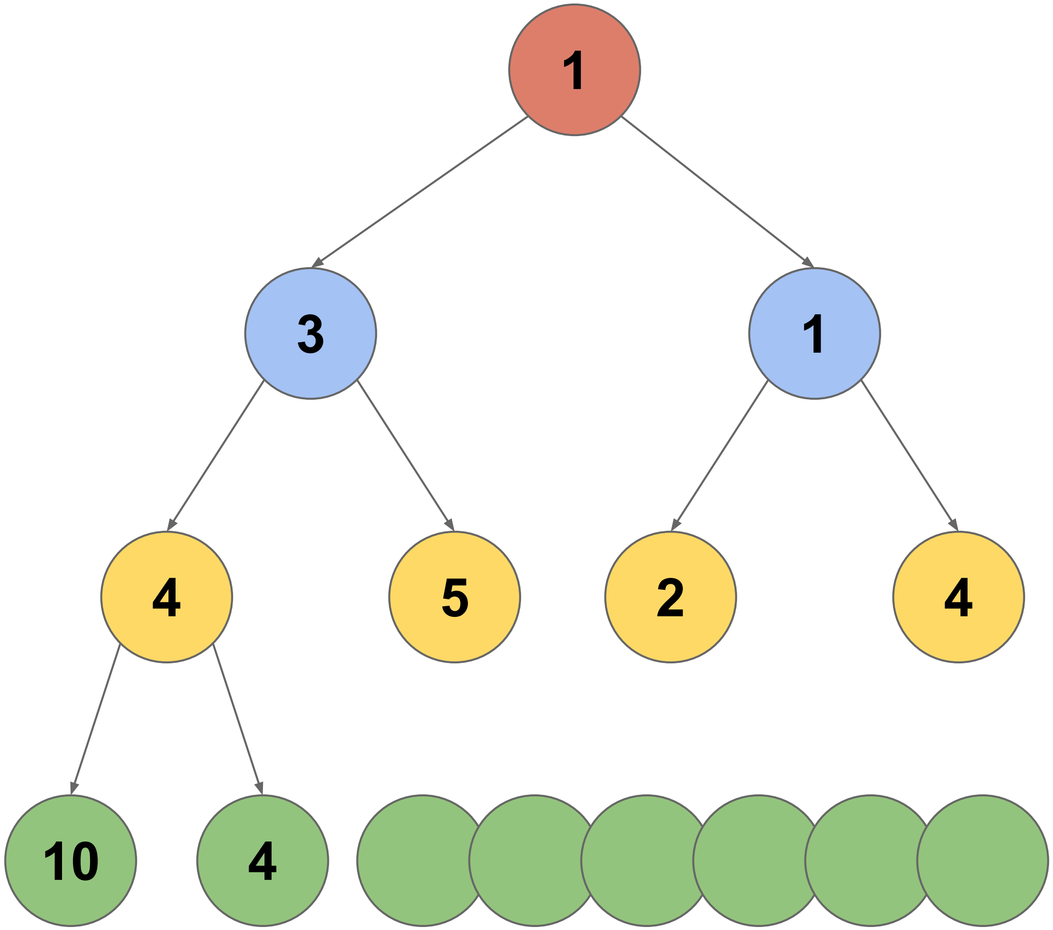
1. Each level has a maximum size
2. Each level grows left-to-right
3. Only the last layer grows

How can we compactly store a heap?

Idea: Use an **`ArrayList`**

Storing Heaps

How can we store this heap in an array buffer?



1	3	1	4	5	2	4	10	4						

Heapify

Input: Array

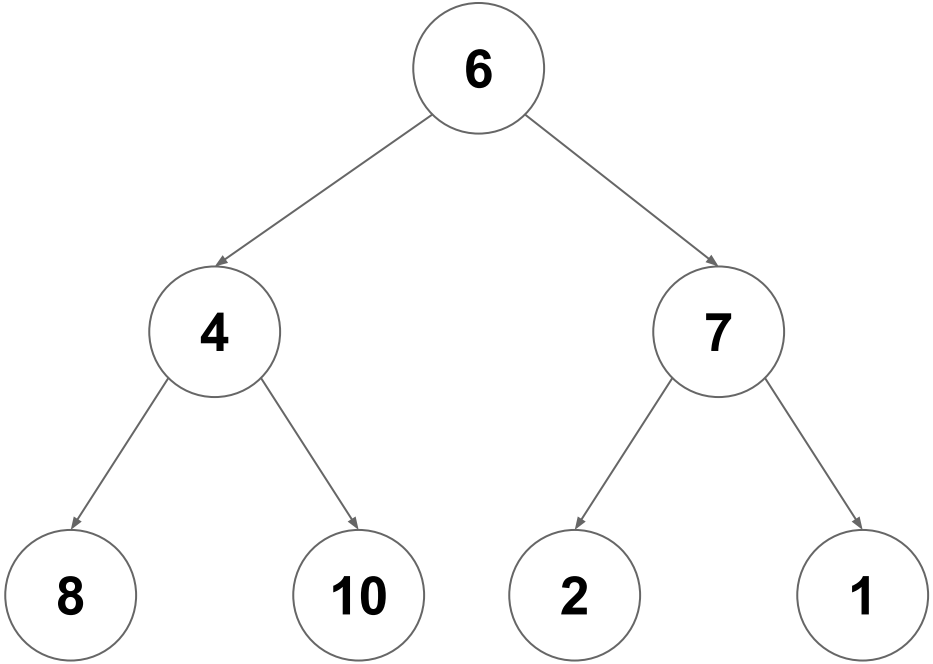
Output: Array re-ordered to be a heap

Idea: **`fixUp`** or **`fixDown`** all ***n*** elements in the array

Given the cost of **`fixUp`** *and* **`fixDown`** *what do we expect the complexity* **`Heapify`** *will be?*

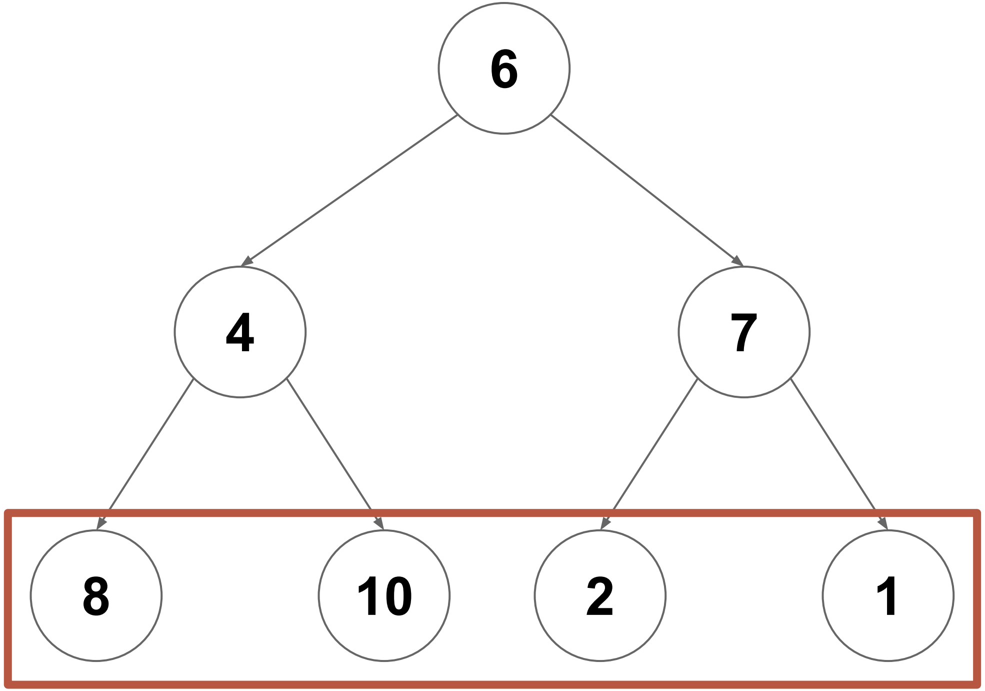
Heapify

Given an arbitrary array (shown as a tree here) turn it into a heap



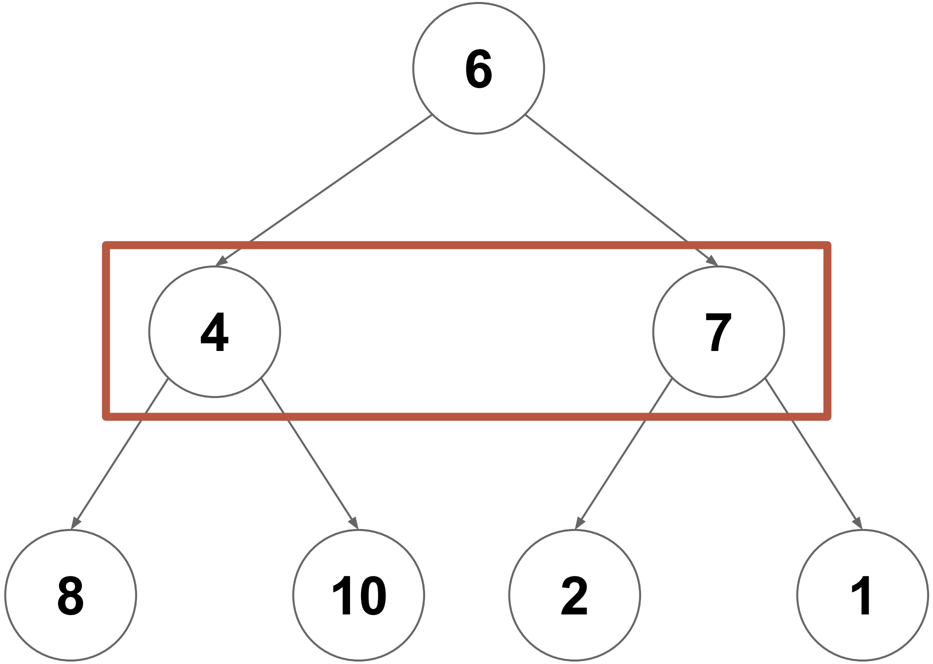
Heapify

Start at the lowest level, and call **`fixDown`** on each node (0 swaps per node)



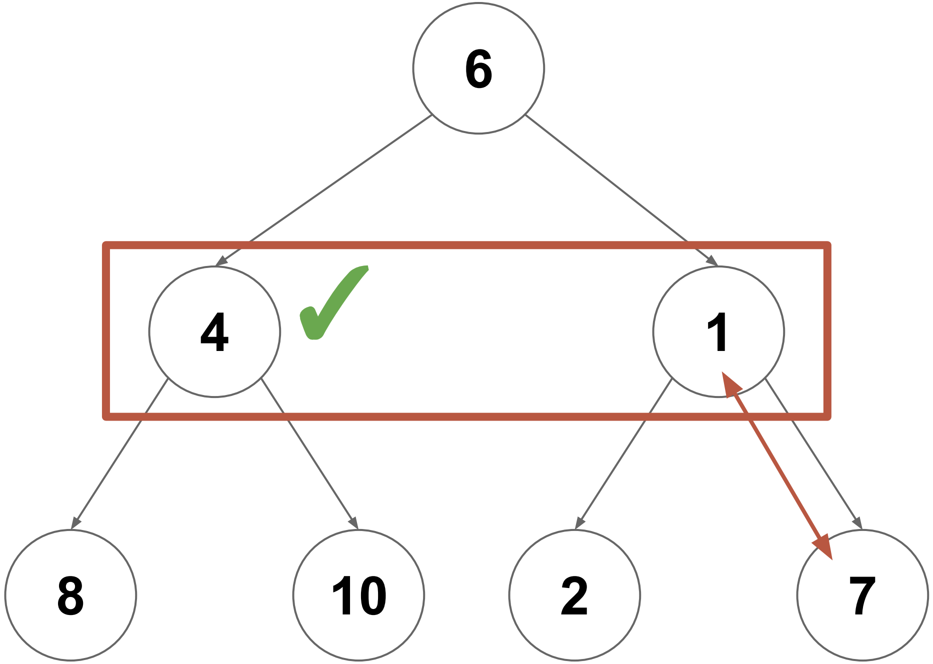
Heapify

Do the same at the next lowest level (at most one swap per node)



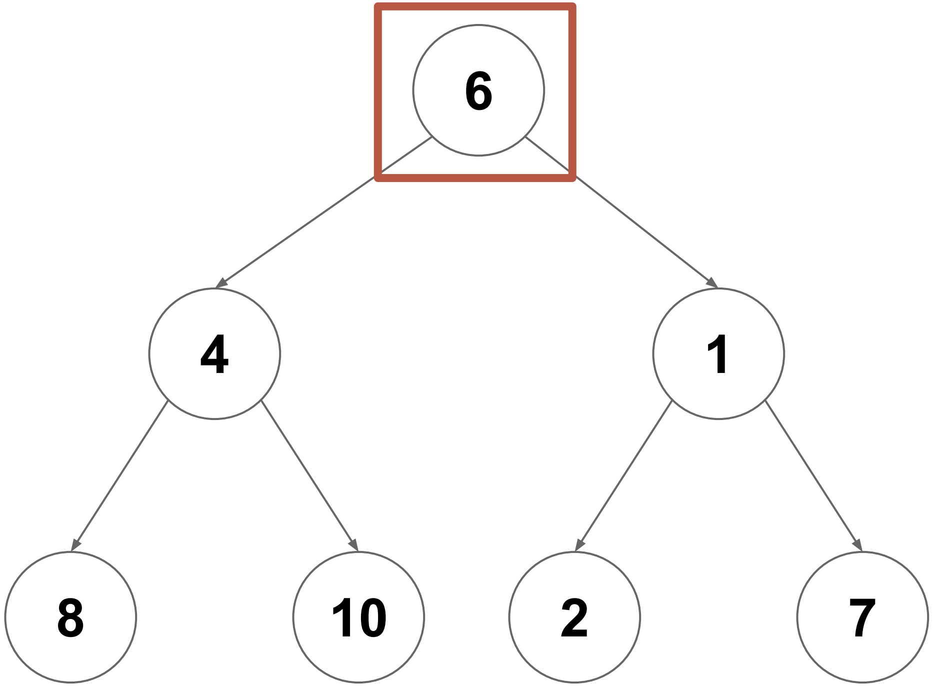
Heapify

Do the same at the next lowest level (at most one swap per node)



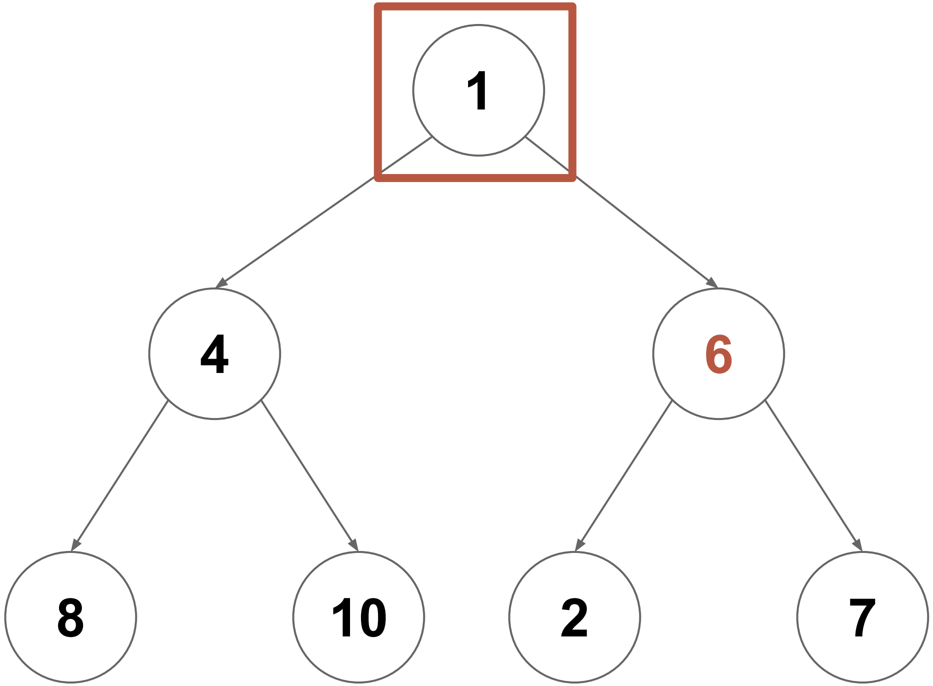
Heapify

Continue upwards (now at most 2 swaps per node)



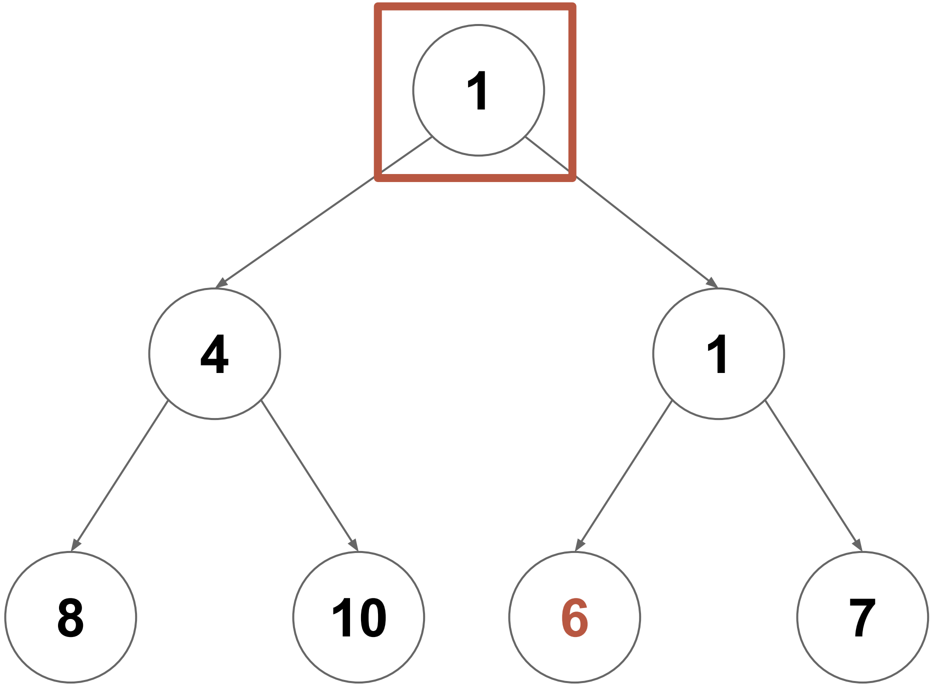
Heapify

Continue upwards (now at most 2 swaps per node)



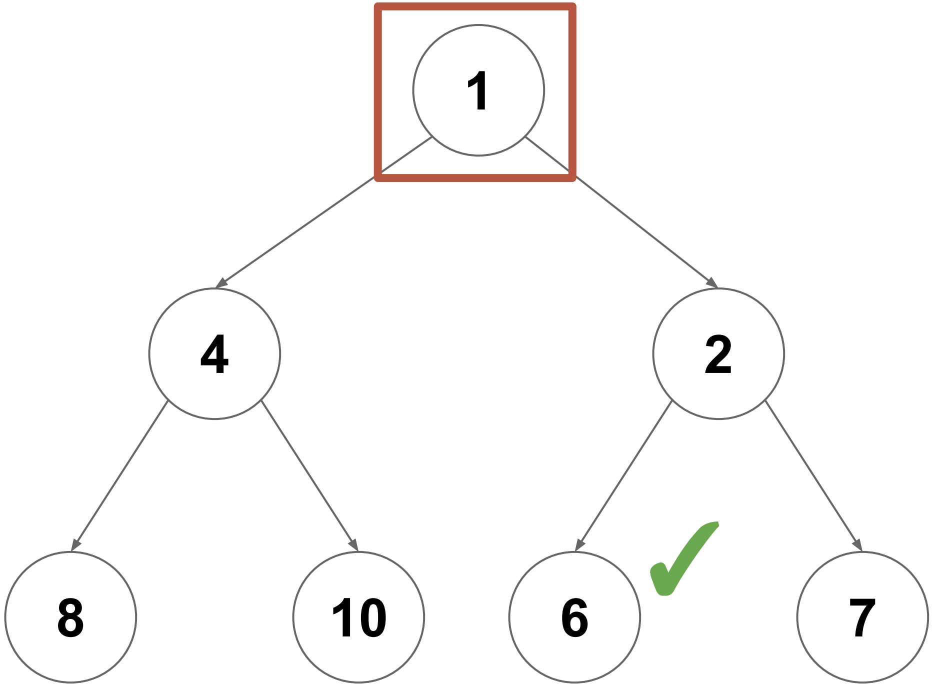
Heapify

Continue upwards (now at most 2 swaps per node)



Heapify

Continue upwards (now at most 2 swaps per node)



Heapify

Therefore we can heapify an array of size *n* in *O*(*n*)

(but heap sort still requires *n* log(*n*) due to dequeue costs)

$$O\left(\sum_{i=1}^{\log(n)} \frac{n}{2^i} \cdot (i+1)\right)$$

$$O\left(n \sum_{i=1}^{\log(n)} \frac{i}{2^i} + \frac{1}{2^i}\right)$$

$$O\left(n \sum_{i=1}^{\log(n)} \frac{i}{2^i}\right)$$

$$O\left(n \sum_{i=1}^{\infty} \frac{i}{2^i}\right) = O(n)$$

Binary Search Tree

A **Binary Search Tree** is a **Binary T** nique key, and the

If what we are storing in the BST does not have a default ordering, we must tell Java how to order the items!!

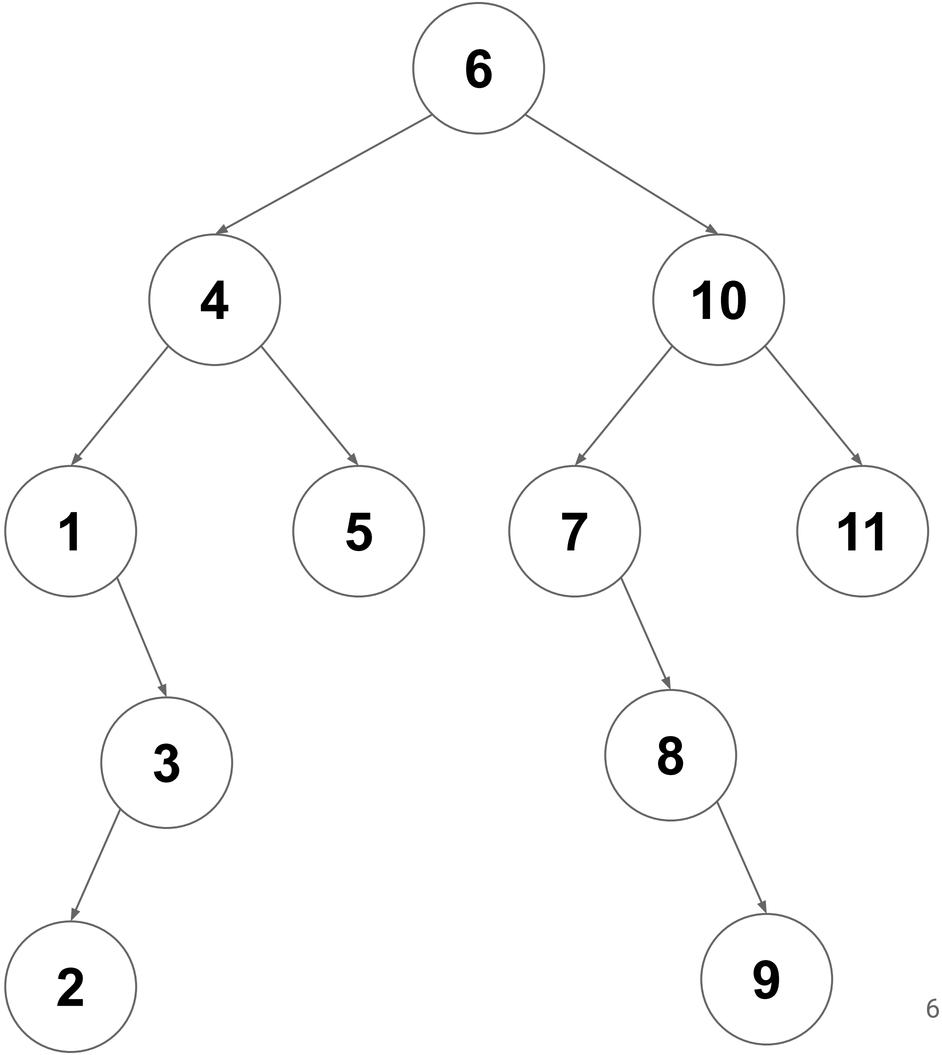
Constraints

- No duplicate keys
- For every node X_L in the left subtree of node X: X_L**.key** < X**.key**
- For every node X_R in the right subtree of node X: X_R**.key** > X**.key**

X **partitions** its children

Is this a valid BST?

Yes!



Finding an Item

Goal: Find an item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (if yes, then the item is not here)
2. Does **`root.value`** have key ***k***? (if yes, done!)
3. Is ***k*** less than **`root.value`**'s key? (if yes, search left subtree)
4. Is ***k*** greater than **`root.value`**'s key? (If yes, search the right subtree)

Inserting an Item

Goal: Insert a new item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (insert here)
2. Does **`root.value`** have key ***k***? (already present! don't insert)
3. Is ***k*** less than **`root.value`**'s key? (call insert on left subtree)
4. Is ***k*** greater than **`root.value`**'s key? (call insert on right subtree)

Removing an Item

Goal: Remove the item with key ***k*** from a BST rooted at **`root`**

1. **`find`** the item
2. Replace the found node with the right subtree
3. Insert the left subtree under the right

BST Operations

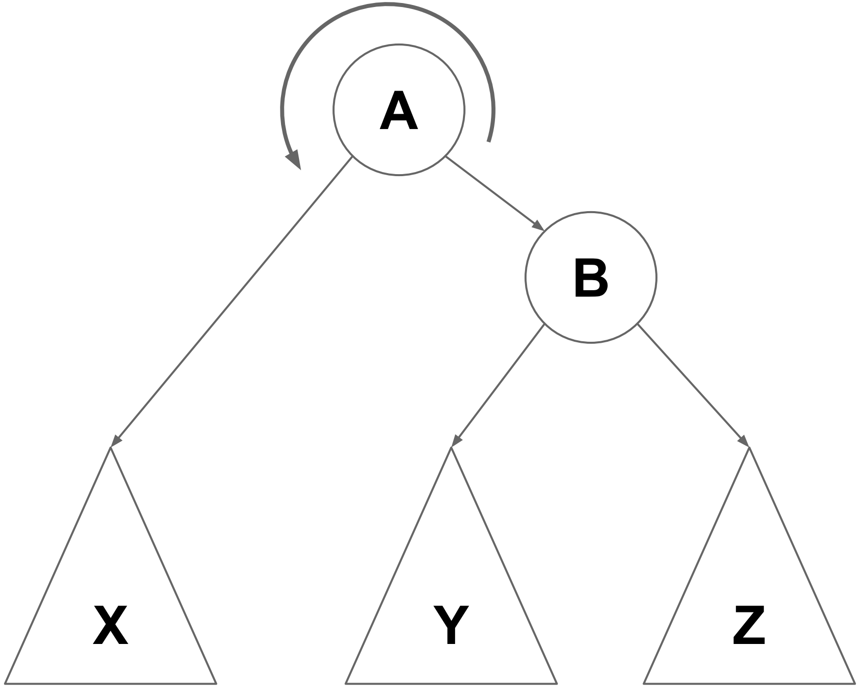
Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

*What is the **d** in terms of **n**?* **$O(n)$**

What about the lower bound? **$\Omega(\log(n))$**

Can we do better? **YES!**

Rebalancing Trees (rotations)



Rotate(A, B)

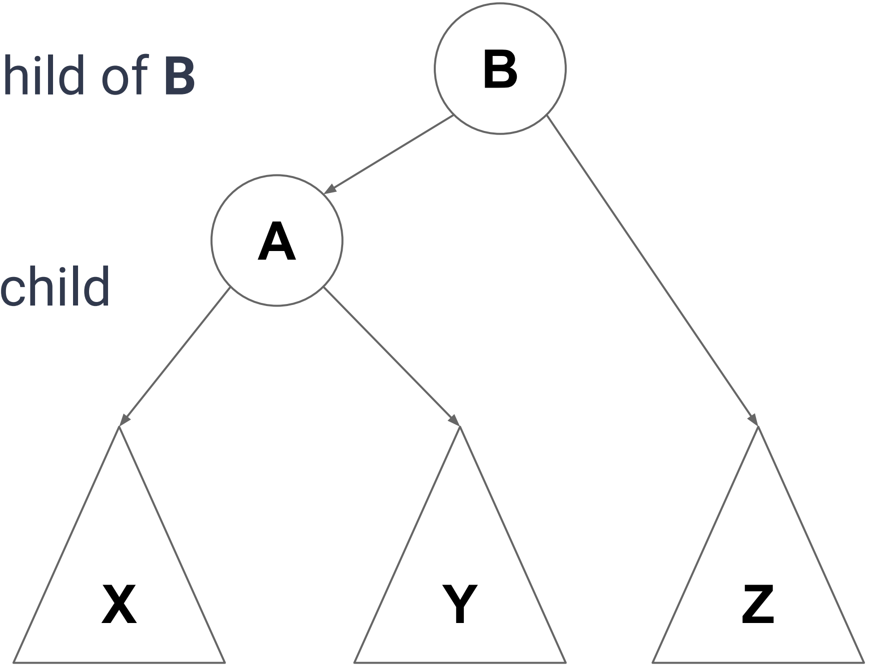
Rebalancing Trees (rotations)

Make **A** the left child of **B**

What about **Y**?

Make it the right child

of **A**



Rotate(A, B)

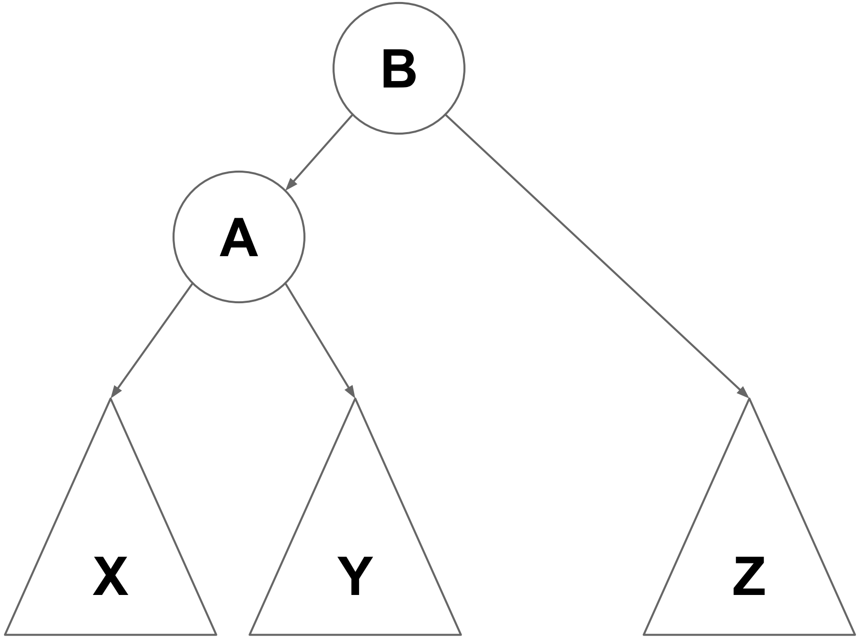
Rebalancing Trees (rotations)

A became **B**'s left child

B's left child became **A**'s right child

Is ordering maintained? **Yes!**

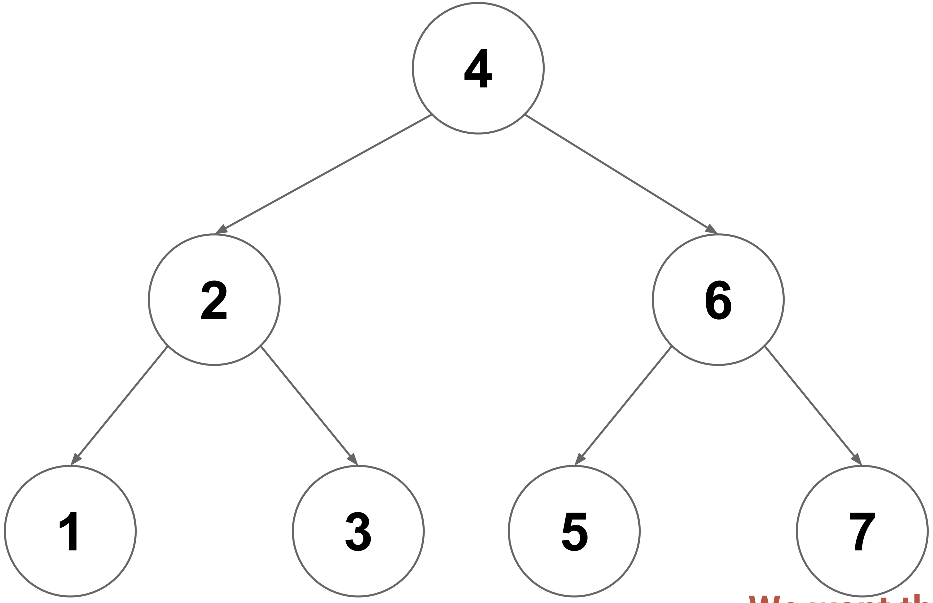
Complexity? ***O*(1)**



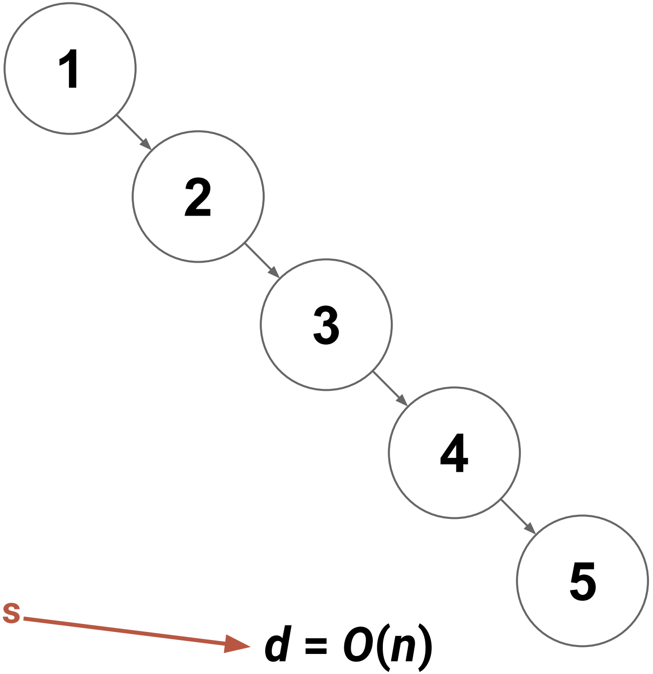
Rotate(A, B)

Tree Depth vs Size

If height(left) ≈ height(right)



If height(left) ≪ height(right)



We want this, not this

$d = O(\log(n))$

$d = O(n)$

AVL Trees

An **AVL tree** (**A**delson-**V**elsky and **L**andis) is a ***BST*** where every subtree is depth-balanced

Remember: Tree depth = height(root)

Balanced: |height(root.right) - height(root.left)| ≤ 1

AVL Trees - Depth Bounds

Question: Does the AVL property result in any guarantees about depth?

YES! Depth balance forces a maximum possible depth of **log(*n*)**

Proof Idea: An AVL tree with depth ***d*** has "enough" nodes

Inserting Records

To insert a record into an AVL Tree:

1. Find the insertion point (remember it is a BST) **$O(d) = O(\log n)$**
2. Insert the new leaf and set balance factor to 0 **$O(1)$**
3. Trace path back up to root and update balance factors **$O(d) = O(\log n)$**
    a. If a balance factor becomes +/-2 then rotate to fix **$O(1)$**

Removing Records

- Removal follows essentially the same process as insertion
    - Do a normal BST removal
    - Go back up the tree adjusting balance factors
    - If you discover a balance factor that goes to +2/-2, rotate to fix

Summary

- We want shallow BSTs (it makes **`find`**, **`insert`**, **`remove`** faster)
- Enforcing AVL constraints makes our BSTs shallow
  - The constraints are |height(right) - height(left)| ≤ 1
  - It will guarantee $\boldsymbol{d = O(\log(n))}$
- Adding/removing from a BST changes height by at most 1
- A rotation can also change a BST height by at most 1
- Therefore after **`insert`**/**`remove`** into an AVL tree, we can reinforce AVL constraints with one (or two) rotations
  - We only need to make one trip back up the tree to do so
  - Therefore **`insert`**/**`remove`** is still $\boldsymbol{O(d) = O(\log(n))}$

Maintaining Balance - Another Approach

Enforcing height-balance is too strict (May do “unnecessary” rotations)

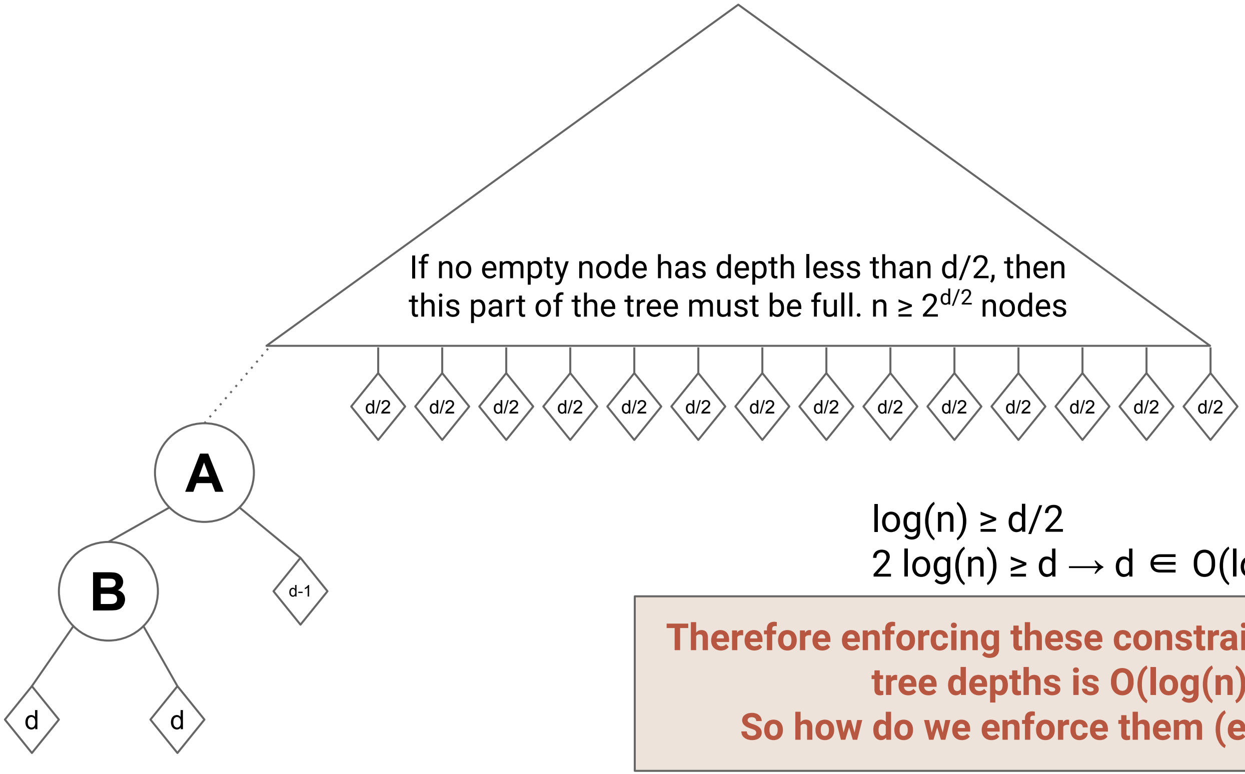
Weaker (and more direct) restriction:

- Balance the depth of empty tree nodes
- If ***a***, ***b*** are EmptyTree nodes, then enforce that for all ***a***, ***b***:
  - depth(***a***) ≥ (depth(***b***) ÷ 2)

    or

  - depth(***b***) ≥ (depth(***a***) ÷ 2)

Depth Balancing



If no empty node has depth less than d/2, then this part of the tree must be full. $n \geq 2^{d/2}$ nodes

$\log(n) \geq d/2$

$2 \log(n) \geq d \rightarrow d \in O(\log(n))$

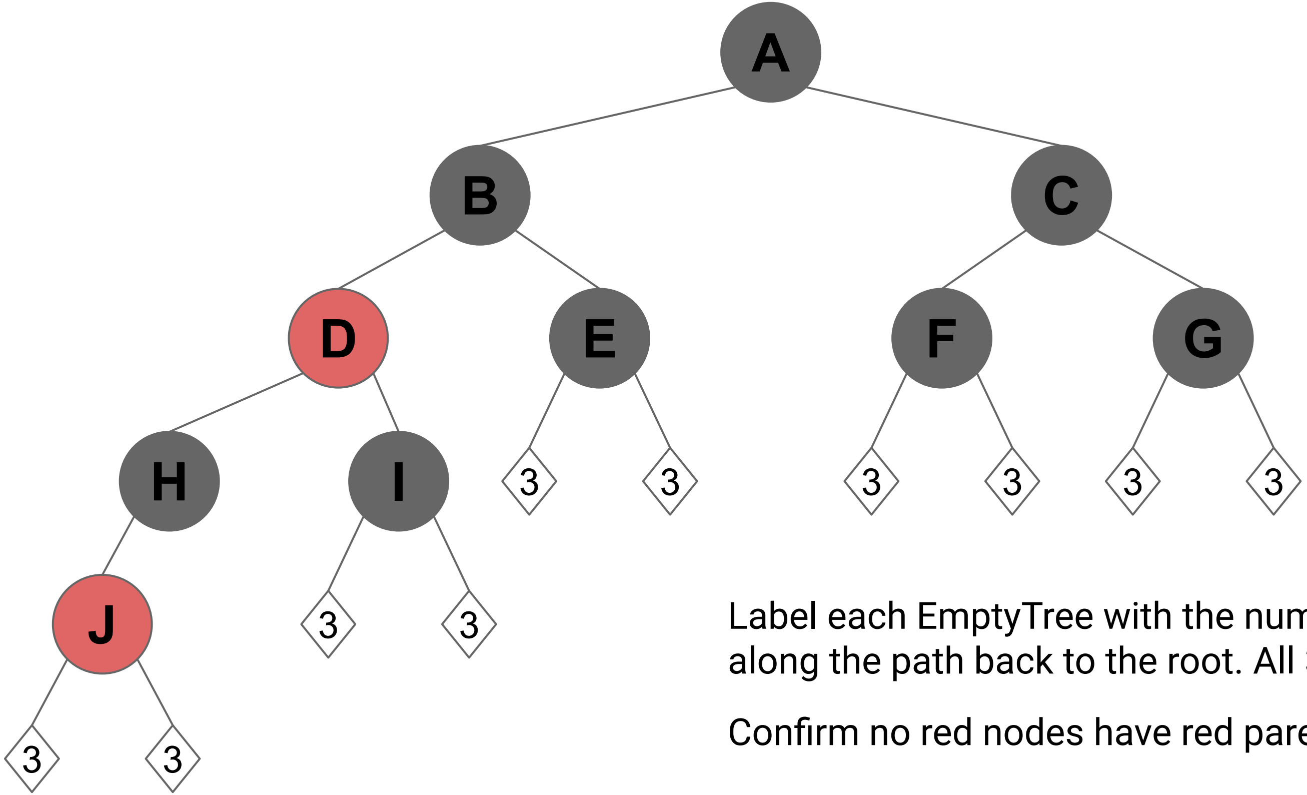
Therefore enforcing these constraints means that tree depths is O(log(n))...
So how do we enforce them (efficiently)?

Red-Black Trees

To Enforce the Depth Constraint on empty nodes:

1. Color each node red or black
    a. The # of black nodes from each empty node to root must be same
    b. The parent of a red node must always be black
2. On insertion (or deletion)
    a. Inserted nodes are red (won't break 1a)
    b. Repair violations of 1b by rotating and/or recoloring
        i. Make sure repairs don't break 1a

Red-Black Trees



Label each EmptyTree with the number of black nodes along the path back to the root. All 3 in this case ✓

Confirm no red nodes have red parents ✓

Red-Black Tree

Note: Each insertion creates at most one red-red parent-child conflict

- O(1) time to recolor/rotate to repair the parent-child conflict
- May create a red-red conflict in grandparent
  - Up to d/2 = O(log(n)) repairs required, but each repair is O(1)
- **Insertion therefore remains O(log(n))**

Note: Each deletion removes at most one black node (red doesn't matter)

- O(1) time to recolor/rotate to preserve black-depth
- May require recoloring (grand-)parent from black to red
  - Up to d = O(log(n)) repairs required
- **Deletion therefore remains O(log(n))**

BST Operations

Operation	BST	AVL	Red-Black
`find`	$O(d) = O(n)$	$O(d) = O(\log n)$	$O(d) = O(\log n)$
`insert`	$O(d) = O(n)$	$O(d) = O(\log n)$	$O(d) = O(\log n)$
`remove`	$O(d) = O(n)$	$O(d) = O(\log n)$	$O(d) = O(\log n)$

The tree operations on a BST are always **$O(d)$** (they involve a constant number of trips from root to leaf at most).

The balanced varieties (AVL and Red-Black) constrain the depth

Misc

Sets

A **Set** is an **unordered** collection of **unique** elements.

(order doesn't matter, and at most one copy of each ~~item~~ key)

The Set ADT

`void add(T element)`

Store one copy of **`element`** if not already present

`boolean contains(T element)`

Return true if **`element`** is present in the set

`boolean remove(T element)`

Remove **`element`** if present, or return false if not

Bags

A **Bag** is an **unordered** collection of **non-unique** elements.

(order doesn't matter, and multiple copies with the same key is OK)

The Bag ADT

`void add(T element)`

Store one copy of **`element`**

`int contains(T element)`

Return the number of copies of **`element`** in the bag

`boolean remove(T element)`

Remove one copy of **`element`** if present, or return false if not

Note: Sometimes referred to as multiset. Java does not have a native Bag/Multiset class.

Implementing Sets/Bags

	`add`	`contains`	`remove`
ArrayList	$O(n)$	$O(n)$	$O(n)$
LinkedList	$O(n)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(\log(n))$	$O(n)$
Sorted LinkedList	$O(n)$	$O(n)$	$O(n)$

Implementing Sets/Bags

	`add`	`contains`	`remove`
ArrayList	$O(n)$	$O(n)$	$O(n)$
			$O(n)$
Sorted ArrayList	$O(n)$	$O(\log(n))$	$O(n)$
Sorted LinkedList	$O(n)$	$O(n)$	$O(n)$

How would our implementations and runtimes look if we implemented Sets and Bags with Trees?

Implementing Sets/Bags

	`add`	`contains`	`remove`
BST	$O(n)$	$O(n)$	$O(n)$
AVL Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$
Red-Black Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$

Implementing Sets/Bags

	`add`	`contains`	`remove`
BST	$O(n)$	$O(n)$	$O(n)$
			$O(\log n)$
Red-Black Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$

What about Bags? How can we allow duplicates in our BSTs?

Implementing Sets/Bags

	`add`	`contains`	`remove`
BST	$O(n)$	$O(n)$	$O(n)$
			$O(\log n)$
Re			$O(\log n)$

What about Bags? How can we allow duplicates in our BSTs?

Option 1: Put ≤ to the left instead of just <
Option 2: Store duplicates in the same node (like in PA1)