CSE 12:
Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Twelve
22 Aug 2011

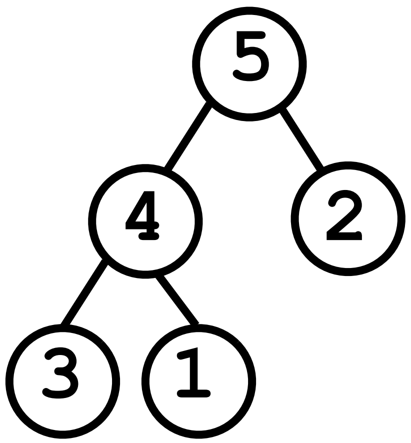
Heaps, continued.

Review from last lecture

- A *heap* is a *complete binary tree* whose last level of nodes is filled left-to-right *and* which satisfies the *heap condition*.
- Heap condition:
  - The root of every sub-tree is *no smaller than any node in the sub-tree*. (For *max*-heap).
- The heap condition ensures that the *largest* element is always stored at the root:
  - $O(1)$ time-cost for **`findLargest`**
  - $O(\log n)$ time-cost for **`removeLargest`**

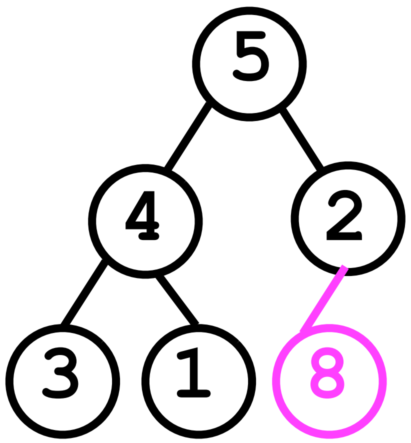
Adding to a heap

- To add a new object o to the heap:
  - Create a new node *n* containing *o*, and add *n* to the last level of the tree (at the left-most position).
    - This may violate the heap condition.
  - Repeatedly "bubble up" n towards the root whenever *n* > parent(*n*).



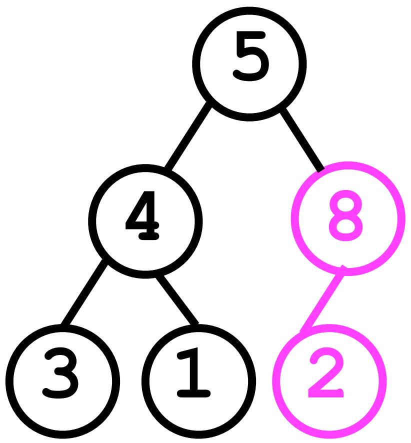
Adding to a heap

- To add a new object o to the heap:
  - Create a new node *n* containing *o*, and add *n* to the last level of the tree (at the left-most position).
    - This may violate the heap condition.
  - Repeatedly "bubble up" n towards the root whenever *n* > parent(*n*).



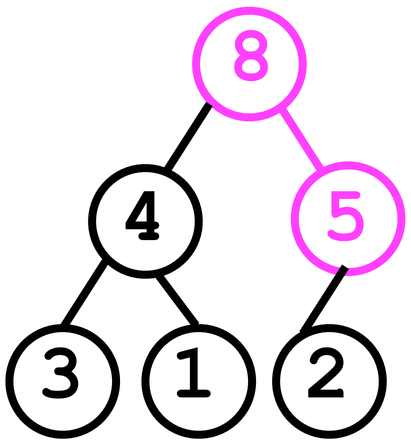
Adding to a heap

- To add a new object o to the heap:
  - Create a new node *n* containing *o*, and add *n* to the last level of the tree (at the left-most position).
    - This may violate the heap condition.
  - Repeatedly "bubble up" n towards the root whenever *n* > parent(*n*).



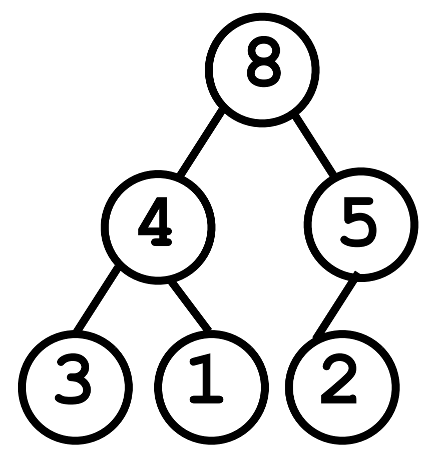
Adding to a heap

- To add a new object o to the heap:
  - Create a new node *n* containing *o*, and add *n* to the last level of the tree (at the left-most position).
    - This may violate the heap condition.
  - Repeatedly "bubble up" n towards the root whenever *n* > parent(*n*).



Adding to a heap

- To add a new object o to the heap:
  - Create a new node *n* containing *o*, and add *n* to the last level of the tree (at the left-most position).
    - This may violate the heap condition.
  - Repeatedly "bubble up" n towards the root whenever *n* > parent(*n*).



The tree is now a valid heap again.

Removing the *largest* element from a heap

- The largest element is always stored at the top of the heap.
  - Hence, just remove the *root*.
- We must then *replace* it with something.
  - Remove the last node *n* in the heap (right-most child of last level) and make it the new root of the tree.
    - This may violate the heap condition.
    - We will then have to recursively swap *n* with one of its children (i.e., back down the tree) until the heap condition is restored. This is called "trickling down".

Removing the *largest* element from a heap

```
void removeLargest () {
  _nodeArray[0] = _nodeArray[_numNodes - 1];
  _numNodes--;
  trickleDown(0);
}

void trickleDown (int index) {
  While node at index is less than one of its children:
    Swap node with the larger child node.
    index = largerChild(index);
}
```

Iterative implementation

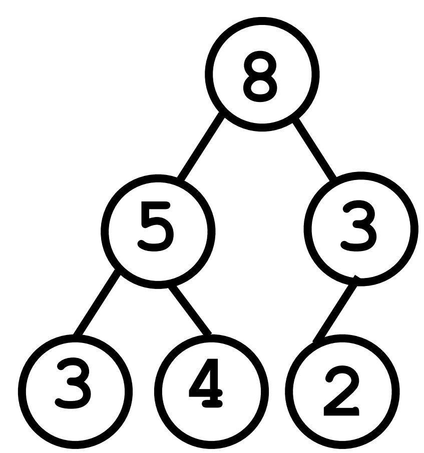
or

```
void trickleDown (int index) {
  If node at index is less than one of its children:
    Swap node with the larger child node.
    trickleDown(largerChild(index));
}
```

Recursive implementation

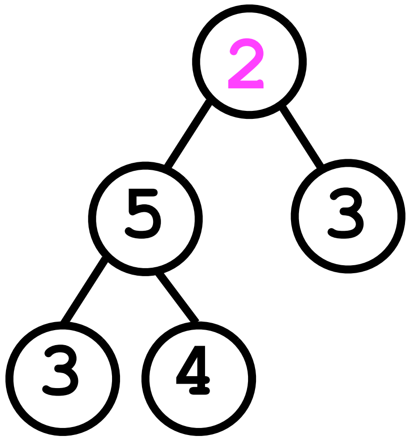
Removing the *largest* element from a heap

```
void removeLargest () {
  _nodeArray[0] = _nodeArray[_numNodes - 1];
  _numNodes--;
  trickleDown(0);
}

void trickleDown (int index) {
  While node at index is less than one of its children:
    Swap node with the larger child node.
    index = largerChild(index);
}
```



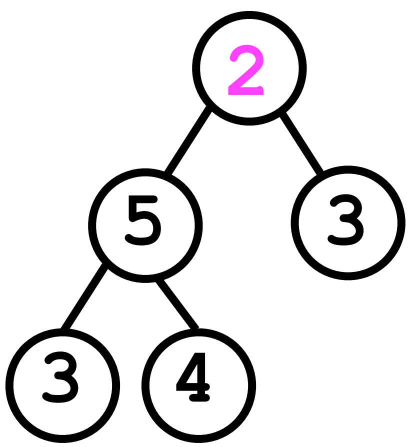
Removing the *largest* element from a heap

```
void removeLargest () {
  _nodeArray[0] = _nodeArray[_numNodes - 1];
  _numNodes--;
  trickleDown(0);
}

void trickleDown (int index) {
  While node at index is less than one of its children:
    Swap node with the larger child node.
    index = largerChild(index);
}
```



Removing the *largest* element from a heap

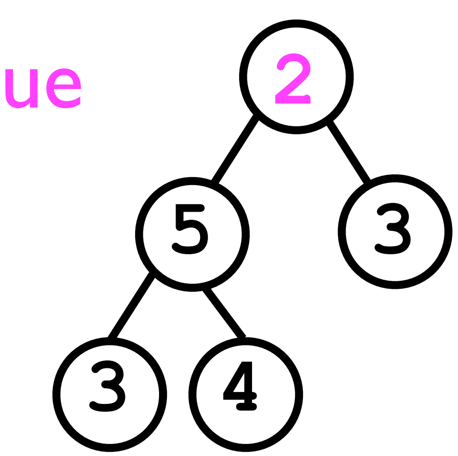
```
void removeLargest () {
  _nodeArray[0] = _nodeArray[_numNodes - 1];
  _numNodes--;
  trickleDown(0);
}

void trickleDown (int index) {
  While node at index is less than one of its children:
    Swap node with the larger child node.
    index = largerChild(index);
}
```



Removing the *largest* element from a heap

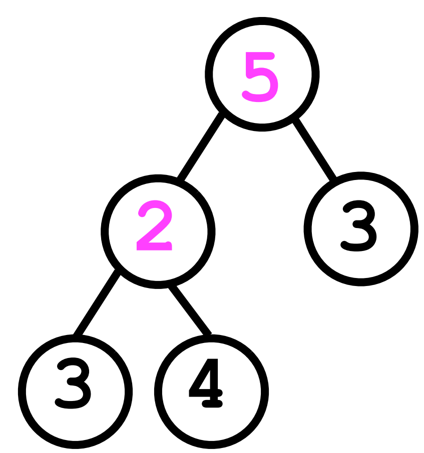
```
void removeLargest () {
  _nodeArray[0] = _nodeArray[_numNodes - 1];
  _numNodes--;
  trickleDown(0);
}

void trickleDown (int index) {
  While node at index is less than one of its children:
    Swap node with the larger child node.
    index = largerChild(index);
}
```

True



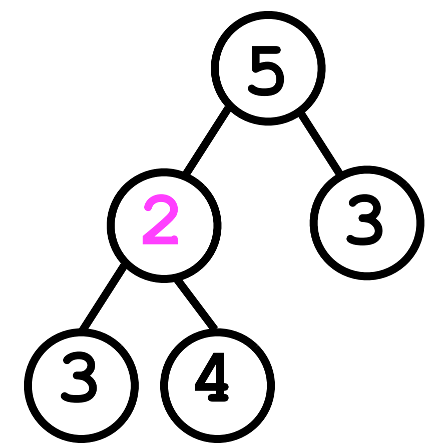
Removing the *largest* element from a heap

```
void removeLargest () {
  _nodeArray[0] = _nodeArray[_numNodes - 1];
  _numNodes--;
  trickleDown(0);
}

void trickleDown (int index) {
  While node at index is less than one of its children:
    Swap node with the larger child node.
    index = largerChild(index);
}
```



Removing the *largest* element from a heap

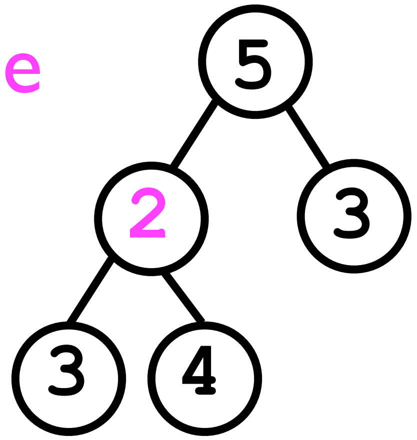
```
void removeLargest () {
  _nodeArray[0] = _nodeArray[_numNodes - 1];
  _numNodes--;
  trickleDown(0);
}

void trickleDown (int index) {
  While node at index is less than one of its children:
    Swap node with the larger child node.
    index = largerChild(index);
}
```



Removing the *largest* element from a heap

```
void removeLargest () {
  _nodeArray[0] = _nodeArray[_numNodes - 1];
  _numNodes--;
  trickleDown(0);
}

void trickleDown (int index) {
  While node at index is less than one of its children:
    Swap node with the larger child node.
    index = largerChild(index);
}
```

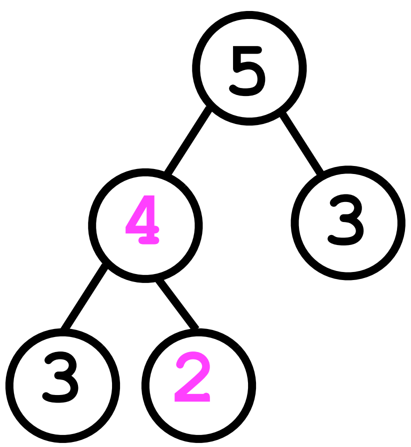
True



Removing the *largest* element from a heap

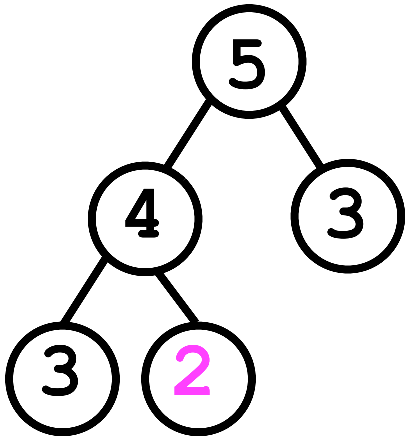
```
void removeLargest () {
  _nodeArray[0] = _nodeArray[_numNodes - 1];
  _numNodes--;
  trickleDown(0);
}

void trickleDown (int index) {
  While node at index is less than one of its children:
    Swap node with the larger child node.
    index = largerChild(index);
}
```

It's crucial we swap with the *larger* child to maintain the heap condition.



Removing the *largest* element from a heap

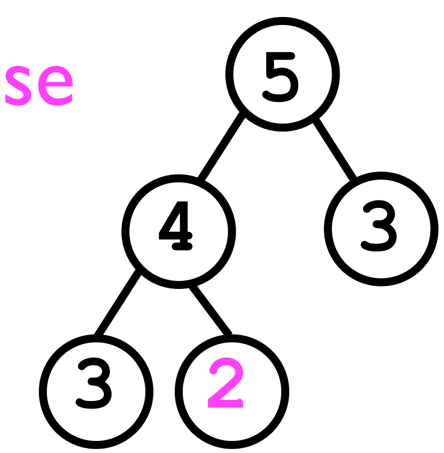
```
void removeLargest () {
  _nodeArray[0] = _nodeArray[_numNodes - 1];
  _numNodes--;
  trickleDown(0);
}

void trickleDown (int index) {
  While node at index is less than one of its children:
    Swap node with the larger child node.
    index = largerChild(index);
}
```



Removing the *largest* element from a heap

```
void removeLargest () {
  _nodeArray[0] = _nodeArray[_numNodes - 1];
  _numNodes--;
  trickleDown(0);
}

void trickleDown (int index) {
  While node at index is less than one of its children:
    Swap node with the larger child node.
    index = largerChild(index);
}
```

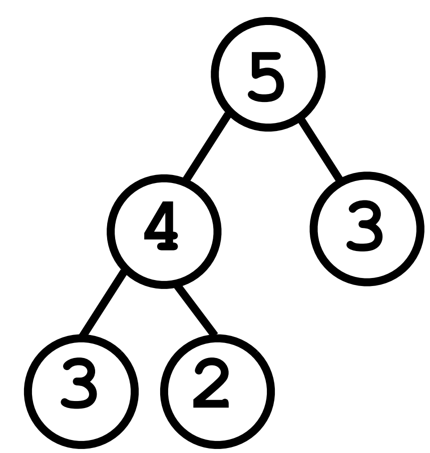
False



Removing the *largest* element from a heap

```
void removeLargest () {
  _nodeArray[0] = _nodeArray[_numNodes - 1];
  _numNodes--;
  trickleDown(0);
}

void trickleDown (int index) {
  While node at index is less than one of its children:
    Swap node with the larger child node.
    index = largerChild(index);
}
```

Done.



Finding an arbitrary node

- Heaps offer fast access to the *largest* node in the heap.
- However, despite their *binary tree* representation, they offer no advantage over simple *lists* in terms of finding an *arbitrary* element.
  - If the element o that the user wishes to find is not the largest, then o could be *anywhere* in the heap.
  - This contrasts with binary *search* trees (more later).
- Hence, to find an object o within a heap, we must search through the *entire heap*.

Finding an arbitrary node

```
T find (T o) {
  final int index = findNode(0, o);
  if (index < 0) {
    throw new NoSuchElementException();
  }
  return _nodeArray[index];
}

int findNode (int rootIdx, T o) {
  if (_nodeArray[rootIdx].equals(o)) {
    return rootIdx;
  }

  int idx;
  if (leftChild(rootIdx) < _numNodes &&
      (idx = find(leftChild(rootIdx), o)) >= 0) {
    return idx;
  } else if (rightChild(rootIdx) < _numNodes &&
      (idx = find(rightChild(rootIdx), o)) >= 0) {
    return idx;
  } else {
    return -1;
  }
}
```

We *could* implement `findNode` by recursively searching through the entire tree.

Finding an arbitrary node

But this is much easier (and slightly faster too).

```
int findNode (T o) {
  for (int i = 0; i < _numNodes; i++) {
    if (_nodeArray[i].equals(o)) {
      return i;
    }
  }
}
```

- This is one of the conveniences of representing the tree as an array.
  - Only possible for *complete* trees in which there are no “holes” in the array (i.e., missing child nodes).

Removing an arbitrary node

- Removing an arbitrary node requires that we first *find* the node *n* to be removed.
  - We can use the `findNode(index, o)` method we just constructed.
- Once found, we can *swap* the *last* node in the heap (right-most child of last level) with *n*.
- Then we just `trickleDown` that node and we're done, right?

Removing an arbitrary node

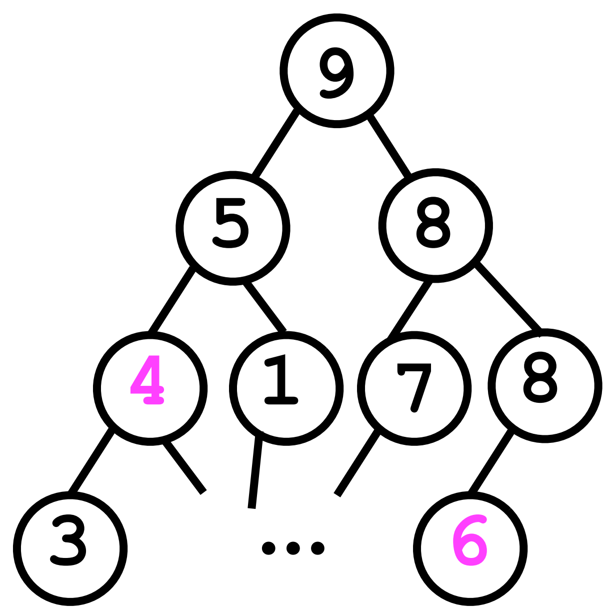
- Removing an arbitrary node requires that we first *find* the node *n* to be removed.
  - We can use the `findNode(index, o)` method we just constructed.
- Once found, we can *swap* the *last* node in the heap (right-most child of last level) with *n*.
- Then we just `trickleDown` that node and we're done, right? Wrong.

Removing an arbitrary node

- The above procedure worked for **`removeLargest()`** because we always started from the *top* (root) of the heap.
  - By trickling down from the top, we guarantee that *every* sub-tree (starting from the very top) is a valid heap.
- When removing an *arbitrary* node, the **`trickleDown`** process will "fix" the sub-tree rooted at *n*, but *not necessarily* the *whole* tree.
- What's an example heap in which this problem would arise?

Removing an arbitrary node

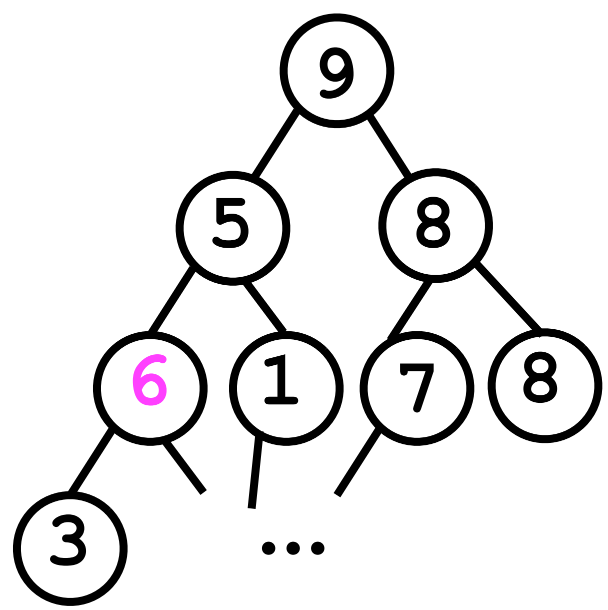
- Suppose we wish to remove the node containing 4.
- If we just replace it with the “last” node (6)...



Valid heap.

Removing an arbitrary node

- ...then the `trickleDown()` method will do nothing (6 is already bigger than its children).
- Moreover, 6 is now bigger than its parent -- a *violation of the heap condition*.



Invalid heap.

Removing an arbitrary node

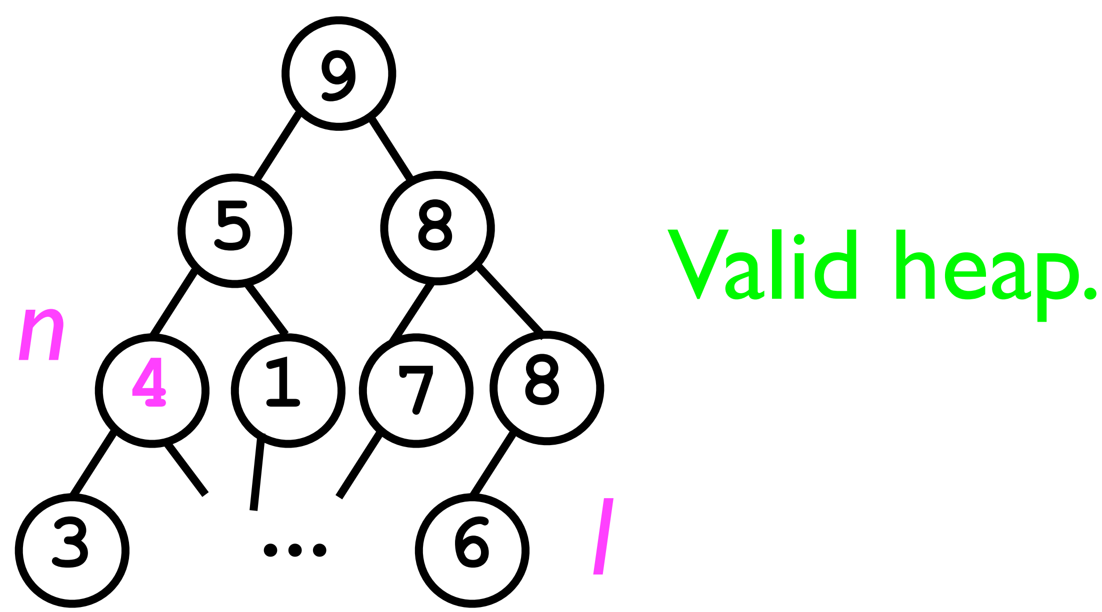
- In a correct implementation of **`remove(o)`** for arbitrary o, we need to *sometimes* **`bubbleUp`** *and sometimes* **`trickleDown`**:

```
void remove (T o) {
  Find the node n containing o.
  Replace n with the "last" node l in the heap.
  If n > l:
    trickleDown on n.
  Else:
    bubbleUp on n.
}
```

Removing an arbitrary node

- In a correct implementation of **`remove(o)`** for arbitrary o, we need to *sometimes* **`bubbleUp`** *and sometimes* **`trickleDown`**:

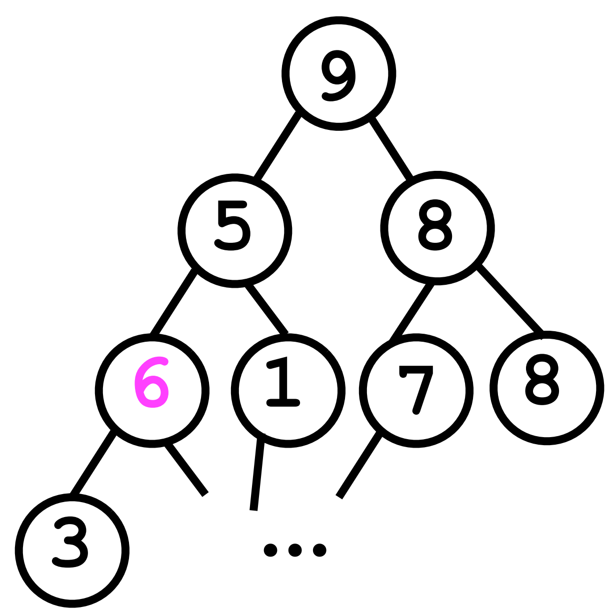
```
void remove (T o) {
  Find the node n containing o.
  Replace n with the "last" node l in the heap.
  If n > l:
    trickleDown on n.
  Else:
    bubbleUp on n.
}
```



Removing an arbitrary node

- In a correct implementation of **`remove(o)`** for arbitrary o, we need to *sometimes* **`bubbleUp`** *and sometimes* **`trickleDown`**:

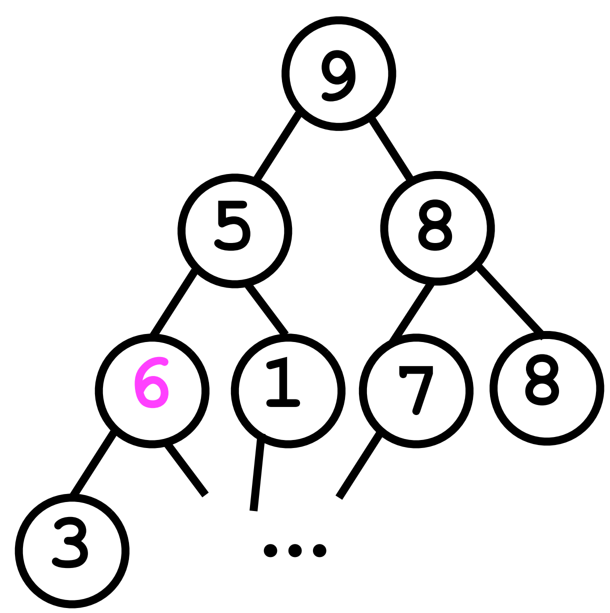
```
void remove (T o) {
  Find the node n containing o.
  Replace n with the "last" node l in the heap.
  If n > l:
    trickleDown on n.
  Else:
    bubbleUp on n.
}
```



Removing an arbitrary node

- In a correct implementation of **`remove(o)`** for arbitrary o, we need to *sometimes* **`bubbleUp`** *and sometimes* **`trickleDown`**:

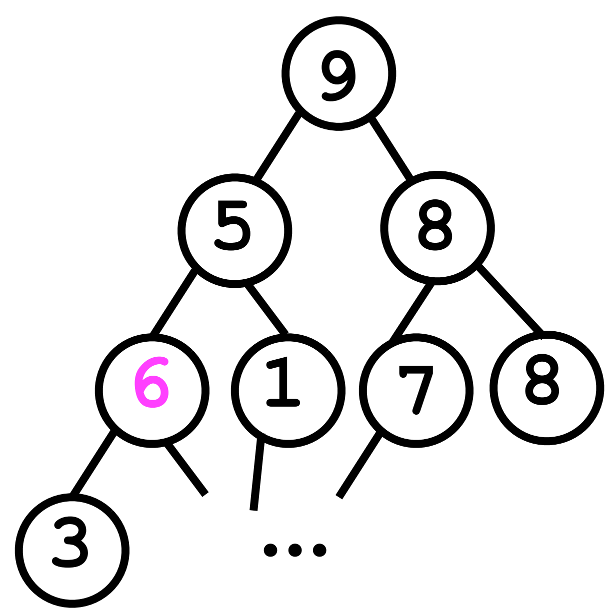
```
void remove (T o) {
  Find the node n containing o.
  Replace n with the “last” node l in the heap.
  If n > l:  // n was 4, l is 6
    trickleDown on n.
  Else:
    bubbleUp on n.
}
```



Removing an arbitrary node

- In a correct implementation of **`remove(o)`** for arbitrary o, we need to *sometimes* **`bubbleUp`** *and sometimes* **`trickleDown`**:

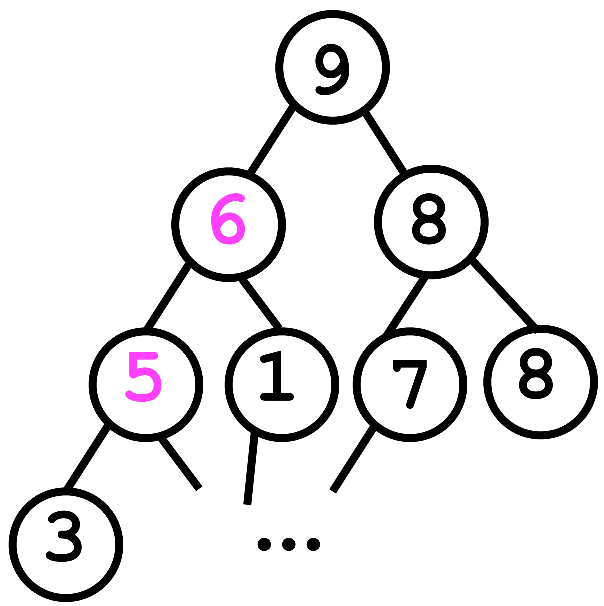
```
void remove (T o) {
  Find the node n containing o.
  Replace n with the "last" node l in the heap.
  If n > l:  // n was 4, l is 6
    trickleDown on n.
  Else:
    bubbleUp on n.
}
```



Removing an arbitrary node

- In a correct implementation of **`remove(o)`** for arbitrary o, we need to *sometimes* **`bubbleUp`** *and sometimes* **`trickleDown`**:

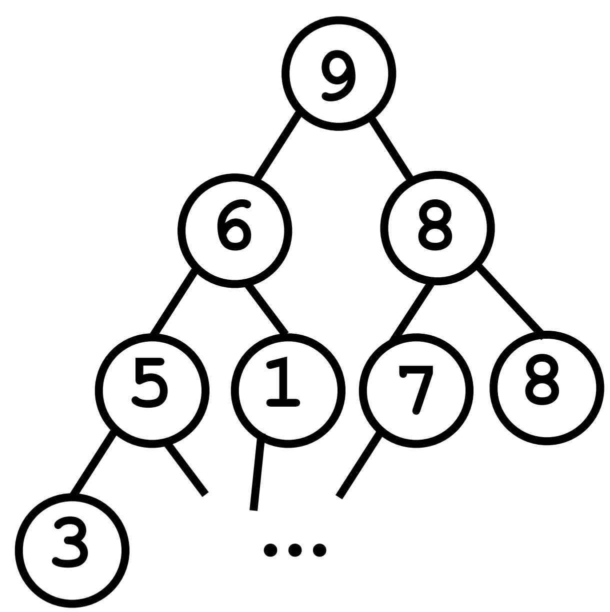
```
void remove (T o) {
  Find the node n containing o.
  Replace n with the "last" node l in the heap.
  If n > l:  // n was 4, l is 6
    trickleDown on n.
  Else:
    bubbleUp on n.
}
```



Removing an arbitrary node

- In a correct implementation of **`remove(o)`** for arbitrary o, we need to *sometimes* **`bubbleUp`** *and sometimes* **`trickleDown`**:

```
void remove (T o) {
  Find the node n containing o.
  Replace n with the “last” node l in the heap.
  If n > l:  // n was 4, l is 6
    trickleDown on n.
  Else:
    bubbleUp on n.
}
```



Valid heap again.

Heap operations: time costs

- The implementations for the **`add`**/**`find`**/**`removeLargest`**/**`remove`** methods depend on the methods **`bubbleUp`** and **`trickleDown`**.

- ```
  void bubbleUp (int idx) {
    While node at idx is “larger” than its parent:
      Swap data in the node and its parent;
      Set idx to be parentIdx(idx);
  }
  ```

  - At each loop iteration, **`idx`** moves one step closer from a leaf to the root of the heap.

    - Hence, loop can execute maximum of *h* times (h is tree height). For heap of *n* nodes, *h* is $\log_2(n)$. Why?

  - Inside loop, the time cost is about 2 operations.

  - Hence, time cost is $O(\log n)$.
  ```

Heap operations: time costs

- ```
  void trickleDown (int index) {
    While node at index is less than one of its children:
      Swap node with the larger child node.
      index = largerChild(index);
  }
  ```
  - At each loop iteration, **`idx`** moves one step closer from the root of the heap to a leaf.
    - Hence, number of iterations is bounded by $h = \log_2(n)$.
  - Inside loop, the time cost is about 2 operations.
  - Hence, time cost is $O(\log n)$.
  ```

Heap operations: time costs

- Given the time costs of `bubbleUp` and `trickleDown`, we can compute the worst-case time costs of the fundamental heap operations:
  - `add(o)`: $O(1)+O(\log n) = O(\log n)$
    - Append a new node to the heap. $O(1)$
    - Bubble it up. $O(\log n)$
  - `removeLargest()`: $O(1)+O(\log n) = O(\log n)$
    - Swap last node with root. $O(1)$
    - Trickle root down. $O(\log n)$

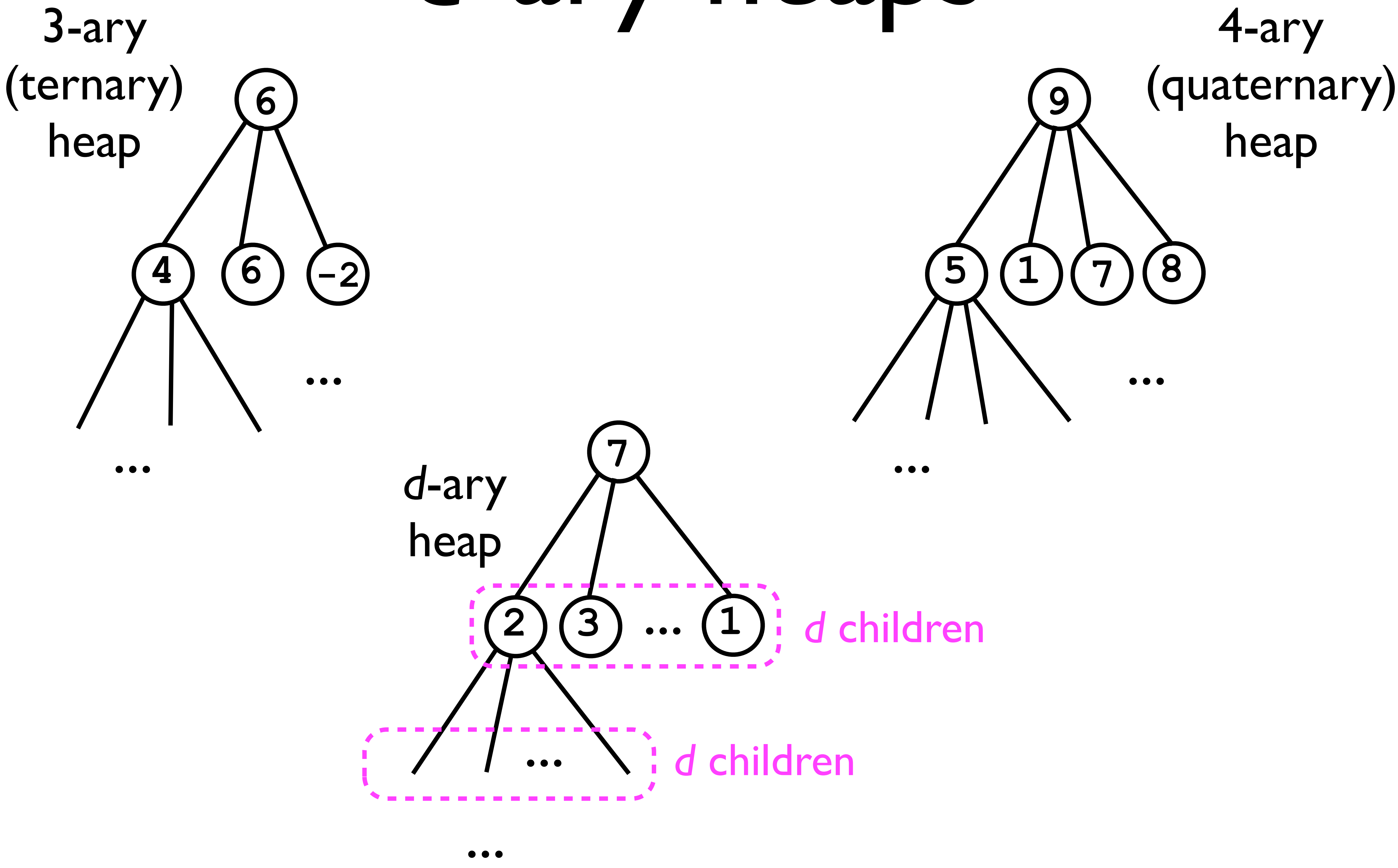
Heap operations: time costs

- `find(o)`: $O(n)$
  - Search through all nodes. $O(n)$
- `remove()`: $O(n)+O(1)+O(\log n) = O(n)$
  - Find the node. $O(n)$
  - Swap node-to-remove with root. $O(1)$
  - *Either* trickle node down *or* bubble it up. $O(\log n)$

General heaps

- We have just described the minimal implementation of a *binary heap*.
  - Binary heaps are the most common.
- In theory, however, *any* tree can be a heap as long as it satisfies the *heap condition* that the root of every sub-tree is no smaller than any node in the sub-tree.
- In particular, we can define a *d*-ary tree in which each node has *d* child nodes (instead of always 2).

d-ary heaps


3-ary
(ternary)
heap
6
4
6
−2
...
...
9
4-ary
(quaternary)
heap
5
1
7
8
...
...
d-ary
heap
7
2
3
...
1
d children
...
d children
...

d-ary heaps: Why?

- *d*-ary heaps can offer a time cost savings compared to binary heaps.
- Consider:
  - The height h of a binary heap is at most $\log_2(n)$.
  - The height h of a ternary heap is at most $\log_3(n)$.
  - The height h of a d-ary heap is at most $\log_d(n)$.
- As the *base* of the logarithm (d) gets *larger*, the *value* of the logarithm itself grows *smaller*.
- Hence, for larger d, operations that depend on the *height* of the tree will become *faster*.

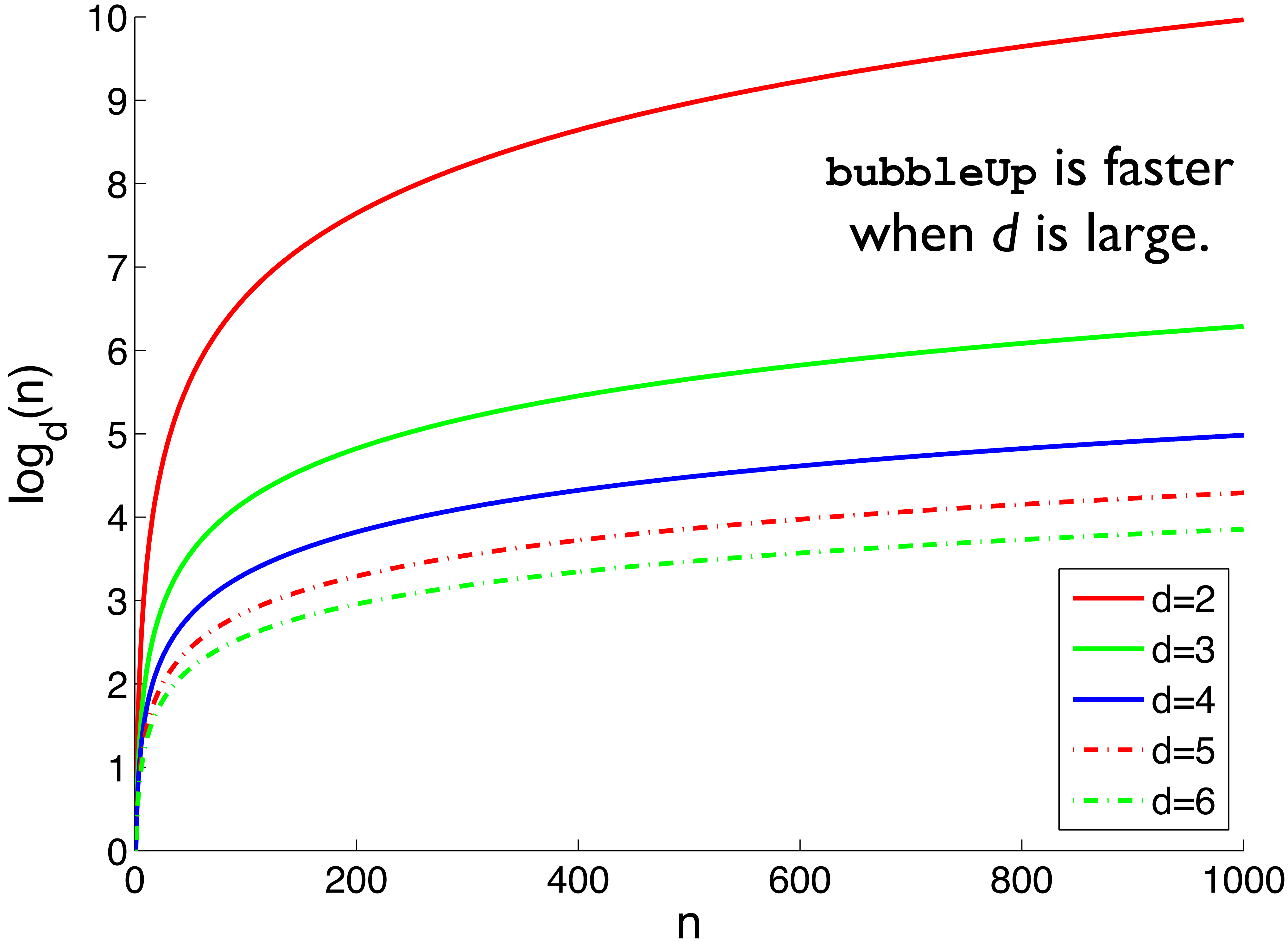
d-ary heaps: Why?

- On the other hand, as *d* increases, so does the *number of children per node.*
- The time cost of **`trickleDown`** (but not **`bubbleUp`**) is affected by the *number* of children:

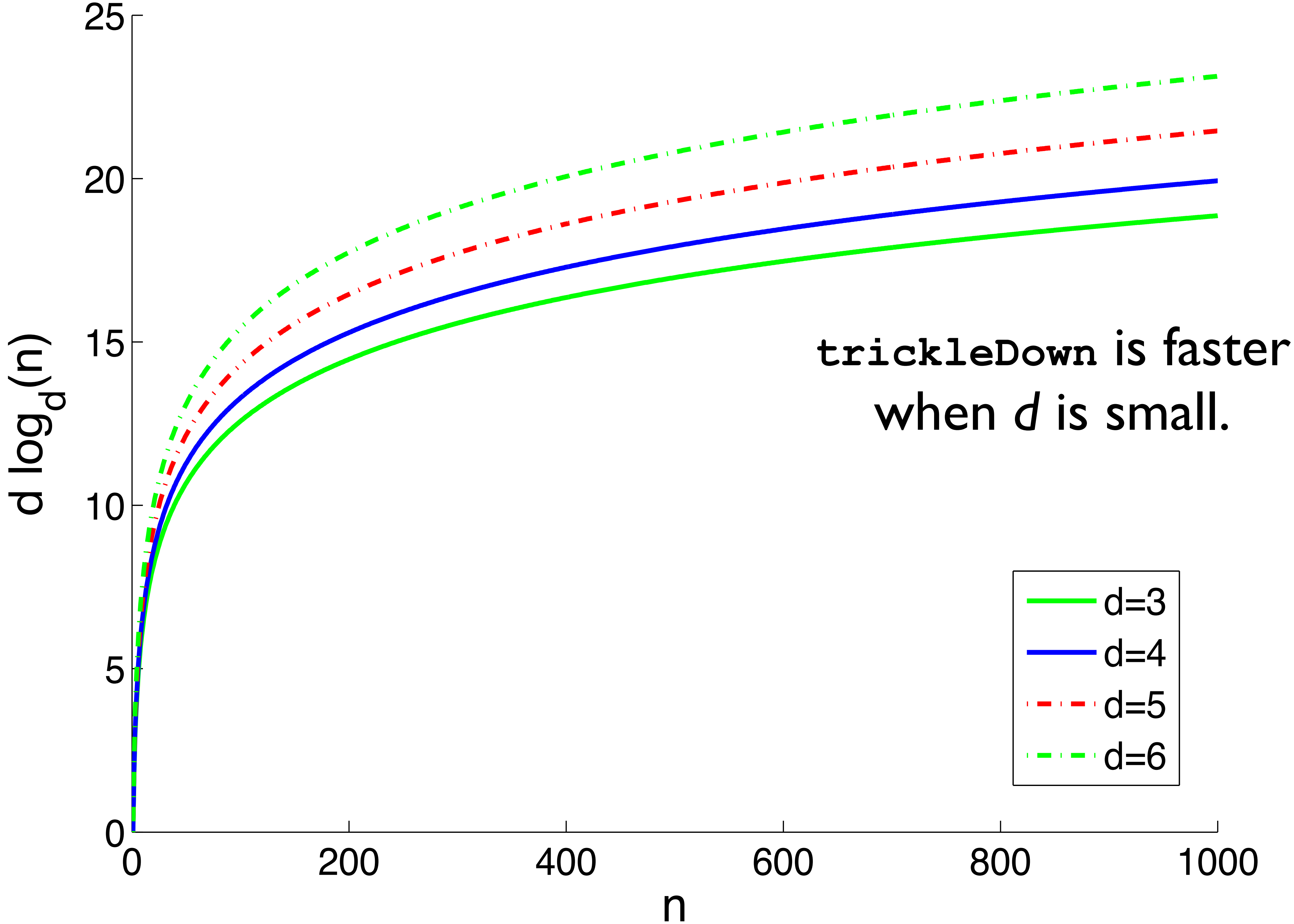
```
void trickleDown (int index) {
  While node at index is less than one of its children:
    ...
}
```

  - Each loop iteration implicitly requires a comparison to all *d* children.
  - The loop runs for at most *h* iterations ($h = \log_d n$), and each iteration takes at least *d* operations.
  - Hence, time cost for **`trickleDown`** is $O(hd) = O(d \log_d n)$.

`bubbleUp`: $O(\log_d n)$



`trickleDown`: $O(d \log_d n)$



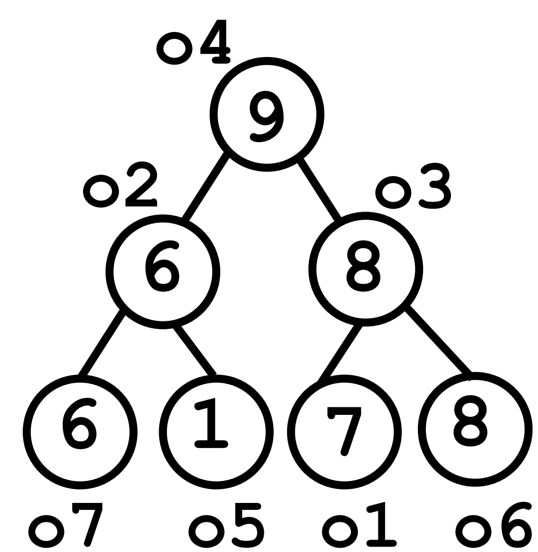
`trickleDown` versus `bubbleUp`

- When would calls to `bubbleUp` occur more frequently than calls to `trickleDown`?
- Consider the use of a heap in implementing a *priority queue*.
  - In priority queues, we want fast access to "highest priority" item.
- Priority queues sometimes offer `increasePriority(o)` and `decreasePriority(o)` methods.
  - These allow the user to *modify* data in the heap without having to *remove* and then *add* it again.

Increasing/decreasing priority

- Example:

```
heap.add(o1);  // Priority 7
heap.add(o2);  // Priority 6
...
heap.add(o7);  // Priority 5
```



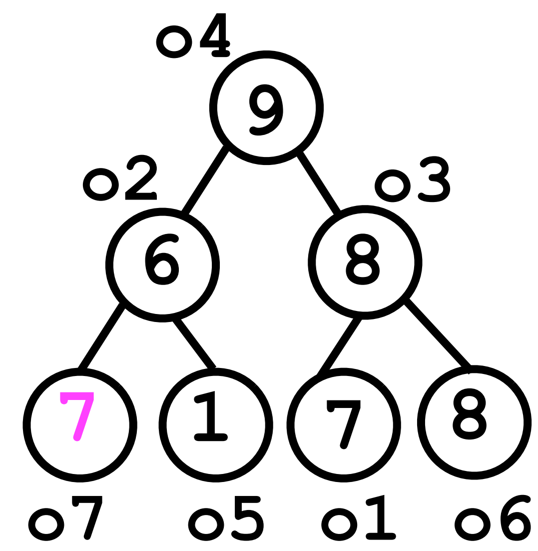
Increasing/decreasing priority

- Example:

```
heap.add(o1);  // Priority 7
heap.add(o2);  // Priority 6
...
heap.add(o7);  // Priority 5
```

- Later on:

```
heap.increasePriority(o7);
```



Now we need to **`bubbleUp`** `o7`.

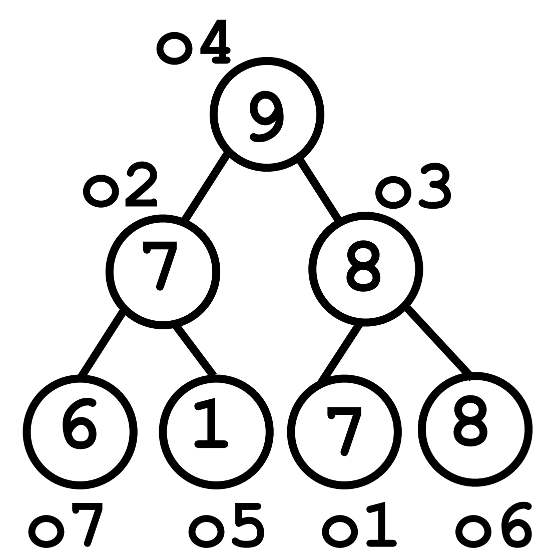
Increasing/decreasing priority

- Example:

```
heap.add(o1);  // Priority 7
heap.add(o2);  // Priority 6
...
heap.add(o7);  // Priority 5
```

- Later on:

```
heap.increasePriority(o7);
```



Done.

`trickleDown` versus `bubbleUp`

- *Increasing* the priority of an item requires `bubbleUp` to be called to maintain the heap condition.
- *Decreasing* the priority of an item requires `trickleDown` to be called to maintain the heap condition.
- In some applications, the user may want to *increase* the priority of items more frequently than they will *decrease* their priority.
  - In this case, `bubbleUp` will be called more frequently than `trickleDown`.
  - By using a *d*-ary heap and setting $d>2$, the time cost of the priority queue may be reduced compared to a binary heap.

Binary search trees

Still something to be desired

- Heaps offer fast access to the largest element in a collection.
  - This is most useful in a priority queue.
- However, finding an *arbitrary* element is still slow -- $O(n)$ time.
- We may want to sacrifice efficiency of access to the *largest* access in exchange for increased efficiency to access any *arbitrary* element.

Binary search trees

- A **binary search tree** (BST) is a binary-tree based data structure that offers $O(\log n)$ *average*-case time costs for:

```
add(o)
find(o)
remove(o)
findLargest/removeLargest(o)
```

- As with heaps, BSTs exploit the order relations among elements.
  - Heaps required the *root* node r of each sub-tree to be no smaller than any *descendant* node of *r*.
  - BSTs impose constraints on the magnitude of nodes in the *left sub-tree* compared to the magnitude of nodes in the *right sub-tree*.

Binary search trees

- More specifically, a binary search tree (BST) is a binary tree (not necessarily complete) that has the following (recursive) *ordering property*:
  - For each node *n*:

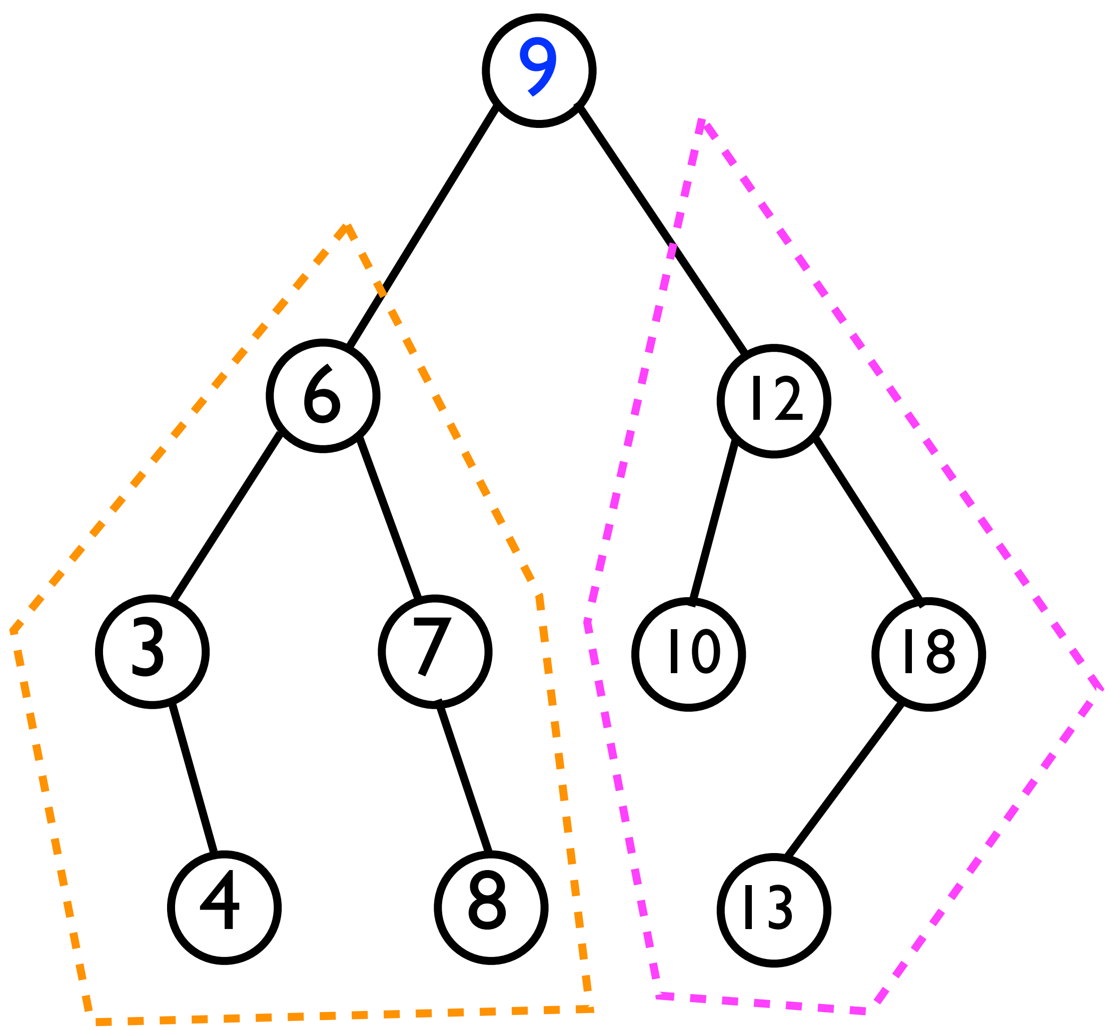
    Base case? Implicit -- *when there are no sub-trees.*

    - All nodes in the *left sub-tree* of *n* are "less than" node *n* itself.
    - All nodes in the *right sub-tree* of *n* are "greater than" node *n* itself.
    - Both the left and right sub-trees are themselves BSTs.

      Recursive part

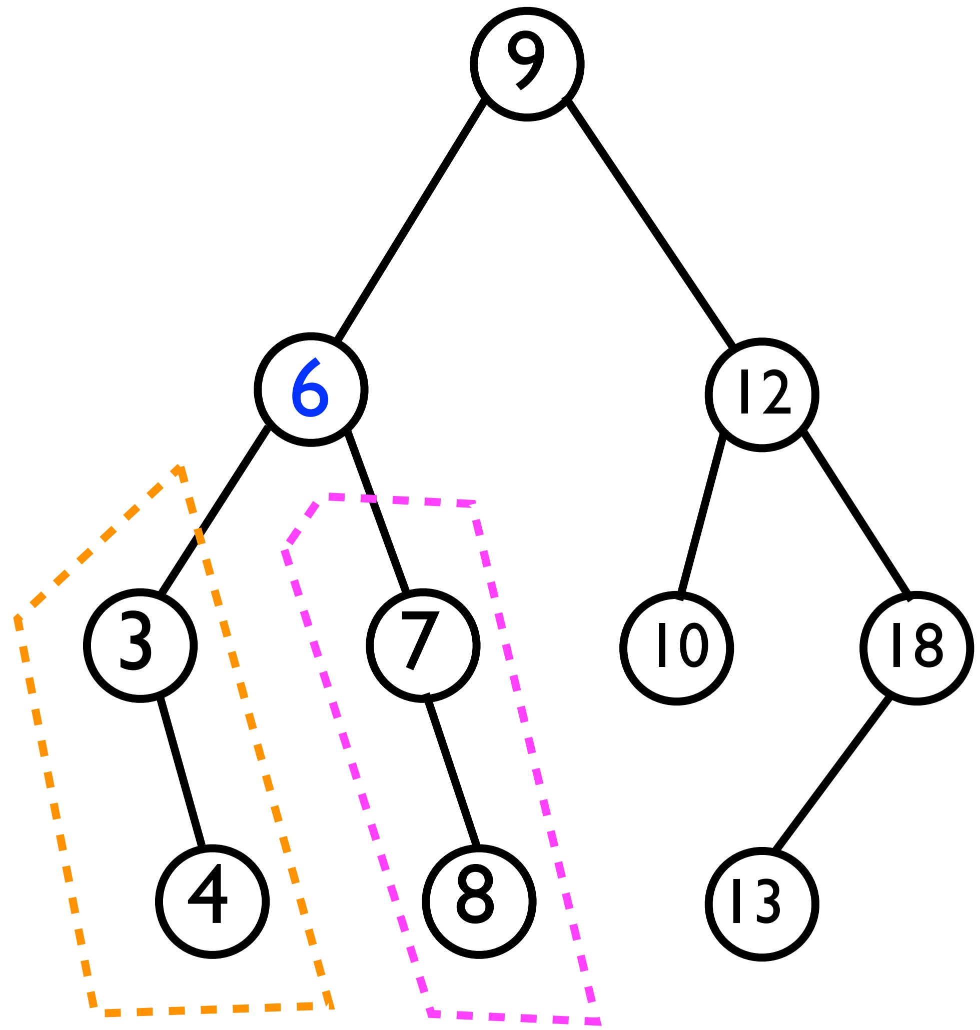
Binary search trees

Left sub-tree < Node (9) < Right sub-tree

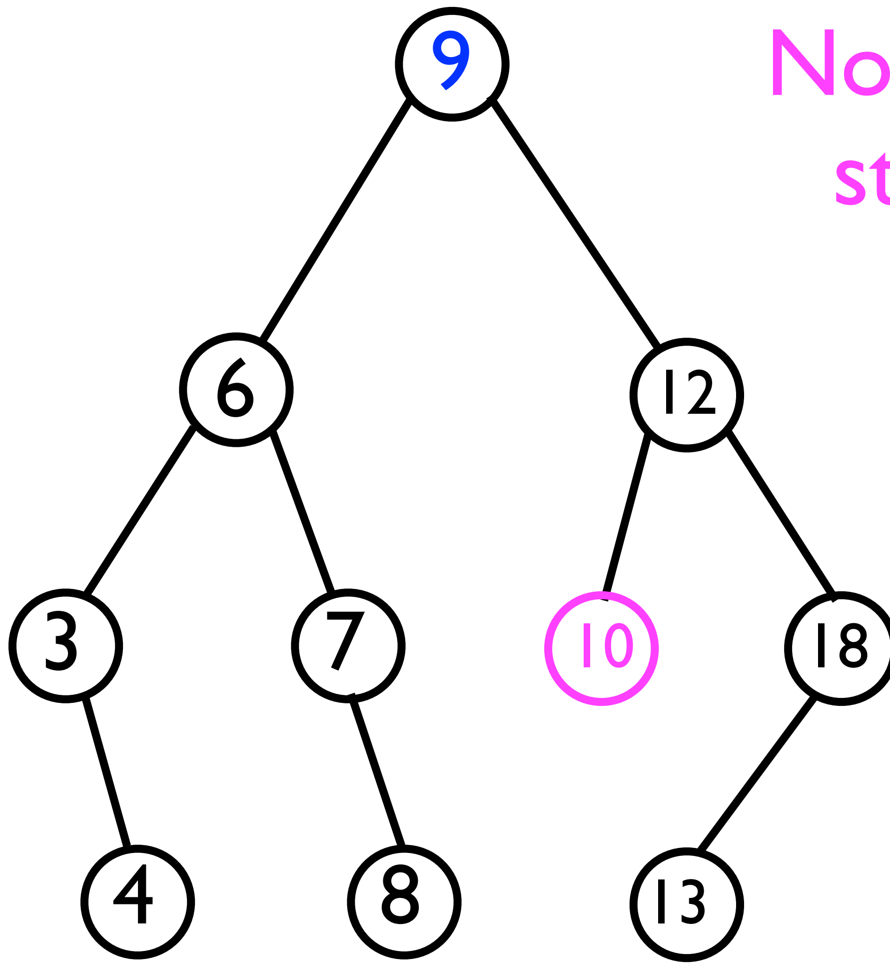


Binary search trees

Left sub-tree < Node (6) < Right sub-tree



Binary search trees



Note that this node must still be greater than 9!

Binary search trees

- In our discussion, we will assume that the keys added to the BST are unique:
  - E.g., we disallow:
    ```
    bst.add(5);
    bst.add(6);
    bst.add(7);
    bst.add(5);  // Error -- the BST already contains 5
    ```
  - This simplifies the exposition slightly.
  - Later, we can relax this restriction.
- In addition, we disallow `null` elements.
  - Unclear what "value" they should have compared to other elements.

Binary search trees

- Let us implement the following operations on BSTs:
  - `T find (T o);`
  - `T findSmallest ();`
  - `T findLargest ();`
  - `add (T o);`
  - `remove (T o);`
- To accomplish this, we will also need a few helper methods (not exposed to user):
  - `Node<T> findNode (Node<T> root, T o);`
  - `Node<T> findSuccessor (Node<T> node);`
  - `Node<T> findParent (Node<T> root, T o);`

Finding the largest element

- Due to the ordering property, finding the largest element of a BST is easy -- we just return the *right-most node* in the whole tree.

```
T findLargest (Node<T> root) {
  while (root._rightChild != null) {
    root = root._rightChild;
  }
  return root._data;
}
```

Finding the smallest element

- Due to the ordering property, finding the smallest element of a BST is easy -- we just return the *left-most node* in the whole tree.

```
T findSmallest (Node<T> root) {
  while (root._leftChild != null) {
    root = root._leftChild;
  }
  return root._data;
}
```

Finding a node

- The ordering property of binary trees also enables efficient search for any *particular* node.

```
// Returns the Node containing o, or else
// null if o is not contained in the BST.
Node<T> findNode (Node<T> root, T o) {
  if (root._data.equals(o) {
    return root;
  } else if (root._data.compareTo(o) < 0 && // Right subtree
             root._rightChild != null) {
     return findNode(root._rightChild, o);
  } else if (root._data.compareTo(o) > 0 && // Left subtree
             root._leftChild != null) {
    return findNode(root._leftChild, o);
  } else {
    return null;
  }
}
```

Due to the ordering property, there is only *one* place in a given BST where value `o` would be stored. If it's not there, then `o` is *not contained in the BST* -- hence, we return `null`.

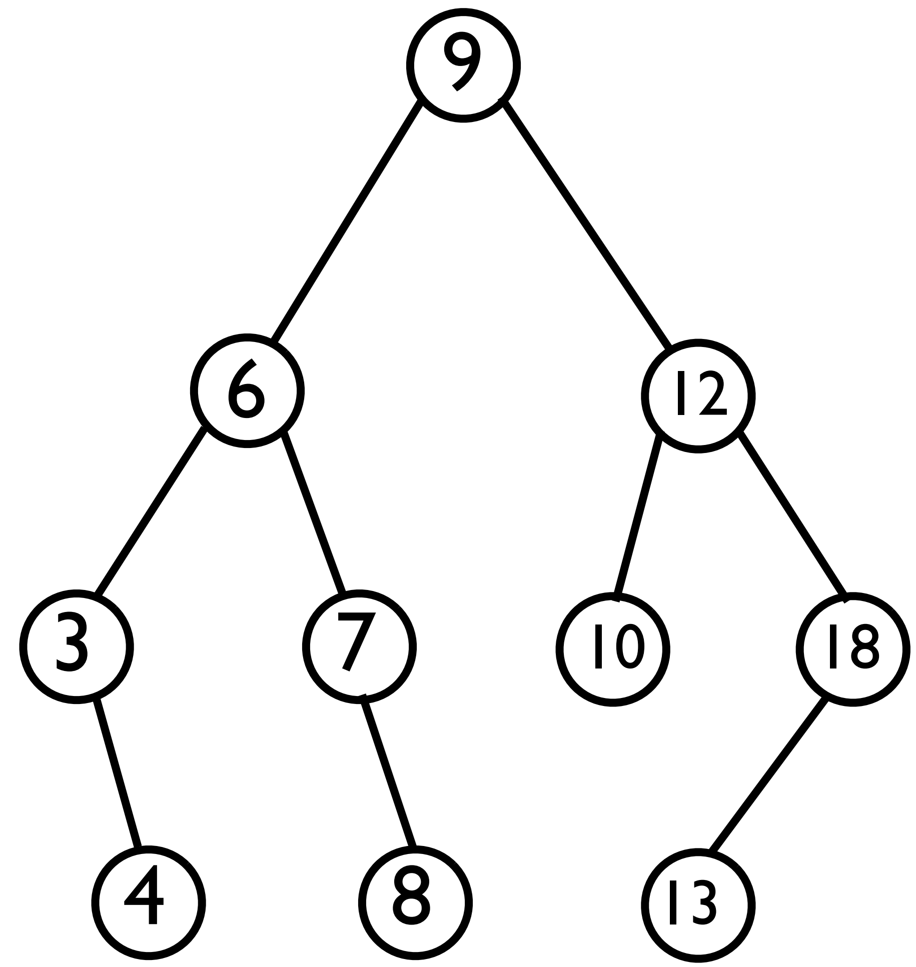
Finding a node

- The **`findNode(root, o)`** method would not be exposed to the user in the BinarySearchTree ADT interface.
- However, we can "wrap" this method with **`T find (T o)`** so that the underlying node infrastructure is hidden:

```
T findNode (T o) {
  if (_root == null) {
    return null;
  } else {
    final Node<T> node = findNode(_root, o);
    if (node == null) {
      return null;
    } else {
      return node._data;
  }
}
```

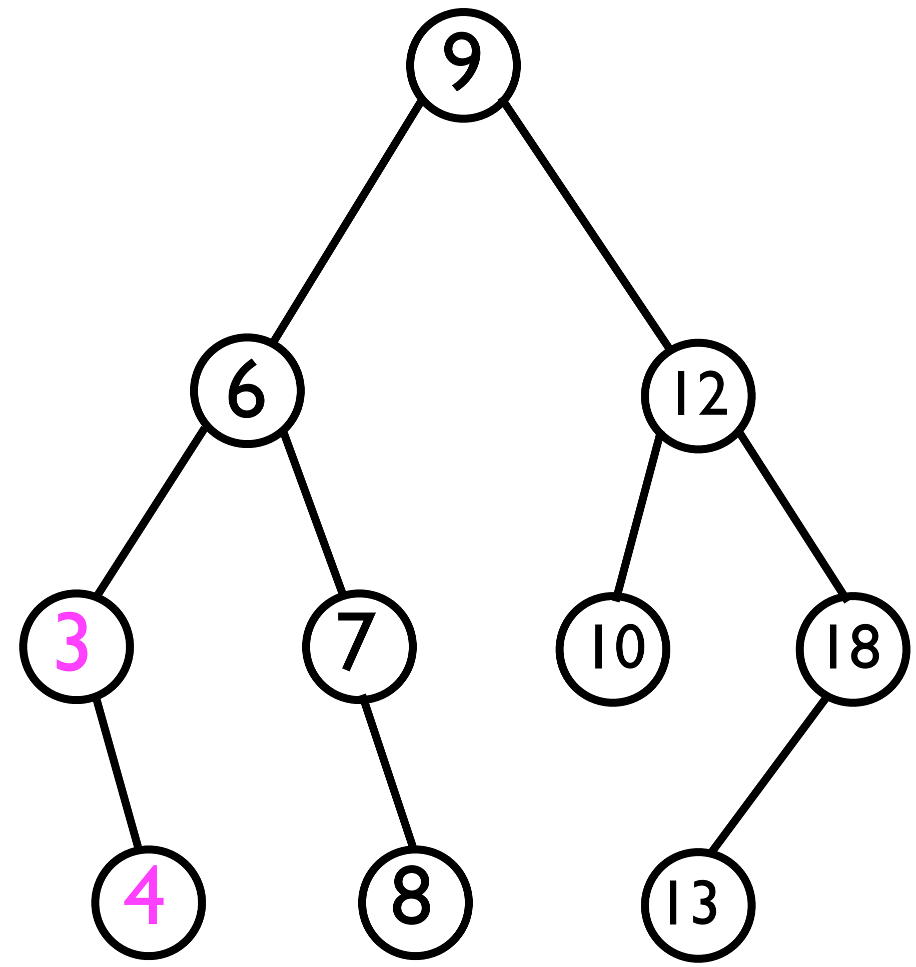
Finding a node's successor

- It will turn out to be useful to be able to find a node's *successor* in the BST.
  - The *successor* of node *n* is the node with the *next higher value*.



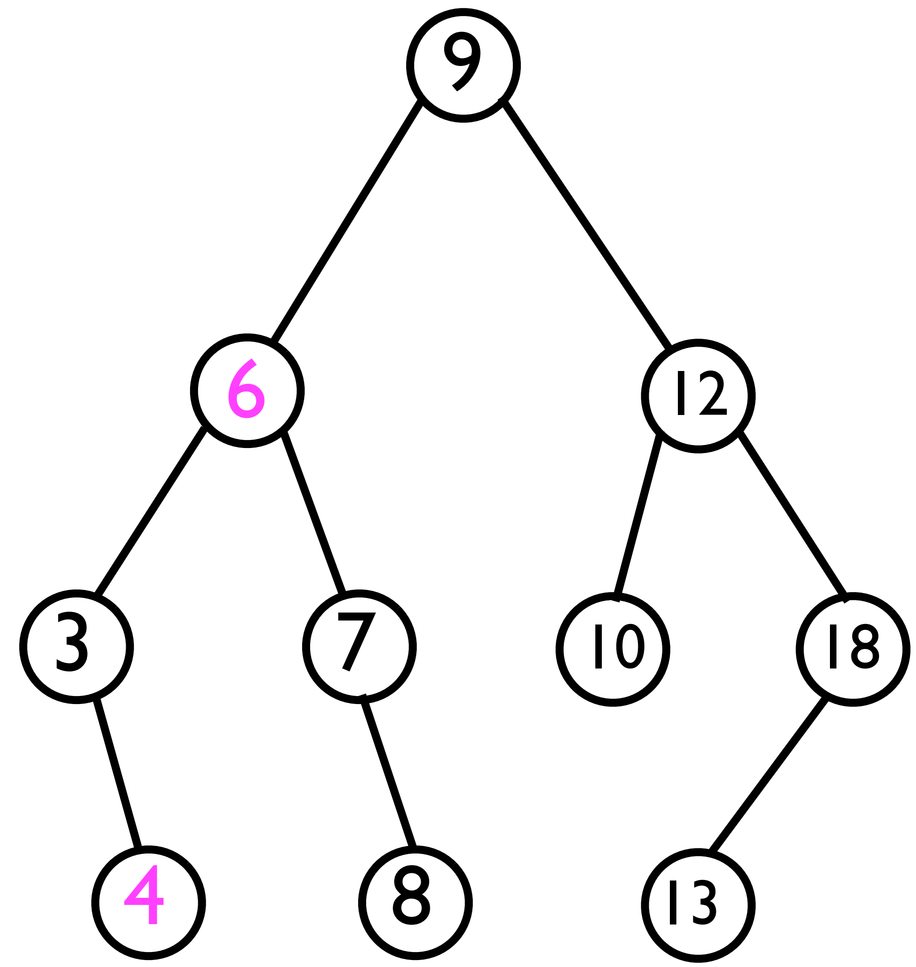
Finding a node’s successor

- It will turn out to be useful to be able to find a node’s *successor* in the BST.
  - The *successor* of node *n* is the node with the *next higher value*.
  - Examples:
    Successor of 3 is 4.
    Successor or 4 is 6.
    Successor of 12 is 13.
    Successor of 8 is 9.



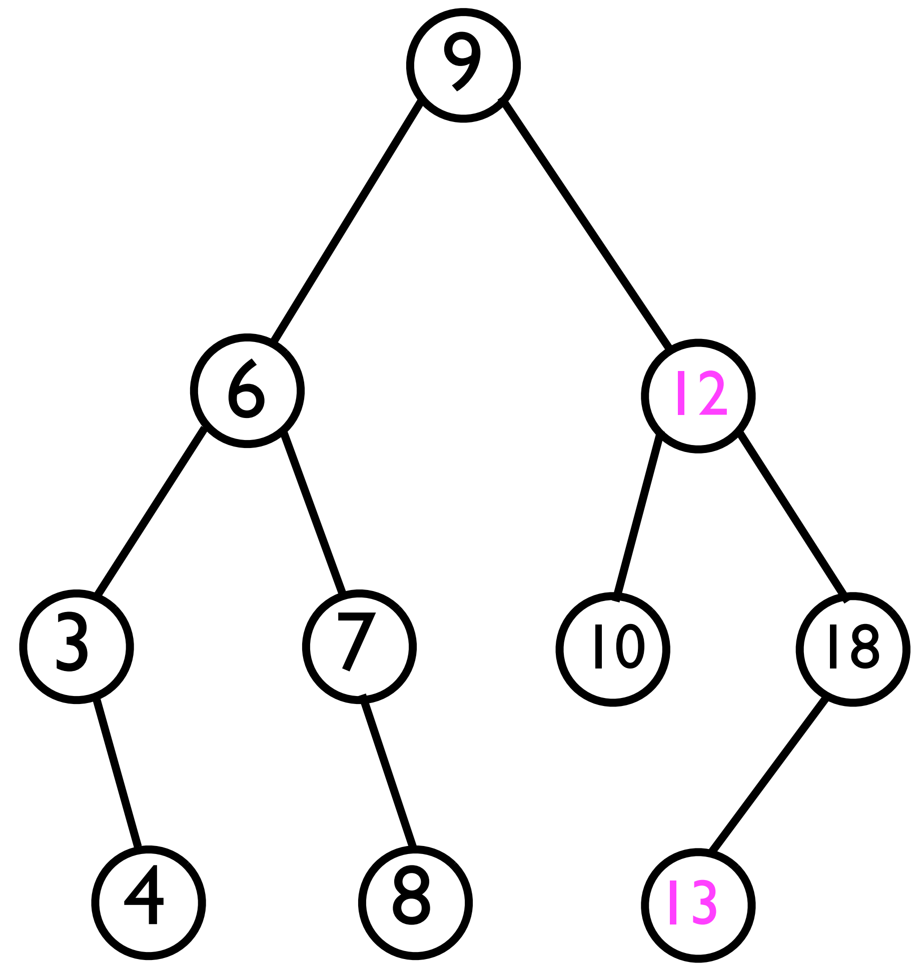
Finding a node's successor

- It will turn out to be useful to be able to find a node's *successor* in the BST.
  - The *successor* of node *n* is the node with the *next higher value*.
  - Examples:
    Successor of 3 is 4.
    Successor or 4 is 6.
    Successor of 12 is 13.
    Successor of 8 is 9.



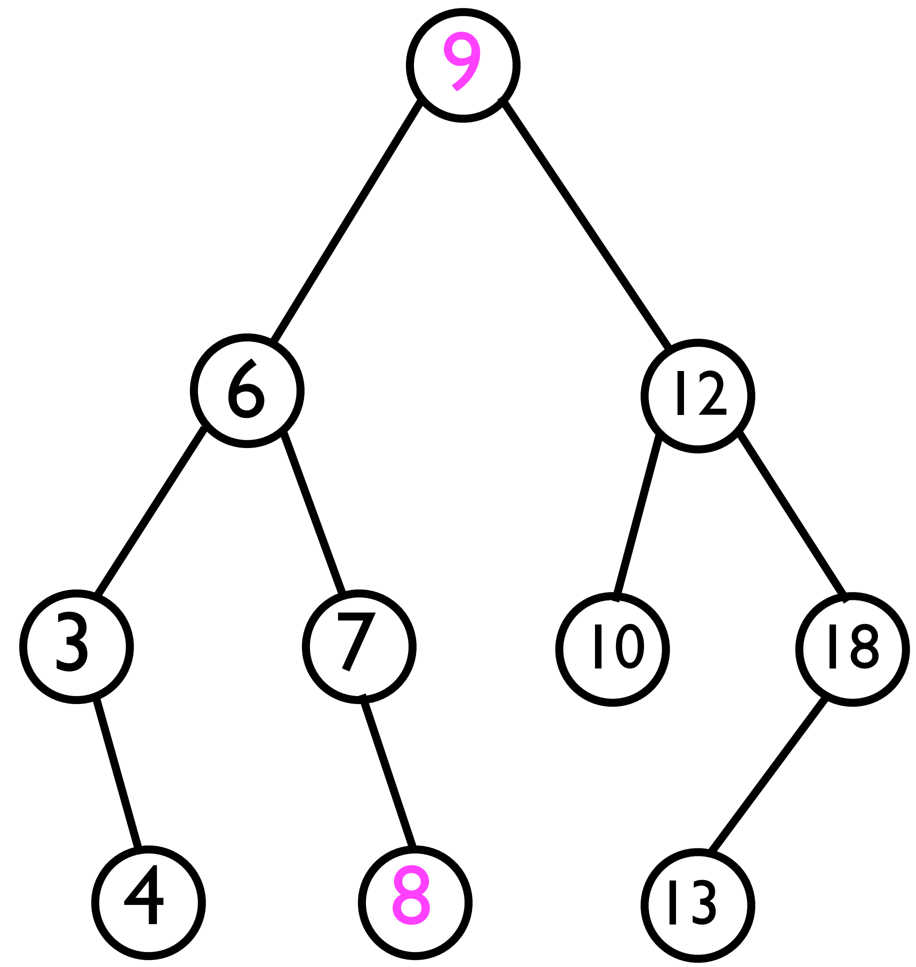
Finding a node's successor

- It will turn out to be useful to be able to find a node's *successor* in the BST.
  - The *successor* of node *n* is the node with the *next higher value*.
  - Examples:
    Successor of 3 is 4.
    Successor or 4 is 6.
    Successor of 12 is 13.
    Successor of 8 is 9.



Finding a node's successor

- It will turn out to be useful to be able to find a node's *successor* in the BST.
  - The *successor* of node *n* is the node with the *next higher value*.
  - Examples:
    Successor of 3 is 4.
    Successor or 4 is 6.
    Successor of 12 is 13.
    Successor of 8 is 9.



Finding a node's successor

- A *successor* node of *n* -- if it exists -- is found by *either*:
  1. Descending into *n*'s right sub-tree, and then recursively selecting left-child until no left child exists.
     - *Intuition*: The right sub-tree has values bigger than n; we want the smallest such value (left-most node).
  2. Finding the *lowest* ancestor of *n* whose left child is also an ancestor of *n*.
     - *Intuition*: Move "up-and-left" in the BST until we can finally "move right" again, i.e., *towards a higher valued node*.

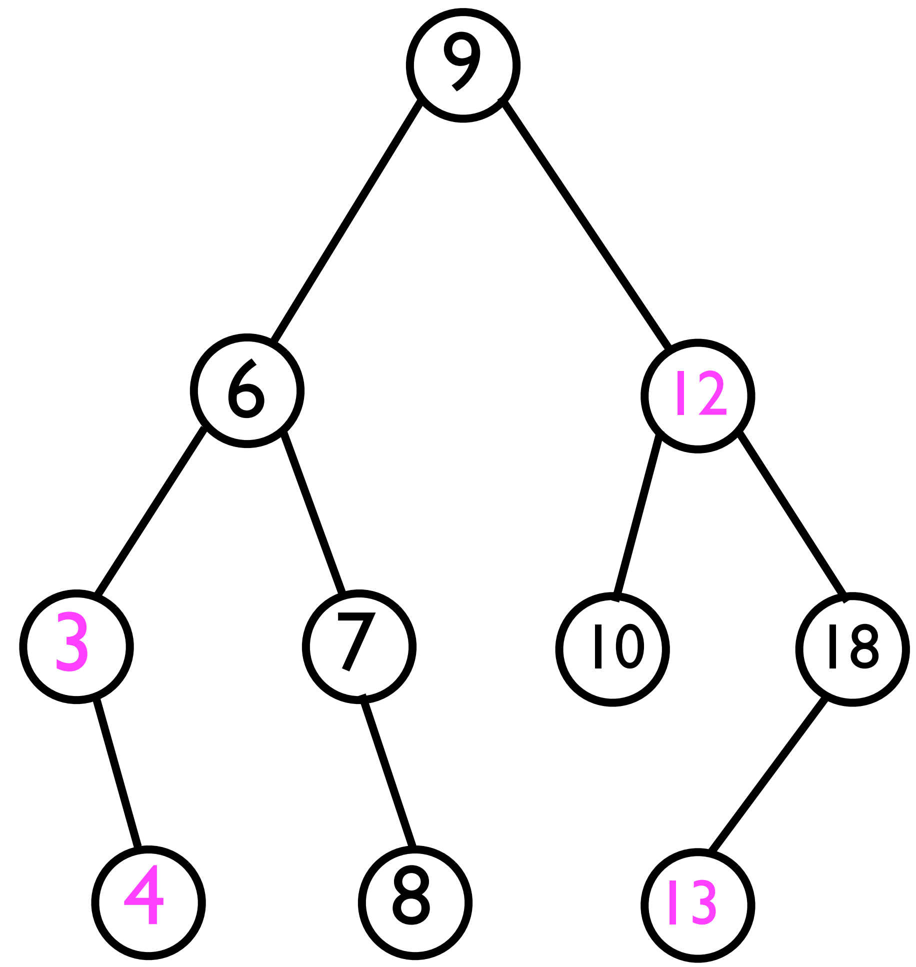
Finding a node's successor

- A *successor* node of *n* -- if it exists -- is found by *either*:

  1. Descending into *n*'s right sub-tree, and then recursively selecting left-child until no left child exists.

  2. Finding the *lowest* ancestor of *n* whose left child is also an ancestor of *n*.

- Examples:
  Successor of 3 is 4.
  Successor or 4 is 6.
  Successor of 12 is 13.
  Successor of 8 is 9.



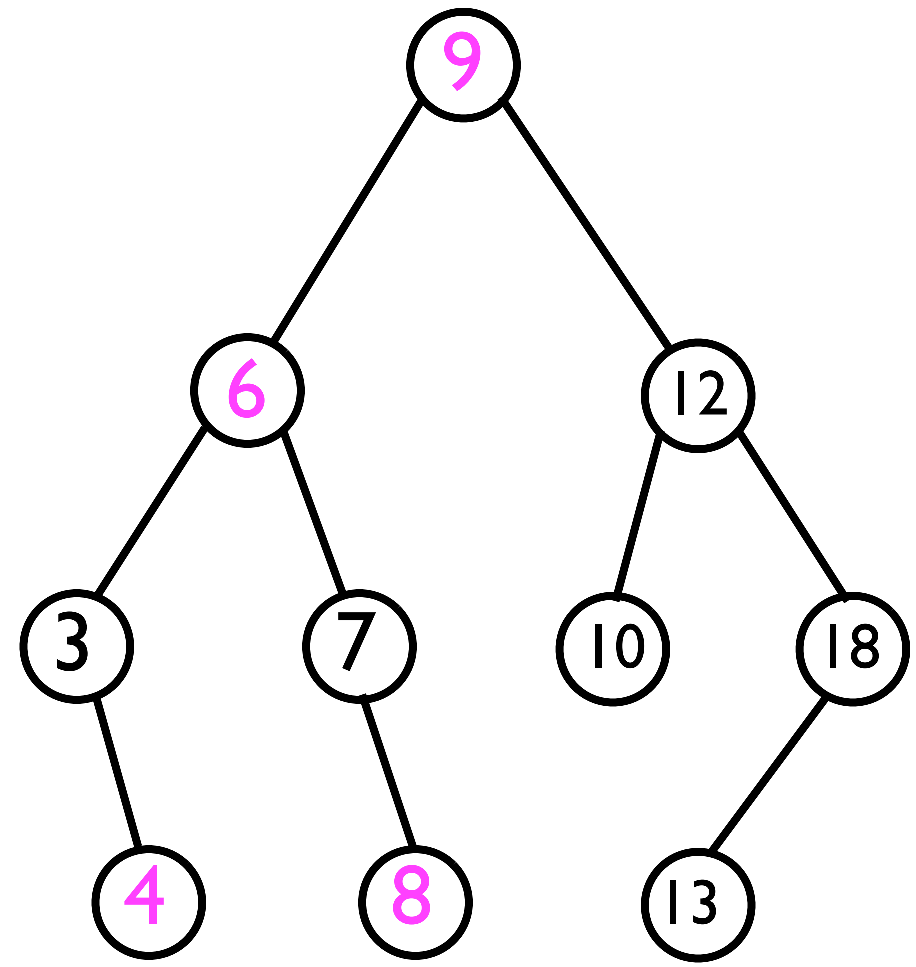
Finding a node's successor

- A *successor* node of *n* -- if it exists -- is found by *either*:

  1. Descending into *n*'s right sub-tree, and then recursively selecting left-child until no left child exists.

  2. Finding the *lowest* ancestor of *n* whose left child is also an ancestor of *n*.

  - Examples:
    Successor of 3 is 4.
    Successor or 4 is 6.
    Successor of 12 is 13.
    Successor of 8 is 9.



Finding a node's successor

- The code for **`Node<T> findSuccessorNode (Node<T> node)`** will be left as an "exercise for the reader".

Adding a new node

- To add a new node, we must distinguish two cases:

1. The new node is the *first* node in the BST.
   - In this case, we simply set this node to be the root.
2. The new node is *not* the first node in the BST.
   - Then we must find the *parent* node of the node we're about to add.
   - We then add the new node as a child of the parent.

Finding the parent of a new node

- To find the parent node of the new node *n* we want to add:
  - Recursive search from root down towards the leaf nodes, *as if node* n *were already inserted*.
  - Eventually, while recursing at node *p*, the search for the node would take us to a left/right child *that does not yet exist.*
    - At that point, we know *p* is the parent of *n*.
    - *p* is the "natural insertion point" for *n*.

Finding the parent of a new node

```
// Searches from root for the parent node to which the
// specified new node should be added.
Node<T> findParentNode (Node<T> root, T o) {
  // Save comparison result
  final int comparison = root._data.compareTo(o);

  if (comparison < 0 && root._rightChild != null) {
    return findParentNode(root._rightChild, o);
  } else if (comparison > 0 && root._leftChild != null) {
    return findParentNode(root._leftChild, o);
  } else {  // The appropriate left/child does not yet exist
    return root;  // Hence, we've found the parent
  }
}
```

Adding a new node

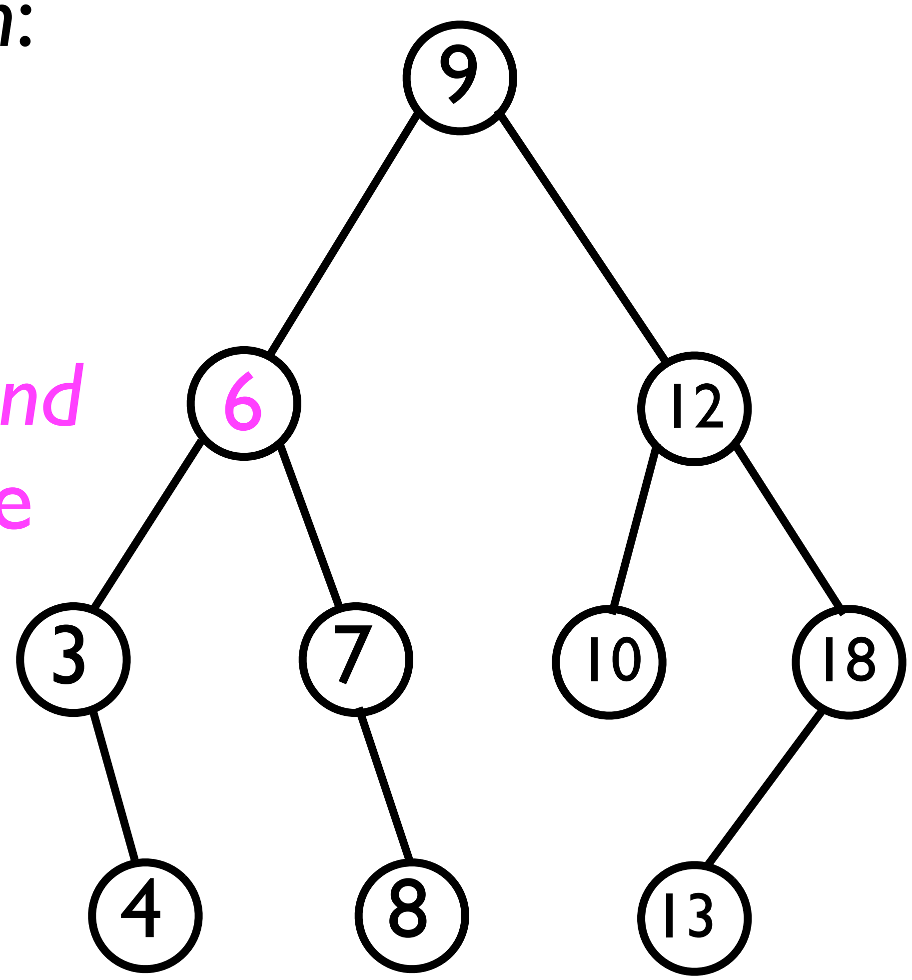
- We can now implement the **`add(o)`** method:

```
void add (T o) {
  final Node<T> node = new Node<T>();
  node._data = o;
  if (_root == null) {  // Case 1
    _root = node;
  } else {              // Case 2
    final Node<T> parent = findParent(_root, o);
    if (parent._data.compareTo(o) < 0) {
      parent._rightChild = node;
    } else {
      parent._leftChild = node;
    }
  }
}
```

Removing a node

- When removing a node *n* from the BST, we must ensure that:
  - The resulting tree is still *connected*.
  - The resulting tree still has the *ordering property*.
- Consider what might "go wrong" when removing an arbitrary node *n*:

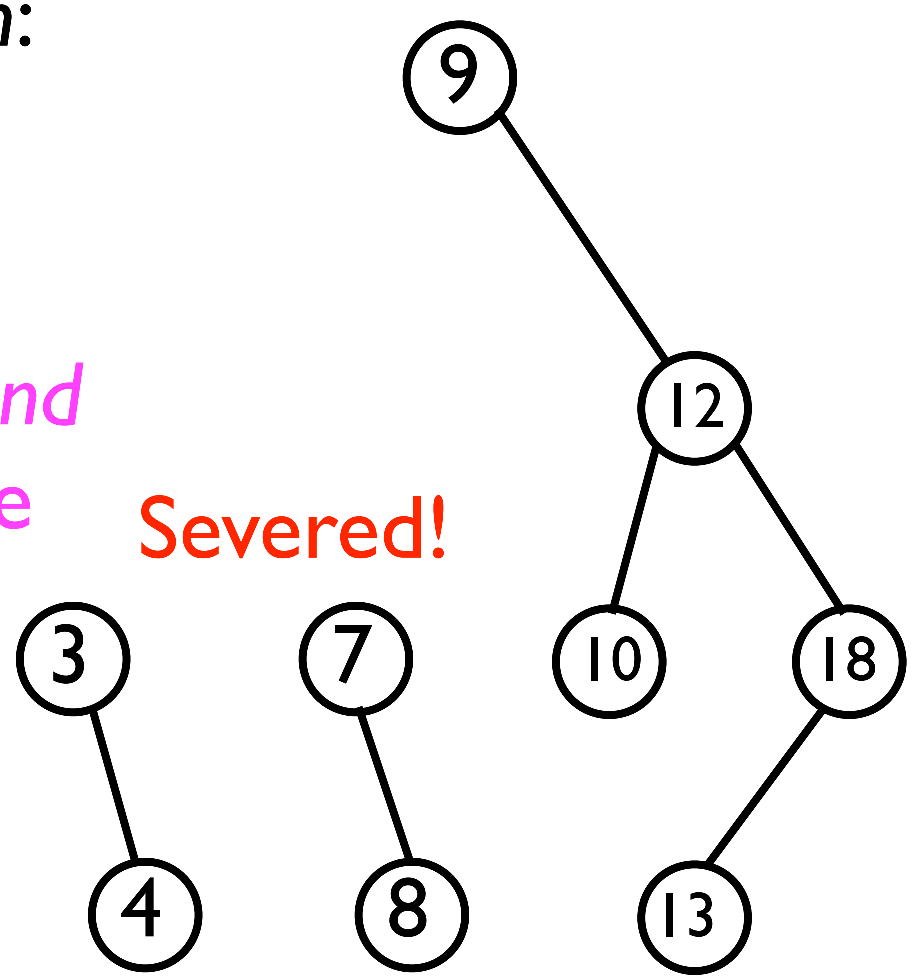
If we remove node 6, then we *sever its left and right sub-trees* from the rest of the BST.



Removing a node

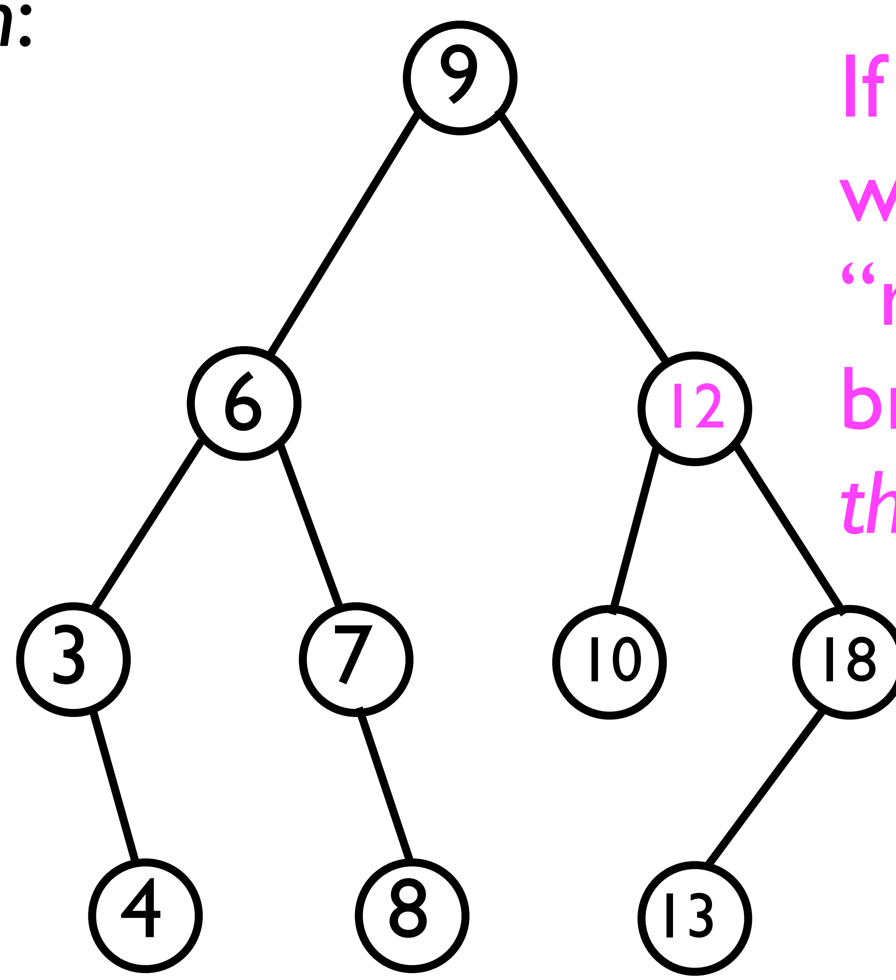
- When removing a node *n* from the BST, we must ensure that:
  - The resulting tree is still *connected*.
  - The resulting tree still has the *ordering property*.
- Consider what might "go wrong" when removing an arbitrary node *n*:

If we remove node 6, then we *sever its left and right sub-trees* from the rest of the BST.



Removing a node

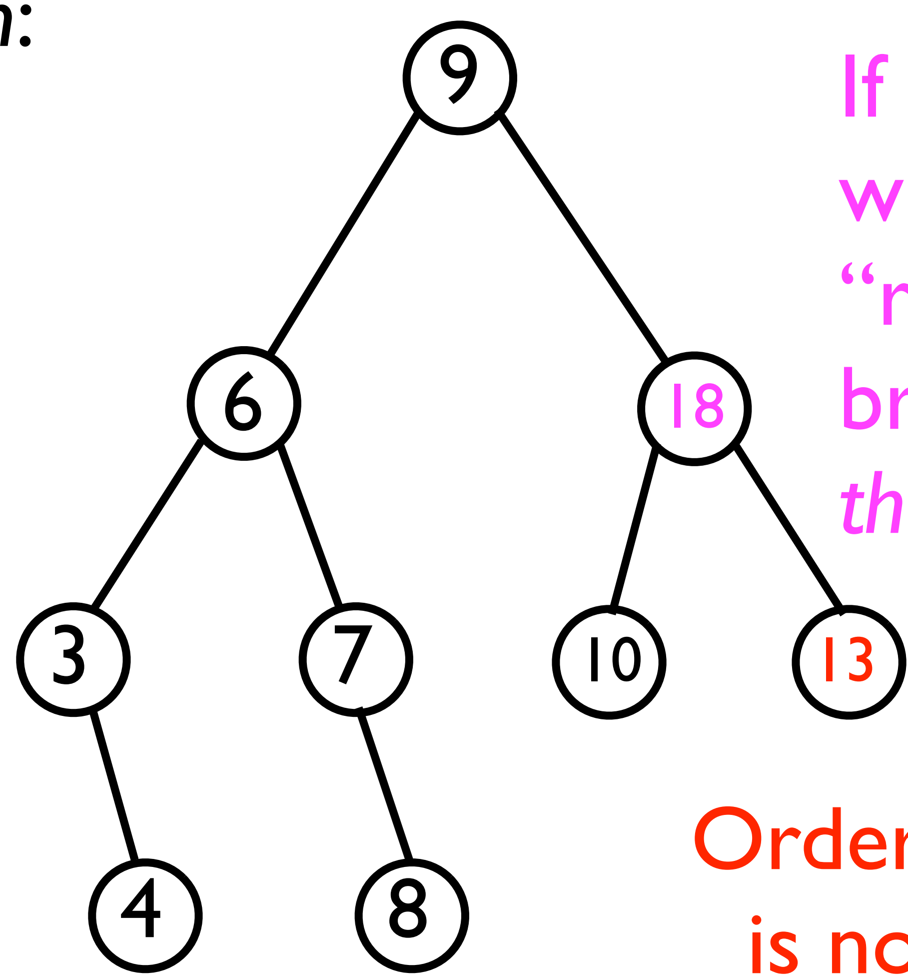
- When removing a node *n* from the BST, we must ensure that:
  - The resulting tree is still *connected*.
  - The resulting tree still has the *ordering property*.
- Consider what might "go wrong" when removing an arbitrary node *n*:



If instead we replace *n* with another node and "reconnect" another branch, we might *violate the ordering property*.

Removing a node

- When removing a node *n* from the BST, we must ensure that:
  - The resulting tree is still *connected*.
  - The resulting tree still has the *ordering property*.
- Consider what might "go wrong" when removing an arbitrary node *n*:



If instead we replace *n* with another node and "reconnect" another branch, we might *violate the ordering property*.

Ordering property is now violated!

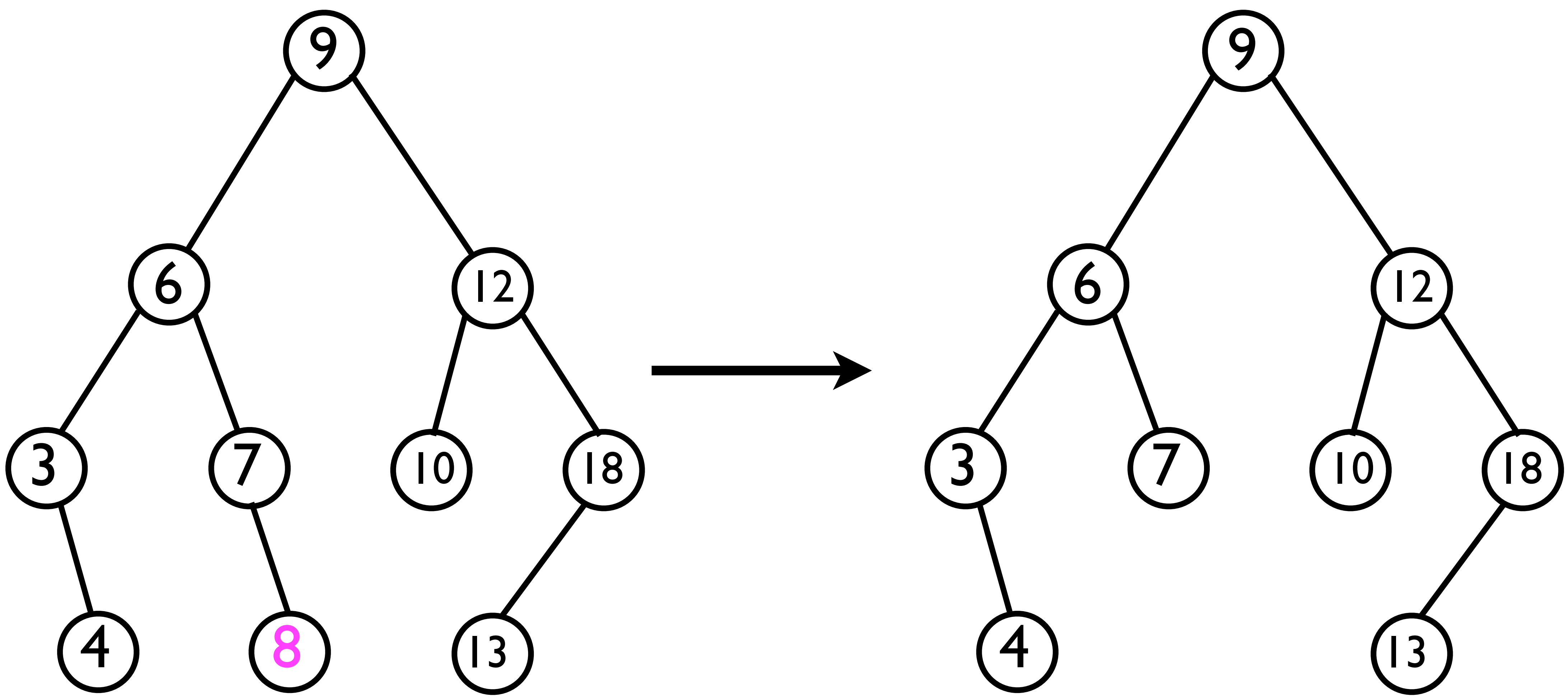
Removing a node

- To remove a node and still ensure the resulting tree is a proper BST, we must distinguish three cases:

1. *n* is a leaf node -- in this case, we just snip it off.
2. *n* is an internal node with only one child.
   - We remove *n* and "splice around" it.
3. *n* is an internal node with two child nodes.
   - We replace *n* with the value of its successor *s*, and then remove *s*.

Removing a leaf node

Example: `bst.remove(8);`

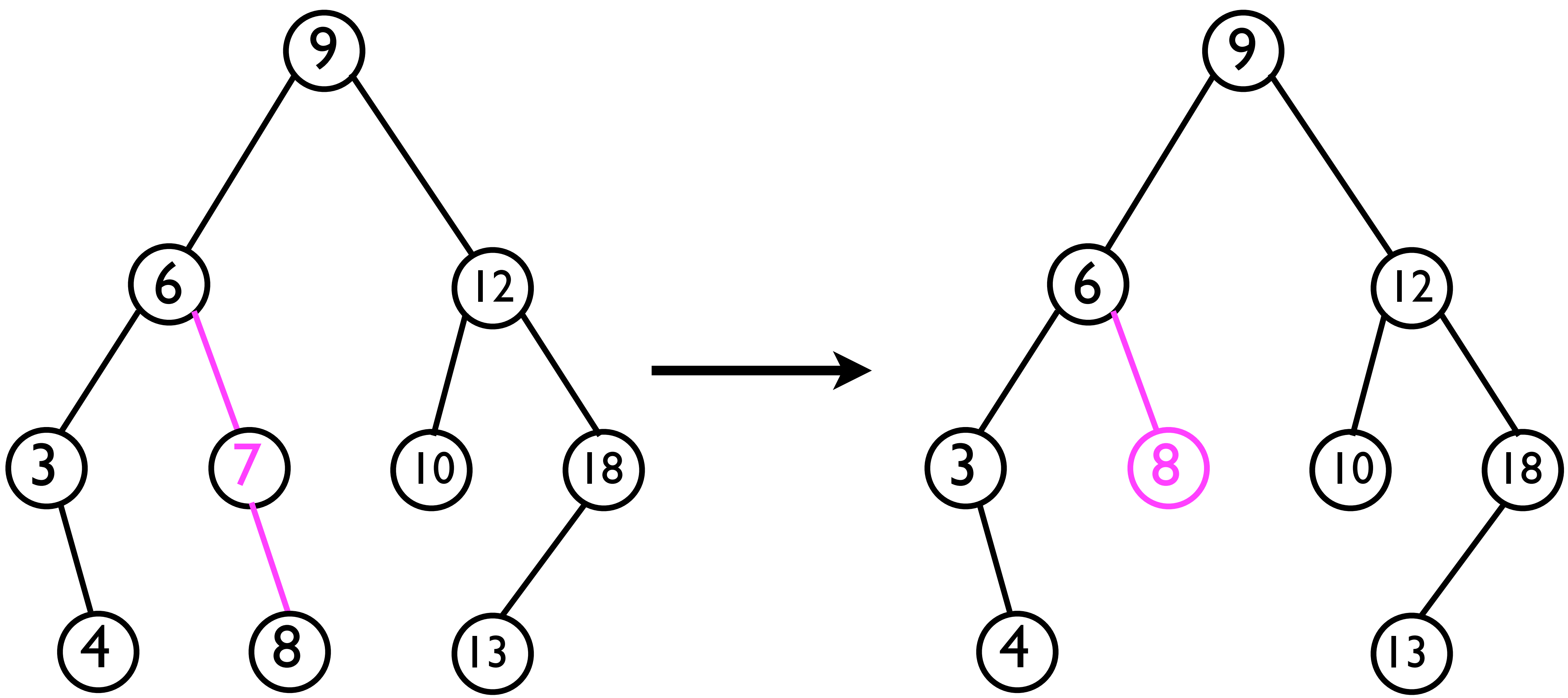


Just snip it off.

Result: We still have a BST with the ordering property preserved.

Removing a node with one child node

Example: `bst.remove(7);`

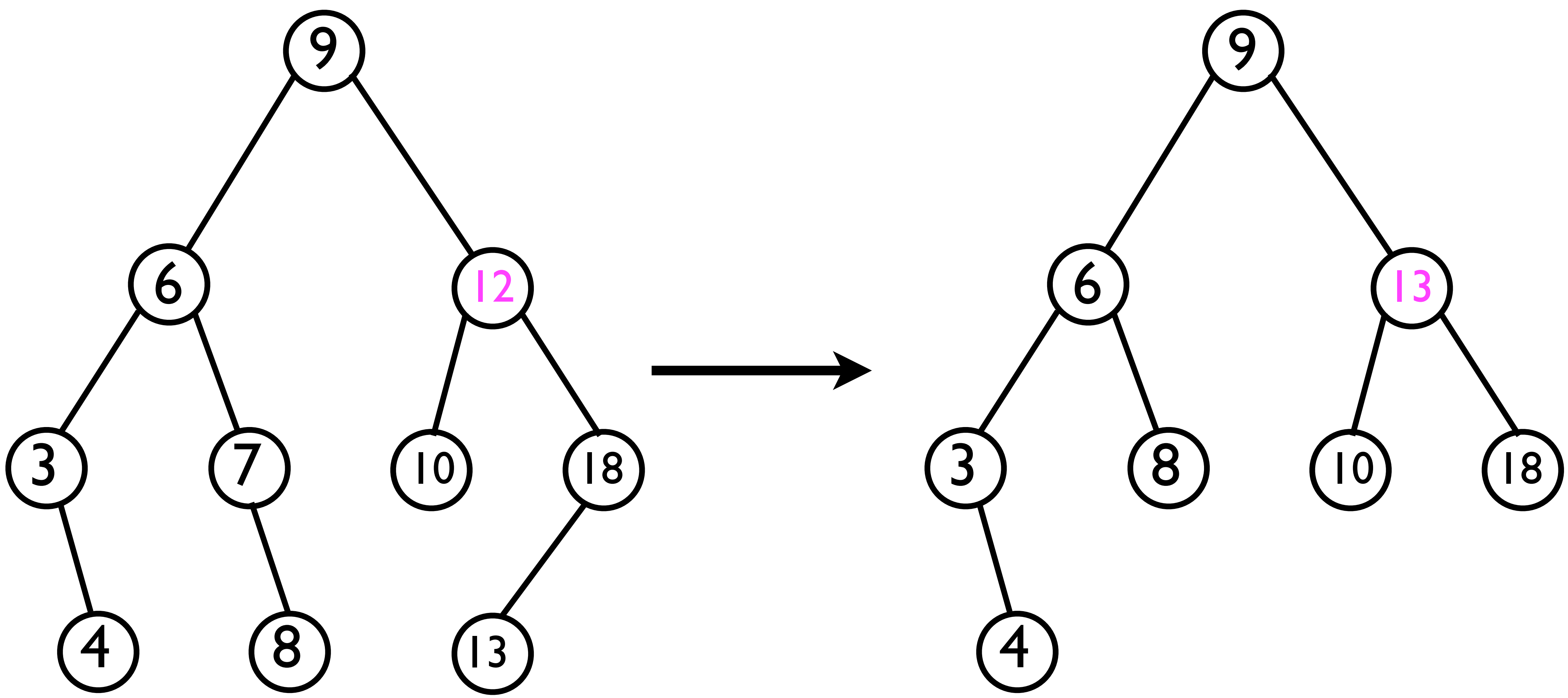


"Splice around" node 7.

Result: We still have a BST with the ordering property preserved.

Removing a node with two child nodes

Example: `bst.remove(12);`



Replace 12 with the value of its *successor*; then remove the successor node.

Result: We still have a BST with the ordering property preserved.

Removing the successor

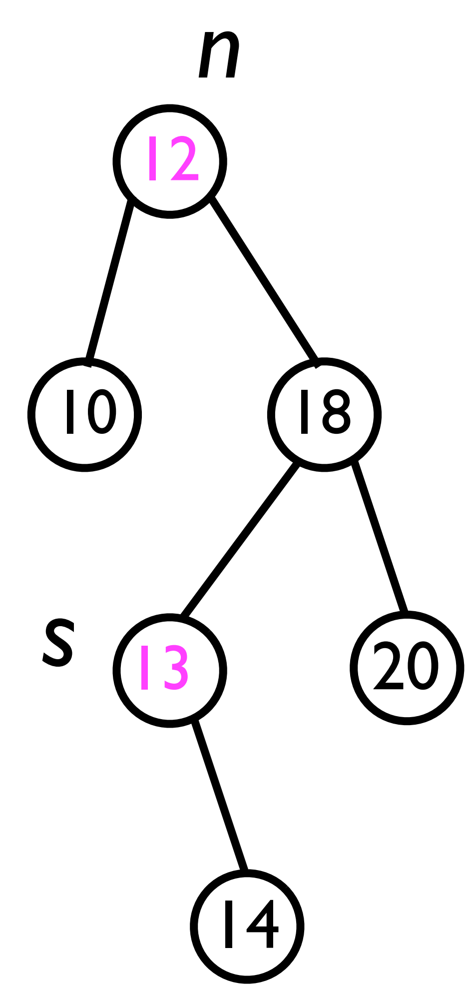
- When removing a node *n* with two children, we replace n with the value of its successor *s*, and then remove *s* itself.
  - But what if *s* *also* has two children; then we need to remove *its* successor, and so on.
  - Will the "removal" process ever terminate?
    - Yes -- if *n* has two children, then its successor *s* *cannot* have a left-child. Why?

Removing the successor

- When removing a node *n* with two children, we replace n with the value of its successor *s*, and then remove *s* itself.
  - But what if *s* *also* has two children; then we need to remove *its* successor, and so on.
  - Will the "removal" process ever terminate?
    - Yes -- if *n* has two children, then its successor *s* *cannot* have a left-child. Why?
      - If it did, then *that left child* would be n's successor, and not *s* itself.

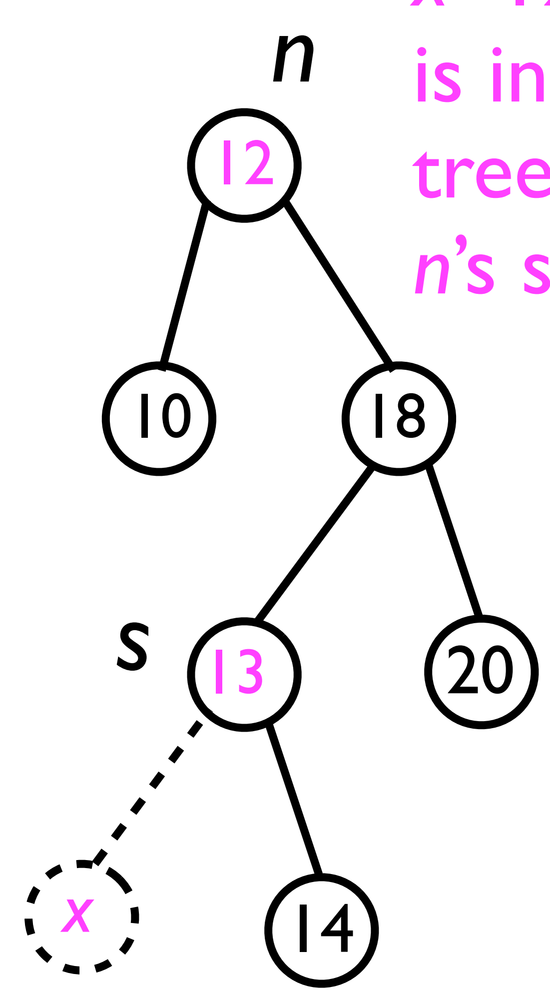
Successor of node with two children

- Example:
  - Let *n* be node 12.
  - Then *n*'s successor *s* is 13.
    - *s* only has one child.



Successor of node with two children

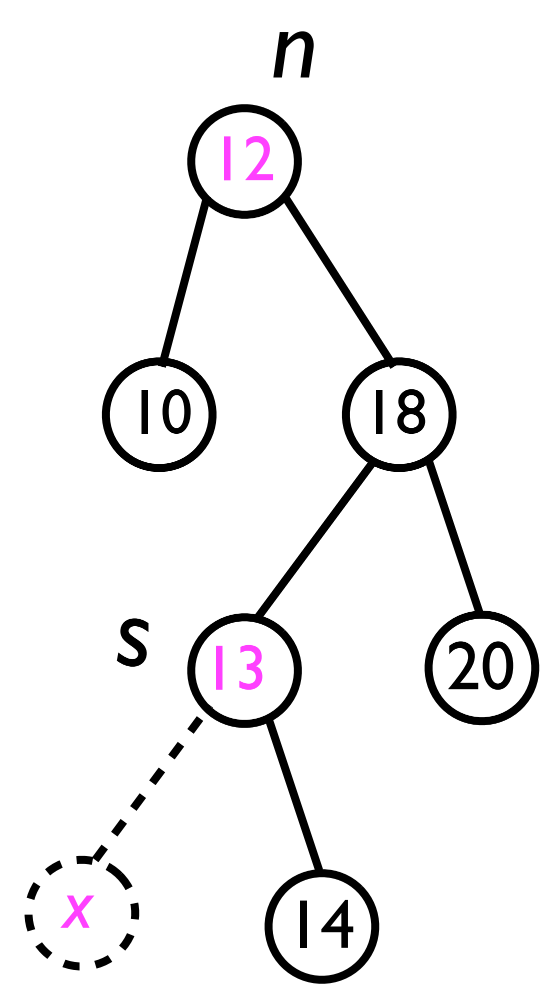
- Example:
  - Let *n* be node 12.
  - Then *n*'s successor *s* is 13.
    - *s* only has one child.
  - Suppose *s* had two children.
    - Then it would have a left child, *x*.
    - *Then x would have to be n's successor.*



Since *x* is still in *n*'s right sub-tree, *x*>12. And since *x* is in *s*'s left sub-tree, *x*<13. So, *x* is *n*'s successor.

Successor of node with two children

- We conclude (by way of contradiction) that, if *n* has two children, then its successor *s* cannot have two children.
- Hence, removing *s* amounts to either just "snipping it off" (case 1), or "slicing around it" (case 2).
- Hence, the **`remove`** method will in fact terminate :-).



`remove(o)`

- We can finally define the **`remove(o)`** method:

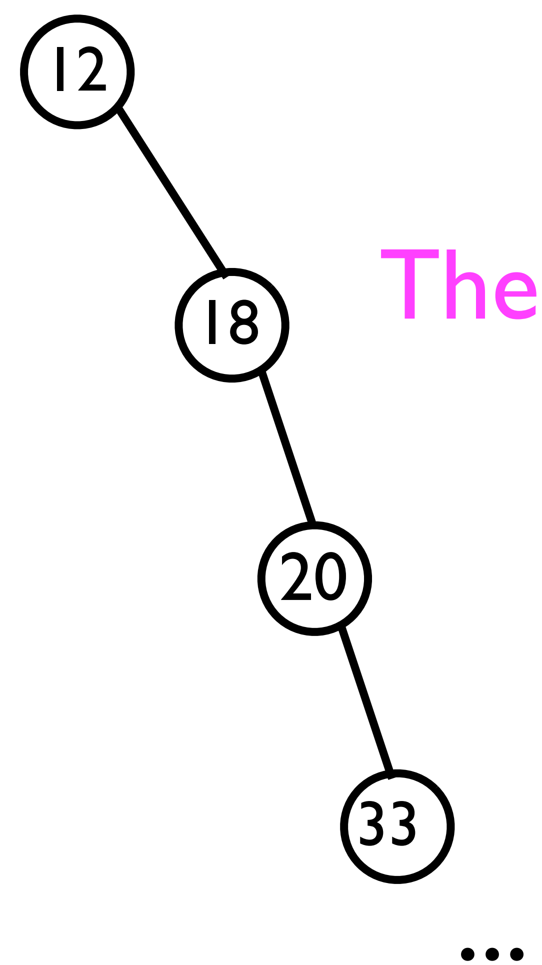
```
void remove (T o) {
  final Node<T> node = findNode(_root, o);
  removeNode(node);
}

void removeNode (Node<T> node) {  // Helper method
  if (node._leftChild == null &&
      node._rightChild == null) {
    // "Snip" node from its parent
  } else if (node._leftChild == null ||
      node._rightChild == null) {
    // "Splice around" node
  } else {
    final Node<T> successor = findSuccessor(_root, o);
    node._data = successor._data;
    removeNode(successor);
  }
}
```

BSTs: Time costs of methods

- All of the fundamental operations -- **`add(o)`**, **`find(o)`**, **`remove(o)`**, and **`findLargest`**/**`findSmallest`** -- take time $O(h)$, where h is the height of the BST.
- In the *average case*, the height h of the BST is $\log n$.
- What about in the *worst case*?

BSTs: Time costs of methods

- In the *worst case*, the user will call **`add`** and **`remove`** in an "unfortunate" order, resulting in a "degenerate" BST of the following variety:
- In this case, the height of the BST is n -- and hence the fundamental BST operations would also be $O(n)$.



The "BST" is just a linked list!

Balancing BSTs

- To prevent this "worst-case" condition from occurring, we need to employ some form of "tree balancing" to keep the tree from degenerating into a linked list.
- Two prominent data structures which ensure a *balanced tree* include:
  - AVL trees.
  - Red-black trees.
- Next lecture...