CSE 12: Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Eleven
18 Aug 2011
Main points from last lecture.
Exploiting relations over data

• Simple data structures such as lists store data without regard to order relations between elements.

• Lists offer $O(1)$ add, but $O(n)$ find and remove operations.

• In many applications, the user will want to find more often than add.

• Even though a user may have “partial knowledge” (key) of an object, it may need the find(o) method to obtain the “whole record” (value).
Exploiting relations over data

• In Java, binary order relations can be defined between pairs of elements using the `Comparable<T>` interface, which includes the `int compareTo (T o)` method.

• Exploiting order relations enables us to achieve superior asymptotic time costs for find/removal operations.

• One prominent example of “accelerating search” is the binary search algorithm:
  
  • Assumes input array is already sorted.
  
  • Achieves $\log_2(n)$ worst-case time cost by recursively dividing the list into two halves and searching only the relevant half.
Recursion

• Recursion is a tool for defining mathematical structures and constructing algorithms.

• Every recursive algorithm/definition contains:
  • A self-referential recursive part; and
  • A base case to prevent circularity.

• Recursive definitions can be specified in formal languages like Backus-Naur Form (BNF) to facilitate automatic generation of code (e.g., “compiler compilers”).
Recursion

- Recursion is ubiquitous in computer science:
  - Binary search executes recursively.
  - The input array to binary search is typically sorted using (recursive) MergeSort or QuickSort.
  - Data structures such as trees, and even linked lists, can be formulated recursively.
    - E.g., “a linked list is either a node, or a node followed by a linked list.”
  - Source code itself is a recursive structure.
  - Compilers “lex” and “parse” the individual symbols/tokens of source code using algorithms generated automatically from recursive definitions of source code structure.
Binary search

- Binary search is a recursive algorithm for finding a target value in a sorted array of data in $\log_2(n)$ (worst-case) time.

- Binary search requires a binary order relation to be defined on all the elements.

- For primitive numeric types like int, double, etc., we can use $>$ or $\geq$ to compare data.

- For objects, we can use the int compareTo(T o) method of the Comparable<T> interface.
Recursive data structures

• Despite the $\log_2(n)$ efficiency of binary search, its utility on lists is limited:

• Binary search would be very efficient on a linked list because of the lack of ability to “jump” to an arbitrary node.

• Binary search on array-based lists is efficient; however, the array must be maintained in sorted order.

• If the user adds a new element, the “correct spot” must be located and all subsequent elements “shifted over”.

Recursive data structures

• It would be desirable to create data structure that offer efficient implementations of both add(o) and find(o)/remove(o).

• Over the next few lectures we will cover two such structures -- heaps and binary search trees.

• Both these ADTs are based on binary trees, which are non-linear recursive data structures.
Binary Trees
Trees

- A tree is an interconnected set of nodes that are organized in a hierarchy.
- There is one node labeled the root of the tree.
- Every node except the root has exactly 1 parent node.
- Each node may have 0 or more child nodes ("children").
- Cycles are prohibited -- only one path may exist between any pair of nodes.
- Parents and children are connected by edges.

Example trees
Trees

- A node with no children is called a *leaf*.
- A node with at least one child is called an *internal node*.
Depth, height, and level

- Depth (iterative definition):
  - The depth of a node \( N \) is the number of edges between \( N \) and the root.
  - The root has depth 0.

- Depth (recursive definition):
  - Base case
    - The depth of a node \( n \) is 0 for the root; or
  - Recursive part
    - \( 1 + \) the depth of \( n \) parent node.
Depth, height, and level

• The *height* of a tree $T$ is the maximum depth of any node in the tree.

• Equivalent to length of longest path from the root to any leaf.

• A *level* of the tree consists of all the nodes at a particular depth.
Sub-trees

• Each node in a tree is the *root* of its own *sub-tree*.

• The gray boxes below show all possible sub-trees.
Binary trees

- A *binary tree* is a tree in which every node has at most 2 children.

Examples of binary trees

Not a binary tree
Binary tree properties

- A binary tree of height $h$ is **full** if every node at depth $d < h$ has 2 children.

Examples of full binary trees

Not a full binary tree
Binary tree properties

• A full binary tree with height $h$ has $2^h$ leaf nodes and $2^{h+1} - 1$ nodes in total.

• Conversely, a full binary tree with $n$ nodes total has height $\log_2(n+1) - 1$. 
Binary tree properties

• More generally, a binary tree $T$ (not necessarily full) with $n$ nodes has:

  • Minimum height $\log_2(n+1) - 1$ (when $T$ is full).
  
  • Maximum height $n - 1$ (when $T$ is just a “chain” of nodes in which no node has more than 1 child).

• Why important?

  • The time cost of important tree operations such as $\text{find}(o)$ depend on the average/maximum height of an arbitrary node in the tree.
Tree nodes

- Like nodes in a linked list, nodes in a tree contain a *data element* (otherwise, trees would be useless for ADTs).

- However, nodes in a tree contain more than 2 “links” (edges) to other nodes.
  - One link to parent node.
  - One link to each child node.
Node class for general trees

- From this description, we can create a `Node` class for use in general trees:

```java
class Node<T> {
    Node<T> _parent;  // link to parent node
    Node<T>[] _children;  // links to children
    int _numChildren;
    T _data;  // data element the node stores
}
```

- Alternatively, we can use a linked list to manage the child nodes:

```java
class Node<T> {
    Node<T> _parent;  // link to parent node
    LinkedList<T> _children;  // links to children
    T _data;  // data element the node stores
}
```
Node class for binary trees

- From binary trees, we can define a Node more simply:

```java
class Node<T> {
    Node<T> _parent;
    Node<T> _leftChild, _rightChild;
    T _data; // Defined to be null if child does not exist.
}
```

- We can then begin creating nodes and assembling a tree:

```java
final Node<String> root = new Node<String>();
root._leftChild = new Node<String>();
root._rightChild = new Node<String>();
root._rightChild._leftChild = new Node<String>();
```
Node class for binary trees

• From binary trees, we can define a Node more simply:

```java
class Node<T> {
    Node<T> _parent;
    Node<T> _leftChild, _rightChild;
    T _data;
}
```

• We can then begin creating nodes and assembling a tree:

```java
final Node<String> root = new Node<String>();
root._leftChild = new Node<String>();
root._rightChild = new Node<String>();
root._rightChild._leftChild = new Node<String>();
```
Node class for binary trees

• From binary trees, we can define a Node more simply:

```java
class Node<T> {
    Node<T> _parent;
    Node<T> _leftChild, _rightChild;
    T _data;
}
```

• We can then begin creating nodes and assembling a tree:

```java
final Node<String> root = new Node<String>();
root._leftChild = new Node<String>();
root._rightChild = new Node<String>();
root._rightChild._leftChild = new Node<String>();
```
Node class for binary trees

• From *binary* trees, we can define a *Node* more simply:

```java
class Node<T> {  
    Node<T> _parent;
    Node<T> _leftChild, _rightChild;
    T _data;
}
```

• We can then begin creating *Nodes* and assembling a tree:

```java
final Node<String> root = new Node<String>();
root._leftChild = new Node<String>();
root._rightChild = new Node<String>();
root._rightChild._leftChild = new Node<String>();
```
Node class for binary trees

- From *binary* trees, we can define a `Node` more simply:

  ```java
  class Node<T> {
    Node<T> _parent;
    Node<T> _leftChild, _rightChild;
    T _data;
  }
  ```

- We can then begin creating `Node`es and assembling a tree:

  ```java
  final Node<String> root = new Node<String>();
  root._leftChild = new Node<String>();
  root._rightChild = new Node<String>();
  root._rightChild._leftChild = new Node<String>();
  ```

 ![Diagram of a binary tree with three nodes]
Tree operations

- We will consider three fundamental operations:

  - `add (o, parent, leftOrRight)` -- add a new node (containing the object `o`) as the `leftOrRight` child of the specified parent.

  - `find (o)` -- find and return the node containing data `o`.

  - `remove (o)` -- remove the node containing the specified data.

- Note that these operations will be used *internally* by ADTs we develop *based on* trees.

  - This is why we find and return the *node* instead of the data contained *inside* the node.

  - They will *not* be exposed to the user of, say, the Heap ADT, which is built using a binary tree.
Adding a node

- Given the parent node, it is straightforward to add a new node that is either the left or right child of the parent:

```java
void add (T o, Node<T> parent,
          ChildType leftOrRight) {
    final Node<T> node = new Node<T>();
    node._data = o;
    if (leftOrRight == ChildType.LEFT_CHILD) {
      parent._leftChild = node;
    } else {
      parent._rightChild = node;
    }
}
```
Adding a node

- Given the parent node, it is straightforward to add a new node that is either the left or right child of the parent:

```java
void add (T o, Node<T> parent, ChildType leftOrRight) {
    final Node<T> node = new Node<T>();
    node._data = o;
    if (leftOrRight == ChildType.LEFT_CHILD) {
        parent._leftChild = node;
    } else {
        parent._rightChild = node;
    }
}
```

A Java enumeration type.
Java enumerations

- Enumerations are types that contain only a few possible values.
- Each value in the enumeration can be given a meaningful name.
- If we define an enumeration type called `ChildType`:
  ```java
enum ChildType {
    LEFT_CHILD, RIGHT_CHILD
  }
```
  
  - ...then we can declare and use a variable of that type:
    ```java
    ChildType leftOrRight = ChildType.RIGHT_CHILD;
    ```
Java enumerations

• Instead of defining an enumeration type, one could instead just use an integer and “assign” meaning to these values, e.g.:

```java
int leftOrRight;
leftOrRight = 1;  // 1 indicates left child
leftOrRight = 2;  // 2 indicates right child
...
if (leftOrRight == 2) {
    // Do something with the right child
}
```

• But what if `leftOrRight` was somehow set to an invalid value?

• With enumerations, the Java compiler prevents this possibility from ever happening.

```java
ChildType leftOrRight = 3;  // Won’t compile
```
Finding a node

- Finding a node in a binary tree is best implemented using recursion. We’ll let node represent the root of the sub-tree we are currently searching.

```java
Node<T> find (Node<T> root, T o) {
    if (root._data.equals(o)) {
        return root;
    }

    Node<T> node;
    if (_leftChild != null &&
        (node = find(_leftChild, o)) != null) {
        return node;
    } else if (_rightChild != null &&
        (node = find(_rightChild, o)) != null) {
        return node;
    } else {
        return null;
    }
}
```

Combined assignment to node and comparison to null. This is compact notation, but it sometimes can also yield more readable code.
Finding a node

• Watch how the method works for `find(a, "e")`:

    root:a
    Node<T> find (Node<T> root, T o) {
        if (root._data.equals(o)) {
            return root;
        }
        Node<T> node;
        if (_leftChild != null &&
            (node = find(_leftChild, o)) != null) {
            return node;
        } else if (_rightChild != null &&
            (node = find(_rightChild, o)) != null) {
            return node;
        } else {
            return null;
        }
    }
Finding a node

- Watch how the method works for `find(a, "e")`:

```java
Node<T> find (Node<T> root, T o) {
    if (root._data.equals(o)) {
        return root;
    }
    Node<T> node;
    if (_leftChild != null &&
        (node = find(_leftChild, o)) != null) {
        return node;
    } else if (_rightChild != null &&
        (node = find(_rightChild, o)) != null) {
        return node;
    } else {
        return null;
    }
}
```
Finding a node

- Watch how the method works for `find(a, "e")`:

```java
Node<T> find(Node<T> root, T o) {
    if (root._data.equals(o)) {
        return root;
    }
    Node<T> node;
    if (_leftChild != null &&
        (node = find(_leftChild, o)) != null) {
        return node;
    } else if (_rightChild != null &&
        (node = find(_rightChild, o)) != null) {
        return node;
    } else {
        return null;
    }
}
```
Finding a node

- Watch how the method works for \texttt{find(a, "e")}:

```java
root:b

Node<T> find (Node<T> root, T o) {
    if (root._data.equals(o)) {
        return root;
    }
    Node<T> node;
    if (_leftChild != null &&
        (node = find(_leftChild, o)) != null) {
        return node;
    } else if (_rightChild != null &&
        (node = find(_rightChild, o)) != null) {
        return node;
    } else {
        return null;
    }
}
```
Finding a node

- Watch how the method works for `find(a, "e")`:

```java
root:b
Node<T> find (Node<T> root, T o) {
    if (root._data.equals(o)) {
        return root;
    }
    Node<T> node;
    if (_leftChild != null && (node = find(_leftChild, o)) != null) {
        return node;
    } else if (_rightChild != null && (node = find(_rightChild, o)) != null) {
        return node;
    } else {
        return null;
    }
}
```
Finding a node

- Watch how the method works for `find(a, "e")`:

```java
Node<T> find (Node<T> root, T o) {
    if (root._data.equals(o)) {
        return root;
    }
    Node<T> node;
    if (_leftChild != null &&
        (node = find(_leftChild, o)) != null) {
        return node;
    } else if (_rightChild != null &&
        (node = find(_rightChild, o)) != null) {
        return node;
    } else {
        return null;
    }
}
```
Finding a node

• Watch how the method works for `find(a, "e")`:

```java
Node<T> find (Node<T> root, T o) {
    if (root._data.equals(o)) {
        return root;
    }
    Node<T> node;
    if (_leftChild != null &&
        (node = find(_leftChild, o)) != null) {
        return node;
    } else if (_rightChild != null &&
               (node = find(_rightChild, o)) != null) {
        return node;
    } else {
        return null;
    }
}
```
Finding a node

• Watch how the method works for `find(a, "e")`:

```java
Node<T> find(Node<T> root, T o) {
    if (root._data.equals(o)) {
        return root;
    }

    Node<T> node;
    if (_leftChild != null &&
        (node = find(_leftChild, o)) != null) {
        return node;
    } else if (_rightChild != null &&
        (node = find(_rightChild, o)) != null) {
        return node;
    } else {
        return null;
    }
}
```
Finding a node

- Watch how the method works for find(a, “e”):

```java
Node<T> find (Node<T> root, T o) {
    if (root._data.equals(o)) {
        return root;
    }
    Node<T> node;
    if (_leftChild != null &&
        (node = find(_leftChild, o)) != null) {
        return node;
    } else if (_rightChild != null &&
        (node = find(_rightChild, o)) != null) {
        return node;
    } else {
        return null;
    }
}
```
Finding a node

- Watch how the method works for `find(a, "e")`:

```java
Node<T> find (Node<T> root, T o) {
    if (root._data.equals(o)) {
        return root;  // YES!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    }
    Node<T> node;
    if (_leftChild != null &&
        (node = find(_leftChild, o)) != null) {
        return node;
    } else if (_rightChild != null &&
        (node = find(_rightChild, o)) != null) {
        return node;
    } else {
        return null;
    }
}
```
Finding a node

• Watch how the method works for \texttt{find(a, “e”)}:

\begin{verbatim}
Node<T> find (Node<T> root, T o) {
    if (root._data.equals(o)) {
        return root;  // The returned node will “propagate back up” the recursive calls.
    }
    Node<T> node;
    if (_leftChild != null &&
        (node = find(_leftChild, o)) != null) {
        return node;
    } else if (_rightChild != null &&
        (node = find(_rightChild, o)) != null) {
        return node;
    } else {
        return null;
    }
}
\end{verbatim}
Finding a node

• Watch how the method works for `find(a, "e")`:

```java
Node<T> find (Node<T> root, T o) {
    if (root._data.equals(o)) {
        return root;
    }
    Node<T> node;
    if (_leftChild != null &&
        (node = find(_leftChild, o)) != null) {
        return node;
    } else if (_rightChild != null &&
        (node = find(_rightChild, o)) != null) {
        return node;
    } else {
        return null;
    }
}
```
Finding a node

• Watch how the method works for `find(a, "e")`:

```java
Node<T> find (Node<T> root, T o) {
    if (root._data.equals(o)) {
        return root;
    }
    Node<T> node;
    if (_leftChild != null &&
        (node = find(_leftChild, o)) != null) {
        return node;
    } else if (_rightChild != null &&
        (node = find(_rightChild, o)) != null) {
        return node;
    } else {  
        return null;
    }
}
```
Finding a node

- Watch how the method works for \texttt{find(a, \textquote{e})}:

\begin{verbatim}
Node<T> find (Node<T> root, T o) {
    if (root._data.equals(o)) {
        return root;
    }
    Node<T> node;
    if (_leftChild != null &&
        (node = find(_leftChild, o)) != null) {
        return node;
    } else if (_rightChild != null &&
        (node = find(_rightChild, o)) != null) {
        return node;
    } else {
        return null;
    }
}
\end{verbatim}

\[
\text{Done!!!!!!!!!!!!!!!!!!!!}
\]

Thursday, August 18, 2011
Removing a node

• How to implement the `remove(o)` operation depends on whether the node containing `o` is a leaf node or an internal node.

• We can use the find method to locate the correct node.

• If the node is a leaf, then we just “snip” it off from its parent, e.g.:

```cpp
node._parent._rightChild = null;
```
Removing a node

• How to implement the `remove(o)` operation depends on whether the node containing `o` is a *leaf* node or an *internal* node.

• We can use the `find` method to locate the correct node.

• If the node is a leaf, then we just “snip” it off from its parent, e.g.:

  ```
  node._parent._rightChild = null;
  ```
Removing a node

- If, however, the node is an internal node, then “snipping” it off would remove the whole sub-tree.
- To just remove the node but not its children, we need to replace the internal node with some other node.
- Instead of actually removing and replacing $n$, we can instead just replace the data it stores with the data of another leaf node (e.g., 1).
- We can then remove the “old” 1.
Removing a node

• If, however, the node is an internal node, then “snipping” it off would remove the whole sub-tree.

• To just remove the node but not its children, we need to replace the internal node with some other node.

• Instead of actually removing and replacing n, we can instead just replace the data it stores with the data of another leaf node (e.g., 1).

• We can then remove the “old” 1.
Array-based binary trees.
Array-based binary trees

- Just as lists can be implemented by either a linked chain of nodes or an array, a binary tree can be implemented as a tree of nodes or an array as well.

- Each “node” in the tree will be assigned a unique index at which its data should be stored.

- Given the index of a particular “node”, the index of its parent, and the indices of its children, can be easily computed.
Array-based binary trees

- The index($n$) of a node $n$ with parent $p$ is:
  - 0 if $n$ is the root node.
  - $2 \times \text{index}(p) + 1$ if $n$ is left child of $p$.
  - $2 \times \text{index}(p) + 2$ if $n$ is right child.

- The parentIndex(idx) of a node stored at idx is $(\text{idx}-1)/2$.

- Examples:
  \[
  \text{index}(c) = 2 \times \text{index}(a) + 2 = 2 \times 0 + 2 = 1 \\
  \text{parentIndex}(4) = (4-1)/2 = 1.5 = 1.
  \]
Array-based binary trees

- Note that this array-based representation applies only to complete binary trees.

- A binary tree is complete if every level of the tree is completely filled except possibly the last and the last level is (partially) filled from left to right.
Array-based binary trees

• Even though the data are being stored in a regular Java array, *their locations in the array still encode a tree structure among them.*

• This means that binary tree-based algorithms we develop can still offer time-cost advantages over linear lists.
Adding a node

- Given that the binary tree must be complete, it is only valid to add a node \( n \) to be the next child on the last level of the tree.

- The index into the array of where this “next child” should be stored is always just \_numNodes\, where \_numNodes\ is the current number of nodes in the tree.
Adding a node

• Given that the binary tree must be complete, it is only valid to add a node \( n \) to be the next child on the last level of the tree.

• The index into the array of where this “next child” should be stored is always just \(_\text{numNodes}\), where \(_\text{numNodes}\) is the current number of nodes in the tree.
Removing a node

- Similarly, it is only valid to remove the right-most child of the last level of the tree.

- All we must do is decrement \_numNodes to indicate that the “slot” in the array of the removed node is no longer valid.

\_numNodes: 5
Removing a node

- Similarly, it is only valid to remove the right-most child of the last level of the tree.
- All we must do is decrement _numNodes to indicate that the “slot” in the array of the removed node is no longer valid.
Finding a node

- To find the index of a node \( n \) whose data element equals \( o \):

```java
int find (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;
    }
}
```

Make sure each child exists before recursing!

Helper methods to determine index of left and right child nodes.
Binary trees to accelerate search.
Binary trees to accelerate search

• We have now constructed considerable “infrastructure” for building binary trees, using either “linked nodes” or a Java array for the tree’s underlying storage.

• Trees are useful in their own right for representing hierarchical structures, e.g., genealogical data.

• However, for the moment we are interested in how they can store and accelerate search of data on which an ordering relation is defined.
Binary trees to accelerate search

• *Heaps* and *binary search trees* are two ADTs based on binary trees that accelerate search.

• A heap offers fast access to the largest element in a collection of related objects.
  
  • $O(1)$ worst-case time cost for `findLargest`.
  
  • $O(\log n)$ worst-case time cost for `removeLargest`.
  
  • $O(\log n)$ worst-case time cost for `add`.
  
  • $O(n)$ worst-case time-cost for `find and remove`.
Binary trees to accelerate search

- A binary search tree (BST) offers:
  - $O(\log n)$ average-case time cost for add, find, remove, and findLargest.
  - $O(n)$ worst-case time cost for add, find, remove, and findLargest.
- AVL trees and red-black trees are more complicated, but they offer:
  - $O(\log n)$ worst-case time cost for add, find, remove, and findLargest.
Why findLargest?

• Why would we want to find the largest data element stored in a container?

• The findLargest method is required by priority queues.

• A priority queue is a queue in which elements are dequeued not in FIFO order, but instead in order of highest-to-lowest priority.

• A priority queue is typically implemented using a heap.

Taken from Paul Kube’s slides.
Heaps.
Heaps

- A *max-heap* is an ADT for storing data so that the *largest element* (according to some binary order relation) can always be found and removed quickly.

- A *min-heap* is defined analogously for the *smallest element*.

- Internally, a *heap* is a *complete* binary tree which satisfies the *heap condition*:
  - The root of every sub-tree is *no smaller than any node in the sub-tree*. (For *max-heap*).
  - The root of every sub-tree is *no larger than any node in the sub-tree*. (For *min-heap*).

- This ensures that, to implement findLargest/findSmallest, *we can always just return the root node of the tree*. 

Heaps

• A max-heap has the following interface:

```java
// All operations must preserve the heap condition.
interface MaxHeap {
    // Adds o to the heap.
    void add (T o);
    // Removes the node whose data element equals o.
    void remove (T o);
    // Removes and returns the largest node in the heap.
    T removeLargest ();
    // Returns the largest node in the heap.
    T findLargest ();
    // Finds and returns the node whose data element
    // equals o.
    T find (T o);
    // Returns the number of data stored in the heap.
    int size ();
}
```
Implementing heaps

• Since heaps are anyway a complete binary tree, it is convenient and efficient to implement them using an array.

• However, they could also be implemented using linked nodes.

• The challenge when implementing a heap is to preserve the heap property upon every mutation to the heap (add/remove).
Adding a node to a heap

• In order to add a new element o to a max-heap while preserving the heap condition, we execute the following procedure:

  • Add a new node \( n \) containing o to the last level of the tree (ensure completeness of the tree).
    
  • This may violate the tree’s heap condition because o may be larger than one of its parents.

  • We then “fix” the heap by “swapping” node \( n \) with its parent \( p \).

  • We repeat this process -- known as bubbling up -- as many times as necessary until the tree is a heap again.
Adding a node to a heap

- Consider the heap to the right. (Notice that it satisfies the *heap condition*).
Adding a node to a heap

• Consider the heap to the right. (Notice that it satisfies the heap condition).

• Suppose we add value 8 to the bottom-level of the heap.

• The tree no longer satisfies the heap condition.

2 is smaller than one of the nodes in its sub-tree!
Adding a node to a heap

- Consider the heap to the right. (Notice that it satisfies the *heap condition*).
- Suppose we add value 8 to the bottom-level of the heap.
  - The tree no longer satisfies the heap condition.
  - We have to “bubble up” the 8 we just added to restore the heap condition.
Adding a node to a heap

- Consider the heap to the right. (Notice that it satisfies the heap condition).
- Suppose we add value 8 to the bottom-level of the heap.
- The tree no longer satisfies the heap condition.
- We have to “bubble up” the 8 we just added to restore the heap condition.

Not done yet -- 5 is still smaller than 8.
Adding a node to a heap

- Consider the heap to the right. (Notice that it satisfies the *heap condition*).

- Suppose we add value 8 to the bottom-level of the heap.

  - The tree no longer satisfies the heap condition.

  - We have to “bubble up” the 8 we just added to restore the heap condition.

Now it is a heap again!
Adding a node to a heap

- Consider the heap to the right. (Notice that it satisfies the heap condition).

- Suppose we add value 8 to the bottom-level of the heap.
  - The tree no longer satisfies the heap condition.
  - We have to “bubble up” the 8 we just added to restore the heap condition.
  - Done!
Adding a node to a heap

• We can implement the \texttt{add(o)} method as:
  
  ```java
  void add (T o) {
    _nodeArray[_numNodes] = o;
    _numNodes++;
    bubbleUp(_numNodes - 1);
  }
  ```

• We must then also implement \texttt{bubbleUp(idx)}:
  
  ```java
  void bubbleUp (int idx) {
    If node at idx is "larger" than its parent:
      Swap data in the node and its parent;
      Recursively bubbleUp(parentIdx(idx));
  }
  ```
Adding a node to a heap

- Alternatively, we can write an *iterative* version of `bubbleUp(idx)):

```java
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);
}
```
Next lecture

• Finding and removing elements.
• “Trickling down” a node.