CSE 12:
Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Ten
23 July 2012
Linear data structures: asymptotic time costs

• Let’s review the “score card” of the ADTs we’ve covered so far.

• Let’s consider three fundamental operations:
  • `void add (T o);`
  • `void remove (T o);`
  • `T find (T o);`
    Search for an element in the container that equals \( o \) and returns it; if no such object exists, then returns `null`. 
# Array-list and linked-list scorecard

<table>
<thead>
<tr>
<th>Operation (o)</th>
<th>Array-list</th>
<th>Linked-list</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(o)</code></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>find(o)</code></td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><code>remove(o)</code></td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Array-list and linked-list scorecard

• There are many occasions where the user will *add* new data relatively *rarely*, but want to *find* data already in the data structure relatively *frequently*.

• In order to improve the asymptotic time cost of the *find(o) and remove(o)* operations, we will make use of order relationships between data elements.

• Once we’ve *found* an element within a data structure, it is typically easy for the data structure to *remove* it.
Why find something?

• It may strike some as odd that an ADT would support the method $T \text{find}(T \ o)$.

• After all, if the user knows the object $o$ he/she is looking for, then why call $\text{find}$ at all?

• Answer: sometimes the user knows part of the information about an object $o$, but does not have the whole record.

• This illustrates the difference between a record’s key and its value.
Keys and values

• The part of the `Student` object that the user always knows is called the `key` (e.g., student ID number at Student Health).

• The rest of the `Student` record is called the `value`.

```java
class Student {
    String _studentID; // Key
    String _firstName, _lastName; // Value
    String _address;

    Student (String studentID) {
        _studentID = studentID;
    }
    Student (String studentID, String firstName, String lastName, String address) {
        _studentID = studentID;
        _firstName = firstName;
        _lastName = lastName;
        _address = address;
    }
}
```
Keys and values

• The user may store many Student objects inside a List12 container, e.g.:

```java
list.add(new Student("A123", "Bill", "Carter", "123 Main St");
list.add(new Student("A213", "Jimmy", "Clinton", "124 Main St");
...
list.add(new Student("B092", "Hillary", "Nixon", "125 Main St");
```

• Later, the user may wish to find a particular Student object using just the key, e.g., the student ID:

```java
final Student cse12Student = list.find(new Student("A123");
```

Student containing both the key and value.  
Student initialized with just the key.
Finding a particular key

• Given a request to find a particular key, and given that keys often have an order relation defined between them, it seems silly to search through the container as if the keys were all unrelated.

• Example: Suppose we are searching for the student ID “c237”. Do we really need to start at the very beginning?

No -- the natural order among keys imposes structure on the “search problem” that lets us find a particular key much more quickly.
Binary order relations

• An example of a binary order relationship is the Java < operator, e.g.:

```java
int a = 3, b = 4;
if (a < b) {
    ...
}
```

• However, the < operator is only valid on primitive numeric variables (int, float, double, etc.).
Binary order relations

- More generally, two Java Objects can be compared if they are Comparable, using the compareTo method:
  ```java
  int compareTo (T o);
  ```

- o1.compareTo(o2) is:
  - < 0 if o1 is “less than” o2
  - == 0 if o1 is “equal to” o2
  - > 0 if o1 is “greater than” o2

- Classes that implement the compareTo(o) method can implement the Comparable<T> interface.
Comparable<T>

- Example:

```java
class Student implements Comparable<Student> {
    ...
    int compareTo (T other) {
        return _studentID.compareTo(
            other._studentID
        );
    }
}
```

In this particular case, we can just delegate to the `String.compareTo(o)` method, since `String` implements `Comparable<String>`.
Faster search using recursion
Searching a sorted list

- How will defining this “ordering relation” using Comparable<T> help us to find a key more quickly?
- Let’s consider a simpler example in which we wish to find an integer within a sorted list of numbers.
- We will implement a method

```java
int search (int[] numbers, int targetNum, int startIdx, int endIdx);
```

which will search through an array of numbers, starting at the startIdx and ending at the endIdx, looking for the targetNum.
Searching a sorted list

• Consider the following example:

```java
search(numbers, targetNum, startIdx, endIdx):
```

where

```java
int targetNum = 79;
int startIdx = 0;
int endIdx = 15;

int[] numbers = {
    16, 26, 31, 40, 43, 45, 51, 55, 58, 67, 69, 73, 79, 87, 88
};
```

• What is the optimal search strategy given that numbers is already sorted?
Binary search

- The optimal search strategy (minimum time cost) for a list of sorted elements is binary search.
- The search is binary because we repeatedly divide the list into 2 pieces.

Search algorithm:

```java
Pick a guessIdx = (startIdx + endIdx) / 2;
if (numbers[guessIdx] == targetNum) {
    return guessIdx;
} else if (numbers[guessIdx] < targetNum) {
    Search the “right half” of the list for targetNum.
} else {
    Search the “left half” of the list for targetNum.
}
```
• Let's look for targetNum=79.

• Search algorithm:
  Pick a guessIdx = (startIdx + endIdx) / 2;
  if (numbers[guessIdx] == targetNum) {
    return guessIdx;
  } else if (numbers[guessIdx] < targetNum) {
    Search the “right half” of the list for targetNum.
  } else {
    Search the “left half” of the list for targetNum.
  }
Binary search

• Let’s look for targetNum=79.

• Search algorithm:
  Pick a guessIdx = (startIdx + endIdx) / 2;
  if (numbers[guessIdx] == targetNum) {
    return guessIdx;
  } else if (numbers[guessIdx] < targetNum) {
    Search the “right half” of the list for targetNum.
  } else {
    Search the “left half” of the list for targetNum.
  }

16, 26, 31, 40, 43, 45, 51, 55, 58, 67, 69, 73, 79, 87, 88
Binary search

- Let’s look for targetNum=79.

- Search algorithm:
  
  Pick a guessIdx = (startIdx + endIdx) / 2;
  if (numbers[guessIdx] == targetNum) {
      return guessIdx;
  } else if (numbers[guessIdx] < targetNum) {
      Search the “right half” of the list for targetNum.
  } else {
      Search the “left half” of the list for targetNum.
  }

16, 26, 31, 40, 43, 45, 51, 55, 58, 67, 69, 73, 79, 87, 88
Binary search

- Let’s look for targetNum=79.

- Search algorithm:
  
  \[
  \text{Pick a guessIdx} = \left(\text{startIdx} + \text{endIdx}\right) / 2;
  \]
  
  \[
  \text{if (numbers[guessIdx] == targetNum) } \{
      \text{return guessIdx;}
  \}
  \]
  
  \[
  \text{else if (numbers[guessIdx] < targetNum) } \{
      \text{Search the “right half” of the list for targetNum.}
  \}
  \]
  
  \[
  \text{else } \{
      \text{Search the “left half” of the list for targetNum.}
  \}
  \]
Binary search

- Let's look for targetNum=79.

- Search algorithm:
  
  Pick a guessIdx = (startIdx + endIdx) / 2;
  if (numbers[guessIdx] == targetNum) {
    return guessIdx;
  } else if (numbers[guessIdx] < targetNum) {
    Search the “right half” of the list for targetNum.
  } else {
    Search the “left half” of the list for targetNum.
  }
Binary search

- Let's look for targetNum=79.

- Search algorithm:
  
  ```java
  Pick a guessIdx = (startIdx + endIdx) / 2;
  if (numbers[guessIdx] == targetNum) {
    return guessIdx;
  } else if (numbers[guessIdx] < targetNum) {
    Search the "right half" of the list for targetNum.
  } else {
    Search the "left half" of the list for targetNum.
  }
  ```
Binary search

• Let’s look for `targetNum=79`.

• Search algorithm:

```java
Pick a guessIdx = (startIdx + endIdx) / 2;
if (numbers[guessIdx] == targetNum) {
   return guessIdx;
}
else if (numbers[guessIdx] < targetNum) {
   Search the “right half” of the list for targetNum.
}
else {
   Search the “left half” of the list for targetNum.
}
```

Done in 4 guesses!

16, 26, 31, 40, 43, 45, 51, 55, 58, 67, 69, 73, 79, 87, 88
Binary search and recursion

• Binary search is a classic example of a recursive algorithm:

  • The algorithm makes repeated calls to itself to get its work done, e.g.:
    “Search algorithm:
    ...
    Search the “right half” of the list for targetNum.
  ”

• Each recursive call operates on a smaller problem than the original (e.g., it searches only half the list).

• Eventually, the algorithm operates on a trivial input size (e.g., a list of 1 element) and terminates.
Let’s return to our example of searching through an array `numbers` of sorted integers for a particular `targetNum`.

Search algorithm:

```java
// Assume targetNum is always somewhere inside numbers
int search (int[] numbers, int targetNum, int startIdx, int endIdx) {
    int guessIdx = (startIdx + endIdx) / 2;
    if (numbers[guessIdx] == targetNum) {
        return guessIdx; // Base case
    } else if (numbers[guessIdx] < targetNum) {
        return search(numbers, targetNum, guessIdx+1, endIdx);
    } else {
        return search(numbers, targetNum, startIdx, guessIdx-1); // Recursive part
    }
}
```
Binary search and recursion

• The worst-case time cost of binary search depends on how many times the list can be divided in half.

• int length = endIdx - startIdx + 1; // 16

16
8
4
2
1

divide in half
divide in half
divide in half
Binary search and recursion

• The worst-case time cost of binary search depends on how many times the list can be divided in half.

• int length = endIdx - startIdx + 1;  // 16

16
  \[ \text{divide in half} \]
  \[ 8 \]
  \[ \text{divide in half} \]
  \[ 4 \]
  \[ \text{divide in half} \]
  \[ 2 \]
  \[ \text{divide in half} \]
  \[ 1 \]

\[ \log_2 16 = 4 \text{ times} \]

• If the list has n elements, then binary search has a worst-case time cost of \( O(\log n) \).

• Huge improvement over \( O(n) \).
Binary search and objects

• What if we want to execute binary search on a list of objects?

• This is easy if the objects are Comparable.

• Search algorithm (to find object o):
  Pick a guessIdx = (startIdx + endIdx) / 2;
  if (objects[guessIdx].compareTo(o) == 0) {
    return guessIdx;
  } else if (objects[guessIdx].compareTo(o) < 0) {
    Search the “right half” of the list for targetNum.
  } else {
    Search the “left half” of the list for targetNum.
  }
Binary search and objects

• What if we want to execute binary search on a list of objects?

• This is easy if the objects are Comparable.

• Search algorithm (to find object o):

  o=Priscilla

  Pick a guessIdx = (startIdx + endIdx) / 2;
  if (objects[guessIdx].compareTo(o) == 0) {
    return guessIdx;
  } else if (objects[guessIdx].compareTo(o) < 0) {
    Search the “right half” of the list for targetNum.
  } else {
    Search the “left half” of the list for targetNum.
  }
Binary search and objects

• What if we want to execute binary search on a list of objects?

• This is easy if the objects are Comparable.

• Search algorithm (to find object o):

  Pick a guessIdx = (startIdx + endIdx) / 2;
  if (objects[guessIdx].compareTo(o) == 0) {
    return guessIdx;
  } else if (objects[guessIdx].compareTo(o) < 0) {
    Search the “right half” of the list for targetNum.
  } else {
    Search the “left half” of the list for targetNum.
  }

Albert  Bertha  Cherry  Doris  Egbert  Franklin  Gertrude  Humphrey  Mo  Nancy  Oliver  Priscilla  Wally  Xavier  Yusuf  Zachary
Binary search and objects

• What if we want to execute binary search on a list of objects?

• This is easy if the objects are Comparable.

• Search algorithm (to find object o):

  Pick a guessIdx = (startIdx + endIdx) / 2;
  if (objects[guessIdx].compareTo(o) == 0) {
    return guessIdx;
  } else if (objects[guessIdx].compareTo(o) < 0) {
    Search the “right half” of the list for targetNum.
  } else {
    Search the “left half” of the list for targetNum.
  }

Albert
Bertha
Cherry
Doris
Egbert
Franklin
Gertrude
Humphrey
Mo
Nancy
Oliver
Priscilla
Wally
Xavier
Yusuf
Zachary
Binary search and objects

• What if we want to execute binary search on a list of objects?

• This is easy if the objects are Comparable.

• Search algorithm (to find object o):

  Pick a guessIdx = (startIdx + endIdx) / 2;
  if (objects[guessIdx].compareTo(o) == 0) {
    return guessIdx;
  } else if (objects[guessIdx].compareTo(o) < 0) {
    Search the “right half” of the list for targetNum.
  } else {
    Search the “left half” of the list for targetNum.
  }
Binary search and objects

• What if we want to execute binary search on a list of objects?

• This is easy if the objects are Comparable.

• Search algorithm (to find object o):
  Pick a guessIdx = (startIdx + endIdx) / 2;
  if (objects[guessIdx].compareTo(o) == 0) {
    return guessIdx;
  } else if (objects[guessIdx].compareTo(o) < 0) {
    Search the “right half” of the list for targetNum.
  } else {
    Search the “left half” of the list for targetNum.
  }
Sorting and recursion

• Recall, however, that binary search requires the list to have been *already* sorted.

• How was this accomplished?

• It turns out that the fastest sorting algorithms are implemented using *recursion*:

• For instance, the MergeSort algorithm (next week) successively divides a list of ordered elements into two halves, sorts them separately, and then combines the results.
Data structures and recursion

• Even though a sorted list of data is useful, what happens if we want to add more data into the list? How do we keep the data in sorted order?

• Using a list in these cases will be inefficient.

• More efficient is a tree-based data structure.

  • Trees are non-linear data structures because each element may be adjacent to more than 2 other elements.

  • Trees are recursive data structures -- each “branch” of a tree forms a “tree” in itself.
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

Root node

Empty tree
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, height, and level

• Depth (recursive definition):
  • The depth of a node $n$ is 0 for the root; or
  • $1 +$ the depth of $n$’s parent node.

Base case
Recursive part
• 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
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.

Complete

Complete

Not complete
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 (**not** for P3!):

  ```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>();
```

Monday, July 23, 12
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`s 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>();
  ```

```
```
```
Tree operations

• We will consider two 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.

• 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,
         boolean isLeftChild) {
    final Node<T> node = new Node<T>();
    node._data = o;
    if (isLeftChild) {
        parent._leftChild = node;
    } else {
        parent._rightChild = node;
    }
}
```
Finding a node

• Finding a node in a binary tree is best implemented using recursion. We’ll let root 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 (root._leftChild != null &&
        (node = find(root._leftChild, o)) != null) {
        return node;
    } else if (root._rightChild != null &&
        (node = find(root._rightChild, o)) != null) {
        return node;
    } else {
        return null;
    }
}
```
Finding a node

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

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

• Watch how the method works for $\text{find}(a, \ "e")$:

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

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

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

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

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

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

```java
Node<T> find (Node<T> root, T o) {
    if (root._data.equals(o)) {
        return root; // YES!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    }
    Node<T> node;
    if (root._leftChild != null &&
        (node = find(root._leftChild, o)) != null) {
        return node;
    } else if (root._rightChild != null &&
        (node = find(root._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;  // The returned node will "propagate back up" the recursive calls.
    }
    Node<T> node;
    if (root._leftChild != null &&
        (node = find(root._leftChild, o)) != null) {
        return node;
    } else if (root._rightChild != null &&
        (node = find(root._rightChild, o)) != null) {
        return node;
    } else {
        return null;
    }
}
```
Finding a node

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

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

Done!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!
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:
  - index(c) = $2 \times \text{index}(a) + 2 = 2 \times 0 + 2 = 1$
  - parentIndex(4) = $(4-1)/2 = 1.5 = 1$. 

\[
\begin{array}{cccccccc}
\text{Idx 0} & \text{Idx 1} & \text{Idx 2} & \text{Idx 3} & \text{Idx 4} & \text{Idx 5} & \text{Idx 6} \\
\text{a} & \text{b} & \text{c} & \text{d} & \text{e} & \text{f} & \text{g} \\
0 & 1 & 2 & 3 & 4 & 5 & 6
\end{array}
\]
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.

OK OK Not OK
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.
• 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.
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.
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.

```plaintext
Idx 0
Idx 1
Idx 2
Idx 3
Idx 4
Idx 5
Idx 6
a   b   c   d
0  1  2  3  4  5  6
_numNodes: 4
```
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.

• They can also be implemented using Node objects, but this is awkward.

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

• Consider the max-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 max-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 max-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 max-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 max-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 `add(o)` method as:

```java
void add (T o) {
    _nodeArray[_numNodes] = o;
    _numNodes++;
    bubbleUp(_numNodes - 1);
}
```

• We must then also implement `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.