LECTURE 24

1) Operations on binary search tree
2) Heap

Assignment

- Experiment with the code received in recitation (in particular, see exercise on slide 20).

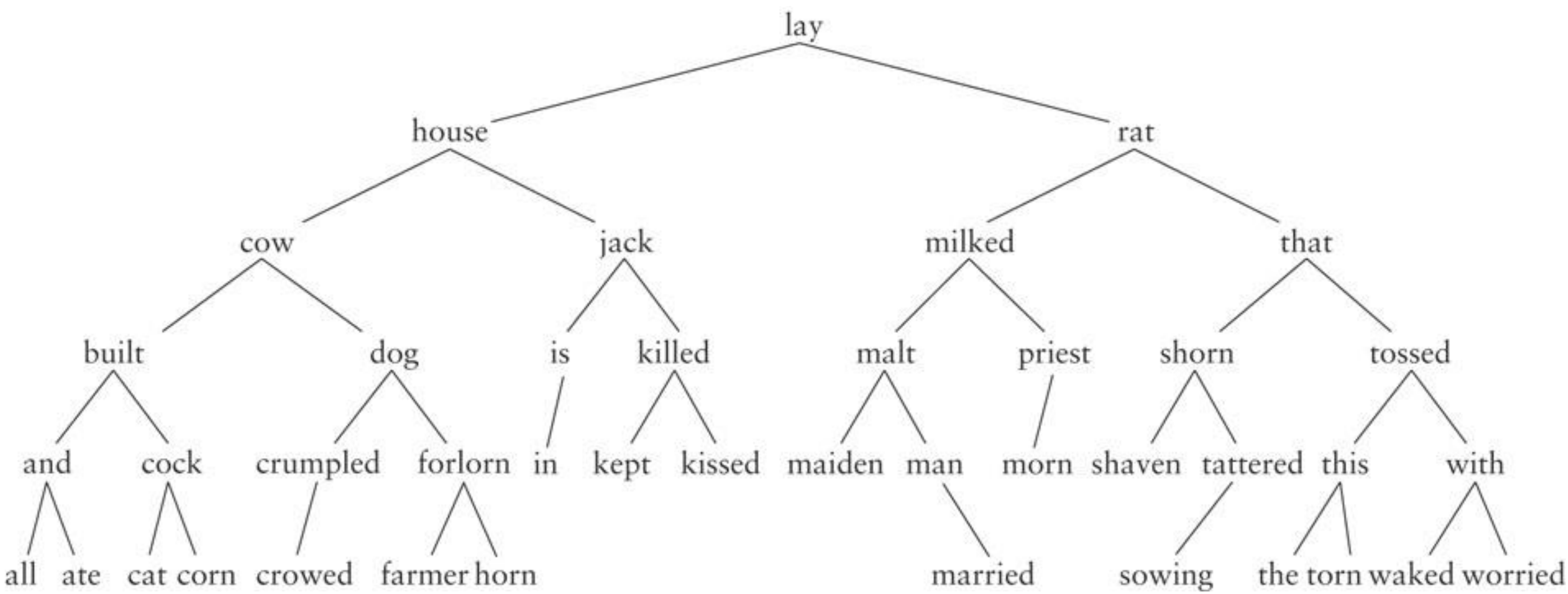
Overview of a Binary Search Tree

- Recall the definition of a binary search tree:

  A set of nodes T is a binary search tree if either of the following is true

  - T is a leaf
  - If T is not a leaf, its root node has two subtrees, T_L and T_R, such that T_L and T_R are binary search trees and the value in the root node of T is greater than all values in T_L and less than all values in T_R

Overview of a Binary Search Tree (cont.)



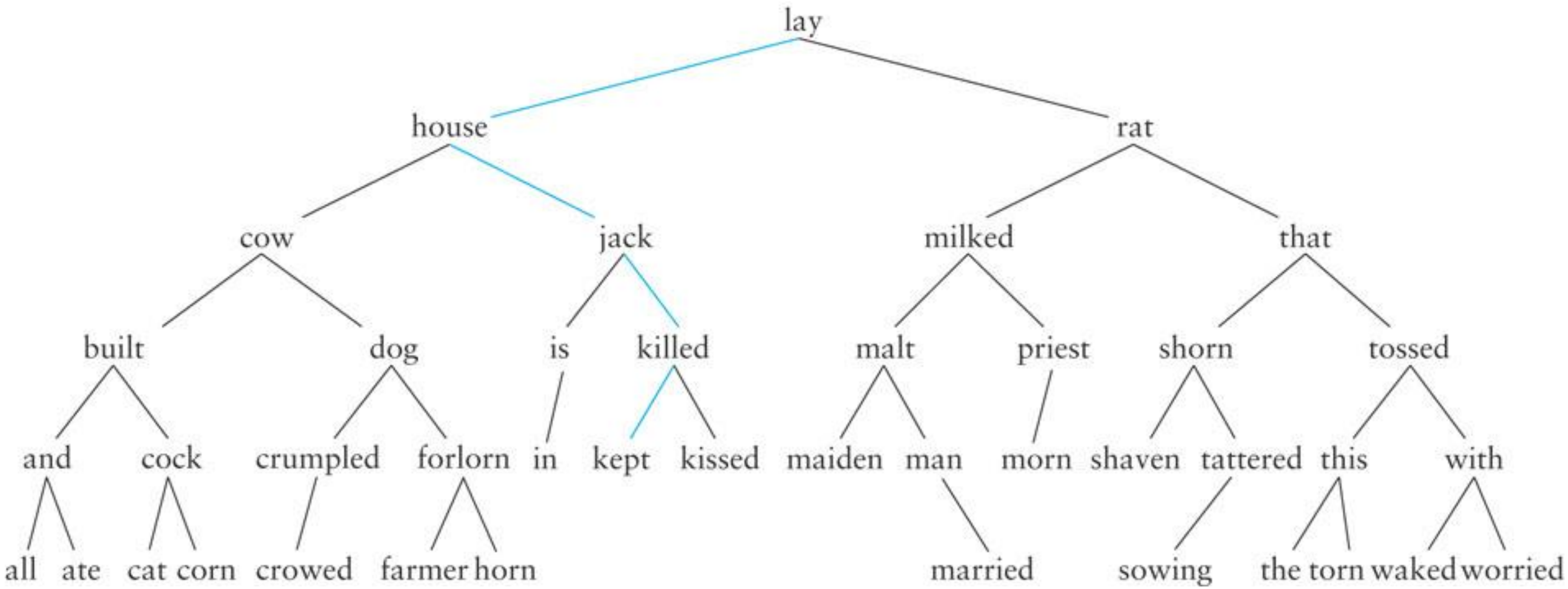
Recursive Algorithm for Searching a Binary Search Tree

1. **`if`** the root is **`null`**
2. the item is not in the tree; return **`null`**
3. Compare the value of **`target`** with **`root.data`**
4. **`if`** they are equal
5. the target has been found; return the data at the root

   `else if` the target is less than **`root.data`**
6. return the result of searching the left subtree

   `else`
7. return the result of searching the right subtree

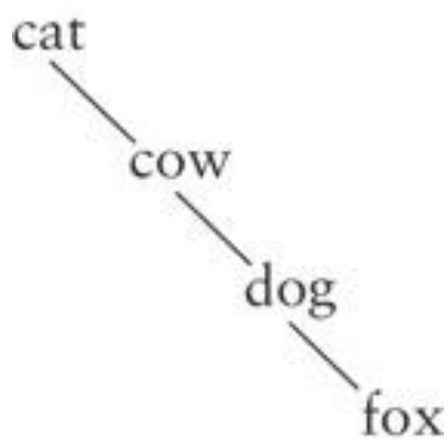
Searching a Binary Tree



Performance

- Search is generally O(log *n*)
- If a tree is not full, performance may degrade:

Searching a tree with only
right subtrees is O(*n*)

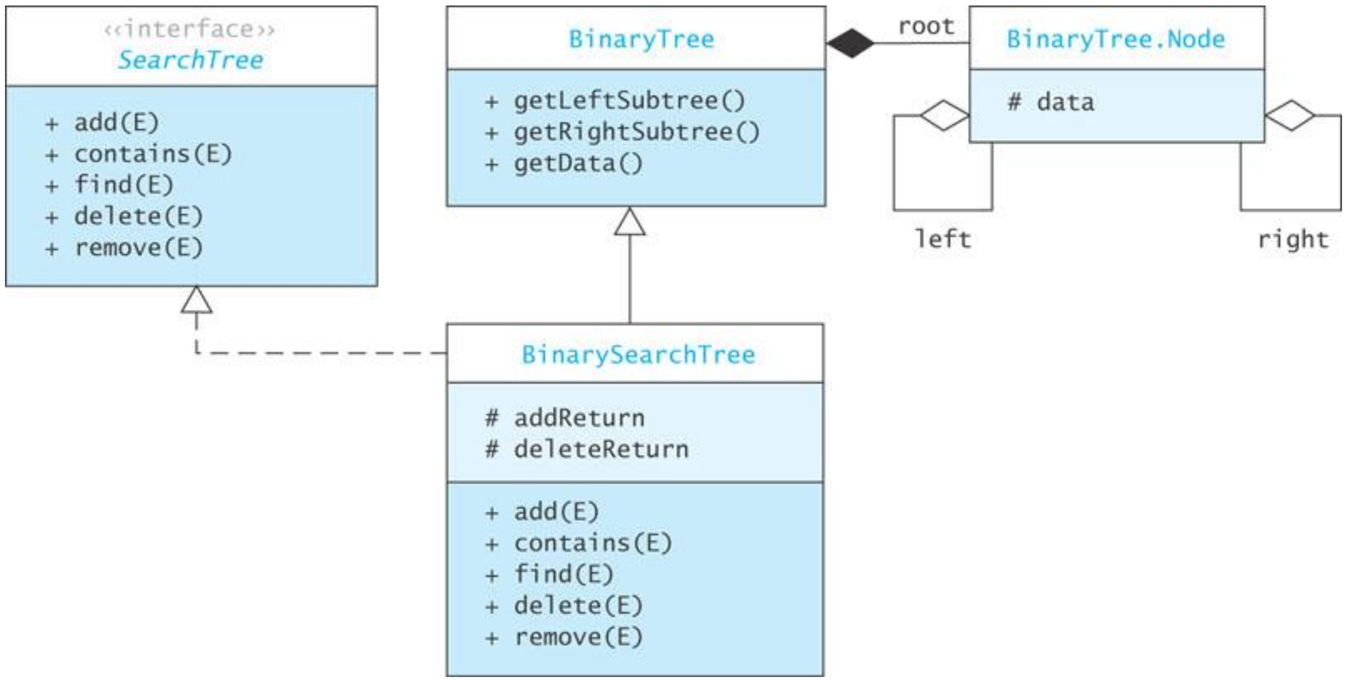


Interface `SearchTree<E>`

Method	Behavior
`boolean add(E item)`	Inserts `item` where it belongs in the tree. Returns **`true`** if item is inserted; **`false`** if it isn't (already in tree).
`boolean contains(E target)`	Returns **`true`** if `target` is found in the tree.
`E find(E target)`	Returns a reference to the data in the node that is equal to `target`. If no such node is found, returns **`null`**.
`E delete(E target)`	Removes `target` (if found) from tree and returns it; otherwise, returns **`null`**.
`boolean remove(E target)`	Removes `target` (if found) from tree and returns **`true`**; otherwise, returns **`false`**.

BinarySearchTree<E> **Class**

Data Field	Attribute
`protected boolean addReturn`	Stores a second return value from the recursive `add` method that indicates whether the item has been inserted.
`protected E deleteReturn`	Stores a second return value from the recursive `delete` method that references the item that was stored in the tree.



Implementing `find` Methods

LISTING 6.3

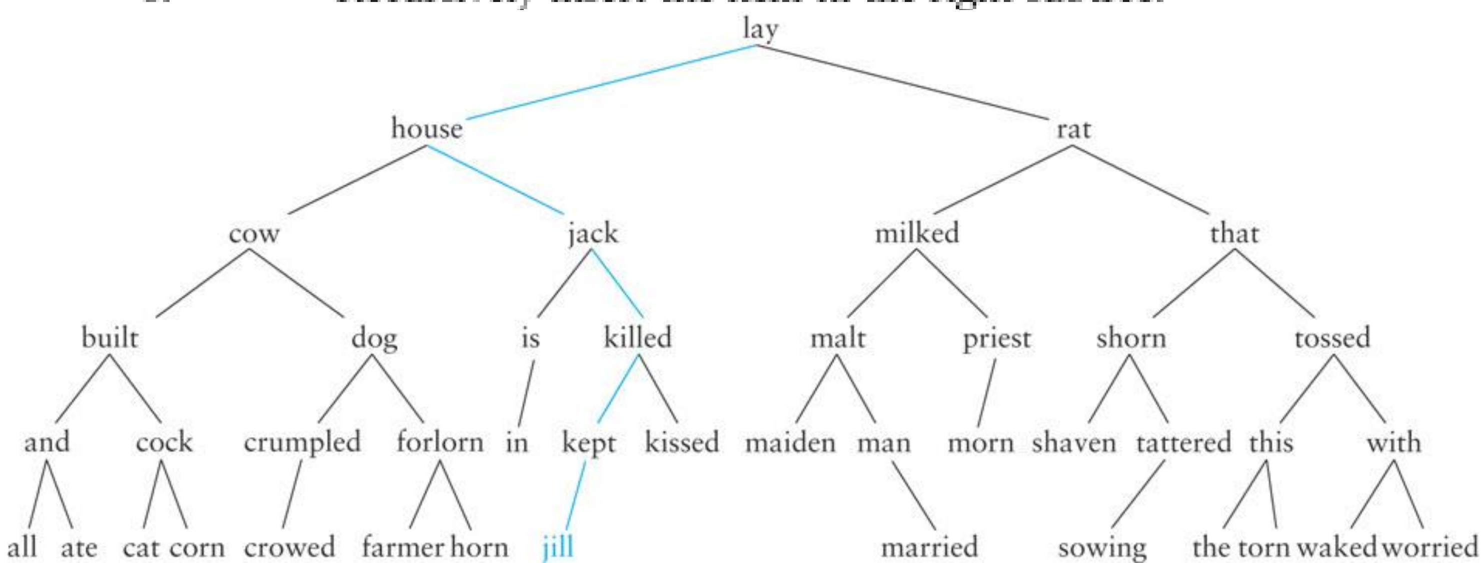
BinarySearchTree find Method

```
/** Starter method find.
    pre: The target object must implement
         the Comparable interface.
    @param target The Comparable object being sought
    @return The object, if found, otherwise null
*/
public E find(E target) {
    return find(root, target);
}

/** Recursive find method.
    @param localRoot The local subtree's root
    @param target The object being sought
    @return The object, if found, otherwise null
*/
private E find(Node<E> localRoot, E target) {
    if (localRoot == null)
        return null;

    // Compare the target with the data field at the root.
    int compResult = target.compareTo(localRoot.data);
    if (compResult == 0)
        return localRoot.data;
    else if (compResult < 0)
        return find(localRoot.left, target);
    else
        return find(localRoot.right, target);
}
```

Insertion into a Binary Search Tree

Recursive Algorithm for Insertion in a Binary Search Tree

1. `if` the root is `null`
2. Replace empty tree with a new tree with the item at the root and return `true`.
3. `else if` the item is equal to `root.data`
4. The item is already in the tree; return `false`.
5. `else if` the item is less than `root.data`
6. Recursively insert the item in the left subtree.
7. `else`
8. Recursively insert the item in the right subtree.



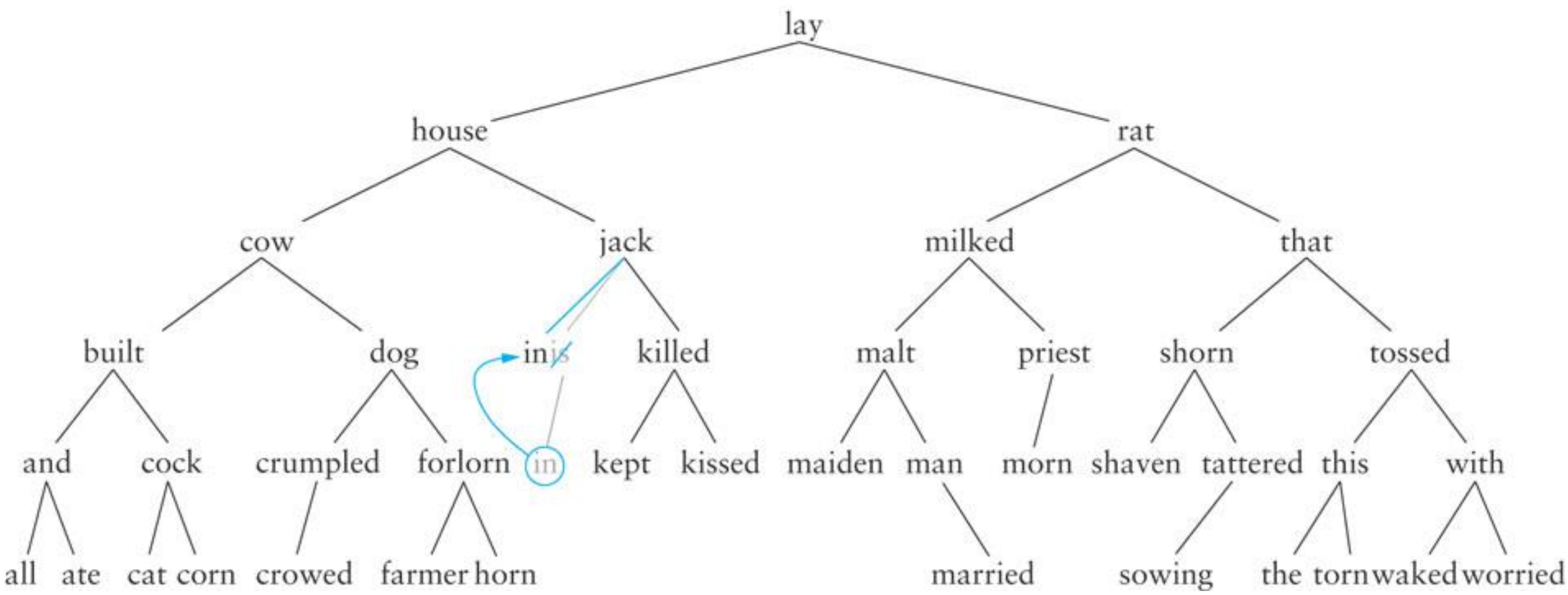
Implementing the `add` Methods

```
/** Starter method add.
    pre: The object to insert must implement the
         Comparable interface.
    @param item The object being inserted
    @return true if the object is inserted, false
            if the object already exists in the tree
*/
public boolean add(E item) {
    root = add(root, item);
    return addReturn;
}
```

Implementing the `add` Methods (cont.)

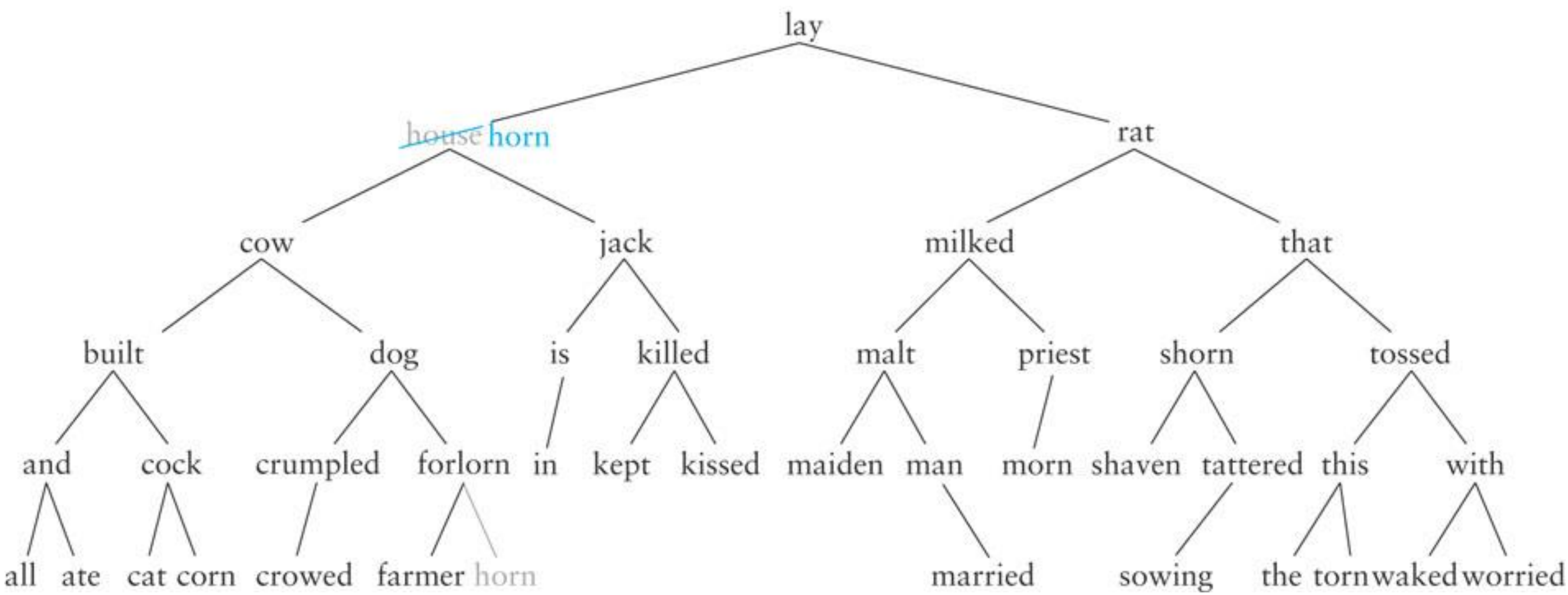
```
/** Recursive add method.
    post: The data field addReturn is set true if the item is added to
         the tree, false if the item is already in the tree.
    @param localRoot The local root of the subtree
    @param item The object to be inserted
    @return The new local root that now contains the
         inserted item
*/
private Node<E> add(Node<E> localRoot, E item) {
    if (localRoot == null) {
         // item is not in the tree – insert it.
         addReturn = true;
         return new Node<E>(item);
    } else if (item.compareTo(localRoot.data) == 0) {
         // item is equal to localRoot.data
         addReturn = false;
         return localRoot;
    } else if (item.compareTo(localRoot.data) < 0) {
         // item is less than localRoot.data
         localRoot.left = add(localRoot.left, item);
         return localRoot;
    } else {
         // item is greater than localRoot.data
         localRoot.right = add(localRoot.right, item);
         return localRoot;
    }
}
```

Removal from a Binary Search Tree

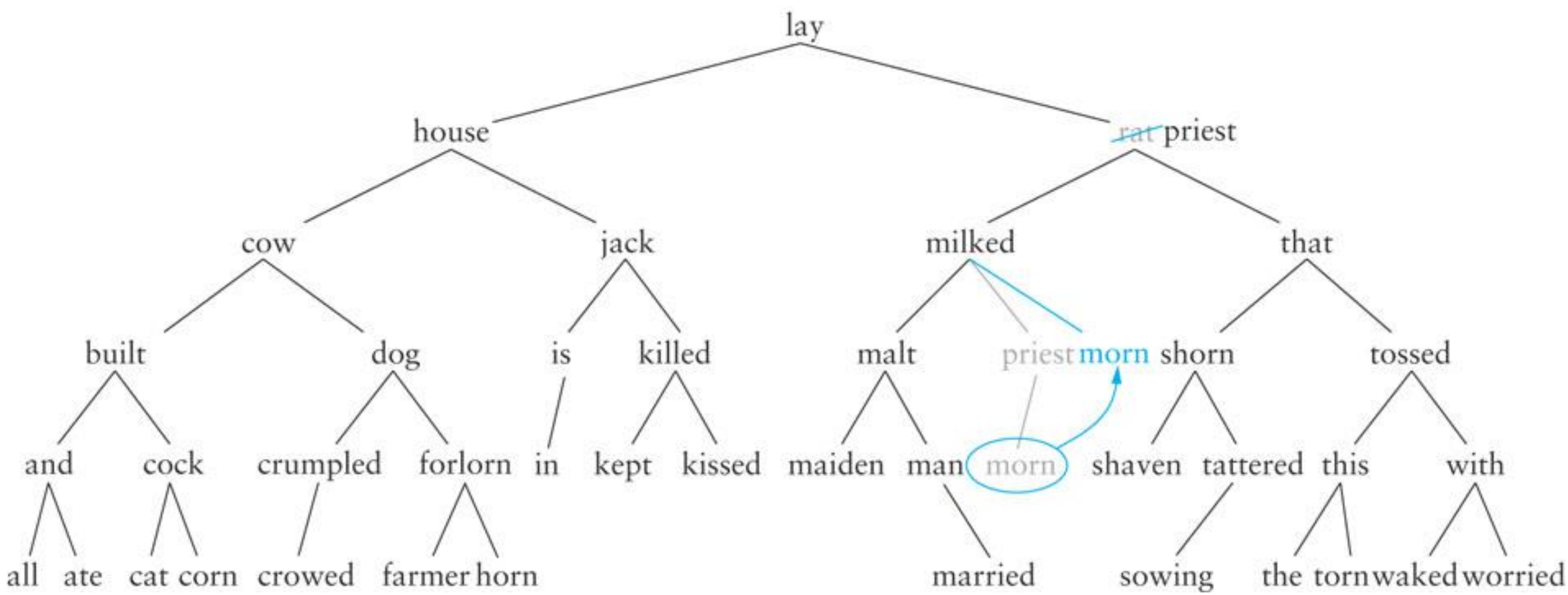


- If the item to be removed has two children, replace it with the largest item in its left subtree – the *inorder predecessor*

Removing from a Binary Search Tree (cont.)



Removing from a Binary Search Tree (cont.)



Algorithm for Removing from a Binary Search Tree

Recursive Algorithm for Removal from a Binary Search Tree

```
1.   if the root is null
2.       The item is not in tree – return null.
3.   Compare the item to the data at the local root.
4.   if the item is less than the data at the local root
5.       Return the result of deleting from the left subtree.
6.   else if the item is greater than the local root
7.       Return the result of deleting from the right subtree.
8.   else // The item is in the local root
9.       Store the data in the local root in deletedReturn.
10.      if the local root has no children
11.          Set the parent of the local root to reference null.
12.      else if the local root has one child
13.          Set the parent of the local root to reference that child.
14.      else // Find the inorder predecessor
15.          if the left child has no right child it is the inorder predecessor
16.              Set the parent of the local root to reference the left child.
17.          else
18.              Find the rightmost node in the right subtree of the left
                 child.
19.              Copy its data into the local root's data and remove it by
                 setting its parent to reference its left child.
```

Implementing the `delete` Method

- Listing 6.5 (`BinarySearchTree delete` Methods; pages 325-326)

Method `findLargestChild`

LISTING 6.6

BinarySearchTree `findLargestChild` Method

```java
/** Find the node that is the
    inorder predecessor and replace it
    with its left child (if any).
    post: The inorder predecessor is removed from the tree.
    @param parent The parent of possible inorder
                  predecessor (ip)
    @return The data in the ip
*/
private E findLargestChild(Node<E> parent) {
    // If the right child has no right child, it is
    // the inorder predecessor.
    if (parent.right.right == null) {
        E returnValue = parent.right.data;
        parent.right = parent.right.left;
        return returnValue;
    } else {
        return findLargestChild(parent.right);
    }
}
```

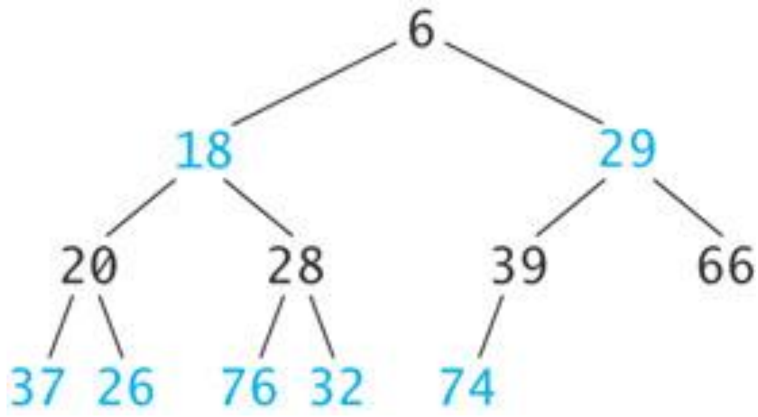
Testing a Binary Search Tree

- To test a binary search tree, verify that an inorder traversal will display the tree contents in ascending order after a series of insertions and deletions are performed

Heaps and Priority Queues

Heaps and Priority Queues

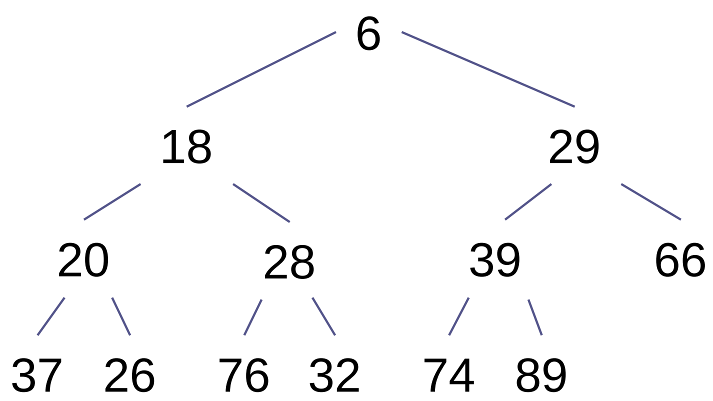
- A heap is a complete binary tree with the following properties
  - The value in the root is the smallest item in the tree
  - Every subtree is a heap



Inserting an Item into a Heap

Algorithm for Inserting in a Heap

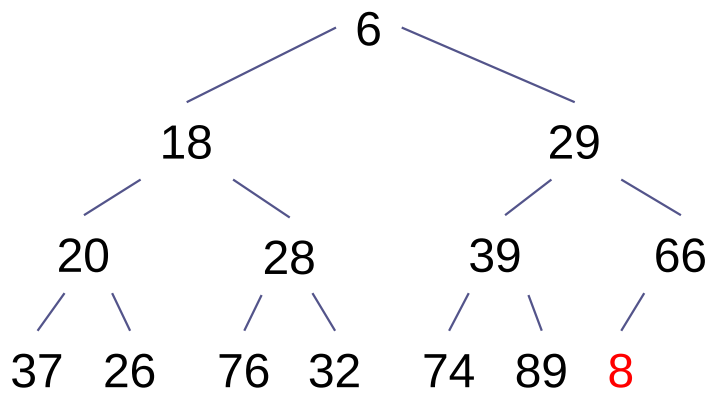
1. Insert the new item in the next position at the bottom of the heap.
2. `while` new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.



Inserting an Item into a Heap (cont.)

Algorithm for Inserting in a Heap

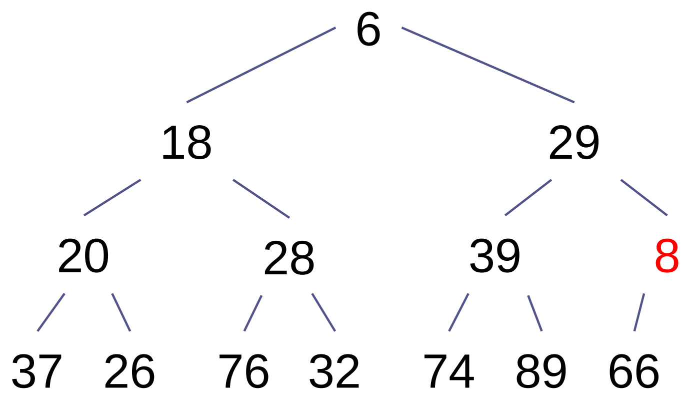
1. Insert the new item in the next position at the bottom of the heap.
2. `while` new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.



Inserting an Item into a Heap (cont.)

Algorithm for Inserting in a Heap

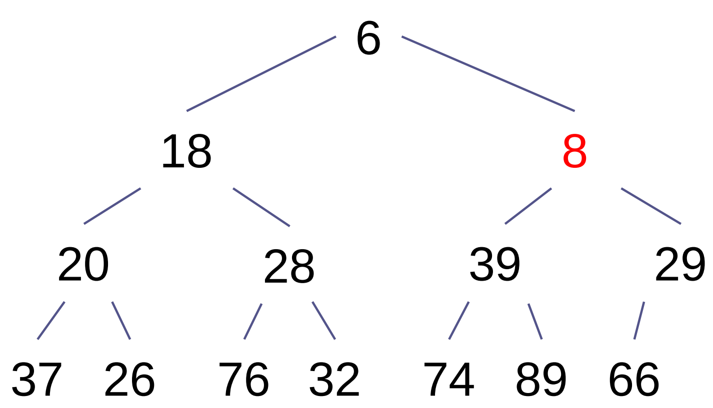
1. Insert the new item in the next position at the bottom of the heap.
2. `while` new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.



Inserting an Item into a Heap (cont.)

Algorithm for Inserting in a Heap

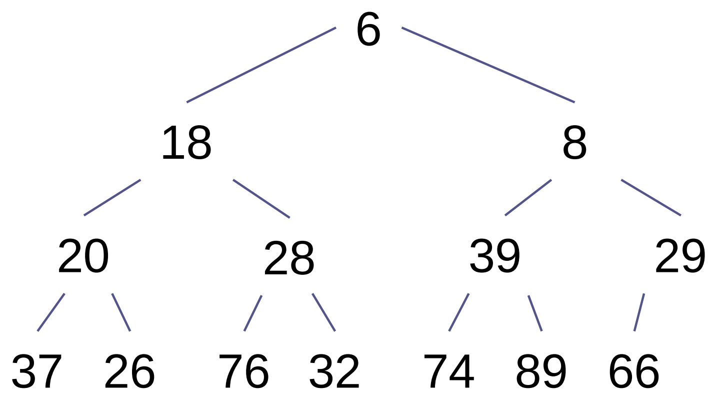
1. Insert the new item in the next position at the bottom of the heap.
2. `while` new item is not at the root and new item is smaller than its parent
3.   Swap the new item with its parent, moving the new item up the heap.



Inserting an Item into a Heap (cont.)

Algorithm for Inserting in a Heap

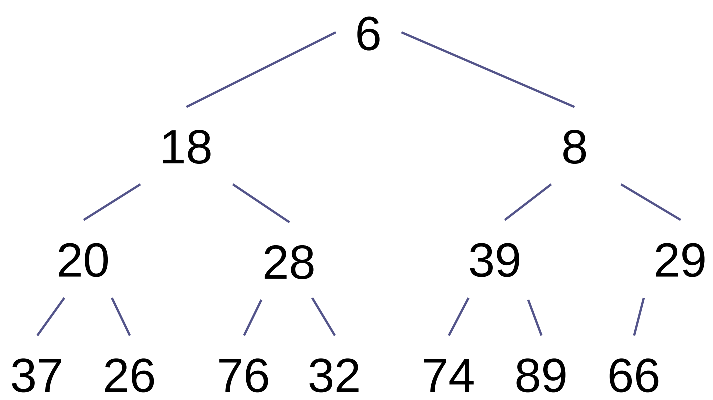
1. Insert the new item in the next position at the bottom of the heap.
2. `while` new item is not at the root and new item is smaller than its parent
3.   Swap the new item with its parent, moving the new item up the heap.



Removing an Item from a Heap

Algorithm for Removal from a Heap

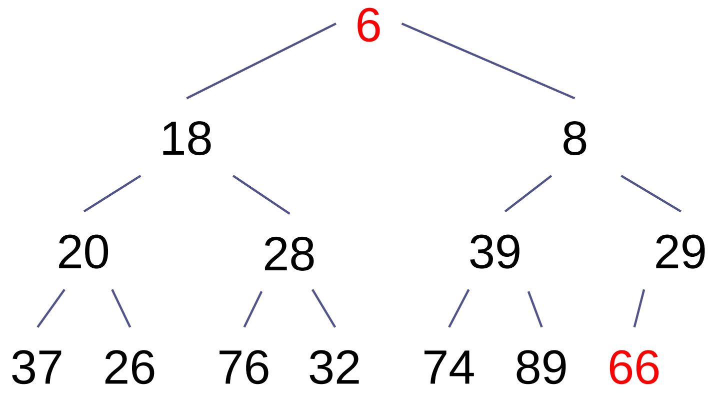
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. `while` item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

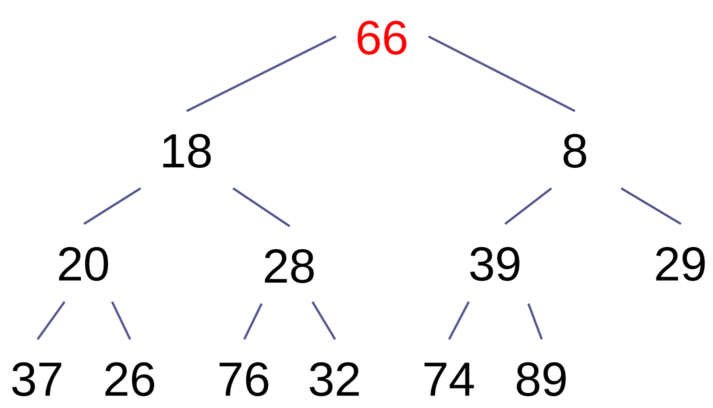
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. `while` item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

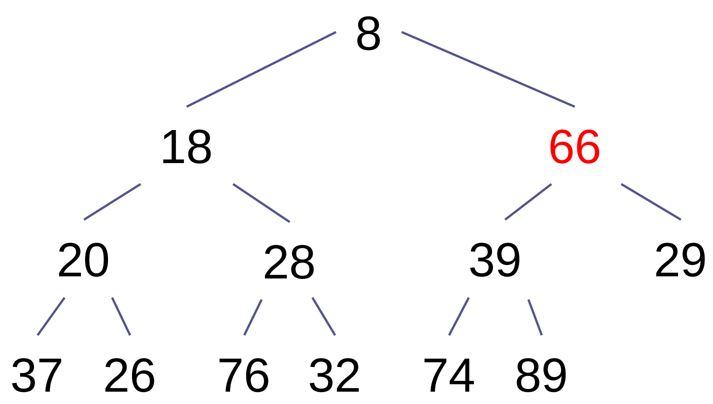
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. `while` item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

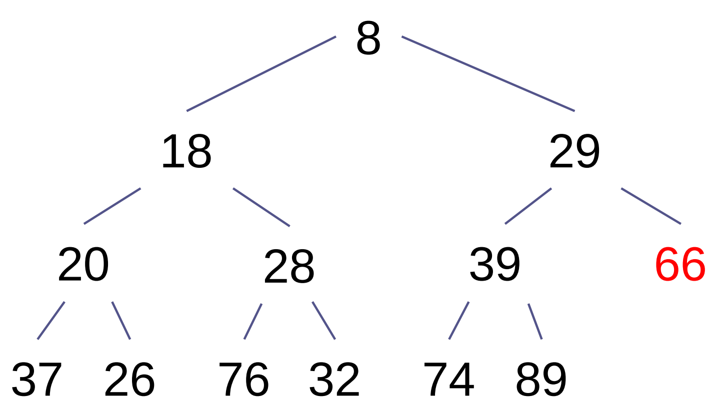
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. `while` item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

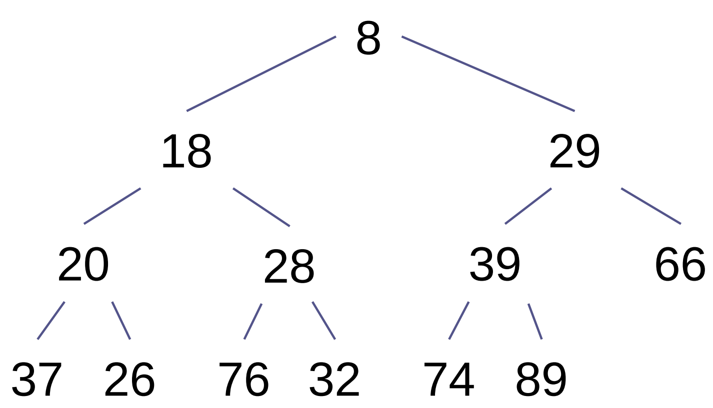
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. `while` item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

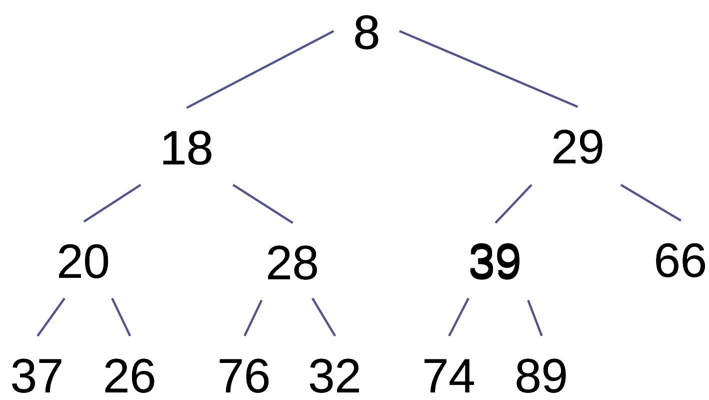
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. `while` item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Implementing a Heap

- Because a heap is a complete binary tree, it can be implemented efficiently using an array rather than a linked data structure

Implementing a Heap (cont.)



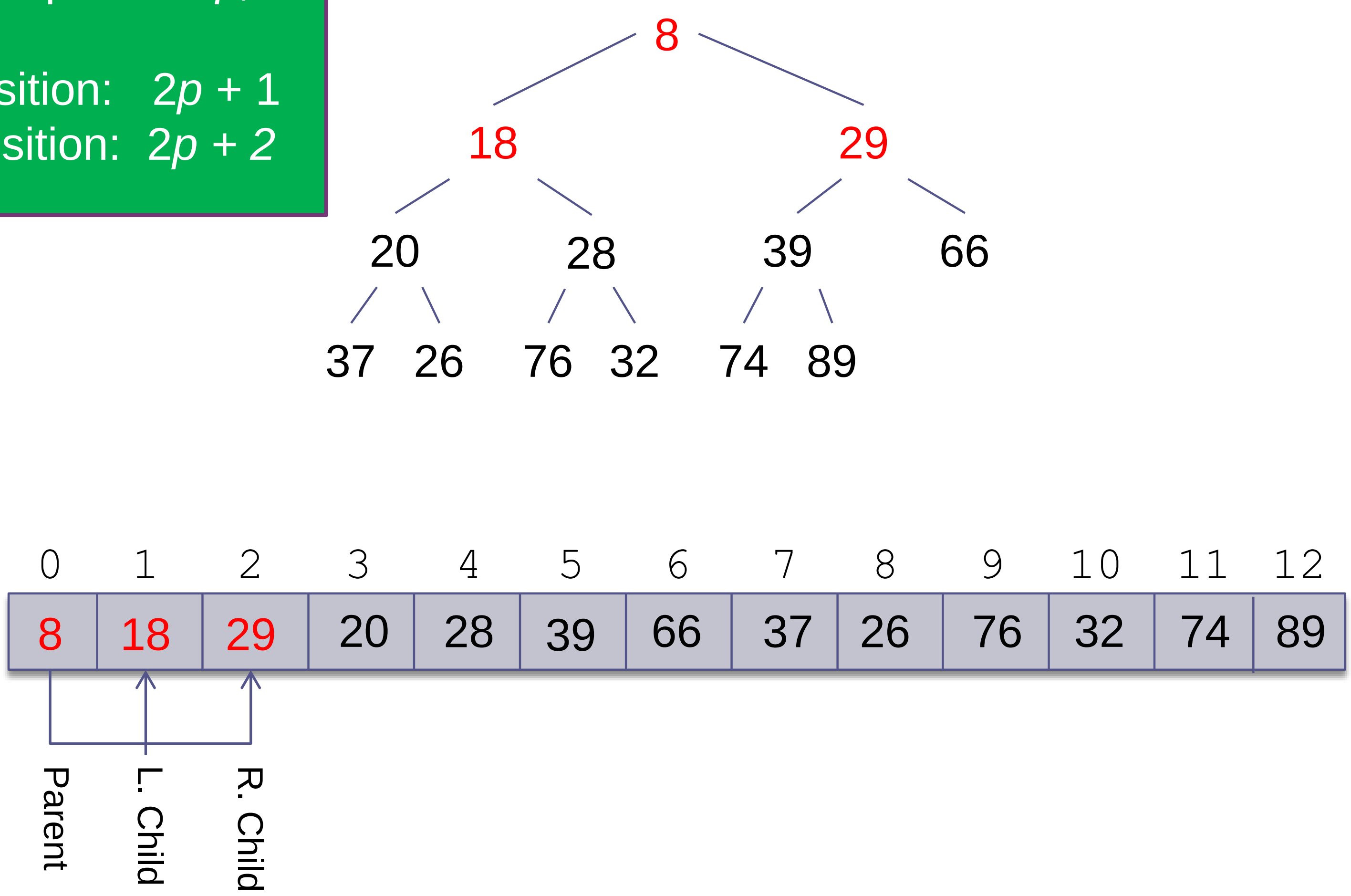
0	1	2	3	4	5	6	7	8	9	10	11	12
8	18	29	20	28	39	66	37	26	76	32	74	89

Implementing a Heap (cont.)

For a node at position *p*,

L. child position: $2p + 1$

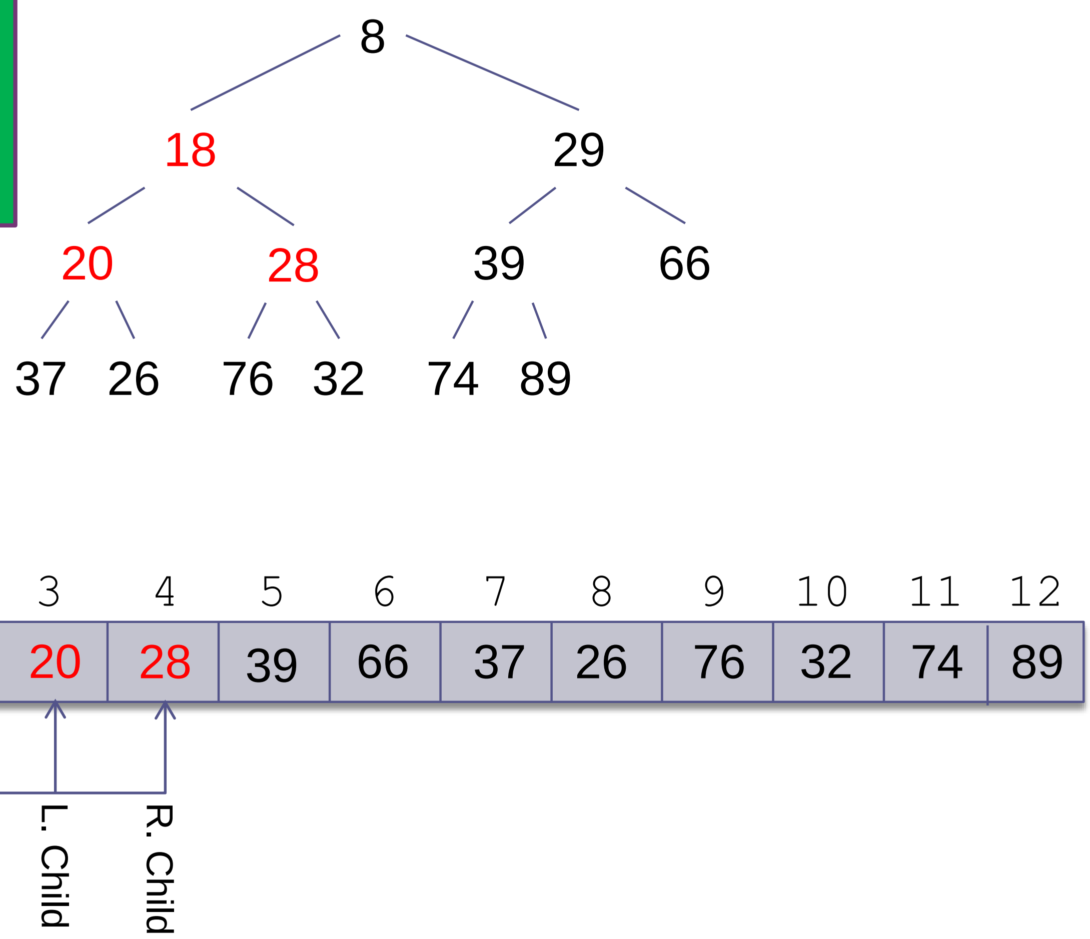
R. child position: $2p + 2$



Implementing a Heap (cont.)

For a node at position *p*,

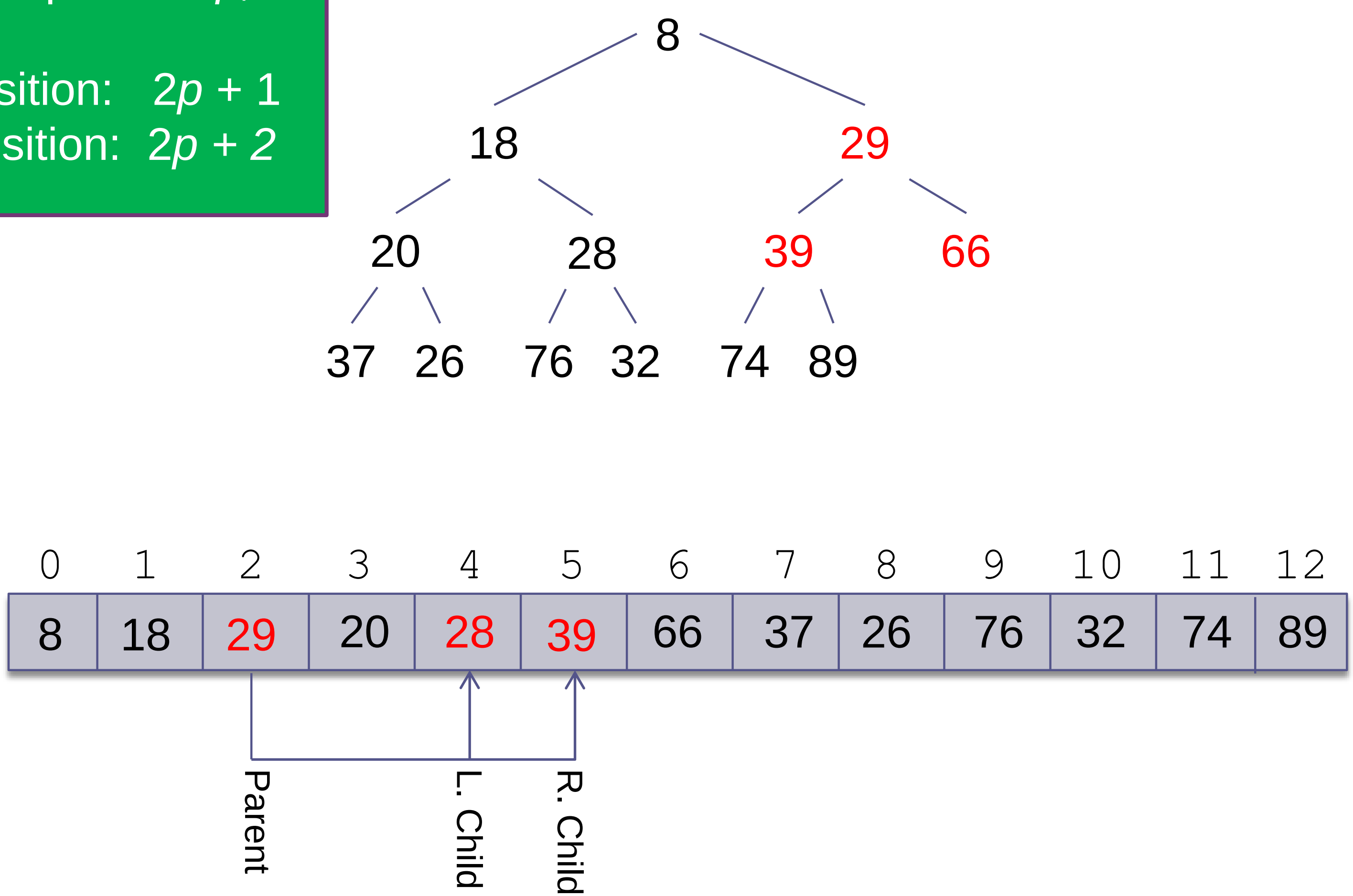
L. child position: $2p + 1$
R. child position: $2p + 2$



Implementing a Heap (cont.)

For a node at position *p*,

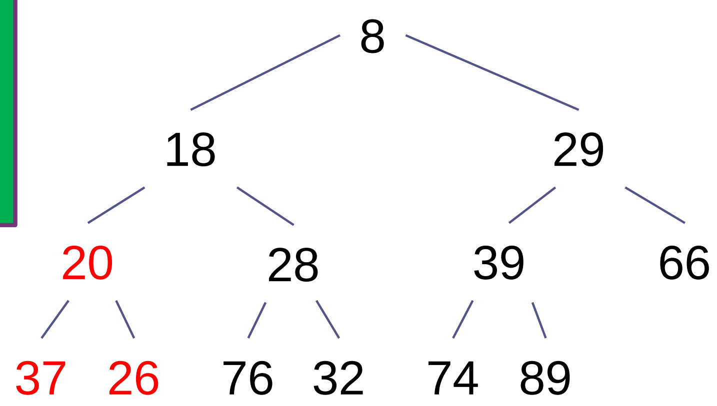
L. child position: $2p + 1$
R. child position: $2p + 2$

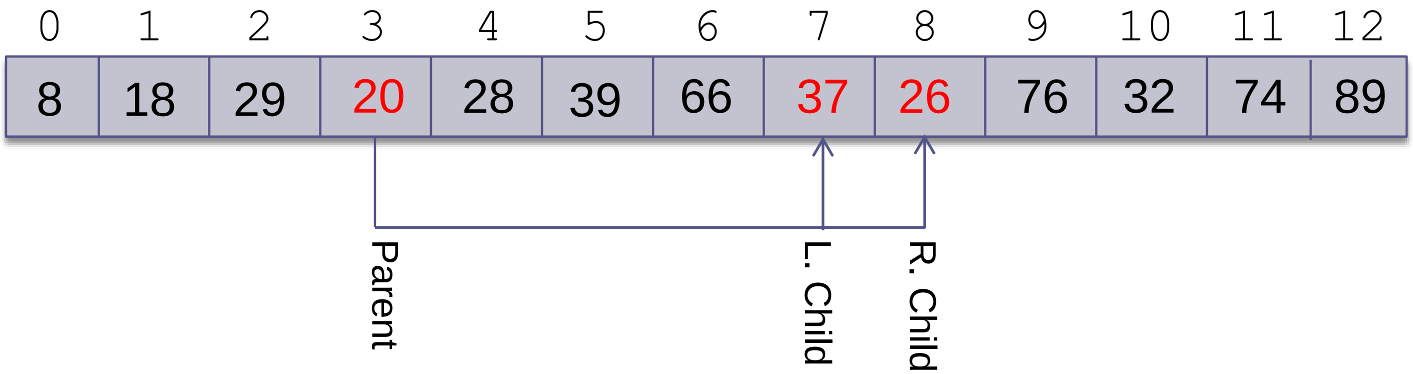


Implementing a Heap (cont.)

For a node at position *p*,

L. child position: 2*p* + 1
R. child position: 2*p* + 2

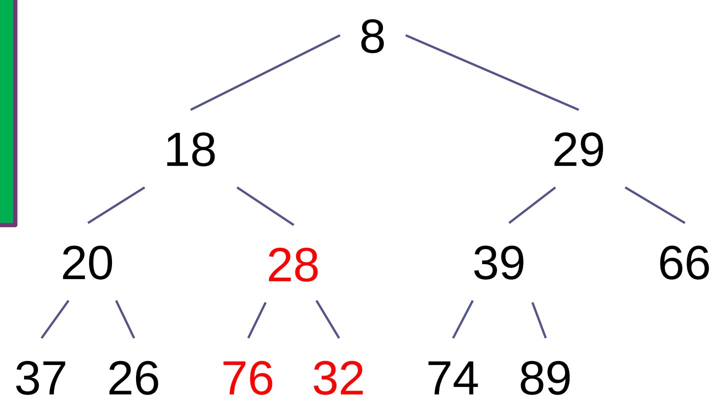


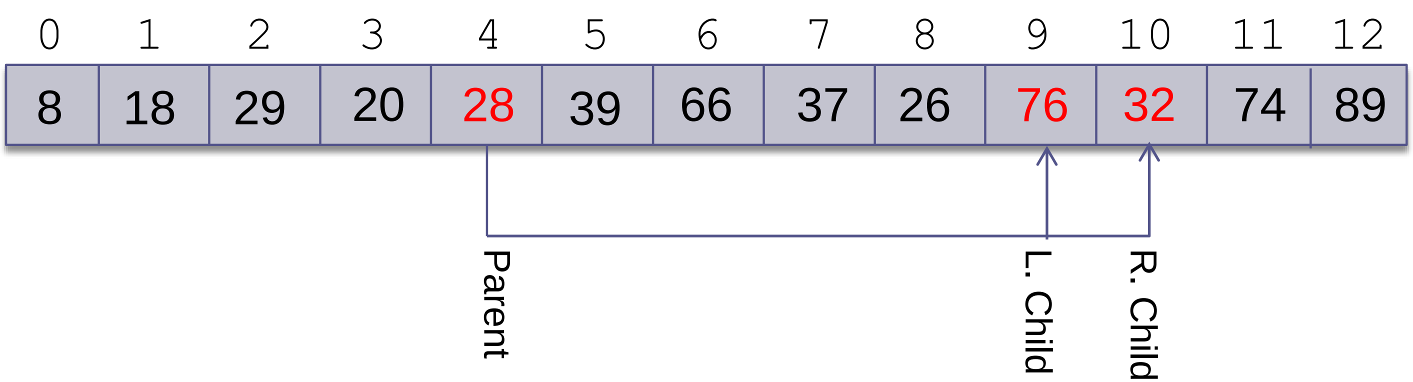


Implementing a Heap (cont.)

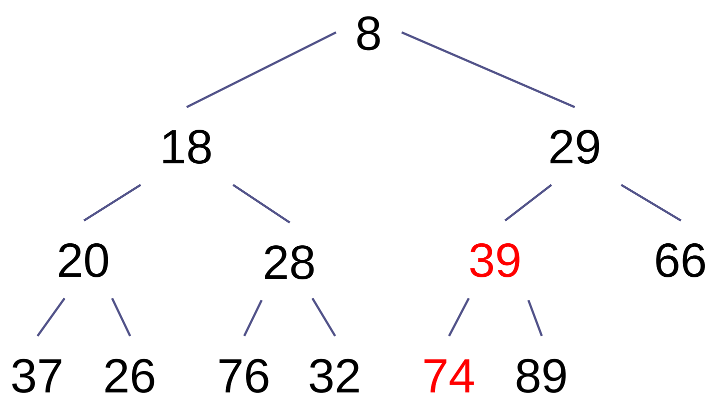
For a node at position *p*,

L. child position: $2p + 1$
R. child position: $2p + 2$





Implementing a Heap (cont.)



A node at position *c* can find its parent at = |_ (*c* – 1)/2_| =

$$\begin{cases} \frac{c-1}{2}, c \text{ is odd} \\ \frac{c-2}{2}, \ c \text{ is even} \end{cases}$$

0	1	2	3	4	5	6	7	8	9	10	11	12
8	18	29	20	28	39	66	37	26	76	32	74	89