Binary Search Trees

Section 6.4

BinaryTree<E> Class (cont.)

```
public class BinaryTree<E> {
   // Data members/fields...just the root is needed
   - Node<E> root;
   // Constructor(s)
   + BinaryTree()
   + BinaryTree(Node<E> node)
   + BinaryTree(E data, BinaryTree<E> leftTree, BinaryTree<E> rightTree)
   // Methods: Accessors
   + E getData() throws BinaryTreeException
   + boolean isEmpty()
   + void clear()
   + BinaryTree<E> getLeftSubtree()
   + BinaryTree<E> getRightSubtree()
   + boolean isLeaf()
   + int height()
   - int height(Node<E> node)
   + String toString()
   - void preOrderTraverse(Node<E> node, int depth, StringBuilder sb)
} // BinaryTree<E> class
```

Coding…

- Run the BinaryTree code…

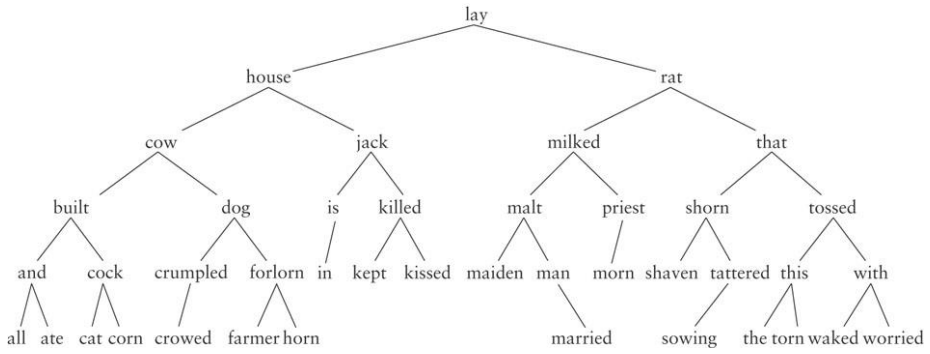
Overview of a Binary Search Tree

- 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 empty
  - If T is not empty, 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.)



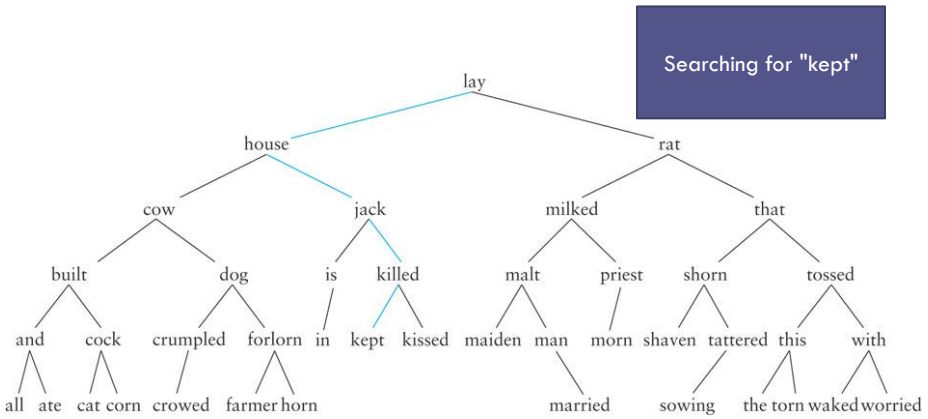
Recursive Algorithm for Searching a Binary Search Tree

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

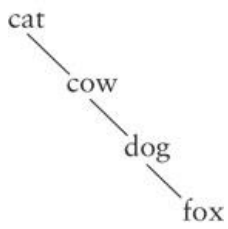
   `else`
7. return the result of searching the right subtree

Searching a Binary Tree



Performance

- Search a tree is generally O(log *n*)
- If a tree is not very full, performance will be worse
- Searching a tree with only right subtrees, for example, is O(*n*)



Implementing Binary Search Trees

```
public class BST<E extends Comparable<E>> extends BinaryTree<E> {
   // Inherit all of the functionality of BinaryTree<E> class
   // Plus, add the following to it…

   private int size = 0;

   // Methods

   …
} // class BST<E>
```

Since it is a search tree, we will need to inherently provide for ordering (comparison) of elements in the tree.

By, specifying that type E extends Comparable, we will be able to freely use the compareTo() method for ordering.

Implementing Binary Search Trees

```
public class BST<E extends Comparable<E>> extends BinaryTree<E> {
   // Inherit all of the functionality of BinaryTree<E> class
   // Plus, add the following to it…

   private int size = 0;

   // Methods

   + int size()
   + void clear()
   + void add(E item)
   + E find(E item)
   + boolean contains(E item)
   + E delete(E item)
   + boolean remove(E item)
} // class BST<E>
```

Implementing Binary Search Trees

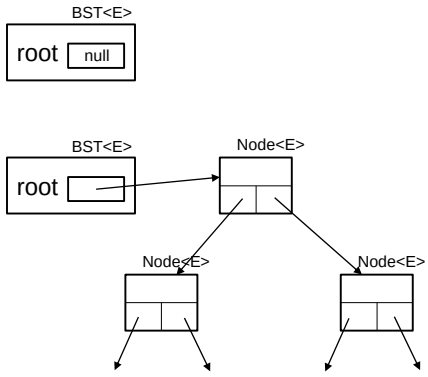
```
public int size() {
   return size;
} // size()


public void clear() {
   root = null;
   size = 0;
} // clear()
```

Implementing Binary Search Trees

```
public class BST<E extends Comparable<E>> extends BinaryTree<E> {
   // Inherit all of the functionality of BinaryTree<E> class
   // Plus, add the following to it…

   private int size = 0;
   // Methods

   + int size()
   + void clear()
   + void add(E item)
   + E find(E item)
   + boolean contains(E item)
   + E delete(E item)
   + boolean remove(E item)
} // class BST<E>
```

Implementing Binary Search Trees

```
public void add(E item) {
   root = add(root, item);
   size++;
} // add(item)
```



Implementing Binary Search Trees

```
private void add(Node<E> node, E item) {
   // Insert item (preserving BST property) at or below node.
   // Returns node at/under which the item was inserted.
} // add(item)
```

Implementing Binary Search Trees

```
private void add(Node<E> node, E item) {
   // Insert item (preserving BST property) at or below node.
   // Returns node at/under which the item was inserted.
   if (node == null) {
      // Insert item right here
   }
   else {
      // if item is less than or equal to data at node
      // insert below node's left
   }
   else {
      // if item is greater than item at node
      // insert below node's right
   }
} // add(item)
```

Implementing Binary Search Trees

```
private void add(Node<E> node, E item) {
   // Insert item (preserving BST property) at or below node.
   // Returns node at/under which the item was inserted.
   if (node == null) {
      // Insert item right here
      return new Node<E>(item);
   }
   else {
      // if item is less than or equal to data at node
      // insert below node's left
      if (item.compareTo(node.data) <= 0) {
         node.left = add(node.left, item);
         return node;
      }
   }
   else {
      // item is greater than item at node
      // insert below node's right
      node.right = add(node.right, item);
      return node;
   }
} // add(item)
```

Implementing Binary Search Trees

```
public class BST<E extends Comparable<E>> extends BinaryTree<E> {
   // Inherit all of the functionality of BinaryTree<E> class
   // Plus, add the following to it…

   private int size = 0;
   // Methods

   + int size()
   + void clear()
   + void add(E item)
   + E find(E item)
   + boolean contains(E item)
   + E delete(E item)
   + boolean remove(E item)
} // class BST<E>
```

Implementing Binary Search Trees

```
public E find(E item) {
   // Find and return the item in BST
   return find(root, item);
} // find(item)


private E find(Node<E> node, E item) {
   // Find and return item at or below node
   …
} // find(node, item)
```

Implementing Binary Search Trees

```
private E find(Node<E> node, E item) {
   // Find and return item at or below node
   if (node == null) {
      // item does not exist
   }
   else {
      // item is equal to item at node…
   }
   else {
      // item is less than item at node…
   }
   else {
      // item is greater than item at node
   }
} // find(node, item)
```

Implementing Binary Search Trees

```
private E find(Node<E> node, E item) {
   // Find and return item at or below node
   if (node == null) {
      // item does not exist
      return null;
   }
   int compResult = item.compareTo(node.data);
   else if (compResult == 0){
      // item is equal to item at node…
      return node.data;
   }
   else if (compResult < 0) {
      // item is less than item at node…look below the left subtree
      return find(node.left, item);
   }
   else {
      // item is greater than item at node…look below right subtree
      return find(node.right, item);
   }
} // find(node, item)
```

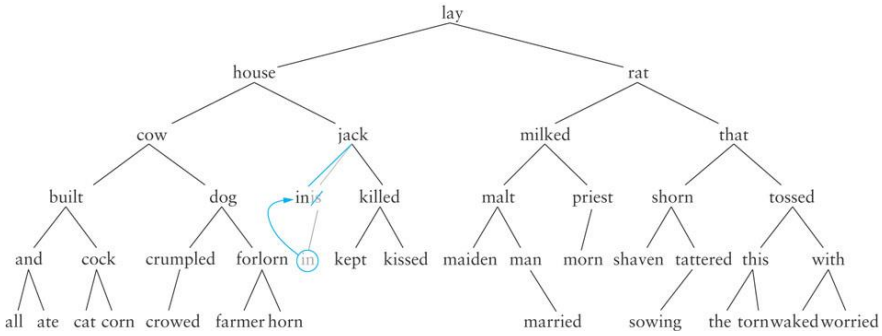
Implementing Binary Search Trees

```
public class BST<E extends Comparable<E>> extends BinaryTree<E> {
   // Inherit all of the functionality of BinaryTree<E> class
   // Plus, add the following to it…

   private int size = 0;
   // Methods

   + int size()
   + void clear()
   + void add(E item)
   + E find(E item)
   + boolean contains(E item)
   + E delete(E item)
   + boolean remove(E item)
} // class BST<E>
```

Removal from a Binary Search Tree

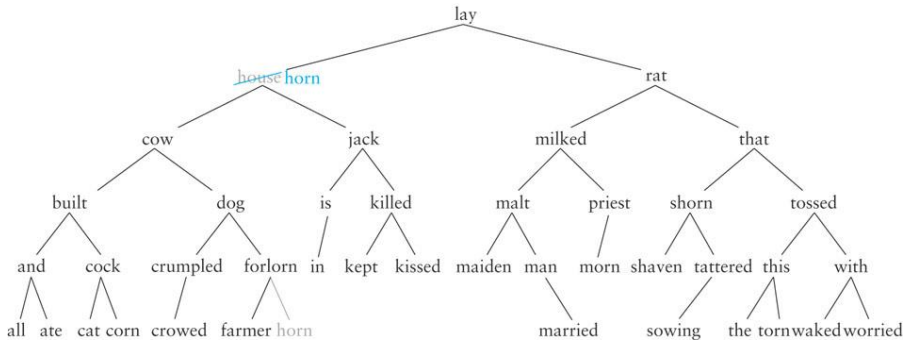
- If the item to be removed has no children, simply delete the reference to the item
- If the item to be removed has only one child, change the reference to the item so that it references the item's only child

Removal from a Binary Search Tree (cont.)

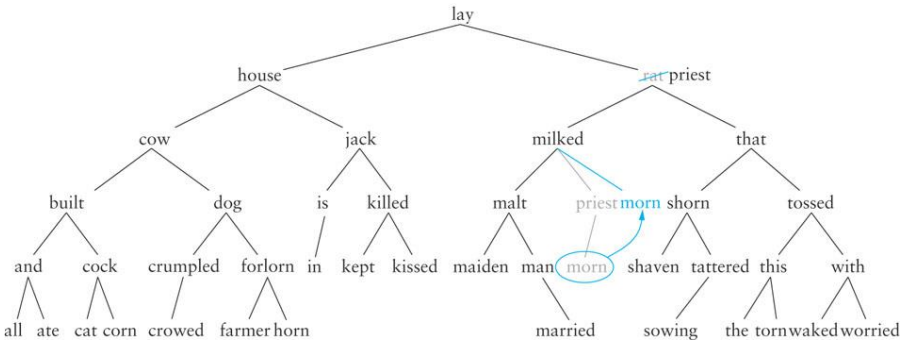


Removing from a Binary Search Tree (cont.)

- 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.)

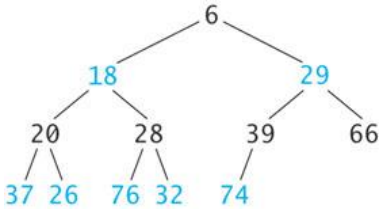


Heaps and Priority Queues

Section 6.5

Heaps and Priority Queues

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

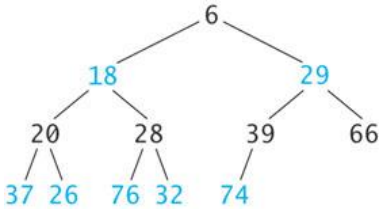


Min Heap: Value in root is smallest item in tree

Max Heap: Value in root is largest item in tree

Heaps and Priority Queues

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

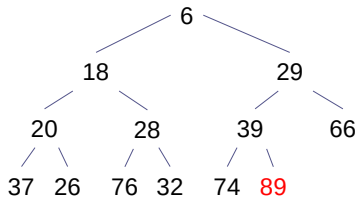


Min Heap: Value in root is smallest item in tree

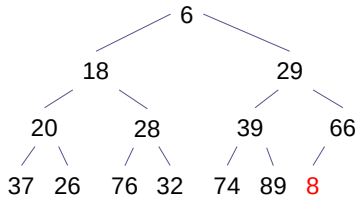
Max Heap: Value in root is largest item in tree

Note: A heap is a **complete binary tree.**

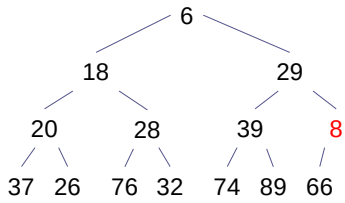
Inserting an Item into a Heap



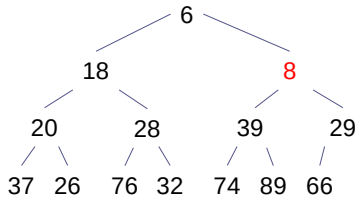
Inserting an Item into a Heap (cont.)



Inserting an Item into a Heap (cont.)



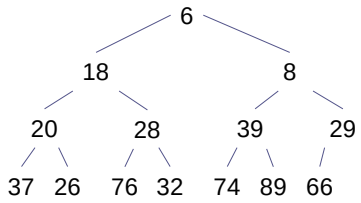
Inserting an Item into a Heap (cont.)



Inserting an Item into a Heap (cont.)

Algorithm for Inserting in a Heap

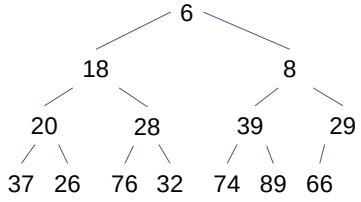
1. Insert the new item in the next position at the bottom of the heap.
2. `while` new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.



Removing an Item from a Heap

Algorithm for Removal from a Heap

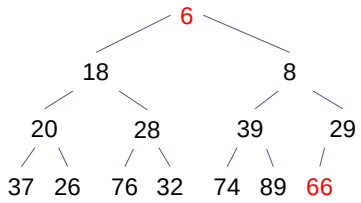
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. `while` item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

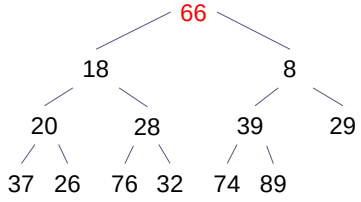
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. `while` item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

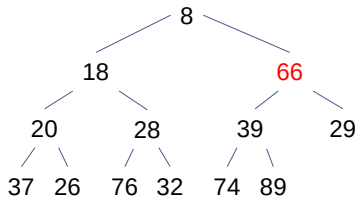
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. `while` item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

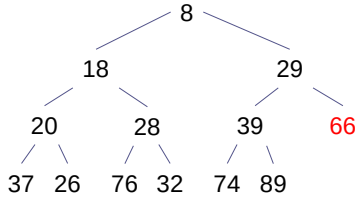
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. `while` item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

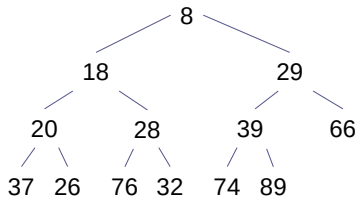
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. `while` item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



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

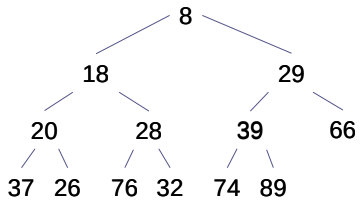
```
public class heap<E> {
     // Data Structure
     // Constructor(s)
     // Methods
     + add(E item)
     + remove(E item)
     …
} // class heap<E>
```

Implementing a Heap

```
public class heap<E> {
     // Data Structure
     // Constructor(s)
     // Methods
     + add(E item)
     + remove(E item)
     …
} // class heap<E>
```

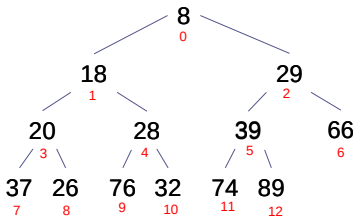
Implementing a Heap

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



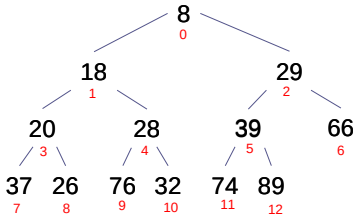
Implementing a Heap

- 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

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

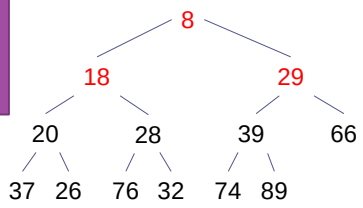


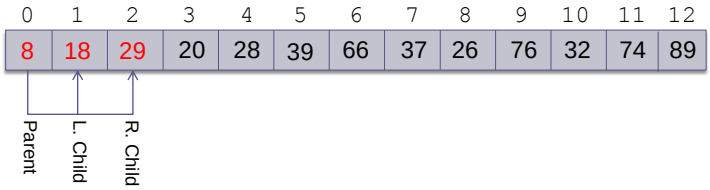
0	1	2	3	4	5	6	7	8	9	10	11	12
8	18	29	20	28	39	66	37	26	76	32	74	89

Implementing a Heap (cont.)

For a node at position *p*,

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

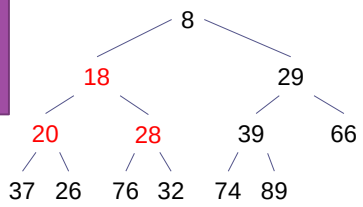


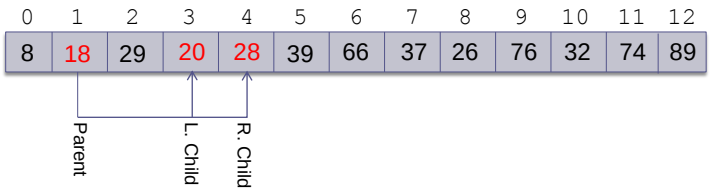


Implementing a Heap (cont.)

For a node at position *p*,

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

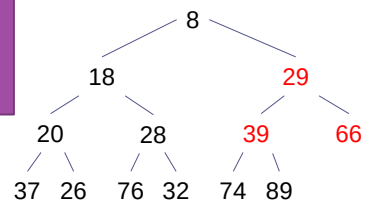


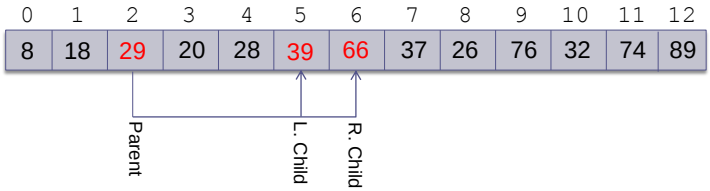


Implementing a Heap (cont.)

For a node at position *p*,

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

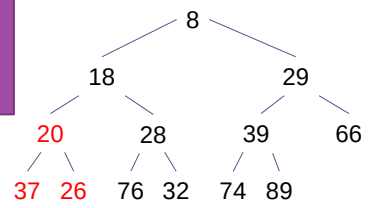


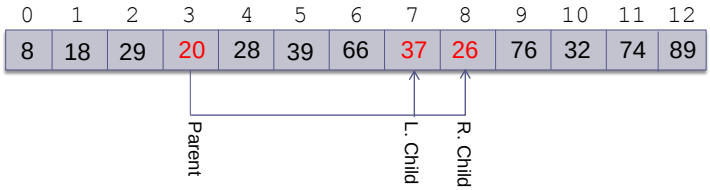


Implementing a Heap (cont.)

For a node at position *p*,

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

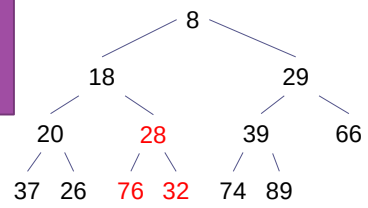


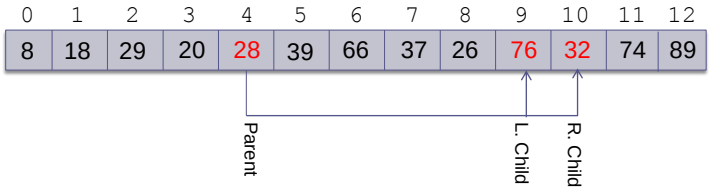


Implementing a Heap (cont.)

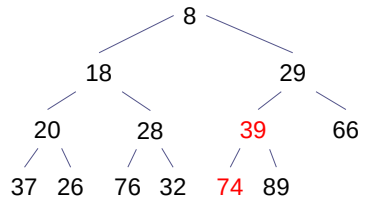
For a node at position *p*,

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

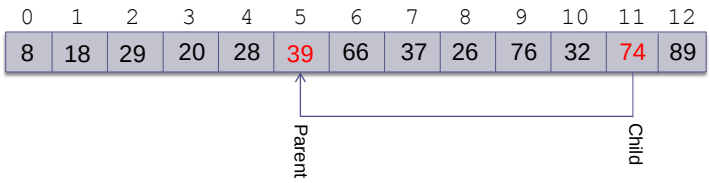




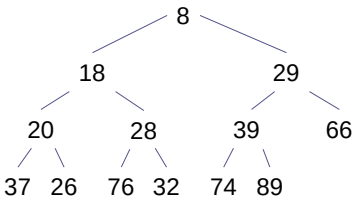
Implementing a Heap (cont.)



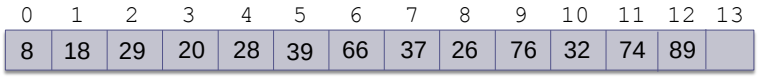
A node at position c can find its parent at $(c - 1)/2$



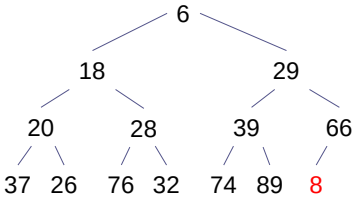
Inserting into a Heap Implemented as an `ArrayList`



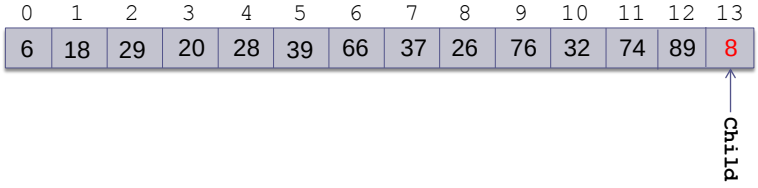
1. Insert the new element at the end of the `ArrayList` and set `child` to `table.size() - 1`



Inserting into a Heap Implemented as an `ArrayList` (cont.)

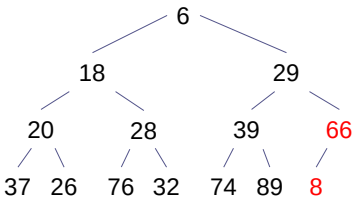


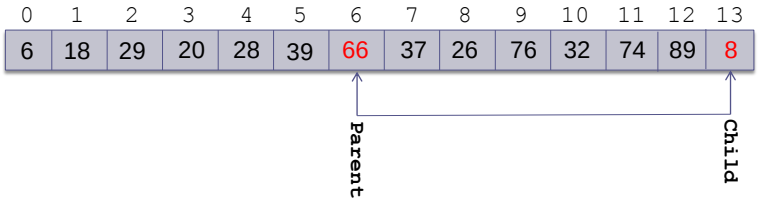
1. Insert the new element at the end of the `ArrayList` and set `child` to `table.size() - 1`



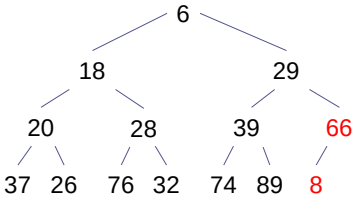
Inserting into a Heap Implemented as an `ArrayList` (cont.)

2. Set `parent` to `(child - 1)/ 2`

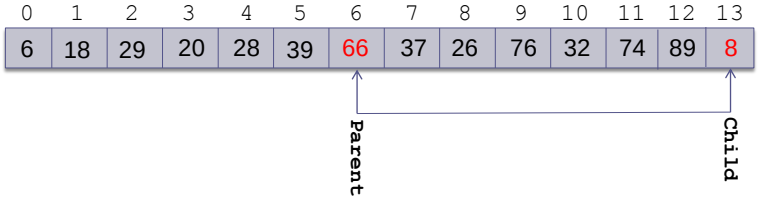




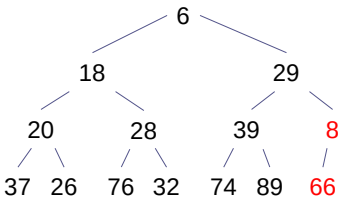
Inserting into a Heap Implemented as an `ArrayList` (cont.)



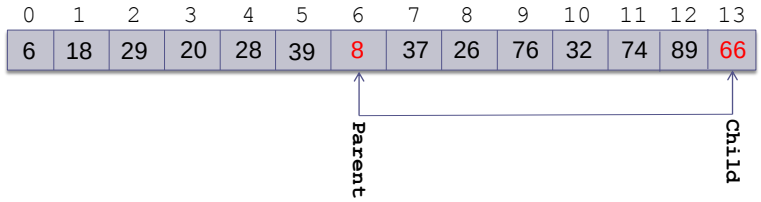
3. while (`parent >= 0` and `table[parent] > table[child]`)
4. Swap `table[parent]` and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`



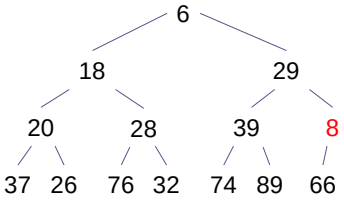
Inserting into a Heap Implemented as an `ArrayList` (cont.)



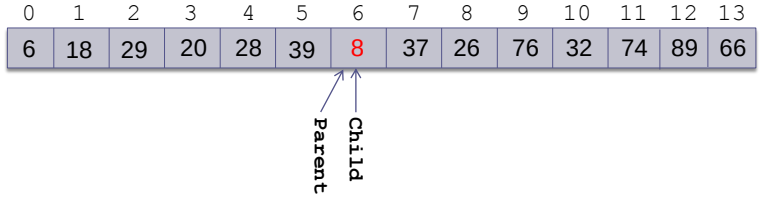
3. while (`parent >= 0` and `table[parent] > table[child]`)
4. Swap `table[parent]` and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`



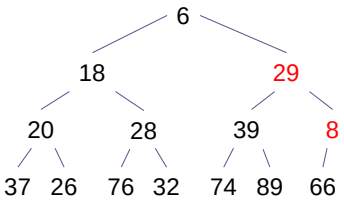
Inserting into a Heap Implemented as an `ArrayList` (cont.)



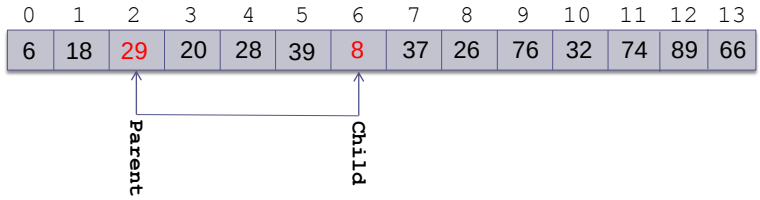
3. while (`parent >= 0` and `table[parent] > table[child]`)
4. Swap `table[parent]` and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`



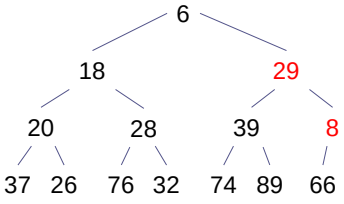
Inserting into a Heap Implemented as an `ArrayList` (cont.)



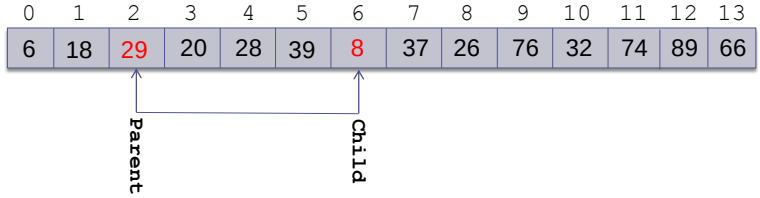
3. while (`parent >= 0` and `table[parent] > table[child]`)
4. Swap `table[parent]` and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`



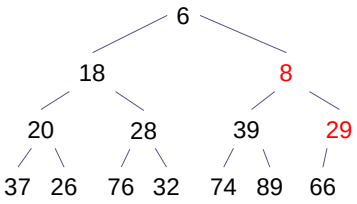
Inserting into a Heap Implemented as an `ArrayList` (cont.)



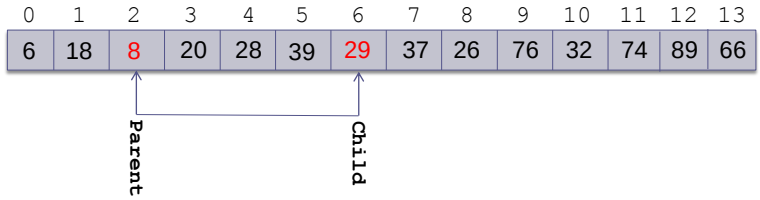
3. while (`parent >= 0` and `table[parent] > table[child]`)
4. Swap `table[parent]` and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`



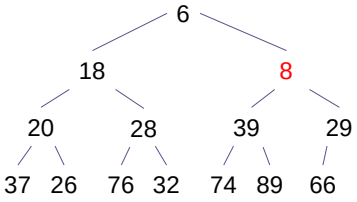
Inserting into a Heap Implemented as an `ArrayList` (cont.)



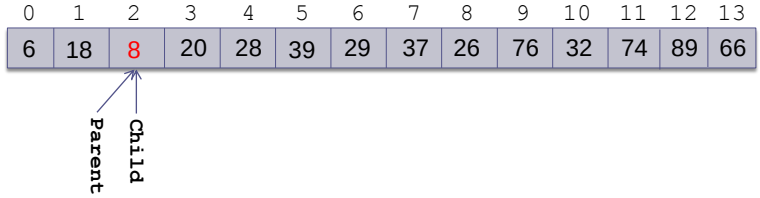
3. while (`parent >= 0` and `table[parent] > table[child]`)
4. Swap `table[parent]` and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`



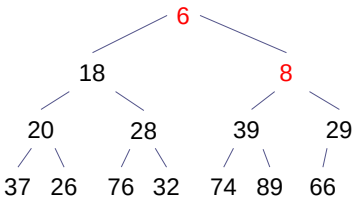
Inserting into a Heap Implemented as an `ArrayList` (cont.)



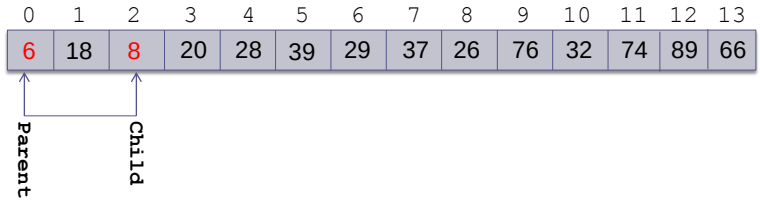
3. while (`parent >= 0` and `table[parent] > table[child]`)
4. Swap `table[parent]` and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`



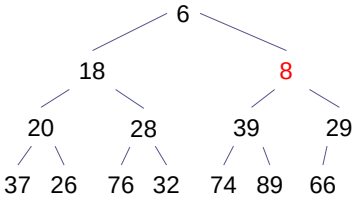
Inserting into a Heap Implemented as an `ArrayList` (cont.)



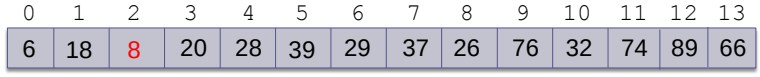
3. while (`parent >= 0` and `table[parent] > table[child]`)
4. Swap `table[parent]` and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`



Inserting into a Heap Implemented as an `ArrayList` (cont.)



3. while (`parent >= 0` and `table[parent] > table[child]`)
4. Swap `table[parent]` and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`



Removal from a Heap Implemented as an `ArrayList`

Removing an Element from a Heap Implemented as an ArrayList

1. Remove the last element (i.e., the one at size() – 1) and set the item at 0 to this value.
2. Set `parent` to 0.
3. **`while (true)`**
4. Set `leftChild` to (2 * `parent`) + 1 and `rightChild` to `leftChild` + 1.
5. **`if leftChild >= table.size()`**
6. Break out of loop.
7. Assume `minChild` (the smaller child) is `leftChild`.
8. **`if`** `rightChild < table.size() and table[rightChild] < table[leftChild]`
9. Set `minChild` to `rightChild`.
10. **`if table[parent] > table[minChild]`**
11. Swap `table[parent]` and `table[minChild]`.
12. Set `parent` to `minChild`.

`else`

13. Break out of loop.

Performance of the Heap

- `remove` traces a path from the root to a leaf
- `insert` traces a path from a leaf to the root
- This requires at most *h* steps where *h* is the height of the tree
- The largest *full* tree of height *h* has 2^h-1 nodes
- The smallest c*omplete* tree of height *h* has $2^{(h-1)}$ nodes
- Both `insert` and `remove` are O(log *n*)

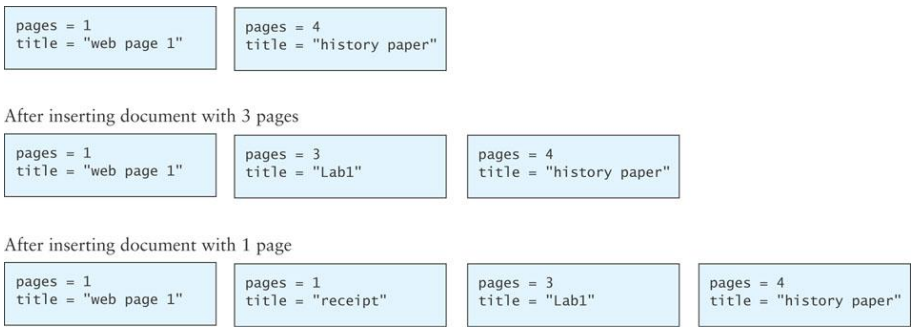
Priority Queues

- The heap is used to implement a special kind of queue called a priority queue
- The heap is not very useful as an ADT on its own
  - We will not create a `Heap` interface or code a class that implements it
  - Instead, we will incorporate its algorithms when we implement a priority queue class and heapsort
- Sometimes a FIFO queue may not be the best way to implement a waiting line
- A priority queue is a data structure in which only the highest-priority item is accessible

Priority Queues (cont.)

- In a print queue, sometimes it is more appropriate to print a short document that arrived after a very long document
- A *priority queue* is a data structure in which only the highest-priority item is accessible (as opposed to the first item entered)

Insertion into a Priority Queue



`PriorityQueue` Class

- Java provides a `PriorityQueue<E>` class that implements the `Queue<E>` interface given in Chapter 4.

Method	Behavior
`boolean offer(E item)`	Inserts an item into the queue. Returns **`true`** if successful; returns **`false`** if the item could not be inserted.
`E remove()`	Removes the smallest entry and returns it if the queue is not empty. If the queue is empty, throws a `NoSuchElementException`.
`E poll()`	Removes the smallest entry and returns it. If the queue is empty, returns **`null`**.
`E peek()`	Returns the smallest entry without removing it. If the queue is empty, returns **`null`**.
`E element()`	Returns the smallest entry without removing it. If the queue is empty, throws a `NoSuchElementException`.

Using a Heap as the Basis of a Priority Queue

- In a priority queue, just like a heap, the smallest item always is removed first
- Because heap insertion and removal is O(log *n*), a heap can be the basis of a very efficient implementation of a priority queue
- While the `java.util.PriorityQueue` uses an `Object[]` array, we will use an `ArrayList` for our custom priority queue, `KWPriorityQueue`