STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

CS 570: Data Structures

Trees (Part 2)

Instructor: Iraklis Tsekourakis

Email: itsekour@stevens.edu

Week 11

- Reading Assignment: Koffman and Wolfgang, Section 6.6

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;
    }
}
```

Implementing the `delete` Method - cont. from Notes part 9


```
/** Recursive delete method.
    post: The item is not in the tree;
          deleteReturn is equal to the deleted item
          as it was stored in the tree or null
          if the item was not found.
    @param localRoot The root of the current subtree
    @param item The item to be deleted
    @return The modified local root that does not contain
            the item
 */
private Node < E > delete(Node < E > localRoot, E item) {
  if (localRoot == null) {
    // item is not in the tree.
    deleteReturn = null;
    return localRoot;
  }
```

Implementing the `delete` Method


```
// Search for item to delete.
int compResult = item.compareTo(localRoot.data);
if (compResult < 0) {
  // item is smaller than localRoot.data.
  localRoot.left = delete(localRoot.left, item);
  return localRoot;
}
else if (compResult > 0) {
  // item is larger than localRoot.data.
  localRoot.right = delete(localRoot.right, item);
  return localRoot;
}
```

Implementing the `delete` Method


```
else {
    // item is at local root.
    deleteReturn = localRoot.data;
    if (localRoot.left == null) {
      // If there is no left child, return right child
      // which can also be null.
      return localRoot.right;
    }
    else if (localRoot.right == null) {
      // If there is no right child, return left child.
      return localRoot.left;
    }
```

Implementing the `delete` Method


```
// Node being deleted has 2 children, replace the
// data with inorder predecessor.
if (localRoot.left.right == null) {
  // The left child has no right child.
  // Replace the data with the data in the
  // left child.
  localRoot.data = localRoot.left.data;
  // Replace the left child with its left child.
  localRoot.left = localRoot.left.left;
  return localRoot;
}
```

Implementing the `delete` Method


```
else {
        // Search for the inorder predecessor (ip)
        //and replace deleted node's data with ip.
        localRoot.data =
                findLargestChild(localRoot.left);
        return localRoot;
      }
    }
  }
}
```

Method findLargestChild

LISTING 6.6

BinarySearchTree findLargestChild Method

```
/** 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

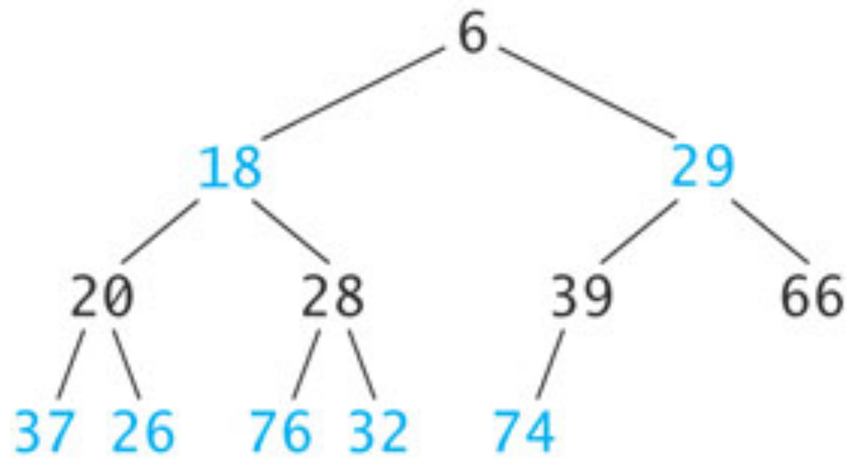
- To test a binary search tree, verify that an inorder traversal will display the tree contents in ascending order after a series of insertions and deletions are performed

Heaps and Priority Queues

Section 6.6

Heaps and Priority Queues

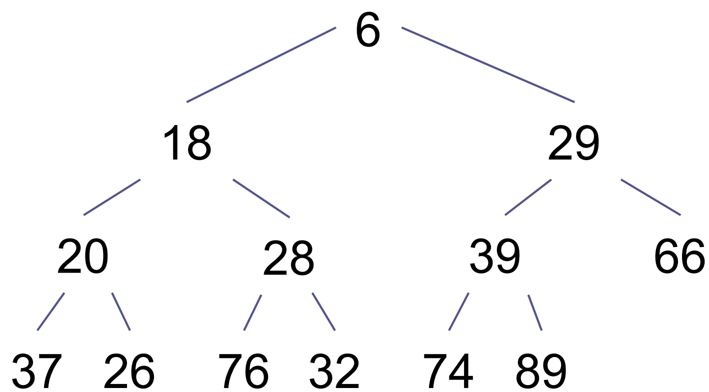
- A heap is a complete binary tree with the following properties
  - The value in the root is the smallest item in the tree
  - Every subtree is a heap



Inserting an Item into a Heap

Algorithm for Inserting in a Heap

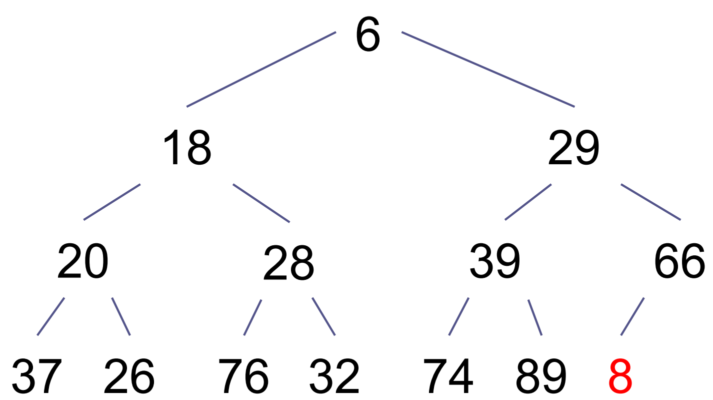
1. Insert the new item in the next position at the bottom of the heap.
2. `while` new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.



Inserting an Item into a Heap (cont.)

Algorithm for Inserting in a Heap

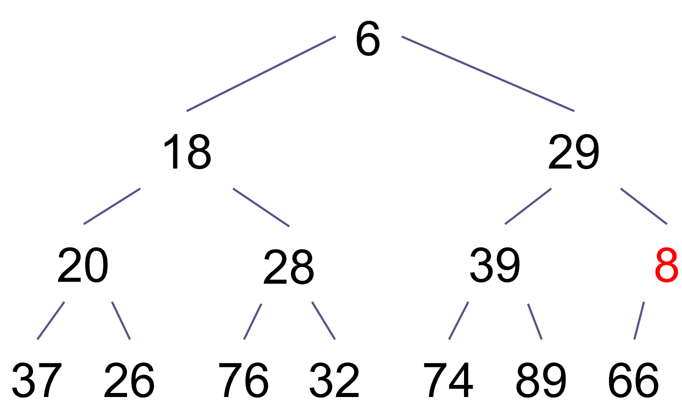
1. Insert the new item in the next position at the bottom of the heap.
2. **`while`** new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.



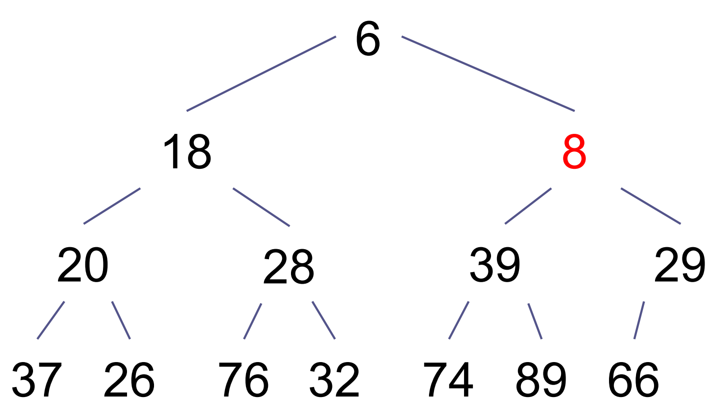
Inserting an Item into a Heap (cont.)

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.



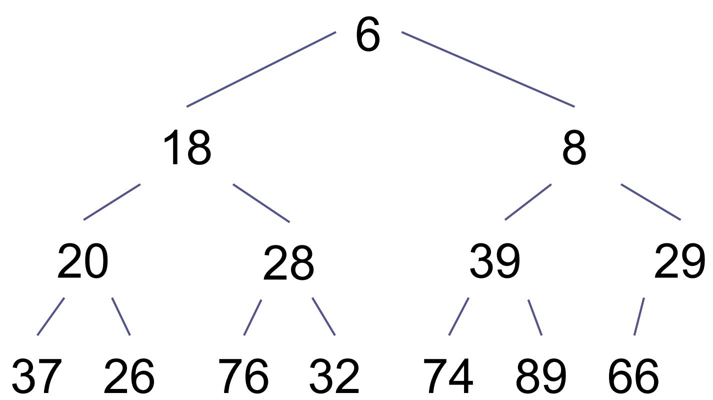
Inserting an Item into a Heap (cont.)



Removing an Item from a Heap

Algorithm for Removal from a Heap

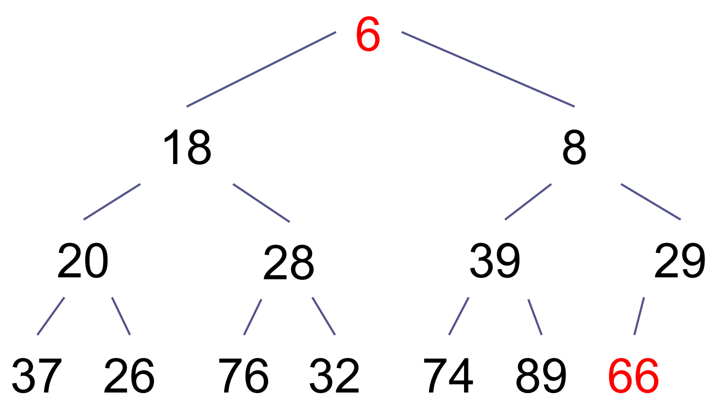
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. `while` item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

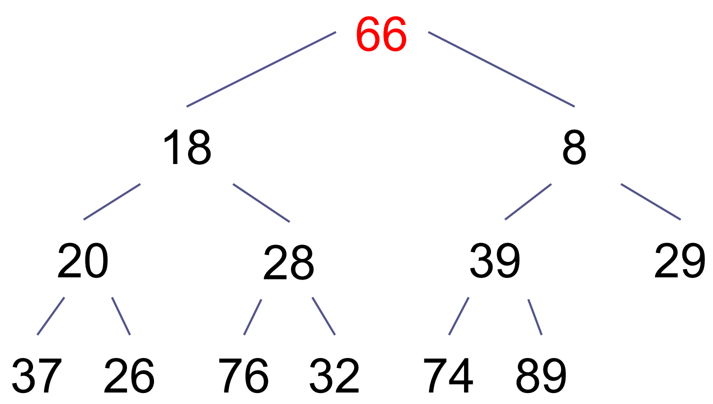
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. `while` item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

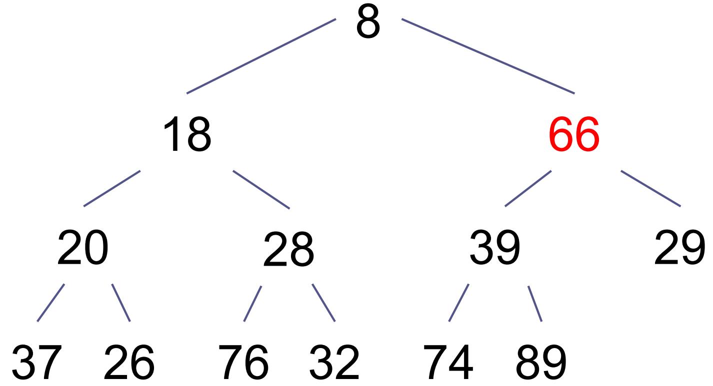
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. `while` item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

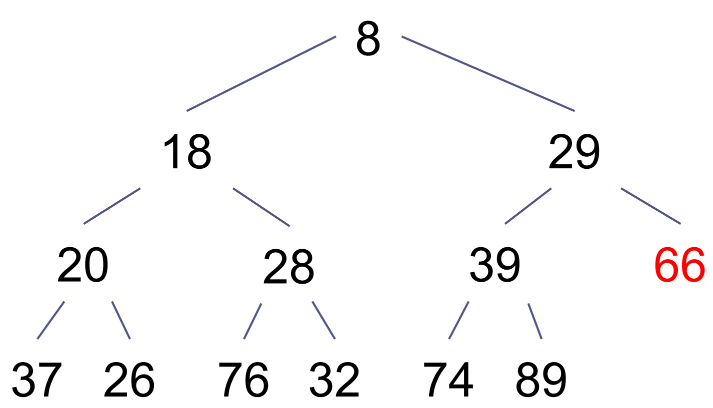
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. `while` item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

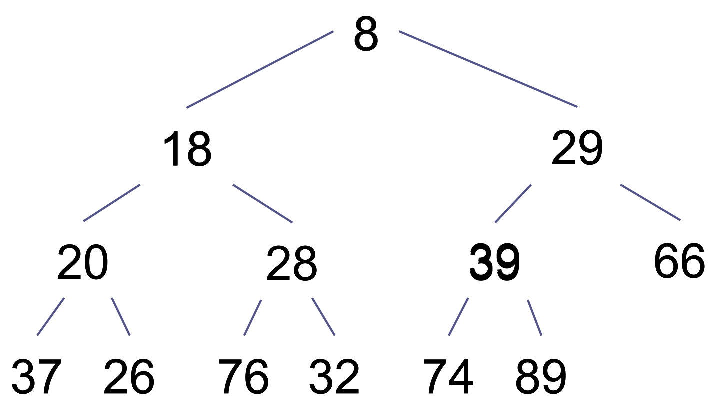
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. `while` item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.



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.)

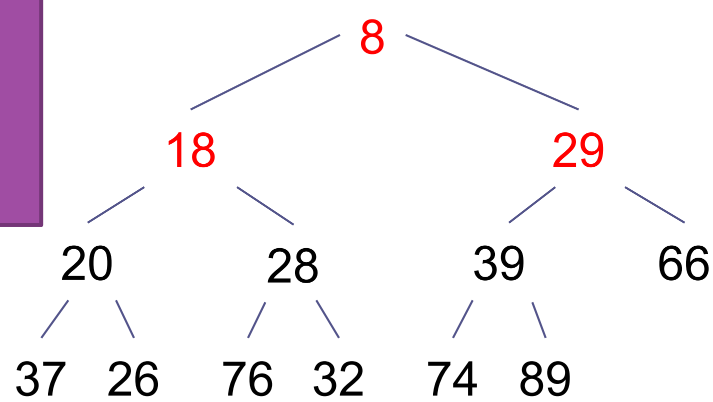


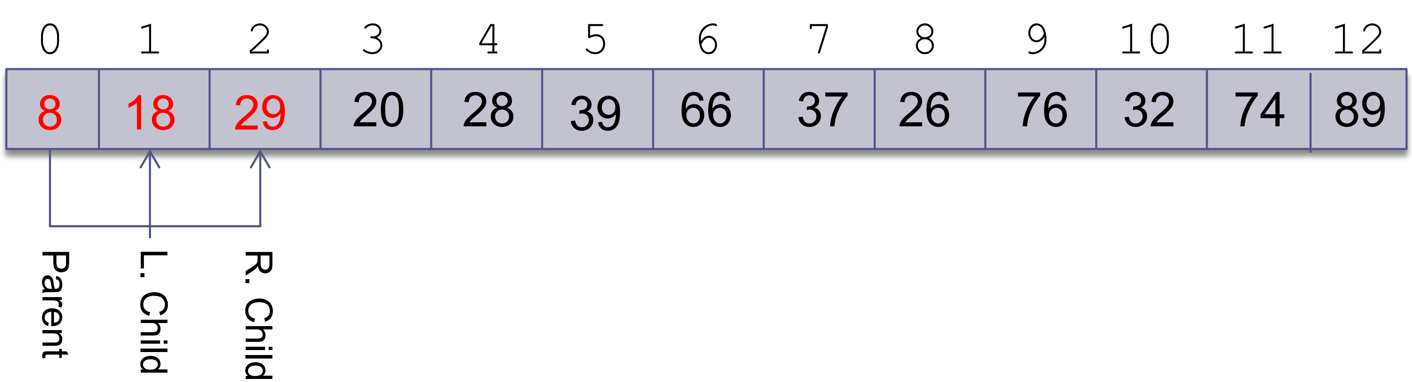
0	1	2	3	4	5	6	7	8	9	10	11	12
8	18	29	20	28	39	66	37	26	76	32	74	89

Implementing a Heap (cont.)

For a node at position *p*,

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

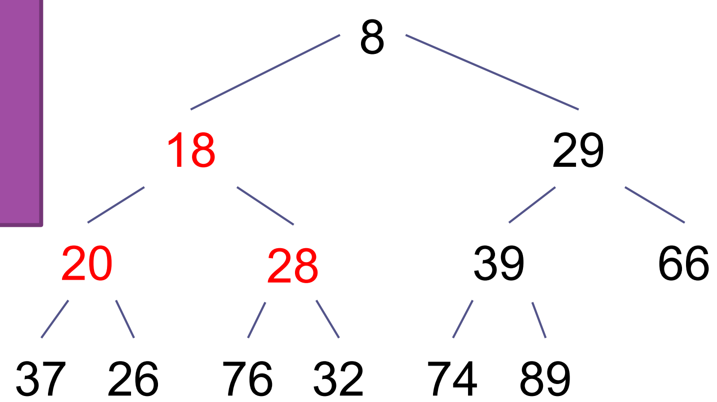


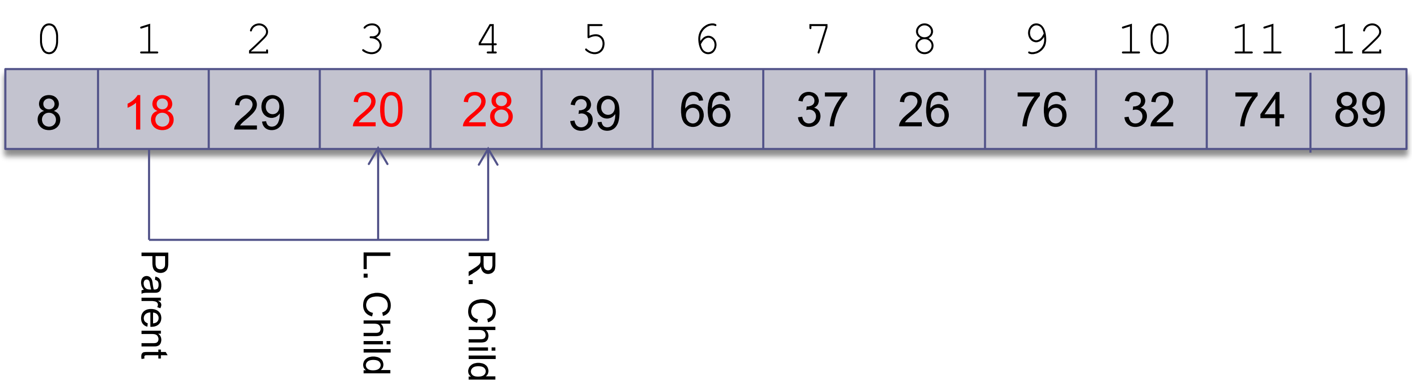


Implementing a Heap (cont.)

For a node at position p,

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

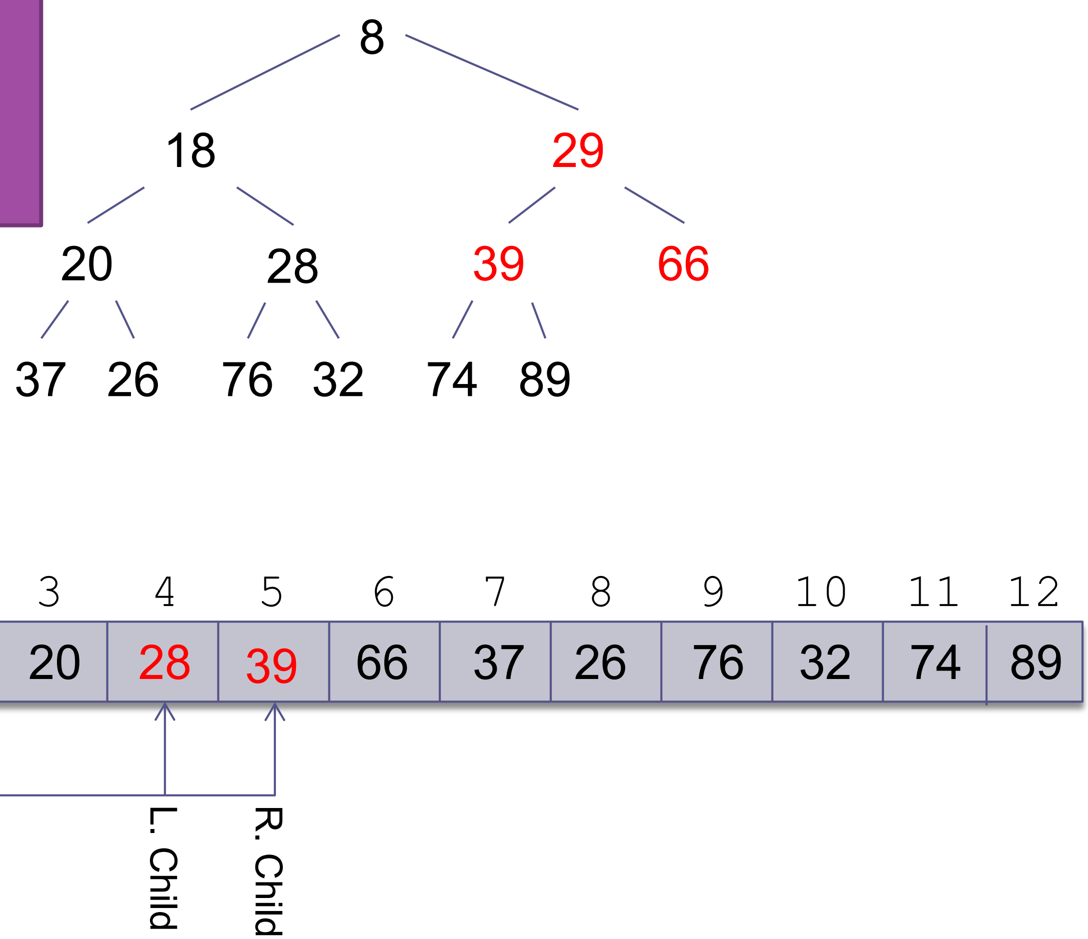




Implementing a Heap (cont.)

For a node at position p,

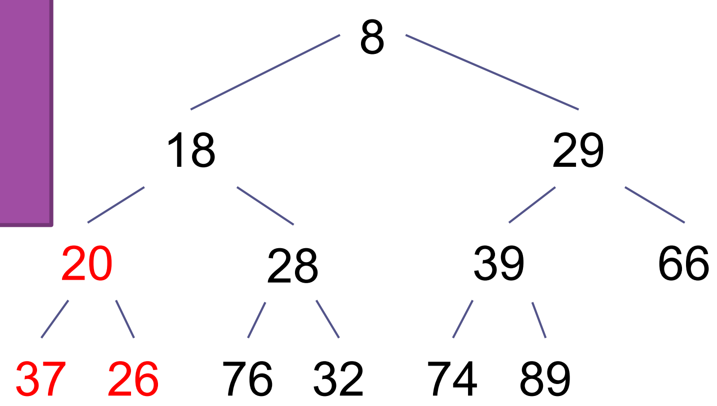
L. child position: $2p + 1$
R. child position: $2p + 2$

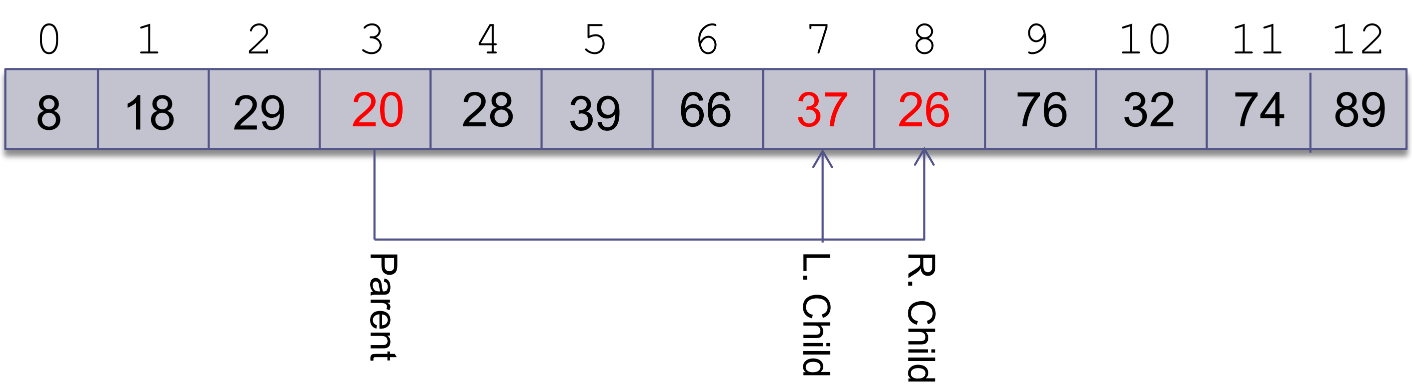


Implementing a Heap (cont.)

For a node at position *p*,

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



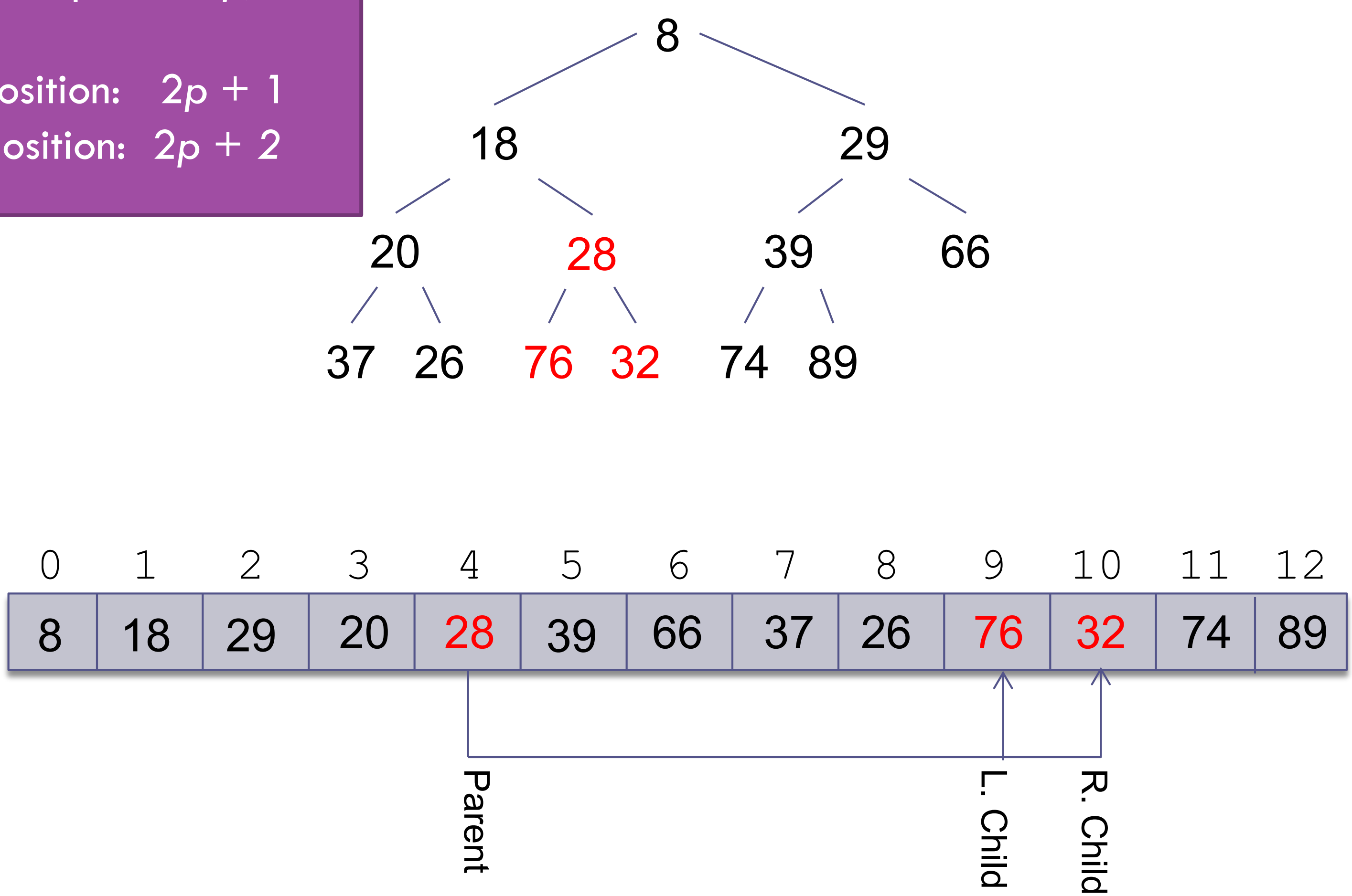


Implementing a Heap (cont.)

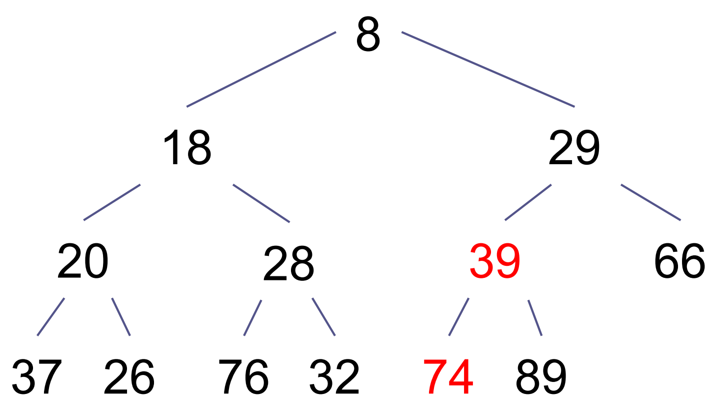
For a node at position *p*,

L. child position: $2p + 1$

R. child position: $2p + 2$



Implementing a Heap (cont.)



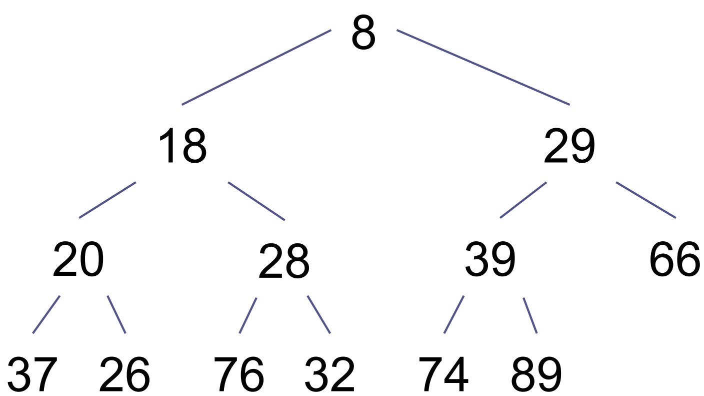
A node at position c can find its parent at $(c - 1)/2$

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

Parent

Child

Inserting into a Heap Implemented as an `ArrayList`

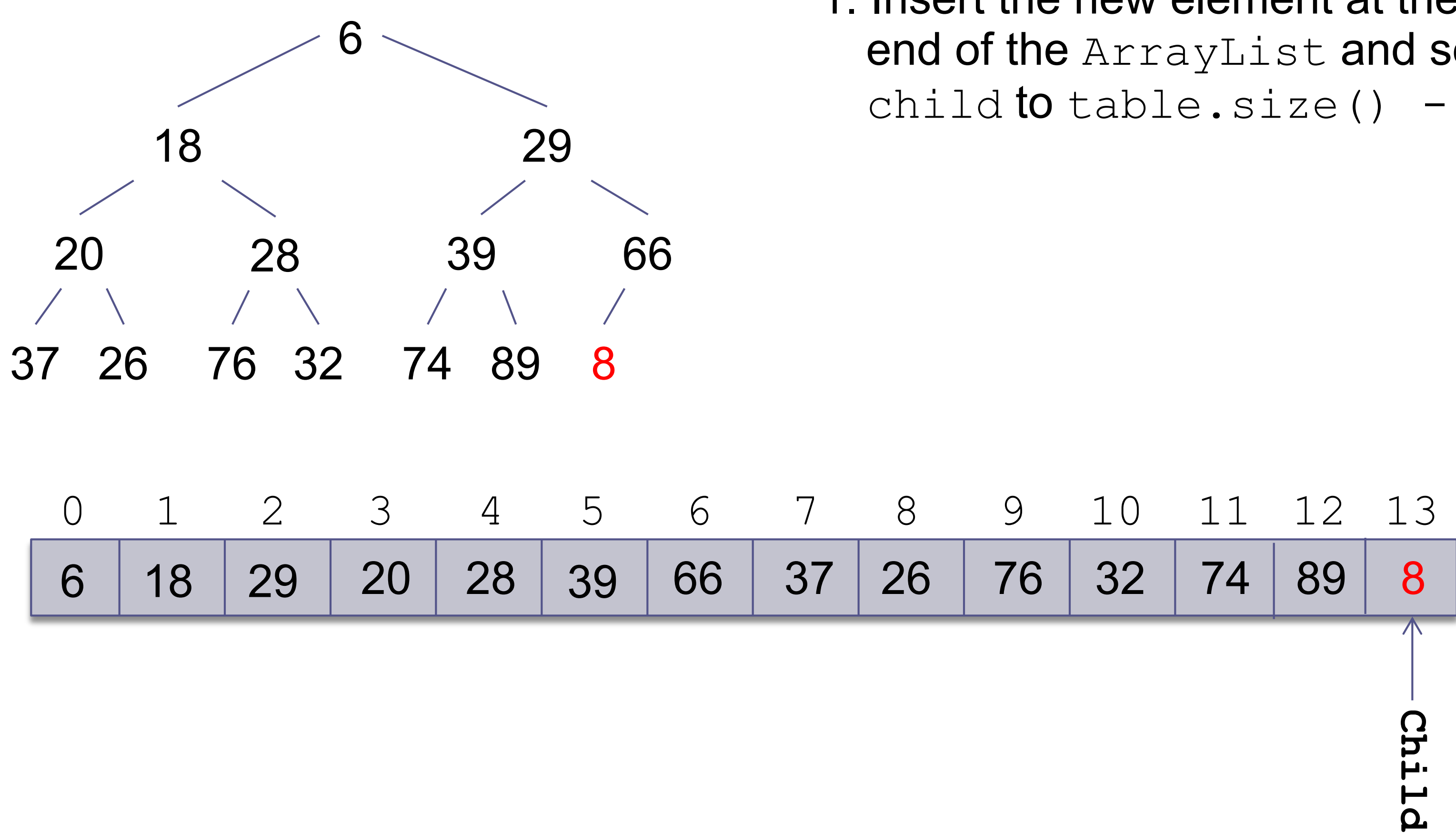


1. Insert the new element at the end of the `ArrayList` and set `child` to `table.size() - 1`

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	29	20	28	39	66	37	26	76	32	74	89	

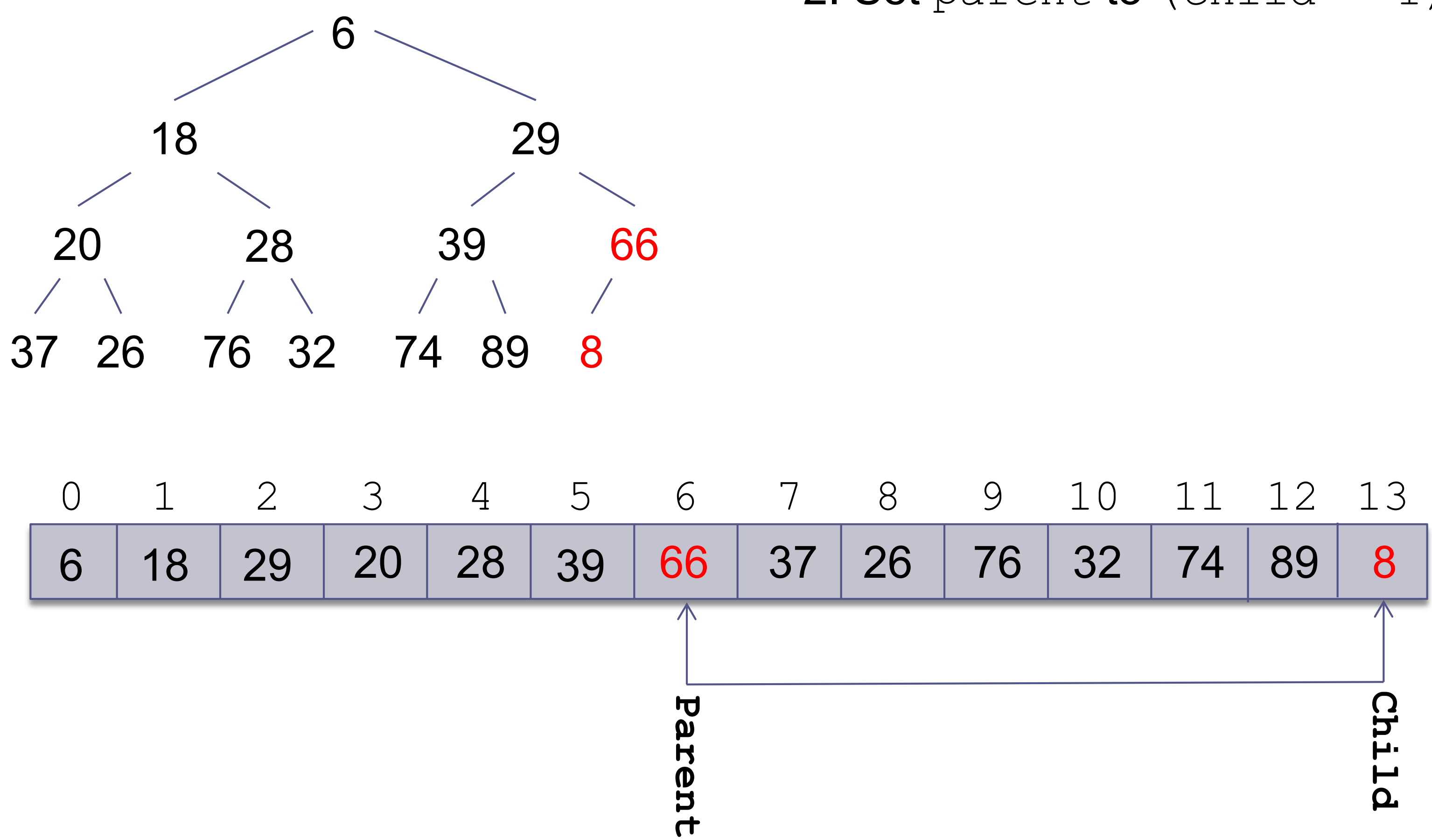
Inserting into a Heap Implemented as an `ArrayList` (cont.)

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

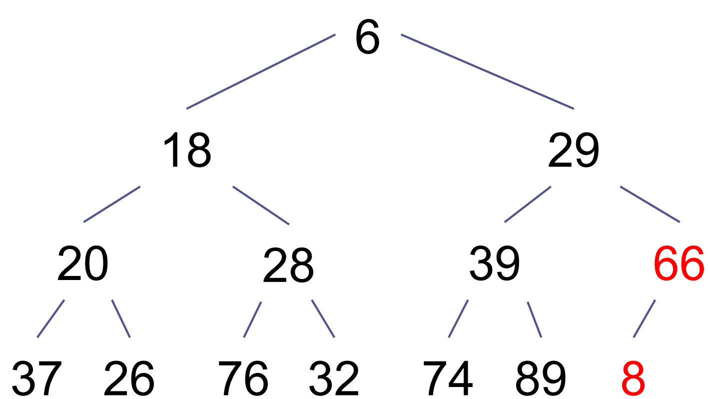


Inserting into a Heap Implemented as an `ArrayList` (cont.)

2. Set `parent` to `(child – 1)/ 2`



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



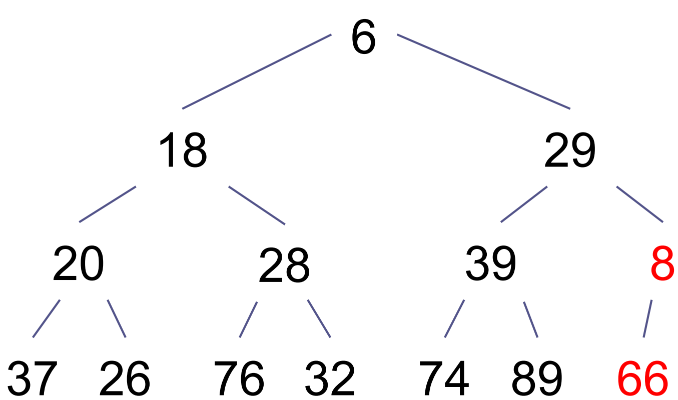
3. while (`parent >= 0`
   `and`
   `table[parent] > table[child]`)
4. Swap `table[parent]`
   and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`

0	1	2	3	4	5	6	7	8	9	10	11	12	13
6	18	29	20	28	39	66	37	26	76	32	74	89	8

`Parent`

`Child`

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

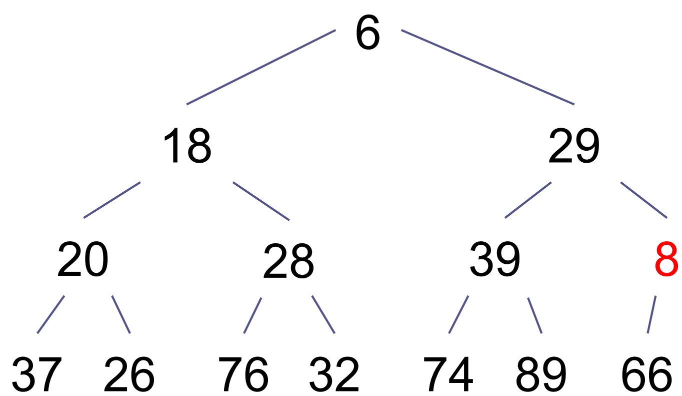


3. while (`parent >= 0`
   `and`
   `table[parent] > table[child]`)
4. Swap `table[parent]` and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`

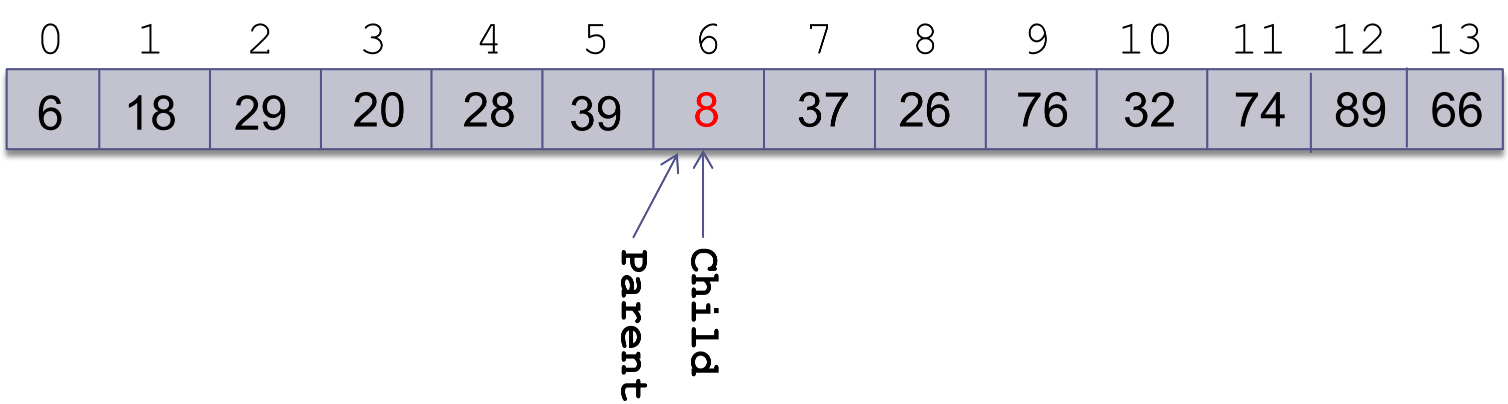
0	1	2	3	4	5	6	7	8	9	10	11	12	13
6	18	29	20	28	39	8	37	26	76	32	74	89	66

`Parent` `Child`

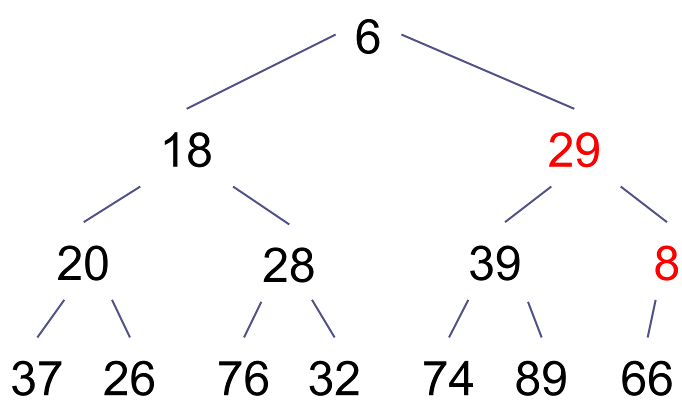
Inserting into a Heap Implemented as an `ArrayList` (cont.)



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.)



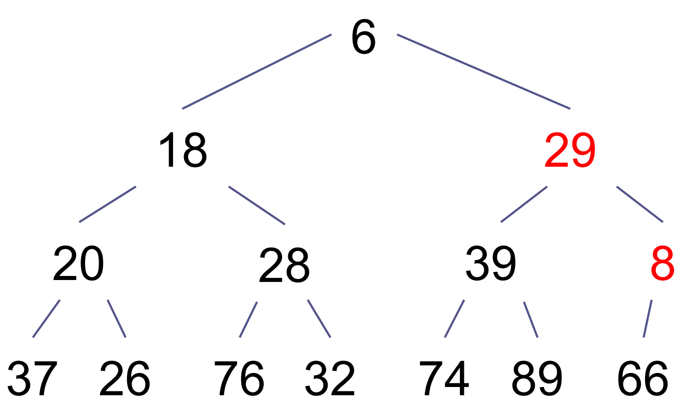
3. while (`parent >= 0`
   `and`
   `table[parent] > table[child]`)
4. Swap `table[parent]`
   and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`

0	1	2	3	4	5	6	7	8	9	10	11	12	13
6	18	29	20	28	39	8	37	26	76	32	74	89	66

Parent

Child

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

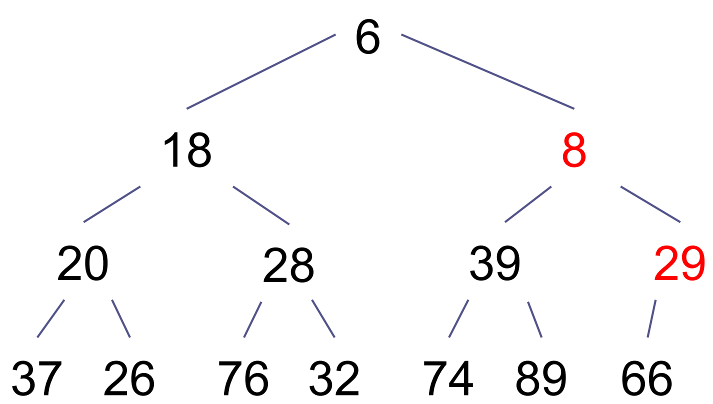


3. while (`parent >= 0`
   `and`
   `table[parent] > table[child]`)
4. Swap `table[parent]`
   and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`

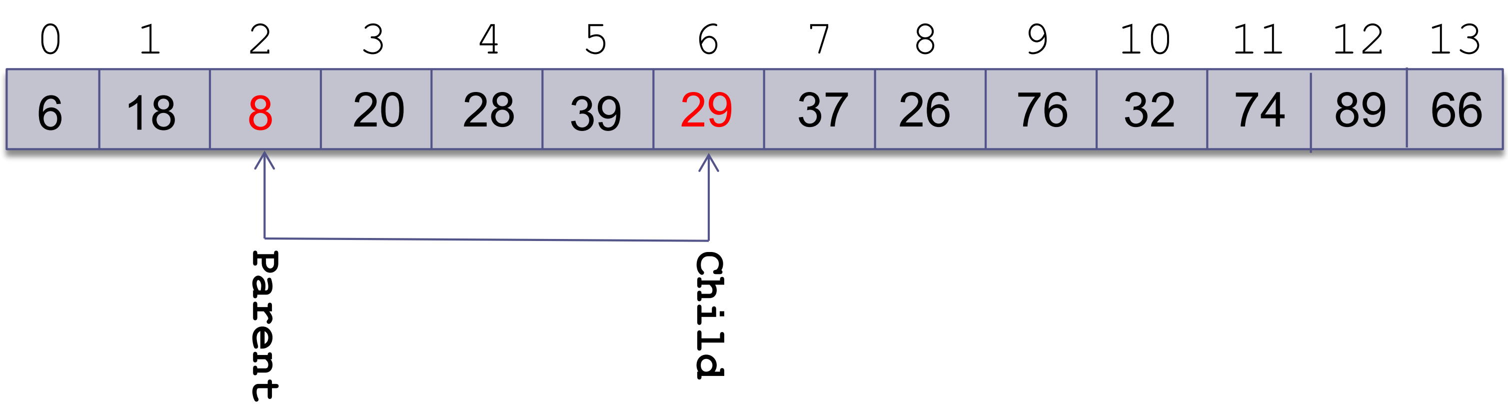
0	1	2	3	4	5	6	7	8	9	10	11	12	13
6	18	29	20	28	39	8	37	26	76	32	74	89	66

`Parent` **`Child`**

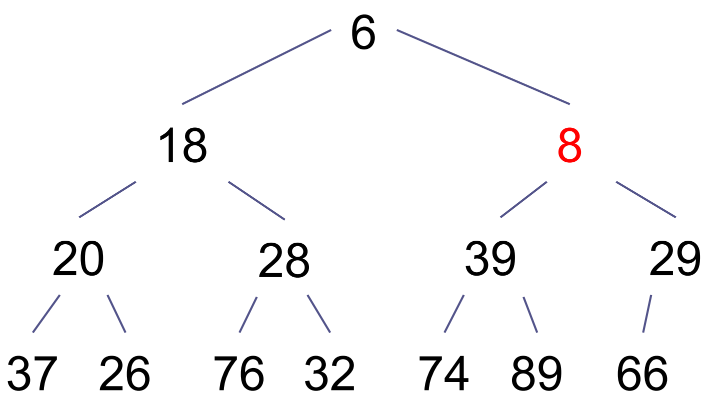
Inserting into a Heap Implemented as an `ArrayList` (cont.)



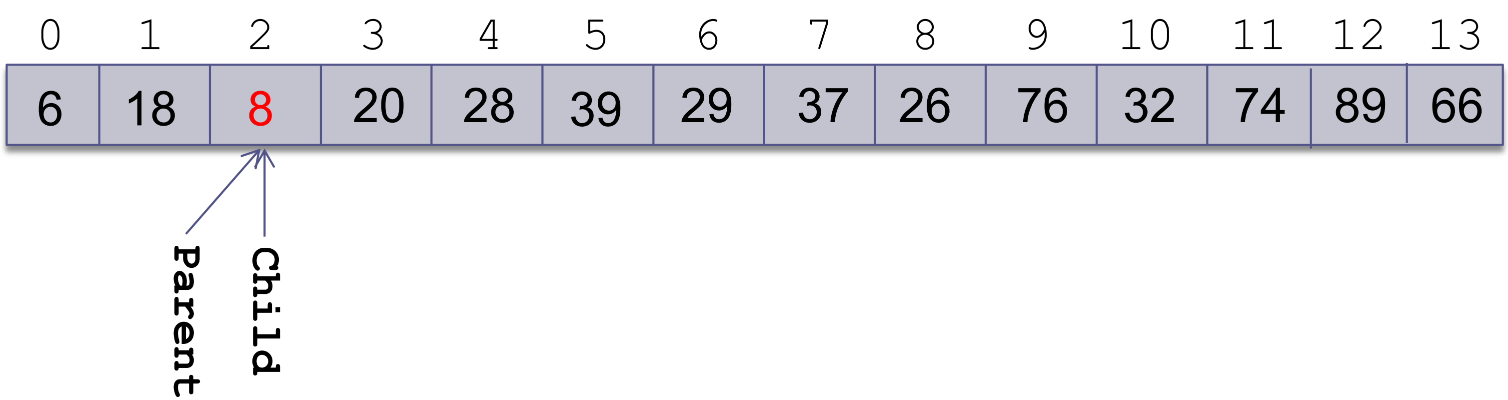
3. while (`parent >= 0`
   `and`
   `table[parent] > table[child]`)
4. Swap `table[parent]` and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`



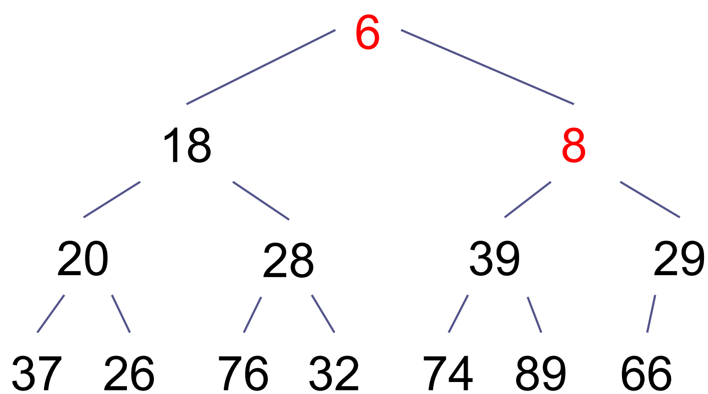
Inserting into a Heap Implemented as an `ArrayList` (cont.)



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.)

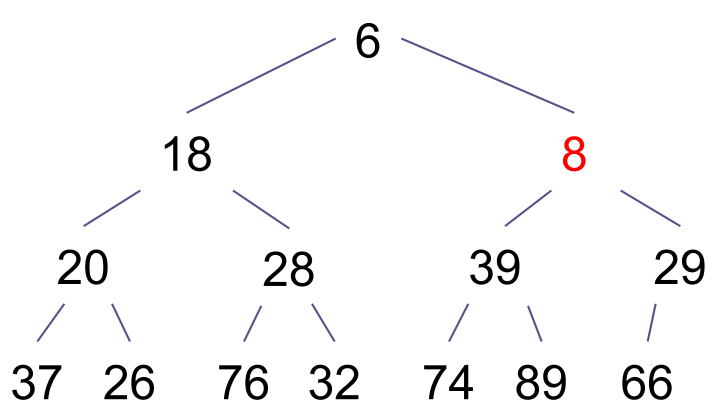


3. while (`parent >= 0`
   `and`
   `table[parent] > table[child]`)
4. Swap `table[parent]`
   and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to `(child-1)/2`

0	1	2	3	4	5	6	7	8	9	10	11	12	13
6	18	8	20	28	39	29	37	26	76	32	74	89	66

`Parent` **`Child`**

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`

0	1	2	3	4	5	6	7	8	9	10	11	12	13
6	18	8	20	28	39	29	37	26	76	32	74	89	66

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.

Removal from a Heap Implemented as an `ArrayList`

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 *complete* 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 quicker 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


```
pages = 1                  pages = 4
title = "web page 1"       title = "history paper"
```

After inserting document with 3 pages

```
pages = 1                  pages = 3                  pages = 4
title = "web page 1"       title = "Lab1"             title = "history paper"
```

After inserting document with 1 page

```
pages = 1                  pages = 1                  pages = 3                  pages = 4
title = "web page 1"       title = "receipt"          title = "Lab1"             title = "history paper"
```

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`

Design of a `KWPriorityQueue` Class

Data Field	Attribute
ArrayList<E> theData	An ArrayList to hold the data.
Comparator<E> comparator	An optional object that implements the Comparator<E> interface by providing a compare method.
Method	**Behavior**
KWPriorityQueue()	Constructs a heap-based priority queue that uses the elements' natural ordering.
KWPriorityQueue (Comparator<E> comp)	Constructs a heap-based priority queue that uses the compare method of Comparator comp to determine the ordering of the elements.
private int compare(E left, E right)	Compares two objects and returns a negative number if object left is less than object right, zero if they are equal, and a positive number if object left is greater than object right.
private void swap(int i, int j)	Exchanges the object references in theData at indexes i and j.

Design of a KWPriorityQueue Class (cont.)


```
import java.util.*;
/** The KWPriorityQueue implements the Queue interface
    by building a heap in an ArrayList. The heap is
    structured so that the "smallest" item is at the top.
*/
public class KWPriorityQueue<E> extends AbstractQueue<E>
                                 implements Queue<E> {
// Data Fields
/** The ArrayList to hold the data. */
private ArrayList<E> theData;
/** An optional reference to a Comparator object. */
Comparator<E> comparator = null;

// Methods
// Constructor
public KWPriorityQueue() {
    theData = new ArrayList<E>();
}
```

offer Method


```
/** Insert an item into the priority queue.
    pre: The ArrayList theData is in heap order.
    post: The item is in the priority queue and
      theData is in heap order.
    @param item The item to be inserted
    @throws NullPointerException if the item to be
      inserted is null.
*/
@Override
public boolean offer(E item) {
    // Add the item to the heap.
    theData.add(item);
```

`offer` Method


```
    // child is newly inserted item.
    int child = theData.size() - 1;
    int parent = (child - 1) / 2; // Find child's parent.
    // Reheap
    while (parent >= 0 && compare(theData.get(parent),
            theData.get(child)) > 0) {
        swap(parent, child);
        child = parent;
        parent = (child - 1) / 2;
    }
    return true;
}
```

poll Method


```
/** Remove an item from the priority queue
    pre: The ArrayList theData is in heap order.
    post: Removed smallest item, theData is in heap
      order.
    @return The item with the smallest priority value
      or null if empty.
*/
@Override
public E poll() {
    if (isEmpty()) {
      return null;
    }
```

`poll` Method


```
// Save the top of the heap.
E result = theData.get(0);
// If only one item then remove it.
if (theData.size() == 1) {
    theData.remove(0);
    return result;
}

/* Remove the last item from the ArrayList and
    place it into the first position. */
theData.set(0, theData.remove(theData.size() - 1));
// The parent starts at the top.
int parent = 0;
```

`poll` **Method** (cont.)

```
while (true) {
    int leftChild = 2 * parent + 1;
    if (leftChild >= theData.size()) {
    break; // Out of heap.
    }
    int rightChild = leftChild + 1;
    // Assume leftChild is smaller.
    int minChild = leftChild;
    // See whether rightChild is smaller.
    if (rightChild < theData.size()
        && compare(theData.get(leftChild),
        theData.get(rightChild)) > 0) {
          minChild = rightChild;
    }
```

`poll` **Method** (cont.)

```
        // assert: minChild is the index of the
        // smaller child.
        // Move smaller child up heap if necessary.
        if (compare(theData.get(parent),
                theData.get(minChild)) > 0) {
            swap(parent, minChild);
            parent = minChild;
        }
        else
        { // Heap property is restored.
            break;
        }
    }
    return result;
}
```

Other Methods

- The `iterator` and `size` methods are implemented via delegation to the corresponding `ArrayList` methods
- Method `isEmpty` tests whether the result of calling method `size` is `0` and is inherited from class `AbstractCollection`
- The implementations of methods `peek` and `remove` are left as exercises

Using a `Comparator`

- To use an ordering that is different from the natural ordering, provide a constructor that has a `Comparator<E>` parameter

```
/** Creates a heap-based priority queue with the specified
  initial capacity that orders its elements according to the
  specified comparator.
    @param cap The initial capacity for this priority queue
    @param comp The comparator used to order this priority
  queue
    @throws IllegalArgumentException if cap is less than 1
*/
```

Using a `Comparator`


```
public KWPriorityQueue(int cap, Comparator<E> comp) {
    if (cap < 1)
      throw new IllegalArgumentException();
    theData = new ArrayList<E>();
    comparator = comp;
}
```

`compare` Method

- If data field `comparator` references a `Comparator<E>` object, method `compare` delegates the task to the object's `compare` method
- If comparator is `null`, it will delegate to method `compareTo`

compare Method (cont.)


```
/** Compare two items using either a Comparator object's compare
method or their natural ordering using method compareTo.
    pre: If comparator is null, left and right implement
       Comparable<E>.
    @param left One item
    @param right The other item
    @return Negative int if left less than right,
          0 if left equals right,
          positive int if left > right
    @throws ClassCastException if items are not Comparable
*/
private int compare(E left, E right) {
    if (comparator != null) { // A Comparator is defined.
       return comparator.compare(left, right);
    } else {                  // Use left's compareTo method.
       return ((Comparable<E>) left).compareTo(right);
    }
}
```

`PrintDocuments` **Example**

- The class `PrintDocument` is used to define documents to be printed on a printer
- We want to order documents by a value that is a function of both size and time submitted
- In the client program, use

```
Queue printQueue =
    new PriorityQuene(new ComparePrintDocuments());
```

PrintDocuments **Example (cont.)**

LISTING 6.8

ComparePrintDocuments.java

```java
import java.util.Comparator;

/** Class to compare PrintDocuments based on both
    their size and time stamp.
*/
public class ComparePrintDocuments implements Comparator<PrintDocument> {
    /** Weight factor for size. */
    private static final double P1 = 0.8;
    /** Weight factor for time. */
    private static final double P2 = 0.2;

    /** Compare two PrintDocuments.
        @param left The left-hand side of the comparison
        @param right The right-hand side of the comparison
```

`PrintDocuments` Example (cont.)


```
        @return -1 if left < right; 0 if left == right;
                and +1 if left > right
    */
    public int compare(PrintDocument left, PrintDocument right) {
        return Double.compare(orderValue(left), orderValue(right));
    }

    /** Compute the order value for a print document.
        @param pd The PrintDocument
        @return The order value based on the size and time stamp
    */
    private double orderValue(PrintDocument pd) {
        return P1 * pd.getSize() + P2 * pd.getTimeStamp();
    }
}
```