Heaps and Priority Queues

• Heap Definition and Operations
• Using Heaps
  – Heap sort
  – Priority queues
• Implementing heaps
  – With links
  – With arrays
  – Analysis of heap implementations
• Reading: L&C 12.1 – 12.5
Heap Definition

• A heap is a binary tree with added properties
  – It is a complete tree (as defined earlier)
  – For each node, the parent is less than or equal to both its left and its right child (a min heap) OR
  – For each node, the parent is greater than or equal to both its left and its right child (a max heap)

• For simplicity, we will study only a min heap

• Class of objects stored in a heap may implement Comparable interface or a Comparator may be used
## Heap Operations

### BinaryTreeADT\(<T>\)\
- `removeLeftSubtree( ) : void`
- `removeRightSubtree( ) : void`
- `removeAllElements( ) : void`
- `isEmpty( ) : boolean`
- `size( ) : int`
- `contains( ) : boolean`
- `find( ) : T`
- `toString( ) : String`
- `iteratorInOrder( ) : Iterator<T>`
- `iteratorPreOrder( ) : Iterator<T>`
- `iteratorPostOrder( ) : Iterator<T>`
- `iteratorLevelOrder( ) : Iterator<T>`

### HeapADT\(<T>\)\
- `addElement(element : T) : void`
- `removeMin() : T`
- `findMin( ) : T`

*Note: toString is missing in L&C Fig 11.2*
Heap Operations

• Notice that we are extending BinaryTreeADT – not BinarySearchTreeADT
• We don’t use BinarySearchTree operations
• We define different heap operations to add and remove elements according to the new definition of the storage order for a heap
  – addElement \( O(\log n) \)
  – RemoveMin \( O(\log n) \)
  – findMin \( O(1) \)
Heap Operations

• Semantics for `addElement` operation
  – Location of new node

Next insertion point
Heap Operations

• Semantics for addElement operation
  – Add 2 in the next available position
  – Reorder (“heapify on add”)
    • Compare new node with parent and swap if needed
    • Continue up the tree until at root if necessary
Heap Operations

- Semantics of removeMin operation
  - Remove root node
  - Replace root with last leaf
  - Reorder the heap ("heapify on remove")

```
Remove
3
5
4
8
7
9

Replace root with

Reorder
9
5
4
8
7

4
5
9
8
7
```
Heap Operations

• Heapify on Remove
  – Start at root
  – Find smallest child
  – If root greater than smallest child (min heap), swap parent with smallest child
  – Continue at child node until no swap needed
Heap Operations

• Semantics of findMin operation
  – The min element is always at the root element
  – Just return a reference to the root’s element

Return a reference to root
Using Heaps

• **Heap Sort**
  – Add each element to a heap
  – Remove them one at a time

• **Sort Order**
  – Min heap $\rightarrow$ ascending order
  – Max heap $\rightarrow$ descending order

• **Performance**
  – $O(n \log n)$
Using Heaps

• Priority Queue Applications
  – Task scheduling in an operating system
  – Traffic scheduling on a network link
  – Job scheduling at a service center
  – Creating a Huffman coding tree
• Write the compareTo method to compare priorities which heap logic uses first
• If there is a tie on priority, heap logic uses order of entry (FIFO within priority)
Implementing Heaps with Links

- `<interface>> HeapADT<T>`

- `Heap<T>`
  - `# lastNode : HeapNode<T>`

- `HeapNode<T>`
  - `# parent : HeapNode<T>`

- `LinkedBinaryTree<T>`
  - `BinaryTreeNode<T>`
Implementing Heaps with Links

• **Class HeapNode<T>**
  
  ```java
  public class HeapNode<T> extends BinaryTreeNode<T> {
      protected HeapNode<T> parent;
      public HeapNode (T obj) {
          super(obj);
          parent = null;
      }
  }
  ```

• **Class Heap<T>**
  
  ```java
  public class Heap<T> extends LinkedBinaryTree<T> implements HeapADT<T> {
      private HeapNode<T> lastNode;
  }
  ```
Implementing Heaps with Links

• Heap<T> addElement operation

```java
public void addElement (T obj)
{
    HeapNode<T> node = new HeapNode<T>(obj);

    if (root == null)
        root=node;
    else
    {
        HeapNode<T> next_parent = getNextParentAdd();
        if (next_parent.getLeft() == null);
            next_parent.setLeft(node);
        else
            next_parent.setRight(node);
        node.setParent(next_parent);
    }
}
```
Implementing Heaps with Links

• Heap<T> addElement operation

```java
lastNode = node;
count++;
if (count > 1)
    heapifyAdd();
}
private void heapifyAdd()       // a helper method
{
    HeapNode<T> next = lastNode;
    while ((next != root) &&
           (((Comparable)next.getElement()).compareTo
            (next.getParent().getElement()) < 0))
    {
        T temp = next.getElement();
        next.setElement(next.getParent().getElement());
        next.getParent().setElement(temp);
        next = next.getParent();
    }
}
```
Implementing Heaps with Links

• Heap<T> addElement operation
  
  private HeapNode<T> getNextParentAdd() //a helper
  {
    HeapNode<T> result = lastNode;
    // go up right branch as needed
    while (result != root &&
      result.getParent().getLeft() != result)
      result = result.getParent();

    // at top of a left branch or at root now
    if (result == root)
      // at root, go down left branch to lower left
      while (result.getLeft() != null)
        result = result.getLeft();
    else
Implementing Heaps with Links

- **Heap<T> addElement operation**

  ```java
  { // need to go over the hump to a right branch
    if (result.getParent().getRight() == null)
      // parent is the next parent
      result = result.getParent();
    else {
      // parent is an ancestor of the next parent
      result = (HeapNode<T>)result.getParent().
                   getRight();
      // go down its left branch to bottom level
      while result.getLeft() != null)
        result = (HeapNode<T>) result.getLeft();
    }
  }
  return result;
  ```
Implementing Heaps with Links

- Heap<T> removeMin operation

```java
public T removeMin() throws EmptyCollectionException {
    if (isEmpty())
        throw new EmptyCollectionException("Empty heap");
    T minElement = root.getElement();
    if (count == 1)
        root = null;
        lastNode = null;
    else
        `
Implementing Heaps with Links

• Heap<T> removeMin operation

```java
{ 
    HeapNode<T> next_last = getNewlastNode();
    if (lastNode.getParent().getLeft() == lastNode)
        lastNode.getParent().setLeft(null);
    else
        lastNode.getParent().setRight(null);
    root.setElement(lastNode.getElement());
    lastNode = next_last;
    heapifyRemove();
}
count--;
return minElement;
}
```
Implementing Heaps with Links

- **Heap<T> removeMin operation**

  ```java
  private HeapNode<T> getNewLastNode() // a helper
  {
    HeapNode<T> result = lastNode;
    while (result != root &&
          result.getParent().getLeft() == result)
      result = result.getParent();
    if (result != root)
      result = (HeapNode<T>) result.getParent().getLeft();
    while (result.getRight() != null)
      result = (HeapNode<T>) result.getRight();
    return result;
  }
  ```
Implementing Heaps with Links

- Heap<T> removeMin operation

```java
private void heapifyRemove()  // a helper method
{
    if (root == null)
        return;
    HeapNode<T> node = null;
    HeapNode<T> next = (HeapNode<T>)root;
    do
    {
        if (node != null) {       // skip on first pass
            T temp = node.getElement();
            node.SetElement(next.getElement());
            next.setElement(temp);
        }
        node = next;
    }
    node = next;
```
Implementing Heaps with Links

• Heap\(<T>\) removeMin operation

```java
HeapNode<T> left = (HeapNode<T>)node.getLeft();
HeapNode<T> right = (HeapNode<T>)node.getRight();

if ((left == null) && (right == null))
    next = null;
else if (left == null)
    next = right;
else if (right == null)
    next = left;
else if (((Comparable)left.getElement()).compareTo(right.getElement()) < 0)
    next = left;
else
    next = right;
} while (next != null &&
    ((Comparable)next.getElement()).compareTo(node.getElement()) < 0);
```