Fundamentals of Data Structure

Mahesh Shirole

VJTI, Mumbai-19

Slides are prepared from

1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014
2. Data Structures and Algorithms in Java, by Robert Lafore, Second Edition, Sams Publishing

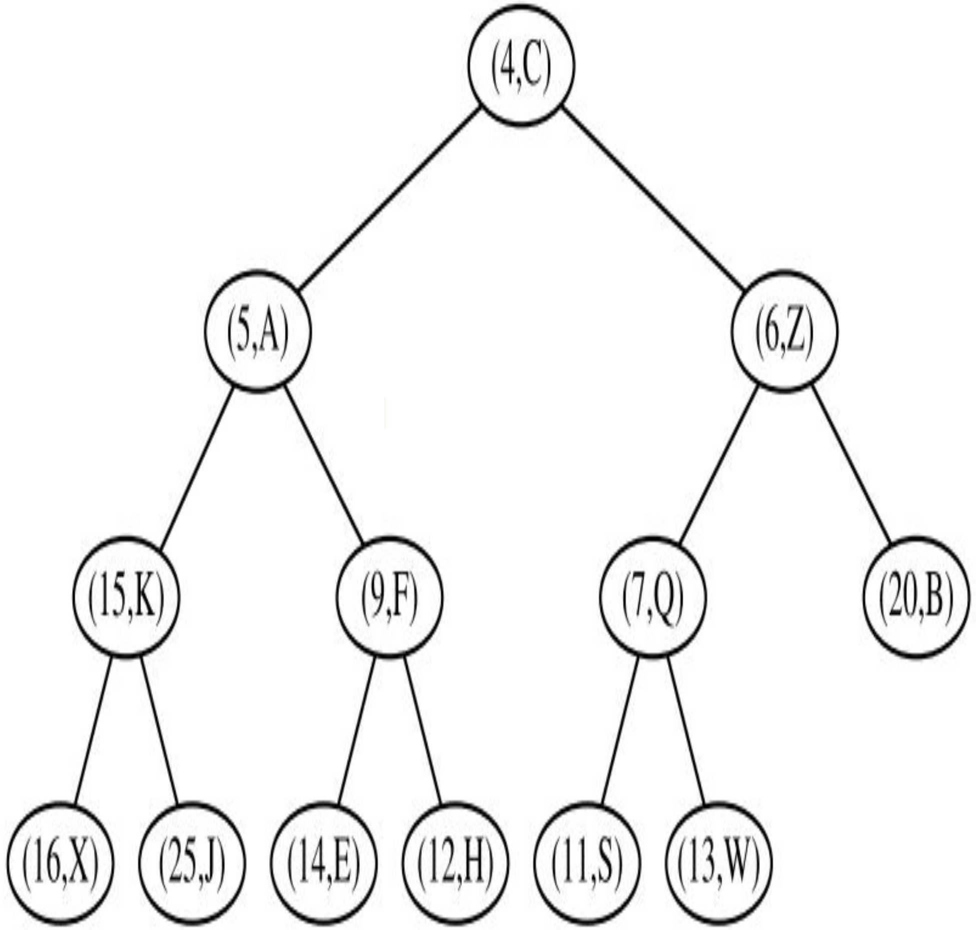
Heaps

	Insert	Search
Unsorted	On	O(1)
	O1	O(n)

- In case of priority queue implementaion
- When using an **unsorted list** to store entries
  - we can perform insertions in O(1) time
  - *finding or removing an element with minimal key requires an O(n)-time*
- When using a **sorted list** to store entries
  - find or remove the minimal element in O(1) time
  - *adding a new element to the queue may require O(n) time*
- we need a more efficient realization of a priority queue
- The answer is a data structure called a ***binary heap***
- This data structure allows us to perform both insertions and removals in ***logarithmic time***

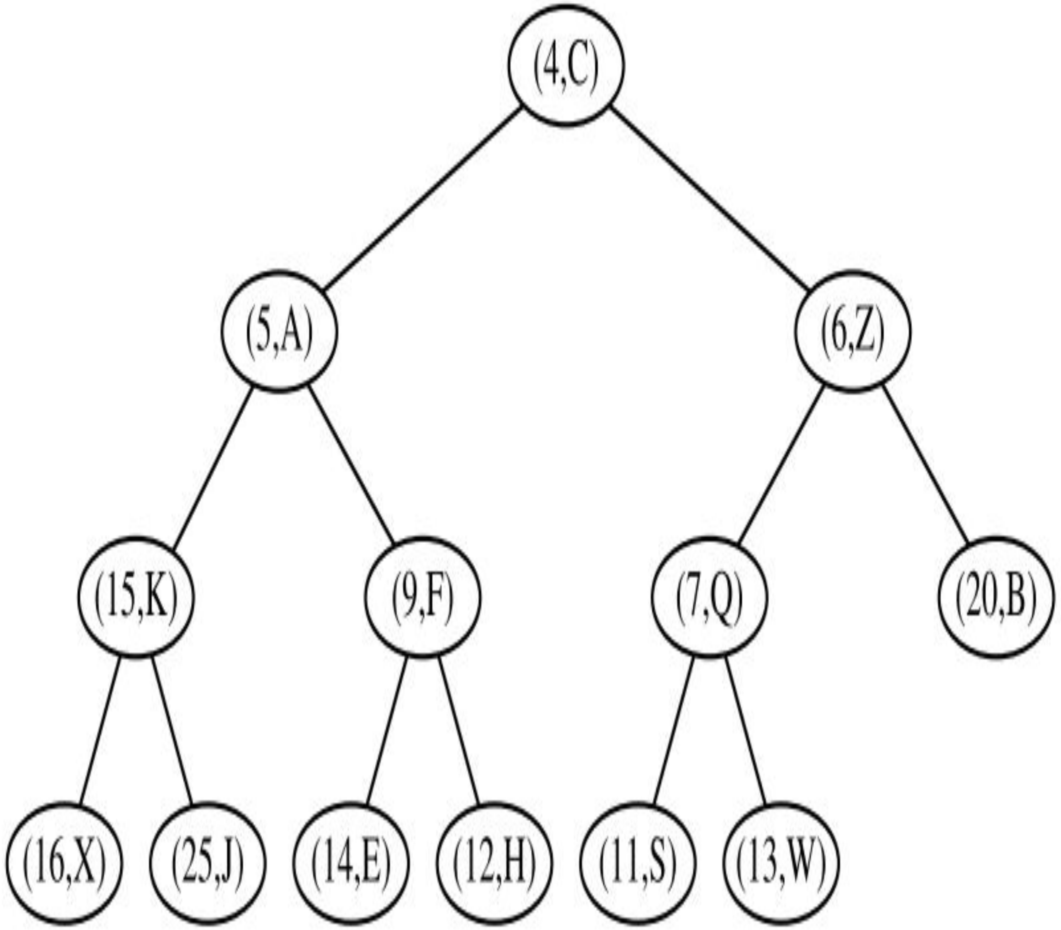
The Heap Data Structure

- A **heap** is a ***binary tree T***
- It stores entries at its positions, and that satisfies two additional properties:
  - a relational property defined in terms of the way keys are stored in ***T*** and
  - a structural property defined in terms of the shape of ***T*** itself
- **Heap-Order Property**: In a heap ***T***, for every position ***p*** other than the **root**, the key stored at ***p*** is **greater than or equal to** the key stored at ***p's*** parent
- **Complete Binary Tree Property**: A heap ***T*** with height ***h*** is a complete binary tree if levels **0,1,2, . . . ,h−1** of ***T*** have the maximal number of nodes possible (namely, level ***i*** has **2** ^{i} nodes, for $0 \leq i \leq$ **h−1**) and the remaining nodes at level **h** reside in the leftmost possible positions at that level



The Heap Data Structure

- As a consequence of the heap-order property, the keys encountered on a path from the **root** to a **leaf** of ***T*** are in ***nondecreasing order***
- A ***minimal key*** is always stored at the *root* of ***T***
- This makes it easy to locate such an entry when **min** or **removeMin** is called, as it is informally said to be "***at the top of the heap***"
- The tree in Figure is *complete because levels 0, 1, and 2 are full*, and the six nodes in level 3 are in the six leftmost possible positions at that level
- A complete binary tree with ***n*** elements is one that has positions with level numbering 0 through n−1



The Height of a Heap

- Let h denote the height of T
- **Proposition:** A heap T storing n entries has height h = ⌊logn ⌋

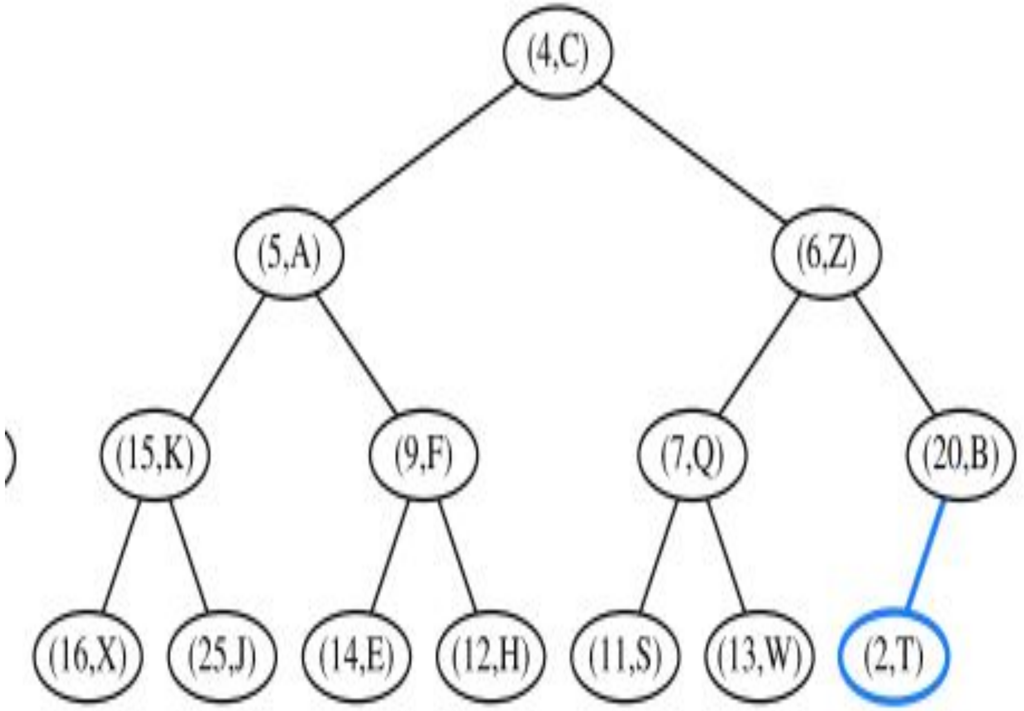
Justification: From the fact that T is complete, we know that the number of nodes in levels 0 through $h-1$ of T is precisely $1+2+4+\cdots+2^{h-1}=2^h-1$, and that the number of nodes in level h is at least 1 and at most 2^h. Therefore

$$n \geq 2^h - 1 + 1 = 2^h \qquad \text{and} \qquad n \leq 2^h - 1 + 2^h = 2^{h+1} - 1.$$

By taking the logarithm of both sides of inequality $n \geq 2^h$, we see that height $h \leq \log n$. By rearranging terms and taking the logarithm of both sides of inequality $n \leq 2^{h+1}-1$, we see that $h \geq \log(n+1)-1$. Since h is an integer, these two inequalities imply that $h = \lfloor \log n \rfloor$. ■

Implementing a Priority Queue with a Heap

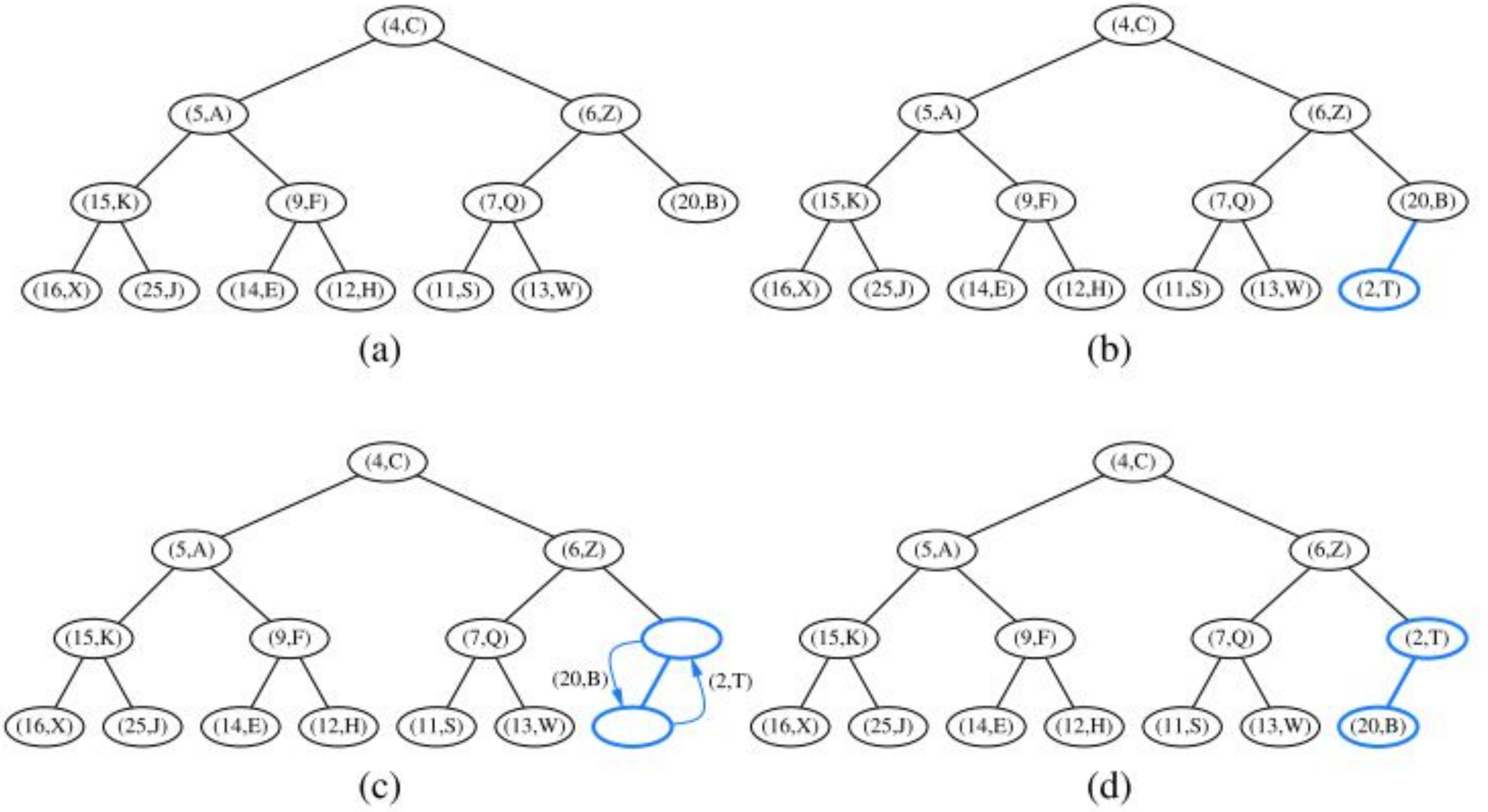
- The heap property assures that the element at the root of the tree has a minimal key, the min operation is trivial
- **Adding an Entry to the Heap**
- We store the pair (k,v) as an entry at a new node of the tree
- To maintain the *complete binary tree property*
  - new node should be placed at a position ***p*** just beyond the rightmost node at the bottom level of the tree
  - Or new node should be placed at the leftmost position of a new level, if the bottom level is already full (or if the heap is empty)
- After this action, the tree ***T*** is complete, but it may *violate the heap-order property*



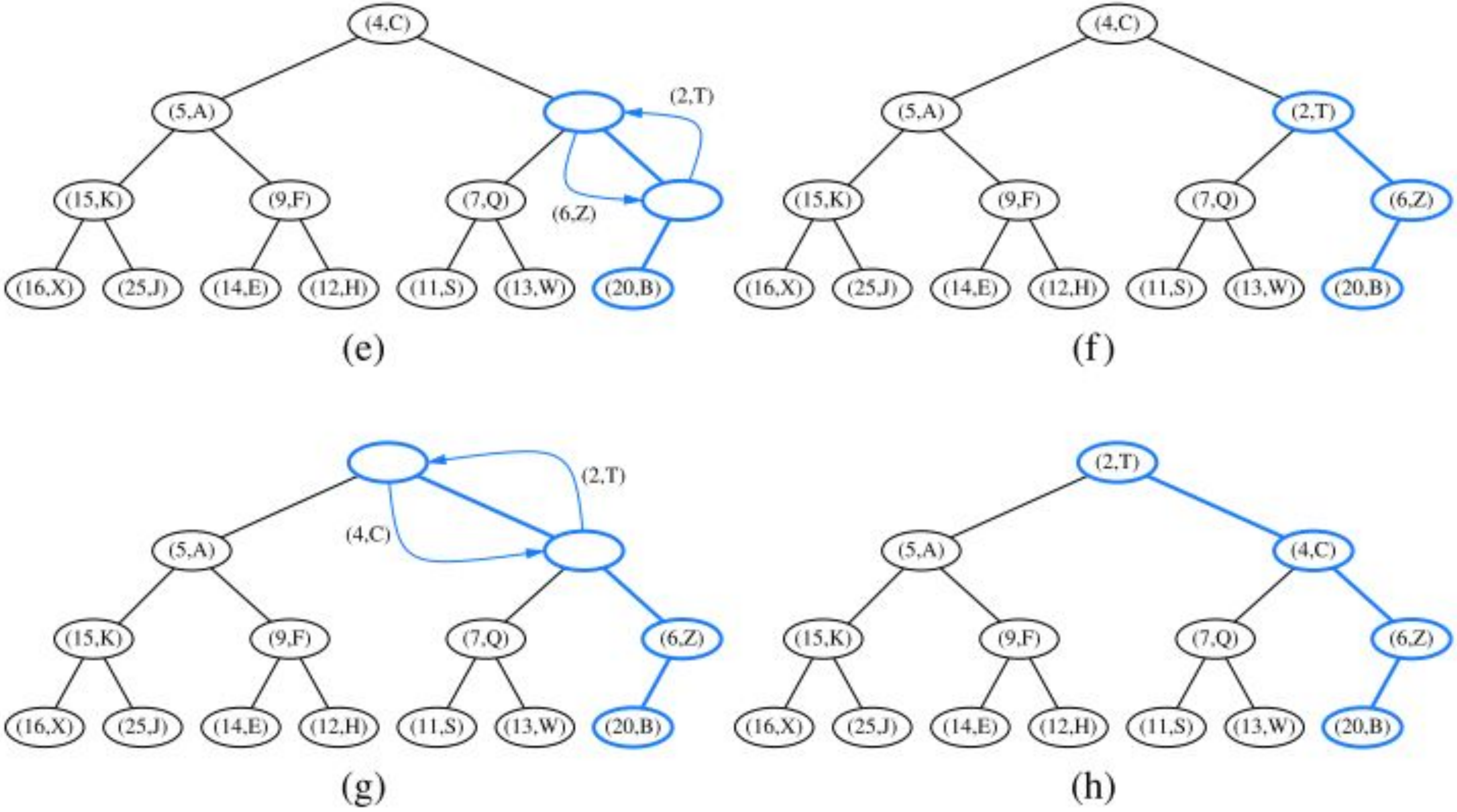
Implementing a Priority Queue with a Heap

- Once we add new node at bottom, the tree ***T*** is complete, but it may *violate the heap-order property*
- To retain the heap-property, position ***p*** needs to be palced appropriately
- We compare the key at position ***p*** to that of ***p's*** parent, which we denote as ***q***
  - If key $k_p \geq k_q$, the heap-order property is satisfied and the algorithm terminates
  - If instead $k_p < k_q$, then we need to restore the heap-order property, which can be locally achieved by swapping the entries stored at positions ***p*** and ***q***
  - This swap causes the new entry to move up one level
  - we repeat the process until no violation of the heap-order property occurs
- The upward movement of the newly inserted entry by means of swaps is conventionally called ***up-heap bubbling***
- A swap either resolves the violation of the heap-order property or propagates it one level up in the heap
- In the *worst case*, upheap bubbling causes the new entry to move all the way up to the root of heap ***T***

Up-Heap Bubbling After an Insertion

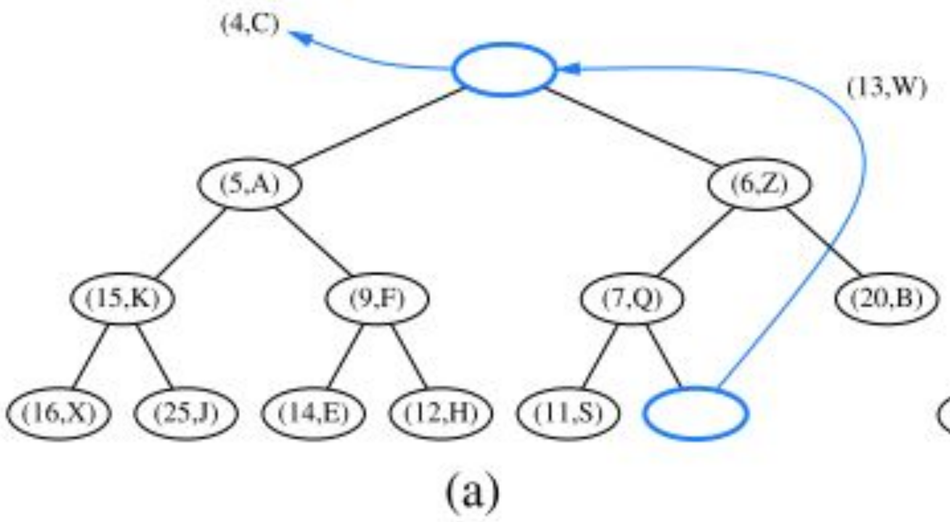


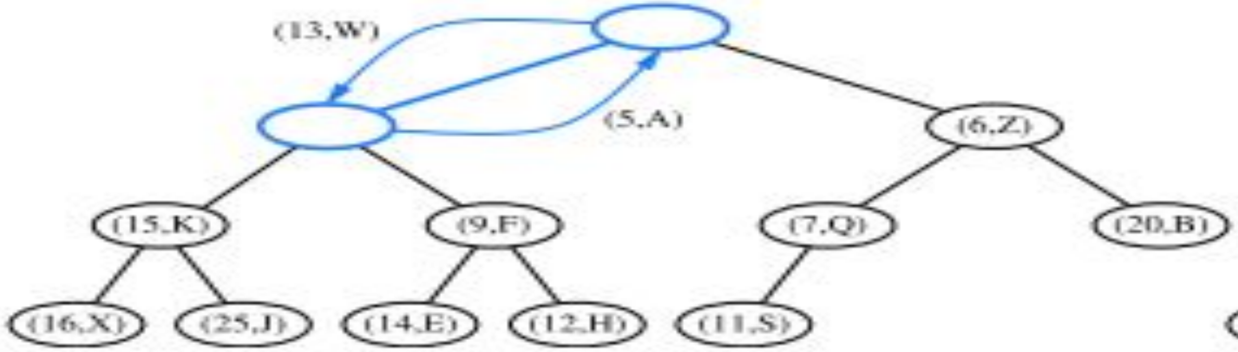
Up-Heap Bubbling After an Insertion



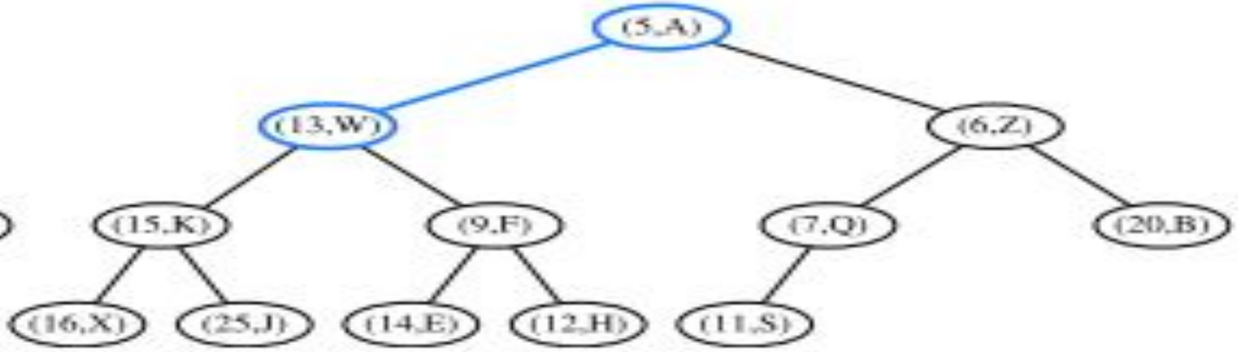
Removing the Entry with Minimal Key

- An entry with the smallest key is stored at the root ***r*** of ***T***
- In method ***removeMin*** of the priority queue ADT, we cannot simply delete node ***r***, because this would leave two disconnected subtrees
- Instead, we ensure that the shape of the heap respects the ***complete binary tree property*** by deleting the *leaf at the last position **p** of **T**,* defined as the rightmost position at the bottom most level of the tree
- To preserve the entry from the last position ***p***, we copy it to the root ***r***
- Then, the node at the last position is removed from the tree

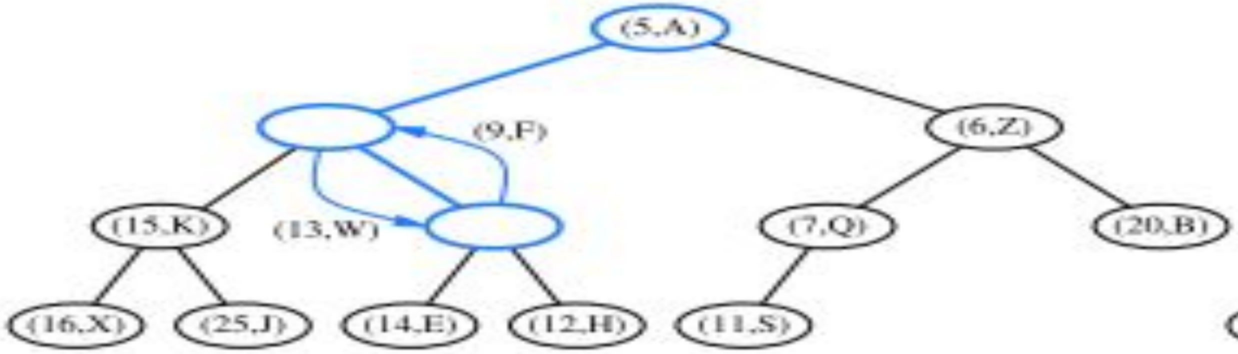



(13,W)
(5,A)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)

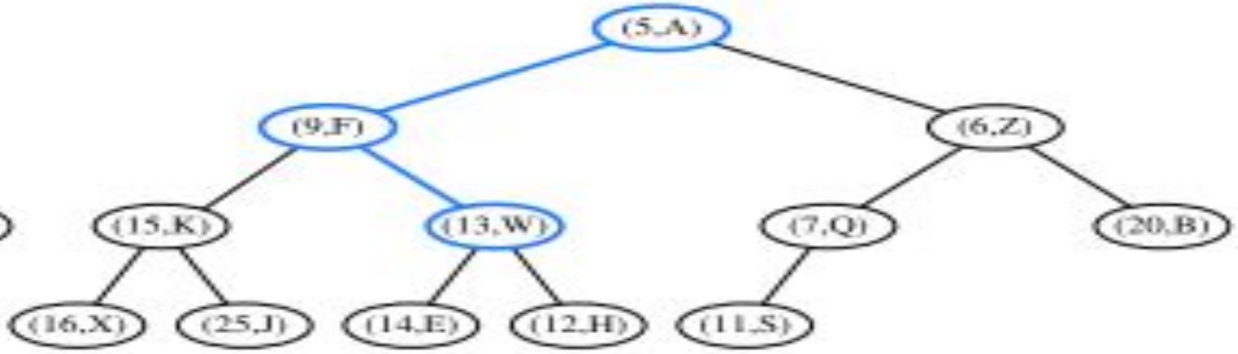
(c)


(5,A)
(13,W)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)

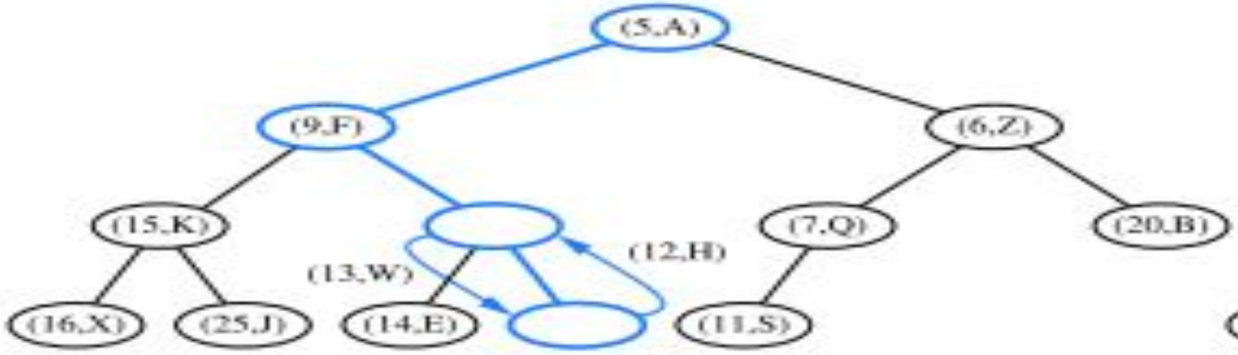
(d)


(5,A)
(9,F)
(6,Z)
(13,W)
(15,K)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)

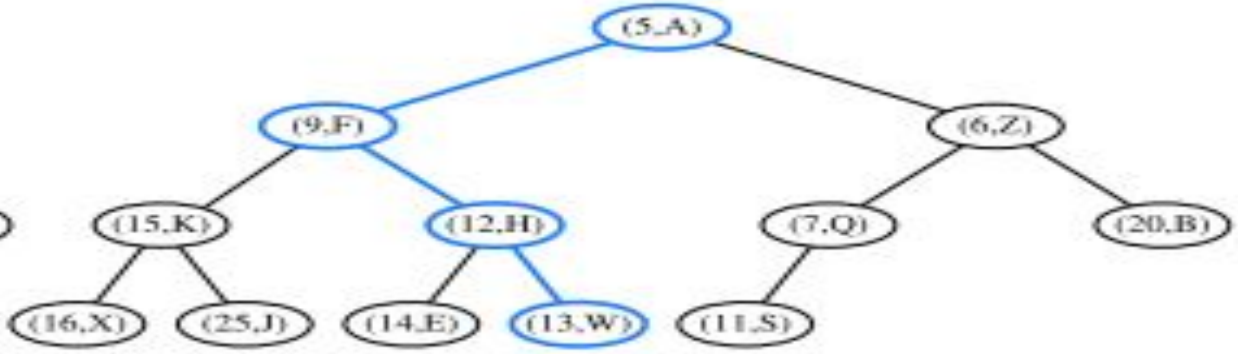
(e)


(5,A)
(9,F)
(6,Z)
(15,K)
(13,W)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)

(f)


(5,A)
(9,F)
(6,Z)
(15,K)
(12,H)
(7,Q)
(20,B)
(13,W)
(16,X)
(25,J)
(14,E)
(11,S)

(g)


(5,A)
(9,F)
(6,Z)
(15,K)
(12,H)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(13,W)
(11,S)

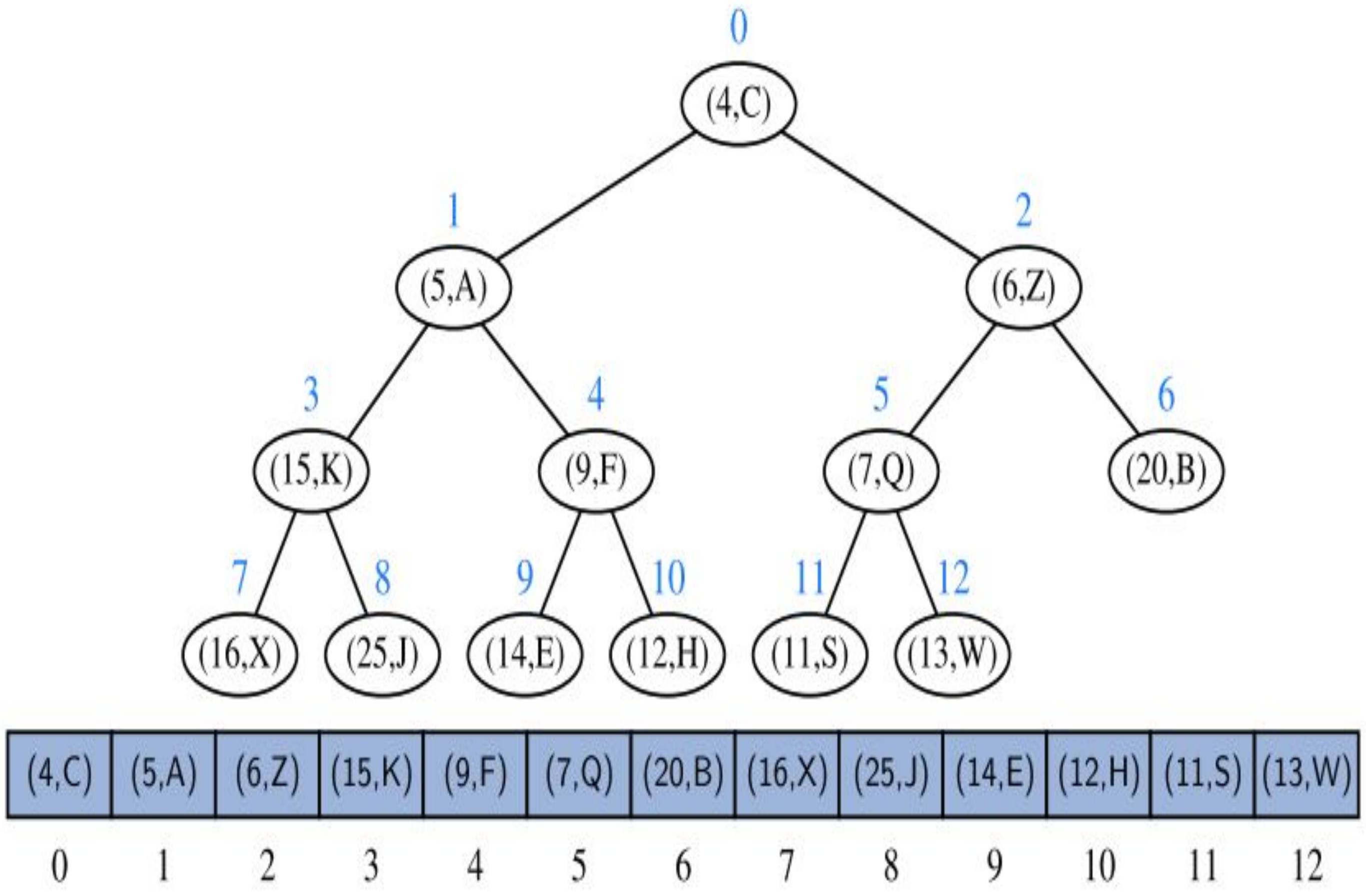
(h)

Array-Based Representation of a Complete Binary Tree

- The array-based representation of a binary tree is especially suitable for a complete binary tree
- The elements of the tree are stored in an array-based list ***A*** such that the element at position ***p*** is stored in ***A*** with index equal to the level number ***f(p)*** of ***p***, defined as follows:
- If ***p*** is the root, then ***f(p)***= 0
- If ***p*** is the left child of position ***q***, then ***f(p)*** = *2* ***f(q)****+1*
- If ***p*** is the right child of position ***q***, then ***f(p)*** = *2* ***f(q)****+2*
- For a tree with of size ***n***, the elements have contiguous indices in the range ***[0, n−1]*** and the last position of is always at index ***n−1***

Array-based representation of a heap

- In the array-based representation of a heap of size **n**, the last position is simply at index **n–1**
- If the size of a priority queue is not known in advance, use of an array-based representation does introduce the need to *dynamically resize the array* on occasion



Java Heap Implementation

- For heap implementation, we prefer to use array-based representation of a tree, maintaining a Java ArrayList of entry composites
- Tree-like terminology of *parent*, *left*, and *right*, the class includes protected utility methods that compute the level numbering of a parent or child of another position
- However, the "positions" in this representation are simply integer indices into the array-list
- Our class has protected utilities ***swap***, ***upheap***, and ***downheap*** for the lowlevel movement of entries within the array-list
- A new entry is added the end of the array-list, and then repositioned as needed with ***upheap***
- To remove the entry with minimal key (which resides at index 0), we move the last entry of the array-list from index n−1 to index 0, and then invoke ***downheap*** to reposition it

An implementation of a priority queue using an array-based heap

```java
/** An implementation of a priority queue using an array-based heap. */
public class HeapPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  /** primary collection of priority queue entries */
  protected ArrayList<Entry<K,V>> heap = new ArrayList<>();
  /** Creates an empty priority queue based on the natural ordering of its keys. */
  public HeapPriorityQueue() { super(); }
  /** Creates an empty priority queue using the given comparator to order keys. */
  public HeapPriorityQueue(Comparator<K> comp) { super(comp); }
  // protected utilities
  protected int parent(int j) { return (j-1) / 2; }        // truncating division
  protected int left(int j) { return 2*j + 1; }
  protected int right(int j) { return 2*j + 2; }
  protected boolean hasLeft(int j) { return left(j) < heap.size(); }
  protected boolean hasRight(int j) { return right(j) < heap.size(); }
```

An implementation of a priority queue using an array-based heap

```
/** Exchanges the entries at indices i and j of the array list. */
protected void swap(int i, int j) {
  Entry<K,V> temp = heap.get(i);
  heap.set(i, heap.get(j));
  heap.set(j, temp);
}
/** Moves the entry at index j higher, if necessary, to restore the heap property. */
protected void upheap(int j) {
  while (j > 0) {            // continue until reaching root (or break statement)
    int p = parent(j);
    if (compare(heap.get(j), heap.get(p)) >= 0) break;    // heap property verified
    swap(j, p);
    j = p;                                 // continue from the parent's location
  }
}
```

An implementation of a priority queue using an array-based heap

```
/** Moves the entry at index j lower, if necessary, to restore the heap property. */
protected void downheap(int j) {
  while (hasLeft(j)) {                    // continue to bottom (or break statement)
    int leftIndex = left(j);
    int smallChildIndex = leftIndex;          // although right may be smaller
    if (hasRight(j)) {
        int rightIndex = right(j);
        if (compare(heap.get(leftIndex), heap.get(rightIndex)) > 0)
          smallChildIndex = rightIndex;       // right child is smaller
    }
    if (compare(heap.get(smallChildIndex), heap.get(j)) >= 0)
      break;                                  // heap property has been restored
    swap(j, smallChildIndex);
    j = smallChildIndex;                      // continue at position of the child
  }
}

// public methods
/** Returns the number of items in the priority queue. */
public int size() { return heap.size(); }
```

An implementation of a priority queue using an array-based heap

```
public Entry<K,V> min() {
  if (heap.isEmpty()) return null;
  return heap.get(0);
}
/** Inserts a key-value pair and returns the entry created. */
public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
  checkKey(key);      // auxiliary key-checking method (could throw exception)
  Entry<K,V> newest = new PQEntry<>(key, value);
  heap.add(newest);                         // add to the end of the list
  upheap(heap.size() − 1);                  // upheap newly added entry
  return newest;
}
/** Removes and returns an entry with minimal key (if any). */
public Entry<K,V> removeMin() {
  if (heap.isEmpty()) return null;
  Entry<K,V> answer = heap.get(0);
  swap(0, heap.size() − 1);                 // put minimum item at the end
  heap.remove(heap.size() − 1);             // and remove it from the list;
  downheap(0);                              // then fix new root
  return answer;
}
}
```

Analysis of a Heap-Based Priority Queue

- The priority queue ADT methods can be performed in ***O(1)*** or in ***O(logn)*** time, where n is the number of entries at the time the method is executed
- The analysis of the running time of the methods is based on the following:
- The heap ***T*** has ***n*** nodes, each storing a reference to a key-value entry
- The height of heap ***T*** is ***O(logn)***, since ***T*** is complete
- The ***min*** operation runs in ***O(1)*** because the root of the tree contains such an element
- Locating the last position of a heap, as required for insert and removeMin, can be performed in ***O(1)*** time for an array-based representation, or O(logn) time for a linked-tree representation
- In the worst case, ***up-heap*** and ***down-heap*** bubbling perform a number of swaps equal to the height of ***T***