Data Structures and Algorithms

Chapter 9

Learning Objectives

- Design of the Priority Queue ADT
- Implementation of the Priority Queue using a Heap with an underlying Array or ArrayList
- Algorithms and runtime of modifier methods for Heap-based Priority Queue
- Sorting using a Priority Queue

Priority Queues

- Each element in a queue is associated with a *key*.
- When an element is removed, an element with a *minimal* (or *maximal*) *key* is removed.
- Usually keys are numbers.
- Objects can be used as keys as far as there is a total ordering among those objects.

Priority Queues
ADT

- insert(*k*, *v*): Create an entry with key *k* and value *v* in the priority queue.
- min( ): Returns (but does not remove) an entry (*k*, *v*) with the minimum key. Returns null if the priority queue is empty.
- removeMin( ): Removes and returns an entry (*k*, *v*) with the minimum key. Returns null if the priority queue is empty.
- size( ): Returns the number of entries in the priority queue.
- isEmpty( ): Returns true if the priority queue is empty. Returns false, otherwise.

Priority Queues
ADT

- Illustration

Method	Return Value	Priority Queue Contents
insert(17, A)		{(17, A)}
insert(4, P)		{(4, P) , (17, A)}
insert(15, X)		{(4, P) , (15, X), (17, A)}
size( )	3	{(4, P) , (15, X), (17, A)}
isEmpty( )	false	{(4, P) , (15, X), (17, A)}
min( )	(4, P)	{(4, P) , (15, X), (17, A)}
removeMin( )	(4, P)	{(15, X), (17, A)}
removeMin( )	(15, X)	{(17, A)}
removeMin( )	(17, A)	{ }
removeMin( )	null	{ }
size( )	0	{ }
isEmpty( )	true	{ }

Priority Queues
Implementation

- An element in a priority queue has *key* and *value*.
- The *Entry* interface is used to store a key-value pair.

```
public interface Entry<K,V> {
  K getKey();
  V getValue();
}
```

Priority Queues
Implementation

- *PriorityQueue* interface

```
public interface PriorityQueue<K,V> {
    int size();
    boolean isEmpty();
    Entry<K,V> insert(K key, V value) throws
                                    IllegalArgumentException;
    Entry<K,V> min();
    Entry<K,V> removeMin();
}
```

Priority Queues
Implementation

- Keys must have *total ordering*.
- Total ordering means there is a linear ordering among all keys.
- Total ordering of a comparison rule, ρ, satisfies the following properties:
  - Comparability property: Either k_1 ρ k_2 or k_2 ρ k_1.
  - Antisymmetric property: If k_1 ρ k_2 and k_2 ρ k_1, then $k_1 = k_2$.
  - Transitive property: If k_1 ρ k_2 and k_2 ρ k_3, then k_1 ρ k_3.
- If keys have total ordering, *minimal key* is well defined
- key_{min} is a key such that: key_{min} ρ k, for all k

Priority Queues
Implementation

- Replacing ρ with ≤:
  - Comparability property: $k_1 \le k_2$ or $k_2 \le k_1$.
  - Antisymmetric property: If $k_1 \le k_2$ and $k_2 \le k_1$, then $k_1 = k_2$.
  - Transitive property: If $k_1 \le k_2$ and $k_2 \le k_3$, then $k_1 \le k_3$.
- If keys have total ordering, *minimal key* is well defined
- key_{min} is a key such that: $key_{min} \le k$, for all k

Priority Queues
Implementation

- Two ways to compare objects in Java
  - *compareTo* and *compare*
- *compareTo* is defined in *java.util.Comparable* interface.
- A class must override and implement the *compareTo* method.
- *Ordering* defined in the *compareTo* method is called *natural ordering*.
- Usage: *a.compareTo*(*b*) returns
  - a negative number, if a < b
  - zero, if a = b
  - a positive number, if a > b
- Many Java classes implement *Comparable* interface.

Priority Queues
Implementation

- *compare* is defined in *java.util.Comparator* interface.
- Use this to compare not by natural ordering
- Need to write a separate customized comparator
- Example: To compare strings by length (natural ordering is lexicographic ordering).
- First, write a customized comparator method

```
public class StringLengthComparator implements Comparator<String> {
   public int compare(String a, String b){
               if (a.length() < b.length()) return -1;
               else if (a.length() == b.length()) return 0;
               else return 1;
   }
}
```

Priority Queues
Implementation

- Then, use it as follows:

```
public class ComparatorTest {
    public static void main(String[] args) {
            StringLengthComparator c = new StringLengthComparator();
            String s1 = "tiger";
            String s2 = "sugar";
            String s3 = "coffee";
            String s4 = "cat";
            System.out.println("Compare s1 and s2: " + c.compare(s1, s2)); // 0
            System.out.println("Compare s1 and s3: " + c.compare(s1, s3)); // -1
            System.out.println("Compare s1 and s4: " + c.compare(s1, s4)); // 1
    }
}
```

Priority Queues

AbstractPriorityQueue Base Class

- Provides common features for different concrete implementations.
- An entry in a queue is implemented as *PQEntry*:

```
protected static class PQEntry<K,V> implements Entry<K,V> {
    private K k;  // key
    private V v;  // value
    public PQEntry(K key, V value) {
        k = key;
        v = value;
    }
    public K getKey() { return k; }
    public V getValue() { return v; }
    protected void setKey(K key) { k = key; }
protected void setValue(V value) { v = value; }
}
```

Priority Queues
Implementing Using a Heap

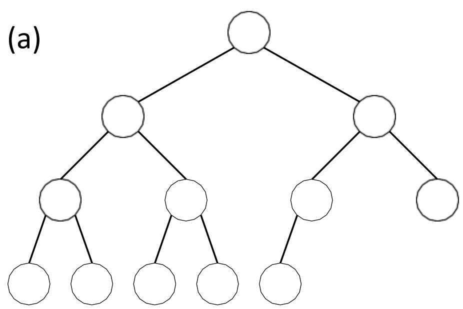
- Implementation with an unsorted list
- Implementation with a sorted list

- We will focus on implementation with *heap*.

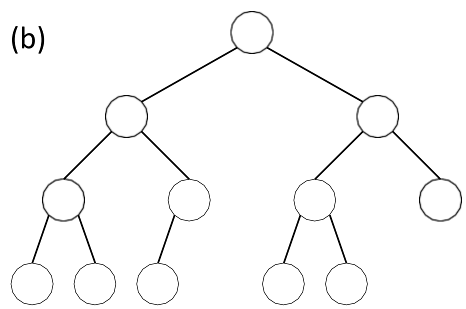
- *Heap* is a binary tree with the following properties:
  - *Heap-order property*: In a heap T, for every position p, except the root, the key stored at p is greater than or equal to the key stored at p's parent. (*minimum-oriented* heap)
  - *Complete binary tree property*: A heap is a complete binary tree.

Priority Queues
Implementing Using a Heap

- Complete binary tree
  - Levels 0, 1, . . ., h – 1 of T have the maximal number of nodes (in other words, level i has 2^i nodes, where $0 \leq i \leq h - 1$), and
  - Nodes at level h are in the leftmost possible positions at that level.



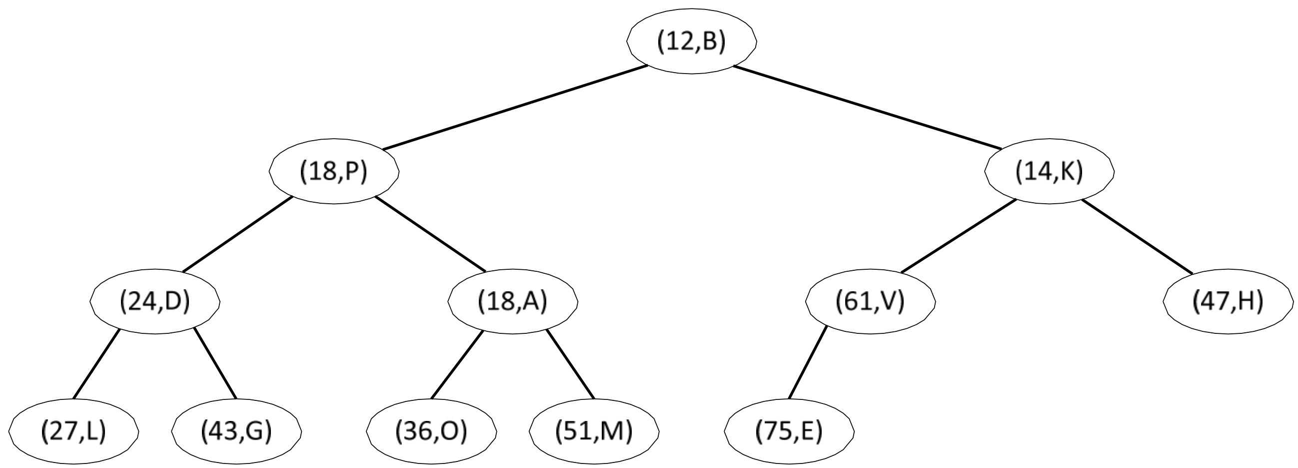
yes



no

Priority Queues
Implementing Using a Heap

- Priority queue implemented using a heap example:



- Height of a heap with *n* entries is $h = \lfloor \log n \rfloor$

Priority Queues
Implementing Using a Heap

- Adding an entry to a heap
  - Step 1: Add new entry at the “end” of the heap
  - Step 2: Reorganize the heap (because adding new entry may violate the heap-order property)

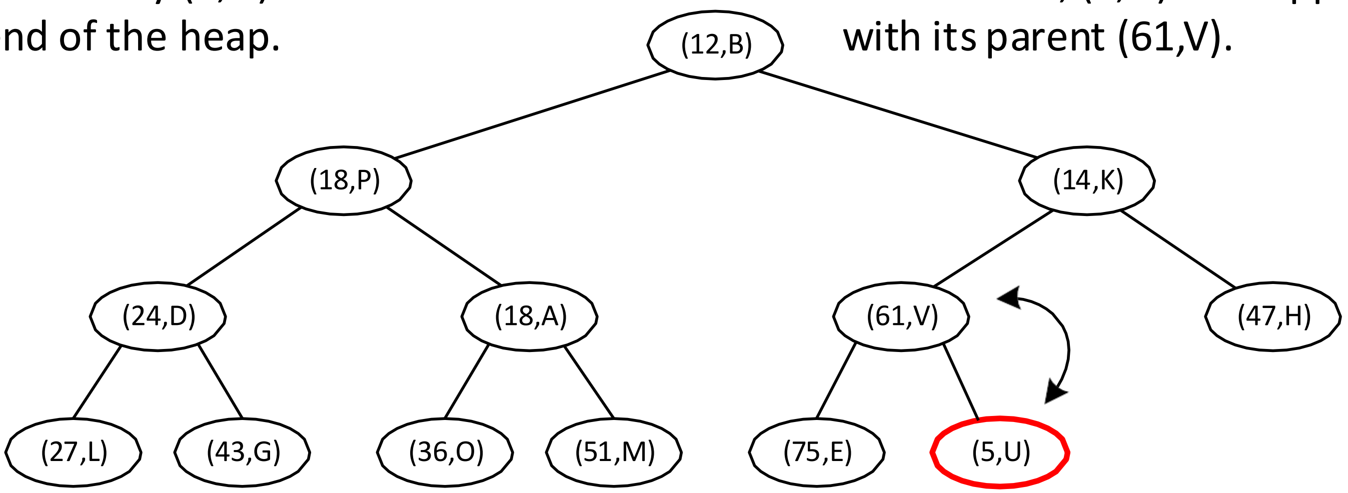
- Reorganization is done by *up-heap bubbling*.

Priority Queues
Implementing Using a Heap

- Illustration

New entry (5,U) is added to the end of the heap.

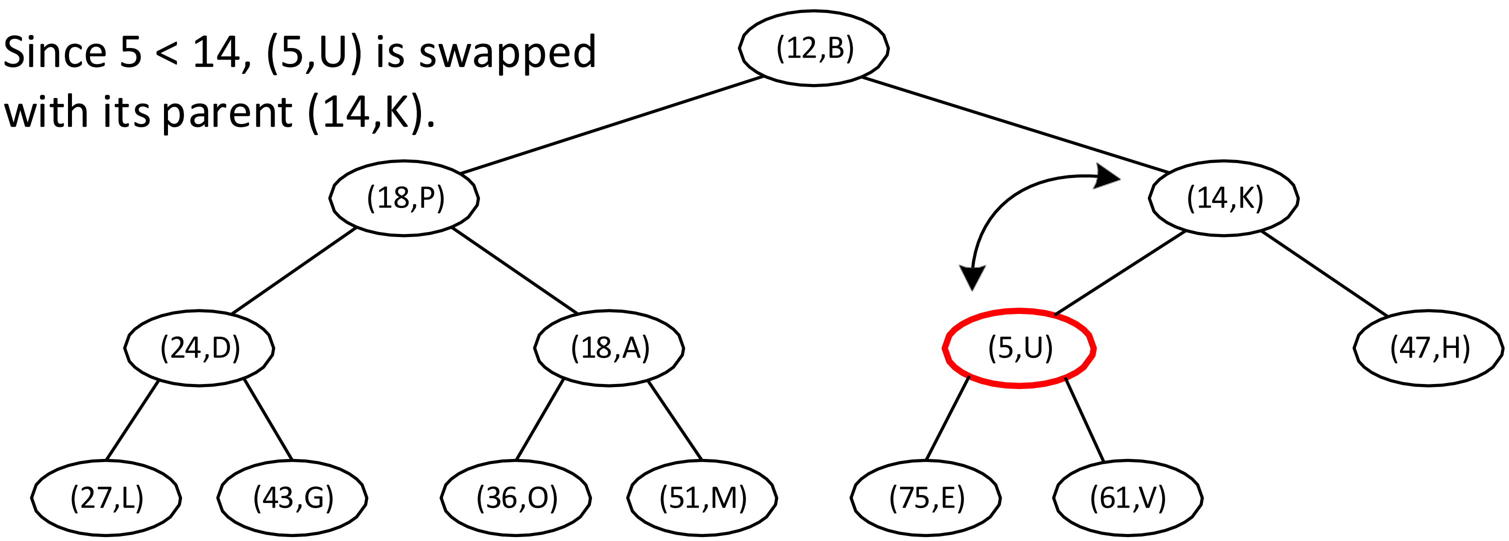
Since 5 < 61, (5,U) is swapped with its parent (61,V).



Priority Queues
Implementing Using a Heap

- Illustration

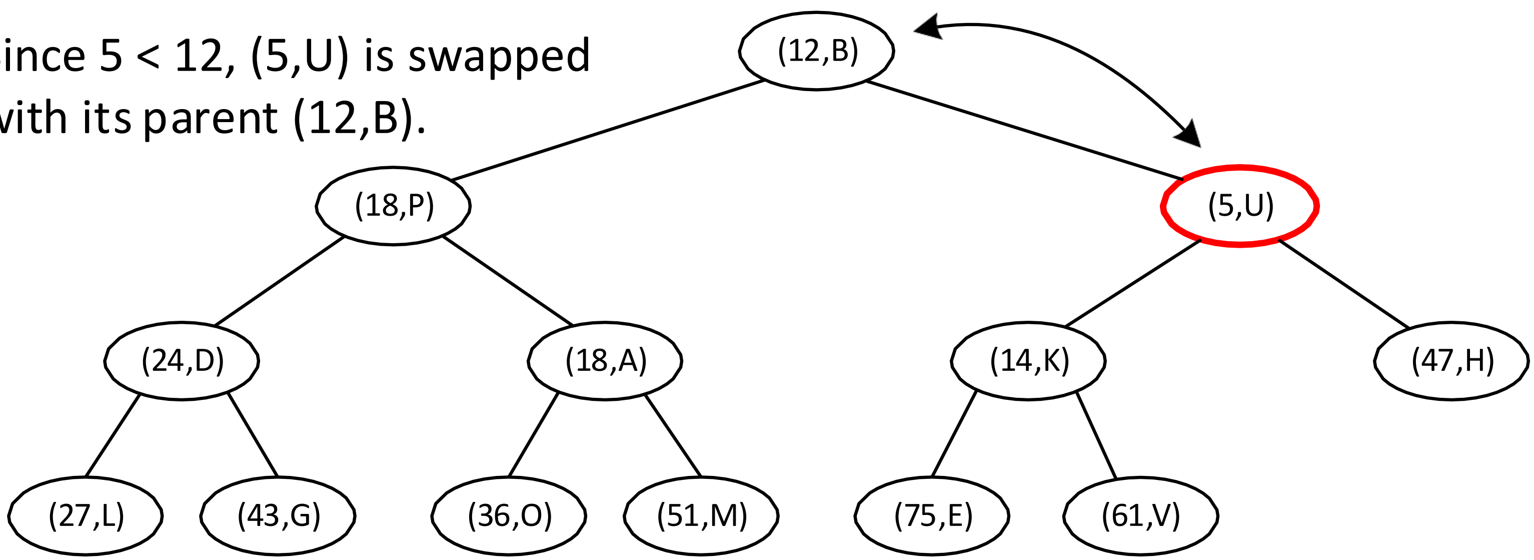
Since 5 < 14, (5,U) is swapped with its parent (14,K).



Priority Queues
Implementing Using a Heap

- Illustration

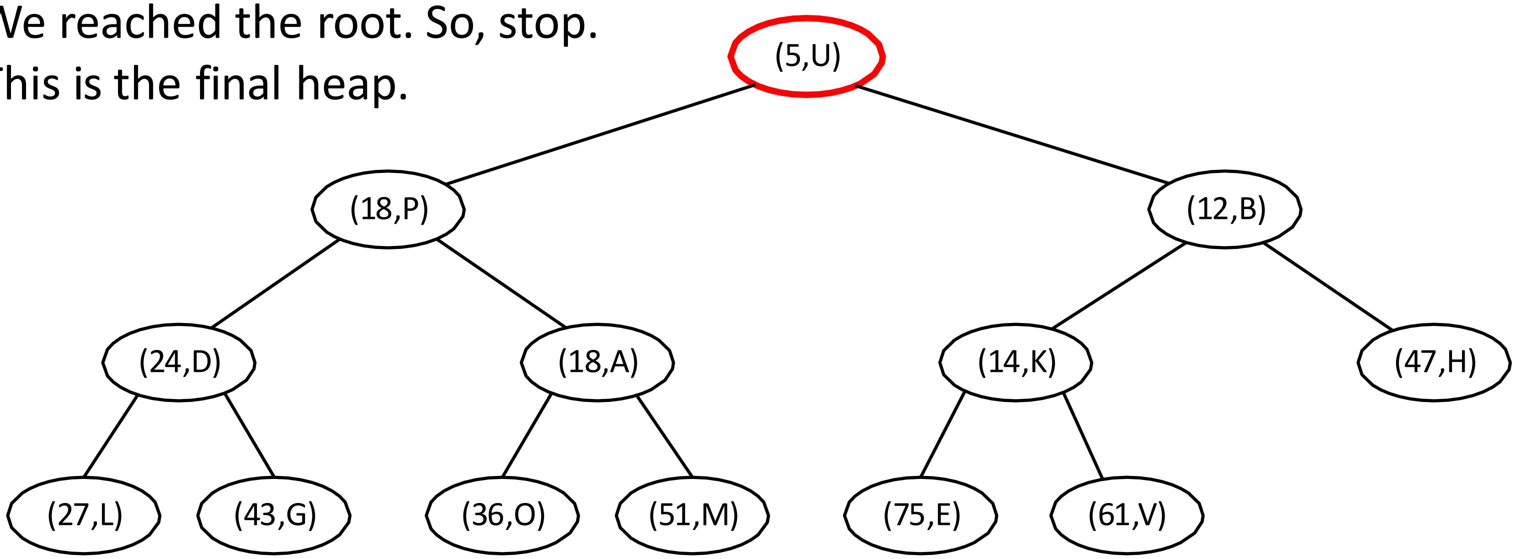
Since 5 < 12, (5,U) is swapped with its parent (12,B).



Priority Queues
Implementing Using a Heap

- Illustration

We reached the root. So, stop.
This is the final heap.



Priority Queues
Implementing Using a Heap

- Removing the entry with minimal key
  - Step1: Remove the root
  - Step 2: Last node moves up to the root and performs *down-heap bubbling*.

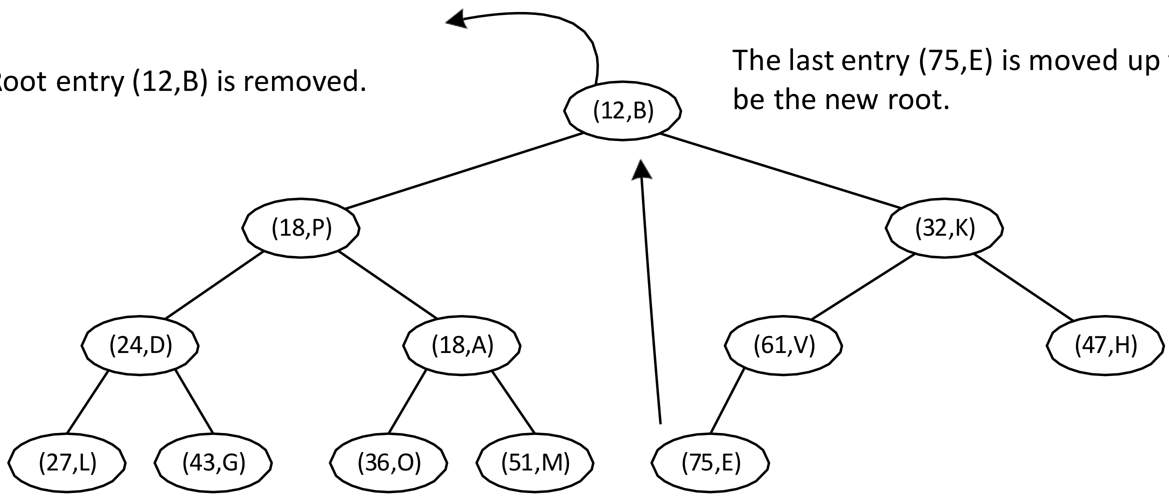
- Down-heap bubbling is opposite of up-heap bubbling.
- Each time we compare the current node with both left and right child, and replace it with the minimum child

Priority Queues
Implementing Using a Heap

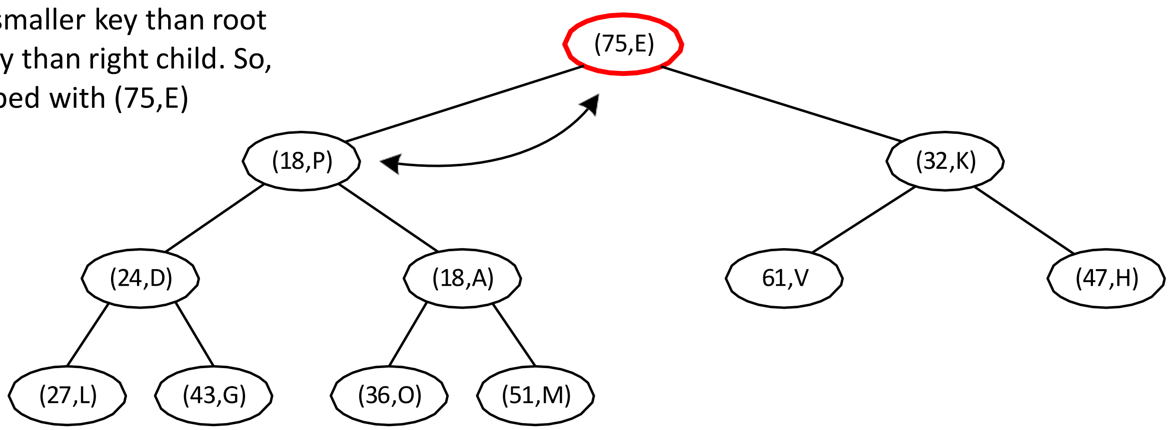
- Illustration

Root entry (12,B) is removed.

The last entry (75,E) is moved up to be the new root.



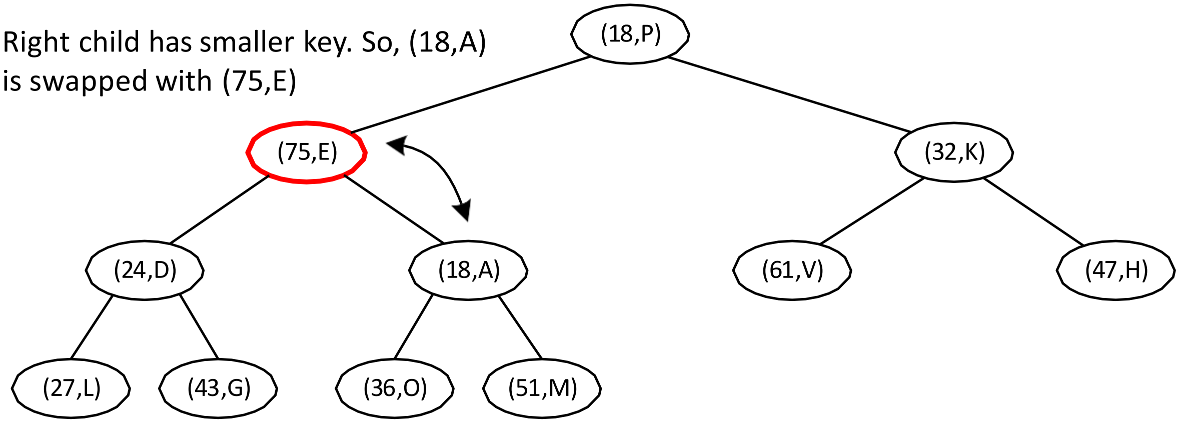
Left child has smaller key than root and smaller key than right child. So, (18,P) is swapped with (75,E)



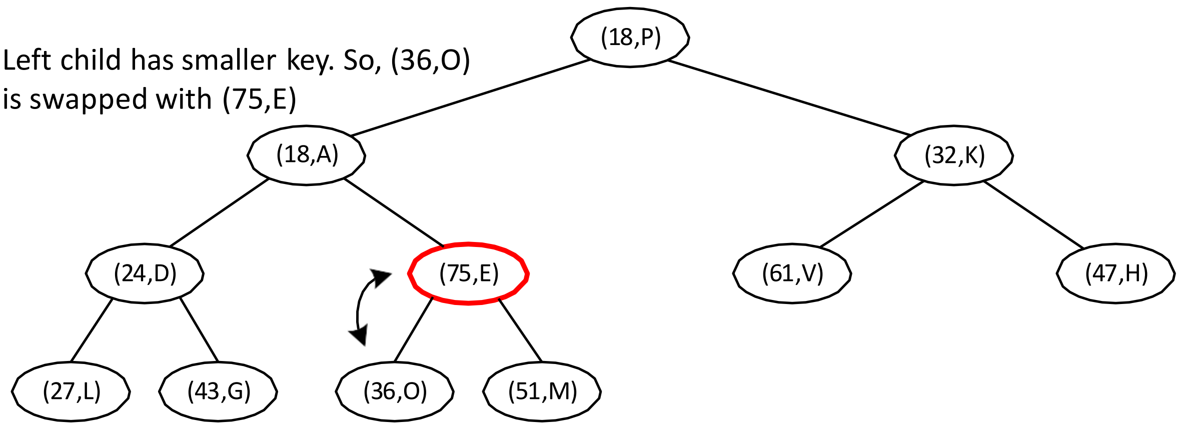
Priority Queues
Implementing Using a Heap

- Illustration

Right child has smaller key. So, (18,A) is swapped with (75,E)



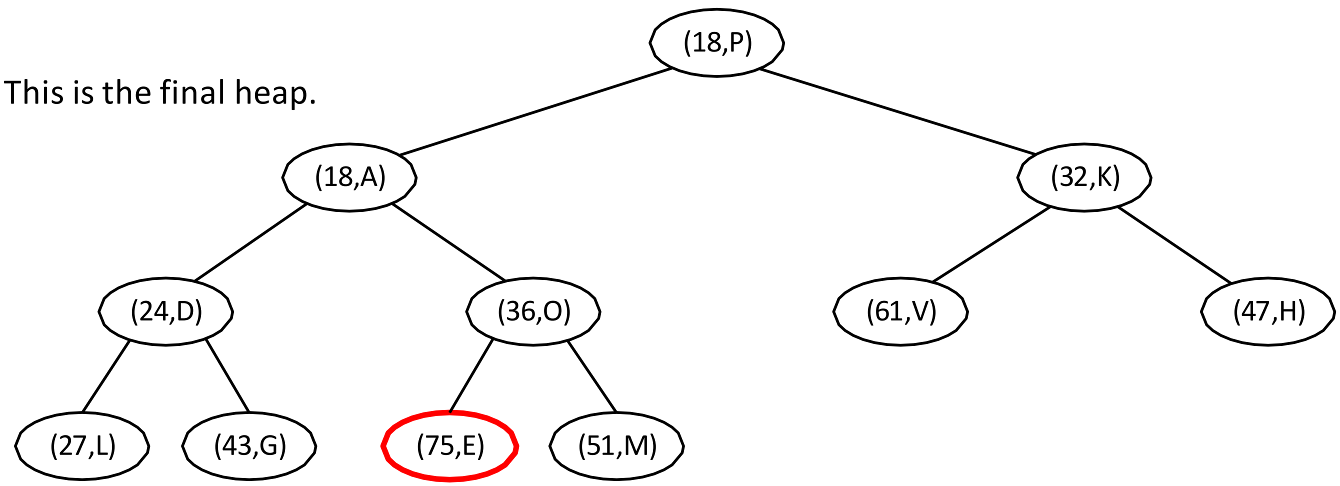
Left child has smaller key. So, (36,O) is swapped with (75,E)



Priority Queues
Implementing Using a Heap

- Illustration

This is the final heap.



Priority Queues
Array-Based Heap

- The level number of a position p, $f(p)$, is defined as follows:

  – If p is the root, $f(p) = 0$
  – If p is the left child of position q, $f(p) = 2*f(q) + 1$
  – If p is the right child of position q, $f(p) = 2*f(q) + 2$

- The level number is used as the index in an array where the entry with position p is stored.

Priority Queues

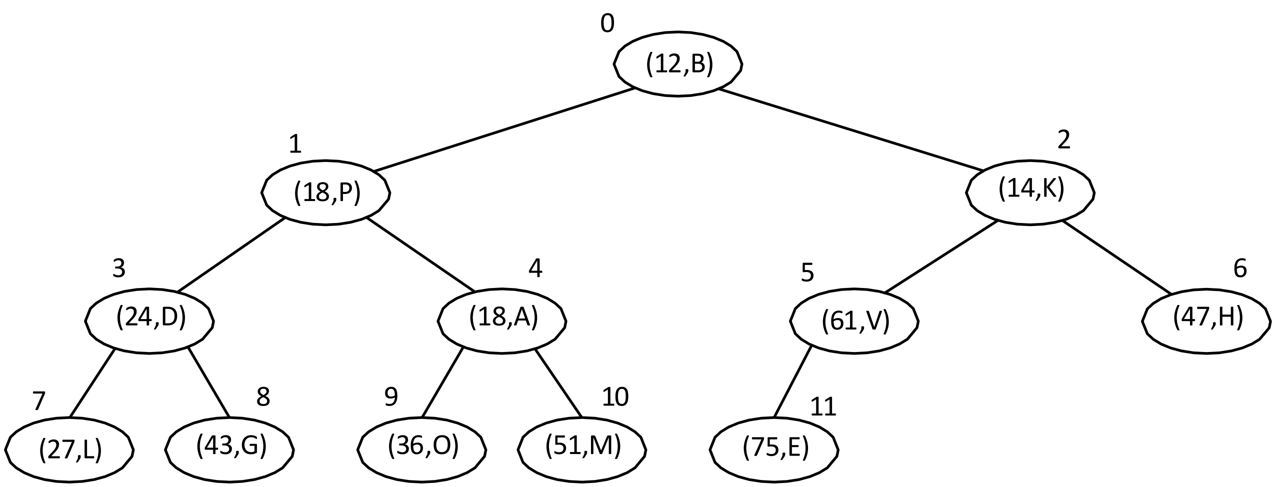
Array-Based Heap

- Then, the entry at position *p* is stored in *A*[*f*(*p*)].

- Index of the root node is 0.
- Index of left child of *p* = 2**f*(*p*) + 1
- Index of right child of *p* = 2**f*(*p*) + 2
- Index of parent of *p* = $\lfloor (f(p) - 1)/2 \rfloor$

Priority Queues
Array-Based Heap

- Example



(12,B)	(18,P)	(14,K)	(24,D)	(18,A)	(61,V)	(47,H)	(27,L)	(43,G)	(36,O)	(51,M)	(75,E)
0	1	2	3	4	5	6	7	8	9	10	11

Priority Queues
Array-Based Heap

- *HeapPriorityQueue* class implements a priority queue using a heap.
- A heap is implemented using *ArrayList*.
- Will briefly discuss *upheap*, *downheap*, *insert*, and *removeMin* methods.

- *HeapPriorityQueue.java* code (Ch. 9.3 Page 377)

Priority Queues
Analysis of Heap-Based Priority Queue

- insertion:
  - *upheap* method takes $O(\log n)$
  - So, insertion takes $O(\log n)$
- removeMin:
  - *downheap* method takes $O(\log n)$
  - So, removeMin takes $O(\log n)$

Method	Running Time
size, isEmpty	$O(1)$
min	$O(1)$
insert	$O(\log n)$
removeMin	$O(\log n)$

Priority Queues
Bottom-up Heap Construction

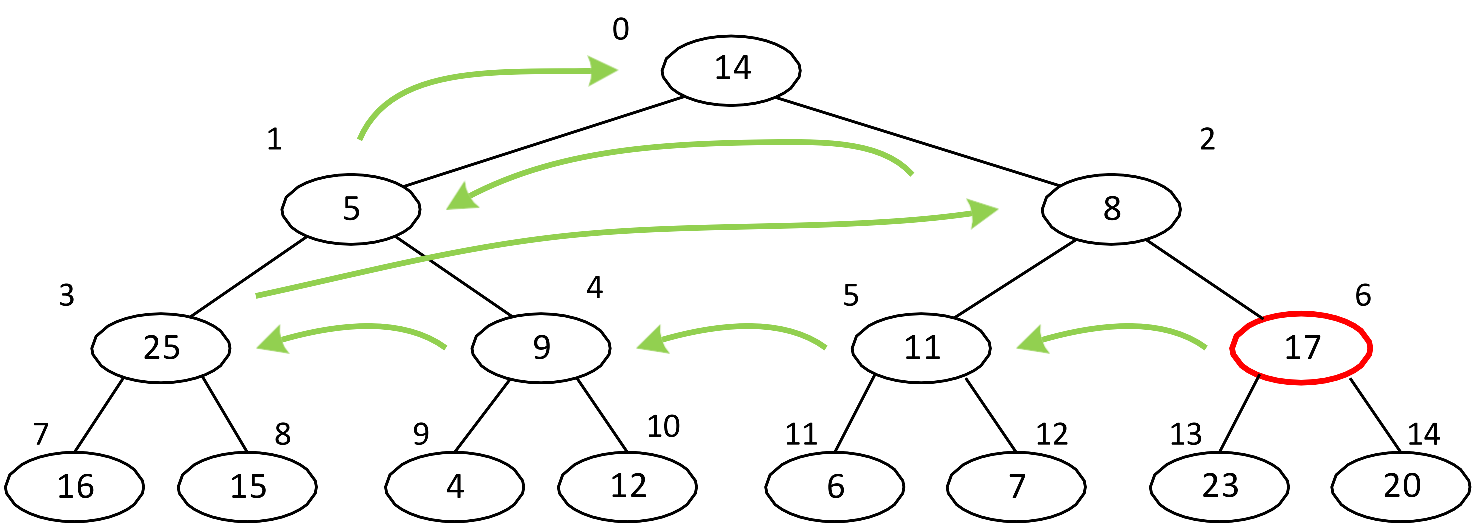
- Given n elements, we can build a heap with n successive insertions => takes $O(n \log n)$ time.

- $O(n)$ time algorithm
  - Begin at the parent of the last node, move backward to the root.
  - At each node, perform down-heap bubbling.

Priority Queues
Bottom-up Heap Construction

- Illustration

14	5	8	25	9	11	17	16	15	4	12	6	7	23	20
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14



Priority Queues
Bottom-up Heap Construction

- Java implementation

```
public HeapPriorityQueue(K[ ] keys, V[ ] values) {
  super();
  for (int j=0; j < Math.min(keys.length, values.length); j++)
      heap.add(new PQEntry<>(keys[j], values[j]));
  heapify();
}

protected void heapify() {
  int startIndex = parent(size()-1);    // start at PARENT of last entry
  for (int j=startIndex; j >= 0; j--)   // loop until processing the root
       downheap(j);
}
```

How is this heap construction O(n)?

- Runtime is controlled by the down-heap bubbling + O(n) steps to arbitrarily place nodes in the heap
- Down-heap takes total O(n) time
- Need to compute the total # of steps the nodes have to take to reach their destination
- Proportional to path length if each node follows the path: right child, left child, left, left, …, leaf
- Total of these paths is O(n) as no edges belong to multiple paths and a binary tree has n edges

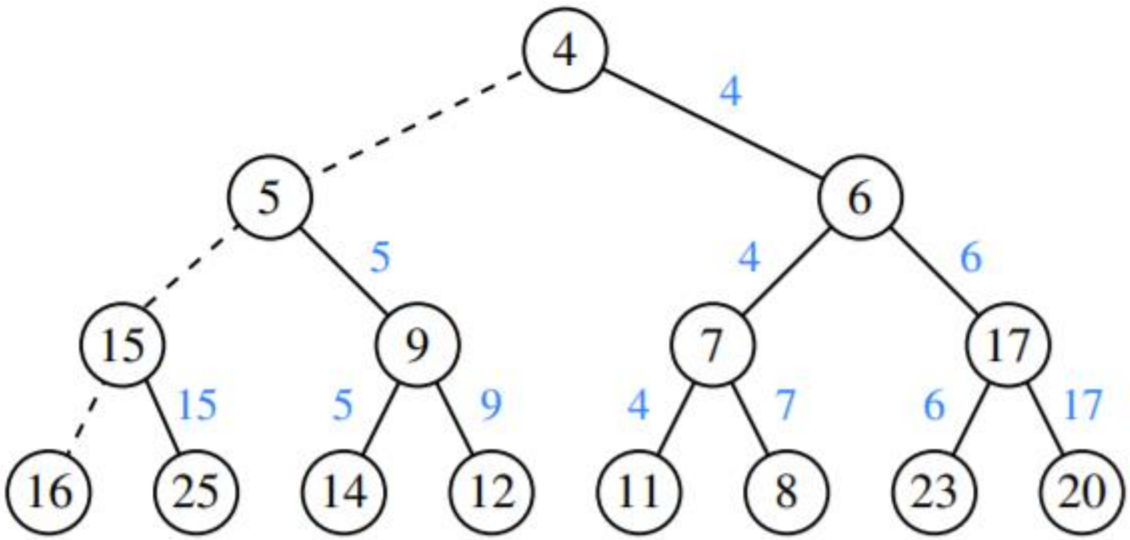


Figure 9.6: Visual justification of the linear running time of bottom-up heap construction. Each edge e is labeled with a node v for which π_v contains e (if any).

Priority Queues
Java's Priority Queue

- *java.util.PriorityQueue*
- An entry is a single element.
- Some operations in Java's *PriorityQueue*
  - add(E e): Inserts the specified element *e* to the priority queue.
  - isEmpty( ): Returns true if the priority queue contains no element.
  - peek( ): Retrieves, but does not remove, a minimal element from the priority queue.
  - remove( ): Removes a minimal element from the priority queue.
  - size( ): Returns the number of elements in the priority queue.

Priority Queues
Heap-Sort

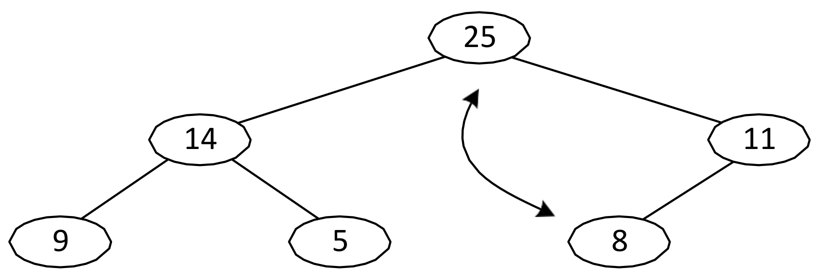
- Uses array-based heap data structure.
- In-place sorting: no additional storage is used.
- Uses a *maximum-oriented* heap.
- *maximum-oriented* heap: In a heap *T*, for every position *p*, except the root, the key stored at *p* is *smaller* than or equal to the key stored at *p*'s parent.
- Sorting steps:
  1. Given *n* elements are inserted into a maximum-oriented heap.
  2. Repeat the following until only one node is left in the heap: Root is swapped with the last node, heap size is decremented, perform down-heap bubbling.

Priority Queues
Sorting with Priority Queue

- Illustration

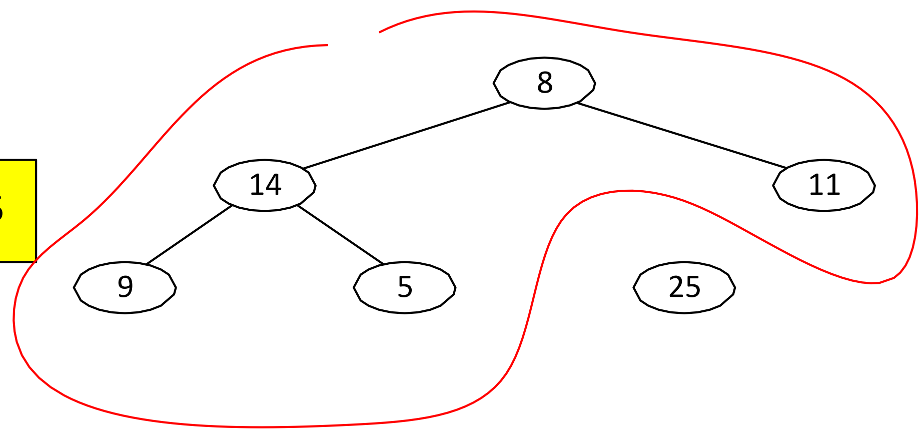
The maximum-oriented heap after the first step.
The root node and the last node is swapped.
Heap size is decremented.

25	14	11	9	5	8



Down-heap bubbling is applied on the root.

8	14	11	9	5	25

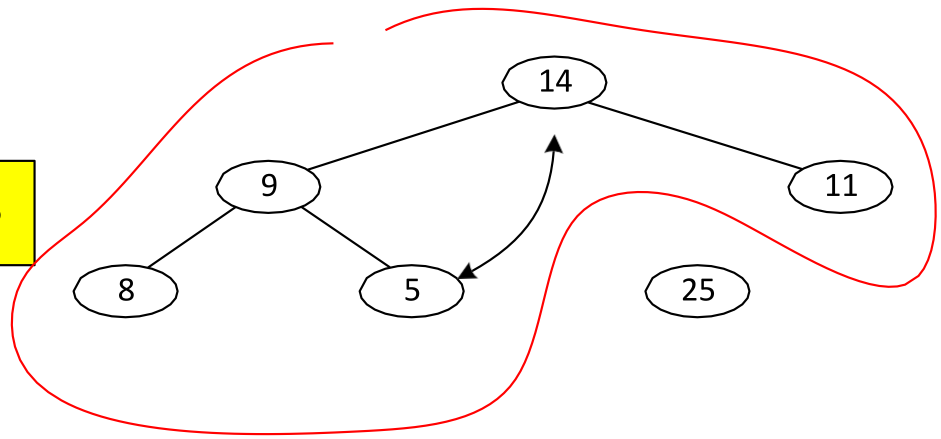


Priority Queues
Sorting with Priority Queue

- Illustration

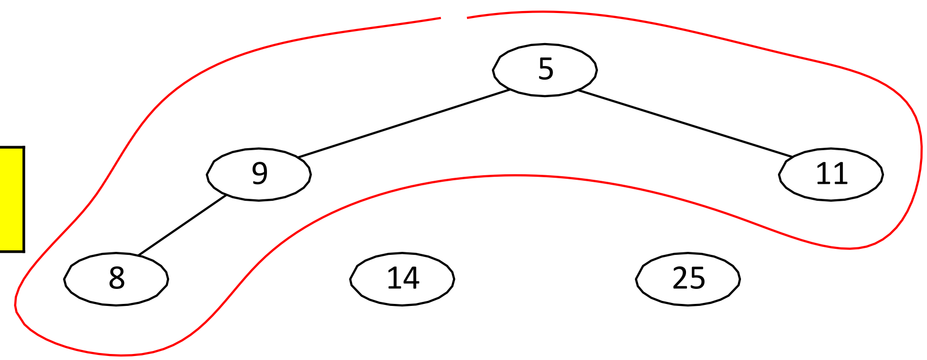
The root node is swapped with the last node. Heap size is decremented.

14	9	11	8	5	25



Down-heap bubbling is applied on the root.

5	9	11	8	14	25

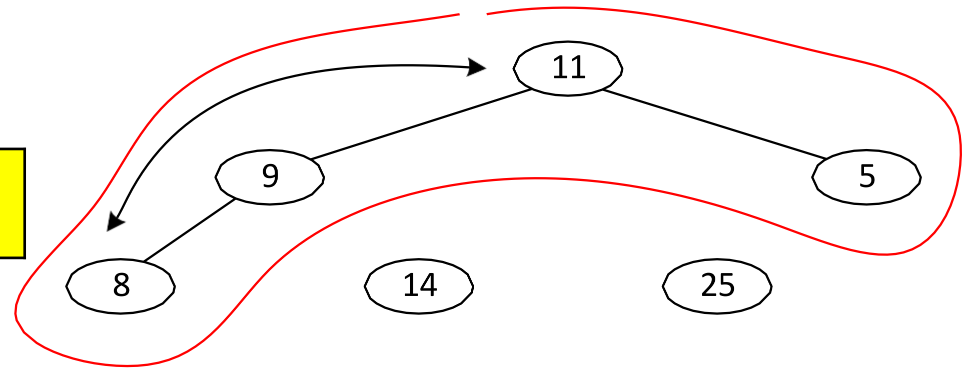


Priority Queues
Sorting with Priority Queue

- Illustration

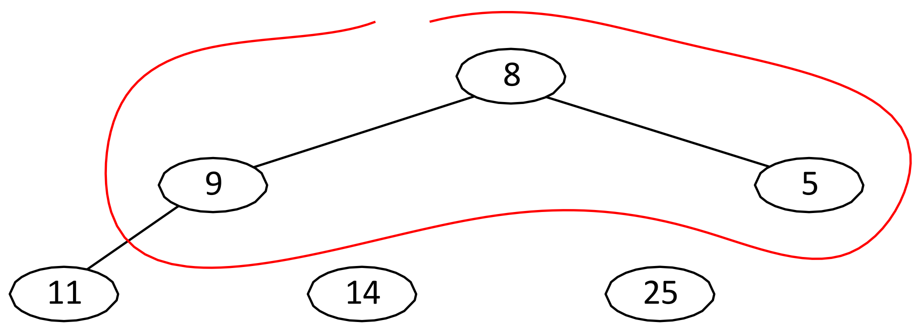
The root node is swapped with the last node. Heap size is decremented.

11	9	5	8	14	25



Down-heap bubbling is applied on the root.

8	9	5	11	14	25

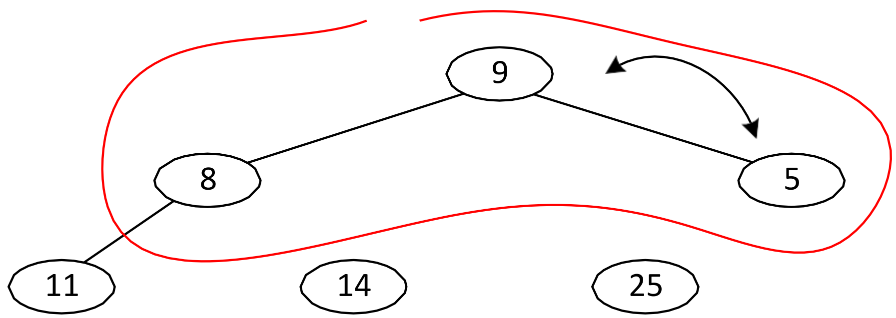


Priority Queues
Sorting with Priority Queue

- Illustration

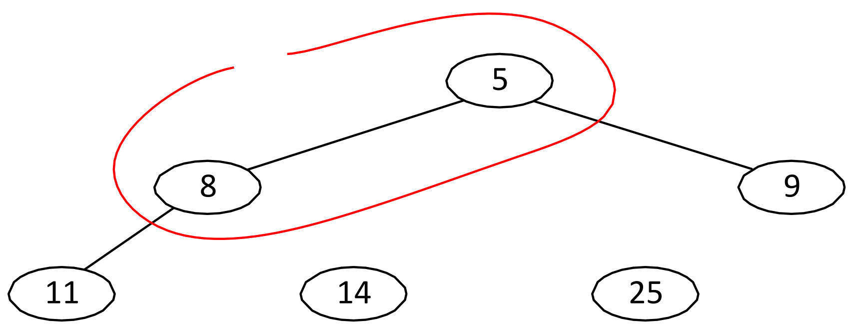
The root node is swapped with the last node. Heap size is decremented.

9	8	5	11	14	25



Down-heap bubbling is applied on the root.

5	8	9	11	14	25

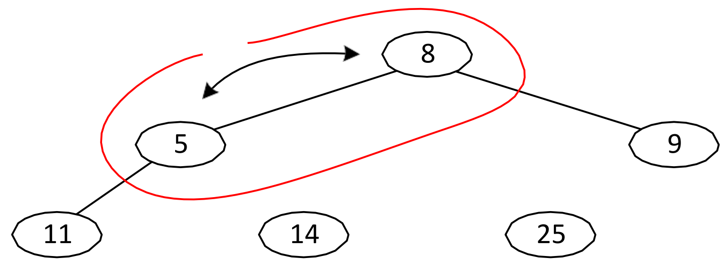


Priority Queues
Sorting with Priority Queue

- Illustration

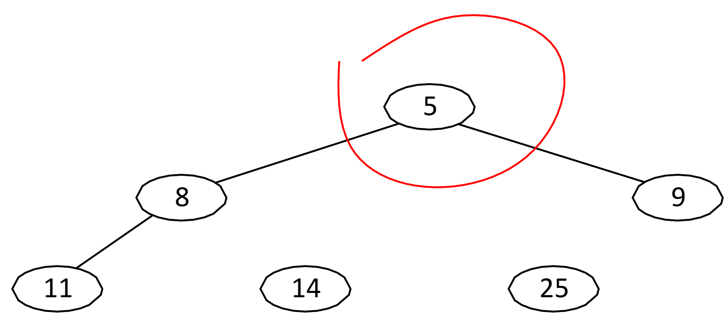
The root node is swapped with the last node. Heap size is decremented.

8	5	9	11	14	25



At this time the array is sorted.

5	8	9	11	14	25



Priority Queues

Adaptable Priority Queue

- Can remove arbitrary entry (not just the root).
- Can replace the key of an entry.
- Can replace the value of an entry.

References

- M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, “Data Structures and Algorithms in Java,” Sixth Edition, Wiley, 2014.