Algorithms

(Priority Queues)

Pramod Ganapathi
Department of Computer Science
State University of New York at Stony Brook

January 15, 2021

Contents

Priority queue

- A priority queue is a tree-based data structure consisting of key-value pairs.
- A priority queue uses the whatever in, priority-out principle.
- A priority queue has two major operations: insert and delete-min.

Applications of priority queues

- Flight queue with customer priority
- Call center queue with customer priority
- Technical support queue with customer priority
- Vaccination queue with citizen priority
- All applications of sorting

Priority queue ADT

Method	Functionality
`insert(k, v)`	Creates an entry with key k and value v in the priority queue.
`min()`	Returns (but does not remove) a priority queue entry (k,v) having minimal key; returns null if the priority queue is empty.
`removeMin()`	Removes and returns an entry (k,v) having minimal key from the priority queue; returns null if the priority queue is empty.
`size()`	Returns the number of entries in the priority queue.
`isEmpty()`	Returns a boolean indicating whether the priority queue is empty.

Operations on a priority queue contents

Method	Return value	Priority queue contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min()	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin()	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin()	(5,A)	{ (7,D), (9,C) }
removeMin()	(7,D)	{ (9,C) }
removeMin()	(9,C)	{ }
removeMin()	null	{ }
isEmpty()	true	{ }

Priority queue complexity

Method	Unsorted list	Sorted list	Array heap	Linked-tree heap
size	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
isEmpty	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
min	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
insert	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)^*$	$\mathcal{O}(\log n)$
removeMin	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)^*$	$\mathcal{O}(\log n)$

* $=$ amortized complexity

Priority queue using an unsorted list

Method	Unsorted list
size	$\mathcal{O}(1)$
isEmpty	$\mathcal{O}(1)$
insert	$\mathcal{O}(1)$
min	$\mathcal{O}(n)$
removeMin	$\mathcal{O}(n)$

- We use a PositionalList which in turn uses DLL.
- insert method inserts a key-value entry at the end of the list in $\mathcal{O}(1)$ time.
- min or removeMin requires scanning the entire list in $\mathcal{O}(n)$ time.

Priority queue using a sorted list

Method	Sorted list
size	$\mathcal{O}(1)$
isEmpty	$\mathcal{O}(1)$
insert	$\mathcal{O}(n)$
min	$\mathcal{O}(1)$
removeMin	$\mathcal{O}(1)$

- We use a PositionalList which in turn uses DLL but sorted by nondecreasing keys.
- min or removeMin requires the retrieval or removal of the first element in $\mathcal{O}(1)$ time.
- insert method inserts a key-value entry at the appropriate position after scanning the list in $\mathcal{O}(n)$ time.

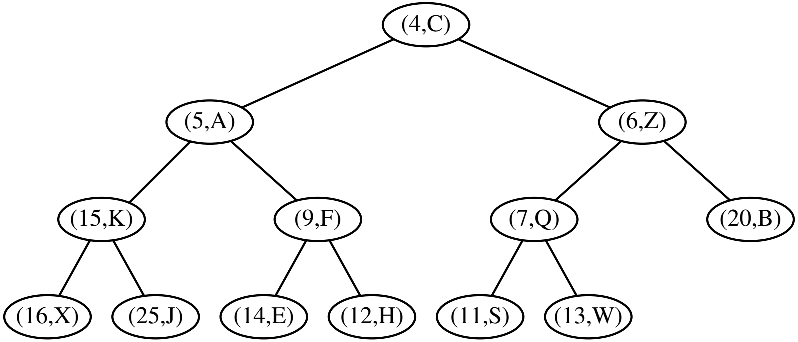
Heap

Heap

Method	Unsorted list	Sorted list	Heap
insert	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
removeMin	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$

- Unsorted list has excellent insert time but worse removeMin time.
  Sorted list has excellent removeMin time but worse insert time.
- Is it possible to get the best of both the worlds?
  Nope! It is impossible to get the best of both the worlds.
  (Why not?)
  However, we can definitely get the better of both the worlds
  using the heap data structure.

Heap: Example



Heap: Properties

A heap is a binary tree T that satisfies two properties:

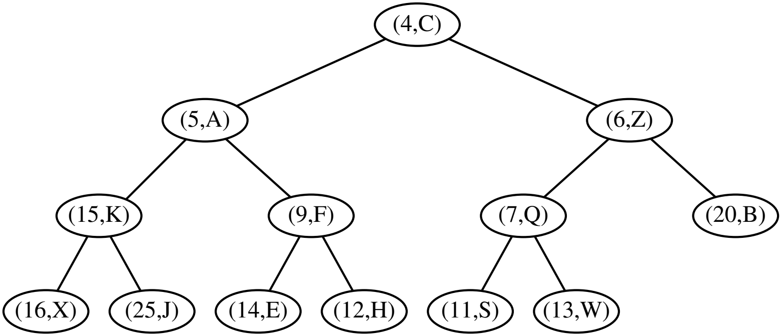
1. Relational property. Deals with how keys are stored in T.
   Heap-order property. Key stored at a node is less than or equal to the keys stored at its child nodes.
2. Structural property. Deals with the shape of T.
   Almost-complete binary tree property. A heap T with height h is an almost-complete binary tree if levels $0, 1, 2, \ldots, h-1$ of T have the maximal number of nodes possible (namely, level i has $2i$ nodes, for $0 \leq i \leq h-1$) and the remaining nodes at level h reside in the leftmost possible positions at that level.
   A heap T storing n entries has height $h = \lfloor \log n \rfloor$.

Heap: Insert

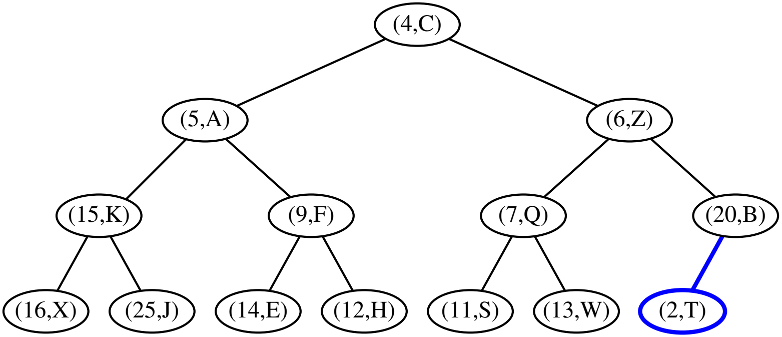
```
T.INSERT$(k, v)$
1. Insert pair $(k, v)$ at the last node
2. Up-heap bubble until heap-order property is not violated
```

- Complexity of insert is $\mathcal{O}(\log n)$

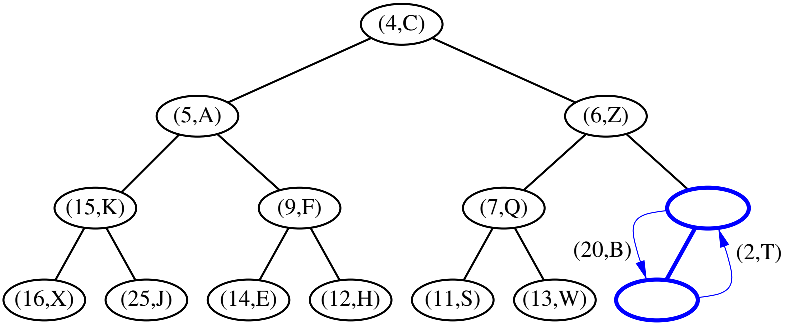
Heap: Insert (2, T)



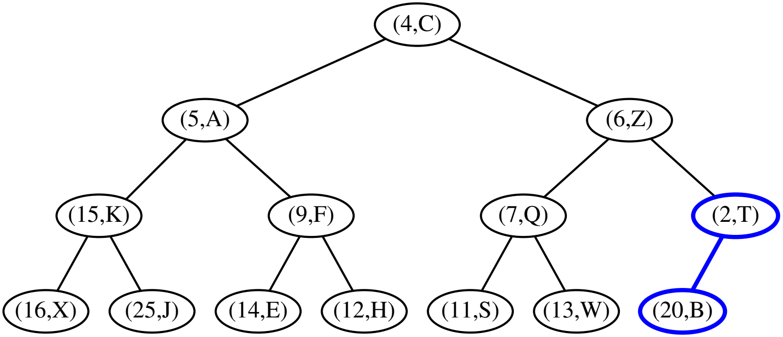
Heap: Insert (2, T)



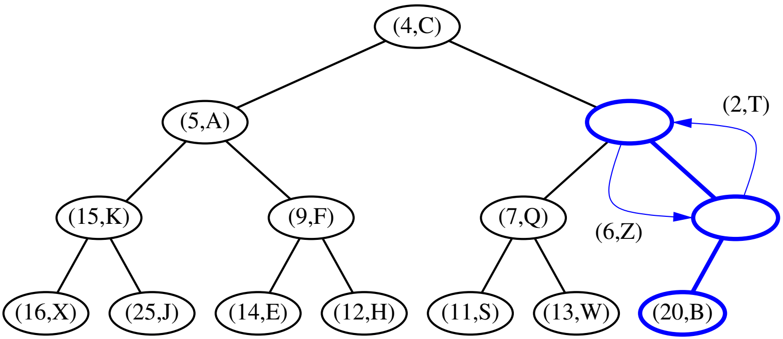
Heap: Insert (2, T)



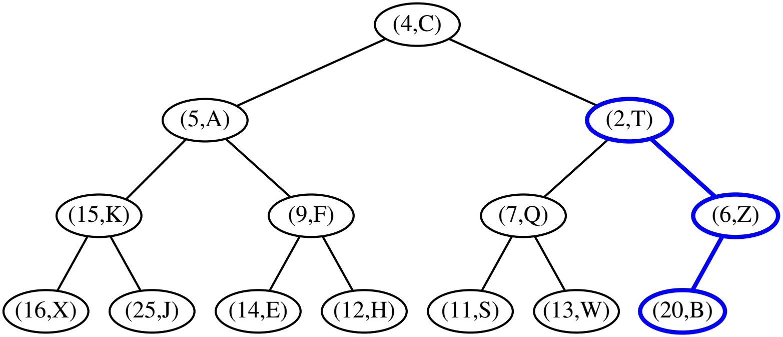
Heap: Insert (2, T)



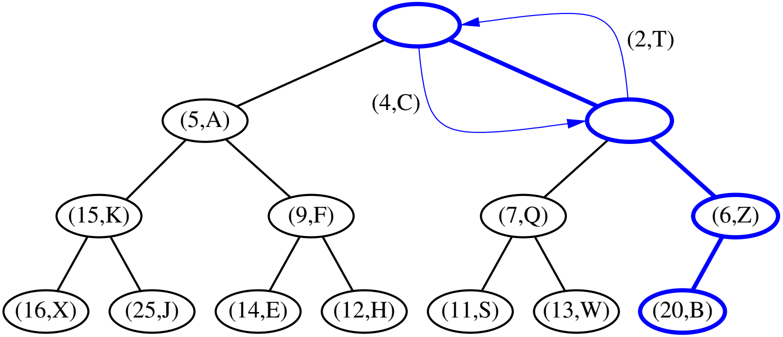
Heap: Insert (2, T)



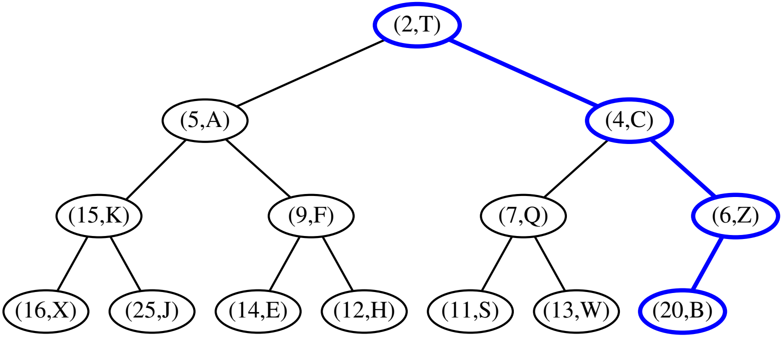
Heap: Insert (2, T)



Heap: Insert (2, T)



Heap: Insert (2, T)

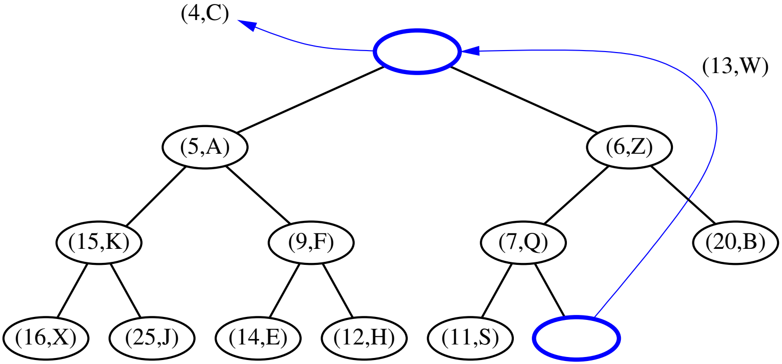


Heap: RemoveMin

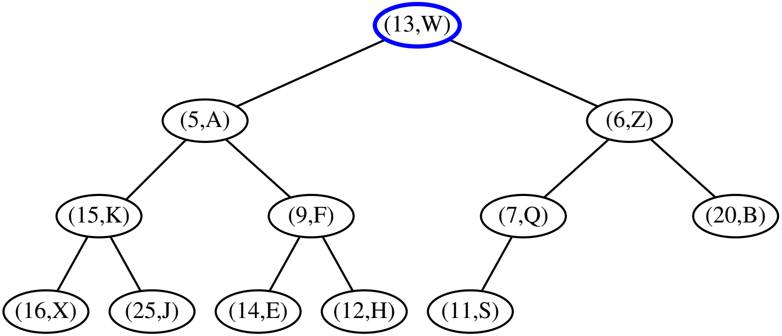
```
T.REMOVEMIN($k, v$)
1. Remove the root node
2. Move the last node to the root node position
3. Down-heap bubble until heap-order property is not violated
```

- Complexity of removeMin is $\mathcal{O}(\log n)$

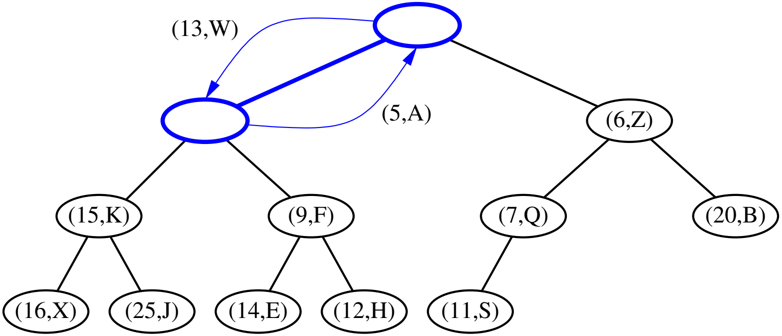
Heap: RemoveMin



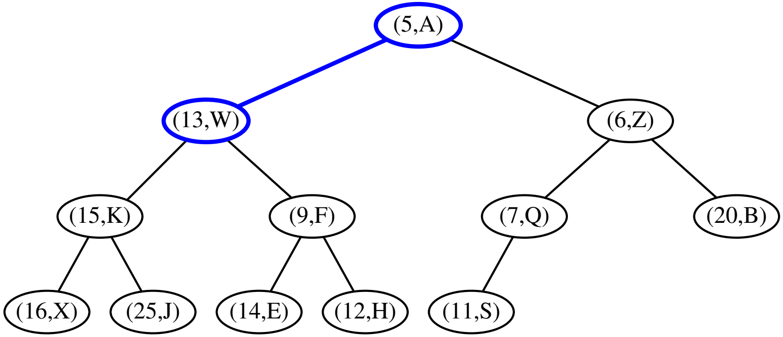
Heap: RemoveMin



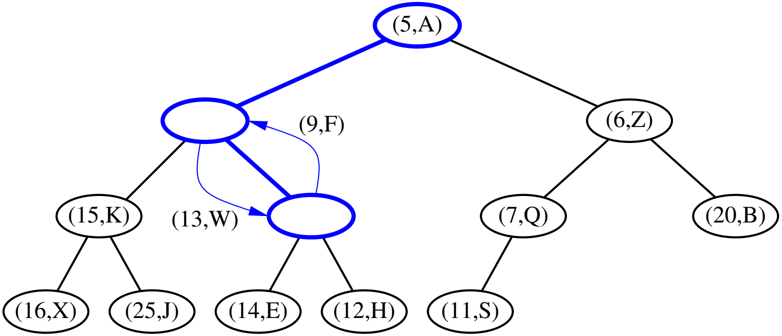
Heap: RemoveMin



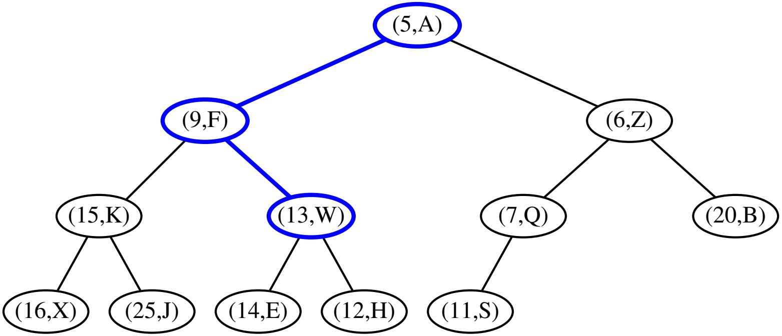
Heap: RemoveMin



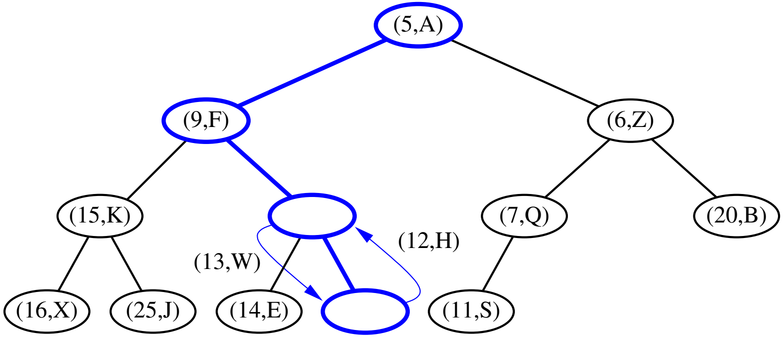
Heap: RemoveMin



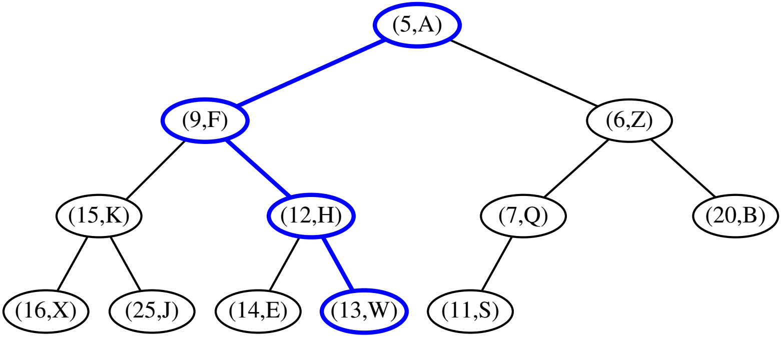
Heap: RemoveMin



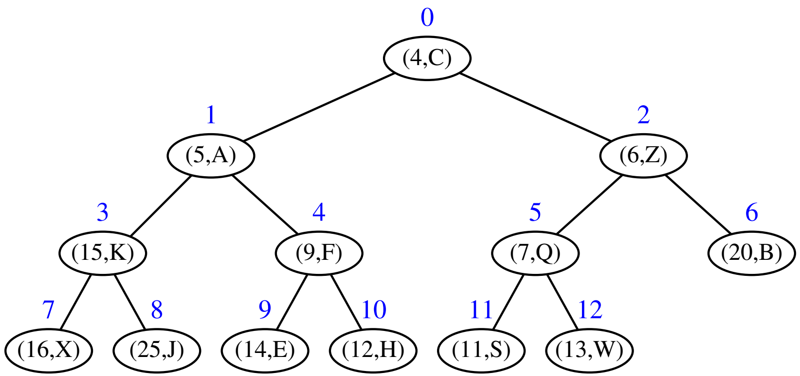
Heap: RemoveMin



Heap: RemoveMin

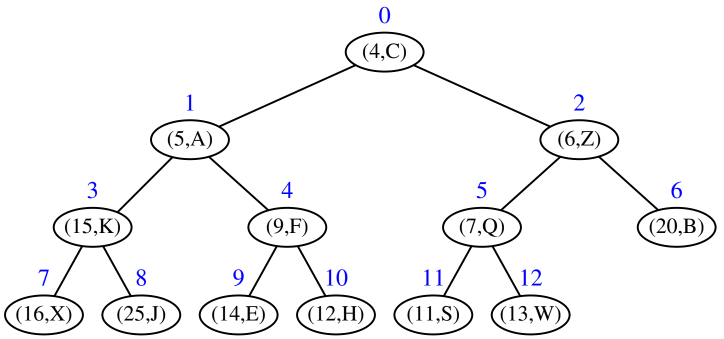


Heap: Array-based representation



(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)
0	1	2	3	4	5	6	7	8	9	10	11	12

Heap: Array-based representation



(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)
0	1	2	3	4	5	6	7	8	9	10	11	12

$$f(p) = \begin{cases} 0 & \text{if } p \text{ is the root,} \\ 2f(q)+1 & \text{if } p \text{ is the left child of position } q, \\ 2f(q)+2 & \text{if } p \text{ is the right child of position } q. \end{cases}$$

Heap: Array-based implementation

```java
public class HeapPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  protected ArrayList<Entry<K,V>> heap = new ArrayList<>();

  public HeapPriorityQueue() { super(); }
  public HeapPriorityQueue(Comparator<K> comp) { super(comp); }

  public int size() { }
  public Entry<K,V> min() {...}
  public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {...}
  public Entry<K,V> removeMin() {...}

  protected void upheap(int j) {...}
  protected void downheap(int j) {...}
  protected void heapify() {...}
  protected void swap(int i, int j) {...}

  protected int parent(int j) {...}
  protected int left(int j) {...}
  protected int right(int j) {...}
  protected boolean hasLeft(int j) {...}
  protected boolean hasRight(int j) {...}
}
```

Heap: Array-based implementation

```
/** Inserts a key-value pair and return 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;
}
```

Heap: Array-based implementation

```
/** Moves the entry at index j higher, 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
  }
}
```

Heap: Array-based implementation

```
/** Removes and returns an entry with minimal key. */
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;
}
```

Heap: Array-based implementation

```java
/** Moves the entry at index j lower, 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;
    swap(j, smallChildIndex);
    j = smallChildIndex;                 // continue at position of the child
  }
}
```

Heap: Array-based implementation

```java
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(); }
```

Heap: Complexity

Method	Array heap	Linked-tree heap
size, isEmpty, min	$\mathcal{O}(1)$	$\mathcal{O}(1)$
insert	$\mathcal{O}(\log n)^*$	$\mathcal{O}(\log n)$
removeMin	$\mathcal{O}(\log n)^*$	$\mathcal{O}(\log n)$

* $=$ amortized complexity

- Up-heap and down-heap bubbling take $\mathcal{O}(\log n)$ time.
- Array runtimes are amortized due to array resizing.

java.util.PriorityQueue class

Our Priority Queue ADT	java.util.PriorityQueue class
insert(k,v)	add(new SimpleEntry(k,v))
min()	peek()
removeMin()	remove()
size()	size()
isEmpty()	isEmpty()

- User-defined priority can be given to the class by sending a comparator object when constructing the priority queue.
- Key-value pair can be considered by using java.util.AbstractMap.SimpleEntry class

Heap Sort

Priority queue sort

```
Priority-Queue-Sort(sequence $S$, priority queue $P$)
1. Insert the $n$ elements of $S$ into $P$
2. RemoveMin the $n$ elements of $P$ into $S$
```

Priority queue sort

```java
/** Sorts sequence S, using initially empty priority queue P. */
public static <E> void pqSort(PositionalList<E> S, PriorityQueue<E,?> P) {
  int n = S.size();
  for (int j = 0; j < n; j++) {
    E element = S.remove(S.first());
    P.insert(element, null); // element is key; null value
  }
  for (int j = 0; j < n; j++) {
    E element = P.removeMin().getKey();
    S.addLast(element); // the smallest key in P is next placed in S
  }
}
```

P = unsorted list: Selection sort

		Sequence S	Priority Queue P
Input		(7, 4, 8, 2, 5, 3, 9)	()
insert	(a)	(4, 8, 2, 5, 3, 9)	(7)
	(b)	(8, 2, 5, 3, 9)	(7, 4)
	⋮	⋮	⋮
	(g)	()	(7, 4, 8, 2, 5, 3, 9)
removeMin	(a)	(2)	(7, 4, 8, 5, 3, 9)
	(b)	(2, 3)	(7, 4, 8, 5, 9)
	(c)	(2, 3, 4)	(7, 8, 5, 9)
	(d)	(2, 3, 4, 5)	(7, 8, 9)
	(e)	(2, 3, 4, 5, 7)	(8, 9)
	(f)	(2, 3, 4, 5, 7, 8)	(9)
	(g)	(2, 3, 4, 5, 7, 8, 9)	()

P = unsorted list: Selection sort

- Phase 1 time $= \sum_{i=1}^{n} \mathcal{O}(1) = \mathcal{O}(n)$
- Phase 2 time $= \sum_{i=n}^{1} \mathcal{O}(i) = \mathcal{O}\left(n^2\right)$
- Total time $= \mathcal{O}\left(n^2\right)$

P = sorted list: Insertion sort

		Sequence S	Priority Queue P
Input		(7, 4, 8, 2, 5, 3, 9)	()
insert	(a)	(4, 8, 2, 5, 3, 9)	(7)
	(b)	(8, 2, 5, 3, 9)	(4, 7)
	(c)	(2, 5, 3, 9)	(4, 7, 8)
	(d)	(5, 3, 9)	(2, 4, 7, 8)
	(e)	(3, 9)	(2, 4, 5, 7, 8)
	(f)	(9)	(2, 3, 4, 5, 7, 8)
	(g)	()	(2, 3, 4, 5, 7, 8, 9)
removeMin	(a)	(2)	(3, 4, 5, 7, 8, 9)
	(b)	(2, 3)	(4, 5, 7, 8, 9)
	⋮	⋮	⋮
	(g)	(2, 3, 4, 5, 7, 8, 9)	()

P = sorted list: Insertion sort

- Phase 1 time $= \sum_{i=1}^{n} \mathcal{O}\left(i\right) = \mathcal{O}\left(n^2\right)$
- Phase 2 time $= \sum_{i=n}^{1} \mathcal{O}\left(1\right) = \mathcal{O}\left(n\right)$
- Total time $= \mathcal{O}\left(n^2\right)$

P = heap: Heap sort

		Sequence S	Priority Queue P
Input		(7, 4, 8, 2, 5, 3, 9)	()
insert	(a)	(4, 8, 2, 5, 3, 9)	(7)
	(b)	(8, 2, 5, 3, 9)	(4, 7)
	⋮	⋮	⋮
	(g)	()	(2, 4, 3, 7, 5, 8, 9)
removeMin	(a)	(2)	(3, 4, 8, 7, 5, 9)
	(b)	(2, 3)	(4, 5, 8, 7, 9)
	(c)	(2, 3, 4)	(5, 7, 8, 9)
	(d)	(2, 3, 4, 5)	(7, 9, 8)
	(e)	(2, 3, 4, 5, 7)	(8, 9)
	(f)	(2, 3, 4, 5, 7, 8)	(9)
	(g)	(2, 3, 4, 5, 7, 8, 9)	()

P = heap: Heap sort

- Phase 1 time $= \sum_{i=1}^{n} \mathcal{O}(\log n) = \mathcal{O}(n \log n)$
- Phase 2 time $= \sum_{i=n}^{1} \mathcal{O}(\log n) = \mathcal{O}(n \log n)$
- Total time $= \mathcal{O}(n \log n)$

Priority queue sort: Complexity

Sorting algorithm	Running time
Selection sort	$\mathcal{O}\left(n^2\right)$
Insertion sort	$\mathcal{O}\left(n^2\right)$
Heap sort	$\mathcal{O}\left(n \log n\right)$