Data Structures

Augustin Cosse.

Spring 2021

April 22, 2021

- When discussing stacks and queues, we introduced the queue ADT as a collection of objects that are added and removed according to the First In First Out (FIFO) principle
- A company's customer call center embodies such a model in which customers are told "calls will be answered" in the order that they were received.

- In practice, there are many applications for which using a queue might be a good idea but for which the first in first out policy might not suffice.
- A first example is the air traffic control center where one has to decide which flight to clear for landing from among many approaching the airport
- The choice may be influenced by factors such as each plane's distance from the runway, time spent waiting in holding pattern, or amount of remaining fuel
- Most of the time, it is quite unlikely that the landing decisions will be based purely on FIFO policy

- In other situations, FIFO might be seem reasonable but might have to be combined with other types of priorities
- As an example of this, suppose that because of the possibility of cancellations on a particular flight, an airline maintains a queue of standby passengers hoping to get a seat.
- Although the priority of a passenger is influenced by the check-in time of that passenger, other considerations might include the fare paid and frequent flyer status

- Today we will consider a new abstract data type known as Priority Queue
- A Priority Queue is a collection of prioritized elements that allows arbitrary element insertion and allows the removal of the element that has the first priority
- When an element is added to a priority queue, the user designates its priority by providing an associated key
- The element with the minimal key will be the next to be removed from the queue

The priority queue ADT (part I)

- Although it is common for priorities to be expressed numerically, any Java object may be used as a key, as long as there exists means to compare any two instances a and b in a way that defines a natural order of the keys
- We model an element and its priority as a key-value composite known as an entry.

The priority queue ADT (part II)

- We will define the priority queue ADT to support the following methods

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 queue is empty
removeMin()	Removes and return an entry (k, v) having minimal key from the priority queue. Returns null if queue is empty
size()	Returns the number of entries in the queue
isEmpty()	Returns a boolean indicating whether the queue is empty or not

The priority queue ADT (part II)

- A priority queue may have multiple entries with equivalent keys, in which case the methods min and removeMin may report an arbitrary choice among those entries having minimal key.

- The table below shows a series of operations and their effects on an initially empty priority queue.

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	{ }

Implementing a Priority Queue

- We use the Entry type in the interface for priority Queues
- This allows us to return both a key and a value as a single object from methods such as `min` and `removeMin`
- We also define the `insert` method to return an entry
- In more advanced, adaptable priority queues, that entry can be subsequently removed or updated.

```java
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();}
```

Comparing keys with Total Order

- In defining the priority queue ADT, we can allow any type of object to serve as a key but we must be able to compare keys to each other in a meaningful way
- Moreover, the result of the comparison must not be contradictory.
- For a comparison rule, which we denote by $\leq$ to be self consistent, it must define a total order relation which is to say that it satisfies the following properties for any keys k_1, k_2 and k_3
  - Comparability: $k_1 \leq k_2$ or $k_2 \leq k_1$
  - Antisymmetry : if $k_1 \leq k_2$ and $k_2 \leq k_1$ then $k_1 = k_2$
  - Transitivity: if $k_1 \leq k_2$ and $k_2 \leq k_3$ then $k_1 \leq k_3$
- Comparability implies Reflexivity: $k \leq k$

Comparing keys with Total Order

- A comparison rule $\leq$ that defines a total order relation will never lead to a contradiction

- Such a rule defines a linear ordering among a set of keys. In particular, if a (finite) set of elements has a total order defined for it, then the notion of a minimal key $k_{\min}$ is well defined as a key for which $k_{\min} \leq k$ for any other key k in our set.

The Comparable and Comparator interfaces

- Java provides two means for comparing object types.

- The first of these is that a class may define what is known as the natural ordering of its instances by formally implementing the `java.lang.Comparable` interface. which includes a single method `compareTo`.

- The syntax `a.compareTo(b)` must return an integer i with the following meaning:

  - $i < 0$ designates that $a < b$

  - $i = 0$ designates that $a = b$

  - $i > 0$ designates that $a > b$

- For example, the `compareTo` method of the `String` class defines the natural ordering of Strings to be lexicographic, which is a case sensitive extension of the alphabetic ordering to Unicode.

The Comparable and Comparator interfaces

- In some applications, we may want to compare objects according to some notion other than their natural ordering
- For example, we might be interested in which of two strings is the shortest, or in defining our own complex rules for judging which of two stocks is more promising
- To support generality, Java defines the `java.util.Comparator` interface
- A comparator is an object that is external to the class of the keys it compares
- It provides a method with the signature `compare(a, b)` that returns an integer with similar meaning to the `compareTo` method

The Comparable and Comparator interfaces

- As a concrete example, the code fragment below defines a comparator that evaluates strings based on their length (rather than their natural lexicographic order)

```
public class StringLengthComparator
              implements Comparator<String>{
/**compares two strings according to their lengths */
  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;

  }
}
```

Comparators and the Priority Queue ADT

- For a general reusable form of a priority queue, we allow a user to choose any key type and to send an appropriate comparator instance as a parameter to the priority queue constructor

- The priority queue will use that comparator anytime it needs to compare two keys to each other

- For convenience, we also allow a default priority queue to instead rely on the natural ordering for the given keys (assuming keys come from the comparable interface). In that case, we build our own instance of a `DefaultComparator` class shown below

```java
public class DefaultComparator<E> implements Comparator<E>{
  public int compare(E a, E b) throws ClassCastException{
    return ((Comparable<E>) a).compareTo(b);
  }}
```

The abstract Priority Queue Base class

- We start by defining a general abstract base class named `AbstractPriorityQueue`

```
public abstract class AbstractPriorityQueue<K,V>
implements PriorityQueue<K,V> {

// Part I: The nested PQEntry class
  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;}
// methods of the Entry interface
  public K getKey( ) { return k; }
  public V getValue( ) { return v; }
// utilities not exposed as part of the Entry interface
  protected void setKey(K key) { k = key; }
  protected void setValue(V value) { v = value; } }
```

The abstract Priority Queue Base class

```
public abstract class AbstractPriorityQueue<K,V>
implements PriorityQueue<K,V> {

// Part II: instance variables and methods
private Comparator<K> comp;
  /** Creates empty queue using the given comparator*/
  protected AbstractPriorityQueue(Comparator<K> c)
  { comp = c; }
  /** Creates empty queue based on natural ordering*/
  protected AbstractPriorityQueue( )
  { this(new DefaultComparator<K>( )); }
  /** Comparing keys */
  protected int compare(Entry<K,V> a, Entry<K,V> b) {
    return comp.compare(a.getKey( ), b.getKey( ));
}
```

The abstract Priority Queue Base class

```
public abstract class AbstractPriorityQueue<K,V>
         implements PriorityQueue<K,V> {
 // Part III: instance variables and methods (ctd)
  private Comparator<K> comp;
  /** Determines whether a key is valid. */
    protected boolean checkKey(K key)
              throws IllegalArgumentException {
      try {
        return (comp.compare(key,key) == 0);
        // see if key can be compared to itself
     } catch (ClassCastException e) {
  throw new
         IllegalArgumentException("Incompatible key");
  }}
  /** Tests whether the priority queue is empty. */
  public boolean isEmpty( ) { return size( ) == 0; }}
```

The abstract Priority Queue Base class

- In our first implementation of a priority queue, we will store entries within an unsorted linked list
- For internal storage, key value pairs are represented as composites , using instances of the inherited `PQEntry` class
- These entries are then stored within a PositionalList that is an instance variable.
- We first assume that the Positional List is implemented by means of a doubly linked list

The abstract Priority Queue Base class

- We will begin with an emtpy list when a new priority queue is constructed
- At any point, the size of the list equal the number of key-value pairs currently stored in the priority queue
- Our priority queue size method hence simply returns the length of the internal list
- Each time a key-value pair is added to the priority queue via the insert method, we create a new `PQEntry` composite for the give key and value and add that entry to the list

The abstract Priority Queue Base class

- The remaining challenge is that when `min` or `removeMin` is called, we must locate the entry with the minimal key
- Because the entries are not sorted, we must inspect all entries to find the one with the minimal key
- In order to do this, we define a private `findMin` utility that returns the position of an entry with minimal key

```java
public class UnsortedPriorityQueue<K,V>
        extends AbstractPriorityQueue<K,V> {

// Part I, constructors and minimal key
  private PositionalList<Entry<K,V>> list =
                    new LinkedPositionalList<>( );
  /** Creates empty queue with natural ordering */
  public UnsortedPriorityQueue( ) { super( ); }
  /** Creates empty queue using compator*/
  public UnsortedPriorityQueue(Comparator<K> comp)
  { super(comp); }

  /** Returns Position of minimal key */
  private Position<Entry<K,V>> findMin( ) {
    Position<Entry<K,V>> small = list.first( );
    for (Position<Entry<K,V>> walk : list.positions( ))
       if(compare(walk.getElement( ), small.getElement( ))<0)
        small = walk; // found an even smaller key
    return small;}
```

- The `min` method simply uses the position to retrieve the entry when preparing a key-value tuple to return

```
// Part II,insert, min and removeMin methods
  public Entry<K,V> insert(K key, V value)
            throws IllegalArgumentException {
    checkKey(key); // key-checking method
    Entry<K,V> newest = new PQEntry<>(key, value);
    list.addLast(newest);
    return newest;}

  public Entry<K,V> min( ) {
    if (list.isEmpty( )) return null;
    return findMin( ).getElement( );}
```

- Knowledge of the position allows the `removeMin` method to invoke the remove method on the positional list

```java
// Part III removeMin
/** Removes and returns an entry with minimal key. */
  public Entry<K,V> removeMin( ) {
    if (list.isEmpty( )) return null;
    return list.remove(findMin( ));
}

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

Implementing a Queue with a sorted list

- The running times for the various methods of the `UnsortedPriorityQueue` class can be found in the table below

Method	**Running Time**
size	$O(1)$
isEmpty	$O(1)$
insert	$O(1)$
min	$O(n)$
removeMin	$O(n)$

Implementing a Queue with a sorted list

- To improve the running times for the `min` and `removeMin` methods, we can decide to store the entries by non decreasing keys. This ensures that the first element of the list is the entry with the smallest key
- For such a sorted Queue, the implementation of `min` and `removeMin` are rather straightforward
- Assuming that the list is implemented with a doubly linked list, operations `min` and `removeMin` take $O(1)$ time

```java
public class SortedPriorityQueue<K,V>
             extends AbstractPriorityQueue<K,V> {
  private PositionalList<Entry<K,V>> list =
                              new LinkedPositionalList<>( );
/** Empty priority queue based on natural ordering */
  public SortedPriorityQueue( ) { super( ); }
/** Empty priority queue using given comparator*/
  public SortedPriorityQueue(Comparator<K> comp)
  { super(comp); }
/** Returns an entry with minimal key */
  public Entry<K,V> min( ) {
    if (list.isEmpty( )) return null;
    return list.first( ).getElement( );}
/** Removes and returns an entry with minimal key */
  public Entry<K,V> removeMin( ) {
    if (list.isEmpty( )) return null;
    return list.remove(list.first( )); }

  public int size( ) { return list.size( ); }}
```

Implementing a Queue with a sorted list

- This benefit comes at a cost though and the method `insert` now requires that we scan the list to find the appropriate position to insert the new entry.
- Our implementation starts at the end of the list, walking backward until the new key is smaller than that of an existing entry
- In the worst case, it progresses until reaching reaching the front of the list
- As a result, the insert method takes $O(n)$ worst case time where n is the number of entries in the priority queue at the time the method is executed

```java
public Entry<K,V> insert(K key, V value)
            throws IllegalArgumentException {
  checkKey(key); // key-checking method
  Entry<K,V> newest = new PQEntry<>(key, value);
  Position<Entry<K,V>> walk = list.last( );
  // walk backward, looking for smaller key
  while (walk != null
        && compare(newest, walk.getElement( )) < 0)
    walk = list.before(walk);
  if (walk == null)
    list.addFirst(newest); // new key is smallest
  else
    list.addAfter(walk, newest);
    // newest goes after walk
return newest;}
```

Implementing a Queue with a sorted list

Method	**Unsorted List**	**Sorted List**
size	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
removeMin	$O(n)$	$O(1)$

Implementing a Queue with a sorted list

- In short, we see an interesting tradeoff when implementing priority queues: unsorted lists support fast insertion but slow queries and deletions while a sorted list allows fast queries and deletions, but slow insertions

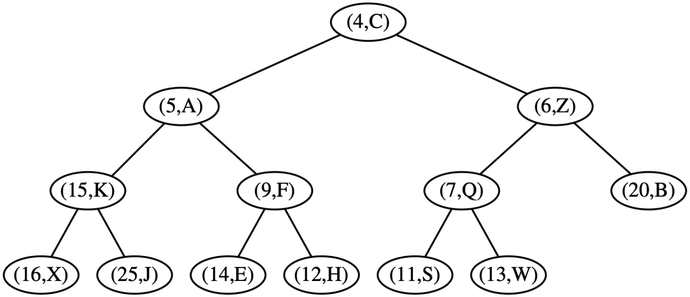
Method	**Unsorted List**	**Sorted List**
size	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
removeMin	$O(n)$	$O(1)$

Heaps

- As we saw, with priority queues, when using an unsorted list to store entries, we can perform insertions in $O(1)$ but finding or removing an element with minimal key requires an $O(n)$ time loop through the entire collection.
- In constrast, if using a sorted list, we can trivially find or remove the minimal element in $O(1)$ time, bu adding a new element to the queue may require $O(n)$ time to restore the sorted order
- We can in fact provide a more efficient realization of a priority queue using a data structure called a binary heap. This data structure will allow us to perform both insertions and removals in logarithmic time
- The fundamental way the heap achieves this improvement is by using the structure of a binary tree to fin a compromise between elements being entirely unsorted and perfectly sorted

Heaps

- A Heap is a binary tree T that stores entries as its positions and that satisfies two additional properties:
  - A relational property defined in terms of the way keys are stored in T
  - A structural property defined in terms of the shape of T itself



Example of Heap with integer keys

Heaps

- The relational property is known as the Heap-Order Property and states that in a heap T, for every position p other than the root, the key stored at p is greater than or equal to the sey stored at p's parent
- 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
- Also 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"

Heaps

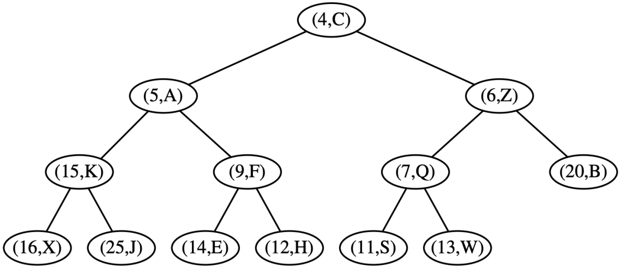
- For effciency reasons, we want the heap T to have as small a height as possible
- We enforce this requirement by insisting that the heap T satisfy an additional structural property known as the Complete Binary Tree Property
- The Complete Binary Tree Property states that a heap T with a height h is a **complete** binary tree if levels $0, 1, 2, \ldots, h-1$ of T have the maximal number of nodes possible (namely level i has 2^i nodes, for $1 \leq i \leq h-1$) and the remaining nodes at level h reside in the leftmost possible position at that level

Heaps

- The tree below is complete because levels 0, 1 and 2 are full and the six nodes at level 3 are in the six leftmost possible positions
- Let h denote the height of T. The fact that T is complete has an important consequence summarized by the following proposition

Proposition

A heap T storing n entries has height $\lfloor \log n \rfloor$



Heaps

Proposition

A heap T storing n entries has height $\lfloor \log n \rfloor$

Proof.

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

$$n \geq 2^h - 1 + 1 \quad \text{and} \quad n \leq 2^{h+1} - 1$$

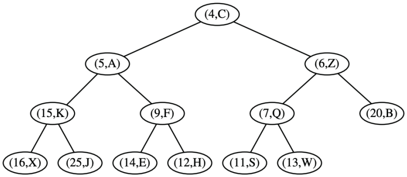
Taking the logarithm on both sides of inequality $n \geq 2^h$ we see that $h \leq \log n$. Taking the logarith on both sides of the inequality $n \leq 2^{h+1} - 1$ we get $\log(n+1) \leq h+1$ since h is an integer, these two inequality imply that $h = \lfloor \log n \rfloor$ □

Implementing a priority queue with a heap

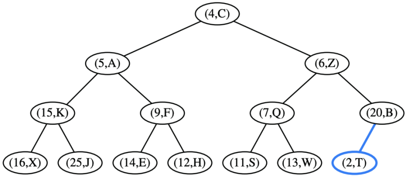
- The previous proposition has an important consequence for it implies that if we can perform update operations on a heap in time proportional to its height, then those operations will run in logarithmic time
- The `size` and `isempty` methods can be implemented based on examination of the tree and the min operation is equally trivial because the heap property assures that the element at the root of the tree has a minimal key.
- The interesting algorithms are those for implementing the `insert` and `removeMin` methods

Implementing a priority queue with a heap

- Let us first consider how to perform `insert(k, v)` on a priority queue implemented with a heap T
- We store the pair (k, v) as an entry at a new node of the tree. To maintain the complete binary tree property, that new node should be placed at a position p just beyond the rightmost node at the lowest level of the tree, or at as the leftmost position of a new level, if the bottom level is already full



(a)



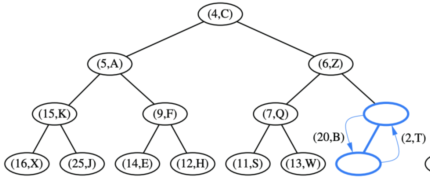
(b)

Implementing a priority queue with a heap

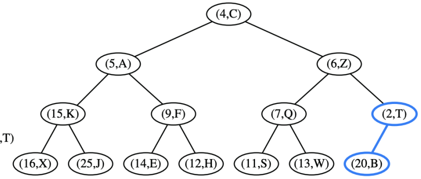
- After this action, the tree T is complete, but it may violate the heap-order property (i.e. the key stored at p is greater than the key stored at p's parent). Hence unless the position p is the root of the tree, we compare the key at position p to that of p's parent which we denote as q.

- 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 position p and q



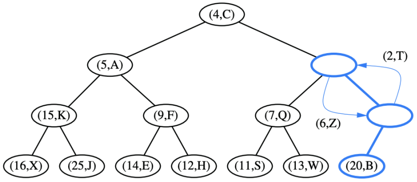
(c)



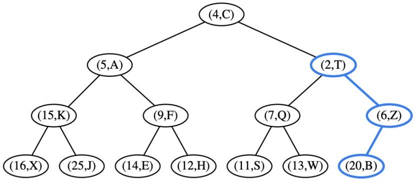
(d)

Implementing a priority queue with a heap

- The swap causes the new entry to move up one level. Again, the heap order property may be violated, so we repeat the process, going up in T 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.



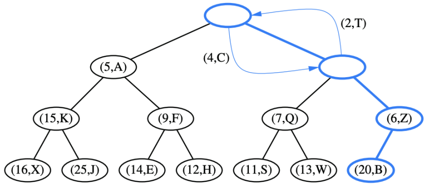
(e)



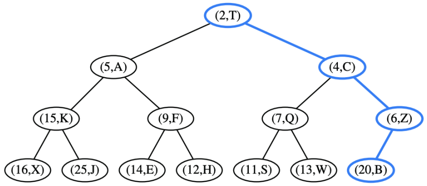
(f)

Implementing a priority queue with a heap

- A swap either resolves the violation of the heap-order property, or propagates it one level up in the heap. In the worst case scenario, up heap bubbling causes the new entry to move all the way up in the heap. Thus in the worst case scenario, teh number of swaps performed in the execution of the method `insert` is equal to the heigth of T which (as we saw on the previous slides) is bounded by $\lfloor \log n \rfloor$



(g)



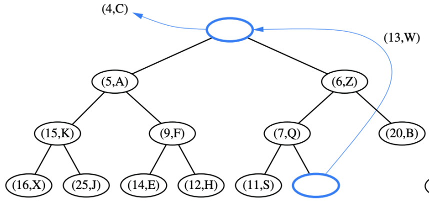
(h)

Removing the entry with the minimal key

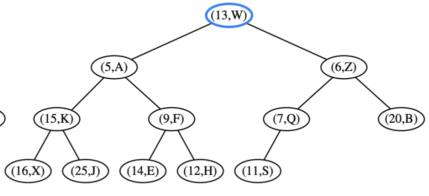
- We now turn to the implementation of the `removeMin` method of the priority queue ADT.
- We know that the entry with the smallest key is stored at the root of the tree T. However, in general we cannot simply delete node r because this would leave two disconnected subtrees
- Instead we should ensure 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 bottommost level of the treee

Removing the entry with the minimal key

- To preserve the entry from the last position p we copy it to the root



(a)



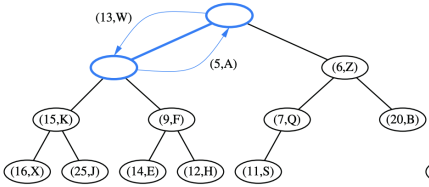
(b)

Down-Heap Bubbling after a Removal

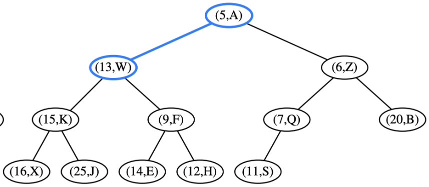
- We are not yet done however, for even though T is now complete, it violates the heap order property.
- If T has only one node, then the heap order property is trivially satisfied and the algorithm terminates. Otherwise, we distinguish two cases
  - If p has no right child let c be the left child of p
  - Otherwise (p has both children), let c be the child with minimal key
- If key $k_p \leq k_c$, the heap order property is satisfied and the algorithm terminates
- If instead $k_p > k_c$, then we need to restore the heap-order property. This can be locally achieved by swapping the entries stored at p and c

Down-Heap Bubbling after a Removal

- It is worth noting that when p has two children, we intentionnally consider the smaller key of the children. In this case, not only is the key of c smaller than that of p, it is at least as small as the key at c's sibling. This ensures that the heap order property is locally restored when that smaller key is promoted above the key that had been at p, and that at c's sibling



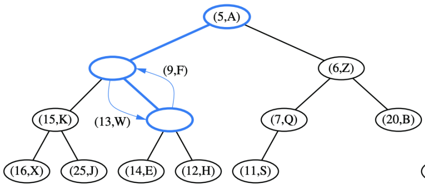
(c)



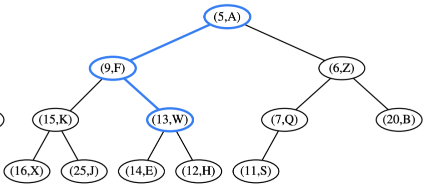
(d)

Down-Heap Bubbling after a Removal

- Having restored the heap property for node p relative to its children, there may be a violation of this property at c. We may therefore have to continue swapping down T until no violation of the heap-order property occurs. This downward swapping process is called down-heap bubbling



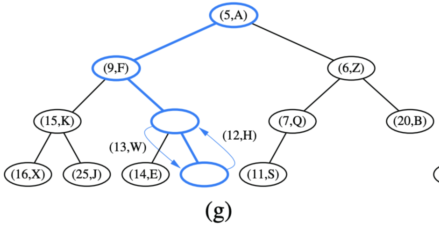
(e)

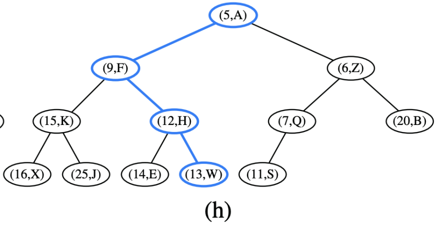


(f)

Down-Heap Bubbling after a Removal

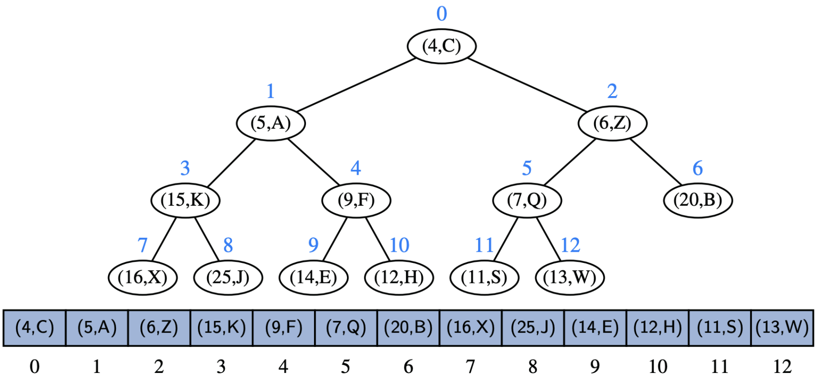
- A swap either resolves the violation of the heap order property, or propagates it one level down in the heap. In the worst case, an entry moves all the way down to the bottom level.
- The number of swaps performed in the execution of the method `removeMin` is, in the worst case, equal to the height of the heap T, that is $\lfloor \log n \rfloor$





Array based representation of Complete Binary tree

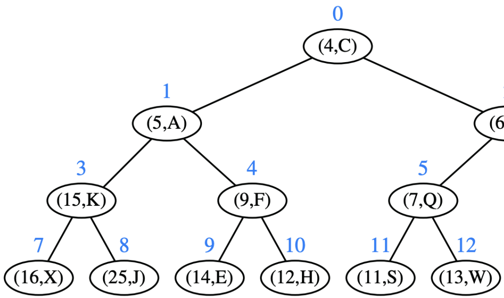
- The array based representation of a binary tree is especially suitable for a complete binary tree. We recall that in this implementation, the elements of the tree are stored in an array-based list A such that the element at position p is stored in A with index equal to the level number $f(p)$ of p



(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

Array based representation of Complete Binary tree

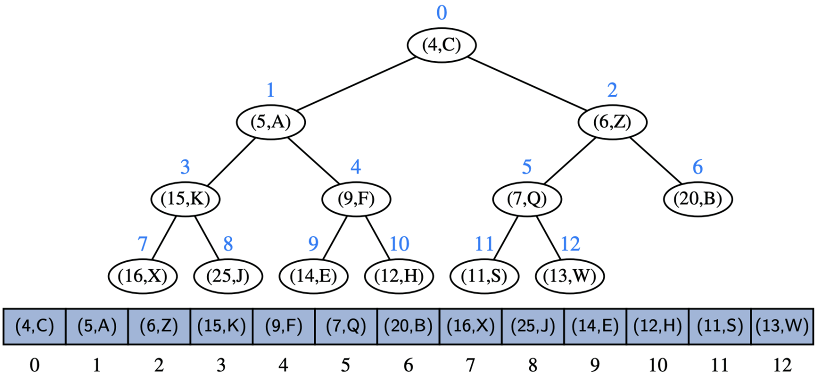
- Recall that the level number $f(p)$ is defined as
  - If p is the root, then $f(p) = 0$
  - If p is the left child of a position q, then $f(p) = 2f(q) + 1$
  - If p is the right child of position q, then $f(p) = 2f(q) + 2$



(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

Array based representation of Complete Binary tree

- For a tree with size n, teh elements have contiguous indices in the range $[0, n-1]$ and the last position is always at index $n-1$



Array based representation of Complete Binary tree

- The array based heap representation avoids some of the complexities of a linked tree structure. Specifically, methods `insert` and `removeMin` depend on locating the last position of the heap while with 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

- The space usage of such an array based representation of a complete binary tree with n nodes is $O(n)$, and the time bounds of methods for adding or removing elements become amortized.

```java
public class HeapPriorityQueue<K,V> extends
                            AbstractPriorityQueue<K,V> {
/**Priority queue using array-based heap (part I). */
protected ArrayList<Entry<K,V>> heap =
                             new ArrayList<>( );
/**Creates empty queue w/ natural ordering*/
public HeapPriorityQueue( ) { super( ); }
/**Creates empty queue w/ given comp. to order keys.*/
public HeapPriorityQueue(Comparator<K> comp)
 { super(comp); }
// protected utilities
protected int parent(int j) { return (j-1)/2;}
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( ); }
```

- We also add to the class protected utilities `swap`, `upheap` and `downheap`

```
public class HeapPriorityQueue<K,V> extends
                              AbstractPriorityQueue<K,V> {
/**Priority queue using array-based heap (part II). */
/** Exchanges entries i and j */
protected void swap(int i, int j) {
  Entry<K,V> temp = heap.get(i);
  heap.set(i, heap.get(j));
  heap.set(j, temp);}
```

- We also add to the class protected utilities `swap`, `upheap` and `downheap`

```java
/** Moves entry at index j higher*/
protected void upheap(int j) {
  while (j > 0) { // continue until root (or break)
    int p = parent(j);
    if (compare(heap.get(j), heap.get(p)) >= 0) break;
    swap(j, p);
    j = p; }}
```

- We also add to the class protected utilities `swap`, `upheap` and `downheap`

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

- A new entry is added at 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.

- We conclude with size, insert and removeMin() methods

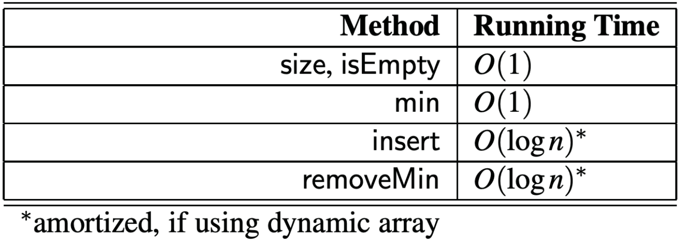
```java
/** Returns the number of items in the queue. */
public int size( ) { return heap.size( ); }
/** Returns (not remove) entry with minimal key*/
public Entry<K,V> min( ) {
  if (heap.isEmpty( )) return null;
  return heap.get(0);
}
/** Inserts and returns a key-value pair*/
public Entry<K,V> insert(K key, V value)
                throws IllegalArgumentException {
  checkKey(key); // auxiliary key-checking method
  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;
}
```

- We conclude with `size`, `insert` and `removeMin()` methods

```java
/** Removes and returns minimal key entry */
public Entry<K,V> removeMin( ) {
  if (heap.isEmpty( )) return null;
  Entry<K,V> answer = heap.get(0);
  swap(0, heap.size( )-1); // put min at the end
  heap.remove(heap.size( )- 1); // remove it from list;
  downheap(0); // then fix new root
  return answer;
}}
```

- The run time analysis of the most important methods of the Priority Queue ADT for a heap implementation can be carried out by considering the following facts:
  - The heap T has n nodes, each storing a reference to a key-value entry
  - The height of the heap is $\log n$ (the heap is complete)
  - The `min` operation runs in $O(1)$ (because the root of the tree contains such an element)
  - Locating the last postion of a heap as required by the `insert` and `removeMin` methods can be performed in $O(1)$ (array based) or $O(\log n)$ (linked tree)
  - In the worst case, `up heap` and `down heap` bubbling perform a number of swaps equal to the height of T

- The heap data structure is a very efficient realization of the priority queue ADT, independent of whether the heap is implemented with a linked structure or an array
- The heap implementation achieves fast running times for both insertion and removal, unlike the implementations that were based on using unsorted or sorted lists



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

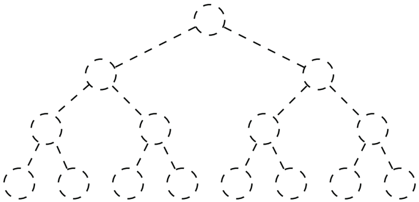
*amortized, if using dynamic array

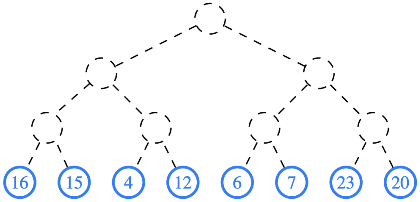
Bottom Up heap construction

- If we start with an initially empty heap, n successive calls to the insert operation will run in $O(n \log n)$ in the worst case. However if all n key value pairs to be stored in the heap are known in advance, there is an alternative bottom up construction that runs in $O(n)$ time
- To introduce the this bottom up heap construction, we will assume that the number of keys satisfies $n = 2^{h+1} - 1$ (that is the heap is a complete binary tree with every level being full). It therefore also have height $h = \log(n+1) - 1$

Bottom Up heap construction

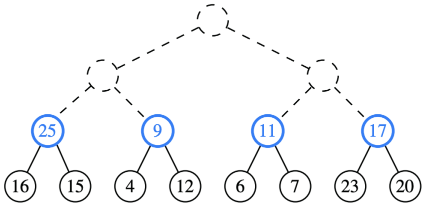
- Bottom up heap construction consists of the following $h + 1 = \log(n + 1)$ steps
  - In the first step, we construct $(n + 1)/2$ elementary heaps storing one entry each

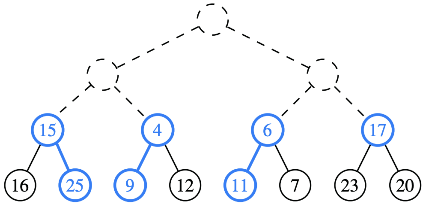




Bottom Up heap construction

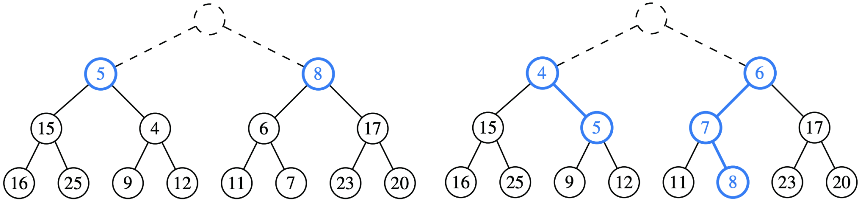
- Bottom up heap construction consists of the following $h + 1 = \log(n + 1)$ steps
  - In the second step, we form $(n + 1)/4$ elementary heaps storing 3 entries each, by joining pairs of elementary heaps and adding one new entry. The new entry is placed at the root and may have to be swapped with the entry stored at a child to preserve the heap-order property





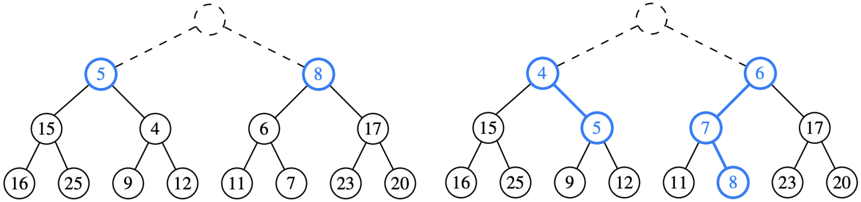
Bottom Up heap construction

- Bottom up heap construction consists of the following $h + 1 = \log(n + 1)$ steps
  - In the third step, we form $(n + 1)/8$ heaps storing 7 entries each, by joining pairs of of 3 entries heaps (constructed in the previous step) and adding a new entry. The new entry is initially placed at the root but may have to move down with a down heap bubbling to preserve the heap order property



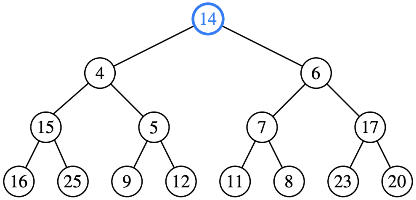
Bottom Up heap construction

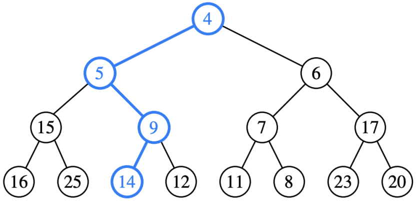
- Bottom up heap construction consists of the following $h + 1 = \log(n + 1)$ steps
  - In the generic i^{th} step, $2 \leq i \leq h$, we for $(n + 1)/2^i$ heaps, each storing $2^i - 1$ entries, by joining pairs of heaps storing $2^{i-1} - 1$ entries (constructed in the previous step) and adding a new entry. The new entry is initially placed at the root but may have to move down with a down heap bubbling to preserve the heap order property



Bottom Up heap construction

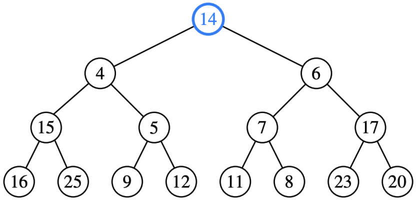
- Bottom up heap construction consists of the following $h + 1 = \log(n + 1)$ steps
  - Finally, in the last step, we form the final heap, storing all n entries by joining two heaps storing $(n - 1)/2$ entries each and adding a new entry. The new entry is placed initially at the root by may again have to move down with a down heap bubbling to preserve the heap order property

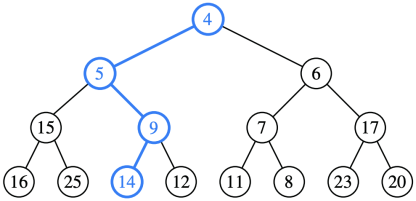




Bottom Up heap construction

- Bottom up heap construction consists of the following $h + 1 = \log(n + 1)$ steps
  - Finally, in the last step, we form the final heap, storing all n entries by joining two heaps storing $(n - 1)/2$ entries each and adding a new entry. The new entry is placed initially at the root by may again have to move down with a down heap bubbling to preserve the heap order property





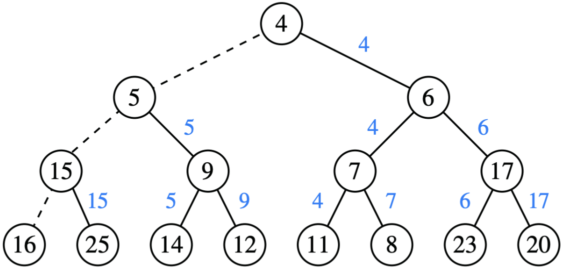
Bottom Up heap construction

Proposition

Bottom-up construction of a heap with n entries takes $O(n)$ time, assuming two keys can be compared in $O(1)$ time

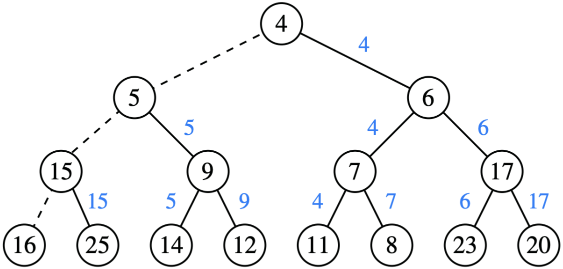
- The primary cost of the algorithm is due to the down heap steps that are performed at each of the non leaf positions

- Let π_v denote the path of T from nonleaf node v to its "'in order succcessor' leaf (that is to say the path that starts at v then goes to the right child of v then does down leftward until



Bottom Up heap construction

- The number of edges of π_v is proportionnal to the height of the subtree rooted at v and gives a bound on complexity of the down heap bubbling step from v
- From this we can bound the total running time of the bottom up heap construction by the sum $\sum_v |\pi_v|$
- The paths π_v are edge disjoints and the total sum of all path length is thus bounded by the total number of edges in the



Sorting with a Priority Queue

- One application of Priority Queues is sorting, where we are given a sequence of elements that can be compared according to a total order relation and we want to rearrange them in increasing order
- The algorithm for sorting a sequence S with a priority queue P is quite simple and consists of the following two phases:
  - In the first phase, we insert the elements of S into an initially empty priority queue P by means of a series of n `insert` operations
  - In the second phase, we extract the elements from P in non decreasing order by means of a series of n `removeMin` operations, putting them back into S in that particular order

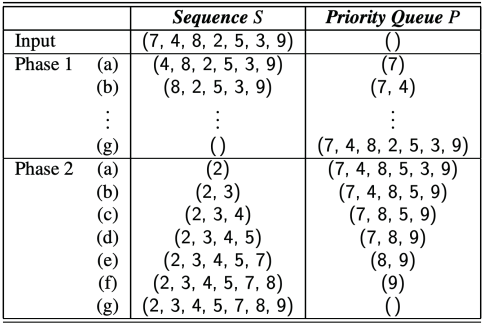
Sorting with a Priority Queue

```
/** Sorts sequence S using priority queue. */
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);
}
  for (int j=0; j < n; j++) {
    E element = P.removeMin( ).getKey( );
    S.addLast(element);
}}
```

- Sorting based on priority queues serves as a basis for several popular sorting algorithms including selection-sort, insertion-sort, heap-sort

Selection Sort

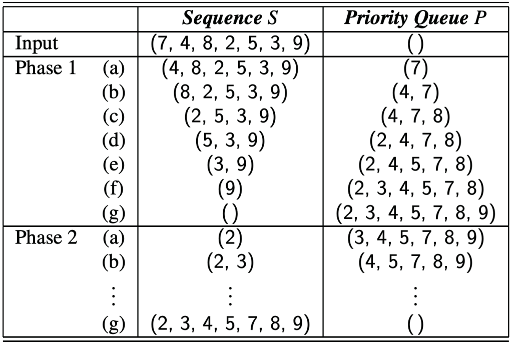
- In phase 1 of our sorting algorithm, we insert the elements into a queue P while in phase 2, we remove the elements using `removeMin`.
- When using an unsorted queue, the insertion phase takes $O(n)$, while each `removeMin` operation takes $O(L)$ where L is the current length of the queue.
- The bottleneck is thus the selection of the min value and we call this approach Selection Sort



		Sequence S	***Priority Queue P***
Input		(7, 4, 8, 2, 5, 3, 9)	()
Phase 1	(a)	(4, 8, 2, 5, 3, 9)	(7)
	(b)	(8, 2, 5, 3, 9)	(7, 4)
	⋮	⋮	⋮
	(g)	()	(7, 4, 8, 2, 5, 3, 9)
Phase 2	(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)	()

Insertion Sort

- When implementing the priority queue with a sorted list, the bottleneck becomes the insertion and the resulting algorithm is known as Insertion-Sort



		Sequence S	***Priority Queue P***
Input		(7, 4, 8, 2, 5, 3, 9)	( )
Phase 1	(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)
Phase 2	(a)	(2)	(3, 4, 5, 7, 8, 9)
	(b)	(2, 3)	(4, 5, 7, 8, 9)
	⋮	⋮	⋮
	(g)	(2, 3, 4, 5, 7, 8, 9)	( )

Insertion Sort

- We can compute the runtime of both of theinsertion and selection sort algorithms by noting that each of their bottleneck operations requires a number of operations proportional to the current size of the list. The complexity of those algorithms is thus bounded as

  $$O(n + (n-1) + \ldots + 1) = O(n^2)$$

  for selection sort and

  $$O(1 + 2 + \ldots + (n-1) + n) = O(n^2)$$

  for insertion sort.

Insertion Sort

- Realizing a priority queue with a heap has the advantage that all the methods in the priority queue run in logarithmic time or better
- If we consider our sorting algorithm but this time with a heap based implementation of the queue
- The i^{th} step of the first phase (`insert` phase) now takes $O(\log i)$ time since the heap has i entries after the previous operations have been carried out. The first phase thus take $O(n\log(n))$ operation.
- For the second phase, recall that we only need to remove the root node in the heap, and we then replace it with the rightmost leaf, after which we perform at most $\log(d)$ down heap bubbling to satisfy the heap order property. We thus again have a $O(n\log n)$ running time.

Insertion Sort

- All in all, this leads to the following proposition

Proposition

The heap sort algorithm sorts a sequence S of n elements in $O(n \log n)$ time, assuming two elements of S can be compared in $O(1)$ time.

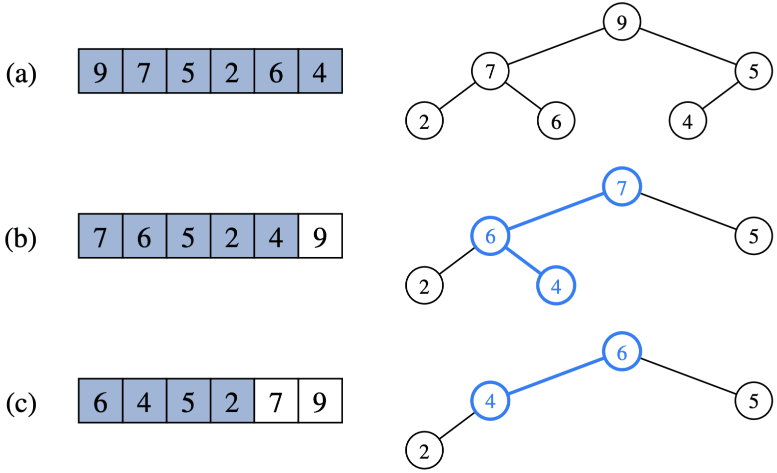
Implementing Heap Sort in place

- If the sequence to be sorted is implemented by means of an array-based sequence, such as ArrayList in Java, we can speed up heap sort and reduce its space requirement by a constant factor by using a portion of the array itself to store the heap.
- In general we say that an algorithm is in place if it uses only a small amount of memory in addition to the sequence storing the objects to be stored

Implementing Heap Sort in place

- This approach can be carrried out through the following steps:
    - We first define the heap to be maximum oriented (each node is now at least as large as its children). at any time during the execution of the algorithm, we use the left portion of the original array S, up to index $i-1$ to store the entries of the heap, and the right portion of S from i to $n-1$ to store the elements of the sequence.
    - In the first phase of the algorithm, we start with an empty heap and move the boundary between the sequence and the heap left to right (at step i, we expand the heap by adding the element at index i)
    - During the second phase, we start with an empty sequence and move the boundary between the heap and the sequence right to left. At step i, we move a maximum elements from the heap and store it at index $n-i$ in the sequence

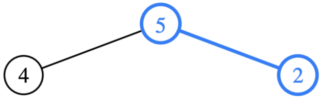
Implementing Heap Sort in place



Implementing Heap Sort in place

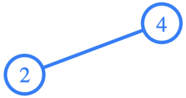
(d)

5	4	2	6	7	9



(e)

4	2	5	6	7	9



(f)

2	4	5	6	7	9