PRIORITY QUEUES

Priorities

- Sometimes the ***First-In, First-Out (FIFO)*** *principle is not enough and the removal is based on some priorities.*
  - ***Ex:*** Air-traffic control center that has to decide which flight to clear for landing from among many approaching the airport. This choice may be influenced by factors such as each plane's distance from the runway, time spent waiting in a holding pattern, or amount of remaining fuel. (By the way some of these values constantly change!!!)
- A new abstract data type known as a *priority queue can be used to handle priority situations.*

Entry ADT

- An entry in a priority queue is simply a key-value pair
- Priority queues store entries to allow for efficient insertion and removal based on keys
- Methods:
  - getKey: returns the key for this entry
  - getValue: returns the value associated with this entry

- As a Java interface:

```
// Interface for a key-value  pair entry
public interface  Entry<K,V>  {
        K getKey();
        V getValue();
  }
```

Priority Queue ADT

- A priority queue stores a collection of entries
- Main methods of the Priority Queue ADT
  - **insert(k, v)**
    inserts an entry with key k and value v
  - **removeMin()**
    removes and returns the entry with smallest key, or null if the priority queue is empty
- Additional methods
  - **min()**
    returns, but does not remove, an entry with smallest key, or null if the priority queue is empty
  - **size(), isEmpty()**
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Example

- A sequence of priority queue methods:

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	{ }

Comparing Keys with Total Order Relations

- Keys in a priority queue can be arbitrary objects on which an order is defined (*must be able to compare keys to each other in a meaningful way)*
- Two distinct entries in a priority queue can have the same key

- Mathematical concept of total order relation $\leq$
  - Comparability property: either $x \leq y$ or $y \leq x$
  - Antisymmetric property: $x \leq y$ and $y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y$ and $y \leq z \Rightarrow x \leq z$

Comparator ADT

- A comparator encapsulates the action of comparing two objects according to a given total order relation
- A generic priority queue uses an auxiliary comparator
- The comparator is external to the keys being compared
- When the priority queue needs to compare two keys, it uses its comparator

- Primary method of the Comparator ADT
- **compare(x, y):** returns an integer i such that
  - **i < 0 if a < b,**
  - **i = 0 if a = b**
  - **i > 0 if a > b**
  - An error occurs if a and b cannot be compared.

Ex: Lexicographic Comparison of 2-D Points

```
/** Comparator for 2D points under the standard
   lexicographic order. */
public class  Lexicographic  implements
   Comparator  {
      int  xa, ya, xb, yb;
public int  compare(Object a, Object b)  throws
   ClassCastException  {
         xa = ((Point2D) a).getX();
         ya = ((Point2D) a).getY();
         xb = ((Point2D) b).getX();
         yb = ((Point2D) b).getY();
         if  (xa != xb)
  return  (xb - xa);
         else
  return  (yb - ya);
     }   }
```

- Point objects:

```
/** Class representing a point in the plane
   with integer coordinates */
 public class  Point2D{
      protected int xc, yc; // coordinates
      public  Point2D(int  x,  int  y)  {
         xc = x;
         yc = y;
    }
      public int  getX()  {
 return  xc;
      }
      public int  getY()  {
 return  yc;
      }
 }
```

Sequence-based Priority Queue

- Implementation with an unsorted list

$$\begin{tikzcd} 4 \arrow[r, dash] & 5 \arrow[r, dash] & 2 \arrow[r, dash] & 3 \arrow[r, dash] & 1 \end{tikzcd}$$

- Implementation with a sorted list

$$\begin{tikzcd} 1 \arrow[r, dash] & 2 \arrow[r, dash] & 3 \arrow[r, dash] & 4 \arrow[r, dash] & 5 \end{tikzcd}$$

Unsorted List Implementation

```
/** An implementation of a priority queue with an unsorted list. */
public class UnsortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
/** primary collection of priority queue entries */
private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>( );
/** Creates an empty priority queue based on the natural ordering of its keys.
*/
public UnsortedPriorityQueue( ) { super( ); }
/** Creates an empty priority queue using the given comparator to order keys. */
public UnsortedPriorityQueue(Comparator<K> comp) { super(comp); }
/** Returns the Position of an entry having minimal key. */
private Position<Entry<K,V>> findMin( ) { // only called when nonempty
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;
 }
```

Unsorted List Implementation, 2

```
/** 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);
    list.addLast(newest);
    return newest;
 }
/** Returns (but does not remove) an entry with minimal key. */
public Entry<K,V> min( ) {
    if (list.isEmpty( )) return null;
     return findMin( ).getElement( );
 }
/** Removes and returns an entry with minimal key.*/
public Entry<K,V> removeMin( ) {
    if (list.isEmpty( )) return null;
    return list.remove(findMin( ));
}
/** Returns the number of items in the priority queue. */
public int size( ) { return list.size( ); } }
```

Sorted List Implementation

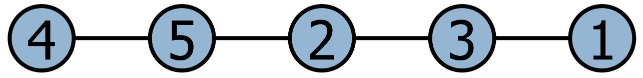
```
/** An implementation of a priority queue with a sorted list. */
public class SortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
/** primary collection of priority queue entries */
private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>( );
/** Creates an empty priority queue based on the natural ordering of its keys.
*/
public SortedPriorityQueue( ) { super( ); }
/** Creates an empty priority queue using the given comparator to order keys. */
public SortedPriorityQueue(Comparator<K> comp) { super(comp); }
/** 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);
    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; }
```

Sorted List Implementation, 2

```
/** Returns (but does not remove) 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( ));
}
/** Returns the number of items in the priority queue. */
public int size( ) { return list.size( ); } }
```

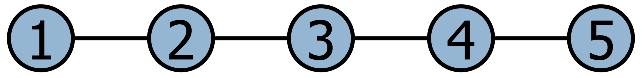
Sequence-based Priority Queue

- Implementation with an unsorted list



- Performance:
  - insert takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin and min take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted list



- Performance:
  - insert takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin and min take $O(1)$ time, since the smallest key is at the beginning

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	$O(n)$	$O(1)$

Worst-case running times of the methods of a priority queue of size n, realized by means of an unsorted and sorted, doubly linked list. The space requirement is O(n).

Priority Queue Sorting

- We can use a priority queue to sort a list of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of removeMin operations
- The running time of this sorting method depends on the priority queue implementation

```
Algorithm PQ-Sort(S, C)
    Input list S, comparator C for the
    elements of S
    Output list S sorted in increasing
    order according to C
    P ← priority queue with
        comparator C
    while ¬S.isEmpty ()
        e ← S.remove(S.first ())
        P.insert (e, ∅)
    while ¬P.isEmpty()
        e ← P.removeMin().getKey()
        S.addLast(e)
```

Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence
- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with n insert operations takes $O(n)$ time
  2. Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

$$1 + 2 + \ldots + n$$

- Selection-sort runs in $O(n^2)$ time

Selection-Sort Example

	Sequence S		Priority Queue P
Input:	(7,4,8,2,5,3,9)		()
Phase 1 $O(1)$			
(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 ($O(n+(n-1)+\cdots+2+1) = n(n+1)/2) =$ **$O(n^2)$**			
(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

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with $\boldsymbol{n}$ insert operations takes time proportional to

     $$1 + 2 + \ldots + \boldsymbol{n}$$

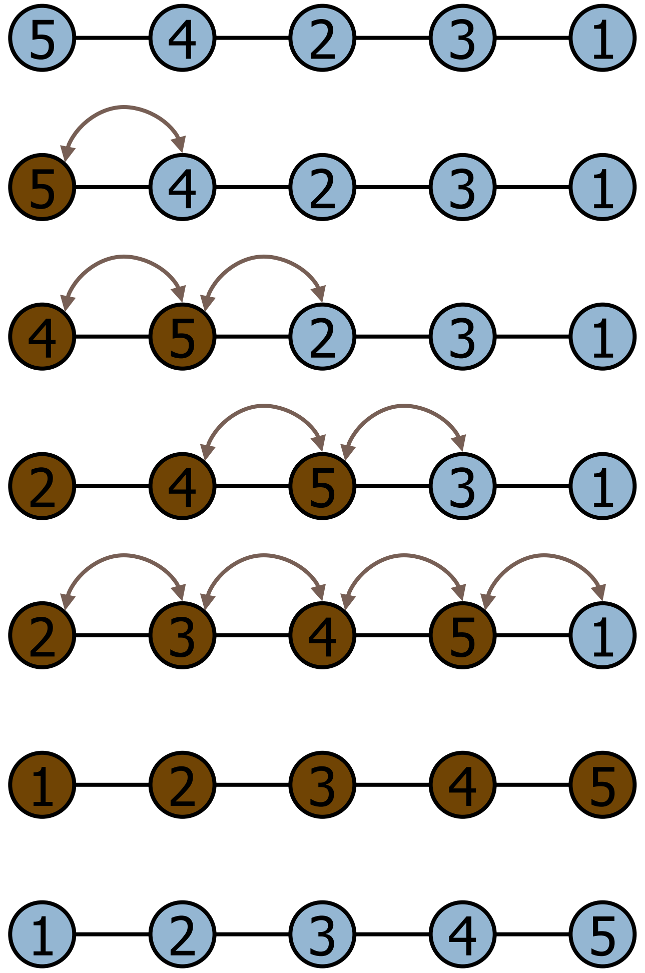
  2. Removing the elements in sorted order from the priority queue with a series of $\boldsymbol{n}$ removeMin operations takes $\boldsymbol{O}(\boldsymbol{n})$ time
- Insertion-sort runs in $\boldsymbol{O}(\boldsymbol{n}^2)$ time

Insertion-Sort Example

	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1 $(O(n+(n-1)+\cdots+2+1)=n(n+1)/2)$ = **O(n^2)**		
(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 O(1)		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

In-place Insertion-Sort

- Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place
- A portion of the input sequence itself serves as the priority queue
- For in-place insertion-sort
  - We keep sorted the initial portion of the sequence
  - We can use swaps instead of modifying the sequence



Recall PQ Sorting

- We use a priority queue
  - Insert the elements with a series of insert operations
  - Remove the elements in sorted order with a series of removeMin operations
- The running time depends on the priority queue implementation:
  - Unsorted sequence gives selection-sort: O(n^2) time
  - Sorted sequence gives insertion-sort: O(n^2) time
- **Can we do better?**

```
Algorithm PQ-Sort(S, C)
    Input sequence S, comparator C
    for the elements of S
    Output sequence S sorted in
    increasing order according to C
    P ← priority queue with
        comparator C
    while ¬S.isEmpty ()
        e ← S.remove (S. first ())
        P.insert (e, e)
    while ¬P.isEmpty()
        e ← P.removeMin().getKey()
        S.addLast(e)
```

Heaps

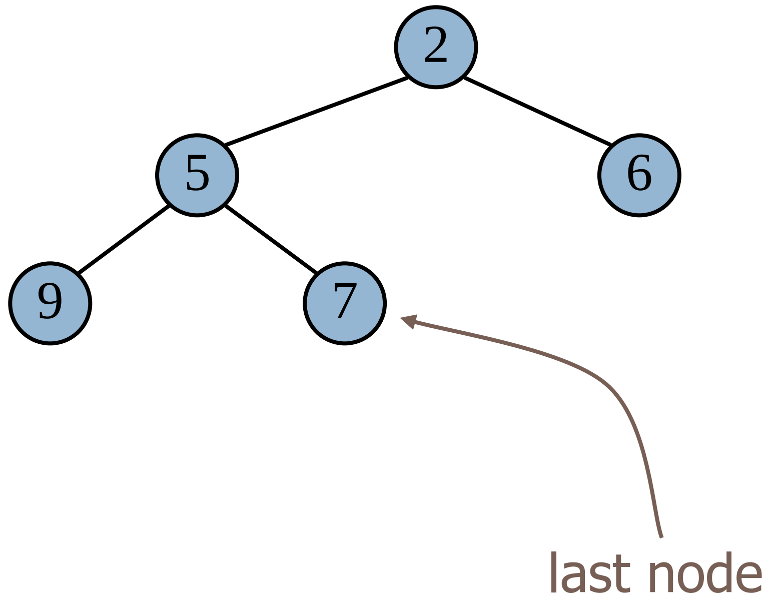
- A more efficient realization of a priority queue is possible with using a data structure called a ***binary heap.***
- ***This data structure allows us to perform both*** insertions and removals in logarithmic time.
- The fundamental way the heap achieves this improvement is to use **the structure of a binary tree to find a compromise between elements being entirely unsorted and perfectly sorted.**

Heaps

A full **binary tree** (sometimes **proper binary tree** or 2-**tree**) is a **tree** in which every node other than the leaves has two children. A **complete binary tree** is a **binary tree** in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

- A heap is a binary tree storing keys at its nodes and satisfying the following properties:
- **Heap-Order: for every internal node v other than the root,**
  $\boldsymbol{key(v) \geq key(parent(v))}$
- Complete Binary Tree: let $\boldsymbol{h}$ be the height of the heap
  - for $\boldsymbol{i} = 0, \dots, \boldsymbol{h} - 1$, there are $2^{\boldsymbol{i}}$ nodes of depth $\boldsymbol{i}$
  - at depth $\boldsymbol{h} - 1$, the internal nodes are to the left of the external nodes

- The last node of a heap is the rightmost node of maximum depth



For the sake of efficiency, as will become clear later, 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; it must be what we term complete.

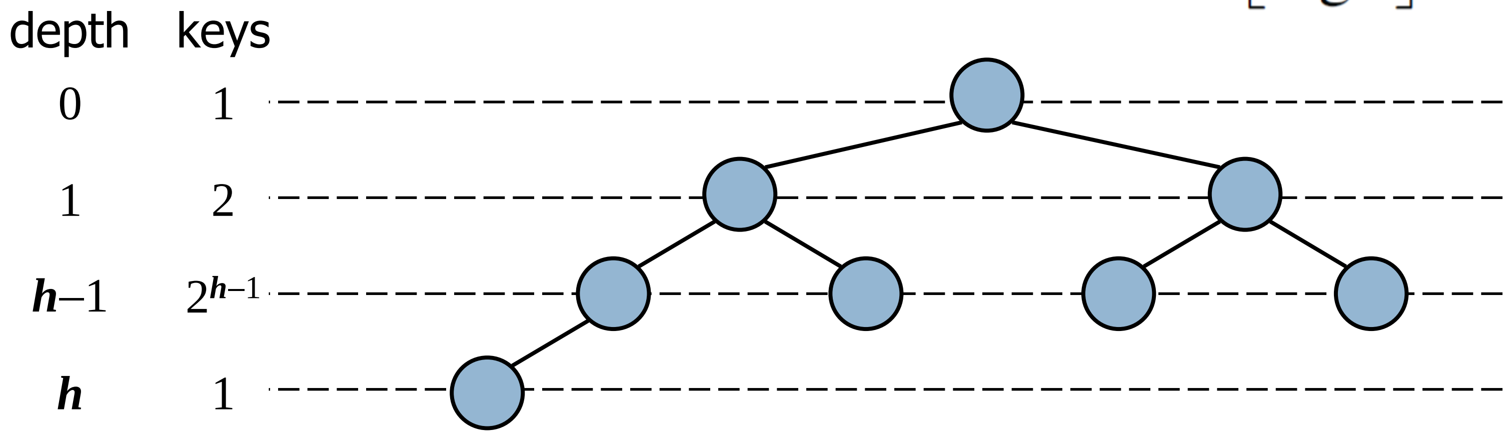
Height of a Heap

- Theorem: A heap storing n keys has height $\lfloor \log n \rfloor$ *Proposition 9.2*

Proof: (we apply the complete binary tree property)

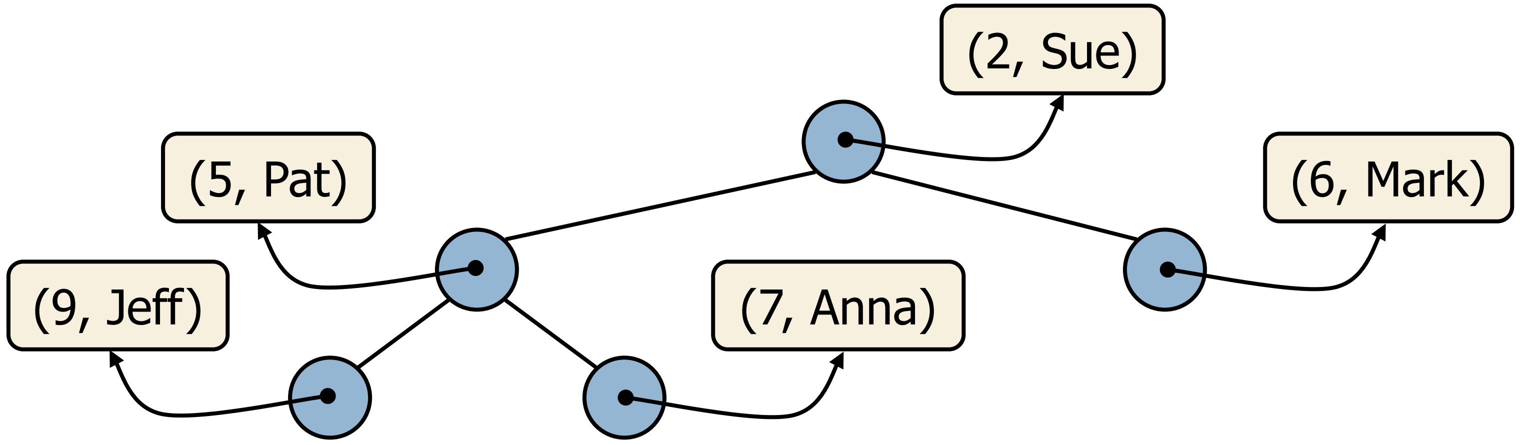
- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
- Thus, $n \geq 2^h$, i.e., $h \leq \log n$

$$h = \lfloor \log n \rfloor$$



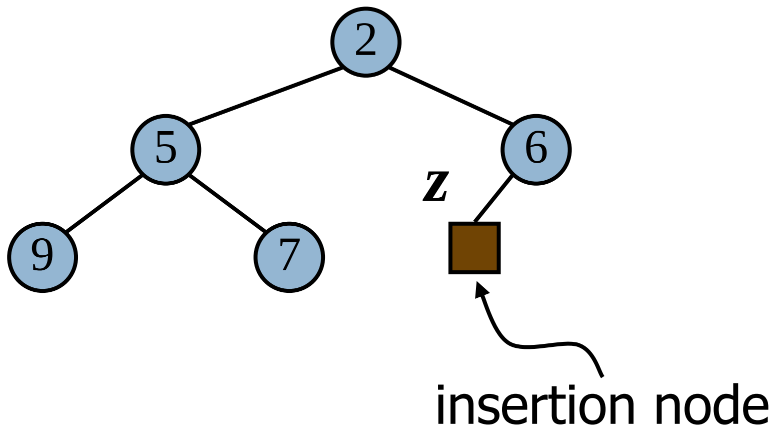
Heaps and Priority Queues

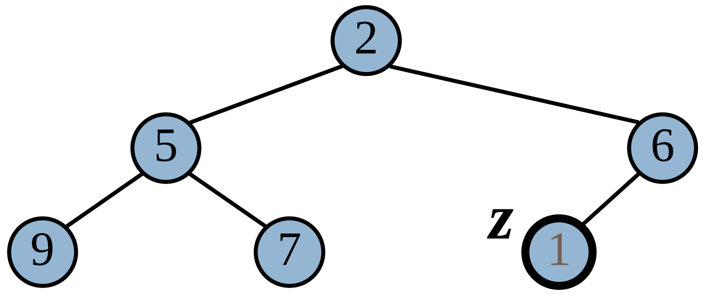
- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node



Insertion into a Heap

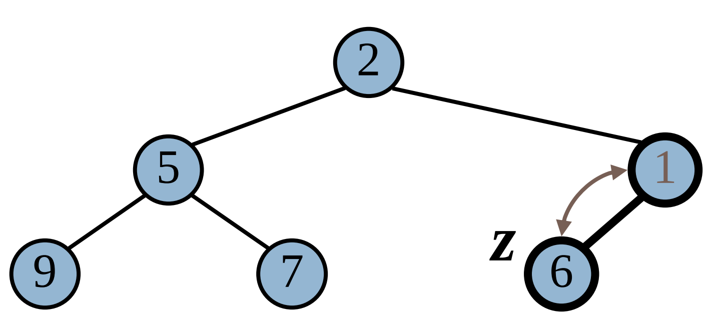
- Method insertItem of the priority queue ADT corresponds to the insertion of a key ***k*** to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node ***z*** (the new last node)
  - Store ***k*** at ***z***
  - Restore the heap-order property (discussed next)

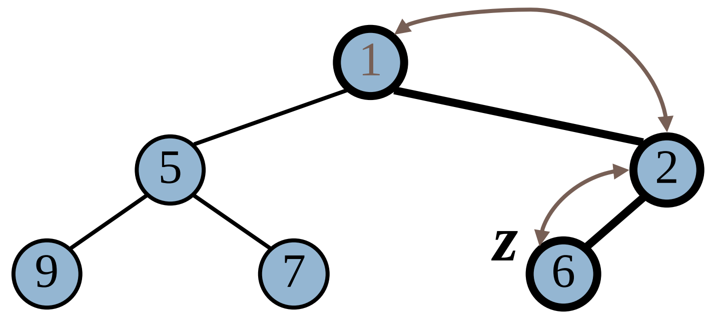




Upheap

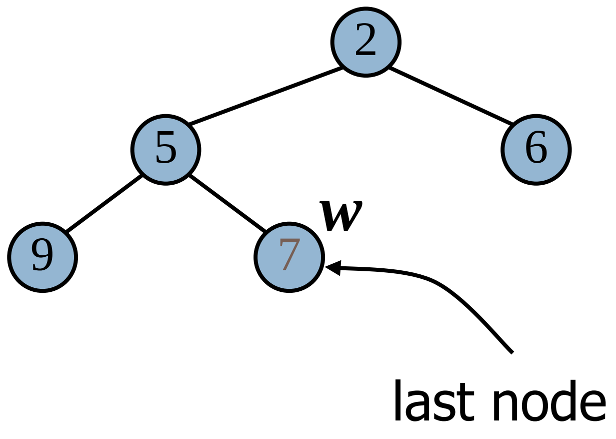
- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time

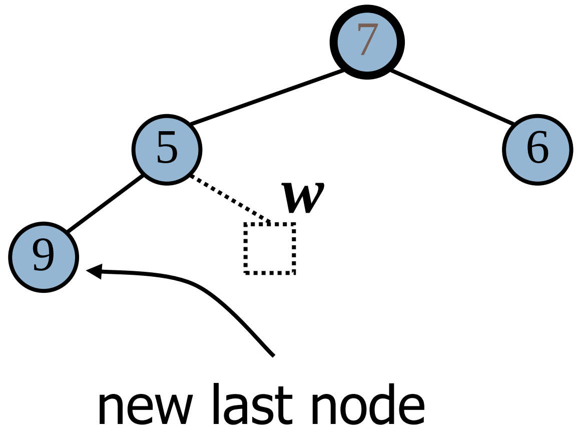




Removal from a Heap

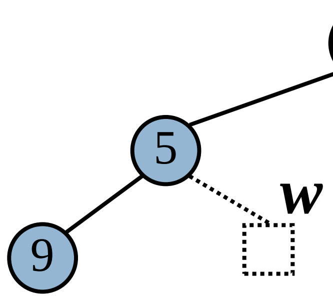
- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node *w*
  - Remove *w*
  - Restore the heap-order property (discussed next)

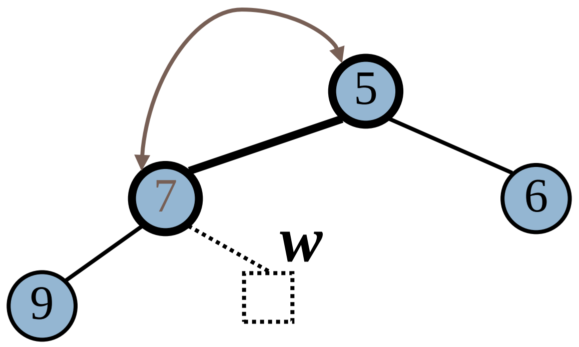




Downheap

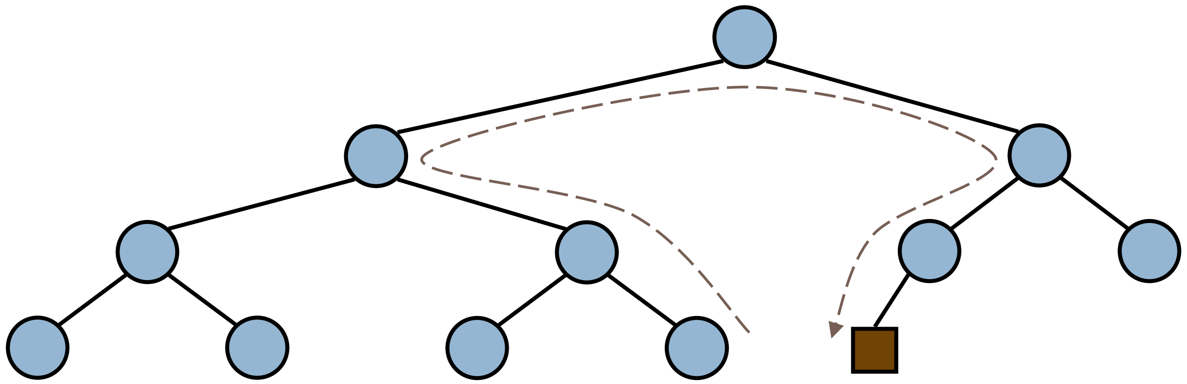
- After replacing the root key with the key ***k*** of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key ***k*** along a downward path from the root
- Algorithm terminates when key ***k*** reaches a leaf or a node whose children have keys greater than or equal to ***k***
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time

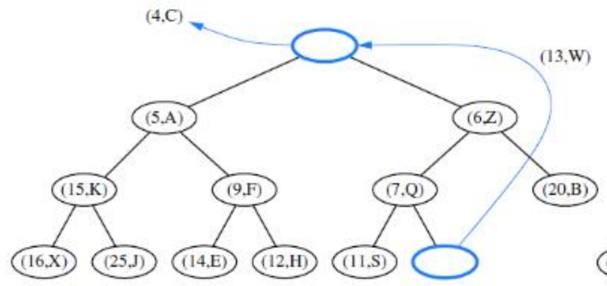




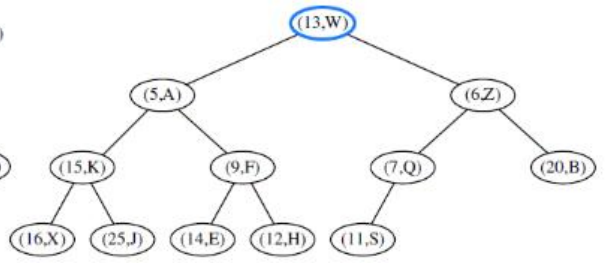
Updating the Last Node

- The insertion node can be found by traversing a path of $O(\log n)$ nodes
  - Go up until a left child or the root is reached
  - If a left child is reached, go to the right child
  - Go down left until a leaf is reached
- Similar algorithm for updating the last node after a removal



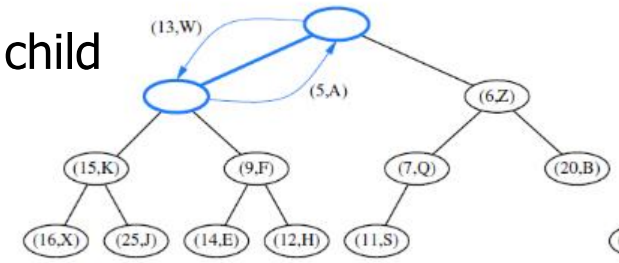


(a)

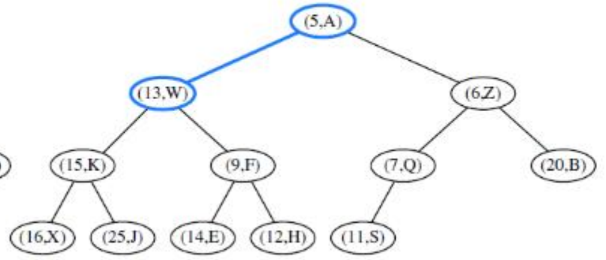


(b)

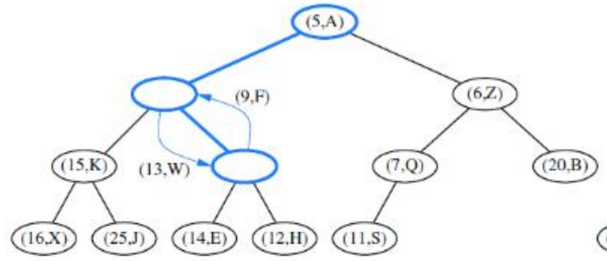
Smallest child



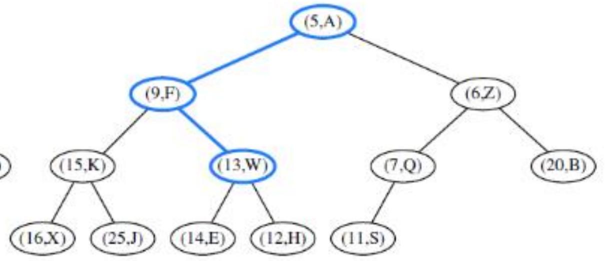
(c)



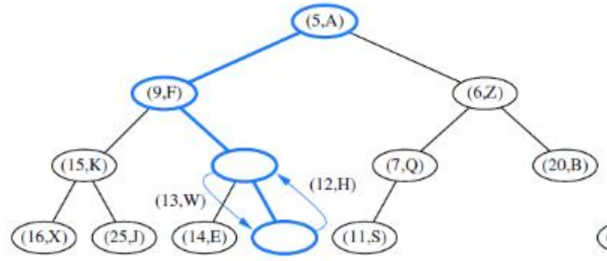
(d)



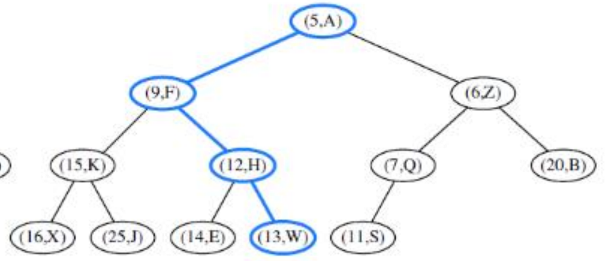
(e)



(f)

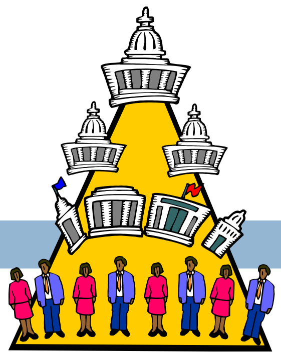


(g)



(h)

Heap-Sort

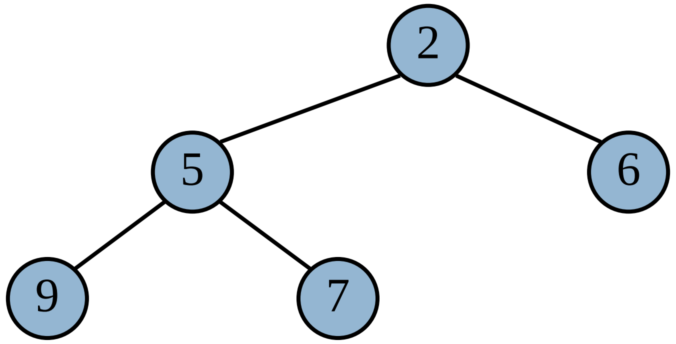


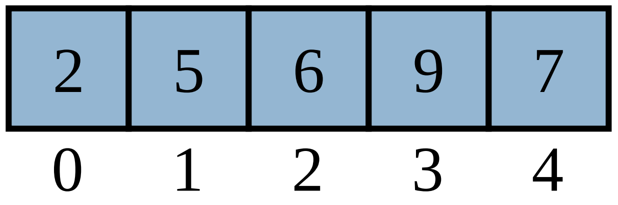
- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods insert and removeMin take $O(\log n)$ time
  - methods size, isEmpty, and min take time $O(1)$ time

- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

Array-Based Heap Implementation

- We can represent a heap with n keys by means of an array of length n
- For the node at rank i
  - the left child is at rank $2i + 1$
  - the right child is at rank $2i + 2$
- Links between nodes are not explicitly stored
- Operation add corresponds to inserting at rank $n + 1$
- Operation remove_min corresponds to removing at rank n
- Yields in-place heap-sort





Java Implementation

```
/** 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(); }
  /** 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
    }
  }
```

Java Implementation

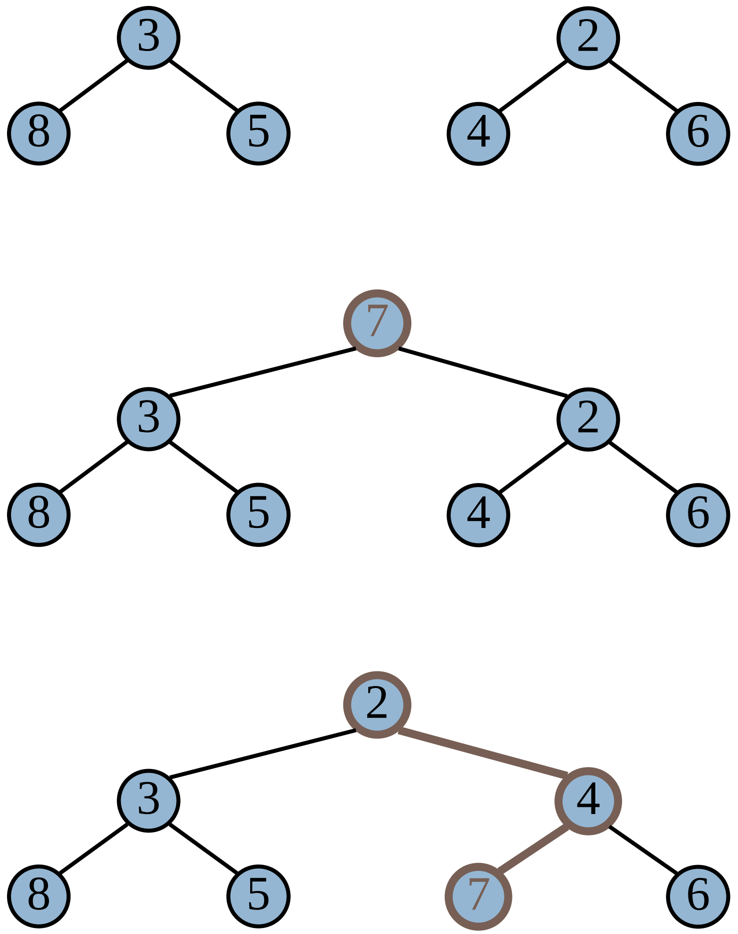
```
/** Moves the entry at index j lower, if necessary, to restore the heap property. */
protected void downheap(int j) {
  while (hasLeft(j)) {                          // continue to bottom (or break statement)
    int leftIndex = left(j);
    int smallChildIndex = leftIndex;                // although 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(); }
/** Returns (but does not remove) an entry with minimal key (if any). */
public Entry<K,V> min() {
  if (heap.isEmpty()) return null;
  return heap.get(0);
}
```

Java Implementation, 3

```
  /** 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;
  }
}
```

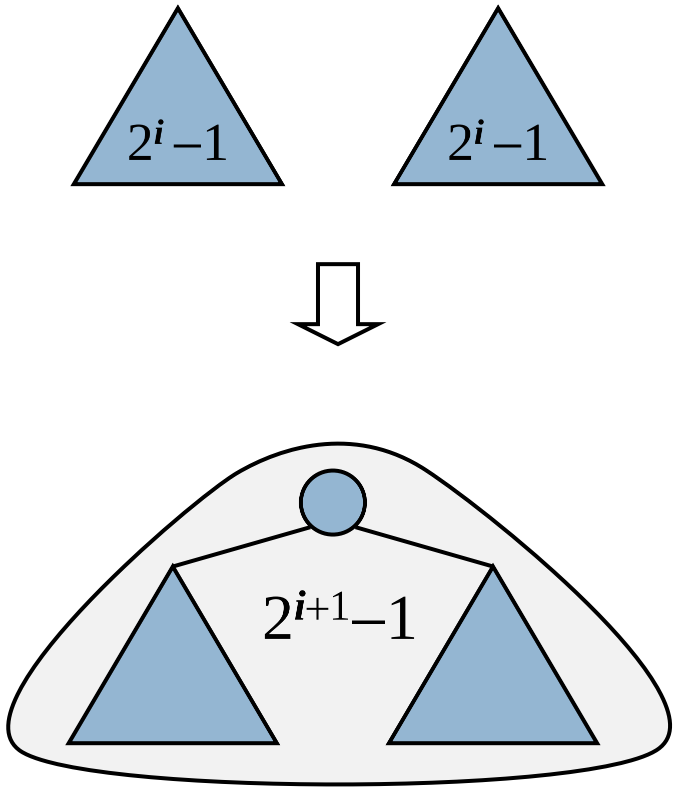
Merging Two Heaps

- We are given two two heaps and a key ***k***
- We create a new heap with the root node storing ***k*** and with the two heaps as subtrees
- We perform downheap to restore the heap-order property

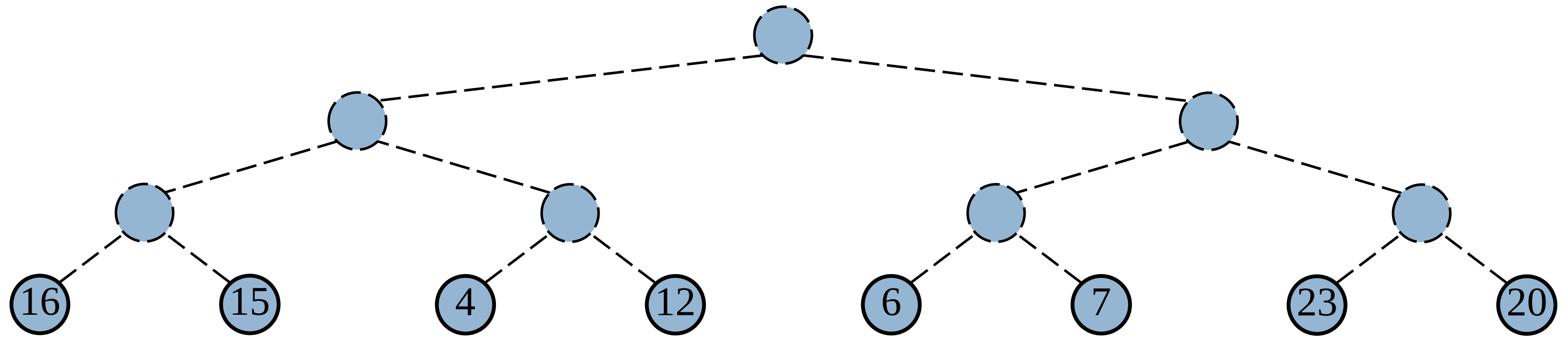


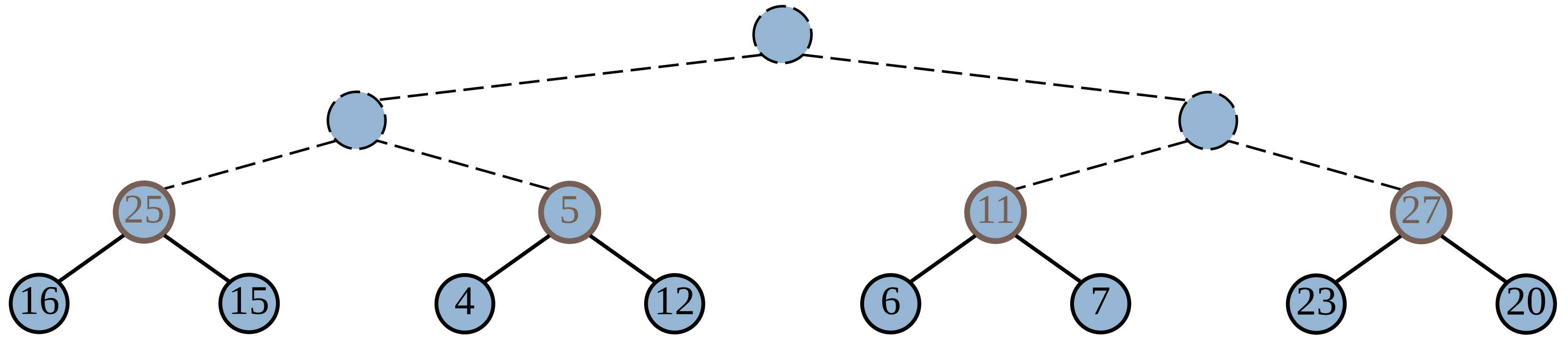
Bottom-up Heap Construction

- We can construct a heap storing $\boldsymbol{n}$ given keys in using a bottom-up construction with $\log \boldsymbol{n}$ phases
- In phase $\boldsymbol{i}$, pairs of heaps with $2^{\boldsymbol{i}}-1$ keys are merged into heaps with $2^{\boldsymbol{i}+1}-1$ keys

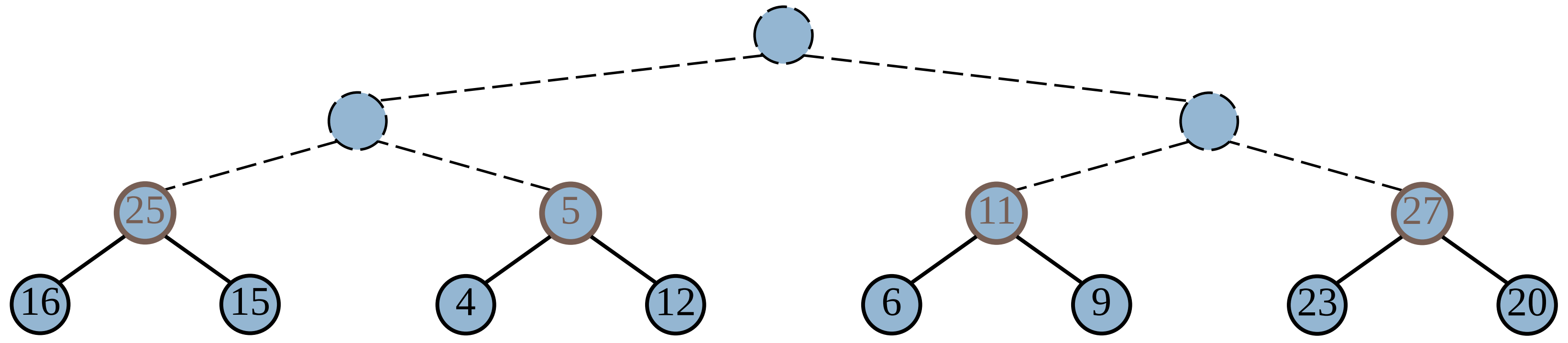


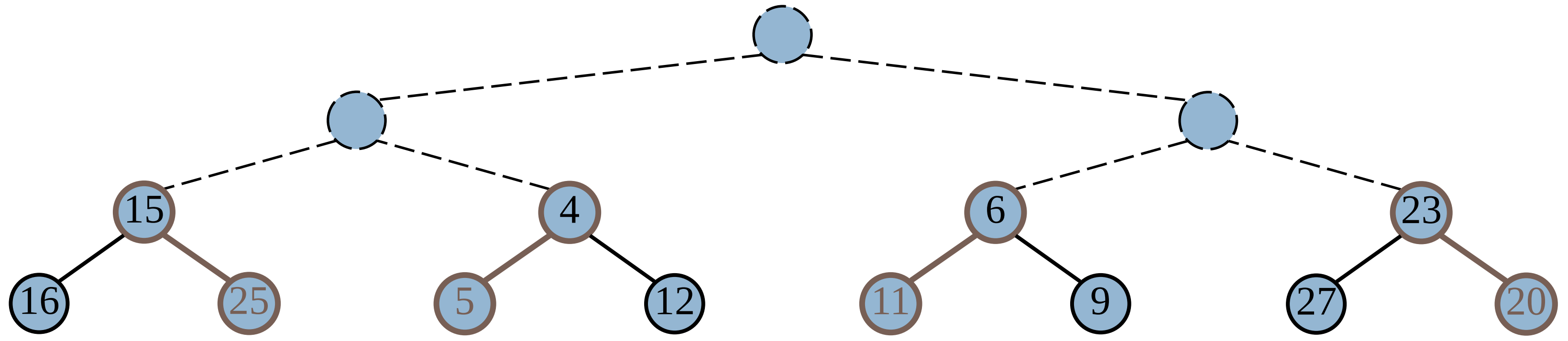
Example



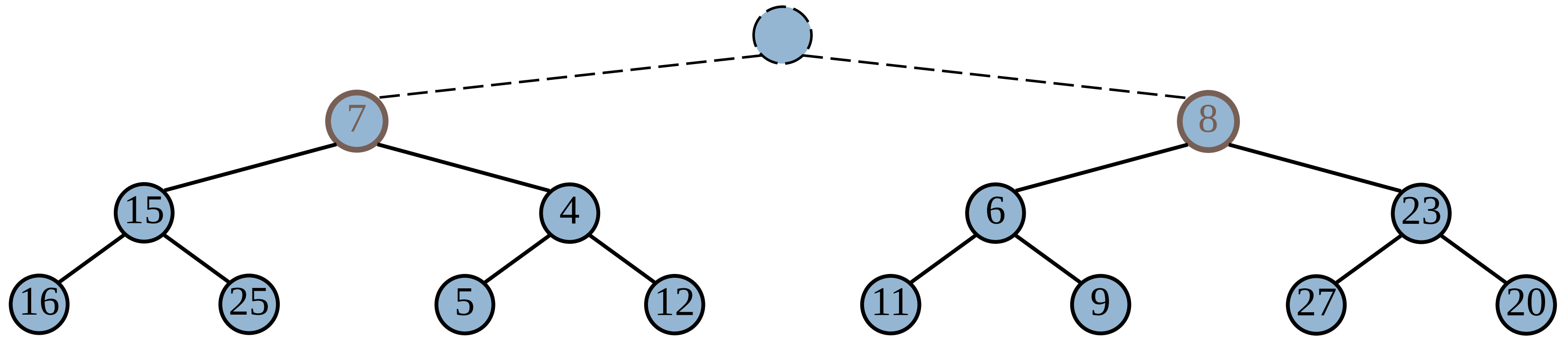


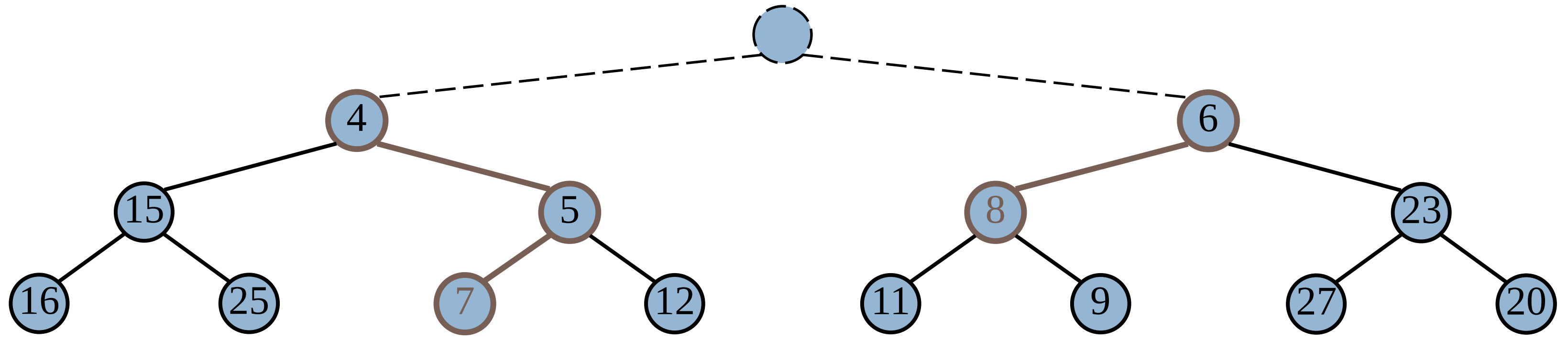
Example (contd.)



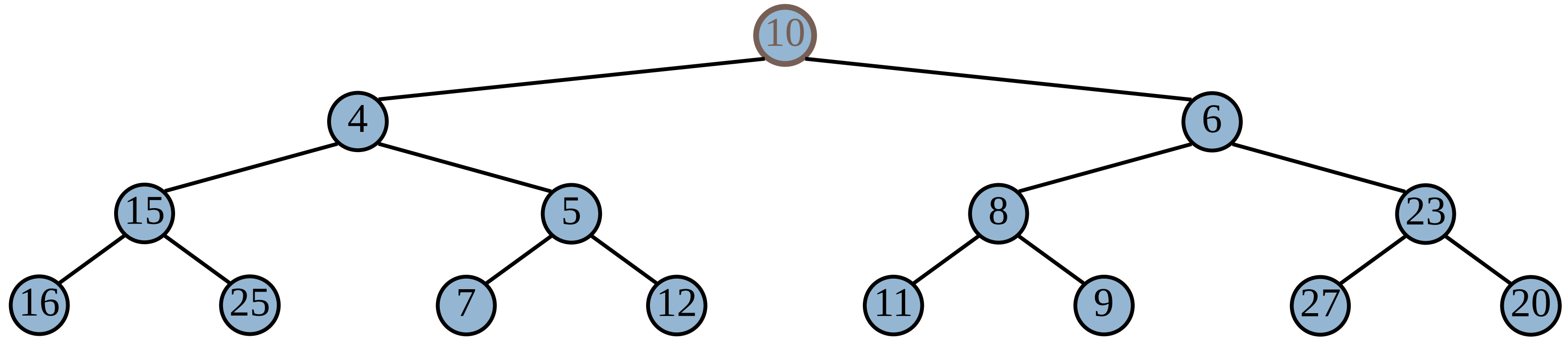


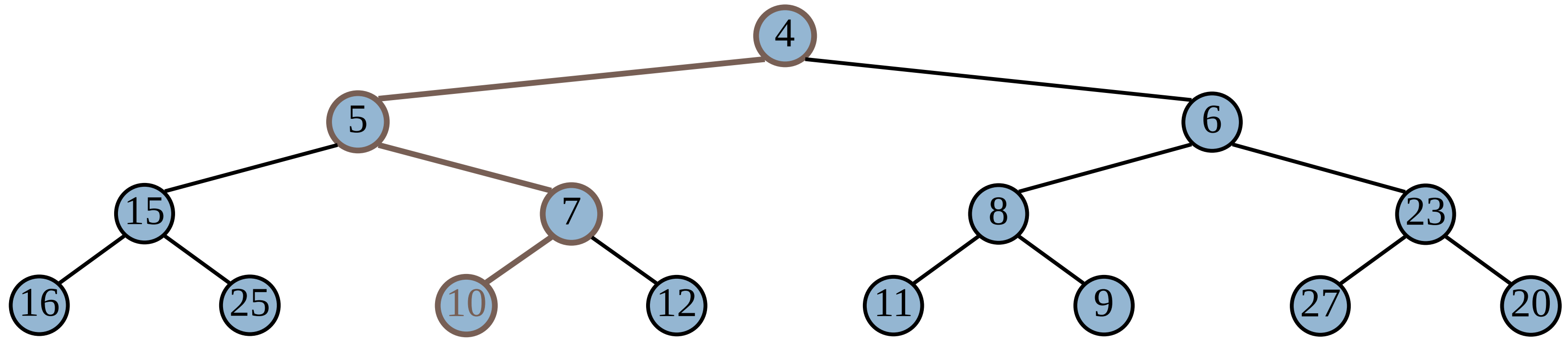
Example (contd.)





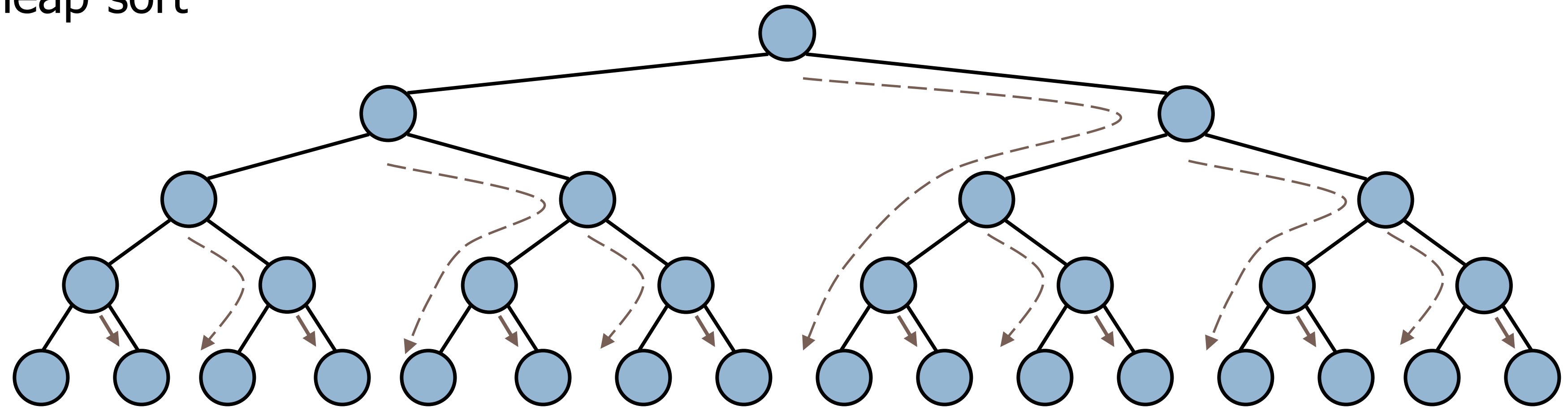
Example (end)



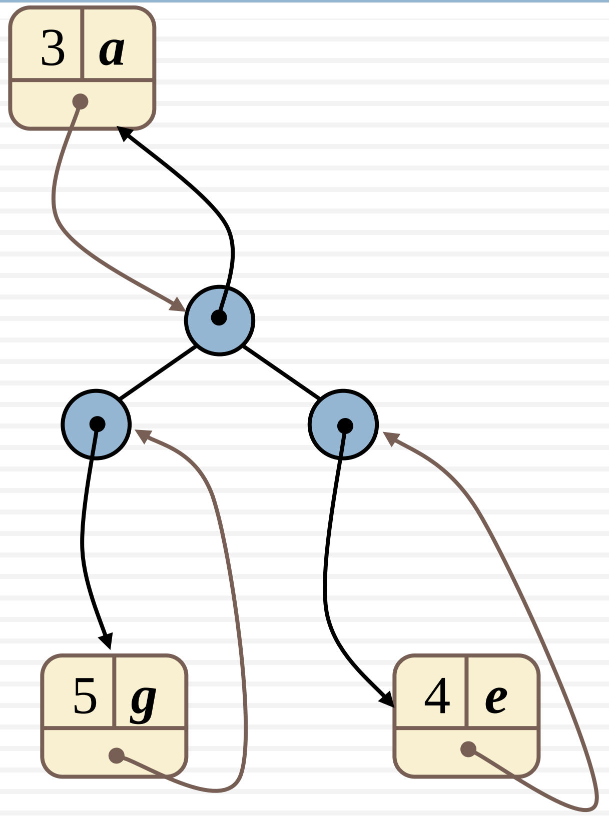


Analysis

- We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path)
- Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy paths is $O(n)$
- Thus, bottom-up heap construction runs in $O(n)$ time
- Bottom-up heap construction is faster than n successive insertions and speeds up the first phase of heap-sort



Adaptable Priority Queues



Entry and Priority Queue ADTs

- An entry stores a (key, value) pair
- Entry ADT methods:
  - getKey(): returns the key associated with this entry
  - getValue(): returns the value paired with the key associated with this entry

- Priority Queue ADT:
  - insert(k, x)
    inserts an entry with key k and value x
  - removeMin()
    removes and returns the entry with smallest key
  - min()
    returns, but does not remove, an entry with smallest key
  - size(), isEmpty()

Example

- Online trading system where orders to purchase and sell a stock are stored in two priority queues (one for sell orders and one for buy orders) as (p,s) entries:
  - The key, p, of an order is the price
  - The value, s, for an entry is the number of shares
  - A buy order (p,s) is executed when a sell order (p’,s’) with price p’ $\leq$p is added (the execution is complete if s’ $\geq$s)
  - A sell order (p,s) is executed when a buy order (p’,s’) with price p’ $\geq$p is added (the execution is complete if s’ $\geq$s)
- What if someone wishes to cancel their order before it executes?
- What if someone wishes to update the price or number of shares for their order?

Methods of the Adaptable Priority Queue ADT

- remove(e): Remove from P and return entry e.
- replaceKey(e,k): Replace with k and return the key of entry e of P; an error condition occurs if k is invalid (that is, k cannot be compared with other keys).
- replaceValue(e,v): Replace with v and return the value of entry e of P.

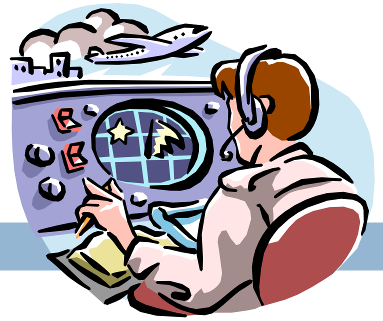
Example

Operation	***Output***	P
insert(5,A)	e_1	(5,A)
insert(3,B)	e_2	(3,B),(5,A)
insert(7,C)	e_3	(3,B),(5,A),(7,C)
min()	e_2	(3,B),(5,A),(7,C)
key(e_2)	3	(3,B),(5,A),(7,C)
remove(e_1)	e_1	(3,B),(7,C)
replaceKey(e_2,9)	3	(7,C),(9,B)
replaceValue(e_3,D)	C	(7,D),(9,B)
remove(e_2)	e_2	(7,D)

Locating Entries

- In order to implement the operations remove(e), replaceKey(e,k), and replaceValue(e,v), we need fast ways of locating an entry e in a priority queue.
- We can always just search the entire data structure to find an entry e, but there are better ways for locating entries.

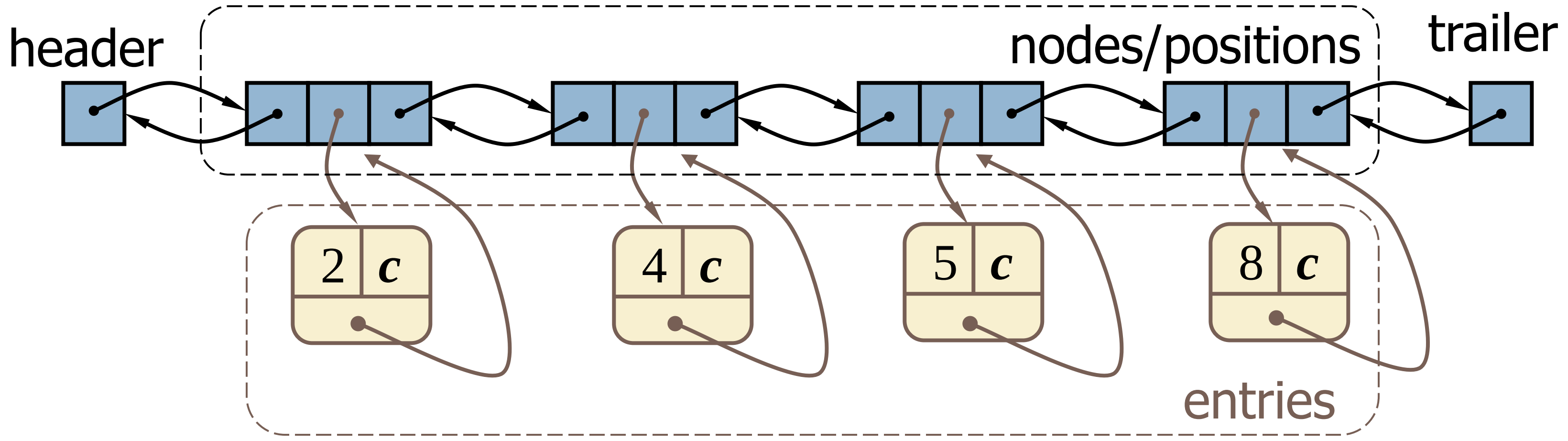
Location-Aware Entries



- A location-aware entry identifies and tracks the location of its (key, value) object within a data structure
- Intuitive notion:
  - Coat claim check
  - Valet claim ticket
  - Reservation number
- Main idea:
  - Since entries are created and returned from the data structure itself, it can return location-aware entries, thereby making future updates easier

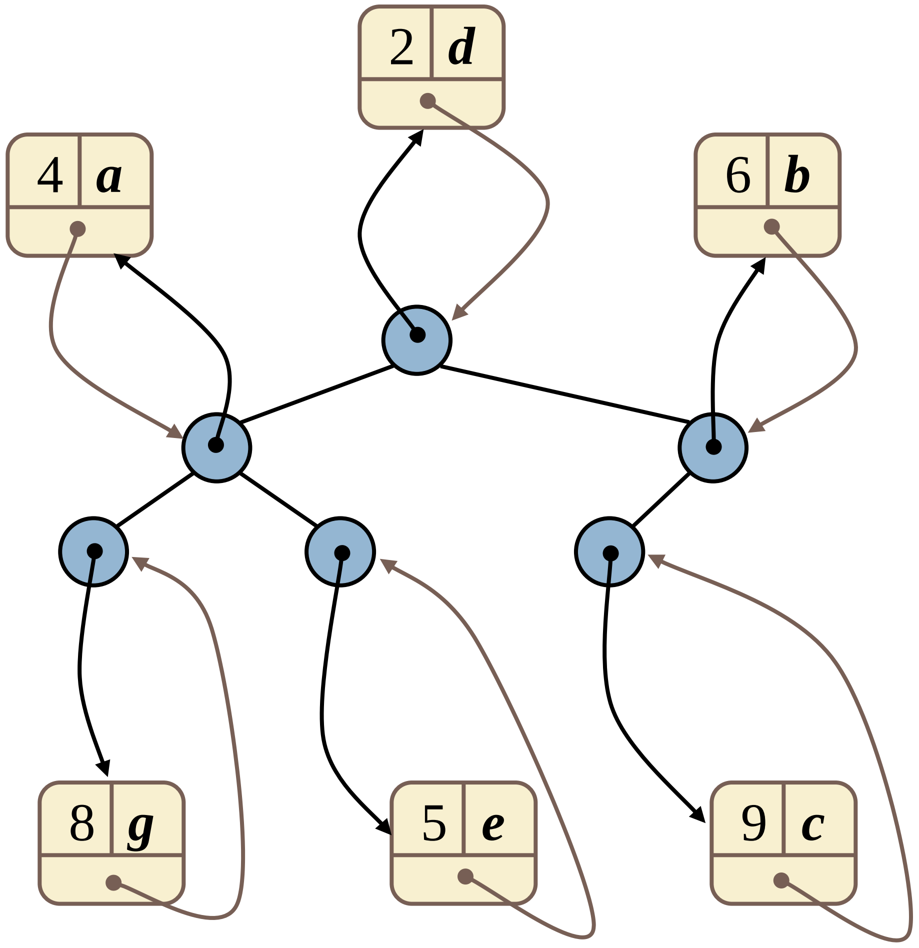
List Implementation

- A location-aware list entry is an object storing
  - key
  - value
  - position (or rank) of the item in the list
- In turn, the position (or array cell) stores the entry
- Back pointers (or ranks) are updated during swaps



Heap Implementation

- A location-aware heap entry is an object storing
  - key
  - value
  - position of the entry in the underlying heap
- In turn, each heap position stores an entry
- Back pointers are updated during entry swaps



Performance

- Improved times thanks to location-aware entries are highlighted in red

Method	**Unsorted List**	**Sorted List**	**Heap**
size, isEmpty	$O(1)$	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$	$O(\log n)$
min	$O(n)$	$O(1)$	$O(1)$
removeMin	$O(n)$	$O(1)$	$O(\log n)$
remove	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(\log n)}$
replaceKey	$\boldsymbol{O(1)}$	$O(n)$	$\boldsymbol{O(\log n)}$
replaceValue	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$

Java Implementation

```
/** An implementation of an adaptable priority queue using an array-based heap. */
public class HeapAdaptablePriorityQueue<K,V> extends HeapPriorityQueue<K,V>
                                       implements AdaptablePriorityQueue<K,V> {

  //---------------- nested AdaptablePQEntry class ----------------
  /** Extension of the PQEntry to include location information. */
  protected static class AdaptablePQEntry<K,V> extends PQEntry<K,V> {
    private int index;                        // entry's current index within the heap
    public AdaptablePQEntry(K key, V value, int j) {
      super(key, value);                      // this sets the key and value
      index = j;                              // this sets the new field
    }
    public int getIndex( ) { return index; }
    public void setIndex(int j) { index = j; }
  } //----------- end of nested AdaptablePQEntry class -----------

  /** Creates an empty adaptable priority queue using natural ordering of keys. */
  public HeapAdaptablePriorityQueue( ) { super( ); }
  /** Creates an empty adaptable priority queue using the given comparator. */
  public HeapAdaptablePriorityQueue(Comparator<K> comp) { super(comp);}

```

Java Implementation, 2

```
// protected utilites
/** Validates an entry to ensure it is location-aware. */
protected AdaptablePQEntry<K,V> validate(Entry<K,V> entry)
                                throws IllegalArgumentException {
  if (!(entry instanceof AdaptablePQEntry))
    throw new IllegalArgumentException("Invalid entry");
  AdaptablePQEntry<K,V> locator = (AdaptablePQEntry<K,V>) entry;   // safe
  int j = locator.getIndex();
  if (j >= heap.size() || heap.get(j) != locator)
    throw new IllegalArgumentException("Invalid entry");
  return locator;
}

/** Exchanges the entries at indices i and j of the array list. */
protected void swap(int i, int j) {
  super.swap(i,j);                                      // perform the swap
  ((AdaptablePQEntry<K,V>) heap.get(i)).setIndex(i);    // reset entry's index
  ((AdaptablePQEntry<K,V>) heap.get(j)).setIndex(j);    // reset entry's index
}
```

Java Implementation, 3

```
/** Restores the heap property by moving the entry at index j upward/downward.*/
protected void bubble(int j) {
  if (j > 0 && compare(heap.get(j), heap.get(parent(j))) < 0)
    upheap(j);
  else
    downheap(j);                          // although it might not need to move
}

/** Inserts a key-value pair and returns the entry created. */
public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
  checkKey(key);                          // might throw an exception
  Entry<K,V> newest = new AdaptablePQEntry<>(key, value, heap.size());
  heap.add(newest);                       // add to the end of the list
  upheap(heap.size() − 1);                // upheap newly added entry
  return newest;
}
```

Java Implementation, 4

```
/** Removes the given entry from the priority queue. */
public void remove(Entry<K,V> entry) throws IllegalArgumentException {
  AdaptablePQEntry<K,V> locator = validate(entry);
  int j = locator.getIndex();
  if (j == heap.size() − 1)                     // entry is at last position
    heap.remove(heap.size() − 1);               // so just remove it
  else {
    swap(j, heap.size() − 1);                   // swap entry to last position
    heap.remove(heap.size() − 1);               // then remove it
    bubble(j);                                  // and fix entry displaced by the swap
  }
}

/** Replaces the key of an entry. */
public void replaceKey(Entry<K,V> entry, K key)
                        throws IllegalArgumentException {
  AdaptablePQEntry<K,V> locator = validate(entry);
  checkKey(key);                                // might throw an exception
  locator.setKey(key);                          // method inherited from PQEntry
  bubble(locator.getIndex());                   // with new key, may need to move entry
}

/** Replaces the value of an entry. */
public void replaceValue(Entry<K,V> entry, V value)
                         throws IllegalArgumentException {
  AdaptablePQEntry<K,V> locator = validate(entry);
  locator.setValue(value);                      // method inherited from PQEntry
}
```