Video 1. Priority Queues: list implementations

priority – key associated with an element that establishes its priority (numeric or any comparable type)

comparison rule $\leq$ (define total order relation)

comparability: $k_1 \leq k_2$ or $k_2 \leq k_1$

anti-symmetry: $k_1 \leq k_2$, $k_2 \leq k_1 \Rightarrow k_1 = k_2$

transitivity: $k_1 \leq k_2$, $k_2 \leq k_3 \Rightarrow k_1 \leq k_3$

reflexivity: $k \leq k$

minimal key $k_{min} \leq k$, k – any key in the set

a.compareTo(b) or compare(a,b):

Negative $i<0 \Rightarrow a<b$

Zero $i=0 \Rightarrow a=b$

Positive $i>0 \Rightarrow a>b$

insert(*k*, *v*)	creates an entry with key *k* and value *v*, (*k*, *v*), in the priority queue
min()	returns (does not remove) an entry (*k*, *v*) with minimal key (**null** if empty)
removeMin()	removes and returns an entry (*k*, *v*) with minimal key (**null** if empty)
size()	returns the number of entries
isEmpty()	returns **true** if empty, **false** otherwise

Priority Queue ADT (Key, Value)

→ insertion at arbitrary positions

→ removal of element with first priority

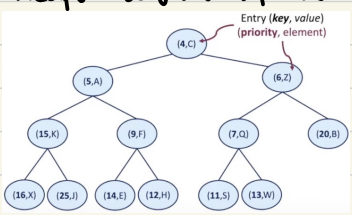
```
public class AbstractPriorityQueue<K,V> implements PriorityQueue<K,V> {
  protected static class PQEntry<K,V> implements Entry<K,V> {        // entry of a priority queue
    private K k;                    // key, the priority
    private V v;                    // value, the element
    public PQEntry(K key, V value) {
      k = key;
      v = value;
    }
    public K getKey() { return k; }
    public V getValue() { return v; }
    protected void setKey(K key) { k = key; }
    protected void setValue(V value) { v = value; }
  }
  private Comparator<K> comp;       // comparator defining the ordering of keys in the Priority Queue
  protected AbstractPriorityQueue(Comparator<K> c) { comp = c; }
  protected AbstractPriorityQueue() { this(new DefaultComparator<K>()); }
  protected int compare(Entry<K, V> a, Entry<K, V> b) {
    return comp.compare(a.getKey(), b.getKey());
  }
  protected boolean checkKey(K key) throws IllegalArgumentException {
    try {
      return (comp.compare(key, key) == 0);         // check if key can be compared to itself
    } catch (ClassCastException e) {
      throw new IllegalArgumentException("Incompatible key");
    }
  }
}
```

```
public interface Entry<K,V> {
    K getKey();                     // key is the priority
    V getValue();                   // value is the element
}

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();
}
```

	Unsorted	Sorted
insert(*k*,*v*)	$O(1)$	$O(n)$
min( )	$O(n)$	$O(1)$
removeMin( )	$O(n)$	$O(1)$
size( )	$O(1)$	$O(1)$
isEmpty( )	$O(1)$	$O(1)$

Video 2. Binary heaps: definition, insertion, deletion



heap – binary tree storing entries at its positions (nodes)

structural property: complete binary tree

- all levels except last are full
- leftmost position of nodes in last level

relational property: heap-order

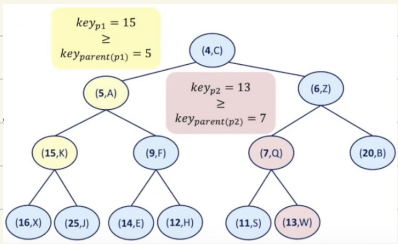
$key_p \geq key_{parent(p)}$ (min heap)

height $h = \lfloor \log_2 n \rfloor$

insert(k,v) → first insert, then sort

up-heap bubbling: $O(\log_2 n)$

(insert last and sort up with swaps



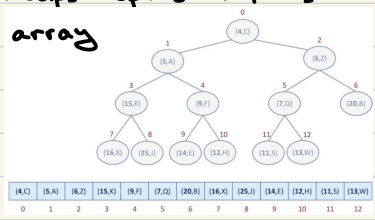
$O(\log_2 n)$

removeMin → remove top element and swap with last one

down-heap bubbling – compare to child with smaller key and swap

Video 3. Binary heaps: representations

array



breadth-first traversal

node: index p left: 2p+1 right: 2p+2

array: internal nodes | external nodes

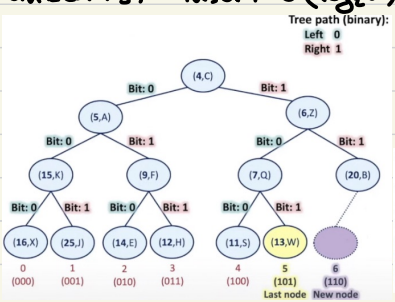
level nodes kept together $\lceil \frac{n-1}{2} \rceil$ → first external

insert: $O(\log_2 n)$ removeMin: $O(1)$

linked-list: insert $O(\log_2 n)$ removeMin $O(\log_2 n)$ → traversal

last: $n - 2^h$ where $h = \lfloor \log_2 n \rfloor$

new: $n + 1 - 2^h$



Space complexity $O(n)$

Time complexity

Method	List	Sorted List	Heap	Motivation (heap with height $O(\log_2 n)$)
min( )	$O(n)$	$O(1)$	$\boldsymbol{O(1)}$	Root contains minimal key.
removeMin( )	$O(n)$	$O(1)$	$\boldsymbol{O(\log_2 n)^*}$	Down-heap bubbling performs $O(\log_2 n)$ swaps.
insert(k, v)	$O(1)$	$O(n)$	$\boldsymbol{O(\log_2 n)^*}$	Up-heap bubbling performs $O(\log_2 n)$ swaps.
size	$O(1)$	$O(1)$	$\boldsymbol{O(1)}$	Size is stored by an instance variable.
isEmpty	$O(1)$	$O(1)$	$\boldsymbol{O(1)}$	Checks size variable.

Video 4. Binary heaps: construction, adaptable PQ

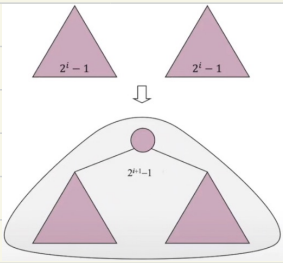
keys not known →

empty heap

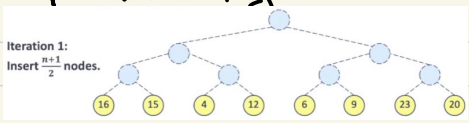
$O(n \log_2 n)$

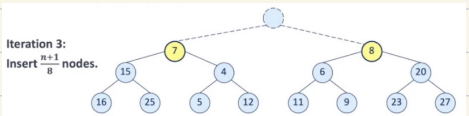
known beforehand →

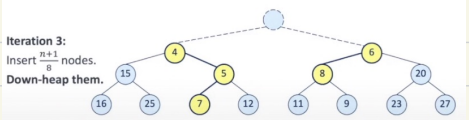
bottom-up heap merging

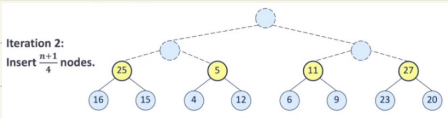


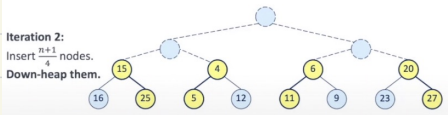
heap with 15 keys:

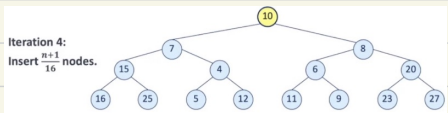


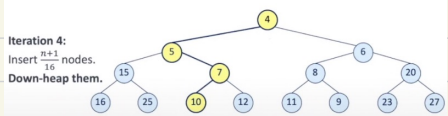












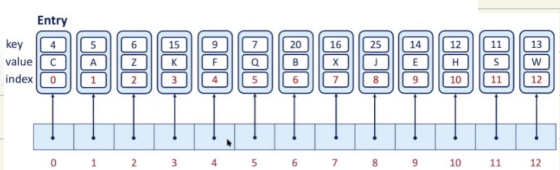
iteration i: insert $\frac{n+1}{2^i}$ nodes

$$\sum_{i=0}^{h} \frac{n+1}{2^i} \cdot i = (n+1) \sum_{i=0}^{h} \frac{i}{2^i} = 2n+2 \quad O(n)$$

```
/**
 * Constructor of array-based heap with keys ks and values vs.
 */
public HeapPriorityQueue(K[] ks, V[] vs) {
   super();                                          // initialize array variable heap
   for (int j = 0; j < Math.min(ks.length, vs.length); j++)   // iterate over keys and values
     heap.add(new PQEntry<>(keys[j], values[j]));    // add all entries to the heap
   heapify();                                        // call heapify to bottom-up restore heap-order
}
/**
 * Heapify: Restores heap-order property in a bottom-up fashion.
 * Starting from the parent of the rightmost position (last level doesn't need
 * down-heap bubbling), and traverses backwards through all indices up to 0 (root).
 * For each node, performs down-heap bubbling to restore the heap-order property.
 */
protected void heapify() {
   int startIndex = parent(size()-1);
   for (int j = startIndex; j >= 0; j--)
      downheap(j);
}
```

Adaptable PQ ADT:

remove(*e*)	Removes entry *e* from the priority queue (error if *e* is invalid; e.g. not in PQ anymore)
replaceKey(*e*,*k*)	Replaces the key of entry *e* with *k* (error if *e* is invalid; e.g. not in PQ anymore)
replaceValue(*e*,*v*)	Replaces the value of entry *e* with *v* (error if *e* is invalid; e.g. not in PQ anymore)
min( )	
removeMin( )	
insert(*k*, *v*)	
size( )	
isEmpty ( )	



Space complexity	$O(n)$			
Time complexity				
Method	**List**	**Sorted List**	**Heap**	**Adaptable Heap**
min( )	$O(n)$	$O(1)$	$O(1)$	$O(1)$
removeMin( )	$O(n)$	$O(1)$	$O(\log_2 n)^*$	$O(\log_2 n)^*$
insert(*k*, *v*)	$O(1)$	$O(n)$	$O(\log_2 n)^*$	$O(\log_2 n)^*$
size( )	$O(1)$	$O(1)$	$O(1)$	$O(1)$
isEmpty( )	$O(1)$	$O(1)$	$O(1)$	$O(1)$
remove( )	-	-	-	$O(\log_2 n)^*$
replaceKey(*e*, *k*)	-	-	-	$O(\log_2 n)^*$
replaceValue(*e*, *v*)	-	-	-	$O(1)$