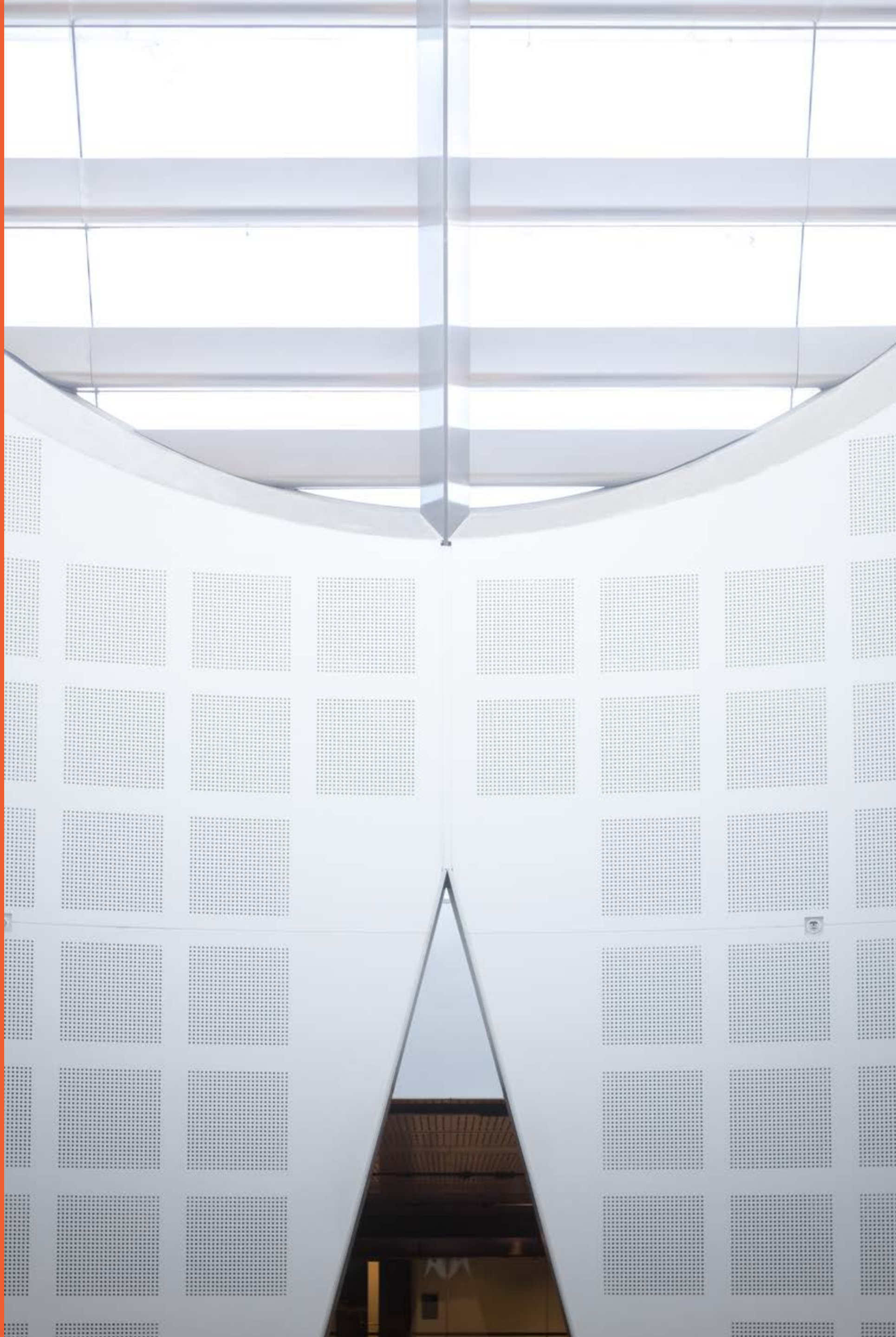
INFO1105/1905/9105
Data Structures
Week 5: Priority Queue, Heap, Sorting

see textbook sections 9.1, 9.2, 9.3, 9.4

Professor Alan Fekete
Professor Seokhee Hong
School of Information Technologies

using material from the textbook
and A/Prof Kalina Yacef

THE UNIVERSITY OF SYDNEY



Copyright warning

- These slides contain material from the textbook (Goodrich, Tamassia & Goldwasser)
  - Data structures and algorithms in Java (6^{th} edition)
- With modifications and additions from the University of Sydney

- The slides are a guide or overview of some big ideas
  - Students are responsible for knowing what is in the referenced sections of the textbook, not just what is in the slides

Reminder! Quiz 2

- Quiz 2 will take place during lab in <u>week 6</u>
- Done online, over a <u>20 minutes</u> duration,
  - during the last 30 minutes of the lab period, or as indicated by your tutor
- A few multiple choice questions,
  - covering material from lectures and labs of <u>weeks 3, 4 and 5</u>
    - stack and queue implementation
    - tree terminology (including trees)
    - binary tree definitions
    - priority queue - ADT
    - sorting

INFO1905 Reminder! AsstX

- AsstX is for info1905 students only
- Due 5pm Friday of week 6
- Instructions are on edstem resources tab
- Submit pdf (not hand-written!) through Turnitin link on eLearning site

Outline

- Priority queues
  - Definition and ADT
  - Java techniques: Entry, Comparator
  - List-based implementations
    - Sorted list, Unsorted list
- Sorting
  - Sorting using Priority queue
  - Insertion sort, Selection sort
- Heap
  - Heap data structure as Binary Tree
  - Heap-based sorting
  - Array-based Heap implementation
  - Refinements: in-place heapsort, heap merging, botton-up heap construction

Priority Queues

- Example: medical emergency queue
- A priority queue stores a collection of prioritised entries
  - Elements are retrieved in order of priority (Eg: by price, length, weight, speed etc)
- Each entry in the Priority Queue is represented by a pair (key, value)
- Key: special characteristic associated to an element by the client at the time it is inserted in the collection
  - used to identify or prioritise that element
  - Eg: price, length, weight, speed
  - Used to store and access the element in order of the defined priority in the collection
- Value: element stored in the collection
- Difference with Stacks and Queue (order defined by WHEN entry inserted) or List (order defined by WHERE entry inserted)

Priority Queue ADT

- A priority queue stores a collection of prioritised entries
- Each entry is a pair (key, value)
- 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( )
- Keys are not necessarily unique
  - If 2 entries have the same key, arbitrary/secondary choice (eg LIFO, FIFO)
- Applications:
  - Job scheduler
  - Customer service with different levels of membership
  - Stock market transactions

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	{ }

Implementing a Priority Queue

- Entries:
  - keep track of the associations between keys and values

- Comparators:
  - compare keys to find the smallest key
    - Needs to be a *total order* relation

- List-based Implementations
  - With sorted list
  - With unsorted list

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

Comparator ADT

- A comparator encapsulates the action of comparing two objects
  - according to a given *total order* relation
- The comparator is external to the objects being compared
  - this is more general than using objects that implement Comparable
- In Priority Queue implementations, we can pass a Comparator in the constructor, or default to using Comparable keys

Primary method of the Comparator ADT

- compare(a, b): 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.

Warning: do not assume that compare(a,b) is always -1, 0, 1

Total Order Relations

- Keys in a priority queue can be arbitrary objects on which an order is defined
- 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: when $x \leq y$ and $y \leq x$, then $x = y$
  - Transitive property: when $x \leq y$ and $y \leq z$, then $x \leq z$
  - Smallest key k_{min} : $k_{min} \leq x$ for all x

Example Comparator

- 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;
   }
}
```

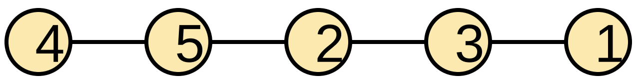
Comparator in Priority Queue implementations

- When the Priority Queue is constructed, client can pass a Comparator<K> in the constructor method
  - this is stored in an instance variable, and invoked whenever keys need to be compared to one another
- If client chooses, and K implements Comparable, client can use overloaded constructor that does not need a Comparator<K> argument
  - a Comparator<K> is created by the Priority Queue itself during construction

```
public int compare(K a, K b) {
      return a.compareTo(b);
}
```

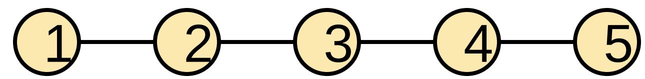
Sequence-based Priority Queue

- Implementation with an unsorted list (Sect 9.2.4)



- 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 (Sect 9.2.5)



- 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

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, cont

```
/** 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

```java
/** 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, cont

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

Running times for list-based PQs

Sorted list implementation

- Insert operation needs to go through the sorted collection and find the correct position to insert the element
- removeMin( ) is simple (just remove the first element it is the smallest)

Unsorted list implementation

- Insert operation is simple (just insert at the beginning of the list)
- removeMin( ) now needs to go through the unsorted collection and select the minimum element to remove.

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

Which is better?

- Unsorted list implementation is quick for insert() but slow for removeMin()
- Sorted list implementation is quick for removeMin() but slow for insert()
- Usually, an application will perform lots of insert() AND lots of removeMin()
  - we expect about the same number of each, as elements don't usually stay for ever in the collection
    - so each implementation has worst-case cost that scales linearly
  - some applications may do a lot of min() without removal, so some advantage to sorted implementation where min() is fast

Outline

- Priority queues
  - Definition and ADT
  - Java techniques: Entry, Comparator
  - List-based implementations
    - Sorted list, Unsorted list
- Sorting
  - Sorting using Priority queue
  - Insertion sort, Selection sort
- Heap
  - Heap data structure as Binary Tree
  - Heap-based sorting
  - Array-based Heap implementation
  - Refinements: in-place heapsort, heap merging, botton-up heap construction

Priority Queue Sorting

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of insert operations
     - element is used as key
     - null is the value (never considered, just goes along to fit the priority queue API)
  2. Remove the elements one-by-one with a series of removeMin operations
     - elements come out in sorted order
- 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.removeFirst* ()
        P.insert (*e*, **null**)
    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 list*
- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with n insert operations takes $n*O(1)$ that is $O(n)$ time
  2. Removing the elements in sorted order from the priority queue with n removeMin operations takes time $n*O(n)$ that is $O(n^2)$

  More precise analysis recognizes that the collection size changes during execution. So phase 2 takes time proportional to

  $$n+(n-1)+(n-2)+\ldots+1 = \frac{1}{2}\, n(n+1)$$

  Still, removing them all takes $O(n^2)$ time
- So, Selection-sort runs in $O(n)+O(n^2)$ which is $O(n^2)$ time
- Selection is the bottleneck computation

Selection-Sort Example

	List 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

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted list
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with n insert operations takes time $n*O(n)$ that is $O(n^2)$

  More precise analysis shows time is proportional to

  $$1 + 2 + \ldots + n = \tfrac{1}{2}n(n+1) \text{ but still } O(n^2)$$

  2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $n*O(1) = O(n)$ time
- Insertion-sort runs in $O(n^2)$ time
- Insertion is the bottleneck computation

Insertion-Sort Example

	List 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)	()

Outline

- Priority queues
  - Definition and ADT
  - Java techniques: Entry, Comparator
  - List-based implementations
    - Sorted list, Unsorted list
- Sorting
  - Sorting using Priority queue
  - Insertion sort, Selection sort
- Heap
  - Heap data structure as Binary Tree
  - Heap-based sorting
  - Array-based Heap implementation
  - Refinements: in-place heapsort, heap merging, botton-up heap construction

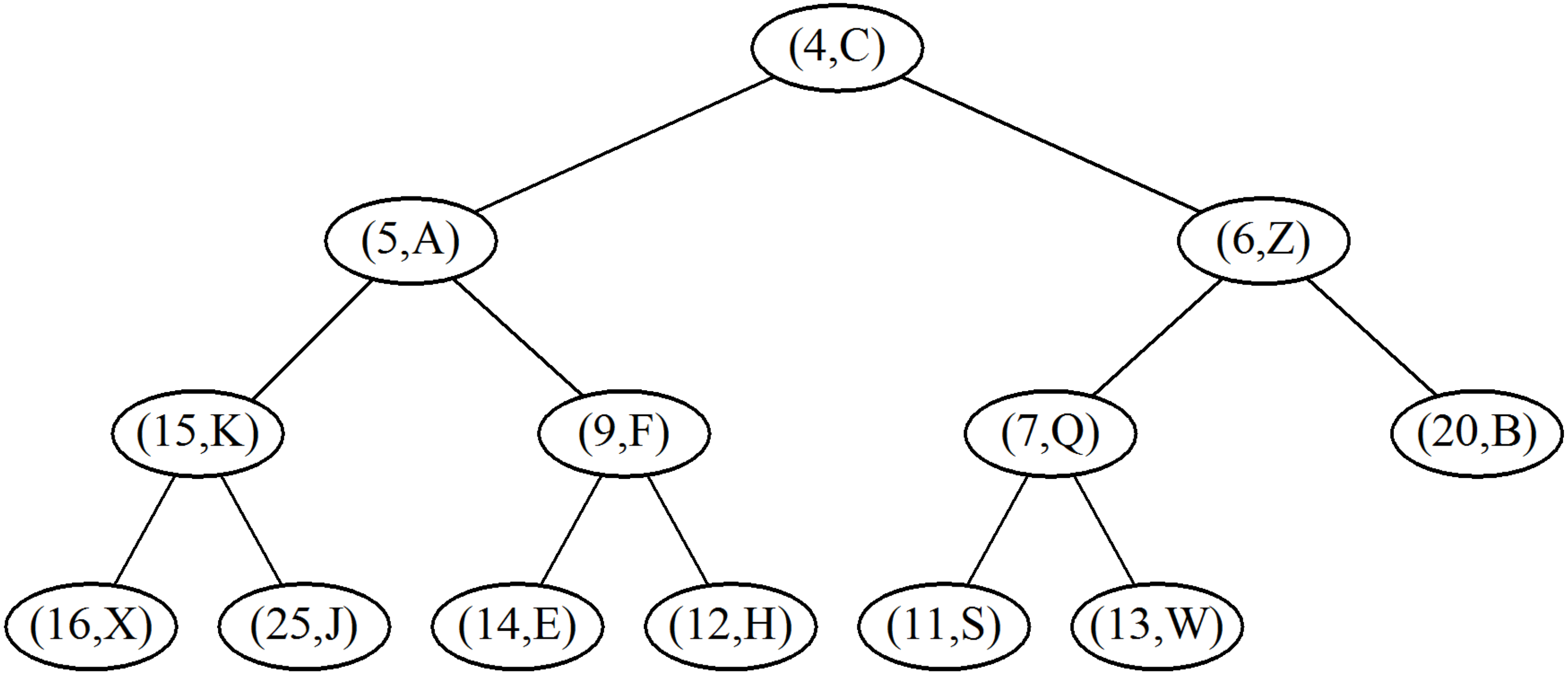
Heap

- Clever data structure for implementing priority queue
  - more efficient overall than the obvious list-based previous solutions
  - Each Insert and RemoveMin is done in ***O(log n) time***
  - Allows sorting to be done in ***O(n log n) time***
- Based on Binary tree
  - with restrictions to enforce that height is O(log n)

Heap data structure

- A **heap** is a binary tree storing entries (k,v) at its nodes and satisfying both the following properties:
- Heap-Order : for every internal node m other than the root, $\boldsymbol{key(m) \geq key(parent(m))}$
  - Keys on any path are in non-decreasing order
  - Therefore, min key is stored at the root
- Complete Binary Tree : let $\boldsymbol{h}$ be the height of the heap
  - for $\boldsymbol{i} = 0, \ldots, \boldsymbol{h} - 1$, there are 2^i nodes of depth $\boldsymbol{i}$ (that is, maximum possible number of nodes at each of these levels)
  - at depth $\boldsymbol{h}$, the remaining nodes are in the leftmost positions

Example

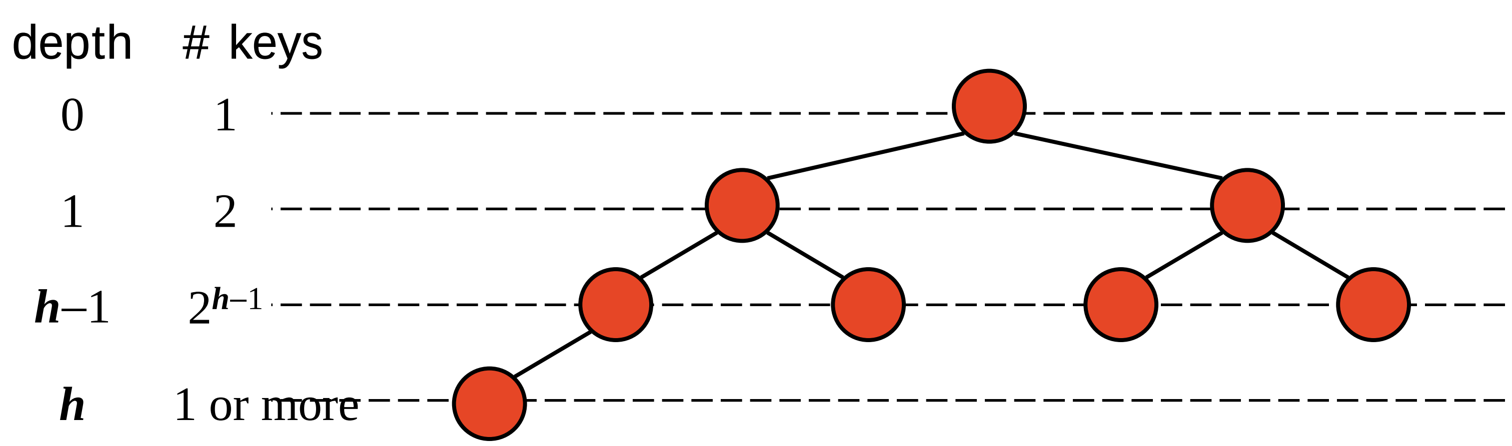


Height of a Heap

- **Theorem:** A heap storing n keys has <u>height $O(\log n)$</u>

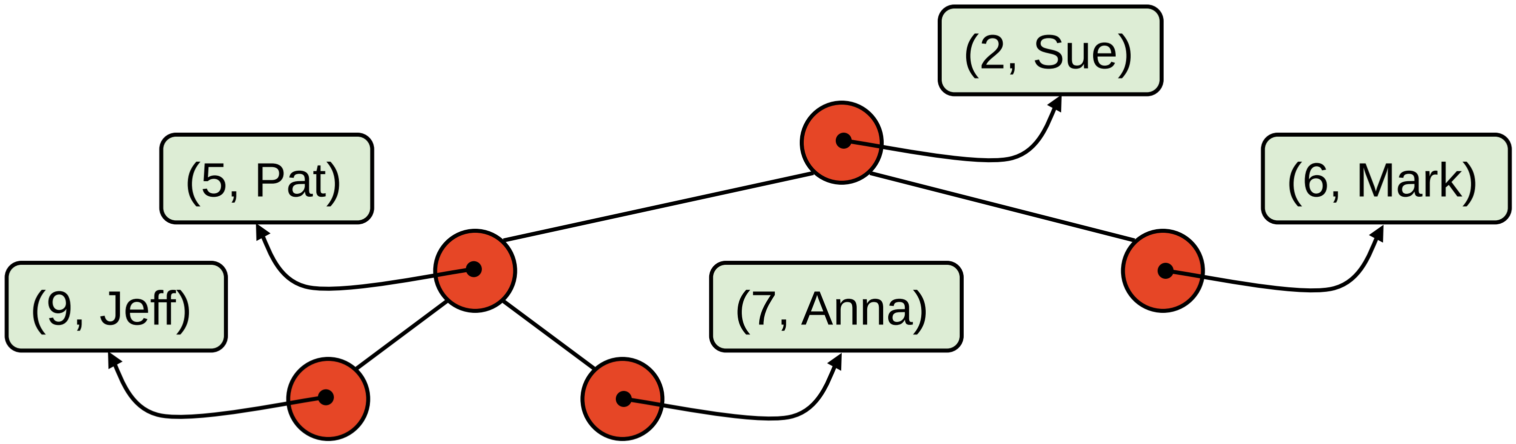
  Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
  - Thus, $n \geq 2^h$, i.e., $h \leq \log n$

  - Logarithmic time to update heaps, as long as we do only O(1) work per level!



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

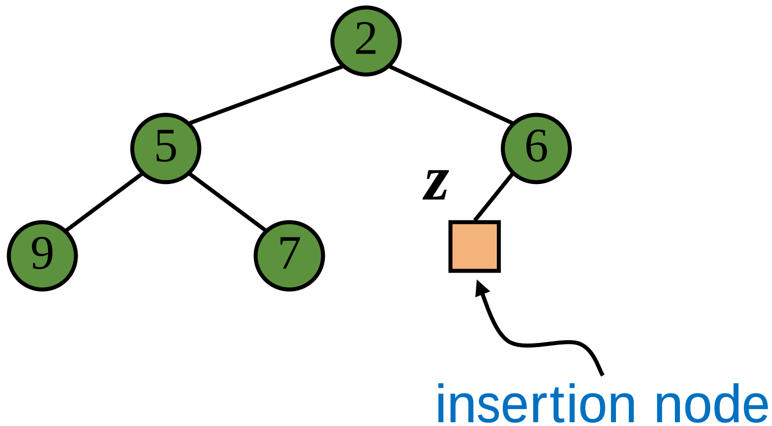


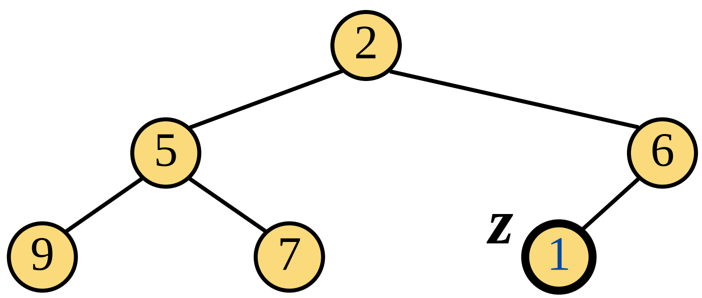
Operations on a heap (overview)

- Typically, it is easy to insert or removeMin() in the binary tree, BUT the new structure may not satisfy the rules of a heap
- So, do it anyway, and then work to return the structure to being a heap
  - restore the complete binary tree shape (easy)
  - restore the heap-order property (by rearranging the nodes)

Insertion into a Heap

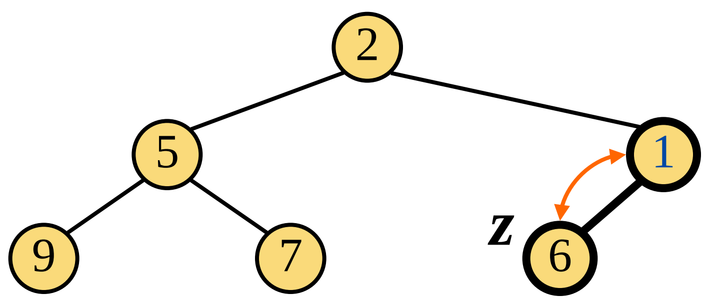
- Method insert of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z (restores the complete binary tree structure)
  - Restore the heap-order property (discussed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

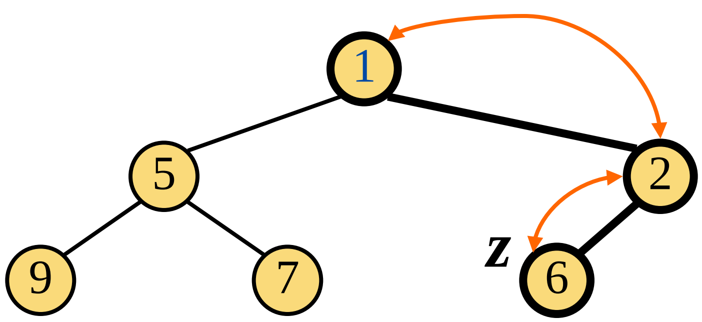




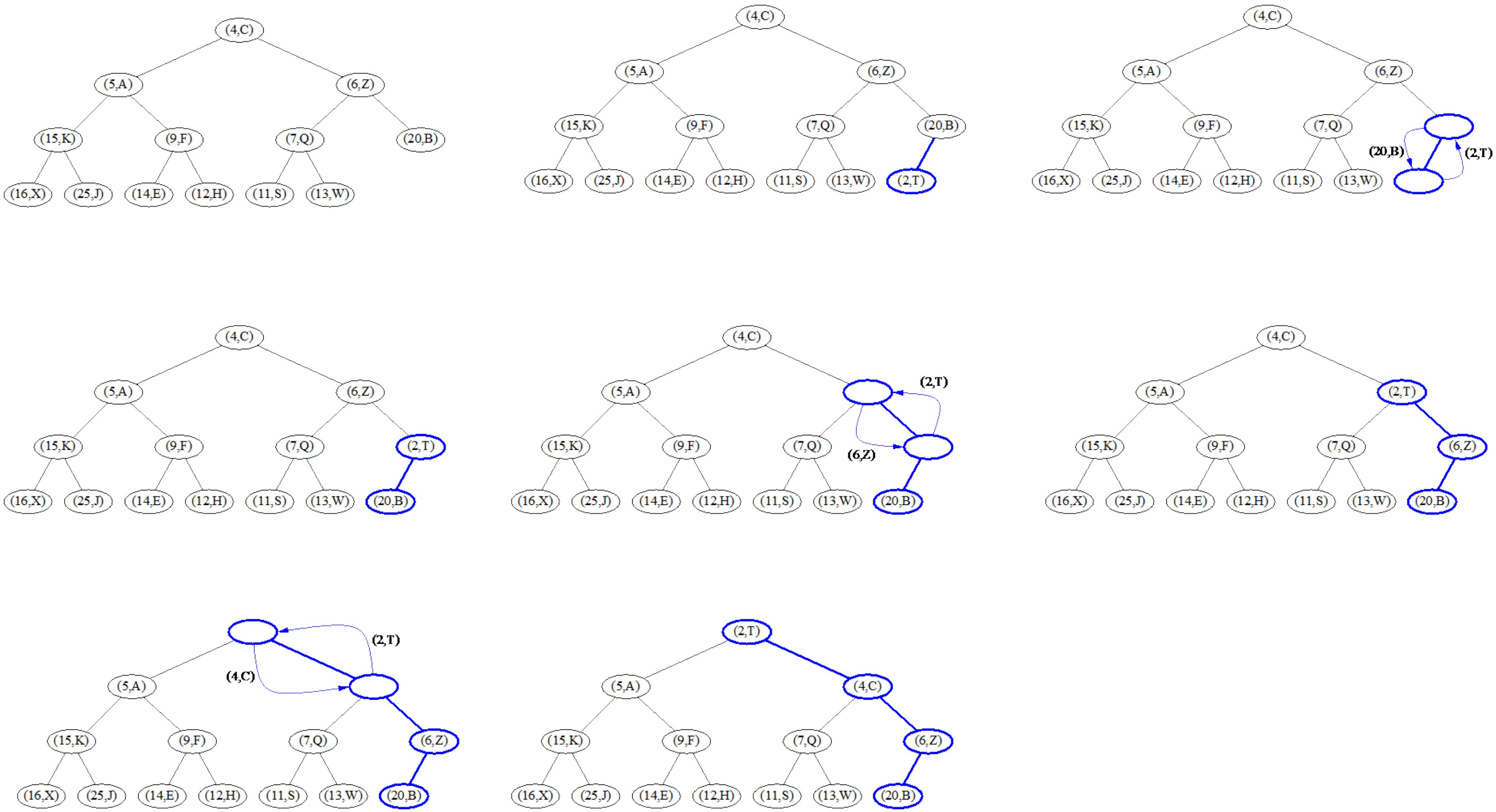
Upheap

- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by <u>swapping k along an upward path from the insertion node</u>
- 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 <u>$O(\log n)$ time</u>



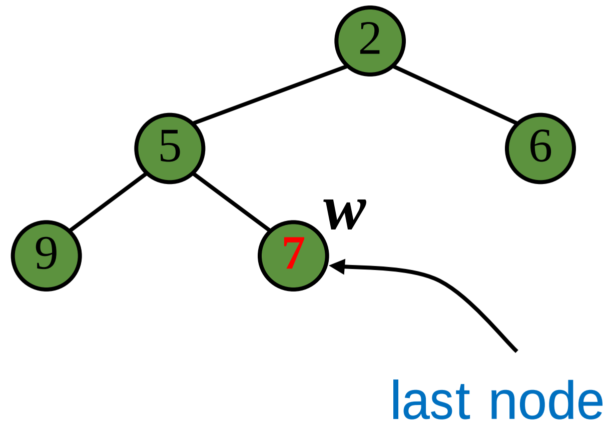


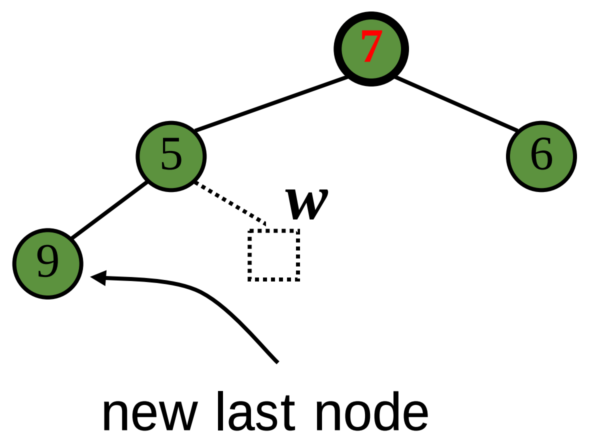
Example insertion



Removal from a Heap

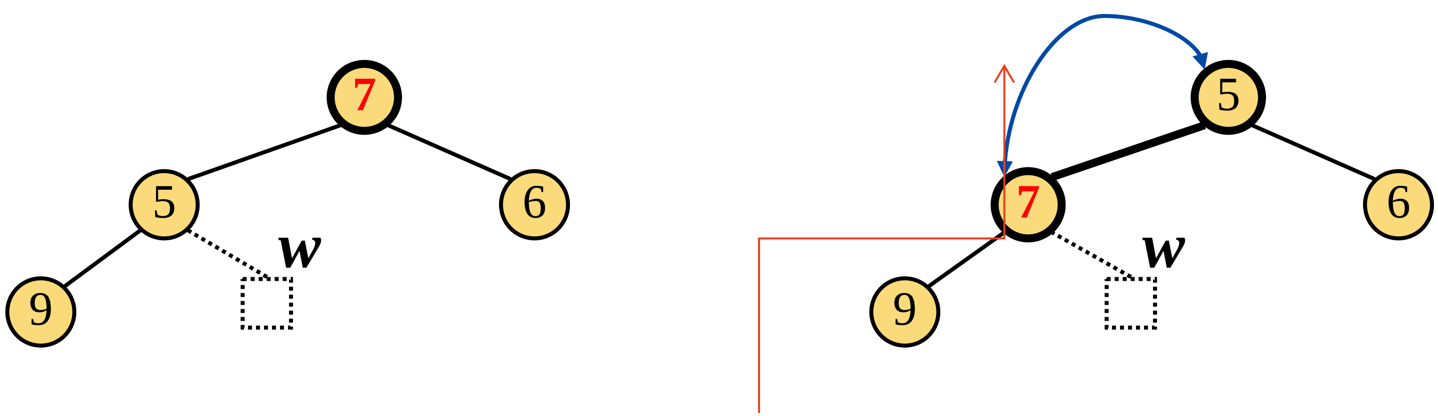
- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node *w*
  - Remove *w* (restores complete binary tree structure)
  - Restore the heap-order property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





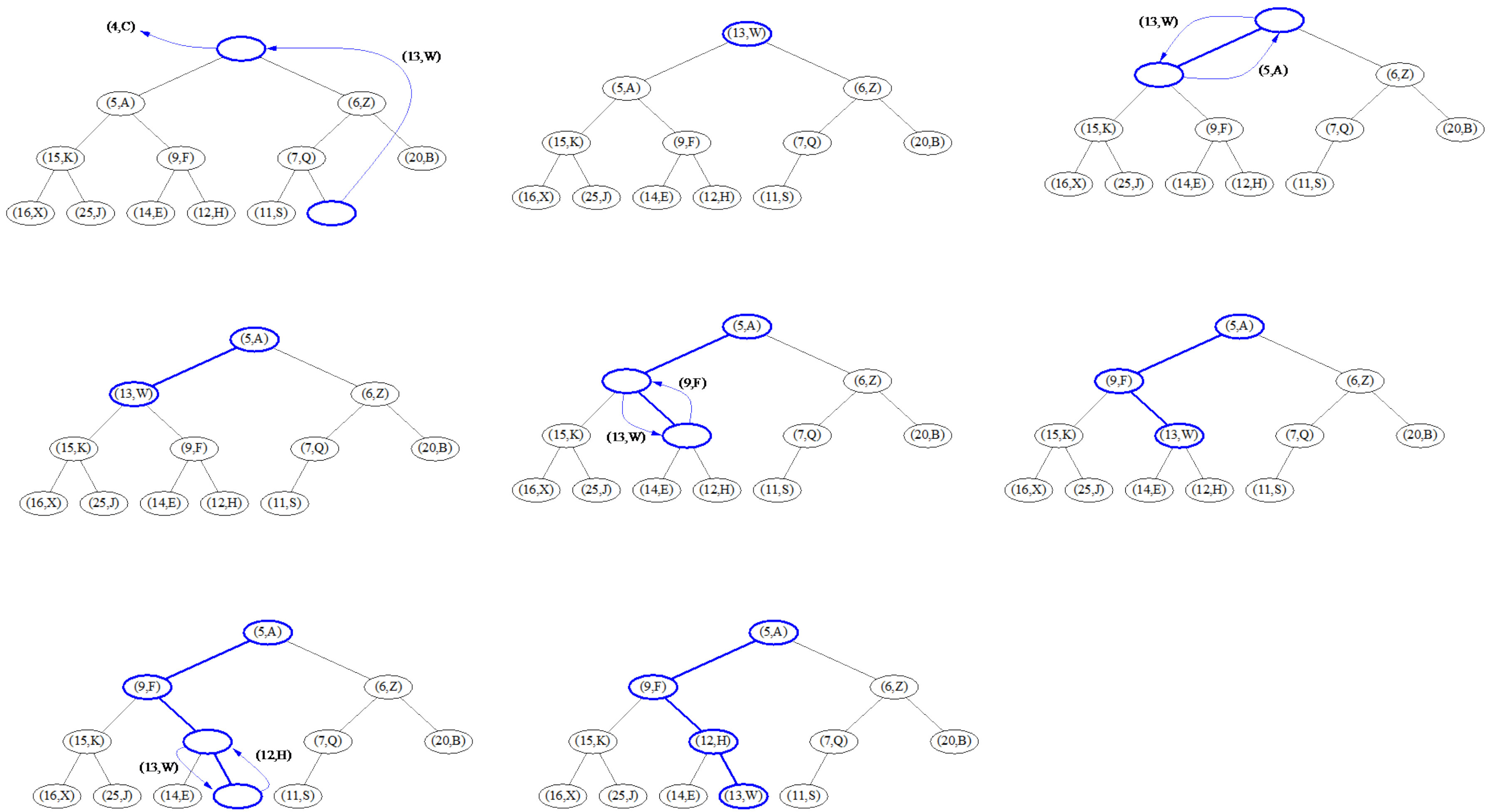
Downheap

- 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 (with the smallest child)
- Downheap 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



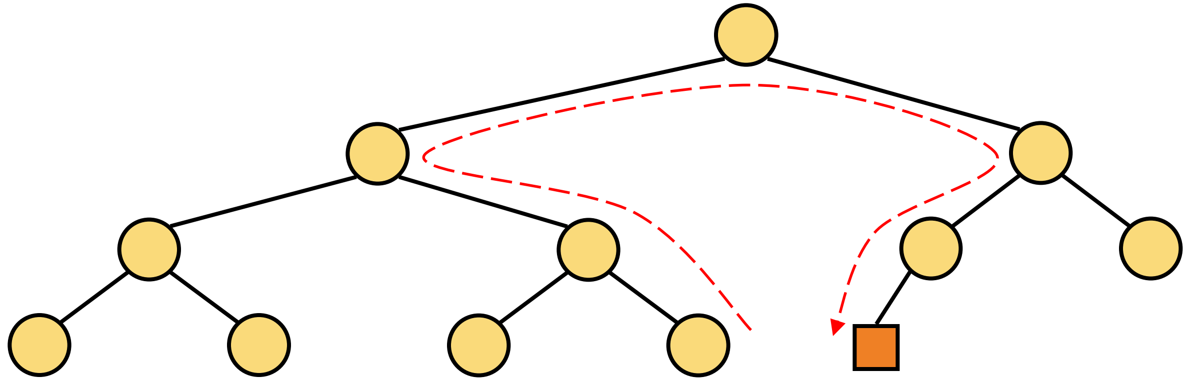
Choose the smallest child for the down-swap

Example removal

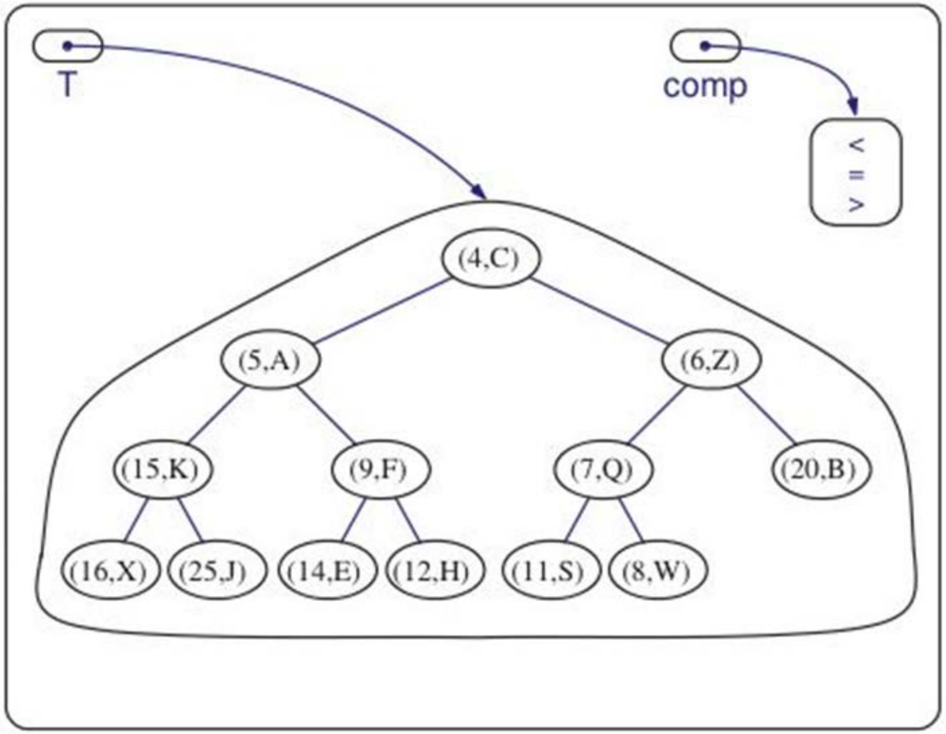


Updating the Last Node

- 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



Heap-based implementation of a priority queue



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

Sorting with Priority Queues (recall)

- Input: S, unsorted
- Output: S, sorted

- Phase 1: insert elements of S as keys in P
- Phase 2: remove elements from P in non-decreasing order and place them in S

- We saw Selection-Sort and Insertion-Sort, both with $O(n^2)$ computation

- What if we use heap for the priority queue?

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 $\underline{O(n \log n)}$ time
- The resulting algorithm is called **heap-sort**
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

Heap Sort Analysis

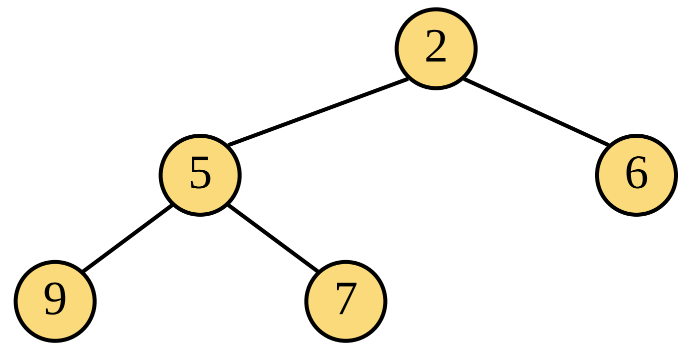
- Phase 1: n insert operations, each takes O(log n) time. Thus O(n log n) time in total

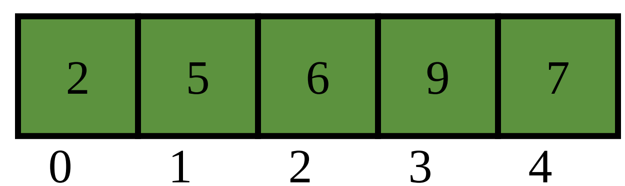
- Phase 2: n removeMin operations, each takes O(log n), thus O(n log n) time in total

- Whole heap-sort procedure takes O(n log n)
  - Note: if we use a resizable array (to deal with the case where the set being sorted is of unknown size), then these bounds are true only when amortized (because the insert could be very expensive, if capacity increase happens in that step)

Array-based Heap Implementation (code pp 377-378)

- We can represent a heap with n keys by means of an <u>array of length n</u> (recall that arrays are an appropriate implementation for <u>complete</u> binary trees)
- For the node at <u>index i</u>
  - the left child is at <u>index $2i+1$</u>
  - the right child is at <u>index $2i + 2$</u>
  - Parent is at index rounddown((i-1)/2)
  - Root is at index 0
- Links between nodes are not explicitly stored
- Operation insert corresponds to inserting at index n and rearranging
- Operation removeMin corresponds to removing at index 0, and then shifting from index $n-1$, and rearranging





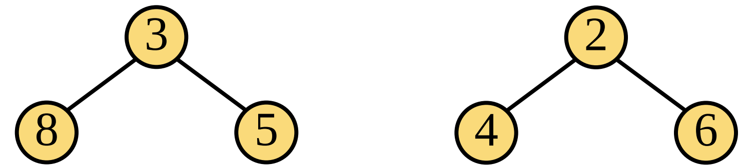
Refinements

- There are several small changes that are made in practice
- Heap-sort can be arranged to work in place (use part of the array for the input/output, and part for the priority-queue)
  - use max-heap variant, where each item is *greater than or equal* to its children
- Construct a whole heap from n items can be done more effectively than by n successive insertions
  - total time for phase 1 is O(n)
  - But heapsort overall is still O(n log n)

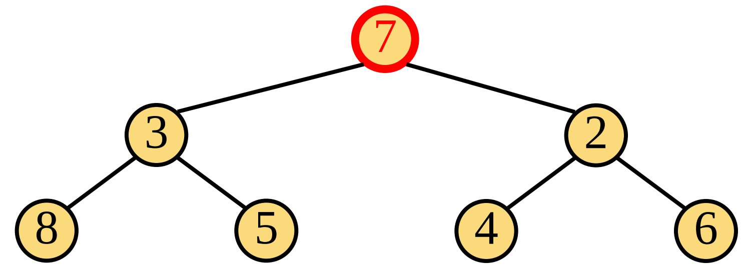
- Details in textbook Sections 9.3.4, 9.4.2
  - examinable, but focus first on the simpler concepts

Merging Two Heaps

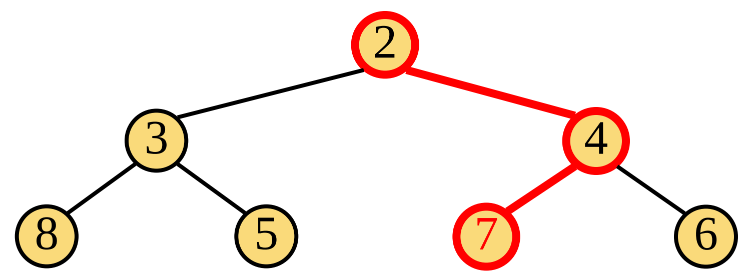
- We are given two heaps and a key ***k***



- We create a new heap with the root node storing ***k*** and with the two heaps as subtrees

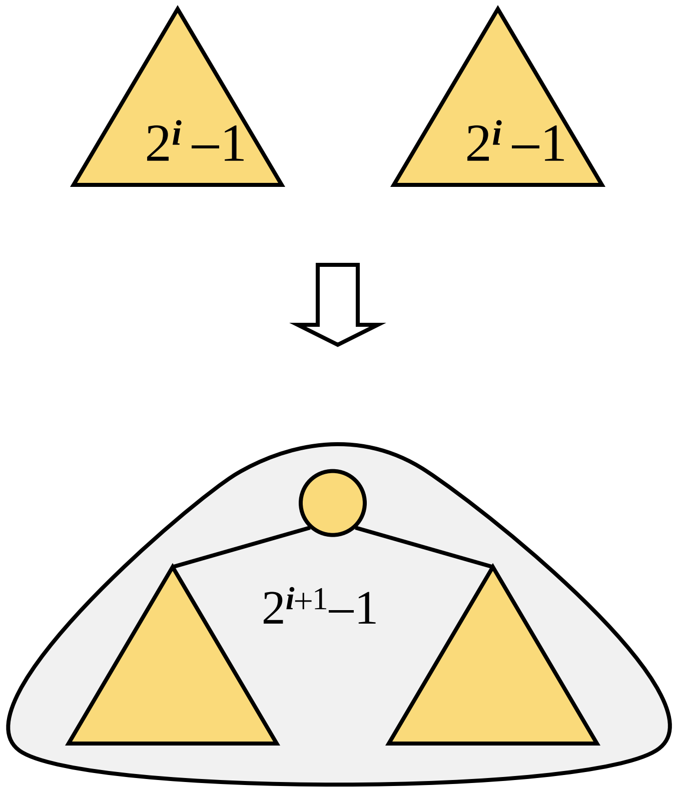


- We perform downheap to restore the heap-order property

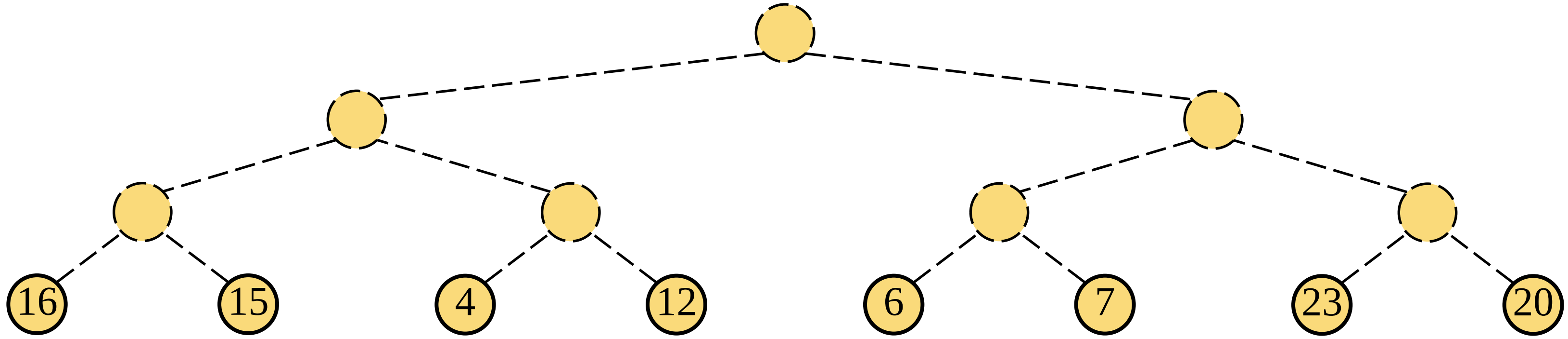


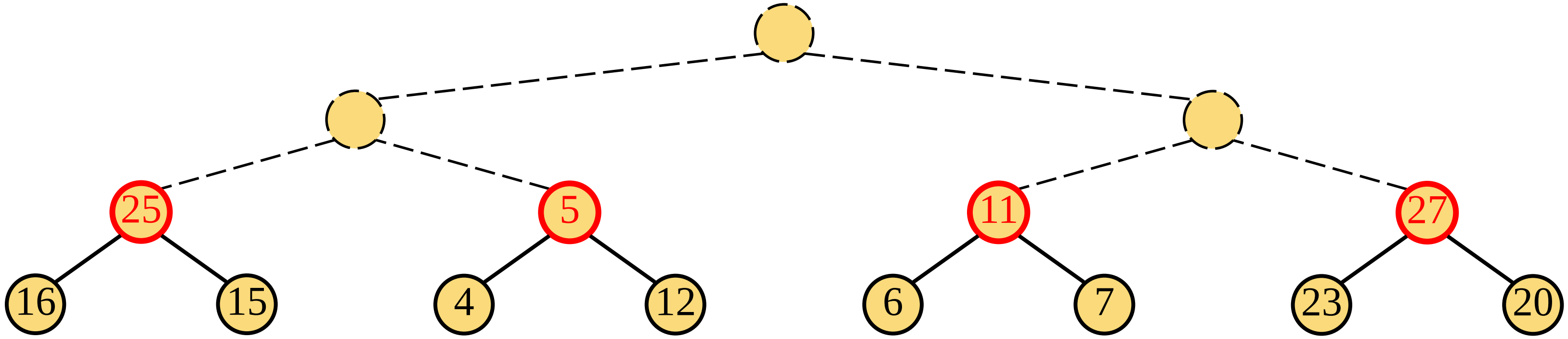
Bottom-up Heap Construction

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

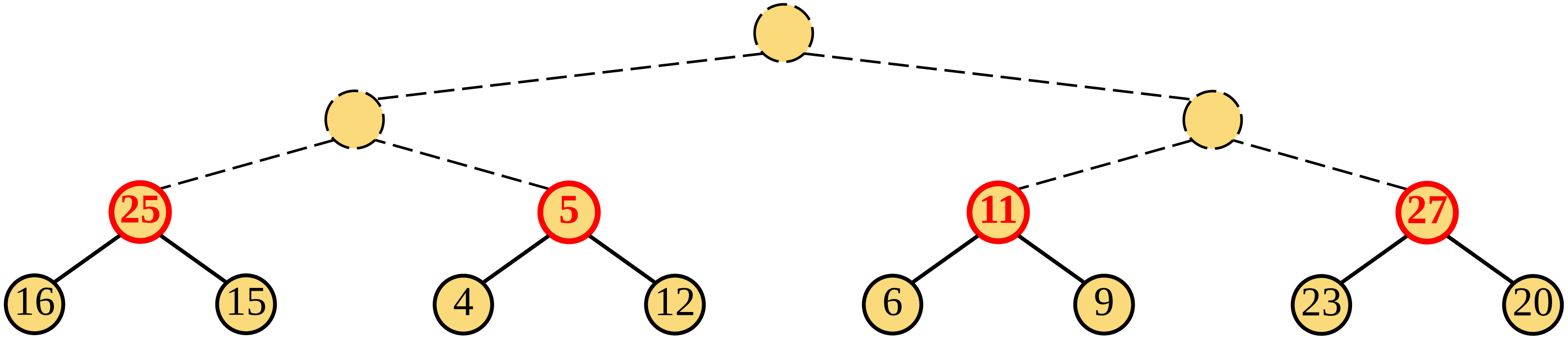


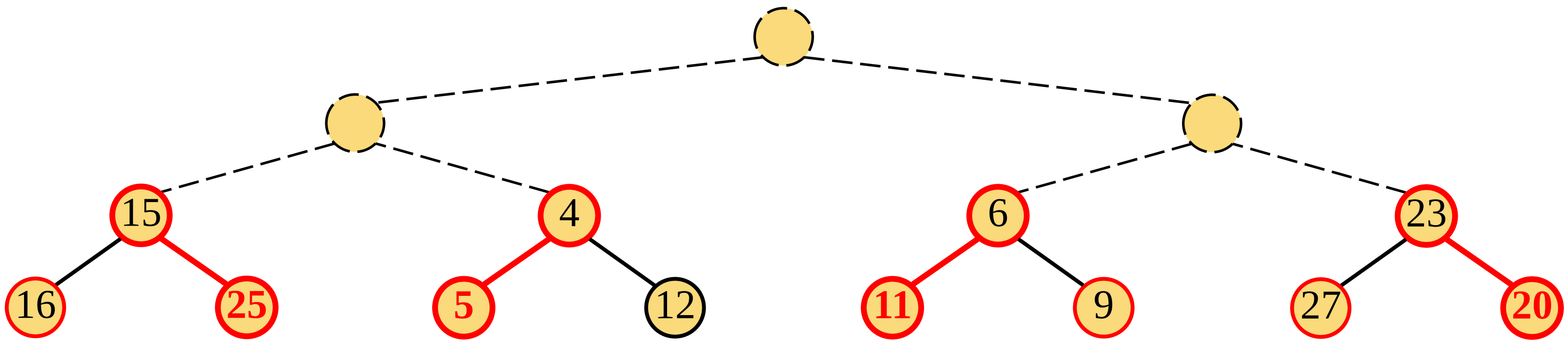
Example



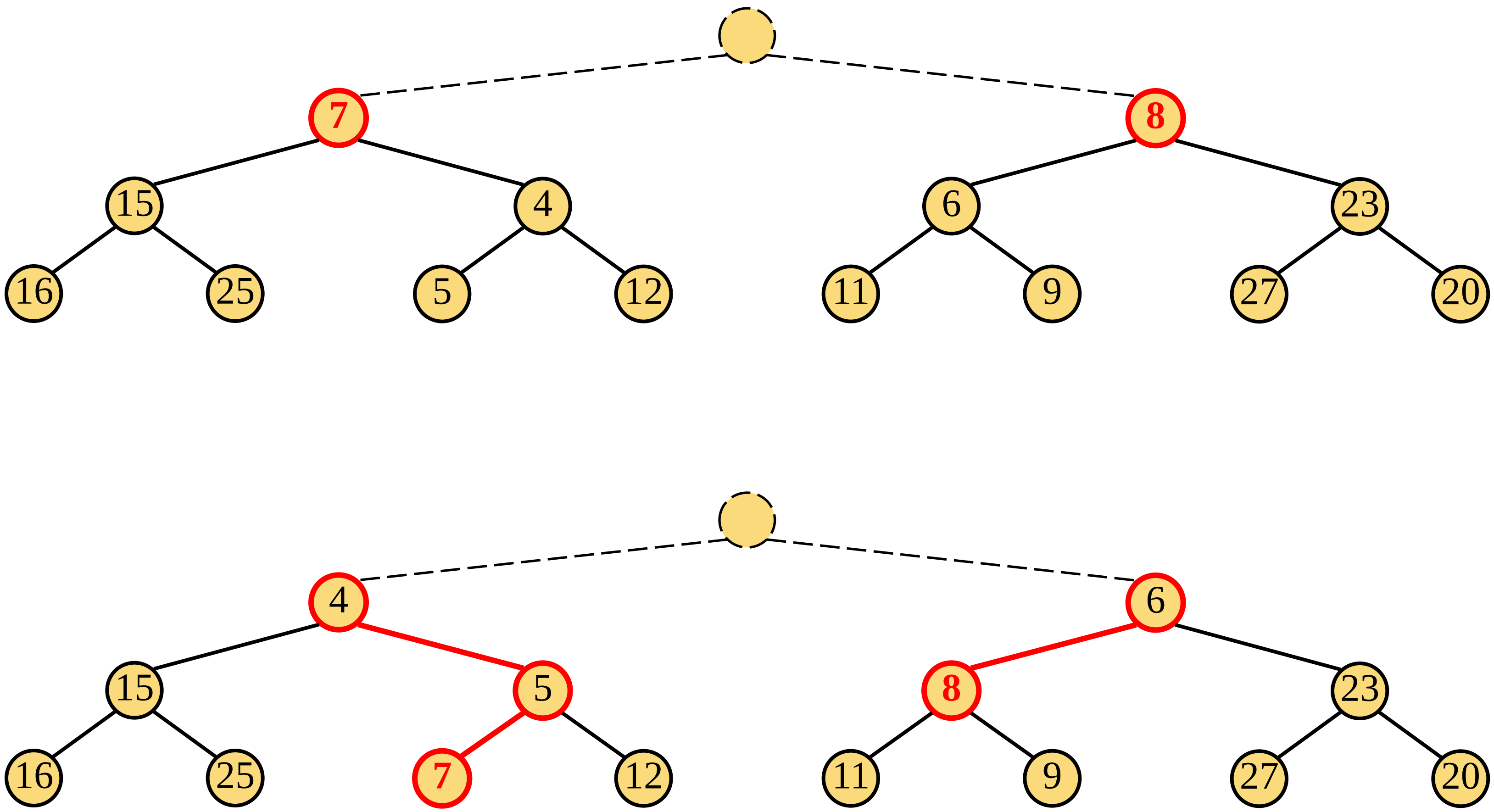


Example (contd.)

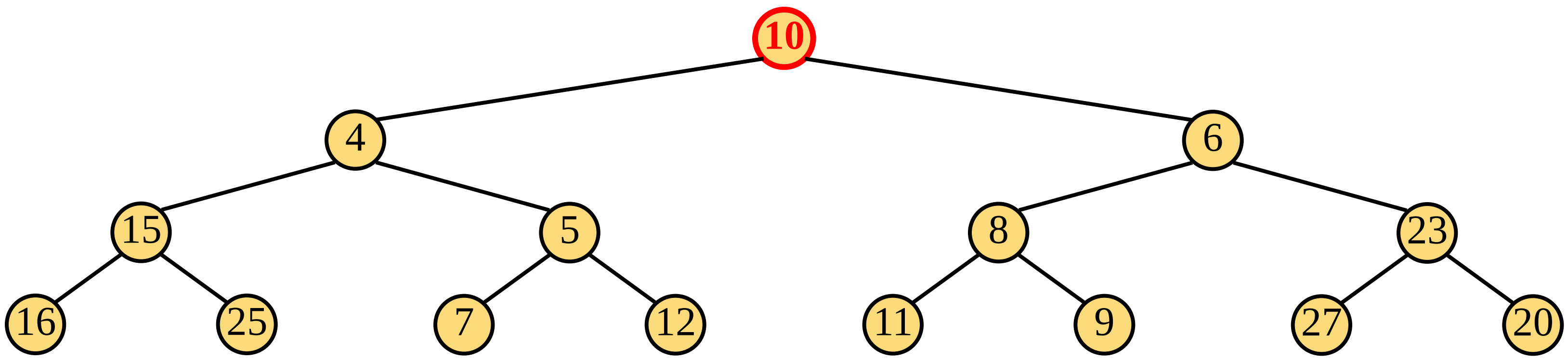


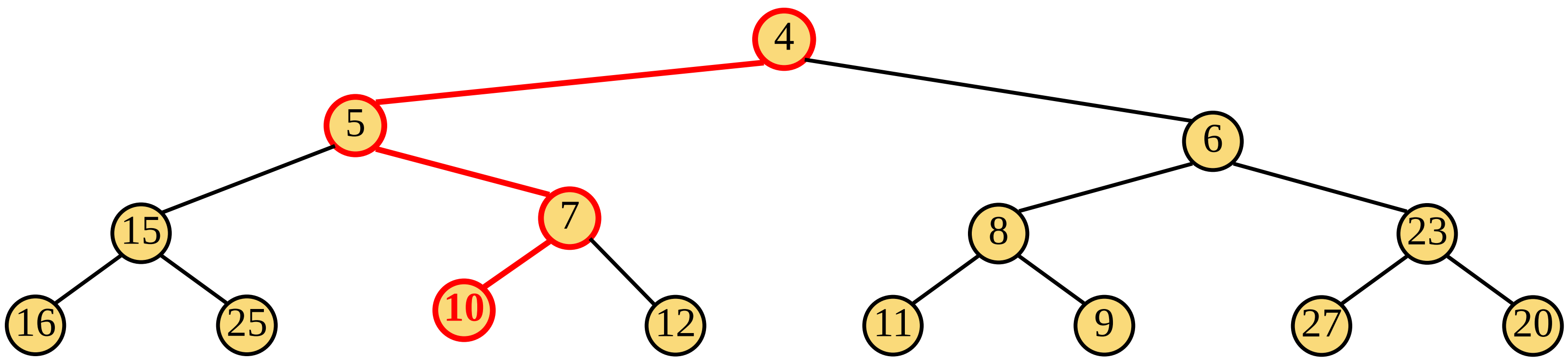


Example (contd.)



Example (end)





Summary: Comparison of priority queue implementations

Know this table by heart for the quizzes and exam!

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	$O(1)$	$O(1)$	$O(\log n)$
replaceKey	$O(1)$	$O(n)$	$O(\log n)$
replaceValue	$O(1)$	$O(1)$	$O(1)$

Summary

- Read Sections 9.1 to 9.4 of the textbook
- Priority queues
  - Definition and ADT
  - Java techniques: Entry, Comparator
  - List-based implementations
    - Sorted list, Unsorted list
- Sorting
  - Sorting using Priority queue
  - Insertion sort, Selection sort
- Heap
  - Heap data structure as Binary Tree
  - Heap-based sorting
  - Array-based Heap implementation
  - Refinements: in-place heapsort, heap merging, botton-up heap construction