Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Priority Queues

Priority Queues

- Define **priority queue** ADT
- Implement a priority queue with an unsorted and sorted lists
- Explain **Heap** data structure
- Implement a priority queue with a heap
- Analyze **heap-based priority queues**
- **Sorting** with a priority queue

Trees Review

- Hierarchical data structure that consists of **nodes** with a **parent-child relation**
- **Root**: node **without parent** (A)
- **Internal node**: node with **at least one child** (A, B, C, F)
- **External node** (a.k.a. **leaf**): node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node: parent, grandparent, grand-grandparent, etc.
- **Depth of a node**: number of ancestors
- **Height of a tree**: maximum depth of any node (3)
- **Descendant of a node**: child, grandchild, grand-grandchild, etc.
- **Subtree**: tree consisting of a node and its descendants
- A tree is ***ordered*** if there is a meaningful linear order among the children of each node
- Tree ADT
  - We **use positions** to abstract nodes
  - **Tree interface** for trees where nodes can have any number of children
    - parent
    - children
    - numCHildren

Trees Review

- isInternal
- isExternal
- isRoot
- isEmpty
- Iterator
- position

- **AbstractTree Base Class implements:**
  - isInternal
  - isExternal
  - isRoot
  - isEmpty

- **Tree Traversal**
  - a **preorder** traversal of a tree T, the root of T is visited first and then the subtrees rooted at its children are traversed recursively
  - **postorder** traversal of a tree it recursively traverses the subtrees rooted at the children of the root first, and then visits the root

Trees Review

- **Binary Trees**
  - Each internal node has **at most two children** (exactly two for proper binary trees)
  - The children of a node are an ordered pair
  - We call the children of an internal node left child and right child
  - Applications
    - arithmetic expressions
    - decision processes
    - searching

- **BinaryTree ADT**
- **BinaryTree** interface extends the Tree interface and adds three methods:
  - position left(p)
  - position right(p)
  - position sibling(p)
- **AbstractBinaryTree** Base Class extends **AbstractTree** and implement **BinaryTree**
- Implements:
  - sibling
  - numChildren
  - children

Trees Review

- **Binary Trees**
  - **inorder traversal** a node is visited after its left subtree and before its right subtree
- **Implementing Trees**
  - Using a **linked structure**
  - A node is represented by an object storing
    - Element
    - Parent node
    - Sequence of children nodes
  - Node objects implements the Position ADT
- **Linked Structure for Binary Trees**
  - A node is represented by an object storing
    - **Element**
    - **Parent node**
    - **Left child node**
    - **Right child node**
  - Node objects implements the Position ADT
- **LinkedBinaryTree class**
  - Inner **Node** class
  - Two instance variables: ***root*** and ***size***
  - Updating operations

Motivation for Priority Queues

- The first-in, first-out (FIFO) principle is a good fit for applications such as a customer call center in which waiting customers are told "calls will be answered in the order that they were received."
- However, an air-traffic control center decision cannot be based purely on FIFO
- There are other factors to consider:
  - each plane's distance from the runway,
  - time spent waiting in a holding pattern,
  - or amount of remaining fuel.

Priority Queue ADT

- **FIFO** principle used by queues does not suffice – priorities must come into play (examples)
- A **priority queue** is **a collection of prioritized entries** that allows arbitrary **entry insertion**, and allows the **removal of the entry that has first priority**
- Each entry is a pair **(key, value)**
- Main methods of the Priority Queue ADT (interface ***PriorityQueue***):
  - 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 the priority queue is empty
- Additional methods
  - min() - **returns**, but does not remove, an entry with smallest key, or null if the the priority queue is empty
  - size()
  - isEmpty()
- Applications:
  - Standby flyers
  - ER queues
  - 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	{ }

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**

- For a comparison rule, which we denote by **≤**, to be self-consistent, it must define a ***total order*** relation
- 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 $y \leq z \Rightarrow x \leq z$

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();
}
```

The Comparable Interface

- Java provides two means for defining comparisons between object types:
  - A class may define the natural ordering of its instances by implementing *Comparable* interface, which contains 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.

Comparator ADT

- In some applications, we may want to compare objects **according to some notion** other than their natural ordering.
- **Comparator** interfaces supports this generality.
- A **comparator** encapsulates the action of **comparing two objects according to a given total order relation, rather than their natural order**.
- 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.

Example Comparator

- As a concrete example, Code Fragment 9.3 defines a **comparator that evaluates strings based on their length** (rather than their natural lexicographic order):

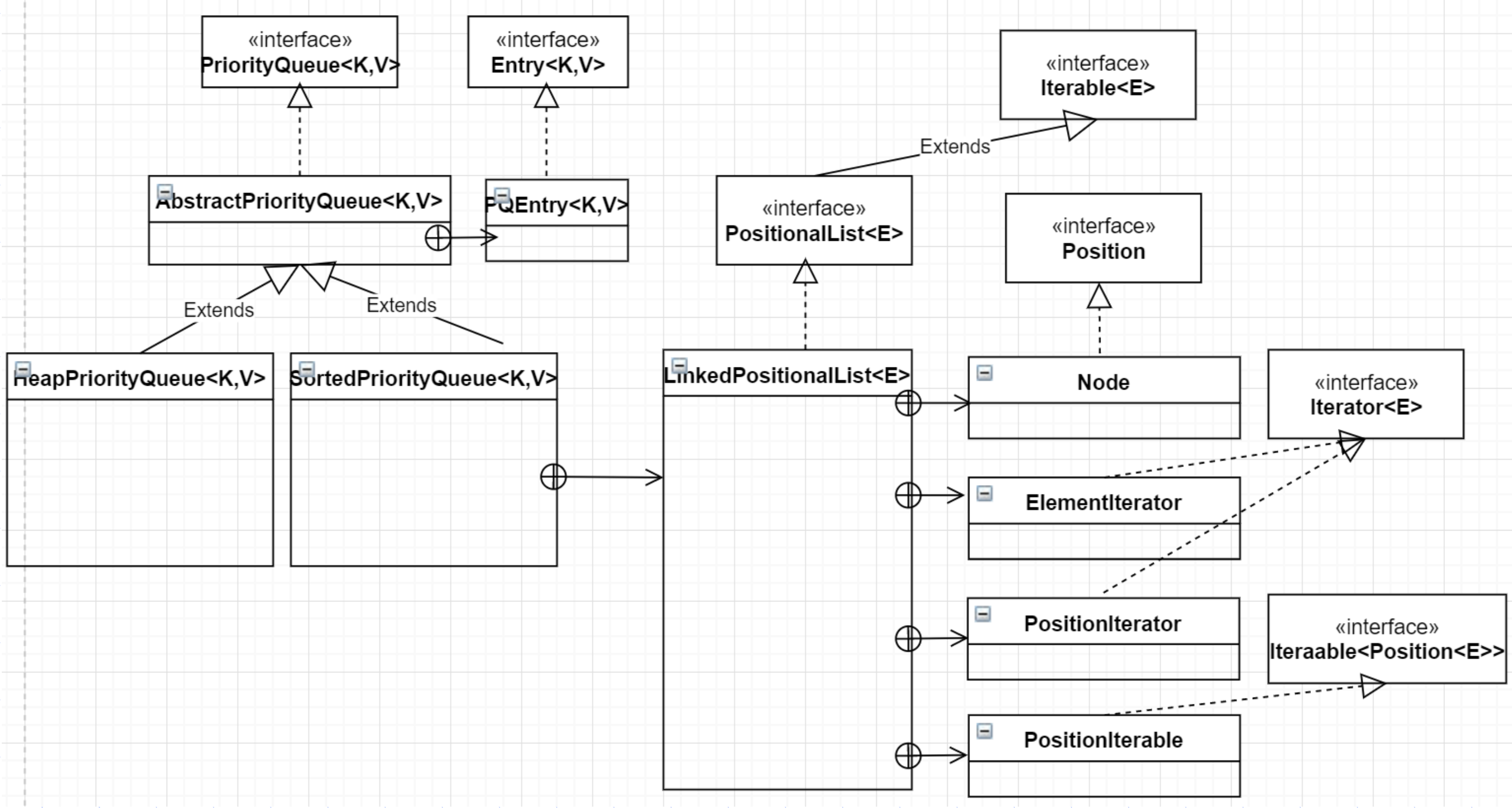
```
public class StringLengthComparator implements Comparator<String> {
  /** Compares two strings according to their lengths. */
  public int compare(String a, String b) {
    if (a.length() < b.length()) return −1;
    else if (a.length() == b.length()) return 0;
    else return 1;
  }
}
```

Code Fragment 9.3: A comparator that evaluates strings based on their lengths.

The AbstractPriorityQueue Base Class

- The base class provides four means of support:
  1. a ***PQEntry*** class as a concrete implementation of the *Entry* interface
  2. an instance variable ***comp*** for a general *Comparator* and a protected method, ***compare*****(a, b)**, that makes use of the comparator.
  3. a boolean ***checkKey*** method that verifies that a given key is appropriate for use with the comparator
  4. an ***isEmpty*** implementation based upon the abstract ***size()*** method.

PriorityQueue ADT



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**

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

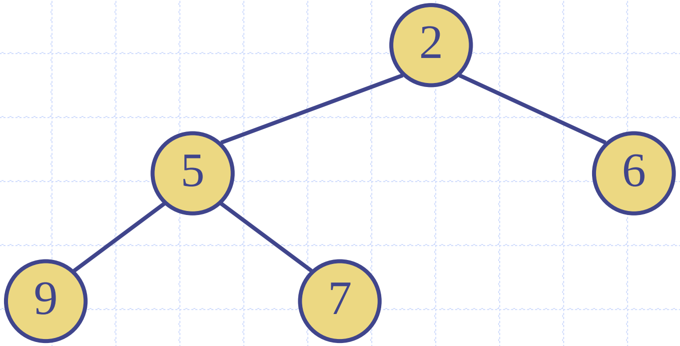
```java
/** An implementation of a priority queue with a sorted list. */
public class SortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  /** primary collection of priority queue entries */
  private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>();

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

Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Heaps

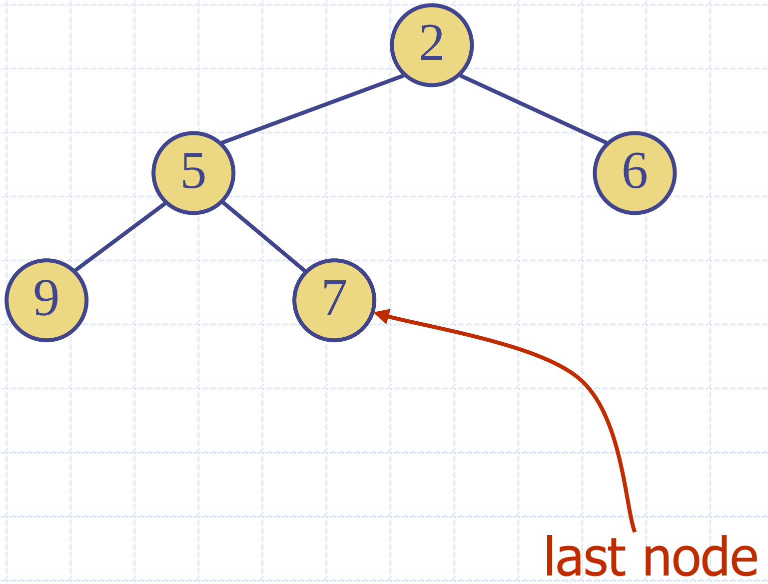


Recall Priority Queue ADT

- A **priority queue** stores a collection of 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
- Additional methods
  - min() - returns, but does not remove, an entry with smallest key
  - size()
  - isEmpty()
- Applications:
  - Standby flyers
  - ER queue
  - Auctions
  - Stock market

Heaps

- 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, $key(v) \geq key(parent(v))$
  - Complete Binary Tree: let h be the **height** of the heap
    - for $i = 0, \ldots, h-1$, there are 2^i nodes of depth i
    - remaining nodes at level **h** reside in the **leftmost possible positions** at that level.
- The two nodes in level 2 are in the two **leftmost possible positions** at that level.
- The last node of a heap is the **rightmost node of maximum depth**

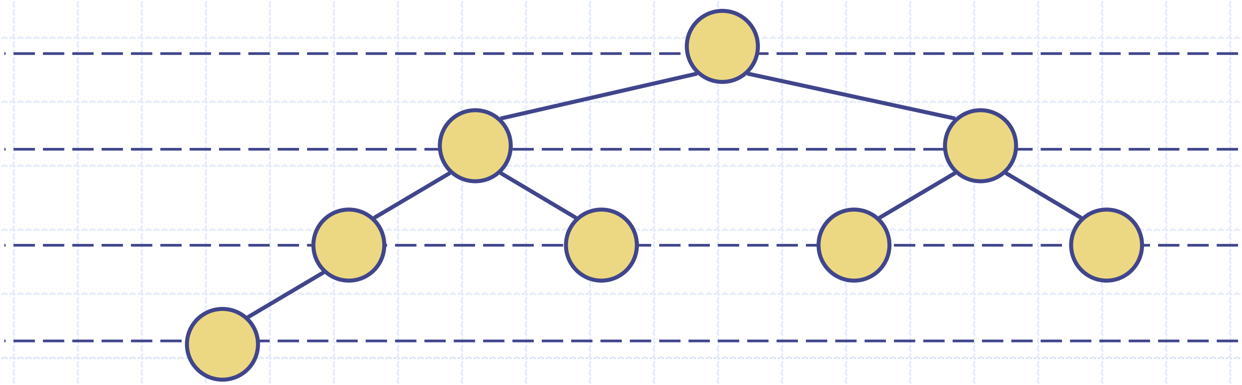


Height of a Heap

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

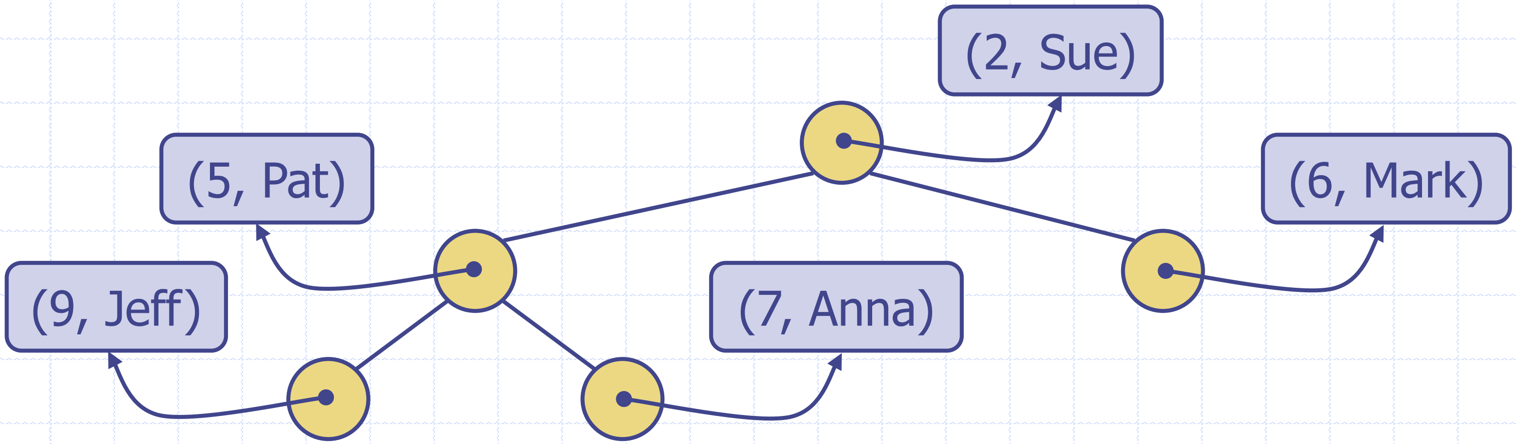
  Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \dots, h-1$ and at least one key at depth h, we have $n \geq (1 + 2 + 4 + \dots + 2^{h-1}) + 1 = (2^h - 1) + 1 = 2^h$
  - Thus, $n \geq 2^h$. By taking the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

depth	keys
0	1
1	2
$h-1$	2^{h-1}
h	1



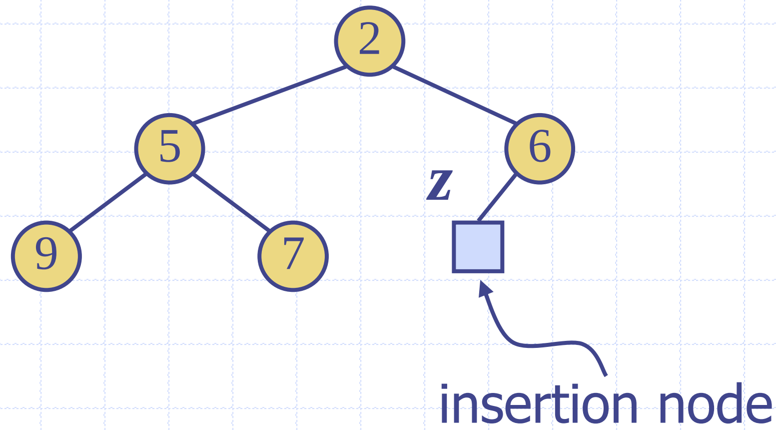
Heaps and Priority Queues

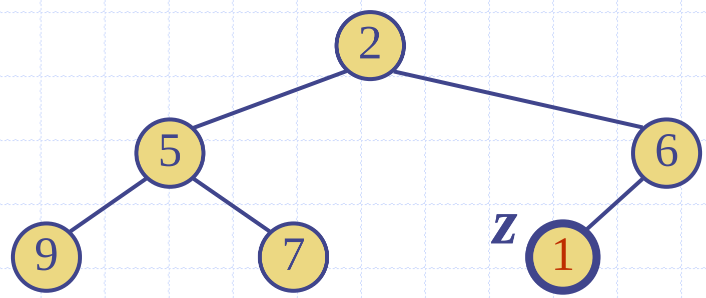
- We can **use a heap to implement a priority queue**
- We store a (key, element) item at each internal node
- We keep track of the position of the last node



Insertion into a Heap

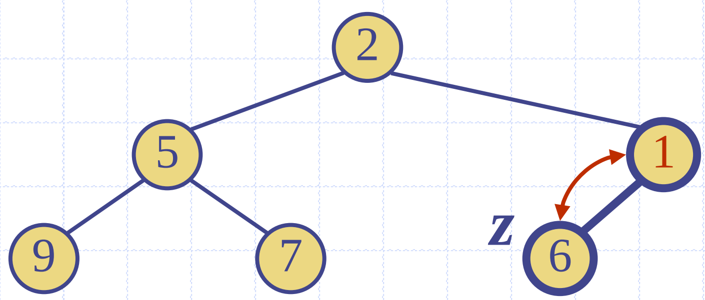
- Method *insertItem* of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z
  - Restore the heap-order property (discussed next)

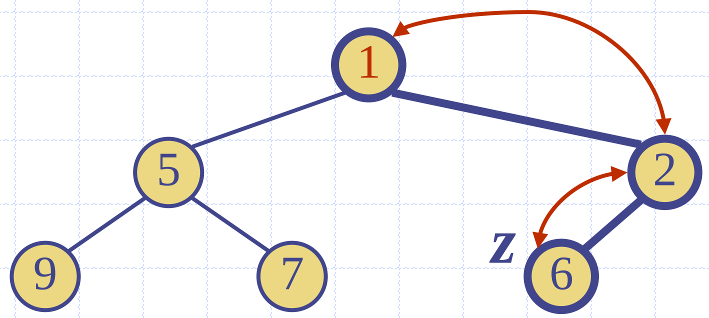




Upheap

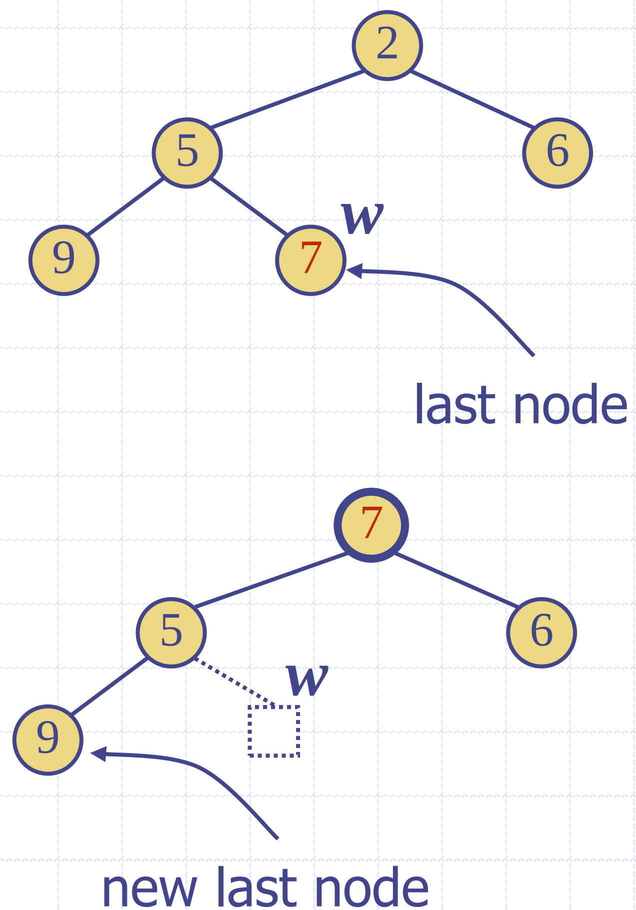
- After the insertion of a new key k, the heap-order property may be violated
- Algorithm **upheap** restores the heap-order property by **swapping k along an upward path from the insertion node**
- **Upheap** terminates when the key k reaches the root or a node whose parent has a key smaller than or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





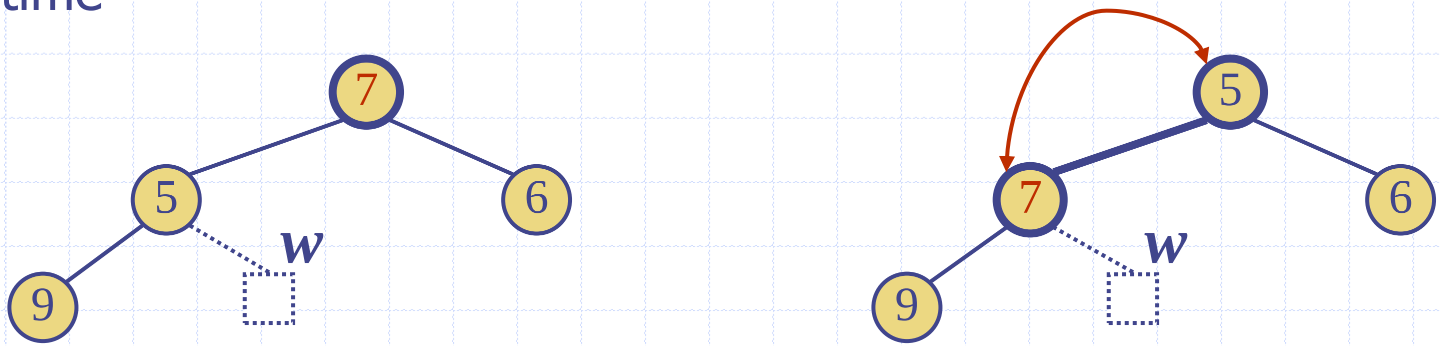
Removal from a Heap

- 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)



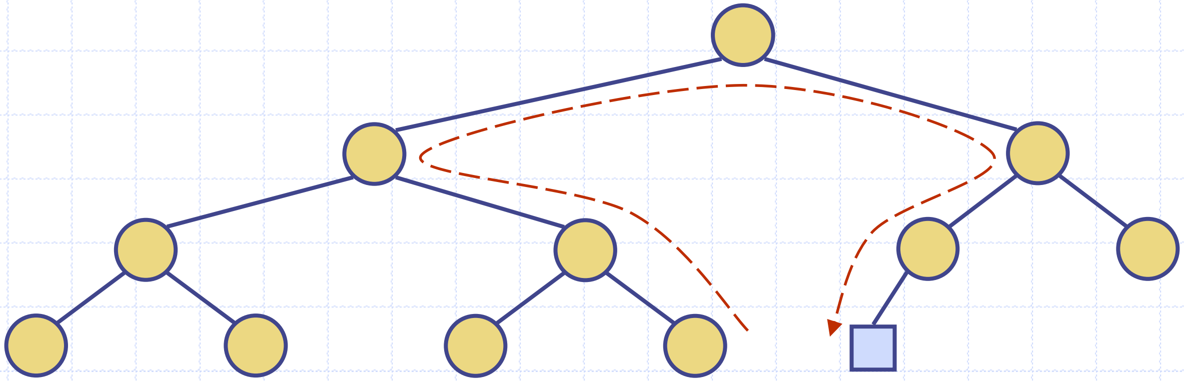
Downheap

- 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**
- **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



Keeping track of the "last" node and "insertion" node

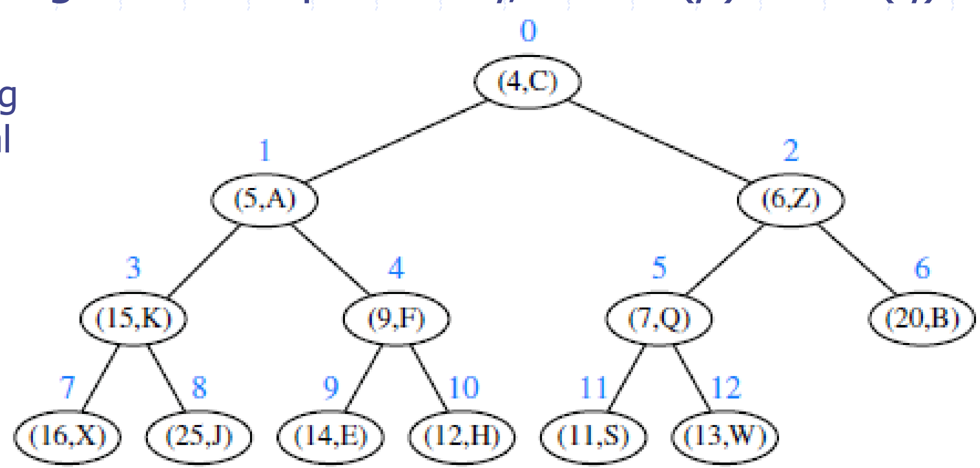
- The insertion node can be found by traversing a path of $O(\log n)$ nodes
  - Start with current last node
  - Go up until a left child or the root is reached
  - If a left child is reached, go to the corresponding right child
  - Go down left until a leaf is reached
- Similar algorithm for updating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



Array-based Heap Implementation

- We can represent a heap with n keys by means of an array of length n - the **element at position *p* is stored in *A* with index equal to the level number $f(p)$ of *p***, defined as follows:
  - If p is the root, then $f(p) = 0$.
  - If p is the left child of position q, then $f(p) = 2\,f(q)+1$.
  - If p is the right child of position q, then $f(p) = 2\,f(q)+2$.

Number nodes using level order traversal



(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

Figure 9.4: Array-based representation of a heap.

Array-based Heap Implementation

- Links between nodes are not explicitly stored
- Operation *add* corresponds to inserting at level $n + 1$
- Operation *remove_min* corresponds to removing at level n
- Yields **in-place heap-sort**
- 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 Implementation

```java
/** 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, 2

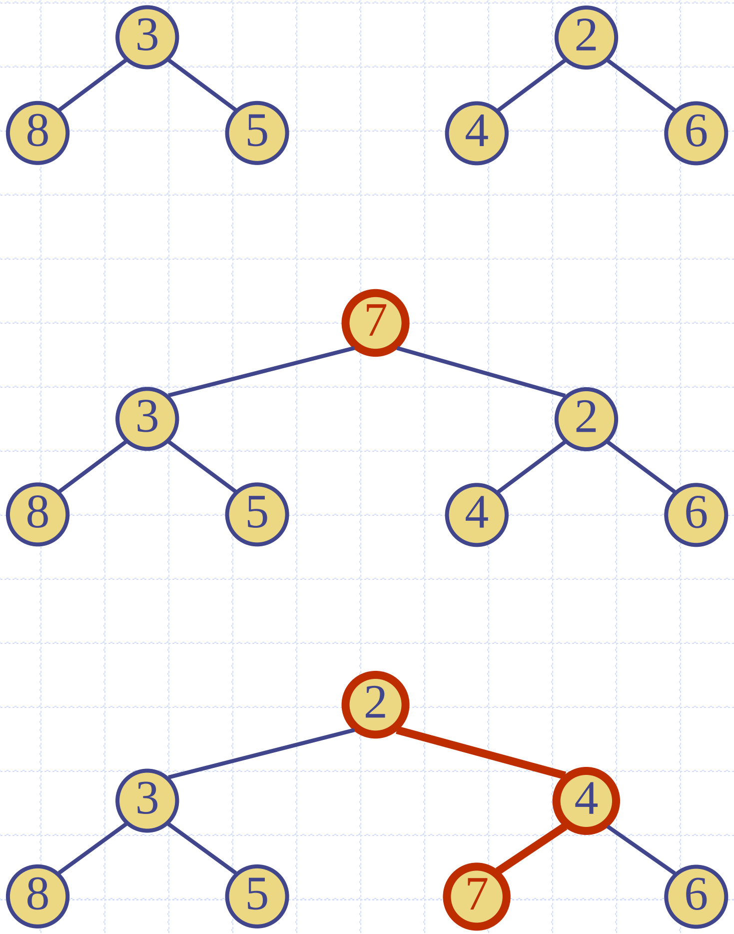
```
/** Moves the entry at index j lower, if necessary, to restore the heap property. */
protected void downheap(int j) {
  while (hasLeft(j)) {                    // continue to bottom (or break statement)
    int leftIndex = left(j);
    int smallChildIndex = leftIndex;      // although right may be 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

```java
  /** 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

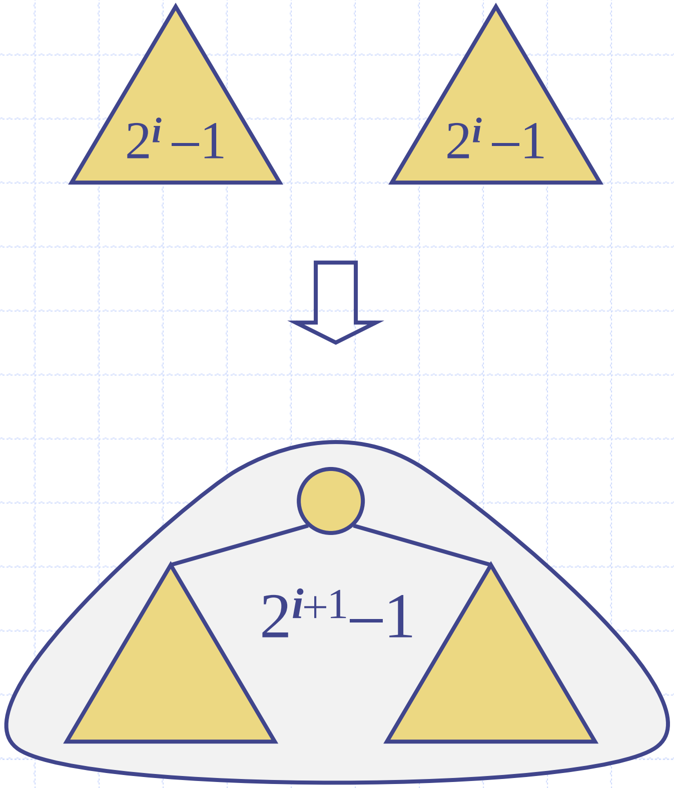


Heap Construction

- **Bottom-up**: Insert the keys in the given order using breadth-first and then fix it.
- **Top-down**: Insert the keys into an (initially) empty heap.

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 $(2^i - 1 + 2^i - 1 + 1)$



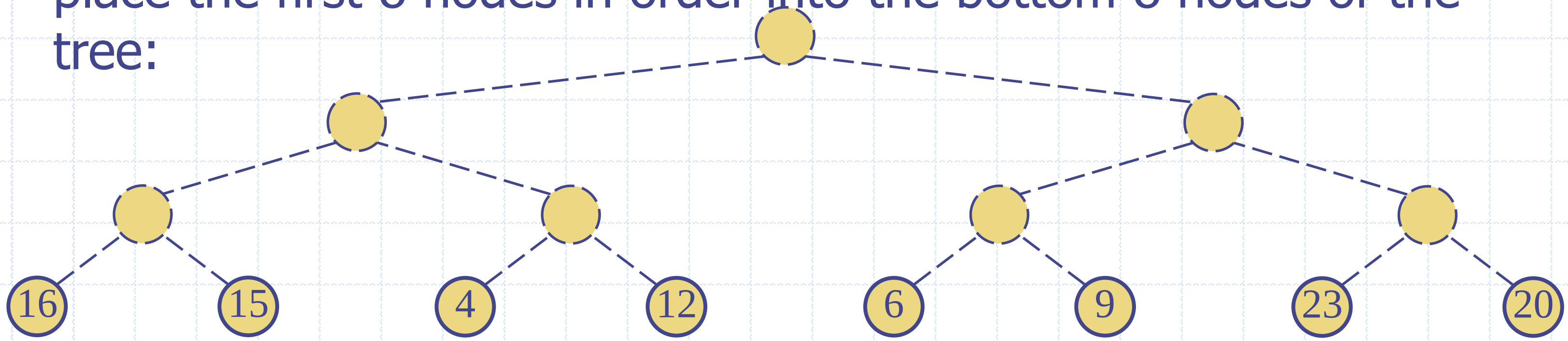
Bottom-up Heap Construction

- In this example, we'll form a heap from the following keys:

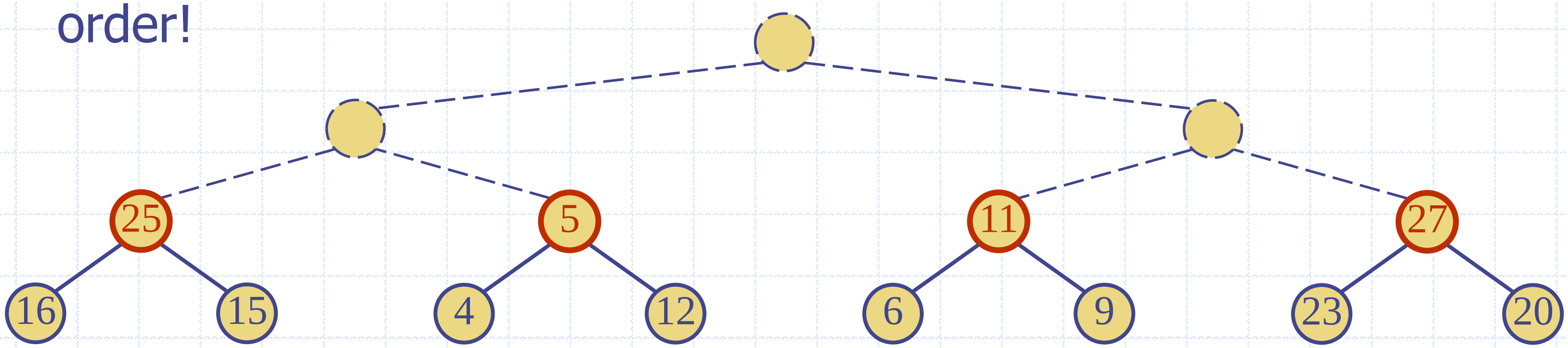
  16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10
- A breadth-first representation gives a 1-2-4-8 heap of 15 internal nodes.
- The bottom row of internal nodes will have 8 nodes; the next up will have 4; the next will have 2, and the last will have one, the root.
- The algorithm starts by placing the first 8 nodes, 16, 15, 4, 12, 6, 9, 23, 20, in that order into the bottom 8 nodes of the tree:

Example

- place the first 8 nodes in order into the bottom 8 nodes of the tree:

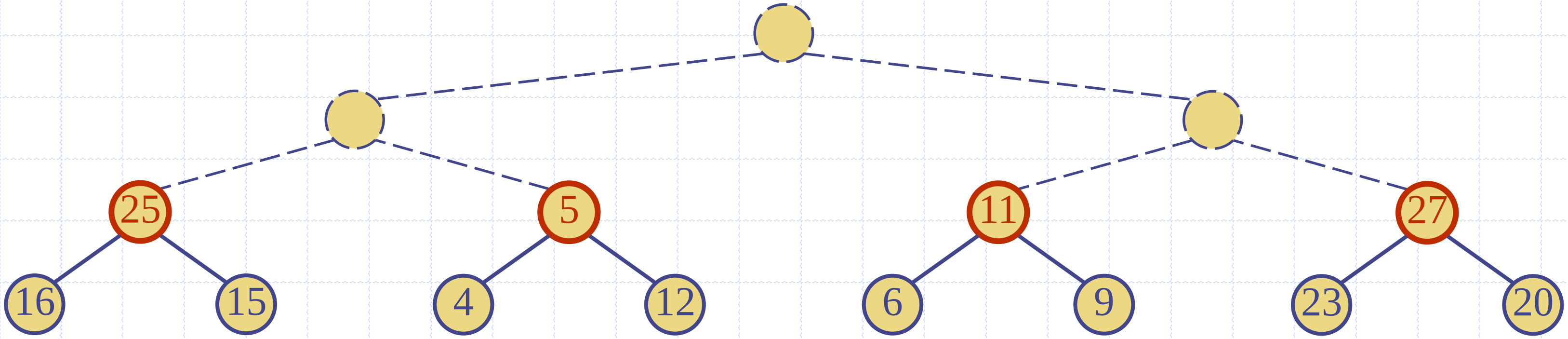


- insert the next 4 nodes into the next row up; this destroys heap order!

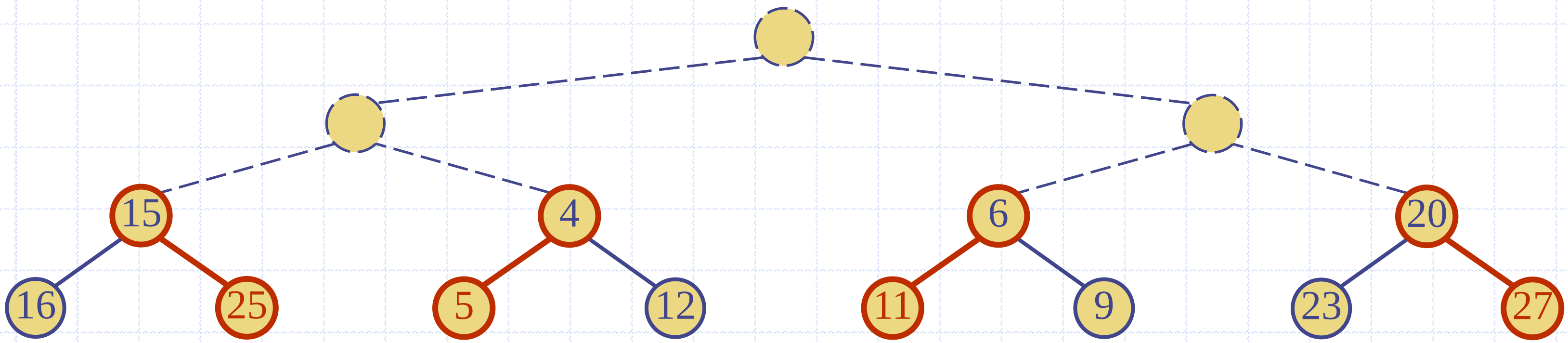


Example (contd.)

- Heap order is destroyed:

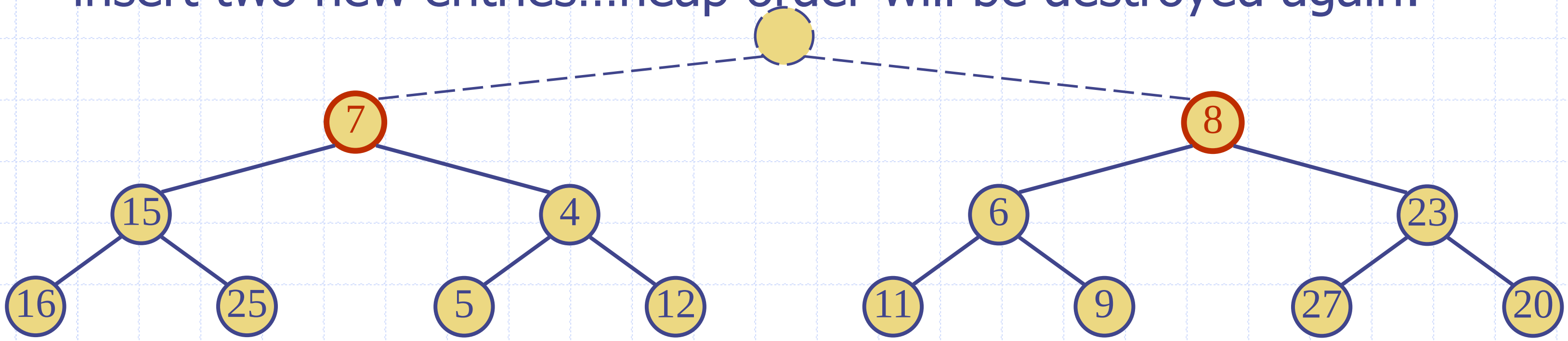


- apply “downheap” to the roots of each of the four trees:

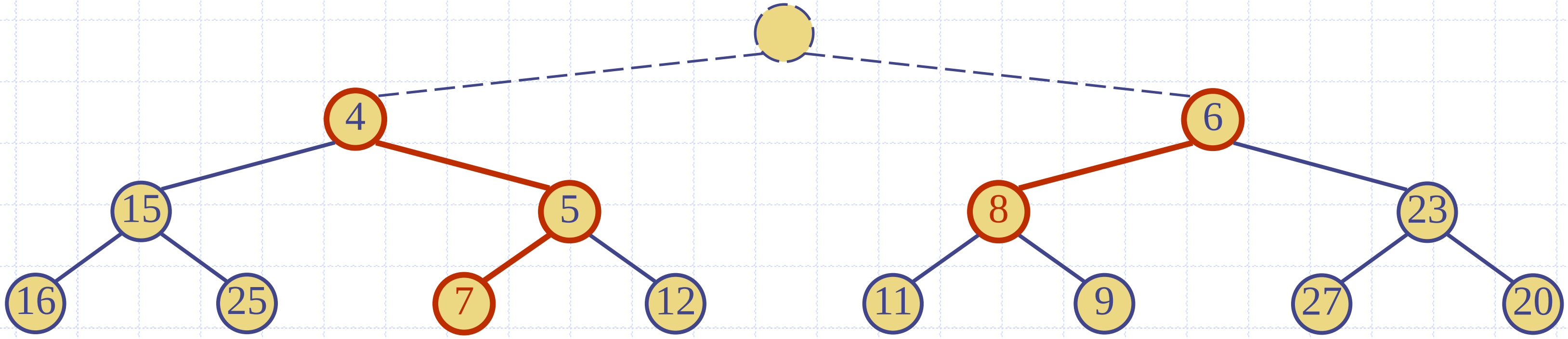


Example (contd.)

- insert two new entries...heap order will be destroyed again:

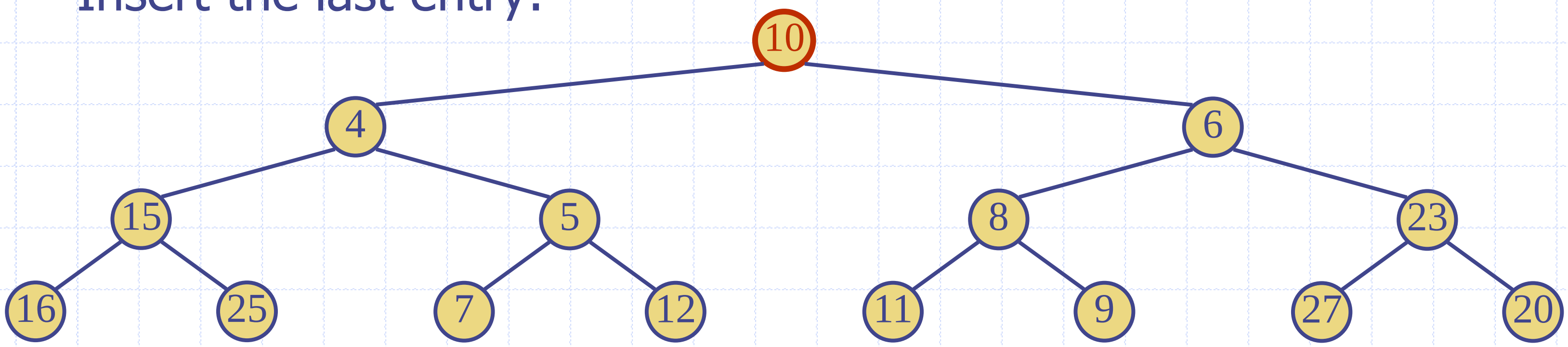


- Apply downheap again:

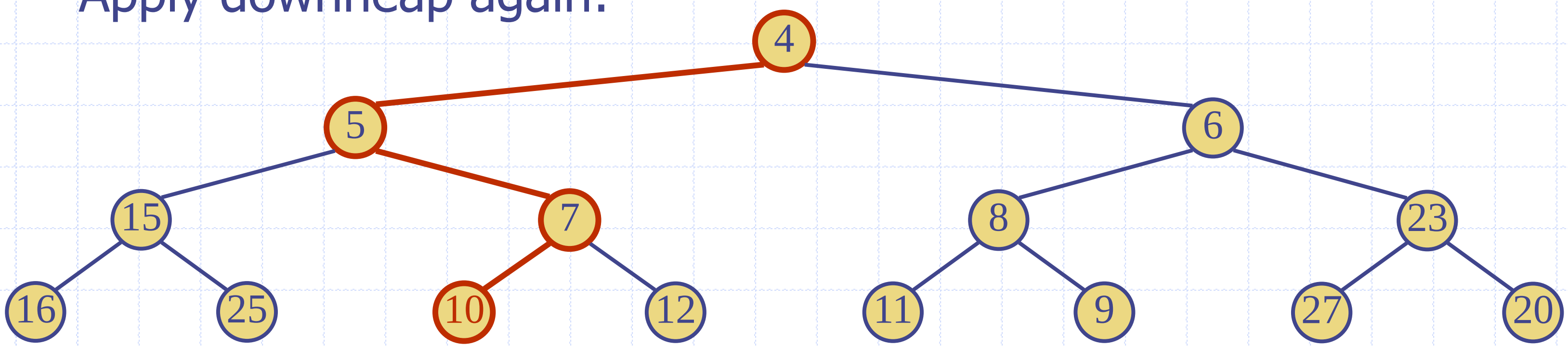


Example (end)

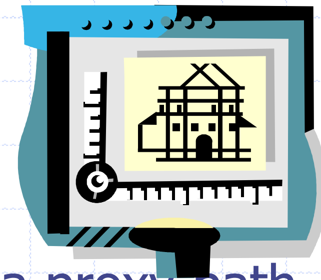
- Insert the last entry:



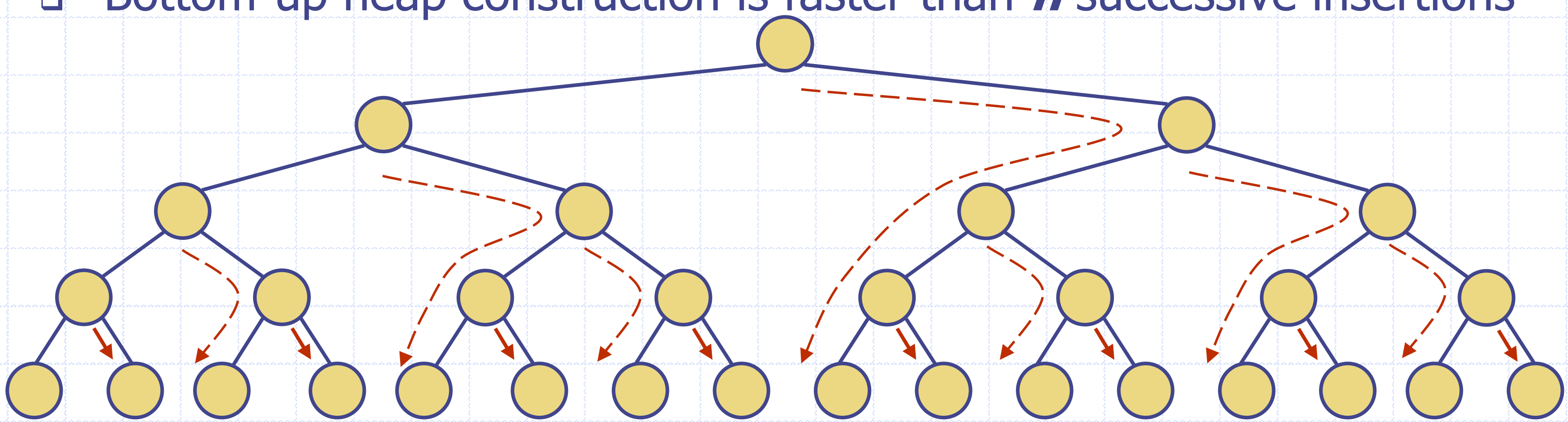
- Apply downheap again:



Analysis



- We visualize the worst-case time of a downheap with a proxy path that starts at the insertion location, 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** (involved in at most two swaps), 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



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*

$P \leftarrow$ priority queue with comparator C

while $\neg$*S.isEmpty* ()

  $e \leftarrow$ *S.remove*(*S.first* ())

  P.insert (*e*, $\emptyset$)

while $\neg$*P.isEmpty*()

  $e \leftarrow$ *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 (one per each element) 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		
(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 sequence**
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with n insert operations (one per each element) takes time proportional to

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

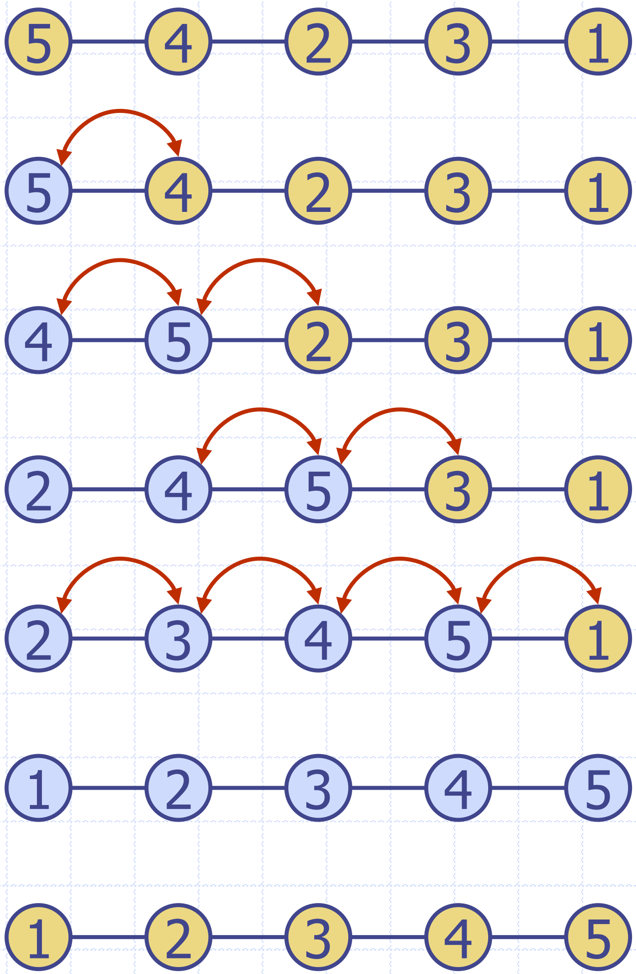
  2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

Insertion-Sort Example

	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)	()

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* ())

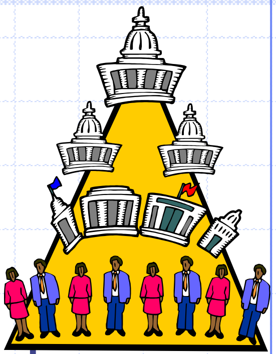
P.insert (*e*, *e*)

while ¬*P.isEmpty*()

e ← *P.removeMin*().*getKey*()

S.addLast(*e*)

Heap-Sort



- Consider a priority queue with n items implemented by means of a heap
  - Insert elements in a heap
  - removeMin elements in no decreasing order
  - 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**