Priority Queues & Heaps

Outline

- The Priority Queue ADT
- Total orderings, the Comparable Interface and the Comparator Class
- Heaps
- Adaptable Priority Queues

Outcomes

- By understanding this lecture, you should be able to:
  - □ Explain and design a priority queue ADT
  - □ Describe suitable applications for priority queues
  - □ Design and implement a heap
  - □ Analyze the run time of a heap
  - □ Identify advantages of and suitable applications for heaps
  - □ Design and implement an efficient adaptable priority queue ADT using location-aware entries
  - □ Identify suitable applications for an adaptable priority queue

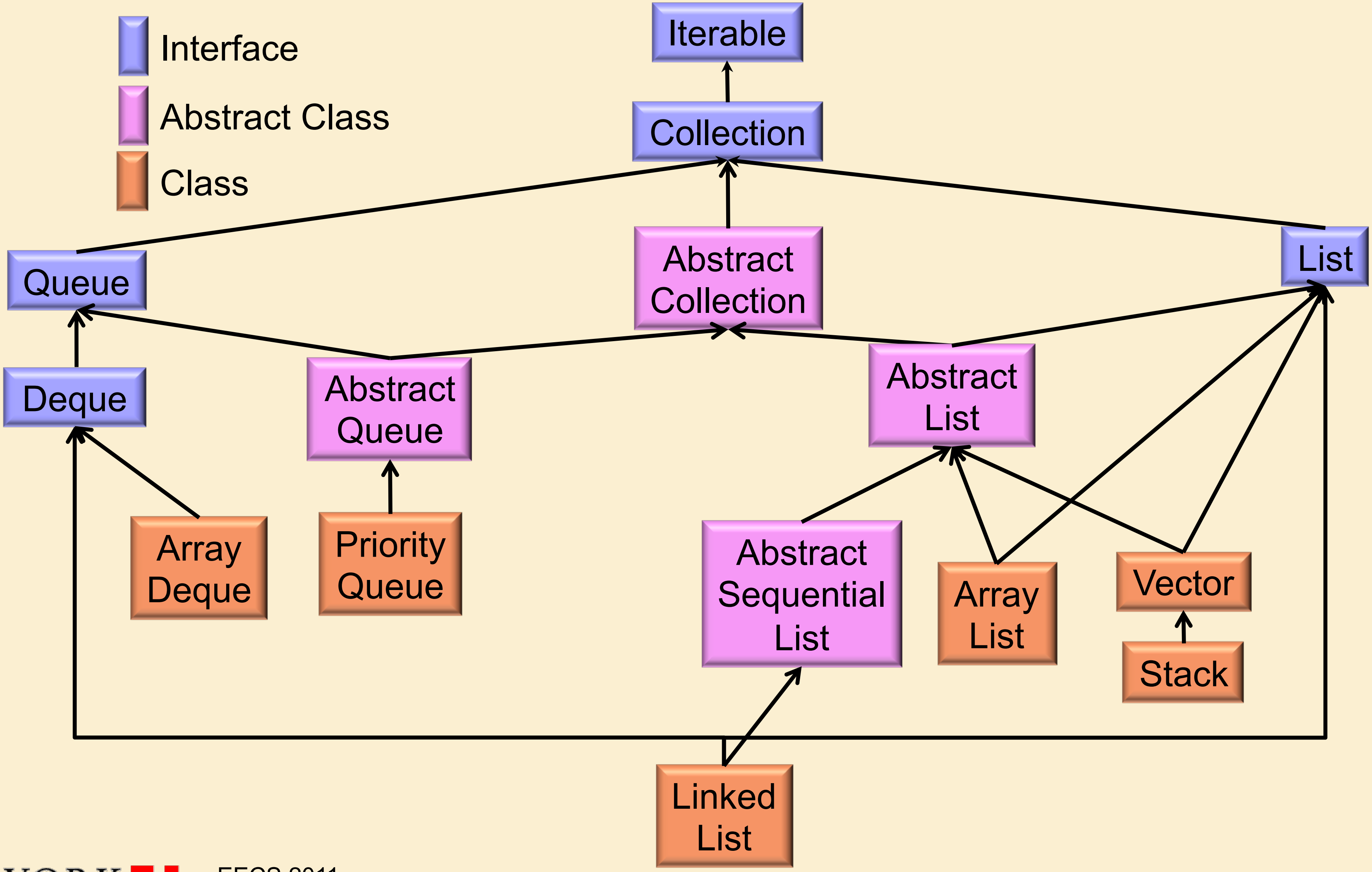
Outline

- **The Priority Queue ADT**
- Total orderings, the Comparable Interface and the Comparator Class
- Heaps
- Adaptable Priority Queues

Priority Queue ADT

- A priority queue stores a collection of **entries**
- Each **entry** is a pair (key, value)
- Entries with **smaller** keys have higher priority.
- 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:
  - Process scheduling
  - Standby flyers

The Java Collections Framework (Ordered Data Types)



🟦 Interface

🟪 Abstract Class

🟧 Class

Iterable

Collection

Queue

Abstract Collection

List

Deque

Abstract Queue

Abstract List

Array Deque

Priority Queue

Abstract Sequential List

Array List

Vector

Stack

Linked List

The **Priority Queue** Class

- Based on priority min-heap
- Elements are prioritized based either on
  - natural order
  - a **comparator**, passed to the constructor.
- **Provides an iterator**

Priority Queue ADT (Textbook)	**PriorityQueue Class (java.util)**
insert(k,v)	add(**new** SimpleEntry(k,v))
removeMin()	remove()
min()	peek()

Outline

- The Priority Queue ADT
- **Total orderings, the Comparable Interface and the Comparator Class**
- Heaps
- Adaptable Priority Queues

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 ≤
  - Reflexive property: $x \leq x$
  - Antisymmetric property: $x \leq y \wedge y \leq x \rightarrow x = y$
  - Transitive property: $x \leq y \wedge y \leq z \rightarrow x \leq z$

Entry ADT

- An **entry** in a priority queue is simply a key-value pair
- Methods:
  - ❑ **getKey**(): returns the key for this entry
  - ❑ **getValue**(): returns the value for this entry

- As a Java interface:

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

The Comparable Interface

- Part of the Collections Framework in java.util.
- Imposes a total ordering on the objects of a class that implements it.
- Objects can be compared using the compareTo method.
- obj1.compareTo(obj2) returns
  - Negative integer if obj1 < obj2
  - Positive integer if obj1 > obj2
  - 0 if obj1 = obj2

Alternative: 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
- The 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.

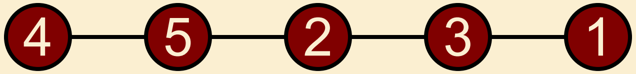
Example Comparator

```
/** 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  (xa - xb);
       else
                return  (ya - yb);
   }
}
```

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

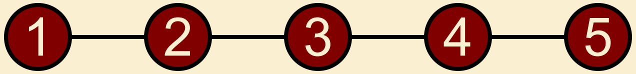
Sequence-based Priority Queue

- Implementation with an unsorted list



- 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 right place to insert the item
  - □ **removeMin** and **min** take $O(1)$ time, since the smallest key is at the beginning

Is this tradeoff inevitable?

Outline

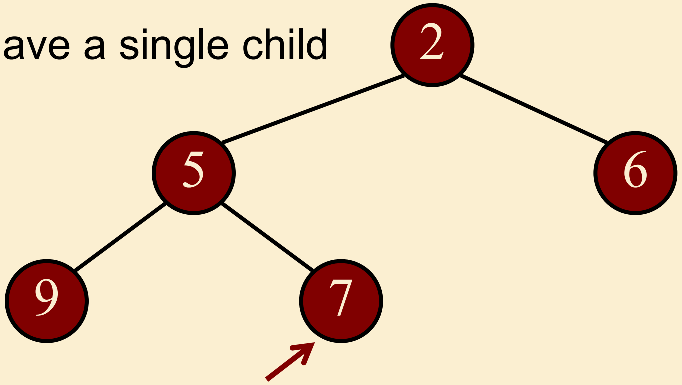
- The Priority Queue ADT
- Total orderings, the Comparable Interface and the Comparator Class
- **Heaps**
- Adaptable Priority Queues

Heaps

- Goal:
  - ❑ O(log n) insertion
  - ❑ O(log n) removal
- Remember that O(log n) is almost as good as O(1)!
  - ❑ e.g., n = 1,000,000,000 → log n ≅ 30
- There are min heaps and max heaps. We will assume min heaps.

Min Heaps

- A min 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
    - ✧ $\boldsymbol{key}(\boldsymbol{v}) \geq \boldsymbol{key}(\boldsymbol{parent}(\boldsymbol{v}))$
  - ❑ Complete binary tree: let $\boldsymbol{h}$ be the height of the heap
    - ✧ for $\boldsymbol{i} = 0, \dots, \boldsymbol{h} - 1$, there are 2^i nodes of depth $\boldsymbol{i}$
    - ✧ at depth $\boldsymbol{h} - 1$
      - ❖ the internal nodes are to the left of the external nodes
      - ❖ Only the rightmost internal node may have a single child



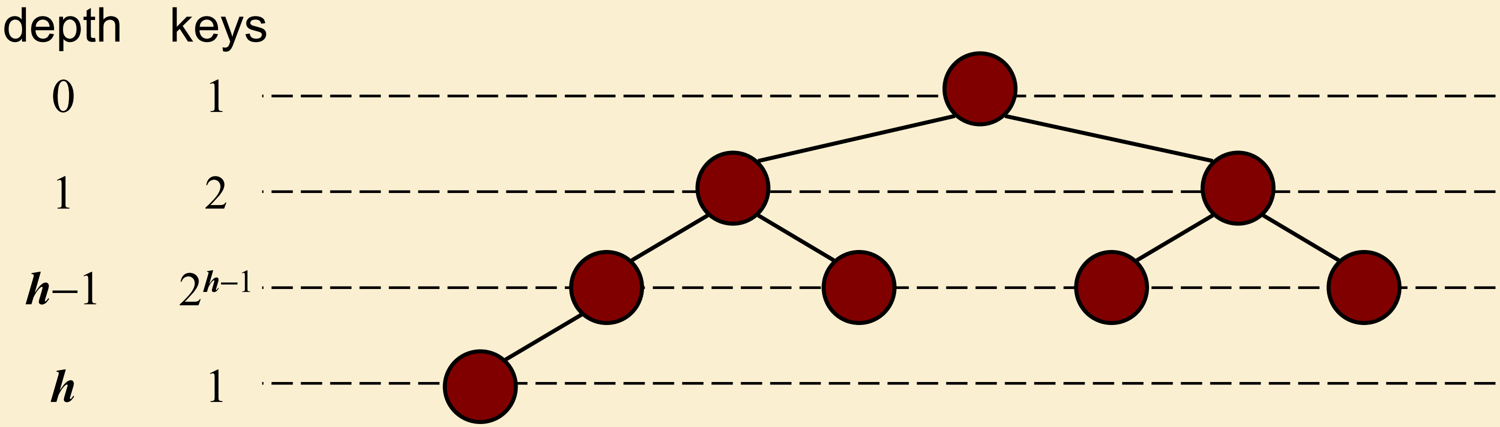
❑ **The last node of a heap is the rightmost node of depth $\boldsymbol{h}$**

Height of a Heap

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

  Proof: (we apply the complete binary tree property)

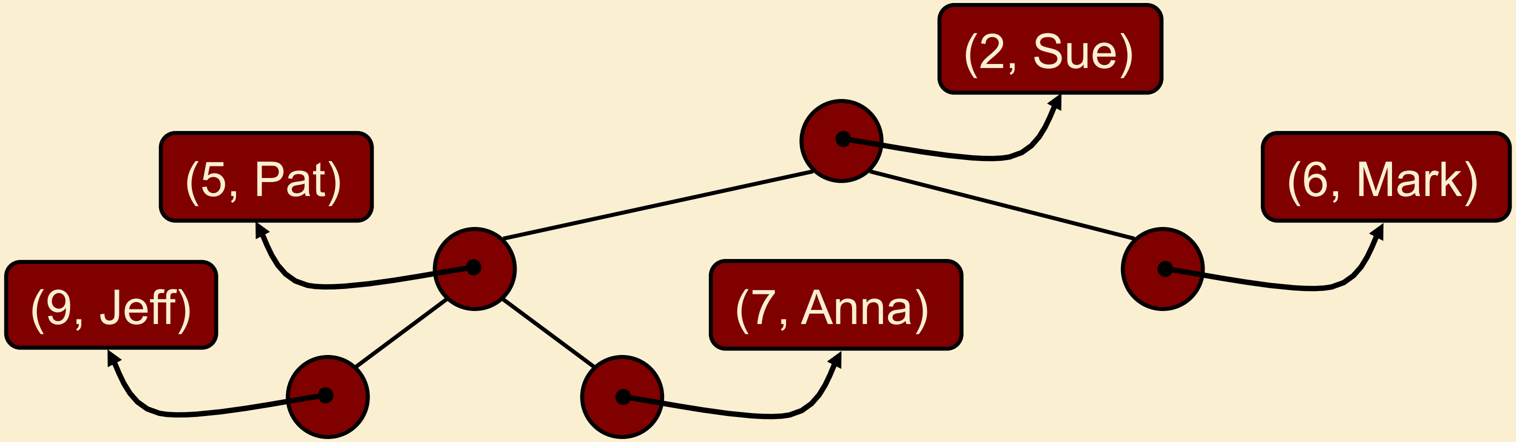
  - ❑ Let $\boldsymbol{h}$ be the height of a heap storing $\boldsymbol{n}$ keys
  - ❑ Since there are 2^i keys at depth $\boldsymbol{i} = 0, \ldots, \boldsymbol{h} - 1$ and at least one key at depth $\boldsymbol{h}$, we have $\boldsymbol{n} \geq 1 + 2 + 4 + \ldots + 2^{\boldsymbol{h}-1} + 1$
  - ❑ Thus,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



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

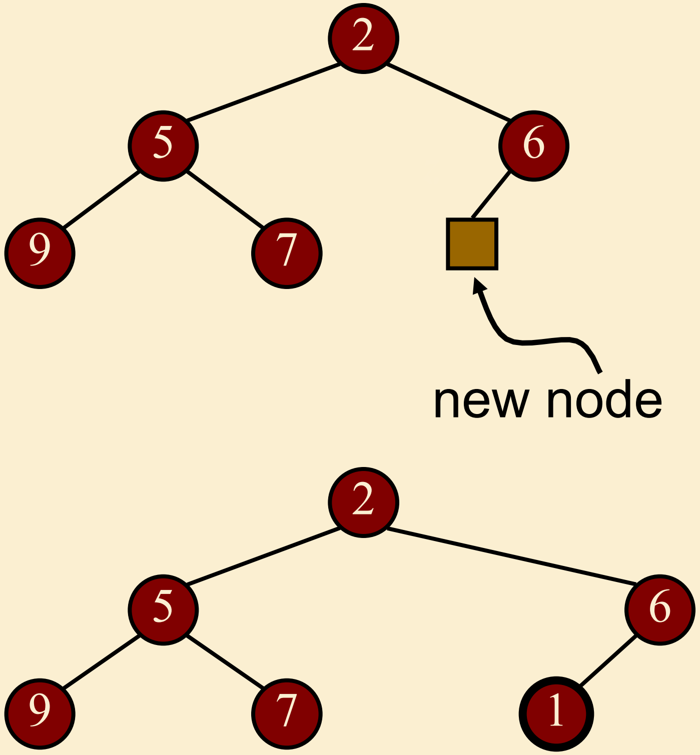
Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each node
- We keep track of the position of the last node
- For simplicity, we will typically show only the keys in the pictures



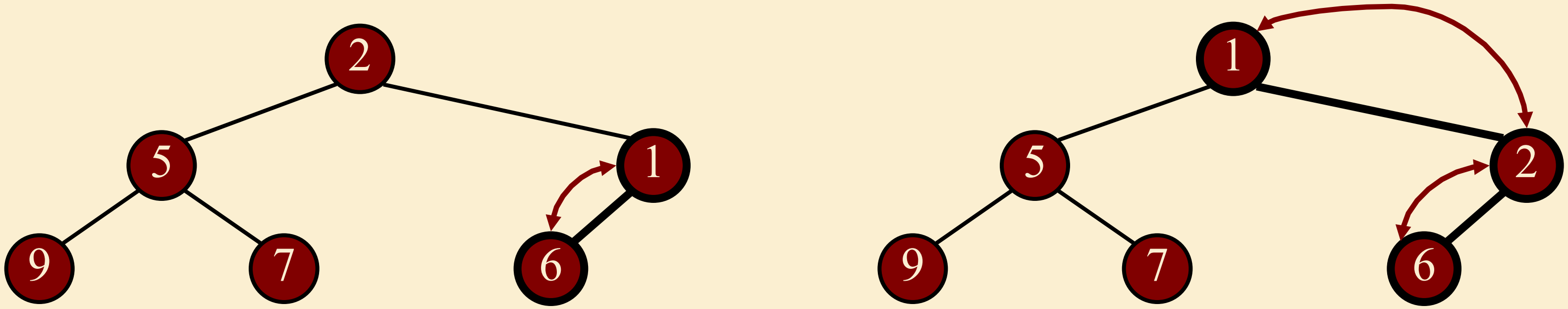
Insertion into a Heap

- Method **insert** of the priority queue ADT involves inserting a new entry with key ***k*** into the heap
- The insertion algorithm consists of two steps
  - Store the new entry at the next available location
  - Restore the heap-order property



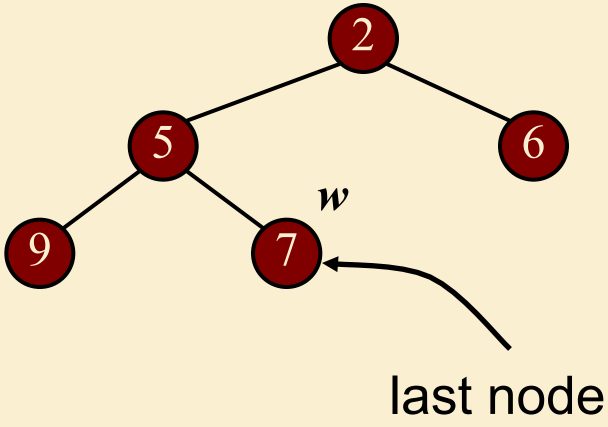
Upheap

- 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

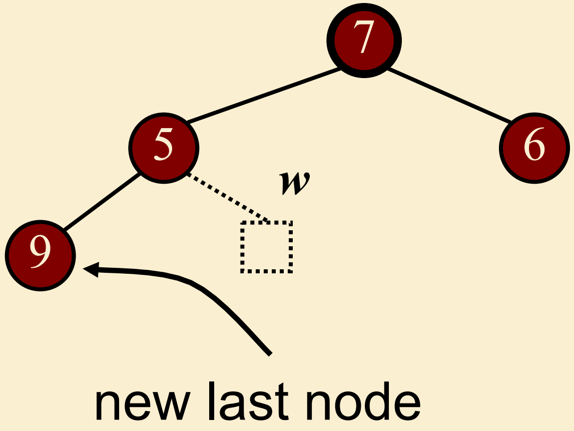


Removal from a Heap

- 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



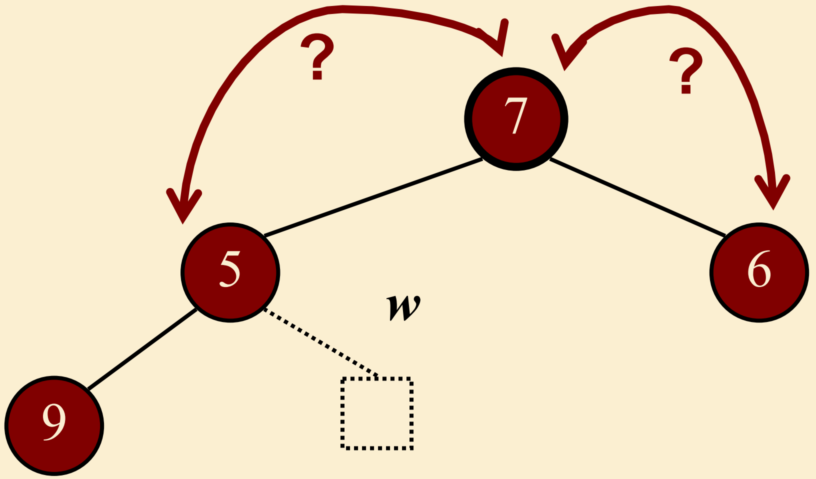
last node



new last node

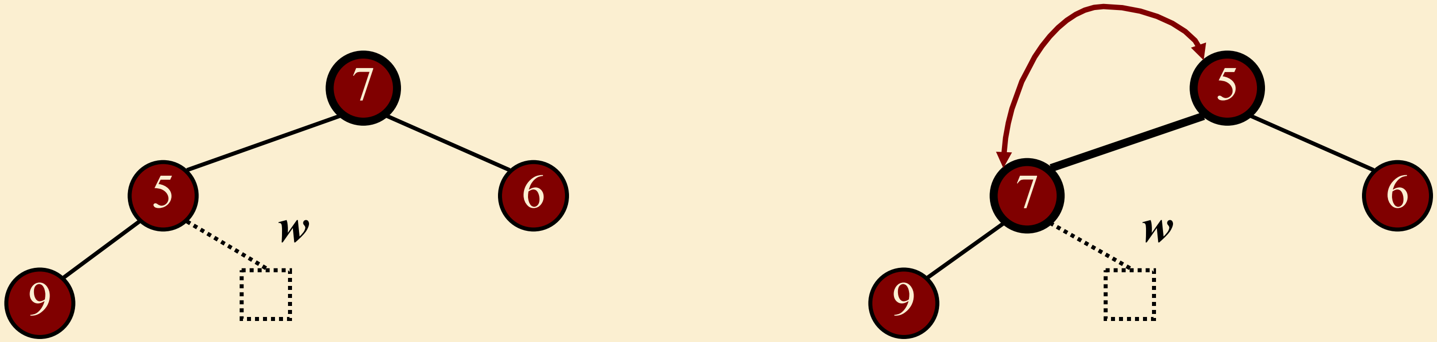
Downheap

- 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
- Note that there are, in general, many possible downward paths – which one do we choose?



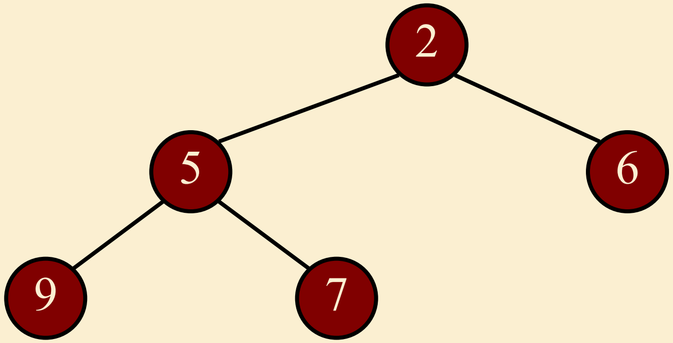
Downheap

- We select the downward path through the **minimum-key** nodes.
- 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



Array-based Heap Implementation

- Since a heap is a binary tree, we can represent a heap with n keys by means of an array of length $n + 1$
- Links between nodes are not explicitly stored
- The cell at rank 0 is not used
- The root is stored at rank 1.
- For the node at rank i
  - ❑ the left child is at rank $2i$
  - ❑ the right child is at rank $2i + 1$
  - ❑ the parent is at rank **floor**(i/2)
  - ❑ if 2i + 1 > n, the node has no right child
  - ❑ if 2i > n, the node is a leaf



	2	5	6	9	7
0	1	2	3	4	5

End of Lecture

Feb 6, 2018

Constructing a Heap

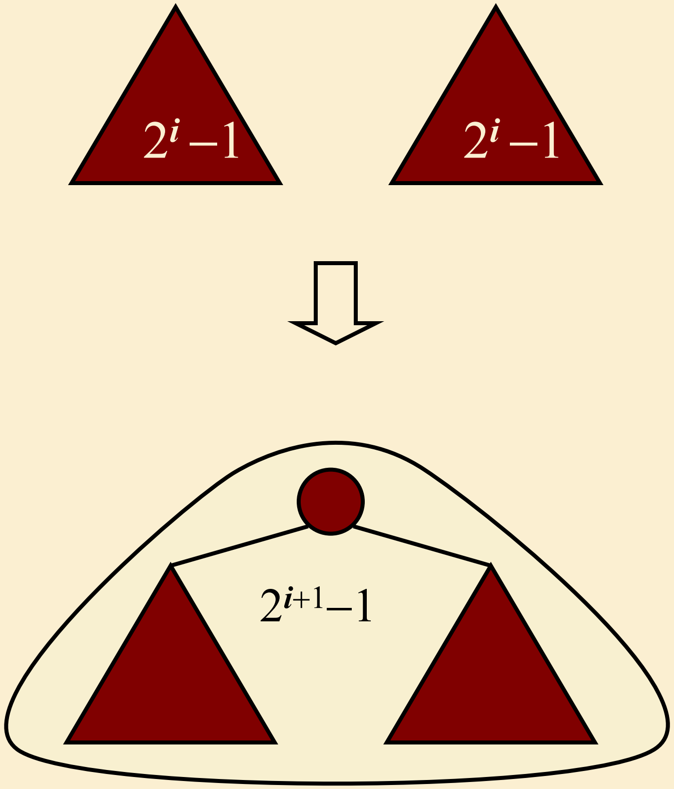
- A heap can be constructed by iteratively inserting entries: example.
- What is the running time?

$$T(n) \propto \sum_{i=1}^{n} \log i \in \theta\left(n \log n\right).$$

- Can we do better?
- **Yes** – if all of the key-value pairs are given in advance.

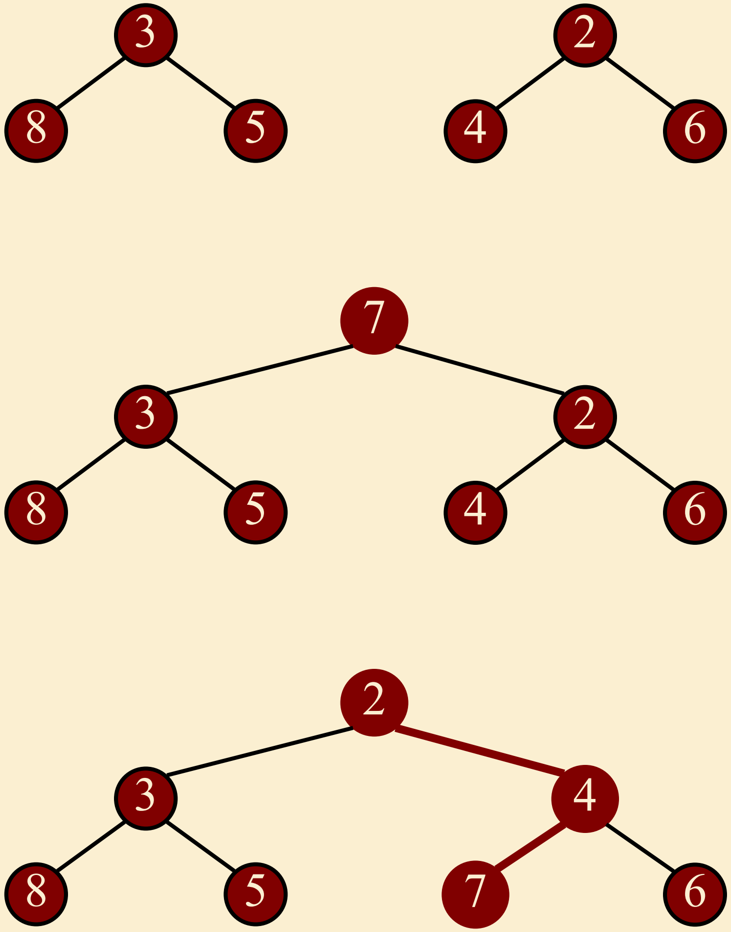
Bottom-up Heap Construction

- We can construct a heap storing *n* keys using a bottom-up construction with log *n* phases
- For simplicity, assume that $n = 2^{h+1}-1$.
- In phase *i*, each pair of heaps with $2^i -1$ keys are merged with an additional node into a heap with $2^{i+1}-1$ keys



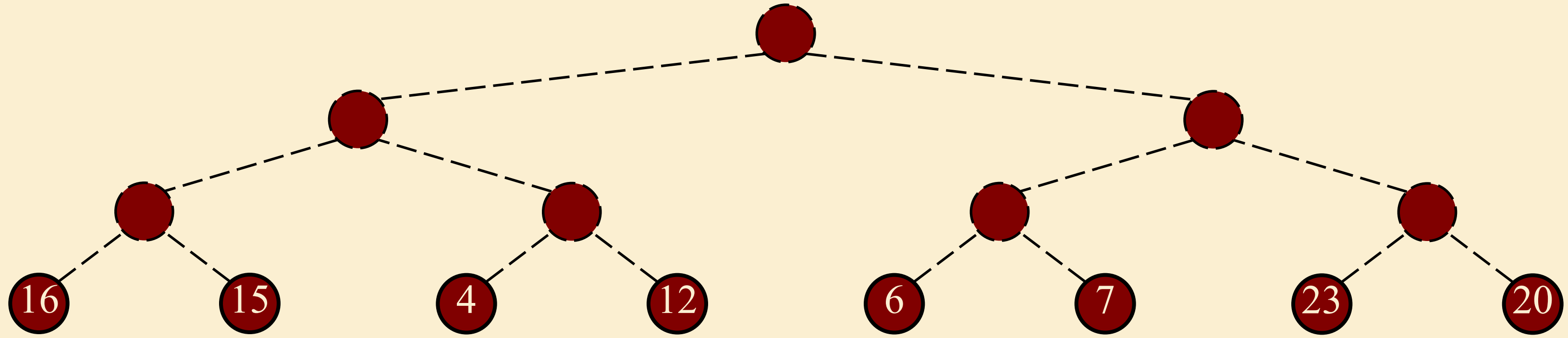
Merging Two Heaps

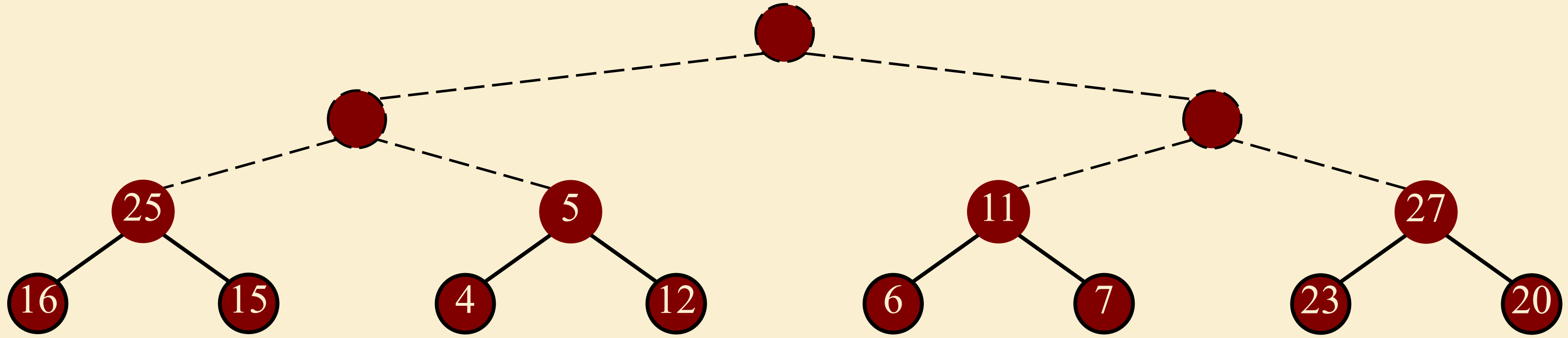
- We are given two heaps and a new key ***k***
- We create a new heap with the root node storing ***k*** and with the two heaps as subtrees
- We perform downheap to restore the heap-order property



Example (15 entries)

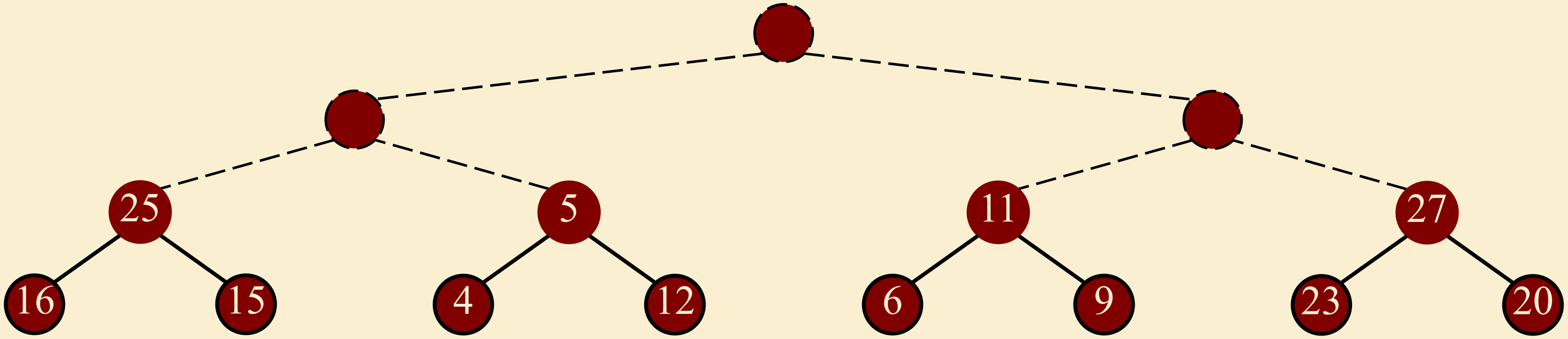
Phase 1. (n+1)/2 heaps of size 1

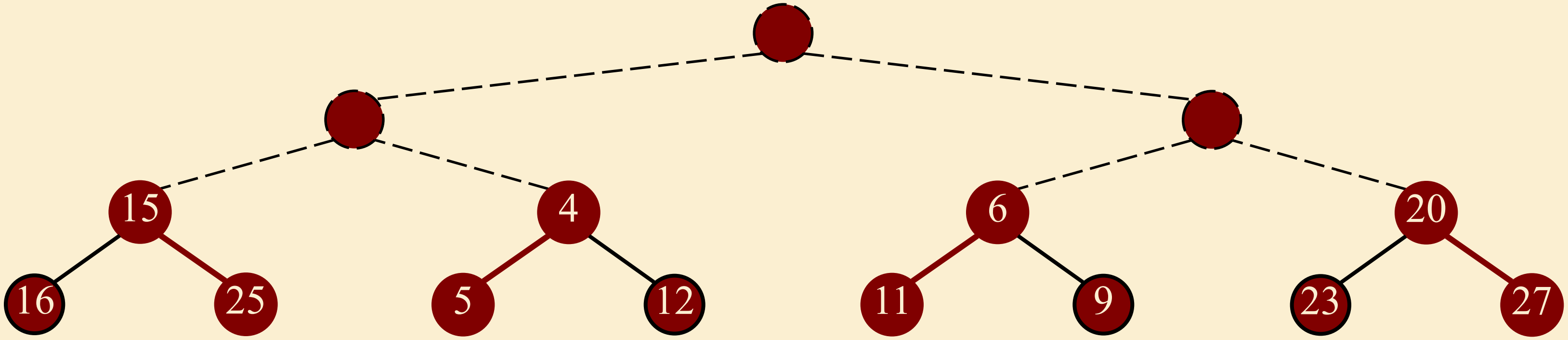




Example (contd.)

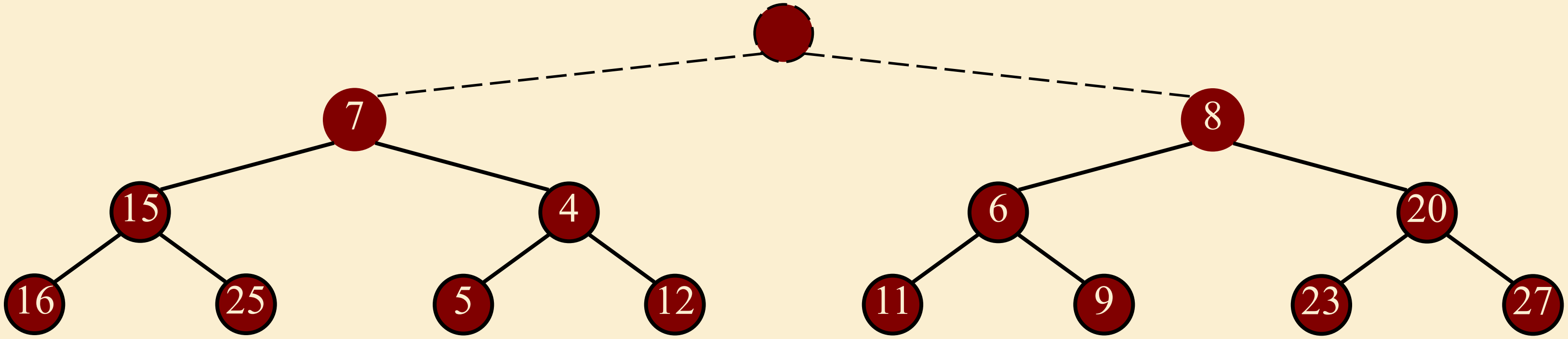
Phase 2. (n+1)/4 heaps of size 3

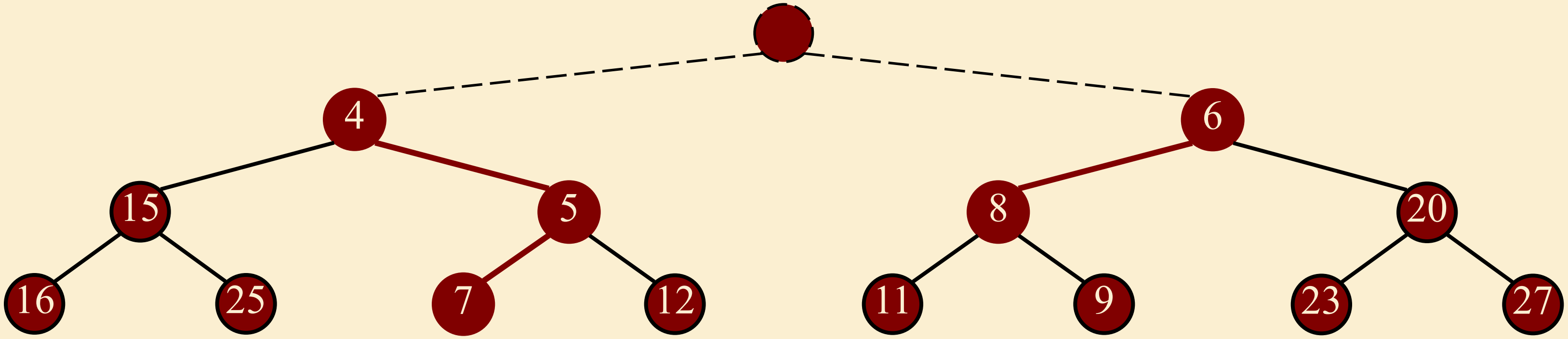




Example (contd.)

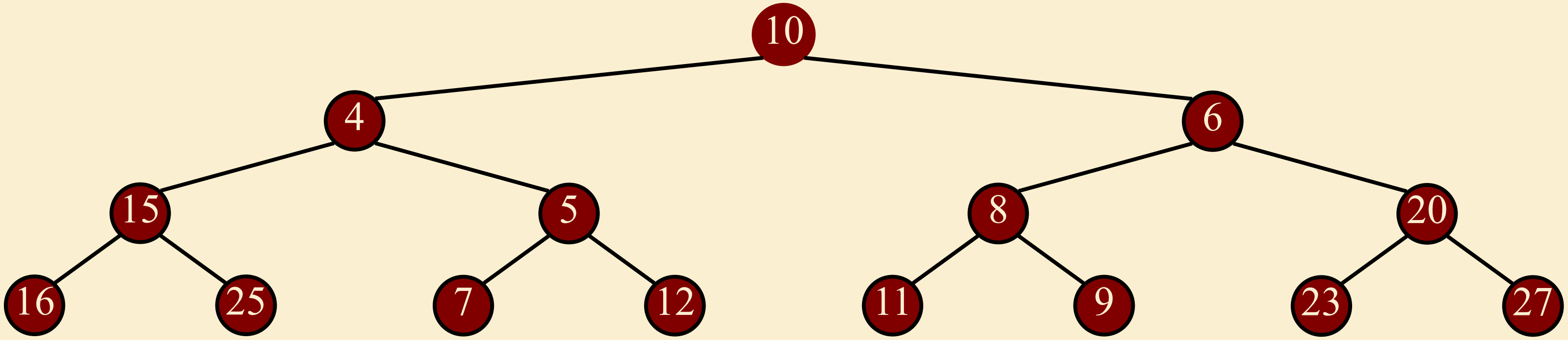
Phase 3. (n+1)/8 heaps of size 7

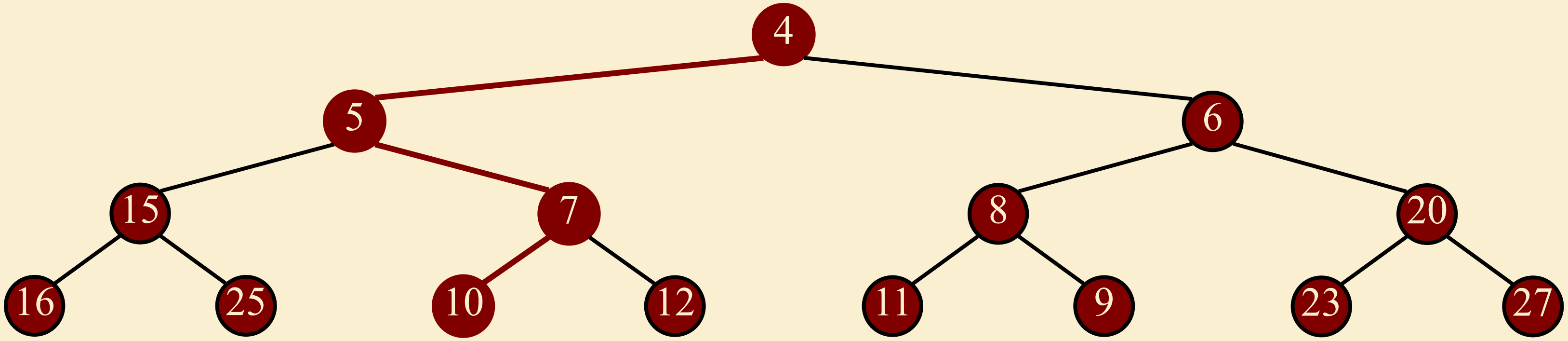




Example (end)

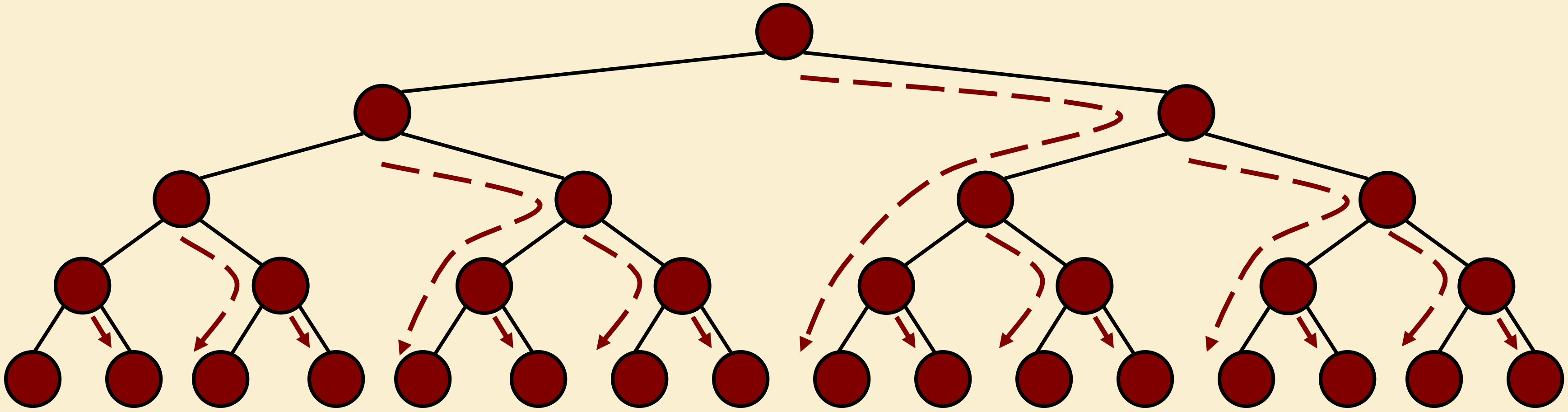
Phase 4. (n+1)/16 heaps of size 15





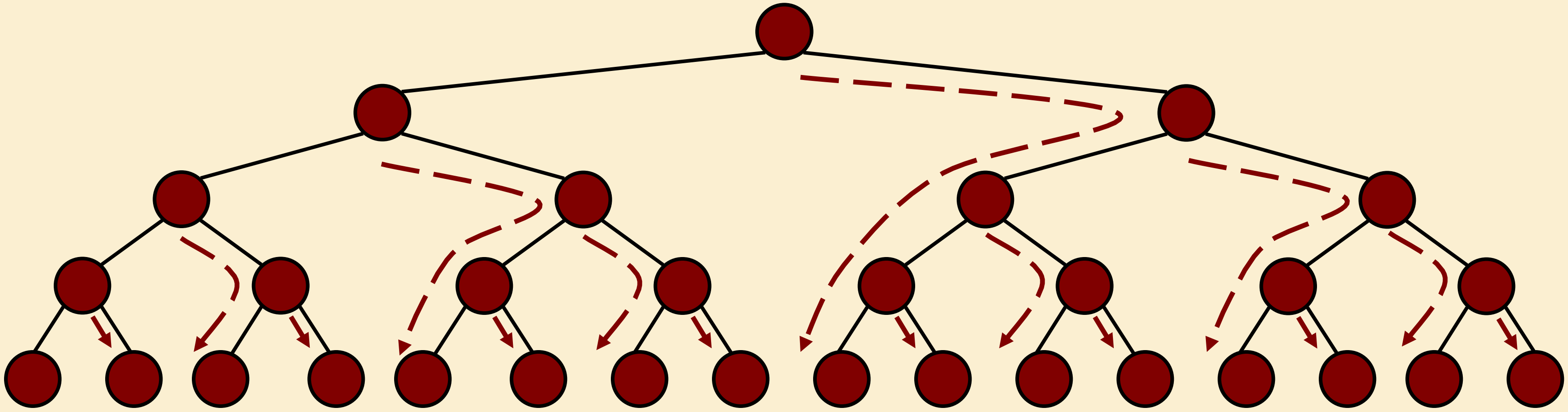
Bottom-Up Heap Construction Analysis

- In the worst case, each added node gets downheaped to the bottom of the heap.
- We analyze the run time by considering the total length of these downward paths through the binary tree as it is constructed.
- For convenience, we can assume that each path first goes right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path, but this will not change the run time)



Analysis

- Each internal node originates at most one downheap path, of the form RLL…L.
- A node's path will never intersect a path originated by one if its children as it will be going L as the child goes R.
- Since each node is traversed by at most two paths, the total length of the paths is $O(n)$
- Thus, bottom-up heap construction runs in $O(n)$ time
- Bottom-up heap construction is faster than n successive insertions (***O(nlogn)***).



Bottom-Up Heap Construction

- Uses downHeap to reorganize the tree from bottom to top to make it a heap.
- Can be written concisely in either recursive or iterative form.

Loop Invariants

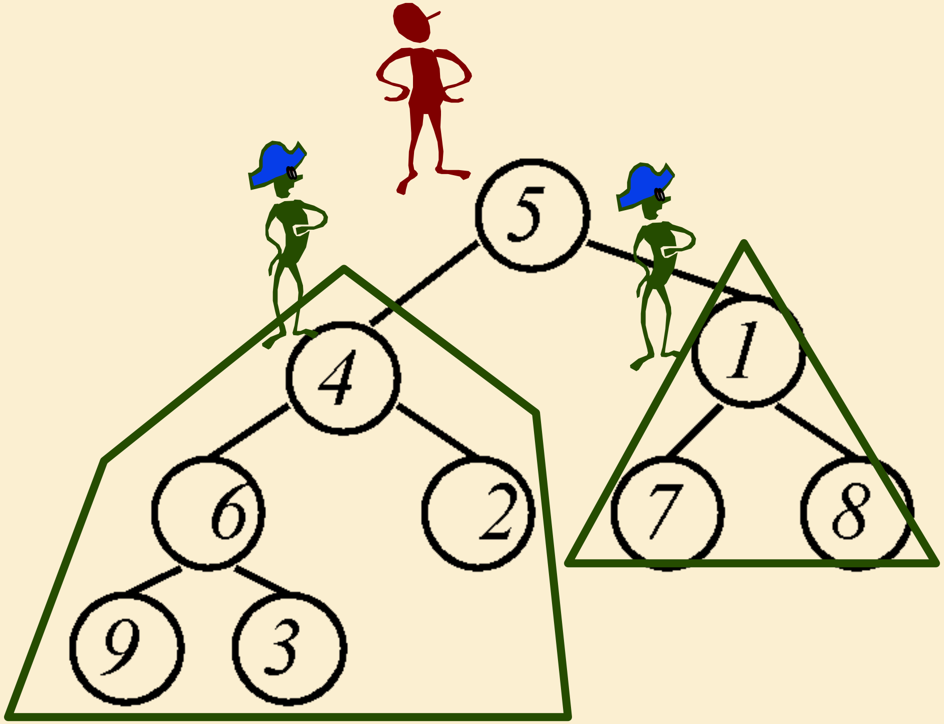
- **Loop Invariant (LI):** An **assertion** about the current state useful for designing, analyzing and proving the correctness of iterative algorithms.

Iterative MakeHeap

```
MakeHeap($A,n$)
<pre-cond>:A[1…n] is a complete binary tree
<post-cond>:A[1…n] is a heap
for $i \leftarrow \lfloor n/2 \rfloor$ downto 1
    $< LI >$: All subtrees rooted at $i+1 \ldots n$ are heaps
    $DownHeap(A,i,n)$
```

Recursive MakeHeap

Get help from friends



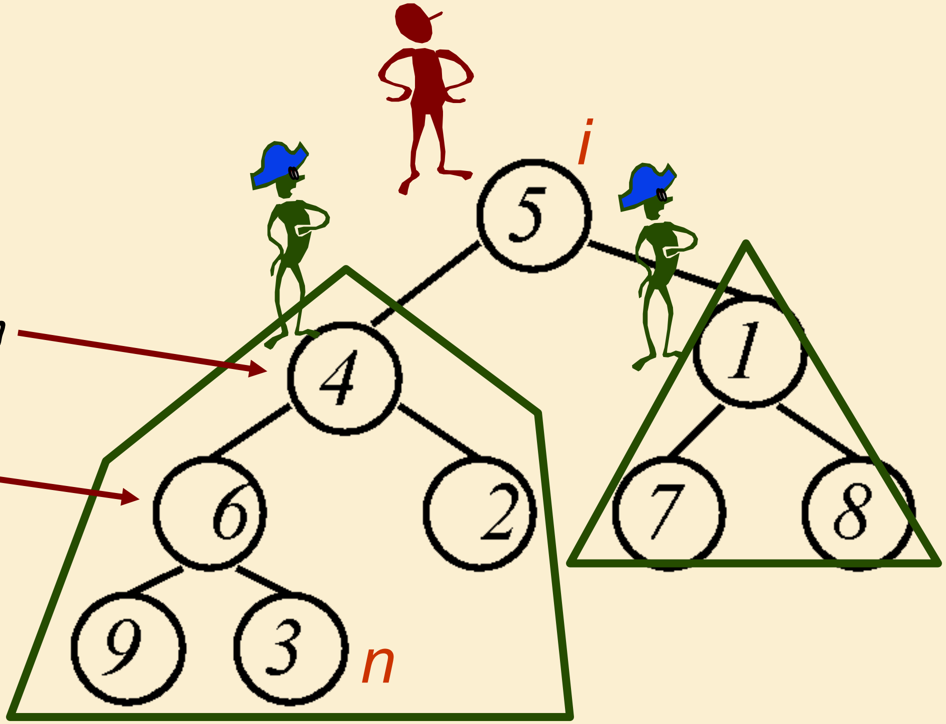
Recursive MakeHeap

Invoke as MakeHeap (*A*, *1*, *n*)

```
MakeHeap(A,i,n)
<pre-cond>:A[i…n] is a complete binary tree
<post-cond>:The subtree rooted at i is a heap
if $i \le n/4$ then
    MakeHeap(A,LEFT(i),n)
    MakeHeap(A,RIGHT(i),n)
Downheap(A,i,n)
```

$\lfloor n/4 \rfloor$ is grandparent of n

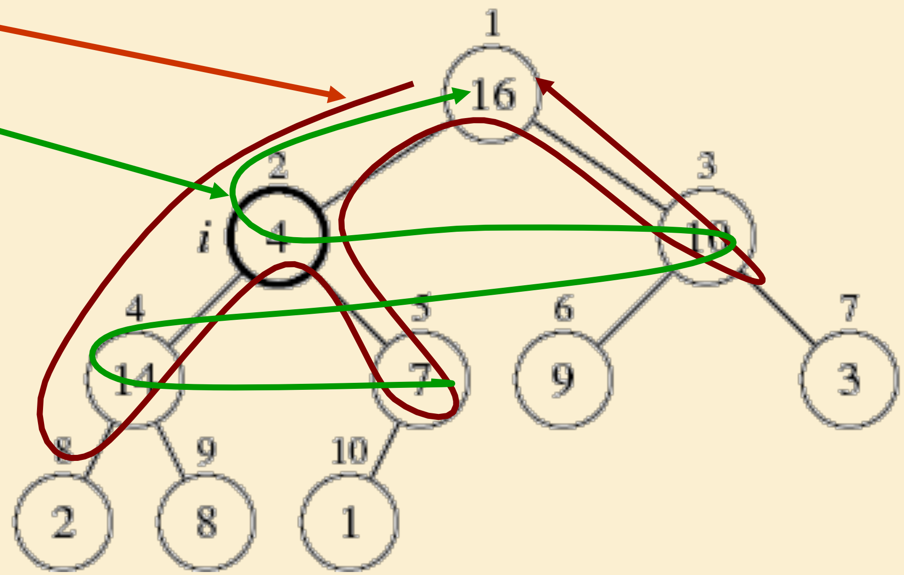
$\lfloor n/2 \rfloor$ is parent of n



Iterative and recursive methods perform exactly the same downheaps but in a different order. Thus both constructions methods are *O(n)*.

Iterative vs Recursive MakeHeap

- Recursive and Iterative MakeHeap do essentially the same thing: Heapify from bottom to top.
- Difference:
  - ❑ Recursive is “depth-first”
  - ❑ Iterative is “breadth-first”



Outline

- The Priority Queue ADT
- Total orderings, the Comparable Interface and the Comparator Class
- Heaps
- **Adaptable Priority Queues**

Recall the Entry and Priority Queue ADTs

- An **entry** stores a (key, value) pair within a data structure
- Methods of the entry ADT:
  - □ 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, v)
    inserts an entry with key k and value v
  - □ removeMin()
    removes and returns the entry with smallest key
  - □ min()
    returns, but does not remove, an entry with smallest key
  - □ size(), isEmpty()

Finding an entry in a heap by key

- Note that we have not specified any methods for removing or updating an entry with a specified key.
- These operations require that we first find the entry.
- **In general, this is an O(n) operation: in the worst case, the whole tree must be explored.**

Motivating Example

- Suppose we have an online trading system where orders to purchase and sell a given 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_b,s_b) is executed when a sell order (p_s,s_s) with price $p_s \le p_b$ is added (the execution is complete if $s_s \ge s_b$)
  - A sell order (p_s, s_s) is executed when a buy order (p_b, s_b) with price $p_b \ge p_s$ is added (the execution is complete if $s_b \ge s_s$)
- 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?

Additional Methods of the Adaptable Priority Queue ADT

- remove(*e*): Remove from *P* and return entry *e*.
- replaceKey(*e*,*k*): Replace key with *k* and return the old key.
- replaceValue(*e*,*v*): Replace value with *v* and return the old value.

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

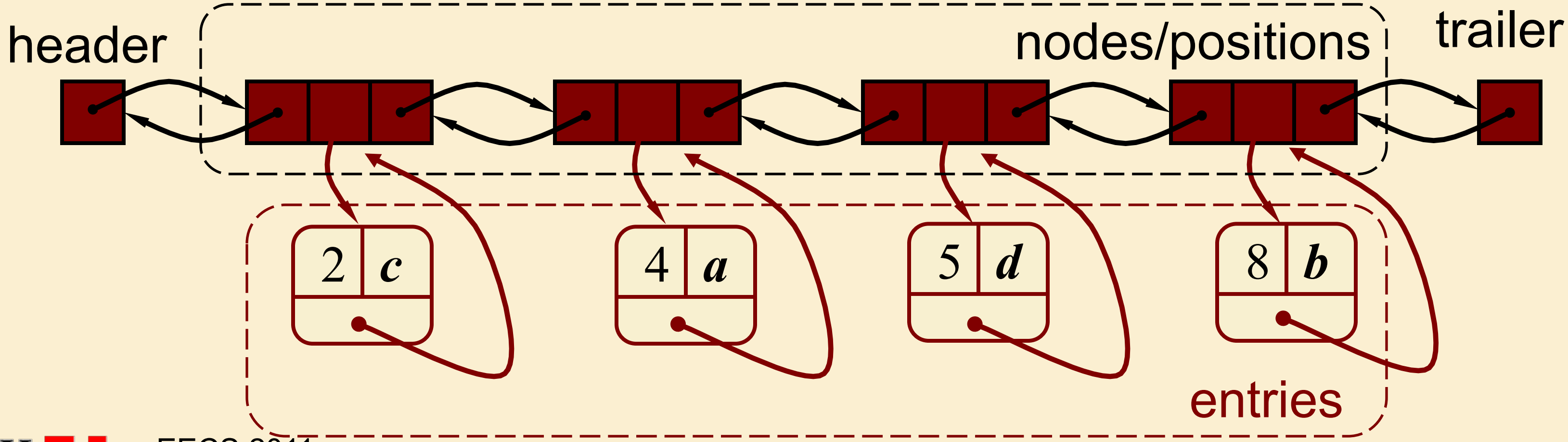
- In order to implement the operations remove(e), replaceKey(e,k), and replaceValue(e,v), we need a fast way of locating an entry e in a priority queue.
- We can always just search the entire data structure to find an entry e, but this takes O(n) time.
- Using location-aware entries, this can be reduced to O(1) time.

Location-Aware Entries

- A location-aware entry identifies and tracks the location of its (key, value) object within a data structure

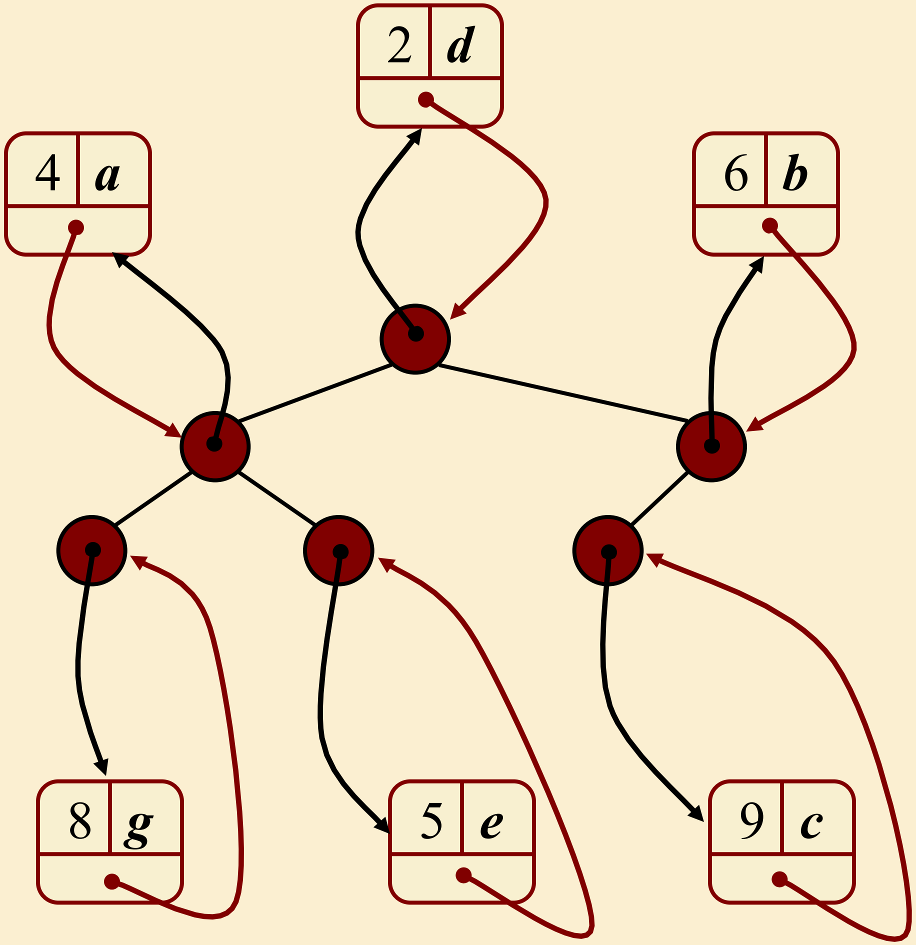
List Implementation

- 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 with Location-Aware Entries

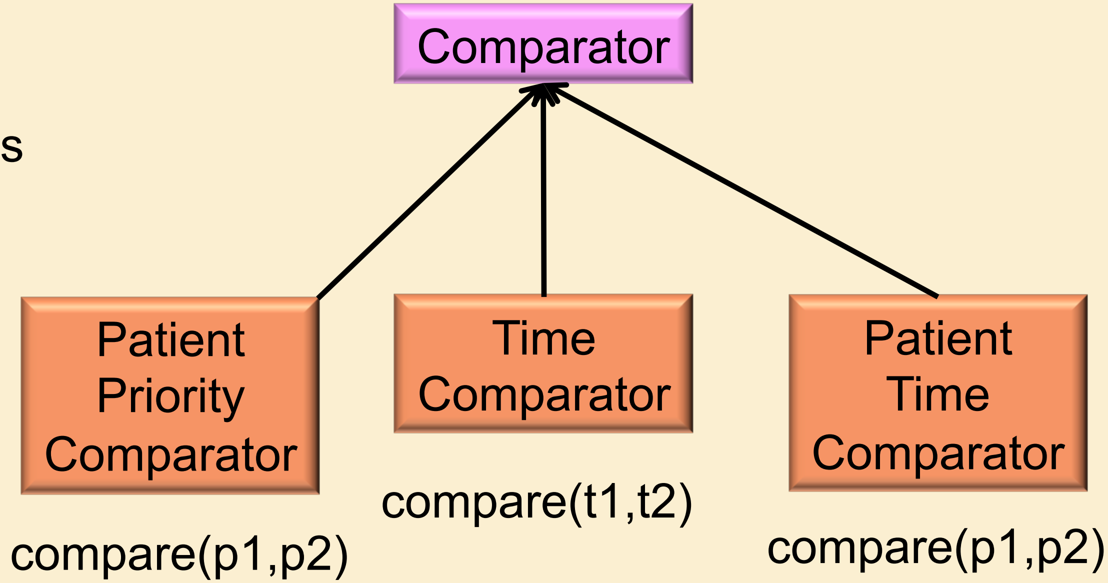
- Times better than those achievable without location-aware entries are highlighted in red:

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	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$

A2Q2: Emergency Ward Triage

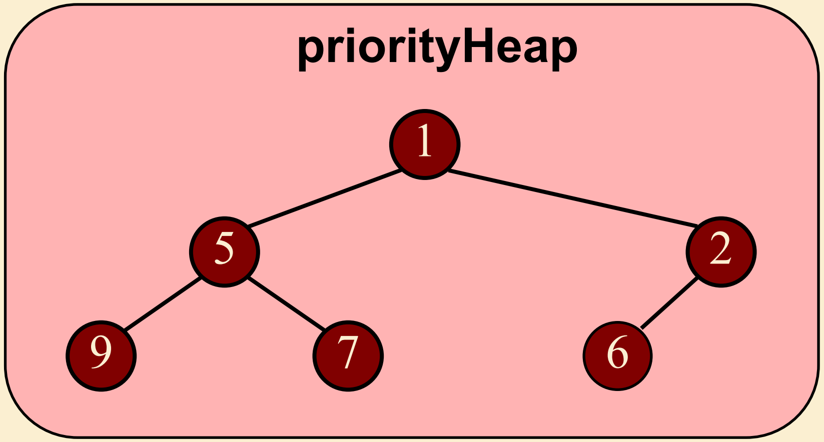
- Patients are normally seen by priority.
- However patients that have waited longer than maxWaitTime are seen next.
- Selecting the next patient thus involves comparing both priorities and wait times.

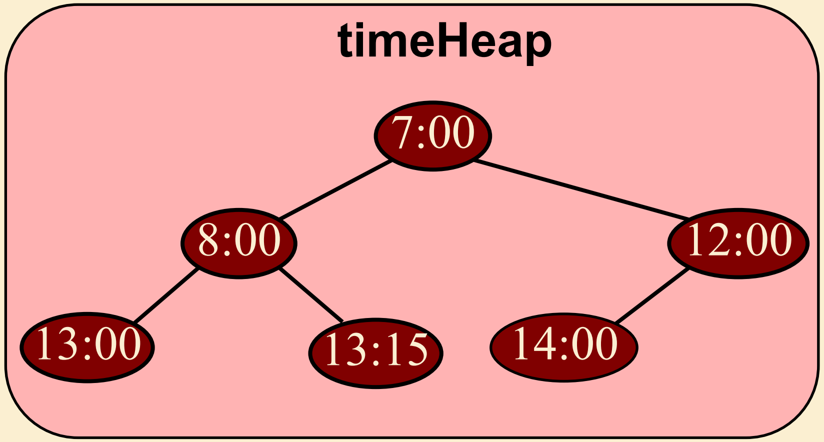
Let
- p1, p2 be patients
- t1, t2 be times



A2Q2: Emergency Ward Triage

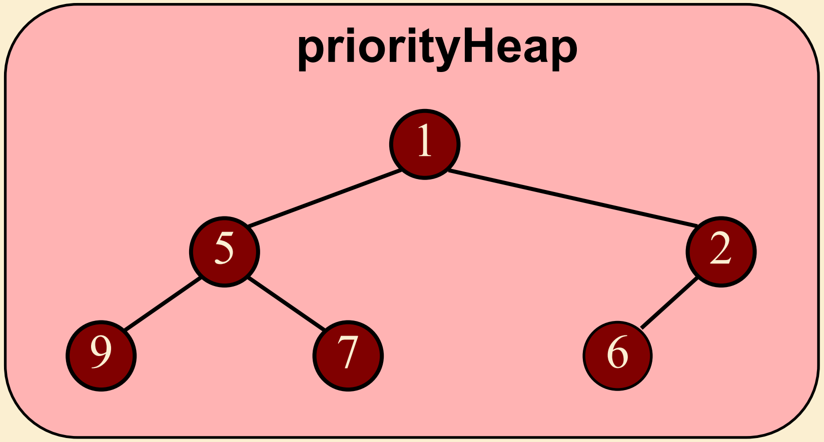
- To support this requirement, we use two priority queues, one for priority and one for arrival time.
- Each is implemented as an array-based min-heap.

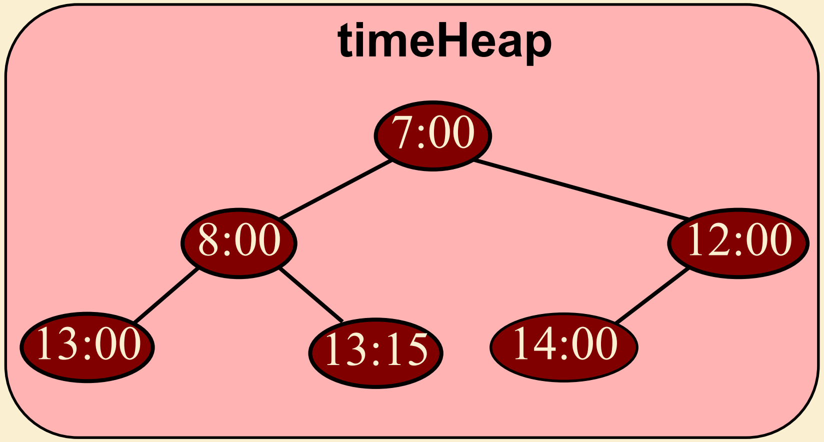




A2Q2: Emergency Ward Triage

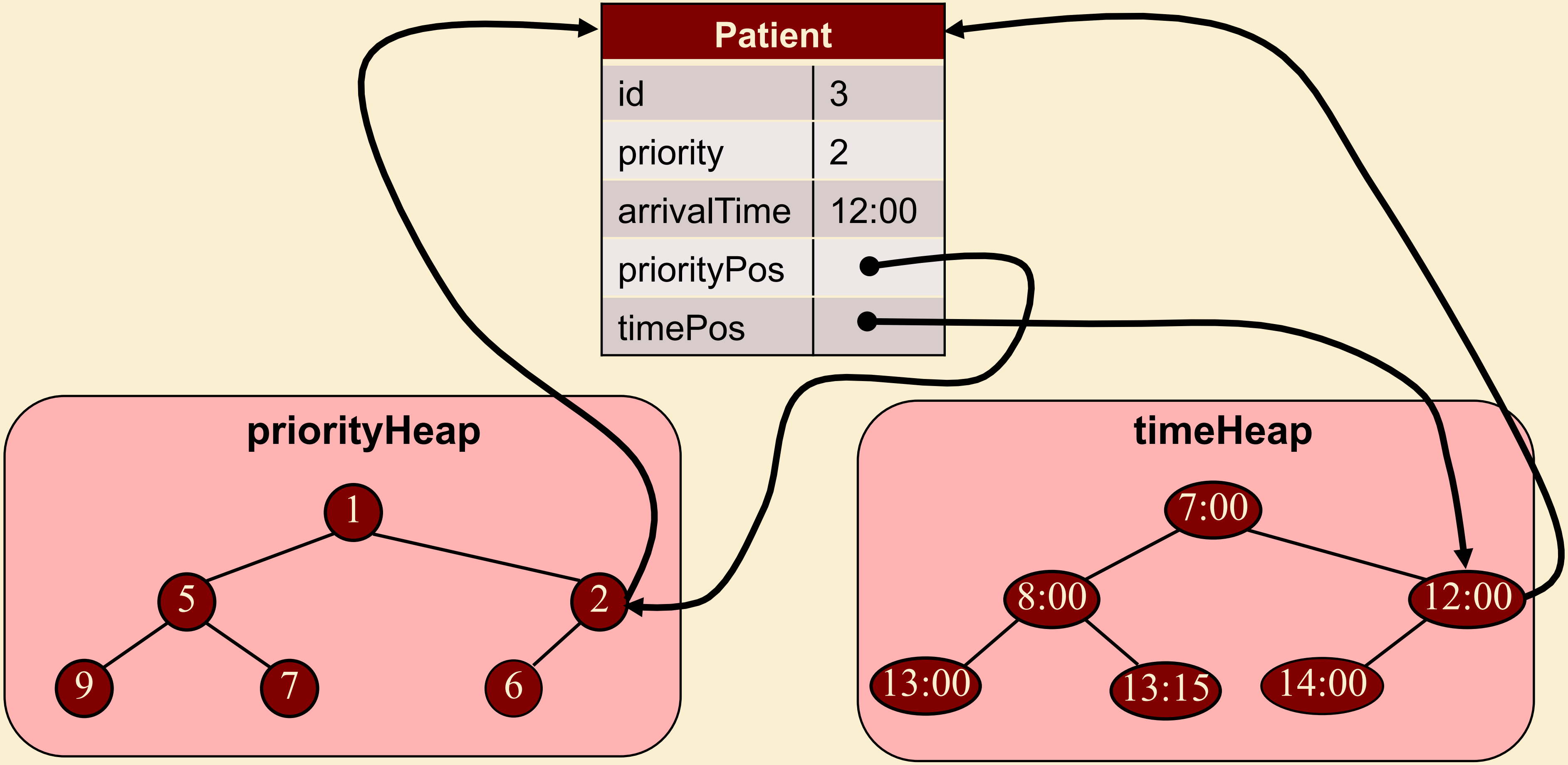
- When a patient is selected, they must be removed from both queues.
- If selected by priority, this requires an O(n) search through the time queue.
- If selected by time, this requires an O(n) search through the priority queue.





A2Q2: Emergency Ward Triage

- This O(n) search can be avoided using location-aware entries that cross-reference the two queues.

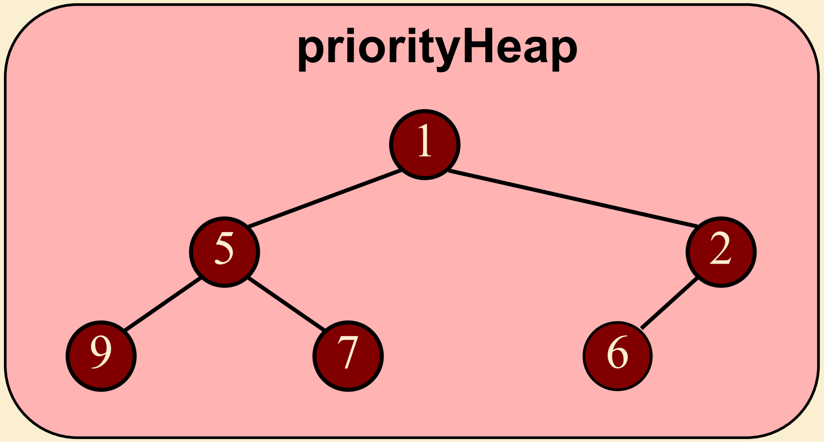


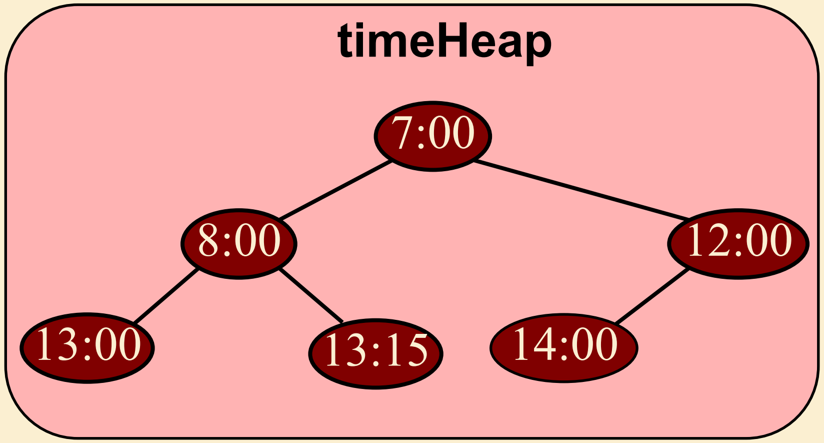
Patient	
id	3
priority	2
arrivalTime	12:00
priorityPos	
timePos	

A2Q2: Emergency Ward Triage

- Whenever the location of an entry within one of the heaps changes, the corresponding location field in the entry must be updated.
  - ❑ offer(e)
  - ❑ remove(p)
  - ❑ poll()
  - ❑ swap(p1, p2)

Patient	
id	3
priority	2
arrivalTime	12:00
priorityPos	
timePos	





APQ Class

- We base our adaptable priority queue (APQ) class on the java.util PriorityQueue design

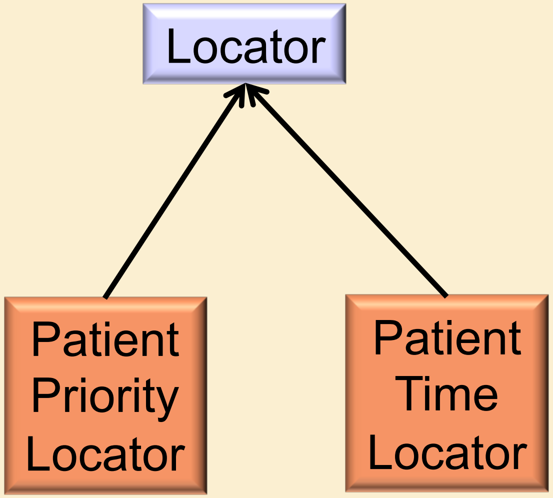
Let

- k be a key
- v be a value
- e be an entry
- p be a position

Adaptable Priority Queue ADT (Textbook)	APQ Class
insert(k,v)	offer(e)
removeMin()	poll()
min()	peek()
remove(e)	remove(p)

A2Q2: Emergency Ward Triage

- To facilitate manipulation of entry positions, we introduce a Locator interface that is specialized into
  - ❑ PatientPriorityLocator
  - ❑ PatientTimeLocator
- These provide two methods, **get(p)** and **set(p),** that allow the position of a patient **p** in the queue to be accessed and updated.



Outline

- The Priority Queue class of the Java Collections Framework
- Total orderings, the Comparable Interface and the Comparator Class
- Heaps
- Adaptable Priority Queues

Outcomes

- By understanding this lecture, you should be able to:
  - ❑ Explain and design a priority queue ADT
  - ❑ Describe suitable applications for priority queues
  - ❑ Design and implement a heap
  - ❑ Analyze the run time of a heap
  - ❑ Identify advantages of and suitable applications for heaps
  - ❑ Design and implement an efficient adaptable priority queue ADT using location-aware entries
  - ❑ Identify suitable applications for an adaptable priority queue