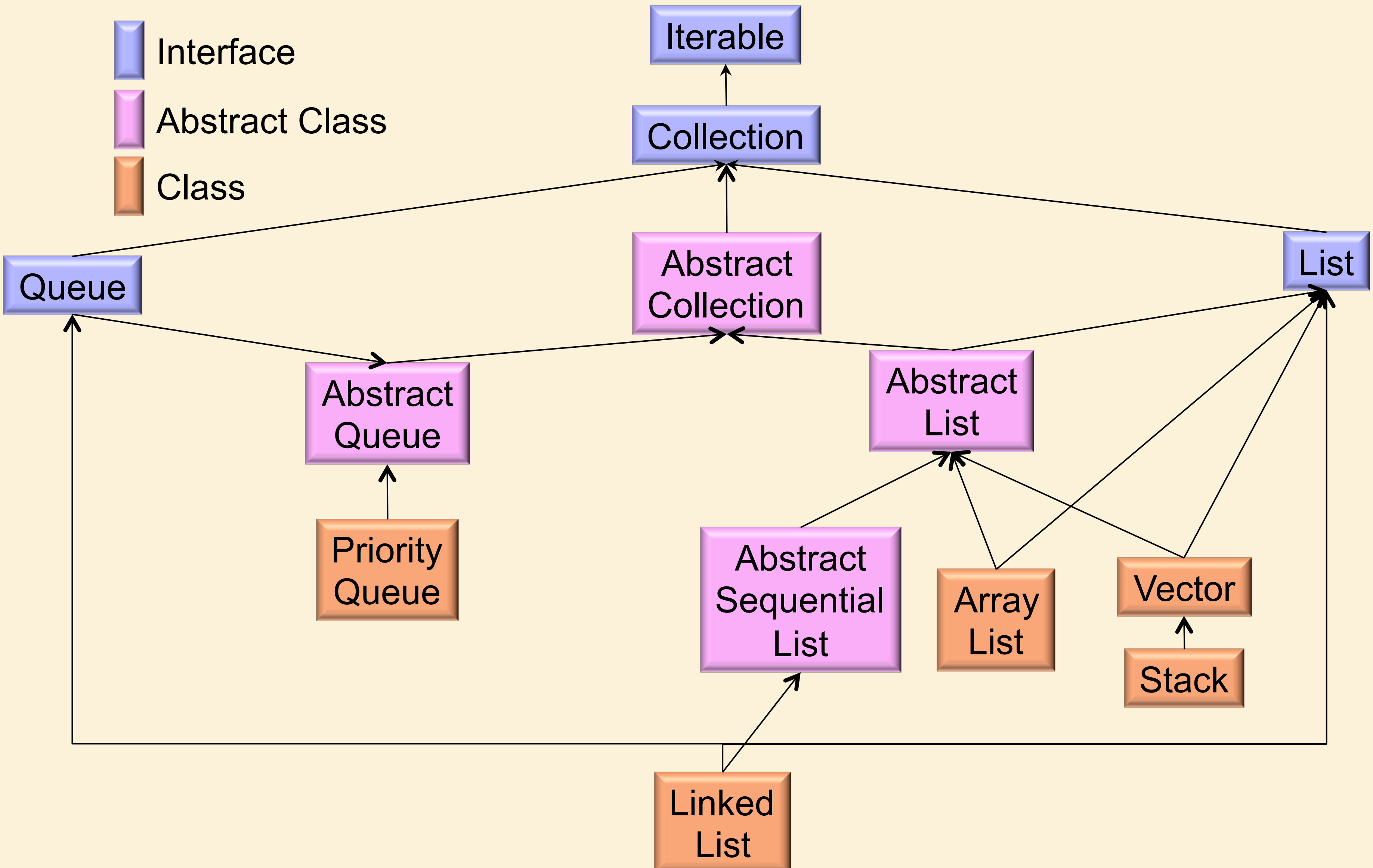
Priority Queues & Heaps

Chapter 8

The Java Collections Framework (Ordered Data Types)



The **Priority Queue** Class

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

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, x) inserts an entry with key k and value x
  - **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

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: $x \le x$
  - Antisymmetric property: $x \le y \wedge y \le x \Rightarrow x = y$
  - Transitive property: $x \le y \wedge y \le z \Rightarrow x \le z$

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  Object getKey();
    public  Object getValue();
}
```

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

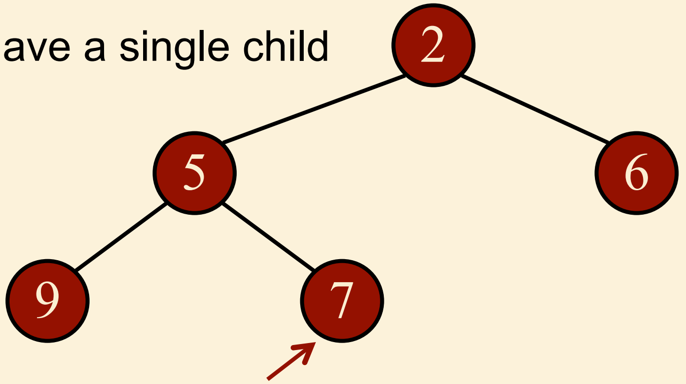
Is this tradeoff inevitable?

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
    - $key(v) \geq key(parent(v))$
  - (Almost) complete binary tree: let h be the height of the heap
    - for $i = 0, \dots, h-1$, there are 2^i nodes of depth i
    - at depth $h-1$
      - 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 h**

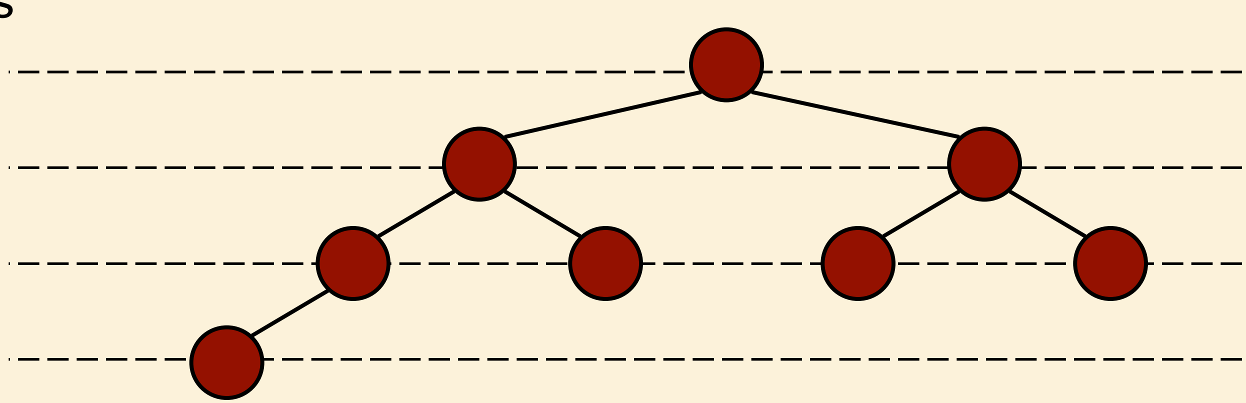
Height of a Heap

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

  Proof: (we apply the complete binary tree property)

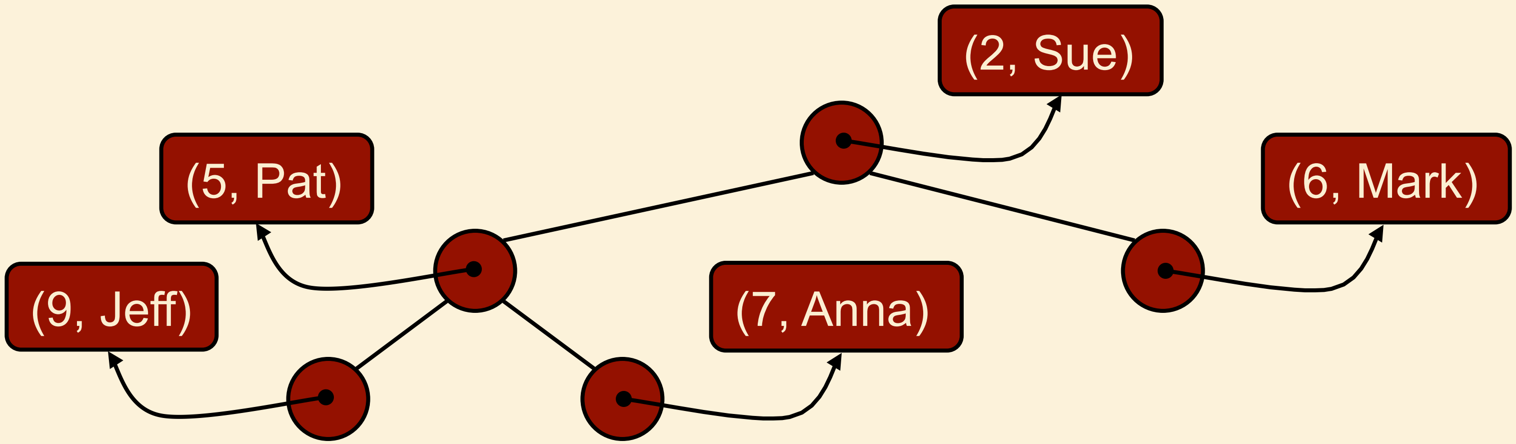
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
  - Thus, $n \geq 2^h$, i.e., $h \leq \log n$

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



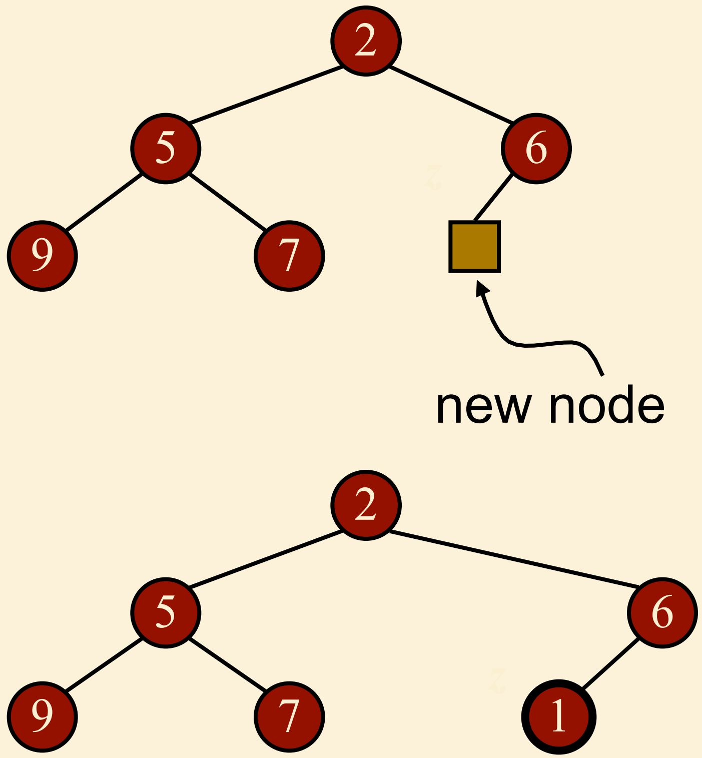
Heaps and Priority Queues

- 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
- For simplicity, we will typically show only the keys in the pictures



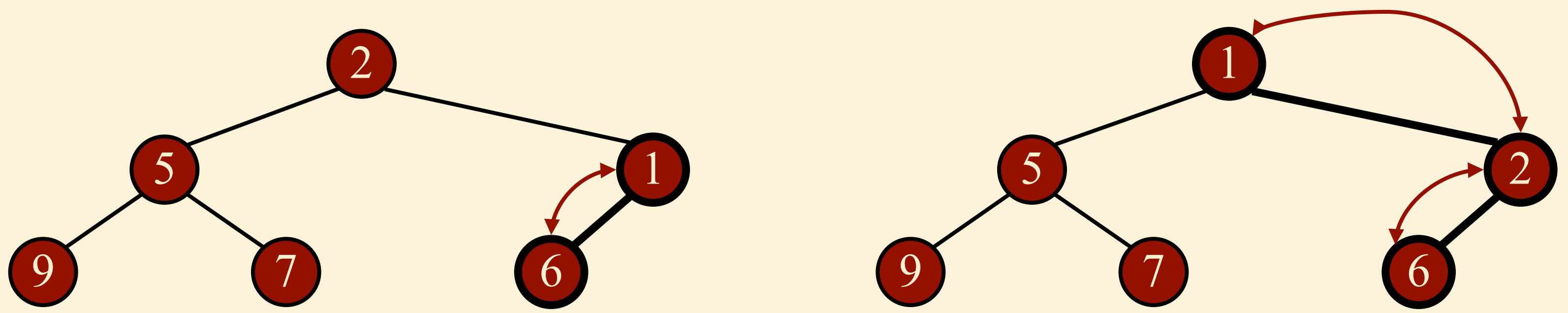
Insertion into a Heap

- 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



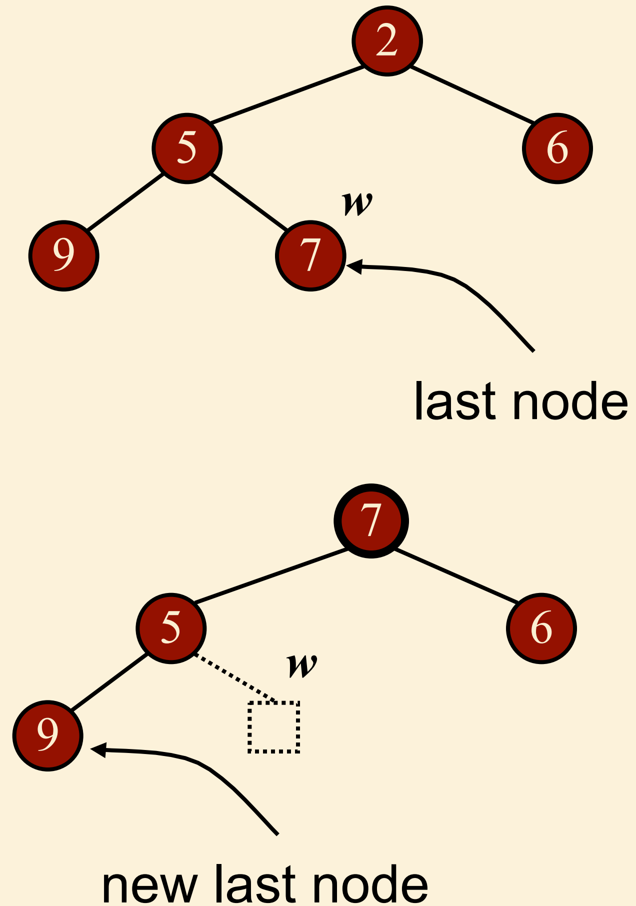
Upheap

- 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



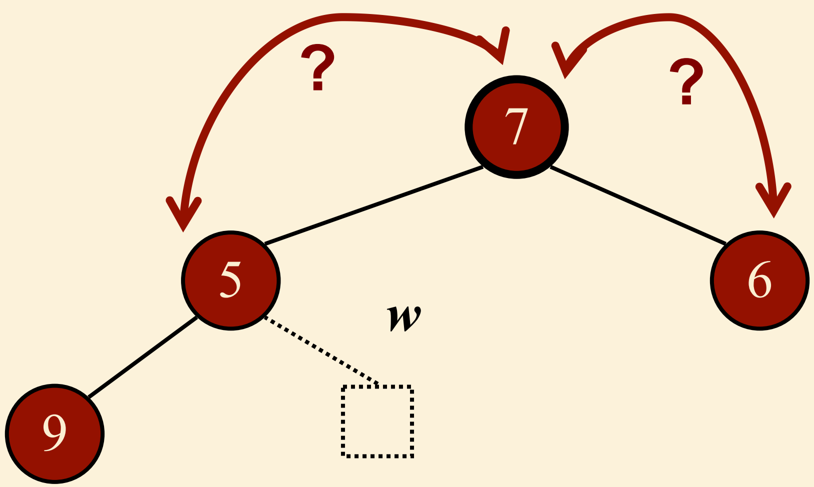
Removal from a Heap

- 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



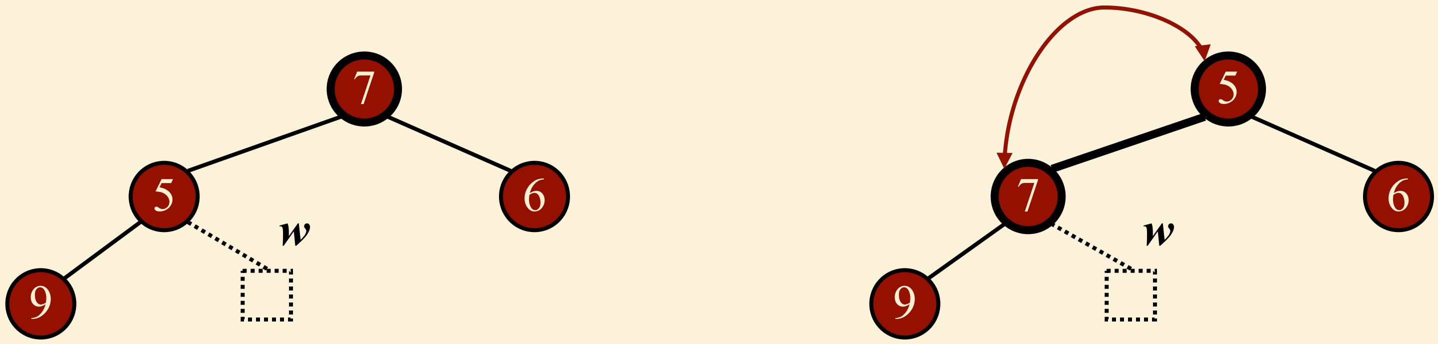
Downheap

- 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?



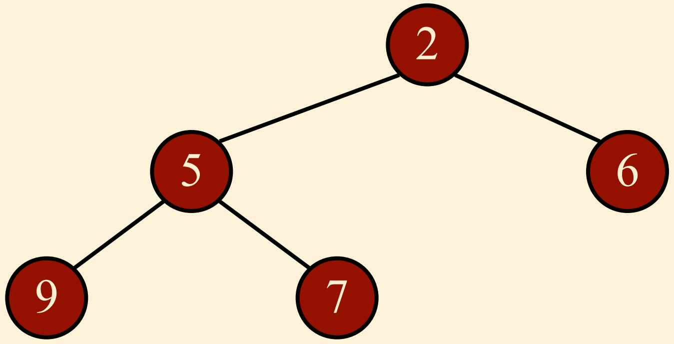
Downheap

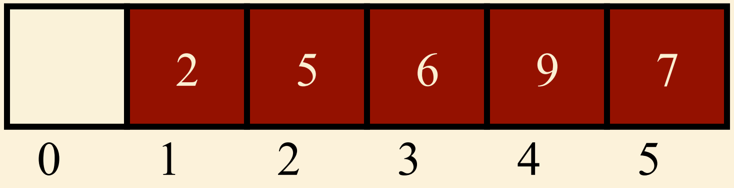
- We select the downward path through the **minimum-key** nodes.
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time



Array-based Heap Implementation

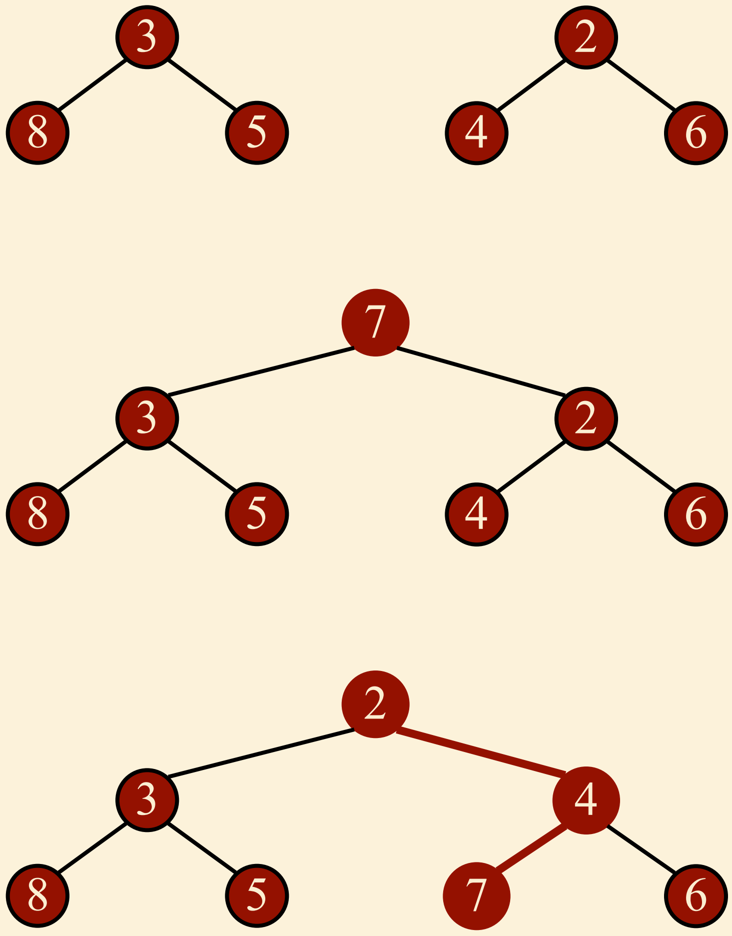
- We can represent a heap with n keys by means of an array of length $n + 1$
- Links between nodes are not explicitly stored
- The cell at rank 0 is not used
- The root is stored at rank 1.
- For the node at rank i
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$
  - the parent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





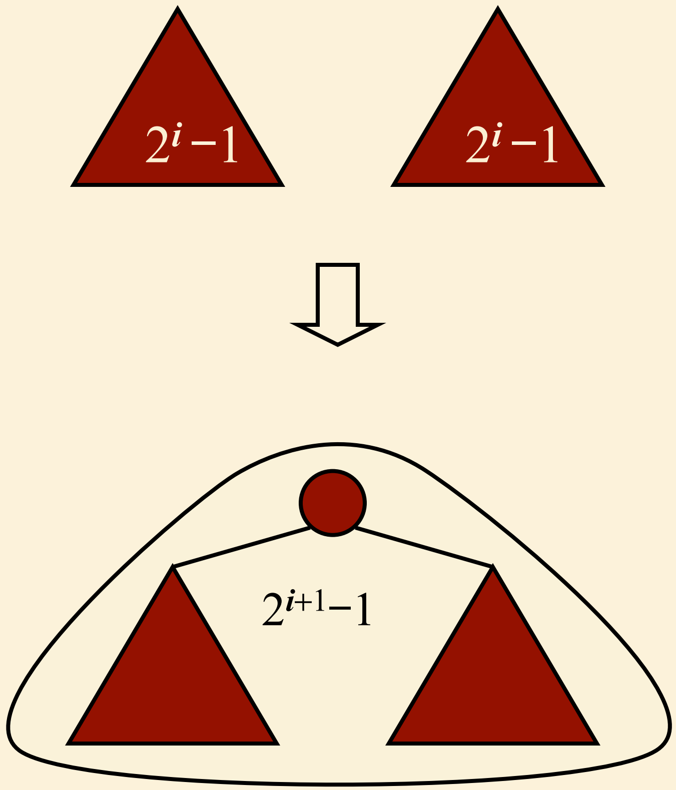
Merging Two Heaps

- 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

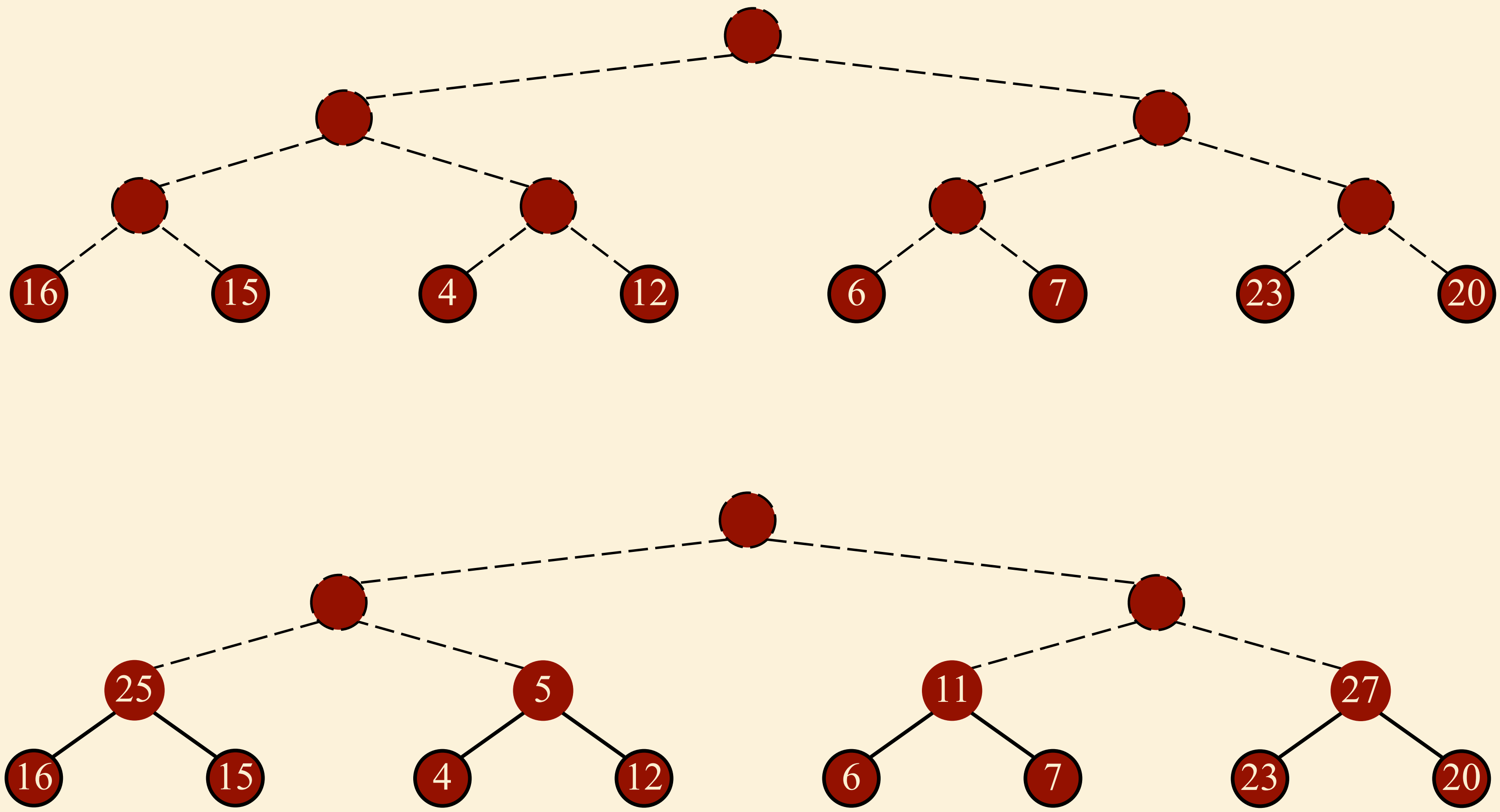


Bottom-up Heap Construction

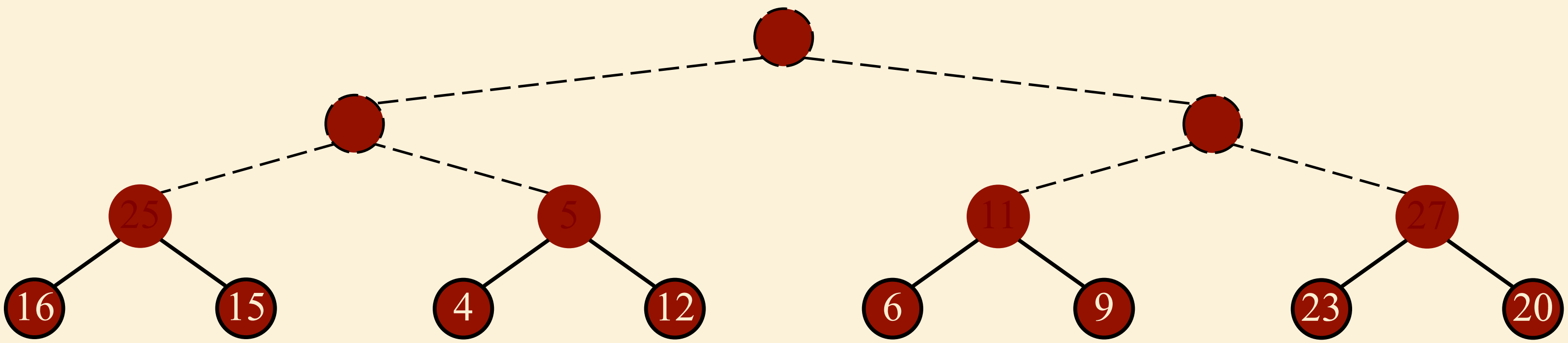
- We can construct a heap storing n keys using a bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys

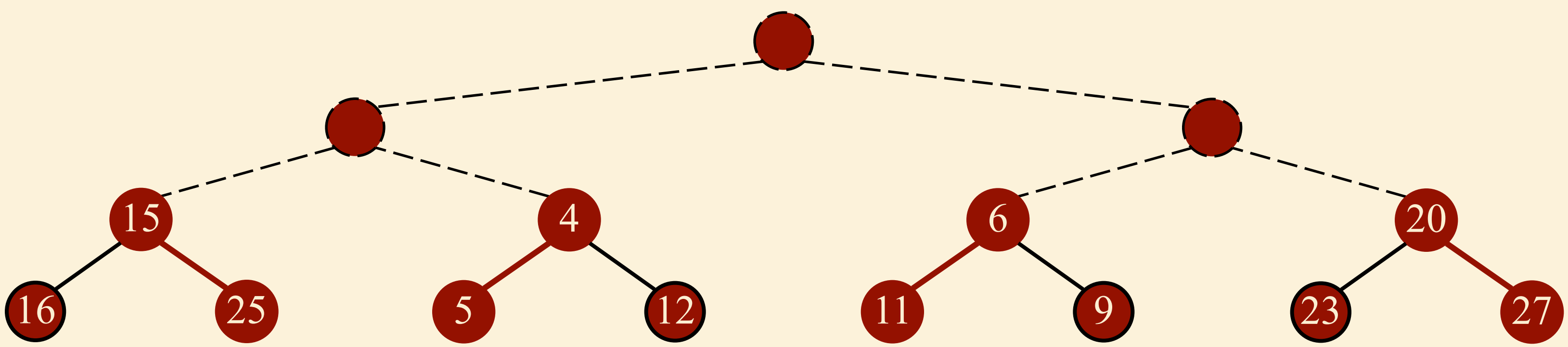


Example

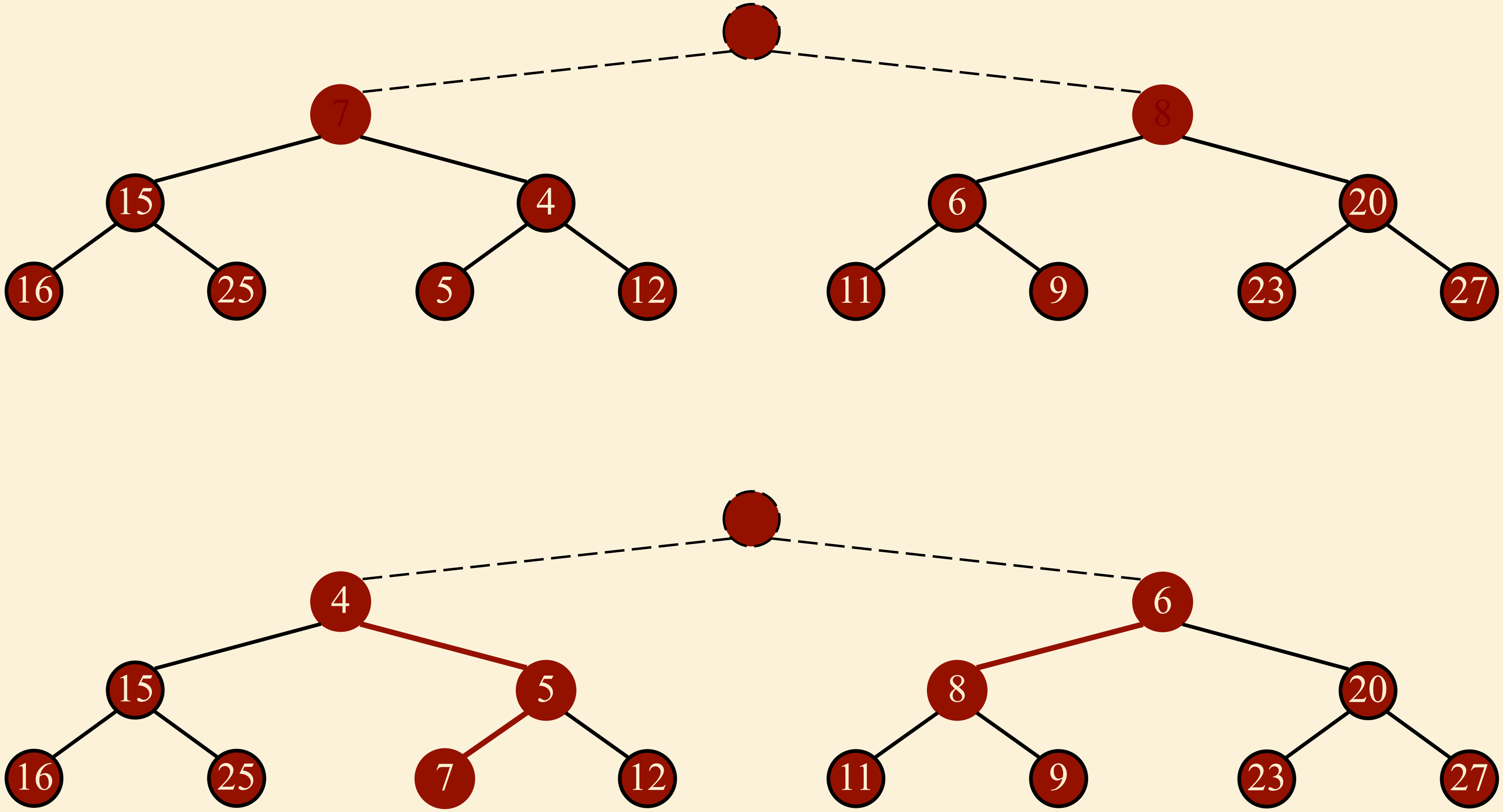


Example (contd.)

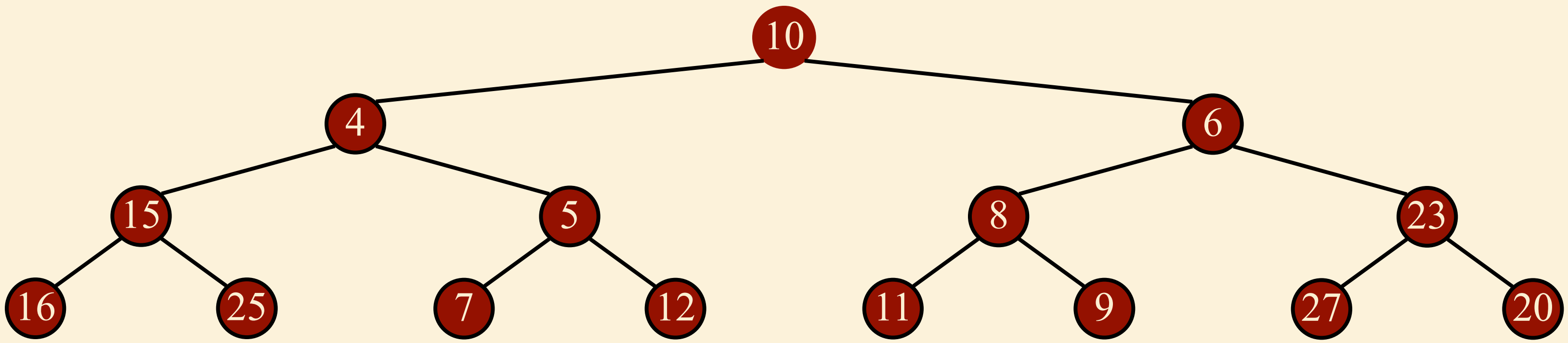


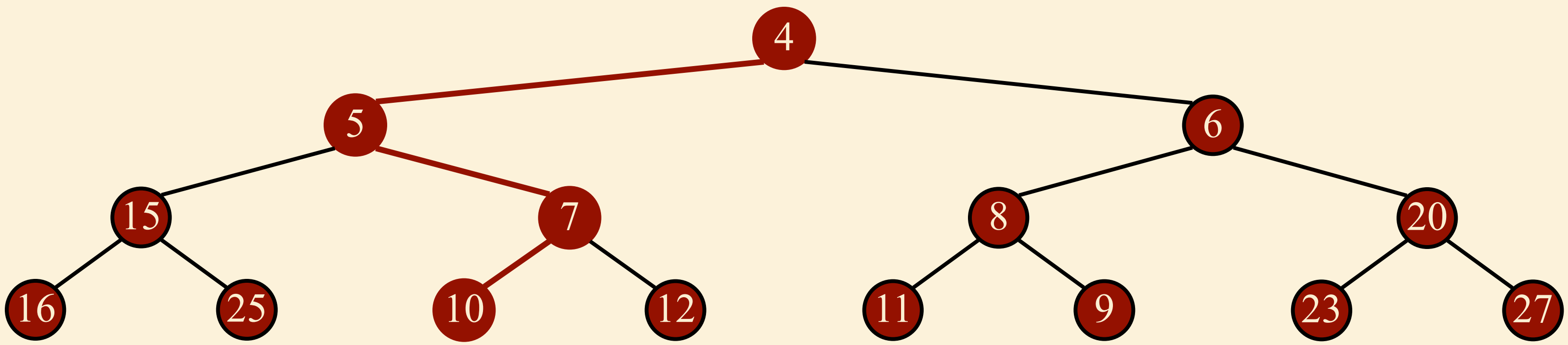


Example (contd.)



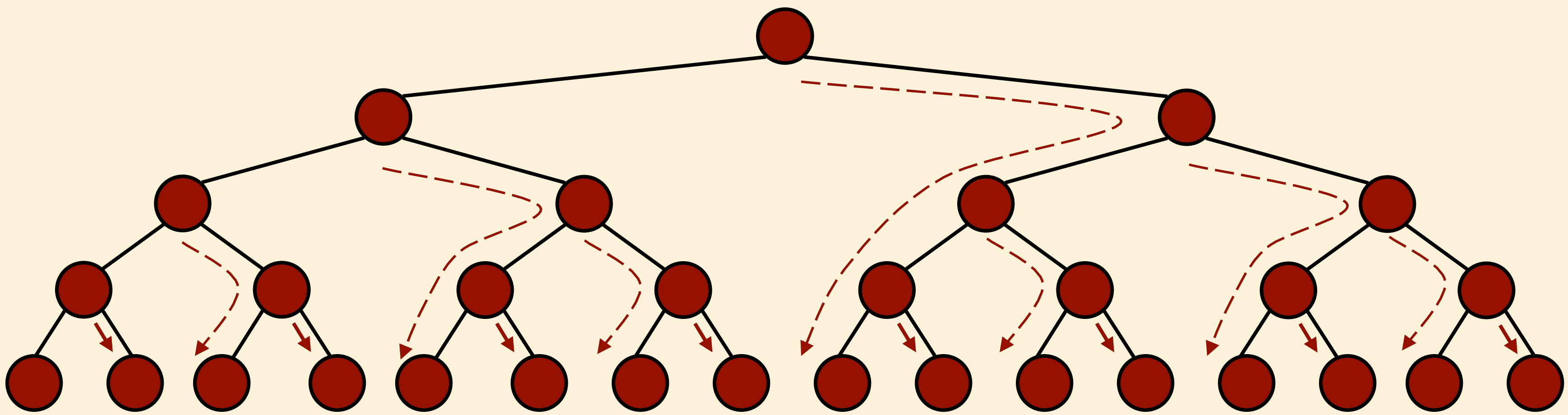
Example (end)





Analysis

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



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.

Iterative MakeHeap

MakeHeap(A, n)

<pre-cond>:A[1…n] is a balanced binary tree

<post-cond>:A[1…n] is a heap

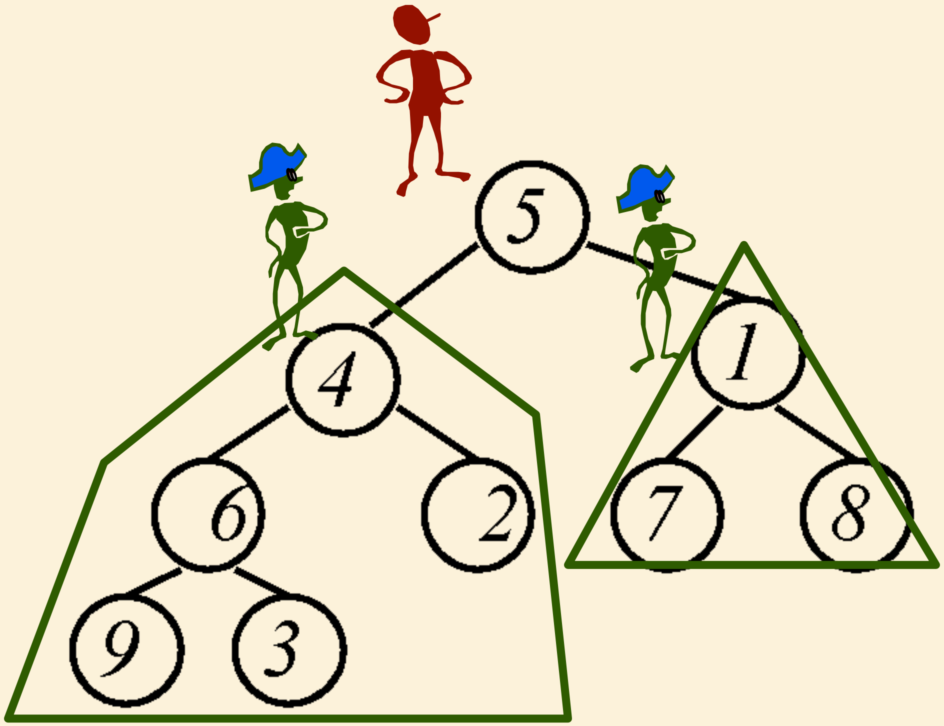
for $i \leftarrow \lfloor n/2 \rfloor$ downto 1

$\quad < LI >$: All subtrees rooted at $i+1 \ldots n$ are heaps

$\quad DownHeap(A, i, n)$

Recursive MakeHeap

Get help from friends



Recursive MakeHeap

MakeHeap(A, i, n)

<pre-cond>:A[i…n] is a balanced binary tree

<post-cond>:The subtree rooted at i is a heap

if $i \le \lfloor n/4 \rfloor$ *then*

    MakeHeap(A, LEFT(i), n)

    MakeHeap(A, RIGHT(i), n)

Downheap(A, i, n)

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

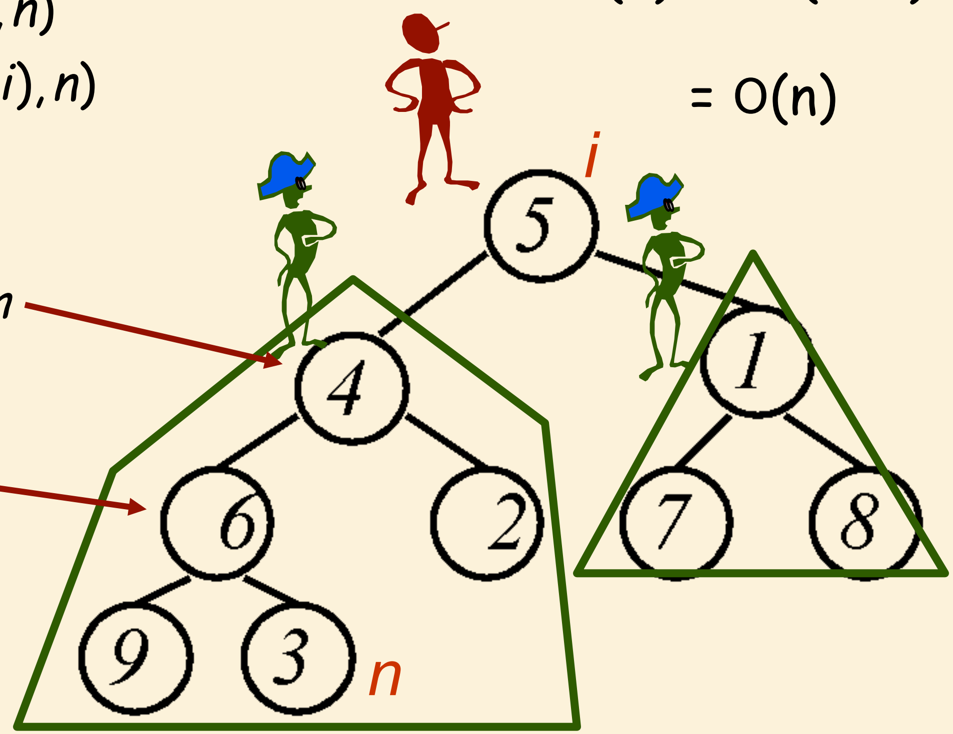
Running time:

$T(n) = 2T(n/2) + \log(n)$

$= O(n)$

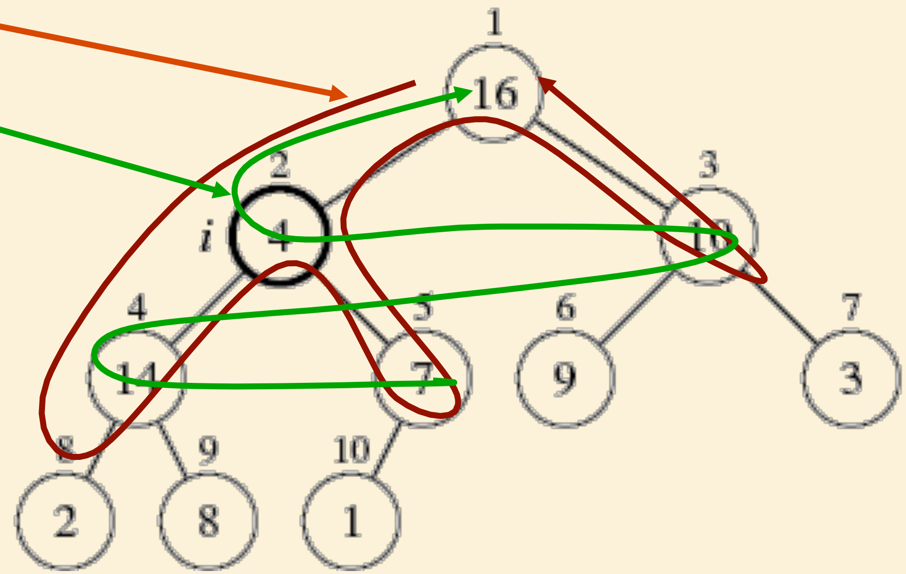
$\lfloor n/4 \rfloor$ is grandparent of n

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

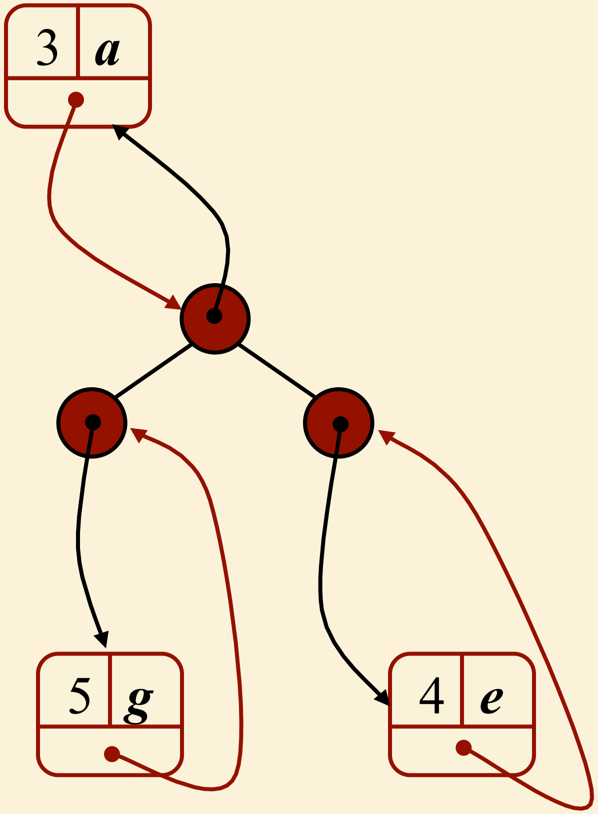


Iterative vs Recursive MakeHeap

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



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

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,s) is executed when a sell order (p’,s’) with price $p' \leq p$ is added (the execution is complete if $s' \geq s$)
  - A sell order (p,s) is executed when a buy order (p’,s’) with price $p' \geq p$ is added (the execution is complete if $s' \geq s$)
- What if someone wishes to cancel their order before it executes?
- What if someone wishes to update the price or number of shares for their order?

Additional Methods of the Adaptable Priority Queue ADT

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

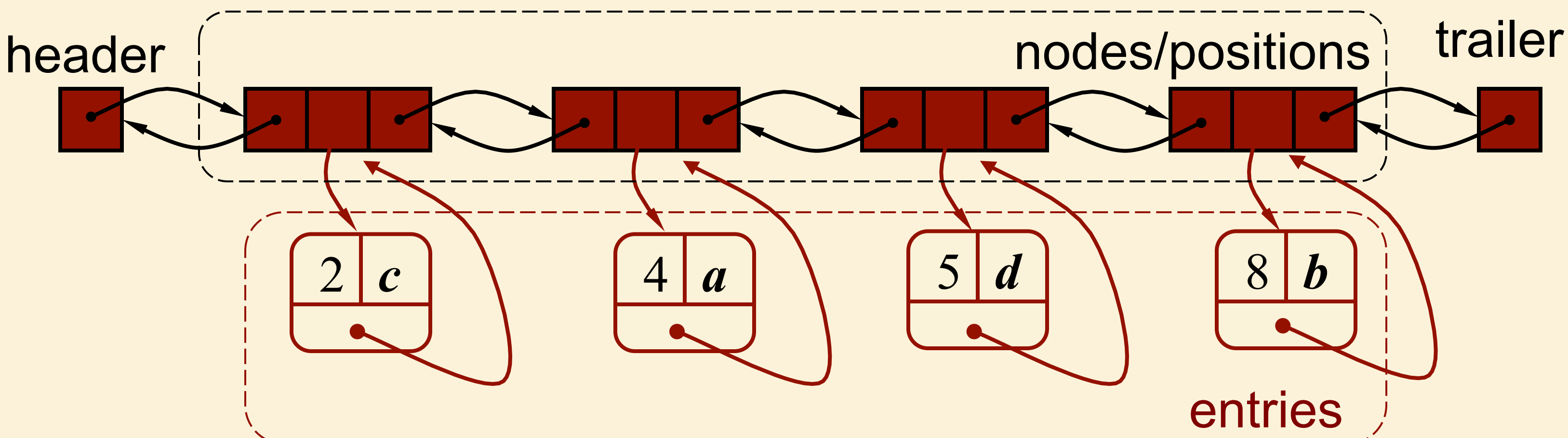
- In order to implement the operations remove(e), replaceKey(e,k), and replaceValue(e,x), we need fast ways of locating an entry e in a priority queue.
- We can always just search the entire data structure to find an entry e, but there are better ways for locating entries.

Location-Aware Entries

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

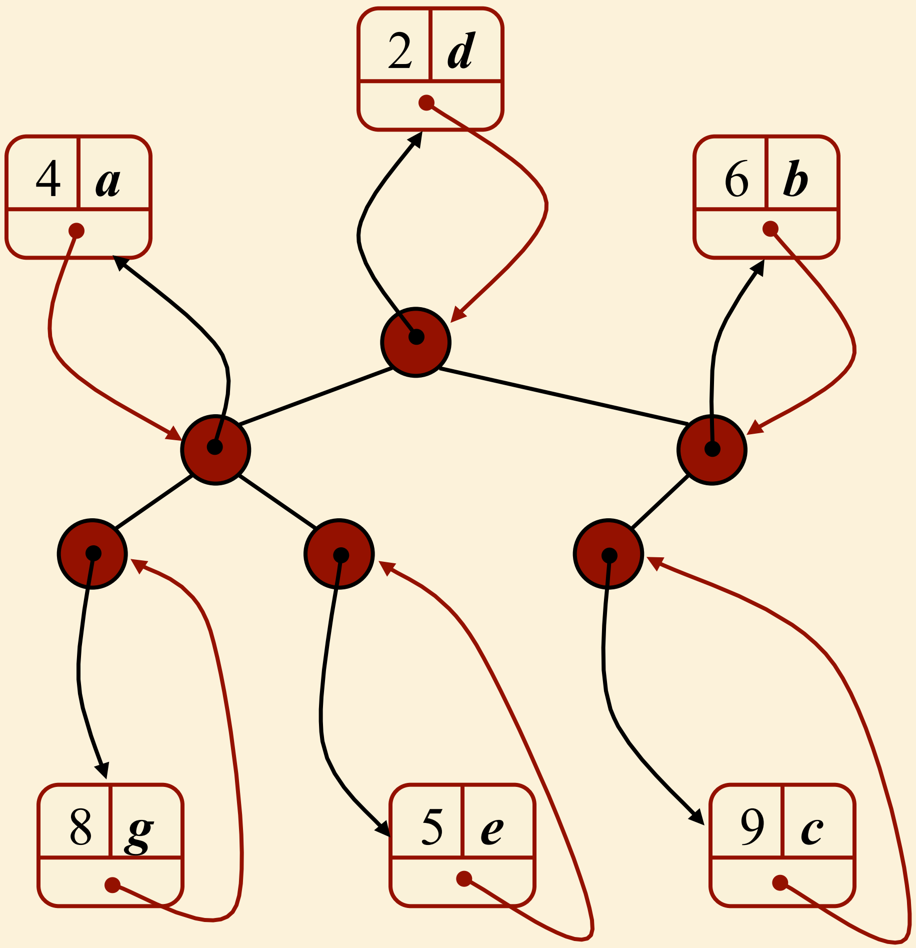
List Implementation

- 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

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