Modified version of the 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 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, 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 lists in airtravel
  - Auctions
  - Stock market
- Note that the max version is also possible

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

- Recall the mathematical concept of a 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 $y \leq z \Rightarrow x \leq z$
  - Reflexive property: $x \leq x$

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();
  }
```

- We have already seen this!

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

- 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.
- The Java interface was in the binary search tree practice

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;
  }
}
```

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

```
1  /** An implementation of a priority queue with an unsorted list. */
2  public class UnsortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
3    /** primary collection of priority queue entries */
4    private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>();
5
6    /** Creates an empty priority queue based on the natural ordering of its keys. */
7    public UnsortedPriorityQueue() { super(); }
8    /** Creates an empty priority queue using the given comparator to order keys. */
9    public UnsortedPriorityQueue(Comparator<K> comp) { super(comp); }
10
11   /** Returns the Position of an entry having minimal key. */
12   private Position<Entry<K,V>> findMin() {    // only called when nonempty
13     Position<Entry<K,V>> small = list.first();
14     for (Position<Entry<K,V>> walk : list.positions())
15       if (compare(walk.getElement(), small.getElement()) < 0)
16         small = walk;          // found an even smaller key
17     return small;
18   }
19
```

Should also force the keys (K) to be "comparable"!

Unsorted List Implementation, 2

```
20    /** Inserts a key-value pair and returns the entry created. */
21    public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
22      checkKey(key);      // auxiliary key-checking method (could throw exception)
23      Entry<K,V> newest = new PQEntry<>(key, value);
24      list.addLast(newest);
25      return newest;
26    }
27
28    /** Returns (but does not remove) an entry with minimal key. */
29    public Entry<K,V> min( ) {
30      if (list.isEmpty( )) return null;
31      return findMin( ).getElement( );
32    }
33
34    /** Removes and returns an entry with minimal key. */
35    public Entry<K,V> removeMin( ) {
36      if (list.isEmpty( )) return null;
37      return list.remove(findMin( ));
38    }
39
40    /** Returns the number of items in the priority queue. */
41    public int size( ) { return list.size( ); }
42  }
```

Sorted List Implementation

```
1  /** An implementation of a priority queue with a sorted list. */
2  public class SortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
3    /** primary collection of priority queue entries */
4    private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>();
5
6    /** Creates an empty priority queue based on the natural ordering of its keys. */
7    public SortedPriorityQueue() { super(); }
8    /** Creates an empty priority queue using the given comparator to order keys. */
9    public SortedPriorityQueue(Comparator<K> comp) { super(comp); }
10
11   /** Inserts a key-value pair and returns the entry created. */
12   public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
13     checkKey(key);      // auxiliary key-checking method (could throw exception)
14     Entry<K,V> newest = new PQEntry<>(key, value);
15     Position<Entry<K,V>> walk = list.last();
16     // walk backward, looking for smaller key
17     while (walk != null && compare(newest, walk.getElement()) < 0)
18       walk = list.before(walk);
19     if (walk == null)
20       list.addFirst(newest);                    // new key is smallest
21     else
22       list.addAfter(walk, newest);              // newest goes after walk
23     return newest;
24   }
25
```

Sorted List Implementation, 2

```
26    /** Returns (but does not remove) an entry with minimal key. */
27    public Entry<K,V> min( ) {
28      if (list.isEmpty( )) return null;
29      return list.first( ).getElement( );
30    }
31
32    /** Removes and returns an entry with minimal key. */
33    public Entry<K,V> removeMin( ) {
34      if (list.isEmpty( )) return null;
35      return list.remove(list.first( ));
36    }
37
38    /** Returns the number of items in the priority queue. */
39    public int size( ) { return list.size( ); }
40  }
```

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
- What are the running times for the sorted and unsorted list implementations?

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.remove*(*S.first* ())
P.insert (*e*, ∅)
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 sequence
- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with n insert operations 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

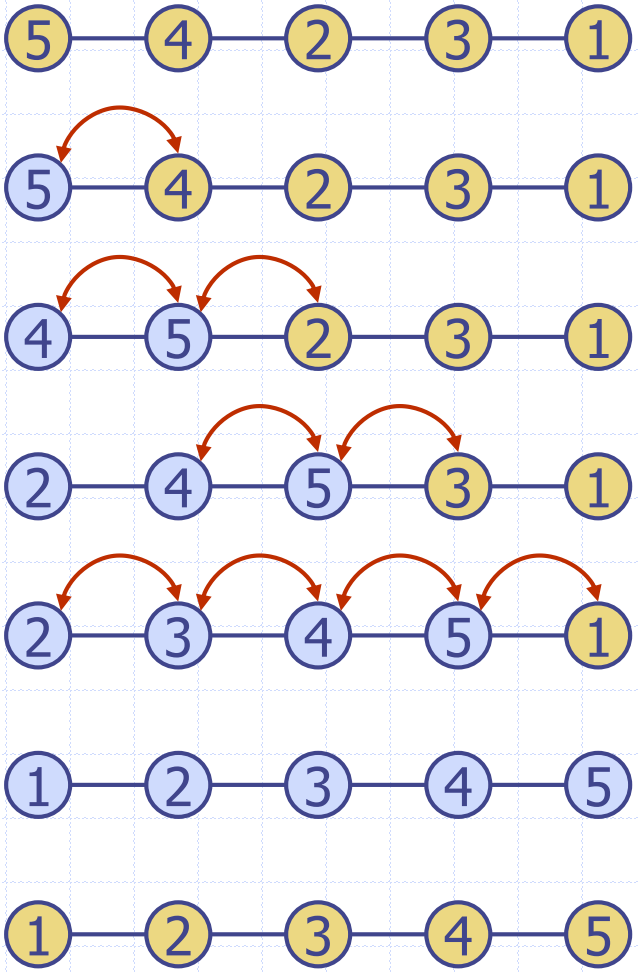
- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with n insert operations takes time proportional to
  $$1 + 2 + \ldots + n$$
  2. Removing the elements in 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

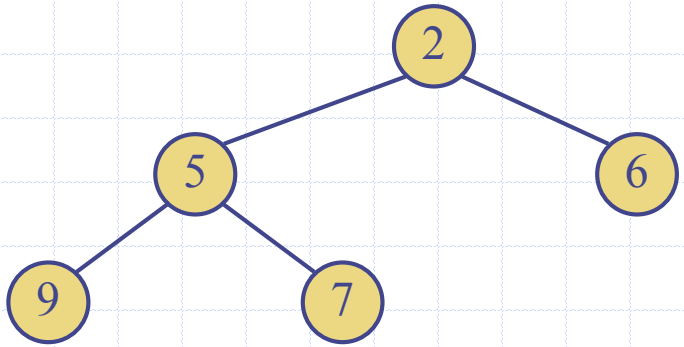
	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

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
- Complexity?



Heaps



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*()

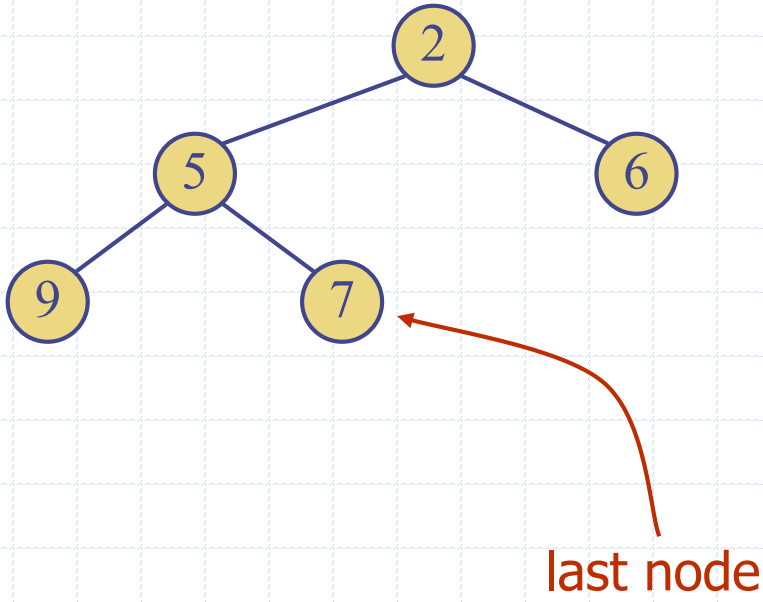
e ← *P.removeMin*().*getKey*()

S.addLast(*e*)

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
  - at depth $h - 1$, the internal nodes are to the left of the external nodes

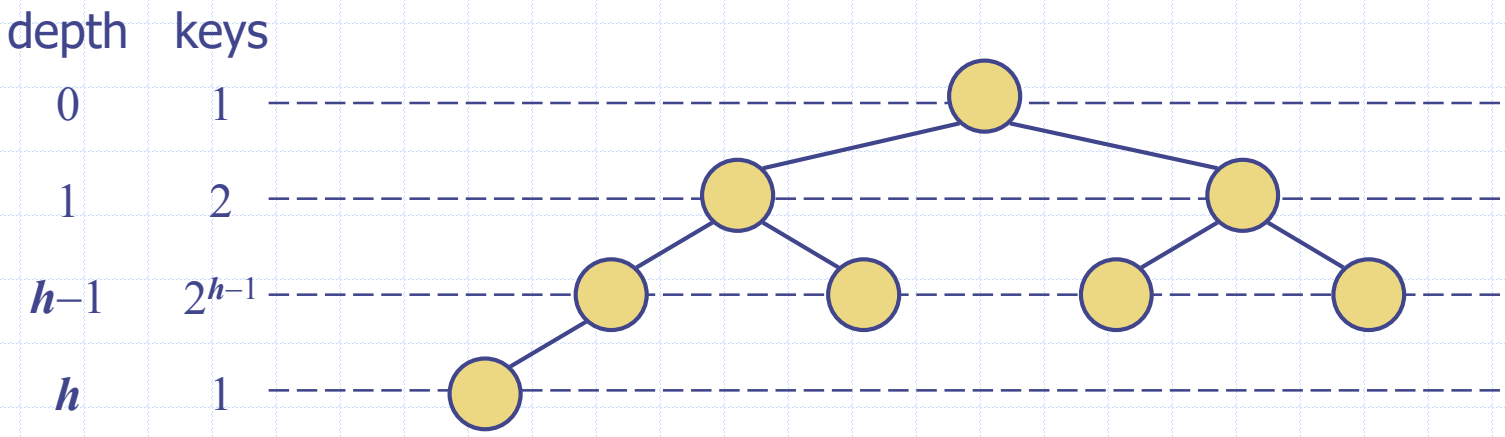
- The last node of a heap is the rightmost node of maximum depth



Height of a Heap

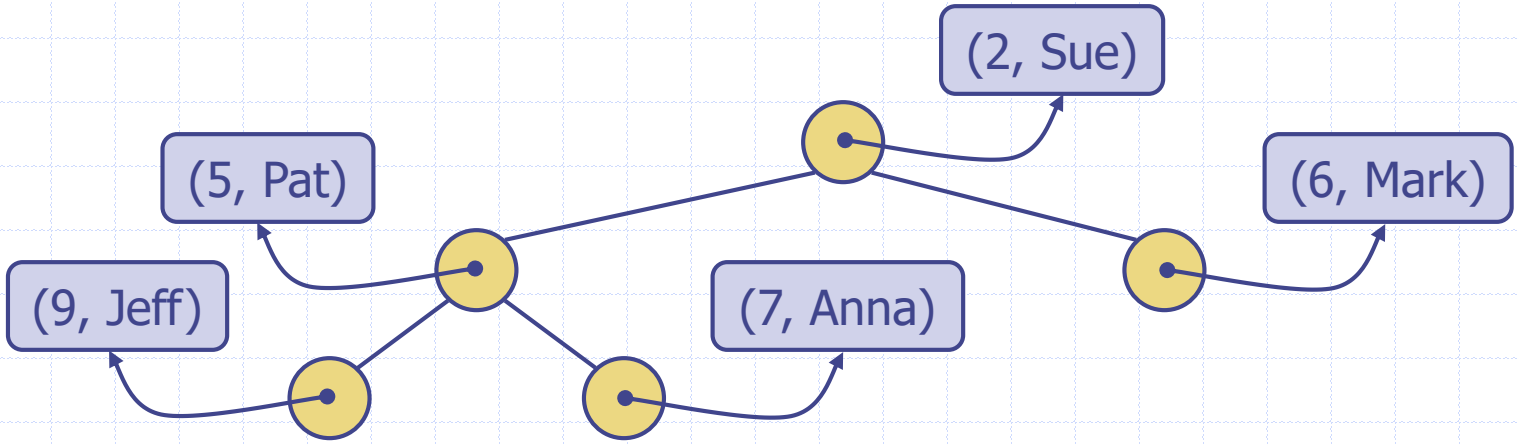
- Theorem: A heap storing n keys has height $O(\log n)$

  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 $i = 0, \ldots, h-1$ 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., $h \leq \log n$



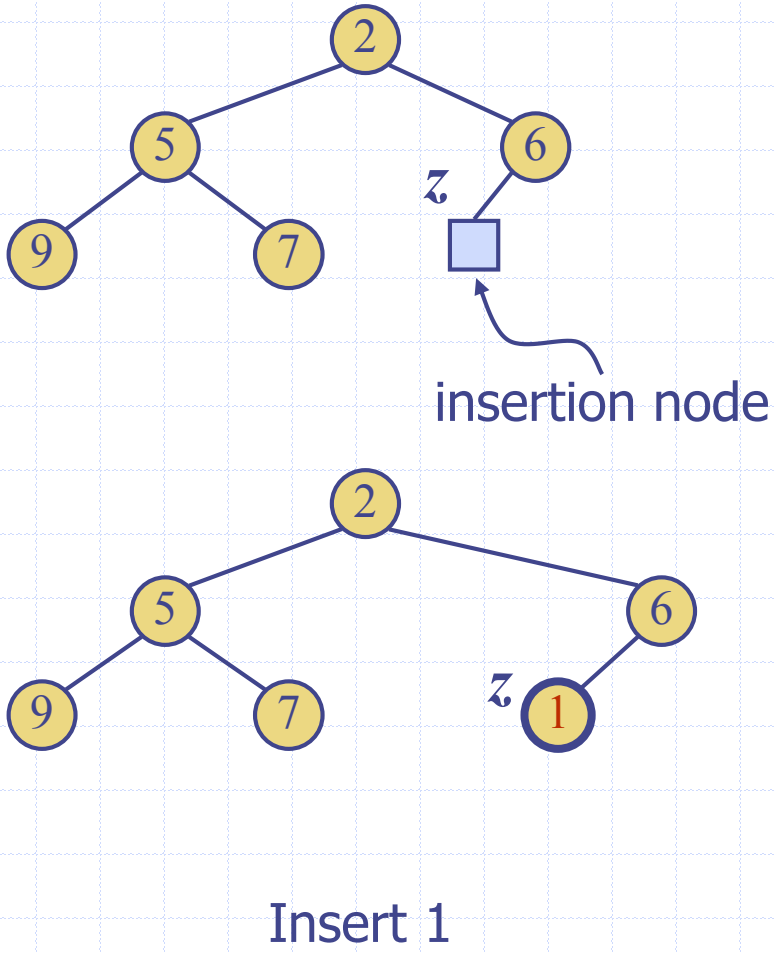
Heaps and Priority Queues

- 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



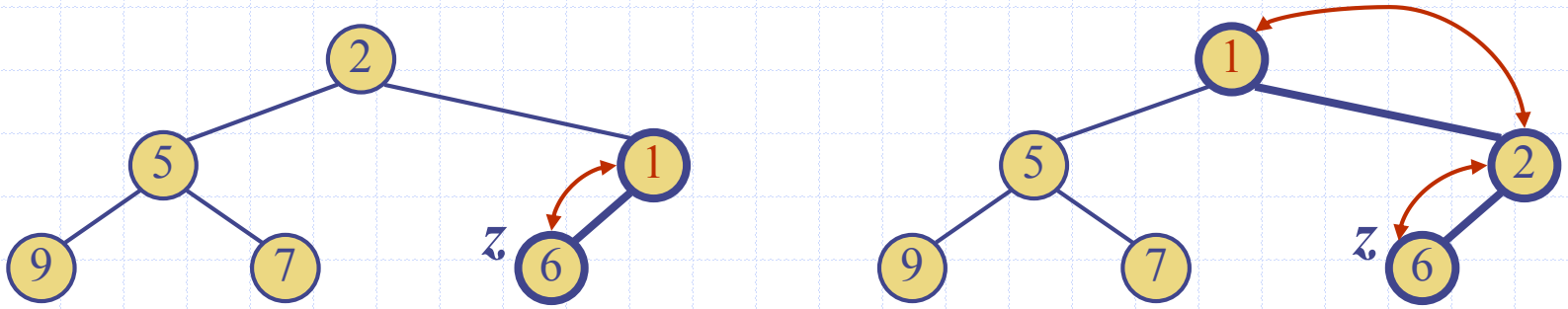
Insertion into a Heap

- 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)



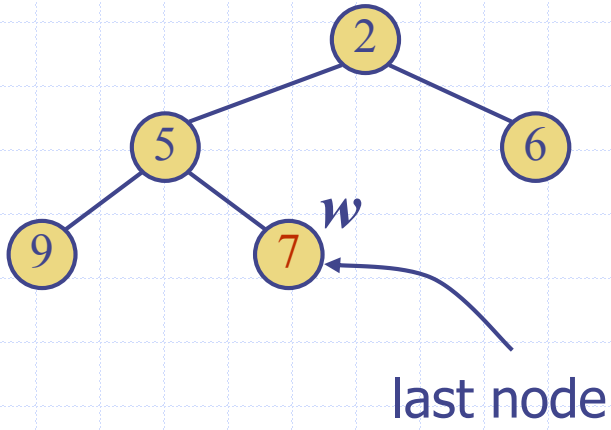
Upheap

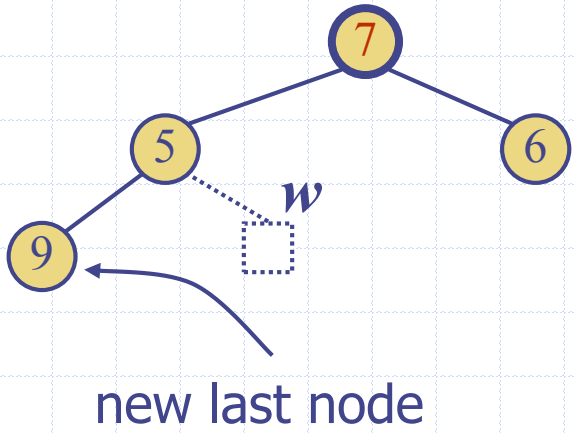
- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k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



Removal from a Heap

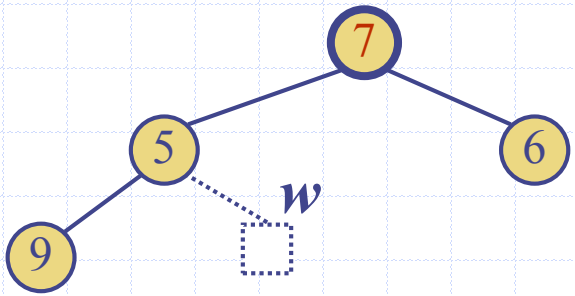
- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Remove w
  - Restore the heap-order property (discussed next)

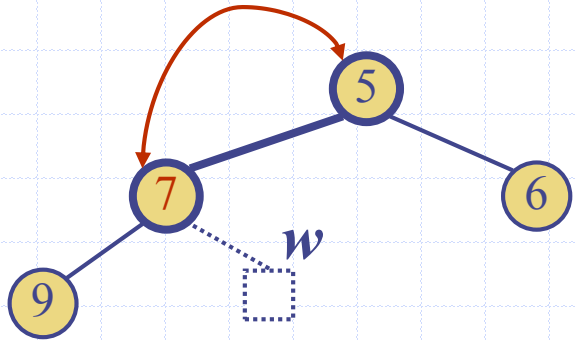




Downheap

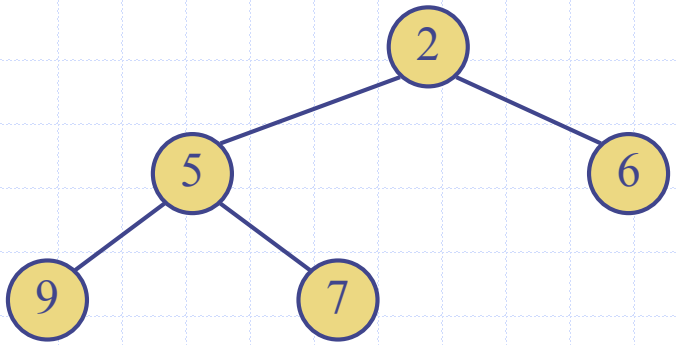
- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time

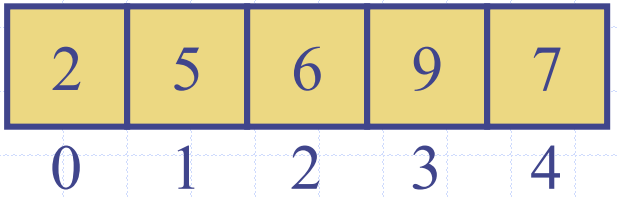




Array-based Heap Implementation

- We can represent a heap with n keys by means of an array of length n
- For the node at rank i
  - the left child is at rank $2i + 1$
  - the right child is at rank $2i + 2$
- Links between nodes are not explicitly stored
- Operation add corresponds to inserting at rank $n + 1$
- Operation remove_min corresponds to removing at rank n
- Yields in-place heap-sort





Updating The Last Node

- For an array based implementation, this is trivial. You always add to/remove from the end of the array and then call upheap/downheap!
- This is because of the complete binary tree property (Why?)
- Thus, the preferred way of implementing heaps are with arrays!

Linked Binary Tree Based Heap Implementation

- Use an ordinary binary tree
- Keep a pointer to the last node
- Change the parent/child pointers as necessary for upheap and downheap operations (insert/delete) and update the pointer to the last node
- Question 1: How to update the pointer?
  - Note that the new last node is either the **level-order predecessor or successor** of the current last node!
- Question 2: Which one is easier between array based and a link based binary tree to implement a heap?

Updating the Last Node, V2

- The previous class we had a slide about how to do for a linked tree based heap implementation.
- The description was incomplete.
- Instead, I am going to tell you about two other ways
  - Level order traversal
  - Binary representation trick
- Note that the preferred implementation of heaps is to use arrays ☺ so these are presented just for completeness

Level Order Traversal

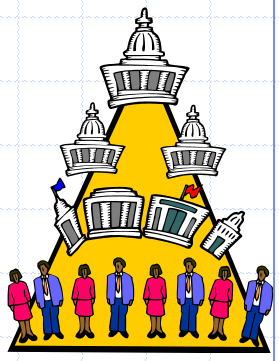
- This is breadth first search
- The last node in the output sequence is the last node of the heap
- Updating the last node for deletion is then trivial
- For insertion, we need to keep track of the node heights
- Unfortunately this is O(n)

```
Algorithm LevelOrder(root, S)
    Input the root of the tree root,
    the output container S
    Output  S with the tree nodes in
    level order
    Q ← empty queue
    Q.enqueue(root)
    while ¬Q.isEmpty ()
        node ← Q .dequeue()
        S.insert (node)
    if(node.left != null)
        Q.enqueue(node.left)
    if(node.right != null)
        Q.enqueue(node.right)
```

Binary Representation Trick

- Assume that we are currently keeping track of the tree size, n
- The binary representation of n-1 can guide us to get to the last node!
- Steps:
  - Convert n-1 to binary
  - Drop the first number
  - Then follow the rest: 0 left child and 1 right child
- Examples on the board!
- O(log n), need to keep the tree size
- Recall that these are not necessary for the array based representation!

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 $O(n \log n)$ 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

- Loop 1:
  - ~ log(1)+log(2)+ … + log(n)
- Loop2:
  - ~ log(1)+log(2)+ … + log(n)

log(1)+log(2)+ … + log(n) =
log(1·2·…·n) = log(n!)

- Sterling's Approximation:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

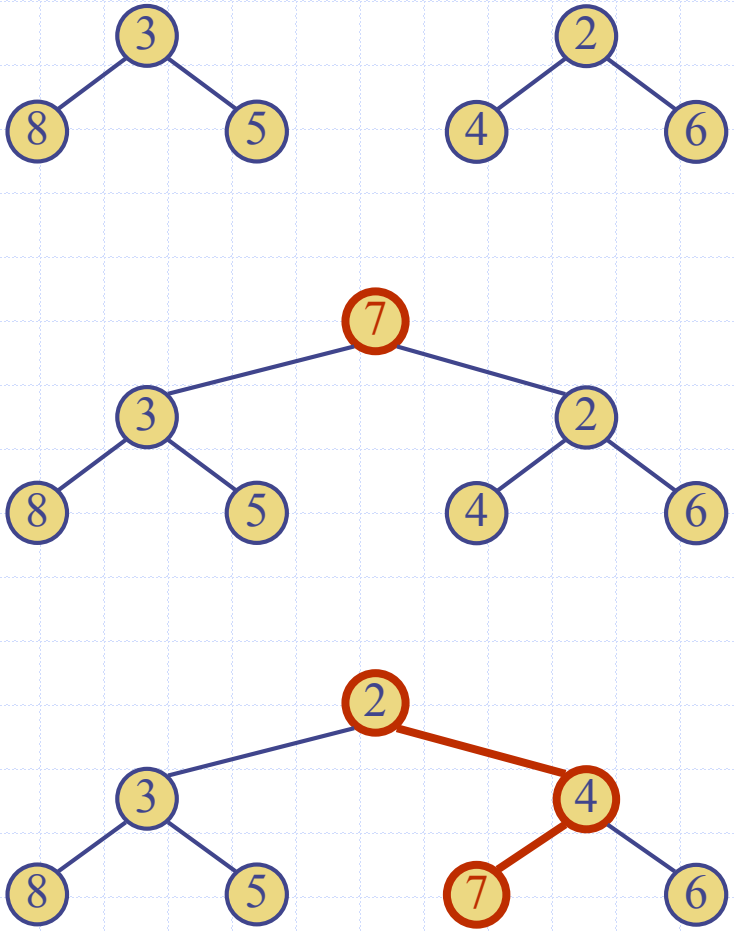
- Thus:

log(n!) is $O(n \log n)$

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*)

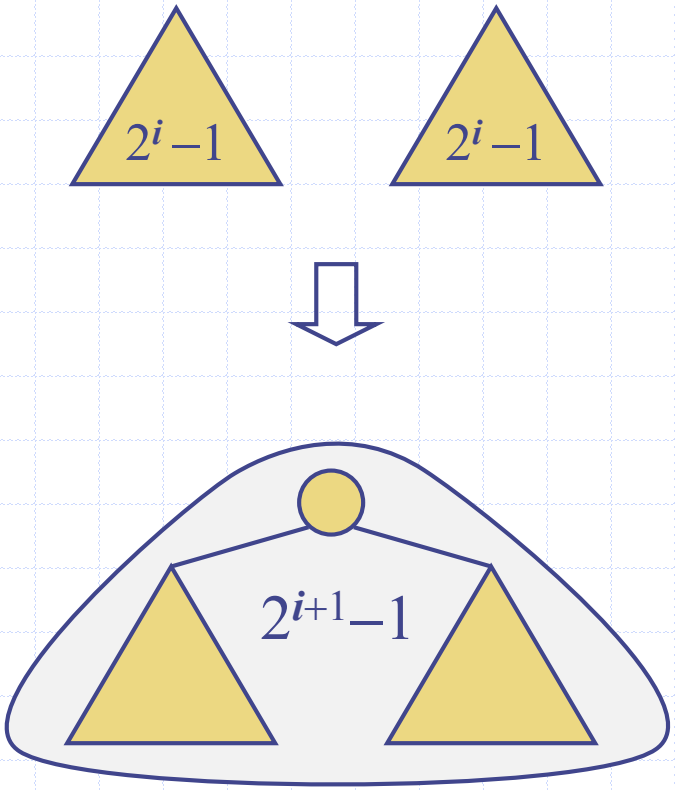
Merging Two Heaps

- 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

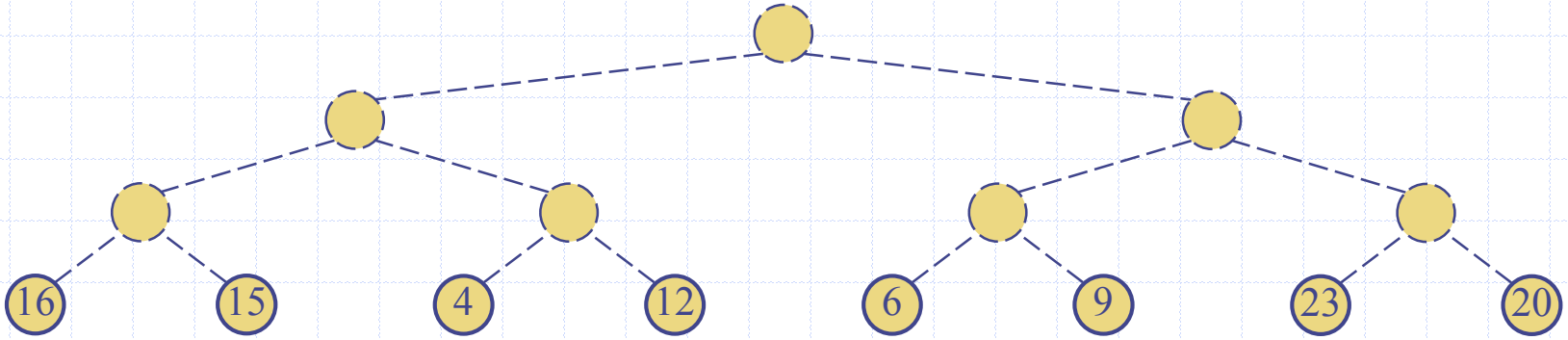


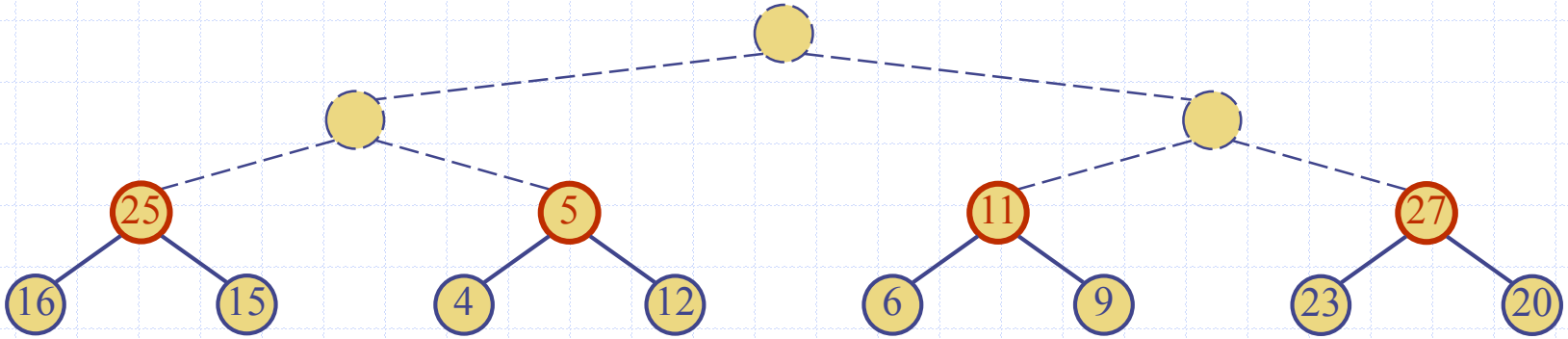
Bottom-up Heap Construction

- For simplicity of presentation, suppose $n = 2^h - 1$, that is, the tree is "full" with height $h=\log(n+1)$ (assuming root height = 1)
- Create $(n+1)/2$ heaps containing one vertex each.
- For $i=1,\ldots,h$:
  - Merge heaps pairwise by adding a new root node.
  - Propagate the new entry to restore the heap order, resulting in $(n+1)/2^{i+1}$ new heaps
  - Finish when all the nodes are added
- When, $n != 2^h - 1$, calculate number of leaves on height h and be careful for i = 1. Then proceed as above!

16 15 4 12 6 9 23 20 25 5 11 27 7 8 10

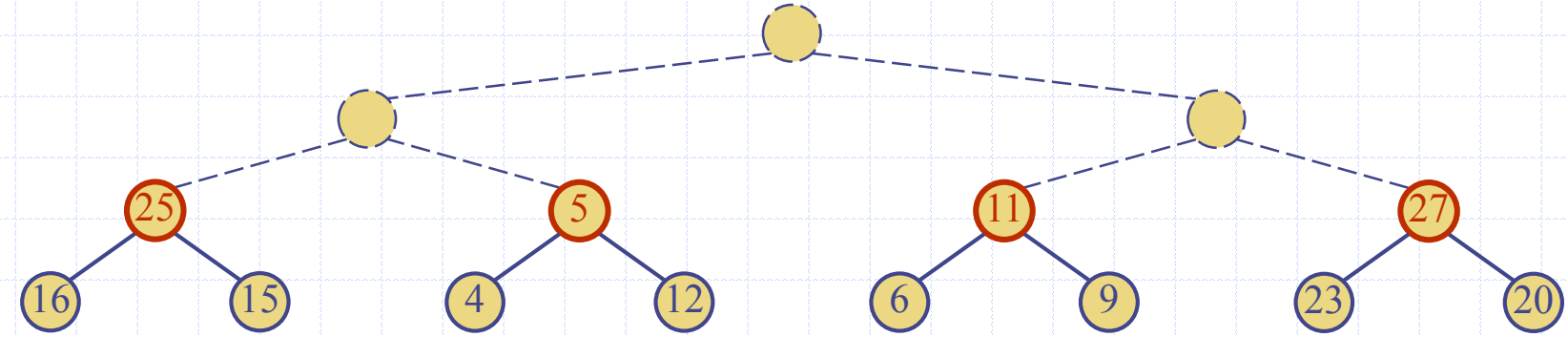
Example

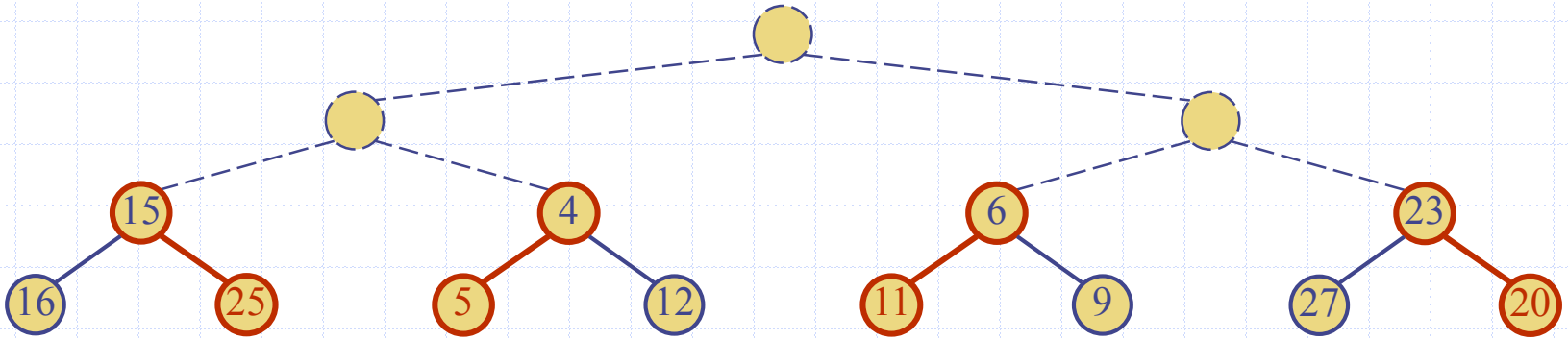




16 15 4 12 6 9 23 20 25 5 11 27 7 8 10

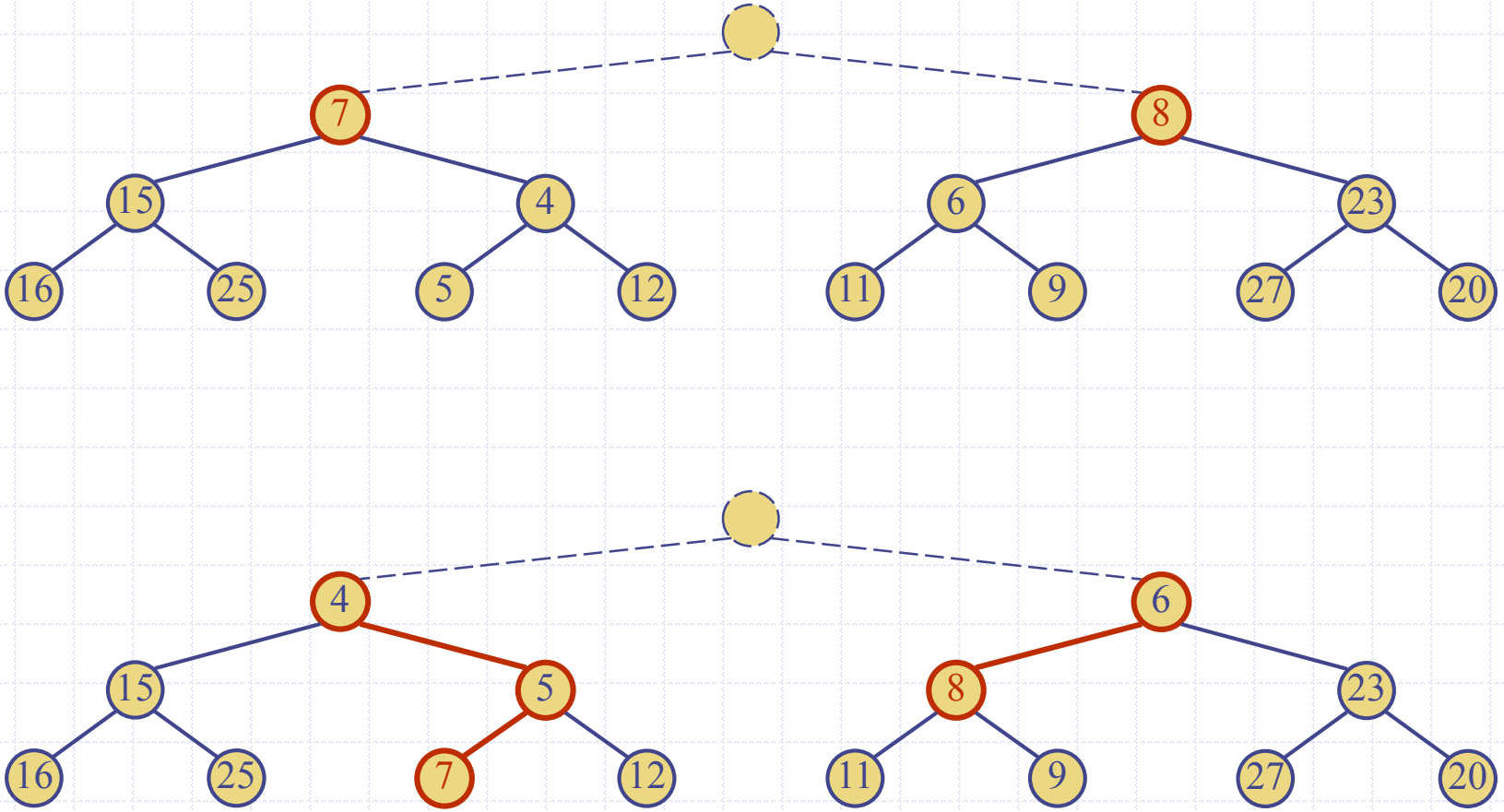
Example (contd.)





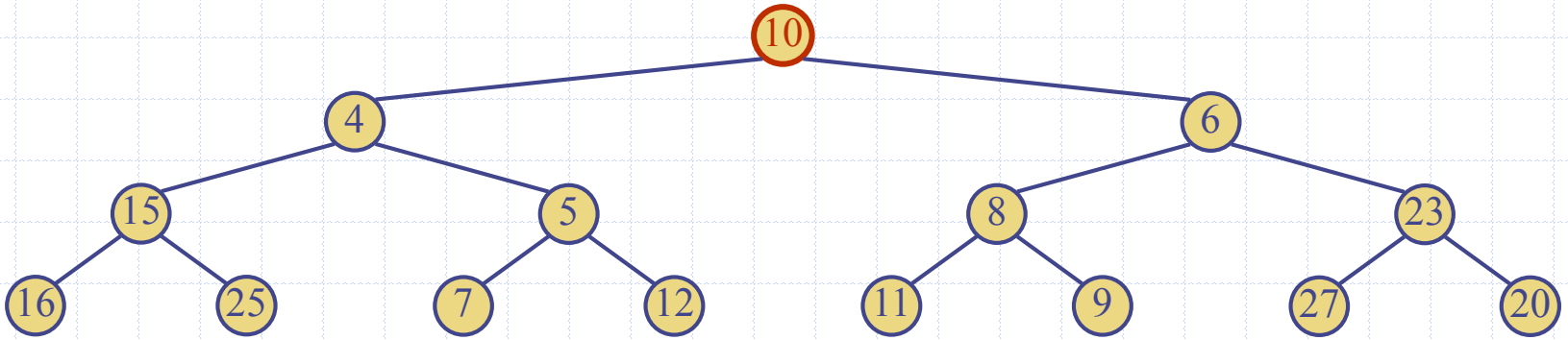
16 15 4 12 6 9 23 20 25 5 11 27 7 8 10

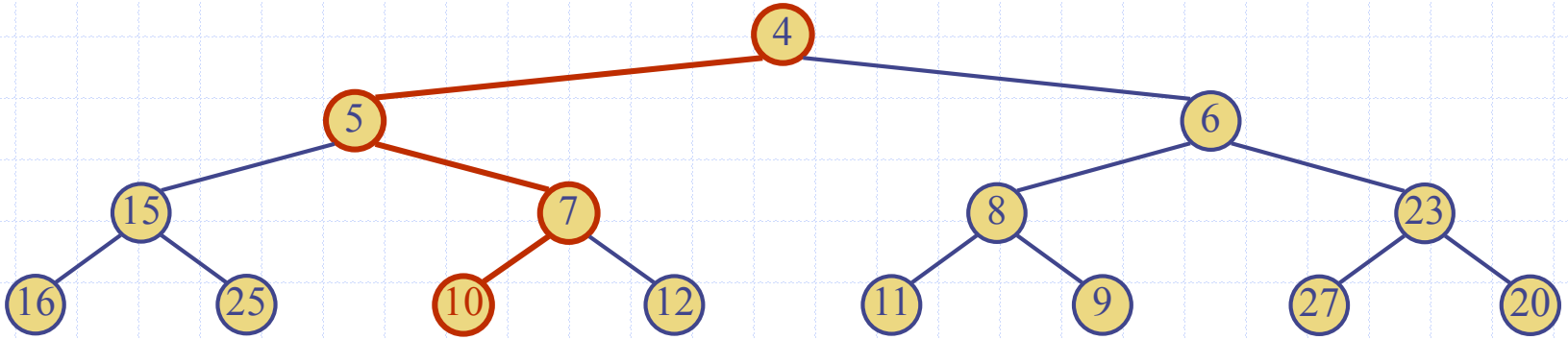
Example (contd.)



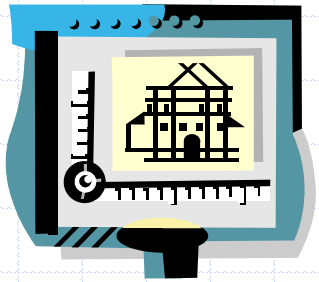
16 15 4 12 6 9 23 20 25 5 11 27 7 8 10

Example (end)

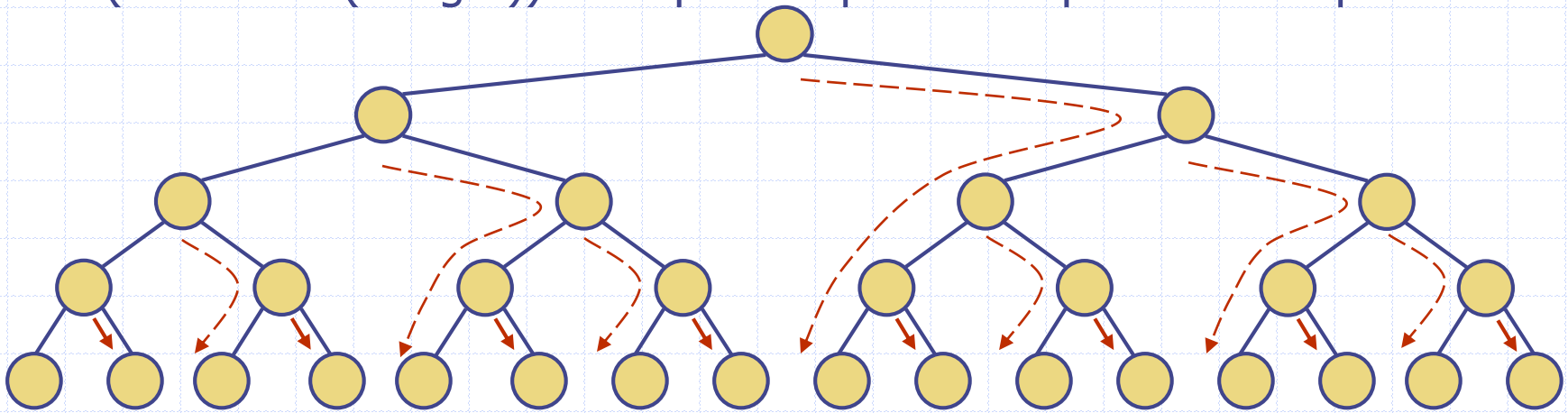




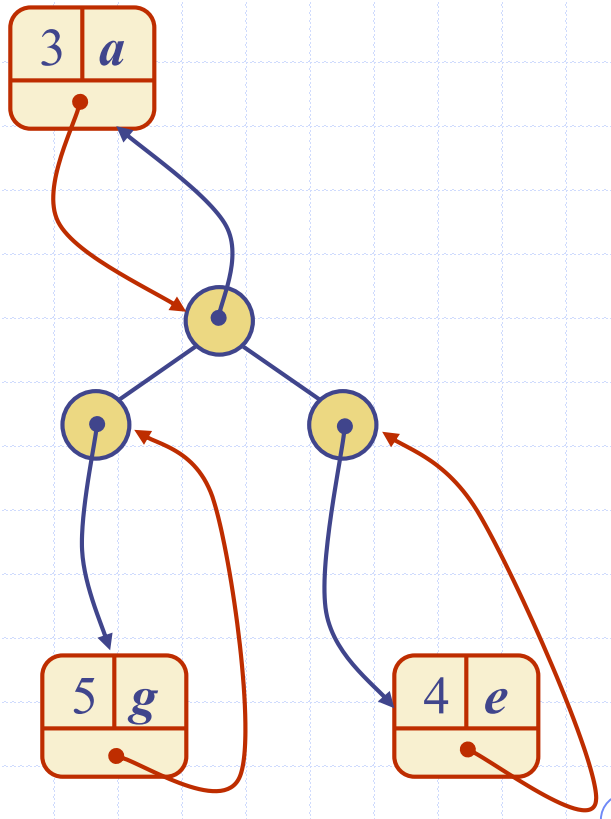
Analysis



- 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 (which is O(n log n)) and speeds up the first phase of heap-sort



Adaptable Priority Queues



Motivation

- Online trading system where orders to purchase and sell a 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?

Entry and Priority Queue ADTs

- An entry stores a (key, value) pair
- Entry ADT methods:
  - 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, **returns** the entry
  - removeMin()
    removes and returns the entry with smallest key
  - min()
    returns, but does not remove, an entry with smallest key
  - size(), isEmpty()

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 key of entry e of P; an error condition occurs if k is invalid (that is, k cannot be compared with other keys).
- replaceValue(e,v): Replace with v and return the value of entry e of P.

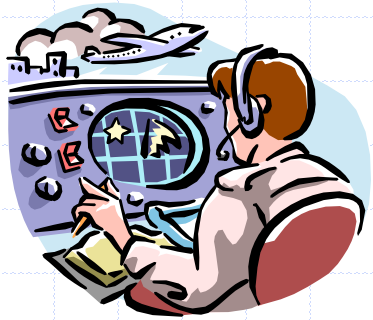
Example

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 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.

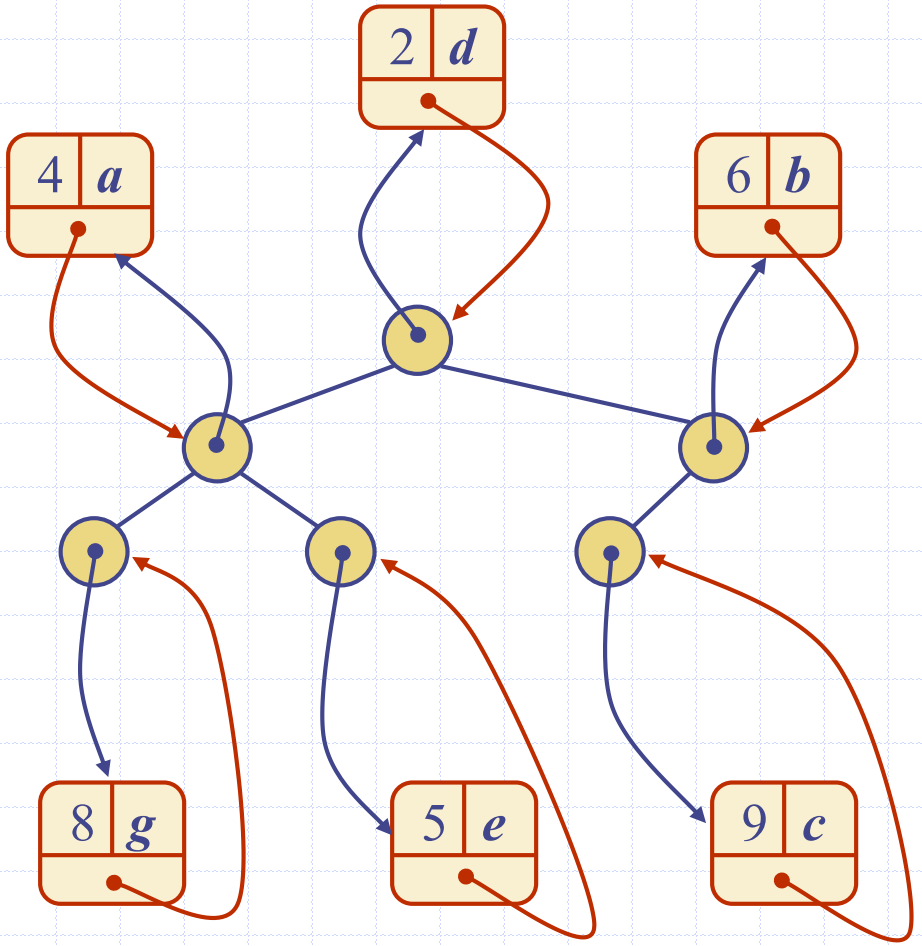
Location-Aware Entries



- A location-aware entry identifies and tracks the location of its (key, value) object within a data structure
- Intuitive notion:
  - Coat claim check
  - Valet claim ticket
  - Reservation number
- Main idea:
  - Since entries are created and returned from the data structure itself, it can return location-aware entries, thereby making future updates easier

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

- Improved times thanks to location-aware entries are highlighted in red (assuming we know the location)

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

Any alternatives?

- What if we do not know the location but still want to update (remove, replaceKey , replaceValue)?
- Heap based implementation:
  - O(n)
- Any other ideas?
  - Balanced Search Trees! (AVL, RB …) – How?
  - O(log n) to find the location
- Drawbacks?
  - min() is no longer O(1)
  - Other log n operations have higher constants

Home Exercise

- Update your array based heap implementation to be adaptable
- Starting from an AVL or a RB Tree, implement an adaptable priority queue

- I am not giving these as graded homeworks but you should really work on the home exercises…

Summary

- Priority Queues – get min or max fast
- Useful for multiple applications including sorting
  - Heap-sort is O(n log n)
- Using a heap data structure is a common way of implementing PQs and heaps themselves are mostly implemented with arrays
- We can merge two heaps, useful in building the heap the first time
- We can also use other data structures (e.g. BSTs)
- Again, the choice is based on the application at hand (e.g. could need adaptable PQs or very fast access to the min node)