

# **COMMONWEALTH OF AUSTRALIA**

## **Copyright Regulations 1969**

### **WARNING**

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

**Do not remove this notice.**

# Data structures and Algorithms

## Lecture 5: Priority Queues [GT 5]

André van Renssen

School of Computer Science

*Some content is taken from material  
provided by the textbook publisher Wiley.*



THE UNIVERSITY OF  
**SYDNEY**



# Priority Queue ADT

Special type of ADT map to store a collection of key-value items where we can only remove smallest key:

- **insert**(k, v): insert item with key **k** and value **v**
- **remove\_min**(): remove and return the item with smallest key
- **min**(): return item with smallest key
- **size**(): return how many items are stored
- **is\_empty**(): test if queue is empty

We can also have a max version of this min version, but we cannot use both versions at once.

# Example

A sequence of priority queue methods:

Method	Return value	Priority queue
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)}
remove_min()	(3,B)	{(5,A),(9,C)}
insert(7,D)		{(5,A),(7,D),(9,C)}
remove_min()	(5,A)	{(7,D),(9,C)}
remove_min()	(7,D)	{(9,C)}
remove_min()	(9,C)	{}
is_empty()	true	{}

# Application: Stock Matching Engines



At the heart of modern stock trading systems are highly reliable systems known as **matching engines**, which match the stock trades of buyers and sellers.

Buyers post bids to buy a number of shares of a given stock at or below a specified price

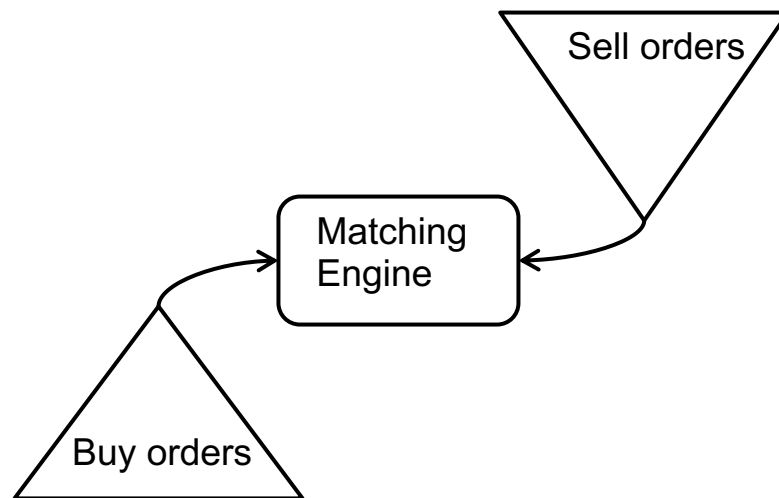
Sellers post offers (asks) to sell a number of shares of a given stock at or above a specified price.

STOCK: EXAMPLE.COM					
Buy Orders			Sell Orders		
Shares	Price	Time	Shares	Price	Time
1000	4.05	20 s	500	4.06	13 s
100	4.05	6 s	2000	4.07	46 s
2100	4.03	20 s	400	4.07	22 s
1000	4.02	3 s	3000	4.10	54 s
2500	4.01	81 s	500	4.12	2 s
			3000	4.20	58 s
			800	4.25	33 s
			100	4.50	92 s

# Application: Stock Matching Engines

Buy and sell orders are organized according to a **price-time priority**, where price has highest priority and time is used to break ties

When a new order is entered, the matching engine determines if a trade can be immediately executed and if so, then it performs the appropriate matches according to price-time priority.



STOCK: EXAMPLE.COM					
Buy Orders			Sell Orders		
Shares	Price	Time	Shares	Price	Time
1000	4.05	20 s	500	4.06	13 s
100	4.05	6 s	2000	4.07	46 s
2100	4.03	20 s	400	4.07	22 s
1000	4.02	3 s	3000	4.10	54 s
2500	4.01	81 s	500	4.12	2 s
			3000	4.20	58 s
			800	4.25	33 s
			100	4.50	92 s

# Application: Stock Matching Engines

A matching engine can be implemented with two **priority queues**, one for buy orders and one for sell orders.

This **data structure** performs element removals based on priorities assigned to elements when they are inserted.

```
while True:
    bid ← buy_orders.remove_max()
    ask ← sell_orders.remove_min()
    if bid.price ≥ ask.price then
        carry out trade (bid, ask)
    else
        buy_orders.insert(bid)
        sell_orders.insert(ask)
```

STOCK: EXAMPLE.COM					
Buy Orders			Sell Orders		
Shares	Price	Time	Shares	Price	Time
1000	4.05	20 s	500	4.06	13 s
100	4.05	6 s	2000	4.07	46 s
2100	4.03	20 s	400	4.07	22 s
1000	4.02	3 s	3000	4.10	54 s
2500	4.01	81 s	500	4.12	2 s
			3000	4.20	58 s
			800	4.25	33 s
			100	4.50	92 s

# Sequence-based Priority Queue

## Unsorted list implementation



- **insert** in  $O(1)$  time since we can insert the item at the beginning or end of the sequence
- **remove\_min** and **min** in  $O(n)$  time since we have to traverse the entire list to find the smallest key

## Sorted list implementation



- **insert** in  $O(n)$  time since we have to find the place where to insert the item
- **remove\_min** and **min** in  $O(1)$  time since the smallest key is at the beginning

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

# Priority Queue Sorting

We can use a priority queue to sort a list of keys:

1. iteratively insert keys into an empty priority queue
2. iteratively **remove\_min** to get the keys in sorted order

Complexity analysis:

- **n** insert operations
- **n** remove\_min operations

Either sequence-based  
implementation take  **$O(n^2)$**

```
def priority_queue_sorting(A):  
    pq ← new priority queue  
    n ← size(A)  
    for i in [0:n] do  
        pq.insert(A[i])  
    for i in [0:n] do  
        A[i] = pq.remove_min()
```

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

# Selection-Sort

Variant of pq-sort using unsorted sequence implementation:

1. inserting elements with  $n$  insert operations takes  $O(n)$  time
2. removing elements with  $n$  remove\_min operations takes  $O(n^2)$

Can be done in place  
(no need for extra space)

Top level loop invariant:

- $A[0:i]$  is sorted
- $A[i:n]$  is the priority queue  
and all  $\geq A[i-1]$

```
def selection_sort(A):  
    n ← size(A)  
    for i in [0:n] do  
        # find s ≥ i minimizing A[s]  
        s ← i  
        for j in [i:n] do  
            if A[j] < A[s] then  
                s ← j  
        # swap A[i] and A[s]  
        A[i], A[s] ← A[s], A[i]
```

# Selection-Sort Example

i	A	s
0	2, 4, 8, <u>2</u> , 5, 3, 9	3
1	2, <u>4</u> , 8, 7, 5, <u>3</u> , 9	5
2	2, 3, <u>8</u> , 7, 5, <u>4</u> , 9	5
3	2, 3, 4, <u>2</u> , <u>5</u> , 8, 9	4
4	2, 3, 4, 5, <u>2</u> , 8, 9	4
5	2, 3, 4, 5, 7, <u>8</u> , 9	5
6	2, 3, 4, 5, 7, 8, <u>9</u>	6

```
def selection_sort(A):  
    n ← size(A)  
    for i in [0:n] do  
        # find s ≥ i minimizing A[s]  
        s ← i  
        for j in [i:n] do  
            if A[j] < A[s] then  
                s ← j  
        # swap A[i] and A[s]  
        A[i], A[s] ← A[s], A[i]
```

# Insertion-Sort

Variant of pq-sort using sorted sequence implementation:

1. inserting elements with  $n$  insert operations takes  $O(n^2)$  time
2. removing elements with  $n$  remove\_min operations takes  $O(n)$

Can be done in place  
(no need for extra space)

Top level loop invariant:

- $A[0:i]$  is the priority queue  
(and thus sorted)
- $A[i:n]$  is yet-to-be-inserted

```
def insertion_sort(A):  
    n ← size(A)  
    for i in [1:n] do  
        x ← A[i]  
        # move forward entries > x  
        j ← i  
        while j > 0 and x < A[j-1] do  
            A[j] ← A[j-1]  
            j ← j - 1  
        # if j>0 ⇒ x ≥ A[j-1]  
        # if j<i ⇒ x < A[j+1]  
        A[j] ← x
```

# Insertion-Sort Example

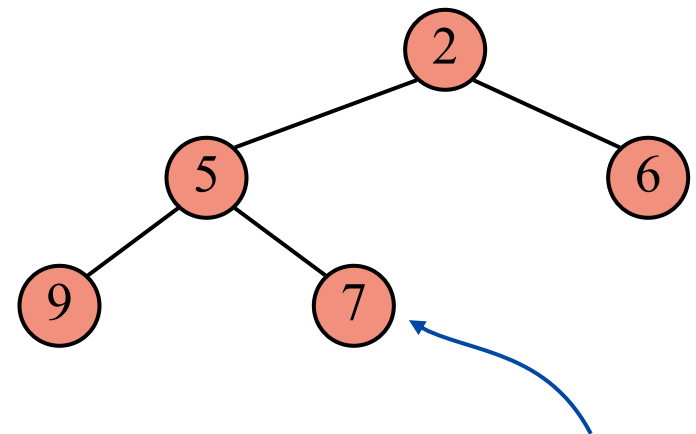
i	A	i
1	Z, <u>4</u> , 8, 2, 5, 3, 9	0
2	4, 7, <u>8</u> , 2, 5, 3, 9	2
3	<u>4</u> , 7, 8, <u>2</u> , 5, 3, 9	0
4	2, 4, Z, 8, <u>5</u> , 3, 9	2
5	2, <u>4</u> , 5, 7, 8, <u>3</u> , 9	1
6	2, 3, 4, 5, 7, 8, <u>2</u>	6

```
def insertion_sort(A):  
    n ← size(A)  
    for i in [1:n] do  
        x ← A[i]  
        # move forward entries > x  
        j ← i  
        while j > 0 and x < A[j-1] do  
            A[j] ← A[j-1]  
            j ← j - 1  
        # if j>0 ⇒ x ≥ A[j-1]  
        # if j<i ⇒ x < A[j+1]  
        A[j] ← x
```

# Heap data structure (min-heap)

A **heap** is a binary tree storing (key, value) items at its nodes, satisfying the following properties:

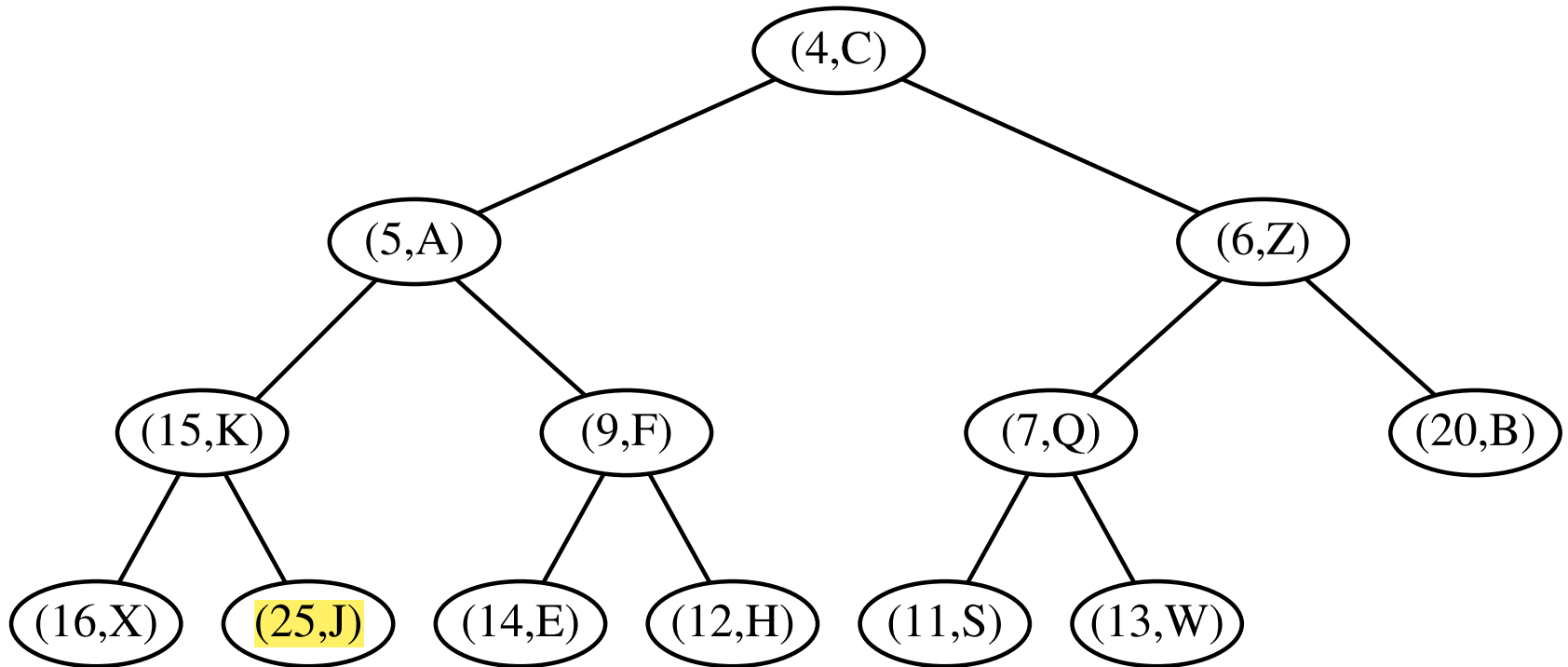
1. **Heap-Order:** for every node  $m \neq \text{root}$ ,  
 $\text{key}(m) \geq \text{key}(\text{parent}(m))$



2. **Complete Binary Tree:** let  $h$  be the height
  - every level  $i < h$  is full (i.e., there are  $2^i$  nodes)
  - remaining nodes take leftmost positions of level  $h$

The **last node** is the rightmost node of maximum depth

# Example

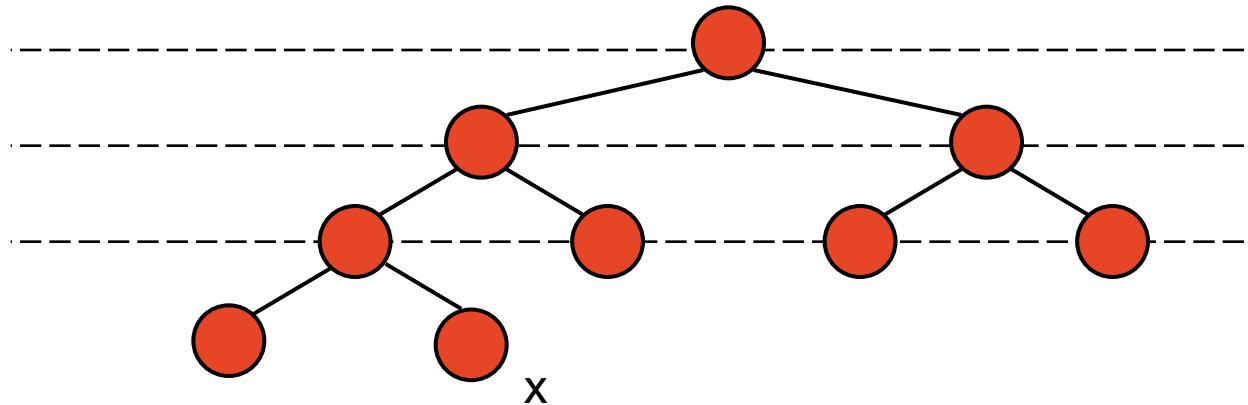


# Minimum of a Heap

**Fact:** The root always holds the smallest key in the heap

**Proof:**

- Suppose the minimum key is at some internal node  $x$
- Because of the heap property, as we move up the tree, the keys can only get smaller (assuming repeats, otherwise contradiction)
- If  $x$  is not the root, then its parent must also hold a smallest key
- Keep going until we reach the root

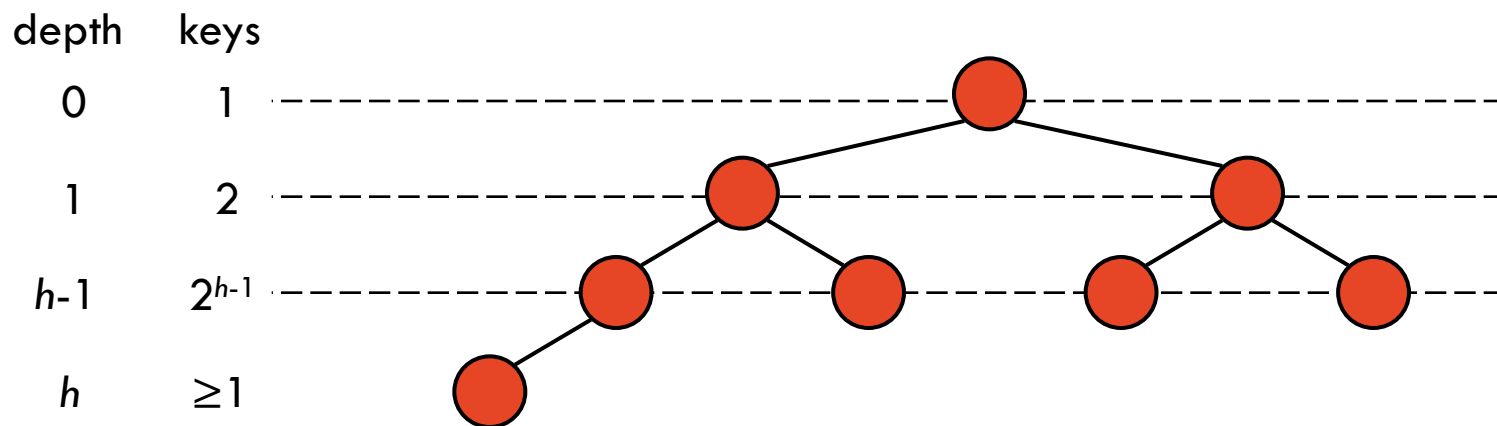


# Height of a Heap

**Fact:** A heap storing  $n$  keys has height  $\log n$

**Proof:**

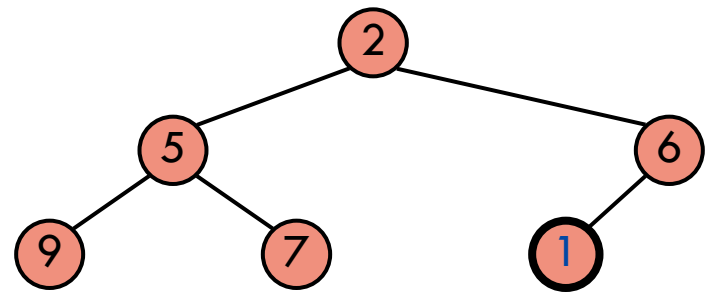
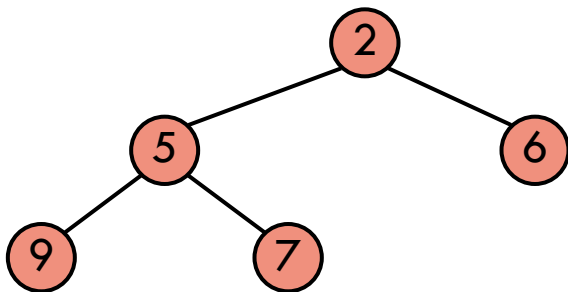
- Let  $h$  be the height of a heap storing  $n$  keys
- Since there are  $2^i$  keys at depth  $i = 0, \dots, h - 1$  and at least one key at depth  $h$ , we have  $n \geq 1 + 2 + 4 + \dots + 2^{h-1} + 1$
- Thus,  $n \geq 2^h$ , applying  $\log_2$  on both sides,  $\log_2 n \geq h$



# Insertion into a Heap

- Create a new node with given key
- Find location for new node
- Restore the heap-order property

insert(1)



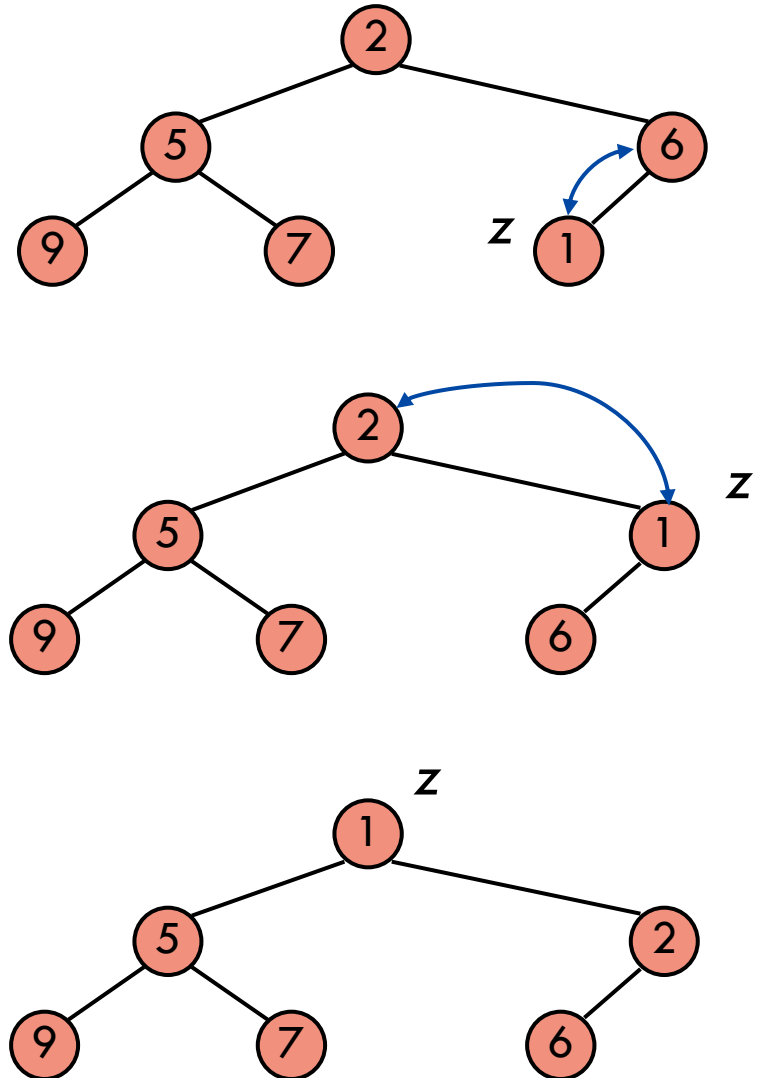
# Upheap

Restore heap-order property by swapping keys along upward path from insertion point

```
def up_heap(z):  
    while z  $\neq$  root and  
        key(parent(z)) > key(z) do  
        swap key of z and parent(z)  
        z  $\leftarrow$  parent(z)
```

**Correctness:** after swapping the subtree rooted at **z** has the property

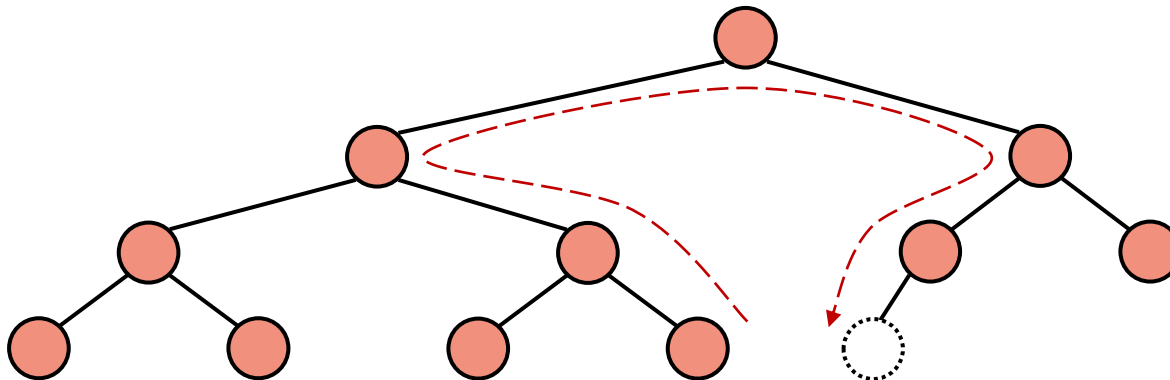
**Complexity:**  $O(\log n)$  time because the height of the heap is  $\log n$



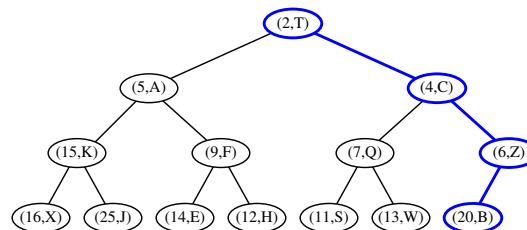
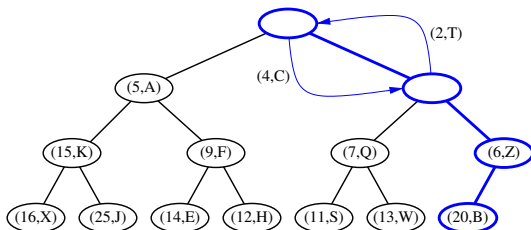
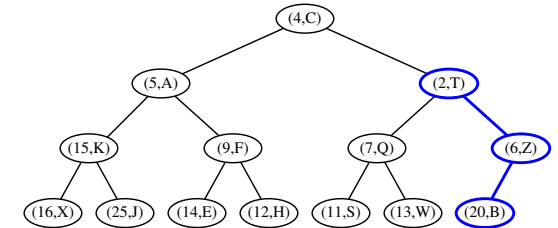
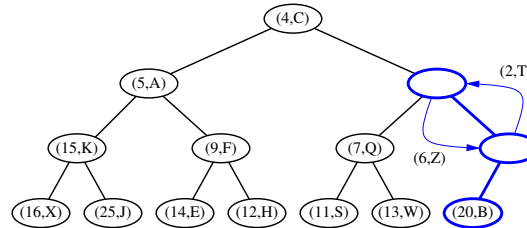
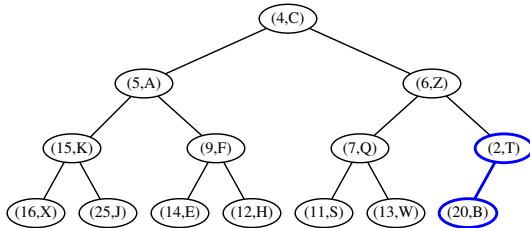
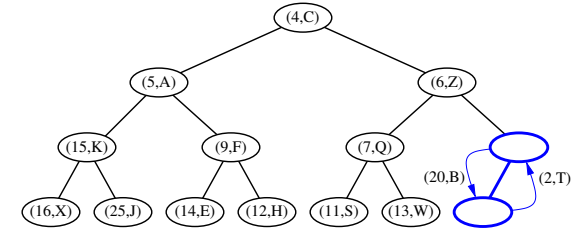
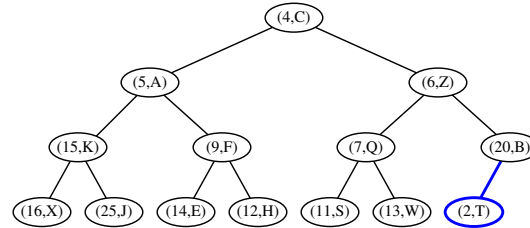
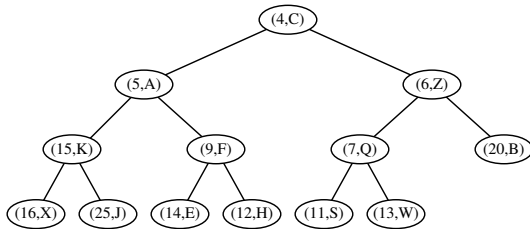
# Finding the position for insertion

- start from the last node
- go up until a left child or the root is reached
- if we reach the root then need to open a new level
- otherwise, go to the sibling (right child of parent)
- go down left until a leaf is reached

Complexity of this search is  $O(\log n)$  because the height is  $\log n$ .  
Thus, overall complexity of insertion is  $O(\log n)$  time



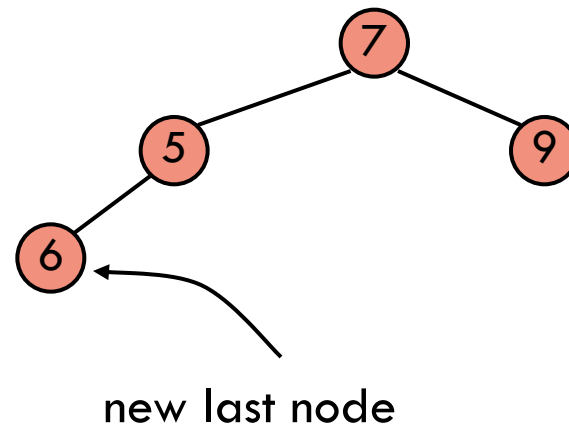
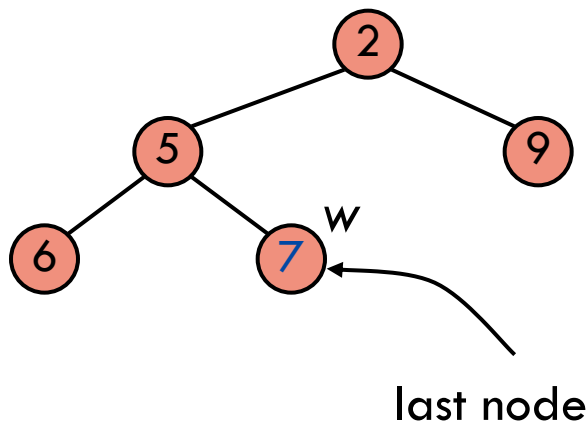
# Example insertion



# Removal from a Heap

- Replace the root key with the key of the last node  $w$
- Delete  $w$
- Restore the heap-order property

`remove_min()`



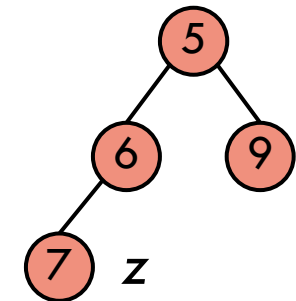
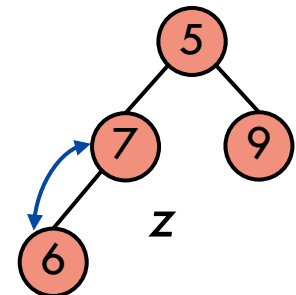
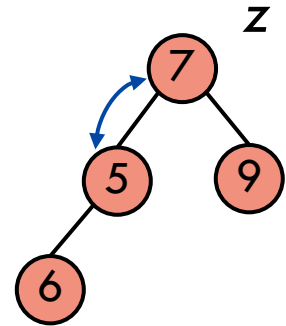
# Downheap

Restore heap-order property by swapping keys along downward path from the root

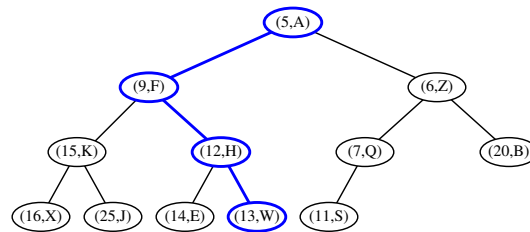
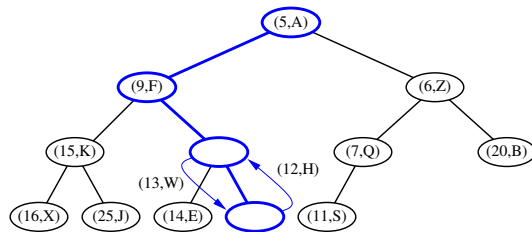
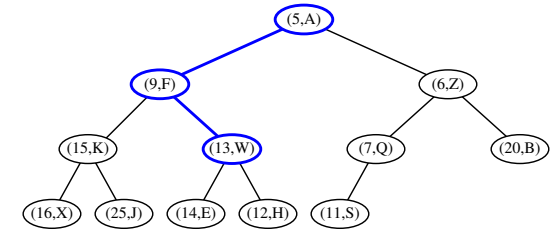
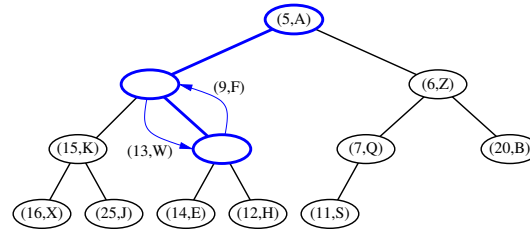
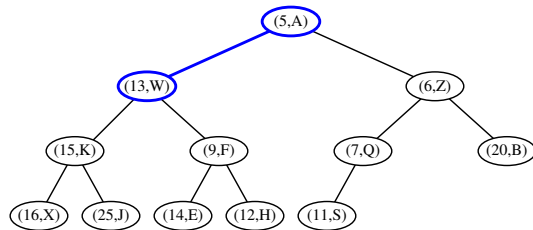
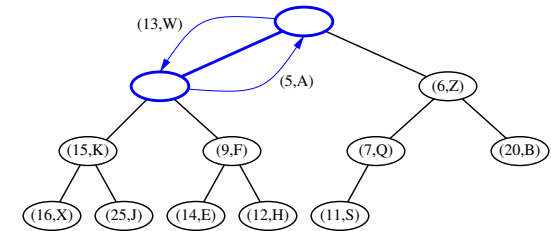
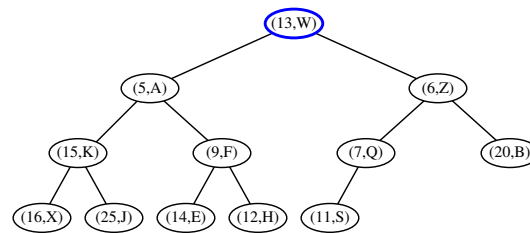
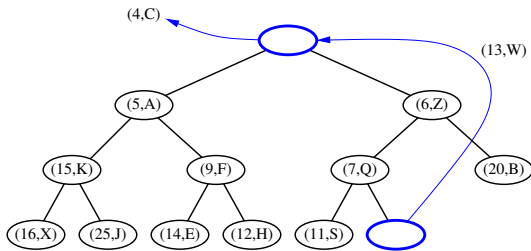
```
def down_heap(z):  
    while z has child with  
        key(child) < key(z) do  
        x ← child of z with smallest key  
        swap keys of x and z  
        z ← x
```

**Correctness:** after swap **z** heap-order property is restored up to level of **z**

**Complexity:**  $O(\log n)$  time because the height of the heap is  $\log n$



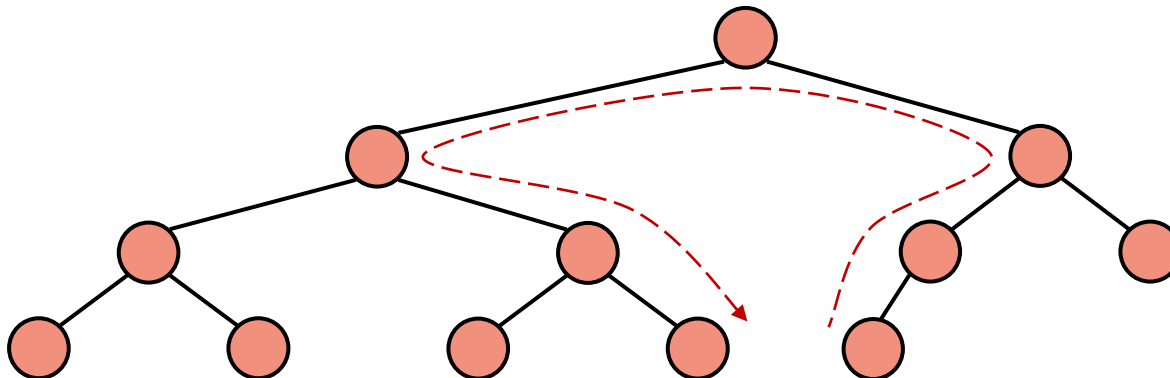
# Example removal



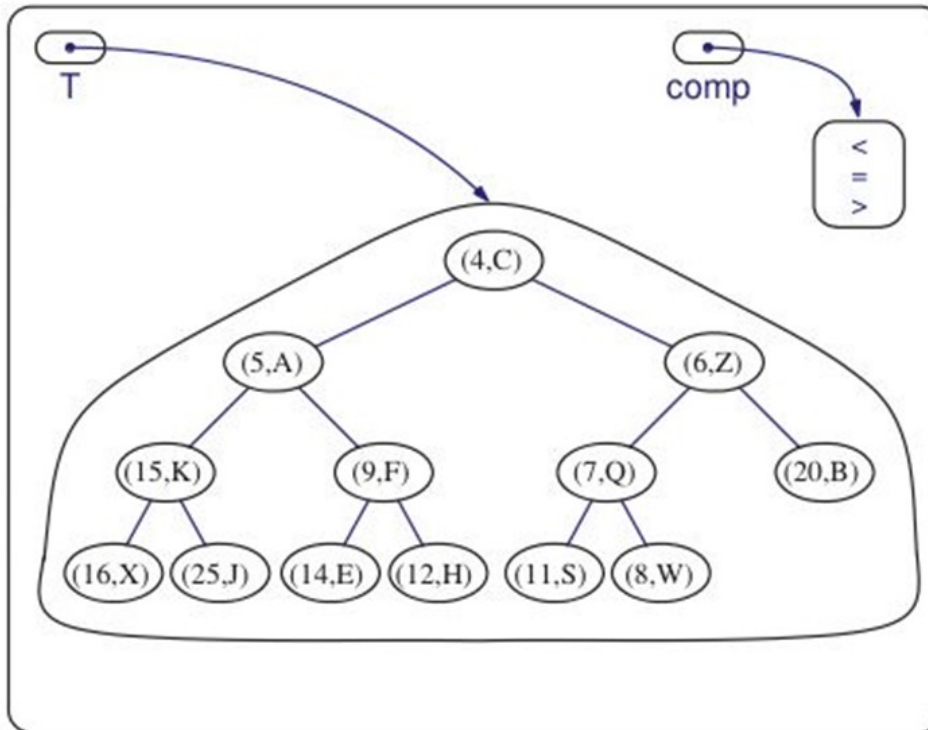
# Finding next last node after deletion

- start from the (old) last node
- go up until a right child or the root is reached
- if we reach the root then need to close a level
- otherwise, go to the sibling (left child of parent)
- go down right until a leaf is reached

Complexity of this search is  $O(\log n)$  because the height is  $\log n$ .  
Thus, overall complexity of deletion is  $O(\log n)$  time



# Heap-based implementation of a priority queue



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

# Heap-Sort

Consider a priority queue with  $n$  items implemented with a heap:

- the space used is  $O(n)$
- methods **insert** and **remove\_min** take  $O(\log n)$

Recall that priority-queue sorting uses:

- $n$  insert ops
- $n$  remove\_min ops

**Heap-sort** is the version of priority-queue sorting that implements the priority queue with a heap. It runs in  $O(n \log n)$  time.

# Heap-in-array implementation

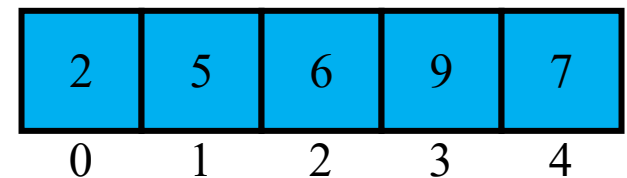
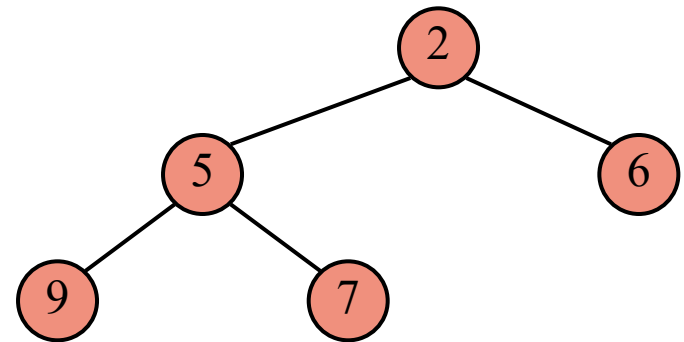
We can represent a heap with  $n$  keys by means of an array of length  $n$

Special nodes:

- root is at 0
- last node is at  $n-1$

For the node at index  $i$ :

- the left child is at index  $2i+1$
- the right child is at index  $2i+2$
- Parent is at index  $\lfloor (i-1)/2 \rfloor$



# Refinements and Generalization

Heap-sort can be arranged to work in place using part of the array for the output and part for the priority queue

A heap on  $n$  keys can be constructed in  $O(n)$  time. But the  $n$  `remove_min` still take  $O(n \log n)$  time

Sometimes it is useful to support a few more operations (all are given a pointer to  $e$ ):

- `remove`( $e$ ): Remove item  $e$  from the priority queue
- `replace_key`( $e, k$ ): update key of item  $e$  with  $k$
- `replace_value`( $e, v$ ): update value of item  $e$  with  $v$

## Summary: Priority queue implementations

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	$O(1)$	$O(1)$	$O(1)$

# Implementing a Priority Queue

Entries: An object that keeps track of the associations between keys and values

Comparators: A function or an interface to compare entry objects

**compare**(a, b): returns an integer **i** such that

- **i** < 0 if **a** < **b**,
- **i** = 0 if **a** = **b**
- **i** > 0 if **a** > **b**

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

## Stock Application Revisited



Online trading system where orders are stored in two priority queues (one for sell orders and one for buy orders) as  $(p, t, s)$  entries:

- The key is  $(p, t)$ , the price of the order  $p$  and the time  $t$  such that we first sort by  $p$  and break ties with  $t$
- The value is  $s$ , the number of shares the order is for

How do we implement the following:

- What should we do when a new order is placed?
- 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?