# Assignment #2
# Discrete Event Simulation
# Let’s do some scheduling…

# Basic Specifications

You want to analyze different CPU scheduling algorithms so you decide to approach this by writing a **Discrete Event Simulation** (DES).

A number of simultaneous processes (threads)  will be simulated, each alternating between bursts of CPU usage and I/O waiting.

The process data will be read in from a data file.

# DES Overview

**Time Steps**

- Set up a loop, jump forward in time with each iteration to whenever the next meaningful event occurs. Some time steps may be small (even 0 if two or more things happen at the same time) and some may be large.

# DES Overview

**Triggers**

- Some events may trigger other events, which are then put on the schedule to be processed when their time comes.

# DES Overview

## Data Structures

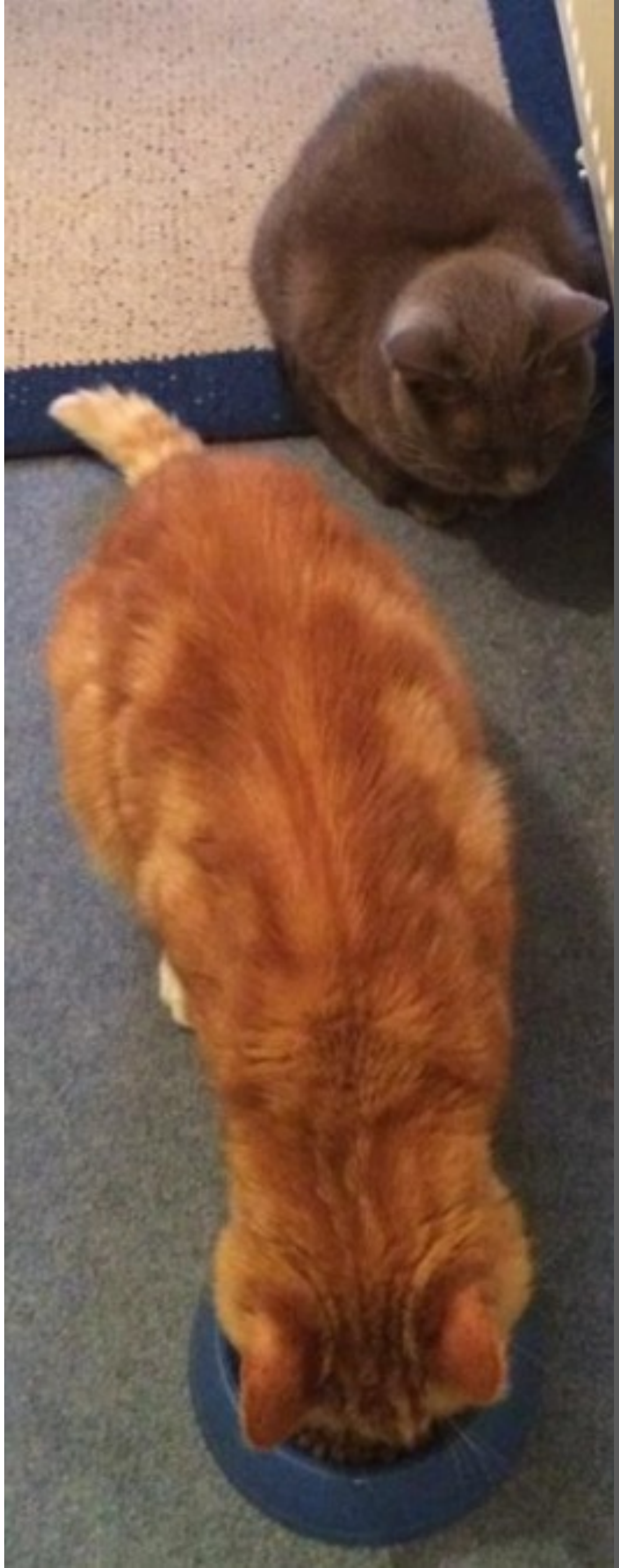
- An important data structure in DES is the **Priority Queue** which holds events to be scheduled.

# Priority Queue ADT

- A **priority queue** stores a collection of items.
- An item is a pair *(key, element).*

## Main Methods

- *insertItem(k, e)* - inserts an item with key *k* and element *e*
- *e = removeMin()* - removes the item with smallest key and returns its element *e*



# Priority Queue ADT

**Additional Methods**

- *minKey()* returns, but does not remove, the smallest key of an item
- *minElement()* returns, but does not remove, the element of an item with the smallest key
- *size(), isEmpty()*

# Priority Queue Example

| Operator | Output | Priority Queue |
| --- | --- | --- |
| insertItem(5, A) | _ | (5,A) |
| insertItem(9, C) | _ | (5,A),(9,C) |
| insertItem(3, B) | _ | (3,B),(5,A),(9,C) |
| insertItem(7, D) | _ | (3,B),(5,A),(7,D),(9,C) |
| minElement() | B | $\boxed{(3,B)}$,(5,A),(7,D),(9,C) |
| minKey() | 3 | (3,B),(5,A),(7,D),(9,C) |
| removeMin() | B | (5,A),(7,D),(9,C) |
| size() | 3 | $\boxed{(5,A)}$,(7,D),(9,C) |
| removeMin() | A | (7,D),(9,C) |
| removeMin() | D | (9,C) |
| removeMin() | C | |
| removeMin() | error | |
| isEmpty() | true | |

# Total Order Relation

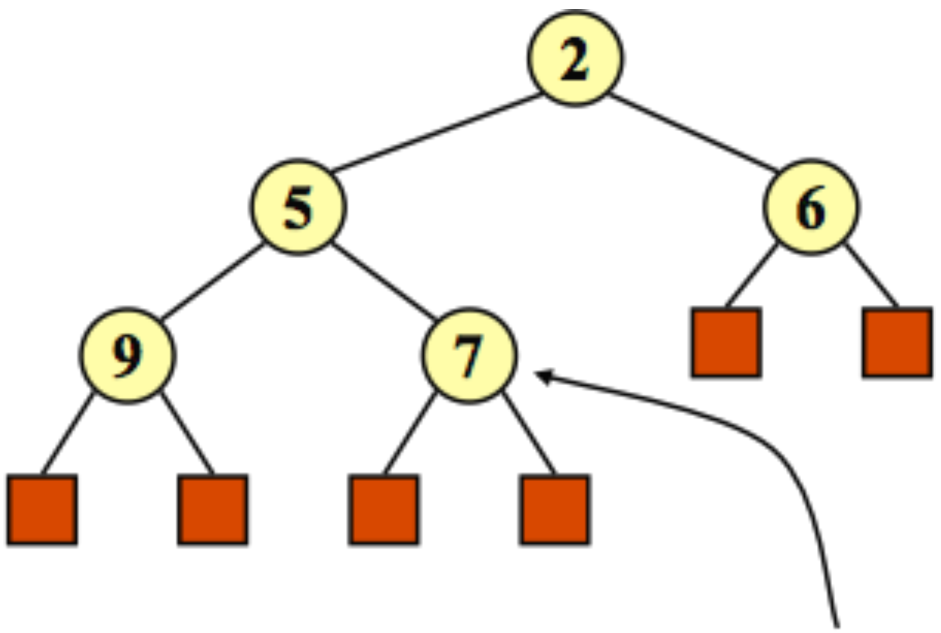
- Keys in a **Priority Queue** can be arbitrary objects on which an order is defined, *e.g.* simulation time.
- Two distinct items in a priority queue can have the same key. For example, two processes arrive at the same time.
- For a pair of events: *(t1, p1)* and *(t2, p2)*, we define a "**happen before**" relation such that:

$$(t1, p1) \longrightarrow (t2, p2) \text{ if } t1 < t2$$

$$(t1, p1) \longrightarrow (t2, p2) \text{ if } t1 == t2 \text{ and } p1 < p2$$

# Using a Heap to Implement a Priority Queue

A **heap** is a **binary tree** storing keys at its internal nodes and satisfying the following properties:



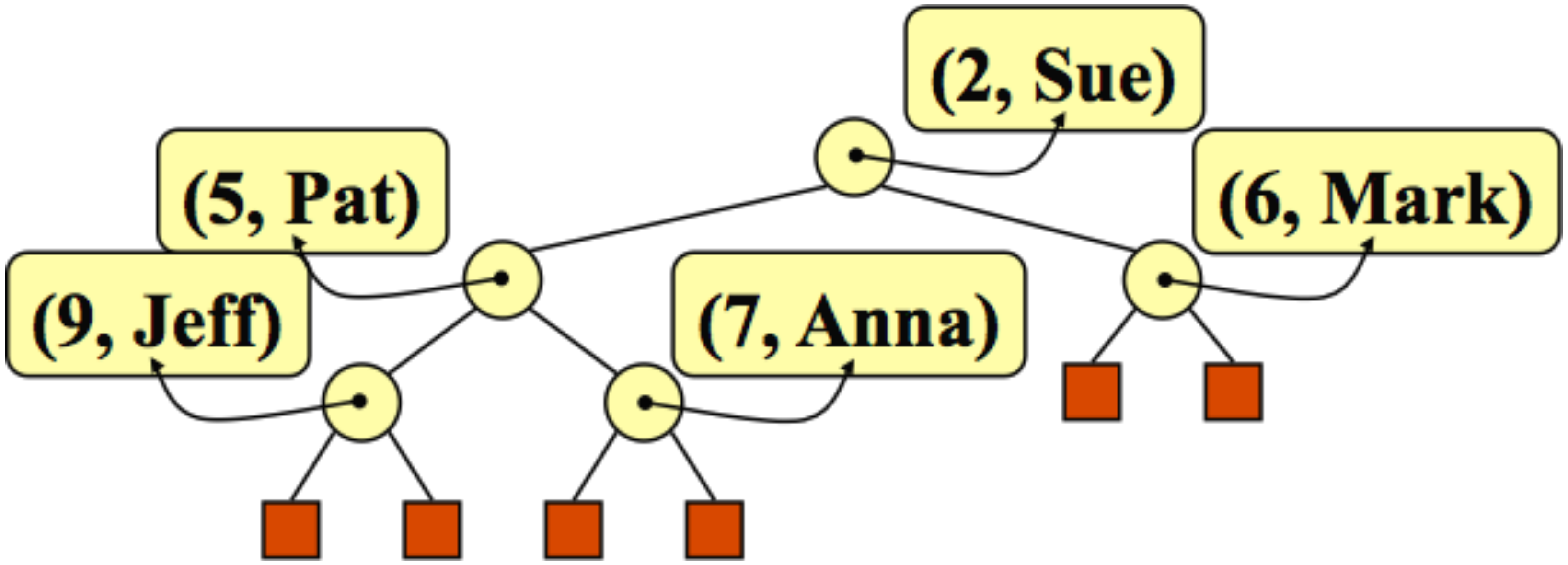

**Heap-Order**: for every internal node *v* other than the *root*, *key(v) >= key(parent(v))*

**Complete Binary Tree**: let *h* be the height of the heap

- *for i = 0, … , h − 1* there are *2i* 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 internal node of depth *h − 1*

# Heaps and Priority Queues

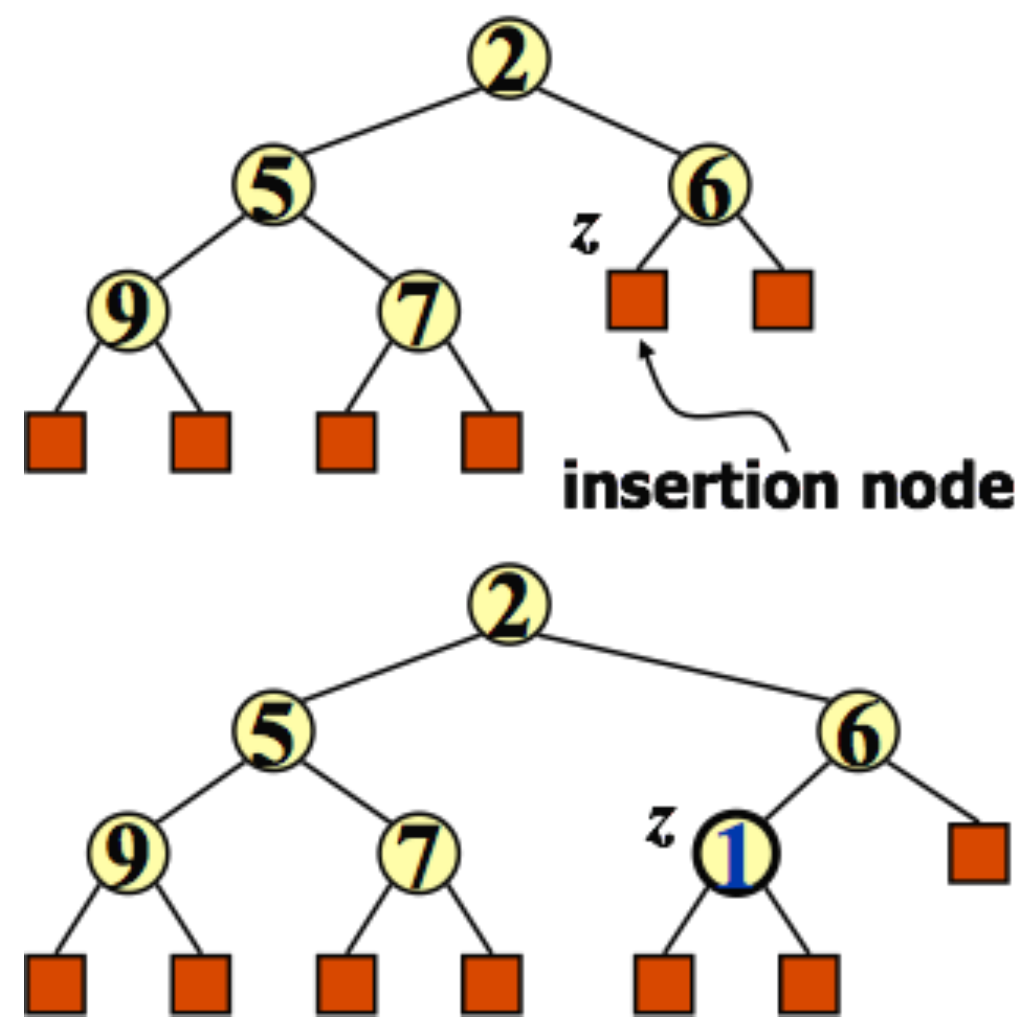
- We can use a **heap** to implement a priority queue.
- We store a *(key, element)* item at each internal node.
- We keep track of the position of the last node.
- For simplicity, we show only the keys in the pictures.



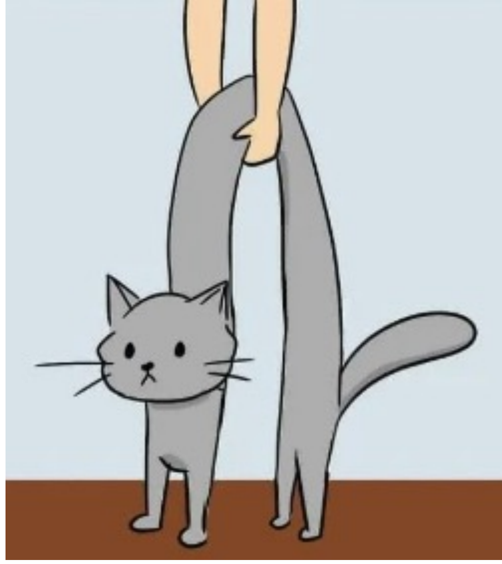
# Insertion into a Heap

The insertion algorithm consists of three steps:

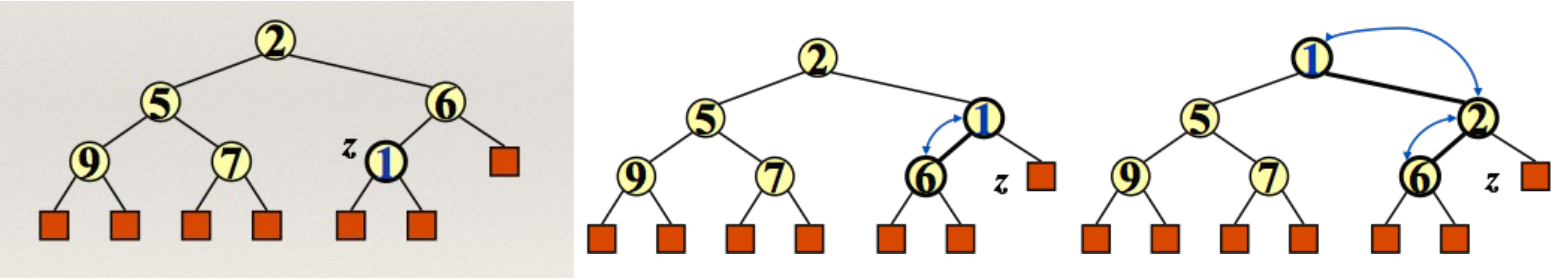
1. Find the insertion position *z* (the new last node).
2. Store *k* at *z* and expand *z* into an internal node.
3. Restore the heap-order property (discussed next).




# Upheap



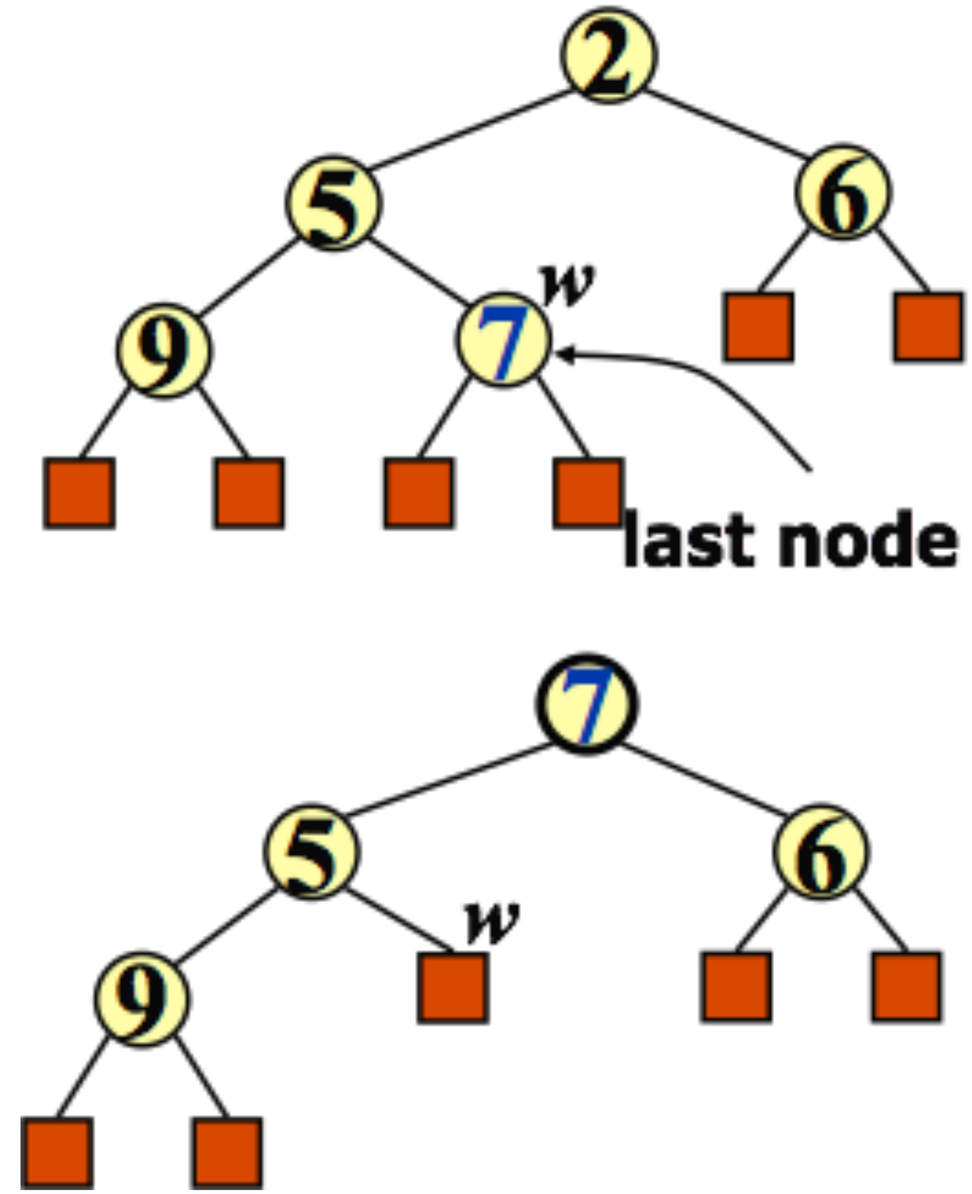
- After the insertion of a new key *k*, the heap-order property may be violated.
- Algorithm **upheap** restores the heap-order property by swapping *k* along an *upward* path from the insertion node.
- **Upheap** terminates when the key *k* reaches the *root* or a node whose parent has a key smaller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



# Removal from a Heap

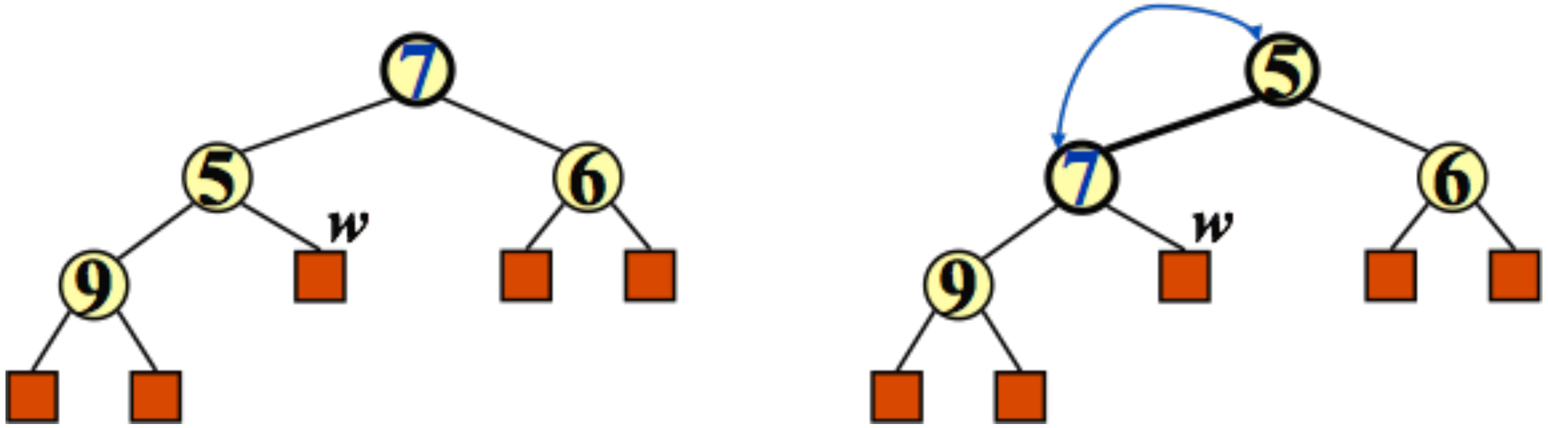
The removal algorithm consists of three steps:

1. Replace the *root* key with the key of the last node *w.*
2. Delete *w.*
3. 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.



# General DES Pseudo Code


```
Initialize PQ
while( PQ not empty ) {
   extract an Event from the PQ
   update time to match the Event
   switch ( type of Event ) {
      Process this event, possibly adding new Events to the PQ
   }
}
Process statistics collected during Event processing & report
```

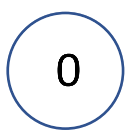
# Input File Format for A2

number_of_processes  thread_switch  process_switch
process_number(1)   number_of_threads(1)
  thread_number(1) arrival_time(1)  number_of_CPU(1)
  1   cpu_time  io_time
  2   cpu_time  io_time
  .
  .
  number_of_CPU(1)   cpu_time
…

# Example Input File

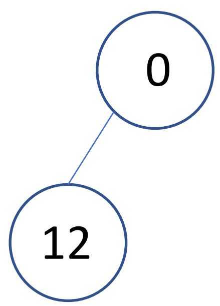
| | |
|---|---|
| 2 3 7 | number_of_processes thread_switch process_switch |
| 1 4 | process_number (1) number_of_threads (1) |
| 1 0 6 | thread_number (1) arrival_time (1) number_of_CPU (1) |
| 1 15 400 | 1 cpu_time io_time |
| 2 18 200 | 2 cpu_time io_time |
| 3 15 100 | 3 cpu_time io_time |
| 4 15 400 | 4 cpu_time io_time |
| 5 25 100 | 5 cpu_time io_time |
| 6 240 | 6 cpu_time |
| 2 12 4 | thread_number(2) arrival_time(2) number_of_CPU(2) |
| … | |

Insert 0



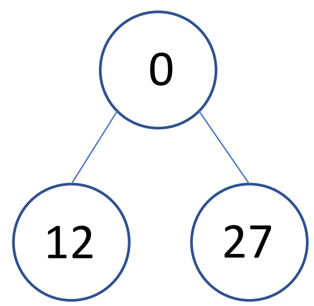

Array: 0

Insert 12



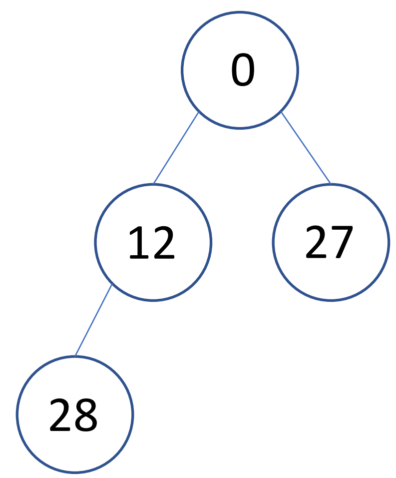

Array: 0 12

Insert 27



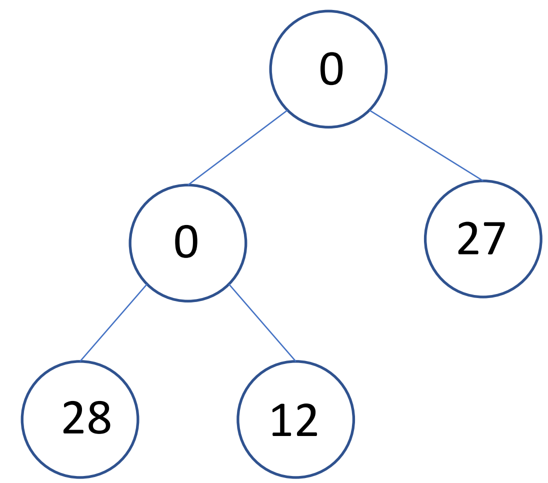

Array: 0 12 27

Insert 28



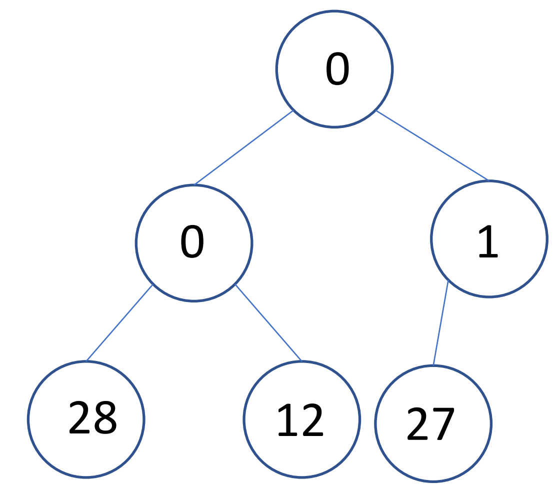

Array: 0 12 27 28

Insert 0

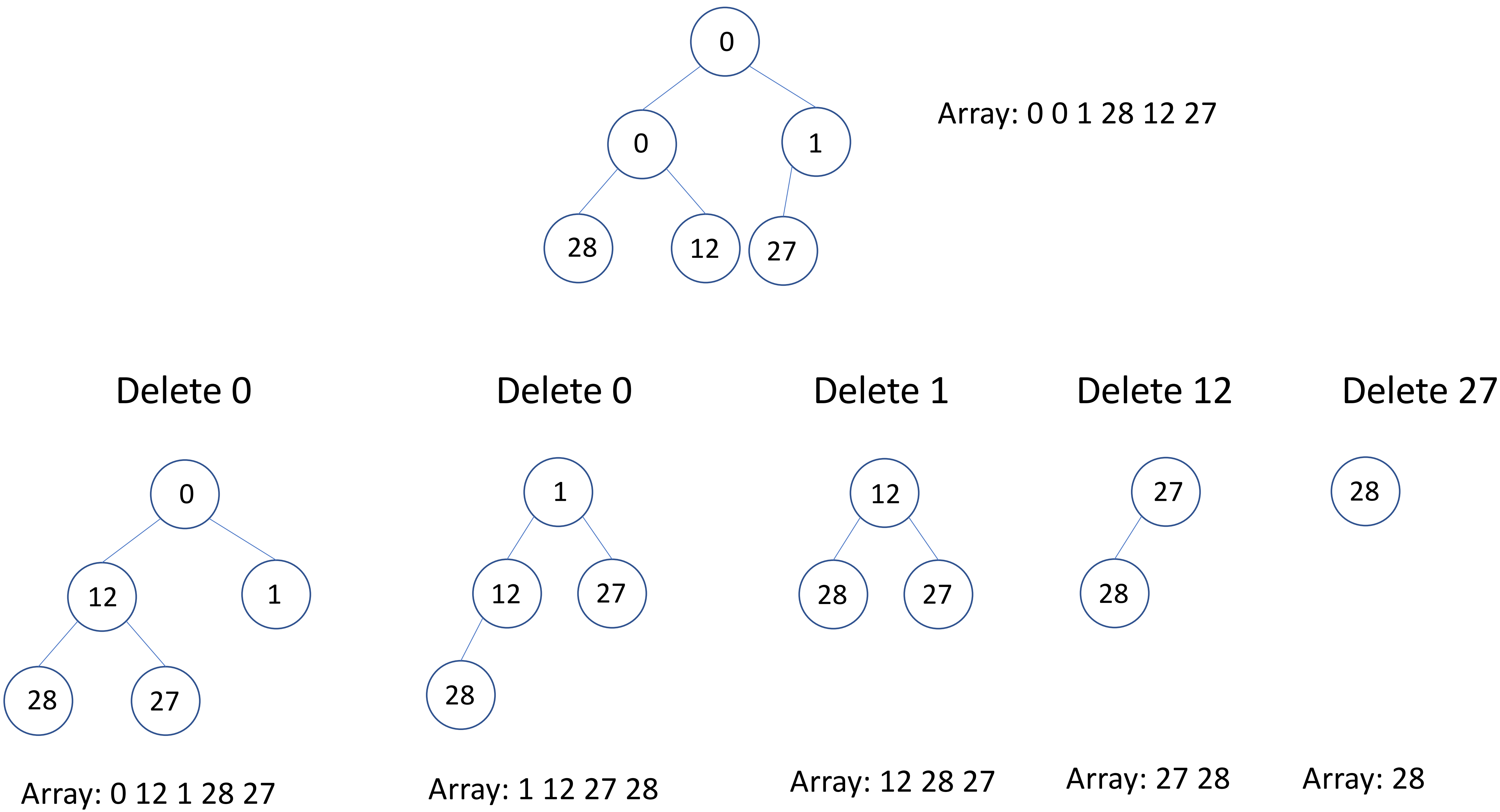



Array: 0 0 27 28 12

Insert 1




Array: 0 0 1 28 12 27


0
0
1
28
12
27
Array: 0 0 1 28 12 27
Delete 0
Delete 0
Delete 1
Delete 12
Delete 27
0
12
1
28
27
Array: 0 12 1 28 27
1
12
27
28
Array: 1 12 27 28
12
28
27
Array: 12 28 27
27
28
Array: 27 28
28
Array: 28