CSC2001F: Data Structures II

Omowunmi Isafiade

Email: omowunmi.isafiade@uct.ac.za

Office: Room 306

Outline

- How Priority Queues are different from hash tables
- Application Areas
- Implementation Approaches – A comparison
- The binary heap
  - Properties
  - Insertions and Deletions

Priority Queue – what it is

- Queues are a standard technique for ordering tasks on a first-come, first-served basis (FIFO principle). (**note**: different from priority queue)

- **Priority Queue:** a data structure that stores tasks based on some priority and supports access and deletion of the minimum or maximum item (priority metric)

- Good for applications or situations in which some partial ordering is required and where access to the maximum or minimum item needs to be done quickly (0 – **high**, 1, 2, … **low**)

Priority Queues versus Hash Tables

- Hash tables **enable fast access** to objects **but do not** provide efficient access to the **minimum or maximum item**.

- **Example:** sorted array implementation makes access to minimum element easier than in hash table

0	22
1	43
2	18
3	
4	

Hash Table

Priority Queue implementation as a sorted array

Priority Queues – Specification

- Priority queues support the operation on a set **S**:

  - **Insert (S,x)**: inserts x into the set S
  - **Max (S)**: returns the maximum element in S
  - **Extract-Max (S)**: removes and returns the element of S with the largest key
  - **Increase-key (S, x, k)**: increases the value of x's key to the new value k (k is assumed to be as large as x)

  - **Note:** Same operations possible with **min**

Priority Queues - Applications

- Priority criterion: typically decided by the application or scenario for which the application is designed.

- **Example 1: Operating Systems** (Job Scheduling)
  - Priority queue holds <u>jobs to be performed and their priority values</u>, as the jobs arrive
  - When a job is completed or interrupted, <u>highest priority job</u> is chosen
  - Scheduler ensures the highest priority job is at the head of a queue (new jobs can be added)
  - **AIM**: Avoid deadlocks or delays

Priority Queues - Applications

- **Example 2: Bandwidth management** (Networking)

  - Give high priority to certain types of network traffic
  - data (high priority) over voice (lower priority)
  - email over chat or social networking traffic
  - **AIM**: Optimise bandwidth usage

Priority Queues - Applications

- **Example 3: Discrete Event Simulation**
  - Software for experimentation
  - E.g automobile traffic on busy routes (traffic light)
  - Traffic light having an event scheduled for its next change
  - When a light changes to red, its next change to green is scheduled within some time interval (Green → Yellow → Red).
  - **AIM**: Effective Traffic Control/ Cost Effective

Priority Queues - Motivation for study

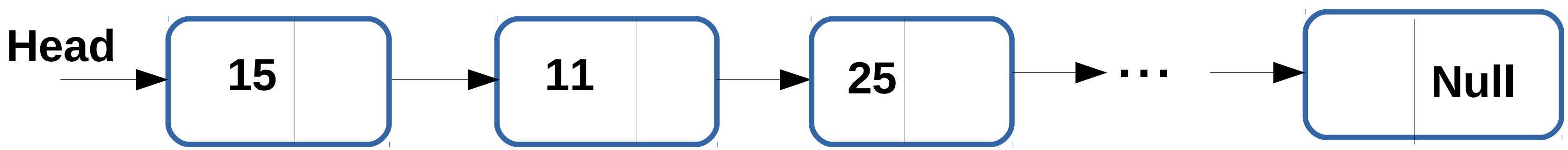
- **Prioritization** – Adhering to real-time constraints
  - **Ordering operations** to minimize execution delays
  - Need for data structures that are computationally efficient
  - **Goal**: performance goal is for operations to be **“fast”**

Priority Queues - Implementation Strategy

- Things to keep in mind...
  - What are some of the approaches to consider?
  - What is the efficiency of using one approach over another?
  - Can consider any merits and/or demerits of the choice(s) made

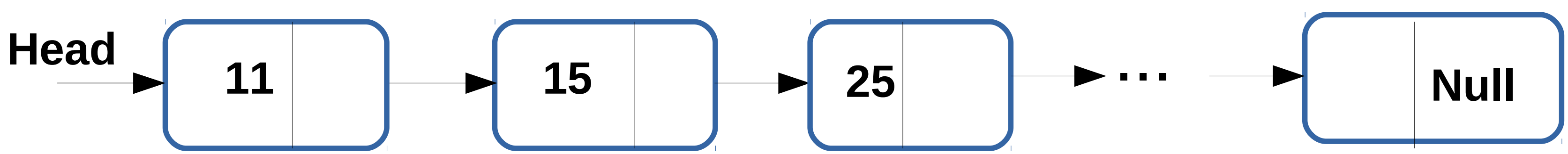
Priority Queues - Implementation Strategy

- **Recall #1**: Priority queue supports access and deletion of max/min item (fast access)
- **Option #1**: Use a simple linked list (**unordered**)
  - insertions at front in constant time [O(1)]
  - Searching and/or deleting the min [O(n)]



Priority Queues - Implementation Strategy

- **Problem**:
  - Finding and/or deletions of max/min requires a linear scan of the list
- **Option #2 (Solution):**
  - Ensure list is always sorted (**ordered linked list**)
  - Makes for cheap access (find) and deletions [O(1)]
  - But!!! Insertions still require scanning the list (linear scan)



Priority Queues - Implementation Strategy

- **Option #3**: Use a binary search tree (BST)
  - Gives an O(log N) average running time (find/delete and insert)
  - Better than scanning through a linked list:
    - Where average running time is O(N)

Priority Queues - Implementation Strategy

- **Problem**:
  - Input typically not sufficiently random
  - Can lead to a linked list (O(N) running time: find/delete and insert)
- **Option #4 (Solution):**
  - Could use **balanced search trees** but implementation is cumbersome and performance in practice is not good

Priority Queues - Implementation Strategy

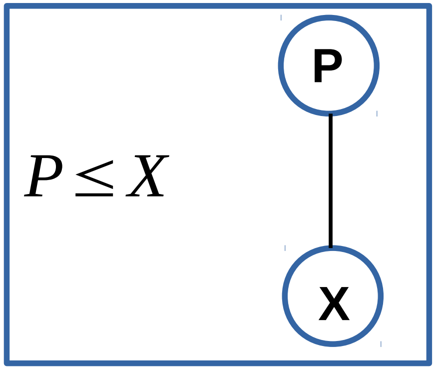
- **Other possibilities**:
  - Use a sorting algorithm to order a sequence of elements
  - Then implement a stack/queue to handle access
- Binary Heap **(Option #5 – our focus)**
  - A priority queue data structure
  - Compromise between a queue and a search tree
  - Classic approach to implementing priority queue

Binary Heap – What it is

- A **binary heap** is a **binary tree** with **two properties**:
  - Structure property: if a complete binary tree is used all algorithm executions need to maintain this data structure
  - Heap order property: if max/min element is at the root this should always be true
- **Efficient:** allows insertion (new items) and deletions (minimum/maximum item) in **logarithmic worst-case time (balanced tree)**

Binary Heap – Ordering Property

- The heap-order property allows a priority queue to perform operations quickly
- So it makes sense – use to find min/max quickly
- **Heap-order property** - "in a heap, for every node *X* with parent *P*, the key in P is smaller than or equal to the key in X, ($P \leq X$)"



- **Note (conversely):** A ***max heap*** supports access to the maximum. Can be implemented with minor changes.

Binary Heap – Ordering Property

- Storage: “**min**” item at root

- Ensure each parent key (P) is less than (or equal to) keys at two other specific (children) positions

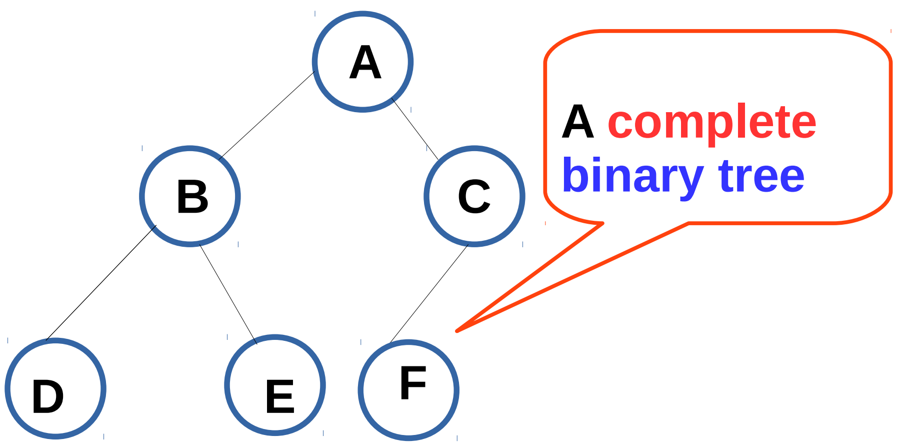
- That is: a complete binary tree with each “**key” less than or equal to** its two children (P <= X)

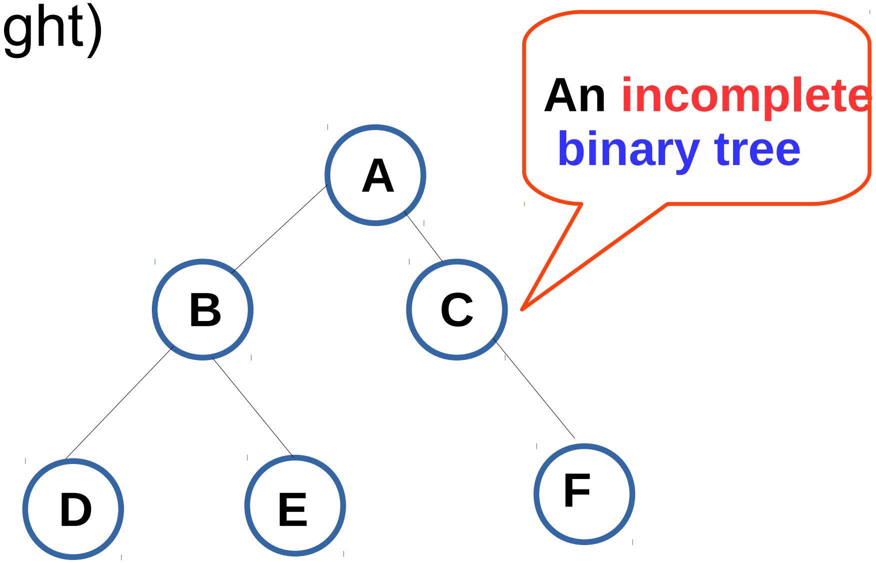
- “A binary tree is **heap ordered** if the key in each node is less than (or equal to) the keys in that node's two children (if any).”

- **Note**: Same applies to ***“max”*** with a slight modification

Binary Heap – Structure Property

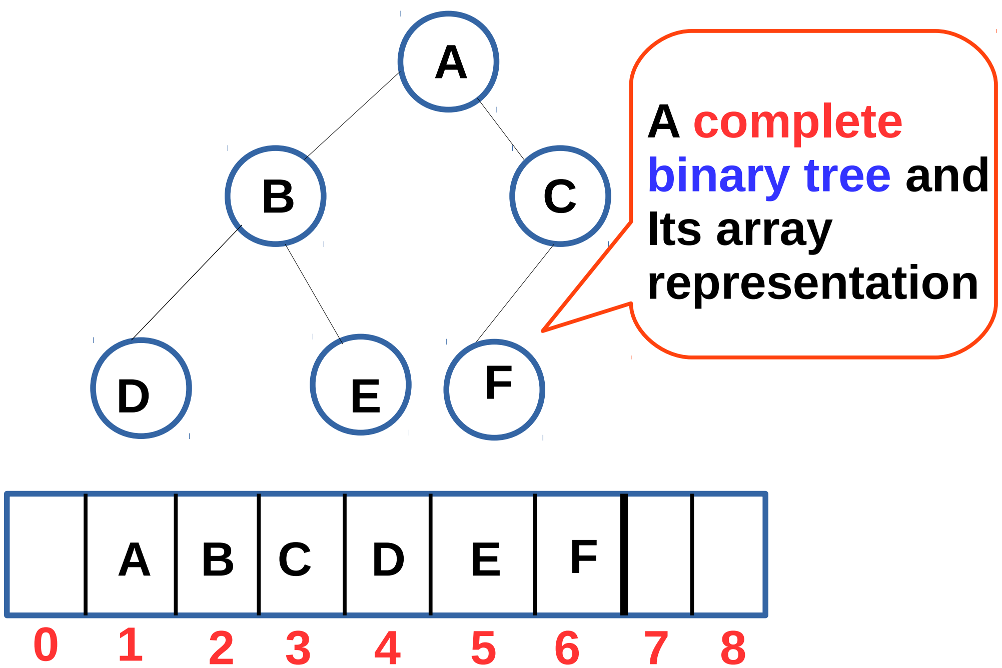
- The heap is a complete binary tree
- Dynamic logarithmic time bounds – tree structure
- **Complete Binary Tree**:
    - Each level is completely filled, with the possible exception of the bottom level (**filled from left** to right)
    - No missing nodes

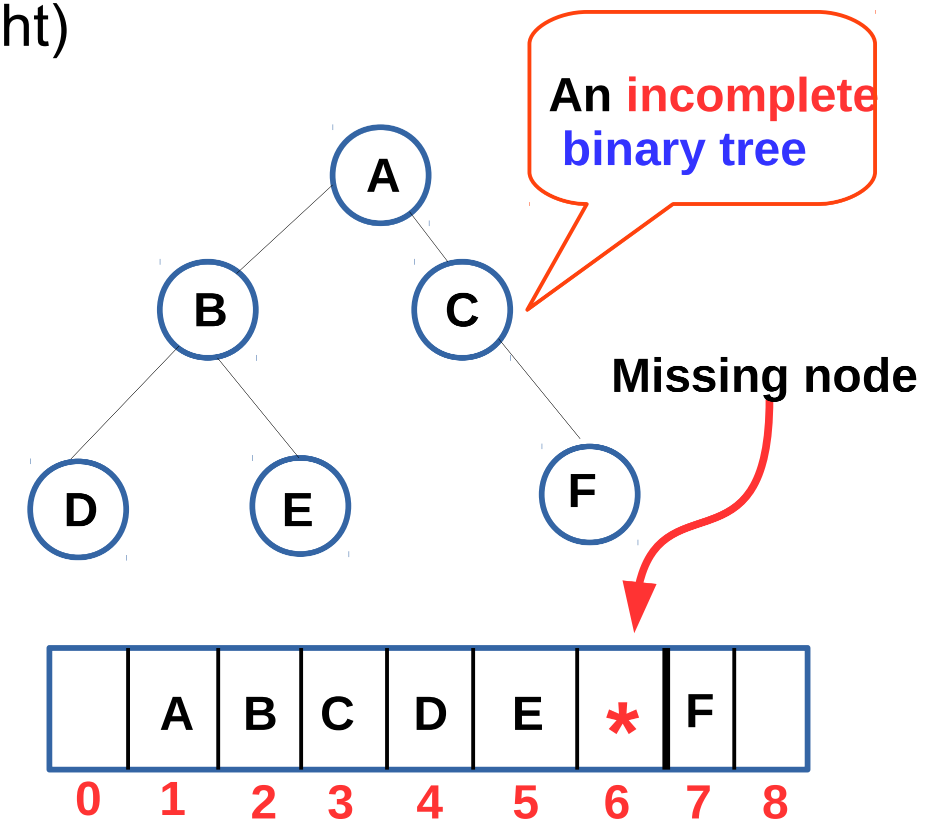




Complete Binary Tree – Structure Property

- **Note**: Root node in position 1 **(not 0- reserved for a dummy item)**
- So for an item in position i, its left child is in position **2i, right** child→2i+1
- **Complete Binary Tree**:
  - Each level is completely filled, with the possible exception of the bottom level (filled from left to right)
  - No missing nodes



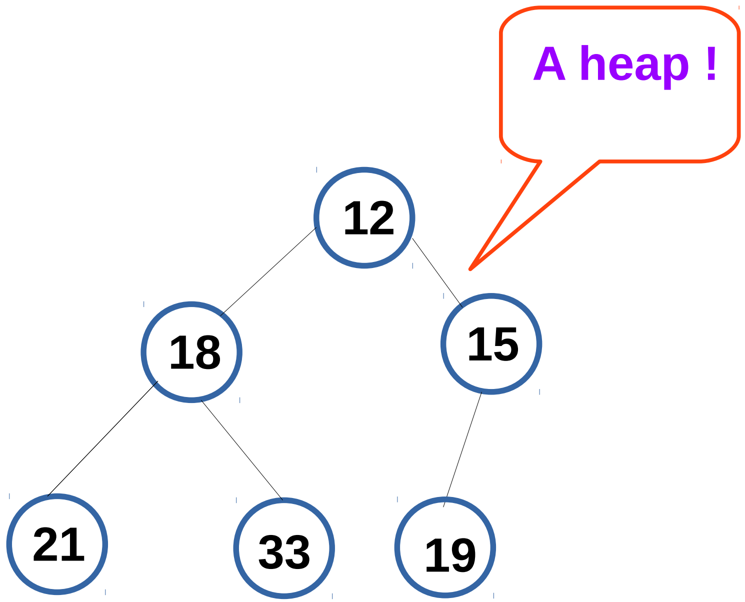


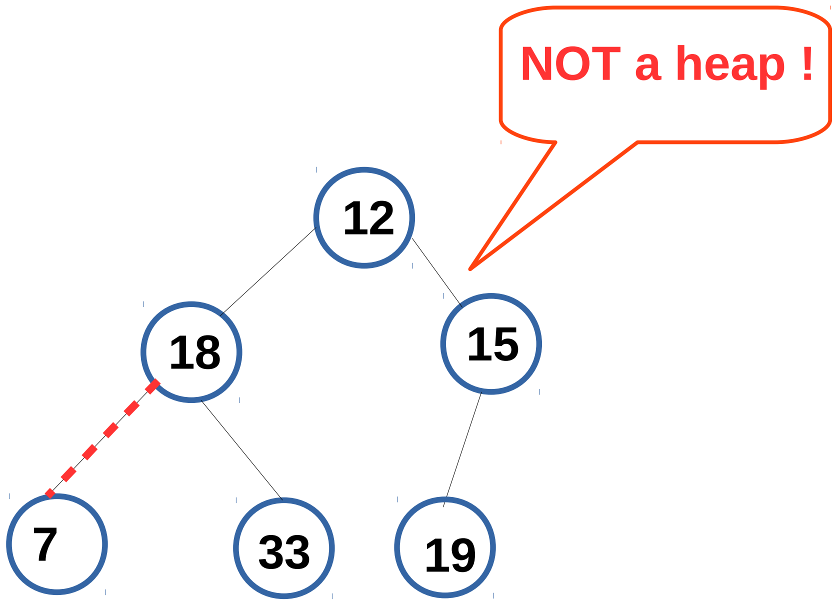
Binary Heap – Structure Property

- The heap structure allows a representation by a simple array (ensures logarithmic depth)
- **Implicit representation**: using an array to store a tree
- Both properties (structure and ordering) need to be satisfied in order to avoid violating either one (operations on a heap could violate either one)
- Binary heap operations should terminate <u>only when both properties are satisfied</u>

Binary Heap - Properties Overview

- The **heap** is a **complete binary tree** with each (parent) key **less than or equal to** its two children (note: case of “**min**” heap)



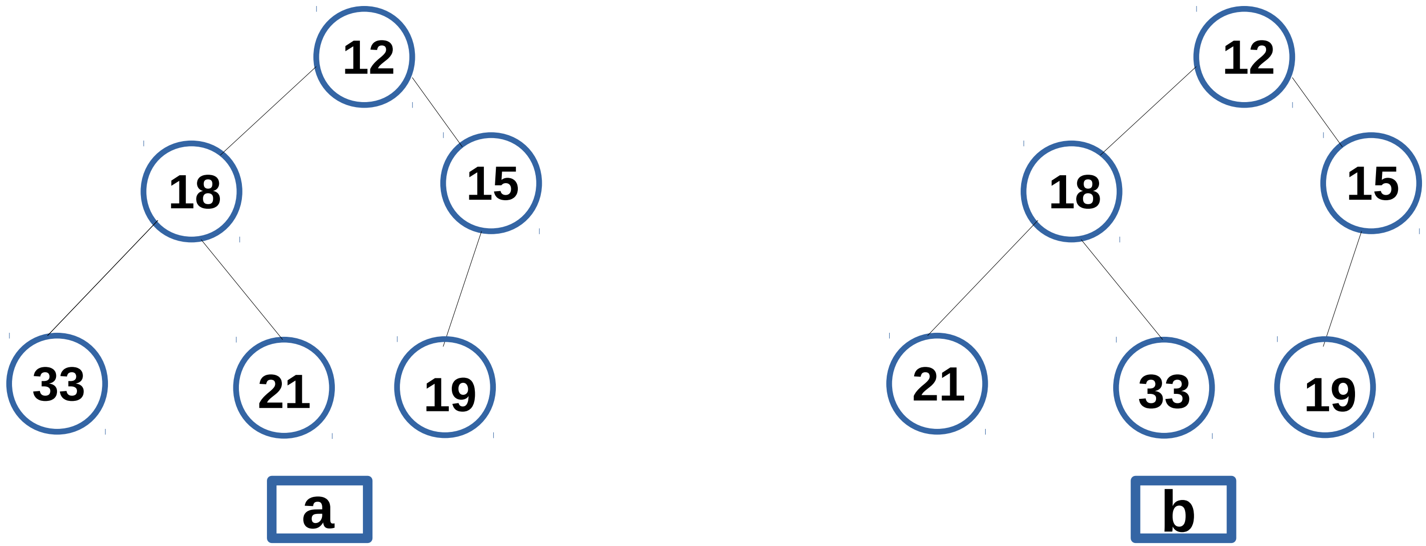


- **Note**: Both are **complete binary trees** **but** ordering property is violated in one (dashed line shows violation of heap order)

Binary Heap - Properties Overview (Exercise)

- The **heap** is a **complete binary tree** with each (parent) key **less than or equal to** its two children (note: case of "**min**" heap)

Which of the trees (a or b) below is a binary heap?

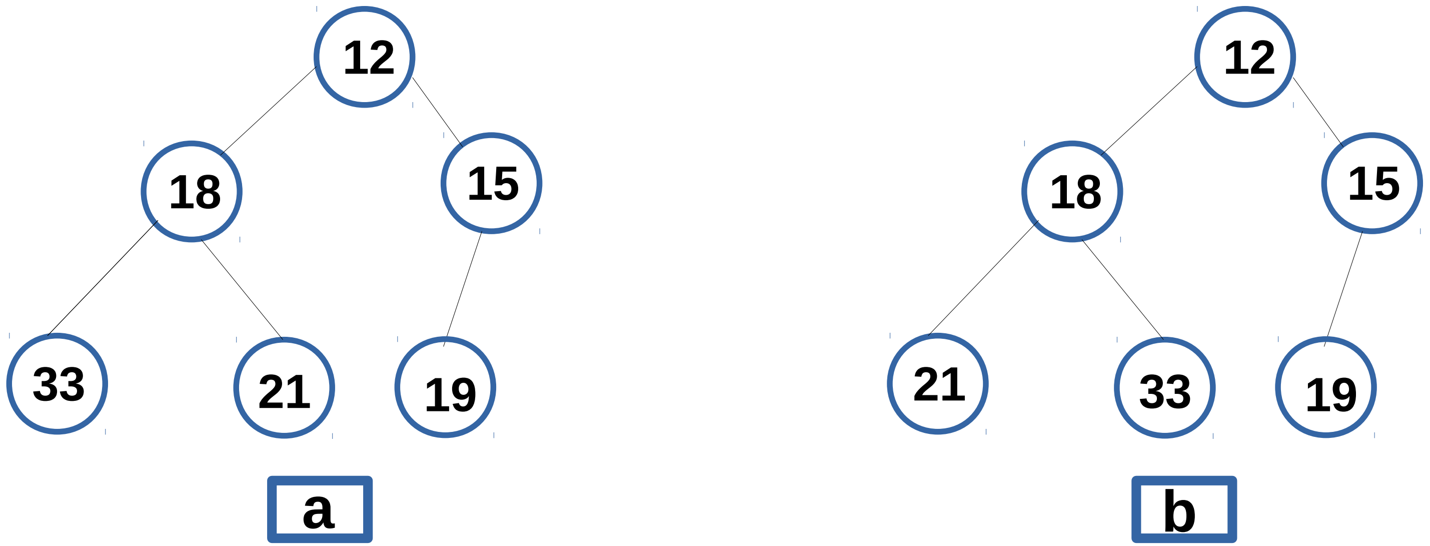


- (i) Only "a" is a heap
- (ii) Only "b" is a heap
- (iii) Both are binary heaps

Binary Heap - Properties Overview (Exercise)

- The **heap** is a **complete binary tree** with each (parent) key **less than or equal to** its two children (note: case of "**min**" heap)

Which of the trees (a or b) below is a binary heap?



- Both are binary heaps
- Since structure (complete binary tree) and ordering (<=) properties are satisfied.

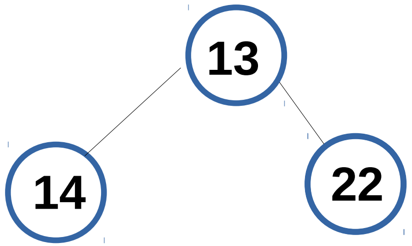
Heap Operations - Insertions

- **Note:** Structure and order properties must always be obeyed
- Methodology:
  - Create a new node in the tree in next available position (to avoid violating structure property – complete binary tree)
  - Check to ensure that ordering property is satisfied

- General Strategy ("***Percolate up"***)
  - Create a hole at the next available location
    - If heap order is not violated, place item in the hole
    - else "bubble-up" the hole toward the root

Heap Operations – Insertion (Example)

- Consider a binary heap formed from the set {14, 13, 22}



- Insert the elements {12, 11, 10, 20} into the heap above

Heap Operations - Deletions

- Deleting the minimum element ("***percolate down***")
- Methodology:
  - Find the minimum element (easy at root node)
  - Delete minimum
  - Restructure tree to form a complete binary tree

- Note: Structure and ordering properties need to be maintained (before your operation terminates)

Heap Operations - Deletions

- Restructuring Principle:
  - Re- order items to ensure structure and ordering properties are obeyed
  - Exercise: Delete the items {10, 11, 12, 13, 14} from the binary heap
    - Think about how you would re-organise the heap to obey both the structuring and ordering properties!

Binary Heaps – Some Considerations

- A priority queue is not a heap (or a binary heap)
- Priority queue is an abstract concept like a list
- A list can be implemented as a linked list or an array
- Likewise – a heap is just a (classical) method of implementing the concept of a priority queue

Binary Heaps – Some Considerations

- **Recall:** Implicit representation
  - An array can be used to store a tree (e.g instead of doing it with a linked list)

- Advantage:
  - No child links required
  - Operations required to traverse tree are simple to implement and efficient (performance)
  - Heap entity: array of objects and an integer (current heap size)

- Disadvantage:
  - Dynamic adjustments of table size can become expensive

Summary and Next Lecture

- Build Heap Operation
  - Takes a complete tree that does not have a heap order and re-instates it
  - Drops cost of handling insertions and deletions
- Implementation considerations