UNIT-III
PRIORITY QUEUES (HEAPS)

What is a Priority Queue?
1) Stores prioritized key-value pairs
2) Implements insertion
• No notion of storing at particular position
3) Returns elements in priority order
• Order determined by *key*

Stacks and Queues
• Removal order determined by order of inserting
Sequences
• User chooses exact placement when inserting and explicitly chooses removal order
Priority Queue
• Order determined by key
• Key may be part of element data or separate

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

Implementing PQ with Unsorted Sequence
Each call to insertItem(k, e) uses insertLast( ) to store in Sequence
• $O(1)$ time
Each call to extractMin( ) traverses the entire sequence to find the minimum, then removes element
• $O(n)$ time

Implementing PQ with Sorted Sequence
Each call to insertItem(k, e) traverses sorted sequence to find correct position, then does insert
• O(n) worst case
Each call to extractMin( ) does removeFirst( )
• $O(1)$ time

Implementing PQ with a BST
Each call to insertItem(k, e) does tree insert
• O(log(n)) worst case
Each call to extractMin( ) does delete( )
• $O(log(n))$ time

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,

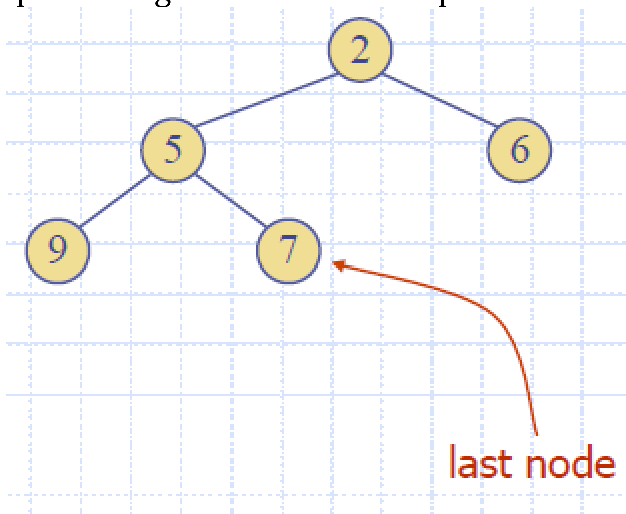
key(**v**) ≥ **key**(**parent**(**v**))

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

for **i** = 0, … , **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 depth **h**



Height of a Heap

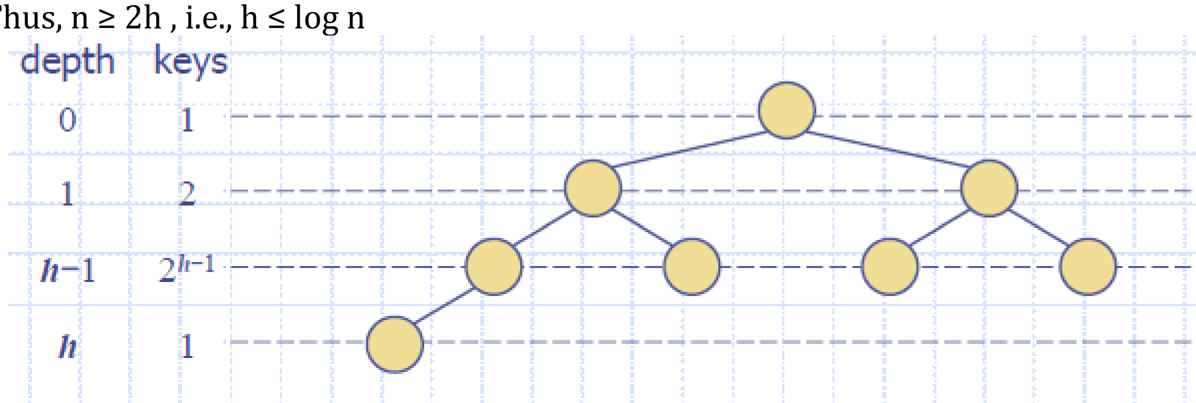
Theorem: A heap storing n keys has height O(log n)

Proof: (we apply the complete binary tree property)

Let h be the height of a heap storing n keys

Since there are 2i keys at depth i = 0, … , h - 1 and at least one key at depth h, we have n ≥ 1 + 2 + 4 + … + 2h-1 + 1

Thus, n ≥ 2h , i.e., h ≤ log n



Heap

Binary tree-based data structure

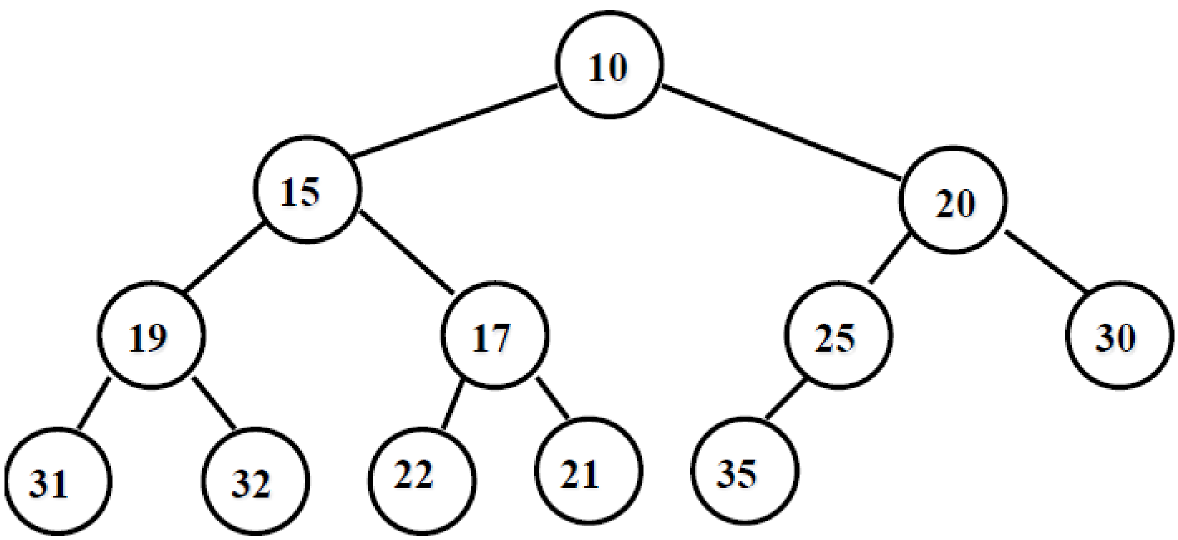
• *Complete* in the sense that it fills up levels as completely as possible

• Height of tree is $O(\log n)$

Can be stored using the array representation (just add at the end of the array)

Use extendable arrays to expand and shrink as Needed

Heap Example



<u>Binary Heaps</u>

• A binary heap is a binary tree (NOT a BST) that is:

› Complete: the tree is completely filled except possibly the bottom level, which is filled from left to right

- ➢ Satisfies the heap order property
  - • every node is less than or equal to its children or every node is greater than or equal to its children
  - • The root node is always the smallest node or the largest, depending on the heap order

<u>Heap order property</u>

• A heap provides limited ordering information

• Each *path* is sorted, but the subtrees are not sorted relative to each other

A binary heap is NOT a binary search tree

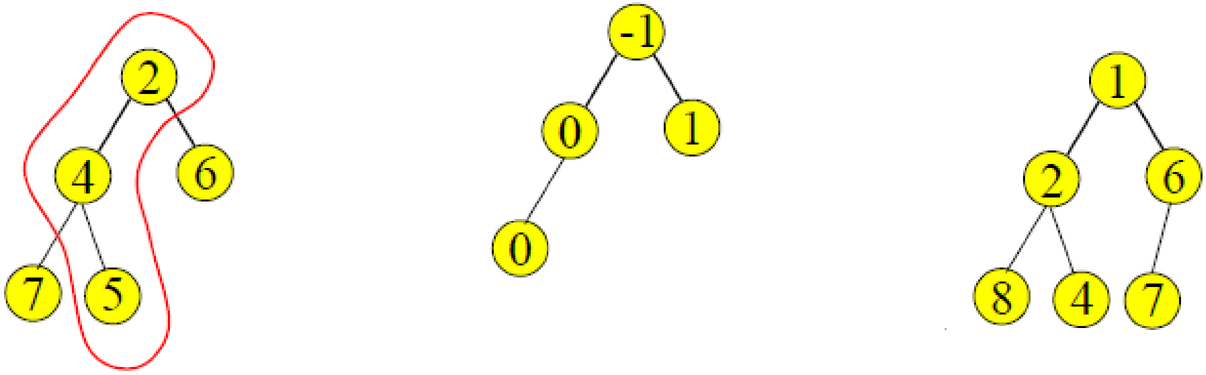
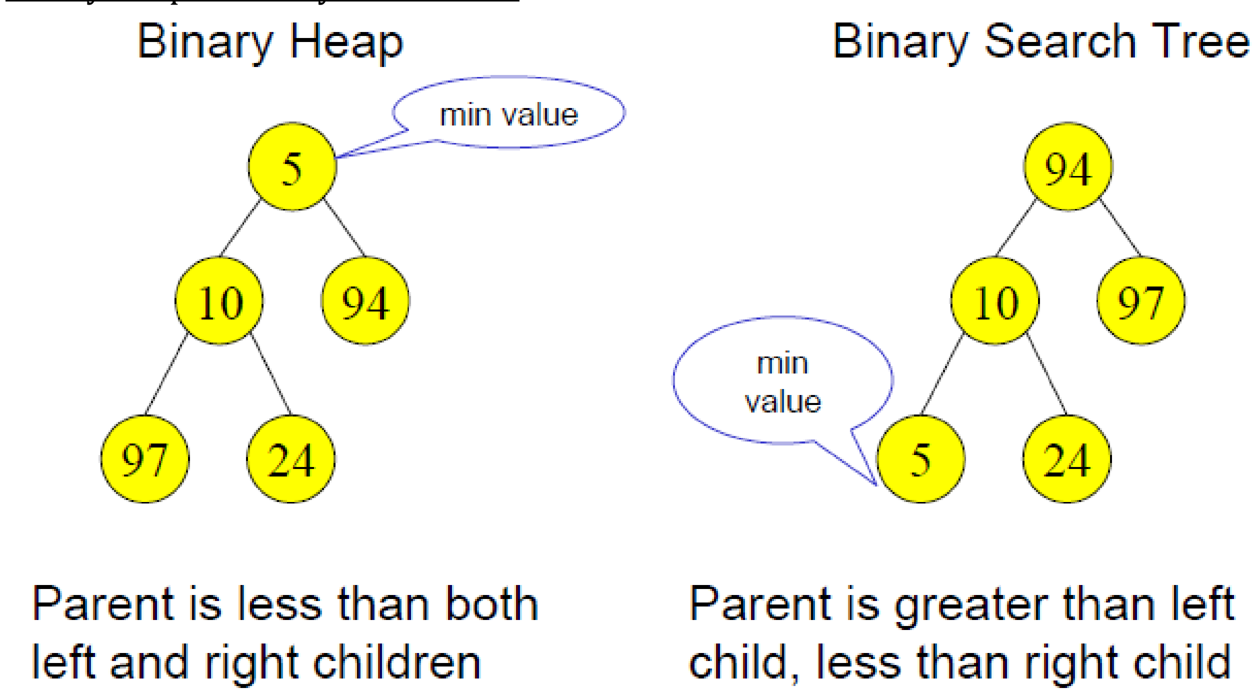


Fig: These are all valid binary heaps (minimum)

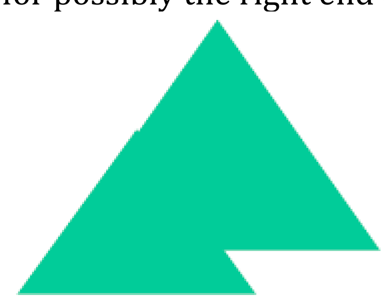
Binary Heap vs Binary Search Tree



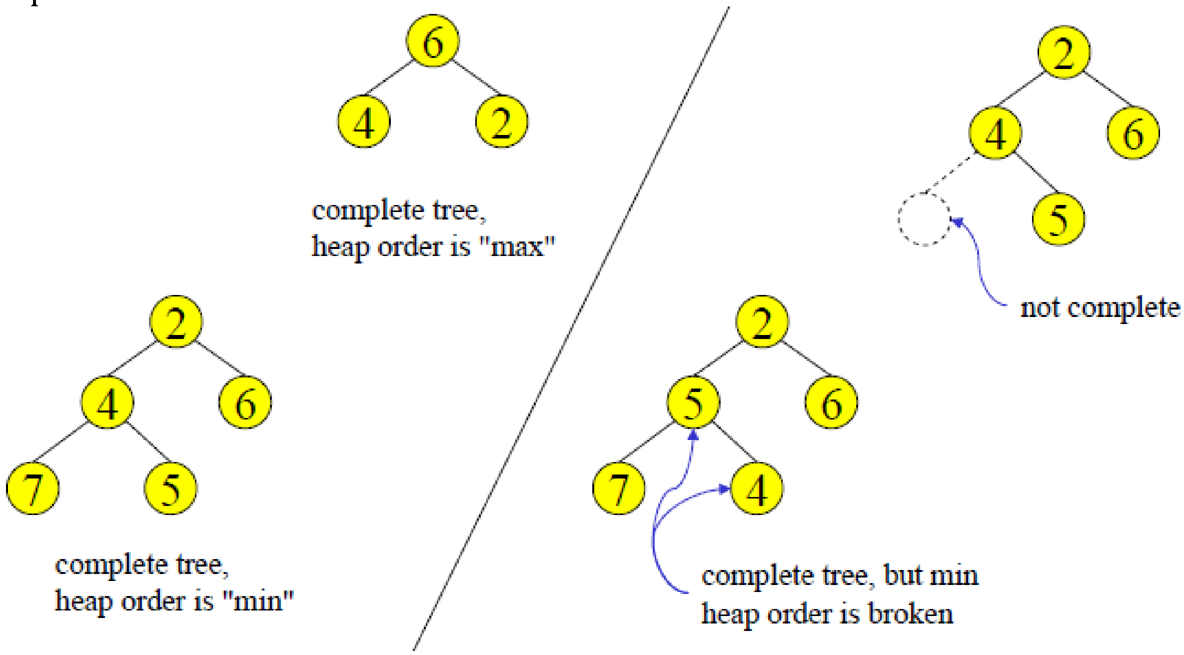
Structure property

• A binary heap is a complete tree

› All nodes are in use except for possibly the right end of the bottom row

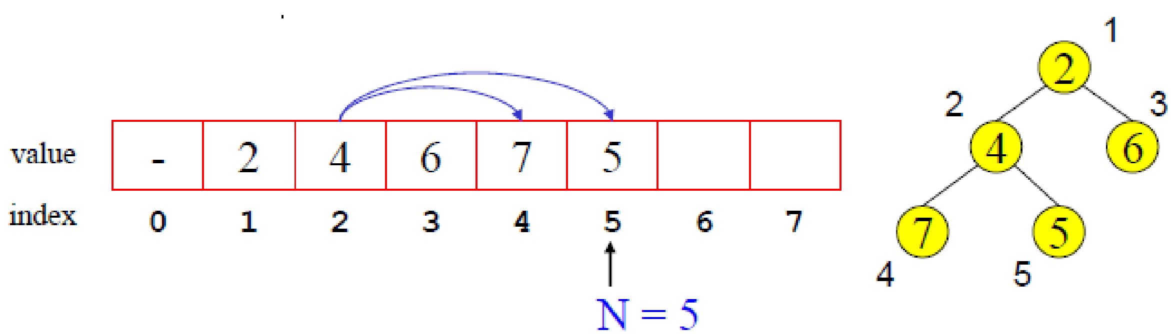


Examples:



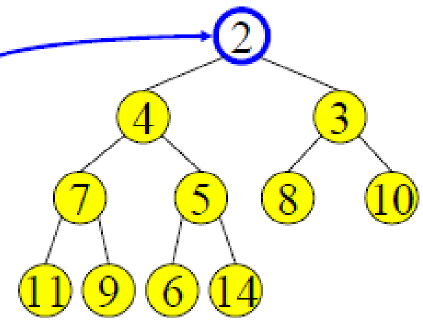
Array Implementation of Heaps

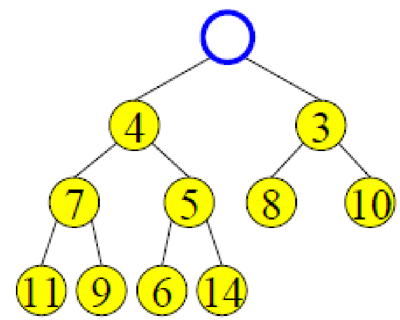
- Root node = A[1]
- Children of A[i] = A[2i], A[2i + 1]
- Keep track of current size N (number of nodes)



FindMin and DeleteMin:

- FindMin:
  - › Return root value A[1]
  - › Run time = ?

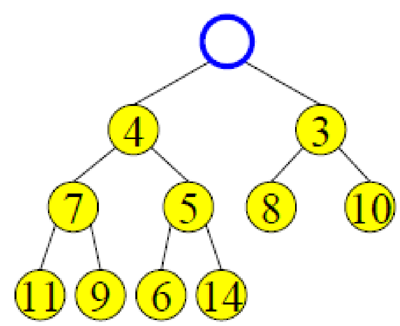


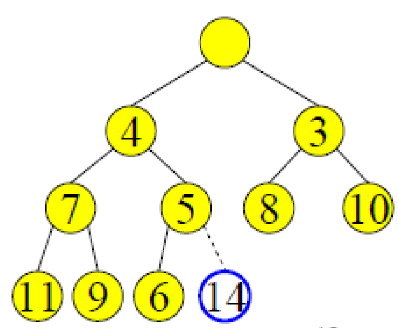


- Delete (and return) value at root node

Maintain the Structure Property

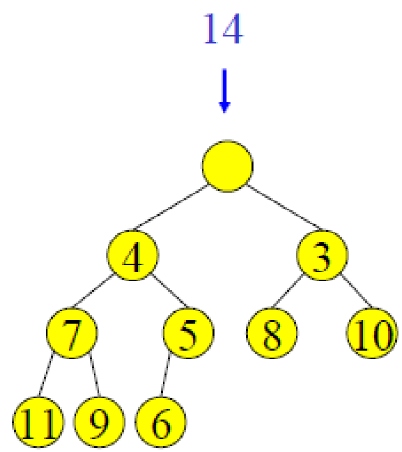
- We now have a “Hole” at the root
  - › Need to fill the hole with another value
- When we get done, the tree will have one less node and must still be complete





Maintain the Heap Property

- The last value has lost its node
  - › we need to find a new place for it
- We can do a simple insertion sort - like operation to find the correct place for it in the tree



Applications of Priority queues:

• Event-driven simulation.	[customers in a line, colliding particles]
• Numerical computation.	[reducing roundoff error]
• Data compression.	[Huffman codes]
• Graph searching.	[Dijkstra's algorithm, Prim's algorithm]
• Computational number theory.	[sum of powers]
• Artificial intelligence.	[A* search]
• Statistics.	[maintain largest M values in a sequence]
• Operating systems.	[load balancing, interrupt handling]
• Discrete optimization.	[bin packing, scheduling]
• Spam filtering.	[Bayesian spam filter]

The Selection Problem Event Simulation Problem:

Change state only when something happens.

- Between collisions, particles move in straight-line trajectories.
- Focus only on times when collisions occur.
- Maintain priority queue of collision events, prioritized by time.
- Remove the minimum = get next collision.

Collision prediction. Given position, velocity, and radius of a particle, when will it collide next with a wall or another particle?

Collision resolution. If collision occurs, update colliding particle(s) according to laws of elastic collisions.

Note: Same approach works for a broad variety of systems

Binomial Queues

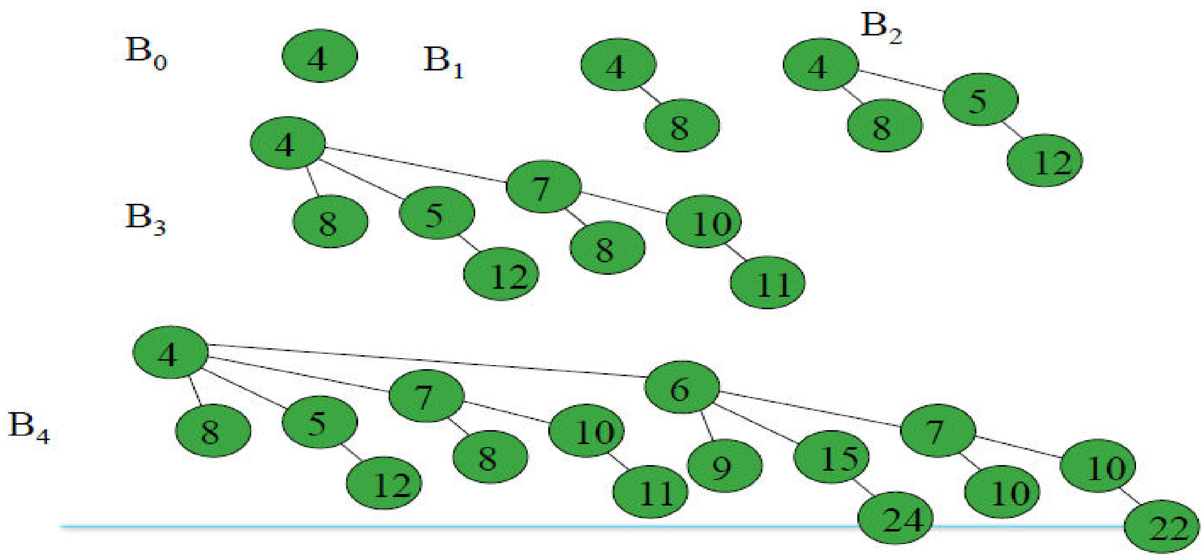
A Binomial Queue is a collection of heap-ordered trees known as a forest. Each tree is a binomial tree.

A recursive definition is:

1. A binomial tree of height 0 is a one-node tree.

2. A binomial tree, Bk, of height k is formed by attaching a binomial tree Bk−1 to the root of another binomial tree Bk−1 .

Examples



Implementing Binomial Queues

1. Use a *k-ary* tree to represent each binomial tree – sibling and child pointers

2. Use a Vector to hold references to the root node of each binomial tree

3. Keep a reference to smallest root for past find min (e.g. a Heap on positions).

Use a k-ary tree to represent each binomial tree.

Use an array to hold references to root nodes of each binomial tree.