Week 10

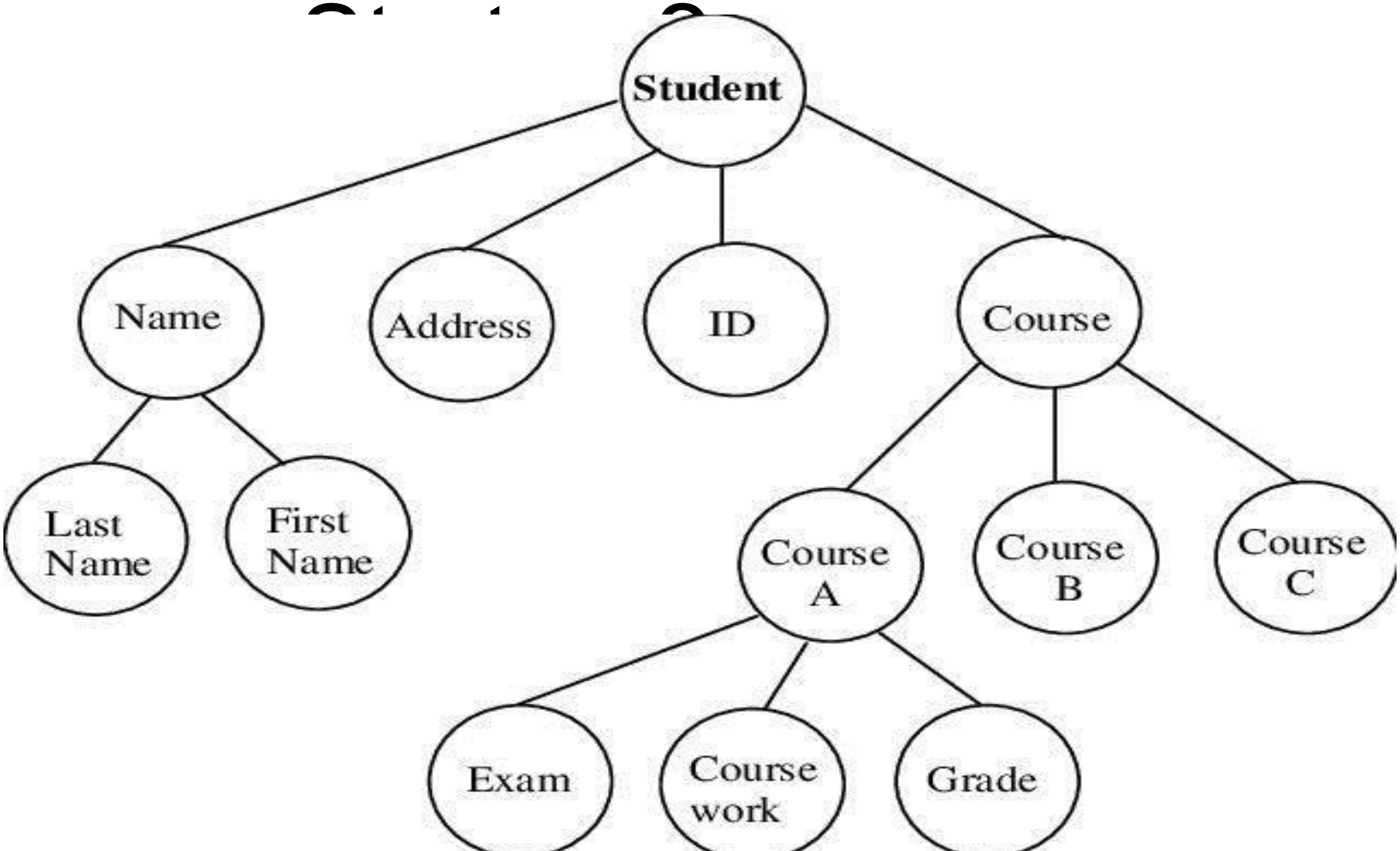
Intro to Trees and Heap

Outline

In this topic, we will cover:

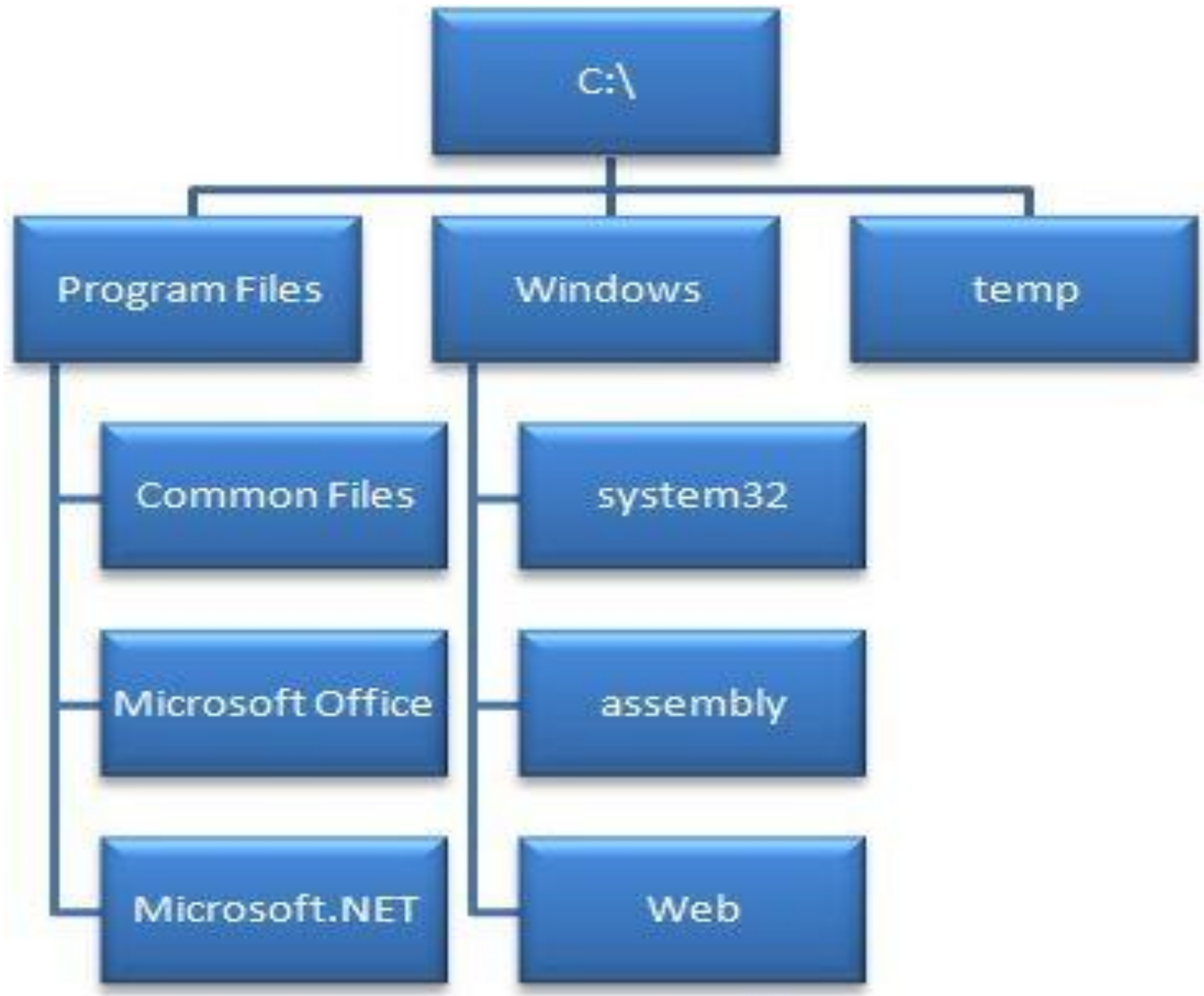
- Definition of a tree data structure and its components
- Concepts of:
  - Root, internal, and leaf nodes
  - Parents, children, and siblings
  - Paths, path length, height, and depth
  - Ancestors and descendants
  - Ordered and unordered trees
  - Subtrees
- Examples

Why Trees

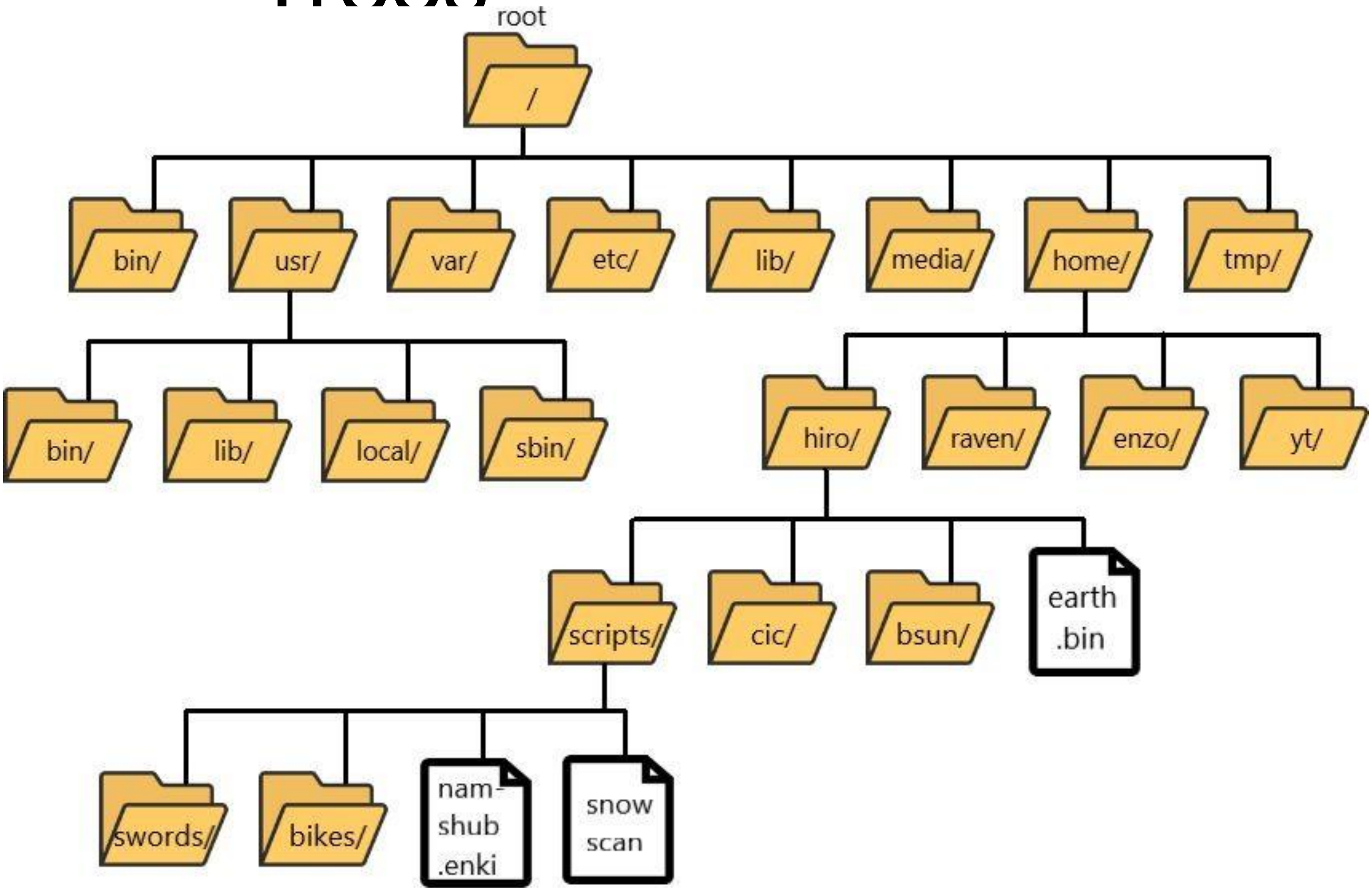


Student
Name
Address
ID
Course
Last Name
First Name
Course A
Course B
Course C
Exam
Course work
Grade

Why Trees

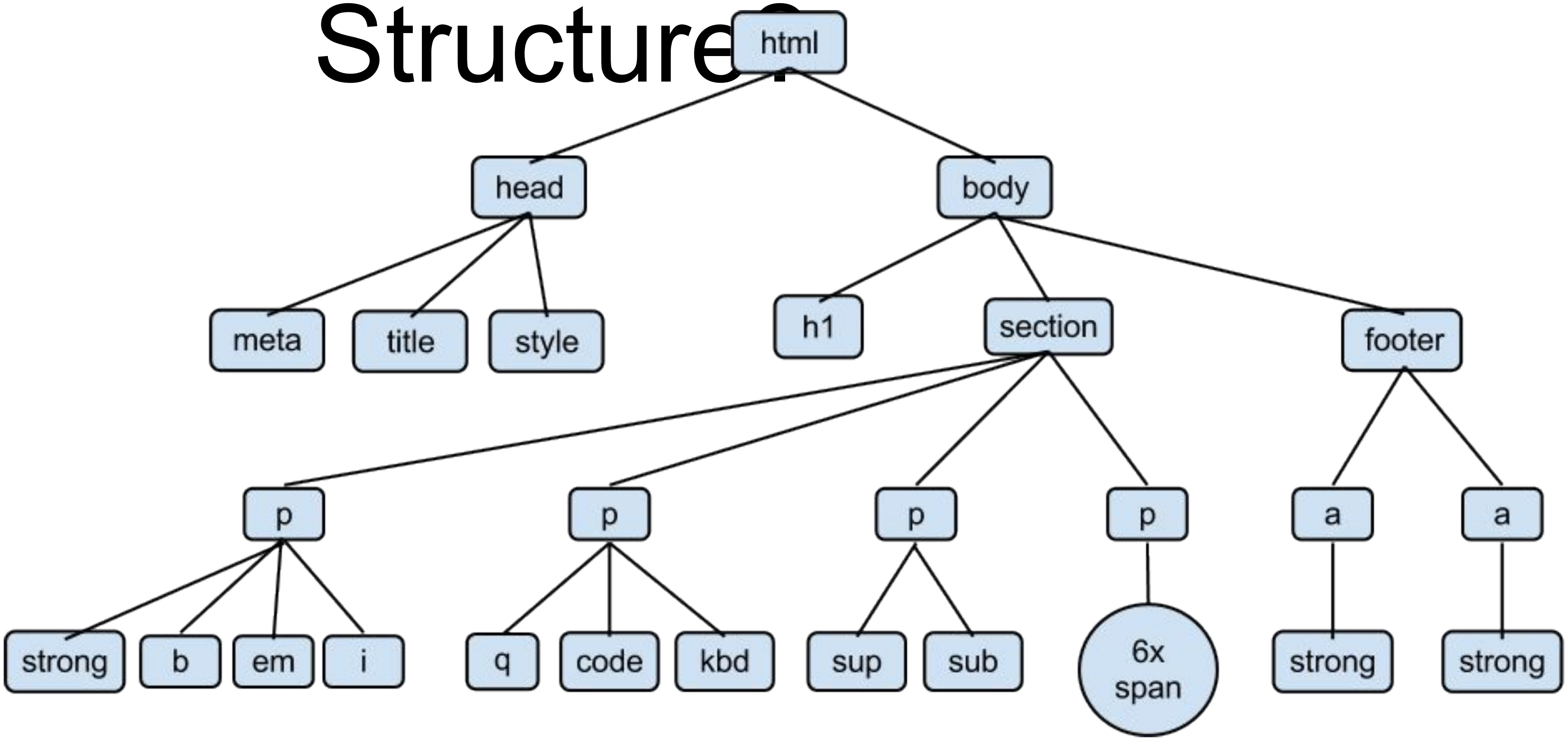

C:\
Program Files
Windows
temp
Common Files
Microsoft Office
Microsoft.NET
system32
assembly
Web

Why Trees

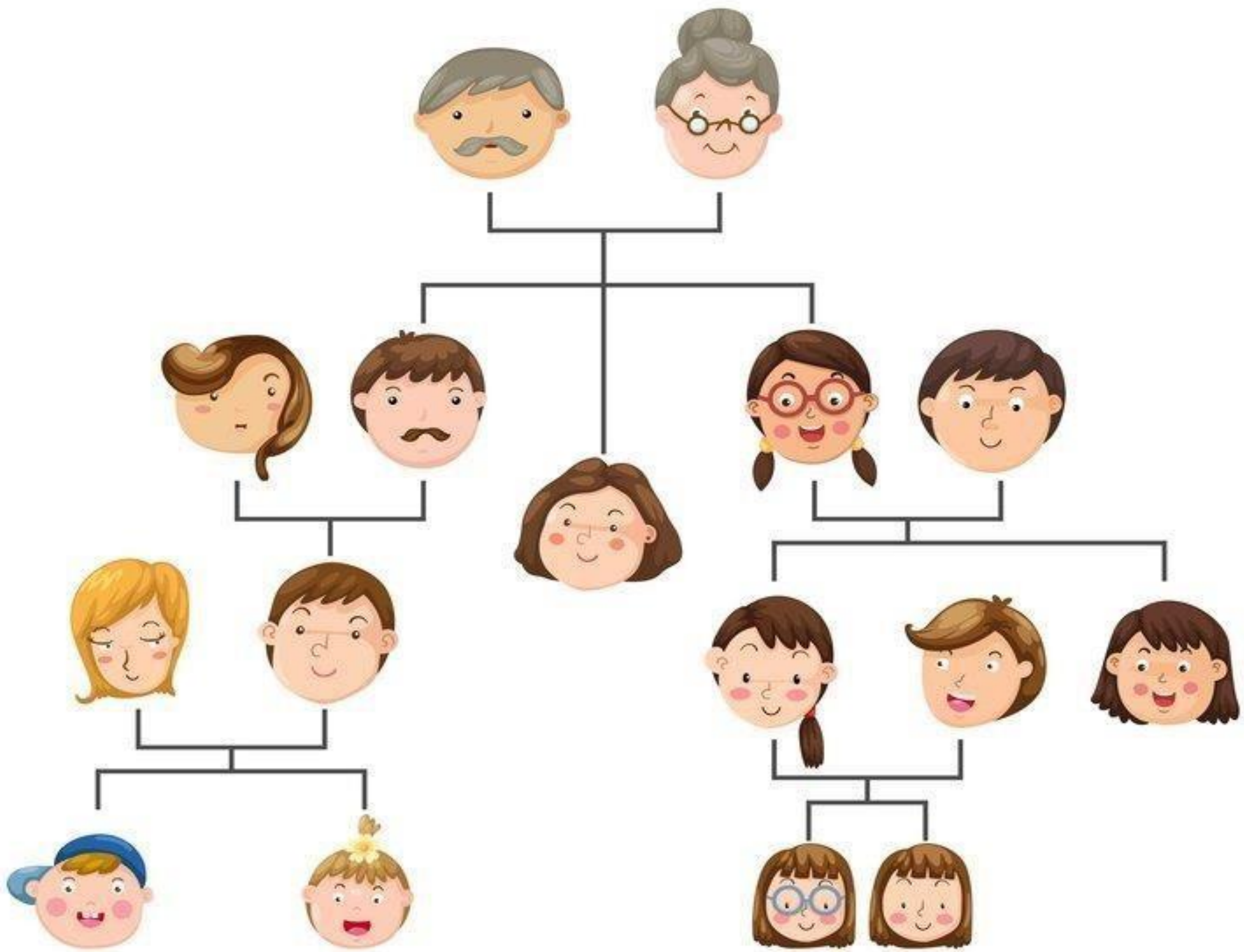


root
/
bin/
usr/
var/
etc/
lib/
media/
home/
tmp/
bin/
lib/
local/
sbin/
hiro/
raven/
enzo/
yt/
scripts/
cic/
bsun/
earth
.bin
swords/
bikes/
nam-
shub
.enki
snow
scan

Why Trees Structure?



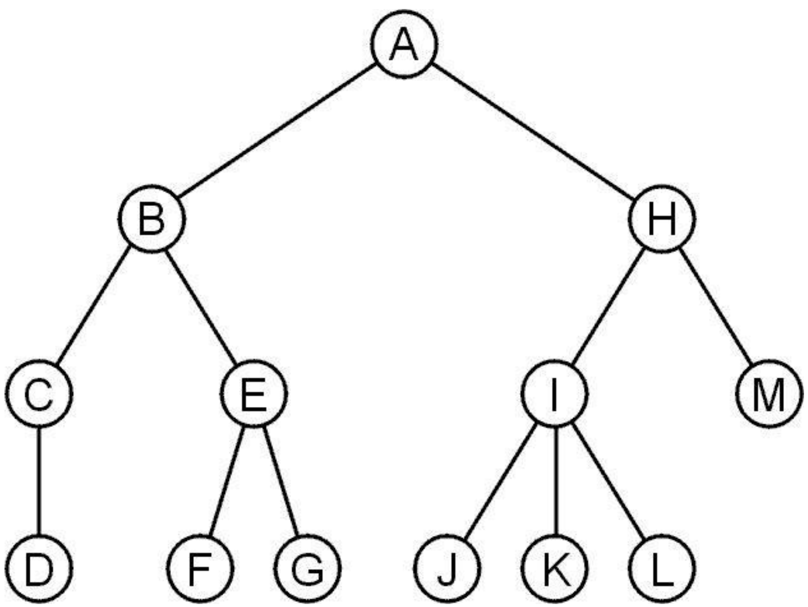
Why Trees



Trees

A rooted tree data structure stores information in *nodes*

- Similar to linked lists:
    - There is a first node, or *root*
    - Each node has variable number of references to successors
    - Each node, other than the root, has exactly one node pointing to it



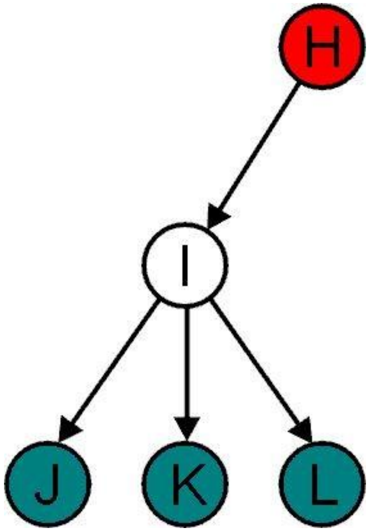
Terminology

All nodes will have zero or more child nodes or *children*

– I has three children: J, K and L

For all nodes other than the root node, there is one parent node

– H is the parent I



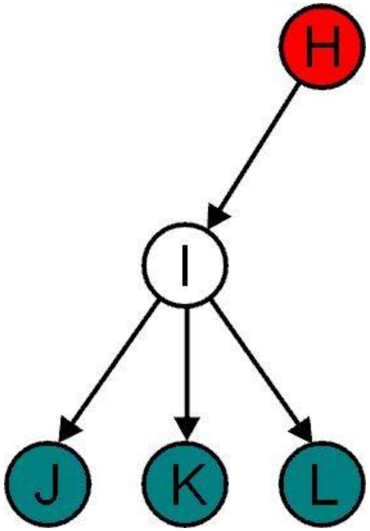
Terminolog y

The *degree* of a node is defined as the number of its children:

$\deg(\mathrm{I}) = 3$

Nodes with the same parent are *siblings*

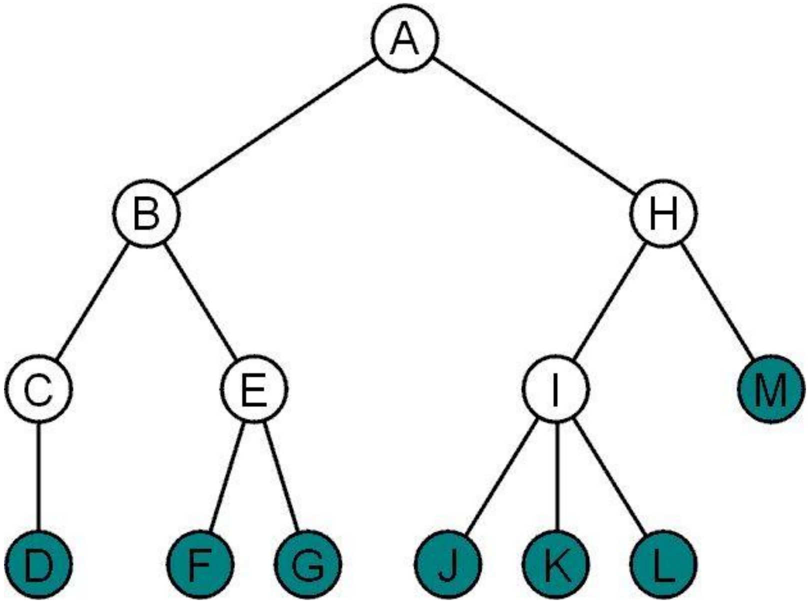
- J, K, and L are siblings



Terminology

Nodes with degree zero are also called *leaf nodes*

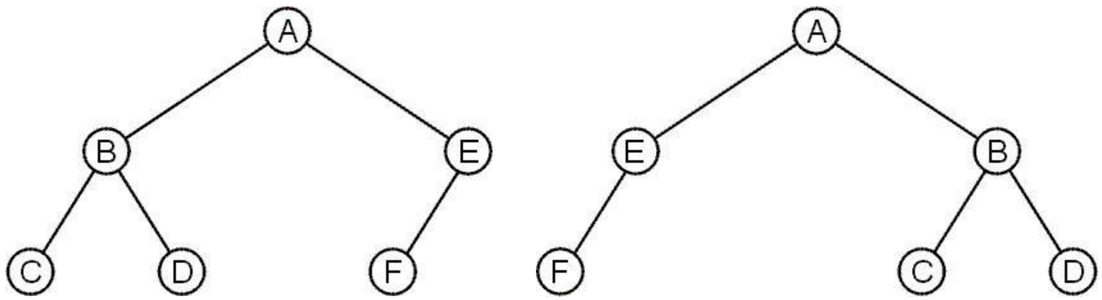
All other nodes are said to be *internal nodes*, that is, they are internal to the tree



Terminology

These trees are equal if the order of the children is ignored

– *unordered trees*



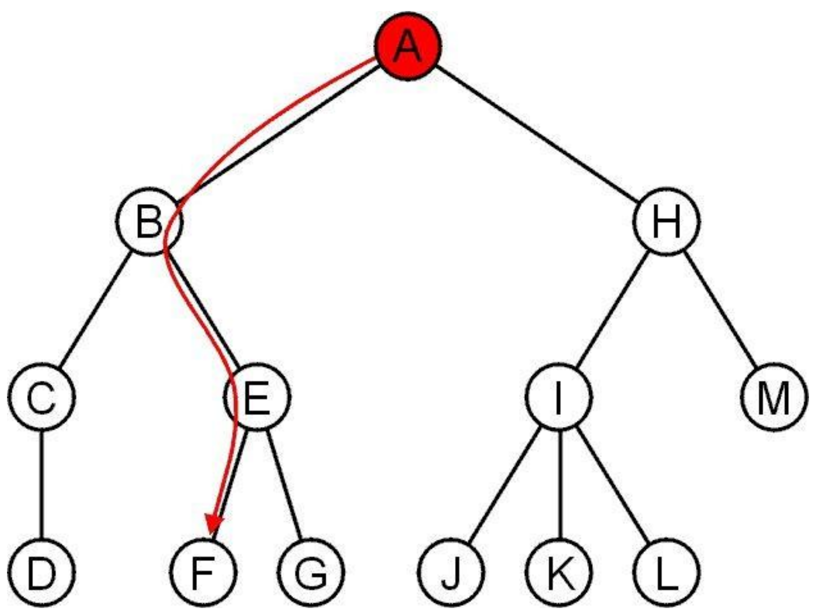
They are different if order is relevant (*ordered trees*)

– We will usually examine ordered trees (linear orders)

– In a hierarchical ordering, order is not relevant

Terminolog
y

The shape of a rooted tree gives a natural flow from the *root node*, or just *root*



Terminology

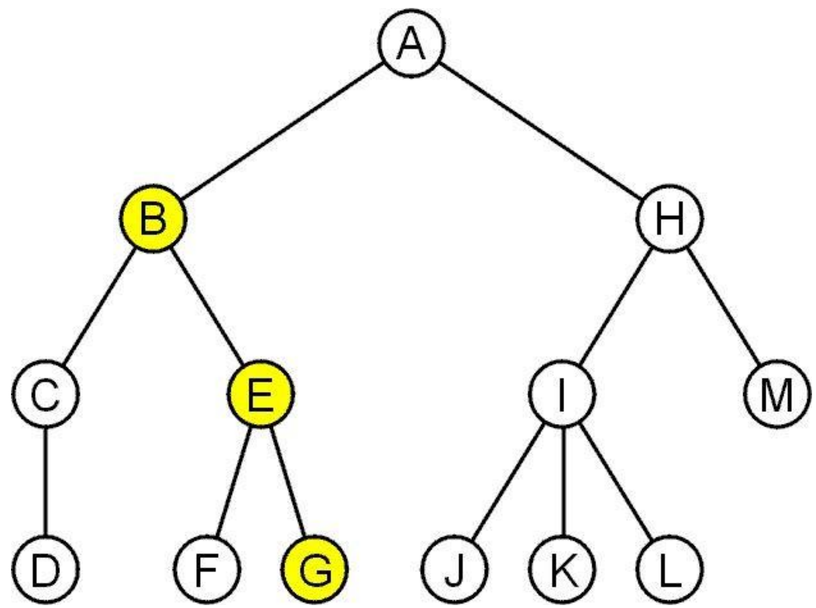
A path is a sequence of nodes

$$(a_0, a_1, ..., a_n)$$

where a_{k+1} is a child of a_k is

The length of this path is n

E.*g*., the path (B, E, G) has length 2

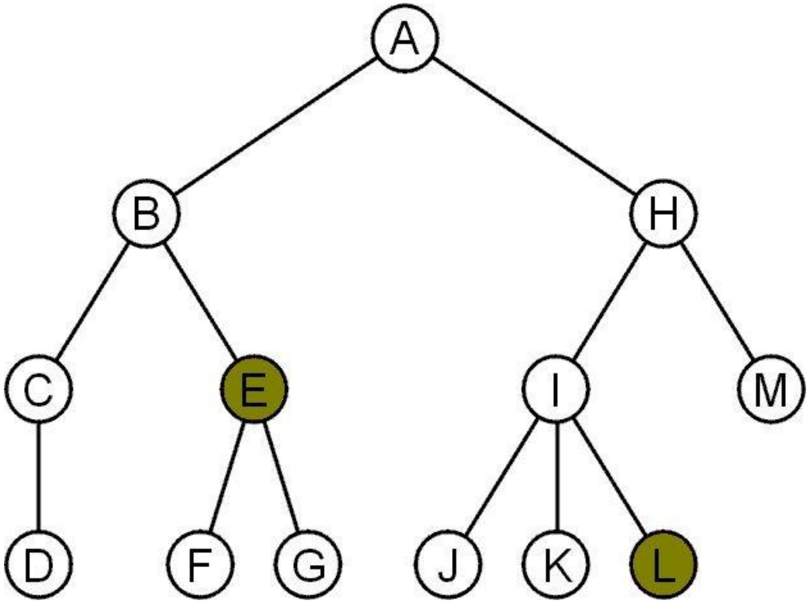


Terminology

For each node in a tree, there exists a unique path from the root node to that node

The length of this path is the *depth* of the node, *e.g.*,

- E has depth 2
- L has depth 3



Terminology

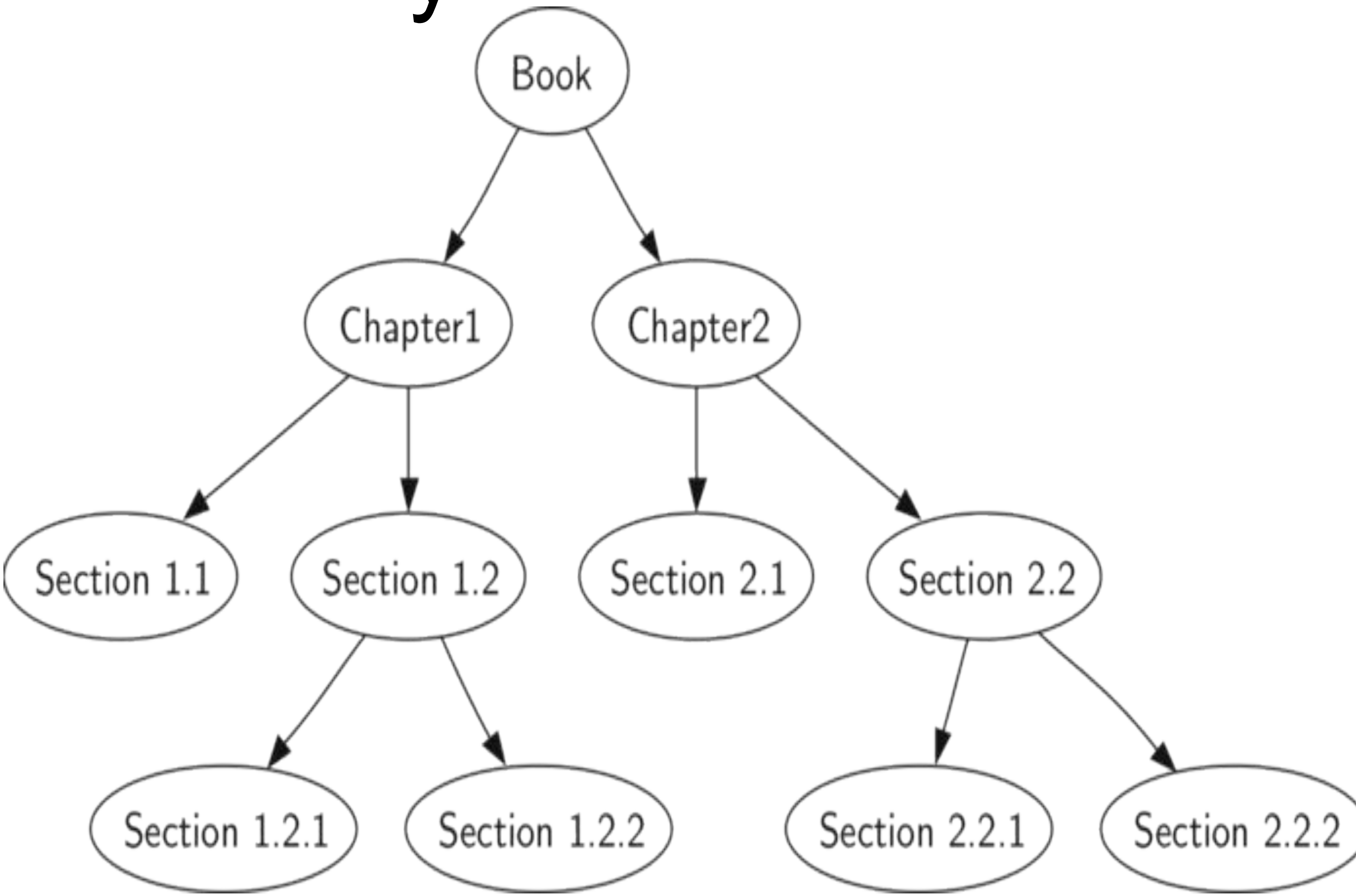
The *height* of a tree is defined as the maximum depth of any node within the tree

The height of a tree with one node is 0

- Just the root node

For convenience, we define the height of the empty tree to be -1

Terminolog
y



Terminolog
y

If a path exists from node a to node b:

– a is an *ancestor* of b

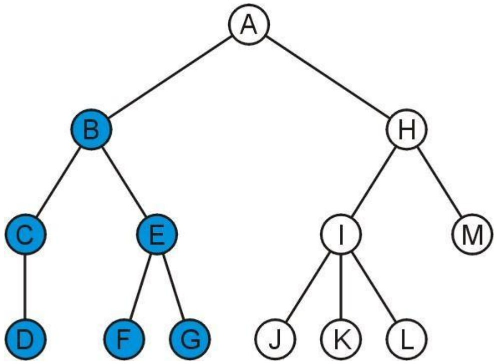
– b is a *descendent* of a

Thus, a node is both an ancestor and a descendant of itself

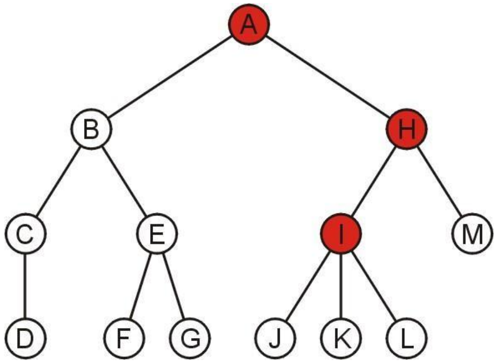
The root node is an ancestor of all nodes

Terminology

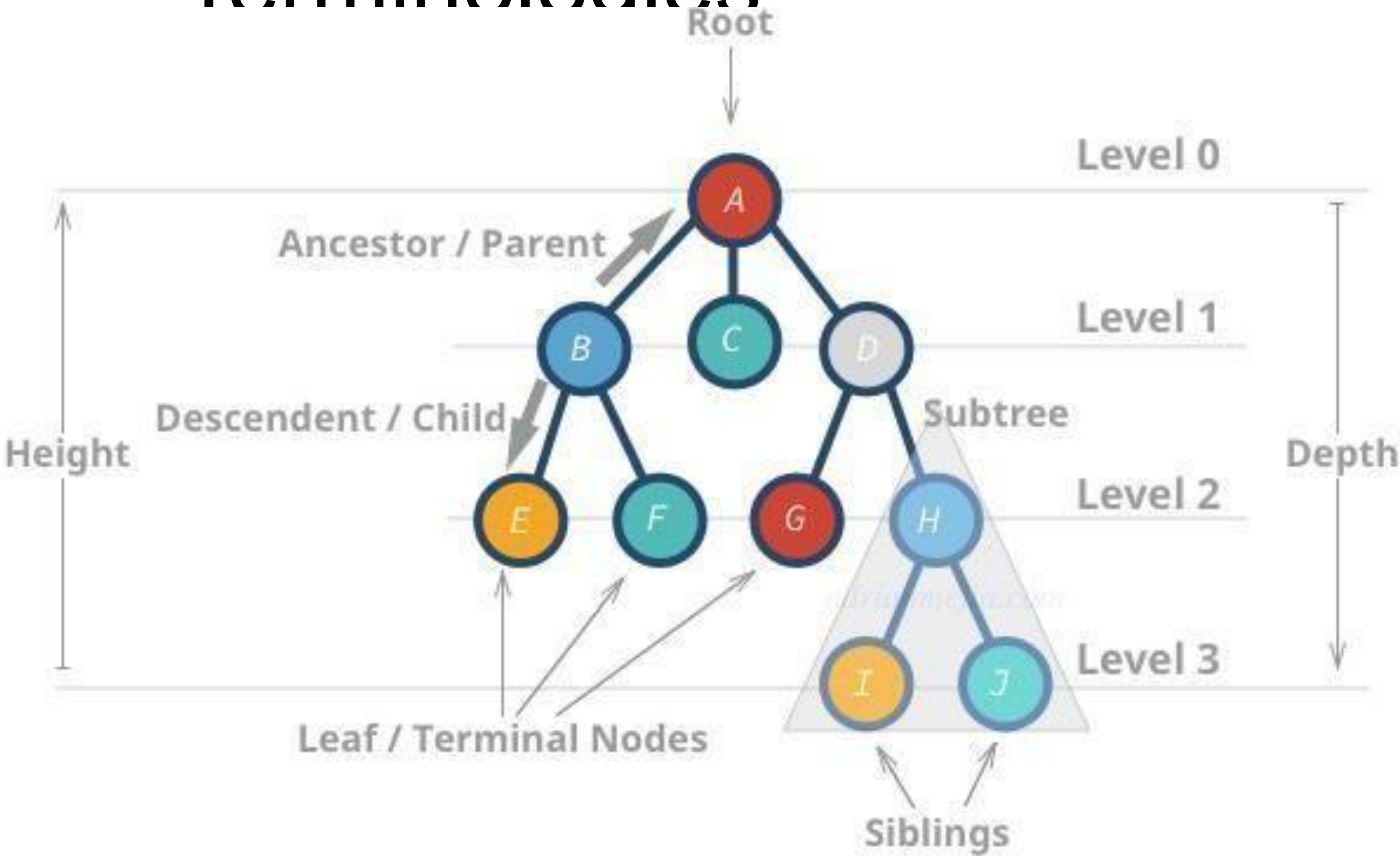
The descendants of node B are B, C, D, E, F, and G:



The ancestors of node I are I, H, and A:



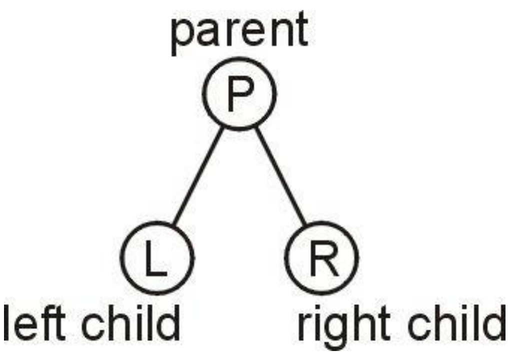
Summarized Terminologies



A Binary Tree

A binary tree is a restriction where each node has exactly two children:

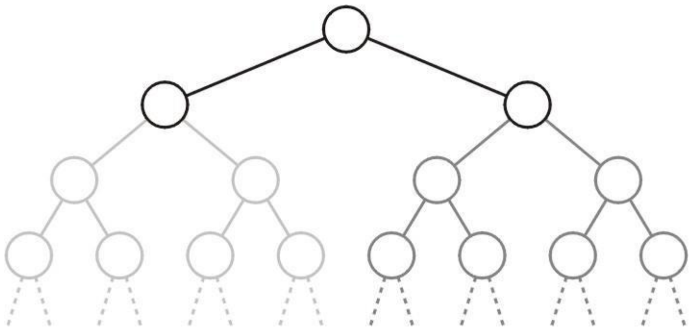
- Each child is either empty or another binary tree
- This restriction allows us to label the children as *left* and *right* subtrees



Binary Sub-trees

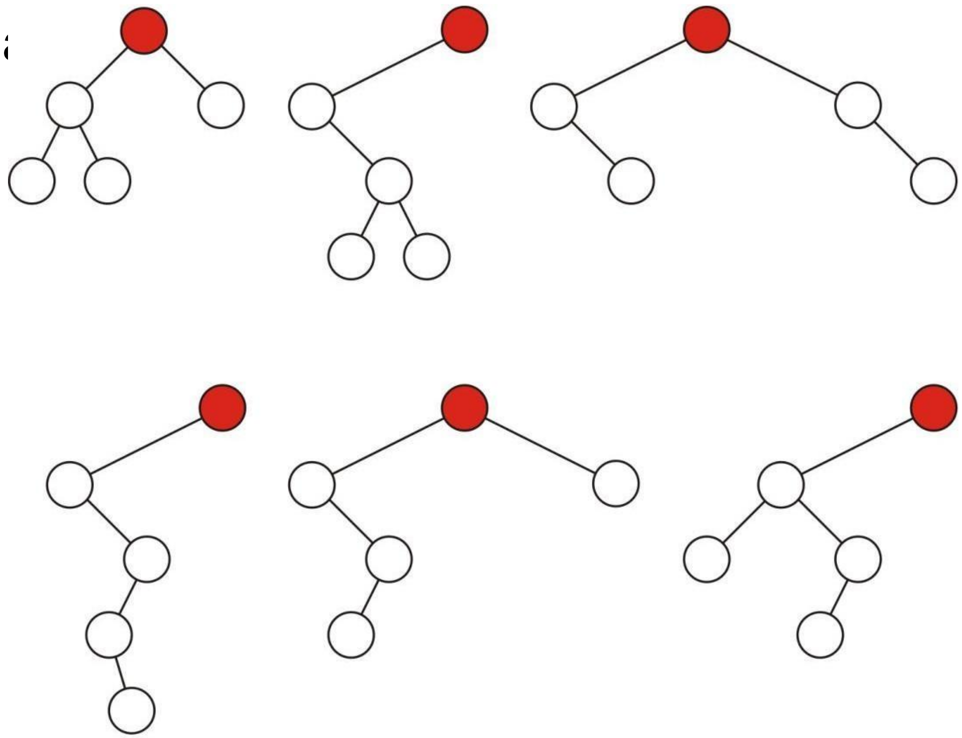
We will also refer to the two sub-trees as

- The left-hand sub-tree, and
- The right-hand sub-tree



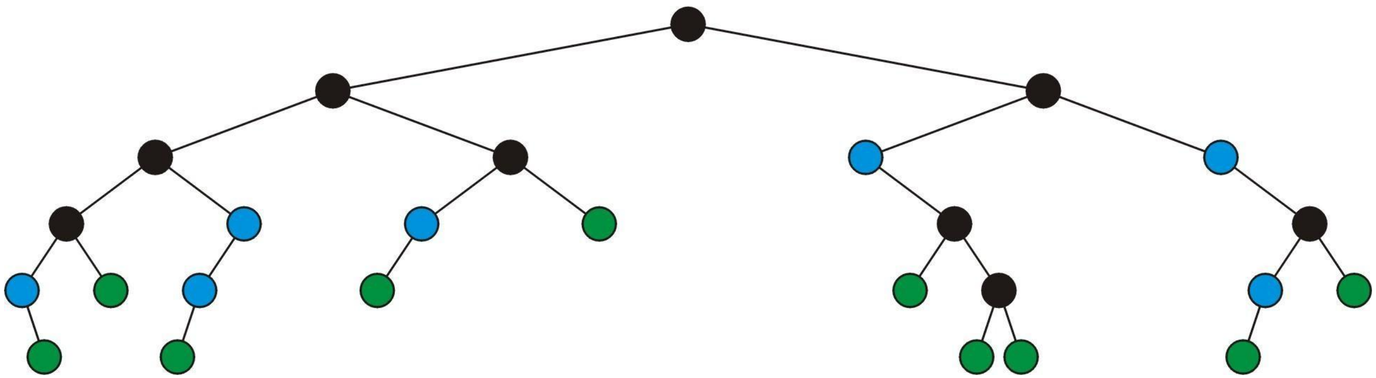
Sample Binary Trees

Sample vari



Definition (Full Node)

A *full* node is a node where both the left and right sub-trees are non-empty trees

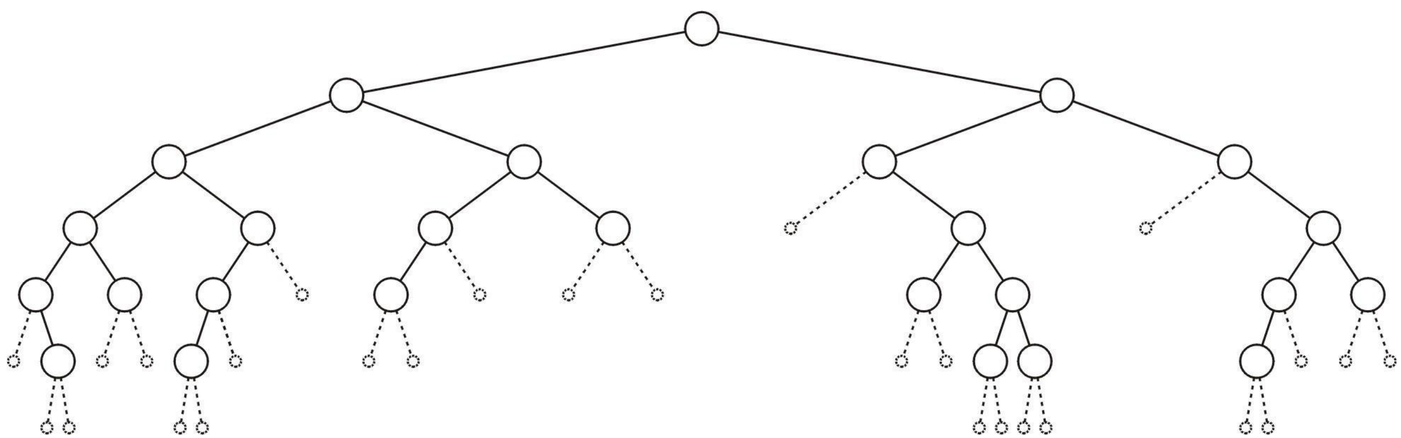


Legend:

full nodes neither leaf nodes

Definition(Empty Node)

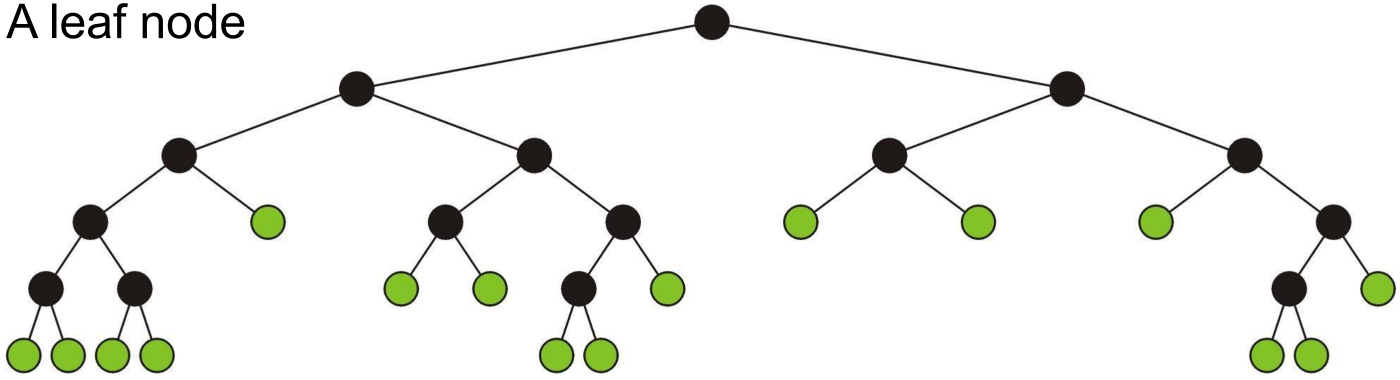
An *empty node* or a *null sub-tree* is any location where a new leaf node could be appended



Full Binary Tree

A *full binary tree* is where each node is:

- A full node, or
- A leaf node



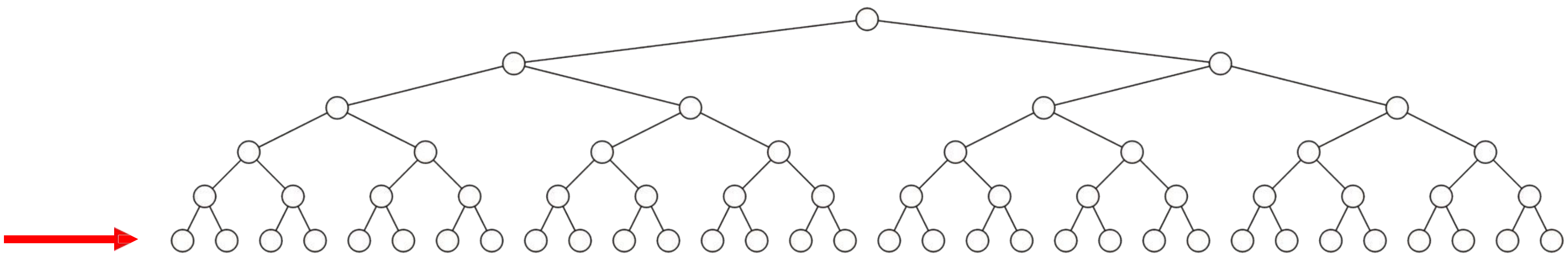
These have applications in

- Expression trees
- Huffman encoding

Perfect Binary Tree

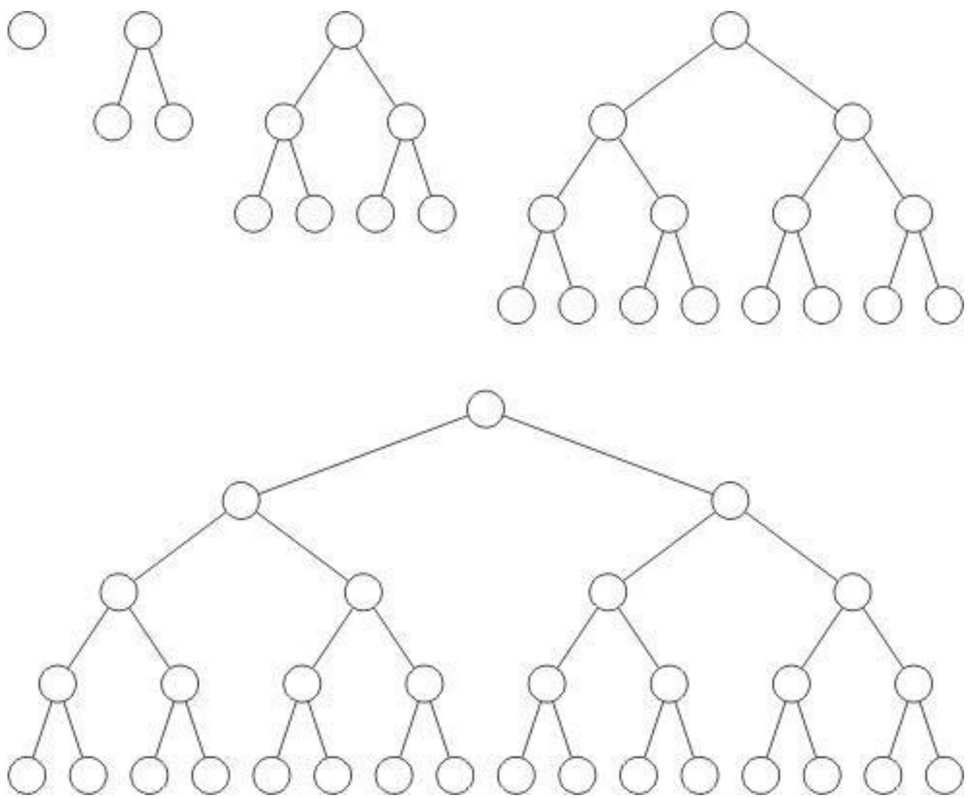
Standard definition:

– A perfect binary tree of height h is a binary tree where
  - All leaf nodes have the same depth h
  - All other nodes are full



Exampl es

Perfect binary trees of height $h = 0, 1, 2, 3$ and 4



Perfect Binary Trees

Perfect binary trees are considered to be the *ideal* case

– The height and average depth are both $\Theta(\ln(n))$

We will attempt to find trees which are as close as possible to perfect binary trees

One of the limitations of perfect binary trees is restricted number of nodes.

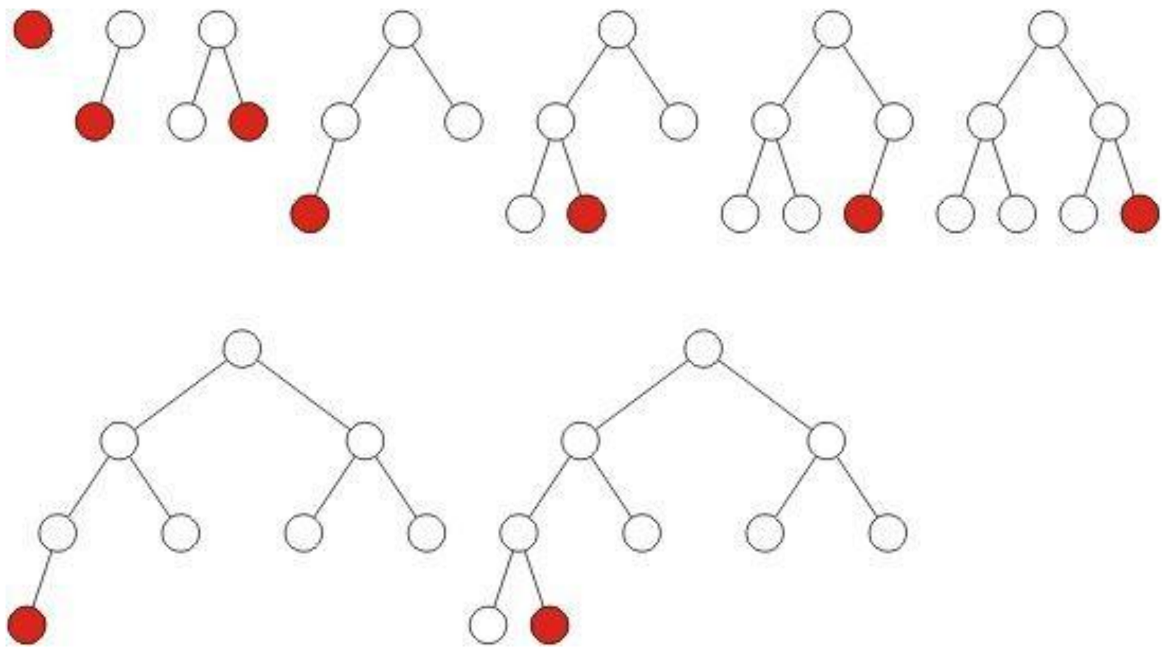
Complete Binary Trees

We require binary trees which are

- Similar to perfect binary trees, but
- Defined for all n

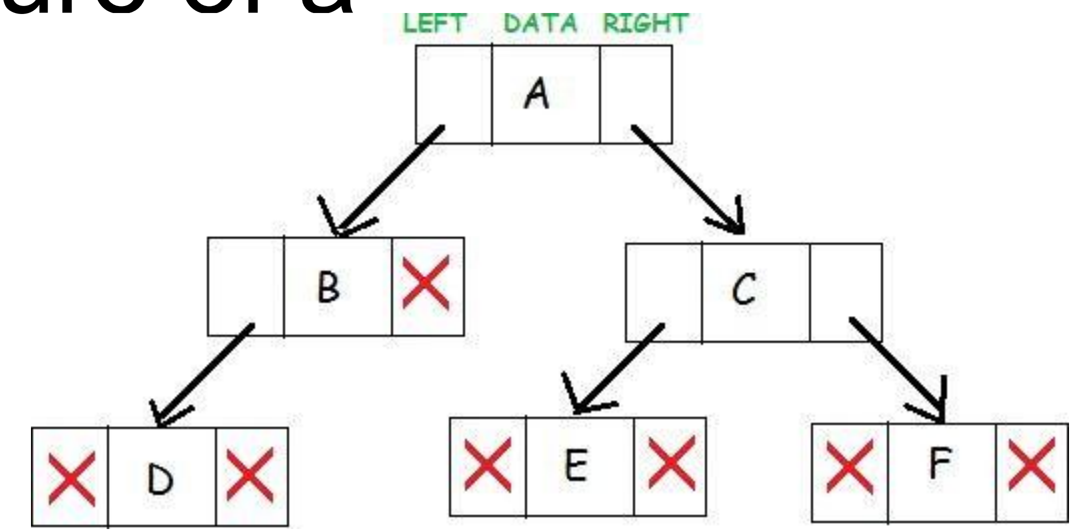
Complete Binary Trees

A complete binary tree filled at each depth from left to right:



Implementation Details Structure of a Node

Struct
node{ int
data;
node*
right;
node* left;
};
Struct node* newNode(int
value){
node->data=valu
e;
Node->left=NULL
;
Node->right=NUL
L; Return (node);
}



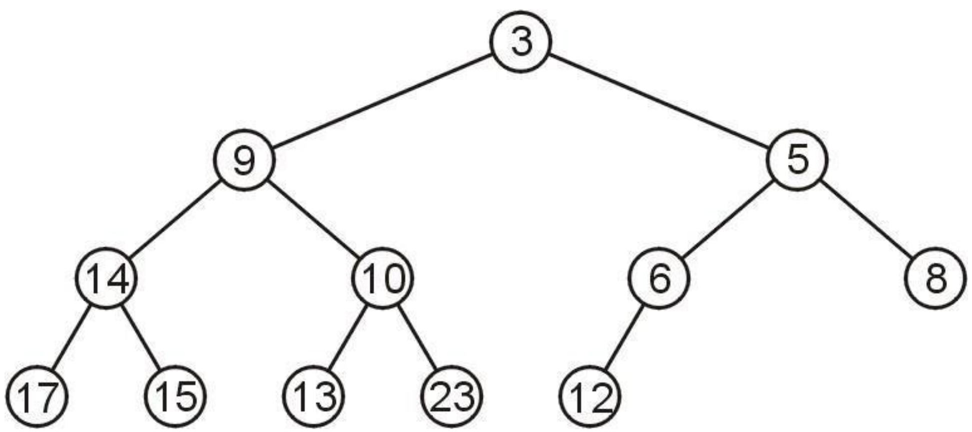
Implementation Details of Complete Binary Tree

- Operations
  - Insert
  - Update
  - Search
  - Delete
  - Traversal

Array Storage

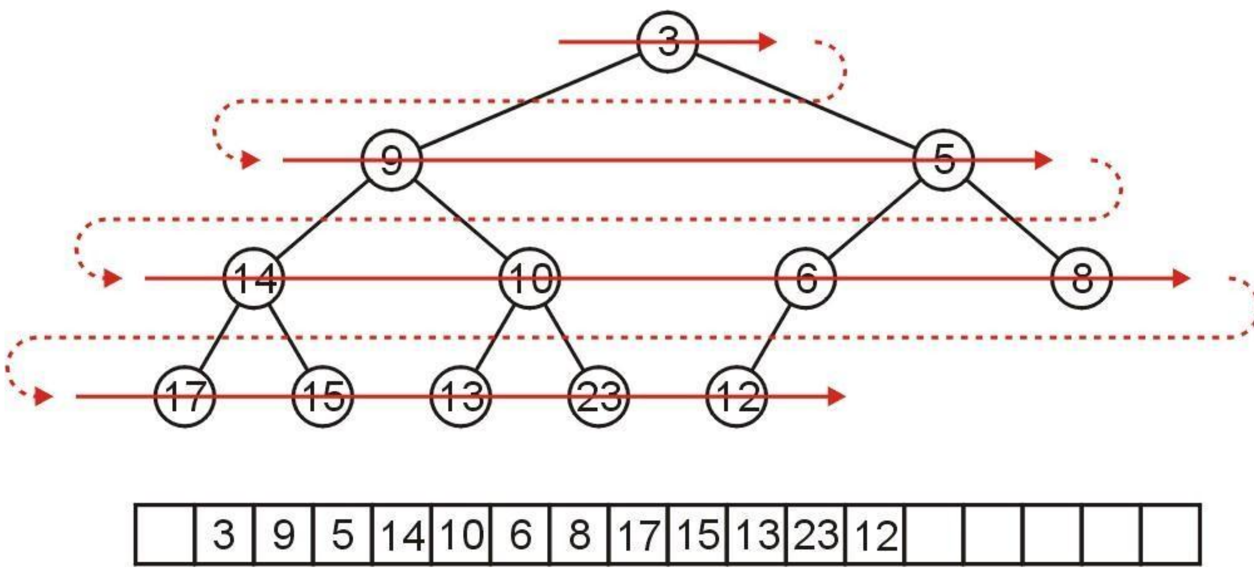
We are able to store a complete tree as an array

– Traverse the tree in breadth-first order, placing the entries into the array



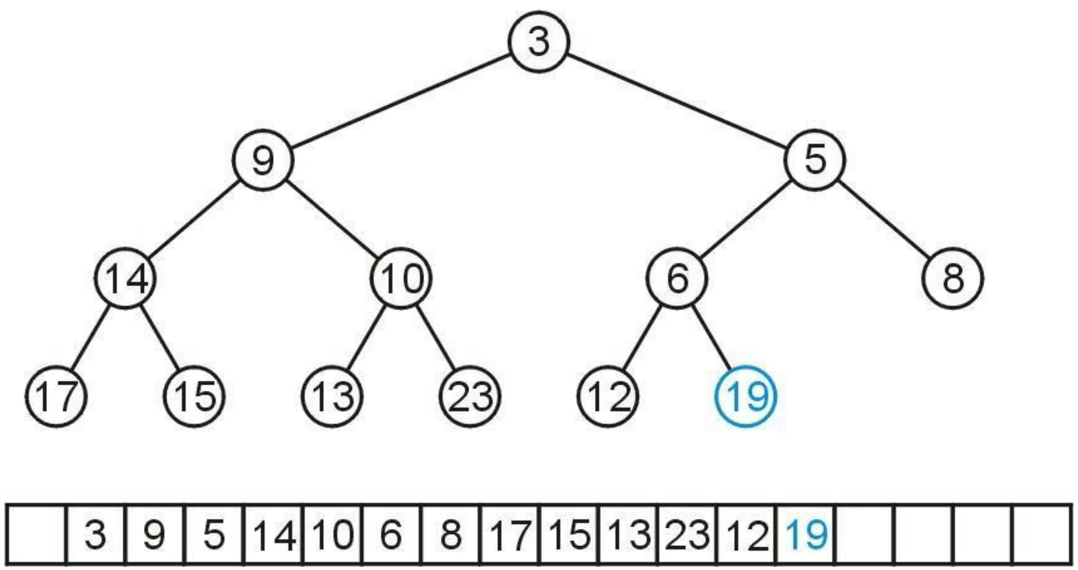
Array Storage (Insertion)

We can store this in an array after a quick traversal:



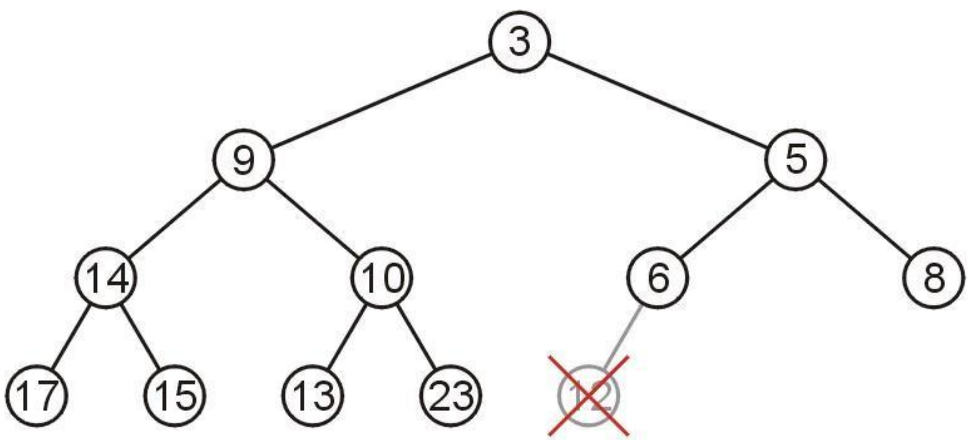
Array Storage (Insertion)

To insert another node while maintaining the complete-binary-tree structure, we must insert into the next array location



Array Storage (Deletion)

To remove a node while keeping the complete-tree structure, we must remove the last element in the array

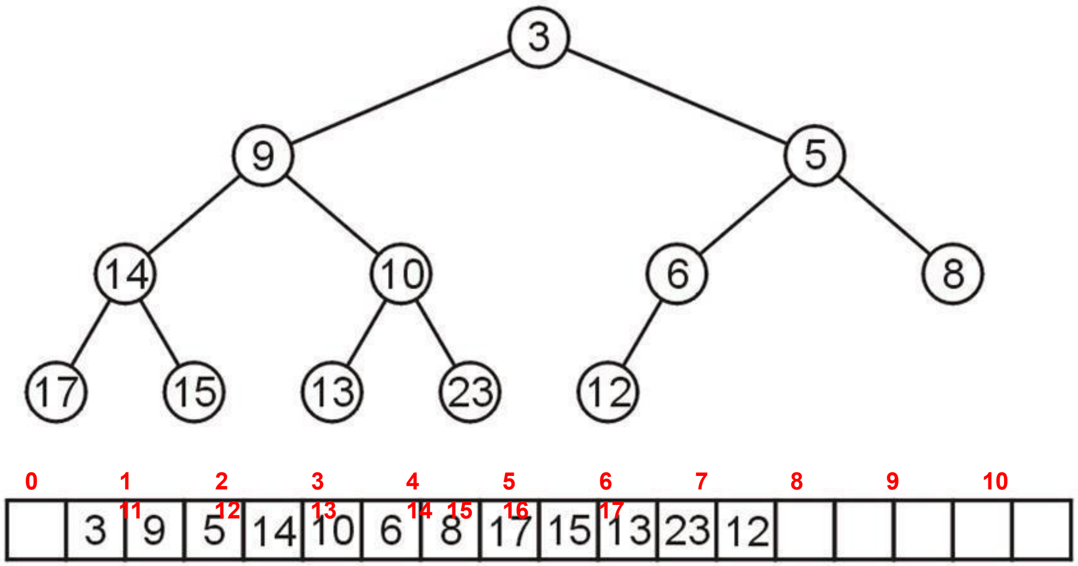


	3	9	5	14	10	6	8	17	15	13	23	~~12~~					

Array Storage(Finding Parent and Child Nodes)

Leaving the first entry blank yields a bonus:

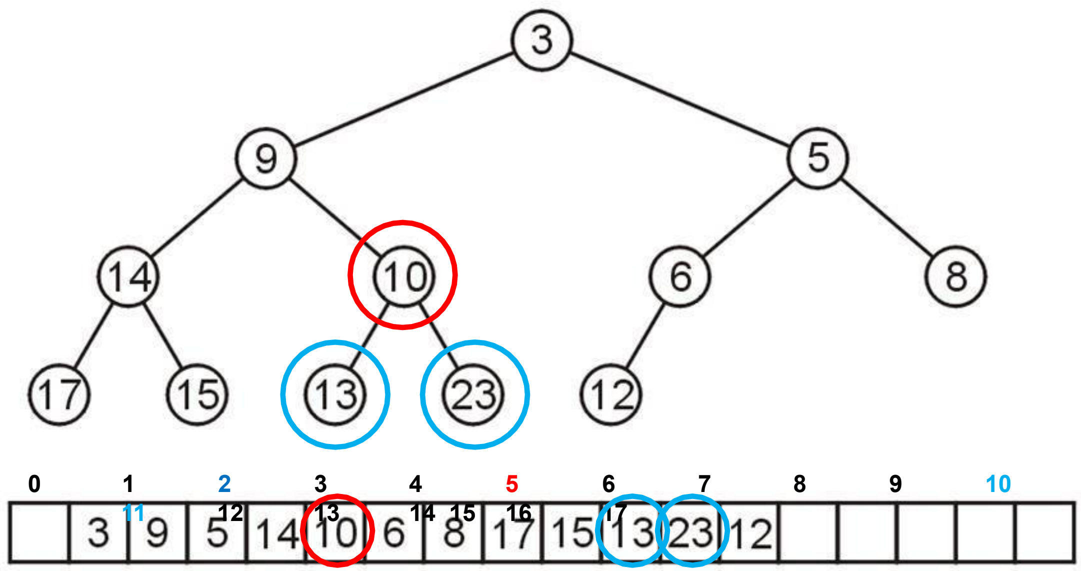
- The children of the node with index k are in $2k$ and $2k + 1$
- The parent of node with index k is in $k \div 2$



Array Storage

For example, node 10 has index 5:

– Its children 13 and 23 have indices 10 and 11, respectively



Outline

- Priority Queue
- Examples of Priority Queue
- Implementation details of Priority Queue
- Binary Heap
  - Min Heap
  - Max Heap
- Heap Sort

Priority Queue

With queues

- The order may be summarized by *first in, first out*

If each object is associated with a priority, we may wish to pop that object which has highest priority

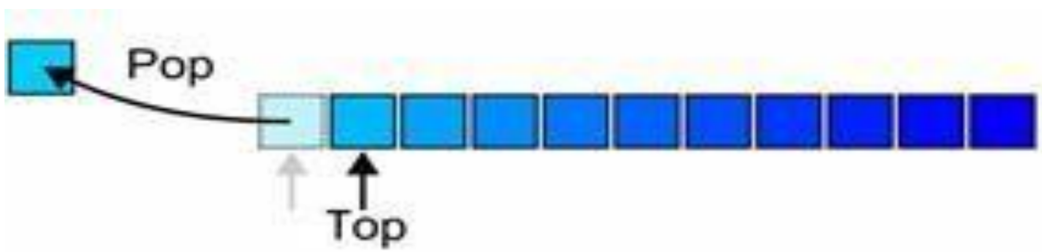
With each pushed object, we will associate a nonnegative integer (0, 1, 2, ...) where:

- The value 0 has the *highest* priority, and
- The higher the number, the lower the priority

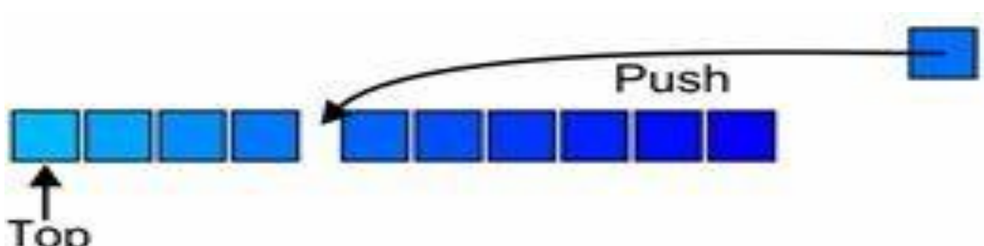
Operations

The top of a priority queue is the object with highest

Popping from a priority queue removes the current highest priority object:



Push places a new object into the appropriate place



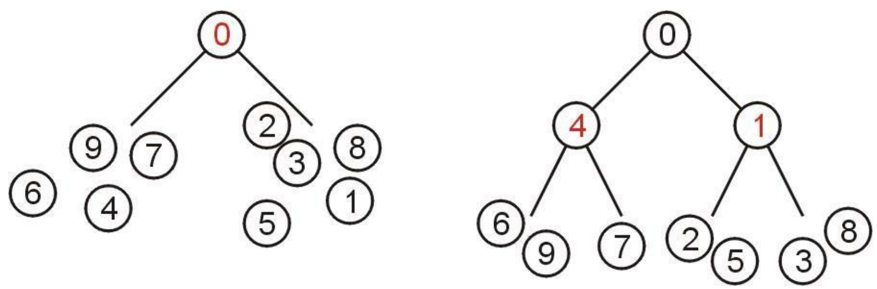
Hea p

- Heap is a tree with the highest priority at the root.
- We will look at binary heaps
- Numerous other heaps exists:
  - D-ary heaps
  - Leftlist heaps
  - Skew heaps
  - Binomial heaps
  - Fibonacci heaps
  - Bi-parental heaps

Heap

A non-empty binary tree is a min-heap if

- The key associated with the root is less than or equal to the keys associated with either of the sub-trees (if any)
- Both of the sub-trees (if any) are also binary min-heaps



From this definition:

- A single node is a min-heap
- All keys in either sub-tree are greater than the root key

Operations on Heap

We will consider three operations:

- Top
- Pop
- Push

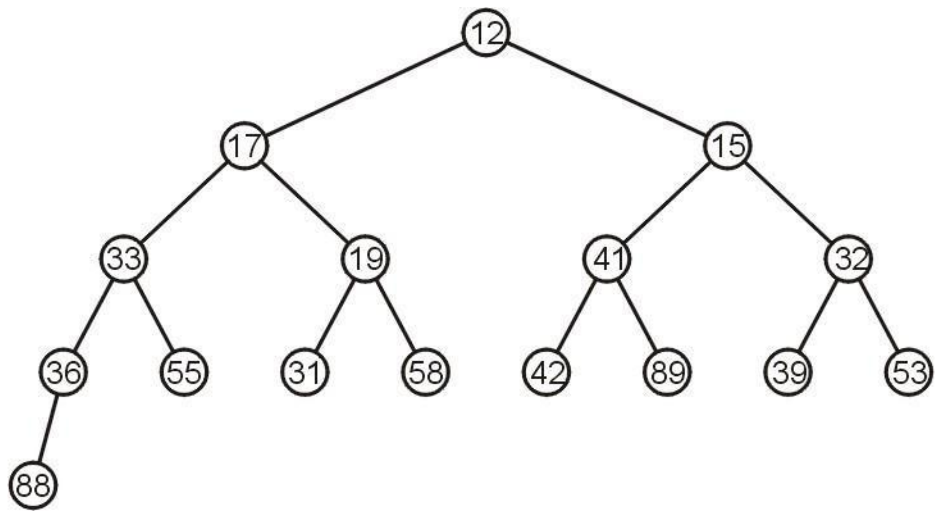
Pop

To remove the minimum object:

- Promote the node of the sub-tree which has the least value
- Recurs down the sub-tree from which we promoted the least value

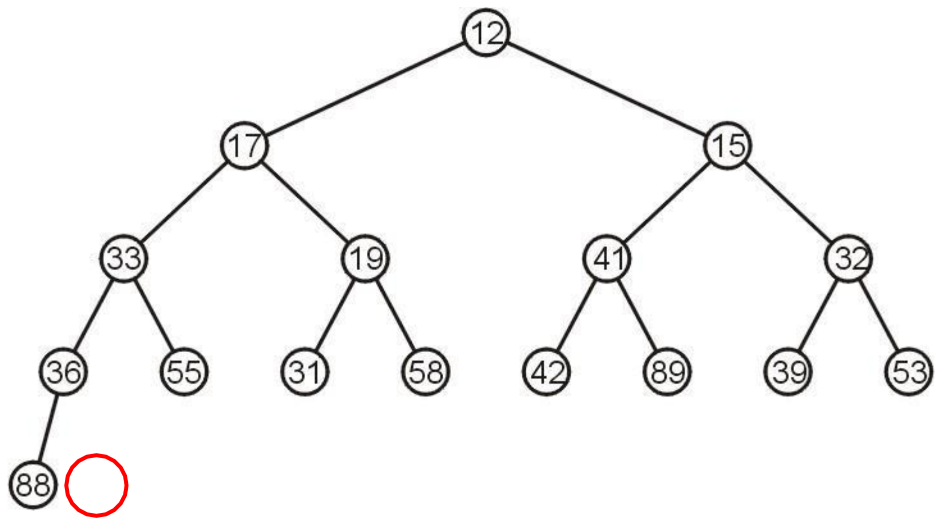
Exampl
e

For example, the previous heap may be represented as the following (non-unique!) complete tree:



Complete Trees: Push

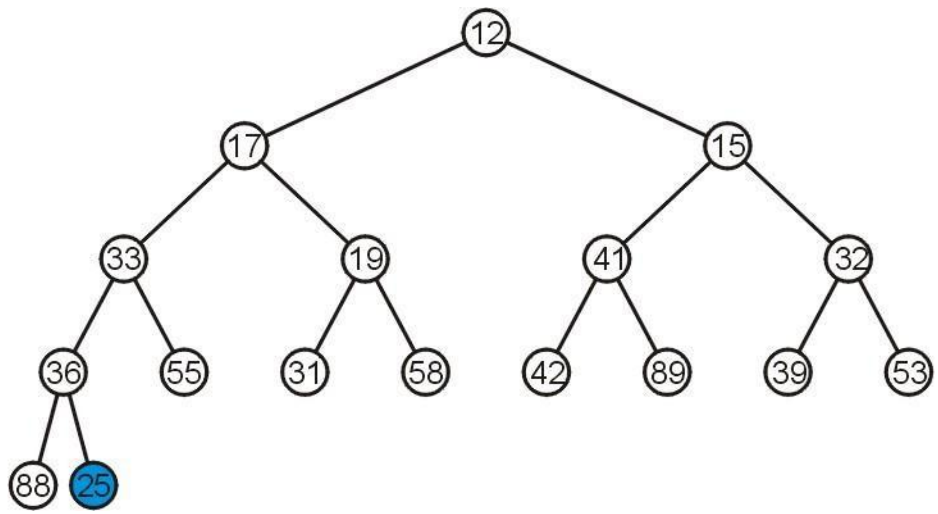
If we insert into a complete tree, we need only place the new node as a leaf node in the appropriate location and percolate up



Complete Trees:

Push

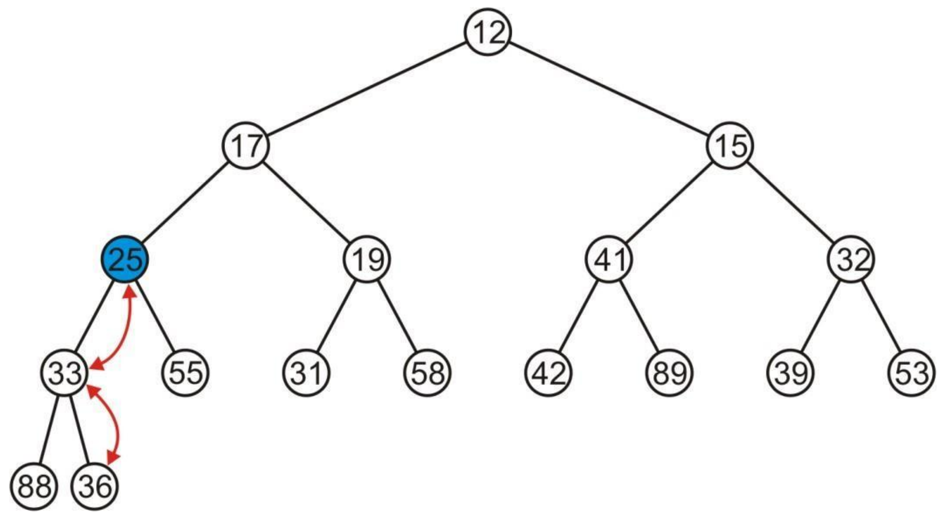
For example, push 25:



Complete Trees: Push:

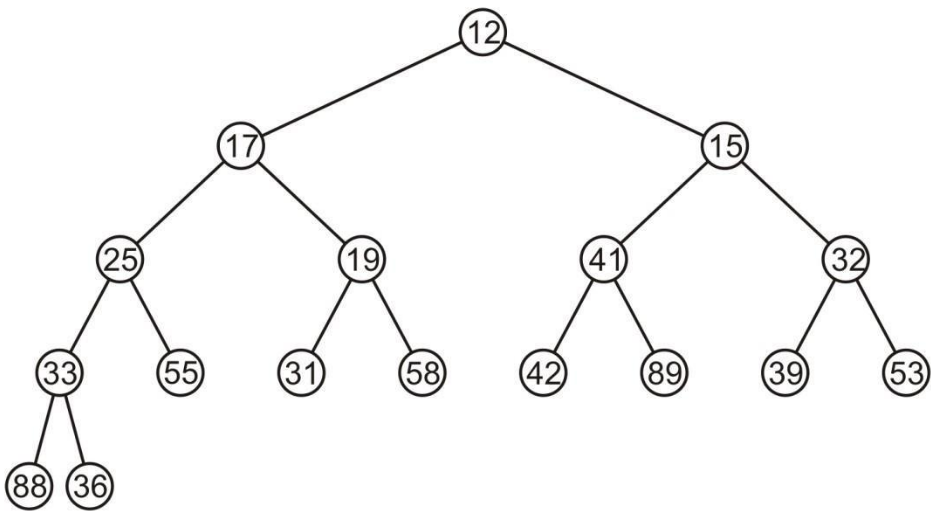
We have to percolate 25 up into its appropriate location

– The resulting heap is still a complete tree



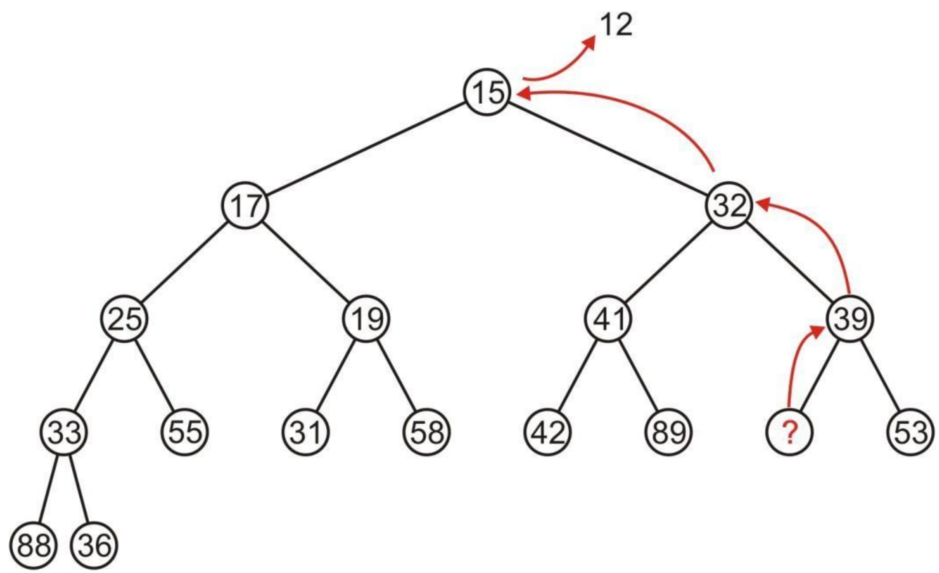
Complete Trees: Pop

Suppose we want to pop the top entry: 12



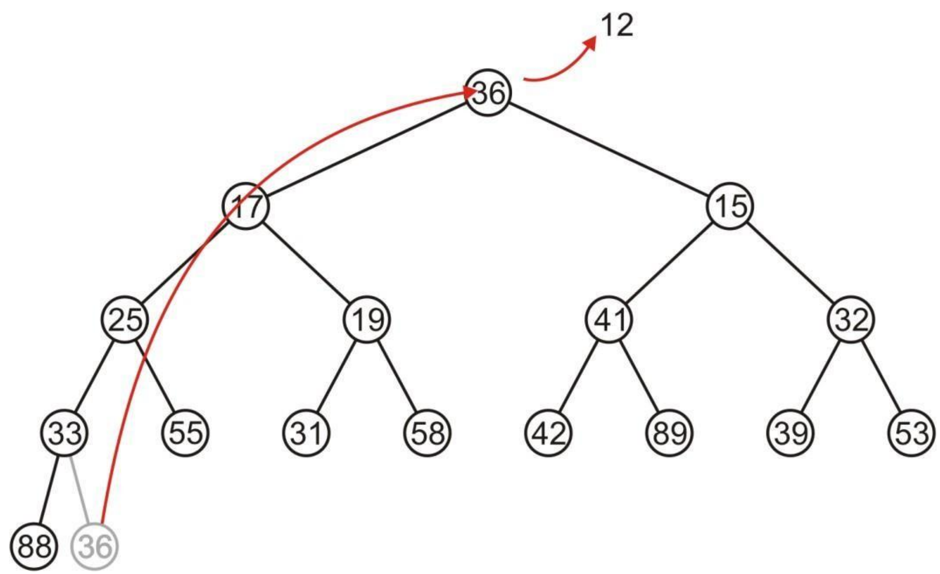
Complete Tree Properties:

Percolating up creates a hole leading to a non-complete tree



Complete Trees: Pop

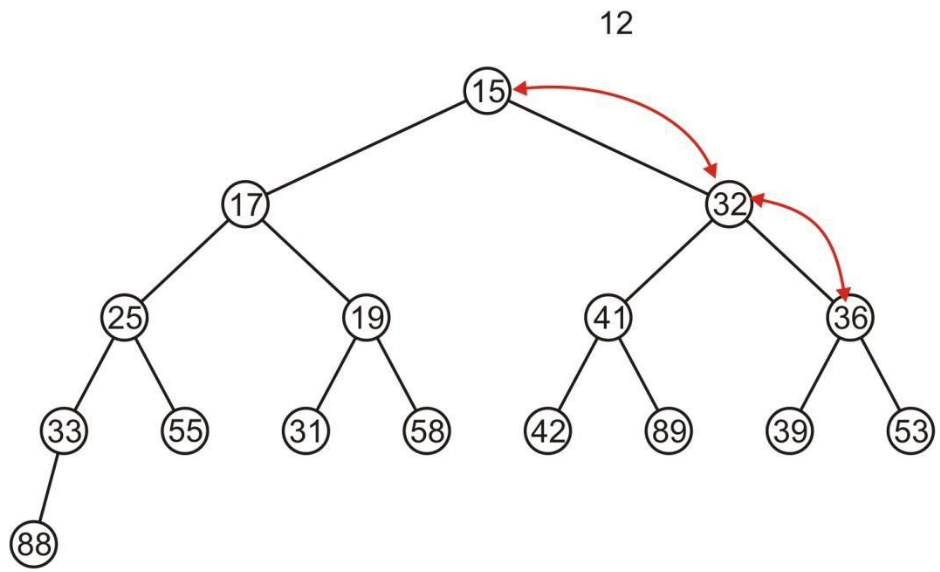
Alternatively, copy the last entry in the heap to the root



Complete Trees: Pop

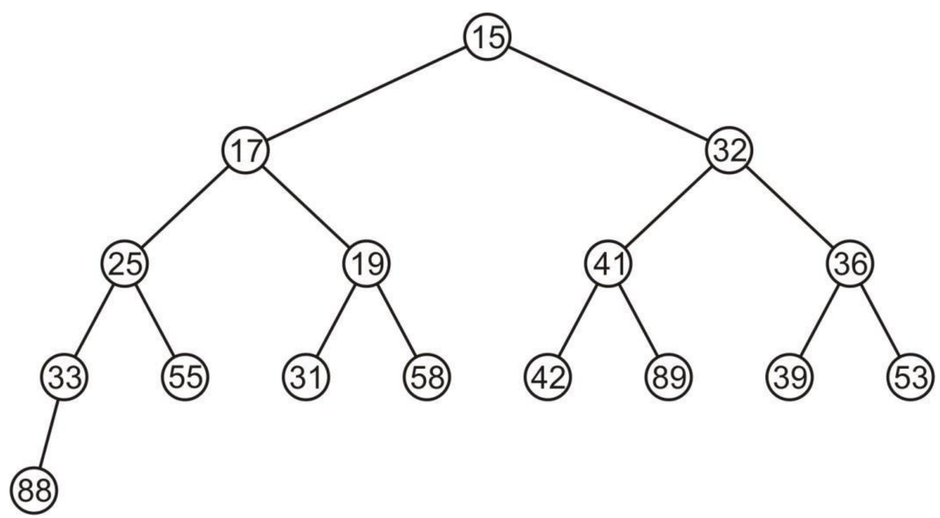
Now, percolate 36 down swapping it with the smallest of its children

- We halt when both children are larger



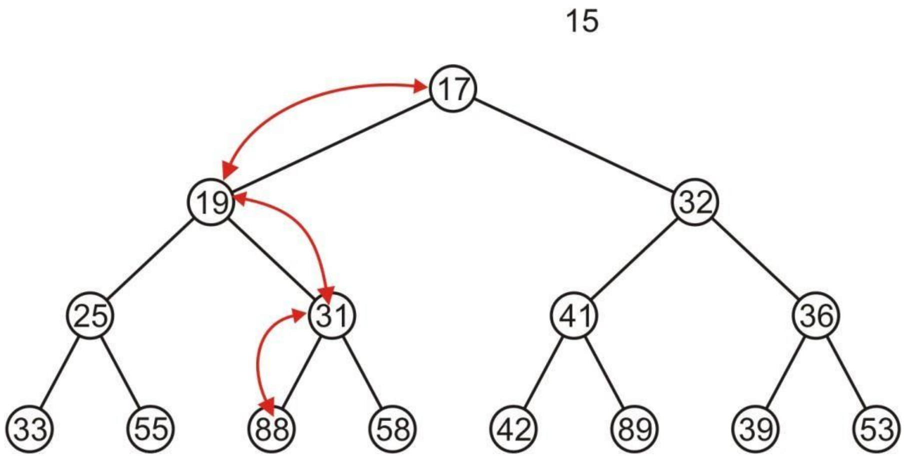
Complete Trees: Pop

The resulting tree is now still a complete tree:



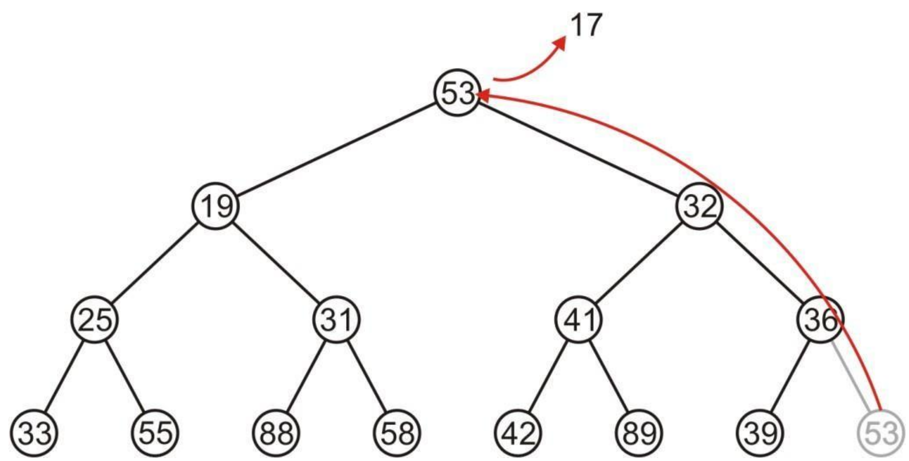
Complete Trees:

This time, it gets percolated down to the point where it has no children



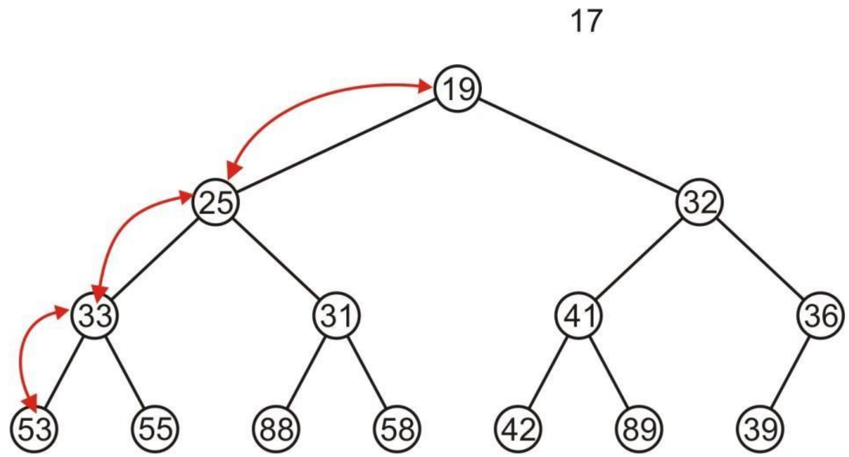
Complete Trees: Pop

In popping 17, 53 is moved to the top



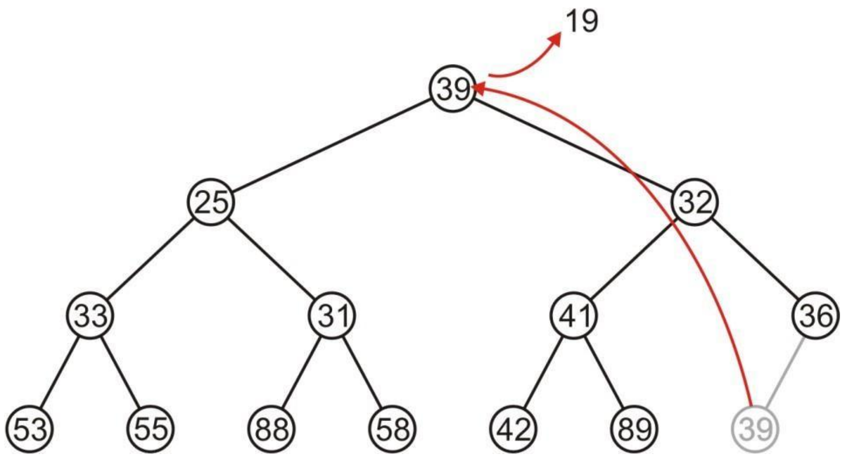
Complete Trees: Pop

And percolated down, again to the deepest level



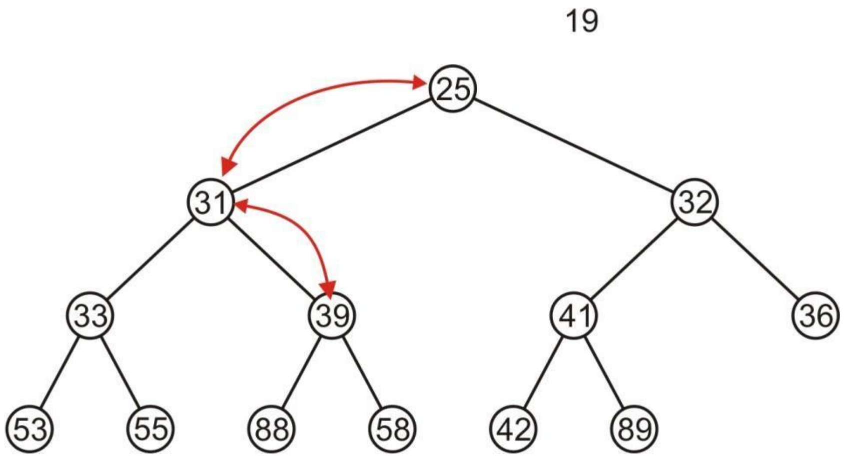
Complete Trees: Pop

Popping 19 copies up 39



Complete Trees:

Which is then percolated down to the second deepest level



Complete Tree

Therefore, we can maintain the complete-tree shape of a heap

We may store a complete tree using an array:

- A complete tree is filled in breadth-first traversal order
- The array is filled using breadth-first traversal

Heap Sort

- Discussion

0	1	2	3	4	5	6	7	8	9
	3	1	4	7	8	6	0	2	5

- Step 01: Build Min Heap/Max Heap
- Step 02: Swap First index with Last index
- Step 03: Update heap size(HS)
- Step 04: Apply heapify on HS

Run-time Analysis

Accessing the top object is $\mathbf{\Theta}(1)$

Popping the top object is $\mathbf{O}(\ln(n))$

– We copy something that is already in the lowest depth—it will likely be moved back to the lowest depth

Priority Queues

Now, does using a heap ensure that that object in the heap which:

- has the highest priority, and
- of that highest priority, has been in the heap the longest

Consider inserting seven objects, all of the same priority (colour indicates order):

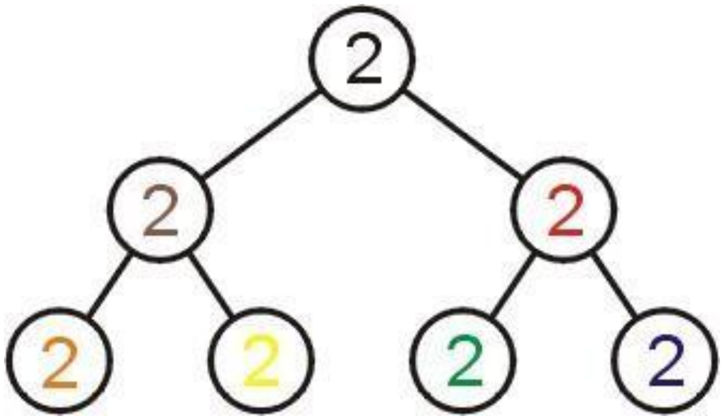
2, 2, 2, 2, 2, 2, 2

Priority Queues

Whatever algorithm we use for promoting must ensure that the first object remains in the root position

- Thus, we must use an insertion technique where we only percolate up if the priority is lower

The result:



Challenge:

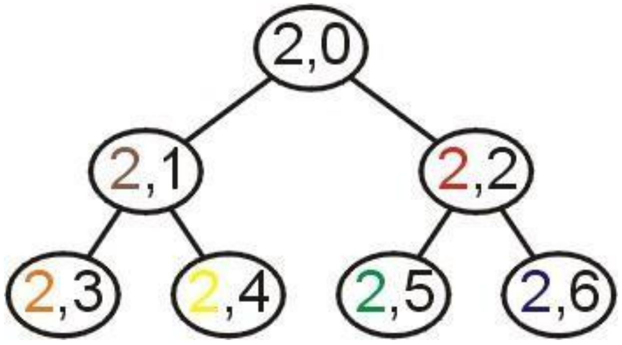
- Come up with an algorithm which removes all seven objects in the original order

Lexicographical Ordering

A better solution is to modify the priority:

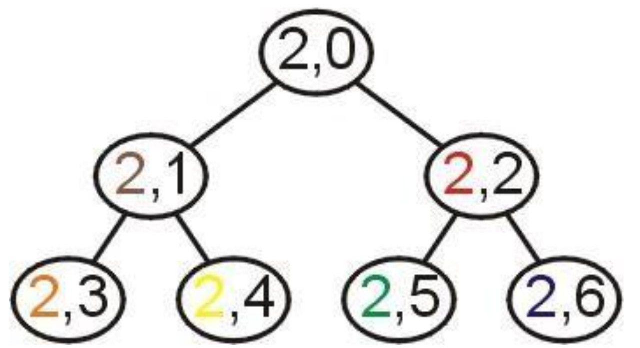
- Track the number of insertions with a counter k (initially 0)
- For each insertion with priority n, create a hybrid priority (n, k) where:

  $(n_1, k_1) < (n_2, k_2)$ if $n_1 < n_2$ or ($n_1 = n_2$ and $k_1 < k_2$)



Priority Queues

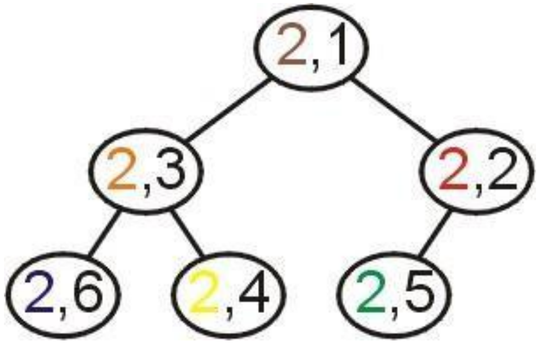
Removing the objects would be in the following order:



Priority Queues

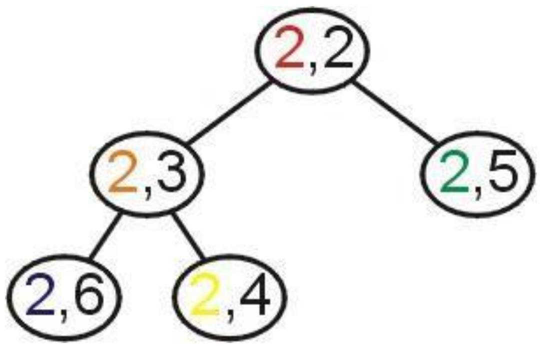
Popped: 2

– First, (2,1) < (2, 2) and (2, 3) < (2, 4)



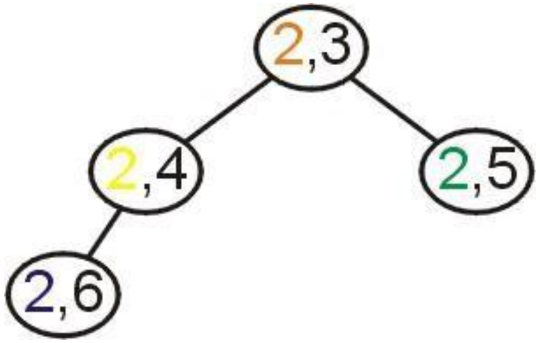
Priority Queues

Removing the objects would be in the following order:



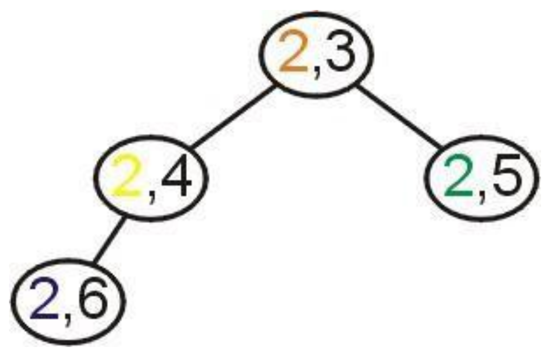
Priority Queues

Removing the objects would be in the following order:



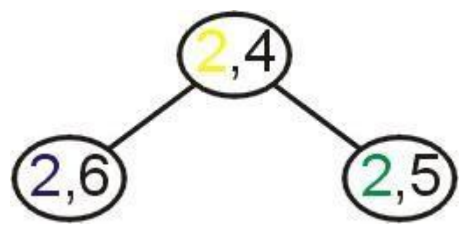
Priority Queues

Removing the objects would be in the following order:



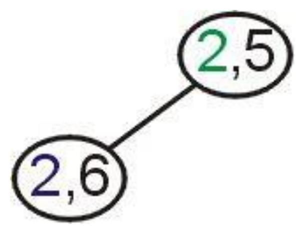
Priority Queues

Removing the objects would be in the following order:



Priority Queues

Removing the objects would be in the following order:



Summary

In this talk, we have:

- Discussed binary heaps
- Looked at an implementation using arrays
- Discussed implementing priority queues using binary heaps
- Discussed Heap Sort
- The use of a lexicographical ordering