Trees and Tree-Based Algorithms

Dr. Robert Amelard
(adapted from Dr. Igor Ivkovic)

ramelard@uwaterloo.ca

Objectives

- Introduction to Trees
- Binary Trees
- Binary Tree Traversals
- Binary Search Trees
- AVL Trees
- Heaps

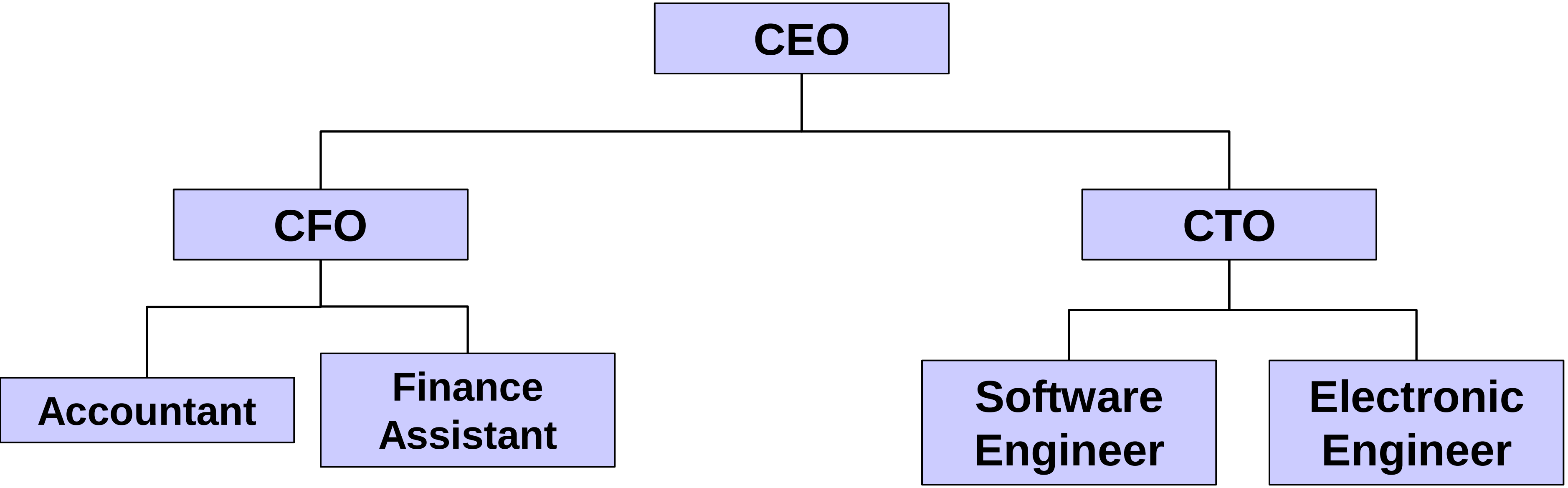
Up Till Now…

- Linear data structures
  - List, Queue, Stack

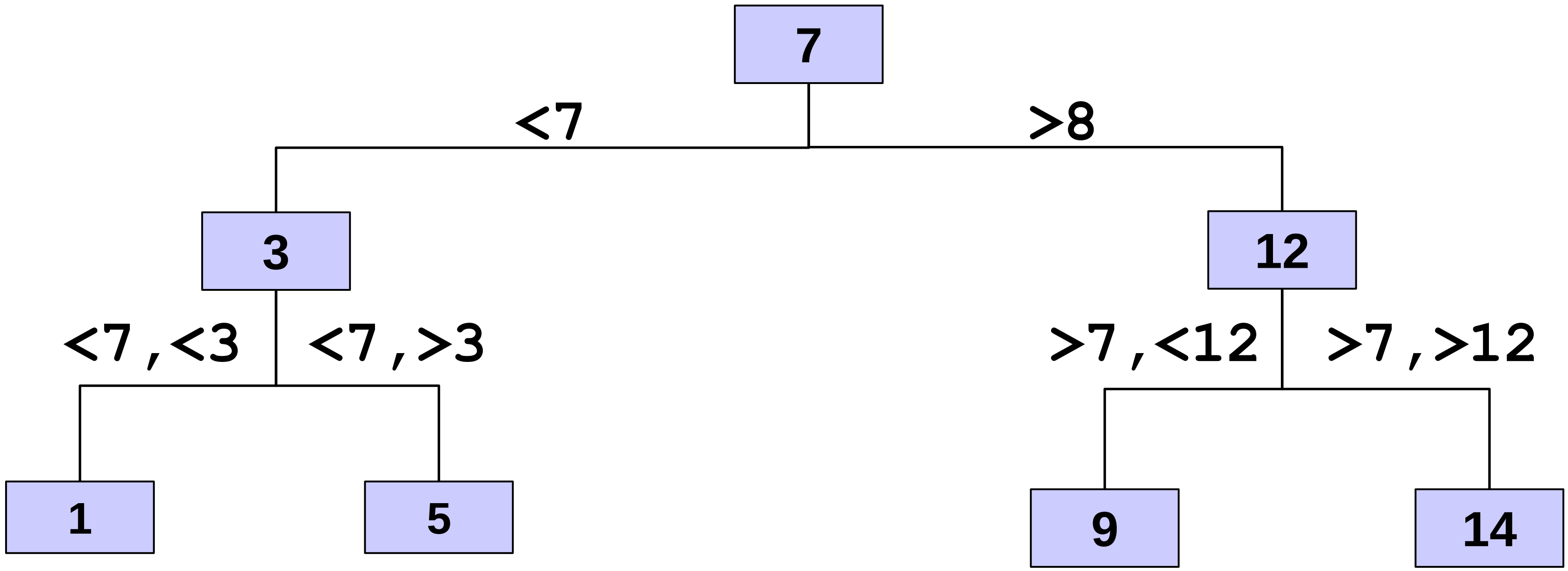
- Time to explore non-linear data structures
  - Powerful for real-life algorithms (searching, sorting, etc.)
  - Graphical depiction/optimization

- Trees: hierarchical

Hierarchical Data Structure Example



Hierarchical Data Structure Example



Introduction to Trees /1

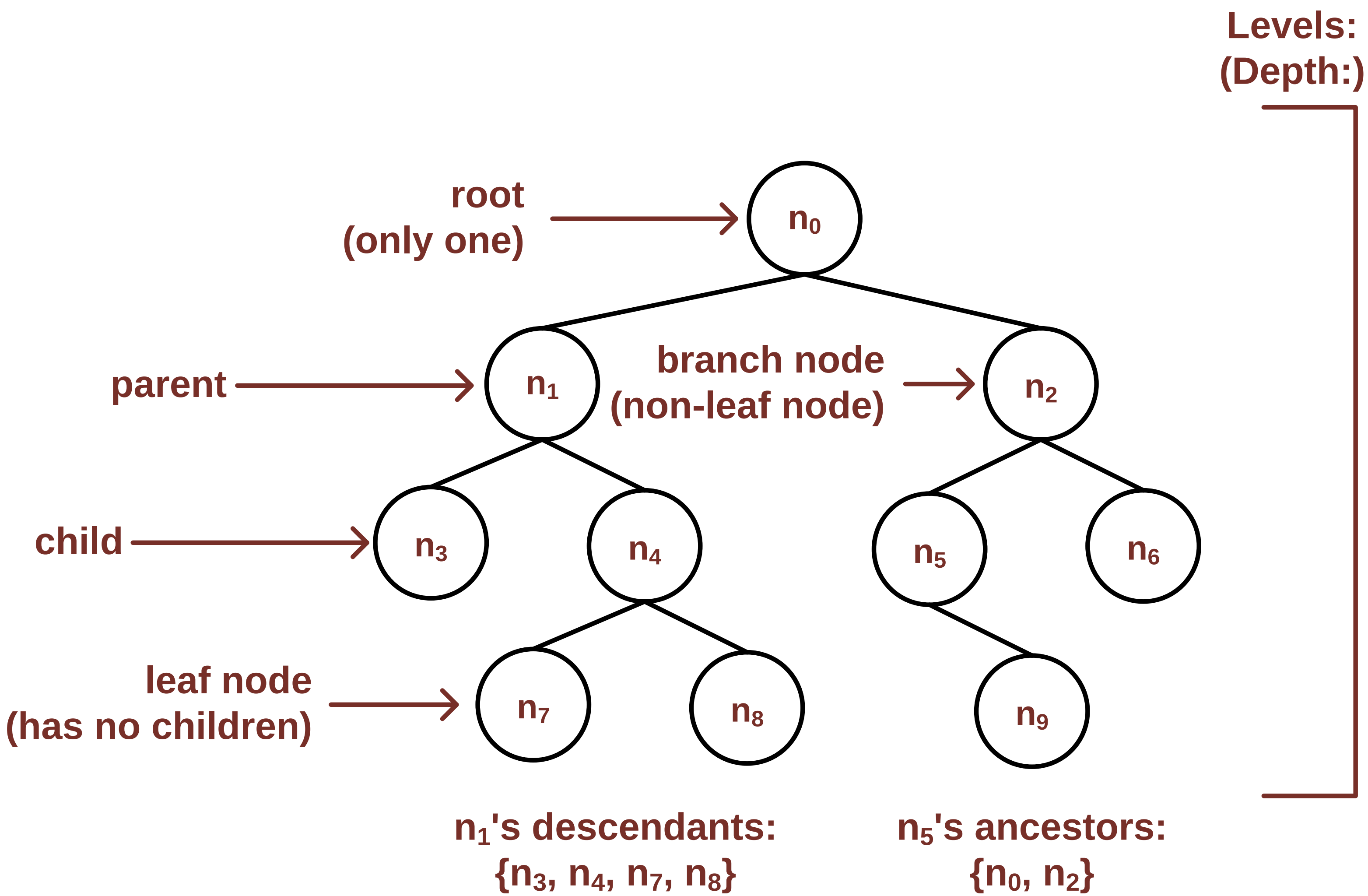
- **Tree (T) is a hierarchical data structure composed of linked nodes connected by edges**
  - There is one root node for the entire structure
  - Each node has zero or more nodes as its children
  - Each node has at most one parent node

  - All the nodes reachable from the current downwards to the bottom of the tree are its descendants
  - All the nodes reachable from the current upwards to the root of the tree are its ancestors

  - Typically, there is an unidirectional downwards relationship between a node and its descendants
  - A node may access its descendants but descendants cannot access their ancestors

Introduction to Trees /2

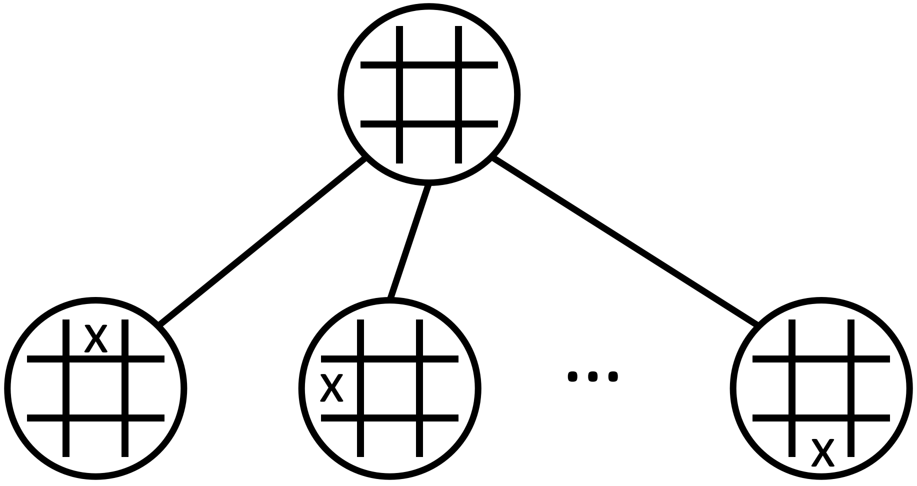
- **Core Tree Terminology:**



Introduction to Trees /3

- **Tree Applications:**
  - In **artificial intelligence**, game trees are used to encode data regarding decision making
  - In a **game tree**, the nodes represent the possible outcomes of making a move or decision
  - At the same time, the edges between the nodes represent possible moves or decisions

- **Example: Tic-Tac-Toe Game Tree**



Introduction to Trees /4

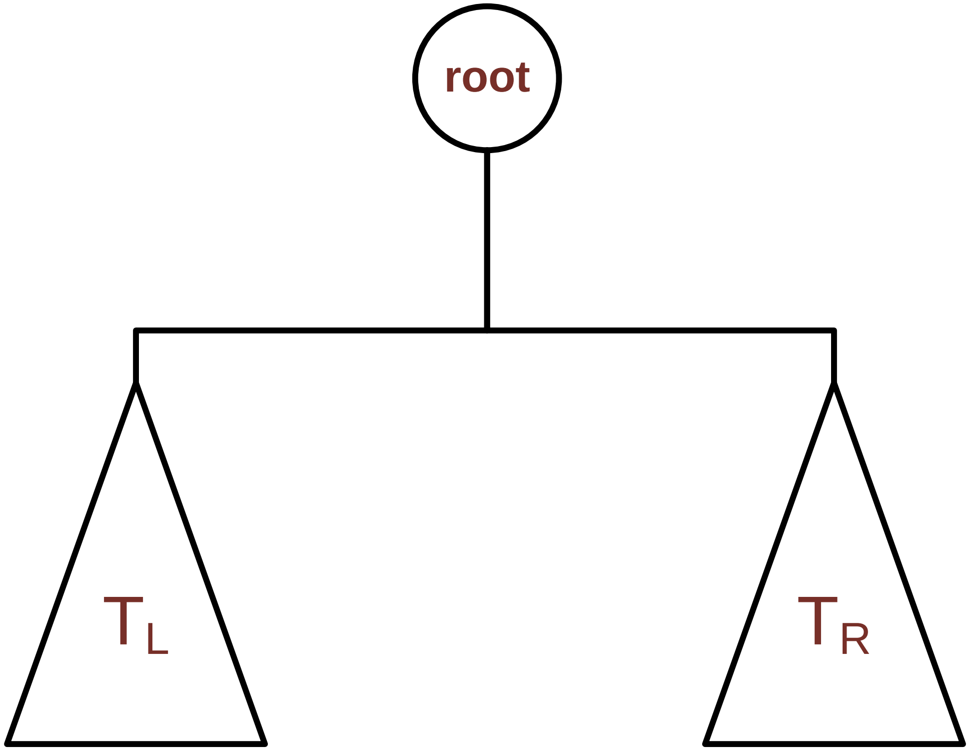
- TIP #1: A powerful outcome of using trees is they usually give us a runtime containing log(n). Why?

Introduction to Trees /5

- TIP #2: visualgo.net is a great resource for visualization tree operations (i.e., study guide)

Binary Tree /1

- **Binary Tree**
  - A finite set of nodes that is either empty or it consists of a root and two disjoint binary trees, T_L and T_R



Binary Tree /2

- **BinaryTreeNode Implementation:**

BinaryTreeNode
-iData : int **-*leftChild : BinaryTreeNode** **-*rightChild : BinaryTreeNode**
+BinaryTreeNode() **+~BinaryTreeNode()** **+left() : BinaryTreeNode** **+right() : BinaryTreeNode**

```
class BinaryTreeNode {
        int iData; //holds the data value at this tree node
        BinaryTreeNode *leftChild; //points to left child
        BinaryTreeNode *rightChild; //points to right child
public:
        BinaryTreeNode(); //default constructor
        ~BinaryTreeNode(); //destructor

        BinaryTreeNode left(); //returns left child
        BinaryTreeNode right(); //returns right child
};
```

Binary Tree /3

- **Binary Tree Node Depth**
  - The length of the path from the root to the current node (counting the edges)

- **Binary Tree Height**
  - The length of the longest path from the root to a leaf node (counting the edges)

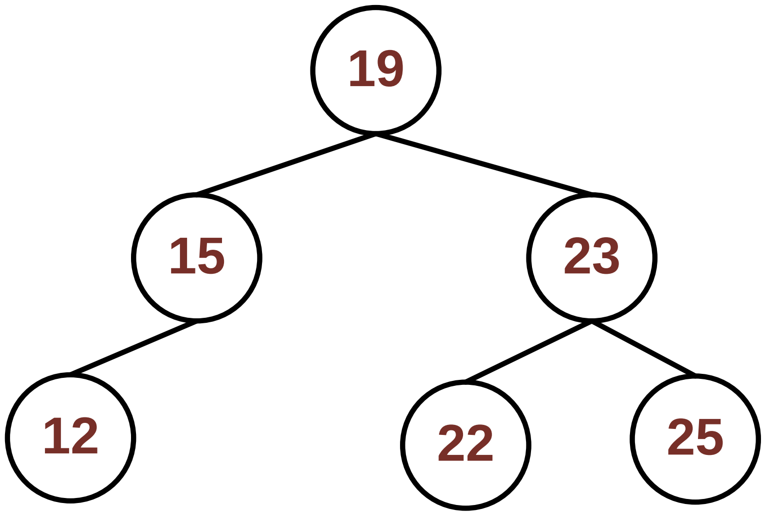
- **Algorithm: Height(T) (recursive)**
  - Input: BinaryTreeNode T
  - Output: the height of the binary tree T as an integer
  - Steps:

```
int Height(BinaryTreeNode* T)
    if (T == NULL) return -1; // returns -1 for an empty tree
    else return 1 + max(Height(T->leftChild), Height(T->rightChild));
```

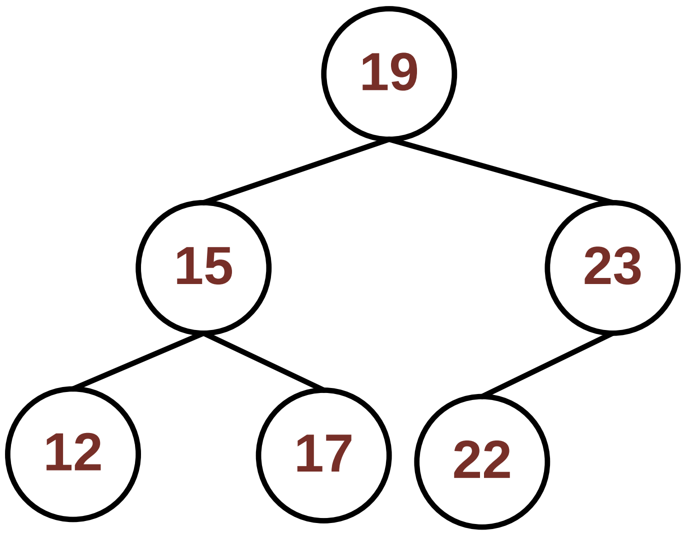
Complete Binary Tree /1

- **Complete Binary Tree:**
  - A binary tree that is completely filled at all levels with the exception of the bottom-most level
  - At the bottom-most level, all leaf nodes are **as far left as possible**

Not a complete binary tree:

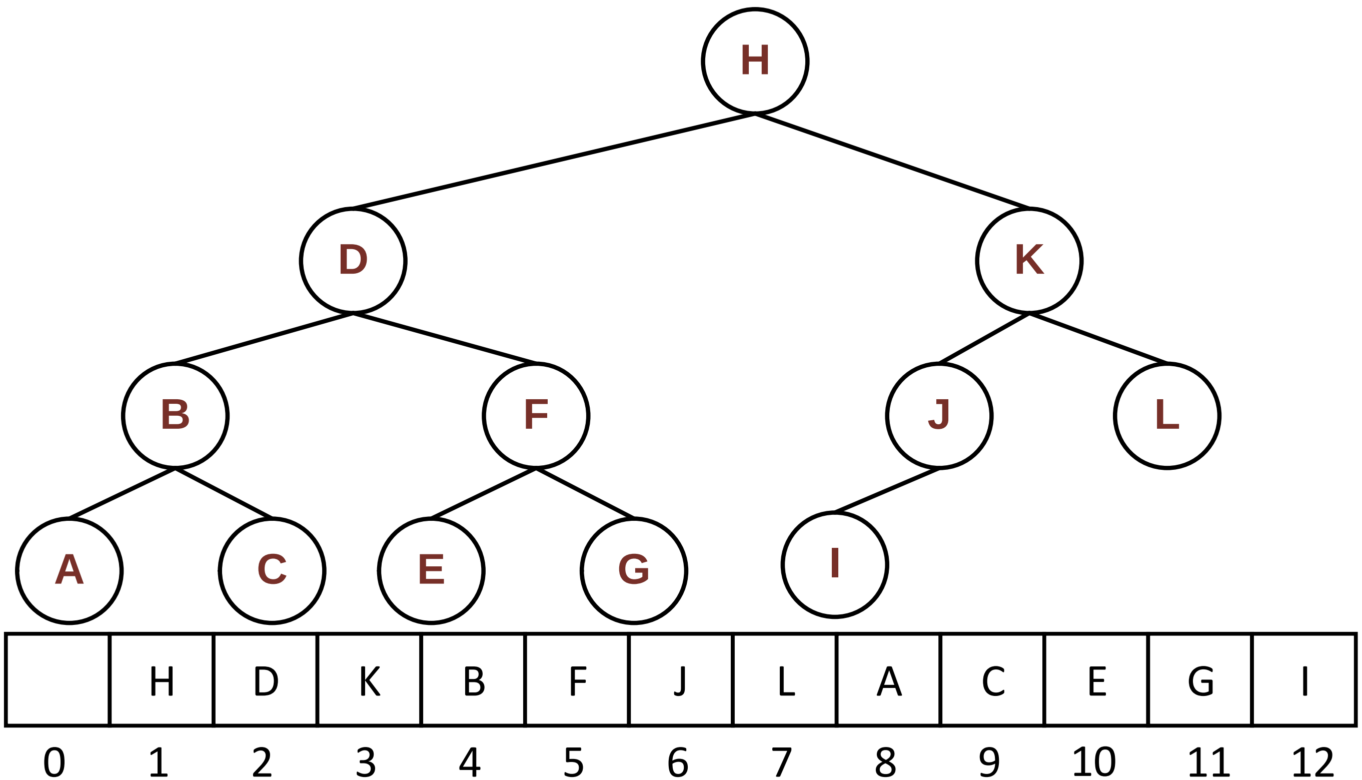


Complete binary tree:



Complete Binary Tree /2

- **Complete Binary Tree Access Formulas:**
  - Root node: array index 1
  - Parent of node i: $\text{floor}(i/2)$
  - Left child of node i: $\text{floor}(2i)$
  - Right child of node i: $\text{floor}(2i+1)$
  - Is node i a leaf: check if $n < 2i$



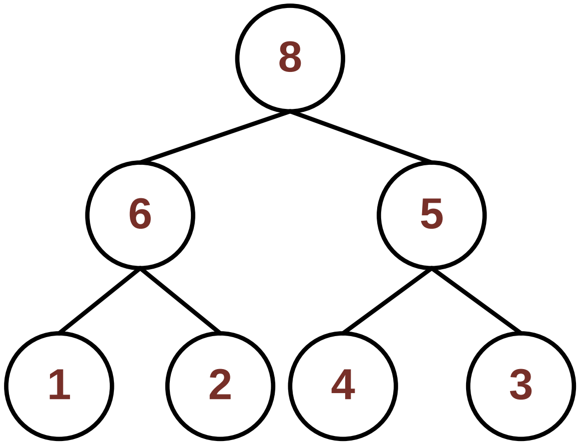
Complete Binary Tree /3

- Why are complete binary trees helpful?
  - Maintains non-linear hierarchical structure
  - Approximate knowledge of height
  - Compact array representation
- All helpful for runtime of algorithms!

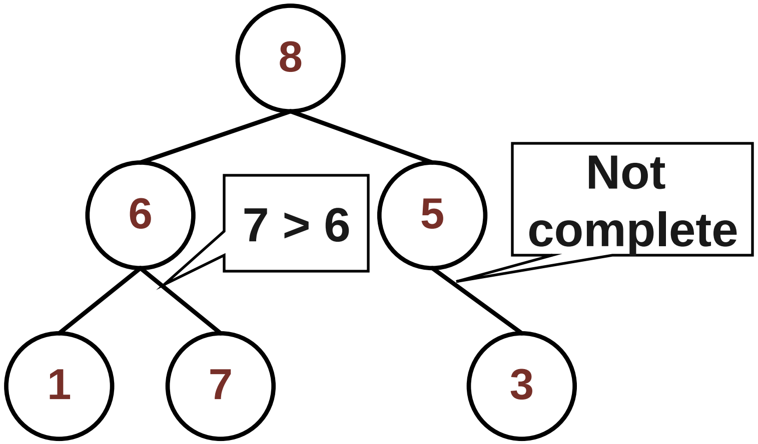
Heaps /1

- **Heap:**
  - A binary tree with keys or `(key,value)` pairs stored in its nodes
  - A heap is a **complete tree**
  - max-heap: all children smaller or equal to parent
  - min-heap: all children larger or equal to parent

- **Max-Heap Example:**



Not a Max-Heap:



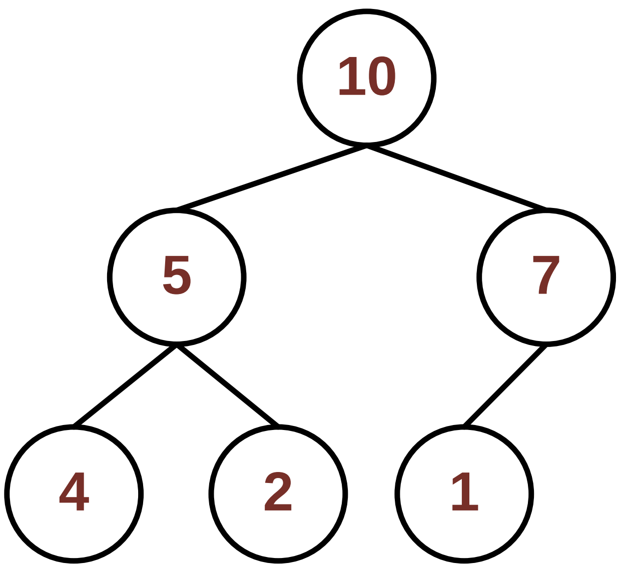
Heaps /2

- **(Max) Heap Properties:**
  - The root of a heap contains the largest key value
  - The subtree rooted at any node of a heap is also a heap
  - Remainder of array is “partially unsorted”
  - A heap can be represented as an array with relations between nodes computed using array indices

- **Heap Applications:**
  - Sorting values since the largest value in a heap will always be at the top
  - Priority queue, where a request of highest priority needs to be the dequeued first
  - CPU scheduling
  - Other?

Heaps /3

- **Heap as an Array – Access Formulas:**
  - Root node: index 1
  - Parent of node i: $\text{floor}(i/2)$
  - Left child of node i: $\text{floor}(2i)$
  - Right child of node i: $\text{floor}(2i + 1)$
  - Is node i a leaf: check if $n < 2i$



0	1	2	3	4	5	6
	10	5	7	4	2	1

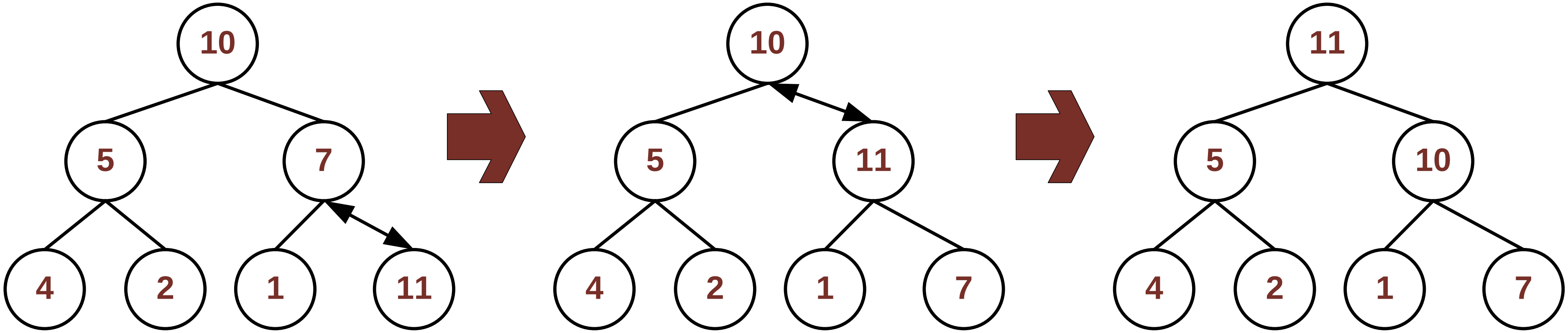
Heaps /4

- **Heap ADT Operations:**
  - **Insert:** insert a node into the heap
  - **Remove:** remove a node from the heap
  - **Heapify:** turn a binary tree into a heap

- **Insert(Node):**
  1. Insert node as bottom-right-most leaf in the tree
  2. If the parent has a smaller value than the node, switch places with the parent
  3. Continue recursing upwards
     - Until the parent has a higher (or equal) value than the node, or
     - Until the inserted node becomes the root node

Heaps /5

- **Insert(11) example:**
  - Insert 11 as the right-most leaf node
  - Ensure heap property compliance in a bottom-up manner
  - Swap 7 and 11 then 10 and 11 then terminate when 11 becomes the root node



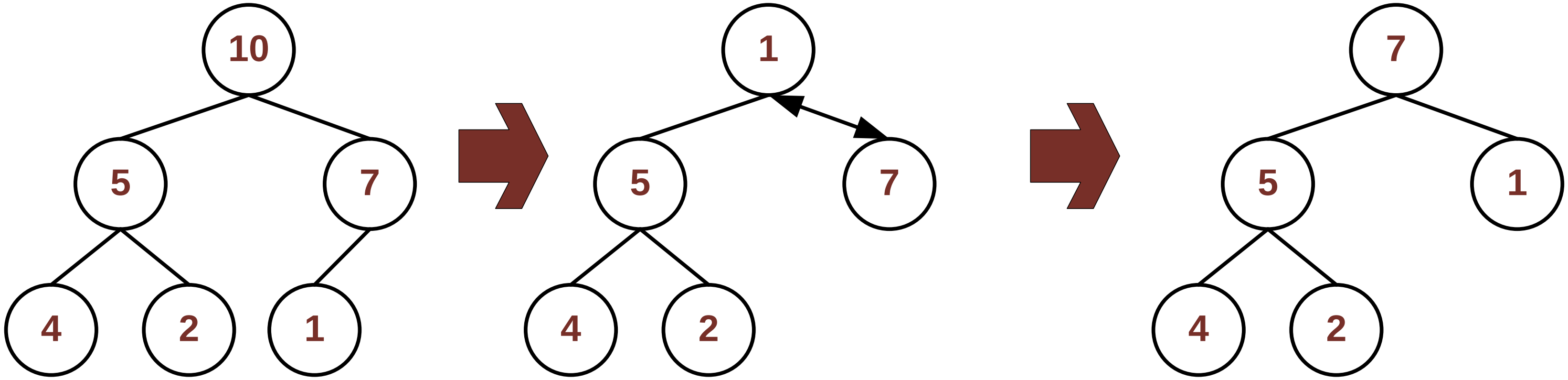
Runtime?

Heaps /6

- **Remove(Node): (always remove the root node)**
  - Replace the root node with the bottom-right-most leaf node in the tree
  - Switch the root node with the highest valued child
  - Continue recursing downwards until there is heap property compliance or until the bottom of the tree is reached

Heaps /7

- **Remove(10) example:**
  - Swap 10 and 1 (bottom-right-most leaf node)
  - Ensure heap property compliance in a top-down manner
  - Swap 1 and 7 then terminate when the bottom is reached



Runtime?

This shows power of partially ordered tree! (when we insert into bottom right)

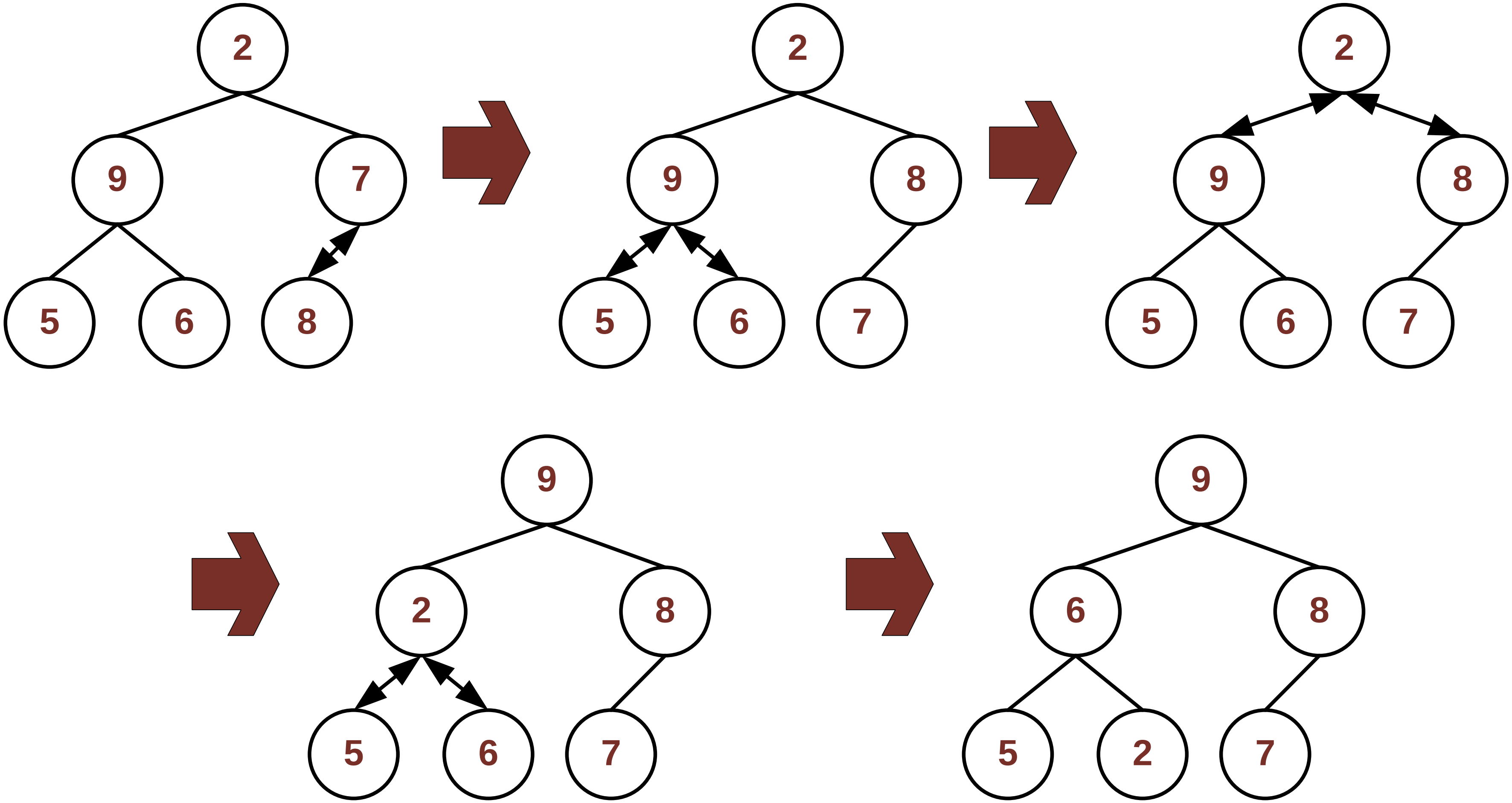
Heaps /8

- **Could insert item-by-item into a heap: O(nlogn)**
  - Why O(nlogn)?

- But there's a better way (O(n)):
- **Heapify(Tree):**
  1. Enumerate the internal nodes of the **existing tree** in reverse level order. That is, traverse all non-leaf and non-root nodes in using reverse level-based traversal
  2. In the order of the enumerated list, ensure heap property compliance of each node in the list in a top-down manner
  3. Ensure heap property compliance of the root node of the tree in a top-down manner, and then terminate

Heaps /9

- **Heapify(Tree) Example:**



Runtime?

Heaps /10

- A primary application of heaps are **priority queues**
  - Fundamental in CPU scheduling, where the OS determined which process to execute on the CPU.