CSI 2103: Data Structures

Trees (Ch 8)

Yonsei University

Spring 2022

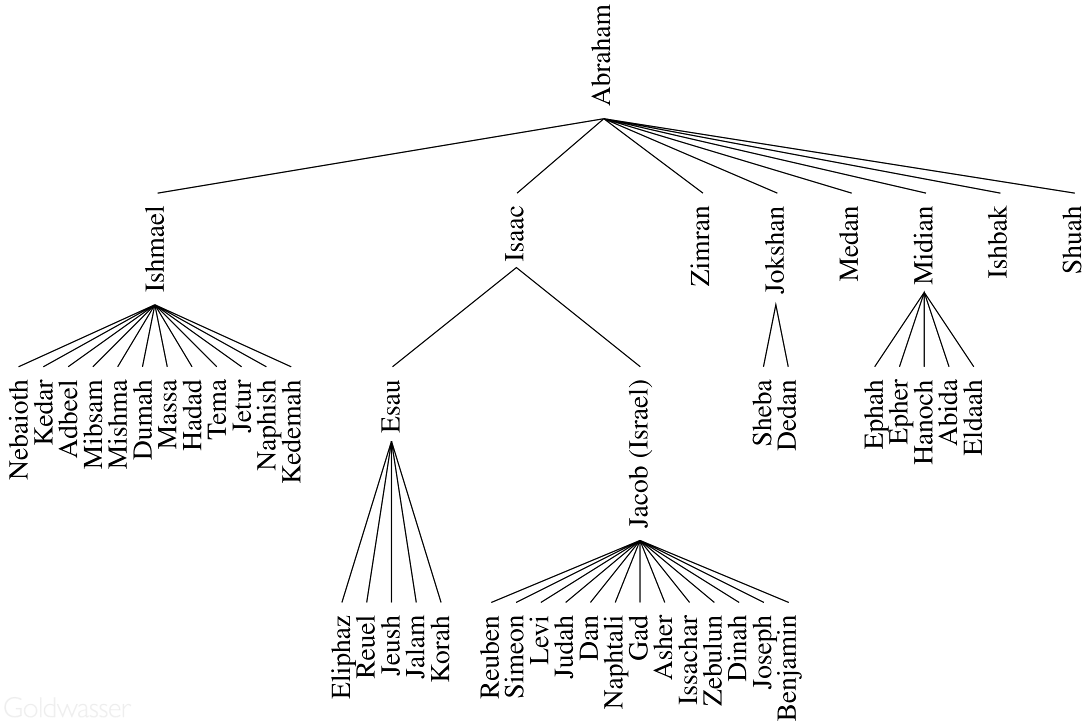
Seong Jae Hwang

Aims

- Tree: first nonlinear data structure we study
- Binary Tree
- How to implement trees
  - linked structure
  - array-based
- How to traverse through trees
- Application of binary tree for binary search

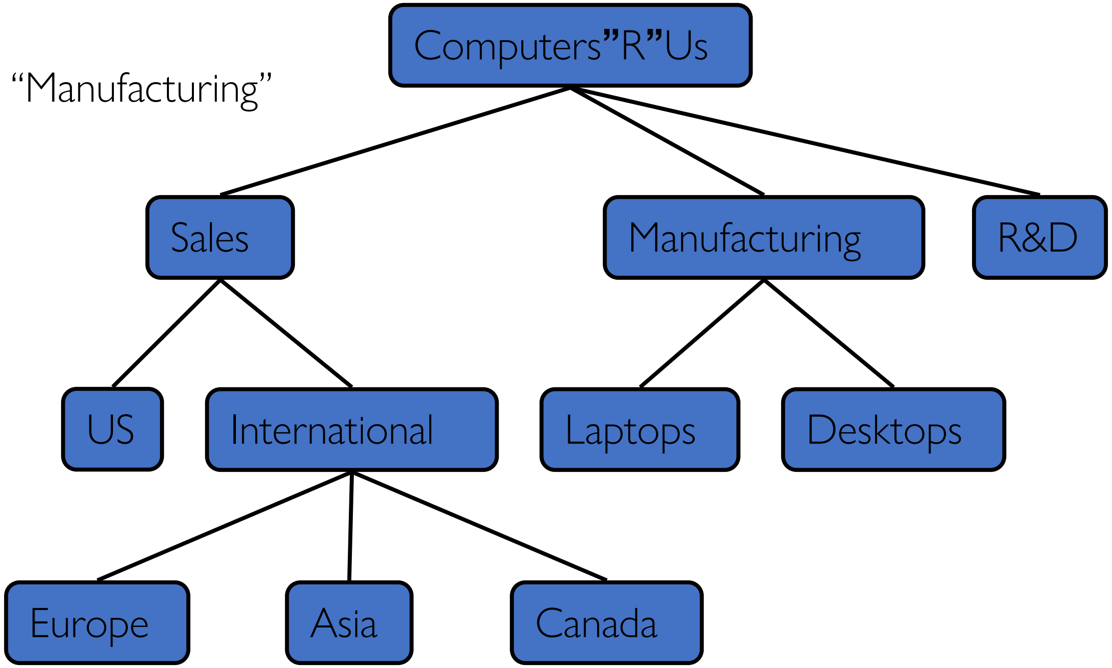
Trees!

- One of the most important nonlinear data structures
  - More than just “before” and “after” relationships
  - hierarchical relationships: “above” and “below” others
- Many algorithms become much faster with trees compared to linear data structures such as array or linked lists



Definitions and Properties

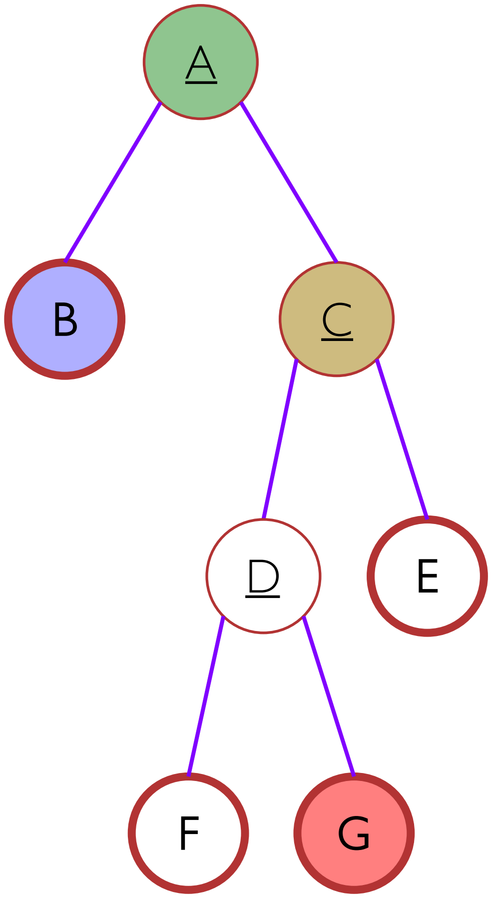
- ADT that stores elements hierarchically
- Each element (except the very top element) has a parent and zero or more children elements
  - root: top/highest element
- Notice that the elements at the same "level" are not necessarily of the same characteristics
  - ex: children of "Sales" and "Manufacturing"



Formal Tree Definition

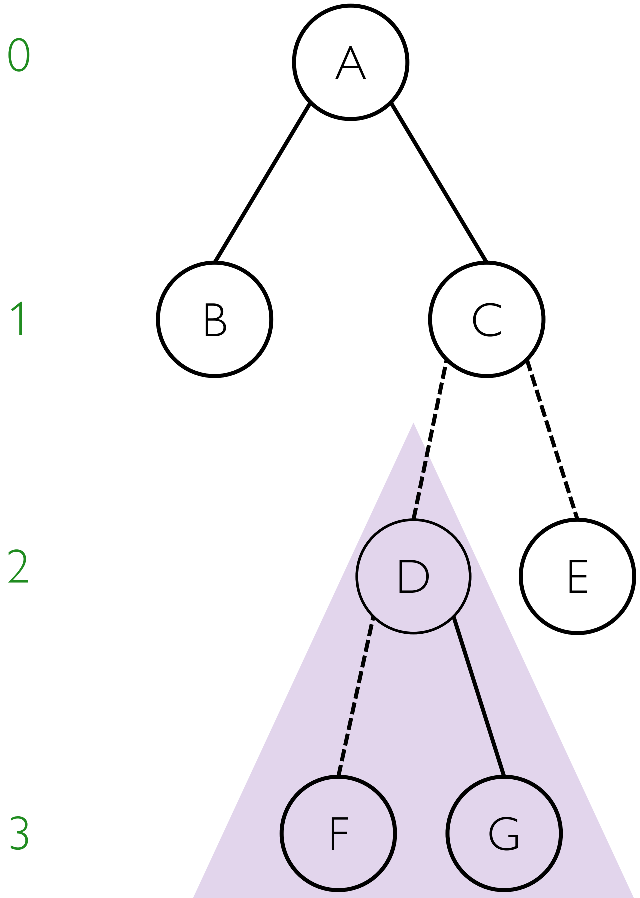
- Formally, a tree T is a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies:
  - If T is nonempty, it has the root of T that has no parent
  - Each node v of T (except for the root) has a unique parent node w; every node with parent w is a child of w
- Technically, a tree can be
  - of only one node (i.e., root) with (possibly empty) subtrees
  - or even empty

Tree Terminology



- *Nodes*
- *Edges* connect two nodes
- A is the *root* of the tree
- B is a *child* of A
- A is the *parent* of B
- C is a *sibling* of B
- G is a *descendant* of C
- C is an *ancestor* of G
- ***Leaves*** do not have children; *internal nodes* do

Tree Terminology



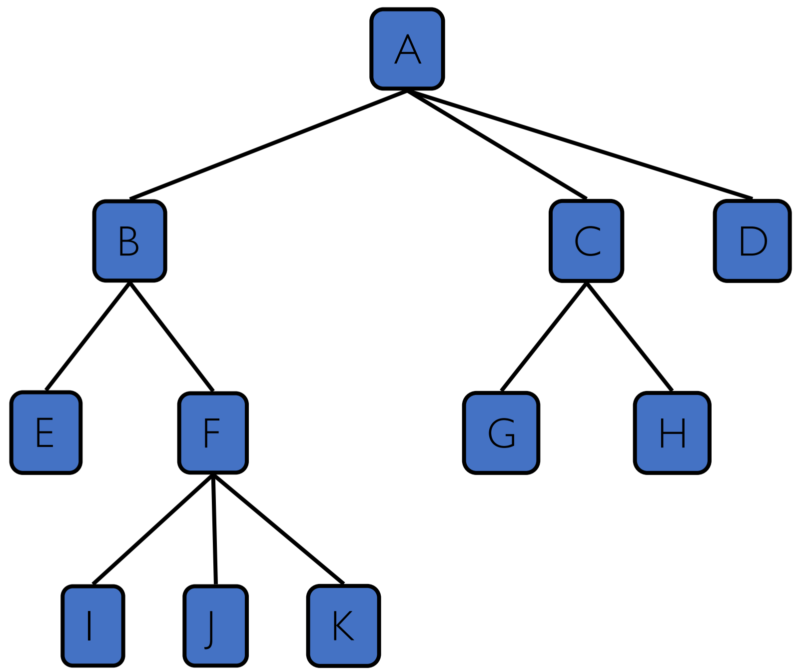
- The *subtree rooted at* D contains 3 nodes
- The *degree* of a node is the number of its children
- A *path* connects two nodes;

  there exists a unique (simple)

  path between any pair of nodes
- The *level** of a node is the length of the unique path from the root to the vertex
- The *height* of a tree is the maximum level of a node

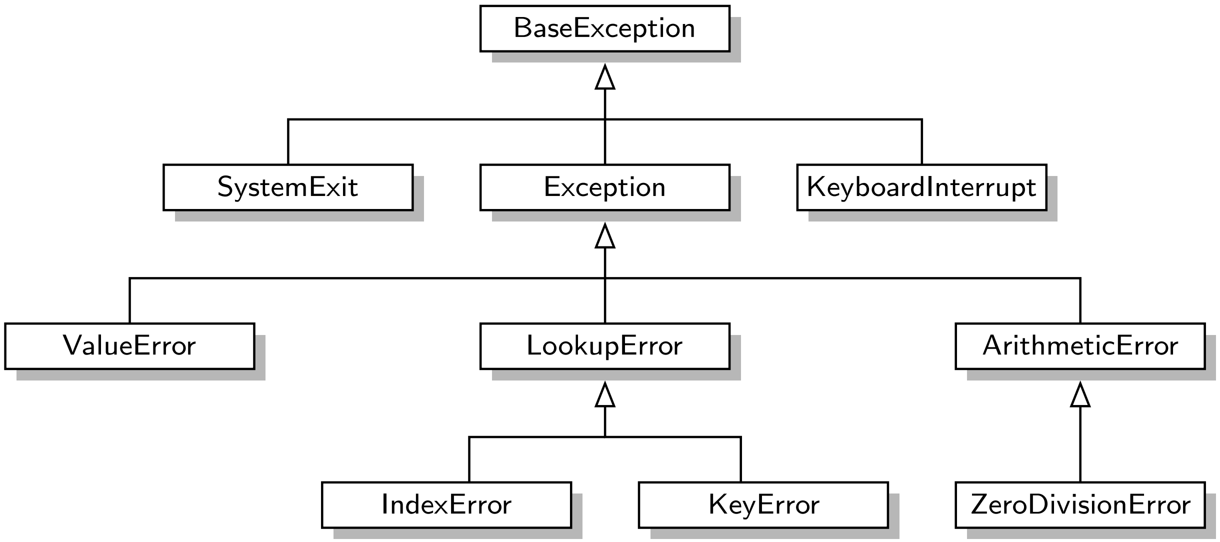
Example

- *Root*: A
- *Internal node*: A, B, C, F
- *Leaves*: E, I, J, K, G, H, D
- *Ancestors of K*: F, B, A
- *Level of J*: 3
- *Level of C*: 1
- *Height*: 3
- *Descendants of B*: F, I, J, K
- *One subtree:* F and its children
- *Path from J to G*: J, F, B, A, C, G



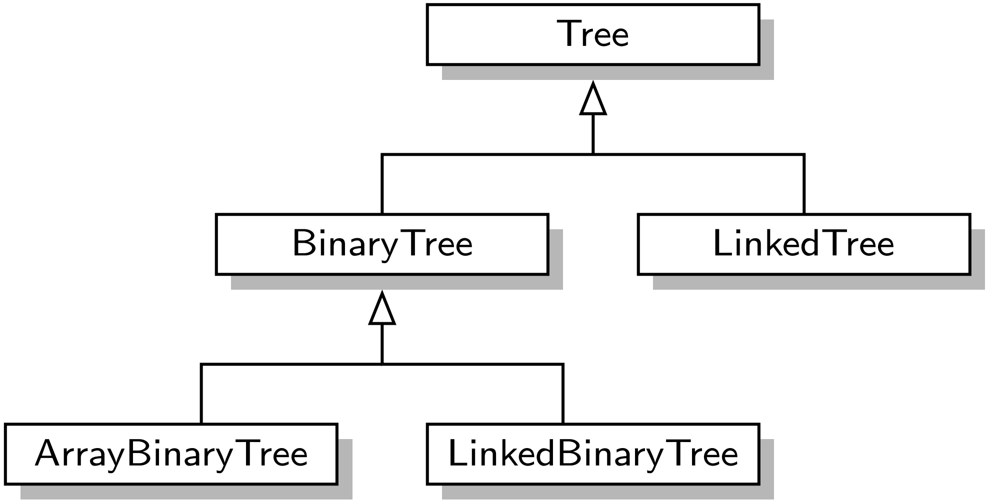
Example

- The hierarchy for Python's exception types
- *BaseException* is the root
- All user-defined exception classes conventionally declared as descendants of *Exception* class



Example

- The inheritance hierarchy for various tree data structures we will cover in this chapter
  - BinaryTree is a type of Tree
  - ArrayBinaryTree is a type of BinaryTree

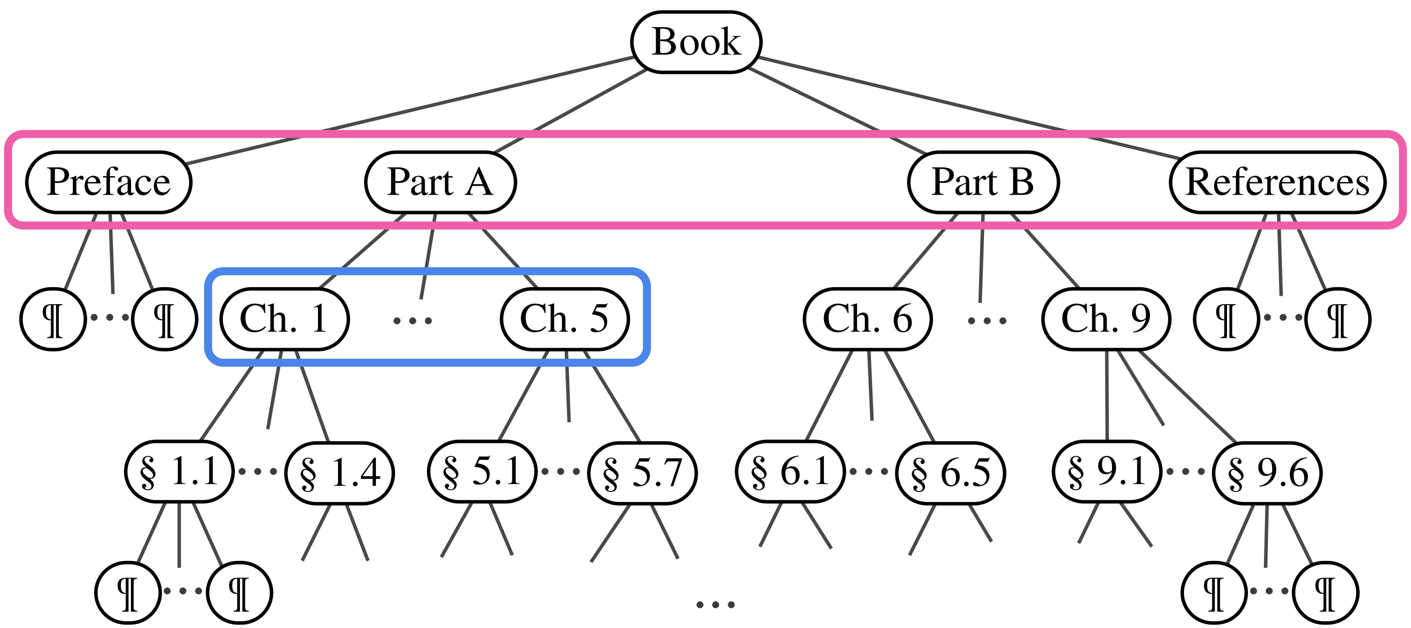


Quick Question: what *is* data structure?

- Let us realize that data structures are not just ways to use large datasets efficiently in terms of speed and space
- Information to be store often have inherent relationships
- Appropriate data structures help us to capture those relationships
  - Not only for efficiency
  - But also for understanding the underlying relationship of our data

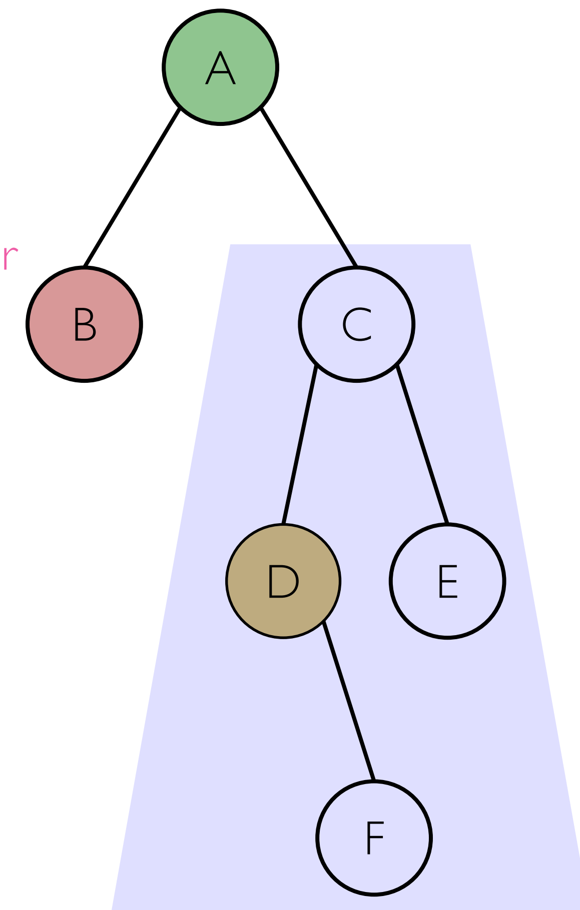
Can We "Order" Trees?

- A tree is ordered if there is a meaningful linear order among the children of each node
  - Visually, ordered from left to right
- Hierarchical *and* linear relationships



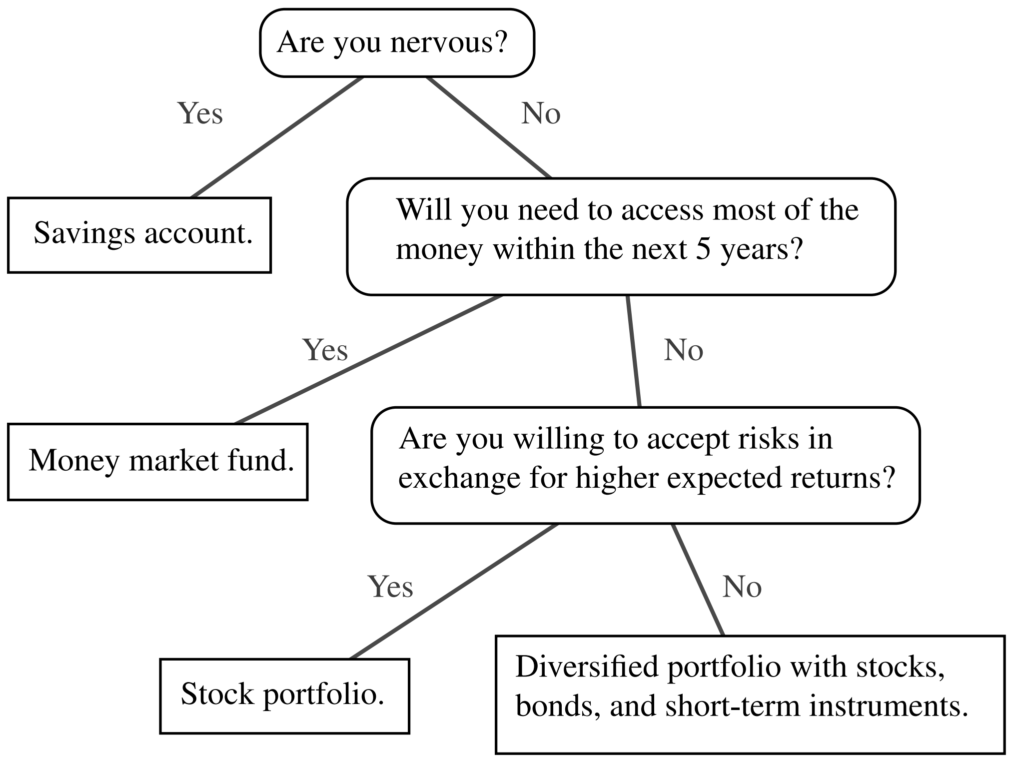
Binary Trees

- A binary tree is an ordered tree which
  - Every node has at most two children
    - When all nodes have exactly zero or two children, the binary tree is proper
    - If not proper, then the binary tree is improper
    - a left child or a right child
  - The children of a node are ordered
- B is the left child of A
- The subtree rooted at C is the right subtree of A
- D has the right child only



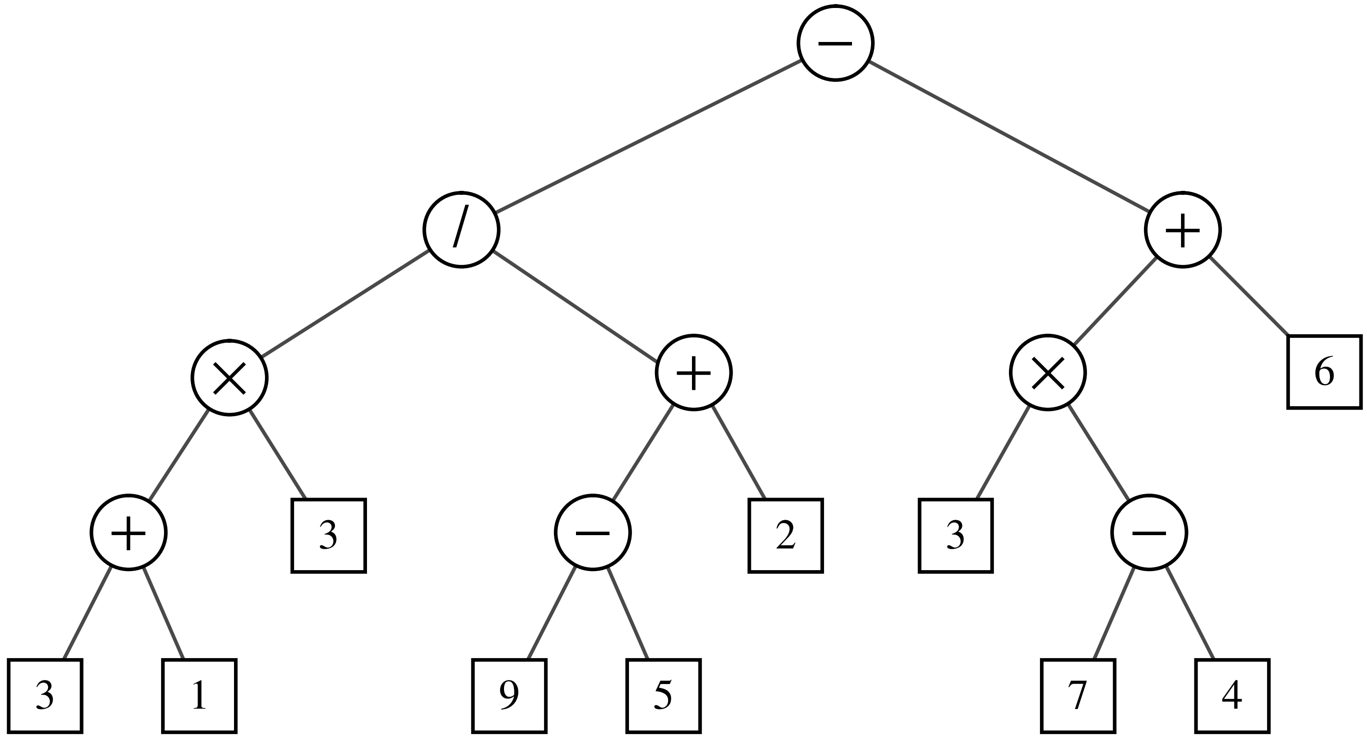
Example

- Decision trees: ask questions and take a step depending on your answer
  - Yes: go to the left child
  - No: go to the right child



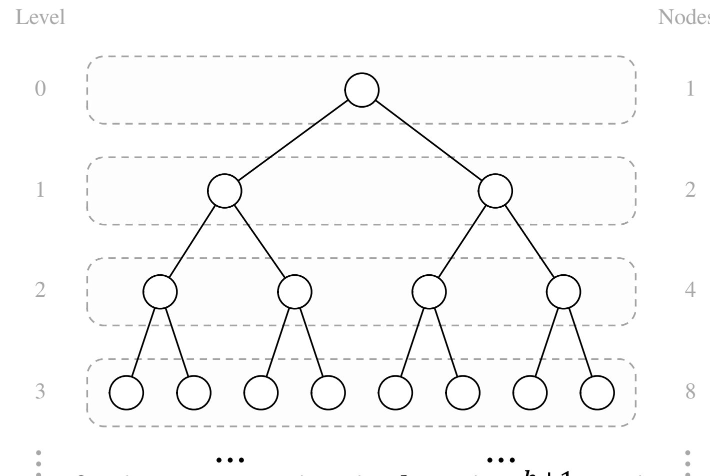
Example

- Arithmetic expression
  - Internal nodes: operators (+, -, x, /)
  - Leaves (external nodes): variables or constants
- $((((3 + 1) \times 3) / (9 - 5) + 2)) - ((3 \times (7 - 4)) + 6))$



Properties of Binary Trees

- Because of such strict structures, binary trees have interesting properties based on
  - n: # of nodes
  - n_E: # of external nodes (leaves)
  - n_I: # of internal nodes
  - h: height

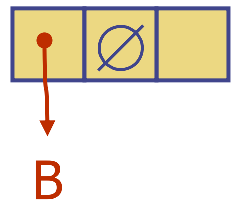


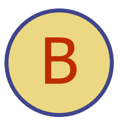
full binary tree: height h with 2^{h+1} nodes

- For a nonempty binary tree T,
  - $h + 1 \le n \le 2^{h+1} - 1$
    - $h + 1$: T with only one child
    - $2^{h+1} - 1$: max possible # of nodes of a binary tree with height h
  - $1 \le n_E \le 2^h$
    - 1: T with root
    - 2^h : max possible # of leaves at level h
  - $h \le n_I \le 2^h - 1$
    - h: T with only one child (note h starts from 0)
    - $2^h - 1$: max possible # of nodes except for level h
  - $\log(n + 1) - 1 \le h \le n - 1$
    - $\log(n + 1) - 1$: height of a full binary tree built from n nodes
    - $n - 1$: height of a one-child binary tree built from n nodes

Implementing Trees

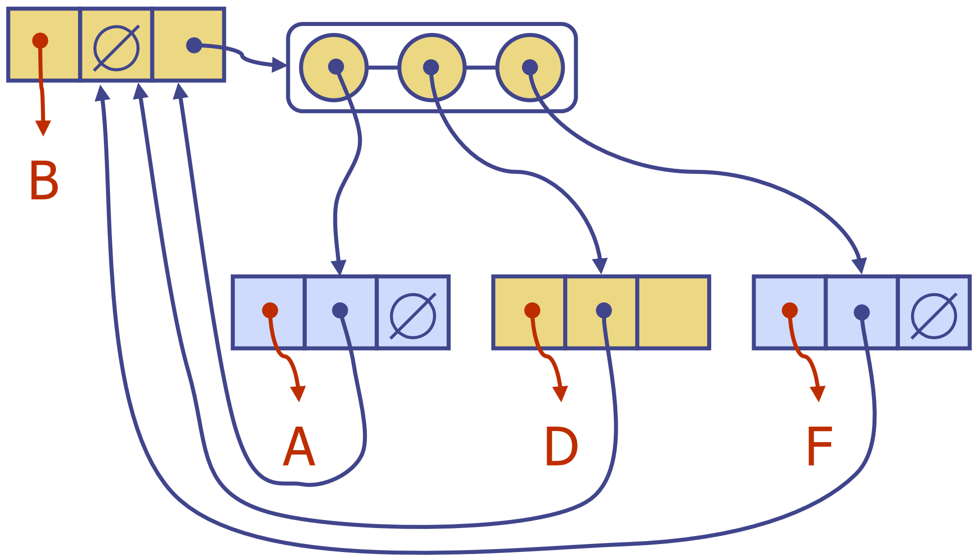
- Linked Structures
- A node as an object storing
  - element
  - parent node
  - sequence of children nodes

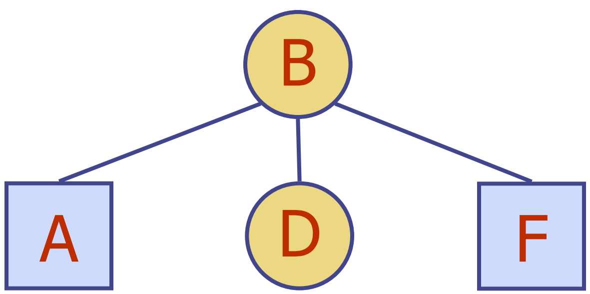




Implementing Trees

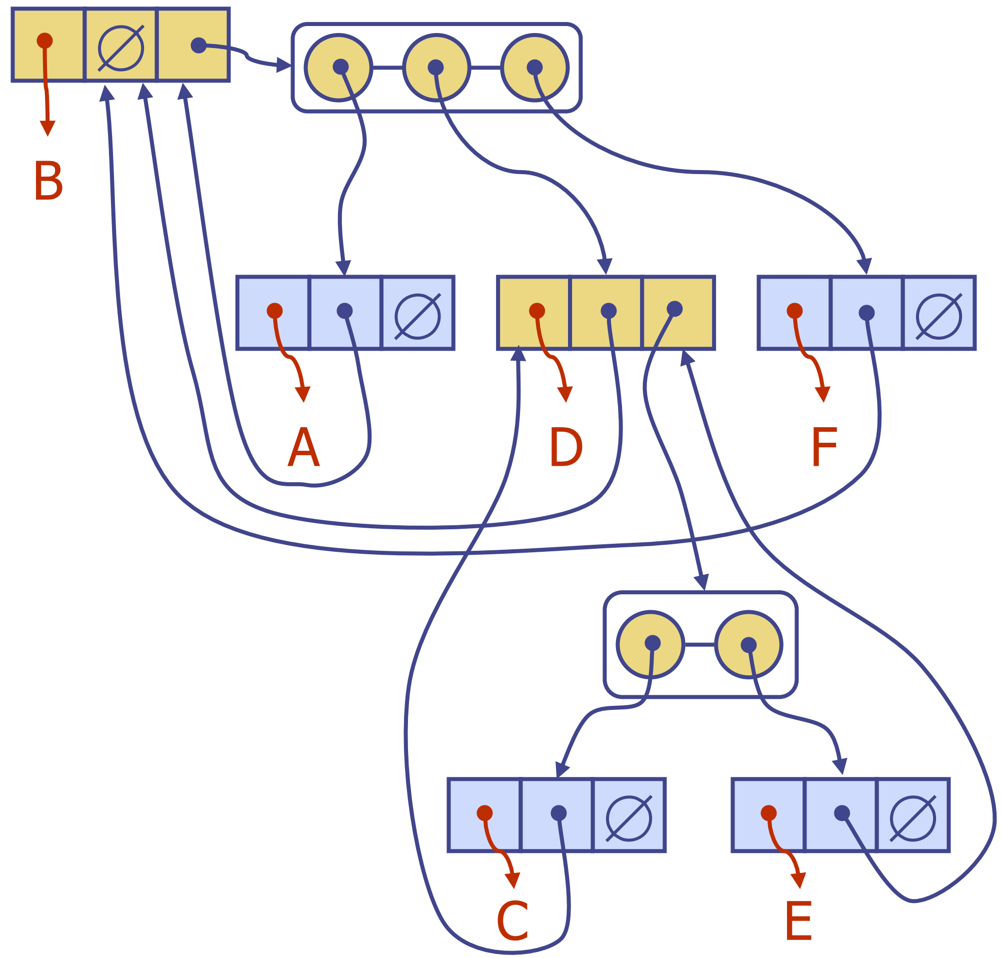
- Linked Structures
- A node as an object storing
  - element
  - parent node
  - sequence of children nodes

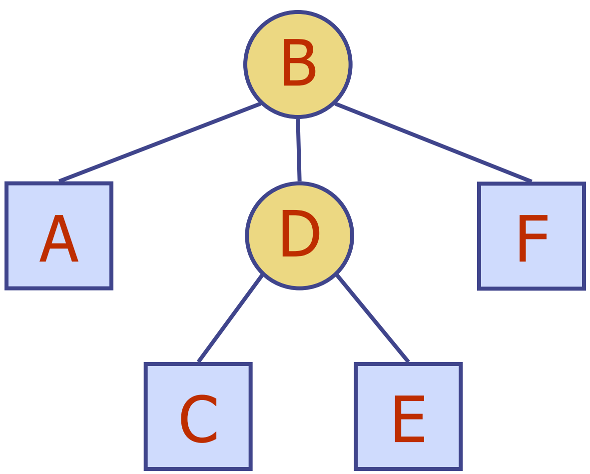




Implementing Trees

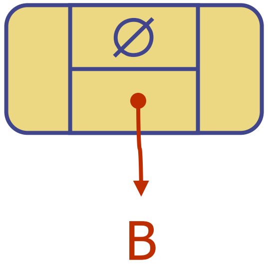
- Linked Structures
- A node as an object storing
  - element
  - parent node
  - sequence of children nodes

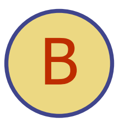




Implementing Binary Trees

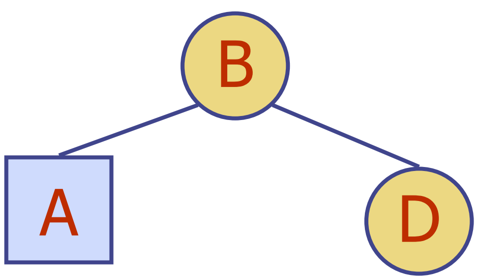
- Again, Linked Structures
- A node as an object storing
  - element
  - parent node
  - left child node
  - right child node

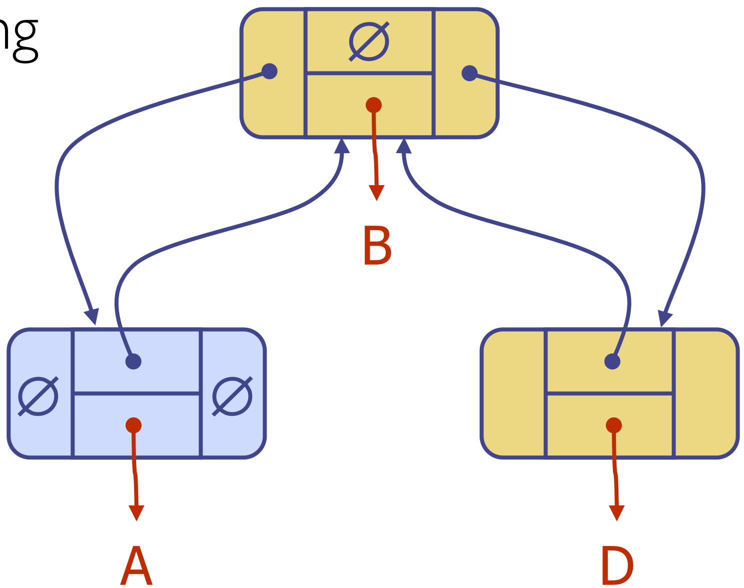




Implementing Binary Trees

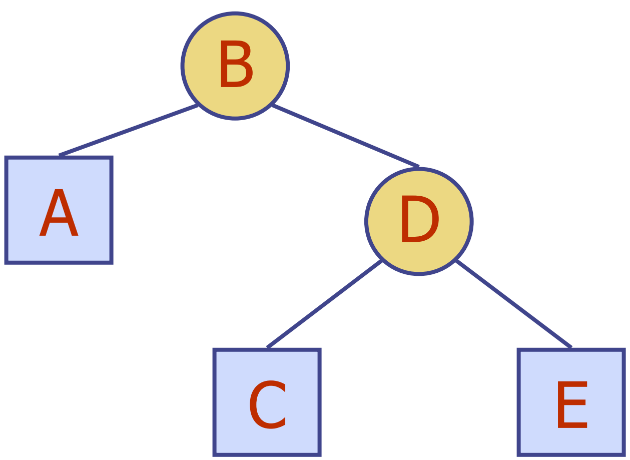
- Again, Linked Structures
- A node as an object storing
  - element
  - parent node
  - left child node
  - right child node

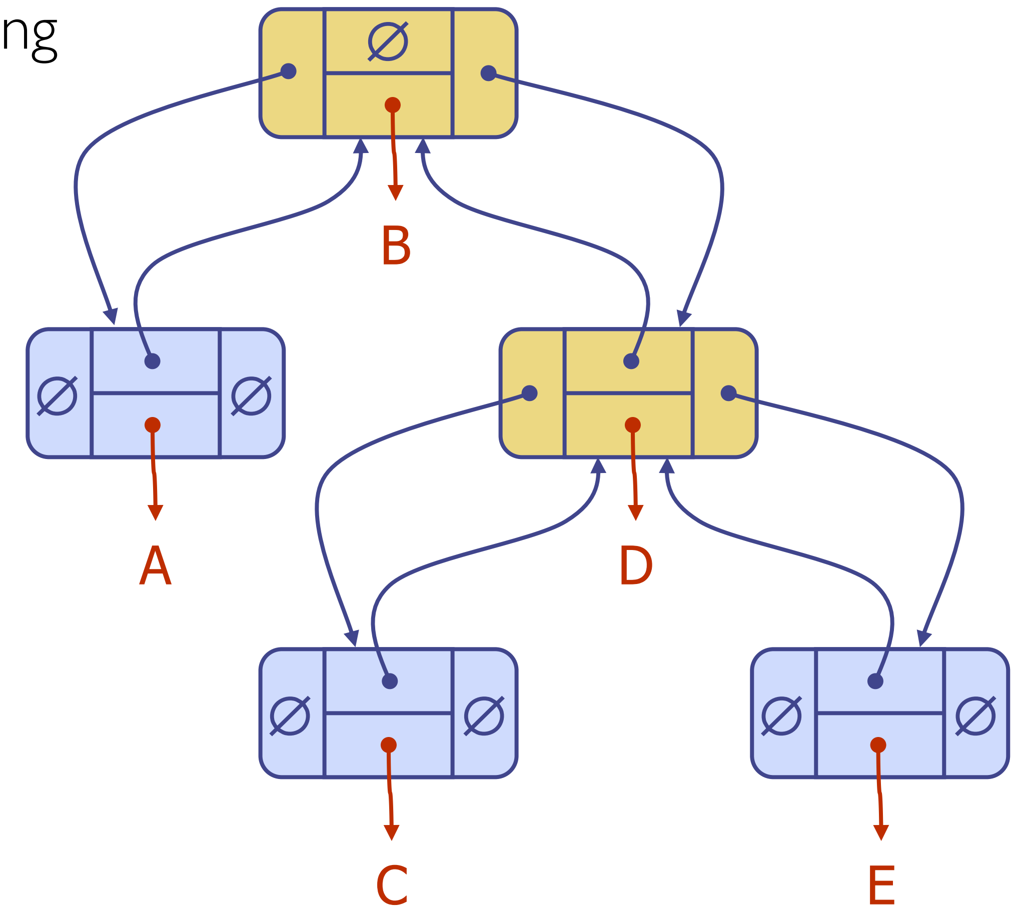




Implementing Binary Trees

- Again, Linked Structures
- A node as an object storing
  - element
  - parent node
  - left child node
  - right child node





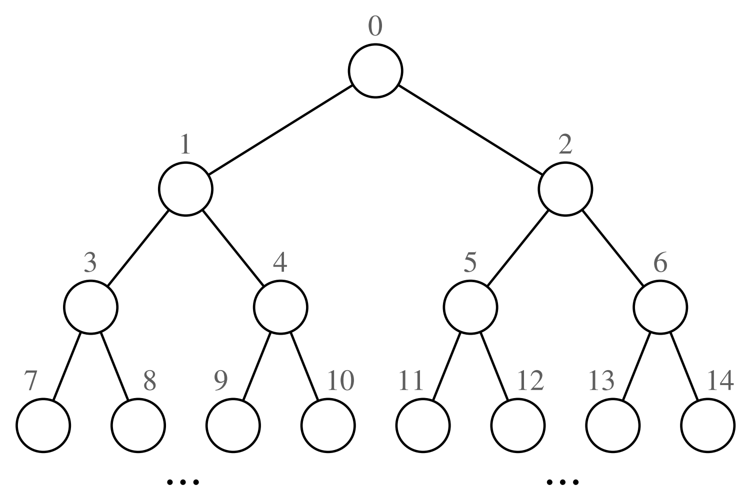
Time Complexity of Binary Tree

- Linked binary tree
- Most operations require a constant number of node relinking
- height needs to check all nodes to find the maximum depth

Operation	Running Time
len, is_empty	$O(1)$
root, parent, left, right, sibling, children, num_children	$O(1)$
is_root, is_leaf	$O(1)$
depth(p)	$O(d_p + 1)$
height	$O(n)$
add_root, add_left, add_right, replace, delete, attach	$O(1)$

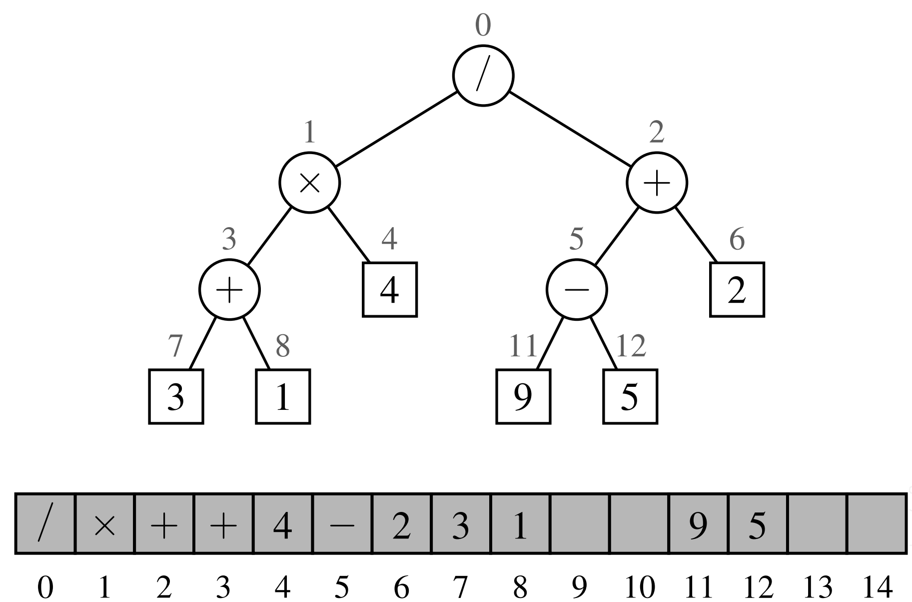
Implementing Binary Trees

- Array-Based
  - Can we actually do this with a simple array?
  - Yes! Assign an index to each node based on its level: level numbering
- For every position p of T, the index/rank $f(p)$ is
  - If p is the root of T, then $f(p) = 0$
  - If p is the left child of position q, then $f(p) = 2f(q) + 1$
  - If p is the right child of position q, then $f(p) = 2f(q) + 2$
- Parent of p? (what is $f(q)$ given p?)
  - left child: $f(q) = \frac{f(p)-1}{2}$
  - right child: $f(q) = \frac{f(p)-2}{2}$
  - left or right child: $f(q) = \left\lfloor \frac{f(p)-1}{2} \right\rfloor$



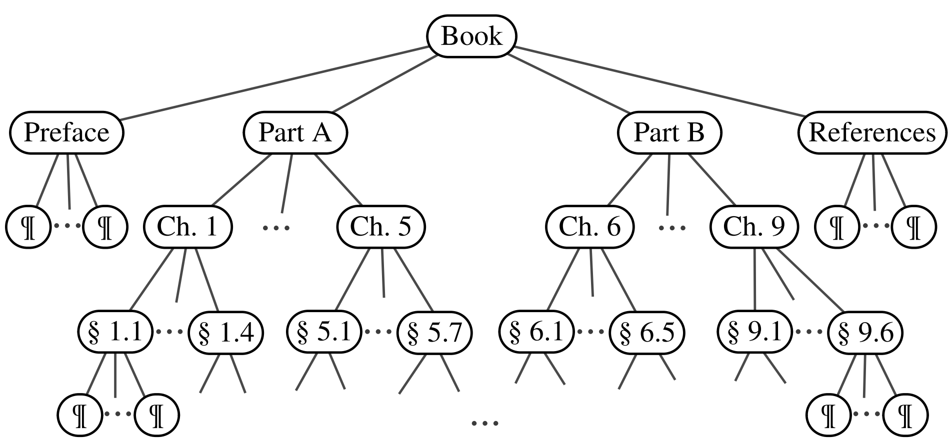
Implementing Binary Trees

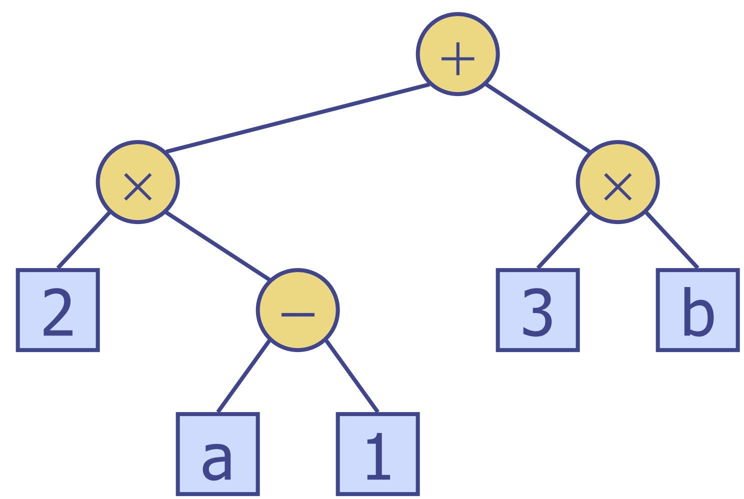
- Indices are based on full binary tree
  regardless of your tree shape
- Advantage:
  - position (element) p can be expressed by a single integer
  - parent, left, and right of p can all be arithmetically computed
- Disadvantage:
  - Size of the array depends on the max f(p)
    - what is the most extreme example?
      right-child-only tree
    - For each new level, how much does the array size grow? doubles!
  - Updating (add or delete) a node is cannot be done efficiently
    - what is the time complexity? O(n)



In what order do we access the nodes?

- From a node, we have two options: visit the sibling, or visit the children
- How do we systematically or algorithmically access the nodes such that
  - a book is structured in the correct order?
  - returning the node elements gives (2 × (a - 1) + (3 × b)) ?
- Depends on the application!

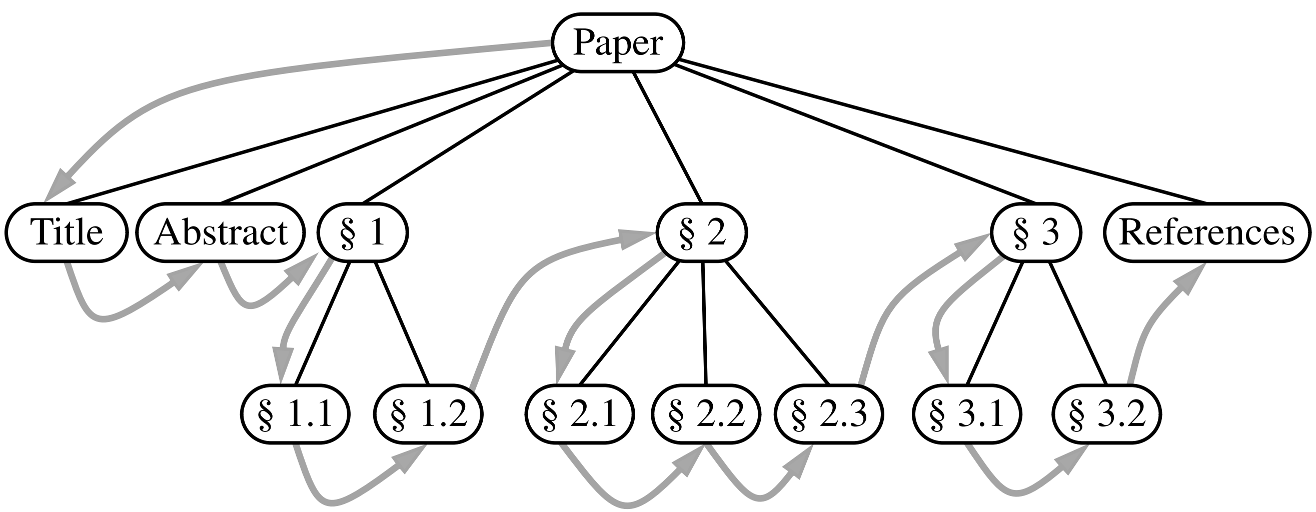




Tree Traversal 1: Preorder Traversal

- In a preorder traversal, a node is visited *before* its descendants

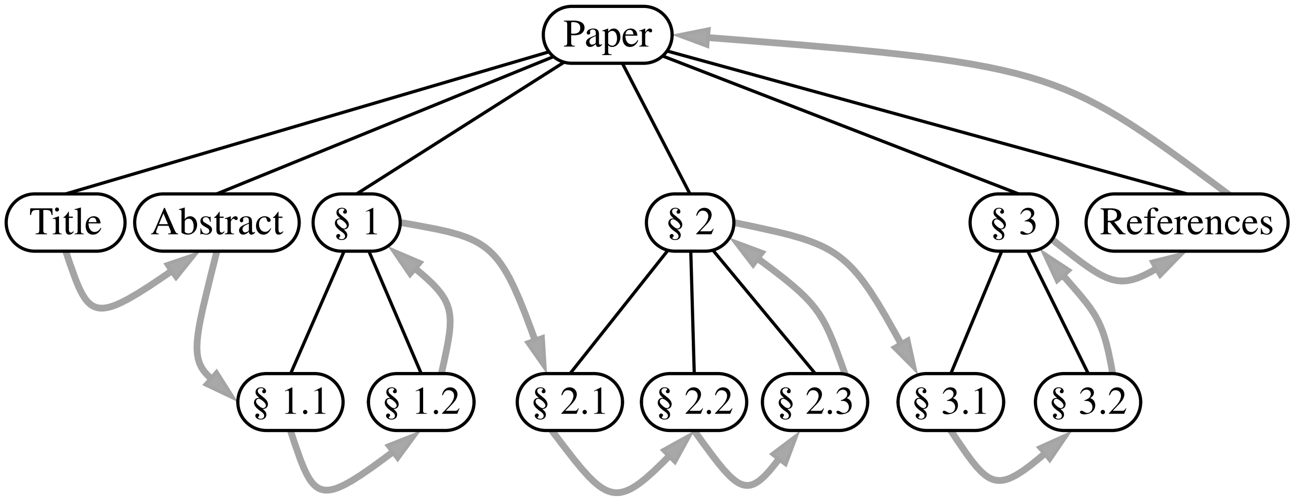
Algorithm preorder(T, p):
  perform the "visit" action for position p
  for each child c in T.children(p) **do**
    preorder(T, c)   {recursively traverse the subtree rooted at c}



Tree Traversal 2: Postorder Traversal

- In a postorder traversal, a node is visited *after* its descendants

Algorithm postorder(T, p):
    for each child c in T.children(p) **do**
        postorder(T, c) {recursively traverse the subtree rooted at c}
    perform the "visit" action for position p

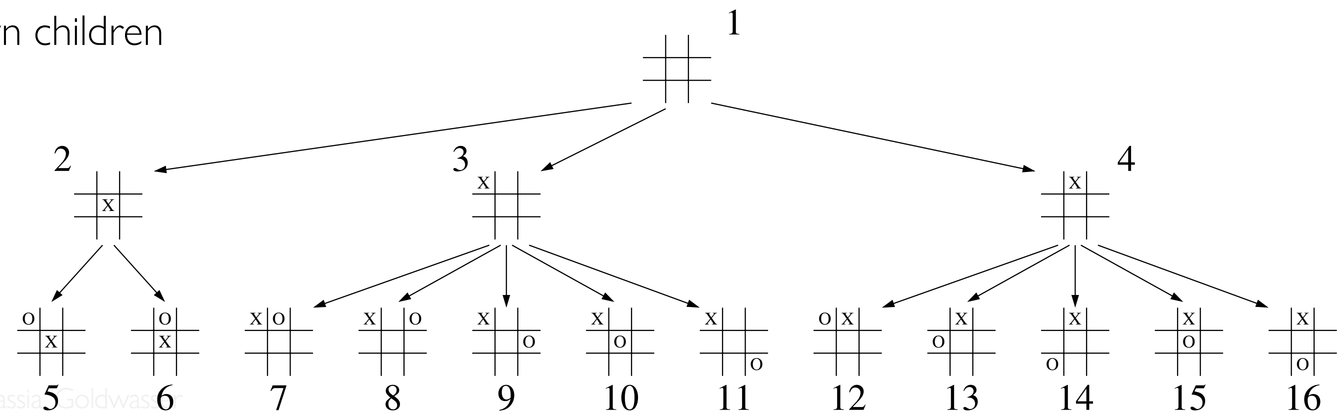


Breath-First Tree Traversal

- Preorder and postorder traversals are recursive
- Breath-First Search: visit all the nodes in each level before checking the nodes at the next level
  - Example: a game tree to check all the moves possible by a player
    - Check all possible moves for the next h moves (where h is as much as the computer can compute)
- How do we implement this non-recursive traversal? Use queue!

```
Algorithm breadthfirst(T):
  Initialize queue Q to contain T.root( )
  while Q not empty do
    p = Q.dequeue( )                              {p is the oldest entry in the queue}
    perform the "visit" action for position p
    for each child c in T.children(p) do
      Q.enqueue(c)      {add p's children to the end of the queue for later visits}
```

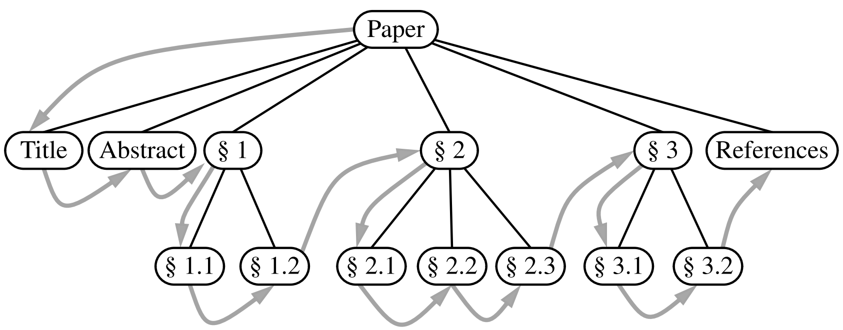
enqueue the children to ensure they are visited before their own children



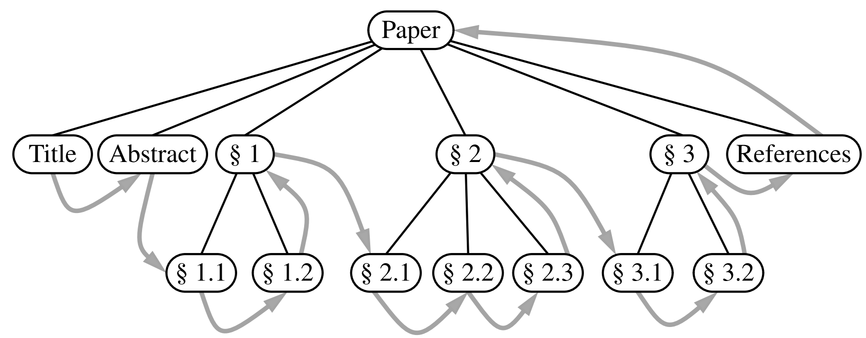
Example

- Preorder (left) or postorder (right) to structure the document?
  - preorder – parent node element needs to appear before its children

```
Paper
Title
Abstract
§1
§1.1
§1.2
§2
§2.1
...
```

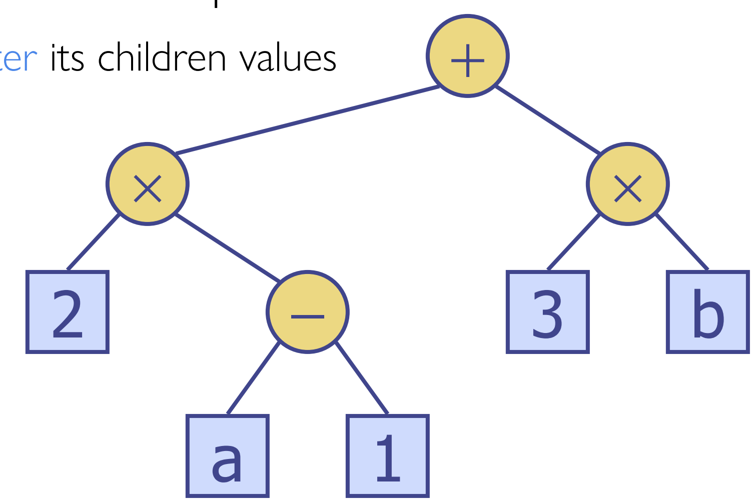


preorder



postorder

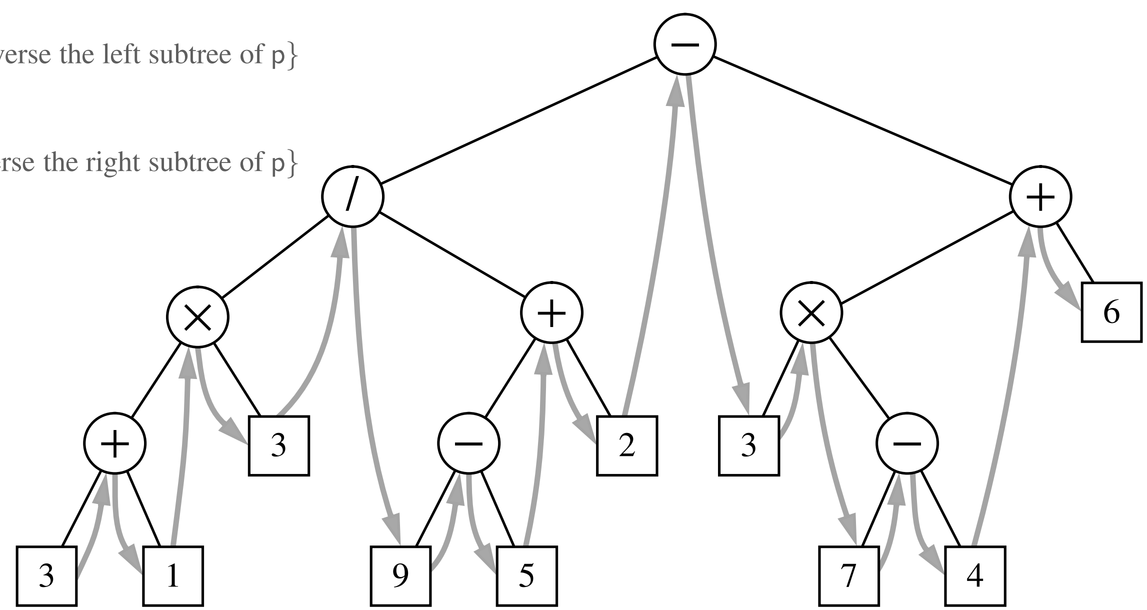
- Preorder or postorder to compute the arithmetic expression tree
  - postorder – parent node operator is applied after its children values
- How about to print the arithmetic tree?



Binary Tree Traversal: Inorder Traversal

- For binary tree:
  - Preorder: root -> left subtree –> right subtree
  - Postorder: left subtree -> right subtree -> root
  - Inorder: left subtree -> root -> right subtree
- 3 + 1 x 3 / 9 – 5 + 2 – 3 x 7 – 4 + 6
  - Missing the parenthesis

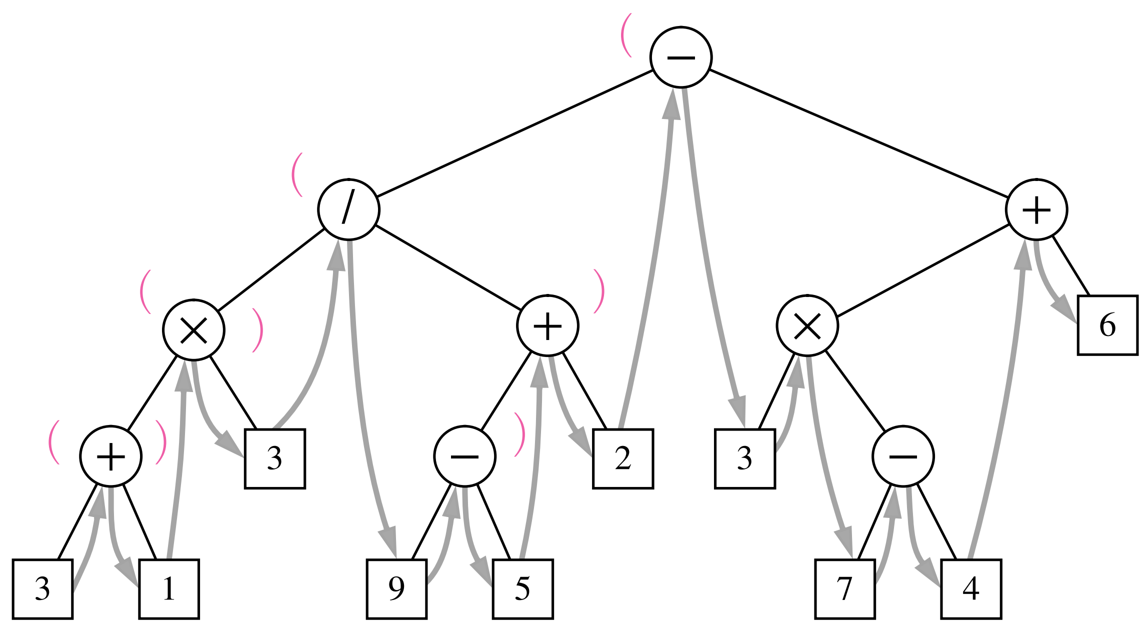
```
Algorithm inorder(p):
  if p has a left child lc then
    inorder(lc)                {recursively traverse the left subtree of p}
  perform the "visit" action for position p
  if p has a right child rc then
    inorder(rc)                {recursively traverse the right subtree of p}
```



Example

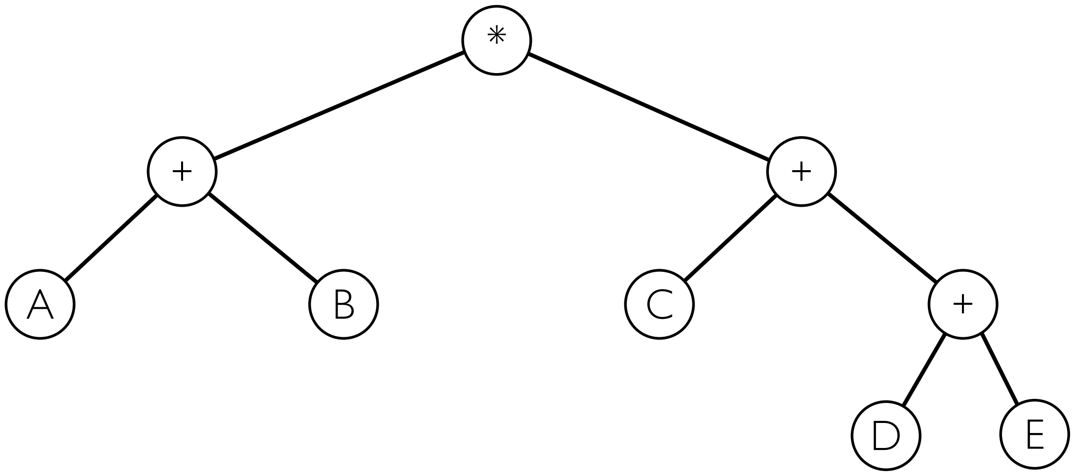
- Print arithmetic expressions
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree
- ((((3 + 1) × 3) / ((9 – 5) + 2)) – …

```
Algorithm printExpression(v)
    if left(v) ≠ null
        print("(")
        inorder (left(v))
    print(v.element ())
    if right(v) ≠ null
        inorder (right(v))
        print (")")
```



Example

- If we simply “output” the data of each node…



- (A+B)*(C+(D+E))
- Preorder traversal: *+AB+C+DE
- Inorder traversal: A+B*C+D+E
- Postorder traversal: AB+CDE++*

Consider a Binary Search

10

3	9	10	13	19	20	23

Consider a Binary Search

10

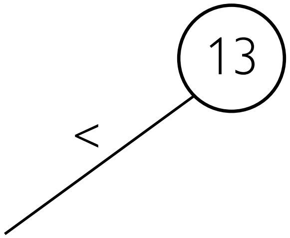
3	9	10	13	19	20	23

13

Consider a Binary Search

10

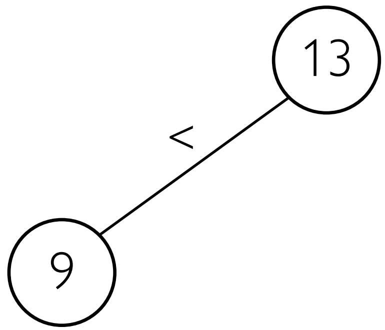
3	9	10	13	19	20	23



Consider a Binary Search

10

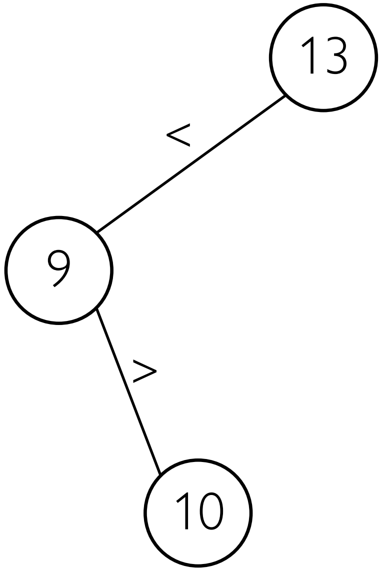
3	9	10	13	19	20	23



Consider a Binary Search

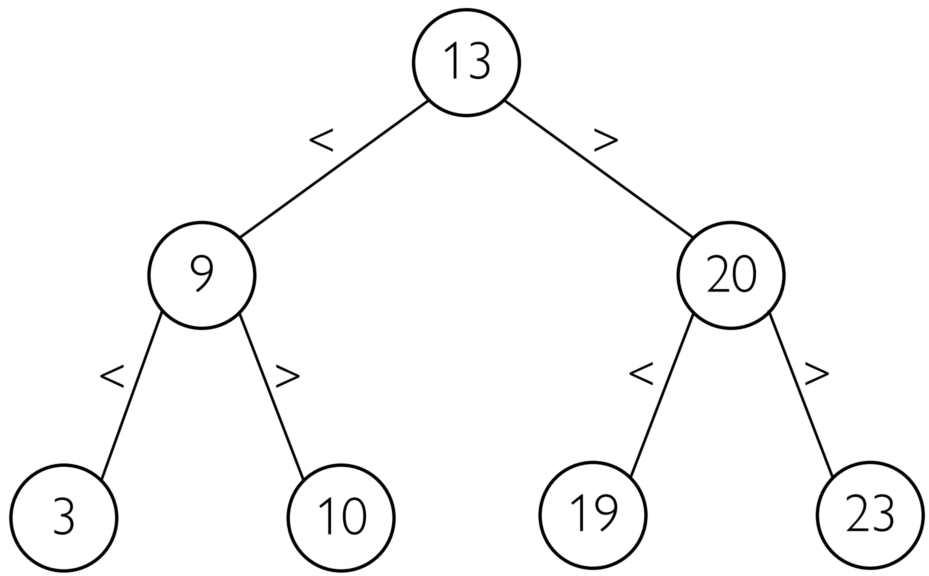
10

3	9	10	13	19	20	23



Consider a Binary Search

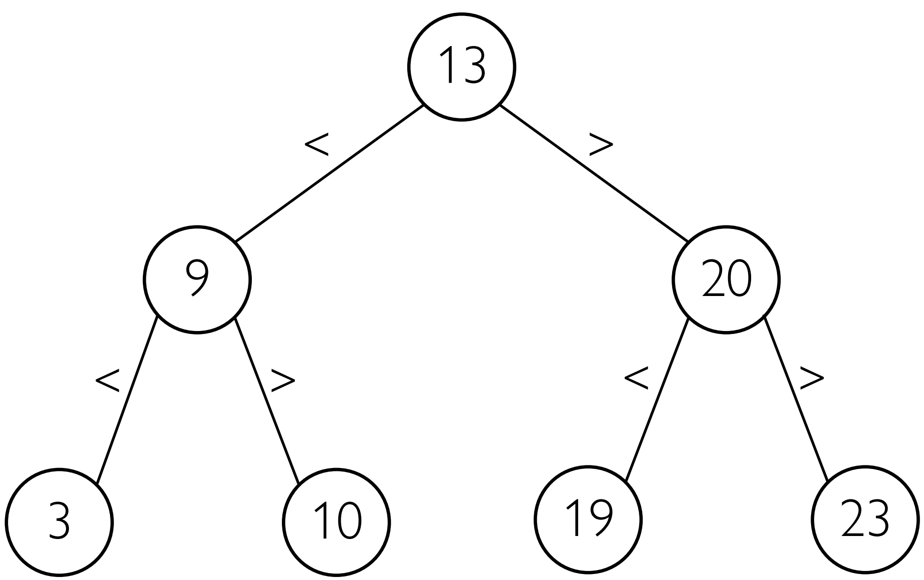
3 9 10 13 19 20 23



The entire binary search can be stored and processed in a binary tree

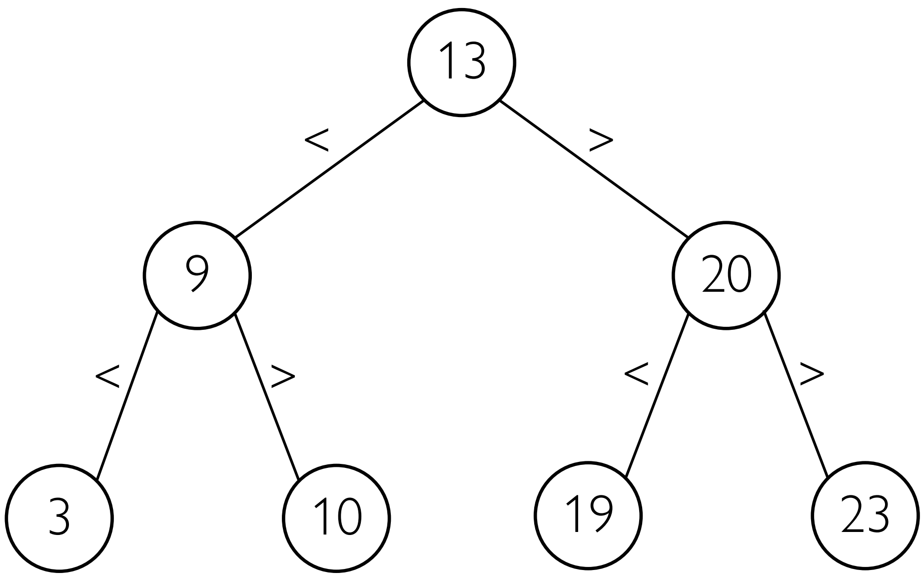
Binary Search Tree

- A binary search tree
  - an empty tree or
  - a binary tree such that
    - root has an element
    - left subtree elements are smaller than the root element
    - right subtree elements are greater than the root element
- Inorder traversal of a BST gives an ordered sequence of elements
- Why use this over a sorted array?



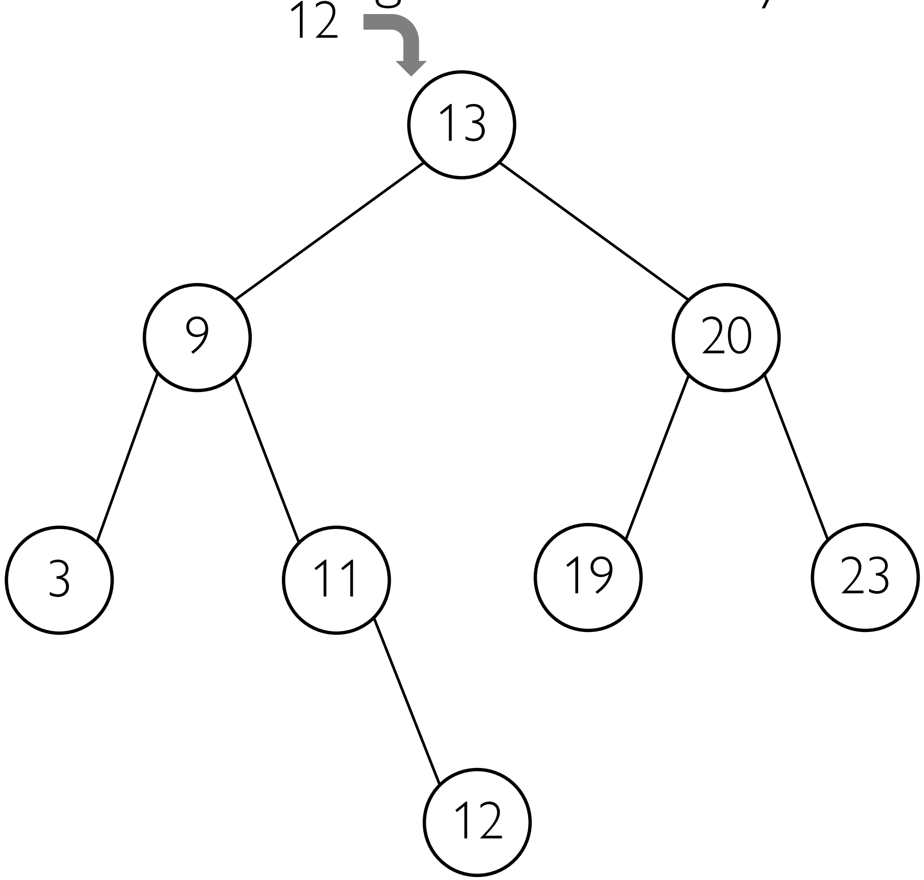
Binary Search Tree

- Insertion:
  - Array: $O(n)$ to find and insert
  - BST: ?



Binary Search Tree

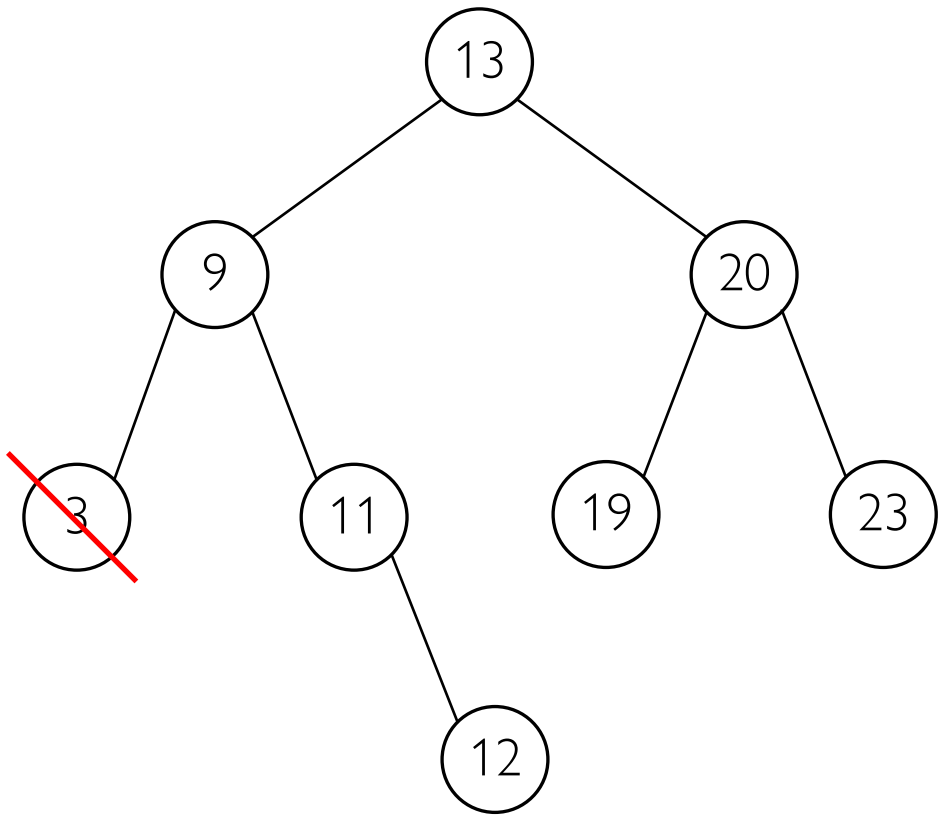
- $O(h)$ time, where h is the height of the binary search tree



Binary Search Tree

- Deleting a leaf

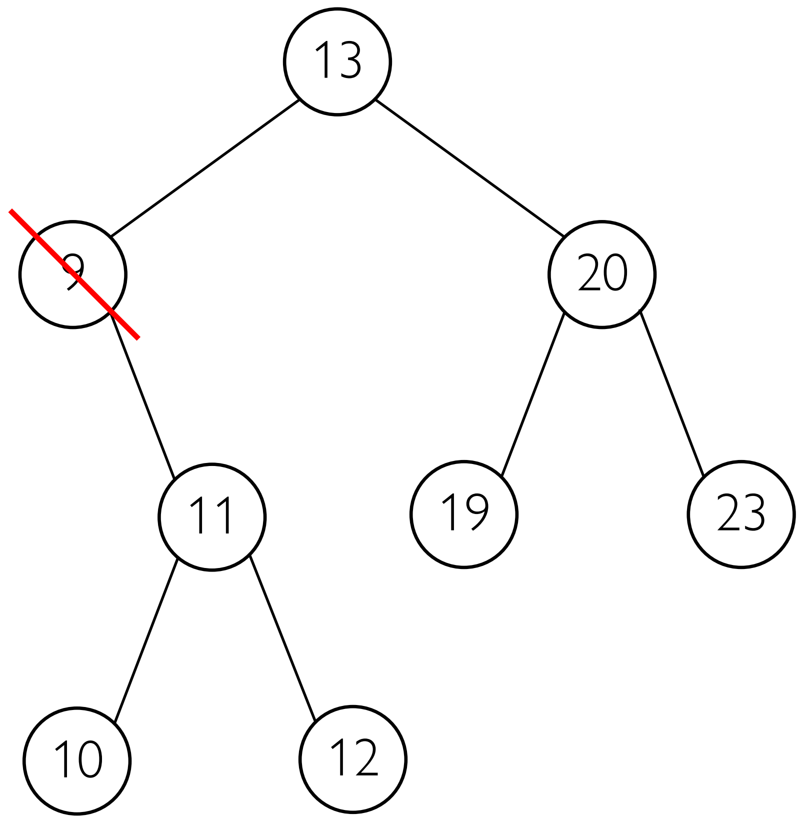
3



Binary Search Tree

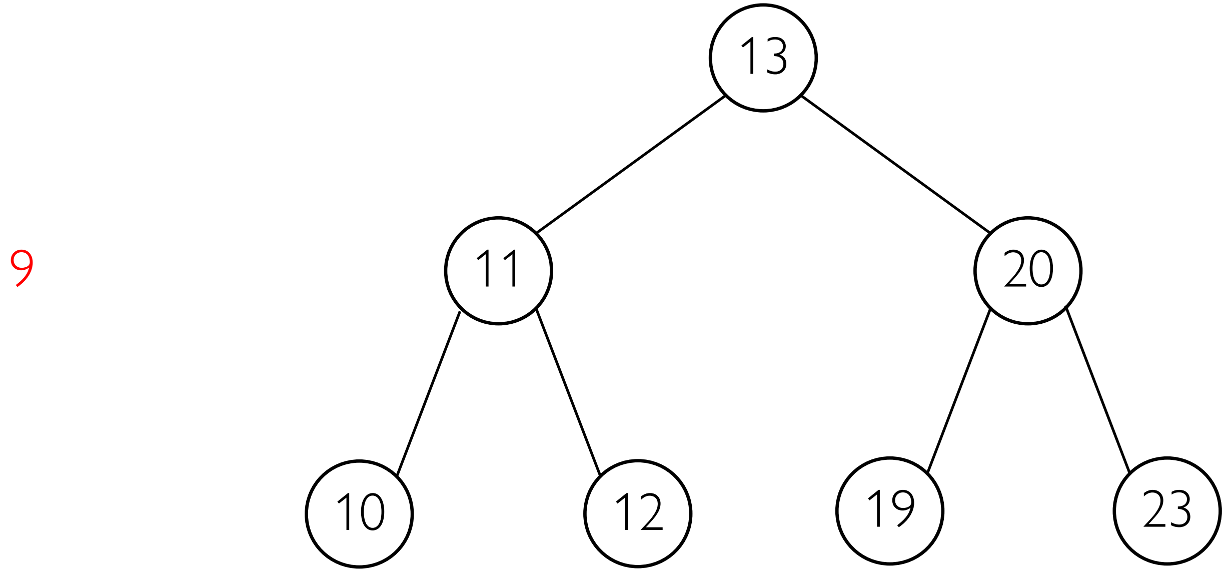
- Deleting a node with a single child

9



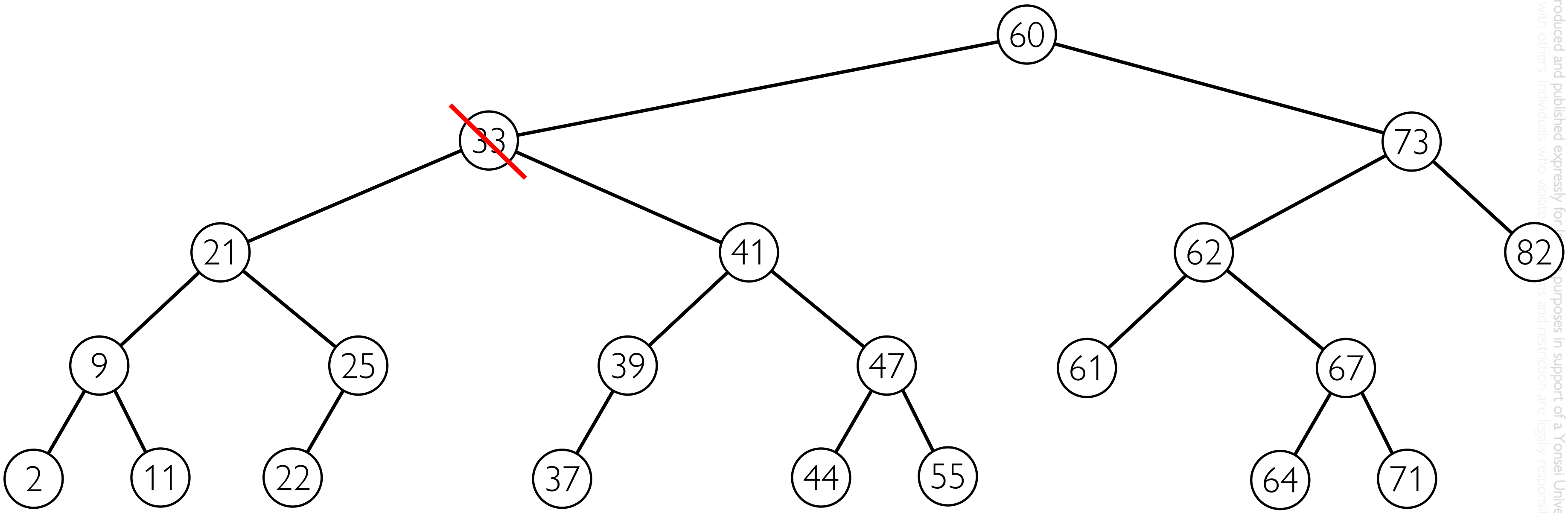
Binary Search Tree

- Deleting a node with a single child



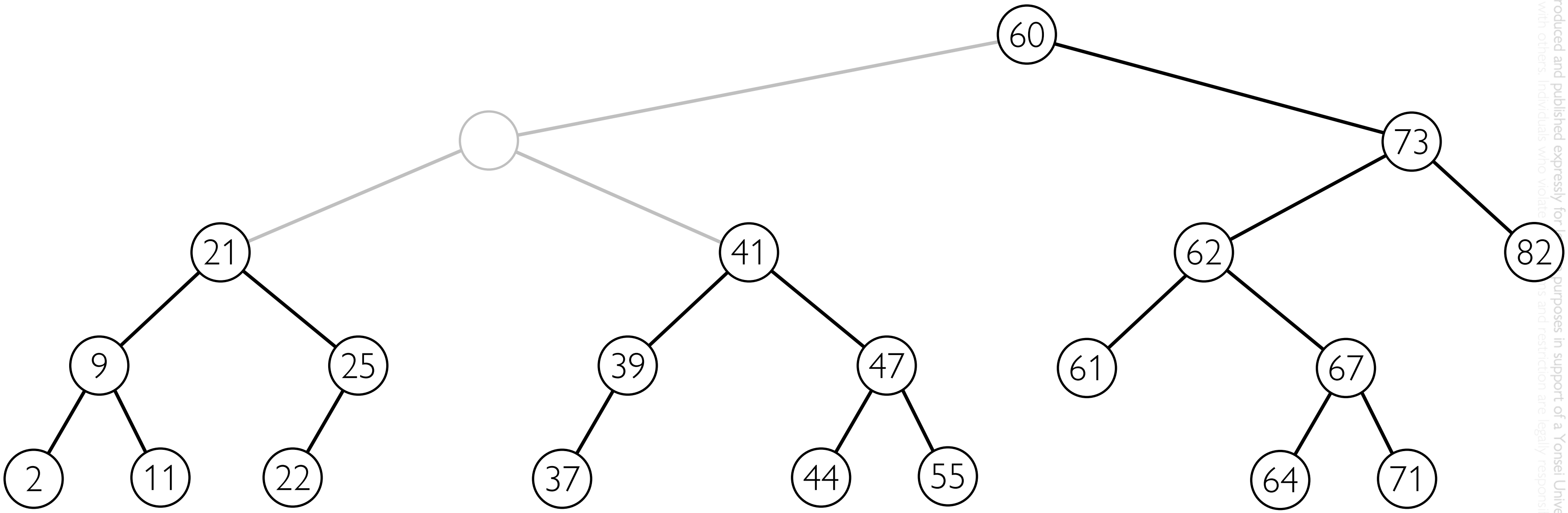
Binary Search Tree

- Deleting a node with a two children



Binary Search Tree

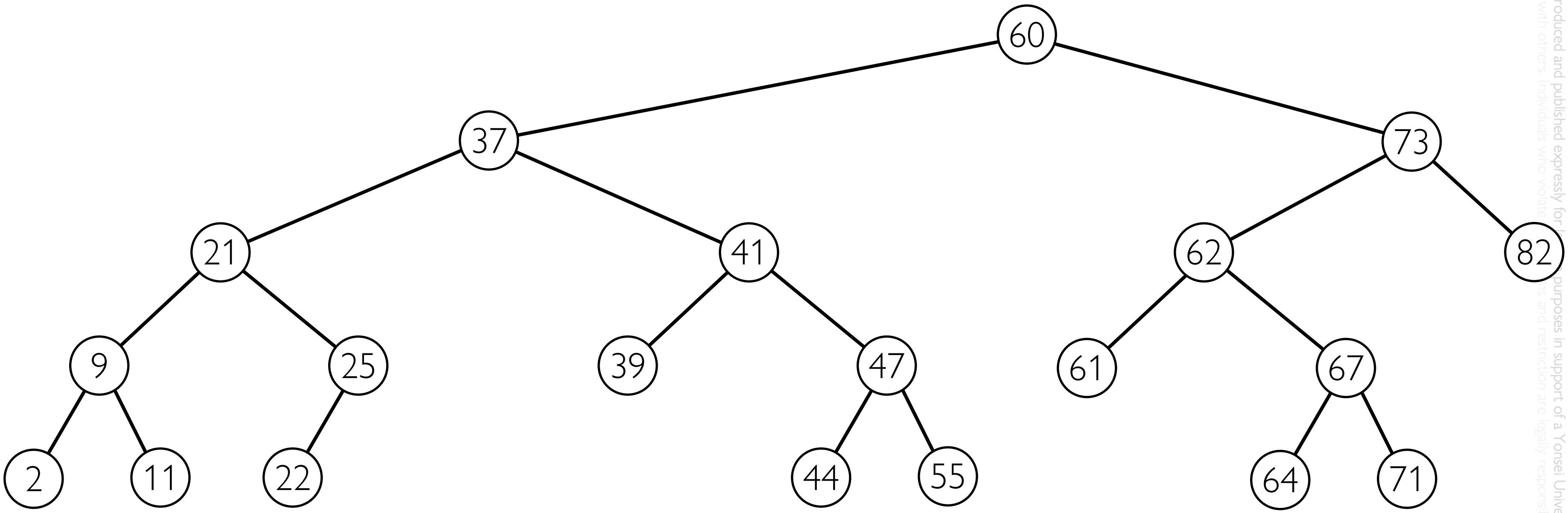
- Deleting a node with a two children



- Replace with max of left subtree or min of right subtree
  - Question: How to find them?

Binary Search Tree

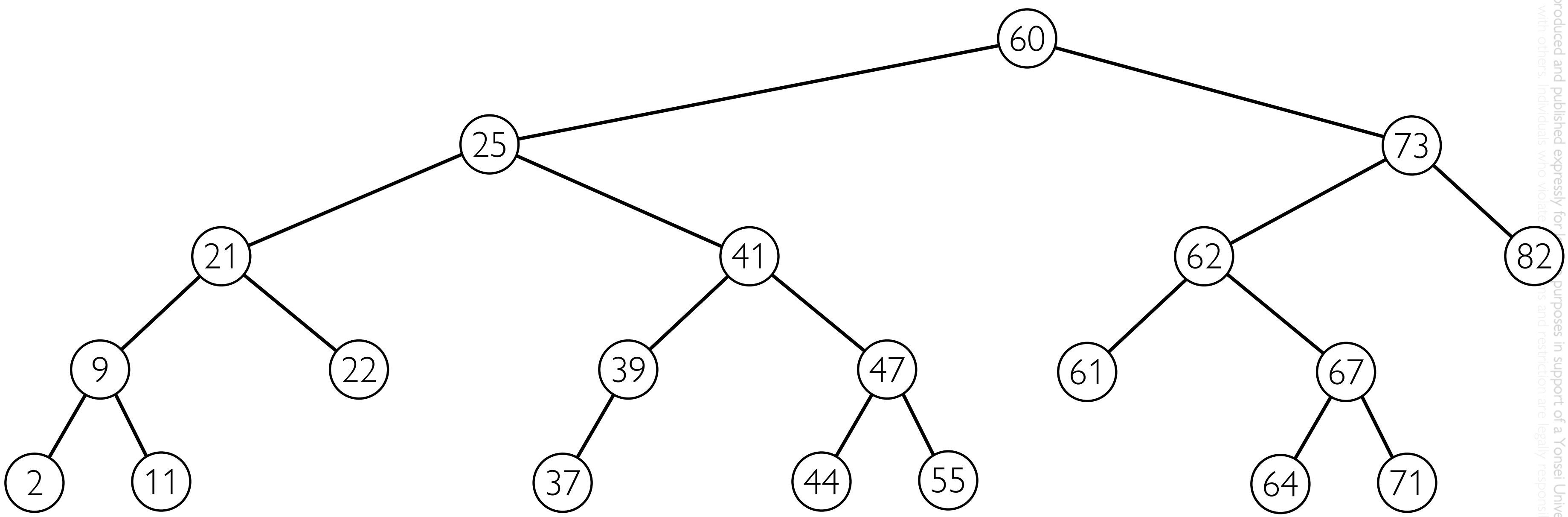
- Deleting a node with a two children



- Replace with max of left subtree or min of right subtree
  - Question: How to find them?

Binary Search Tree

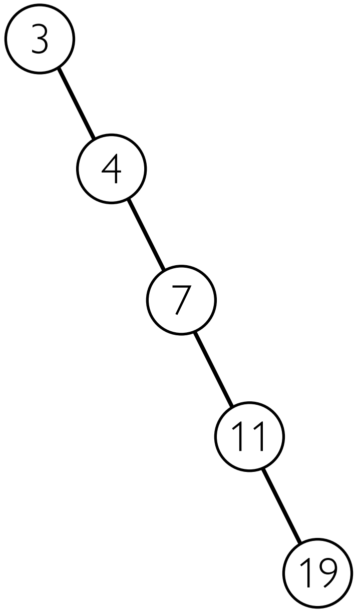
- Deleting a node with a two children



- Replace with max of left subtree or min of right subtree
  - Question: How to find them?

Time Complexity

- Sorted array
  - $O(\log n)$ search
  - $O(n)$ insertion
  - $O(n)$ deletion
- Binary search tree
  - $O(h)$ search
  - $O(h)$ insertion
  - $O(h)$ deletion
- but $\log(n+1) - 1 \leq h \leq n - 1$ for n nodes
- Need to balance the tree to have small height
  - Will get back to this in Chapter 11



Summary

- Tree is a nonlinear data structure
- Allows nonlinear hierarchical relationship + linear relationship
- General trees
- Binary trees
  - linked vs. array
- Traversals
  - preorder, postorder, breath-first, inorder (binary tree only)
- Binary search tree (BST)
  - another natural way to store ordered elements
- Next: see how binary tree becomes the basis of another data structure