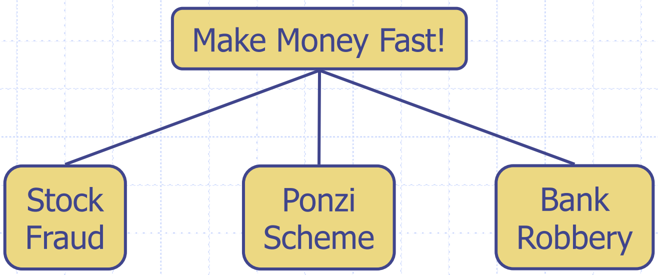
# So Far

- Now, familiar with
  - Order of running time
  - Big-Oh function
  - Amortized analysis
- Vector and List
  - Storing elements in a linear fashion
- Position
  - Containers and Iterators



# Trees



# Summary

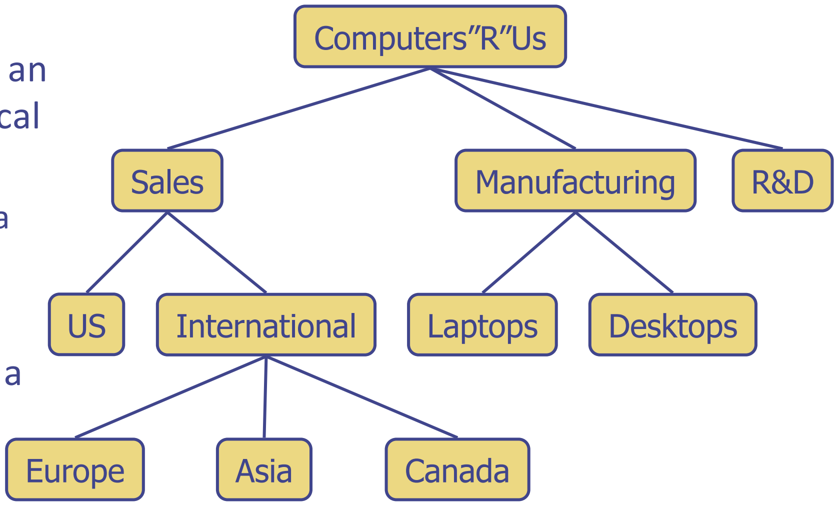
- Reading: Chapters 7.1, 7.2, 7.3
  - This chapter: Basics
  - Later in Chapter 10, we will cover:



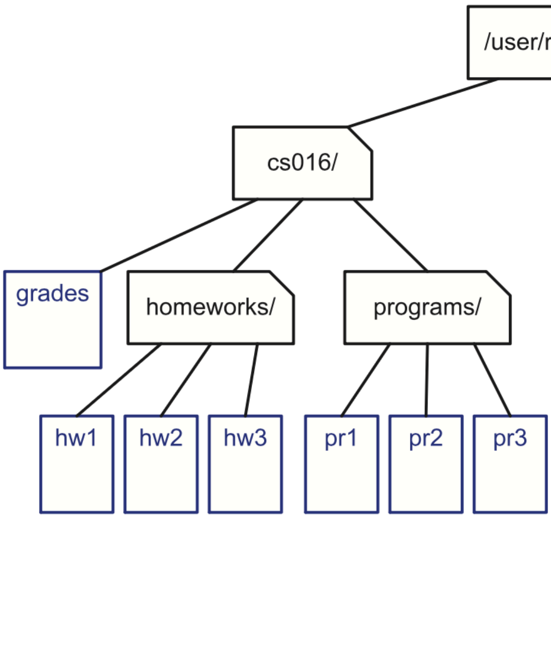


# What is a Tree?

- A graph without cycles
- In software systems, a tree is an abstract model of a hierarchical structure
  - Compared with "linear" data structures
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



# Example: File System

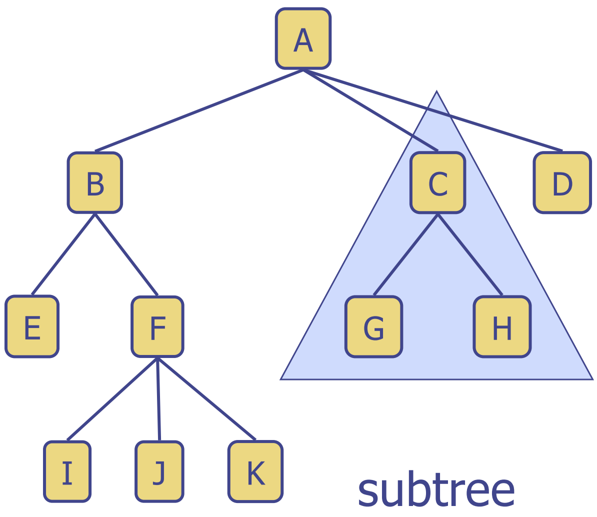





# Tree Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf ): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild, grand-grandchild, etc.

- Subtree: tree consisting of a node and its descendants



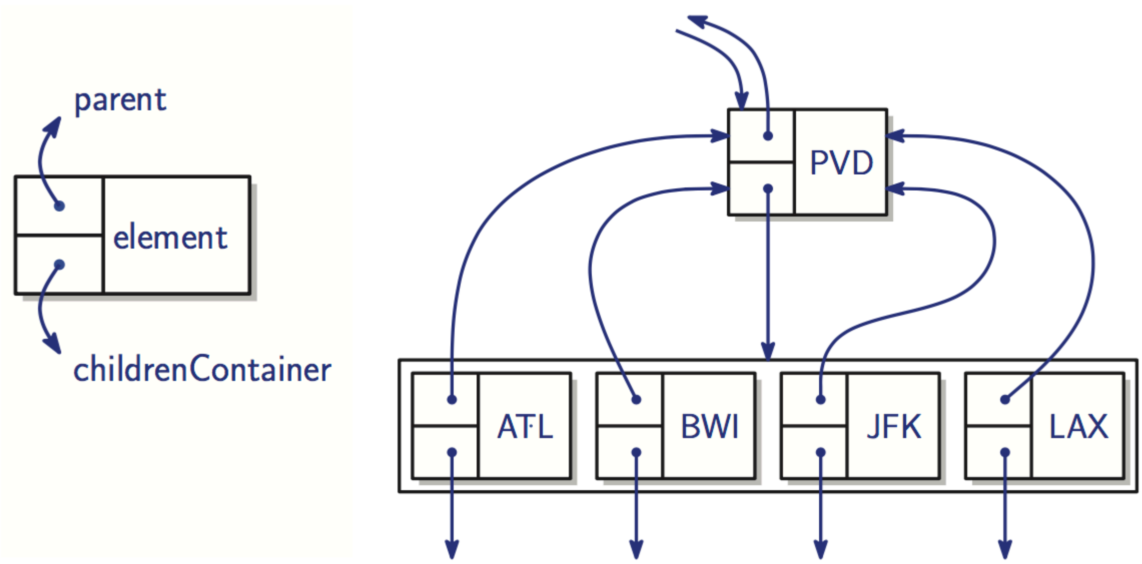
# Tree ADT

- We can use positions to abstract nodes
- Generic methods:
  - integer size()
  - boolean empty()
- Accessor methods:
  - position root()
  - list<position> positions()
- Position-based methods:
  - position p.parent()
  - list<position> p.children()
- Query methods:
  - boolean p.isRoot()
  - boolean p.isExternal()
- Additional "update" methods may be defined by data structures implementing the Tree ADT
  - Remove the node at some position
  - Swap a parent and its specific child
  - Etc …



# A linked structure for General Trees

- One way of implementing a general tree




(a) (b)

# Tree Traversal Algorithms



## Traversal Computations

1. Depth?

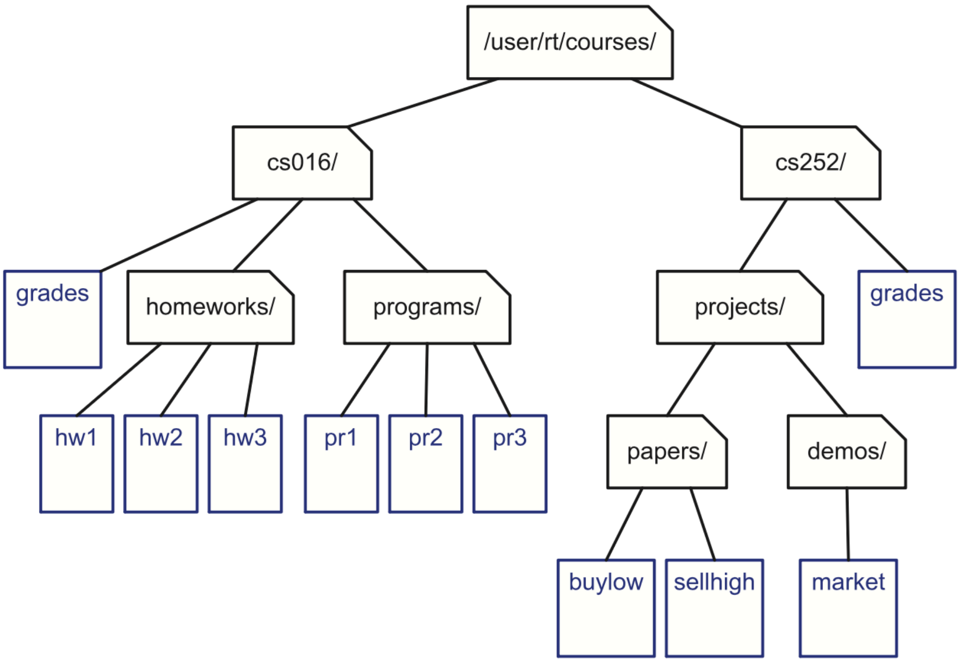
2. Height?

3. Visit every nodes
   - Preorder
   - Postorder
   - Inorder

◆ These are the basic things to do for a given tree

# Example: "du" command

$> du –s .  Print the aggregate file sizes from the current directory






# 1. Depth of a node

```
int depth(const Tree& T, const Position& p) {
  if (p.isRoot())
    return 0;                              // root has depth 0
  else
    return 1 + depth(T, p.parent());       // 1 + (depth of parent)
}
```

Complexity? $O(d_p)$, worst-case $O(n)$

# 2. Height of a tree T: height1

- Equal to the maximum depth of its leaves
- OK. Then, what about this algorithm?

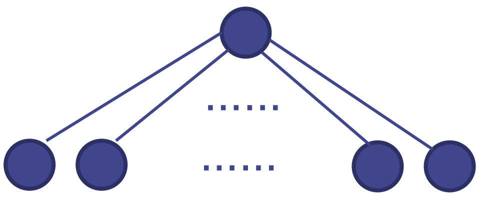
```
int height1(const Tree& T) {
  int h = 0;
  PositionList nodes = T.positions();          // list of all nodes
  for (Iterator q = nodes.begin(); q != nodes.end(); ++q) {
    if (q−>isExternal())
      h = max(h, depth(T, *q));                // get max depth among leaves
  }
  return h;
}
```

- Complexity?

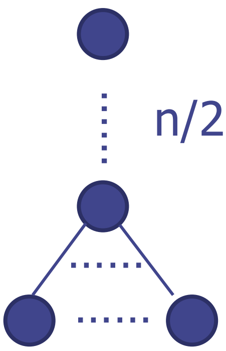
$O(n + \sum_p (1 + d_p))$ Worst-case: $O(n^2)$



# Two Trees



n-1 leaves



n/2 non-leaves

n/2 leaves

# 2. Height of a tree T: height2

- Why is height1 inefficient?

```
int height2(const Tree& T, const Position& p) {
  if (p.isExternal()) return 0;            // leaf has height 0
  int h = 0;
  PositionList ch = p.children();          // list of children
  for (Iterator q = ch.begin(); q != ch.end(); ++q)
    h = max(h, height2(T, *q));
  return 1 + h;                            // 1 + max height of children
}
```

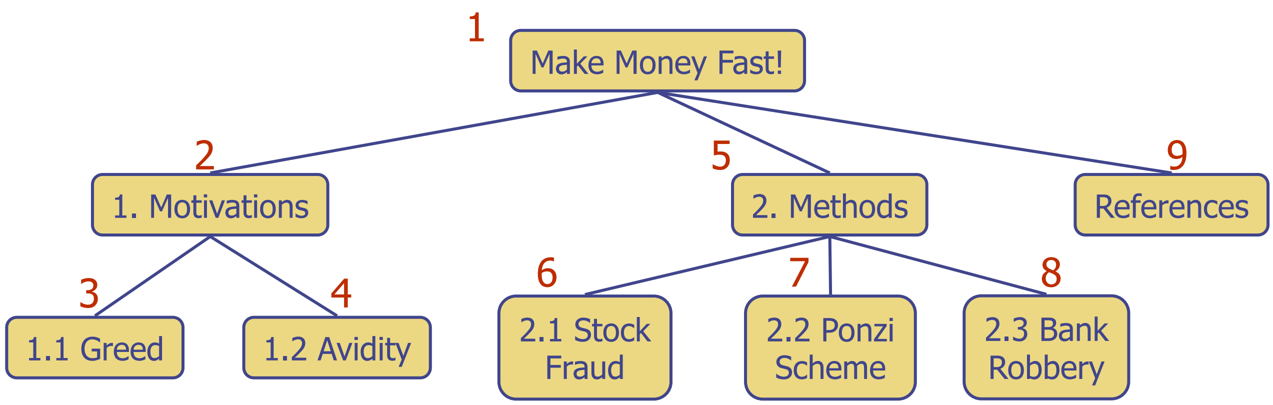
$$O(\sum_p (1 + c_p))$$ Worst-case: *O(n)*



# 3. Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

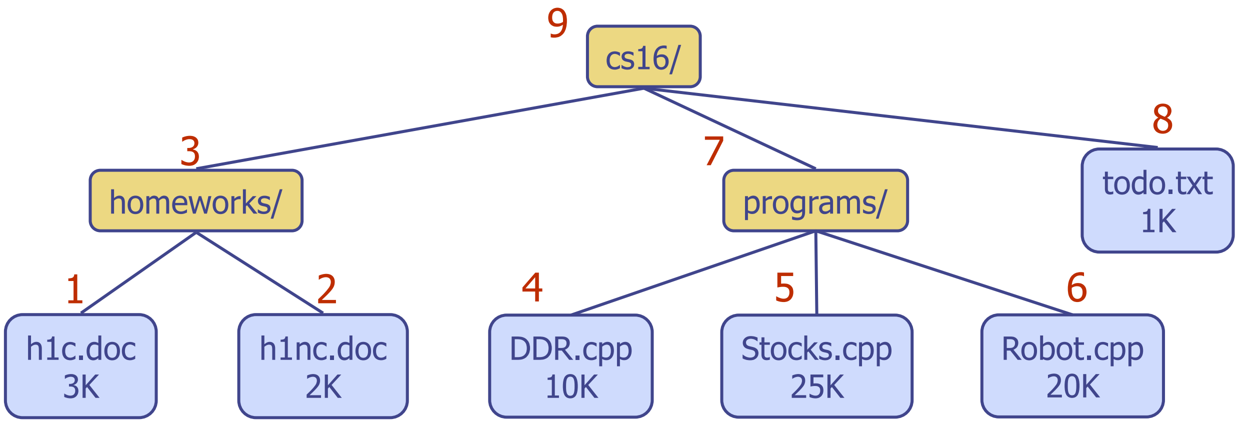
**Algorithm** *preOrder*(*v*)
    *visit*(*v*)
    **for each** child *w* of *v*
        *preorder* (*w*)



# 3. Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

**Algorithm** *postOrder*(*v*)
    **for each** child *w* of *v*
        *postOrder* (*w*)
    *visit*(*v*)

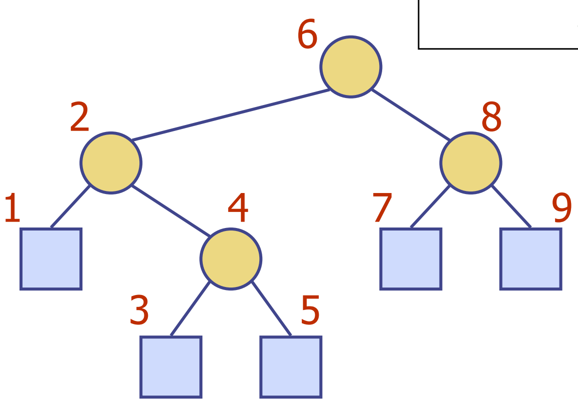





# 3. Inorder Traversal

- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

**Algorithm** *inOrder*(*v*)
    **if** ¬ *v.isExternal*()
        *inOrder*(*v.left*())
    *visit*(*v*)
    **if** ¬ *v.isExternal*()
        *inOrder*(*v.right*())



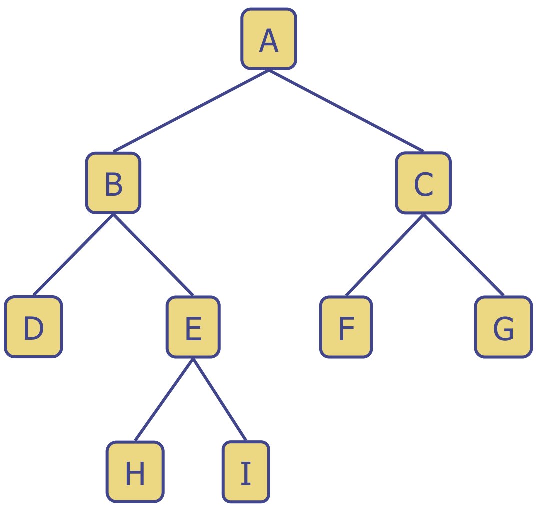
# Binary Tree



# Binary Trees

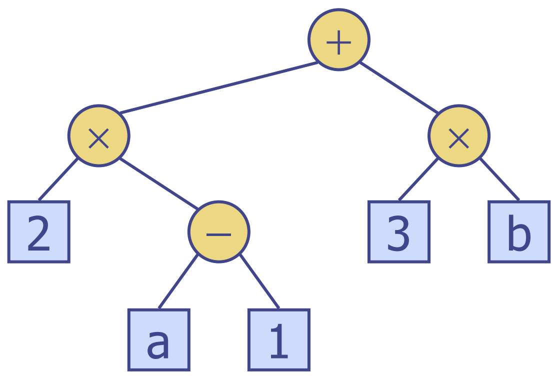
- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (exactly two for proper binary trees)
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree

- Applications:
  - arithmetic expressions
  - decision processes
  - searching



# Arithmetic Expression Tree

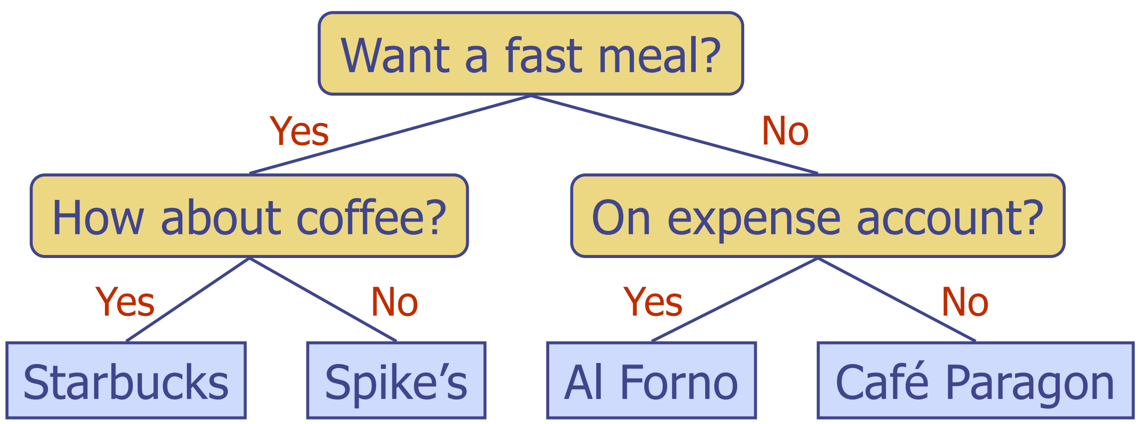
- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$






# Decision Tree

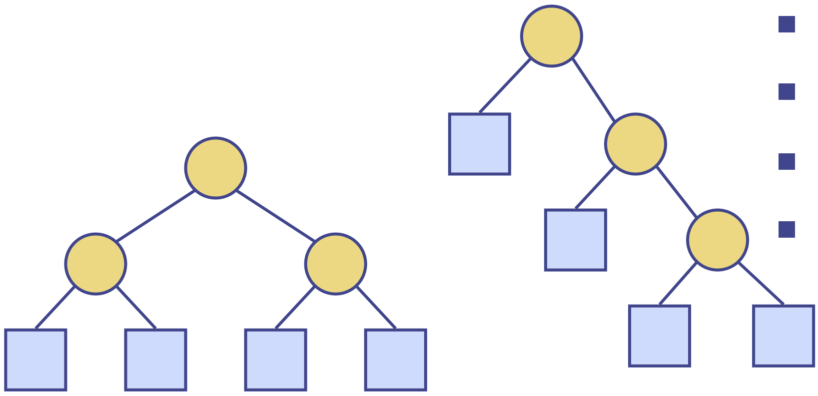
- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



# Properties of Proper Binary Trees

- Notation
  - $n$ number of nodes
  - $e$ number of external nodes
  - $i$ number of internal nodes
  - $h$ height

- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$





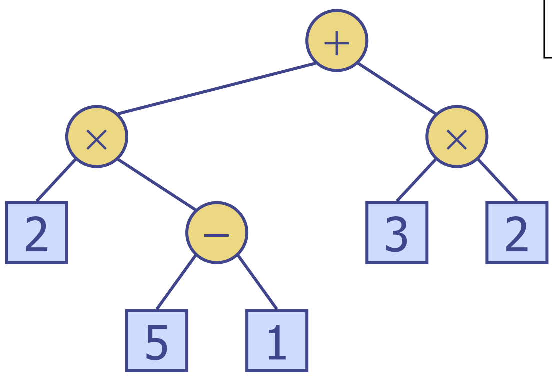
# BinaryTree ADT

- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - position p.left()
  - position p.right()
- Update methods may be defined by data structures implementing the BinaryTree ADT
- Proper binary tree: Each node has either 0 or 2 children

# Evaluate Arithmetic Expressions

- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

```
Algorithm evalExpr(v)
    if v.isExternal()
        return v.element()
    else
        x ← evalExpr(v.left())
        y ← evalExpr(v.right())
        ◊ ← operator stored at v
        return x ◊ y
```

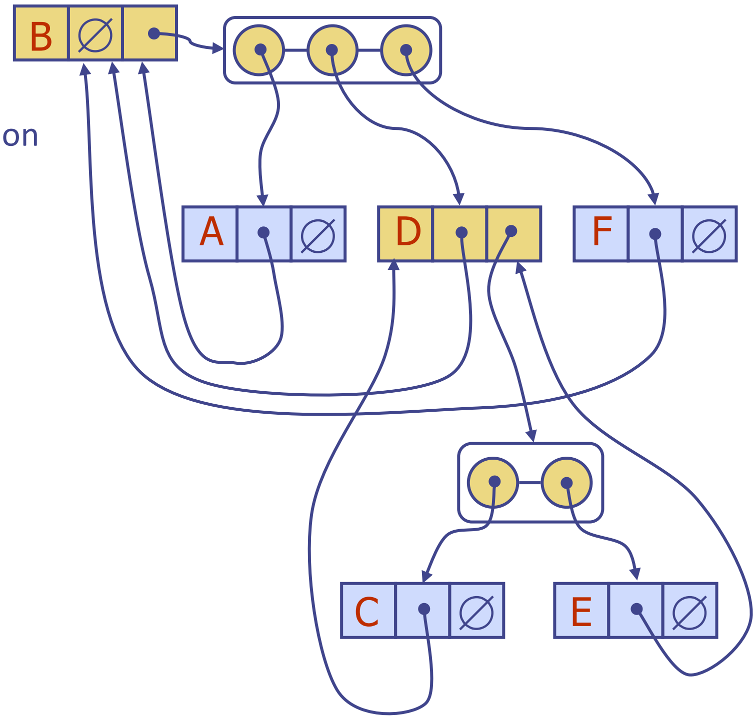


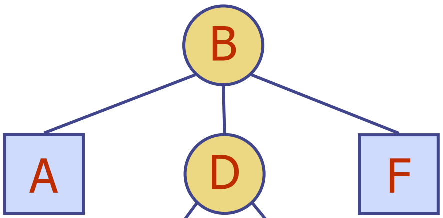



# How to represent trees in programming language?

# Recall: Linked Structure for Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes
- Node objects implement the Position ADT

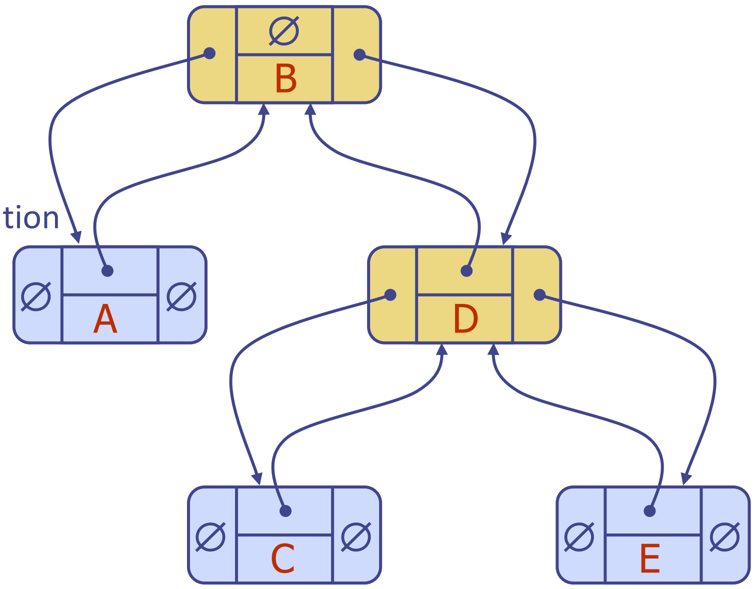


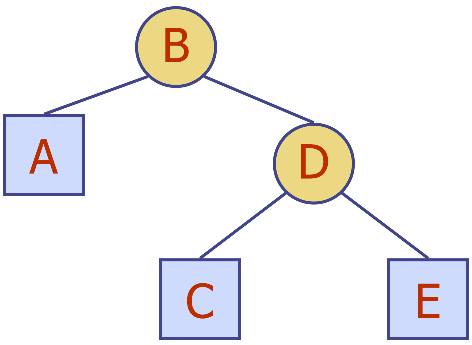






# Linked Structure for Binary Trees

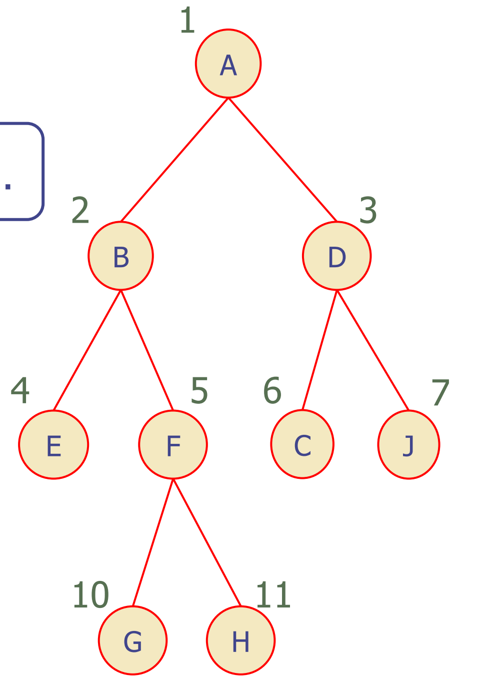
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT



# Array-Based Representation of Binary Trees

- Nodes are stored in an array A


- Node v is stored at A[rank(v)]
  - rank(root) = 1
  - if node is the left child of parent(node), rank(node) = 2 · rank(parent(node))
  - if node is the right child of parent(node), rank(node) = 2 · rank(parent(node)) + 1



# Questions?