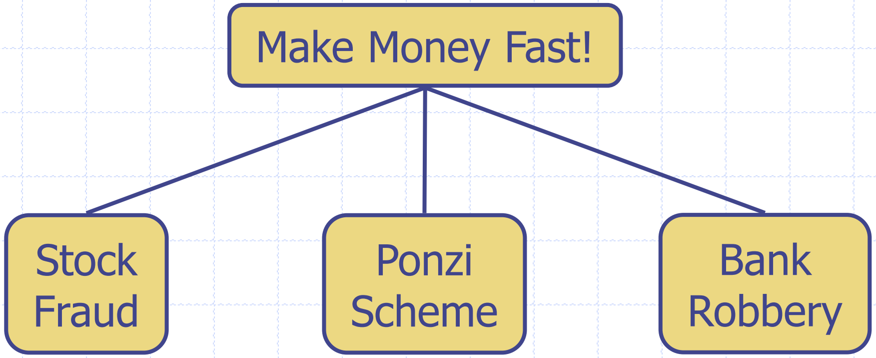
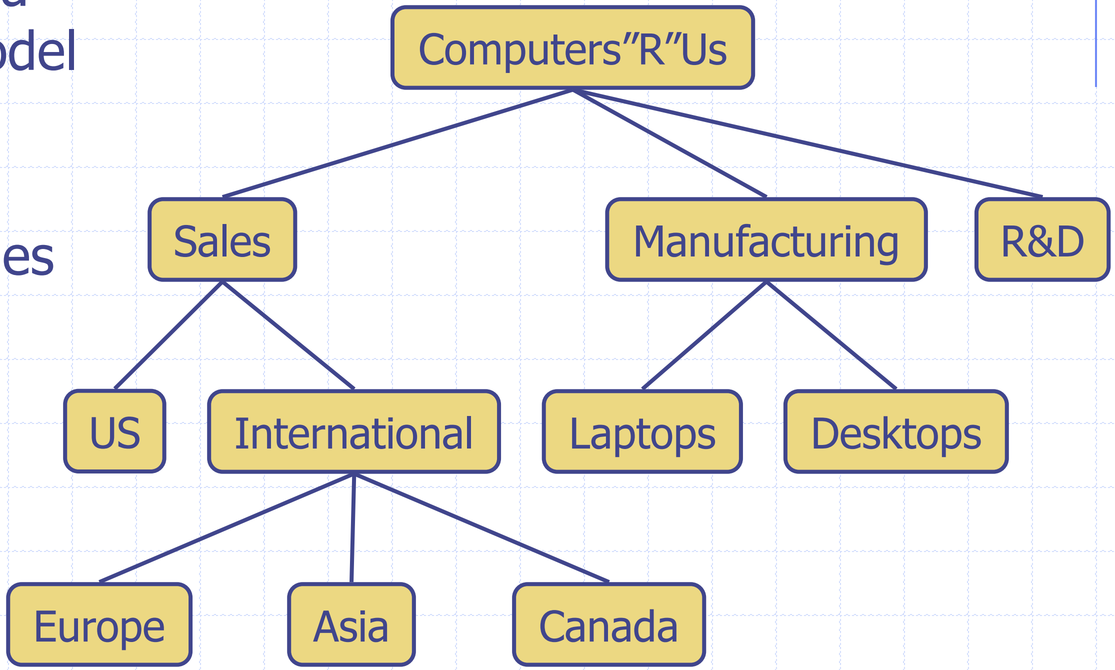
Trees

Sections 7.1-7.2



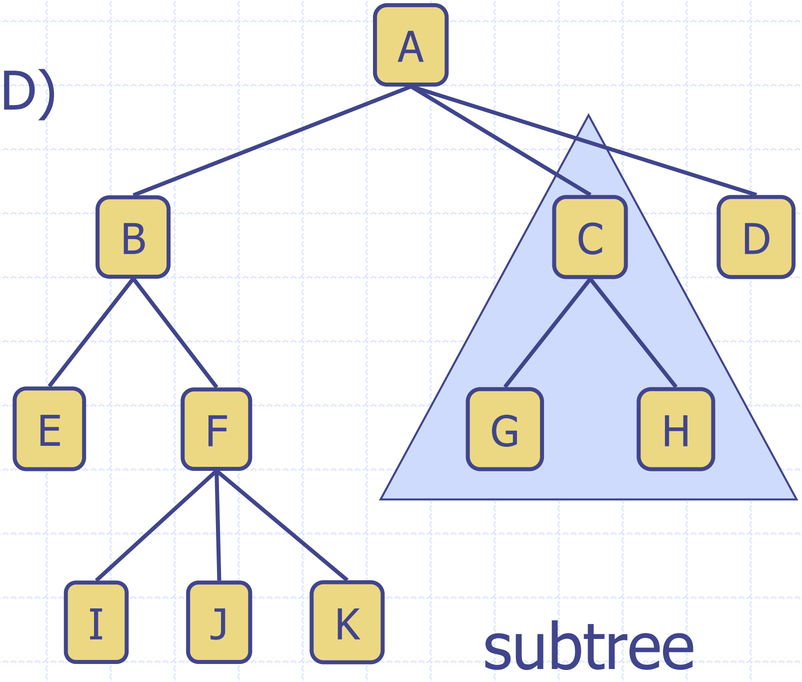
What is a Tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



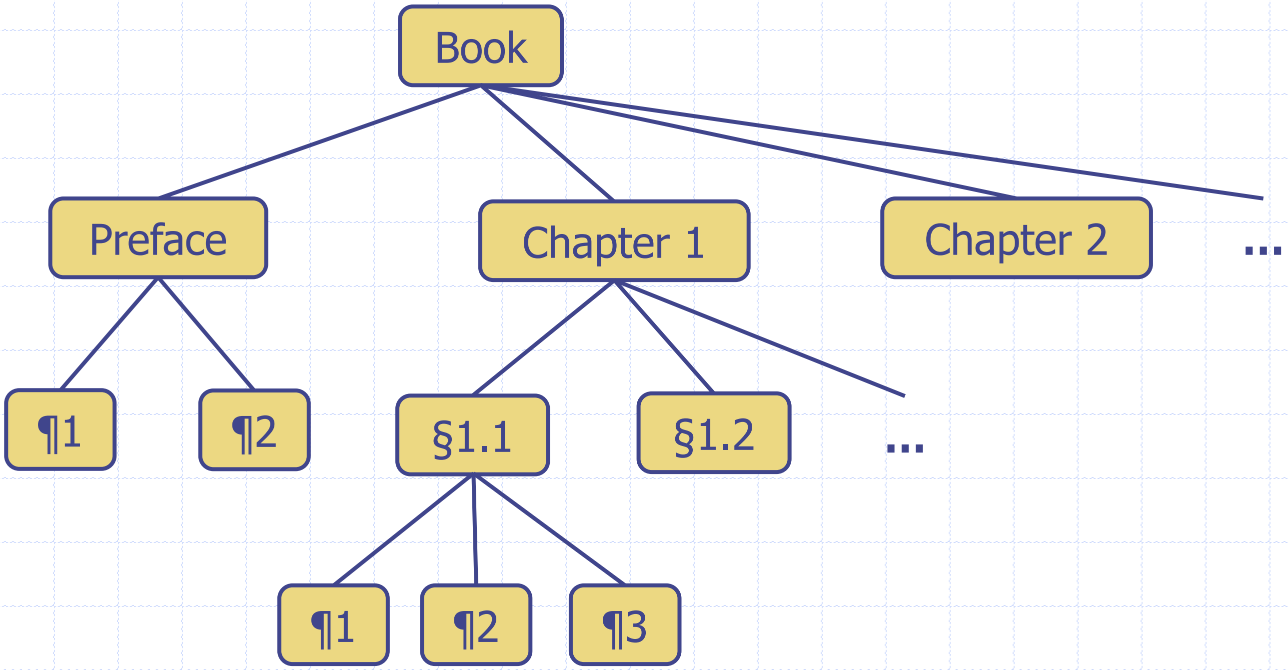
Tree Terminology

- 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, great-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, great-grandchild, etc.
- Subtree: tree consisting of a node and its descendants



Ordered Trees

- An ordered tree is a rooted tree where there is a linear ordering defined for the children of each node.
- In other words, we can identify the children of a node as the first, the second, the third, etc.
- Normally the children are drawn from left (first) to right (last).



Tree ADT

- We 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

Informal Position Interface

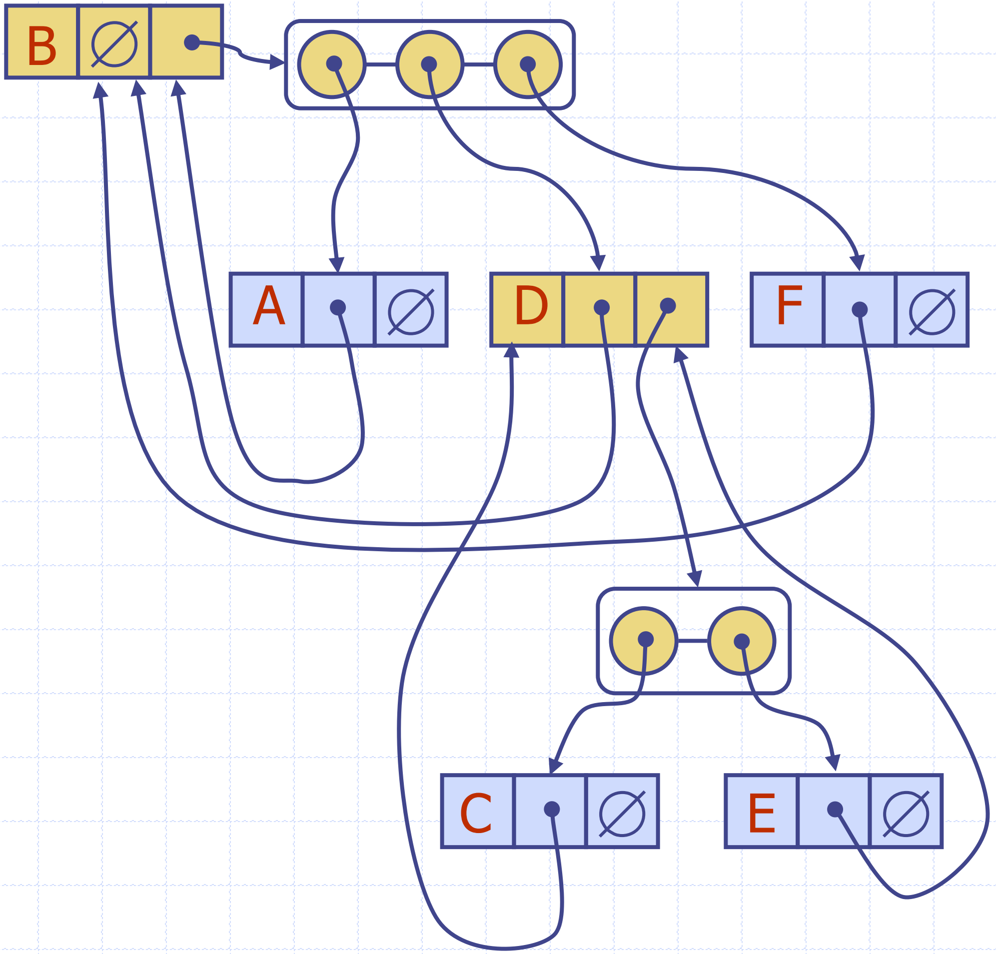
```
template <typename E>
class Position<E> {
public:
    E& operator*();
    Position parent() const;
    PositionList children() const;
    bool isRoot() const;
    bool isExternal() const;
}
```

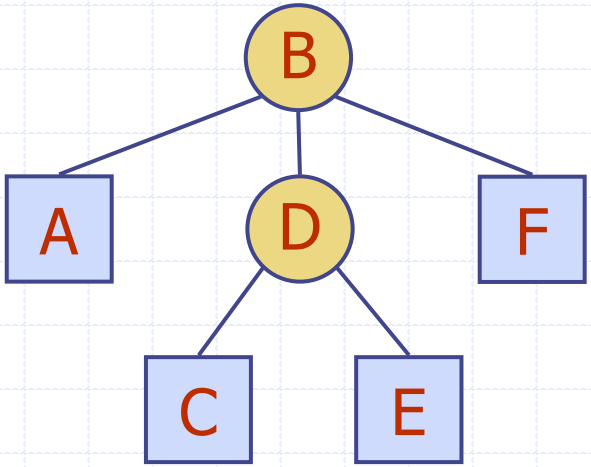
Informal Tree Interface

```
template <typename E>
class Tree<E> {
public:
    class Position;
    class PositionList;
    int size() const;
    bool empty() const;
    Position root() const;
    PositionList positions() const;
}
```

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





Analysis of Tree ADT Implementation

Using the linked structure, we can achieve the following running times for the operations of the Tree ADT:

Operation	***Time [for iterator]***
isRoot()	O(1)
isExternal()	O(1)
parent()	O(1)
children(p)	$O(c_p)$ [O(1)]
size()	O(1)
empty()	O(1)
root()	O(1)
positions()	O(n) [O(1)]

c_p is the number of children of node p

n is the number of nodes in the tree

Depth of a Node

- Recall that the depth of a node is the number of ancestors it has.
- We can recursively define the depth of a node p as follows:
  - if p is the root, then its depth is 0
  - otherwise, the depth of p is 1 plus the depth of the parent of p

```
Algorithm depth(T, p)
    if p.isRoot()
        return 0
    else
        return 1 + depth(T, p.parent())
```

Analysis of Depth()

- The algorithm *depth* is recursive.
- The base case is when p is the root of the tree.
- It makes progress towards the base case with every recursive call, since the call has a parameter of the parent of p, which is closer to the root than p.
- In the worst case, *depth* could take O(n) time.
- It is more accurate to characterize the running time in terms of the output parameter rather than the input (!)
- If *depth*(*p*) has an output of d_p, then the running time is $O(d_p)$, since it makes 1 call for each ancestor of p, and each call takes O(1) time. [Here the running time is ~$1+d_p$, but the O() swallows the 1.]
- The parameter d_p is often much smaller than n, so characterizing the worst-case time as $O(d_p)$ gives you more information than characterizing it as O(n).

Height of a Node

- The height of a node is defined recursively as well:
  - if p is external, then its height is 0
  - otherwise, the height of p is 1 plus the maximum height of a child of p.
- The height of a tree is the height of the root of the tree.
- Or, the height of a tree is equal to the maximum depth of its external nodes.

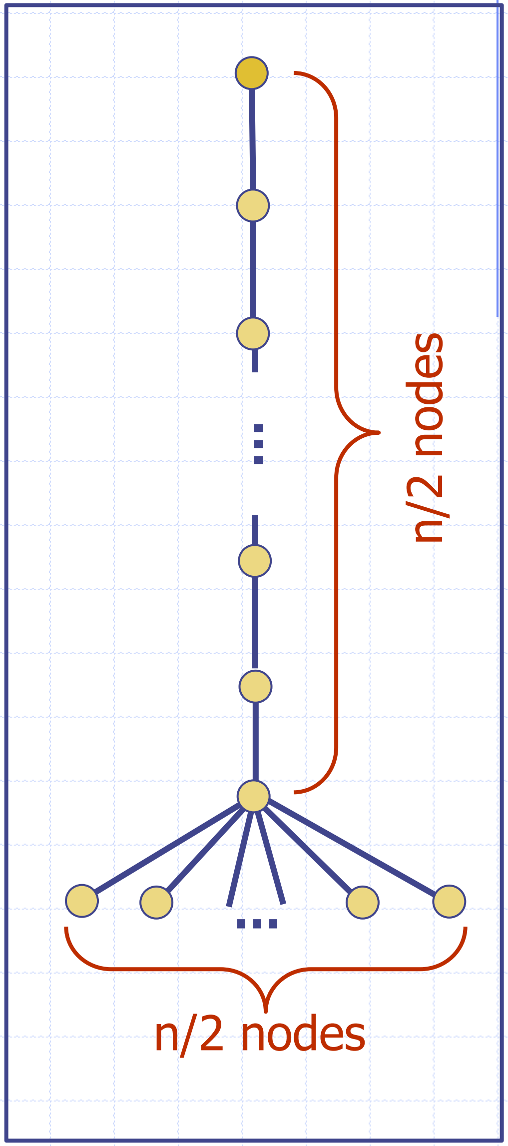
```
Algorithm height1(T)
    h = 0
    for each p in T.positions() do
        if p.isExternal() then
            h = max(h, depth(T, p))
    return h
```

Analysis of Height1()

- The algorithm *height1* is not very efficient.
- It takes O(n) time simply to go through all of the positions and check if they are external.
- It takes additional time to compute the depths of all of the external nodes. Let E be the set of external nodes of our tree T, and d_p be the depth of node p in the tree. The time to compute the depths is proportional to:

$$\sum_{p \in E} (1 + d_p)$$

- In the worst case, this sum is $\Theta(n^2)$.
- Therefore, *height1* takes $O(n^2)$ time.



Height of a Node, Again

- A better algorithm uses the recursive definition of height directly.
  - if p is external, then its height is 0
  - otherwise, the height of p is 1 plus the maximum height of a child of p.

```
Algorithm height2(T, p)
    if p.isExternal() then
        return 0
    else
        h = 0
        for each q in p.children() do
            h = max(h, height2(T, q))
        return 1 + h
```

Analysis of Height2()

- The algorithm *height2* is more efficient than *height1*.
- *height2* takes time $O(1 + c_p)$ time to perform the nonrecursive part: $O(1)$ time for the **if**, the **return**s, and the initial assignment to h, and $O(c_p)$ for the **for** loop. Here c_p is the number of children of node p.
- If initially called with the root of the tree T as p, it will eventually be called once for each node of the tree.
- Thus, the total time taken by *height2* is the sum of the nonrecursive time over all nodes of the tree.

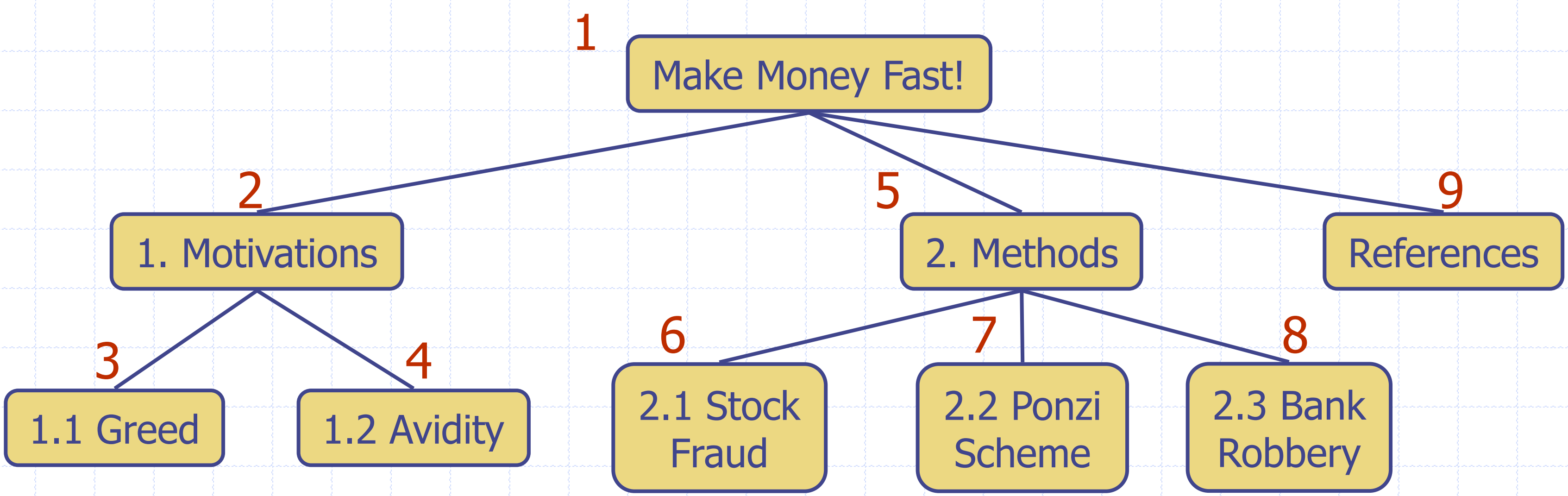
$$\sum_{p \in T} (1 + c_p)$$

- This sum is $2n - 1$.
- Therefore, *height2* takes $O(n)$ time.

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

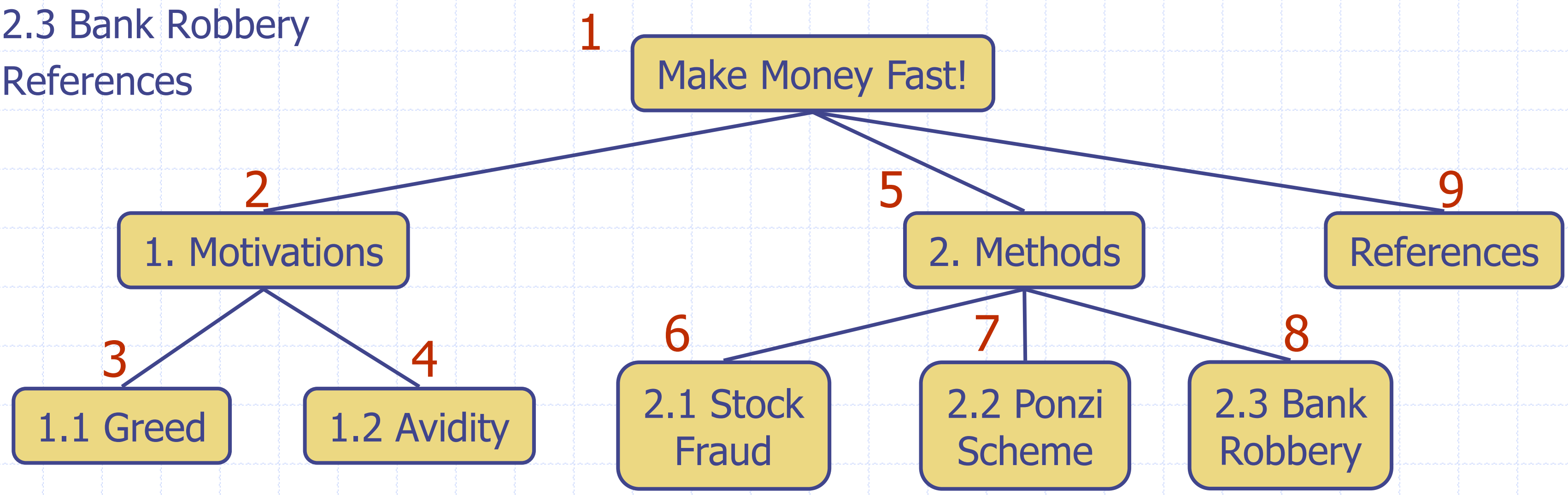
Algorithm *preorder*(v)
    visit(v)
    for each child w of v
        preorder (w)



Preorder Print

Make Money Fast!
1. Motivations
1.1 Greed
1.2 Avidity
2. Methods
2.1 Stock Fraud
2.2 Ponzi Scheme
2.3 Bank Robbery
References

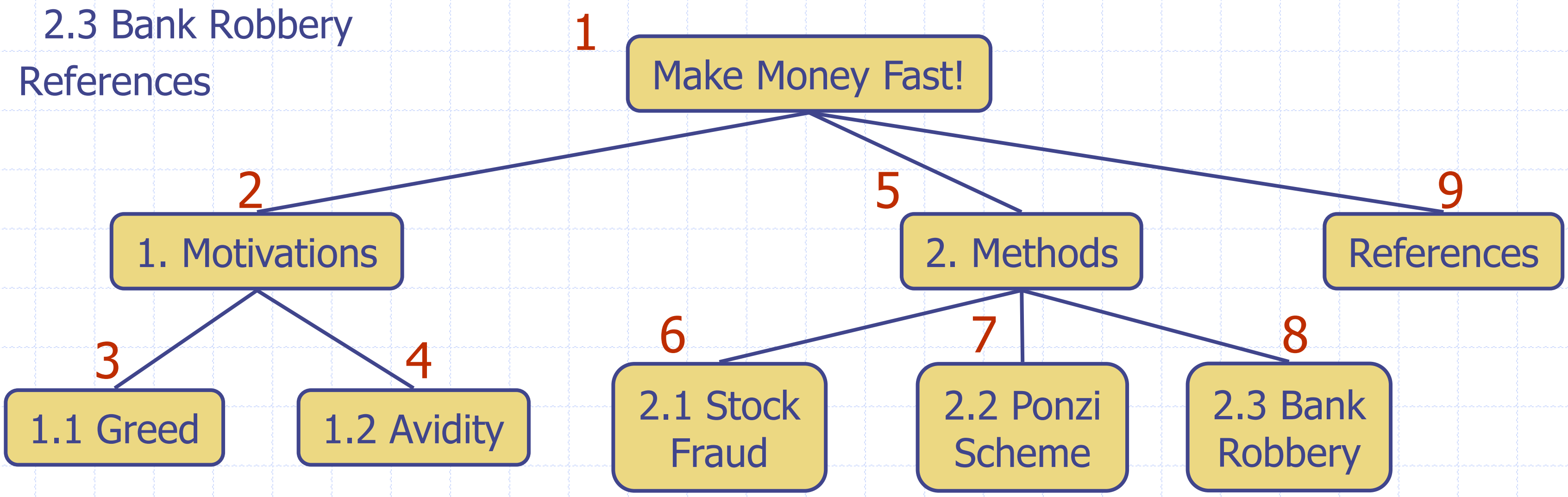
Algorithm ***preorderPrint*(*v*)**
 ***print*(*v*)**
 for each child ***w*** of ***v***
  ***preorderPrint*(*w*)**



Preorder Print

Make Money Fast!
  1. Motivations
    1.1 Greed
    1.2 Avidity
  2. Methods
    2.1 Stock Fraud
    2.2 Ponzi Scheme
    2.3 Bank Robbery
  References

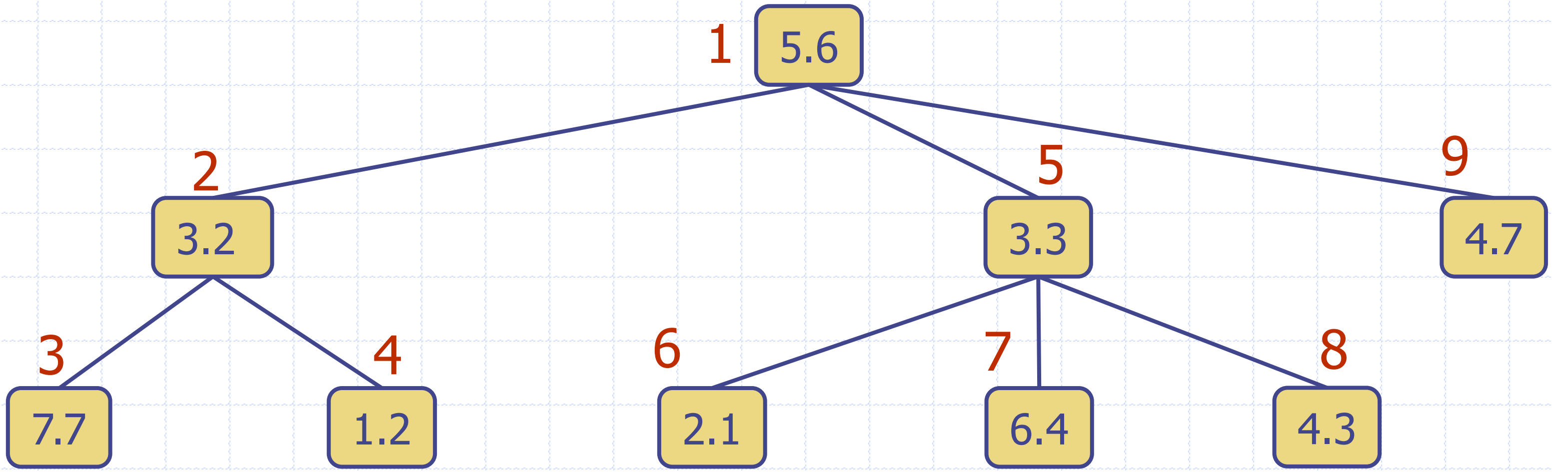
Algorithm *preorderPrint*(*v*, *indent*)
  *print indent*2 spaces*
  print(*v*)
  for each child *w* of *v*
    preorderPrint(*w*, *indent* + *1*)



Preorder Sum

PreorderSum(v) returns the sum of the elements of the subtree of v.

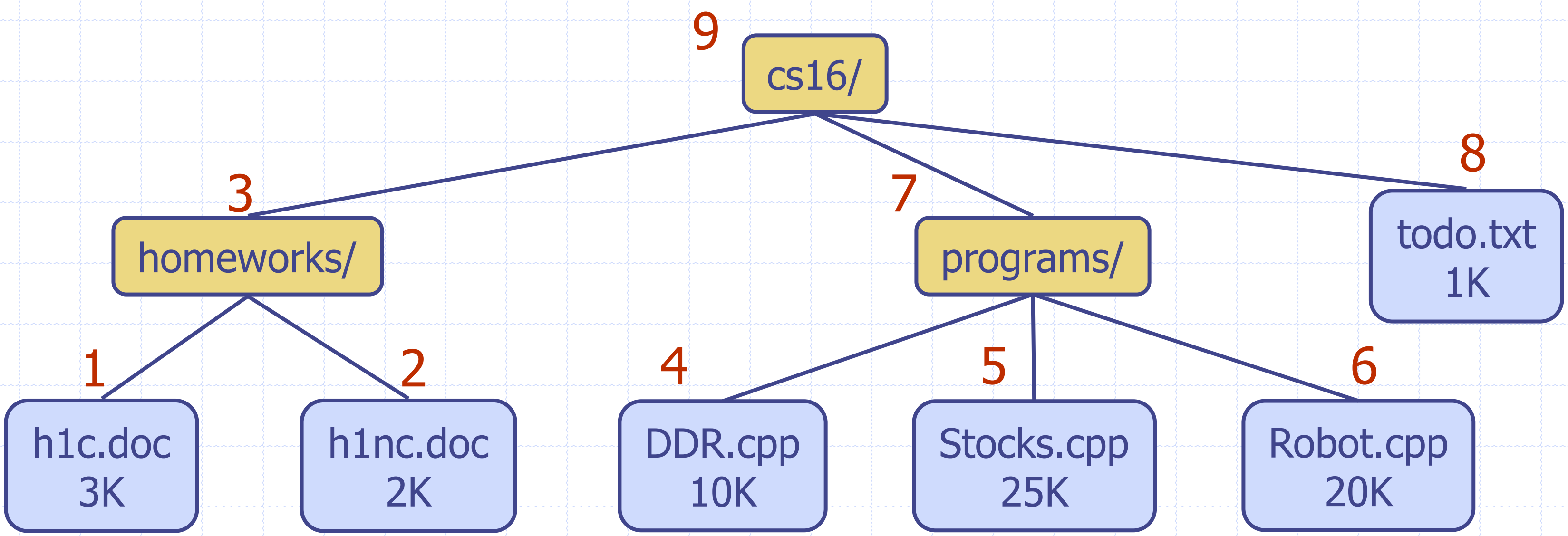
Algorithm *preorderSum*(*v*)
    sum = *v.element*( )
    for each child *w* of *v*
        sum = *sum* + *preorderSum*(*w*)
    return *sum*



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

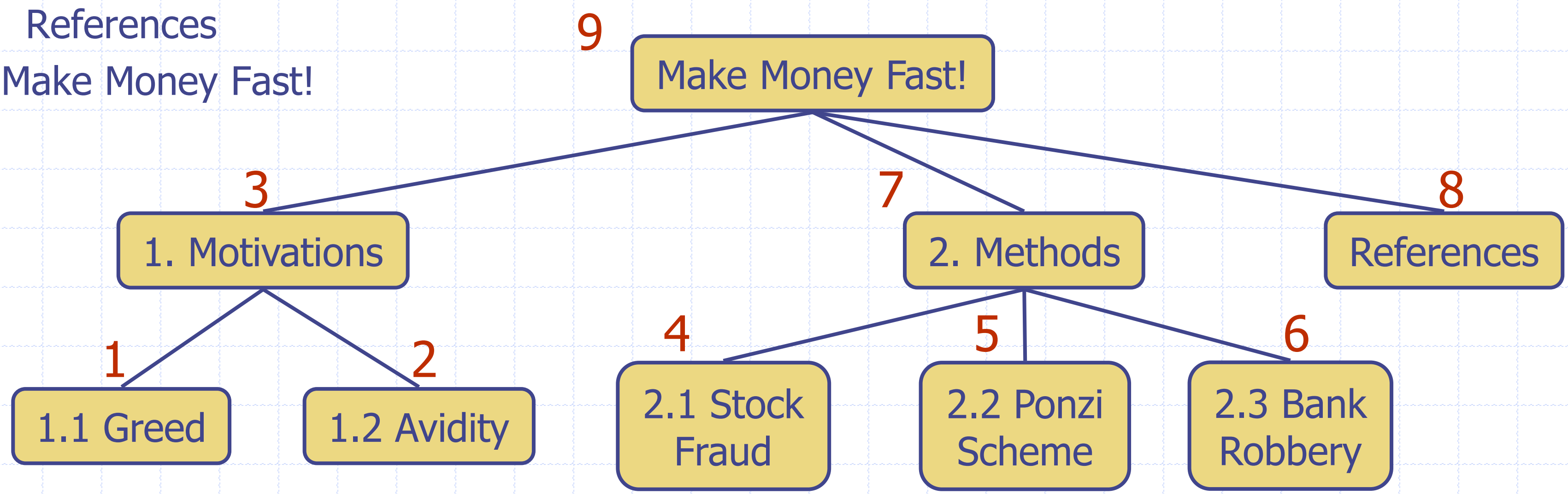
Algorithm ***postorder***(v)
    for each child w of v
        postorder (w)
    visit(v)



Postorder Print

1.1 Greed
1.2 Avidity
1. Motivations
2.1 Stock Fraud
2.2 Ponzi Scheme
2.3 Bank Robbery
2. Methods
References
Make Money Fast!

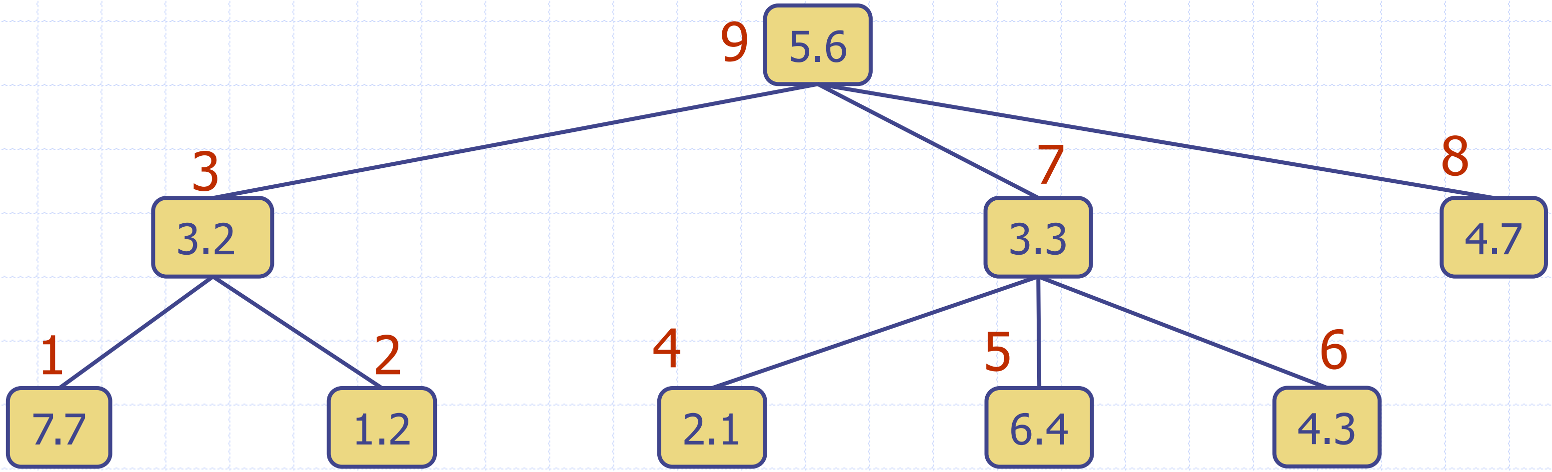
Algorithm *postorderPrint*(*v*, *indent*)
  for each child *w* of *v*
    postorderPrint(*w*, *indent* + *1*)
  *print indent*2 spaces*
  print(*v*)



Postorder Sum

- PostorderSum(v) returns the sum of the elements of the subtree of v.

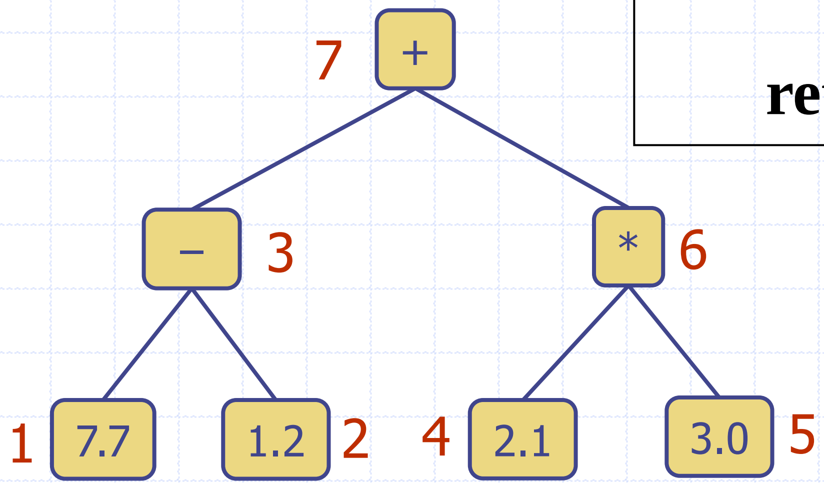
Algorithm *postorderSum*(*v*)
    sum = *0*
    for each child *w* of *v*
        sum = *sum* + *postorderSum*(*w*)
    return *sum* + *v.element*( )



Expression Evaluation (Postorder)

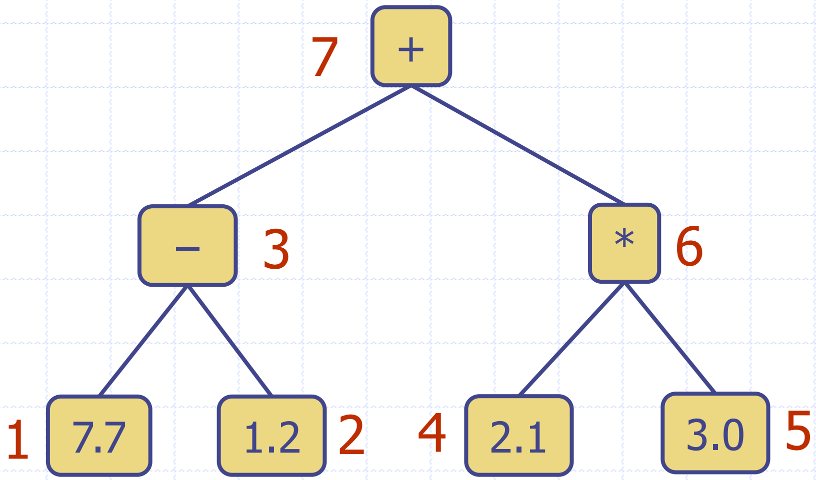
- evaluate(v) returns the value of the expression represented by the subtree of v.

Algorithm *evaluate*(*v*)
    if *v.isExternal*( )
        return *v.element*( )
    else
        arglist = { }
        for each child *w* of *v*
            arglist = *arglist* + *evaluate*(*w*)
    return *apply*(*v.element*( ), *arglist*)



(7.7 – 1.2) + (2.1 * 3.0)

Expression Evaluation (Postorder)



n7 calls n3
n3 calls n1
n1 returns 7.7
n3 calls n2
n2 returns 1.2
n3 executes apply(–, {7.7, 1.2})
n3 returns 6.5
n7 calls n6
n6 calls n4
n4 returns 2.1
n6 calls n5
n5 returns 3.0
n6 executes apply(*, {2.1, 3.0})
n6 returns 6.3
n7 executes apply(+, {6.5, 6.3})
n7 returns 12.8