CS101 Algorithms and Data Structures

Trees

Textbook Ch B.5, 10.4

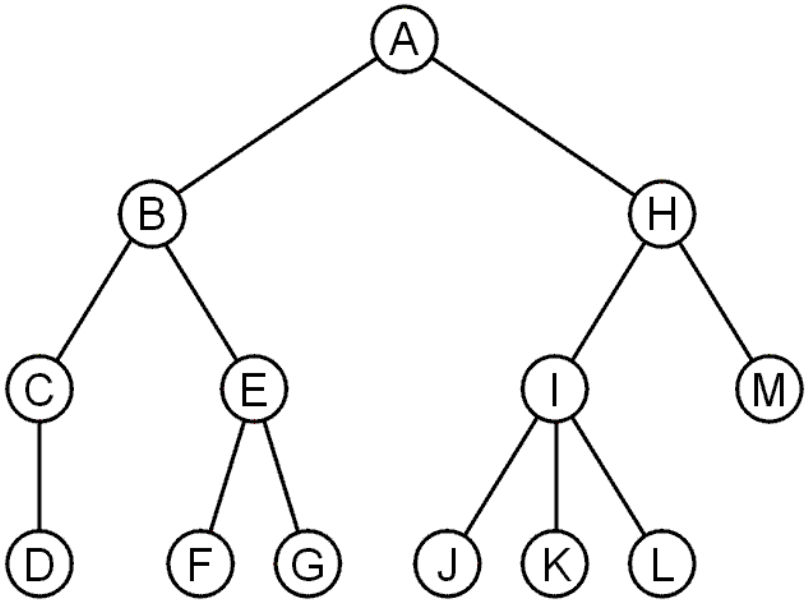
Outline

- Tree structure
- Implementation
- Tree traversal
- Forest

Trees

A rooted tree data structure stores information in *nodes*

- 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



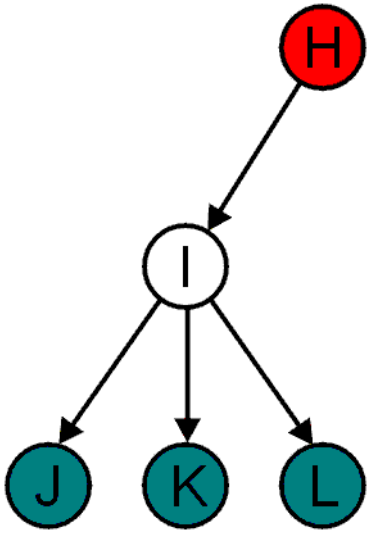
Terminology

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 of I



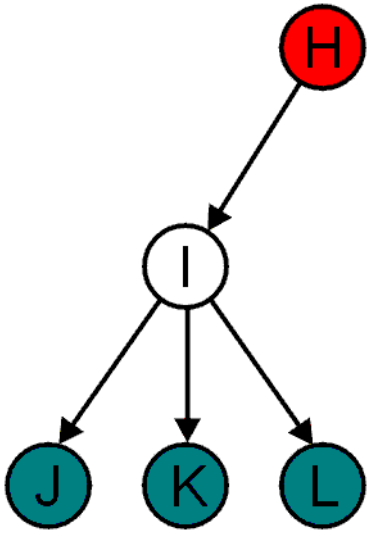
Terminology

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

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

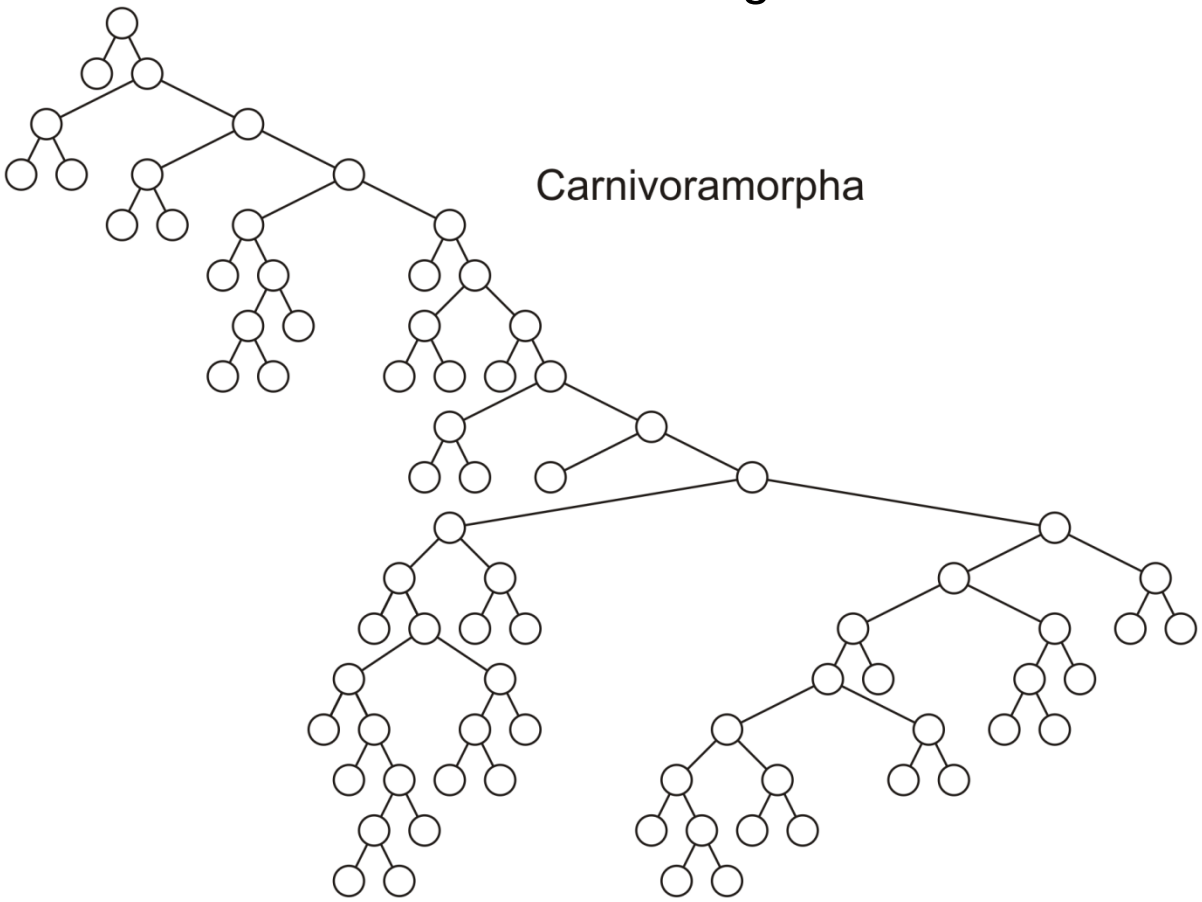
Nodes with the same parent are *siblings*

– J, K, and L are siblings



Terminology

Phylogenetic trees have nodes with degree 2 or 0:

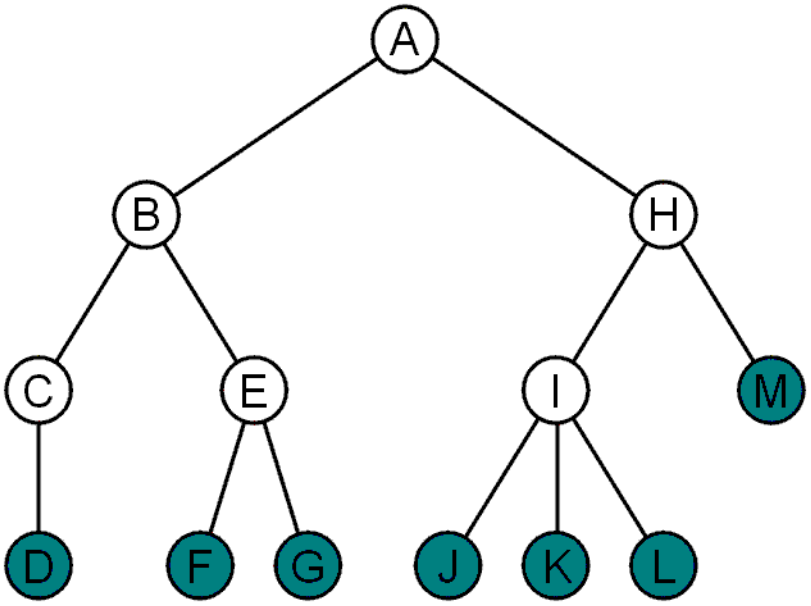


Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Terminology

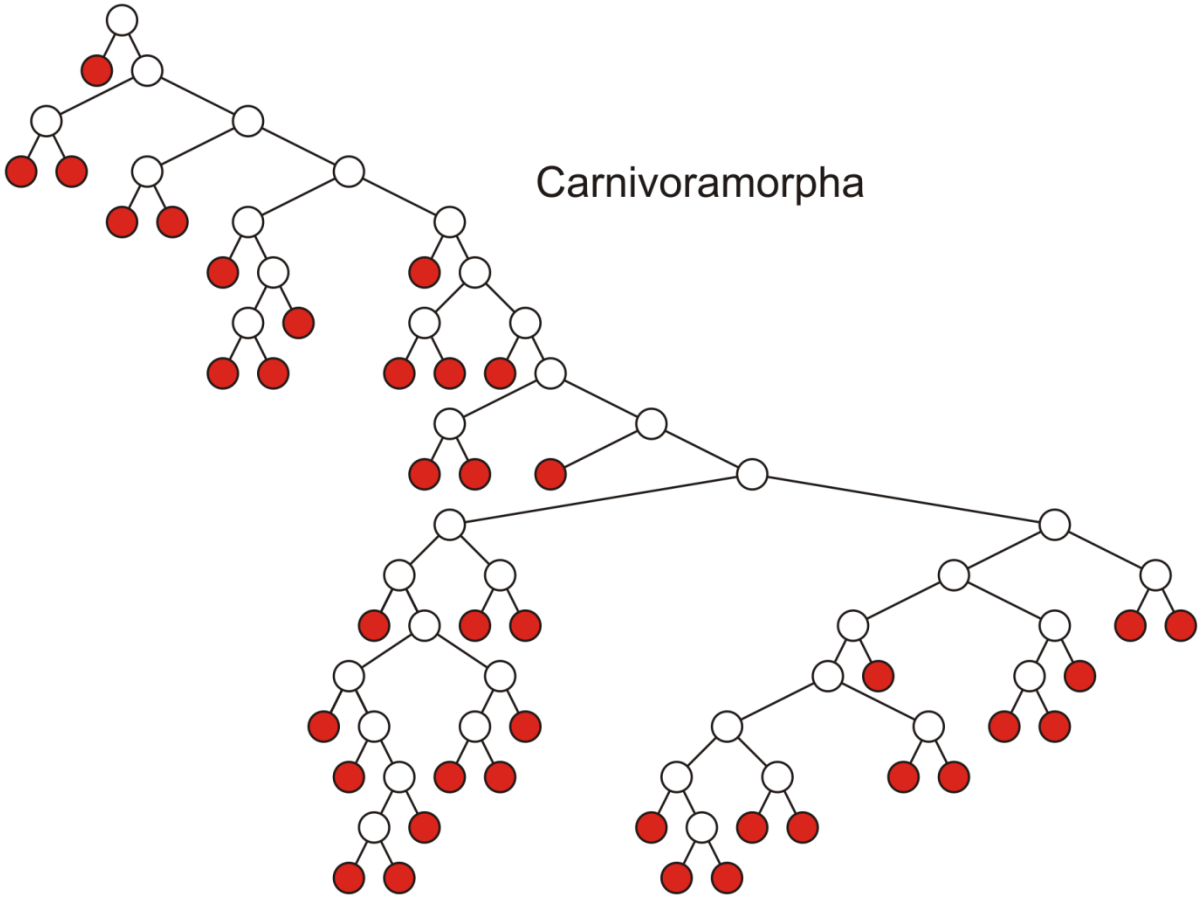
Nodes with degree zero are also called *leaf nodes*

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



Terminology

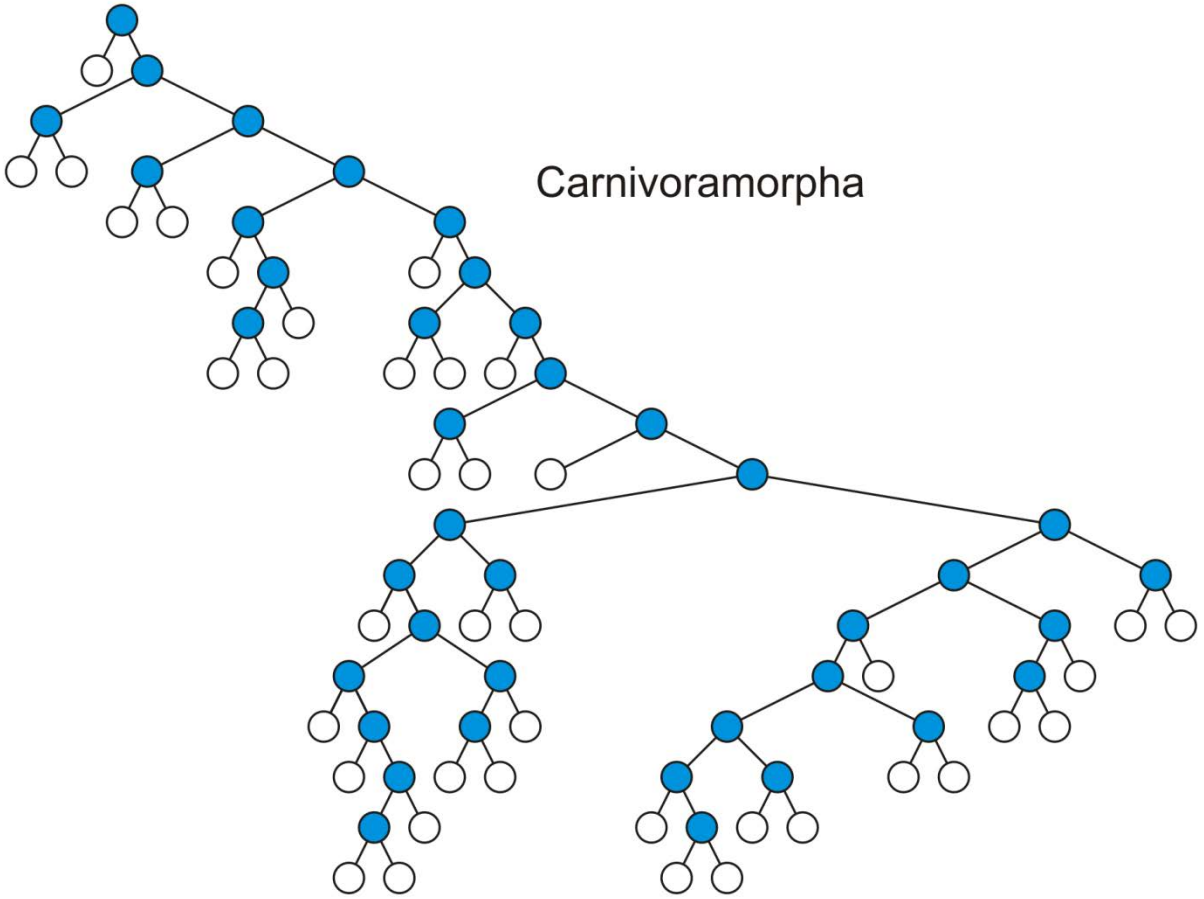
Leaf nodes:



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Terminology

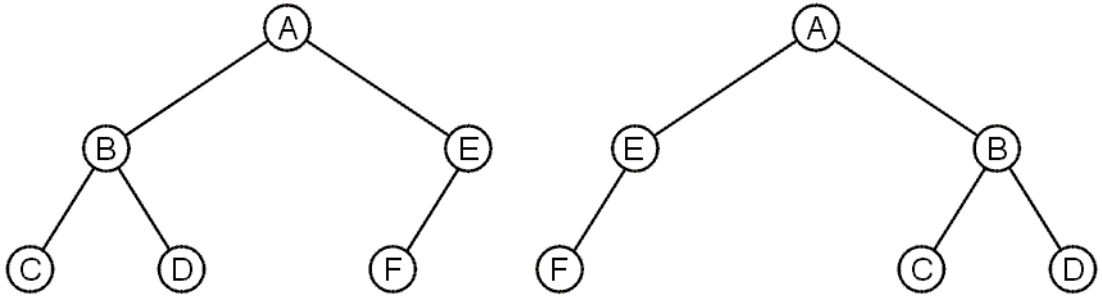
Internal nodes:



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

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)

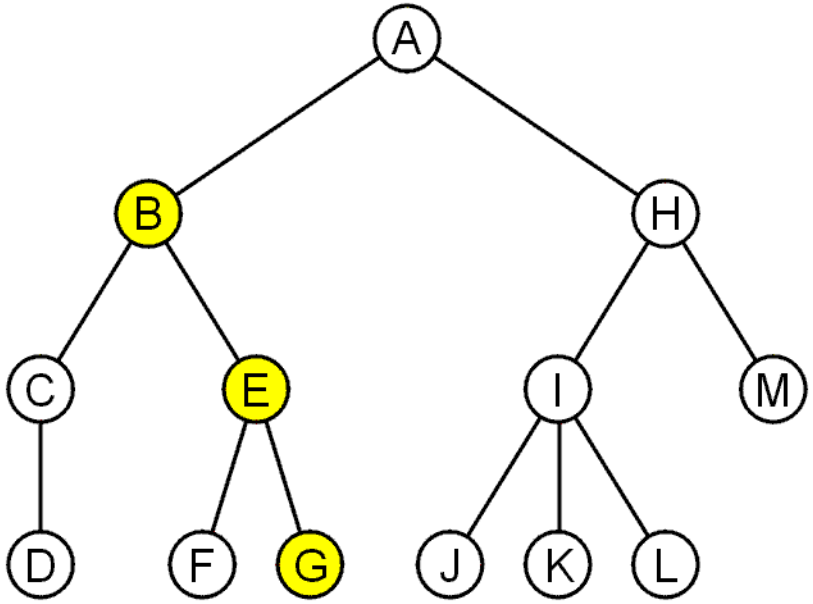
Terminology

A path is a sequence of nodes: $(a_0, a_1, ..., a_n)$

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

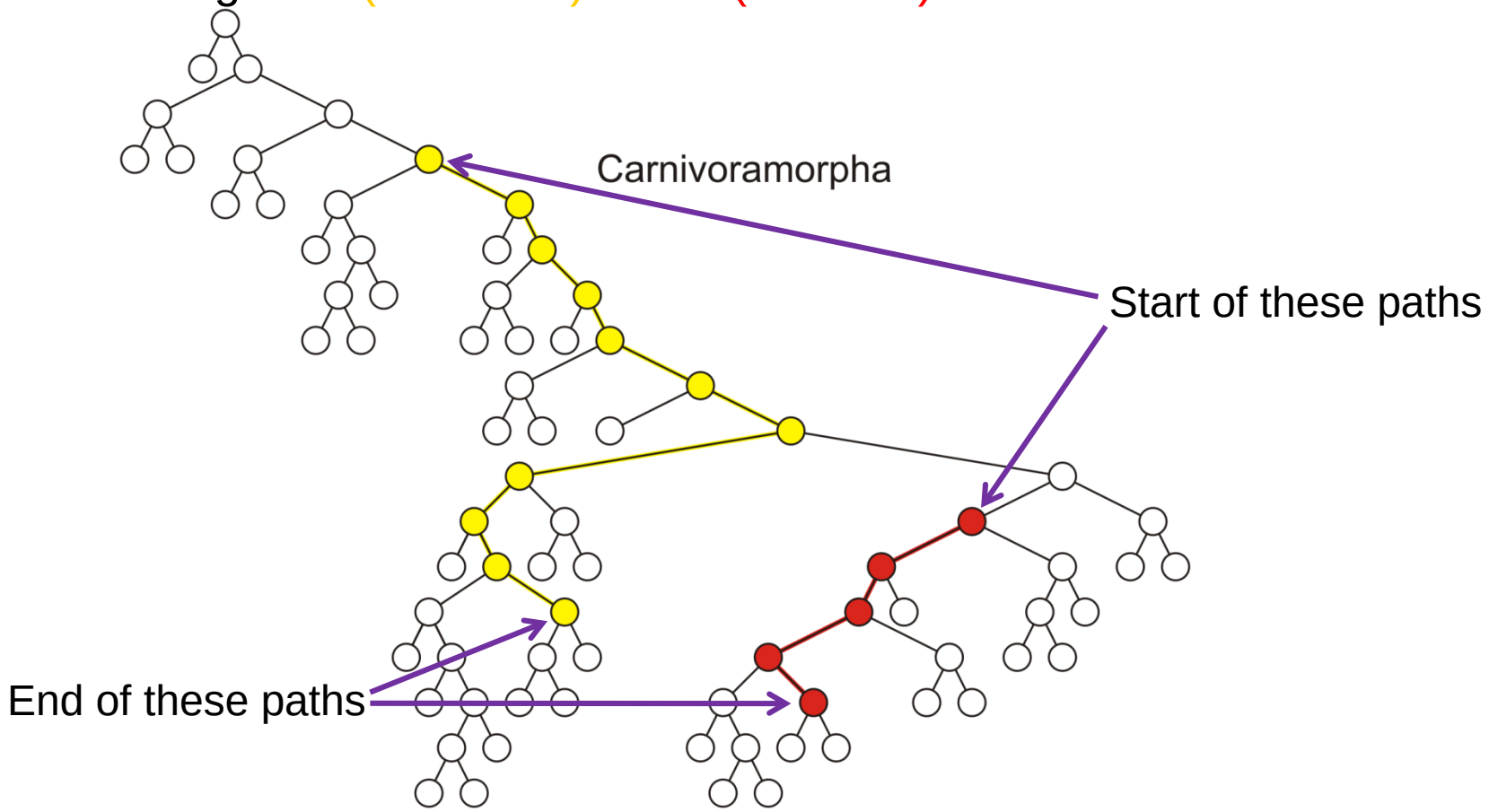
The length of this path is n

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



Terminology

Paths of length 10 (11 nodes) and 4 (5 nodes)



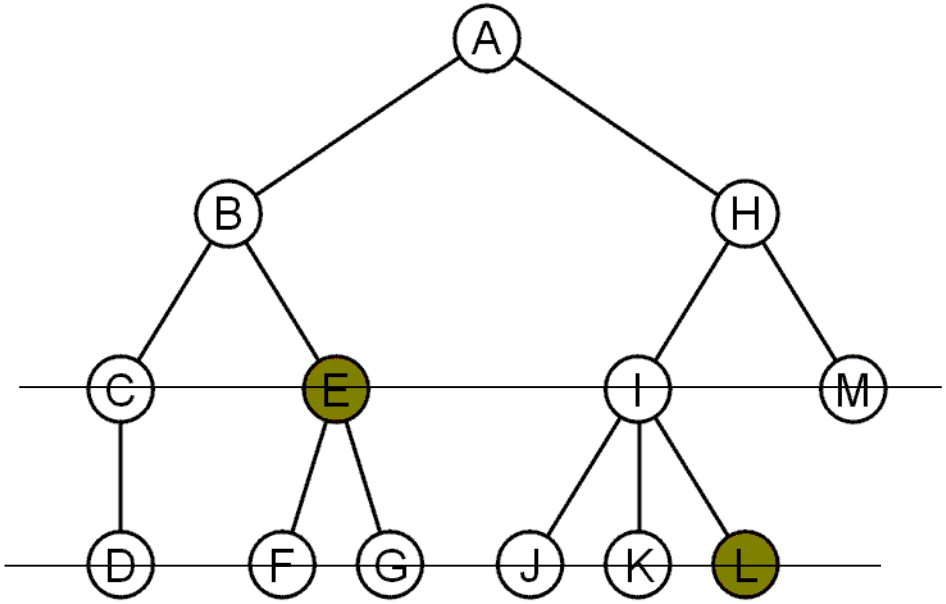
Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Terminology

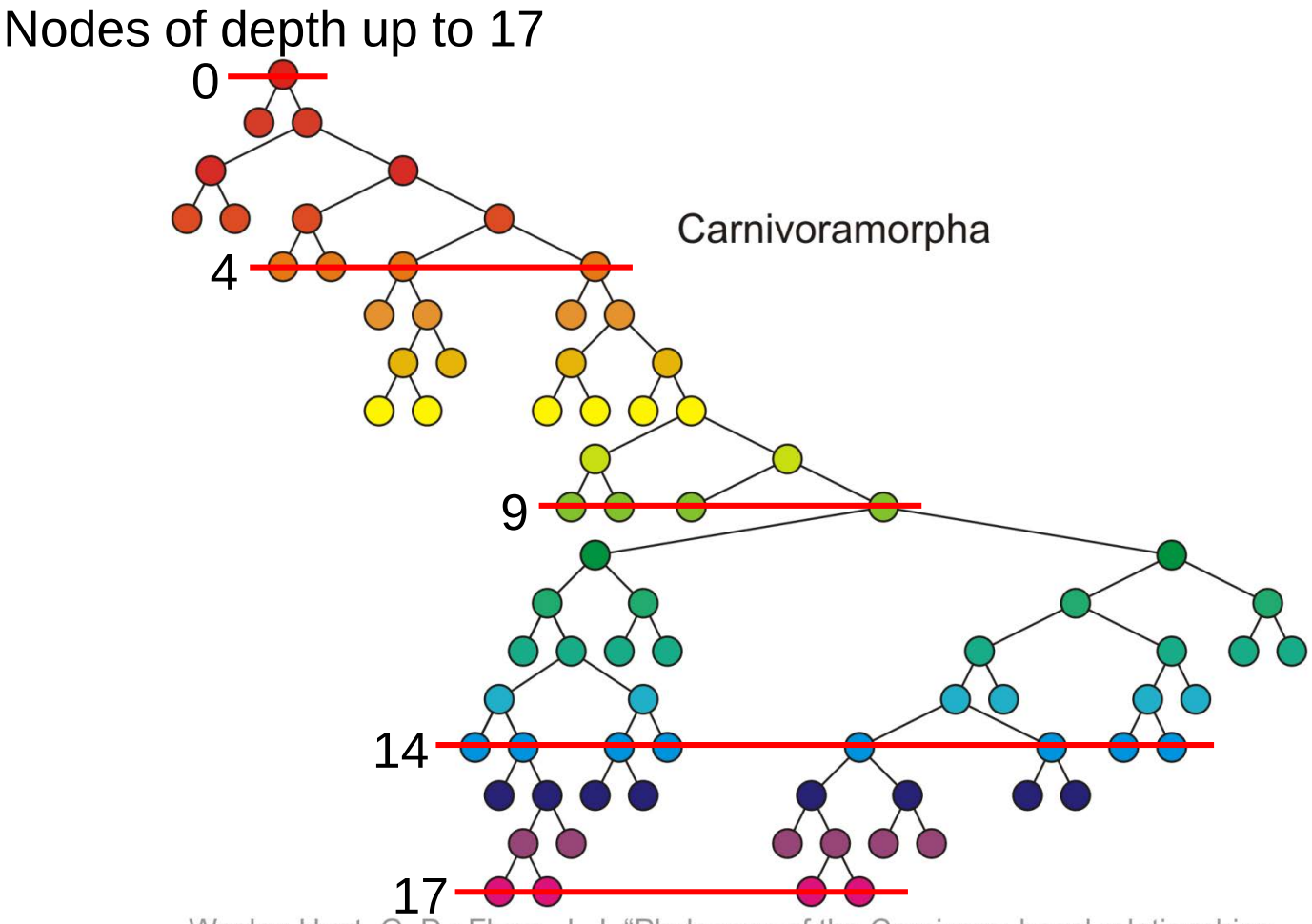
For each node in a tree, there exists a unique path from the root node to that node

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

- E has depth 2
- L has depth 3



Terminology



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Terminology

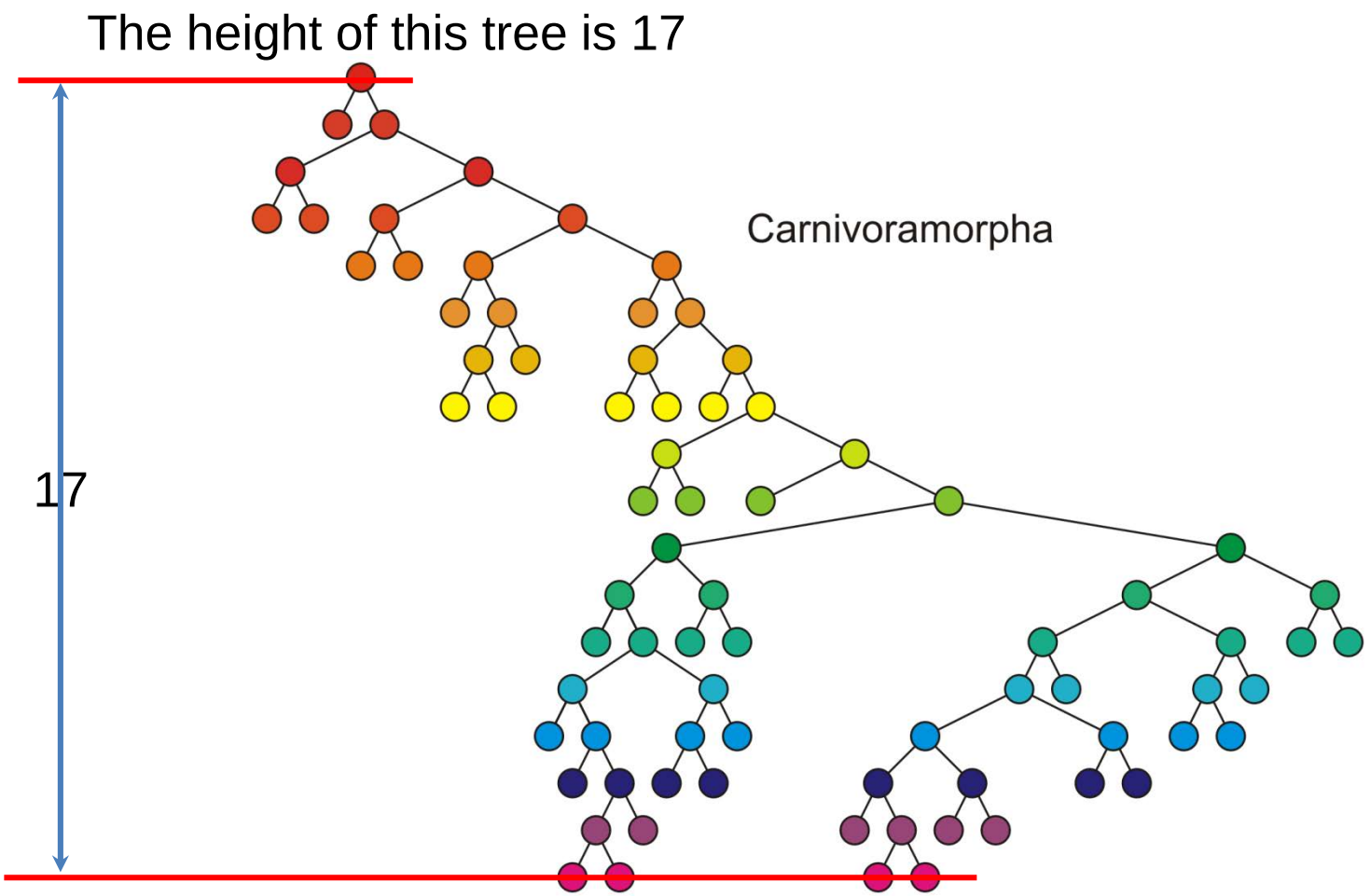
The *height* of a tree is defined as the maximum depth of any node within the tree

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

Terminology



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Terminology

If a path exists from node a to node b:

- a is an *ancestor* of b
- b is a *descendant* of a

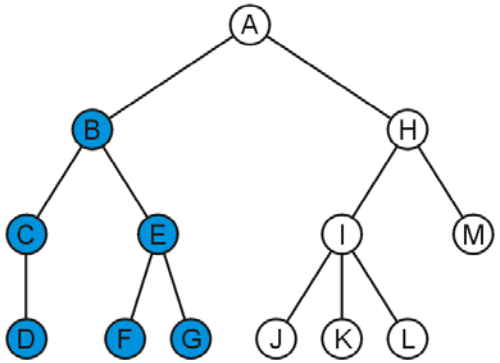
Thus, a node is both an ancestor and a descendant of itself

- We can add the adjective *strict* to exclude equality: a is a *strict descendant* of b if a is a descendant of b but $a \neq b$

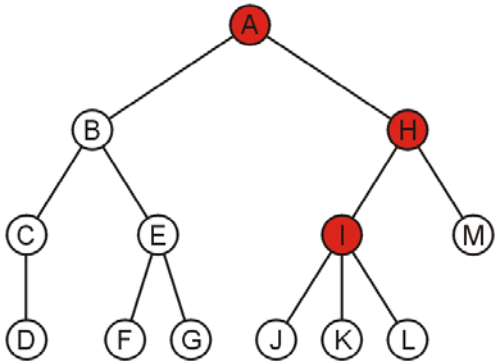
The root node is an ancestor of all nodes

Terminology

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

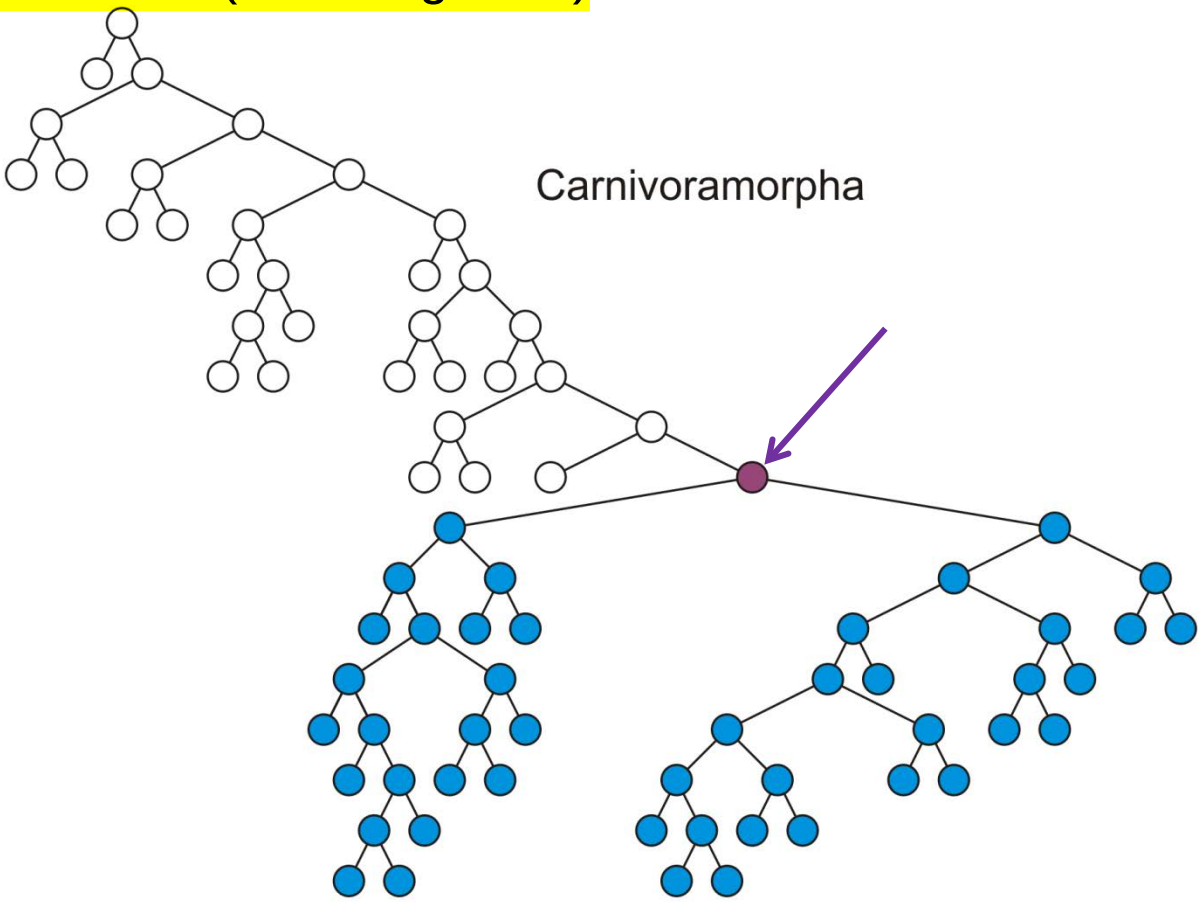


The ancestors of node I are I, H, and A:



Terminology

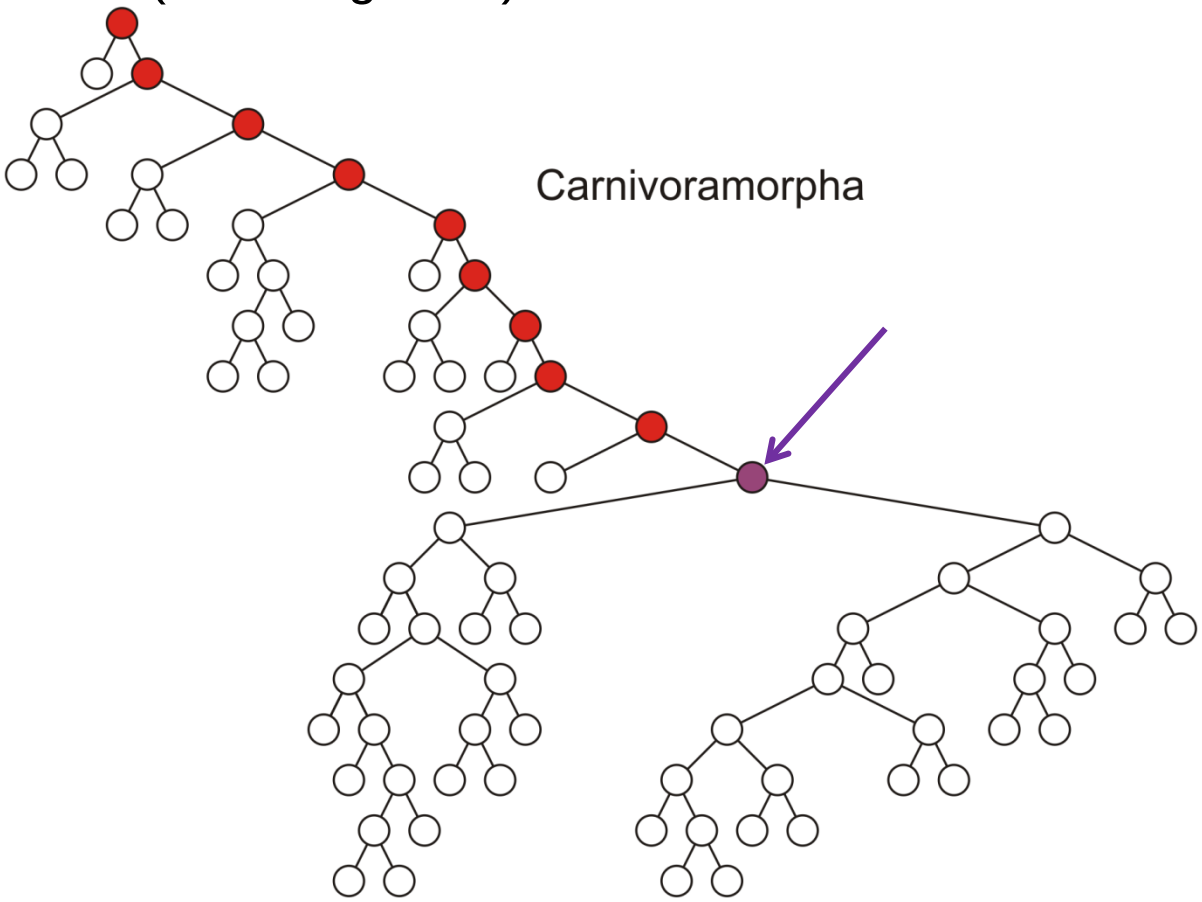
All descendants (including itself) of the indicated node



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Terminology

All ancestors (including itself) of the indicated node



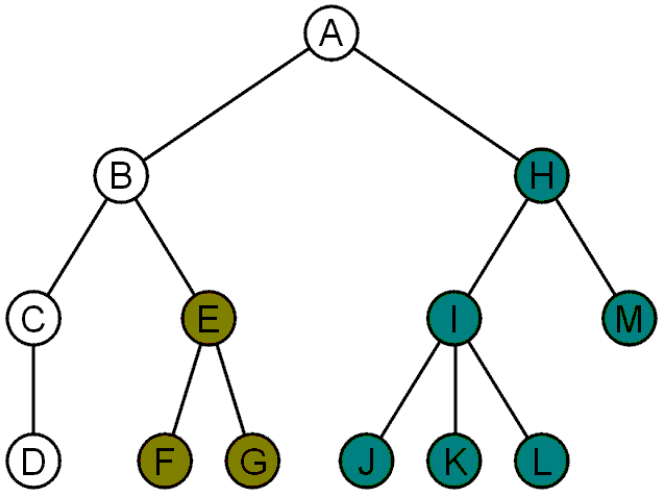
Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Terminology

A recursive definition of a tree:

- A degree-0 node is a tree
- A node with degree n is a tree if it has n children and all of its children are disjoint trees (*i.e.*, with no intersecting nodes)

Given any node a within a tree, the collection of a and all of its descendants is said to be a *subtree of the tree with root* a



Outline

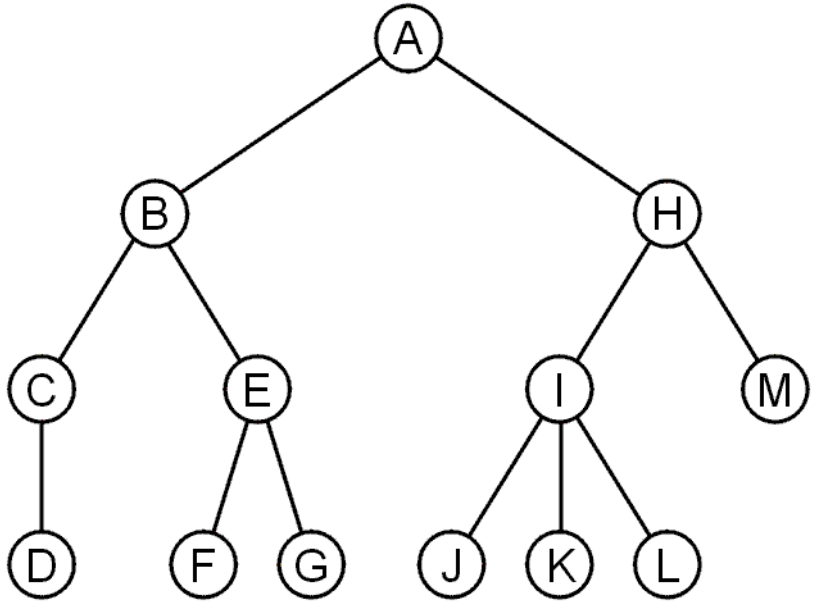
- Tree structure
- Implementation
- Tree traversal
- Forest

General Trees

An abstract tree does not restrict the number of nodes

- In this tree, the degrees vary:

Degree	Nodes
0	D, F, G, J, K, L, M
1	C
2	B, E, H
3	I



Operations

Operations on a tree include:

- Accessing the root:
- Given an object in the container:
  - Access the parent of the current object
  - Find the degree of the current object
  - Get a reference to a child,
  - Attach a new sub-tree to the current object
  - Detach this tree from its parent

Implementation

We can implement a general tree by using a class which:

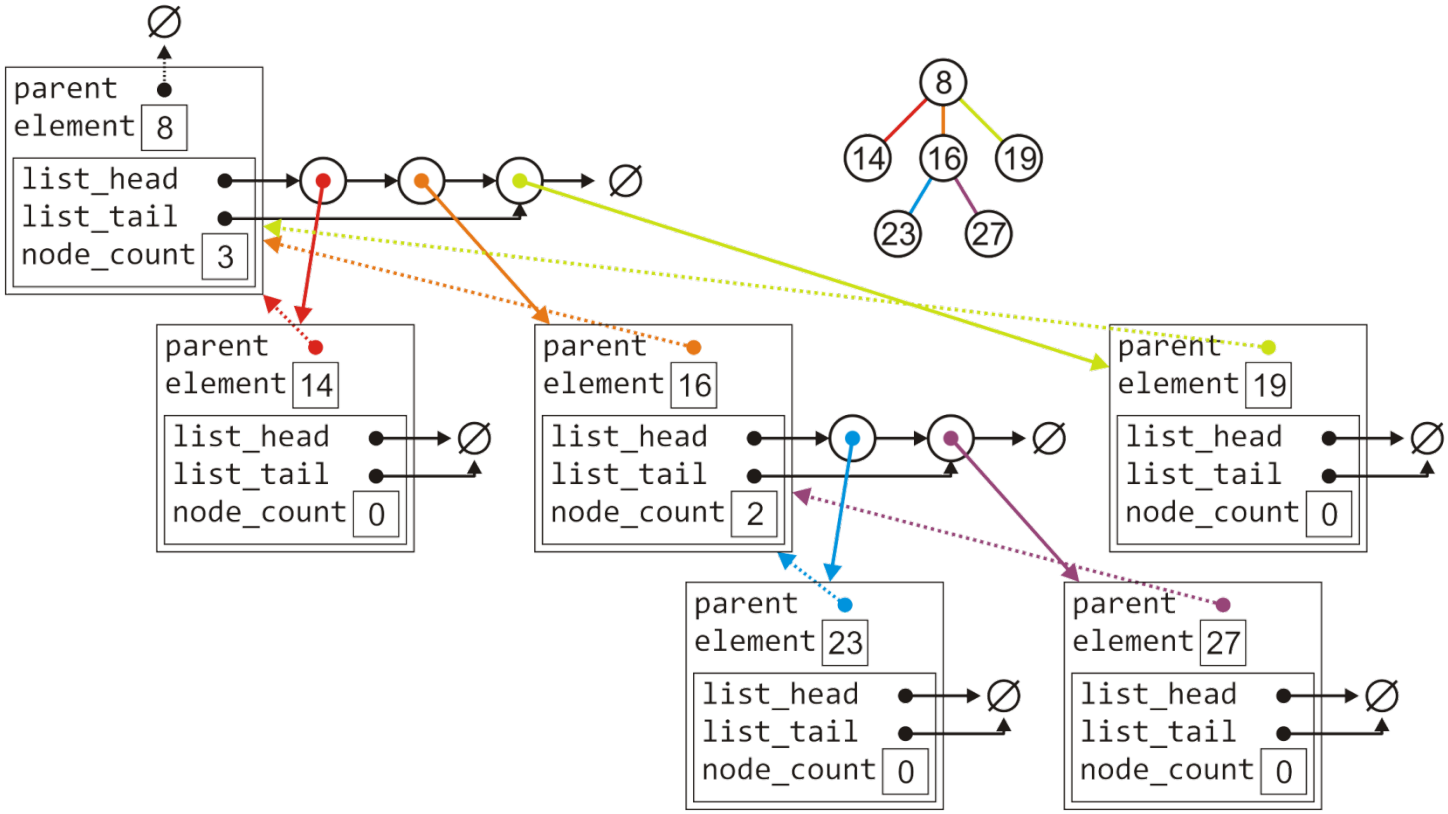
- Stores an element
- Stores the children in a list

Implementation

```
template <typename Type>
class Simple_tree {
    private:
        Type element;
        Simple_tree *parent_node;
        Single_list<Simple_tree *> children;

    public:
        Simple_tree( Type const & = Type(), Simple_tree * = nullptr );

        Type retrieve() const;
        Simple_tree *parent() const;
        int degree() const;
        bool is_root() const;
        bool is_leaf() const;
        Simple_tree *child( int n ) const;
        int height() const;

        void insert( Type const & );
        void attach( Simple_tree * );
        void detach();
};
```

Implementation

The tree with six nodes would be stored as follows:



Implementation

Much of the functionality is similar to that of the `Single_list` class:

```
template <typename Type>
Simple_tree<Type>::Simple_tree( Type const &obj, Simple_tree *p ):
element( obj ),
parent_node( p ) {
    // Empty constructor
}

template <typename Type>
Type Simple_tree<Type>::retrieve() const {
    return element;
}

template <typename Type>
Simple_tree<Type> *Simple_tree<Type>::parent() const {
    return parent_node;
}
```

Implementation

Much of the functionality is similar to that of the `Single_list` class:

```
template <typename Type>
bool Simple_tree<Type>::is_root() const {
    return ( parent() == nullptr );
}

template <typename Type>
int Simple_tree<Type>::degree() const {
    return children.size();
}

template <typename Type>
bool Simple_tree<Type>::is_leaf() const {
    return ( degree() == 0 );
}
```

Implementation

Accessing the n^{th} child requires a for loop ($\Theta(n)$):

```
template <typename Type>
Simple_tree<Type> *Simple_tree<Type>::child( int n ) const {
    if ( n < 0 || n >= degree() ) {
        return nullptr;
    }

    Single_node<Simple_tree *> *ptr = children.head();

    for ( int i = 1; i < n; ++i ) {
        ptr = ptr->next();
    }

    return ptr->retrieve();
}
```

Implementation

Attaching a new object to become a child is similar to a linked list:

```
template <typename Type>
void Simple_tree<Type>::attach( Type const &obj ) {
    children.push_back( new Simple_tree( obj, this ) );
}
```

Implementation

To detach a tree from its parent:

- If it is already a root, do nothing
- Otherwise, erase this object from the parent's list of children and set the parent pointer to NULL

```
template <typename Type>
void Simple_tree<Type>::detach() {
    if ( is_root() ) {
        return;
    }

    parent()->children.erase( this );
    parent_node = nullptr;
}
```

Implementation

Attaching an entirely new tree as a sub-tree, however, first requires us to check if the tree is not already a sub-tree of another node:

- If so, we must detach it first and only then can we add it

```
template <typename Type>
void Simple_tree<Type>::attach( Simple_tree<Type> *tree ) {
    if ( !tree->is_root() ) {
        tree->detach();
    }

    tree->parent_node = this;
    children.push_back( tree );
}
```

Implementation

Suppose we want to find the size of a tree:

- If there are no children, the size is 1
- Otherwise, the size is one plus the size of all the children

```
template <typename Type>
int Simple_tree<Type>::size() const {
    int s = 1;

    for (
        Single_node<Simple_tree *> *ptr = children.head();
        ptr != nullptr;
        ptr = ptr->next()
    ) {
        s += ptr->retrieve()->size();
    }

    return s;
}
```

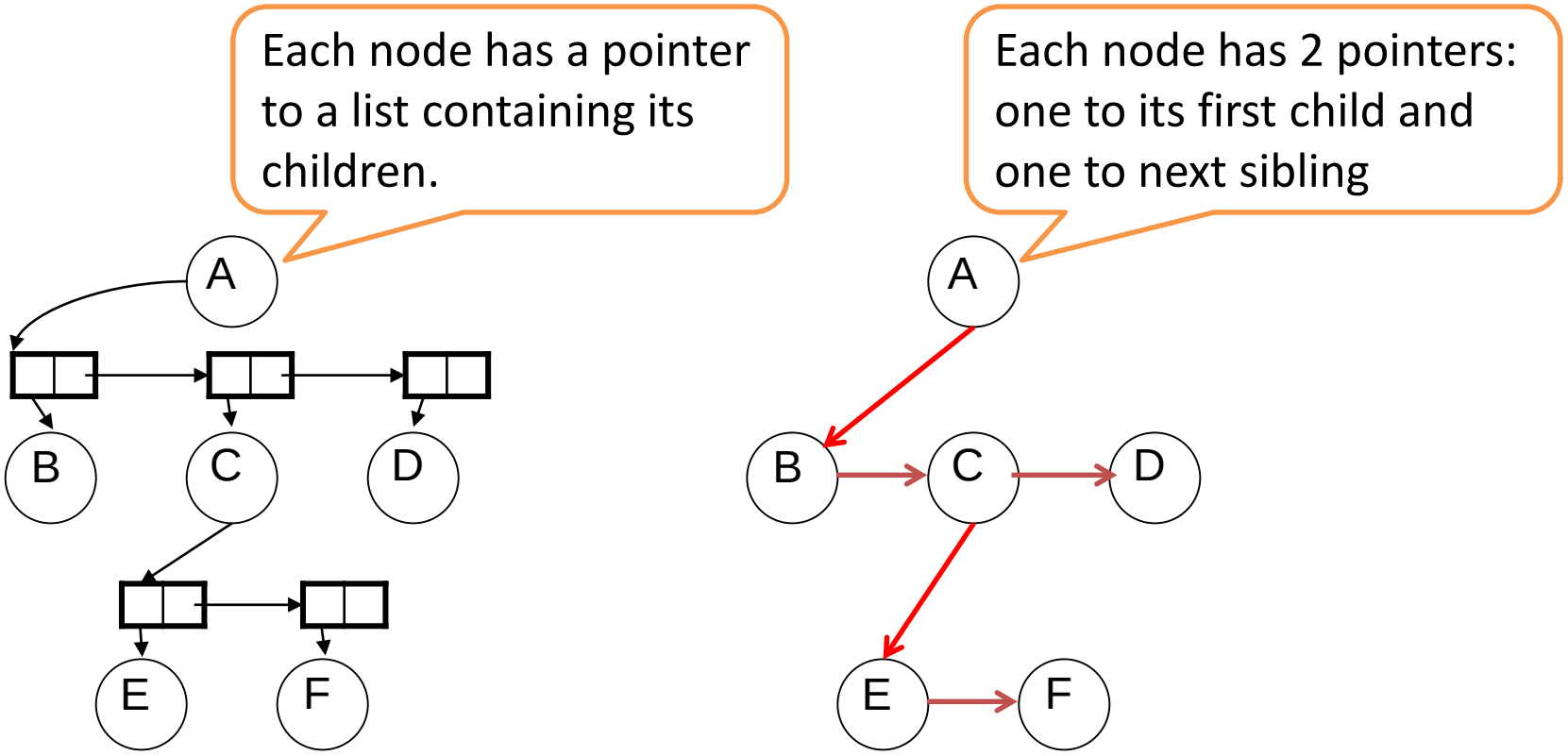
Implementation

Suppose we want to find the height of a tree:

- If there are no children, the height is 0
- Otherwise, the height is one plus the maximum height of any sub tree

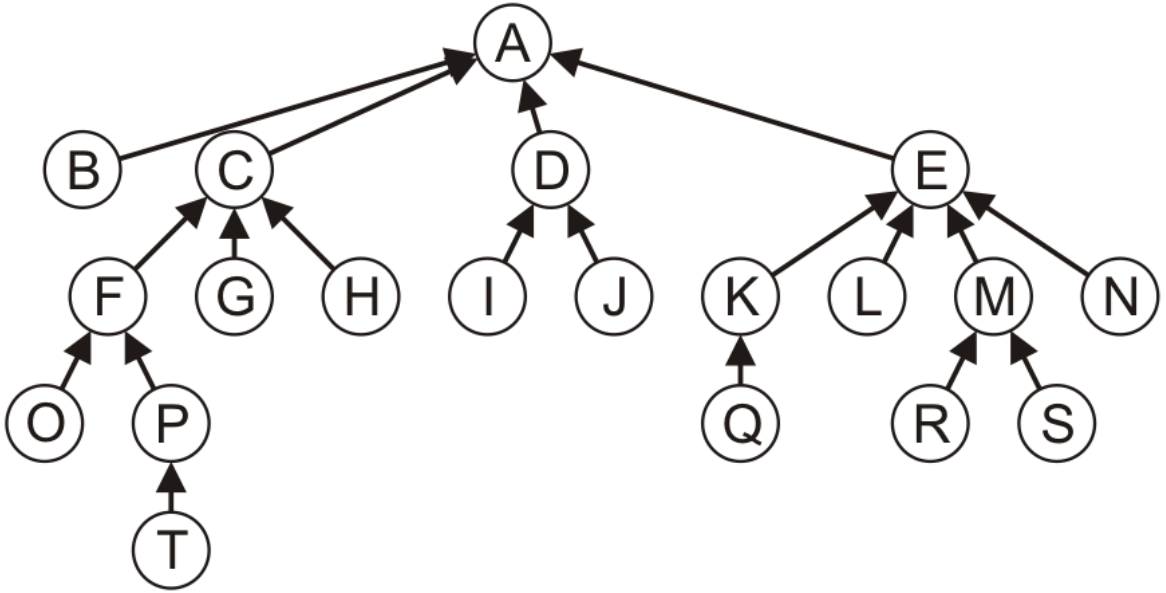
```
template <typename Type>
int Simple_tree<Type>::height() const {
    int h = 0;

    for (
        Single_node<Simple_tree *> *ptr = children.head();
        ptr != nullptr;
        ptr = ptr->next()
    ) {
        h = std::max( h, 1 + ptr->retrieve()->height() );
    }

    return h;
}
```

Alternative Implementation



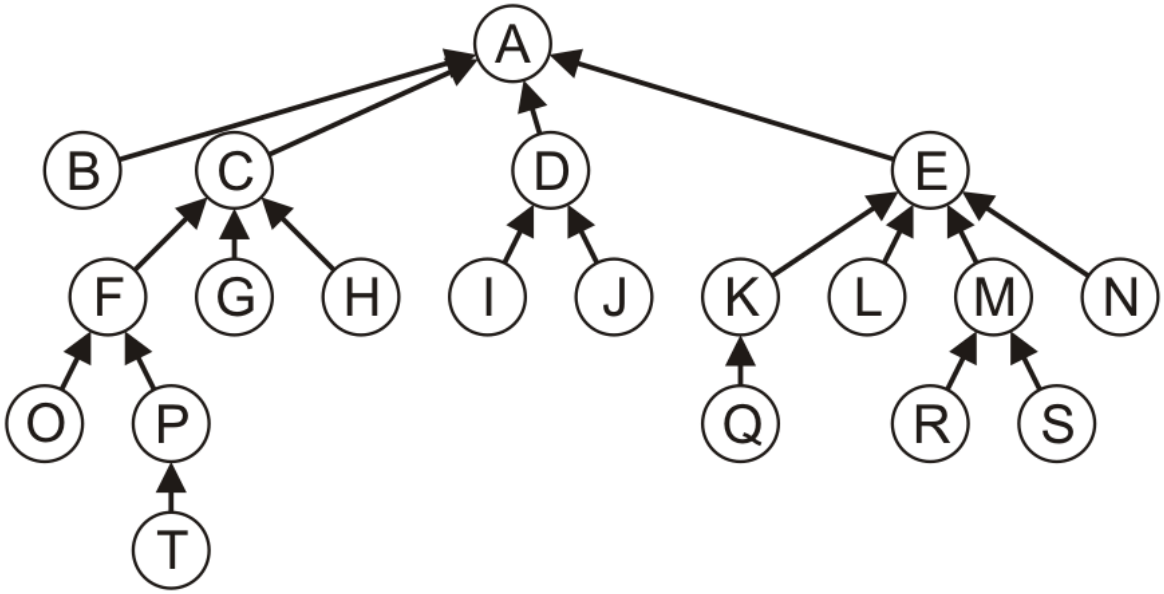
Parent Pointer Tree

A *parent pointer tree* is a tree where each node only keeps a reference to its parent node



Parent Pointer Tree

This requires significantly less memory than our general tree structure, as no data structure is required to track the children



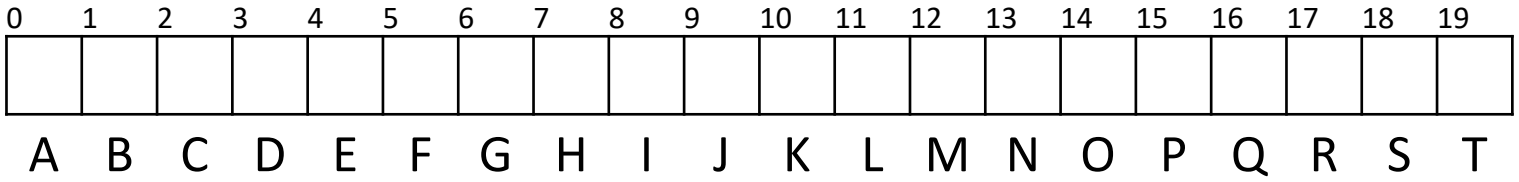
Implementation

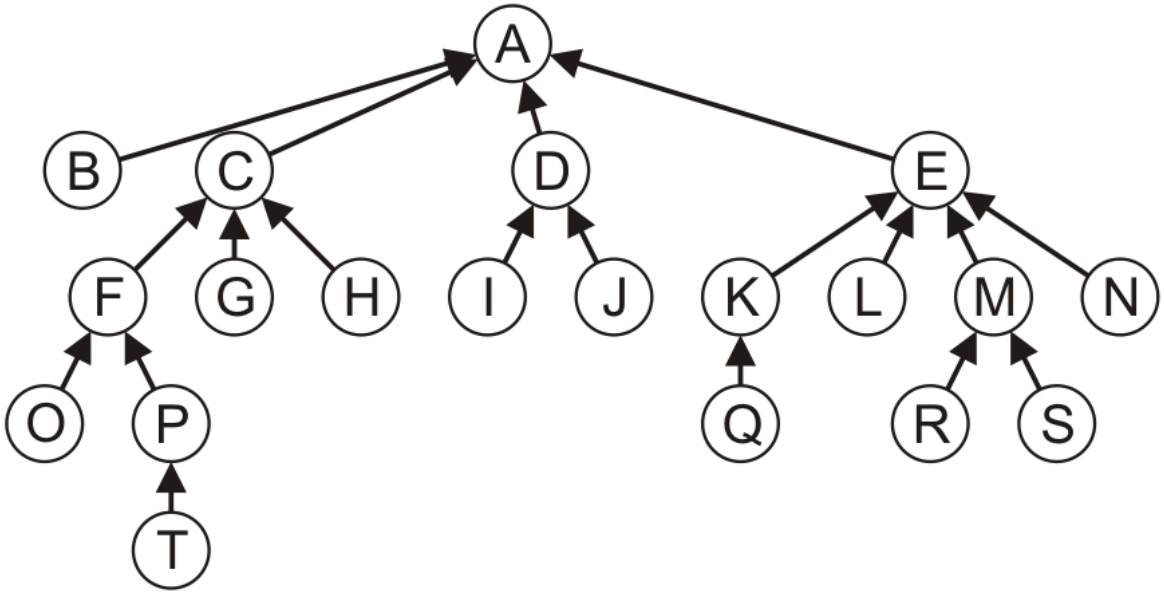
A naïve implementation may also be node based:

```
template <typename Type>
class Parental_tree {
    private:
        Type element;
        Parental_tree *parent;
    public:
        // ...
};
```

Implementation

Instead, generate an array of size n and associate each entry with a node in the tree



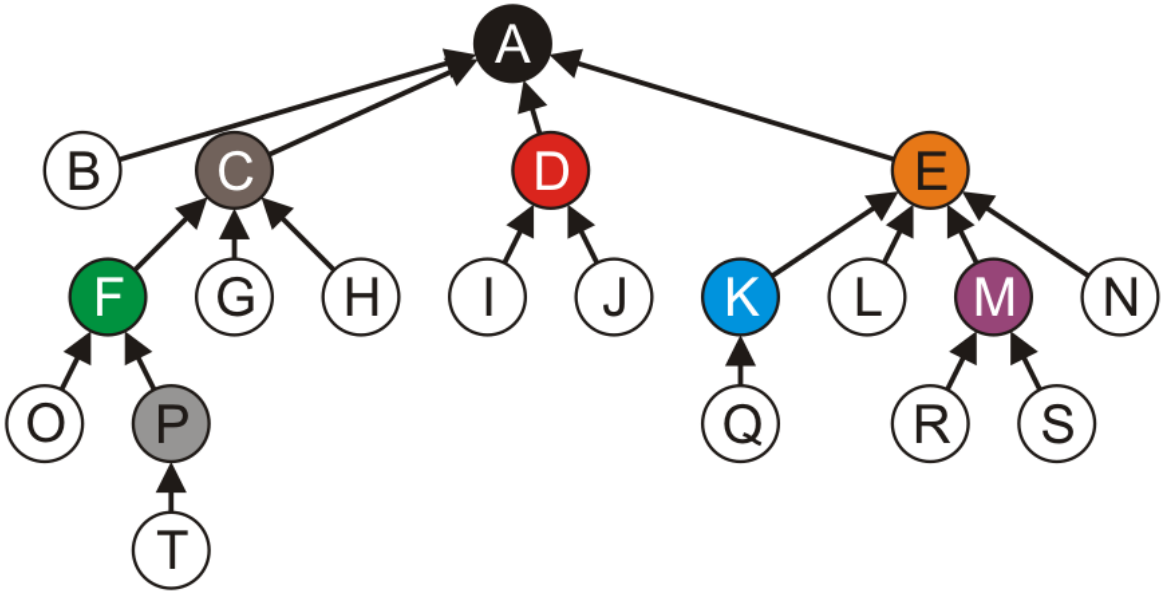


Implementation

Store the index of the parent in each node

– The root node, wherever it is, points to itself

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	0	0	0	0	2	2	2	3	3	4	4	4	4	5	5	10	12	12	15
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T



Outline

- Tree structure
- Implementation
- Tree traversal
- Forest

Tree Traversals

Question: how can we iterate through all the objects in a tree in a predictable and efficient manner

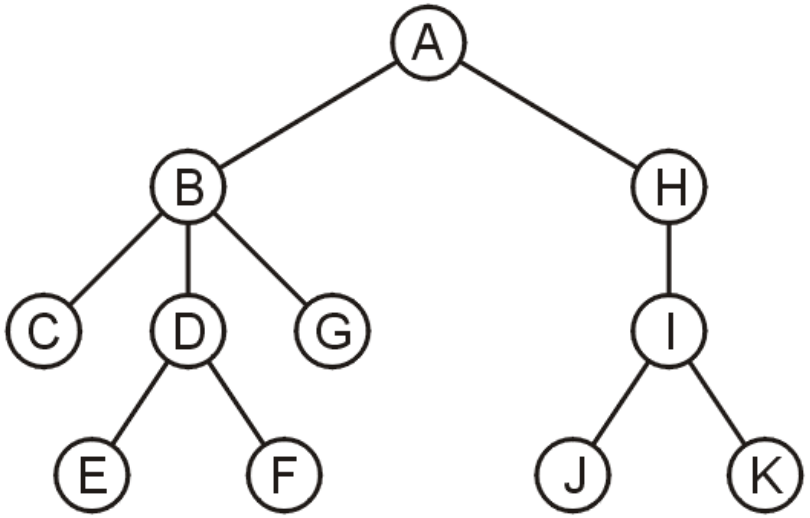
- Requirements: $\Theta(n)$ run time and $o(n)$ memory

Two types of traversals

- Breadth-first traversal
- Depth-first traversal

Breadth-First Traversal

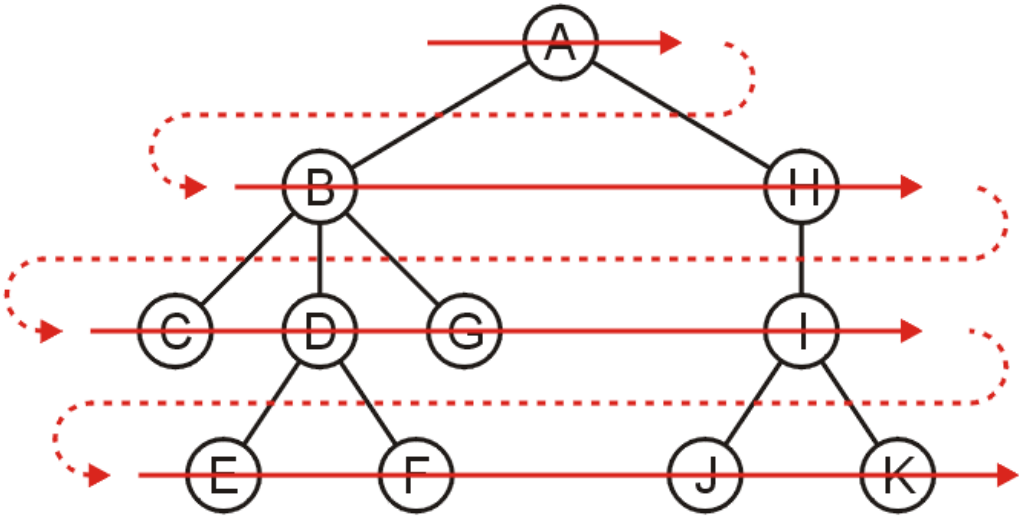
Breadth-first traversals visit all nodes at a given depth before descending a level



Breadth-First Traversal

Breadth-first traversals visit all nodes at a given depth before descending a level

- Order: A B H C D G I E F J K



Breadth-First Traversal

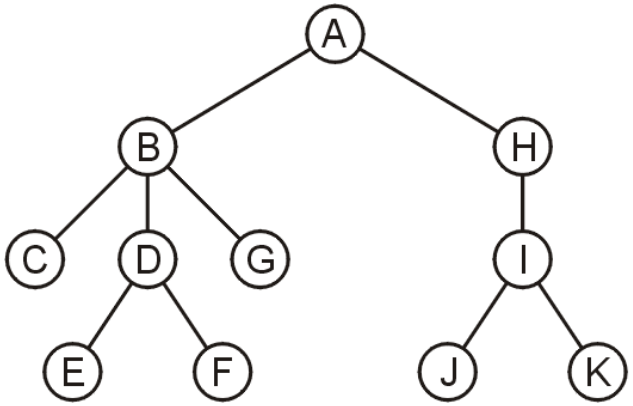
The easiest implementation is to use a queue:

- Place the root node into a queue
- While the queue is not empty:
  - Pop the node at the front of the queue
  - Push all of its children into the queue

The order in which the nodes come out of the queue will be in breadth-first order

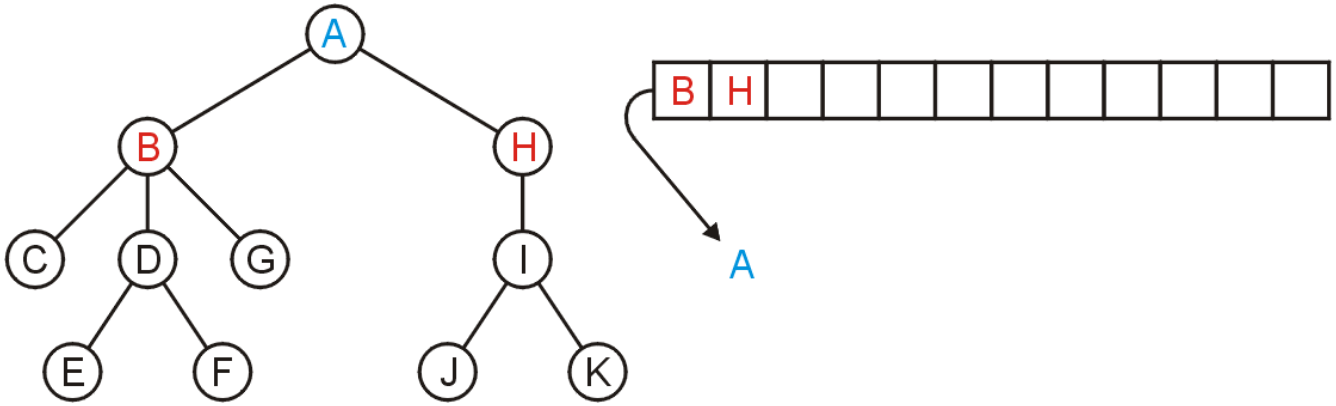
Breadth-First Traversal

Push the root directory A



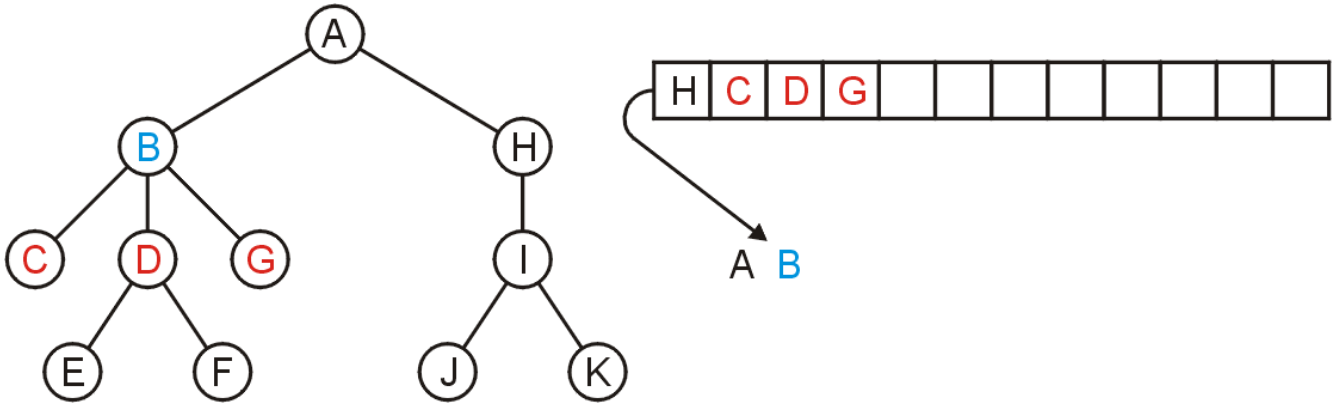
Breadth-First Traversal

Pop A and push its two sub-directories: B and H



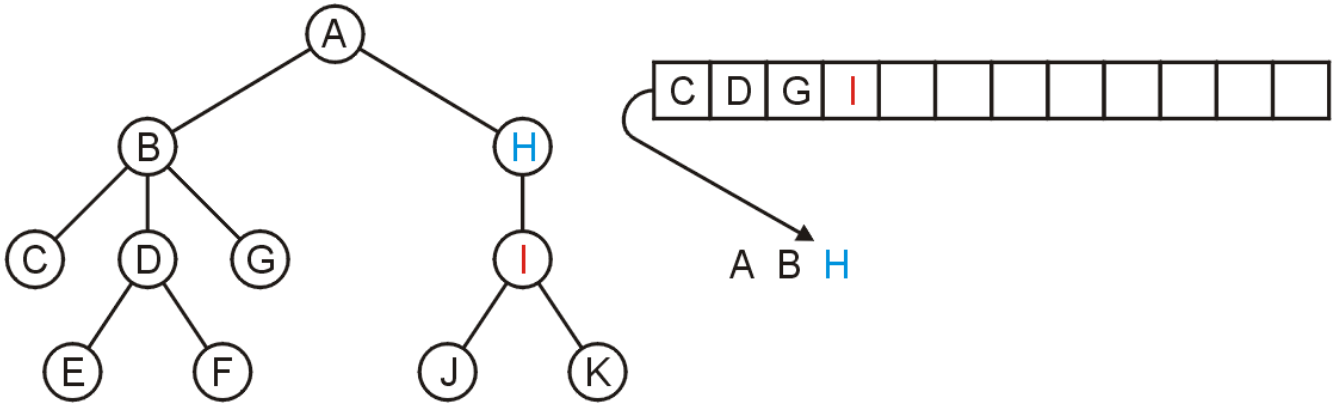
Breadth-First Traversal

Pop B and push C, D, and G



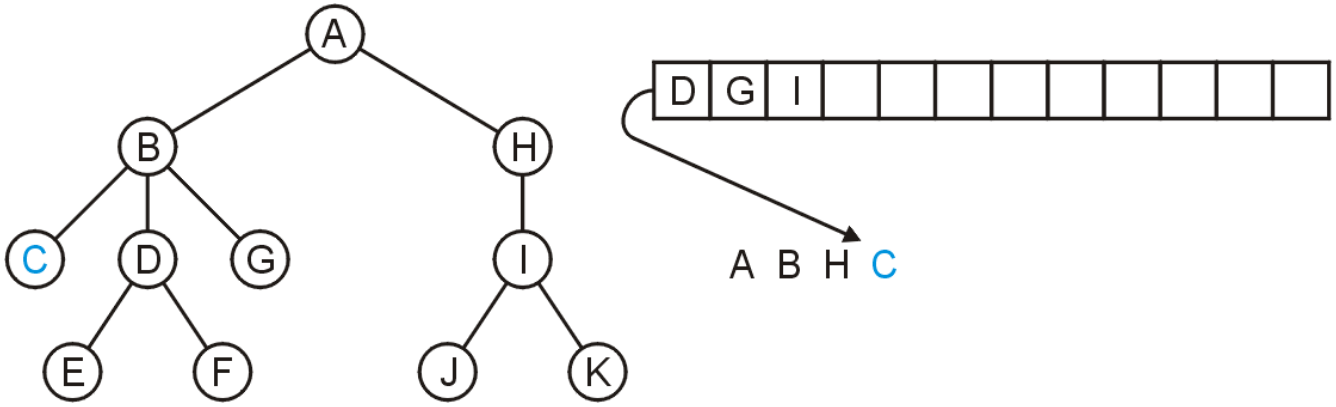
Breadth-First Traversal

Pop H and push its one sub-directory I



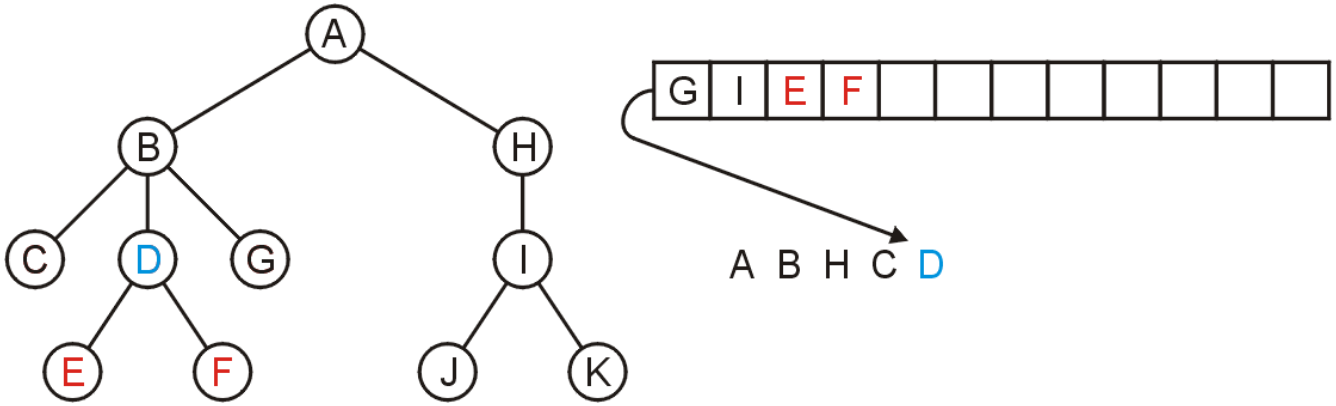
Breadth-First Traversal

Pop C: no sub-directories



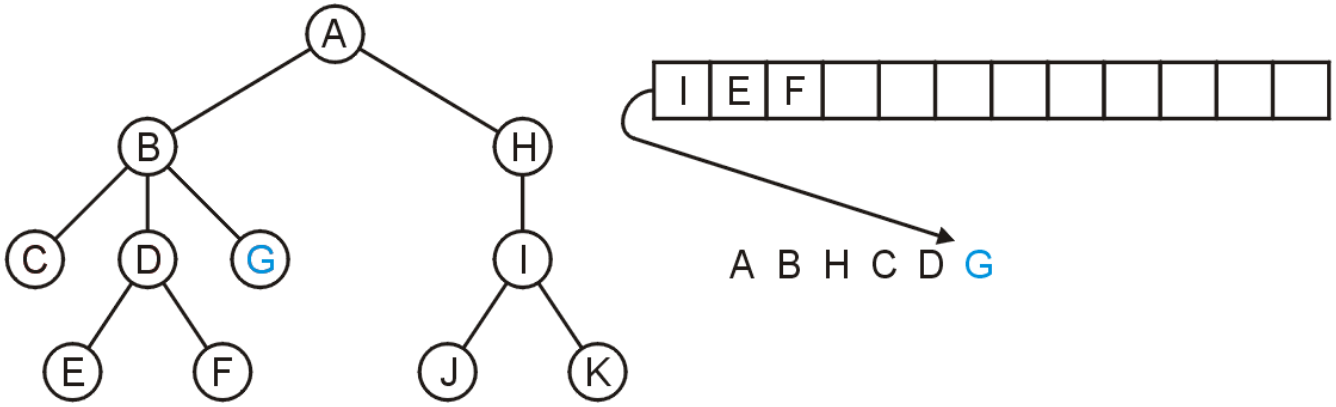
Breadth-First Traversal

Pop D and push E and F



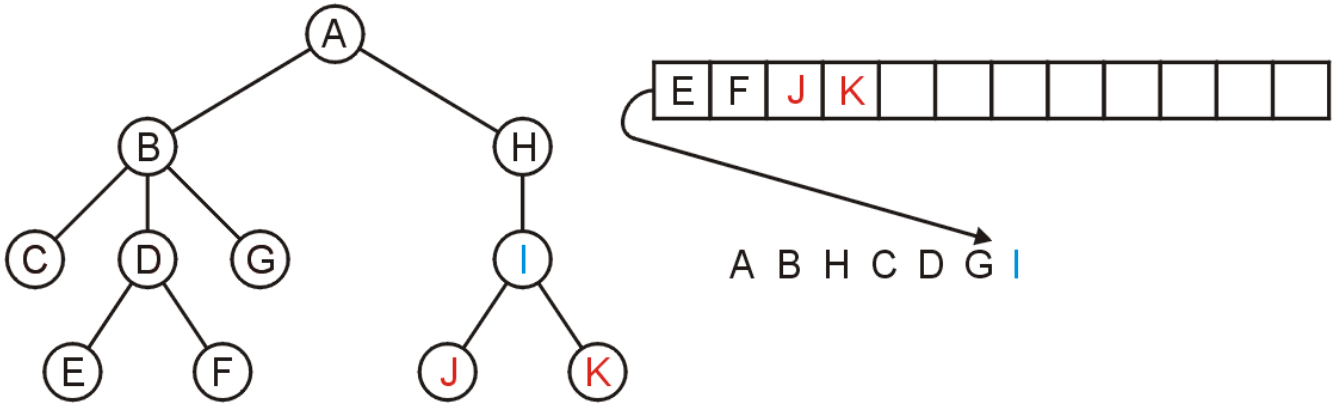
Breadth-First Traversal

Pop G



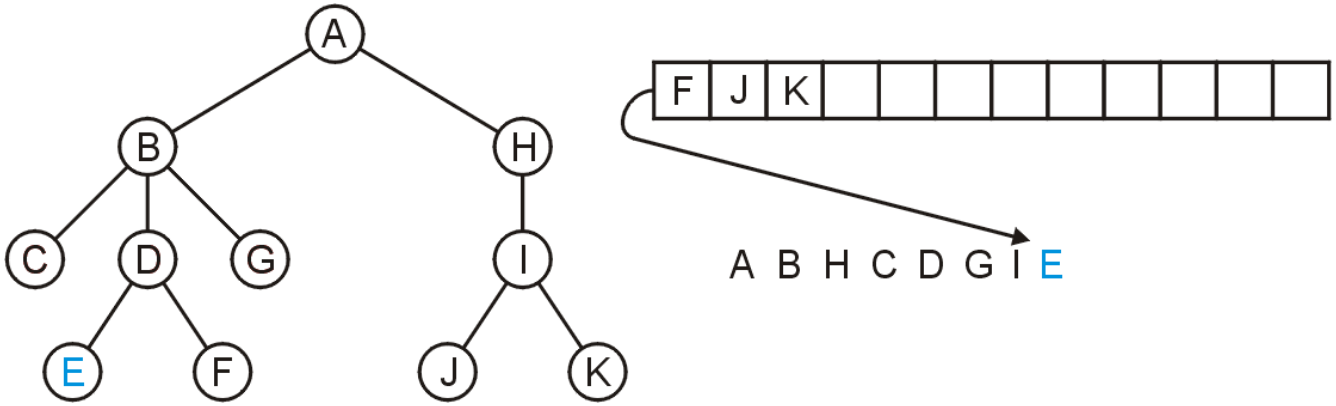
Breadth-First Traversal

Pop I and push J and K



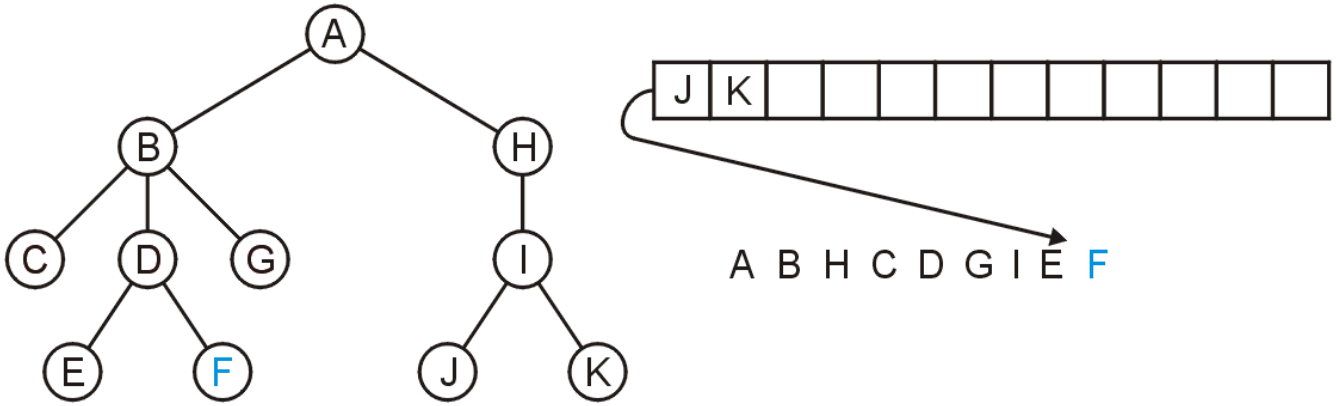
Breadth-First Traversal

Pop E



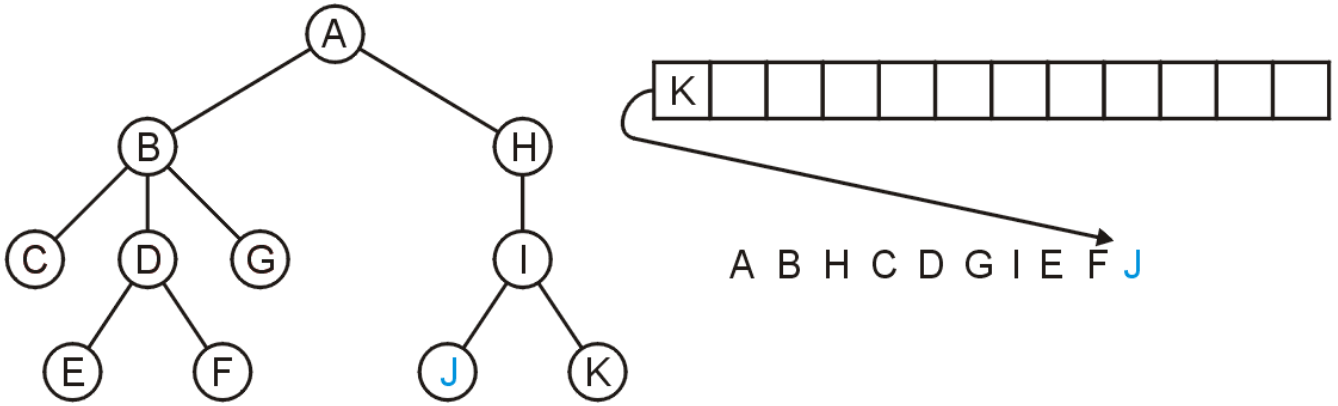
Breadth-First Traversal

Pop F



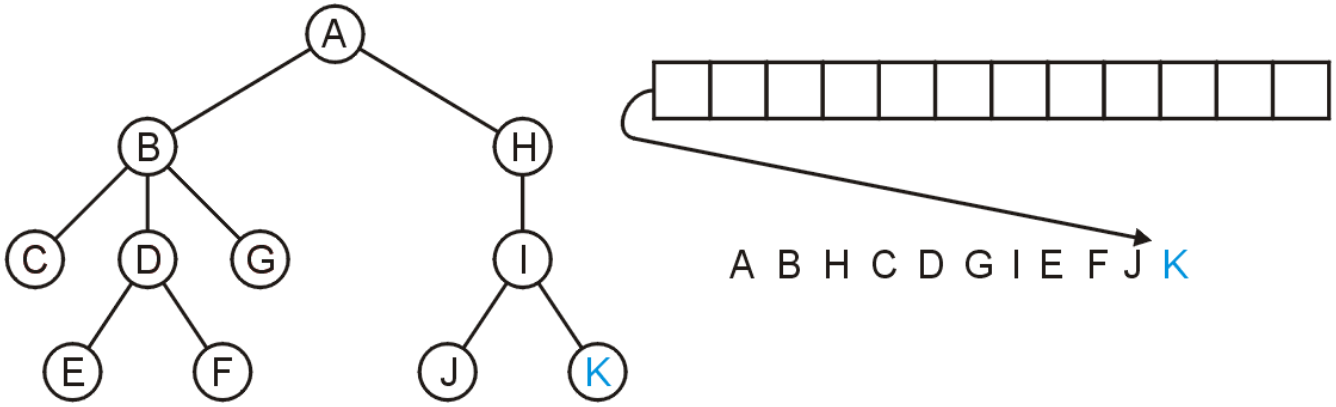
Breadth-First Traversal

Pop J



Breadth-First Traversal

Pop K and the queue is empty

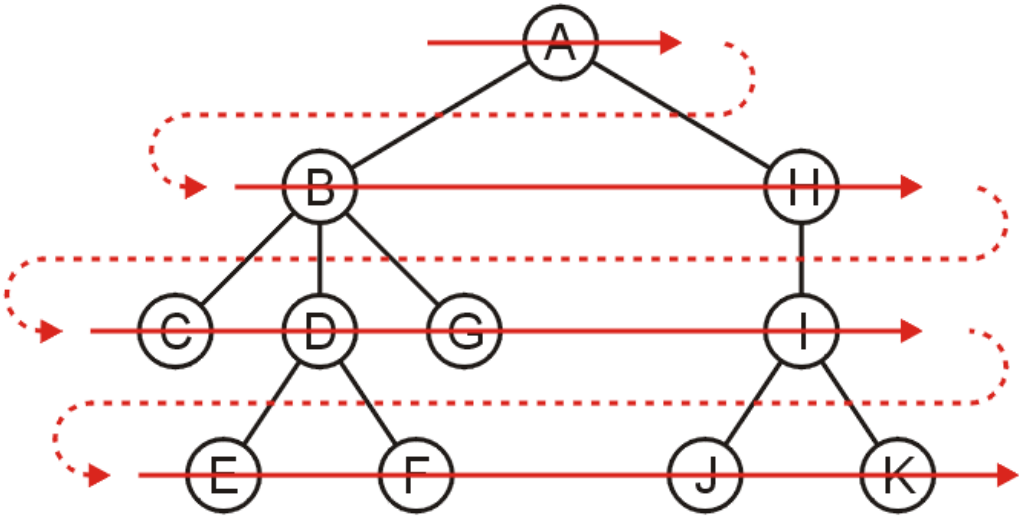


Breadth-First Traversal

The resulting order

A B H C D G I E F J K

is in breadth-first order:



Breadth-First Traversal

Computational complexity

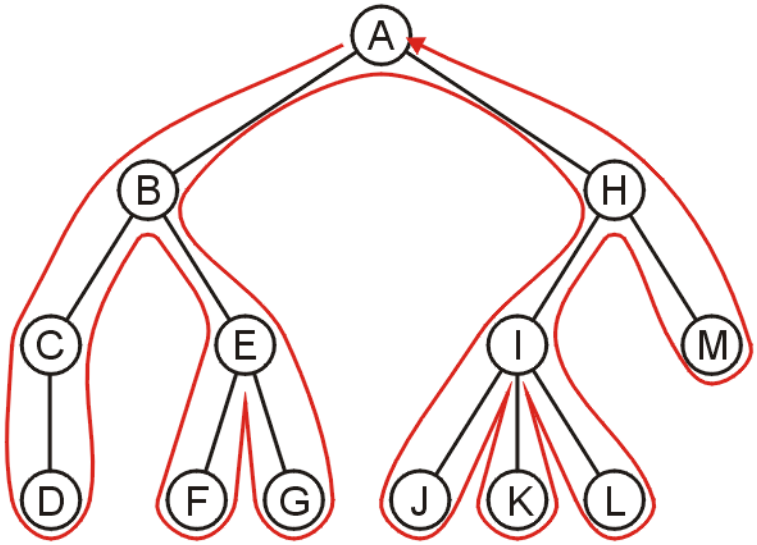
- Run time is $\Theta(n)$
- Space: maximum nodes at a given depth, $O(n)$

Depth-first Traversal

A backtracking algorithm for stepping through a tree:

- At any node, proceed to the first child that has not yet been visited
- If we have visited all the children (of which a leaf node is a special case), backtrack to the parent and repeat this process

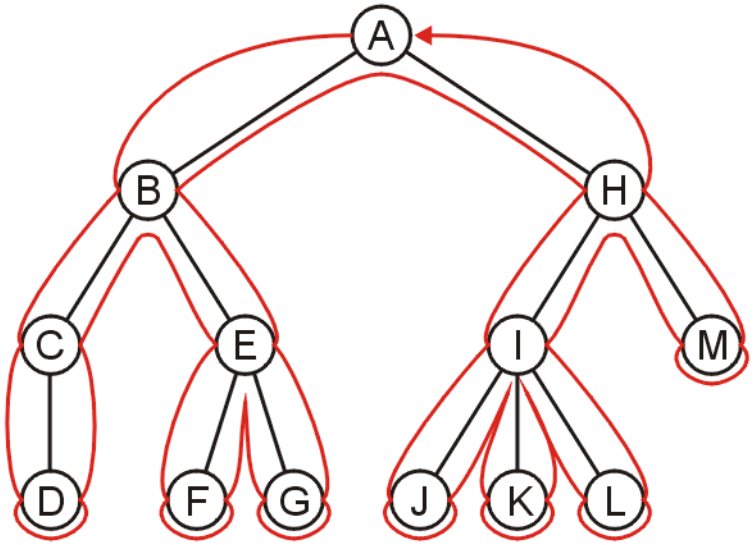
We end once all the children of the root are visited



Depth-first Traversal

Each node is visited multiple times in such a scheme

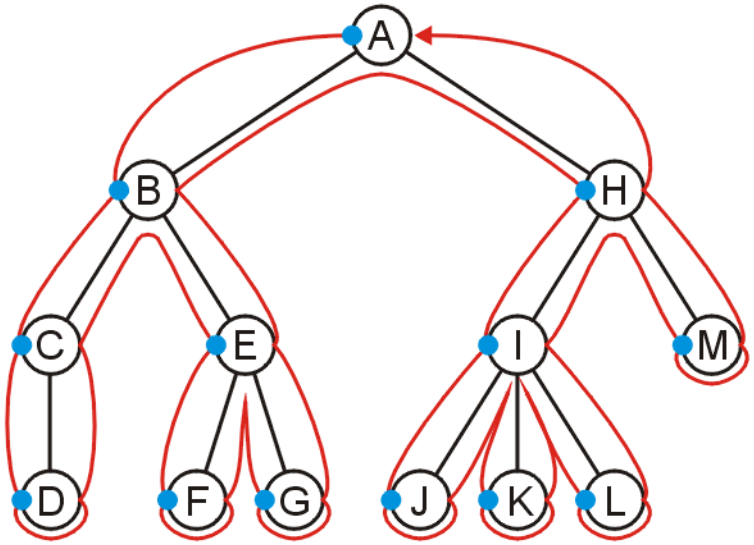
- First time: before any children
- Last time: after all children, before backtracking



Pre-ordering

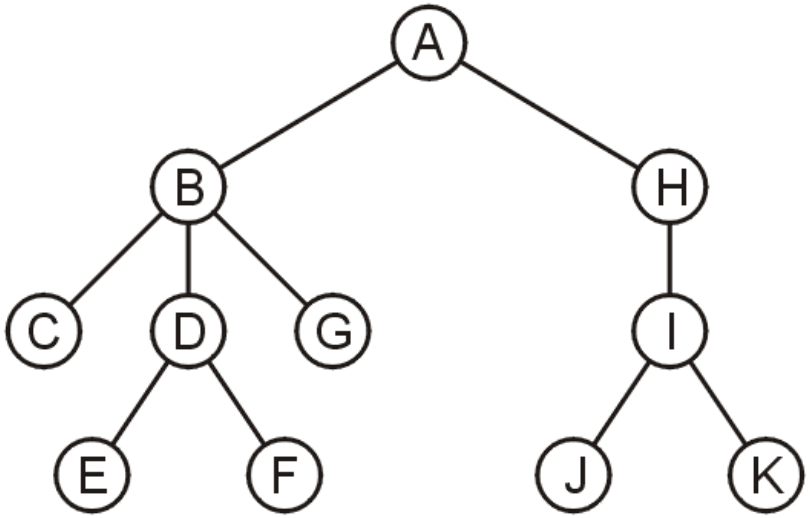
Ordering nodes by their first visits results in the sequence:

A, B, C, D, E, F, G, H, I, J, K, L, M



Exercise

- What is the preordering and postordering of the following tree?
  - Preordering:
  - Posterordering:



Implementing Depth-First Traversals

Implementation with recursion:

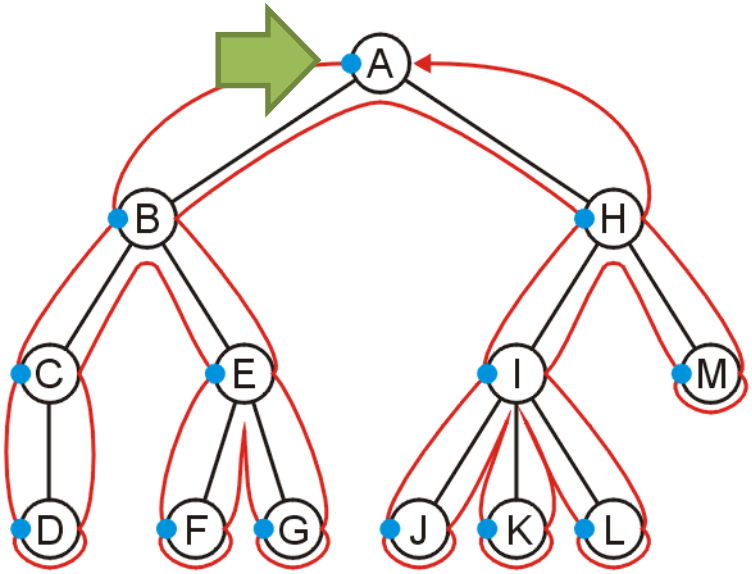
```
template <typename Type>
void Simple_tree<Type>::depth_first_traversal() const {
    // Perform pre-visit operations on the element
    std::cout << element << ' ';

    // Perform a depth-first traversal on each of the children
    for (
        Single_node<Simple_tree *> *ptr = children.head();
        ptr != 0; ptr = ptr->next()
    ) {
        ptr->retrieve()->depth_first_traversal();
    }

    // Perform post-visit operations on the element
    std::cout << element << ' ';
}
```

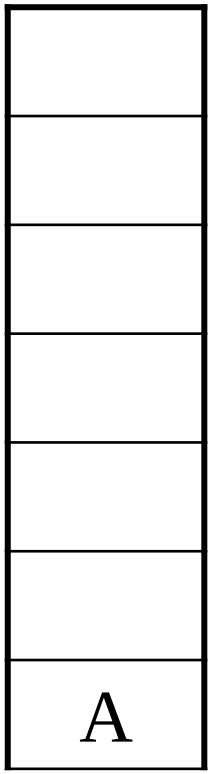
Implementing Depth-First Traversals

Alternatively, we can use a stack:

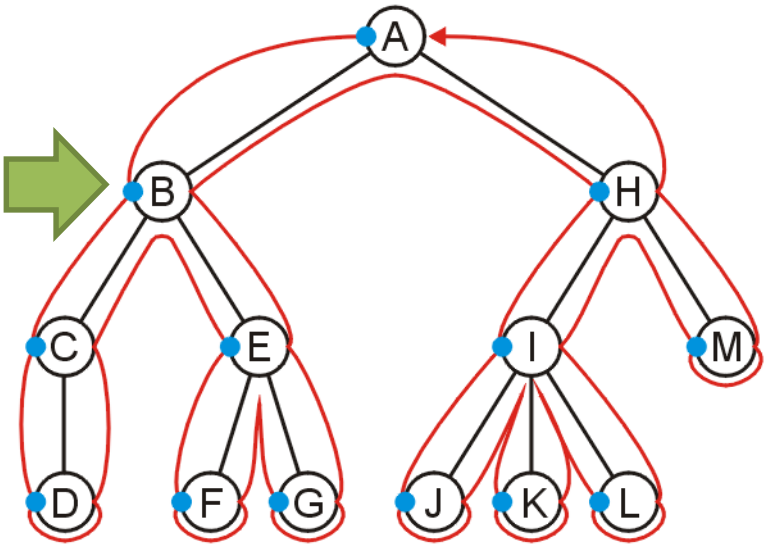
- Create a stack and push the root node onto the stack
- While the stack is not empty:
  - Pop the top node
  - Push all of the children of that node to the top of the stack in reverse order

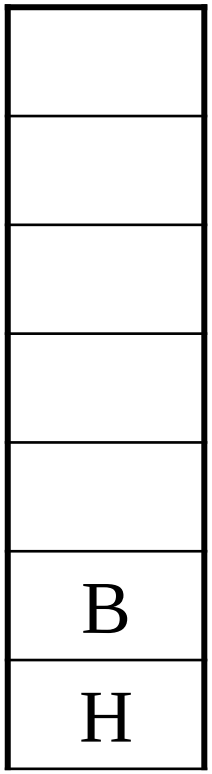
DFS using a Stack



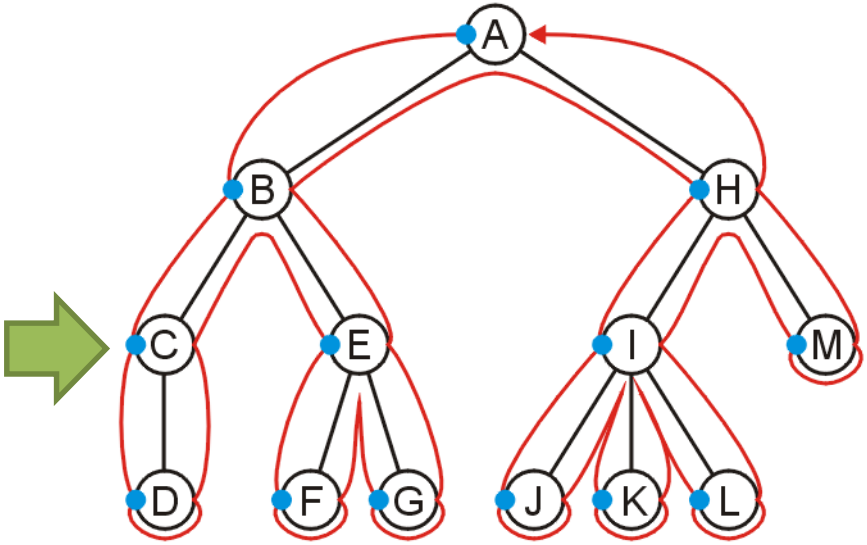


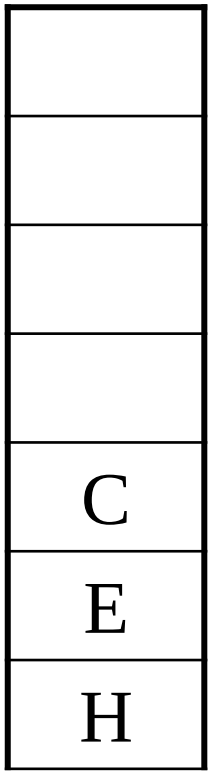
DFS using a Stack



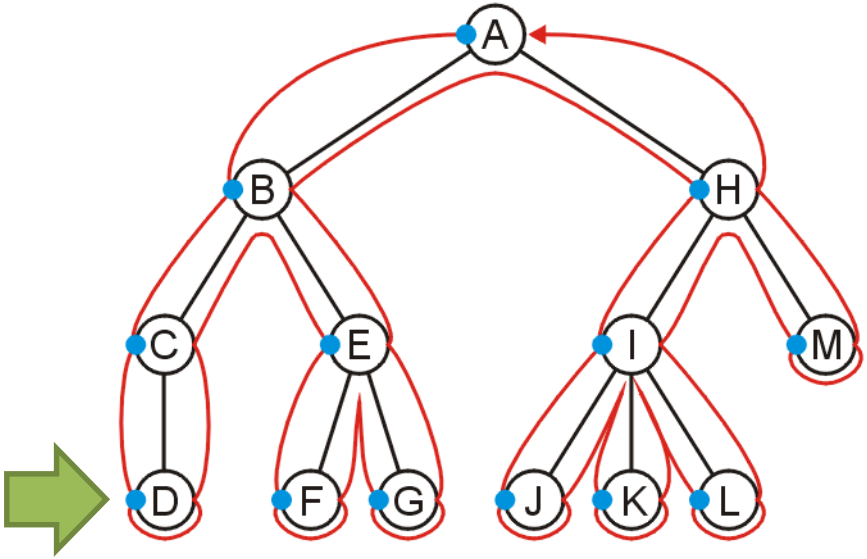


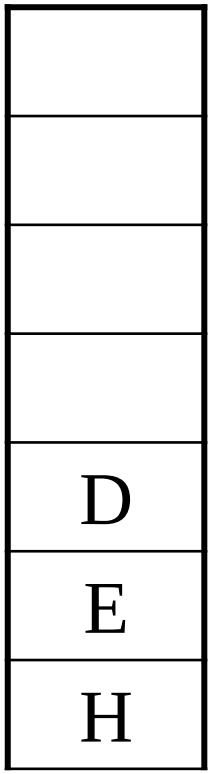
DFS using a Stack



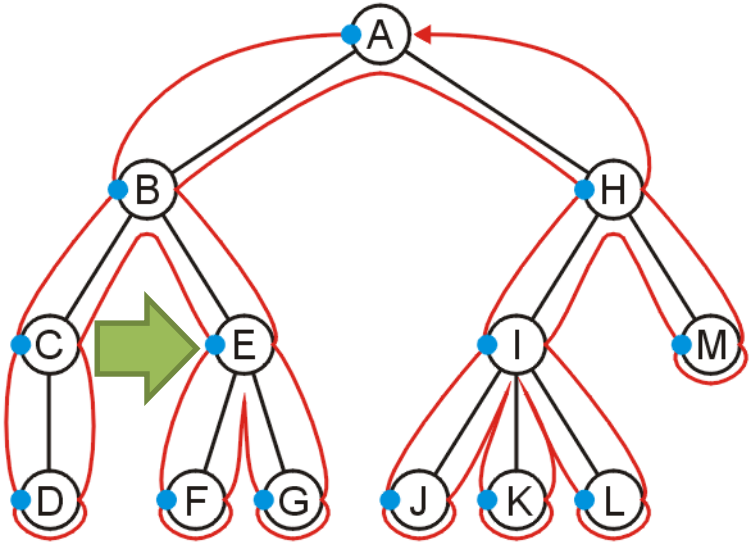


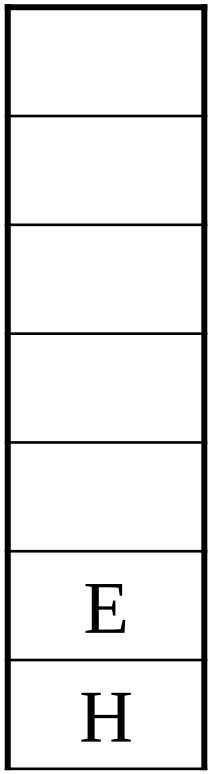
DFS using a Stack



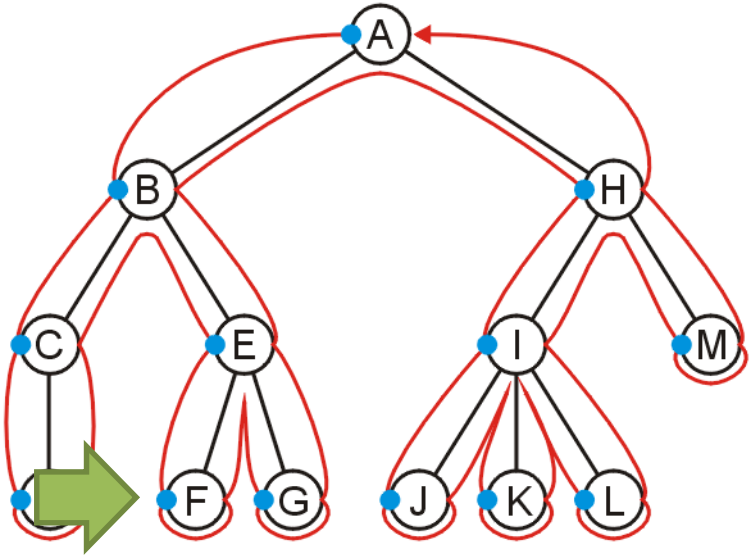


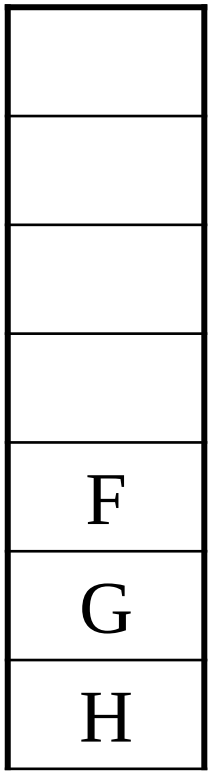
DFS using a Stack



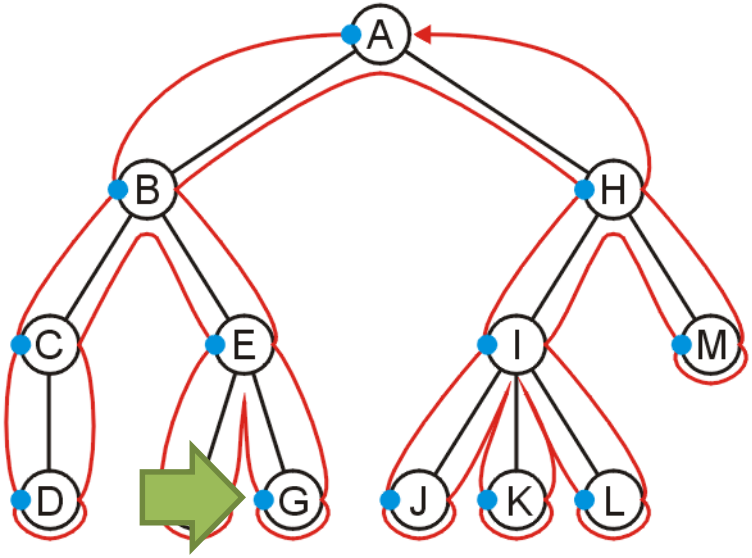


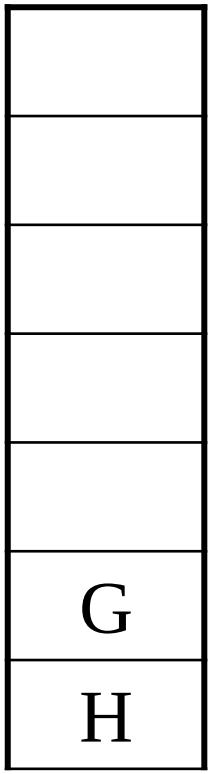
DFS using a Stack



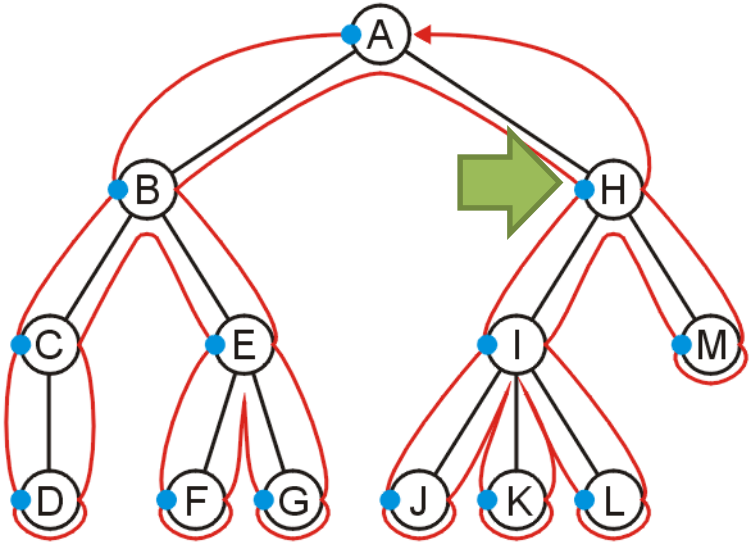


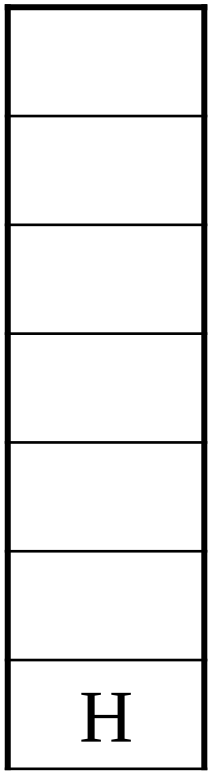
DFS using a Stack



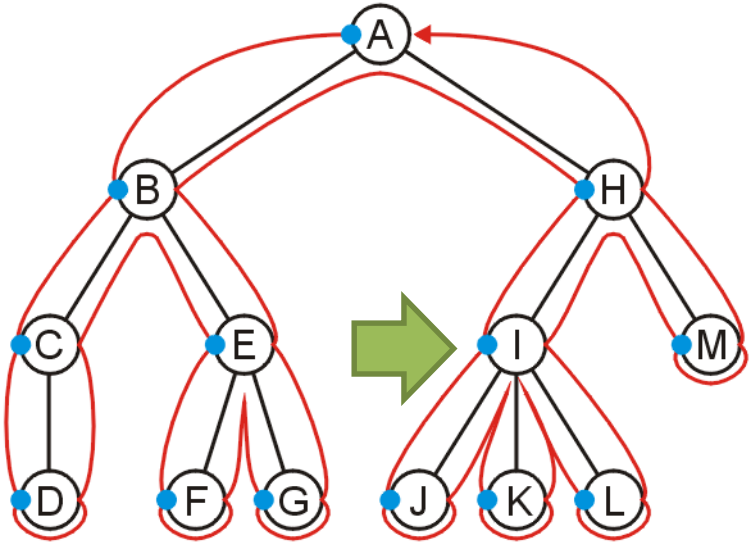


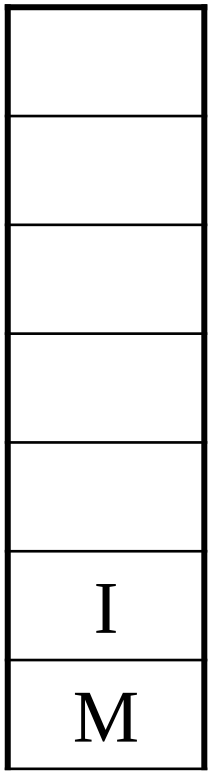
DFS using a Stack



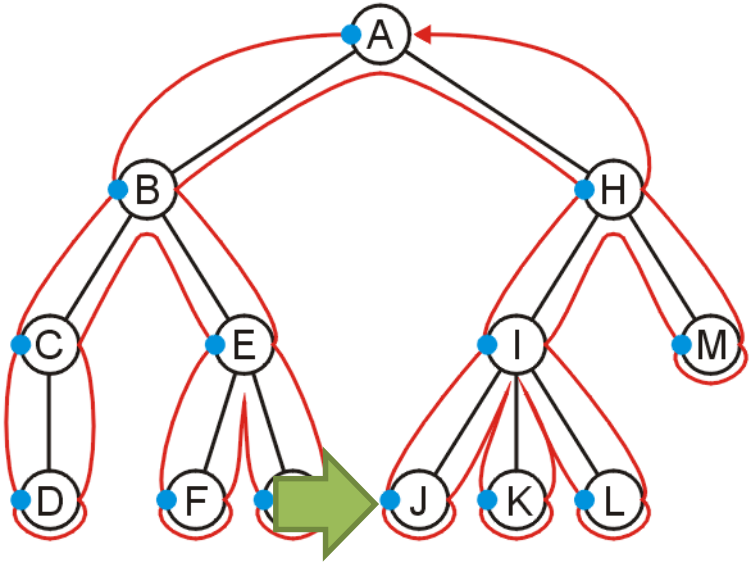


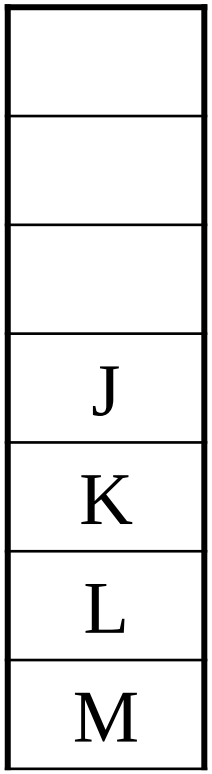
DFS using a Stack



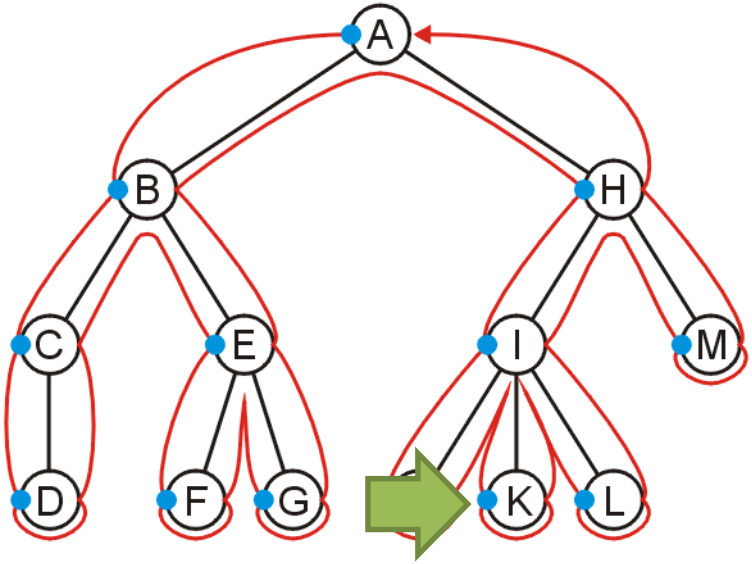


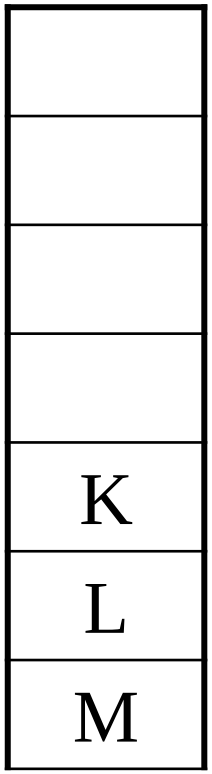
DFS using a Stack



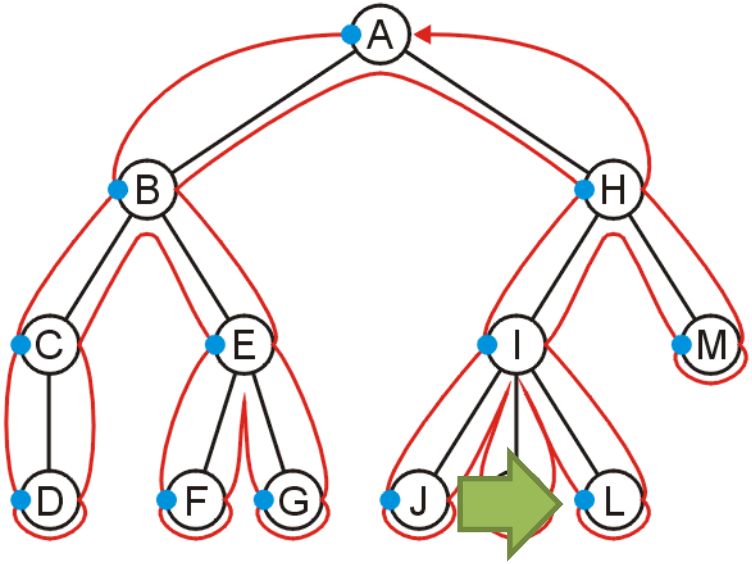


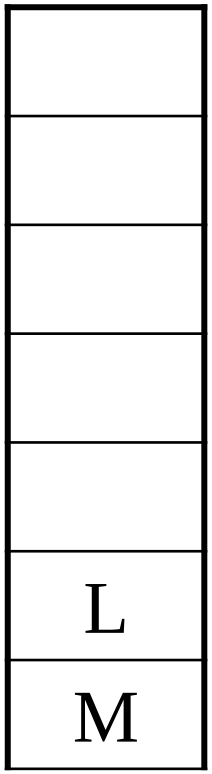
DFS using a Stack



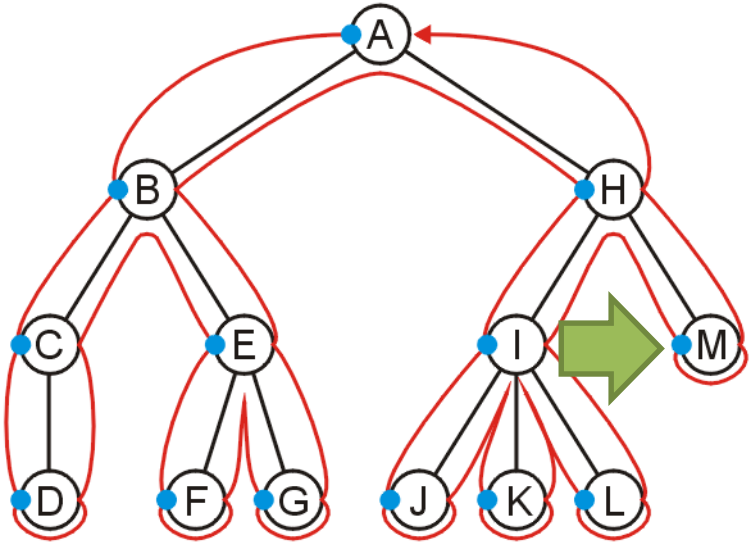


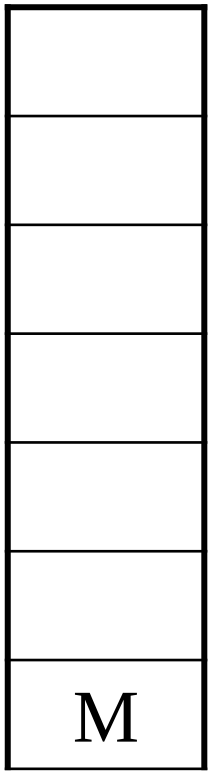
DFS using a Stack



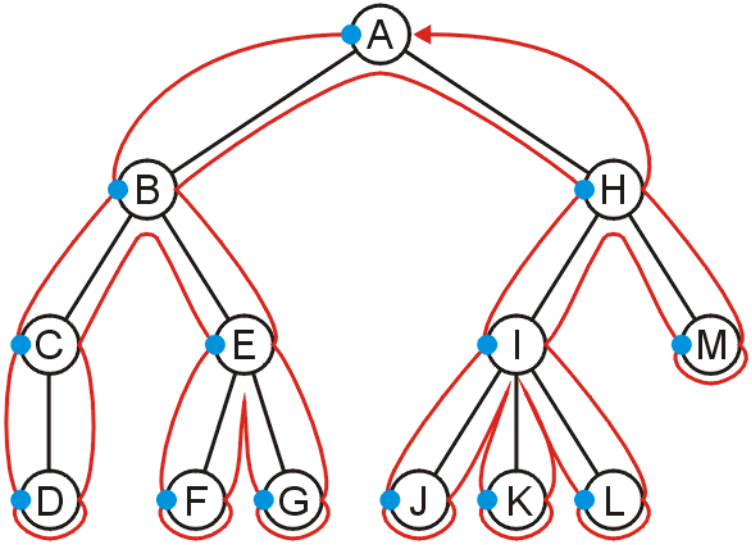


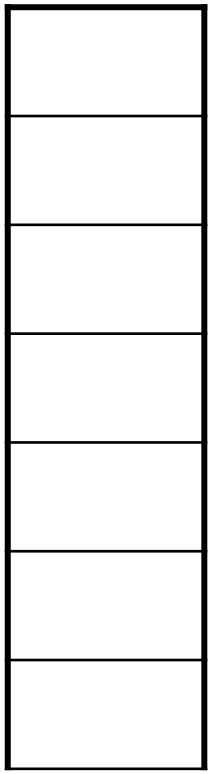
DFS using a Stack





DFS using a Stack





Implementing Depth-First Traversals

Computational complexity of DFS using stack

- Run time is $\Theta(n)$
- The objects on the stack are all unvisited siblings from the root to the current node
  - If each node has a maximum of two children, the memory required is $\Theta(h)$: the height of the tree

Computational complexity of DFS using recursion?

- The same complexity?

Guidelines

DFS is used when at each node we need information from:

- all its children or descendants, or
- all its ancestors

In designing a DFS, it is necessary to consider:

1. Before the children are traversed, what initializations, operations and calculations must be performed?
2. In recursively traversing the children:
   a) What information must be passed to the children during the recursive call?
   b) What information must the children pass back, and how must this information be collated?
3. Once all children have been traversed, what operations and calculations depend on information collated during the recursive traversals?
4. What information must be passed back to the parent?

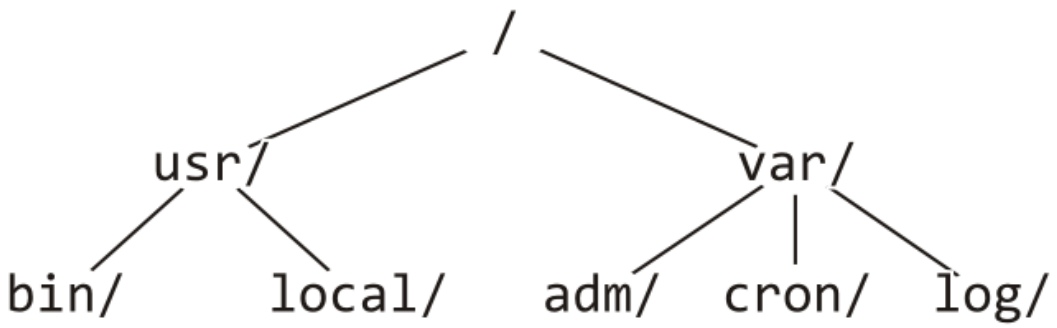
Applications of DFS

Displaying information about directory structures and the files contained within

- Printing a hierarchical structure
- Determining memory usage

Printing a Hierarchy

Consider the directory structure presented on the left—how do we display this in the format on the right?




```
/
    usr/
        bin/
        local/
    var/
        adm/
        cron/
        log/
```

What do we do at each step?

Printing a Hierarchy

For a directory, we initialize a tab level at the root to 0

We then do:

1. Before the children are traversed, we must:
   a) Indent an appropriate number of tabs, and
   b) Print the name of the directory followed by a `'/'`
2. In recursively traversing the children:
   a) A value of one plus the current tab level must be passed to the children, and
   b) No information must be passed back
3. Once all children have been traversed, we are finished

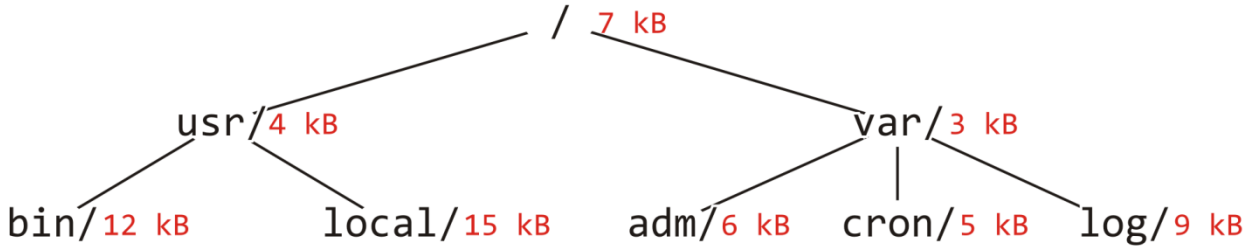
Printing a Hierarchy

Assume the function `void print_tabs( int n )` prints *n* tabs

```
template <typename Type>
void Simple_tree<Type>::print( int depth ) const {
   print_tabs( depth );
   std::cout << retrieve()->name() << '/' << std::endl;

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      ptr->retrieve()->print( depth + 1 );
   }
}
```

Determining Memory Usage

Printing the directory memory usage of this directory structure:




```
   bin/ 12
   local/ 15
usr/ 31
   adm/ 6
   cron/ 5
   log/ 9
var/ 23
/ 61
```

Determining Memory Usage

For a directory, we initialize a tab level at the root to 0

We then do:

1. Before the children are traversed, we must:
   a) Initialize the memory usage to that in the current directory.
2. In recursively traversing the children:
   a) A value of one plus the current tab level must be passed to the children, and
   b) Each child will return the memory used within its directories and this must be added to the current memory usage.
3. Once all children have been traversed, we must:
   a) Print the appropriate number of tabs,
   b) Print the name of the directory followed by a "/ ", and
   c) Print the memory used by this directory and its descendants
4. Return the memory usage by this directory and its descendants

Printing a Hierarchy

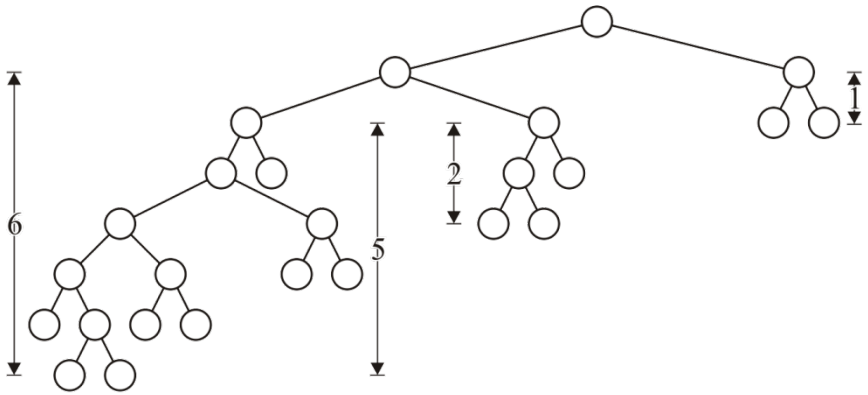
```
template <typename Type>
int Simple_tree<Type>::du( int depth ) const {
   int usage = retrieve()->memory();

   for (
      Single_node<Simple_tree *> *ptr = children.head();
      ptr != 0; ptr = ptr->next()
   ) {
      usage += ptr->retrieve()->du( depth + 1 );
   }

   print_tabs( depth );
   std::cout << retrieve()->name() << "/ " << usage << std::endl;

   return usage;
}
```

Height

The `int height() const` function is recursive in nature:

1. Before the children are traversed, we assume that the node has no children and we set the height to zero: $h_{\text{current}} = 0$
2. In recursively traversing the children, each child returns its height h and we update the height if $1 + h > h_{\text{current}}$
3. Once all children have been traversed, we return h_{current}

When the root returns a value, that is the height of the tree



Outline

- Tree structure
- Implementation
- Tree traversal
- Forest

Hierarchical relation

Recall the properties of a hierarchical relation:

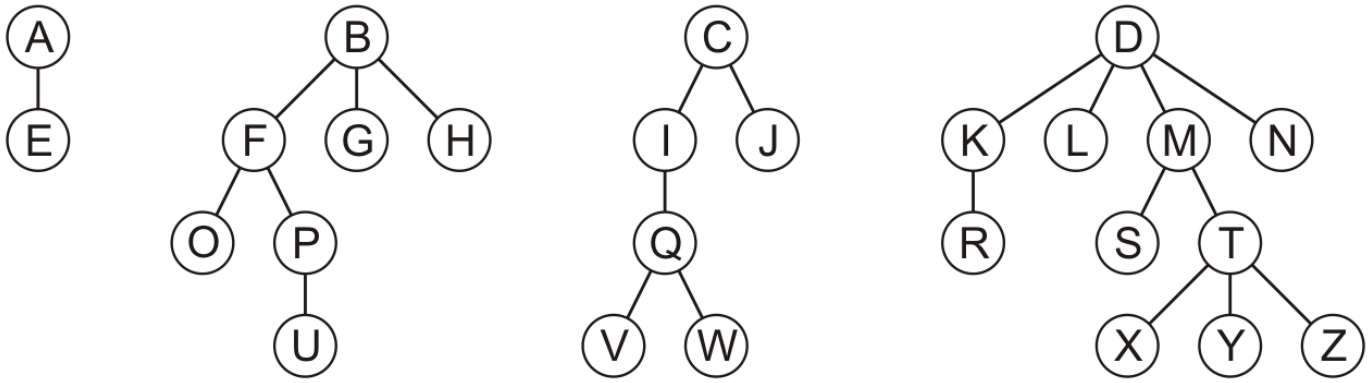
- It is never true that $x \prec x$
- If $x \prec y$ then $y \not\prec x$
- If $x \prec y$ and $y \prec z$, it follows that $x \prec z$
- There is a root r such that $r \prec x$ for all x
- If is $x \prec z$ and $y \prec z$, it follows that either $x \prec y$, $x = y$ or $x \succ y$

If we remove the restriction that there is a unique root r, we allow for the possibility of a number of roots

- If a set S has such a relationship on it, we can define a tree rooted at a point x as the collection of all y such that $x \prec y$
- For a finite set S, there is a set of points R such that for each $r \in R$, there are no points $x \in S$ such that $x \prec r$
  - We call R the set of roots

Forest

A rooted forest is a data structure that is a collection of disjoint rooted trees



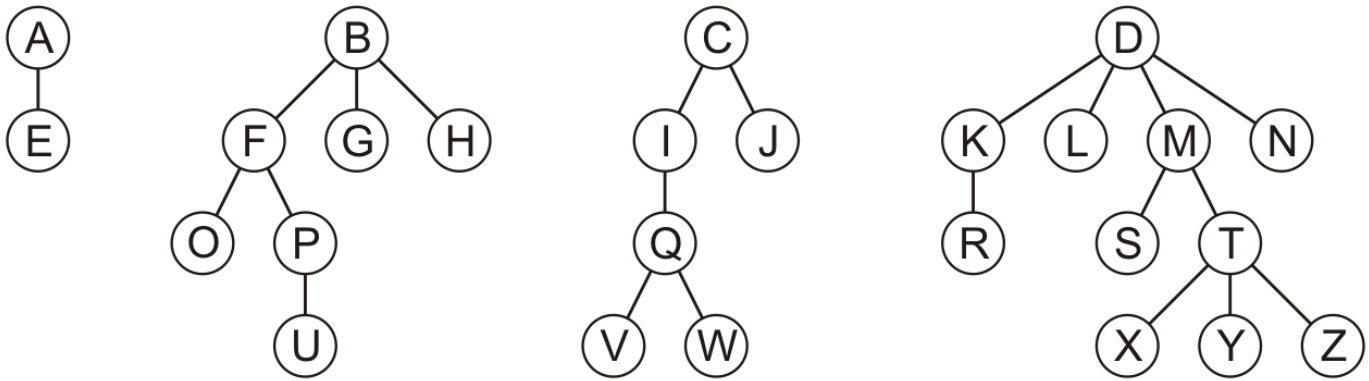
Forest

Note that:

- Any tree can be converted into a forest by removing the root node
- Any forest can be converted into a tree by adding a root node that has the roots of all the trees in the forest as children

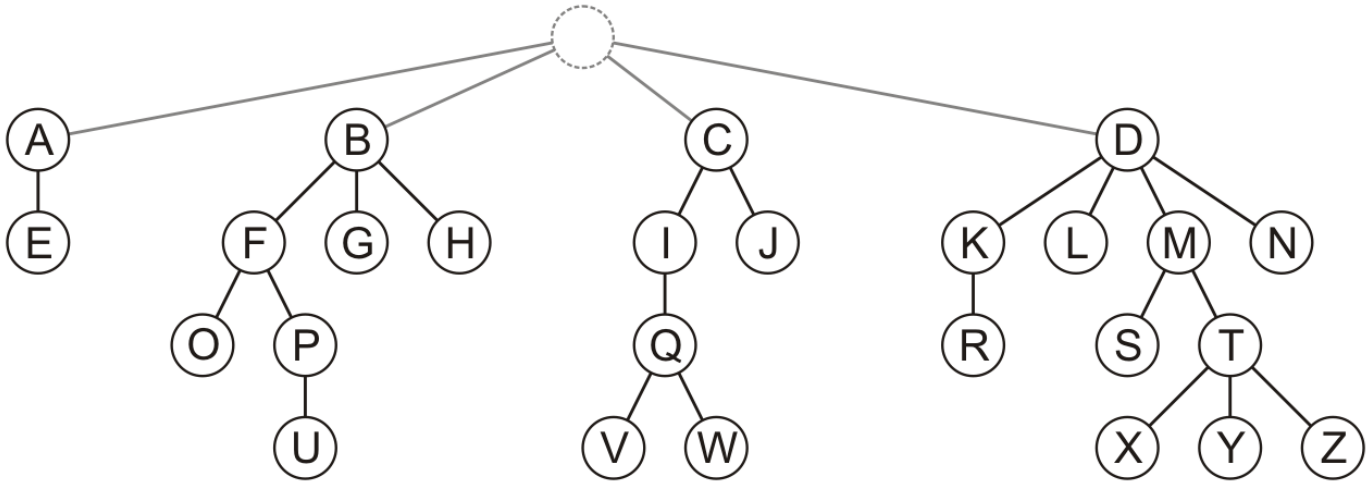
Traversals

Traversals on forests can be achieved by treating the roots as children of a notional root



Traversals

Traversals on forests can be achieved by treating the roots as children of a notional root



Pre-order traversal: A E B O F P U G H C I Q V W J D K R L M S T X Y Z N

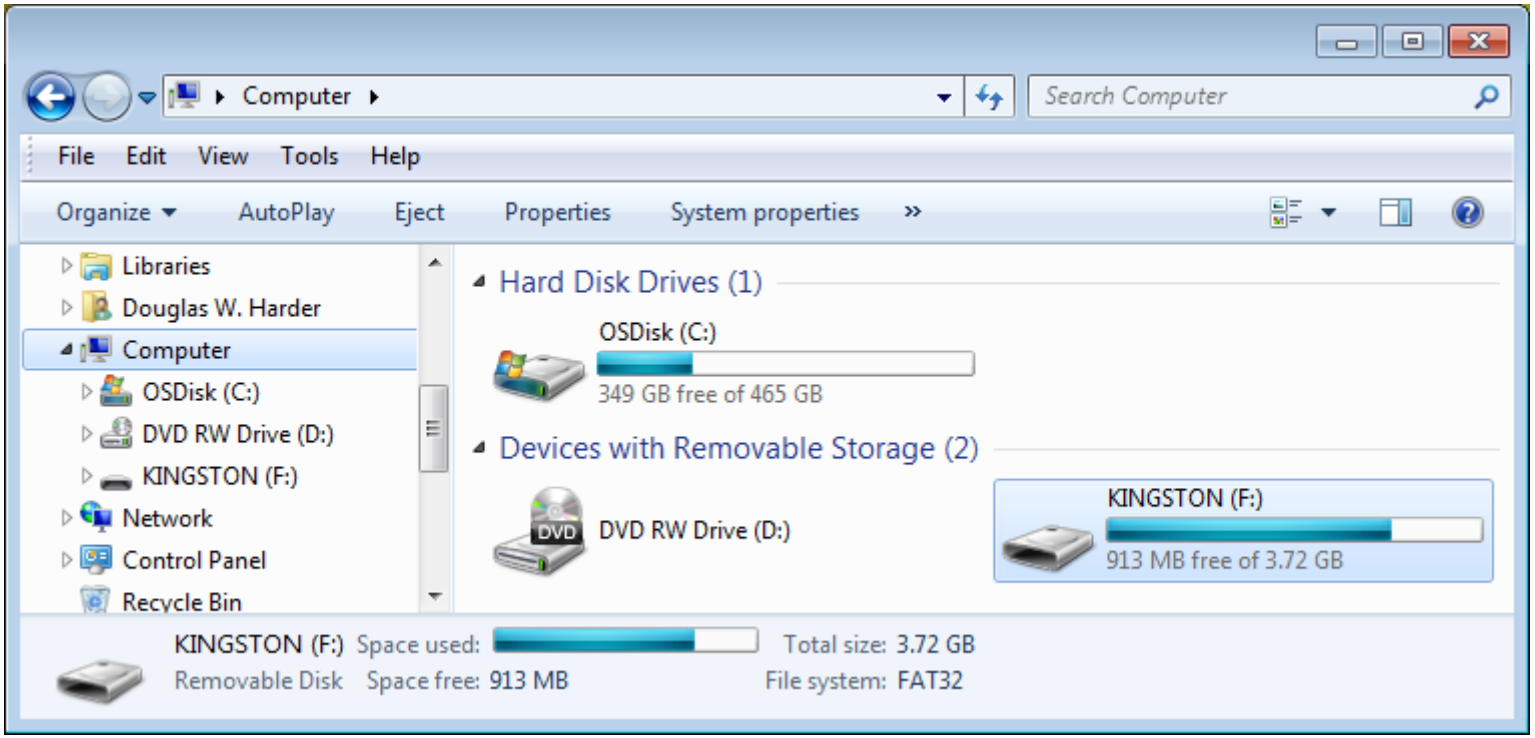
Post-order traversal: E A O U P F G H B V W Q I J C R K L S X Y Z T M N D

Breadth-first traversal: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Application

In Windows, each drive forms the root of its own directory structure

- Each of the directories is hierarchical—that is, a rooted tree



Application

In C++, if you do not use multiple inheritance, the class inheritance structure is a forest

- In Java and C#, it is a rooted tree with `Object` being the root class

If you allow multiple inheritance in C++, you have a partial order

- A *directed acyclic graph* data structure allows you store such a relation

Summary

- Tree structure
  - Terminology
- Implementation
  - Children in a list
- Tree traversal
  - BFS, using queue
  - DFS, using recursion or stack
- Forest