Discussion Week 4

Topic:

bubble sort, insert sort,

DFS, BFS

Bubble Sort

The basic algorithm

Starting with the first item, assume that it is the largest

Compare it with the second item:

- If the first is larger, swap the two,
- Otherwise, assume that the second item is the largest

Continue up the array, either swapping or redefining the largest item

The Basic Algorithm

After one pass, the largest item must be the last in the list

Start at the front again:

- the second pass will bring the second largest element into the second last position

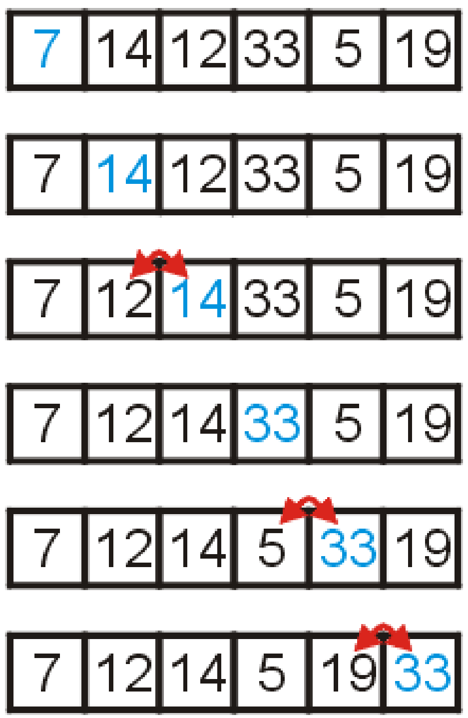
Repeat $n - 1$ times, after which, all entries will be in place

Example

Consider the unsorted array to the right

We start with the element in the first location, and move forward:

- if the current and next items are in order, continue with the next item, otherwise
- swap the two entries



Example

Each point (x, y) indicating that the value y is stored at index x. The series of image proceeds in # iterations.

Checkpoint

Suppose you have the following list of numbers to sort: **[19, 1, 9, 7, 3, 10, 13, 15, 8, 12]** which list represents the partially sorted list after three complete passes of bubble sort?

A. [1, 9, 19, 7, 3, 10, 13, 15, 8, 12]
B. [1, 3, 7, 9, 10, 8, 12, 13, 15, 19]
C. [1, 7, 3, 9, 10, 13, 8, 12, 15, 19]
D. [1, 9, 19, 7, 3, 10, 13, 15, 8, 12]

Checkpoint

Suppose you have the following list of numbers to sort: **[19, 1, 9, 7, 3, 10, 13, 15, 8, 12]** which list represents the partially sorted list after three complete passes of bubble sort?

A. [1, 9, 19, 7, 3, 10, 13, 15, 8, 12]
B. [1, 3, 7, 9, 10, 8, 12, 13, 15, 19]
C. [1, 7, 3, 9, 10, 13, 8, 12, 15, 19]
D. [1, 9, 19, 7, 3, 10, 13, 15, 8, 12]

Analysis

Here we have two nested loops, and therefore calculating the run time is straight-forward:

$$\sum_{k=1}^{n-1}(n-k) = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2} = \Theta(n^2)$$

Implementations and Improvements

The next few slides show some implementations of bubble sort together with a few improvements:

- halting if the list is sorted,
- limiting the range on which we must bubble
- alternating between bubbling up and sinking down

Flagged Bubble Sort

One useful modification would be to check if no swaps occur:

- If no swaps occur, the list is sorted
- In this example, no swaps occurred during the 5th pass

3	9	5	1	0	2	6	8	4	7
3	5	1	0	2	6	8	4	7	9
3	1	0	2	5	6	4	7	8	9
1	0	2	3	5	4	6	7	8	9
0	1	2	3	4	5	6	7	8	9

Use a Boolean flag to check if no swaps occurred

Range-limiting Bubble Sort

Intuitively, one may believe that limiting the loops based on the location of the last swap may significantly speed up the algorithm

- For example, after the second pass, we are certain all entries after 4 are sorted

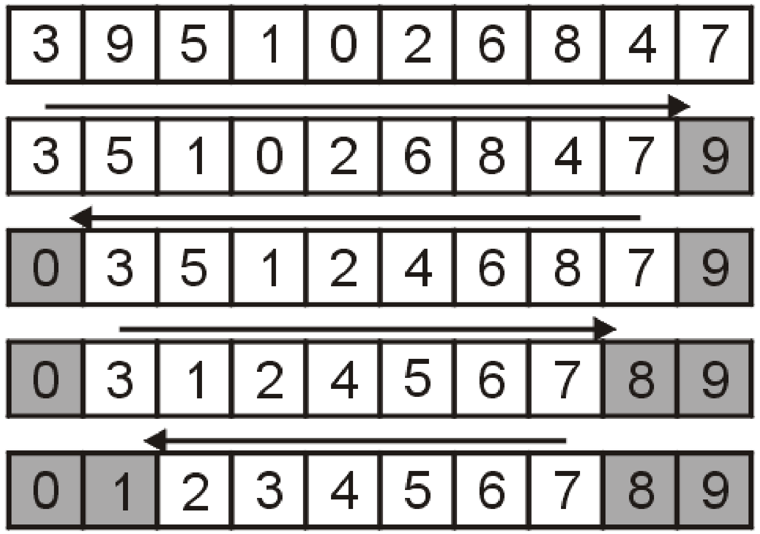
The implementation is easier than that for using a Boolean flag

Unfortunately, in practice, this does little to affect the number of comparisons

Alternating Bubble Sort

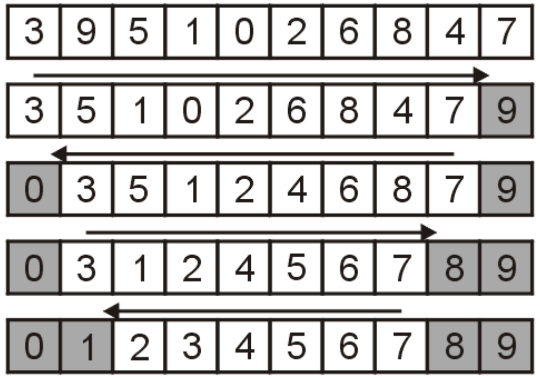
One operation which does significantly improve the run time is to alternate between

- bubbling the largest entry to the top, and
- sinking the smallest entry to the bottom



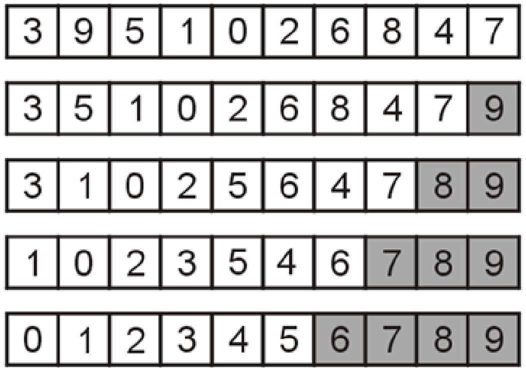
Checkpoint

Which one of them illustrates the process of alternating Bubble Sort?



(A)

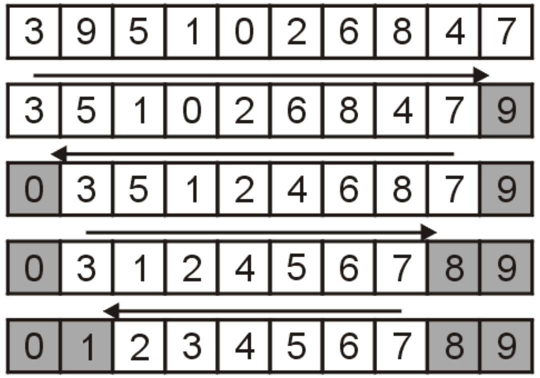
(B)



(C)

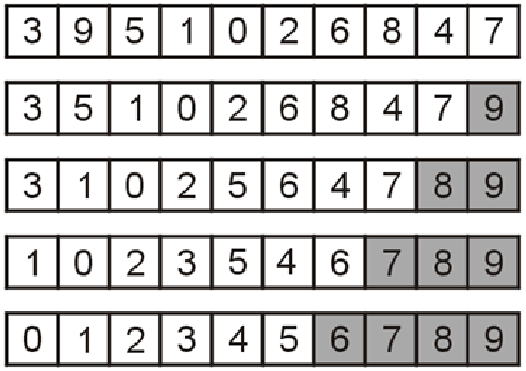
Checkpoint

Which one of them illustrates the process of alternating Bubble Sort?



(A)

(B)

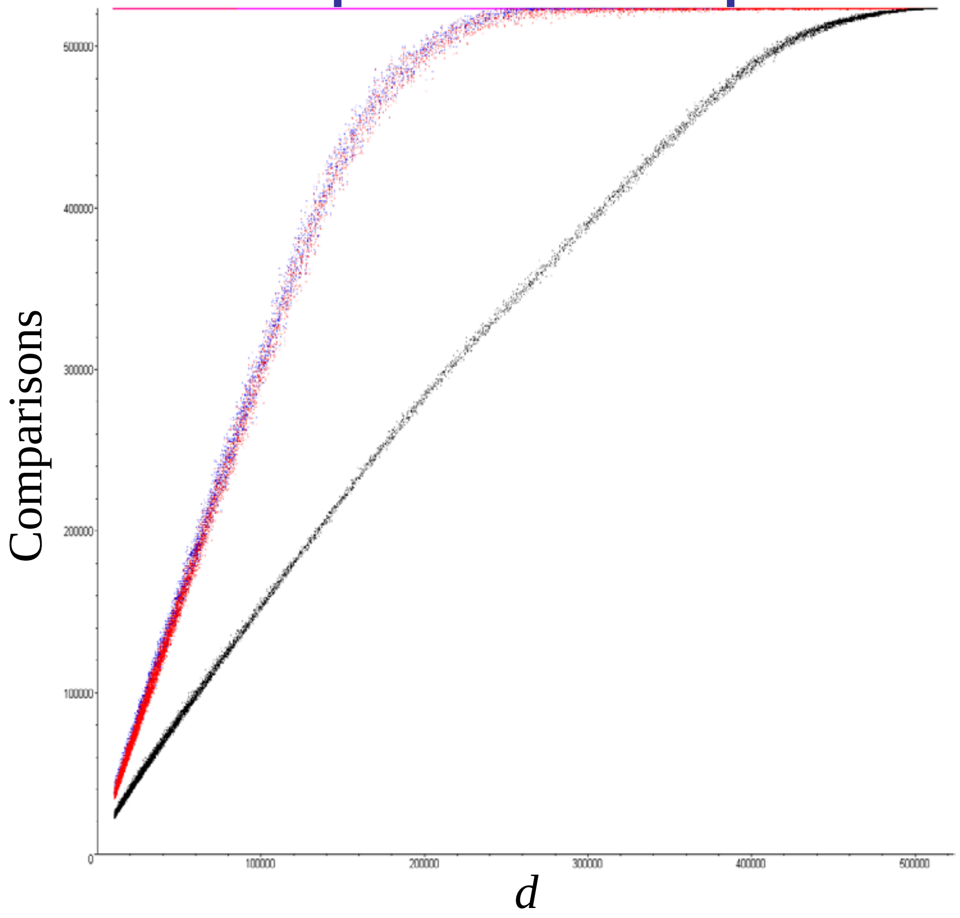


(C)

Empirical Analysis

Each point (d, c) is the number of inversions in an unsorted list d (# inversion)and the number of required comparisons c

Basic implementation
Flagged
Range limiting
Alternating

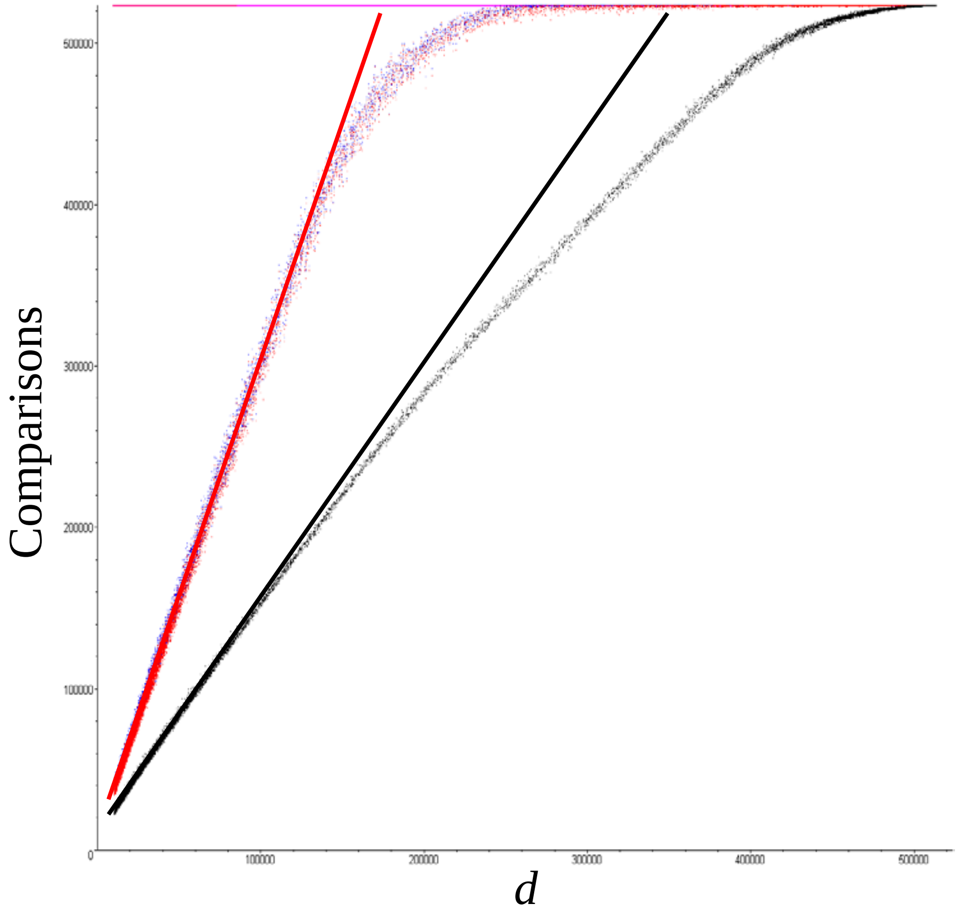


Empirical Analysis

The number of comparisons with the flagged/limiting sort is initially $n + 3d$

For the alternating variation, it is initially $n + 1.5d$

Basic implementation
Flagged
Range limiting
Alternating

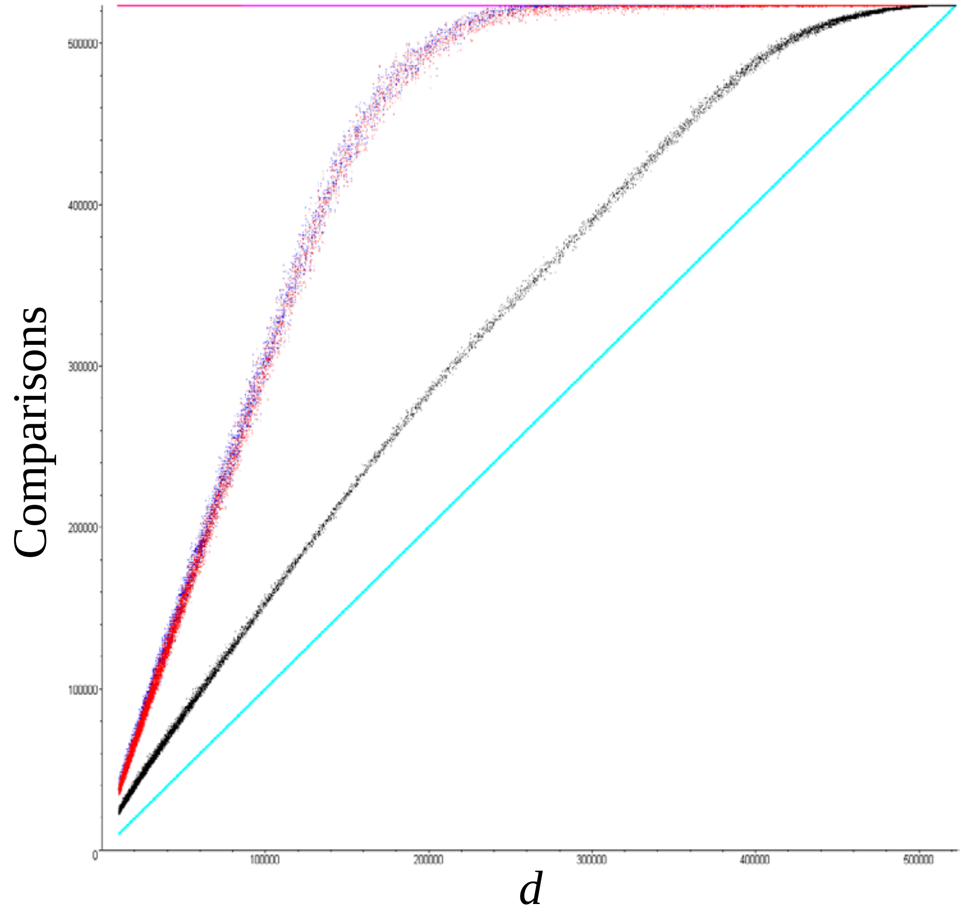


Empirical Analysis

Unfortunately, the comparisons for insertion sort is $n + d$*(introduced in next section)* which is better in all cases except when the list is

- Sorted, or
- Reverse sorted

Basic implementation
Flagged
Range limiting
Alternating
Insertion Sort



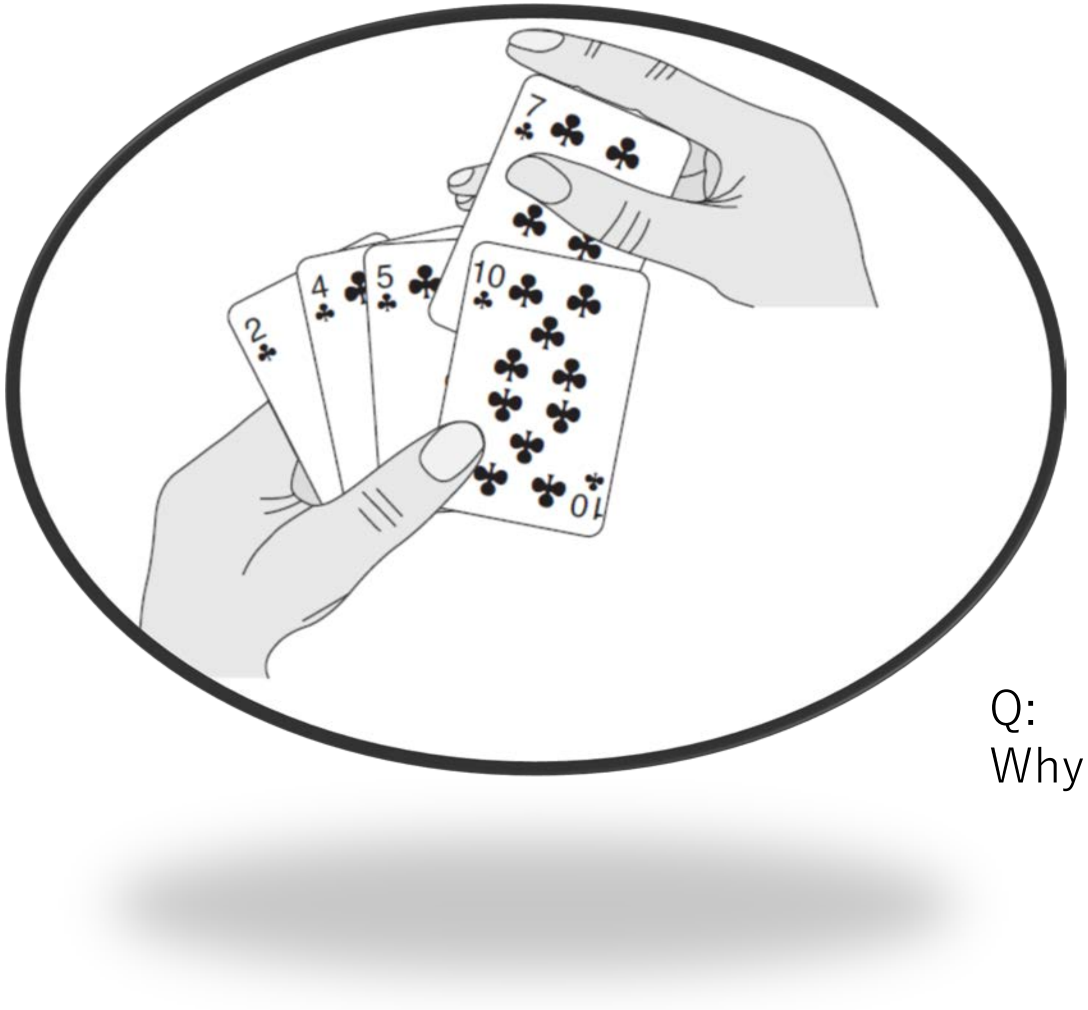
Run-Time

The following table summarizes the run-times of our modified bubble sorting algorithms; however, they are all worse than insertion sort in practice

Case	Run Time	Comments
Worst	$\Theta(n^2)$	$\Theta(n^2)$ inversions
Average	$\Theta(n + d)$	Slow if $d = \omega(n)$
Best	$\Theta(n)$	$d = \mathrm{O}(n)$ inversions

Insertion Sort

Insertion Sort

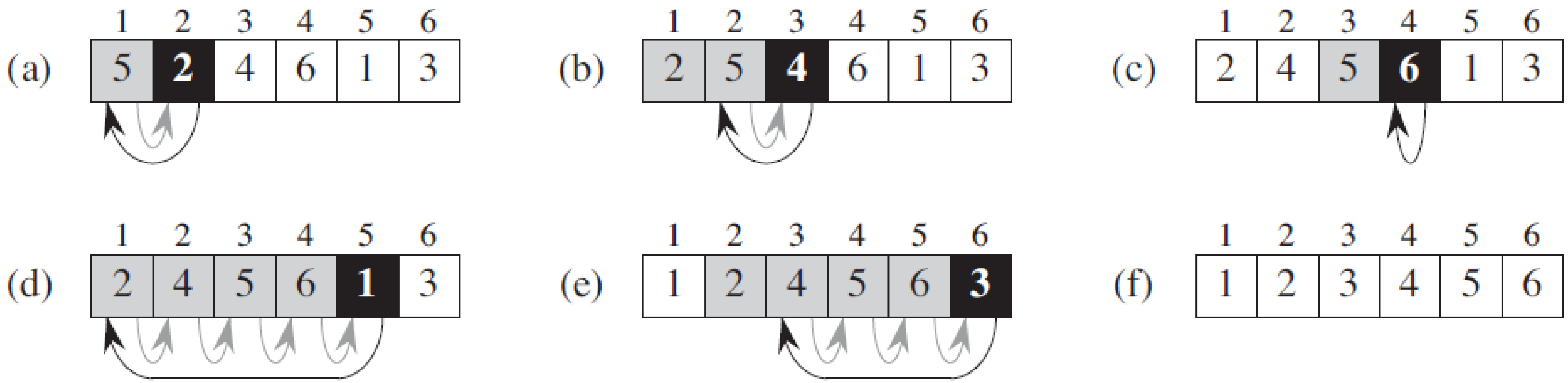


Very intuitive.

Widely-used when you play cards.

Q:
Why it is not an efficient sorting algorithm in programming now?

Visualization

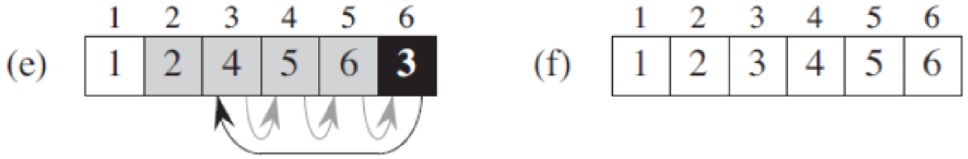


Moving an element backward is much more difficult than moving a card backward.

Implementation

```c
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
   int i, key, j;
   for (i = 1; i < n; i++)
   {
       key = arr[i];
       j = i-1;

       /* Move elements of arr[0..i-1], that are
          greater than key, to one position ahead
          of their current position */
       while (j >= 0 && arr[j] > key)
       {
           arr[j+1] = arr[j];
           j = j-1;
       }
       arr[j+1] = key;
   }
}
```



Run-time Analysis

Because the bubble sort simply swaps adjacent entries, it cannot be any better than insertion sort which does $n + d$ comparisons where d is the number of inversions(will be introduced in next section).

Inversion is defined as a pair of entries which are reversed(a.k.a: (a_j, a_k) if j < k but $a_j > a_k$ for ascending order).

Run-time Analysis

Insertion sort which does $n + d$ comparisons where d is the number of inversions.

If we take a closer look at the insertion sort code, we can notice that every iteration of inner loop reduces one inversion. The while loop executes only if $i > j$ and $arr[i] < arr[j]$.

Run-time Analysis

Insertion sort which does $n + d$ comparisons where d is the number of inversions.

```
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i-1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key;
    }
}
```

Outer loop: O(n)

Inner loop(remove inversion) O(d)

Run-time Analysis

Insertion sort which does $n + d$ comparisons where d is the number of inversions.

The while loop executes only if i > j and arr[i] < arr[j].

Therefore overall time complexity of the insertion sort is O(n + d) where d is inversion count. If the inversion count is O(n), then the time complexity of insertion sort is O(n).

Run-time Analysis

Insertion sort which does $n + d$ comparisons where d is the number of inversions.

In worst case, there can be n*(n-1)/2 inversions. The worst case occurs when the array is sorted in reverse order. So the worst case time complexity of insertion sort is $O(n^2)$.

Tree traversals

Tree traversals

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

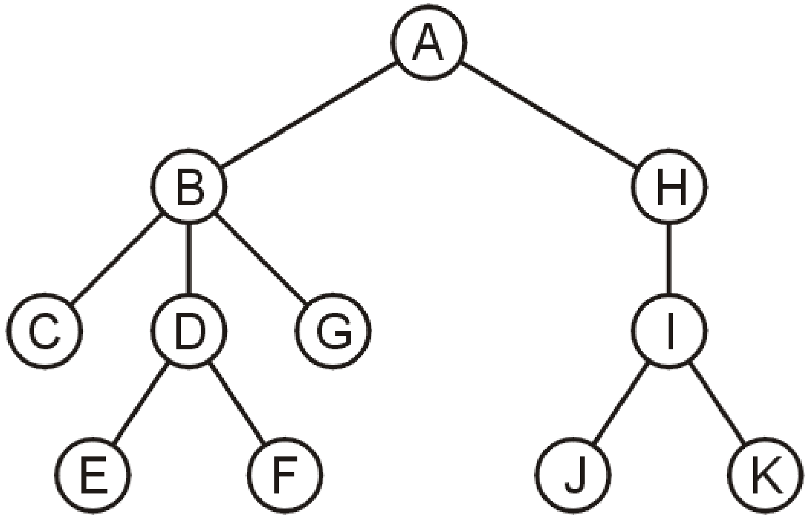
- Requirements: $\Theta(n)$ run time and $O(n)$ memory

Two types of traversals

- Breadth-first traversal
- Depth-first traversal

Breadth-First Traversal

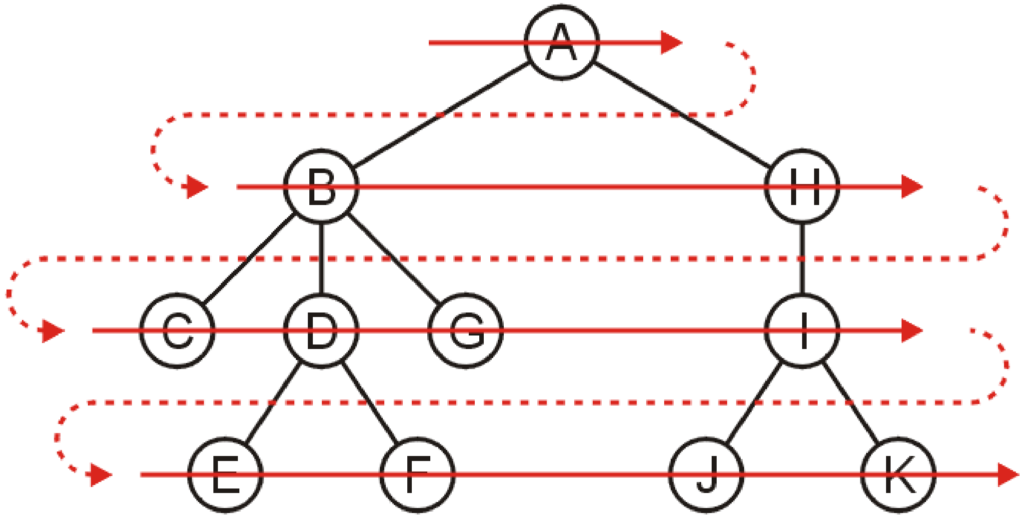
Breadth-first traversals visit all nodes at a given depth before descending a level



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

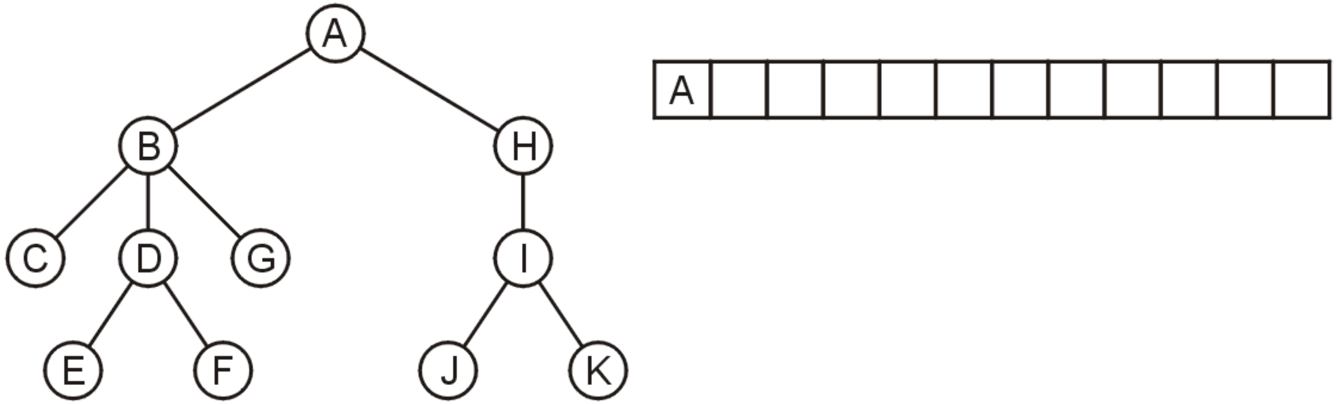
The easiest implementation is to use a queue:

- 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

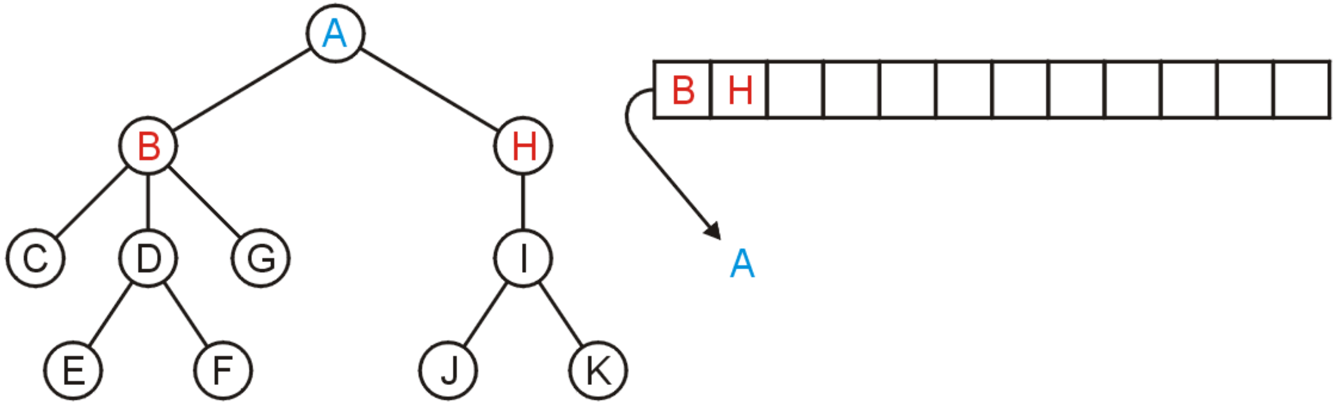
Breadth-First Traversal

Push the root directory A



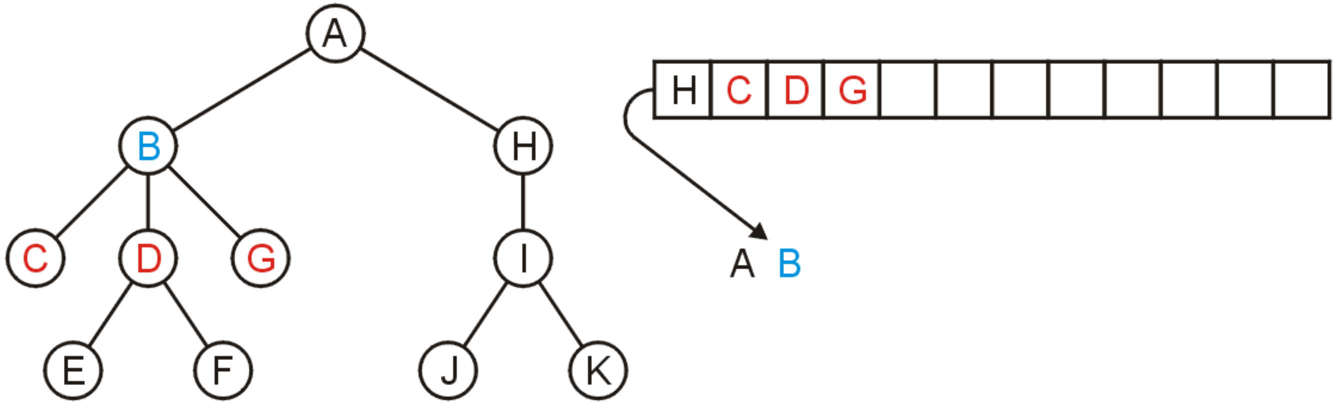
Breadth-First Traversal

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



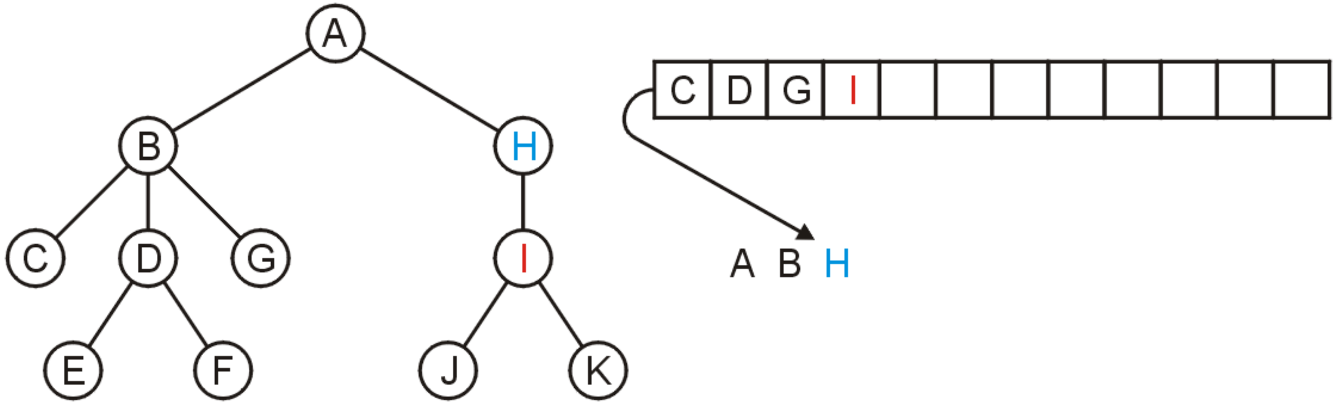
Breadth-First Traversal

Pop B and push C, D, and G



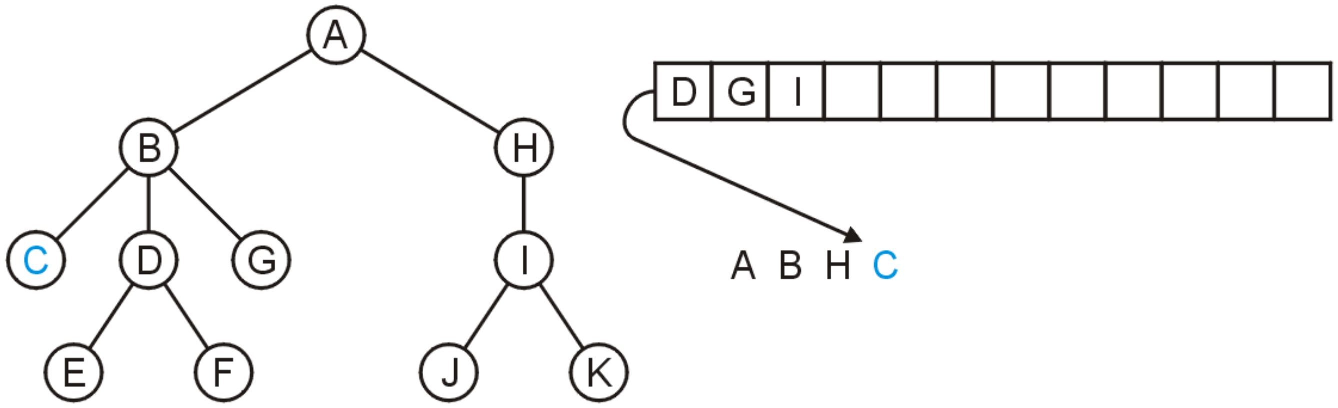
Breadth-First Traversal

Pop H and push its one sub-directory I



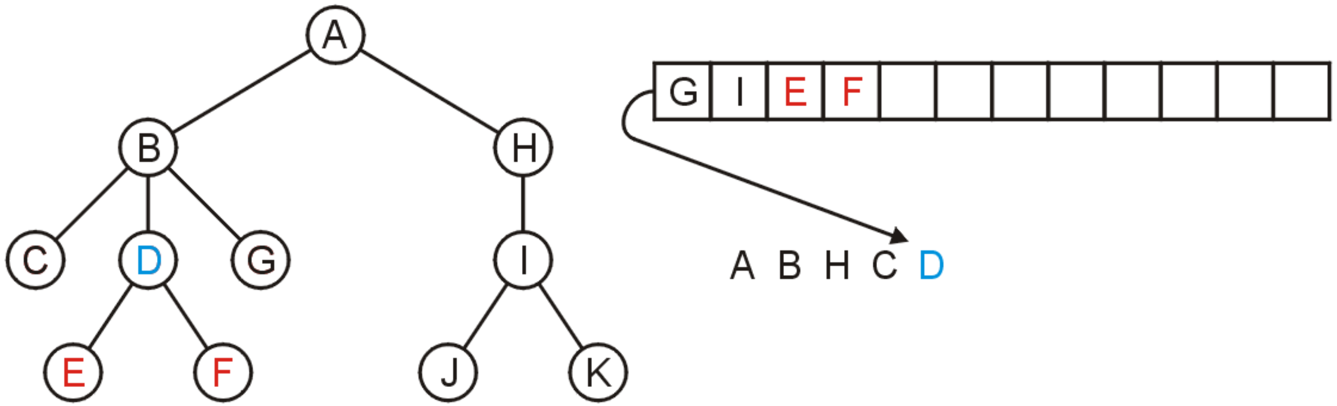
Breadth-First Traversal

Pop C: no sub-directories



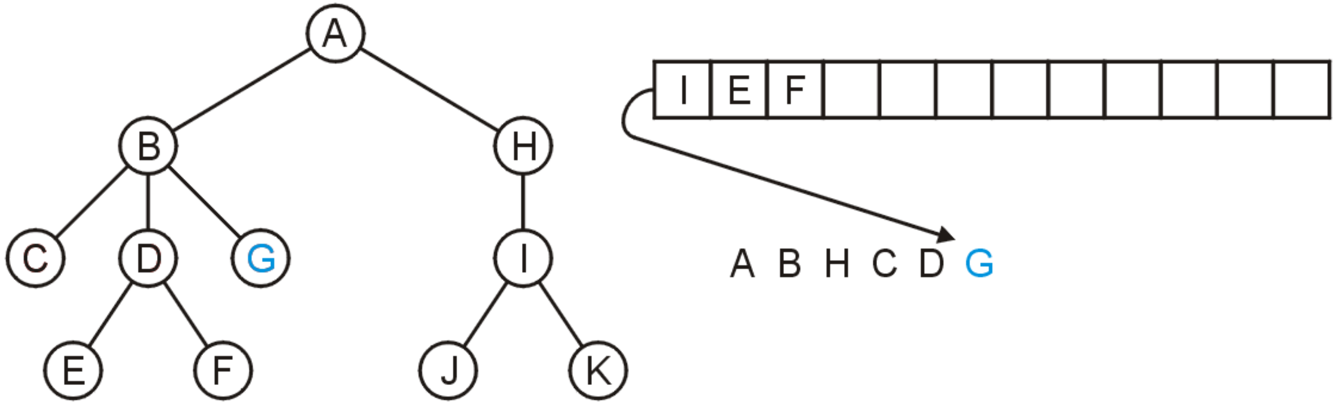
Breadth-First Traversal

Pop D and push E and F



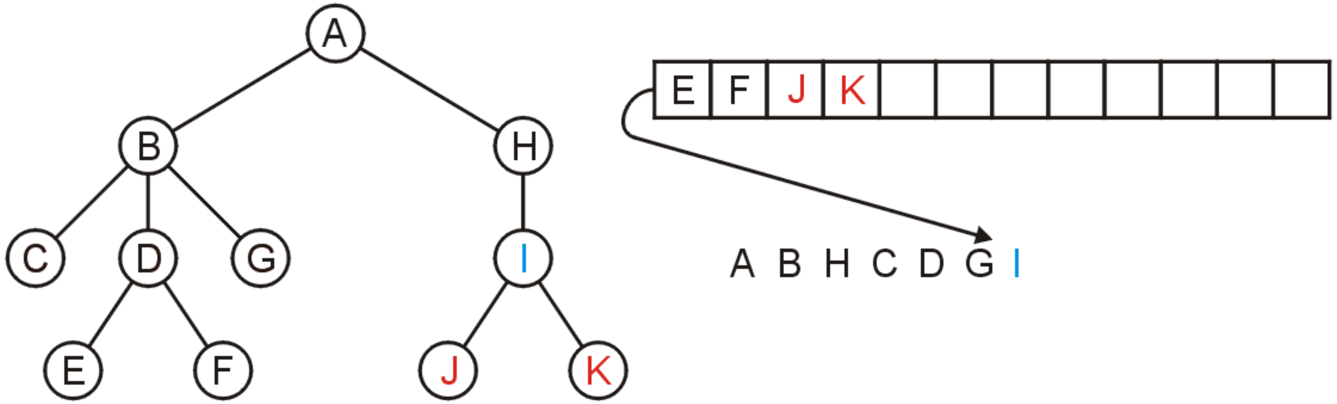
Breadth-First Traversal

Pop G



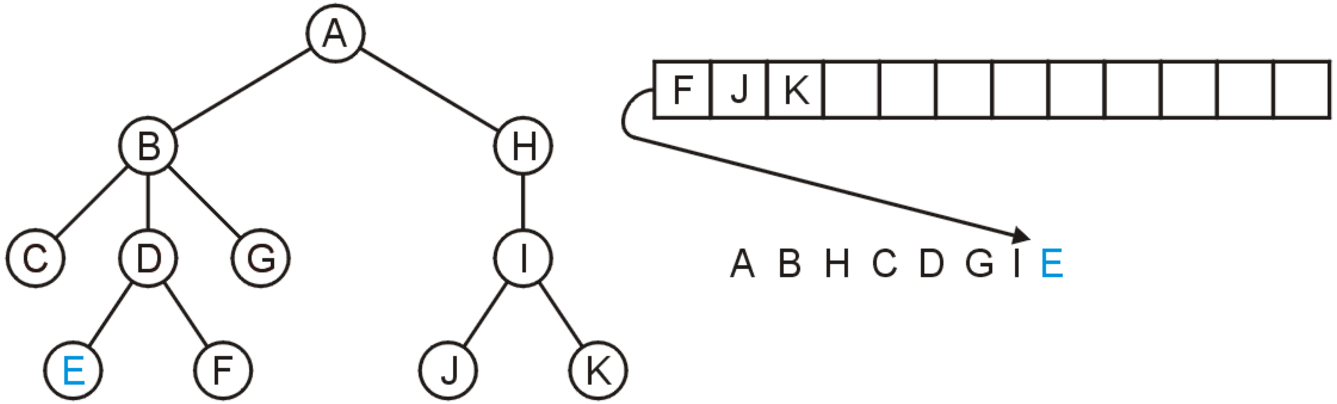
Breadth-First Traversal

Pop I and push J and K



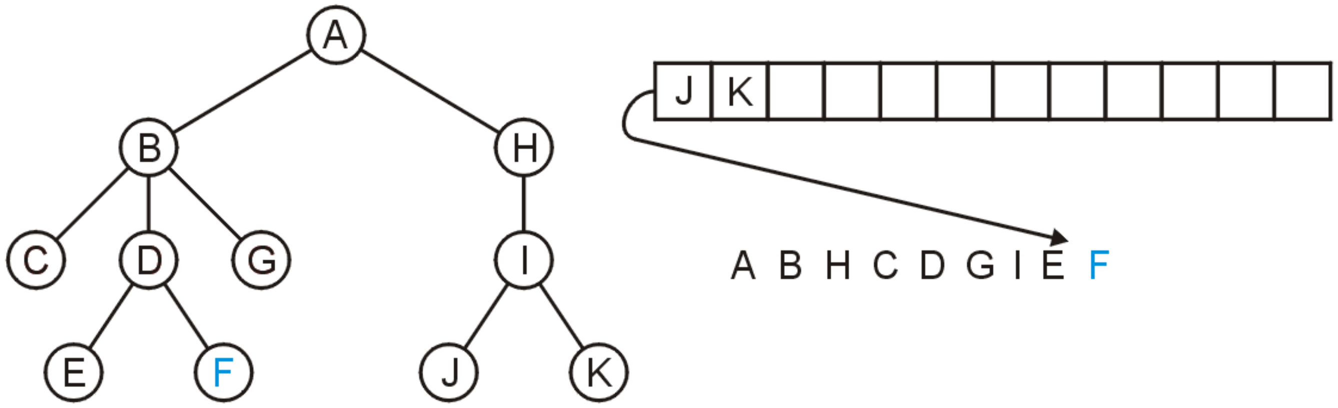
Breadth-First Traversal

Pop E



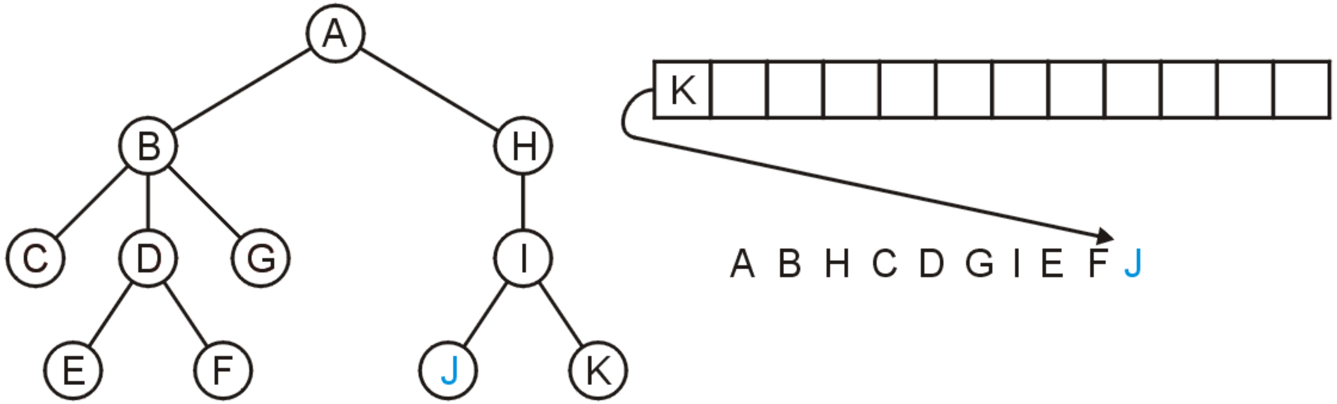
Breadth-First Traversal

Pop F



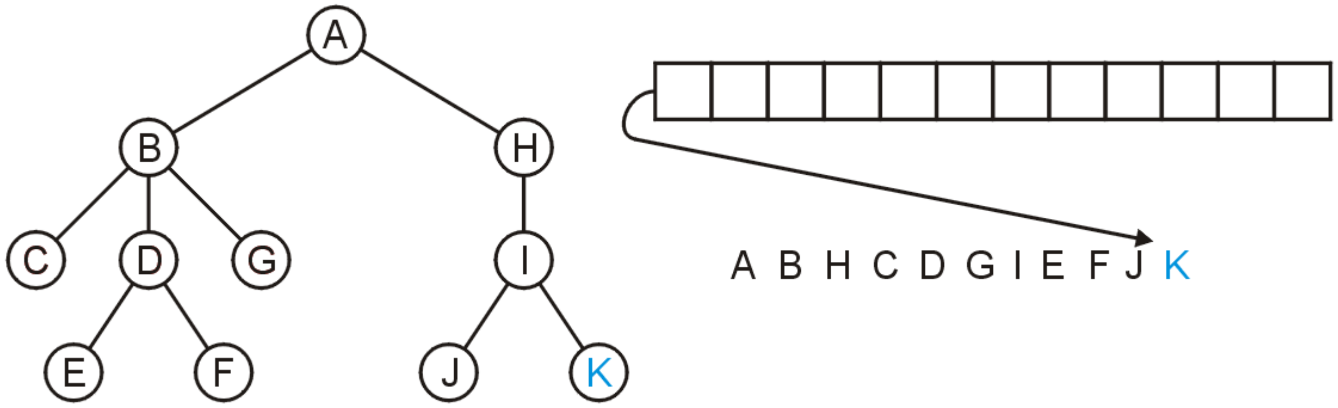
Breadth-First Traversal

Pop J



Breadth-First Traversal

Pop K and the queue is empty

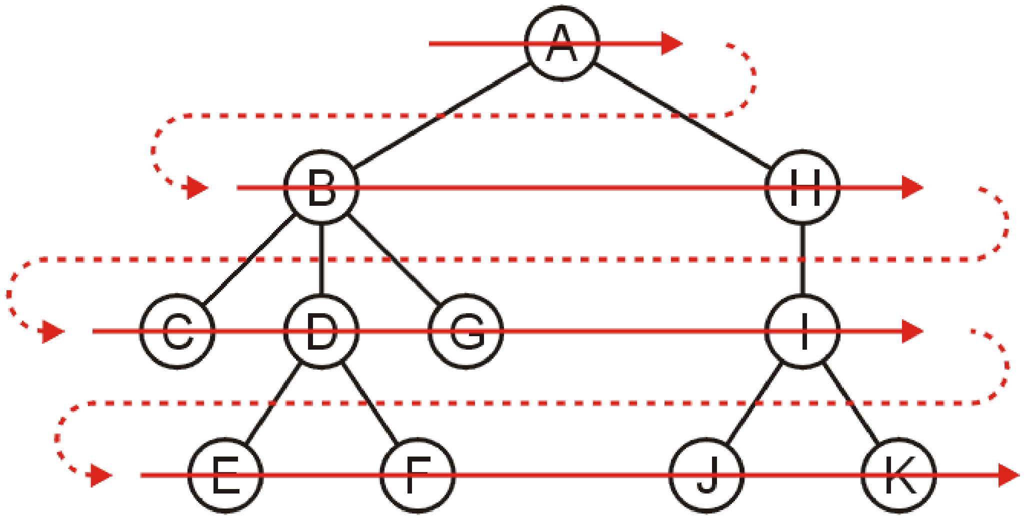


Breadth-First Traversal

The resulting order

A B H C D G I E F J K

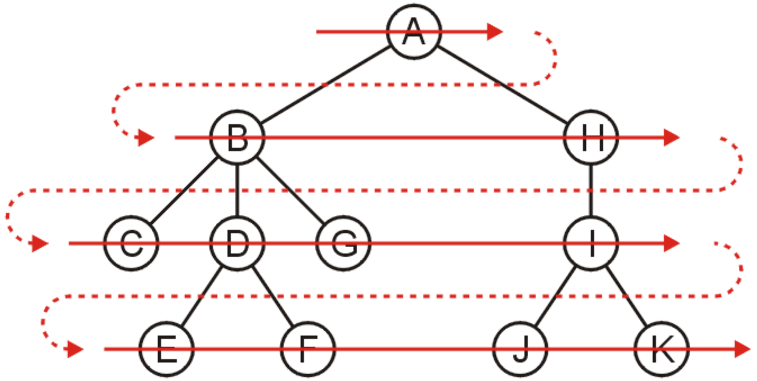
is in breadth-first order:



Breadth-First Traversal

```
void BFS(Node *pRoot)
{
        if (pRoot==NULL)  return;
        queue<Node*> Q;
        Q.push(pRoot);

        while(!Q.empty())
        {
                Node *node = Q.pop();
                output(node)
                for(child-node in node->children)
                        Q.push(child-node);
        }
}
```



Breadth-First Traversal

Computational complexity

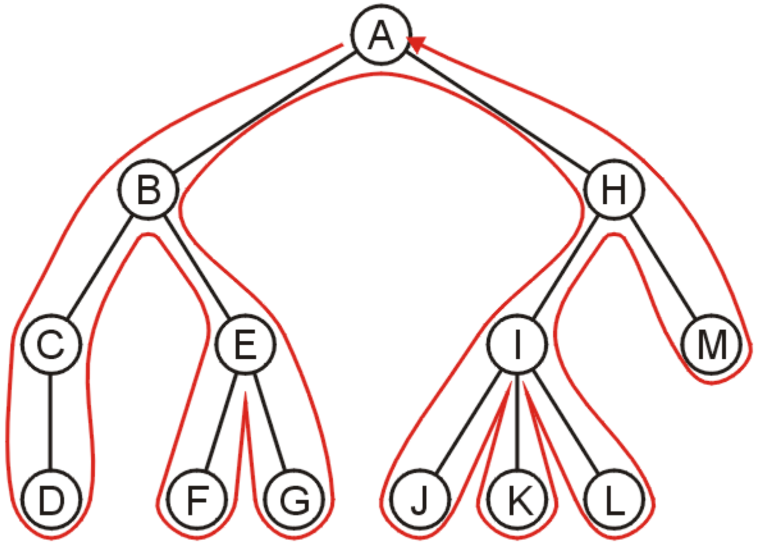
- Run time is $\Theta(n)$
- Space: maximum nodes at a given depth, $\mathrm{O}(n)$

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

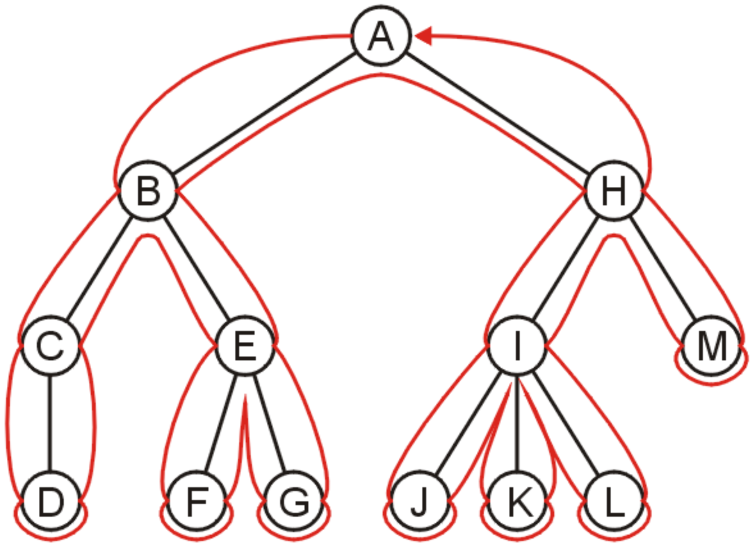
We end once all the children of the root are visited



Depth-first Traversal

Each node is visited multiple times in such a scheme

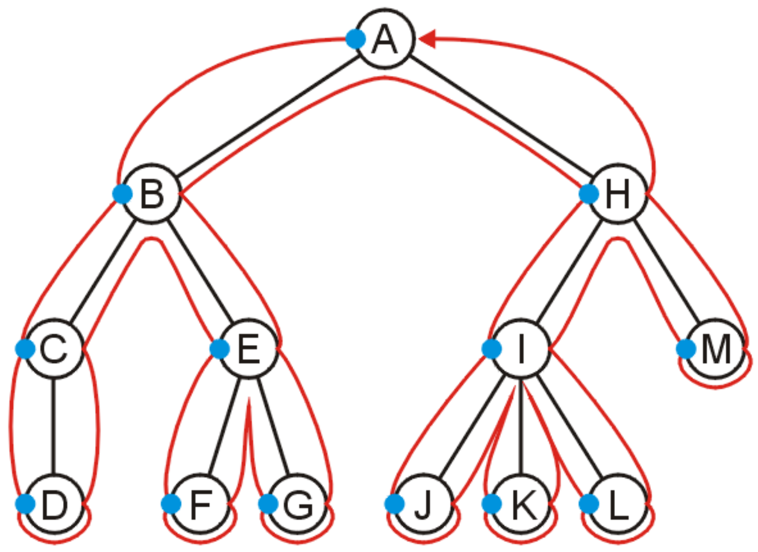
- First time: before any children
- Last time: after all children, before backtracking



Pre-ordering

Ordering nodes by their first visits results in the sequence:

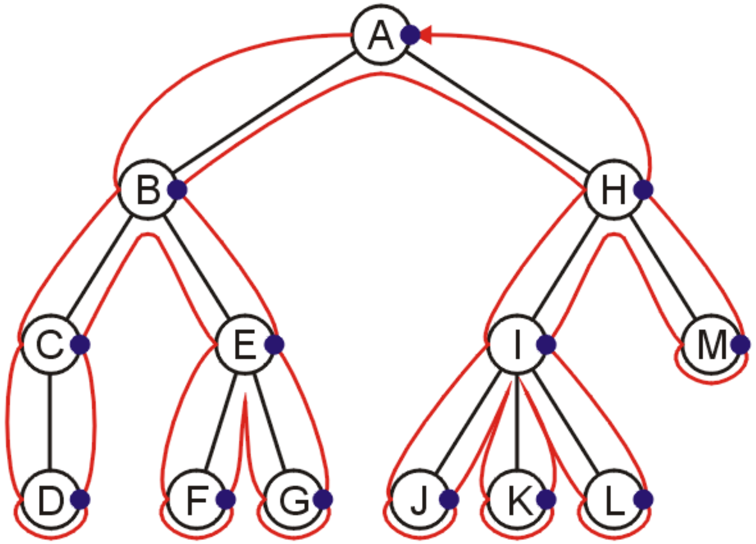
A, B, C, D, E, F, G, H, I, J, K, L, M



Post-ordering

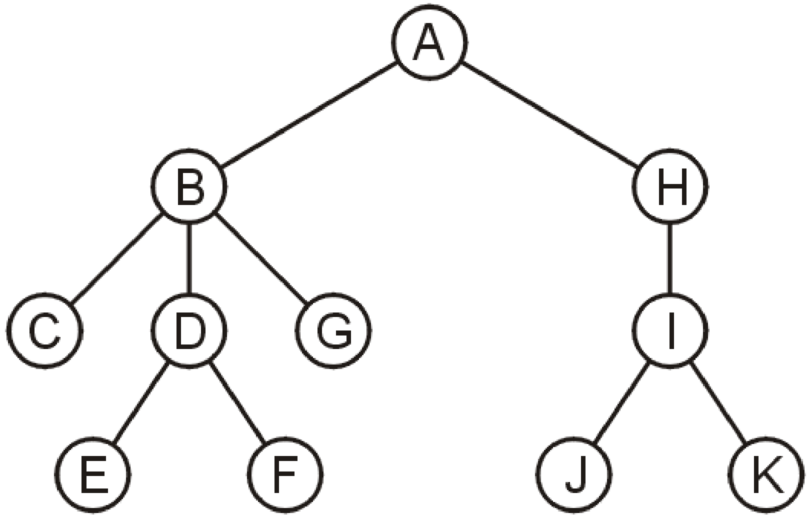
Ordering nodes by their last visits results in the sequence:

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



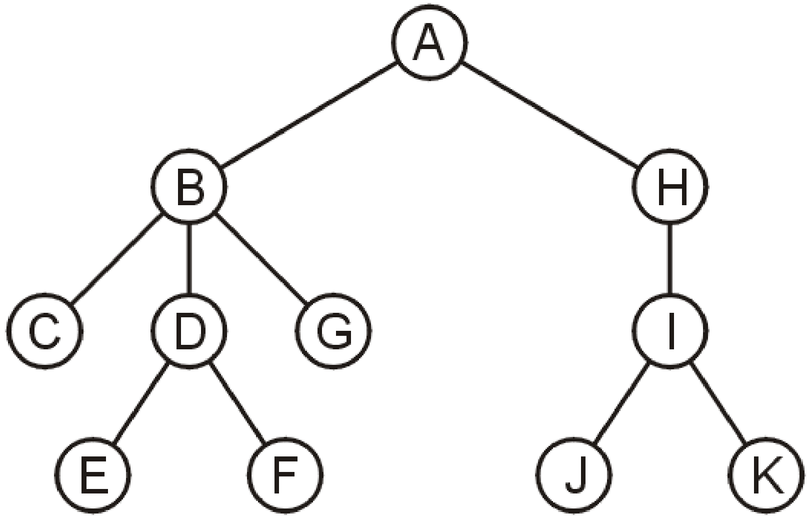
Exercise

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



Exercise

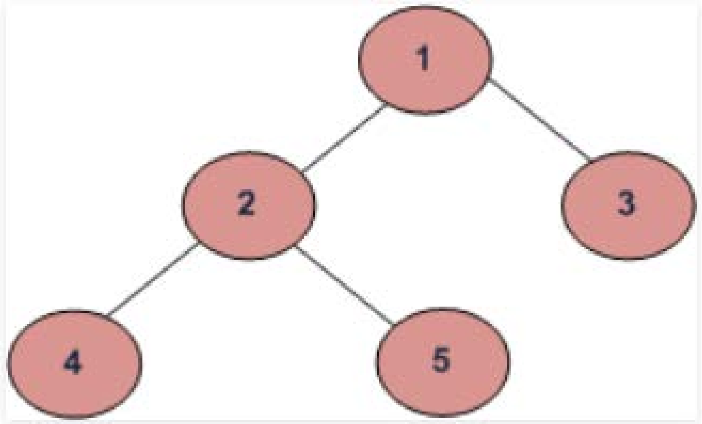
- What is the preordering and postordering of the following tree?
  - Preordering: A B C D E F G H I J K
  - Post-ordering: C E F D G B J K I H A



Special case - binary tree

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)



Inorder (Left, Root, Right) :
4 2 5 1 3

Exercise

Question *What is the post-order sequence of a binary tree with pre-order sequence AMBDJEFQ and in-order sequence BDMJAQFE?*

(A) BDJMFQEA
(B) DBJMFQEA
(C) DBJMQFEA
(D) BDJMQFEA

Exercise

Question *What is the post-order sequence of a binary tree with pre-order sequence AMBDJEFQ and in-order sequence BDMJAQFE?*

(A) BDJMFQEA
(B) DBJMFQEA
(C) DBJMQFEA
(D) BDJMQFEA

Goal: Reconstruct tree from In&pre

Goal: Reconstruct tree from In&pre

pre-order: AMBDJEFQ
in-order: BDMJAQFE

Step1: *Find the root*

Goal: Reconstruct tree from In&pre

pre-order: AMBDJEFQ
in-order: BDMJAQFE

Step1: *Find the root*

Goal: Reconstruct tree from In&pre

pre-order: AMBDJEFQ ←*Root of the tree must be at first one of post-order*

in-order: BDMJAQFE

↑ *left subtree* ↑ *right subtree*

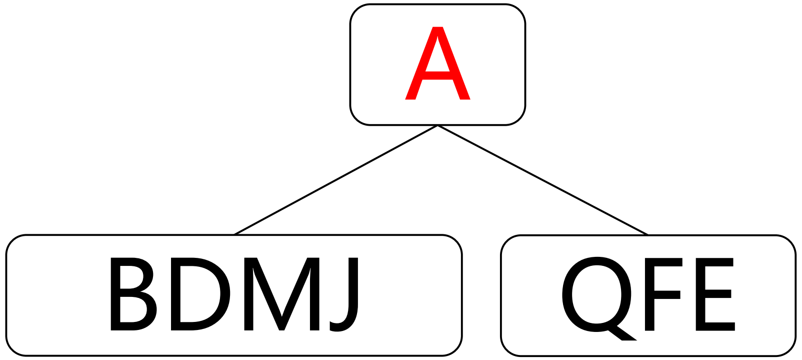
Step1: *Find the root*

Step2: *Reconstruct root-level subtree*

Goal: Reconstruct tree from In&pre

pre-order: AMBDJEFQ
in-order: BDMJAQFE

Step1: *Find the root*
Step2: *Reconstruct root-level subtree*
Step3: *Recursively apply that process to the left&right subtree the until the subtree includes only a node.*



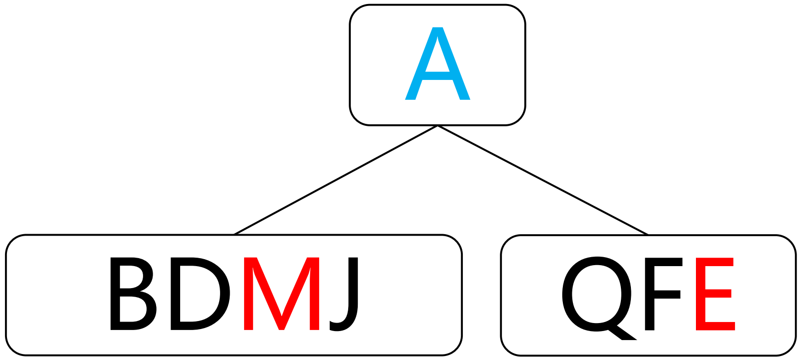
Goal: Reconstruct tree from In&pre

pre-order: A MBDJ EFQ

in-order: BDMJ A QFE

↑

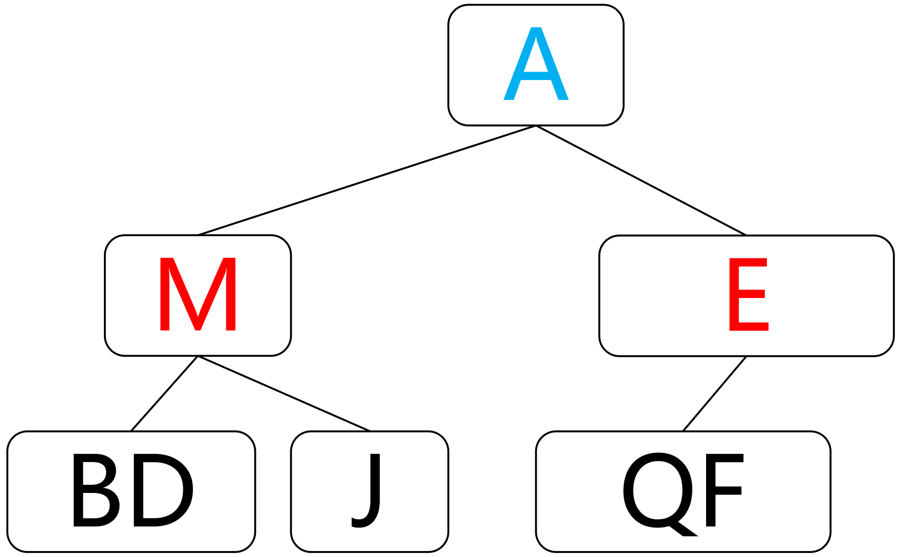
Root of the subtree



Goal: Reconstruct tree from In&pre

pre-order: A MBDJ EFQ

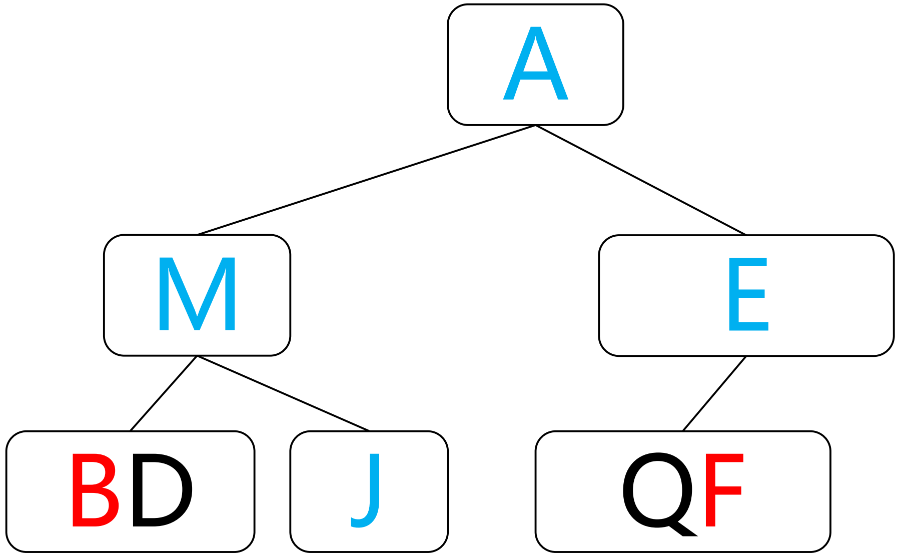
in-order: BDMJ A QFE



Goal: Reconstruct tree from In&pre

pre-order: A M BD J E FQ

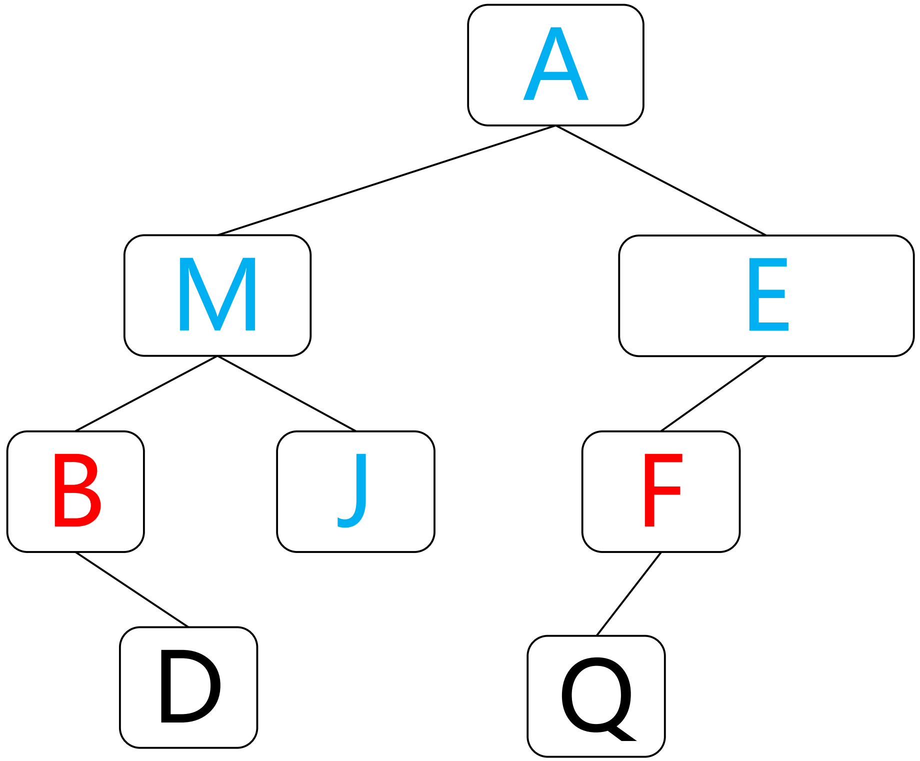
in-order: BD M J A QF E



Goal: Reconstruct tree from In&pre

pre-order: A M BD J E FQ

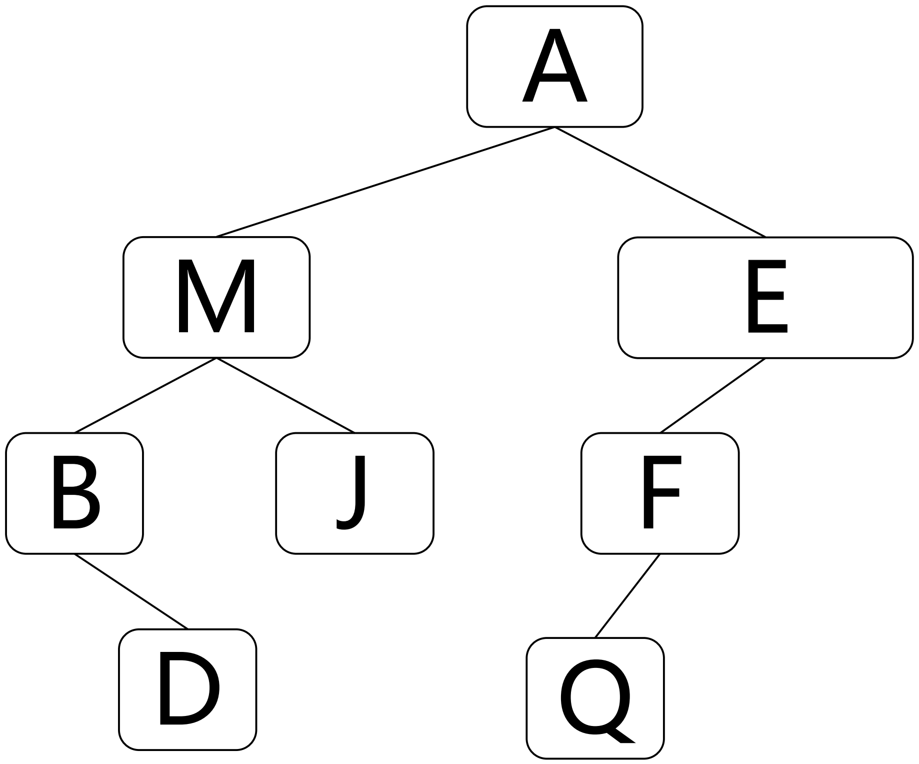
in-order: BD M J A QF E



Goal: Reconstruct tree from In&pre

pre-order: *A M B D J E F Q*

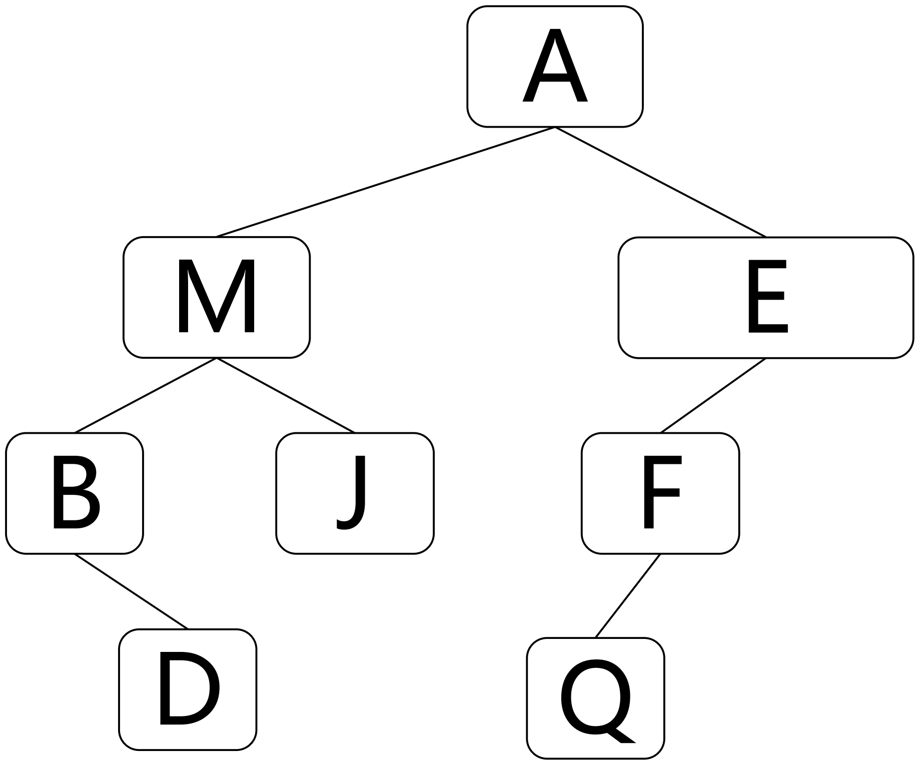
in-order: *B D M J A Q F E*



Goal: Reconstruct tree from In&pre

Post-order?

(A) BDJMFQEA
(B) DBJMFQEA
(C) DBJMQFEA
(D) BDJMQFEA

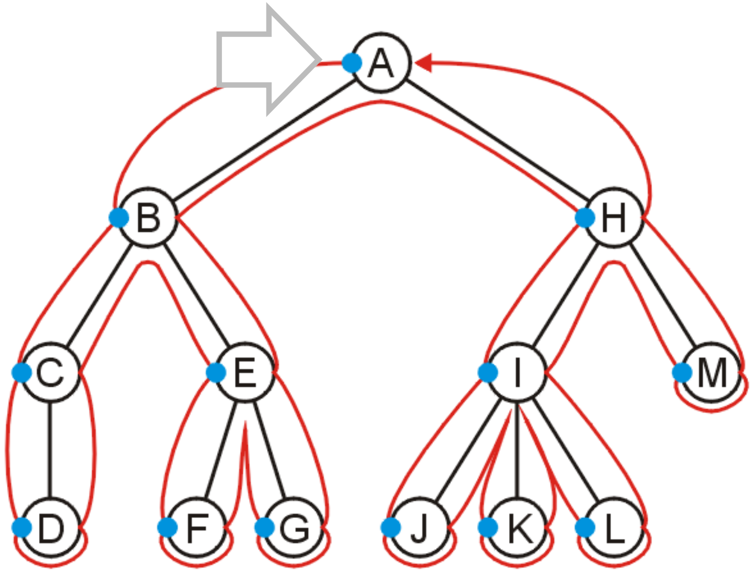


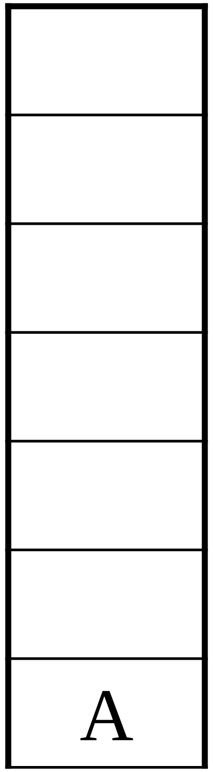
Implementing Depth-First Traversals

Alternatively, we can use a stack:

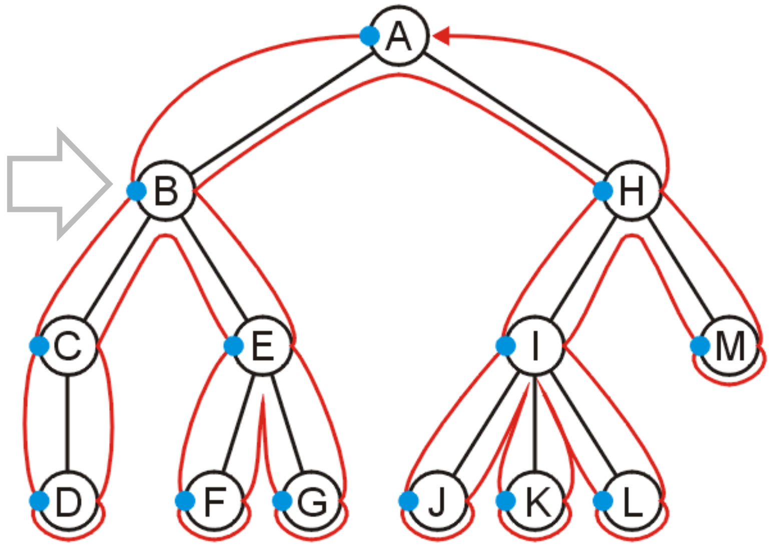
- Create a stack and push the root node onto the stack
- While the stack is not empty:
  - Pop the top node
  - Push all of the children of that node to the top of the stack in reverse order

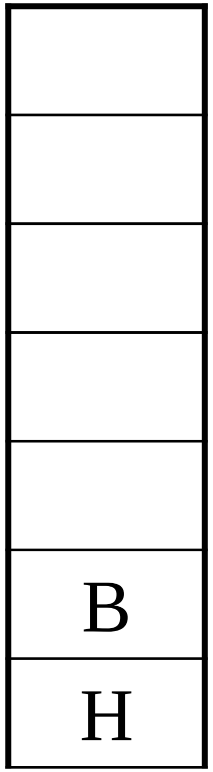
DFS using a Stack



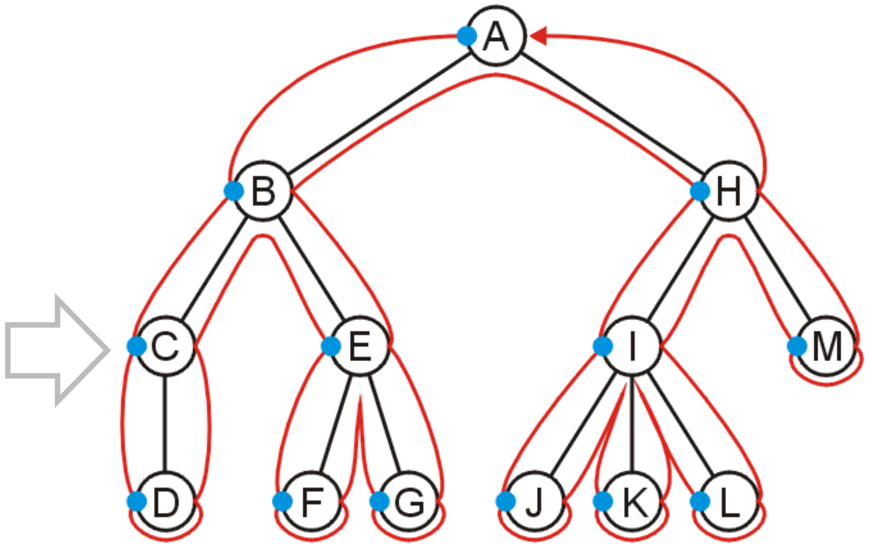


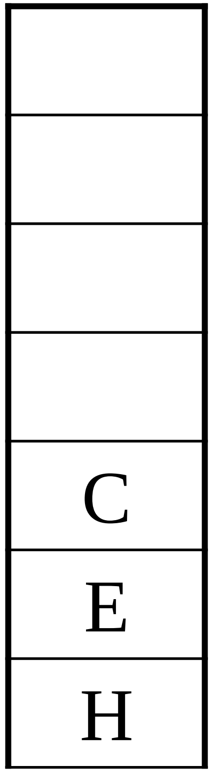
DFS using a Stack



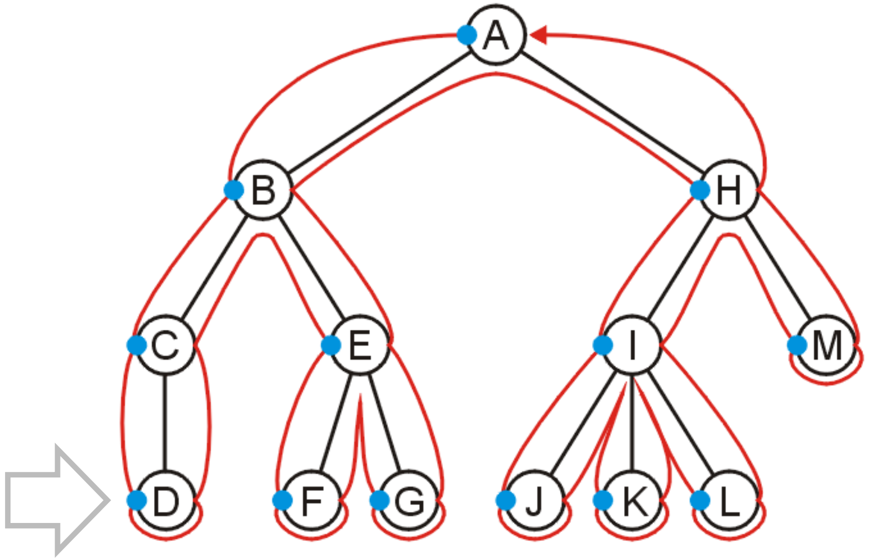


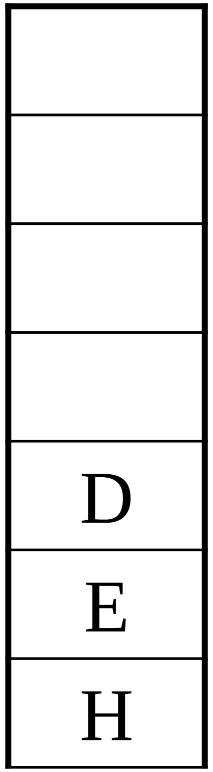
DFS using a Stack



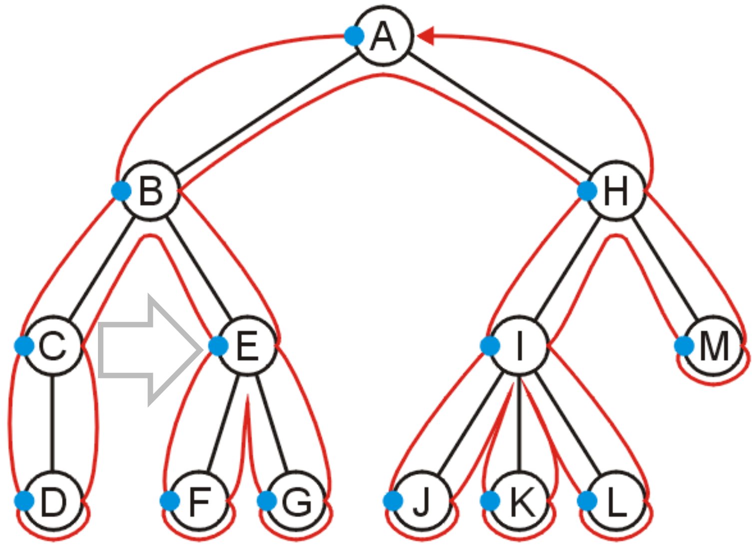


DFS using a Stack



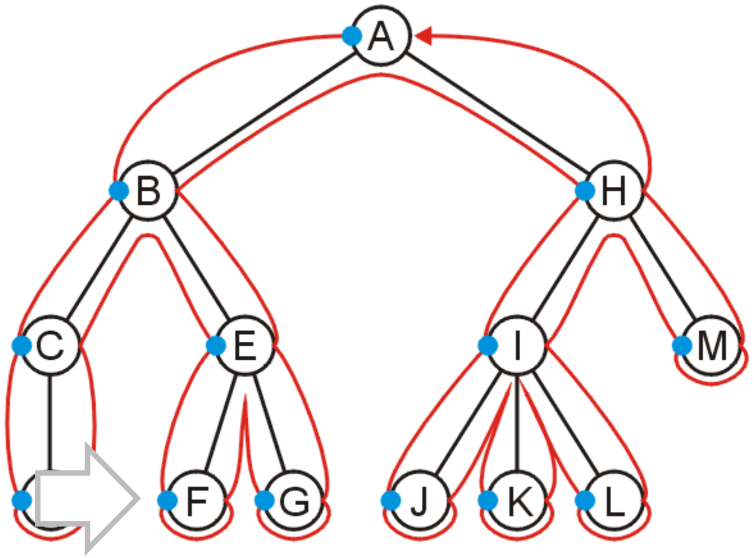


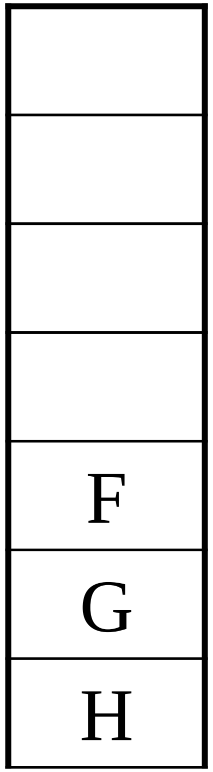
DFS using a Stack



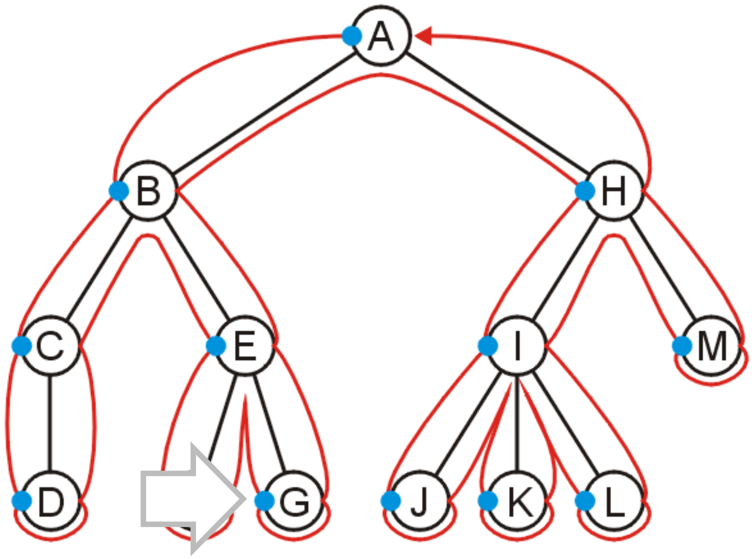
E
H

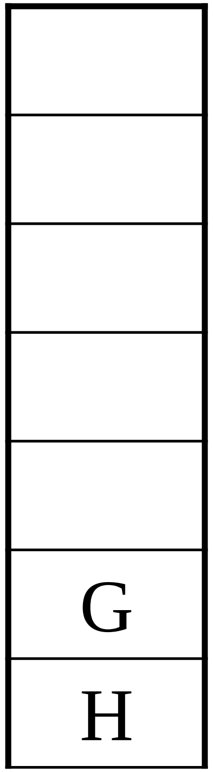
DFS using a Stack



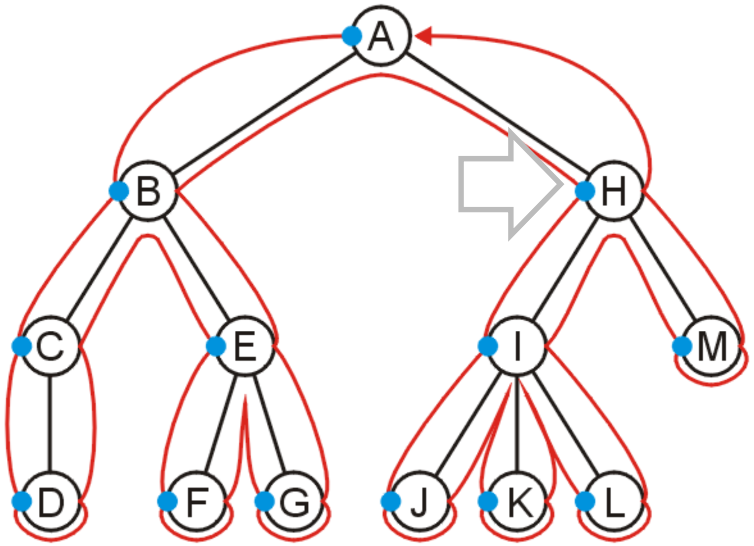


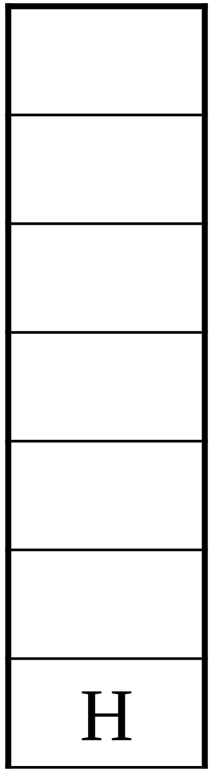
DFS using a Stack



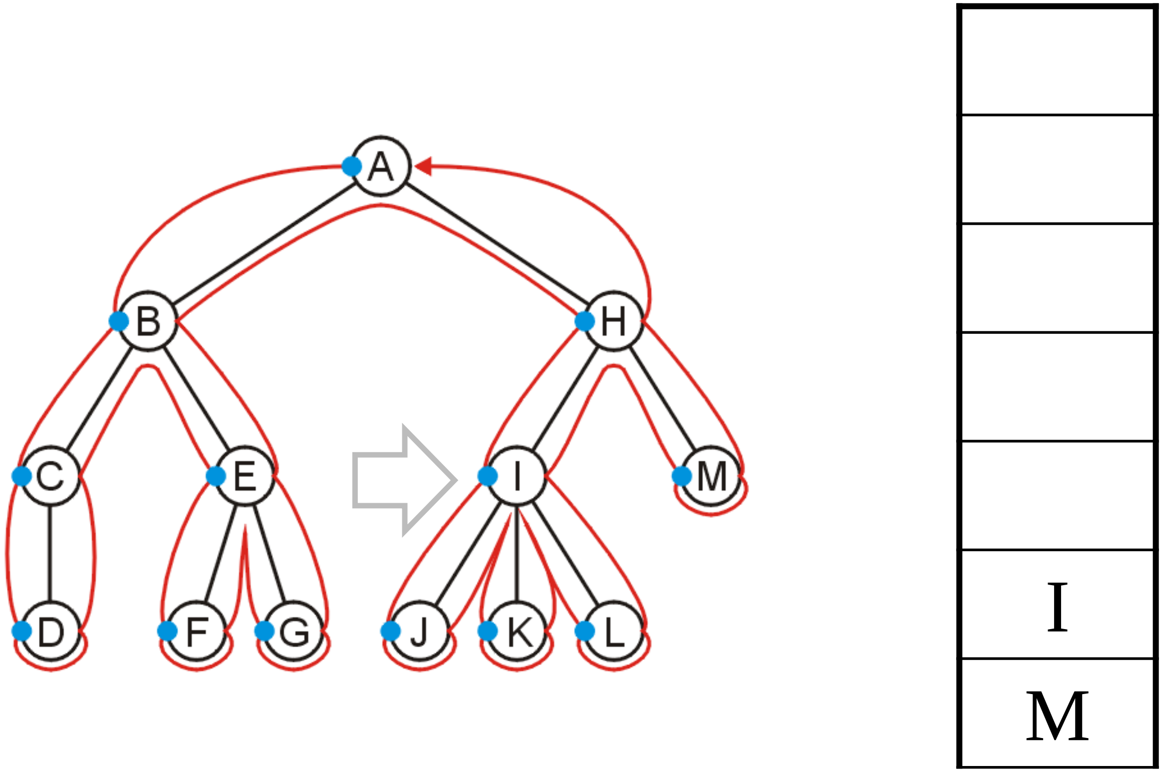


DFS using a Stack

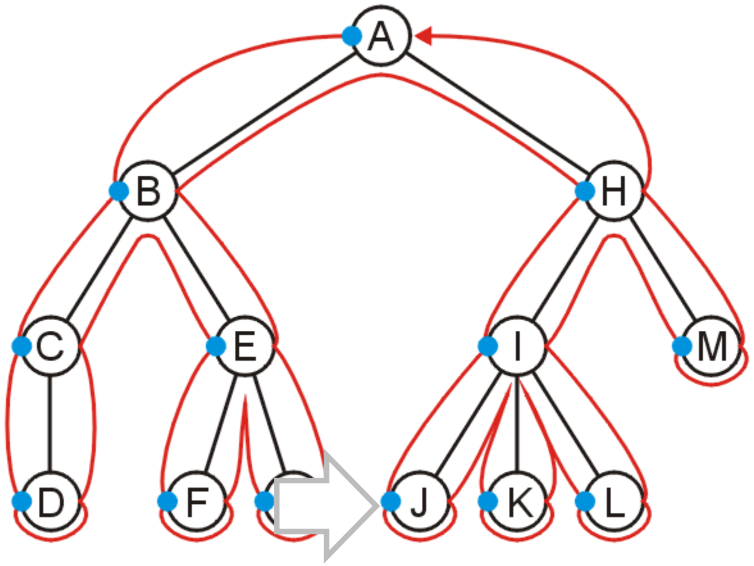




DFS using a Stack

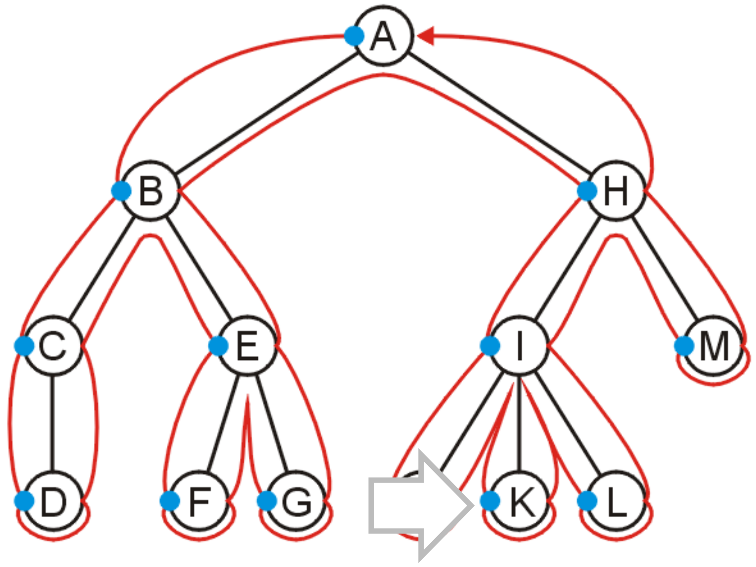


DFS using a Stack



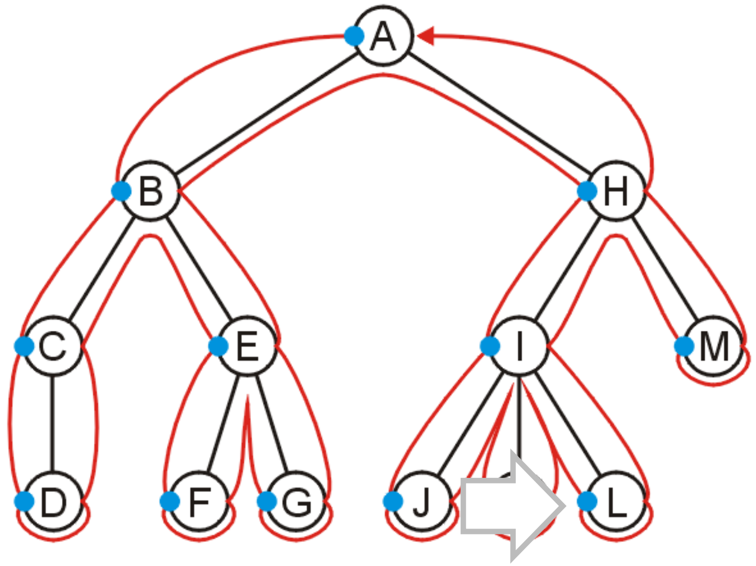
J
K
L
M

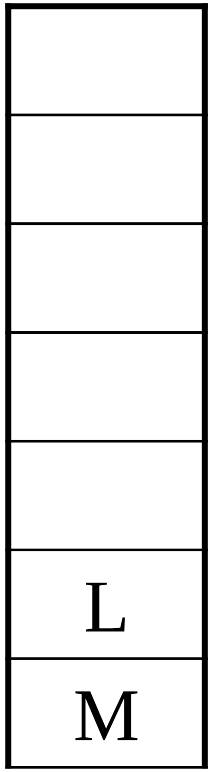
DFS using a Stack



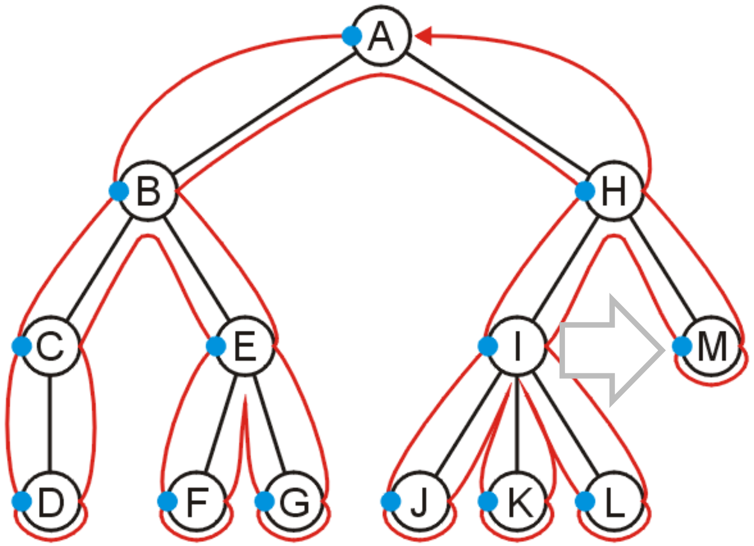
K
L
M

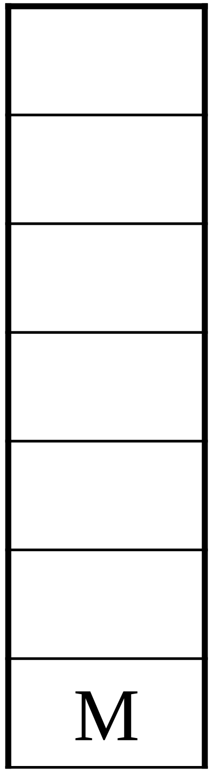
DFS using a Stack



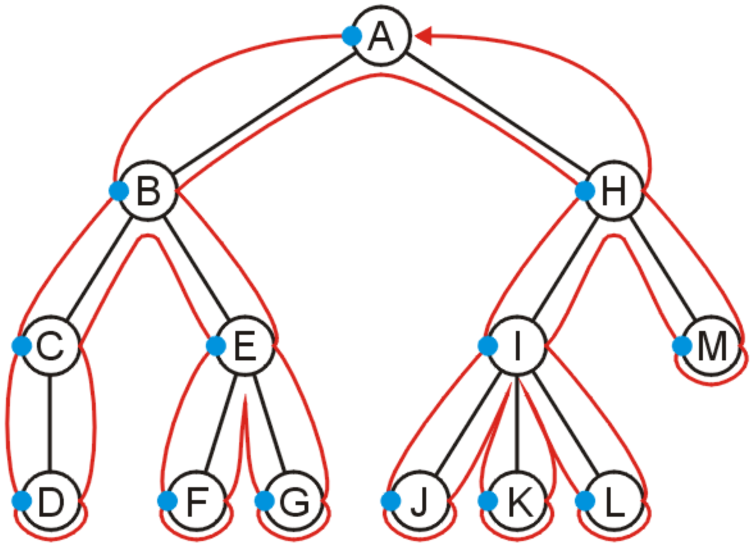


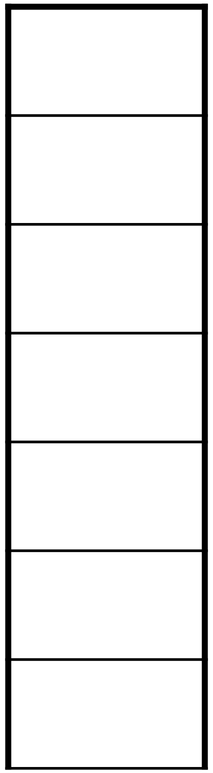
DFS using a Stack





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

DFS using recursion?

- The same complexity?

Implementing Depth-First Traversals

DFS using recursion?

```
void DFS(Node* pRoot)
{
        if (pRoot==NULL) return;
        output(pRoot);
         for(var child-node in node->children)
                        -----------?-----------;
}
```

Implementing Depth-First Traversals

DFS using recursion?

```
void DFS(Node* pRoot)
{
        if (pRoot==NULL) return;
        output(pRoot);
         for(var child-node in node->children)
                    DFS(child-node);
}
```

Quiz Time