VE281

Data Structures and Algorithms

Hashing and Trees

Announcement

- Programming Project One will be announced by tonight.
  - Due in two weeks by 11:59 pm on Nov. 6th , 2012.

- For all assignments, do it yourself. Collaboration is **not** allowed.

- If we believe that you have cheated, such as copying other' s homework or code, we will report your case to **the Honor Council at JI**.

Review

- Collision Resolution: Separate Chaining
- Collision Resolution: Open Addressing
  - Probe with a sequence of hash functions h_0, h_1, h_2, . . .
- Linear Probing

  $$h_i(x) = (h(x) + i) \% M$$

  - **insert**, **find**, **remove**
  - The problem of clustering
- Quadratic Probing

  $$h_i(x) = (h(x) + i^2) \% M$$

- Double Hashing

  $$h_i(x) = (h(x) + i*g(x)) \% M$$

Review

- Average number of comparisons
  - Depends on the **load factor** $L = N/M$, where N is the number of items in the hash table and M is the size of the hash table.
  - We analyze both the case of unsuccessful search ($U(L)$) and the case of successful search ($S(L)$).

Outline

- Hash Table Size and Rehashing
- Trees
- Binary Trees
- Binary Tree Traversal

Determine Hash Table Size

- First, given performance requirements, determine the maximum permissible load factor.

- Example: we want to design a hash table based on linear probing so that on average
  - An unsuccessful search requires no more than 13 compares.
  - A successful search requires no more than 10 compares.

$$U(L) = \frac{1}{2}\left[1 + \left(\frac{1}{1-L}\right)^2\right] \le 13 \Rightarrow L \le \frac{4}{5}$$

$$S(L) = \frac{1}{2}\left[1 + \frac{1}{1-L}\right] \le 10 \Rightarrow L \le \frac{18}{19}$$

$$\boxed{L \le \frac{4}{5}}$$

Determine Hash Table Size

- For a fixed table size, estimate maximum number of items that will be inserted.

- Example: no more than 1000 items.
  - For load factor $L = \frac{N}{M} \leq \frac{4}{5}$, table size

$$M \geq \frac{5}{4} \cdot 1000 = 1250$$

  - Pick M as a **prime** number or an odd number with no prime divisors smaller than 20.

However, sometimes there is no limit on the number of items to be inserted.

Rehashing

Motivation

- With more items inserted, the load factor increases. At some point, it will exceed the threshold (4/5 in the previous example) determined by the performance requirement.

- For the separate chaining scheme, the hash table becomes inefficient when load factor L is too high.
  - If the size of the hash table is fixed, search performance deteriorates with more items inserted.

- Even worse, for the open addressing scheme, when the hash table becomes full, we cannot

Rehashing

- To solve these problems, we need to **rehash**:
  - Create a larger table, scan the current table, and then insert items into new table using the new hash function.

- We can approximately double the size of the current table.

- The single operation of rehashing is time-consuming. However, it does not occur frequently.
  - How should we justify the time complexity of rehashing?

Amortized Analysis

- **Amortized analysis**: A method of analyzing algorithms that considers the entire sequence of operations of the program.
  - The idea is that while certain operations may be costly, they cannot occur at a high frequency to weigh down the entire program, because the number of less costly operations will far outnumber the costly ones in the long run, "paying back" the program over a number of iterations.
  - The cost is **averaged** over a sequence of operations.
  - In contrast, our previous complexity analysis only considers a single operation, e.g.,

Amortized Analysis of Rehashing

- Suppose the threshold of the load factor is 0.5. We will double the table size after reaching the threshold.
- Suppose we start from an empty hash table of size $2M$.

- Assume $O(1)$ operation to insert up to M items.
  - Total cost of inserting the first M items: $O(M)$
- For the $(M+1)$-th item, create a new hash table of size $4M$.
  - Cost: $O(1)$
- Rehash all M items into the new table. Cost: $O(M)$
- Insert new item. Cost: $O(1)$

Total cost for inserting $M+1$ items is $2O(M) + 2O(1) = O(M)$.

Amortized Analysis of Rehashing

- $\boxed{\text{Total cost for inserting } M+1 \text{ items is } O(M).}$
- The average cost to insert $M+1$ items is O(1).
  - Hash table doubling cost is **amortized** over individual inserts.

Hash Table

Conclusion

- Choice of the hash function.
- Collision resolution scheme.
- Size of the hash table and rehashing.

- Time complexity of hash table versus sorted array
  - insert(): $O(1)$ (amortized) versus $O(n)$
  - find(): $O(1)$ versus $O(\log n)$

- When **NOT** to use hash?
  - **Rank search**: return the k-th largest item.
  - **Sort**: return the values in order.

Outline

- Hash Table Size and Rehashing
- Trees
- Binary Trees
- Binary Tree Traversal

Trees

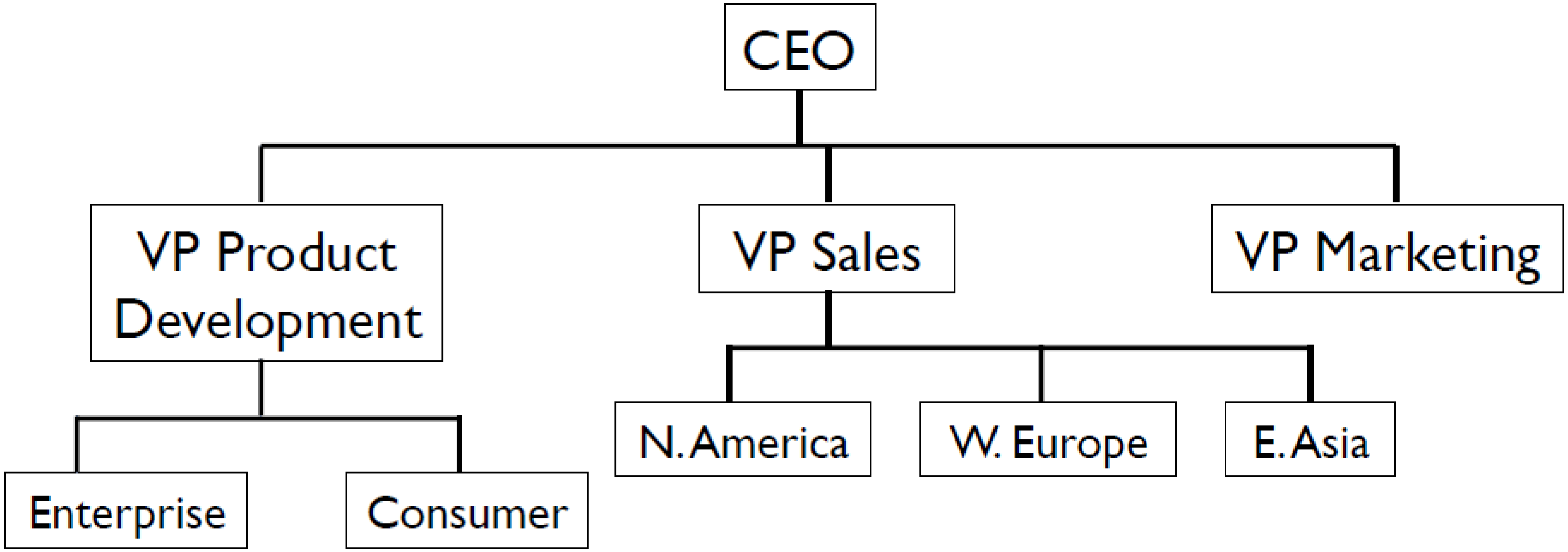
- Tree is an extension of linked list data structure:
  - Each node connects to **multiple** nodes.

- A tree is a "natural" way to represent hierarchical structure and organization.

- A lot of problems in computer science can be solved by breaking it down into smaller pieces and arranging the pieces in some form of hierarchical structure.
  - For example: binary search.

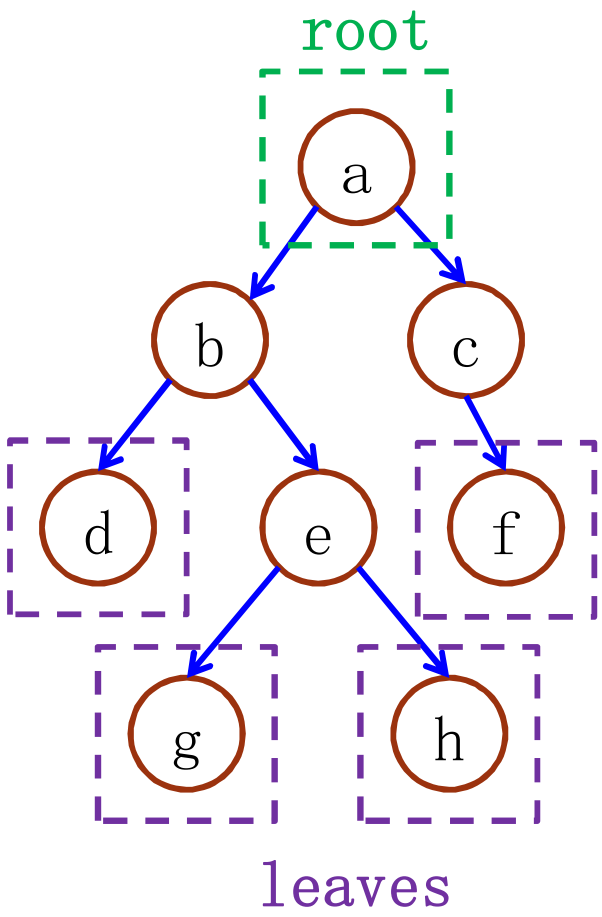
Hierarchical Structures

- Corporation’ s organization chart:

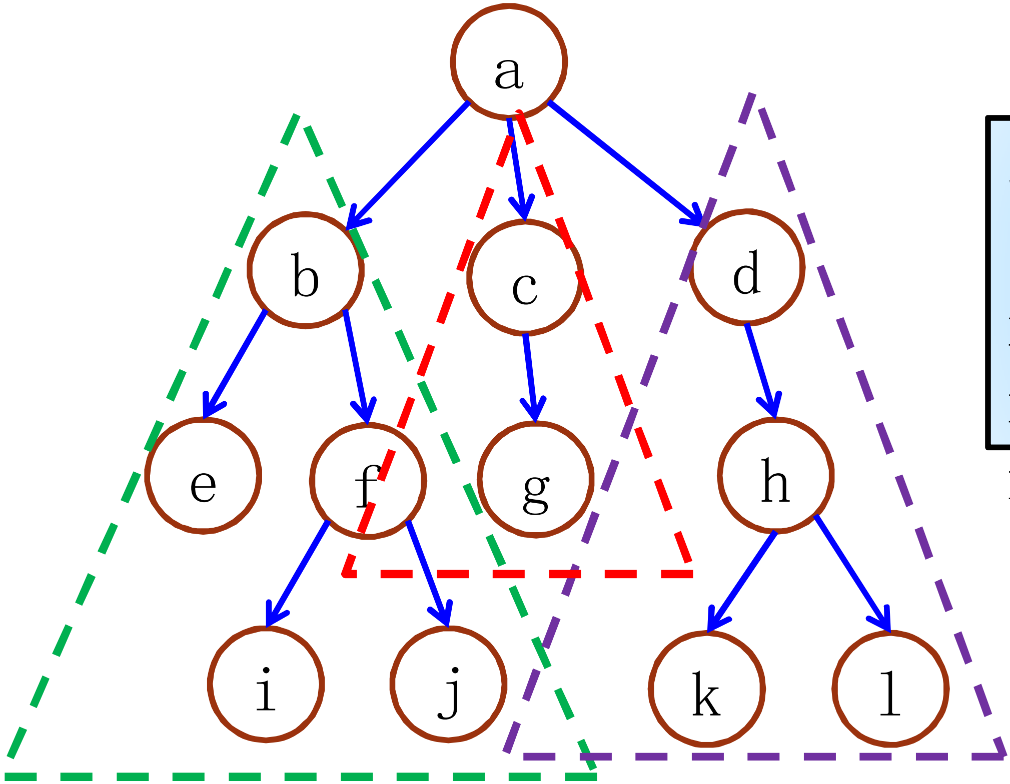


Tree Terminology

- Just like lists, trees are collections of nodes.
- The node at the top of the hierarchy is the **root**.
- Nodes are connected by **edges**.
- Edges define **parent-child** relationship.
  - Root has no parent.
  - All other node has exactly one parent.
- A node with no children is called a **leaf**.



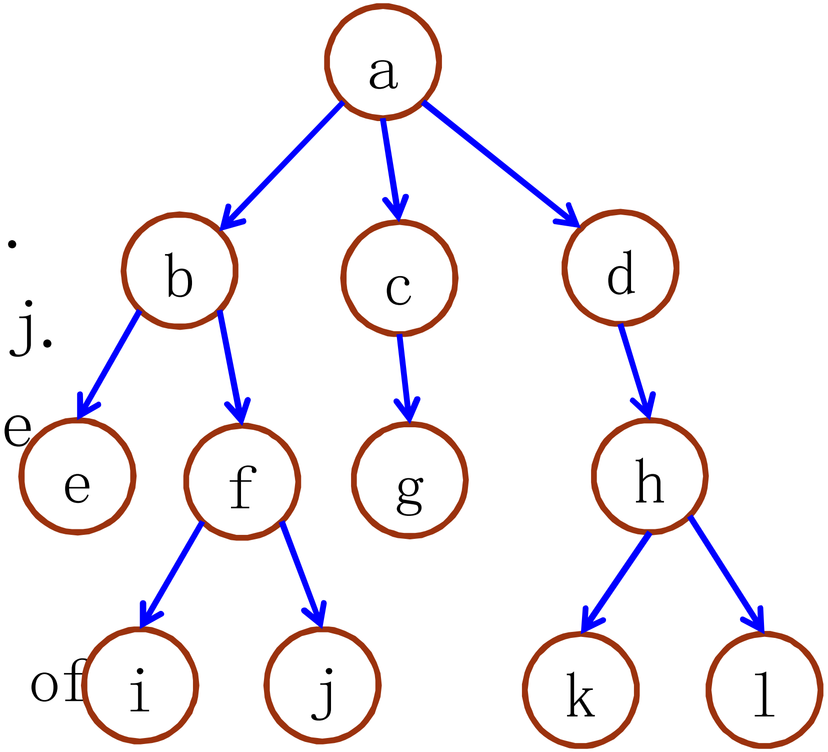
Subtrees



Subtree can be defined for any node in general, not just for the root node.

More Tree Terminology

- f is the child of b.
- b is the parent of f.
- j is the grandchild of b.
- b is the grandparent of j.
- Nodes that share the same parent
  are siblings.
  - b and c are the siblings of d.
  - e is the sibling of f.

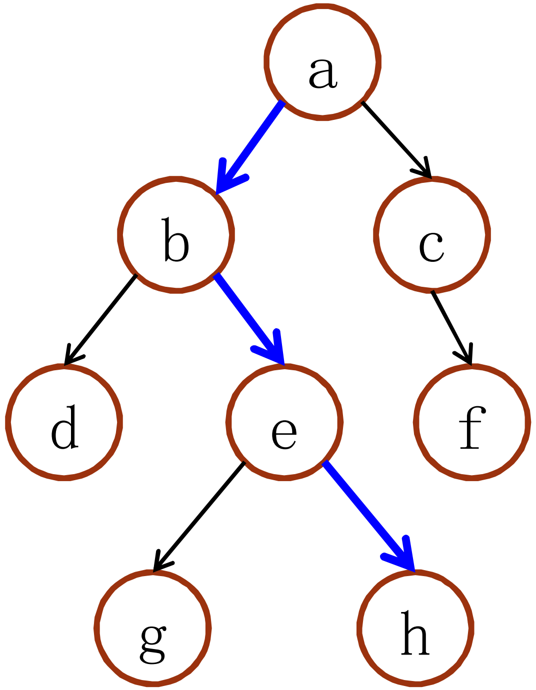


Path

- A path is a sequence of nodes such that the next node in the sequence is a child of the previous.
  - E.g., a→b→e→h is a path.
  - The path length is 3.

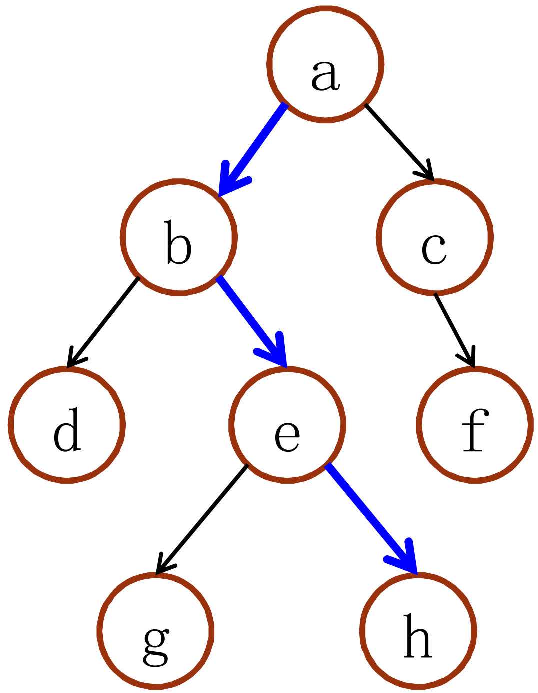
- Path length may be 0, e.g., b going to itself is a path.

- If a path exists between two nodes, then there is a unique path between this pair of nodes.



Ancestors and Descendants

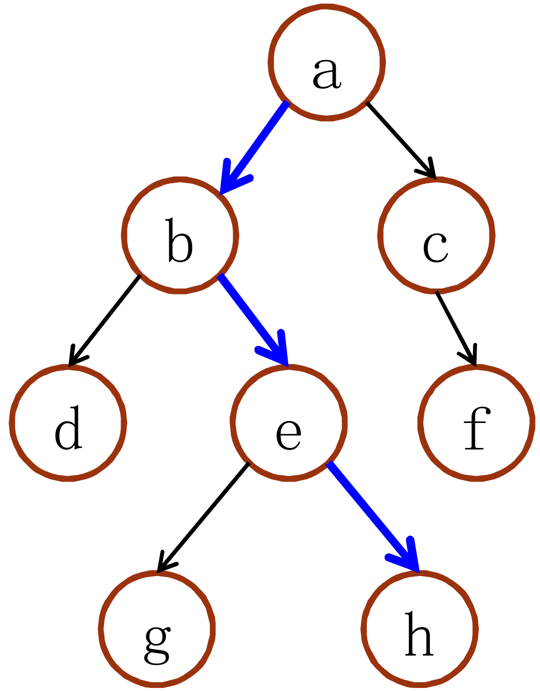
- If there exists a path from a node A to a node B, then A is an ancestor of B and B is a descendant of A.
  - E.g., a is an ancestor of h and h is a descendant of a.



Depth, Level, and Height of a Node

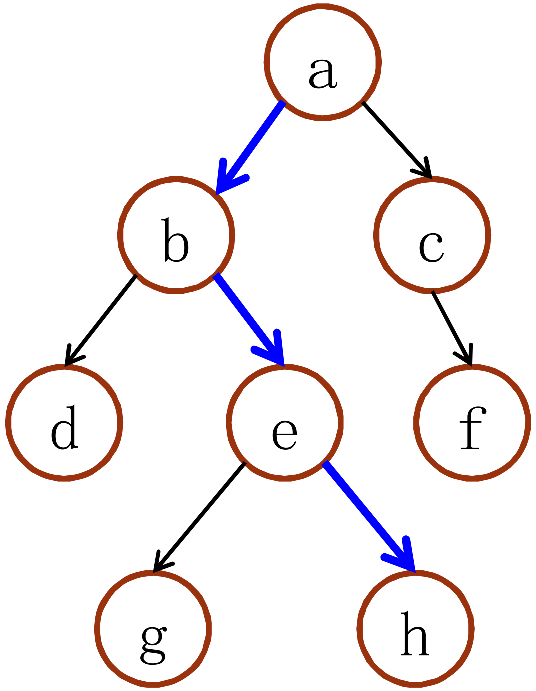
- The depth or level of a node is the length of the unique path from the root to the node.
  - E.g., depth(b)=1, depth(a)=0.

- The height of a node is the length of the longest path from the node to a leaf.
  - E.g., height(b)=2, height(a)=3.
  - All leaves have height zero.



Depth, Level, and Height of a Tree

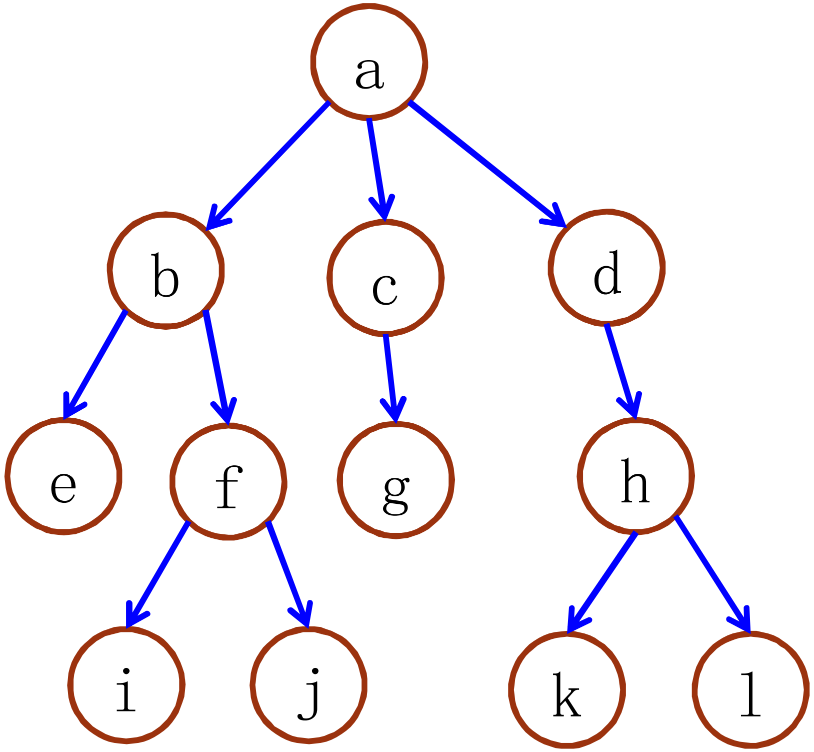
- The height of a tree is the height of its root.
  - This is also known as the depth of a tree.
  - The depth of the tree on the right is 3.



- The number of levels of a tree is the height of the tree plus one.
  - The number of levels of the tree on the right is 4.

Degree

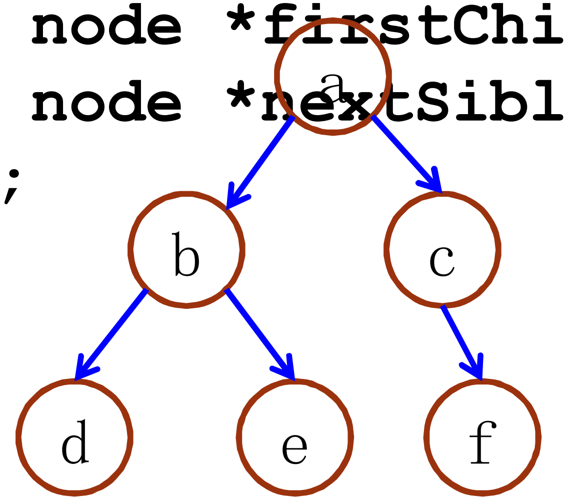
- The degree of a node is the number of children of a node.
  - E.g., degree(a) = 3, degree(c) = 1.
- The degree of a tree is the maximum degree of a node in the tree.
  - The degree of the tree on the right is 3.

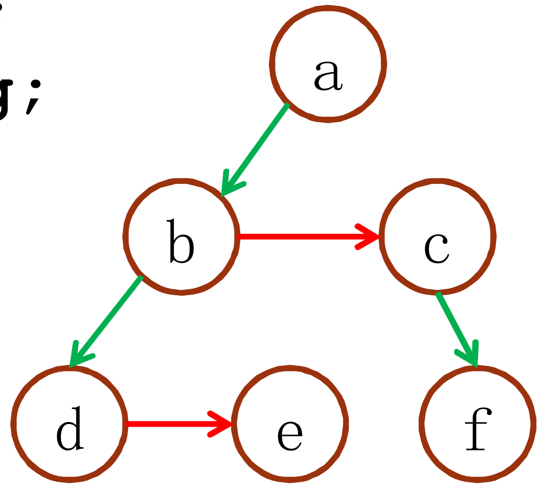


A Simple Implementation of Tree

- Each node is part of a linked list of siblings.
- Additionally, each node stores a pointer to its first child.

```
struct node {
  Item item;
  node *firstChild;
  node *nextSibling;
};
```





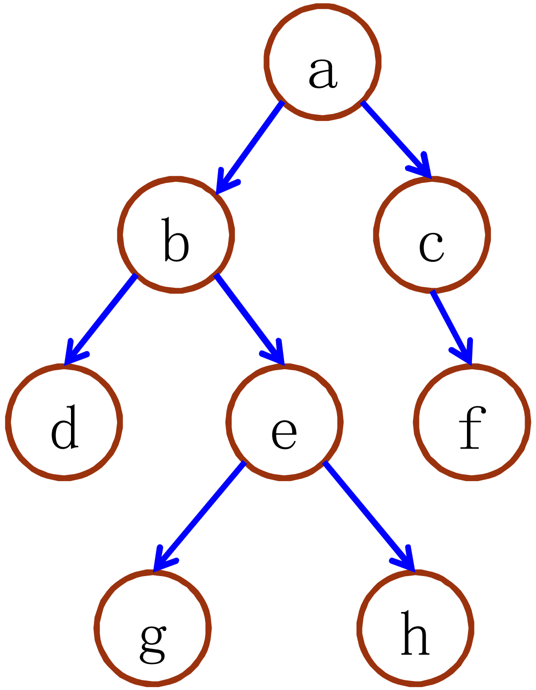
Outline

- Hash Table Size and Rehashing
- Trees
- Binary Trees
- Binary Tree Traversal

Binary Tree

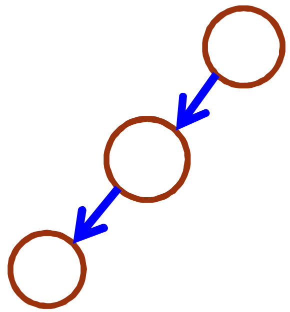
- Every node can only have **at most two** children.

- An empty tree is a special binary tree.



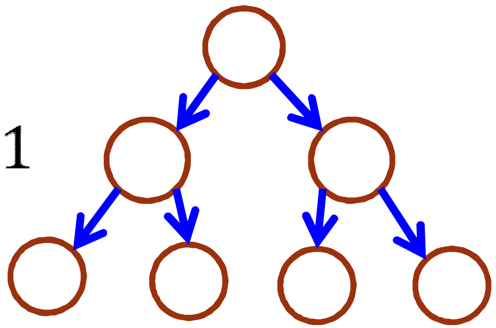
Binary Tree Properties

- What is the **minimum** number of nodes in a binary tree of height h (i.e., has $h+1$ levels)?
  - Answer: **At least** one node at each level.
  - $h+1$ levels means at least $h+1$ nodes.



- What is the **maximum** number of nodes in a binary tree of height h (i.e., has $h+1$ levels)?
  - Answer: At most 2^h nodes at level h.
  - Maximum number of nodes is

$$1+2+2^2+\cdots+2^h=2^{h+1}-1$$



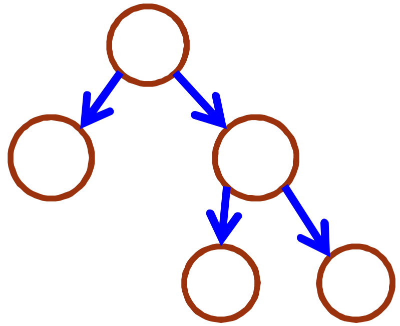
Number Of Nodes and Height

- Let n be the number of nodes in a binary tree whose height is h (i.e., has $h + 1$ levels).
  - We have $h + 1 \leq n \leq 2^{h+1} - 1$.

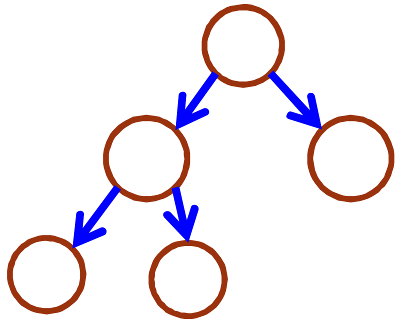
- Question: given n nodes, what is the height h of the tree?
  - $\log_2(n + 1) - 1 \leq h \leq n - 1$

Types of Binary Trees

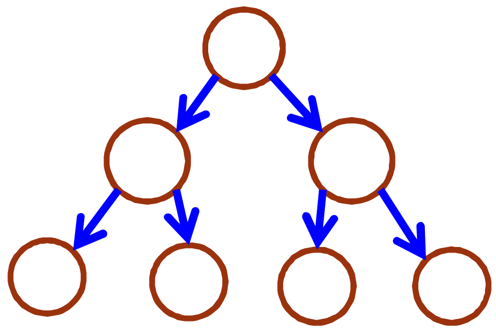
- A binary tree is **proper** if every node has 0 or 2 children.



- A binary tree is **complete** if:



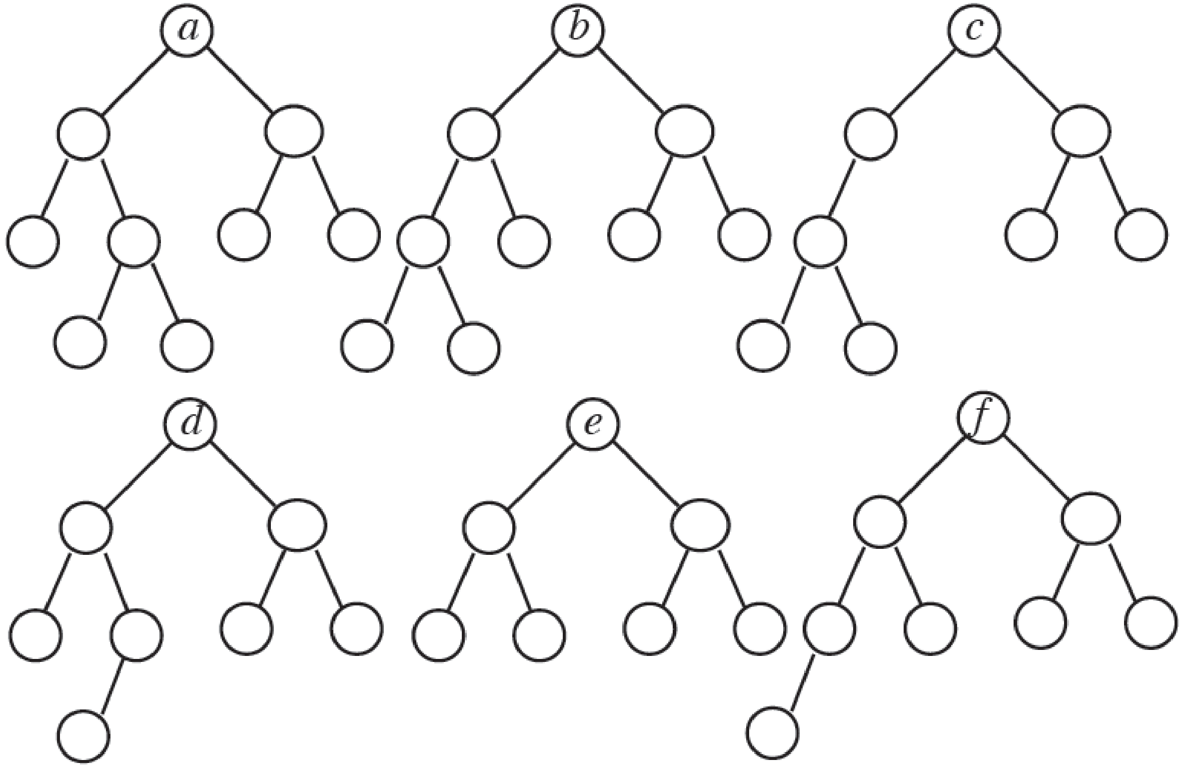
1. every level **except** the lowest is fully populated, and
2. the lowest level is populated from left to right.



- A binary tree is

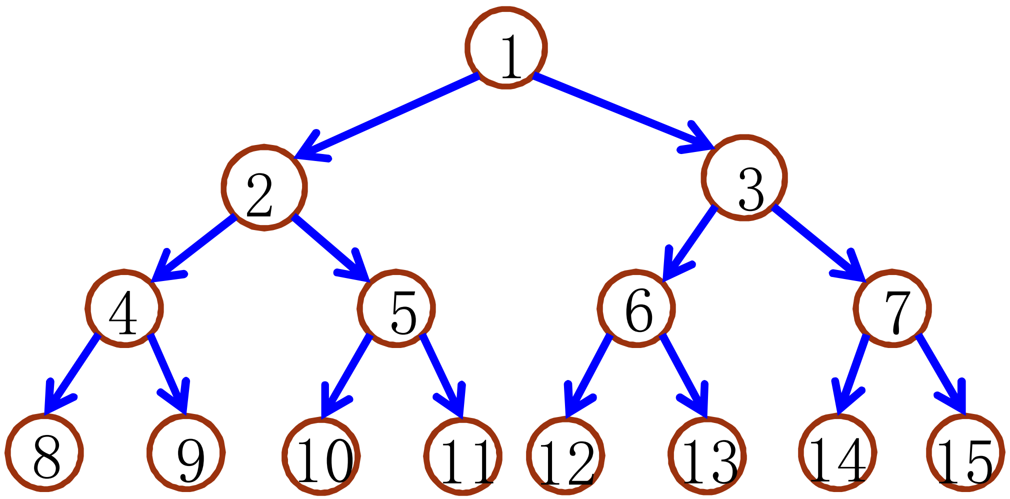
Exercises

- Identify any proper, complete, and perfect binary trees below:

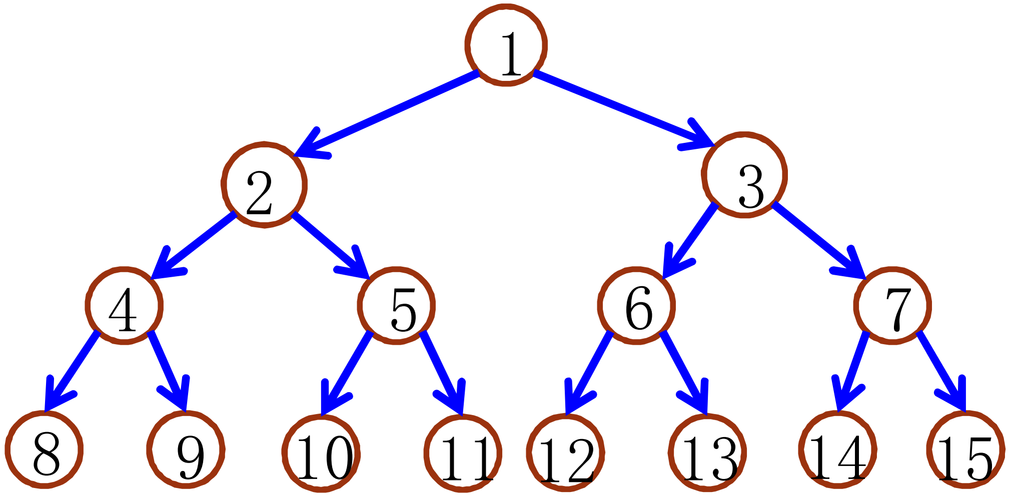


Numbering Nodes In a Perfect Binary Tree

- Numbering nodes from 1 to $2^{h+1} - 1$.
- Numbering **from top to bottom** level.
- Within a level, numbering **from left to right**.



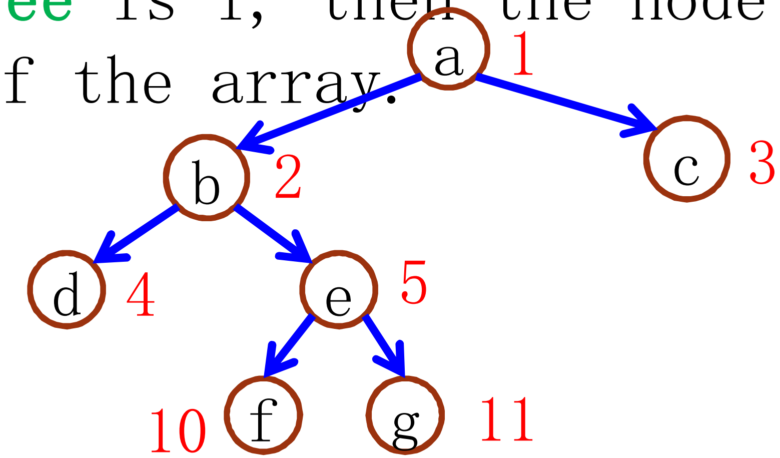
Numbering Nodes In a Perfect Binary Tree

- 

- What is the parent of node i?
  - For $i \neq 1$, it is $i/2$. For node 1, it has no parent.
- What is the left child of node i? Let n be the number of nodes.
  - If $2i \leq n$, it is $2i$; If $2i > n$, no left child.
- What is the right child of node i?
  - If $2i + 1 \leq n$, it is $2i + 1$; If $2i + 1 > n$, no right child.

Representing Binary Tree Using Array

- Based on the numbering scheme for a **perfect** binary tree.
- If the number of the node **in a perfect binary tree** is i, then the node is put at index i of the array.

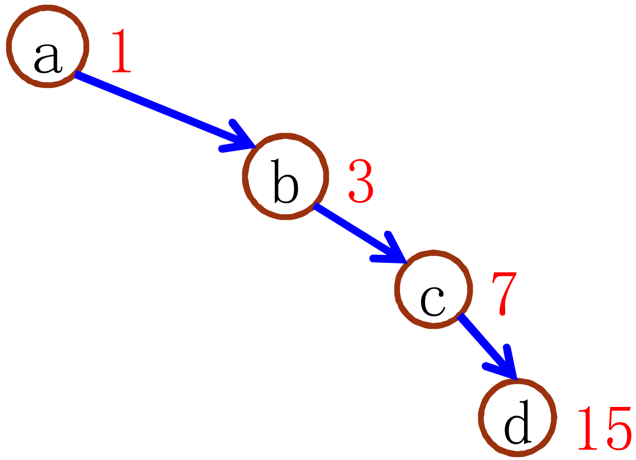


–	a	b	c	d	e	–	–	–	–	f	g
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Representing Binary Tree Using Array

Space Efficiency

- How would you represent a **right-skewed** binary tree?



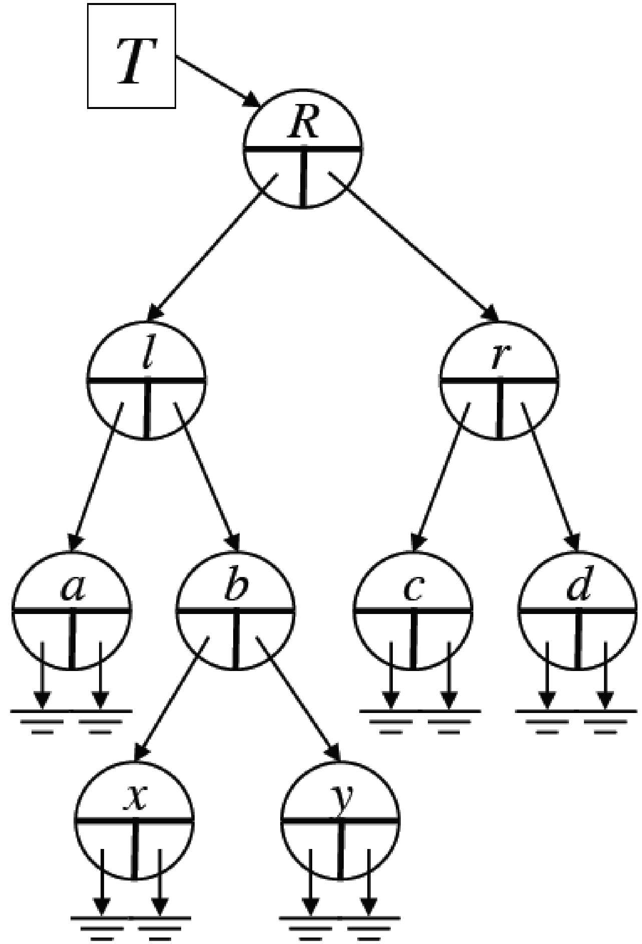
–	a	–	b	–	–	–	c	–	–	–	–	–	–	–	d
[0]	[1]		[3]		[5]		[7]		[9]		[11]		[13]		[15]

An n node binary tree needs an array whose length is between $n + 1$ and 2^n.

Representing Binary Tree Using Linked Structure

```
struct node {
  Item item;
  node *left;
  node *right;
};
```

- **left/right** points to a left/right subtree.
  - If the subtree is an empty one, the pointer points to NULL.
- For a leaf node, both its **left** and **right** pointers are NULL.



Outline

- Hash Table Size and Rehashing
- Trees
- Binary Trees
- Binary Tree Traversal

Binary Tree Traversal

- Many binary tree operations are done by performing a **traversal** of the binary tree.
- In a traversal, each node of the binary tree is visited **exactly once**.
- During the visit of a node, all actions (making a clone, displaying, evaluating the operator, etc.) with respect to this node are taken.

Binary Tree Traversal Methods

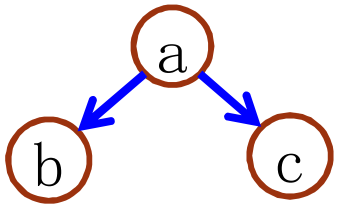
- Depth-first traversal
  - Pre-order
  - Post-order
  - In-order

- Level order traversal

Pre-Order Depth-First Traversal

Procedure

- Visit node
- Visit its left subtree
- Visit its right subtree



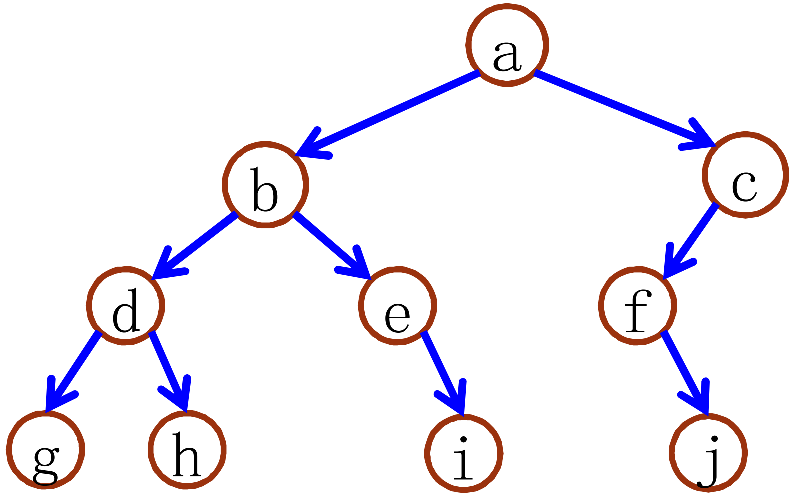
a→b→c

```
void preOrder(treeNode *n) {
  if(!n) return;
  visit(n);
  preOrder(n->left);
  preOrder(n->right);
}
```

Pre-Order Depth-First Traversal

Example

a
b
d
g
h
e
i
c
f
j

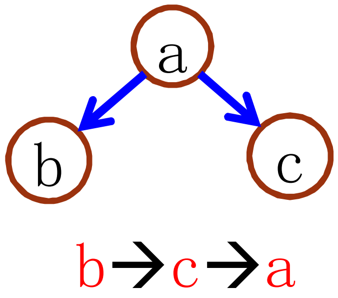


a→b→d→g→h→e→i→c→f→j

Post-Order Depth-First Traversal

Procedure

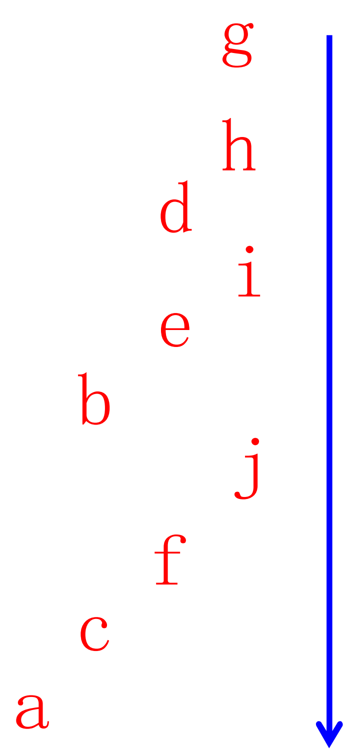
- Visit the left subtree
- Visit the right subtree
- Visit node

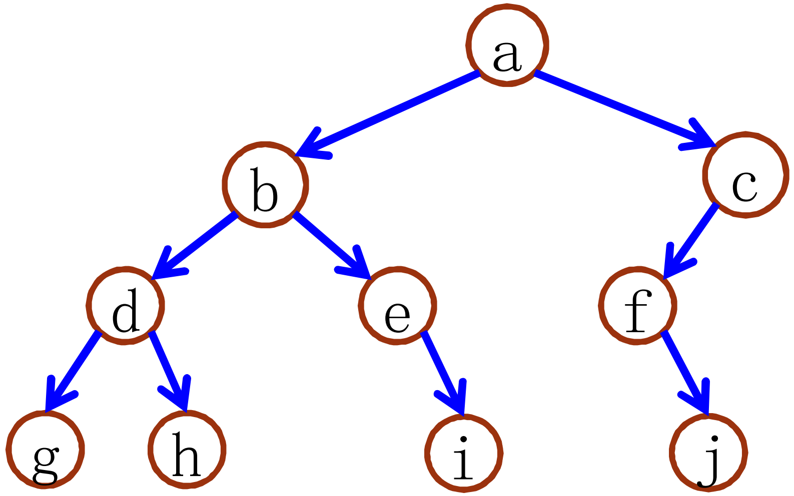



```
void postOrder(treeNode *n) {
  if(!n) return;
  postOrder(n->left);
  postOrder(n->right);
  visit(n);
}
```

Pre-Order Depth-First Traversal

Example



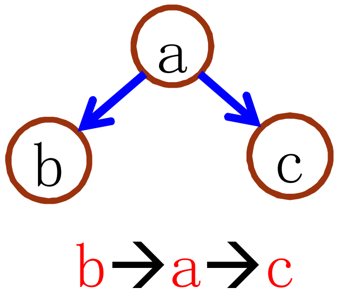


g→h→d→i→e→b→j→f→c→a

In-Order Depth-First Traversal

Procedure

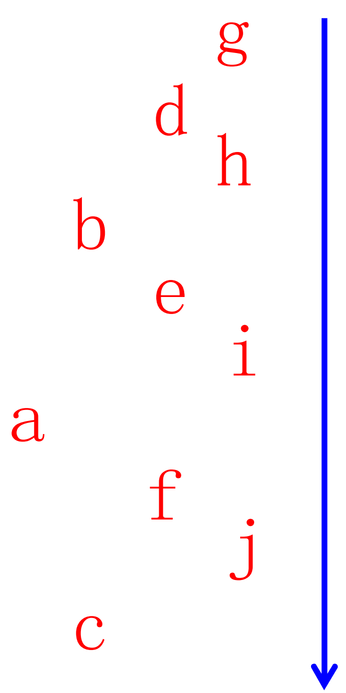
- Visit the left subtree
- Visit node
- Visit the right subtree

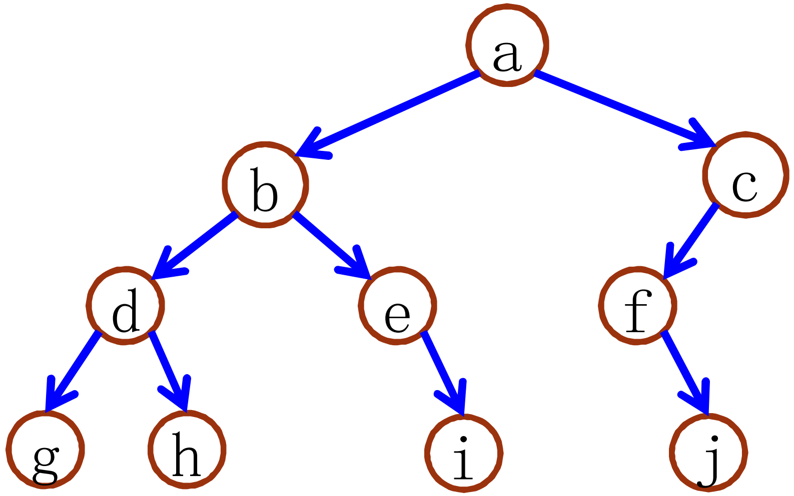



```
void inOrder(treeNode *n) {
  if(!n) return;
  inOrder(n->left);
  visit(n);
  inOrder(n->right);
}
```

In-Order Depth-First Traversal

Example





g→d→h→b→e→i→a→f→j→c