VE281

Data Structures and Algorithms

Bloom Filter; Tree; Binary Tree Traversal

Announcement

- Written Assignment Three Posted
  - On hashing and binary trees
  - Due time: 5:40 pm on Nov. 2, 2016

Midterm Exam

- Time: Oct. 31st, in class.
- Location: see Canvas announcement.
- A written exam.
  - Like our written assignments.
  - Pseudo-code OK (but make sure we can **understand** it!)
- Closed book and closed notes.
- Only basic calculator is allowed.
  - No other electronic devices, including laptops and cell phones.
  - We will show a clock on the screen.
- Abide by the **Honor Code**!

Midterm Topics

- Asymptotic Algorithm Analysis
- Sorting
  - Comparison sort
  - Non-comparison sort
- Linear-time selection
- Hashing
- Tree and Binary Tree Traversal

Outline

- Bloom Filter

- Trees
- Binary Trees
- Binary Tree Traversal

Review: Bloom Filter

- Supports **fast insert** and **find**
- Comparison to hash tables:
  - Pros: more space efficient
  - Cons:

1. Can't store an associated object
2. No deletion (There are variations support deletion, but this operation is complicated)
3. Small **false positive** probability: may say x has been inserted even if it hasn't been
   - But no false negative (x is inserted, but says not inserted)

Bloom Filter Implementation: Components

- An array of n **bits**. Each bit 0 or 1
  - $n = b|S|$, where b is small real number. For example, $b = 8$ for 32-bit IP address (That's why it is space efficient)
- k hash functions $h_1, \ldots, h_k$, each mapping inside $\{0,1, \ldots, n-1\}$.
  - k usually small.
  - These k functions can be randomly chosen from a universal family of hash functions

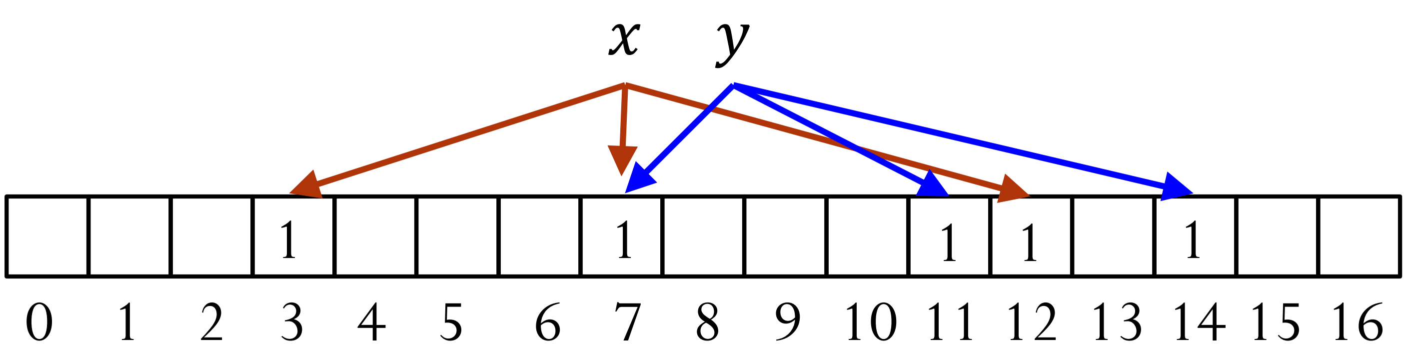
Bloom Filter Insert

- Initially, the array is all-zero.
- Insert x: For $i = 1,2, \ldots, k$, set $A[h_i(x)] = 1$
  - No matter whether the bit is 0 or 1 before

<u>Example</u>: $n = 17$, 3 hash functions

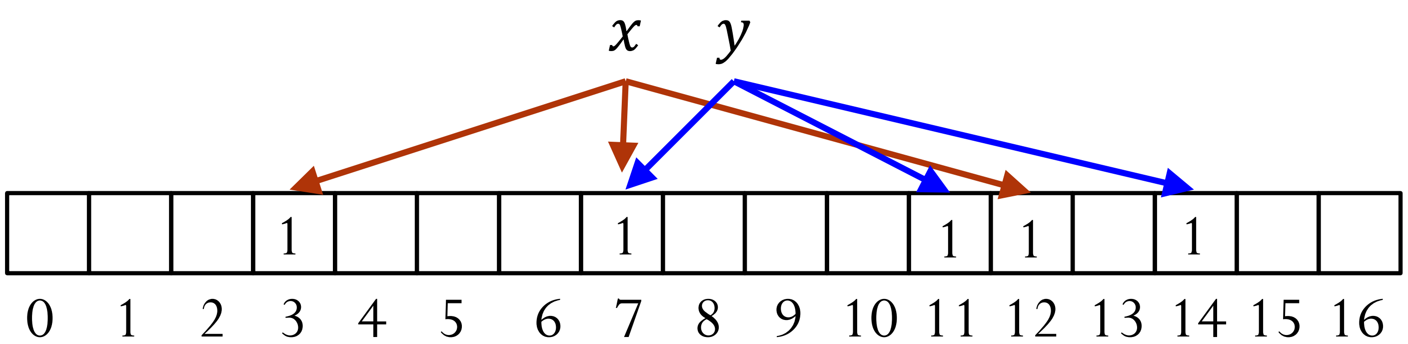
$h_1(x) = 7, h_2(x) = 3, h_3(x) = 12$

$h_1(y) = 11, h_2(y) = 14, h_3(y) = 7$



Bloom Filter Find

- Find x: return true if and only if $A[h_i(x)] = 1, \forall i = 1, \ldots, k$



Suppose $h_1(x) = 7, h_2(x) = 3, h_3(x) = 12$. Find x? Yes!

Suppose $h_1(z) = 3, h_2(z) = 11, h_3(z) = 5$. Find z? No!

- No false negative: if x was inserted, find(x) guaranteed to return true
- False positive possible: consider $h_1(w) = 11, h_2(w) = 12, h_3(w) = 7$ in the above example

Heuristic Analysis of Error Probability

- Intuition: should be a trade-off between space (array size) and false positive probability
  - Array size decreases, more reuse of bits, false positive probability increases
- Goal: analyze the false positive probability
- Setup: Insert data set S into the Bloom filter, use k hash functions, array has n bits
- Assumption: All k hash functions map keys uniformly random and these hash functions are independent

Probability of a Slot Being 1

- For an arbitrary slot j in the array, what's the probability that the slot is 1?

- Consider when slot j is 0
  - Happens when $h_i(x) \neq j$ for all $i = 1, \dots, k$ and $x \in S$
  - $\Pr(h_i(x) \neq j) = 1 - \frac{1}{n}$
  - $\Pr(A[j] = 0) = \left(1 - \frac{1}{n}\right)^{k|S|} \approx e^{-\frac{k|S|}{n}} = e^{-\frac{k}{b}}$
    - $b = \frac{n}{|S|}$ denotes # of bits per object
- $\Pr(A[j] = 1) \approx 1 - e^{-\frac{k}{b}}$

False Positive Probability

- For x not in S, the false positive probability happens when all $A[h_i(x)] = 1$ for all $i = 1, \ldots, k$
  - The probability is $\epsilon \approx \left(1 - e^{-\frac{k}{b}}\right)^k$
- For a fixed b, ϵ is minimized when $k = (\ln 2) \cdot b$
- The minimal error probability is $\epsilon \approx \left(\frac{1}{2}\right)^{\ln 2 \cdot b} \approx 0.6185^b$
  - Error probability decreases exponentially with b
- Example: $b = 8$, could choose k as 5 or 6. Min error probability $\approx$ 2%

Outline

- Bloom Filter

- Trees
- Binary Trees
- Binary Tree Traversal

Trees

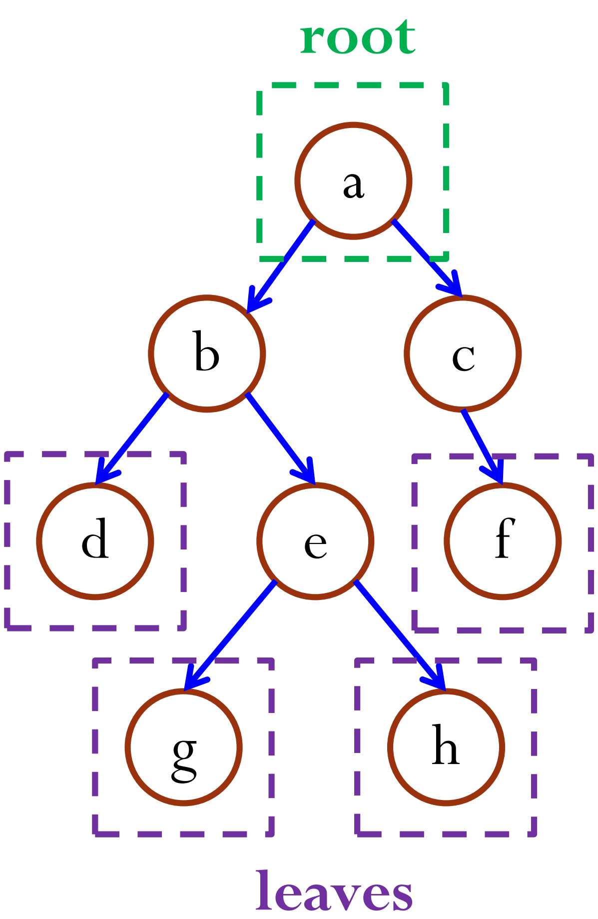
- Tree is an extension of linked list data structure:
  - Each node connects to **multiple** nodes.

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

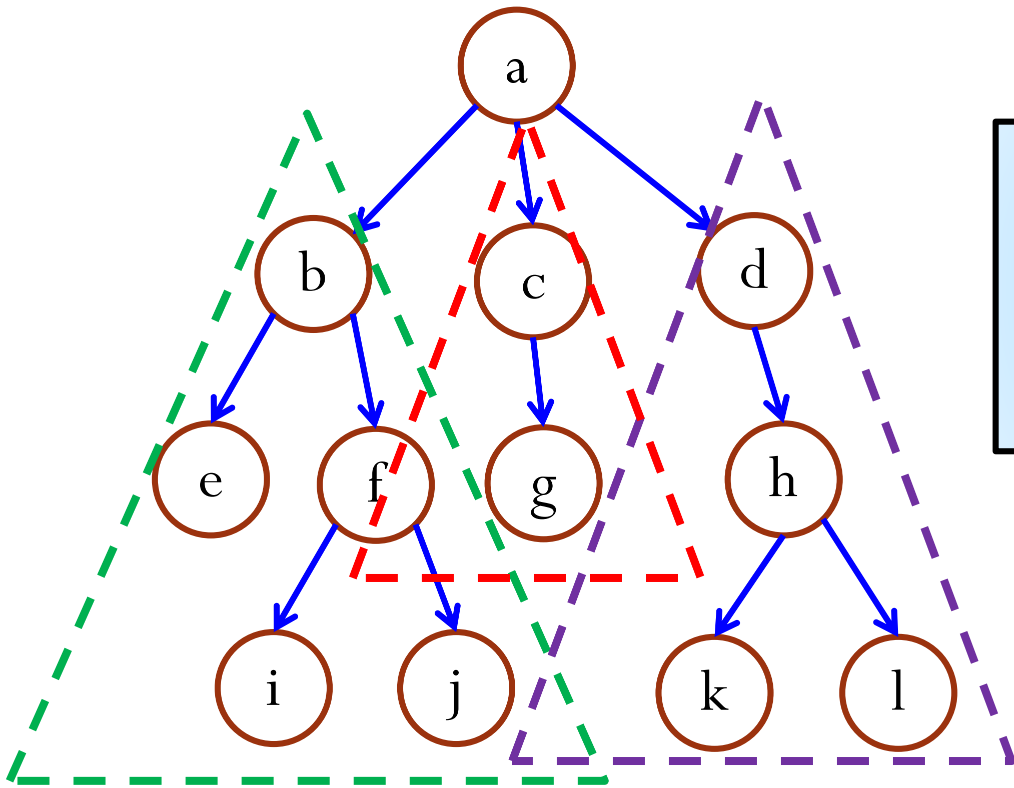
- Many problems in computer science can be solved by breaking it down into smaller pieces and arranging the pieces in some form of hierarchical structure.
  - For example: merge sort.

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

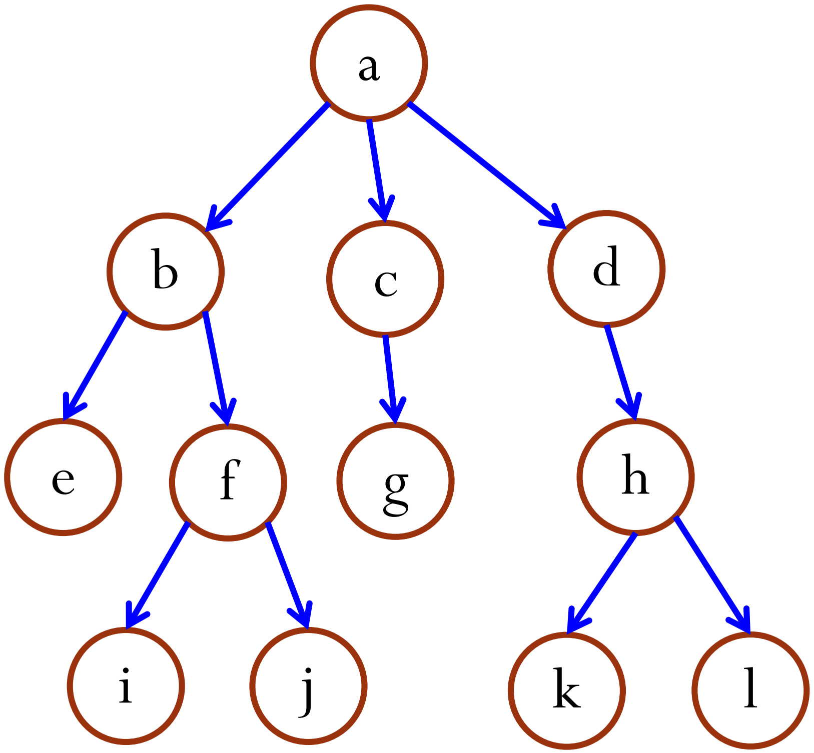


Subtree can be defined for any node in general, not just for the root node.

More Tree Terminology

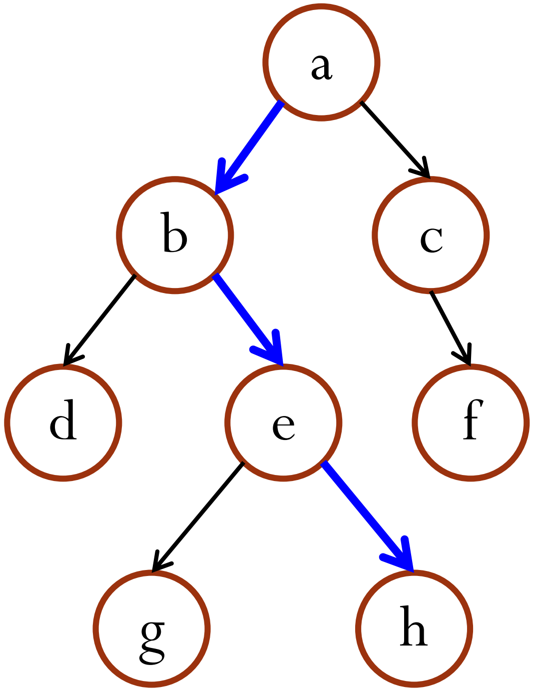
- f is the **child** of b.
- b is the **parent** of f.

- Nodes that share the same parent are **siblings**.
  - b and c are the **siblings** of d.
  - e is the **sibling** of f.



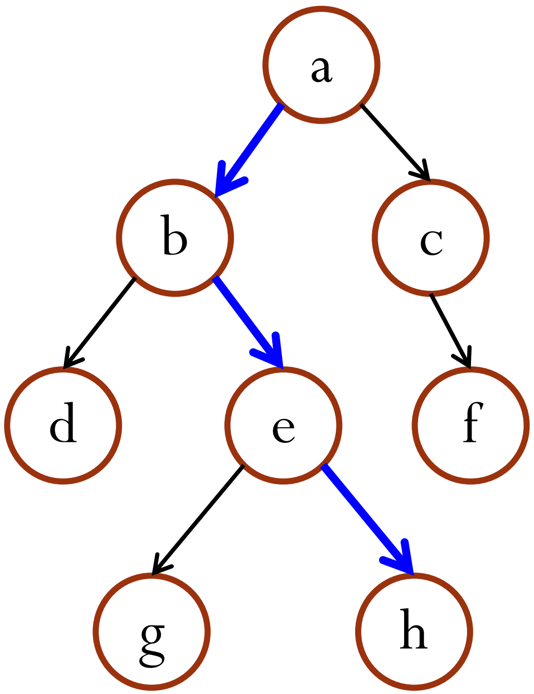
Path

- 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 and its length is 0.
- **<u>Claim</u>**: If there exists a path between two nodes, then this path is the **unique** path between these two nodes.



Ancestors and Descendants

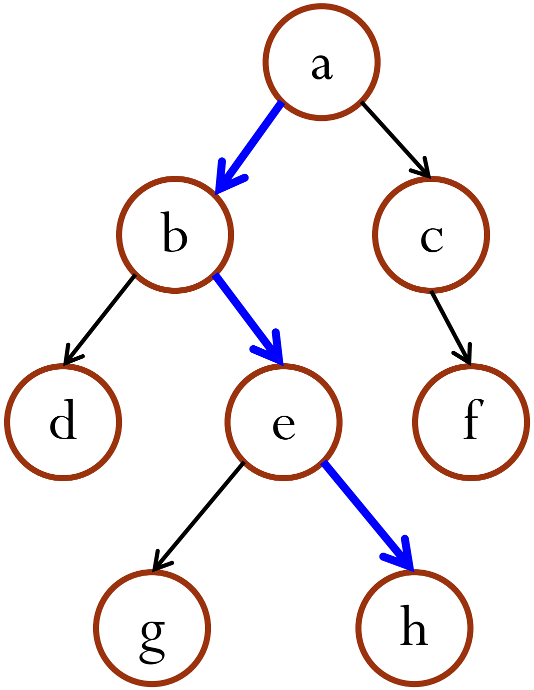
- If there exists a path from a node A to a node B, then A is an **ancestor** of B and B is a **descendant** of A.
  - E.g., a is an ancestor of h and h is a descendant of a.



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.

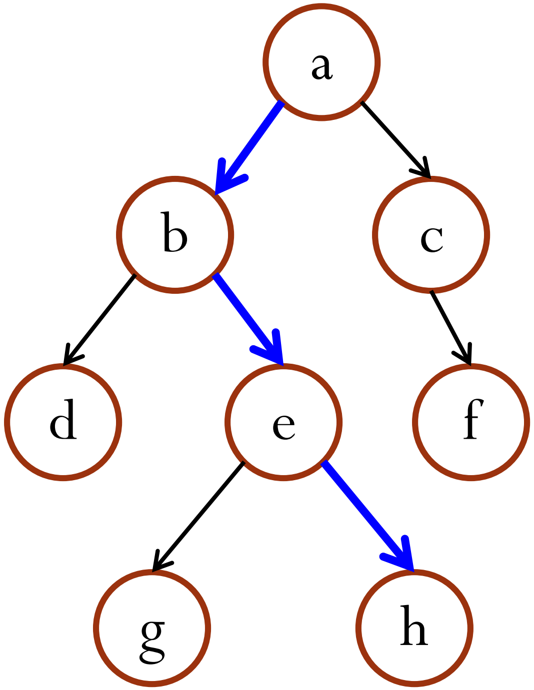
- The **height of a node** is the length of the **longest** path from the node to a **leaf**.
  - E.g., height(b)=2, height(a)=3.
  - All leaves have height zero.



Depth, Level, and Height of a Tree

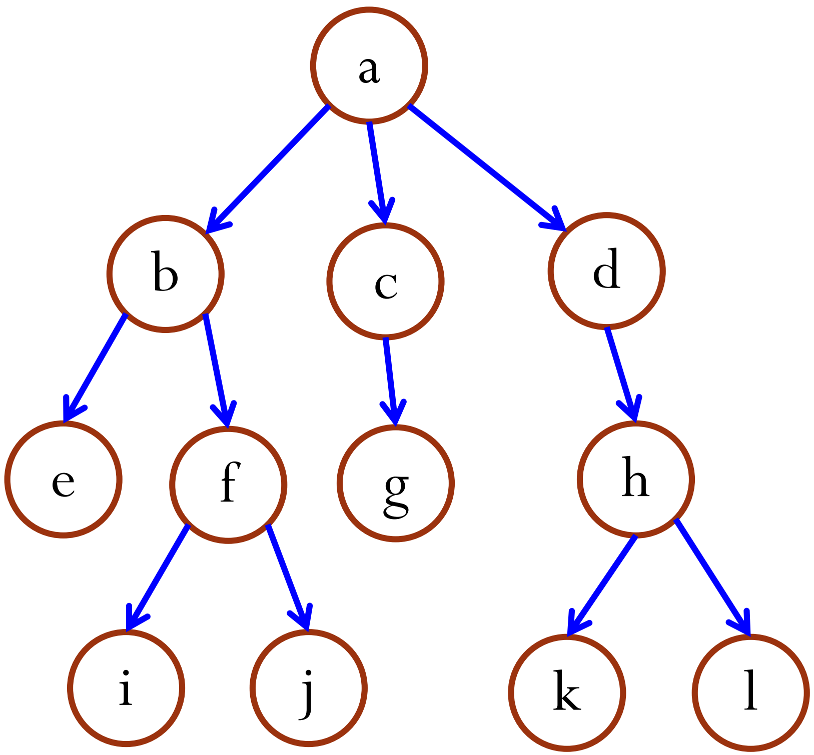
- The **height of a tree** is the height of its root.
  - This is also known as the **depth of a tree**.
  - The depth of the tree on the right is 3.

- 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

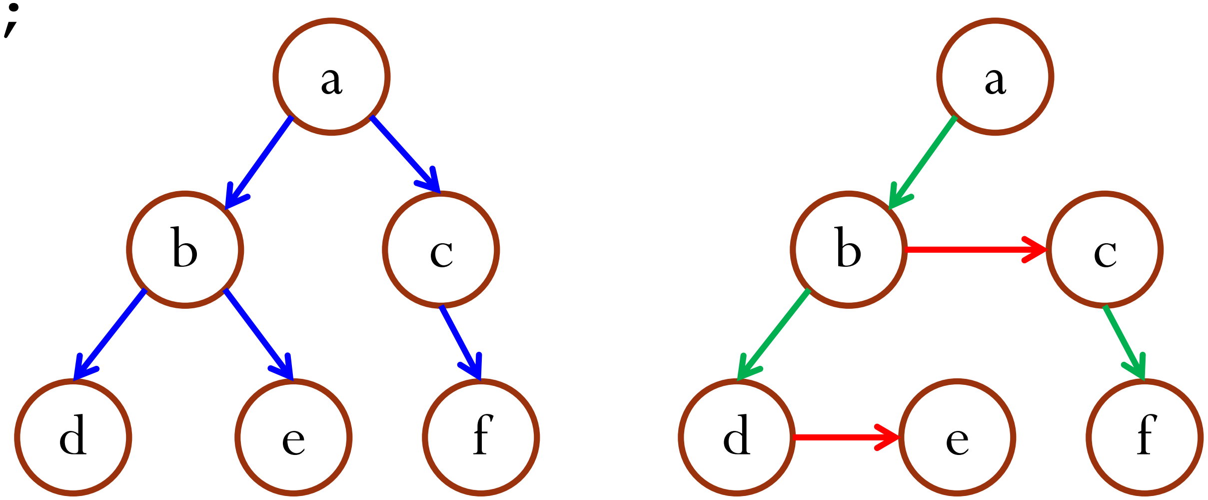
- The **degree of a node** is the number of children of a node.
  - E.g., degree(a) = 3, degree(c) = 1.
- The **degree of a tree** is the **maximum** degree of a node in the tree.
  - The degree of the tree on the right is 3.



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;
};
```



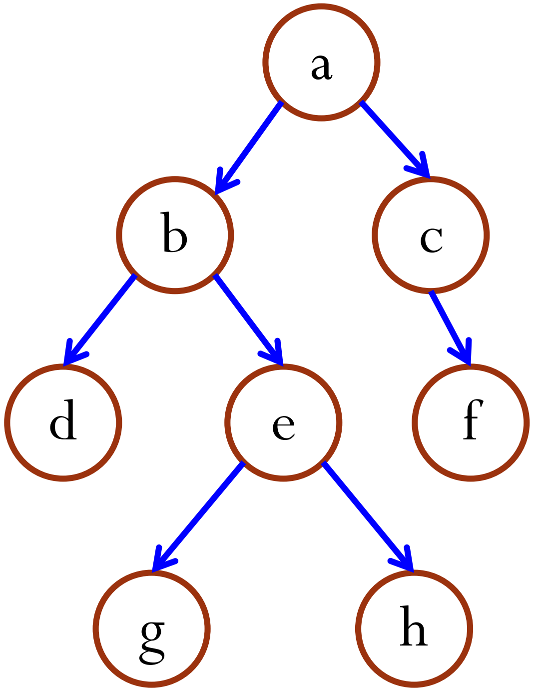
Outline

- Bloom Filter
- Trees
- Binary Trees
- Binary Tree Traversal

Binary Tree

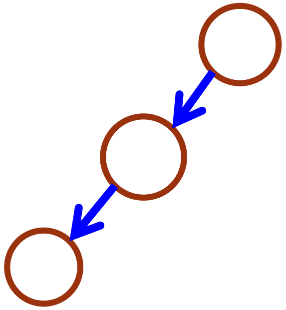
- Every node can only have **at most two** children.

- An empty tree is a special binary tree.



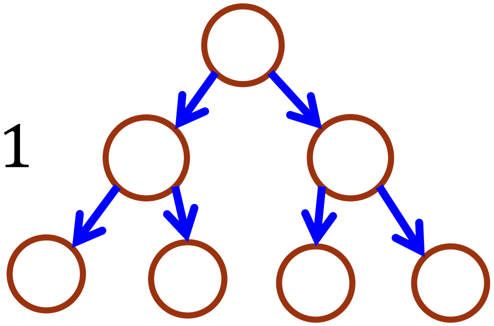
Binary Tree Properties

- 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

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



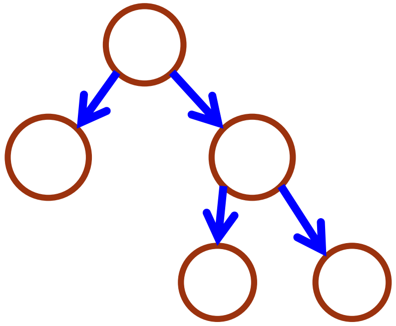
Number Of Nodes and Height

- **Claim** (from the previous slide): 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?
  - $\log_2(n+1)-1 \leq h \leq n-1$

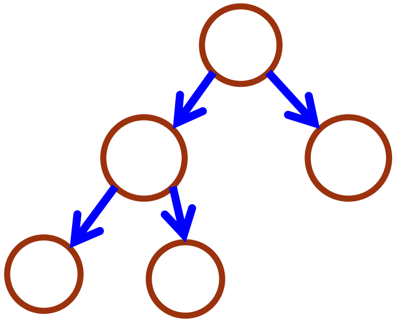
Types of Binary Trees

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

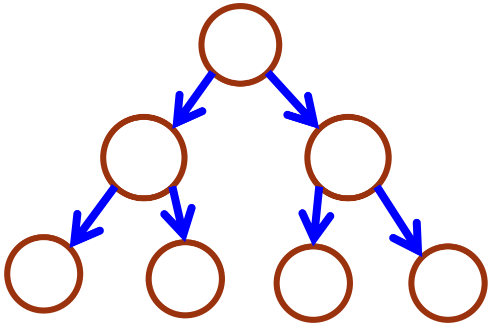


- A binary tree is **complete** if:

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

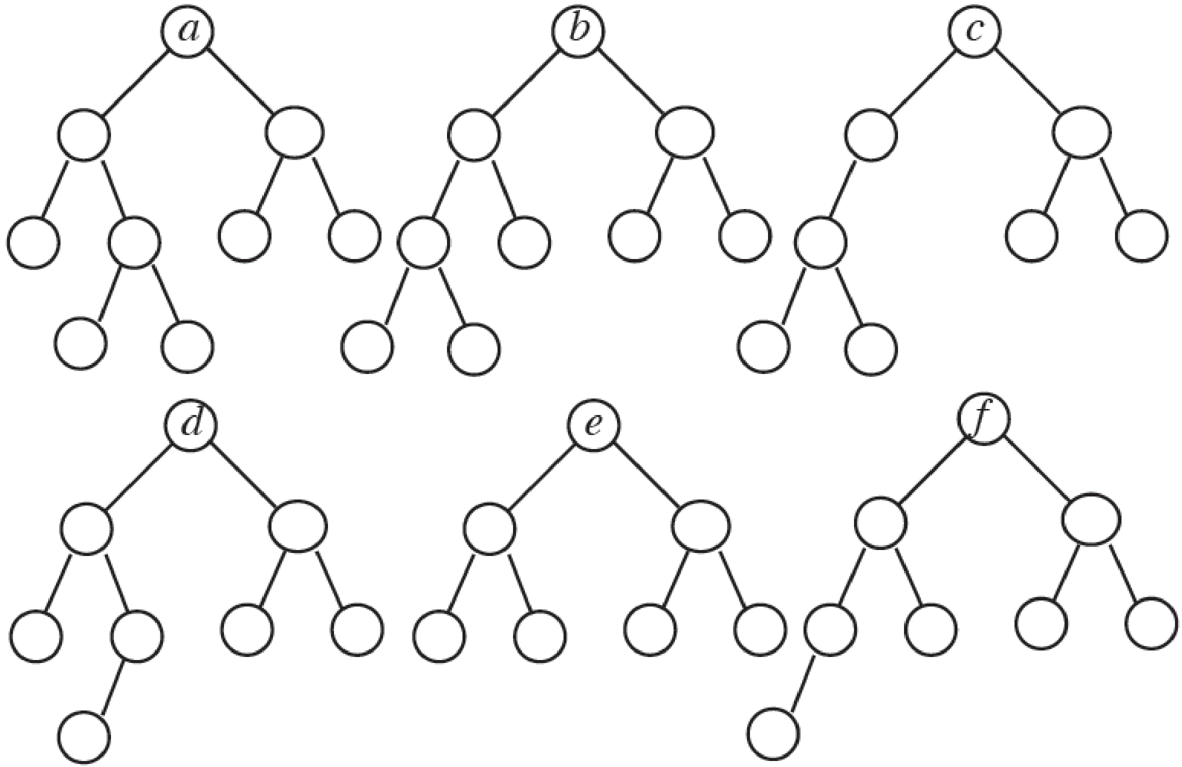


- A binary tree is **perfect** if **every level** is fully populated.



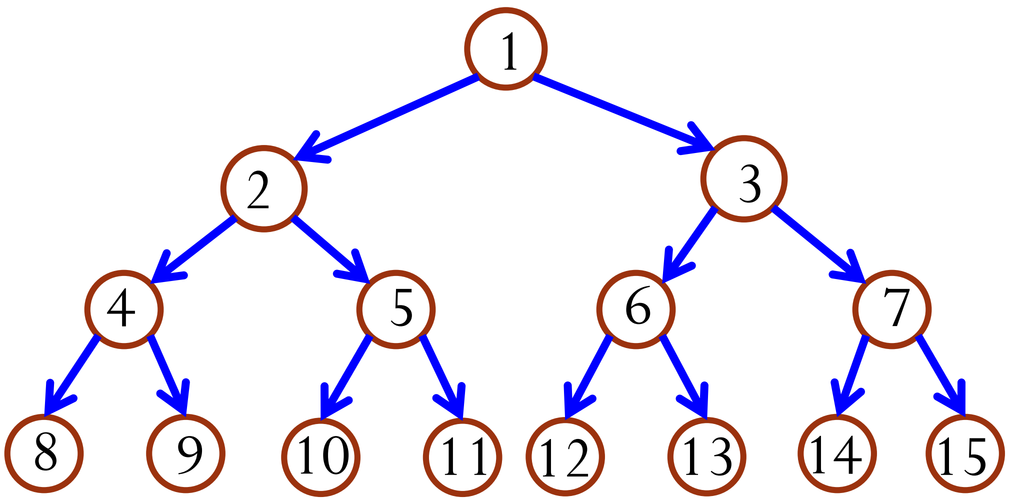
Exercises

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

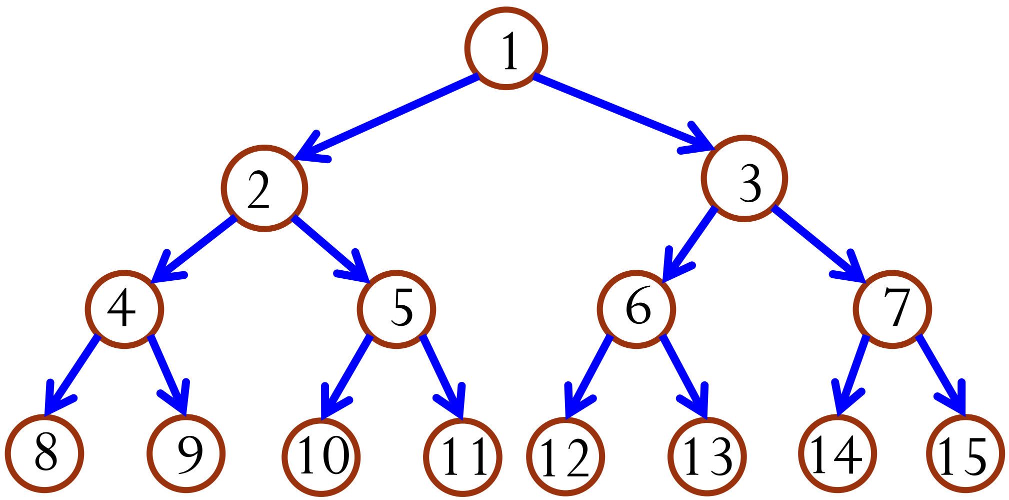


Numbering Nodes In a Perfect Binary Tree

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



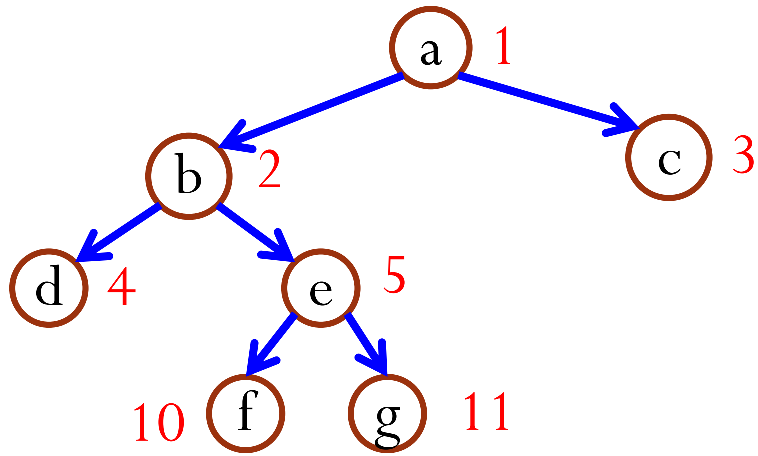
Numbering Nodes In a Perfect Binary Tree



- 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.

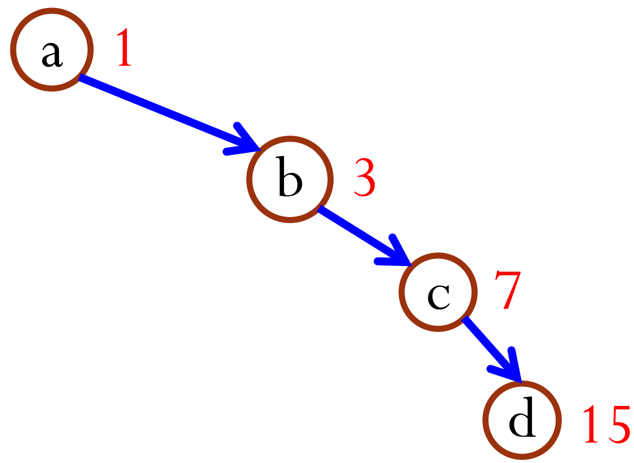


–	a	b	c	d	e	–	–	–	–	f	g
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Representing Binary Tree Using Array

Space Efficiency

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



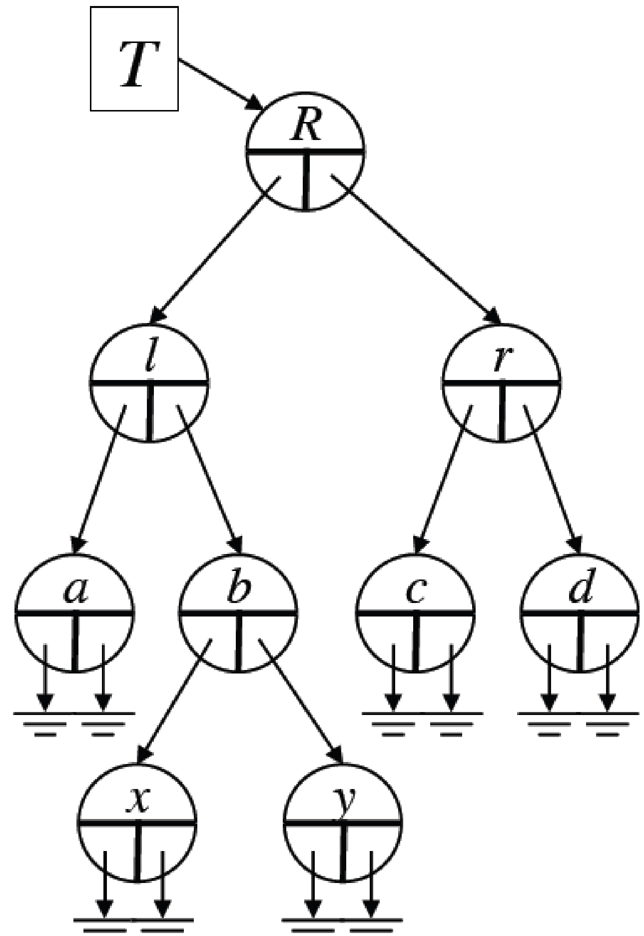
–	a	–	b	–	–	–	c	–	–	–	–	–	–	–	d
[0]	[1]		[3]		[5]		[7]		[9]		[11]		[13]		[15]

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

- Bloom Filter

- 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

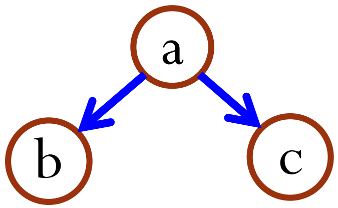
- Depth-first traversal
  - Pre-order
  - Post-order
  - In-order

- Level order traversal

Pre-Order Depth-First Traversal

Procedure

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



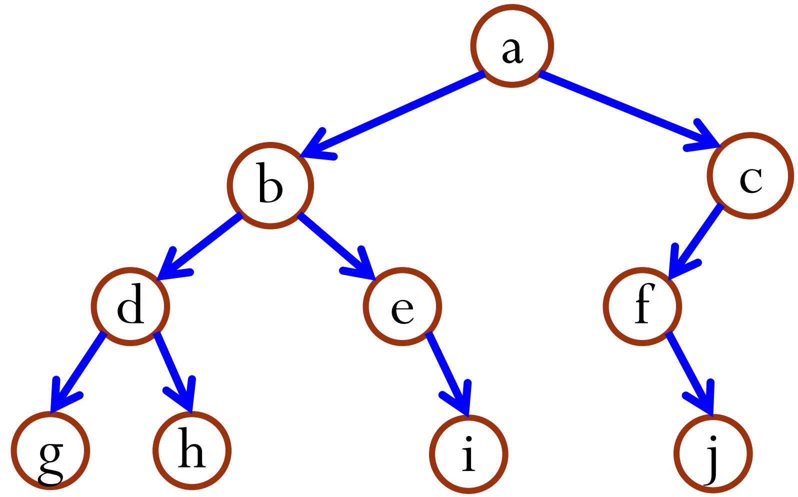
a→b→c

```
void preOrder(node *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

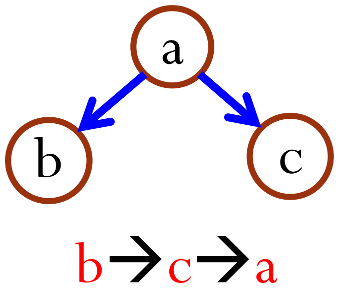


a → b → d → g → h → e → i → c → f → j

Post-Order Depth-First Traversal

Procedure

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

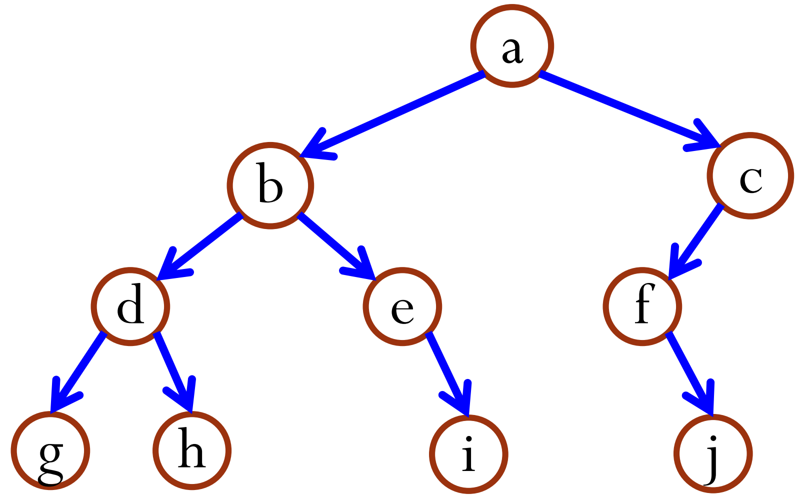



```
void postOrder(node *n) {
   if(!n) return;
   postOrder(n->left);
   postOrder(n->right);
   visit(n);
}
```

Post-Order Depth-First Traversal

Example

g
h
d
i
e
b
j
f
c
a

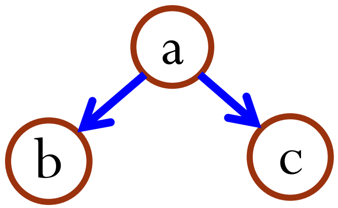


g→h→d→i→e→b→j→f→c→a

In-Order Depth-First Traversal

Procedure

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



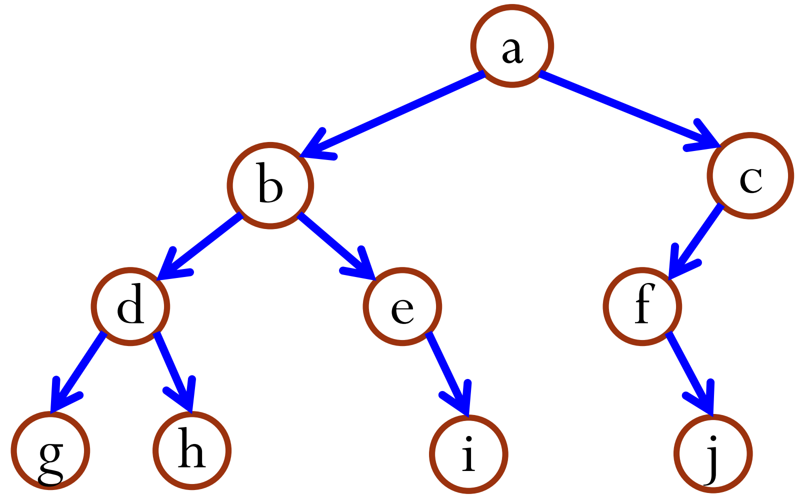
b→a→c

```
void inOrder(node *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



g→d→h→b→e→i→a→f→j→c