Indian Institute of Science
Bangalore, India
भारतीय विज्ञान संस्थान
बंगलौर, भारत

DS221

Data Structures, Algorithms & Data Science Platforms

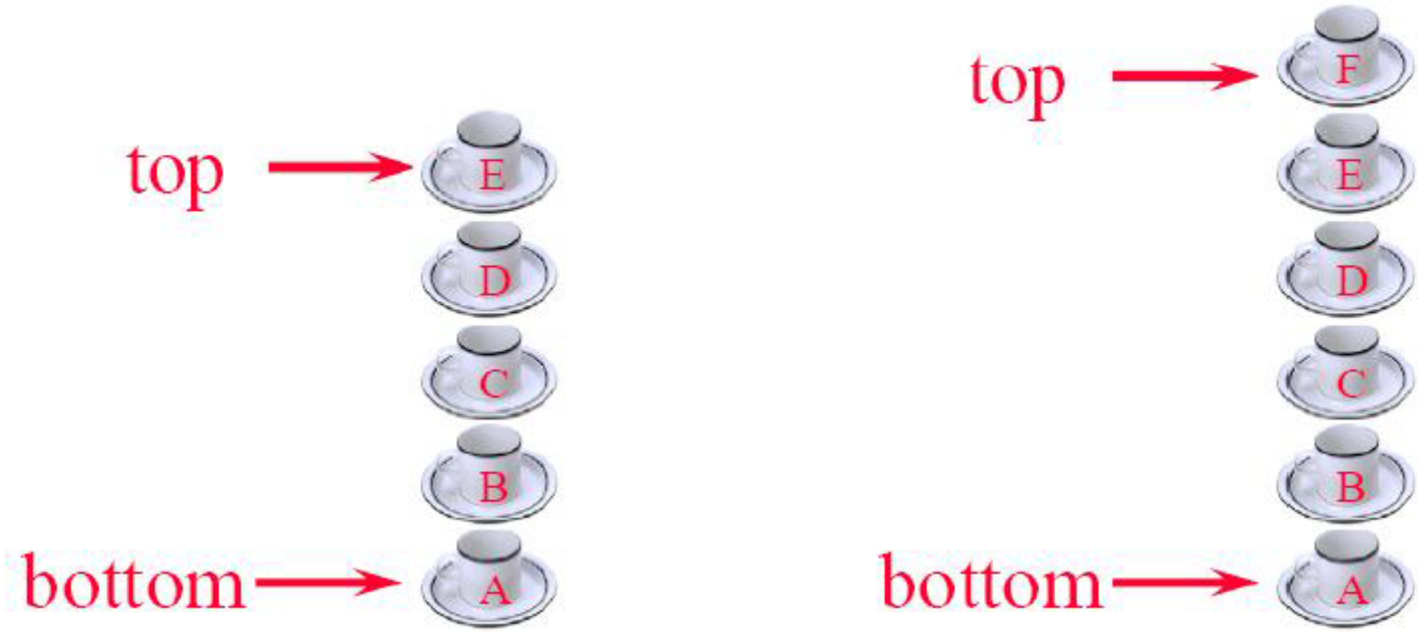
Instructor: Chirag Jain
(slides from Prof. Simmhan)

CDS
The Department of Computational and Data Science

L2: More on Basic Data Structures

Stack, Queue, Trees

Stacks



- Add a cup to the stack.
- Remove a cup from new stack.
- A stack is a **LIFO** list: *Last in, First out*

Stacks

- Container of objects that are inserted and removed according to the LIFO principle
- Objects can be inserted at any time, but only the last object can be removed.
  - Inserting :“pushing”
  - Removing : “Popping”

Stacks (definition)

- **New**() creates a new stack
- **Push**(item) inserts the *item* onto top of stack
- item **Pop**() removes and returns the top *item* of stack
- item **Top**() returns (but retains) the top *item* of stack (sometimes called **Peek**())
- int **Size**() returns number of objects in stack
- Invariants
  - `S.Push(v);S.Top();` *Returns the value* `v`
  - `S.Push(v);S.Pop();` *Same stack state as before push*

Parenthesis Matching

- Problem: Match the left and right parentheses in a character string

- (a*(b+c)+d)
  - Left parentheses: positions 0 and 3
  - Right parentheses: positions 7 and 10
  - Left at position 0 matches with right at position 10

- (a+b))*((c+d)
  - (0,4) match
  - (8,12) match
  - Right parenthesis at 5 has no matching left parenthesis
  - Left parenthesis at 7 has no matching right parenthesis

Parenthesis Matching

0 1 2 3 4 5 6 …

(((a+b)*c+d-e)/(f+g)-(h+j)*(k-1))/(m-n)

- Output pairs (u,v) such that the left parenthesis at position u is matched with the right parenthesis at v.
- (2,6) (1,13) (15,19) (21,25) (27,31) (0,32) (34,38)

- **How do we implement this using a stack?**

Parenthesis Matching

0 1 2 3 4 5 6 …

(((a+b)*c+d-e)/(f+g)-(h+j)*(k-1))/(m-n)

- Output pairs (u,v) such that the left parenthesis at position u is matched with the right parenthesis at v.
- (2,6) (1,13) (15,19) (21,25) (27,31) (0,32) (34,38)

- **How do we implement this using a stack?**

1. Scan expression from left to right
2. When a left parenthesis is encountered, add its position to the stack
3. When a right parenthesis is encountered, remove matching position from the stack

Example

- (a*(b+c)+d)

0	1	2	3	4	5	6	7	8	9	10
(	a	*	(	b	+	c	)	+	d	)

0	3	0	
	0		
		3,7	0,10

Queue ADT

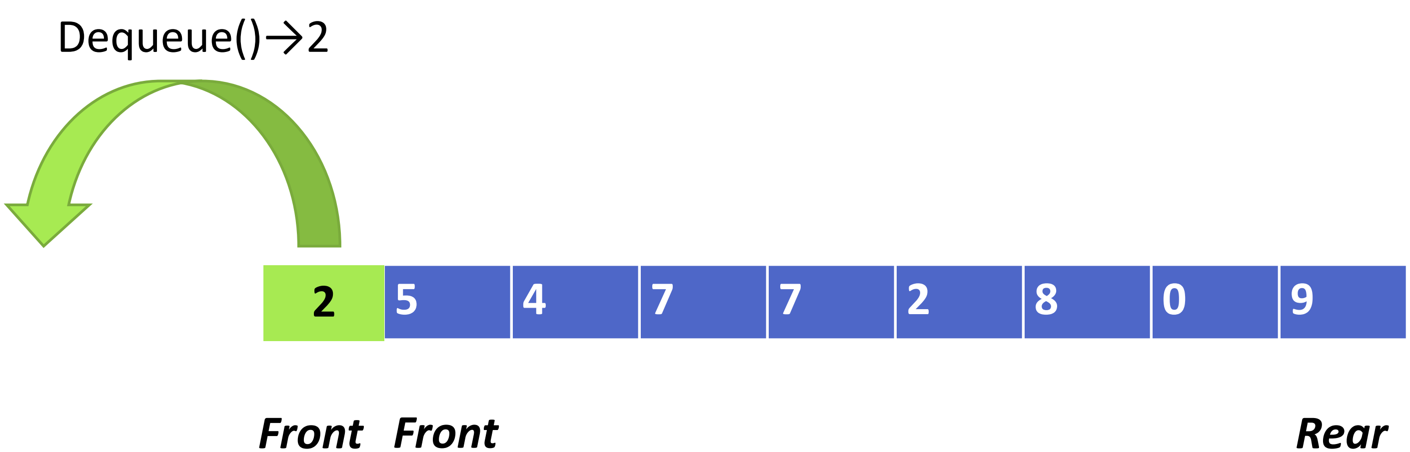
- **FIFO** Principle: *First in, First Out*
- Elements **inserted only at rear** (enqueued) end and **removed from front** (dequeued)
  - Also called "Head" and "Tail"

2	5	4	7	7	2	8	0	9

Front ***Rear***

Queue ADT

- **FIFO** Principle: *First in, First Out*
- Elements **inserted only at rear** (enqueued) end and **removed from front** (dequeued)
  - Also called "Head" and "Tail"



Queue ADT

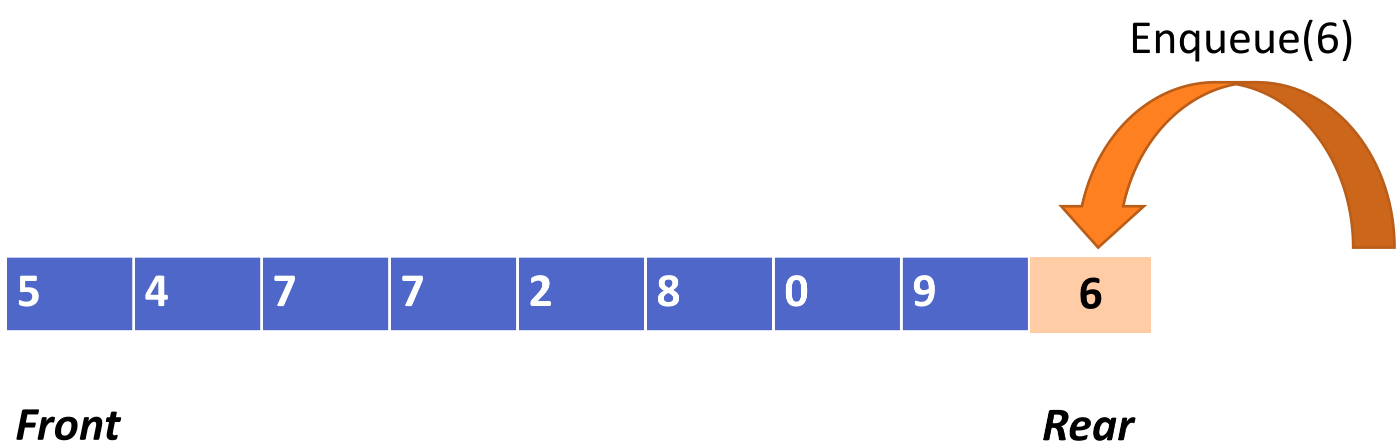
- **FIFO** Principle: *First in, First Out*
- Elements **inserted only at rear** (enqueued) end and **removed from front** (dequeued)
  - Also called "Head" and "Tail"

5	4	7	7	2	8	0	9

Front ***Rear***

Queue ADT

- **FIFO** Principle: *First in, First Out*
- Elements **inserted only at rear** (enqueued) end and **removed from front** (dequeued)
  - Also called "Head" and "Tail"



Queue -Methods

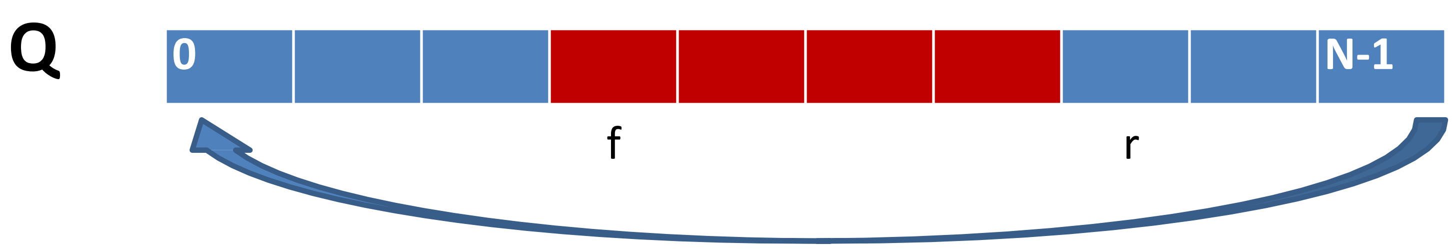
- queue **New**() – Creates and returns an empty queue
- **Enqueue**(item v) – Inserts object **v** at the *rear* of the queue
- item **Dequeue**() – Removes the object from *front* of the queue. Error occurs if the queue is empty
- item **Front**() – Returns, but does not remove the front element. An error occurs if the queue is empty (also called **Peek**())
- int **Size**() – number of items in queue

Queue –Invariants

- `Q=New();Q.Enqueue(v);Q.front();` *returns* `v`
- `Q=New();Q.Enqueue(v);Q.Dequeue();` *has same queue state as* `New()`
- `Q.Enqueue(w);Q.Enqueue(v);Q.front();` *returns the same value as* `Q.Enqueue(w);Q.front();`
- `Q.Enqueue(w);Q.Enqueue(v);Q.Dequeue();` *has same queue state as* `Q.Enqueue(w);Q.Dequeue();Q.Enqueue(v);`

Array Implementation of Queue

- Using array in *circular* fashion
  - **Wraparound** using mapping function (recollect from List ADT discussion)
- A max size N is specified
- Q consists of an N element array and 2 integer variables having array index:
  - **f**: index of the front element (head, for dequeue)
  - **r**: index of the element after the rear one (tail, for enqueue)



Array Implementation of Queue

Q

- What does `f=r` mean ?

- Resolve Ambiguity:
  - We will never add n^{th} element to Queue (declare full if the size of queue is N-1) .

Pseudo Code

```
int front()
  If size()==0 then Return QueueEmptyException
  Else Return Q[f]
int Dequeue()
  If isEmpty() then Return QueueEmptyException
  v = Q[f]
  Q[f] = null
  f = (f+1) mod N
  Return v
Enqueue(v)
  If size()==N-1 then Return QueueFullException
  Q[r] = v
  r = (r+1) mod N
int size()
   Return (N-f+r) mod N
```

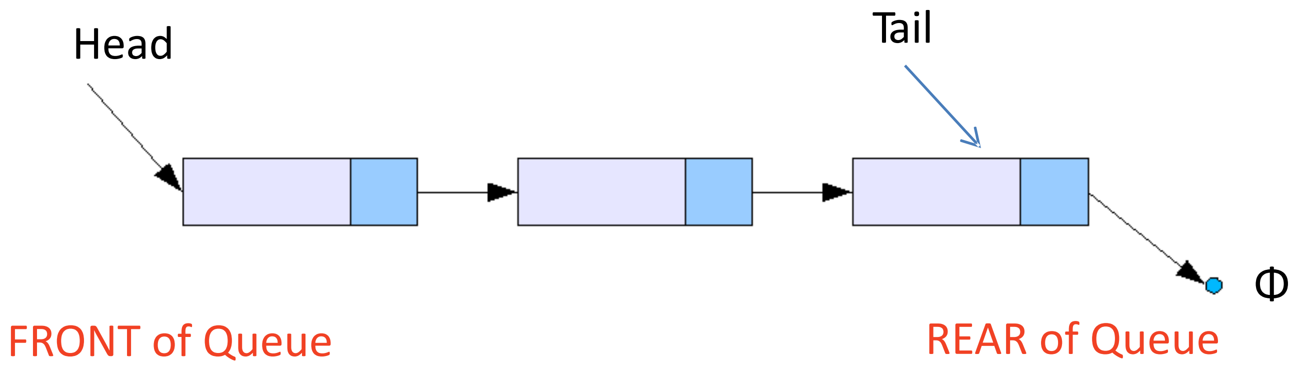
Compute Complexity? Storage Complexity?

Queue using a Linked List

- Problem with array: Requires the number of elements (capacity) *a priori*.

Queue using a Linked List

Nodes (data, pointer) connected in a chain by links

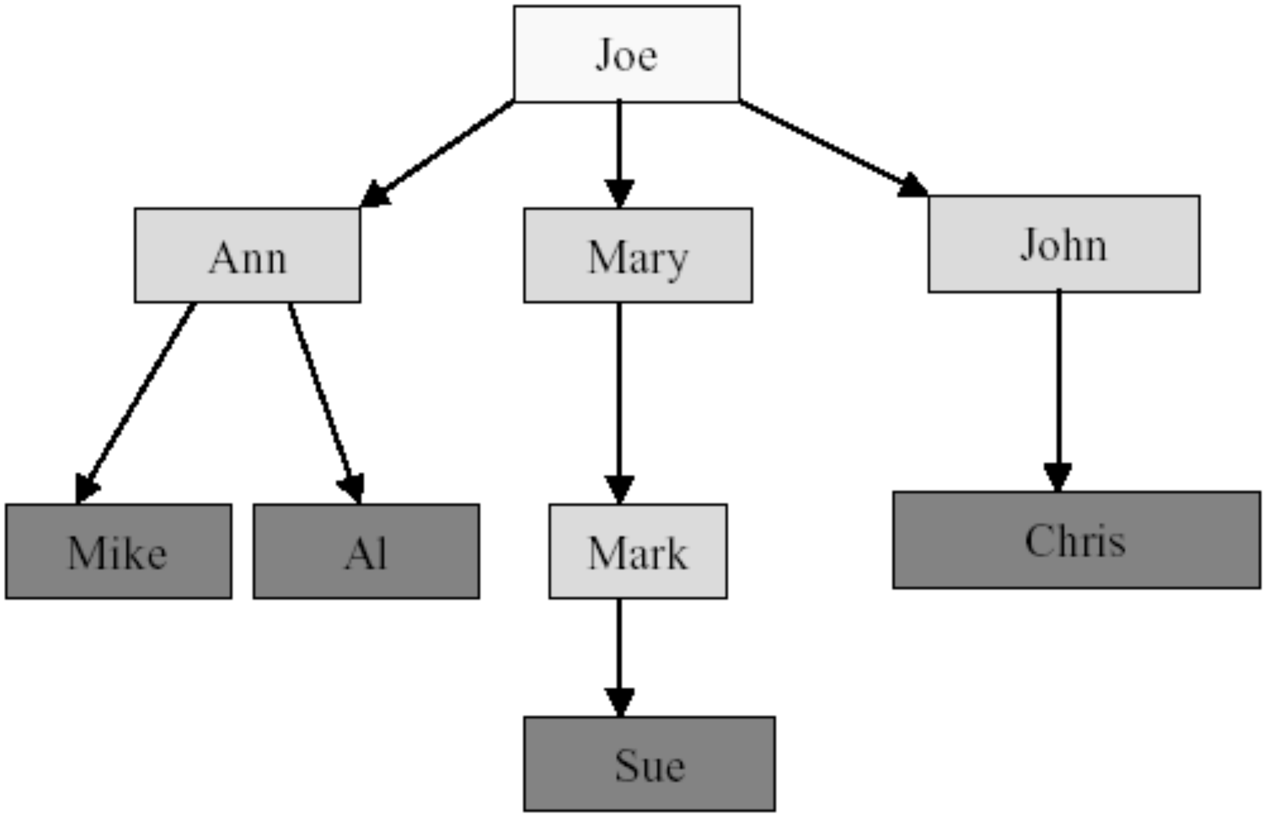


- Maintain two pointers, to head and tail of linked list.
- The head of the list is FRONT of the queue, the tail of the list is REAR of the queue.
- Why not the opposite?

Linear Lists vs. Trees

- Linear lists are useful for serially ordered data
  - $(e_1,e_2,e_3,\ldots,e_n)$
  - Days of week
  - Months in a year
  - Students in a class
- Trees are useful for hierarchically ordered data
  - Joe's descendants
  - Corporate structure
  - Government Subdivisions
  - Software structure

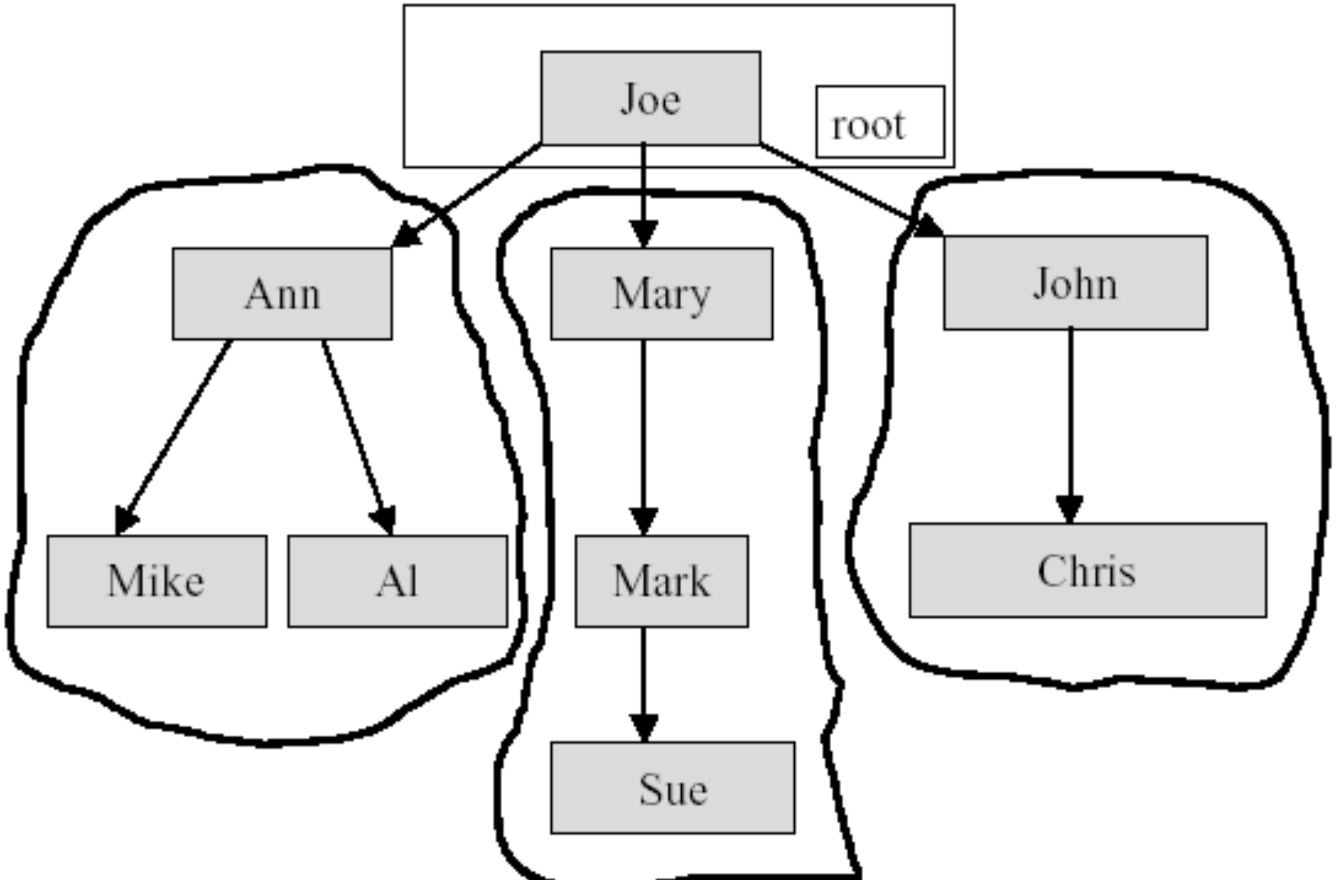
Joe's Descendants



Definition of Tree

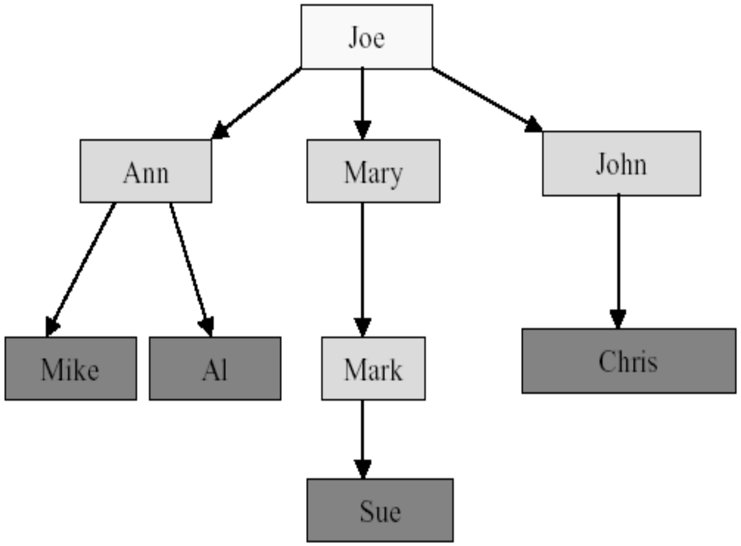
- A tree *t* is a finite non-empty set of elements
- One of these elements is called the root
- The remaining elements, if any, are partitioned into trees, which are called the subtrees of *t*.

Subtrees



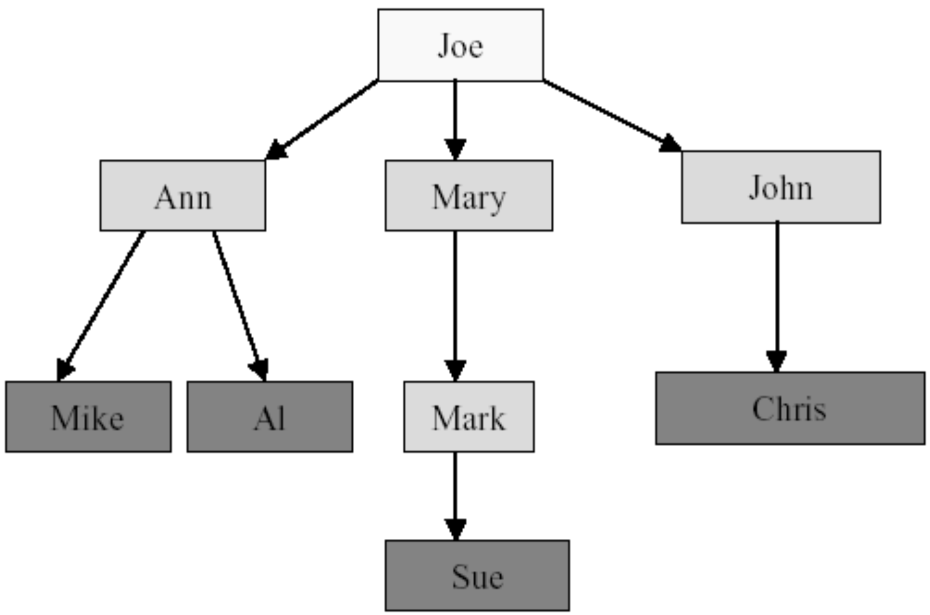
Tree Terminology

- The element at the top of the hierarchy is the **root**.
- Elements next in the hierarchy are the **children** of the root.
- Elements next in the hierarchy are the **grandchildren** of the roo and so on.
- Elements at the lowest level of the hierarchy are the **leaves**.



Tree Terminology

- Leaves, Parent, Grandparent, Siblings, Ancestors, Descendents



Leaves = {Mike,AI,Sue,Chris}

Parent(Mary) = Joe

Grandparent(Sue) = Mary

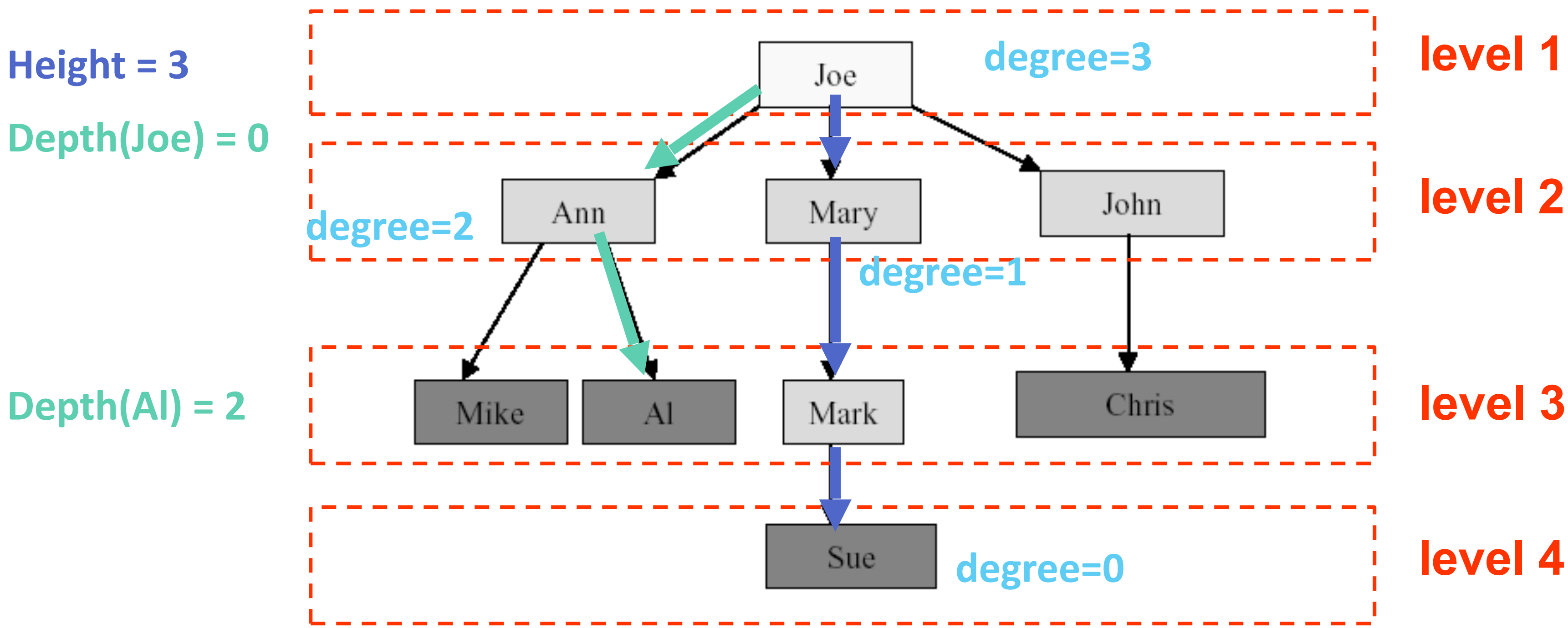
Siblings(Mary) = {Ann,John}

Ancestors(Mike) = {Ann,Joe}

Descendents(Mary)={Mark,Sue}

Tree Terminology

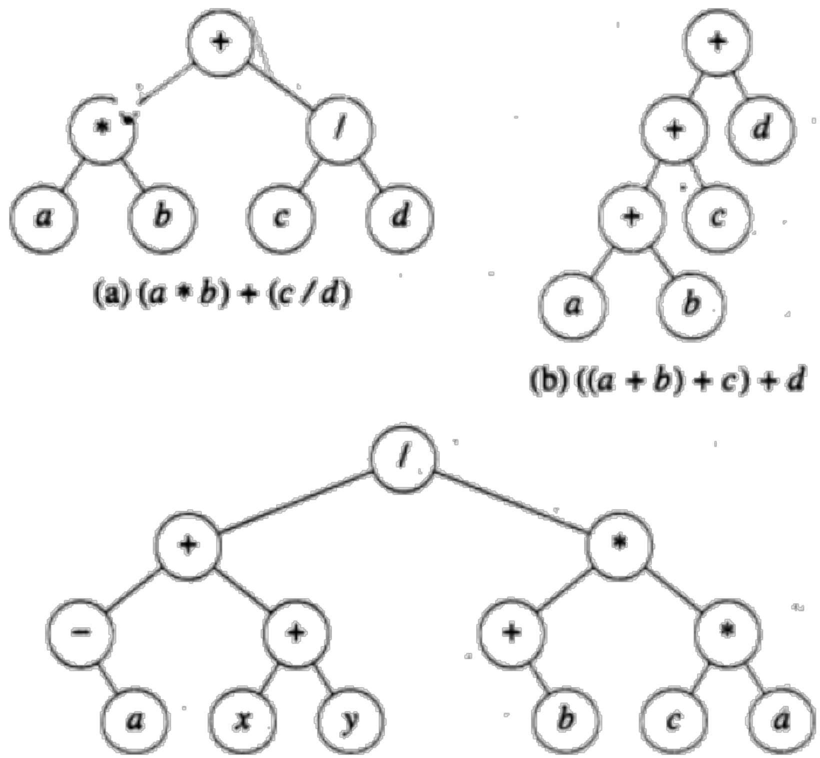
- **Depth** of Node = No. of edges from the root to that node
- **Height** of Tree = No. of edges from root to farthest leaf
- Number of **Levels** of a Tree = Height + 1
- Node **degree** is the number of children it has



Binary Tree

- A finite (possibly empty) collection of elements
- A non-empty binary tree has a root element and the remaining elements (if any) are partitioned into two binary trees
- They are called the left and right sub-trees of the binary tree

Binary Tree for Expressions



(a) $(a * b) + (c / d)$

(b) $((a + b) + c) + d$

(c) $((-a) + (x + y)) / ((+b) * (c * a))$

Figure 11.5 Expression trees

Binary Tree Properties

1. The drawing of every binary tree with n elements, $n > 0$, has exactly n-1 edges.
   – Each node has exactly 1 parent (except root)

2. A binary tree of height h, $h >= 0$, has at least $h + 1$ and at most $2^{h+1} - 1$ elements in it.
   - h+1 levels; at least 1 element at each level → #elements = h+1
   - At most 2^{i-1} elements at i-th level → $\Sigma\, 2^{i-1} = 2^{h+1} - 1$

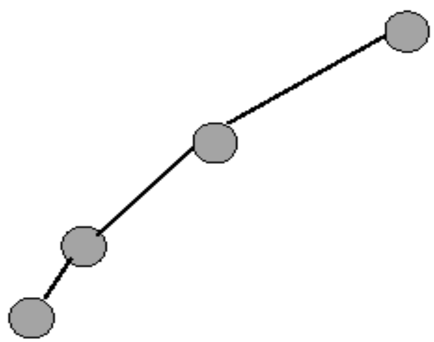
$$a+ar^1+ar^2+\ldots+ar^n = a(r^{n+1}-1)/(r-1)$$

Note: Some tree definitions differ between computer science & discrete math

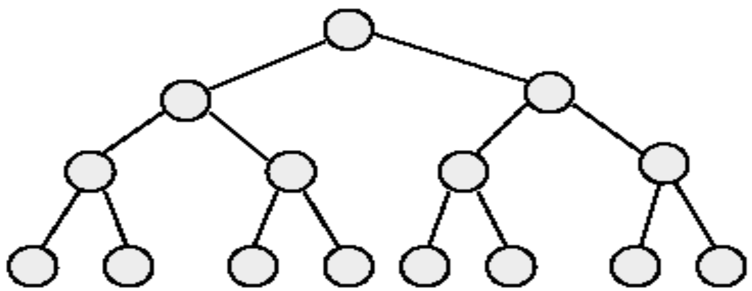
Binary Tree Properties

3. The height of a binary tree that contains *n* elements, *n* >= 0, is at least $\lfloor \log_2 n \rfloor$ and at most *n-1*.

   – At least one element at each level → h_{max} = #elements - 1
   – From prev: h_{min} = ceil(log(n+1))



minimum number of elements

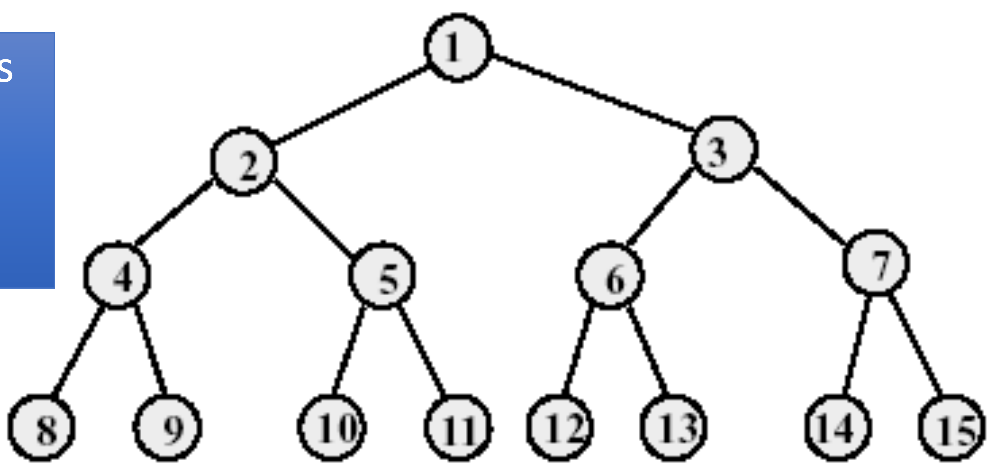


maximum number of elements

Full Binary Tree

- A full binary tree of height h has exactly $2^{h+1}-1$ nodes
- Numbering the nodes in a full binary tree
  - Number the nodes 1 through $2^{h+1}-1$
  - Number by levels from top to bottom
  - Within a level, number from left to right

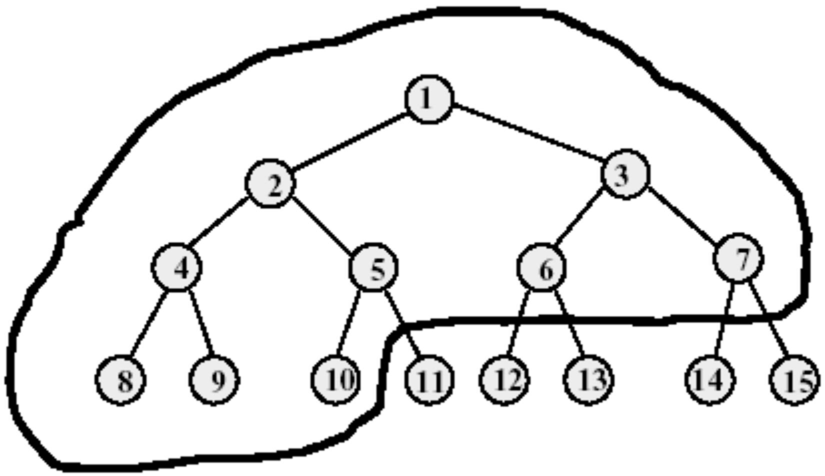
Note: Some definitions of full, complete trees are NOT consistently used everywhere



Complete Binary Tree with N Nodes

- Start with a full binary tree that has at least *n* nodes
- Number the nodes as described earlier
- The binary tree defined by the nodes numbered 1 through n is the n-node complete binary tree
- A full binary tree is a special case of a complete binary tree

Complete Binary Tree



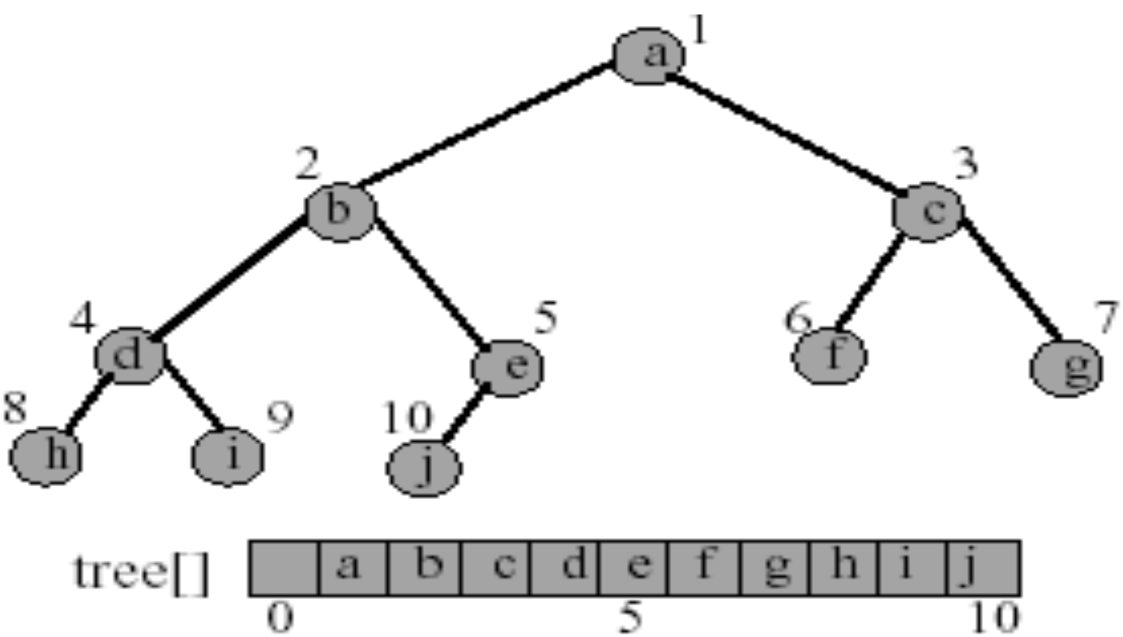
- Complete binary tree with 10 nodes.
- Same node number properties (as in full binary tree) also hold here.

Binary Tree Representation

- Array representation
- Linked representation

Array Representation

- The binary tree is represented in an array by storing each element at the array position corresponding to the number assigned to it.



Incomplete Binary Trees

Complete binary tree with some missing elements

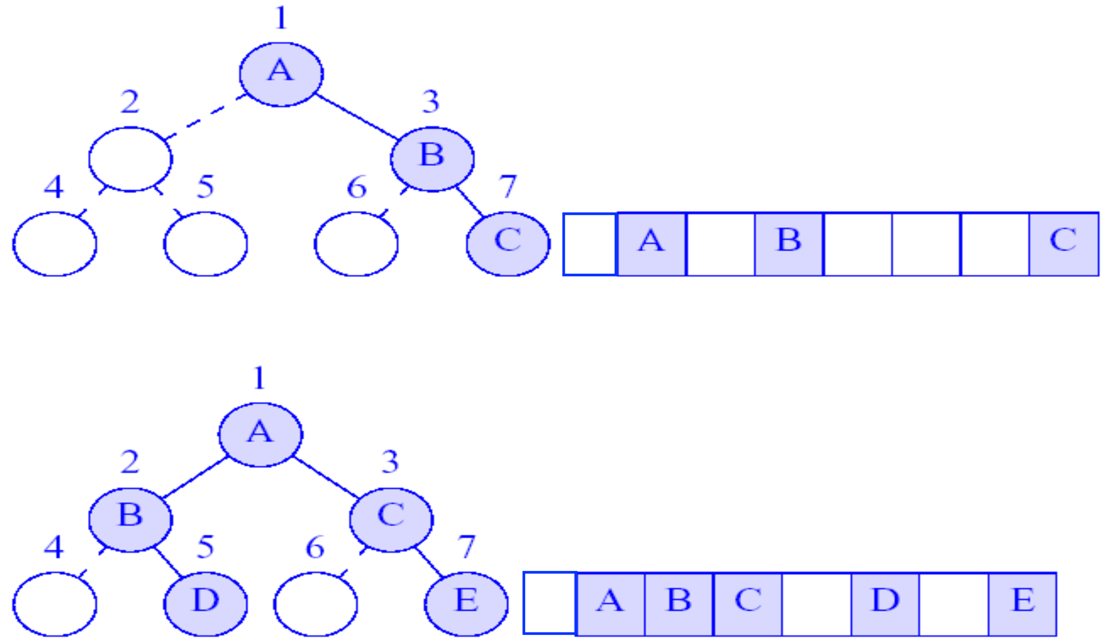
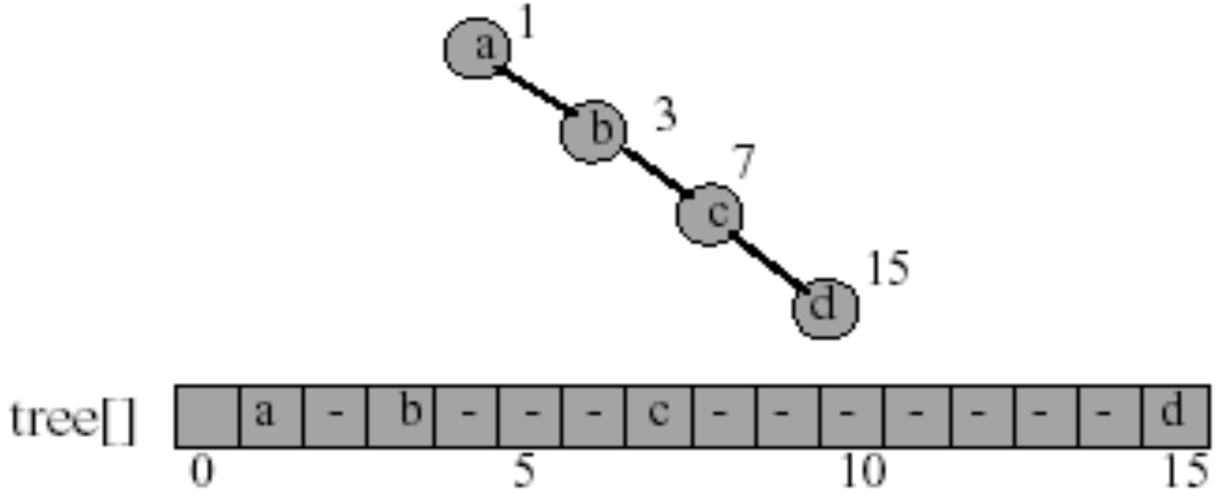


Figure 8.8 Incomplete binary trees

What tree type would lead to least space wastage?
What tree type would lead to most space wastage?

Right-Skewed Binary Tree

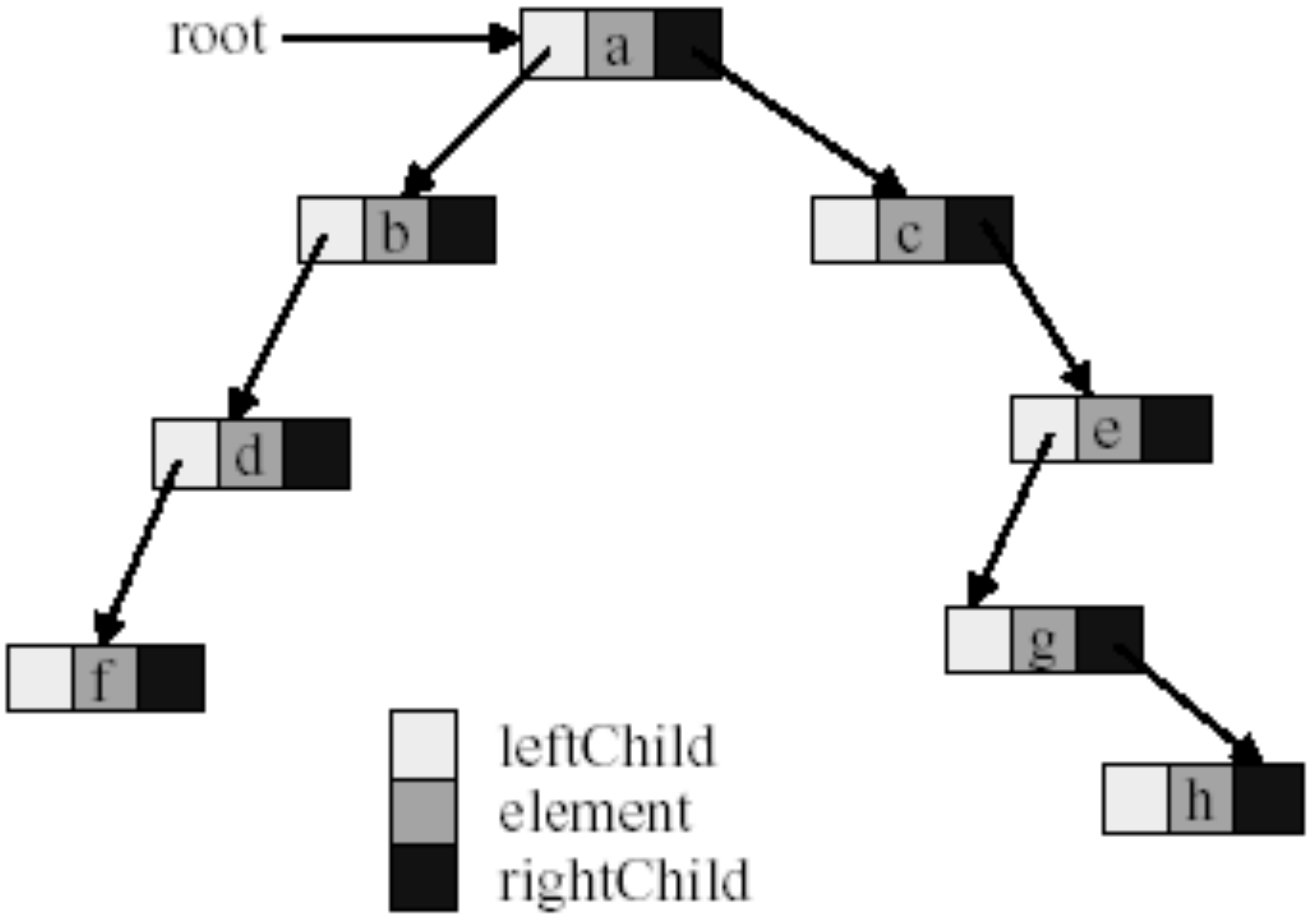


- An *n* node binary tree needs an array whose length is between ***n+1*** and $\boldsymbol{2^n}$.
- Right-skewed binary tree wastes the most space
- What about left-skewed binary tree?
  - *Equally bad,* $O(2^n)$ *space*

Linked Representation

- The most popular way to present a binary tree
- Each element is represented by a node that has two link fields (leftChild and rightChild) plus an item field
- Each binary tree node is represented as an object whose data type is BinTreeNode
- The space required by an n node binary tree is `n*sizeof(BinTreeNode)`

Linked Representation



Node Class For Linked Binary Tree

```
class BinTreeNode {
   int item;
   BinTreeNode *left, *right;

   BinTreeNode() {
      left = right = NULL;
   }
}
```

Binary Tree Traversal

- Many binary tree operations are done by performing a **traversal** of the binary tree
- In a traversal, each element of the binary tree is **visited exactly once**
- During the visit of an element, all **actions** *(make a copy, display, evaluate the operator, etc.)* with respect to this element are taken

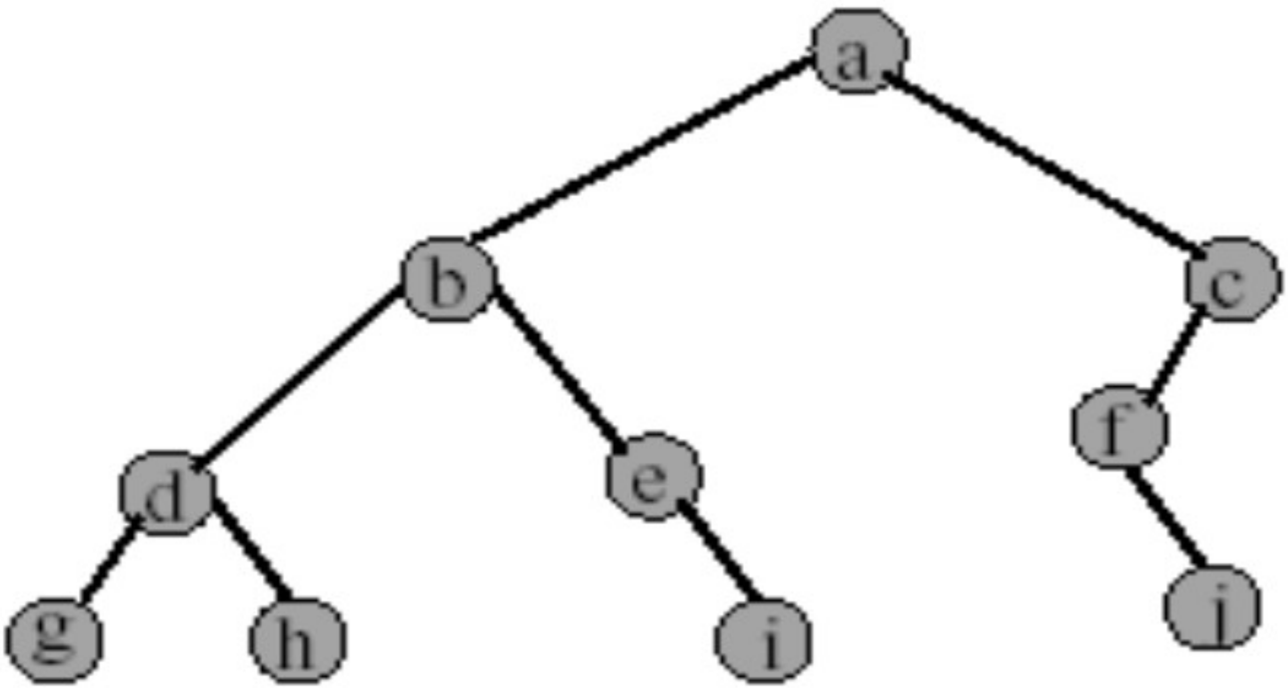
Binary Tree Traversal Methods

- Preorder
  - The **root** of the subtree is processed **first** before going into the **left then right subtree** (root, left, right)
- Inorder
  - After the complete processing of **the left subtree first** the **root** is processed followed by the processing of the complete **right subtree** (left, root, right)
- Postorder
  - The **left and right subtree** are completely processed, before the **root** is processed (left, right, root)
- Level order
  - The tree is processed one level at a time
  - First all nodes in level *i* are processed from left to right
  - Then first node of level *i+1* is visited, and rest of level *i+1* processed

Preorder Traversal

```
void preOrder(BinTreeNode *t) {
  if (t != NULL) {
    visit(t);              // Visit root 1st
    preOrder(t->left);     // Left
Subtree
    preOrder(t->right);    // Right
Subtree
  }
}
```

Preorder Example
(visit action = print)

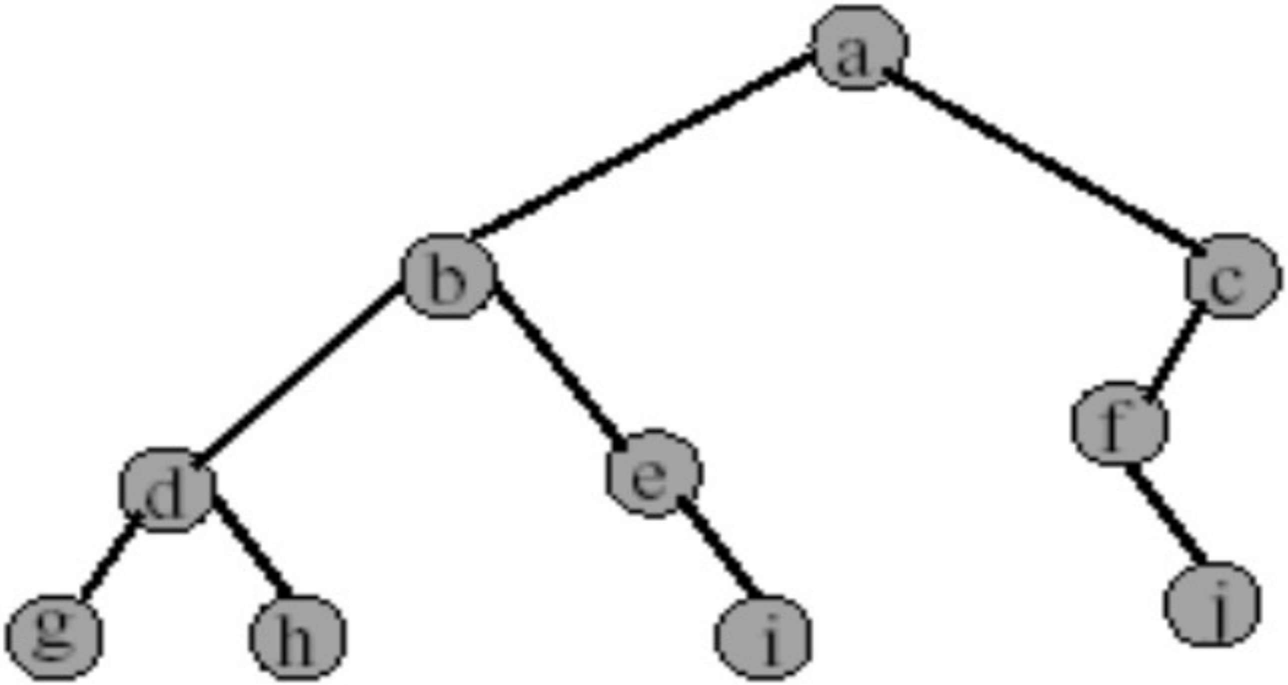


a b d g h e i c f j

Inorder Traversal

```
void inOrder(BinTreeNode *t) {
   if (t != NULL) {
      inOrder(t->left);// Left Subtree 1st
      visit(t);             // Visit root
      inOrder(t->right);    // Right
Subtree last
   }
}
```

Inorder example

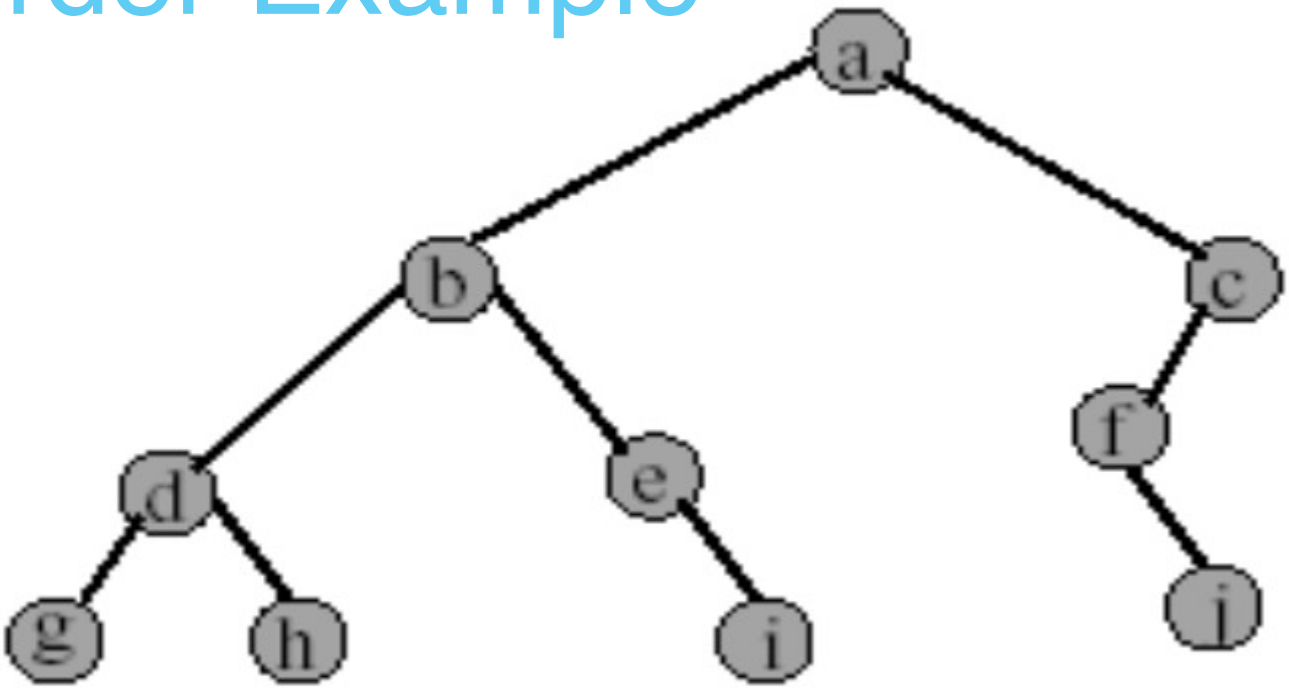


g d h b e i a f j c

Postorder Traversal

```
void postOrder(BinTreeNode *t) {
   if (t != NULL) {
      postOrder(t->left);    // Left Subtree 1st
      postOrder(t->right);// Right Subtree
      visit(t);              // Visit root last
   }
}
```

Postorder Example

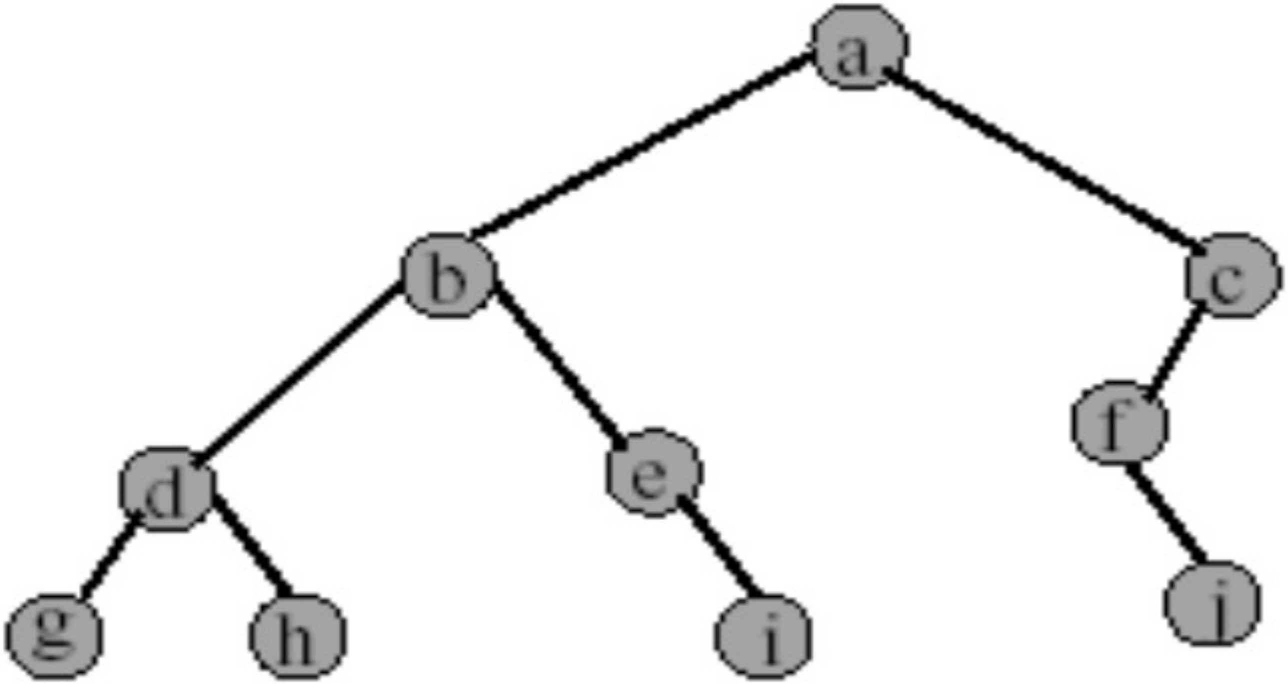


g h d i e b j f c a

Level Order Traversal

```
void levelOrder(BinTreeNode *t){
  Queue<BinTreeNode*> q;
  while (t != NULL) {
    visit(t); // visit t
    // push children to queue
     if (t->left) q.enqueue(t->left);
     if (t->right) q.enqueue(t->right);
    t = q.dequeue();    // next node to
visit
  }
}
```

Level Order Example



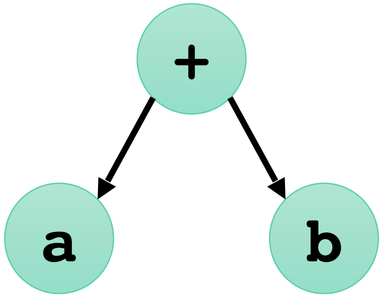
- Add and delete nodes from a queue
- Output: a b c d e f g h i j

Space and Time Complexity

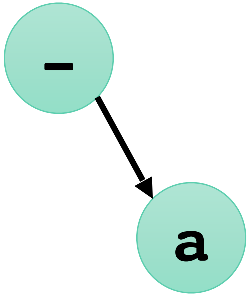
- The worst-case space complexity of each of the four traversal algorithms is **O(n)**
  - Find out the best-case space complexity for each traversal
- The worst-case time complexity of each of the four traversal algorithm is **O(n)**
  - Each node visited only once

Math Expression Evaluation: Binary Tree Form

- a + b

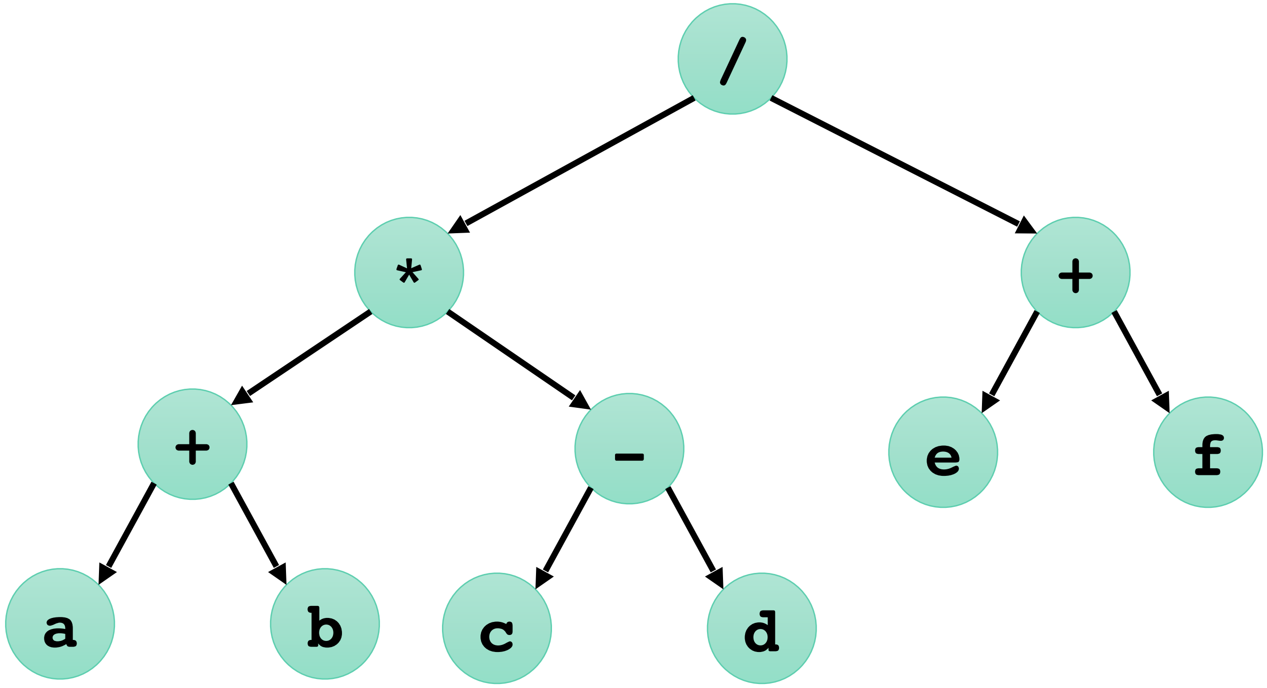


- - a



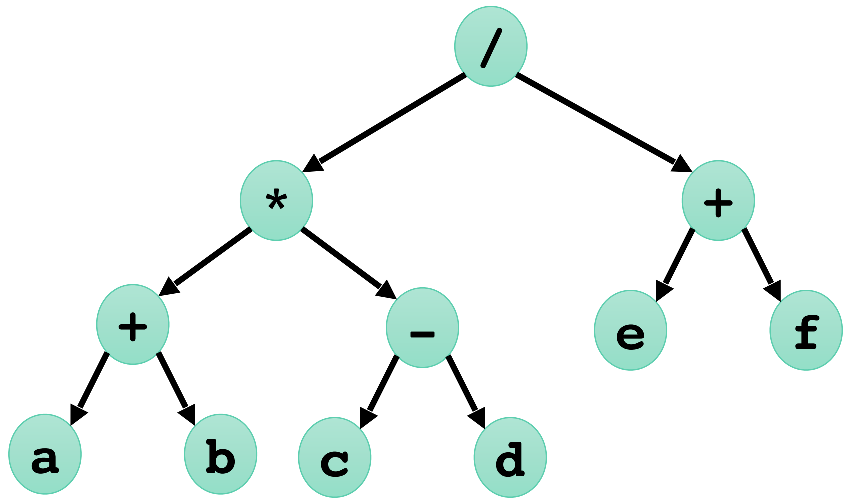
Binary Tree Form

- (((a + b) * (c – d)) / (e + f))



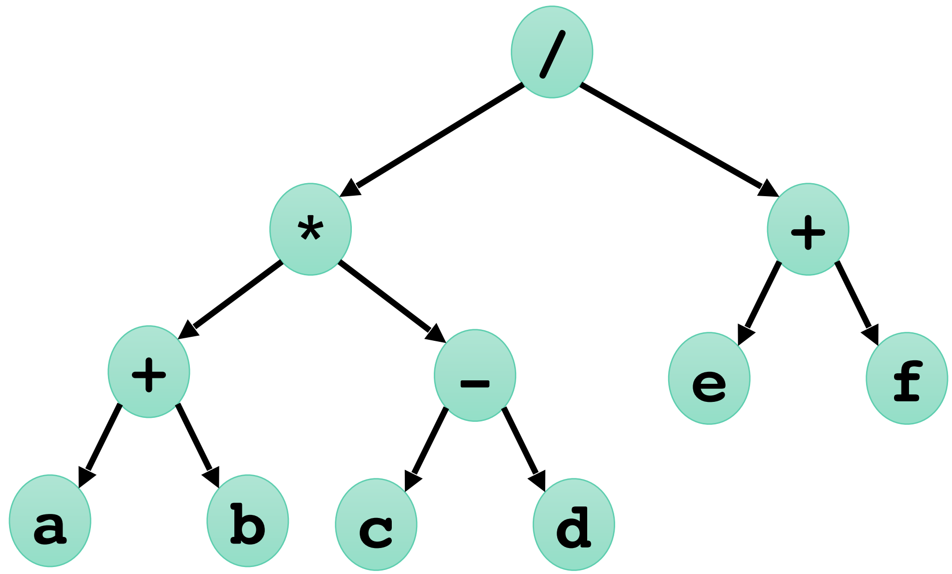
Merits Of Binary Tree Form

- Left and right operands are easy to visualise
- Code optimisation algorithms work with the binary tree form of an expression
- Simple recursive evaluation of expression



Work it out!

Postorder of Expression Tree

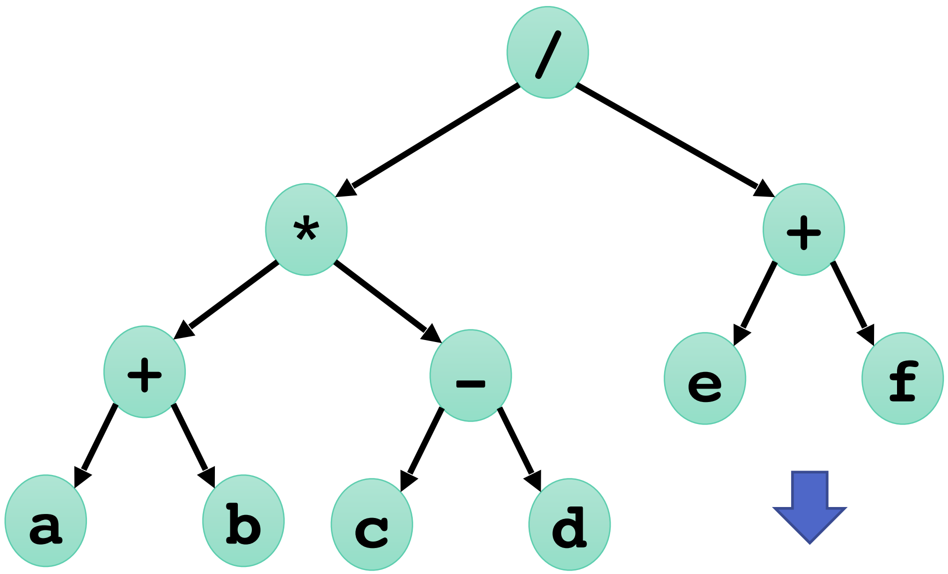


a b + c d - * e f + /

Gives postfix form of expression.

How can you evaluate this postfix expression using a stack? Try out yourself.

Inorder of Expression Tree



a + b * c – d /e + f

- Gives infix form of expression, which is how we normally write math expressions.
  - What about parentheses?
  - Fully parenthesized output of the above tree?

Tasks

- Self study (Sahni Textbook)
  - Chapter 8, Stacks
  - Chapter 9, Queues from textbook
  - Chapter 11.0-11.6, Trees & Binary Trees from textbook