M7. Tree (ADT/DS)

Instructor: Manikandan Narayanan

Weeks 8-9

CS2700 (PDS) Moodle: https://courses.iitm.ac.in/course/view.php?id=4892

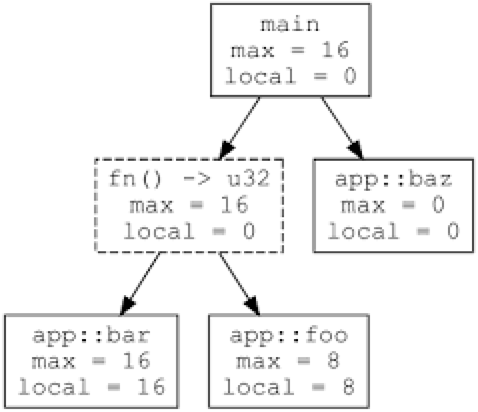
Acknowledgment of Sources

- Slides based on content from related
  - Courses:
    - IITM – Profs. **Rupesh**/Krishna(S)/Prashanth/Kartik's PDS (Thy/Lab) offerings (slides, quizzes, notes, lab assignments, etc. for instance from Rupesh's Jul 2019 offering - www.cse.iitm.ac.in/~rupesh/teaching/pds/jul19/)
    - *Most slides are based on Rupesh Nasre's slides – we thank him and acknowledge by marking **[RN]** in the bottom right of these slides.*
  - Books:
    - **Main textbook:** *"Data Structures and Algorithm Analysis in C++"* by ***Weiss*** (content, figures, slides, exercises/questions, etc.). – cited as [WeissBook]
    - Additional/optional book: *"Practice of Programming"* by *Kernighan and Pike* (style of programming, programming exercises/questions, etc.) – cited as [KPBook]

Outline for Module M7

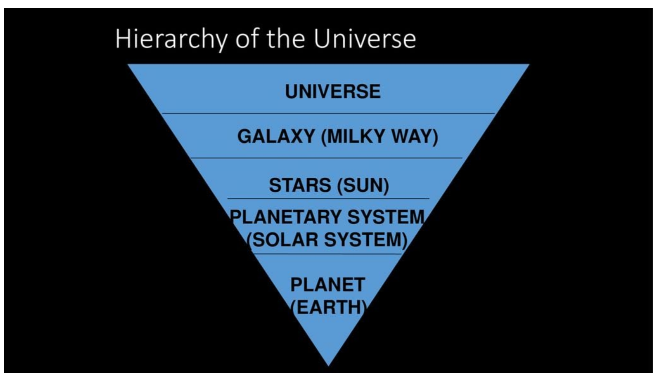
- **M7 Trees**
  - **M7.1 General Trees**
    - **Definition and Properties**
    - ADT and DS (Implementation)
    - Traversals
  - M7.2 Special Trees (e.g., Binary Trees)
    - Definition, ADT/DS and Properties
    - Applications and Traversals

Manager-Employee Relation

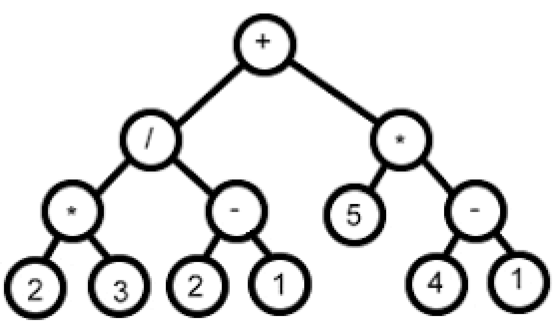


Modeling Computation

Trees



Planetary Hierarchy



Expression tree for 2*3/(2-1)+5*(4-1)

Expression Evaluation

[RN]

Nomenclature

- Root
- Stem
- Branches
- Leaves
- ~~Fruits~~
- ~~Flowers~~

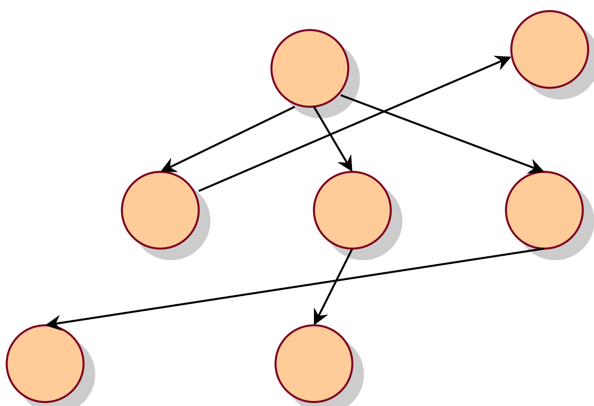
Edges

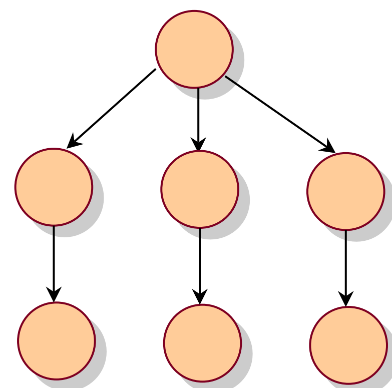
[RN]

Definition

A tree is a collection of nodes.
It could be empty.
// base case

Otherwise, it contains a root node,
connected to zero or more (child) nodes,
each of which is a tree in itself! // recursive





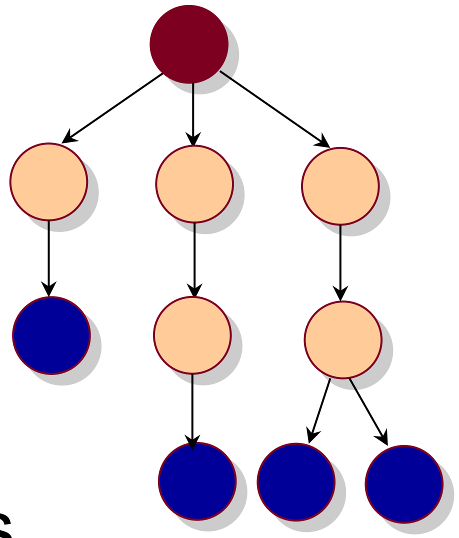
Alternatively, a tree is a collection of nodes and directed edges, such that each node except one has a single parent. The node without a parent node is the root.

[RN]

Nomenclature

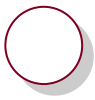
Root has no parent.
Leaves have no children.
Non-leaves are **internal nodes**.

Each node is *reachable* from the root.
The whole tree can be accessed via root.
Each node can be viewed as the root of its unique *subtree*.

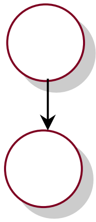


Empty Tree

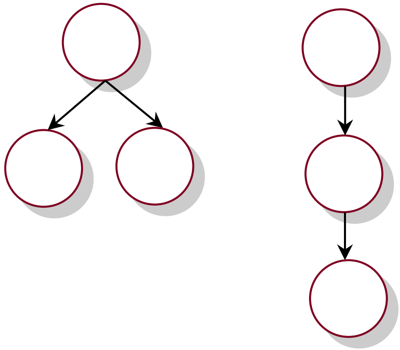
Tree with one node



Tree with two nodes



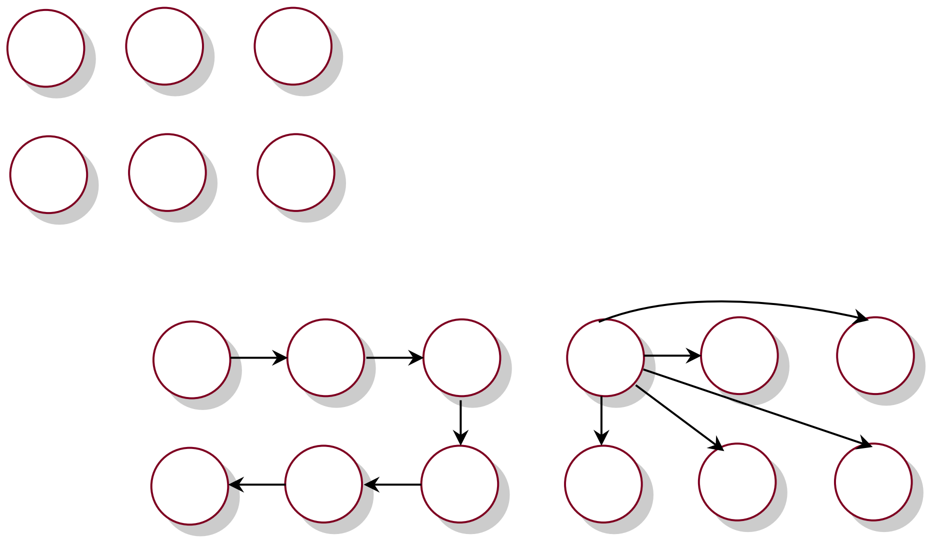
Trees with three nodes



[RN]

Properties - Warmup

- A tree has six nodes.
  - What is the minimum number of edges in the tree?
  - What is the maximum?
  - Generalization for N nodes?

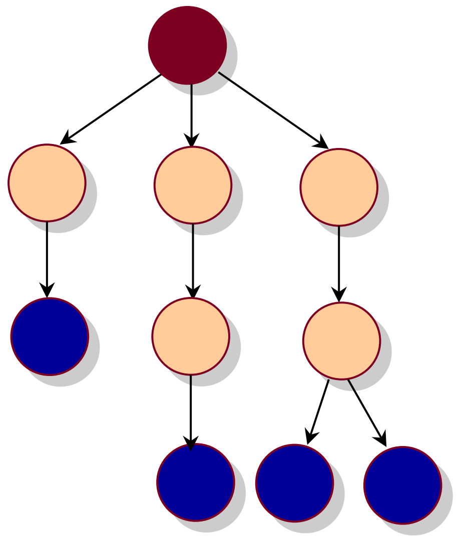


- How many (undirected) paths exist between two nodes?

[RN]

More Nomenclature

- Sibling
  - What is the maximum number of siblings a node may have in an N node tree?
- Grandparent, grandchild
- Ancestor, descendant
- Path, length
- Height, depth



[RN]

Exercises

- Given (a pointer to) a node in an employee tree, list all its direct and indirect subordinates.
- Same as above with the name of the employee given.
- Find distance between two nodes.
- Find tree diameter (max. distance).
- Convert infix to postfix (using a tree).
- Mirror a tree.
- Find if there is a directed path from p to q.

[RN]

Learning Outcomes

- Apply tree data structure in relevant applications.
- Construct trees in C++ and perform operations such as insert.
- Perform traversals on trees.
- Analyze complexity of various operations.

[RN]

Outline for Module M7

- **M7 Trees**
  - **M7.1 General Trees**
    - Definition and Properties
    - **ADT and DS (Implementation)**
    - **Traversals**
  - M7.2 Special Trees (e.g., Binary Trees)
    - Definition, ADT/DS and Properties
    - Applications and Traversals

Tree ADT

- **"Current Position":** As in List ADT, we've a notion of current (node-based location or) position in TreeADT, which is implemented using "PtrToNode (TreeNode *)"
  - **Key Position:** PtrToNode root;
- **Operations:**
  - **Traversals**: Depth-first (pre/post-order), Breadth-first (level-order) - O(N)
  - **Find/Search**: Done using our tree traversals – O(N) (Design decision – what if there are duplicates in the tree?)
  - **Insert**: TreeNode::addChild(PtrToNode x) – adds x as last child of current TreeNode - O(1)
  - **Remove**: Update parent's pointer to NULL (and free memory) – O(1) or O(N) depending on whether the removed node has children.

[RN]

Tree DS

Tree ADT can be implemented using arrays or linked data structures, with the latter being typically used.
We will use linked data structures (TreeNodes linked via parent-child relations) to implement Tree ADT.

Implementation

- A challenge is that the maximum number of children is unknown, and may vary dynamically.

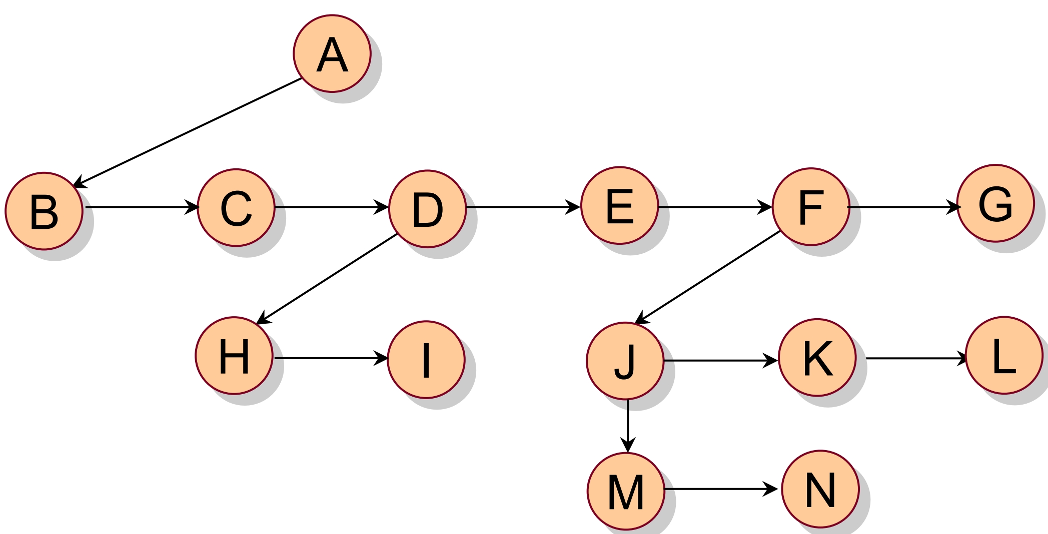
```c
typedef struct TreeNode *PtrToNode;

struct TreeNode {
        char data;
        PtrToNode firstChild;
        PtrToNode nextSibling;
};
```

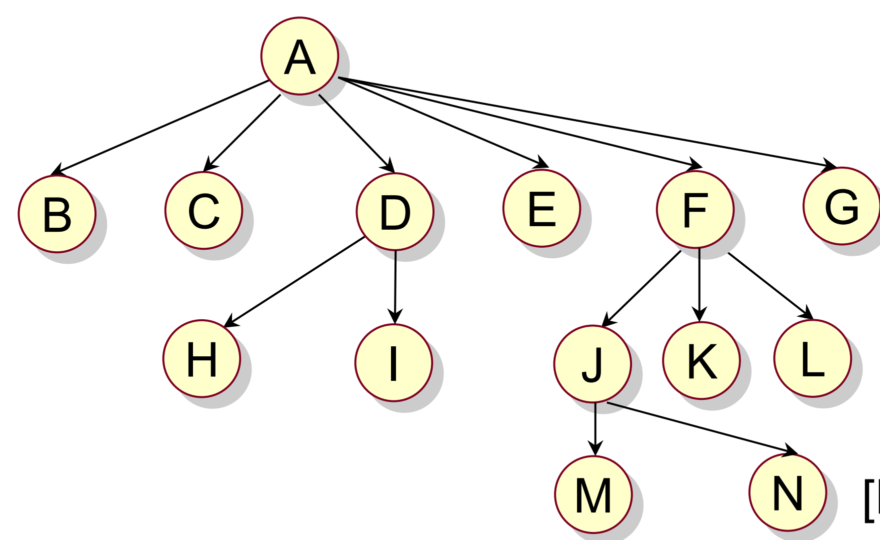
```cpp
#include <vector>
typedef struct TreeNode *PtrToNode;

struct TreeNode {
        char data;
        std::vector<PtrToNode> children;
};
```

C



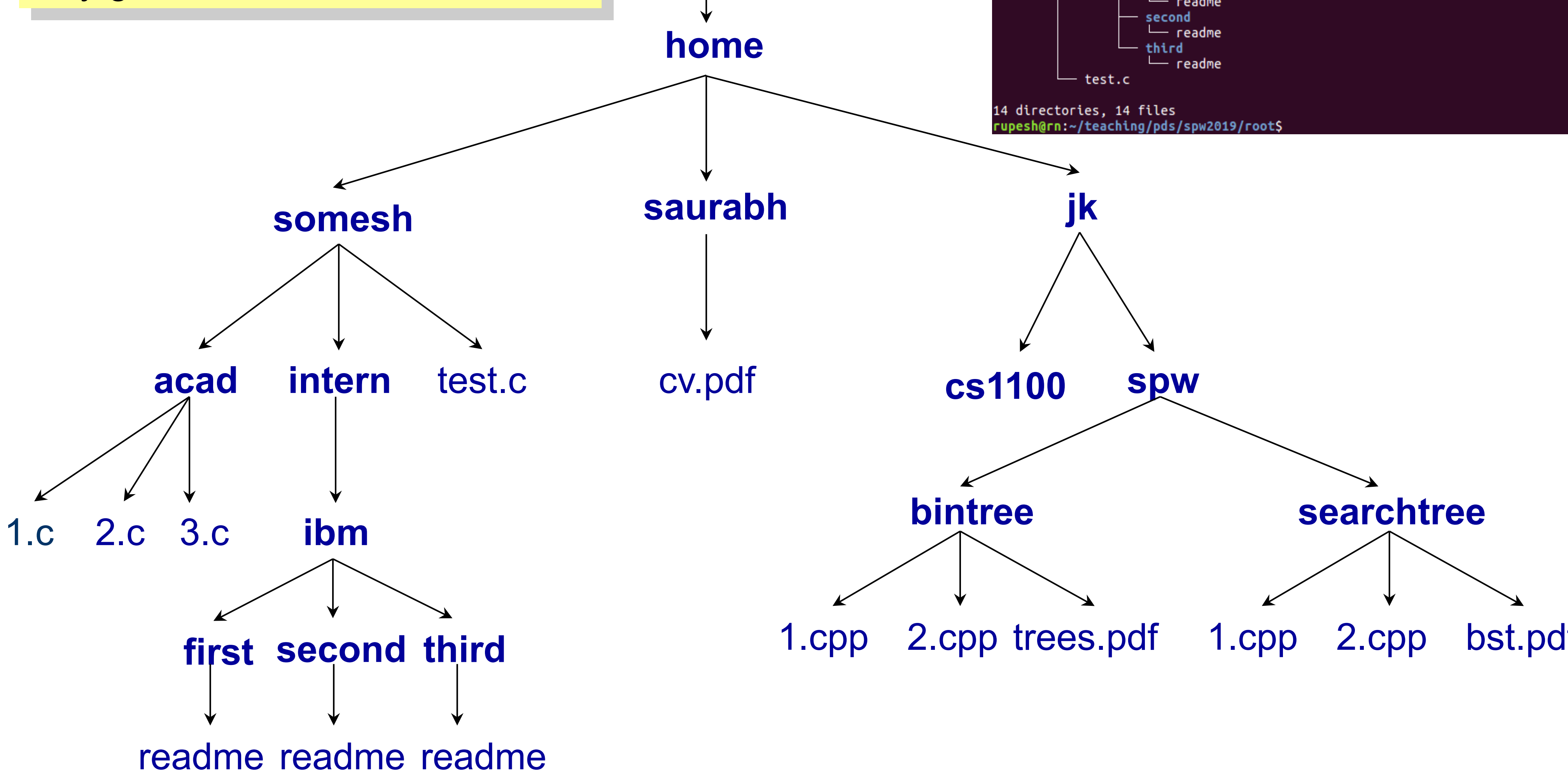
C++



[RN]

Directory Listing

There is a Linux command to list a directory in a tree-like format. Any guesses for the command?



Source: generaltree*.{hpp,cpp} [RN]

Traversals - Depth-first

- Preorder
  - Process each node <u>before</u> processing its children.
  - Children can be processed in any order.
- Postorder
  - Process each node <u>after</u> processing its children.
  - Children can be processed in any order.
- Preorder and postorder are examples of Depth-First Traversal.
  - Children of a node are processed <u>before</u> processing its <u>siblings</u>.
  - The other way is called Breadth-First or Level-Order Traversal.

[RN]

Traversals – Bread-first or Level-order traversal

Process nodes based on the level they are in, with closest to root first before other nodes!

Specifically, process all nodes in a lower level before processing nodes at higher levels.

For all nodes in the same level, processing typically done in left to right direction, but arbitrary order of processing such nodes in the same level is also sometimes allowed.

Preorder

Iterative

```
void Tree::preorder() {
	std::stack<PtrToNode> stack;
	stack.push(root);

	while (!stack.empty()) {
		PtrToNode rr = stack.top();
		stack.pop();
		if (rr) {
			rr->print();
			for (auto child : views::reverse(rr-
>children))
				stack.push(child);
		}
	}
}
```

Recursive

```
void Tree::preorder(PtrToNode rr) {
	if (rr) {
		rr->print();
		for (auto child : rr->children)
				preorder(child);
	}
}
void Tree::preorder() {
	preorder(root);
}
```

Source: generaltree*.{hpp,cpp} (views::reverse in “#include <ranges>”)

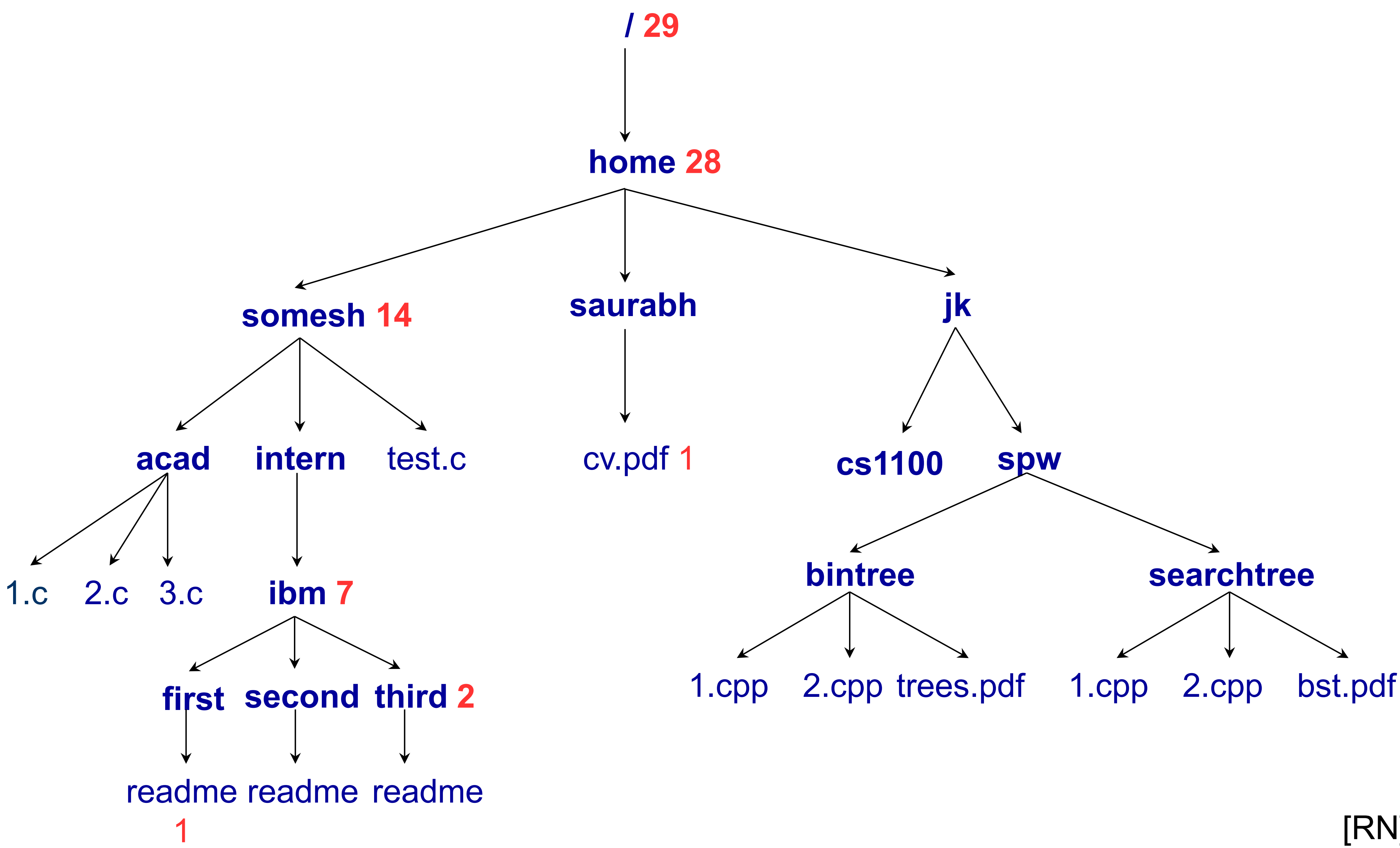
Homework: Indent files as per their depth. What is the code complexity? Note that indentation time also needs to be considered.

[RN]

Stack view (for iterative) and Fn. call tree view (for recursive)

Showed in board for an example

Find full size of each directory



[RN]

Postorder

Iterative

Try it out offline.
A bit trickier than iterative *pre*order!

Recursive

```
void Tree::postorder(PtrToNode rr) {
	if (rr) {
		for (auto child : rr->children)
			postorder(child);
		rr→print();
	}
}
void Tree::postorder() {
	postorder(root);
}
```

Source: generaltree*.{hpp,cpp}

[RN]

Level-order (BF) traversal – code and queue view

Iterative (Recursive is not natural here!!)

```
void Tree::levelorder() {
	std::queue<PtrToNode> queue;
	queue.push(root);

	while (!queue.empty()) {
		PtrToNode rr = queue.front();
		queue.pop();
		if (rr) {
			rr->print();
			for (auto child : rr->children)
				queue.push(child);
		}
	}
}
```

Source: generaltree*.{hpp,cpp}

Show queue view in board

Outline for Module M7

- **M7 Trees**
  - M7.1 General Trees
    - Definition and Properties
    - ADT and DS (Implementation)
    - Traversals
  - **M7.2 Special Trees (e.g., Binary Trees)**
    - **Definition, ADT/DS and Properties**
    - **Applications and Traversals**

Story so far...

- General trees
  - arbitrary number of children
  - Resembles several situations such as employees, files, ...
- Special trees
  - Fixed / bounded number of children
  - Resembles situations such as expressions, boolean flows, …
  - All the children may not be present.

  - *Binary Tree ADT*: Same as *Tree ADT*, but with additional **left and right child** for each node (null if not present), and additional **inorder** traversal.

[RN]

K-ary Trees

```
typedef struct TreeNode *PtrToNode;

struct TreeNode {
        int data;
        PtrToNode firstChild;
        PtrToNode nextSibling;
};
```

```
#include <vector>
typedef struct TreeNode *PtrToNode;

struct TreeNode {
        int data;
        std::vector<PtrToNode> children;
};
```

For a fixed K

```
typedef struct TreeNode *PtrToNode;

struct TreeNode {
        int data;
        PtrToNode children[K];
};
```

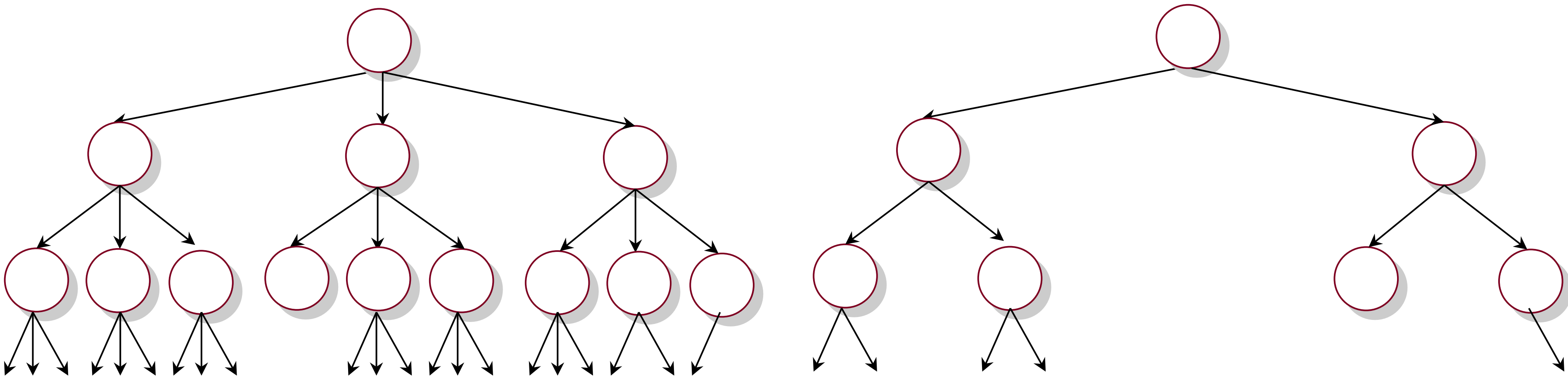
When K == 2

```
typedef struct TreeNode *PtrToNode;

struct TreeNode {
        int data;
        PtrToNode left;
        PtrToNode right;
};
```

[RN]

K-ary Trees



Ternary

Binary

For a fixed K

```
typedef struct TreeNode *PtrToNode;

struct TreeNode {
        int data;
        PtrToNode children[K];
};
```

When K == 2

```
typedef struct TreeNode *PtrToNode;

struct TreeNode {
        int data;
        PtrToNode left;
        PtrToNode right;
};
```

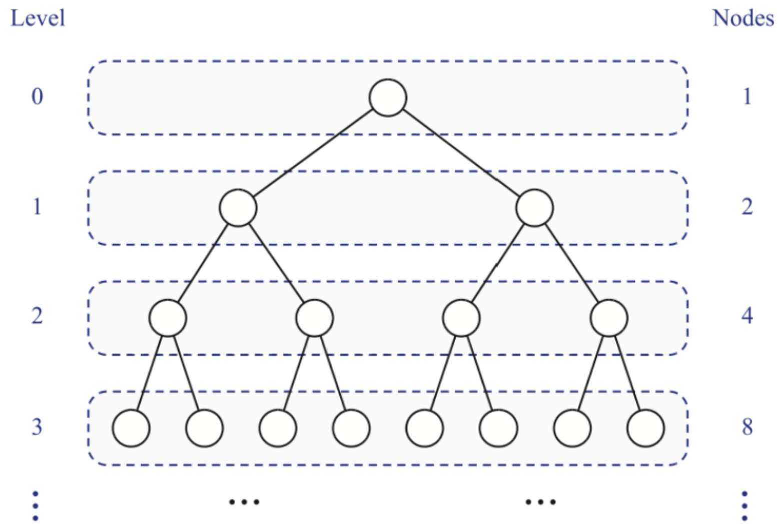
[RN]

Properties of Binary Trees

- For an N node binary tree (N > 0):
  - What is the maximum height? **N-1**
  - What is the minimum height? **floor($\log_2$(N)) (or ceil($\log_2$(N+1)-1))**
  - How many NULL pointers? **N + 1**
  - How many min/max leaves? **0/1, N / 2**
- What is the maximum number of nodes a binary tree of height H may have? **2^{H+1} - 1**
- Full nodes (nodes with two children):
  - how many minimum, maximum? **0, N / 2 - 1**
- Show that #full nodes + 1 == #leaves in a non-empty binary tree.

[RN]

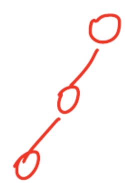
Properties of binary trees (contd.)



Binary Tree T

n, n_E, n_I : # of nodes, external nodes, internal nodes

h: height



(I) $h+1 \leq n \leq 2^{h+1}-1$

(II) $1 \leq n_E \leq 2^h$ ← (Note: What if a leaf is at a lower level (level < h)? Can you increase the number of leaves in such a tree, without increasing the tree height beyond h?)

(III) $h \leq n_I \leq 2^h - 1$

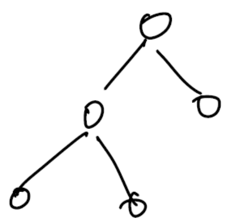
(IV) $\log(n+1)-1 \leq h \leq n-1$

[Source: Prashanth]

Properties of **proper** binary trees

For a proper tree T,

(1) $2h+1 \le n \le 2^{h+1}-1$



(2) $h+1 \le n_E \le 2^h$

(3) $h \le n_I \le 2^h-1$

(4) $\log(n+1)-1 \le h \le \frac{n-1}{2}$

[Source: Prashanth]

Properties of **proper** binary trees (contd.)

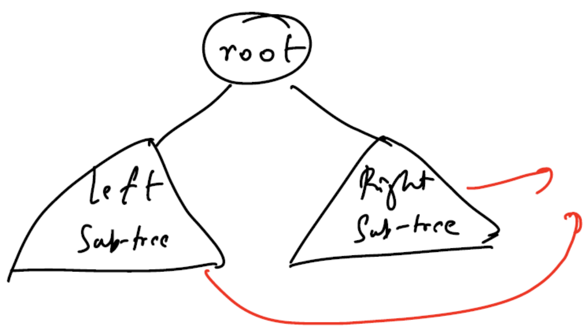
Claim: In a non-empty "proper" binary tree T, $\quad n_E = n_I + 1$

Pf: Induction

Base Case: Just the root node.

$n_I = 0, \quad n_E = 1$

Induction step:



Satisfy induction hypothesis

$n_E^l = n_I^l + 1 \qquad n_E^r = n_I^r + 1$

Overall, $\quad n_E = n_I^l + n_I^r + 2,$

$n_I = n_I^l + n_I^r + 1$ ← root

[Source: Prashanth]

Applications of Tree ADT/DS

Expression tree – prefix/infix/postfix

Huffman encoding (will look at later during Heaps)

RRTs (Rapidly Evolving Random Trees) for Robot Path Planning -
https://github.com/nikhilchandak/Rapidly-Exploring-Random-Trees/

Space-filling trees or fractal trees for artistic visualization/generation of trees -
https://www.istockphoto.com/search/more-like-this/1493934770?assettype=image

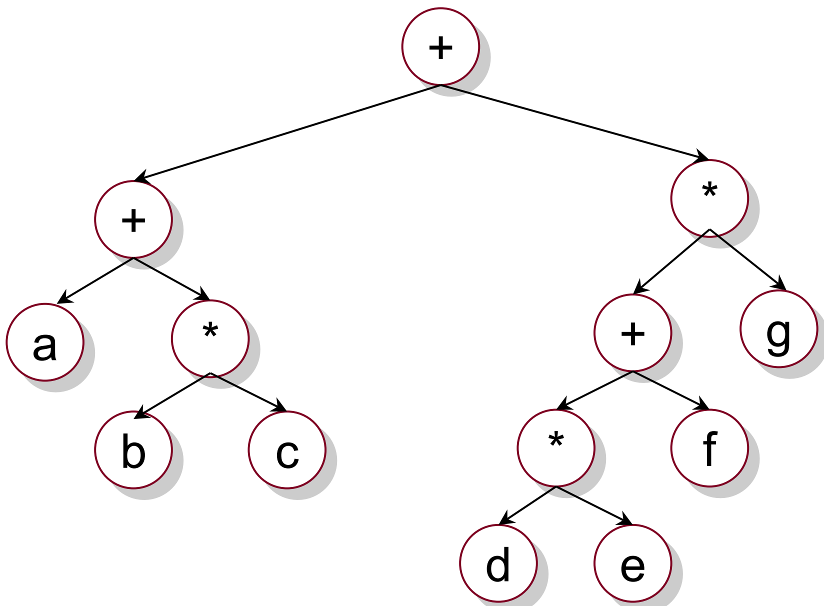
Phylogenetic Tree or Tree of Life (often parsed/imported from a Newick-format string)

…

Homework: Look at code to parse a binary tree from its Newick-formatted string here: **Source: {binarytree.hpp, binarytree_parsenewick.cpp}**

Expression Trees

((a + (b * c)) + (((d * e) + f) * g))



Where did the parentheses go?
Can we write the expression itself in a way that no parentheses are required?

[RN]

Traversals

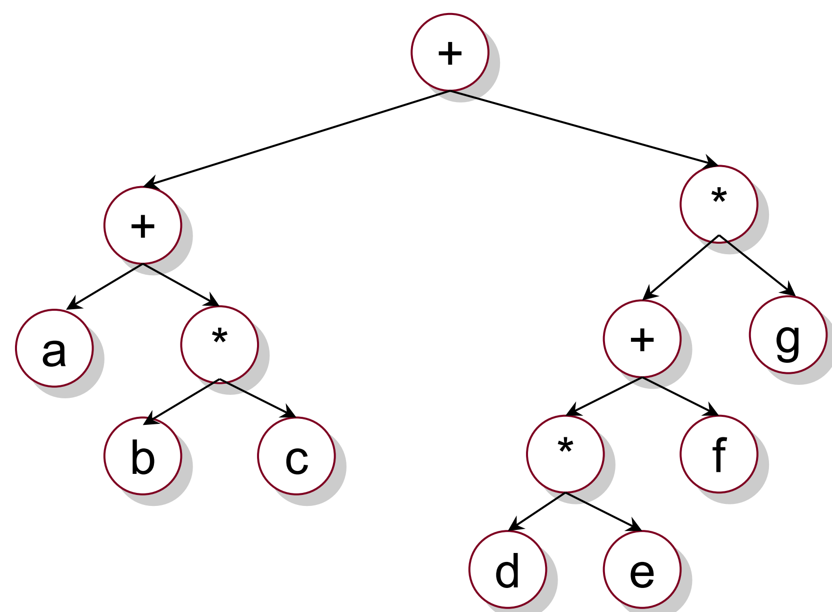
- preorder (NLR)
- postorder (LRN)
- inorder (LNR)

```
void Tree::inorder(PtrToNode rr) {
	if (rr) {
		inorder(rr->left);
		rr->print();
		inorder(rr->right);
	}
}
void Tree::inorder() {
	inorder(root);
	std::cout << std::endl;
}
```

Find output of this code on this example tree.

a+b*c+d*e+f*g

Actual expression:
((a + (b * c)) + (((d * e) + f) * g))



[RN]

Traversals

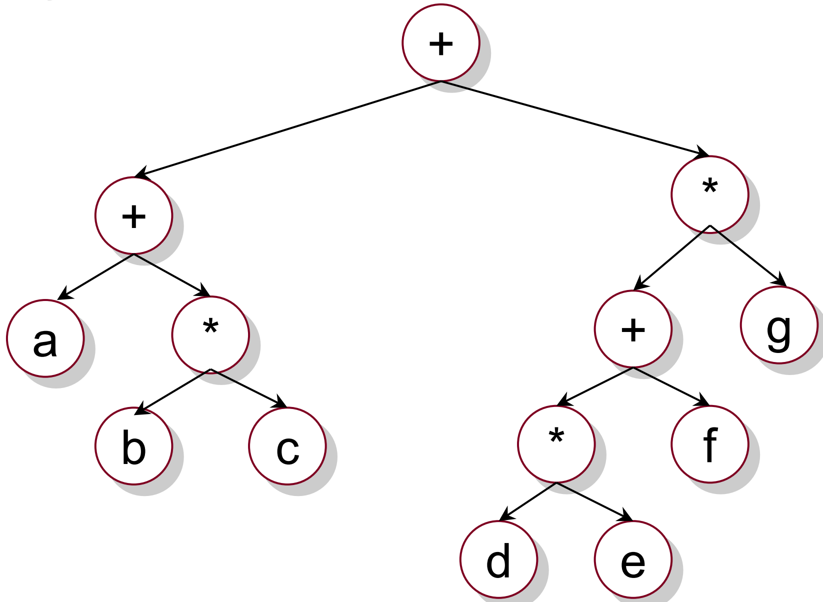
- preorder
- **Postorder**
- inorder

```
void Tree::postorder(PtrToNode rr) {
        if (rr) {
                postorder(rr->left);
                postorder(rr->right);
                rr->print();
        }
}
void Tree::postorder() {
        postorder(root);
        std::cout << std::endl;
}
```

Find output of this code on this example tree.

abc*+de*f+g*+

Operator precedence encoded.



[RN]

Infix, Prefix, Postfix

Infix	Prefix	Postfix
A + B * C + D		
(A + B) * (C + D)		
A * B + C * D		
A + B + C + D		
A * B * C + D		

[RN]

Infix, Prefix, Postfix

Infix	Prefix	Postfix
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +
A * B * C + D	+ * * A B C D	A B * C * D +

- The order of operands (A, B, C, D) remains the same in all the expressions.
- Operators in prefix are in the opposite order compared to their postfix versions.

[RN]

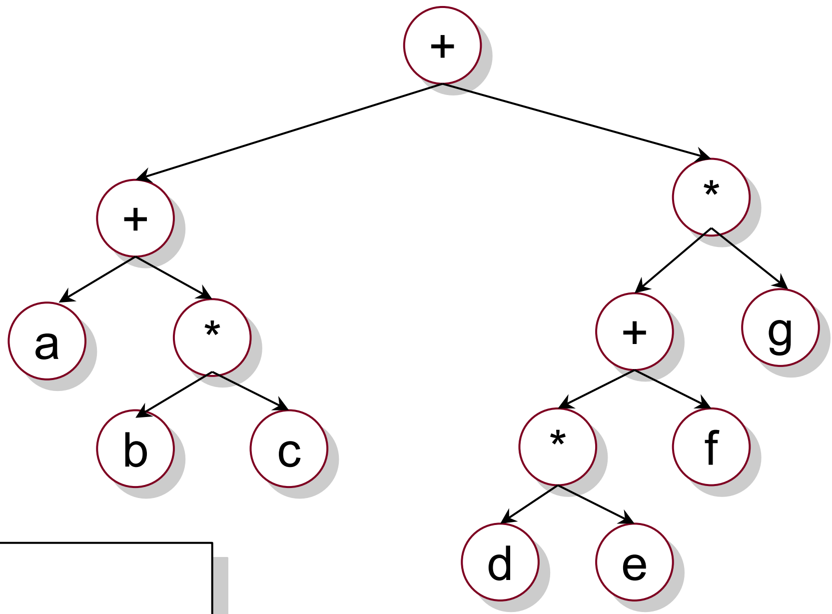
Recall: Evaluating postfix

- Find the value of `5 1 2 3 * - 4 + 6 * -`.
- Write a program to evaluate a postfix expression.
  - Assume digits, +, –, *, /.

```
For each symbol in the expression
        If the symbol is an operand
                Push its value to a stack
        Else if the symbol is an operator
                Pop two nodes from the stack
                Apply the operator on them
                Push result to the stack
```

[RN]

Similar Logic: Postfix to Expression Tree



a b c * + d e * f + g * +

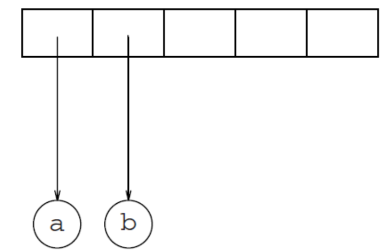
For each symbol in the expression
    If the symbol is an **operand**
        Push its node to stack
    Else if the symbol is an **operator**
        Pop two nodes from the stack
        Connect those to the operator
        Push root of the tree to stack

Source: binarytree.hpp, binarytree_expntree.cpp

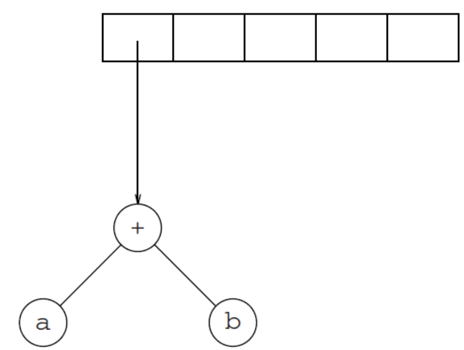
[RN]

Postfix2ExpTree

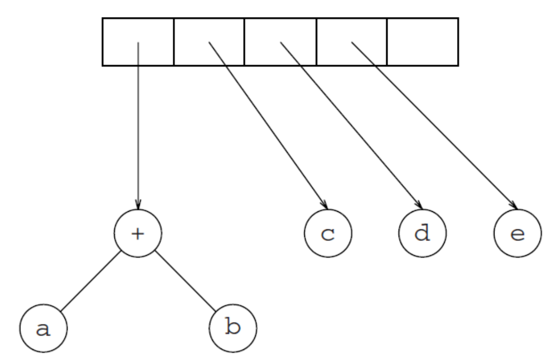
ab+cde+**



Next, a + is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



Next, c, d, and e are read, and for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



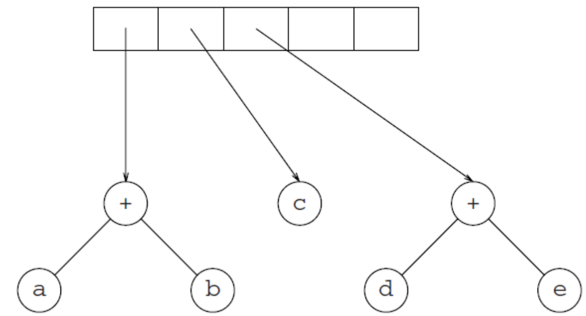
Now a + is read, so two trees are merged.

[2] For convenience, we will have the stack grow from left to right in the diagrams.

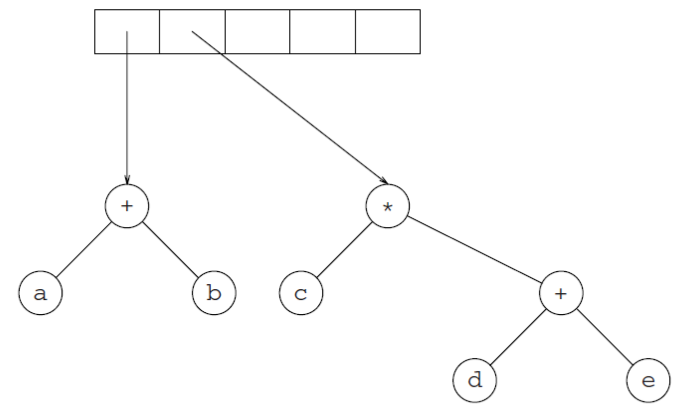
[WeissBook]

Postfix2ExpTree

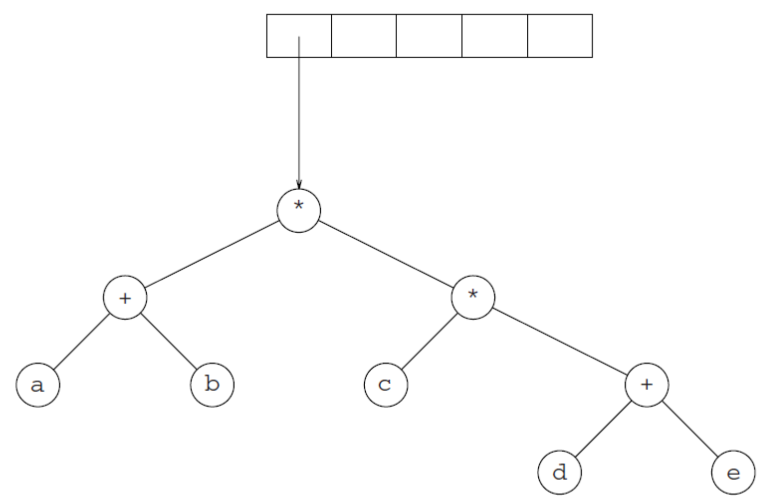
ab+cde+**



Continuing, a * is read, so we pop two tree pointers and form a new tree with a * as root.



Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.



[WeissBook]

Expression Tree Application – Final Notes

Once expression tree is constructed by parsing postfix form, we can do several operations on it, like

converting it to infix or prefix (using in- or pre-order traversal),

evaluating the expression directly from the tree (by applying operator recursively on the evaluated left and right subtrees in a postorder fashion), etc.

Source: binarytree.hpp, binarytree_expntree.cpp

Tree traversals – final summary/notes

Preorder (for any tree): *(impl.: recursive or stack-based iterative)*
Visit the root first, before recursively visiting the child subtrees of this root.
Useful for listing the contents of the tree like in the UNIX tree command.

Postorder (for any tree): *(impl.: recursive or stack-based iterative)*
Traverse the child subtrees of root, and then visit the root.
Used in:
- post-processing subtrees (for instance to find storage space of directories),
- creating a tree bottom-up (postorder) by L-to-R parsing of a postfix expn or Newick string, and
- deleting an entire directory (where the subdirs are deleted before deleting the current dir).

Inorder (for binary tree): *(impl.: recursive or stack-based iterative)*
Traverse the left subtree first, visit the root, and then traverse the right subtree.
This traversal produces sorted output for ``Binary Search Trees''.

Levelorder (for any tree): *(impl.: queue-based iterative)*
Traverse all nodes in lower levels before those in higher levels.
Useful for finding all individuals of the same generation in a family tree (later, we will discuss its uses in graphs for finding shortest paths between nodes).

Learning Outcomes

- Apply tree data structure in relevant applications.
- Construct trees in C++ and perform operations such as insert.
- Perform traversals on trees.
- Analyze complexity of various operations.

[RN]