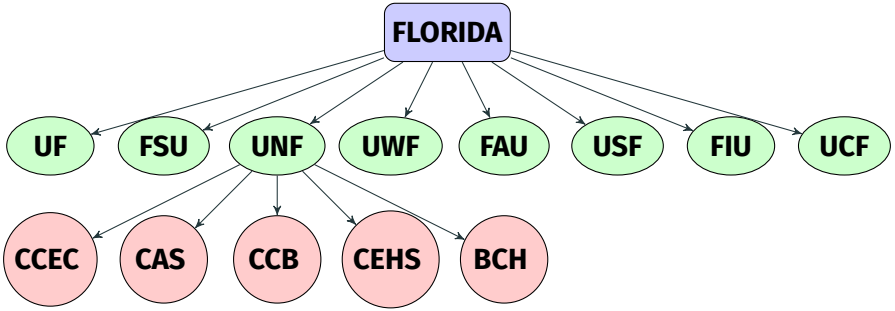
Trees

Dr. Anirban Ghosh

School of Computing
University of North Florida

Introduction

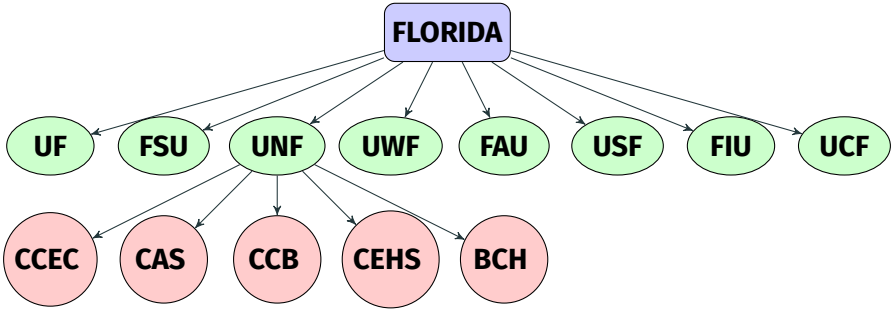


Informal definition

A **tree** is an abstract data type that stores elements hierarchially. With the exception of the top element, each element in a tree has a parent element and zero or children elements.

☞ ***Your first non-linear ADT!***

Formal definition

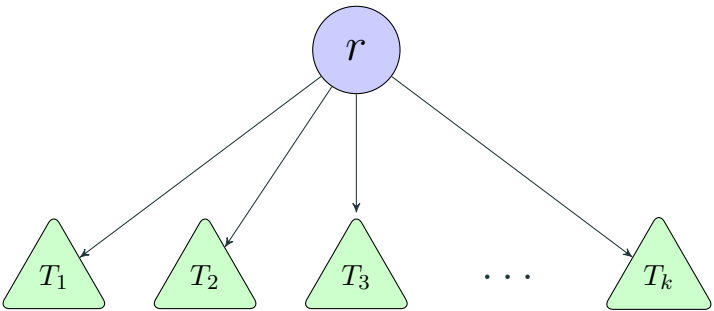


Definition

A tree T is a set of **nodes** storing elements such that the nodes have a **parent-child** relationship that satisfies the following properties:

1. If T is nonempty, it has a special node, called the **root** of T, that has no parent.
2. Each node v of T different from the root has a unique **parent** node w; every node with parent w is a **child** of w

Recursive definition



Recursive definition

A tree T is either empty or consists of a node r, called the root of T, and a (possibly empty) set of subtrees $T_1, T_2, \ldots, T_k$, whose roots are the children of r.

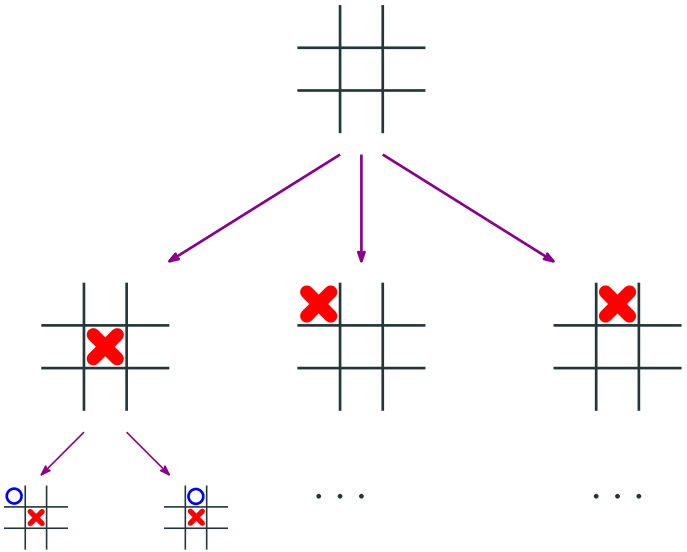
Applications

- Easy real-world uses: family tree, organizational charts, etc.
- Are widely used in computing to represent various kinds of hierarchical structures: directory structure, topologies in computer networks, etc.
- Game programming
- Machine learning
- Compiler design
- Operating systems
- Computer graphics
- Database management systems
- Searching algorithms
- Used as an auxiliary data structure for many algorithms
- ...

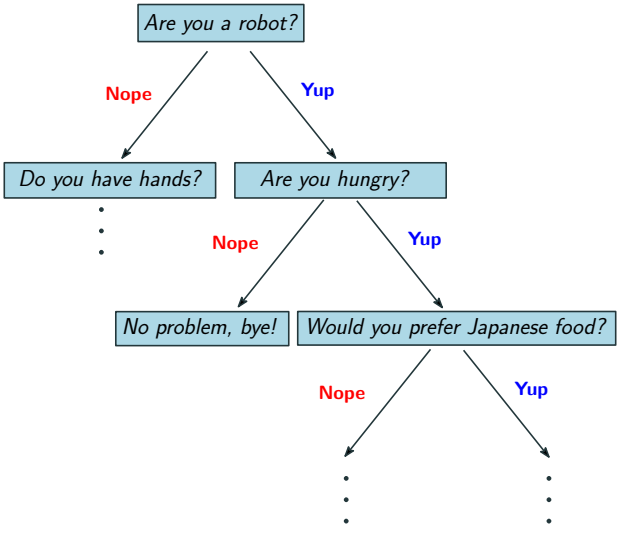
Directory structures in file systems

```
n01388139@UNF-C02DREMKMD6M Code % tree -L 3
.
└── COP3530
    ├── COP3530.iml
    ├── out
    │   └── production
    ├── oz.txt
    └── src
        ├── analysis
        ├── arraysandLLs
        ├── exam1
        ├── hashing
        ├── hw1
        ├── hw2
        ├── hw3
        ├── hw4
        ├── ood
        ├── primer
        ├── quiz
        ├── recursion
        ├── stacksandqueues
        └── tester

19 directories, 2 files
```

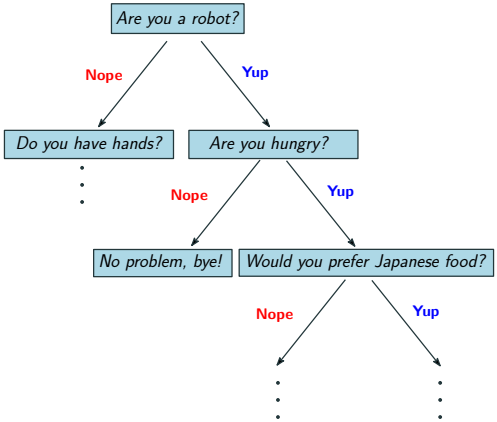
Game trees in AI



Binary trees



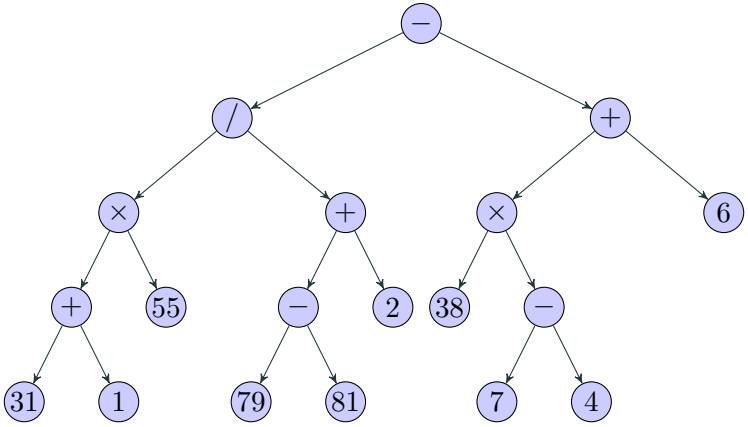
Binary trees



Definition

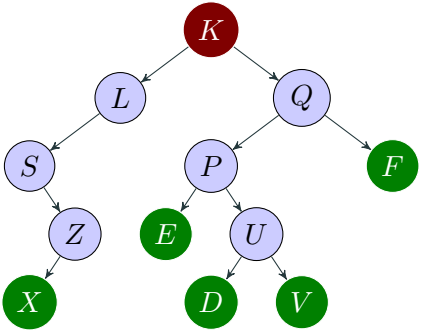
A binary tree is a tree in which every node has ≤ 2 children

Expression trees



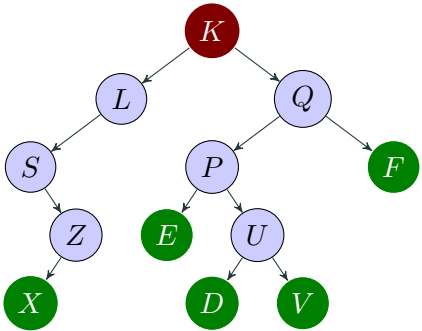
$$31 + 1 \times 55/79 - 81 + 2 - 38 \times 7 - 4 + 6$$

Terminologies



- **Root.** The node that has no parent
- **Leaf node.** A node that has no children
- **Internal node.** A node is internal if it has at least one child
- **Ancestors of node.** The nodes that are on the path from the root to the node
- **Depth of a node.** The number of ancestors it has; the root has depth zero
- **Height of a tree.** Maximum possible depth of a node in the tree

Fun results

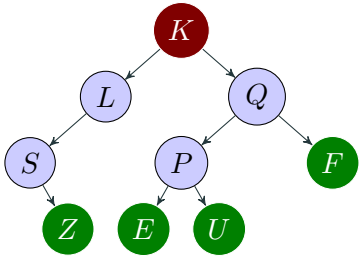


Let h denote the height and n the number of nodes. Also, let n_2 denote the number of nodes having two children and n_ℓ denote the number of leaves.

- $h + 1 \leq n \leq 2^{h+1} - 1$
- $\log_2(n + 1) - 1 \leq h \leq n - 1$
- $n_\ell = n_2 + 1$

How to represent binary trees?

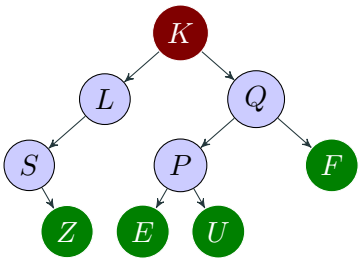
Approach 1: using a linked structure. Every node has left and right fields that points to the left and right child, respectively; node structure looks quite similar to that of a doubly linked-list node class

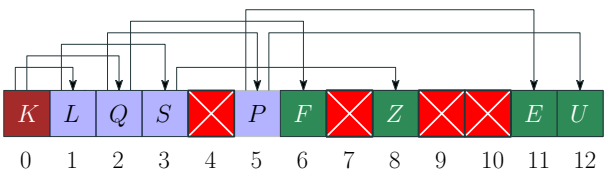


Space efficient; uses $O(n)$ space

How to represent binary trees?

Approach 2: using an array. The root is at index 0. If a node has index i, then its left child (if any) is at index $2i + 1$ and its right child (if any) is at $2i + 2$. Parent of the node at index i can be found at $\lfloor (i - 1)/2 \rfloor$





Space inefficient; uses $O(2^n)$ space

How to traverse a tree?

inorder(node)

1. if `node` has a left child, then recursively call `inorder(node.left);`
2. `visit(node.element);`
3. if `node` has a right child, then recursively call `inorder(node.right);`

preorder(node)

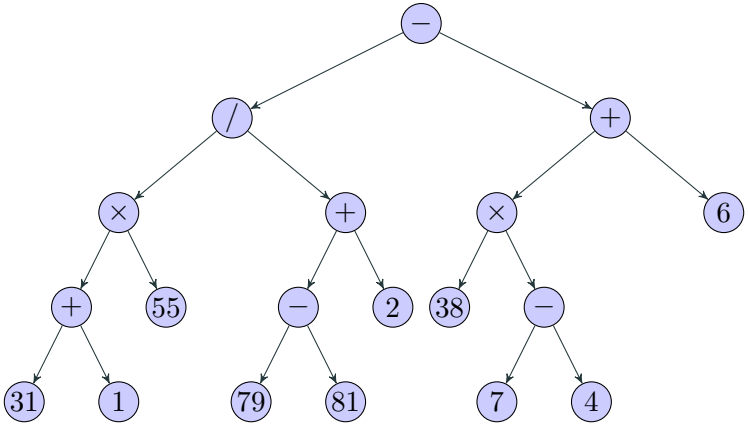
1. `visit(node.element);`
2. if `node` has a left child, then recursively call `preorder(node.left);`
3. if `node` has a right child, then recursively call `preorder(node.right);`

postorder(node)

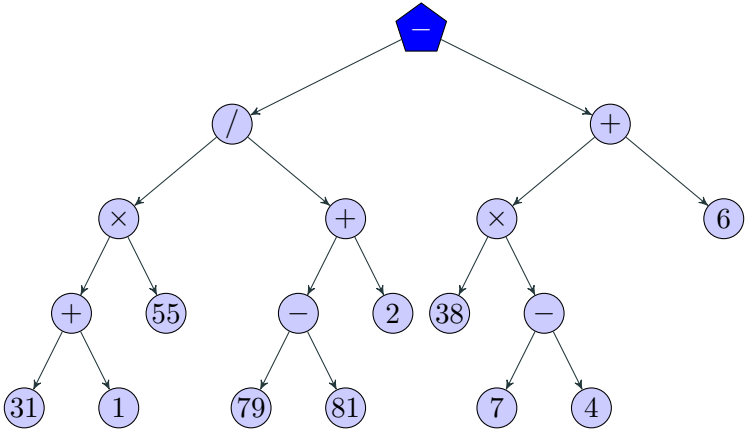
1. if `node` has a left child, then recursively call `postorder(node.left);`
2. if `node` has a right child, then recursively call `postorder(node.right);`
3. `visit(node.element);`

Inorder traversal demo

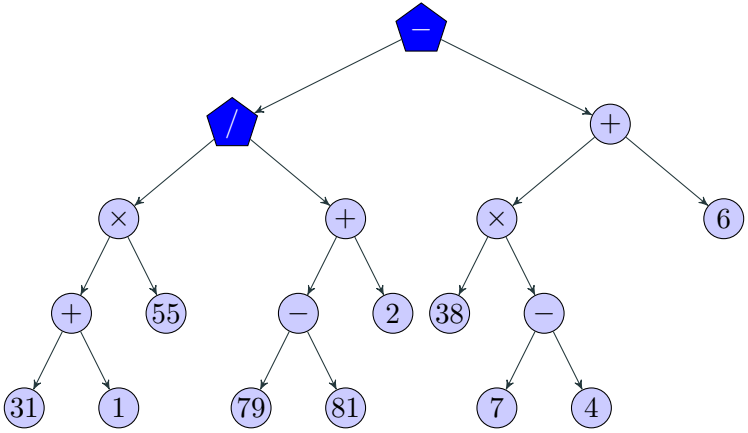
Inorder traversal on an expression tree



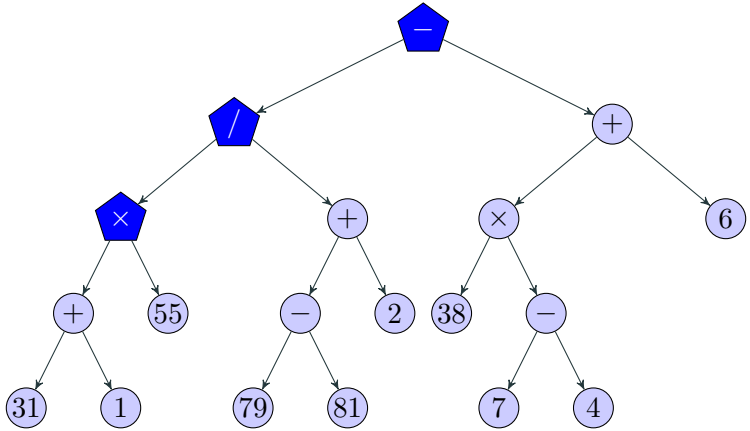
Inorder traversal on an expression tree



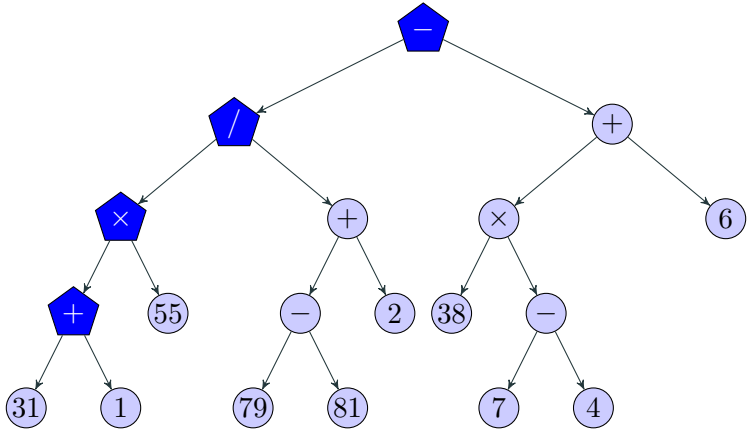
Inorder traversal on an expression tree



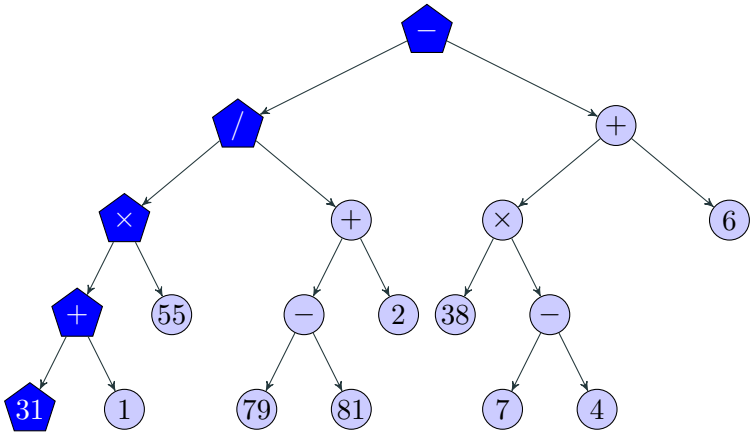
Inorder traversal on an expression tree



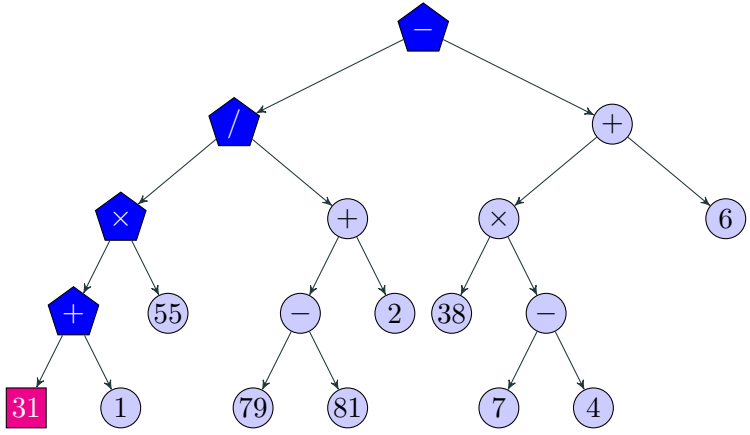
Inorder traversal on an expression tree



Inorder traversal on an expression tree

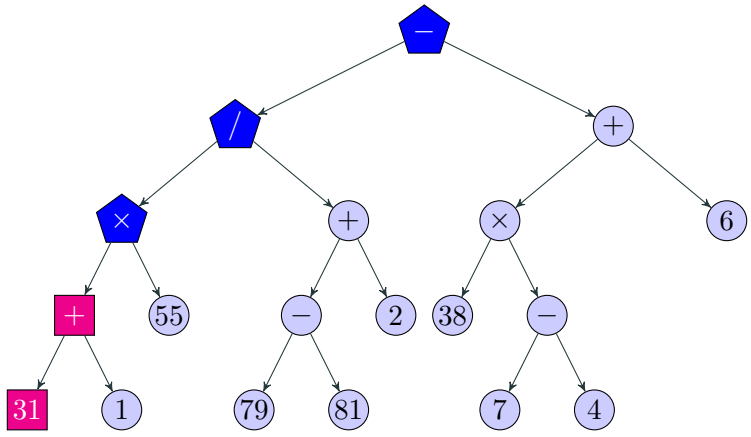


Inorder traversal on an expression tree



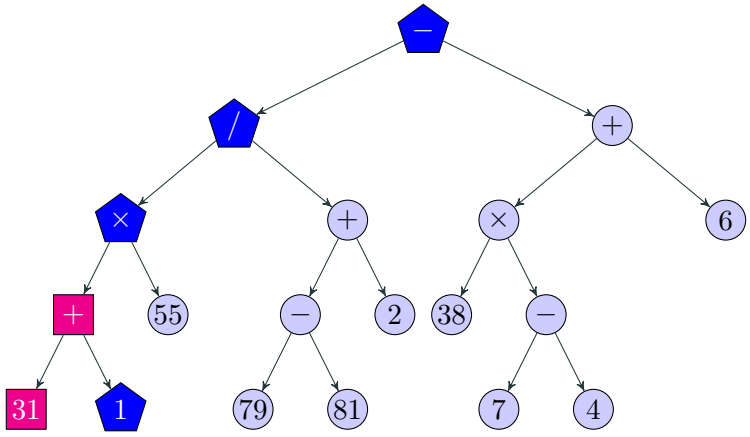
31

Inorder traversal on an expression tree



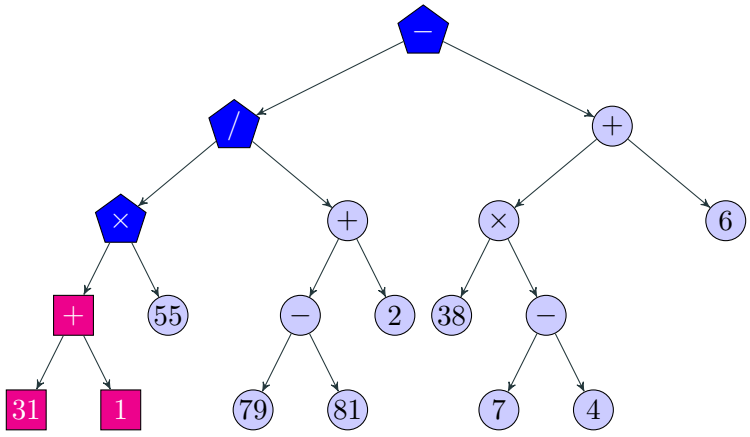
$31 +$

Inorder traversal on an expression tree



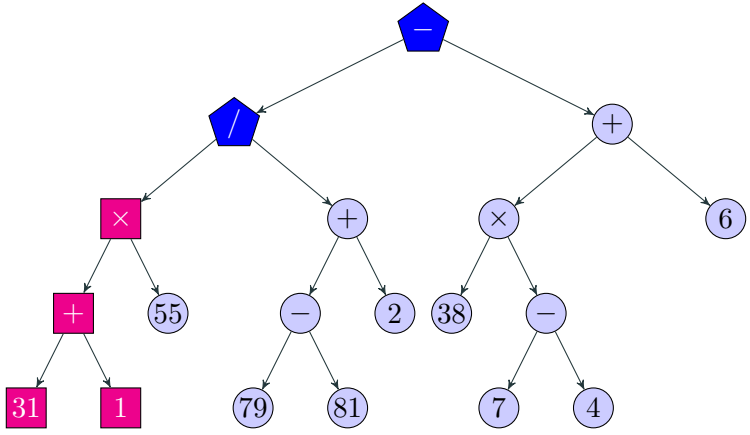
$31 +$

Inorder traversal on an expression tree



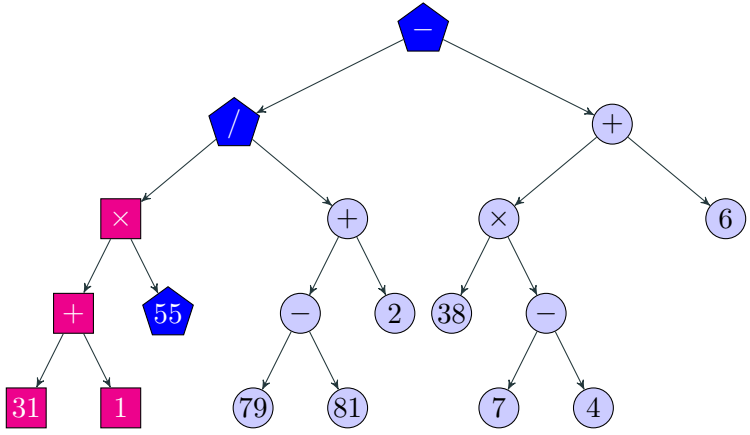
$31 + 1$

Inorder traversal on an expression tree



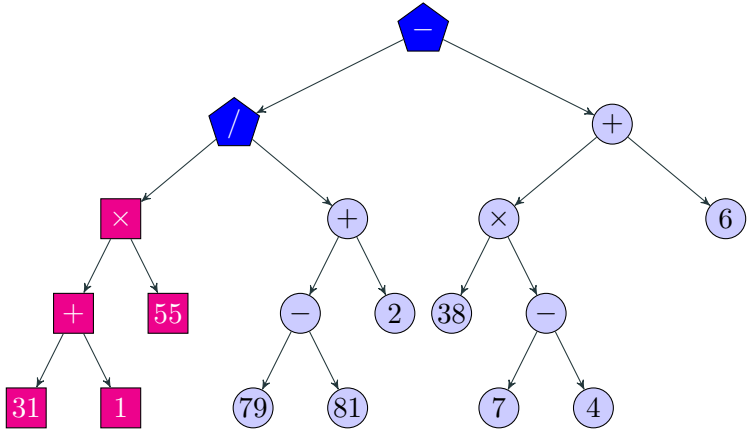
$31 + 1 \times$

Inorder traversal on an expression tree



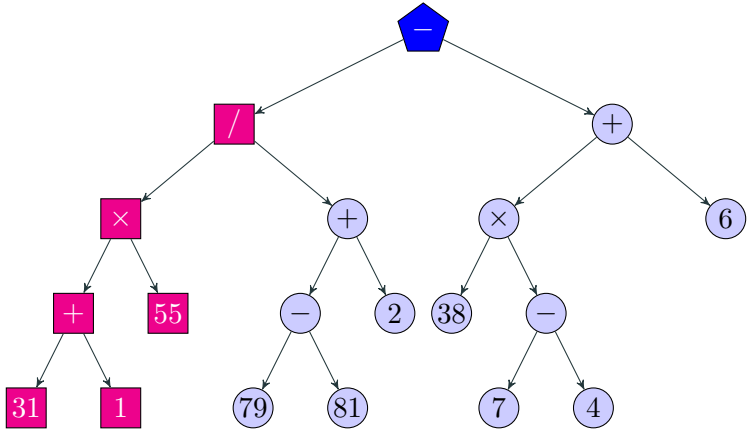
$31 + 1 \times$

Inorder traversal on an expression tree



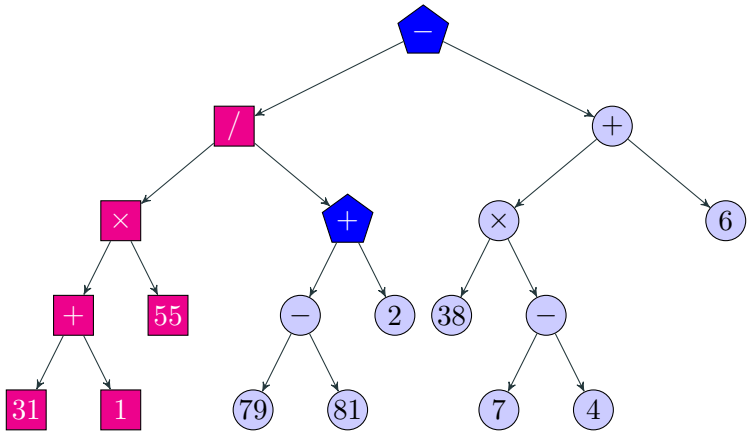
$$31 + 1 \times 55$$

Inorder traversal on an expression tree



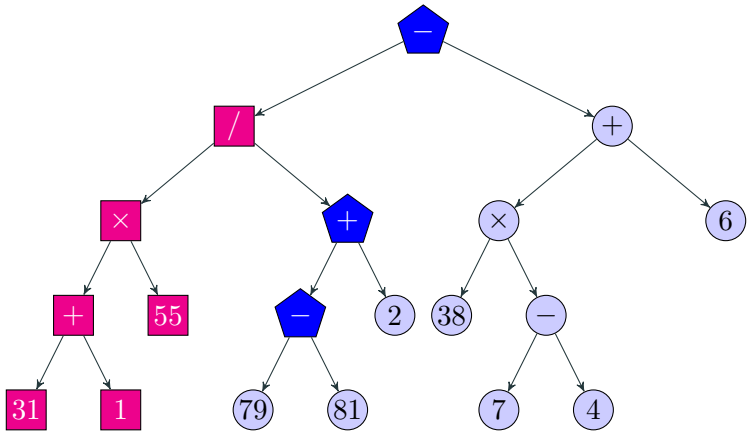
$$31 + 1 \times 55/$$

Inorder traversal on an expression tree



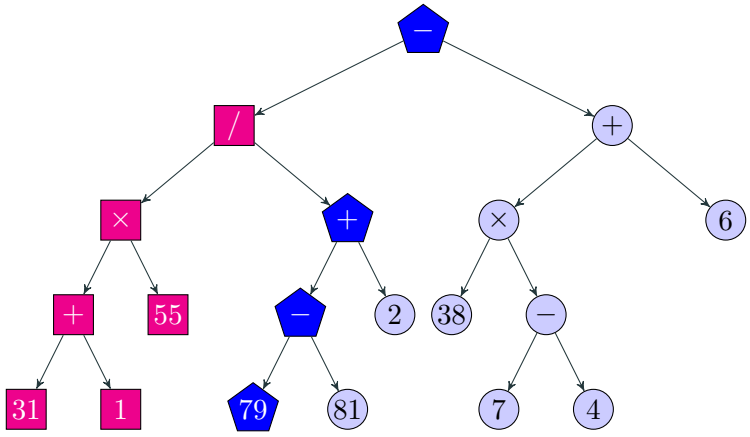
$$31 + 1 \times 55/$$

Inorder traversal on an expression tree



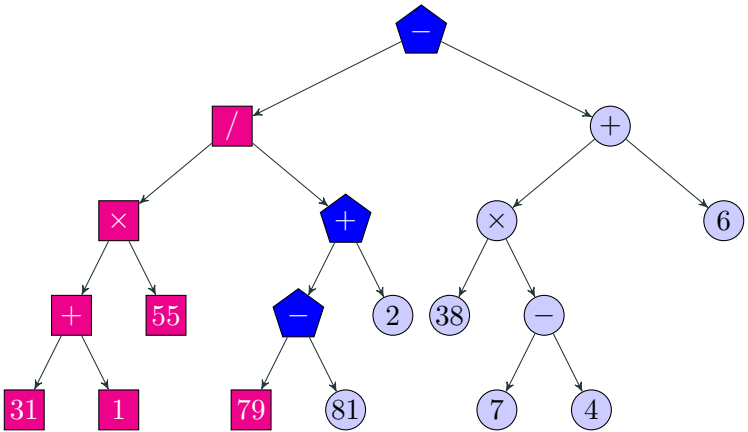
$$31 + 1 \times 55/$$

Inorder traversal on an expression tree



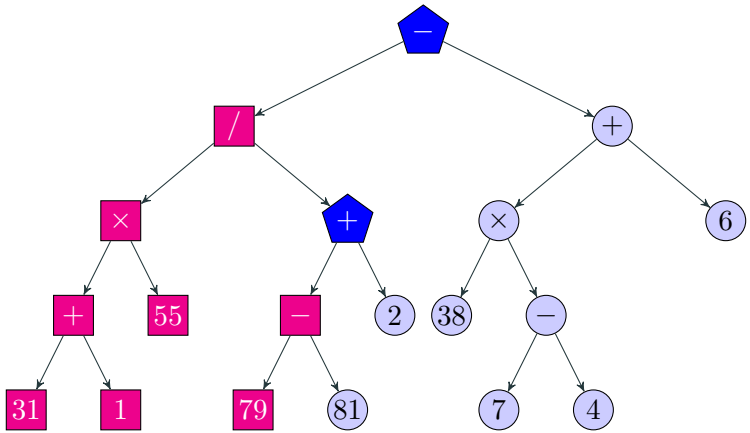
$$31 + 1 \times 55/$$

Inorder traversal on an expression tree



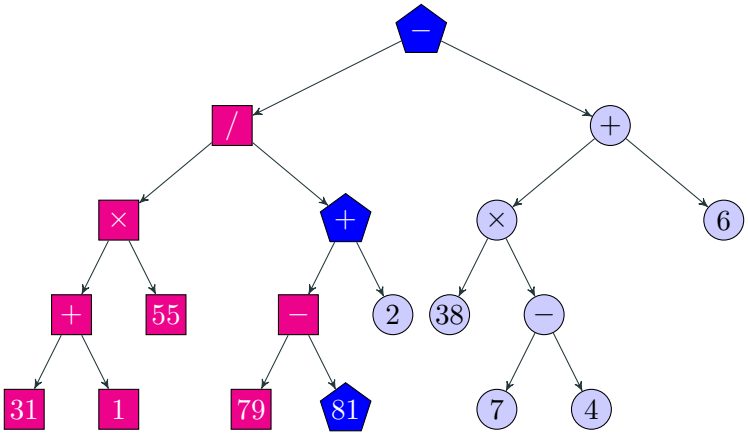
$$31 + 1 \times 55/79$$

Inorder traversal on an expression tree



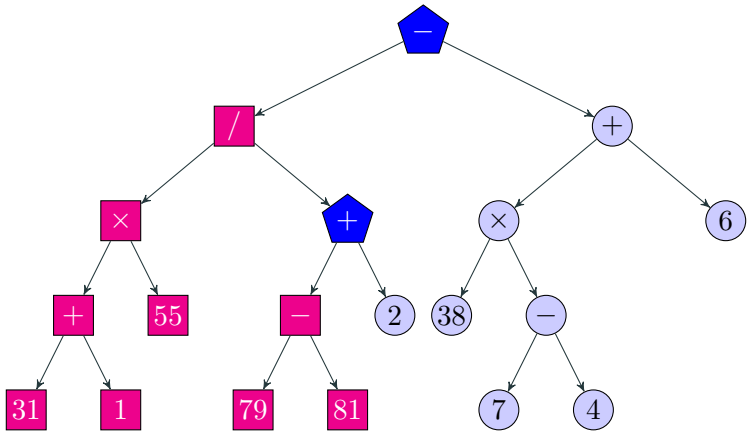
$$31 + 1 \times 55/79 -$$

Inorder traversal on an expression tree



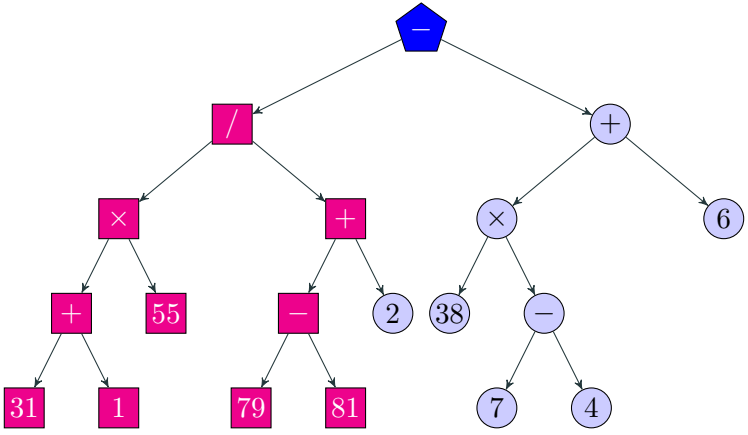
$$31 + 1 \times 55/79 -$$

Inorder traversal on an expression tree



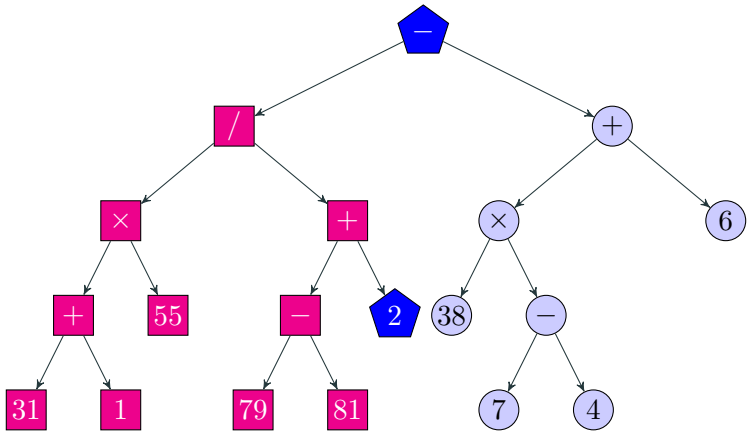
$$31 + 1 \times 55/79 - 81$$

Inorder traversal on an expression tree



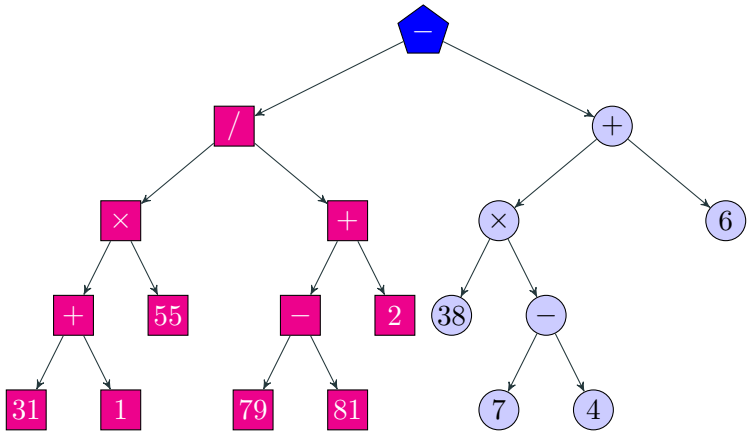
$$31 + 1 \times 55/79 - 81 +$$

Inorder traversal on an expression tree



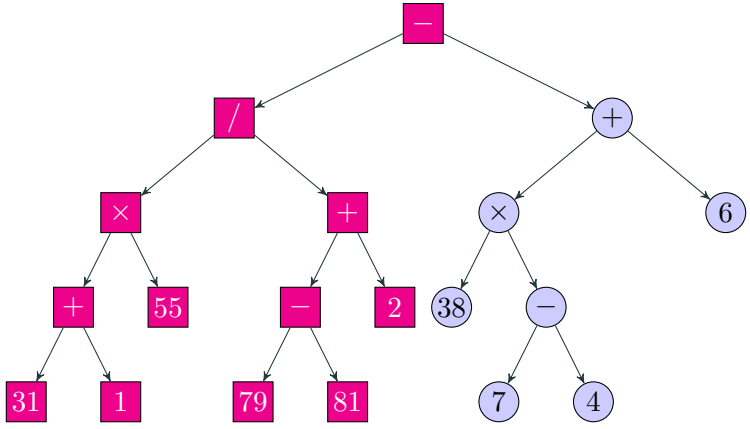
$$31 + 1 \times 55/79 - 81 +$$

Inorder traversal on an expression tree



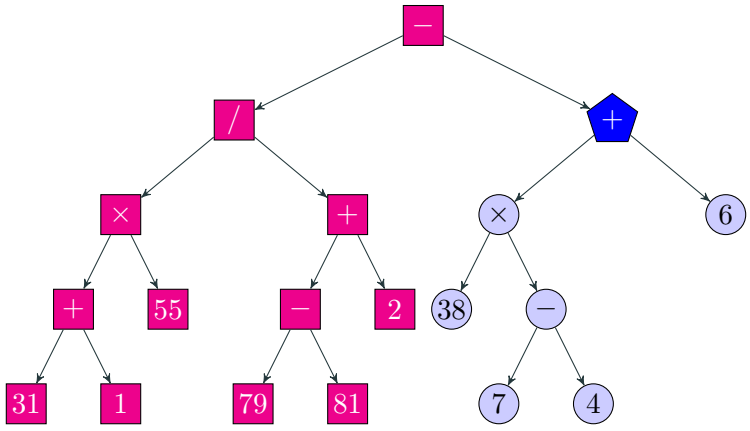
$$31 + 1 \times 55/79 - 81 + 2$$

Inorder traversal on an expression tree



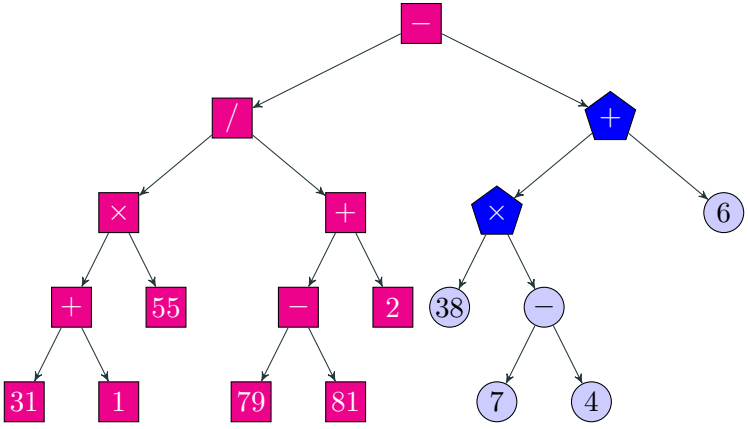
$$31 + 1 \times 55/79 - 81 + 2 -$$

Inorder traversal on an expression tree



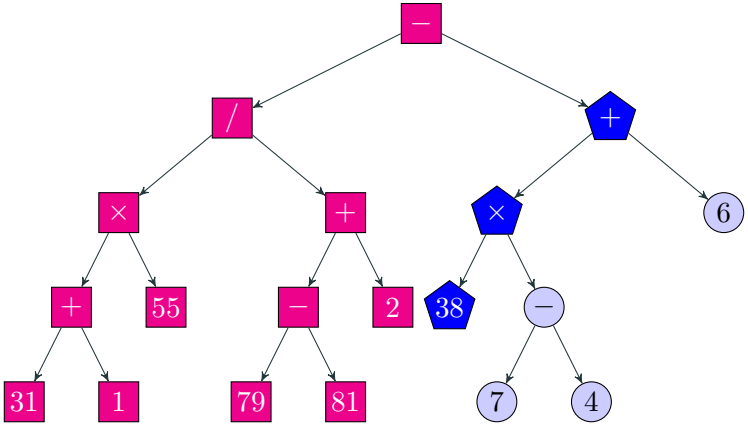
$$31 + 1 \times 55/79 - 81 + 2 -$$

Inorder traversal on an expression tree



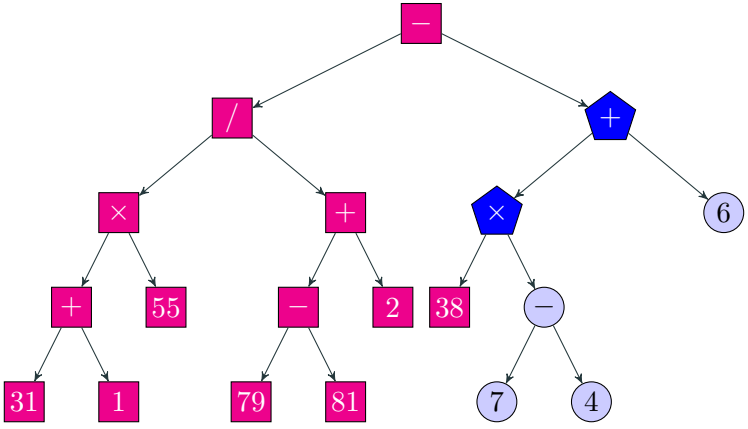
$$31 + 1 \times 55/79 - 81 + 2 -$$

Inorder traversal on an expression tree



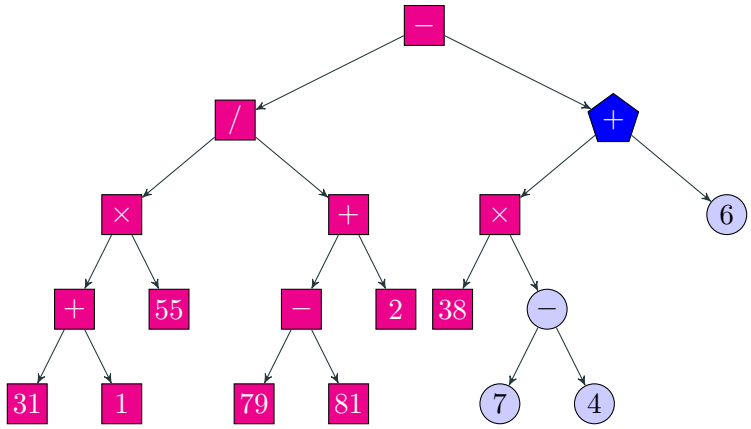
$$31 + 1 \times 55/79 - 81 + 2 -$$

Inorder traversal on an expression tree



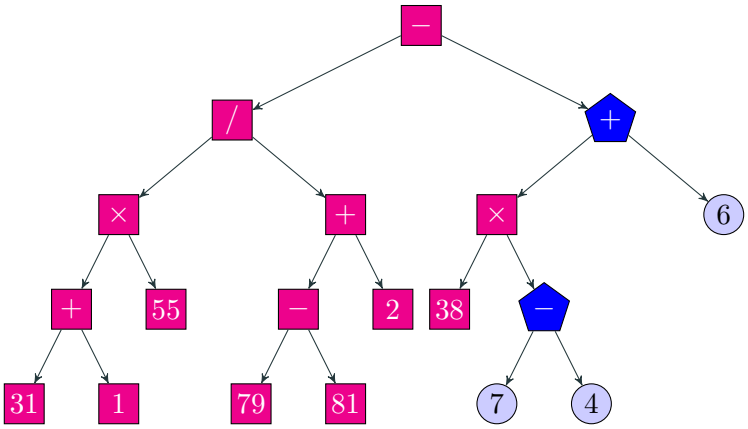
$$31 + 1 \times 55/79 - 81 + 2 - 38$$

Inorder traversal on an expression tree



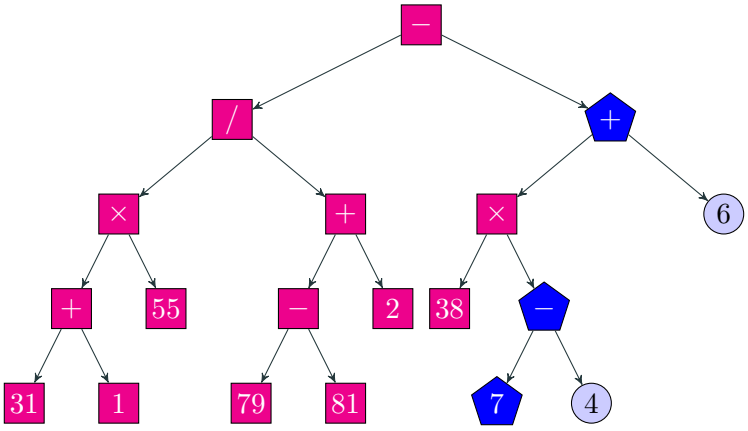
$$31 + 1 \times 55/79 - 81 + 2 - 38 \times$$

Inorder traversal on an expression tree



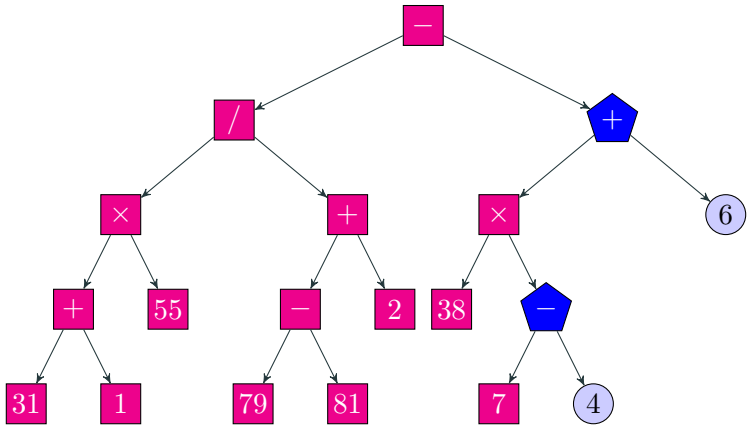
$$31 + 1 \times 55/79 - 81 + 2 - 38 \times$$

Inorder traversal on an expression tree



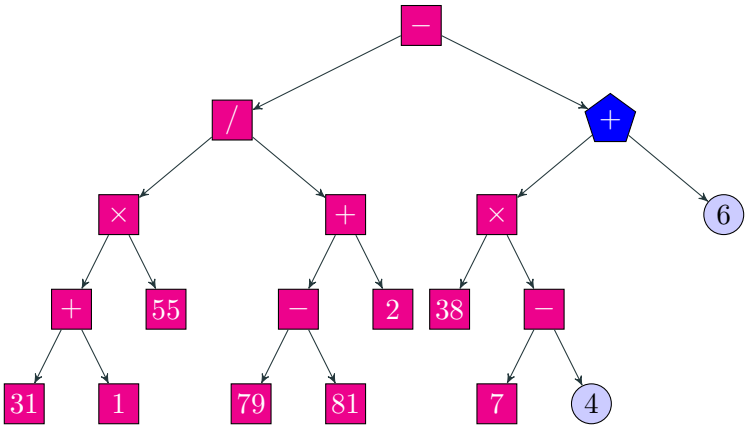
$$31 + 1 \times 55/79 - 81 + 2 - 38 \times$$

Inorder traversal on an expression tree



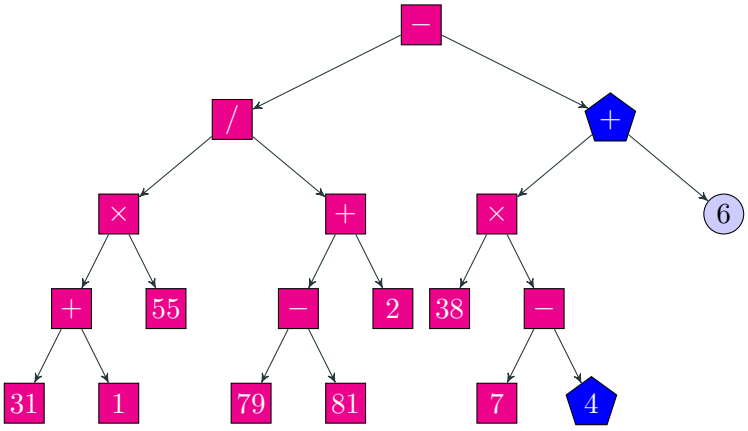
$$31 + 1 \times 55/79 - 81 + 2 - 38 \times 7$$

Inorder traversal on an expression tree



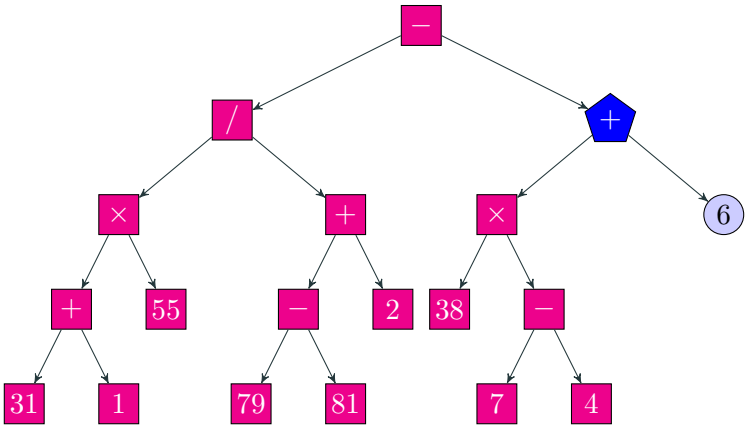
$$31 + 1 \times 55/79 - 81 + 2 - 38 \times 7 -$$

Inorder traversal on an expression tree



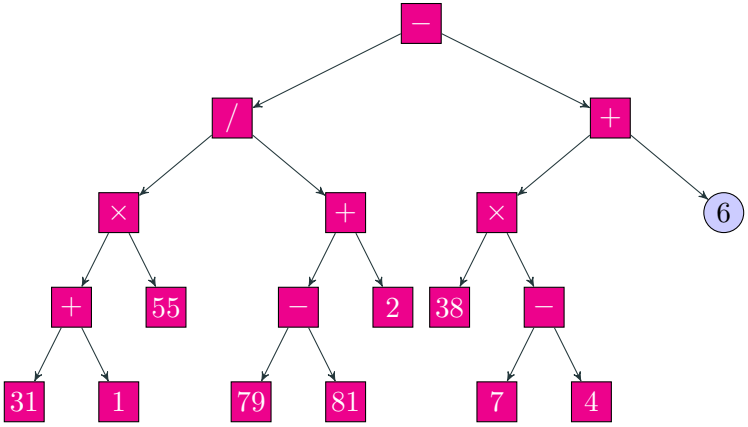
$$31 + 1 \times 55/79 - 81 + 2 - 38 \times 7 -$$

Inorder traversal on an expression tree



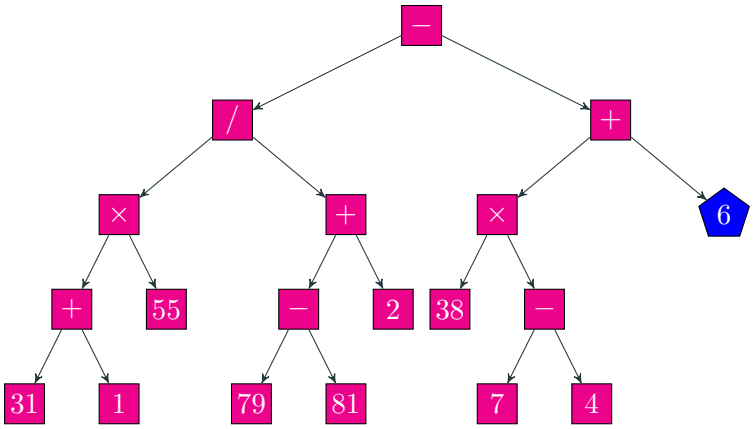
$$31 + 1 \times 55/79 - 81 + 2 - 38 \times 7 - 4$$

Inorder traversal on an expression tree



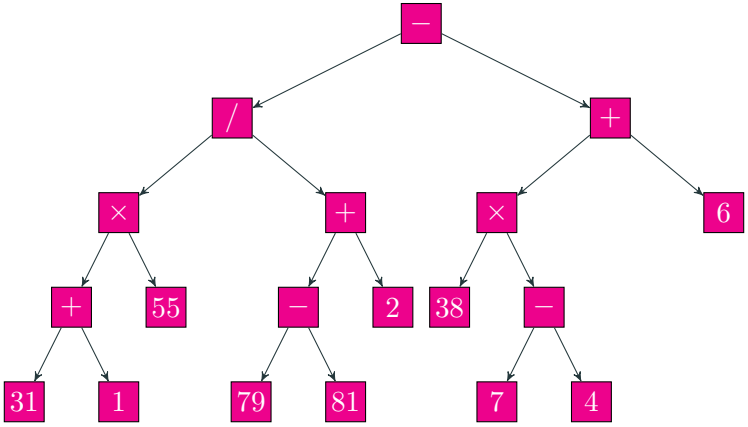
$$31 + 1 \times 55/79 - 81 + 2 - 38 \times 7 - 4 +$$

Inorder traversal on an expression tree



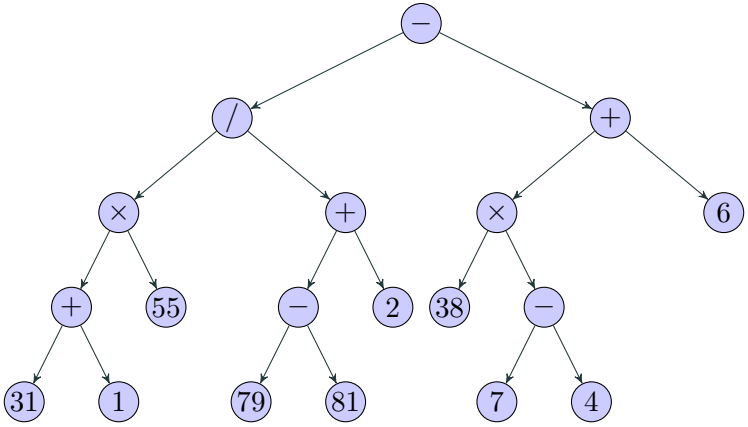
$$31 + 1 \times 55/79 - 81 + 2 - 38 \times 7 - 4 +$$

Inorder traversal on an expression tree



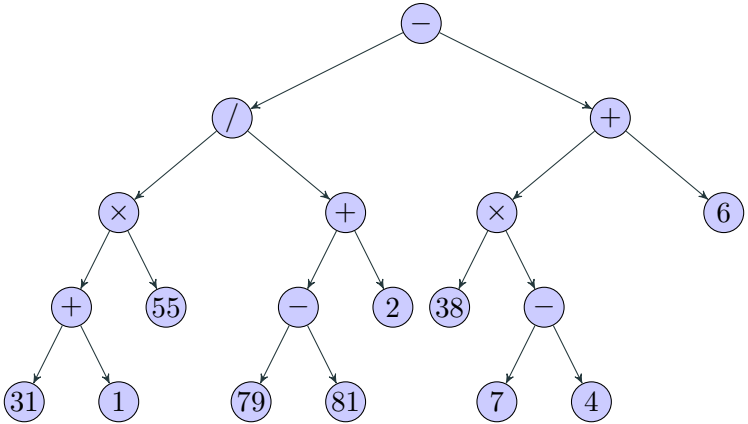
$$31 + 1 \times 55/79 - 81 + 2 - 38 \times 7 - 4 + 6$$

Preorder traversal: node, left subtree, right subtree



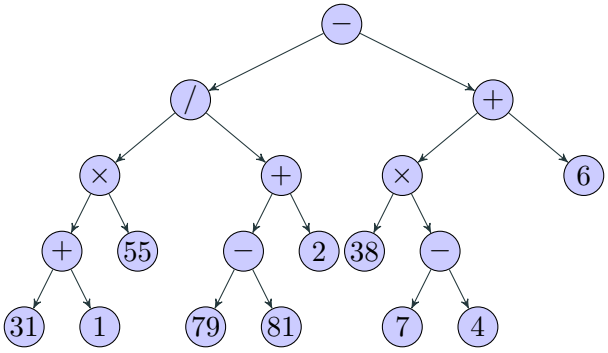
$$- \; / \; \times \; + \; 31 \; 1 \; 55 \; + \; - \; 79 \; 81 \; 2 \; + \; \times \; 38 \; - \; 7 \; 4 \; 6$$

Postorder traversal: left subtree, right subtree, node



$$31\ 1\ +\ 55\ \times\ 79\ 81\ -\ 2\ +\ /\ 38\ 7\ 4\ -\ \times\ 6\ +\ -$$

Three traversals together

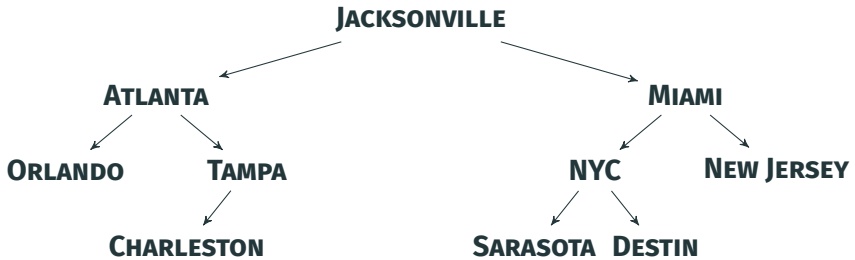


Inorder: $31 + 1 \times 55 / 79 - 81 + 2 - 38 \times 7 - 4 + 6$

Preorder: $- / \times + 31\ 1\ 55 + - 79\ 81\ 2 + \times 38 - 7\ 4\ 6$

Postorder: $31\ 1 + 55 \times 79\ 81 - 2 + / 38\ 7\ 4 - \times 6 + -$

Another example



Preorder Traversal

Jacksonville, Atlanta, Orlando, Tampa, Charleston, Miami, NYC, Sarasota, Destin, New Jersey

Inorder Traversal

Orlando, Atlanta, Charleston, Tampa, Jacksonville, Sarasota, NYC, Destin, Miami, New Jersey

Postorder Traversal

Orlando, Charleston, Tampa, Atlanta, Sarasota, Destin, NYC, New Jersey, Miami, Jacksonville

From traversal sequences to trees

- It is natural to ask the following question: *can we construct back the tree **uniquely** when we are given one or more its traversals?*
- If exactly one traversal is given, the answer is **NO**
- If both the preorder and postorder traversals are given, the answer is **NO**
- If the inorder + postorder/preorder traversals are given, the answer is **YES**

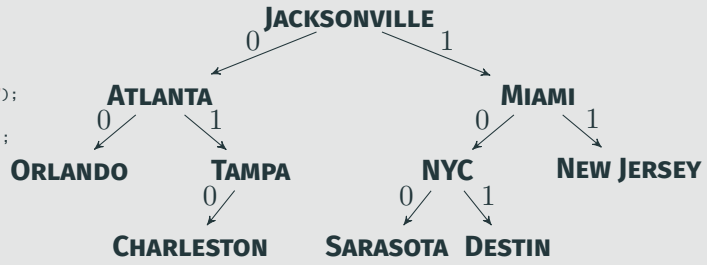
Recommended excercise

Given an inorder + postorder/preorder traversals, recursively construct the corresponding binary tree

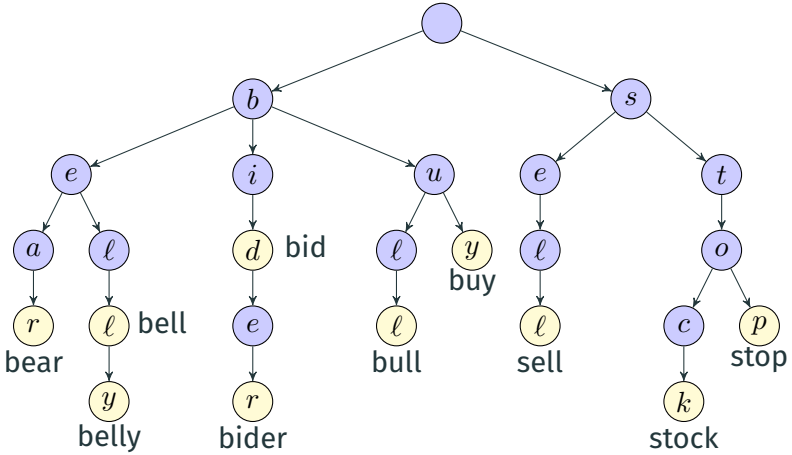
See the class `BinaryTree<E>`

Code

```
public class TestBinaryTree {
   public static void main(String[] args) {
      BinaryTree<String> cityTree = new BinaryTree<>();

      cityTree.insert("Jacksonville","root");
      cityTree.insert("Atlanta", "0");
      cityTree.insert("Miami", "1");
      cityTree.insert("Orlando", "00");
      cityTree.insert("Tampa", "01");
      cityTree.insert("Charleston", "010");
      cityTree.insert("NYC", "10");
      cityTree.insert("New Jersey", "11");
      cityTree.insert("Sarasota", "100");
      cityTree.insert("Destin", "101");

      System.out.print("PreOrder: ");
      cityTree.printPreOrder();
      System.out.print("\nInOrder: ");
      cityTree.printInOrder();
      System.out.print("\nPostOrder: ");
      cityTree.printPostOrder();

      System.out.print("\nNumber of nodes in the tree: " + cityTree.countNodes( cityTree.getRoot() ) );
   }
}
```



Tries: how do they look like?



$$C = \{a, b, \ldots, z\}, \qquad m = 26,$$

$$S = \{\text{bear, bell, belly, bid, bider, bull, buy, sell, stock, stop}\}$$

Tries

Definition

- Consider a set C of m characters $c_1, c_2, \ldots, c_m$ and a set S of strings made up of characters taken only from C (characters may repeat in the strings)
- A **trie** T on C is a m-ary tree (every node has at most m children) such that
  - Each node of T, except the root node, is labeled with a character of C.
  - The children of any node have distinct labels, taken from C.
  - The root has an empty label.
  - Each node t in the tree is associated with a string s, obtained by concatenating the labels of the nodes from the root to that node. If $s \in S$, then store s with t, else store a `null`.

☞ If $m = 2$, the resulting trie is a binary tree

Node structure for tries when $m = 26$

```
private static class TrieNode {
   String word = null;
   TrieNode parent = null;
   TrieNode[] children = new TrieNode[26]; // every cell contains a 'null' by default
}
```

An array of children references (some possibly empty)

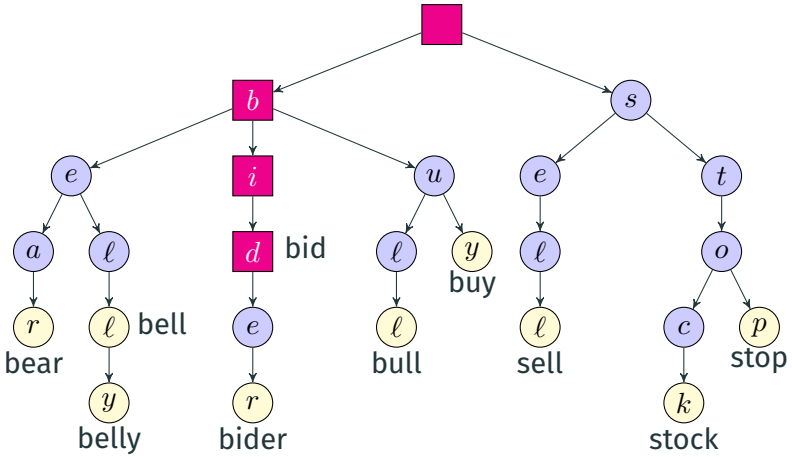
How to find the child id corresponding to a character in our case?

```
int childID = s.charAt(i) - 'a'; // childID will be an integer between 0 and 25
```

☞ *Note that this simple relation may not work if C contains other types of characters. In that case, some kind of mapping must be used for efficiency (stay tuned for maps!).*

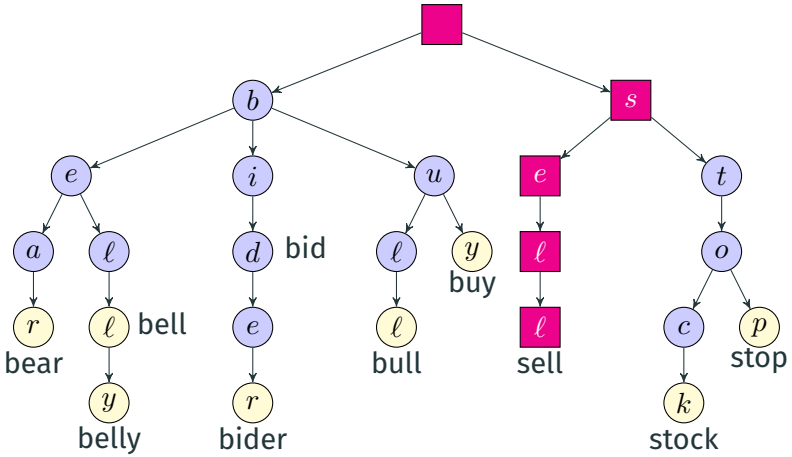
Searching

Searching the string ‘bid’



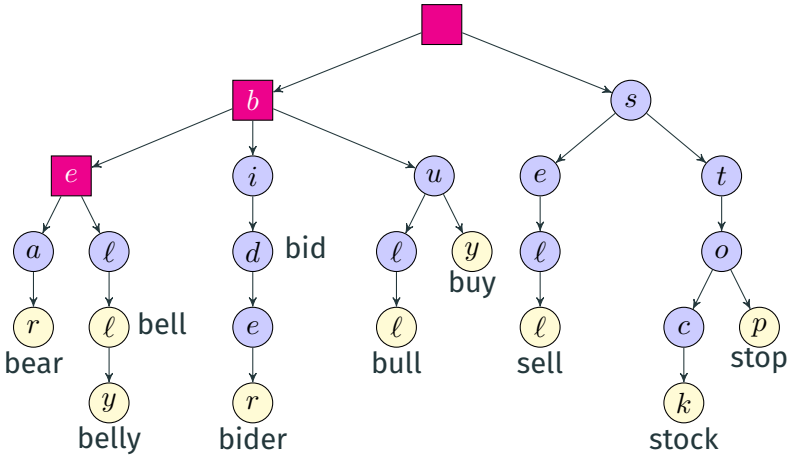
Found!

Searching the string ‘sell’



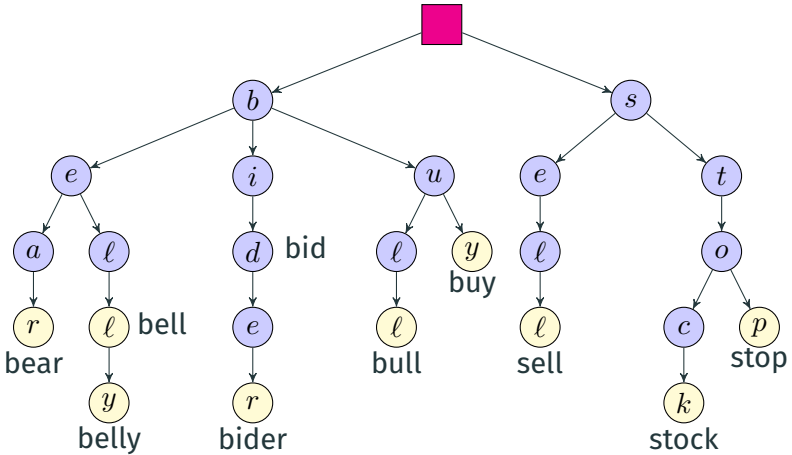
Found!

Searching the string ‘best’



`Not found!`

Searching the string ‘donkey’



`Not found!`

Searching time

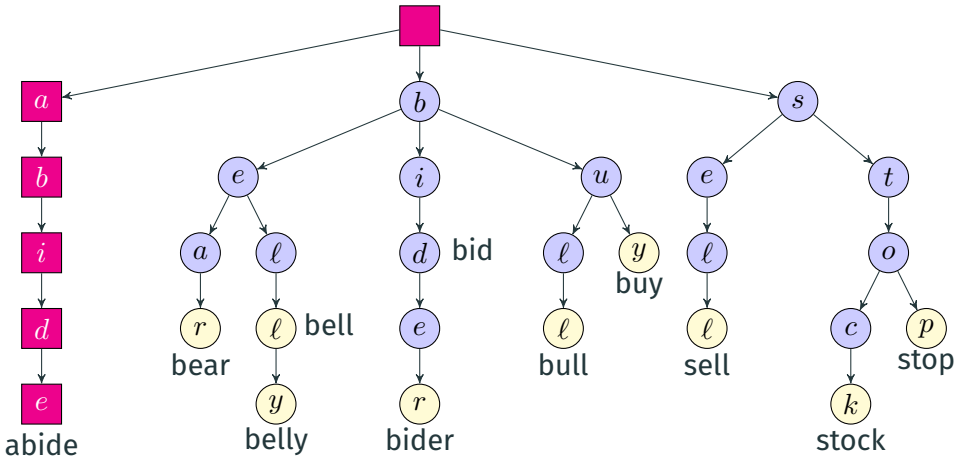
- Let n be the length of the string that needs to be searched
- In the worst case, we need to visit $n + 1$ nodes (including the root node); less than $n + 1$ nodes will be visited if the string is not present in the trie
- At every node, we need to spend $O(m)$ time to figure out next child in the search path
- At the last node, we check if the a word is stored over there or not; takes $O(1)$ time for verification
- **Total time taken.** $(n + 1) \cdot O(m) + O(1) = O(nm)$
- In our case (while playing with English letters only), it takes $O(1)$ time to figure out the next child in the search path using the following mapping formula and hence, search time is $O(n)$

```
int childID = s.charAt(i) - 'a'; // obtain the child id
```

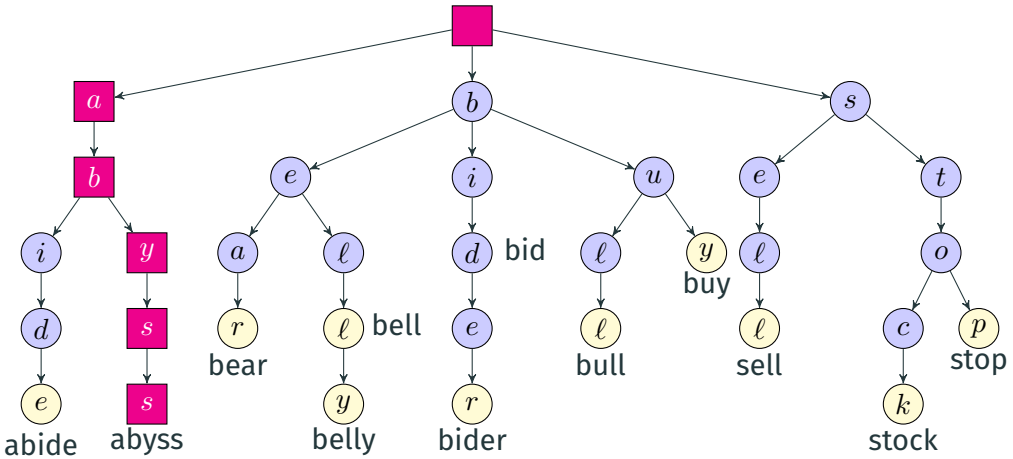
☞ *Tries can be used where fast lookups are required quite often*

Insertion

Inserting the string ‘abide’



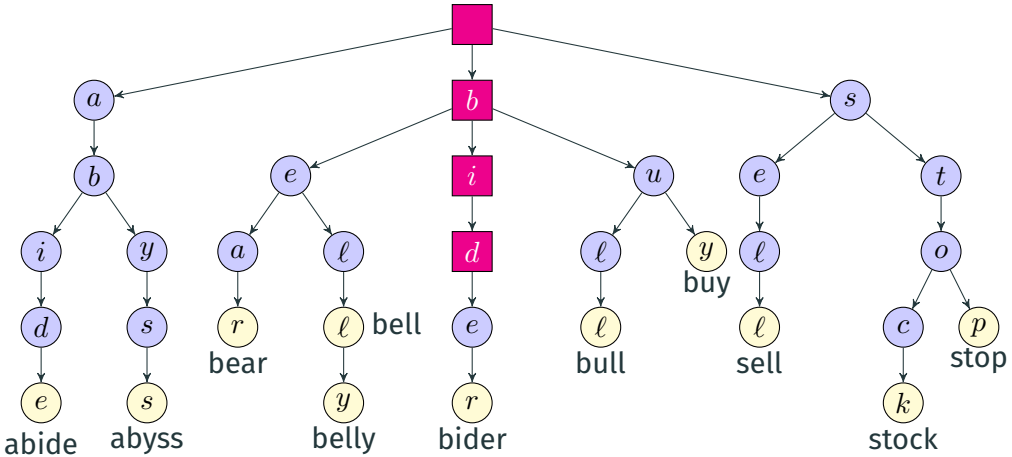
Inserting the string ‘abyss’



Takes $O(nm)$ time, similar to the search operation

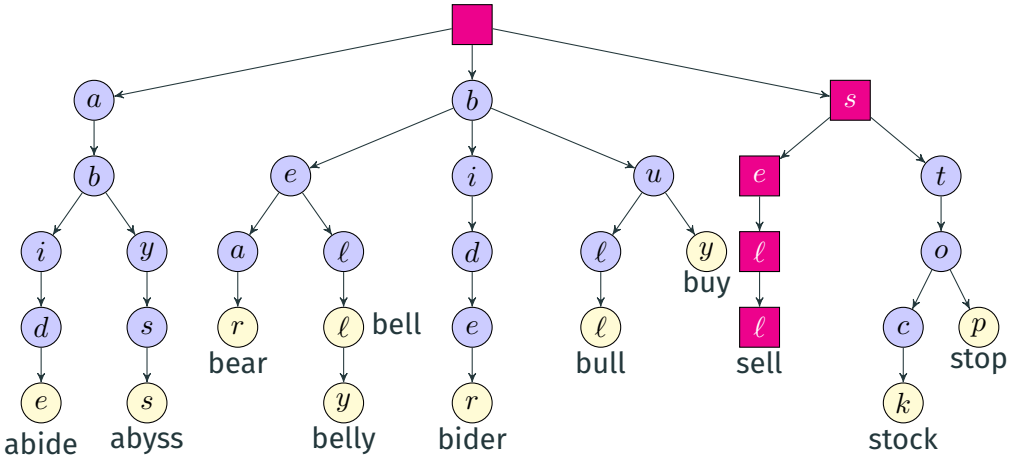
Deletion

Deleting the string ‘bid’



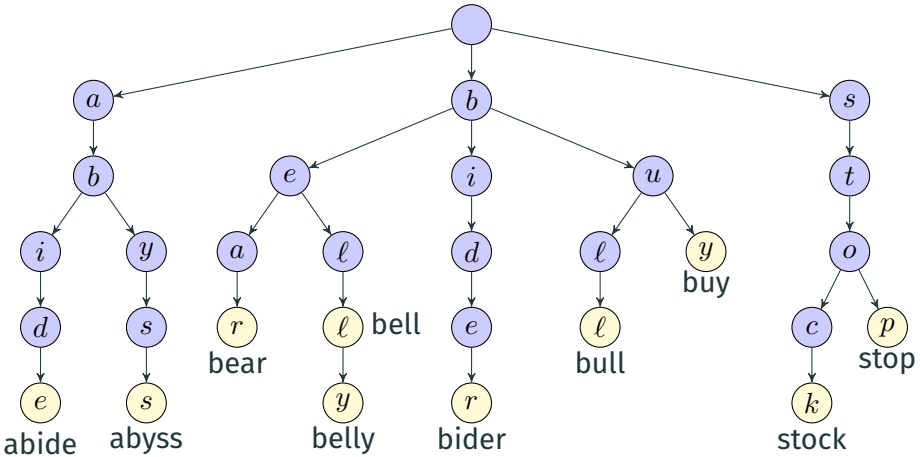
Remove the word ‘bid’ stored at the node d

Deleting the string ‘sell’



After detecting ‘sell’, delete the three nodes ℓ, ℓ, e

Deleting the string ‘sell’



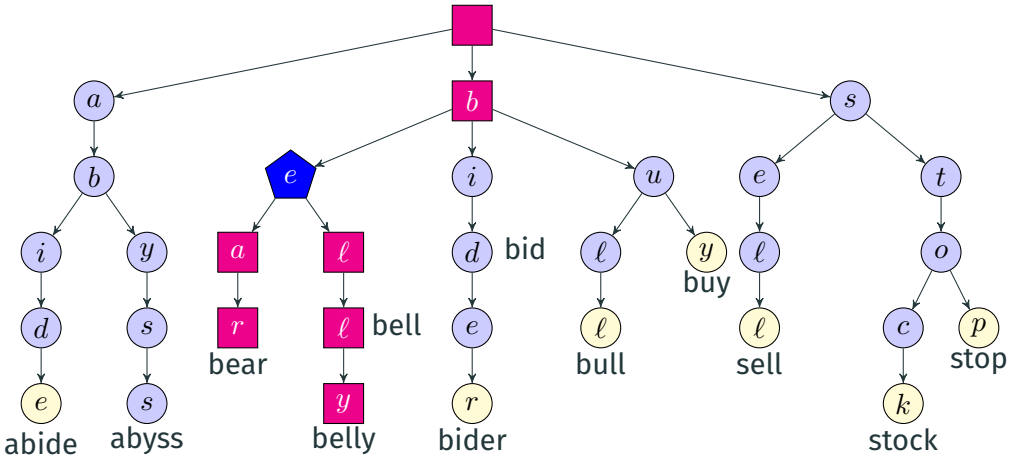
The tree after deleting ‘sell’

Deletion time

- Locating the word in the tree takes $O(nm)$ time by climbing down
- Then, we need to climb up for deleting the useless nodes; this takes $O(nm)$ time as well
- **Total time taken.** $O(nm) + O(nm) = 2 \cdot O(nm) = O(nm)$

Report the words with a given prefix

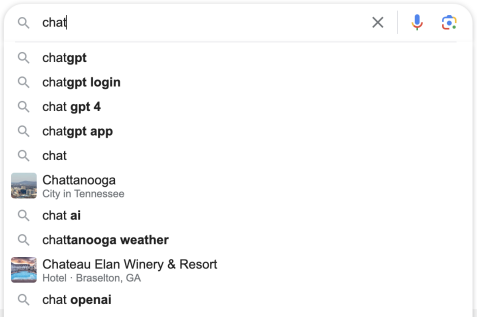
Find the words with prefix ‘be’



Report every word present in the subtree rooted at the last character of the prefix; in this example the subtree of interest is rooted at e (the pentagon)

See the class `Trie`

Auto-completion: an application of tries



To find the auto-complete choices, one can use a pre-built trie and easily find the entries which start with the string typed so far by running the `wordsHavingPrefix(...)` method on the trie

Reading

Chaper 12 from

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/index.html