Binary Trees

CMPT 125
Mo Chen
SFU Computing Science
18/3/2020

Lecture 27

Today:

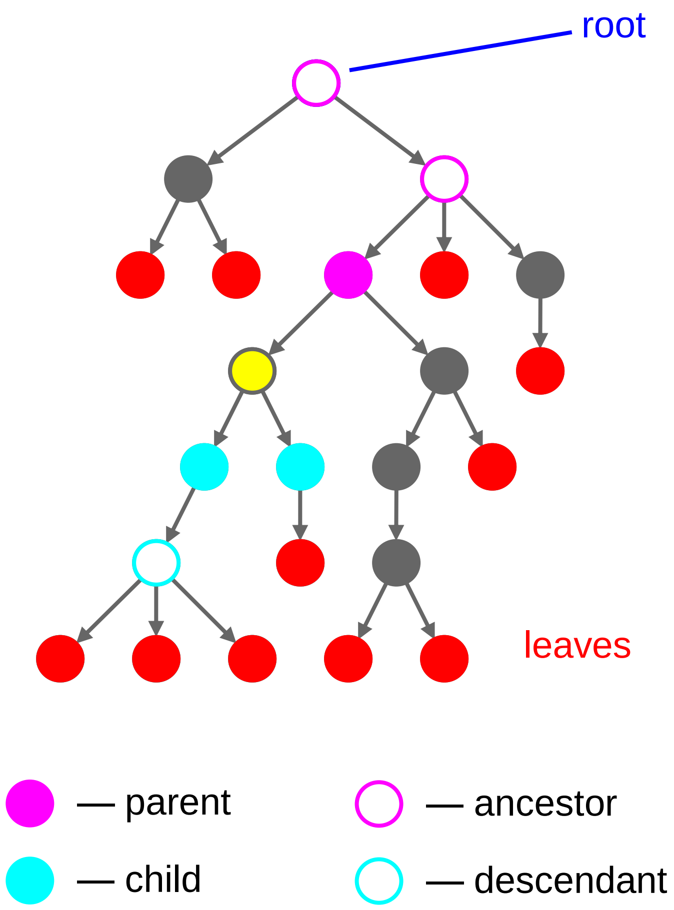
- Binary Trees
- Recursive Definitions of Trees
- Binary Tree Implementation
- Expression Trees
- Traversals
- Grammars

Rooted Trees (Review)

A *rooted tree* is a tree where all but one vertex has exactly one inbound edge (from its *parent*).

- usually drawn by level, top down
- *root* vertex has no inbound edge
- *leaf* vertex has no outbound edge
- parents point to *children*
- *ancestors* point to *descendants* via a downward path

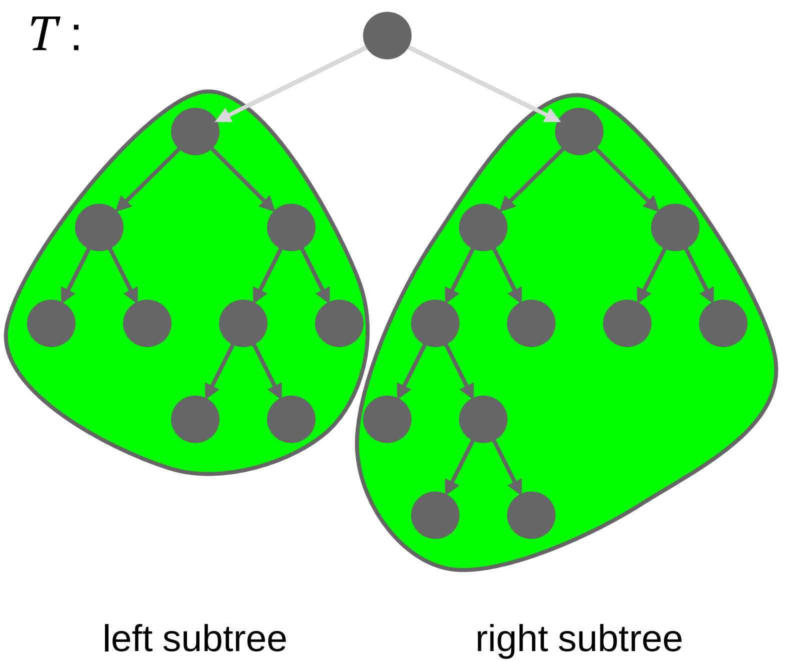
A *binary tree* is a rooted tree in which no vertex has more than 2 children.



Subtrees and Recursive Definitions

There are many *subtrees* within a binary tree:

- the two most important are the left and right subtrees
- rooted at the left and right children of the root
- to visualize, remove the root!



Leads to a recursive definition:

T is a binary tree when:

- T is an empty tree (i.e., no vertices)

  OR . . .
- T has a root vertex whose left and right subtrees are binary trees

Subtrees and Recursive Definitions

There are many *subtrees* within a binary tree:

- the two most important are the left and right subtrees
- rooted at the left and right children of the root
- to visualize, remove the root!

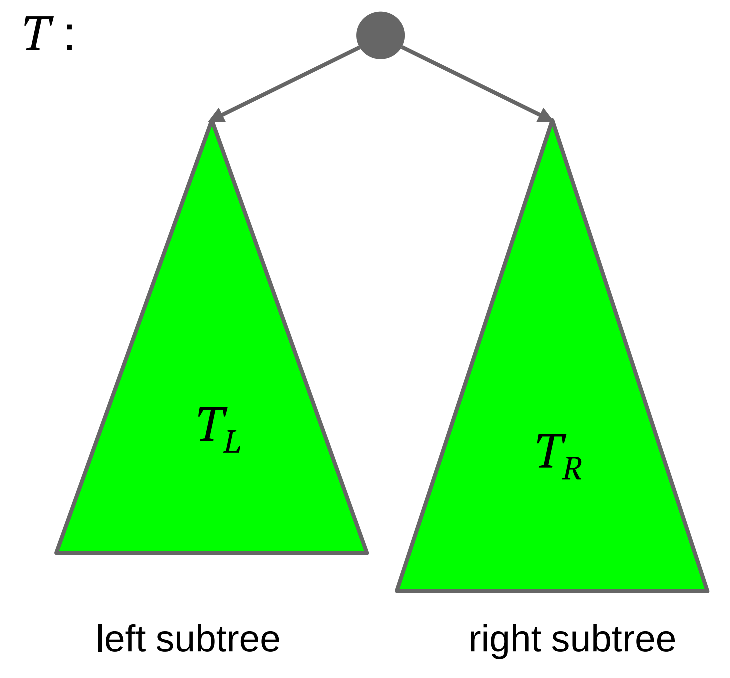
Leads to a recursive definition:

T is a binary tree when:

- T is an empty tree (i.e., no vertices)

  OR . . .

- T has a root vertex whose left and right subtrees are binary trees



Trees and Recursion

Recursive definitions benefit you in two ways:

- allow you to write recursive algorithms
- allow you to reason about the structure by recursion (or induction)

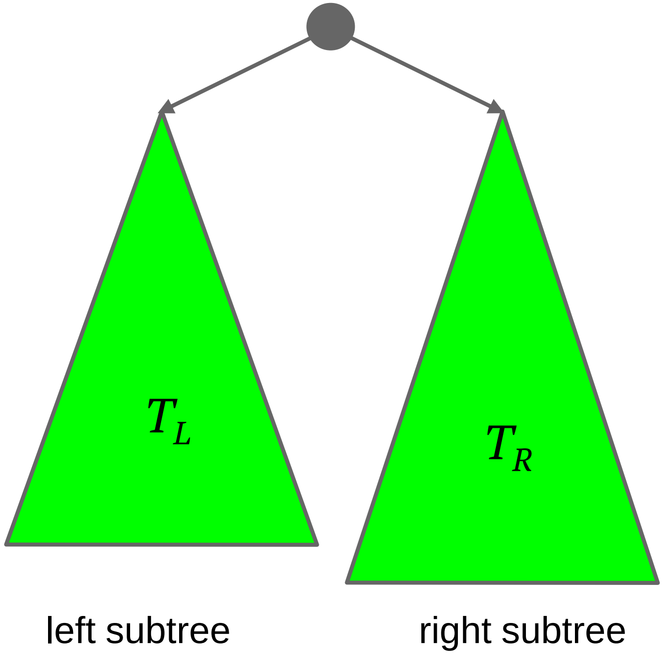
How to build a binary tree in C++?

- use the recursive definition
- adopt similar strategy to a linked list

```
struct LLnode {
    int data;
    struct LLnode * next;
};
```

T is a binary tree when either:

- T is an empty tree
- T has a root vertex whose left and right subtrees are binary trees



Trees and Recursion

Recursive definitions benefit you in two ways:

- allow you to write recursive algorithms
- allow you to reason about the structure by recursion (or induction)

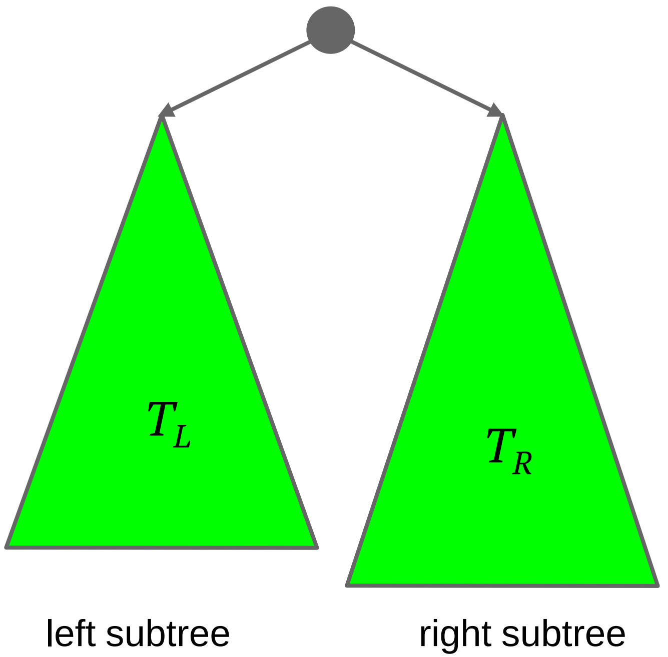
How to build a binary tree in C++?

- use the recursive definition
- adopt similar strategy to a linked list

```
struct BTnode {
    int data;
    struct BTnode * next;
};
```

T is a binary tree when either:

- T is an empty tree
- T has a root vertex whose left and right subtrees are binary trees



Trees and Recursion

Recursive definitions benefit you in two ways:

- allow you to write recursive algorithms
- allow you to reason about the structure by recursion (or induction)

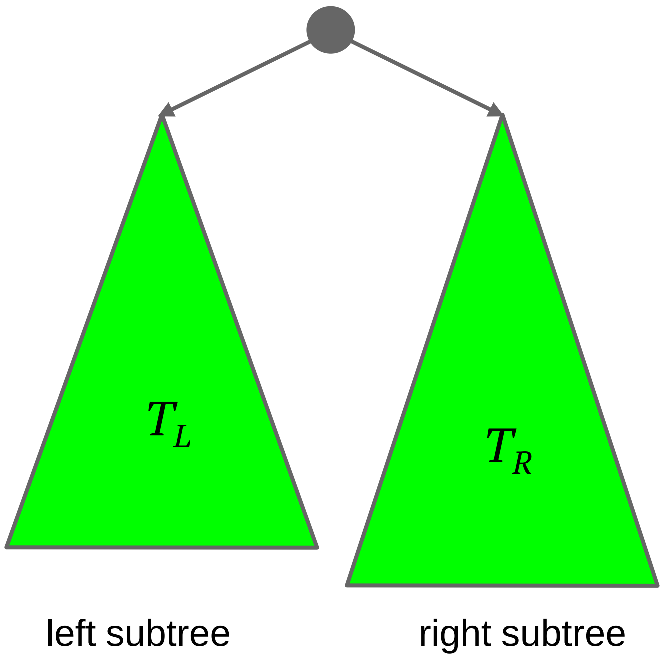
How to build a binary tree in C++?

- use the recursive definition
- adopt similar strategy to a linked list

```
struct BTnode {
    int data;
    struct BTnode * left;
    struct BTnode * right;
};
```

T is a binary tree when either:

- T is an empty tree
- T has a root vertex whose left and right subtrees are binary trees



Trees and Recursion

Recursive definitions benefit you in two ways:

- allow you to write recursive algorithms
- allow you to reason about the structure by recursion (or induction)

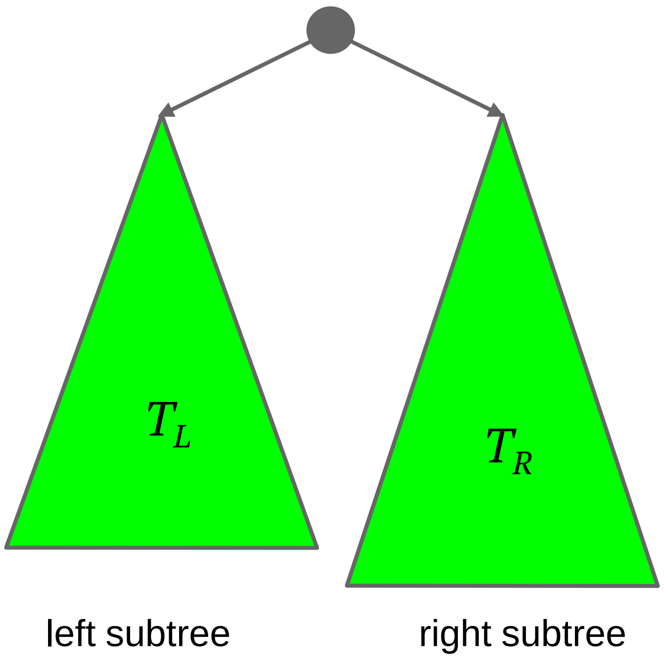
How to build a binary tree in C++?

- use the recursive definition
- adopt similar strategy to a linked list

```
struct BTnode {
    int data;
    struct BTnode * left;
    struct BTnode * right;
    struct BTnode * parent;
};
```

T is a binary tree when either:

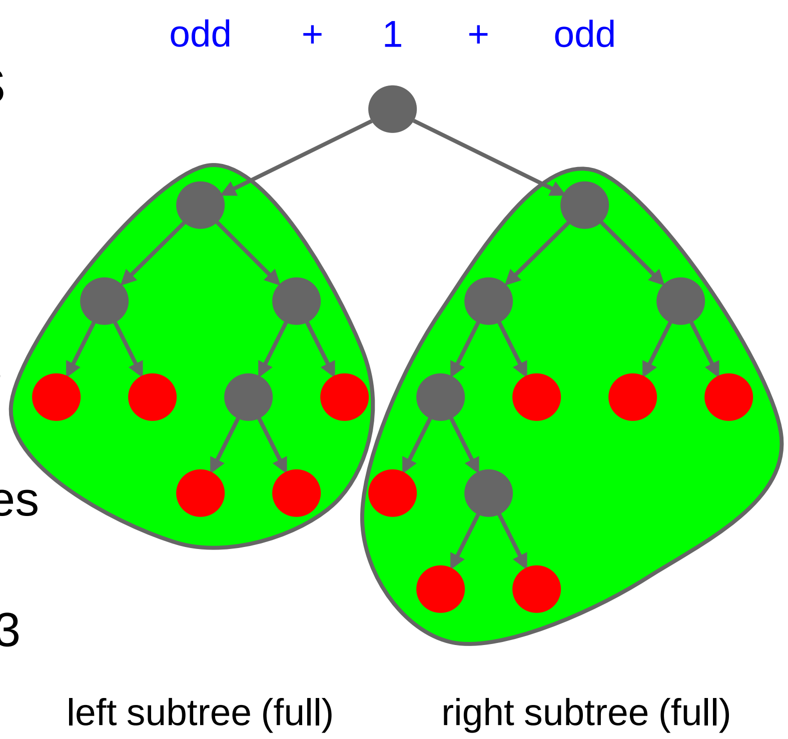
- T is an empty tree
- T has a root vertex whose left and right subtrees are binary trees



Reasoning about Trees

A *full binary tree* is a non-empty binary tree, where each vertex has exactly 0 or 2 children.

Theorem: A full binary tree always has an odd number of vertices.



Proof by induction:

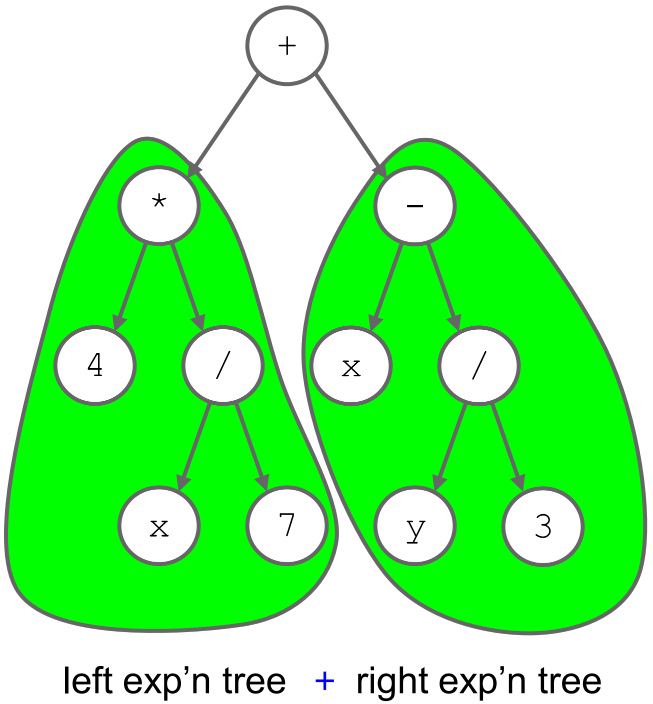
- If the root has 0 children, then the tree has only one vertex, which is odd.
- If the root has 2 children, then their subtrees must also be full, and by induction, odd. The total number of vertices is the sum of 3 odd numbers, which is odd.

Expression trees are full.

Expression Trees

An *expression tree* is a full binary tree that represents an arithmetic calculation:

- internal nodes are binary operators
- leaves are numbers



Expression Trees and Postfix

An *expression tree* is a full binary tree that represents an arithmetic calculation:

- internal nodes are binary operators
- leaves are numbers

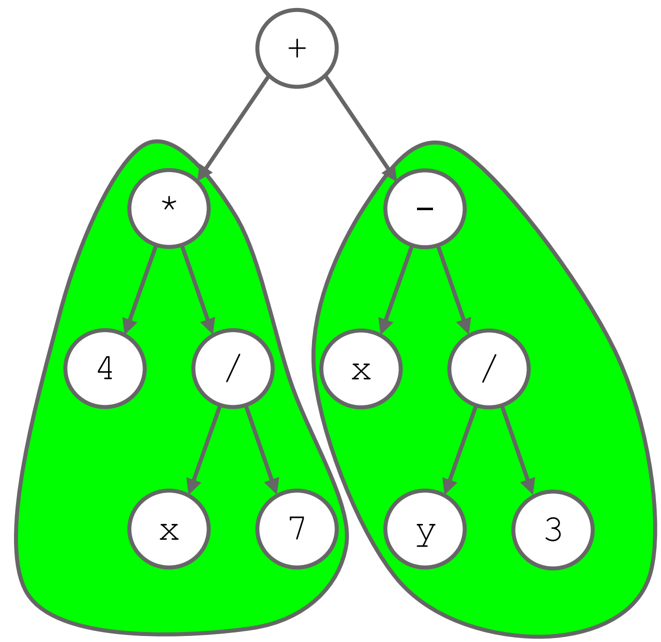
Thus, postfix expressions can be defined recursively, too.

E is a *postfix expression* when:

- E is a number, OR . . .
- E is two postfix expressions followed by a binary operator ($+$, $-$, $*$, $/$)

Algorithm to evaluate expression tree:

- bottom up



left exp'n tree right exp'n tree +

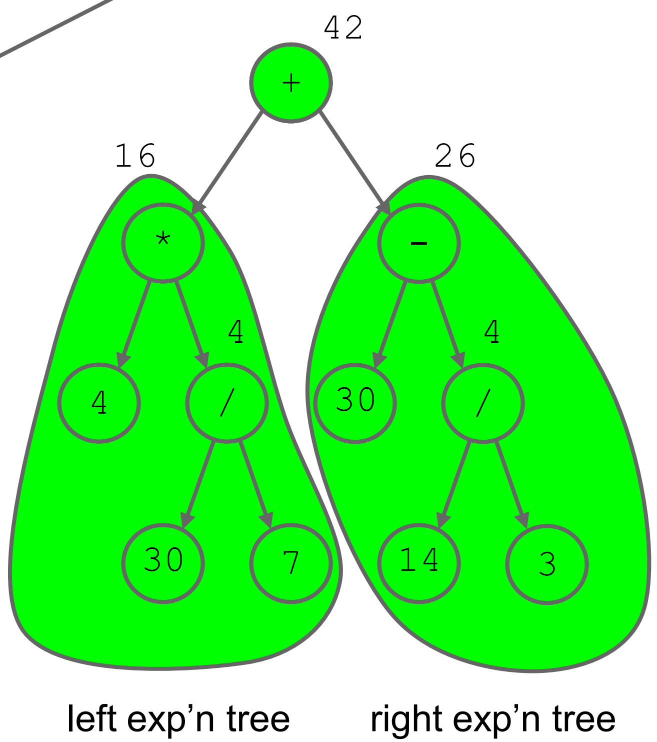
`4 x 7 / *` `x y 3 / -` +

1st operand 2nd operand

Tree Evaluation: Traversals

Algorithm to evaluate a tree rooted at vertex x: *recursion*

- If x has a number, then return that number
- If x has an operator, then:
    - evaluate the left subtree
    - evaluate the right subtree
    - return (left *op* right)



Known as a *post-order traversal*

- evaluate the children first, then yourself
- follows the order: left → right → self

Other common traversals:

- self → left → right: *pre-order*
- left → self → right: *in-order*

Evaluated: `4 30 7 / * 30 14 3 / - +`

Stack-Based Postfix Calculator

Use a Stack ADT to evaluate postfix.

Algorithm:

```
Create an empty stack S
while there is still input {
    if next input token is a number
        push the number to S
    if next input token is an operator {
        pop from S → b
        pop from S → a
        push (a op b) to S
    }
}
pop from S → result
```

Stack-Based Postfix Calculator

Use a Stack ADT to evaluate postfix.

Algorithm:

```
Create an empty stack S
while there is still input {
    if next input token is a number
        push the number to S
    if next input token is an operator {
        pop from S → b
        pop fro
        push (a
    }
}
pop from S → result
```

If any pop fails, then it’s invalid postfix.

If S ends nonempty then it’s invalid postfix.

Building Expression Trees from Postfix

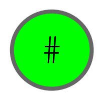
Adapt postfix calculator algorithm to build trees from postfix.

Algorithm:

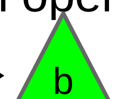
Create an empty stack S

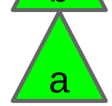
while there is still input {

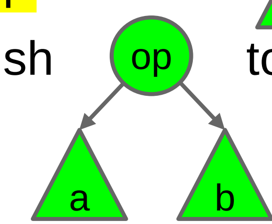
if next input token is a number

push  to S

if next input token is an operator {

pop from S → 

pop from S → 

push  to S

}

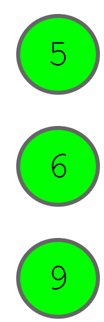
}

pop from S → result

Example:

9 6 5 + 6 9 − * −

S:



Building Expression Trees from Postfix

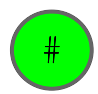
Adapt postfix calculator algorithm to build trees from postfix.

Algorithm:

Create an empty stack S

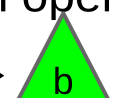
while there is still input {

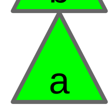
if next input token is a number

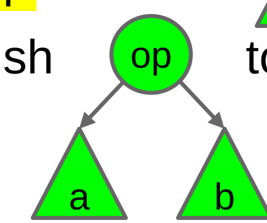
push 

to S

if next input token is an operator {

pop from S → 

pop from S → 

push 

to S

}

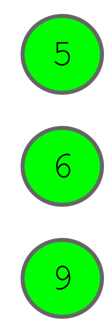
}

pop from S → result

Example:

9 6 5 + 6 9 − * −

S:



Building Expression Trees from Postfix

Adapt postfix calculator algorithm to build trees from postfix.

Algorithm:

Create an empty stack S

while there is still input {

if next input token is a number

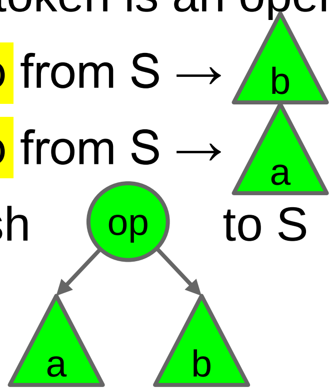
push # to S

if next input token is an operator {

pop from S → b

pop from S → a

push op to S



}

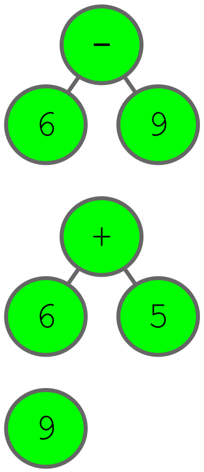
}

pop from S → result

Example:

9 6 5 + 6 9 − * −

S:



Building Expression Trees from Postfix

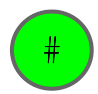
Adapt postfix calculator algorithm to build trees from postfix.

Algorithm:

Create an empty stack S

while there is still input {

if next input token is a number

push 

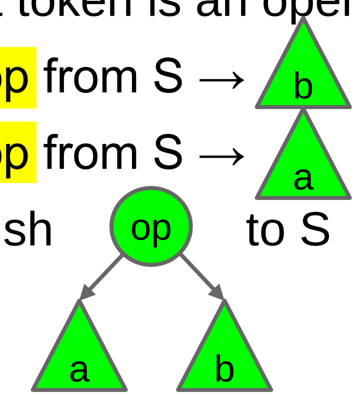
to S

if next input token is an operator {

pop from S →

pop from S →

push op to S



}

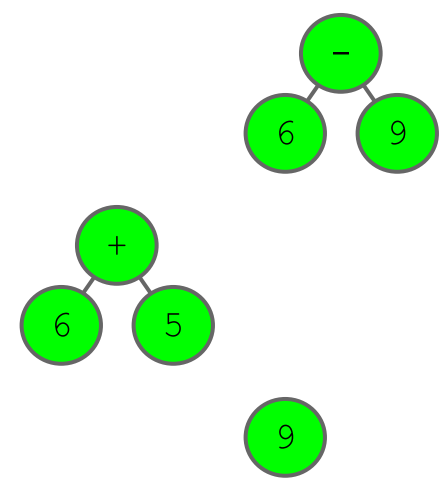
}

pop from S → result

Example:

9 6 5 + 6 9 – * –

S:



Building Expression Trees from Postfix

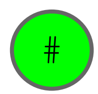
Adapt postfix calculator algorithm to build trees from postfix.

Algorithm:

Create an empty stack S

while there is still input {

if next input token is a number

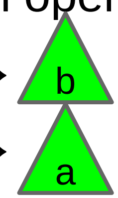
push 

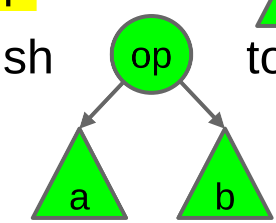
to S

if next input token is an operator {

pop from S →

pop from S →



push 

to S

}

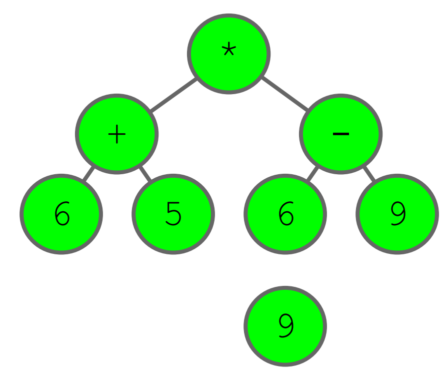
}

pop from S → result

Example:

9	6	5	+	6	9	–	*	–

S:



Building Expression Trees from Postfix

Adapt postfix calculator algorithm to build trees from postfix.

Algorithm:

Create an empty stack S
while there is still input {
    if next input token is a number
        push # to S
    if next input token is an operator {
        pop from S → b
        pop from S → a
        push op (a, b) to S
    }
}
pop from S → result

Example:

9 6 5 + 6 9 − * −

S: