ساختمان داده و الگوریتم ها

مبحث دوازدهم:
درخت

سجاد شیرعلی شهرضا

پاییز 1402

دوشنبه، 22 آبان 1402

Based on Washington University CSE 373 (Autumn 2020 by Hunter Schafer)

اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 10.4

درخت

Data Structures

- Data structure
  - Organization or format for storing or managing data
  - Concrete realization of an abstract data type
- Operations
  - Always a tradeoff: some operations more efficient, some less, for any data structure
  - Choose efficient data structure for operations of concern

Example Data Structures

Data Structure	add(val v)	get(int i)
Array [2 | 1 | 3 | 0]		
Linked List 2 → 1 → 3 → 0		

```
add(v): append v
get(i): return element at position i
```

Example Data Structures

Data Structure	add(val v)	get(int i)
Array 2 1 3 0	$O(n)$	
Linked List 2 → 1 → 3 → 0		

```
add(v): append v
get(i): return element at position i
```

Example Data Structures

Data Structure	add(val v)	get(int i)
Array $\boxed{2}\boxed{1}\boxed{3}\boxed{0}$	$O(n)$	
Linked List $2 \to 1 \to 3 \to 0$	$O(1)$	

```
add(v): append v
get(i): return element at position i
```

Example Data Structures

Data Structure	add(val v)	get(int i)
Array [2 | 1 | 3 | 0]	$O(n)$	$O(1)$
Linked List (2) → (1) → (3) → (0)	$O(1)$	

```
add(v): append v
get(i): return element at position i
```

Example Data Structures

Data Structure	add(val v)	get(int i)
Array [2, 1, 3, 0]	$O(n)$	$O(1)$
Linked List 2 → 1 → 3 → 0	$O(1)$	$O(n)$

```
add(v): append v
get(i): return element at position i
```

Example Data Structures

Data Structure	add(val v)	get(int i)	contains(val v)
Array [2 | 1 | 3 | 0]	$O(n)$	$O(1)$	
Linked List 2 → 1 → 3 → 0	$O(1)$	$O(n)$	

```
add(v): append v
get(i): return element at position i
contains(v): return true if contains v
```

Example Data Structures

Data Structure	add(val v)	get(int i)	contains(val v)
Array 2 1 3 0	$O(n)$	$O(1)$	$O(n)$
Linked List 2 → 1 → 3 → 0	$O(1)$	$O(n)$	

```
add(v): append v
get(i): return element at position i
contains(v): return true if contains v
```

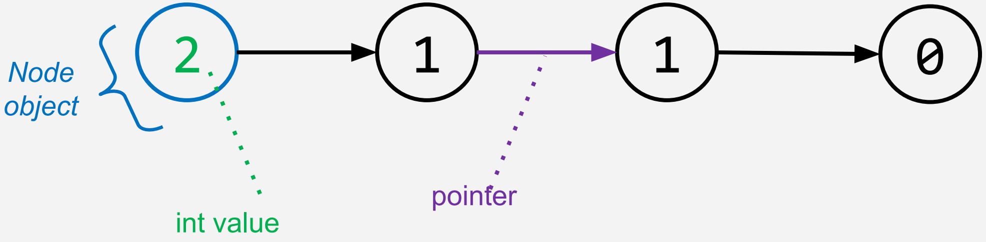
Example Data Structures

Data Structure	add(val v)	get(int i)	contains(val v)
Array | 2 | 1 | 3 | 0 |	$O(n)$	$O(1)$	$O(n)$
Linked List 2 → 1 → 3 → 0	$O(1)$	$O(n)$	$O(n)$

```
add(v): append v
get(i): return element at position i
contains(v): return true if contains v
```

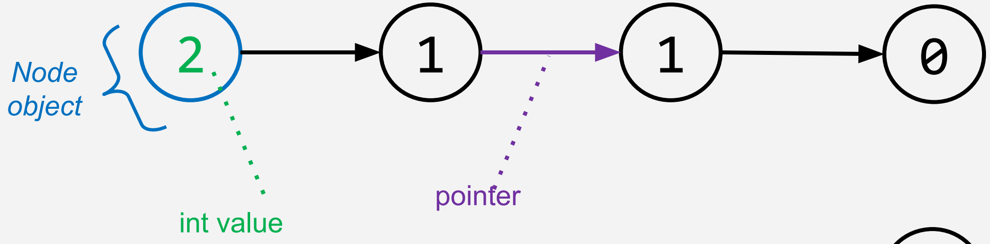
Linked List

Singly linked list:

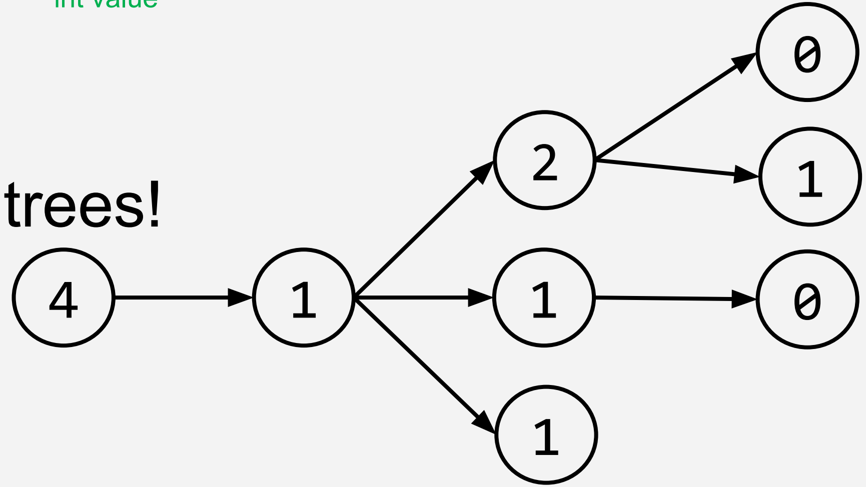


Generalized Linked List (i.e., Tree)

Singly linked list:

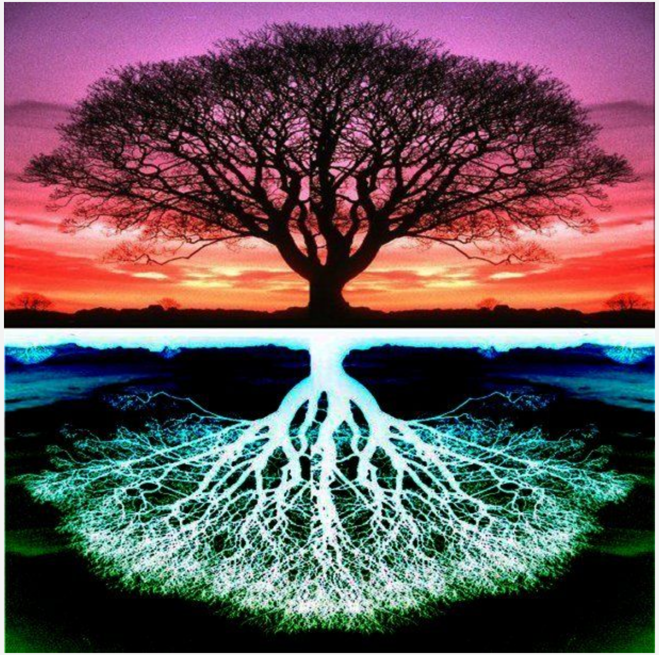


Today: trees!



Tree

- In CS, we draw trees “upside down”



Tree Overview

- Tree:
  - Data structure with nodes
  - Similar to linked list

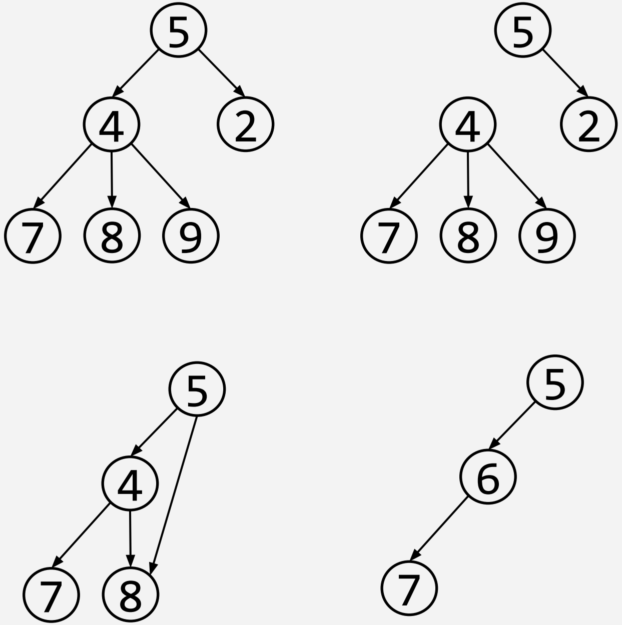
Tree Overview

- Tree:
  - Data structure with nodes
  - Similar to linked list
- Nodes:
  - Zero or more successors (children)
  - Exactly one predecessor (parent)
    - Except the root, which has none
- All nodes are reachable from root

Tree Overview

- Tree:
  - Data structure with nodes
  - Similar to linked list
- Nodes:
  - Zero or more successors (children)
  - Exactly one predecessor (parent)
    - Except the root, which has none
- All nodes are reachable from root

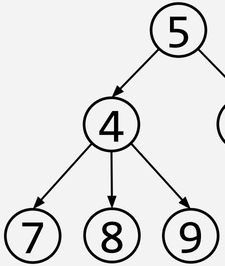
A tree or not a tree?

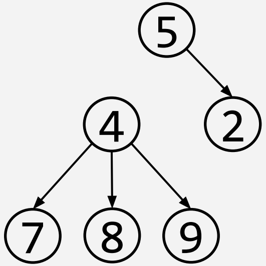


Tree Overview

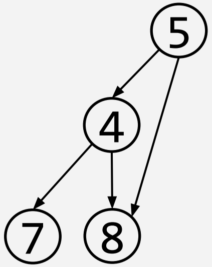
- Tree:
  - Data structure with nodes
  - Similar to linked list
- Nodes:
  - Zero or more successors (children)
  - Exactly one predecessor (parent)
    - Except the root, which has none
- All nodes are reachable from root

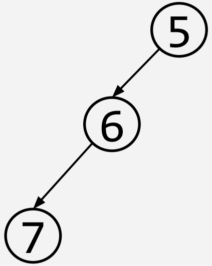
A tree or not a tree?





✅A tree

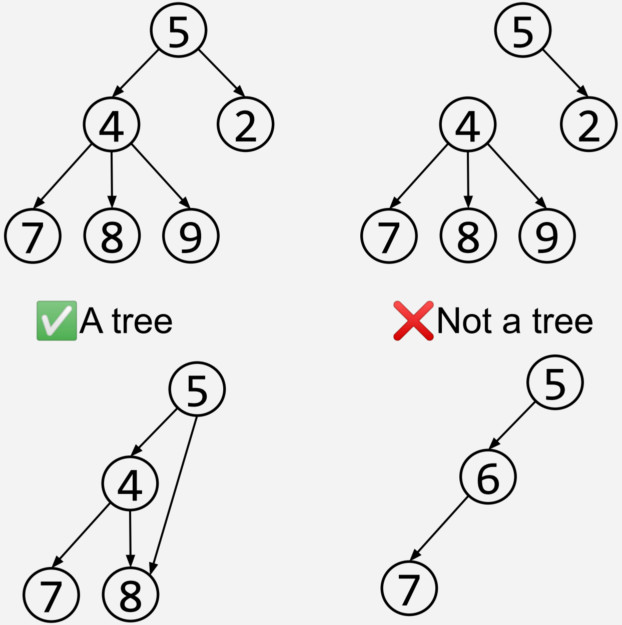




Tree Overview

- Tree:
  - Data structure with nodes
  - Similar to linked list
- Nodes:
  - Zero or more successors (children)
  - Exactly one predecessor (parent)
    - Except the root, which has none
- All nodes are reachable from root

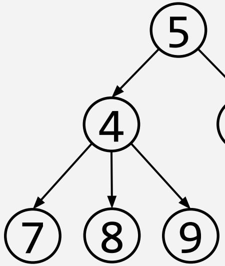
A tree or not a tree?



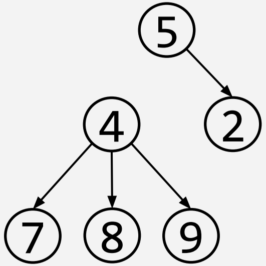
Tree Overview

- Tree:
  - Data structure with nodes
  - Similar to linked list
- Nodes:
  - Zero or more successors (children)
  - Exactly one predecessor (parent)
    - Except the root, which has none
- All nodes are reachable from root

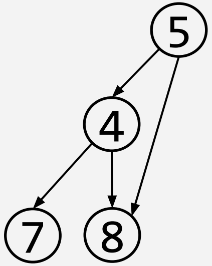
A tree or not a tree?



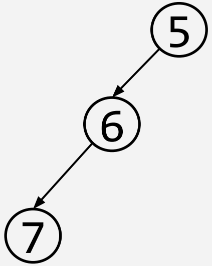
✅A tree



❌Not a tree



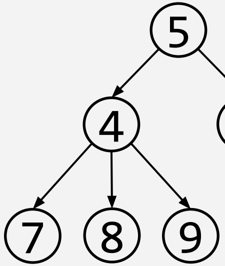
❌Not a tree



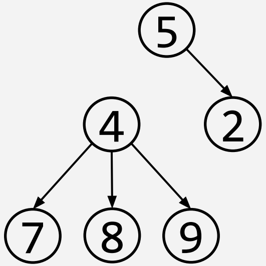
Tree Overview

- Tree:
  - Data structure with nodes
  - Similar to linked list
- Nodes:
  - Zero or more successors (children)
  - Exactly one predecessor (parent)
    - Except the root, which has none
- All nodes are reachable from root

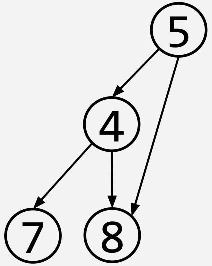
A tree or not a tree?



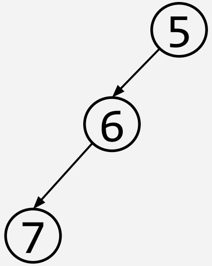
✅A tree



❌Not a tree



❌Not a tree



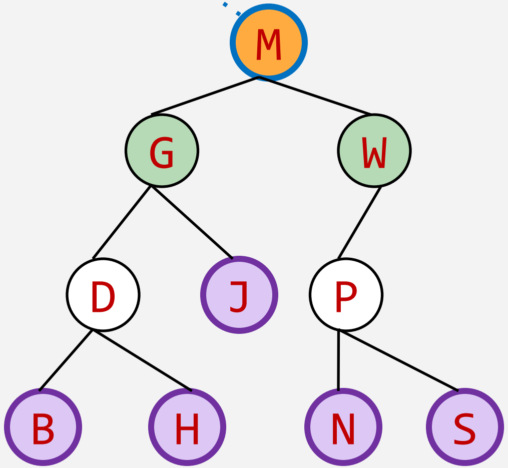
✅A tree

سوال؟

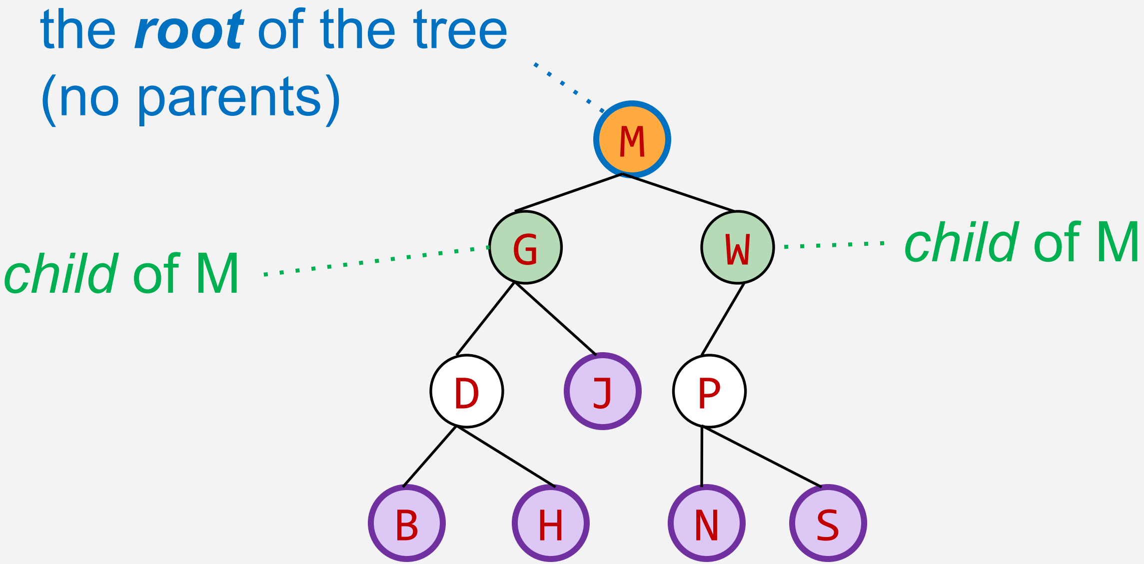
اصطلاحات درخت

Parent, Child, Leaves, Root

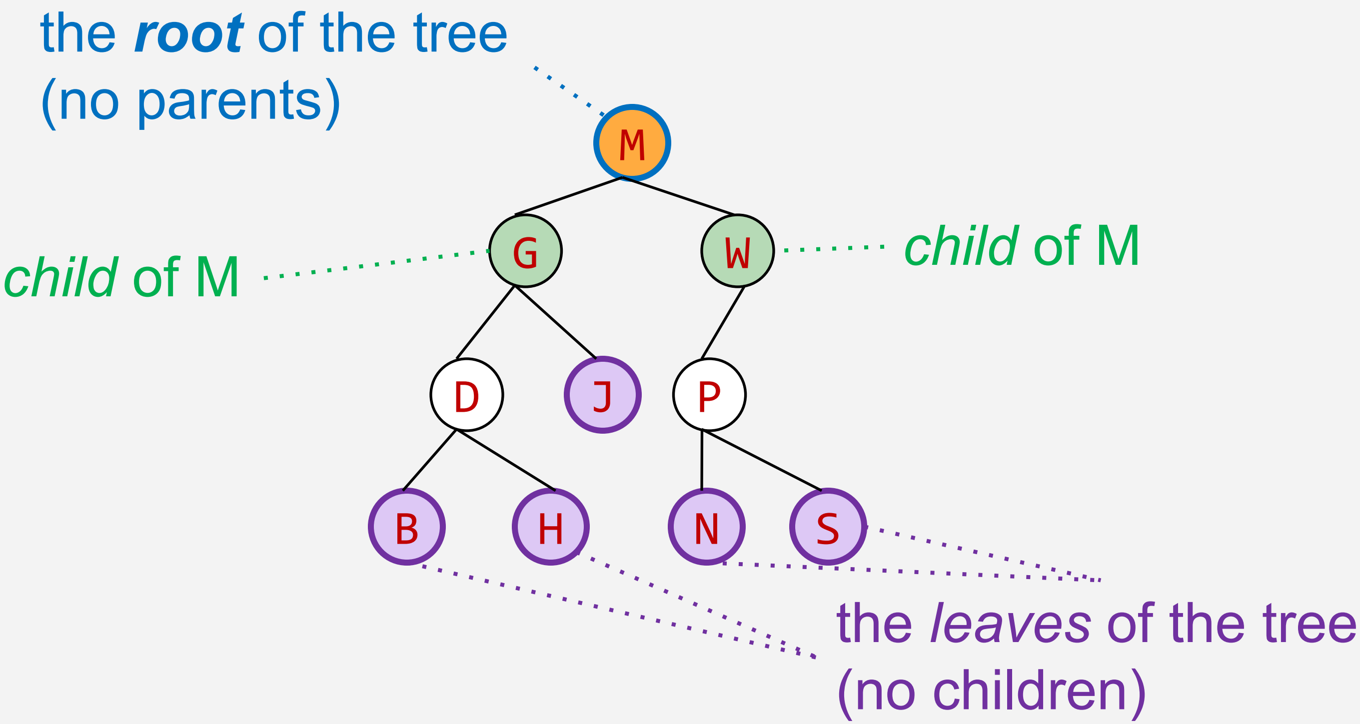
the ***root*** of the tree
(no parents)



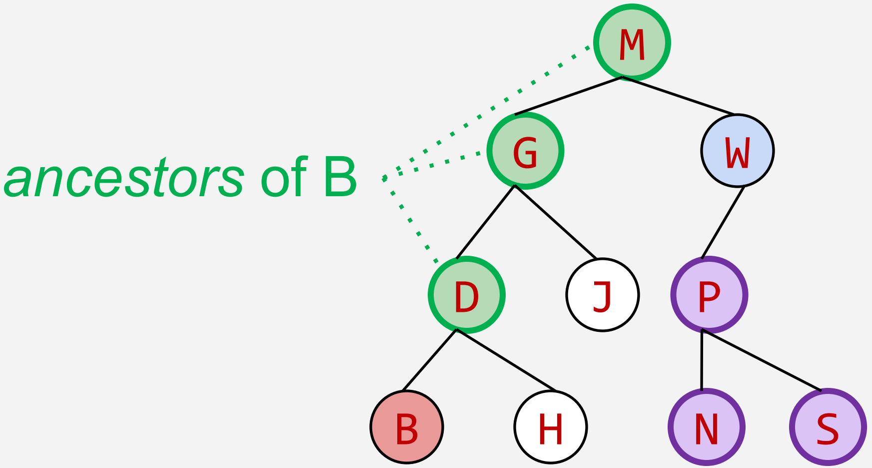
Parent, Child, Leaves, Root



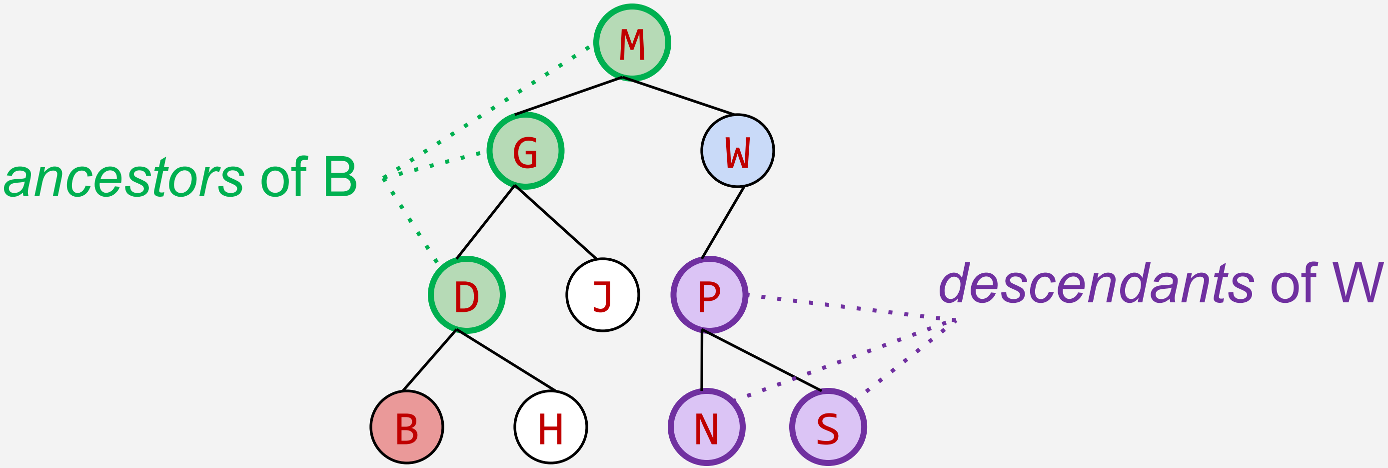
Parent, Child, Leaves, Root



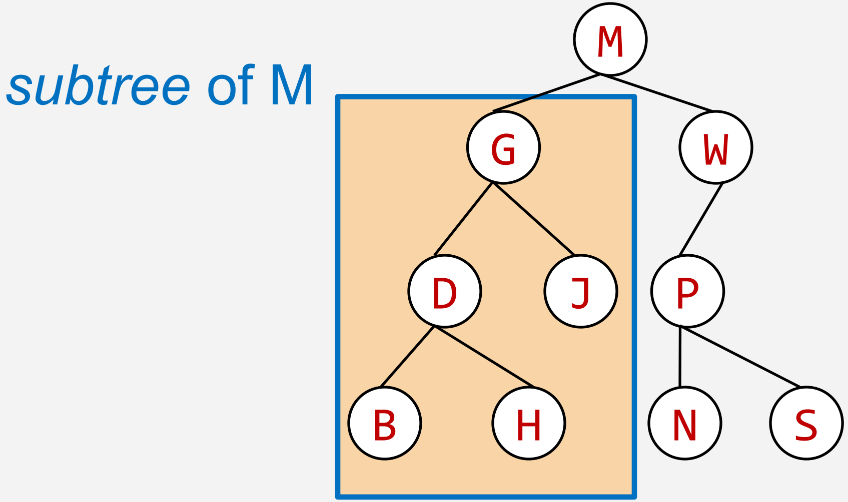
Ancestors and Descendants



Ancestors and Descendants

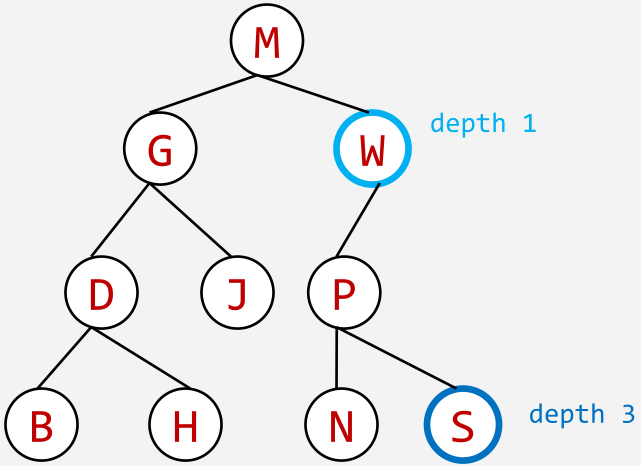


Subtree



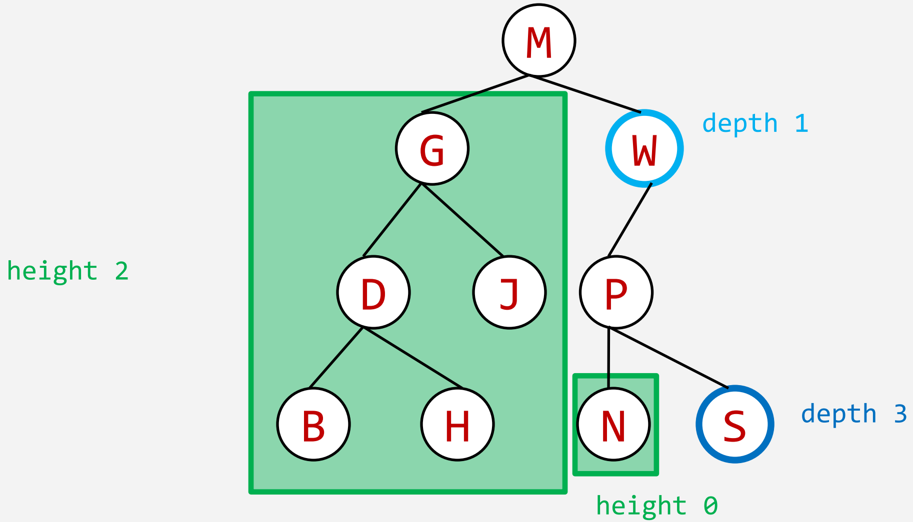
Depth & Height

- **Node depth**: the length of the path to the root



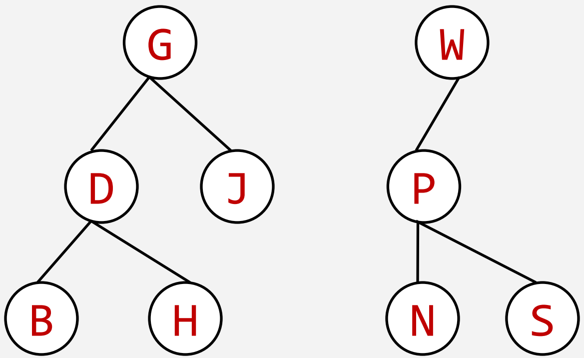
Depth & Height

- **Node depth**: the length of the path to the root
- **Tree (or subtree) height**: the length of the longest path from the root to a leaf



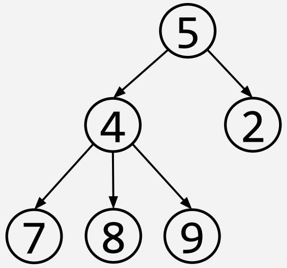
Forest

- Multiple trees!



General vs. Binary Trees

- **General tree**: every node can have an arbitrary number of children



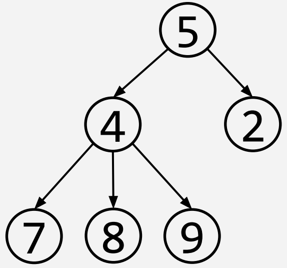
General tree

General vs. Binary Trees

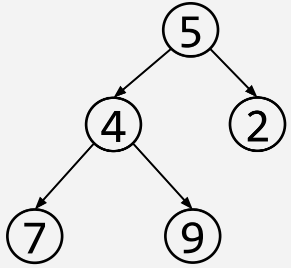
- **General tree**: every node can have an arbitrary number of children

- **Binary tree**: at most two children, called left and right

  ...often "tree" means binary tree



General tree



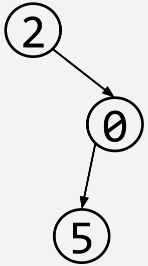
Binary tree

Nodes at each level in binary tree

- Maximum # of nodes at depth d: 2^d

Nodes at each level in binary tree

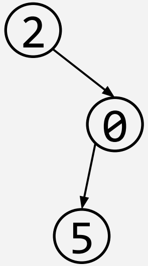
- Maximum # of nodes at depth d: 2^d
- If height of tree is h:
  - Minimum # of nodes: $h + 1$



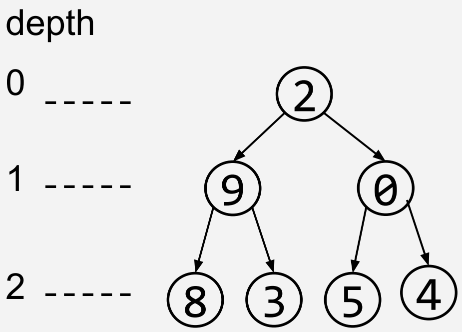
Height 2,
minimum number of nodes

Nodes at each level in binary tree

- Maximum # of nodes at depth d: 2^d
- If height of tree is h:
  - Minimum # of nodes: h + 1
  - Maximum # of nodes:
    - $2^0 + \ldots + 2^h = 2^{h+1} - 1$
    - Known as **Perfect tree**



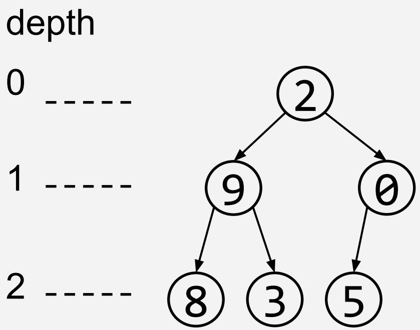
Height 2,
minimum number of nodes



Height 2,
maximum number of nodes

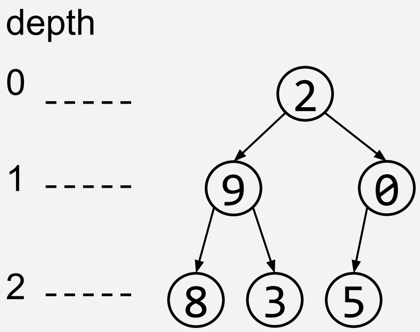
Complete binary tree

- Every level, except last, is completely filled
- Nodes on bottom level as far left as possible
    - I.e., no holes



Complete binary tree

- Every level, except last, is completely filled
- Nodes on bottom level as far left as possible
    - I.e., no holes
- We saw it before in priority queue (heap)!

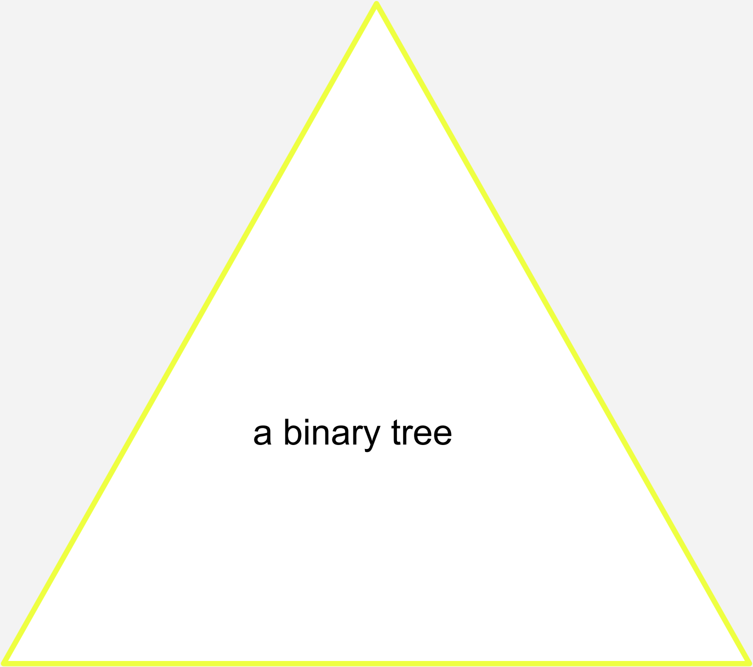


سوال؟

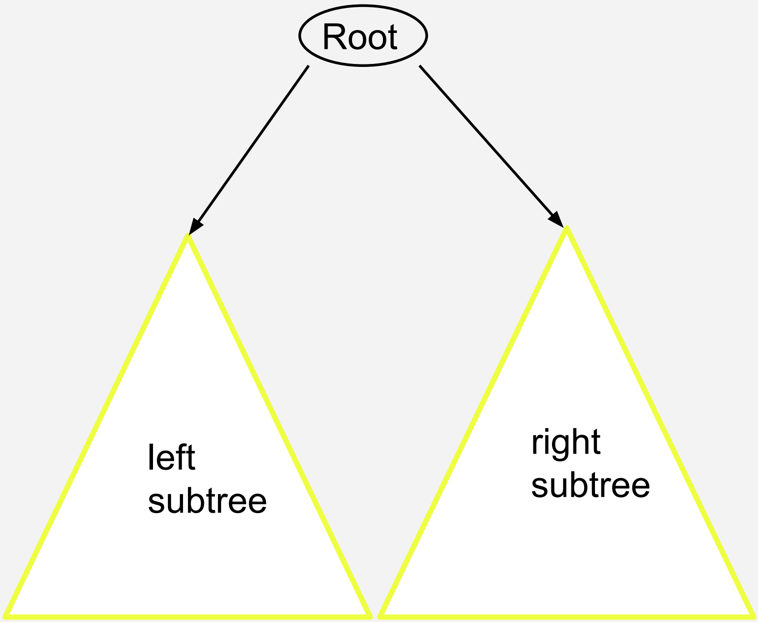
پردازش درخت

انجام عملیات بر روی درخت

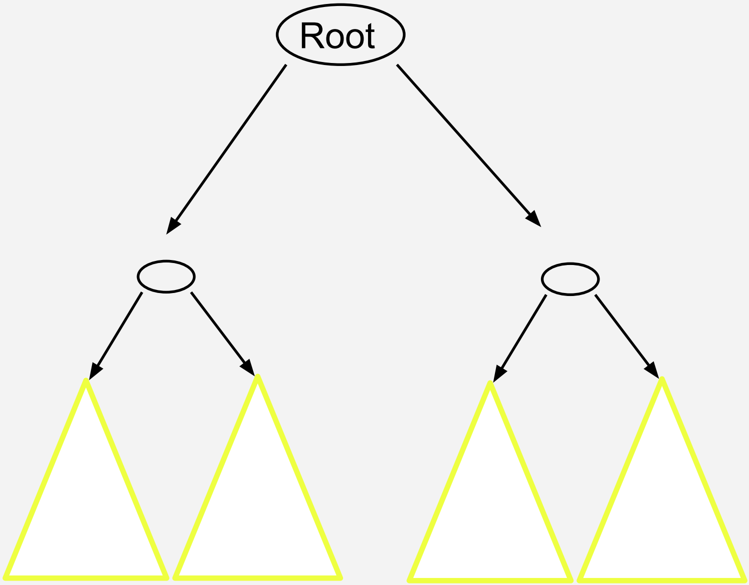
Recursive Definition



Recursive Definition

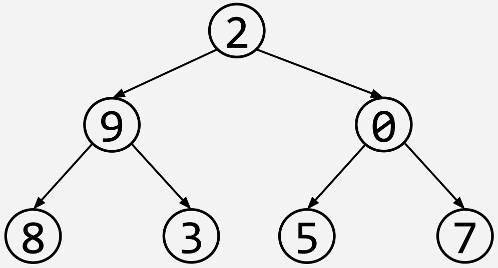


Recursive Definition

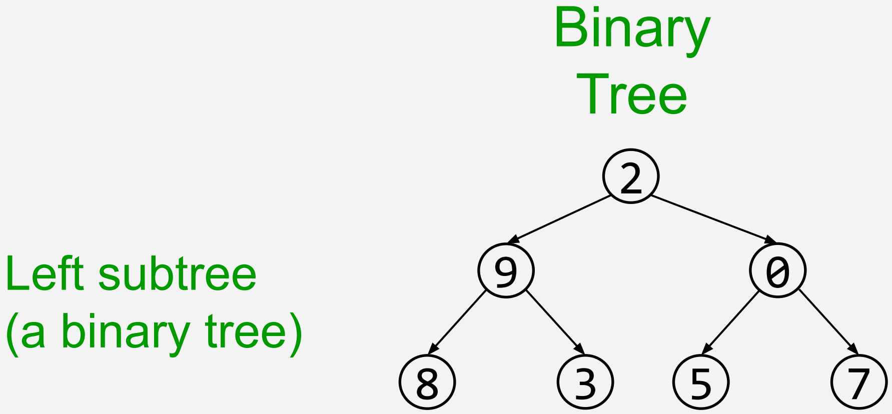


Binary Tree

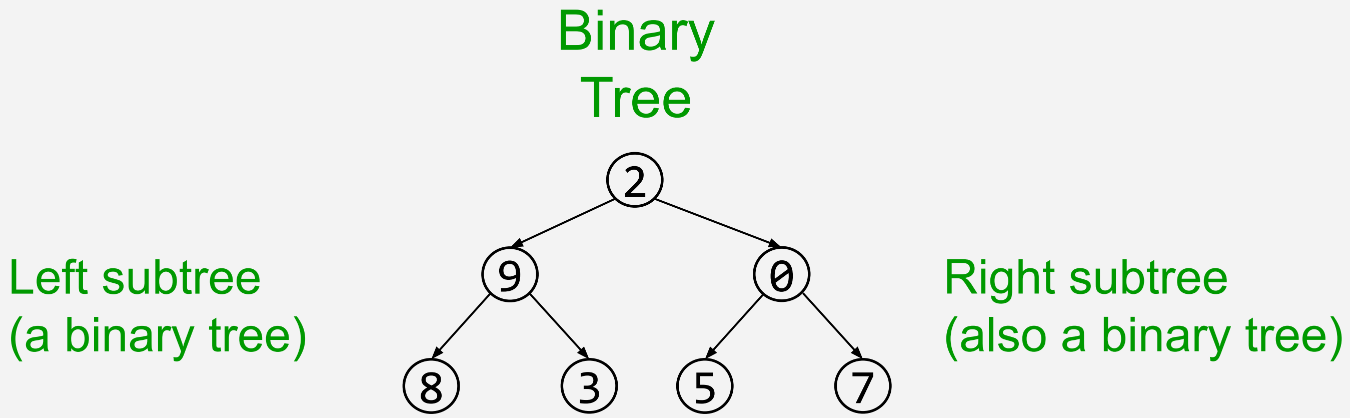
Binary Tree



Binary Tree



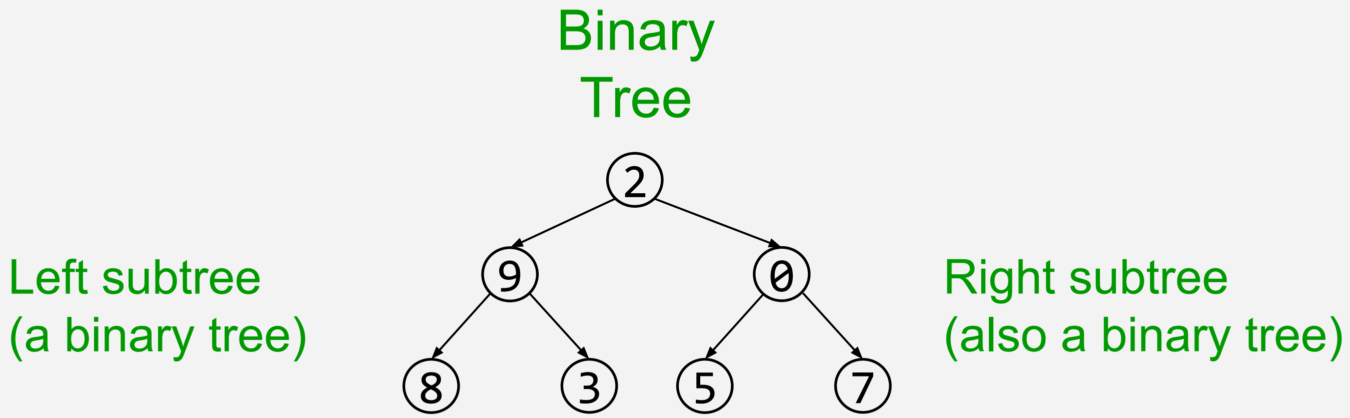
Binary Tree



Binary
Tree
2
9
0
8
3
5
7
Left subtree
(a binary tree)
Right subtree
(also a binary tree)

Binary Tree

- A binary tree is either
  - Null
  - An object consisting of a value, a left binary tree, and a right binary tree



Recipe for Recursive Functions

- Base case:
  - If the input is “easy,” just solve the problem directly.

- Recursive case:
  - Get a smaller part of the input (or several parts).
  - Call the function on the smaller value(s).
  - Use the recursive result to build a solution for the full input.

Recipe for Recursive Functions on Binary Trees

- Base case:
    - If the input is "easy," just solve the problem directly.

- Recursive case:
    - Get a smaller part of the input (or several parts).
    - Call the function on the smaller value(s).
    - Use the recursive result to build a solution for the full input.

Recipe for Recursive Functions on Binary Trees

- Base case:
  - If the input is "~~easy~~," just solve the problem directly.

    an empty tree (null), or a leaf

- Recursive case:
  - Get a smaller part of the input (or several parts).
  - Call the function on the smaller value(s).
  - Use the recursive result to build a solution for the full input.

Recipe for Recursive Functions on Binary Trees

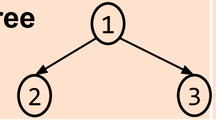
- Base case:
  - If the input is ~~"easy,"~~ just solve the problem directly.

    an empty tree (null), or a leaf

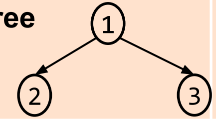
- Recursive case:
  - ~~Get a smaller part of the input (or several parts).~~
  - Call the function on ~~the smaller value(s).~~ each subtree.
  - Use the recursive result to build a solution for the full input.

سوال؟

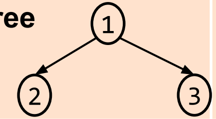
Search in a Tree

Data Structure	add(val v)	get(int i)	contains(val v)
Array 2 1 3 0	$O(n)$	$O(1)$	$O(n)$
Linked List 2 → 1 → 3 → 0	$O(1)$	$O(n)$	$O(n)$
Binary Tree 			

Search in a Tree

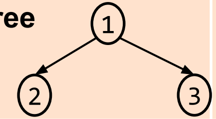
Data Structure	add(val v)	get(int i)	contains(val v)
Array 2 1 3 0	$O(n)$	$O(1)$	$O(n)$
Linked List 2 → 1 → 3 → 0	$O(1)$	$O(n)$	$O(n)$
Binary Tree 			$O(n)$

Search in a Tree

Data Structure	add(val v)	get(int i)	contains(val v)
Array 2 1 3 0	$O(n)$	$O(1)$	$O(n)$
Linked List 2 → 1 → 3 → 0	$O(1)$	$O(n)$	$O(n)$
Binary Tree 			$O(n)$

Node could be *anywhere* in tree

Search in a Tree

Data Structure	**add**(val v)	**get**(int i)	**contains**(val v)
Array 2 1 3 0	$O(n)$	$O(1)$	$O(n)$
Linked List 2 → 1 → 3 → 0	$O(1)$	$O(n)$	$O(n)$
Binary Tree 			$O(n)$

Node could be *anywhere* in tree

Binary search on arrays: O(log n)
Requires invariant: array sorted
…analogue for trees?
TO BE CONTINUED!
(in a future lecture)

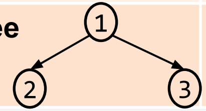
سوال؟

پیمایش درخت

پیمایش و ذخیره یک درخت

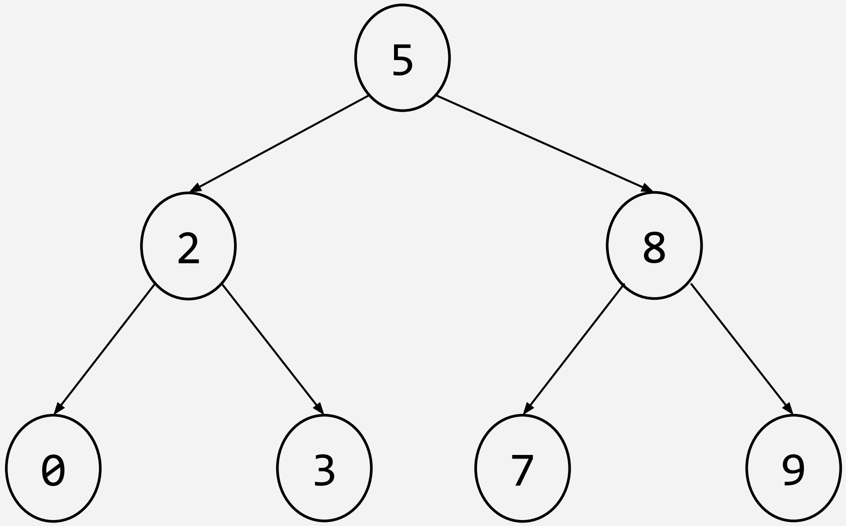
Iterate through data structure

- Iterate: process elements of data structure
  - Sum all elements
  - Print each element

Data Structure	**Order to iterate**
Array 2 1 3 0	Forwards: 2, 1, 3, 0 Backwards: 0, 3, 1, 2
Linked List 2 → 1 → 3 → 0	Forwards: 2, 1, 3, 0
Binary Tree 	**???**

Iterate through a tree

- What would a reasonable order be?



Tree traversals

- Iterating through tree is also known as tree traversal

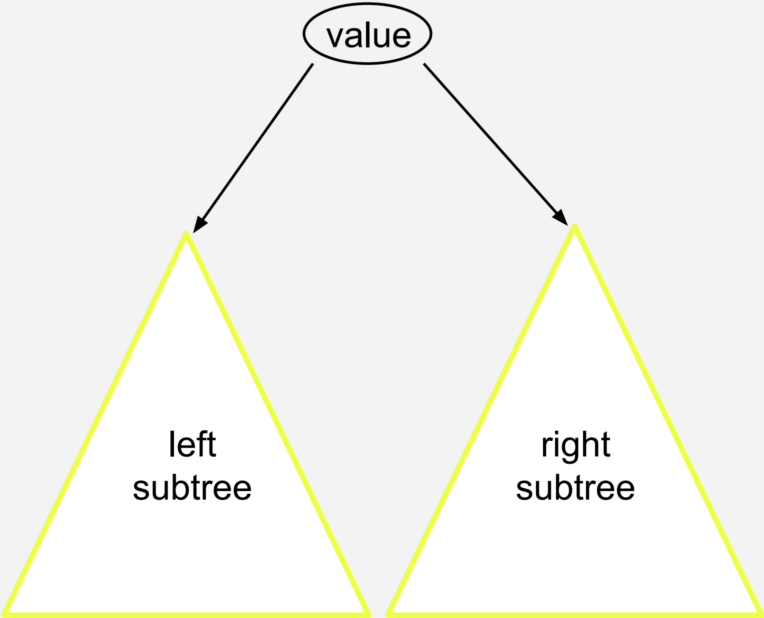
- Well-known recursive tree traversal algorithms:
  - Preorder
  - Inorder
  - Postorder

- Another, non-recursive: level order (BFS!)

پیمایش پیش ترتیب

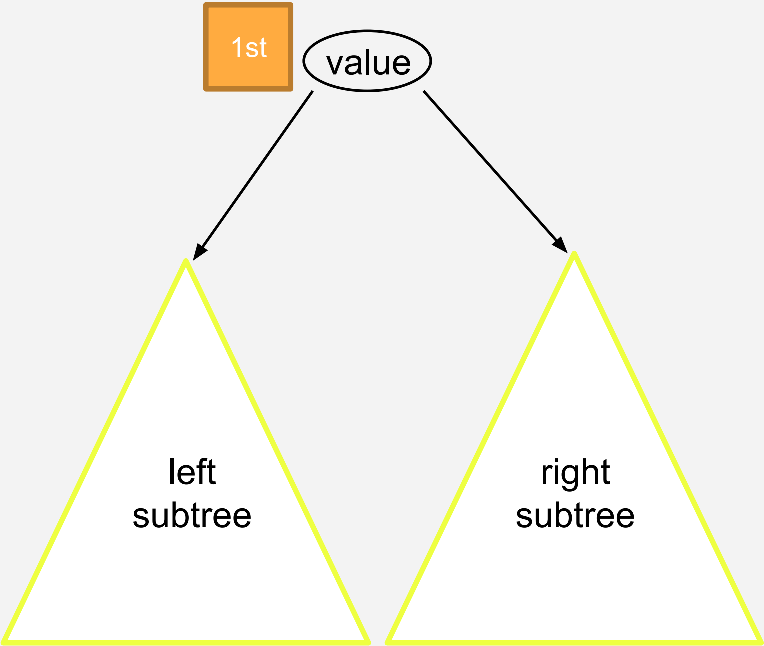
Preorder

- "Pre:" process root before subtrees



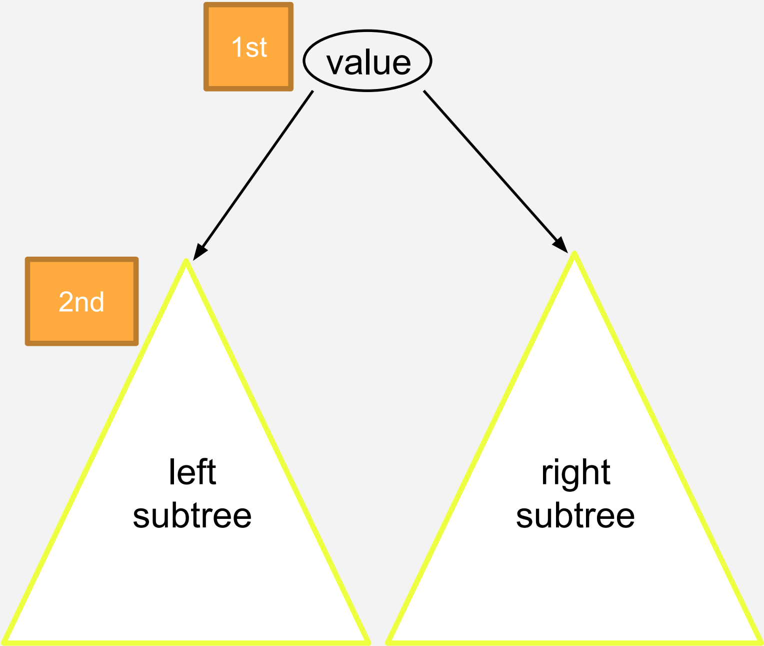
Preorder

- “Pre:” process root before subtrees



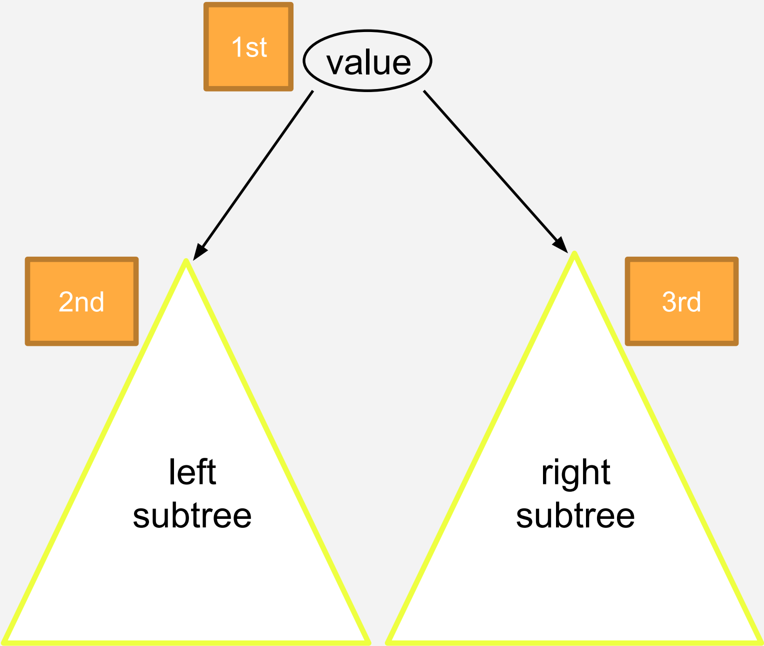
Preorder

- “Pre:” process root before subtrees



Preorder

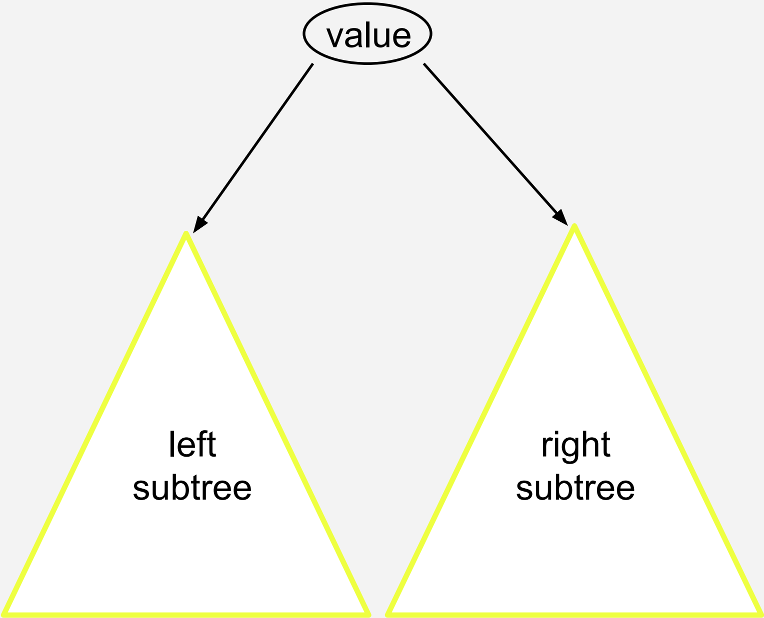
- “Pre:” process root before subtrees



پیمایش میان ترتیب

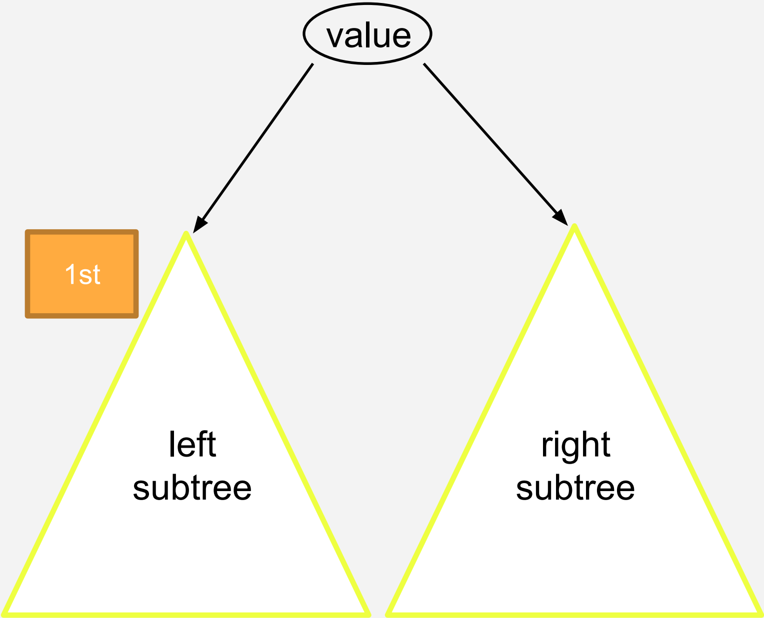
Inorder

- “In:” process root in-between subtrees



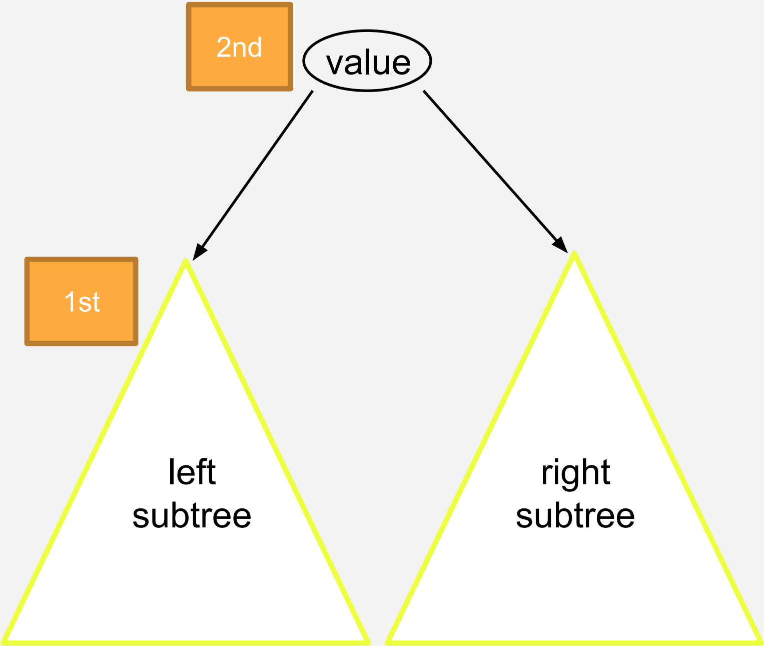
Inorder

- "In:" process root in-between subtrees



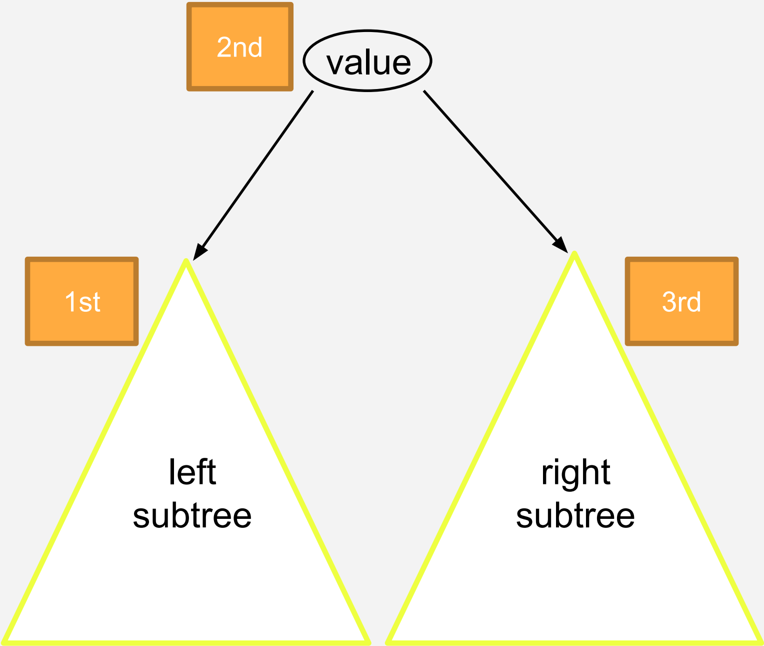
Inorder

- "In:" process root in-between subtrees



Inorder

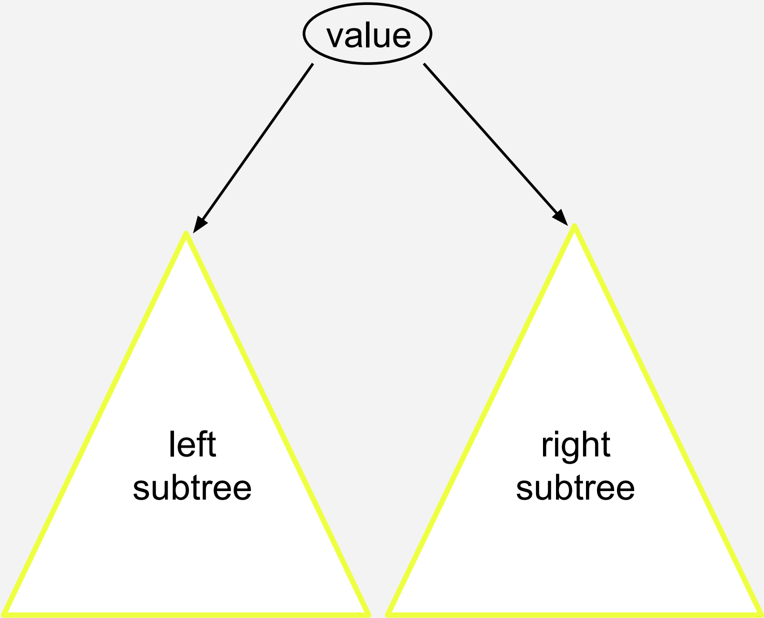
- "In:" process root in-between subtrees



پیمایش پس ترتیب

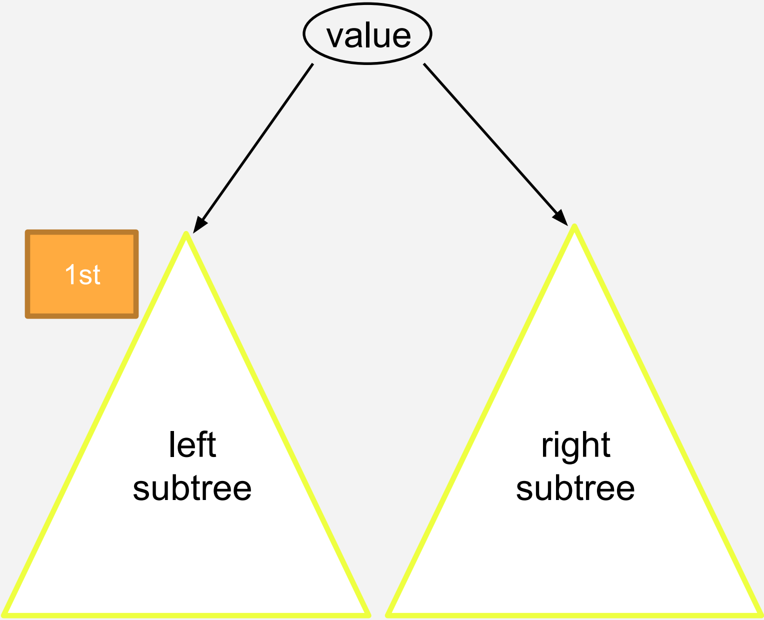
Postorder

- “Post:” process root after subtrees



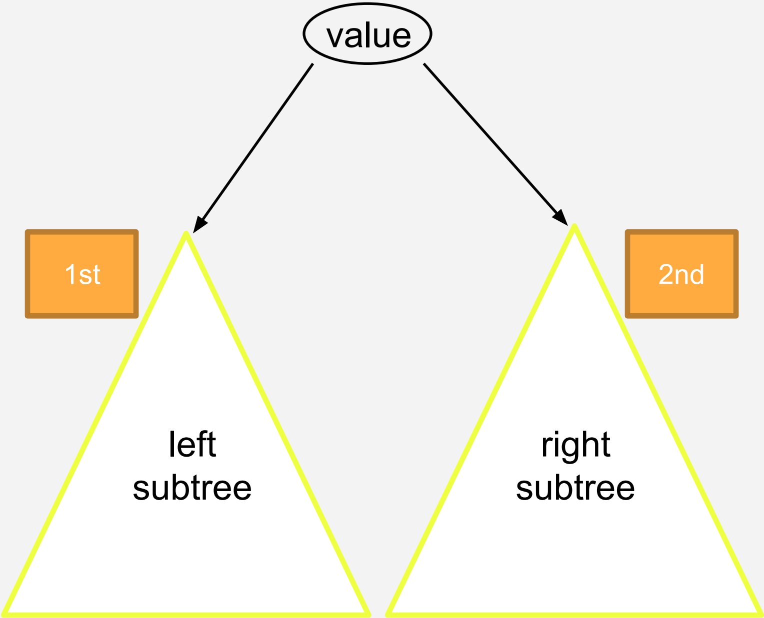
Postorder

- "Post:" process root after subtrees



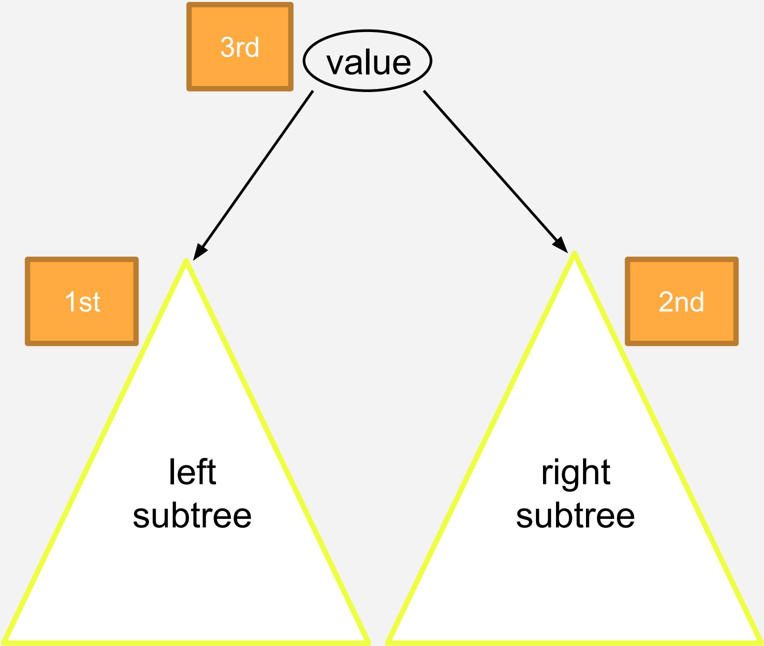
Postorder

- “Post:” process root after subtrees



Postorder

- “Post:” process root after subtrees



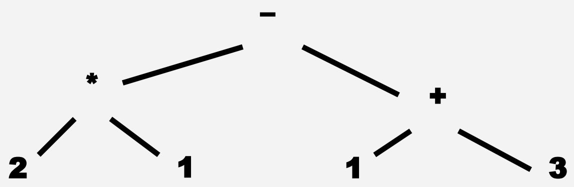
سوال؟

درخت عبارت

نمونه ای از کاربرد درخت و پیمایش آن

Syntax Trees

- Trees can represent expressions (Java, math, …)
- Expression: 2 * 1 – (1 + 3)
- Tree:



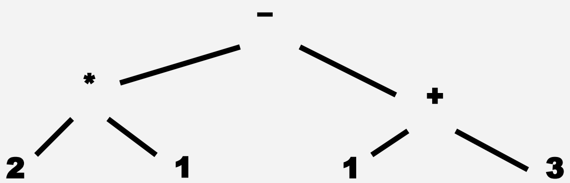
پیمایش پیش ترتیب عبارت

Preorder Traversals of Expression Tree

1- Visit the root

2- Visit the left subtree

3- Visit the right subtree



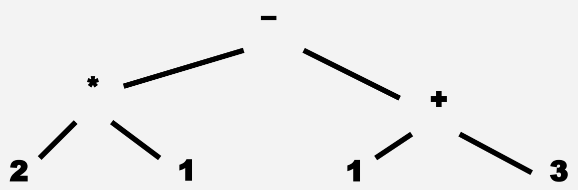
Preorder:

Preorder Traversals of Expression Tree

1- Visit the root

2- Visit the left subtree

3- Visit the right subtree



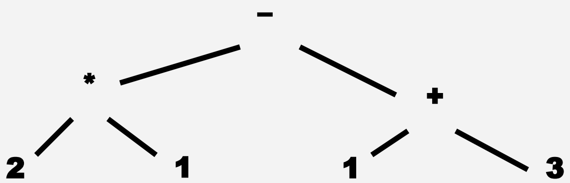
Preorder: -

Preorder Traversals of Expression Tree

1- Visit the root

2- Visit the left subtree

3- Visit the right subtree



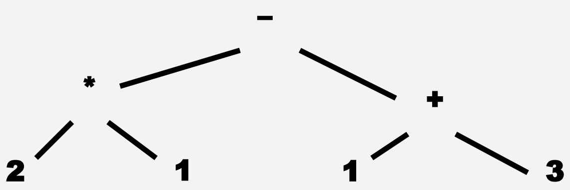
Preorder: - *

Preorder Traversals of Expression Tree

1- Visit the root

2- Visit the left subtree

3- Visit the right subtree



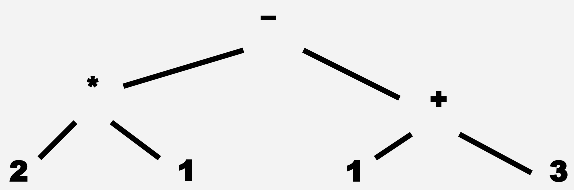
Preorder: - * 2

Preorder Traversals of Expression Tree

1- Visit the root

2- Visit the left subtree

3- Visit the right subtree



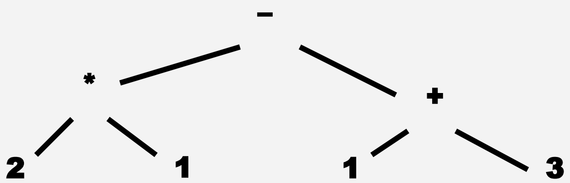
Preorder: - * 2 1

Preorder Traversals of Expression Tree

1- Visit the root

2- Visit the left subtree

3- Visit the right subtree



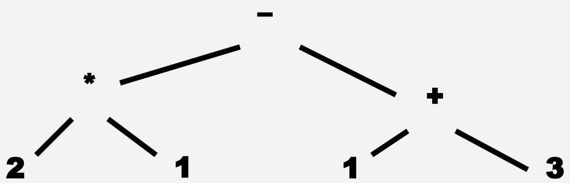
Preorder: - * 2 1 +

Preorder Traversals of Expression Tree

1- Visit the root

2- Visit the left subtree

3- Visit the right subtree



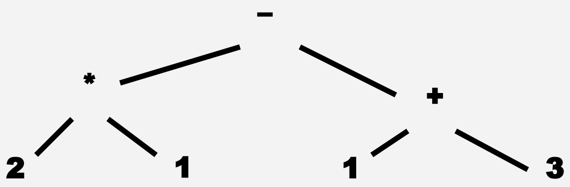
Preorder: - * 2 1 + 1

Preorder Traversals of Expression Tree

1- Visit the root

2- Visit the left subtree

3- Visit the right subtree



Preorder: - * 2 1 + 1 3

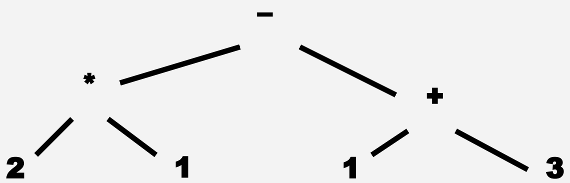
پیمایش پس ترتیب عبارت

Postorder Traversals of Expression Tree

1- Visit the left subtree

2- Visit the right subtree

3- Visit the root



Preorder: - * 2 1 + 1 3

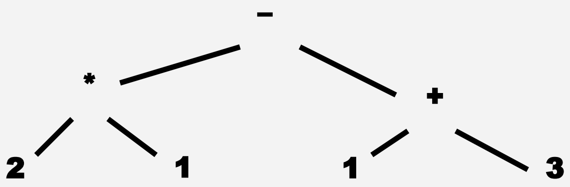
Postorder:

Postorder Traversals of Expression Tree

1- Visit the left subtree

2- Visit the right subtree

3- Visit the root



Preorder: - * 2 1 + 1 3

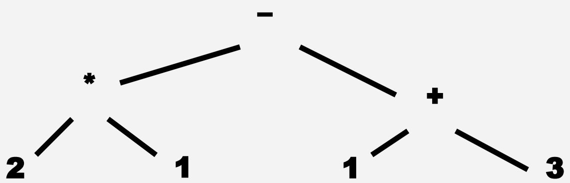
Postorder: 2

Postorder Traversals of Expression Tree

1- Visit the left subtree

2- Visit the right subtree

3- Visit the root



Preorder: - * 2 1 + 1 3

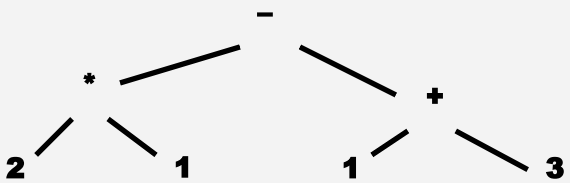
Postorder: 2 1

Postorder Traversals of Expression Tree

1- Visit the left subtree

2- Visit the right subtree

3- Visit the root



Preorder: - * 2 1 + 1 3

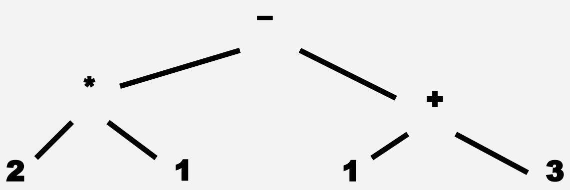
Postorder: 2 1 *

Postorder Traversals of Expression Tree

1- Visit the left subtree

2- Visit the right subtree

3- Visit the root



Preorder: - * 2 1 + 1 3

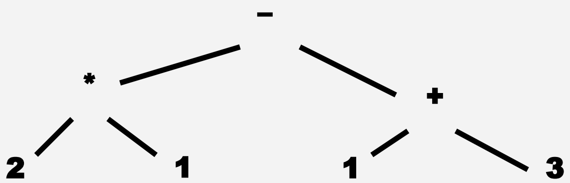
Postorder: 2 1 * 1

Postorder Traversals of Expression Tree

1- Visit the left subtree

2- Visit the right subtree

3- Visit the root



Preorder: - * 2 1 + 1 3

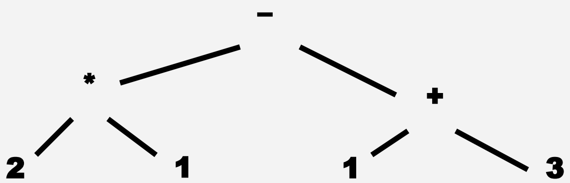
Postorder: 2 1 * 1 3

Postorder Traversals of Expression Tree

1- Visit the left subtree

2- Visit the right subtree

3- Visit the root



Preorder: - * 2 1 + 1 3

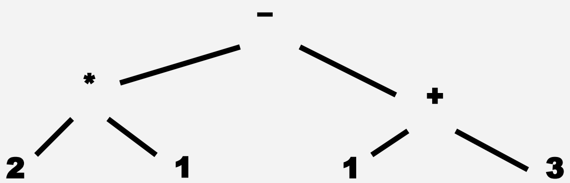
Postorder: 2 1 * 1 3 +

Postorder Traversals of Expression Tree

1- Visit the left subtree

2- Visit the right subtree

3- Visit the root



Preorder: - * 2 1 + 1 3

Postorder: 2 1 * 1 3 + -

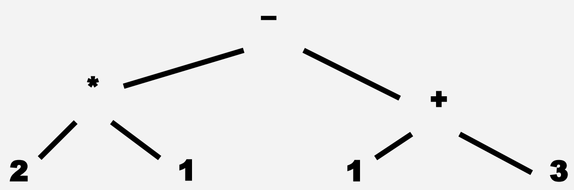
پیمایش میان ترتیب عبارت

Inorder Traversals of Expression Tree

1- Visit the left subtree

2- Visit the root

3- Visit the right subtree



Preorder: - * 2 1 + 1 3

Postorder: 2 1 * 1 3 + -

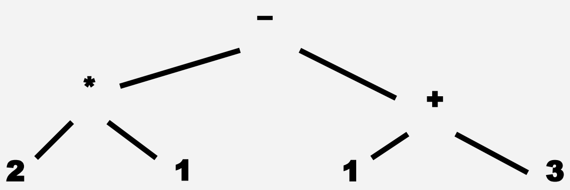
Inorder:

Inorder Traversals of Expression Tree

1- Visit the left subtree

2- Visit the root

3- Visit the right subtree



Preorder: - * 2 1 + 1 3

Postorder: 2 1 * 1 3 + -

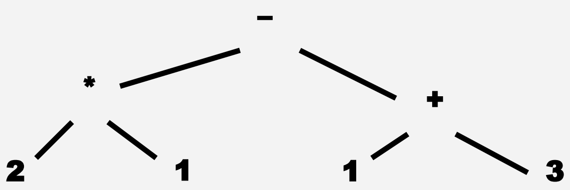
Inorder: 2

Inorder Traversals of Expression Tree

1- Visit the left subtree

2- Visit the root

3- Visit the right subtree



Preorder: - * 2 1 + 1 3

Postorder: 2 1 * 1 3 + -

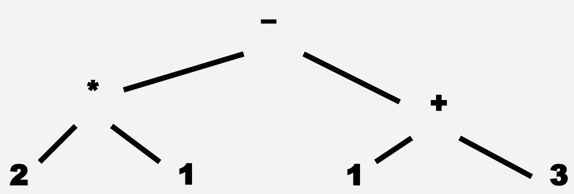
Inorder: 2 *

Inorder Traversals of Expression Tree

1- Visit the left subtree

2- Visit the root

3- Visit the right subtree



Preorder: - * 2 1 + 1 3

Postorder: 2 1 * 1 3 + -

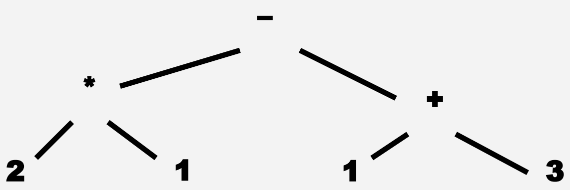
Inorder: 2 * 1

Inorder Traversals of Expression Tree

1- Visit the left subtree

2- Visit the root

3- Visit the right subtree



Preorder: - * 2 1 + 1 3

Postorder: 2 1 * 1 3 + -

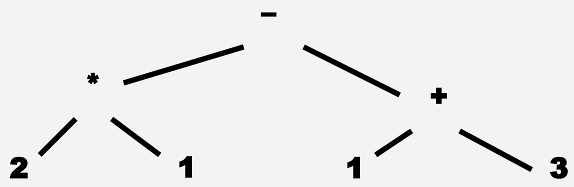
Inorder: 2 * 1 -

Inorder Traversals of Expression Tree

1- Visit the left subtree

2- Visit the root

3- Visit the right subtree



Preorder: - * 2 1 + 1 3

Postorder: 2 1 * 1 3 + -

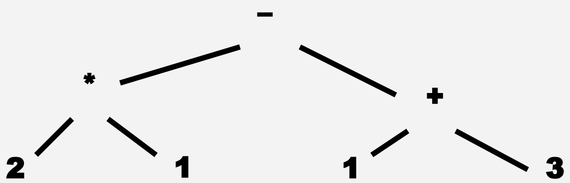
Inorder: 2 * 1 - 1

Inorder Traversals of Expression Tree

1- Visit the left subtree

2- Visit the root

3- Visit the right subtree



Preorder: - * 2 1 + 1 3

Postorder: 2 1 * 1 3 + -

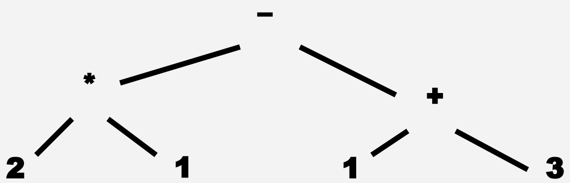
Inorder: 2 * 1 - 1 +

Inorder Traversals of Expression Tree

1- Visit the left subtree

2- Visit the root

3- Visit the right subtree



Preorder: - * 2 1 + 1 3

Postorder: 2 1 * 1 3 + -

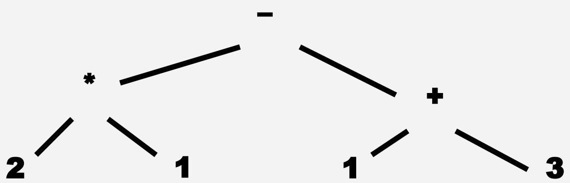
Inorder: 2 * 1 - 1 + 3

Inorder Traversals of Expression Tree

1- Visit the left subtree

2- Visit the root

3- Visit the right subtree



Original expression, except for parenthesis

Preorder: - * 2 1 + 1 3

Postorder: 2 1 * 1 3 + -

Inorder: $\boxed{2 * 1 - 1 + 3}$

سوال؟

Prefix Notation

- Function calls in most programming languages use prefix notation:
  - E.g., **`add(37, 5)`**
- Aka **Polish notation** (PN)
  - In honor of inventor, Polish logician Jan Łukasiewicz
- Some languages (Lisp, Scheme, Racket) use prefix notation for everything
  - Makes the syntax uniform

```
(- (* 2 1) (+ 1 3))

(define (fib n)
  (if (<= n 2)
      1
      (+ (fib (- n 1) (fib (- n 2))))))
```

Postfix Notation

- Some languages (Forth, PostScript, HP calculators) use postfix notation
- Aka **reverse Polish notation** (RPN)

```
2 1 mul 1 3 add sub

/fib { dup
       3 lt
         { pop 1 }
         { dup 1 sub fib exch 2 sub fib add }
       ifelse
     } def
```

Implementing Syntax Tree in Code

```
public interface Expr {
  int eval();
  String inorder();
}
```

Implementing Syntax Tree in Code

```
public interface Expr {
  int eval();
  String inorder();
}

public class Int implements Expr {
  private int v;
  public int eval() { return v; }
  public String inorder() { return " " + v + " "; }
}
```

Implementing Syntax Tree in Code

```
public interface Expr {
  int eval();
  String inorder();
}

public class Int implements Expr {
  private int v;
  public int eval() { return v; }
  public String inorder() { return " " + v + " "; }
}

public class Add implements Expr {
  private Expr left, right;
  public int eval() { return left.eval() + right.eval(); }
  public String inorder() {
    return "(" + left.infix() + "+" + right.infix() + ")";
  }
}
```

سوال؟

بازسازی درخت

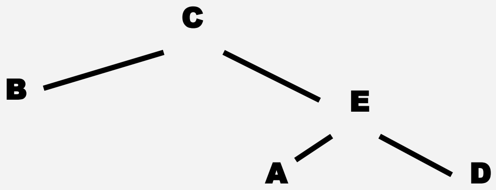
ساخت درخت از روی یک پیمایش آن

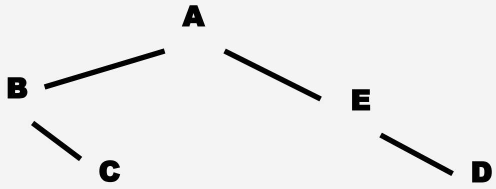
Recover tree from traversal

- Suppose inorder is B C A E D
- Can we recover the tree uniquely?

Recover tree from traversal

- Suppose inorder is B C A E D
- Can we recover the tree uniquely? NO!





Recover tree from traversals

- Suppose inorder is B C A E D
- preorder is A B C D E
- Can we determine the tree uniquely?

Recover tree from traversals

- Suppose inorder is B C A E D
- preorder is A B C D E
- Can we determine the tree uniquely? Yes!

Recover tree from traversals

- Suppose inorder is B C A E D
- preorder is A B C D E
- Can we determine the tree uniquely? Yes!

- What is root?

Recover tree from traversals

- Suppose inorder is B C A E D
- preorder is A B C D E
- Can we determine the tree uniquely? Yes!

- What is root? Preorder tells us: A

Recover tree from traversals

- Suppose inorder is B C A E D
- preorder is A B C D E
- Can we determine the tree uniquely? Yes!

- What is root? Preorder tells us: A
- What comes before/after root A?

Recover tree from traversals

- Suppose inorder is B C A E D
- preorder is A B C D E
- Can we determine the tree uniquely? Yes!

- What is root? Preorder tells us: A
- What comes before/after root A?
    - Inorder tells us:
        - Before: B C
        - After: E D

Recover tree from traversals

- Suppose inorder is B C A E D
- preorder is A B C D E
- Can we determine the tree uniquely? Yes!

- What is root? Preorder tells us: A
- What comes before/after root A?
  - Inorder tells us:
    - Before: B C
    - After: E D
- Now recurse! Figure out left/right subtrees using same technique.

Recover tree from traversals

- Suppose inorder is B C A E D
- preorder is A B C D E
- Root is A; left subtree contains B C; right contains E D

Recover tree from traversals

- Suppose inorder is B C A E D
- preorder is A B C D E
- Root is A; left subtree contains B C; right contains E D

Left:
Inorder is B C
Preorder is B C
- What is root? Preorder: B
- What is before/after B?
  Inorder:
  - Before: nothing
  - After: C

Recover tree from traversals

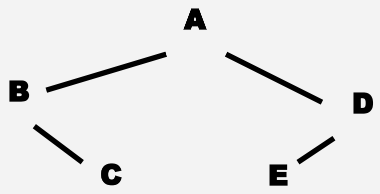
- Suppose inorder is B C A E D
- preorder is A B C D E
- Root is A; left subtree contains B C; right contains E D

Left:
Inorder is B C
Preorder is B C
- What is root? Preorder: B
- What is before/after B?
  Inorder:
  - Before: nothing
  - After: C

Right:
Inorder is E D
Preorder is D E
- What is root? Preorder: D
- What is before/after D?
  Inorder:
  - Before: E
  - After: nothing

Recover tree from traversals

- Suppose inorder is B C A E D
- preorder is A B C D E
- Root is A; left subtree contains B C; right contains E D
- Tree:



سوال؟