Binary Trees

Binary Trees

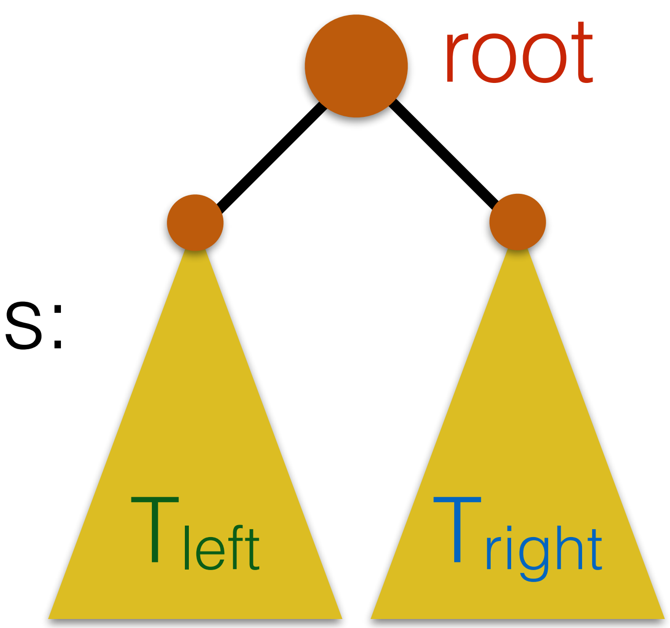
A **binary tree** is a structure T defined on a finite set of nodes such that either

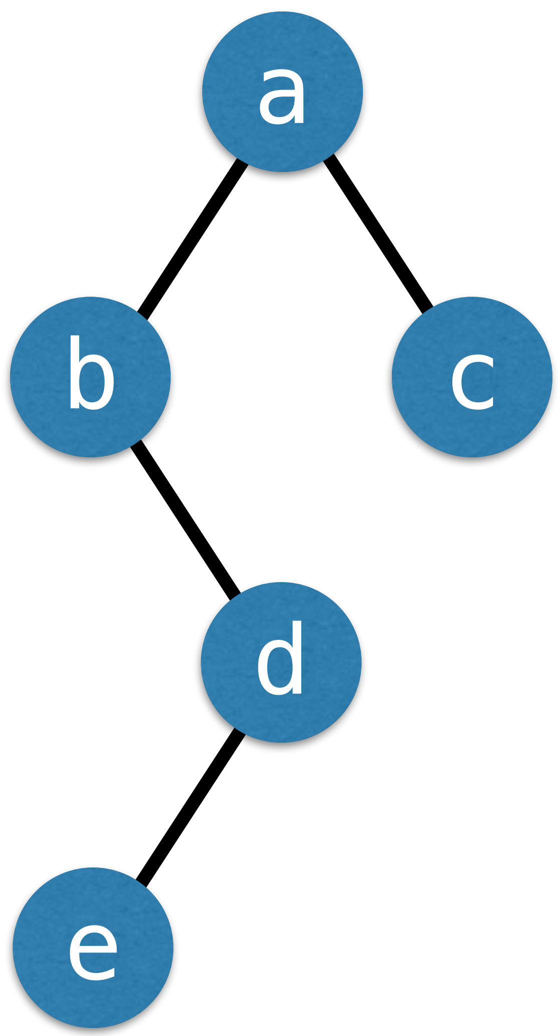
- T is empty (contains no nodes), or
- T is composed of three disjoint sets of nodes:
  - a **root** node,
  - a binary tree called the **left subtree** of T, and
  - a binary tree called the **right subtree** of T.

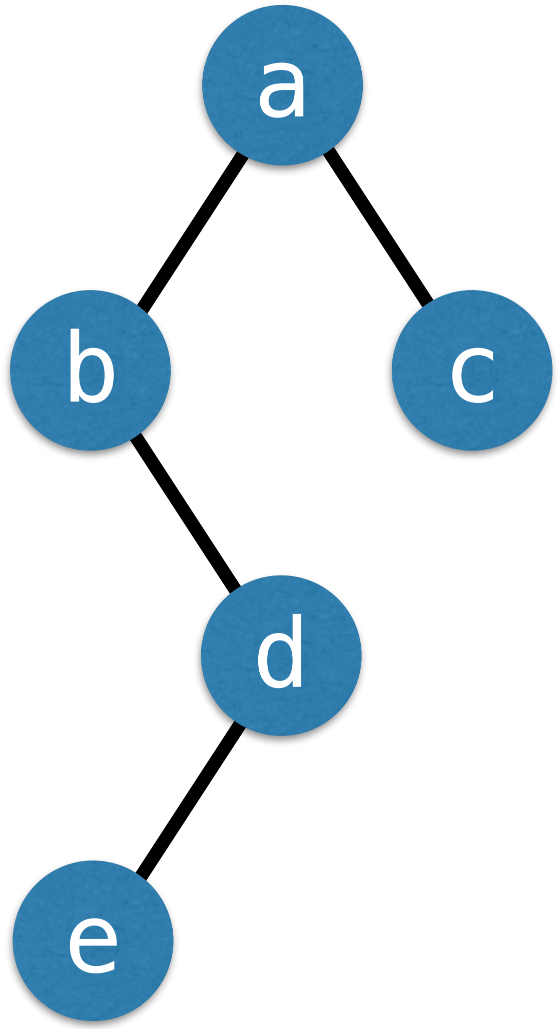
Binary Trees

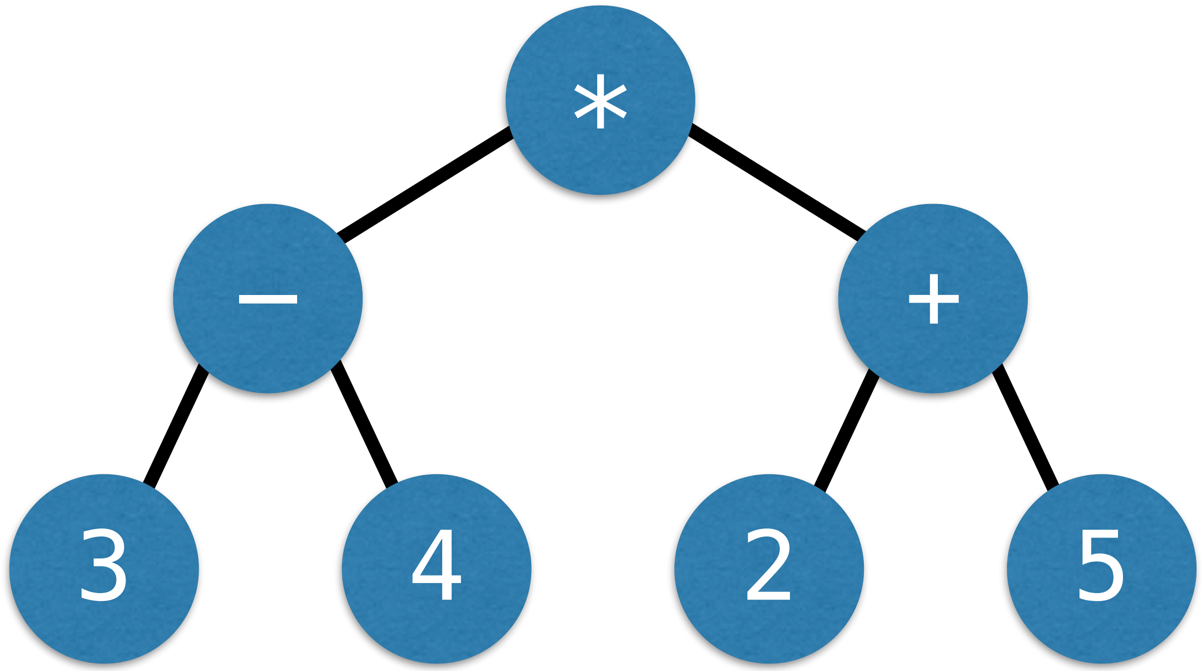
A **binary tree** is a structure T defined on a finite set of nodes such that either

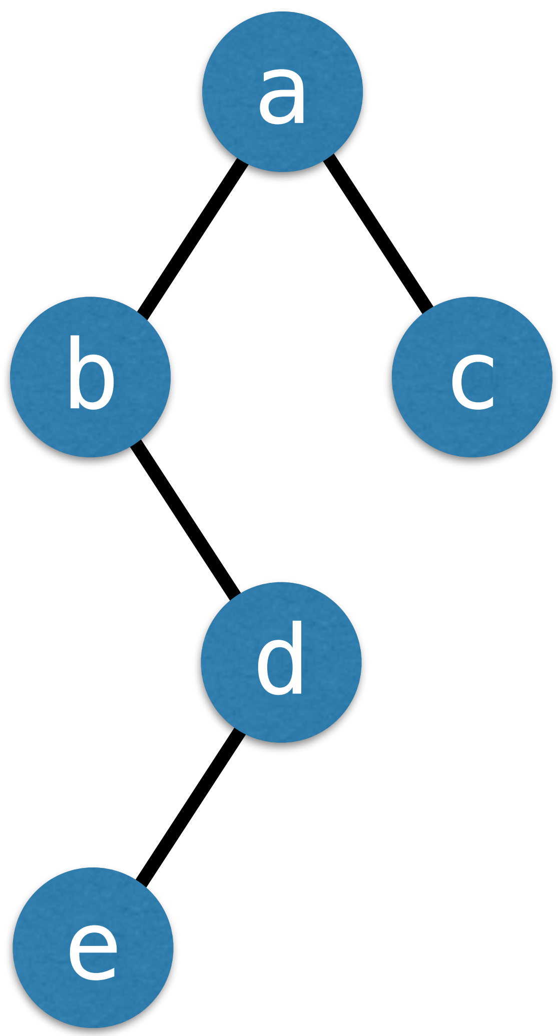
- T is empty (contains no nodes), or
- T is composed of three disjoint sets of nodes:
  - a **root** node,
  - a binary tree called the **left subtree** of T, and
  - a binary tree called the **right subtree** of T.

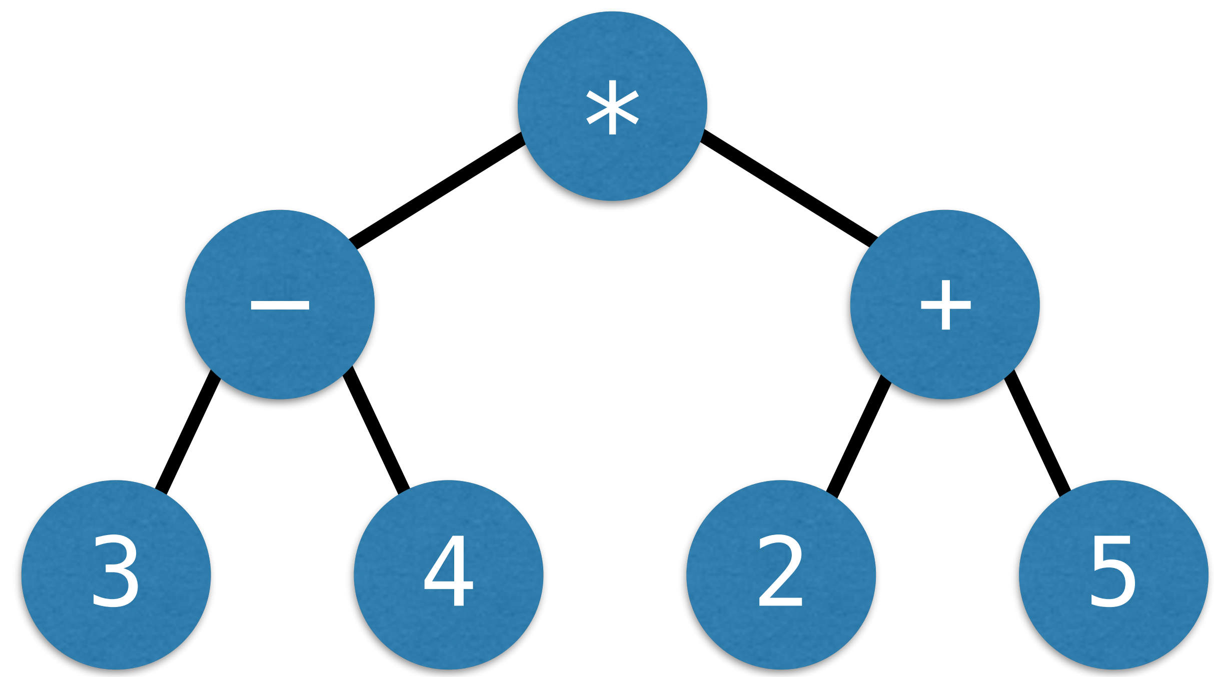



a
b
c
d
e

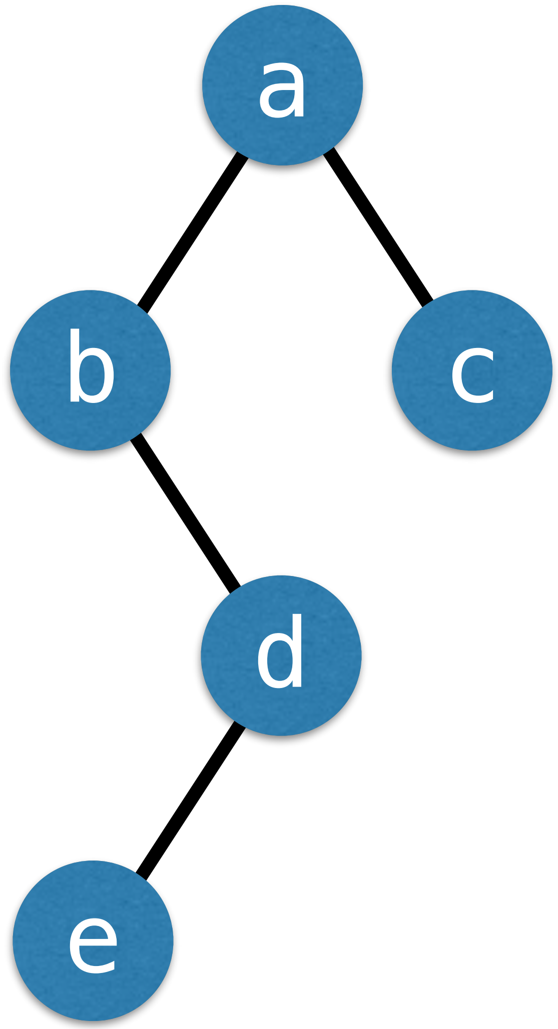

a
b
c
d
e


*
−
+
3
4
2
5

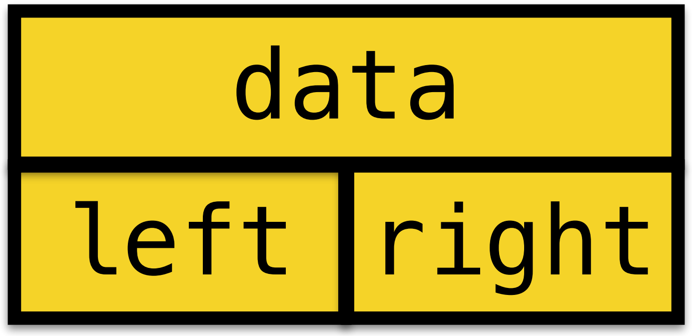


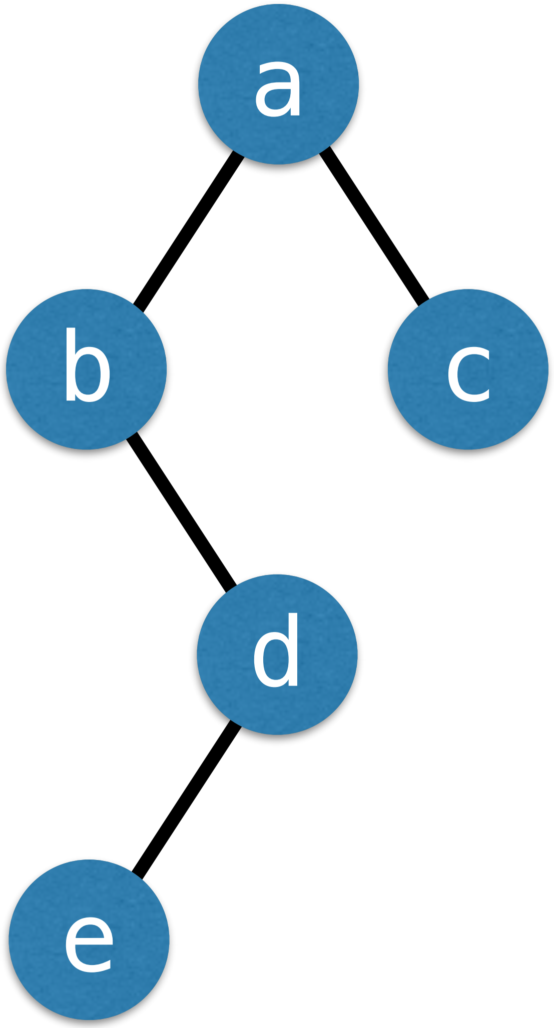


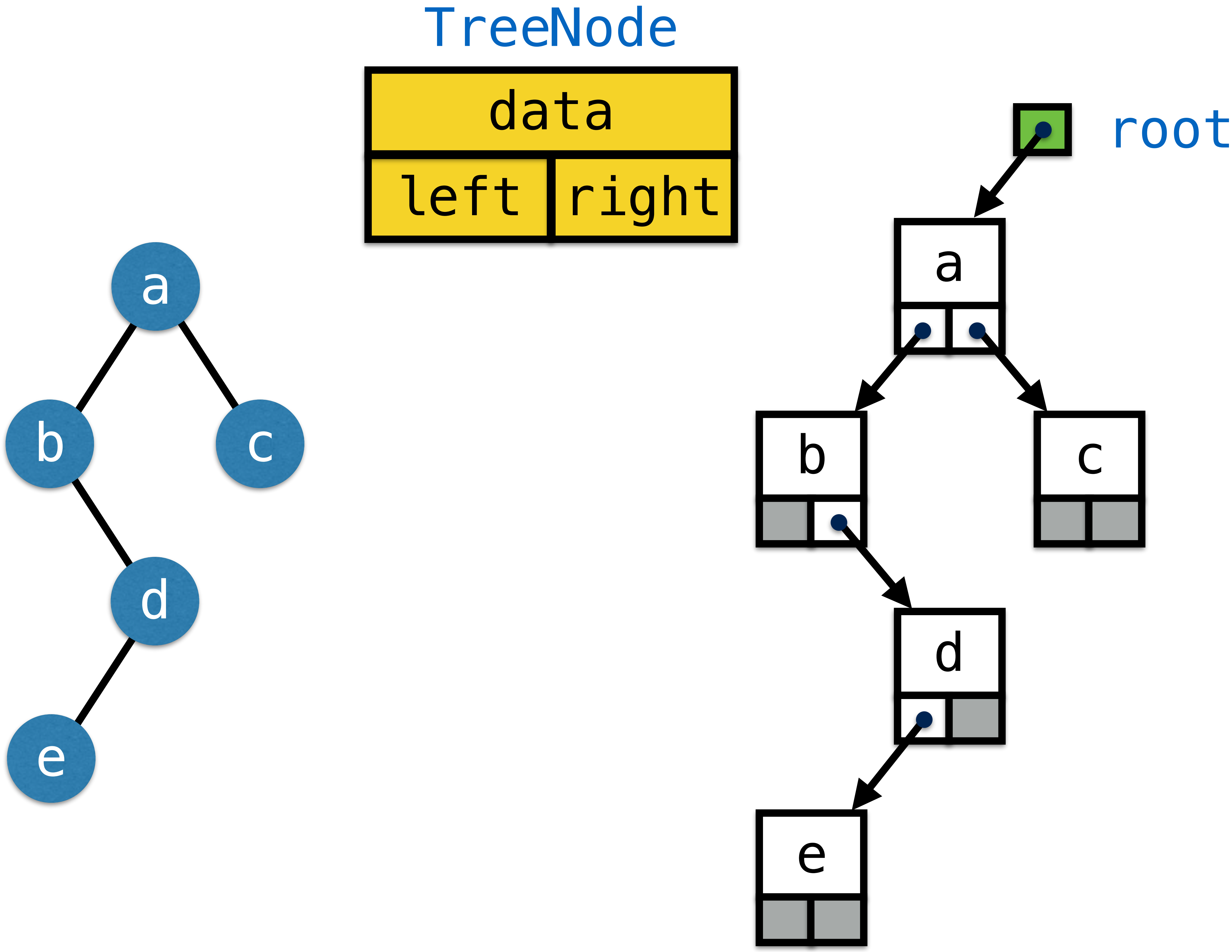
$(3-4)*(2+5)$


a
b
c
d
e

TreeNode






TreeNode
data
left
right
root
a
b
c
d
e
a
b
c
d
e

```java
public class TreeNode<E>
{
  protected TreeNode<E> left;
  protected TreeNode<E> right;
  protected E data;

  public TreeNode(){}

  public TreeNode(E data)
  {
    this(data, null, null);
  }

  public TreeNode(E data,
                  TreeNode<E> left,
                  TreeNode<E> right)
  {
    this.left = left;
    this.right = right;
    this.data = data;
  }
```

Tree Traversal

Tree Traversal

- A **traversal** is a way of **visiting** each node in a tree once.

Tree Traversal

- A **traversal** is a way of **visiting** each node in a tree once.
- What you do when visiting a node depends on the application. E.g., you may

Tree Traversal

- A **traversal** is a way of **visiting** each node in a tree once.

- What you do when visiting a node depends on the application. E.g., you may

  - print the data in the node, or

Tree Traversal

- A **traversal** is a way of **visiting** each node in a tree once.

- What you do when visiting a node depends on the application. E.g., you may

  - print the data in the node, or

  - perform a calculation.

```
preOrder(T):

  if T is empty
    return

  let T_left be the left subtree of T
  let T_right be the right subtree of T

  visit the root of T
  preOrder(T_left)
  preOrder(T_right)
```

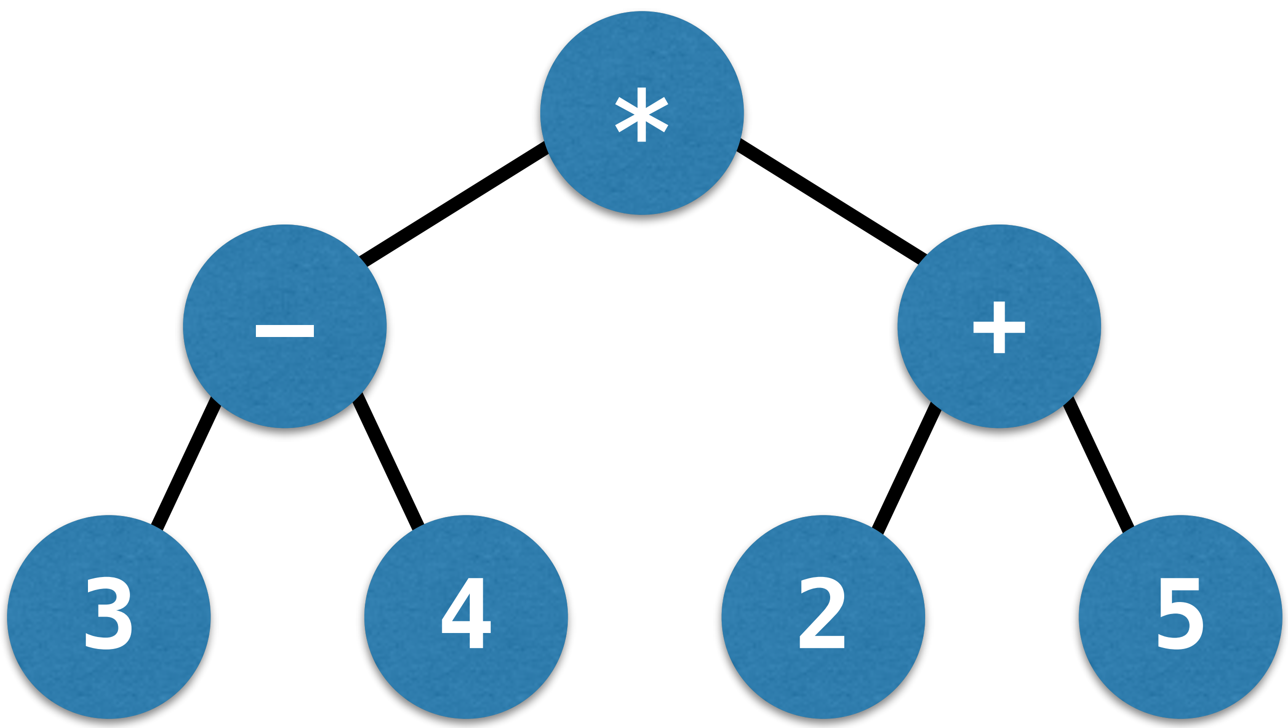
```
postOrder(T):

  if T is empty
     return

  let Tleft be the left subtree of T
  let Tright be the right subtree of T

  postOrder(Tleft)
  postOrder(Tright)
  visit the root of T
```

```
inOrder(T):

  if T is empty
    return

  let T_left be the left subtree of T
  let T_right be the right subtree of T

  inOrder(T_left)
  visit the root of T
  inOrder(T_right)
```

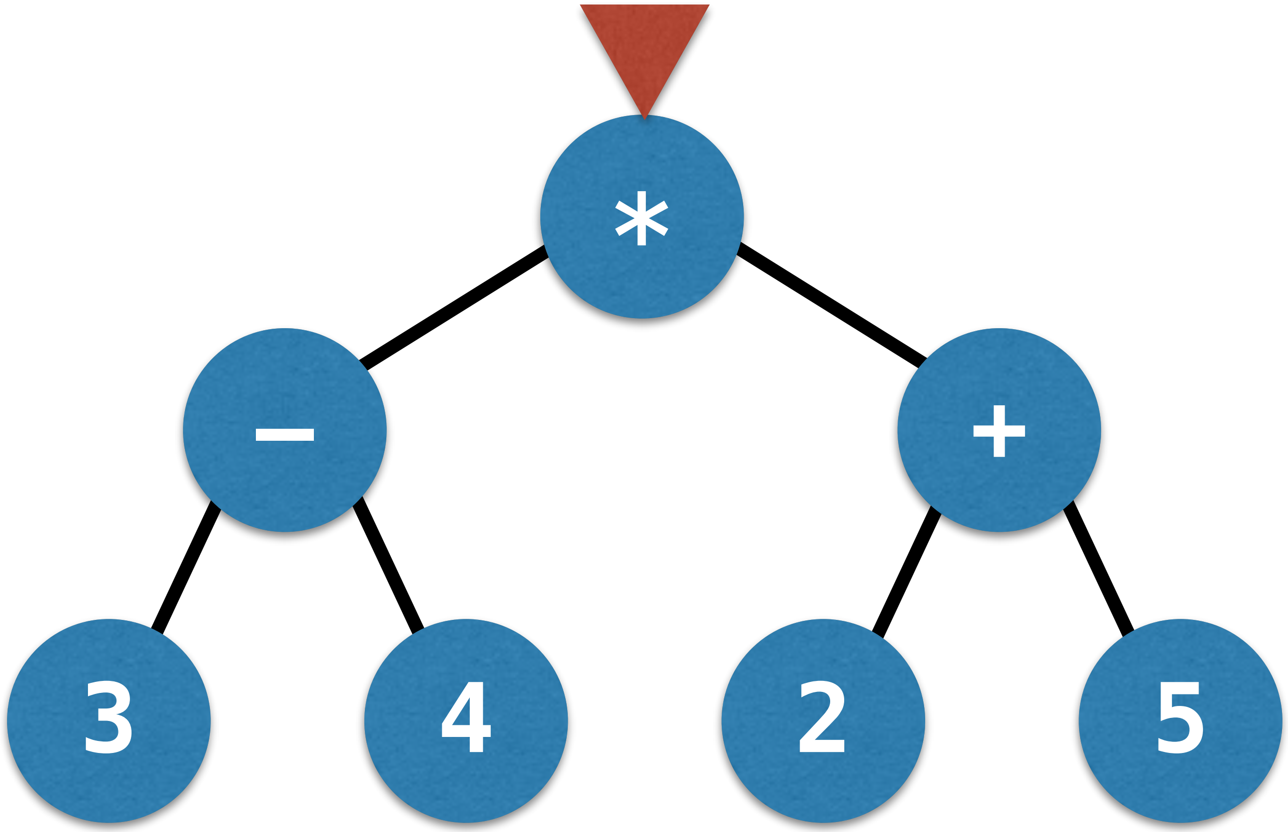
Pre-order in Java

```java
public static void traversePreorder(TreeNode<?> node)
 {
   if (node == null) return;
   System.out.print(node.data().toString() + " ");
   traversePreorder(node.left());
   traversePreorder(node.right());
 }
```

Pre-order

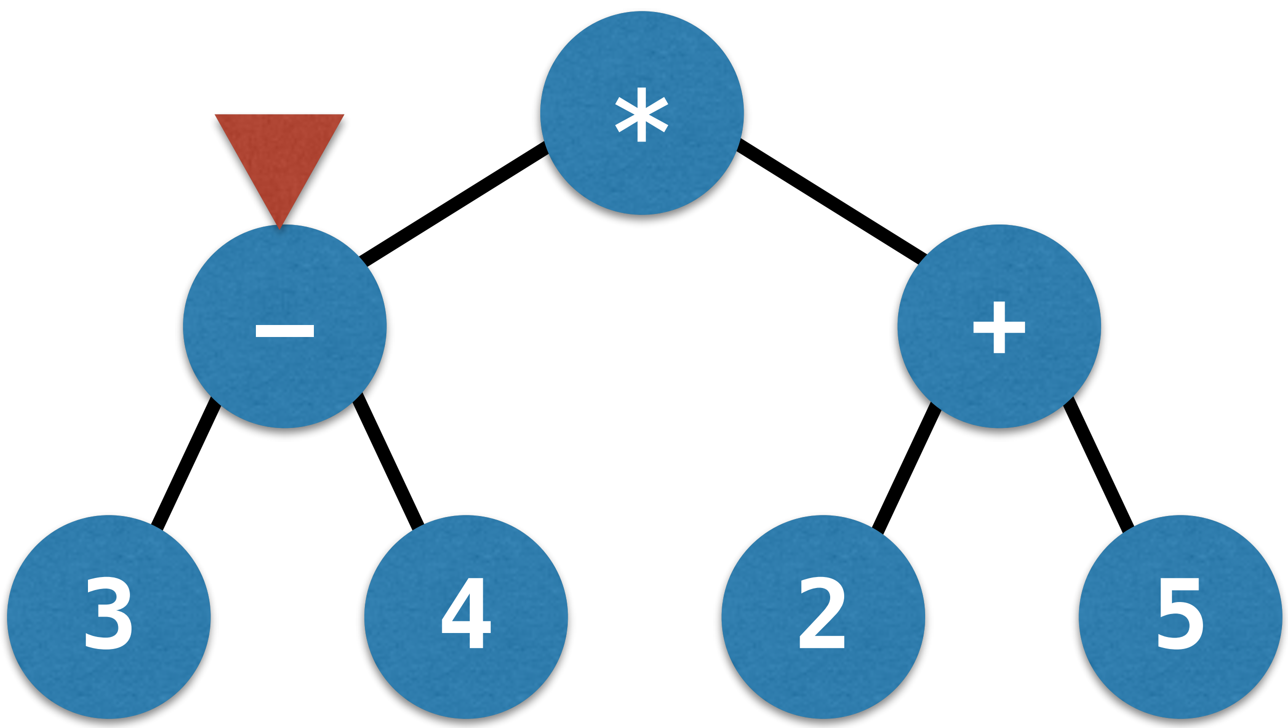


Pre-order



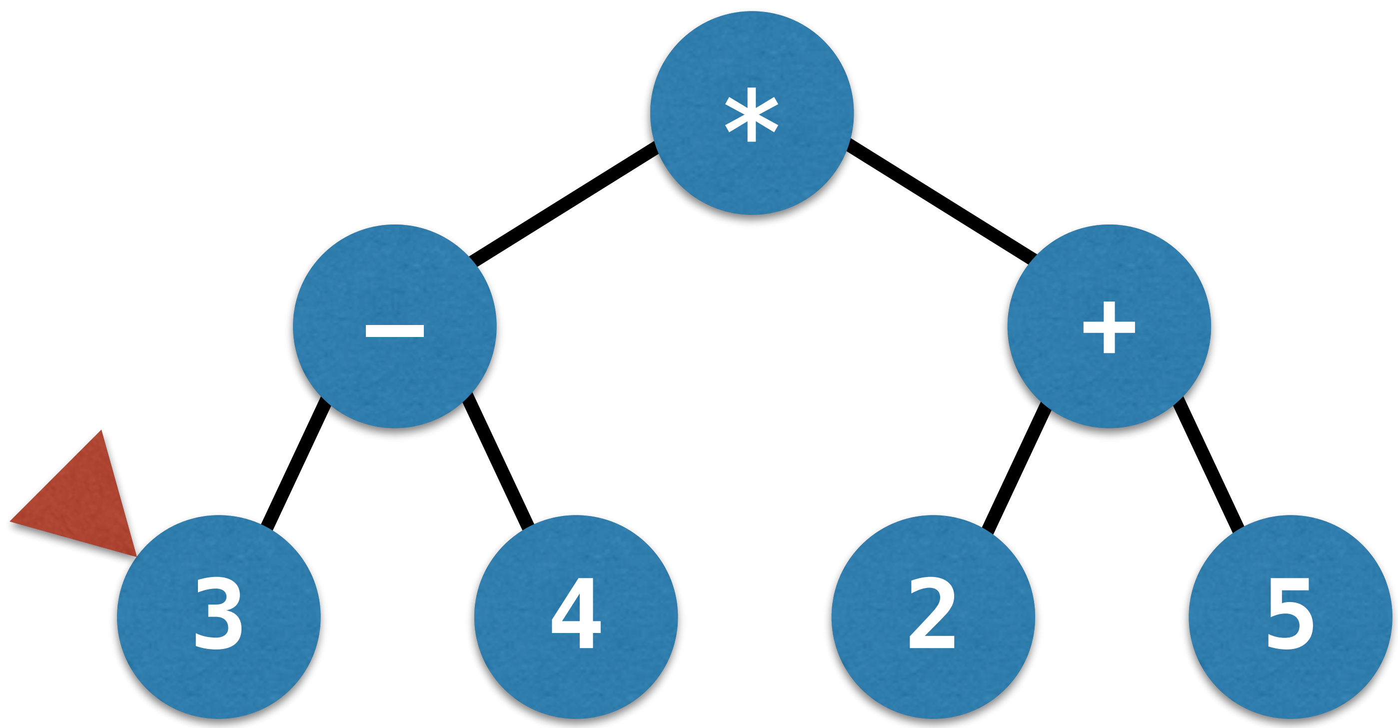
*

Pre-order



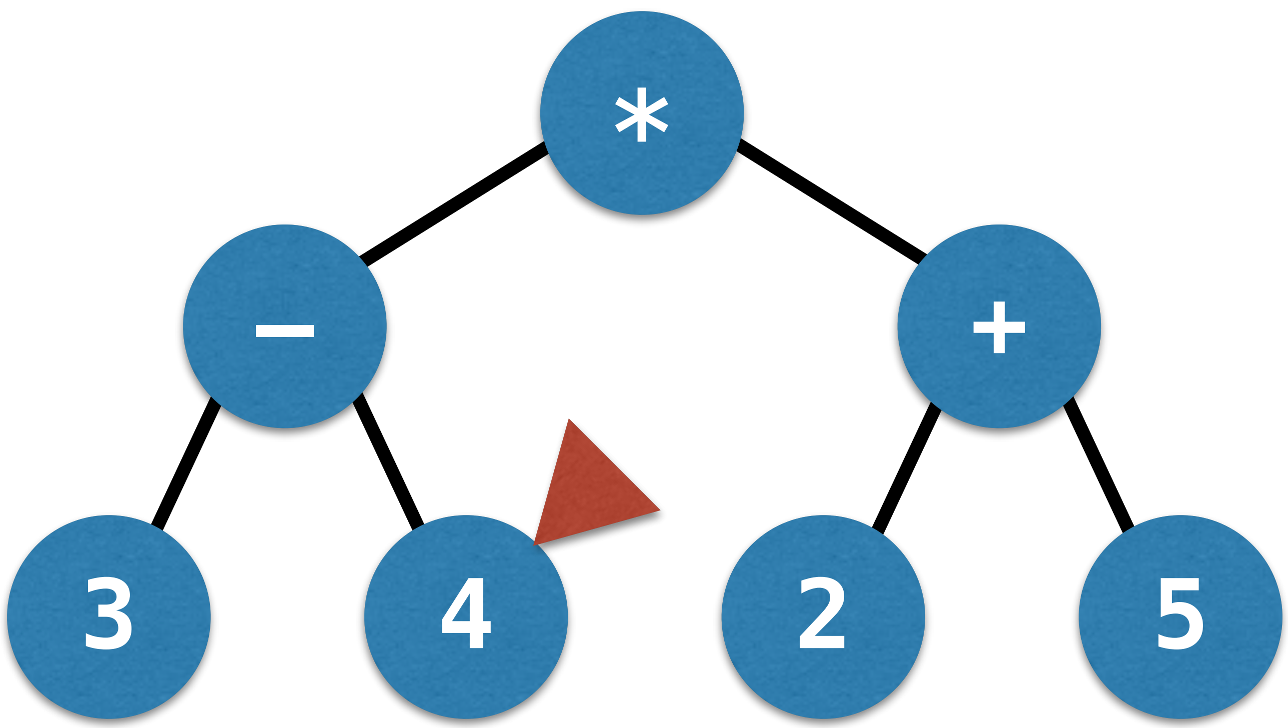
* −

Pre-order



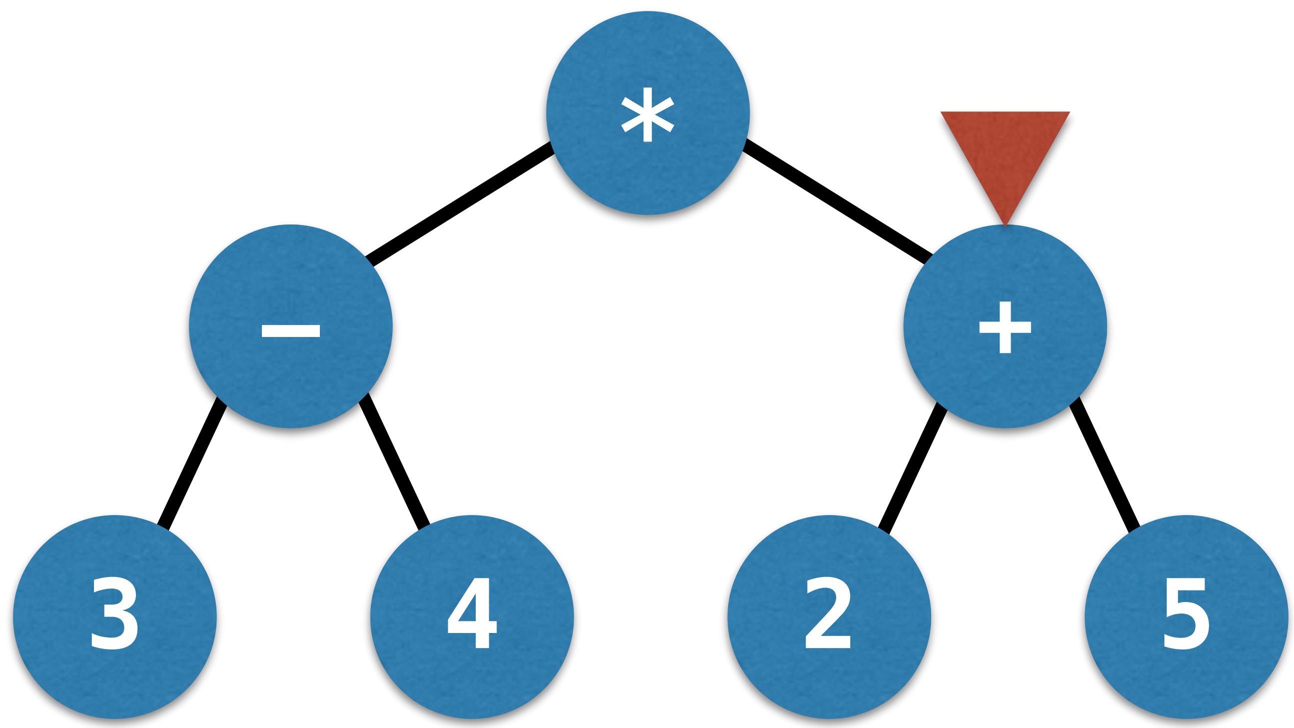
$* \; - \; 3$

Pre-order



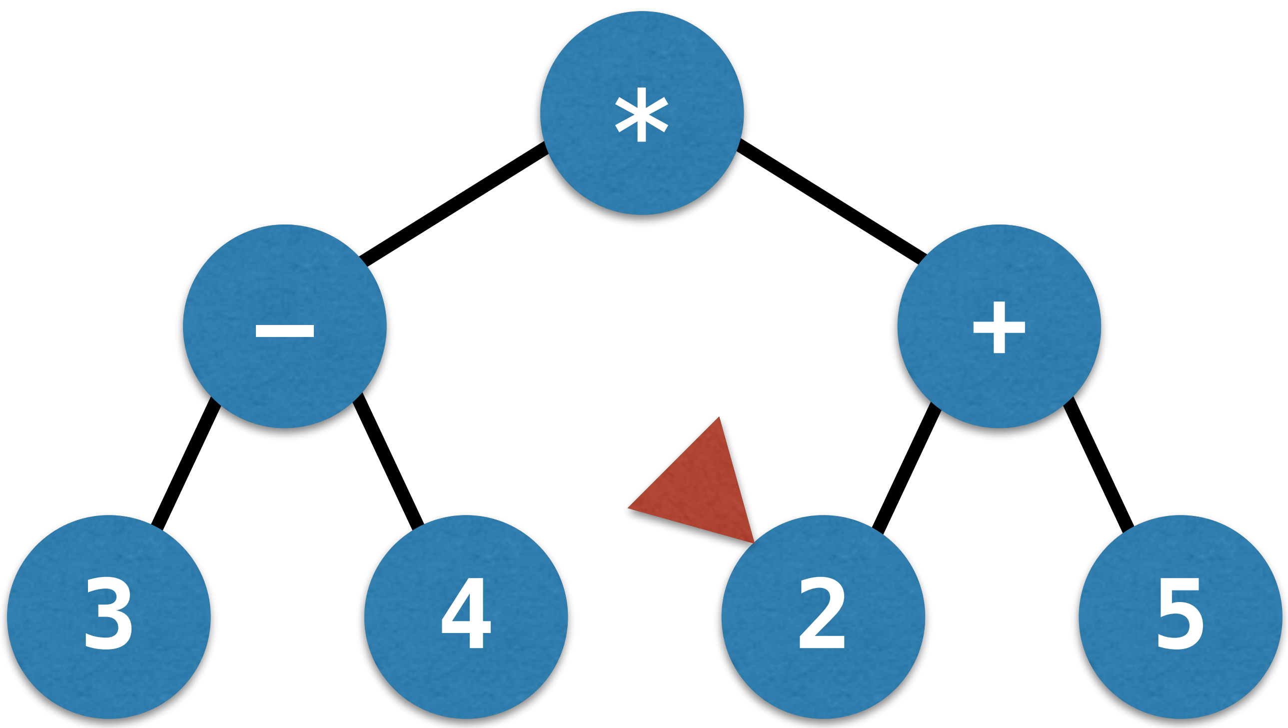
∗ − 3 4

Pre-order



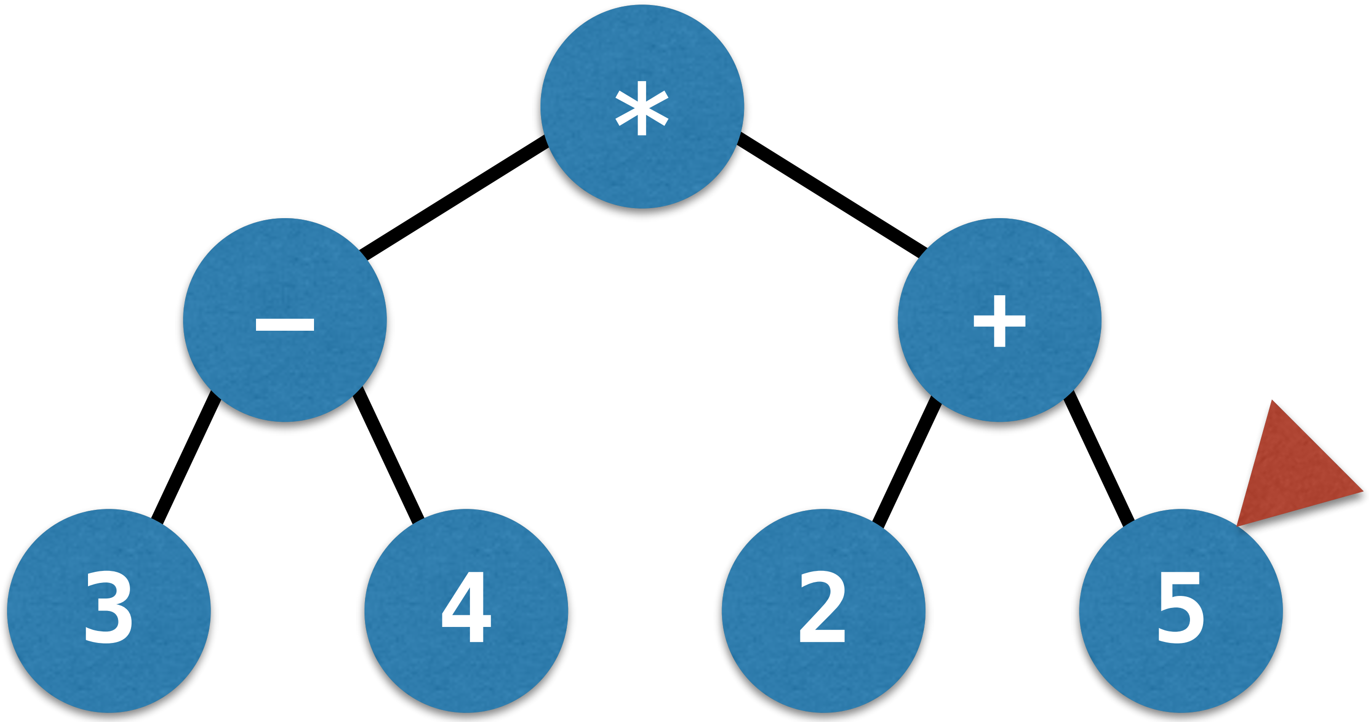
∗ − 3 4 +

Pre-order



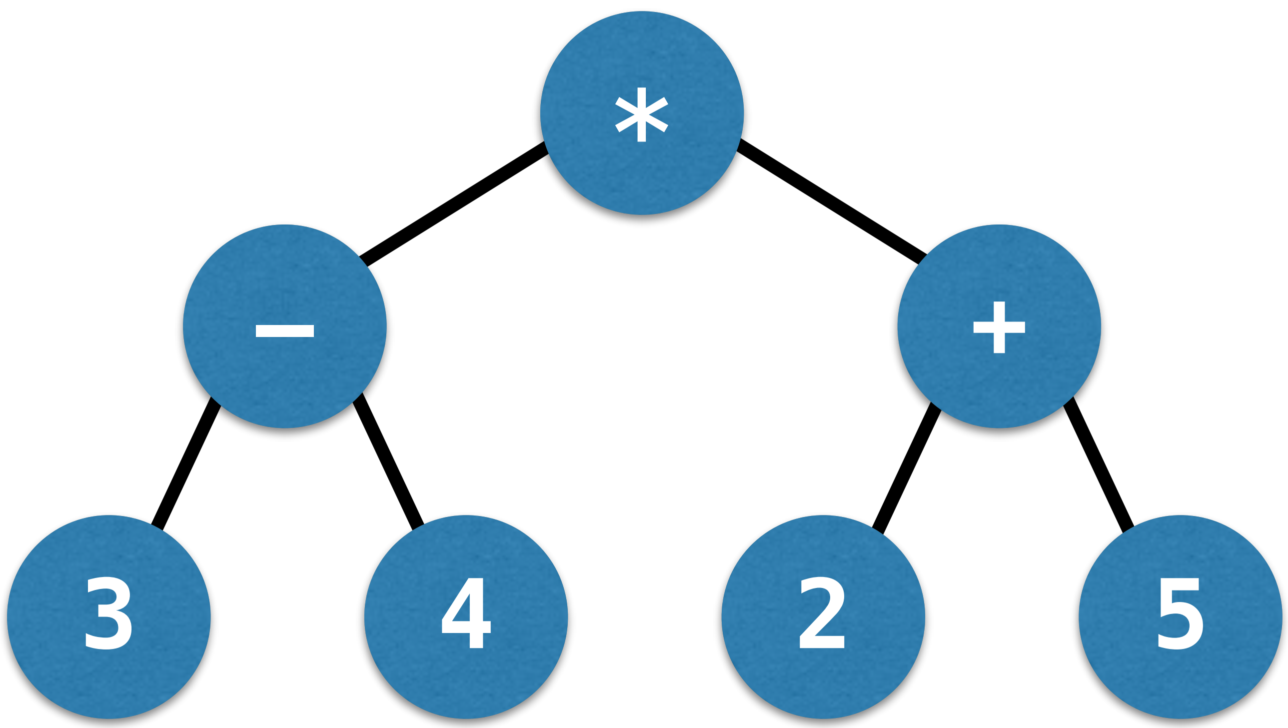
* − 3 4 + 2

Pre-order

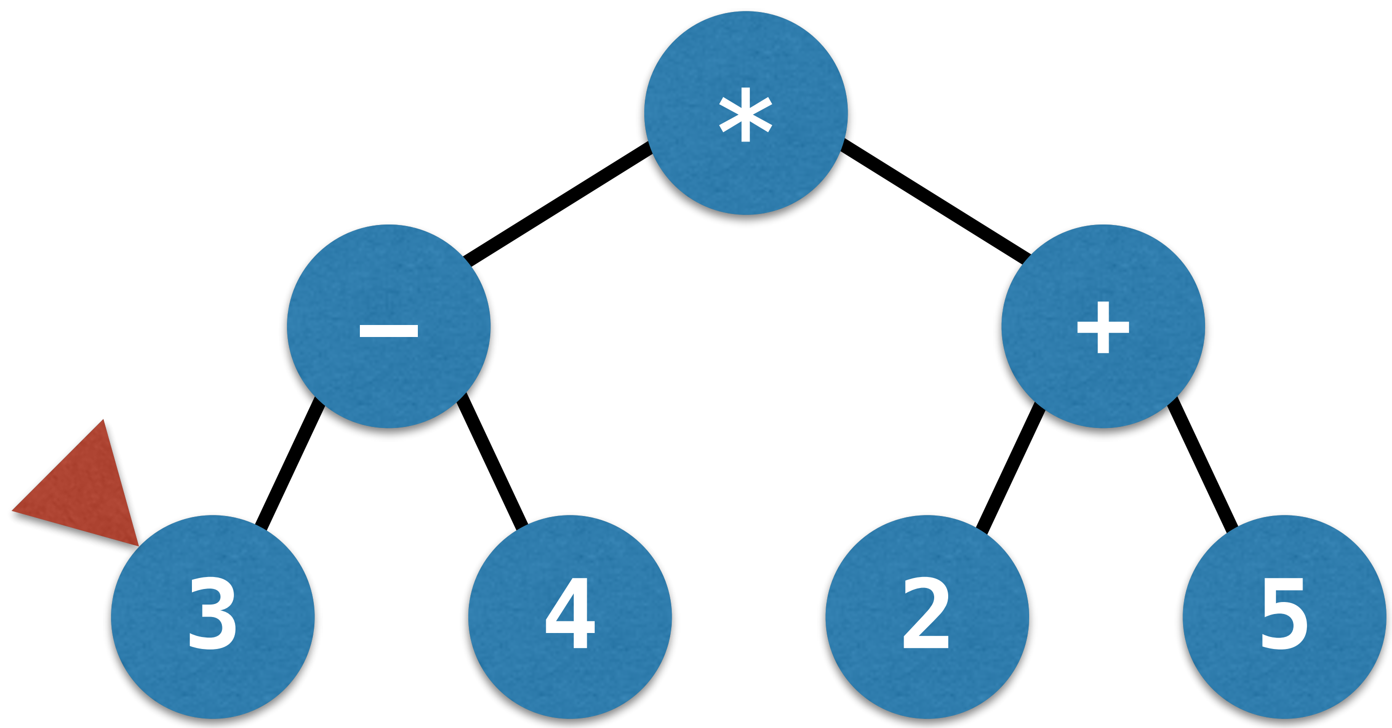


* − 3 4 + 2 5

Post-order

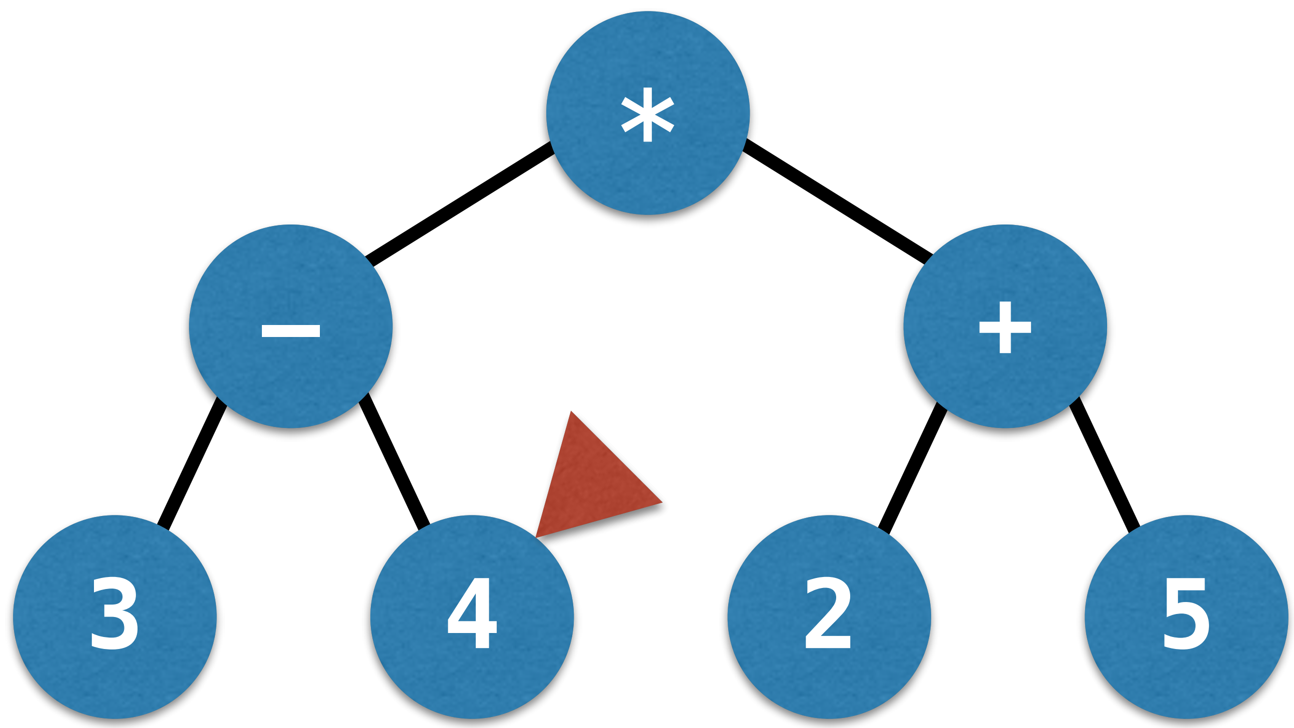


Post-order



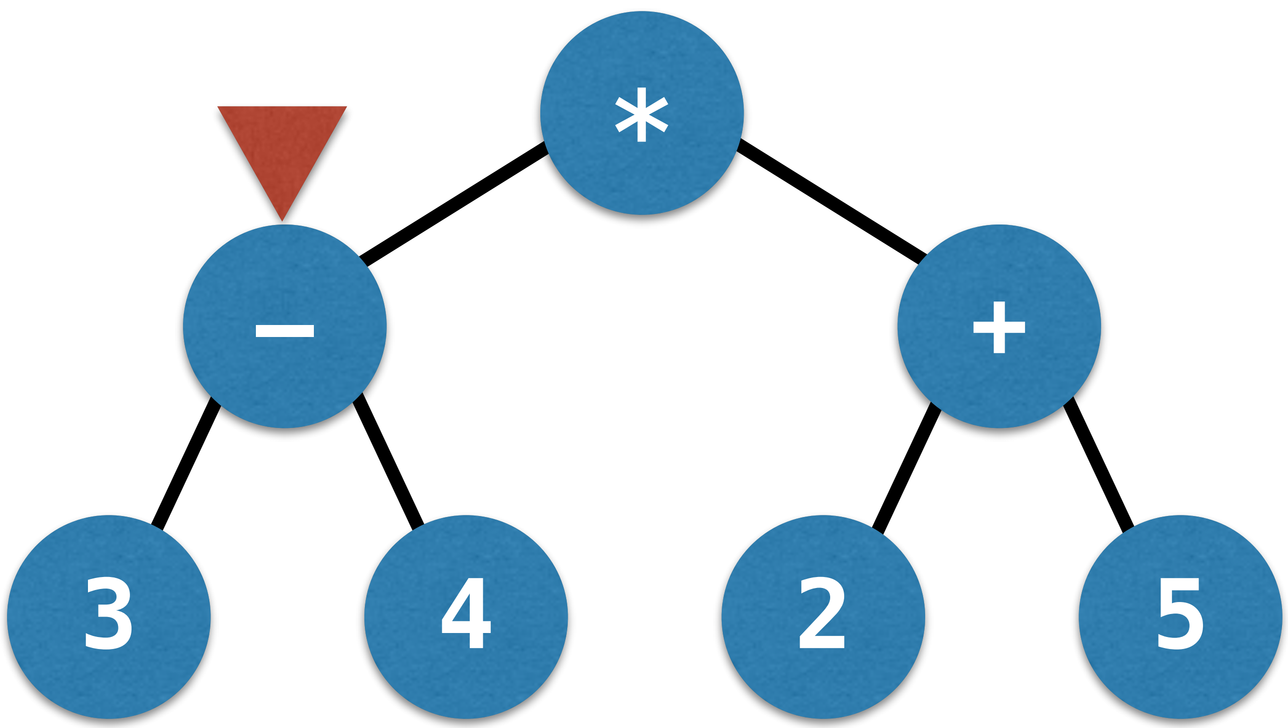
3

Post-order



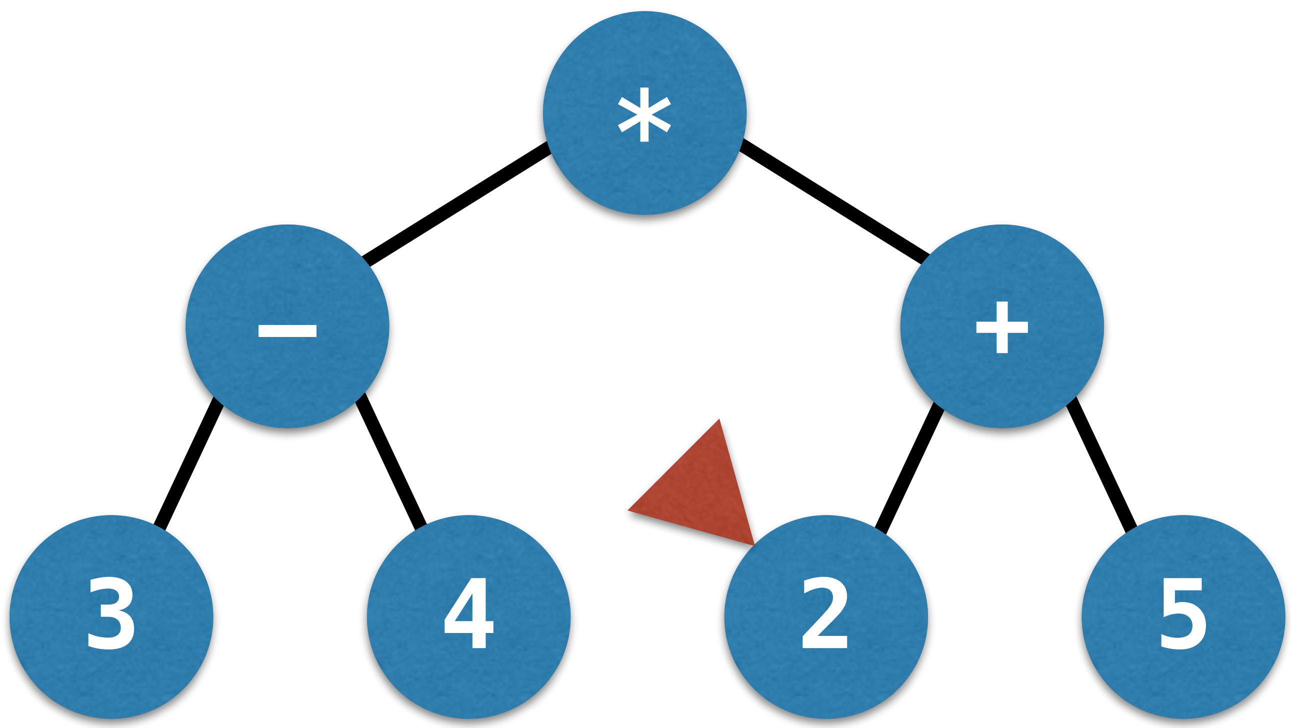
3 4

Post-order



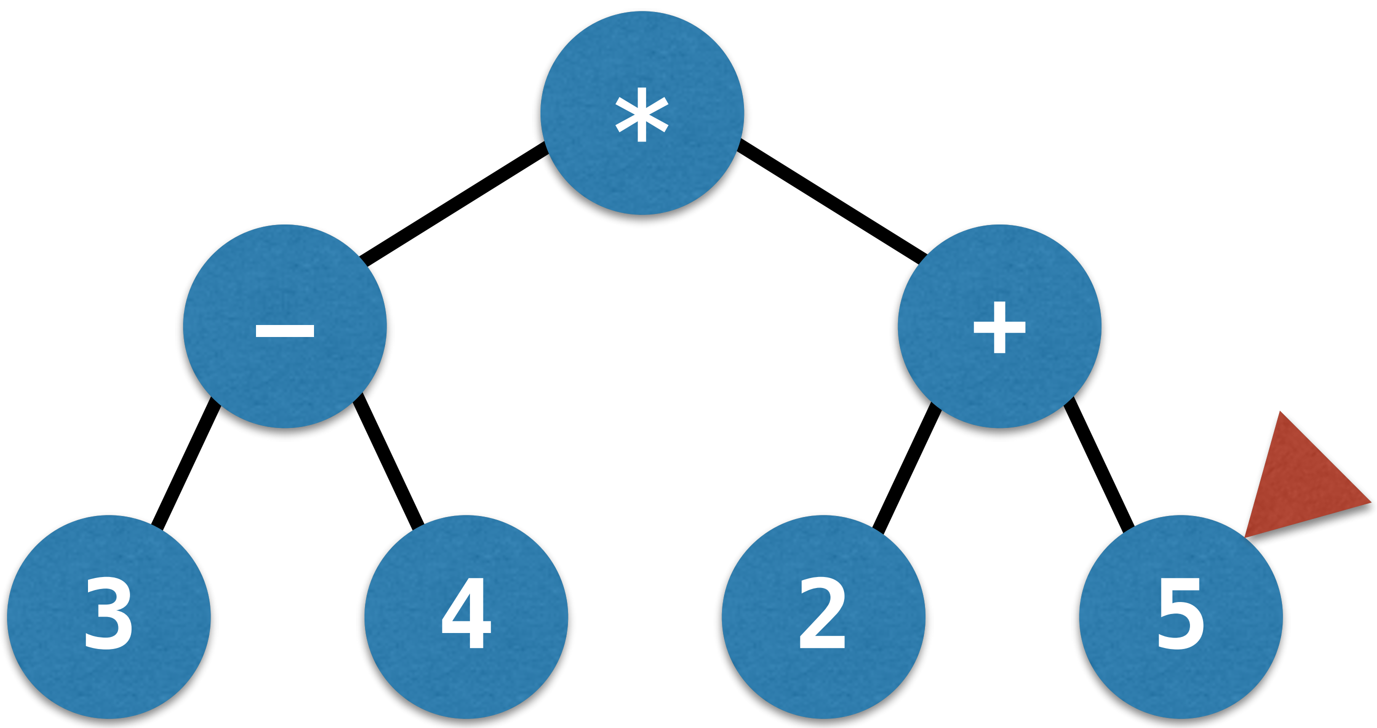
3 4 –

Post-order



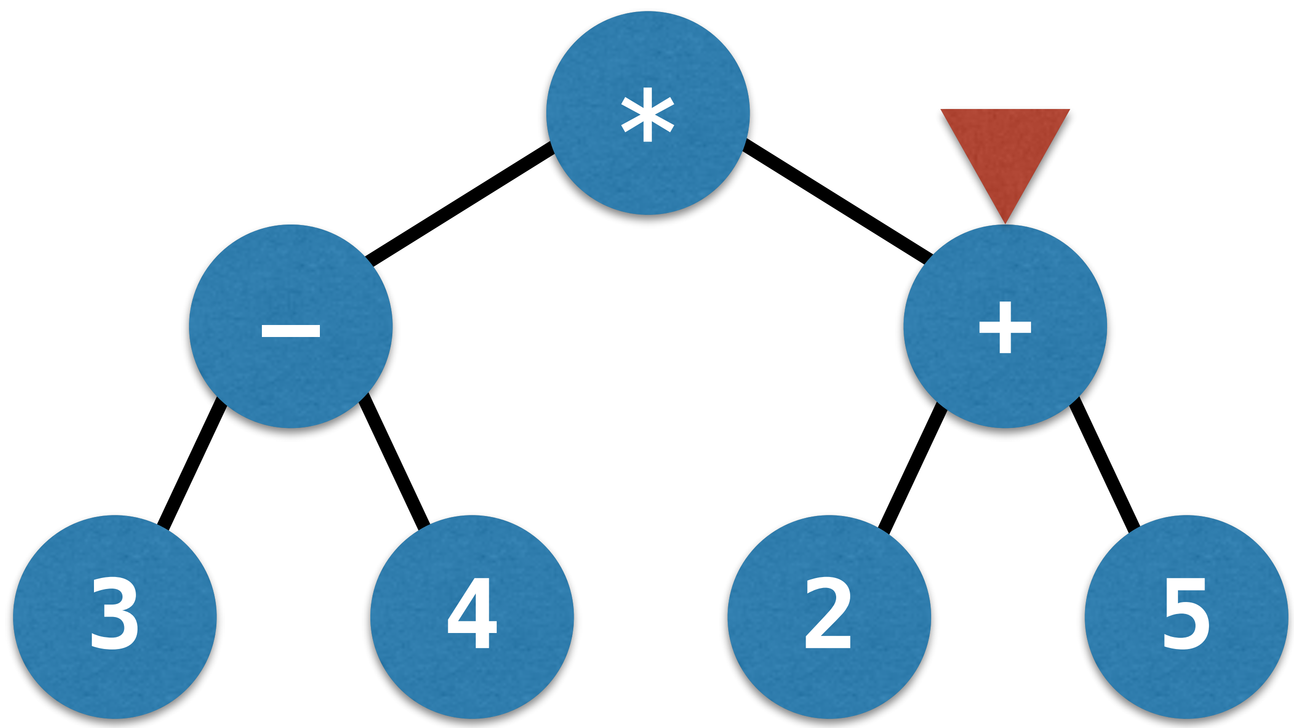
3 4 – 2

Post-order



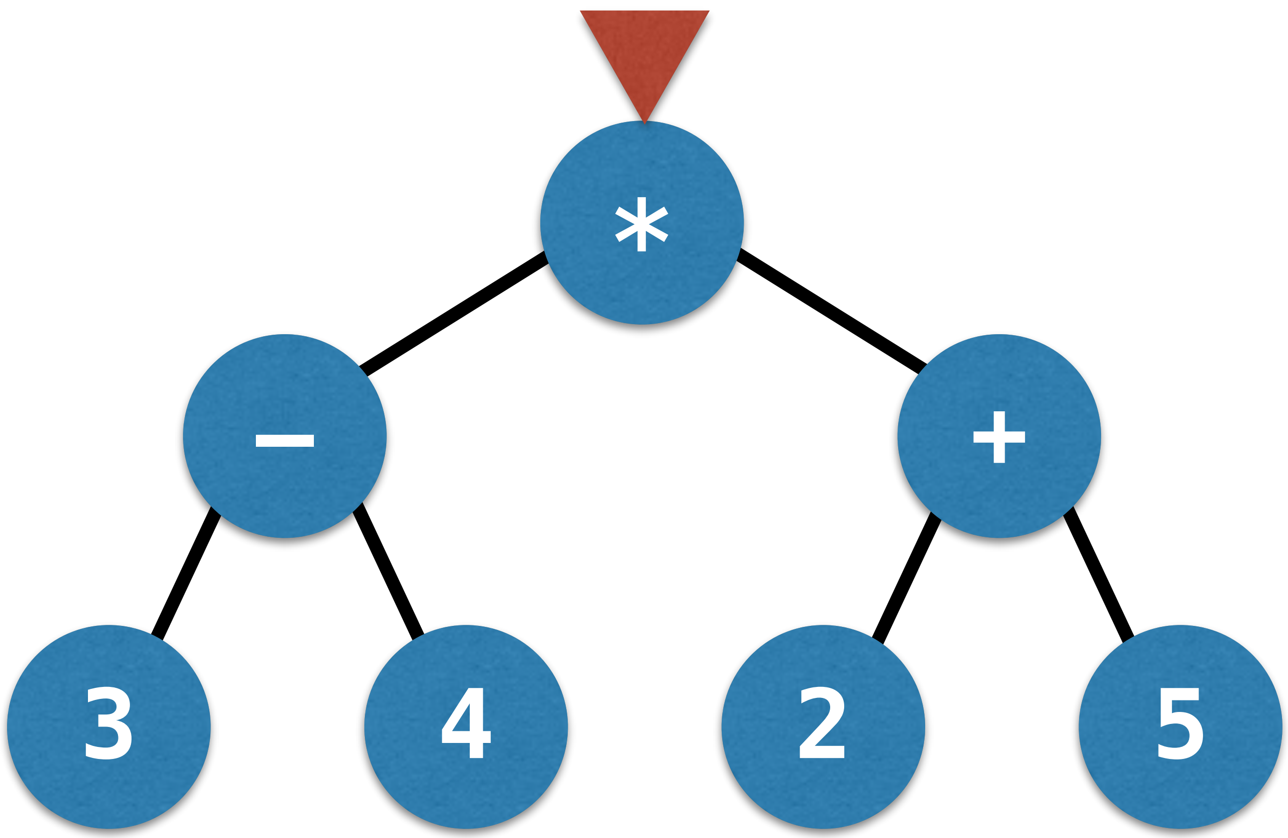
3 4 – 2 5

Post-order



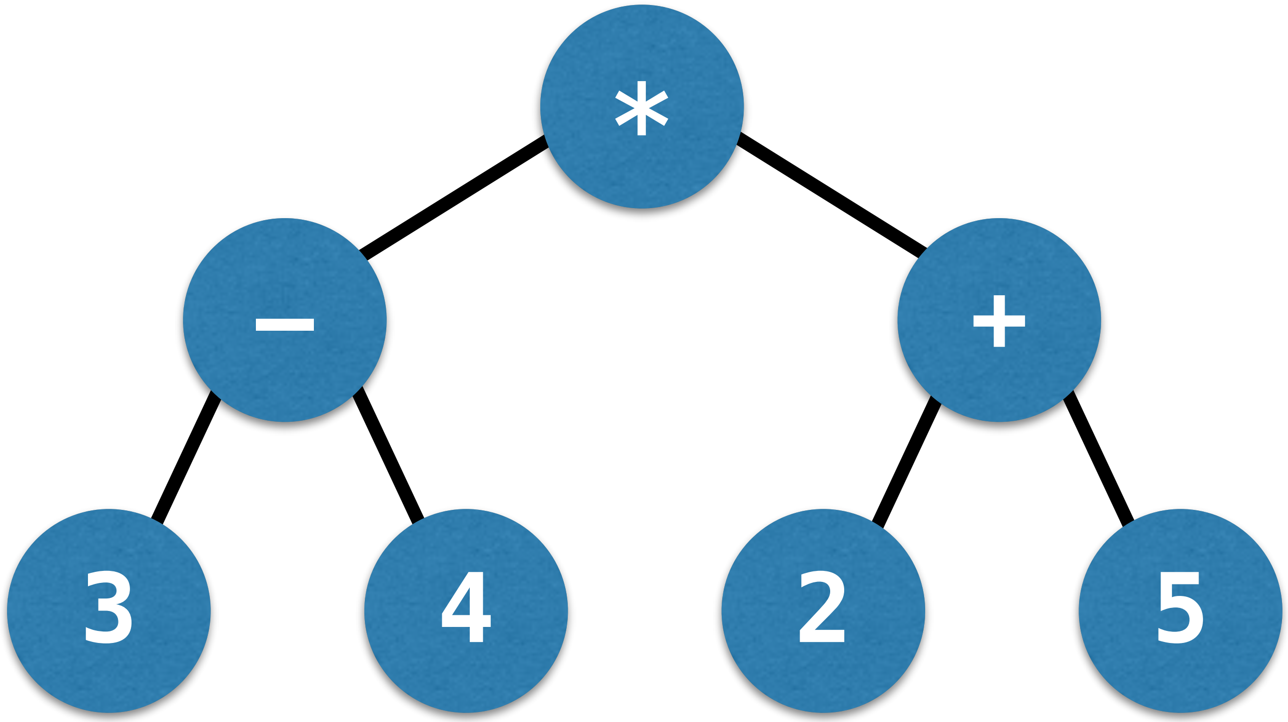
3 4 – 2 5 +

Post-order

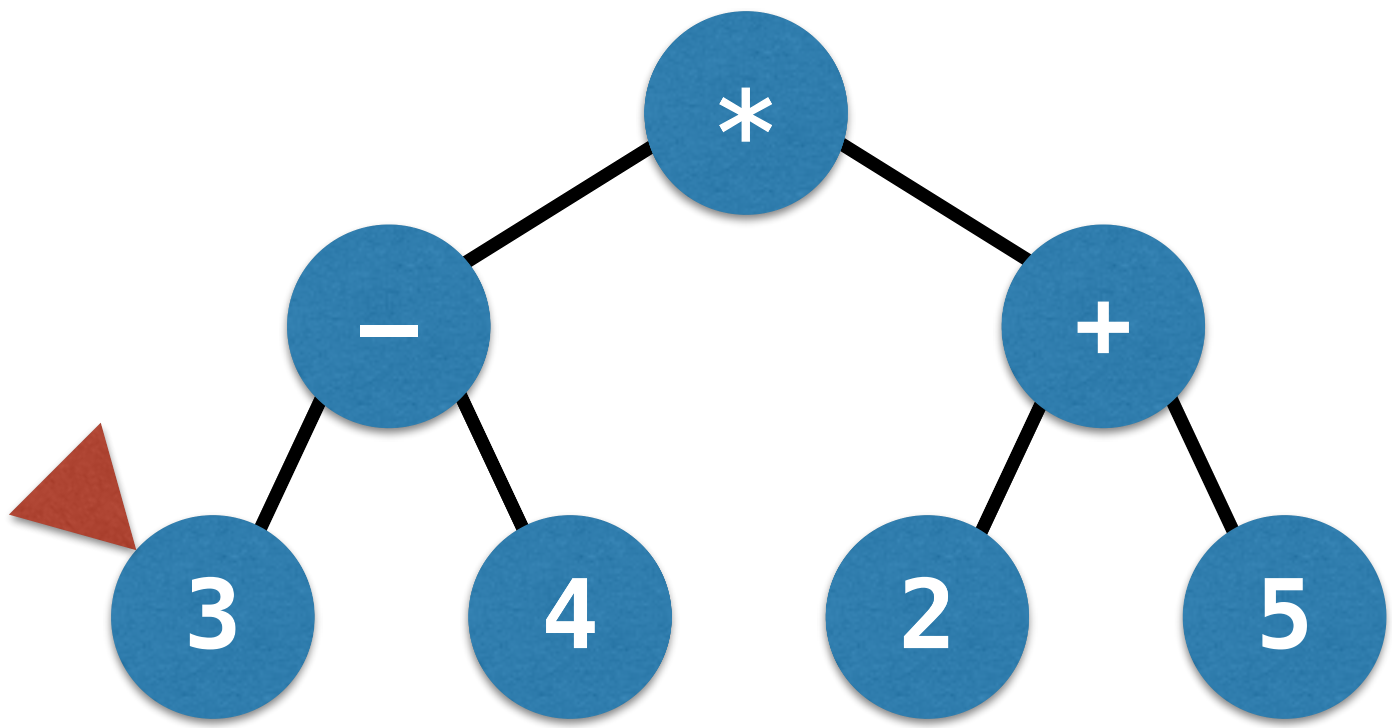


3 4 – 2 5 + *

In-order

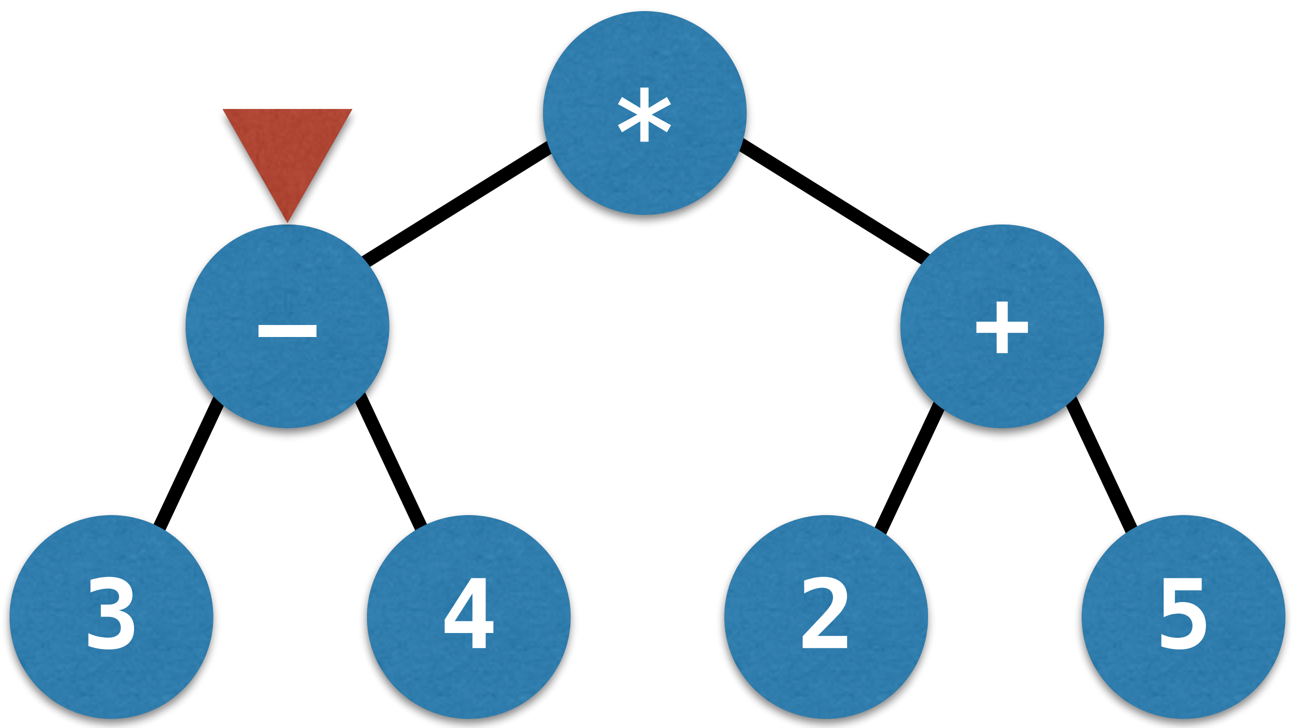


In-order



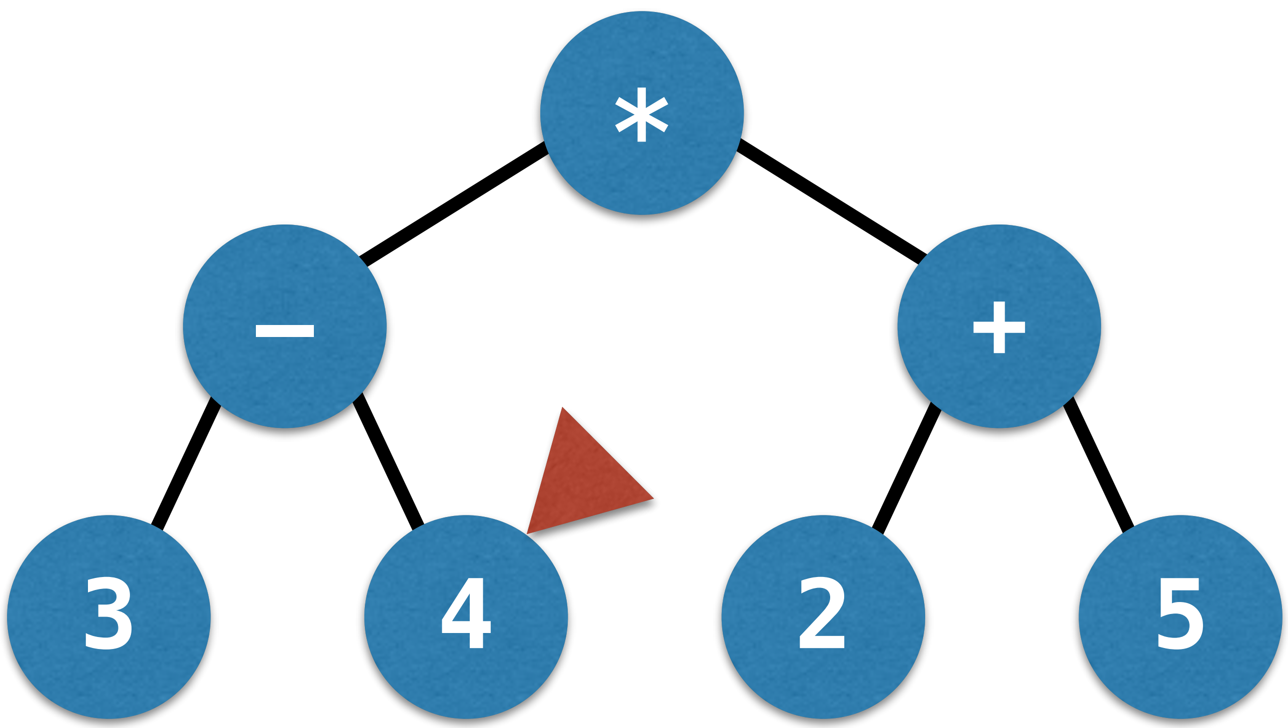
3

In-order



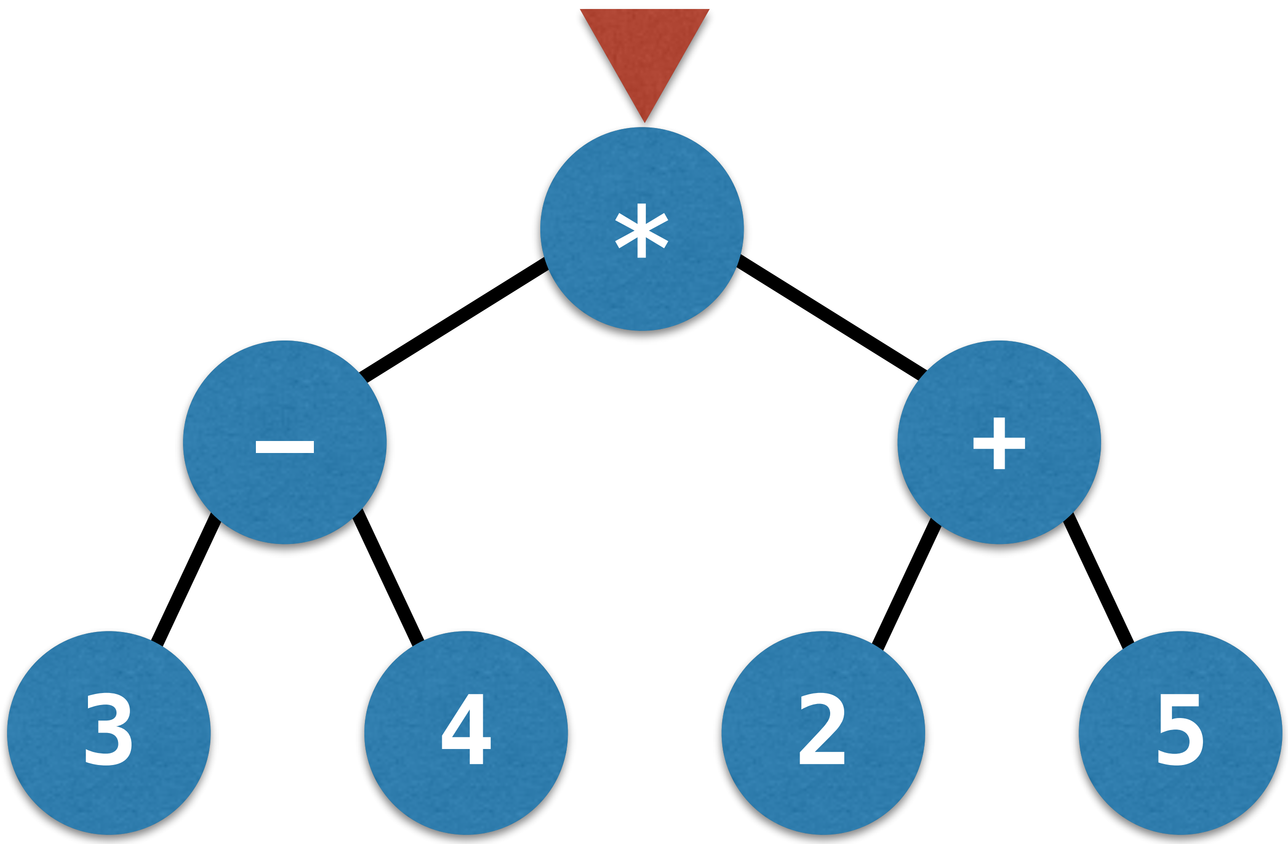
3 –

In-order



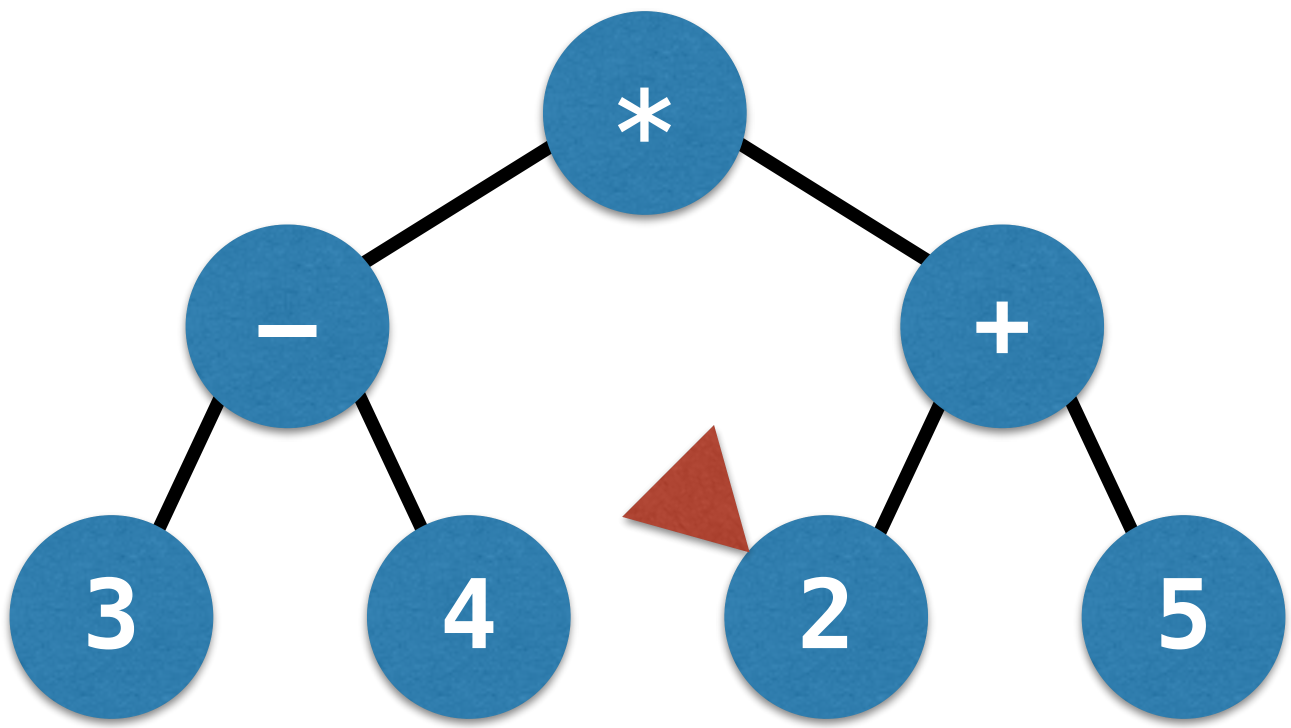
3 – 4

In-order



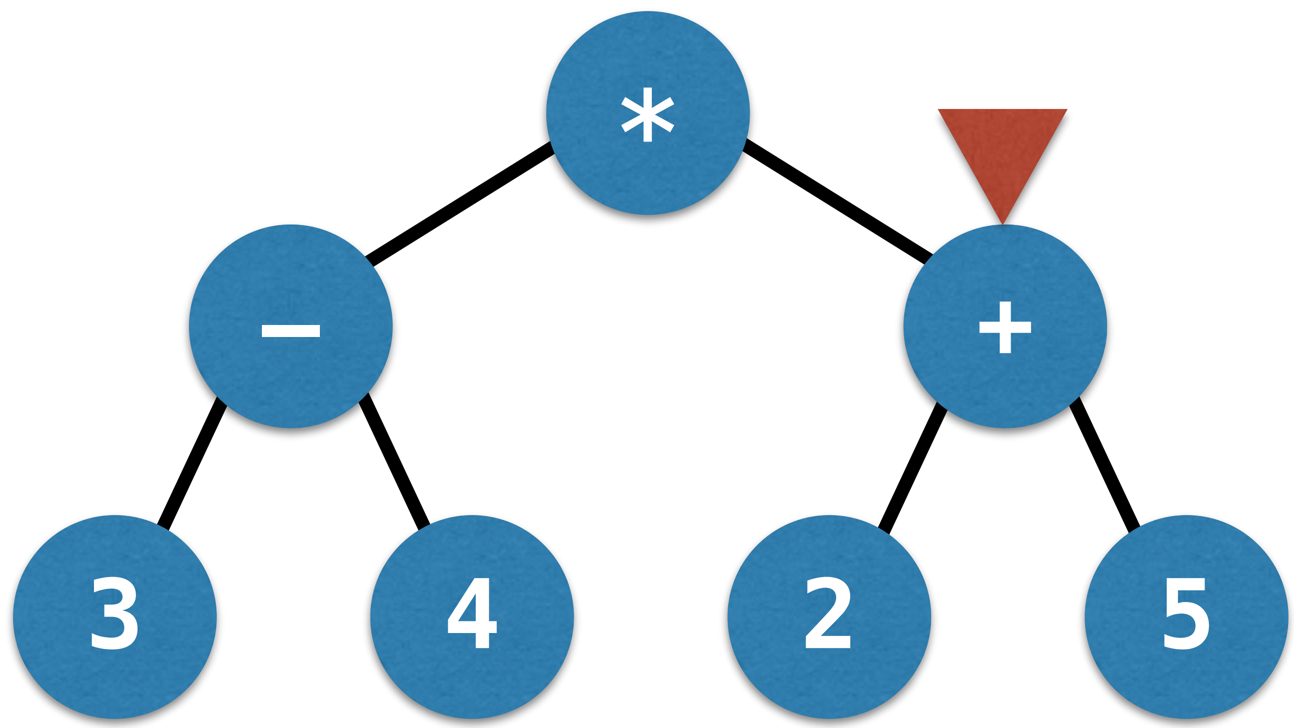
3 − 4 *

In-order



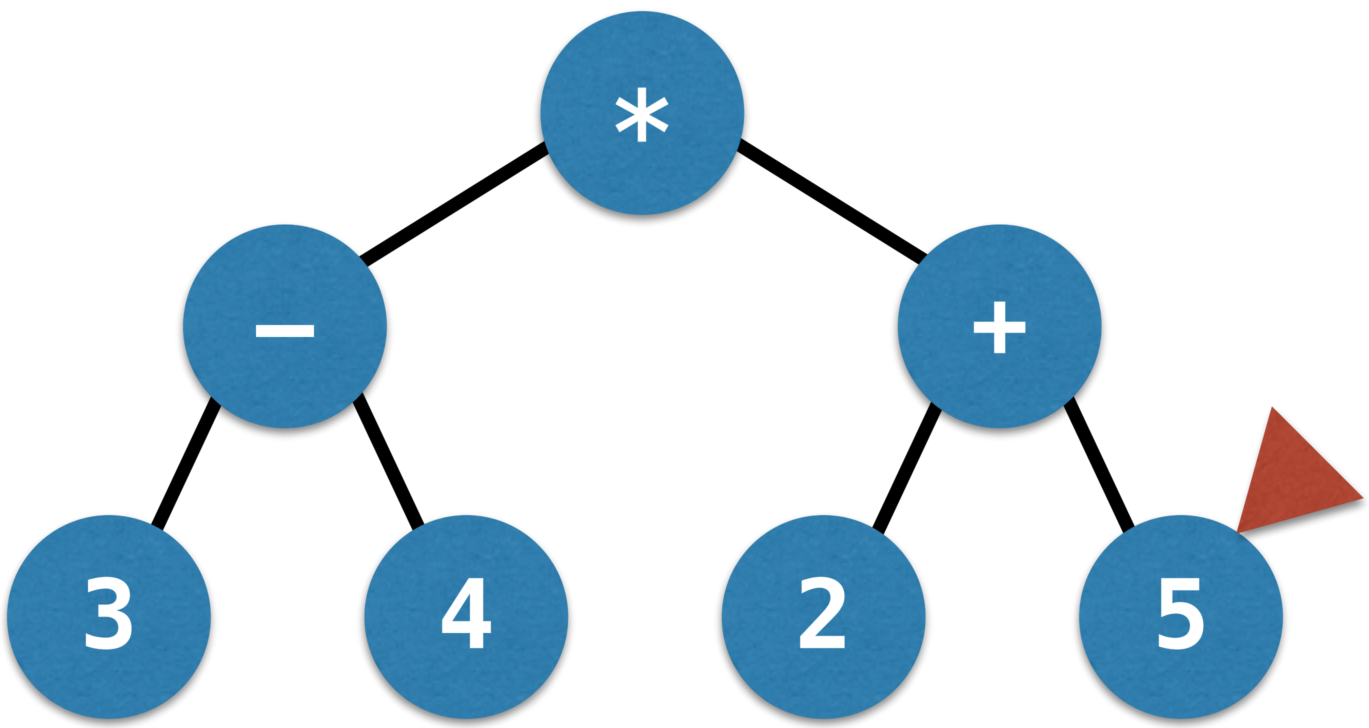
3 – 4 ∗ 2

In-order

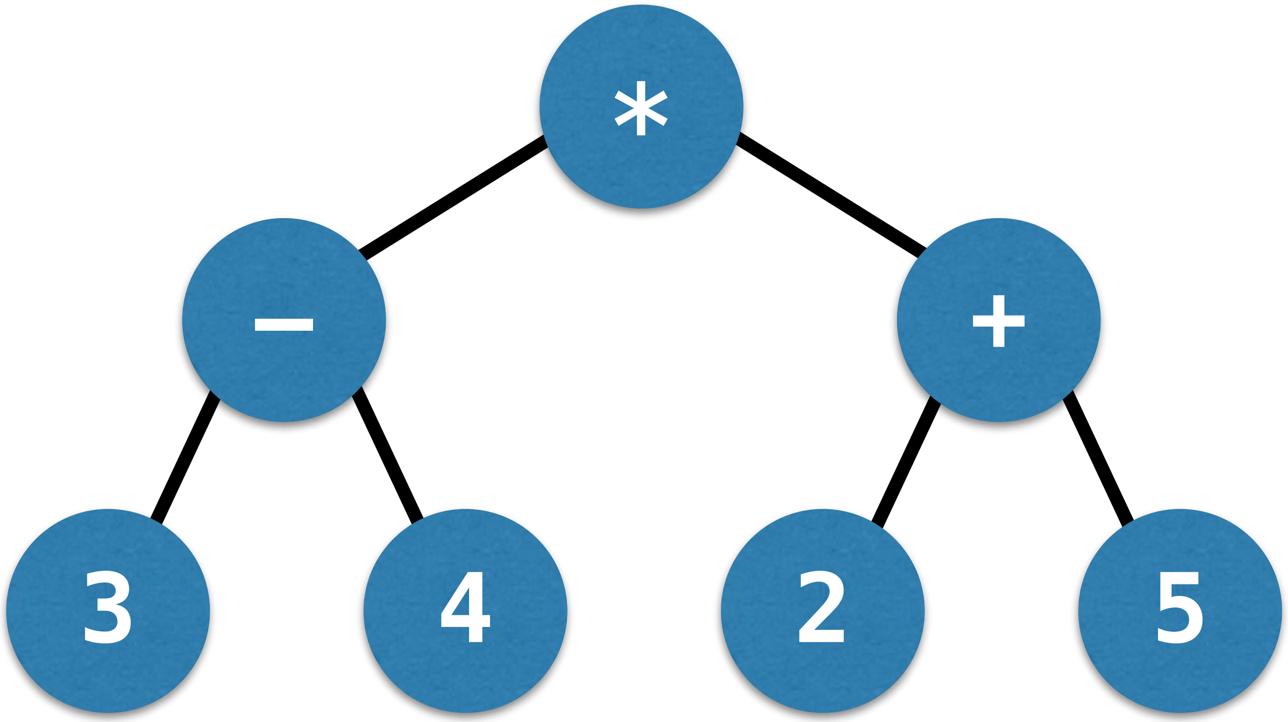


3 − 4 ∗ 2 +

In-order



$3 - 4 * 2 + 5$



Pre: ∗ − 3 4 + 2 5

In: 3 − 4 ∗ 2 + 5

Post: 3 4 − 2 5 + ∗

Computing the height of a binary tree

Computing the height of a binary tree

- The **height** of a tree T is the length of the longest path from the root of T to a leaf.

Computing the height of a binary tree

- The **height** of a tree T is the length of the longest path from the root of T to a leaf.
- Recursive definition:

Computing the height of a binary tree

- The **height** of a tree T is the length of the longest path from the root of T to a leaf.

- Recursive definition:

  - If T is empty, then height(T) = −1.

Computing the height of a binary tree

- The **height** of a tree T is the length of the longest path from the root of T to a leaf.

- Recursive definition:

  - If T is empty, then height(T) = −1.

  - Otherwise,

Computing the height of a binary tree

- The **height** of a tree T is the length of the longest path from the root of T to a leaf.
- Recursive definition:
  - If T is empty, then $\text{height}(T) = -1$.
  - Otherwise,
    - Let T_{left} be the left subtree of T.

Computing the height of a binary tree

- The **height** of a tree T is the length of the longest path from the root of T to a leaf.
- Recursive definition:
  - If T is empty, then height(T) = −1.
  - Otherwise,
    - Let T_{left} be the left subtree of T.
    - Let T_{right} be the right subtree of T.

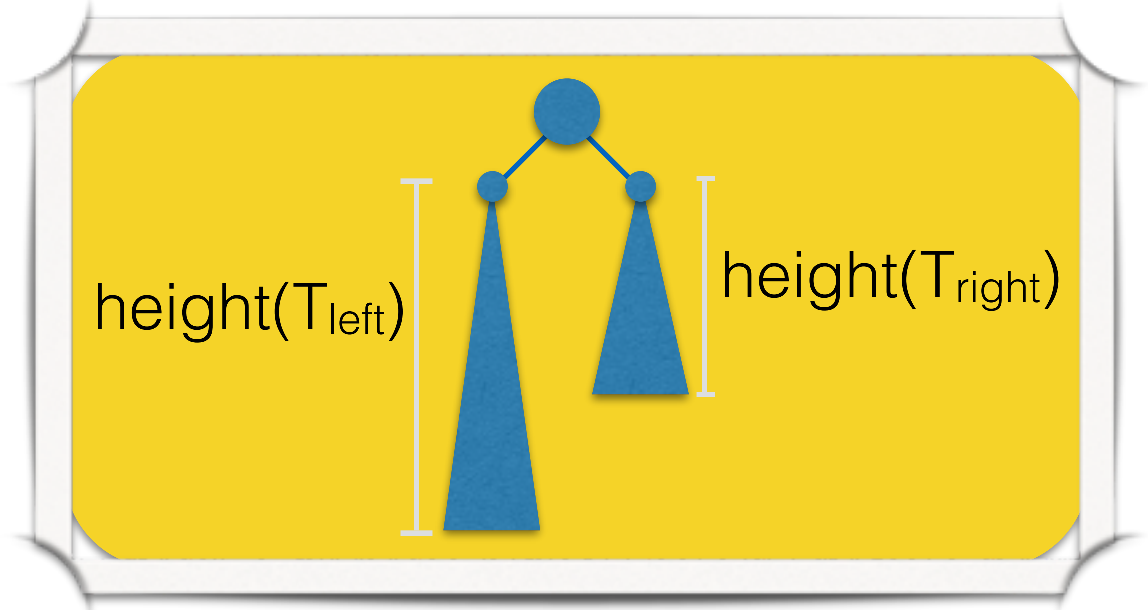
Computing the height of a binary tree

- The **height** of a tree T is the length of the longest path from the root of T to a leaf.
- Recursive definition:
  - If T is empty, then height(T) = –1.
  - Otherwise,
    - Let T_{left} be the left subtree of T.
    - Let T_{right} be the right subtree of T.
    - height(T) = 1 + max {height(T_{left}), height (T_{right})}.

Computing the height of a binary tree

- The **height** of a tree T is the length of the longest path from the root of T to a leaf.

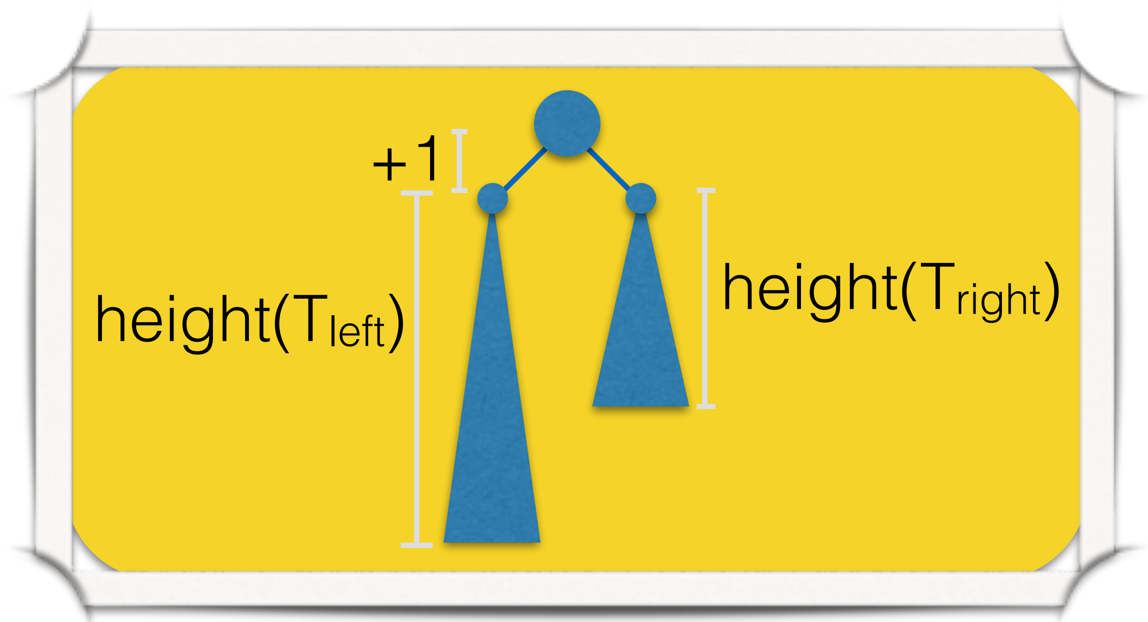
- Recursive definition:
  - If T is empty, then $\text{height}(T) = -1$.
  - Otherwise,
    - Let T_{left} be the left subtree of T.
    - Let T_{right} be the right subtree of T.
    - $\text{height}(T) = 1 + \max\{\text{height}(T_{left}), \text{height}(T_{right})\}$.



Computing the height of a binary tree

- The **height** of a tree T is the length of the longest path from the root of T to a leaf.

- Recursive definition:
  - If T is empty, then height(T) = –1.
  - Otherwise,
    - Let T_{left} be the left subtree of T.
    - Let T_{right} be the right subtree of T.
    - height(T) = 1 + max {height(T_{left}), height (T_{right})}.



```java
public static int height(TreeNode<?> node)
{
  if (node == null)
    return -1;

  int lHeight = height(node.left());
  int rHeight = height(node.right());

  if (lHeight > rHeight)
    return lHeight + 1;
  else
    return rHeight + 1;
}
```

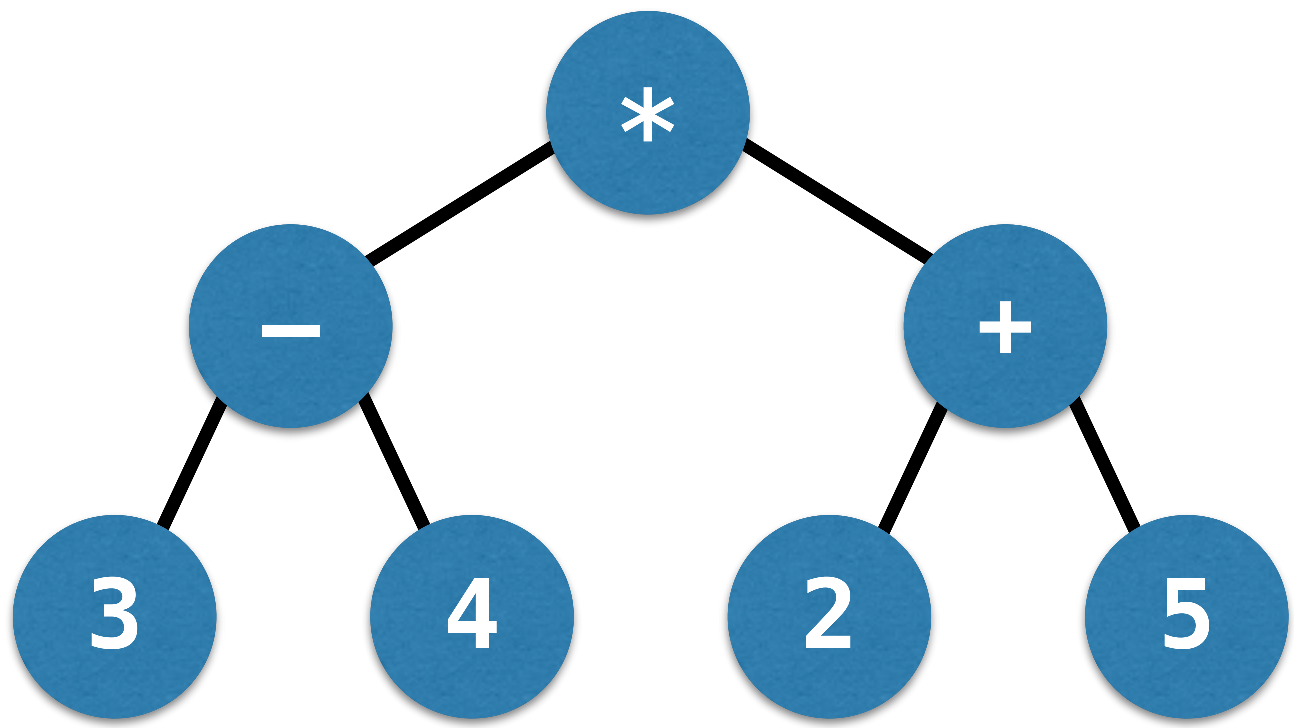
```java
public static int height(TreeNode<?> node)
{
  if (node == null)
    return -1;

  int lHeight = height(node.left());
  int rHeight = height(node.right());

  if (lHeight > rHeight)
    return lHeight + 1;
  else
    return rHeight + 1;
}
```

Post-order

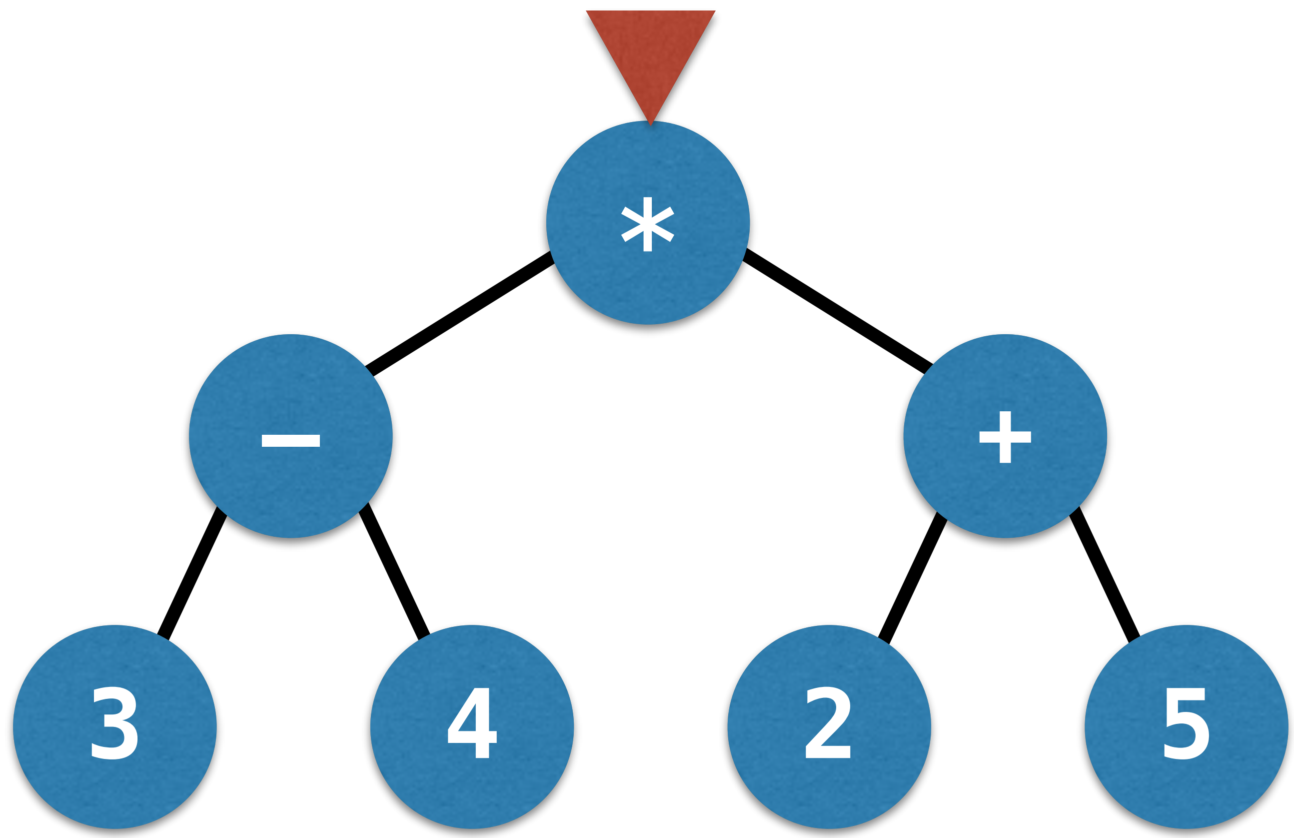
Level Order

- Visit the root (depth 0).
- Visit all nodes at depth 1 from left to right.
- Visit all nodes at depth 2 from left to right.
- Continue visiting nodes level by level, and left-to-right within each level until all nodes are visited.

Level Order

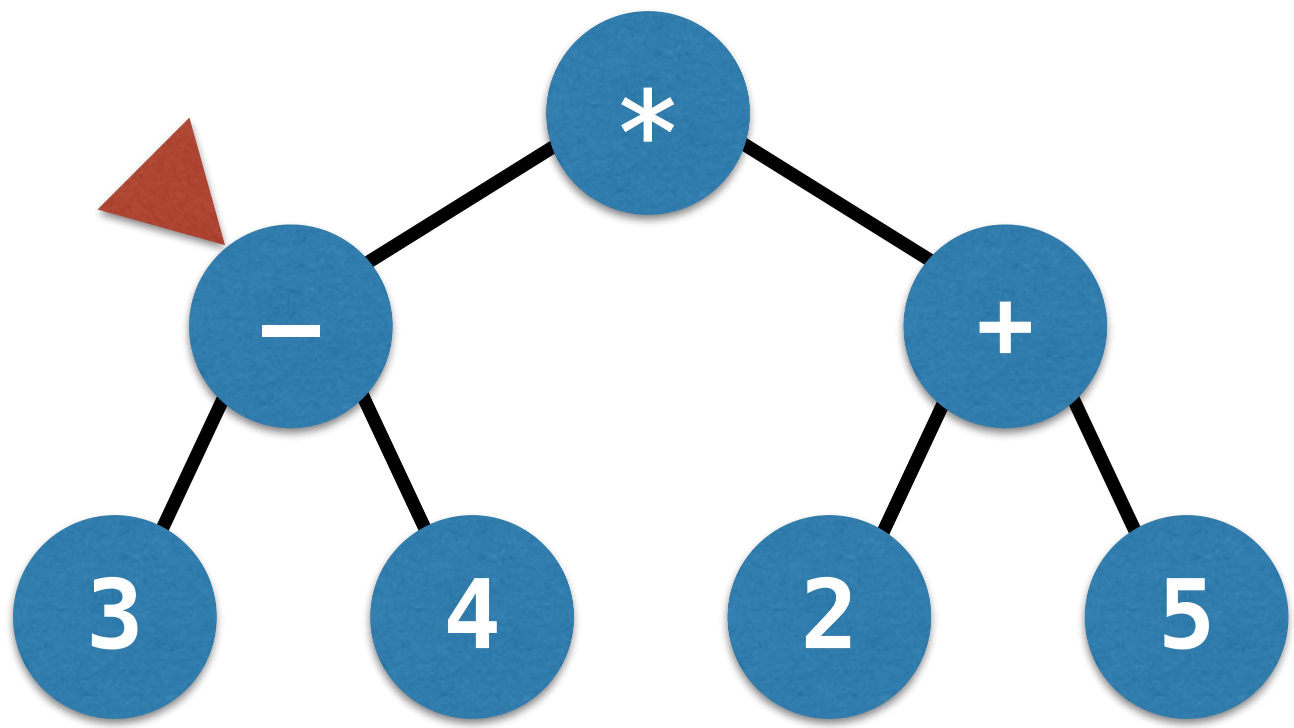


Level Order



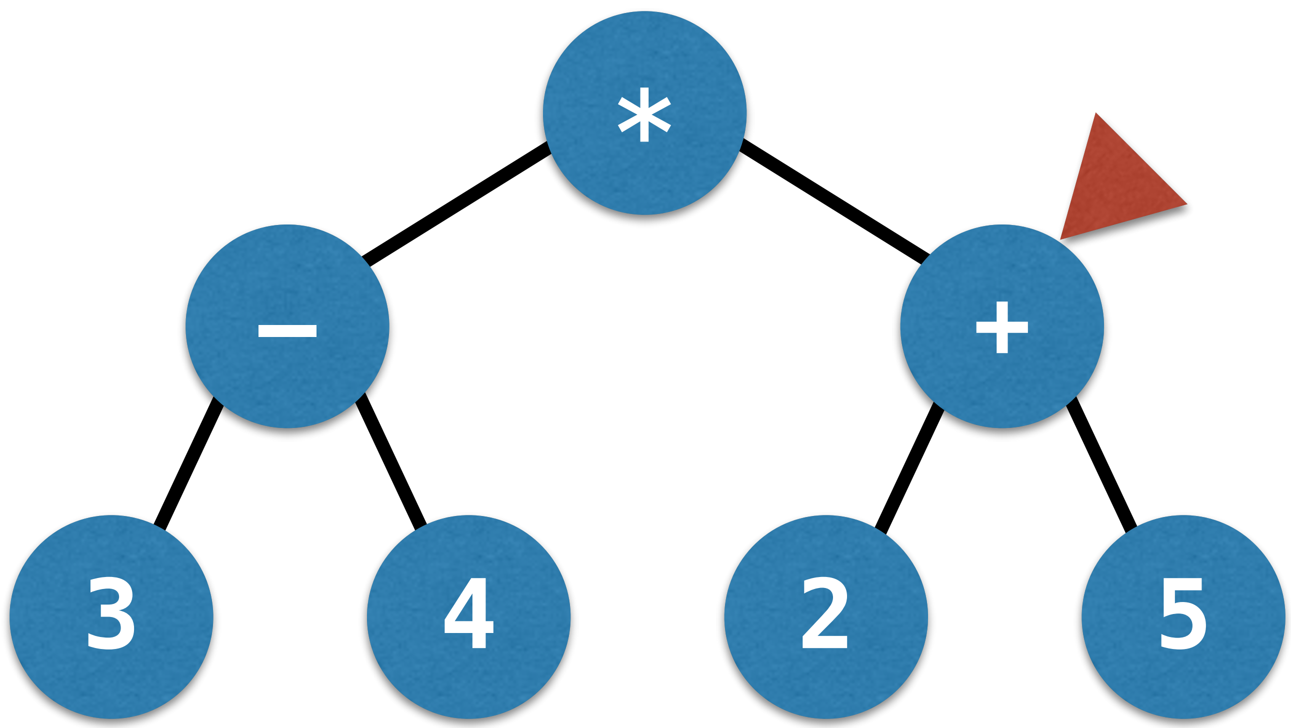
*

Level Order



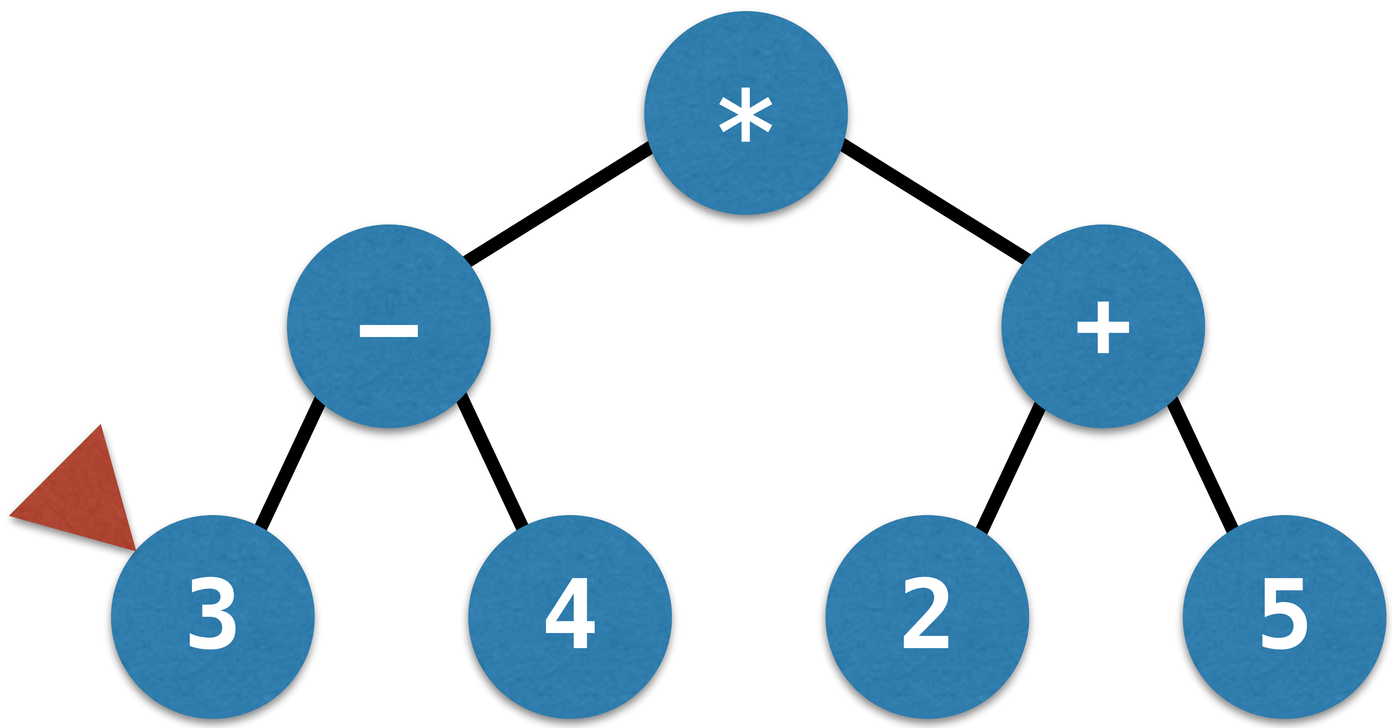
* −

Level Order



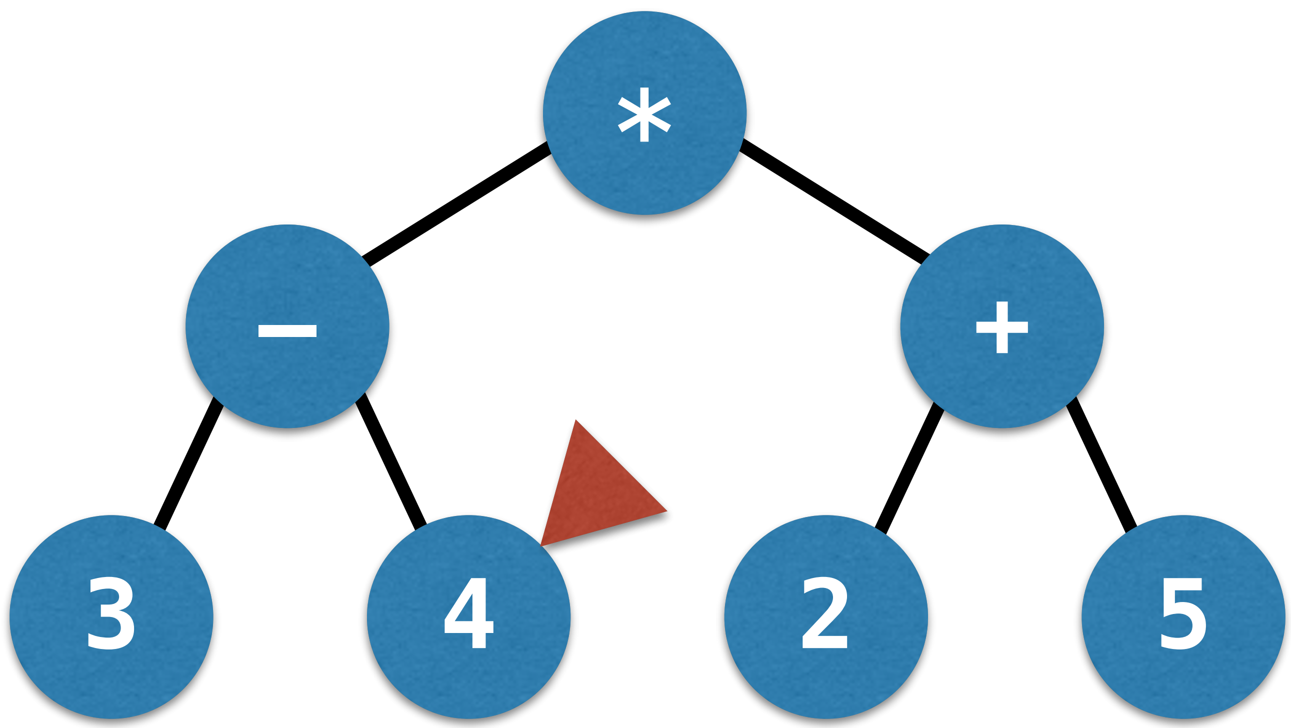
∗ − +

Level Order



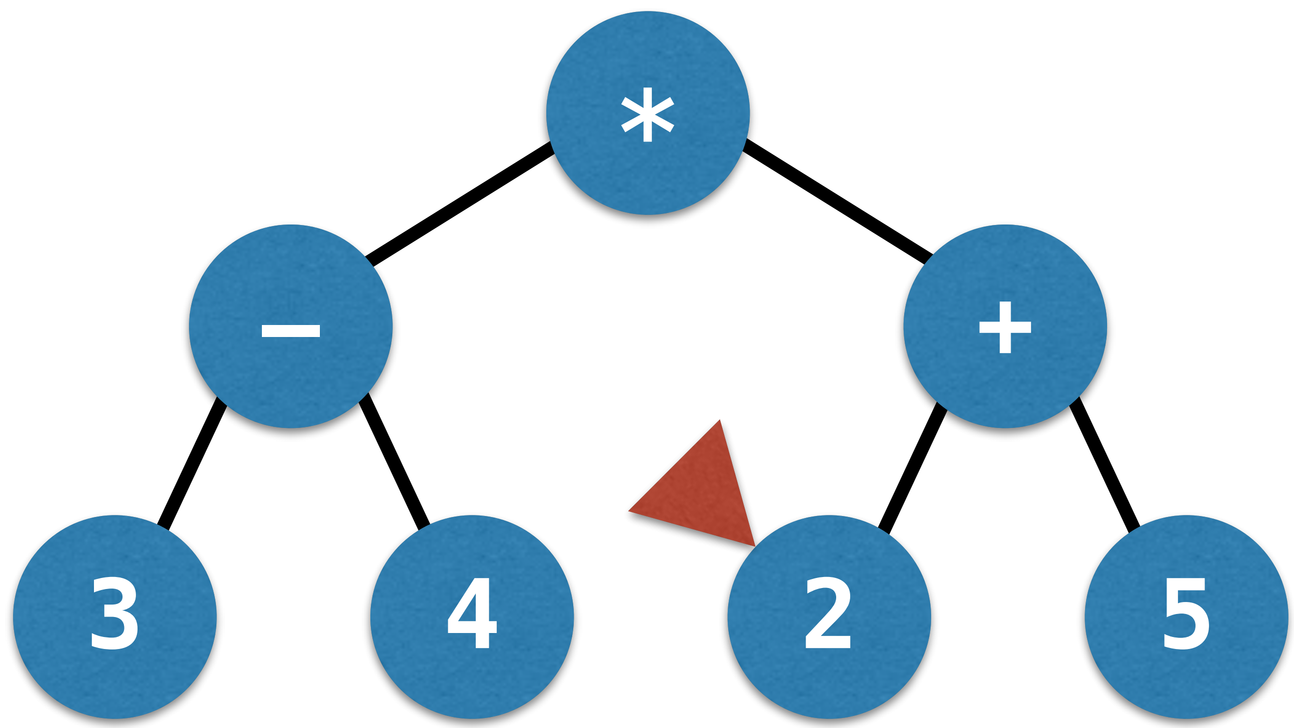
$* \quad - \quad + \quad 3$

Level Order



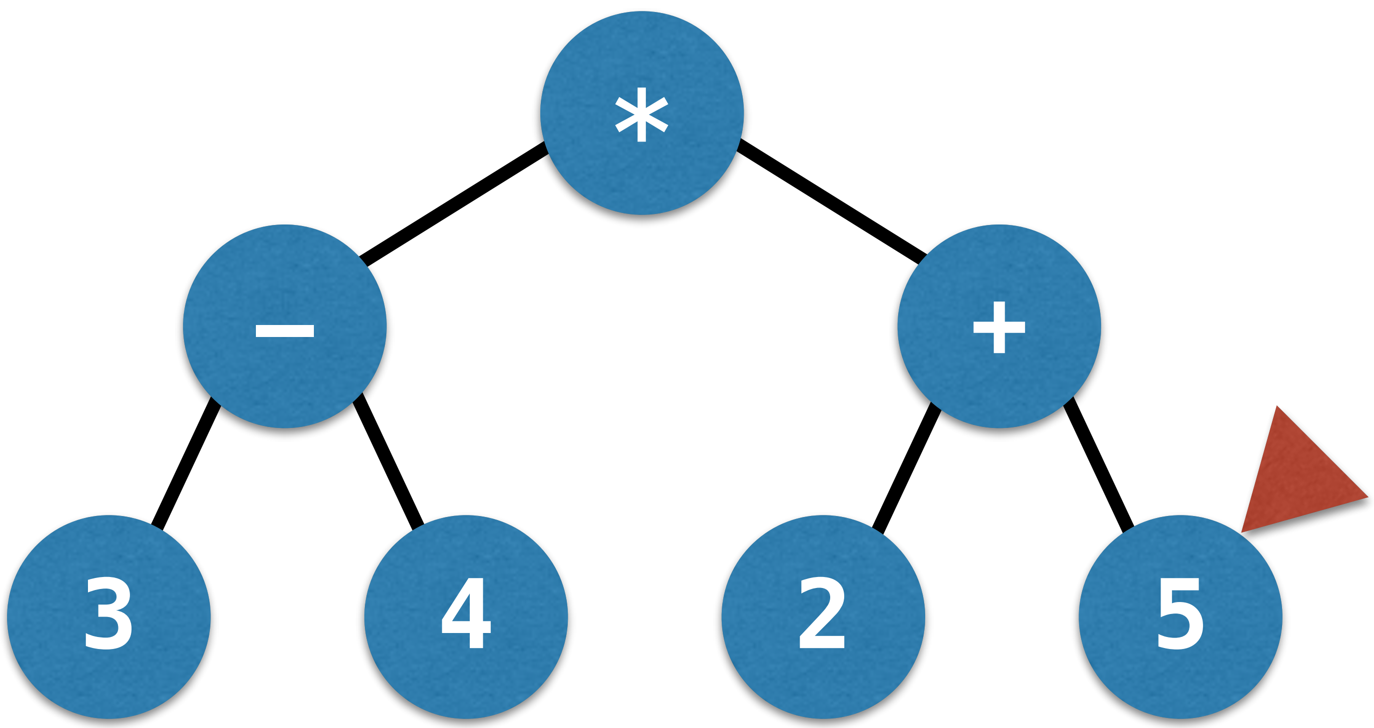
* – + 3 4

Level Order



$* \quad - \quad + \quad 3 \quad 4 \quad 2$

Level Order



$* \; - \; + \; 3 \; 4 \; 2 \; 5$

```
levelOrder(T):

  Create an empty queue q

  if T is not empty
    q.enqueue(root)

  while !q.isEmpty()
    x = q.dequeue()
    let y_left be the left child of x
    let y_right be the right child of x
    if y_left != null
      q.enqueue(y_left)
    if y_right != null
      q.enqueue(y_right)
    visit x
```