

**C I T**

---

**5 9 4 0**

# ***LINKED LISTS & TREES INTRO***





# ***Agenda***

1. ArrayList Memory Usage
2. Introducing the Node
3. Implementing the List ADT with LinkedList
4. Comparing List Implementations
5. Introducing Trees
6. Traversing Trees & Expression Tree Demo



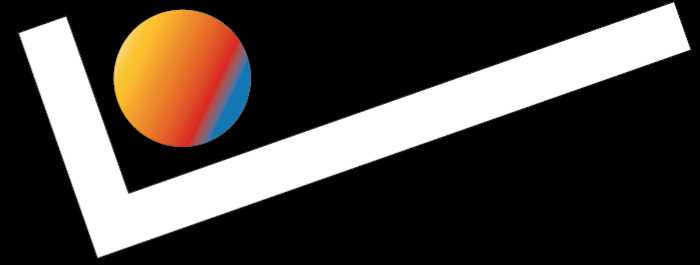
**C I T**

**5 9 4 0**

**ArrayList**

**Memory Usage**





## ***Motivation: Course Registration***

Imagine tracking student enrollment in CIT 5940:

- Initial capacity: 100 students
- After add/drop: 65 students remain

What happens to the allocated array space?



# ***ArrayList*** Internal Storage

```
@SuppressWarnings("unchecked")
private void grow() {
    E[] newElements = (E[]) new Object[elements.length * 2];
    for (int i = 0; i < size; i++) {
        newElements[i] = elements[i];
    }
    elements = newElements;
}

private void ensureCapacity() {
    if (size == elements.length) {
        grow();
    }
}
```

The array *grows*, but we haven't implemented a way of shrinking it.

# Memory Visualization

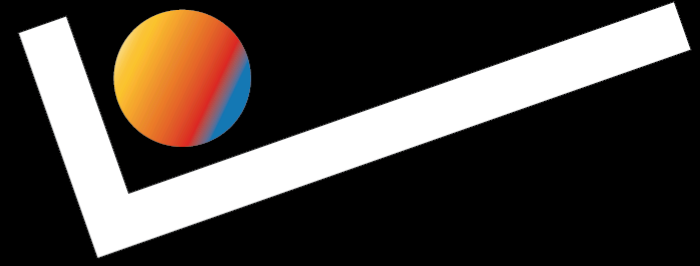
Before Drop/Add Period:

```
Capacity: 100  
Size: 100  
[S1][S2][S3][S4]...[S100]
```

After Drop/Add Period:

```
Capacity: 100  
Size: 65  
[S1][S2][S3]...[S65][x][x]...[x]  
                ^ 35 empty slots
```

Implementing `shrink()` doesn't save us—have to do so sparingly to avoid blowing up runtime cost.



# ***The Space-Time Tradeoff***

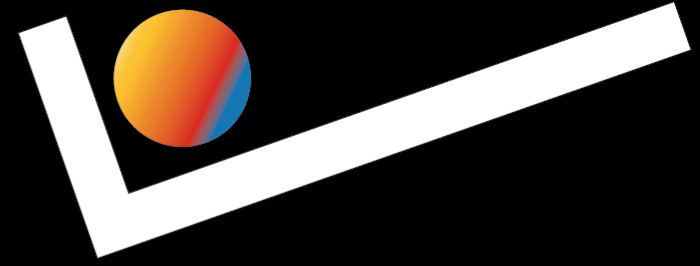
Benefits of extra capacity:

- Fast append operations (usually  $O(1)$ )
- Quick random access
- Memory locality (cache hits!!! CIT 5950!!)

Drawbacks:

- Wasted memory
- Need to periodically resize
- Cannot easily insert/remove from middle





## ***Time to Rethink...***

Key questions:

- Do we need contiguous memory?
- Can we store elements anywhere in memory?
- How would we keep track of element order?



**C I T**

**5 9 4 0**

# ***THE NODE CLASS***



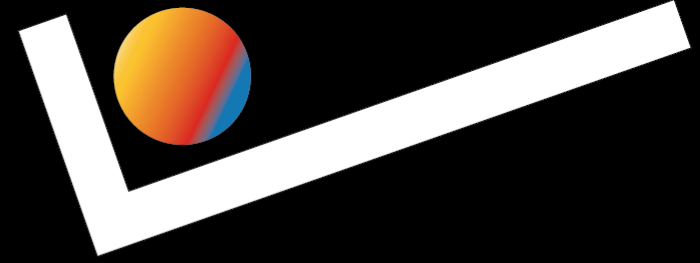


## ***Non-Sequential Storage***

Instead of storing records next to each other in memory:

- Each record is represented by its own object instance
- A record contains the data for one element
- Records are linked together through references





## ***Node Structure***

We'll use a `Node` to represent an individual record in this context. A `Node` contains:

1. The `data` element
2. A reference to the `next` `Node`

```
public class Node<E> {  
    E data;  
    Node<E> next;  
}
```



# Memory Layout

For storing values C, D, E, an ArrayList uses this organization for a List stored at address 2:

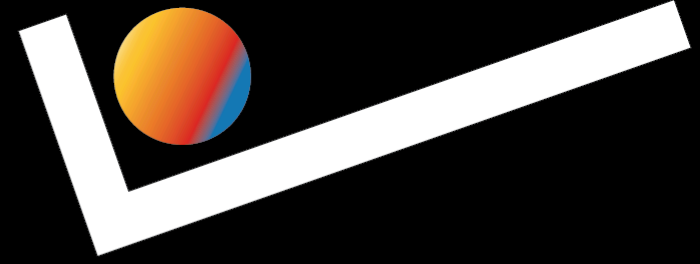
A	B	[C	D	E]	Q	Z	P	0
0	1	2	3	4	5	6	7	8

For a List of Node objects starting at address 2, we might have this shape:

A	(D6)	(C1)	D	Z	Q	(E/)	P	0
0	1	2	3	4	5	6	7	8

*Top row are values, bottom row are toy addresses.*





## ***Basic Node Implementation***

Let's implement:

- Constructor  
(...and that's pretty much it!)

A `Node` doesn't really "do" anything other than represent an individual record!



## Node Usage Example

```
Node<String> first = new Node<>("A");  
Node<String> second = new Node<>("B");  
first.setNext(second);
```

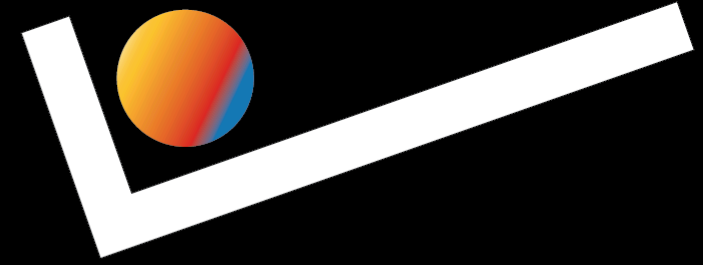
Creates an arrangement like so:

[A|→]



[B|/]

Null pointers (/ above) designate the end of a linked sequence of Nodes



## ***Node Usage Example***

Given a `Node` to start at, how would we visit reach record accessible from that start?

```
// Traverse
Node<String> current = first;
while (current != null) { // null reference indicates end of sequence
    System.out.println(current.getData());
    current = current.getNext();
}
```



## ***Think-Pair-Share***

Given a `Node` to start at, how would we visit reach record...

- in reverse order, and
- using only constant additional space

What's the runtime complexity of your solution?





```
Node<String> current = first;
int length = 0;
while (current != null) {
    length++;
    current = current.getNext();
}
for (int i = length - 1; i >= 0; i--) {
    current = first;
    for (int j = 0; j < i; j++) {
        current = current.getNext();
    }
    System.out.println(current.getData());
}
```



# ***Why Nodes?***

## Benefits:

- No wasted space
- Easy insertion/deletion
- Flexible growth

## Drawbacks:

- Extra memory per element (have to store the next pointer)
- No random access
- Non-contiguous memory (fewer cache hits)



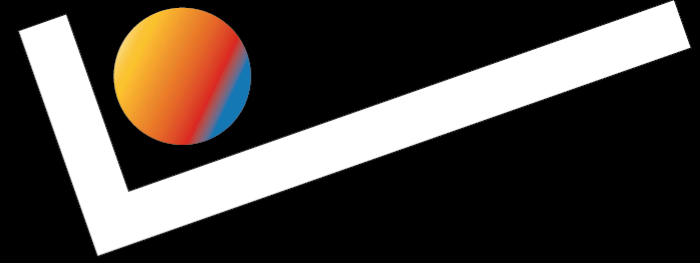
**C I T**



**5 9 4 0**

# ***LINKED LISTS***

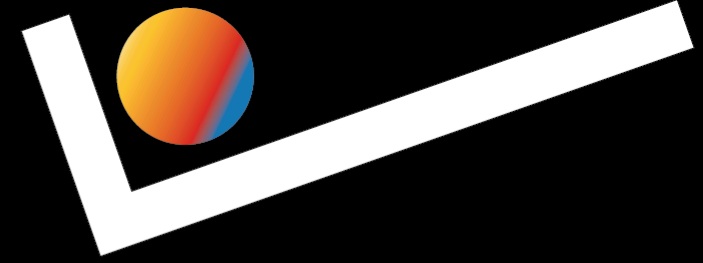




# Structure

```
class LinkedList<E> {  
    Node<E> head;  
    int size;  
}
```

Head points to first node or null if empty



# Add at Index

```
void add(int index, E element) {  
    if (index == 0) {  
        head = new Node<>(element, head);  
    } else {  
        Node<E> current = head;  
        for (int i = 0; i < index - 1; i++) {  
            current = current.next;  
        }  
        current.next = new Node<>(element, current.next);  
    }  
    size++;  
}
```

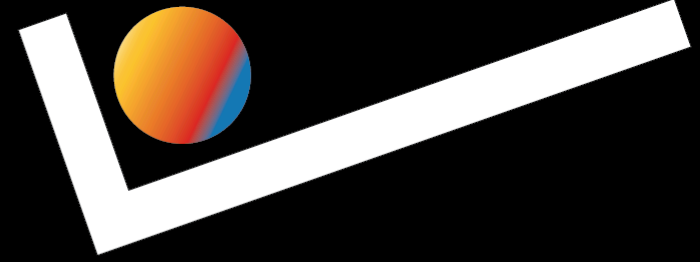
Runtime:  $O(n)$  and  $\Theta(i)$  due to navigation to addition spot.





# *Remove at Index*

```
E remove(int index) {  
    E data;  
    if (index == 0) {  
        data = head.data;  
        head = head.next;  
    } else {  
        Node<E> current = head;  
        for (int i = 0; i < index - 1; i++) {  
            current = current.next;  
        }  
        data = current.next.data;  
        current.next = current.next.next;  
    }  
    size--;  
    return data;  
}
```

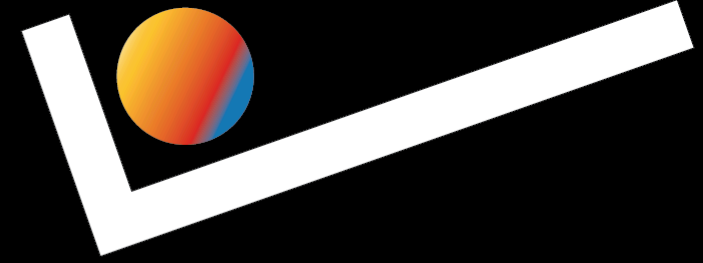


## Get Element

```
E get(int index) {  
    Node<E> current = head;  
    for (int i = 0; i < index; i++) {  
        current = current.next;  
    }  
    return current.data;  
}
```

Runtime:  $O(n)$  and  $\Theta(i)$  due to navigation to query spot.





## ***Runtime Analysis Summary***

Operation	ArrayList	LinkedList
add(i, e)	$O(n)$	$O(n)$ and $\Theta(i)$
get(i)	$O(1)$	$O(n)$ and $\Theta(i)$
remove(i)	$O(n)$	$O(n)$ and $\Theta(i)$

Special cases for  $i=0$

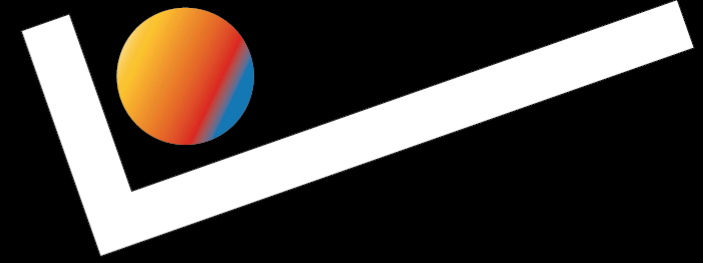




## ***Doubly Linked Lists***

1. Change `Node` to contain both a `next` and a `previous` pointer
2. Maintain a reference to the `head` AND the `tail` nodes
3. When performing an operation based on indices, start from the front or back based on whichever is closer to the target destination.





## ***Runtime Analysis Summary***

Operation	ArrayList	LinkedList	Doubly Linked List
add(i, e)	$O(n)$	$O(n)$ and $\Theta(i)$	$O(n)$ and $\Theta(\min(i, n - i))$
get(i)	$O(1)$	$O(n)$ and $\Theta(i)$	$O(n)$ and $\Theta(\min(i, n - i))$
remove(i)	$O(n)$	$O(n)$ and $\Theta(i)$	$O(n)$ and $\Theta(\min(i, n - i))$



**C I T**

**5 9 4 0**

# ***SPACE ANALYSIS***



# ***Space Complexity***

**Overhead** refers to all information stored by a data structure aside from the actual data (bad)

- Array Lists
  - Size must be predetermined before the array can be allocated
  - Unused space (overhead) if the array contains few elements
  - No overhead when array is full
- Linked Lists
  - Only need space for the elements in the list
  - Needs space for next and/or prev pointers (overhead)



## ***Which to choose?***

Given :

$n$  the number of elements in the list

$P$  the size of a pointer

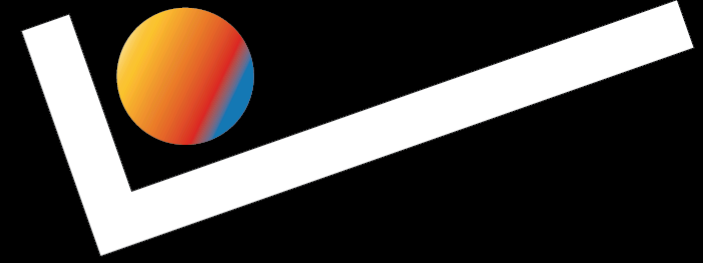
$E$  the size of a data element

$D$  the maximum number elements that can be stored in the array

Space complexity

- Array List:  $DE$
- Linked Lists:  $n(P + E)$





## ***Break-Even***

$$n(P + E) = DE$$

Solving this for  $n$  gives us the break-even point beyond which the array-based implementation is more space efficient

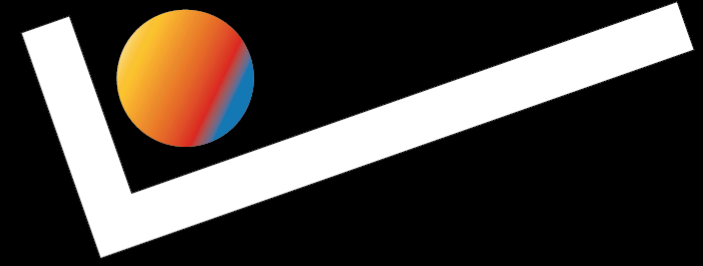
$$n = \frac{DE}{P + E}$$

If we assume  $P = E$  then break-even point is  $\frac{D}{2}$  (array half full)

$$n = \frac{DE}{2E} = \frac{D}{2}$$

Linked Lists take more space when  $n > \frac{D}{2}$  but Array Lists win out otherwise.





## ***Rule of Thumb***

- Linked Lists are more space efficient when the number of elements varies widely or is unknown
- Array Lists are more space efficient when you know the eventual size of the list in advance.

But also: you probably just want to use an Array List.



**C I T**

**5 9 4 0**

# ***FROM LISTS TO TREES***



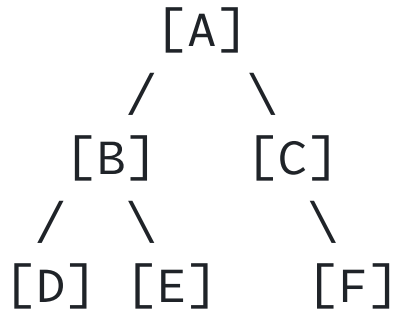


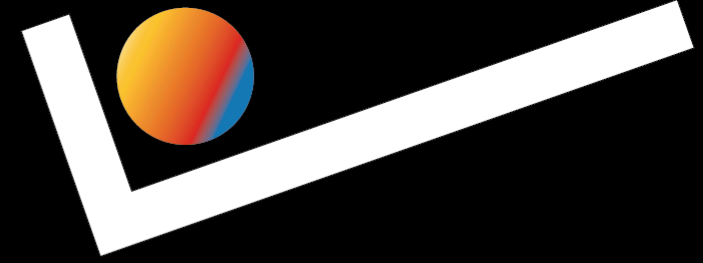
# ***Beyond Linear Structures***

Lists: One Next Node

[A] → [B] → [C] → [D]

(Binary) Trees: Multiple (up to two) Children



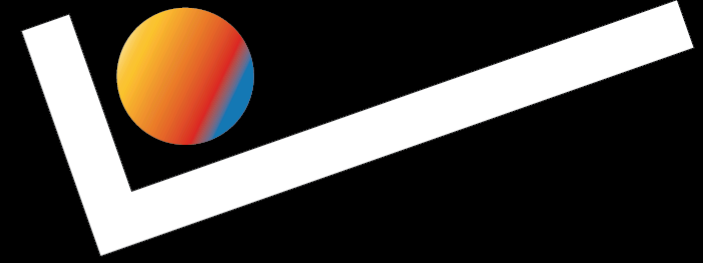


# ***BinaryTreeNode Structure***

(normally we'll just call it `Node`, but we want some contrast here...)

```
class BinaryTreeNode<E> {  
    E data;  
    BinaryTreeNode<E> left;  
    BinaryTreeNode<E> right;  
}
```

If a `BinaryTreeNode` has no children, we call it a **leaf**; otherwise it's an **internal** node.



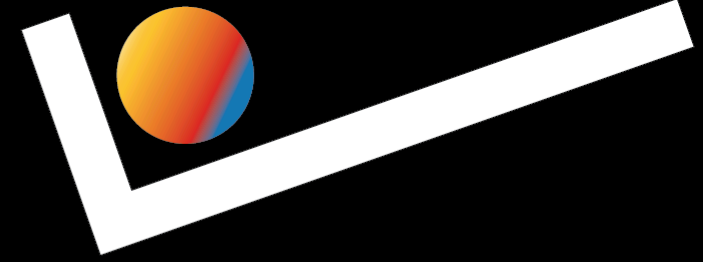
## Example: Building a Tree

```
BinaryTreeNode<String> root = new BinaryTreeNode<>("A");  
root.left = new BinaryTreeNode<>("B");  
root.right = new BinaryTreeNode<>("C");  
root.left.left = new BinaryTreeNode<>("D");
```

Creates:

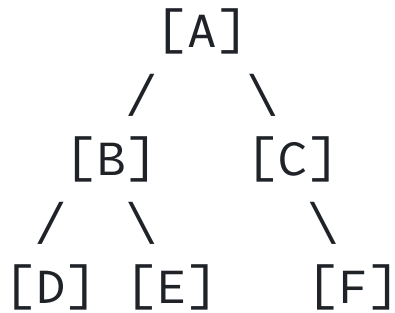
```
      root node ---> [A]  
                    /  \  
internal node -----> [B]  [C] <----- this is a leaf  
                        /  
                      [D] <----- this is a leaf
```





## ***Relationships Among Nodes***

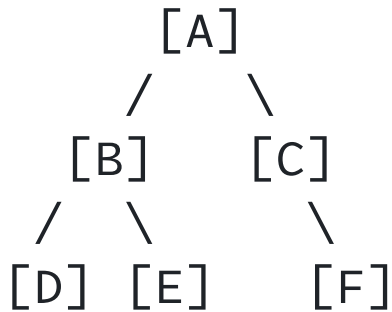
If a Node `c` is the `left` or `right` child of a Node `p`, then we say that `p` is a **parent** of `c`.



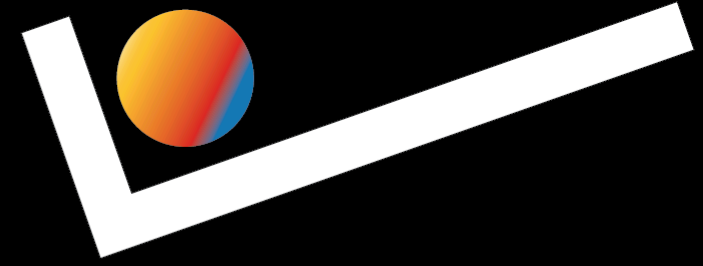
`A` is a parent of `B` and `C`.

# Paths

- A sequence of nodes  $v_1, v_2, \dots, v_n$  forms a **path** of length  $n - 1$  if there exist edges from  $v_i$  to  $v_{i+1}$  for  $1 \leq i \leq n$ .
- $v_i$  is an **ancestor** of  $v_j$  if  $i < j$  for some path
- Two nodes are **siblings** if they have the same parent & **cousins** if they share an ancestor.



$A \rightarrow B \rightarrow E$  forms a path of length 2.  $A$  and  $B$  are both ancestors of  $E$ .  $B$  and  $C$  are siblings, while  $B$  and  $F$  are cousins.



## *Probing the Depths*

- The **depth** of a Node  $m$  in a tree is the length of the path from the root of the tree to  $m$ .
- The **height** of a Tree is the depth of its deepest Node.
- All Nodes at depth  $d$  are at **level**  $d$  in the Tree. (The root is at level 0 and its children are at level 1)

# Binary Tree Rules

Mandatory:

- Each node has at most two children for a *generalized binary tree*.

Optional Variants:

- Left child < Parent < Right child for a *binary search tree* (used for `TreeSet/TreeMap`)
- All internal nodes have two children in a *full binary tree*
  - Neat property:  $\# \text{ leaves} = \# \text{ internals} + 1$
- All levels filled except last for a *complete binary tree* (used for *heaps*)
- All levels filled for a *perfect binary tree* (not that important)



# ***Applications***

- Expression Trees (today!)
- Huffman Coding (Wednesday!)
- Heaps & Priority Queues (Wednesday and beyond!)
- Binary Search Trees (in a couple weeks)



**C I T**

**5 9 4 0**

# ***TREE TRAVERSALS***





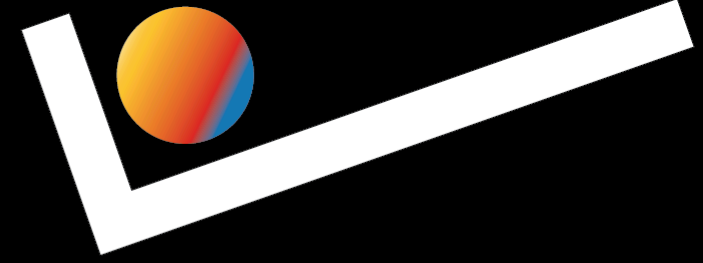
# Expression Trees

Trees that represent arithmetic expressions ordered **semantically**.

**Leaf Nodes** are always numeric values, e.g. 2, 3, 4

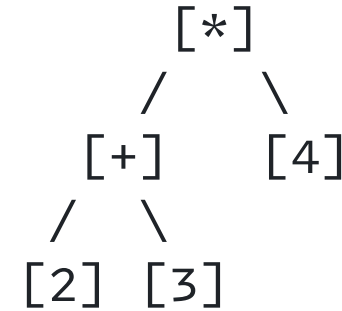
**Internal Nodes** are always operators, e.g. +, \*

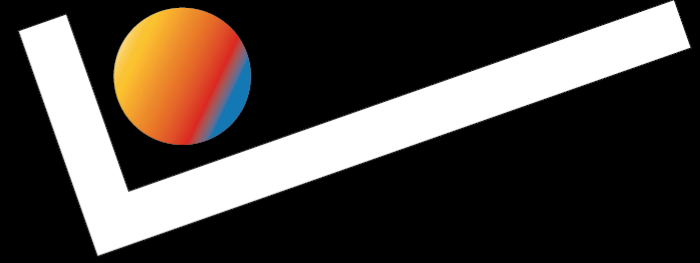
*Expression Trees are not "testable material" (won't need to remember these rules for recitation quizzes) but they are useful for thinking about traversals, which are "testable".*



## Three Ways to Visit

- **Pre-order:** Node, Left, Right
  - \*, +, 2, 3, 4
- **In-order:** Left, Node, Right
  - 2, +, 3, \*, 4
- **Post-order:** Left, Right, Node
  - 2, 3, +, 4, \*

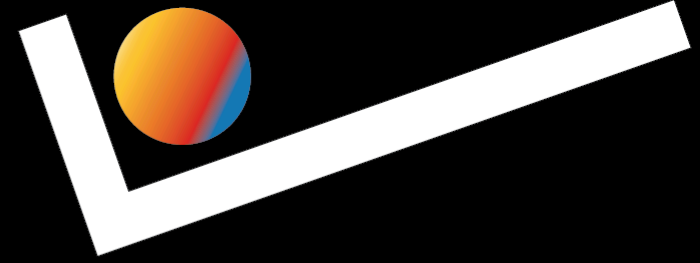




# Implementation

```
void preorder(TreeNode<E> root) {  
    if (root != null) {  
        process(root);           // Visit node  
        preorder(root.left);     // Traverse left  
        preorder(root.right);    // Traverse right  
    }  
}
```

Other orders: just rearrange the three lines!

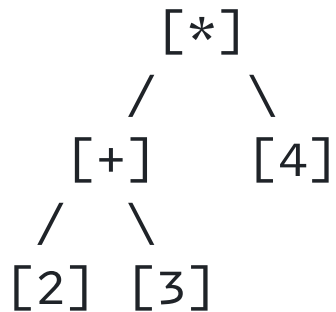


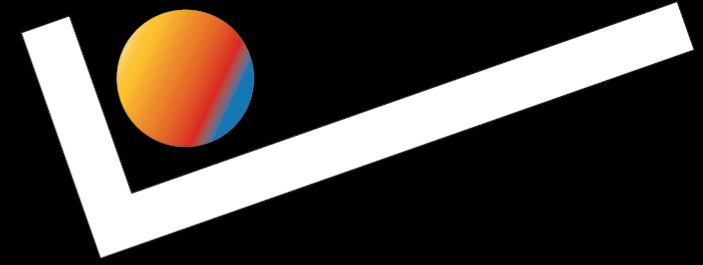
# Expression Trees

To get the human-readable expression, you'll use an **in-order** traversal.

To process the arithmetic result, you'll use **post-order**.

Example:  $2 + 3 * 4$





# ***Creating Expression Trees From Postfix Expressions***

For each token:

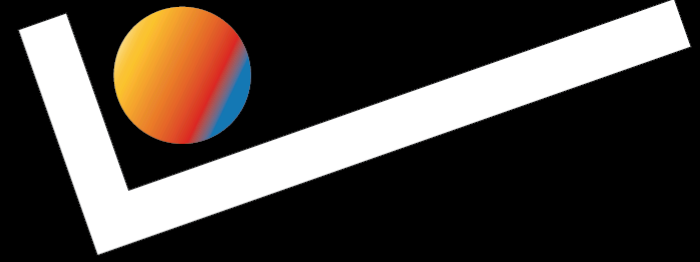
- If operand: Create leaf node, push onto a stack
- If operator:
  - i. Create operator node
  - ii. Pop two operands
  - iii. Make them children
  - iv. Push result

Demo on 3 7 + 1 8 7 + \* \*

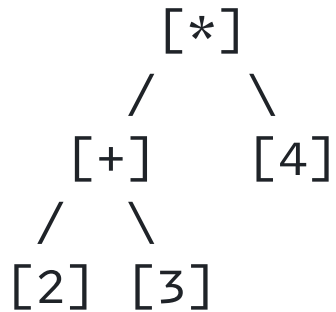


# Expression Evaluation

```
int evaluate(TreeNode<String> root) {  
    // Empty or Leaf is a base case  
    if (root == null) return 0;  
    if (root.left == null && root.right == null) {  
        return Integer.parseInt(root.data);  
    }  
    // Post-order: process children first  
    int left = evaluate(root.left);  
    int right = evaluate(root.right);  
    // Recursive case: internal nodes are operators  
    switch(root.data) {  
        case "+": return left + right;  
        case "*": return left * right;  
        default: throw new IllegalArgumentException();  
    }  
}
```



## Example Evaluation



1.  $\text{evaluate}(2) = 2$

2.  $\text{evaluate}(3) = 3$

3.  $\text{evaluate}(+) = 2 + 3 = 5$

4.  $\text{evaluate}(4) = 4$

5.  $\text{evaluate}(*) = 5 * 4 = 20$

