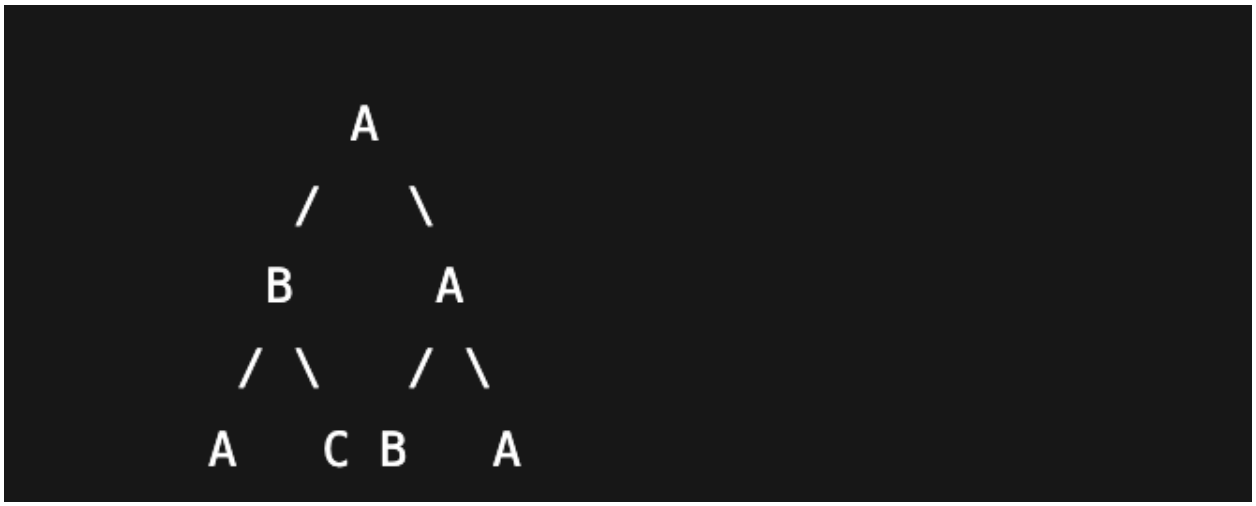
Previous Quiz 4 Answer -

```java
public static <E> int numNotElemBalanced(BinaryTree<E> tree, E elem) {
    if (tree.isEmpty()) return 0;
    return helper(tree, tree.root(), elem)[1];  // index 1 = unbalanced count
}

private static <E> int[] helper(BinaryTree<E> tree, Position<E> p, E elem) {
    // returns [countOfElem, unbalancedNodes]
    if (p == null) return new int[]{0, 0};

    int[] left = helper(tree, tree.left(p), elem);
    int[] right = helper(tree, tree.right(p), elem);

    int count = left[0] + right[0];
    if (p.getElement().equals(elem)) {
        count = count + 1;
    }

    int unbalanced = left[1] + right[1];
    if (left[0] != right[0]) {
        unbalanced = unbalanced + 1;
    }

    return new int[]{count, unbalanced};
}
```



```
numNotElemBalanced(tree, 'A')
```

Questions for Today's Class

Qs 10.1, 3, 33, 61

10.25, 26, 27, 28

R-10.1 What is the worst-case running time for inserting n key-value pairs into an initially empty map M that is implemented with the UnsortedTableMap class?

Step-by-Step Reasoning

1. How UnsortedTableMap Works:
    - Internally, it stores all key–value pairs in an unsorted list (like an ArrayList or linked list).
    - To insert a new key–value pair, it must:
        1. Search linearly through the existing entries to check if the key already exists.
        2. If found, update the value.
        3. If not found, append the new key–value pair to the list.
2. Cost per insertion:
    - Searching for an existing key takes O(m) time when there are m elements already in the map.
    - Appending the new element itself is O(1).
    - Therefore, the worst-case cost of inserting one pair (when we have m−1elements already) is:
      O(m)
3. Total cost for n insertions:
    - When inserting sequentially:
      T(n)=O(1+2+3+⋯+n)=O(n^2)

R-10.3 The use of null values in a map is problematic, as there is then no way to differentiate whether a null value returned by the call get(k) represents the legitimate value of an entry (k, null), or designates that key k was not found. The java.util.Map interface includes a boolean method, containsKey(k), that resolves any such ambiguity. Implement such a method for the UnsortedTableMap class.

In a map, we use the method get(k) to retrieve the value associated with key k.
However, if get(k) returns null, there's an ambiguity:

- It could mean that the key k is not in the map, or
- It could mean that the key k is in the map, but its value is actually null.

This is a problem for maps that allow null values, like UnsortedTableMap, because you can't distinguish between:

```
public boolean containsKey(K key) {
    // use the helper method that finds the index of a key
    return findIndex(key) != -1;
}
```

```
private int findIndex(K key) {
    for (int i = 0; i < n; i++)
        if (table[i].getKey().equals(key))
            return i;
    return -1;
}
```

Each containsKey() call does a linear scan of all existing entries.
T(n)=O(n)
since the map is unsorted and may have to check every key.

C-10.33 Consider the goal of adding entry (k, v) to a map only if there does not yet exist some other entry with key k. For a map M (without null values), this might be accomplished as follows.

```
if (M.get(k) == null)
  M.put(k, v);
```

While this accomplishes the goal, its efficiency is less than ideal, as time will be spent on the failed search during the get call, and again during the put call (which always begins by trying to locate an existing entry with the given key). To avoid this inefficiency, some map implementations support a custom method putIfAbsent(k, v) that accomplishes this goal. Given such an implementation of putIfAbsent for the UnsortedTableMap class.

In the original case

```
if (M.get(k) == null)
    M.put(k, v);
```

Let's analyze what happens internally in an UnsortedTableMap (which stores entries in a list or array):

1. M.get(k)
    - The get() method must search through all entries in the map to check whether k exists.
    - This search is linear - it may need to compare k against every key in the map.
    - Worst-case time: O(n)
2. M.put(k, v)
    - The put() method also starts by calling its own internal search (usually findIndex(k)) to check whether k already exists.
    - Again, this is a linear scan through up to n elements.
    - Worst-case time: O(n)

So, in the worst case, you do:

- one full traversal during get(k),
- another full traversal during put(k, v).

That's two separate linear scans, i.e.:
T(n)=O(n)+O(n)=O(2n)

Improvement

```
public V putIfAbsent(K key, V value) {
   int index = findIndex(key);     // helper that scans the table for key

   if (index == -1) {              // key not found
      table.add(new MapEntry<>(key, value));
      n++;
      return null;                 // nothing was replaced
   } else {
      return table.get(index).getValue();   // key already exists, return existing value
   }
}
```

C-10.61 An ***inverted file*** is a critical data structure for implementing applications such an index of a book or a search engine. Given a document D, which can be viewed as an unordered, numbered list of words, an inverted file is an ordered list of words, L, such that, for each word w in L, we store the indices of the places in D where w appears. Design an efficient algorithm for constructing L from D.

```
D = ["data", "structures", "data", "algorithms", "data"]
```

```
L = {
  "data"       → [0, 2, 4],
  "structures" → [1],
  "algorithms" → [3]
}
```

```
Map<String, List<Integer>> L = new HashMap<>();

for (int i = 0; i < D.length; i++) {
   String word = D[i];
   if (!L.containsKey(word)) {
      L.put(word, new ArrayList<>());     // initialize list
   }
   L.get(word).add(i);                    // record the index
}
```

R-10.25 Give a description, in pseudocode, for implementing the removeAll method for the set ADT, using only the other fundamental methods of the set.

```
public void removeAll(MySet<E> otherSet) {
      for (E e : otherSet.elements) {
         if (this.contains(e)) {
            this.remove(e);
         }
      }
   }
```

R-10.26 Give a description, in pseudocode, for implementing the retainAll method for the set ADT, using only the other fundamental methods of the set.

```
Algorithm retainAll(S)
Input: A set S whose elements should be retained in the current set A
Output: A modified set A containing only elements also found in S

for each element e in A do
    if NOT S.contains(e) then
        A.remove(e)
    end if
end for
```

R-10.27 If we let n denote the size of set S, and m denote the size of set T, what would be the running time of the operation S.addAll(T), as implemented on page 446, if both sets were implemented as skip lists?

We need to implement this

for each element x in T:
  S.add(x)

So the total time depends on the complexity of S.add(x) for each element in T.
In a Skip List, search and insertion both take O(log n) expected time,
because the structure maintains multiple linked “levels” for fast lookup.

Operation	Average Time	Worst Case
search(k)	O(log n)	O(n)
insert(k, v)	O(log n)	O(n)
remove(k)	O(log n)	O(n)

So, for each element $x \in T$

- We must search for x in S to check if it already exists.
- If not found, we insert x.

Both are O(logn) expected.

For all m elements of T:

T(n,m)=O(mlogn)

R-10.28 If we let n denote the size of set S, and m denote the size of set T, what would be the running time of the operation S.addAll(T), as implemented on page 446, if both sets were implemented using hashing?

```
public void addAll(Set<E> other) {
  for (E element : other)          // rely on iterator( ) method of other
    add(element);                  // duplicates will be ignored by add
}
```

Hashing -> In Avg case - O(m)
In worst case - O(M*N) If we have a bad hashing function.