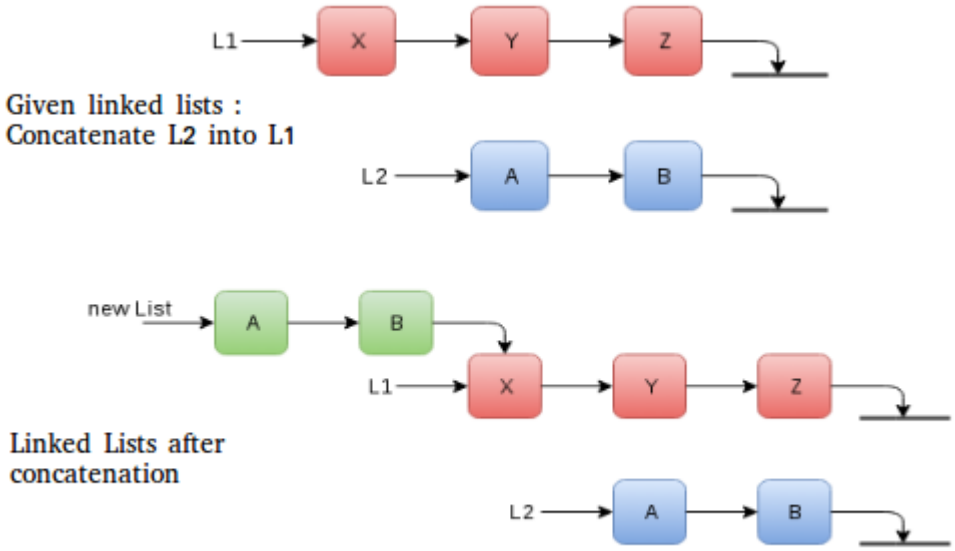
Persistent data structures

All the data structures discussed here so far are non-persistent (or ephemeral). A persistent data structure is a data structure that always preserves the previous version of itself when it is modified. They can be considered as 'immutable' as updates are not in-place. A data structure is **partially persistent** if all versions can be accessed but only the newest version can be modified. **Fully persistent** if every version can be both accessed and modified. Confluently persistent is when we merge two or more versions to get a new version. This induces a DAG on the version graph. Persistence can be achieved by simply copying, but this is inefficient in CPU and RAM usage as most operations will make only a small change in the DS. Therefore, a better method is to exploit the similarity between the new and old versions to share structure between them.

Examples:

1. **Linked List Concatenation :** Consider the problem of concatenating two singly linked lists with n and m as the number of nodes in them. Say n>m. We need to keep the versions, i.e., we should be able to original list. One way is to make a copy of every node and do the connections. O(n + m) for traversal of lists and O(1) each for adding (n + m – 1) connections. Other way, and a more efficient way in time and space, involves traversal of only one of the two lists and fewer new connections. Since we assume m<n, we can pick list with m nodes to be copied. This means O(m) for traversal and O(1) for each one of the (m) connections. We must copy it otherwise the original form of list wouldn't have persisted.



Pseudo code to implement it:

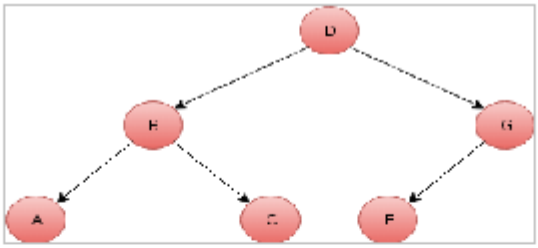
```
LinkedList concatenate(LinkedList list1, LinkedList list2) {
    Node *current = list1.head;
    while (current->next != nullptr) {
        current = current->next;
    }
    current->next = list2.head;
    list1.tail = list2.tail;
    return list1;
}
```

Explanation:

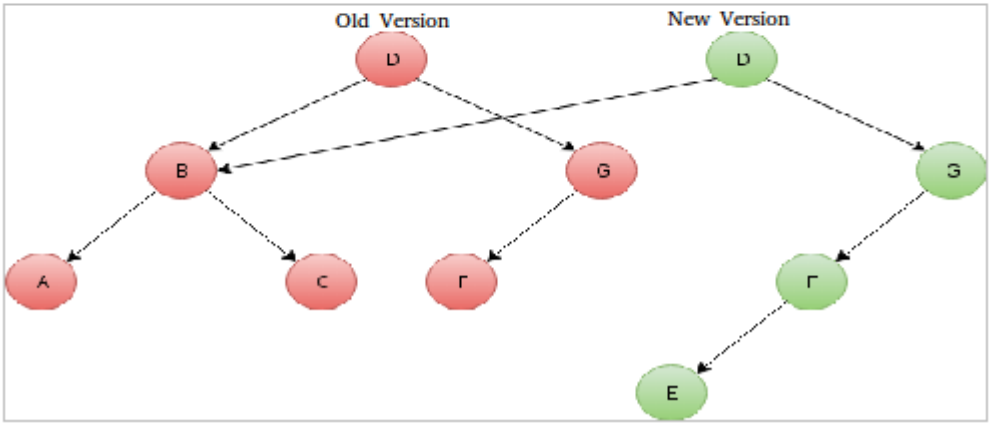
In this code, LinkedList represents a linked list, and Node represents a node in the linked list. The concatenate function takes two linked lists as input and concatenates them by modifying the next pointer of the last node of the first linked list to point to the head of the second linked list. The tail of the first linked list is then set to the tail of the second linked list. Finally, the modified first linked list is returned.

1. **Binary Search Tree Insertion :** Consider the problem of insertion of a new node in a binary search tree. Being a binary search tree, there is a specific location where the new node will be placed. All the nodes in the path from the new node to the root of the BST will observe a change in structure (cascading). For example, the node for which the new node is the child will now have a new pointer. This change in structure induces change in the

complete path up to the root. Consider tree below with value for node listed inside each one of them.



Let's add node with value 'E' to this tree.



Approaches to make data structures persistent For the methods suggested below, updates and access time and space of versions vary with whether we are implementing full or partial persistence.

1. **Path copying:** Make a copy of the node we are about to update. Then proceed for update. And finally, cascade the change back through the data structure, something very similar to what we did in example two above. This causes a chain of updates, until you reach a node no other node points to — the root. How to access the state at time t? Maintain an array of roots indexed by timestamp.
2. **Fat nodes:** As the name suggests, we make every node store its modification history, thereby making it 'fat'.

3. **Nodes with boxes:** Amortized O(1) time and space can be achieved for access and updates. This method was given by Sleator, Tarjan and their team. For a tree, it involves using a modification box that can hold: a. one modification to the node (the modification could be of one of the pointers, or to the node's key or to some other node-specific data) b. the time when the mod was applied The timestamp is crucial for reaching to the version of the node we care about. One can read more about it here. With this algorithm, given any time t, at most one modification box exists in the data structure with time t. Thus, a modification at time t splits the tree into three parts: one part contains the data from before time t, one part contains the data from after time t, and one part was unaffected by the modification (Source: MIT OCW). Non-tree data structures may require more than one modification box, but limited to in-degree of the node for amortized O(1).
4. **Copy-on-write:** This approach creates a new version of the data structure whenever a modification is made to it. The old version is still available for reading, while the new version is used for writes. This approach is memory-efficient, as most of the time, only one version of the data structure is active.
5. **Snapshotting**: This approach involves taking a snapshot of the data structure at a particular moment in time and then storing the snapshot. The snapshots can be used for reading, and modifications can be made to the latest snapshot. This approach is simple to implement but can be memory-inefficient as multiple snapshots of the data structure are kept in memory.
6. **Persistent arrays:** This approach involves creating a new version of the data structure whenever a modification is made to it, but only modifying the elements that have changed. The rest of the elements are shared between the different versions. This approach is memory-efficient but can be complex to implement.
7. **Path copying:** This approach involves creating a new version of the data structure whenever a modification is made to it, but only copying the portion of the data structure that has changed. This approach is memory-efficient and can be simple to

implement, but it may result in long chains of small updates that can slow down the system.

8. **Log-structured data structures**: This approach involves logging all modifications made to the data structure, and using the log to recreate the data structure at a later time. This approach is memory-efficient, but it can be complex to implement and may result in slow updates.