LIMITATIONS:

UNROLLED LINKED LIST:

1. **Memory Overhead**: While ULLs can reduce memory overhead compared to traditional linked lists by storing multiple elements in each node, they still incur overhead due to maintaining additional pointers and metadata (data about data) for each node.
2. **Insertion and Deletion Overhead**: Although ULLs can offer better performance for insertion and deletion operations compared to traditional linked lists, they still have a worst-case time complexity of O(n) for these operations. This is because finding the correct node for insertion or deletion may require traversing the entire list in some cases, and splitting or merging nodes may be necessary.
3. **Search Complexity**: While searching for an element in a ULL can be faster due to better cache locality, the worst-case time complexity for searching is still O(n), similar to traditional linked lists. This is because in the worst case, all nodes may need to be traversed to find the desired element.
4. **Fixed Node Size**: Many implementations of ULLs use a fixed node size to store elements as we use the array structure in each node. This fixed size may not be optimal for all scenarios and could lead to wasted memory if nodes are not fully utilized or inefficient use of space if nodes are too small for the elements being stored.
5. **Complexity of Splitting and Merging Nodes**: Splitting and merging nodes in a ULL can be complex operations, especially if the data structure needs to maintain certain invariants, such as keeping the number of elements in each node within a certain range. These operations may require additional memory allocation and copying of elements, leading to overhead.
6. **Limited Flexibility**: ULLs are not as flexible as other data structures like arrays or dynamic arrays (vectors in C++). Once the node size is chosen, it cannot be easily changed, and resizing nodes dynamically can be challenging and inefficient.
7. **Not Suitable for All Data Types**: ULLs may not be suitable for all types of data. For example, if the elements being stored are large objects or have varying sizes, the benefits of reduced memory overhead and improved cache locality may be less significant.
8. **Implementation Complexity**: Implementing and maintaining a ULL can be more complex compared to traditional linked lists or arrays.
9. **Portability**: ULLs are not as widely used or standardized as other data structures like arrays or linked lists, which may make it more challenging to find libraries or frameworks that support them.

TANGO TREE:

Tango Trees, a self-adjusting binary search tree, offer numerous benefits, but they also come with some limitations:

1. **Complexity**: Tango Trees are more complex to implement and understand compared to traditional binary search trees. Their self-adjusting nature requires complex algorithms for rotation and rebalancing, which can be challenging to implement correctly.
2. **Memory Overhead**: Tango Trees may have higher memory overhead compared to simpler binary search trees due to additional metadata and record the data required for maintaining tree structure and performing tree rotations.
3. **Performance Overhead**: While Tango Trees aim to provide better average-case performance than traditional binary search trees, they may introduce performance overhead due to the additional operations required for self-adjustment. In some cases, these overheads may outweigh the benefits of self-adjustment.
4. **Implementation Complexity**: Implementing Tango Trees correctly requires a deep understanding of binary search tree algorithms and self-adjusting data structures. Developers must carefully manage tree rotations and maintain balance while ensuring correctness and efficiency.
5. **Limited Practical Use Cases**: Tango Trees may not be suitable for all scenarios. In many cases, simpler binary search trees like AVL trees may offer comparable or better performance with less implementation complexity.
6. **Unproven in Practice**: While Tango Trees have been proposed as a theoretical improvement over traditional binary search trees, their practical performance benefits compared to other self-adjusting trees like AVL trees have not been extensively studied or proven in real-world scenarios.
7. **Algorithmic Complexity**: The time complexity of operations in Tango Trees, such as insertion, deletion, and search, may be more difficult to analyse compared to traditional binary search trees. While Tango Trees aim to provide better amortized performance, analysing worst-case time complexity can be challenging due to their self-adjusting nature.
8. **Maintenance Overhead**: Tango Trees may require more frequent rebalancing and adjustment operations compared to traditional binary search trees, leading to higher maintenance overhead in terms of CPU time and memory usage.
9. **Lack of Standardization**: Tango Trees are not as widely adopted or standardized as other binary search tree variants like AVL trees . This lack of standardization may make it harder to find libraries, resources, or community support for implementing and using Tango Trees.