# List

1. Introduction to "list" Container:
    - The "list" container in STL is an implementation of a doubly-linked list data structure.
    - It allows efficient insertion and deletion of elements at both ends and at any position within the list.
    - Unlike vectors or arrays, lists do not provide random access to elements.
2. Features and Characteristics:
    - Doubly-Linked List: Each element in a list contains pointers to both the previous and the next elements, allowing efficient insertion and deletion operations.
    - Dynamic Size: The size of a list can grow or shrink dynamically as elements are added or removed.
    - No Random Access: Unlike vectors or arrays, lists do not provide direct access to elements using indices. To access an element, we need to traverse the list from the beginning or end.
    - Iterators: Lists support bidirectional iterators, which allow iteration over elements in both forward and reverse directions.
3. Common Operations and Complexity:
    - Insertion and Deletion:
        - Insertion at the beginning or end: O(1) constant time complexity.
        - Insertion at a specific position: O(1) constant time complexity for inserting or erasing an element at any position.
    - Accessing Elements:
        - Traversing the list: O(n) linear time complexity, as each element needs to be visited.
    - Searching:
        - Linear search: O(n) linear time complexity, as all elements need to be checked until the desired element is found.
4. Use Cases:
    - When frequent insertion or deletion of elements is required, especially at the beginning or end of the list.

- When the order of elements is important, and random access is not necessary.
- When memory allocation and reallocation are a concern, as lists dynamically manage memory without requiring large contiguous blocks.

The Time and Space Complexity of the functions used in the code.

1. **myList.empty()**
    - Time Complexity: O(1)
    - Space Complexity: O(1)
2. **myList.push_back(10)**
    - Time Complexity: O(1)
    - Space Complexity: O(1)
3. **printList(myList)**
    - Time Complexity: O(n), where n is the number of elements in the list
    - Space Complexity: O(1)
4. **myList.insert(it, 15)**
    - Time Complexity: O(1)
    - Space Complexity: O(1)
5. **myList.remove(10)**
    - Time Complexity: O(n), where n is the number of elements in the list
    - Space Complexity: O(1)
6. **printListReverse(myList)**
    - Time Complexity: O(n), where n is the number of elements in the list
    - Space Complexity: O(1)
7. **myList.size()**
    - Time Complexity: O(1)
    - Space Complexity: O(1)
8. **myList.front()**
    - Time Complexity: O(1)
    - Space Complexity: O(1)

9. **myList.back()**
    - Time Complexity: O(1)
    - Space Complexity: O(1)
10. **myList.assign(otherList.begin(), otherList.end())**
    - Time Complexity: O(n), where n is the number of elements in the other list
    - Space Complexity: O(n)
11. **myList.clear()**
    - Time Complexity: O(n), where n is the number of elements in the list
    - Space Complexity: O(1)
12. **myList.splice(myList.begin(), otherList)**
    - Time Complexity: O(1)
    - Space Complexity: O(1)
13. **sortedList.merge(myList)**
    - Time Complexity: O(n+m), where n is the number of elements in sortedList and m is the number of elements in myList
    - Space Complexity: O(1)
14. **sortedList.unique()**
    - Time Complexity: O(n), where n is the number of elements in the list
    - Space Complexity: O(1)
15. **sortedList.resize(3)**
    - Time Complexity: O(n), where n is the new size of the list
    - Space Complexity: O(1)
16. **sortedList.erase(++it1)**
    - Time Complexity: O(1)
    - Space Complexity: O(1)