Scott Elliott
Design

Sorted List Class with Template

1. Specification

1.1 Problem to be Solved

This program aims to implement a **template-based sorted list container** in C++ that automatically maintains its elements in ascending order. The SortedList<Object> class will utilize a **doubly-linked list** with a **dummy header node** to ensure efficient insertion, removal, and access of elements while preserving the sorted sequence. The design prioritizes STL-like functionality, offering a simpler interface without iterators for scenarios requiring automatic ordering.

1.2 Assumptions

To facilitate the implementation, the following assumptions are made regarding the Object type and client behavior:

- The Object type must possess a **default constructor**.
- The Object type must support **copy/move operations**.
- Comparison operators (<, ==, !=) for the Object type must be **consistent** and define a **total ordering**.
- Client code is responsible for **checking return values** from insert() and remove() operations to handle success or failure.

1.3 Special Cases

The implementation must robustly handle various edge cases to ensure stability and correctness:

- **Empty List Operations**: Operations such as remove(), operator[], size(), empty(), and clear() must behave correctly when performed on an empty list. remove() should return false, operator[] will throw std::out_of_range, size() should return 0, empty() should return true, and clear() should act as a no-op.
- **Duplicate Elements**: Duplicates are permitted and maintained in insertion order, meaning a new duplicate will be placed *after* existing equal elements. remove() will only eliminate the first occurrence of a duplicate.
- **Out-of-Bounds Access**: Accessing operator[] with a negative or out-of-bounds index (>= size()) will throw std:out_of_range exception.
- **Single Element List**: Operations on a list containing a single element, such as removal or access, must function correctly.
- **Self-Assignment**: Copy and move assignment operators (operator=) must include self-assignment checks (if (this == &rhs)) to prevent resource deallocation issues and ensure the list remains unchanged.

- **Memory Allocation Failure**: The insert() method should return false if new fails to allocate memory (though std::bad_alloc is typically thrown), leaving the list in a valid state.
- **Move from Moved-From Object**: After a move operation, the source object (other) must be left in a valid but empty state (e.g., header = nullptr, listSize = 0) to prevent issues upon its destruction.
- **Comparison of Empty Lists**: The operator== must correctly compare empty lists (returning true for two empty lists).
- **Merge with Empty Lists**: The operator+ must handle cases where one or both operand lists are empty.

1.4 Input Data

Input to the SortedList class is accepted through several mechanisms:

- **Constructors**: Input can be an empty call (default), a const SortedList& other (copy), or a SortedList&& other (move). Default, copy, and move constructors allow for initialization of empty lists, deep copies from existing lists, or efficient transfer of ownership.
- **insert() Method**: The input is a const Object& item. Individual elements of any comparable type Object (supporting <, ==, !=) can be inserted. The list automatically determines the correct sorted position.
- **operator+ (Merge)**: The input is a const SortedList& rhs. Two SortedList instances can be merged to create a new sorted list.

1.5 Output Data

The SortedList provides output in various forms:

- **Stream Output (operator<<)**: Overloaded operator<< displays elements in sorted, comma-separated format (e.g., "element1, element2, element3") without a trailing comma.
- **Index Access (operator[])**: Returns the element at a zero-based index. An out-of-bounds index returns a default-constructed Object.
- **Boolean Operation Results**: Methods like insert() and remove() return bool to indicate success or failure. Comparison operators (==, !=) also return bool.
- **Size/Empty Status**: size() returns the current element count, and empty() returns true if the list contains no elements.

1.6 Error Detection and Messages

Error handling is primarily managed through return values and the design of operator[]:

- **remove()**: Returns false if the item to be removed is not found.
- **insert()**: Returns false if dynamic memory allocation fails.
- **operator[]**: Throws std::out_of_range for out of bounds access. Client code is expected to handle this exception using try-catch blocks.
- **Self-Assignment**: Handled internally by assignment operators to prevent logical errors.

2. Program Design

2.1 Overall Structure

The SortedList class is implemented using a **doubly-linked list** with a **dummy header node**. This design simplifies operations by removing special cases for manipulations at the list's beginning. The use of a template parameter Object allows the class to be generic across comparable types. All node management is encapsulated within private members, exposing only a high-level public interface that adheres to STL conventions for naming and const correctness, while intentionally avoiding the complexity of iterators.

2.2 Class Relationships: Public (+), Private (-), and Protected (#)

```
classDiagram
```

```
    class SortedList<Object> {
        - Node* header
        - int listSize
        + SortedList()
        + SortedList(const SortedList& other)
        + SortedList(SortedList&& other)
        + ~SortedList()
        + operator=(const SortedList& rhs) SortedList&
        + operator=(SortedList&& rhs) SortedList&
        + insert(const Object& item) bool
        + remove(const Object& item) bool
        + operator[](int index) Object
        + operator+(const SortedList& rhs) SortedList
        + size() int const
        + empty() bool const
        + clear() void
        + operator==(const SortedList& sl) bool const
        + operator!=(const SortedList& sl) bool const
        - findInsertPosition(const Object& item) Node*
        - copyFrom(const SortedList& other) void
    }

    class Node {
        - Object data
        - Node* next
        - Node* prev
        - Node(const Object& d, Node* p, Node* n)
    }

    SortedList o-- Node : contains (private) header
    Node <--> Node : next/prev
```

```
    SortedList ..> ostream : friend operator<<
```

2.3 Class Descriptions

Node (Private Nested Structure): The Node class serves as the fundamental building block of the doubly-linked list. Each instance encapsulates a data element of type Object and maintains pointers to its preceding and succeeding nodes. This structure enables bidirectional traversal and efficient linking/unlinking during list modifications.

Attributes (Member Variables):

- - Object data: Stores the actual value of the element.
- - Node* next: Pointer to the next node in the sequence. nullptr if it's the last node.
- - Node* prev: Pointer to the previous node in the sequence. Never nullptr due to the dummy header.

Operations (Methods):

- - Node(const Object& d = Object{}, Node* p = nullptr, Node* n = nullptr):
  - **Description**: Constructor that initializes a Node with provided data and optional prev/next pointers. Default parameters allow flexible node creation.
  - **Parameters**: d (data), p (previous node pointer), n (next node pointer).
  - **Return Value**: None (constructor).

SortedList<Object> (Main Template Class): The SortedList class is the primary container, providing a robust and automatically sorted collection of Object instances. It manages the underlying Node objects to ensure that elements are always kept in ascending order, offering a comprehensive set of operations for manipulation, access, and comparison, all while adhering to a simplified, iterator-free interface.

Attributes (Member Variables):

- - Node* header: A pointer to the dummy header node. This node always exists, simplifying list operations by acting as a sentinel.
- - int listSize: Stores the current number of actual elements in the list, excluding the dummy header.

Operations (Methods):

- + SortedList():
  - **Description**: Default constructor. Initializes an empty list by creating a dummy header node and setting listSize to 0.
  - **Parameters**: None.
  - **Return Value**: None.
- + SortedList(const SortedList& other):

    - **Description**: Copy constructor. Performs a deep copy of all elements from other into a new SortedList instance, maintaining sorted order.
    - **Parameters**: other (a constant reference to another SortedList to be copied).
    - **Return Value**: None.
- + SortedList(SortedList&& other):
    - **Description**: Move constructor. Transfers ownership of other's resources (nodes) to a new SortedList instance, leaving other in a valid, empty state.
    - **Parameters**: other (an rvalue reference to another SortedList to be moved).
    - **Return Value**: None.
- + ~SortedList():
    - **Description**: Destructor. Deallocates all Node objects, including the dummy header, ensuring no memory leaks.
    - **Parameters**: None.
    - **Return Value**: None.
- + operator=(const SortedList& rhs): SortedList&:
    - **Description**: Copy assignment operator. Clears the current list, then performs a deep copy of rhs. Includes self-assignment protection.
    - **Parameters**: rhs (a constant reference to the SortedList to be assigned).
    - **Return Value**: A reference to the current SortedList (*this) for chaining.
- + operator=(SortedList&& rhs): SortedList&:
    - **Description**: Move assignment operator. Clears the current list, then transfers ownership of rhs's resources. Includes self-assignment protection and leaves rhs empty.
    - **Parameters**: rhs (an rvalue reference to the SortedList to be moved).
    - **Return Value**: A reference to the current SortedList (*this) for chaining.
- + insert(const Object& item): bool:
    - **Description**: Inserts item into its correct sorted position within the list. Duplicates are allowed and placed after existing equal elements. Increments listSize on success.
    - **Parameters**: item (a constant reference to the Object to be inserted).
    - **Return Value**: true on successful insertion, false if memory allocation fails.
- + remove(const Object& item): bool:
    - **Description**: Removes the first occurrence of item from the list. Decrements listSize on success and adjusts pointers.
    - **Parameters**: item (a constant reference to the Object to be removed).
    - **Return Value**: true if item was found and removed, false otherwise.
- + operator[](int index): Object:
    - **Description**: Provides access to the element at the specified zero-based index. Traverses the list (O(n) time complexity).
    - **Parameters**: index (the zero-based position of the element).
    - **Return Value**: The Object at index, Throws std::out_of_range exception if index is out of bounds.
- + operator+(const SortedList& rhs): SortedList:

    - **Description**: Merges the current list with rhs into a new SortedList, maintaining sorted order. Does not modify the original lists.
    - **Parameters**: rhs (a constant reference to the SortedList to be merged).
    - **Return Value**: A newly constructed SortedList containing elements from both operands.
- + size(): int const:
    - **Description**: Returns the number of elements currently in the list.
    - **Parameters**: None.
    - **Return Value**: An int representing the listSize.
- + empty(): bool const:
    - **Description**: Checks if the list is empty.
    - **Parameters**: None.
    - **Return Value**: true if listSize is 0, false otherwise.
- + clear(): void:
    - **Description**: Removes all elements from the list, leaving only the dummy header. Resets listSize to 0.
    - **Parameters**: None.
    - **Return Value**: None.
- + operator==(const SortedList& sl): bool const:
    - **Description**: Compares the current list with sl for equality. Returns true if both lists have the same size and all elements are identical in order.
    - **Parameters**: sl (a constant reference to another SortedList for comparison).
    - **Return Value**: true if lists are equal, false otherwise.
- + operator!=(const SortedList& sl): bool const:
    - **Description**: Returns the logical opposite of operator==.
    - **Parameters**: sl (a constant reference to another SortedList for comparison).
    - **Return Value**: true if lists are not equal, false otherwise.

Friend Functions:

- template<typename T> friend ostream& operator<<(ostream& os, const SortedList<T>& sl):
    - **Description**: Overloaded stream insertion operator. Allows SortedList objects to be printed to an ostream (e.g., cout) in a comma-separated format.
    - **Parameters**: os (an ostream reference), sl (a constant reference to the SortedList to be printed).
    - **Return Value**: A reference to the ostream (os) for chaining.

Private/Helper Methods:

- - findInsertPosition(const Object& item): Node*:
    - **Description**: A helper method used by insert() to locate the Node* before which a new item should be inserted to maintain sorted order.

    - **Parameters**: item (a constant reference to the Object to find the position for).
    - **Return Value**: A pointer to the node that should follow the new node, or nullptr if the item should be inserted at the end.
- - copyFrom(const SortedList& other): void:
    - **Description**: A helper method used by the copy constructor and copy assignment operator to perform a deep copy of elements from other.
    - **Parameters**: other (a constant reference to the SortedList to copy from).
    - **Return Value**: None.

3. Implementation and Test Plan

Phase 1: Core Node and SortedList Structure

1. **Implement Node**: Create the private nested Node struct with its constructor.
2. **Implement SortedList Basics**: Implement the default constructor (creating the dummy header), destructor (clear()), size(), empty(), and clear() methods.

How to Test (Unit Tests):

- **Constructor**: Create a SortedList and verify size() is 0 and empty() is true.
- **clear()**: Test on an empty list and a populated list (after Phase 2) to ensure size() becomes 0.

Phase 2: insert() and operator<<

3. **Implement insert()**: Implement the logic to find the correct insertion point and link the new node.
4. **Implement operator<<**: Implement the stream insertion operator to print the list's contents for easy debugging and verification.

How to Test (Unit Tests):

- **insert()**: Insert into an empty list. Insert at the beginning, middle, and end of an existing list. Insert duplicates. Verify size() and list content using operator<<.

Phase 3: Remaining SortedList Methods

5. **Implement remove()**: Find and unlink a node.
6. **Implement operator[]**: Traverse the list to the specified index.
7. **Implement operator+**: Merge two lists into a new one.
8. **Implement operator== and operator!=**: Compare two lists for equality.

How to Test (Unit Tests):

- **remove()**: Remove existing items (first, middle, last). Remove non-existent items. Remove duplicates (only first should be removed). Verify size() and list content after each removal.
- **operator[]**: Access elements at valid indices. Access out-of-bounds indices (negative, too large) within a try-catch block and verify that std::out_of_range is thrown. Verify const correctness.
- **operator+**: Merge two non-empty lists, an empty list with a non-empty list, and two empty lists. Verify the resulting list is correctly sorted and contains all elements, and original lists are unchanged.
- **operator== and operator!=**: Compare identical lists, different lists, lists of different sizes, and empty lists. Verify boolean results are correct.

Phase 4: Integration Testing

9 Implement the copy constructor, copy assignment operator, move constructor, and move assignment operator.

How to Test (Integration Tests):

- **Copy Constructor**: Create a list, then initialize a new list from it. Verify the new list is an identical but separate copy.
- **Copy Assignment**: Create two lists, then assign one to the other. Verify the assignment target is a deep copy. Test self-assignment.
- **Move Constructor/Assignment**: Test move semantics to ensure resource transfer and that the source object is left empty.

Sample Driver code: To illustrate basic usage and serve as an integration test, the following driver.cpp can be used:

```
#include "SortedList.h"
#include <iostream>
#include <string>
#include <stdexcept> // Required for std::out_of_range

using namespace std;

int main(){

SortedList<int> alist;

SortedList<string> strlist;

alist.insert(3);
alist.insert(1);
try {
    cout << "Accessing valid index [0]: " << alist[0] << endl;
```

```
    cout << "Accessing invalid index [10]: ";
    cout << alist[10] << endl; // This should throw an exception
} catch (const std::out_of_range& oor) {
     cerr << "Caught an out_of_range exception: " << oor.what()
          << endl; }

cout << alist[0] << endl;

SortedList<int> blist(alist);
cout << blist[1] << endl;

strlist.insert("k");
strlist.insert("a");

return 0;
}
```

References

AI assistance was provided by Claude (Anthropic's Claude Sonnet 4.5) for grammatical review, structural organization, and documentation formatting. All technical content, design decisions, algorithm specifications, and implementation strategies are the original work of Scott Elliott.

Original prompt to Claude: "Review this SortedList class design for grammatical errors and suggest improvements to document structure.