/\*

\* List Abstract Data Type (ADT)

\*

\* This interface defines a List ADT that supports a sequence of elements with the ability to perform

\* insertion, deletion, access, and traversal operations. It does not specify the internal implementation,

\* allowing different data structures (such as arrays or linked lists) to implement it according to their

\* own space/time trade-offs.

\*

\* The key design feature of this ADT is the concept of a "current position," which enables element access

\* and movement through the list. The operations defined below provide a foundation for list management

\* with flexibility for different use cases.

\*

\* Variants of List Implementations:

\* 1. \*\*Array-based List\*\*: Uses a contiguous block of memory to store elements.

\* - Advantages: Fast random access, low overhead for accessing elements.

\* - Disadvantages: Fixed capacity unless dynamically resized, costly insertions/removals in the middle.

\*

\* 2. \*\*Singly Linked List\*\*: Uses nodes connected by pointers, each containing a value and a reference to the next node.

\* - Advantages: Efficient insertions and deletions at arbitrary positions.

\* - Disadvantages: No direct access to elements (requires traversal), additional memory overhead for pointers.

\*

\* 3. \*\*Doubly Linked List\*\*: Each node contains pointers to both the next and previous nodes.

\* - Advantages: Efficient traversal in both directions, improved deletion performance.

\* - Disadvantages: Increased memory overhead compared to singly linked lists.

\*

\* 4. \*\*Circular Linked List\*\*: Similar to a singly or doubly linked list, but the last node points back to the first node.

\* - Advantages: Useful for buffering applications, round-robin scheduling.

\* - Disadvantages: Requires careful handling of pointers to avoid infinite loops.

\*

\* 5. \*\*Skip List\*\*: A probabilistic data structure that maintains multiple layers of linked lists for fast search operations.

\* - Advantages: Faster search times than normal linked lists, with performance comparable to balanced trees.

\* - Disadvantages: Increased complexity and memory usage.

\*/

typedef int ListItemType; //in order to keep thigs simple

class List { // List class ADT

public:

// Destructor

virtual ~List() = default;

/\*

\* Removes all contents from the list, resetting it to an empty state.

\* This operation should reclaim any allocated memory if necessary.

\*/

virtual void clear() = 0;

/\*

\* Inserts an element "it" at the current position in the list.

\* The client must ensure that the list's capacity is not exceeded.

\* Shifts existing elements to the right if necessary.

\*

\* @param it - The element to be inserted.

\* @return True if insertion is successful, false otherwise.

\*/

virtual bool insert(const ListItemType& it) = 0;

/\*

\* Appends an element "it" at the end of the list.

\* The client must ensure that the list's capacity is not exceeded.

\*

\* @param it - The element to be appended.

\* @return True if the append operation is successful, false otherwise.

\*/

virtual bool append(const ListItemType& it) = 0;

/\*

\* Removes and returns the current element from the list.

\* Shifts remaining elements to the left if necessary.

\*

\* @return The removed element.

\*/

virtual ListItemType remove() = 0;

/\*

\* Moves the current position to the start of the list.

\* After execution, the first element (if any) will be the current element.

\*/

virtual void moveToStart() = 0;

/\*

\* Moves the current position to the end of the list.

\* After execution, the current position is set beyond the last element.

\*/

virtual void moveToEnd() = 0;

/\*

\* Moves the current position one step to the left (towards the start of the list).

\* If already at the beginning, the position remains unchanged.

\*/

virtual void prev() = 0;

/\*

\* Moves the current position one step to the right (towards the end of the list).

\* If already at the end, the position remains unchanged.

\*/

virtual void next() = 0;

/\*

\* Returns the number of elements currently in the list.

\*

\* @return The length of the list.

\*/

virtual int length() = 0;

/\*

\* Returns the index of the current position within the list.

\*

\* @return The current position index.

\*/

virtual int currPos() = 0;

/\*

\* Moves the current position to the specified index "pos."

\* If "pos" is out of bounds, the position remains unchanged.

\*

\* @param pos - The target position to move to.

\* @return True if the move is successful, false otherwise.

\*/

virtual bool moveToPos(int pos) = 0;

/\*

\* Checks if the current position is at the end of the list.

\*

\* @return True if the current position is at the end, false otherwise.

\*/

virtual bool isAtEnd() = 0;

/\*

\* Returns the value of the current element in the list.

\*

\* @return The value of the current element.

\*/

virtual ListItemType getValue() = 0;

/\*

\* Checks if the list is empty.

\*

\* @return True if the list contains no elements, false otherwise.

\*/

virtual bool isEmpty() = 0;

/\*

\* Retrieves the element at a specified index without modifying the current position.

\* If the index is out of bounds, the behavior is undefined and should be handled by implementations.

\*

\* @param pos - The index of the element to retrieve.

\* @return The element at the specified position.

\*/

virtual ListItemType get(int pos) const = 0;

};