# Doubly Linked List Specification (DBLLinkedList)

1. *General description*

   A **doubly linked list** is a data structure that stores a sequence of elements, where each element has references to both its **previous** and **next** elements. This bidirectional linking allows easy traversal in both directions. It enables efficient **insertion** and **deletion** at the beginning and end in constant time but requires more memory than singly linked lists due to the extra pointer.

2. *Stored data description*

   The *DBLLinkedList* class contains:

   - **Size** of the list (number of nodes).
   - A **Node structure** that stores relevant data.
   - **Pointers** to the previous and next nodes.
   - A **headNode pointer**, storing the reference to the first node.

3. *Operations specification (all functions are defined as member functions under DBLLinkedList::)*

   ### *Constructors*

   - Default constructor, `DBLLinkedList()`
     - Precondition: None
     - Postcondition: Sets *size* to *0* and *headNode* to *nullptr*.
     - Time Complexity: O(1)
   - Parameterized constructor, `DBLLinkedList(int n)`
     - Precondition: None
     - Postcondition: Creates a doubly linked list of length *n*.
     - Time Complexity: O(n) (as it initializes n nodes)
   - File constructor, `DBLLinkedList(std::wifstream& inputFile)`
     - Precondition: *inputFile* must exist.
     - Postcondition: Creates a doubly linked list where nodes contain data read from the file.
     - Time Complexity: O(n) (depending on the number of lines in the file)

   ### *Printing Functions*

   - `PrintList()`
     - Precondition: The list must exist.
     - Postcondition: Prints the elements in order, numbered from *1* to *n*.
     - Time Complexity: O(n)

- `PrintListCursor(DBLLinkedList::Node* currentNode)`
  - Precondition: The list must exist.
  - Postcondition: Prints the list, marking *currentNode*'s position with a “>” symbol.
  - Time Complexity: O(n)
- `PrintListReverse()`
  - Precondition: The list must exist.
  - Postcondition: Prints the elements in *reverse* order.
  - Time Complexity: O(n)

### *Retrieval and Modification*

- `GetNode(int index)`
  - Precondition: The list must exist, and *index* must be between *0* and *size*.
  - Postcondition: Returns the node at *index*.
  - Time Complexity: O(n) (worst case, if searching from the head)
- `GetData(int index)`
  - Precondition: The list must exist, and *index* must be between *0* and *size*.
  - Postcondition: Returns the data of the node at *index*.
  - Time Complexity: O(n)
- `SetValue(int index, DBLLinkedList::Node::Song data)`
  - Precondition: The list and *data* node must exist. *index* must be valid.
  - Postcondition: Updates the node at *index* with *data*.
  - Time Complexity: O(n)
- `InsertValue(int index, DBLLinkedList::Node::Song data)`
  - Precondition: The list and *data* node must exist. *index* must be valid.
  - Postcondition: Updates the node at *index* with *data*.
  - Time Complexity: O(n) (finding the position is O(n), inserting is O(1))
- `DeleteNode(int index)`
  - Precondition: The list must exist, and *index* must be valid.
  - Postcondition: Deletes the node at *index*.
  - Time Complexity: O(n) (finding the node is O(n), deleting is O(1))
- `SwapValues(int index1, int index2)`
  - Precondition: The list must exist, and both indices must be valid.
  - Postcondition: Swaps values of nodes at *index1* and *index2*.
  - Time Complexity: O(n) (finding the two nodes is O(n), swapping is O(1))

## *Utility Functions*

- `IsEmpty()`
  - Precondition: The list must exist.
  - Postcondition: Returns *true* if the list is empty, else *false*.
  - Time Complexity: O(1)
- `GetSize()`
  - Precondition: The list must exist.
  - Postcondition: Returns the size of the list.
  - Time Complexity: O(1)
- `GetIndex(DBLLinkedList::Node* data)`
  - Precondition: The list must exist.
  - Postcondition: Returns the index of *data* if found, else *-1*.
  - Time Complexity: O(n) (since it may need to traverse the entire list)
- `ShuffleList()`
  - Precondition: The list must exist.
  - Postcondition: Randomly shuffles the elements of the list.
  - Time Complexity: O(n)

## *Sorting*

- `IsSorted(const std::string& angleBracket)`
  - Precondition: The list must exist.
  - Postcondition:
    - If *angleBracket* == *"<"*, returns *true* if the list is sorted in **ascending** order.
    - If *angleBracket* == *">"*, returns *true* if the list is sorted in **descending** order.
    - Otherwise, returns *false*.
  - Time Complexity: O(n)
- `SortList(const std::string&)`
  - Precondition: The list must exist.
  - Postcondition:
    - If *angleBracket* == *"<"*, sorts the list in **ascending** order.
    - If *angleBracket* == *">"*, sorts the list in **descending** order.
    - Uses *Quicksort* with *Hoare* partitioning. Best/Average case: O(n log n), Worst case: $O(n^2)$ (when the list is already sorted but in reverse order, leading to unbalanced partitions).