SWE3 - Übung 6

WS 2025/26

Aufwand in h: 3

Tim Peko

Inhaltsverzeichnis

1. Aufgabe: Doppelt verkettete Liste 2
   1.1. Lösungsidee 2
      1.1.1. Doppelt verkettete Liste 2
         1.1.1.1. Interne Struktur der Liste 2
         1.1.1.2. Designentscheidungen 2
      1.1.2. Iterator-Implementierung 3
         1.1.2.1. Bidirektionaler Iterator 3
         1.1.2.2. Iterator-Operationen 3
         1.1.2.3. Iterator-Struktur 3
      1.1.3. Komplexitätsanalyse 3
      1.1.4. Foreach-Methode 4
      1.1.5. Design-Überlegung: Iteration und Modifikation 4
         1.1.5.1. Problemstellung 4
         1.1.5.2. Analyse 5
         1.1.5.3. Sichere Lösung: remove_if-Muster 5
         1.1.5.4. Alternative: Stabile Iteratoren 5
   1.2. Teststrategie 6
   1.3. Ergebnisse 6
      1.3.1. Code-Metriken 6
      1.3.2. Testfälle 7

1. Aufgabe: Doppelt verkettete Liste

1.1. Lösungsidee

Die Aufgabe besteht darin, eine generische doppelt verkettete Liste (`DoublyLinkedList<T>`) zu implementieren, die einen bidirektionalen Iterator zur Verfügung stellt. Die Implementierung folgt modernen C++-Prinzipien und ist mit der C++ Standard Library kompatibel.

1.1.1. Doppelt verkettete Liste

1.1.1.1. Interne Struktur der Liste

Die Liste verwendet eine klassische Sentinel-Knoten-Architektur:

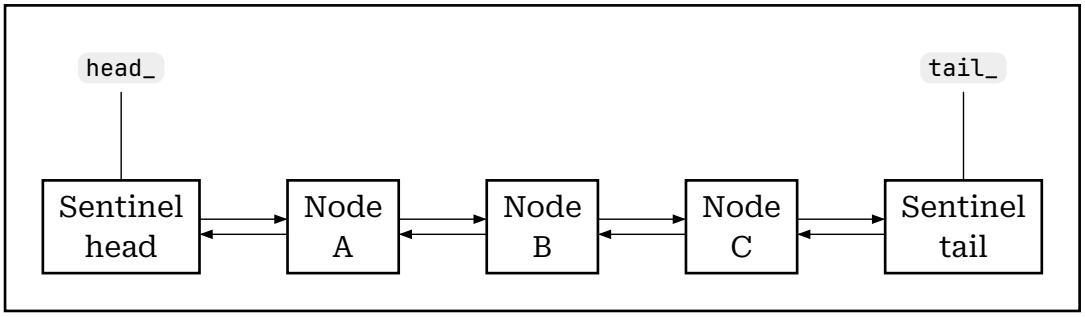

Abbildung 1: Visualisierung der Sentinel-Knoten Architektur

Jeder Knoten (`Node<T>`) enthält:
- `data`: Das gespeicherte Element vom Typ `T`
- `prev`: Zeiger auf den vorherigen Knoten
- `next`: Zeiger auf den nächsten Knoten

Die Sentinel-Knoten (`head_` und `tail_`) sind immer vorhanden und enthalten keine echten Daten. Sie dienen als Markierungen für den Anfang und das Ende der Liste und vereinfachen die Implementierung von Einfüge- und Löschoperationen erheblich.

1.1.1.2. Designentscheidungen

1. Sentinel-Knoten: Durch die Verwendung von Sentinel-Knoten am Anfang und Ende der Liste werden Sonderfälle bei Einfüge- und Löschoperationen vermieden. `begin()` zeigt immer auf `head_→next`, `end()` zeigt immer auf `tail_`.

2. Template-Implementierung: Die Liste ist vollständig templatebasiert, um beliebige Datentypen (primitive Typen, Objekte, Smart Pointer) zu unterstützen.

3. Gecachte Größe: Die Größe wird in `size_` gecached, um `size()` in $O(1)$ zu ermöglichen.

4. Move-Semantik: Alle relevanten Methoden unterstützen Move-Semantik für effiziente Handhabung von nicht-kopierbaren oder teuren Objekten.

5. STL-Kompatibilität: Die Iteratoren sind vollständig STL-konform und ermöglichen die Verwendung mit Standard-Algorithmen wie `std::find(...)`, `std::count(...)`, `std::for_each(...)` etc.

6. Öffentlicher Namespace: Die Klassen sind im globalen Namespace definiert, um die Verwendung zu vereinfachen.

1.1.2. Iterator-Implementierung

1.1.2.1. Bidirektionaler Iterator

Die Wahl eines **bidirektionalen Iterators** (`std::bidirectional_iterator_tag`) ist natürlich für eine doppelt verkettete Liste:

Begründung:
1. Die doppelte Verkettung ermöglicht natürlich Vorwärts- und Rückwärtstraversierung
2. Ein Random-Access-Iterator wäre nicht sinnvoll, da Indexzugriff $O(n)$ erfordern würde
3. Ein Forward-Iterator würde die Rückwärtstraversierung nicht unterstützen

1.1.2.2. Iterator-Operationen

```cpp
// STL-konformer Iterator-Typ-Alias
using iterator_category = std::bidirectional_iterator_tag;
using value_type        = T;
using difference_type   = std::ptrdiff_t;
using pointer           = T*;
using reference         = T&;

// Operationen
*it          // Dereferenzierung
it->member   // Zeigerzugriff
++it, it++   // Inkrement (vorwärts)
--it, it--   // Dekrement (rückwärts)
it1 == it2   // Gleichheit
it1 != it2   // Ungleichheit
```

1.1.2.3. Iterator-Struktur

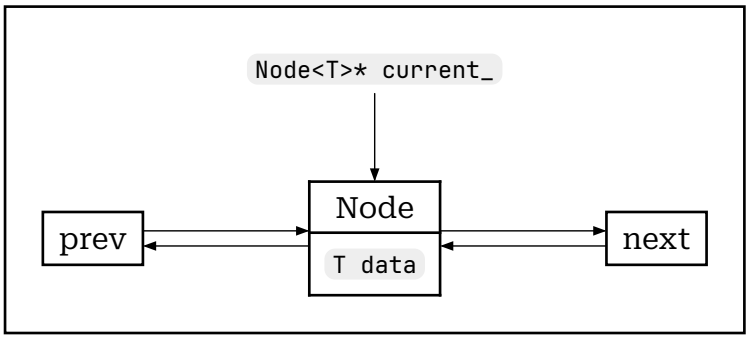

Abbildung 2: Visualisierung der Iterator-Struktur

Der Iterator speichert einen Zeiger auf den aktuellen Knoten und navigiert durch Zugriff auf `prev` und `next`.

1.1.3. Komplexitätsanalyse

Operation	Komplexität	Begründung
`push_front`	$O(1)$	Direkter Zugriff auf `head_`

`push_back`	$O(1)$	Direkter Zugriff auf `tail_`
`pop_front`	$O(1)$	Direkter Zugriff auf `head_->next`
`pop_back`	$O(1)$	Direkter Zugriff auf `tail_->prev`
`size`	$O(1)$	Gecachte Größe
`find`	$O(n)$	Lineare Suche erforderlich
`begin/end`	$O(1)$	Direkter Zugriff auf Sentinel-Knoten
`foreach`	$O(n)$	Traversierung aller Elemente
`insert`	$O(1)$	Nur Zeiger-Operationen
`erase`	$O(1)$	Nur Zeiger-Operationen

1.1.4. Foreach-Methode

Die `foreach`-Methode akzeptiert jeden aufrufbaren Typ durch Template-Parameter:

```cpp
// Template-based (efficient, allows inlining)
template <typename Func>
void foreach(Func func);

// Usage examples:
list.foreach([](int& x) { x *= 2; });           // Lambda
list.foreach(MyFunctor{});                      // Functor
list.foreach(&myFunction);                      // Function pointer
```

Vorteile des Template-Ansatzes:
- Keine Laufzeit-Overhead durch `std::function`
- Compiler kann Aufrufe inlinen
- Beliebige aufrufbare Typen werden akzeptiert

1.1.5. Design-Überlegung: Iteration und Modifikation

1.1.5.1. Problemstellung

Kann die Liste *während* der Iteration verändert werden? Insbesondere: Können Elemente gelöscht werden?

1.1.5.2. Analyse

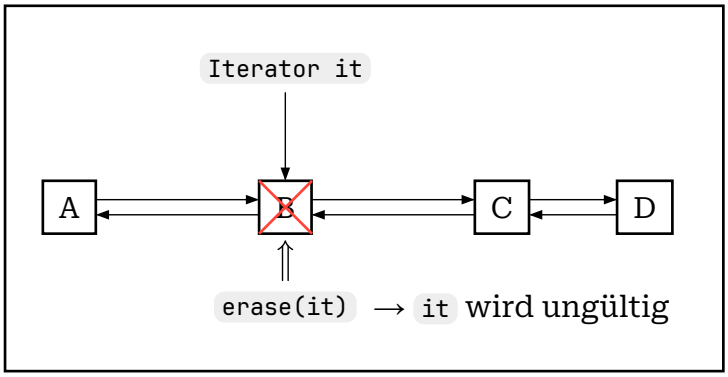

Abbildung 3: Problematische Situation bei intra-iterativer Entfernung eines Knoten

Bei einer naiven Implementierung würde das Löschen des Knotens, auf den der Iterator zeigt, zu einem *dangling pointer* führen. Der Iterator zeigt dann auf freigegebenen Speicher.

1.1.5.3. Sichere Lösung: remove_if-Muster

Die implementierte Lösung verwendet das *remove_if*-Muster:

```cpp
template <typename Predicate>
size_type remove_if(Predicate pred) {
    size_type removed = 0;
    iterator it = begin();
    while (it != end()) {
        if (pred(*it)) {
            it = erase(it);  // erase returns iterator to NEXT element
            ++removed;
        } else {
            ++it;
        }
    }
    return removed;
}
```

Schlüsselprinzip: `erase()` gibt einen Iterator auf das *nächste gültige Element* zurück.

1.1.5.4. Alternative: Stabile Iteratoren

Eine fortgeschrittenere Lösung wäre die Implementierung *stabiler Iteratoren*:

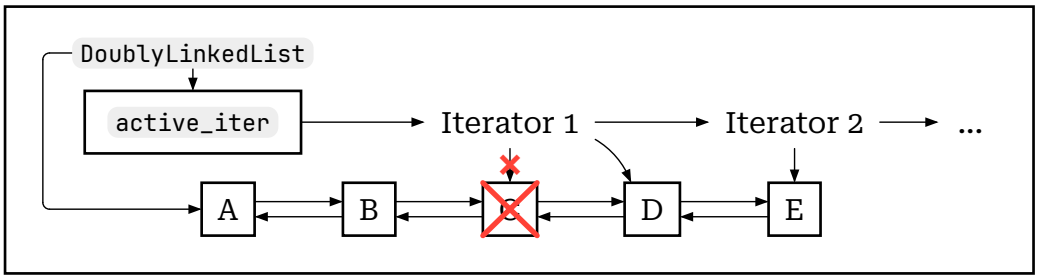

Abbildung 4: Stabile Iterator-Architektur

Bei dieser Architektur würde:
1. Die Liste alle aktiven Iteratoren tracken

2. Bei `erase()` alle betroffenen Iteratoren auf das nächste Element umleiten
3. Der Speicher-Overhead und die Komplexität deutlich steigen

Fazit: Für die meisten Anwendungsfälle ist das *remove_if*-Muster ausreichend und effizienter.

1.2. Teststrategie

Die Tests folgen dem AAA-Prinzip (Arrange-Act-Assert) und decken folgende Kategorien ab:

- Konstruktor-Tests
- Basis-Operationen
- Such-Operationen
- Iterator-Tests
- Foreach-Tests
- Insert/Erase-Tests
- Edge Cases
- STL-Kompatibilität

In Abschnitt 1.3.2 sind die Testfälle und deren Ergebnisse tabellarisch aufgelistet.

1.3. Ergebnisse

Die Implementierung erfüllt alle geforderten Anforderungen:

1. **Template-basiert:** Unterstützt beliebige Datentypen
2. **O(1) Operationen:** `push_front`, `push_back` sind konstant
3. **Bidirektionaler Iterator:** STL-konform mit korrektem `iterator_tag`
4. **foreach-Methode:** Akzeptiert Lambdas, Funktoren und Funktionszeiger
5. **Sichere Modifikation:** `remove_if` ermöglicht sicheres Löschen während Iteration
6. **Umfassende Tests:** Über 100 Unit-Tests decken alle Funktionalitäten und Edge Cases ab.

1.3.1. Code-Metriken

Typ	Codezeilen	Kommentarzeilen	Leerzeilen
C Header	391	430	102
C++	789	229	320
NuGet Config	4	0	0
Visual Studio Project	146	0	0
Total	1330	659	422

Tabelle 1: Code-Metriken für das Übungsprojekt

Weil die tabellarische Code-Metrik Darstellung in Tabelle 1 noch nciht genug ist, sind die gleichen Daten nochmal grafisch in Abbildung 5 dargestellt.

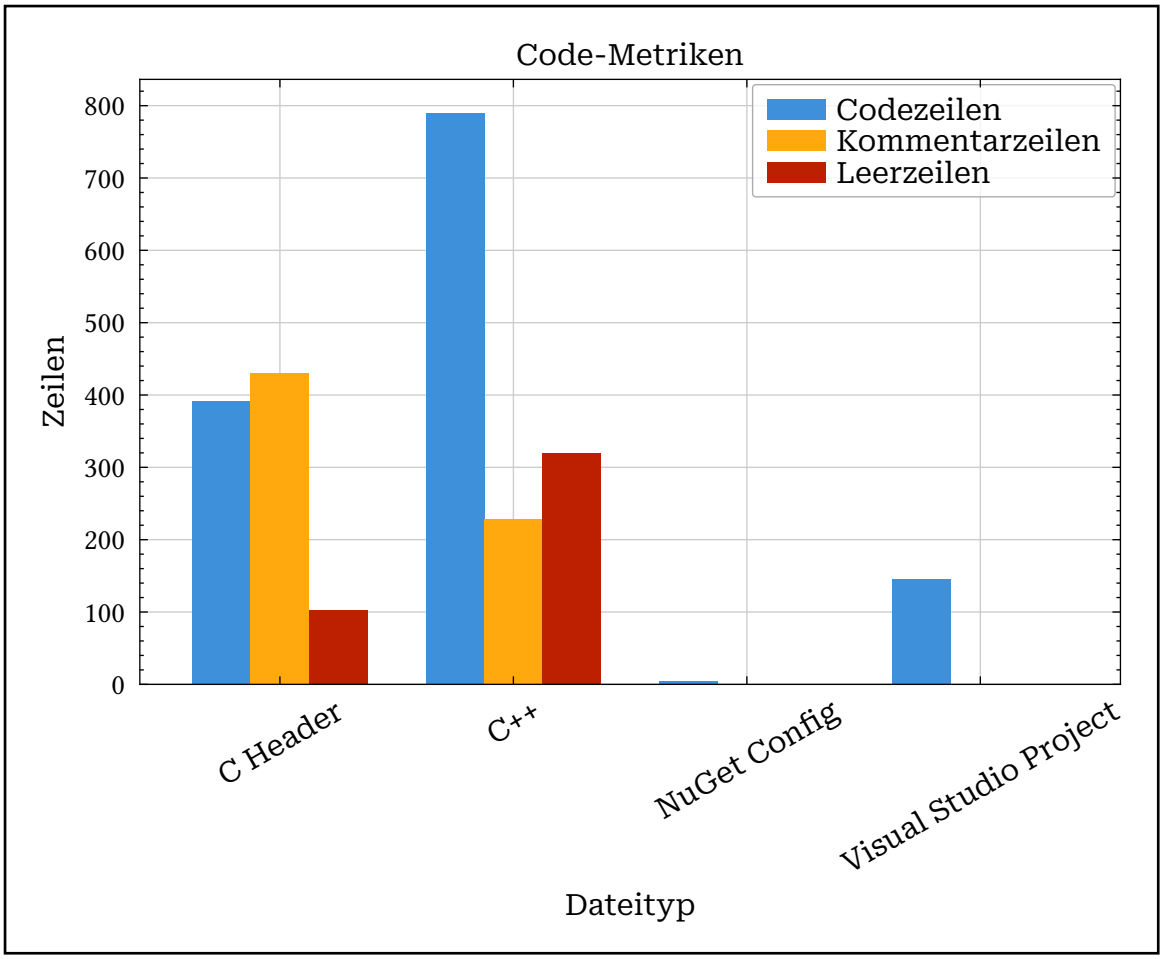

Abbildung 5: Grafische Darstellung der Code-Metriken

1.3.2. Testfälle

Das Google-Test Framework wird in der `main.cpp` aufgerufen und erlaubt so, die Testfälle aus der kompilierten Binärdatei auszuführen. Mit dem `--gtest_output=json:assets/test_resuls.json`-Flag wird die Ausgabe in Datei geschrieben, die grob in Tabelle 2 und detailliert in Tabelle 3 dargestellt wird.

Testsuite	Tests	Fehler
Total	114	0
DoublyLinkedListConstructor	9	0
DoublyLinkedListTest	93	0
DoublyLinkedListEdgeCases	8	0
DoublyLinkedListMemory	2	0
DoublyLinkedListStress	2	0

Tabelle 2: Übersicht der Testfälle und Ergebnisse

Testname	Status	Zeit
DoublyLinkedListConstructor		
DefaultConstructor_CreatesEmptyList	✓ Erfolgreich	0s
InitializerListConstructor_CreatesListWithElements	✓ Erfolgreich	0s

Testname	Status	Zeit
DoublyLinkedListConstructor		
InitializerListConstructor_EmptyInitializerList	✓ Erfolgreich	0s
CopyConstructor_CreatesIndependentCopy	✓ Erfolgreich	0s
CopyConstructor_EmptyList	✓ Erfolgreich	0s
MoveConstructor_TransfersOwnership	✓ Erfolgreich	0s
CopyAssignment_CopiesToExistingList	✓ Erfolgreich	0s
CopyAssignment_SelfAssignment	✓ Erfolgreich	0s
MoveAssignment_TransfersOwnership	✓ Erfolgreich	0s
DoublyLinkedListTest		
PushFront_OnEmptyList_AddsElement	✓ Erfolgreich	0s
PushFront_OnNonEmptyList_AddsAtBeginning	✓ Erfolgreich	0s
PushFront_MultipleElements_MaintainsOrder	✓ Erfolgreich	0s
PushFront_MoveSemantics_MovesValue	✓ Erfolgreich	0s
PushBack_OnEmptyList_AddsElement	✓ Erfolgreich	0s
PushBack_OnNonEmptyList_AddsAtEnd	✓ Erfolgreich	0s
PushBack_MultipleElements_MaintainsOrder	✓ Erfolgreich	0s
PushBack_MoveSemantics_MovesValue	✓ Erfolgreich	0s
Size_EmptyList_ReturnsZero	✓ Erfolgreich	0s
Size_SingleElement_ReturnsOne	✓ Erfolgreich	0s
Size_MultipleElements_ReturnsCorrectCount	✓ Erfolgreich	0s
Size_AfterPushAndPop_ReturnsCorrectCount	✓ Erfolgreich	0s
Empty_EmptyList_ReturnsTrue	✓ Erfolgreich	0s
Empty_NonEmptyList_ReturnsFalse	✓ Erfolgreich	0s
Empty_AfterClear_ReturnsTrue	✓ Erfolgreich	0s
Find_ExistingElement_ReturnsValidIterator	✓ Erfolgreich	0s
Find_NonExistingElement_ReturnsEndIterator	✓ Erfolgreich	0s
Find_EmptyList_ReturnsEndIterator	✓ Erfolgreich	0s
Find_FirstElement_ReturnsBeginIterator	✓ Erfolgreich	0s
Find_LastElement_ReturnsValidIterator	✓ Erfolgreich	0s
Find_DuplicateElements_ReturnsFirstOccurrence	✓ Erfolgreich	0s
Find_WithStrings_FindsCorrectly	✓ Erfolgreich	0s
Find_ConstList_ReturnsConstIterator	✓ Erfolgreich	0s
Contains_ExistingElement_ReturnsTrue	✓ Erfolgreich	0s
Contains_NonExistingElement_ReturnsFalse	✓ Erfolgreich	0s
Contains_EmptyList_ReturnsFalse	✓ Erfolgreich	0s
Front_NonEmptyList_ReturnsFirstElement	✓ Erfolgreich	0s
Back_NonEmptyList_ReturnsLastElement	✓ Erfolgreich	0s

Testname	Status	Zeit
DoublyLinkedListTest		
Front_EmptyList_ThrowsException	✓ Erfolgreich	0.001s
Back_EmptyList_ThrowsException	✓ Erfolgreich	0s
Front_Modifiable_ChangesElement	✓ Erfolgreich	0s
Back_Modifiable_ChangesElement	✓ Erfolgreich	0s
PopFront_NonEmptyList_RemovesFirstElement	✓ Erfolgreich	0.001s
PopBack_NonEmptyList_RemovesLastElement	✓ Erfolgreich	0s
PopFront_SingleElement_MakesListEmpty	✓ Erfolgreich	0s
PopBack_SingleElement_MakesListEmpty	✓ Erfolgreich	0s
PopFront_EmptyList_ThrowsException	✓ Erfolgreich	0s
PopBack_EmptyList_ThrowsException	✓ Erfolgreich	0s
Clear_NonEmptyList_MakesListEmpty	✓ Erfolgreich	0s
Clear_EmptyList_RemainsEmpty	✓ Erfolgreich	0s
Clear_ListCanBeReused	✓ Erfolgreich	0s
Begin_NonEmptyList_ReturnsIteratorToFirstElement	✓ Erfolgreich	0s
End_NonEmptyList_ReturnsIteratorPastLastElement	✓ Erfolgreich	0s
BeginEnd_EmptyList_AreEqual	✓ Erfolgreich	0s
CBegin_ReturnsConstIterator	✓ Erfolgreich	0s
Iterator_PreIncrement_MovesToNextElement	✓ Erfolgreich	0s
Iterator_PostIncrement_MovesToNextElement	✓ Erfolgreich	0s
Iterator_PreDecrement_MovesToPreviousElement	✓ Erfolgreich	0s
Iterator_PostDecrement_MovesToPreviousElement	✓ Erfolgreich	0s
Iterator_ArrowOperator_AccessesMember	✓ Erfolgreich	0s
Iterator_Equality_ComparesCorrectly	✓ Erfolgreich	0s
Iterator_BidirectionalTraversal_WorksCorrectly	✓ Erfolgreich	0s
Iterator_Modification_ChangesListElement	✓ Erfolgreich	0s
Iterator_Category_IsBidirectional	✓ Erfolgreich	0s
Iterator_TypeAliases_AreCorrect	✓ Erfolgreich	0s
ConstIterator_TypeAliases_AreCorrect	✓ Erfolgreich	0s
RangeBasedFor_TraversesAllElements	✓ Erfolgreich	0s
RangeBasedFor_ModifiesElements	✓ Erfolgreich	0s
RangeBasedFor_EmptyList_NoIterations	✓ Erfolgreich	0s
Foreach_Lambda_AppliesFunction	✓ Erfolgreich	0s
Foreach_Lambda_ModifiesElements	✓ Erfolgreich	0s
Foreach_EmptyList_NoInvocations	✓ Erfolgreich	0s
Foreach_Functor_AppliesFunction	✓ Erfolgreich	0s
Foreach_FunctionPointer_AppliesFunction	✓ Erfolgreich	0s

Testname	Status	Zeit
DoublyLinkedListTest		
Foreach_ConstList_OnlyReadsElements	✓ Erfolgreich	0s
Foreach_WithCapturingLambda_AccessesExternalState	✓ Erfolgreich	0s
Foreach_StdFunction_AppliesFunction	✓ Erfolgreich	0s
Insert_AtBegin_AddsAtStart	✓ Erfolgreich	0s
Insert_AtEnd_AddsAtEnd	✓ Erfolgreich	0s
Insert_InMiddle_InsertsCorrectly	✓ Erfolgreich	0s
Erase_FirstElement_RemovesCorrectly	✓ Erfolgreich	0s
Erase_LastElement_RemovesCorrectly	✓ Erfolgreich	0s
Erase_MiddleElement_RemovesCorrectly	✓ Erfolgreich	0s
Erase_Range_RemovesMultipleElements	✓ Erfolgreich	0s
RemoveIf_RemovesMatchingElements	✓ Erfolgreich	0s
RemoveIf_NoMatches_RemovesNothing	✓ Erfolgreich	0s
RemoveIf_AllMatch_ClearsList	✓ Erfolgreich	0s
RemoveIf_EmptyList_ReturnsZero	✓ Erfolgreich	0.001s
RemoveIf_ConsecutiveElements_RemovesCorrectly	✓ Erfolgreich	0s
Equality_SameLists_ReturnsTrue	✓ Erfolgreich	0s
Equality_DifferentElements_ReturnsFalse	✓ Erfolgreich	0s
Equality_DifferentSizes_ReturnsFalse	✓ Erfolgreich	0s
Equality_BothEmpty_ReturnsTrue	✓ Erfolgreich	0s
StdFind_FindsElement	✓ Erfolgreich	0s
StdCount_CountsElements	✓ Erfolgreich	0s
StdFindIf_FindsMatchingElement	✓ Erfolgreich	0s
StdTransform_ModifiesElements	✓ Erfolgreich	0.001s
StdAccumulate_SumsElements	✓ Erfolgreich	0s
StdAllOf_ChecksAllElements	✓ Erfolgreich	0s
StdAnyOf_ChecksSomeElements	✓ Erfolgreich	0s
StdNoneOf_ChecksNoElements	✓ Erfolgreich	0s
StdForEach_AppliesFunction	✓ Erfolgreich	0.001s
ConstList_AllowsReadOperations	✓ Erfolgreich	0s
DoublyLinkedListEdgeCases		
LargeList_HandlesCorrectly	✓ Erfolgreich	0.006s
AlternatingPushFrontBack_MaintainsOrder	✓ Erfolgreich	0s
StringType_WorksCorrectly	✓ Erfolgreich	0s
CustomType_WorksCorrectly	✓ Erfolgreich	0s
UniquePtr_MoveOnlyType	✓ Erfolgreich	0s
RepeatedClearAndRefill_WorksCorrectly	✓ Erfolgreich	0s

Testname	Status	Zeit
DoublyLinkedListEdgeCases		
IteratorInvalidation_AfterPush	✓ Erfolgreich	0s
BackwardIteration_FromEnd	✓ Erfolgreich	0s
DoublyLinkedListMemory		
DestructorFreesAllMemory	✓ Erfolgreich	0.001s
ClearFreesElementMemory	✓ Erfolgreich	0.003s
DoublyLinkedListStress		
ManyPushPop_MaintainsIntegrity	✓ Erfolgreich	0.001s
ManyCopies_WorkCorrectly	✓ Erfolgreich	0s

Tabelle 3: Detailierte Ergebnisse der Testfälle