

1 Praktikum 1: Twitter-Programm als doppelt verkettete Liste

Die Hauptaufgabe im ersten Praktikum von ADS ist es, ein kleines Twitter-Programm in C++ zu erstellen. Dazu werden von der Tastatur kleine Tweets (Nachrichten) eingelesen. Diese werden mit Datum und Uhrzeit in eine (doppelt) verkettete Liste gespeichert. Alle bisherigen Tweet-Nachrichten sollen nach der Eingabe angezeigt werden.

Um den Einstieg leichter zu gestalten, wird die Aufgabe in 3 Teilaufgaben strukturiert. Zu jeder Teilaufgabe legen Sie bitte ein Projekt an, damit Sie die einzelnen Entwicklungsschritte im Praktikum erklären und zeigen können.

Aufgabe 1.1: Programmierung einer Datenstruktur, die die Funktionalität einer doppelt verketteten Liste mit integer Werten erfüllt.

Aufgabe 1.2: Umprogrammierung der Datenstruktur aus Aufgabe 1.1 in eine Template-basierte Datenstruktur, um auch mit anderen Datentypen als integer zu arbeiten

Aufgabe 1.3: Die Umsetzung der Twitter-Aufgabe mithilfe der Template-Datenstruktur aus Aufgabe 1.2 und einem zusammengesetzten Datentypen *tweet* als Template-Datentyp.

In jeder Teilaufgabe erhalten Sie Lösungshinweise und Testbeispiele und vorgegebenen Programmier-Code. Vervollständigen Sie die gegebenen Programmteile. Gehen Sie unbedingt in der Reihenfolge der Aufgabenstellung vor.

1.1 Aufgabenstellung

1. Der erste Teil der Aufgabe besteht darin, dass Sie eine Klasse bereit stellen, die die Datenstruktur einer dynamisch, doppelt verketteten Liste zur Verfügung stellt. Um die Aufgabe zunächst zu vereinfachen gehen wir von der Speicherung von integer-Werten aus. Zur Speicherung der Daten benötigen wir eine Klasse Node (stellt den klassischen Node mit next-/prev-Pointer dar). Die Klasse List dient zur Verwaltung der Datenstruktur für die verkettete Liste und die Klasse Node zur Speicherung ihrer integer Werte (Knoten).

Wir benötigen folgende Dateien:

Node.h	Headerdatei der Klasse Node
Node.cpp	c++ Quelle der Klasse Node
List.h	Headerdatei der Klasse List
List.cpp	c++ Quelle der Klasse List
main.cpp	Testprogramm für die Klasse List

Diese Dateien liegen als Download auf dem ILIAS-Server (Praktikum>Praktikum 1>Praktikum 1 Vorlage>Vorlage_1.1[zip-Archiv]).

Kopieren Sie die Dateien in ein neues C++ Projekt. Verändern Sie **nicht** die Headerdateien und auch nicht die Datei "Node.cpp".

Die Klasse List stellt eine doppelt Verkettete Liste bereit. Jeder neue Eintrag in die Liste vergrößert diese. In einem Objekt der Klasse List gibt es mindestens ein Objekt der Klasse Node. Das erste Objekt der Klasse Node wird im Konstruktor der Klasse List erzeugt (Anker: Head_Tail). Dieses Objekt zeigt auf den **Anfang und gleichzeitig auf das Ende** der Liste.

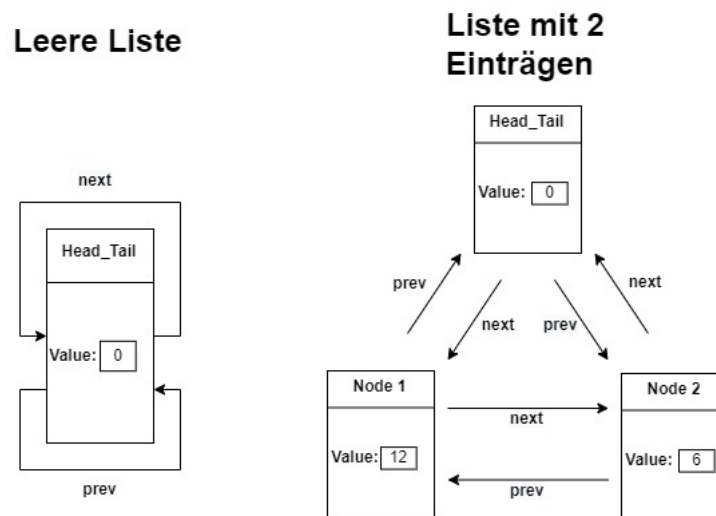


Abbildung 1: Aufbaustruktur der Liste (jeweils Leer und mit 2 Einträgen)

In der C++ Quelle der Klasse List (List.cpp) sind einige Methoden nur allgemein sprachlich in der zu erfüllenden Funktion beschrieben, der Funktionsrumpf muss von Ihnen definiert werden. Fügen Sie den fehlenden Code in die Quelldateien unter den Kommentaren mit **//ToDo** ein.

(a) Dekonstruktor

List::~~List() (List.cpp Zeile 46)

Implementieren Sie den Dekonstruktor, der alle Knoten der Liste löscht und damit den allokierten Speicher wieder freigibt, sodass kein memory leak Fehler entsteht.

(b) Knoten am Anfang einfügen **void List::insertFront(int value)** (List.cpp Zeile 57)

Erzeugen Sie einen neuen Knoten mit dem Wert *value* und fügen Sie diesen in die doppelt verkettete Liste am **Anfang** ein.

(c) Liste am Anfang einfügen

void List::insertFront(List& _List) (List.cpp Zeile 68)

Einfügen der Liste *_List* an den **Anfang** der vorhandenen Liste. Die Knoten sollen weiterverwendet werden. D.h. die Knoten dürfen nicht gelöscht und einfach neu erzeugt werden.

Die Zeiger der einzelnen Knoten müssen entsprechend neu zugewiesen werden, sodass am ende eine neue Liste entsteht. Entsprechend wie bei der Funktion in (b) nur mit einer Liste.

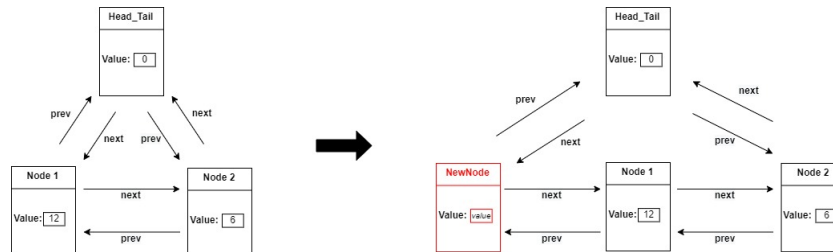


Abbildung 2: Liste nach Benutzen der *insertFront*-Methode (ist der Parameter eine Liste gilt entsprechendes)

(d) Knoten am Ende einfügen

void List::insertBack(int value) (List.cpp Zeile 91)

Erzeugen Sie einen neuen Knoten mit dem Wert *value* und fügen diesen in die doppelt verkettete Liste am **Ende** ein.

(e) Liste am Ende einfügen

void List::insertBack(List& _List) (List.cpp Zeile 102)

Einfügen der Liste *_List* an das **Ende** der vorhandenen Liste. Die Knoten sollen weiterverwendet werden. D.h. die Knoten dürfen nicht gelöscht und einfach neu erzeugt werden. Die Zeiger der einzelnen Knoten müssen entsprechend neu zugewiesen werden, sodass am zum Schluss eine neue Liste entsteht. Entsprechend wie bei der Funktion in (d) nur mit einer Liste.

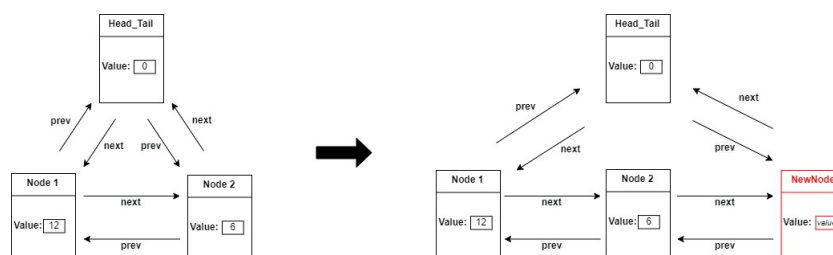


Abbildung 3: Liste nach Benutzen der *insertBack*-Methode (ist der Parameter eine Liste gilt entsprechendes)

(f) Knoten am Anfang entnehmen

bool List::getFront(int& value) (List.cpp Zeile 126)

Der erste Knoten der Liste soll gesucht und anschließend der Wert (*value*) dieses Knoten in den Parameter *value* geschrieben werden. Daraufhin soll der Knoten aus der Liste gelöscht

und die Listen-Struktur aktualisiert werden. Das Löschen darf **nicht** von einer anderen Methode (z.B. delete) übernommen werden.

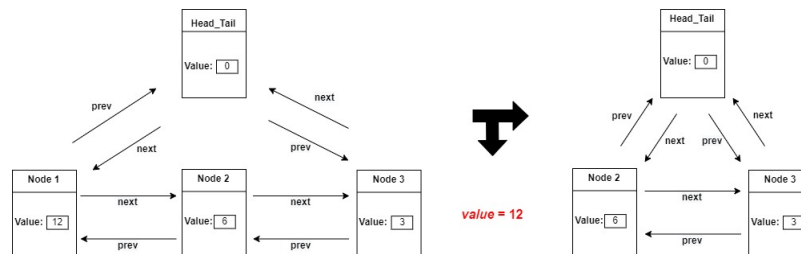


Abbildung 4: Liste nach Benutzen der *getFront* Methode

(g) Knoten am Ende entnehmen

bool List::getBack(int& value (List.cpp Zeile 147)

Der letzte Knoten der Liste soll gesucht und anschließend der Wert (*value*) dieses Knoten in den Parameter *value* geschrieben werden. Daraufhin soll der Knoten aus der Liste gelöscht und die Listen-Struktur aktualisiert werden. Das Löschen darf **nicht** von einer anderen Methode (z.B. delete) übernommen werden.

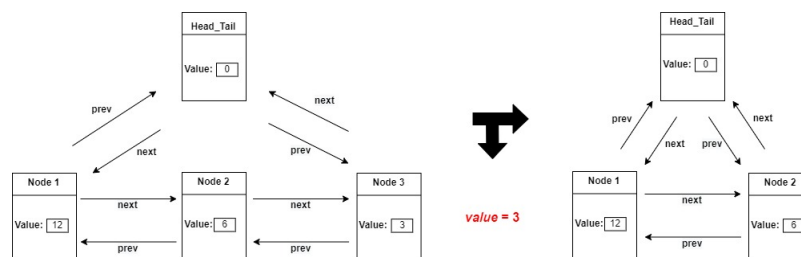


Abbildung 5: Liste nach Benutzen der *getBack* Methode

(h) Beliebigen Knoten löschen

bool List::del(int value) (List.cpp Zeile 170)

Realisiert das Löschen eines beliebigen Knotens aus der Liste. Der zu löschende Knoten wird über den Parameter *value* identifiziert. Wird ein solcher Knoten gefunden, wird dieser gelöscht. Im Anschluss wird die Listen-Struktur aktualisiert und die Größe der Liste angepasst.

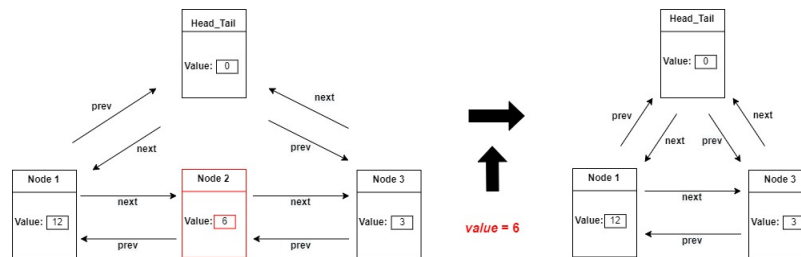


Abbildung 6: Liste nach Benutzen der *delete* Methode

(i) Knoten Suchen

bool List::search(int value) (List.cpp Zeile 190)

Realisiert das Suchen eines beliebigen Knotens in der Liste. Der zu suchende Knoten wird über den Parameter *value* identifiziert. Wird ein solcher Knoten gefunden, soll *true* von der Funktion zurückgeschickt werden. Wird der Knoten in der Liste nicht gefunden, soll *false* zurückgeschickt werden. Die Methode verändert **nicht** die Listenstruktur und Ihre Einträge.

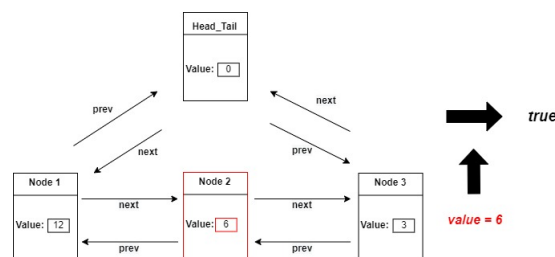


Abbildung 7: Liste nach Benutzen der *search* Methode

(j) Beliebige Knoten tauschen

bool List::swap(int value1, value2) (Liste.cpp Zeile 210)

Realisiert das Tauschen von 2 beliebigen Knoten aus der Liste. Die Parameter *value1* und *value2* dienen zum identifizieren der zu tauschenden Knoten. Wird nur einer oder keiner der Knoten gefunden soll *false* zurückgeschickt werden. Die Knoten dürfen beim tauschen nicht gelöscht werden. Der Tausch soll durch eine Neuzuweisung der Zeiger realisiert werden. Wichtig ist, dass es hier im allgemeinen **zwei** Fälle zu beachten gibt. Zum einen der Fall, dass die Knoten direkt nebeneinander liegen und zum anderen den Fall, dass die Knoten nicht nebeneinander liegen. Das **muss** von Ihnen beim implementieren dringend beachtet werden. Ist der Tausch erfolgreich, soll *true* zurückgeschickt werden.

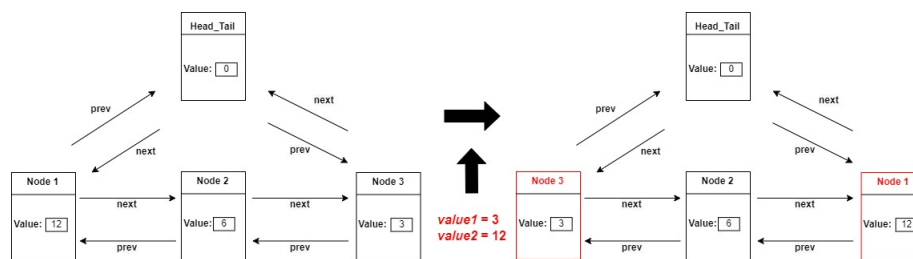


Abbildung 8: Liste nach Benutzen der *swap* Methode

(k) Anzahl Knoten in der Liste

int List::size() (List.cpp Zeile 233)

Diese Funktion soll die Anzahl der aktuell in der Liste vorkommenden Knoten zurückschicken. Dabei soll die Funktion nicht einfach die Liste traversieren und dabei jeden Knoten mit zählen, die Funktion soll mit einem konstanten Aufwand ($O(1)$) die Anzahl der Knoten zurückschicken.

Tipp: Die Anzahl der Knoten kann nach jeder der o.g. Operationen aktualisiert werden.

Die main.cpp / test.cpp enthält ein Testprogramm mit dem die Methoden der Klasse geprüft werden können (Unittest). Außerdem steht in der Klasse List eine Methode test zur Verfügung. Diese Methode überprüft die Zeiger in next- und dann in prev-Richtung. Mit diesem Hilfsmittel können Sie überprüfen welche Funktion nicht korrekt arbeitet und so die Fehler besser zu lokalisieren. Dieser Sachverhalt wird mit den folgenden zwei Bildern veranschaulicht.

```
C:\Users\VR_Mitarbeiter_Lukas\Desktop\praktikum_test\P1\x64\Debug\p1_buggy.exe

=====
is a Catch v2.5.0 host application.
Run with -? for options

-----
List Testing
  Hinzufuegen von Nodes aus zweiter Liste am Anfang
-----
C:\Users\VR_Mitarbeiter_Lukas\Desktop\praktikum_test\P1\p1_buggy\Test.cpp(72)
-----
C:\Users\VR_Mitarbeiter_Lukas\Desktop\praktikum_test\P1\p1_buggy\Test.cpp(90): FAILED:
  REQUIRE( value == 7 )
with expansion:
  5 == 7

=====
test cases: 1 | 0 passed | 1 failed
assertions: 36 | 35 passed | 1 failed
```

Abbildung 9: Fehler in "Hinzufügen von Nodes aus Liste am Anfang"

```
C:\Users\VR_Mitarbeiter_Lukas\Desktop\praktikum_test\P1\x64\Debug\p1_buggy.exe

=====
all tests passed (42 assertions in 1 test case)

< 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109 >>
100: gefunden
109: nicht gefunden
```

Abbildung 10: Keine Fehler

Bevor Sie weitermachen: **Testen Sie mit Hilfe des Debuggers, ob der Dekonstruktor korrekt arbeitet. Dies führen Sie im Praktika Ihrem Betreuer vor.**

2. Der zweite Teil der Aufgabe besteht darin, dass Sie die Klasse List und die Klasse Node je zu einer Template Klasse umbauen.

Gehen Sie wie folgt vor:

Kopieren Sie die Dateien main.cpp, Node.h, Node.cpp, List.h und List.cpp in ein neues Projekt. Der Unit-Test wird nicht mehr benötigt. Diesen sollten Sie jetzt deaktivieren (Dafür in der main.cpp die Zeilen 2, 4 und 9 löschen). Anschließend wird der Programmteil von Node.cpp in die Datei Node.h kopiert (unterhalb der Klassendefinition). Anschließend kopieren Sie den Programmteil von List.cpp in List.h und löschen dann die Dateien Node.cpp und List.cpp.

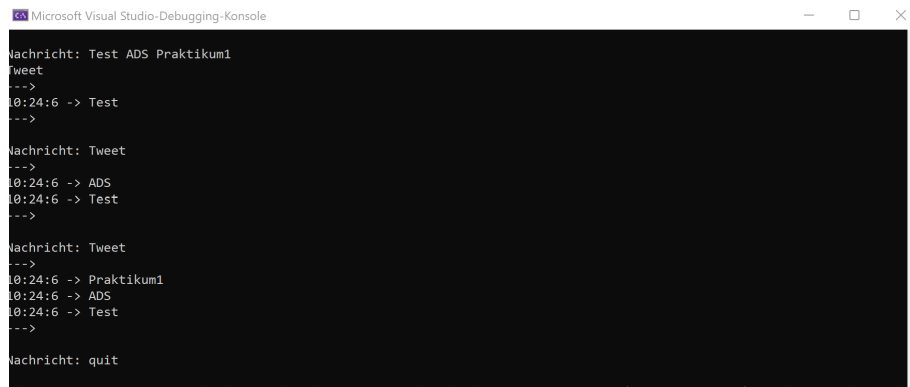
Jetzt bauen Sie beide Klassen zu Template Klassen um. Dafür wird überall in der Datei Node.h der Datentyp die variable *value* in ein template-Datentyp geändert. Entsprechendes muss dann ebenfalls in List.cpp geändert werden. Falls es da Schwierigkeiten gibt, kann entweder der Stoff aus GIP wiederholt werden, oder Sie können Fragen auf Illias stellen.

Hinweis zu friend ... im ursprünglichen List.h-Teil der Klasse List: Hier muss folgendes hinzugefügt werden:

```
template <typename Tf>
friend std::ostream & operator << (std::ostream & stream, const List<Tf> & Liste);
```

Wählen Sie für die Template-Klasse List den Datentyp <int> im main-Programm und prüfen Sie die Lauffähigkeit Ihres Programms

- Der letzte Teil der Aufgabe besteht nun darin, die ursprüngliche Aufgabe zu realisieren. Kopieren Sie dazu wieder die Dateien List.h und Node.h vom 2. Projekt (Teilaufgabe 1-2) in ein neues 3. Projekt. Die Datei main.cpp und die Datei tweet.h erhalten Sie durch den download der Dateien vom ILIAS-Server (Praktikum>Praktikum 1>Praktikum 1 Vorlage>Vorlage_1_3[zip-Archiv]). Wenn alles korrekt implementiert wurde, sollte das Programm in der Datei man.cpp eine Twitter-Programm simulieren. Falls nicht, fixxen Sie die Fehler und versuchen das Programm zum laufen zu bringen. Wenn alles gut klappet sieht die Twitter-Anwendung beispielhaft wie folgt aus.



```
Microsoft Visual Studio-Debugging-Konsole
Nachricht: Test ADS Praktikum1
tweet
-->
10:24:6 -> Test
-->
Nachricht: Tweet
-->
10:24:6 -> ADS
10:24:6 -> Test
-->
Nachricht: Tweet
-->
10:24:6 -> Praktikum1
10:24:6 -> ADS
10:24:6 -> Test
-->
Nachricht: quit
```

Abbildung 11: Beispiel für ein laufendes Twitter-Programm