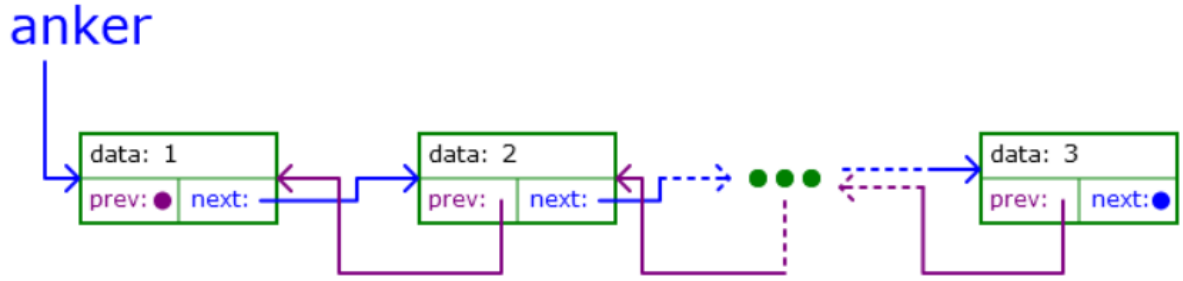
Praktikumstermin Nr. 09, INF: Dynamische Datenstruktur, Suchen in C-String

2019-12-04: Eine Diskrepanz zwischen vorgegebenem Hauptprogramm (letzte Version) und Testlauf behoben. Dadurch ändert sich eine Zeile in der letzten Version des Hauptprogramms. Änderung ist blau markiert.

Aufgabe INF-09.01: Dynamische Datenstruktur (Pointer, Speicherallokation auf dem Heap, Dynamische Datenstrukturen)

Nehmen Sie das Programm für die dynamische Datenstruktur „einfach verkettete" Liste (siehe unten) und erweitern Sie das Programm (sowohl die Datenstruktur als auch die Funktionen) so, dass die resultierende Datenstruktur eine *doppelt verkettete Liste* bildet.

Bei der *doppelt verketteten Liste* zeigt jedes Listenelement sowohl auf seinen Nachfolger (Pointer `next`) als auch auf seinen Vorgänger (Pointer `prev`, von „previous"). Der erste Listenknoten hat den Nullpointer als Wert von `prev`.



Als Verankerung der Datenstruktur soll weiterhin der Pointer `anker` auf den ersten Listenknoten verwendet werden, auch wenn dadurch die doppelte Verkettung nicht viel Nutzen bringt.

Nehmen Sie das folgende C++ Programm als Ausgangssituation für Ihre Erweiterungen.

```
#include <iostream>
using namespace std;

struct TListenKnoten
{
    int data;
    TListenKnoten *next;
};

void hinten_anfuegen(TListenKnoten *&anker, const int wert)
{
    TListenKnoten *neuer_eintrag = new TListenKnoten;
    neuer_eintrag->data = wert;
    neuer_eintrag->next = nullptr;
    if (anker == nullptr)
        anker = neuer_eintrag;
    else
    {
        TListenKnoten *ptr = anker;
        while (ptr->next != nullptr)
            ptr = ptr->next;
        ptr->next = neuer_eintrag;
    }
}

string liste_als_string(TListenKnoten * anker)
{
    string resultat = "";

    if (anker == nullptr)
        return "Leere Liste.";
    else
    {
        resultat += "[ ";
        TListenKnoten *ptr = anker;
        do
        {
            resultat += std::to_string(ptr->data);

            if (ptr->next != nullptr) resultat += " , ";
            else resultat += " ";

            ptr = ptr->next;
        } while (ptr != nullptr);
        resultat += "]";
    }

    return resultat;
}
```

```
void liste_ausgeben(TListenKnoten * anker)
{
    cout << liste_als_string(anker) << endl;
}

int main()
{
    const int laenge = 10;
    TListenKnoten *anker = nullptr;

    liste_ausgeben(anker);

    for (int i = 0; i < laenge; i++)
        hinten_anfuegen(anker, i*i);

    liste_ausgeben(anker);

    system("PAUSE");
    return 0;
}
```

Ändern Sie die Funktion `hinten_anfuegen()` so, dass auch der `prev` Pointer der Listenknoten jeweils korrekt gesetzt wird.

Fügen Sie ferner Ihrem Programm eine neue Funktion `void liste_ausgeben_rueckwaerts (TListenKnoten* anker)` hinzu, welche die Liste rückwärts ausgibt. Die Funktion soll sich ausgehend von `anker` erst bis zum Ende der Liste „durchhangeln" und dann die `prev` Verkettung in Rückrichtung bei der Ausgabe nutzen.

Erweitern Sie das Hauptprogramm wie folgt um den Aufruf dieser Funktion:

```
int main()
{
    int laenge = 10;
    TListenKnoten *anker = nullptr;
    liste_ausgeben(anker);
    liste_ausgeben_rueckwaerts(anker); // neu
    for (int i = 0; i < laenge; i++)
        hinten_anfuegen(anker, i*i);
    liste_ausgeben(anker);
    liste_ausgeben_rueckwaerts(anker); // neu
    system("PAUSE");
    return 0;
}
```

Diese Version des Hauptprogramms sowie den zugehörigen folgenden Testlauf brauchen Sie nicht im Praktikum vorzuzeigen, da die Funktion `liste_ausgeben_rueckwaerts()` *auch durch die später folgenden Testläufe für* `in_liste_einfuegen()` *geprüft wird.*

(Nicht vorzuzeigender) Testlauf *(keine Benutzereingaben)*:

```
Leere Liste.
Leere Liste.
[ 0 , 1 , 4 , 9 , 16 , 25 , 36 , 49 , 64 , 81 ]
[ 81 , 64 , 49 , 36 , 25 , 16 , 9 , 4 , 1 , 0 ]
Drücken Sie eine beliebige Taste . . .
```

Fügen Sie dann Ihrem Programm eine neue Funktion …

```
void in_liste_einfuegen(TListenKnoten* &anker,
                        int wert_neu,
                        int vor_wert)
```

… hinzu, welche einen neuen Wert `wert_neu` in die Liste einfügt, und zwar vor der Stelle des ersten Vorkommens des Wertes `vor_wert`.
Sollte der Wert `vor_wert` nicht in der Liste vorkommen, so soll `wert_neu` ans Ende der Liste angehängt werden.
Die Funktion `in_liste_einfuegen()` soll auch in der Lage sein, einen Wert in eine bisher leere Liste einzufügen.

Verwenden Sie jetzt folgendes weiter modifizierte Hauptprogramm:

```
int main()
{
    const int laenge = 10;
    TListenKnoten *anker = nullptr;

    for (int i = 0; i < laenge; i++)          // neu
        in_liste_einfuegen(anker, i*i, 9999); // neu

    liste_ausgeben(anker);
    liste_ausgeben_rueckwaerts(anker);

    int wert_neu = 0, vor_wert = 0;                   // neu
    cout << "Einzufuegender Wert: "; cin >> wert_neu; //neu
    cout << "Vor welchem Wert? "; cin >> vor_wert;    // neu
    in_liste_einfuegen(anker, wert_neu, vor_wert);    // neu

    liste_ausgeben(anker);
    liste_ausgeben_rueckwaerts(anker);

    system("PAUSE");
    return 0;
}
```

Testläufe (Benutzereingaben sind unterstrichen):

```
[ 0 , 1 , 4 , 9 , 16 , 25 , 36 , 49 , 64 , 81 ]
[ 81 , 64 , 49 , 36 , 25 , 16 , 9 , 4 , 1 , 0 ]
Einzufuegender Wert: 99
Vor welchem Wert? 0
[ 99 , 0 , 1 , 4 , 9 , 16 , 25 , 36 , 49 , 64 , 81 ]
[ 81 , 64 , 49 , 36 , 25 , 16 , 9 , 4 , 1 , 0 , 99 ]
Drücken Sie eine beliebige Taste . . .
```

```
[ 0 , 1 , 4 , 9 , 16 , 25 , 36 , 49 , 64 , 81 ]
[ 81 , 64 , 49 , 36 , 25 , 16 , 9 , 4 , 1 , 0 ]
Einzufuegender Wert: 99
Vor welchem Wert? 81
[ 0 , 1 , 4 , 9 , 16 , 25 , 36 , 49 , 64 , 99 , 81 ]
[ 81 , 99 , 64 , 49 , 36 , 25 , 16 , 9 , 4 , 1 , 0 ]
Drücken Sie eine beliebige Taste . . .
```

```
[ 0 , 1 , 4 , 9 , 16 , 25 , 36 , 49 , 64 , 81 ]
[ 81 , 64 , 49 , 36 , 25 , 16 , 9 , 4 , 1 , 0 ]
Einzufuegender Wert: 99
Vor welchem Wert? 99
[ 0 , 1 , 4 , 9 , 16 , 25 , 36 , 49 , 64 , 81 , 99 ]
[ 99 , 81 , 64 , 49 , 36 , 25 , 16 , 9 , 4 , 1 , 0 ]
Drücken Sie eine beliebige Taste . . .
```

```
[ 0 , 1 , 4 , 9 , 16 , 25 , 36 , 49 , 64 , 81 ]
[ 81 , 64 , 49 , 36 , 25 , 16 , 9 , 4 , 1 , 0 ]
Einzufuegender Wert: 99
Vor welchem Wert? 25
[ 0 , 1 , 4 , 9 , 16 , 99 , 25 , 36 , 49 , 64 , 81 ]
[ 81 , 64 , 49 , 36 , 25 , 99 , 16 , 9 , 4 , 1 , 0 ]
Drücken Sie eine beliebige Taste . . .
```

Fügen Sie dann Ihrem Programm eine neue Funktion …
`void aus_liste_loeschen(TListenKnoten* &anker, int wert)`
… hinzu, welche in der Liste den *ersten* Knoten mit Wert `wert` löscht und den `anker`, falls die Liste dadurch leer wird, auf den `nullptr` Wert zurücksetzt. Sollten mehrere Knoten mit dem Wert `wert` in der Liste vorkommen, so werde *nur der Erste* dieser Knoten gelöscht.
Die Funktion `aus_liste_loeschen()` soll auch in der Lage sein, für eine leere Liste aufgerufen zu werden, als auch für eine Liste, in welcher der Wert `wert` gar nicht vorkommt. In diesen Fällen soll letztendlich nichts passieren, da kein Knoten zu löschen ist. Beachten Sie auch, was passieren muss, wenn der Wert `wert` im ersten Knoten der Liste vorkommt.

Fügen Sie dann Ihrem Programm eine neue Funktion …
`void liste_loeschen(TListenKnoten* &anker)`
… hinzu, welche alle Knoten der Liste löscht und den `anker` auf den `nullptr` Wert zurücksetzt. Beachten Sie, dass diese Funktion auch wirklich den Speicher aller Listenknoten auf dem Heap mittels `delete` freigeben soll.
Die Funktion `liste_loeschen()` soll auch in der Lage sein, für eine leere Liste aufgerufen zu werden. Dann soll de-facto nicht passieren, da keine Knoten zu löschen sind.

Verwenden Sie jetzt folgendes weiter modifizierte Hauptprogramm:

```
int main()
{
    const int laenge = 10;
    TListenKnoten *anker = nullptr;

    liste_ausgeben(anker);
    liste_ausgeben_rueckwaerts(anker);
    liste_loeschen(anker);

    hinten_anfuegen(anker, 77);
    hinten_anfuegen(anker, 88);
    hinten_anfuegen(anker, 99);
    liste_ausgeben(anker);
    liste_ausgeben_rueckwaerts(anker);

    liste_loeschen(anker); // war: aus_liste_loeschen(anker, 99);
    liste_ausgeben(anker);
    liste_ausgeben_rueckwaerts(anker);

    for (int i = 0; i < laenge; i++)
        in_liste_einfuegen(anker, i*i, 9999);
```

```
    liste_ausgeben(anker);
    liste_ausgeben_rueckwaerts(anker);

    in_liste_einfuegen(anker, -1, 0);
    in_liste_einfuegen(anker, 24, 25);
    in_liste_einfuegen(anker, 80, 81);
    in_liste_einfuegen(anker, 99, 9999);
    liste_ausgeben(anker);
    liste_ausgeben_rueckwaerts(anker);

    aus_liste_loeschen(anker, 24);
    aus_liste_loeschen(anker, 80);
    liste_ausgeben(anker);
    liste_ausgeben_rueckwaerts(anker);

    liste_loeschen(anker);
    liste_ausgeben(anker);
    liste_ausgeben_rueckwaerts(anker);

    system("PAUSE");
    return 0;
}
```

Testlauf *(keine Benutzereingaben mehr)*:

```
Leere Liste.
Leere Liste.
[ 77 , 88 , 99 ]
[ 99 , 88 , 77 ]
Leere Liste.
Leere Liste.
[ 0 , 1 , 4 , 9 , 16 , 25 , 36 , 49 , 64 , 81 ]
[ 81 , 64 , 49 , 36 , 25 , 16 , 9 , 4 , 1 , 0 ]
[ -1 , 0 , 1 , 4 , 9 , 16 , 24 , 25 , 36 , 49 , 64 , 80 , 81 , 99 ]
[ 99 , 81 , 80 , 64 , 49 , 36 , 25 , 24 , 16 , 9 , 4 , 1 , 0 , -1 ]
[ -1 , 0 , 1 , 4 , 9 , 16 , 25 , 36 , 49 , 64 , 81 , 99 ]
[ 99 , 81 , 64 , 49 , 36 , 25 , 16 , 9 , 4 , 1 , 0 , -1 ]
Leere Liste.
Leere Liste.
Drücken Sie eine beliebige Taste . . .
```

Nur dieser letzte Testlauf muss im Praktikum vorgezeigt werden, da er alle wesentlichen vorherigen Operationen beinhaltet.