Listen

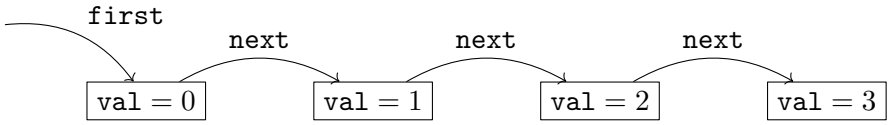
Nicholas Schwab

5. Dezember 2017

1. Einfach verkettete Listen

1.1. Theorie

Eine einfach verkettete Liste ist eine Datenstruktur, die aus Elementen, die immer einen Zeiger (`next`) auf das nächste Element haben, besteht. Dabei enthält jedes Element einen Wert (`val`)(hier wird das ein `int` sein, aber man kann dafür jeden Datentyp oder auch ein eigenes `struct` nehmen).



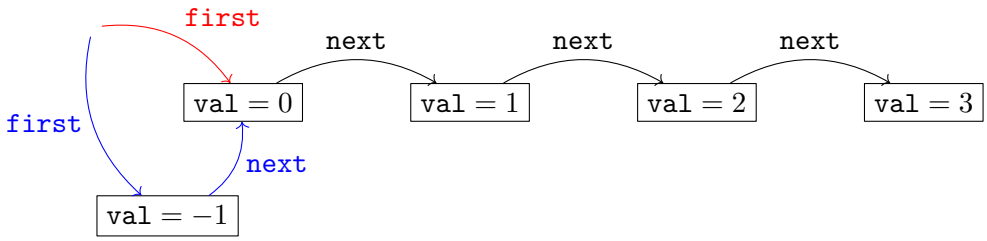
Natürlich hat hier das letzte Element keinen Nachfolger, deswegen wäre `next` hier ein Nullpointer. Die Liste an sich ist keine Variable, man speichert einen Pointer `first` auf das erste Element, da man dadurch, durch wiederholtest Folgen des Pointers `next`, zu allen anderen Elemente kommt. Dies ähnelt dem Array, bei dem man einen Pointer auf das erste Elemente speichert und dann durch Rechnen mit diesem Pointer auf die anderen Elemente zugreifen kann.

Nun kann man mit Listen verschiedene Operationen sehr gut und schnell durchführen.

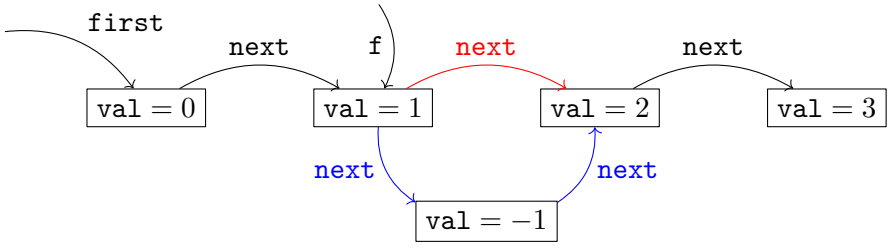
1. Ein Element vorne anfügen
2. Ein Element hinter einem gegebenen Element einfügen
3. Ein Element vorne löschen
4. Den Nachfolger eines Elements löschen
5. Über die komplette List iterieren (d.h. von vorne nach hinten alle Elemente durchgehen)

Die ersten beiden Punkte sind bei einem Array nur schwer möglich, da dazu alle späteren Elemente verschoben werden müssten. Bevor wir uns der Implementation widmen, schauen wir erst einmal, was bei den jeweiligen Operationen zu tun ist.

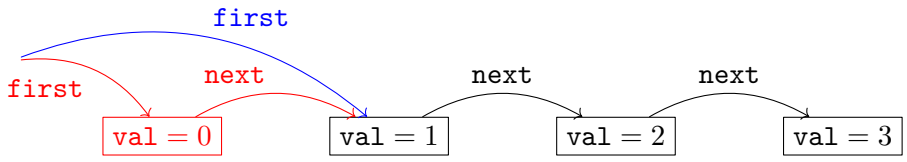
Vorne Anfügen Angenommen, ich will eine Element `e` (hier mit `val` $= -1$) vorne Anfügen. Danach soll der Nachfolger von `e` das bisherige erste Element sein und `e` das neue erste Element sein. Folglich setzen wir das `next` von e auf `first` und lassen danach `first` auf `e` zeigen. Hier sehen wir die neuen Pointer in blau und den alten in rot:



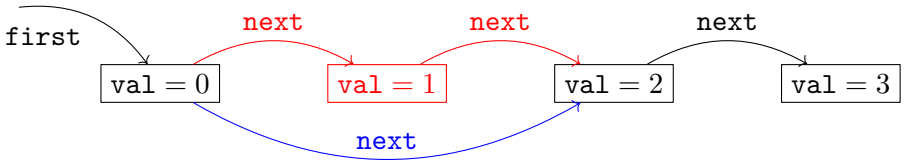
Hinter einem Element einfügen Wir wollen nun das Element `e` hinter dem Element mit `val` $= 1$ einfügen. Das geht nur, wenn wir aus einer anderen Quelle (beispielsweise gehen wir gerade die Liste durch) einen Pointer `f` auf dieses Element haben, da wir die Elemente nicht durch ihren Wert aufrufen können. Also muss nach dem Einfügen der Nachfolger des Elements, auf das `f` zeigt, der Nachfolger von `e` sein und `e` wiederum der neue Nachfolger des Elements auf das `f` zeigt. Wir setzen also das `next` von `e` auf das `next` von `f` (bzw. von `*f`, wenn man korrekt sein will, `f` ist ja ein Pointer, kein Element). Danach lassen wir das `next` von `f` auf `e` zeigen:



Vorne löschen Um das erste Element zu löschen, setzt man einfach `first` auf den Nachfolger des ersten Elements. Man sollte auch das `next` des entfernten Elements auf den Nullpointer setzen, damit man nicht später dieses Element irgendwo einfügt und aus Versehen einen Pointer auf ein falsches Element hat.

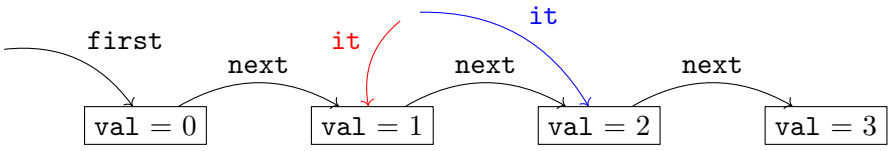


Nachfolger löschen Um den Nachfolger eines Elements zu löschen, setzt man das `next` dieses Elements auf das `next` des Nachfolgers. Wieder will man auch das `next` des gelöschten Elements auf den Nullpointer setzen. Man beachte, das hier Löschen beidesmal das Löschen aus der Liste bezeichnet. Das gelöschte Element besteht noch im Speicher.



Man kann so nur einen Nachfolger und nicht etwa ein gegebenes Element selbst löschen, da man immer Zugriff auf den Vorgänger des zu löschenden Elements braucht.

Iterieren über die Liste Für das Iterieren brauchen wir wie beim Iterieren über ein Array eine Laufvariable, die anzeigt, wo wir uns gerade befinden. Diese ist beim Array ein `int` mit dem aktuellen Index. Hier jedoch nehmen wir einen Pointer `it` auf das aktuelle Element. Diesen setzen wir zu Beginn auf `first` (so wie man den Index auf 0 setzt). Um nun ein Element weiter zu gehen (den Index um 1 zu erhöhen) will man `it` auf das nächste Element zeigen lassen. Also überschreibt man `it` mit dem `next` des Elements, auf das `it` gerade zeigt. Dies machen wir solange, wie es geht, also bis der nächste Pointer ein Nullpointer ist.



1.2. Implementation

Um das ganze in C++ zu implementieren nutzen wir `struct`. Eine `Element` muss ein `int val` und einen Pointer `next` auf das nächste `Element` enthalten.

```
struct Element{
    int val;
    Element *next = nullptr;
};
```

Wir nutzen hier ein Feature von C++, nämlich, dass wir Variablen in einem `struct` Defaultwerte zuweisen kann, hier setzen wir automatisch `next` erstmal auf einen Nullpointer. Wir arbeiten immer mit Pointern auf Elementen, damit wir diese nicht kopieren müssen. Dies könnte sehr langsam werden, wenn der Wert ein großes `struct` statt einem `int` ist.

Angenommen ich würde nun ein Element erstellen wollen, geht das in C++ über das `new`-Keyword, das quasi `malloc` entspricht, da es Speicher allokalisiert und einen Pointer darauf zurückgibt.

```
// Erzeuge ein Element mit Wert 0
Element *e = new Element;
e->val = 0;
```

Dabei ist `e->val` eine Abkürzung für `(*e).val`, da man `e` erst dereferenzieren muss, bevor man auf `val` zugreifen kann.

Der Pointer `first`, der auf das erste Element meiner Liste zeigt, ist im Falle einer leeren Liste einfach der Nullpointer.

Vorne Anfügen Wir wollen nun eine Methode schreiben, die ein `Element` (gegeben durch einen Pointer darauf) vorne an eine Liste (gegeben durch einen Pointer `first` auf das erste Element) anfügt.

```
Element *vorne(Element *e, Element *first){
    e->next = first;
    first = e;
    return first;
}
```

Hier müssen wir das neue `first` zurückgeben, da es sich geändert hat. Ein Aufruf der Methode muss dementsprechend so

```
first = vorne(e, first);
```

aussehen.

Hinter einem Element einfügen Wir wollen nun ein `Element` (gegeben durch einen Pointer) hinter einem `Element` (wieder gegeben durch einen Pointer) der Liste einfügen.

```
void einfuegen(Element *e, Element *f){
    e->next = f->next;
    f->next = e;
}
```

Würde man nun beispielsweise ein `Element` an der zweiten Stelle einfügen wollen, könnte man das so machen:

```
einfuegen(e, first);
```

Man muss hier aufpassen, dass `f` kein Nullpointer ist, sonst gibt es Fehler.

Vorne löschen Wir wollen nun eine Methode schreiben, die das erste Element einer Liste aus der Liste (gegeben durch einen Pointer `first` auf das erste `Element`) löscht. Diese muss wieder einen Pointer auf das neue erste `Element` zurückgeben.

```
Element *deleteFirst(Element *first){
    Element *tmp = first;
    first = first->next;
    tmp->next = nullptr;
    //delete tmp;
    return first;
}
```

Die Zeile mit dem `delete` würde (wäre sie nicht auskommentiert) den Speicher, der bisher von `tmp` (also dem ehemaligen ersten Element) benutzt wurde, freigeben. Man sollte das nicht in der allgemeinen Löschmethode machen, da man häufig nur das Element aus der Liste löschen will, aber die Daten selbst noch behalten will und vielleicht in eine andere Liste einfügen will oder so. Wenn man den Speicher zusätzlich freigeben will, sollte man das daher eher so machen:

```
Element *tmp = first;
first = deleteFirst(first);
delete tmp;
```

Nachfolger löschen Die folgende Methode löscht den Nachfolger eines Elements (gegeben durch einen Pointer `e` darauf) aus der Liste und gibt einen Pointer auf das gelöschte `Element` zurück (damit man danach damit weiterarbeiten, den Speicher freigeben kann, etc.)

```
Element *deleteAfter(Element *e){
    Element *tmp = e->next;
    e->next = tmp->next;
    tmp->next = nullptr;
    return tmp;
}
```

Iterieren über die Liste Dies lässt sich mit einer `for`-Schleife realisieren. Erinnert euch daran, dass im Kopf einer `for`-Schleife drei Ausdrücke stehen. Der erste wird vor dem ersten Durchlauf aufgerufen und dient zum initialisieren der Laufvariable. Der zweite wird vor jedem Durchlauf geprüft und bricht die Schleife ab, wenn der Ausdruck zu `false` evaluiert, überprüft also, ob die Laufvariable noch im richtigen Bereich ist. Der dritte Ausdruck wird nach jedem Durchlauf aufgerufen und soll die Laufvariable auf den nächsten Wert setzten.

```
for(Element *it = first; it != nullptr; it = it->next){
    // Tue etwas mit dem aktuellen Element it
    // z.B. ausgeben:
    std::cout << it->val << std::endl;
}
```

1.3. Aufgaben

Aufgabe 1.1. Überlegt euch die Laufzeiten jeder der obrigen Operationen.

Aufgabe 1.2. Überlegt euch, wie man das Einfügen eines Elements ganz hinten realisieren kann.

Tipp 1: Es geht besser, als jedesmal bis zum letzten Element durchzulaufen.

Tipp 2: Man sollte dafür etwas mehr speichern.

Aufgabe 1.3. Implementiert Insertion-Sort mit einer Eingabe- und einer Ausgabeliste.

Tipp: Bei Insertion-Sort müsst ihr immer wieder einfügen, aber das könnt ihr nur hinter einem Element. Überlegt euch, wie man erreicht, dass man nicht vor einem Element einfügen muss.

Aufgabe 1.4. Implementiert Selection-Sort mit einer Eingabe- und einer Ausgabeliste.

Tipp: Ihr müsst in der Eingabeliste löschen, also sollte man das Minimum der aktuellen Eingabe nicht direkt speichern, sonst kann man es nicht löschen.

Aufgabe 1.5. Implementiert Bubble-Sort auf einer Liste.

Tipp: Ihr wollt nicht die `Element` vertauschen.

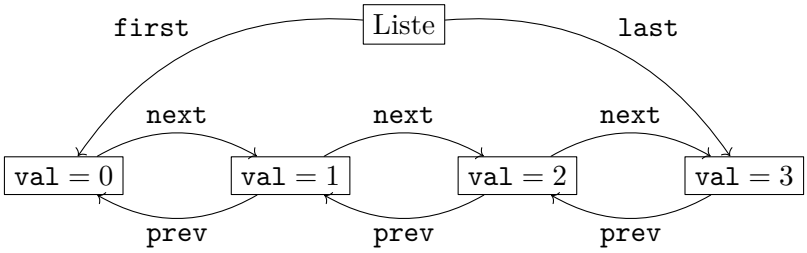
2. Doppelt verkettete Listen

2.1. Theorie

Wir hatten jetzt mehrfach folgende Probleme:

1. Wir konnten nicht auf vorherige Element zugreifen.
2. Wir würden gerne auch auf das letzte Elemente zugreifen.
3. Es ist irgendwie unelegant, Pointer zurückgeben zu müssen, wenn man `first` verändert.

Diese Probleme lösen wir nun durch doppelte verkettet Listen (auch bekannt als double-ended queue oder deque). Bei dieser speichern wir nicht nur einen Pointer `next` auf den Nachfolger, sonder auch einen Pointer `prev` auf den Vorgänger. Wir speichern des Weiteren nicht nur den Pointer `first` auf das erste Element sondern auch einen Pointer `last` auf das letzte Element. Schließlich fassen wir die beiden Pointer in einem `struct` Liste zusammen, das einfach nur aus den beiden Pointern besteht. Es ist `next` beim letzten Element ein Nullpointer genauso wie `prev` beim ersten.

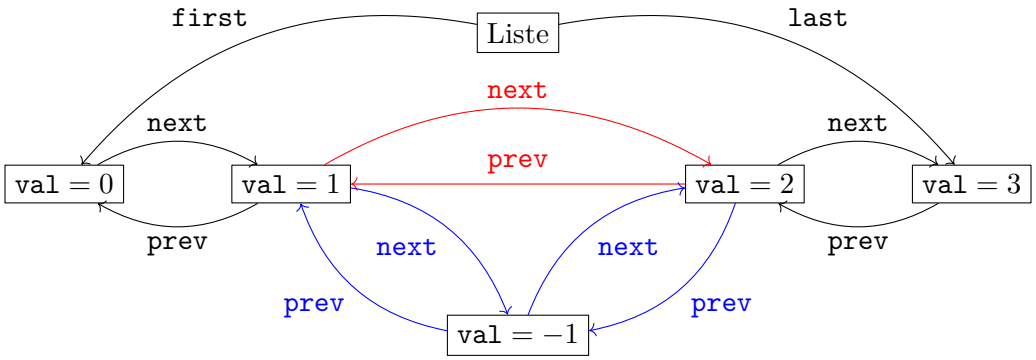


Diese Liste kann alles, was auch die einfach verkettete Liste kann, aber macht es ein bisschen anders, man muss jedesmal mehr Pointer aktualisieren. Zusätzlich kann sie jedoch:

1. Ein Element vor einem anderen einfügen
2. Ein gegebenes Element löschen
3. Das letzte Element löschen
4. Von hinten über die Liste iterieren

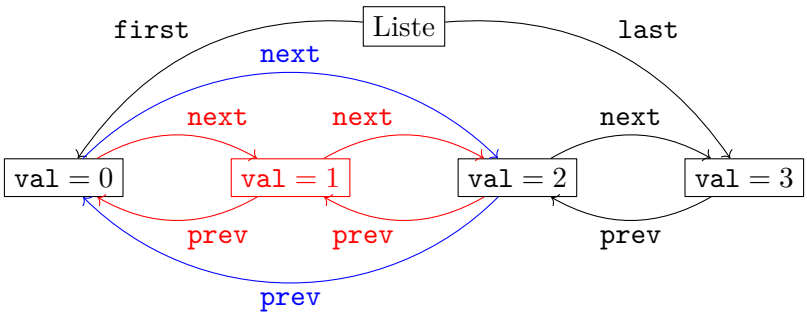
Offensichtlich ist 3. ein Spezialfall von 2. und ebenso wurde das Löschen des ersten Elements ein Spezialfall von 2. Um von hinten über die Liste iterieren, muss man bei `last` anfangen und den `prev` folgen.

Element einfügen Wir wollen das Einfügen eines Elements allgemein besprechen, da man nun sehr leicht zwischen Vorgänger und Nachfolger springen kann. Daher entspricht das Einfügen vor einem Element dem Einfügen nach dem Vorgänger und so weiter. Wir betrachten also den Fall, dass ich das Element `e` mit `val` $= -1$ zwischen den Element mit den Werten 1 und 2 einfügen möchte. Dann muss ich `next` des vorderen Element und `prev` des hinteren Elements auf `e` zeigen lassen. Die beiden Pointer `next` und `prev` von `e` müssen jeweils auf das vordere bzw. das hintere Element zeigen.



Wie genau das Einfügen vorne und hinten funktioniert, darf sich der geneigte Leser in einer ruhigen Minute selbst überlegen.

Element löschen Wenn wir nun ein Element löschen wollen, müssen wir die Pointer, die auf es zeigten entsprechend umleiten. Wenn also das zu löschende Element sich zwischen zwei anderen befindet muss man den `next`-Pointer des Vorgängers auf den Nachfolger zeigen lassen und den `prev`-Pointer des Nachfolgers auf auf den Vorgänger. Beim Löschen des ersten und des letzte Elements muss man natürlich wieder auf `first` und `last` aufpassen.



2.2. Implementation

Wir arbeiten wieder mit `struct`. Wir müssen unser `Element` ein bisschen verändern und ein `struct Liste` erstellen:

```
struct Element{
  int wert;
  Element *next = nullptr;
  Element *prev = nullptr;
};

struct Liste{
  Element *first = nullptr;
  Element *last = nullptr;
};
```

Zu beginn müssen wir eine neu `Liste` erstellen, die wir `list` nennen. Das geht sowohl direkt als auch mit `new`, wobei wir einen Pointer zurückbekommen.

```
Liste list;
// Alternativ
Liste *list = new Liste;
```

Wir gehen in Zukunft davon aus, dass `list` den Typ `*Liste` hat, da wir auch stets diesen übergeben werden. Zu Beginn sind `first` und `last` Nullpointer. Das ist immer genau dann der Fall, wenn die Liste leer ist, also sollten Abfragen wie `list->first == nullptr` immer als die Abfrage, ob `list` leer ist gesehen werden.

Einfügen Beim Einfügen betrachten wir zuerst die beiden Spezialfälle des vorne und hinten Einfügen, die beide Einfacher sind. Wir übergeben an die Methode jeweils einen Pointer auf das einzufügende `Element` und einen auf die `Liste`. Dadurch werden Veränderungen an der `Liste` immer auch auf der übergebenen durchgeführt. Hier ist `append` das Anhängen, also einfügen hinten und `push` das einfügen vorne.

```
void append(Element *e, Liste *l){
  e->next = nullptr;
  e->prev = nullptr;
```

```
  if(l->last == nullptr){
    l->first = e;
    l->last = e;
  }else{
    e->prev = l->last;
    l->last->next = e;
    l->last = e;
  }
}

void push(Element *e, Liste *l){
  e->next = nullptr;
  e->prev = nullptr;
  if(l->first == nullptr){
    l->first = e;
    l->last = e;
  }else{
    e->next = l->first;
    l->first->prev = e;
    l->first = e;
  }
}
```

Nun nutzen wir diese Spezialfälle um das allgemeine Einfügen von `e` hinter `hinter` bzw. vor `vor` zu implementieren:

```
void insertAfter(Element *e, Element *hinter, Liste *l){
  e->prev = nullptr;
  e->next = nullptr;
  if(hinter->next == nullptr){
    append(e, l);
  }else{
    e->next = hinter->next;
    e->prev = hinter;
    hinter->next->prev = e;
    hinter->next = e;
  }
}

void insertBefore(Element *e, Element *vor, Liste *l){
  e->next = nullptr;
  e->prev = nullptr;
  if(vor->prev == nullptr){
    push(e, l);
  }else{
    e->next = vor;
    e->prev = vor->prev;
```

```
    vor->prev->next = e;
    vor->prev = e;
  }
}
```

Löschen Um eine Element aus einer Liste zu löschen, muss man die Fälle unterscheiden, in denen es vorne bzw. hinten steht. Aus Implementationsgründen wird auch der Fall unterschieden, in denen es das einzige Element der Liste ist.

```
Element *pop_front(Liste *l){
  Element *e = l->first;
  e->next->prev = nullptr;
  l->first = e->next;
  e->next = nullptr;
  return e;
}

Element *pop_back(Liste *l){
  Element *e = l->last;
  e->prev->next = nullptr;
  l->last = e->prev;
  e->prev = nullptr;
  return e;
}

void deleteElem(Element *e, Liste *l){
  if(e->prev == nullptr && e->next == nullptr){
    // e ist das einzige Element der Liste
    l->first = nullptr;
    l->last = nullptr;
    return;
  }
  if(e->prev == nullptr){
    pop_front(l);
    return;
  }
  if(e->next == nullptr){
    pop_back(l);
    return;
  }

  e->prev->next = e->next;
  e->next->prev = e->prev;
  e->prev = nullptr;
  e->next = nullptr;
}
```

Wir lassen bei `pop_front` und `pop_back` (also Löschen vorne und hinten) jeweils das gerade gelöschte Element zurückgeben, weil das häufig gebraucht wird.

2.3. Aufgaben

Aufgabe 2.1. Mache jede Aufgabe aus dem 1. Abschnitt mit einer Deque.

Aufgabe 2.2. Gegeben sei eine `Liste` mit dem Namen `list`. Schreibe ein Programm in einer Zeile, das `list` komplett leert.

Tipp: `for`-Schleife