ILV Datenstrukturen und Algorithmen

06: Verkettete Listen, Verschachtelte Klassen

Stefan Huber

FH Salzburg, Studiengang MMT / 2012

Kapitel

Verkettete Listen

Motivation

Organisation von Daten bis jetzt: Array, etwa `int* a = new int[n];`

Nachteile:

- Das Array `a` kann nicht vergrößert werden. (Daher: neues Array allozieren und kopieren.)
- Löschen eines Elements `a[i]` ist aufwändig: Verschieben von $n - 1 - i$ Elementen.
- Einfügen eines Elements an die i-te Position ist aufwändig: Verschieben von $n - i$ Elementen und eventuell zuvor Array vergrößern.

Vorteile:

- Zugriff auf das i-te Element sehr schnell: `a[i]` kann in einen (oder wenige) Maschinenbefehle umgesetzt werden.
- Arrays sind speichereffizient. Für n Elemente vom Typ T braucht man `n * sizeof(T)` Bytes.

Fazit: Arrays sind unflexibel.

Motivation

Verkettete Listen:

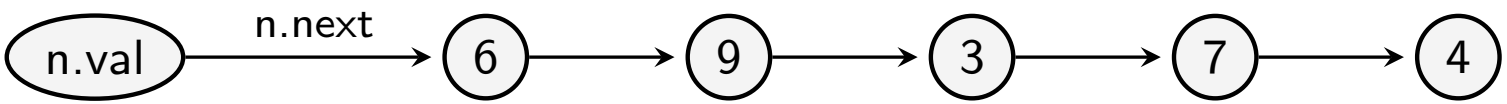
- Effizientes Löschen und Einfügen.
- Die Größe muss nicht im Vorhinein bekannt sein.
- "There ain't no such thing as a free lunch":
  - Zugriff auf das i-te Element ist im Allgemeinen langsam. (Auf die Elemente am Anfang und/oder am Ende der Liste kann man ggf. schneller zugreifen.)
  - Wir brauchen zusätzlich Speicher für die Organisation der Daten.

Einfach verkettete Liste: Knoten

- Daten werden in Knoten gespeichert.
- Jeder Knoten enthält
  - ein Datum und
  - einen Verweis (Pointer) zum nächsten Knoten.

```
1 struct Node
2 {
3   int val;        // Das Datum dieses Knoten
4   Node* next;     // Verweis zum naechsten Knoten
5 };
```

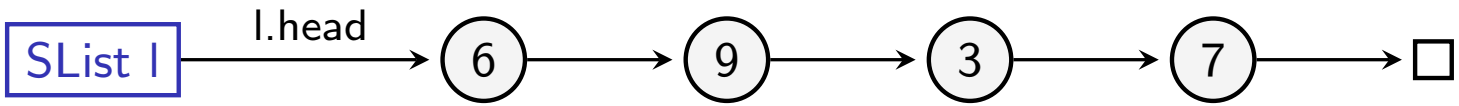
Node n



Einfach verkettete Liste: Kopf

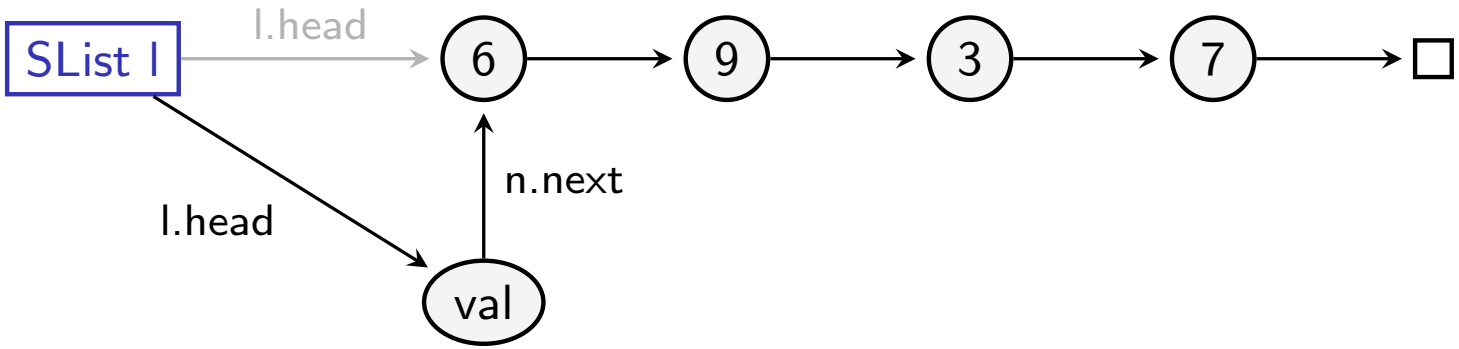
- Eine verkettete Liste `SList l` hat einen Kopf-Knoten.
  - `l.head` ist `0`, falls die Liste leer ist.
- Jener Knoten `n`, der keinen nachfolgenden Knoten hat, hat `n.next = 0`.

```
1 class SList
2 {
3   Node* head;
4 };
```



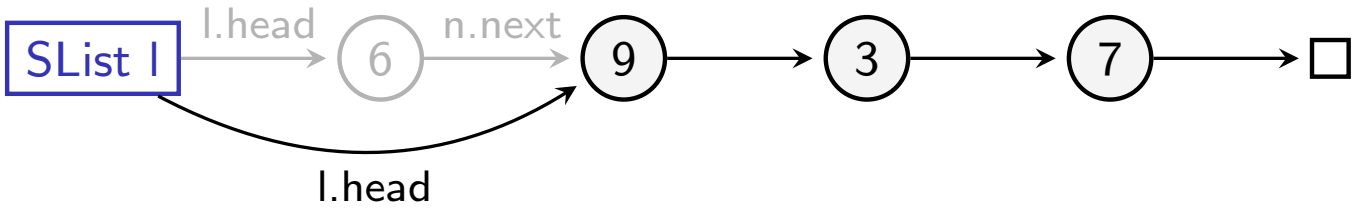
Einfach verkettete Liste: Einfügen am Kopf

- Ein Knoten mit Wert `val` soll an den Beginn der Liste eingefügt werden.




```
1  void SList::push_front(int val)
2  {
3    //Works even if list is empty!
4    Node* n = new Node;
5    n->val = val;
6    n->next = head;
7    head = n;
8  }
```

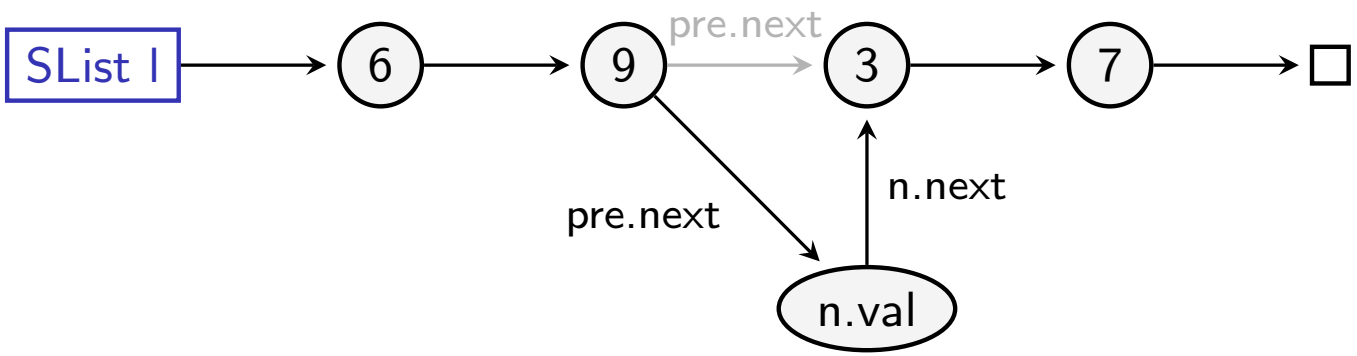
Einfach verkettete Liste: Entfernen am Kopf




```
1  void SList::pop_front()
2  {
3    //Works even if list becomes empty
4    Node *n = head;
5    assert( n != 0 );
6    head = n->next;
7    delete n;
8  }
```

Einfach verkettete Liste: Einfügen nach Knoten

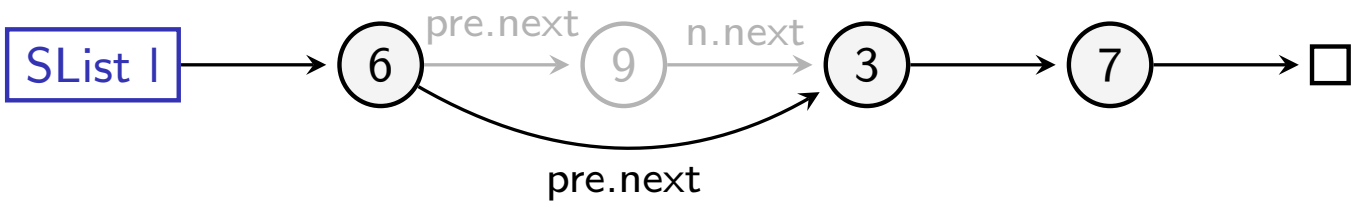
- Ein Knoten mit Wert `val` soll nach einem bestehenden Knoten `pre` eingefügt werden.




```
1 void SList::insert(Node& pre, int val)
2 {
3   //Works even if pre is the tail node!
4   Node* n = new Node;
5   n->val = val;
6   n->next = pre.next;
7   pre.next = n;
8 }
```

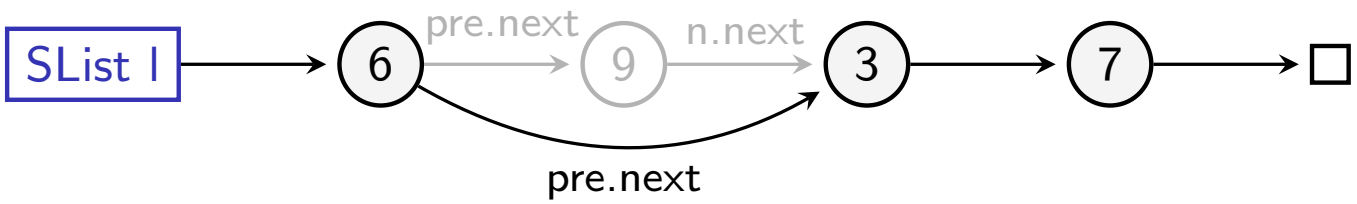
Einfach verkettete Liste: Entfernen eines Nachfolgers

- Der Knoten `n` nach dem Knoten `pre` soll entfernt werden.




```
1 void SList::removeAfter(Node& pre)
2 {
3   //Works even if pre's successor is the tail node
4   Node* n = pre.next;
5   assert(n != 0 );
6   pre.next = n->next;
7   delete n;
8 }
```

Einfach verkettete Liste: Entfernen eines Knoten




```
1  void SList::remove(Node& n)
2  {
3    if( head == &n)
4    {
5      pop_front();
6      return;
7    }
8
9    Node* pre = head;
10   while( pre->next != &n)
11   {
12     pre = pre->next;
13     assert(pre!=0);
14   }
15   assert(pre->next != 0);
16   removeAfter(pre);
17 }
```

Einfach verkettete Liste: Zählen

- Überprüfen, ob die Liste leer ist, geht schnell.
- Zählen der Knoten ist langsam.

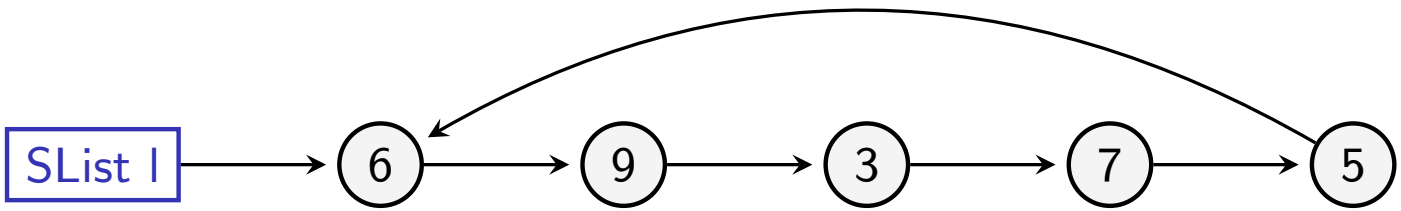
```
1 bool List::isEmpty() const
2 {
3   return head==0;
4 }
```

```
1 unsigned List::getSize() const
2 {
3   unsigned cnt=0;
4   for( Node* n=head; n!=0; n=n->next)
5   {
6     cnt++;
7   }
8   return cnt;
9 }
```

Fazit: Bei manchen Datenstrukturen ist `getSize()==0` viel langsamer als `isEmpty()`!

Einfach verkettete Liste: Ergänzungen

- Wenn man zusätzlich einen Pointer auf das letzte Element unterhält, etwa `Node* SList::tail`, dann ist auch das Einfügen am Ende (`SList::push_back`) effizient.
- Manchmal werden für den Kopf (und das Ende) sogenannte dummy Knoten verwendet.
  - Das sind Knoten, die den Kopf (bzw. das Ende) repräsentieren und deren Wert (`Node::val`) ignoriert wird.
  - Der erste „echte“ Knoten ist nach nach dem Kopf-Knoten.
  - Damit erspart man sich die Spezialbehandlung des Pointers `SList::head`, etwa in `SList::remove`.
- Anstatt den letzten Knoten auf `0` verweisen zu lassen, kann man auch auf den Kopf verweisen. Man erhält dann eine sogenannte „zirkuläre Liste“.



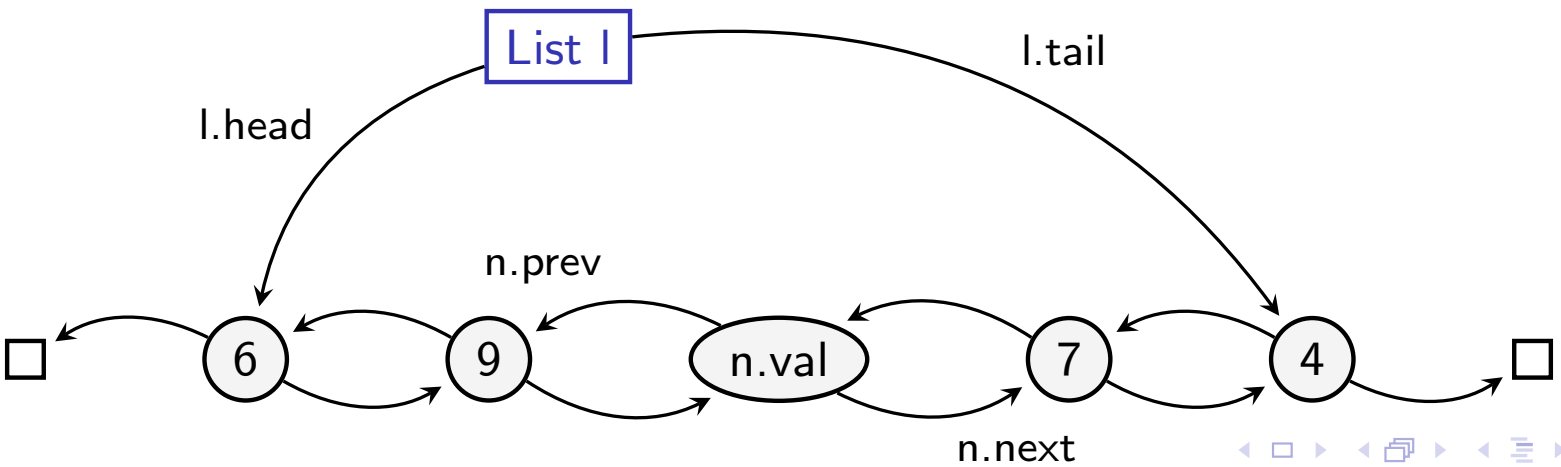
Zweifach verkettete Liste

- Einfach verkettete Liste: Durchlaufen der Liste ist nur in eine Richtung effizient.
- Bei `SList::remove` müssen wir aufwändig das Element vor dem zu löschenden Element bestimmen.
- Idee: Zweifach verkettete Liste:
  - Jeder Knoten ist mit seinem Vorgänger und Nachfolger verbunden.
  - Die Liste unterhält einen Pointer auf den Kopf und das Ende.

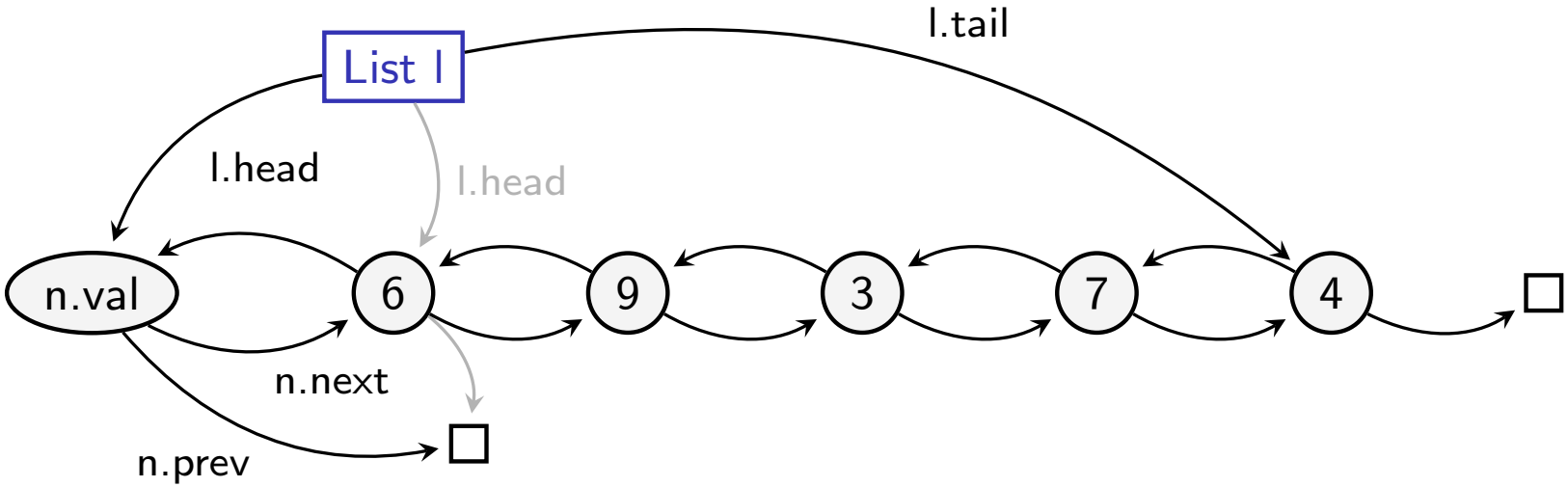
Zweifach verkettete Liste

```
1 struct Node
2 {
3   Node* prev;
4   Node* next;
5   int val;
6 };
```

```
1 class List
2 {
3   Node *head;
4   Node *tail;
5 };
```

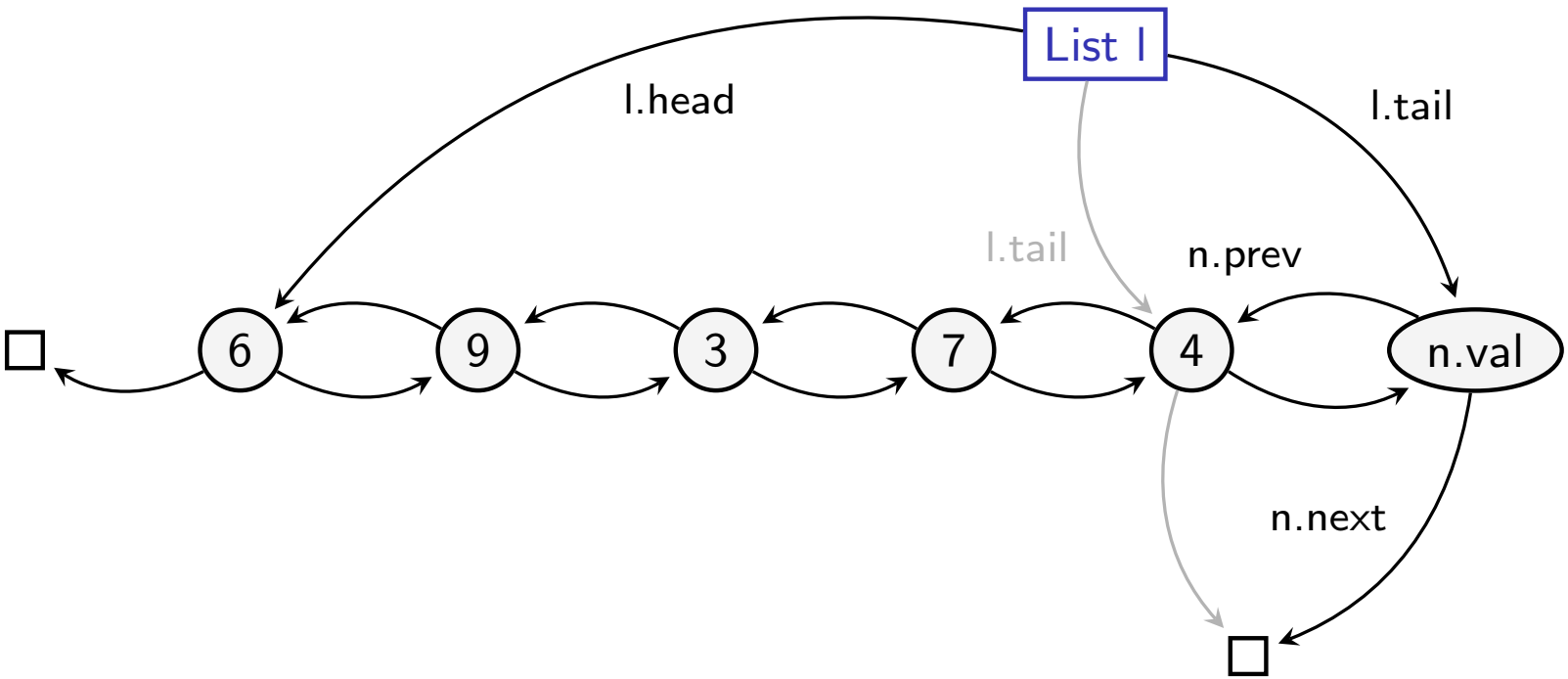


Zweifach verkettete Liste: Einfügen am Kopf

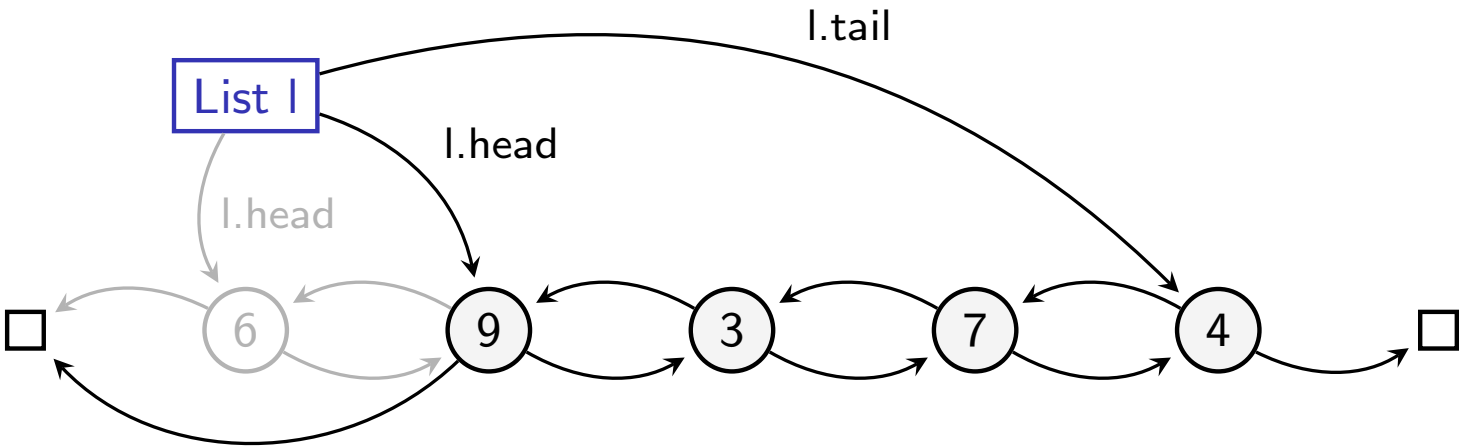



```
1  void List::push_front(int val)
2  {
3    Node* n = new Node;
4    n->val = val;
5    n->next = head;
6    n->prev = 0;
7    if( head != 0 )
8      head->prev = n;
9    head = n;
10   if( tail == 0 )
11     tail = head;
12 }
```

Zweifach verkettete Liste: Einfügen am Ende



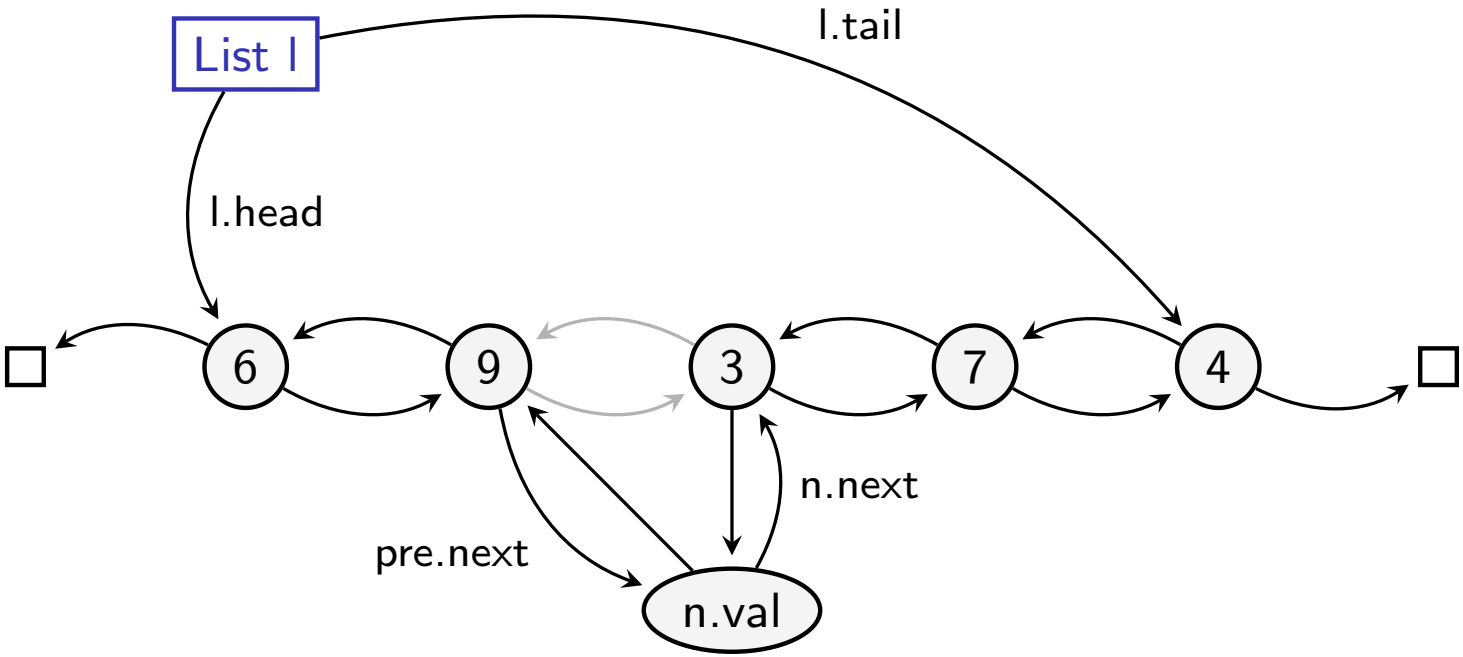
Zweifach verkettete Liste: Entfernen des Kopfs/Endes




```
1  void List::pop_front()
2  {
3    assert( head != 0 );
4    if( head == tail ) {
5      delete head;
6      head = tail = 0;
7      return;
8    }
9    Node* n = head;
10   head = head->next;
11   head->prev = 0;
12   delete n;
13 }
```

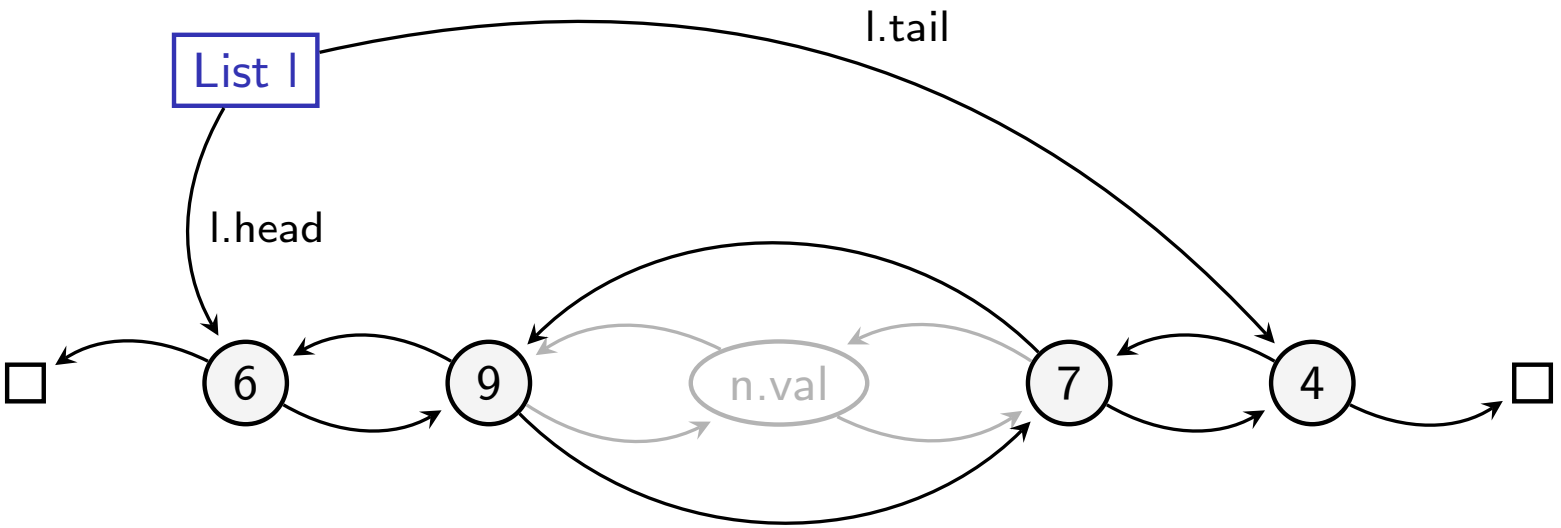
Zweifach verkettete Liste: Einfügen nach Knoten

- Einfügen eines Knoten mit Wert `val` nach einem Knoten `pre`.
- Spezialbehandlung wenn `pre` der letzte Knoten ist.



Zweifach verkettete Liste: Entfernen eines Knoten

- Entfernen des Knotens `n`.
- Spezialbehandlung wenn `n` bei Kopf oder Ende.



Zusammenfassung

- Speicheraufwand für n Elemente vom Typ T:
  - Array: `n*sizeof(T)`
  - Einfach verkettete Liste: `n*sizeof(T) + (n+1)*sizeof(Node*)`
  - Zweifach verkettete Liste: `n*sizeof(T) + 2*(n+1)*sizeof(Node*)`
- Einfügen eines Werts an einer Stelle:
  - Array: benötigt durchschnittlich $\frac{n}{2}$ Verschiebungen.
  - Verkettete Liste: nur ein paar Pointer ändern.
- Löschen eines Wertes:
  - Array: benötigt durchschnittlich $\frac{n}{2}$ Verschiebungen.
  - Einfach verkettete Liste: benötigt ca. $\frac{n}{2}$ Schritte um Knoten `pre` zu finden. Kennt man den Knoten `pre` schon, dann geht es schnell.
  - Zweifach verkettete Liste: nur ein paar Pointer ändern.
- Finden eines Elements an der Stelle i:
  - Array `a`: einfacher Zugriff mittels `a[i]`
  - Verkettete Liste: benötigt i (oder $n - 1 - i$) Schritte um Knoten zu finden.
- Auflisten aller Elemente:
  - Geht für alle gleich schnell.

Kapitel

Verschachtelte Klassen

Motivation

- Sowohl für die einfach als auch bei der zweifach verketteten Liste hätte man gerne eine `struct Node` definiert.
- Überladen von Klassen- und Strukturnamen ist nicht erlaubt.
- Lösung:
  - Variante 1: Einführen von sogenannten Namespaces.
  - Variante 2: Verwenden von verschachtelten Klassen/Strukturen (nested class/struct).

Verschachtelte Struktur: Deklaration

```
1  class SList
2  {
3    public:
4      struct Node
5      {
6        int val;
7        Node* next;
8      };
9
10     Node* head;
11 };
```

- Zugriffsrechte der äußeren Klasse werden berücksichtigt.
  - Wäre `Node` privat, dann könnte man auf den Typ `Node` außerhalb von `SList` nicht zugreifen.

Bereichsauflösungsoperator

```
1 void test()
2 {
3   SList l;
4   SList::Node head = l.head;   // Use the scope resolution operator
5 }
```

- Um auf den Typ `Node` in `SList` zuzugreifen muss der Bereichsauflösungsoperator `::` verwendet werden.

Freunde von Klassen

- Oft soll es möglich sein, dass die äußere Klasse auf private Attribute von inneren Klassen (oder umgekehrt) zugreifen kann.
  - Das Schlüsselwort `friend` sagt dem Compiler, dass eine bestimmte Klasse auf private Attribute zugreifen darf.
- Das Schlüsselwort `friend` funktioniert nicht nur für verschachtelte Typen, soll aber nur gut überlegt und zurückhaltend eingesetzt werden.

```
1 class A
2 {
3   class N
4   {
5     friend A;   // Class A can access private members of N
6   };
7   friend N;   // Class N can access private members of A
8 };
```

Beispiel SList

```
1  class SList
2  {
3    public:
4      class Node
5      {
6        friend SList; // SList can access next pointer!
7        private:
8          Node* next;
9        public:
10         int val;
11         Node* getNext();
12     };
13 };
```

- Wenn also `SList` Pointer zu Knoten der Liste nach außen gibt, dann kann von außen der `next`-Pointer nicht verändert werden. `SList` selbst kann das allerdings schon.
- Noch besser: man gibt nicht Pointer zu Knoten zurück sondern Iteratoren. (Siehe spätere Vorlesung.)