# Einführung in die Programmierung

## Dynamische Datenstrukturen

# Übersicht

- Listen
  - Typische Listenoperationen
  - *Einfach* und *doppelt* verkettete Listen
  - *Direkte* und *indirekte* Listen
- Bäume
  - *Binäre* Suchbäume
- Hash-Tabellen

# Felder

- Felder sind sehr effektiv für alle Algorithmen, die innerhalb eines *dicht besetzten* Indexraumes *wahlfrei* zugreifen müssen
- Sie sind jedoch *speicherverschwenderisch* bei *dünn besetzten* Strukturen
- *Einfügeoperationen* an beliebigen Stellen sind extrem ineffizient
- Felder mit *dynamisch veränderbarer* Größe versursachen entweder hohe *Kopierkosten* und/oder ineffektive Speicherausnutzung

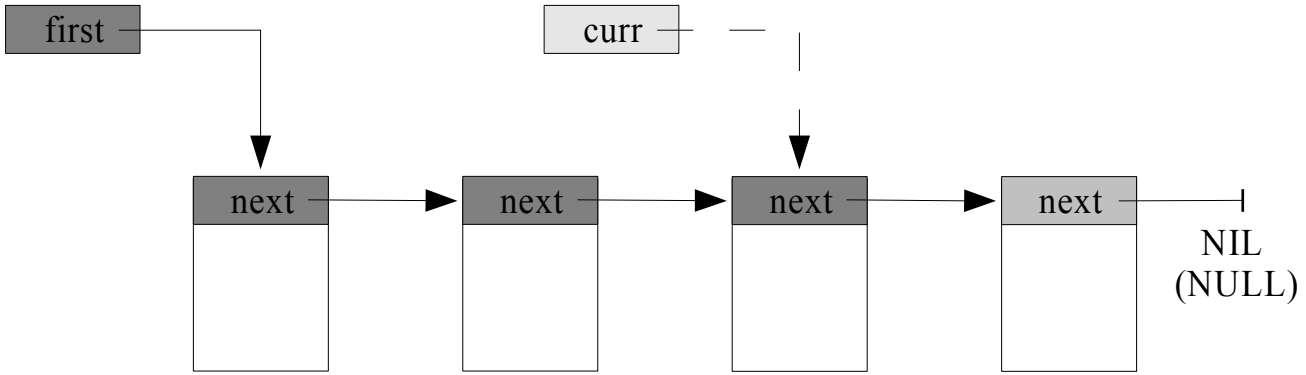
# Lineare Listen (1)

- Listen sind *dynamisch* veränderbar in der Größe
- . . . ermöglichen guten *sequentiellen* Zugriff auf die Listenelemente
- *Einfügeoperationen* sind je nach Implementierung der Liste einfach und effizient
- Sie sind aber *sehr schlecht* für *wahlfreien* Zugriff geeignet

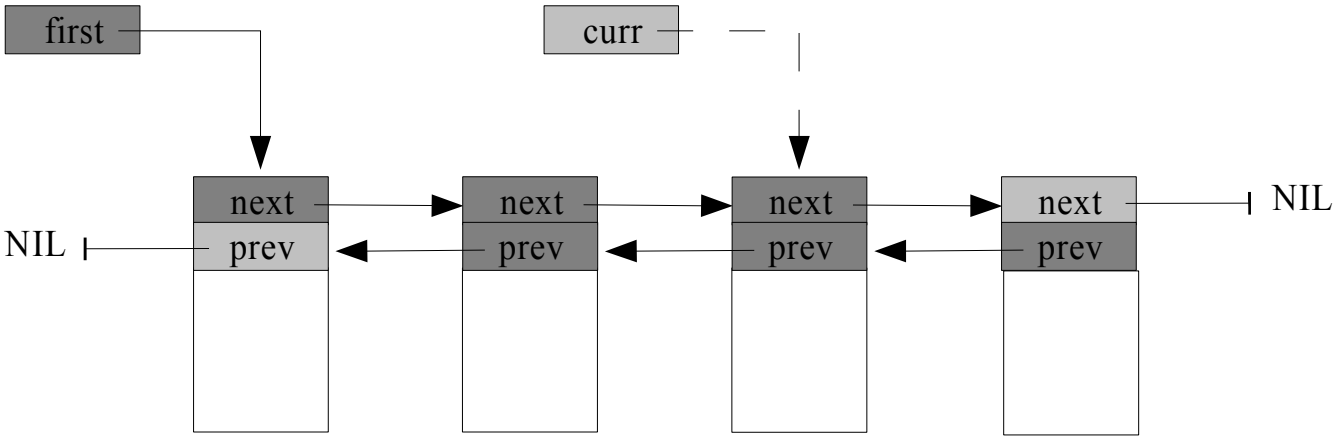
# Lineare Listen (2)

- Listen werden meistens über *Zeiger* realisiert
- Bei *einfach verketteten* Listen kennt jedes Listenelement nur seinen direkten Nachfolger
- Bei *doppelt verketteten* Listen kennt jedes Element Vorgänger und Nachfolger

# Einfach verkettete Liste



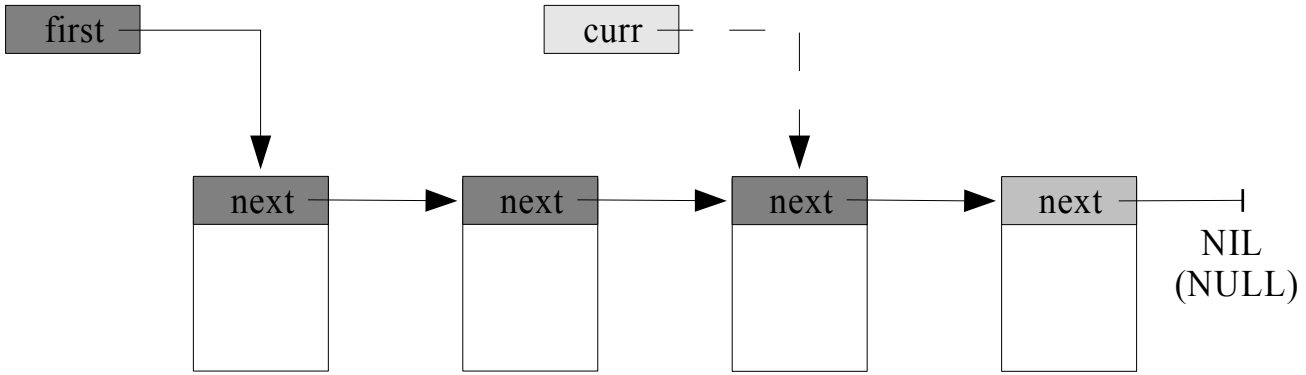
# Doppelt verkettete Liste



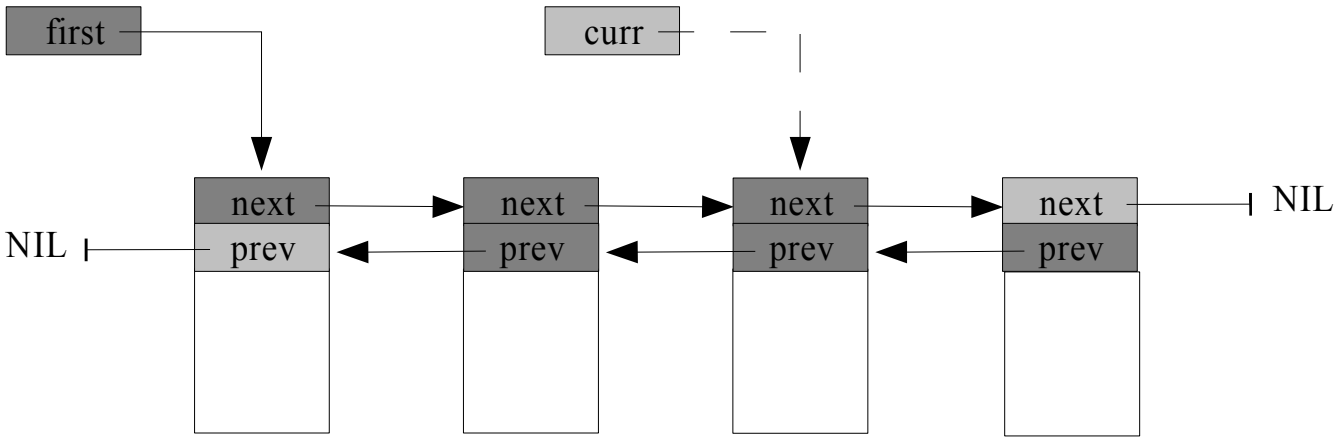
# Lineare Listen (3)

- Bei *direkten* Listen sind die Elemente direkt miteinander verkettet
- Bei *indirekten* Listen wird die Liste über Hilfselemente geführt, die jeweils einen Zeiger auf das zugehörige Listenelement haben
- *Direkte* Listen sind sehr *effizient* aber verbergen die Implementierung der Liste nicht vor den Elementen
- *Indirekte* Listen haben durch die Indirektion einen *höheren Aufwand* aber die Listenelemente müssen nichts über die Implementierung der Liste wissen

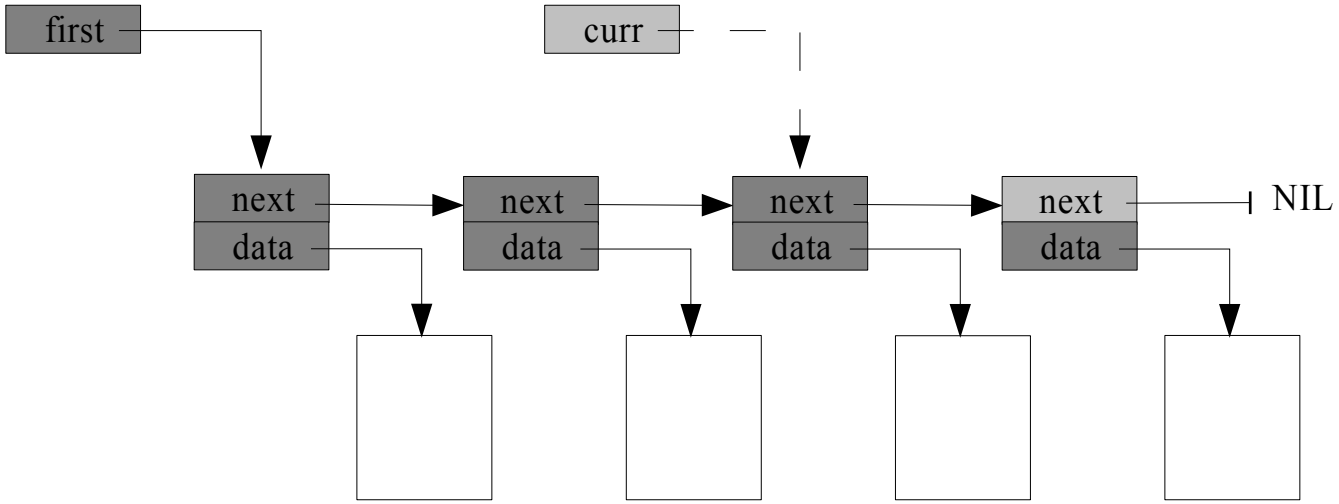
# Direkte Liste (einfach verkettet)



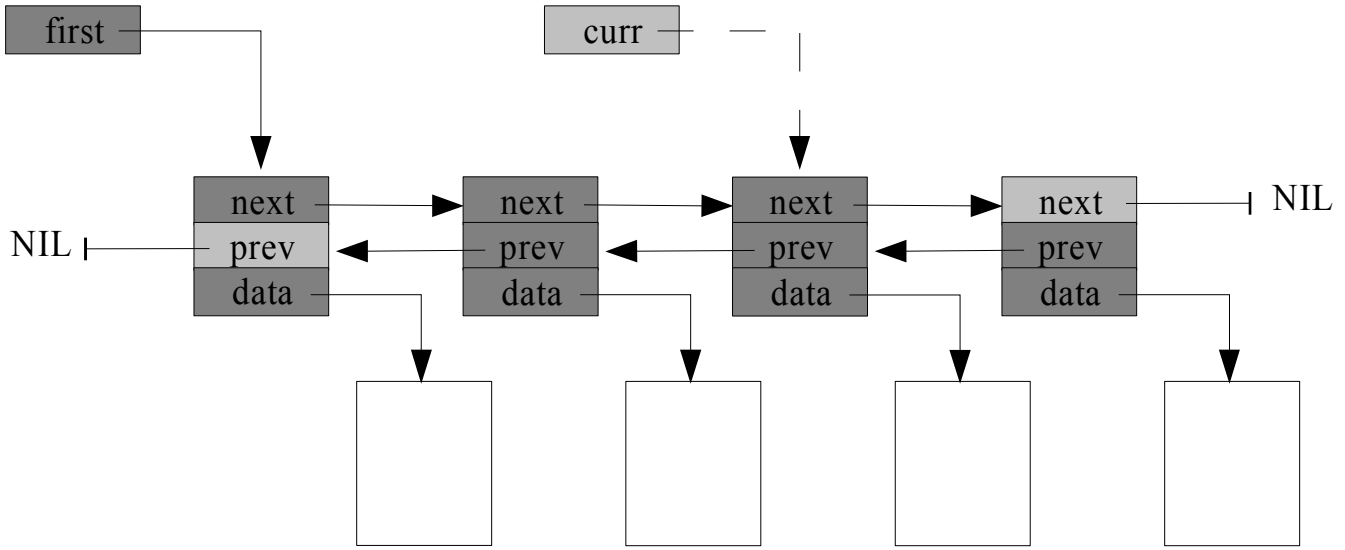
# Direkte Liste (doppelt verkettet)



# Indirekte Liste (einfach verkettet)



# Indirekte Liste (doppelt verkettet)



# Typische Listenoperationen

- *Navigation* (Zugriffsoperationen)
  - Kopf, Ende, benachbarte Elemente bestimmen
  - *Wahlfreier* Zugriff ist idR sehr *ineffizient*
- Elemente *einfügen*
  - Am Kopf, Ende oder an beliebiger Stelle
- Elemente *entfernen*
  - Am Kopf, Ende oder beliebiger Stelle
- Elemente suchen

# Navigation in Listen

```
Element* head(List* l);
Element* tail(List* l);
Element* succ(Element* e);
Element* pred(Element* e);


Element* elementAt(List* l, int pos);
void insertAt(List* l, Element* e, int pos);
```

# Elemente zu Listen hinzufügen

```
void append(List* l, Element* e);
void prepend(List* l, Element* e);
void insert(List* l, Element* e, Element* at);
void insertBehind(List* l, Element* e, Element* at);
```

# Elemente von Listen entfernen

```
Element* removeHead(List* l);
void remove(List* l, Element* e);
```

# Listen erzeugen und zerstören

```
#include<memory.h>
#include<stdio.h>
. . .
List l;
int i;

init(&l);

for (i = 0; i < 10; i ++) {
   Element* e = (Element*)malloc(sizeof(Element));
   . . .
   if (e != NULL)
      append(&l, e);
}
. . .
while (head(&l) != NULL)
   free(removeHead(&l));
```

# Listen benutzen

```
. . .

List l;
Element* curr;

curr = head(&l);
while (curr != NULL) {
      . . .
      curr = succ(curr);
}
```

# Ein minimaler Listenkopf

```
struct list {
    struct Element* first;
};

typedef struct list List;

void init(List *l) { l->first = NULL; }

Element* head(List* l)
{
    return l->first;
}
```

# Listenelemente (einfach verkettet)

```
struct element {
    struct element* next;
    . . .
};
typedef struct element Element;

Element* succ(Element* e)
{
    return e->next;
}
```

# Die prepend()-Funktion

```
void prepend(List* l, Element* e)
{
    e->next = l->first;
    l->first = e;
}
```

# Die removeHead()-Funktion

```
Element* removeHead(List* l)
{
    Element* result = l->first;
    if (result != NULL)
        l->first = result->next;
    return result;
}
```

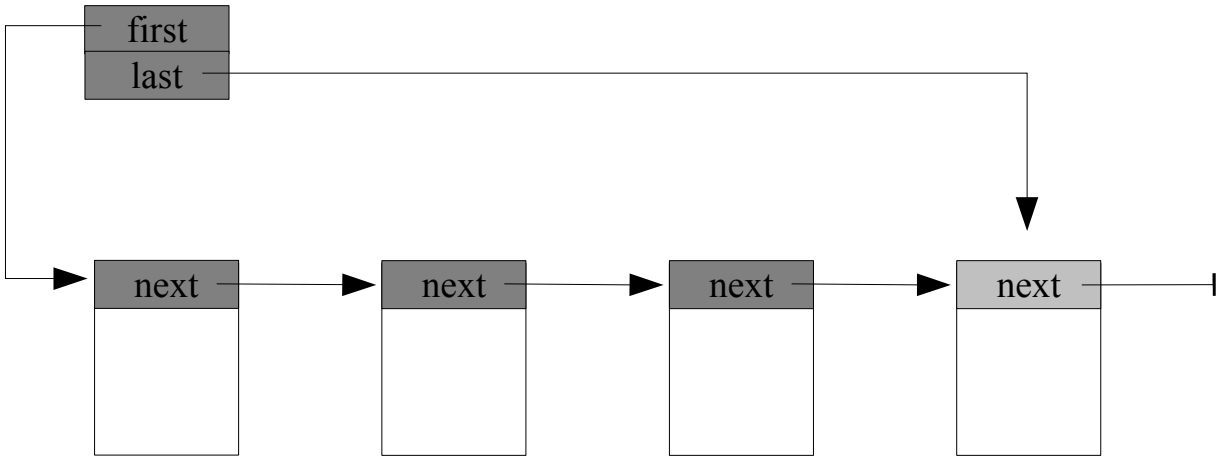
# Die append()-Funktion

```
void append(List* l, Element* e)
{
    Element* curr = l->first;
    e->next = NULL;
    if (curr == NULL) {
        l->first = e;
    } else {
        while (curr->next != NULL) {
            curr = curr->next;
        }
        curr->next = e;
    }
}
```

# Die Schlange (Queue)

- Die *Queue* ist eine der am häufigsten verwendeten Datenstrukturen
  - Bildet eine Warteschlange oder fixe Reihenfolge nach
  - Am Ende einfügen mit der `enqueue()`-Operation
  - Vom Anfang entfernen mit der `dequeue()`-Operation
- Wir brauchen zur Implementierung nur eine *einfach verkettete* Liste
  - . . . die aber für das Einfügen am Ende *optimiert* werden muß

# Queue (FIFO-Liste)



# FIFO-Liste (Queue)

```
struct queue {
     struct Element *first, *last;
};

typedef struct queue Queue;

void init(Queue *q)
{
   q->first = NULL;
   q->last = NULL;
}

Element* head(Queue* q) { return q->first; }
Element* tail(Queue* q) { return q->last; }
```

# Die enqueue()-Funktion

```
void enqueue(Queue *q, Element* e)
{
   e->next = NULL;

   if (q->last == NULL) {
        q->first = e;
   } else {
        q->last->next = e;
   }
   q->last = e;
}
```

# Die dequeue()-Funktion

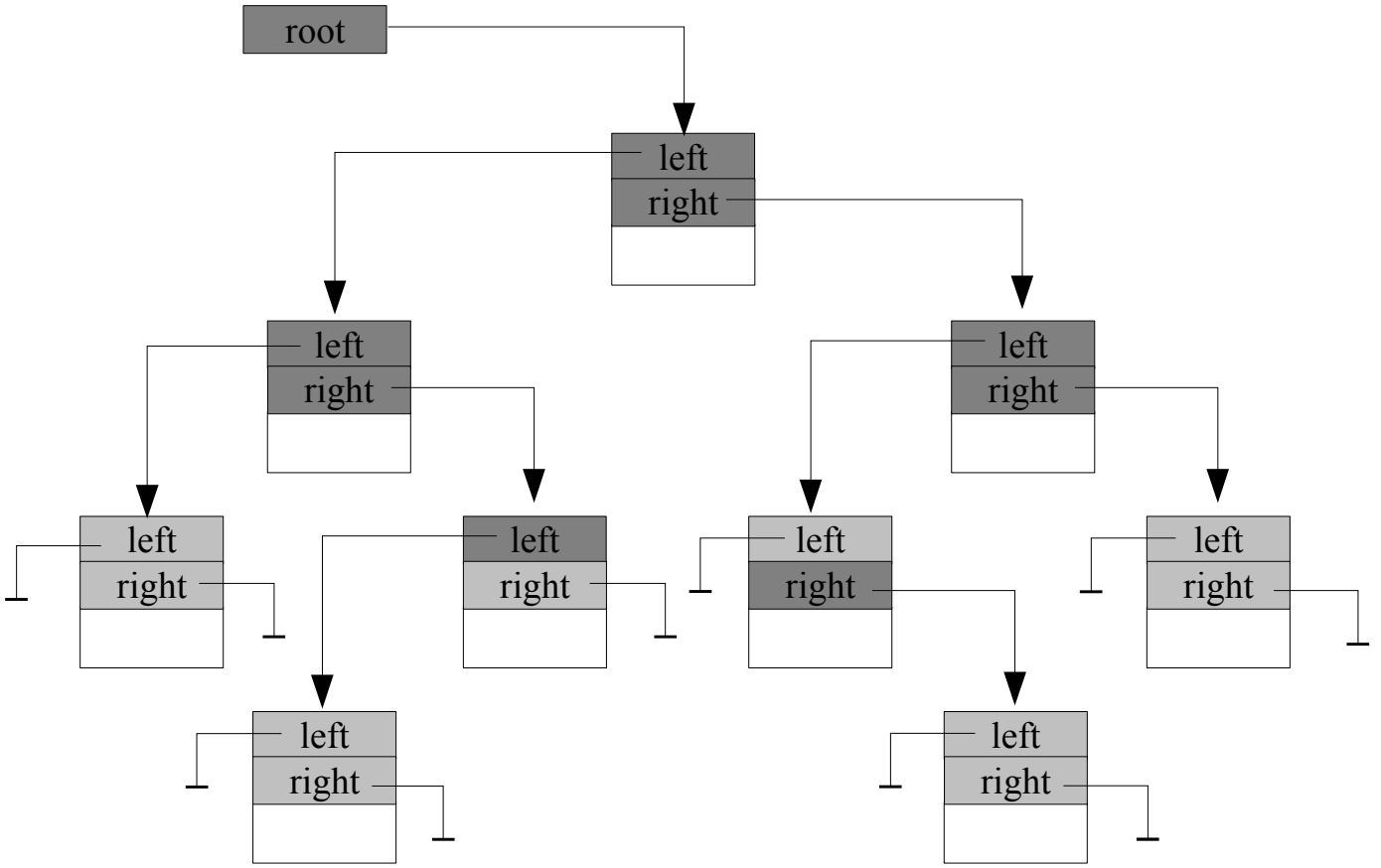
```
Element* dequeue(Queue *q)
{
   Element* e = q->first;
   if (e != NULL) {
        q->first = e->next;

        if (q->first == NULL)
             q->last = NULL;
   }
   return e;
}
```

# Binäre Bäume

- Suchen auf *Listen* ist immer *O(N)*
- Wir brauchen häufig dynamische Datenstrukturen, die eine effektivere Form der Suche ermöglichen
- Mit Hilfe von *binären Bäumen* können wir mit einem Aufwand von O(ld N) suchen

# Ein binärer Baum



# Binäre Suchbäume (1)

```
struct treenode {
      struct treenode *left, *right;
      int value;
};

typedef treenode TreeNode;
```

# Knoten einfügen

```
void insert(TreeNode* t, TreeNode* n)
{
     if (n->value <= t->value) {
          if (t->left == NULL) {
               t->left = n;
          } else {
               insert(t->left, n);
          }
     } else if (t->right == NULL) {
          t->right = n;
     } else {
          insert(t->right, n);
     }
}
```

# Knoten suchen

```
TreeNode* search(TreeNode* t, int key)
{
     TreeNode* result = NULL;
     if (t->value == key) {
          result = t;
     } else if (t->value < key) {
          if (t->left != NULL) {
               result = search(t->left, key);
          }
     } else if (t->right != NULL) {
          result = search(t->right, n);
     }
     return result;
}
```

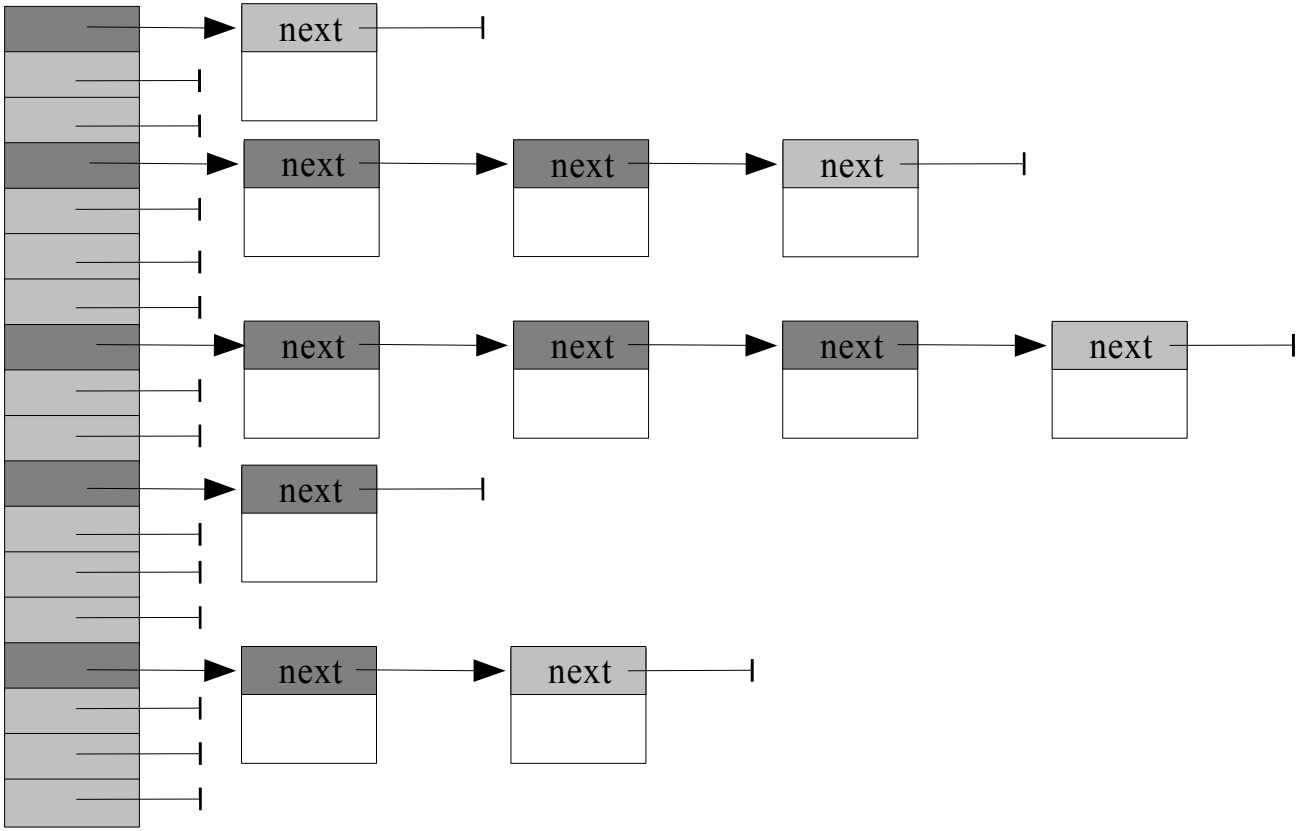
# Binäre Suchbäume (2)

- Binäre Bäume sind effektiv für die Suche, sie können jedoch zu linearen Listen *degenerieren*
  - Der Baum muß neu *balanciert* werden
  - *Aufwendig* und kompliziert
- Jeder Knoten muß *zwei* Zeiger enthalten
  - Im Vergleich zur linearen Liste doppelt soviel Speicherverbrauch zur Verwaltung der Datenstruktur

# Hash-Tabellen (1)

- Hash-Tabellen vereinigen in sich die *Flexibilität* der linearen Liste mit der *Effizienz* einer Feldindizierung
  - Ein *gemischtes* Zugriffsverfahren bestehend aus *Indizierung* und *linearer Suche*
- Die Tabelle enthält eine *Feld* von *Listenköpfen*
  - Mit Hilfe einer *Hash-Funktion* wird ein Feldindex bestimmt, um die Liste zu bestimmen
  - Danach wird auf der Liste linear gesucht

# Hash-Tabellen (2)



# Hash-Tabellen (3)

- Hash-Tabellen können als eine Generalisierung eines Feldes angesehen werden
  - Dynamische Größe,  große Laufzeit- und Speichereffizienz
  - Die Laufzeiteffizienz hängt jedoch stark von der *Anzahl* der verwendeten Listen und der *Hash-Funktion* ab
- Eine der *meistbenutzten* Datenstrukturen!

# Zusammenfassung

- *Dynamische Datenstrukturen* finden in Algorithmen aller Art Verwendung
- Die *lineare Liste* ist die einfachste dynamische Datenstruktur und universell einsetzbar, sofern kein wahlfreier Zugriff benötigt wird
- *Bäume* und *Hash-Tabellen* ermöglichen *wahlfreien* Zugriff und finden bei Abbildungsproblemen aller Art Anwendung