

Datenstrukturen & Algorithmen

Peppo Brambilla
Universität Bern
Frühling 2018

Übersicht

Elementare Datenstrukturen für dynamische Mengen

- Stapel & Warteschlangen
- Verkettete Listen
- Bäume
- Anwendungsbeispiel: kd-Bäume

Dynamische Mengen

- Daten in einer Menge speichern und abfragen
- Grösse der Menge ist dynamisch
- Daten: Records aus Schlüssel und Satellitendaten

```
class Record {  
    int key;  
    SatelliteData data;  
}
```

```
class SatelliteData{  
    String name;  
    String address;  
    ...  
}
```

Operationen

- Dynamische Menge S , Schlüssel k , Referenz auf Record x
 - $\text{Search}(S, k)$
 - $\text{Insert}(S, x)$
 - $\text{Delete}(S, x)$
 - $\text{Minimum}(S)$
 - $\text{Maximum}(S)$
 - $\text{Successor}(S, x)$
 - $\text{Predecessor}(S, x)$

Datenstrukturen

- Geeignete Datenstruktur **je nach Operationen**, die effizient unterstützt werden soll
- Zeitkomplexität wird als Funktion der Grösse n der Menge gemessen
- Beispiel: Heaps (Min-Heap/Max-Heap)
 - Abfragen von Minimum/Maximum in $O(1)$
 - Entfernen von Minimum/Maximum in $O(\lg n)$
 - Einfügen in $O(\lg n)$

Stapel & Warteschlangen

- Dynamische Mengen mit effizientem Einfügen und Entfernen in $O(1)$
 - $\text{Insert}(S, x)$
 - $x = \text{Remove}(S)$

Stapel

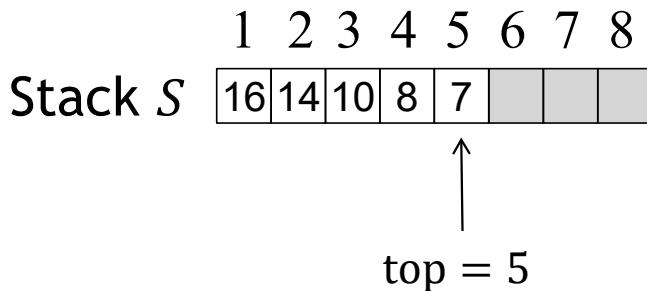
- Einfügen und Entfernen mit **last-in-, first-out**-Strategie (LIFO)

Warteschlangen

- Einfügen und Entfernen mit **first-in-, first-out**-Strategie (FIFO)

Stapel (LIFO)

- Einfügen = **push**, Entfernen = **pop**, Aufwand: $O(1)$
- Hier: Implementation mit Feld
 - Implementation mit anderen Datenstrukturen möglich
 - Elemente hier der Einfachheit halber Integers, Records mit Satellitendaten funktionieren gleich
- Statusvariable `top`

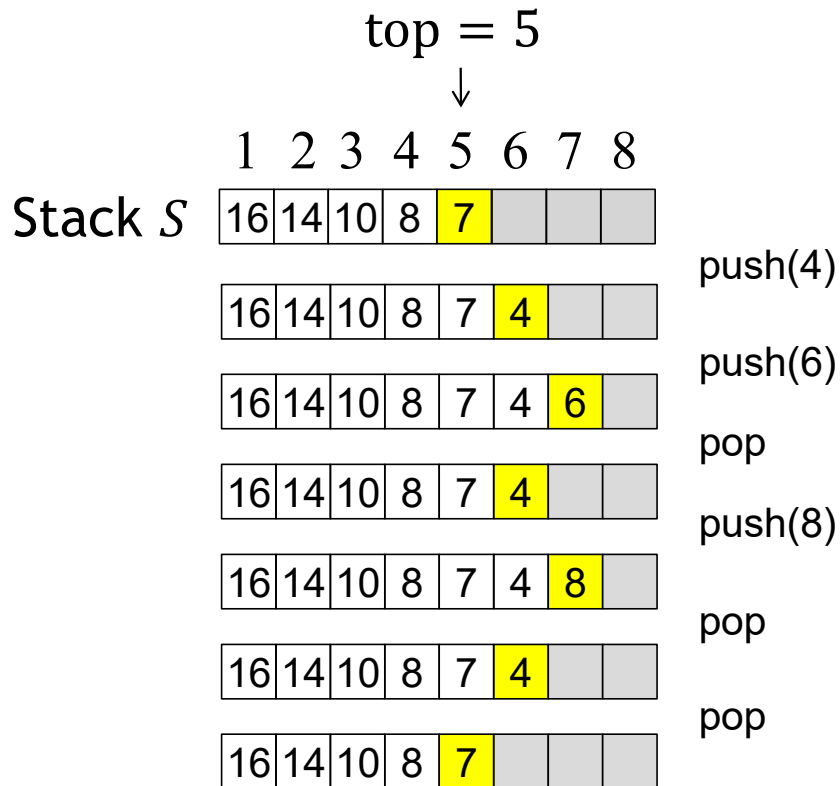


```
class Stack {  
    private int top;  
    private int S[];  
    /* constructor,  
       methods push, pop */  
}
```

Stapel

- Beispiel

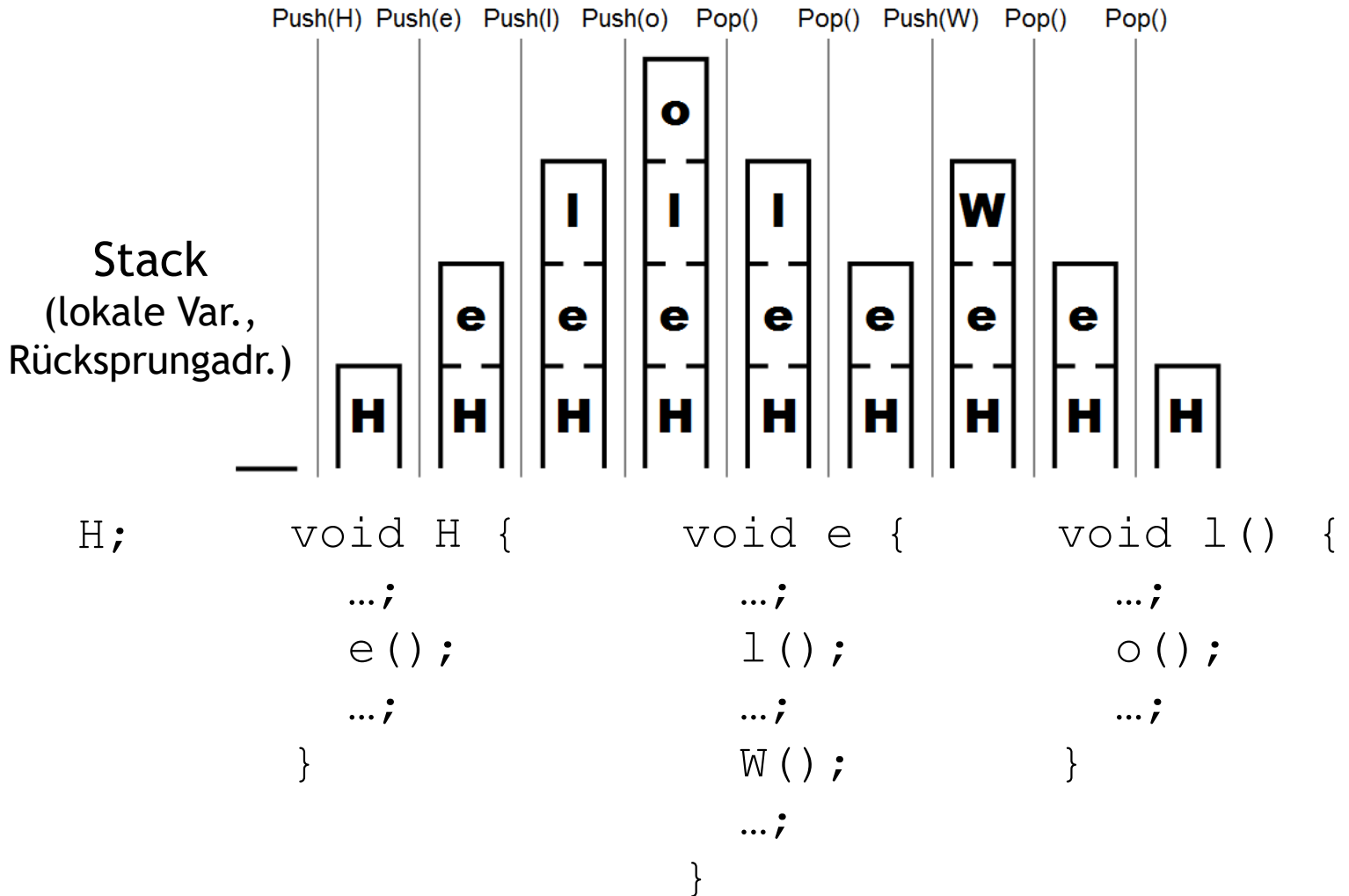
push(4); push(6); pop; push(8); pop; pop;



Anwendung: Call Stack

- Verwaltung von Methodenaufrufen
 - Speichert Rücksprungadresse und lokale Variablen
- Aufruf einer Methode
 - Push: Rücksprungadresse
 - Push: Speicherplatz für lokale Variablen der aufgerufenen Methode
- Ende der Methode
 - Pop: lokale Variablen
 - Pop: Rücksprungadresse

Anwendung: Call Stack



Stack overflow

- Zu tiefe Verschachtelung von Aufrufen
- Zu hohe Rekursionstiefe, so dass Stack Speicher zu klein
- Führt zu Terminierung des Programms
- Beispiel: worst case Ablauf von Quicksort
- Rekursive Aufrufe können durch eigene Verwaltung eines Stacks vermieden werden
- Vorteile
 - Kleinerer Aufwand
 - Programm kann mehr Speicher für Stack allokieren wenn nötig

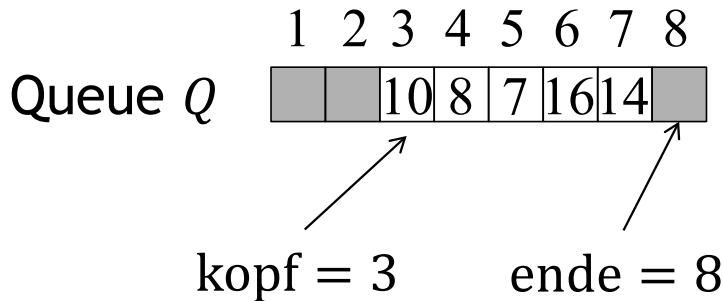
Anwendung

- Nicht-rekursiver QuickSort mit Stack

```
void quickSort(int array[], int left, int right) {  
    Stack leftStack = new Stack();  
    Stack rightStack = new Stack();  
    leftStack.push(left);  
    rightStack.push(right);  
    while( ! leftStack.isEmpty() ) {  
        left = leftStack.pop();  
        right = rightStack.pop();  
        if( left < right ) {  
            int p = partition(array, left, right);  
            leftStack.push(left);  
            rightStack.push(p-1);  
            leftStack.push(p+1);  
            rightStack.push(right);  
        }  
    }  
}
```

Warteschlange (FIFO)

- Einfügen = **queue**, Entfernen = **dequeue**, Aufwand: $O(1)$
- Hier: Implementation mit Feld
 - Implementation mit anderen Datenstrukturen möglich (z.B. verkettete Listen)
- Statusvariable **kopf**, **ende**
 - **ende** zeigt auf **erstes freies** Element

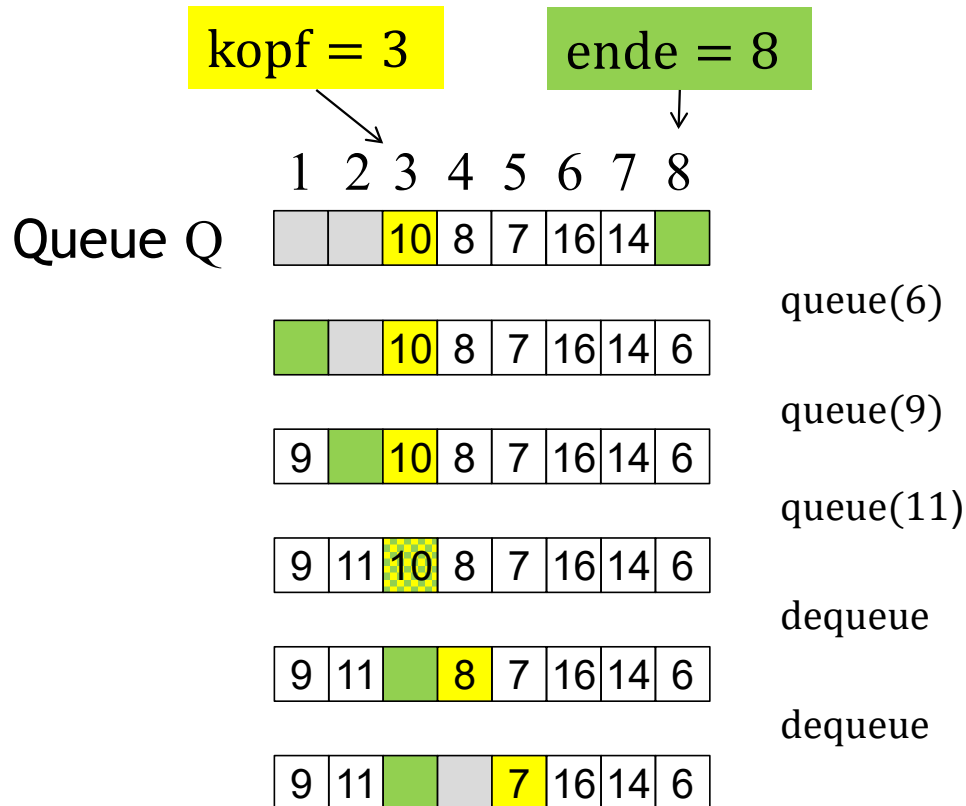


```
class Queue {  
    private int kopf, ende;  
    private int Q[];  
    /* constructor,  
       methods queue,  
       dequeue */  
}
```

Warteschlange

- Beispiel

queue(6); queue(9); queue(11); dequeue; dequeue



Anwendungen

- Verwaltung von Aufträgen, die mittels einer **first-come, first-serve** Strategie bearbeitet werden sollen
 - Z.B. Printer Server, Datenbankabfragen über Internet

Übersicht

Elementare Datenstrukturen

- Stapel & Warteschlangen
- Verkettete Listen
- Bäume
- Anwendungsbeispiel: kd-Bäume

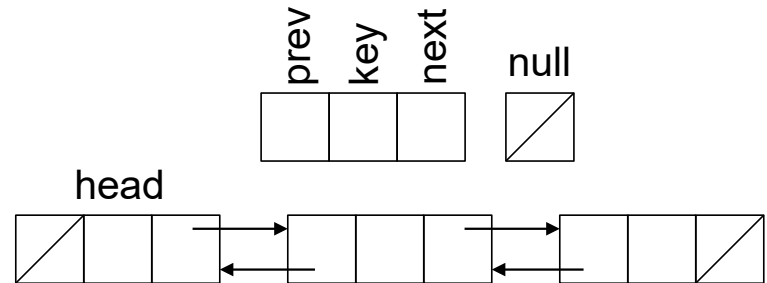
Verkettete Listen

- Vorteil gegenüber Feldern: maximale Anzahl Elemente ist **dynamisch**
- Nachteil: Zugriff auf ein Element in konstanter Zeit mittels Index nicht möglich
- Können Stacks und Warteschlangen auch mit verketteten Listen implementieren
- Varianten
 - Einfach verkettete Liste
 - Doppelt verkettete Liste
 - Zyklische Liste
 - (Sortierte Liste)

Verkettete Listen

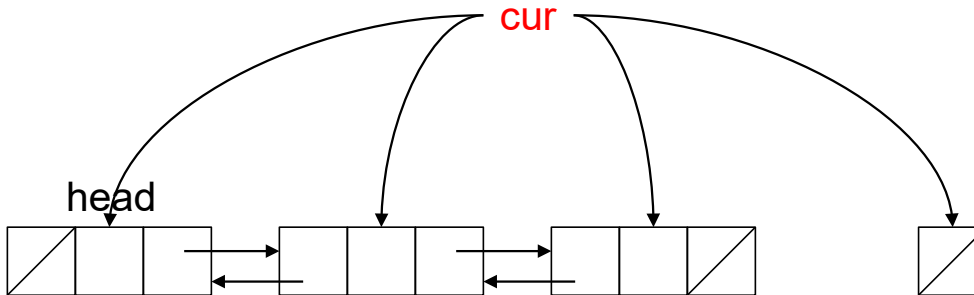
- Listenelemente bestehen aus
 - Schlüssel (oder Referenz auf andere Daten)
 - Zeiger next auf nachfolgendes Listenelement
 - **Doppelt verkettet**: zusätzlich Zeiger prev auf vorheriges Listenelement
- Anfang: Zeiger head auf erstes Listenelement
- Ende der Liste: null Zeiger

```
class LinkedList {  
    private class ListElement {  
        ListElement prev, next;  
        int key;  
    }  
    ListElement head;  
    /* methods to add elements, etc. */  
}
```



Durchsuchen

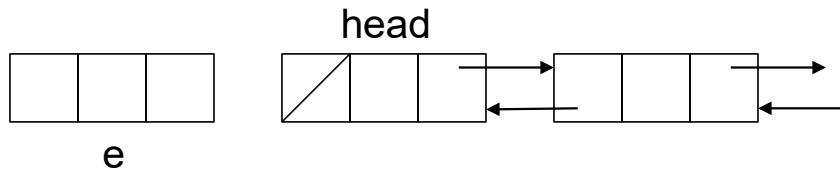
```
ListElement Search(int k) {  
    ListElement cur = head;  
    while( cur != null && k != cur.key )  
        cur = cur.next;  
    return cur;  
}
```



Einfügen

- Doppelt verkettete Liste

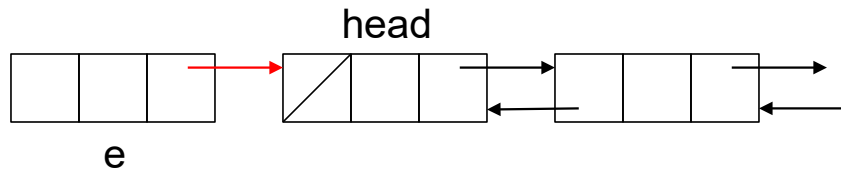
```
1 void insert(ListElement e) {  
2     e.next = head;  
3     if( head != null )  
4         head.prev = e;  
5     head = e;  
6     e.prev = null;  
7 }
```



Einfügen

- Doppelt verkettete Liste

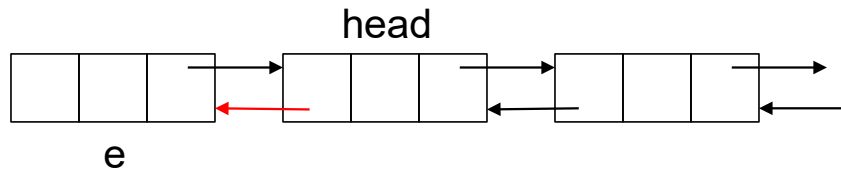
```
1 void insert(ListElement e) {  
2     e.next = head;  
3     if( head != null )  
4         head.prev = e;  
5     head = e;  
6     e.prev = null;  
7 }
```



Einfügen

- Doppelt verkettete Liste

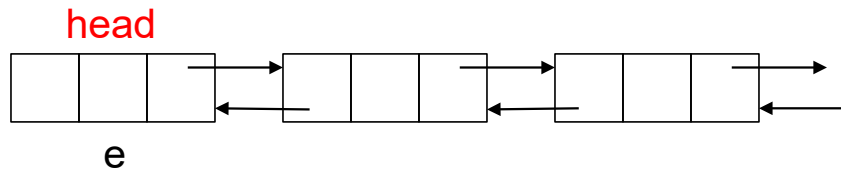
```
1 void insert(ListElement e) {  
2     e.next = head;  
3     if( head != null )  
4         head.prev = e;  
5     head = e;  
6     e.prev = null;  
7 }
```



Einfügen

- Doppelt verkettete Liste

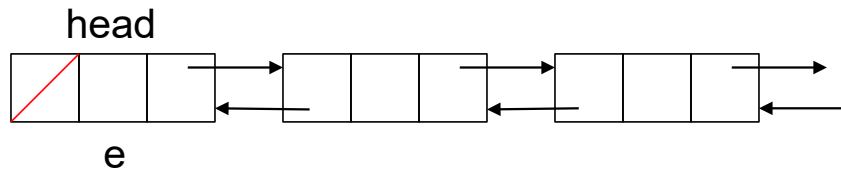
```
1 void insert(ListElement e) {  
2     e.next = head;  
3     if( head != null )  
4         head.prev = e;  
5     head = e;  
6     e.prev = null;  
7 }
```



Einfügen

- Doppelt verkettete Liste

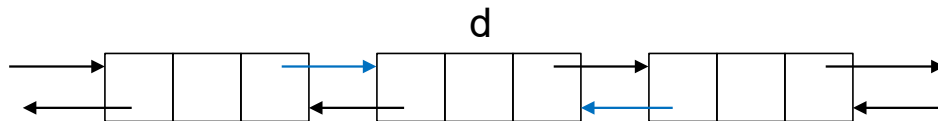
```
1 void insert(ListElement e) {  
2     e.next = head;  
3     if( head != null )  
4         head.prev = e;  
5     head = e;  
6     e.prev = null;  
7 }
```



Entfernen

- Doppelt verkettete Liste

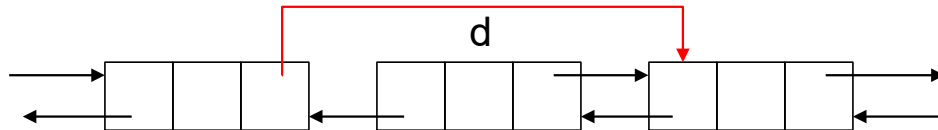
```
1 void delete(ListElement d) {  
2     if( d.prev != null )  
3         d.prev.next = d.next;  
4     else  
5         head = d.next;  
6     if( d.next != null )  
7         d.next.prev = d.prev;  
8 }
```



Entfernen

- Doppelt verkettete Liste

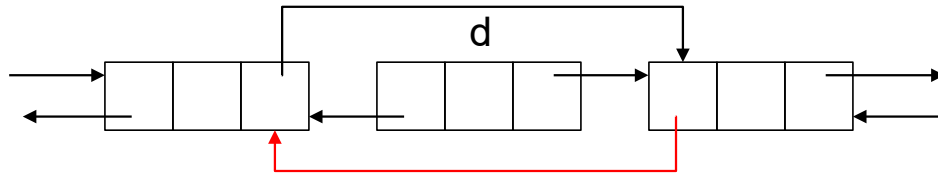
```
1 void delete(ListElement d) {  
2     if( d.prev != null )  
3         d.prev.next = d.next;  
4     else  
5         head = d.next;  
6     if( d.next != null )  
7         d.next.prev = d.prev;  
8 }
```



Entfernen

- Doppelt verkettete Liste

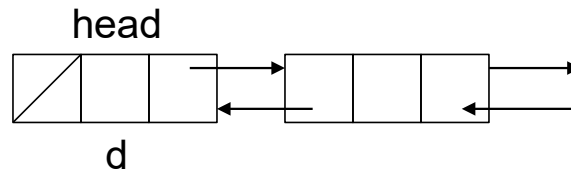
```
1 void delete(ListElement d) {  
2     if( d.prev != null )  
3         d.prev.next = d.next;  
4     else  
5         head = d.next;  
6     if( d.next != null )  
7         d.next.prev = d.prev;  
8 }
```



Entfernen

- Doppelt verkettete Liste

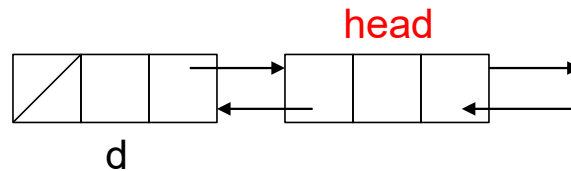
```
1 void delete(ListElement d) {  
2     if( d.prev != null )  
3         d.prev.next = d.next;  
4     else  
5         head = d.next;  
6     if( d.next != null )  
7         d.next.prev = d.prev;  
8 }
```



Entfernen

- Doppelt verkettete Liste

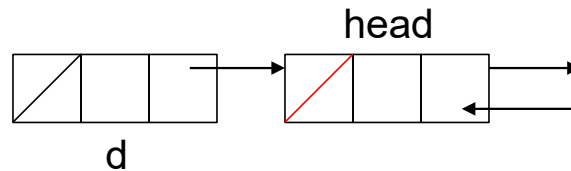
```
1 void delete(ListElement d) {  
2     if( d.prev != null )  
3         d.prev.next = d.next;  
4     else  
5         head = d.next;  
6     if( d.next != null )  
7         d.next.prev = d.prev;  
8 }
```



Entfernen

- Doppelt verkettete Liste

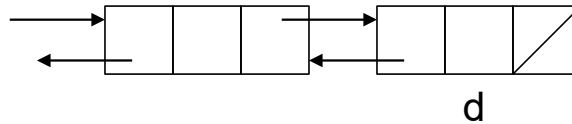
```
1 void delete(ListElement d) {  
2     if( d.prev != null )  
3         d.prev.next = d.next;  
4     else  
5         head = d.next;  
6     if( d.next != null )  
7         d.next.prev = d.prev;  
8 }
```



Entfernen

- Doppelt verkettete Liste

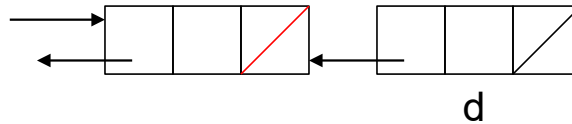
```
1 void delete(ListElement d) {  
2     if( d.prev != null )  
3         d.prev.next = d.next;  
4     else  
5         head = d.next;  
6     if( d.next != null )  
7         d.next.prev = d.prev;  
8 }
```



Entfernen

- Doppelt verkettete Liste

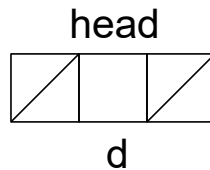
```
1 void delete(ListElement d) {  
2     if( d.prev != null )  
3         d.prev.next = d.next;  
4     else  
5         head = d.next;  
6     if( d.next != null )  
7         d.next.prev = d.prev;  
8 }
```



Entfernen

- Doppelt verkettete Liste

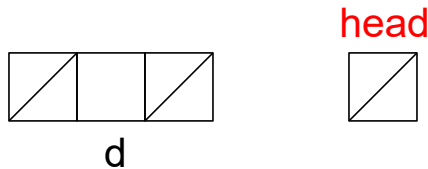
```
1 void delete(ListElement d) {  
2     if( d.prev != null )  
3         d.prev.next = d.next;  
4     else  
5         head = d.next;  
6     if( d.next != null )  
7         d.next.prev = d.prev;  
8 }
```



Entfernen

- Doppelt verkettete Liste

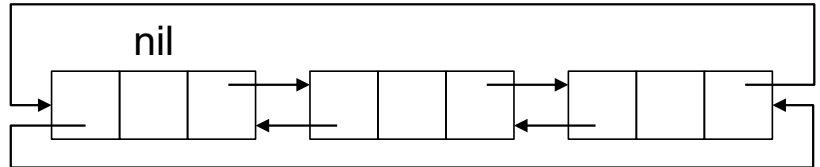
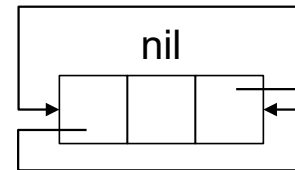
```
1 void delete(ListElement d) {  
2     if( d.prev != null )  
3         d.prev.next = d.next;  
4     else  
5         head = d.next;  
6     if( d.next != null )  
7         d.next.prev = d.prev;  
8 }
```



Zyklische Liste mit Wächter

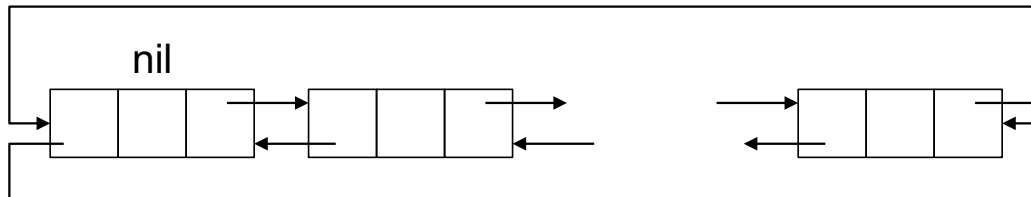
- Verknüpfte Liste zu einem Ring (**zyklische Liste**)
- Spezielles Wächterobjekt nil, um Randbedingungen zu vereinfachen
- Zeiger auf Wächter nil, ersetzt head
- Hier doppelt verkettete Liste

```
class CyclicList {  
    private class ListElement {  
        ListElement prev, next;  
        int key;  
    }  
    ListElement nil;  
    /* methods to add elements, etc. */  
}
```



Einfügen in Liste mit Wächter

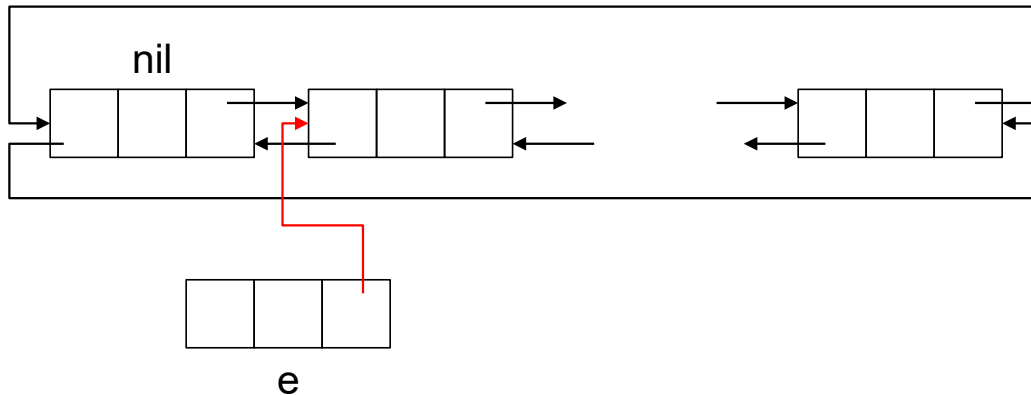
```
1 void insertElement(ListElement e) {  
2   e.next = nil.next;  
3   nil.next.prev = e;  
4   nil.next = e;  
5   e.prev = nil;  
6 }
```



e

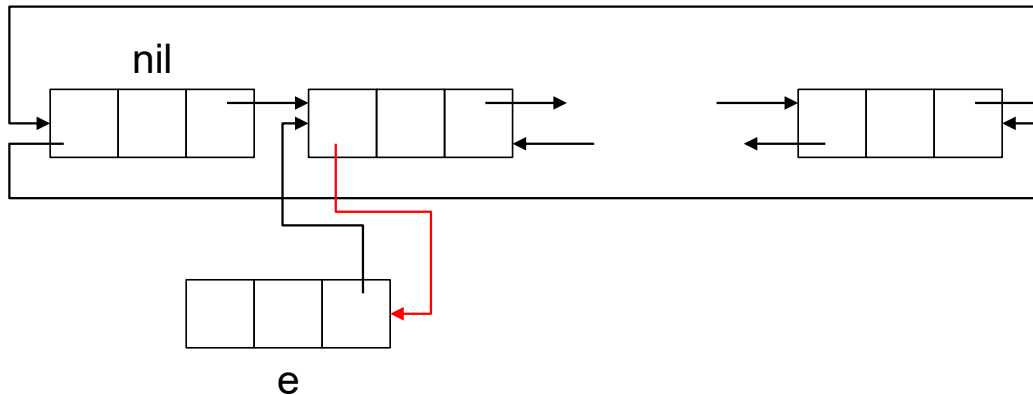
Einfügen in Liste mit Wächter

```
1 void insertElement(ListElement e) {  
2   e.next = nil.next;  
3   nil.next.prev = e;  
4   nil.next = e;  
5   e.prev = nil;  
6 }
```



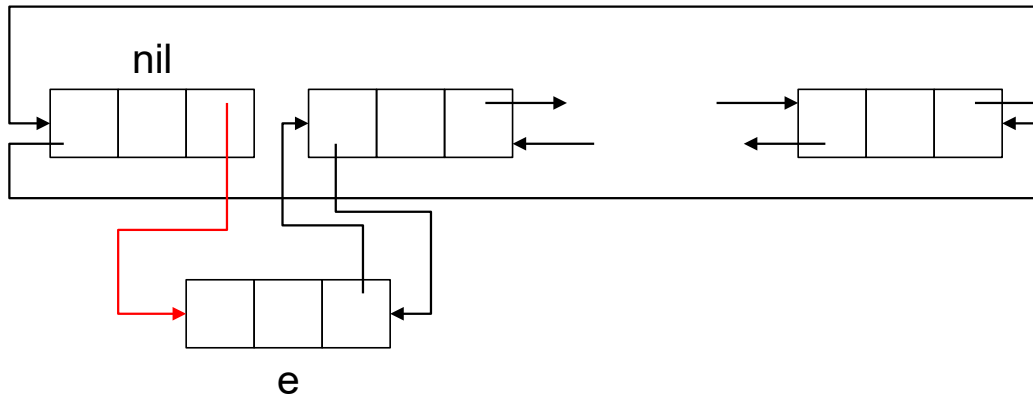
Einfügen in Liste mit Wächter

```
1 void insertElement(ListElement e) {  
2   e.next = nil.next;  
3   nil.next.prev = e;  
4   nil.next = e;  
5   e.prev = nil;  
6 }
```



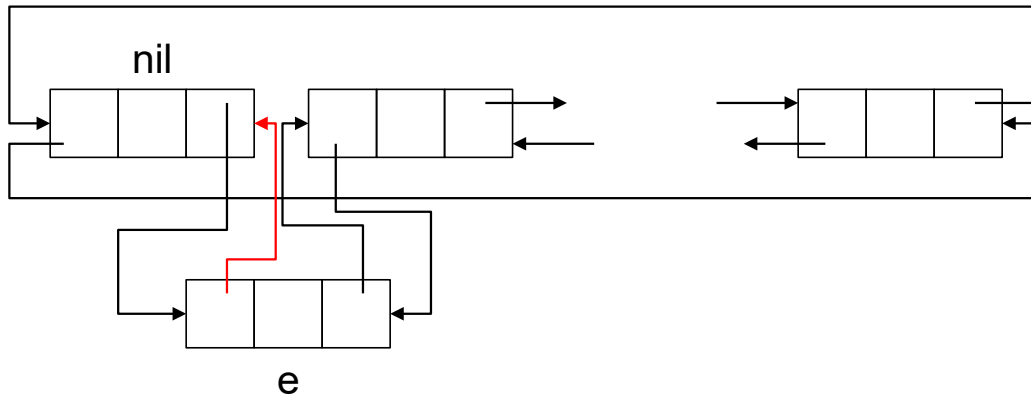
Einfügen in Liste mit Wächter

```
1 void insertElement(ListElement e) {  
2   e.next = nil.next;  
3   nil.next.prev = e;  
4   nil.next = e;  
5   e.prev = nil;  
6 }
```



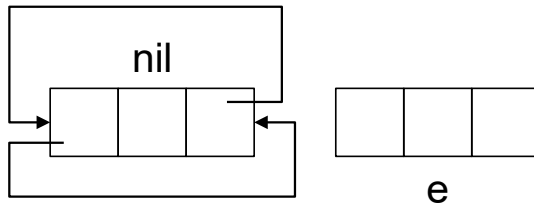
Einfügen in Liste mit Wächter

```
1 void insertElement(ListElement e) {  
2   e.next = nil.next;  
3   nil.next.prev = e;  
4   nil.next = e;  
5   e.prev = nil;  
6 }
```



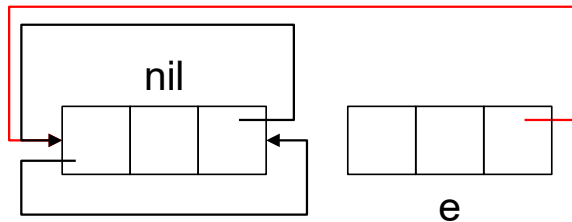
Einfügen in Liste mit Wächter

```
1 void insertElement(ListElement e) {  
2     e.next = nil.next;  
3     nil.next.prev = e;  
4     nil.next = e;  
5     e.prev = nil;  
6 }
```



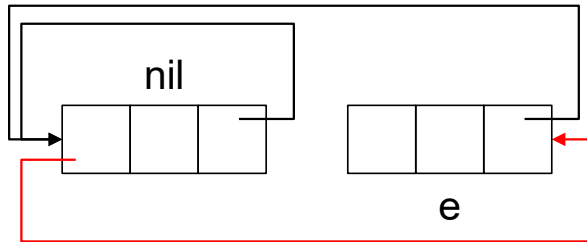
Einfügen in Liste mit Wächter

```
1 void insertElement(ListElement e) {  
2     e.next = nil.next;  
3     nil.next.prev = e;  
4     nil.next = e;  
5     e.prev = nil;  
6 }
```



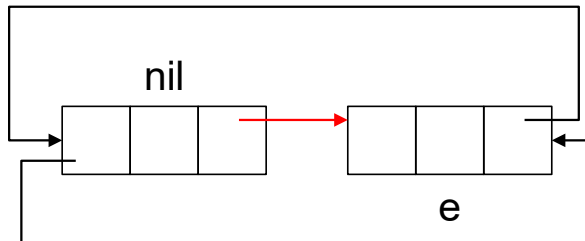
Einfügen in Liste mit Wächter

```
1 void insertElement(ListElement e) {  
2   e.next = nil.next;  
3   nil.next.prev = e;  
4   nil.next = e;  
5   e.prev = nil;  
6 }
```



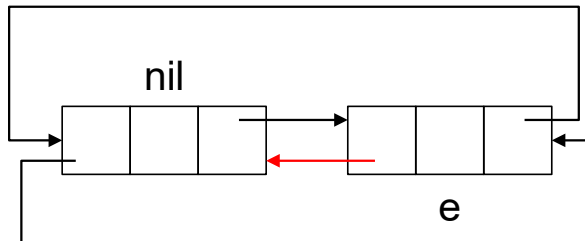
Einfügen in Liste mit Wächter

```
1 void insertElement(ListElement e) {  
2     e.next = nil.next;  
3     nil.next.prev = e;  
4     nil.next = e;  
5     e.prev = nil;  
6 }
```



Einfügen in Liste mit Wächter

```
1 void insertElement(ListElement e) {  
2     e.next = nil.next;  
3     nil.next.prev = e;  
4     nil.next = e;  
5     e.prev = nil;  
6 }
```



Implementierung

- Möchten Instanzen (Objekte) beliebiger Klassen in Listen speichern können
- Wollen gleichen Code für Listen verwenden
 - Soll für Objekte beliebiger Klassen funktionieren
- Naive Lösung in Java: die Implementierung der Liste akzeptiert Instanzen der Klasse Object, die Basisklasse aller Klassen
- Nachteil: keine Typprüfung mehr möglich
- Bessere Lösung: Parametrisierte Klassen

Parameterisierte Klasse in Java

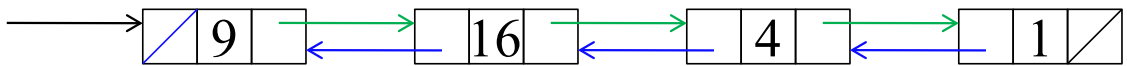
```
class LinkedList<T> {  
    private class Element {  
        private Element prev, next;  
        private T data; // Satellitendaten  
    }  
    private Element nil;  
    LinkedList() {  
        nil = new Element();  
        nil.next = nil;  
        nil.prev = nil;  
    }  
    void insert(T e) {...}  
    /* add other operations */  
}
```

```
// Make a list containing integers and add an element  
LinkedList<Integer> integerList = new LinkedList<Integer>();  
integerList.insert(12);
```

Zeiger und Objekte mit Feldern

- Was, wenn eine Programmiersprache keine Zeiger zur Verfügung stellt?
- Implementiere Zeiger und Objekte mit Feldern

Zeiger und Objekte mit Feldern

- Liste **head** 

- Mehrfelddarstellung

1 2 3 4 5 6 7 8

next		3	/		2		5	
key		4	1		16		9	
prev		5	2		7		/	

- Einfelddarstellung

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

			4	7	13	1	/	4				16	4	19				9	13	/			
--	--	--	---	---	----	---	---	---	--	--	--	----	---	----	--	--	--	---	----	---	--	--	--

Allokieren und Freigeben von Objekten

- Wollen Operationen auf dynamischen Listen anbieten wie vorher
 - Insbesondere, allokalieren und einfügen neuer Objekte
 - Allokieren ähnlich „new ()“ in Java
 - Brauchen Zugriff auf freie Elemente in Felder
- Unterhalte zwei Listen
- Liste mit **Daten**
 - Doppelt verknüpft
 - Kopf heisst head
- Liste mit **freien Objekten**
 - Einfach verknüpft
 - Kopf heisst free

				free			head	
	1	2	3	4	5	6	7	8
next	/	3	/	8	2	1	5	6
key		4	1		16		9	
prev		5	2		7		/	

Allokieren und Freigeben

```
x = allocateObject();  
insert(x);  
key[x] = 25;  
delete(5);
```

1 2 3 4 5 6 7 8

next	/	3	/	8	2	1	5	6
key		4	1		16		9	
prev		5	2		7		/	

Allokieren und Freigeben

```
x = allocateObject();  
insert(x);  
key[x] = 25;  
delete(5);
```

1 2 3 4 5 6 7 8

next	/	3	/	8	2	1	5	6
key		4	1		16		9	
prev		5	2		7		/	

x

1 2 3 4 5 6 7 8

next	/	3	/	8	2	1	5	6
key		4	1		16		9	
prev		5	2		7		/	

Allokieren und Freigeben

```
x = allocateObject();
```

```
insert(x);
```

```
key[x] = 25;
```

```
delete(5);
```

x

1 2 3 4 5 6 7 8

next	/	3	/	8	2	1	5	6
key		4	1		16		9	
prev		5	2		7		/	

x

1 2 3 4 5 6 7 8

next	/	3	/	7	2	1	5	6
key		4	1		16		9	
prev		5	2	/	7		4	

Allokieren und Freigeben

```
x = allocateObject();  
insert(x);  
key[x] = 25;  
delete(5);
```

	x							
	1	2	3	4	5	6	7	8
next	/	3	/	7	2	1	5	6
key		4	1		16		9	
prev		5	2	/	7		4	

	x							
	1	2	3	4	5	6	7	8
next	/	3	/	7	2	1	5	6
key		4	1	25	16		9	
prev		5	2	/	7		4	

Allokieren und Freigeben

```
x = allocateObject();  
insert(x);  
key[x] = 25;  
delete(5);
```

1 2 3 4 5 6 7 8

next	/	3	/	7	2	1	5	6
key		4	1	25	16		9	
prev		5	2	/	7		4	

1 2 3 4 5 6 7 8

next	/	3	/	7	8	1	2	6
key		4	1	25	16		9	
prev		7	2	/	7		4	

Übersicht

Elementare Datenstrukturen

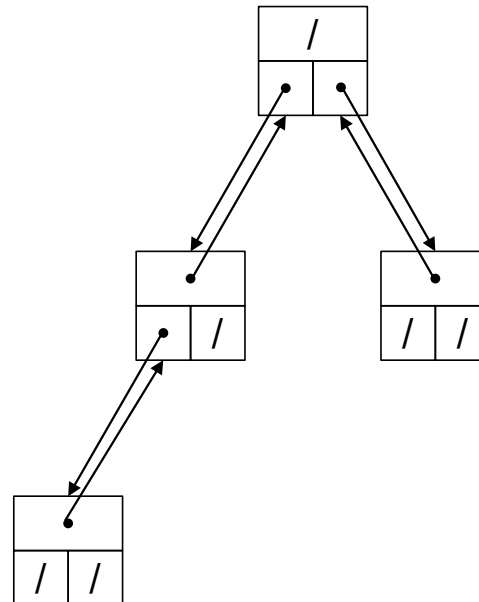
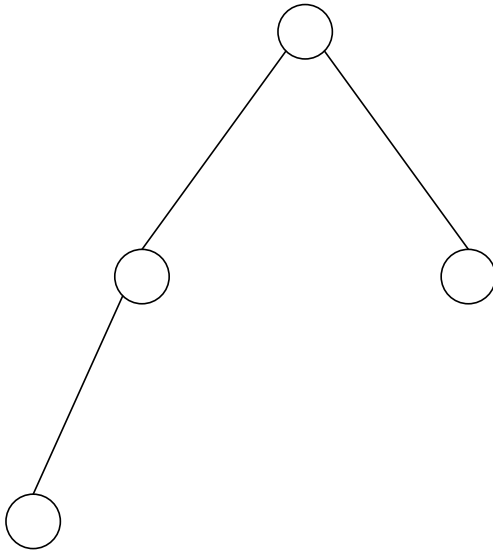
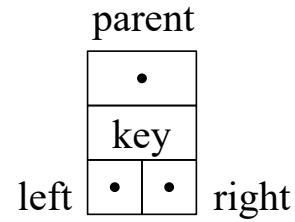
- Stapel & Warteschlangen
- Verkettete Listen
- **Bäume**
- Anwendungsbeispiel: kd-Bäume

Gerichtete Bäume

- Verallgemeinerung von linearen Listen mit Verkettung zu einer Baumstruktur
- Wie bei Listen: Elemente enthalten
 - Schlüssel (oder Referenzen auf Satellitendaten)
 - Referenzen zur Verkettung
- Grundlage für viele Algorithmen
 - z.B. Suchen
- Heute
 - Binäre Bäume
 - Bäume mit unbeschränktem Grad

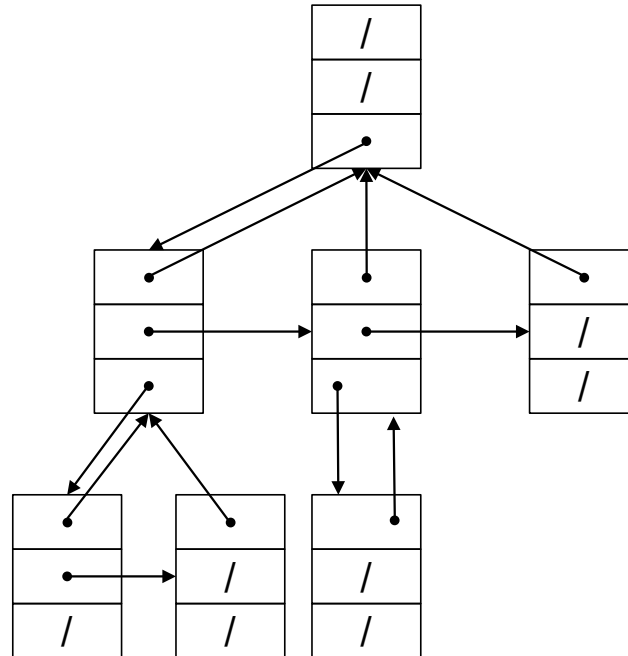
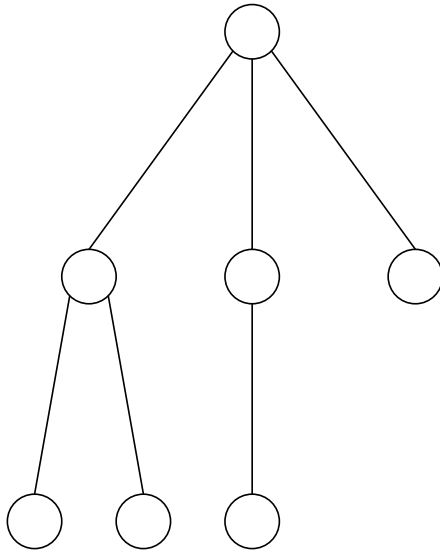
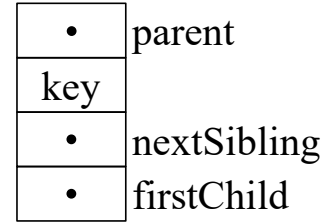
Binäre Bäume

```
class BinaryNode {  
    BinaryNode parent, left, right;  
    int key;  
}
```



Bäume mit unbeschränktem Grad

```
class Node {
    Node parent, firstChild, nextSibling;
    int key;
}
```



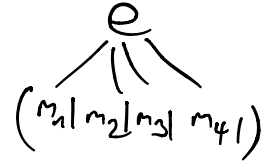
Übersicht

Elementare Datenstrukturen

- Stapel & Warteschlangen
- Verkettete Listen
- Bäume
- Anwendungsbeispiel: kd-Bäume

kd-Bäume

Problem



- Speichere Position einer Menge von Punkten im d -dimensionalen Raum
- Wollen effiziente Abfragen wie
 - Gegeben Abfragepunkt, finde nächsten Nachbarn in der Punktmenge
 - Finde k -nächste Nachbarn
 - Finde alle Nachbarn in einem gewissen Radius
- Naiver Ansatz erlaubt Abfragen in $O(n)$

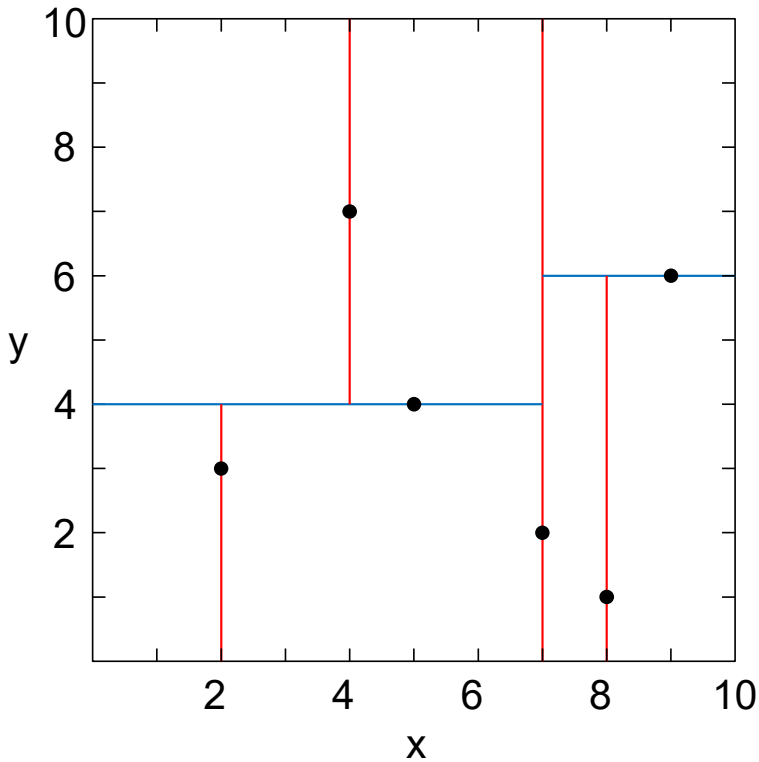
kd-Bäume

- **Räumliche** Datenstruktur
 - Aufbau der Datenstruktur hat mit räumlicher Anordnung der Daten zu tun
- Erlaubt effiziente Abfragen in $O(\lg n)$
- Anwendungen
 - Künstliche Intelligenz, Klassifikation
 - http://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm
 - http://en.wikipedia.org/wiki/Nearest_neighbor_search
 - Computergrafik, Ray Tracing

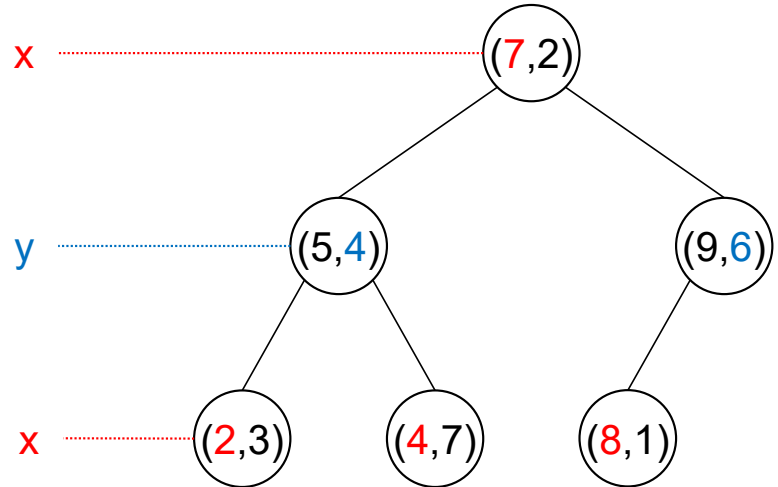
Konstruktion

- kd-Baum teilt Raum rekursiv entlang einer **Split-Ebene** in zwei Halbräume
- Jeder Knoten
 - Speichert einen Punkt
 - Definiert eine Split-Ebene durch den Punkt
- Split-Ebenen
 - Sind immer **achsenparallel**
 - Orientierung der Split-Ebene gegeben durch Tiefe im Baum
 - Alle Punkte im linken Teilbaum liegen „unter“ der Split-Ebene, im rechten Teilbaum „über“ der Split-Ebene
- Siehe auch
<http://en.wikipedia.org/wiki/Kd-tree>

Beispiel



Menge von Punkten in 2D



kd-Baum

Konstruktion

- Balancierter kd-Baum

Teilen mit jeder Split-Ebene die Menge der Punkte gleichmässig auf.

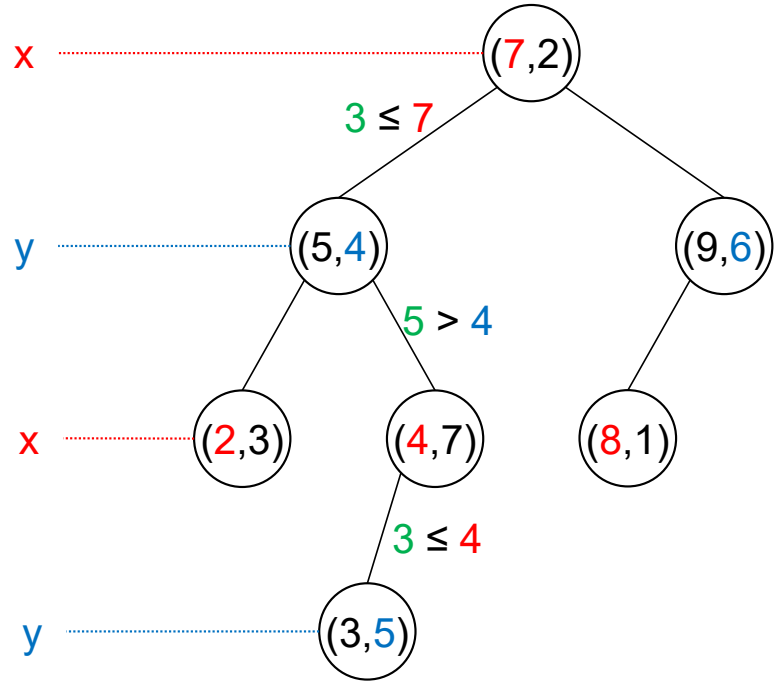
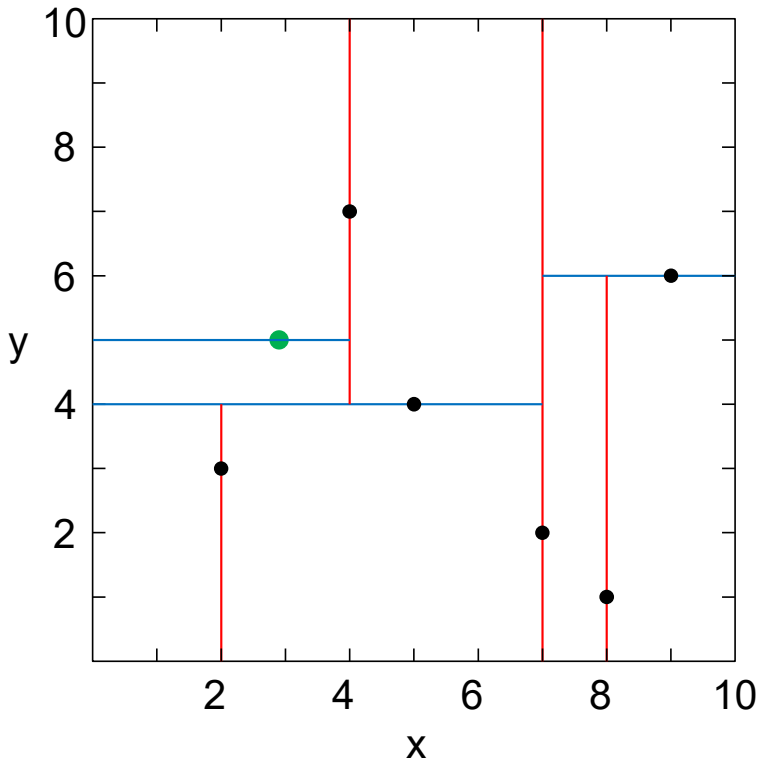
- `axis` ist eine Koordinatenachse (x oder y in 2D)
- `cycle(axis)` rotiert durch die Koordinatenachsen:
 $\text{cycle}(x) = y, \text{cycle}(y) = x$

```
node kdtree (set of points, axis) {  
    if set of points is empty return null;  
    else {  
        select median point along axis  
        split set of points into set below and above median  
        create node with location of median point  
        node.left = kdtree(points below median, cycle(axis))  
        node.right = kdtree(points above median, cycle(axis))  
        return node  
    }  
}
```

Einfügen

- Eingabe: neuer Punkt
- Ausgabe: kd-Baum der neuen Punkt enthält
- Ablauf
 - Traversiere kd-Baum bis zu einem Blatt
 - Bei jedem Knoten, entscheide basierend auf Vergleich von Position von Knoten und neuem Punkt, in welchen Teilbaum gegangen wird
 - Wenn Blatt erreicht, erstelle neuen Knoten, der dem Blatt als Kind angehängt wird

Einfügen

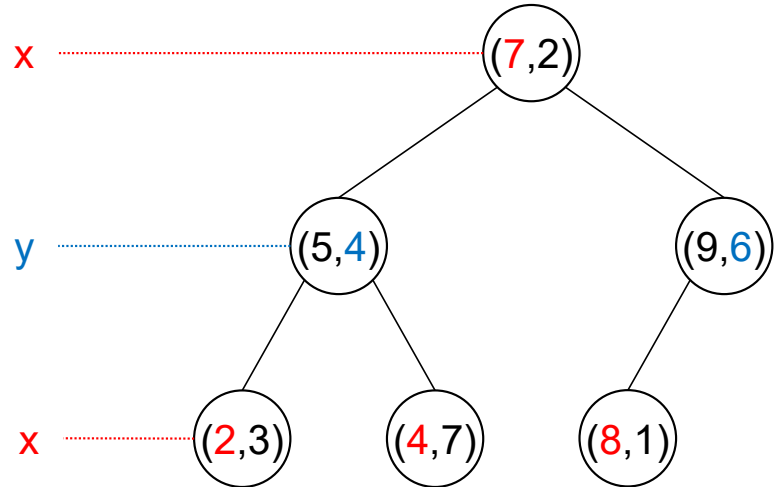
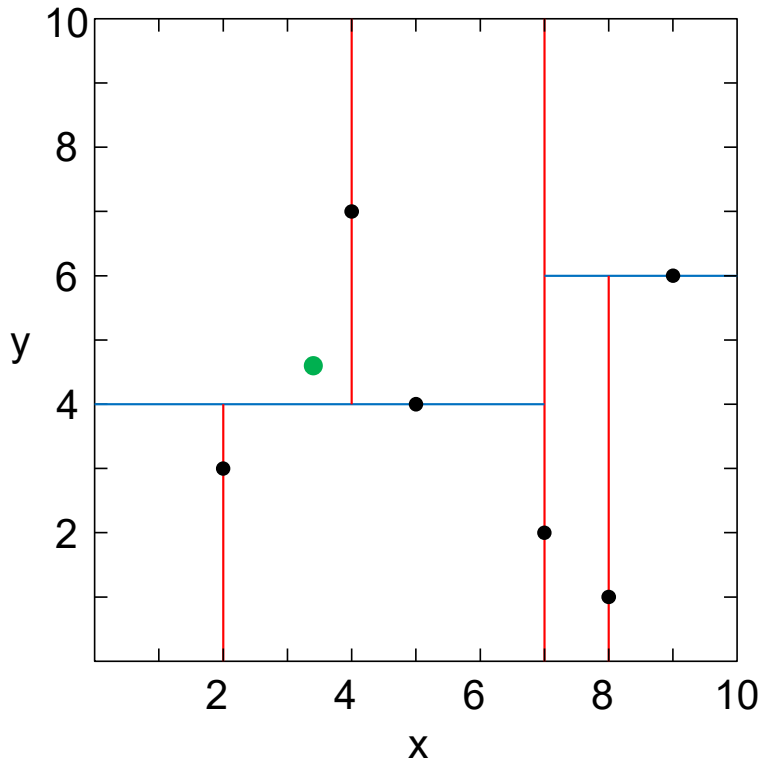


Neuer Punkt (3,5)

Nächster Nachbar

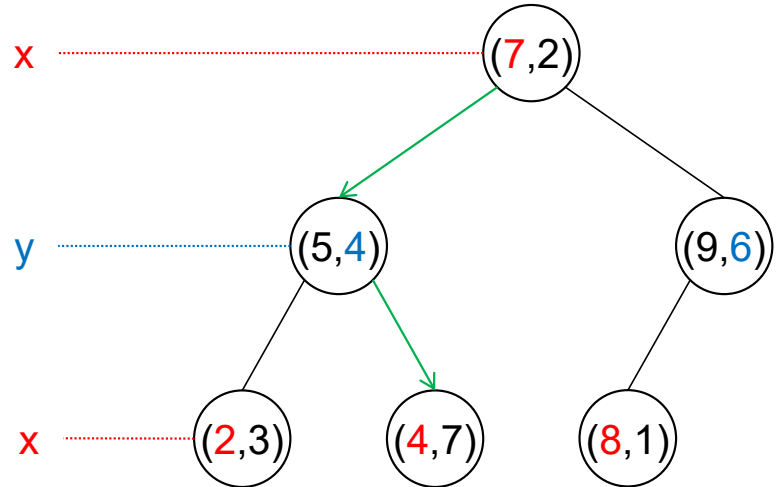
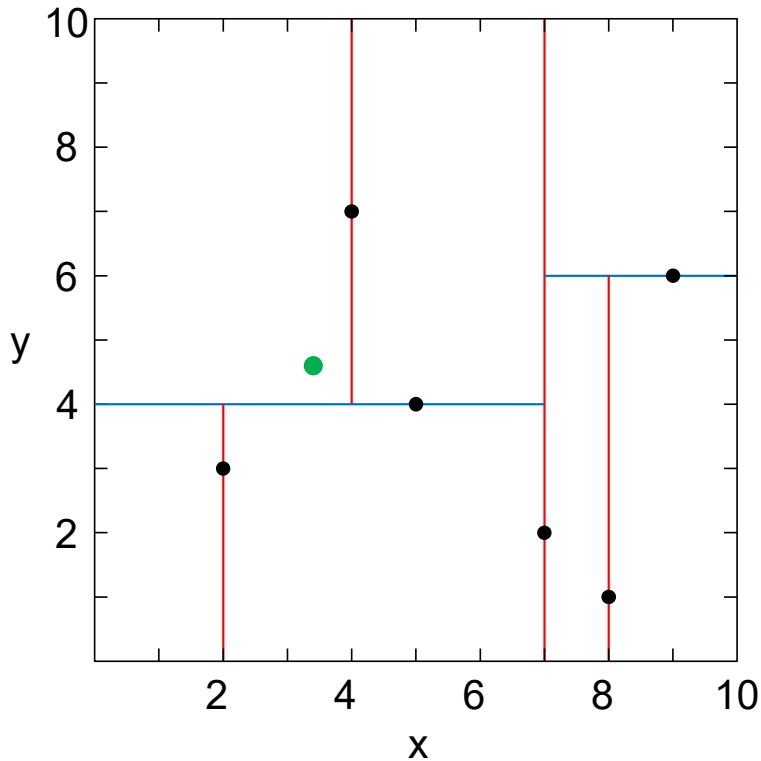
- Input: Abfragepunkt, Punktemenge in kd-Baum
- Output: Punkt im kd-Baum, der am nächsten beim Abfragepunkt liegt
- Ablauf
 - Traversiere kd-Baum bis zum Blatt, in das der Abfragepunkt fällt
 - Blattknoten ist Kandidat für nächster Nachbar
 - Traversiere Baum zurück vom Blatt zur Wurzel; bei jedem Knoten
 - Falls Knoten näher als bisheriger Kandidat, wird neuer Kandidat
 - Falls im anderen Teilbaum näherer Punkt möglich, traversiere anderen Teilbaum

Beispiel



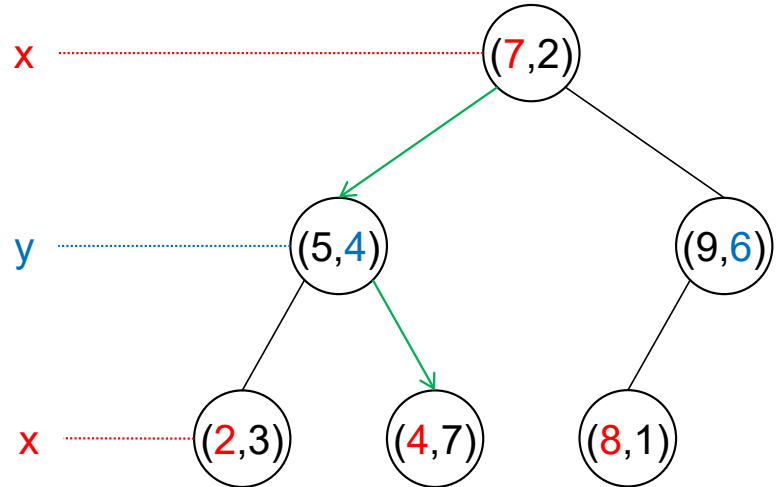
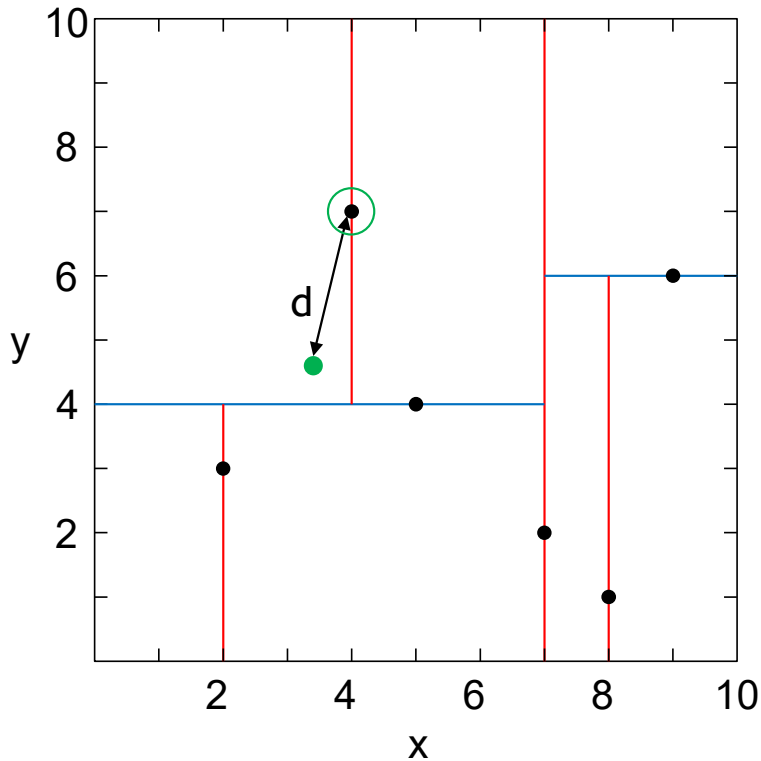
Nächster Nachbar von Abfragepunkt (3.5, 4.5)

Beispiel



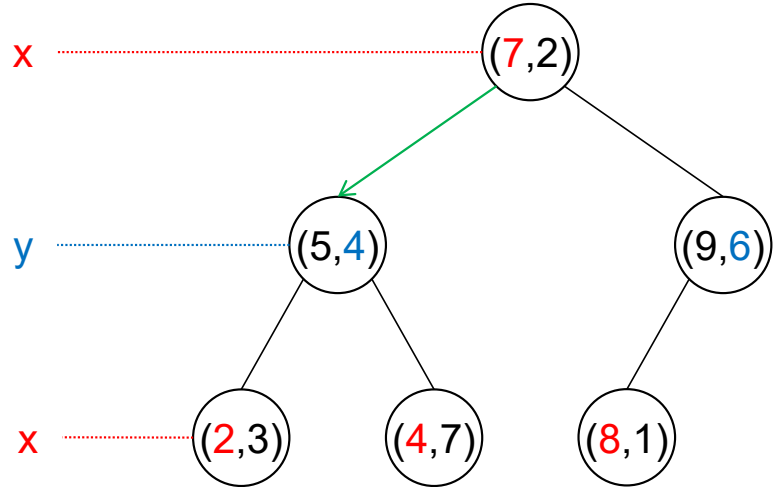
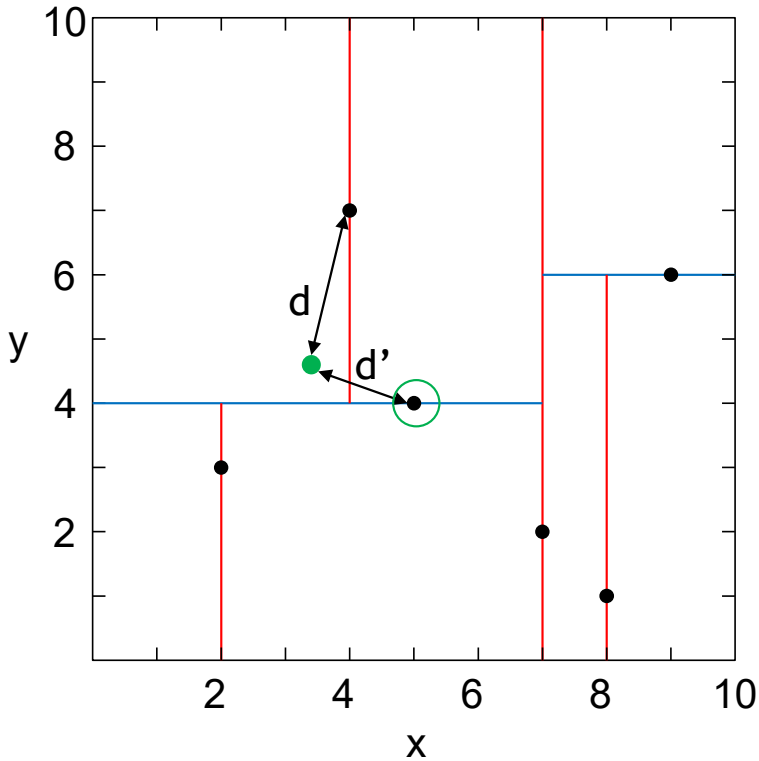
Traversiere bis zum Blatt in das der Abfragepunkt fällt

Beispiel



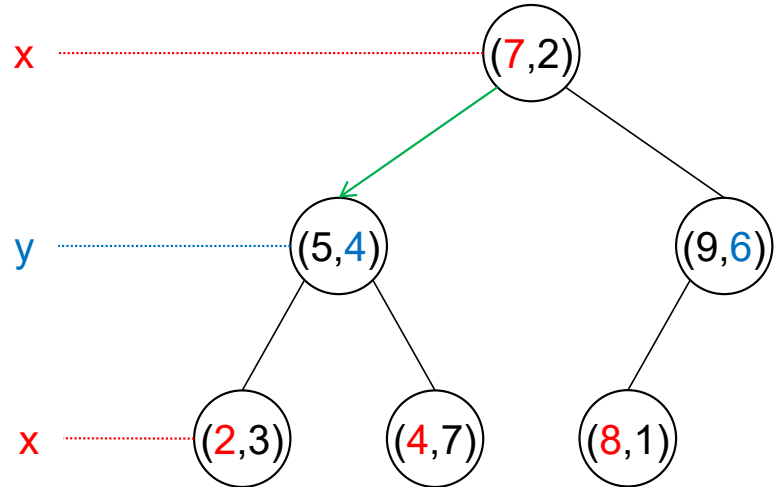
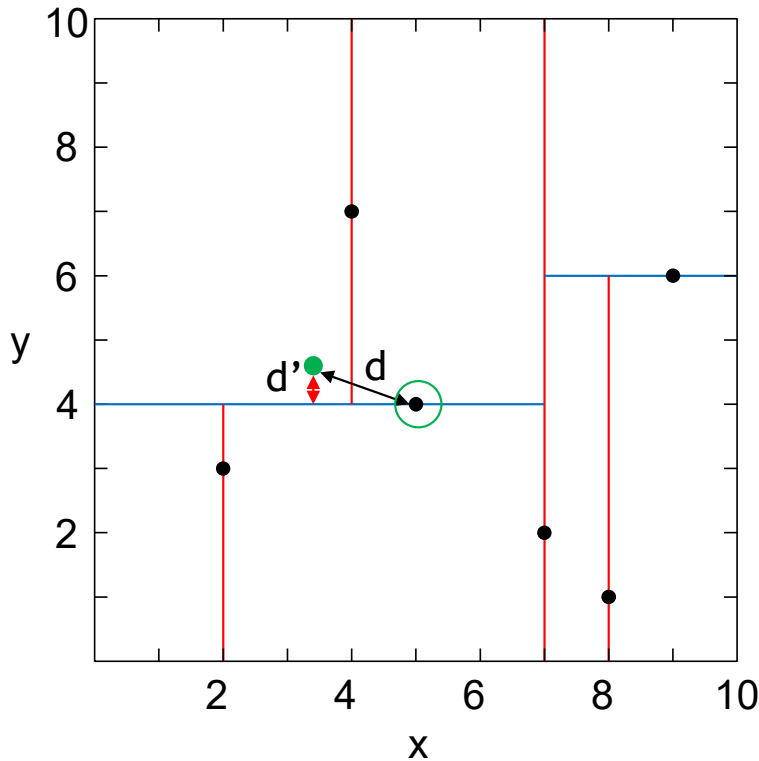
Blattknoten (4,7) ist Kandidat für nächster Nachbar.

Beispiel



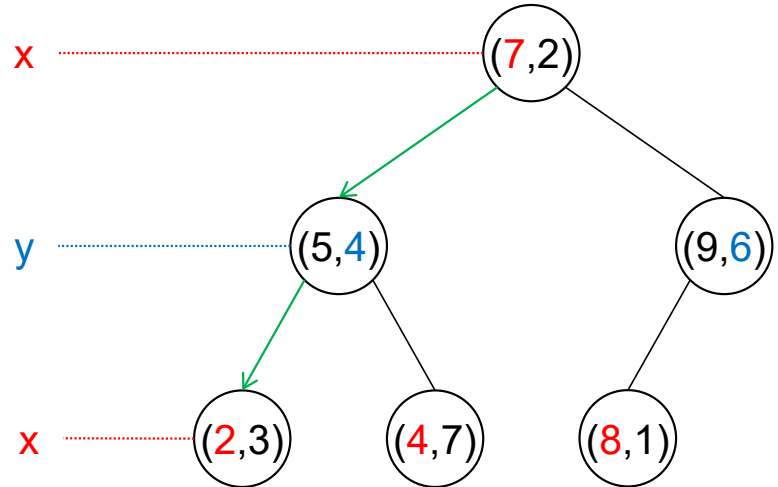
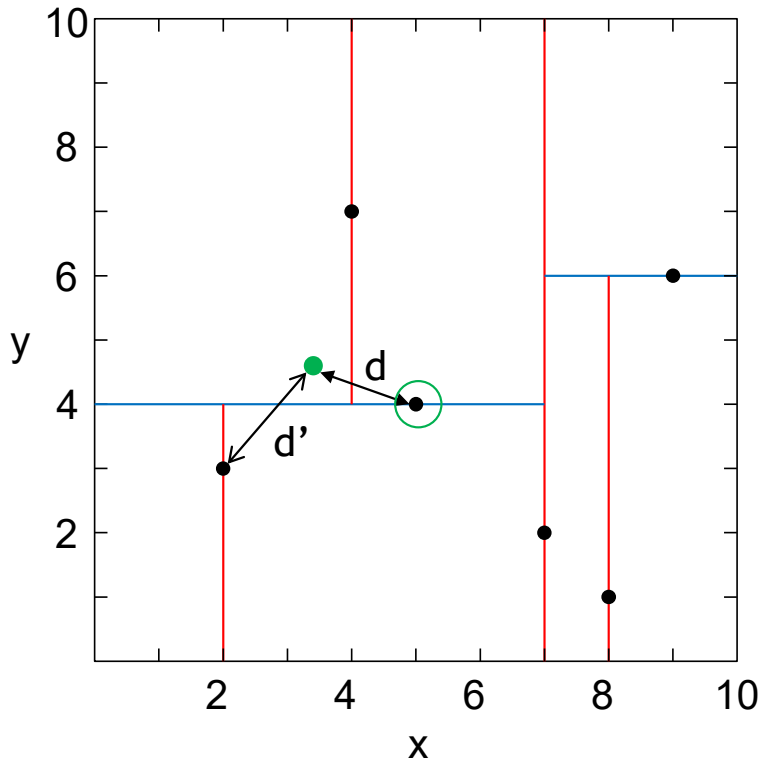
Traversiere zurück. Punkt (5,4) ist Kandidat für nächster Nachbar. Liegt näher als bisheriger Kandidat.
→ wähle Punkt (5,4) als neuen Kandidaten.

Beispiel



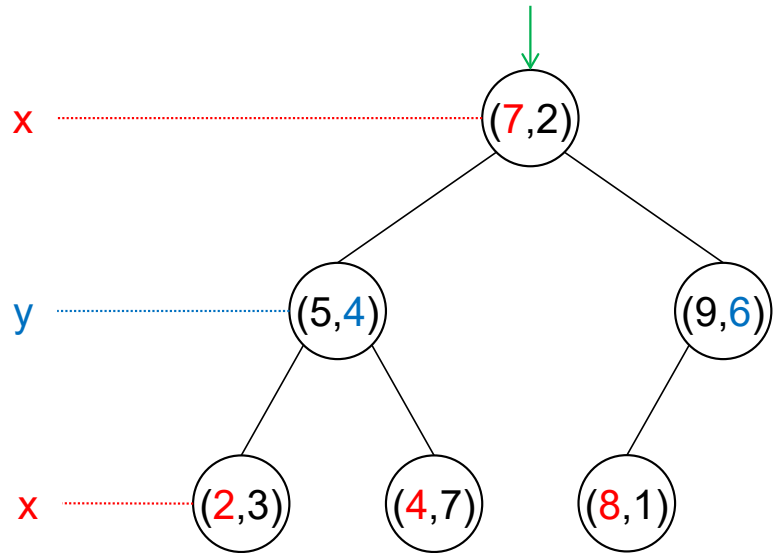
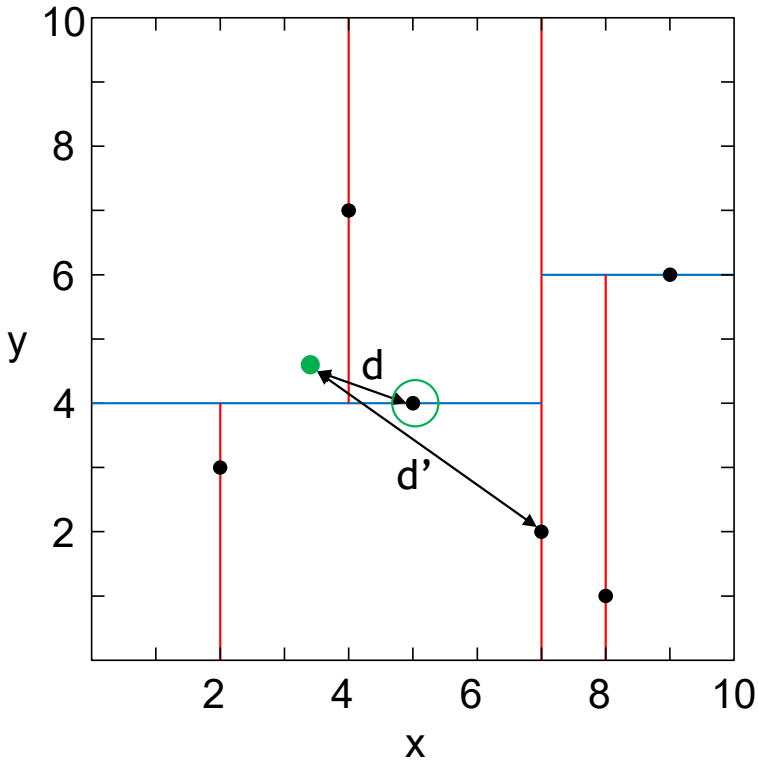
Prüfe, ob im anderen Teilbaum nähere Punkte möglich sind. Abstand d' zu unterem Quadranten ist kleiner als Abstand d zum aktuellen Kandidaten \rightarrow traversiere linken Teilbaum.

Beispiel



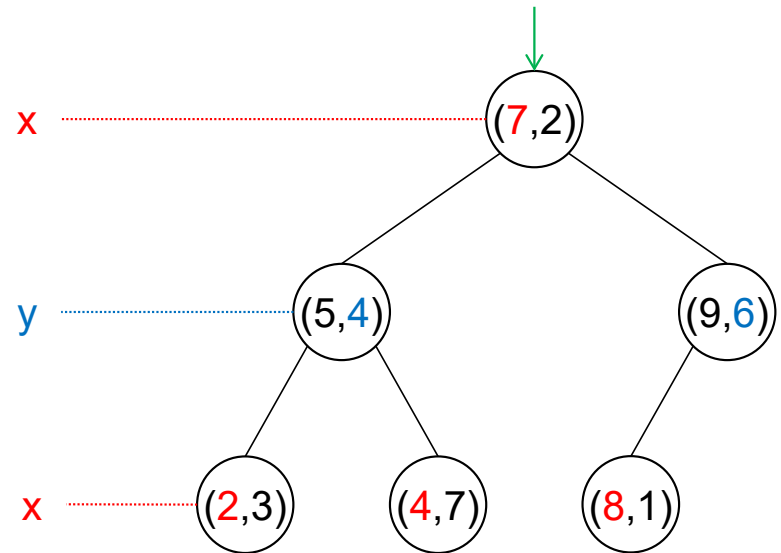
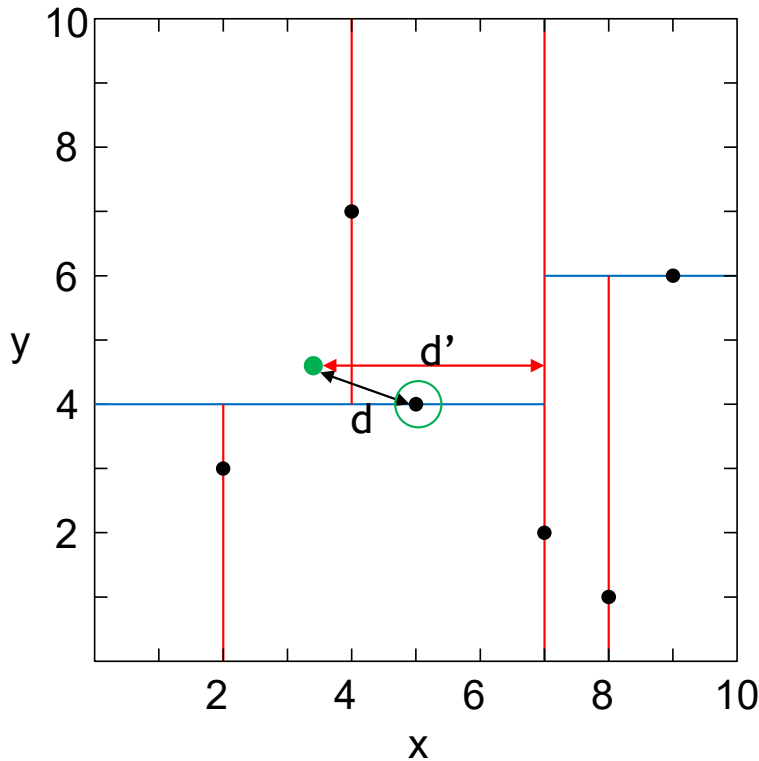
Punkt $(2,3)$ ist Kandidat für nächster Nachbar.
Liegt weiter weg als bisheriger Kandidat ($d < d'$).
Behalte bisherigen Kandidaten.

Beispiel



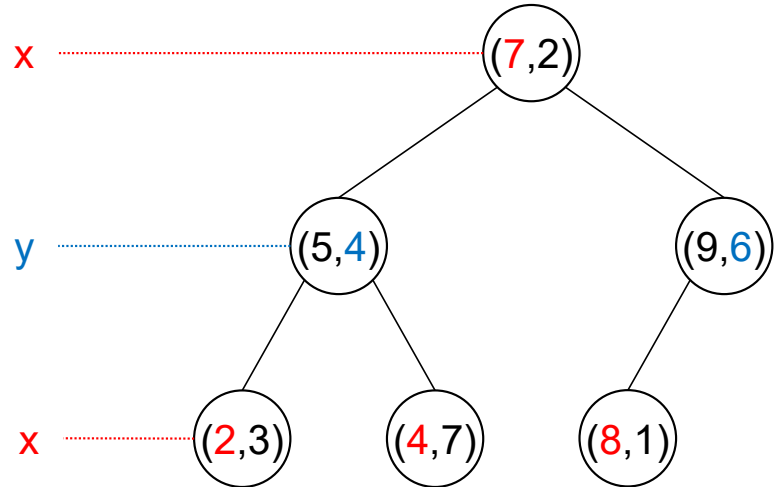
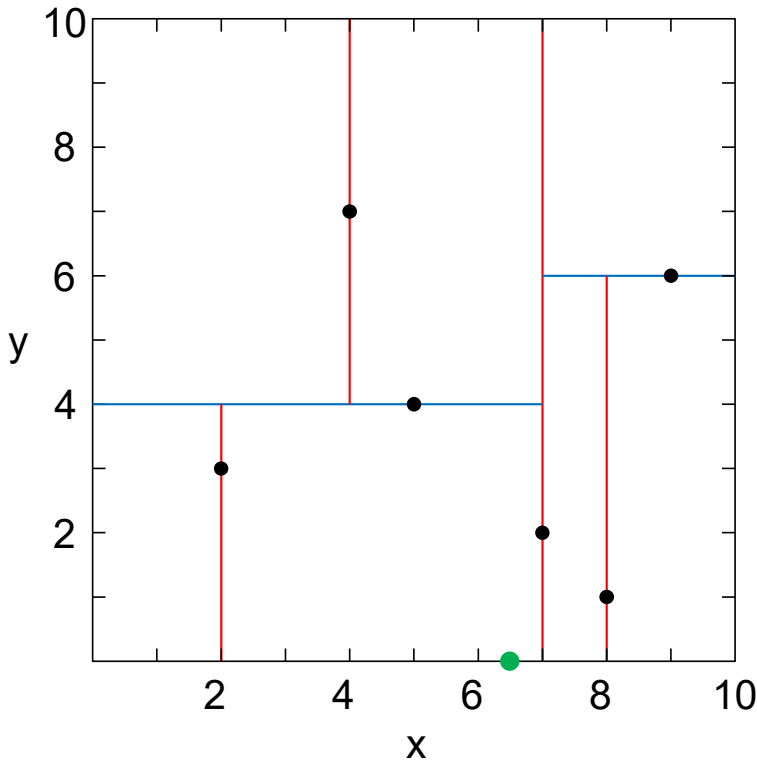
Traversiere zurück. Punkt (7,2) ist Kandidat für nächster Nachbar. Liegt weiter weg als bisheriger Kandidat. Behalte bisherigen Kandidaten.

Beispiel



Prüfe, ob im anderen Teilbaum nähere Punkte möglich sind. Abstand d' zu rechtem Quadranten ist grösser als Abstand d zum aktuellen Kandidaten. Fertig, $(5,4)$ ist nächster Nachbar.

Beispiel



Nächster Nachbar von Abfragepunkt (6.5, 0).
Ganzer Baum muss durchsucht werden.
Selber durchspielen.

Nächstes Mal

- Kapitel 11: Hashtabellen