1 Datenstrukturen

1.1 Abstrakte Datentypen

1.2 Lineare Strukturen

1.3 Bäume

1.4 Prioritätsschlangen

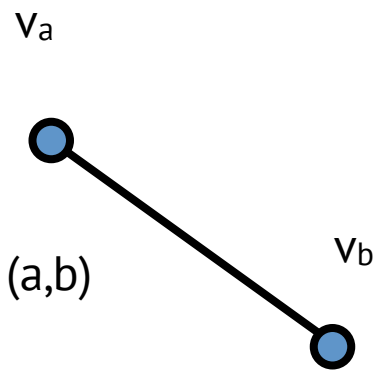
1.5 Graphen

1.3 Bäume

- Hierarchische Datenstruktur
  - Zusammenfassung von Gruppen (z.B. Bund / Länder / Gemeinden)
  - Eindeutige Schachtelung
- Typische Anwendungen
  - Such- / Entscheidungsprobleme
  - Strukturierte Aufzählung
  - Komplexitätsreduktion

Definitionen

- Menge von Knoten $V = \{v_1, \ldots, v_n\}$
  ($v_i \in W$ … beliebiger Datentyp)
- Menge von Kanten $E = \{(a_1,b_1), \ldots (a_m,b_m)\}$
  ($a_i,b_i \in N$ … Knotenindizes)
- Falls $(a,b) \in E$, dann ist
  - v_a Vorgänger von v_b
  - v_b Nachfolger von v_a



Definitionen

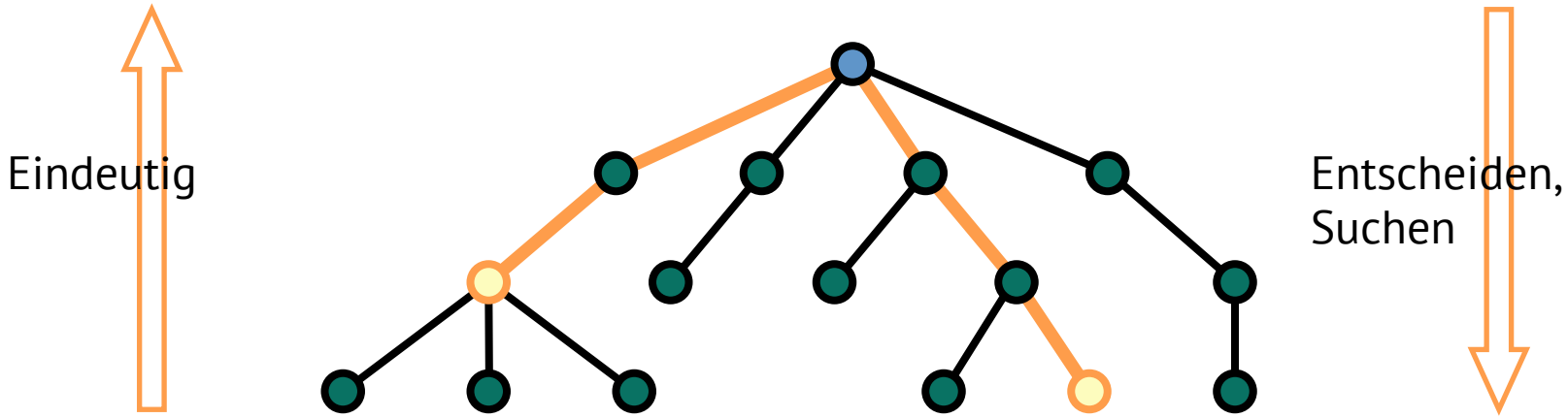
- Eine Folge von Kanten $(i_1,i_2),(i_2,i_3),..,(i_{k1},i_k)$ heißt Pfad von v_{i1} nach v_{ik}
- Für $i_1 = i_k$ ist dieser Pfad ein Zyklus

Definitionen

- Ein Baum B=(V,E) besteht aus einer Menge von Knoten V und einer Menge von Kanten E, so dass gilt:
  - Es gibt keine Zyklen
  - Jeder normale Knoten hat genau einen Vorgänger
  - Es existiert genau ein Wurzelknoten, der keinen Vorgänger hat

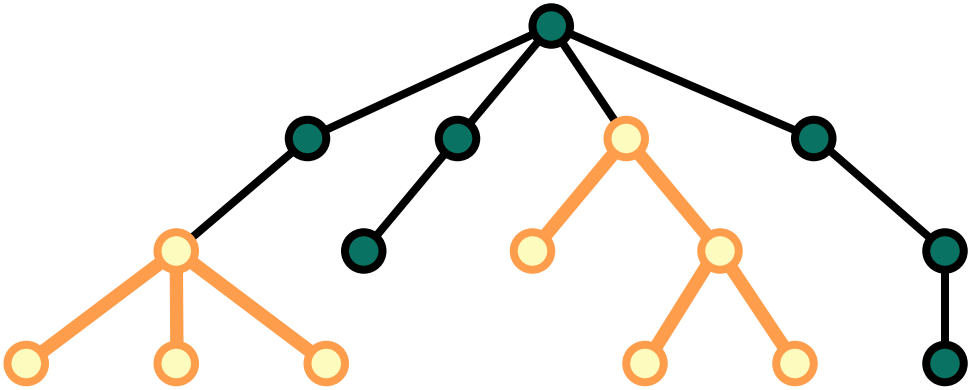
Abgeleitete Eigenschaften

- Für jeden Knoten existiert ein eindeutiger Pfad, der ihn mit dem Wurzelknoten verbindet.



Abgeleitete Eigenschaften

- Jeder Knoten ist Wurzelknoten eines zugehörigen Sub-Baumes (wird später zur Definition des ADT verwendet).



Weitere Begriffe

- Die Tiefe eines Knotens ist gleich der Länge des zugehörigen Pfades von der Wurzel.
- Die Höhe eines Baumes ist gleich der Tiefe des tiefsten Knotens.
- Der Grad eines Knotens ist die Anzahl seiner Nachfolger.

Weitere Begriffe

- Der eine Knoten ohne Vorgänger heißt Wurzelknoten
- Knoten ohne Nachfolger heißen Blätter
- Knoten mit Nachfolgern heißen innere Knoten

Weitere Begriffe

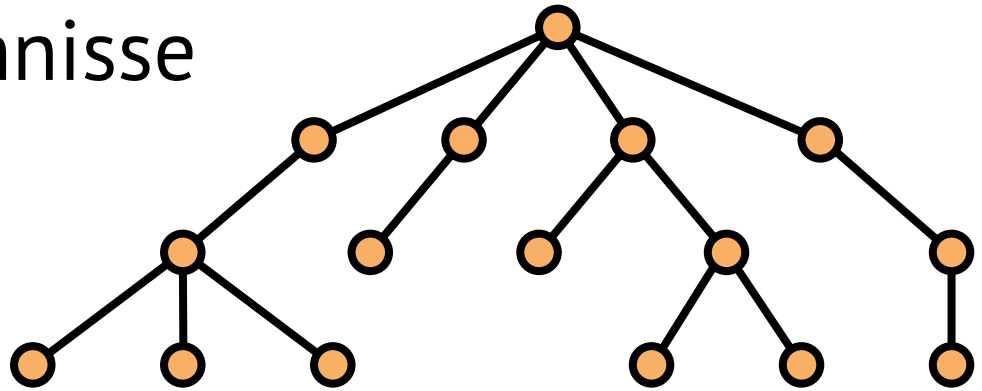
- Seien $w_1, \ldots, w_k$ die Nachfolger des Knotens v, dann gilt für alle w_i:
  $height(subtree(w_i)) \leqslant height(subtree(v))-1$
- Gilt außerdem für alle w_i:
  $height(subtree(w_i)) \geqslant height(subtree(v))-2$
- dann heißt der Baum balanciert.

Weitere Begriffe

- Gilt für alle Knoten v mit Nachfolgern $w_1,\ldots,w_k$ eines Baumes: $height(subtree(w_i)) = height(subtree(v))-1$ so heißt er vollständig.
- Bei vollständigen Bäumen ist allerdings erlaubt, dass für $height(subtree(v)) = 1$ die Sub-Bäume $subtree(w_i)$ entweder die Höhe 0 haben oder leer sind.

Darstellung von Bäumen

- Graph
- Klammerung
- Mengen
- Strukturierte Inhaltsverzeichnisse



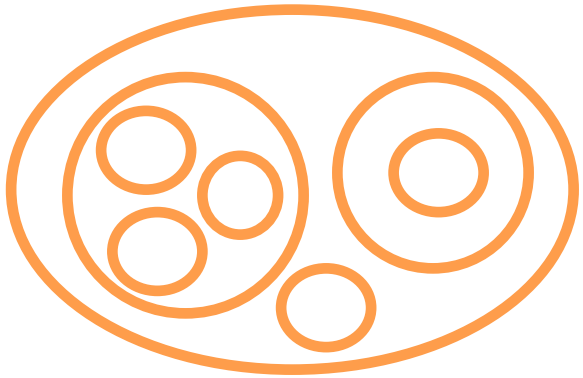
Darstellung von Bäumen

- Graph
- Klammerung
- Mengen
- Strukturierte Inhaltsverzeichnisse

A(B(C,D),E(F,G))

Darstellung von Bäumen

- Graph
- Klammerung
- Mengen
- Strukturierte Inhaltsverzeichnisse



Darstellung von Bäumen

- Graph
- Klammerung
- Mengen
- Strukturierte Inhaltsverzeichnisse

1) A
   1.1) B
   1.2) C
2) D
   2.1) E
      2.1.1) F

Konventionen

- In den meisten Anwendungen betrachten wir nur Bäume mit beschränktem oder konstantem Knotengrad (z.B. Binärbäume).
- Wie bei Listen benötigen wir einen Marker, der anzeigt, wo die nächste Operation angewendet werden soll. In der Regel ergibt sich dieser aus dem Algorithmus.

Datentyp: Binärbaum

- Knotentyp … beliebig
- Create : → Tree
- Node : Tree × Val × Tree → Tree
- Left : Tree → Tree
- Right : Tree → Tree
- Value : Tree → Val
- Empty : Tree → Bool

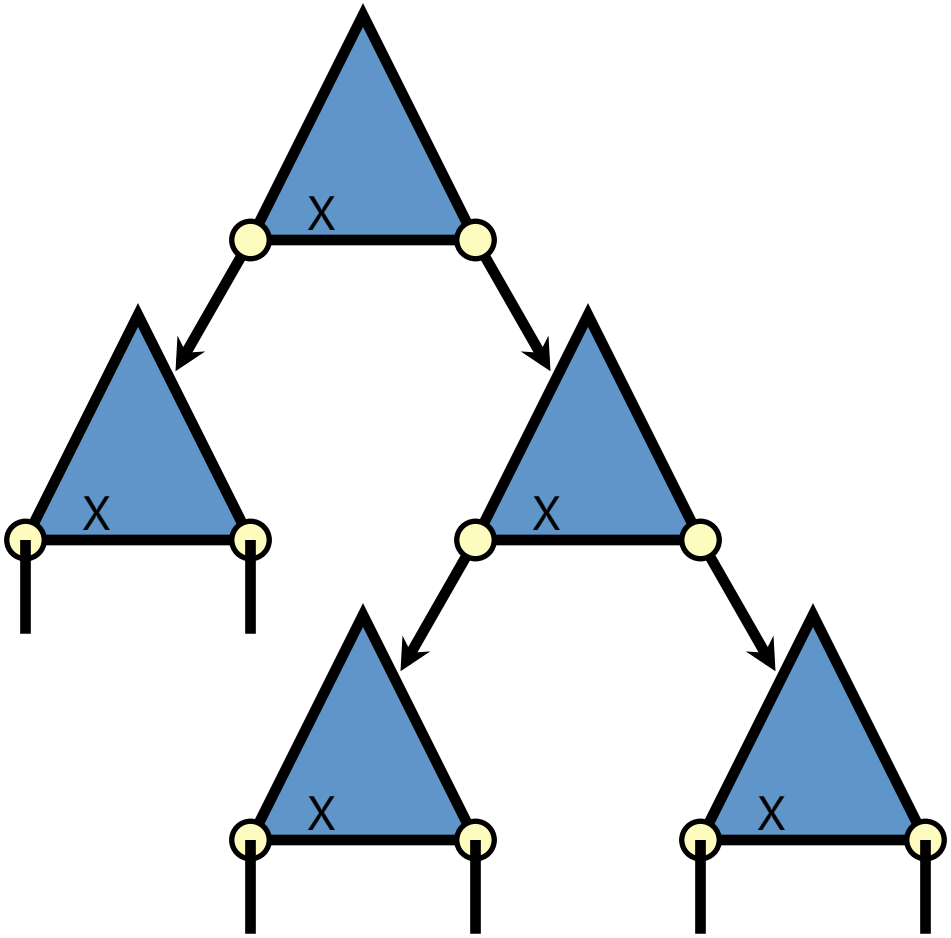
Datentyp: Binärbaum

- Empty(Create()) = true
- Empty(Node(L,V,R)) = false
- Left(Node(L,V,R)) = L
- Right(Node(L,V,R)) = R
- Value(Node(L,V,R)) = V

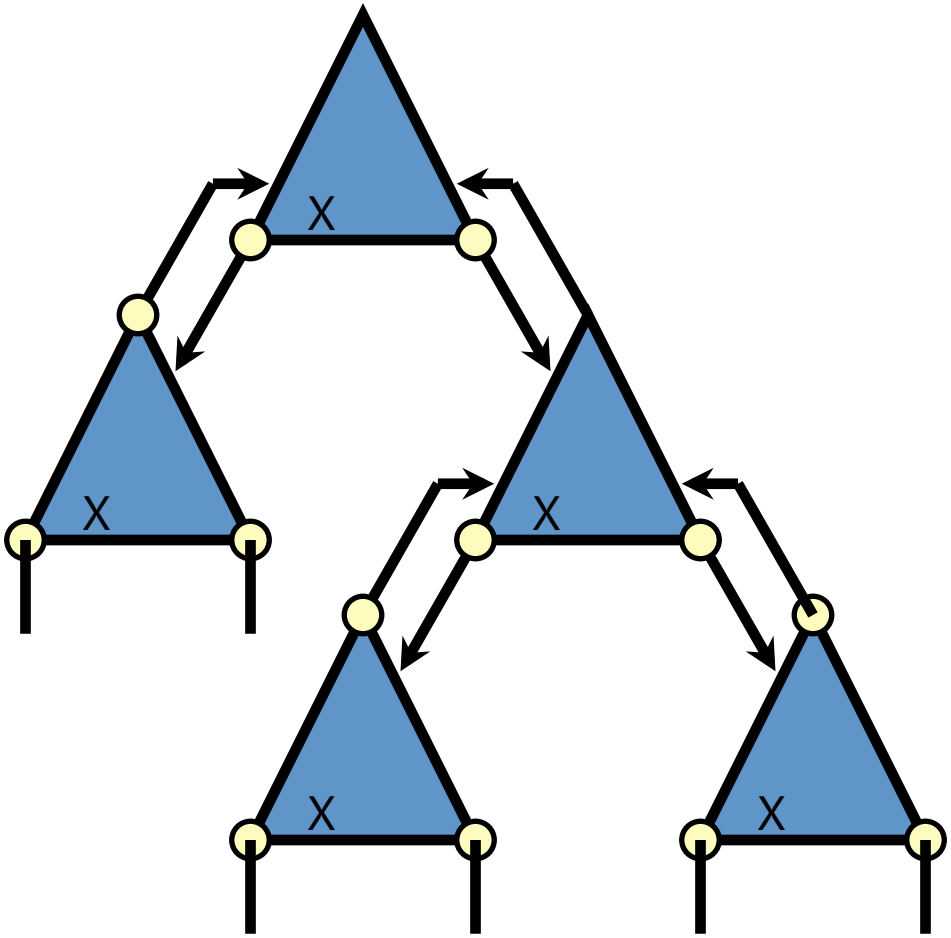
Implementierung

- Pointer (einfach / doppelt)
- Anchor / Sentinel
- Knotengrade (fest / variabel)
- Array-Implementierung für vollständige Bäume

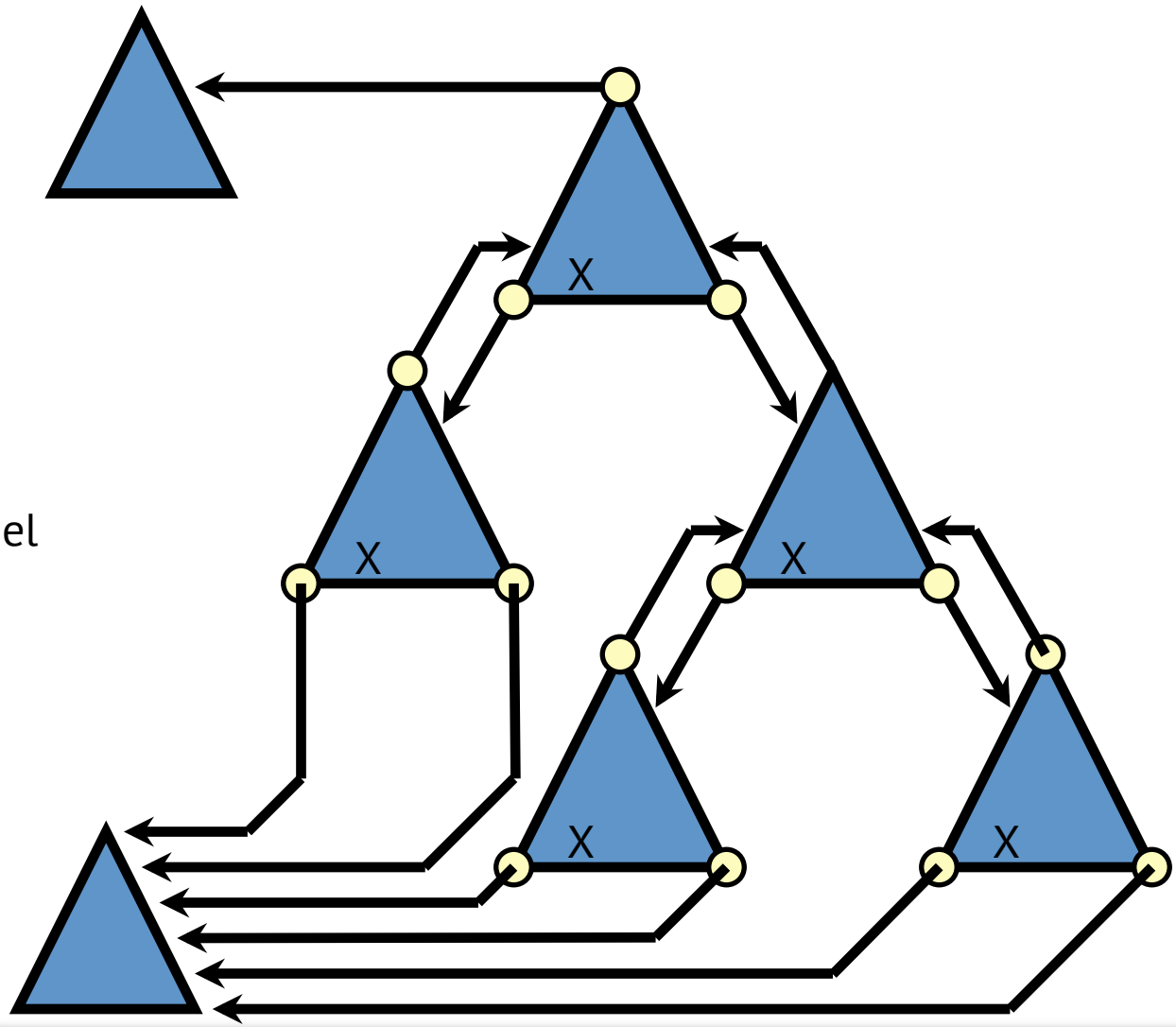
Implementierung



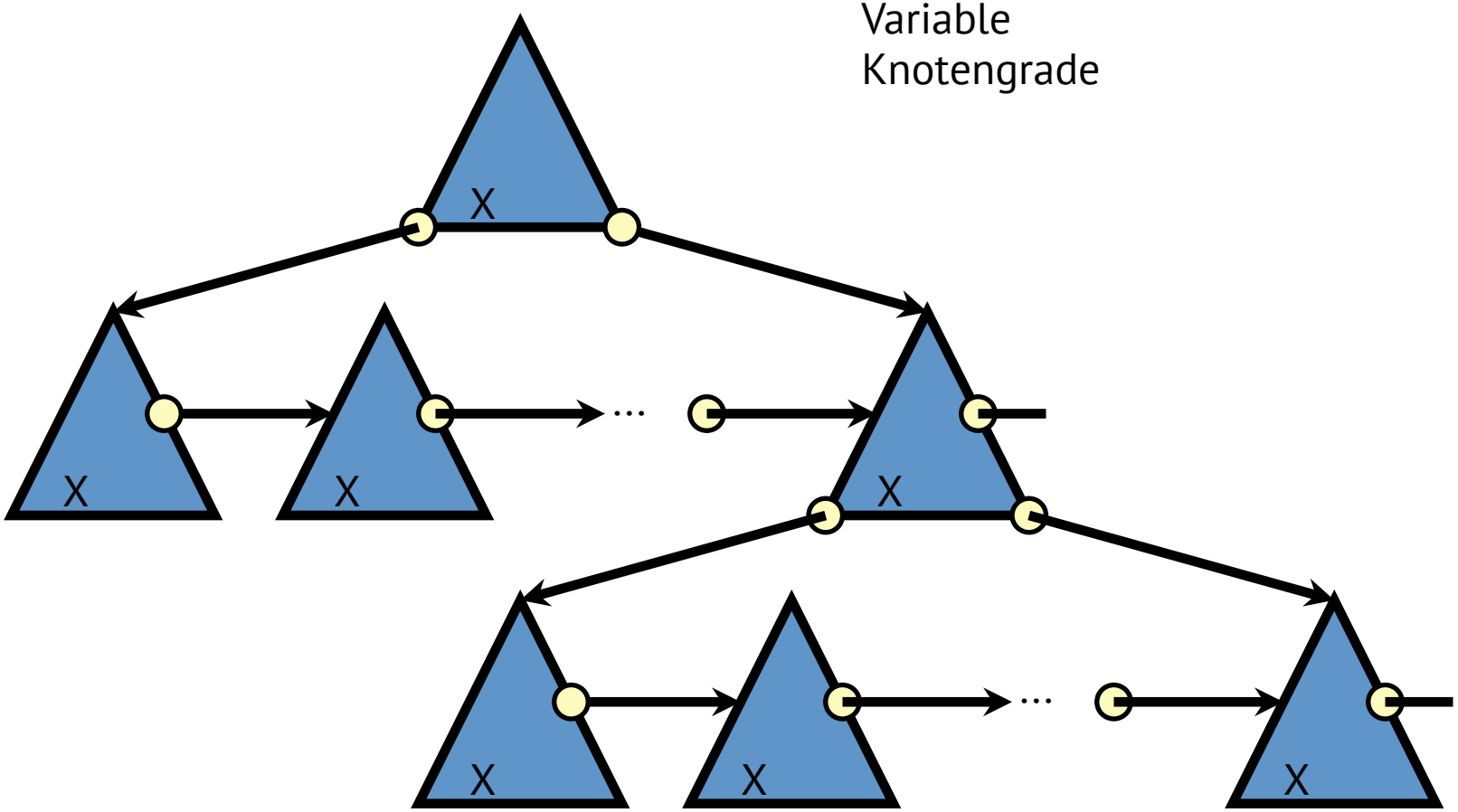
Implementierung



Implementierung



Implementierung

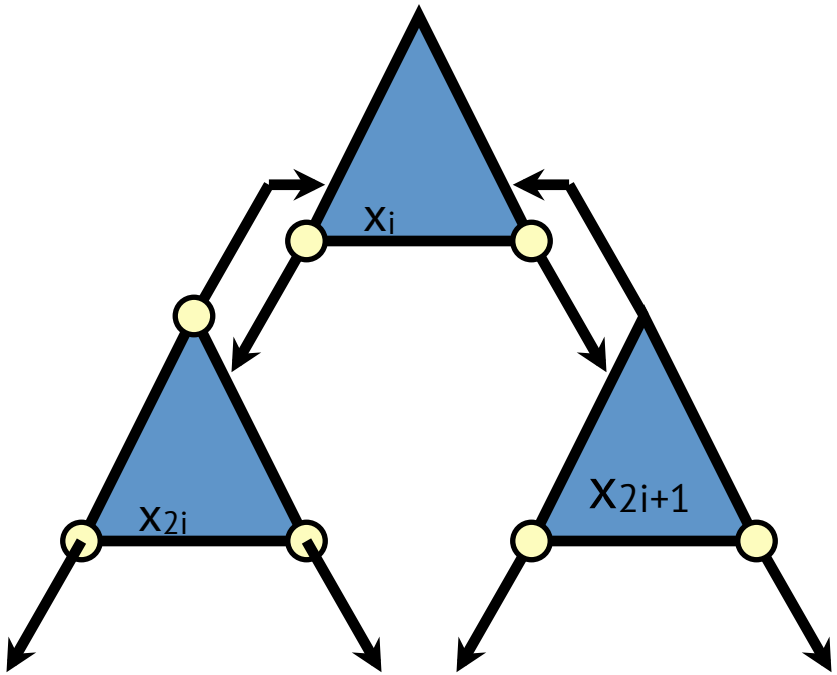


Array-Implementierung

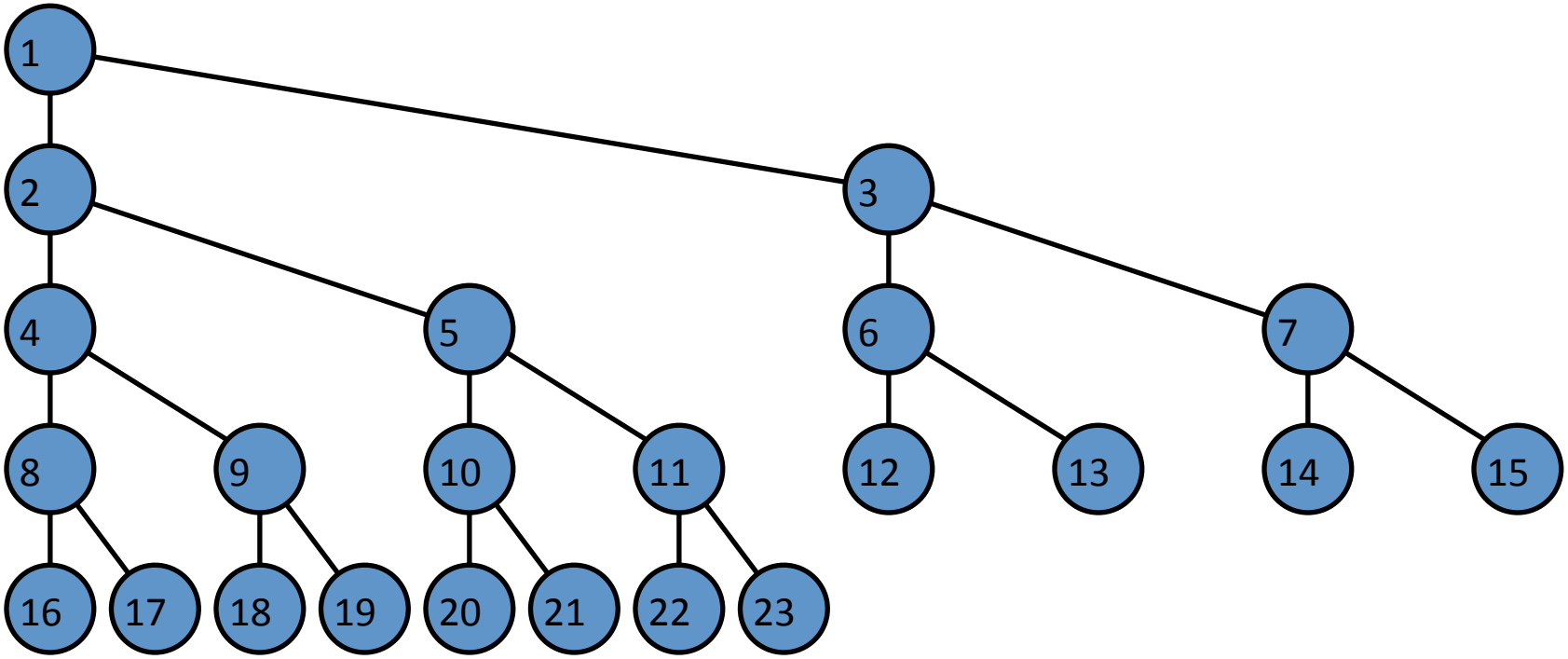
- Vollständige Bäume (z.B. Binärbäume)
- $N(k)$ = Anzahl der Knoten der Tiefe k
- $N(k) = 2 \times N(k-1) \rightarrow N(k) = 2^k$
- Speichere die Knoten der Tiefe k in den Array-Einträgen $A[2^k..2^{k+1}-1]$
- Jeder Knoten $A[i]$ findet seine Nachfolger in $A[2i]$ und $A[2i+1]$

Array-Implementierung

...	x_i	...	...	x_{2i}	x_{2i+1}	...



Array-Implementierung

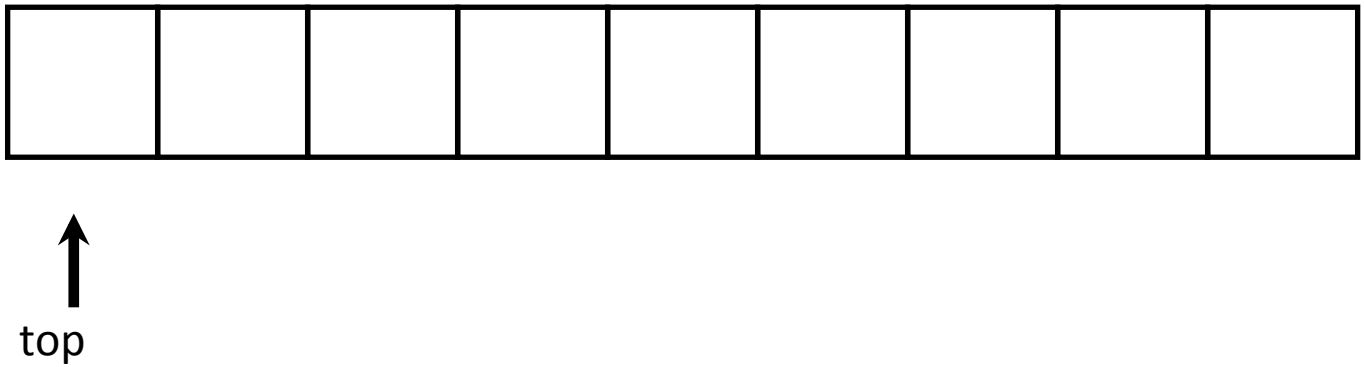


Beispiel: Arithmetische Terme

- „normale“ (Infix) Notation:
  A + B × (C + D) ÷ E
- Postfix Notation:
  A B C D + × E ÷ +
- Präfix Notation:
  + A ÷ × B + C D E
  (Polnische Notation)
- Vorteil: keine Klammern notwendig!

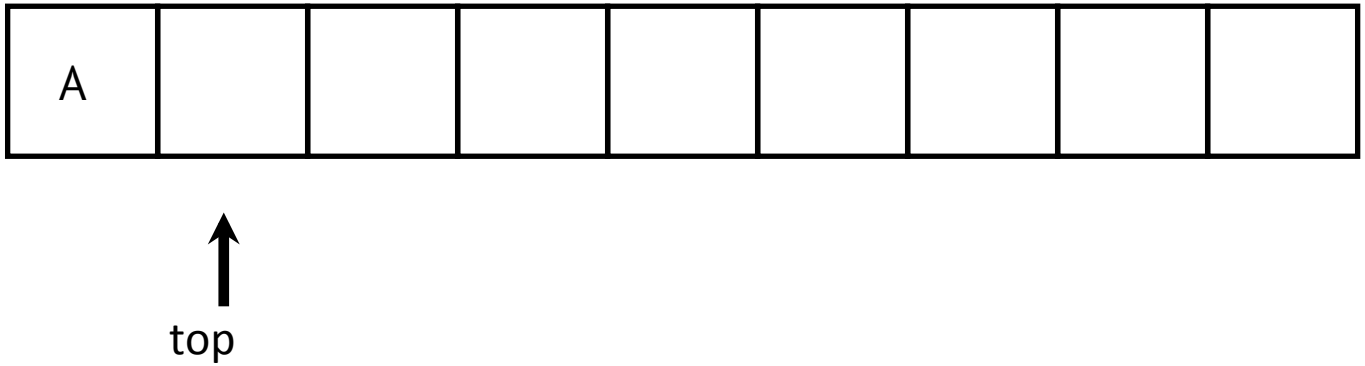
Stack-Computer

A B C D + * E / +



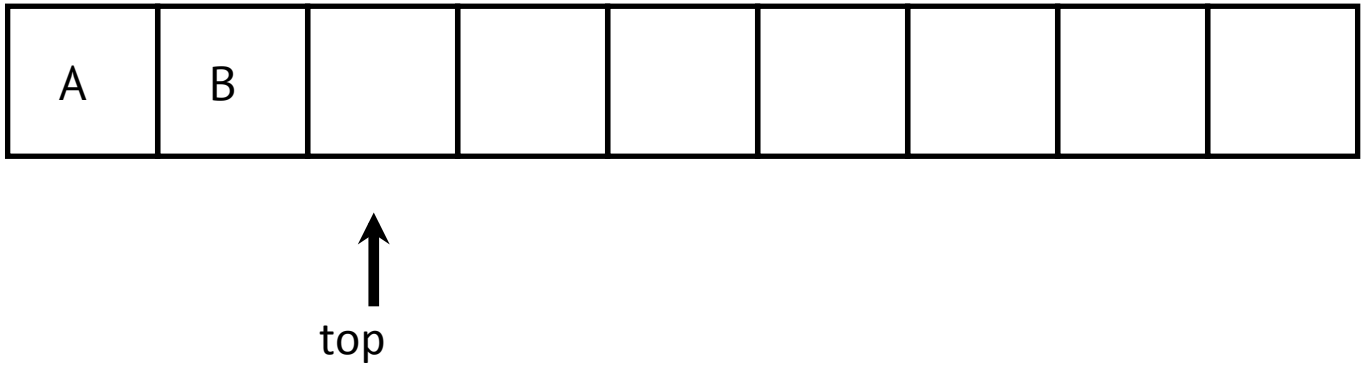
Stack-Computer

A B C D + * E / +



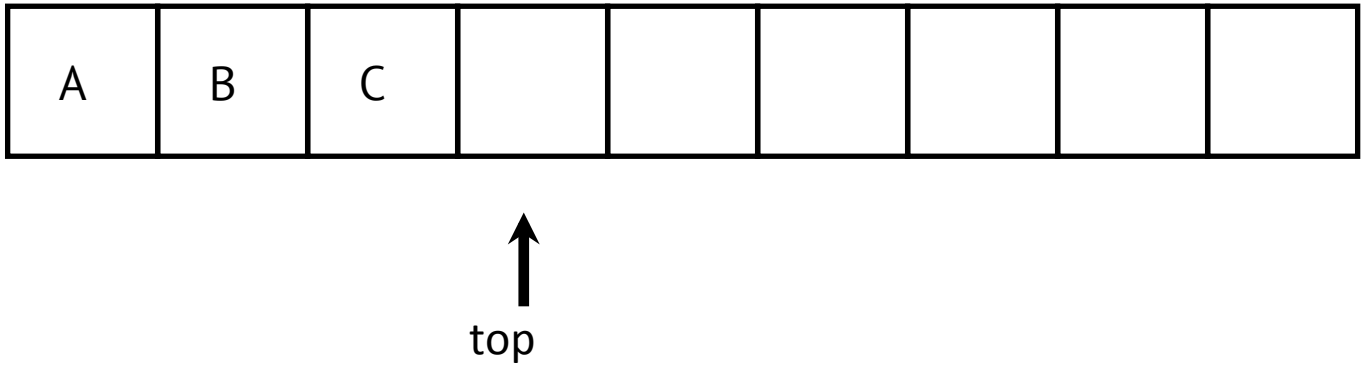
Stack-Computer

AB C D + * E / +



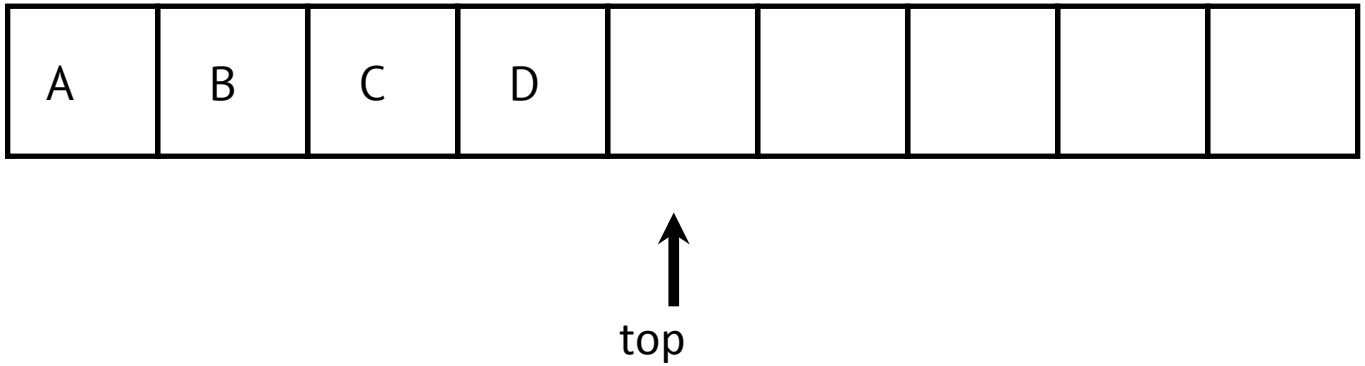
Stack-Computer

ABC D + * E / +



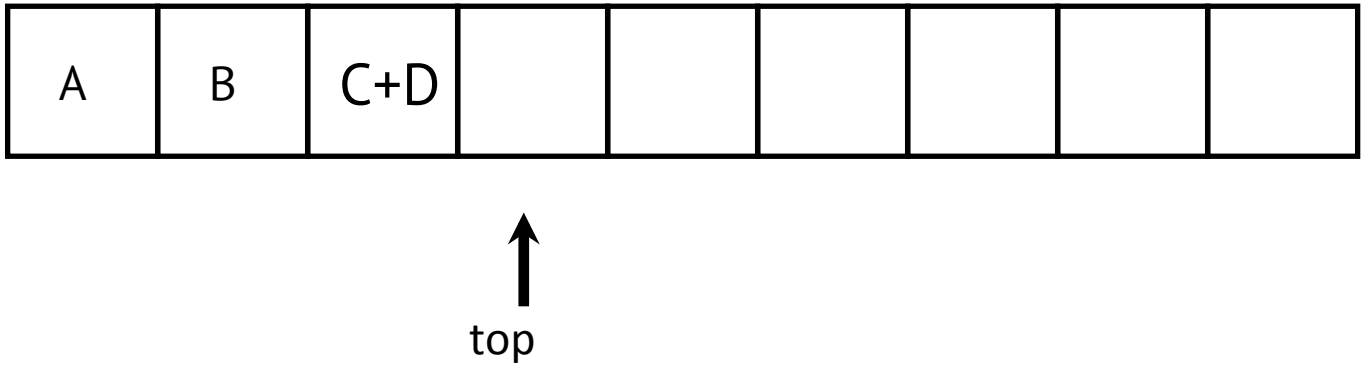
Stack-Computer

ABCD + * E / +



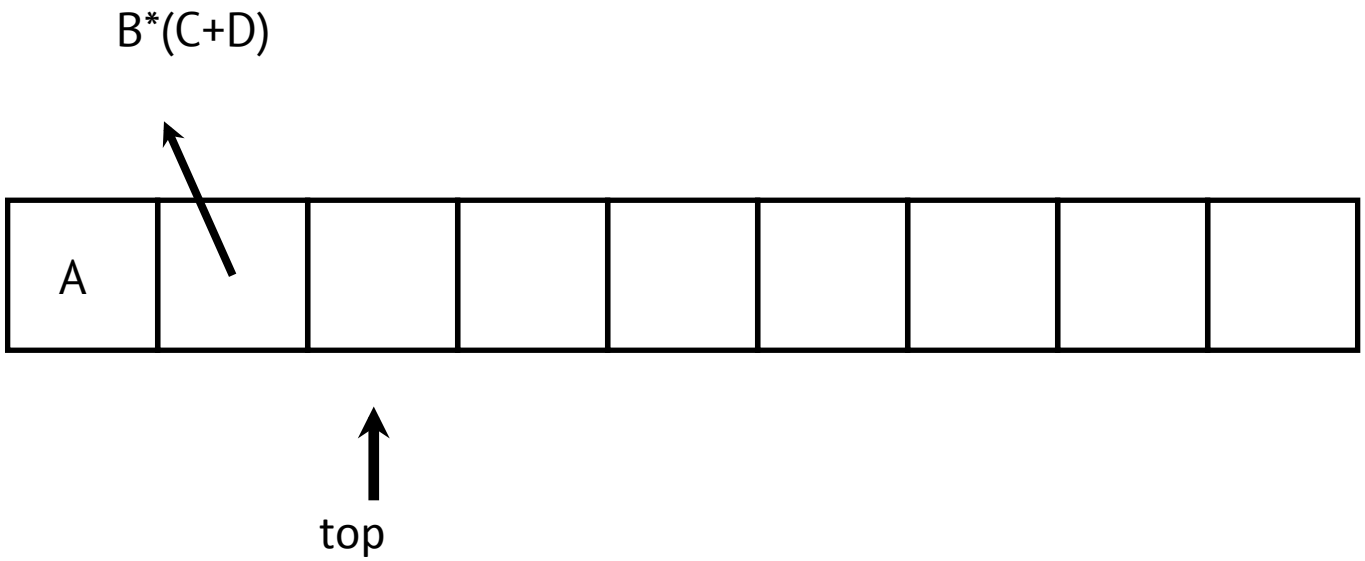
Stack-Computer

ABCD+ * E / +



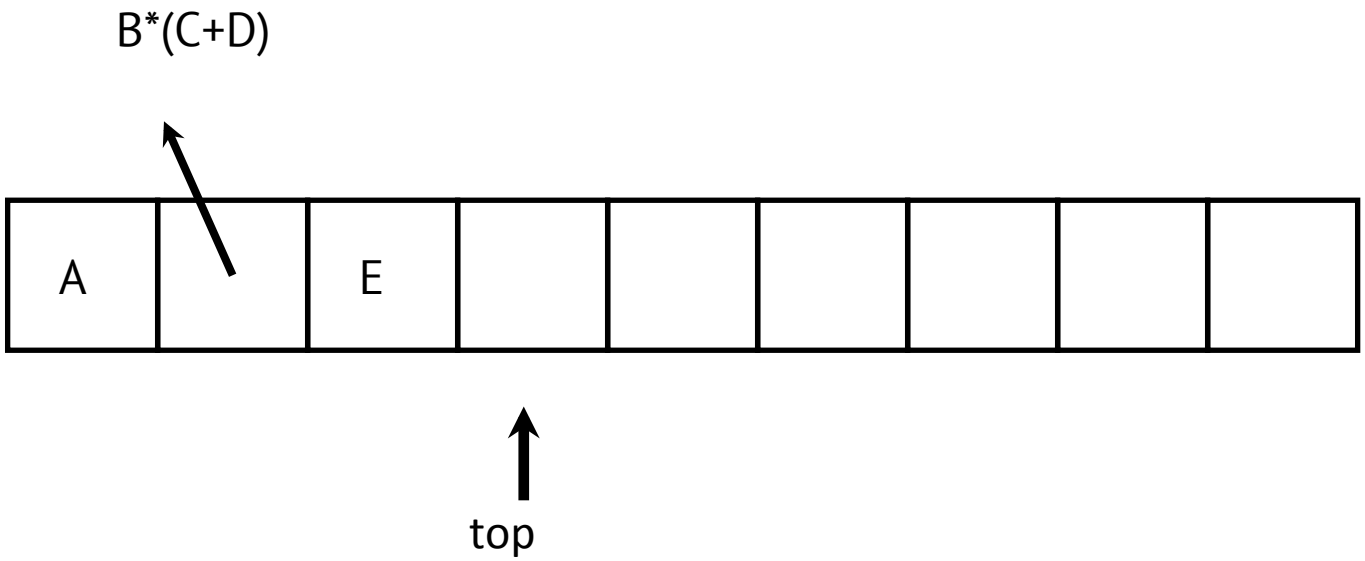
Stack-Computer

ABCD+* E / +



Stack-Computer

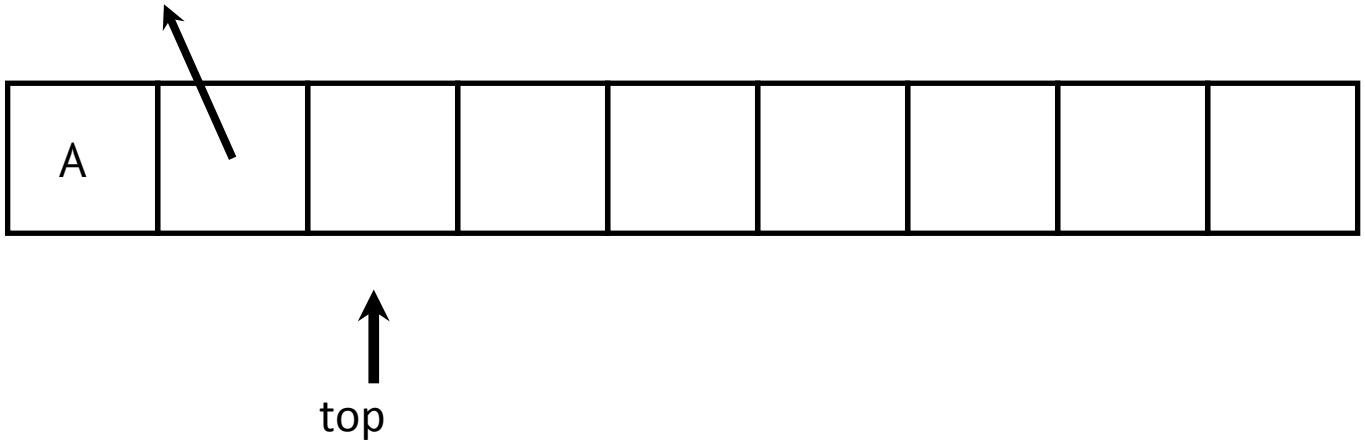
ABCD+*E / +



Stack-Computer

ABCD+*E/ +

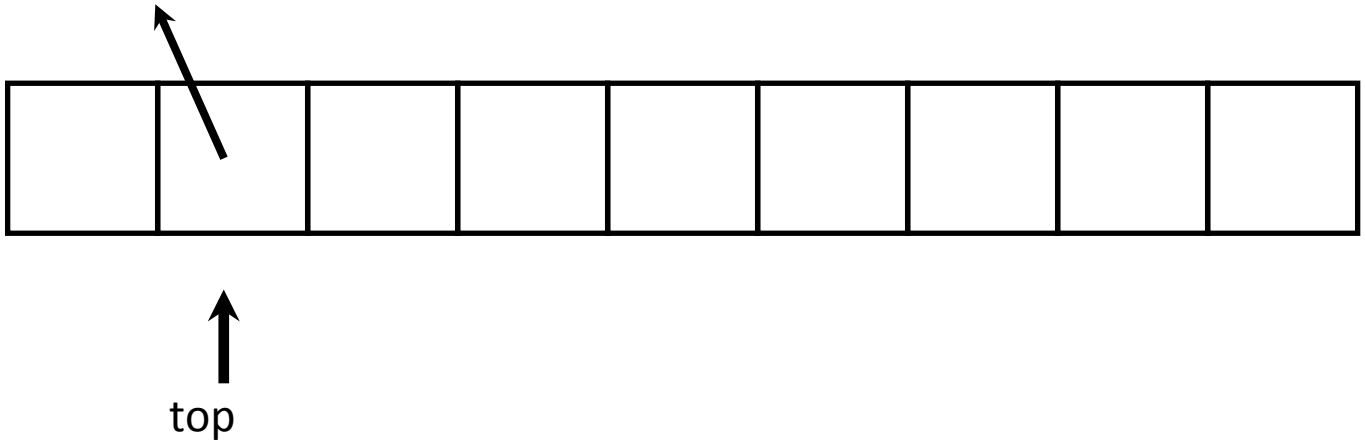
(B*(C+D))/E



Stack-Computer

ABCD+*E/+

A+(B*(C+D))/E

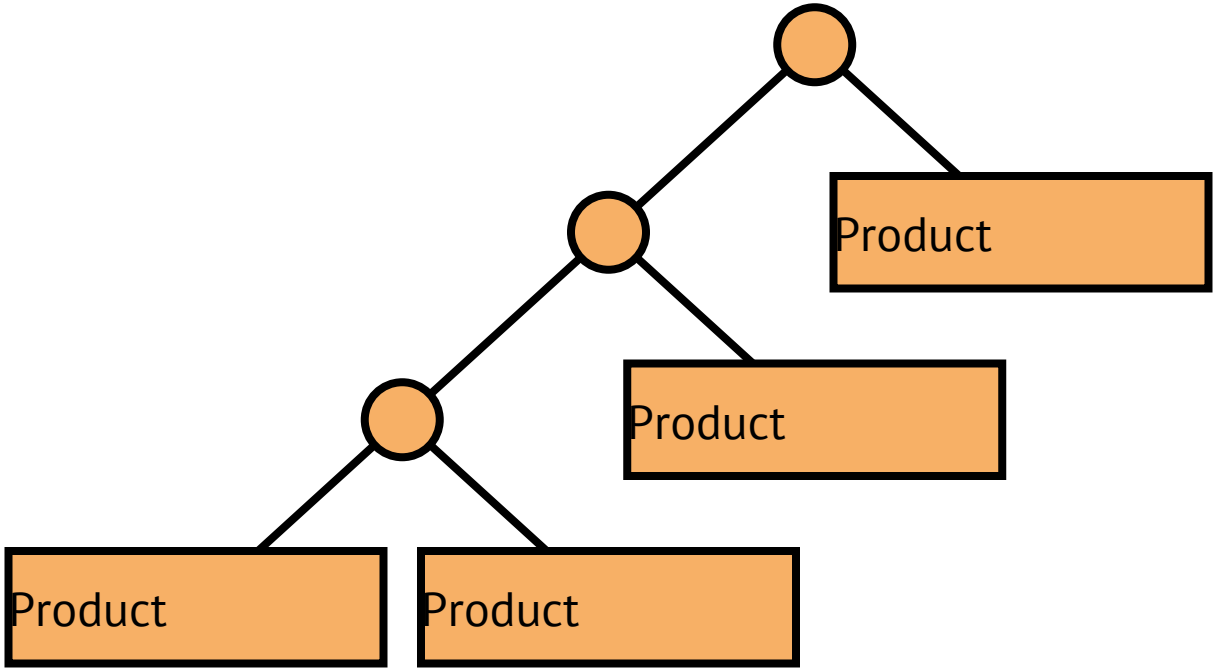


Arithmetische Terme

- Term = Variable oder
  Summe von Produkten
- Produkt = Multiplikation von Termen
- Hierarchische Struktur → Baumstruktur
  - Grundoperationen : $R \times R \rightarrow R$
    - Binärbaum
- Rekursive Struktur → Rekursive Prozedur

Term → Binärbaum

Term:



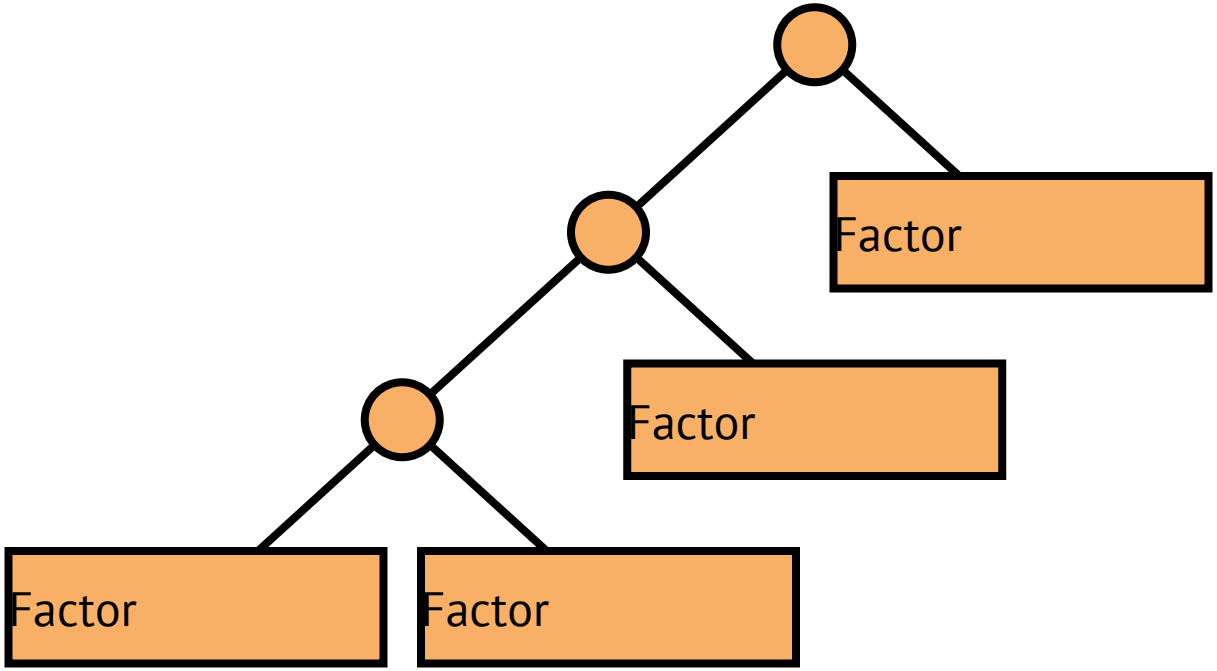
Term → Binärbaum

- BinTree ReadTerm()

```
L ← ReadProduct()
op ← getChar()
while op = '+' or op = '–' do
R ← ReadProduct()
L ← Node(L,op,R)
op ← getChar()
return L
```

Term → Binärbaum

Product:

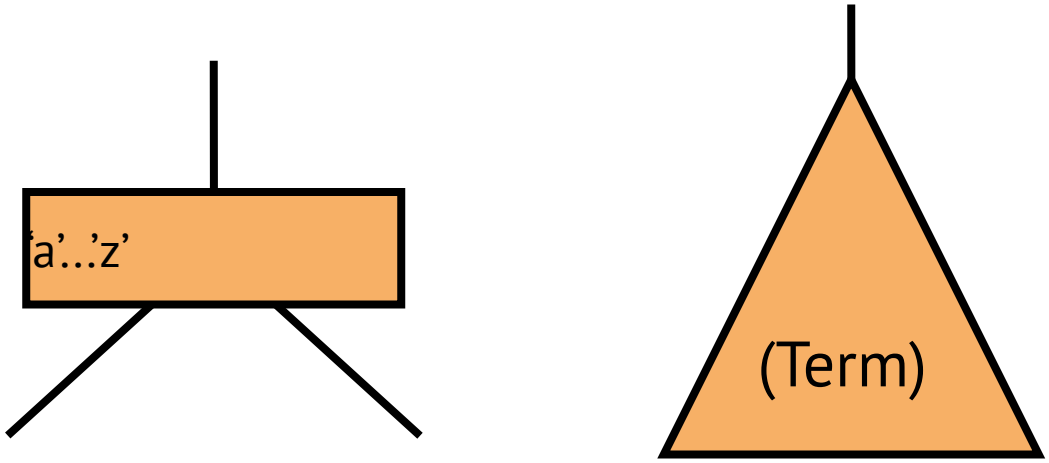


Term → Binärbaum

- BinTree ReadProduct()
  L = ReadFactor()
  op ← getChar()
  while op = '*' or op = '/' do
  R ← ReadFactor()
  L ← Node(L,op,R)
  op ← getChar()
  return L

Term → Binärbaum

Factor:



Term → Binärbaum

- BinTree ReadFactor()

```
c ← getChar()
if c in { `a`,..., `z` } then
return Node(Create(),c,Create())
else  //  c = `(`
L ← ReadTerm()
c ← getChar()  //  c = `)`
return L
```


Term → Binärbaum

A+B*(C+D)/E

Term → Binärbaum

A +B*(C+D)/E

ReadTerm()
ReadProduct()
ReadFactor()

Term → Binärbaum

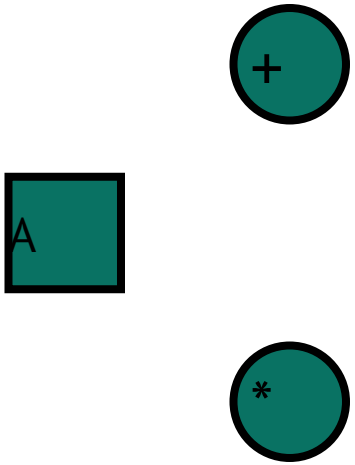
A + B*(C+D)/E

ReadTerm()

Term → Binärbaum

A + B*(C+D)/E

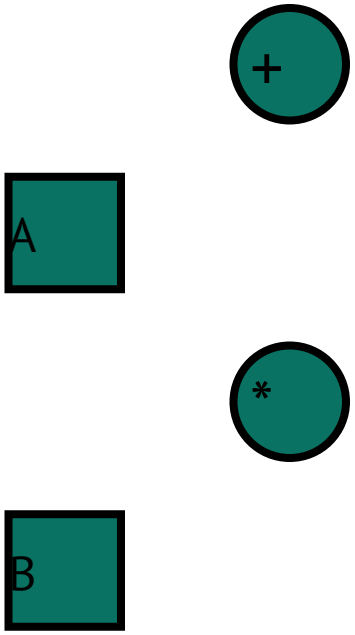
ReadTerm()
ReadProduct()
ReadFactor()



Term → Binärbaum

A + B*(C+D)/E

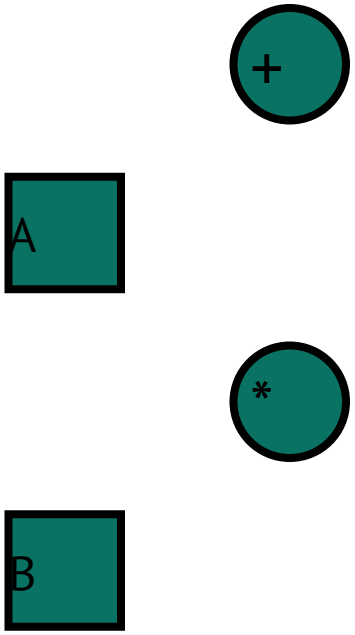
ReadTerm()
ReadProduct()



Term → Binärbaum

A + B*(C+D)/E

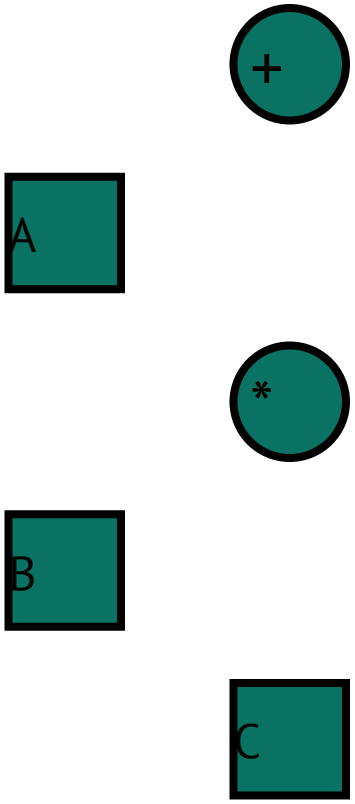
ReadTerm()
ReadProduct()
ReadFactor()
ReadTerm()
ReadProduct()
ReadFactor()



Term → Binärbaum

A + B*(C+D)/E

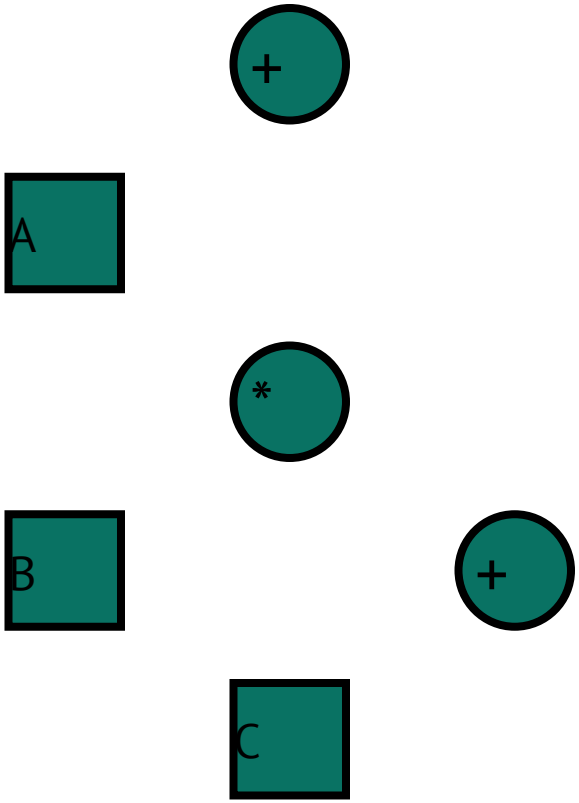
ReadTerm()
ReadProduct()
ReadFactor()
ReadTerm()
ReadProduct()
ReadFactor()



Term → Binärbaum

A + B*(C+D)/E

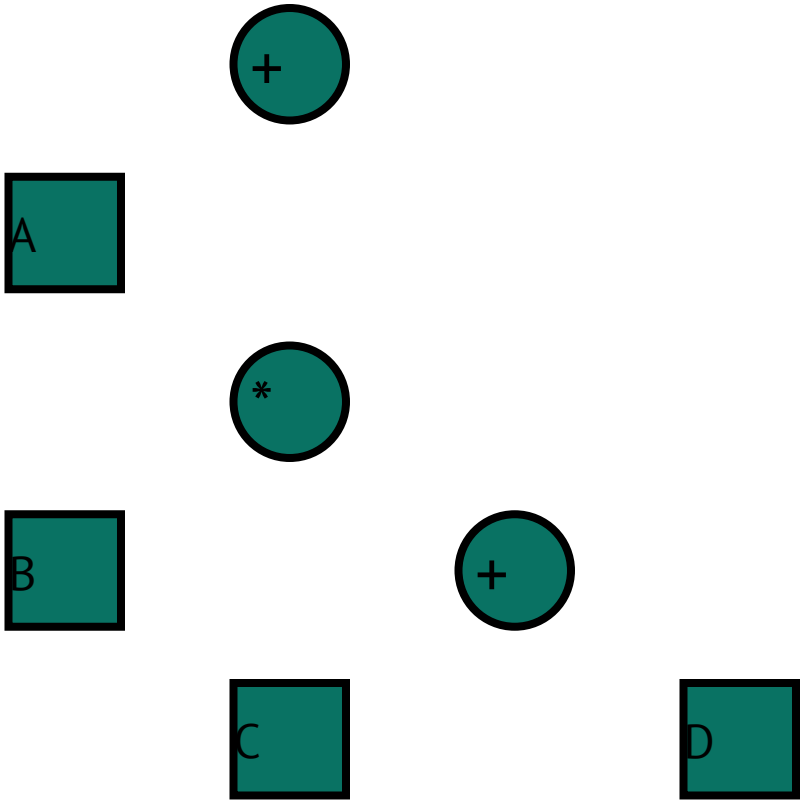
ReadTerm()
ReadProduct()
ReadFactor()
ReadTerm()



Term → Binärbaum

A + B*(C+D)/E

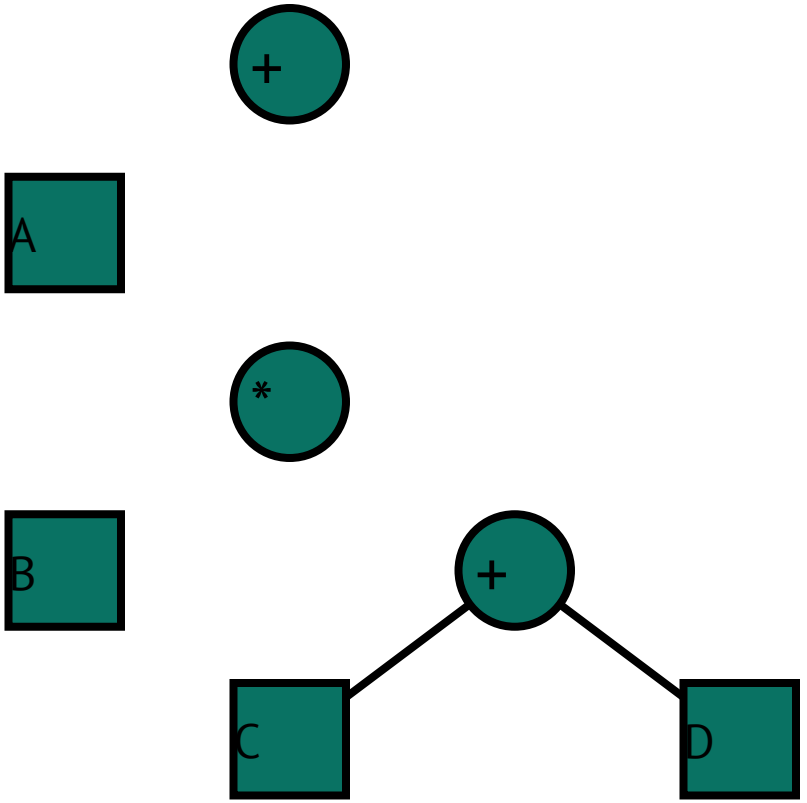
ReadTerm()
ReadProduct()
ReadFactor()
ReadTerm()
ReadProduct()
ReadFactor()



Term → Binärbaum

A + B*(C+D)/E

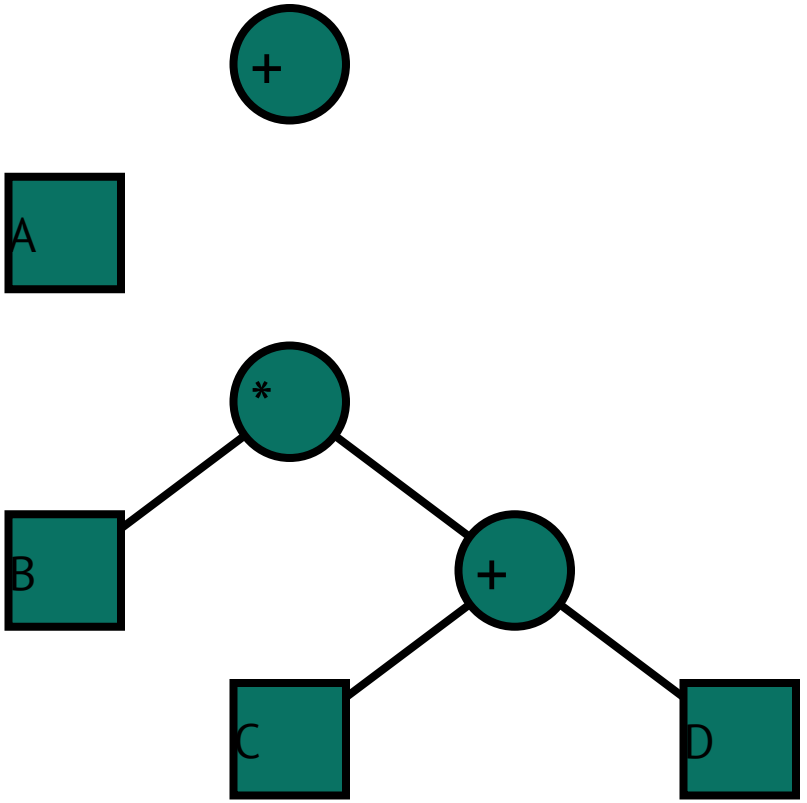
ReadTerm()
ReadProduct()
ReadFactor()



Term → Binärbaum

A + B*(C+D)/E

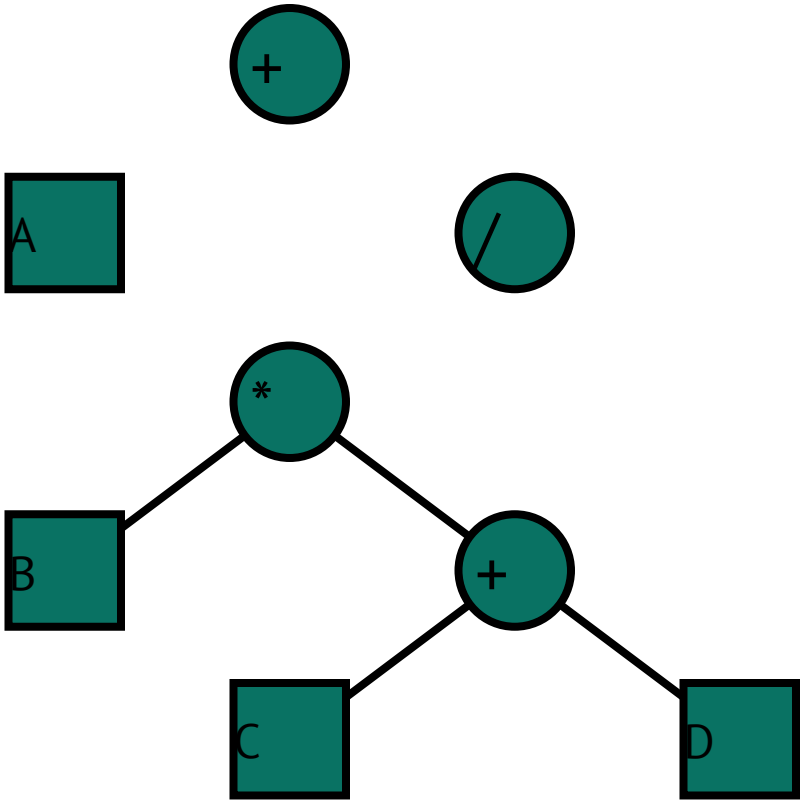
ReadTerm()
ReadProduct()



Term → Binärbaum

A + B*(C+D)/E

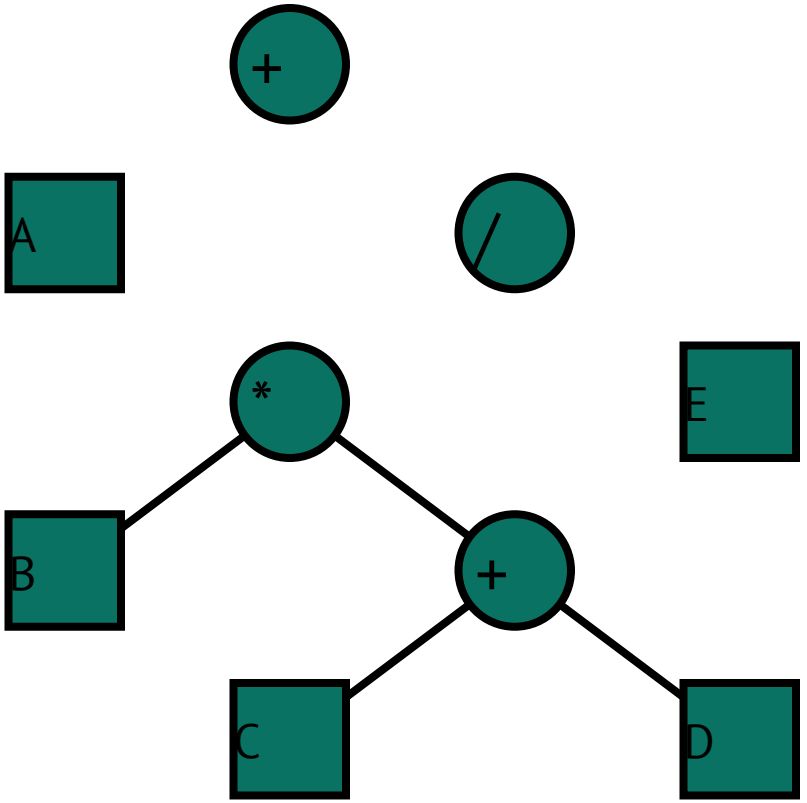
ReadTerm()
ReadProduct()



Term → Binärbaum

A + B*(C+D)/E

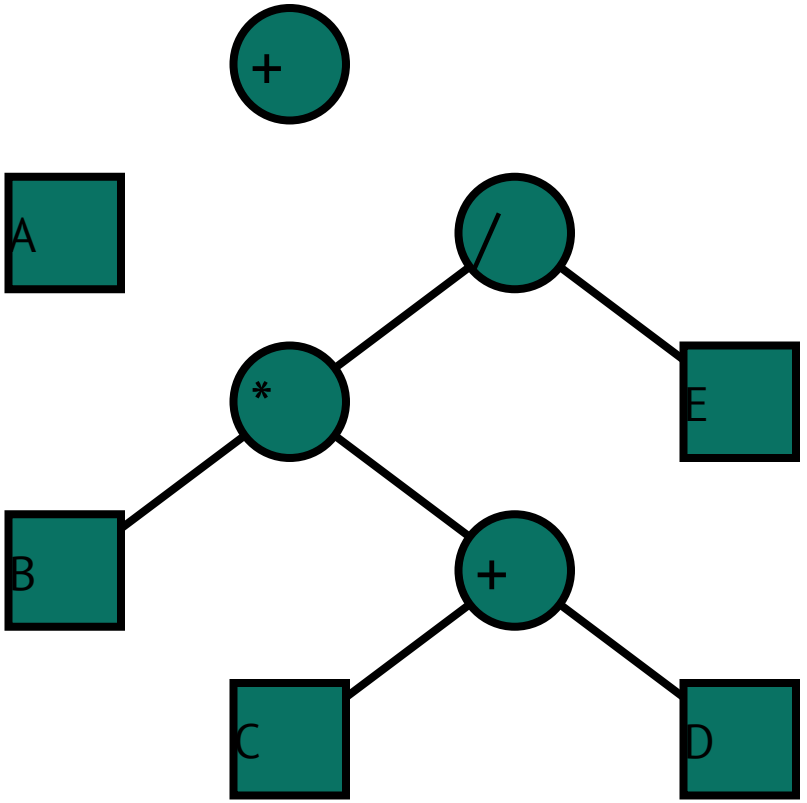
ReadTerm()
ReadProduct()
ReadFactor()



Term → Binärbaum

A + B*(C+D)/E

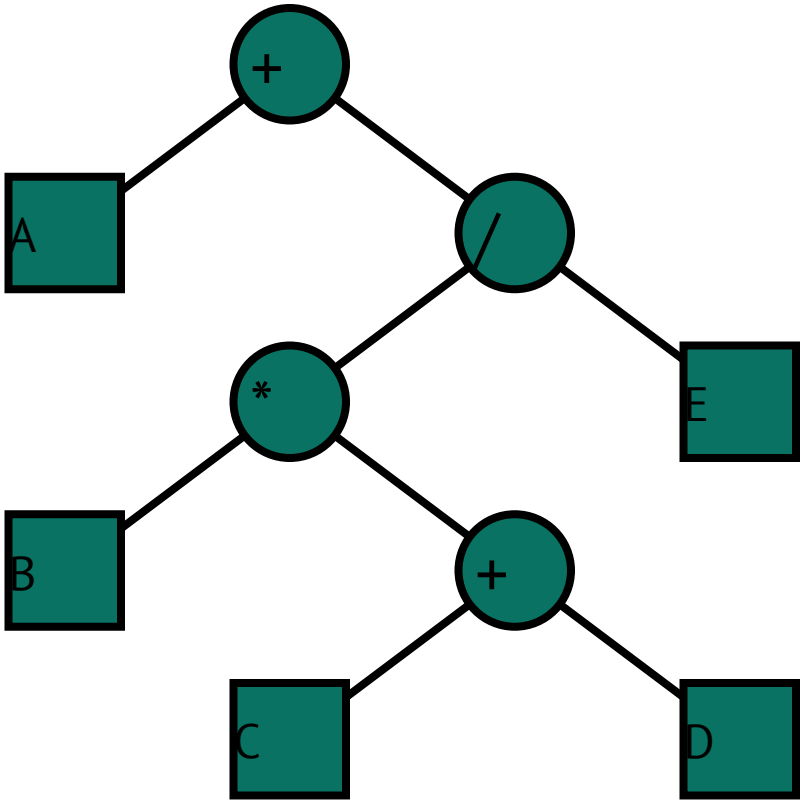
ReadTerm()
ReadProduct()



Term → Binärbaum

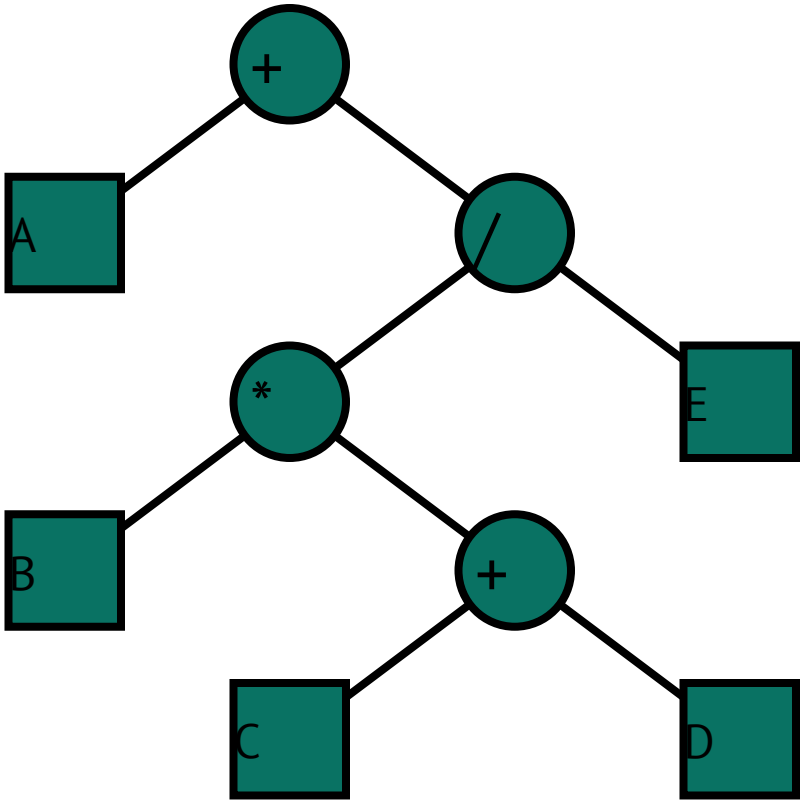
A + B*(C+D)/E

ReadTerm()

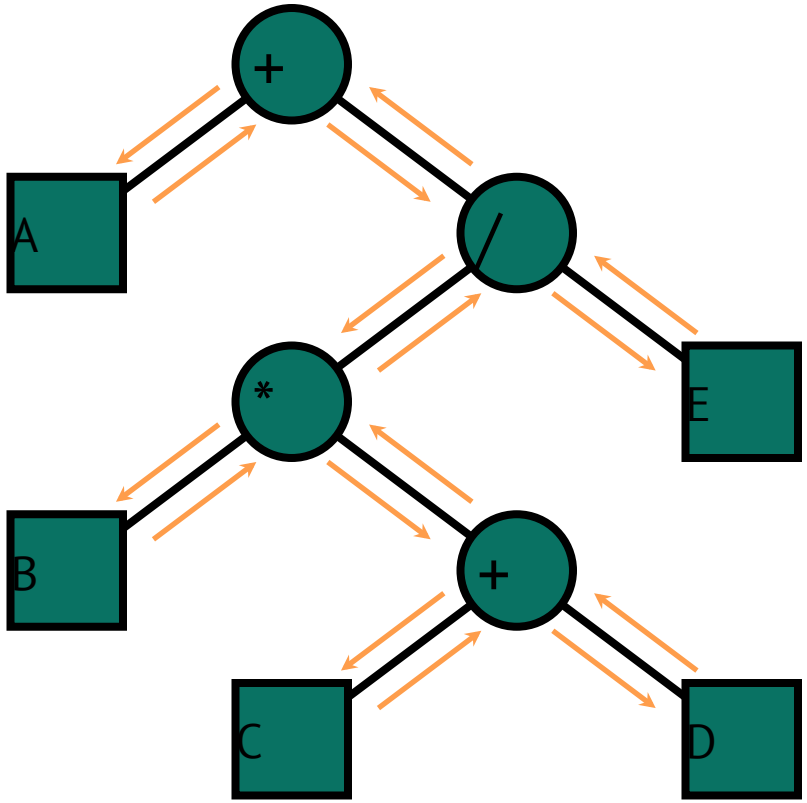


Term → Binärbaum

A + B*(C+D)/E



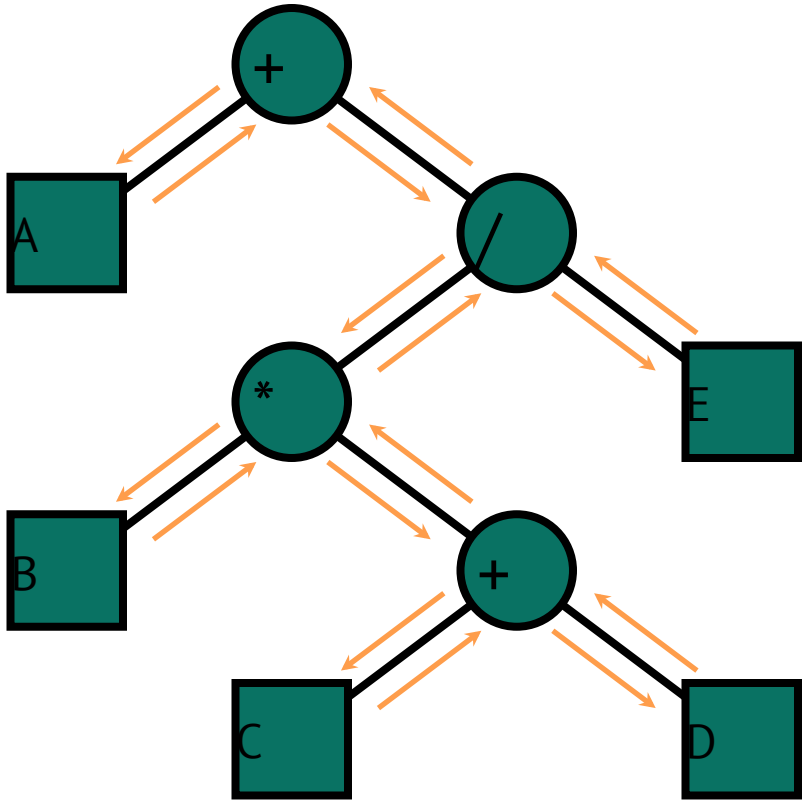
Traversierung



Traversierung

- Traverse(BinTree T)
  if Leaf(T) then
    output(Value(T))
  else
    Traverse(Left(T))
    Traverse(Right(T))

Traversierung



Traversierung

- Operator (= Knotenwert) steht
  - vor den Argumenten → Präfix
  - zwischen den Argumenten → Infix
  - nach den Argumenten → Postfix
- Einfache Varianten der rekursiven Traverse

Präfix

- Traverse(BinTree T)
  if Leaf(T) then
    output(Value(T))
  else
    output(Value(T))
    Traverse(Left(T))
    Traverse(Right(T))

Infix

- Traverse(BinTree T)
  if Leaf(T) then
    output(Value(T))
  else
    Traverse(Left(T))
    output(Value(T))
    Traverse(Right(T))

Infix

- Traverse(BinTree T)

```
if Leaf(T) then
        output(Value(T))
else
        output(`(`)
        Traverse(Left(T))
        output(Value(T))
        Traverse(Right(T))
        output(`)`)
```

Achtung: Klammern für korrekte Syntax notwendig !!!

Postfix

- Traverse(BinTree T)
  if Leaf(T) then
    output(Value(T))
  else
    Traverse(Left(T))
    Traverse(Right(T))
    output(Value(T))

Traversierungsstrategien

- Rekursive Algorithmen
  - Depth-first
    - Präfix / Infix / Postfix
- Nicht-rekursive Algorithmen
  - Depth-first (Stack)
  - Breadth-first (Queue)

Depth-First

- Traverse(BinTree T)
  if Leaf(T) then
    output(Value(T))
  else
    Traverse(Left(T))
    Traverse(Right(T))

Depth-First

- Traverse(BinTree T)
  - S ← CreateStack()
  - S ← Push(T,S)
  - DepthFirst(S)

Depth-First

- DepthFirst(Stack S)

```
while not Empty(S) do
        T ← Top(S)
        S ← Pop(S)
        … do something with T …
if not Empty(Right(T)) then
            S ← Push(Right(T),S)
if not Empty(Left(T)) then
            S ← Push(Left(T),S)
```

Breadth-First

- Zähle die Knoten nach ihrer Tiefe sortiert auf, d.h. alle Knoten mit Tiefe k vor dem ersten Knoten mit Tiefe $k+1$
- Labyrinth-Suche: Gehe nicht bis zur Sackgasse (depth-first), sondern probiere erst alle Pfade der Länge k vor den Pfaden mit Länge $k+1$ ...
- Verhindert unendliches Suchen

Breadth-First

- Traverse(BinTree T)
  - Q ← CreateQueue()
  - Q ← Enq(T,Q)
  - BreadthFirst(Q)

Breadth-First

- BreadthFirst(Queue Q)

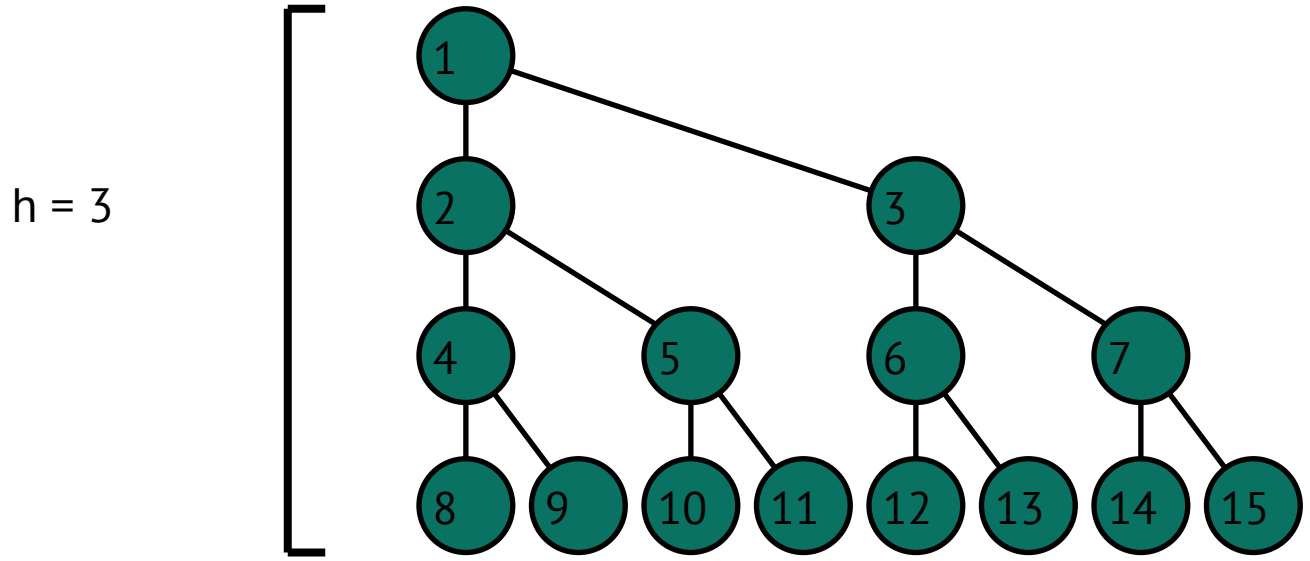
```
while not Empty(Q) do
        T ← Get(Q)
        Q ← Deq(Q)
        ... do something with T ...
if not Empty(Left(T)) then
            Q ← Enq(Left(T),Q)
if not Empty(Right(T)) then
            Q ← Enq(Right(T),Q)
```

Suchbäume

- Bäume ermöglichen wesentlich schnelleren Zugriff auf Elemente als bei lineare Strukturen
- Sortiere die Knoten (kommt später)
  - 1-dimensional
  - k-dimensional

Anzahl der Knoten

- Minimale Anzahl von Knoten in einem Binärbaum der Höhe h: $N_{min}(h) = h+1$
- Maximale Anzahl: $N_{max}(h) = 2^{h+1}-1$

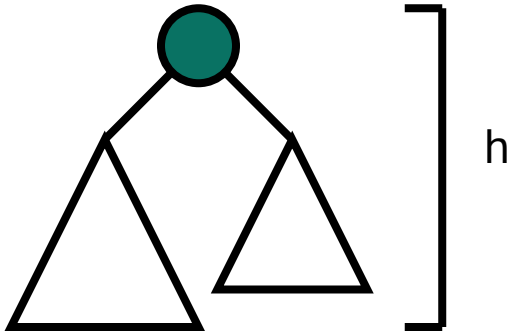


Höhe des Baumes

- Für n Knoten ist die maximale Höhe des Binärbaumes: $H_{max}(n) = n-1$
- Die minimale Höhe: $H_{min}(n) = \lceil \log(n+1)/\log(2) \rceil - 1$
- Die Tiefe+1 eines Knoten beschreibt die Anzahl der Zugriffe, um ihn zu finden.

Fragen der Effizienz

- Optimaler Zugriff bei vollständigen Bäumen. Diese sind aber nicht immer einfach zu konstruieren.



- Balancierte Bäume ...
  - $N_{bal,max}(h) = 2^{h+1}-1$
  - $N_{bal,min}(h) = 1 + N_{bal,min}(h-1) + N_{bal,min}(h-2)$

Fragen der Effizienz

- Minimale Anzahl von Knoten

$N_{bal,min}(h) = 1 + N_{bal,min}(h-1) + N_{bal,min}(h-2)$

$\geqslant 2 \times N_{bal,min}(h-2)$

$\geqslant 4 \times N_{bal,min}(h-4)$

$\geqslant$ $\begin{cases} 2^{(h-1)/2} \times N_{bal,min}(1) & \text{... H ungerade} \\ 2^{h/2} \times N_{bal,min}(0) & \text{... H gerade} \end{cases}$

$\geqslant \sqrt{2}^{h}$

Fragen der Effizienz

- Minimale Anzahl von Knoten

  $N_{bal,min}(h) \geqslant \sqrt{2}^h$

- Maximale Höhe für n Knoten

$$H_{bal,max}(n) \leqslant \lceil \log(n)/\log(\sqrt{2}) \rceil$$

$$= 2 \times \lceil \log(n)/\log(2) \rceil$$

$$\approx 2 \times H_{min}(n)$$

Suchen in Suchbäumen

- Knoten sind sortiert angeordnet (Sortieralgorithmen später)
  - max(T) = maximaler Knotenwert im Baum T
  - min(T) = minimaler Knotenwert im Baum T
  - Sortierung … für alle Knoten/Sub-Bäume gilt:
    $\max(\text{Left}(T)) \leqslant \text{Value}(T) < \min(\text{Right}(T))$

Suchen in Suchbäumen

- Depth-first Traversal
  - Mit jedem Test kann bis zu der Hälfte der Knoten ausgeschlossen werden.

Suchen in Suchbäumen

- Bool Search(Element X, BinTree T)

```
if X = Value(T) then
        return true
else if Leaf(T) then
        return false
else if X < Value(T) then
        Search(X,Left(T))
else
        Search(X,Right(T))
```

Einfügen und Löschen

- Bei Suchbäumen bestimmt der Wert des Elementes die Position im Baum (impliziter Marker)
- Einfügereihenfolge bestimmt die Struktur (bessere Algorithmen später)
- Löschen innerer Knoten nicht trivial

Einfügen

- Insert(X,Create()) =
  Node(Create(),X,Create())
- Insert(X,Node(L,Y,R)) =
  Node(Insert(X,L),Y,R) … if X ⩽ Y
  Node(L,Y,Insert(X,R)) … if X > Y

Löschen

- Remove(X,Node(L,Y,R)) =
  - Node(Remove(X,L),Y,R) ... if X < Y
  - Node(L,Y,Remove(X,R)) ... if X > Y

// X ≠ Y

- Max(Node(L,X,Create()) = X
  - Max(Node(L,X,R)) = Max(R)
- Min(Node(Create(),X,R) = X
  - Min(Node(L,X,R)) = Min(L)

Löschen

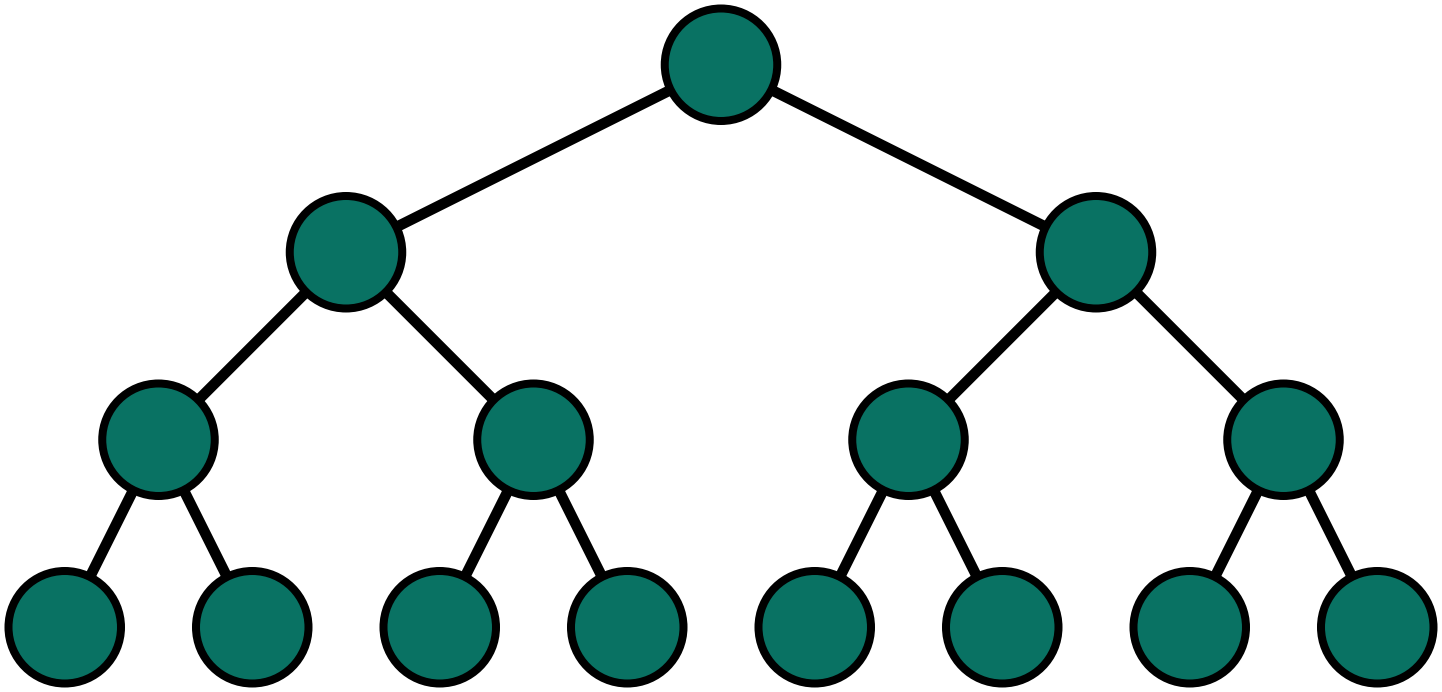
- Remove(X,Node(L,X,R)) =

Node(Remove(Max(L),L),Max(L),R)
... if L ≠ Create()

Node(L,Min(R),Remove(Min(R),R))
... if L = Create() and R ≠ Create()

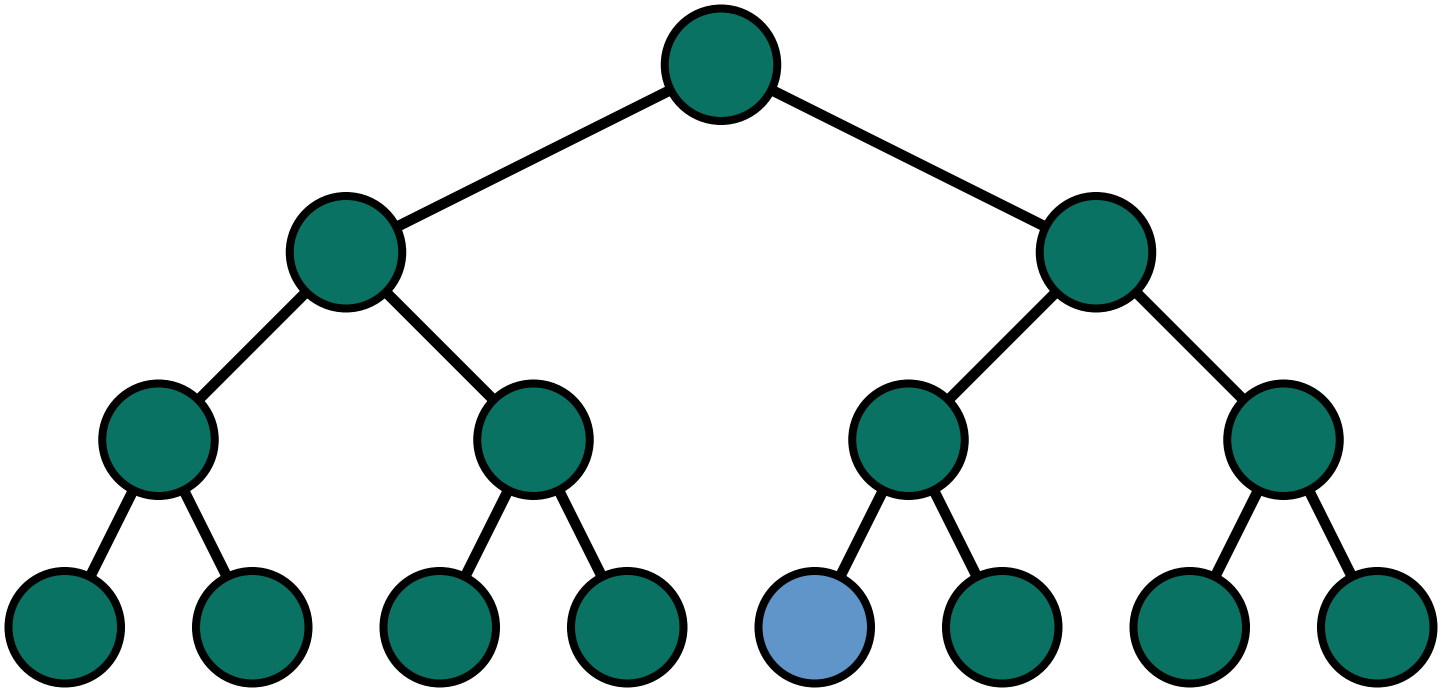
Create() ... if L = R = Create()

Beispiel



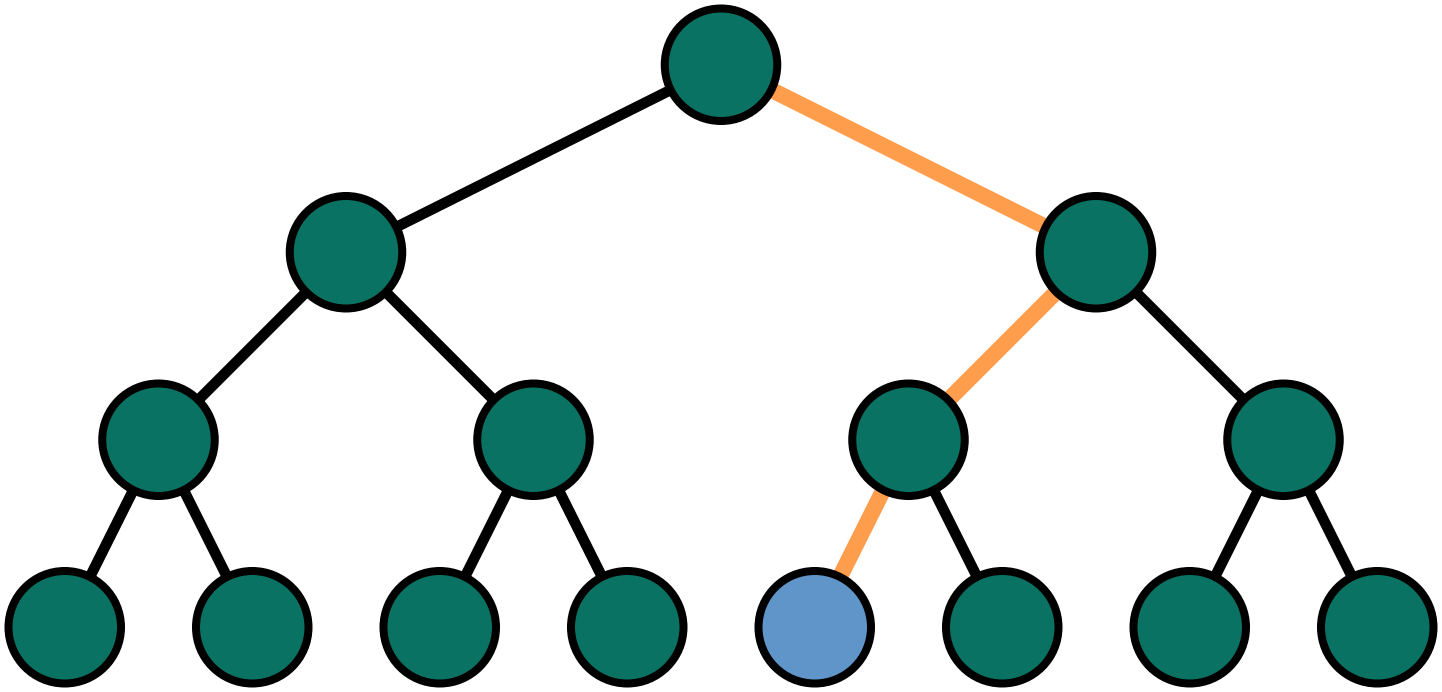
Beispiel

Remove(,T)



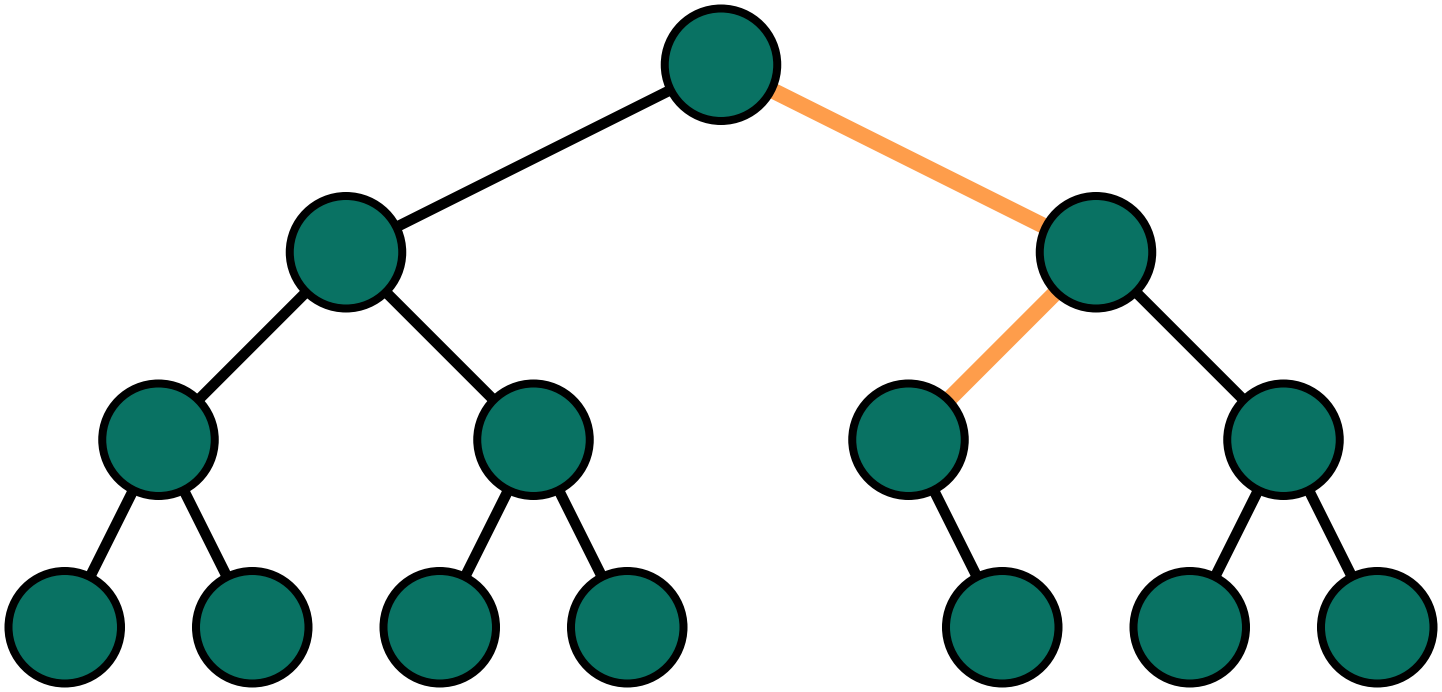
Beispiel

Remove(,Node(Create(), ,Create()))

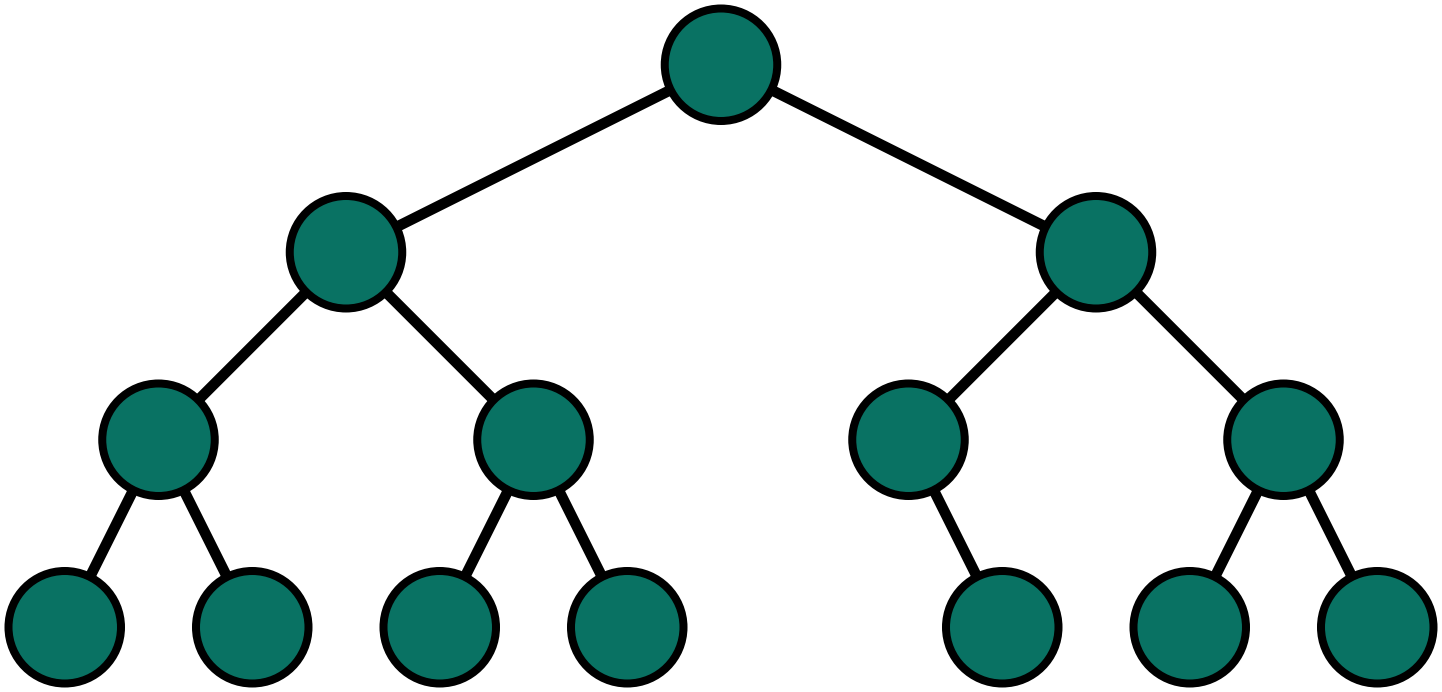


Beispiel

Create()

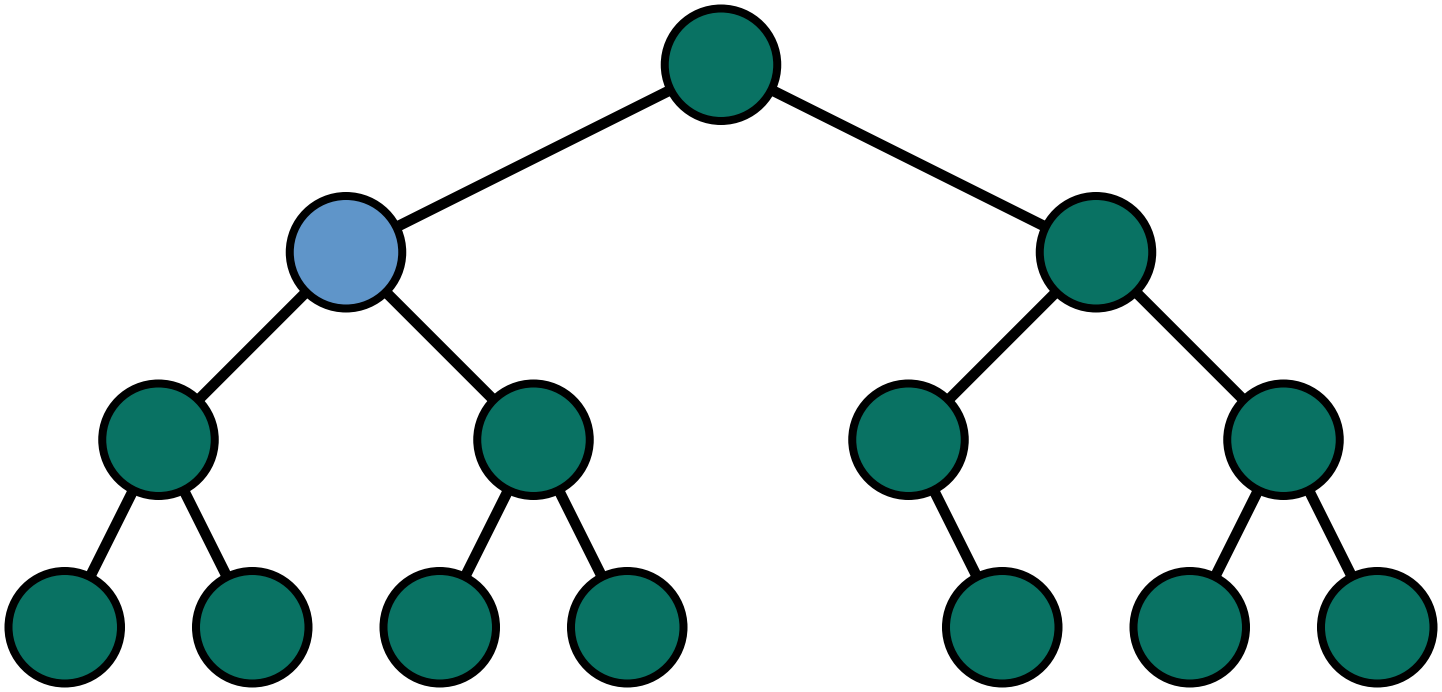


Beispiel



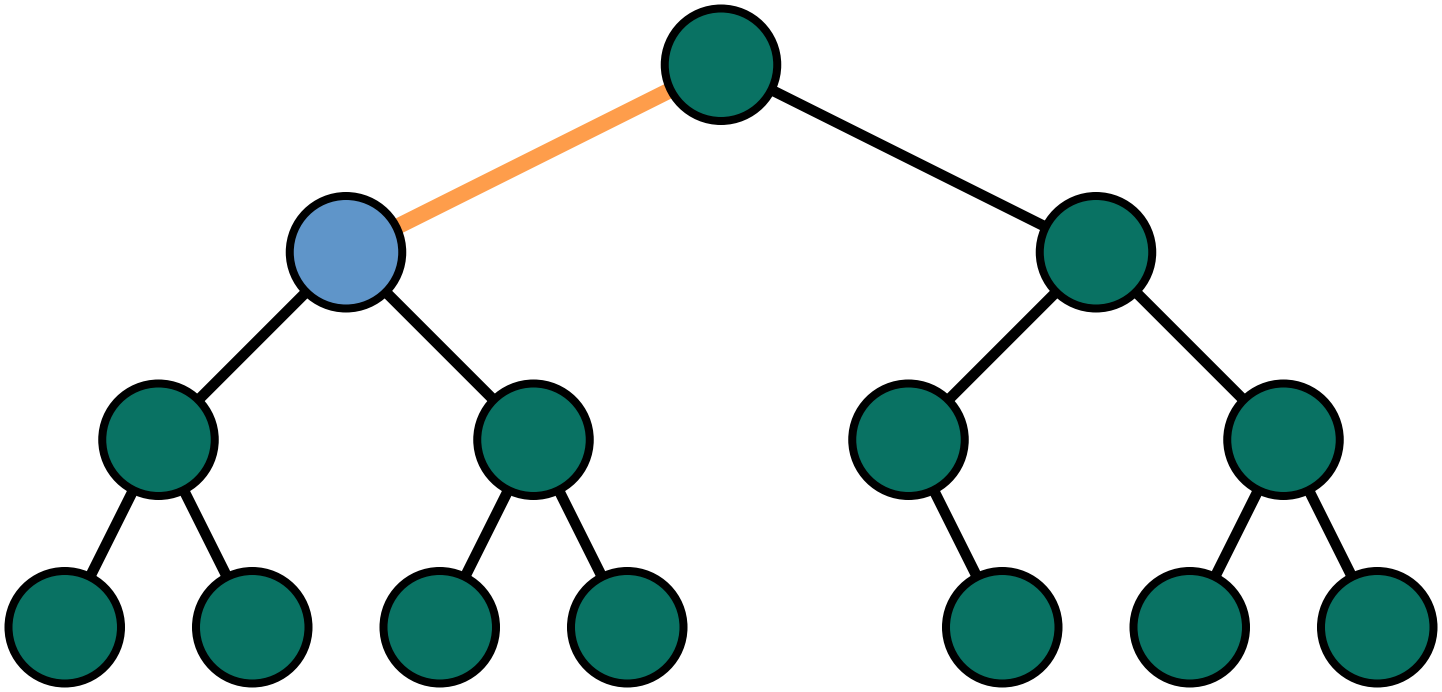
Beispiel

Remove(,T)



Beispiel

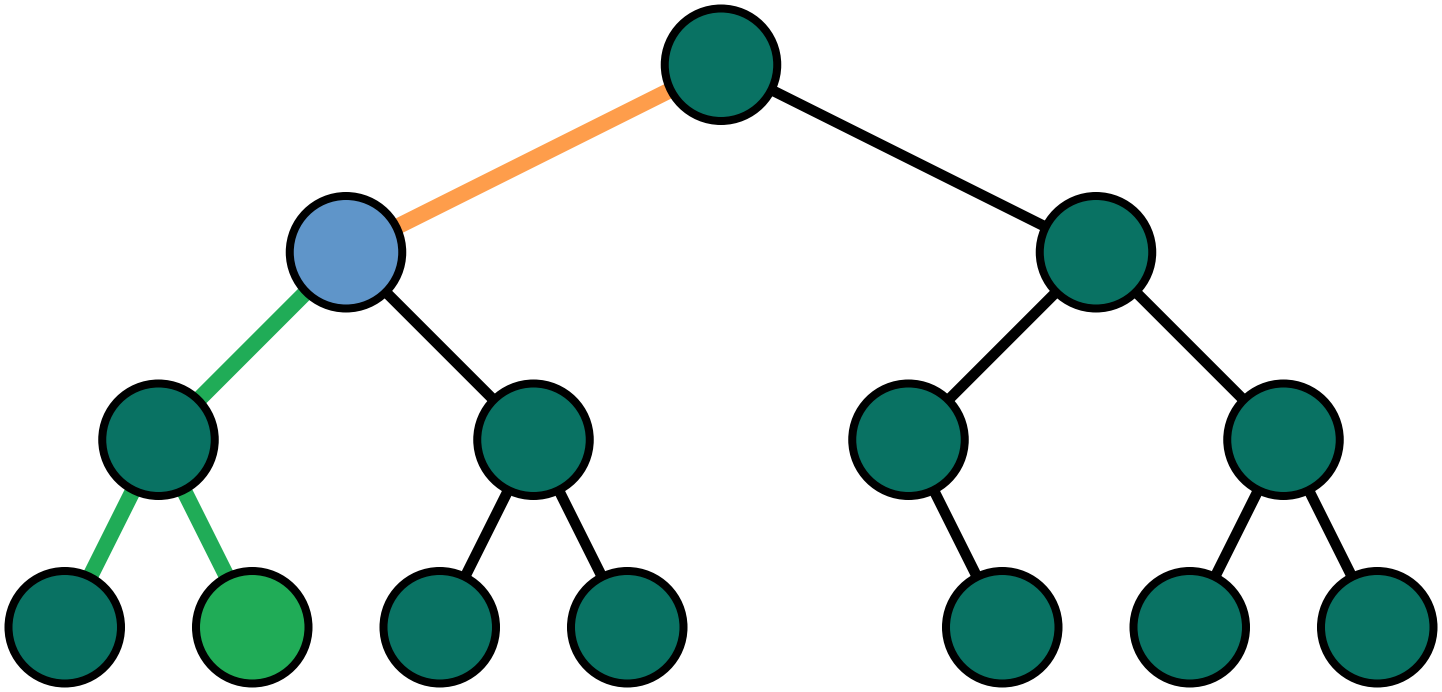
Remove(,Node(,R))



Beispiel

Remove(,Node(,R))

= Max(L)

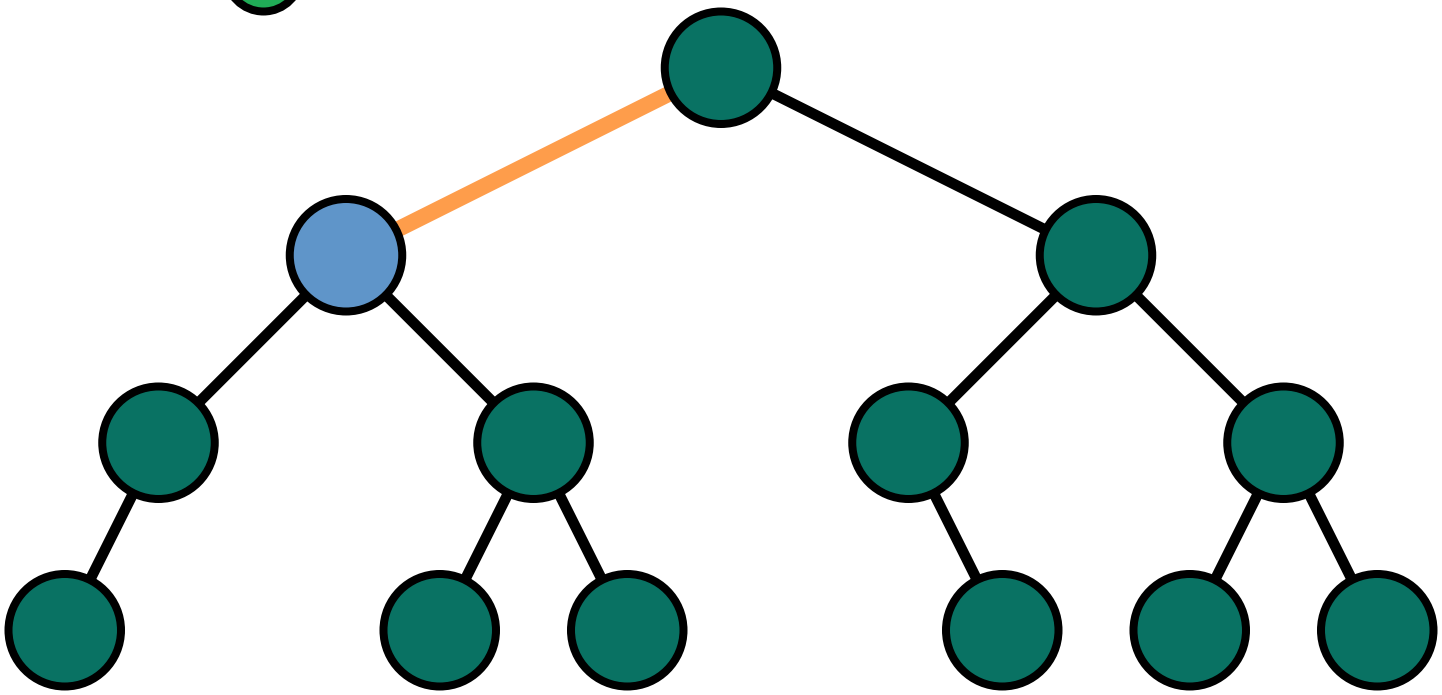


Beispiel

Remove(,Node(,R))

= Max(L)

Remove(,L)

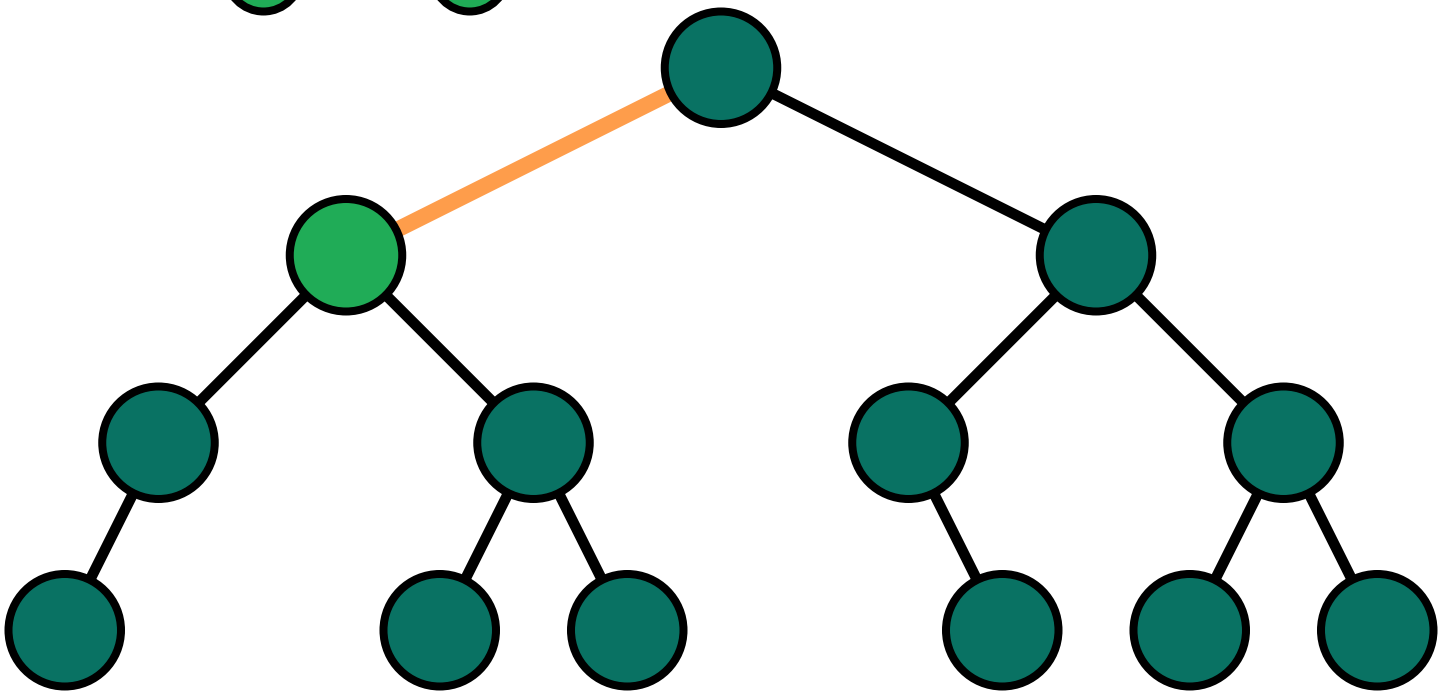


Beispiel

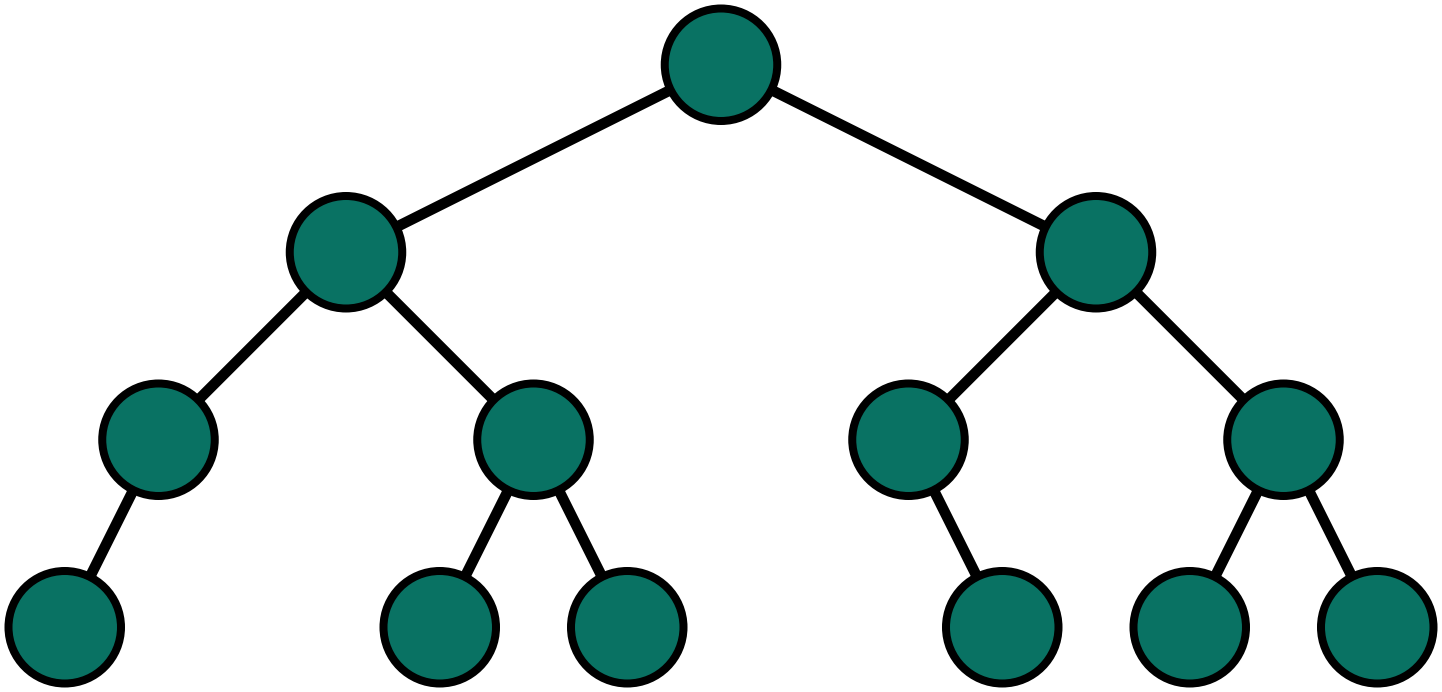
Remove(,Node(,R))

= Max(L)

Node(Remove(,L), ,R)



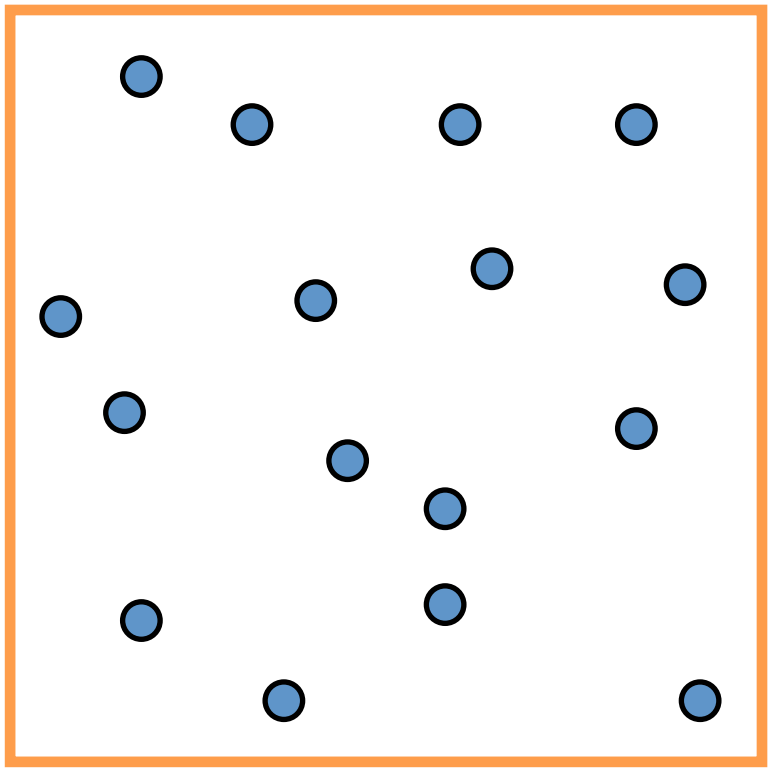
Beispiel



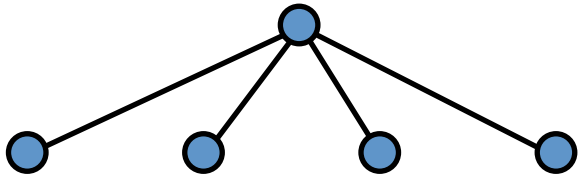
k-dimensionale Suchbäume

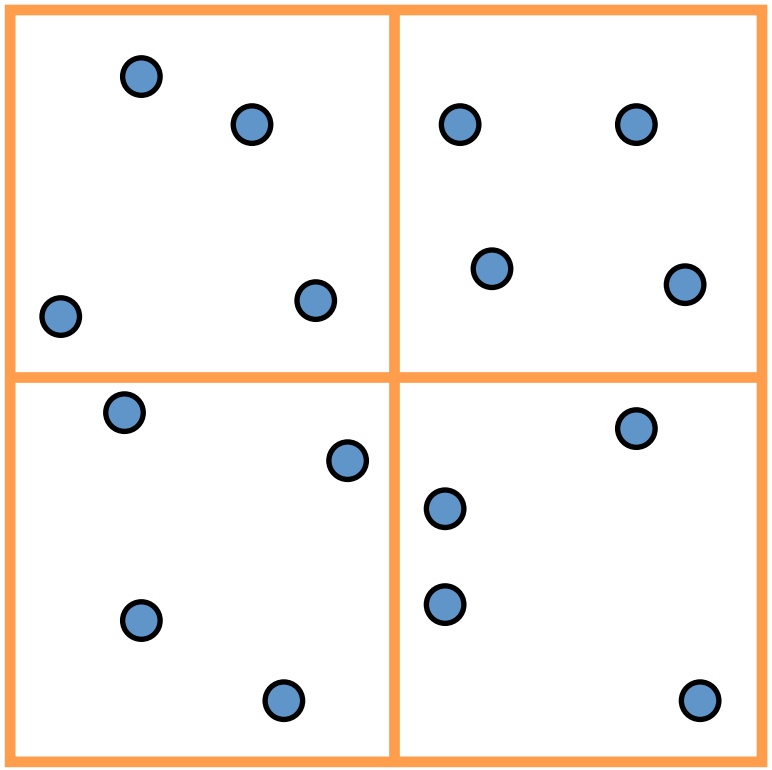
- Beispiel: Suche nach Punkten im R^2 (nächste Tankstelle, …) oder im R^k
- Verwende 2^k-näre Bäume
  - Quadtree (R^2)
  - Octree (R^3)
  - …
- kD-Bäume (k-dimensionale Binärbäume)

Quadtree

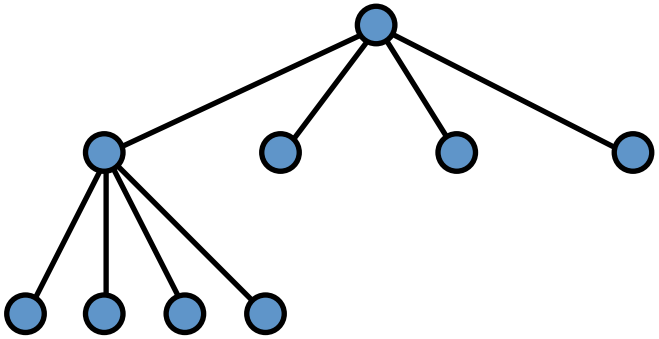


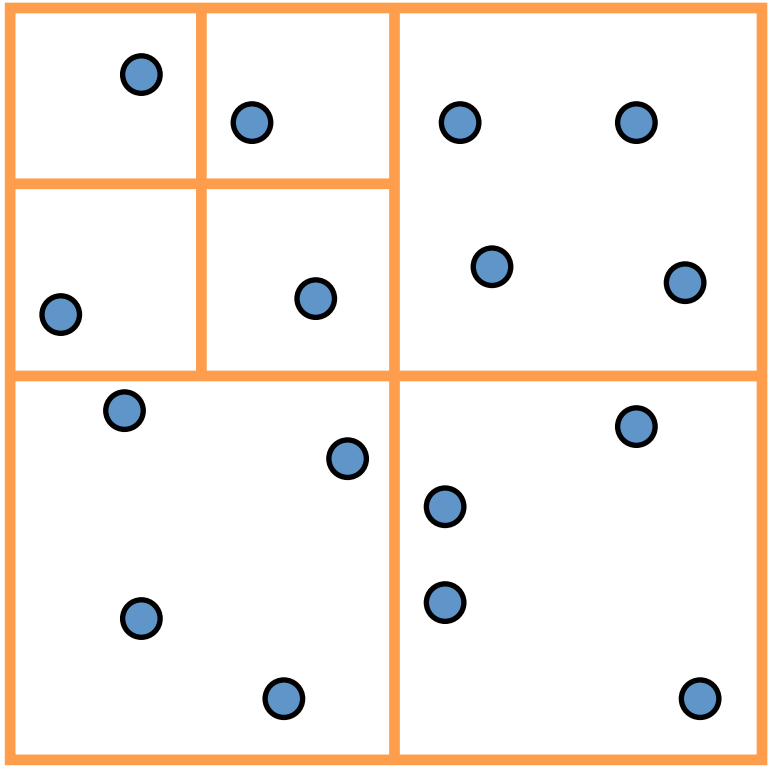
Quadtree



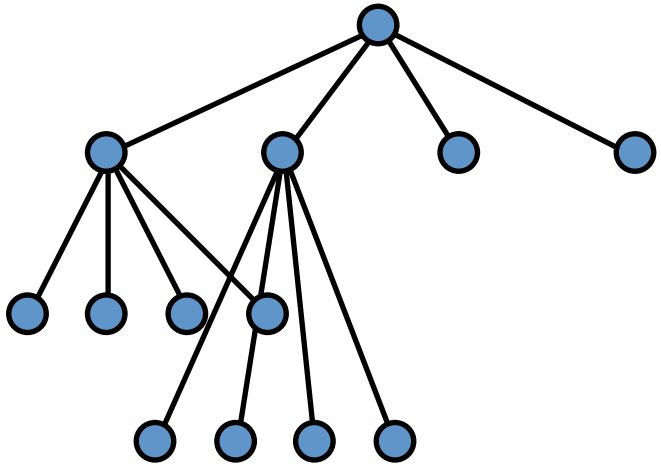


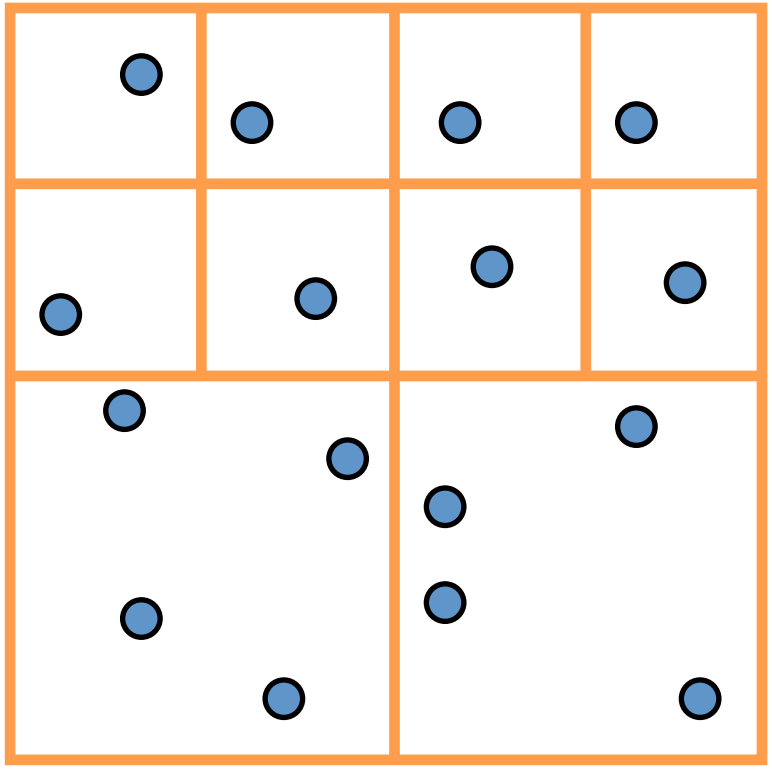
Quadtree



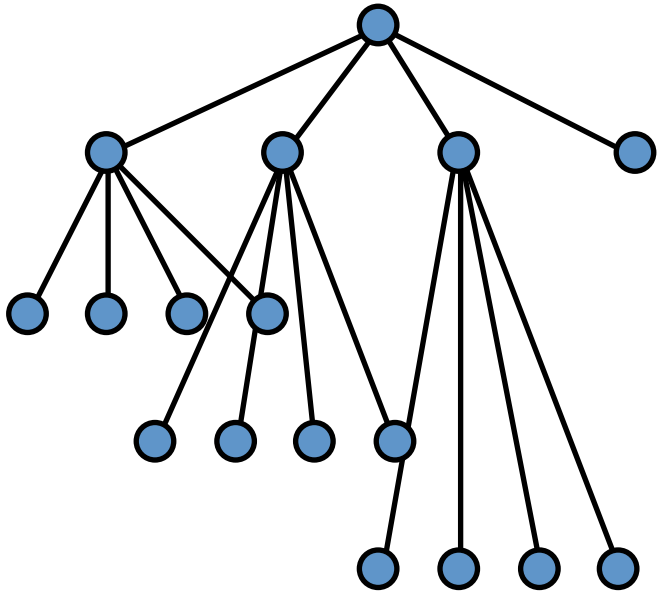


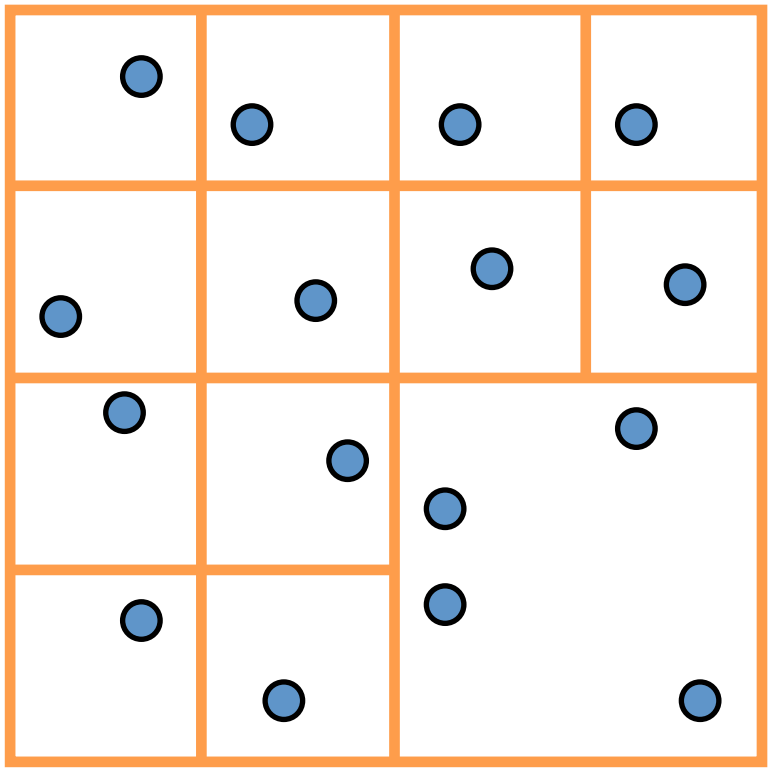
Quadtree



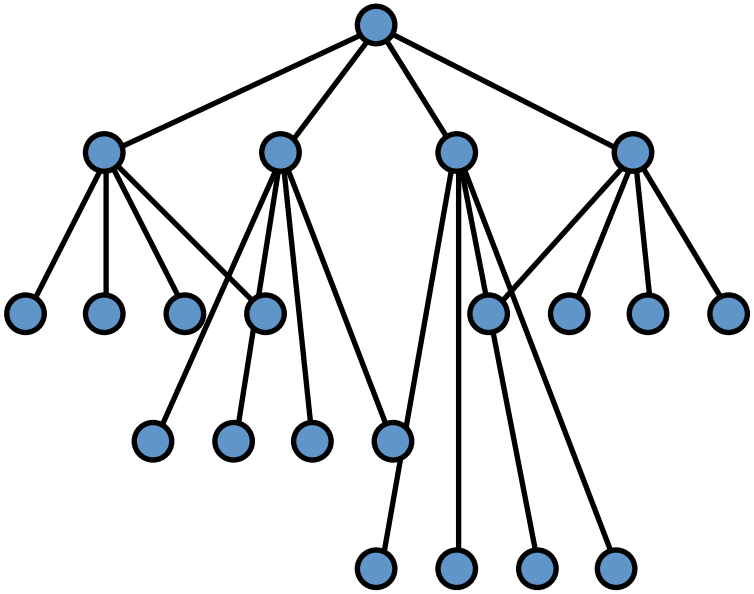


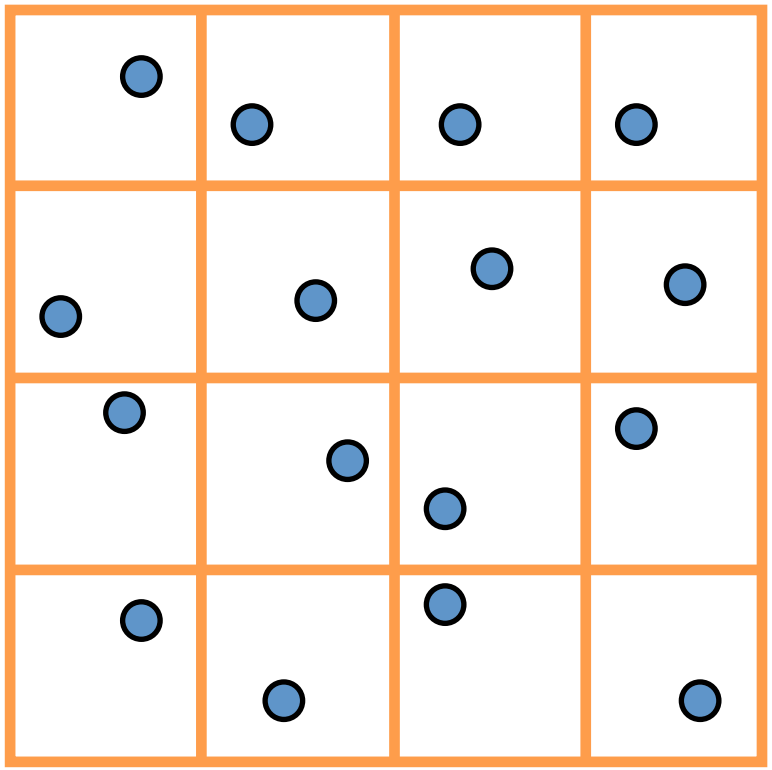
Quadtree





Quadtree

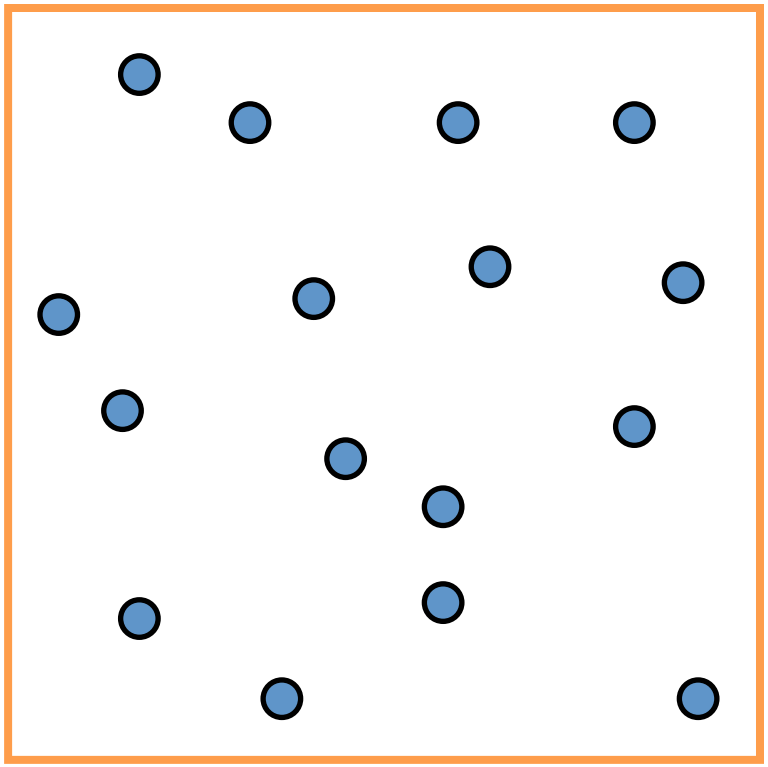




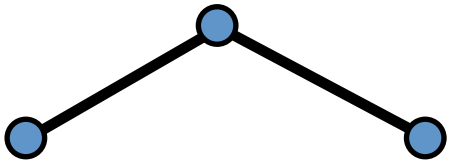
kD-Bäume

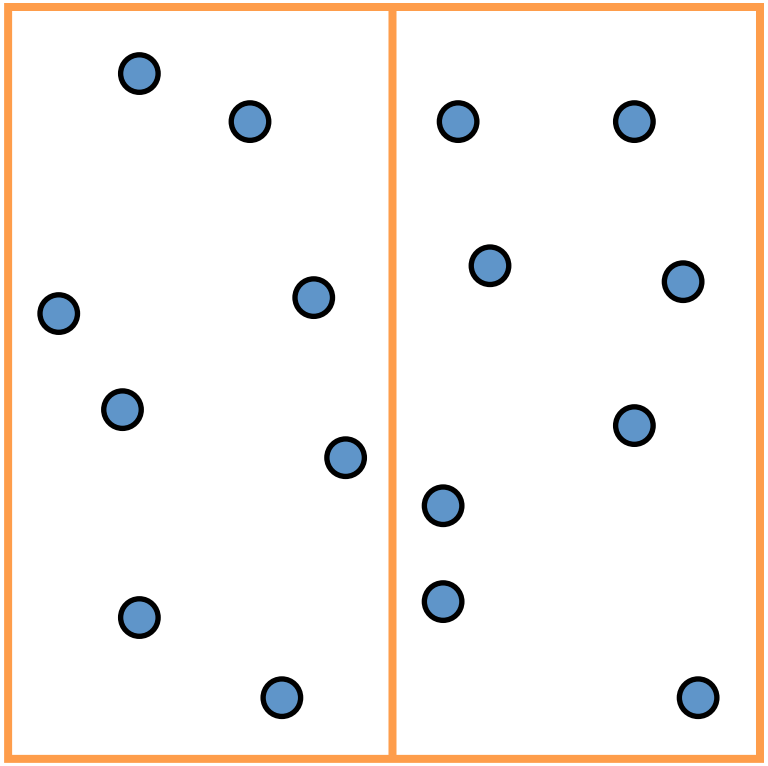
- Datentyp: Binärbaum
- Jeder Knoten enthält einen k-dim. Wert
- Sortierung für Knoten T der Tiefe h: $\max_h(\text{Left}(T)) \leqslant \text{Value}_h(T) < \min_h(\text{Right}(T))$
- Subskript h bedeutet: Verwende die (h mod k)-te Koordinate

kD-Bäume

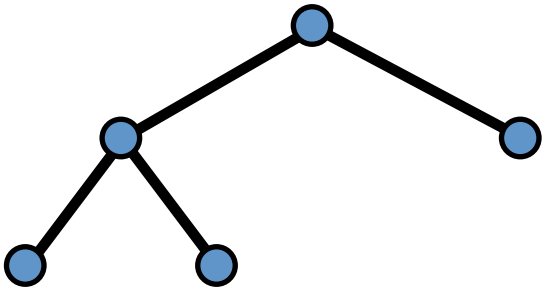


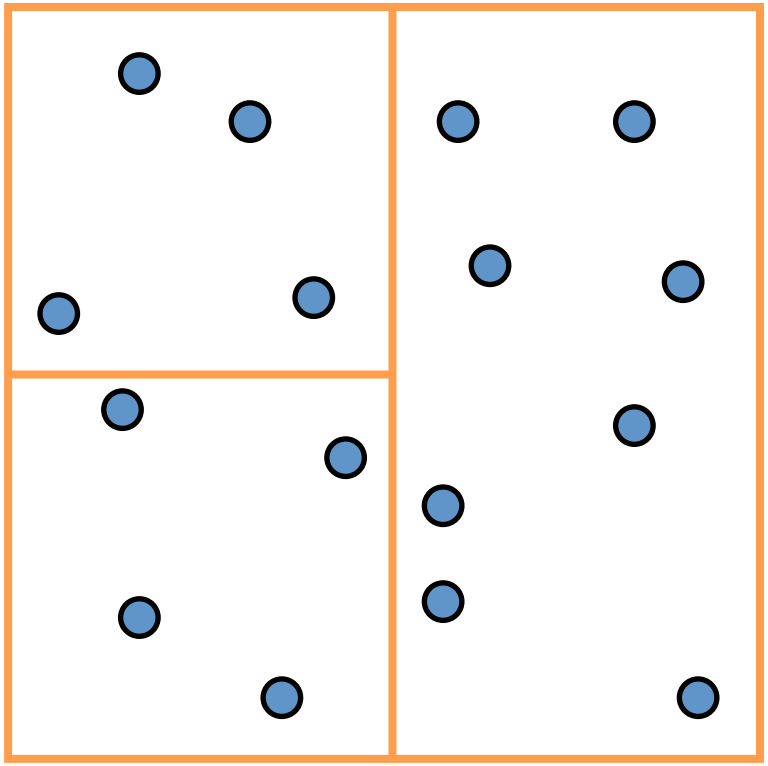
kD-Bäume



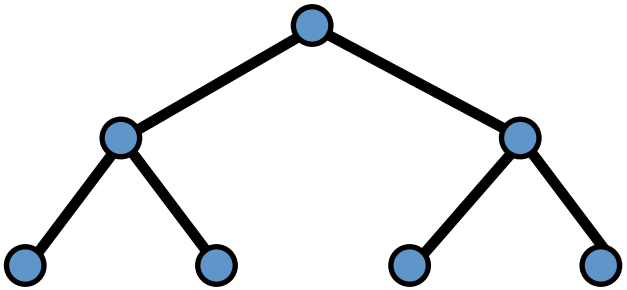


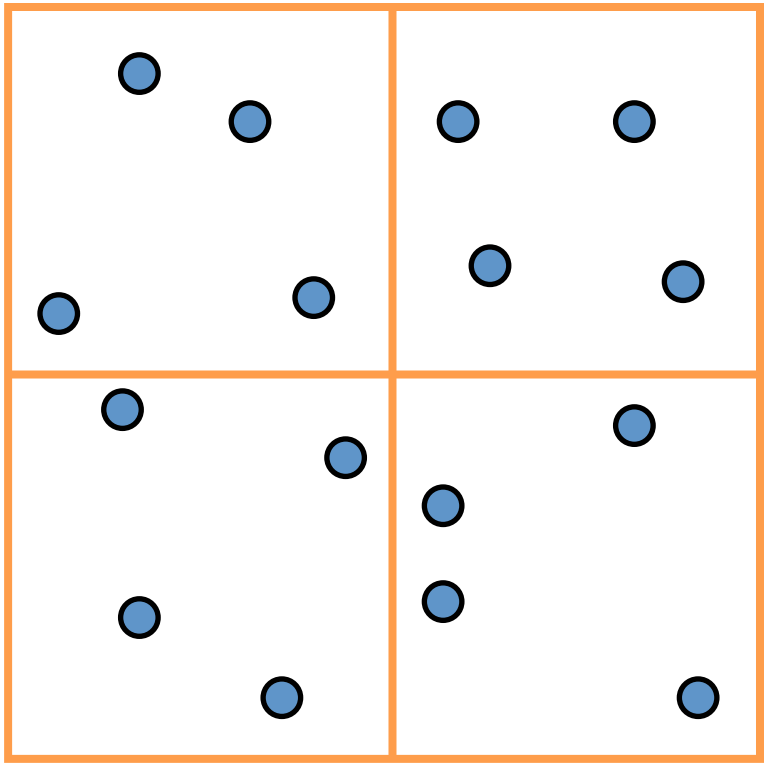
kD-Bäume



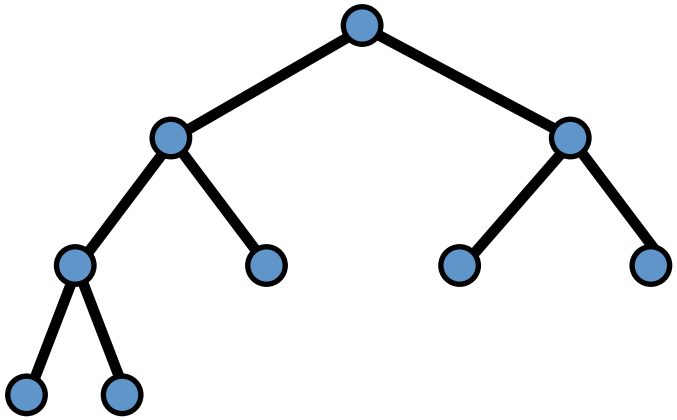


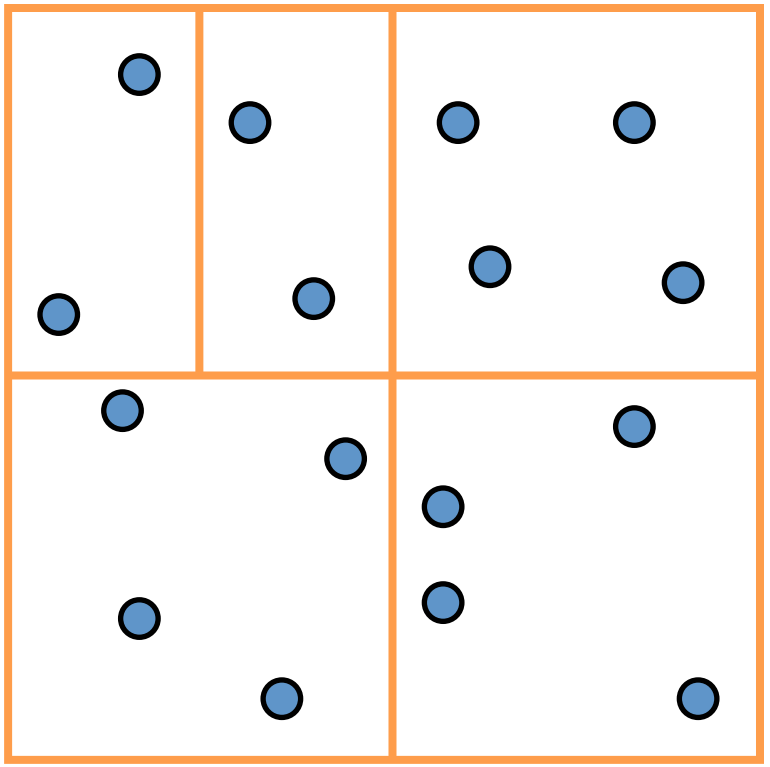
kD-Bäume



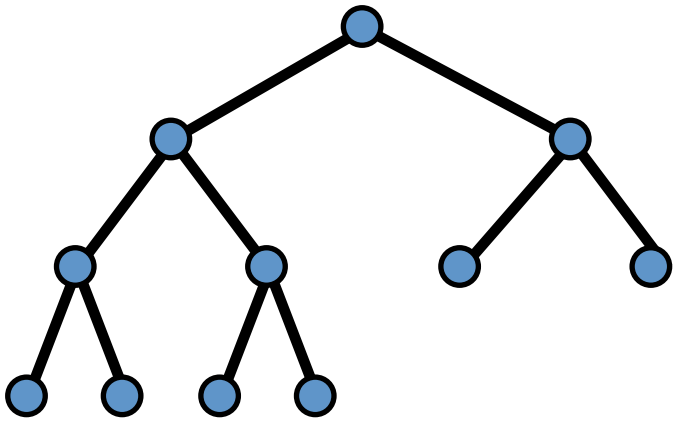


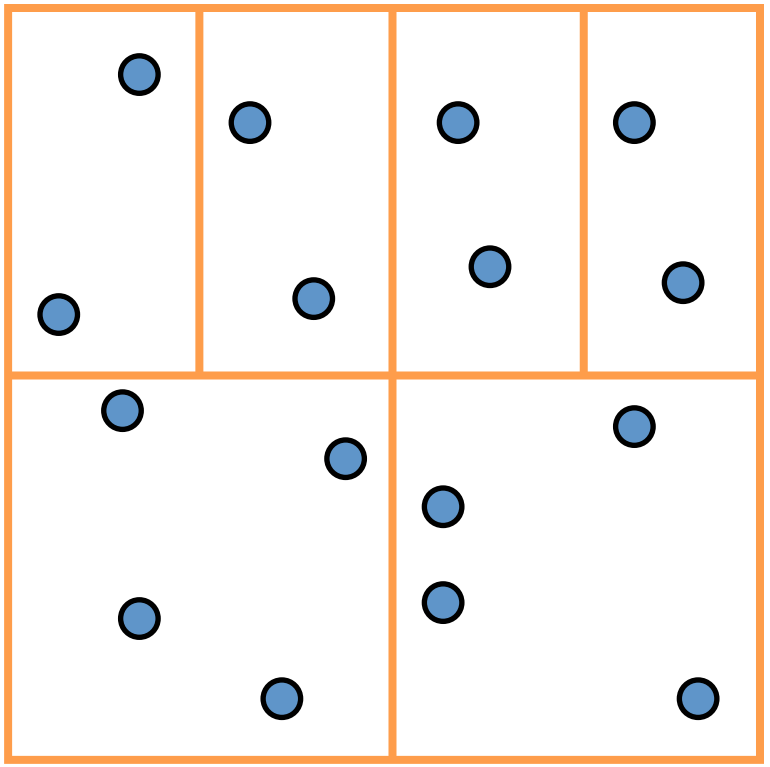
kD-Bäume



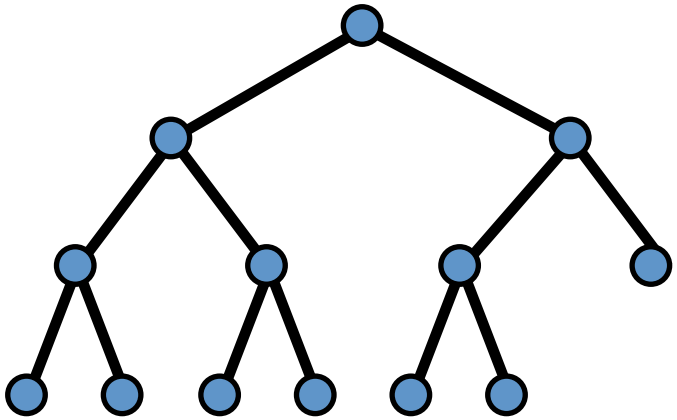


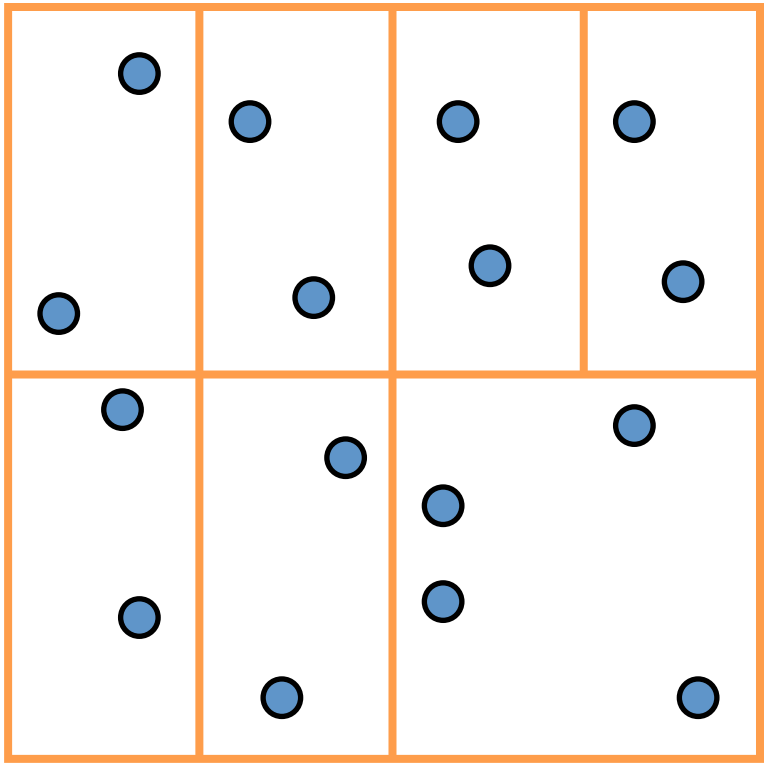
kD-Bäume



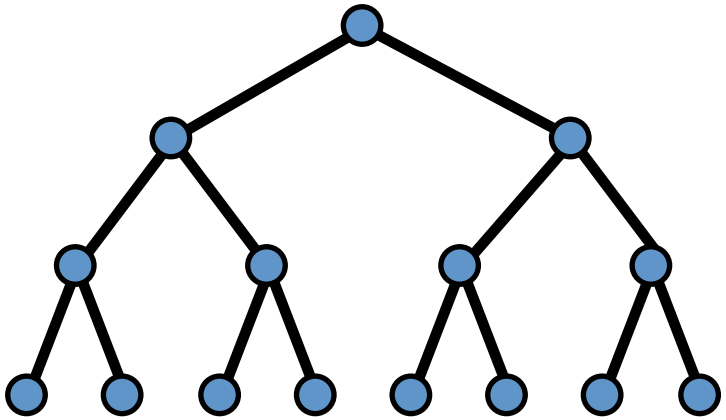


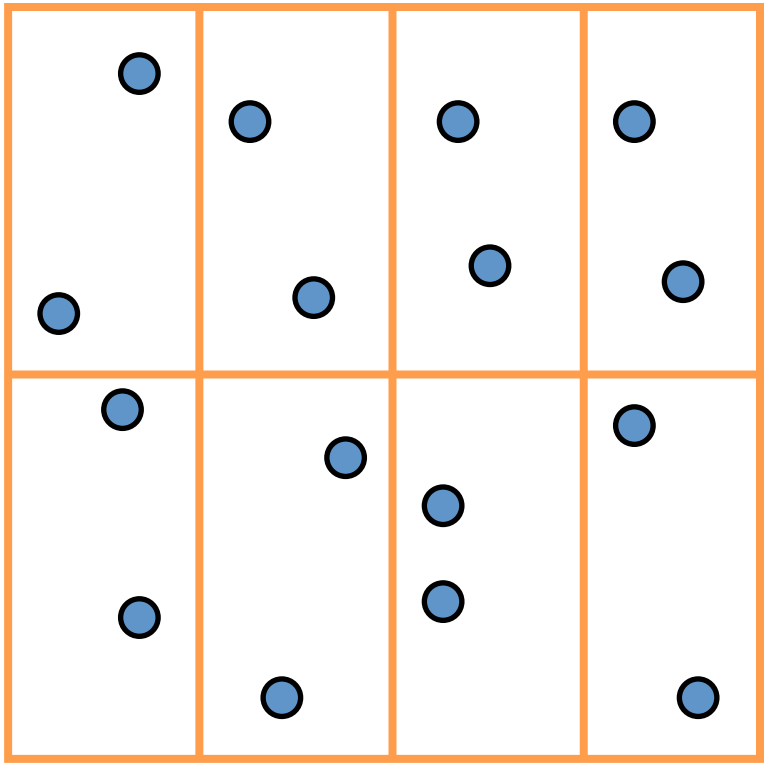
kD-Bäume



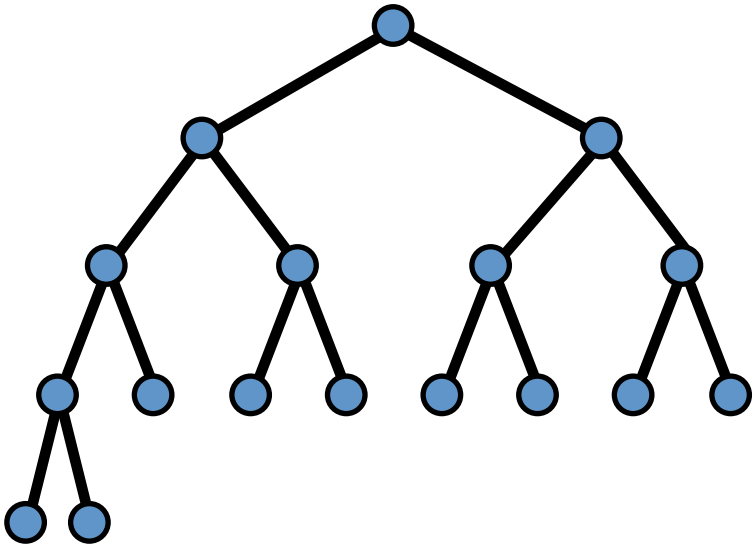


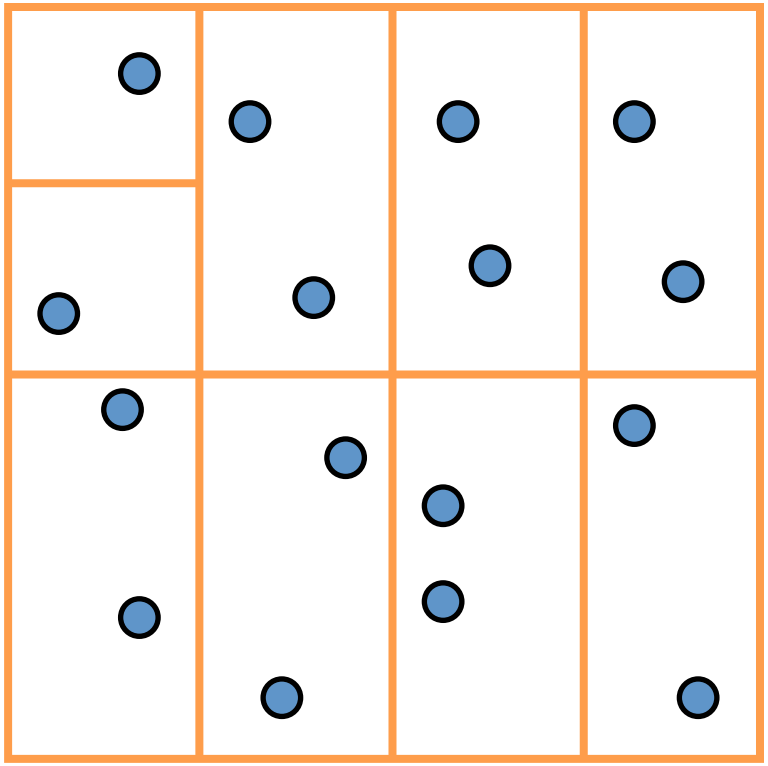
kD-Bäume



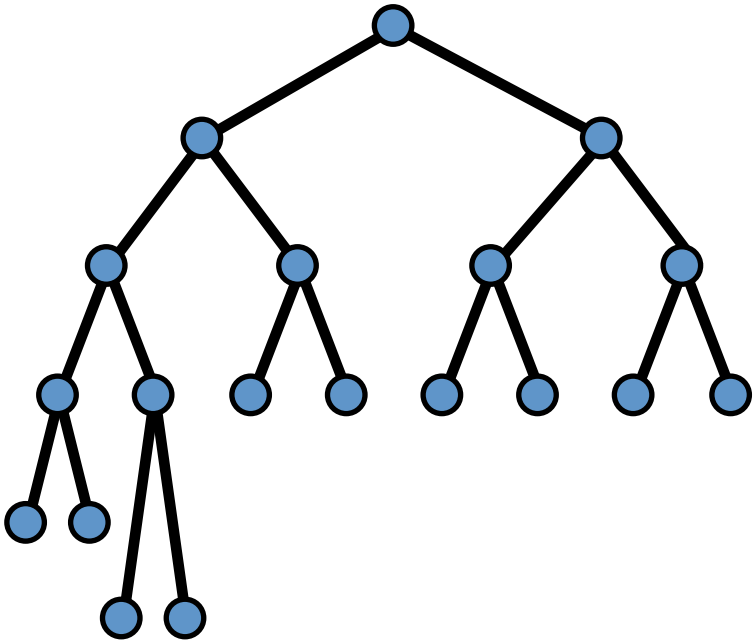


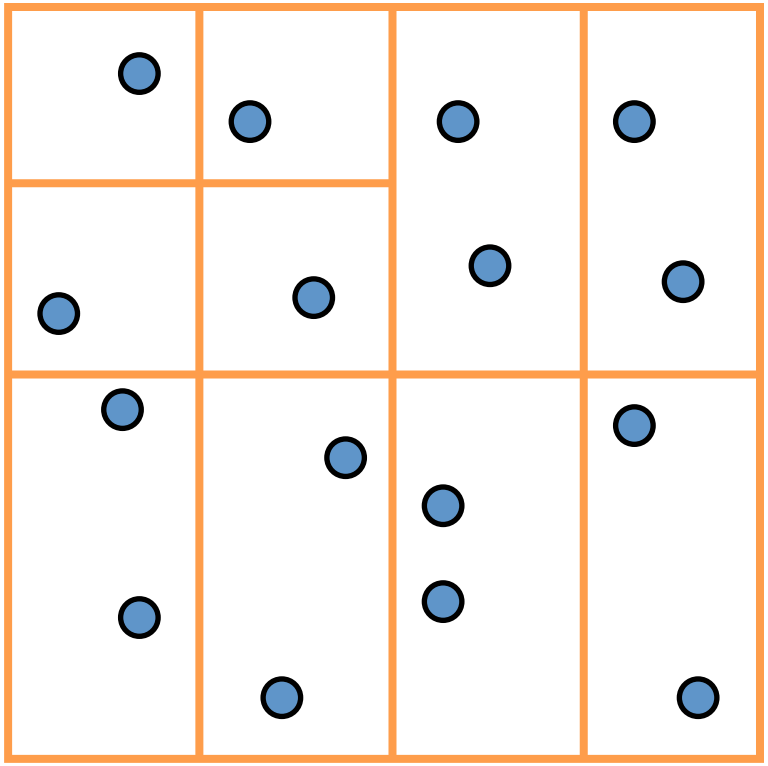
kD-Bäume



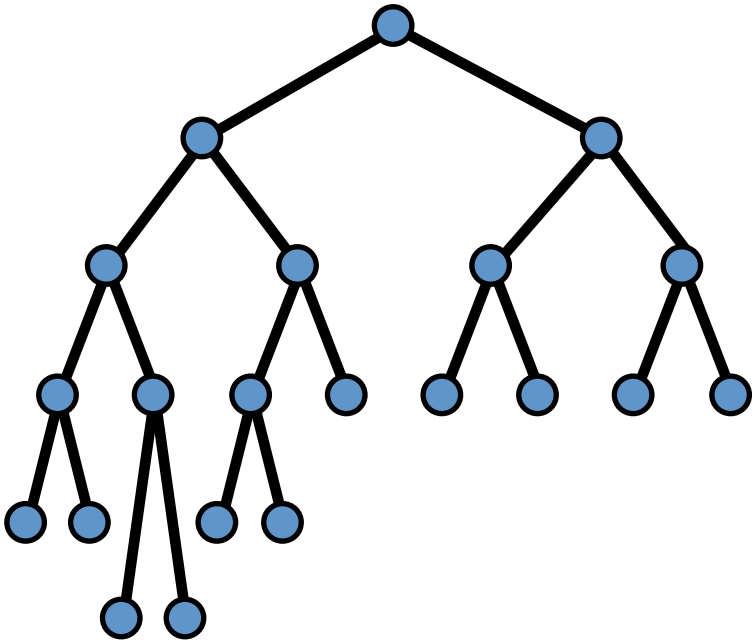


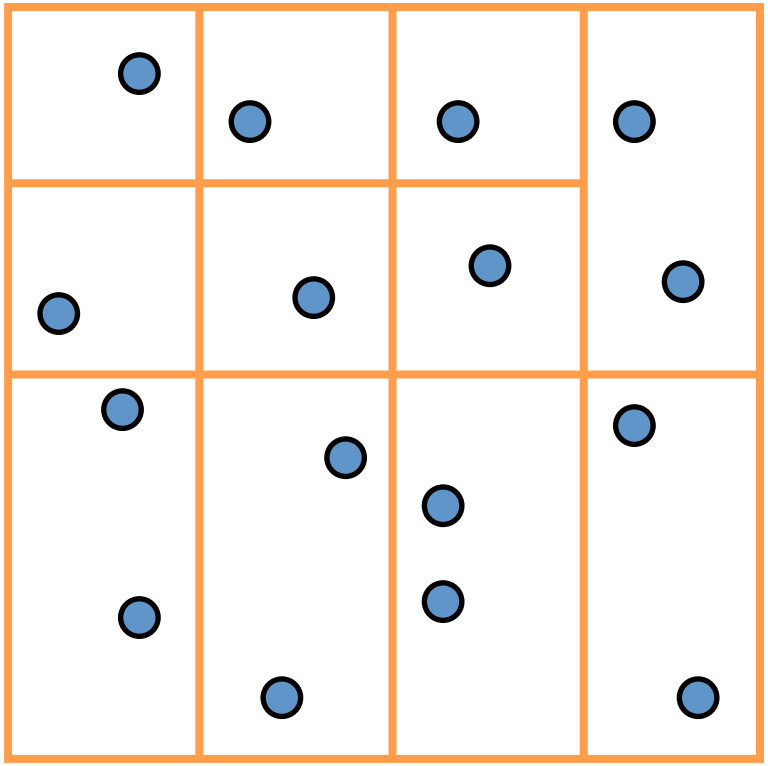
kD-Bäume



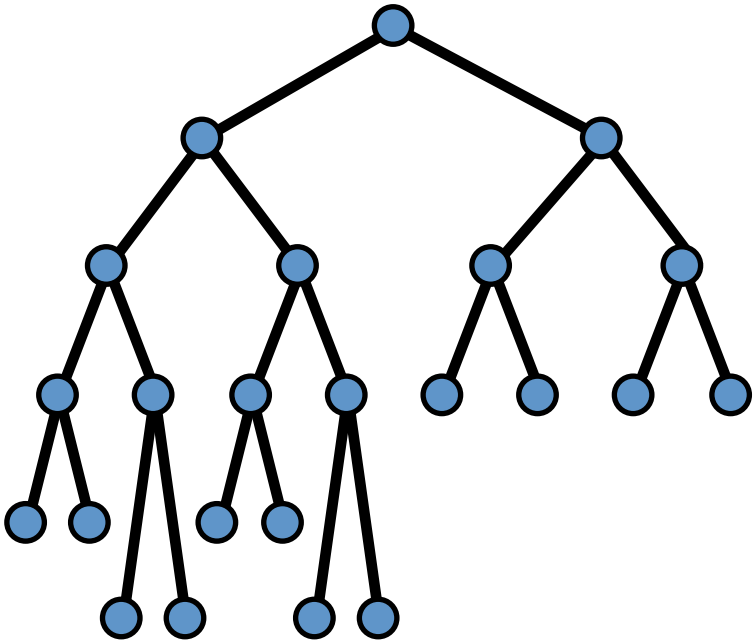


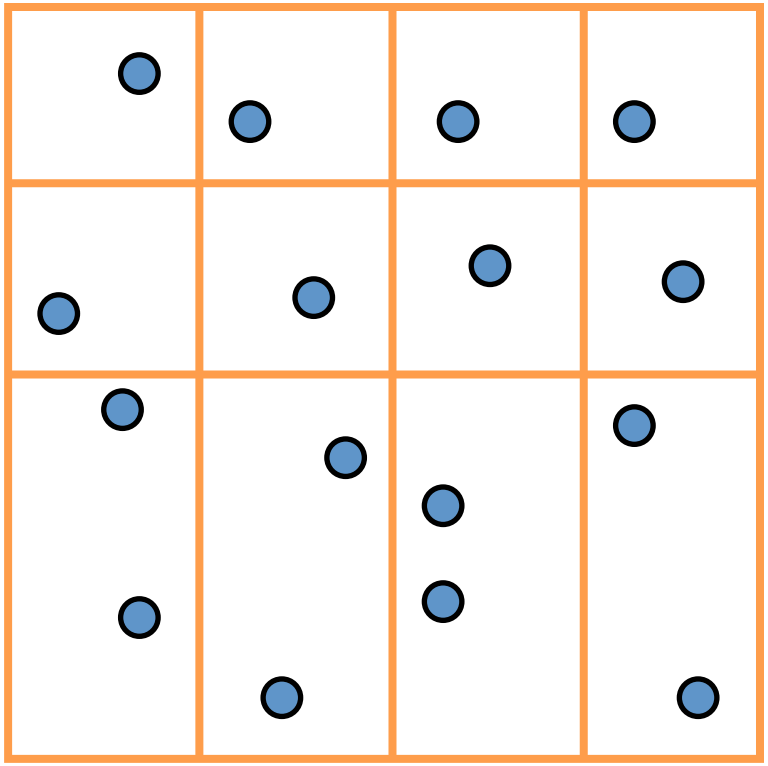
kD-Bäume



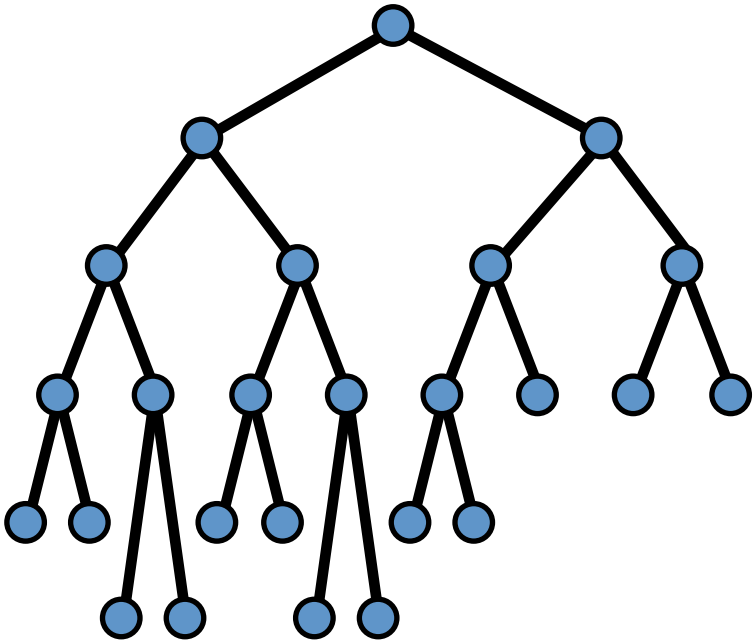


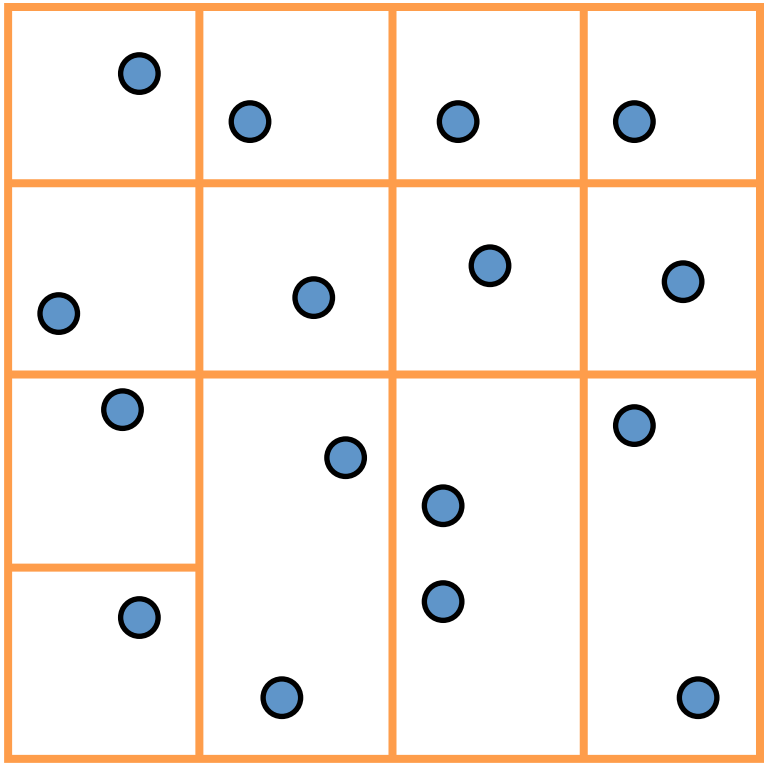
kD-Bäume



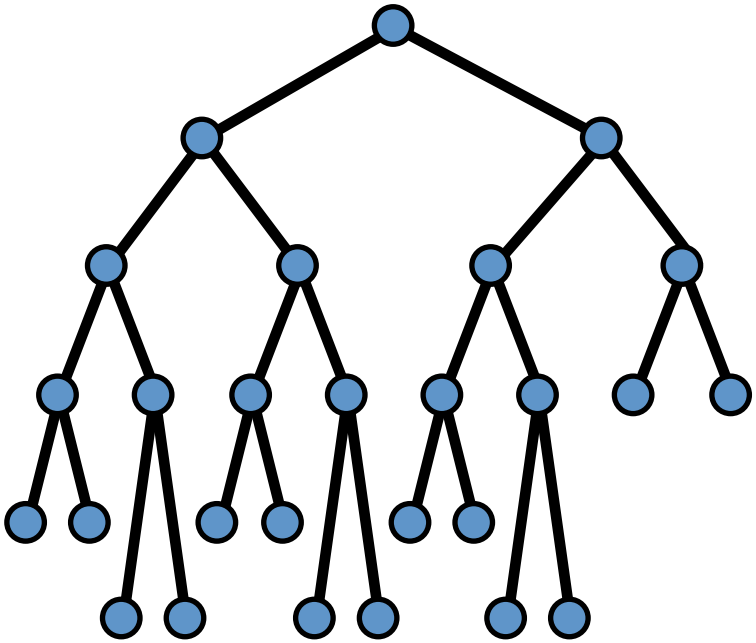


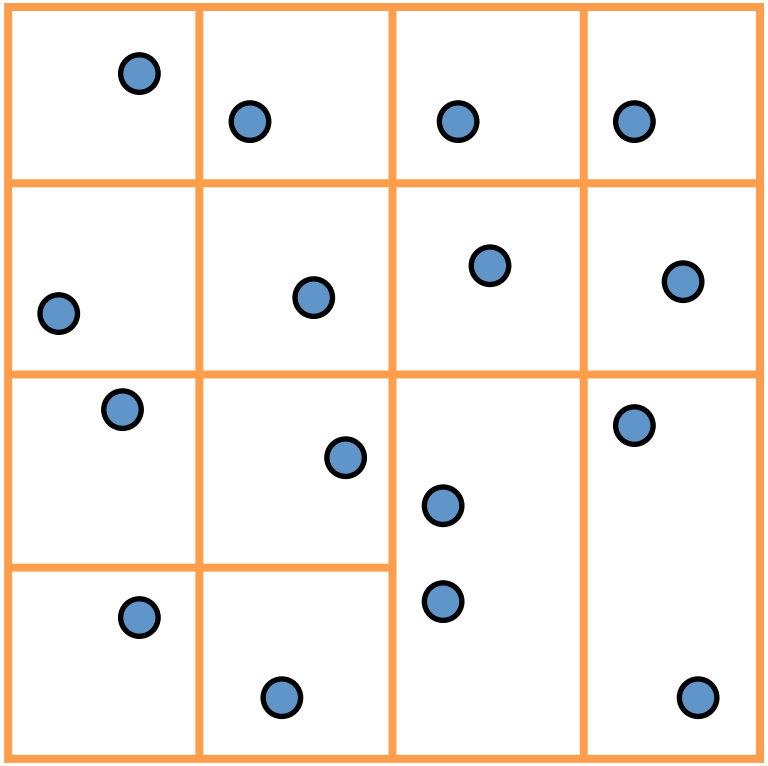
kD-Bäume



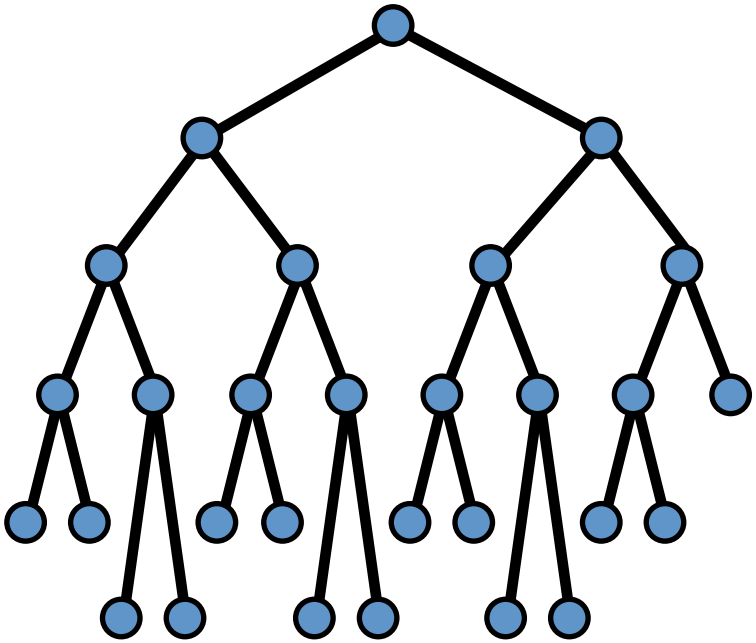


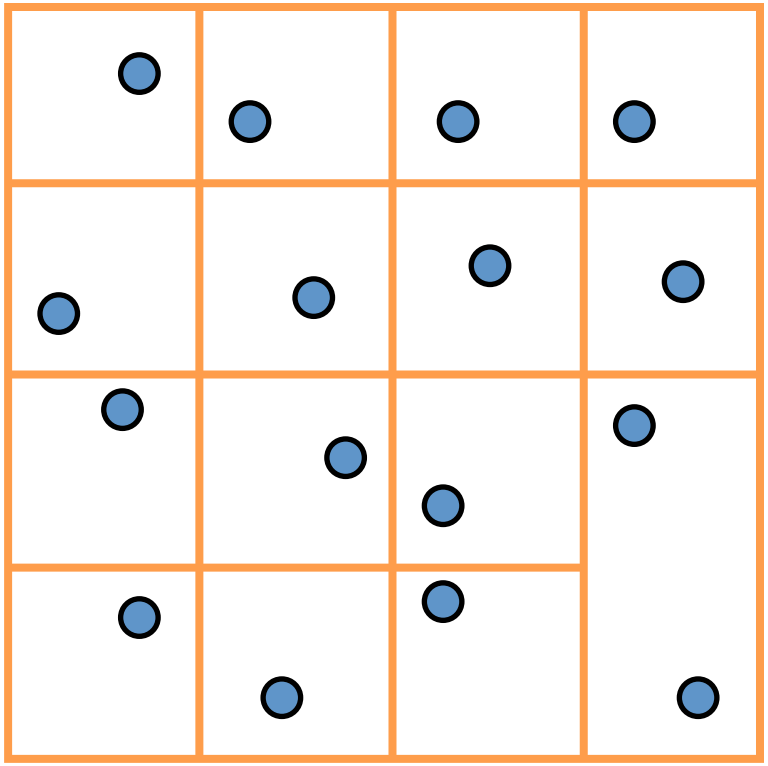
kD-Bäume



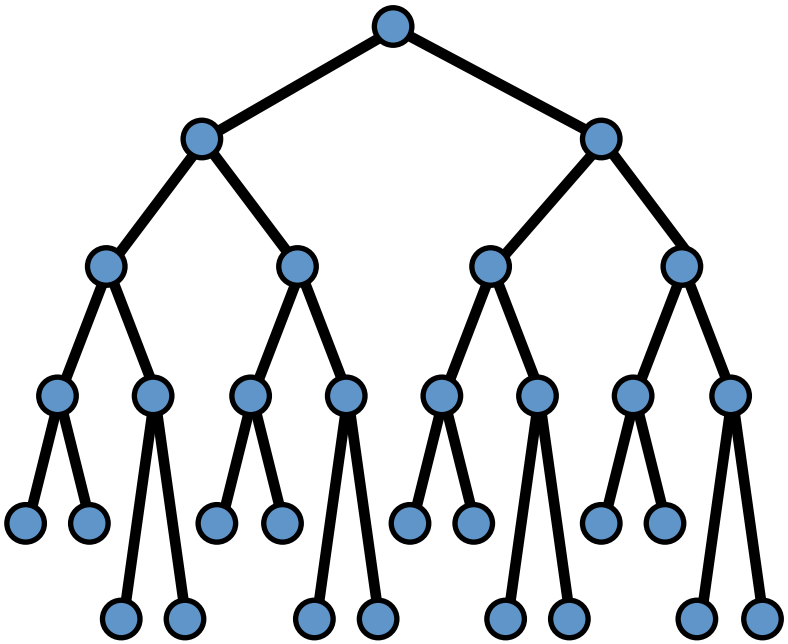


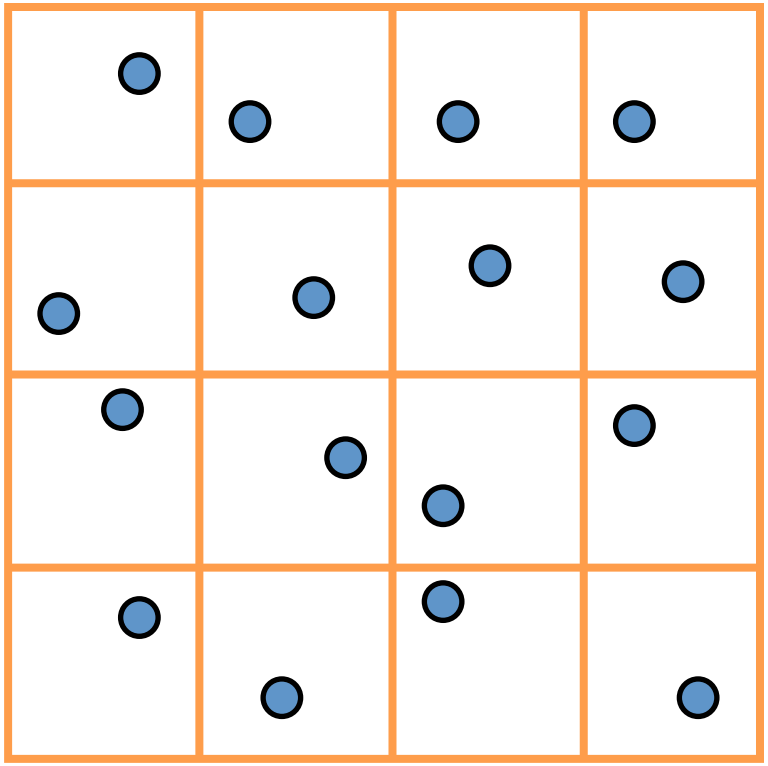
kD-Bäume





kD-Bäume





Weitere Baumstrukturen ...

- Baum-Strukturen beschleunigen die Suche in großen Datenmengen.
- Einfügeoperationen beeinflussen die Verteilung der Knoten im Baum.
- Löschungsoperationen können eine komplexe Umstrukturierung des Baumes bewirken.