INTERNET NETWORK ARCHITECTURES

Ausblick: Online Algorithmen

Übersicht

- Beispiele
- Notation
- Selbstorganisierende Suchstrukturen

Beipiele

Ski-Problem:

- Ein Paar Ski kann für 500 € gekauft oder für 50 € geliehen werden.
  Wie lange soll geliehen werden, bevor man sich für den Kauf entscheidet, wenn nicht bekannt ist, wie lange man noch Ski fährt?

- Optimale Strategie (falls Zukunft nicht bekannt): So lange Ski leihen, bis die Leihkosten gleich den Kaufkosten sind.

Beispiele

Geldwechselproblem:

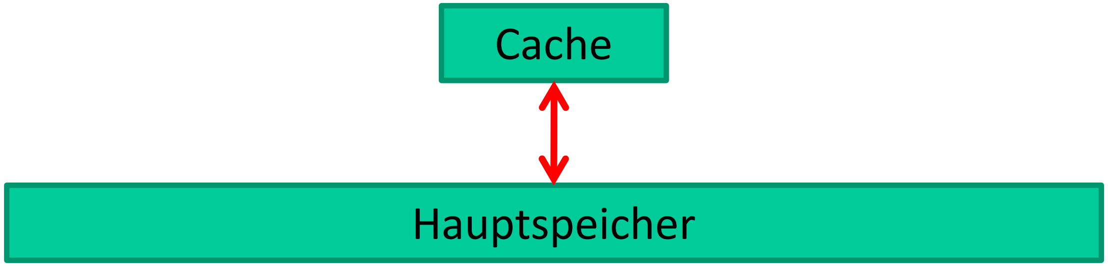
- Ein Geldbetrag (z.B. 10.000 €) soll in eine andere Währung (z.B. $) gewechselt werden. Zu welchem Zeitpunkt soll getauscht werden, wenn nicht bekannt ist, wie sich der Wechselkurs entwickelt?

INTERNET NETWORK ARCHITECTURES

Beispiele

Paging/Caching:

- Es soll ein zweistufiges Speichersystem verwaltet werden, das aus einem schnellen Speicher mit kleiner Kapazität und einem langsamen Speicher mit großer Kapazität besteht. Dabei müssen Anfragen auf Speicherseiten bedient werden. Welche Seiten hält man im Cache, um die Anzahl der Cache-Misses zu minimieren?



Beispiele

Scheduling:

- Jobs mit im Voraus bekannter Bearbeitungsdauer treffen hintereinander ein und sollen von m Maschinen bearbeitet werden. Die Jobs müssen dabei unmittelbar einer bestimmten Maschine zugeordnet werden. Hier gibt es verschiedene Optimierungsziele, z.B. die Minimierung der Gesamtbearbeitungszeit.

Online Problem

Online Problem:

- Statt einer vollständig gegebenen Eingabe I haben wir jetzt eine Eingabesequenz $\sigma=(\sigma(1), \sigma(2),\ldots, \sigma(t))$, in der die Eingabeteile $\sigma(i)$ erst nach und nach an den Online Algorithmus übergeben werden.
- Nach jedem $\sigma(i)$ muss der Online Algorithmus eine Entscheidung treffen, bevor er $\sigma(i+1)\ldots\sigma(t)$ gesehen hat. Diese Entscheidung kann (im Standard-Online-Modell) nicht wieder zurück-genommen werden.

Notation

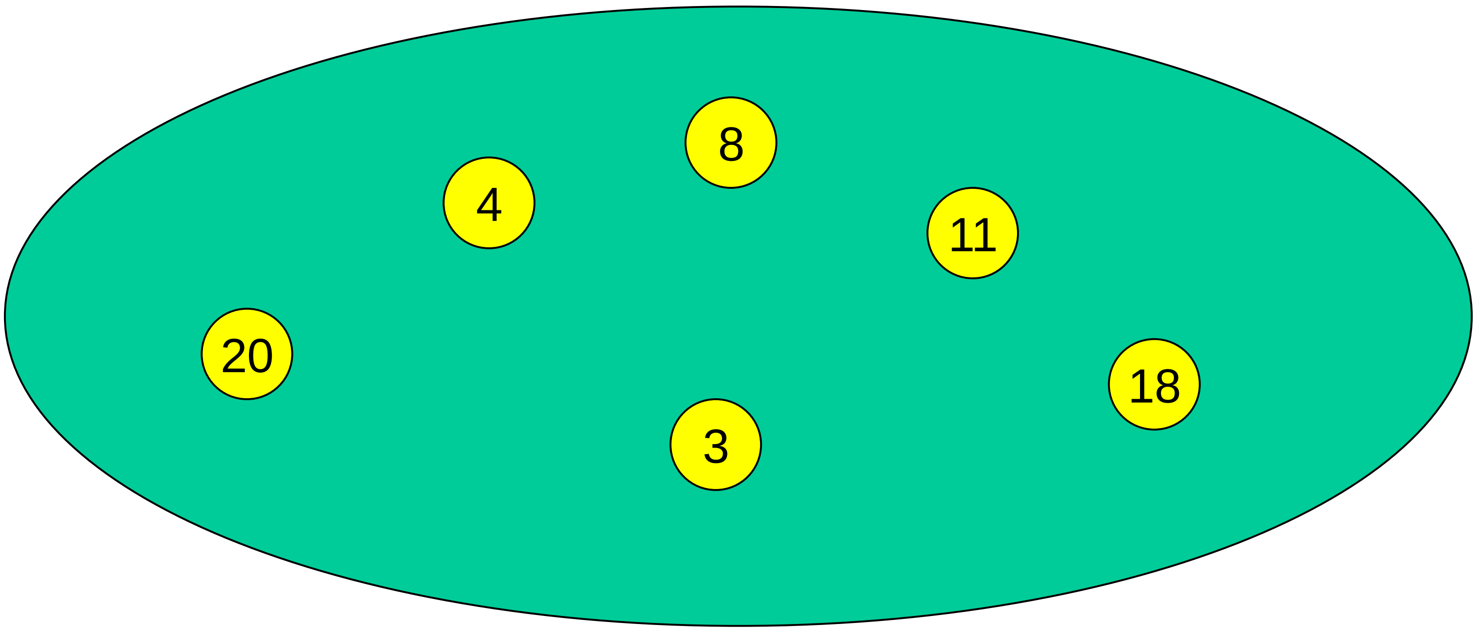
Definition: Sei Π ein Optimierungsproblem und A ein Online Algorithmus für Π. Für eine beliebige Eingabesequenz $\sigma=(\sigma(1), \sigma(2),\ldots, \sigma(t))$ seien $A(\sigma)$ die Kosten von A für σ. A heißt c-kompetitiv, wenn es einen von t unabhängigen Parameter a gibt, so dass für alle Eingabesequenzen σ gilt:

$$A(\sigma) \leq c \cdot OPT(\sigma) + a$$

Übersicht

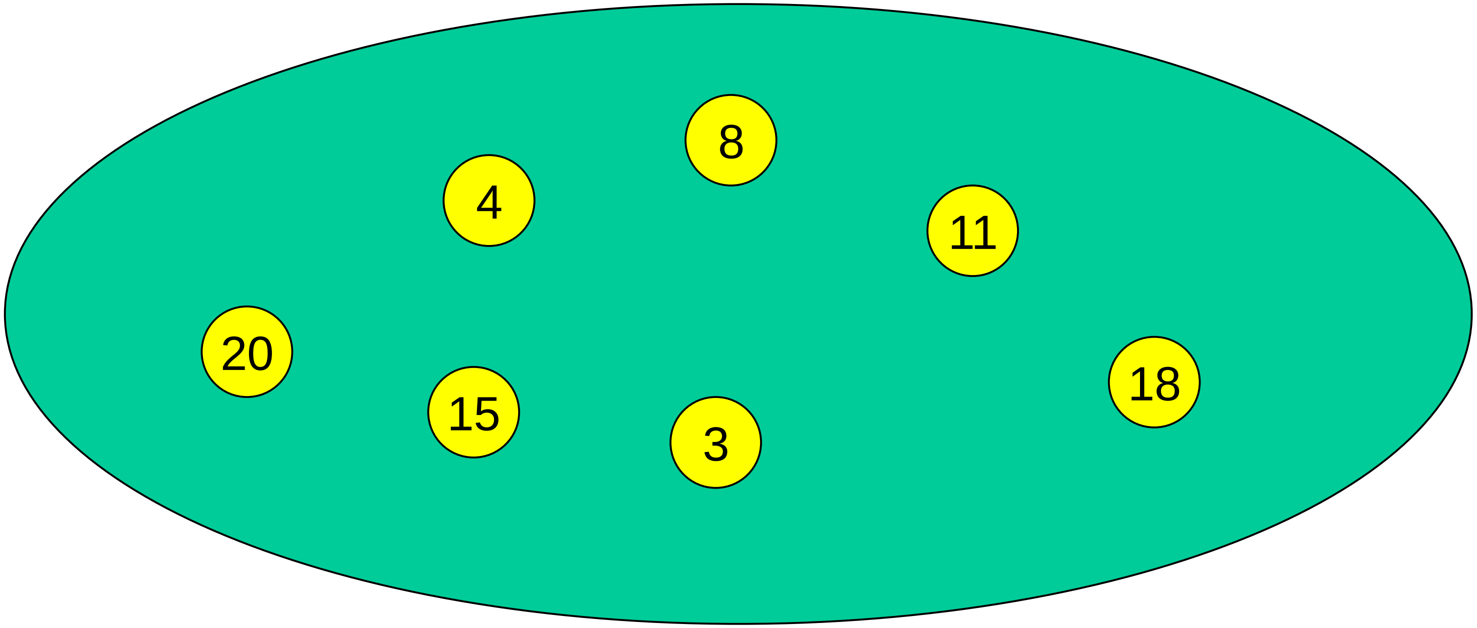
- Beispiele
- Notation
- Selbstorganisierende Suchstrukturen

Wörterbuch

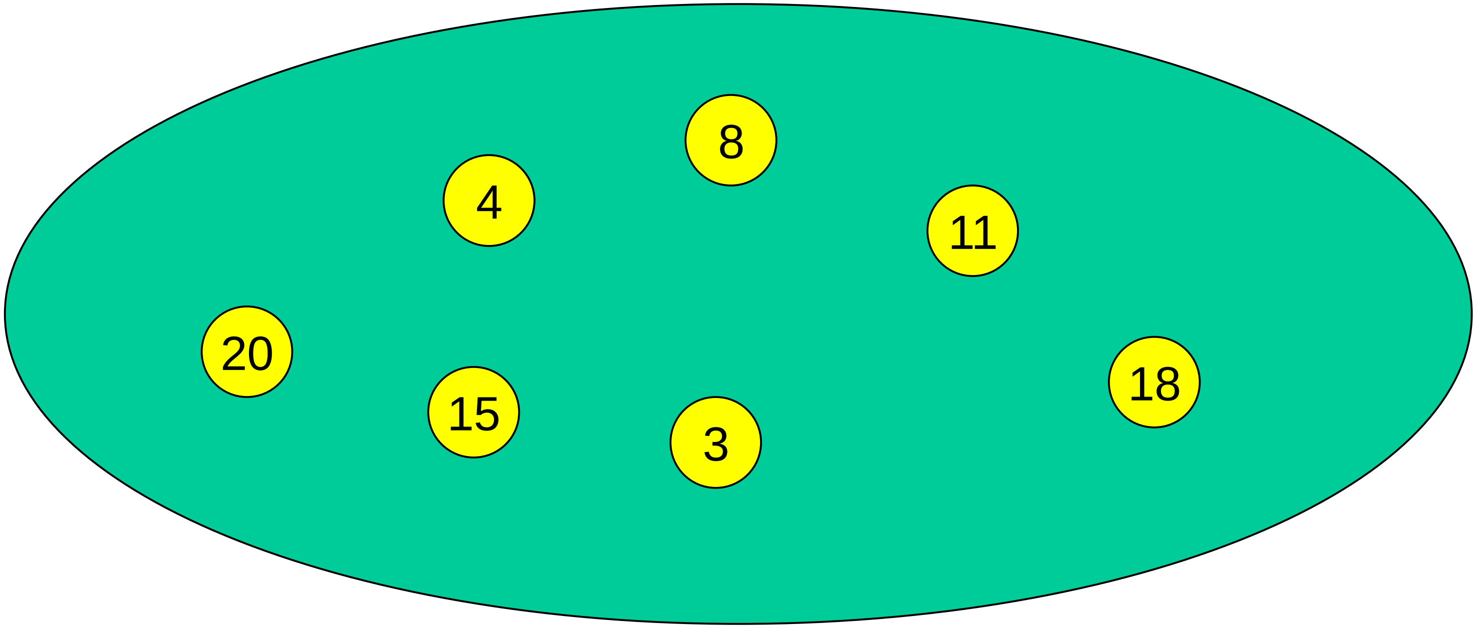


Wörterbuch

einfügen(15)

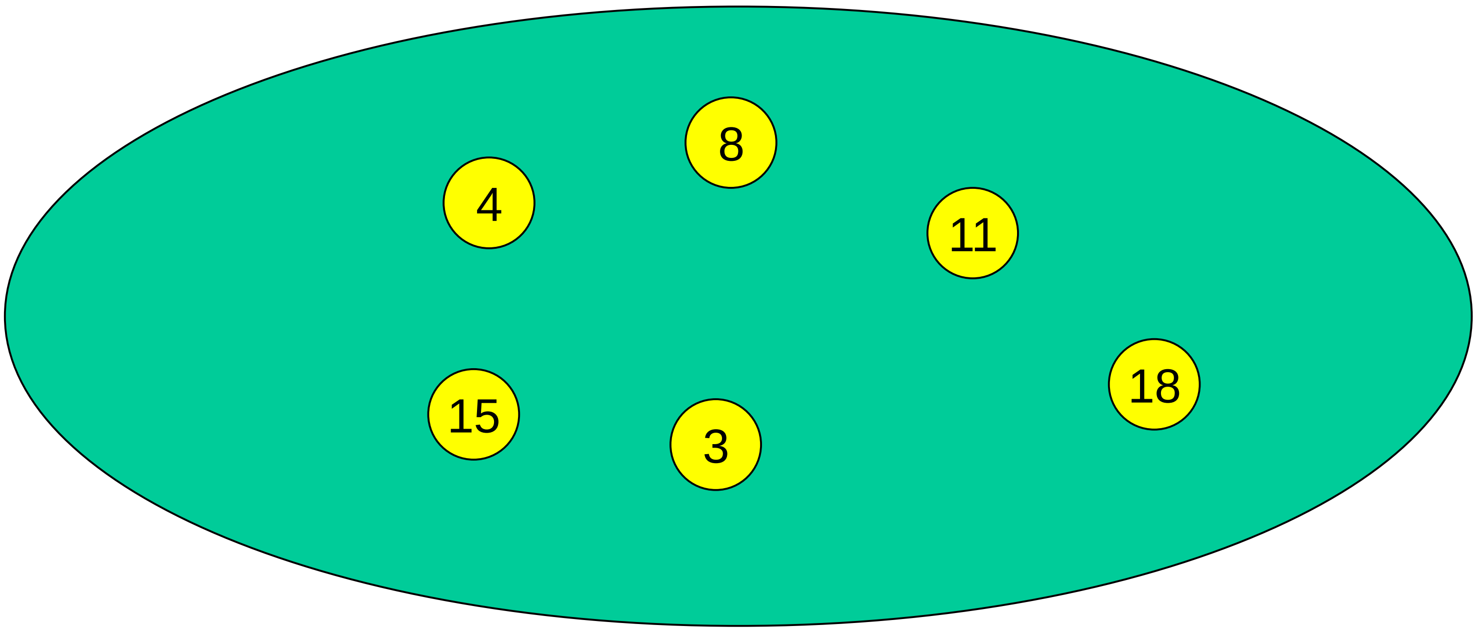


löschen(20)



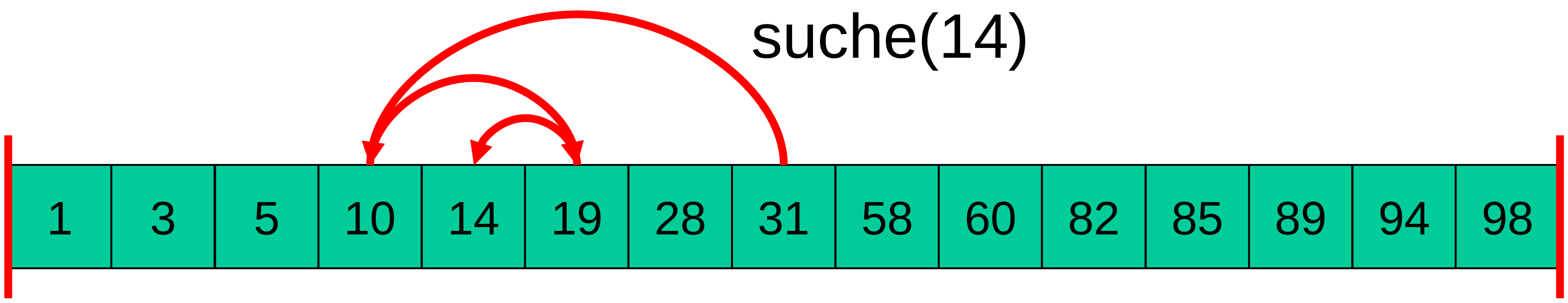
Wörterbuch

suche(8) ergibt 8



Statische Suchstruktur

1. Speichere Elemente in sortiertem Feld.

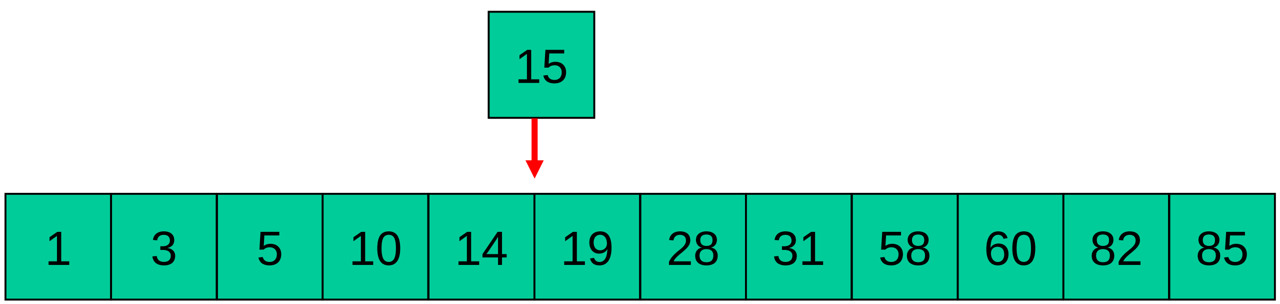


suche: mittels binäre Suche O(log n) Zeit

Dynamische Suchstruktur

Einfügen und löschen Operationen:

Sortiertes Feld schwierig zu aktualisieren!



Worst case: $\Theta(n)$ Zeit

Suchstruktur als Online Problem

Problem: Ständig wird 19 angefragt

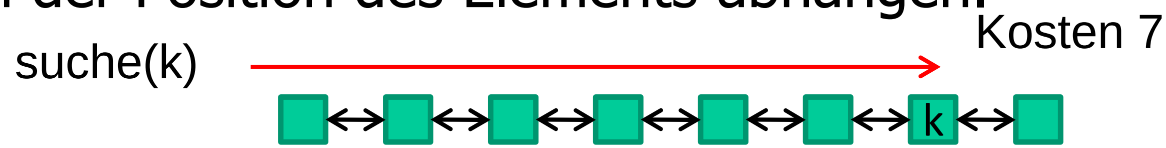
Mögliche Lösung: Sortiere Elemente nach Zugriffshäufigkeit, aber Zugriffshäufigkeit nicht von vornherein bekannt. Was ist online möglich?

Suchstruktur als Online Problem

Zentrale Frage: für welche einfügen, löschen und suche Operationen weichen wir für jede gegebene Operationsfolge σ so wenig wie möglich ab von der Laufzeit eines optimalen Offline Algorithmus, der alle Anfragen zu Beginn an kennt und damit eine optimale Strategie für die Anordnung und Umordnung der Elemente z.B. in einer Liste berechnen kann?

Selbstorganisierende Liste

- Sei eine beliebige suche-Anfragefolge σ gegeben.
- Bei einem Zugriff auf ein Listenelement entstehen Kosten, die von der Position des Elements abhängen.



- Das angefragte Element kann nach seiner Anfrage an eine beliebige Position weiter vorne in der Liste bewegt werden (mit keinen Zusatzkosten).

- Außerdem ist es möglich, zwei benachbarte Elemente mit Kosten 1 zu vertauschen.

Selbstorganisierende Liste

Die folgenden Online Strategien bieten sich an:

- Move-to-Front (MTF): Das angefragte Element wird an den Kopf der Liste bewegt.
- Transpose: Das angefragte Element wird mit dem Vorgänger in der Liste vertauscht.
- Frequency-Count (FC): Verwalte für jedes Element der Liste einen Zähler, der mit Null initialisiert wird. Bei jeder Anfrage wird der Zähler um eins erhöht. Die Liste wird nach jeder Anfrage gemäß nicht steigenden Zählerständen sortiert.

Interessanterweise ist MTF die beste Strategie.

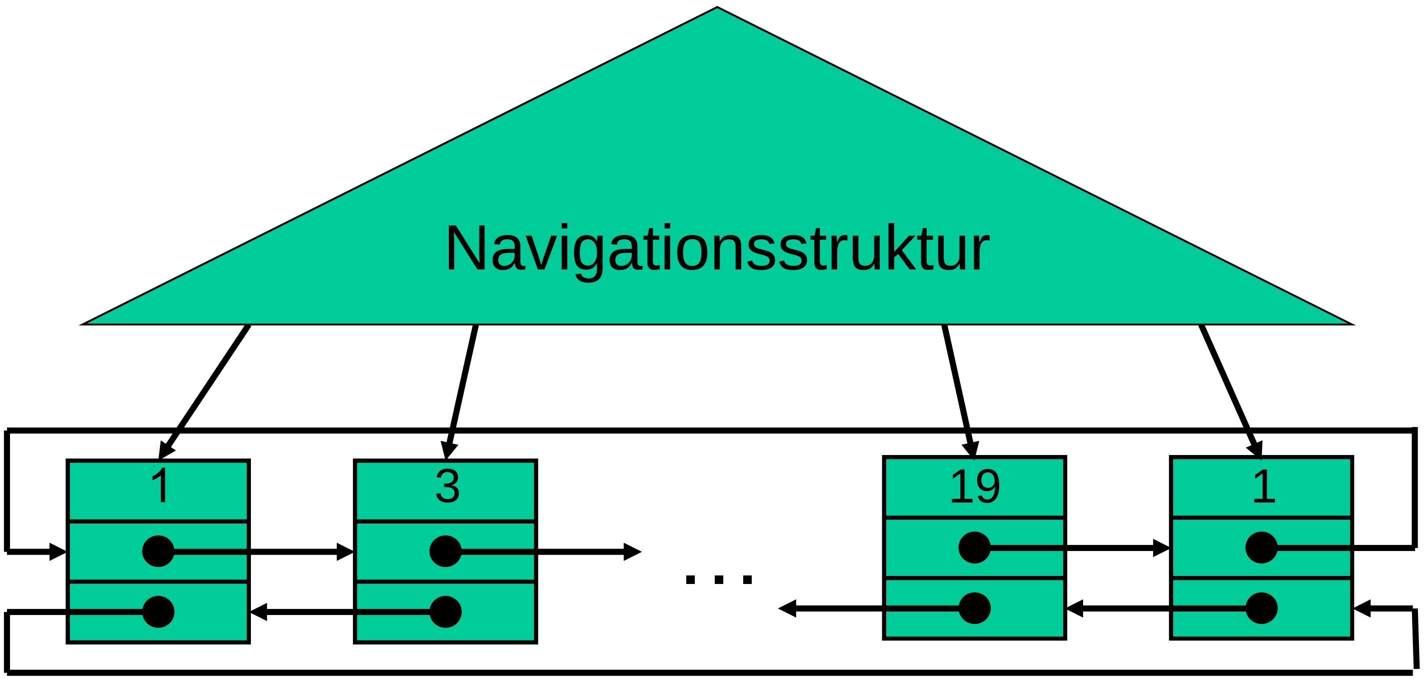
Suchstruktur als Online Problem

Selbstorganisierende Liste hat auch im online Setting das Problem, dass Einfügen, Löschen und Suche im worst case $\Theta(n)$ Zeit kosten

Idee: Wenn Suche effizient zu implementieren wäre, dann auch alle anderen Operationen

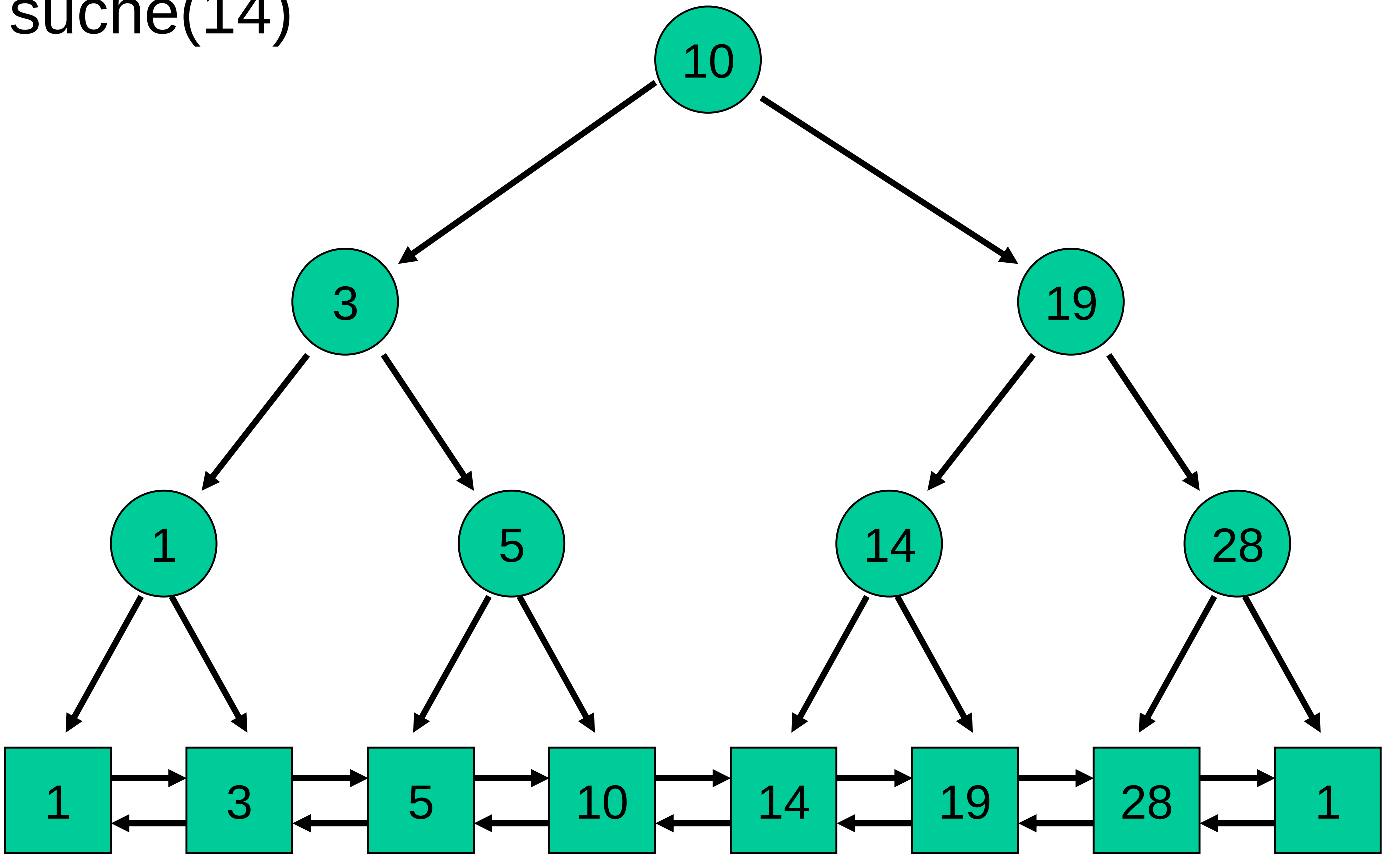
Suchstruktur

Idee: füge Navigationsstruktur hinzu, die Suche effizient macht

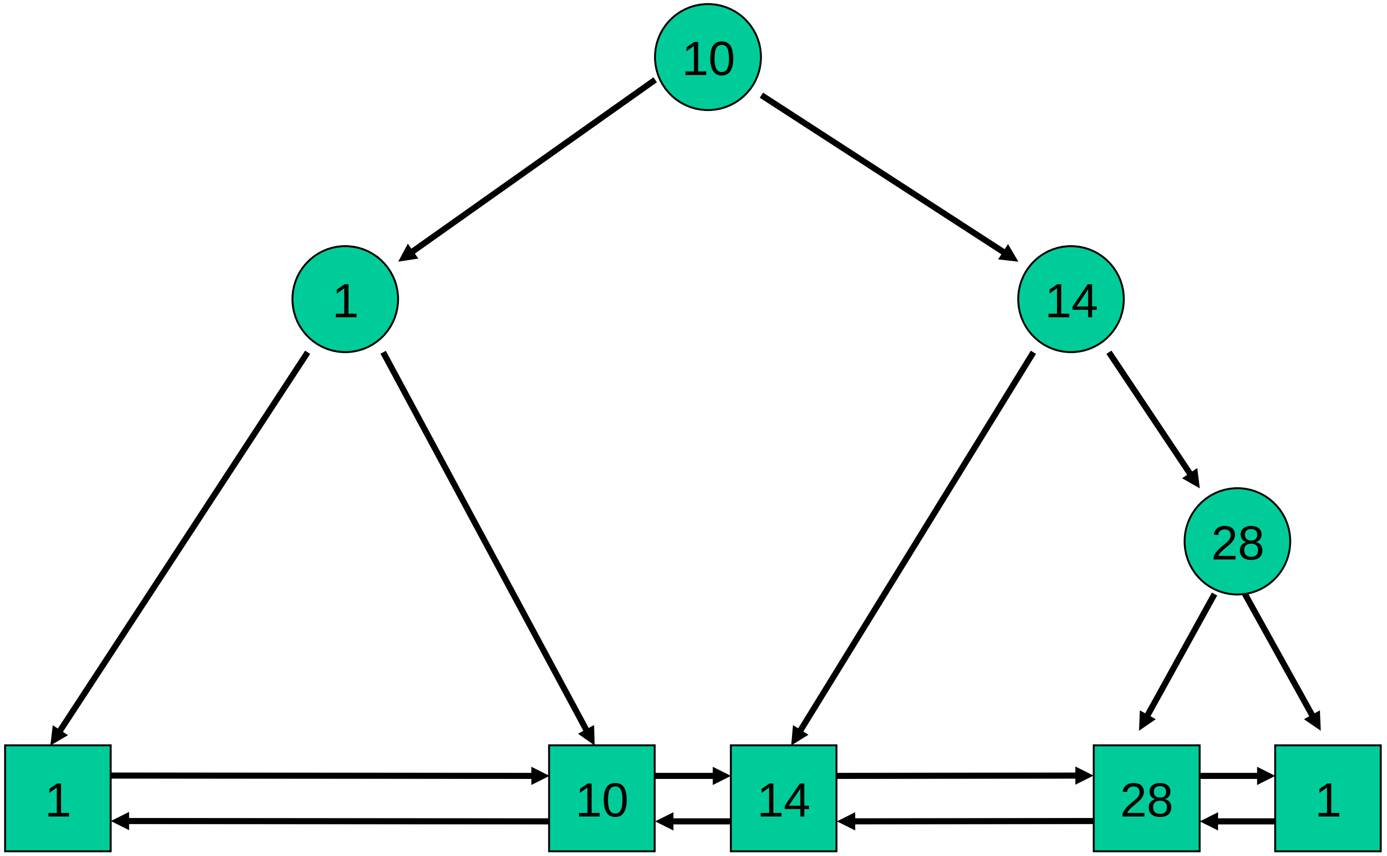


Binärer Suchbaum (ideal)

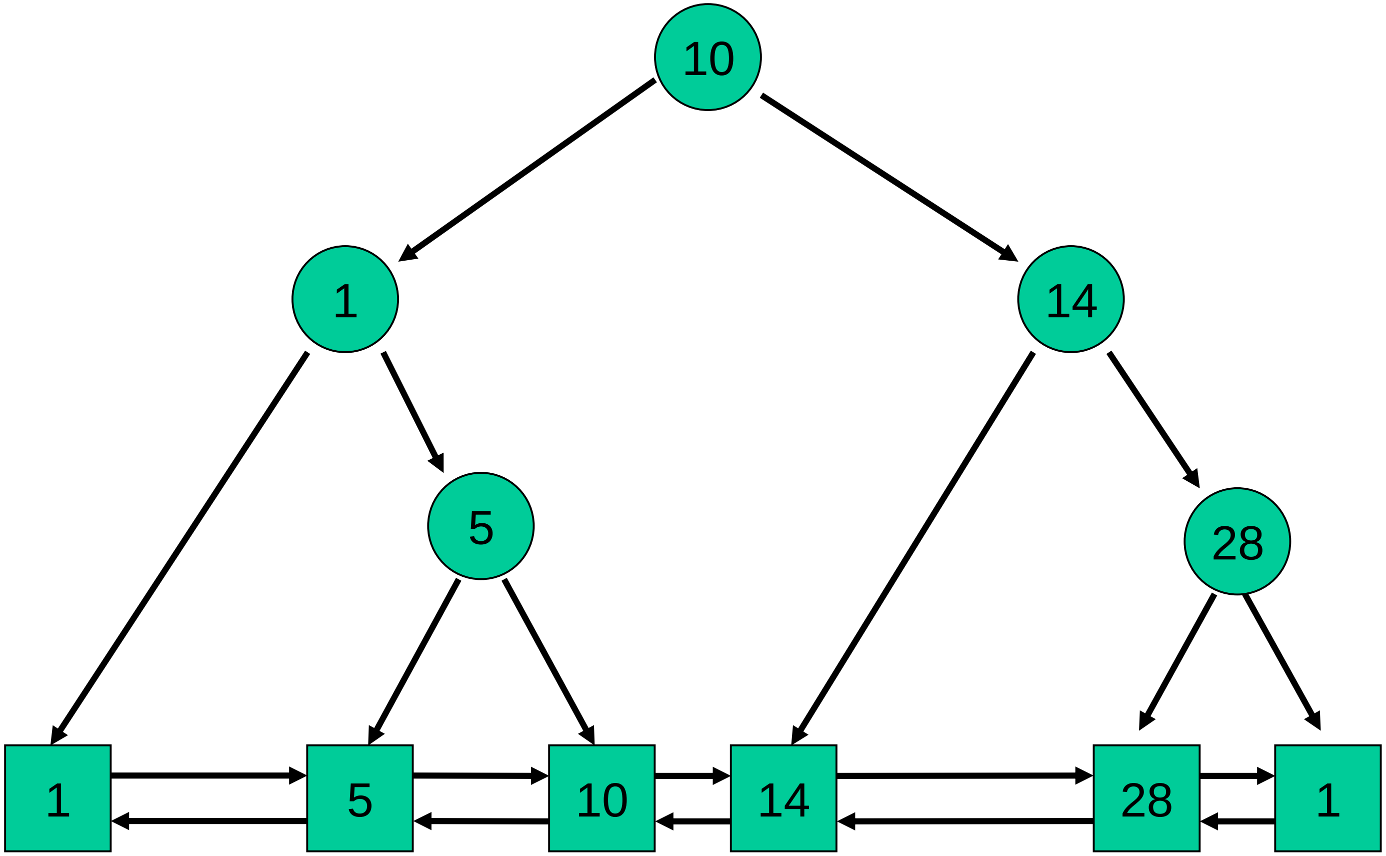
suche(14)



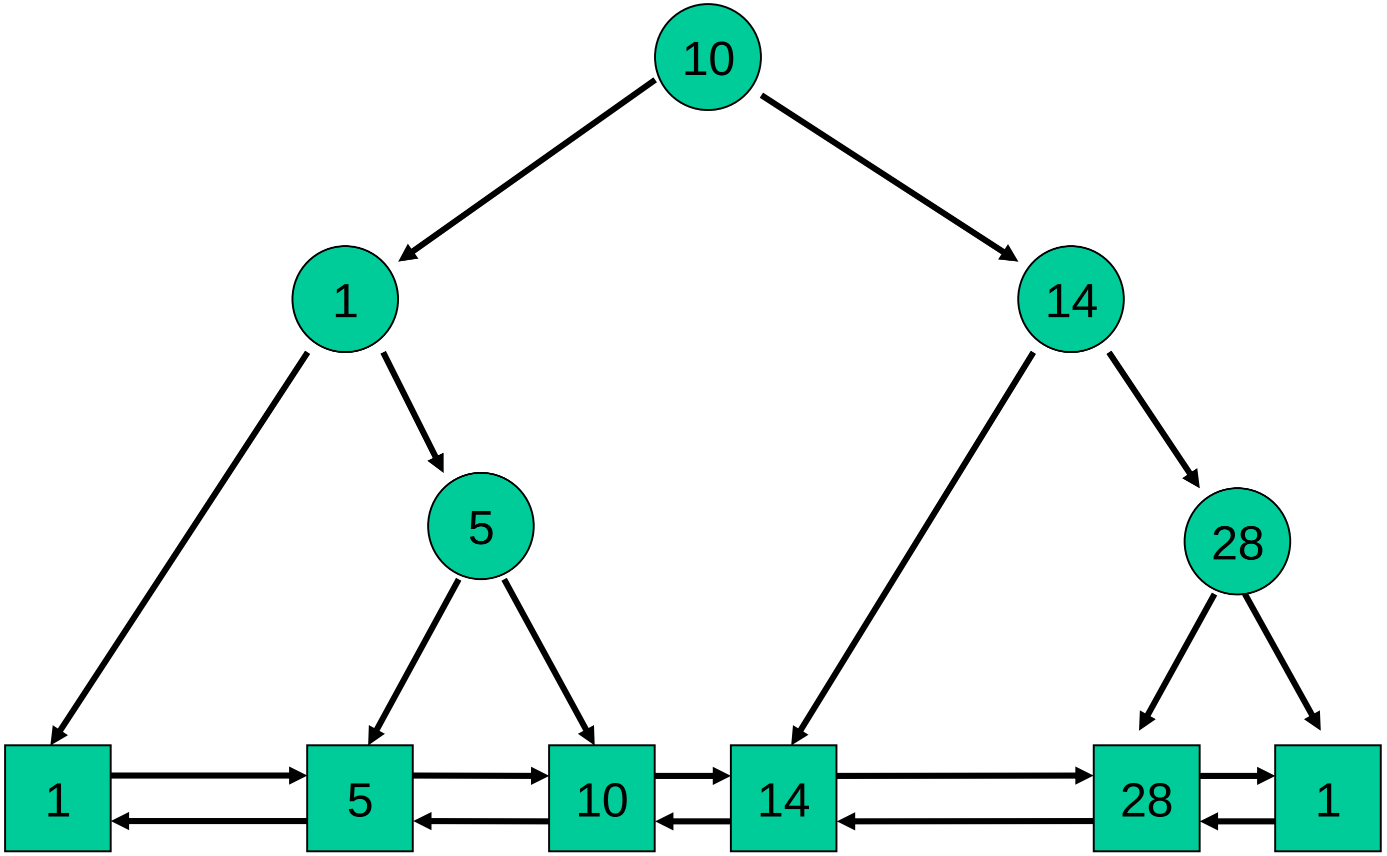
einfügen(5)



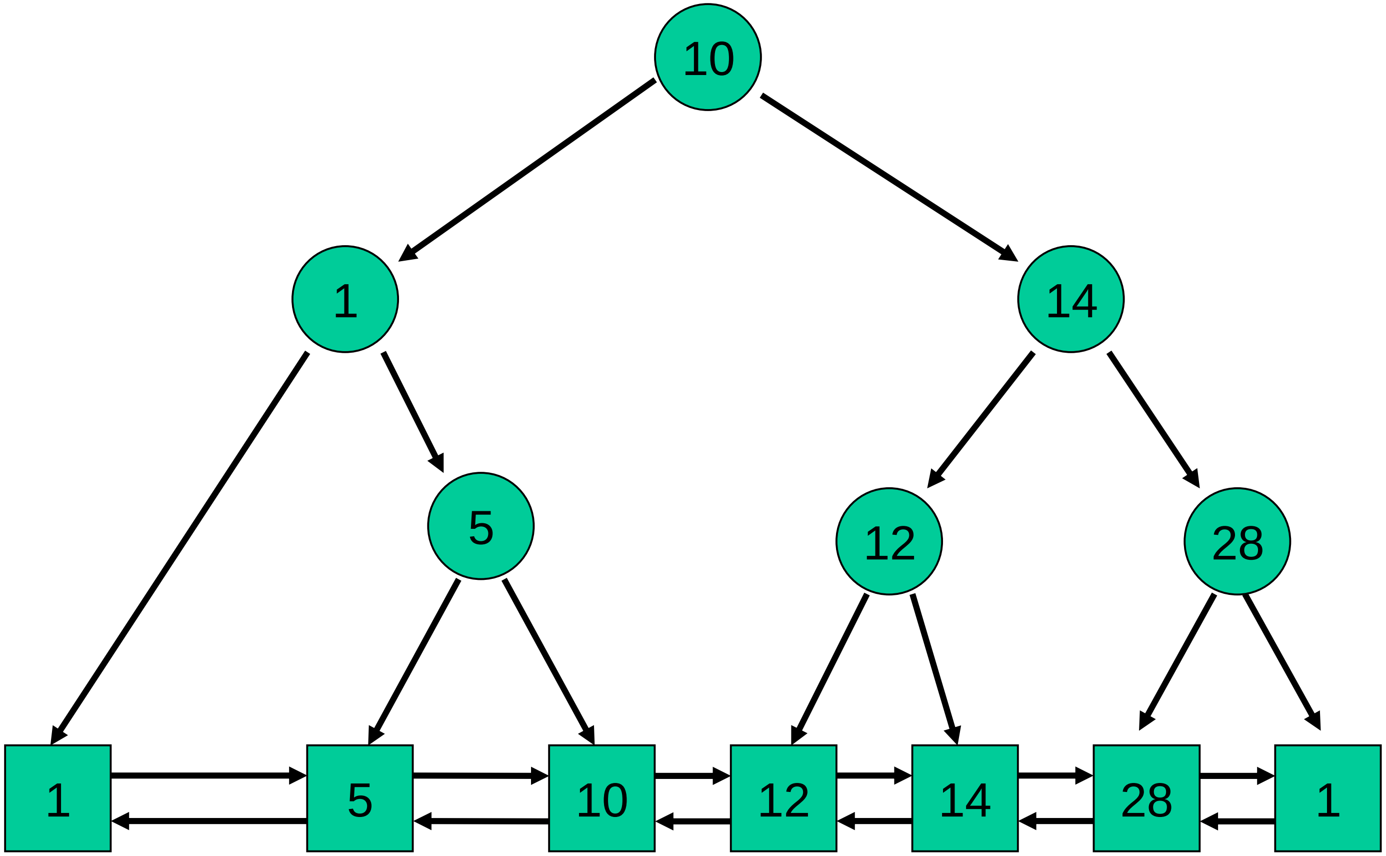
einfügen(5)



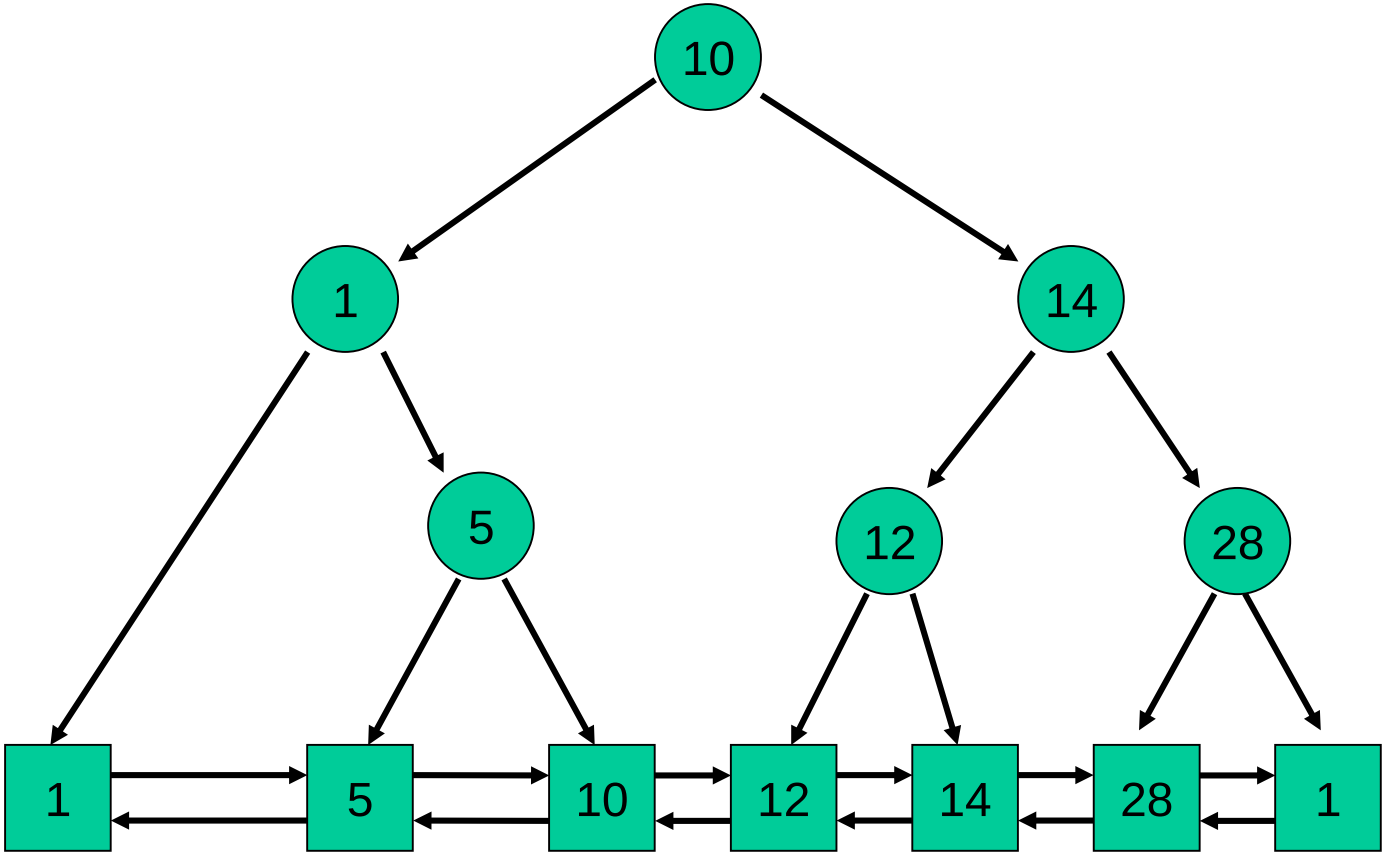
einfügen(12)



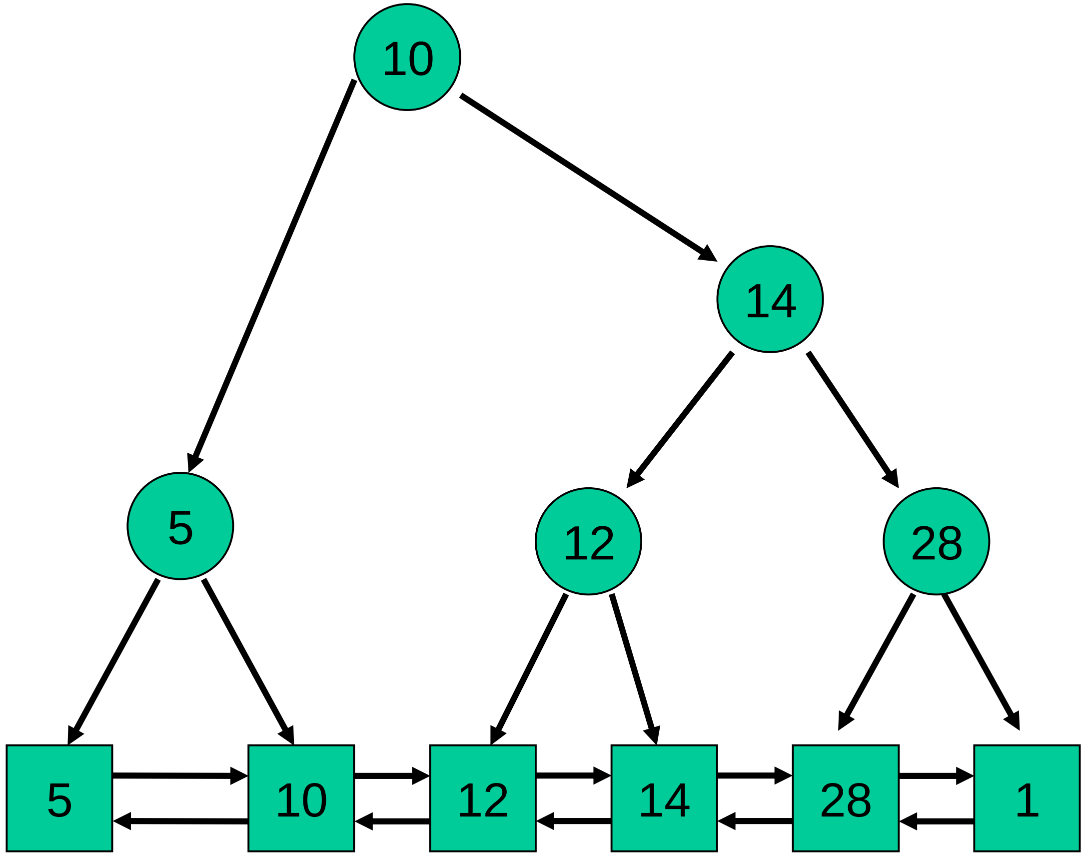
einfügen(12)



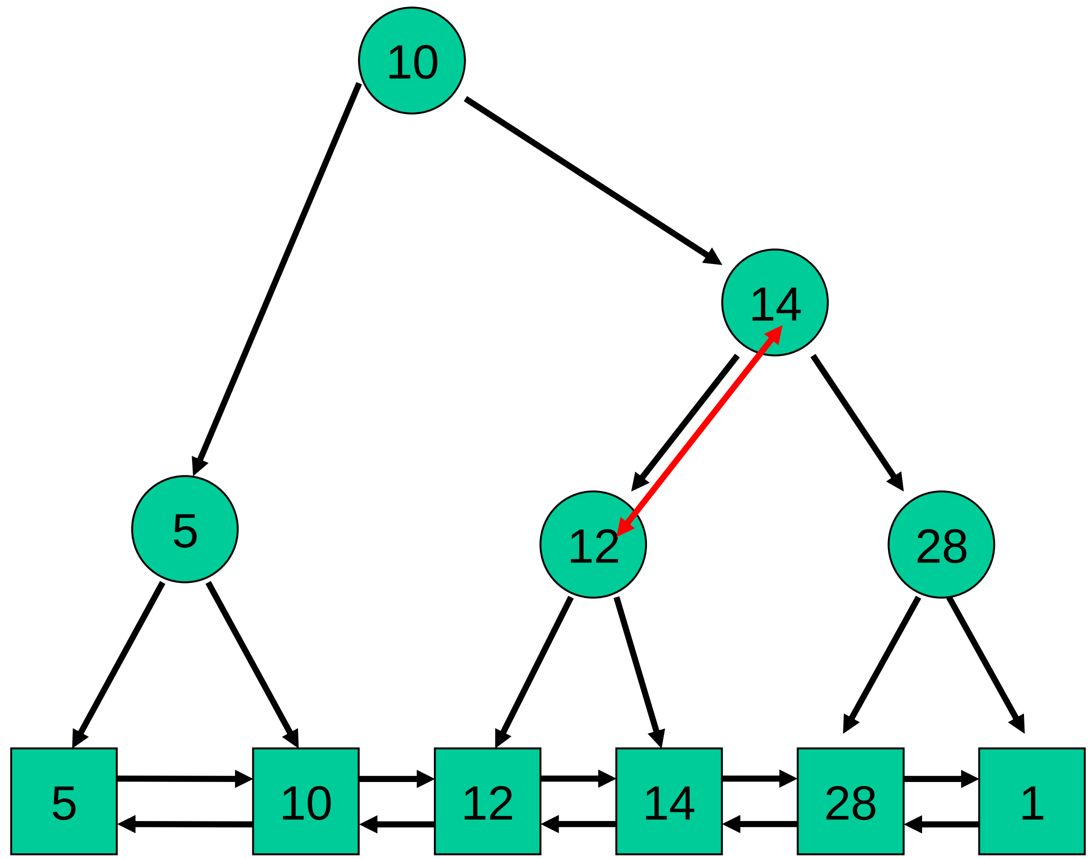
löschen(1)



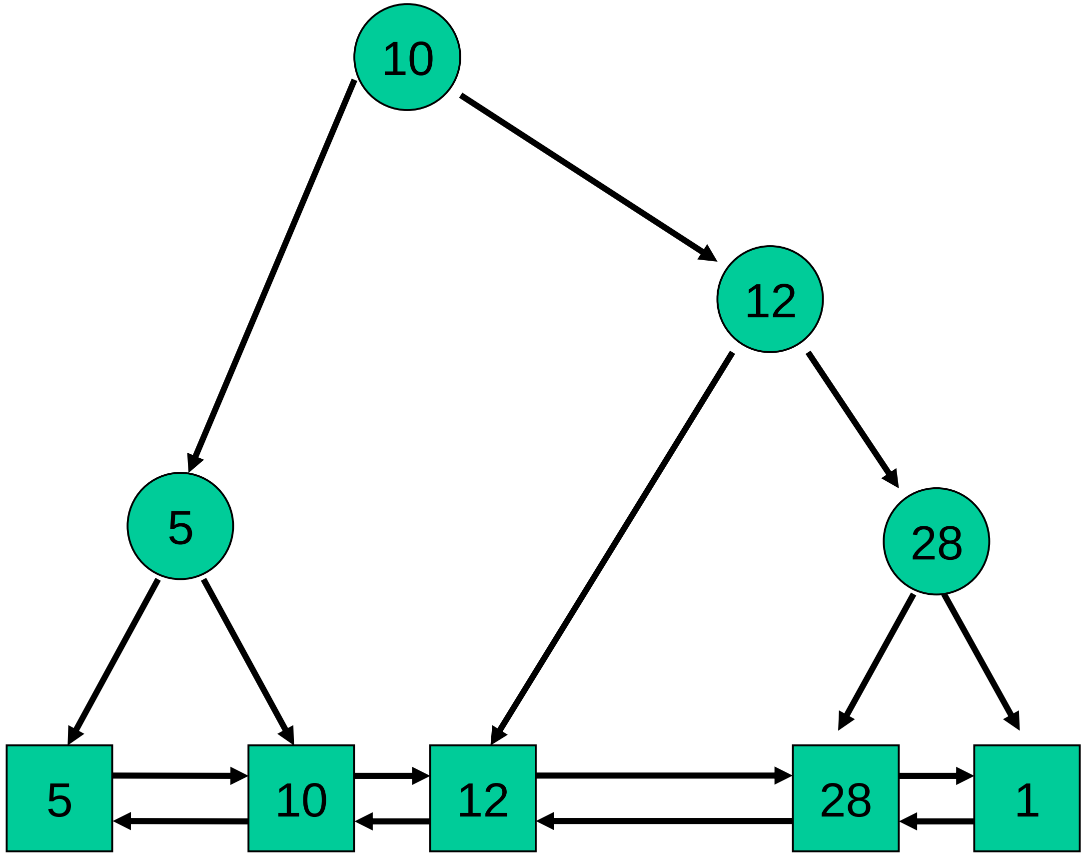
löschen(1)



löschen(14)



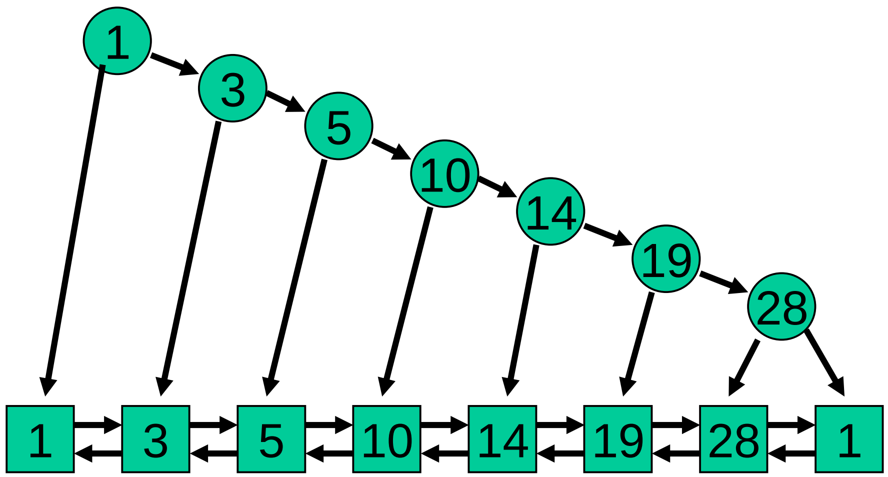
löschen(14)



Binärbaum

Problem: Binärbaum kann entarten!

Beispiel: Zahlen werden in sortierter Folge eingefügt



Suche benötigt $\Theta(n)$ Zeit im worst case

Lösung: Splay Baum

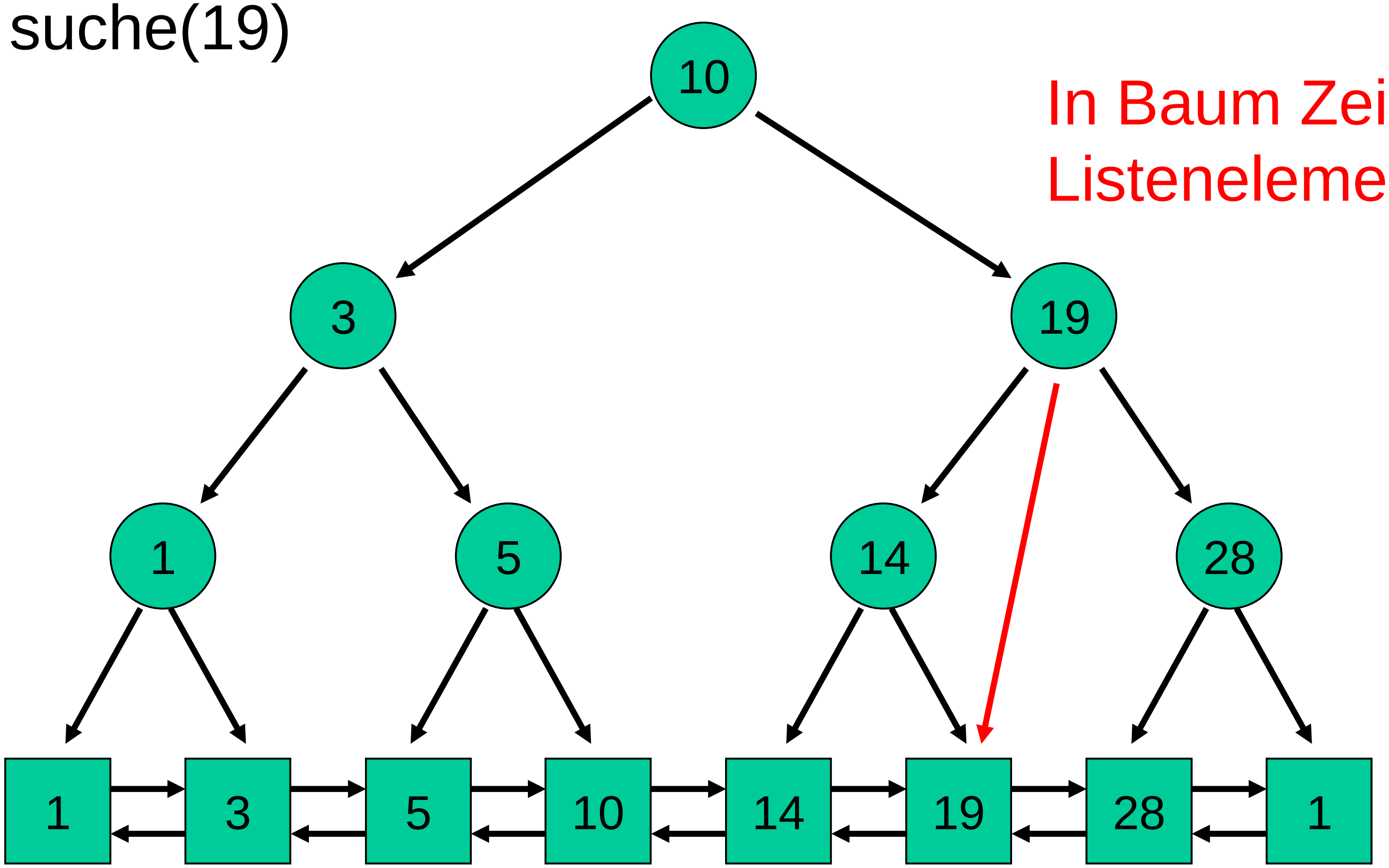
Splay-Baum

Üblicherweise: Implementierung als interner Suchbaum (d.h. Elemente direkt integriert in Baum und nicht in extra Liste)

Hier: Implementierung als externer Suchbaum (wie beim Binärbaum oben)

Splay-Baum

suche(19)



Splay-Baum

Ideen:

1. Im Baum Zeiger auf Listenelemente
2. Bewege Schlüssel von zugegriffenem Element immer zur Wurzel

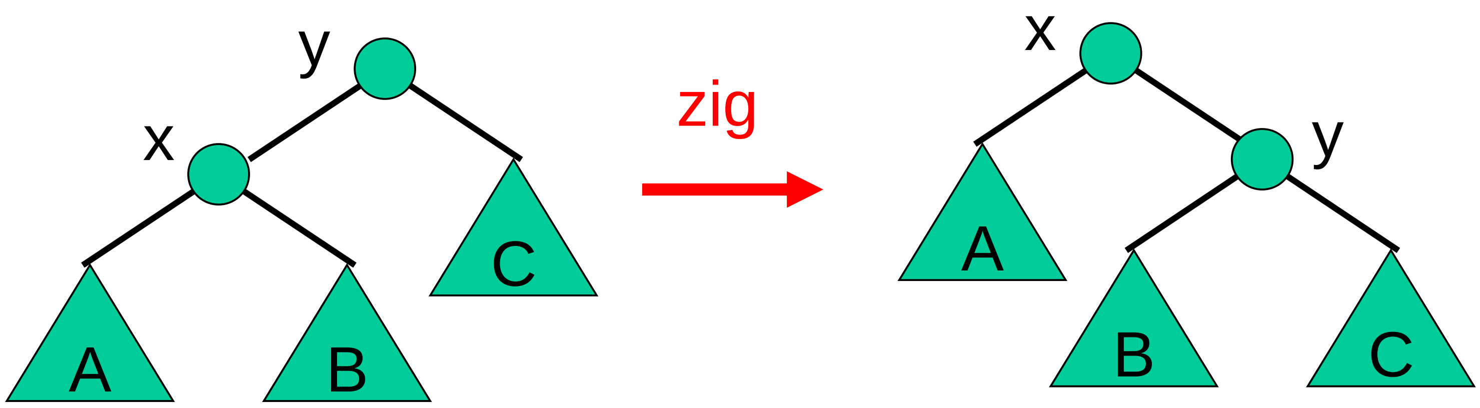
Wie: über Splay-Operation

Splay-Operation

Bewegung von Schlüssel x nach oben:

Wir unterscheiden zwischen 3 Fällen.

1a. x ist Kind der Wurzel:

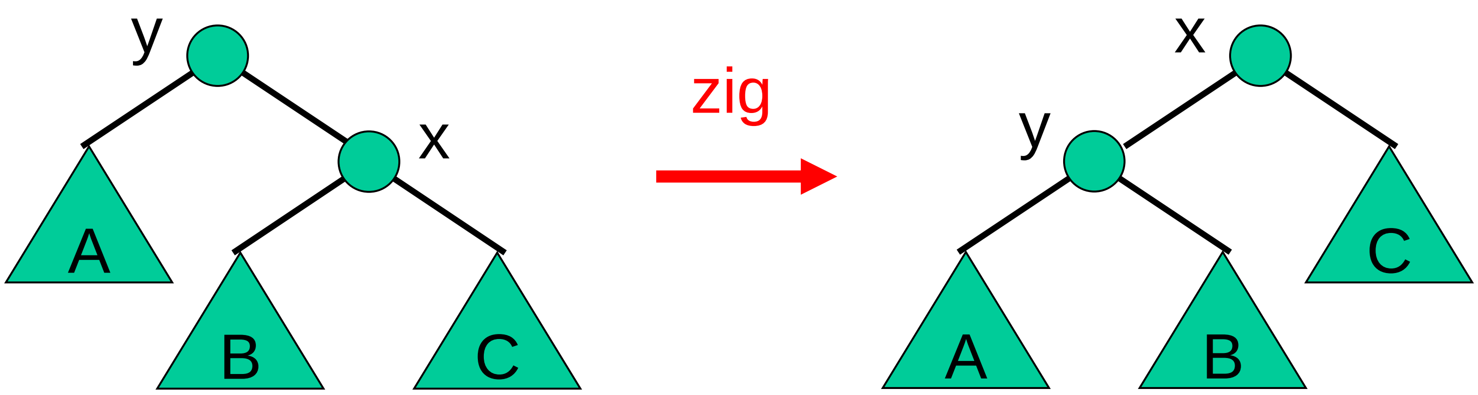


Splay-Operation

Bewegung von Schlüssel x nach oben:

Wir unterscheiden zwischen 3 Fällen.

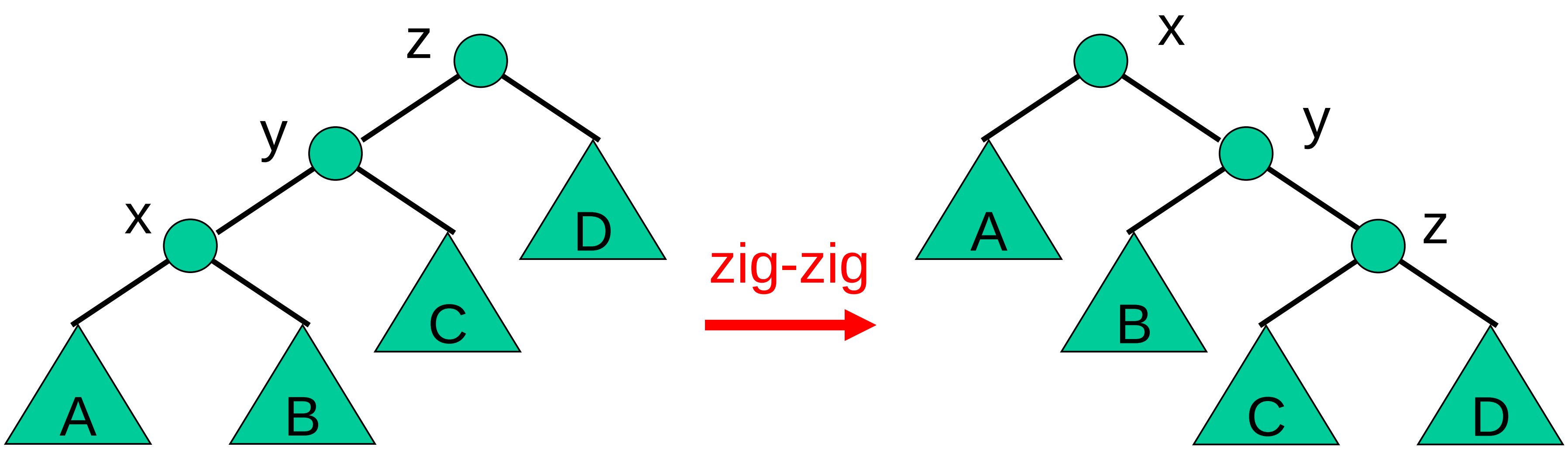
1b. x ist Kind der Wurzel:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

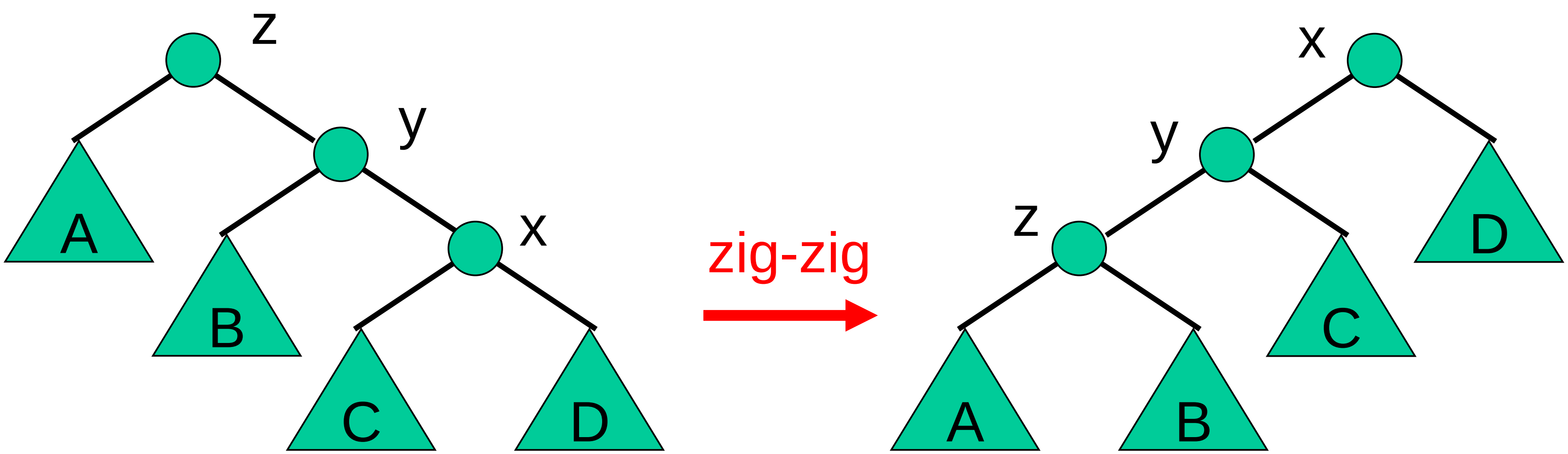
2a. x hat Vater und Großvater rechts:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

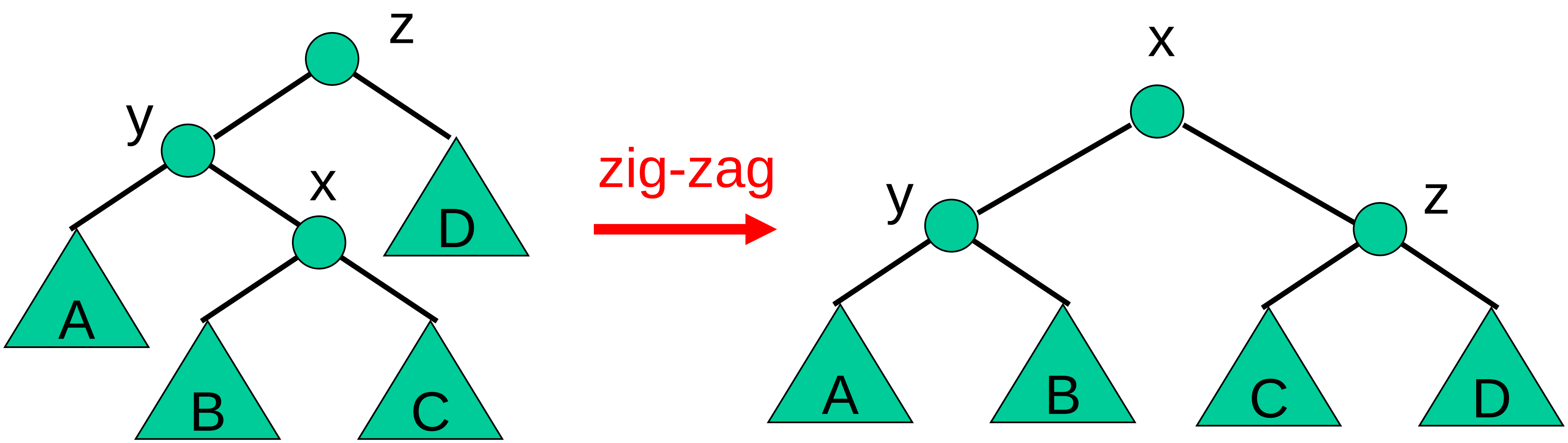
2b. x hat Vater und Großvater links:



Splay-Operation

Wir unterscheiden zwischen 3 Fällen.

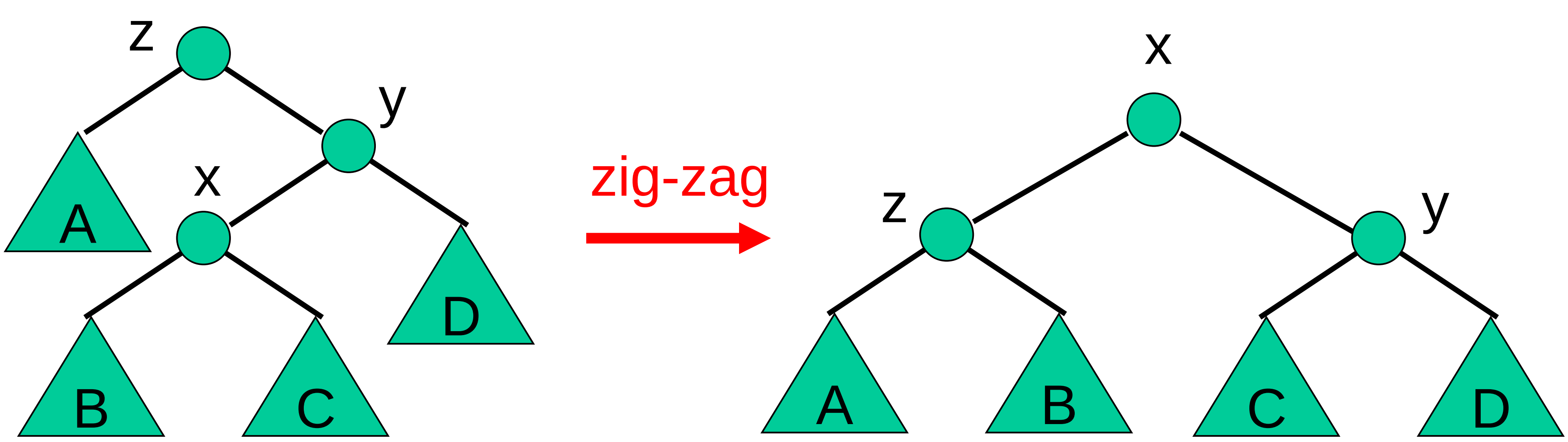
3a. x hat Vater links, Großvater rechts:



Splay-Operation

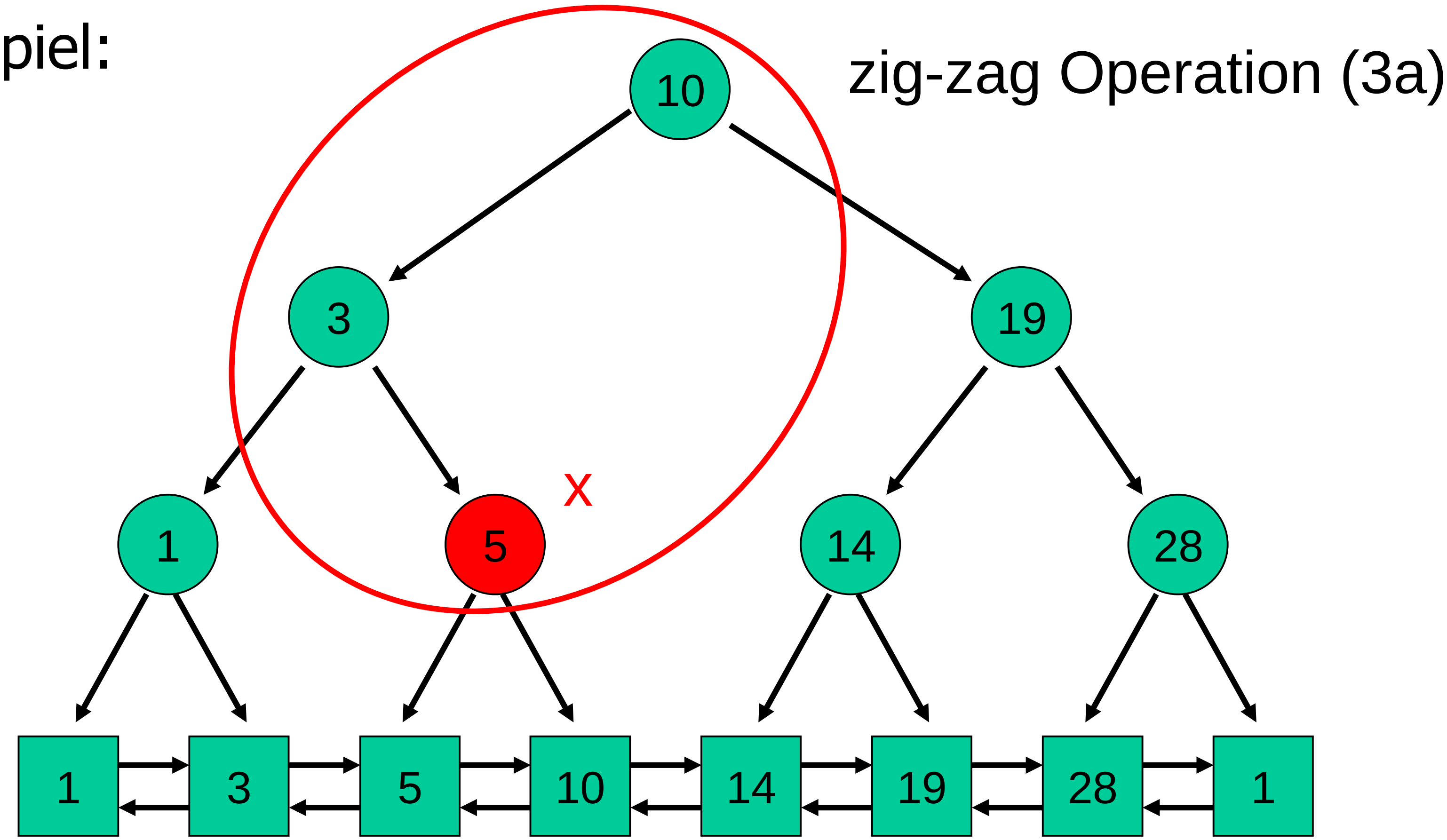
Wir unterscheiden zwischen 3 Fällen.

3b. x hat Vater rechts, Großvater links:

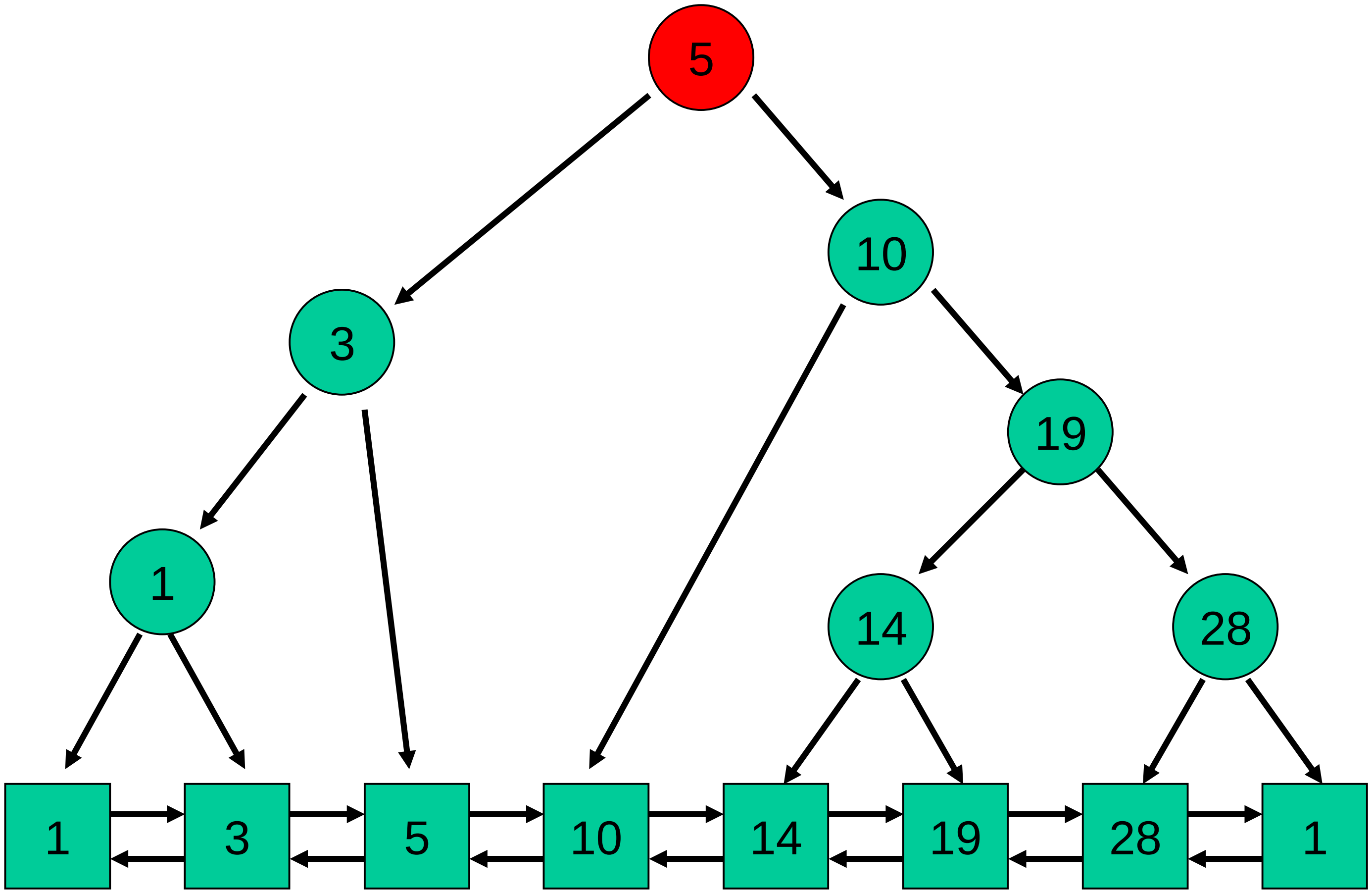


Splay-Operation

Beispiel:

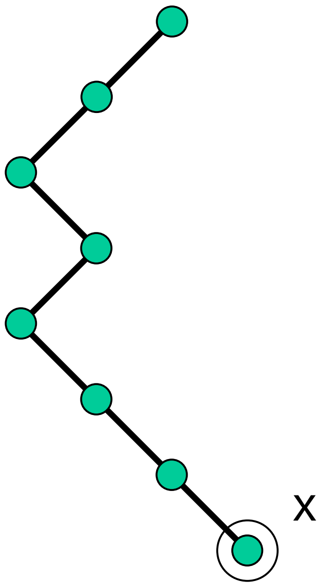


Splay-Operation

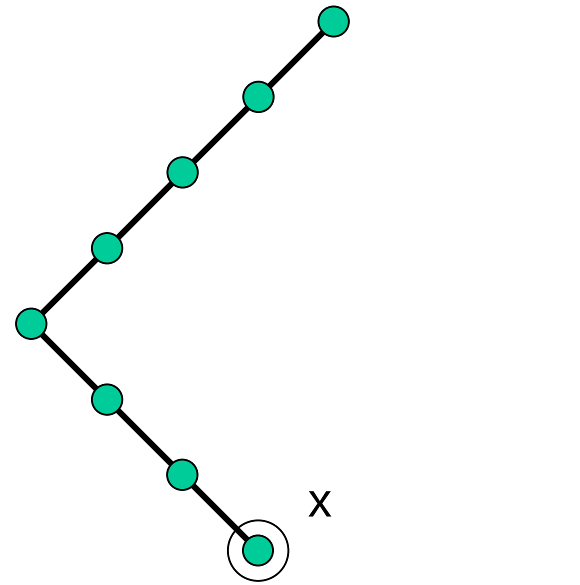


Splay-Operation

Beispiele:



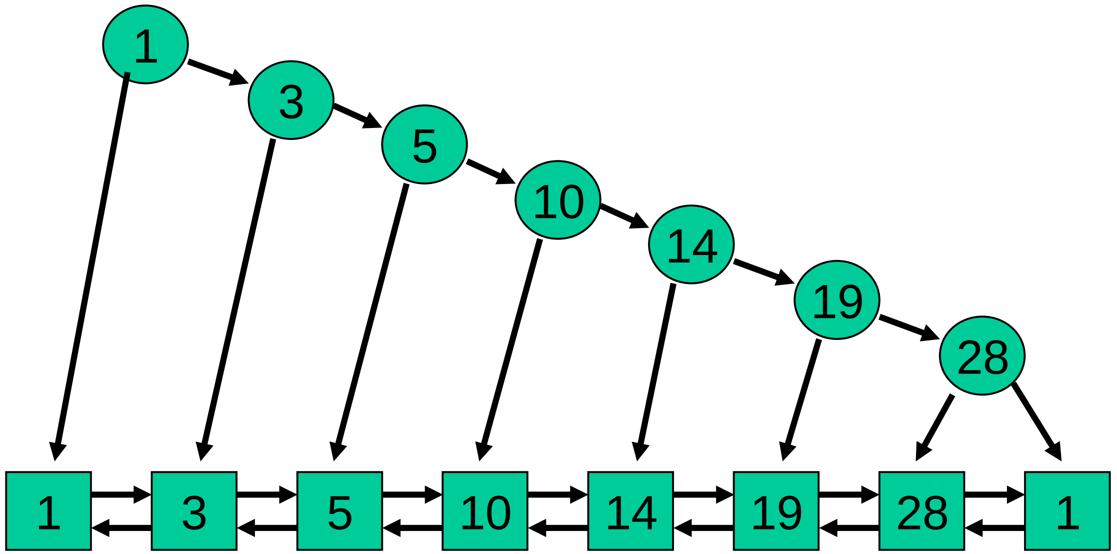
zig-zig, zig-zag, zig-zag, zig



zig-zig, zig-zag, zig-zig, zig

Splay-Operation

Baum kann im worst-case immer noch sehr unbalanciert werden! Aber amortisierte Kosten sind sehr niedrig.



Splay-Operation

suche(k)-Operation:

- Laufe von Wurzel startend nach unten, bis k im Baumknoten gefunden (Abkürzung zur Liste) oder bei Liste angekommen
- k in Baum: rufe splay(k) auf

Laufzeit Analyse:

m Splay-Operationen auf beliebigem Anfangsbaum mit n Elementen ($m>n$) ergibt eine (amortisierte) Laufzeit von:

$$O(m+(m+n)\log n).$$

(also pro Suche im Mittel logn)

Splay-Baum Operationen

Bemerkung: Die amortisierte Analyse kann erweitert werden auf einfügen und löschen, wenn diese wie folgt implementiert sind.

einfügen(e):

- Wie im binären Suchbaum
- Splay-Operation, um key(e) in Wurzel zu verschieben

löschen(k):

- Wie im binären Suchbaum

Fragen?