Inhaltsverzeichnis

1. Laufzeiten

Lineare Suche:

- Gesucht wird der Index einer Zahl x in einem **ungeordneten** Array

- alle Werte durchgehen und mit x vergleichen

- wird das Element gefunden kann abgebrochen werden

→ Die lineare Suche benötigt im **Worst-case** eine **lineare Laufzeit** abhängig von Eingabegröße n

Binäre Suche:

- Gesucht wird der Index einer Zahl in einem (aufsteigend) **sortierten** Array

- In jedem Schritt wird das Element in der Mitte verglichen, ob es größer/kleiner gleich ist

→ Nun lässt sich immer die Hälfte der Werte ausschließen

- Falls das Element enthalten ist wird es nach **log2 (n)** Vergleichen gefunden

- Falls es nicht gefunden wird, wird irgendwann auf einem leeren Bereich gesucht

→ Man weiß, dass es nicht enthalten war

Die Laufzeit eines Algorithmus:

- Soll unabhängig vom System sein

- schwierig von der konkreten Eingabe zu bestimmen

→ Konzept der Laufzeit abhängig der **Eingabelänge n** (wenn man sich auf z.b. Int Werte begrenzt ist ihre Größe Dauer zu Addition etc. Konstant)

… in **Best-case** (nicht sehr nützlich)

… in **Worst-case** (einfach zu bestimmen gute obere Schranke)

… in **Average-case** (etwas schwerer zu bestimmen)

- Eine Operation besteht aus einer maximalen Anzahl an Basisoperationen und die benötigen eine maximale konstante Zeit, wenn man von Ints ausgeht

→Für die Laufzeit lassen sich die Anzahl der Operationen zählen

1.2 O-Notation

Die O-Notation ist eine Laufzeitbeschreibung in Abhängigkeit von der Eingabegröße n.

Dabei beschreibt O(n) die obere Schranke:

- Ab einer gewissen Größe ist c * O(n| n^3|log n...) (*c konstant*) immer größer als die echte Laufzeit

Regeln:

- Produkt-regel: O(f)*O(g) = O(f*g)

- Summen-regel: O(f)+O(g) = O(f+g)

- Absorptions- regel: O(f+g) = O(f), wenn gε O(f)

Zusätzlich beschreibt **Ω(n)** eine **untere Schranke** und **Θ(n)** einen **Durchschnittswert** in dem alle Werte liegen

2. Sortieren

Die untere Schranke für das sortierten eines Arrays im Worst-case ist

Ω(n log(n)) (*Beweis über alle Permutationen in einem Binärbaum*)

<table>
<tr><th>Algorithmus</th><th>Worst-Case</th><th>Best-Case</th><th>Average-Case</th><th>Bemerkungen</th></tr>
<tr><td>SelectionSort</td><td rowspan="5">$\mathcal{O}(n^2)$</td><td>$\mathcal{O}(n^2)$</td><td rowspan="5">$\mathcal{O}(n^2)$</td><td></td></tr>
<tr><td>BubbleSort</td><td rowspan="4">$\mathcal{O}(n)$</td><td></td></tr>
<tr><td>InsertionSort</td><td></td></tr>
<tr><td>ShakerSort</td><td></td></tr>
<tr><td>GnomeSort</td><td></td></tr>
<tr><td>HeapSort</td><td rowspan="2">$\mathcal{O}(n \log(n))$</td><td rowspan="3">$\mathcal{O}(n \log(n))$</td><td rowspan="3">$\mathcal{O}(n \log(n))$</td><td></td></tr>
<tr><td>MergeSort</td><td>rekursiv, $\mathcal{O}(n)$ zus. Platz</td></tr>
<tr><td>QuickSort</td><td>$\mathcal{O}(n^2)$</td><td>rekursiv</td></tr>
</table>

- Sortieren von Listen ⇒ **MergeSort**

- Kleine Arrays ⇒ **InsertionSort**
- Mittlere Arrays ⇒ **QuickSort**
  - bei kleinen Teilen zu **InsertionSort** wechseln.
- Große Arrays ⇒ **HeapSort**

Zusammenfassung Algo-Endklausur

Selectionsort

- In jedem Durchlauf wird das größte/(kleinste) Element gesucht und an die erste,zweite… Stelle getauscht → schlechteste Version

Bubble- ,Insertion-,Shakersort

- In jedem Durchlauf werden immer nur die Nachbarn verglichen und getauscht, wird in einem Durchlauf nicht mehr getauscht kann gestoppt werden.

Gnomesort

- Es wird nur eine Schleife genutzt

- i und i+1 haben keinen Fehler gehen nach rechts vor

- i und i+1 haben einen Fehler tausche und gehe nach links zurück

Heapsort

Ein Heap ist ein Binärbaum, bei dem der Wert des Vaters immer größer als der seiner Kinder ist.

- Das Array kann einen Binärbaum darstellen (Vater i → Kinder 2i+1|2)

- Wende auf jeden Knoten des Symbolisierten Arrays Heapify an → Man erhält einen Heap

In jeder Runde

- Trage die Wurzel als größtes Element in die Liste ein und entferne sie

- Tausche ein Blatt an die Stelle der Wurzel

- Wende Heapify auf die Wurzel an

Heapify

- Wird auf einen Knoten angewendet dabei müssen alle Knoten darunter die Heapbedingung erfüllen

- Hat der gewählte Knoten größere Kinder, dann tausche mit dem größten

→ Gehe mit dem Knoten immer weiter runter bis dies nicht mehr der Fall ist

Mergesort

- Rekrusives Verfahren

- Ist die Größe der Element 2(arbiträr klein) sortiere (z.B. Bubblesort)

- Teile den zu sortierenden Teil in der Hälfte und rufe auf beide Mergesort auf

- Mische die Stapel zusammen man muss immer nur die obersten Elemente vergleichen

- Es wird zusätzlicher Speicher verbraucht, da viele Arrays genutzt werden müssen oder man legt das ganze in einer Liste an

- In jeder Ebene des Rekrusionsbaums wird O(n) aufwand betrieben. Durch die Halbierung ist der Baum log(2) tief → was zur Laufzeit führt

Quicksort

- Rekrusives Verfahren ohne zusätzlichen Speicher

- Wähle ein Element aus dem Array als **Pivot** Element

- Sortiere die größeren Elemente davor und die kleineren dahinter

- Rufe auf beiden Teilen wieder Quicksort auf

- Die Wahl des Pivot Element ist wichtig, wenn man das Maximum bzw. Minimum wählt ist es Selectionsort

- Somit wählt man das Pivot Element zufällig um nicht auf vorsortieren Arrays immer eine schlecht Laufzeit zu haben

- Oder man nutzt Clever Quicksort, welches die Mitte aus 3 Elementen nimmt → Damit lässt sich der Faktor etwas verbessern

3.Lineare Datenstrukturen

Die ADT lassen sich oft generisch für alle möglichen Datentypen nutzen um diese zu verwalten

3.1 Liste

- Einzelne Elemente vom Typ Item die kreuz und quer im Speicher stehen können

- Um eine Liste durchzugehen wird ein Iterator gebraucht, der durch alle Elemente geht

Einfache Liste ohne Durchlauf

Die Liste speichert Objekte der Klasse `Data`.

- `List()` - Leere Liste anlegen, Konstruktor
- `bool empty()` - prüfen, ob sie leer ist
- `Data front()` - erstes Element holen
- `void push_front(Data d)` - Element am Anfang hinzufügen
- `void pop_front()` - erstes Element löschen
- `Data back()` - letztes Element holen
- `void push_back(Data d)` - Element am Ende hinzufügen
- `void pop_back()` - letztes Element löschen
- `void insert(pos, Data d)` - Füge Element vor dem an Position pos ein
- `void erase(pos)` - Lösche das Element an Position pos

Die Attribute

- `head : Item`
- `tail : Item`

Die Klasse `Item`

- **Attribute**
  - `data : Data`
  - `next : Item`
  - `prev : Item`
- **Methoden**
  - `Item(d : Data)`
  - `getData() : Data`
  - `setData(d : Data)`
  - `getNext() : Item`
  - `setNext(n : Item)`
  - `getPrev() : Item`
  - `setPrev(n : Item)`

3.2 Queues

- Warteschlange bei der immer nur auf das zuerst hinzugefügte Element zugegriffen werden kann

- Reihenfolge ändert sich nur bei Priority Queues

Der ADT `Queue`

In der Queue werden Zeiger auf Elemente der Klasse `Data` gespeichert.

- `Queue()` erstellt eine leere Queue
- `isEmpty() : boolean` prüft, ob die Queue leer ist.
- `peek() : Data` gibt das erste Element zurück, ohne es zu entfernen.
- `enqueue(data : Data)` fügt ein Element am Ende der Queue hinzu
- `dequeue() : Data` nimmt das erste Element aus der Queue

3.3 Stack

- Stapel bei dem man immer nur auf das zuletzt hinzugefügte Element zugreifen kann

Der ADT Stack

In dem Stack werden Zeiger auf Elemente der Klasse `Data` gespeichert.

- `Stack()` erstellt einen leeren Stack
- `isEmpty() : boolean` prüft, ob der Stack leer ist.
- `top() : Data` gibt das oberste Element zurück, ohne es zu entfernen.
- `push(data : Data)` legt ein Element auf den Stack.
- `pop() : Data` nimmt das oberste Element vom Stack.

- Gut im arithmetische Ausdrücke zu überprüfen, Klammern zu zählen usw.

4. Binärbäume

Rekursive Definition: Binärbaum

Ein **Binärbaum** T mit Werten aus $\mathcal{V}$ ist

- entweder der **leere Baum** $\perp$,
- ein einzelner **Knoten** (v) mit $v \in \mathcal{V}$, genannt **Blatt**
- oder ein Tripel (L, v, R) mit einem Wert $v \in \mathcal{V}$ und **zwei Binärbäumen** L und/oder R, den sogenannten **linken** und **rechten Teilbäumen**.

Tiefe

- Wurzel Tiefe 0 sonst Anzahl der Vorgänger

- Hat ein Baum Tiefe d, dann hat er maximal 2^d+1

Traverse

Postorder: Links Rechts Wurzel	*Rechts vom Knoten*
Preorder: Wurzel Links Rechts	*Links vom Knoten*
Inorder: Links Wurzel Reachts	*Unter dem Knoten*

4.1 Suchbäume

Suchbäume sind Binärbäume, bei dem alle Werte im rechten Teilbaum größer sind alle Werte im Linken Teilbaum kleiner sind

- Durch die Inorder Traverse lassen sich Suchbaum in aufsteigender Reihenfolge ausgeben

- In Suchbäumen sich einfach nach Werte suchen, indem man immer nur entscheidet ob das Element kleiner oder größer ist. Die Anzahl der Vergleiche hängt nur von der Tiefe ab

- Knoten werden als Blatt eingefügt da wo man sie beim Suchen erwarten würde

- Beim Löschen von Knoten die nicht Blätter sind, ersetzt man den gelöschten Knoten gegen den größten des rechten Teilbaums oder den kleinsten des linken

4.2 AVL Bäume

Ein Binärbaum heißt Balanciert, wenn sich die Tiefe jedes Knoten des linken und rechten Teilbaums um maximal 1 unterscheidet.

Definition: Balancefaktor

Der **Balancefaktor** $bal(v)$ eines Knotens v ist die Differenz der Höhe des linken Kindes und des rechten Kindes:

$$bal(v) = height(\text{linkes Kind von } v) - height(\text{rechtes Kind von } v).$$

Falls das entsprechende Kind nicht existiert wird die entsprechende Höhe auf -1 gesetzt.

Fibonacci-Bäume sind Balancierte Bäume mit minimaler Anzahl Knoten für ihre Tiefe dabei gilt:

Lemma

Es gilt $N(h) = f_{h+3} - 1$ für alle Höhen h.

Satz

Ein balancierter Baum mit n Knoten hat eine Höhe von $\mathcal{O}(\log(n))$.

AVL Bäume

sind Balancierte Suchbäume

- Einfügen funktioniert wie bei der Binären Suche

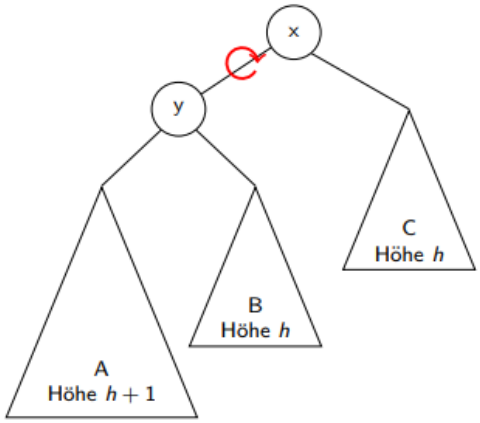
- Nach dem Einfügen muss der Baum wieder balanciert werden, indem man die Faktoren auf dem Rückweg aktualisiert

- löschen Funktioniert auch wie oben, auf dem Rückweg muss dann auch wieder die Höhen aktualisiert werden

- Mit einer Operation Löschen bzw. Einfügen wird der Balancefaktor um maximal 1 geändert

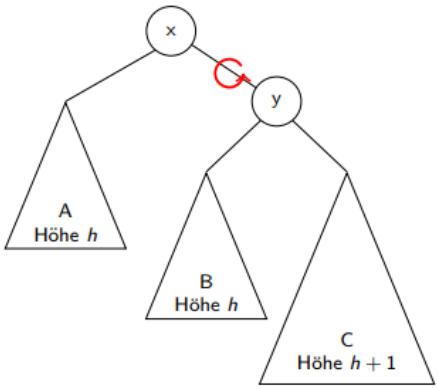
- Insgesamt ergeben sich die folgenden Laufzeiten:
  - Suche: $\mathcal{O}(\log(n))$
  - Einfügen: $\mathcal{O}(\log(n))$
  - Löschen: $\mathcal{O}(\log(n))$

AVL Bäume balancieren



- $bal(x) = 2$
- $bal(y) = 1$

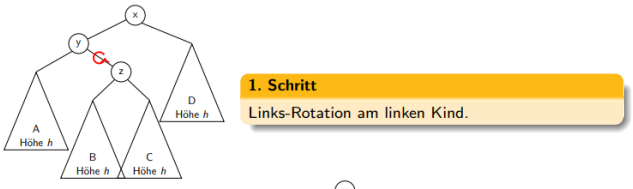
Rechtsrotation an X

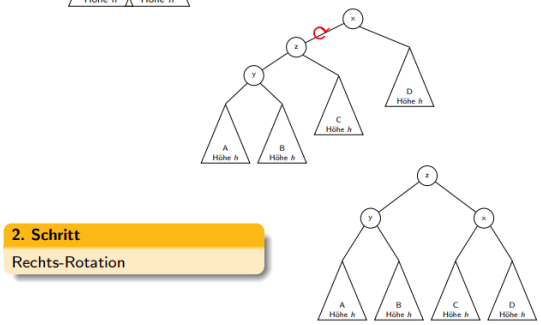


- $bal(x) = -2$
- $bal(y) = -1$

Linksrotation an X

Wenn bei Faktoren das selbe Vorzeichen haben wird nur eine einzelne Rotation benötigt B wird neues Kind von y





x= 2 y= -1

Links (linkes Kind) und dann Rechts an xy

Simultan für

x= -2 y =1

Rechts (rechtes Kind) und dann Links an xy

5. Graphen

Breitensuche

- Starte bei einem Knoten und Markiere in als fertig

- Tue alle Nachbarn die noch nicht abgearbeitet sind in die Queue

- Arbeite solange weiter bis die Queue leer ist

→ Die Ausgabe der Knoten ist nach Entfernung zum Startknoten sortiert

- O(|V|+|E|)

Tiefensuche

- Starte bei einem Knoten markiere diesen als besucht

- Lege alle nicht besuchten Nachbarn auf den Stack

- Arbeite den Stack ab bis dieser leer ist

- O(|V|+|E|)

Minimale Spannbäume

Algorithmus von Jannik & Prim:

Wähle Knoten und checke in einer Tabelle alle Knoten wie teuer es ist diese anzuschließen

→ günstigsten anschließen und wiederholen

- O(n^2+m)

Algorithmus von Kruskal:

- Starte ohne Kanten

- sortiere die Kanten aufsteigend

- Wähle eine Kante immer, wenn sie keinen Kreis erzeugt

- Um zu checken ob zwei Knoten schon vorher verbunden waren nutzt man die Unionfind Datenstruktur die für eine Zusammenhangkomponente immer einen Repräsentanten hat

Algorithmus von Dijkstra (Kürzeste Wege)

- Hebe den Startknoten hoch

- Die Prognose eines Knoten, die Knote zu einem Nachbarknoten der schon hochgehoben wurde, sonst ist sie unendlich

- Hebe nach und nach die Knoten hoch und aktualisiere dabei immer die Prognose und die Vorgänger

4. Möglichkeit: Vorrangwarteschlange mit Fibonacci-Heaps

Die verbliebenen Knoten werden in einem **Fibonacci-Heap** gespeichert. Dann gilt

$$T_{Dijkstra}(n, m) = \mathcal{O}(n \log(n) + m).$$

Zum Vergleich:

- Lineare Suche: $\mathcal{O}(n^2 + m)$
- Min-Heaps: $\mathcal{O}((n + m) \log(n))$

6. Greedy Algorithmen

Greedy Algorithmen sind eine Klasse von Algorithmen die in jedem Schritt die geizigste/ gewinnbringenste Wahl treffen, sodass es immer noch möglich bleibt die Optimale Lösung zu treffen

- Greedy Schritte sind optimale Lösungen für Teilprobleme

- Austauschargument existiert eine Optimale Lösung und eine Unterlösung die nicht Teil dieser Optimalen Lösung ist, so lässt sich etwas austauschen, sodass die Teillösung immer noch optimal und Teil dieser ist

Rucksack Problem

- hier lässt sich **kein** Greedy Algorithmus nutzen um ein optimales Ergebnis zu erzielen

- Wähle in jedem Schritt das Element, welches am wenigsten platz verbraucht| am wertvollsten ist| den meisten nutzen pro Platz hat

Intervall Scheduling

- Greedy-Wahl: wähle immer den Job der möglich ist und als erstes endet

Codebäume

Für einen optimalen Präfixcode lässt sich einfach Hufmann nutzen

$$\text{Kompressionfaktor}(w) := \frac{|code^*(w)|}{|w|}$$

$$\text{Kompressionrate}(w) := \frac{|w|}{|code^*(w)|}$$

7. Amortisierte Kostenanalyse

Nicht Klausurrelevant

8. Priority

- Datenstruktur als Queue, bei dem sich Elemente mit einer größeren Priorität vordrängeln können.

- Realisierung als Dynamisches Array (aufwändig)

- Realisierung als Fibonacci-Heap

  - Ansammlung an Min-heaps in einer doppelt verketteten Liste
  - Jedes Kind ist größer als der Vater
  - Es gibt eine Referenz auf das kleinste Element

Piority Queues

Es wird eine spezielle Form der Warteschlange verwendet, die **Priority Queue** oder **Vorrangwarteschlange**. Jedem Element in ihr ist eine **Priorität** in Form einer Zahl zugeordnet. Sie stellt die folgenden Operationen zur Verfügung:

- `void insert(Object obj, int prio)` – Fügt `obj` mit der Priorität `prio` ein.
- `boolean isEmpty()` – Prüft, ob die Vorrangwarteschlange leer ist.
- `Object deleteMin()` – Löscht das Objekt mit der **niedrigsten** Priorität.
- `void decreasePriority(Object obj, int prio)` – Senkt die Priorität auf `prio`. Wenn der neue Wert größer ist als der aktuelle, geschieht nichts.

→ lässt sich gut nutzen um den Dijkstra durchzuführen

Fibonacci-Heaps

Die amortisierten Kosten

- Erzeugen: $\mathcal{O}(1)$
- Einfügen : $\mathcal{O}(1)$
- Minimum löschen: $\mathcal{O}(D(n))$
- Senken: $\mathcal{O}(1)$

9. Rekursion

Fakt:

Alle primitiv rekursiven Funktionen lassen sich auch in iterativer Schreibweise schreiben und sind dabei auch effizienter

- haben Rekrusionsanker

- haben Rekursionschritte wie z.B. n*n-1 (Fakultät)

Euklidischer Algorithmus GGT

Der größte gemeinsame Teiler

$$ggT(a,b) = \begin{cases} ggT(a-b,b) & \text{falls } a \geq b > 0 \\ ggT(b,a) & \text{falls } b > a \\ a & \text{falls } b = 0 \end{cases}$$

End/Tailrekrusionen führen die Rekursion als letztes durch (lassen sich auch durch Iterationen ersätzen)

Quickselect

Gesucht wird das k größte Element aus einem Array

- Suche ein Pivot Element aus

- sortieren die Elemente davor und dahinter

→ rufe rekrusiv auf den übrigen Teil Quickselect auf

- Weil der Worstcase nur selten auftritt hat Quickselect eine erwartete Laufzeit O(n)

Median der Mediane

- Mit Quickselect wird in 5 Blöcken jeweils der Median gewählt

- Dann wird der Median dieser Mediane als Pivot Element gewählt

→ Eine Aufteilung ist dann im schlechtesten Fall 0.3 / 0.7

9.1 Master Theorem

Satz: Das Master-Theorem

Seien $a \geq 1$ und $b > 1$ Konstanten, $f(n)$ eine Funktion und sei $T(n)$ auf den nicht-negativen ganzen Zahlen definiert durch:

$$T(n) = a \cdot T(n/b) + f(n),$$

wobei n/b entweder $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$ ist.

1. Gilt $f(n) \in \mathcal{O}(n^d)$ mit $d = \log_b a - \varepsilon$ für ein $\varepsilon > 0$, dann ist
$$T(n) \in \Theta(n^{\log_b a})$$
2. Gilt $f(n) \in \mathcal{O}(n^d)$ mit $d = \log_b a$, dann ist
$$T(n) \in \Theta(n^d \log(n))$$
3. Gilt $f(n) = \Omega(n^d)$ mit $d = \log_b a + \varepsilon$ für ein $\varepsilon > 0$, und ist $af(n/b) \leq cf(n)$ für eine Konstante $c < 1$ und alle hinreichend großen n, dann ist
$$T(n) \in \Theta(f(n)).$$

a: Anzahl der Teile

b: 1/b größter Anteil eines Teilproblem

d: f(n) = Zeit für die Aufteilung und Kombination

9.2 Backtracking

Wird ein Ergebnis in z.b. einen Binärbaum gefunden, dann der Ruckweg genutzt werden um den Weg wiederherzustellen, oder die Berechnung durchzuführen

9.3 Dynamische Programmierung

Bei einer merhfach Rekrusion werden oft einige Teilprobleme öfter berechnet → führt zu viel längerer Laufzeit bsw. Rekursive Berechnung der Fibonacci Zahlen

- Durch Speicherung von Zwischenergebnissen kann schneller auf diese zugegriffen werden ohne sie schon zu bearbeiten

→ Nur wenn es wirklich viele Redundanzen in der Rekrusion gibt, sollte man den größeren Speicheraufwand auf sich nehmen

- Speicherung der Werte funktioniert am besten in einem Array (ein oder zweidimensional)