Interreg

EUROPÄISCHE UNION

Österreich-Tschechische Republik

Europäischer Fonds für regionale Entwicklung

INFORMATIK

Algorithmen und Datenstrukturen

UNIVERSITY OF APPLIED SCIENCES UPPER AUSTRIA

VŠTE

EUROPÄISCHE UNION

Inhalt

1. ALGORITHMUS

Unter dem Begriff Algorithmus versteht man eine präzise Anweisung oder ein Verfahren, welche bzw. welches in der Lage ist, einen bestimmten Auftrag auszuführen. Er beschreibt das theoretische Lösungsprinzip, anders als ein exakter (spezifischer) Eintrag in einer bestimmten Programmiersprache.

Der Algorithmus sollte bestimmte Eigenschaften haben:

- **Endlichkeit:** Jeder Algorithmus muss nach dem Ausführen der maximal möglichen Anzahl von Schritten enden. Es kann eine beliebige Anzahl von Schritten geben, aber sie muss für jeden individuellen Input begrenzt sein. Ein Verfahren, das sich nicht an diese Regel hält, darf nicht als Algorithmus, sondern nur als Rechenmethode bezeichnet werden.

- **Allgemeingültigkeit:** Der Algorithmus behandelt nicht nur ein spezielles Problem, sondern eine allgemeine Klasse ähnlicher Probleme.

- **Bestimmbarkeit:** Jeder Schritt im Algorithmus muss eindeutig und präzise definiert sein. In jeder Situation muss klar sein, was zu tun ist, wie es zu tun ist und wie der Algorithmus fortsetzen soll. Einige Algorithmen sind nicht vollumfänglich festgelegt: Sie enthalten ein zufälliges Element (zB genetische Algorithmen).

- **Ausgabe (oder Resultativität):** Der Algorithmus hat zumindest eine Ausgabe, die Menge richtet sich nach der vorgeschriebenen Beziehung zu den Eingaben. Die Ausgabe ist die Antwort auf das Problem.

- **Einfachheit:** Der Algorithmus besteht aus einer begrenzten Anzahl einfacher Schritte.

1.1. Entwicklungsverfahren für Algorithmen

Es gibt verschiedene Verfahren, die eingesetzt werden, um Algorithmen zu entwickeln. Dazu zählen unter anderem:

Top-Down-Methode: Prozesslösungen werden in einfachere Operationen zerlegt, bis die elementaren Schritte erreicht sind.

Bottom-Up-Methode: Ausgehend von den elementaren Schritten werden Ressourcen erstellt, welche es unter Umständen möglich machen, eine spezifische Aufgabe zu bewältigen.

Alternativ gibt es noch eine **Kombination aus beiden Methoden**, wenn eine Top-Down-Methode von einem „partiellen Bottom-Up Schritt“ vervollständigt wird (Verwendung der Funktionsbibliothek, höhere Programmiersprache oder Systemprogrammierung.)

Gebräuchliche Verfahren in Algorithmen sind die folgenden Methoden: Teile-und-herrsche, gierige Algorithmen, dynamische Programmierung und Rückverfolgung.

- Die **Teile-und-herrsche Methode** unterteilt die Probleme in Sub-Aufgaben, die unabhängig voneinander sein müssen. Im Anschluss werden diese Sub-Aufgaben gelöst. Diese Methode wird häufig rekursiv oder iterativ angewendet.

- Ein **gieriger Algorithmus (Greedy Algorithm)** wird vor allem eingesetzt, um Optimierungsprobleme zu lösen. Er wählt immer ein lokales Minimum oder Maximum und versucht, davon ausgehend auf ein globales Minimum oder Maximum zu schließen. Dieser Prozess ist nicht immer ideal und führt nicht zwangsläufig zu einem globalen Minimum (Maximum).

- **Dynamische Programmierung** unterteilt das Problem in Sub-Aufgaben, wie es bei der Teile-und-herrsche-Methode der Fall ist. Der grundlegende Unterschied ist, dass es hier zwischen den Sub-Aufgaben Abhängigkeiten geben darf.

- **Die Rückverfolgung (Suchrücklauf)** ist eine Möglichkeit, algorithmische Probleme zu lösen, indem man den Statusbaum des Problems sucht. Es handelt sich hierbei um eine Brute-Force-Suche.

1.2. Arten von Algorithmen

Algorithmen können in verschiedene Arten unterteilt werden:

- **Rekursive Algorithmen**, welche sich selbst verwenden bzw. abrufen.
- **Randomisierte Algorithmen**, welche basierend auf zufälligen (pseudo-zufälligen) Zwischenergebnissen Entscheidungen treffen.
- **Parallele Algorithmen**, welche Aufgaben auf mehrere Computer (Prozessoren, Threads) verteilen.
- Ein weiterer Typ ist der **genetische Algorithmus**, welcher auf der Imitation evolutionsbiologischer Prozesse basiert.
- **Heuristische Algorithmen** suchen nicht nach einer präzisen, spezifischen Lösung, sondern nur nach einem geeigneten Näherungswert. Sie werden in Situationen verwendet, in denen die verfügbaren Ressourcen für die Ausführung exakter Algorithmen nicht ausreichend. Sie kommen auch zum Einsatz, wenn keine passenden exakten Algorithmen bekannt sind.

2. ABSTRAKTER DATENTYP – ADT

2.1. Beschreibung

Abstrakter Datentyp – kurz ADT – ist ein Ausdruck für Datentypen, die unabhängig von ihrer eigenen Anwendung sind.

Durch die Verwendung von ADTs versucht man, das Programm, welches Operationen mit dem vorgegebenen Datentyp ausführt, zu vereinfachen und zu straffen.

Jeder ADT kann genutzt werden, wenn man grundlegende algorithmische Operationen wie Belegungen, Additionen, Multiplikationen und bedingte Sprünge verwendet.

2.2. Eigenschaften von ADTs

ADTs sollten die folgenden Eigenschaften haben:

- **Allgemeine Anwendbarkeit:** Einmal entwickelt, können ADTs eingebettet werden und laufen problemlos in jedem Programm.
- **Exakte Beschreibung:** Die Verknüpfung zwischen der Anwendung und der Schnittstelle muss eindeutig und vollständig sein.
- **Einfachheit:** Der Nutzer sollte nicht mit der internen Anwendung und Administration von ADTs im Speicher beschäftigt sein.
- **Abkapselung:** Die Schnittstelle ist ein geschlossener Teil. Der Nutzer weiß, was der ADT macht, aber nicht, wie er funktioniert.
- **Integrität:** Der Nutzer kann nicht in die interne Datenstruktur eingreifen.
- **Modularität:** Das „modulare“ Programmierprinzip ist gut angeordnet und erlaubt den einfachen Austausch von Code-Teilen. Bei der Suche nach Fehlern können die individuellen Module als kompakte Einheiten betrachtet werden. Es ist nicht notwendig, in das gesamte Programm einzugreifen, um ADTs zu verbessern.

Wenn ADTs objektorientiert programmiert sind, werden diese Eigenschaften typischerweise standardmäßig eingehalten.

2.3. Operationen in ADTs

Durch die Verwendung von ADTs können elementare Operationen ausgeführt werden. Diese umfassen den Konstruktor, Selektor und Modifizierer.

- Der **Konstruktor** erstellt einen neuen ADT. Er bildet – basierend auf bestimmten Parametern – eine gültige interne Darstellung des Werts heraus.

- Der **Selektor** holt die Werte ein, welche die Komponenten oder Eigenschaften eines spezifischen ADT-Werts vervollständigen.

- Der **Modifizierer** nimmt Änderungen an Datentyp-Werten vor.

3. ANALYSE VON ALGORITHMEN

3.1. Effektivitätsvergleich von Prozessen

Ein Problem kann für gewöhnlich durch die Anwendung verschiedener Algorithmen gelöst werden. Es ist daher notwendig, ein Werkzeug zu haben, welches die einzelnen Verfahren hinsichtlich der Effektivität des Prozesses vergleicht. Algorithmen können dabei entweder **experimentell** oder **theoretisch** miteinander verglichen werden.

Eine **experimentelle** Analyse nimmt viel Zeit in Anspruch. Klarerweise dauert sie – je nach Menge und Größe der Eingaben – länger bzw. kürzer. Diese Art der Analyse erfordert eine spezielle Anwendung des Algorithmus, welche wiederum weiteres Wissen voraussetzt. Es ist schwierig, einen durchschnittlichen Fall zu finden. Aus diesem Grund ist es besser, sich auf den schlimmstmöglichen Fall zu konzentrieren. Dieser ist einfach zu analysieren und wichtig für die meisten Anwendungen, seien es nun Spiele, Finanzapplikationen, Robotertechnik oder automatisierte Operationen.

Die experimentelle Analyse der benötigten Zeit findet in einer Umgebung statt, in welcher der Algorithmus (das Programm) meist unter Verwendung einer internen Zeitmessungsfunktion ausgeführt wird. Der Programmdurchlauf hängt von den Eingaben und deren Zusammensetzung ab, und nicht alle Eingaben sind in jedem Programmdurchlauf enthalten. Um zwei Algorithmen vergleichen zu können, benötigt man dieselbe Hard- und Software sowie dieselbe Speicherbelegung.

Anstelle einer experimentellen Analyse können auch bestimmte **theoretische** Verfahren verwendet werden. Die theoretische Analyse verwendet die Beschreibung des Algorithmus mittels Operationen anstatt zur spezifischen Ausführung. Sie berücksichtigt alle Eingaben und ermöglicht es, die Geschwindigkeit des Algorithmus unabhängig von Hardware oder Software einzustufen.

Eines dieser Werkzeuge ist der **Pseudocode**. Dieser erlaubt es, höhere Ebenen von Algorithmus-Beschreibungen zu verwenden. Die Beschreibung ist strukturierter als ein gewöhnlich geschriebener Text, aber nicht so detailliert wie eine spezifische Anwendung. Es handelt sich beim Pseudocode um eine bevorzugte Schreibweise, um Algorithmen zu beschreiben. Sein Vor- und Nachteil gleichermaßen ist, dass er Probleme einer besonderen Anwendung ausblendet. Der Pseudocode verwendet Schlüsselwörter, um den Algorithmus zu beschreiben:

Zur Durchlaufkontrolle:

-If...then...else (condition),
-while...do, repeat...until, for...do (cycles)

Header: Algorithmus `Name(arg1, arg2…)`, input, output

Verfahrensabruf (Methoden, Algorithmus): `var.Name(arg1,arg2,…)`

Wertrückgabe: `return` *`Expression`*

Ausdrücke:

←	Zuschreibung
=	Gleichheit
+, -, n2,…	Mathematische Operationen

3.2. Primitive Operationen

Eine weitere Möglichkeit ist, Algorithmen mithilfe **primitiver Operationen** zu analysieren.

Eine primitive Operation ist eine grundlegende Operation, welche von einem Algorithmus ausgeführt wird. Sie ist in einem Pseudocode identifizierbar, unabhängig von der Programmiersprache und sollte präzise definiert sein. Solch eine primitive Operation könnte zum Beispiel die Auswertung eines Ausdrucks sein, aber auch die Zuweisung eines Werts zu einer Variable, deren Indexierung in einem Feld, ein Verfahrensabruf, eine Rückgabe usw.

Ebenso kann man eine **asymptotische Notation (big O, Bachmann-Landau Notation)** verwenden. Sie stellt fest, wie operational anspruchsvoll der Algorithmus ist, indem ermittelt wird, wie sich das Verhalten des Algorithmus in Abhängigkeit von der Größe der eingegebenen Daten verändert. Die asymptomatische Zeit und Raumkomplexität kommen hier für gewöhnlich zum Einsatz.

Die verschwendete Schreibweise bedeutet, dass der Algorithmus weniger als A+B f(N) erfordert, wobei A und B adäquat ausgewählte Konstanten sind und N die Variable bezeichnet, welche der Größe der Dateneingabe entspricht. Die zusätzlichen und multiplikativen Konstanten, sprich *O(N+1000)=O(1000 N)=O(N)*, werden ignoriert. Das Interesse gilt rein dem Verhalten eines hohen N-Wertes.

Um die Zeit zu ermitteln, die der Algorithmus, welcher die Big O Notation verwendet, benötigt, muss man die größtmögliche Nummer an primitiven Operationen finden, welche dann mithilfe der Big O Notation ausgedrückt wird.

4. QUEUES UND STACKS

4.1. Stacks

Ein **Stapelspeicher (Stack)** ist eine Datenstruktur, welche dazu dient, Daten zu speichern. Charakteristisch für ihn ist die Möglichkeit zur Datenmanipulation. Er greift auf die Daten zu, indem er das LIFO (**Last In First Out**)-Prinzip verwendet. Man kann sich das al seine Art Container vorstellen.

Der ADT-Stack muss zumindest die folgenden Funktionen enthalten:

- Einfügen eines Objekts
- Rückgabe und Entfernung des letzten Objekts
- Abfragen am oberen Ende des Stacks
- Stack-Größe
- Stack-Inhalt (leer oder Daten vorhanden)

Wenn wir versuchen, eine Pop- oder Top-Operation in einem leeren Stack auszuführen, bekommt man eine *EmptyStackException*-Ausnahme.

Stack-Anwendungen sind beispielsweise der Verlauf des Web-Browsers, die Rückgängig-Reihenfolge in Textverarbeitungsprogrammen oder individuelle Abrufverfahren. Der Stack kann als eine Hilfsdatenstruktur für andere Algorithmen oder als Teil einer anderen Datenstruktur verwendet werden.

Am einfachsten lässt sich ein Stack in Form einer Datenreihe (Array) umsetzen. Elemente werden von links nach rechts ergänzt und die Hilfsvariable besitzt den Index des jeweils letzten Elements.

Dank der Array-Eigenschaften erhält man die folgenden Eigenschaften:

- n – Anzahl der Elemente in dem Stack
- Anforderungen an den Speicher - O (n)
- Dauer der einzelnen Operationen - O (1)

Ein Array unterliegt jedoch auch Einschränkungen:

- Zu Beginn muss die Stack-Größe definiert werden
- Die Stack-Größe kann nicht Wunsch verändert werden
- Das Hinzufügen eines Elements zu einem vollen Stack wird zu einer anwendungs-spezifischen Ausnahmebedingung führen

4.2. Queues

Die **Reihen-Datenstruktur (Queue)** setzt auf das FIFO-Prinzip (**First in First Out**).

In ihrer minimalen Umsetzungsform muss eine Queue die folgenden Funktionen enthalten:

- Einfügen eines Objekts am Ende der Queue
- Auswahl eines Objekts vom Anfang der Queue
- Abfrage am Anfang der Queue
- Queue-Länge und –Belegung

Ebenso wie ein Stack kann eine Queue während einer „Aus der Warteschlange enfernen"-Operation eine Ausnahmebedingung ausgeben oder sich über eine leere stack?queue (*EmptyStackException*) einreihen.

Queue-Anwendungen sind zum Beispiel Wartelisten, Warteschlangen, Zugriff auf gemeinsam genutzte Ressourcen wie Drucker oder Multi-Programmierung. Die Queue kann auch als Hilfsdatenstruktur für andere Algorithmen oder als Teil anderer Datenstrukturen verwendet werden.

Die Queue kann mithilfe eines Arrays umgesetzt werden. Um ihre Eigenschaften zu verbessern, wird ein zyklisches Array verwendet. Folglich gibt es zwei Variablen: f, den Index des ersten Elements und r, den Index des letzten Elements plus eins (zeigt auf den ersten freien Platz).

5. VEKTOREN, LISTEN UND SEQUENZEN

5.1. Vektoren

Vektoren erweitern das Konzept des Arrays, indem sie die Reihung jedes Objekts speichern. Ein Element im Vektor kann gelesen, eingefügt und entfernt werden, indem man dessen Reihenfolge ermittelt.

Ein Vektor ermöglicht **grundlegende Operationen:**

- Spezifische Reihung von Elementen
- Ersetzen von Elementen an einer bestimmten Stelle
- Einfügen eines Elements an einer bestimmten Position
- Entfernen eines Elements von einer bestimmten Position.
- Darüber hinaus lässt sich feststellen, wie groß ein Vektor ist und ob er leer ist.

Vektorenoperationen können im Fall eines falschen Index' eine Ausnahmebedingung ausgeben, welche in der Regel negativ ist. Eine Vektoranwendung ist eine sortierte Objektsammlung, sprich elementare Datenbank.

Vektoren können mithilfe eines Arrays umgesetzt werden. Dies führt zu folgenden Eigenschaften:

- Variable ***n*** zeigt die Länge des Vektors an
- Operation *isEmpty () elemAtRank (r) replaceAtRank (r, O) - Zeitkomplexität O(1)*
- Operation *insertAtRank (r, O) - Zeitkomplexität O (n)*
- Operation *removeAtRank (R) - Zeitkomplexität O (n)*

5.2. Listen

Eine weitere Datenstruktur ist die Liste. Die Liste ist eine Reihung von Positionen, welche Daten jedweder Form speichert. Sie führt Beziehungen zwischen vor- und nachgereihten Positionen ein.

Allgemeine Operationen sind die Abfrage der Größe und eine leere Listenabfrage. Darüber hinaus kann man herausfinden, ob ein bestimmtes Element das erste oder das letzte ist. Überlässt ist es möglich, das erste und letzte sowie das vorherige und nächste Element zu erhalten.

Die **ADT-Liste** enthält die folgenden **Funktionen**:

Ersetzen	`replaceElement (p,o)`
Vertauschen	`swapElements (p, q)`
Danach einfügen	`insertAfter (p, o)`
Als erstes einfügen	`insertFirst (o)`
Als letztes einfügen	`insertLast (o)`
Entfernen	`remove (p)`

Listen können in eine einfach (Single Linked List) und eine doppelt verketteten Liste (Double Linked List) unterteilt werden. In einer Single Linked List enthält das Element einen Verweis auf den nächsten Knoten. In einer Double Linked List gibt es darüber hinaus einen Verweis auf den vorhergehenden Knoten.

5.3. Sequenzen

Eine ADT-Sequenz ist eine Kombination aus Vektor und Liste, auf Elemente kann daher sowohl durch Verwendung der Position als auch der Reihenfolge zugegriffen werden. Neben den Vektor- und Listenfunktionen enthalten Sequenzen auch die miteinander verbindenden Operationen `atRank (r)` und `rankOf (p)`.

Eine Sequenz ist ein allgemeiner, wesentlicher Datenstrukturtyp der genutzt werden kann, um ein geordnetes Set von Elementen zu speichern. Sie dient als allgemeiner Ersatz für einen Stack, eine Queue, einen Vektor oder eine Liste. Darüber hinaus kann sie als kleine Datenbank verwendet werden.

6.BÄUME

Bäume repräsentieren ein hierarchisches Strukturmodell, welches aus Knoten besteht, zwischen denen eine Eltern/Kind-Beziehung besteht. Bäume können als Organigramm, Datensystem oder Programmierumgebung verwendet werden.

Die folgende Terminologie wird verwendet, um Bäume und ihre Teile zu beschreiben.

6.1. Knotenarten

- **Wurzel** (root)
- **Innerer Knoten:** Knoten, der weder eine Wurzel noch ein Blatt ist
- **Liste (Blattknoten, Externer Knoten):** Knoten ohne Spross
- **Elternknoten:** unmittelbar vor einem Knoten befindlicher Knoten auf dem Pfad vom Blatt zur Wurzel
- **Kinderknoten:** unmittelbar auf einen Knoten folgender Knoten auf dem Pfad von der Wurzel zum Blatt
- **Geschwisterknoten:** bezieht sich auf die Knoten mit demselben Elternknoten
- **Vorfahrknoten, Vorgängerknoten:** vor einem bestimmten Knoten liegender Knoten auf dem Pfad zur Wurzel (direkt angrenzender Vorfahrknoten ist der Elternknoten)
- **Nachfolgerknoten:** Knoten, der sich auf dem Pfad von der Wurzel hin zu irgendeinem Blatt hinter einem bestimmten Knoten befindet (nächster Nachfahre ist d. Nachfolger)
- **Tiefe:** Die Baumtiefe ist die Länge des längsten Pfads von der Wurzel zum Blatt, wobei einem leeren Baum der Wert -1 zugewiesen wird
- **Stufe:** wird gemeinhin für eine Sammlung von Knoten verwendet, welche sich in derselben Entfernung zur Wurzel befinden. Gezählt wird die Anzahl der Knoten.

6.2. Struktur

- **Unterbaum:** Sub-Diagramm eines Baums, welches seinerseits ebenfalls ein Baum ist (zumeist werden Unterbäume gebildet, indem man einen Baumknoten als eine neue Wurzel definiert, während der Rest der Struktur beibehalten wird)
- **Ast:** jeder Pfad von der Wurzel zum Blatt

6.3. Funktionen zur Manipulation von Bäumen

Allgemeine Funktionen:

- integer size()
- boolean isEmpty()
- objectIterator elements()
- positionIterator position()

Zugriffsfunktionen:

- position root()
- position parent()
- positionIterator children(p)

Abfragefunktionen:

- boolean isInternal(p)
- boolean isExternal(p)
- boolean isRoot(p)
- Aktualisierungsoperationen
- swapElements(p, q)
- object replaceElement(p, o)

Da ein Baum eine hierarchische Struktur hat, kann er auf verschiedenen Wegen nachverfolgt werden.

Pre-Order Datendurchlauf:

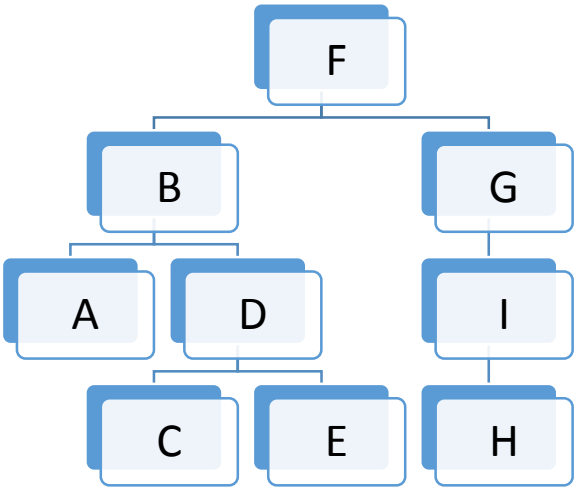
- Überprüfung, ob der Knoten leer oder nicht vorhanden ist
- Anzeigen der aktuellen Knotendaten
- Datendurchlauf des linken Unterbaums mittels eines rekursiven, vertikal hierarchischen Funktionsabrufs (von oben nach unten)
- Datendurchlauf des rechten Unterbaums mittels eines rekursiven, vertikal hierarchischen Funktionsabrufs (von oben nach unten)

In-Order Datendurchlauf:

- Überprüfung, ob der Knoten leer oder nicht vorhanden ist
- Anzeigen der aktuellen Knotendaten
- Datendurchlauf des linken Unterbaums mittels eines rekursiven, aufsteigend sortierten Funktionsabrufs (chronologisch/alphabetisch)
- Datendurchlauf des rechten Unterbaums mittels eines rekursiven, aufsteigend sortierten Funktionsabrufs (chronologisch/alphabetisch)

Post-Order Datendurchlauf:

- Überprüfung, ob der Knoten leer oder nicht vorhanden ist
- Anzeigen der aktuellen Knotendaten
- Datendruchlauf des linken Unterbaums mittels eines rekursiven, vertikal hierarchischen Funktionsabrufs (von unten nach oben)
- Datendurchlauf des rechten Unterbaums mittels eines rekursiven, vertikal hierarchischen Funktionsabrufs (von unten nach oben)



Jede Art des Datendurchlaufs führt zu einem anderen Resultat

- **Pre-Order-Ergebnis:** F, B, A, D, C, E, G, I, H
- **In-Order-Ergebnis:** A, B, C, D, E, F, G, H, I
- **Post-Order-Ergebnis:** A, C, E, D, B, H, I, G, F

Wenn man einen ADT-Baum verwendet, ist es des Weiteren möglich, einen Binär-Baum oder andere Arten zu definieren.

Der Binär-Baum erweitert die Definition des Baums um die Vorgabe, dass kein Knoten mehr als zwei Kinder haben darf, womit ein geordnetes Paar aus linkem und echtem Nachfahren gebildet wird.

Der Binär-Baum fügt zusätzliche Funktionen hinzu:

- position leftChild(p)
- position rightChild(p)
- position sibling(p)

7. VORRANG-QUEUES UND HEAPS

7.1. Vorrang-Queues

Eine **Vorrang-Queue** (Vorrangwarteschleife) speichert eine Objektsammlung, in der die Objekte nach Schlüssel/Wert-Paaren (Vorrangigkeit) geordnet sind.

Deren **elementare Umsetzung** sollte folgende Funktionen enthalten:

- InsertItem (k, o)
- removeMin ()
- Minkey (k, o)
- minElement ()
- Size ()
- isEmpty ()

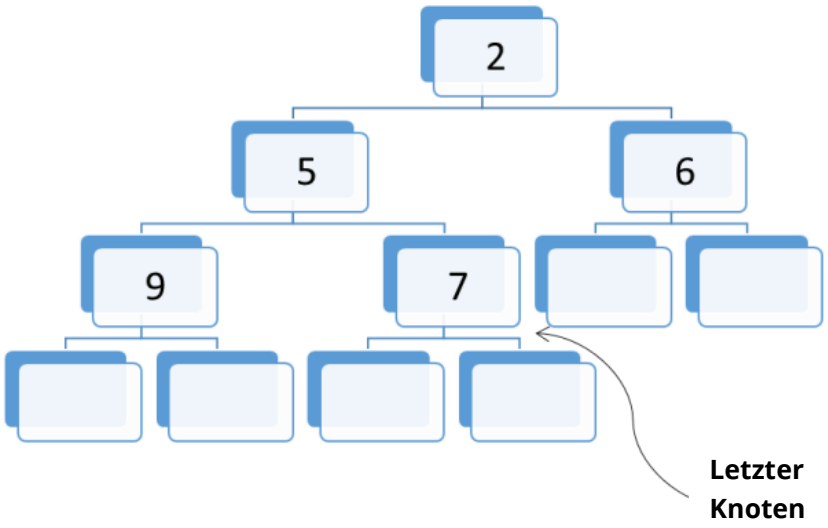
Beispiele für Vorrang-Queues sind Auktionen und Aktienbörsen.

Der **Vorrang-Queue-Schlüssel** kann jedes Objekt mit einer definierten Reihenfolge sein, welches sich ordnen lässt. Zwei verschiedene Elemente (Werte) können denselben Schlüssel (Vorrang) haben. Um eine Vorrang-Queue zu verwenden, muss eine ADT-Vergleichseinrichtung implementiert werden, welche den Vergleich zweier Objekte ermöglicht.

7.2. Heaps

Bei einem **Heap** handelt es sich um einen Binär-Baum, welcher Schlüssel als interne Knoten abspeichert. Für jeden Knoten außerhalb der Wurzel gilt, dass dessen Knotenschlüssel relevanter ist als jener des Elternknotens. Für jeden Heap wird ein kompletter Binär-Baum definiert. Wenn h die Höhe des Baums ist, gibt es für *i* von *0* bis *h-1 2i* Knoten mit der Tiefe *i*.

Der letzte Heap-Knoten ist ein interner Knoten, welcher sich am äußersten rechten Rand auf der Stufe *h -1* befindet.



Ein Heap kann eingesetzt werden, um eine Vorrang-Queue zu realisieren. Im Anschluss wird ein Objekt (Schlüssel, Wert) in jedem internen Knoten gespeichert, wobei ein Verweis auf die Position des letzten Elements behalten wird.

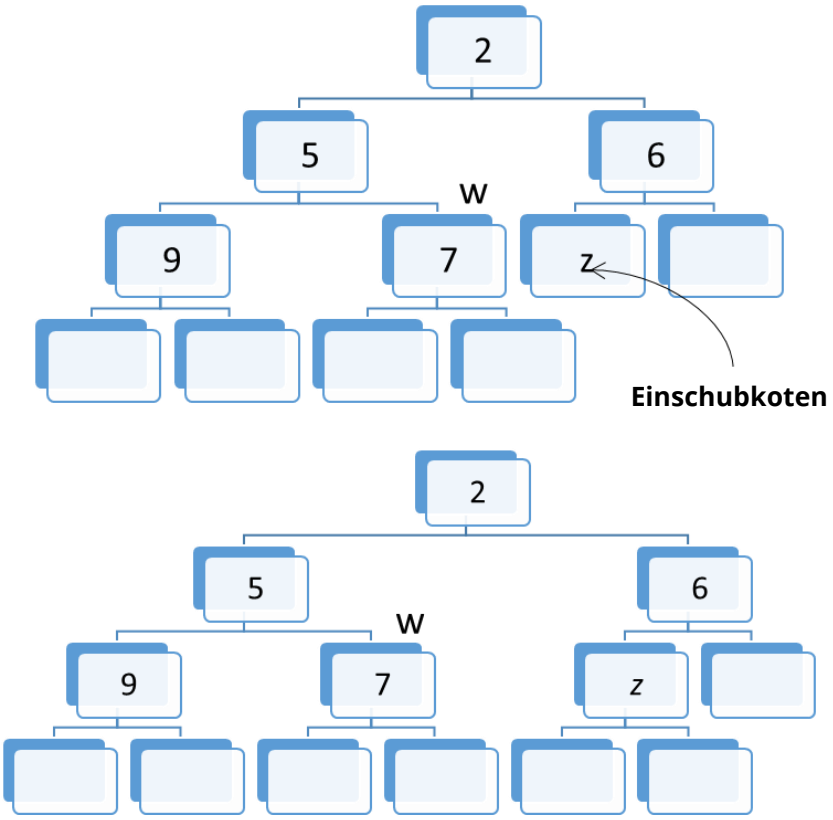
7.3. Handhabung von Heaps

InsertItem (k, o) Funktion

- Aufspüren des Knotens, in den das Objekt eingefügt wird
- Speichern des k-Schlüssels in den z-Knoten, der z-Knoten wird in einen internen Knoten umgewandelt
- Wiederherstellung der Heap-Reihenfolge (Prüfung der Eigenschaften) – upheap Funktion

RemoveMinch () Funktion

- Bezieht sich auf die Entfernung der Wurzel vom Heap (Knoten 2)
- Verändert die Wurzel für den letzten Knoten (2-7)
- Verändert den w-Knoten and seine Kinder in der Liste
- Stellt die Heap-Reihenfolge wieder her – downheap () Funktion



upheap()

- Das Einfügen eines Knotens kann sich auf die Anordnung/das Layout auswirken
- Der upheap-Algorithmus stellt die Ordnung wieder her, indem er den k-Schlüssel oberhalb des eingefügten Knotens austauscht
- Er endet, wenn der interne Knoten zur Wurzel wird oder der Elternschlüssel kleiner oder gleich k ist

downheap()

- Das Entfernen eines Knoten kann sich auf die Anordnung/das Layout auswirken
- Der downheap-Algorithmus stellt die Ordnung wieder her, indem er den k-Schlüssel unterhalb des entfernten Knotens austauscht
- Er endet, wenn der interne Knoten zu einem Blatt wird oder der Kind-Schlüssel größer oder gleich k ist

8. DICTIONARY UND HASHTABELLEN

8.1. Dictionary

Ein **Dictonary** bezieht sich auf einen ADT, welcher eine Schlüssel/Wert-Sammlung enthält, nach der gesucht werden kann. Die folgenden **Funktionen** können mithilfe eines Wörterbuchs durchgeführt werden:

- FindElement (k)
- insertItem (k, o)
- removeElement (k)
- size ()
- isEmpty ()
- keys ()

Beispiele für Wörterbuchanwendungen sind Verzeichnisse, die Kreditkartenautorisierung, Wörterbucher sowie die Domainnamen-Übersetzung in eine IP-Adresse.

Ein Logdaten-Wörterbuch wird in Form einer Zufallssequenz als Double Linked List umgesetzt. Die Objekte werden dabei in einer beliebigen Reihenfolge gespeichert.

Komplexität der Funktionen:

- Objekt einfügen O (1)
- Element finden, Element entfernen O (n)

Eine **Log-Datei** ist für kleine Wörterbücher oder Anwendungen geeignet, in welchen die am häufigsten gebrauchte Funktion das Einfügen ist, jedoch nur selten etwas gesucht oder entfernt wird.

Die Funktion **findElement (k)**, welche in Form einer Sequenz in ein Wörterbuch implementiert ist, baut auf einem Schlüssel-basierten Array auf und wird als binäre Suche ausgeführt. Bei jedem Schritt wird die Anzahl der Kandidaten durch zwei geteilt, wobei die Funktion nach einer logarithmischen Anzahl an Schritten endet.

Eine **Nachschlagetabelle** ist ein Wörterbuch, welches mithilfe einer geordneten Sequenz umgesetzt wird. Hier werden die Wörterbuch-Einträge in einer Sequence gespeichert, welche auf einem Schlüssel-basierten Array aufbauen. In diesem Zusammenhang wird ein externer Schlüssel-Komparator benötigt.

Komplexität der Funktionen:

- Aufspüren des Elements *O (log (n))*
- Einfügen eines Elements, Entfernen eines Elements *O (n)*

Eine Nachschlagetabelle erweist sich als effektiv bei kleinen Wörterbüchern oder Anwendungen, wo Suchen eine häufig verwendete Funktion ist.

Der **Binäre Such-Baum** ist ein binärer Baum, für welchen die folgenden Regeln gelten:

- u, v und w sind solche Knoten, für welche es stimmt, dass sich u in dem linken Unterbaum von v und dass sich w im rechten Unterbaum von v befindet
- $key(u) \leq key(v) \leq key(w)$
- Externe Knoten speichern keinerlei Objekte
- In-Order-Datendurchlauf vergibt die Schlüssel in aufsteigender Reihenfolge

8.2. Hashtabelle

Hash-Funktion *h* ist eine Funktion, welche einem Schlüssel einer bestimmten Art einen Integralwert aus dem Intervall zwischen *0* und *N-1* zuweist. Der Zweck dieser Funktion ist die gleichmäßige Aufteilung der Schlüssel bei einem bestimmten Intervall.

Eine Hashtabelle für eine bestimmte Schlüsselart enthält eine Hash-Funktion und ein Array (eine Tabelle) der Größe N.

Der Schlüssel wird durch einen Hash-Wert ersetzt. Es kann jedoch auch vorkommen, dass für die zwei Schlüssel derselbe Hash-Wert generiert wird. In diesem Fall tritt eine Kollision auf. Dieses Problem kann auf zwei Arten gelöst werden:

- Variante 1 ist die **Verkettung**, in deren Zuge die miteinander in Konflikt stehenden Objekte als Sequenzen abgespeichert werden.

- Eine weitere Möglichkeit ist das **freie Adressieren**, bei dem das Objekt an einer anderen Stelle in der Tabelle gespeichert wird.

9. SORTIERALGORITHMEN

9.1. Erklärung

Sortieralgorithmen werden eingesetzt, um die Daten in einer Datei in einer speziellen Reihenfolge zu sortieren, entweder alphabetisch oder numerisch. Das Schlüssel/Wert-Paar wird nach dem Schlüssel gewichtet, der Wert findet hingegen keine Berücksichtigung.

Sortieralgorithmen können in **stabile** und **instabile** unterteilt werden, was davon abhängt, ob sie die Reihenfolge der Objekte mit demselben Schlüssel beibehalten. Unterschieden werden kann auch zwischen **natürlichen** und **unnatürlichen** Sortieralgorithmen, wobei natürliche mit einem teilweise geordneten Datensatz schneller arbeiten.

Darüber hinaus können sie auf den **Sortiertyp** heruntergebrochen werden:

- durch Auswahl
- durch Einfügen
- durch Ersetzen
- durch Zusammenführen

Die **bekanntesten Algorithmen** sind

- Bubble Sort
- Heap Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Selection Sort

Andere Algorithmen, welche auf verschiedenen Prinzipen basieren, sind unter anderem:

- Bucket Sort
- Radix Sort
- Counting Sort

9.2. Bubble Sort

- Einfach zu implementieren
- Universell, lokal (in-situ, kein Bedarf an zusätzlichem Speicherplatz)
- Der Algorithmus beginnt am Anfang des Datensets. Er vergleicht die ersten zwei Elemente und wenn das erste größer als das zweite ist, vertauscht er diese. Diesen Vorgang wiederholt der Algorithmus bei jedem Paar benachbarter Elemente, bis er am Ende des Datensatzes angekommen ist. Danach beginnt der von Neuem mit den ersten zwei Elementen und wiederholt diesen Vorgang so lange, bis im letzten Durchlauf kein Austausch mehr möglich ist.

Der Bubble Sort-Algorithmus im Detail:

```
procedure bubbleSort( A : list sortable items )
    n = length(A)
    repeat
        swapped = false
        for i from 1 to n-1 inclusive do
      if A[i-1] > A[i] then
                        swap( A[i-1], A[i] )
                swapped = true
            end if
        end for
    until not swapped
end procedure
```

9.3. Heap Sort

- Ein vergleichsbasierter Sortieralgorithmus
- Nicht stabil
- Verwendet die Heap-Datenstruktur und deren Eigenschaften

Der Heap Sort-Algorithmus im Detail:

```
procedure heapsort(a, count) is
    input: an unordered array a of length count
    heapify(a, count)
   end ← count - 1
    while end > 0 do
         swap(a[end], a[0])
        (the heap size is reduced by one)
        end ← end - 1
        (the swap ruined the heap property, so restore it)
        shiftDown(a, 0, end)
```

Erstellen eines Heaps aus einem Array: Wenn das kleinste Element die Wurzel ist, wird es an die erste Stelle im Array gesetzt und die Wurzel wird entfernt.

Downheap(): Wiederherstellen des Heaps durch Befolgen der Regeln: Die Wurzelentfernung und Heap-Wiederherstellung wird so lange wiederholt, bis der Heap leer ist.

9.4. Insertion Sort

- Ein einfacher Sortieralgorithmus, welcher der unsortierten Eingabefolge ein beliebiges Element entnimmt und dieses an der richtigen Stelle in die zu Anfang noch leere Ausgabefolge einfügt
- Einfache Implementation
- Effizient bei (ziemlich) kleinen Datensätzen
- Effizient bei Datensätzen, die bereits zum Teil sortiert sind
- Stabil, on-line, in-place

Der Insertion Sort-Algorithmus im Detail:

```
for i = 1 to length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
    end while
end for
```

9.5. Merge Sort

- Ist ein effizienter, allgemein-zweckmäßiger, vergleichsbasierter Sortieralgorithmus
- Verwendet die Teile-und-herrsche-Methode
- Ein stabiler Teile-und-herrsche-Algorithmus, leicht zu parallelisieren
- Grenz- und durchschnittliche Zeit: *O(NlogN)*
- Bedarf an zusätzlichem Speicher: ein Array der Größe N

Der Merge Sort-Algorithmus im Detail:

```
mergesort(m)
      var list left, right
      if length(m) ≤ 1
            return m
      else
           middle = length(m) / 2
           for each x in m up to middle
                add x to left
           for each x in m after middle
                add x to right
left = mergesort(left)
           right = mergesort(right)
           result = merge(left, right)
      return result
```

9.6. Quick Sort

- Effizienter Sortieralgorithmus
- Benötigt: O(NlogN) – O(N2)
- Teile-und-herrsche-Algorithmus, nicht stabil, in-place
- Rekursiv

- Verwendet die folgenden Schritte:
    - Auswahl eines Dreh- und Angelpunkts aus dem Array.
    - Partitionierung: Das Array wird neu geordnet, sodass alle Elemente mit einem Wert, der geringer ist als jener des Angelpunkts, vor ebendiesem gereiht werden. Elemente mit einem höheren Wert werden danach eingeordnet, identische Werte können sowohl vor als auch nach dem Angelpunkt gereiht werden. Nach dieser Partitionierung ist der Dreh- und Angelpunkt an seiner finalen Stelle. Dieser Vorgang nennt sich Partitionsoperation.
    - Die oben genannten Schritte sollen in der Folge auch rekursiv für die beiden Sub-Arrays angewendet werden, welche die Elemente mit den kleineren und größeren Werten enthalten. Hierbei ist darauf zu achten, dass die Schritte in jedem Sub-Array separat angewendet werden.

- Auswahl des **Dreh- und Angelpunkts**: der Median ist ideal
    - Erstes Element (beliebig festgelegte Position) – nicht geeignet für bereits teilweise vorsortierte Datensätze
    - Zufälliges Element – in Wahrheit pseudo-zufällig
    - Median aus drei (fünf…) – oder einer anderen Anzahl an Elementen, die sich an den festgelegten oder zufälligen Positionen befinden

Wenn der Dreh- und Angelpunkt richtig gewählt ist, bedarf es keines zusätzlichen Speicherplatzes. Quicksort ist ein instabiler Algorithmus. Die Methode der Angelpunkt-Auswahl beeinflusst die Sortierung, aber im Schnitt ist es der schnellste allgemeine Sortieralgorithmus für Arrays im operationalen Computerspeicher.

9.7. Selection Sort

Ein einfacher Algorithmus, dessen Zeitkomplexität O (N 2),beträgt. Es ist für kleine Datenvolumina geeignet. Er ist allgemeingültig, lokal und instabil.

Ablauf:

- Unterteilen der Sequenz in einen geordneten und ungeordneten, unbestimmten Teil
- Aufspüren des Elements mit dem geringsten Wert in dem ungeordneten Teil dieser Sequenz
- Ersetzen des Elements durch das Element an der ersten Stelle des ungeordneten

- Das erste Element des ungeordneten Teils wird in den geordneten Teil aufgenommen. Gleichzeitig wird der ungeordnete Teil um ein Element von links reduziert.
- Der Restbestand der Sequenz wird durch das Wiederholen der Schritte 2 bis 5 im verbleibenden ungeordneten Teil geordnet.

Überblick: Vergleich der Sortieralgorithmen

Name	Time complexity			Extra memory	Stable	Natural	Method
	Minimum	Average	Maximum				
Bubble sort	*O(n)*	$O(n^2)$	$O(n^2)$	*O*(1)	yes	yes	Exchange
Heapsort	*O*(*n* log *n*)	*O*(*n* log *n*)	*O*(*n* log *n*)	*O*(1)	no	no	Heap, exchange
Insertion sort	*O(n)*	$O(n^2)$	$O(n^2)$	*O*(1)	yes	yes	Inserting
Merge sort	*O*(*n* log *n*)	*O*(*n* log *n*)	*O*(*n* log *n*)	*O*(log n)	yes	yes	Merging
Quicksort	*O*(*n* log *n*)	*O*(*n* log *n*)	$O(n^2)$	*O*(log *n*)	no	no	Exchanging
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	*O*(1)	no	no	Selection

9.8. Bucket Sort

Dieser Sortieralgorithmus unterteilt Daten in verschiedene Eimer, die dafür benötigte Zeit ist O (n * k), wobei k = n / m, Eingabe n, die Anzahl der Eimer ist m.

Um Bucket Sort verwenden zu können, sind **bestimmte Bedingungen** erforderlich:

- Geeignet für gleichmäßig verteilte Eingabedatenwerte
- Der Algorithmus zur Anordnung der Eimer muss stabil sein

Verfahren:

- Die Eingabedaten werden auf eine vordefinierte Anzahl an Eimern verteilt.
- Für jeden Eimer wird ein stabiler Sortieralgorithmus abgerufen.
- Die individuellen Eimer werden schrittweise in das Ausgabe-Array kopiert.

Die **Vorteile** von Bucket Sort sind, dass der Algorithmus gut parallelisierbar ist und dass er nicht alle Daten gleichzeitig im Speicher benötigt.

9.9. Radix Sort

Der Algorithmus sortiert ganze Zahlen, indem er alle Ziffern durchsucht. Es gibt zwei Ansätze:

- LSD (Least Significant Digit)
- MSD (Most Significant Digit)

Benötigte Zeit: *O((z+n)*logzu)*, wobei z die Basis des ausgewählten Zahlensystem ist, n für die Zahlen bei der Eingabe steht und u dem Maximum an Zahlen bei der Eingabe entspricht.

Radix Sort eignet sich nicht für eine unbegrenzte Eingabegröße.

Beispiel eines LSD-Radix: 170, 45, 75, 90, 802, 2, 24, 66 ⇒ 170, 90, 802, 2, 24, 45, 75, 66 ⇒ 802, 2, 24, 45, 66, 170, 75, 90 ⇒2, 24, 45, 66, 75, 90, 170, 802

9.10. Counting sort

Stabil und geeignet für große Daten mit einer kleinen Menge abstrakter Werte. Die benötigte Zeit ist *O(N+M)* und der zusätzliche Speicherbedarf beträgt *O(M.)*

Voraussetzungen:

- Die Anzahl der verschiedenen Werte (M) ist erheblich geringer als die Gesamtzahl der Elemente (N).
- Hilfs-Array: Es schreibt und liest zeitlich konstant (das Array wird durch Werte oder Hash-Werte indexiert)

Spezifika des Algorithmus:

- Er durchläuft das Eingabe-Array von links (oder von rechts)
- Jedes Element tritt häufiger im Hilfs-Array auf
- Jedem Objekt wird die Zahl des Auftretens aller vorhergehenden Objekte (der Algorithmus erhält die exakte Position der Grenze) beigefügt
- Beginnt, das ungeordnete Array von rechts zu durchlaufen
- Bei jedem Element schaut der Algorithmus in das Hilfs-Array an der Spitze der Anordnungsgrenze
- Counting Sort platziert das Element in dieser Grenze und reduziert es um eins
- Dieser Vorgang wird wiederholt, bis das gesamte Array durchlaufen ist

10. MUSTERERKENNUNG UND PRÄFIXBÄUME

10.1. Mustererkennung

Mustererkennung ist im Wesentlichen die Suche einem bestimmten Muster in einer vorgegebenen Sequenz. Für gewöhnlich wird nach einer Sub-Zeichenfolge in einer Zeichenkette (String) gesucht.

Zunächst muss hierfür ein String defniert werden. Ein String ist eine Sequenz aus Zeichen eines vorgegebenen Alphabets. Ein Alphabet ist ein Sammlung aller möglichen Zeichen, zB ASCII, UniCode, {0,1} oder {A, C, G, T}.

Wenn die Länge des *P*-Strings *m* beträgt, dann besteht der Sub-String *P [i..j]* von P aus Zeichen zwischen *i* und *j*. Der String vor dem Index *i* ist ein Präfix. Der String, welcher sich hinter dem Index *j* befindet, ist ein Suffix.
Anwendungsgebiete: Textverarbeitungsprogramme, Suchwerkzeuge, Biologische Forschung

Es gibt einige Algorithmen für die Mustererkennung:

Der wohl wesentlichste ist der **Brute-Force-Algorithmus**.

Dieser durchläuft den Text von links nach rechts.

Er vergleicht das P-Muster an jeder möglichen Stelle mit dem T-Text bis:

- Eine Übereinstimmung gefunden wird
- Alle möglichen Positionen überprüft wurden

Die Zeit, die dieser Algorithmus benötigt, beträgt *O (nm).*

Der **Boyer-Moore Algorithmus** beginnt den Text vom Ende an zu durchlaufen, sprich von rechts nach links.

Hier gilt es folgendes zu definieren: Der Index *i* zeigt die Stelle im Text *T* an, der Index *j* zeigt auf *P*.

Während der Suche können vier mögliche Situationen auftreten:

- *T (i)* ist nicht in *P* enthalten: In diesem Fall wird i an der Länge von *P* vorbeigeschoben (*P* wird auf den nächsten Buchstaben T ausgerichtet, welcher *T (i + 1)* ist)

- *T (i)* stimmt mit *P (j)* überein – Es werden beide nach links versetzt und der Vorgang wird wiederholt (wie im Brute-Force-Algorithmus)
- *T (i)* ist *P (j), T (i)* steht vor dem Index *j P -> P* wird nach rechts ausgerichtet, damit *T (i)* mit dem Auftreten von *P* übereinstimmt und der Vorgang wird wiederholt.
- *T (i)* ist *P (j), T (i) P* ist ein Index *j -> P* wird um eins nach rechts ausgerichtet und der Vorgang wird wiederholt (nicht wiederkehrend, aber nicht weiter nach unten versetzend)

Rückgabe von *i*, wenn das gesamte Muster gefunden wird.

Dieser Algorithmus ist schneller als der Brute-Force-Algorithmus, allerdings kann dessen Komplexität *O (mn + A)* betragen, wobei A der Größe des Alphabets entspricht.

Um diesen Algorithmus anwenden zu können, ist eine Datenvorbehandlung notwendig.

- In deren Zuge wird die Position der Buchstaben, beginnend von links, herausgefunden
- Sieht das Muster beispielsweise wie folgt aus: "ABRAKADABRA",
- So erhält A den Index 10, *B* erhält den Index 8, *K* = 4, *D* = 6 und *R* = 9. Anderen Buchstaben wird der Wert -1 zugewiesen.
- Im Anschluss wird die Funktion Last (char input) implementiert, welche den Index gemäß den Buchstaben zurückgibt. Dann, in den Fällen 3 und 4, werden der letzte *(T (i))* und *j* verglichen, woraus sich schließen lässt, ob der Algorithmus an die Stelle Last *(T (ii)* (wenn Last *(T (i))* <j ist) oder nur *i*++ versetzt werden soll.

Knuth-Morris-Pratt (KMP) Algorithmus

Dieser Algorithmus sucht den Text von links nach rechts und unterscheidet sich dadurch vom Brute-Force-Algorithmus, welcher nicht alle Vergleiche durchführt. Wenn eine Unstimmigkeit gefunden wird, verschiebt sich der Algorithmus um mehr als einen Buchstaben. Wenn ein *P* (vom Anfang, ein Präfix) gefunden wird, stimmen die Zeichen dieses Präfix' mit dem Text überein, weshalb keine Notwendigkeit für eine erneute Prüfung besteht. Das Ende des gefundenen Sub-Strings kann auch am Anfang dieses Sub-Strings enthalten sein. Solch eine Übereinstimmung entspricht klarerweise dem ganzen gefundenen *P*, weshalb nach einem *P + 1* gesucht wird. Deshalb geht man vom linken und rechten Ende des gefundenen *P*-Stücks aus, und wenn man keine Übereinstimmung findet, wird klar, wie viel man verschieben kann. Dies kann in einer Tabelle gerechnet werden: In diesem Fall ist alles *O(1).*

10.2. Präfixbaum

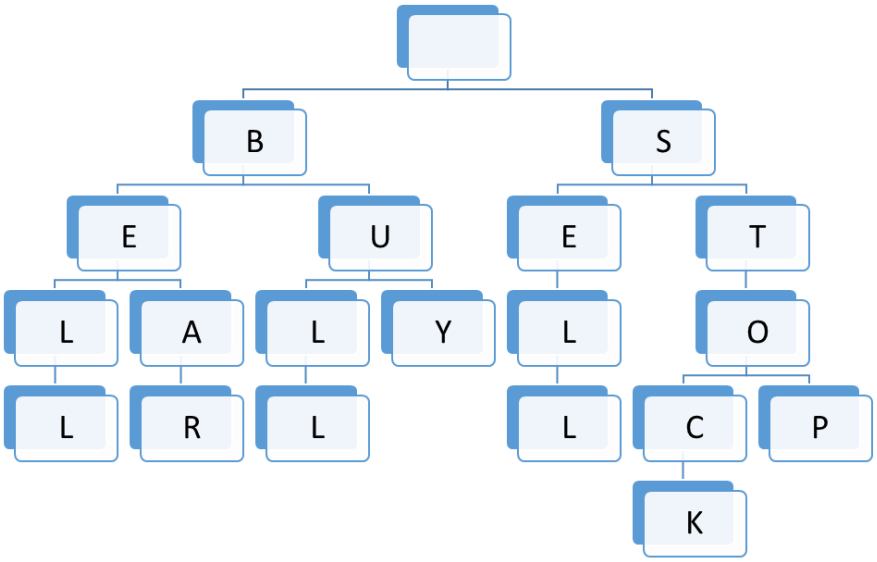
Ein Präfixbaum ist eine Baumstruktur, die dazu dient, einen Text vorzubereiten. Jeder Knoten hat einen Buchstaben. Die Länge des Pfads vom Knoten zur Spitze stellt die Position des Buchstabens im Wort fest.

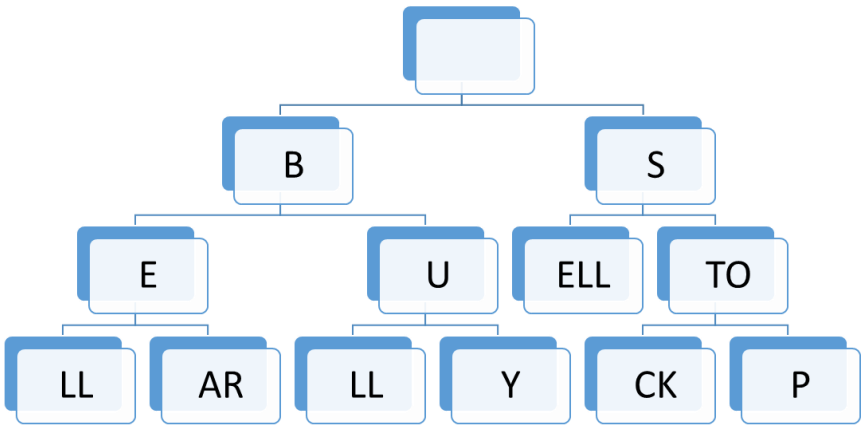
Die Suche erfordert eine bestimmte Zeit *O (dm),* wobei d der Größe des Alphabets und m der Länge des Worts entspricht.

Es ist möglich, den gesamten Text in einer Präfixbaum-Struktur zu speichern. Nach diesem Vorgang enthält jeder Knoten ein Wort.

Zum Speichern kann ein sogenannter komprimierter Präfixbaum definiert warden. Dieser Baum enthält dann zumindest Knoten zweiten Grades (zwei Buchstaben pro Knoten).

Beispiel: S={BELL, BEAR, BULL, BUY, SELL, STOCK, STOP}





11. GRAPHENTHEORIE

Ein Graph ist ein geordnetes Paar *(V, E)*, wobei *V* einem Set aus Scheitelpunkten und E einem Set von Kanten entspricht. Jede Kante wird nur mittels zweier Scheitelpunkte bestimmt, oder optional durch die Richtung oder das Gewicht.

11.1. Arten von Kanten:

- **Orientierte**: geordnetes Paar von Scheitelpunkten *(u, v),* wobei u der Anfang und v das Ende bzw. das Ziel ist
- **Orientierungslose:** geordnetes Paar von Scheitelpunkten *(u, v)*
- **Schleifen:** Kante beginnt und endet am selben Scheitelpunkt
- **Multiple Kanten (mehrfach, parallel):** mehrere Kanten zwischen den Scheitelpunkten *(u, v)*

11.2. Arten von Graphen

- **Orientierte:** Alle Kanten sind ausgerichtet
- **Orientierungslose:** Keine Kante ist ausgerichtet
- **Multigraph:** enthält mehrere Kanten

11.3. Terminologie

- Endscheitelpunkte (oder Endpunkte) einer Kante
- Kanten-Störfall an einem Scheitelpunkt
- Benachbarte Scheitelpunkte
- Grad eines Scheitelpunkts
- Parallele Kanten
- Eigen-Schleife
- Pfad
    - Sequenz alternierender Scheitelpunkte und Kanten
    - Beginnt mit einem Scheitelpunkt
    - Endet mit einem Scheitelpunkt
    - Direkt vor und nach jeder Kante sind deren Endpunkte
- Einfacher Pfad
    - Ein Pfad, bei dem sich alle Scheitelpunkte und Kanten voneinander unterscheiden

- Zyklus
    - Zirkuläre Sequenz alternierender Scheitelpunkte und Kanten
    - Direkt vor und nach jeder Kante sind deren Endpunkte
- Einfacher Zyklus
    - Ein Zyklus, bei dem sich alle Scheitelpunkten und Kanten voneinander unterscheiden

11.4. Graph-ADT-Funktionen

Zugangsfunktionen

- aVertex()
- incidentEdges(v)
- endVertices(e)
- isDirected(e)
- origin(e)
- destination(e)
- opposite(v,e)
- areAdjecent(v,w)

Update-Funktionen

- insertVertex(o)
- insertEdge(v, w, o)
- insertDirectedEdge(v, w, o)
- removeVertex(v)
- removeEdge(e)

Allgemeine Funktionen

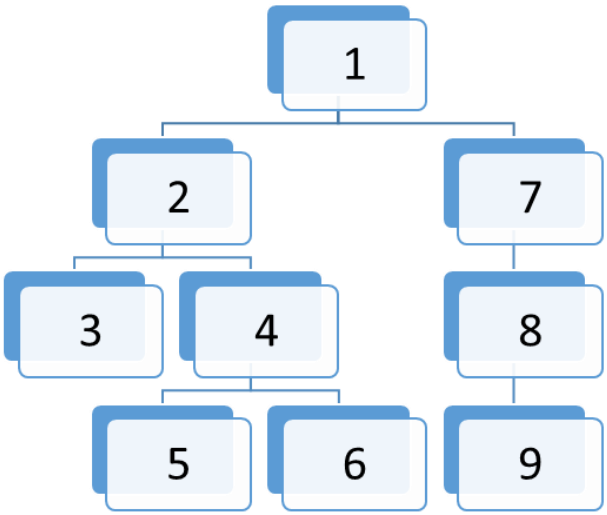
- numVertices()
- numEdges()
- vertices()
- edges()

Die Graphen-Struktur muss irgendwie durchsucht werden. Hierfür gibt es zwei Optionen: die DFS (Depth-First Search) und die BFS (Breadth-First Search).

11.5. Depth-First Search

Die **Depth-First Search** ist ein vollständiger Algorithmus, welcher jeden Knoten durchläuft. Sein Prinzip besteht im Umstand, dass er den ersten Nachfolger jedes Höchstwerts vergrößert, wenn dieser noch nicht besucht wurde. Wenn er auf einen Höchstwert stößt, von dem aus es nicht möglich ist den Vorgang fortzusetzen (es gibt keine Nachfolger oder alle davon sind bereits besucht worden), geht er mittels Rückverfolgung zurück.

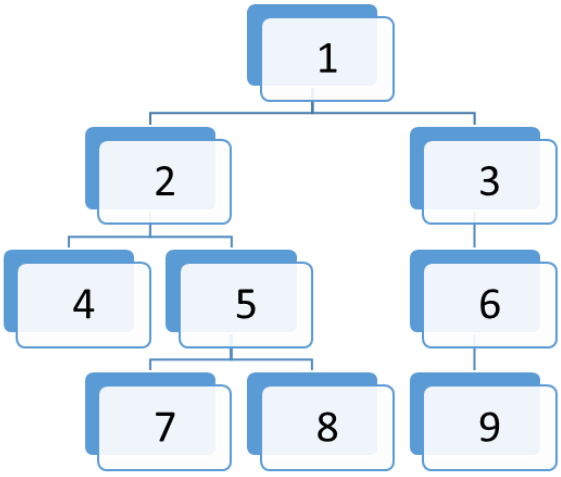
Er durchläuft die Knoten in der folgenden Reihenfolge:



11.6. Breadth-First Search

Breadth-First Search ist ein ähnlicher Algorithmus, der alle Nachbarn ausgehend vom anfänglichen Höchstwert durchläuft, danach die Nachbarn der Nachbarn usw., bis die gesamte Verbindungskomponente durchlaufen ist.

Er durchläuft die Knoten in der folgenden Reihenfolge:



Eine essentielle Frage der Graphen-Theorie ist es, wie man den kürzesten Pfad zwischen zwei Knoten findet. Hierfür gibt es verschiedene Algorithmen.

Dijkstras Algorithmus

- Keine negativen Kantenwerte
- $O(|V|2+|E|)$ – V Anzahl der Scheitelpunkte, E Anzahl der Kanten

Bellman-Ford Algorithmus

- Graph kann negative Kanten haben
- $O(V\,E)$ – langsamer als Dijsktras Algorithmus

Floyd-Warshall Algorithmus

- Ausgerichteter/orientierter Graph ohne negative Kanten
- Findet den kürzesten Pfad zwischen allen Scheitelpunkten
- Benötigte Zeit: $O(V3)$, benötigter Speicher: $O(V^2)$

Johnsons Algorithmus

- Orientierter Graph, kann negative Kanten enthalten
- Findet den kürzesten Pfad zwischen allen Scheitelpunkt-Paaren in einem zerstreuten, nach Kanten gewichteten, ausgerichteten Graphen.
- Erlaubt, dass einige Kantenwerte negative Zahlen sind
- O(V2 log2(V)+ VE)

12. GENETISCHE ALGORITHMEN

12.1. Erklärung

Genetische Algorithmen gehören zu den evolutionären Algorithmen und sind Teil der künstlichen Intelligenz. Sie sind eine Klasse der heuristischen Algorithmen. Sie nutzen das Wissen der Evolutionsbiologie, um komplexe Probleme zu lösen, für die es keine exakten Algorithmen gibt. Sie imitieren dazu die Methoden der Evolutionsbiologie:

- Vererbung
- Mutation
- Natürliche Auslese
- Kreuzung

Genetischer Algorithmen arbeitet nach dem im folgenden Schema ausgedrückten **Prinzip**:

- **Initialisierung:** Schaffung einer Generation 0
- **Beginn des Zyklus:** Zufällige Auswahl einiger Individuen aus der gesamten Population
- **Schaffen einer neuen Generation** durch die Verwendung folgender Methoden:
    - *Kreuzung:* "Austauschen" von Teilen weniger Individuen
    - *Mutation:* zufällige Veränderung einiger Gene
    - *Reproduktion:* Kopieren von unveränderten Individuen
- **Berechnung der Tauglichkeit** der neuen Generation
- **Beendigung des Zyklus** – Der Vorgang wird von Punkt 2 an wiederholt, bis eine Endbedingung erreicht ist
- **Ende des Algorithmus** – Das Individuum mit der höchsten Leistungsfähigkeit ist die Ausgabe des Hauptalgorithmus und repräsentiert die bestmögliche Lösung

12.2. Terminologie

- **Phänotyp:** Kennzeichnung eines Individuums
- **Erbgut, Genom, Chromosom:** Verkörperung eines Phänotyps
- **Chromosom:** unterteilt in verschiedene, linear angeordnete Gene (*i* -th chromosomische Gene derselben Art, welche dasselbe Merkmal repräsentieren)
- **Allele:** verschiedene Gen-Werte
- **Fitness-Wert:** reicht von 0-1, drückt die Qualität jedes Individuums aus

Jedes Individuum kann auf unterschiedliche Weise kodiert (genetisch beschrieben) werden. Die Beschreibungsmethode entscheidet über Erfolg oder Misserfolg beim Lösen einer speziellen Aufgabe.

12.3. Beispiel:

Generation 0 (Fitness-Wert # "1"):

- 0100011011 *f=0,5*
- 0101000100 *f=0,3*
- 1010110000 *f=0,4*
- 1110111000 *f=0,6*

Auswahl

- *Gewichtete Rollkurve:* $p_i = \frac{f_i}{\sum_1^N f_i}$
    - Wahrscheinlichkeit, ein Elternteil zu sein

- *Turnier-Methode*
    - Aus den Gruppen jeder Eltern-Gruppe wird die Person mit dem höchsten Fitness-Wert ausgewählt

- *Beschneidung*
    - Alle Individuen werden gemäß des f-Werts sortiert, der geringwertige Teil wird abgeschnitten, es werden Eltern aus dem restlichen Pool ausgewählt

- *Zufällige Auswahl*
    - Einfachste Methode: der f-Wert spielt keine Rolle in der Auswahl von Individuen für die Aufgabe der weiteren „Erziehung“

- *Kreuzung*
    - Eltern tauschen Teile ihres genetischen Codes aus
    - Die einfachste Methode ist die Einpunkt-Kreuzung
    - Die Stelle für die Beschneidung wird zufällig ausgewählt

    - X: 010001|1011
    - Y: 111011|1000
    - P: 0100011000 *f=0,3*
    - Q: 1110111011 *f=0,8*

    - Die Mehrpunktkreuzung bietet die Möglichkeit, mehr als 2 Eltern zu kreuzen

- *Mutation*
    - Zufällige Veränderung zufälliger Erbfaktoren in einem Individuum
    - Sehr geringe Wahrscheinlichkeit

- 0100011011 ⇒ 0101011011
- 0101000100 ⇒ 0101100100
- 1010110000 ⇒ 1010110100
- 1110111000 ⇒ 1010111000

- Es ist möglich, Eigenschaften zu erhalten, welche sich nicht in der ursprünglichen Generation finden

Beendigung

- Dieser allgemeine Prozess wird wiederholt, bis die Abschlussbedingungen erreicht sind. Gebräuchliche Abschlussbedingungen sind:

  - Es wird eine Lösung gefunden, welche die Minimal-Kriterien erfüllt
  - Eine festgelegte Anzahl an Generationen wird erreicht
  - Das vorgegebene Budget (Rechenzeit/Geld) wird erreicht
  - Die Fitness der am besten bewerteten Lösung erreicht bzw. erreichte eine Ebene, auf der erfolgreiche Iterationen zu keinen besseren Resultaten mehr führen
  - Manuelle Überprüfung
  - Eine Kombination aus den oben genannte