Prüfungsprotokoll Datenstrukturen (1663)

Datum: 13.11.2006
Prüfer: Prof. Dr. Schlageter
Dauer: ca 20 min

Was ist ein Datentyp?
Wie kann man ihn (den Datentyp) beschreiben?
mit der Frage hat er mich gleich erwischt. Ich habe zwar irgendwas gesagt, aber so ganz richtig war es dann wohl doch nicht. → Also durchaus auch die erste KE nochmal anschauen. Prof. Schlageter hat dann glücklicherweise von weiteren Nachfragen abgesehen und hat das Thema gewechselt.

Welche Strukturen für Ditcionaries gibt es ?
(sortiert nach Grösse der Objektmengen für die sie sich eignen)
Listen, Hashing, Bäume, AVL-Baum, B-Baum

Thema Listen: Was ist der Vorteil von doppelt verketteten Listen?
Ich habe was vom Vorteil beim Löschen erzählt. Wenn ich ein Element löschen will, dann kennt man ja den Vorgänger und den Nachfolger von dem Element. Und so kann man leicht die beiden dann miteinander verketten. Ohne Doppelverkettung müste man sich immer das Vorgänger-Element merken.

Thema Hashing: Wann würden Sie Hashing verwenden?:
Unterschied zwischen offenen und geschlossenen Hashing?
Welchen Probleme können auftreten?
Welche Kollisionsstrategien gibt es?
Wo bringt das quadtratische Sondieren Vorteil?
Wann: geringe Updatewahrscheinlichkeit, wenig Löschvorgänge...
Unterschied: siehe Skript bzw. alte Prüfungsprotokolle.
Probleme: Kollisionen (mit Erklärung)
Kollisionsstrategien: lineares Sondieren, quadratisches Sondieren, Doppel-Hashing
(jeweils kurze Erklärung)
Vorteil quatr. Sond. : bei Sekundärkollisionen.

Thema AVL-Baum: Was ist ein AVL-Baum?
Was für eine Laufzeit garantiert er?
Was ist mit dem Aufwand um in „balance“ zu bleiben?
Was: Ein AVL-Baum ist ein ausgeglichener Baum.
Der AVL-Baum ist eine Abschwächung des Balanciertheitskriteriums.
[Ein Baum heißt balanciert, wenn eine maximale Tiefe von **log n** garantiert wird.]
Die Abschwächung besteht darin, das die Tiefe der Teilbäume eines jeden Knotens sich maximal um 1 unterscheiden darf.
Maximale Tiefe ist maximal 44% über **log n**.
Laufzeit : O(log n) – aufgrund der Tiefe.
Aufwand: Wenn Knoten aus Balance geraten ist, dann EinfachRotation oder Doppelrotation.
Beim Rotieren müssen zwei Knoten und die Teilbäume angefasst werden
→ Konstanter Aufwand. Konstanten werden in der O-Notation „weggelassen“.

Thema B-Bäume: Wo landet ein Element wenn man es in den Baum einfügen will? Wann und wie wächst ein B-Baum?

Wo: immer in einem Blatt.

Wann: Wenn das Blatt schon voll ist, dann Overflow (kurz erklären) . Das kann sich dann sich dann bis zur Wurzel hin weiterziehen. Und wenn die Wurzel voll ist, dann wächst der B-Baum.

Jetzt kommen wir zu Sortierverfahren: Welche Aufwände gibt es bei den Verfahren?

einfache Verfahren: $O(n^2)$ und gute Verfahren $O(n \log(n))$.

Welche Verfahren kennen Sie denn mit $O(n \log(n))$?

Merge-Sort, Quick-Sort, Heap-Sort

Worin unterscheiden sich Quick- und Heap-Sort?

Im worst-case hat der Quick-Sort eine Laufzeit von $O(n^2)$ und der Heap-Sort garantiert auch im worst-case eine Laufzeit von $O(n \log(n))$.

Wieso entartet der Heap-Sort nicht?

Weil der als Grundlage einen partiell geordneten Baum hat. Das globale Minimum ist bei diesem Baum immer in der Wurzel.

Woher kommt denn beim Quicksort das zweite n?

Jedes Element muss ja mindestens einmal betrachtet werden.

Warum zieht man trotzdem oftmals den Quick-Sort dem Heap-Sort vor?

Weil der Quicksort im Durchschnitt etwas schneller ist als der Heapsort (kurze Erklärung, beim Heapsort immer 2 Vergleiche....).

Außerdem kann beim Quicksort das Splitelement in einem schnellen Register gehalten werden.

Und zum Schluss nochmal kurz zu Graphen. Wie kann man einen Graphen darstellen?

Adjazenzmatrix und Adjazenzlisten.

Was ist eine Adjazenzmatrix?

Eine boolsche n*n Matrix (True – heißt es gibt eine Kante, und False – heißt es existiert keine Kante zwischen den entsprechenden Knoten).

Anstatt einer boolschen Matrix können aber auch die Kosten (z.B. die Entfernung) zwischen den Knoten drin stehen.

Was für ein Algorithmus gibt es zum Finden des kürzesten Weges von einem Knoten zu allen anderen? Und wie funktioniert er??

Djikstra – Algorithmus. Funktionsweise siehe Skript...

Fazit:

Prof. Schlageter ist absolut empfehlenswert. Mit seiner ruhigen Art nimmt er einem die Nervosität. Die Benotung ist ausgesprochen fair.

Prüfung Kurs 1663
Prüfer: Prof. Schlageter
Termin: 04.09.2006

Fragen:
- Was verstehen Sie unter einem Datentypen
- Wie kann man ihn beschreiben
- Was ist ein Algorithmus
- Welche Anforderungen hat man ihn

- Welchen Vorteil bieten doppelt verkettete Listen
- Welche Strukturen für Dictionaries gibt es
  (Aufsteigend sortiert nach Grösse der Objektmengen für die sie sich eignen)
- Wann würden Sie Hashing verwenden, wann nicht
- Welche Laufzeit im günstigsten Fall
- Welche Laufzeit im worst case
- Welches Problem tritt beim Hashing auf
- Wie begegnet man ihm (dem Problem)
- Welche Struktur für eine grössere Datenmenge?
- Warum kann die Laufzeit beim binären Suchbaum zu O(n) entarten
- Wo "landet" ein im B-Baum eingefügtes Element
- Wo kann im B-Baum ein Overflow stattfinden

- Wann ist ein Sortieralgorithmus "gut"
- Welche Verfahren mit dieser Laufzeit kennen Sie
- Welche Laufzeit hat Quicksort wenn er entartet
- Warum kann Heapsort nicht entarten
- Warum entscheidet man sich trotzdem oft für Quicksort
- Woher kommt das "erste n" in O(n*logn) bei Quicksort-Laufzeit

- Welche Strukturen kennen sie zur Darstellung eines gerichteten Graphen
- Was ist eine Adjazenzmatrix
- Was sagt ein Eintrag in der Adjazenzmatrix aus
- Wozu benötigt man Adjazenzmatrix und Adjazenzliste

Fazit:
Ich kann mich auch nur allen anderen anschliessen: Prof. Schlageter ist als Prüfer uneingeschränkt
zu empfehlen. Der Prüfungsbeginn hatte sich leicht verzögert - was bei mir leider ausgereicht hat
meine "übliche Nervosität" in hirnleistungsblockierende PrüfungsPANIK umzuwandeln. Durch
Prof. Schlageters ruhige, humorvolle und unaufgeregte Art hab ich nach den ersten Fragen doch
noch die Kurve bekommen. Netterweise wurde mein eher unbeholfenes Gestammel zu Beginn
nicht mit in Gesamtbewertung einbezogen.

Prüfungsprotokoll

Kurs: 1663 Datenstrukturen
Prüfling: Günther Rasch
Prüfer: Prof. Dr. G. Schlageter
Datum: 19.12.2005 / 10:30 Uhr

Listen:

- Sie sollen eine Firma bzgl. eines Datenstammes beraten. Was muss man zu den Datenstrukturen wissen?
    - Größe der Datenstruktur
    - Updatewahrscheinlichkeit
    - Dynamische oder statische Datenstruktur....
- Sie haben eine kleine Menge, welche Datenstruktur?
    - Listen
- Es gibt verschiedene Arten von Listen, u.a. Stacks. Welche Besonderheiten haben Stacks?
    - Man kann Elemente nur oben drauflegen und oben wieder entnehmen, also eine LIFO-Struktur
    - Stacks sind ein Spezialfall von Listen ($list_1$)
- Listen, bei denen man an einem Ende anhängen und am anderen Ende entnehmen kann?
    - Queues, FIFO-Struktur
    - ebenfalls ein Spezialfall von Listen

Dictionaries:

- Wir haben nun viele Daten, also Hashing. Welche Probleme gibt es beim Hashing und wie beseitigt man diese?
    - Das Problem sind Kollisionen, man beseitigt diese durch Kollisionsstrategien:
        - lineares, quadratisches Sondieren und Doppelhashing
- Warum ist das beim offenen Hashing kein Problem
    - Beim offenen Hashing fasst man die Buckets als beliebig erweiterbare Listen von Schlüsselfolgen auf
- Welche Laufzeiten hat Hashing
    - O(n) im worst case
    - O(1) im Durchschnitt
- Welche weiteren Dictionary-Implementierungen gibt es?
    - Binäre Suchbäume
    - AVL-Bäume
    - B-Bäume
- Was ist ein AVL-Baum?
    - AVL-Baum ist ein balancierter Baum
    - für jeden Knoten eines Baums unterscheidet sich die Höhen seiner Teilbäume höchstens um 1
- Was heisst hier balanciert?
    - Eigentlich ausgeglichen: für jeden Knoten in einem Baum unterscheidet sich die Anzahl der Knoten in seinem Teilbäumen höchstens um 1
- Welche Laufzeiten bietet der AVL-Baum und warum?
    - O(log n)
    - weil der AVL-Baum balanciert ist
- Wie schafft man das, dass der AVL-Baum ausgeglichen bleibt?
    - Rotation / Doppelrotation
- Beim Suchen ist O(log n) klar, was ist aber mit Einfügen / Löschen, genauer mit dem Aufwand beim Rotieren? Warum taucht dieser in der Laufzeit nicht auf?
    - Beim Rotieren müssen zwei Knoten und die Teilbäume angefasst werden --> O(1) --> konstant --> O-Notation eleminiert Konstanten!
-

- Kommen wir zu B-Bäumen. Ein Element soll eingefügt werden, wie funktioniert das? Wo landet das Element?
    - Man vergleicht die Schlüsselwerte, < nach links, > nach rechts
- Was ist zu beachten, wenn der Knoten voll besetzt ist
    - Overflow erklären
- Wann wächst der B-Baum?
    - Die Overflow-Behandlung kann sich bis zur Wurzel fortpflanzen

Sortieralgorithmen:

- Welche Verfahren gibt es?
    - Verfahren mit $O(n^2)$ und $O(n \log n)$ – Verfahren aufgezählt
- Welche Verfahren gibt mit $O(n \log n)$
    - Quicksort, Mergesort und Heapsort
- Wie unterscheiden sich Quicksort und Heapsort?
    - Laufzeit im worst case ($O(n^2)$ zu $O(n \log n)$)
- Warum garantiert Heapsort $O(n \log n)$ im worst case
    - wg. des partiell geordnete Baums
- Was ist ein partiell geordneter Baum, was ist hier besonderes?
    - Das globale Minimum steht in der Wurzel
- Warum hat Quicksort $O(n^2)$ im worst case?
    - Weil der rekursive Aufrufbaum bei ungeschickter Wahl des Splitwerts entarten kann
- Wie setzt sich das $O(n^2)$ zusammen?
    - $O(n)$ Rekursionsschritte, $O(n)$ Divide-Schritte
- Woher kommt das zweite n?
    - Jedes Element muss beim Divideschritt mindestens einmal betrachtet werden

Graphen:

- Ein Graph ist eine Menge von Knoten zusammen mit einer Relation, den Kanten. Wie kann man einen Graphen darstellen?
    - mit einer Adjazenzmatrix, diese ist eine boolesche nXn-Matrix, die booleschen Werte geben an, ob eine Kante zwischen zwei Knoten existiert oder nicht
- man kann statt der booleschen Werte auch Kantenmarkierungen in die Matrix eintragen. Was stellen denn diese Markierungen dar?
    - Bewertung, Gewichtung, z.B. Entfernung zwischen zwei Punkten

Fazit:

Wie schon in allen anderen Protokollen erwähnt: Hr. Prof. Dr. Schlageter ist als Prüfer uneingeschränkt zu empfehlen, er nimmt dem Prüfling die Nervosität durch einen Plausch am Beginn der Prüfung, während der Prüfung erklärt er auch Zusammenhänge genauer – auch dies in einem angenehmen Gespräch. Die Schwerpunktthemen sind aus den Fragen ja ersichtlich, es kam nichts zur Sprache, was nicht schon in den Protokollen erwähnt wurde. Wichtig ist, dass man nicht zu kompliziert und in Definitionen denkt! Diese wollte Hr. Prof. Schlageter kein einziges Mal wissen. Wichtiger war ihm, dass man verstanden hatte, wie ein bestimmter Sachverhalt funktioniert (z.B. O-Notation, wie kommen diese Zeiten zustande / Einsortieren in einen B-Baum usw.). Ich zumindest hatte hier einige Male Probleme zu verstehen, was er genau hören wollte, weil ich einfach um ein paar Ecken zu kompliziert dachte – dabei waren die Antworten einfach und ganz und gar nicht kompliziert.
Also nochmal, uneingeschränkte Empfehlung von mir für diesen absolut super netten und sympathischen Prüfer!

Prüfungsprotokoll zur Vordiplomsprüfung
Datenstrukturen (1663)
bei **Prof. Dr. Schlageter**
März.2004 (ergänzt Juni 2004!)

Themen:
Datenstrukturen: Anwendungen und Eigenschaften von einfachen Listen, Hashing-Verfahren, AVL-Bäumen, B-Bäumen
Sortierverfahren: Vergleich Quicksort/Heapsort bzgl. Laufzeit
Graphen: kurze Beschreibung des Dijkstra-Algorithmus und des Floyd-Algorithmus (neu!)

Ablauf/Atmosphäre:
Es ist nichts vorgekommen, was nicht schon in den Protokollen steht. Ich musste nichts aufschreiben oder exakt definieren. Ich habe alles recht blumig beschreiben können. Es gab Zwischenfragen, die aber nie wirklich in die Tiefe gingen. Mit dem Ergebnis kann ich mehr als zufrieden sein.
Herr Schlageter hat mir am Anfang mit einem Plausch die Aufregung genommen und war während der ganzen Prüfung außerordentlich freundlich. Am Anfang habe ich mit Kanonen auf Spatzen (bzw. Bäumen und Hashes auf eine Handvoll Daten) schießen wollen, was sich in er Note aber nicht niedergeschlagen hat.

Herr Schlageter ist als Prüfer sehr zu empfehlen!

Protokoll der mündlichen Diplomvorprüfung

Datenstrukturen (1663)

Datum: 16.11.99

Prüfer: Prof. Dr. Schlageter

Note: 1,0

Dauer: ca. 15 Minuten

Die Frage kennen Sie sicher schon aus alten Prüfungsprotokollen: Sie sollen für ein Unternehmen eine Kundendatei aufbauen. Was müssen Sie wissen um eine geeignete Datenstruktur auszuwählen?

- Updatewahrscheinlichkeit, Anzahl der Datensätze

Nehmen wir an wenig Datensätze, geringe Updatewahrscheinlichkeit.

- Hashing

Warum sind Hashverfahren für dynamische Datenstrukturen nicht geeignet?

- Elemente werden bei geschlossenen Hashing nicht gelöscht, sondern nur als 'deletet' gekennzeichnet
- wenn die Tabelle mehr als 80% gefüllt ist, wird das Laufzeitverhalten sehr schlecht

Warum wird das Laufzweitverhalten schlecht? Was passiert beim geschlossenen Hashing?

- Kollision, Kollisionsbehandlung (Rehashing), Kollisionsstrategien (lineares/quadratisches Sondieren, Doppelhashing)

Kommen wir zu Sortierverfahren: Wo liegt der Unterschied in den beiden wohl bekanntesten Sortierverfahren: Heapsort und Quicksort?

- Worst-case bei Quicksort n^2
- Quicksort ist im Durchschnitt schneller, weil Heapsort 2 Vergleiche in der Schleife benötigt.

Woher kommt das worst-case Verhalten von n^2 bei Quicksort?

- Worst-case: Aufteilung in 1 und n-1 Elemente beim Divide.
- Kommen wir noch einmal kurz zu B-Bäumen. Wie wächst ein B-Baum?Ein neues Element wird in einem B-Baum jeweils in einem Blatt eingefügt.
- Wenn ein Blatt überläuft (2m+1 Elemente), wird ein Split durchgeführt.
- Dabei kann wiederum der Vaterknoten überlaufen, das kann sich bis zur Wurzel fortpflanzen. Wenn die Wurzel überläuft, wächst der B-Baum.

Fazit: Angenehme Prüfung, lockere Atmosphäre. Bei der Vorbereitung habe ich mich an den alten Prüfungsprotokollen orientiert.

Kurs: Datenstrukturen
Prüfer: Prof. Schlageter
Datum: 14. Januar 1998

- Die erste Frage kennen Sie sicherlich schon: was für Datenstrukturen würden Sie einem Unternehmen mit großen Kundenstamm empfehlen?
Ich habe geantwortet, daß ich noch nicht genügend Informationen habe um dies zu entscheiden. Ich habe dann einen kurzen Vortrag gehalten, was ich noch alles wissen müßte und habe was über Bäume, Hashverfahren, etc. erzählt.
- Viele Kunden, fester Kundenstamm!
Hier habe ich einen Kurzvortrag über Hashingverfahren gehalten.
- Was für Probleme können hierbei auftreten?
Ich habe Kollisionen, Clusterbildung und Probleme des Löschens erwähnt.
- Und wenn ich viele Kunden habe, die stark wachsen?
Ich habe dann AVL-Bäume kurz erwähnt und ausführlich über B-Bäume berichtet, die hier wohl die bessere Wahl sind.
- Was ist der Unterschied zwischen balancierten Bäumen und vollständig ausgeglichenen Bäumen?
Daraufhin habe ich gesagt, daß ich balanciert eigentlich nicht so schön als Begriff finde und ich lieber ausgeglichen sagen will. Ich habe dann die Definition von ausgeglichen und vollständig ausgeglichen angegeben. Kurz noch etwas über das ausgleichen *(Doppelrotation, Einfachrotation)* erzählt.
- Wann wächst ein B-Baum?
Hier habe ich wieder sehr ausführlich den Underflow erklärt.
- Was kann beim QuickSort passieren?
Ich meinte, der QuickSort könnte zum SlowSort werden und habe wiederum sehr detailliert erklärt was da passieren kann. Hier wollte Prof. Schlageter mir die Antwort schon vorwegnehmen, da das wohl etwas zu ausführlich war. Ich habe dann geantwortet, daß das passiert, wenn der Rekursionsbaum entartet. Deshalb hat man statt O(log(n)) O(n) Divide Schritte und deshalb O(n^2) im worst case.
- Woher kommt jetzt das zweite n?
Weil ich alle Elemente letztendlich einmal betrachten muß.

Fazit:
Prof. Schlageter ist ein sehr freundlicher Prüfer. Meiner Meinung nach legt Prof. Schlageter sehr viel Wert darauf, ob man ein Gebiet verstanden hat und dieses auch kommunizieren kann. Dabei ist ihm offensichtlich bewußt, daß - zumindest einige der Fragen - bereits aus alten Prüfungsprotokollen hinreichend bekannt sind. Insofern eignen sich die Prüfungsprotokolle hervorragend zur Vorbereitung. Prof. Schlageter ist uneingeschränkt zu empfehlen.

Prüfungsprotokoll Diplomvorprüfung Informatik

Prüfungsinhalt: Datenstrukturen und Algorithmen

Prüfer: Prof. Dr. Schlageter

Datum: 1997

Fragen

Für diese Prüfung habe ich mir aus der Vielzahl der vorhandenen Prüfungsprotokolle einen Frage- und Antwortkatalog zusammengestellt. Alle Fragen meiner eigenen Prüfung sind darin enthalten.

Die bevorzugten Themen sind die Kapitel: 4.2, 4.3, 6.2, 6.3, 6.4, 8.1, des Kurses 1663 in der Ausgabe WS 93/94.

Was wissen Sie über Hashing?

- Grundidee des Hashing (Dictionary-Implementierung, 2 Arten: offenes und geschlossenes Hashing)
- Erläuterung des geschlossenen Hashing mit Kollisionsstrategien
- Definition von Primär- und Sekundärkollisionen
- Praktisches Beispiel für lineares und quadratisches Sondieren (Array zeichnen und andeuten, wo zwei Werte abgespeichert werden)
- Linear Probing, quadratic Probing

Warum ist quadratic Probing besser als linear Probing?

- Weil Sekundärkollisionen dadurch besser vermieden werden
- Unabhängige Hashfunktionen
- Zeitkomplexität O(1)
- Geschlossenes Hashing entartet bei Arrays, die zu mehr als 80% gefüllt sind
- Offenes Hashing entartet, wenn die Listen hinter den Slots zu lang werden

Was ist Hashing?

Wie kann man Kollisionen vermeiden?

Was ist der Vorteil beim quadratischen Sondieren?

Wann sollte man kein Hashing benutzen?

- Hashtabelle mehr als 80% gefüllt
- Geschlossenes Hashing ist für dynamische Update- (insbesondere Lösch-) Operationen nicht geeignet

Ein Unternehmen will für seine Kundendatei Hashing-Verfahren einführen. Wie würden Sie das Unternehmen beraten?

- Fragen nach Dateigrößen
- Updatewahrscheinlichkeiten
- Sollen die Daten auch sortiert ausgegeben werden? Etc.

Angenommen, wir hätten uns für das Hash-Verfahren entschieden. Was ist zu tun?

- Dynamisches oder geschlossenes Hashing wählen
- Notwendige Tabellengröße ermitteln
- Hashfunktion finden (jeweils mit Entscheidungsgründen)

Wie können Sekundär-Kollisionen beim Hashing behandelt werden?

- Lineares Sondieren und Doppelhashing mit unabhängigen Funktionen

Was sagen Sie zu quadratischem Sondieren?

Wo bringt quadratisches Sondieren eine Verbesserung?

Was passiert eigentlich beim Löschen?

Kurzvortrag Hashing

Was verstehen Sie unter Hashing?

- Sinn und Zweck des Hashing
- Eigenschaften der Hashfunktion
- Kollisionsstrategien
- lineares Sondieren -> führt zur Kettenbildung
- quadratisches Sondieren (primär <-> sekundär Kollision) Voteil: Herabsetzung der Wahrscheinlichkeit für Kettenbildung
- oppel-Hashing -> nah am idealen Hashing -> problematisch: Nachweis der Unabhängigkeit der Funktionen

Welche Probleme gibt es beim Hashing?

- schlechte worst-case Komplexität O(n)
- Löschen eines Elements: Kennzeichnung als deletet – Unterschiedliche Behandlung bei weiteren Inserts und Abfragen

AVL-Bäume

Andere Datenstruktur für das gleiche Problem?

- Definition AVL-Bäume
- Probleme beim Einfügen und Löschen (evtl. Einfach- oder Doppelrotation nötig zum Rebalancieren)
- Zeitkomplexität für Einfügen, Löschen (spez. Rotationen), Suchen

Was ist ein AVL-Baum?

Welche Zugriffszeit hat ein AVL-Baum?

- O(logn) für Einfügen, Löschen und Suchen

Warum sind hier alle drei Operationen in O(logn) Zeit realisierbar?

Warum ist Hashing für obige Aufgabe nicht geeignet?

Welche Datenstruktur sollte statt Hashing bei häufigeren Delete-Operationen gewählt werden?

- AVL-Baum. Komplexität:

Ausgeglichener Baum, wodurch die Komplexität auf O(logn) für das Suchen begrenzt wird

Da beim Insert nur eine (Doppel-)Rotation notwendig ist => O(logn)

a beim Delete maximal bis zur Wurzel zurück rotiert wird => O(logn)

B-Bäume

Datenstruktur für das gleiche Problem aber größere Datenmenge?

Definition: Binäre Bäume

externe Datenstruktur

Welche Datenstruktur benutzt man zum Verwalten eines Lexikon? Sie soll gut im Suchen und im Einfügen sein.

Welche Listenart sollte man in jedem Fall verwenden?

geordnete Liste

In welcher Zeit kann man Einfügen, Löschen und Suchen im binären Suchbaum realisieren? Best-case, worst-case?

Welche Parameter-Übergabeart würden Sie in einer Procedure zum Suchen in einem Binärbaum wählen?

Erläuterung zu call-by-value und call-by-reference (hier gut aufgrund Speicherplatzersparnis)

Notation eines Binärbaumns in Pascal

Wie fügt man den ersten Knoten ein?

mit ‚p:=new(tRef)'

Was bewirkt dieser Befehl?

Speicher wird allokiert und Adresse des Speichers wird dem Pointer zugewiesen

Wann sind Binärbäume vorzuziehen und wann Hashing?

Vorteile Hashing: Zugriffszeit O(1) im Idealfall d.h. ohne Kollisionen

orteile Bäume: max. Anzahl Elemente muß nicht bekannt sein. Löschen ist leichter. Sortierte Ausgabe ist viel einfacher.

Schreiben Sie eine Prozedur zum Suchen eines Elementes in einem binären Suchbaum. Rekursives Suchen in Bäumen.

Welche Parameterübergabe benutzen Sie hierbei und warum?

Eindruck

Prof. Dr. Schlageter ist ein sehr guter Prüfer. Die Atmosphäre ist locker. Die Benotung ist sehr fair.

Prüfungsprotokoll Praktische Informatik

Datum: 13.06.95

Dauer: 15 Min.

Prüfer: Prof. Dr. Schlageter

Inhalt : nur Kurs 1662 Datenstrukturen (neue Prüfungsordnung)

Prüfungsinhalte

- Was ist ein AVL-Baum ? (Definition)
- Wie ist die Zeitkomplexität bei den 3 Grundoperationen (Einfügen,Suchen,Löschen)
- Wie kann man schneller als in O(n) suchen ? (Binäres suchen)
- Welche Datenstruktur eignet sich zur Darstellung großer Mengen? (B-Bäume)
- Erklären *sie* die B-Bäume !
- Wann verändert sich die Höhe des B-Baumes ?
- Varianten von B-Bäumen (B* - Bäume, Präfix- B - Bäume)
- Welche Datenstruktur für große Mengen gibt es noch ? (Hashing)
- Wann ist Hashing geeignet, wann nicht?
  (- keine große Veränderung der Datenmenge, wahlfreier Zugriff
  - keine sequentielle Verarbeitung)

Sehr lockere und angenehme Prüfungsatmosphäre. Nicht zu kompliziert denken.

Prüfer: **Prof. Schlageter**
Beisitzer: **Hr. Berkel**
Termin: **23.02.95**
Dauer: **15 min**
Note: 1,0
Prüfungsinhalte: Kurs 1663

Prüfungseindruck:

Prof. Schlageter ist ein sehr guter Prüfer! Er versucht dem Prüfling die Nervosität zu nehmen, indem er ihn auffordert zu einem vorgegebenen Thema das anzubringen, was der Prüfling weiß. Es kommen kaum Zwischenfragen, die einen unterbrechen, sondern weiterführende bzw. vertiefende Fragen (siehe Prüfungsverlauf).
Die bevorzugten Themen sind die Kapitel: 4.2, 4.3, 6.2, 6.3, 6.4, 8.1, des Kurses 1663 (in der Ausgabe WS 93'94).
Ein globales Verständnis, insbesondere der Zusammenhänge, ist wesentlich wichtiger als Details der einzelnen Algorithmen zu kennen!

Prüfungsverlauf:

1. Was verstehen sie unter Hashing:
    - Sinn und Zweck des Hashing
    - Eigenschaften der Hashfunktion
    - Kollisionsstrategien
        - lineares Sondieren → führt zur Kettenbildung
        - quadratisches Sondieren (primär ↔ sekundär Kollision)
          Vorteil: Herabsetzung der Wahrscheinlichkeit für Kettenbildung
        - doppel-Hashing → nah am idealem Hashing
          → problematisch: Nachweis der Unabhängigkeit der Funktionen
2. Welche Probleme gibt es beim Hashing?
    - schlechtes worst-case Komplexität O(n)
    - Löschen eines Elements: (Kennzeichnung als deletet / Unterschiedliche Behandlung bei weiteren Inserts und Abfragen)
3. Welche DS sollte statt Hashing bei häufigeren Delete-Operationen gewählt werden?
    - AVL-Baum.

      Komplexitäten
        - ausgeglichener Baum, wodurch die Komplexität auf O(log n) für das Suchen begrenzt wird
        - da beim Insert nur eine (Doppel-)Rotation notwendig ist ⇒ O(log n)
        - da beim Delete maximal bis zur Wurzel zurück rotiert wird ⇒ O(log n)
4. Warum wird i.d.R. Quicksort dem Heapsort vorgezogen?
    - Entartung beim Quicksort ist selten
    - Durchschnittskomplexität in der Analyse mit Faktor ≈ 2 schlechter als Optimum (Quicksort hin gegen nur ≈ 1,38)
5. Warum dieser Unterschied im Faktor?
    - Weil das einsinkende Element i.d.R. bis auf die unteren Ebenen des Heaps durchsinkt.
      ⇒ Ansatz für Bottom-Up-Heapsort

Diplom-Vorprüfung Praktische Informatik

Kurs: 1663 Datcnsuukturen (WS 94/95)
Prüfer: Prof. Dr. Schlageter
Termin: Februar 1995
Dauer: 15 Minuten
Note: 1.0

Was wissen Sie über **Hashing?**

- Grundidee des Hashing (Dictionary-Implementierung, 2 Arten: offenes und geschlossenes Hashing)
- Erläuterung des geschlossenen Hashing mit Kollisionsstrategien
- Definition von Primär- und Sekundärkollisionen
- Praktisches Beispiel für lineares und quadratisches Sondieren (Array zeichnen und andeuten. wo zwei Werte abgespeichert werden)

Andere Datenstruktur für das gleiche Problem'?

⇒ **AVL-Bäume**

- Definition
- Probleme beim Einfügen und Löschen (evtl. Einfach- oder Doppelrotation notig zum Rebalancieren)
- Zeitkomplexitäten für Einfügen, Löschen (spez. Rotationen), Suchen

Datenstruktur für gleiches Problem aber größere Datenmenge'?

⇒ **B-Bäume**

- externe Datenstruktur
- Definition

Warum wird **Quickort** häutig dem **Heapsort** vorgezogen'?

- (geringfügig) bessere Laufzeit im average case
- einfacher zu implementieren
- Split-Element kann im Register Sehalten werden (Kommentar von Prof. Schlageter: "Ach. die Frage kennen Sie auch schon?")

Fazit:

Die Prüfung verlief in einer lockeren, sehr angenehmen Atmosphäre. obwohl Prof. Schlageter wegen einer Verspätung etwas unter Termindruck stand. Vermutlich brauchte ich daher nur Teilgebiete der Themen darzustellen. Wichtiger schienen mir die Zusammenhänge zwischen den einzelnen Gebieten zu sein, wie sich an den Überleitungen erkennen läßt.
Prof. Schlageter ist als Prüfer aufjeden Fall zu empfehlen!

Protokoll zur Vordiplom-Prüfung : Praktische Informatik

Prüfer : Prof. Dr. Schlageter
Datum : 23.11.1994

- Welche Datenstruktur benutzt man zum Verwalten eines Lexikon? (Sie soll gut im Suchen und im Einfuegen sein).

- Welche Listenart sollte man in jedem Fall verwenden?
  (geordnete Liste!)

- In welcher Zeit kann man Einfügen, Löschen und Suchen im binären Suchbaum realisieren?
  (best case, worst case?)

- Was ist ein AVL-Baum?

- Warum sind hier alle drei Operationen in O(logn) Zeit realisierbar?

- Warum ist Hashing (für die obige Aufgabe) nicht geeignet?

- Was ist Hashing?

- Wie kann man Kollisionen vermeiden?

- Was ist der Vorteil beim quadratischen Sondieren?

- Wann sollte man kein Hashing benutzen?
  (Hashtabelle mehr als 80% gefüllt.
  Geschlossenes Hashing ist für dynamische Update- (insbesondere Lösch-) Operationen nicht geeignet.)

- Welche Sortierverfahren kennen sie?

- Quicksort und Heapsort erläutern

- Warum benutzt man trotz des worst case von Quick Sort lieber dieses Verfahren anstatt Heapsort?
  (Sortiert im Durchschnitt schneller.
  Man kann den Schlüsselwert in ein Register speichern.)

- Welche Parameter-übergabeart wurden sie in einer Procedure zum Suchen in einem Binärbaum wählen

- Erläuterung zu call-by-value und call-by-reference
  (call-by-reference kann man hier auch gut wählen, aufgrund der Speicherplatzersparnis)

Diese Prüfung hat mir unheimlich viel Spaß gemacht. Herr Prof. Dr. Schlageter und sein Beisitzer waren ausgesprochen nett. Ich kann auch nur sagen angenehmste Prüfungs-atmosphäre, klare und verständliche Fragen. Als Prüfer unbedingt zu empfehlen.

Vordiplom Informatik
Praktische Informatik **August 1994**

Kurse 1611 und 1663

bei Prof. Dr. Schlageter

Dauer: ca. 15 Min.

- ❑ Ein Unternehmen will für seine Kundendatei Hashing-Verfahren einführen. Wie würden Sie das Unternehmen beraten?
  (Fragen nach Dateigrößen, Updatewahrscheinlichkeiten, Sollen die Daten auch sortiert ausgegeben werden etc.)

- ❑ Angenommen, wir hätten uns für das Hash-Verfahren entschieden. Was ist zu tun?
  (Dynamisches oder geschlossenes Hashing wählen, Notwendige Tabellengröße ermitteln, Hashfunktion finden, . . [jeweils mit Entscheidungsgründen])

- ❑ Wie können **Sekundär-Kollisionen** beim Hashing behandelt werden?
  (Lineares Sondieren und Doppelhashing mit unabhängigen Funktionen benannt), Nachfragen: Was sagen Sie zu quadratischem Sondieren? Wo bringt quadratisches Sondieren eine Verbesserung? Was passiert eigentlich beim Löschen?

- ❑ Wie beurteilen Sie die **Qualität von Algorithmen?**
  (Korrektheit, Schnelligkeit, Speicherbedarf)

- ❑ **Wie würden Sie Heapsort und Quicksoft unter diesen Qualitätsgesichtspunkten vergleichen?**
  (gleicher Speicherbedarf, im Durchschnitt ist Standard-Quicksott etwas schneller, aber im worst case mit $O(n^2)$ gegenüber $O(n \log n)$ unterlegen.)

- ❑ Wie kann man worst case bei Quicksort vermeiden?
  (vermeiden nicht, aber unwahrscheinlich bei Clever-Quicksort)

- ❑ Warum ist Heapsort im Durchschnitt eigentlich langsamer?
  (zwei Vergleiche in der Schleife, dadurch Komplexität ($2 n \log n$), Quicksort benötigt nur einen Vergleich)

Das hat ihm gereicht.

Alles in allem eine sehr angenehme und lockere Prüfung. Auch die Note war angenehm. Herr Schlageter bleibt weiterhin als Prüfer empfehlenswert.

Viel Glück!

Datenstrukturen

Zusammenfassung bezogen auf alten Prüfungsfragen *(Prof. Schlageter)*

Wie beurteilen Sie die Qualität von Algorithmen?

Ein Algorithmus ist ein Verfahren zur **Lösung eines Problems**. Er ist eine Beschreibung eines Problems, wie etwas berechnet werden soll und **ist unabhängig von einer Programmiersprache**. Ein Algorithmus wird nach folgenden Punkten **qualitativ beurteilt**:

1. Korrektheit
2. einfach zu verstehen
3. einfache Implementierung
4. Laufzeit und Platzbedarf

Wie ist die O-Notation definiert?

Seien f, g: $N \rightarrow R^+$

$f=O(g) :\Leftrightarrow \exists\, n_0 \in N, c \in R, c > 0 : \forall\, n \geq n_0 : f(n) \leq c*g(n)$

f wächst also höchstens so schnell wie g.

Existiert (f(n)/g(n)) für $n \rightarrow$ unendlich, dann gilt f=O(g).

Was heißt, ein Algorithmus ist optimal?

Ein Algorithmus heißt *(asymptotisch)* optimal, wenn die obere Schranke für seine Laufzeit mit der unteren Schranke für die Komplexität zusammenfällt. Ein allgemeiner Sortieralgorithmus mit Laufzeit O(n*Log(n)) ist zum Beispiel optimal.

Was wissen Sie über Hashing?

Hashing *("Zerhacken")* ist ein **Dictonary**, also ein Datentyp mit den Operationen **INSERT, DELETE und MEMBER**. Eventuell auch mit den Operationen EMPTY und ISEMPTY.

Hashing ist ein **Kompromiß zwischen Zeit- und Platzbedarf**. Die Grundidee des Hashing ist folgende:

Aus dem Wert eines zu speichernden Mengenelements wird seine Adresse im Speicher berechnet. Den Speicher zur Aufnahme der Mengenelemente faßt man als Buckets *("Behälter")* auf, die etwa mit $B_0, B_1, ..., B_{m-1}$ numeriert werden. Der Wertebereich, aus dem die Mengenelemente stammen, kann beliebig groß sein. In der Regel gilt also |D| >> m.

Zur Berechnung der Adresse der Schlüssel verwendet man die sogenannte **Hash-Funktion**

$h: D \rightarrow \{0,1,...,m-1\}$

Die **Hash-Funktion** sollte folgende Eigenschaften besitzen:

1. Sie sollte **surjektiv** sein, also alle Buckets erfassen
2. Sie sollte die zu speichernden Schlüssel **gleichmäßig** über alle Buckets verteilen
3. Sie sollte **effizient zu berechnen** sein.

Aufgrund des Kompromisses kann es beim Hashing zu sogenannten **Kollisionen** kommen, daß heißt mehrere Schlüssel bzw. Mengenelemente können auf denselben Mengenbehälter abgebildet werden.

Die **Kollisionswahrscheinlichkeit** um n Schlüsselwerte auf m Buckets (n<m) ist:

$P_{Kollision}=1-(m*(m-1)*(m-2)*...*(m-n+1))/(m^n)$

In der Praxis sind Kollisionen nahezu unvermeidbar *("Geburtstagsparadoxon")*.

Deshalb verwendet man verschiedene Arten von Hashing, namentlich das **offene Hashing** und das **geschlossene Hashing**.

Offenes Hashing:

Beim offenen *("separate Claining")* Hashing kann ein Bucket beliebig viele Mengenelemente aufnehmen. Dies wird zum Beispiel durch verkettete Listen realisiert. Kollisionen stellen dann kein Problem mehr da. Die durchschnittliche Listenlänge ist dann n/m. Unter der Annahme, daß n gleich m ist und die Hashfunktion tatsächlich die Mengenelemente gleichmäßig auf die Buckets verteilt, ist der erwartete Zeitaufwand für Suchen, Einfügen und Entfernen jeweils O(1+n/m)=O(1).
Der **worst case** ist bei allen Hashverfahren sehr schlecht nämlich **O(n)**.
Der Platzbedarf ist folglich im worst case **O(n+m)**.

Geschlossenens Hashing:

Beim geschlossenen Hashing ist die Zahl der Einträge n durch die Kapazität der Tabelle m*b begrenzt. Beim geschlossenen Hashing haben Methoden zur **Kollisionsbeseitigung** entscheidende Bedeutung. Die allgemeine Methode zur Kollisionsbeseitigung, daß sogenannte **rehashing** *("offene Adressierung")*, beruht auf folgende Idee:

1. Neben h_0 gibt es weitere Hashfunktionen h_1, h_2, ...,h_{m-1}, welche für einen gegebenen Schlüssel x die Buckets $h_0(x)$, $h_1(x)$, ..., $h_{m-1}(x)$ inspizieren.
2. Sobald eine erste freie, oder als gelöscht markierte Stelle gefunden wurde, wird das Mengenelement dort eingetragen.
3. Beim Suchen wird die gleiche Folge von Buckets betrachtet, bis entweder x gefunden wurde, oder die erste freie Stelle signalisiert, daß x nicht vorhanden ist. <u>Aus diesem Grund können Mengenelemente nicht einfach gelöscht</u> werden, da sonst das Suchen fehlschlägt. Vielmehr muß eine Zelle als gelöscht markiert werden.

Geschlossenes Hashing eignet sich aus diesem Grund nicht sehr für dynamische Anwendungen mit vielen Lösch- und Einfügeoperationen. Ist die Hashtabelle zu 80 Prozent belegt, sollte **reorganisiert** werden.
Man spricht von **idealen *(oder uniformes)* geschlossenen Hashing**, wenn Kettenbildung vermieden wird und alle möglichen Konfigurationen von n besetzten und m-n freien Zellen mit gleicher Wahrscheinlichkeit auftreten.
Folgende **Kollisionsstrategien** gibt es:

- **Lineares Sondieren:**

  $h_i(x)=(h(x)+c*i) \bmod m; \qquad 1 \le i \le m-1$

  c Konstante, c und m sollten teilerfremd sein, damit alle Zellen getroffen werden. Es entstehen Ketten mit Abstand c.

- **Quadratisches Sondieren:**

  $h_i(x)=(h(x)+i^2) \bmod m$

  beziehungsweise verfeinert:

  $h_{2i-1}(x)=(h(x)+i^2) \bmod m \qquad 1 \le i \le ½(m-1)$

  $h_{2i}(x)=(h(x)-i^2) \bmod m$

  mit m=4*j+3 Primzahl. Dann wird, nach einen Ergebnis aus der Zahlentheorie jede der m Zellen getroffen.
  Quadratisches Sondieren ergibt keine Verbesserung bei **Primärkollisionen**, also $h_0(x)=h_0(y)$, jedoch vermeidet es Clusterbildung *("Anhäufungen, Kettenbildungen")* bei

Sekundärkollisionen, also $h_0(x)=h_k(y)$, $k>0$.
Also: Vorteil bei Sekundärkollisionen: keine Clusterbildung.

- **Doppel-Hashing**
  Man wählt zwei Hashfunktionen h und h', <u>die voneinander unabhängig</u> sind.
  <u>Annahme:</u> **P(h(x)=h(y))=1/m=P(h'(x)=h'(y))**, dann gilt, wenn h und h' **unabhängig** sind für die Wahrscheinlichkeit **P(h(x)=h(y) ∩ (h'(x)=h'(y))=1/m²**
  Die **Hashfunktion** ist dann:
  $$h_i(x)=(h(x)+h'(x)*i^2) \bmod m$$
  Unabhängige Funktionen sind <u>jedoch schwer zu finden</u>.

Wahl der Hashfunktionsbasis

- **Divisionsmethode**
  Einfachste denkbare Hashfunktion. Seien die natürlichen Zahlen der Schlüsselbereich, dann wählt man h(k) = k mod m. M ist hierbei die maximale Anzahl von Einträgen.
  <u>Nachteil</u>: Aufeinanderfolgende Zahlen werden aufeinanderfolgend abgebildet.
- **Mittel-Quadrat-Methode**
  k sei die Ziffernfolge $(k_i)_{i\in\{1,\dots,r\}}$ Man bilde $k^2=s_{2r}s_{2r-1}\dots s_1$ und entnehme einen Block von mittleren Ziffern als Adresse h(k). Da die mittleren Ziffern von <u>allen</u> Ziffern in k abhängen, werden aufeinanderfolgende Werte besser gestreut.

<u>Zusammenfassung:</u>

1. worst case: **O(n)** bei allen drei Dictonary-Funktionen
2. gutes Durchschnittsverhalten (average case): **O(1)**
3. bei dynamischen Anwendungen offenes Hashing, da Entfernen und Hinzufügen von Elementen problemlos ist.
4. <u>geschlossenes</u> Hashing für Anwendungen mit feststehender Maximalzahl von Elementen, bei denen wenig gelöscht wird. Hier sollte die <u>Auslastung 80% nicht überschreiten.</u>
5. Nachteil bei allen Hashverfahren ist, daß die Schlüssel nicht effektiv sortiert ausgegeben werden können.

Kennen Sie noch eine andere Datenstruktur (also: andere Dictonaries) für das gleiche Problem?

Zum Beispiel der **binäre Suchbaum**:
Zuerst:
<u>Definition:</u> Binärer Baum:

1. Der leere Baum ist ein binärer Baum
2. Wenn x ein Knoten ist und T_1 und T_2 binäre Bäume sind, dann ist auch das Tripel (T_1,x,T_2) ein binärer Baum.
   - x heißt Wurzel
   - T_1 heißt linker Teilbaum von T
   - T_2 heißt rechter Teilbaum von T
   - Die Wurzeln von T_1 und T_2 heißen Söhne von x. x ist ihr Vaterknoten.
   - Ein Knoten, der keine Nachfolger (Söhne) hat, heißt Blatt

- Die **maximale** Höhe (Nur Wurzel: Höhe 0) eines Binärbaums mit n Knoten ist folglich n-1. Die **minimale** Höhe eines Binärbaums mit n Knoten ist exakt $[\mathrm{Log}_2(n+1)-1]$, bzw. O(Log(n))
- Ein **vollständig binärer Baum** ist ein binärer Baum, indem alle Ebenen bis auf

die letzte vollständig besetzt sind.

- Ein Baum ist **vollständig ausgeglichen**, wenn sich für jeden Knoten die Anzahl der Knoten in seinen linken und rechten Teilbaum um höchstens 1 unterscheiden.

Baumdurchläufe:

inorder(l,x,r):=inorder(l)°<x>°inorder(r)
preorder(l,x,r):=<x>°preorder(l)°inorder(r)
postorder(lx,r):=postorder(l)°postorder(r)°<x>

Jetzt noch Definition für Knotenmarkierung:

Definition: Knotenmarkierung

Sei T ein Baum. Man bezeichnet die Knotenmenge von T ebenfalls mit T. Eine **Knotenmarkierung** ist eine Abbildung

$$\mu : T \rightarrow D$$

mit irgendeinen geordneten Wertebereich:

Nun kann der binäre Suchbaum definiert werden:

Definition: binärer Suchbaum

Ein knotenmarkierter binärer Baum T heißt genau dann **binärer Suchbaum**, wenn für jeden Teilbaum T' von T mit $T'=(T_l,y,T_r)$ gilt:

$$\forall\ x \in T_l : \mu(x) < \mu(y)$$
$$\forall\ z \in T_r : \mu(z) > \mu(y)$$

Vereinfacht ausgedrückt. Alle Schlüssel im linken Teilbaum sind kleiner als der Schlüssel in der Wurzel, bzw. alle Schlüssel im rechten Teilbaum sind größer, als der Schlüssel in der Wurzel.

Ein **inorder** Durchlauf liefert die eingetragenen Elemente in **sortierter** Reihenfolge.

Kosten für INSERT, DELETE, MEMBER:

Der Aufwand für alle drei Funktionen ist proportional zur Pfadlänge. Die maximale Pfadlänge in einem zur Liste **degenerierten** binären Suchbaum ist **O(n)**. In einem **balancierten** binären Baum ist die maximale Pfadlänge **O(Log(n)).**

Worst Case:

Der Aufwand für den **Aufbau** ist demnach:

$$1+2+3+...+n = (n(n+1))/2 = \mathbf{O(n^2)}$$

Alle drei Operationen *(INSERT, DELETE, MEMBER)* benötigen **O(n)** Zeit.

Average Case:

Der durchschnittliche Aufwand für alle drei Operationen ist **O(Log(n)**. *(Genau: 2 * Ln(2) *Log (n)+ const). Also etwa 38 Prozent über Log(n)* Das verhältnismäßig gute Durchschnittsverhalten gilt allerdings nicht mehr, wenn sehr lange, gemischte Folgen von Einfüge und Löschoperationen betrachtet werden. Der Baum wird dann mit der Zeit **linkslastig**. Dies liegt daran, daß beim Löschen jeweils der Nachfolgeschlüssel die freie Position einnimmt. Man behebt dieses Problem, indem man die Löschprozedur so implementiert, daß zufällig ausgewählt wird, ob der Nachfolger oder der Vorgänger die Position des zu löschenden Schlüssels einnimmt.

Kennen Sie noch eine andere Datenstruktur, die für größere Datenmengen in Frage kommt?

Der AVL-Baum bzw. der B-Baum.

Definition: AVL-Baum

Ein **AVL-Baum ist ein binärer Suchbaum**, in dem sich für jeden Knoten die

Höhe seiner zwei Teilbäume um höchstens 1 unterscheidet.

Folglich ist ein AVL-Baum (nach N.Wirth) ausgeglichen. Im Skript wird er balancierter Suchbaum genannt. balancierter Baum = ausgeglichener Baum? Ausgeglichen paßt eigentlich besser als balanciert, da jeder vollständig ausgeglichener Baum auch ausgeglichen ist. Die Umkehrung gilt i.A. nicht.

- Die Grundidee des AVL-Baums ist, die sogenannte **Strukturinvariante** einzuführen. Die Strukturinvariante ist eine Abschwächung des Balanciertheitkriteriums vollständig balancierter *Bäume (also vollständig ausgeglichener Bäume).*
- Der AVL-Baum ist ein balancierter *(=ausgeglichener)* binärer Suchbaum
- Garantiert folglich auch im **worst case** eine Laufzeit von **O(log(n))** für alle drei Dictonary Funktionen. Als Speicherplatz wird **O(n)** garantiert.
  Bei einem Update, also Einfüge *(INSERT)* und/oder Lösch *(DELETE)* Operationen, muß die **Strukturinvariante** überprüft werden. Der AVL-Baum muß also **Rebalanciert** werden, wenn die **Strukturinvariante** verletzt wurde:

  Einfügen: Fügt ein neues Blatt hinzu. Dieses Blatt gehört zu allen Teilbäumen von Knoten, die auf den Pfad von der Wurzel zu diesem Blatt liegen. Die Höhen dieser Teilbäume können um 1 wachsen.

  Löschen: Vernichtet einen Knoten *(Blatt oder inneren).* Teilbäume auf dem Pfad von der Wurzel zum Vaterknoten sind betroffen. Ihre Höhe kann sich um 1 ändern.

  Balancierphase: Man läuft von der aktuellen Position aus den Pfad bis zur Wurzel zurück und überprüft die Balance. Dann führt man eventuell Balancieroperationen durch:

  Rebalancieren:

  - Äußerer Teilbaum größer als der innere: **Einfach Rotation**
  - Innerer Teilbaum größer als der äußere: **Doppel-Rotation**

  Das Rebalancieren wird wie folgt implementiert: Man speichert für jeden Knoten des Suchbaums zusätzlich die Balance, bzw. Höhe ein. Also: balance(p):=height(p.left)-height(p.right) ein. Balance hat dann bei einem balancierten *(ausgeglichenen)* Baum die Werte {-1,0,+1} und bei einem nicht balancierten dementsprechend auch -2 und +2.
  Ein AVL-Baum ist höchstens 44 Prozent höher, als ein vollständig ausgeglichener Baum.
  Da ein AVL-Baum auch ein binärer Suchbaum ist, liefert ein **inorder** Durchlauf auch hier die Elemente in sortierter Reihenfolge.

Warum sind im AVL-Baum alle drei Dictonary Operationen in O(Log(n)) Zeit realisierbar?

Da der AVL-Baum, im Gegensatz zum binären Suchbaum ausgeglichen *(=balanciert)* ist, beträgt die obere Schranke für die Höhe exakt **1,4404 Log(n)+const**. Da die Diconary-Operationen von der Höhe abhängen, werden folglich alle Operationen in **O(Log(n))** Zeit realisiert.

Was ist der Vorteil beim quadratischen Sondieren?

Der Vorteil beim quadratischen Sondieren ist, daß bei einer Sekundärkollision eine Clusterbildung vermieden wird.

Was ist ein externer Algorithmus?

Man spricht von einem externen Algorithmus, wenn zur Verarbeitung einer Objektmenge der **Größe n** lediglich **O(1)** interner Speicherplatz zur Verfügung steht, jedoch **Ω (n)** externer Speicherplatz. Ein typischer externer Speicher ist etwa die Magnetplatte *(Festplatte).*
Beim Entwurf von externen Algorithmen muß man Ihre Zugriffscharakteristika beachten. Da zum Beispiel bei der Magnetplatte ein Großteil der Zugriffszeit auf Positionieren des

Schreib/Lesekopfs entfällt, liest man als Konsequenz die Daten in Blöcken aus *(Speicherseiten, Cluster)*. Typische Clustergrößen moderner Festplatte liegen bei 512 Bytes bis 8192 Bytes. Aus diesem Grund betrachtet man als **Kostenmaß** für die **Laufzeit eines externen Algorithmus** meist die **Anzahl der Seitenzugriffe**.
Da der externer Speicherplatz in Form von Seiten zur Verfügung gestellt wird mißt man den **Platzbedarf einer Speicherstruktur als Anzahl der belegten Seiten**. Diese sind aus organisatorischen Gründen i.A. nicht vollständig belegt. Aus diesem Grund ist ein weiteres interessantes Maß die **Speicherplatzausnutzung**.

"Anzahl benutzer Bytes" / ("Anzahl Seiten" * "Anzahl Bytes pro Seite")
Angabe gewöhnlich in Prozent

Was ist ein B-Baum?

Der B-Baum ist ein Dictonary-Datentyp der sich insbesondere für große Datenmengen eignet, die u.U. extern gespeichert werden.
Ein B-Baum ist ein spezieller Vielweg-Baum.

Definition: Vielweg-Suchbaum

1. Der leere Baum ist ein Vielweg-Suchbaum mit Schlüsselmenge $\emptyset$
2. Seien $T_0, ..., T_S$ Vielweg-Suchbäume mit Schlüsselmengen $T'_0, ..., T'_S$ und seien $k_1, ..., k_S$ eine Folge von Schlüsseln, so daß gilt:

$$k_1 < k_2 < ... < k_S$$

Dann ist die Folge

$$T_0 k_1 T_1 k_2 T_2 k_2 .....k_S T_S$$

ein Vielweg-Suchbaum genau dann, wenn gilt:

$$\forall\ x \in T'_i : k_i < x < k_{i+1} \text{ für } i=1,...,s-1$$
$$\forall\ x \in T'_0 : x < k_1$$
$$\forall\ x \in T'_S : k_S < x$$

Für die Schlüsselmenge gilt dann: $\{ k_1, ..., k_S\} \cup \cup T'_i$

Definition: B-Baum:

Ein B-Baum der Ordnung m ist ein Vielweg-Suchbaum mit folgenden Eigenschaften:

1. Die Anzahl der Schlüssel in jedem Knoten mit Ausnahme der Wurzel liegen zwischen m und 2m. Die Wurzel enthält ebenfalls maximal 2m Schlüssel. Wenn die Wurzel kein Blatt ist, enthält sie mindestens einen Schlüssel.
2. Alle Pfadlängen von der Wurzel zu einem Blatt sind gleich.
3. Jeder innerer Knoten mit s Schlüsseln hat genau s+1 Söhne. Es gibt also keine leeren Teilbäume.

Die Kosten für eine Einfüge- oder Löschoperation sind offensichtlich proportional zur Höhe des B-Baumes. Der B-Baum der Ordnung **m** unterstützt alle drei Dictonary-Operationen in **O(Log$_{m+1}$(n)) Zeit**. Der **Platzbedarf** ist **O(n)**. Mit Ausnahme der Wurzel ist die **Speicherplatzausnutzung** besser als **50 Prozent,** da die Seiten immer mindestens bis zur Hälfte gefüllt sind.
Der Suchalgorithmus eines B-Baumes ist offensichtlich. Innerhalb des Knotens könnte binär gesucht werden.
Beim B-Baum kann die Struktur unter **Updateoperationen** einfach aufrechterhalten werden. Beim Einfügen können temporär Knoten mit **2m+1 Schlüsseln** entstehen *(Overflow)*, beim Entfernen entsprechend Knoten mit **m-1 Schlüssel** *(Underflow)*.

<u>Overflow</u>: Ein Overflow eines Knotens wird mit einer Operation split(p) behandelt. Der Knoten **p** mit **2m+1** Schlüsseln wird am mittleren Schlüssel k_{m+1} geteilt. Es entstehen also zwei Knoten mit $k_1...k_m$ und $k_{m+2}...k_{2m+1}$. Jeder dieser Knoten hat trivialerweise **m** Knoten. K_{m+1} wandert nach oben, entweder in den

Vaterknoten oder in einen neuen Wurzelknoten. Eventuell entsteht dann auch beim Vaterknoten ein Overflow. Die Behandlung des Overflows kann sich also von einem Blatt bis zur Wurzel des Baumes fortpflanzen.

Underflow: Ein Nachbar eines Knotens sei ein direkt benachbarter Bruder. Um einen Underflow in p zu behandeln, werden der oder die Nachbarn von p betrachtet. Wenn einer der Nachbarn genügend Schlüssel hat, wird seine Schlüsselfolge mit der von p ausgeglichen *(Operation balance)*, so daß beide etwa gleichviele Schlüssel haben. Andernfalls wird p mit dem Nachbarn zu einem einzigen Knoten verschmolzen *(Operation merge)*.

Eigenschaften eines B-Baumes mit Ordnung n nach N.Wirth:

1. *Jede Seite (entspricht Wurzel, Knoten) enthält höchstens 2n Elemente (Schlüssel).*
2. *Jede Seite, außer der Wurzelseite, enthält mindestens n Elemente.*
3. *Jede Seite enthält entweder eine Blattseite, d.h. hat keine Nachfolger, oder sie hat m+1 Nachfolger, wobei m die Zahl Ihrer Schlüssel ist.*
4. *Alle Blattseiten liegen auf der gleichen Stufe*

Wann verwendet man einen B-Baum?

Beim Problem des externen Suchens

Welchen Datenstruktur benutzt man zum Verwalten eines Lexikons?

Hier kommen Dictonaries *(z.B. Sequentiell geordnete Liste im Array, ungeordnete Liste, Geordnete Liste, Bitvektorfeld, Hashing, Binäre Suchbäume)* in Frage. Man sollte aber noch genau abwägen, welche Datenstruktur man letztendlich verwendet.

Wann sind Binär-Baume vorzuziehen, wann Hashing-Verfahren?

Nachteil beim geschlossenen und offenen Hashingverfahren ist, daß nicht effektiv sortiert ausgegeben werden kann. Zwar hat Hashing ein gutes durchschnittliches Leistungsverhalten von O(1), jedoch ist der worst case mit O(n) inakzeptable. Ebenso eignet sich bei dynamischen Anwendungen *(viel Update-Operationen)* in der Regel nur das offene Hashing. Beim geschlossenen Hashing gibt es Probleme beim Löschen von Elementen, da der Speicherplatz nur als gelöscht markiert wird.

Bei Binär-Bäumen muß nun zwischen ausgeglichenen *(balancierten)*, vollständig ausgeglichen und Binär-Suchbäumen unterschieden werden.

Welche Zugriffszeit hat ein AVL-Baum?

Da der AVL-Baum ein ausgeglichener *(balancierter)* Baum ist, und die Dictonary-Operationen proportional zur Höhe des AVL-Baums sind, ist, auch im worst case, die Laufzeit **O(Log(n))**.

Klassifizieren Sie Sortiertalgorithmen?

Intern/Extern: Man spricht von einem **internen** Verfahren, wenn alle Records gleichzeitig im Hauptspeicher gehalten werden können. Ein **externes** Verfahren lädt jeweils nur eine Teilmenge der Datensätze in den Hauptspeicher.

methodisch:

- Sortieren durch Einfügen (z.B. Insertion-Sort)
- Sortieren durch Auswählen (z.B. Selection-Sort)
- Divide and Conquer (z.B. Quicksort)
- Fachverteilen (z.B. Bucketsort)

nach Effizienz: Einfache Verfahren haben einen Laufzeit von $O(n^2)$, z.B. Selection-Sort, gute erreichen O(n*Log(n)). Bei manchen Methoden unterscheidet sich Worst Case und Average Case ganz erheblich.

Im Array oder nicht: Im Array ist einfach zu implementieren, man braucht hier keinen

zusätzlichen Platz für Zeiger. **IN SITU** heißt hier, Vertauschen innerhalb eines einzigen Arrays.

allgemeine/eingeschränkte Verfahren: Manche Methoden *(z.B. Sortieren durch Fachverteilen)* lassen sich nur auf eine Folge von Records mit speziellen Eigenschaften anwenden.

Wesentliche Operationen bei einem Sortierverfahren:

- Vergleich zweier Schlüssel
- Vertauschen von Datensätzen

Ein Sortierverfahren heißt **stabil**, wenn sichergestellt ist, daß die Reihenfolge von Sätzen mit gleichem Schlüsselwert nicht verändert wird. Einfache Methode sind in der Regel stabil. Gute Methode sind i.d.R. nicht stabil. Stabilität ist zum Beispiel dann gewünscht, wenn etwa bereits alle Elemente nach einen zweitrangigen Schlüssel sortiert vorliegen.

Gibt es untere Schranken für (allgemeine) Sortiertverfahren?

Ω (n) ist offensichtlich eine untere Schranke für Sortiertalgorithmen, da man sicherlich jedes Element mindestens einmal betrachten muß. Allgemeine Verfahren, daß man nur eine Möglichkeit hat, auf Schlüsselwerte zuzugreifen, nämlich einen Vergleichsoperator auf sie anzuwenden. Jeder solche Algorithmus benötigt Ω (n*Log(n)) Vergleiche im worst case.

Stellen Sie Heapsort und Quicksort gegenüber?

Quicksort benutzt das **Divide and Conquer** Paradigma.

IF die objektmenge ist klein genug
THEN löse das problem direkt
ELSE
Divide: Zerlege die Menge in mehrere Teilmengen, wenn möglich, gleicher Größe
Conquer: Löse das Problem rekursiv für jede Teilmenge
Merge: Berechne aus dem für die Teilmenge erhaltenen Lösungen eine Lösung des Gesamtproblems.
FI

Der Algorithmus von Quick-Sort ist der folgende:

ALGORITHM QuickSort(S)
IF |S|=1 **THEN RETURN** S
ELSE IF alle Schlüssel sind gleich THEN RETURN S
ELSE
Divide: Wähle $x=s_j$.key so, daß x nicht minimal in S ist. Berechne eine Teilfolge S_1 aus S mit den Elementen, deren Schlüsselwert kleiner als x ist und eine Teilfolge S_2 mit Elementen ≥ x.
Conquer: S'_1:= QuickSort(S_1); S'_2:=Quicksort(S_2)
Merge: **RETURN** Concat(S'_1,S'_2)
FI;
FI

Im **Worst Case** wird die Folge der **Länge n** in eine Folge der **Länge 1** und einer der **Länge n-1** zerlegt. In diesem Fall wird der Quicksort zum Slowsort. Die Laufzeit beträgt aufgrund des entarteten Aufrufbaums **Ω (n^2)** und zwar für jeden Quicksort Implementierung.

Der **average case** beträgt **O(n*Log(n))**. Das Problem liegt an der Wahl eines geeigneten x-Wertes *(Problem der Median-Suche).*

Um das Problem zur Suche eines geeigneten x-Wertes befriedigend zu lösen, verwendet

man den **Clever-Quicksort**. Dabei wird der mittlere Wert einer Folge von drei zufällig gewählten Elementen als Split-Element benutzt. Im Gegensatz zum normalen **Quicksort**, der etwa **38,6 Prozent** über den optimalen Wert liegt, liegt **Clever-Quicksort nur 18 Prozent über dem optimalen Wert**.
Ohne Modifikationen hat der **Quicksort** für den **Rekursionsstack** einen **Platzbedarf** von **O(n)**. Man kann ihn jedoch so modifizieren, daß dieser zusätzliche Platz **O(Log(n))** beträgt.

Heapsort:
Heapsort benutzt einen partiell geordneten Baum *(Heap)*, um das Minimum in O(Log(n)) Zeit zu finden und zu entnehmen.

Definition: partiell geordneter Baum

Ein **partiell geordneter Baum *(Heap)* ist ein knotenmarkierter binärer Baum T**, in dem für jeden Teilbaum T' mit Wurzel x gilt:

$$\forall\, y \in T' : \mu(x) \leq \mu(y)$$

Das heißt also, daß in der Wurzel das (globale) Minimum ist.

Standard Heapsort:

Wegen der Effizienten Implementierbarkeit eines partiell geordneten Baumes im Array ist der Heapsort auch in der Praxis interessant. Er ist das beste bekannte Sortierverfahren, welches **O(n*Log(n))** im **worst case** garantiert.

Grundidee:

1. Die n zu sortierenden Elemente werden in einem Heap eingefügt. Die **Komplexität** dafür ist **O(n*Log(n))**
2. Es wird anschließend n-mal das Minimum aus dem Heap entnommen. **Komplexität O(n*Log(n))**

Implementierung:

Um den Aufbau des Heaps zu beschleunigen und in situ sortieren zu können, wird die sortierte Folge rückwärts am hinteren Ende des Arrays, welches den Heap enthält, eingetragen.

Analyse:

In einem vollständig ausgeglichenen binären Baum liegen 50% der Knoten auf der untersten Ebene, 25% auf der vorletzten usw. Wenn ein einsinkendes Element auf allen Positionen mit gleicher Wahrscheinlichkeit landet, dann durchläuft es im Durchschnitt fast immer einen kompletten Pfad bis zu einem Blatt. Beim Absinken werden zwei Vergleiche benötigt. Daher liegt die durchschnittliche Anzahl der Vergleiche nahe bei

2*n*Log(n) + O(n)

Dies ist der Grund für das schlechte Durchschnittsverhalten von Heapsort im Vergleich zu Quicksort, der ja im Durchschnitt nur

1,386*n*Log(n)+O(n)

Vergleiche benötigt.
Mit den zwei Vergleichen auf jeder Ebene werden zwei Entscheidungen getroffen:

1. Welcher Pfad ist im Heap zu verfolgen?
2. Muß das bearbeitete Element noch tiefer einsinken?

Botton-Up-Heapsort:

Die zweite Entscheidung fällt fast immer positiv aus. Aus diesem Grund kam man auf die Idee des Botton-Up *("Von unten nach oben")* Heapsort. Hier werden die beiden Entscheidungen getrennt voneinander betrachtet.

1. Durchlaufe den Heap top-down, *("Von der Spitze nach unten")* um

den Pfad zu bestimmen, auf dem das einzusinkende Element e irgendwo liegen muß. Dazu sind in jedem Knoten lediglich die beiden Söhne zu vergleichen und der kleinere auszuwählen. Dieser Pfad heißt Zielpfad. Er endet in einem Blatt b.

2. Durchlaufe den Zielpfad von b aus bottom-up und vergleiche jeden Schlüssel mit e. Abbruch, sobald ein Schlüssel q<e gefunden wird.
3. Entnimm das Wurzelelement e und schiebe alle Elemente auf dem Zielpfad bis zu q um eine Position nach oben. Füge e auf der alten Position q ein.

Der **Vorteil** gegenüber dem Standard-Heapsort besteht darin, daß die Schleife im Schritt 2 meist nach einem oder zwei Durchläufen abbricht. Die Anzahl der Vergleich für Botton-Up Heapsort liegt deshalb viel näher an

1*n*Log(n)+O(n) also an
2*n*Log(n)+O(n).

Durch Experimente wurden die Analysen bestätigt, daß

- Bottom-Up Heapsort für **n≥ 400** besser als Quicksort
- Bottom-Up Heapsort für **n≥ 16000** besser als Clever-Quicksort

Warum entartet der Heapsort nicht?

Weil der Heapsort einen partiell geordneten Baum verwendet. Bei diesem Baum betragen alle drei Dictonary-Operationen O(Log(n)).

Wann entartet der Quicksort und wie kann das vermieden werden?

Der Quicksort entartet, wenn die Folge der Länge n in eine Folge der Länge 1 und n-1 zerlegt wird. Die passiert, wenn der Median *(x-Element, mit dem verglichen wird)* ungeschickt gewählt wird.
Man kann dies vermeiden *(aber nicht ganz ausschließen)*, wenn man entweder zufällig ein Element wählt oder den Clever-Quicksort einsetzt.

Wieviel Divide-Schritte werden beim Quicksort durchgeführt?

O(Log(n)) bis O(n).

Warum ist Heapsort im Durchschnitt eigentlich langsamer als der Quicksort?

siehe Analyse Heapsort.

Warum wird der Quicksort dem Heapsort trotzdem vorgezogen?

Antwort aus alten Prüfungsprotokollen: Da bei der Implementierung das Trennelement in einem schnellen Register gehalten werden kann. Also allgemein gesprochen: die Operationen pro Verarbeitungsschritt sind i.A. nicht so komplex, als beim Heapsort. In der Praxis ist, bei Verwendung von speziellen Quicksort-Version *(z.B. Clever-Quicksort)* ist eine Entartung *(→ worst case)* relativ selten.

Welche Probleme können beim Hashing auftreten?

Die Hashfunktion ist nicht injektiv
Kollisionsstrategien
Probleme beim Löschen von Elementen
Schechtes worst case Verhalten beim offenen Hashing

Was ist primär Kollision und sekundär Kollision?

Primär Kollision allgemein ist, wenn die durch die Hashfunktion berechnete

Speicherzelle bereits durch ein anderes Element besetzt ist. Also:$h_0(x)=h_0(y)$.
Von Sekundär Kollision spricht man, wenn $h_0(x)=h_k(y)$, mit $k>0$ gilt.

☠ ----------Ab hier eigene Fragen:

Wie kann man extern Sortieren?

Zum Beispiel mit Mergesort.

<u>Mergesort:</u>

- Es stehen Dateien f_1,f_2,g_1 und g_2 zur Verfügung.
- Zu Beginn stehen alle n Datensätze in g_1
- f_1,f_2,g_2 sind jeweils leer

<u>Definition Lauf:</u>

Ein aufsteigend geordnete Teilfolge von Sätzen innerhalb einer Datei wird als Lauf bezeichnet.

Zu Anfang liest eine Prozedur *InitialRuns* die Datei g_1 und schreibt abwechselnd Läufe der Länge 1 nach f_1 und f_2. Im einfachsten Fall werden jeweils Läufe der Länge 1 erzeugt. *InitialRuns* gibt <u>die Anzahl erzeugter Läufe</u> in einer Variable r zurück. Anschließend wird der Algorithmus *EMergeSort* aufgerufen, der seinerseits jede Mischphase mit der Prozedur *Merge* realisiert.

```
InitialRuns(g1,f1,f2,g2,r)
EMergeSort(f1,f2,g1,g2,r)
ALGORITHM EMergeSort(f1,f2,g1,g2,r)
    Laufzahl:=r
    LOOP
        Merge(f1,f2,g1,g2);
        Laufzahl:=[Laufzahl/2];
        IF Laufzahl=1 THEN EXIT FI;
        Merge(g1,g2,f1,f2)
        IF Laufzahl=1 THEN EXIT FI;
    END LOOP
ALGORITHM Merge(f1,f2,g1,g2)
VAR g: dateiname;
g:=g1;
WHILE es gibt noch mindestens einen unbearbeiteten lauf in f1 oder f2 DO
    IF f1 und f2 enthalten jeweils noch mindestens einen lauf THEN
        verschmelze den ersten lauf in f1 mit dem ersten lauf in f2, wobei
        der ergebnislauf nach g geschrieben wird.
    ELSE
        schreibe den noch übrigen lauf von f1 und f2 nach g
    FI
    IF g=g1 THEN g:=g2 ELSE g:=g1 FI
OD
```

<u>Analyse:</u> In InitialRuns und in jeder Phase werden sämtliche Datensätze jeweils einmal sequentiell gelesen und geschrieben mit Kosten **O(n/b),** wobei b die Seitengröße ist. Der Gesamtaufwand ist daher **O((n/b)*Log(n))** Seitenzugriffe. Der Interne Zeitbedarf ist **O(n*Log(n))**.

Bei diesem Verfahren wird Platz im Hautspeicher kaum *(Lediglich ein paar Seiten Puffer)* genutzt. In der Praxis sind daher <u>Kombinationen von internen und externen Verfahren</u> von Bedeutung.

Wie ist ein allgemeiner Baum definiert:

<u>Definition:</u> allgemeiner Baum

1. Ein einzelner Baum ist ein Knoten
2. Wenn x ein Knoten ist und T_1, T_2,....,T_k Bäume sind, dann ist auch das (k+1)-Tupel (x, T_1, T_2,....,T_k) ein Baum.

Der **Grad eines Knotens** ist die Anzahl seiner Söhne. Der **Grad eines Baumes** ist der maximale Grad eines Knotens im Baum. Ein **Wald** ist eine Menge von m ($m \geq 0$) Bäumen mit disjunkten Knotenmengen. Man erhält einen Wald, wenn man zum Beispiel die Wurzel eines Baumes entfernt.

Die maximale Höhe eines Baumes vom Grad d mit n Knoten ist n-1. Die minmale Höhe eines Baumes mit Grad d und n Knoten ist $O(Log_d(n))$

Was sind Priority-Queues?

<u>Allgemein</u>: Priority-Queues sind Warteschlangen *(FIFO, Elemente werden nur vorne entnommen und hinten angehängt)* die gemäß einer Priorität geordnet *(sortiert)* sind.