

ADS: Algorithmen und Datenstrukturen

Teil $[\pi]$

Uwe Quasthoff

Institut für Informatik
Abteilung Automatische Sprachverarbeitung
Universität Leipzig

24.10.2017

[Letzte Aktualisierung: 23/10/2017, 16:59]

Aufgabe: Finde das i -kleinste Element einer Liste L

— Spezialfälle: kleinstes (Minimum), zweitkleinstes, ... Element, mittlerer Wert (Median), größtes (Maximum)

Bei sortierter Liste: trivial

Bei unsortierter Liste: Minimum/Maximum-Bestimmung hat lineare Kosten $O(n)$

Algorithmus: Einfache Lösung für i -kleinstes Element

Wiederhole $i-1$ mal:

Bestimme kleinstes Element in L und entferne es aus L
gib Minimum der Restliste als Ergebnis zurück

Komplexität $O(in)$, bzw. für $i \in O(n)$ (z.B. "Median"): $O(n^2)$

JEDOCH: es geht besser

Allgemeines Verfahren, um große Probleme schneller zu lösen:

- 1 Ist das Problem zu gross, um sich “trivial” effizient lösen zu lassen, zerlege das Problem in zwei oder mehrere Teilprobleme. (Divide)
- 2 Löse die Teilprobleme einzeln nach der gleichen Methode (d.h. rekursiv). (Conquer)
- 3 Konstruiere die Gesamtlösung aus den Teillösungen. (Merge)

Das Verfahren wird in Schritt 2 rekursiv angewendet, bis hinreichend triviale (kleine) Probleme vorliegen, die sich mit anderen Mitteln lösen lassen.

Frage: Wieso lässt sich damit die Komplexität verringern?

Sehen wir uns zunächst Beispiele an

Suche in sortierter Liste als Beispiel für D&C

Sequenzielle Suche → Binärsuche

Teilungsschritt: Die Liste wird in der Mitte geteilt

Triviale Liste: Liste der Länge 1; einfacher Vergleich

Gesamtlösung: Erfolgreich, falls in einem Teil gefunden

Unser Algorithmus zur Binärsuche lässt sich umformen zu:

Falls die Liste die Länge 1 hat: Prüfe, ob es sich um das gesuchte Element handelt und gib JA oder NEIN zurück.

Sonst zerlege die Liste in der Mitte und prüfe durch Vergleich des gesuchten Elements mit der Trennstelle, in welcher Hälfte das gesuchte Element liegen muss.

Wende diesen Algorithmus rekursiv auf diese Teilliste an.

Zur Erinnerung:

Verbesserung der Zeitkomplexität von $O(n)$ zu $O(\log n)$

 **Wie folgt diese Komplexität aus dem Mastertheorem?**

2. D&C-Beispiel: Maximale Teilsumme

Idee: Wie zuvor zerlege die gegebene Folge in zwei Teile; bestimme die maximalen Teilsummen für beide Teile.

i	0	1	2	3	4	5	6	7	8	9	10
$F[i]$	+2	+5	-8	+4	+2	+5	+7	-2	-7	+3	+5

ABER: Die maximale Teilfolge könnte gerade die Schnittstelle einschließen

DESHALB: linkes und rechtes Randmaximum einer Folge F

- linkes Randmaximum von F : maximale Summe aller Teilfolgen, die bis zum linken Rand (Anfang) von F reichen
- analog: rechtes Randmaximum

Randmaxima lassen sich in $O(n)$ bestimmen!

Rekursiver (D&C-)Algorithmus für maximale Teilsumme

- falls Eingabefolge F nur aus einer Zahl z besteht, nimm $\max(z, 0)$
- falls F wenigstens 2 Elemente umfasst
 - Zerlege F in etwa gleich große Hälften
 - bestimme *rekursiv* die maximalen Teilsummen m_l und m_r der linken und der rechten Hälfte
 - bestimme (in linearer Zeit)
 - das rechte Randmaximum s_l der linken Hälfte und
 - das linke Randmaximum s_r der rechten Hälfte
 - Maximale Teilsumme von F ist $\max(m_l, s_l + s_r, m_r)$



Komplexitätsverbesserung?

Zurück zum Auswahlproblem

Suche i -kleinstes Element von n unterschiedlichen Elementen

- 1 bestimme Pivot-Element p (z.B. letztes Element)

$$L = [-----]p$$

- 2 Teile die n Elemente bezüglich p in 2 Teillisten L_1 und L_2 . L_1 enthält die i Elemente $< p$; L_2 die restlichen Elemente $> p$.

$$L' = [.....L_1.....]p[.....L_2.....]$$

- 3 Falls $L_1.length = i - 1$, dann ist p das i -kleinste Element
Falls $L_1.length > i - 1$, wende das Verfahren rekursiv auf L_1 an
Falls $L_1.length < i - 1$, wende Verfahren rekursiv auf L_2 an; aber bestimme das $(i - 1 - L_1.length)$ -kleinste Element.
- 4 **Komplexität:** Nehmen wir an, dass die Liste jeweils halbiert werden kann, dann ergibt sich aus $T(n) = T(n/2) + \Theta(n)$ nach Mastertheorem: $O(n)$



Ist die Annahme für Komplexitätsabschätzung realistisch?

Weitere Beispiele folgen beim Thema Sortieren, z.B. Quicksort.



In welchem Sinne war unsere schnelle Multiplikation eine Anwendung von Teile-und-Herrsche?

Annahme: wir wollen ein Problem der Größe $n = 2^r$ lösen und zerlegen jeweils in zwei gleich grosse Teile.

Beobachtungen:

- 1 die maximale Rekursionstiefe ist $r = \log_2 n$
- 2 die Anzahl der Teilprobleme in Rekursionstiefe k ist 2^k
- 3 muss in jeder Tiefe nur ein Teilproblem gelöst werden (wie beim Suchen), so sind es insgesamt $r = \log_2 n$ Teilprobleme
- 4 so entsteht der Faktor $\log n$ bei der Komplexität
- 5 muss jedes Teilproblem gelöst werden, so sind n elementare Probleme zu lösen und diese Lösungen zu verknüpfen. Das macht speziell Sinn für Probleme der Komplexität $O(n^2)$ oder schlechter (z.B. maximale Teilsumme)



Erklären Sie damit, dass Teile-und-Herrsche keine Verbesserung bei sequenzieller Suche in einer ungeordneten Liste bringt.

- Sequentielle Speicherung erlaubt schnelle Suchverfahren
 - falls Sortierung vorliegt
 - da jedes Element über Indexposition direkt ansprechbar
- Nachteile der sequentiellen Speicherung
 - hoher Änderungsaufwand durch Verschiebekosten: $O(n)$
 - schlechte Speicherplatzausnutzung
 - inflexibel bei starkem dynamischem Wachstum
- Abhilfe: verkettete (lineare) Liste
- Spezielle Kennzeichnung erforderlich für
 - Listenanfang (Anker)
 - Listenende
 - leere Liste

Verkettete Liste: Implementierung

- Listenanfang wird durch speziellen Zeiger `head` (Kopf, Anker) markiert
- Leere Liste: `head = null`
- Listenende: `next-Zeiger = null`

Element:

keydata	next
---------	------

Listenanfang: `head`-Zeiger auf das erste Element

Listenende: `next`-Zeiger = `null`-pointer \emptyset

Beispiel für verkettete Liste:



Anmerkung (Implementierung von Zeigern): jedes Listenelement liegt im Speicher an einer bestimmten Stelle, die durch eine ganze Zahl angegeben wird (=Adresse). Wert eines Zeigers = Adresse = Zahl: z.B.

head=64; ... 32:

13...50

 ... 42:

99...0

 ... 50:

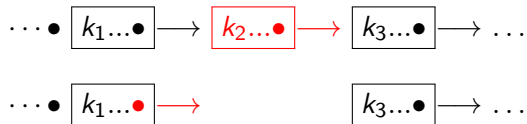
34...42

 ... 64:

7...32

 ...

Verkettete Liste: Löschen



Lösche das Element hinter dem Element, auf das der Zeiger p zeigt:

```
if ( p != null AND p → next != null )  
    p → next = p → next → next
```

Verkettete Liste: Suchen

1. (Initialisieren): Setze $p := \text{head}$
2. (Test): Gesuchtes Element hier gefunden?
Falls ja, `return("gefunden")`
3. (Abbruch?): $p = \text{Listenende}$?
Falls ja, `return("nicht gefunden")`.
4. (Iteration): Setze $p := \text{nächstes_Element}$.
Gehe zu 2.

Nur sequentielle Suche möglich (ob geordnet oder ungeordnet)!

Aber: Einfügen und Löschen eines Elementes mit Schlüsselwert x erfordert vorherige Suche.



Wie funktioniert Löschen eines Elementes auf das ein Zeiger p zeigt? Wie funktioniert Verketteten ("Hintereinanderfügen") von 2 Listen ?

```
SEARCH( Schlüssel x, Liste L ) {  
    Zeiger p=L.head;  
    while (p → key != x) { p = p → next; }  
    return p;  
}
```

Wir nehmen hier an, dass das gesuchte Element enthalten ist. Was gibt SEARCH(x,L) zurück?



Was passiert, wenn kein Element mit Schlüssel x vorkommt?
Was sollte an dem Code geändert werden?

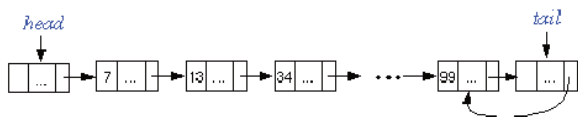
Bisheriger Nachteil von Listen: Bei Listenoperationen müssen viele Sonderfälle abgeprüft werden (Zeiger auf Null prüfen etc.)

Alternativ: Liste mit Kopf- und Schwanzzeiger

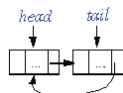
Dummy Elemente: `head` und `tail`

(ausserdem: `L.head` und `L.tail` zeigen auf diese Elemente)

- `tail` → `next` zeigt auf das letzte echte Listenelement
- Next-Zeiger des Dummy-Elementes am Listende verweist auf vorangehendes Element (erleichtert Hintereinanderfügen zweier Listen und Abprüfen von Sonderfällen)



Implementierung "leere Liste":



[`head` → `next` → `next` = `head`]



Wie vereinfacht das Listenoperationen?
z.B. **VERKETTEN(L1,L2)**

Zeiger nicht nur zum Nachfolger *next*, sondern auch zum Vorgänger *previous*; (zusätzlich wieder Dummy-Elemente *head*, *tail*)

Eigenschaften

- Bestimmung des Vorgängers in konstanter Zeit ($p \rightarrow \text{previous}$)
- höherer Speicherplatzbedarf als bei einfacher Verkettung
- Aktualisierungsoperationen etwas aufwendiger (Anpassung der Verkettung)
- Suchaufwand in etwa gleich hoch
- Flexibler: z.B. Löschen des Elements, auf das p zeigt, wird einfacher (Operation $\text{DELETE}(p, L)$)

Flexibilität der Doppelverkettung besonders vorteilhaft, wenn Element gleichzeitig Mitglied mehrerer Listen sein kann (Multilist-Strukturen)

Zusammenfassung: Eigenschaften verketteter Listen

- Suchaufwand
 - erfolgreiche Suche: $C_{avg} = (n + 1)/2$ (Standardannahmen: zufällige Schlüsselauswahl; stochastische Unabhängigkeit der gespeicherten Schlüsselmenge)
 - erfolglose Suche:
 - falls unsortiert: vollständiges Durchsuchen aller n Elemente
 - falls sortiert: nur noch Inspektion von $(n + 1)/2$ Elementen (Bei Abbruch der Suche mit gleicher Wahrscheinlichkeit an allen Positionen der Liste)
- Einfügen / Löschen
 - konstante Kosten für Einfügen an Listenanfang (oder an bestimmter Position)
 - konstante Kosten für positionsbezogenes Löschen (bei Doppelverkettung)
 - lineare Kosten für schlüsselbezogenes Löschen
 - lineare Kosten für schlüsselbezogenes Einfügen, d.h. nach einem Element mit gegebenem Schlüssel; auch Einfügen in Sortierreihenfolge
- Verkettung: konstant bei Listen mit head und tail Zeigern

Skip-Listen

Ziel: Datenstruktur mit logarithmischem Aufwand für Suche, Einfügen und Löschen von Schlüsseln

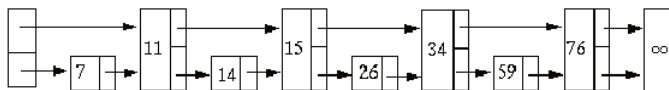
Prinzip

Verwendung sortierter verketteter gespeicherter Liste mit zusätzlichen Zeigern

- Elemente werden in Sortierordnung ihrer Schlüssel verkettet
- Führen mehrerer Verkettungen auf 2 oder mehreren Ebenen:

Verkettung auf Ebene 0 verbindet alle Elemente;

z.B. Verkettung auf Ebene 1 verbindet jedes zweite Element

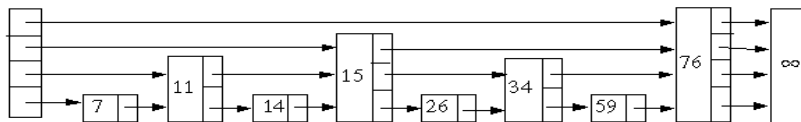


Allgemein: ein Zeiger auf Ebene i zeigt auf 2^i -nächstes Element



Warum kann man jetzt schneller suchen als in einfacher Liste?

Perfekte Skip-Liste - Verkettung



Zeiger:

(Annahme: Listenlänge $n = 2^k$)

Ebene 0: Zeiger an jeder Position (auf nächste P.) n

Ebene 1: Zeiger an jeder 2-ten Position (auf übernächste P.) $n/2$

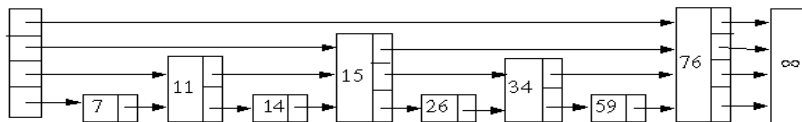
Ebene 2: Zeiger an jeder 4-ten Position (auf 4.-nächste P.) $n/4$

...

Ebene k : Zeiger an erster Position (auf tail) 1

Insgesamt $n + n/2 + n/4 + \dots + 1 = \sum_{i=0}^k \frac{n}{2^i} < 2n$ Zeiger

Perfekte Skip-Liste: weitere Eigenschaften & Suche



Anzahl der Ebenen = $1 + \lfloor \log_2 n \rfloor$

Höhe eines Elements = Anzahl der Zeiger des Elements - 1

maximale Höhe: $\lfloor \log_2 n \rfloor$

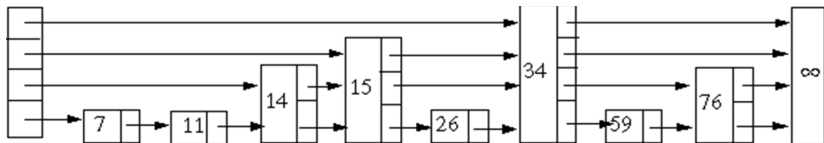
Gesamtzahl der Zeiger $< 2n$

Suche: ähnlich zu binärer Suche $\Rightarrow O(\log n)$

ABER: perfekte Skip-Listen zu aufwendig bezüglich Einfügungen und Löschvorgängen (vollständige Reorganisation erforderlich, Kosten $O(n)$)

Randomisierte Skip-Listen

- Aufgeben der “Perfektion”: keine starre Festlegung der Höhe eines Elementes nach seiner Position
- Höhe eines neuen Elementes x wird zufällig gewählt, aber mit Wahrscheinlichkeit jeder Höhe so wie in perfekter Skipliste, d.h. $P(h) = 1/(2 \cdot 2^h)$ (für Höhen $h = 0, 1, 2, \dots$)
- Somit entsteht eine “zufällige” Struktur der Liste
- Kosten für Einfügen und Löschen im wesentlichen durch Aufsuchen der Einfügeposition bzw. des Elementes bestimmt: $O(\log n)$



Stacks als spezielle Listen

Synonyme: Stapel, Keller, LIFO-Liste usw.

Stack kann als spezielle Liste aufgefaßt werden, bei der alle Einfügungen und Löschungen nur an einem Ende, TOP genannt, vorgenommen werden

Abstrakter Daten Typ (ADT) **Stack** gegeben durch Operationen:

- 1 CREATE: Erzeugt den leeren Stack
- 2 INIT(S): Initialisiert S als leeren Stack
- 3 PUSH(S, x): Fügt das Element x als oberstes Element von S ein
- 4 POP(S): Löschen des Elementes, das als letztes in den Stack S eingefügt wurde
- 5 TOP(S): Abfragen des Elementes, das als letztes in den Stack S eingefügt wurde
- 6 EMPTY(S): Abfragen, ob der Stack S leer ist

Alle Operationen mit konstanten Kosten realisierbar: $O(1)$

Beispiel: $S = |a, b$ POP(S) $S = |a$ PUSH(S, c) $S = |a, c$

EMPTY(S) \rightarrow false

Definition

- $()$ ist ein wohlgeformter Klammerausdruck (wgK)
- Sind w_1 und w_2 wgK, so ist auch ihre Konkatenation $w_1 w_2$ ein wgK
- Mit w ist auch (w) ein wgK
- Nur die nach den vorigen Regeln gebildeten Zeichenreihen bilden wgK

Beispiel: $((()) ())$ ist wohlgeformt.

Algorithmus: "Erkennen von wgK"

Idee: Benutze Stapel zum Speichern öffnender Klammern.

- Einlesen des Ausdrucks von links nach rechts.
- Bei Lesen von öffnender Klammer: Speichere diese auf Stack (PUSH)
- Lesen schließender Klammer: Entferne oberstes Stack-Element (POP)
- wgK liegt vor, wenn der Stack vor POP nie leer war und er am Ende leer ist (EMPTY)

Anwendung: Berechnung von Ausdrücken in Umgekehrt Polnischer Notation (UPN, Postfix-Ausdrücke)

UPN = Postfix-Notation arithmetischer Ausdrücke; erlaubt besonders einfache Auswertung mit Stack

Beispiel: $(1+2) * (4+6/3) \implies 1\ 2\ +\ 4\ 6\ 3\ /\ +\ *$

Auswertung:

- Lese Symbole in UPN von links nach rechts.
- Wenn Zahl: lege auf Stack.
- Wenn m -stelliger Operator: nehme m Operanden vom Stack, führe Operation aus und lege Ergebnis auf Stack.

Beispiel: $(1+2) * (4+6/3) \implies 1\ 2\ +\ 4\ 6\ 3\ /\ + *$

Stack	Schritt	1
		1 2 +

Beispiel: $(1+2) * (4+6/3) \implies 1\ 2\ +\ 4\ 6\ 3\ /\ + *$

Stack Schritt 2
 3

Beispiel: $(1+2) * (4+6/3) \implies 1\ 2\ +\ 4\ 6\ 3\ /\ +\ *$

Stack	Schritt	3
		3 4 6 3 /

Beispiel: $(1+2) * (4+6/3) \implies 1\ 2\ +\ 4\ 6\ 3\ /\ + *$

Stack	Schritt	4
3	4	2

Beispiel: $(1+2) * (4+6/3) \implies 1\ 2\ +\ 4\ 6\ 3\ /\ + *$

Stack	Schritt	5
	3 4 2 +	

Beispiel: $(1+2) * (4+6/3) \implies 1\ 2\ +\ 4\ 6\ 3\ /\ + *$

Stack	Schritt	6
3	6	

Beispiel: $(1+2) * (4+6/3) \implies 1\ 2\ +\ 4\ 6\ 3\ /\ + *$

Stack Schritt 7
3 6 *

Beispiel: $(1+2) * (4+6/3) \implies 1\ 2\ +\ 4\ 6\ 3\ /\ + *$

Stack Schritt 8
18

Synonyme: FIFO-Schlange, Warteschlange, Queue

Spezielle Liste, bei der die Elemente an einem Ende (hinten) eingefügt und am anderen Ende (vorne) entfernt werden

ADT Schlange / Operationen:

- CREATE: Erzeugt die leere Schlange
- INIT (Q): Initialisiert Q als leere Schlange
- ENQUEUE (Q, x): Fügt das Element x am Ende der Schlange Q ein
- DEQUEUE (Q): Löschen des Elementes, das am längsten in der Schlange verweilt (erstes Element)
- FRONT (Q): Abfragen des ersten Elementes in der Schlange
- EMPTY (Q): Abfragen, ob die Schlange leer ist

Beispiel:

$Q = [a, b, c]$ ENQUEUE(Q, d) $[a, b, c, d]$ DEQUEUE(Q) $[b, c, d]$

Vorrangwarteschlangen (Priority Queues)

Jedes Element erhält eine Priorität. Entfernt wird stets Element mit der höchsten Priorität (Aufgabe des FIFO-Verhaltens einfacher Warteschlangen)

ADT Priority Queue / Operationen:

- CREATE: Erzeugt die leere Schlange
- INIT(P): Initialisiert P als leere Schlange
- INSERT(P, x): Fügt neues Element x in Schlange P ein
- DELETE(P): Löschen des Elementes mit der höchsten Priorität aus P
- MIN(P): Abfragen des Elementes mit der höchsten Priorität
- EMPTY(P): Abfragen, ob Schlange P leer ist.

Sortierung nach Prioritäten beschleunigt Operationen DELETE und MIN auf Kosten von INSERT.

Beispiel: $P=[9, 7, 4, 3]$ INSERT(P,5) $[9, 7, 5, 4, 3]$
INSERT(P,1) $[9, 7, 5, 4, 3, 1]$ DELETE(P) $[7, 5, 4, 3, 1]$

- Verkettete Listen
 - dynamische Datenstrukturen mit geringem Speicheraufwand und geringem Änderungsaufwand
 - Implementierungen: einfach vs. doppelt verkettete Listen
 - hohe Flexibilität
 - hohe Suchkosten
- Skip-Listen
 - logarithmische Suchkosten
 - randomisierte statt perfekter Skip-Listen zur Begrenzung des Änderungsaufwandes
- ADT-Beispiele: Stack, Queue, Priority Queue
 - spezielle Listen mit eingeschränkten Operationen (LIFO bzw. FIFO)
 - formale ADT-Spezifikation zur Festlegung der implementierungsunabhängigen Eigenschaften
 - effiziente Implementierbarkeit der Operationen: $O(1)$
 - zahlreiche Anwendungsmöglichkeiten