4. Suchen

Vgl. Skript Plödereder, SS 01, Folien 65 bis 177

Grundaufgabe:

Gegeben: Menge $A = \{a_1, ..., a_n\} \subseteq B$ sowie ein Element $b \in B$.

Realisiere A in einer geeigneten Datenstruktur, so dass die folgenden drei Operationen "effizient" durchgeführt werden:

- Entscheide, ob b in A liegt (und gib ggf. an, wo). FIND
- Füge b in A ein. INSERT
- Entferne b aus A. DELETE

Statt des meist sehr umfangreichen Elements b betrachten wir nur eine Komponente s von b. Dieses s nennen wir "Suchelement" oder meistens "**Schlüssel**" (englisch: "**key**").

Weitere Aufgabe: Wähle eine Datenstruktur so, dass einige der folgenden Tätigkeiten effizient durchführbar sind.

Gegeben seien zwei Mengen $A_1, A_2 \subseteq B$.

- Vereinige A_1 und A_2. UNION
- Schneide A_1 und A_2. INTERSECTION
- Bilde das Komplement $B \setminus A_1$. Complement
- Entscheide, ob A_1 leer ist. EMPTINESS
- Entscheide, ob $A_1 = A_2$ ist. EQUALITY
- Entscheide, ob $A_1 \subseteq A_2$ ist. SUBSET
- Entscheide, ob $A_1 \cap A_2$ leer ist. Empty Intersection

Annahme: Die Mengen haben keine Struktur (insbesondere sind sie nicht geordnet).

In diesem Fall muss man die die Elemente der Menge "wie sie kommen" in eine Liste einfügen.

Worst-Case-Aufwand der drei Operationen der Grundaufgabe, wenn die Menge A genau n Elemente enthält:

FIND: Durchlauf durch die Liste, also O(n).

INSERT: Füge das neue Element am Anfang ein: O(1). (Elemente treten dann mehrfach in der Liste auf. Will man dies nicht, dann: O(n).)

DELETE: Zunächst das Element finden, dann aus der Liste ausklinken: O(n).

4.1 Suchen in "flachen" Strukturen

Unter "flachen" Strukturen verstehen wir Felder (arrays) und lineare Listen.

Wir nehmen im Folgenden stets an, dass die zugrunde liegenden Mengen angeordnet sind.

Grund: Alle Elemente werden im Rechner durch eine Folge von Nullen und Einsen dargestellt. Dies impliziert eine (lexikografische) Anordnung, die wir stets ausnutzen können.

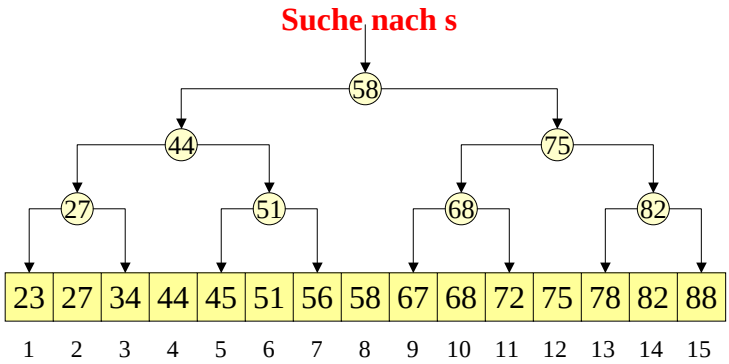
Die Mengen seien also geordnet.

1. Wir wählen ein array als Datenstruktur, in das die Menge der Größe nach geordnet eingetragen wird. Dann

- dauert FIND nur O(log(n)) Schritte, sofern man ein geordnetes array verwendet und hierauf mit der Intervallschachtelung sucht (vgl. Abschnitt 2.4, Seite 29-39),
- dauert INSERT aber O(n) Schritte, da beim Einfügen alle größeren Elemente um eine Komponente nach hinten verschoben werden müssen,
- dauert DELETE ebenfalls O(n) Schritte, da beim Löschen alle größeren Elemente um eine Komponente nach vorne verschoben werden müssen.

Erinnerung aus Abschnitt 2.4: Intervallschachtelung

Wir skizzieren die Verhältnisse, wobei wir hier $n=15=2^4-1$ wählen:



Lässt sich eine der Operationen noch beschleunigen? Unter der folgenden Bedingung, ja, für die Operation FIND.

Bei der Intervallschachtelung (=binäre Suche) halbieren wir jeweils das gesamte noch verbleibende Feld A(links..rechts). Als Teilungsindex berechnen wir (vgl. Abschnitt 2.4):
m = (rechts + links) / 2 = links + (rechts - links) / 2.

Kann man mit den Schlüsseln "rechnen" und sind die Schlüssel recht gleichmäßig über den Indexbereich verteilt, so kann man den ungefähren Bereich, wo ein Schlüssel s im Feld A(links..rechts) liegen muss, genauer angeben durch den Teilungsindex

$$m = links + \frac{s - A(links)}{A(rechts) - A(links)} (rechts - links).$$

So geht man beispielsweise beim Suchen in einem Lexikon vor.

Man kann zeigen, dass mit dieser "Interpolationssuche" die Operation FIND nur noch den unitären Zeitaufwand O(log(log(n))) benötigt und, falls die Schlüssel gleichverteilt sind, so braucht man nur mit 1·log(log(n)) + 1 Schritten zu rechnen.

Da log(log(n)) eine sehr schwach wachsende Funktion ist, sollte man die Interpolationssuche einsetzen, wo immer es statthaft ist.

2. Bitvektoren

Da $A = \{a_1, ..., a_n\} \subseteq B = \{b_1, ..., b_r\}$ eine Teilmenge ist (mit $a_i \leq a_j$ und $b_i \leq b_j$ für $i < j$), kann man A auch durch einen Bitvektor $x = (x_1, ..., x_r)$, mit $x_i \in \{0, 1\}$, der Länge r darstellen, wobei für alle i gilt: $x_i = 1 \Leftrightarrow b_i \in A$.

Wird nach dem Element $s \in B$ gesucht und kennt man die Nummer, die s in der Anordnung von B trägt (also das i mit $s=b_i$), dann gilt für eine Teilmenge A mit Bitvektor x:
FIND: $s = b_i \in A \Leftrightarrow x_i = 1$.
INSERT: Setze $x_i = 1$.
DELETE: Setze $x_i = 0$.
Im unitären Komplexitätsmaß läuft dann alles in O(1), also in konstanter Zeit ab!

Dennoch verwendet man diese Darstellung mit Bitvektoren nur selten, weil in der Praxis B meist sehr groß (wenn nicht sogar unendlich groß) ist. Auch benötigen alle Operationen der "weiteren Aufgabe" (vgl. Seite 2) den Aufwand O(r), während man in der Praxis höchstens auf O(n) kommen darf (n = Größe der betrachteten Teilmengen von B).

Ist dagegen die Kardinalität $r = |B|$ nicht zu groß, dann sind Bitvektoren eine geeignete und vor allem eine leicht zu implementierende Darstellung für Mengen; auch die üblichen Mengenoperationen wie Vereinigung, Durchschnitt usw. lassen sich leicht implementieren.

Im weiteren Verlauf der Vorlesung wurde folgende Binärbaum-Animation vorgestellt:

http://w3studi.informatik.uni-stuttgart.de/~fassbebn/