Föreläsning 5
Datastrukturer (DAT037)

Fredrik Lindblad[1]

2016-11-14

[1]Slides skapade av Nils Anders Danielsson har använts som utgångspunkt. Se
`http://www.cse.chalmers.se/edu/year/2015/course/DAT037`

Förra gången:

- Cirkulära arrayer
- Prioritetskö
- Binära heapar

Leftistheapar

`merge`

- Det verkar inte gå att slå ihop två binära heapar, implementerade med arrayer, på ett effektivt sätt.
- Med *leftistheapar*: $O(\log n)$

Leftistheapar

- Heapordnade (ofta pekarbaserade) binära träd, som inte är kompletta utan uppfyller annan balanseringsinvariant, leftist träd-egenskapen.
- Grundläggande operation: `merge`.
- Lätt att implementera `insert`, `delete-min` med hjälp av `merge`.

Null path length

För en nod, X, i ett träd är $\mathrm{npl}(X)$ (null path length of X) kortaste vägen till ett löv. (Jämför höjden som är längsta vägen till ett löv.) För tomt träd är null path length -1 (liksom för höjden).

Leftist träd-egenskapen

För varje nod i trädet gäller för dess vänster- och högerbarn, l och r, att $\mathrm{npl}(l) \geq \mathrm{npl}(r)$.
För ett leftist träd gäller att den högraste vägen har max $\lfloor \log(N+1) \rfloor$ noder, där N är totalt antal noder.

Merge

Behöver lagra npl för varje nod.

Rekursiv implementering:

- Om något av träden är tomma, ta det andra trädet.
- Låt $h_{\leq}$ vara trädet med minst rot och $h_{>}$ det andra. Slå ihop $h_{\leq}.h$ (höger delträd) med $h_{>}$ (rekursivt anrop).
- Ersätt $h_{\leq}.h$ med det nya trädet, som vi får anta (induktion) är en leftist heap med alla element som förekommer i $h_{\leq}.h$ och $h_{>}$.

Merge

- Eftersom vi valde delträd av det med minst rot vet vi att det nya delträdet uppfyller heap-ordningen gentemot föräldern. Hela det resulterande trädet uppfyller alltså heap-ordningen.
- Men kanske inte leftist-egenskapen. Om $\mathrm{npl}(h_{\leq}.v) < \mathrm{npl}(h_{\leq}.h)$ så byt helt enkelt plats på barnen.
- Slutligen uppdatera npl i roten ($h_{\leq}$). Den är $\mathrm{npl}(h_{\leq}.h) + 1$.

Tidskomplexitet för merge

Om vi anropar merge för h_1 och h_2 så är antalet rekursiva anrop begränsat av antalet noder i högraste vägen i h_1 + antalet noder i högraste vägen i h_2. Dessa är $O(\log(|h_1|))$ resp. $O(\log(|h_2|))$. Varje ekvering av den rekursiva operationen (exklusive det rekursiva anropet) tar konstant tid. Alltså är komplexiteten för merge $O(\log(|h_1|) + \log(|h_2|)) = O(\log(\max(|h_1|, |h_2|))$, d.v.s. $O(\log(n))$ om n är storleken på den största heapen.

Implementering av insert och deleteMin

- Insert implementeras genom att slå ihop trädet med ett träd bestående av ett element (det nya). Tidskomplexitet: $O(\log(n))$.
- deleteMin genom att ersätta roten med hopslagningen av vänster och höger delträd. Tidskomplexitet: $O(\log(n))$.

Mängder, avbildningar

Mängder

ADT:

- Konstruerare för tom mängd.
- `insert`(x): Lägger till x till mängden. Mängder innehåller ej dubbletter.
- `member`(x): Avgör om x finns i mängden.
- `delete`(x): Tar bort x från mängden.
- Gränssnitt i Java: `Set`

Avbildningar/maps

ADT:

- Konstruerare för tom avbildning.
- `insert`(k, v): Lägger till bindningen $k \mapsto v$, mellan en nyckel och ett värde.
  Om det finns en gammal bindning $k \mapsto v'$ skrivs den (t ex) över.
- `lookup`(k): Om det finns en bindning $k \mapsto v$ så ges v som svar.
- `delete`(k): Tar bort bindningen $k \mapsto v$ (om det finns en sådan bindning).
- Gränssnitt i Java: `Map`

En mängd är en avbildning

Mängd är ett specialfall av avbildning (en avbildning där värdet inte innehåller något). I java finns samma implementeringar för `Map` som för `Set`:

- `HashMap` och `HashSet`
- `TreeMap` och `TreeSet`
- ...

Krav på nyckeltypen

Olika mängd-/avbildningsdatastrukturer har olika krav på nyckeltypen:

- Likhetstest.
- Olikhetstest/komparator.
- Hashfunktion.

Hashtabeller

Hashtabeller

- Implementerar mängd- eller avbildnings-ADTn.
- Använder array för att lagring av alla element/bindningar.
- Nyckeltypen kan vara vad som helst, t.ex. stora heltal eller strängar.
- Kan inte ha en plats för varje tänkbar nyckel (de är alldelas för många).
- Idé: Funktion som säger i vilken "hink" man ska lägga elementen/bindningarna.
- `hashCode()` finns i Javas `Object`.

Hashfunktioner

- $h(k)$ funktion från nyckel till heltal.
- Dessa funktioner är kopplade till nyckel-typen och känner inte till hashtabellens storlek.
- Låt index vara $f(k) = h(k) \bmod n \in \{0 \ldots n-1\}$ där n är arrayens storlek.
- Krav: Om $x = y$ ska $h(x) = h(y)$.
- Dock kan $h(x) = h(y)$ och $f(x) = f(y)$ för $x \neq y$: kollision.
- Finns olika metoder för att hantera kollisioner, separat kedjning (chaining), öppen adressering

Rehashing

- När hashtabellen fylls blir så småningom antalet kollisioner för många.
- Likt dynamisk array, allokera (multiplikativt) större array.
- Nycklarna har samma hash-kod men p.g.a. att arrayens storlek ändras kommer de hamna på andra platser och fördelar över hela nya arrayen.

Skapa en hashtabell med kapacitet 5, och stoppa in följande värden: $3, 7, 8, 2, 9, 11$.
Använd hashfunktionen $h(x) = x$.
Hur många kollisioner inträffar?
(Tabellens kapacitet ändras inte.)

- 0.
- 1.
- 2.
- 3.
- 4.
- 5.

Hashtabeller, separat kedjning

Varje hink kan lagra en mängd element, traditionellt i en länkad lista. På nästföljande slides följer pseudo-kod för en implementering.

Hashtabeller, separat kedjning

```
class HashTable<A>:
  private int          size
  private List<A> [] table

  HashTable(int capacity):
    initialise(capacity)

  private initialise(int capacity):
    if capacity <= 0 then
      raise error

    size  = 0
    table = new array of size capacity
    for each position i in table do
      table[i] = new LinkedList<A>()
```

Hashtabeller, separat kedjning

```
member(A x):
  List<A> bucket = table[x.hash() mod table.length()]
  return bucket.contains(x)

delete(A x):
  List<A> bucket = table[x.hash() mod table.length()]

  if bucket.contains(x) then
    bucket.remove(x)
    size--
```

Hashtabeller, separat kedjning

```
insert(A x):
  List<A> bucket = table[x.hash() mod table.length()]

  if bucket.contains(x) then
    bucket.remove(x)
    bucket.add(x)
  else
    bucket.add(x)
    size++
    if size is "too large" then
      rehash
```

Hashtabeller, separat kedjning

```
private rehash:
  oldtable = table

  initialise("suitable" capacity)

  for each position i in oldtable do
    for each element x in oldtable[i] do
      insert(x)
```

Hashtabeller, öppen adressering

- Inga länkade listor.
- Vid kollision:
  element sparas på annan position i arrayen.
- Måste hantera det faktum att borttagna element en gång fanns där, annars kan man misslyckas att hitta andra element.
- Vid försök i tittar man på plats $(f(k) + p(i)) \bmod n$, där $p(0) = 0$ (första försöket).
- Klassiska val av $p(i)$ är linear probing, quadratic probing.
- Annat alternativ är dubbelhashning.

Sammanfattning

- Leftistheapar
- Mängder/avbildningar
- Hashtabeller