DM507 Algoritmer og datastrukturer

Forår 2013

Projekt, del I

Institut for matematik og datalogi
Syddansk Universitet

5. marts, 2013

Dette projekt udleveres i to dele. Hver del har sin deadline, således at afleveringerne, og dermed arbejdet, strækkes over hele semesteret. Deadline for del I er fredag den 15. marts. Projektet skal besvares i grupper af størrelse to.

Mål

Målet for del I af projektet er at implementere to datastrukturer: en *prioritetskø* og en *ordnet dictionary* (ordbog). Implementationerne skal have veldefinerede interfaces, så de kan bruges som klasser i andre programmer. Prioritetskøen skal bruges direkte som biblioteksfunktion senere i del II af projektet, mens arbejdet med dictionary'en kan ses som en forberedelse til elementer af del II.

Opgaver

Opgave 1

Der skal i Java implementeres en datastruktur, som tilbyder (i Java: implementerer) følgende interface:

```
public interface PQ {
    public Element extractMin();
    public void insert(Element e);
}
```

Her er `Element` en simpel type, der implementerer et (prioritet,data)-par. Denne type er defineret ved følgende simple klasse:

```
public class Element {
    public int key;
    public Object data;
    public Element(int i, Object o){
        this.key = i;
        this.data = o;
    }
}
```

De to dele af et objekt `e` af typen `Element` skal blot tilgås som `e.key` og `e.data`.

Elementers prioriteter er altså af typen `int`, og deres associerede data er af typen `Object`. Metoden `extractMin()` returnerer elementet i prioritetskøen med mindst prioritet (er der flere elementer med mindste prioritet, da et vilkårligt af disse elementer). Det må antages at metoden kun kaldes på en ikke-tom prioritetskø. Metoden `insert(e)` indsætter elementet `e` i prioritetskøen. Det må antages at metoden kun kaldes på en prioritetskø med plads til endnu et element. Dette betyder, at det overlades til brugeren af prioritetskøen at sikre at ovenstående antagelser er opfyldt under brug, f.eks. ved at holde styr på antal elementer i prioritetskøen.[1]

Implementationen skal laves ved hjælp af strukturen *heap* i et array af `Element`s, baseret på pseudo-koden i Cormen et al. kapitel 6 (hovedsageligt i afsnit 6.5). Bemærk at der i dette projekt skal laves en *min*-heap struktur, mens bogen formulerer sin pseudo-kode for en max-heap struktur og derfor skal have alle uligheder vendt. Bemærk også at bogens pseudo-kode indekserer arrays startende med 1, mens Java starter med 0 – en simpel måde at anvende bogens pseudo-kode på i Java, er at lægge een til den ønskede længde på array'et, og så ikke at bruge pladsen med index 0 til noget. Bemærk endvidere at parametrene i metoder ikke er præcis de samme i bogens pseudo-kode som i interfacet ovenfor. Dette skyldes dels at i objektorienteret programmering kaldes metoder på et objekt `Q` med syntaksen `Q.metode()` fremfor `metode(Q)`, og dels at bogen kun opererer med prioriteter og ikke elementer. Derudover er `A` i bogens pseudo-kode et array indeholdende heapen, hvilket *ikke* skal kunne tilgås direkte af brugere af et prioritetskø-objekt `Q` på anden måde end gennem metoderne fra interfacet.

Implementationen skal være i form af en Java-klasse, som kan bruges af andre programmer. Klassen skal hedde `PQHeap`, og skal implementere interfacet ovenfor. Klassen skal have een constructor metode `PQHeap(int maxElms)`, der returnerer en ny, tom prioritetskø, og der som argument tager en øvre grænse for antallet af elementer i køen. Der vil være behov for at implementere metoder udover dem i interfacet, til internt brug i klassen.

[1] Dette er *ikke* en robust måde at lave biblioteksfunktioner på, men er simpelt og godt nok til vores formål.

Opgave 2

Der skal i Java implementeres en datastruktur, som tilbyder følgende interface:

```
public interface Dict {
    public void insert(int k);
    public int[] orderedTraversal();
    public boolean search(int k);
}
```

Nøgler er af typen `int`, og elementer består kun af nøgler (der er ikke yderligere data tilknyttet en nøgle). Metoden `search(k)` returnerer blot en boolean, som angiver om nøglen `k` er i træet. Metoden `insert(k)` indsætter nøglen `k` i træet. Metoden `orderedTraversal()` returnerer en kopi af træets elementer i et array (i sorteret orden), fremfor at printe dem på skærmen som i bogens pseudo-kode.

Implementationen skal laves ved hjælp af strukturen *binært søgetræ*, som beskrevet i Cormen et al. kapitel 12. Som det fremgår af ovenstående interface skal der kun implementeres indsættelse (pseudo-kode side 294), søgning (pseudo-kode side 290 eller 291), og inorder gennemløb (pseudo-kode side 288). Træet skal ikke holdes balanceret (dvs. der skal ikke bruges metoder fra kapitel 13).

Implementationen skal være i form af en Java-klasse, som kan bruges af andre programmer. Klassen skal hedde `DictBinTree`, og skal implementere ovenstående interface. Klassen skal have een constructor-metode ved navn `DictBinTree()`, som returnerer en ny, tom dictionary. Der vil være behov for at implementere metoder udover dem i interfacet, til internt brug i klassen. Det vil også være fornuftigt at definere en separat klasse til at repræsentere knuder i træer, således at objekter af typen `DictBinTree` indeholder en reference til knuden der er rod i træet, samt eventuel anden relevant global information on træet, f.eks. dets størrelse. Bemærk at parametrene i bogens pseudo-kode er anderledes end i interfacet ovenfor. Dette skyldes at implementationsdetaljer, såsom at der findes knuder i dictionary'en, ikke skal være synlige for brugere af datastrukturen. Der vil derfor for flere metoder være tale om to udgaver, den officielle fra interfacet, og en intern, som gør det virkelige arbejde. Den officielle kalder blot den interne, og tilføjer i kaldet yderligere parametre med relevante værdier.

Opgave 3

Implementer to sorteringsalgoritmer baseret på din implementationer af ovenstående interfaces. Den ene algoritme skal implementere Heapsort ved

gentagne indsættelser i en prioritetskø, efterfulgt af gentagne extractmin's. Datadelen af elementerne kan her sættes til at være `null`.

Den anden algoritme skal implementere, hvad der kan kaldes Treesort, ved gentagne indsættelser i en dictionary, efterfulgt af et inorder gennemløb.

De to algoritmer skal implementeres hver for sig i to programmer kaldet henholdsvis `Heapsort` og `Treesort`. Disse programmer skal bruge de ovenfor udviklede klasser `PQHeap` og `DictBinTree` som biblioteksfunkioner.

Programmerne skal læse fra standard input, og skrive til standard output. De skal antage, at input består af sekvens af `char`'s bestående af heltal adskildt af whitespace (man kan således bruge en `Scanner` fra biblioteket `java.util` og metoden `nextInt()` til at indlæse tallene). De skal som output skrive tallene fra input i sorteret orden på standard output, (dvs. på skærmen), adskilt af whitespace.

Med andre ord skal `Heapsort` kunne kaldes således:

```
Java Heapsort
34 645 -45 1 34 0
Ctrl-D
```

og skal så give flg. output på skærmen:

```
-45
0
1
34
34
645
```

Programmet `Treesort` skal opføre sig på samme måde. Bemærk at ved hjælp af redirection kan man anvende programmerne på filer:

```
Java Heapsort   < inputfile   > outputfile
```

En detaljer er, at man under scan af input ikke ved, hvor mange der er. Dette er et problem for Heapsort, som skal kende det maksimale antal elementer i prioritetskøen. Een løsning er at allokere en passende størrelse i starten, f.eks. 1000, og så hvis denne fyldes op midlertidig stoppe med scan af input, allokere en heap af den dobbelte størrelse, flytte alle elementer over i denne (via extractmins og inserts), og derefter fortsætte scan med den nye prioritetskø. Løber denne fuld, gentages fordoblingen, etc.[2] En

[2]Man kan med metoder fra kurset DM508 vise, at hvis man udvider via fordobling, bliver den samlede køretid stadig $O(n \log n)$.

anden løsning er at indlæse i en `ArrayList`, og derfra (når scan af input stopper, og man kender antallet af elementer) lave inserts i en prioritetskø.

Formalia

Lav en rapport, som beskriver dine løsninger af opgaverne ovenfor. Da besvarelsen i store træk består af kode, skal rapporten struktureres som følger:

- En kort introduktion, som giver overblik over, hvad rapporten indeholder og besvarer.
- Din kode (evt. undtagen de mest trivielle dele), indlejret mellem tekstdele, som forklarer hvad koden indeholder og gør. Hvis du har nogle interessante designvalg (dvs. nogle, der ikke er eksplicit beskrevet i opgaveteksten), så beskriv disse i størst detalje. For at indlejre kode(bidder) i tekst i LaTeX kan du f.eks. bruge pakken `listings`, eller blot environment'et `verbatim`.
- Et afsnit med beskrivelse af de test, du har udført for at sikre dig at koden fungerer korrekt. Fokuser mest på, hvad du har valgt at teste, mindre på at dokumentere resultatet, sålænge dette er som forventet. Hvis der er måder, du er bekendt med, hvorpå programmet ikke løser den stillede opgave, skal du skrive dette eksplicit.

Husk at skrive navnene på personerne i gruppen på forsiden af rapporten.

Rapporten skal afleveres udprintet i instruktorens boks på Imada (vælg een af instruktorerne, hvis personerne i gruppen går på forskellige hold).

Derudover skal rapporten (i *pdf*-format) samt alle Java source-filer afleveres elektronisk i Blackboard med værktøjet "SDU Assignment", som findes i menuen til venstre på kursussiden i Blackboard. De skal enten afleveres som individuelle filer eller som eet `zip`-arkiv (med alle filer på topniveau, dvs. uden nogen directory struktur).

Det er *vigtigt* at overholde alle syntaktiske regler ovenfor (navngivning af klasser, kald af programmer, `zip`-arkiv uden directory struktur), da programmerne vil blive testet på automatiseret måde.

Aflever materialet senest:

Fredag den 15. marts, 2013, kl. 12:00.