ALDA VT12:Hashning och prioritetsköer

Leon Hennings leonh

Kamyar Sajjadi kamy-saj

2 februari 2012

Innehåll

Koden finns att ladda ner på `http://people.dsv.su.se/~kamy-saj/download/ALDA/`
Detta för att det ska bli lättare för er att göra en peer-review i er favorit editor.

Koden till zip-filen är: **FgH9sg6**

1 MyMiniHeap - Konstruktor och privata variabler

```
1  /**
2  * MyMiniHeap is a d-heap
3  */
4 public class MyMiniHeap<T extends Comparable<? super T>> implements MiniHeap<T>
5 {
6   private T[] heap;
7   private int size;
8   private int d;
9   private static final int DEFAULT_CAPACITY = 10;
10
11   /**
12    * Constructor with no arguments.
13    * This constructor will call the constructor with 2 arguments.
14    * The standard size of the d-heap i 10 with 2 childs
15    */
16   public MyMiniHeap()
17   {
18     this(2,DEFAULT_CAPACITY);
19   }
20
21   /**
22    * Constructor with one arguments.
23    * @param childs The number of childs
24    * Creates a d-heap with standard capacity of 10 and number
25    * of childs is the argument. This constructor will call the
26    * constructor with 2 arguments.
27    */
28   public MyMiniHeap(int childs)
29   {
30     this(childs,DEFAULT_CAPACITY);
31   }
32
33   /**
34    * Constructor with two arguments.
35    * @param childs The number of childs.
36    * @param capacity The size of the heap.
37    * @throws IllegalStateException If childs is less then 2.
38    * Creates a d-heap with given capacity and childrens
39    */
40   @SuppressWarnings("unchecked")
41   public MyMiniHeap(int childs, int capacity)
42   {
43     if(childs < 2)
44       throw new IllegalArgumentException();
45
46     size = 0;
47     d = childs;
48     heap = (T[]) new Comparable[capacity+1];
49   }
```

2 MyMiniHeap - public void insert(T element)

```
1  /**
2   * Inserts an element into the heap, placing it correctly according to heap
3   * properties.
4   * @param element the element to insert.
5   * @throws IllegalArgumentException if the element to insert is null.
6   * hole är den första tomma platsen. I metoden finns det en loop
7   * Denna loop kollar om föräldrarna behöver flyttas, detta då det
8   * nya elementet kan vara mindre än element som redan finns i listan.
9   * Vi använder oss av getParent(int) för att få föräldern för varje
10  * element.
11  */
12 public void insert(T element)
13 {
14   if(element == null)
15     throw new IllegalArgumentException();
16
17   if( size == heap.length - 1 )
18     enlargeheap(heap.length * 2+1);
19
20   int hole = ++size;
21
22   for( ; hole > 1 &&
23        element.compareTo(heap[getParent(hole)]) < 0; hole=getParent(hole))
24     heap[ hole ] = heap[getParent(hole)];
25   heap[ hole ] = element;
26 }
```

3 MyMiniHeap - private void enlargeheap(int newSize)

```
1   /**
2    * Method to enlarge the heap
3    * @param int the new size of the heap
4    */
5   @SuppressWarnings("unchecked")
6   private void enlargeheap( int newSize )
7   {
8     T [] old = heap;
9     heap = (T []) new Comparable[ newSize ];
10    for( int i = 0; i < old.length; i++ )
11      heap[ i ] = old[ i ];
12  }
```

4 MyMiniHeap - private void percolateDown(int hole)

```
1   /**
2    * Internal method to percolate down in the heap.
3    * @param hole the index at which the percolate begins.
4    */
5   private void percolateDown(int hole)
6   {
7     int child;
8     T tmp = heap[ hole ];
9
10    for( ; getChild(hole) <= size; hole = child )
11    {
12      child = getChild(hole);
13
14      //Kollar vilket av barnen som är det minsta och sätter
15      //variabeln child till detta barn.
16      for(int i = 0; i<d; i++)
17        if(child+i <= size &&
18                heap[getChild(hole) + i].compareTo(heap[child]) < 0)
19          child = getChild(hole) + i;
20
21      if( heap[child].compareTo(tmp) < 0 )
22        heap[ hole ] = heap[child];
23      else
24        break;
25    }
26    heap[ hole ] = tmp;
27  }
```

5 MyMiniHeap - public int getChild(int parent)

```
1  /**
2   * Finds the index of the first child for a given parent's index. This
3   * method is normally private, but is used to test the correctness of the
4   * heap.
5   *
6   * @param parent the index of the parent.
7   * @return an integer with the index of the parent's first child.
8   */
9  public int getChild(int parent)
10 {
11   return d * (parent - 1) + 2;
12 }
```

6 MyMiniHeap - public int getParent(int child)

```
1  /**
2   * Finds the index of a parent for a given child's index. This method is
3   * normally private, but is used to test the correctness of the heap.
4   *
5   * @param child the index of the child.
6   * @return an integer with the child's parent's index.
7   */
8  public int getParent(int child)
9  {
10   return ((child - 2) / d) + 1;
11 }
```

7 Hashing

Hashing är ett sätt att placera in element i en tabell för att sedan snabbt kunna leta upp dem. Detta görs baserat på ett datafält i elementet vilket kallas för dess nyckel. En hashfunktion kommer med hjälp av nyckeln och storleken på tabellen att generera den plats som datan ska lagras på. Då elementets plats i datastrukturen är en funktion av nyckeln kan sökning därför göras på konstant tid genom att mata in nyckeln för det sökta elementet, beräkna dess hashvärde och hämta elementet på platsen motsvarande detta. Detta gör att det går snabbt att göra insättningar, borttag och sökningar hashtabeller. Undantaget för detta är de fall då hashfunktionen beräknar samma värde för två olika nycklar, detta kallas en kollision och gör att elementet måste placeras på en annan plats i tabellen. Effektivitet för hashtabeller påverkas därför inte av hur stor tabellen är utan istället pratar man om load factor, hur många element det är i tabellen jämfört med storleken på tabellen.

Hashfunktionen beräknar ett värde baserat på elementets nyckel och får då ett värde som används för att indexera elementet i hashtabellen. Hashfunktionen måste vara referentiell transparent, dvs, den måste alltid generera samma hashvärde för en viss nyckel. Genom att generera ett hashvärde för ett sökt elements nyckel får man möjlighet att komma åt samma index utan att behöva gå igenom listan i onödan. Hashfunktioner måste kunna generera värden som distribuerar elementen så jämt som möjligt över hela tabellens längd. Beroende på vad nyckeln är kan hashfunktionerna behöva fungera väldigt olika. String klassen i java använder följande för att beräkna hashCode():

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

För att beräkna hashvärdet för ett heltal är det vanligt att använda nyckeln modulo hashtabellens längd som hashfunktion. I java heter hashfunktionen hashCode() och alla klasser måste ha en. Default hashCode funktionen använder JVMs interna 32-bits adress till Object vilket gör den oanvändbar för sökning då identiska objekt inte kommer få samma hashvärde.

I vissa fall kan hashfunktionen generera samma värde för olika element. Detta innebär att dessa element ska dela på samma plats. Ett sätt att lösa detta på är genom att använda sig av Separete chaining. Det fungerar på så sätt att varje plats i hashtabellen innehåller en referens som pekar ut det första elementet som är lagrat på den platsen. Varje element i sig refererar till nästa element som har samma hashvärde. Med andra ord så pekar referensen till det första elementet i en länkad lista. När ett element ska läggas till och hashvärdet blir den samma som ett annat elements hashvärde så kommer det nya elementet att läggas in längst fram i den länkade listan. Nu kommer platsen i hashtabellen att referera till det nya elementet och det nya elementet kommer att referera till det elementet som låg först innan insättningen. Anledningen till att man oftast använder sig av en länkad lista framför träd eller andra datastrukturer är att det är inte tänkt att listorna ska bli så långa. Om man har en hashfunktion som genererar värden som inte krockar så ofta för olika element så kommer listan aldrig att bli så stor och därför finns det ingen anledning att använda sig av avancerade datastrukturer. Hashtabeller som använder separate chaining kan väntas prestera bra upp till en load faktor av 1. Eftersom separate chaining använder sig av länkade listor gör det insättningar av element långsamt. Några alternativa sätt att lösa kollisioner finns under samlingsnamnet open addressing och innefattar linear probing, quadratic probing och double hashing.

Ett annat sätt att lösa kollisioner av hashvärden med open addressing heter linear probing. Det fungerar på så vis att när en kollison uppstår söks tabellen igenom sekvensiellt tills en tom plats upptäcks. Ifall slutet av tabellen nås fortsätter sökningen i början av tabellen. Detta kan resultera i att block av av element lagras i sekvens i hashtabellen och kallas primär klustring. Klustringen gör också så att sökning efter element kan ta lika lång tid som insättning. Linear probings effektivitet minskar desto mer tabellen fylls men kan väntas prestera väl så länge tabellen inte har en load factor på 0.5.

För att undvika klustringen som uppstår vid linjär probing kan man istället implementera quadratic probing. Det fungerar på liknande vis som linear probing med skillnaden att det söker med ett intervall som ökar kvadratiskt. Detta förhindrar att kluster uppstår men istället uppstår

en annan komplikation, det inte garanterat att en tom plats kan hittas. Så länge tabellens storlek är ett primtal så kan man förutsätta att det går att sätta in element så länge tabellens load factor är mindre än 0.5.

Eftersom både linear och quadratic probing förlitar sig på andra tidigare insatta element vid sökning så skapar detta komplikationer vid borttagning av elementen. Därför är det nödvändigt att använda lazy deletion i probande hashtabeller. Man markerar elementen som borttagna och hoppar över dem vid sökning tills tabellen blir för full och då de tas bort och tabellen rehashas.

Double hashing är ett annat sätt att lösa kollisions problemet som uppstår vid generering av hash värden. Metoden går ut på att när en kollison uppkommer så kommer en annan hash funktion att anropas. Denna hash funktion kommer att returnera hur många platser längre fram i hash tabellen det ska ske en insättning. Om en ny kollision uppkommer vid det nya indexet så kommer samma process att upprepas tills en tom plats hittas. Det viktiga är att denna hashfunktion ej returnerar värdet 0. Om den skulle returnera 0 så skulle det innebära att den inte kommer att undersöka någon annan plats för insättning av det nya elementet. En annan sak man måste ta hänsyn till är att hash tabellens storlek ska vara ett primtal. Detta för att hashvärdet som genereras ska få alla möjliga platser i tabellen. Nackdelen med att använda sig av double hashing kan vara om det krävs mycket resurser för att generera ett hashvärde. Exempel på detta är string som har hash funktioner som är en dyrbar operation.

När en hashtabell blir för full kommer insättning och sökning börja ta för lång tid. Lösningen på detta är att skapa en ny tabell som är närmsta primtalet till det dubbla av aktuella storleken. Sedan sätter man in alla element i den nya tabellen baserat på nya hashvärden. Det är en dyr operation eftersom den måste gå igenom alla element och flytta dem till den nya tabellen, men eftersom det sker sällan är det okej.

Hashfunktioner används i andra situationer än hashtabeller. Bland annat så används det för att generera checksums för filer så man kan upptäcka om datan är korrumperad. Inom kryptering används också hashfunktioner, kallade envägs funktioner, för att bland annat omvandla lösenord till ett tillstånd som är svårt att dekonstruera men som lätt kan nås genom att ge samma lösenord igen och jämföra dess hashvärde med det lagrade. Andra saker hashfunktioner används till är bloom filter som avgör om element är en del av ett set och i transposition tables. Transposition tables används i spel för att lagra sekvenser av operationer för slippa upprepa beräkningar av dem, t ex använder sig Schack datorer av detta.

I vissa tillfällen är det nödvändigt att tänka på vilken kollisionshanteringsmetod man använder sig av. Till exempel kräver separate chaining extra minne och tid för att skapa nya noder vid insättning av element. Om elementen är stora är denna extra minnesanvändning försumbar men om mindre element lagras kan minnesanvändningen vara överdriven. Därför är separate chaining mer effektiv för lagring av större element och linear probing är bättre för små element. Jämfört med open adressing är tiden för sökning mer eller mindre konstant även vid högre load faktorer. Detta eftersom den som värst behöver gå igenom de element som hashats till samma värde. Användning av open addressing kan leda till bildning av kluster som i värsta fall gör att sökningars tidskomplexitet är kvadratisk vid hög load faktor. Hash tabeller som använder open adressing bör inte ha en load faktor på över 0.5 vilket leder till att mycket utrymme går till spillo. Det orsakar också att tabellens utrymme måste ökas oftare och detta tar ännu mer tid pga rehashning. Separate chaining är att föredra om man förväntar sig en hög load faktor och open addressing mer lönsam om det är låg load faktor.

8 Poolfråga

8.1 Fråga 1

För objekt av typen person, med attributen str:namn, str:adress, int:ålder och int:personnummer, beskriv en hashfunktion för att lagra dem i en hashtabell. Argumentera för varför detta kommer ge minimalt antal kollisioner.

8.2 Fråga 2

Förklara varför längden på hashtabeller generellt bör vara ett primtal. För högre betyg ska du även ange en typ av kollisionshantering där det är kritiskt att tabellängden är ett primtal, visa varför det är så och vad som kan hända ifall tabellens längd inte är ett primtal.

8.3 Fråga 3

Beskriv de strukturella krav som en prioritetskö måste uppfylla och de operationer den måste ha. Ge exempel på en situation då en prioritetskö är att föredra över en vanlig kö.

8.4 Fråga 4

Ge exempel på två enkla sätt att implementera en prioritetskö med en länkad lista. För högre betyg ska du argumentera för vilken du tycker är bäst och varför.