ALDA VT12:Träd samt mera algoritmanalys

Leon Hennings leonh
Kamyar Sajjadi kamy-saj

26 januari 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: **H7grSpQ**

1 AvlTree - size()

```
1  /**
2   * Return size of the tree.
3   */
4  public int size()
5  {
6    return size(root);
7  }
8
9  /**
10  * Internal method to get the size.
11  * @param node the node that roots the subtree.
12  * @return the size of the tree (int).
13  */
14 private int size(AvlNode<AnyType> node)
15 {
16   if(node == null)
17     return 0;
18   else
19     return size(node.left) + size(node.right) + 1;
20 }
```

2 AvlTree - maxHeight()

```
1
2   /**
3    * Return the max height of the tree.
4    */
5   public int maxHeight()
6   {
7     return maxHeight(root);
8   }
9
10  /**
11   * Internal method to get the max height of the tree.
12   * @param node the node that roots the subtree.
13   * @return the max height of the tree.
14   */
15  private int maxHeight(AvlNode<AnyType> node)
16  {
17    int height = 0;
18
19    if(node == null)
20      return -1;
21    else
22      //returnera det högsta av det vänstra och det högra subträdet.
23      return Math.max(maxHeight(node.left)+1, maxHeight(node.right)+1);
24  }
```

3 AvlTree - hasCorrectHeightInfo()

```
1  /**
2   * Controll if the height of the tree is correct.
3   */
4  public boolean hasCorrectHeightInfo()
5  {
6    return hasCorrectHeightInfo(root);
7  }
8
9  /**
10  * Internal method to controll the height info
11  * @param node the node that we want to check
12  * @return true or false
13  *
14  * Är absolut värdet av differansen för de två barnen
15  * är större än 1 så är det något av subträden som
16  * inte har korrekt höjd värde
17  * Annars, Om det barnnoden med högst vikt inte är
18  * ett mindre än nodens höjd så har den fel höjd info
19  * Annars, returnera sant ifall de båda subträden har korrekt höjd
20  */
21 private boolean hasCorrectHeightInfo(AvlNode<AnyType> node)
22 {
23   if(node == null)
24     return true;
25
26   int diff = height(node.left) - height(node.right);
27
28   if(Math.abs(diff)>1)
29     return false;
30   else if(node.height - Math.max(height(node.left), height(node.right)) != 1)
31     return false;
32   else
33     return hasCorrectHeightInfo(node.right) && hasCorrectHeightInfo(node.left);
34 }
```

4 AvlTree - isSearchTree()

```
1   /**
2    * See if the tree is a binary tree.
3    */
4   public boolean isSearchTree()
5   {
6     return isSearchTree(root);
7   }
8
9   /**
10   * Internal method to control if the tree is a binary tree.
11   * @param node the node that roots the subtree.
12   * @return true if the tree is binary else false.
13   *
14   * Undersöker om trädet är ett sökträd genom att se att
15   * högerbarn har högre värde och vänsterbarn har lägre värde
16   * Ifall trädet är tomt returneras true.
17   * Ifall det inte finns ett vänsterbarn sätts boolean l till true.
18   * Annars jämförs elementets värde med vänsterbarnets värde.
19   * Om vänsterbarnet är lägre kallas isSearchTree rekursivt
20   * på vänsterbarnet och sätter det till dess returvärde.
21   * Annars blir l false.
22   * Sedan görs motsvarande för högerbarnet.
23   * Ifall både boolean l och boolean r satts till true
24   * returneras true.
25   */
26  private boolean isSearchTree(AvlNode<AnyType> node)
27  {
28    boolean l, r;
29
30    // Basecase
31    if(isEmpty())
32      return true;
33
34    if(node.left == null)
35      l = true;
36    else
37    {
38      int left = node.element.compareTo(node.left.element);
39      if(left > 0)
40        l = isSearchTree(node.left);
41      else
42        l = false;
43    }
44
45    if(node.right == null)
46      r = true;
47    else
48    {
49      int right = node.element.compareTo(node.right.element);
50      if(right < 0)
51        r = isSearchTree(node.right);
```

```
52        else
53          r = false;
54      }
55      return l && r;
56    }
```

5 AvlTree - Remove()

```
1    /**
2    * Remove from the tree. Nothing is done if x is not found.
3    * @param x the item to remove.
4    */
5   public void remove( AnyType x )
6   {
7     if(isEmpty())
8       System.out.println("The tree is empty");
9     else
10      root = remove(x, root);
11  }
12
13 /**
14  * Internal method for remove.
15  * @param x the item(element) to remove.
16  * @param node the root node.
17  * @return the root node after all changes.
18  */
19 public AvlNode<AnyType> remove( AnyType x, AvlNode<AnyType> node )
20 {
21   if(node == null)
22     return null;
23   //Noden är mindre då går vi ner i vänstra barnet.
24   else if(x.compareTo(node.element) < 0)
25   {
26     //Anropa remove med vänsterbarnet och sätt nodens
27     //vänsterbarn till det som remove kommer returnera.
28     node.left = remove(x, node.left);
29
30     //Om trädet är obalanserat så kollar vi vilken sida
31     //som är den tunga sidan och gör rotationerna.
32     //Antingen är den "höger höger tung" eller så
33     //är den "höger vänster tung".
34     if(height(node.right) - height(node.left) == 2)
35       if(height(node.right.right) >= height(node.right.left))
36         node = rotateWithRightChild(node);
37       else
38         node = doubleWithRightChild(node);
39
40     //Uppdatera vikten i noden.
41     node.height = maxHeight(node);
42   }
43   //Noden är större då går vi ner i högra barnet.
44   else if(x.compareTo(node.element) > 0)
45   {
46     //Anropa remove med högerbarnet och sätt
47     //det som returneras till noden högerbarn.
48     node.right = remove(x, node.right);
49
50     //Om trädet är obalanserat så kollar vi vilken
51     //sida som är den tunga sidan och gör rotationerna.
```

```
52      //Antingen är den "vänster vänster tung" eller så
53      //är den "vänster höger tung"
54      if(height(node.left) - height(node.right) == 2)
55        if(height(node.left.left) >= height(node.left.right))
56          node = rotateWithLeftChild(node);
57        else
58          node = doubleWithLeftChild(node);
59
60      //uppdatera vikten i noden.
61      node.height = maxHeight(node);
62    }
63    else //Rätt nod är hittad. GRATTIS
64      // Om den inte har något högerbarn ta det vänstra barnet.
65      if(node.right == null)
66      {
67        AvlNode<AnyType> tmpReturn = node.left;
68        node = null; // Ta bort den noden vi inte ska ha kvar.
69        return tmpReturn; //returnerar det subträdet som ska kopplas.
70      }
71      // Om den inte har något vänsterbarn ta det högra barnet.
72      else if(node.left == null)
73      {
74        AvlNode<AnyType> tmpReturn = node.right;
75        node = null;
76        return tmpReturn;
77      }
78      else // Noden har två barn.
79      {
80        // rad 83: Hämta den högsta noden i vänstra barnet.
81        // rad 84: remove på findMax noden.
82        // rad 85: sätter nodens element till vänstra barnets högsta element.
83        AvlNode<AnyType> tmpReturn = findMax(node.left);
84        remove(tmpReturn.element, node);
85        node.element = tmpReturn.element;
86      }
87    return node;
88  }
```

6 Algoritmanalys

6.1 boolean add(E element)

Metoden kallar konstruktorn för Element, vilken skapar nytt element och stoppar in den i listan genom att länka dess next till tail och dess previous till elemented innan tail. Eftersom alla operationer i add och operationerna som körs i konstruktorn är konstanta blir även add metoden konstant.

6.2 void add(int index, E element)

Kallar först checkIndex som består av ett villkor som ifall det uppfylls kastar exception, denna metod är konstant. O(1) Sedan kallas getElement som skapar ett temp element av head för att sedan loopa igenom listan fram till index. getElement är linjär, O(N). Därefter kallas Element konstruktorn som är konstant och sedan ökas två variabler med ett, vilket också görs på konstant tid. Då alla satser görs sekventiellt är det den största som avgör ordo värdet, vilken blir O(N).

6.3 boolean addAll(Collection<? extends E>)

Börjar med en loop som går igenom hela den givna samlingen och kallar add(Element E) för varje element i den. add() är konstant och loopen är O(N). Därefter kallas size på den givna collectionen. Eftersom vi inte kan veta vilken sorts collection det är får vi anta att den kan få loopa igenom hela listan för att få reda på antalet element i den. size är därför i värsta fall O(N). Totala tidskomplexiteten blir därför O(2N), O(N) eftersom vi inte räknar konstanter.

6.4 boolean addAll(int index, Collection<? extends E>)

Som metoden ovan har den en loop som itererar N varv. I loopen kallas add(int index, Element E) som i värsta fall går igenom hela this och därför är O(M) vilket gör tidskomplexiteten för loopen till O(NM).

6.5 int indexOf(Object o)

Denna metod har en loop som går igenom samtliga element i den länkade listan. Själva loophuvudet har en tidskomplexitet på O(N). I loopen är det en if else sats med villkor som kollar om det givna objektet är null, om det inte är det så jämförs objectet med ett element i listan med hjälp av equals metoden. Då det givna objectet mycket väl skulle kunna vara en array som måste jämföra varje index med elementet, måste vi anta att .equals är O(M). Allt annat i if-satsen görs på konstant tid. Detta ger tidskomplexiteten O(NM). Då vi inte vet vad det är för slags objekt som finns i listan skulle det kunna vara högre tidskomplexitet för equals, men vi antar hypotesen om den gode programmeraren att det inte är en rysk babuschka docka av datasamlingar i datasamlingar i vår lista.

6.6 int lastIndexOf(Object o)

Först skapas en listIterator, i dess konstruktor kallas checkIndex som är konstant och sedan returneras en ny iterator med ett anrop till getElement som har tidskomplexiteten O(N). Sedan körs en while loop med hasPrevious som villkor. hasPrevious är konstant och while loopen kommer som värst gå igenom hela listan vilket ger loophuvudet O(N). I while loopen kallas först previous som returnerar element före i listan på konstant tid. Efter det är det en if sats som kollar om objektet är null, annars kollar den ifall objektet är lika med elementet som pekats ut av iteratorn, med hjälp av equals. Som tidigare antar vi att equals är O(M). Ifall vilkoret uppfylls körs nextIndex som tar konstant tid. While loopens N gånger equals M ger O(NM).

6.7 boolean retainAll(Collection<?> c)

Först görs två programsatser på konstant tid. Därefter kommer en while loop som kör N gånger. I while loopen ligger ett villkor som kollas varje gång. Villkoret anropar get() och jämför dess returvärde med containsmetoden för samlingen. get() anropar checkIndex som tar konstant tid och getElement som tar O(N). Collections contains metod har vi ingen information om men vi antar att den är O(M) där M är samlingens storlek. Då anropet av get görs före contains kommer deras ordo värden inte multipliceras utan endast contains O(M) räknas. Ifall villkoret inte uppfylls kallas remove(index) som i sin tur kallar getElement vilken är O(N). Denna ligger i sekvens med contains och därför blir loopens innehåll O(M) + O(2N) vilket förenklas till O(N). Detta multiplicerat med while loopens iterationer ger O(N^2).

7 Poolfråga

7.1 Fråga 1

Hur stor skillnad blir tidskomplexiteten för sökning i ett obalanserat träd jämfört med ett balanserat träd? Hur stor blir höjdskillnaden om vi exempelvis sätter in talen 1 till 10 i sekvens.

7.2 Fråga 2

Givet en viss serie insättningar och borttagningar, vilka rotationer kommer göras i AVL trädet?

a) 1, 2 , 3
b) 1, 3 , 2
c) 3 , 2 , 1
d) 3, 1 , 2

Löses för högre betyg:

e) 2 , 1 , 4 , 3 , remove(1)
f) 1, 2 , 3 , 4 , 5 , 6 , 7 , remove(4), remove(5), remove(6)
g) Ifall man lägger till värdena 1 till 100 i sekvens i ett AVL träd kommer bara en sorts rotation att göras, vilken?

7.3 Fråga 3

När en datastruktur blir för stor för att få plats i primärminnet kommer dess hastighet begränsas av antalet accesser till skivminnet. Nämn en trädstruktur som löser detta och beskriv hur den fungerar.