Hemuppgift 4 - DD1340 Introduktion till Datalogi

Peter Boström

2009-02-08

1 Tidskomplexitet

Uppgiften är att beräkna tidskomplexiteten för *medelfallen* av insättning, borttagning och sökning efter element i diverse datastrukturer, inkl. hash table.

1.1 Osorterad vektor

Sökning

$\mathcal{O}(n)$ eftersom elementet kan ligga var som helst i vektorn måste alla element gås igenom. Medelfallet förväntas hittas i mitten. Dvs. måste man gå igenom $\frac{n}{2}$ element, vilket är $\mathcal{O}(n)$.

Insättning

$\mathcal{O}(1)$ eftersom det endast är att placera elementet i slutet av listan. Räknar inte med omallokering av listan, utan antar att det finns plats för elementet.

Borttagning

$\mathcal{O}(n)$ eftersom hälften av listan måste kopieras ett steg till vänster då medelfallet är att elementet tas bort från mitten. $\mathcal{O}(\frac{n}{2}) = \mathcal{O}(n)$.

1.2 Sorterad vektor

Sökning

Binärsökning ger $\mathcal{O}(log(n))$.

Insättning

$\mathcal{O}(n)$. Att hitta var elementet ska placeras är $\mathcal{O}(log(n))$. Det ska genomsnittligt placeras i mitten av listan vilket kräver att hälften av alla element flyttas ett steg till höger. Att flytta hälften av alla element är $\mathcal{O}(n)$.

Borttagning

$\mathcal{O}(n)$ av samma anledning som insättning. Oavsett om sökning efter elementet som ska tas bort ingår i borttagningen eller inte är operationen $\mathcal{O}(n)$. $\mathcal{O}(log(n)) < \mathcal{O}(n)$

1.3 Osorterad enkellänkad lista

Sökning

$\mathcal{O}(n)$. Alla element måste gås igenom tills det hittas. Medelfallet hittas i mitten, vilket är $\mathcal{O}(n)$.

Insättning

$\mathcal{O}(1)$. Att bryta en länk och stoppa in ett annat element är konstant. Eftersom listan är osorterad spelar det ingen roll var utan elementet läggs till längst fram eller i slutet.

Borttagning

$\mathcal{O}(1)$. Av samma anledning som insättning.

1.4 Sorterad enkellänkad lista

Sökning

Fortfarande $\mathcal{O}(n)$ eftersom man måste gå igenom hälften av alla element för att komma till mitten. Går inte att binärsöka.

Insättning

$\mathcal{O}(n)$. Hitta platsen där elementet ska sitta är $\mathcal{O}(n)$. Att sedan sätta in elementet är $\mathcal{O}(1)$. Tillsammans blir det $\mathcal{O}(n)$.

Borttagning

$\mathcal{O}(n)$ av samma anledning som insättning. Om det inte ingår att söka upp elementet eftersom det tidigare hittades med en iterering så är operationen konstant, $\mathcal{O}(1)$

1.5 Hashtabell

Tidskomplexiteten för hashtabellen antar att antalet element är ungefär lika med hashtabellens storlek.
Analysen antar även att hashfunktionen har uniform distribution.

Sökning

$\mathcal{O}(1)$. Vid medelfallet finns inga kollisioner vid hashen mod storlek. Detta gör att hashen hittas på konstant tid. I värre mindre ovanliga fall får man gå igenom två-tre element, men det är mindre vanligt vid uniform distribution. Medelfallet är konstant.

Insättning

$\mathcal{O}(1)$. Fortfarande inga kollisioner för medelfallet. Att lägga till elementet i en tom lista är konstant, annars är det litet nog.

Borttagning

$\mathcal{O}(1)$. Fortfarande inga kollisioner för medelfallet. Att ta bort elementet i en tom lista är konstant, annars är det litet nog.

2 Källkod

HashStringDictionary.java

```
import java.util.List;
import java.util.LinkedList;

/**
 * @author  lemming
 * @version 0xFF
 */
public class HashStringDictionary implements StringDictionary{
    private List<String>[] table;
    private int lists;

    /**
     * Creates a hash table with the given capacity.
     *
     * @throws IllegalArgumentException if capacity <= 0.
     */
    public HashStringDictionary(int capacity) {
        if (capacity <= 0)
            throw new IllegalArgumentException("capacity <= 0");

        /*
         * The array will contain only LinkedList<String>
         * instances, all created in the add method. This
         * is sufficient to ensure type safety.
         */
        @SuppressWarnings("unchecked")
            List<String>[] t = new LinkedList[capacity];
        table = t;

        for(int i = 0; i < capacity; ++i){
            table[i] = new LinkedList<String>();
        }

        lists = capacity;
    }

    /**
     * Adds the given string to this dictionary.
     * Returns <code>true</code> if the dictionary
     * did not already contain the given string.
     *
     * Complexity: O(1) expected time.
     */
    public boolean add(String s) {
        if(contains(s))
            return false;

        table[Math.abs(s.hashCode() % lists)].add(s);
        return true;
    }

    /**
```

```
     * Removes the given string from this dictionary
     * if it is present. Returns <code>true</code> if
     * the dictionary contained the specified element.
     *
     * Complexity: O(1) expected time.
     */
    public boolean remove(String s) {
        return table[Math.abs(s.hashCode() % lists)].remove(s);
    }

    /**
     * Returns <code>true</code> if the string is
     * in this dictionary.
     *
     * Complexity: O(1) expected time.
     */
    public boolean contains(String s) {
        return table[Math.abs(s.hashCode() % lists)].contains(s);
    }
}
```

HashStringDictionaryTest.java

```
import junit.framework.TestCase;

/**
 * @author lemming
 *
 */
public class HashStringDictionaryTest extends TestCase {
    private HashStringDictionary dict;

    /* (non-Javadoc)
     * @see junit.framework.TestCase#setUp()
     */
    protected void setUp() throws Exception {
        dict = new HashStringDictionary(1024);
        super.setUp();
    }

    public void testAdd(){
        assertTrue(dict.add("Shoop"));
        assertTrue(dict.add("that"));
        assertTrue(dict.add("Woop"));
        assertTrue(!dict.add(new String("Shoop")));
    }

    public void testContains(){
        assertTrue(!dict.contains(""));
        assertTrue(!dict.contains("foo"));
        dict.add("string");
        assertTrue(dict.contains("string"));
        assertTrue(!dict.contains("bar"));
    }

    public void testRemove(){
```

```
        dict.add("string");
        dict.add("strang");
        dict.add("strang");
        dict.add("strung");
        assertTrue(dict.remove("string"));
        assertTrue(!dict.contains("string"));
        assertTrue(dict.remove("strang"));
        assertTrue(!dict.contains("strang"));
        assertTrue(dict.contains("strung"));
    }

}
```