Implementing collection of sets with trie : a stepping stone for performances?

Simon Bachelard, Olivier Raynaud, and Yoan Renaud

LIMOS - Université Blaise Pascal,
Campus universitaire des Cézeaux, Clermont-Ferrand, France
{raynaud, renaud}@isima.fr

Abstract. Main operations of the *Set Collection* **A**bstract **D**ata **T**ype are *insertion*, *research* and *deletion*. A well known option to implement these operations is to use hashtable. Another option is to use the data structure known as the trie. Aim of this article is to evaluate performances of the trie data structure when using it to implement the *Set Collection* A.D.T.

1 Introduction

In a general point of view, the concept notion could be seen as a couple of sets (intention/extension). By this way the set of concepts is a collection of sets. In the same manner the context itself is a collection of sets. The natural operations to manage such a collection are *insertion*, *research* and *deletion*. By moving together a kind of objects and a set of operations we are able to define a *Set collection* **A**bstract **D**ata **T**ype (A.D.T.).
One question is then to evaluate the efficiency of the implementation of the operations. We think the efficiency should be supported by two main criterions: need for memory and practical time complexity.
To simulate the *Set Collection* A.D.T. we can use a Map A.D.T. similar to the Map interface of Java language. This Map A.D.T. maps keys to values. In case of set collection, the keys are the collection sets. The Map A.D.T. supplies the following operators: `new()`, `get(e)`, `put(e,value)` and `remove(e)`, `e` being the set, `value` the mapping value. Java language defines an abstract class *Map* (or Interface) and proposes two main implementations using a binary tree for the first one (TreeMap) and hashtables for the other one (HashMap). We propose three new abstract class *Map* implementations on trie structure. The aim is to evaluate the previous two criterions for these three implementations.

2 Trie Implementations and Protocol

The three new implementations use the trie structure. The variations concern the choice made to store the children of each node.

- **ADT_MappingList** (class ADT_L): The children set of a given node is implemented with chained list.

- **ADT_MappingTable** (class ADT_T): The children set of a given node is implemented with an array.
- **ADT_MappingMap** (class ADT_M): The children set of a given node is implemented using the Java class HashTable.

Evaluation of our three Map A.D.T implementations and the two Java Map A.D.T is made on three different kind of integer set collections. The first one is the **Full** collection which consists in all subsets of the ground set X. The **Short** collection consists in collection of sets where size of each set is short compared to the size of X (5%). Finally, the **Large** collection contains large sets compared to the size of X (40%). Moreover, tests were made on different sizes of collection(10^3, 10^4, 10^5, 10^6) and on different sizes of ground set X (100, 500). By this way, we evaluate time and memory performances for the *get()*, the *put()* and the *remove()* operators.

3 Experimental evaluation

3.1 "Need for memory" comparison

All test results concerning Java HashMap and TreeMap are similar. A first observation shows that the trie efficiency is proven (with a list or array implementation) in a **Full** context. A second observation shows that a trie implementation (with list or array) is very competitive in a **Short** context. But greater is the ground set size, less the array implementation is efficient. A third observation concerns the case of **Large** context: our three new implementations need much more memory than the Java ones (multiplied by 2) .

3.2 CPU time executions comparison

First, in general, HashMap has better performances than TreeMap for the operator *put* but this efficiency variation is not significant for *get* and *remove* operators. Then, we notice that ADT_T has appreciably better performances than the two other new implementations for the operator *put*. Nevertheless, the performance of ADT_T decreases when the collection size or the ground set size grows up.
In a global way, we see that performances of the three new implementations for the operator *put* are, in general, better than the two Java ones on **Full** and **Short** collections. If we considere results on all kind of collections, the ADT_L have a good behavior. Finally, if we consider all results, we observe that CPU time execution for the operator *put* depends on the treated collection size. But it doesn't seem to be the same for operators *get* and *remove*. This result confirms theoretical evaluation complexity which shows that these operation complexities do not depend on the collection size.
See [1] for detailed results about this work.

References

1. O. Raynaud S. Bachelard and Y. Renaud. Implementing collection of sets with trie : a stepping stone to performances. Technical report, 2006.