Data Structures and Algorithms

Lab Practice 2: Hash Tables

2019-2020 Course

Contents

1 Introduction

In this lab practice we aim to achieve the following objectives:

- Implement the *Permanent Dictionary* ADT by means of a hash table with linear open addressing.
- Implement a mechanism to dynamically expand hash tables.

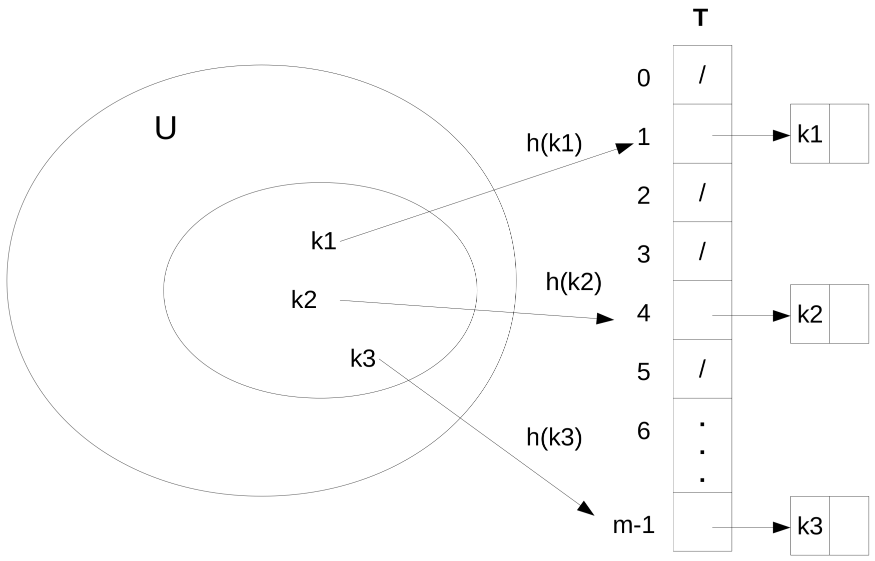
2 Permanent dictionary ADT

The permanent dictionary ADT allows access to a collection of elements using keywords, working with what we commonly call key→value pairs or entries. Below is shown the generic interface of this ADT (`PermanentDictionary`):

```java
package P2;
public interface PermanentDictionary<K,V> {
    void insert(K key, V value) throws ExistingElementException,
   FullDictionaryException;
    V search(K key) throws ElementNotFoundException;
    int size();
    String toString();
}
```

where `K` (*Key*) is the generic type of keys and `V` (*Value*) is the generic type of values (elements). With this interface we can add and get entries, but not remove them, because as its own name indicates, it is a dictionary in which its elements are permanent. We can also get the number of items stored, and we will ensure that we can obtain an appropriate string representation to debug our implementation:

```java
PermanentDictionary<String,Integer> d = ... // create dictionary
d.insert("First",1); // insert entry
d.insert("Second",2); // insert entry
int v = d.search("Second"); // get entry
System.out.println(d); // print dict contents
```



3 Hash tables

To implement the `PermantentDictionary` interface we will use a generic hash table. Hash tables are extremely efficient because in the best and average cases all operations have a constant asymptotic cost $O(1)$. However, when the occupancy of the table is high, the probability of key collision increases and asymptotic costs degenerate to linear $O(n)$.

Internally, a hash table consists of a static vector whose elements are called buckets. Buckets can contain either a key→value entry, or a pointer to another data structure that actually stores entries belonging to that bucket (e.g. a linked list). Given a key→value entry, a hash function is applied to the key, returning the identifier of the bucket (its position in the buckets vector) in which we should find or add such entry.

In case of key collisions within the hash function (two different keys get the same bucket identifier), there are different resolution methods. At theory sessions we have seen the chaining and direct addressing methods. In the first case, entries are stored sequentially within a dynamic list inside the corresponding bucket, while in the second case, entries are relocated by applying a new hash function which depends on the number of previous collisions. Having seen the implementation of the chaining method in theory sessions, in the lab, instead, we will implement the simplest direct addressing method, that is, the linear open addressing method.

4 Hash table with linear open addressing

In a hash table with key collision resolution by linear open addressing, key→value entries (k, v) are directly stored at the m buckets available in the table. Each time the table is accessed to insert or get an entry (k, v), we compute the hash value $h(k, i)$ corresponding to the key k and dependent on the number of previous consecutive collisions $i = \{0 \dots m\}$. The number of collisions is upper-bounded by m (number of buckets), because if m collisions occur this would mean that the table is full. In other words, the $h(k, i)$ function returns the bucket identifier that would correspond to an entry with k key after i consecutive key collisions. This function performs a linear projection of the base index $hash(k)$, and can be formalized as follows:

$$h(k,i) = \begin{cases} hash(k) & \text{if} \quad i = 0 \\ h(k,i-1)+1 & \text{if} \quad i > 0,\ h(k,i-1)+1 < m \\ (h(k,i-1)+1)-m & \text{if} \quad i > 0,\ h(k,i-1)+1 \geq m,\ (h(k,i-1)+1)-m < hash(k) \end{cases}$$

The first line of the function is the base case of the recursion $h(k, 0)$: on the first attempt ($i = 0$, there are no previous collisions) we calculate the index of the key k by calling to the $hash(k)$ function. The second line is applied in case of having consecutive previous collisions ($i > 0$), and basically what we do is to add 1 to the index obtained in the previous step, except if we exceed the m length of the buckets vector. If so, we have to go to the third line of the function, in which we "turn around" the vector starting from position 0 and adding 1 in each step until reaching the origin position $hash(k)$, if necessary.

On the practical side, this method can be implemented as follows. Given an input (k, v) entry, first we will calculate $hash(k)$ to obtain its corresponding bucket index. If that bucket is empty, we will consider that such entry is not contained in the dictionary. However, if the bucket does contain an entry (k', v'), we must check if its key k' matches the key k of the element we intend to add or search. If it matches ($k = k'$), we will have found the searched item. If it does not match ($k \neq k'$), we will have found a key collision and therefore we will have to look for the key in the next bucket, and so on until we find the searched key or find an empty bucket, in which case we will conclude that the entry (k, v) does not exist in the table. It is important to keep in mind that, as long as there are collisions, we must sequentially process all buckets, which may imply having to " turn around" the vector, as we have stated previously.

In short, what this method does is much simpler than what it seems: it performs a linear (and circular) search, starting from the initial position $hash(k)$, over all consecutive buckets until finding an entry containing the searched key or an empty bucket.

4.1 Implementation details

To implement a hash table with collision resolution through open addressing, we will define a generic class `LinearOAHashTable<K, V>` (from *Linear Open Addressing Hash Table*), where `K` (*Key*) is the generic type of keys and `V` (*Value*) is the generic type of values. This class will implement the `PermanentDictionary<K, V>` interface. It will contain a fixed-size table of `m` buckets, that will be represented by a vector of key (`K`) $\rightarrow$ value (`V`) entries. These entries will be implemented by the `TableEntry<K, V>` class. For design reasons and to facilitate code maintenance, it is recommended to nest this class within `LinearOAHashTable<K, V>`, since `TableEntry<K, V>` is exclusively for internal use in the parent class.

```java
public class LinearOAHashTable<K, V> implements PermanentDictionary<K, V> {
    class TableEntry<K,V> {
        ...
        public boolean equals(Object obj) { ... }
        ...
    }
    ...
}
```

Since we are going to compare `TableEntry<K,V>` entries to check whether an entry is contained or not by our hash table, this class must override the `equals(Object)` method inherited from the `Object` class. We will assume that two entries with the same key are equal, regardless of the values they store. Therefore, this method will return `true` only if the keys of both instances are equal, and `false` in any other case.

Another important detail to keep in mind is that, as we must create a vector of `TableEntry<K, V>` objects which contain generic type arguments, due to the limitations imposed by the Java language in this regard, a feasible solution for this particular case would be to use the raw type of `TableEntry` at the time of creating the vector, that is, without specifying the generic type arguments `<K, V>` (`new TableEntry[m]`). This will produce a warning in compilation time, though it will work for us:

```java
public class LinearOAHashTable<K, V> implements PermanentDictionary<K, V> {
    ...
    TableEntry<K,V>[] table;
    public LinearOAHashTable(int m) {
        table = new TableEntry[m];
        ...
    }
    ...
}
```

Regarding the hash function $hash(k)$, we will use the division approach with the `hashCode()` method inherited from the `Object` class[1]. We will consider the absolute value of the integer value returned by this method to avoid negative indices.

```
private int hash(K key) {
    return Math.abs(key.hashCode()) % this.m;
}
```

This class must implement the `toString()` method to be able to print the entries contained by the hash table at any time. An example representation is shown below, showing, from left to right, the bucket identifier, and the pair key and value or `null`, if the corresponding bucket is empty.

```
{
        0:      second  =>      Yohan BLAKE
        1:      third   =>      Justin GATLIN
        2:      null
        3:      first   =>      Usain BOLT
}
```

In addition, to facilitate code debugging, the program will report every time a collision occurs, showing the bucket identifier and the keys involved in the collision. For example:

```
Key Collision in bucket 2: {third,second}
```

For more information about the implementation of the hash table and the linear open addressing method, it is recommended to visit and take as reference the theory notes of the subject.

5 Expandable hash table

One of the main drawbacks when working with a hash table is to determine it size, that is, to choose a proper number of buckets m with respect to the estimation of the number of entries that will be stored in it, in order to minimize the number of key collisions and to guarantee a constant cost in insertion and search operations at all times.

According to the definition of the `LinearOAHashTable<K, V>` class, it contains a hash table of fixed size m which, once created, cannot be resized. Our goal is therefore to improve this implementation with a new class `ExpandableLinearOAHashTable<K, V>` that extends the previous

[1]https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#hashCode()

one with the ability to dynamically adapt the size of the hash table depending on its usage. Thus, if we insert many entries, the hash table will be expanded by increasing the number of buckets, to prevent it from becoming filled, and also, to reduce the probability of key collision.

5.1 Implementation details

To take or not the decision of enlarging the hash table, we have to make use of the concept of current load factor α:

$$\alpha = n/m$$

where n is the number of elements or entries contained within the hash table at a certain time instant, and m the total number of buckets.

For our expandable hash table implementation, we will define a constant with the minimum or initial number of buckets (e.g. 4 buckets). In addition, the hash table will be configured with a maximum load factor $\alpha*$ (`maxLoad`).

```
public class ExpandableLinearOAHashTable<K, V> ... implements
    PermanentDictionary<K, V>{

     public static final int MIN_BUCKETS = 4;
     public ExpandableLinearOAHashTable(double maxLoad) {
          ...
     }
     ...
}
```

On every insert operation, we will check the current load factor α. If this factor exceeds the maximum load factor $\alpha*$, then we will increase the number of buckets. It is important to note that if we modify the number of buckets m of the hash table, the distribution of the elements along the table will be different as the behaviour hash function changes with m. It will therefore be necessary to redistribute all the entries in the extended bucket vector.

To facilitate the debugging of the code, the program will report every time the table is expanded, showing the number of buckets and the load factor before and after the expansion. For example:

```
Expanding Hash Table: (m: 4, alpha: 0.75) -> (m: 8, alpha: 0.375)
```

6 Deliverables

As part of the assessment of the lab practice 2, the following exercises must be done:

1. To implement the `LinearOAHashTable<K, V>` class.
2. To implement the `ExpandableLinearOAHashTable<K, V>` class.

These exercises will be bundled into a single compressed file (`.zip`, `.tgz`, etc.) containing the source code of the projecti, and delivered via a poliformaT task created for this purpose. All `.java` files should be encapsulated within the same package called `P2` (`package P2;`).