(Repo) Documentation for the Symbol Table Data Structure Implementation

Lab 2, Alexandrescu Andrei-Robert (931)

Introduction

For the Symbol Table representation I chose the Hash Table data structure mainly for its fast retrieval times, but also for its intuitive representation and implementation. The following public operations reside in my implementation: (specified at the end of the document)

- int getCapacity()
- pair<int,int> insert(string elem)
- pair<int,int> getPosition(string elem)
- int hash(string elem)
- bool exists(string elem)
- void resize()
- vector<string> getAllElements()
- void print()

Note: The specifications for each method are presented at the end of the document.

Representation

The HashTable is implemented using three variables: a capacity (integer), a variable to keep track of how many elements there are in the table (denoted "n") and a double pointer of **Nodes**.

A **Node** is a structure consisting of: a value (string) and a Node pointer (Node*) called "next".

```
struct Node {
	std::string value;
	Node* next;
};
```

These nodes are essential for the representation which uses linked lists for storing conflicting elements (i.e. elements whose hash value is the identical).

The Hash Function

For the hash function I have used the sum of ASCII codes for each character of the element to be added, divided by the capacity of the hash table. Moreover, as a side note, the capacity should be a prime number for minimizing the number of conflicts. If for instance capacity=13 and the element to be added is the string "abc", then its hash value will be 8. This means the element should be added to the 8^{th} bucket. How was this value computed? (97+98+99) % 13 = 8 where 97,98,99 are the ASCII (decimal) codes for characters 'a', 'b' and 'c' respectively.

A trick that I used for computing the hash function is to apply the mod operation after each addition of a new character to the sum. This way, expensive computations are avoided. Consider the case in which the sum of characters from a 100 characters string very large (millions, even more). Performing the mod operation would be pretty costly. On the other hand, performing the mod operation at each additive step can also drain the computation time of the hash function. For our small programs this should not be that important, although it is mentionable.

Conflict Resolution

As stated before, conflicts are solved by adding the conflicting element at the end of the bucket it must be inserted in. In my code, I gave an example with two values whose hash function is the same: “T” and “d” when capacity=8. They both hash to position 4.

Position definition

Since we are using the linked-list conflict resolution approach, to localize an element in the hash table we need two values: first we must specify the bucket, then the position inside the bucket. For instance, in the example above (with “T” and “d”, capacity=8), bucket 4 looks like this: [T d]. “T” is the first element and “d” is the second one. Their positions are: “T”: <4,0> and “d”:<4,1>. Note that all indexing starts from 0.

Resize

The resize operations is automatically called when the number of elements in the hash table is equal to the capacity of the table. First, all elements are saved to a temporary vector. Then, a larger hash table (double the size) is allocated in the memory. Finally, all elements saved in the temporary vector are reinserted in the freshly created table (to different hashes obviously). This leads to the modification of the PIF (program internal form) whenever a resize operation is performed.

Methods specifications (specification – signature)

```
Constructor method
Input: m - the capacity of the hash table
Output: -

SymbolTable::SymbolTable(int m);
```

```
Returns the capacity of the table
Input: -
Output: the capacity of the table as an integer value

int SymbolTable::getCapacity();
```

```
Insert an element in the symbol table
Input: elem - element of type string
Output: - <-1,-1> if the element already exists
        - otherwise the position where the element was added, <int,int> =
        <bucket,position inside bucket>

std::pair<int,int> SymbolTable::insert(std::string elem);
```

```
        Returns the position of a given element
        Input: elem - element of type string to be searched for
        Output: - <-1,-1> if the element could not be found
                - the position of the element, otherwise

        std::pair<int, int> SymbolTable::getPosition(std::string elem);
```

```
        Toy hash function (sum of ASCII codes % capacity)
        Input: elem - element of type string
        Output: sum of ASCII codes of all characters from the input string modulo the
capacity

        int SymbolTable::hash(std::string elem);
```

```
        Checks whether a given element exists in the table
        Input: elem - element of type string
        Output: - true,  if the element was found
                - false, otherwise

        bool SymbolTable::exists(std::string elem);
```

```
        Resizes the symbol table by doubling its capacity and rehashing all elements
        Input: -
        Output: -

        void SymbolTable::resize();
```

```
        Retrieves a vector containing all elements from the symbol table
        Input: -
        Output: vector of strings containing all elements, sorted by bucket and
position inside the bucket

        std::vector<std::string> SymbolTable::getAllElements();
```

```
        Displays in the console the contents of the table
        Input: -
```

```
Output: - (console output)

void SymbolTable::print();
```