CHAPTER

13

SYMBOL TABLES

13.1 Introduction

Since childhood, we all have used a dictionary, and many of us have a word processor (say, Microsoft Word) which comes with a spell checker. The spell checker is also a dictionary but limited in scope. There are many real time examples for dictionaries and a few of them are:

- Spell checker
- The data dictionary found in database management applications
- Symbol tables generated by loaders, assemblers, and compilers
- Routing tables in networking components (DNS lookup)

In computer science, we generally use the term 'symbol table' rather than 'dictionary' when referring to the abstract data type (ADT).

13.2 What are Symbol Tables?

We can define the *symbol table* as a data structure that associates a *value* with a *key*. It supports the following operations:

- Search whether a particular name is in the table
- Get the attributes of that name
- Modify the attributes of that name
- Insert a new name and its attributes
- Delete a name and its attributes

There are only three basic operations on symbol tables: searching, inserting, and deleting.

Example: DNS lookup. Let us assume that the key in this case is the URL and the value is an IP address.

- Insert URL with specified IP address
- Given URL, find corresponding IP address

Key[Website]	Value [IP Address]
www.CareerMonks.com	128.112.136.11
www.AuthorsInn.com	128.112.128.15
www.AuthInn.com	130.132.143.21
www.klm.com	128.103.060.55
www.CareerMonk.com	209.052.165.60

13.3 Symbol Table Implementations

Before implementing symbol tables, let us enumerate the possible implementations. Symbol tables can be implemented in many ways and some of them are listed below.

Unordered Array Implementation

With this method, just maintaining an array is enough. It needs $O(n)$ time for searching, insertion and deletion in the worst case.

Ordered [Sorted] Array Implementation

In this we maintain a sorted array of keys and values.

- Store in sorted order by key
- keys[i] = i^{th} largest key
- values[i] = value associated with i^{th} largest key

Since the elements are sorted and stored in arrays, we can use a simple binary search for finding an element. It takes $O(logn)$ time for searching and $O(n)$ time for insertion and deletion in the worst case.

Unordered Linked List Implementation

Just maintaining a linked list with two data values is enough for this method. It needs $O(n)$ time for searching, insertion and deletion in the worst case.

Ordered Linked List Implementation

In this method, while inserting the keys, maintain the order of keys in the linked list. Even if the list is sorted, in the worst case it needs $O(n)$ time for searching, insertion and deletion.

Binary Search Trees Implementation

Refer to *Trees* chapter. The advantages of this method are: it does not need much code and it has a fast search [$O(logn)$ on average].

Balanced Binary Search Trees Implementation

Refer to *Trees* chapter. It is an extension of binary search trees implementation and takes $O(logn)$ in worst case for search, insert and delete operations.

Ternary Search Implementation

Refer to *String Algorithms* chapter. This is one of the important methods used for implementing dictionaries.

Hashing Implementation

This method is important. For a complete discussion, refer to the *Hashing* chapter.

13.4 Comparison Table of Symbols for Implementations

Let us consider the following comparison table for all the implementations.

Implementation	Search	Insert	Delete
Unordered Array	n	n	n
Ordered Array (can be implemented with array binary search)	$logn$	n	n
Unordered List	n	n	n
Ordered List	n	n	n
Binary Search Trees ($O(logn)$ on average)	$logn$	$logn$	$logn$
Balanced Binary Search Trees ($O(logn)$ in worst case)	$logn$	$logn$	$logn$
Ternary Search (only change is in logarithms base)	$logn$	$logn$	$logn$
Hashing (O(1) on average)	1	1	1

Notes:

- In the above table, n *is* the input size.
- Table indicates the possible implementations discussed in this book. But, there could be other implementations.