**COMP20230: Data Structures & Algorithms** **2018-19 Semester 2**

# Lecture 11: 27 FEB 2019

*Lecturer: Dr. Andrew Hines* *Scribes: Patrick Kennedy & Kieran Curtin*

**Note**: *LaTeX template courtesy of UC Berkeley EECS dept.*

**Disclaimer**: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

## 11.1 Outline

In this lecture we discuss several new data structures such as the stack, dynamic arrays, doubly linked lists, and symbol tables. The hash table is introduced in this lecture along with two implementations, the separate chaining symbol table, and the linear-probing hash table.

## 11.2 Abstract Data Type: Stack

A Stack is a data structure that functions as a Last In First Out (LIFO) system. It is analogous to a can of pringles or a stack of plates in a cafeteria, where it is easy to add and remove items from the top of the structure. Therefore, the principle operations on stacks are ”push” and ”pop”. Stacks first emerged in 1946 when Alan Turing was working with data structures that allowed ”bury” and ”unbury” operations. The stack only has 2 main operations as well as other non-essential operations suck as ”peek” (to look at the top), however the unique characteristics of this structure can be seen in it’s applications.

### 11.2.1 Application: Backtracking

Consider the problem of navigating through a maze. When a decision on direction has been made, it is not knows if the chosen path is correct. Here, stacks can be implemented as choices (left/right) can be ”pushed” onto the stack. It is also possible to ”pop” back to a stage in the maze before an incorrect decision was made. The resulting stack will contain the precise correct directions through the maze, without errors.

Stacks are also implemented in many programming languages in ”compile time memory management”. Subroutines receive their parameters and return results in a structure called the ”call stack”. Stack overflow occurs when a program tries to use more memory than is available in the call stack(Wikipedia, 2019b).

## 11.3 Dynamic Arrays

In Python, the list is a dynamic array - this allows the array to contain elements of different types (integer, string, variable). These different types will take up different amounts of space in memory, and due to this *dynamic* memory allocation is implemented. A concept of ”logical” -vs- ”actual” size ensures that there is enough space in the list for new elements.

In this scenario, a compromise has to be made between efficiency and size. As it is not known how many extra items are going to be added to the list, it is created with the number of elements as the logical capacity, and *additional*, *unused* space for possible additions as the actual capacity. As items are added, the actual capacity of the list only needs to be extended on every "$n - th$" item (as the list comes closer and closer to the actual memory capacity). The frequency at which the list is extended / amount of new space allocated relationship determines the performance of the algorithm. Allocating new memory is a time consuming task so this should be kept to a minimum, however, allocating larger chunks of memory could result in wasted/unused space in memory.

## 11.4 Doubly Linked Lists

A Doubly linked list is similar to an ordinary linked list. It is a collection of nodes and pointers, however each node has 2 pointers - to the item before and item after the current node. This results in a more efficient data structure when it comes to performance (no need to store the value of previous/next element), but it takes up more space in memory due to the additional pointer (java2novice, 2019).

### 11.4.1 Browser applications

A common implementation of doubly linked lists can be seen in the navigation characteristics of a web browser. Changing to the previous/next page with a back/forward button is a functionality provided by this ADT. Selecting to view the browsing history is in fact a doubly linked list of websites visited in chronological order.

### 11.4.2 Saved states

Doubly linked lists can also be employed to provide extra functionality to text editing software for example. If a mistake is made and "Ctrl-Z" or undo action is called, this is actually reverting the program to a previous saved state. "Undo/Redo" actions in Microsoft Word or Photoshop are an example of doubly linked lists that are pointing to different saved states of the program.

## 11.5 Symbol tables

An example of a Concrete Data Structure symbol table is the python dictionary. Here values are accessed and updated using key:value pairs. The principle operations of "put", "get", and "delete" are powerful when compared to older structures that symbol tables have replaced. One of the reasons that telephone books are becoming obsolete is the unsupported operations on the structure. Symbol tables are powerful structures that are used by programming language compilers and interpreters to translate values. There is no limit to the number of key:value pairs in a symbol table. Examples of data structures that support efficient searchable symbol table implementations are hash tables, binary search trees and balanced search tress. Hash tables are covered below.

### 11.5.1 Hash tables

At its simplest, a hash table is a data structure that maps keys to values. It is comprised of the hash table itself, as well as the hash function, which performs this mapping.

The time complexities of a hash table are given below (Rowell, 2013):

| | Search | Insertion | Deletion |
|---|---|---|---|
| Average Case | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ |
| Worst Case | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |

The requirements of a good hash function are:

1. It needs to return the same number (i.e. array index) for the same input value (key).
2. It should map different inputs (keys) to different numbers.

That being said, collisions are inevitable, so any good implementation of a hash table needs to take this into account.

### 11.5.2 Collisions

Collisions occur when two keys hash to the same array index. What can we do to resolve these collisions? The most common method of collision resolution is called **chaining**. At it's simplest, when collisions do occur in the array, start a linked list at that slot.

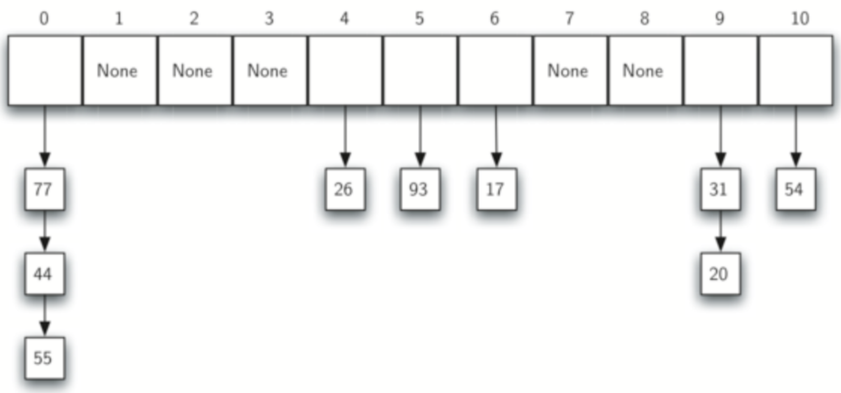

Figure 11.1: Collision resolution via linked lists (Miller & Ranum, 2014)

An implementation of a hash table that uses this solution is called the **Separate Chaining Symbol Table**, described below.

### 11.5.3 Separate Chaining Symbol Table

The Separate Chaining Symbol table implements the Hash, Insert and Search methods as follows.

***Hash:*** Map key to integer i between 0 and M - 1 (where M is the number of linked lists)

***Insert:*** Put at front of $i$th chain (if not already present)

***Search:*** Need to only search chain associated with hashed key.

A problem with this method is how to balance the length of the chains for both insert and search methods.

Typically, hash tables are resized according to the following rules (where N is the number of keys):

When N/M $\geq$ 8, **double** the size of array M.

When N/M $\leq$ 2, **halve** the size of array M.

### 11.5.4 Linear-probing Hash Table

Another solution to the collision problem is the use of **open addressing**. With this method, all elements occupy the hash table itself. If a collision occurs after hashing the key, the element occupies the next open slot in the table.

Linear Probing implements the Hash, Insert and Search methods as follows.

***Hash:*** Map key to integer i between 0 and M - 1 (where M is the number of linked lists)

***Insert:*** Put at table index i if free, otherwise try i+1, i+2 etc. This is the probing element, whereby we examine the table until we find an empty slot.

***Search:*** Search table index i; if occupied but no match, try i+1, i+2 etc.

Resizing is of almost greater importance in the case of linear-probing, as without increasing the size of the hash table, overflows will occur.

With linear-probing hash tables, resizing works as follows:

When N/M $\leq$ 1/2, **double** the size of array M.

When N/M $\leq$ 1/8, **halve** the size of array M.

Need to rehash all keys when resizing.

### 11.5.5 Use Cases

Hash tables have many applications in computer science. These include caching and object representation.

#### 11.5.5.1 Caching

Caches allow for the fast retrieval of data, for instance in DNS lookups, where a server is given a URL and from that determines its IP address. As fast lookup is essential here, a hash table with $\Theta(1)$ look-up is an ideal solution.

#### 11.5.5.2 Object Representation

Many dynamic languages such as Python and JavaScript use hash tables to implement objects, where the keys are the names of the methods and members of the object and the values are pointers to their implementation or value (Wikipedia, 2019a).

## References

java2novice. (2019). *Doubly linked list implementation.* Retrieved from `http://www.java2novice.com/data-structures-in-java/linked-list/doubly-linked-list/` (Last accessed 04 March 2019)

Miller, B. N., & Ranum, D. L. (2014). *Sorting and searching: Hashing.* Retrieved from `https://interactivepython.org/runestone/static/pythonds/SortSearch/Hashing.html` (Last accessed 05 March 2019)

Rowell, E. (2013). *Big-o cheat sheet.* Retrieved from `http://bigocheatsheet.com/` (Last accessed 05 March 2019)

Wikipedia. (2019a). *Hash table.* Retrieved from `https://en.wikipedia.org/wiki/Hash_table` (Last accessed 05 March 2019)

Wikipedia. (2019b). *Stack (abstract data type).* Retrieved from `https://en.wikipedia.org/wiki/Stack_(abstract_data_type)` (Last accessed 02 March 2019)

Link to source: https://www.overleaf.com/project/5c7678eb79ff584d5d8541bc