COMP20230: Data Structures & Algorithms **March 2019**

Lecture 11: Stack ADTs, Family of Arrays, Doubly Linked Lists and Hash Tables

Lecturer: Dr. Andrew Hines *Scribes: Aidan O'Hora & John Hackett*

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.*

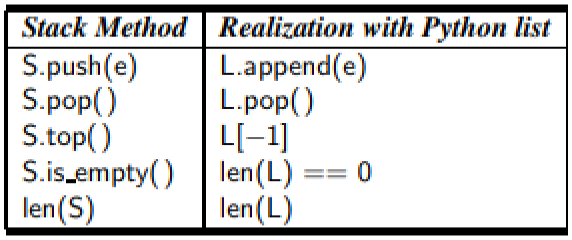
11.1 Outline

In this lecture, we began by revisiting an ADT touched on in a previous lecture, the stack. After this, we took a more in-depth look at the array family of data structures and analysed the properties of dynamic arrays using the example of Python's lists. Building on our previous examination of linked lists, we then introduced the concept of the doubly linked list and compared the two data structures. Finally, we introduced the concept of the hash table and two common strategies for dealing with it's problem of collision: separate chaining and open addressing.

11.2 The Stack

The stack data structure is very similar to the queue data structure discussed in lecture 10, with the major difference being that while objects are added to the back of a queue and removed from the front, *first-in, last-out* (LIFO), objects are both added to and removed from the front of a stack, *first-in, first-out* (FIFO). Essentially, while we are free to add or remove objects from a stack at any time, we may only do so at the "top" of the stack (the location of the most recently added objects). A real world object that share this fundamental feature of the stack ADT is a pez dispenser, where pez can only be easily added to or removed from the top of the pez stacked inside the dispenser:

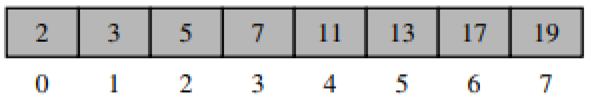
The operation of adding objects to a stack is called ”pushing” while removing objects is called ”popping”. Why? The stack ADT was inspired by spring-loaded cafeteria plate stacks which are ”pushed” down when a new plate is added by it’s weight and ”pop” up when a plate is removed. Actually, that’s just a rumour, but it certainly sounds convincing. The stack ADT also has three other supporting methods, which are equivalent to the size(), is_empty() and front() methods possessed by the queue ADT.



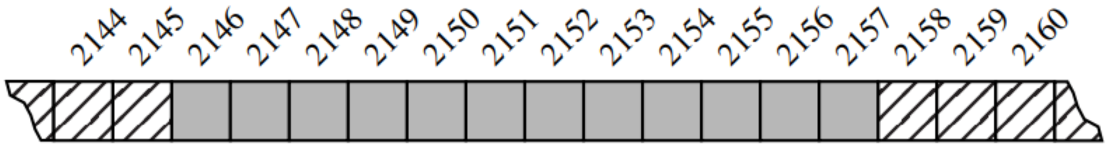
Stack Method	*Realization with Python list*
S.push(e)	L.append(e)
S.pop()	L.pop()
S.top()	L[−1]
S.is_empty()	len(L) == 0
len(S)	len(L)

11.3 The Array Data Structure Family

The defining characteristic of array data structures, and their main advantage over linked lists, is that any object in an array can be efficiently accessed using it’s index. For example, in the array below, the int primitive 5 can be accessed at index 2, 7 can be accessed at index 3, and so on.



Due to this support of access by index, arrays, unlike linked lists, allow for highly efficient insertion or deletion of objects at any location in the sequence. However, while editing the contents of an array is much easier than in the case of a linked list, the underlying structure of the array is much more difficult to modify. This is due the limitation that arrays must be initialised with a set length and a set maximum size for the objects they will contain. This limitation makes sense when the physical implementation of a low-level array in a computer (represented in the image below) is considered, which requires the allocation of a contiguous stretch of memory divided into cells of fixed size to the array at it’s creation. As the memory locations on either side of this low-level array may be in use, expanding such an array may not be possible. This presents a problem when the maximum number of objects that the array may be required to hold is not known at initialisation as well as when an array may be required to contain objects/primitives requiring varying amounts of space (e.g. a mix of strings and ints). This limitation can be overcome through the algorithmic slight-of-hand of the ”dynamic array”, an example of which is the Python list class.



11.3.1 Dynamic Arrays

Understanding the logic behind a dynamic array first requires that the distinction be made between the underlying structure of the array (the size of the pez dispenser, which is not easily changed) and the list sequence of elements stored in it (the stack of pez inside the dispenser, which is easily changed). When a dynamic array is initialised, the underlying array created is usually of a larger size than what is needed to contain it's initial list of elements. This allows for new elements to easily be appended to the list as they can simply be stored at the next available cell in the array. Still, if enough elements are appended we will be back to square one with the list of elements taking up the full size of the array, preventing the further expansion of the list. However, the design of the dynamic array anticipates this, requesting a new array from the system of a larger size once the current array becomes full, which is then reclaimed by the system. While adding an element to the list when it has grown to the full size of the current array will take longer than when there is still space left, this is accounted for in the sense that the new array requested will be larger than what is needed to grow the list by one so that if more elements still need to be appended, it can be done quickly. Below is a pseudocode representation of inserting an element at the end of a list (appending) in a dynamic array:

```
Algorithm insert_at_the_end
Input: $DA$ a dynamic array, $s$ and $c$ two integers representing the size and the capacity of $DA$, $e$ an element
Output: the size of $DA$ grows by 1 and $e$ is inserted at the end of $DA$
  if $s = c$ then
    increase the capacity by a factor of X (you can pick whatever you think if the best progression here)
    For instance:
    Increase the capacity to $c \leftarrow c \times 2$
  end if
  $DA[s] \leftarrow e$
  $s \leftarrow s + 1$
```

A good analogy for the dynamic array is the way that a hermit crab abandons it's current shell once it has outgrown it for a larger one, which it can continue to grow inside until it must find a new larger shell once again.

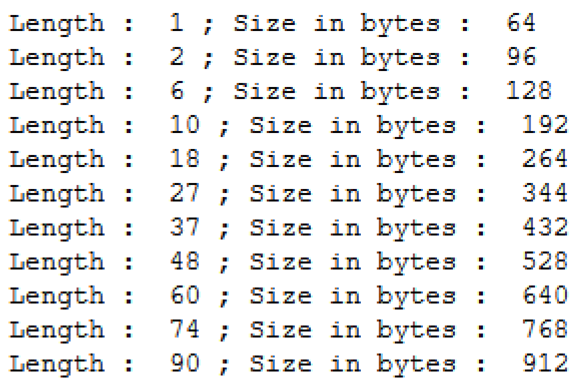
While the dynamic array makes appending to a list efficient and simple, especially when there is already extra space after the list in the array, inserting elements into the middle is more complex and takes more work. Just as with appending, there must be sufficient free space in the array to accommodate the inserting of the new element, otherwise, the list is allocated a new, larger array. In addition to this however, each element from the index of insertion to the end of the list must be moved one cell over, i.e. relocating to their current index + 1. The pseudocode for inserting an element into a dynamic array somewhere other than the end of the list is shown below:

```
Algorithm insert_not_at_the_end
Input: $DA$ a dynamic array, $s$ and $c$ two integers representing the size and the capacity of $DA$, $e$ an element that
  we wish to insert at rank $i$
Output: the size of $DA$ grows by 1 and $e$ is inserted at position $i$
  if $s = c$ then
    increase the capacity by a factor of X (you can pick whatever you think if the best progression here)
    For instance:
    Increase the capacity to $c \leftarrow c \times 2$
  end if
  for $j = i$ to $s$ do
    $DA[j+1] \leftarrow DA[j]$
  end for
  $DA[i] \leftarrow e$
  $s \leftarrow s + 1$
```

11.3.2 Python Lists

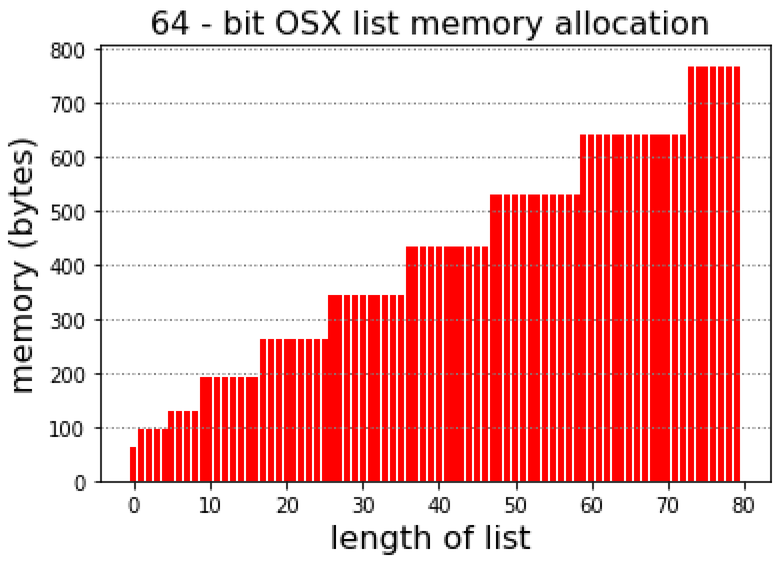
```python
import sys
my_list = [] # This is my list
my_list_size =[] # This list will store the size
b = 0
for i in range (0, 100):
    last_capacity = b
    a = len( my_list)+1
    b = sys.getsizeof(my_list)
    if b > last_capacity:
        print (" Length : ", a , "; Size in bytes : ", b)
    my_list . append (i )
    my_list_size . append (b)
```

One implementation of the dynamic array that we are already familiar with is the Python list class. The code snippet above when run will produce output showing how a Python list is expanded as it is repeatedly reaches it's maximum capacity, which is shown below:



```
Length :  1 ; Size in bytes :  64
Length :  2 ; Size in bytes :  96
Length :  6 ; Size in bytes :  128
Length :  10 ; Size in bytes :  192
Length :  18 ; Size in bytes :  264
Length :  27 ; Size in bytes :  344
Length :  37 ; Size in bytes :  432
Length :  48 ; Size in bytes :  528
Length :  60 ; Size in bytes :  640
Length :  74 ; Size in bytes :  768
Length :  90 ; Size in bytes :  912
```

Investigating the above output, we can see that this Python list was allocated 64 bytes of memory at initialisation. Each time it reaches it's capacity, it is moved to a new dynamic array with more memory and therefore greater capacity. The relationship between the length of a Python list and the memory allocated to it is graphed below:



11.4 Doubly Linked Lists

A doubly linked list consists of nodes forming a linear sequence containing two memory references to the previous and next elements in the list. This provides advantages to specific situations like inserting before an element or deleting before. In fact this can be done in 0(1) time if the position of the node to be modified is known. Access by index is inefficient using a linked list as this index may change over time with modification to the list. So in order to achieve a constant running time, we need to have a reference to the node in question. This is done using the positional list ADT.

11.4.1 The Positional List ADT

The methods in the Positional list ADT takes an element and returns a reference to a node. Returning a direct reference to a node violates object oriented design principles as it allows users to access values directly and potentially make changes. To preserve these principles, the positional list returns an abstract data type called a position which is associated to the node. The positional list ADT is fantastic for modelling situations like queues, where people may cut in or drop out at specific points in the queue.

11.4.2 Header and Trailer Sentinels

Doubly linked lists can include two empty nodes at the first and last position, both pointing to null. These are called sentinels and are used to increase the efficiency of searching through lists in special cases. In a list with no sentinels, if an element is inserted at the first position, then there is no need to check for the previous node being null.

11.5 Hash Tables

A hash table maps keys in a dictionary to an index of a lookup table or potentially an array. This means that for a dictionary containing k items, and n is the largest key in k items, a table of length N must be made where N ¿= n. However this leads to obvious weaknesses whereby a dictionary with two values (1,C) and (100,D) would require a table of length 100. Furthermore, if a key is not an integer or is negative, it may be difficult to use the above method. In order to avoid these issues we use a hash function to map keys to an index.

11.5.1 The Hash Function

The hash function consists of two parts, the hash code and the compression function. The hash function is considered good if it does not create a large number of collisions. Collisions occur when two items are given the same index as a reference. This is solved using bucket arrays for each item in the lookup table generated. The hash code is unbounded and can generate negative integers. The compression factor maps these integers to integers in the look up table. The compression function attempts to make the average distribution of items in the look up table (1/N) where N is the length of the table.

The compression function can help to reduce collisions. Two method of implementing the compression function are the division method and the M.A.D method. The division method maps i to i modulo N (where N is the capacity of the bucket array).

$$i \bmod N,$$

If N is a prime number, this will help space elements out. For example if we were to map 100,200,300 to a bucket size of 100 then we would have collisions for each hash code. But if the bucket size is 101, then there are no collisions.

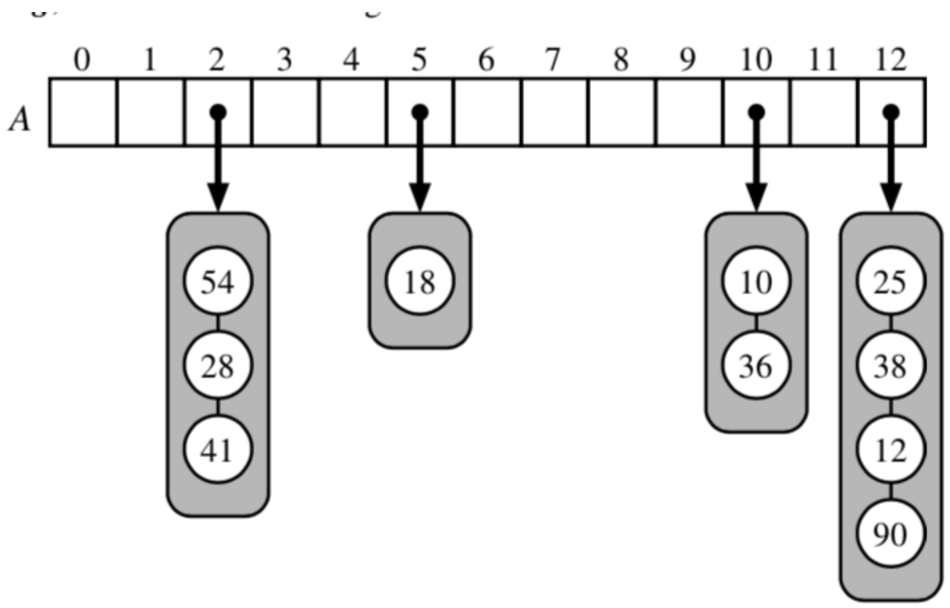
The M.A.D method uses the formula below:

$$[(ai + b) \bmod p] \bmod N,$$

Where p is a prime number larger than N, N is the size of the bucket array as before and a and b are random integers selected from the range [0,p-1], with a being greater than 0. The M.A.D method helps eliminate repeated patterns in a set of integer keys.

11.5.2 Separate Chaining

Separate chaining relates to the creation of bucket array to avoid collisions. If there are n items to be hashed using a table of length N then the expected size of each bucket array will be n/N. This ratio is also called the load factor and determines the complexity of the core operations of the hash table. Preferably the load factor is bounded by one.



Separate chaining suffers from the memory required to store the auxiliary data structure which is handling the data collisions. Open addressing avoids this by storing items directly in the bucket array itself.

11.5.3 Open Addressing

Open Addressing handles collisions by detecting collisions and then trying to insert the data in the next cell. It suffers from large consecutive blocks of data due to this. When an item is trying to be found, the hash code will direct the search to a cell and if the value is not what is expected, the next consecutive cell will be checked, until the correct value is found or an empty cell is identified. When deleting a value in this array, a special marker is used to replace the value so that when searching the array we do not stop searching due to a deleted value which otherwise would be an empty cell.

References

Goldwasser, H., Goodrich, T. & Tammasia, R. (2013) *Data Structures and Algorithsm in Python.* Hoboken: Wiley.