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

Lecture 7: 12 FEB 19

Lecturer: Dr. Andrew Hines *Scribes: Ivan Wahlrab, Luke Deaton*

7.1 Outline

Lecture 7 begins with a jigsaw puzzle analogy, and the methods used to try and solve jigsaws efficiently. Andrew compares this to how this module is covering different topics that may not have an apparent link yet. Just because the big picture has not been revealed yet, does not mean the different topics are not all small parts of the puzzle. Lecture 7 will look at Abstract Data Types and Concrete Data Structures and introduce important considerations when using both.

7.2 Lecture Summary

7.2.1 Abstract Data Types (ADT)

ADTs are independent of any programming language. They are high level tools to help us work with different types of data and can be compared to pseudocode.

An ADT is implemented in a language as a **concrete data structure** or **CDS**. The same ADT can be implemented as more than one CDS within a language. Significant differences can exist between implementations of data structures such as arrays and lists in different languages.

7.2.1.1 Sequence

The sequence ADT is one of the most fundamental data types and it is made up of a homogeneous ordered collection of objects of any type. Python examples: List, Tuple, Str. Some sequence operations are:

- get_elem_at_rank(r) Returns the element of S with rank r.
- set_elem_at_rank(r,e) Replace the element at rank r with e.
- insert_elem_at_rank(r,e) Insert a new element into S with rank r.
- remove_elem_at_rank(r) Remove the element at rank r from S.
- size() Returns the number of items in S.

Many common data structures are variations on a sequence:

- Stack: a sequence where elements can only be added and removed from one end, enforcing ”first-in-first-out”
- Queue: a sequence where elements can be added to one end and removed from the other end only

When storing a data structure, generally a memory address is assigned and then a certain number of bytes is allocated depending upon the data type.

7.2.1.2 Linked List

Linked lists can be quickly and efficiently modified. This is because, rather than assigning each element a fixed address in a contiguous block of memory, it is stored with a pointer to the next element's address. This means an item can be added to the middle of the list by modifying one pointer and adding the appropriate pointer to the new element, avoiding the need to move all the succeeding elements.

However, since it is necessary to iterate through each element in turn to find the location of the last element, accessing elements in a linked list has a time complexity of $O(n)$

Python has an effective way of dynamically assigning memory to lists that grow in size. It assigns extra memory in chunks so that it is not adding extra memory every time the list is added to.

Updating often, accessing seldom: Use Linked List

7.2.1.3 Array

Elements in an array can be accessed very efficiently. When an array is assigned space in memory, the start address and block size (which depends on the data type store in the array) is known. This allows the memory address of element at index n to be calculated as follows:

startAddress + n*sizeOfBlock

which runs in $O(\mathit{1})$ time regardless of the value for n or the size of the array.

However, modifying an array is more difficult than modifying a linked list. For example, because each index in an array represents a specific address in memory, inserting an element into the middle of an array requires the succeeding items to all be moved, giving a worst case time complexity of $O(n)$. Additionally, arrays rely on a contiguous block of memory, which can lead to external fragmentation.

Accessing and modifying often, inserting and deleting seldom: Use Array

7.2.2 Summary

- Abstract Data Types: a higher level representation of a data structure, separate from implementation
- Concrete Data Structures: data structures as implemented in a language
- Trade-offs: trade-offs must be considered depending on the use case - storage complexity vs modification complexity vs access complexity.
- Updating often, accessing seldom: Use Linked List.
- Accessing and modifying often, inserting and deleting seldom: Use Array