Data Structures

This document contains summary of some of the items discussed as part of the COMS 3110 Lectures. This document **does not replace the lecture materials**. This document may contain some topics that are not covered as part of the lecture; you will not be tested on those parts, they are made available to you for gaining further knowledge on topics/concepts that are related to class-lecture.

1 Topics Covered

- Dynamic arrays
- Linked lists
- Stacks and queues
- Binary search tree
- Hashing
- Hash tables

2 Dynamic Arrays

Dynamic array is used to store and retrieve data elements, when the actual number of elements is not known a priori and hence, it is not possible to "statically" allocate the array. The basic idea of dynamic array is

1. Allocate some initial space for array to store the data
2. If there is not enough space to store new data, double the space allocation and create a new array. Copy the data from the old array to new one.

Worst case runtime for inserting data to a dynamic array corresponds to the case, when the array capacity is full. In this case, there is a cost for copying the data resulting in runtime $O(n)$.

The following concept of amortized runtime for dynamic array will not be assessed. This is just presented as interesting/good-to-know information.

Overall time for inserting n elements in a dynamic array. Consider that initially, an array with capacity 1 is created. For each insert, there is an unit time cost for inserting the element and every time, the array size is doubled, there is a time cost for copying elements. Therefore, the time for inserting n elements is

$$\begin{aligned} & \underbrace{(1+1+\ldots+1)}_{n \text{ inserts}} + \underbrace{(1+2+2^2+\ldots+2^k)}_{k \text{ doubling of capacity}} \\ = \quad & n + (2^{k+1}-1) \end{aligned}$$

We know that 2^k is greater than or equal to n as otherwise, the array would not hold n elements. Proceeding further, $2^{k+1} = 2n$ and hence, the overall time for inserting n elements in the dynamic array is $O(n)$.

We say that the amortized runtime for inserting one element in the dynamic array is $O(1)$ (intuitively, this is the average over n inserts).

3 Linked Lists

Please refer to section 10.2 in the textbook.

4 Summary of Runtimes for Common Operations

Understanding the time complexity of common operations across different data structures is crucial for choosing the right one for a given problem.

Operation	**Dynamic Array**	**Linked List**	**Sorted Array**
Search	$O(n)$	$O(n)$	$O(\log n)$
Add	$O(1)$ amortized[1] $O(n)$ worst case	$O(1)$[1]	$O(n)$
Delete	$O(n)$	$O(n)$	$O(n)$

5 Stacks and Queues

Please refer to section 10.1 in the textbook.

6 Binary Search Trees

Please refer to section 12.1 and 12.2 in the textbook.

7 Hashing and Hash Tables

The objective is to design a data structure that stores a collection of elements, each uniquely identified by a key. Such a key can be of any type (e.g., string, integer) with the property that elements have unique keys. The domain of possible keys can be (significantly) larger compared to the number of elements to be stored. For example, ISU id has 9 digits; however, we do not have enough ISU members to exhaust all the 9 digit numbers. Given this, we want a storage system that supports efficient operations for adding new elements, finding existing elements, searching for elements.

A **hash table** is a data structure well-suited for this task. It stores keys-value pairs and allows efficient insertion, deletion, and lookup operations. At its core, a hash table uses a hash function to maps each key to some integer value (also known as a "hash code") such that the integer value can be used as an index of a storage location (e.g., array).

Formally, a **hash function** on a set S is a function $h : S \to \{0, 1, , T-1\}$ that maps elements of S to positions in a hash table of size T. Intuitively, given a key k and a hash function h, $h(k)$ is the index of the storage location where the element with key k is stored.

A hash function is called a **perfect hash function** on S if it maps all distinct elements of S to distinct indices, i.e., for every $x, y \in S$, if $x \neq y$, then $h(x) \neq h(y)$.

Suppose h is a perfect hash function for a set S. We can create a hash table of size T to store the elements of S as follows.

- Create an array A of size T, where each cell initially contains the value NULL.
- To insert an element $x \in S$, compute $h(x)$ and set $A[h(x)] = x$.
- To remove an element x, compute $h(x)$ and set $A[h(x)] =$ NULL.
- To search for an element y, compute $h(y)$ and check whether $A[h(y)] = y$.

[1] Assuming insertion at the end (for dynamic arrays) or at the head (for linked lists). Insertion elsewhere may cost $O(n)$.

The time complexity for insertion, deletion, and lookup is $O(t_h)$, where t_h is the time taken to compute the hash function h.

Typically, the storage capacity is proportional to the number of elements being stored. Even when the capacity is larger, it is not guaranteed that the hash function will always map two distinct keys to two distinct indices. This is because the domain of key is much larger than the number of elements being stored and hence is also much larger than the storage capacity.

When the hash function maps two or more keys to the same index of the storage location, it is referred as hash collision. To handle such unavoidable scenarios, each storage location, instead of having the capability of holding exactly one element, points to a linked list, where the list contains all the elements whose keys are hashed to this storage location (index). At a high-level, consider the storage to be of the form: an array of linked lists. We will term this as the hash table (not to be mixed with the built-in classes in programming languages), a conceptual data structure, and the process of hashing to this table is termed as **chain hashing**.

To insert an element to the hash table T, we need to apply the hash function h on the key k of the element, and add the element to linked list associated to $T[h(k)]$. The runtime is proportional to the cost of applying the hash function, which involves basic arith-operations and hence is $O(1)$.

To find an element in the hash table T, we need to apply the hash function h on the key k of the element, and perform linear search on the linked list associated to $T[h(k)]$. Hence, the runtime is proportional to the length of the linked list, which, in the worst case, is $O(n)$, where n is the number of elements in T. Therefore, the objective is to reduce the length of the list such that the $O(n)$ runtime overhead can be avoided.

Consider that the hash function is such that (a) each index of the hash table is equally likely to be the result of application of hash function on any key and (b) result of application of hash function on one key does not depend on or influenced by the result of hash function on a different key. Based on these two properties, one can infer that the probability that an element will stored in any location of the hash table T is $\frac{1}{m}$, where m is the total capacity of the array structure implementing T (due to property (a)). The expected length of a list at any index of the array implementing T is $\frac{n}{m}$ (due to property (b)). This is referred to as the load factor of the hashing, intuitively capturing that as n (elements being stored) increases compared to m (storage capacity), the hash collision increases. If m is chosen such that it is proportional to n, i.e., $m \in \theta(n)$, then $\frac{n}{m} \in O(1)$.

Therefore, the expected runtime for finding element in a hash table is $O(1)$.

In summary, the two main points to remember and use are

1. the **load factor** of a chaining based hash table is number of elements to store divided by the capacity of the hash table, and
2. the runtime of search/delete operations in such a hash table is **expected**.

Rolling hash. A rolling hash function is commonly used to solve string matching problem to efficiently compute the hash of a sliding window of length m in a string.

Given a string t of length n, the hash value of its substring of length m starting at position j is defined as:

$$h(t, j) = \sum_{i=1}^{m-1} p^{m-i-1} t[i + j],$$

where p is a fixed prime base (e.g., 31 or 37) and $t[i+j]$ represents the ASCII or integer value of the character at position $i + j$.

As an exercise, derive a formula to compute $h(t, j + 1)$, i.e., the hash value of the substring of length m starting at position $j + 1$ in terms of $h(t, j)$, $t[j]$, and $t[j + m]$. How can this formula be used to compute hashes of all length-m substrings of t in $O(n)$ time instead of $O(nm)$?