Problem 1

Question:

A dynamic array is a data structure that maintains an array and allows insertion and deletion operations of elements. As opposed to the conventional (static) arrays which have fixed and pre-determined lengths, for a dynamic array, an arbitrary number of elements can be inserted and thus the data structure needs to be able to grow in length. A common implementation of a dynamic array involves storing elements in a back-end array T. Whenever the number of elements in the array exceeds $\lambda \cdot T$.length where $\lambda \in (0, 1]$ is the load factor, the data structure creates a new empty array T' with twice the length of the current array, and places all elements in T contiguously at the start of T', before setting T' as the new T. In this way the back-end array T "grows" creating more capacity for new elements.

Your task is to explain the following fact: Starting from the empty array, suppose we perform a sequence of m insertion operations. The total running time of these operations is $O(m)$. This means that the average running time of inserting an element into a dynamic array over a sequence of operations is $O(1)$.

Answer:

Suppose we have an empty array and we start adding m (where $m = 2^k$) elements, and we hit the worst case where double resizing occurs on the very last element.

The total cost for m insertions is as follows:

$$(m) + (1 + 2 + 4 + ... + 2^k) = m + \frac{2^{k+1} - 1}{2 - 1} = m + 2 \cdot 2^k - 1 = m + 2m - 1 = \mathcal{O}(m)$$

So since inserting m elements takes about $\mathcal{O}(m)$, we can say on average the time taken is $\frac{\mathcal{O}(m)}{m}$ however since $\mathcal{O}(m) \leq c \cdot m$ we get that

$$\frac{\mathcal{O}(m)}{m} \leq \frac{c \cdot m}{m} = c = \mathcal{O}(c) = \mathcal{O}(1)$$

Therefore on *average* the running time of inserting an element into a dynamic array over a sequence of operations is $\mathcal{O}(1)$.

Problem 2

Question:

Suppose the length w of integers in the universe is bounded by 8 and there are 16 buckets. Suppose we insert $142, 9, 204, 57, 43, 158, 201, 198, 89, 15, 177, 59$ using hash function $h(x) = x \bmod 16$, show the resulting hash table with

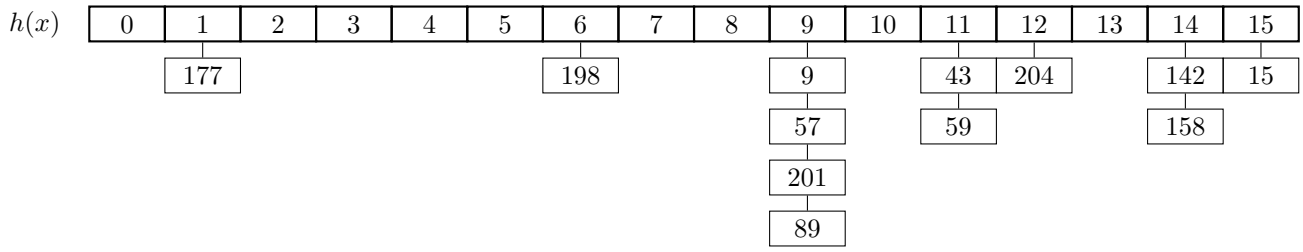
- (a) chaining
- (b) linear probing
- (c) double hashing with $h_2 = 7 - (x \bmod 7)$

Answer:

We can produce this table to make our tables much easier:

x	142	9	204	57	43	158	201	198	89	15	177	59
$h(x)$	14	9	12	9	11	14	9	6	9	15	1	11
$h_2(x)$	5	5	6	6	6	3	2	5	2	6	5	4

(a) Chaining:



(b) Linear Probing:

$h(x)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x	89	15	177	59			198			9	57	43	204	201	142	158

(c) Double Hashing:

$h(x)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x	177	158		89		15	198	59		9		43	204	201	142	57

Problem 3

Question:

The *text pattern matching problem* aims to find the first occurrence of a pattern string $p = p[0]p[1]...p[l-1]$ in a long document $A = A[0]A[1]A[2]...A[n-1]$. A simple way to solve this problem is to examine length l substrings of A of the form $A = A[i]A[i+1]...A[i+l-1]$ where $0 \leq i \leq n-l$ and compare them with the input pattern p. This procedure will take time $O(ln)$. We can improve the running time by utilising hashing. Suppose we use the hash function $h(s) = (s[0] + s[1] + ... + s[l-1]) \bmod 2^d$. The procedure first computes the hash code $h(p)$. Then it compares $h(p)$ with the hash values of substrings $A[0]...A[l-1], A[1]...A[l], A[2]...A[l+1]$ and so on. If we have a match of hash values, then the algorithm compares the pattern string p with that substring character by character to verify the match. The algorithm returns the substring if it does matches with p, and it continues if the match is false. Show that this procedure can take time $O(l+n)$ plus the time spent refuting false matches.

Answer:

We will first consider the *naive* version of this algorithm which takes $\mathcal{O}(l \cdot n)$ time.

Consider this piece of code (written on python) I have written:

```python
def hash(sum):
    return sum % (2 ** d)   #O(1), only one modulo operation

def naive_hash_pattern_finding(pattern, text):  #O(n * l) technically (O(l + n * l))
    pattern_sum = 0
    for i in range(len(pattern)):
        pattern_sum += pattern[i]   #O(l)
    pattern_hash = hash(pattern_sum)

    #entire loop below is O(n * l)
    for i in range(len(text)):      #O(n) as there are n substrings
        sum = 0
        for j in range(len(pattern)):   #O(l)
            sum += text[j]
        substring_hash = hash(sum)      #O(1)

        if pattern_hash == substring_hash:
            #check if match is legit or false match
```

Initially we have to find the hash code of the *pattern* which takes $\mathcal{O}(l)$ time.

Then we have to go through the text and check each possible substring and *compare* each possible substrings hash code with the patterns hash code (and if it is then check if the match is legitimate or false) this whole process takes around $\mathcal{O}(l \cdot n)$.

Therefore the total complexity of this *naive* approach is $\mathcal{O}(l + l \cdot n) = \mathcal{O}(l \cdot n)$.

However we can optimize this algorithm so instead of calculating of the sum of the substring every time, we take the previous sum, minus the first character of the previous substring and add the last character of our new substring:

So consider this new piece of code:

Problem 3 continued on next page...

```python
def optimised_hash_pattern_finding(pattern, text): #O(l+n) (technically O(l+l+n))
    pattern_sum = 0
    for i in range(len(pattern)):   #O(l)
        pattern_sum += pattern[i]
    pattern_hash = hash(pattern_sum)

    sum = 0                     #work out our first sum
    for i in range(len(pattern)):   #O(l)
        sum += text[i]

    for i in range(len(text)):  #O(n) as there are n substrings
        substring_hash = hash(sum)

        if pattern_hash == substring_hash:
            #check if match is legit or false match

        sum -= text[i]
        sum += text[i+len(pattern)]
```

Initially we have to find the hash code of the *pattern* which takes $\mathcal{O}(l)$ time.

We then sum up our text (which is l numbers) therefore this takes $\mathcal{O}(l)$ time.

Now here is where we do the optimizing:

We loop through all possible substrings in the text (which takes $\mathcal{O}(n)$ time) and get the hash code for the substring and compare it with the pattern hash code, after to find the new substring sum, we minus the first character of the previous substring and add the last character of our new substring. So it is kind of like a "sliding window" (example below).

Therefore the total time complexity of this algorithm is: $\mathcal{O}(l + l + n) = \mathcal{O}(l + n)$ plus the time taken to refute false matches.

We can walk through our algorithm with an example as such:

For example if our text was "abcdefgh" and our pattern was "def":

We will start by finding the sum of "def" ($\mathcal{O}(l)$) and then find the hash code of "def" ($\mathcal{O}(1)$).

Then we find the sum of "abc" ($\mathcal{O}(l)$) and enter the loop:

Hash the sum of "abc", compare it with the pattern hash (there is no match then move on)

Find the sum and hashcode of "bcd", compare it with the pattern hash (there is no match then move on)

Find the sum and hashcode of "cde", compare it with the pattern hash (there is no match then move on)

Find the sum and hashcode of "def", compare it with the pattern hash (there is a match, so compare strings and return true)

The whole process takes at most ($\mathcal{O}(n)$)

Therefore the total time complexity is $\mathcal{O}(n + l)$ plus the time taken to refute false matches..

Problem 4

Question:

Chris commute by train each morning from Manukau to Britomart. 90% of trains departing Manukau station on time. 80% of trains arriving in Britomart on time. 75% of trains depart on time and arrive on time.

- (a) Chris takes a train that departs on time. What is the probability that it will arrive on time?
- (b) Chris arrives in Britomart on time. What is the probability that his train departed on time?
- (c) Are the events, departing on time and arriving on time, independent?

Answer:

We know these things from the question:

Let events be: $A =$ Train departing from Manukau on time and $B =$ Train arriving at Britomart ont time

$$\begin{aligned} P(A) &= 0.9 \\ P(A^c) &= 0.1 \\ P(B) &= 0.8 \\ P(B^c) &= 0.2 \\ P(A \cap B) &= 0.75 \end{aligned}$$

(a) The question can be rephrased as: Given the situation where Chris takes a train that departs on time (A), what is the probability that he will arrive on time (B):

Therefore we use the conditional probability formula:

$$\boxed{P(B|A) = \frac{P(A \cap B)}{P(A)} = \frac{0.75}{0.9} = \frac{5}{6}}$$

(b) The question can be rephrased as: Given the situation where Chris arrives at Britomart at time (B), what is the probability that his train departed from Manukau on time (A)?

$$\boxed{P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{0.75}{0.8} = \frac{15}{16}}$$

(c) The test for independence is if $P(A \cap B) = P(A) \cdot P(B)$ therefore

Therefore since we have that $P(A \cap B) = 0.75$ and $P(A) = 0.9$ and $P(B) = 0.8$ we then have that

$$\boxed{P(A \cap B) = P(A) \cdot P(B) \Leftrightarrow 0.75 \neq 0.72}$$

therefore we conclude that the events departing on time from Manukau and arriving on time at Britomart are not independent.