Rushil Gupta October 19, 2024

CS-1204 Problem Set 1

Collaborators: *none*

Problem 1

Our goal is to improve the time complexity of checking for equivalence in Linked Lists, where equivalence is defined as 2 linked lists of the same size and same elements in the same order. We can do this by a time-space tradeoff, where we use hashing methods to check for equivalence.

Method

Let a linked list have n nodes, with values $v_1, v_2, \ldots, v_n$. We can hash each node to a unique integer, using a hash function $h(v_i) = h_i$, which uses returns mod m, which is the same as saying the hash has m buckets.

Now, note that each h_i is less than m, so we can think of the hash as a number in base m, with $h_0, h_1, \ldots, h_{n-1}$ as the digits. So, we create a unique "hash" for each linked list:

$$H = h_0 + h_1 m + h_2 m^2 + \cdots + h_{n-1} m^{n-1}$$

This way, we know if this number is not the same for two linked lists, they cannot be equivalent. However, if the number is the same, we still need to check if the two linked lists are equivalent, since hashing can cause collisions. So, in this case, we need to traverse both linked lists to check for equivalence.

Time Complexity

We want to make sure that the time complexity of insertion and deletion are not affected too much. Let's start by looking at insertion.

Since we always add to the end of the linked list, all we need to do is take the current hash and add $h_n m^n$ to it, where h_n is the hash of the new node, and n is the number of nodes in

the linked list before the insertion. This takes $O(n)$ time, where n is the number of nodes in the linked list (note that this is unchanged).

Now, looking at deletion, we need to remove a node from the linked list. Usually, deletion takes $O(n)$ time, since we have to traverse the list linearly. By adding this hash method, we know we need to iteratre over the list one more time to calculate the new hash and the deletion, so the deletion has a time complexity of $O(n)$ (which is also unchanged).

Now, let's look at the time complexity of comparing for equivalence. When the hashes of 2 linked lists are not the same, we know the linked lists cannot be equivalent, so this takes $O(1)$ time. When the hashes are the same, we know need to compare the linked lists. Since we are comparing each element, this takes $O(n)$ time. Now, to look at the average case, note that for 2 linked lists to have the same hash and different elements, there needs to be at least one collision. The probability of this is the same as $1 - p$, where p is the probability of no collisions.

Clearly, $p = \left(1 - \frac{1}{m^n}\right)$, since the probability of no collisions is the same as the probability of no collisions in a hash table with m buckets and n elements. So the average time complexity is:

$$T(n) = P(\text{collision}) \cdot O(n) + P(\text{no collision}) \cdot O(1) = (1 - p) \cdot O(n) + p \cdot O(1)$$

Since $(1 - \frac{1}{m^n})$ approaches 0 very fast:

$$T(n) = O(1)$$

- Best Case: $O(1)$
- Worst Case: $O(n)$
- Average Case: $O(1)$

Problem 2

We have been given a hash function h with the information that it is a simple uniform hash function. This means that for any value x, $h(x)$ is equally likely to be any value in the range $[0, m-1]$. To show that there exists a subset $U \subseteq \mathbb{Z}$ such that $h_U(x) : U \to [m]$ and $h_U(x) = h(x)$ for all $x \in U$, we start by considering:

$X = h^{-1}(n)$ for some $n < m$. Clearly, X is infinite in size. Also note that $X \subseteq \mathbb{Z}$. But, $\forall x \in X, h(x) = n$. So, when $U = X$, h_U is not a uniform hash function.

Part 2: Implementation

This part of the assignment has been done in q2.cpp. The Ashoka IDs of UG2023 and UG2024 students were taken as the dataset. Here are some tabular results:

Name	Lowest Freq.	Avg Freq.	Highest Freq.
`Modulo`	2.4%	3.12%	3.7%
`Scale, Translate, and Modulo`	2.9%	4.59%	6.3%
`SHA256`	0.7%	1.59%	3.1%

Problem 3

Lemma 1

Lemma 1: If the size of the hash table is m, then at most $(m+1)/2$ unique indices would be checked by quadratic probing when looking for the next available index using the function:

$$h(k) + i^2 \mod m$$

Proof: Proof:

We first show that $i^2 \mod m$ has at most $\frac{m+1}{2}$ unique values when i ranges from 0 to $m-1$.

Consider the set $M = \left\{0, 1, 2, \ldots, \frac{m-1}{2}\right\}$. For each $i \in M$, compute $r_i = i^2 \mod m$.

Note that for any i, $(-i)^2 \equiv i^2 \mod m$. This means the squares are symmetric around $i = 0$ modulo m. Therefore, the values of $i^2 \mod m$ for $i = \frac{m+1}{2}, \frac{m+3}{2}, \ldots, m-1$ are repetitions of the values obtained from $i = 1$ to $\frac{m-1}{2}$.

Including $i = 0$ (since $0^2 \mod m = 0$), there are exactly $\frac{m+1}{2}$ unique values of $i^2 \mod m$:

- $\frac{m-1}{2}$ unique non-zero residues from $i = 1$ to $\frac{m-1}{2}$, - Plus one value for $i = 0$.

Therefore, when using quadratic probing with the function $h(k) + i^2 \mod m$, we can get at most $\frac{m+1}{2}$ unique indices before the sequence of $i^2 \mod m$ values starts repeating.

Hence, at most $\frac{m+1}{2}$ unique indices would be checked by quadratic probing when searching for the next available index.

□

Lemma 2

Lemma 2: If the size of the hash table m is prime, the choice of i renders $(m+1)/2$ unique values for:

$$h(k) + i^2 \mod m$$

Proof: Since m is a prime number, we can leverage properties of quadratic residues in modular arithmetic to determine the number of unique values produced by the function $h(k) + i^2 \mod m$.

Earlier we saw we only get $\frac{m+1}{2}$ unique values for $i^2 \mod m$, so the number of unique values for $h(k) + i^2 \mod m$ is also $\frac{m+1}{2}$.

However, when m is a prime, we know that whenever $i^2 \equiv j^2 mod m$, $(i = j)$, i.e.:

$$(i^2 \equiv j^2) \implies (i = j)$$

Proof:

$$i^2 \equiv j^2 \implies i^2 - j^2 \equiv 0 \implies (i - j)(i + j) \equiv 0$$

Since m is prime, $i - j \equiv 0$ or $i + j \equiv 0$. But, $i - j \equiv 0$ implies $i = j$. The other case is not possible, since we saw earlier that i and j are both less than $(m - 1)/2$, so their sum cannot be 0 modulo m.

Hence, we get $\frac{m+1}{2}$ unique values for $h(k) + i^2 \mod m$.

□

Problem 4

We will solve this using Doubly linked lists.

Pseudocode Implementation

```
struct Node:
    data: element
    prev: pointer to previous Node
    next: pointer to next Node

class Deque:
    head: pointer to Node
    tail: pointer to Node
    size: integer

    constructor():
        head = null
        tail = null
        size = 0

    pushFront(element):
        newNode = new Node(element, null, head)
        if isEmpty():
            tail = newNode
        else:
            head.prev = newNode
        head = newNode
        size++

    pushBack(element):
        newNode = new Node(element, tail, null)
        if isEmpty():
            head = newNode
        else:
            tail.next = newNode
        tail = newNode
        size++
```

```
popFront():
    if isEmpty():
        throw EmptyDequeException
    element = head.data
    head = head.next
    if head is null:
        tail = null
    else:
        head.prev = null
    size--
    return element

popBack():
    if isEmpty():
        throw EmptyDequeException
    element = tail.data
    tail = tail.prev
    if tail is null:
        head = null
    else:
        tail.next = null
    size--
    return element

isEmpty():
    return size == 0

getSize():
    return size
```

This implementation uses a doubly linked list, where each node contains the element, a pointer to the previous node, and a pointer to the next node. The deque maintains pointers to both the head and tail of the list, allowing constant-time access to both ends.

All four main operations (pushFront, pushBack, popFront, and popBack) have a time complexity of O(1) because they only involve updating pointers at the ends of the list. The isEmpty and getSize operations are also O(1) as they simply check or return the size variable.