Design & Analysis of Algorithms

Pranav Umakant Rajar 1001965075

HW-9

11.2-2) We need to use the following steps:-

- $k = 5$, $h(5) = 5 \bmod 9 = 5$
  Inserted in slot #5
- $k = 28$,
  Inserted in slot #1
- $k = 19$, $h(19) = 19 \bmod 9 = 1$
  Collision in slot #1, chained after 28.
- $k = 15$, $h(15) = 15 \bmod 9 = 6$
  Inserted in slot #6
- $k = 20$, $h(20) = 20 \bmod 9 = 2$
  Inserted in slot #2
- $k = 33$, $h(33) = 33 \bmod 9 = 6$
  Collision at slot 6, chained after 15
- $k = 12$, $h(12) = 12 \bmod 9 = 3$
  Inserted in slot #3
- $k = 17$, $h(17) = 17 \bmod 9 = 8$
  Inserted in slot #8
- $k = 10$, $h(10) = 10 \bmod 9 = 1$
  Collision in slot 1, chained after 19

Final Structure

- 3 keys - 28, 19, 10 are chained in slot #1
- 2 keys - 15, 33, are chained in slot #6
- All other slot are either empty or have 1 element each.

11.3-1) With hash values stored alongside keys, we can optimize in the following manner:-

```
def search_with_hash(linked_list, target_key):
    target_hash = hash_function(target key)
    current = linked_list.head
    while current:
        if current.hash == target_hash:
            if current_key == target_key:
                return current
        current = current.next
    return None
```

This method allows for O(1) comparisons instead of O(n) comparisons.

We only need to compare the actual strings when the hash values match. And good hash functions make it so that collisions are rare.

TC of this method: O(1) best case

O(n*m) worst case,

where n = length of linked list and m = length of strings being used as keys.

O(n) avg. case.

12.4-1) Using linear probing :-

- 10 : slot 10
- 22 : slot 9 (after probe)
- 31 : slot 8 (after probe)
- 4 : slot 4
- 15 : slot 5 (after probe)
- 28 : slot 6 (after probe)
- 17 : slot 7 (after probe)
- 88 : slot 0 (after probe)
- 59 : slot 1 (after probe)