# Assignment 5 - Set C: Theoretical Questions and Answers

## Hashing - Time Complexity and Collision Resolution

---

## Question 1

**Calculate the time complexity of hash table with Linear Probing.**

### Answer

The time complexity of hash table operations with linear probing depends on the **load factor (α)**.

```
Load Factor (α) = n/m
where:
```

```
n = number of elements in the hash table
m = size of the hash table
```

---

## 1. Best Case (No Collisions)

**Insert:** O(1)

- Hash function computation: O(1)
- Direct insertion at computed index
- No probing needed

**Search:** O(1)

- Hash function computation: O(1)
- Element found at first location

**Delete:** O(1)

- Search the element: O(1)
- Mark as deleted: O(1)

---

## 2. Average Case

For a load factor **α < 1:**

| Operation | Time Complexity |
|---|---|

**Explanation:**

- Expected number of probes = 1 / (1 - α)
- As table fills up, more collisions occur
- Performance degrades with higher load factor

---

## 3. Worst Case (All Collisions)

**Insert:** O(n)

- All elements hash to same cluster
- Must probe through entire cluster

**Search:** O(n)

- Element at end of long probe sequence
- Must check all positions

**Delete:** O(n)

- Must search through entire table

---

## 4. Analysis by Load Factor

| Load Factor (α) | Avg Probes (Insert/Search) |
| --- | --- |
| 0.25 | ~1.33 |
| 0.50 | ~2.00 |
| 0.75 | ~4.00 |
| 0.90 | ~10.00 |
| 0.99 | ~100.00 |

### Formulas for Average Probes

**Successful Search:**

```
Average Probes = (1/2) × (1 + 1/(1-α))
```

**Unsuccessful Search:**

```
Average Probes = (1/2) × (1 + 1/(1-α)²)
```

---

## 5. Primary Clustering Problem

Linear probing suffers from **primary clustering**:

- Consecutive occupied slots form clusters
- Clusters grow longer as table fills
- New insertions more likely to collide with clusters
- Performance degrades with higher load factors

### Example of Clustering

```
Table: [12, 13, 14, EMPTY, EMPTY, 15, 16, 17, EMPTY, EMPTY]
       ↑-----cluster-----↑            ↑---cluster---↑
```

---

## 6. Practical Time Complexity

### Recommended Load Factor: α ≤ 0.5

- Maintains near O(1) performance
- Insert/Search/Delete: **O(1) average case**
- Wastes 50% space for better performance

### High Load Factor (α > 0.8)

- Severe performance degradation
- Operations approach **O(n)**
- Not recommended in practice

---

## 7. Comparison with Other Methods

| Method | Best | Average | Worst |
|---|---|---|---|
| Separate Chaining | O(1) | O(1 + α) | O(n) |

---

## 8. Space Complexity

**Space:** O(m) where m is table size

- Fixed size array
- No additional pointers (unlike chaining)
- Memory efficient

---

## Time Complexity Summary

| Scenario | Time Complexity |
|---|---|
| Best Case | O(1) |
| Average Case | O(1) when $\alpha<0.5$ |
| Worst Case | O(n) |

### For Practical Use

- Keep load factor below **0.5-0.7**
- Resize table when load factor exceeds threshold
- Use good hash function to minimize collisions

**Performance degrades significantly as table fills up due to primary clustering.**

---

## Question 2

**The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function h(k) = k mod 10 and linear probing. What is the resultant hash table?**

### Answer

**Given:**

- Hash Function: `h(k) = k mod 10`
- Table Size: 10 (indices 0-9)
- Collision Resolution: Linear Probing

**Initial Table:**

```
Index: 0  1  2  3  4  5  6  7  8  9
Value: -  -  -  -  -  -  -  -  -  -
```

---

# Step-by-Step Insertion

### 1. Insert 12

```
h(12) = 12 mod 10 = 2
Index 2 is empty → Insert at index 2

Table: -   -   12   -   -   -   -   -   -   -
       0   1    2   3   4   5   6   7   8   9
```

### 2. Insert 18

```
h(18) = 18 mod 10 = 8
Index 8 is empty → Insert at index 8

Table: -   -   12   -   -   -   -   -   18   -
       0   1    2   3   4   5   6   7    8   9
```

### 3. Insert 13

```
h(13) = 13 mod 10 = 3
Index 3 is empty → Insert at index 3

Table: -   -  12  13  -  -  -  -  18  -
       0   1   2   3  4  5  6  7   8  9
```

#### 4. Insert 2

```
h(2) = 2 mod 10 = 2
Index 2 is occupied (collision!)
Linear probe: Try index 3 → occupied
Linear probe: Try index 4 → empty → Insert at index 4

Table: -   -  12  13  2  -  -  -  18  -
       0   1   2   3  4  5  6  7   8  9
```

#### 5. Insert 3

```
h(3) = 3 mod 10 = 3
Index 3 is occupied (collision!)
Linear probe: Try index 4 → occupied
Linear probe: Try index 5 → empty → Insert at index 5

Table: -   -  12  13  2  3  -  -  18  -
       0   1   2   3  4  5  6  7   8  9
```

### 6. Insert 23

```
h(23) = 23 mod 10 = 3
Index 3 is occupied (collision!)
Linear probe: Try index 4 → occupied
Linear probe: Try index 5 → occupied
Linear probe: Try index 6 → empty → Insert at index 6

Table: -   -  12  13  2  3  23  -  18  -
       0   1   2   3  4  5   6  7   8  9
```

### 7. Insert 5

```
h(5) = 5 mod 10 = 5
Index 5 is occupied (collision!)
Linear probe: Try index 6 → occupied
Linear probe: Try index 7 → empty → Insert at index 7

Table: -   -  12  13  2  3  23  5  18  -
       0   1   2   3  4  5   6  7   8  9
```

### 8. Insert 15

```
h(15) = 15 mod 10 = 5
Index 5 is occupied (collision!)
Linear probe: Try index 6 → occupied
Linear probe: Try index 7 → occupied
Linear probe: Try index 8 → occupied
Linear probe: Try index 9 → empty → Insert at index 9

Table: -   -   12  13  2  3  23  5  18  15
       0   1   2   3   4  5  6   7  8   9
```

---

## Final Hash Table

```
┌───────┬─────┬──────────────────┐
│ Index │ Key │ Number of Probes │
├───────┼─────┼──────────────────┤
│   0   │  -  │        -         │
│   1   │  -  │        -         │
│   2   │ 12  │  1 (no collision)│
│   3   │ 13  │  1 (no collision)│
│   4   │  2  │  3 (at 2, 3, 4)  │
│   5   │  3  │  3 (at 3, 4, 5)  │
│   6   │ 23  │  4 (at 3,4,5,6)  │
│   7   │  5  │  3 (at 5, 6, 7)  │
│   8   │ 18  │  1 (no collision)│
│   9   │ 15  │  5 (at 5,6,7,8,9)│
└───────┴─────┴──────────────────┘
```

---

## Final Result

**Position:** `[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]`

**Value:** `- - 12 13 2 3 23 5 18 15`

---

## Analysis

**Total Collisions:** 7

**Load Factor:** 8/10 = 0.8 (80% full)

**Average Probes:** (1+1+3+3+4+3+1+5) / 8 = **2.625**

### Observations

1. **Primary clustering** visible at indices 2-7
2. **High load factor** leads to more collisions
3. Key **15** required **5 probes** (worst case in this example)

4. Keys **12, 13, 18** had no collisions (lucky hash values)

---

## Question 3

**Which data structure is appropriate for simple chaining? Why?**

### Answer

**Linked List** is the most appropriate data structure for simple chaining (separate chaining).

---

## Reasons Why Linked List is Best

### 1. Dynamic Size

**Advantage:**

- Can grow/shrink dynamically as elements are inserted/deleted
- No need to know collision chain length in advance

- No fixed size limitations

**Example:**

```
hash[5]: 15 → 25 → 35 → 45 → NULL
Can keep adding elements to chain without worrying about space
```

**Alternative (Array):** Would need to pre-allocate fixed size, wasting space or risking overflow.

---

### 2. Efficient Insertion

**Time Complexity:** O(1)

- Insert new node at beginning of chain
- Only requires updating pointers
- No shifting of elements needed

**Code:**

```
newNode->next = hashTable[index];
hashTable[index] = newNode;
```

**Alternative (Array):** O(n) to shift elements or find empty slot

---

### 3. Efficient Deletion

**Time Complexity:** O(1) after finding element

- Simply update pointers to bypass deleted node
- No gaps or reorganization needed

**Code:**

```
prev->next = current->next;
free(current);
```

**Alternative (Array):** O(n) to shift elements and fill gap

---

### 4. Memory Efficiency

**Advantage:**

- Uses exactly the memory needed
- No wasted space for empty slots

- Grows only when necessary

**Memory Usage:**

- Per node: key + next pointer
- Only allocates for actual elements, not potential maximum

**Alternative (Array):** Must pre-allocate maximum possible size

---

### 5. No Overflow Issues

**Advantage:**

- Chains can grow indefinitely (within system memory)
- No "chain full" scenario
- Handles heavy collisions gracefully

**Example:**

Even if 100 keys hash to same index, linked list can accommodate all

**Alternative (Array):** Fixed size, will overflow

---

#### 6. Easy Traversal

**Advantage:**

- Simple sequential traversal with next pointer
- Can easily iterate through all elements in chain

**Code:**

```
while (temp != NULL) {
    process(temp->key);
    temp = temp->next;
}
```

---

#### 7. Flexibility

**Advantage:**

- Can store additional information (like frequency, timestamps)
- Can implement sorted chains for faster search
- Can use doubly linked list for faster deletion

**Enhanced Node:**

```
struct Node {
```

```
    int key;
    int value;
    int frequency;
    struct Node *next;
};
```

---

## Comparison with Other Data Structures

| Data Structure | Advantages | Disadvantages |
| --- | --- | --- |
| **Linked List** | Dynamic, O(1) insert, | O(n) search, |
| **(BEST CHOICE)** | no overflow, memory efficient | pointer overhead |
| Array | Cache friendly, random access | Fixed size, overflow |
| BST | O(log n) search | Complex, may skew |
| Dynamic Array | Resizable | Costly resize |

---

## Implementation

```c
typedef struct Node {
    int key;
    struct Node* next;
} Node;

Node* hashTable[TABLE_SIZE];

// Insert at beginning (O(1))
void insert(int key) {
    int index = hash(key);
    Node* newNode = createNode(key);
    newNode->next = hashTable[index];
    hashTable[index] = newNode;
}

// Search in chain (O(chain length))
int search(int key) {
    int index = hash(key);
    Node* temp = hashTable[index];
    while (temp != NULL) {
        if (temp->key == key) return 1;
        temp = temp->next;
    }
    return 0;
}
```

## Practical Considerations

### 1. Cache Performance

- Arrays are better for cache locality
- But linked lists acceptable for moderate chain lengths

### 2. Load Factor

- Keep **$\alpha = n/m < 1$** for good performance
- Average chain length = $\alpha$
- With $\alpha < 1$, search time ≈ **$O(1 + \alpha)$**

### 3. Hybrid Approach

- Use array initially for small chains
- Convert to linked list when chain grows
- Best of both worlds

---

## Conclusion

Linked List is the standard and most appropriate choice for separate chaining because:

- ✓ **Dynamic size** (no overflow)
- ✓ **O(1) insertion**
- ✓ **Memory efficient**
- ✓ **Simple implementation**

- ✓ **Flexible and extensible**
- ✓ **Handles collisions gracefully**

The slight pointer overhead and cache performance cost are outweighed by the flexibility and simplicity it provides. This is why virtually all implementations of hash tables with chaining use linked lists.

---