Lecture Notes: Hashing

Introduction to Hashing

Hashing is a technique used to map data to a fixed-size value or key, called a hash code that represents the original data. It enables **constant time complexity O(1)** for insertions, deletions, and lookups in the average case, making it one of the most efficient methods for managing data in many applications. Hashing converts an input of arbitrary size into a fixed-size output, allowing rapid access to data.

Applications of Hashing

- **Hash Tables**: Efficient data retrieval structures used for dictionaries, sets, and caches
- **Cryptography**: Ensures data integrity and security
- **Databases**: Indexing and searching
- **File Systems**: Managing blocks of storage efficiently

Key Components of Hashing

1. **Hash Function**:
   - A function that takes an input and returns a hash code.
   - Characteristics of a good hash function:
     - Fast to compute.
     - Uniformly distributes keys across the hash table.
     - Minimizes collisions (different inputs yielding the same hash value).
2. **Hash Table**:
   - An array-like data structure where the hash value determines the index.
3. **Collisions**:
   - Occur when multiple keys map to the same hash value.
   - Collision resolution strategies are crucial for maintaining constant time complexity.

Types of Hashing

1. Closed Hashing (Open Addressing)

- All elements are stored within the hash table itself.
- When a collision occurs, the algorithm searches for the next available slot.

Techniques for Open Addressing:

a) Linear Probing:

- Search sequentially for the next available slot.

Example:

Given keys: `49, 63, 56, 52`, and a hash table of size 7 with `h(k) = k % 7`:

Index	Values
0	56
1	63
2	52
3	49
4	
5	
6	

Pros:

- Does not require additional memory.
- Maintains O(1) average time complexity for search and insertion under low load factors.

Cons:

- Clustering of values reduces efficiency
- Performance degrades as the table fills

b) Quadratic Probing:

- Probes using a quadratic formula: `h'(k, i) = (h(k) + i^2) % size`.

Example:

Keys: `10, 22, 31, 4, 15`, and a hash table of size 7:

Index	Values
0	22
1	
2	4
3	31
4	15
5	
6	10

c) Double Hashing:

- Uses two hash functions: `h1(k) and h2(k)`.
- Probe sequence: `h'(k, i) = (h1(k) + i * h2(k)) % size`.

Example:

Keys: 50, 700, 76, 85, 92, and hash functions:

- `h1(k) = k % 7`
- `h2(k) = 5 - (k % 5)`

Index	Values
0	85
1	92
2	700
3	50
4	76
5	
6	

2. Open Hashing (Separate Chaining)

- Each index in the hash table points to a linked list (or other data structure) that stores all elements mapping to that index.

Example:

Given keys: `15, 11, 27, 8, 12`, and a hash table of size 7 with a hash function `h(k) = k % 7`:

Index	Values
0	
1	8
2	15, 12
3	
4	11
5	
6	27

Example Code (C++):

```
#include <iostream>
#include <vector>
#include <list>
using namespace std;

class HashTable {
    vector<list<int>> table;
    int size;

public:
    HashTable(int s) : size(s) {
        table.resize(s);
    }

    int hashFunction(int key) {
        return key % size;
    }
    void insert(int key) {
        int index = hashFunction(key);
        table[index].push_back(key);
    }
    void display() {
        for (int i = 0; i < size; i++) {
            cout << i << ": ";
            for (int x : table[i])
                cout << x << " -> ";
            cout << "NULL" << endl;
        }
    }
};

int main() {
    HashTable ht(7);
    ht.insert(15);
    ht.insert(11);
    ht.insert(27);
    ht.insert(8);
    ht.insert(12);
    ht.display();
    return 0;
}
```

Choosing Between Open and Closed Hashing

- Use **Open Hashing** when:
    - Frequent insertions and deletions are expected.
    - Memory usage is not a constraint.
- Use **Closed Hashing** when:
    - Memory is limited.
    - Cache performance is critical.

Summary

Hashing is a powerful technique for efficient data storage and retrieval. With its average-case **constant time complexity (O(1))**, it is a cornerstone of modern computing. By understanding and implementing various hashing strategies, such as open and closed hashing, you can design systems that balance performance and memory constraints effectively.