Chapter 3.4: Hash Tables

Chapter 3.4: Hash Tables

Introduction

A hash table is a data structure that provides efficient insertion, deletion, and search operations. It uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

Hash Function

A hash function maps keys to indices in a hash table. A good hash function minimizes collisions and uniformly distributes keys.

Properties of a Good Hash Function

1. **Deterministic:** Same input yields the same output.
2. **Efficiently Computable:** Should be quick to compute.
3. **Uniform Distribution:** Keys should be uniformly distributed across the hash table.

Collision Resolution

When two keys hash to the same index, a collision occurs. Two main strategies for collision resolution are:

Separate Chaining

Each bucket in the hash table is a linked list. When a collision occurs, the new key is added to the linked list at the hashed index.

Example: Separate Chaining Implementation

```java
public class SeparateChainingHashST<Key, Value> {
    private int M = 97; // number of chains
    private Node[] st = new Node[M];

    private static class Node {
        private Object key;
        private Object val;
        private Node next;

        public Node(Object key, Object val, Node next) {
            this.key = key;
            this.val = val;
            this.next = next;
        }
    }

    private int hash(Key key) {
        return (key.hashCode() & 0x7fffffff) % M;
    }

    public Value get(Key key) {
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next) {
            if (key.equals(x.key)) return (Value) x.val;
        }
        return null;
```

```
    }

    public void put(Key key, Value val) {
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next) {
            if (key.equals(x.key)) {
                x.val = val;
                return;
            }
        }
        st[i] = new Node(key, val, st[i]);
    }
}
```

Linear Probing

Linear probing is an open addressing scheme where, upon collision, we linearly probe for the next available slot.

Example: Linear Probing Implementation

```java
public class LinearProbingHashST<Key, Value> {
    private int M = 30001;
    private Value[] vals = (Value[]) new Object[M];
    private Key[] keys = (Key[]) new Object[M];

    private int hash(Key key) {
```

```
        return (key.hashCode() & 0x7fffffff) % M;
    }

    public void put(Key key, Value val) {
        int i;
        for (i = hash(key); keys[i] != null; i = (i + 1) % M) {
            if (keys[i].equals(key)) {
                vals[i] = val;
                return;
            }
        }
        keys[i] = key;
        vals[i] = val;
    }

    public Value get(Key key) {
        for (int i = hash(key); keys[i] != null; i = (i + 1) % M) {
            if (keys[i].equals(key)) return vals[i];
        }
        return null;
    }
}
```

Load Factor

The load factor is the ratio of the number of keys to the size of the hash table. A high load factor means more collisions, while a low load factor means wasted space.

Resizing

To maintain an efficient load factor, hash tables are resized (usually doubled in size) when the load factor exceeds a certain threshold.

Performance

- **Search, Insertion, Deletion:** Average case O(1), worst case O(N) due to collisions.
- **Resizing:** O(N) due to rehashing all keys.

Applications

1. **Database Indexing:** Quick access to records.
2. **Caching:** Fast retrieval of previously computed results.
3. **Symbol Tables:** Efficiently manage a large set of keys.

Example Usage

```java
public class HashTableExample {
    public static void main(String[] args) {
        SeparateChainingHashST<String, Integer> st = new SeparateChainingHashST<>();
        st.put("apple", 1);
        st.put("banana", 2);
        st.put("cherry", 3);

        System.out.println("Value for 'apple': " + st.get("apple")); // Output: 1
        System.out.println("Value for 'banana': " + st.get("banana")); // Output: 2
    }
}
```

```

## Conclusion

Hash tables are a powerful data structure that provide efficient average-case performance for dynamic set operations. By using effective hash functions and collision resolution strategies, they ensure fast access to data.
```