KAZAKH KBTU BRITISH
TECHNICAL
UNIVERSITY

LECTURE 8 – HASH TABLES & RABIN-KARP ALGORITHM

Prepared by: Izbassar Assylzhan

Course: Algorithms & Data Structures – Fall 2025

LECTURE 8

In this lecture, we'll cover some of the efficient data structures – **hash tables** with the methods as *searching*, *insertion*, and *deletion*. Then using the idea of hashing, we will try to implement the **Rabin-Karp algorithm**, for finding the patterns inside the text.

TOPICS WE'LL COVER:

- Hash Tables
- Collisions & Handling them
- Rehashing / Resizing
- Rabin-Karp algorithm
- Practice Problems

GOALS FOR THIS LECTURE:

- Understand the Concept and Structure of Hash Tables
- Implement and Analyze Collision Handling Techniques
- Apply Hashing in String Matching with the Rabin–Karp Algorithm

HASH TABLES

KAZAKH BRITISH TECHNICAL UNIVERSITY

- Hash Tables
- Collisions & Handling them
- Rehashing / Resizing
- Rabin-Karp algorithm
- Practice Problems

Hashing is a technique used to map data (keys) to a fixed-size value (index) using a hash function.

- The hash function takes an input (key) and returns an integer (hash code);
- That hash code is then used to determine where to store the data in an array — called a **hash table**.

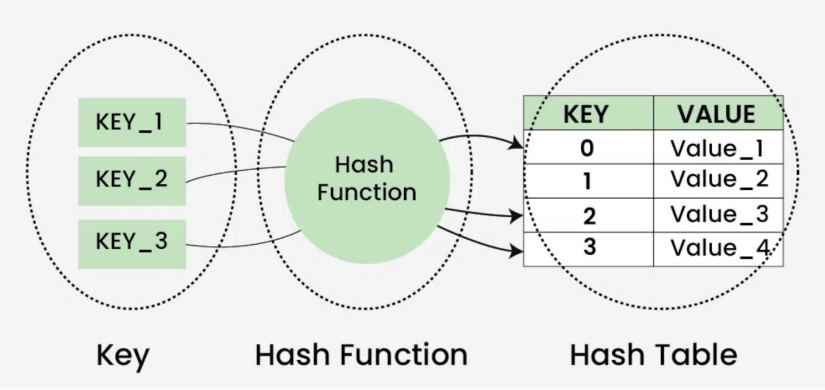

Figure 1 – Structure of Hash Table storage [1].

Components of Hashing:

- **Key** – the data you want to store or find (e.g., a word, ID);
- **Hash Function** – maps the key to an integer index;
- **Hash Table** – an array where elements are stored.

[1] https://www.geeksforgeeks.org/dsa/hash-table-data-structure/

HASH TABLES (CONT.)

- Hash Tables
- Collisions & Handling them
- Rehashing / Resizing
- Rabin-Karp algorithm
- Practice Problems

For implement the hash table, we can use vector, and linked list, then determine the hash function for mapping the keys into indices as shown in Figure 2.

Every hash table should support these basic methods:

- *insert(key, value)* — add a new key-value pair.
- *search(key)* — find a value by key.
- *remove(key)* — delete a key-value pair.

```
class HashTable {
private:
    int capacity;
    vector<list<pair<int, string>>> table;
    list<pair<int, string>>::iterator it;

    int hash(int key) {
        return key % capacity;
    }
```

Figure 2 – Representation of the Hash Table.

Bucket array → *stores the actual key-value pairs (or linked lists for chaining).*

COLLISIONS & HANDLING THEM

KAZAKH KBTU BRITISH TECHNICAL UNIVERSITY

- Hash Tables
- Collisions & Handling them
- Rehashing / Resizing
- Rabin-Karp algorithm
- Practice Problems

When two keys map to the same index, we have two methods for handling them:

1. **Separate Chaining.** When we store multiple elements in the same bucket using a linked list or vector.
2. **Open Addressing** - allows us to find another empty slot within the array using:

- Linear probing: check next index → index + 1
- Quadratic probing: check index + i^2
- Double hashing: use a second hash function.

```
int start = index;
while (!table[index].empty()) {
    index = (index + 1) % capacity;
    if (index == start) {
        cout << "Hash table full!\n";
        return;
    }
}
```

Figure 3 – Linear probing of the key.

REHASHING / RESIZING

- Hash Tables
- Collisions & Handling them
- Rehashing / Resizing
- Rabin-Karp algorithm
- Practice Problems

The load factor (α), as shown (1) indicates how full the table is:

$$\alpha = \frac{number\ of\ elements}{number\ of\ buckets}, \quad (1)$$

where α is the load factor. Typically, threshold less than 0,75.

When the load factor exceeds a threshold:

- Increase capacity (usually double it).
- Recompute hashes for all existing keys and insert them again.

```
void rehash() {
    int oldSize = table.size();
    int newSize = oldSize * 2; // resize

    vector<list<pair<int, string>>> newTable(newSize);

    for (vit=table.begin(); vit!=table.end(); ++vit) {
        for (it=(*vit).begin(); it!=(*vit).end(); ++it) {
            int newIndex = (*it).first % newSize;
            newTable[newIndex].push_back(make_pair((*it).first, (*it).second));
        }
    }
    table = std::move(newTable);
}
```

Figure 4 – Rehashing & resizing the hash table.

RABIN-KARP ALGORITHM

- Hash Tables
- Collisions & Handling them
- Rehashing / Resizing
- Rabin-Karp algorithm
- Practice Problems

The Rabin–Karp algorithm is a string-matching algorithm used to find occurrences of a pattern within a larger text efficiently.

Given:
Text = 315265,
Pattern = 26, and
Base = 11;

P mod **Base** = **26** mod **11** = **4**

After calculating the hash value of the pattern, we can check it with every sub-hash of length of pattern in the given text, and search it.

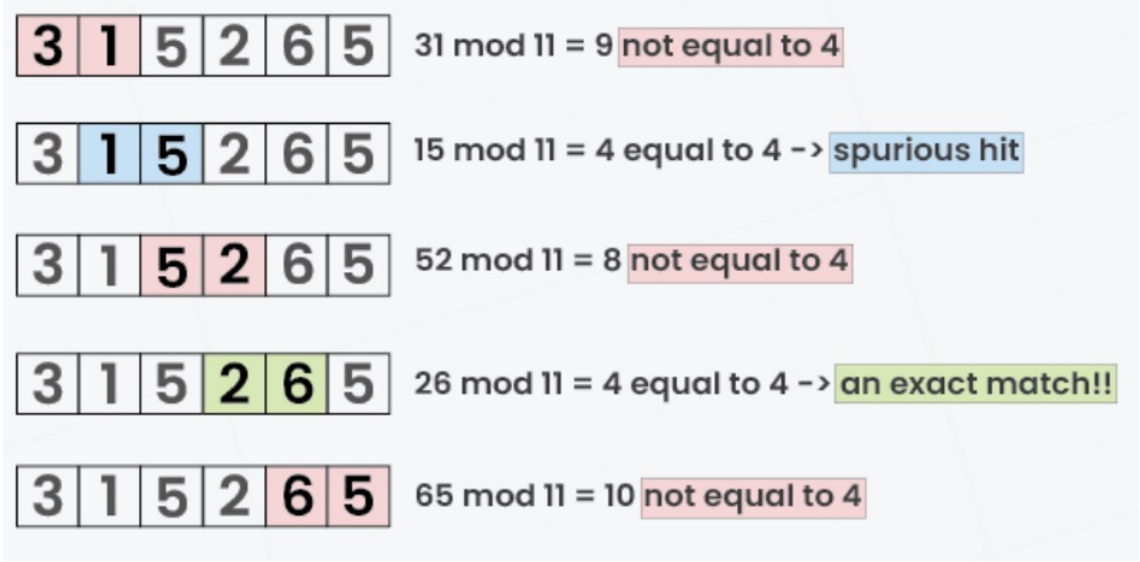

Figure 5 – Pattern matching representation [2].

[2] https://www.geeksforgeeks.org/cpp/implementation-of-rabin-karp-algorithm-in-cpp/

RABIN-KARP ALGORITHM (CONT.)

KAZAKH KBTU BRITISH TECHNICAL UNIVERSITY

As we already noticed, simple hashing can lead some collisions, and our goal is to **calculate unique hash linked to every character of the string**. For that case we are using the expression as in formula below (2).

$$hash_{string} = \sum_{i=0}^{n-1} s_i * p^{(n-1)-i} \mod m \quad (2)$$

where s_i - character of the string, p – base of prime number, n – length of the string, m – some big number. Then, Rabin-Karp algorithm will be:

Step 1: Initializing the parameters: p and m**;**

Step 2: Pre-calculating the powers of prime number p;

Step 3: Calculating the $hash_{i+1}$ of given text / string;

Step 4: Calculating the $hash_{pattern}$;

Step 5: Match the $hash_{i+n} - hash_i$ with $hash_{pattern} \bmod \mathrm{m}$, and if equal save position of i.

RABIN-KARP ALGORITHM (CONT.)

KAZAKH BRITISH TECHNICAL UNIVERSITY

- Hash Tables
- Collisions & Handling them
- Rehashing / Resizing
- Rabin-Karp algorithm
- Practice Problems

Step 1: Initializing the parameters: p and m:

```
const int p = 31;
const int mod = 1e9 + 7;

int n = s.size();
int m = t.size();
```

Step 2: Pre-calculating the powers of prime number p:

```
vector<long long> p_pow(max(n, m));
p_pow[0] = 1;
for (int i=1; i < (int) p_pow.size(); ++i) {
    p_pow[i] = (p_pow[i-1]*p) % mod;
}
```

Step 3: Calculating the $hash_{i+1}$ of given text / string;

```
vector<long long> h(m + 1, 0);
for (int i=0; i < m; ++i) {
    h[i+1] = (h[i] + (t[i] - 'a' + 1) * p_pow[i]) % mod;
}
```

Step 4: Calculating the $hash_{pattern}$;

```
long long h_s = 0;
for (int i=0; i < n; ++i) {
    h_s = (h_s + (s[i] - 'a' + 1) * p_pow[i]) % mod;
}
```

Step 5: Match the $hash_{i+n} - hash_i$ with $hash_{pattern} \bmod m$, and if equal save position of i:

```
vector<int> occurrence;
for(int i=0; i + n - 1 < m; ++i){
    long long cur_h = (h[i+n] - h[i] + mod) % mod;
    if (cur_h == (h_s * p_pow[i]) % mod) {
        occurrence.push_back(i);
    }
}
return occurrence;
```

KAZAKH KBTU BRITISH TECHNICAL UNIVERSITY

PRACTICE PROBLEMS

- Hash Tables
- Collisions & Handling them
- Rehashing / Resizing
- Rabin-Karp algorithm
- Practice Problems

We have two tasks to solve:

- 1. Two Sum;
- 796. Rotate String;
- 686. Repeated String Match.

Q & A