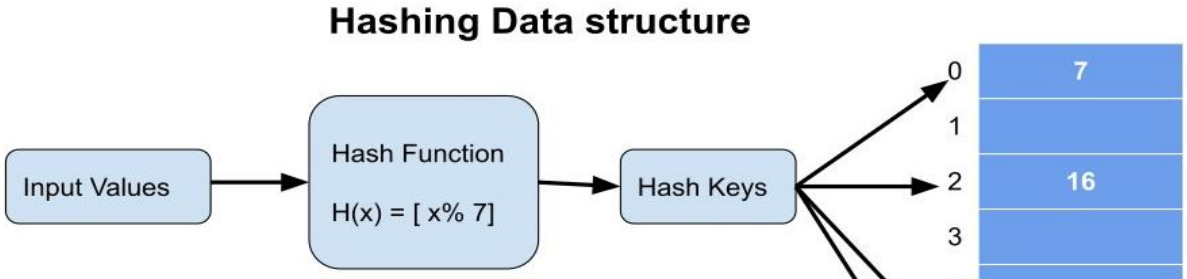
CL2001 Data Structures Lab [DS]	Lab 11 Hashing and its types, String Algorithms

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Fall 2025

Hashing

Hashing refers to the process of generating a fixed-size output from an input of variable size using the mathematical formulas known as hash functions. This technique determines an index or location for the storage of an item in a data structure.



Inputs	Hash Function	Hash Keys
7	=7 % 7	0
16	=16 % 7	2
33	=33 % 7	5
42	=42 % 7	6

Components of Hashing:

There are majorly three components of hashing:

1. **Key:** A Key can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
2. **Hash Function:** The hash function receives the input key and returns the index of an element in an array called a hash table. The index is known as the hash index.
3. **Hash Table:** Hash table is a data structure that maps keys to values using a special function called a hash function. Hash stores the data in an associative manner in an array where each data value has its own unique index.

Inserting In A Hash Table:

1. **Choose a Hash Function:** The first step is selecting or designing a hash function suitable for the data and the hash table size. The function should map input keys to indices within the range of the hash table size, ensuring uniform distribution.

   H(key) = key % sizeOfHashTable

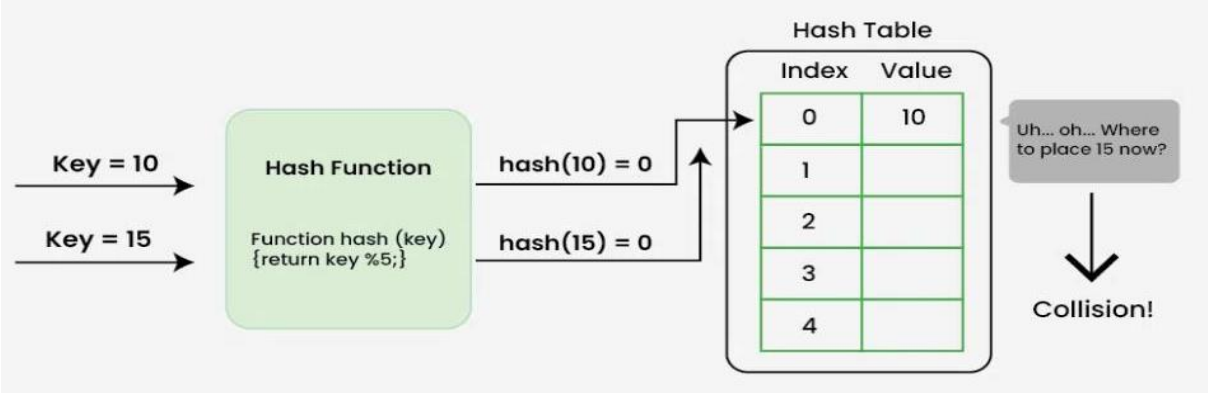
   The hash index must be within the range of hash table size, so the key is usually taken modulo the table size to produce a valid index.
2. **Calculate Hash Code:** For a given key, apply the hash function to generate an index where that key will be inserted in the hash table.

3. **Insert Data:**

i) Calculate the index using the hash function.

ii) Check if the computed index in the hash table is empty o If it's empty, place the key at that index. o If it's occupied (collision occurs), resolve the collision using a chosen method:

- **Separate Chaining:** Add the key-value pair to a linked list at that index.
- **Open Addressing:** Probe for the next available slot based on the probing technique (e.g., linear, quadratic, or double hashing).



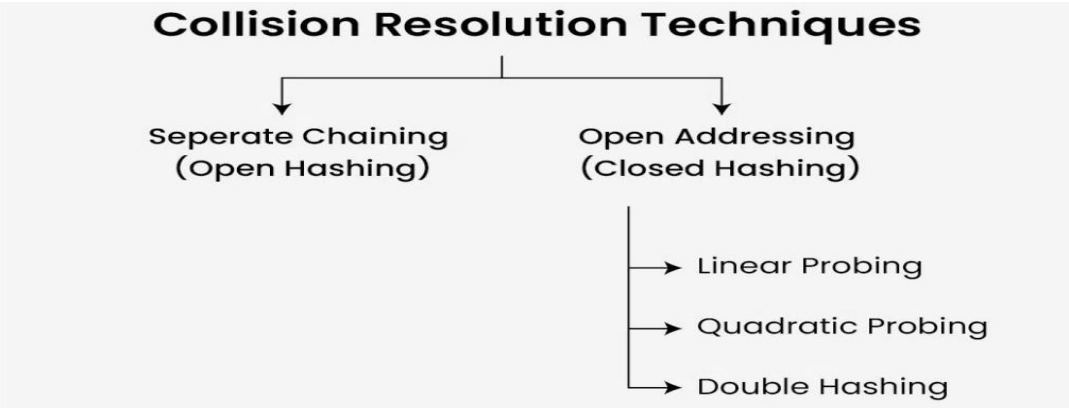
Operations of Chain Hashing:

```
void insert(int key, string value) {
        int index = hashFunction(key);
        Node* newNode = new Node(key, value);

        if (table[index] == nullptr) {
            // No collision
            table[index] = newNode;
        } else {
            // Collision occurred → add to the end of linked list
            Node* temp = table[index];
            while (temp->next != nullptr) {
                if (temp->key == key) {
                    // Update existing key
                    temp->value = value;
                    delete newNode;
                    return;
                }
                temp = temp->next;
            }
            // Check last node
            if (temp->key == key) {
                temp->value = value;
                delete newNode;
                return;
            }
            temp->next = newNode;
        }
    }
```

```
string search(int key) {
        int index = hashFunction(key);
        Node* temp = table[index];

        while (temp != nullptr) {
            if (temp->key == key)
                return temp->value;
            temp = temp->next;
        }
        return "Key not found";
    }
```

```
// Delete a key
void remove(int key) {
    int index = hashFunction(key);
    Node* temp = table[index];
    Node* prev = nullptr;

    while (temp != nullptr && temp->key != key) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == nullptr) {
        cout << "Key " << key << " not found!\n";
        return;
    }

    if (prev == nullptr)
        table[index] = temp->next; // Remove head node
    else
        prev->next = temp->next;   // Remove middle or last node

    delete temp;
    cout << "Key " << key << " deleted successfully.\n";
}
```

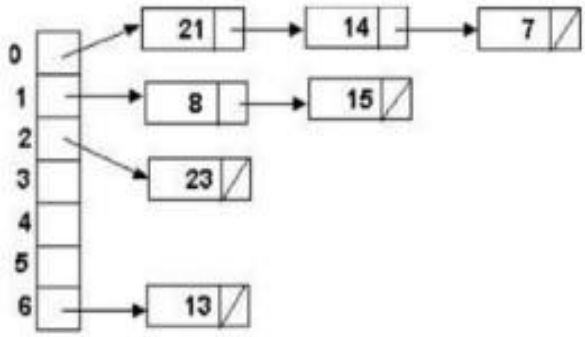
COLLISION RESOLUTION



1. <u>SEPARATE CHAINING:</u> This method involves making a linked list out of the slot where the collision happened, then adding the new key to the list.

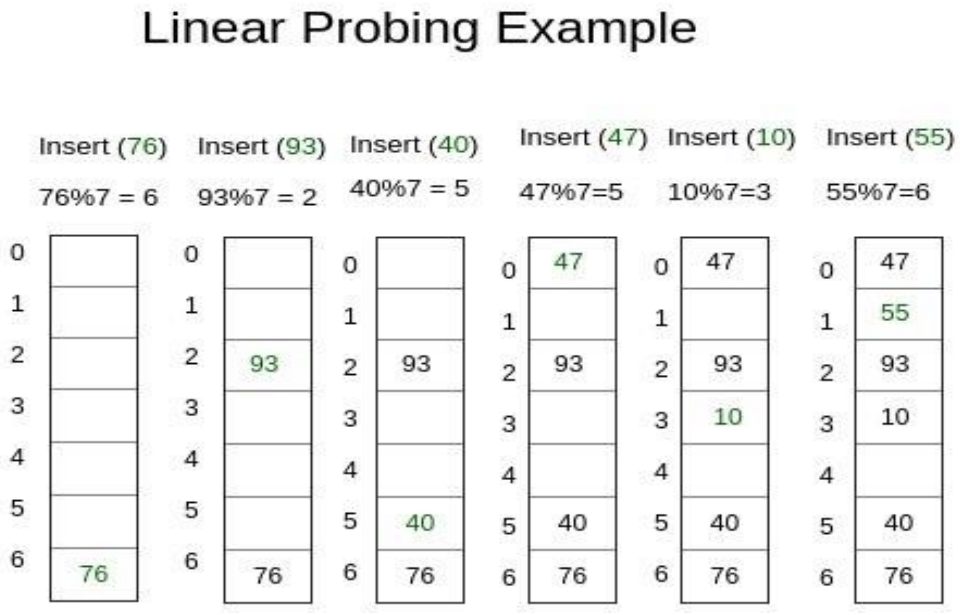
- **Time complexity:** Its worst-case complexity for *searching* and *deletion* is o(n).
- The hash table never fills full, so we can add more elements to the chain.
- It requires more space for element links.

h(23) = 23 % 7 = 2
h(13) = 13 % 7 = 6
h(21) = 21 % 7 = 0
h(14) = 14 % 7 = 0 collision
h(7) = 7 % 7 = 0 collision
h(8) = 8 % 7 = 1
h(15) = 15 % 7 = 1 collision



2. <u>OPEN ADDRESSING:</u> To prevent collisions in the hashing table, open addressing is employed as a collision-resolution technique. No key is kept anywhere else besides the hash table. As a result, the hash table's size is never equal to or less than the number of keys. (Note that we can increase table size by copying old data if needed). Additionally, known as closed hashing.

a) <u>Linear probing</u>: In linear probing, if a collision occurs at an index i, the algorithm checks the next slot (i + 1) % table_size, then (i + 2) % table_size, and so on until an empty slot is found. This technique often leads to **clustering**, where contiguous blocks of filled slots are formed, which can slow down search times



```
// Insert key-value pair
  void insert(int key, string value) {
      int index = hashFunction(key);
      int startIndex = index; // To detect full table

      // Linear probing for empty slot
      while (occupied[index]) {
          if (keys[index] == key) {
              // Update existing key
              values[index] = value;
              cout << "Updated key " << key << " with new value.\n";
              return;
          }
          index = (index + 1) % SIZE;
          if (index == startIndex) {
              cout << "Hash table is full! Cannot insert.\n";
              return;
          }
      }

      // Insert new key-value pair
      keys[index] = key;
      values[index] = value;
      occupied[index] = true;
  }
```

```
    // Delete a key
    void remove(int key) {
        int index = hashFunction(key);
        int startIndex = index;

        while (occupied[index]) {
            if (keys[index] == key) {
                occupied[index] = false;
                cout << "Key " << key << " deleted.\n";
                return;
            }
            index = (index + 1) % SIZE;
            if (index == startIndex)
                break;
        }

        cout << "Key " << key << " not found!\n";
    }
```

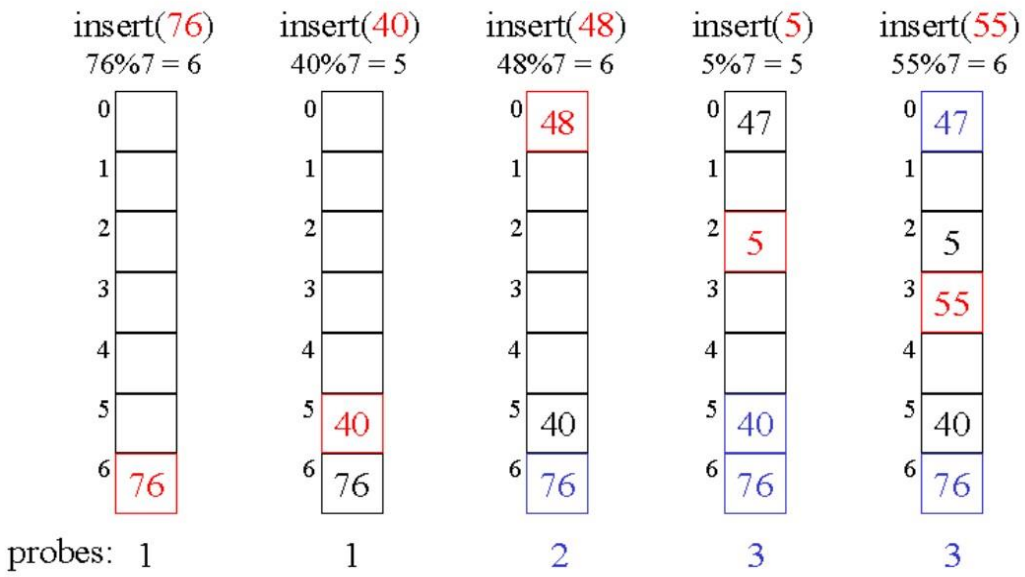
b) **Quadratic probing**: When a collision occurs at index i, the next slots checked are

(i + 1^2) % table_size,
(i + 2^2) % table_size,
(i + 3^2) % table_size.

, and so on.

This spreads out the potential positions, reducing clustering but requiring a well-sized table to ensure all slots can be reached.

Quadratic Probing Example ☺



Double hashing: Double hashing uses a second hash function to calculate the step size for probing. When a collision occurs at index `i`, the next slot is determined by (i + j * hash2(key)) % table_size, where `hash2` is a secondary hash function, and `j` increments with each probe. Double hashing generally provides a good spread across the table and minimizes clustering.

Lets say, Hash1 (key) = key % 13

Hash2 (key) = 7 – (key % 7)

Hash1(19) = 19 % 13 = 6

Hash1(27) = 27 % 13 = 1

Hash1(36) = 36 % 13 = 10

Hash1(10) = 10 % 13 = 10

Hash2(10) = 7 – (10%7) = 4

(Hash1(10) + 1*Hash2(10))%13= 1

(Hash1(10) + 2*Hash2(10))%13= 5

Collision

REHASHING

Rehashing is the process of resizing a hash table and reassigning all the elements to new positions within it. This is done to reduce the load factor, minimize collisions, and improve the performance of hash operations. In essence, rehashing involves creating a new, larger hash table and re-inserting each key-value pair from the old table using a new hash function or the same one adjusted to the new table size.

When is Rehashing Needed?

Rehashing is typically triggered when the load factor of the hash table reaches or exceeds a certain threshold, usually around 0.7 to 0.75. The load factor is defined as:

$$\text{Load Factor} = \frac{\text{Number of Elements in the Table}}{\text{Size of the Table}}$$

A high load factor means there are more elements relative to the number of slots, leading to a higher probability of collisions and therefore longer search times. Rehashing alleviates this by expanding the table size and redistributing elements.

Steps for Rehashing

1. **Calculate the New Table Size** Generally, the new size is a prime number roughly double the current size.
   Prime numbers help in reducing clustering when using hash functions. For instance, if the current table has 10 slots, the new table might have 23 or 29 slots.
2. **Create the New Hash Table** Allocate a new hash table with the updated size.
3. **Rehash All Existing Elements** for each element in the old hash table:
   - Calculate a new index using the hash function and the updated table size.
   - Insert the element at this new index in the new table.
   - o This step can be time-consuming, as every element must be rehashed and inserted into the new table.
4. **Replace the Old Table with the New Table** Once all elements have been rehashed into the new table, replace the old table with the new one. o Update any references to the old table, effectively freeing its memory.

Introduction to String Searching

In C++, strings are sequences of characters stored in a char array. Matching a pattern in a string involves searching for a specific sequence of characters (the pattern) within a given string. Efficient string search algorithms reduce the computational overhead of naive matching methods.

Brute Force Algorithm

A brute force algorithm is a simple, comprehensive search strategy that systematically explores every option until a problem's answer is discovered. It's a generic approach to problem-solving that's employed when the issue is small enough to make an in-depth investigation possible. However, because of their high temporal complexity, brute force techniques are inefficient for large-scale issues.

- **Initialization**: Start at the beginning of the text and slide a "window" (of the pattern's length) over the text.
- **Character Comparison**: Compare each character in the pattern with the corresponding character in the text.
- **Mismatch Handling**:
  - If a mismatch is found, move the pattern window one position to the right and restart the comparison.
- **Match Handling**:
  - If all characters in the pattern match the text within the current window, record the starting position of the match.
  - Slide the pattern window one position to the right to check for more occurrences.
- **Repeat**: Continue the process until the pattern window reaches the end of the text.

```
  NOBODY NOTICED HIM
1 NOT
2  NOT
3   NOT
4    NOT
5     NOT
6      NOT
7       NOT
8        NOT
```

Example:

```
void bruteForceSearch(string text, string pattern) {
    int n = text.length();
    int m = pattern.length();

    for (int i = 0; i <= n - m; i++) {
        int j = 0;
        while (j < m && text[i + j] == pattern[j]) {
            j++;
        }
        if (j == m) { // Match found
            cout << "Pattern found at index " << i << endl;
        }
    }
}
```

Rabin-Karp Algorithm

The Rabin-Karp Algorithm is an algorithm utilized for searching/matching patterns in the text with the help of a hash function. Unlike the Naïve String-Matching algorithm, it doesn't traverse through every character in the initial phase. It filters the characters that do not match and then performs the comparison.

A hash function is a utility to map a larger input value to a smaller output value. This output value is known as the Hash value.

Understanding the Algorithm

Step 1: Choose a suitable base and a modulus:

- Select a prime number 'p' as the modulus. This choice helps avoid overflow issues and ensures a good distribution of hash values.
- Choose a base 'b' (usually a prime number as well), which is often the size of the character set (e.g., 256 for ASCII characters).

Step 2: Initialize the hash value:

- Set an initial hash value 'hash' to 0.

Step 3: Calculate the initial hash value for the pattern:

- Iterate over each character in the pattern from left to right.
- For each character 'c' at position 'i', calculate its contribution to the hash value as **'c * ($b^{pattern\_length - i - 1}$) % p'** and add it to 'hash'.
- This gives you the hash value for the entire pattern.

Step 4: Slide the pattern over the text:

- Start by calculating the hash value for the first substring of the text that is the same length as the pattern.

Step 5: Update the hash value for each subsequent substring:

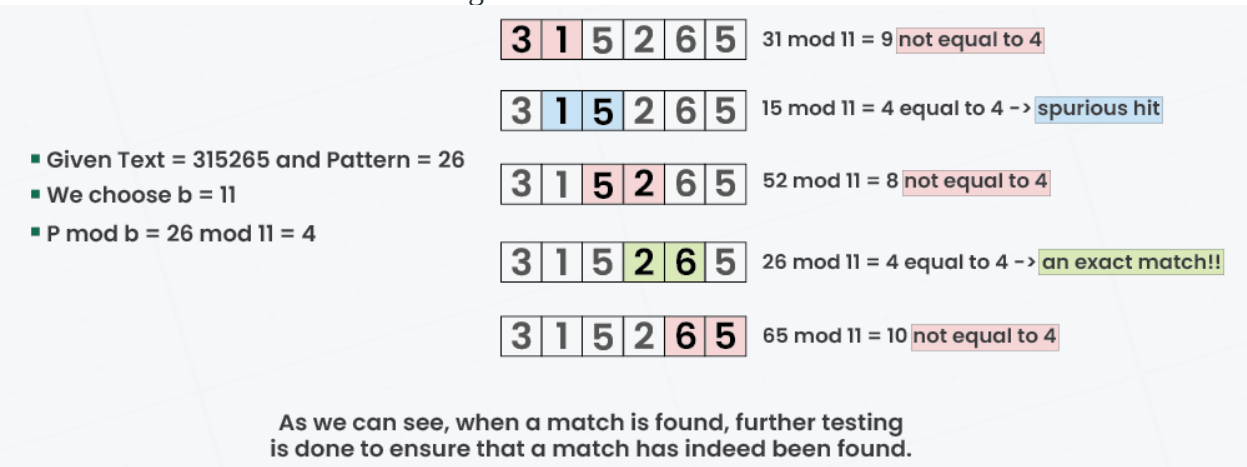
- To slide the pattern one position to the right, you remove the contribution of the leftmost character and add the contribution of the new character on the right.
- The formula for updating the hash value when moving from position ‘i’ to ‘i+1’ is:

$$\text{hash} = (\text{hash} - (\text{text}[i - \text{pattern\_length}] * (b^{\text{pattern\_length} - 1})) \% p) * b + \text{text}[i]$$

Step 6: Compare hash values:

- When the hash value of a substring in the text matches the hash value of the pattern, it’s a potential match.
- If the hash values match, we should perform a character-by-character comparison to confirm the match, as hash collisions can occur.

Below is the Illustration of above algorithm:




```
int calculateHash(const string &str, int base, int length) {
    int hash = 0;
    for (int i = 0; i < length; i++) {
        hash += (str[i] - '0') * pow(base, length - i - 1);
    }
    return hash;
}

void rabinKarp(const string &text, const string &pattern, int base) {
    int n = text.length();
    int m = pattern.length();
    int patternHash = calculateHash(pattern, base, m);
    int textHash = calculateHash(text.substr(0, m), base, m);
    for (int i = 0; i <= n - m; i++) {

        if (textHash == patternHash) {
            if (text.substr(i, m) == pattern) {
                cout << "Pattern found at index " << i << endl;
            }
        }


        if (i < n - m) {
            textHash = (textHash - (text[i] - '0') * pow(base, m - 1)) * base + (text[i + m] - '0');
        }
    }
}
```

Boyer-Moore Algorithm

Boyer Moore is a combination of the following two approaches.

1. Bad Character Heuristic
2. Good Suffix Heuristic

Both of the above heuristics can also be used independently to search a pattern in a text. Let us first understand how two independent approaches work together in the Boyer Moore algorithm.
If we take a look at the Naive algorithm, it slides the pattern over the text one by one. KMP algorithm does preprocessing over the pattern so that the pattern can be shifted by more than one. The Boyer Moore algorithm does preprocessing for the same reason. It processes the pattern and creates different arrays for each of the two heuristics. At every step, it slides the pattern by the max of the slides suggested by each of the two heuristics. So, it uses greatest offset suggested by the two heuristics at every step.

Unlike the previous pattern searching algorithms, the Boyer Moore algorithm starts matching from the last character of the pattern.

Bad Character Heuristic

The idea of bad character heuristic is simple. The character of the text which doesn't match with the current character of the pattern is called the Bad Character. Upon mismatch, we shift the pattern until:

1. The mismatch becomes a match.
2. Pattern P moves past the mismatched character.

Case 1 – Mismatch become match

We will look up the position of the last occurrence of the mismatched character in the pattern, and if the mismatched character exists in the pattern, then we'll shift the pattern such that it becomes aligned to the mismatched character in the text T.

Explanation:

In the above example, we got a mismatch at position **3**.

Here our mismatching character is “A”. Now we will search for last occurrence of “A” in pattern. We got “A” at position 1 in pattern (displayed in Blue) and this is the last occurrence of it. Now we will shift pattern 2 times so that “A” in pattern get aligned with “A” in text.

<u>Case 2</u> – Pattern moves past the mismatch character

We’ll lookup the position of last occurrence of mismatching character in pattern and if character does not exist, we will shift pattern past the mismatching character.

Explanation:

Here we have a mismatch at position **7**.

The mismatching character “C” does not exist in pattern before position 7 so we’ll shift pattern past to the position 7 and eventually in above example we have got a perfect match of pattern (displayed in Green). We are doing this because “C” does not exist in the pattern so at every shift before position 7 we will get mismatch, and our search will be fruitless.

```
void preprocessBadCharacter(string pattern, int badChar[NO_OF_CHARS]) {
    int m = pattern.length();
    fill(badChar, badChar + NO_OF_CHARS, -1);
    for (int i = 0; i < m; i++) {
        badChar[(int)pattern[i]] = i;
    }
}

void boyerMoore(string text, string pattern) {
    int n = text.length();
    int m = pattern.length();
    int badChar[NO_OF_CHARS];
    preprocessBadCharacter(pattern, badChar);

    int shift = 0;
    while (shift <= (n - m)) {
        int j = m - 1;

        while (j >= 0 && pattern[j] == text[shift + j]) {
            j--;
        }

        if (j < 0) {
            cout << "Pattern found at index " << shift << endl;
            shift += (shift + m < n) ? m - badChar[text[shift + m]] : 1;
        } else {
            shift += max(1, j - badChar[text[shift + j]]);
        }
    }
}
```

Knuth-Morris-Pratt (KMP) Algorithm

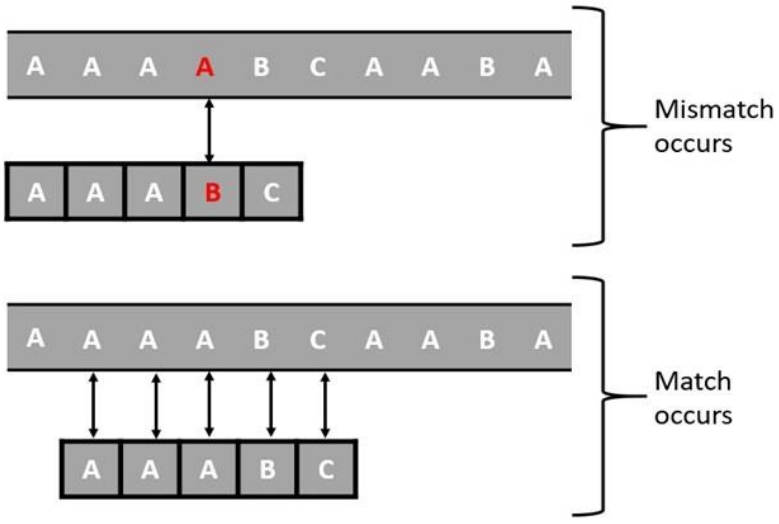
The KMP algorithm is used to solve the pattern matching problem which is a task of finding all the occurrences of a given pattern in a text. It is very useful when it comes to finding multiple patterns. For instance, if the text is "aabbaaccaabbaadde" and the pattern is "aabaa", then the pattern occurs twice in the text, at indices 0 and 8.

The naive solution to this problem is to compare the pattern with every possible substring of the text, starting from the leftmost position and moving rightwards. This takes O(n*m) time, where 'n' is the length of the text and 'm' is the length of the pattern. When we work with long text documents, the brute force and naive approaches may result in redundant comparisons. To avoid such redundancy, Knuth, Morris, and Pratt developed a linear sequence-matching algorithm named the KMP pattern matching algorithm. It is also referred to as Knuth Morris Pratt pattern matching algorithm.

How does KMP Algorithm work?

The KMP algorithm starts the search operation from left to right. It uses the prefix function to avoid unnecessary comparisons while searching for the pattern. This function stores the number of characters matched so far which is known as LPS value. The following steps are involved in KMP algorithm −

- Define a prefix function.
- Slide the pattern over the text for comparison.
- If all the characters match, we have found a match.
- If not, use the prefix function to skip the unnecessary comparisons. If the LPS value of previous character from the mismatched character is '0', then start comparison from index 0 of pattern with the next character in the text. However, if the LPS value is more than '0', start the comparison from index value equal to LPS value of the previously mismatched character.



LPS Array Construction

The LPS array helps in determining the next position in the pattern to consider after a mismatch. Here's how to construct it:

- Start with an LPS value of 0 for the first character (lps[0] = 0).
- For each subsequent character:
    a. If the current character matches the character at the prefix index, increment the LPS value.
    b. If a mismatch occurs, use the previous LPS value to continue checking for a smaller prefix.
- Repeat until the entire pattern is processed.

```
int* computeLPSArray(const string &pattern) {
    int m = pattern.length();
    int* lps = new int[m];    // dynamic array
    lps[0] = 0;

    int len = 0;
    int i = 1;

    while (i < m) {
        if (pattern[i] == pattern[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
    return lps;
}
void KMPSearch(const string &text, const string &pattern) {
    int n = text.length();
    int m = pattern.length();

    int* lps = computeLPSArray(pattern);

    int i = 0;
    int j = 0;

    while (i < n) {
        if (text[i] == pattern[j]) {
            i++;
            j++;
        }

        if (j == m) {
            cout << "Pattern found at index " << i - j << endl;
            j = lps[j - 1];
        }
        else if (i < n && text[i] != pattern[j]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }

    delete[] lps; // free memory
}
```

LAB TASKS

1. You must build a phone directory using a hash table of size 20. The hash function is:

   h(number) = (number / 1000) % 20

   You insert the phone numbers: 9081234567, 9089876543, 9085551212, 7321119988, 7325550199. Implement hashing and determine which numbers hash to the same index. Implement collision handling using separate chaining and print the final structure of the chains.
2. Given an array arr[] of size n filled with numbers from 1 to n-1 in random order. The array has only one repetitive element. The task is to find the repetitive element.
3. You're implementing a word frequency counter using a hash map where keys are words and values are counts. If a word already exists, its count must be incremented. Given the input text: the cat and the dog and the mouse. Write a function insertOrUpdate(word) using chaining. Show the internal hash table after inserting all words. Display the word with the highest frequency.
4. Your university wants to build a lightweight plagiarism checker. Given two paragraphs of text, the program must find whether the smaller text snippet (pattern) appears inside the larger text body (text). Since the paragraphs are long, a hash-based search is preferred. Implement the Rabin–Karp algorithm to search for the pattern snippet in the main paragraph using a rolling hash technique. Return all starting indices where the pattern occurs in the text.
5. Given a 2d matrix of strings arr[][] of order n * 2, where each array arr[i] contains two strings, where the first string arr[i][0] is the employee and arr[i][1] is his manager. The task is to find the count of the number of employees under each manager in the hierarchy and not just their direct reports. Note: Every employee reports to only one manager. And the CEO reports to himself. Print the result in sorted order based on employee name.