Data Structures Lab 12

Course: Data Structures (CL2001) **Semester:** Fall 2024
Instructor: Mr. Sameer Faisal

Note:

- Understand and implement different string searching algorithms, including {**Brute Force**, **Rabin-Karp**, **Boyer-Moore**, and **Knuth-Morris-Pratt (KMP), Graphs}**
- Maintain discipline during the lab.
- Just raise your hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.
- Don't just blatantly copy the same code make changes to it accordingly

---

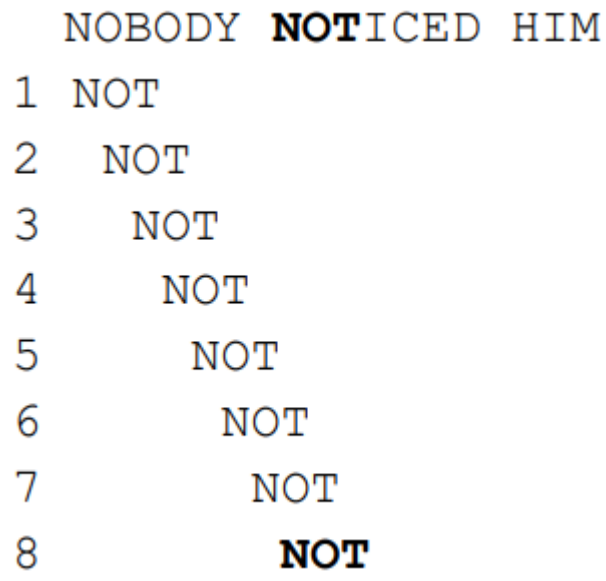
Introduction to String Searching

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.



Code Example

```
#include <iostream>
#include <string>
using namespace std;

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;
        }
}
}

int main() {
string text = "ABAAABCDBBABCDDEBCABC";
string pattern = "ABC";
bruteForceSearch(text, pattern);
return 0;
}
```

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:

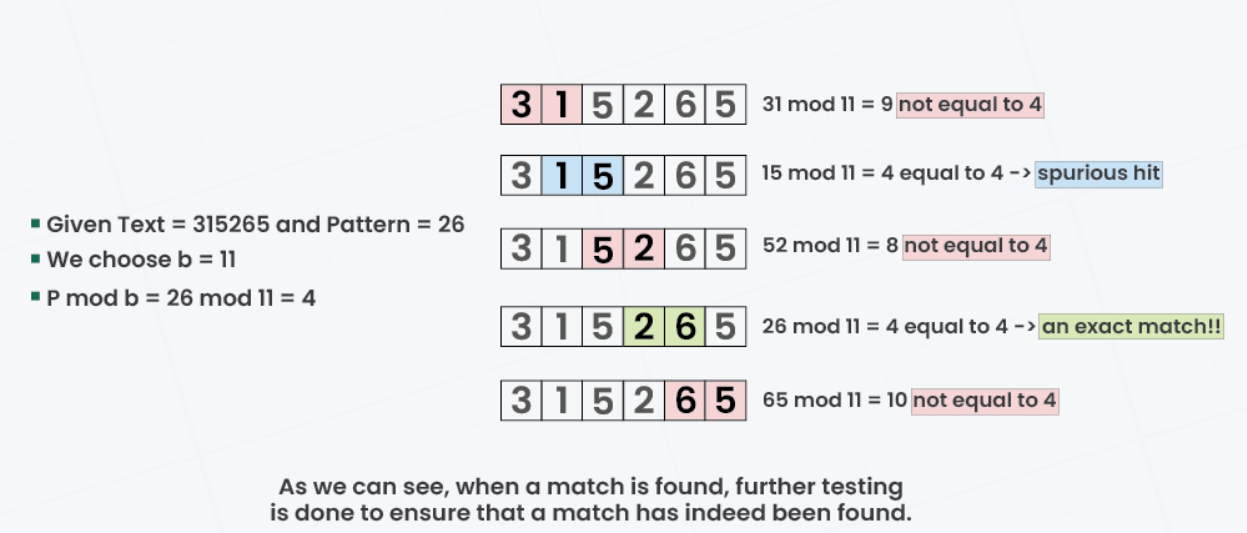
- To slide the pattern one position to the right, you remove the contribution of the leftmost character and add the contribution of the new character on the right.
- The formula for updating the hash value when moving from position 'i' to 'i+1' is:

```
hash = (hash - (text[i - pattern_length] * (b^(pattern_length - 1)) % p) * b + 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:



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.

<u>Case 1</u> – Mismatch become match

We will lookup 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.

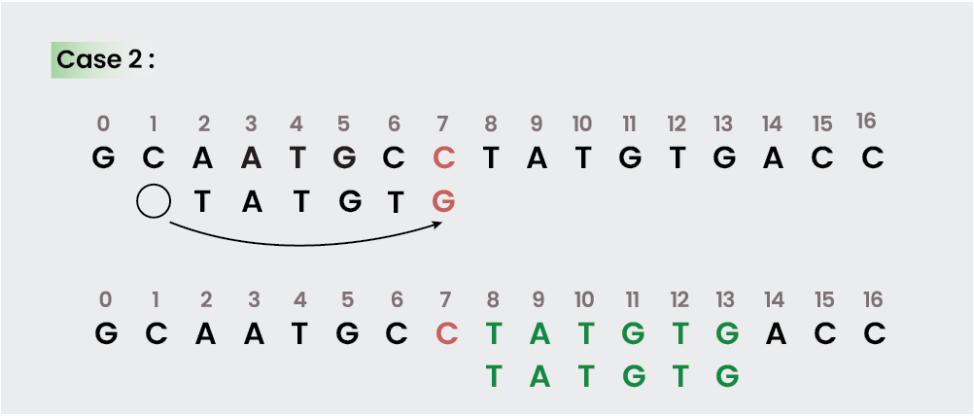
Explanation:

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.

Knuth-Morris-Pratt (KMP) Algorithm

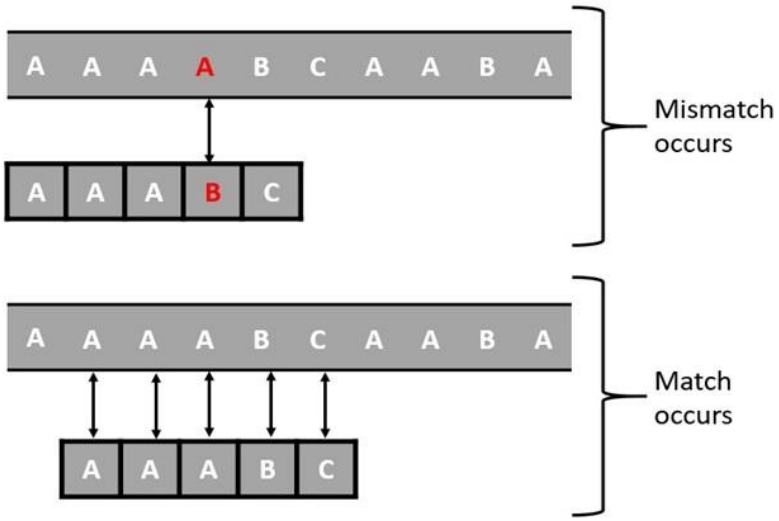
The **KMP** algorithm is used to solve the pattern matching problem which is a task of finding all the occurrences of a given pattern in a text. It is very useful when it comes to finding multiple patterns. For instance, if the text is "aabbaaccaabbaadde" and the pattern is "aabaa", then the pattern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

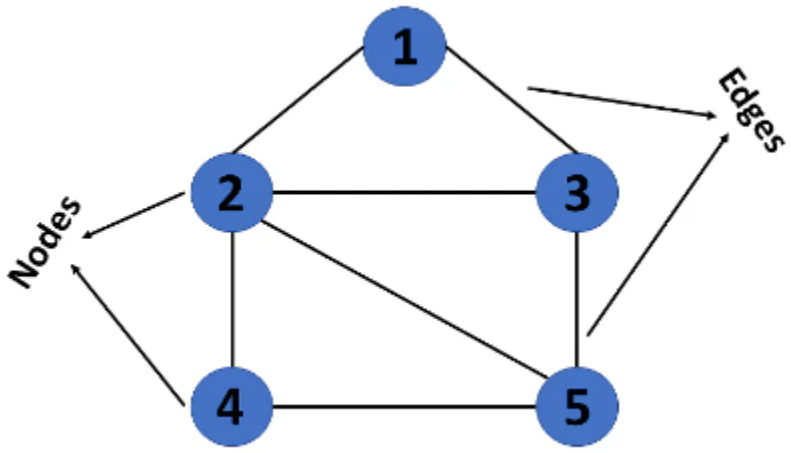
- Start with an LPS value of 0 for the first character (lps[0] = 0).
- For each subsequent character:
    a. If the current character matches the character at the prefix index, increment the LPS value.
    b. If a mismatch occurs, use the previous LPS value to continue checking for a smaller prefix.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

Introduction to Graphs

- Graphs in data structures are non-linear data structures made up of a finite number of nodes or vertices and the edges that connect them.
- Graphs in data structures are used to address real-world problems in which it represents the problem area as a network like telephone networks, circuit networks, and social networks.

What Are Graphs in Data Structure?

- A graph is a non-linear kind of data structure made up of nodes or vertices and edges.
- The edges connect any two nodes in the graph, and the nodes are also known as vertices.



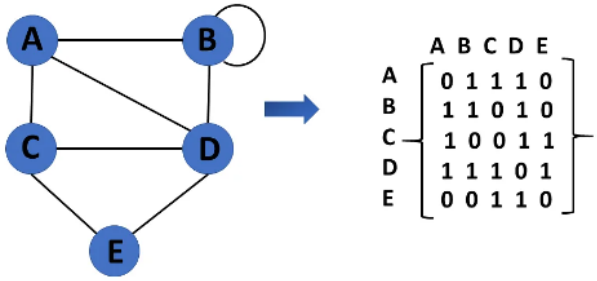
The graph has vertices V = { 1,2,3,4,5} and set of edges E={(1,2),(1,3),(2,3),(2,4),(2,5),(3,5),(4,50)}

Representation of Graphs in Data Structures
The most frequent graph representations are the two that follow:

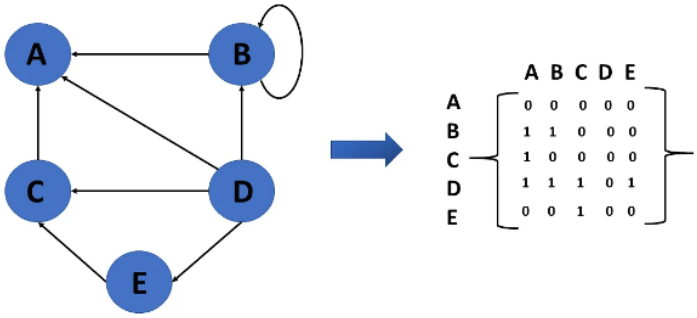
1. Adjacency matrix
2. Adjacency list

Adjacency Matrix
It's used to show which nodes are next to one another. I.e., is there any connection between nodes in a graph? If an edge exists between vertex a and vertex b, the corresponding element of G, gi,j = 1, otherwise gi,j = 0.

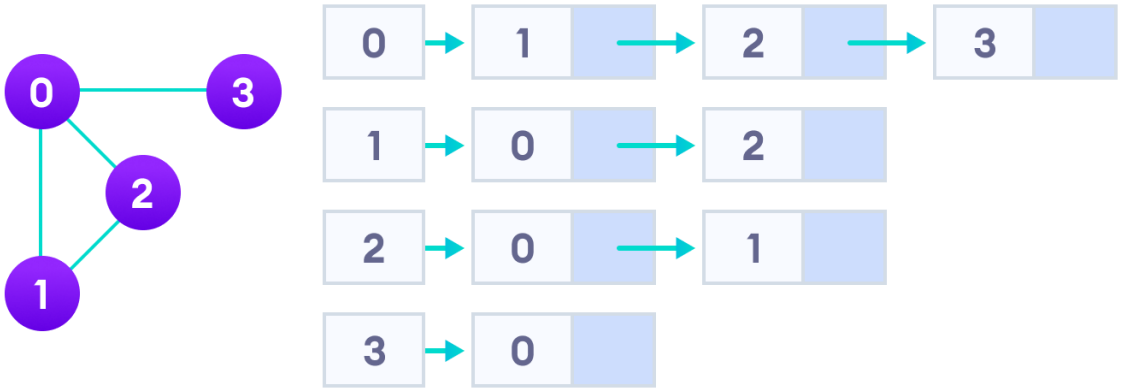
Undirected Graph Representation



Directed Graph Representation



Adjacency List
A linked representation is an adjacency list.
You keep a list of neighbors for each vertex in the graph in this representation. It means that each vertex in the graph has a list of its neighboring vertices.



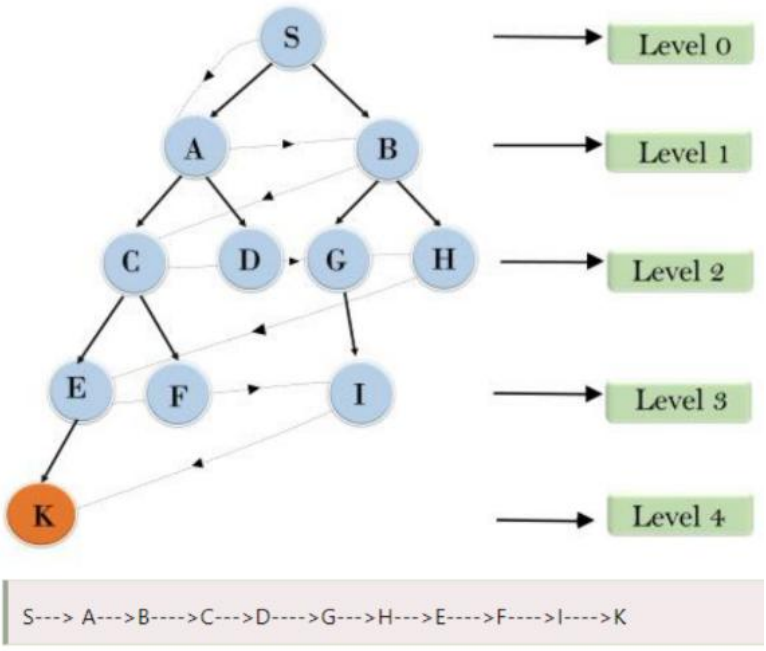
You have an array of vertices indexed by the vertex number, and the corresponding array member for each vertex x points to a singly linked list of x's neighbors.

Graph Traversal Techniques:

1. BFS (Breadth First Search).
2. DFS (Depth First Search).

Breadth-First Search (BFS):

BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.



```
Graph::Graph(int V)
{
this->V = V;
adj = new list<int>[V];
}
void Graph::addEdge(int v, int w)
{
adj[v].push_back(w); // Add w to v’s list.
}
void Graph::BFS(int s)
{
// Mark all the vertices as not visited
bool *visited = new bool[V];

for(int i = 0; i < V; i++)
visited[i] = false;
// Create a queue for BFS
list<int> queue;
// Mark the current node as visited and enqueue it
visited[s] = true;
queue.push_back(s);
// 'i' will be used to get all adjacent
// vertices of a vertex
list<int>::iterator i;
while(!queue.empty())
{
// Dequeue a vertex from queue and print it
s = queue.front();
cout << s << " ";
queue.pop_front();
// Get all adjacent vertices of the dequeued
// vertex s. If a adjacent has not been visited,
// then mark it visited and enqueue it
for (i = adj[s].begin(); i != adj[s].end(); ++i)
{
if (!visited[*i])
{
visited[*i] = true;
queue.push_back(*i);
}
}
}
}
```

Depth-First Search (DFS):