CS 600 A - Advanced Algorithms – Homework 8

Name: Deepali Nagwade | 20013393

Question 20.6.3: For what values of d is the tree T of the previous exercise an order- d B-tree?

Answer: In the context of an order-d B-tree:

- Each internal node can have at most (d-1) keys.
- Each internal node can have at most d children.

For the specified multi-way tree T:

- Each internal node can have at most (d-1) keys, where d = number of children.
- Each internal node can have at most d children.

Given that the number of keys equals the number of children -1, it follows that for tree T:

- Each internal node can have at most (d-1) keys, with d ranging from 5 to 8 (inclusive).

Hence, to fulfil the conditions of an order-d B-tree, valid values for 'd' encompass 6, 7, and 8, as these values satisfy the specified criteria for a tree with 5 to 8 children in each internal node.

Question 20.6.21: For what values of d is the tree T of the previous exercise an order- d B-tree?

Answer: To efficiently implement a queue in an external-memory setting with disk transfers in mind, we can use a B-tree with delayed merging. Key features include:

- Each B-tree node represents a disk block and can store multiple elements.
- Delayed merging is employed, meaning nodes are merged only when full, and the merging operation is deferred.
- Sequential I/O access is optimized for disk transfers, ensuring efficient read and write operations on disk blocks.
- Enqueue operations maintain element order, and if a node becomes full during insertion, merging is delayed.
- Dequeue operations remove elements from the front, triggering delayed merging if it results in a node being less than half full.

This structure minimizes disk transfers, with the total number of transfers for a sequence of enqueue and dequeue operations being O(n/B). This is achieved by deferring merges until necessary and employing a sequential access pattern, making it suitable for handling a large number of operations in a consumer-producer process with minimal I/O overhead.

Question 20.6.21: Imagine that you are trying to construct a minimum spanning tree for a large network, such as is defined by a popular social networking website. Based on using Kruskal's algorithm, the bottleneck is the maintenance of a union-find data structure. Describe how to use a B-tree to implement a union-find data structure so that union and find operations each use at most O(log n/ logB) disk transfers each.

Answer:

The union-find problem involves given union operations with find queries interspersed. Combining the batched union-find algorithm with a persistent B-tree resolves this issue. Unlike the batched union-find algorithm, which defers using find query information until the end, the persistent B-tree efficiently handles ray-shooting queries with a worst-case complexity of O(logB N) IO per query. The batched union-find algorithm is executed until persistent B-trees are generated for respective segments. The construction of persistent B-trees, taking linear space, occurs in O(SORT(N)) IO. Consequently, a linear-sized data structure for the semi-online union-find problem is established, enabling O(logB N) IO for answering find queries at any given time.

Algorithm:

input: a sequence Σ of union and find operations;

output: a set R of (x, %(x)) pairs for each element x involved in Σ.

if Σ can be processed in main memory then Call an internal memory algorithm;

else Split Σ into two halves Σ1 and Σ2; R1 = UNION-FIND(Σ1);

(a) For $\forall$(x, %(x)) $\in$ R1, replace all x in Σ2 with %(x); R2 = UNION-FIND(Σ2);

(b) For $\forall$(x, %(x)) $\in$ R1, if $\exists$(y, %(y)) $\in$ R2

s.t. y = %(x), replace (x, %(x)) with (x, %(y)) in R1;

return R1 $\cup$ R2.

Question 23.7.11: What is the longest prefix of the string "cgtacgttcgtacg" that is also a suffix of this string?

Answer: To find the longest prefix of a string that is also a suffix, you can use the Knuth-Morris-Pratt (KMP) algorithm. The KMP algorithm preprocesses the pattern to efficiently search for occurrences of a pattern within a text.

To use the KMP algorithm to find the longest prefix of a string that is also a suffix of that string, we first need to compute the longest prefix-suffix (LPS) array for the pattern string. The LPS array for a pattern string P is an array of length |P| that stores the length of the longest proper prefix of each suffix of P that is also a prefix of P.

Once we have computed the LPS array for the pattern string, we can use it to find the longest prefix of the text string that is also a suffix of the text string. We do this by comparing the pattern string to the text string one character at a time, and using the LPS array to skip over characters that we have already matched.

Here is an example of how to use the KMP algorithm to find the longest prefix of the string "cgtacgttcgtacg" that is also a suffix of that string:

Step 1: Compute the LPS array for the pattern string one by one so example is this "cgtacg"

Step 2: Compare the pattern string "cgtacg" to the text string "cgtacgttcgtacg"

Step 3: If j == len(P), then the longest prefix of the text string that is also a suffix of the text string is the pattern string

```
P = "cgtacg"
lps = [0, 0, 0, 0, 0, 1]
i = 0,j = 0
while i < len(T) and j < len(P):
    if T[i] == P[j]:
        i += 1,       j += 1
    else:
        if j == 0:
            i += 1
        else:
            j = lps[j - 1]
if j == len(P):
print("The longest prefix of the string", T, "that is also a suffix of this string is", P)
else:
    print("No such prefix exists")
```

In this case, j == len(P), so the longest prefix of the string "cgtacgttcgtacg" that is also a suffix of that string is the pattern string "cgtacg".

Question 23.7.15: Give an example of a text T of length n and a pattern P of length m that force the brute-force pattern matching algorithm to have a running time that is $omega$ (nm)

Answer: The Brute Force algorithm compares the pattern to the text, one character at a time, until matching characters are found.

Brute Force pattern matching algorithm compares the pattern P with text T starts from index i that ranges from 0 to n-m, where n is length of the String T and m in length of the pattern p the search will continue until

Match is found.

In Brute Force pattern matching algorithm the scan is made from left to right.

Algorithm Brute Force Pattern Matching(T,P):

Input: Strings T (text) with n characters and P (pattern) with m characters

Output: Starting index of the first substring of T matching P, or an indication

that P is not a substring of T.

for i ← 0 to n − m // for each candidate index in T do

for j ← 0 to m

while (j<m and T[i + j] = P[j]) do

j ← j + 1

if j = m then

return i (matched at index value i)

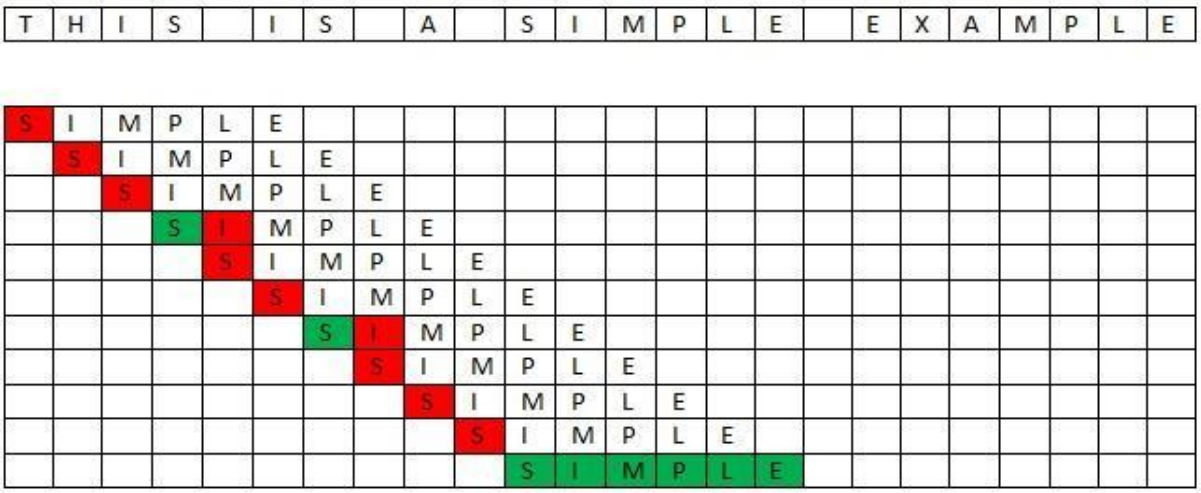
else

return “There is no substring of T matching P.”

Let us understand this with the following example below:

Let our text (T) be as, "THIS IS A SIMPLE EXAMPLE"

Pattern (P) be as, "SIMPLE"



In above red boxes says mismatch letters against letters of the text and green boxes says match letters against letters of the text.

In first row we check whether first letter of the pattern is matched with the first letter of the text. It is mismatched, because "S" is the first letter of pattern and "T" is the first letter of text. Then we move the pattern by one position as shown in second row.

Then check first letter of the pattern with the second letter of text. It is also mismatched. Likewise we continue the checking and moving process. In fourth row we can see first letter of the pattern matched with text. Then we do not do any moving but we increase testing letter of the pattern. We only move the position of pattern by one when we find mismatches.

We continue this process until we find the matching pattern P in the given text T. In the above example's last row, we can see all the letters of the pattern matched with the some letters of the text continuously.

In the above Brute Force algorithm, outer for-loop is executed at most $n - m + 1$ times, and the inner loop is executed at most m times. Thus, the running time of the brute-force method is $O((n - m + 1)m)$, which is $O(nm)$.

Question 23.7.32: One way to mask a message, M, using a version of steganography, is to insert random characters into M at pseudo-random locations so as to expand M into a larger string, C. For instance, the message, ILOVEMOM, could be expanded into AMIJLONDPVGEMRPIOM. It is an example of hiding the string, M, in plain sight, since the characters in M and C are not encrypted. As long as someone knows where the random characters where inserted, he or she can recover M from C. The challenge for law enforcement, therefore, is to prove when someone is using this technique, that is, to determine whether a string C contains a message M in this way. Thus, describe an $O(n)$-time method for detecting if a string, M, is a subsequence of a string, C, of length n.

Answer:

Algorithm: Let M be a subsequence of C, and the first character of M is matched to a position i. But, the first character of M also occurs at position $j < i$ in C. By considering the character at j as being part of M, it is still valid, and M is a subsequence of C starting from position j as well.

Hence, for each character of M, only the first match starting from the match of the previous character is enough to consider.

Hence, the algorithm is as following.

Keep two indices a and b, to iterate over M and C respectively. Initialize both of them to 0.

If the characters M[a] and C[b] match, then increment both indices by one. This means a character of M has been matched.

Else, increment b by one.

If the end of M is reached before reaching end of C, then M is a subsequence. Hence, output true. Otherwise, output false.

Note that in the worst case, b is incremented as many times as the length of C, hence the algorithm takes O(n) time in the worst case.

Here is a psuedocode:

a + 0,0, found + false\n

while true

if Ma == CU\n

aa+1\n

b+b+1\n

else

b+b+1\n

endif

if a == len(M)\n

found = true\n

break

```
endif
if b == len(C)\n
break
endif
endwhile

return found
```