1.
For what values of d is the treeT of the previous exercise an order-d B-tree?

Solution:
It is given that each internal node has at least and at most eight children . So the order of the B tree can be 5,6, 7 and 8..

2.
Suppose you are processing a large number of operations in a consumer-producer process, such as a buffer for a large media stream. Describe an external-memory data structure to implement a queue so that the total number of disk transfers needed to process a sequence of
N enqueue and dequeue operations are O(n/B).

Solution:
Consider several inefficient external-memory dictionary implementations based on sequences. If the sequence representing a dictionary is implemented as an unsorted, doubly linked list, then insert and remove may be done with O(1) transfers each, with each block containing an item to be deleted.
Furthermore, searching needs (n) transfers in the worst-case scenario, because each link hop we do may reach a different block. This search time may be reduced to O(n/B) transfers, where n represents the number of Enqueue and Dequeue operations and B is the number of list nodes that can fit inside a block. We could also use a sorted array to implement the sequence.

In this scenario, a binary search performs $O(\log_2 n)$ transfers.
However, in the worst-case scenario, we will need (n/B) transfers to accomplish an insert or removal operation since we may need to access all blocks to shift pieces up or down. As a result, implementations of sequence dictionaries are inefficient for external memory.

Only $O(\log_B n) = O(\log n/\log B)$ transfers are required to perform dictionary queries and updates.

The main idea for improving the dictionary implementations' external-memory performance is to perform up to O(B) internal-memory accesses to avoid a single disk transfer, where B denotes the size of a block.

This many internal-memory visits are performed by the disk only to move a block into internal memory, and this is only a minor portion of the cost of a disk transfer. Thus, O(B) high-speed internal-memory accesses represent a little cost to avoid a time-consuming disk transfer.

We can represent our dictionary using a multiway search tree, which is a generalization of the (2, 4) tree data structure to a structure known as an (a, b) tree, to reduce the importance of the performance difference between internal-memory and external-memory accesses for searching.

Thus, a Buffered Repository Tree is a structure with a lower insertion cost than a higher lookup cost.

3.

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 (from Section 7.2) so that union and find operations each use at most O(log n/log B)disk transfers each.

Solution:

We will use a simple divide-and-conquer algorithm that bypasses the need of an MST algorithm. It uses O((N) · log(N /M)) I/Os, but has a much smaller hidden constant, and can be easily implemented. The input to a recursive call is a sequence Σ of union and find operations. The recursive call outputs the answers of all FIND(xi) queries in Σ and returns a set R of (x, ϱ(x)) pairs, one for each element x involved in any operation in Σ, where ϱ(x) is the representative of the set containing x after all union operations in Σ are performed. The basic idea behind a recursive call, outlined in Algorithm 1, is the following. If Σ fits in main memory, we use an internal memory algorithm; otherwise we split Σ into two halves Σ1 and Σ2. We solve Σ1 recursively. Before solving Σ2 recursively, we use the element-representative set R1, returned by the recursive call for Σ1, to pass on “information” to Σ2 about how the sets are joined in Σ1. We do so by replacing each element x involved in any operation in Σ2 with ϱ(x) if (x, ϱ(x)) ∈ R1 (line (a)). When the second recursive call on Σ2 finishes, we need to return the complete and correct element-representative set to the upper level calls. All element-representative pairs in R2 are correct, but some in R1 might get updated. We update each (x, ϱ(x)) ∈ R1 with (x, ρ(y)), if there exists some (y, ϱ(y)) ∈ R2 such that ϱ(x) = y (line (b)). Finally we return the union of R1 and R2. Both line (a) and (b) can be performed by a constant number of sort and scan steps (details omitted from this abstract), so the total cost of the algorithm is O(N * log(N / M)) I/Os.

Algorithm 1: Recursive call UNION-FIND(Σ)

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 ∀(x, %(x)) ∈ R1, replace all occurrences of x in Σ2 with ϱ(x);
  R2 = UNION-FIND(Σ2);
(b) For ∀(x, %(x)) ∈ R1, if ∃(y, ϱ(y)) ∈ R2 s.t. y = ϱ(x), replace (x, ϱ(x)) with (x, ϱ(y)) in R1;
return R1 ∪ R2.

Reference: https://www.cse.ust.hk/~yike/union-find/paper.pdf

4.

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

Solution:

T he longest prefix of the string "cgtacgttcgtacg" that is also a suffix of this string is **“cgtacg”**

5.

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 Ω(nm).

Solution:

Let n be the text and let m be the pattern that needs to be find.

Let’s say we have n = thththththththththththe

And m = the

We need to check whether the string ‘the’ exists in the string n

The pseudo code is:

Algorithm: Pattern_finding

Input: A string n and m

Output: Return true if m exists in n.

```
For i in range (0, len(n)):

        if(n[i] != m[0]) then

                continue;

        else

                counter← 0;

                for j in range(0, len(m)

                        if((j + i) >= len(n))

                                break;

                        else if(m[j] != n[i + j]) then

                                break;

                        counter++;

                if(counter == len(m)) then
```

return true;

return false;

The time complexity of the above algorithm is $\Omega(nm)$ and the space complexity is O(1)

6.

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 Min 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.

Solution:

Assume M is a subsequence of C, and the first character of M corresponds to position i. However, the initial character of M appears at position j < i in C. It is still legitimate to treat the character at j as part of M, and M is also a subsequence of C beginning at point j.

As a result, just the first match starting from the preceding character's match is taken into account for each character of M, as a result, the algorithm is as follows.Keep two indices, a and b, to iterate over M and C. Set both of them to zero.

Increase both indices by one if the characters M[a] and C[b] match. The character M has been matched, hence this indicates. Otherwise, raise b by one.

M is a subsequence if the end of M is reached before the end of C. So, produce true. If not, output false.

Keep in mind that the algorithm requires O(n) time in the worst case because b is increased by the same number of times as C's length.