Homework 8 - Solution

R-20.3
(a, b) tree is a multi-way search tree T, where a and b are integers, such that $2 \leq a \leq (b+1)/2$ where

1) Each internal node has at least a child, unless it is the root, and has at most b children.
2) All the external nodes have the same depth.

From 20.2, T is a valid (a, b) tree for $2 \leq a \leq \min\{5, (b+1)/2\}$ and $b \geq 8$.
For example, T could be a (4, 8) tree, a (5, 9) tree, but not a (5, 8) tree.

A B-tree of order d is an (a, b) tree with $a = \lceil d/2 \rceil$ and $b = d$.
To maintain each internal node at least five and at most eight children, the possible values of d are 8, 9 and 10.

A-20.1
The external memory data structure that can be used is a linked list with each of size B.

When an element is enqueued, we take the element to the end of the linked list & check whether there are B elements. If there are B elements, then create a new list into disk and add the element. Thus, there will be $O(n/B)$ times disk transfer.

When doing dequeue, we first do a disk transfer, then remove the first elements in this disk, until there is no element, we then do next disk transfer. If we want to dequeue n elements in worst case it would be $O(n/B)$ times disk transfer.

A-20.2
We will use tree-based implementation of union-find data structure.

The find operation will get the element's position in the tree, which takes $O(1)$ time. Then we trace the path till the root of the tree, in order to get the set name. Since we store the node in an B tree, it will take $O(log\ n/\ log\ B)$ disk transfers.

For union operation we will insert the smaller B tree to the larger B tree. It will first find the right position of the root of smaller tree in larger tree, which takes $O(log\ n/\ log\ B)$ disk transfers, then the restructuring, which will take $O(log\ n/\ log\ B)$ disk transfers.

R-23.11
The longest prefix that is also the suffix of the string "cgtacgttcgtacg" is "cgtacg".

C-23.1
Consider an example of a text *T* of length *n* is 'aaaaaaaaaaaaaaaaaaah', which has *n*-1 'a's and one 'h' in the last position. Consider a pattern *P* of length *m* as 'aaah' (*m*-1 'a's and one 'h' in the last position). Now, in brute-force pattern matching algorithm, we will be comparing *m*-1 characters of pattern *P* with each of *n*-1 characters of text *T*, thus the total running time will be $\Omega(nm)$.

C-23.11
We will use bottom-up approach to delete a string from a trie. First we find the external node of the trie that is the last part of our string. This can be achieved in $O(dm)$, where d is the size of the alphabet and *m* is the length of the string. Then we remove all nodes that are specific to that string which would be located in the last node in case of compressed trie. At last we merge the parent node with the other child of end node of our string. So delete operation will take $O(1)$ time. The total running time of the algorithm will be $O(dm)$ time.

A-23.1
We will use compressed trie as the data structure to store the webpage content. To check if a web page exists in the trie, we can start from the first character of the web page and the root of the compressed trie; and we check if the corresponding character is on current level or if the character is the starting of any compressed word in this level. If it matches with a single character, we recur the trie. If it's the start of a compressed word, we compare along the word and the compressed word. If the character doesn't match with any word in the level, we add the rest of the word as a compressed word on this level. If the character matches the beginning part of a compressed word and doesn't match the rest, we break the compress word at wherever they don't match. The rest of the part of the compressed word is attached as a child.

Given a web page of length n, searching may take at most $O(n)$ time. But insertion will take only additional $O(1)$ time.

A-23.5
In order to detect whether a sting *M* is a subsequence of string *C* or not, we traverse both strings from left to right. If we find a matching character then we increment both the pointers of strings *M* and *C*. Otherwise, we only increment pointer of string *C*. Now, if we find end of the string *M* while traversing then we can say that *M* is a subsequence of *C*, else *M* is not a subsequence of *C*. This algorithm takes $O(n)$ time because we are traversing both strings once and *C* is the larger string of length n.