**R-20.7**

Paging is a virtual memory system technique, which link virtual memory with physical memory for larger memory space with a cost of an extra transfer time between accessing different pages. To pay less transfer time we store some most demanding pages in cache and get page in cache first when fetching. However, if the page is not in cache we would get from disk and put a copy into cache and kicked the least-recent-used page in cache if full, which is the LRU algorithm.

My approach would be creating a table for time, request and cache to find the missed requests:

|  |  |  |
| --- | --- | --- |
| Time | Requested Page Number (Is it Missed?) | Cache (Last-Use Time) |
| 0 | Empty | Empty, Empty, Empty, Empty |
| 1 | 2 (Miss) | 2(1), Empty, Empty, Empty |
| 2 | 3 (Miss) | 2(1), 3(2), Empty, Empty |
| 3 | 4 (Miss) | 2(1), 3(2), 4(3), Empty |
| 4 | 1 (Miss) | 2(1), 3(2), 4(3), 1(4) |
| 5 | 2 (In Cache) | 2(5), 3(2), 4(3), 1(4) |
| 6 | 5 (Miss) | 2(5), 5(6), 4(3), 1(4) |
| 7 | 1 (In Cache) | 2(5), 5(6), 4(3), 1(7) |
| 8 | 3 (Miss) | 2(5), 5(6), 3(8), 1(7) |
| 9 | 5 (In Cache) | 2(5), 5(9), 3(8), 1(7) |
| 10 | 4 (Miss) | 4(10), 5(9), 3(8), 1(7) |
| 11 | 1 (In Cache) | 4(10), 5(9), 3(8), 1(11) |
| 12 | 2 (Miss) | 4(10), 5(9), 2(12), 1(11) |
| 13 | 3 (Miss) | 4(10), 3(13), 2(12), 1(11) |

The concept is straight forward, the total miss count is 9 misses.

**C-20.1**

Since the insertion should be O(1) and transfer and searches is O(n/B), we cannot use B-Trees which is O(logB n) in transfer and search. However, we can use a simpler approach with a simple linked list which each node’s size is exact B, same as the block size. For each insertion, we transfer the tail node from disk to memory and then do the checking. If it’s not full, we add the element to the tail and transfer the node back to disk. If it’s full we create a new node in memory, add the element, and create a link from new node to previous one. Then transfer back to disk.

The insertion is just transferring the tail node to memory and transferring it back, which should be O(2) => O(1). The search need to go through each block of memory, we have n elements and B elements can fill a block, thus, we need to examine n/B blocks and thus O(n/B) time for the search.

**A-20.4**

According to textbook page 590, we should sort external-memory to put all same keys to same file and then perform a linear scan through. The sorting need block transfers, where M is the size of the internal memory, B is the size o disk blocks, n is number of elements. The rest is just comparing the key to each sorted file and get the file we want. The total time is O (+m), where m is the number of files after the sorting.

**R-23.12**

The Karp-Rabin algorithm still need to compare the real value after the hashed value matched. To confirm the match is not collision, it needs O(m) time to compare the length-m pattern. The length of the text is n, if it needs to compare with each element in text, then it would be O(nm) as follows: Text: aaaaaaaa…aaa <= the number of “a” is n; Pattern: aaaaa…aaa <=the number of “a” is m.

**C-23.4**

All we need to do is modifying the KMP function. Note that the failure function is un-changed.

**Algorithm** *KMPMatch\_Modified*(***T, P***)

***F*** ← ***failureFunction*** (***P***)

***i*** ← 0

***j*** ← 0

// Initialization

prefixIndex ← 0 //Modified

prefixLength ← 0 //Modified

prefixMaxLength ← 0 //Modified

**while *i*** < ***n***

**if *T***[***i***]= ***P***[***j***]

// Matched pattern => max length += 1

prefixLength ← prefixLength + 1 //Modified

**if j**= ***m*** -1

**return i**- ***j*** {match}

**else**

***i*** ← ***i*** +1

***j*** ← ***j*** +1

**else**

**if j**> 0 (we have advanced in P)

***j*** ← ***F*** [***j*** -1]

// compare and set max length

**if prefixLength > prefixMaxLength** //Modified

prefixMaxLength = prefixLength //Modified

prefixIndex = i – j //Modified

**else**

***i*** ← ***i*** +1

// reset temp length whatever case

prefixLength = 0 //Modified

**return -**1 { no match }

After the algorithm returned, we can get the prefixMaxLength and prefixIndex denote the longest prefix of P that is a substring of T at certain position in T. This should still take O(n+m).

**A-23.2**

We can use compressed Tries structure to store all the stop words like: {he, her, hers, herself, him, himself, his} is as following. Note that “/0” means theEnd of a word.

In this case we can get stop-word identification in constant time for all constant-length stop words. A search for d digit key is O(d).