***Chapter 20***

***R-20.7*** Consider an initially empty memory cache consisting of four pages. How many

page misses does the LRU algorithm incur on the following page-request sequence?

(2*,* 3*,* 4*,* 1*,* 2*,* 5*,* 1*,* 3*,* 5*,* 4*,* 1*,* 2*,* 3)

Answer:

Initially: (\_,\_,\_,\_)

Step (2): Page Miss: (2, \_, \_, \_)

Step (3): Page Miss: (2, 3, \_, \_)

Step (4): Page Miss: (2, 3, 4, \_)

Step (1): Page Miss: (2, 3, 4, 1)

Step (2): Page Hit (no change): (2, 3, 4, 1)

Step (5): Page Miss: (3, 4, 1, 5)

Step (1): Page Hit (no change): (3, 4, 1, 5)

Step (3): Page Hit (no change): (3, 4, 1, 5)

Step (5): Page Hit (no change): (3, 4, 1, 5)

Step (4): Page Hit (no change): (3, 4, 1, 5)

Step (1): Page Hit (no change): (3, 4, 1, 5)

Step (2): Page Miss: (4, 1, 5, 2)

Step (3): Page Miss: (1, 5, 2, 3)

Total Page Miss = 7

***C-20.1*** Show how to implement a dictionary in external memory, using an unordered sequence

so that insertions require only *O* (1) transfers and searches require *O*(*n/B*)

transfers in the worst case, where *n* is the number of elements and *B* is the number

of list nodes that can fit into a disk block.

Answer:

This problem can be solved by using hashing double linked list which allows to insert and remove operations in O (1) transfers each. So, it can be assumed that the node is a block of size B. insert operations are executed just like they are performed in the linked list. Last block of the list is accessed to insert operation and if the outcome is empty block new element is inserted in that block. Then this block is transferred back to the disk when the process is done. On the other side of the process if the block is found full then new block space is allocated, and then the new element is inserted in the list. Same way element can be removed from the list when which block holds the item to be removed is known. Insertions require O (1) transfer in the double linked list. The search requires O (n/b) transfers in the worst case here (b) denotes the number of nodes of the list that fits in a block. This process occurs because every link traverse in the hash could access the different block. If we consider n number of nodes in the dictionary and B contains a certain number of nodes, then at each Hop, the size is reduced by B nodes. And hence each Hop takes O (n/b) time.

***A-20.4*** In the MapReduce framework, for performing a parallel computation, a crucial

step involves an input that consists of a set of *n* key-value pairs, (*k, v*), for which

we need to collect each subset of key-value pairs that have the same key, *k*, into a

single file. Describe an efficient external-memory algorithm for constructing all

such files. How many disk transfers does your algorithm perform?

Answer:

To solve this problem efficiently first all the elements are sorted through executing a sorting algorithm, multiway merge sort algorithm can be used here.

Multimerge(k,v)

Here the set of n key-value pairs (k,v) would be the input and the output would be a collection of each subset of key-value pairs that have the same key into a single file.

Multiway merge sort divides S into the x numbers of subsets S1, S2, S3, …. SX of almost equal sizes, and recursively sorts each subset Si, and then simultaneously merge all x sorted lists into sorted representation of S. This recursive algorithm is applied to sort each subset Si, such a way that the pair that has the duplicate key are merged into single key with an increased count. And at the same time, Pairs with same keys can be collected in single file by merging all d sorted list.

The time required: O((n/B) log (n/B) / log (M/B)) here n is the number of elements, b is the size of disk blocks and M is the size of internal memory.

***Chapter 23***

***R-23.12*** Give an example of an input instance for lexicon matching problem, with just a

single pattern in the lexicon, *L*, that forces the Karp-Rabin algorithm given in

Algorithm 23.11 to run in Ω(*nm*) time.

Answer:

Karp-Rabin occurs when all characters of text and pattern are same.

The time complexity for Rabin-Karp is O (lnm + l log l) in the worst case where n is the length of the character string, T and m is the length of the pattern & l is the number of patterns.

The time complexity would be O (nm +0) = O(nm) because there is only a single pattern. The instance input is T = Apple, L = ppl

Karp-Rabin algorithm can verify every possible match because is this case there is a match for every substring of T. Thus, applying the algorithm hash match (L, T) will run in Ω(nm).

***C-23.4*** Let *T* be a text of length *n* and let *P* be a pattern of length *m*. Describe an

*O* (*n* + *m*)-time method for finding the longest prefix of *P* that is a substring

of *T*.

Answer:

Knuth Morris Pratt Algorithm which can give us the output in O (n+m) can be used to get the desired solution. There are few key points it covers.

If P[j] = T[i] where P is pattern and T is the String sequence. Then, last(P) is checked whether it has reached and increment I and j. if j>0 checks whether we found any common character match in the pattern and proceeded and stuck. If Yes, then j will be an index from P which did not match. Else if the first index did not match the n increment i. Similarly, instead of checking a match of the pattern track of count it kept.

KMPcount(T, P) algorithm. Strings T (text with n characters and P pattern with m characters is the Input and output are starting Index of the first substring of T attaching P or largest prefix.

f ← KMPFailureFunction(P)

i←0, j ←0, Max = 0, count = 0

while I <n do

if P[j] = T[i] then

if j = m-1 then

count = 1 – m + 1

return count

i←i +1

j←j+1

count = count +1

else if j > 0

if (max <count) then

max – count

count =0

j← f(j-1)

else

i ← i +1

max = 0

count = 0

This algorithm will run in item O (n+m)

***A-23.2*** Search engines need a fast way to detect and ignore stop words, that is, words,

such as prepositions, pronouns, and articles, that are very common and carry no

meaningful information content. Describe an efficient method for storing and

searching a set of stop words in a way that supports stop-word identification in

constant time for all constant-length stop words.

Answer:

This can be solved using a hash table since a lookup in a hashTable takes O (1) time. The hashtable uses a key & values. All keys in a hashTable are unique and are searched by the search engine while the values, in this case, would be the frequency of words. As each file is parsed HashTable is also simultaneously defined that will tokenize and map each word. Instead of an array, a hashtable is used which allows keeping track of a number of occurrences.

The top N keys with the highest number of occurrences are the Stop Words & Matching each word in the HashTable would result in the time in O (1).