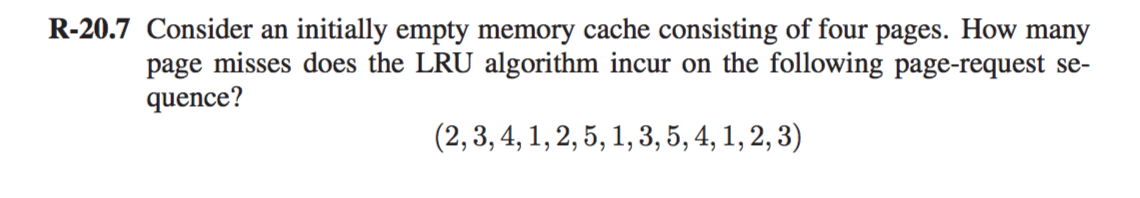
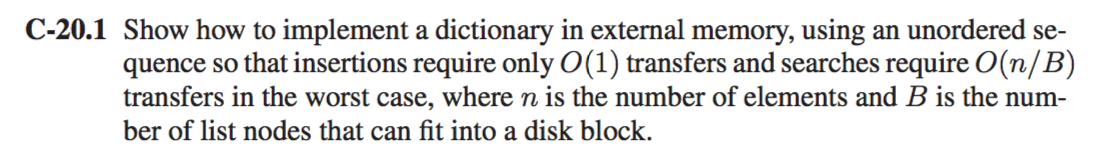
CS 600 Homework 9 | CWID 10430147 | Divyendra Patil | Username: dpatil3  
Date: 11/10/2017



Solution: 2 3 4 1 2 5 1 3 5 4 1 2 3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Page 1 | **2** | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | **4** | 4 | 4 | 4 |
| Page 2 |  | **3** | 3 | 3 | 3 | **5** | 5 | 5 | 5 | 5 | 5 | 5 | **3** |
| Page 3 |  |  | **4** | 4 | 4 | 4 | 4 | **3** | 3 | 3 | 3 | **2** | 2 |
| Page 4 |  |  |  | **1** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Page Misses: 9



Solution:

To solve this problem statement, we approach it by using hashing using doubly linked list which will allow the insert and remove operations in **O(1)** transfers each.

1] We assume the node to be a block of size B.

2] Carry out insert operations similarly as we perform in linked list.

3] To perform the insert operation, we firstly access the last block of the list & if we come across an empty block, we insert new element in that block & then transfer this block back to the disk when we are done.

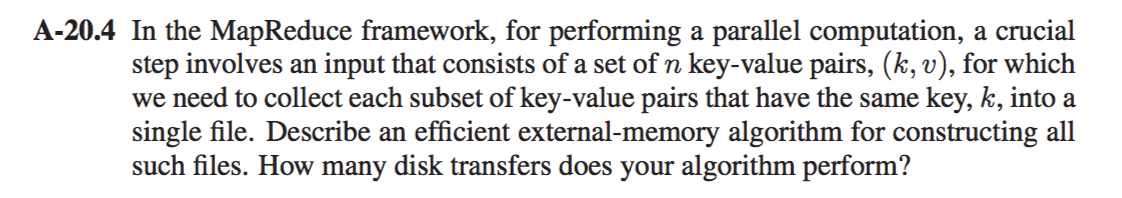
4] On the other hand, If we find that the block is full, we allocate a new block space and then insert the new element in the list.

5] Likewise, we can remove the element from the above list considering we know which block holds an item to be removed.

6] Insertions require O(1) transfer in Doubly Linked List

7] Search requires O(n/B) transfers in worst case (B denotes the number of nodes of the list that can fit in a block). This happens because every link traverse in the hash could access the different block.

8] If we consider n number of nodes in the dictionary and B contains a certain number of nodes, then at each Hop, the size get reduced by B nodes. Therefore, each Hop takes around O(n/B)time.



Solution:

To solve this problem, an efficient way to approach it is to first sort all the elements using an external memory sorting algorithm. Hence, we use multiway merge sort algorithm to solve it.

**Algorithm** MultiMerge(k, v)

**Input:** Set of n key value pairs (k, v)

**Output:** Collection of each subset of key value pairs that have the same key into a single file.

**Explanation**:

Multi-way merge-sort divides S into d subsets S1, S2, . . ., Sd of almost equal size, and recursively sorts each subset Si, and then simultaneously merge all “d” sorted lists into a sorted representation of “S”.

Now, apply recursive algorithm to sort each subset Si, in a manner such that the **pairs with duplicate key** are merged to a single key with an increased count. At the same time, Pairs with same keys can be collected in single file by merging all d sorted list

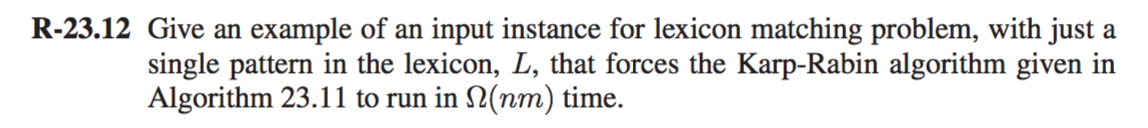
The time required:

**O((n/B) log(n/B) / log(M/B))**

where n = number of elements,

B = the size of disk blocks &

M = the size of internal memory.



Solution:

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

Time Complexity for Rabin-Karp is O(lnm + l log l) in worst case

where n is the length of the character string,

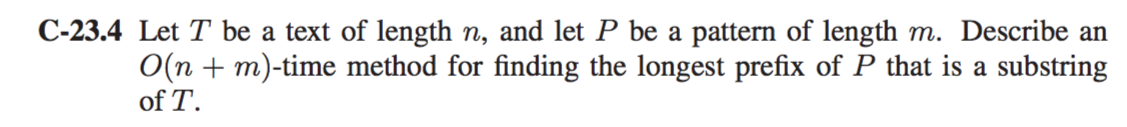
T and m is the length of the pattern &

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 in this case there is a match for every substring of T. Thus, applying the algorithm hashMatch(L, T) will run in Ω(nm).



Solution:

Knuth Morris Pratt Algorithm which can give us the output in O(n + m) can be used to give us the desired solution. It works on these key points:

1) If P[j] = T[i] where P is pattern and T is the String sequence.

Then, check if last(P) is reached and increment I and j.

2) If j > 0, checks whether we found any common character match in the pattern and

proceeded and stuck.   
If YES, then j will be index from P which did not match.

3) Else if the first index didn’t match the n increment i.

Similarly, instead of checking a match of the pattern we will keep track of count.

**Algorithm** KMPCount(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 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 = i − m + 1

return count

i ← i + 1

j ← j + 1

count = count + 1

else if j > 0 // no match, but we have advanced in P then

If(max < count) then

max = count

Count = 0

j ← f(j − 1)

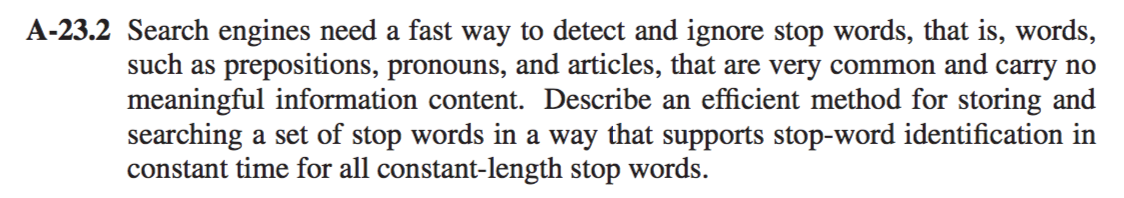
else

i ← i + 1

max = 0

Count = 0

This algorithm will run in time **O(n + m).**



Solution:

1] We solve the above problem using a hash table, since a lookup in a HashTable takes O(1) time.

2] 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.

3] As we examine/scan/search/parse each files/documents/website, we simultaneously define a HashTable that will tokenize & map each word.

4] Instead of array we use HashTable which will also keep track of number of occurrences. (concept of inverted file)

5] The top N keys with highest number of occurrences are the STOP WORDS & Matching each word in the HashTable would result the time in **O(1).**