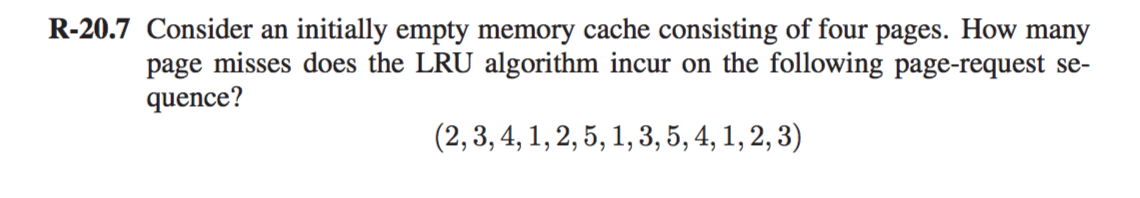
Homework 8

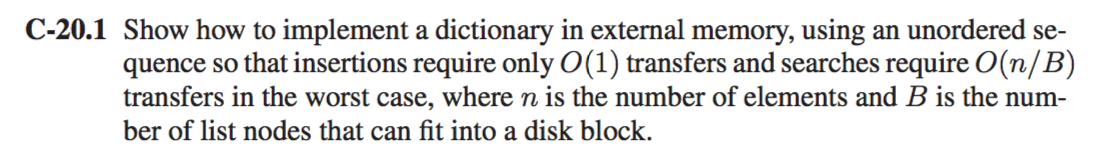
Atiq Patel CWID:10432833



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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1st Page | **2** | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | **4** | 4 | 4 | 4 |
| 2nd Page |  | **3** | 3 | 3 | 3 | **5** | 5 | 5 | 5 | 5 | 5 | 5 | **3** |
| 3rd Page |  |  | **4** | 4 | 4 | 4 | 4 | **3** | 3 | 3 | 3 | **2** | 2 |
| 4th Page |  |  |  | **1** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Total Pages misses: 9



**Solution:**

We can solve this problem by making the use of a hashing table implemented using doubly linked list which takes care of insertions and deletions at **O (1)** time.

Let B be the block size node. We perform insertions as same as how we perform in the linked list. We check the last block of the linked list and if we get an empty block all we do insert an element in the block and then transfer it back to the block to the disk once we are finished.

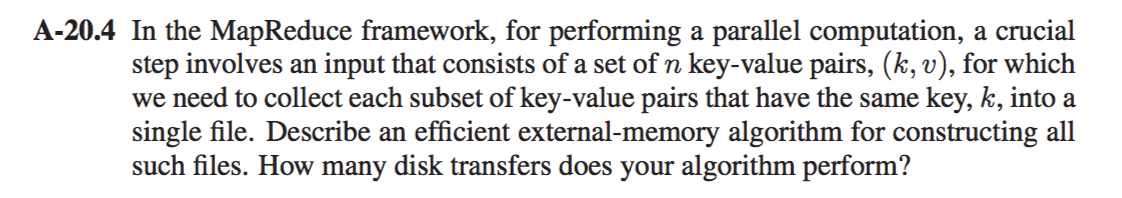
But if the Block is full, new block space allocation is done and we insert the new element in the list.

In the same way deletion of an element can be done from the list, since we would know that which one of these block would hold the element that needs to be deleted.

The process run time is O (1) in the Doubly List.

The searching takes O (n/B) considering the worst case where B stands for the number of nodes that can be fit inside a block. This happens because every traversal in the link might access a different block.

If we assume that there are “n” number of nodes inside the dictionary and “B” contains number of nodes, then every time it transfers, the size get reduced by B nodes. Thus, each and every Hop requires **O(n/B)** transfer time.



**Solution:**

A way in which we can solve this is by sorting using an external memory algorithm like Multiway Merge sort.

**Algorithm MutliwayMergeSort(k,v):**

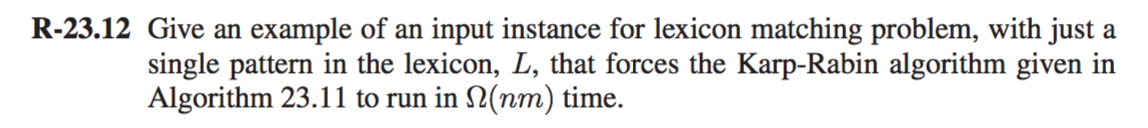
**Input:** A set of n key value pairs (k,v).

**Output:** Collection of key value pairs that have the same key k sorted into the file.

Sort using multi way merge sort S which divides S into n subsets S1,S2…,Sd of almost equal size, and recursively sorts, each subsets Si and merge simultaneously all the “d” sorted list together into a sorted list “S”.

After applying recursion to sort out each individual subset Si in a way that all the pairs with the same key k are merged together into a single key with increased value (i.e. count) and collecting all the pairs with the same key and stored into a single sorted list.

The total time required would be **O((n/B) log (n/B)/log (M/B))** where n are the number of elements, B stands for the size of the disk blocks and M stands for the size of internal memory.



**Solution:**

The run time for Karp-Robin algorithm is *O* (*lnm* + *l* log *l*) wherein n is the length of the character string, T and m are the length of the pattern and l is the number of pattern.

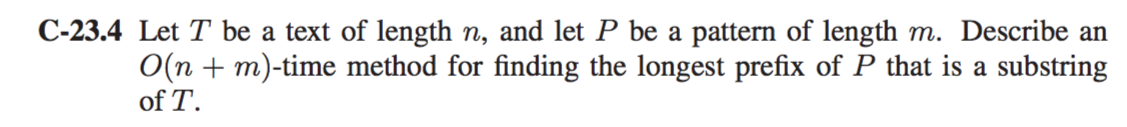
Since there is only a single pattern. The total run time would be O(nm + 0) = O(nm).

Let’s take an example for the following input

T= BANANA

L = ANANA

This algorithm can seemingly match every possible match in this case for the substring of T. Therefore, if we use the algorithm HashMatch(L,T). It will run in **Ω(nm).**



**Solution:**

We will use the Knuth Morris Pratt Algorithm which works on these three factors: -

1. If P[i] = P[j] do

Check if the last Pattern has reached then do

i🡨i+1 and j🡨j+1

1. If j>0 then check for any common character match in the pattern that has been found or proceeded and stuck.

If true then assign j as the index from P where the pattern did not match.

1. Else if the first index that didn’t match the n increment i.

We keep a count rather than checking a match of a pattern.

**Algorithm Count(P,T):**

**Input:** Texts T of length n characters and P patterns of length m.

**Output:** The longest prefix of P that is a substring of T.

J 🡸 0

Totmax 🡸 0

Counter 🡸 0

**While** i🡨0 to n **do**

**If** P[i] = T[i] **do**

**If** j=m-1 **do**

Counter = I - m + 1

**Return** Counter

i++

j++

counter ++

**else if** j > 0 **do**

**if** (totmax < Counter ) **do**

totmax = Counter

Counter 🡸 0

J 🡸 KMPfailurefunction(j-1)

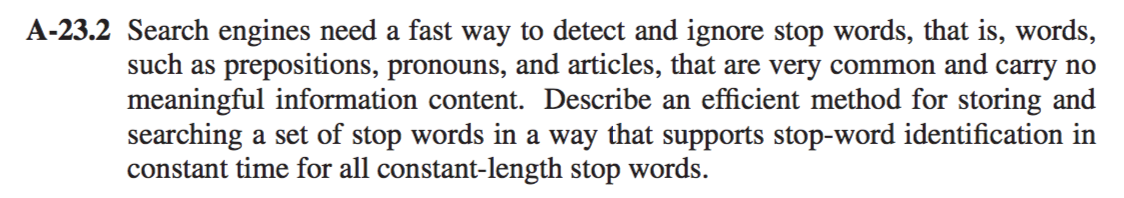
**Else do**

I 🡸 I + 1

Totmax 🡸 0

Counter 🡸 0

The run time for this algorithm would be **O (n+m)**.



**Solution:**

A very efficient way to get searching and checking for stop words would be O (1), which is possible with the use of Hash Tables.

Hash Table have key value pairs. Keys can be the each of the stop words and the values could be the frequency of those stop words.

So let’s suppose a search engine wants to wants to start its searching, all the algorithm has to break down the string and tokenize each individual words.

After that’s done all the algorithm has to do is search through the lookup table in the hash table and see for the stop words inside the hash table.

The “n” number of words which occur very often other are the stop words. The process of matching would be **O (1) time.**