CS 600 Homework 8 Solutions

R-20.7 Page Misses LRU Algorithm

We have the following page-request sequence $(2, 3, 4, 1, 2, 5, 3, 5, 4, 1, 2, 3)$ and an initially empty cache consisting of four pages.

The LRU strategy evicts the page that was least recently used. Here is a table describing the process. The pages in the cache are arranged in ascending order of most recently used. Therefore, the least recently used page appears in the last position of the cache. The current page being requested and the page in the cache that will be evicted in the round are underlined.

Time	Page Requests	Cache	Hit or Miss
1	2,3,4,1,2,5,1,3,5,4,1,2,3	{}	Miss
2	3,4,1,2,5,1,3,5,4,1,2,3	{2}	Miss
3	4,1,2,5,1,3,5,4,1,2,3	{3,2}	Miss
4	1,2,5,1,3,5,4,1,2,3	{4,3,2}	Miss
5	2,5,1,3,5,4,1,2,3	{1,4,3,2}	Hit
6	5,1,3,5,4,1,2,3	{2,1,4, 3}	Miss
7	1,3,5,4,1,2,3	{5,2,1,4}	Hit
8	3,5,4,1,2,3	{1,5,2, 4}	Miss
9	5,4,1,2,3	{3,1,5,2}	Hit
10	4,1,2,3	{5,3,1, 2}	Miss
11	1,2,3	{4,5,3,1}	Hit
12	2,3	{1,4,5, 3}	Miss
13	3	{2,1,4, 5}	Miss
14	-	{3,2,1,4}	-

The final state of the cache is $\{3, 2, 1, 4\}$ in our representation. There are a total of 9 page misses.

C-20.1 Dictionary in External Memory

We are required to implement a dictionary in external memory, using an unordered sequence so that insertions require $O(1)$ 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.

Consider a linked list implementation of the dictionary where every node is a block of size B. We perform an insertion by first reading in the last block. If this block is not full, then we add the new element to the end and transfer this block back. If this block is full, then we allocate a new block for the new element, transferring this block back to the disk when we are done. We also create a link in the previous block to this new block and we transfer the previously last block back to the disk. We also create a link in the previous block to this new block and we transfer the previously last block back to external memory.

This ensures insertions in $O(1)$ time. Note that searching in linked lists takes $O(n)$ time. In the worst case, the item we are searching for is at the end of the list. Here we perform $O(n/B)$ disk transfers since each disk block contains B list nodes.

A-20.4 MapReduce File Construction

In the MapReduce framework, a crucial step involves an input that consists of n key-value paris, (k, v), for which we need to collect each subset of key-value pairs that have the same key, k, into a single file.

Sort the elements in S using an external-memory sorting algorithm, such as the *Multi-way Merge-sort* algorithm (page 590 of the textbook). This brings together the elements with the same keys. We perform one more scan and write each set of elements with the same key to a file. The time for this algorithm is dominated by the time to perform the sort. This can be done using $O(\frac{(n/B)\log(n/B)}{\log(M/B)})$ block transfers, where n is the number of elements; B is the size of a block; and M is the size of primary memory.

R-23.12 Lexicon Matching Problem

This problem requires an example of a lexicon, L, with a single pattern, that forces the Karp-Rabin algorithm to run in $\Omega(nm)$ time.

To ensure a lower bound $\Omega(nm)$, we need to make sure that nm comparisons are made. This can be achieved when each character of the text string T matches every character in every string in the lexicon. As an example,

$$T = \underbrace{(\text{a a a} \ldots \text{a})}_{\text{n times}} \quad P = \underbrace{(\text{a a a} \ldots \text{a})}_{\text{m times}} \quad L = \{P\}$$

C-23.4 Longest Prefix that is a Substring

T is a text of length n, and P is a pattern of length m. We require an $O(n+m)$ time method for finding the longest previs P that is a substring of T. Since there is no information on the alphabet, Σ, it's best to not use algorithms that exhibit complexities based on $|\Sigma|$ (such as the original Boyer-Moore algorithm, which has a run-time complexity of $O(n+m+|\Sigma|)$.

Modify the Knuth-Morris-Pratt algorithm to maintain variables maxIndex which is the index of the longest prefix found; maxLen which is the length of the longest prefix found; and currentLen which is the length of the current prefix. Initialize all three variables to zero and modify the loop in KMPMatch as follows,

- if $T[i] = P[j]$, then currentLen $\leftarrow$ currentLen $+ 1$
- If $T[i] \neq P[j]$ and $j > 0$, then,
  - if currentLen $>$ maxLen, then maxLen $\leftarrow$ currentLen, and maxIndex $\leftarrow i - j$. In both cases, set currentLen $\leftarrow 0$.

When the algorithm terminates, maxIndex, and maxLen, will hold the location and length of the longest prefix.

A-23.2 Constant Time Stop Word Identification

Search engines require a fast way to detect and ignore stop words. Stop words are words such as prepositions, pronouns, and articles, that are very common and provide little information. This problem requires a constant time method for stop-word identification for all constant-length stop words.

To solve this problem, we can either use a compressed trie or a cuckoo hash table. Both of these ensure that we get a constant-time stop-word identification for any constant-length stop word.