Department of Electrical and Computer Engineering
McGill University

ECSE-322 Computer Engineering

22 January 2008

Problem Set 2 - Solutions

1. Hashing:

We need to use a hashing function in order to store, in an array of strings, family names of the 10 top students of a class of 1000 students. We have the following family of hashing functions:

$H_k(e) = ((e) \text{Mod} M + k) \text{Mod } M$, where k is the hashing function number, and M is the number of positions in the array.

(a) Give some examples of what we can use as the key: *e*.
Solution:
The position of the 1 letter of the student's name in the alphabet can be used as key. This will lead to some collisions as more than 1 student's family name might start with the same letter. Another possibility is to use the ASCII representation of the first letter, or maybe the sum of all ASCII representations of the letters in the family names. But given the random nature of family names, and the fact that we are mapping a large number of possible family names onto a smaller space, collisions are bound to occur.

(b) What size array should we dedicate for storage?
Solution:
Since we know that the names of 10 students are to be stored, the size of the array should be 10.

(c) Describe when hashing is best suited to store data (in general). Is it well suited for this problem?
Solution:
Hashing is best suited when we need to map the domain into a considerably smaller space. It is well suited for this case since we are trying to store a limited amount of data from a large pool of possible elements; and we want to be able to retrieve them later in an effective manner.

(d) Explain why the 1^{st} Mod function is useful? The 2^{nd}?
Solution:

The first Mod function is useful for mapping the key onto the available positions in the array. The second Mod function is useful for wrap-around indexing when collisions occur and the family of hashing functions is used (i.e. after using a function that maps the element to the last position and there is a collision, the next function will map the element onto the first position in the array).

(e) How would you go about retrieving such data?

Solution:

Retrieving the data involves applying the hashing function used to store the data originally. If, using the key, the first hashing function is used and the element is not found (i.e. there was a collision), consequent hashing functions in the family of functions are applied until the data is found.

(f) What problem with hashing does bucket hashing help solve?

Solution:

Bucket hashing helps deal with the problem of collision. If a collision occurs, there is no need to apply the hashing function again, the element is placed in a bucket in the same position. A problem occurs of course if the number of collisions in a particular space is bigger than the number of available buckets. An attempt to avoid this is to properly design the hashing function depending on the expected nature of the data to be stored.

(g) Show the array of hashed elements for the following names using e=(sum(ASCII(1st letter of name)+ ASCII(last letter of name))):

Names: Daniels, Arnold, Beaudet, Stephens, Durocher, Atkins, Moore, Cameron, Ini, Lanf

(i) Using the family of hashing function described

Solution:

Daniels: ((68+115)Mod10+0)Mod10 = 3
Arnold: ((65+100)Mod10+0)Mod10 = 5
Beaudet: ((66+116)Mod10+0)Mod10 = 2
Stephens: ((83+115)Mod10+0)Mod10 = 8
Durocher: ((68+114)Mod10+0)Mod10 = 2 -> collision
((68+114)Mod10+1)Mod10 = 3 -> collision
((68+114)Mod10+2)Mod10 = 4
Atkins: ((65+115)Mod10+0)Mod10 = 0
Moore: ((77+101)Mod10+0)Mod10 = 8 -> collision
((77+101)Mod10+1)Mod10 = 9
Cameron: ((67+110)Mod10+0)Mod10 = 7
Inog: ((73+103)Mod10+0)Mod10 = 6
Lane: ((67+101)Mod10+0)Mod10 = 7 -> collision
((67+101)Mod10+1)Mod10 = 8 -> collision
((67+101)Mod10+2)Mod10 = 9 -> collision
((67+101)Mod10+3)Mod10 = 10 -> collision (wrap around indexing)
((67+101)Mod10+4)Mod10 = 1

Atkins	Lane	Beaudet	Daniels	Durocher	Arnold	Inog	Cameron	Stephens	Moore

(ii) Using bucket hashing

Solution:

Atkins	Beaudet		Daniels		Arnold	Inog	Cameron	Stephens	
	Durocher					Lane		Moore	

2. Design a data structure for representing signed integer numbers of arbitrary size, with efficient algorithms for addition and comparison. What is the running time of your algorithms?

Solution:

To represent unbounded sizes we need a dynamical structure. Since operations on integers generally involve traversal of the bits of an integer in positional order, we can solve the problem with a linear structure, i.e. a linked list.

We can perform addition and comparison using simple bitwise algorithms. For bitwise comparison, we need to traverse the list from the MSbit to the LSbit. If the bits are different, the comparison is decided, unless the lists have different lengths, in which case the longer number is also the larger. For bitwise addition, we need to traverse the list from the LSbit to the MSbit and, at each position, sum up two bits and a carry bit. Hence we need a doubly linked list.

The *worst-case* running time of the bitwise comparison and addition algorithms is linear in the number of bits, since both involve a full traversal of the lists.

An optimization of the bitwise comparison algorithm is to store the length of the list separately, to avoid performing bitwise comparisons for numbers that have different lengths. If the lists do not need to be fully traversed, we can achieve constant (O(1)) *average* running time for bitwise comparison if the probability of two given bits being different is 1/2. To see that, calculate the average running time as follows: the bits on position N (MSB) are compared each time; the bits on position N-1 are compared half the times; the bits on position N-2 are compared 1/4 of the times; etc. The average number of comparisons is therefore $1 + 1/2 + 1/4 + 1/8 + \ldots = 2$.

Hence we settle for a doubly-linked list with an extra integer representing the length on the side.

To save on the extra storage, we also have the option of storing several bits (a "word") at each node of the list. The word length can be sized depending on the probability distribution of integers of given length.

3 Determine the performance of Quicksort and Bubblesort in the following particular cases:

(a) All the elements of the array have the same value.

(b) The elements of the array are already sorted in increasing order.
(c) The elements of the array are sorted in decreasing order.

Solution:

Quicksort:

(a) Each iteration will result in no swaps, and a list that is shorter by one than the list in the previous iteration. Hence, the performance degrades to $O(N^2)$: there are (N-1) + (N-2) + … + 1 comparisons.

(b) Same as (a).

(c) Each iteration will result in just one swap, and a list that is shorter by one than the list in the previous iteration. Hence, the performance degrades to $O(N^2)$: there are (N-1) + (N-2) + (N-3) + … comparisons, i.e. a sum of approximately N(N-1)/2 elements.

Note, however, that the probability of (b) or (c) happening is 1/ N!, and the factorial grows even quicker than an exponential. The probability of (a) happening is even smaller. This is why Quicksort is still really good on the average case.

Bubblesort:

(a) If all elements in the array have the same value, no swaps occur during the first iteration. If the algorithm is applied directly, there will still be $O(N^2)$ comparisons. However, we can be more clever than this. We can set a stopping criterion for algorithm when no swaps occur in the first iteration (since the data is already ordered). In this case, the first iteration required (N-1) comparisons which leads to a complexity of O(N).

(b) Same as (a) (Since no swaps occur)

(c) For the first iteration, the leftmost element (largest one) is compared to all other elements and N-1 swaps occur. This element is then place in the rightmost position and the N-1 remaining elements are in decreasing order. The second pass requires N-2 comparisons and swaps, the third requires N – 3 … the last pass requires 1 comparison. This leads to N-1+N-2+…+1 which is of $O(N^2)$.

4. Extend the array vectoring formula in the notes to a 6-dimensional array E:
   (a) For an array of 4 x 6 x 5 x 3 x 7 x 2, what is the index of
      (i) E[2,1,4,2,5,1]
      (ii) E[0,0,0,0,0,1]
      (iii) E[0,3,0,0,1,1]
   (b) How much memory is required to store the array?
   (c) What is the time required for index computation?

(d) Develop an indexing function for the form of vectoring for sparse multi dimensional arrays. This function should return the presence or lack of an entry in a specific array position.

Solution:

(a) If we start indexing from position 1:

E[i,j,k,k,m,n] = (((((i-1)J+j-1)K+k-1)L+l-1)M+m-1)N+n

If we start indexing from position 0:

E[i,j,k,l,m,n] = ((((i*J+j)K+k)L+l)M+m)N+n.

For parts (ii) and (iii), the indices clearly start from zero. For part (i), it could be either case.

For an array of I x J x K x L x M x N

(i) Starting from 1:

(E[2,1,4,2,5,1] = (((((2-1)*6+1-1)*5+4-1)*3+2-1)*7+5-1)*2+1 = 1409

Starting from 0:

E[2,1,4,2,5,1] = ((((2*6+1)*5+4)*3+2)*7+5)*2+1 = 2937

(ii) E[0,0,0,0,0,1] = (((0*6+0)*5+0)*3+0)*7+0)*2+1 = 1

(iii) E[0,3,0,0,1,1] = (((0*6+3)*5+0)*3+0)*7+1)*2+1 = 633

(b) Depending on the type of data to be stored, 4 x 6 x 5 x 3 x 7 x 2 = 5040 elements are required to be stored.

(c) This requires 5 multiplications, decrements and additions.

(d) To find whether a specific array position exists, we need to see if an element that has the specified row and column number is present in the array. We fist find whether any elements are stored in the specified column. This is done by looking at the vector array. Finding whether the row exists simply consists of checking if the row number exists the in the row array.

Assume we have a Boolean called *presence,* to specify whether the element is present in the array or not, a variable *required_column* that is the column number of the element we want, *required_row* that is the row number of the element we want, a *vector_array* to store the positions of new columns in the linear array, and *row_index_array* to store the row indices of all elements. Then,

```
Presence = 0;
For(I = vector_array[required_column] to I = vector_array[required_column+1]
-1)
//check all the elements of the desired column to see if the row exists
        {
                if row_index_array[I] = = required_row
                        presence = true;
      }
```

```
return(presence);
```

5. An MxN array is stored by rows. What minimum amount of temporary storage is required to rewrite it so it is stored by columns? How much time is needed? Can time be traded for temporary storage in this case?

Solution:

Storing by rows results in an addressing polynomial:

$$S(i,j) = A ((i-1)*N + j)$$

Storing by columns, the addressing polynomial becomes:

$$S(i,j) = A ((j-1)*M + i)$$

It appears that the smallest amount of temporary storage is 1 location of the same size as those used in the matrix. However, to implement an algorithm using this means that the data in the array is addressed randomly, i.e. starting at location S(1,2) (row 1 column 2), this was stored in A(2), it will be stored in A(M+1).Thus the data in A(M+1) should be moved into temporary storage. If M/N = K with a remainder L, then the data that was in location M+1 was that from S(K+1,L+1). This needs moving to location A(L*M + K + 1). Thus this process can be continued. While it seems intuitively true that this should cover the entire matrix before it terminates, it is not easy to prove without writing a test program. In fact, it can be seen that, if the matrix is square, i.e. M=N, then the process will oscillate on the first move, i.e. the algorithm fails! However, in this particular case, it can be shown that a single temporary location will work but the algorithm involves cycling through the upper triangle of the matrix and swapping elements, i.e. S(i,j) goes to S(j,i), the diagonal terms stay where they are. Thus, in general, a single temporary location will not work.

An alternate, and much more obvious approach, is to create a second linear array of the same size as the first and then to read out the first array by cycling the row indices fastest and copy the column sequentially into the new array. This, however, is expensive in terms of array storage.

The process is a linear pass through the data and thus has a time complexity of O(N).

Any process to reorder the data has to require at least one pass through the data. Thus O(N) is the fastest alogrithm around and this cannot be improved on by using extra storage space.

6. In a square sparse matrix (of dimension several hundred), only the non-zero terms are to be stored. The row and column locations are to be indicated in associated integer arrays in as compact a form as possible, i.e. by using vectoring if possible. If an integer occupies 2 bytes and a real number occupies 4 bytes. Determine the approximate level of sparsity (in terms of the percentage of the full matrix) below which it is not worth storing only the non-zeroes.

Solution:

Assume that the storage system stored by columns and that the matrix is of dimension M by M. Thus the matrix would be stored as follows:

One real array of size N - where N is the total number of non-zeroes
One integer of size N where N is the total number of non-zeroes - this array will contain the row indices.
One real array of size M to store the base addresses of each column.

If the matrix was to be stored with all its zeroes, the total space requirement would be 4*M*M bytes.

In the sparse storage scheme the total space taken is 4N + 2N + 2M.

However, M is much smaller than N so the space is approximately 6N.

Hence the "Break even" point on storage occurs when 6N = 4M*M.

Thus N = 0.66M*M.

This gives a level of sparsity where the matrix is about 70% full.

7. Write the conditions for testing a queue for emptiness. Assume a queue is implemented as an array (NAME) of size k. How many elements may be stored in the queue? Draw pictures illustrating the queue and typical positions for the pointers FRONT and REAR when the queue (a) is empty, (b) contains one element, and (c) is full.

Solution:

The amount of data in a queue is indicated by the positions of the FRONT and REAR pointers.

Items are added at the FRONT and removed at the REAR.

Thus to add an item to the queue, FRONT is set to FRONT+1 and the new item is stored in NAME(FRONT)
To remove and item from the queue, REAR is set to REAR+1.

Thus if there is nothing in the queue, REAR would be pointing one item ahead of FRONT.

i.e. REAR is equal to FRONT+1

Note that this is defining a convention for EMPTY.

In order to distinguish EMPTY from FULL, the queue being FULL must have a different pointer layout from the EMPTY case. Thus FULL can be defined as REAR equal to FRONT + 2 (if this is a circular queue)

Thus, if the array is of length k, then k-1 items can be stored in the queue.

In the drawings, when one item is stored on the stack, the FRONT and REAR pointers point at the same location.

EMPTY STATE:

REAR	
FRONT	

ONE ELEMENT:

REAR = FRONT	ELEMENT 1

FULL:

FRONT	ELEMENT 4
	ELEMENT 3

	ELEMENT 2
REAR	ELEMENT 1

8. The row and column locations of a sparse matrix are to be indicated in associated integer arrays, possibly with vectoring. Determine the approximate levels of sparsity (in terms of the percentage of the full matrix) for which it is best to use array storage, sparse matrix, or vectoring.

Solution:
Assume the matrix has M rows, N columns, and the ratio of non-zero elements over all elements is S.
The amount of storage required by each is:
array: MN
sparse matrix: 3SMN (three integers are stored for each non-zero element: the element, row, and column)
vectoring: M + 2SMN (for each row, one integer pointing to the beginning of the row; for each element, one integer for the element and one integer for the column)
vectoring by columns: N + 2SMN

If memory is the only tradeoff consideration, then:
sparse matrix wins over array iff (if and only if) 3SMN < MN, i.e. S < 1/3;
vectoring (by rows) wins over array iff M+2SMN<MN, i.e. 1+2SN < N, i.e. S < (N-1)/(2N);
vectoring (by rows) wins over sparse matrix iff M + 2SMN < 3SMN, i.e. M < SMN, i.e. S>1/N

In short, if 1>S>(N-1)/(2N), we prefer the array; if (N-1)/(2N)>S>1/N, we prefer vectoring by rows; if 1/N>S>0, we prefer sparse matrix.
(The tradeoff for vectoring by columns can be computed in a similar manner.)

Extra Challenging Questions for the Brave:

9. A copy of the address book of a handheld computer is installed on a desktop PC. Both the handheld and the PC versions of the address book can be updated independently by editing a record, inserting a new record, or deleting an existing record. Design an ADT (Abstract Data Type) for the address book that supports the updates above, as well as a "hot sync" feature that transfers the latest changes on each record from the PC to the handheld and from the handheld to the PC.

Solution:

The ADT is defined by its operations. To support the updates, we need the following operations:

insert (newrecord)
edit (oldrecord, newrecord)
delete (oldrecord)

To support the hotsync feature, we need to determine the last time when a record was last modified, and transfer the modification with the latest date to the other computer. This implies that deleted records must be preserved until the deletion can be transferred to the other computer; therefore each record needs a “deleted” status flag.

Hence, we need to attach some status attributes to each record (the time of last update, and the “deleted” flag). The operations of the ADT will handle these attributes as follows.

```
insert (newrecord)
  addressbook.set(newrecord)
  newrecord.deleted ← false
  newrecord.lastupdate ← now

edit (oldrecord, newrecord)
  addressbook.purge(oldrecord)
  addressbook.add(newrecord)
  newrecord.deleted ← false
  newrecord.lastupdate ← now

delete (oldrecord)
  newrecord.deleted ← true
  newrecord.lastupdate ← now

hotsync()
  for each record r
    external ← otheraddressbook.get (r)
    if (r.lastupdate < external.lastupdate)
{
      edit (r, external)
      r.lastupdate ← external.lastupdate
      if (external.deleted)
        addressbook.purge(r)
        otheraddressbook.purge(external)
    }
 else
   {
      otheraddressbook.edit (external, r)
      external.lastupdate ← r.lastupdate
      if (r.deleted)
        addressbook.purge(r)
        otheraddressbook.purge(external)
```

}

Finally, from the above it follows that the ADT has three new operations:

addressbook.set: sets a record with the given content
addressbook.get: retrieves a record by its content
addressbook.purge: eliminates a record from the database

10. A web browser is configured to hold the most recently visited web pages in a buffer of N1 Kbytes. Assume the average page size is N2 Kbytes and that the hit rate (number of page requests that can be served from the cache over the total number of page requests) is R. Is it more efficient to implement the buffer as an array sorted by visiting date, an unsorted array, a sorted linked list, an unsorted linked list, or a hash table? What are the tradeoffs?

Let N = [N1/N2]. Assume that the browser replaces the oldest page when it retrieves a new page. If a page can be served from the buffer, the visiting time of that page is updated. Under these assumptions, the time to serve a page is:

serve = R*retrieve + (1-R)(findold + replace)

where *retrieve* is the time to find a page in the buffer by its URL, *findold* is the time to find the page with the earliest visitation date in the buffer, and *replace* is the time to substitute the oldest page by a new page.

Solution:

sorted array: retrieve = O(N) (the retrieval is by URL, not by date!), findold = O(1), replace = O(N) (moves all elements).
unsorted array: retrieve = O(N), findold = O(N), replace = O(1) (no moves needed).
sorted linked list (sorted by the visiting date): retrieve = O(N) (requires traversal even if threaded by URL), findold = O(1), replace = O(1).
unsorted linked list: retrieve = O(N), findold = O(N), replace = O(1).
hash table: retrieve = O(1) (if the URL is the key), findold = O(M) (where M is the size of the hash table, unless the hash table has a "next item" operation that permits traversal of the items stored in a manner similar to a linked list), replace = O(1).

So, the performance runner ups are the hash table and the sorted linked list. If R is relatively small and M is relatively large (which is quite common in practice), the sorted linked list fares better.

Other important tradeoffs include memory requirements and simplicity (development effort!), and here the sorted linked list fares better than the hash table.