Comp 182
Data Structures
Sample Midterm Examination

5 November 2014

Name:________ ANSWERS_______________________________
1. Design an ADT that represents a simple Social Networking Site. It should allow a user to post messages to all of his/her friends so that when a friend logs on to the site, they can see the message. Assume that the friends must be confirmed and when once done, each user is a friend of the other.

a. Write a Specification for this problem. Be sure to note that a person may have many friends.

System must have the ability to create a separate module for each specified set of friends. Any set of two or more individuals can request the creation of a module; once the site is created other individuals can request access to that site; if approved by a majority vote of the originators, they will be given access to the site and be deemed to be friends of all users of the site. An individual can be a member of more than one site.

The site must have the ability to accept messages from any member of the site and distribute them to all other members of the site. It should also have the ability to maintain all past messages and make them available for recall by any member of the site.

b. Design a solution to the problem. Determine what objects along with their data fields and associated methods would be necessary to solve this problem.

- queue of new messages awaiting distribution
- each message contains the originators name
- for each message a graph of response messages
- list of topics accepted & distributed
- for each topic, the ability to retrieve all messages concerning that topic
c. Draw a UML diagram that reflects the design in part b above.

<table>
<thead>
<tr>
<th>Social Networking Site N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Queue</td>
</tr>
<tr>
<td>FriendList( )</td>
</tr>
<tr>
<td>List of Topics</td>
</tr>
<tr>
<td>Graph of Messages</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>+ insertMessage( )</td>
</tr>
<tr>
<td>+ accessTopicList( )</td>
</tr>
<tr>
<td>+ accessMessageGraph(Topic : Message)</td>
</tr>
<tr>
<td>+ accessFriendList( )</td>
</tr>
<tr>
<td>+ addFriend( )</td>
</tr>
<tr>
<td>+ returnFriendCount( )</td>
</tr>
</tbody>
</table>
2. Greatest Common Divisor

THEOREM

If \( a > b > 0 \) such that \( b \) is not a divisor of \( a \) then
\[
gcd(a, b) = \gcd(b, a \mod b)
\]

if \( b \) divides \( a \) then
\[
b = \gcd(a, b)
\]
hence
an appropriate choice for the base case is 
\[
a \mod b = 0
\]

Recursive Definition
\[
gcd(a, b) = \begin{cases} 
b & \text{if } (a \mod b) = 0 \\
gcd(b, a \mod b) & \text{otherwise}
\end{cases}
\]

Java Method
\[
\text{public static int gcd( int a, int b) }
\{
\text{ if ( a } \% \text{ b } = = 0 )}
\text{ return b; // base case}
\text{ else}
\text{ return gcd(b, a } \% \text{ b);}
\}
\]

ANSWERS

a) What happens if \( b > a \)?
\[
gcd(a, b) \Rightarrow gcd(b, a) \quad \text{from "return gcd(b, a\%b)"}
\]

b) How is the problem getting smaller? I.e., Does the method always approach the base case? from “return gcd(b, a\%b)”, i.e., \( a\%b < a \)

c) Why is the base case appropriate?
\[
a\%b = = 0 \Rightarrow b \text{ is a divisor of } a
\]
continuation of the algorithm yields the gcd
3.

a. Given the sorted linked list

For the node held by the reference variable

write Java code to insert the node into the appropriate location in the sorted linked list. Be sure to include methods to place the reference variables `curr` and `prev` in the appropriate locations prior to implementing the insertion of the specified node. After you have written the code, review your algorithms to make sure that they actually do what you want them to do!

```java
/* place curr & prev in appropriate locations */

curr = head;
while (curr.value > create.value) {
    prev = curr;
    curr = curr.next;
}

/* link new node into existing linked list */

create.next = curr;
prev.next = create;
```
b. Given the sorted linked list

![Diagram of linked list]

i. Write a Java Method to delete the node \[33\] from the list.

Be sure to include Java code to place the reference variables \(\text{curr}\) and \(\text{prev}\) in the appropriate locations to implement the deletion of the specified node.

/* place curr & prev in appropriate locations */

```
curr = head;
while( curr.value < 33 )
{  prev = curr;
   curr = curr.next;
}
```

/* delete the designated node */

```
prev.next = curr.next;
```

ii. Write a Java Method to delete the node \[21\] from the list.

Be sure to include Java code to place the reference variables \(\text{curr}\) and \(\text{prev}\) in the appropriate locations to implement the deletion of the specified node.

/* answer */

```
curr = head;
curr = curr.next;
head = curr;
```
4. The recursive method `getNumberEqual` searches an array `x` of `n` integers for occurrences of the integer `val`. It returns the number of integers in `x` that are equal to `val`. For example, if `x` contains the nine integers 2, 4, 7, 5, 6, 9, 7, 8, and 7 then `getNumberEqual(x, 9, 7)` returns the value 3 since 7 occurs three times in `x`.

```java
public static int getNumberEqual(int x[], int n, int val)
{
    int count;

    if (n <= 0) return 0;
    else if (x[n-1] == val) count = 1;
    else count = 0;

    return getNumberEqual(x, n-1, val) + count;
}
```

Using the Criteria for Constructing Recursive Solutions, listed below, demonstrate exactly how the method meets each of the criterions.

Criteria for Constructing Recursive Solutions
I. The problem must be capable of being restated as smaller problem of the same type.

    each call to `getNumberEqual` reduces the index `n` by 1 hence it ultimately must reach zero

II. A limited sequence of recursive calls, if not each recursive call, must diminish the size of the problem.

    each call to `getNumberEqual` reduces the index `n` by 1 hence it ultimately must reach zero

III. It must be possible to isolate an instance of the problem that can serve as a base case.

    Since all indexes e.g., `n >= 0` when the index `n <= 0` is a reasonable base case

IV. It must always be possible to reach the base case within a finite number of recursive calls.

    each call to `getNumberEqual` reduces the index `n` by 1 hence it ultimately must reach zero
3. Binomial Coefficients Function

```c
int binCoeff ( int n, int k )
{
    int y1, y2;

    if ((k == 0) || (n == k)) return 1;
    else
    {
        y1 = binCoeff ( n-1, k );
        y2 = binCoeff ( n-1, k-1 );
        return y1 + y2;
    }
}
```

a) Manually trace the algorithm

```
binCoeff (5, 3)

binCoeff (4, 3)  binCoeff (4, 2)

binCoeff (3, 3)  binCoeff (3, 2)
    1

binCoeff (2, 2)  binCoeff (2, 1)
    1

binCoeff (1, 1)  binCoeff (1, 0)
    1  1

binCoeff (3, 2)

binCoeff (2, 2)  binCoeff (2, 1)
    1

binCoeff (1, 1)  binCoeff (1, 0)
    1  1

binCoeff (3, 1)

binCoeff (2, 2)  binCoeff (2, 1)
    1

binCoeff (1, 1)  binCoeff (1, 0)
    1  1

binCoeff (2, 0)
    1
```

```
b) Compute binCoeff (5, 3) by using the tree in section a) above. 10

c) Count the number of recursive calls to compute binCoeff (5, 3)

Strictly speaking, without counting the original call, i.e., binCoeff (5, 3), there are nineteen (19) function calls; however, there are only nine (9) recursive calls; see the yellow functions. The underlined functions are terminal calls which are returns from the recursive calls. Depending on how one counts there are either eight, nine or nineteen recursive calls. During processing, all of the functions are on the stack.