CPSC 221: Sample Exam Problems (solutions)

Last Updated: December 3, 2008

Due: NOT TO BE TURNED IN.

Pointers and Dynamic Data

1. (a) `int *p = NUM;`

   Pointer is set to a constant value, which can cause a segmentation fault in the next statement.

   (b) `x = *q;`

   Dereferencing a null pointer.

   (c) `&x = NUM;`

   Address of a variable cannot be changed.

2. (a) `for (int i = 0; i <= 10; i++)`

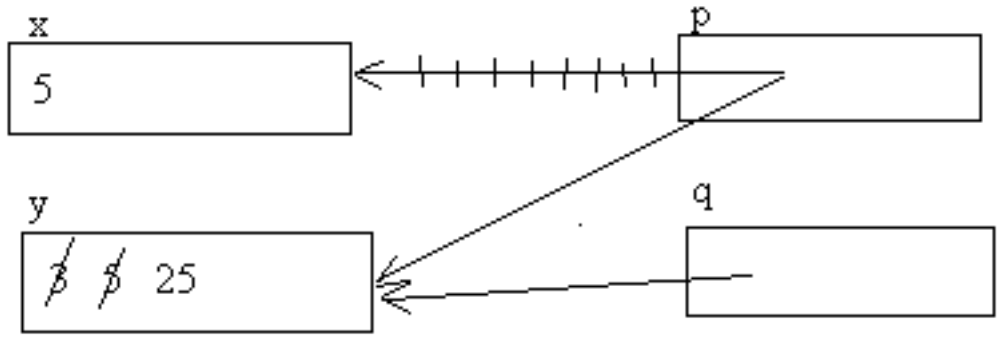
   Can cause segmentation fault. Should be: `i < 10;`

   (b) `if (0 <= n && n <= 10)`

   Can cause segmentation fault. Should be: `n < 10`

   (c) `return a + n;`

   Returns an address of a local variable. This space may be overwritten.

3. 


4. ```
   int* A;
   A = new int[100];
   for (int i=0; i < 100; i++)
     A[i] = 0;
   ```

5. ```
   SomeClass::~SomeClass() {
     delete [] sa;
     delete [] ia;
   }
   ```

Linked Structures

1. ```
   int length( node* head) {
     int count=0;
     while ( head != NULL ) {
       count++;
       head = head->next;
     }
     return count;
   }
   ```

2. ```
   node* concat( node* list1, node* list2 ) {
     if (list1 == NULL)
       return list2;

     node* curNode = list1;
     while( curNode->next != NULL ) {
       curNode = curNode->next;
     }

     curNode->next = list2;
     if (list2 != NULL) {
       list2->prev = curNode;
     }

     return list1;
   }
   ```

3. ```
   void add_ordered( node* & head, int it) {
     node* cur = head;
     node* prev = NULL;
     while ( cur && cur->item <= it ) {   // find the right position
       prev = cur;
       cur = cur->next;
     }
     node* p = new node;                   // create new node
     p->item = it;
     p->prev = prev;
     p->next = cur;

     if ( cur != NULL ) {
       cur->prev = p;
     }
     if ( prev != NULL ) {
       prev->next = p;
     } else {                             // insertion at the front
   ```
   ```

```
      head = p;
    }
  }
```

4. ```
   int count( node* root ) {
     if ( root == NULL )
       return 0;
     else
       return ( count(root->left) + count(root->right) + 1 );
   }
   ```

Data Structures, etc.

1. Use a doubly linked list. Why? We need to update the node before the last.

2. ```
   int height( Bnode* node) {
     if ( node == NULL || (node->left == NULL && node->right == NULL) )
       return 0;
     int leftHeight = height( node -> left );
     int rightHeight = height( node -> right );
     if (leftHeight > rightHeight)
       return leftHeight + 1;
     else
       return rightHeight + 1;
   }
   ```

3. If the linked lists do not need to be sorted then the worst-case running time is $O(n)$ (insert at the head). If they must be in sorted order, then the worst-case running time is $O(n^2)$.

4. To count the number of 1's in A, we do binary search on each row of A to determine the position of the last 1 in that row. Then we simply sum up these values to obtain the total number of 1's in A. This takes $O(\log n)$ time to find the last 1 in each row. Done for each of the n rows, then this takes $O(n \log n)$ time.

5. First, we sort the objects of A. Then, we can go through the sorted sequence and remove all duplicates. This takes $O(n \log n)$ time to sort and $O(n)$ time to remove the duplicates. Overall, this takes $O(n \log n)$ time.

Data Structures and Graph Theory

1. The algorithms are straightforward; you just need to account for the complexities of the given calls (i.e., the data structure itself has been analyzed for you).

   For Algorithm A: The open operation is $O(1)$. The while-loop is executed $O(u)$ times, and for each iteration: Inserting a 6-tuple in the ROCK-tree can be done in $O(6 \log n) = O(\log n)$ time. (Other operations in the while-loop are $O(1)$.) Thus Algorithm A runs in $O(u) * O(\log n) = O(u \log n)$ time. You could also write $O(u \log(n + u))$ time, since the tree will grow as you insert elements, but we know the number of updates is much less than n.

For Algorithm B: Searching the ROCK-tree takes $O(\log n)$ time to find the band's tuple, plus $O(6d^2 + x) = O(d^2 + x)$ time to return all tuples in the hypersphere centered at the band's location. This sums to $O(\log n + d^2 + x)$ time. Furthermore, regardless of whether or not the same call tells us the number of tuples in the hypersphere, an extra $O(x)$ factor for adding them up adds nothing to the complexity. Note that we can't simplify the expression further since n, d, and x are all unknowns at this point. It is true that x could be n in the worst-case, but we're after the expected number of tuples, so we should leave in the x.

Printing out all x tuples in the results list adds $O(x)$ time. (If the function had returned the band names instead of the tuples, it might have taken $O(\log n)$ time per band to look up their tuple.)

Thus Algorithm B runs in $O(\log n + d^2 + x) + O(x) = O(\log n + d^2 + x)$ expected time.

2. (a) $\left\lfloor \frac{4096}{30+10} \right\rfloor = 102$ data entries per leaf. (You could also say "4096 - 20" instead of "4096" and get 101 instead.)

   (b) $\lceil \frac{10000}{102} \rceil = 99$ leaf nodes plus $\lceil \frac{99}{102} \rceil = 1$ internal node equals 100 4K pages.

   (c) (a) $\left\lfloor \frac{4096}{30+5(4)} \right\rfloor = 81$ data entries per leaf.

   (b) $\lceil \frac{10000}{81} \rceil = 124$ leaf nodes. An internal node can hold $\left\lfloor \frac{4096-10}{30+10} \right\rfloor = 102$ keys and 103 associated pointers. Therefore, we estimate: $\lceil \frac{124}{103} \rceil = 2$ parents of leaves, plus one grandparent (the root). Conclusion: We need $1 + 2 + 124 = 127$ 4K pages.

3. (a) The band performs at Vancouver, Surrey, and Burnaby in one of 3! orderings. The band performs at the other four cities in one of 4! orderings. The band may decide to perform at 0,1,2, or 3 of Vancouver, Surrey, and Burnaby before going on tour. This is 4(4!)(3!) ways overall.

   (b) Visiting each vertex once, with the same start/end vertex is a Hamilton cycle (of the airport cities). A Hamilton cycle is possible since we can get from any city to any other. For example, one possibility is: Van-Tor-NY-LA-SF-Van.

   (c) Yes, S-B-V-Tor-NY-LA-SF. If they want to return to Vancouver, then the answer depends on the definition of "visit". If they can fly in/out of Vancouver without visiting Vancouver then the answer is "yes" otherwise "no".

4. We can take the adjacency matrix and turn it into a directed graph. An Euler cycle requires that we visit each edge exactly once, and return to the same city that we started from. Notice that there are no vertices with odd degree, and that there are as many outgoing edges as ingoing edges for each vertex. One Euler cycle is: V-S-L-N-T-L-T-N-L-V-L-S-V.

5. (a) The graph is a tree, because there is a unique simple path between any two vertices. Choosing any vertex except the center vertex as the root will result in a 3-ary tree. Choosing the middle right-most vertex as the root will result in a balanced tree.

   (b) The graph is not a tree, because it contains a cycle. Since the graph is not a tree, it can be neither a 3-ary rooted tree nor a balanced tree.

6. We know that $m \leq \binom{n}{2} = n(n-1)/2$ is $O(n^2)$. It follows, therefore, that $\log m = O(\log(n^2)) = O(2 \log n) = O(\log n)$.

7. Let F be the set of faces in a planar embedding of G. Since G is simple and $|V| \geq 3$, all faces in F have at least 3 bounding edges. Let boundary(f) be the number of edges bounding the face f. So,

$$\sum_{f \in F} \text{boundary}(f) \geq 3|F|.$$

Since every edge bounds two faces,

$$2|E| = \sum_{f \in F} \text{boundary}(f).$$

Thus $2|E| \geq 3|F|$. By Euler's formula, $|V| - |E| + |F| = 2$, so $|V| - |E| + 2|E|/3 \geq 2$ or $|E| \leq 3|V| - 6$.