

CS 14 - Summer 2003 - Quiz 4

July 23, 2003

1 True/False

Circle *T* or *F* as appropriate.

1. **T F** Hash tables using linear probing to resolve collisions have a worst case *search* time of $O(n)$, the number of items in the table. - Worst case: everything hashes to the same value. Shouldn't happen, but nothing prevents it from happening in the general case.
2. **T F** Hash tables using quadratic probing to resolve collisions have a worst case *search* time of $O(n)$, the number of items in the table. - Worst case: exactly the same as above.
3. **T F** If a hash is initially allocated small and expands by doubling when nearing the acceptable capacity threshold, the *average – case* time for insertion is $O(n) - O(lgn)$ - We do n operations at constant time, and lgn operations in $O(n)$ time for the expansion, for a total of $O(nlgn)$ for inserting n items. Take the average of that, we get $O(lgn)$.
4. **T F** If a hash is initially allocated small and expands by doubling when nearing the acceptable capacity threshold, the *worst – case* time for insertion is $O(n)$ - Takes linear time to resize
5. **T F** A good hash function should produce values that are all prime. - This would mean that values like 4, 6, 8 would never be chosen as hash values. This doesn't have uniformity.
6. **T F** A graph consists of a set of *vertices*, and a set of *edges* connecting those, and instructions for where to draw the vertices. - The graphical representation of a graph isn't mathematically important.
7. **T F** A graph is *complete* if every pair of vertices have an *path* between them. - A graph is *connected* if there is a path between each of them.
8. **T F** Order is important for the nodes in the linked-lists in an adjacency-list representation of a graph. - In general, this is not true. Some slight optimizations could be made if it were true, but they wouldn't actually alter the running time of the graph operations.

2 Multiple Choice

Circle the letter of the correct answer. Running-time questions are all in terms of input size n

1. Which of the following is *not* an important feature of a good hash function:
 - (a) Uniform output distribution
 - (b) Good “spread” (nearby inputs get distributed far apart)
 - (c) **Random output** - This would mean hashing one value twice could get you two different results!
 - (d) Fast running time
2. A binary search tree can be thought of as a ___ with a value at each node
 - (a) Directed, weighted, acyclic¹ graph
 - (b) Undirected weighted graph
 - (c) **Directed, unweighted, acyclic graph**
 - (d) None of the above
3. An adjacency matrix for n nodes
 - (a) **Uses $O(n^2)$ space and gives $O(1)$ lookup to find edges**
 - (b) Uses $O(n)$ space and gives $O(1)$ lookup to find edges
 - (c) Uses $O(n^2)$ space and takes $O(n)$ lookup to find edges
 - (d) Uses $O(n)$ space and takes $O(n)$ lookup to find edges

3 Short Answer

If you want the option to request a regrade available, your answers to these questions must be made in pen.

1. A “perfect hash function” is a special kind of hash that can be created when you know ahead of time exactly which keys are going to be inserted in the table. Describe in English why a perfect hash is useful. (Think about it for a while and take your time. A correct answer to this should be about 2-4 sentences.)

With a perfect hash, we can make a perfect 1-1 mapping of keys and hash values. With this information, we can completely prevent collisions, eliminating the need for linked-lists, linear/quadratic probing, and dirty-bits. All operations will now be $O(1)$, because we can pre-emptively size the table to be just the right size for the values that will be inserted. We also get perfect memory efficiency.

¹No cycles

2. A graph representing the prerequisites for the CS courses here at UCR could have each course as a vertex and a directed edge from node a to node b if a is a prereq for b . (Thus, anyone can take a course with no incoming edges in this graph.)

(a) Given the following list of prereqs, draw the appropriate graph:

- CS 10 comes before CS 12
- CS 12 comes before CS 14
- CS 10 comes before CS 61
- CS 14 comes before CS 141
- CS 141 comes before CS 100
- CS 20 has no prereqs and is not required for any other course.

See http://www.cs.ucr.edu/cs14-p/cs14_03sum/quiz4graph.png

(b) Which of the following terms apply to the graph you constructed (mark yes or no for each)

- Directed - Yes
- Acyclic - Yes
- Connected - No (CS 20 not reachable from any other node)
- Complete - No

(c) Is it reasonable to have a graph representing prereqs containing a cycle? Describe why or why not.

If a prereq graph contained a cycle, then nobody could enroll in any of those courses. Prerequisites are *transitive*, meaning if a is a prereq for b and b is a prereq for c , a is effectively a prereq for c . If there is a cycle, then by transitivity everything in that cycle is a prereq for itself, preventing anyone from enrolling for the first time.

(d) Is it reasonable to have a graph representing prereqs that is connected? Describe why or why not.

Yes, this would simply mean that there was a set of nodes S such that every node not in S has a member of S as a prereq. (Essentially, there are a few nodes which allow you to get to every other node.)

3. Write *bool hashInsert(pair<string, bool>* table, unsigned int tableSize, string toInsert)*, given a hash function *unsigned int hashString(string toHash)*. You may assume that *tableSize* is much larger than the number of elements that will be inserted. Your function should return true unless *toInsert* already exists in the table. Use **quadratic probing**. (Note that the *bool* values in *table* represent the “dirty-bits”, and that the only *pair* operations you need are *table[i].first* and *table[i].second*.)

```
bool hashInsert(pair<string, bool>* table, unsigned int tableSize, string toInsert)
{
    unsigned int k = hashString(toInsert);
    unsigned int firstSpot = tableSize + 1;
    unsigned int offset = 0;
    while (table[(k + offset * offset) % tableSize].second)
    {
        if ((table[(k + offset * offset) % tableSize].first == "") && (firstSpot > tableSize))
            firstSpot = (k + offset * offset) % tableSize;
        if ((table[(k + offset * offset) % tableSize].first == toInsert))
            return false;
        offset++;
    }
    table[(k + offset * offset) % tableSize].first = toInsert;
    table[(k + offset * offset) % tableSize].second = true;
    return true;
}
```