CMSC–341 Data Structures
Fall 1999 1 November 1999 Exam 2 Review

The exam will be drawn entirely from the following questions. There will be no "surprise" or "trick" questions on the exam. Every question on the exam is in this handout except that some questions may be phrased differently or involve different data.

Exam 1 Questions

1. Every question from Exam 1. From the exam itself, **not** from the exam 1 review questions.

Binary Tree

2. Suppose a binary tree is implemented in an array `arr` such that `arr[i]` contains the element of the node in the `i`'th level-order position in the tree. The level-order position is the position the element would occupy if the tree were perfect. For example, the tree

```
      A
     / \
    /   \
   B     C
    \   /
     E D
```

could be represented in the array as:

```
index   0  1  2  3  4  5  6  7
value   #  A  B  C  #  E  D  #
```

where # represents an unfilled array slot.

Using that representation of a BT

(a) write a C++ function

```
template <class Comparable>
int depth(Vector<Comparable> & tree_array);
```

that returns the depth of the tree.

(b) write a C++ function

```
template <class Comparable>
void postorder(Vector<Comparable> & tree_array);
```

that does a post-order traversal of the tree. You may assume the existence of a template function `void visit(const Comparable &)`.

(c) write a C++

```
template <class Comparable>
int TPL(Vector<Comparable> & tree_array);
```

that returns the total path length (sum of the depths of all nodes) of the tree.

3. Given the existence of the `BinaryNode` class (defined on page 5), write a function

```
template <class Comparable>
int NPL(BinaryNode<Comparable> *);
```

that returns the null path length of the given node. You may assume that the caller of the function has access to the public methods of `BinaryNode`.

Hash Tables

4. Name two desirable properties of a good hash function.

5. What is a "collision" in a hash table?

6. What is the "clustering" problem in hash tables?

7. Describe the "division" and "multiplication" methods of generating hash values.

8. Describe the "separate chaining" and "open addressing" collision resolution strategies.

9. What do the terms "linear probing" and "quadratic probing" mean?

10. An open-addressing hash function is a function:

$$h : U \times \{0, 1, \ldots\} \rightarrow \{0, 1, \ldots, m - 1\}$$

where m is the size of the hash table and U is the set of key values. Using a hash-table size of 10, and $U = \{89, 18, 49, 58, 69\}$, write a hash function that uses linear probing. Use that hash function to generate the indices for the elements of U. Do the same using quadratic-probing.

11. The average time performance of insertion and searching operations on a hash table are $O(1)$. This is much better than the performance of a binary search tree for the same operations. Given this wonderful performance of hash tables compared to binary search trees, when would you want to use a binary search tree instead of a hash table?

12. In an open-addressing hash table using linear probing, the average number of probes for successful, S, and unsuccessful, U, search are

$$S \approx \frac{1}{2}\left(1 + \frac{1}{1 - \lambda}\right)$$

$$U \approx \frac{1}{2}\left(1 + \frac{1}{(1 - \lambda)^2}\right)$$

where λ is the load factor of the table.

Suppose you want a hash table that can hold at least 1000 elements. You want successful searches to take no more than 4 probes on average and unsuccessful searches to take no more than 50.5 probes on average. What is the maximum load factor you can tolerate in your hash table? If table size is to be prime, what is the smallest table you can use?

Priority Queues and Heaps

Note: You will be given the algorithm for merging leftist heaps.

13. Define "priority queue."

14. Define "binary heap."

15. Define "null path length" in a binary tree.

16. Define "leftist binary tree."

17. Define "leftist heap."

18. Insertion and deletion in a binary heap is in $O(lg\, n)$ on average. Explain why this is so.

19. Finding the minimum element in a binary heap (a min-heap) is in $O(1)$ worst case. Explain why this is so.

20. Prove that the largest element in a min binary heap is a leaf.

21. For a min binary heap of `N` elements, what is the range of indices in which the maximum element will be found?

22. Describe, in English, an algorithm to find the largest element in a binary min-heap. What is the asymptotic worst case performance of your algorithm?

23. The array representing a binary heap contains the following elements in the order $2, 8, 3, 10, 16, 7, 18, 13, 15$. Show the order that results at **each** stage of inserting the element 4.

24. The array representing a binary heap contains the following elements in the order $2, 8, 3, 10, 16, 7, 18, 13, 15$. Show the order that results at **each** stage of deleting the minimum element.

25. Describe, in English, the process for constructing a binary heap from a given set of initial values (*i.e.*, heapify an array of elements).

26. Construct a binary heap using the initial values $18, 2, 13, 10, 15, 3, 7, 16, 8$. Show the heap at **each** stage of construction.

27. Given a drawing of a binary tree, state if it is a leftist tree and if it is a leftist heap. Give reasons.

28. Prove that any complete binary tree is a leftist tree.

29. Prove: for any leftist tree having N vertices, the number of vertices, R, on the rightmost path to a non-full vertex is given by
$$R \leq \lfloor \lg(N+1) \rfloor$$

30. Describe how to do `findMin` in a leftist heap.

31. Using the `merge` (*aka* `meld`) operation, describe how to do the `insert` and `deleteMin` operations for leftist heaps.

32. Give pseudocode for constructing a leftist heap from a given set of values. Your code must construct a leftist heap of N elements in $O(N)$ time (**not** in $O(N \lg N)$).

33. Given drawings of two leftist heaps H1 and H2, draw the leftist heap that results from the operation `merge(H1,H2)`.

AVL Trees

Note: You will be given the rules for insertion and deletion in AVL trees.

34. Define *AVL tree*.
35. Show the result of inserting `2,1,4,5,9,3,6,7` into an initially empty AVL tree (show the tree at the end of **each** insertion).
36. Show the result of deleting a given node in the tree.
37. What is the "Big-Oh" performance (in terms of the number of nodes in the tree) for each operation `find`, `insert`, and `remove` for AVL trees in the best, worst, and average cases?
38. What property of AVL trees is most significant in explaining their "Big-Oh" behavior for the operations `find`, `insert`, and `remove`?
39. Prove that every complete binary tree is AVL-balanced.

Red-Black Tree

Note: You will be given the rules for insertion and deletion in Red-black trees.

40. Define *Red-Black tree*.
41. Show the result of inserting `2,1,4,5,9,3,6,7` into an initially empty Red-Black tree (show the tree at the end of **each** insertion).
42. Show the result of deleting a given node in the tree.
43. What is the "Big-Oh" performance (in terms of the number of nodes in the tree) for each operation `find`, `insert`, and `remove` for Red-Black trees in the best, worst, and average cases?
44. What property of Red-Black trees is most significant in explaining their "Big-Oh" behavior for the operations `find`, `insert`, and `remove`.

Splay Tree

Note: You will be given the rules for insertion and deletion in Splay trees.

45. Define *Splay tree*.
46. Show the result of inserting `2,1,4,5,9,3,6,7` into an empty Splay tree (show the tree at the end of **each** insertion).
47. Show the result of deleting a given node in the tree.
48. What does the following statement mean?

> "a splay tree has $O(n \lg n)$ amortized performance over sequences of `insert`, `remove`, and `find` operations"

In particular, what does '`n`' mean?

Definition of `BinaryNode` Class

```
template <class Comparable>
class BinaryNode
{
public:
  BinaryNode(const Comparable & theElement,
     BinaryNode * lt, BinaryNode * rt);
  BinaryNode(const BinaryNode & rhs);
  ~BinaryNode();
  const BinaryNode & operator=(const BinaryNode & rhs);

  BinaryNode * getLeftChild();
  BinaryNode * getRightChild();
  const Comparable & getElement();
private:
  Comparable   _element;
  BinaryNode * _left;
  BinaryNode * _right;
};
```