EECS 281, Week 8: Wednesday

Hash Tables

1. Describe hashing as a two-step process.

   We first compute the hash code by converting the key that we use in the hash table to an integer. Then we compress the hash code to range [0, m), where m is the number of slots in the array used by the hash table.

2. Consider the hash function for a `string` below:

```
int hash(const string &s) {
   return toupper(s[0]) - 'A';
}
```

   Critique this hash function, identifying and explaining its downsides.

   Since the hash function bases its output only on the first letter of its input, it will tend to output some values more frequently than others if we use it on English words (some letters are more common at the start of English words than others).

   In addition, this hash function not distinguish between lowercase and uppercase.

3. What is a collision?

   A collision occurs when two keys map to the same index in the array representing the hash table.

4. When can collisions occur?

   At the step of creating the hash code, so that two different keys map to the same hash code. Then, no matter which compression function we use, the hash code will compress to the same slot in the array.

   The compression method can cause clusters if the keys that we insert have patters and the size of the hash table is not a prime number.

5. How are collisions different from clusters?

   Clustering is the tendency for a collision resolution scheme to create long runs of filled slots.

6. What are the properties of a good hash function?

   Uniformly distributing some (possibly non-uniform) domain over a range.

   Generates very different hash codes for inputs that only differ slightly.

   Employs only very simple, quick operations such as bitwise manipulations.

7. What is a good way to generate hash codes for a string?

   $h(s) = s_0 \cdot a^{n-1} + s_1 \cdot a^{n-2} + \ldots + s_{n-1} \cdot a^0$, where where a is a constant > 1.

8. What is load factor α?

   Load factor $\alpha = \frac{n}{m}$, where n is the number of elements stored in the hash table and m is the size of the underlying array.

EECS 281, Week 8: Wednesday

9. What is dynamic hashing?

   Dynamic hashing refers to growing (and shrinking) the hash table when its load factor exceeds the maximum load factor (or gets below some minimum threshold).

10. What is simple uniform hashing?

    Simple uniform hashing means that the probability of element i hashing to slot k is $\frac{1}{m}$.

11. Suppose we use a hash function h to hash n distinct keys into an array of length m. Assuming simple uniform hashing, what is the expected number of collisions?

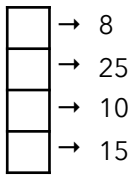
    The probability that elements i and j both hash to the same slot is $\frac{1}{m}$. We use linearity of expectation to sum over all possible pairs i and j:

    E[number of colliding pairs] = $\sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{m} = \Theta\left(\frac{n^2}{m}\right)$.

12. Demonstrate what happens when we insert the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \mod 9$.

    TODO

13. Suppose we have a hash table that stores integer keys and uses chaining as its collision resolution technique. Assume that the hash code for an integer is that integer itself.

    Demonstrate what happens when we insert the keys 18 and 5. Be sure to resize the hash table by doubling the array when the load factor reaches 1.5.

14. When you modify a key that has been inserted into a hash table, will you be able to retrieve that entry again? Explain.
    A. Always
    B. Sometimes
    C. Never

    B. Sometimes. If the hash code for the key happens to change as a result of the modification, then we won't be able to reliably retrieve the key.

15. When you modify a value that has been inserted into a hash table, will you be able to retrieve that entry again? Explain.
    A. Always
    B. Sometimes
    C. Never

    A. Always. The bucket index for an entry in a hash table is decided by the key, not by the value. Mutating the value does not affect the lookup procedure.

16. (T/F) While a good hash table that contains N elements and uses separate chaining has average-case constant time for look-up, the worst case running time for a lookup is $\Theta(\log N)$.

    False

17. (T/F) `std::vector` can be used as a key in `std::unordered_map`.

    False

18. What do we need to do in order to use `std::unordered_map` with custom types?

    First, we need to implement a custom hash functor that overloads `operator()` or specialize the `std::hash` template. Second, we need to provide a comparison function for equality, either by specializing `std::equals` or by overloading `operator==` for the custom type.

19. (T/F) A hash table can be used to store a dictionary in a spell-checker program.

    True

20. Consider this definition of a hash table that stores English words and uses separate chaining as its collision resolution technique:

```
struct HashTableNode {
    string word;
    HashTableNode* next = nullptr;
};

class HashTable {
public:
    size_t size() const;
private:
    vector<HashTableNode*> slots;
}
```

    Suppose that `HashTable` has been implemented without an instance variable to keep track of the number elements. Complete the implementation of `size` method below in such a way that it returns the number of words stored in the hash table. Feel free to add other helper (private) functions.

```
size_t HashTable::size() const {

    size_t count = 0;
    for (size_t i = 0; i < slots.size(); i++) {
        HashTableNode* ptr = slots[i];
        while (ptr != nullptr) {
            count += 1;
            ptr = ptr->next;
        }
    }
    return count;
}
```

21. Consider the definition of a `TicTacToeGame` class.

```
class TicTacToeGame {
    public:
        // always 3-by-3
        const size_t size = 3;
        // hash code to use for std::hash()
        int hashCode() const;
    private:
        // either 'X', 'O' or ' '
        char tiles[size][size];
        // keep track of turns
        bool isXTurn = true;
};
```

Implement the `hashCode` methods that computes the hash code to specialize `std::hash` for the Tic-Tac-Toe game. Ensure that different board configurations have different hash codes.

```
int TicTacToeGame::hashCode() const {
    int code = 0;
    if (isXTurn) {
        code = 1;
    }
    for (size_t row = 0; row < size; row += 1) {
        for (size_t column = 0; column < size; column += 1) {
            code *= 3;
            char tile = tiles[row][column];
            if (tile == 'X') {
                code += 1;
            } else if (tile == 'O') {
                code += 2;
            }
        }
    }
    return code;
}
```

We multiply by 3 to uniquely identify the particular position for an element. Intuitively, we can think of `TicTacToeGame::hashCode` as taking a board like `"O XOOX O"` and turning it into a number like `202122102`.

22. How many different configurations of the Tic-Tac-Toe board are there?

$2 \cdot 3^9$

EECS 281, Week 8: Wednesday

Tries

23. (T/F) A trie is a binary tree.

    False

24. To use objects of type T as keys in a trie, what property must the type T have?

    It must be string-like (we must be able to break it into characters or components) and characters (components) must be from some alphabet of fixed size.

25. Suppose we have a trie that stores n English words, and each of trie's nodes contains an array of p pointers to other nodes. What is the time time complexity of looking up a word of size k in this trie?

    $O(k)$

26. (T/F) Tries can be used to store valid words (i.e., if they are in the dictionary), but cannot be used to store the words' definition.

    False

27. (T/F) Unlike hash tables, we cannot implement tries in a way to store duplicate keys.

    False

28. (T/F) Let T be a trie that stores N strings. We can traverse T in sorted order in $\Theta(N)$ time.

    True

29. (T/F) Let T be a trie that stores strings backwards, with the top of trie corresponding to the last, rather than the first character of the string. We have traverse T in sorted order in $\Theta(N)$ time.

    False. The characters at the end of a string say almost nothing about the string's ordering, so no ordinary traversal of the trie will find its keys in sorted order.

30. (T/F) We can you use a trie to do a range query, e.g., to get all strings between "maxim" and "pizza" in the dictionary in $\Theta(N)$ time.

    True

31. If we traverse the same node while searching for two strings, S_1 and S_2, in a trie and that node is at depth k in the trie (where the root is depth 0), what can we say about S_1 and S_2?

    (At least) the first k characters of S_1 and S_2 are the same.

32. Describe one disadvantage of using a trie to store a dictionary of English words?

    Tries tend to waste lots of memory since many nodes' pointers end up unused since there are far fewer English words than there are possible permutations of the English alphabet's letters.

33. Suppose we have a trie that stores *n* English words, and each of trie's nodes contains an array of *p* pointers to other nodes. Describe two ways to improve memory usage of this trie.

    In each node, use a hash table instead of an array to store pointers to other nodes. If multiple trie nodes in a row have only one child, compress the nodes into one, so that the single node represents multiple letters in the word instead of a single letter (e.g., "ght" instead of "g", "h" and "t").

34. Consider this definition of a case-insensitive trie for English words:

```
struct TrieNode {
    bool isWord;
    TrieNode* children[26];
};

class Trie {
public:
    size_t size() const;
private:
    TrieNode* root;
}
```

    Suppose that `Trie` has been implemented without an instance variable to keep track of the number of words in trie. Complete the implementation of `size` method below in such a way that it returns the number of words stored in the trie. Feel free to add other helper (private) functions.

```
size_t Trie::size() const {
    return countWords(root);
}

size_t Trie::countWords(TrieNode* node) {
    size_t count = 0;
    if (node != nullptr) {
        if (node->isWord) {
            count += 1;
        }
        for (size_t i = 0; i < ALPHABET_SIZE; i += 1) {
            count += countWords(node->children[i]);
        }
    }
    return count;
}
```

35. When should we use a hash table over a trie? Trie over a hash table?

    TODO

EECS 281, Week 8: Wednesday

Trees

36. What is the depth of a node?

    The length of the path from the node to the root of the tree.

37. Implement depth, a function that takes a `BinaryTreeNode*` and returns the depth of that node. Assume that `BinaryTreeNode` struct has a `parent` pointer.

```
int depth(BinaryTreeNode* node) {
    if (node->parent == nullptr) {
        return 0;
    } else {
        return 1 + depth(node->parent);
    }
}
```

38. What is the height of a node?

    The length of the path from the node to its deepest descendant. In other word, it is the longest path from the node to a leaf.

39. Implement height, a function that takes a `BinaryTreeNode*` and returns the height of the tree rooted that node. Assume that `BinaryTreeNode` struct has a `left` pointer and a `right` pointer, and that the height of an empty tree is -1 and that the height of a tree with one node is 0.

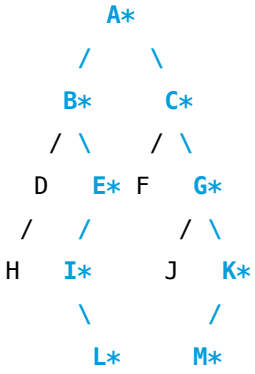
```
int height(BinaryTreeNode* node) {
    if (node == nullptr) {
        return -1;
    } else {
        return 1 + max(height(node->left), height(node->right));
    }
}
```

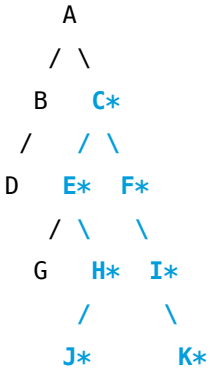
40. Complete the table below with amortized time complexities and the data structure generally used by the STL for these containers.

Operation	set	map	unordered_set	unordered_map
insert				
find				
remove				
Data structure				

EECS 281, Week 8: Wednesday

41. The *diameter* of a tree is the maximum number of edges on any path connecting two nodes of the tree. For example, here are two sample trees and their diameters. In each case the longest path is shown in blue with stars next to all nodes in the path. Note that there can be more than one longest path.





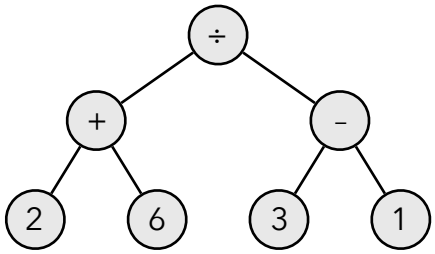
Implement the function `diameter` that computes the diameter of a binary tree represented by a pointer to an object of `BinaryTreeNode` class.

```
int diameter(BinaryTreeNode* node) {
    if (node == nullptr) {
        return 0;
    } else {
        int childDiameter = max(diameter(node->left), diameter(node->right));
        int nodeDiameter = 2 + height(node->left) + height(node->right);
        return max(childDiameter, nodeDiameter);
    }
}
```

42. (T/F) The height of a binary tree with N elements is $\log N$.

    False

43. What are pre-order, in-order, post-order and level-order traversals of this tree?



    Pre-order: ÷ + 2 6 - 3 1

    In-order: 2 + 6 ÷ 3 - 1

    Post-order: 2 6 + 3 1 -

    Level-order: ÷ + - 2 6 3 1

44. (T/F) All four traversals can be performed on all trees.

    False. In-order traversal can be performed only on binary trees.

45. (T/F) For any non-empty binary tree with A leaves and B nodes of degree 2, $A = B + 1$.

    True

46. (T/F) For any non-empty binary tree with A nodes and B nodes of degree 2, $A = B + 1$.

    False

47. (T/F) In-order traversal of a binary search tree will visit the nodes in sorted order.

    True

48. A binary tree has a post-order traversal D A B E C and an in-order traversal D E B A C. Draw the binary tree and give its pre-order traversal.

    TODO

49. (T/F) BSTs are balanced trees.

    False

50. (T/F) Searching for an element in a binary search tree with N nodes where each node has either 0, 1 or 2 children takes $\Theta(N)$ time in the worst case.

    True

51. (T/F) Searching for an element in a binary search tree with N nodes where each node has either 0 or 2 children takes $\Theta(\log N)$ time in the worst case.

    False

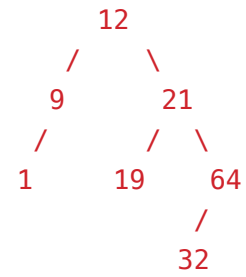
52. (T/F) In a binary search tree, all internal nodes, except the ones at the last level, have two children.

    False

53. What is the purpose of AVL trees?

    AVL trees are binary search trees that are guaranteed to be balanced. They guarantee $O(\log n)$ operations on the tree.

54. Insert the following values into an empty AVL tree: 9, 21, 64, 12, 1, 19, 32. Then remove 1 and 21.



55. Insert the following values into an empty AVL tree: 21, 32, 64, 72, 17. Then remove 21, 32, 64, 72 and 17.

EECS 281, Week 8: Wednesday

Graphs and Graph Algorithms

56. Describe the differences between trees and graphs.

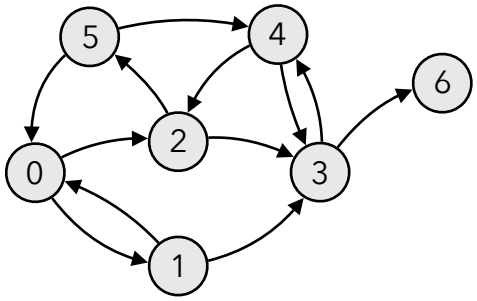
    A tree is an acyclic graph that has only one path between any two vertices.

57. Breadth-first search traversal of a graph is equivalent to which traversal of a tree?

    Level-order traversal.

58. Describe an algorithm for finding connected components in a graph.

Consider the following graph for the next three questions:



59. Is the graph weighted? Directed? Cyclic?

    Unweighted, directed, cyclic.

60. Give the adjacency matrix representation for the graph shown above.

61. Give the adjacency list representation for the graph shown above.

62. Give the (pre-order) DFS traversal for the graph shown above, starting at node 0.

    0, 1, 3, 4, 2, 5, 6

63. Give the post-order DFS traversal for the graph shown above.

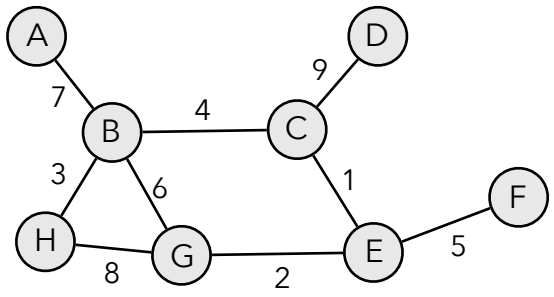
    5, 2, 4, 6, 3, 1, 0

64. Give the BFS traversal for the graph shown above.

    0, 1, 2, 3, 5, 4, 6

EECS 281, Week 8: Wednesday

Consider the following undirected graph for the next five questions:



65. Give the sequence of vertices added to the minimum spanning tree for the above graph when running Prim's algorithm. Start with vertex C.

    C, E, G, B, H, F, A, D.

66. Give the sequence of vertices added to the maximum spanning tree for the above graph when running Prim's algorithm. Start with vertex C.

    TODO

67. Give the sequence of edges added to the minimum spanning tree for the above graph when running Kruskal's algorithm.

    C–E, E–G, B–H, B–C, E–F, B–G, A–B, H–G, C–D.

68. Give the sequence of edges added to the maximum spanning tree for the above graph when running Kruskal's algorithm.

    TODO

69. Give the sequence of edges that would be "explored" when running Dijkstra's algorithm to find the shortest path from node D to node H.

    D–C, C–E, E–G, B–C, E–F, B–H.

70. Prim's, Kruskal's and Dijkstra's are considered _____ algorithms because they solve the problems by progressively making the locally optimal choice at each stage or iteration.

    Greedy

71. (T/F) Dijkstra's algorithm always finds the shortest path on any graph.

    False. It does not work with negative edges.

72. An airline is attempting to restructure its services so that that every city in its network can connect to any other city while minimizing the total length of its routes. What problem is the airline trying to solve?

    Minimum spanning tree

73. An airline's network is set up so that most cities have flights to only a few destinations, and a few cities are hubs, with connections to many destinations. Given this setup, which algorithm should the airline use to create its new network?

    Kruskal's Algorithm

74. An airline wants to generate a minimum spanning tree of its network and represents it with the following distance matrix:
    Could this be a valid solution to the airline's MST? Explain.

    No. The graph has a cycle, so it is not a valid (spanning) tree.

75. Suppose that a directed weighted graph is represented with an adjacency matrix. Implement `reverse`, a function that would replace all edges (*v*, *w*) with (*w*, *v*).

```
void reverse(vector<vector<int>> &graph) {
```

Disjoint Sets

76. What operations can disjoint sets perform efficiently?

    Union and find

77. How can we represent disjoint set with an array?

    To represent the tree as an array, we make as assumption that the items in the universe are integers from 0 to n – 1. We use an array of size n and each item i is represented by the element in the array at index i. We record the parent of that item in the array.

78. What optimizations can we perform on disjoint sets?

    Path compression, union by size

79. (T/F) Disjoint sets can be used with Prim's algorithm.

    False

80. (T/F) Disjoint sets can be used with Kruskal's algorithm.

    True

EECS 281, Week 8: Wednesday

Algorithm Design

81. What do C, L, R and S stand for in CLRS?

    Cormen, Leiserson, Rivest and Stein, the authors of CLRS.

82. (T/F) Any divide-and-conquer algorithm is an example of a dynamic programming algorithm.

    False

83. (T/F) Merge sort and Quicksort are considered combine-/divide-and-conquer algorithms rather than dynamic programming because they split the work into overlapping subproblems.

    False. They split work into non-overlapping subproblems.

84. Top-down dynamic programming trades _____ for _____.

    Memory; speed

85. Suppose you wanted to know the number of possible subsets of *N* tasks that cost less than a certain amount in total. What type of algorithm should you use? Explain.

    Brute Force always works, though you should not use it when other options exist.

    Dynamic programming would help optimize the algorithm by saving solutions to subproblems that we have already computed.

    Backtracking would help optimize the algorithm by eliminating partial solutions once we determine that they cannot lead to a correct complete solution.

86. What would be the worst case asymptotic time complexity of the algorithm in the previous question?

    $\Theta(2^N)$. It is possible that the result will be all of the subsets of tasks.

87. Suppose you wanted to know the subset of *N* tasks that contains the most tasks while costing less than a certain amount. What type of algorithm should you use? Explain.

    Greedy. Keep adding the cheapest task remaining until adding the next task drives the cost over the bound.

88. What would be the worst case asymptotic time complexity of the algorithm in the previous question?

    $\Theta(N \log N)$, since we need to sort the tasks, or $\Theta(kN)$, where k is the number of tasks in the solution.

89. What is the difference between backtracking and branch-and-bound?

    Backtracking and branch-and-bound use similar approach to solving problems: a solution tree where each level represents a step of solving the problem and where leaves represent possible solutions. Backtracking solves constraint-satisfaction problems (e.g., N-queens and sudoku), where the solution must satisfy some constraints. We might care about finding a single solution or all possible solutions. On the other hand, branch-and-bound solves optimization problems and we care about finding the solution that minimizes some cost or maximizes some benefit. The constraints that we use update as we find better solutions to the problem.

90. Implement `countDuplicates`, a function that counts the total number of duplicate numbers in a vector. For example, in a vector with elements {183, 490, 381, 281, 381, 281, 280, 281}, the total number of duplicates is 3 (2 for 281 and 1 for 381).

```
int countDuplicates(const vector<int> &numbers) {
    int count = 0;
    unordered_map<int, int> counts;
    for (auto number : numbers) {
        counts[number]++;
    }
    for (auto &countForNumber : counts) {
        count += countForNumber.second - 1;
    }
    return count;
}
```

91. Write an efficient function to find the sum of contiguous elements in a subarray within a one-dimensional vector of integers which has the largest sum. What is the time complexity? What is the space complexity? What kind of algorithm is it?

```
int findMaxSubarraySum(const vector<int> &numbers) {

    int maxSoFar = 0;
    int maxUpToI = 0;
    for (size_t i = 0; i < numbers.size(); i += 1) {
        maxUpToI = maxUpToI + numbers[i];
        if (maxUpToI >= maxSoFar) {
            maxSoFar = maxUpToI
        }
        if (maxUpToI < 0) {
            maxUpToI = 0;
        }
    }
    return maxSoFar;
}
```

92. State the N-Queens problem. How to solve it?

    TODO

93. State the Knapsack problem. Then solve it with backtracking and branch-and-bound.

    TODO

94. Implement isSubsetSum, a function that solves the Subset Sum problem: given a set of *n* integers *S* and a value *W*, determine if there is a subset of integers that sum to *W*.

```
bool isSubsetSum(const vector<int> &numbers, int sum) {
```

95. Given a text and a wildcard pattern, implement wildcard pattern matching algorithm that finds if wildcard pattern is matched with text. The matching should cover the entire text (not partial text). The wildcard pattern can include the characters `?` (matches any single character) and `*` (matches any sequence of characters, including the empty sequence).

```
Text = "baaabab",
Pattern = “*****ba*****ab", output : true
Pattern = "baaa?ab", output : true
Pattern = "ba*a?", output : true
Pattern = "a*ab", output : false
```

```
int match(const string &text, const string &pattern) {
```

96. Given a weighted graph (whose edges can have negative weights) represented with an adjacency matrix, find the lengths of the shortest paths between all pairs of vertices. Print the lengths of all shortest paths at the end of the function.

    Hint: this is Floyd–Warshall algorithm.

```
void shortestPaths(const vector<vector<int>> &graph) {
```