**Data Structures**

**Fall 2019**

**Exam 2**

**Section 0 – Student Information**

s0\_name\_and\_id.py

**Open the file and make sure you type your last name, first name, and UTEP ID**

**DON’T FORGET THIS SECTION**

**DON’T FORGET THIS SECTION**

**DON’T FORGET THIS SECTION**

**DON’T FORGET THIS SECTION**

**DON’T FORGET THIS SECTION**

**DON’T FORGET THIS SECTION**

**Section 1 - Multiple Choice Questions**

s1\_multiple\_choice.py

**Problem 1 (4 points):** We say that a binary tree T is height-balanced if:

1) Left subtree of T is balanced

2) Right subtree of T is balanced

3) The difference between heights of left subtree and right subtree is not more than 1.

Which of the following trees is not a height-balanced binary tree?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Option 0: | Option 1: | Option 2: | Option 3: | Option 4:  All of them are height balanced binary trees |

**Problem 2 (4 points):** Consider the following B-Tree.

A close up of a piece of paper

Description automatically generated

What would *print(self.root.children[-3].keys[-1])* print?

Option 0: 1

Option 1: 2

Option 2: 4

Option 3: 5

Option 4: 7

Option 5: 8

Option 6: 10

Option 7: None of the above

**Problem 3 (4 points):** Consider the following heap.

A picture containing object, watch, clock

Description automatically generated

What would *print(self.tree[2])* print?

Option 0: 8

Option 1: 5

Option 2: 6

Option 3: 2

Option 4: 3

Option 5: 1

Option 6: 4

Option 7: None of the above

**Problem 4 (4 points):**Show the configuration of an initially empty hash table of size 4 that solves collisions by chaining after performing the following sequence of operations.

ht = HashTable(4)

ht.insert(12)

ht.insert(0)

ht.insert(1)

ht.insert(3)

ht.insert(5)

ht.insert(7)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Option 0:    0: [12, 0]  1: [5, 1]  2: []  3: [7, 3] | Option 1:  0: [12, 0]  1: [5, 1]  2: []  3: [3, 7] | Option 2:  0: [0, 12]  1: [1, 5]  2: []  3: [3, 7] | Option 3:  0: [12, 0]  1: [1, 5]  2: []  3: [3, 7] | Option 4:  Answer is not listed |

**Problem 5 (4 points):** Consider an initially empty max-heap. Show the result of performing the following sequence of insertions: 4, 2, 5, 1, 0, 10

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Option 0:  [10, 2, 5, 1, 0, 4] | Option 1:  [10, 2, 5, 0, 1, 4] | Option 2:  [10, 2, 5, 1, 4, 0] | Option 3:  [10, 2, 5, 4, 1, 0] | Option 4:  Answer is not listed |

**Problem 6 (4 points):** What is the running time of the *extract* operation in a max-heap?

|  |  |  |  |
| --- | --- | --- | --- |
| Option 0:    O(1) | Option 1:  O(log n) | Option 2:  O(n) | Option 3:  O(n log n) |

**Section 2 – Binary Search Trees**

s2\_binary\_tree\_section.py

**Problem 7 (6 points):** Complete the implementation of the \_*sum* method. This method sums the values of all of the nodes in the binary search tree (The parameter *node* represents the root of the [sub]tree)*.* If the three is empty, return 0.

**Problem 8 (6 points):** Complete the implementation of the *\_sum\_at\_depth* method. This method returns the sum of the values of all the nodes at depth *d* in the binary search tree (The parameter *node* represents the root of the [sub]tree)*.*  If there are no nodes at the specified depth, return 0.

**Problem 9 (6 points):** Complete the implementation of the *\_max\_val* method. This method returns the largest value in a binary search tree. (The parameter *node* represents the root of the [sub]tree)*.*  If the three is empty, return -math.inf.

**Problem 10 (6 points):** Complete the implementation of the *\_search* method. This method returns the **node** where *k* is in the three, or None if *k* is not in the tree. (The parameter *node* represents the root of the [sub]tree)*.*  Your method needs to make use of the special property of binary search threes (left child < parent < right child). If you do not take advantage of this property, you will get a 0.

**Section 3 – B-Trees**

s3\_btrees\_section.py

**Problem 11 (6 points):** Complete the implementation of the \_*height* method. This method computes the height of a b-tree (The parameter *node* represents the root of the [sub]tree)*.*  The height of an empty tree is -1. The height of a tree with only one node is 0.

**Problem 12 (6 points):** Complete the implementation of the *\_num\_nodes* method. This method returns the number of nodes in the b-tree (The parameter *node* represents the root of the [sub]tree)*.*

**Problem 13 (6 points):** Complete the implementation of the *\_sum\_at\_depth* method. This method returns the sum of all keys in the b-tree that are stored in nodes that have depth *d* (The parameter *node* represents the root of the [sub]tree)*.*  If there are no nodes at the specified depth, return 0. Consider this B-Tree as an example:

A close up of a piece of paper

Description automatically generated

Example 1: If d = 1 -> Return 1 + 2 + 4 + 5 + 7 + 8 +10

Example 2: If d = 0 -> Return 3 + 6 + 9

**Problem 14 (6 points):** Complete the implementation of the *\_contains* method. This method returns True if *k* is in the three, or False if *k* is not in the tree. (The parameter *node* represents the root of the [sub]tree)*.*

**Section 4 - Hash Tables**

s4\_hash\_tables\_section.py

**Problem 15 (6 points):** Complete the implementation of the *insert* method. This method inserts *k* into the table if and only if it is not already in it.

**Problem 16 (6 points):** Complete the implementation of the *search* method. This method returns a reference to the list (bucket) where *k* is in the table. If k is not in the table, return None.

**Problem 17 (6 points):** Complete the implementation of the *min\_key* method. This method return the smallest key in the table. If the table is empty, return math.inf

**Problem 18 (6 points):** Complete the implementation of the *load\_factor* method. The load factor of a hash table is the number of elements in the table divided by the size of the table, or equivalently, the average length of the lists (buckets) in the table.

**Section 5 - Heaps**

s5\_heaps\_section.py

**Problem 19 (6 points):** Complete the implementation of the *max\_grandpa\_gap* method. The max grandpa gap in a max-heap is defined as the maximum difference between the value of a node and its grandchildren. If the heap does not have at least 4 nodes, return -math.inf.

Example:

A clock in the middle of a watch

Description automatically generated

Answer: max(10-7, 10-0, 10-4, 10-3, 9-(-3), 9-2) 🡪 12

**Problem 20 (6 points):** Complete the implementation of the *is\_valid* method. The purpose of this method is to verify that the contents of *self.tree* represent a valid max-heap. An empty heap is a valid heap.

**Problem 21 (6 points):** Complete the implementation of the *try\_replace* method. This methods receives integers *i* and *val* as input and replaces the item in self.tree at index *i* by *val* only if doing so maintains the heap property. You should do nothing if *i* is not a valid index.

**Problem 22 (6 points):** Complete the implementation of the method *create\_path*. This method receives an integer *i* as input and stores all the elements in the path from index *i* to the root of the heap in a list. You should return an empty list if *i* is not a valid index. Note that an empty list is not the same thing as None.

*Total numbers of points: 122 (graded out of 100)*