4M26 - Examples Paper 2 - Advanced Data Structures

1.

Briefly answer multiple questions below.

(i) Explain the general idea of a hash function. If the data you are working with involves string of characters, suggest a scheme that might be used to compute reasonable hash values.

Write your answer here.

(ii) Assume your hash table has a space for maximum of n records in it but only k are in use at some stage. You can assume that the open adressing scheme with double-hashing is used and the hash function has perfect statistical properties. Estimate the number of table accesses (probes) needed when inserting a new entry into the table.

Write your answer here.

(iii) Given a binary search tree and a key that is present in it explain an efficient procedure that can be used to update the tree so that the deletion of the given key becomes straight-forward. Indicate expected costs for your procedure. In your answer you can assume that the tree is well balanced. What if the tree is not balanced?

Write your answer here.

(iv) Suppose that the search for key, k, in a binary search tree ends up in a leaf. Consider three sets: A - the keys to the left of the search path; B - the keys on the search path; and C - the keys to the right of the search path. Is the following claim correct: *any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$*? Justify your answer.

Write your answer here.

(v) Given a binary search tree and a key that is present in it, describe a procedure that rearranges the tree so that the specified key is in the root of the updated tree. Is this always possible?

Write your answer here.

(vi) What is a *red-black tree*? What are the key properties of red-black trees and what problem do they solve?

Write your answer here.

(vii) Suppose that a priority queue is to be implemented using red-black trees. How can the smallest item in the queue be found? Starting from an empty queue how much work is required to build a queue with n items in it?

Write your answer here.

(viii) Give an upper and a lower bound for the number of nodes in B-tree of height, h, and degree, t. Hence if a B-tree holds n values find an upper bound on its height.

Write your answer here.

(ix) Explain how search works in a B-tree. In your answer you can ignore disk read and write operations.

Write your answer here.

2.

Write the function, **rolling_hash(S)**, which:

(i) takes a two element list, S, which contains two strings of length n and m correspondingly and;

(ii) finds the index of the first location in the string, $S[1]$, starting from which the string, $S[0]$, appears in string, $S[1]$, in full. If the first string is not a substring of the second string, your function should output, `-1` .

The expected runtime complexity of your method should be $O(m+n)$ when a substring can not be found and the expected number of collisions in the hash table is constant for a typical input.

Examples:

Input: `["abc","abcdefgh"]`
Output: `0`

Input: `["fgh","abcdefgh"]`
Output: `5`

Constraints:

- $1 \leq n \leq 1000.$
- $1 \leq m \leq 1000.$

Code:

In [ ]:
```
def rolling_hash(S):

    #Write your code here
```

Tests:

Run example test case 1:

In [ ]:
```
input_value = ["abc","abcdefgh"]
print (rolling_hash(input_value))
```

Run example test case 2:

In [ ]:
```
input_value = ["fgh","abcdefgh"]
print (rolling_hash(input_value))
```

3.

Write functions, **inorder**($node$), **preorder**($node$), **postorder**($node$), each of which traverses a binary search tree in the corresponding order.

You are provided with the function, **bst_manipulate**($insert\_keys$), which takes a list of keys as an input, creates a binary search tree (BST), using $BinarySearchTree$ class and calls the aforementioned BST traversal functions to be implemented. The list of three lists containing keys in the order of inorder, preorder and postorder traversal of the BST is returned as an output.

Examples:

Input: `[2, 1, 3]`
Output: `[[1, 2, 3],[2, 1, 3],[1, 3, 2]]`

Input: `[5, 3, 2, 7, 1, 8, 9, 12]`
Output: `[[1, 2, 3, 5, 7, 8, 9, 12],[5, 3, 2, 1, 7, 8, 9, 12],[1, 2, 3, 12, 9, 8, 7, 5]]`

Constraints:

- Keys are assumed to be unique, integer values.
- $1 \leq \text{len}(insert\_keys) \leq 100$.

Code:

In [ ]:
```
from utils.binary_search_tree import BinarySearchTree, Node, insert

def inorder(node):

    #Write your code here

def preorder(node):

    #Write your code here

def postorder(node):

    #Write your code here

def bst_manipulate(insert_keys):

    bst = BinarySearchTree()

    node_list = [Node(key) for key in insert_keys]

    for node in node_list:
        insert(bst, node)

    return [inorder(bst.root),preorder(bst.root),postorder(bst.root)]
```

Tests:

Run example test case 1:

In [ ]:
```
input_value = [5, 3, 2, 7, 1, 8, 9, 12]
print (bst_manipulate(input_value))
```

Run example test case 2:

In [ ]:
```
input_value = [2, 1, 3]
print (bst_manipulate(input_value))
```

4.

Write the function, **delete**$(bst, node)$, which deletes a node, $node$, from a binary search tree, bst.

You are provided with the function, **bst_delete**($delete\_keys$), which takes a list of keys to be deleted as an input, creates a binary search tree (BST), using a provided $BinarySearchTree$ class, and then calls the **delete** function for nodes with keys in $delete\_keys$ list.

Note, this question can be tackled at varying diffuculty. In the easiest case, access to functions **minimum**($node$), **shift_nodes**(bst,old_node,new_node) is allowed. The former function finds the node with the smallest key in the subtree of node, $node$. The latter function, shifts the nodes from the subtree at new_node to the position of old_node.

In the hardest case, one should attempt to implement the whole delete procedure, including the aforementioned functions.

Examples:

In all test examples it is assumed that the tree is first built by inserting nodes with the following keys: `[5, 3, 2, 7, 1, 8, 9, 12]` .

Input: `[5]`
Output: `[7, 3, 2, 1, 8, 9, 12]`

Input: `[7,1,8]`
Output: `[5, 3, 2, 9, 12]`

Constraints:

- Keys are assumed to be unique, integer values.
- $1 \le \text{len}(insert\_keys) \le 100$.
- Each key that is about to be deleted is guaranteed to exist in the tree.

Code:

In [ ]:

```python
from utils.binary_search_tree import BinarySearchTree, Node, insert, preorder, search
from utils.binary_search_tree import minimum, shift_nodes

def delete(bst, node):

    #Write your code here

def bst_delete(delete_keys):

    bst = BinarySearchTree()

    insert_keys = [5, 3, 2, 7, 1, 8, 9, 12]
    node_list = [Node(key) for key in insert_keys]

    for node in node_list:
        insert(bst, node)

    for key in delete_keys:
        node = search(key,bst.root)
        delete(bst, node)

    return preorder(bst.root)
```

Tests:

Run example test case 1:

In [ ]:

```python
input_value = [5]
print (bst_delete(input_value))
```

Run example test case 2:

```
In [ ]:
input_value = [7,1,8]
print (bst_delete(input_value))
```

5.

Below you can find a partial code ($IntervalRedBlackTree$ class) to implement an interval tree by building upon a red-black tree ($RedBlackTree$ class).

An interval is represented by a list of two numbers (e.g. [low,$high$]) indicating the lower and higher end of the interval correspondingly. An interval tree is a binary search tree in which all intervals of all the nodes on the left child subtree of a particular node have their lower value lower than the low value of this node and all intervals of the nodes on the subtree of the right child have their lower value higher than the low value of the interval of this particular node. Ie. it is similar to a binary search tree where keys are represented by lower ends of intervals. However, additional information is stored in the node attributes (see $IntervalNode$ class).

An important operation on interval trees is finding a node which overlaps with an interval of interest. Write the function, **search_interval**($self$,$interval$), which implements the aforementioned functionality for a $IntervalRedBlackTree$ class. Note, your implementation may need a function, **update_max_value**($self$,$node$), which for a particular node and its children updates the attribute, max_val, which stores to the maximum $high$ value of all intervals in the subtree of that particular node.

Note, you are provided with the function, **manipulate_interval_tree**($interval$), which creates an interval tree, calls **update_max_value** function for the root node (runtime $O(n)$, n - number of intervals in a tree) and returns the output of the **search_interval** function.

Examples:

Input: `[11,13]`
Output: `Node(key/low = 13, high=14 color=red, max_val=14), parent=9, left=Nil, right=Nil`

Input: `[-7,-9]`
Output: `Node(Nil, color=black)`

Constraints:

- $1 \leq n \leq 100$.

Code:

```
In [ ]:
from utils.red_black_tree import RedBlackTree, Node, Nil

class IntervalNode(Node):

    def __init__(self, low, high, parent=None, left=None, right=None, color=None,max_

        self.high = high #High end of an interval
        self.max_val = max_val, #The largest high end value of all intervals of the s
        super().__init__(low, parent = parent, left = left, right = right, color = co


    def __repr__(self):
        summary = f"Node(key/low = {self.key}, high={self.high} color={self.color}, m
        if self.parent:
            summary += f", parent={self.parent.key}"
        if self.left:
            summary += f", left={self.left.key}"
        if self.right:
            summary += f", right={self.right.key}"
        return summary

class IntervalRedBlackTree(RedBlackTree):

    def __init__(self):
        super().__init__()

    def update_max_value(self,node):

        #Write your code here

    def search_interval(self,interval):

        #Write your code here

def manipulate_interval_tree(interval):

    rbt = IntervalRedBlackTree()

    interval_list = [[5,6], [3,4], [2,3], [7,8], [1,2], [8,9], [9,10], [13,14]]

    node_list = [IntervalNode(interval[0],interval[1]) for interval in interval_list]

    for node in node_list:
        rbt.insert(node)

    rbt.update_max_value(rbt.root)

    return rbt.search_interval(interval)
```

Tests:

Run example test case 1:

In [ ]:
```
input_value = [11,13]
print (manipulate_interval_tree(input_value))
```

Run example test case 2:

In [ ]:
```
input_value = [-7,-9]
print (manipulate_interval_tree(input_value))
```

6.

A B-tree of minimum degree, t, is a data structure with the property that every node may contain at most $2t-1$ keys. For this question, we will assume that all keys are distinct, that all nodes contain keys but not values, and that all nodes of the B-tree fit into memory (no disk read or writes are required).

Implement a function to insert a given key, key, in a B-tree by completing the **insert_key**(key) function below. For a B-tree storing n keys, your solution should run in $O(t\log_t n)$ time.

Note that the function, **insert_key**, is implemented for a class, $Btree\_new$, which is derived from an original $Btree$ implementation provided with the lectures. Additional code is also provided in function, **btree_manipulate**(key_list), which helps to initialise initialise a B-tree, calls your function to insert a list of keys, visualises it, and then deletes a multiple keys in a sequence from a Btree. A list key_list

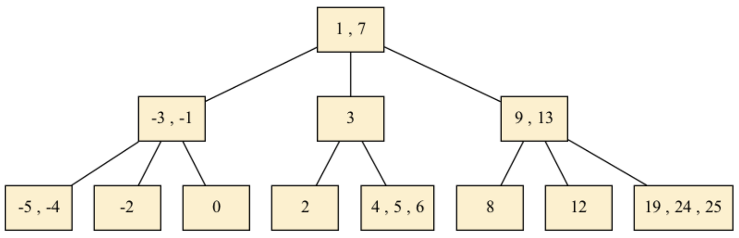
containing two lists of keys to insert and keys to delete is provided as an input to **btree_manipulate**. The minimum and maximum keys left after insertion and deletion steps are performed should be returned by **btree_manipulate**.

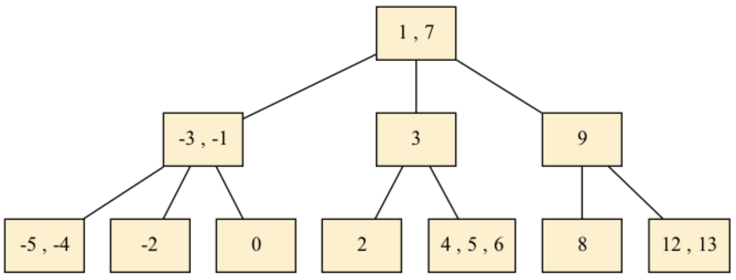
Remark. Insertion operation in a Btree is not examinable, however, it is a very useful exercise to implement this operation as it helps to understand the inner workings of the Btree. If you find that the implementation is taking too long, you can attempt implementing a search function instead.

Examples:

Input: `[[5, 3, 2, 7, 1, 8, 9, 12, 13, 4, 0, 6, -1, 19, 24, 25, -2, -3, -4, -5], [25,24,19]]`
Output: `Minimum key: -5, maximum key: 24`

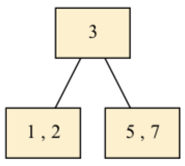
Explanation:

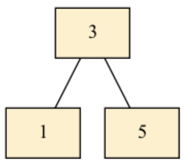
Tree before deletion: 

Tree after deletion: 

Input: `[[5, 3, 2, 7, 1],[2,7]]`
Output: `Minimum key: 1, maximum key: 5`

Explanation:

Tree before deletion: 

Tree after deletion: 

Constraints:

- $1 \leq n \leq 100$.
- Each key that is about to be deleted is guaranteed to exist in the B-tree.

Code:

In [ ]:

```
from utils.btree import Btree, Node
from pathlib import Path
from matplotlib import pyplot as plt
import cv2

class Btree_new(Btree):
    def __init__(self, t):
        super().__init__(t,root=Node(is_leaf=True),verbose=False)

    def insert_key(self, key):
        # Write your code here

def btree_manipulate(key_list):

    insert_keys, keys_to_delete = key_list[0], key_list[1]

    btree = Btree_new(t=2)

    for key in insert_keys:
        btree.insert_key(key)

    btree.viz_btree(dest_path=Path("btree.png"), refresh=True)

    print('Before deletion:')
    plt.imshow(cv2.imread("btree.png")[:,:,::-1])
    plt.show()

    print("Keys to be deleted:")
    print(keys_to_delete)

    for key in keys_to_delete:
        btree.delete(btree.root, key)

    btree.viz_btree(dest_path=Path("btree-after-deletions.png"), refresh=True)

    print('After deletion:')
    plt.imshow(cv2.imread("btree-after-deletions.png")[:,:,::-1])
    plt.show()

    return "Minimum key: "+str(btree.minimum(btree.root))+', maximum key: '+str(btree
```

Tests:

Run example test case 1:

In [ ]:

```
input_value = [[5, 3, 2, 7, 1, 8, 9, 12, 13, 4, 0, 6, -1, 19, 24, 25, -2, -3, -4, -5]
print (btree_manipulate(input_value))
```

Run example test case 2:

In [ ]:

```
input_value = [[5, 3, 2, 7, 1],[2,7]]
print (btree_manipulate(input_value))
```