Andrew Taylor

Atayl136

Homework 9: Searching Ordered Data and Search Trees

**For questions 1 – 2, compare the efficiency of using sequential search on an ordered table of size n and an unordered table of the same size for the key *target*:**

*I read this question tersely. It asks about tables and sequential search, trees are not mentioned, so I answered the questions thinking about arrays.*

1. **a) If no record with the key *target* is present.**

**b) If one record with the key *target* is present and only one is sought.**

Comparing ***sequential*** search's efficiency on a table of size n when it is ordered and unordered for the target key:

a) When the key target is not present in the table:

Ordered table: Worst-case O(n) as you need to search all elements.

Unordered table: Worst-case O(n) as you need to search all elements.

b) When the key target is present once and only one is sought:

Ordered table: Worst-case O(n) as you need to search all elements, if it is last.

Unordered table: Worst-case O(n) as you need to search all elements, if it is last.

Ordered table: Average-case O(n/2) since, on average, you will search through half of the elements before finding the target.

Unordered table: Average-case O(n/2) since, on average, you will search through half of the elements before finding the target

1. **a) If more than one record with the key *target* is present and it is desired to find only the first one.**

**b) If more than one record with the key *target* is present and it is desired to find them all.**

a) When multiple records with the key target are present, and only the first one is desired:

Ordered table: Best-case O(1) if the first target is at the beginning; otherwise, average-case O(n/2) for finding the first one.

Unordered table: Average-case O(n/2) since, on average, you will search through half of the elements before finding the first target.

b) When multiple records with the key target are present, and all of them are desired:

Ordered table: Worst-case O(n) if all keys are the target; otherwise, linear time complexity based on the number of target keys. Also, all target keys are encountered in a group due to the ordering. The search can be terminated early if it encounters a greater key.

Unordered table: Worst-case O(n) if all keys are the target; otherwise, linear time complexity based on the number of target keys.

*I am confused by the question. Strictly speaking, a sequential search of a table will generally continue to the end regardless of ordering. Perhaps you meant to ask about an ordered tree?*

1. **Write a method delete(key1, key2) to delete all records with keys between key1 and key2 (inclusive) from a binary search tree whose nodes look like this:**

|  |  |  |
| --- | --- | --- |
| Left | **key*i*** | **right** |

class BSTNode:

def \_\_init\_\_(self, key, left=None, right=None):

self.key = key

self.left = left

self.right = right

# finds smallest key in the tree

def findMinValue(node):

current = node

while current.left is not None:

current = current.left

return current.key

# deletes the keys, inclusive, and rejoins the tree recursively

def deleteKeysInRange(node=root, key1, key2):

if node is None:

return None

node.left = deleteKeysInRange(node.left, key1, key2)

node.right = deleteKeysInRange(node.right, key1, key2)

if node.key >= key1 and node.key <= key2:

# seals the tree if the endpoints are leaves

if node.left is None:

return node.right

elif node.right is None:

return node.left

else:

# this sets every qualifying node’s key to the lowest value in the tree

# eliminating the reference to the node so it is garbage collected

# we know the minimum value stays in the tree so we can use it

node.key = findMinValue(node.right)

# it attempts it repeatedly on the right subtree to take out all keys before key2

node.right = deleteKeysInRange(node.right, node.key, node.key)

# the below logic seals the gap by finding the edges, continuing the recursion

# these nodes are visited in the recursion too so they can be referenced

elif node.key = key1:

leftedge = node.left

node.right = deleteKeysInRange(node.right, node.key, node.key)

elif node.key = key2:

rightedge = node.right

node.right = deleteKeysInRange(node.right, node.key, node.key)

elif rightedge and leftedge:

rightedge.left = leftedge

node.right = deleteKeysInRange(node.right, node.key, node.key)

return

1. **Write a method to delete a record from a B-tree of order n.**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| p0 | **r1** | **p1** | **r2** | **p2** | **r3** | **…….** | **pn-1** | **rn** | **pn** |

# this makes the tree conveniently so every key is represented in the root node. Every node contains its key and the subtrees.

class BTreeNode:

def \_\_init\_\_(self, min\_degree, is\_leaf=True):

self.min\_degree = min\_degree

self.is\_leaf = is\_leaf

# the root/node has an index for every node

self.keys = []

# the two subtrees are indicated as index 0 and index 1, index n, etc…

self.children = []

# this lets you refer to the nodes by index in the parent

def findIndexOfKey(node, key):

index = 0

while index < len(node.keys) and node.keys[index] < key:

index += 1

return index

# just a function to indicate the leaf as a flag

def isLeaf(node):

return node.is\_leaf

# finds and deletes a node preserving rules

def deleteKeyFromBTree(node=root, key):

# base case when the bottom of the tree is reached

if node is None:

return

index = findIndexOfKey(node, key)

# If the key is found in the root or it’s children

if index < len(node.keys) and node.keys[index] == key:

if node.isLeaf:

# Remove the key from a leaf node

node.keys.pop(index)

else:

# Remove the key from the non-leaf node

if len(node.children[index].keys) >= node.min\_degree:

pred = getPredecessor(node, index)

node.keys[index] = pred

deleteKeyFromBTree(node.children[index], pred)

# if an additional child is needed to meet B-Tree

elif len(node.children[index + 1].keys) >= node.min\_degree:

succ = getSuccessor(node, index)

node.keys[index] = succ

deleteKeyFromBTree(node.children[index + 1], succ)

else:

# if more than one child is needed we need to merge nodes

mergeNodes(node, index)

deleteKeyFromBTree(node.children[index], key)

else:

# if the root is a “leaf”

if node.isLeaf:

return "Key not found in the tree."

else:

# Determine if the key is in the last subtree pointed to by the last child

is\_last\_subtree = (index == len(node.keys))

# Ensure the child node has at least the minimum number of keys before recursion

if len(node.children[index].keys) < node.min\_degree:

if is\_last\_subtree and index > 0:

index -= 1

mergeNodes(node, index)

# Recurse on the appropriate child node

if is\_last\_subtree and index < len(node.keys):

# goes right

deleteKeyFromBTree(node.children[index + 1], key)

else:

# goes left

deleteKeyFromBTree(node.children[index], key)

return