Robert Morgan
Dec. 2020, Jan. 2021

Programming Interview Practice

Problems taken from “Programming Interviews Exposed: Secrets to Landing Your Next Job” by John Mongan and Noah Suojanen

Repository of Materials: GitHub Repo

Language of Choice: Python

Chapter 3: Linked Lists

Problem: *Stack Implementation*. Implement a stack using either a linked list or dynamic array (justify your decision). Design the interface to your stack to be complete, consistent, and easy to use

Solution

I choose to implement the stack as a linked list. The disadvantage of a linked list is that you need to iterate through the list to find an element, however, since we are implementing a stack, we only need to access the first element. Dynamic arrays have convenient, index-based element access for interacting with the top of the stack, in practice will be less efficient for a stack. The reasoning here is that arrays store their data in contiguous memory locations, so every time elements are popped or pushed onto the stack, the memory addresses of all elements will need to be updated, making these operations O(n). A linked list will only require the adjustment of a couple pointers, which will be O(1) complexity.

```
class DoublyLinkedListNode:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class Stack:
    def __init__(self):
        self.head = None

    def push(self, data):
        new_node = DoublyLinkedListNode(data)
        if self.head is None:
            self.head = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
```

```
    def pop(self):
        if head is None:
            raise IndexError(“Cannot pop from an empty Stack”)
        data_to_pop = self.head.data
        self.head = self.head.next
        self.head.prev = None
        return data_to_pop

    def peek:
        if head is None:
            raise IndexError(“Cannot peek from an empty Stack”)
        return self.head.data
```

Notes based on book solution:

- A dynamic array doesn’t require additional memory allocation for pointers, so if the elements are all small integers (require much less memory than a pointer) then a dynamic array may use less memory space
- LinkedLists are also much less complicated than dealing with dynamic resizing, so that is another advantage, especially in an interview

Problem: *Maintain Linked List Tail Pointer*. `head` and `tail` are global pointers to the first and last elements, respectively, of a singly linked list of integers. Implement functions to `delete` and `insert_after`. The argument to `delete` is the element to be deleted. The two arguments to `insert_after` give the data for the new element and the element after which the new element is to be inserted. It should be possible to insert at the beginning of the list by calling `insert_after` with `None` as the element argument. These functions should return 1 if successful and 0 if unsuccessful.

Solution

Not really much to talk about, let’s just code.

```
class Element:
    def __init__(self, data):
        self.data = data
        self.next = None

def locate(element):
    current, prev = head, None
    while current.data != element.data:
        if current.next is None:  # Assumes tail.next is None
            raise ValueError(f”{element} not in linked list”)
        prev = current
        current = current.next
    return prev, current


def delete(element):
```

```
    if element is head:
        head = head.next
        return 1

    try:
        current, prev = locate(element)
    except ValueError:
        return 0  # Unsuccessful

    if current is tail:
        tail = prev

    prev.next = current.next
    del current
    return 1

def insert_after(element, data):
    new_element = Element(data)

    if element is None:  # Insert at beginning
        new_element.next = head
        head = new_element
        return 0
    elif element is tail:  # Insert at end
        current = tail
        tail = new_element
    else:  # Lookup element
        try:
            current, _ = locate(element)
        except ValueError:
            return 0  # Unsuccessful

    new_element.next = current.next
    current.next = new_element
    return 1
```

Notes based on book solution:

- Make sure to update head and tail global pointers if they are affected by the insertion or deletion
- Account for corner cases of the list being empty or exceeding some maximum length that will overflow memory

Problem: *Bugs in `remove_head`.* Find and fix the bugs in the following function that is supposed to remove the head element from a singly linked list:

```
def remove_head(head):
    del head
    head = head.next
```

Robert Morgan
Dec. 2020, Jan. 2021

Solution

The bug in this code is that `head` is deleted prior to the pointer to the next node being utilized to set the new head of the linked list. Therefore, the interpreter will raise a `NameError` on the variable "`head`" when the second line of the function is executed. To fix the bug, we need to remove the `del head` line and leave the removal of the old head from memory up to garbage collection, or utilize a temporary variable to point to head such that we can remove it from memory ourselves. If the individual elements in this list take up large chunks of memory, the second approach is preferred such that we can ensure the freed-up memory is available immediately.

```
def remove_head(head):
    head = head.next

# OR to ensure memory is immediately free

def remove_head(head):
    temp = head.next
    del head
    head = temp
```

Notes based on book solution:

- Probably good practice to check that the object passed to the function has the next attribute and really is a pointer to the head of a linked list

Problem: *Mth-to-Last Element of a Linked List*. Given a singly linked list, devise a time- and space-efficient algorithm to find the mth-to-last element of the list. Implement your algorithm, taking care to handle relevant error conditions. Define mth-to-last such that when m=0, the last element in the list is returned.

Solution

The brute force approach is to iterate through the list twice, the first time counting the elements and the second time accessing the mth-to-last element. This is space-efficient but not quite time-efficient (we can probably get creative and do better than O(2n)).

A slightly better solution would be to iterate through the list fully once, and load each element into a hashtable with the index as the key. This solution would only require one iteration through the list and one constant time lookup using the hashtable, making it O(n+1), but now we run into the problem of being space-inefficient since the hashtable will require just as much memory as the original list. Plus inserting each element into the hashtable is n O(1) operations, so this is basically O(2n+1) even with the hashtable.

My proposal is to use two pointers simultaneously. Start the first pointer at head. If the end of the list is reached before we have gotten to the mth element, handle the error case that the list

doesn't have an mth-to-last element. If the list has enough elements, start the second pointer at head once the first pointer reaches the mth element. Then step through the list incrementing both pointers at the same time. Once the first pointer reaches the tail, return the element where the second pointer is pointing.

```
def get_mth_to_last(head, m):
    if m < 0:
        raise ValueError("Value of m must be nonnegative")

    lead_pointer = head
    if lead_pointer is None:
        raise ValueError("Linked List cannot be empty")

    counter = -1
    rear_pointer = None
    while lead_pointer.next != None:  # Indicating we're at the end
        if rear_pointer is None:
            counter += 1
            if counter == m:
                rear_pointer = head
        lead_pointer = lead_pointer.next
        rear_pointer = rear_pointer.next

    if rear_pointer is None:  # Indicating list was shorter than m
        raise IndexError("linked list must contain at least m elements")

    return rear_pointer.data
```

Let's do some basic tests. If the list has a length of 10 and we use m=0, then neither value error is raised, nor is the index error raised. We begin the while loop. rear_pointer is None, so counter is incremented. Counter is equal to m, so rear_pointer is set to head (which is the same as lead_pointer). Thus when we get to the end of the list, we will return the last element. We should probably raise a value error if m is not an integer at the top so the m<0 line doesn't throw an error. We should probably also check that the node passed through head has the necessary attributes of a singly linked list node. If m = 2, rear_pointer gets set to head when lead_pointer is head.next.next, so we will return the second-to-last element when lead_pointer reaches the tail.

Notes based on book solution:

- None

Problem: *List Flattening*. Start with a standard doubly linked list. Now imagine that in addition to next and previous pointers, each element has a child pointer, which may or may not point to a separate doubly linked list. These child lists may have a child of their own, which may have its own child, and so on. Flatten the list so that all the nodes appear in a single-level, doubly linked list. You are given the head and tail of the first level of the list. Each node is an instance of the following class:

Robert Morgan
Dec. 2020, Jan. 2021

```
class Node:
    def __init__(self, data):
        self.next = None
        self.prev = None
        self.child = None
        self.data = data
```

Solution

I would first want to know whether the final order mattered, since that would affect the ordering of the output. Let's assume it doesn't and then adapt the algorithm if necessary.

The multi-level data structure described in the problem is also referred to as an n-ary tree. Particularly since each node only has one possible child, this data structure can be thought of as a binary tree.

```
class Node:
    def __init__(self, data):
        self.next = None   # same as "right" in a BT node
        self.prev = None   # same as "parent" in a BT node
        self.child = None   # same as "left" in a BT node
        self.data = data
```

Therefore, one way we can flatten the data structure is traversing the data structure using either a breadth-first or depth-first search (depending on how the desired ordering of the output) and appending each element to a new array each time. However, this approach is spatially inefficient since copying the elements into a new array requires double the space.

To flatten the list without requiring additional space, we'll need to adjust the pointers instead of copying the elements. We can do this flattening slightly more efficiently by using a breadth-first search (ie. traverse the list with the next pointer, and every time a node has a child, load the child into a queue, then once you reach the tail pointer, set the next element to be the first element dequeued). This method only requires a lot of memory if the number of children in a particular level is huge, otherwise the space needed is much less than the full array.

The most efficient solution would not need a temporary data structure to store nodes in progress. Perhaps iterate through the list using the next pointer, and then whenever a node has a child, set the next pointer of tail to point to the child (and prev of child to point to tail), and then update the tail pointer to … Hmm. Updating the tail pointer kind of sucks because you have to iterate through all the next attributes of the child. Perhaps a depth first search is best, and each time you encounter a child set it to be the next attribute. The issue here is losing the connection to the next attribute while processing the information from the child. I think the intermediate data structure might be the best we can do.

```
class Queue:
    def __init__(self):
```

```
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        try:
            return self.queue.pop(0)
        except IndexError:
            raise IndexError("Queue is empty, nothing to dequeue")

    def is_empty(self):
        return len(self.queue) == 0

def flatten_list(head, tail):
    if head is None or tail is None:
        raise ValueError("Cannot flatten empty list")

    queue = Queue()
    queue.enqueue(head)
    while not queue.is_empty():
        current = queue.dequeue()
        if current.child is not None:
            queue.enqueue(current.child)
        if current.next is not None:
            queue.enqueue(current.next)

        tail.next = current
        current.prev = tail
        tail = current
        del current.child
        del current.next
```

Notes based on book solution:

- Instead of loading into a queue, just stick the child at the end, and you'll eventually get to it to handle its children once your iteration reaches that point. This is O(n), simpler, and requires no additional space for an intermediate data structure

```
def flatten_list(head, tail):
    if head is None or tail is None:
        raise ValueError("Cannot flatten empty list")

    current = head
    end_node = tail
    while current is not end_node:
        if current.child is not None:
            tail.next = current.child
            current.child.prev = tail
            current.child.next = None
            tail = current.child
```

```
        current = current.next
```

Problem: *List Unflattening*. After having flattened the list above, restore the original multi-level data structure.

Solution

We have the advantage that our approach never changed the child pointers of the nodes, so that is what we will have to use to reconstruct the multi-level list. Because we appended to the end, each time a node in the new list has a child, that indicates a new level. All we need to do is adjust the next / prev pointers and we should be good to go.

This will involve some recursion to explore the possible children of the child.

```
def unflatten_list(head, tail):
    if head is None or tail is None:
        raise ValueError("Cannot flatten empty list")

    def explore_and_separate(start):
        if start is None:
            return

        if start.child is None:
            return

        else:
            start.child.prev = None  # Use None pointer to indicate new level
            start.child.prev.next = None
            explore_and_separate(start.child)

    current = head
    tail_updated = True # update tail after first child found, then use to terminate
    while current is not tail:
        if current.child is not None:
            if not tail_updated:  # Update the tail when the first child is found
                tail = current.child.prev
                tail_updated = True

            explore_and_separate(current.child)

        current = current.next
```

Notes based on book solution:

- None

Problem: *Null or Cycle*. You are given a linked list that is either null-terminated (acyclic) or ends in a cycle (cyclic). Write a function that takes a pointer to the head of the list and determines if the list is cyclic or acyclic. You may not modify the list in any way. Note that the data contained

by a node might be identical to the data contained in another node, but that the two nodes are independent.

Solution

Since we cannot modify the list, we are not able to add a boolean `has_visited` flag to each element as we traverse the list. This makes me think we should copy the elements to a secondary data structure. Let's think about a hashset (something with constant time lookup). We cannot use the data as the keys, since two nodes can have the same data. We could use a counter to index the nodes, but we risk updating the counter incorrectly when a cycle does occur. Hashset might be out.

Obviously our solution should have the property that if it encounters a pointer to `None`, it stops and returns "acyclic" as the answer. The brute force approach would copy every node as it is encountered and add a boolean flag, however this is spatially inefficient.

A fun python-specific approach is to store the memory address of each node in a constant-time lookup data structure, then ask if the next node in question's memory address is in the array. Since we'd only be storing memory addresses, the overhead would be significantly lighter than copying the whole list.

```
def is_cycle(head):
    if head is None:
        raise ValueError("Null pointer passed to is_cycle")

    memory_addresses = set()
    current = head
    while current is not None:
        if id(current) in memory_addresses:
            return True
        else:
            memory_addresses.add(id(current))
        current = current.next
    return False
```

Notes based on book solution:

- You could compare the node to all its predecessors by starting at the head and iterating until you reach the current node. If any nodes are equal then you've found a cycle. The trouble here is that this approach is O(n^2).
- A clever solution is to use two pointers, and to increment them at different speeds. Therefore, if a cycle is present, the fast pointer will eventually equal the slow pointer. This approach is spatially efficient and O(n).

```
def is_cycle(head):
    if head is None:
        raise ValueError("Null pointer passed to is_cycle")
```

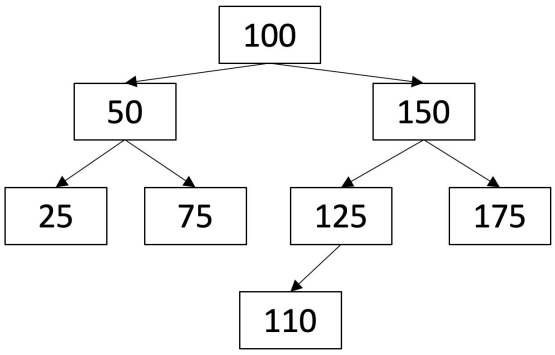
Robert Morgan
Dec. 2020, Jan. 2021

```
fast_pointer = head
slow_pointer = head
try:
    while fast_pointer.next is not None:
        if fast_pointer is slow_pointer:
            return True
        fast_pointer = fast_pointer.next.next
        slow_pointer = slow_pointer.next

# Attribute error caused by end of list not having a next attribute, so acyclic
except AttributeError:
    return False

return False
```

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 4: Trees and Graphs

Problem: *Preorder Traversal*. Informally, a preorder traversal involves walking around the tree in a counter-clockwise manner starting at the root, sticking close to the edges, and printing out the nodes as you encounter them. For the tree to the right, the result is 100, 50, 25, 75, 150, 125, 110, 175. Perform a preorder traversal of a binary search tree, printing the value of each node.



Solution

In a preorder traversal, we visit the root, then visit the left subtree, then visit the right subtree. We can straightforwardly implement this traversal with a recursive function.

```
class BSTNode:
    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data

    def __str__(self):
        return str(self.data)

def preorder_traversal(root):
    if root is None:
        return

    print(root)
    preorder_traversal(root.left)
    preorder_traversal(root.right)
```

Testing this solution on the example tree yields the output 100, 50, 25, 75, 150, 125, 110, 175 as intended.

Notes based on book solution:

- None

Problem: *Preorder Traversal, No Recursion*. Perform a preorder traversal of a binary search tree, printing the value of each node. This time, you may not use recursion.

Solution

Robert Morgan
Dec. 2020, Jan. 2021

Sure. Why not. A preorder traversal of a binary tree is the same as a depth-first search. Thus, we can load the root onto a stack, and while the stack is not empty pop and print, then add the children to the stack. The only trick here is to match the preorder behavior, we have to load the right child onto the stack first, such that the left child is always last-in-first-out.

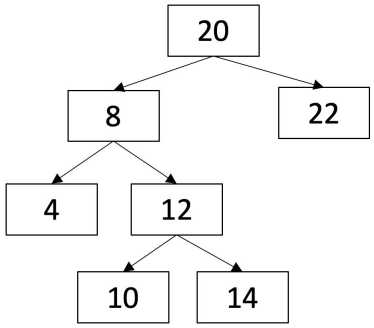
```
class Stack:
    # Stack.peek() is not needed for this example
    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        return self.stack.pop()

    def is_empty(self):
        return len(self.stack) == 0

def preorder_traversal(root):
    stack = Stack()
    stack.push(root)
    while not stack.is_empty():
        current = stack.pop()
        print(current)
        if current.right is not None:
            stack.push(current.right)
        if current.left is not None:
            stack.push(current.left)
```

Testing on the example tree yields 100, 50, 25, 75, 150, 125, 110, 175 as intended.

Notes based on book solution:

- None

Problem: *Lowest Common Ancestor*. Given the value of two nodes in a binary search tree, find the lowest common ancestor. You may assume that both values already exist in the tree. For example, assume 4 and 14 are given as the two values for the tree to the right. The lowest common ancestor would be 8 because it's an ancestor to both 4 and 14 and there is no node lower on the tree that is an ancestor to both 4 and 14.



Solution

Unfortunately we cannot treat the tree as an array because it is not complete. This has the feel of a recursive problem where for a given subtree you ask "are both 4 and 14 in my left (or right)

subtree?" and if the answer is yes you ask the same question of your left (or right) child. Once you get to a point where the answer is no, meaning one of 4 and 14 is in the left child and one is in the right child, you have found the lowest common ancestor. This approach is somewhat inefficient because it essentially does log(n) depth first searches to verify that 4 and 14 are in the children, so maybe we can do better than reassessing the same nodes several times.

One property we did not use is the "search" property of the binary search tree (i.e. left child < root < right child). This may save us some time. Let's start at 20. Since both 4 and 14 are less than 20, we know they both must be in the left subtree, so we go to 8. Since 4 is less than 8 and 14 is greater than 8, we know 4 must be in the left tree and 14 must be in the right tree. Thus, we've found our common ancestor by only visiting two nodes. This approach is O(log(n)). Let's implement it.

```
def find_lowest_common_ancestor(root, value1, value2):
    assert value1 < value2
    if value1 < root.data and value2 < root.data:
        find_lowest_common_ancestor(root.left, value1, value2)
    elif value1 > root.data and value2 > root.data:
        find_lowest_common_ancestor(root.right, value1, value2)
    elif value1 < root.data and root.data < value2:
        return root.data
    else:
        raise RuntimeError("Invalid binary search tree passed as root")
```

Testing the example, we start at 20 and we execute the if clause. We then move to 8, and this time we execute the second elif clause, returning the correct answer of 8.

Notes based on book solution:

- Since we're not analyzing several branches and just going down the tree once, an iterative approach may be simpler that a recursive approach
- An iterative approach will also have fewer function calls, so in practice it may be faster than the recursive solution even though both are O(log(n))

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 5: Arrays and Strings

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 6: Recursion

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 7: Other Programming Topics

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 8: Counting, Measuring, and Ordering Puzzles

Robert Morgan
Dec. 2020, Jan. 2021

Chapter 9: Graphical and Spatial Puzzles