IN2010 2022: Obligatorily assignment 1

Can Hicabi Tartanoglu

20th September 2022

Abstract

I use the Python dataclass to define my components; however, I would prefer the Pydantic base model (de-facto standard for dataclasses). Pydantic is not in the stdlib, but Guido does respect it; I respect the IN2010 rules in my answers and don't use it.

I tested my code on Python 3.10! This might be necessary for you also because of the type hints and walrus operator on 4a!

Question 2

I will use a modified version of linked lists to implement Teque. I have a node that has a previous and next attribute in addition to its value. In the Teque class, I store the farthest back (teque.back), furthest front (teque.front), and the middle node (teque.middle). I could just store one of these nodes, but this implementation speeds up the push operations in the requirement, $\mathcal{O}(1)$.

2d

In big O notation n could be interpreted as $\lim_{n\to\infty}$; however, when n has an upper boundary it replaces infinity in the limit. We can replace n with the upper boundary; therewidth, the only difference in my case is the get function, $\mathcal{O}(10^6)$, which could be $\mathcal{O}(10^6/2)$ if I wasn't lazy.

2a and c

I noted $\mathcal{O}$ in each sub-sub-section tittle. % is used as the remainder division operator in the pseudocode. I found some bugs in earlier version of my Python code during debugging, and might've failed to update my pseudocode properly!

push_back: $\mathcal{O}(1)$

```
   Input: N where N is an integer, 1 ≤ N ≤ 10^6
 1 node ← new Node;
 2 node.value ← N;
 3 if teque.back = null then
 4 |   teque.back ← node;
 5 |   teque.middle ← node;
 6 end
 7 else if teque.front = null then
 8 |   node.previous ← teque.back;
 9 |   teque.back.next ← node;
10 |   teque.front ← node;
11 end
12 else
13 |   teque.front.next ← node;
14 |   node.previous ← teque.front;
15 |   teque.front ← node;
16 |   if teque.size % 3 = 0 then
17 |   |   teque.middle ← teque.middle.next;
18 |   end
19 end
20 teque.size ← teque.size + 1;
```

push_front: $\mathcal{O}(1)$

```
   Input: N where N is an integer, 1 ≤ N ≤ 10^6
 1 if teque.back = null then
 2 |   return teque.push_back(N);
 3 end
 4 node ← new Node;
 5 node.value ← N;
 6 if teque.front = null then
 7 |   teque.back.previous ← node;
 8 |   node.next ← teque.back;
 9 |   teque.front ← teque.back;
10 |   teque.back ← node;
11 |   teque.middle ← node;
12 end
13 else
14 |   teque.back.previous ← node;
15 |   node.next ← teque.back;
16 |   teque.back ← node;
17 |   if teque.size % 3 = 0 then
18 |   |   teque.middle ← teque.middle.next;
19 |   end
20 end
21 teque.size ← teque.size + 1;
```

push_middle: $\mathcal{O}(1)$

```
   Input: N where N is an integer, 1 ≤ N ≤ 10^6
 1 if teque.back = null OR teque.front = null then
 2 |   return teque.push_back(N);
 3 end
 4 node ← new Node;
 5 node.value ← N;
 6 if teque.middle.previous = null then
 7 |   node.previous ← teque.back;
 8 end
 9 else
10 |   node.previous ← teque.middle.previous;
11 end
12 node.next ← teque.middle;
13 teque.middle.next ← node;
14 teque.middle = node;
15 teque.size ← teque.size + 1;
```

get: $\mathcal{O}(n)$

I could make this $\mathcal{O}(n/2)$, but there is no limit to the time complexity on this question.

```
   Input: i where i is an integer, 1 ≤ i ≤ teque.size
 1 if teque.back = null OR teque.size = 0 then
 2 |   return null;
 3 end
 4 if teque.front = null OR i = 0 then
 5 |   return teque.back.value;
 6 end
 7 if i = teque.size - 1 then
 8 |   return teque.front.value;
 9 end
10 current_node ← teque.back;
11 for c ← 0 To i do
12 |   current_node ← current_node.next;
13 end
14 return current_node.value;
```

3a

```
   Input: Kitten node value (current_value); Map that has parent node
          as keys and set of children as values (p2c)
   Output: The node sequence describing the path from the kitten to the
           root
 1 path ← list;
 2 add current_value to the list path;
 3 no_parent ← true;
 4 while true do
 5 |   forall (node_value, children) ∈ p2c do
 6 |   |   if current_value ∈ children then
 7 |   |   |   add node_value to the list path;
 8 |   |   |   current_value ← node_value;
 9 |   |   |   no_parent ← false;
10 |   |   |   break;
11 |   |   end
12 |   end
13 |   if no_parent then
14 |   |   return path;
15 |   end
16 |   no_parent ← true;
17 end
```

4

4a

I use the same Node class from 1a to store progress.

Helper function: bst_sequence

```
Input: Node of BST (start with root node)
Output: Data sequence of BST left2right, top2bottom—starting from
        root node
1 if node = null then
2 |   return null
3 end
4 stdout ← node.value;
5 bst_sequence(node.left);
6 bst_sequence(node.right);
```

sorted_array_to_bst

```
Input: Sorted Array (array) with length N; start index; end index
Output: Root node of BST
1 if start > end then
2 |   return null;
3 end
4 mid ← ⌊(start + end)/2⌋;
5 node ← new Node;
6 node.left ← sorted_array_to_bst(array, start, mid);
7 node.right ← sorted_array_to_bst(array, mid + 1, end);
8 return node;
```

main

```
Input: Sorted Array (array) with size N
Output: Sorted elements for a balanced BST left2right, top2bottom
1 root ← sorted_array_to_bst(array, 0, N);
2 bst_sequence(root);
```

4b

heap_to_bst

```
   Input: Binary heap (heap)
   Output: The current node value
 1 if heap is empty then
 2 |   return null
 3 end
 4 left ← BinaryHeap;
 5 right ← BinaryHeap;
 6 distance_to_node_value ← ⌊lengthofheap/2⌋
 7 for i ← 0 to distance_to_node_value do
 8 |   value ← get and remove first value from heap;
 9 |   add value to left;
10 end
11 value ← get and remove first value from heap;
12 stdout ← value;
13 while heap is not empty do
14 |   value ← get and remove first value from heap;
15 |   add value to right;
16 end
17 heap_to_bst(right);
18 heap_to_bst(left);
```