Programming and Data Structures in Python

Sample Questions, I Semester, 2021–2022

1. (a) Construct a max-heap from the list [16,19,3,19,22,12,22] by inserting each value into the heap, from left to right. Draw the heap after each insertion. You don't have to explain each step or draw sub-steps, just draw the heap as it appears after each insertion is completed.

   (b) Redraw the heap after **two** *deletemax* operations.

2. Recall the different ways we can recursively traverse a binary tree to enumerate its elements:

   - *Inorder*: Print the left subtree using inorder traversal, then the root, then the right subtree using inorder traversal.
   - *Preorder*: Print the root, then the left subtree using preorder traversal, then the right subtree using preorder traversal.
   - *Postorder*: Print the left subtree using postorder traversal, then the right subtree using postorder traversal, then the root.

   Reconstruct the binary tree corresponding to the following information:

   - *Inorder traversal:* `d a j h c f m b k i`
   - *Preorder traversal:* `c a d h j b f m i k`

3. Assume we work with linked lists in Python built up from the basic class `Node` whose structure is indicated by the definition to the right.

   Write a Python function to reverse the list pointed to by a `Node`.

```
class Node:
  def __init__(self,initval=None):
    self.value = initval
    self.next = None
    return

  def reverse(self):
    # To be written by you
```

4. Let `Tree` be a class that implements binary trees. For an object `t` of type `Tree`, the attributes `t.value`, `t.left` and `t.right`, and the functions `t.isempty()` and `t.isleaf()` have the usual interpretation. Suppose we add this function `foo()` to the class. Given an object `mytree` of type `Tree`, what would `mytree.foo()` compute? Explain your answer.

```
def foo(self):
  if self.isempty():
    return(None)
  elif self.isleaf():
    return(self.value)
  else:
    return(max(self.value, max(self.left.foo(), self.right.foo())))
```

The following instructions apply to the next two questions.

- *Assume that for a list `l`, you can add a value at either end (i.e., `l.append(x)`, `l.insert(0,x)` in Python), and access an arbitrary element (read or update `l[j]` for any valid `j`) in constant time.*
- *Your algorithms should be described in Python-like pseudo-code. You should be as precise as possible. You will not be penalized for minor syntax errors.*

---

5. Let `l1` and `l2` be two lists of integers, each sorted in ascending order with no duplicate elements. We can think of `l1` and `l2` as denoting sets of integers S_1 and S_2, respectively.

   Describe efficient algorithms to compute `union(l1,l2)` and `intersect(l1,l2)` that take `l1` and `l2` as inputs and return lists without duplicates sorted in ascending order that correspond to $S_1 \cup S_2$ and $S_1 \cap S_2$, respectively. Assuming that both `l1` and `l2` are of size n, your functions should work in time $O(n)$.

6. Assume that we have a list `a` of size n with elements `a[0]` to `a[n-1]` arranged in ascending order. We can search for a value x in array `a` in time $O(\log n)$ using binary search. If x appears in `a`, binary search will report some index `i` such that `a[i] == x`.

   If the elements in `a` are not distinct, a value x that appears in `a` will be present in a contiguous range of positions $i_\ell, i_\ell{+}1, \ldots, i_r$. Our aim is to adapt binary search to a function `leftpos()` such that `leftpos(x,a)` returns the *leftmost* position of `x` in `a`, if `x` is present in `a` and returns `-1` if `x` is not present in `a`.

   Describe an algorithm for `leftpos()`. Analyze the complexity of your algorithm.

7. Your final exams are over and you are catching up on sports on TV. You have a schedule of interesting matches from all over the world during the next week. You hate to start or stop watching a match midway, so your aim is to watch as many complete matches as possible during the week.

   Suppose there are n such matches $\{M_1, M_2, \ldots, M_n\}$ available during the coming week. The matches are ordered by starting time, so for each $i \in \{1, 2, \ldots, n{-}1\}$, M_i starts before M_{i+1}. However, match M_i may not end before M_{i+1} starts, so for each $i \in \{1, 2, \ldots, n{-}1\}$, $Next[i]$ is the smallest $j > i$ such that M_j starts after M_i finishes.

   Given the sequence $\{M_1, M_2, \ldots, M_n\}$ and the values $Next[i]$ for each $i \in \{1, 2, \ldots, n{-}1\}$, your aim is to compute the maximum number of complete matches that can be watched.

   (a) Let $Watch[i]$ denote the maximum number of complete matches that can be watched among $\{M_i, M_{i+1}, \ldots, M_n\}$. Write a recursive formula for $Watch[i]$ in terms of $Watch[j]$, $j > i$.

   (b) Describe the structure of the table you would need to compute $Watch[1]$ using dynamic programming, and the sequence in which you would fill the table.