Spring 2022

[EC2202] Data Structures

Final Exam: 1 pm, Tuesday, Jun 14

INSTRUCTIONS

- **Do not turn your exam sheet** until directed to do so, otherwise you would be considered cheating.
- You have **1 hour and 50 minutes** to complete the exam (7 problems; maximum of 100 points).
- The exam is closed book, closed notes, closed computer, and closed calculator.
- Mark your answers on the exam sheet itself and be sure to **write your answers in the space (boxes) provided**. We will not consider the answers given outside the boxes (use other space for brainstorming).

POLICIES & CLARIFICATIONS

- If you need to use the **restroom**, bring your exam sheet to the back of the room. Only one person is allowed at a time.
- You may use built-in Python functions that do not require import, such as min, max, pow, len, abs, and ord.
- For **all problems**, assume that you are implementing your program in the Colab environment.
- For **coding problems**, we will ignore minor grammar mistakes for evaluation (focus on the logic and flow). Moreover, your code must be indented correctly.
- Use **English** for your answers and name.

Student ID Number	Name

Q1. (10 points) Dictionaries

Imagine you have ***a special keyboard with all keys in a single row***. The layout of characters on a keyboard is denoted by a string `s1` of length 26. `s1` is indexed from 0 to 25. Initially, your finger is at index 0. To type a character, you have to move your finger to the index of the desired character. ***The time taken to move your finger from index i to index j is |j-i|***, where || denotes absolute value. Find the time taken to type the string `s2` with the given keyboard layout.

```
def find_time(s1, s2):
  """
  >>> find_time("abcdefghijklmnopqrstuvwxyz", "cba")
  4
  >>> find_time("zyxwvutsrqponmlkjihgfedcba", "a")
  25
  >>> find_time("wsjvfpcxkanqgeitdyuhobzmlr", "wrwr")
  75
  >>> find_time("cntzhyejxuodgpfsrbliqmwkav", "cccc")
  0
  >>> find_time("zkqsneutodrvyfbxpgchmiljaw", "ziayqbw")
  69
  """
```

```
  mp = [0] * 26
  for i in range(26):
    mp[ord(s1[i])-97] = i

  ans = 0
  pos = 0
  for i in range(len(s2)):
    ans += abs(mp[ord(s2[i])-97] - pos)
    pos = mp[ord(s2[i])-97]

  return ans
```

Q2. (15 points) Tree Has a Path

(a) (**5 points**) Implement the following function that prints the labels of a given tree with depth-based indentation. Before implementing the function, refer to the code snippet of the Tree class.

```
class Tree:
  def __init__(self, label, branches=[]):
    self.label = label
    for branch in branches:
      assert isinstance(branch, Tree)
    self.branches = list(branches)

  def is_leaf(self):
    return not self.branches

def print_tree(t, indent=0):
  """Prints the labels of T with depth-based indent.
  >>> t = Tree(3, [Tree(1), Tree(2, [Tree(1), Tree(1)])])
  >>> print_tree(t)
  3
    1
    2
      1
      1
  """
```

```
  print(indent * " " + str(t.label))
  for b in t.branches:
    print_tree(b, indent + 2)
```

(b) (**10 points**) Implement the following function that checks whether there is a path in a Tree where the entries along the path spell out the given term.

```
def has_path(t, term):
  """Return whether there is a path in a Tree where the entries along the path
  spell out a particular term.
  >>> greetings = Tree('h', [Tree('i'),
  ...                        Tree('e', [Tree('l', [Tree('l', [Tree('o')])]),
  ...                                   Tree('y')])])
  >>> print(greetings)
  h
    i
    e
      l
        l
          o
      y
  >>> has_path(greetings, 'h')
  True
  >>> has_path(greetings, 'i')
  False
  >>> has_path(greetings, 'hi')
  True
  >>> has_path(greetings, 'hello')
  True
  >>> has_path(greetings, 'hey')
  True
  >>> has_path(greetings, 'bye')
  False
  >>> has_path(greetings, 'hint')
  False
  """
```

```
  assert len(term) > 0, 'no path for empty term.'
  if t.label != term[0]:
    return False
  elif len(term) == 1:
    return True
  for b in t.branches:
    if has_path(b, term[1:]):
      return True
  return False
```

Q3. (10 points) Objects Gone Wrong

Consider the following implementation of the Point class to represent a 2D point in the Cartesian space. For the following implementation, (a) (**3 points**) describe *two problems* of the given implementation, (b) (**5 points**) fix the problems by implementing additional methods, and (c) (**2 points**) describe how users might abuse your implementation in (b) and how to block the abuse.

```
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __repr__(self):
    return "Point(%s, %s)" % (self.x, self.y)
```

(a) The naive implementation does not support
  (1) the comparison of two points (checking equality) and
  (2) the hash functionality (checking the inclusion of an object in a set)

(b)

```
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __repr__(self):
    return "Point(%s, %s)" % (self.x, self.y)

  def __eq__(self, rhs):
    return self.x == rhs.x and self.y == rhs.y

  def __hash__(self):
    return hash((self.x, self.y))
```

(c) Users of (b) could *mutate the objects inside a set* (or a dictionary as key). Then, this incurs hashcode changes; objects located in a wrong slot afterwards. To block this type of abuse, we can make the object *immutable*.

Q4. (25 points) Priority Queue

Complete the implementation of the priority queue below. The below priority queue additionally supports the following functionalities: (1) the internal data array grows when the queue becomes full, and (2) the queue in the initialization stage accepts a list of elements from which the queue builds a binary heap. Each method is worth ***5 points***.

```
class PriorityQueue:
  DEFAULT_CAPACITY = 100

  def __init__(self, els=None):
    if els is None:
      self._data = [ None ] * PriorityQueue.DEFAULT_CAPACITY
      self._size = 0
    else:
      self._data = els
      self._size = len(els) - 1  # because slot 0 is not used
      self.build_heap()

  def __len__(self):
    return self._size

  def findmin(self):
    if self._size == 0:
      raise ValueError("empty priority queue")
    return self._data[1]

  def build_heap(self):
    for i in range(self._size // 2, 0, -1):
      self._bubble_down(i)

  def insert(self, x):
    if self._size + 1 == len(self._data):
      # double size of the array storing the data
      self._data.extend( [ None ] * len(self._data) )
    self._size += 1
    hole = self._size
    # bubble up
    while x < self._data[hole // 2]:
      self._data[hole] = self._data[hole // 2]
      hole //= 2
    self._data[hole] = x

  def deletemin(self):
    min_item = self.findmin()         # raises error if empty
    self._data[1] = self._data[self._size]
    self._size -= 1
    self._bubble_down(1)
    return min_item
```

```
  def _bubble_down(self, i):
    value = self._data[i]
    hole = i
    child = self._smaller_child(hole, value)
    while child != 0:
      self._data[hole] = self._data[child]
      hole = child
      child = self._smaller_child(hole, value)
    self._data[hole] = value

  # Returns index of the child, or 0 if no child is smaller
  def _smaller_child(self, index, value):
    if 2 * index <= self._size:
      child = 2 * index
      if child != self._size and self._data[child + 1] < self._data[child]:
        child += 1
      if self._data[child] < value:
        return child
    return 0
```

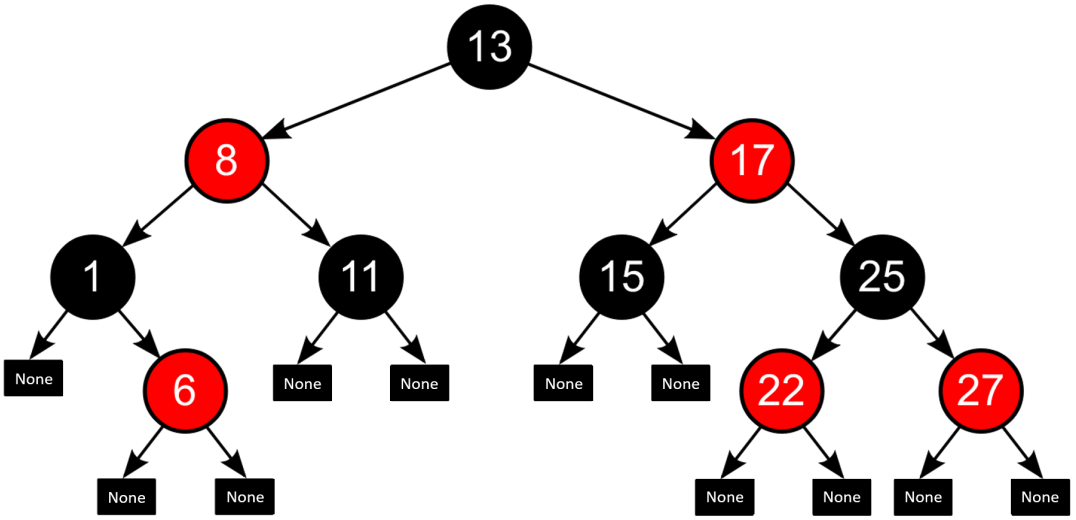
Q5 (15 points) Various Trees

(a) (**10 points**) State the *smallest* number of nodes N(h) that an AVL tree with height h can have in a recurrence form, justify your statement, and derive the upper bound of h in respect of n (the number of nodes in the tree).

N(h) = N(h - 1) + N(h - 2) + 1

- One of the subtrees (say the left subtree) must be an AVL tree of **height h - 1**
- Thus, it has at least **N(h - 1) nodes**
- The other subtree has height **either h - 1 or h - 2**
- Therefore, it has at least **N(h - 2) nodes**
- In conclusion, **N(h) = N(h - 1) + N(h - 2) + 1**
- Since clearly N(h - 1) ≥ N(h - 2), **N(h) ≥ 2N(h - 2)**
- Therefore, if the number of nodes in the tree is n: **$n \geq N(h) \geq 2^{h//2}$**
- which implies **(log n) ≥ h // 2**
- Thus, **h ≤ 2 log n**

(b) (**5 points**) Convert the following red-black tree into the corresponding 234 tree.



Q6. (15 points) Shortest Paths

Given an n x n binary matrix grid, return the length of the shortest clear path in the matrix. If there is no clear path, return -1. A clear path in a binary matrix is a path from the top-left cell, i.e., (0, 0) to the bottom-right cell, i.e., (n - 1, n - 1) such that: (1) ***all the visited cells of the path are 0*** and (2) all the adjacent cells of the path are ***8-directionally connected*** (i.e., they are different and they share an edge or a corner). The length of a clear path is the number of visited cells of this path. Implementations that use additional memory space less than $O(n^2)$ will receive ***15 points***; otherwise ***10 points***.

```
def shortest_path(grid):
  '''
  >>> grid = [[0, 1], [0, 1]]
  >>> shortest_path(grid)
  2
  >>> grid = [[0, 0, 0], [1, 1, 0], [1, 1, 0]]
  >>> shortest_path(grid)
  4
  >>> grid = [[1, 0, 0], [1, 1, 0], [1, 1, 0]]
  >>> shortest_path(grid)
  -1
  '''
```

#1 (15 points) Solution that does not require additional memory space (destructive)

```
  max_x, max_y = len(grid) - 1, len(grid[0]) - 1

  # check if the first and last cells are open.
  if grid[0][0] != 0 or grid[max_x][max_y] != 0:
    return -1

  # set up the BFS.
  queue = deque()
  queue.append((0, 0))
  grid[0][0] = 1
  dx, dy = [1, -1, 0, 0, 1, 1, -1, -1], [0, 0, 1, -1, -1, 1, -1, 1]

  while queue:  # carry out the BFS.
    x, y = queue.popleft()
    dist = grid[x][y]
    if x == max_x and y == max_y:
      return dist
    for cx, cy in zip(dx, dy):
      nx = x + cx
      ny = y + cy
      if 0 <= nx <= max_x and 0 <= ny <= max_y:
        if grid[nx][ny] == 0:
          grid[nx][ny] = dist + 1
          queue.append((nx, ny))

  # There was no path.
  return -1
```

#2 (10 points) Solution that requires additional memory space (non-destructive)

```
  max_x, max_y = len(grid) - 1, len(grid[0]) - 1

  # check if the first and last cells are open.
  if grid[0][0] != 0 or grid[max_x][max_y] != 0:
    return -1

  # set up BFS
  queue = dequeu()
  queue.append((0, 0, 1))
  visited = [[False for i in range(max_y))] for j in range(max_x)]
  dx, dy = [1, -1, 0, 0, 1, 1, -1, -1], [0, 0, 1, -1, -1, 1, -1, 1]

  while len(queue) != 0:  # carry out BFS
    x, y, d = queue.popleft()
    if x == max_x and y == max_y:
      return d
    visited[x][y] = True
    for cx, cy in zip(dx, dy):
      nx = x + cx
      ny = y + cy
      if 0 <= nx <= max_x and 0 <= ny <= max_y:
        if grid[nx][ny] == 0 and not visited[nx][ny]:
          queue.append((nx, ny, d + 1))

  # there was no path.
  return -1
```

Q7. (10 points) Dijkstra Algorithm

Given a weighted, undirected and connected graph, ***find the shortest distance of all the vertices' from the source vertex***. All vertices are represented by integers. A graph with n vertices has vertices from 0 to n-1. The graph is given in the form of a list of edges; (i, j, d) means that vertex i and vertex j are connected with distance d. The first argument given as an input means the number of vertices and the second argument is the source vertex, and the last argument is the list of edges. Assume that input graphs ***do not contain any negative weight cycles***.

```
def dijkstra(n, source, edges):
  '''
  >>> dijkstra(2, 0, [(0, 1, 9)])
  [0, 9]
  >>> dijkstra(3, 2, [(0, 1, 1), (1, 2, 3), (2, 0, 6)])
  [4, 3, 0]
  >>> dijkstra(4, 2, [(0, 1, 4), (0, 2, 1), (0, 3, 5), (1, 2, 2), (3, 2, 8)])
  [1, 2, 0, 6]
  >>> dijkstra(5, 4, [(0, 1, 1), (0, 4, 5), (1, 4, 8), (1, 2, 10), (2, 4, 2), (2, 3,
5), (3, 4, 6)])
  [5, 6, 2, 6, 0]
  >>> dijkstra(5, 4, [(0, 1, 1), (0, 4, 5), (1, 4, 8), (1, 2, 10), (2, 4, 2), (2, 3,
3), (3, 4, 6)])
  [5, 6, 2, 5, 0]
  >>> dijkstra(6, 0, [(0, 3, 1), (0, 1, 2), (1, 3, 2), (1, 2, 3), (0, 2, 5), (2, 3, 3),
(3, 4, 1), (2, 4, 1), (2, 5, 5), (4, 5, 2)])
  [0, 2, 3, 1, 2, 4]
  '''
```

```
  graph, INF = [[] for i in range(n)], int(1e9)
  for edge in edges:
    graph[edge[0]].append((edge[1], edge[2]))
    graph[edge[1]].append((edge[0], edge[2]))

  visited, distance = [False] * n, [INF] * n
  visited[source], distance[source] = True, 0
  for j in graph[source]:
    distance[j[0]] = j[1]

  for i in range(n - 1):  # iterate for n-1 nodes (except the starting node)
    # select the node with the minimum distance and process it
    now, min_value = 0, INF
    for k in range(n):
      if distance[k] < min_value and not visited[k]:
        min_value, now = distance[k], k
    visited[now] = True
    for j in graph[now]:  # check the neighbor nodes of the current node
      cost = distance[now] + j[1]
      if cost < distance[j[0]]:
        distance[j[0]] = cost
  return distance
```