A zero mark will be given
if you copy someone else's
work or you let someone
copy your work.

| COMP122/22-T01 Data Structures and Algorithms | |
|---|---|
| Test 1: Linear Structures | 2022-02-23<br>*Due Date* — **In Class** |
| *Class Code* | |
| *Student No.* | DO NOT WRITE YOUR NAME |

**This is a CLOSED BOOK test, 80 minutes, 100 full marks.**

## I Iterators and Generators

1. Suppose $s$ is an iterable. Write a generator function *take_alt*($s$) to yield the alternate elements of $s$, starting from the second element. For examples, the alternate elements of 'A','B','C','D','E','F' are 'B','D','F', and the alternate elements of 2,3,5,7,11,13,17 are 3,7,13.

```
def take_alt(s):
    c = False              1 setup flag
    for x in s:            2 loop
        if c:              1 check flag
            yield x        2 yield
        c = not c          2 toggle flag
```
(1). 8

2. Suppose $s$ and $t$ are two iterables. Write a generator function *chain*($s$, $t$) to yield all the elements of $s$, followed by all the elements of $t$. For examples, the *chain* of 1,2,3,4 and 10,20,30 are 1,2,3,4,10,20,30.

```
def chain(s, t):
    yield from s      2
    yield from t      2
```
(2). 4

3. Write an expression to produce a list of the alternate elements of iterable $s$ starting from the first element, by using the *take_alt* and *chain* above. For example, if $s$ is 1,2,3,4,5, the list should be [1,3,5].

```
list(take_alt(chain([0], s)))
1    1        1      1    1
```
(3). 5

4. Generator function $g$ is defined below.

```
def g():
    for x in range(100):
        if x % 3 == 0 and x % 5 != 0:
            yield x*x
```

Write a generator expression equivalent to g().

```
(x*x for x in range(100) if x%3 == 0 and x%5 != 0)
  2   2                  2
```
(4). 6

## II Singly Linked Lists

The *Node* class of a singly linked list is defined below.

```
class Node:
    def __init__(self, elm, nxt):
        self.elm, self.nxt = elm, nxt
```

Each node has two attributes, where *elm* stores the element and *nxt* points to the next node. A linked list is terminated with None. Suppose $h$ points to the head node of such a linked list.

5. Write a function *max_node*($h$) to return the node containing the maximum element in linked list $h$. If $h$ is empty, the function should return None.

```
def max_node(h):
```

```
    if h is None:
        return None          2 check for empty list and return
    m = h                    1 assume first node
    p = h.nxt                1 start from second node
    while p is not None:     2 while-loop and condition
        if p.elm > m.elm:    1 check for greater
            m = p            1 change to greater
        p = p.nxt            2 move to the next node
    return m                 1 return
```

(5). 11

6. Suppose linked list $h$ has $n$ nodes, what is the time complexity of function *max_node*($h$)?

$\mathcal{O}(n)$ (6). 3

7. Let $n$ be an integer. Write a function *cons*($n$) to construct a linked list consisting of

$$1, 2, \ldots, n-1, n, n-1, \ldots, 2, 1.$$

For example, *cons*(5) constructs a linked list $1 \to 2 \to 3 \to 4 \to 5 \to 4 \to 3 \to 2 \to 1$. The function returns the head node of the constructed linked list. If $n \leqslant 0$, the function should return None.

```
def cons(n):
```

```
    h = None                 1 init
    for i in range(n-1):     1 second half loop range
        h = Node(i+1, h)     3 push
    for i in range(n):       1 first half loop range
        h = Node(n-i, h)     3 push
    return h                 1 return
```

(7). 10

8. What is the time complexity of function *cons*($n$), in terms of $n$?

$\mathcal{O}(n)$ (8). 3

9. Suppose linked list $h$ has $n$ nodes. Write a function $ins(h,\ i,\ x)$ to insert a new node with element $x$ at index $i$, and return the head node of the updated linked list. Assume $0 \leqslant i \leqslant n$. For example, if $h$ is $3 \to 4 \to 5$, $ins(h,\ 2,\ 100)$ updates the linked list to $3 \to 4 \to 100 \to 5$.

```
def ins(h, i, x):

    p = h                      (1) init current pointer
    q = None                   (1) init previous pointer
    for j in range(i):         (1) for loop
        q = p                  (1) advance previous pointer
        p = p.nxt              (1) advance current pointer
    if q is None:              (1) check head node case
        return Node(x, p)      (2) push to head and return
    else:
        q.nxt = Node(x, p)     (1) push to the middle
        return h               (1) return old head
```

(9) . 10

---

10. For a linked list $h$, write a function $del\_eq(h,\ x)$ to delete the first node with element equal to $x$ from $h$, and return the head node of the updated linked list. If $x$ is not in $h$, the function changes nothing. For example, if $h$ is `'Ada'` $\to$ `'Bob'` $\to$ `'Tom'` $\to$ `'Joe'` $\to$ `'Tom'`, $del\_eq(h,$ `'Tom'`$)$ updates the linked list to `'Ada'` $\to$ `'Bob'` $\to$ `'Joe'` $\to$ `'Tom'`. You have three cases to handle, (1) $x$ is in the head node, (2) $x$ is in a middle node, and (3) $x$ is not found.

```
def del_eq(h, x):

    p = h                                    (1) init current pointer
    q = None                                 (1) init previous pointer
    while p is not None and p.elm != x:      (2) while loop, 2 conditions
        q = p                                (1) advance previous pointer
        p = p.nxt                            (1) advance current pointer
    if p is None:                            (1) check element not found
        return h                             (1) return old head
    elif q is None:                          (1) check head node case
        return p.nxt                         (1) return second node
    else:
        q.nxt = p.nxt                        (1) skip deleted node
        return h                             (1) return old head
```

(10) . 12

---

## III Stacks and Queues

A LIFO stack is defined by the *Stack* class including the methods:

$$push(self, x),\ pop(self),\ top(self) \text{ and } \_\_bool\_\_(self).$$

A FIFO queue is defined by the *Queue* class including the methods:

$$push\_back(self, x),\ pop(self),\ top(self) \text{ and } \_\_bool\_\_(self).$$

11. Write a function *reverse*($s$, $q$) to reverse the elements in *Stack* $s$ with the help of an initially empty *Queue* $q$. You must *not* create any other structures.

```
def reverse(s, q):

    while s:
        q.push_back(s.pop())
    while q:
        s.push(q.pop())
```

(1) while-loop and condition
(2) pop and push_back
(1) while-loop and condition
(2) pop and push

(11). (6)

12. For a *non-empty Stack* $s$, write a function *pop_min*($s$, $t$) to return and remove the minimum element from $s$, with the help of another *Stack* $t$. The remaining elements must be still stored in $s$ *without* the need to keep the original order. Since $t$ is possibly not empty initially, you need to count the number of elements transferred from $s$ to $t$, to move them back. You must *not* create any other structures.

```
def pop_min(s, t):

    m = s.pop()
    n = 0
    while s:
        x = s.pop()
        if x < m:
            t.push(m)
            m = x
        else:
            t.push(x)
        n += 1
    for i in range(n):
        s.push(t.pop())
    return m
```

(1) assume top min
(1) reset counter
(1) while-loop
(1) pop for comparison
(1) new min found
(1) push old min
(1) change to new min
(1) push not-a-min
(1) increase counter
(1) loop to move elements back
(1) move
(1) return

(12). (12)

13. With the *pop_min* function we can sort the elements in a *Stack* $s$, by repeatedly popping minimum elements from $s$. Write a function *sort*($s$, $t$) to arrange the elements in $s$ such that they are in increasing order from top to bottom, with the help of another initially empty *Stack* $t$. When you push all the min elements to $t$, you have a stack of decreasingly arranged elements, so that you can transfer them back to $s$ to accomplish the sorting. You must *not* create any other structures.

```
def sort(s, t):

    while s:
        t.push(pop_min(s, t))
    while t:
        s.push(t.pop())
```

(1) while-loop to select min
(4) push min to t, others back to s
(1) while-loop to move and reverse elements back
(1) move

(13). (7)

14. Suppose $s$ has $n$ elements. What is the time complexity of function *sort*($s$, $t$)?

$\mathcal{O}(n^2)$ (14).

(3)