Algorithms and Data Structures

Mock Exam

July 12, 2022

Last Name: ____

First Name: ____

Signature: ____

1. (a) Order the following functions by their asymptotic growth rate:

$$n^2 + n^4, n^2 \log n, n^2, (\log n)^2, n^3, (\log n)^3$$

6 4 3 1 5 2

(b) An algorithm executes an $O(\log n)$-time computation for each entry of an n-element sequence. Give a Big-Oh upper bound and a Big-Omega lower bound on its running time.

(c) Give the best possible Big-Oh upper bound for the running time of the following Python function which takes as input two Python lists `A` and `B` with respective lengths n and m.

```python
def mystery_function (A,B):
    i = 0
    j = 0
    while i<len(A) and j<len(B):
        if A[i]==B[j]:
            return True
        elif A[i]<B[j]:
            i += 1
        else:
            j += 1
    return False
```

(d) What is `mystery_function` good for? State, if necessary, conditions on the input arrays `A` and `B` that make `mystery_function` perform useful work.

(5+5+5+5)

(b) Upper bound per execution is $O(\log n)$, it's performed n times, so the overall upper bound is $O(n \log n)$.

The upper bound $O(\log n)$ permits that the actual compute time is asymptotically smaller than $\log n$. The smallest possible lower bound per execution is $\Omega(1)$. As the algorithm is executed n times, the overall lower bound is $\Omega(n)$.

(c) $O(n) + O(m)$

(d) Assuming that A and B are sorted (smallest first), mystery_function returns True if and only if A and B have at least one element in common.

2. Is each of the following statements true or false? Explain your answer in 1–2 sentences.

   (a) One can implement a *stack* based on a *linked list* such that each push or pop operation completes in $O(1)$-time.

   (b) One can implement a *stack* based on a *dynamic array* such that each push or pop operation completes in $O(1)$-time.

   (c) It is possible to append a *linked list* to another in $O(1)$-time.

   (d) *Heap-sort* is always faster then *insertion-sort*.

   (e) *Heap-sort* is faster than *insertion-sort* when the input is a list containing n copies of the same number.

(2+2+2+2+2)

(a) True, even with a singly-linked list if we push/pop at the start of the list.

(b) False. Adding or removing an element to a dynamic array can trigger a resize operation which is $O(n)$ individually and only $O(1)$ amortized.

(c) True, if using doubly-linked lists where access to the end of a list is $O(1)$.

(d) False, insertion-sort is $O(n)$ if the list is already sorted. In this situation, heap-sort need not "bubble", but has more comparisons, so it also runs $O(n)$ but with a larger constant.

(e) False, see explanation for (d).

3. (a) The nodes of a complete binary tree have keys that represent their position in a breadth-first traversal of the tree. Argue that this tree is a heap.
   (b) Give a pseudo-code (or Python) representation of the breadth-first traversal of a tree with an auxiliary queue.
   (c) Argue that the run-time of this algorithm is $O(n)$, where n is the number of nodes in the tree.
   (d) Alternatively, you can process the nodes of this tree breadth-first by repeatedly calling the `remove_min` method of the heap. Does this algorithm also run in $O(n)$ time? Explain!

(5+5+5+5)

(a) A heap is a complete binary tree with the heap-order property which says that a parent is always $\leq$ than its children.

In breadth-first, children are visited only after all parents at the same level of the tree have been visited, which gurantees that heap-order is satisfied.

(b) Input: a tree T

```
Q = Queue()
Q.enqueue(T.root())
while Q.not_empty():
    p = Q.dequeue()
    # process p
    for c in T.children(p):
        Q.enqueue(c)
```

(c) Every node is enqueued exactly once, and dequeued exactly once, giving a running time of $O(n)$.

(d) No, the remove_min operation will move the rightmost leaf at the bottom level to the root, from which it'll bubble down in $O(\log n)$ time.

Thus, total running time will be $O(n \log n)$, so this is not a good use case for a heap.

4. Attached is an excerpt of a code listing for a buggy implementation of a priority queue with a binary heap.

   (a) Draw an example of a heap with exactly 5 nodes so that a call to `remove_min` produces an invalid heap.

   (b) Identify the part of the code that is buggy. Explain!

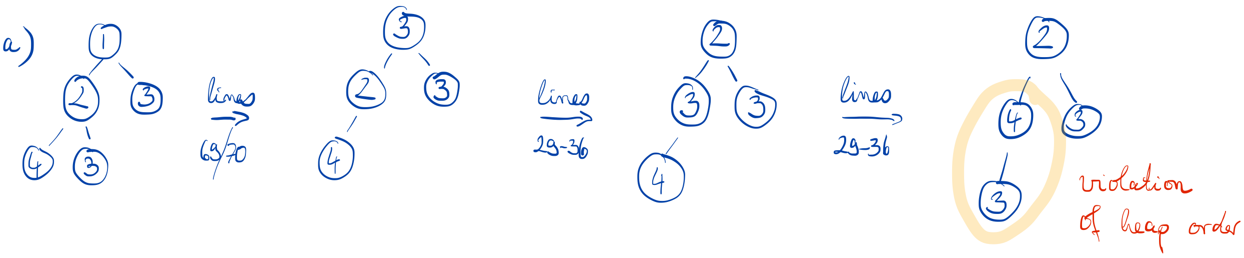
   Hint: The functions `_parent` to `_has_right` which implement the index arithmetic are correct, you do not need to look there.

   (c) Fix the bug.

   (d) Rewrite the function `_upheap` to use a loop instead of recursion.

(5+5+5+5)

(a)



(b) Lines 36 and 37 which swap parent and smallest child must only be executed when the parent is larger than the child. This check is missing.

(c) See listing.

(d) See listing.

```python
class HeapPriorityQueue(PriorityQueueBase):

  def _parent(self, j):
    return (j-1) // 2

  def _left(self, j):
    return 2*j + 1

  def _right(self, j):
    return 2*j + 2

  def _has_left(self, j):
    return self._left(j) < len(self._data)

  def _has_right(self, j):
    return self._right(j) < len(self._data)

  def _swap(self, i, j):
    """Swap the elements at indices i and j of array."""
    self._data[i], self._data[j] = self._data[j], self._data[i]

  def _upheap(self, j):
    parent = self._parent(j)
    if j > 0 and self._data[j] < self._data[parent]:
      self._swap(j, parent)
      self._upheap(parent)
```

NON-RECURSIVE VERSION:

```
while j>0:
    p = self._parent[j]
    if self._data[p] < self._data[j]
        return
    self._swap(j, p)
    j = p
```

```python
  def _downheap(self, j):
    if self._has_left(j):
      left = self._left(j)
      small_child = left
      if self._has_right(j):
        right = self._right(j)
        if self._data[right] < self._data[left]:
          small_child = right
      self._swap(j, small_child)
      self._downheap(small_child)
```

```
if self._data[j] > self._data[small_child]:
    self._swap(j, small_child)
    self._downheap(small_child)
```

```python
  def __init__(self):
    """Create a new empty Priority Queue."""
    self._data = []

  def __len__(self):
    """Return the number of items in the priority queue."""
    return len(self._data)

  def add(self, key, value):
    """Add a key-value pair to the priority queue."""
    self._data.append(self._Item(key, value))
    self._upheap(len(self._data) - 1)

  def min(self):
    """Return but do not remove (k,v) tuple with minimum key.

    Raise Empty exception if empty.
```

```python
    """
    if self.is_empty():
      raise Empty('Priority queue is empty.')
    item = self._data[0]
    return (item._key, item._value)

  def remove_min(self):
    """Remove and return (k,v) tuple with minimum key.

    Raise Empty exception if empty.
    """
    if self.is_empty():
      raise Empty('Priority queue is empty.')
    self._swap(0, len(self._data) - 1)
    item = self._data.pop()
    self._downheap(0)
    return (item._key, item._value)
```

5. Write an algorithm `min_list`, in pseudo-code or in Python, that returns a list with the values of all nodes of a heap whose key is identical to the minimal key. (10)

```
def min_list(T):

    _min_list(T, 0)        # 0 is the index of the root node


def _min_list(T, j):

    if T[j]._key > T[0]._key           # current key at j is not minimal

        return []                      # empty list, nothing to return

    if T[j]._has_right():              # must also have left child as T is complete

        return T[j]._value + _min_list(T[j]._left) + _min_list(T[j]._right)

    if T[j]._has_left():

        return T[j]._value + _min_list(T[j]._left)

    return T[j]._value
```