MONASH University

Office Use Only

Semester Two 2014
Examination Period

Faculty of Information Technology

EXAM CODES: **FIT1008**
TITLE OF PAPER: **COMPUTER SCIENCE**
EXAM DURATION: 3 hours writing time
READING TIME: 10 minutes

THIS PAPER IS FOR STUDENTS STUDYING AT:(tick where applicable)

☐ Berwick ✓ Clayton ☐ Malaysia ☐ Off Campus Learning ☐ Open Learning
☐ Caulfield ☐ Gippsland ☐ Peninsula ☐ Enhancement Studies ☐ Sth Africa
☐ Parkville ☐ Other (specify)

During an exam, you must not have in your possession, a book, notes, paper, electronic device/s, calculator, pencil case, mobile phone or other material/item which has not been authorised for the exam or specifically permitted as noted below. Any material or item on your desk, chair or person will be deemed to be in your possession. You are reminded that possession of unauthorised materials, or attempting to cheat or cheating in an exam is a discipline offence under Part 7 of the Monash University (Council) Regulations.

No exam paper or other exam materials are to be removed from the room.

AUTHORISED MATERIALS

OPEN BOOK	☐ **YES**	✓ **NO**
CALCULATORS	☐ **YES**	✓ **NO**
SPECIFICALLY PERMITTED ITEMS	☐ **YES**	✓ **NO**

if yes, items permitted are:

Candidates must complete this section if required to write answers within this paper

STUDENT ID: __ __ __ __ __ __ __ __ __ DESK NUMBER: __ __ __ __ __

Page		Mark
3		5
5		10
7		6
9		4
11		6
13		4
15		4

Page		Mark
17		10
19		10
21		7
23		6
25		8
Total:		**80**

Blank Page

Question 1 (4 + 1 = 5 marks)

(a) Write a function, **`insertion_sort`**, in Python which takes as input a list, **`the_list,`** of numbers and sorts this list into **increasing** order using insertion sort.

```
def insertion_sort(a_list):
    for mark in range(1,len(a_list)):
        item = a_list[mark]
        i=mark-1
        while i>=0 and a_list[i]>item:
            a_list[i+1] = a_list[i]
            i-=1
        a_list[i+1]=item
```

4 marks in total:

1 mark for having a 'mark' that moves forward through the list (effectively splitting between the sorted and unsorted sections

1 mark for moving the item to within the sorted sub list

1 mark for shuffling appropriately

1 mark for sorting into ascending order

This means it is possible to get 1 mark for providing any sorting algorithm and 2 if it involves shuffling

(b) Explain why the following algorithm does not implement a stable sorting method.

```
def unstable_sort(the_list):
    n = len(the_list)
    for _ in range(n-1):
        for j in range(n-1):
            if the_list[j] >= the_list[j+1]:
                the_list[j], the_list[j+1] = the_list[j+1], the_list[j]
```

The use of a >= sign means that identical items can be moved past one another, where a > sign would not.

1 mark for an explanation that indicates they understand stability in this context.

5

Question 2 (6 + 2 + 2 = 10 marks)

(a) This question is intended to test your skills at programming with *queues* and *iterators.*

Consider a **`Queue`** class which implements a queue using some data structure (you do not need to know which one) and defines the following methods:

```
__init__()
append(item)
serve()
is_empty()
__iter__()
```

Define a new method (outside this class, so you have no idea how the queue is implemented and, therefore, can ONLY use the above methods):

`pushBack(queue)`

which takes as input a queue and returns a new queue where every negative integer has been pushed to the back of the queue preserving the same order.

For example, if the input queue holds the values **5, 8, -9, 10, -3, -7, 4,** where **5** is at the front of the queue, then the method should return a new queue with the characters in the following order:

5, 8, 10, 4, -9, -3, -7

Your method is *NOT* allowed to modify the input queue. Of course, if the input queue is empty, the output queue should also be empty.

```
def pushBack(a_queue):
        pos_q = Queue()
        neg_q = Queue()

        for item in a_queue:
            if item<0:
                neg_q.append(item)
            else:
                pos_q.append(item)

        for item in  neg_q:
            pos_q.append(item)

        return pos_q
```

6 marks for this question:

- 1 mark for checking if negative when serving from the original queue
- 1 mark for splitting into a positive and negative queue
- 2 marks for recombining into a single queue
- 1 mark for appropriately looping over the queue (using the appropriate Queue functions)
- 1 mark for not changing the input queue

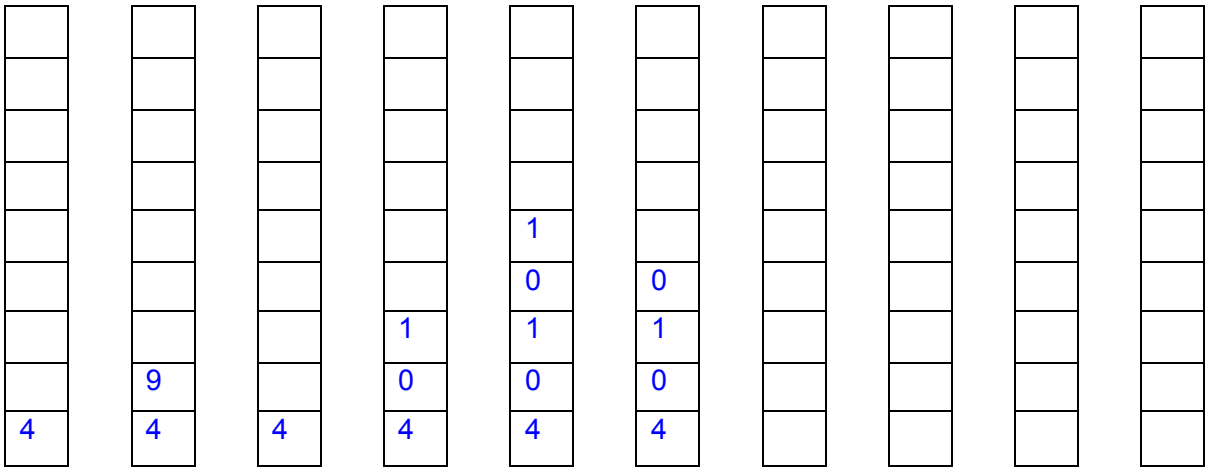
(b) This question is about *understanding code*. Consider a stack data type provided by the **`Stack`** class which is implemented using some data structure (you do not need to know which one) and defines the following methods:

```
__init__()
pop()
push(item)
is_empty()
```

Consider the following function that uses the above methods:

```
def mystery(list):
    the_stack = Stack()
    for item in list:
        if item > 0:
            the_stack.push(item)
        elif item == 0:
            for i in range(2):
                the_stack.push(i)
        elif not the_stack.is_empty():
            the_stack.pop()
        # HERE
```

Use the sequence of stacks below to draw the elements on the stack every time the line **# HERE** is reached during the execution of **`mystery([4, 9, -3, 0, 0, -3])`**.



Question is worth 2 marks, award 0.5 marks for each of the 2nd, 3rd, 4th, and 5th
Full marks if the student grows the stack down rather than up.

(c) This question is about *choosing data structures*. Several data types (such as stacks, queues, and lists) can be implemented using arrays or using linked nodes. Explain an advantage and a disadvantage of the array implementation compared to a linked node implementation.

10

Arrays are more memory efficient when they are close to full while linked nodes are more efficient when sparsely populated. This is as each element in an array takes up less space than each element in a linked list but for an array the memory must be allocated in advance for all items while a linked list can have nodes dynamically created.

Additionally arrays have random access which means you can access any element using their index in constant time where linked nodes would require you to search through the list from start to end to find a given element O(n).

Lastly, when performing an insertion or deletion from an array data structure we have to shuffle forwards or backwards leading to an O(n) operation where a linked structure could do this in O(1) as they merely need to change a few links.

There are 2 marks associated with this question:

1 mark for the explanation of an advantage of arrays

1 mark for the explanation of a disadvantage of arrays do not award marks for advantages or disadvantages without explanation, if the explanation is poorly reasoned but present, award half marks.

Question 3 (6 marks)

This question about *array implementation of lists.* Consider the class **`SortedList`** which implements a **sorted list data type** using an array

```
class SortedList:
      def __init__(self, size):
          assert size > 0, "size should be positive"

          self.the_array = size*[None]
          self.count = 0

      def __len__(self):
          return self.count

      def is_empty(self):
          return len(self) == 0

      def is_full(self):
          return len(self) >= len(the_array)
```

Define a method for this class **`insert(self, the_item)`** which adds a new item, **`the_item`**, to the list in the correct position. If the list is full the method should return **`False`**, else it should return **`True.`**

```
def insert(self, the_item):
      if self.is_full():
          return False
      else:
          i=0
          while I < self.count and self.the_array[i] > the_item:
                i +=1

          j = self.count
          while j > i:
              self.the_array[j] = self.the_array[j-1]
              j -= 1

          self.the_array[i] = the_item
          self.count += 1
          return True
```

There are 6 marks associated with this question:

2.5 marks for shuffling when the item does not go at the end
2.5 marks for finding correct position to insert the item
0.5 marks for handling a full list and returning False
0.5 for correctly updating count when the item is inserted and returning True

Full marks if the student assumes the list is sorted in descending order rather than ascending order.

6

Blank Page

Question 4 (1 + 1 + 1 + 1 = 4 marks)

This question is about *time complexity*. For each of the given Python functions, identify the time complexity for both best and worst case by providing an explanation. (*No explanation means no marks.*)

(a)
```
def product_func(n):
    product = 1
    for _ in range(n):
        for num in range(5):
            product *= num
```

(b)
```
def total_func(n):
    total = 0
    for _ in range(n):
        for num in range(n):
            total += num
```

(c)
```
def another_total_func(n):
    total = 0
    for num in range(n):
        total += num
    for num in range(n+1):
        total += num
```

(d)
```
def division_func(n):
    product = 1
    while n > 1:
        product *= 2
        n //= 2
```

4

Blank Page

a) O(n) , as the inner loop is executed a constant number of times, and the outer loop is executed n times.

b) O(n^2), as there are two nested loops. The outer loop is executed n times, and each time the outer loop is executed the inner loop is executed n times.

c) O(n), there are two loops. The first loop is executed n times, and the second loop is executed n+1 times. Therefore the number of steps during execution is linear in terms of n.

d) O(logN). The loop is executed k times, where $n = 2^k$. So, $k = \log_2 n$.

1 mark for each correct answer, must have an explanation.

Question 5 (6 marks)

This question is about *linked structures*. Consider the two classes **`Node`** and **`List`** as seen in the lectures, which define a **list data type** implemented using a linked structure:

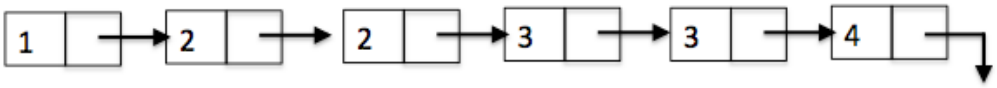
```
class Node:
    def __init__ (self, item = None, link = None):
        self.item = item
        self.next = link

class List:
    def __init__ (self):
        self.head = None
    def is_empty (self):
        return self.head is None
```

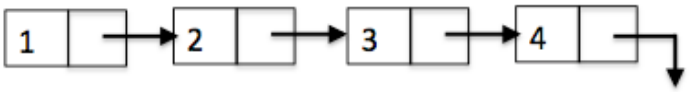
Add the following method to the **`List`** class:

```
def remove_duplicates(self):
```

which removes all the duplicates of each item in a sorted list. For example, consider a **`List`** object **`a_list`** whose **`a_list.head`** points to the first node of the list whose **`item`** is **1**.



After calling **`a_list.remove_duplicates()`**, **`a_list`** should contain only the unique items.




```
def remove_duplicates(self):
if self.head:
    curr = self.head
    while not curr.next is None:
        if curr.item == curr.next.item:
            curr.next = curr.next.next
        else:
            curr = curr.next
```

There are 6 marks in total:

- 2 correctly updating the link to the next node
- 2 marks for looping until either a pointer is None
- 1 mark moving pointer along if not a duplicate
- 0.5 marks for starting at the head and 0.5 for only proceeding if head isn't None

6

Blank Page

Question 6 (4 marks)

This question is about *iterators*. Define a **`PrimeIterator`** iterator class, where given an integer **n**, an instance of this class would produce the first prime starting from **n**, then the second prime, the third prime, and so on indefinitely.
For example:

```
>>> it = PrimeIterator(3)
>>> next(it)
3
>>> next(it)
5
>>> next(it)
7
>>> next(it)
11
```

In other words, define the following three methods for the class **`PrimeIterator`**: **`__init__`**, **`__iter__`** and **`__next__`**.

```
class PrimeIterator:
    def __init__(self, start):
        self.current = start

    def __iter__(self):
        return self

    def __next__(self):
        if self.current < 2:
            val = 2
            self.current = 3
        else:
            while not self.isPrime(self.current):
                if self.current %2 == 0:
                    self.current += 1
                else:
                    self.current += 2

            val = self.current
            self.current += 2

        return val

    def isPrime(self, num):
        k = 2
        while k*k <= num:
            if num % k == 0:
                return False
            k += 1

        return True
```

4

Blank Page

4 marks for this question:

1 mark for correctly defining an init to start at the given value

2.5 marks for checking if a given number is prime by some means. If mostly correct award the 2.5, if somewhat correct award 1.5. If shows some relevant thought 0.5, otherwise 0

0.5 for updating their current number (no requirement to increase by 2 but better)

Question 7 (2+2 = 4 marks)

(a) Consider the following code for the binary search algorithm, where the **`the_array`** is an array and target is the item you are searching for in **`the_array`**. Find the bugs in code and describe how you would fix them.

```
def binary_search(the_array, target):
      low = 0
      high = len(the_array)

      while (low < high):
          mid = (low + high) // 2

          if (the_array[mid] == target):
              return mid
          elif (target < the_array[mid]):
              high = mid
          else:
              low = mid

      return -1
```

- high should start at len(array)-1
- Should be while low<=high
- high should update to mid-1
- low should update to mid+1

0.5 marks for fixing each of these bugs

(b) Explain the best and worst time complexity for the binary search algorithm.

best is O(1) where the item is precisely in the middle of the list O(logN) where not in the list as we keep halving the section of list to look at with each step and there are log_2(N) steps to perform hence the time is O(logN)

1 mark for the best and 1 mark for the worst case. they must include an explanation for these complexities. award only 0.5 marks if the explanation is poor and no marks if there is no explanation.

4

Blank Page

Answer 8(b)

Recursive functions may use more memory as every function call will have it's own stack frame associated with it; these stack frames each have their own copies of local variables, arguments, etc. Given that recursive functions are functions that call themselves as 'iterations' every recursive call is a new stack frame. On the other hand, an iterative function has iterations in the same function to perform the task rather than calling new functions.

As such recursion can use more memory than iterative functions.

2 marks for a complete explanation mentioning stack frames (they don't need to use this term as long as it's clear that's what they're talking about) and why recursion has many while iterative functions would just have the one

Question 8 [8 + 2 = 10 marks]

(a) This question is about *MIPS programming and understanding function calls*. Translate the following Python code faithfully into MIPS assembly language. Make sure you follow the MIPS function calling and memory usage conventions.

Python Code	MIPS Code

`def collatz(n):`	collatz
	addi $sp, $sp, -8
	sw $ra, 4($sp)
	sw $fp, 0($sp)
	addi $fp, $sp, 0

`if n % 2 == 0:`	lw $t0, 8($fp) #take the first argument
	addi $t1, $0, 2
	div $t0, $t1
	mfhi $t2 #t2 now holds n%2
	
	beq $t2,$0,if
	j else

`return n // 2`	if: lw $t0, 8($fp) #take the first argument
	addi $t1, $0, 2
	div $t0, $t1
	mflo $v0 #v0 now holds n//2
	
	lw $fp, 0($sp)
	lw $ra, 4($sp)
	addi $sp, $sp, 8
	j $ra

`return 3*n + 1`	else: addi $t3, $0,3
	lw $t0, 8($fp)
	mult $t0,$t3
	mflo $t2 #this now holds 3n
	addi $t2, $t2,1 #3n+1
	add $v0, $t2, $0 #move into v0
	
	lw $fp, 0($sp)
	lw $ra, 4($sp)
	addi $sp, $sp, 8
	j $ra

mark by proportional correctness

.

(b) Explain why a recursive function may use more memory than an iterative function.

10

See opposite page

Question 9 (6 + 4 = 10 marks)

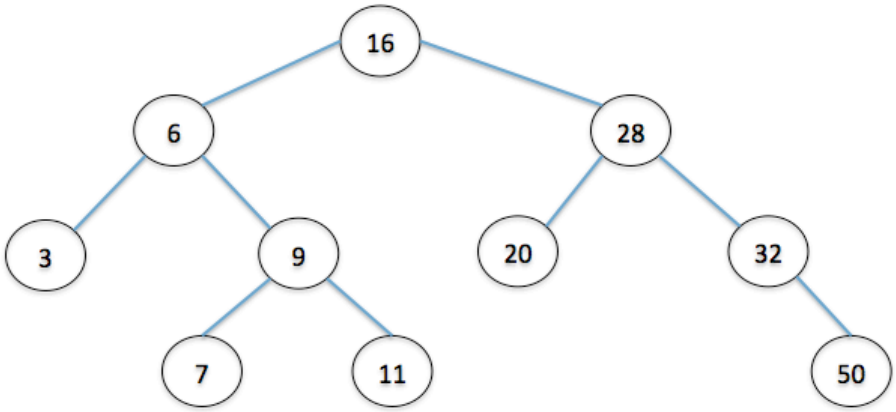
(a) This question is about *recursive programming* and *linked structures.* Consider the two classes **`BinarySearchTreeNode`** and **`BinarySearchTree`** which define a binary search tree data type implemented using linked nodes, and which are defined as follows:

```
class BinarySearchTreeNode:
    def __init__ (self, key, item=None, left=None, right=None):
      self.key = key
      self.item = item
      self.left = left
      self.right = right

class BinarySearchTree:
    def __init__ (self):
      self.root = None
```

Define a recursive method **`collect_leaves(self)`** inside the **`BinarySearchTree`** class that collects the keys of all the leaf nodes (i.e. nodes without children) in the binary search tree and returns them inside a Python list in increasing order.

For example, if the binary search tree is:



the method will return the Python list **`[3,7,11,20,50].`** If the binary search tree is empty the list returned will also be empty.

IMPORTANT: you are defining the method inside the class **`BinarySearchTree`** and therefore you can access anything inside that class. Also, you are using Python lists and therefore you can use any operation that Python makes available on them, such as **`+`**.

Write your answer of this part on the next page

Write your answer for Question 9(a) here

```
def collect_leave(self):
    return self.collect_aux(self.root)

def collect_aux(self, node):
    if node.left is None and node.right is None:
        return [node.item]
    elif node.left is None:
        return self.collectAux(node.right)
    elif node.right is None:
        return self.collectAux(node.left)
    else:
        return self.collectAux(node.left) + self.collectAux(node.right)
```

6 marks:

- 1 mark for giving the correct result for an empty tree
- 2 marks for dealing with leaves (1 mark for condition and 1 for adding key to list)
- 1 mark for dealing with left subtree
- 1 mark for dealing with right subtree
- 1 mark for the combines the result of each subtree.

(b) This question is about *recursive sorts*. Both merge sort and quick sort are divide and conquer algorithms, i.e., they divide the original problem into sub-problems (hopefully of roughly equal size), they solve each sub-problem independently, and then combine the solutions to solve the original problem.

What are the main differences between merge sort and quick sort in terms of the amount of effort put into the dividing and combining operations mentioned above?

In merge sort the work is being done in recombining the sublists and the division into sublists is trivial. Splits precisely in two each time, recombines as if performing insertions into a sorted list.
In quick sort, the work is being done to split the sublists and the recombining is trivial. splits are done about a pivot (which must be chosen well) and the sublists are split between less than and greater than the pivot. In the end there will simply be a list of pivots which are in order.

4 marks to distribute:

- 2 for merge sort:
  - 1 mark for explaining the splitting
  - 1 mark for explaining the recombining
- 2 for quick sort:
  - 1 mark for explaining the splitting
  - 1 mark for explaining the recombining

10

Blank Page

Question 10 (4 + 2 + 1 = 7 marks)

This question is about *Binary Trees.*

(a) Explain how a Binary Search Tree being balanced or unbalanced affects the best and worst time complexity for searching for an item.

With binary trees, we know that items to the left of a node are smaller than it and items to the right are larger than it. A balanced tree will have most paths having the same length (or rather the depth of the tree is consistent). This means that with each new depth you may cut out half of the remaining subtree leading to O(logN) time complexity to find.

With an unbalanced tree this does not hold, at worst, one may end up with a single long chain of nodes with each having a single child. In this case, with each new depth we travel, we can only reduce the number of nodes to look at by one, this is effectively a linked list with time complexity of O(N)

4 marks for this question:

They should make some mention of the depth of the tree and its effects on search
2 marks will be for explaining how balanced trees lead to depth = log(#nodes)
2 marks will be for explaining how unbalanced trees (at worst) have depth=#nodes
Give only 1 mark if the explanation is poor but had some thought involved, otherwise 0

(b) Construct an expression tree for the following arithmetic expression.

4 + 2 * 3 - 5 * 8

2 marks for this question; 1 mark for an expression tree. 1 mark for correctness.

(c) Traverse the Expression Tree you constructed in Part (b) in preorder and write the values of the nodes as you visited them.

1 mark for correctness of traversal, award 0.5 if they instead perform post-order or in-order traversal

7

Blank Page

.

Question 11 (2 + 4 = 6 marks)

This question is about *heaps* and in this question the Heap is a ***max Heap***.

(a) What is the minimum number of elements that must be moved during a "retrieve the maximum element" operation on a heap? Give an example of a heap with 7 elements for which a "retrieve the maximum element" operation will require this minimum number of moves.

If every element is the same then retrieving the max can be performed with a single swap. Normally, if all elements are unique, the number of moves would be equal to the depth +1 where the one is from swapping the root with the last element.

2 marks:

0.5 marks for giving such an example;
1.5 marks for explaining why that is the case;
Award 1 mark if the explanation is solid but incomplete or only 0.5 if the explanation is poor but suggests some thought. If they give a heap with no explanation give 0.5.

(b) Provide an implementation of the method **`delete(self, k)`** that deletes from a **Heap** the element at position ***k***. You can assume that the heap's elements are numbers stored in an array called **`array`** (with the root at position 1), that you have an instance variable **`count`**, and that the **Heap** has the following methods:

```
sink(self, k)
rise(self, k)
swap(self, i, j).
```

```
def delete(self, k):
    self.array[k] = self.array[1] + 1
    rise(k)
    self.swap(1, self.count)
    sink(1)
    self.count -= 1
```

4 marks to distribute:

1 mark for the heap remaining a heap after their function terminates
2.5 marks for appropriate use of sink and or rise operations (or equivalent if they choose to write it from scratch
0.5 marks for decreasing the count.

6

Blank Page

Assuming each location in the array stores [key,value] and initially populated with None.

```
def __setitem__(self, the_key, data):
        if self.count==self.tablesize:
              self.rehash()

        start = self.hash(the_key)%self.tablesize #get the first position to check

       for offset in range(tablesize):
            pos = (start+offset)%self.tablesize

             if self.array[pos] is None:
                   self.array[pos] = (the_key,   data)
                  self.count+=1
                  return
            elif self.array[pos][0] == the_key:
                 self.array[pos] = (the_key,   data)
                 return
```

4 marks to distribute here:

1 mark for updating an item if found
1 mark for creating a new item if not found
0.5 marks for handling a full hashtable
1 mark for updating position as per linear search
0.5 marks for using the hash function appropriate for the start location

Question 12 (2 + 2 + 4 = 8 marks)

This question is about *hash tables*.

(a) Why is Quadratic Probing better than Linear Probing when inserting a new element? Explain in terms of the probe chain (*no explanation means no marks*).

With linear probing clusters end up forming and any items nearby part of a probe chain will get caught in the same probe chain since a collision simply means we check the item underneath.

Quadratic probing on the other hand is less likely to have different keys getting stuck in the same probe chain. This is as a collision means we move 1 down, then 2, then 4, then 8, etc. This means a collision in a nearby position with a different key will almost certainly be in a different chain.

2 marks to distribute:

0.5 for explaining how linear probing handles collisions

0.5 for how quadratic probing handles collisions

1 mark for using this to reason why quadratic probing is often better.

(b) Describe an operation that is more efficient to implement with a Binary Search Tree than a Hash Table?

Hashtables are superior at getting individual values very quickly but keep them in (seemingly) random positions. As such determining the maximum or minimum value within a hash table would require we look at every single position in the table (as you can't know which values have already been entered). With a Binary Search Tree however, you simply head down the tree to the rightmost or leftmost element; an O(logN) operation as opposed to an O(N) operation. Naturally this assumes the Binary Tree is mostly balanced.

2 marks to distribute:

0.5 mark for providing an example where this holds (there may be other operations)

1.5 marks for a correct explanation of why this is so.

1 mark if the explanation is incomplete but on the right track, 0.5 marks if it seems some thought has been involved; otherwise 0

(c) Consider the class **`Hash`** which has the instance variables **`array`**, **`tablesize`**, and **`count`**, and the following methods:

```
__init__()
hash(key)
rehash()
```

Using Linear Probing, define a method **`__setitem__(self, the_key, data)`** which does the following:

- If there is an entry in the hash table with key, **`the_key`**, then it changes the value associated with **`the_key`** to **`data`**.
- If there is not entry with key, **`the_key`**, then an entry is inserted in the table with key, **`the_key`**, and value, **`data`**.

See opposite page for solution.

8

End of Exam