Question 1 – Short questions [10 marks = 10 x 1]

In this part you are required to answer ten short questions. Your answer should be concise. As a guideline, it should require no more space than the one provided.

(1) What is the purpose of using `jal` instead of `j` when working with functions in MIPS?
Jump and link will automatically store the return address in the `$ra`, which is necessary to that the program can return to the point where the function was called once the function finishes execution.
1.0 correct explanation, no partial marks.

(2) How can we avoid primary clustering in open addressing collision resolution? Explain your answer.
Primary clustering can be avoided by taking a different step each time the probing finds an occupied position. One example is quadratic probing, in which the step is incremented quadratically each time.
1.0 correct explanation, no partial marks.

(3) The best case time complexity for a method that traverses a list of elements can never be when the list is empty. Explain why.
Time complexity assumes a large input size, so an empty list is never a best case because $n = 0$
1.0 correct explanation, no partial marks.

(4) Mention and explain one disadvantage of using a Binary Search Tree over a Hash Table for implementing a Dictionary ADT.

- Compared to a relatively full hash table it will use more memory, since it needs to keep two references for each item on the structure
- Given a good hash function constant search far outperforms the logarithmic time of the BST.

1.0 correct argument, no partial marks.

(5) Can you modify a simple Linked List implementation to achieve constant time append to the end operations? Explain your answer.
Yes, by keeping a reference to the last element append to the end needs a constant number of operation regardless of how big the list is.
1.0 correct explanation, no partial marks.

(6) What is the difference between `is` and `==` in Python? Explain.
`is` checks if two variables point to the same object (are the same), i.e., same memory location, whereas `equals` checks if two objects (possibly in different memory addresses) are identical as given by an appropriate `__eq__` implementation.
1.0 correct explanation, no partial marks.

(7) What is the main advantage of a circular queue implementation over a linear one? Explain.
A circular queue avoids wasting space, by wrapping around the array that supports the queue.
1.0 correct explanation, no partial marks.

(8) When does the worst case time complexity arise in Bubble sort? Explain your answer.
Worst case for bubble sort is when the list is in reversed order, and you have to swap all the elements.
1.0 correct explanation, no partial marks.

(9) If the main concern is minimising the number of swaps, which of the non-recursive sorting algorithms (covered in the lectures) should be chosen and why?

Selection sort would do less swaps than bubble sort and insertion sort because it chooses max (min) and puts it at the end (beginning) with only one swap, thus would do n swaps in worst case .
1.0 correct explanation, no partial marks.

(10) Draw a tree with two nodes that is both a Binary Search Tree and a maxHeap. Explain your choice.
1.0 correct understanding of both concepts BST and Heap, no partial marks.

7.5

Question 2 – Array Containers [10 marks = 5 + 2 + 3]

The following code gives a partial implementation of a `DoubleStack` class, which holds two Stacks in **a single array**, one starting from the front and one from the back. The class should have two push operations (`push1` and `push2`), two pop operations (`pop1` and `pop2`), and three length operations: `len1`, `len2` and `len`, which give the number of items on stack 1, on stack 2, and on the double stack, respectively. As it can be inferred from the partial implementation, the left-hand side of the array is used for the first stack, while the right-hand side of the array will be used for the second stack. The double stack will be full only when the total number of elements equals the maximum capacity of the array, as given on the constructor.

```
from referential_array import build_array


class DoubleStack:

    def __init__(self, max_capacity=30):
        self.array = build_array(max_capacity)
        self.top1 = -1
        self.top2 = max_capacity

    def push1(self, item):
        if len(self) = len(self.array):
            raise Exception('Stack␣is␣full')
        else:
            self.top1 += 1
            self.array[self.top1] = item

    def len1(self):
        return self.top1 + 1
```

(a) Implement method `push2(self)`, which pushes `item` onto the second stack. This method should raise an Exception if there is no space available.

```
def push2(self, item):
    if len(self) = len(self.array):
        raise Exception('Stack␣is␣full')
    else:
        self.top2 -= 1
        self.array[self.top2] = item
```

- 1 mark for correct handling of full case
- 2 marks for correctly updating top
- 2 marks for assigning item to correct location

(b) Implement method `len2(self)`, which returns the number of items in the second stack.

```
def len2(self):
    return len(self.array) - self.top2
```

- 2 marks for correctly computing size, no partial marks

(c) Implement method `__str__(self)`, which returns a string representing the elements of the two stacks separated by an & symbol. For example, the following string `"1 2 3 & 0 -3 "` would be returned for a double stack where the first stack contained the elements 1, 2, 3 (where 3 is on top) and the second contained elements 0 and -3 (where 0 is on top). Make sure you do not modify the stacks.

```
def __str__(self):
    ans = ""
    for i in range(self.top1 + 1):
        ans += str(self.array[i])
        ans += '␣'
    ans += "&"
    for i in range(self.top2, len(self.array)):
        ans += str(self.array[i])
        ans += '␣'
    return ans
```

- 1 marks for correct indices and looping through first stack
- 1 marks for correct indices and looping through second stack
- 1 mark for correct output formatting

Reduce 0.5 mark for minor errors in each part.

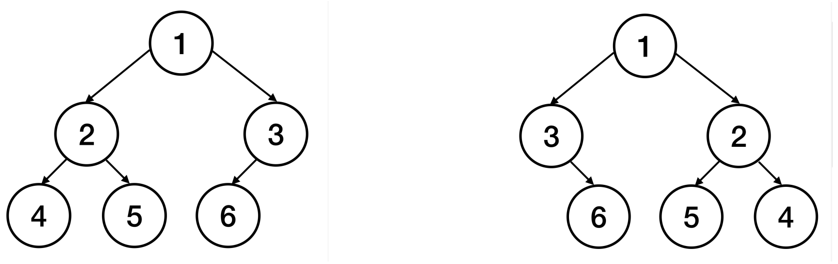
3

Question 3 – Binary Trees [12 marks = 5 + 3 + 2]

Consider the partial implementation of a binary tree given below, which uses the TreeNode class:

```
class TreeNode:

    def __init__(self, item=None, left=None, right=None):
        self.item = item
        self.left = left
        self.right = right

class BinaryTree:

    def __init__(self):
        self.root = None

    def is_empty(self):
        return self.root is None
```

(a) Implement the method `mirror(self)` within the `BinaryTree` class, which modifies the tree to make it the mirror image of itself. For example, if the method is applied to the binary tree on the left, it will modify it to be the one on the right, and vice-versa.

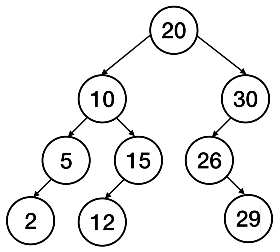


```
def mirror(self):
    return self.mirror_aux(self.root)

def mirror_aux(self, current):
    if current is None:
        return current
    left = self.mirror_aux(current.left);
    right = self.mirror_aux(current.right);
    current.left = right
    current.right = left
    return current
```

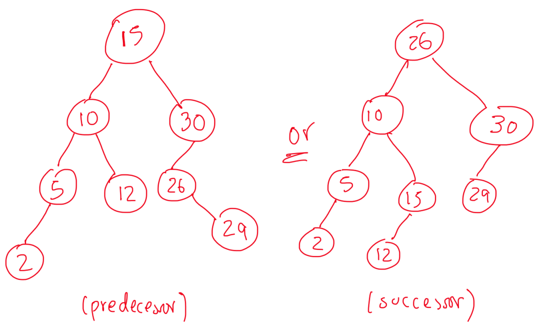
- 1 mark for correctly setting up first call and signature of auxiliary method
- 1 mark for correct base case
- 2 mark for correct recursive calls and concept
- 1 mark for correct update of variables after recursion

Reduce marks (1 or 0.5) for minor mistakes in each part.

(b) Assume you want to delete the root node in the Binary Search Tree shown on the right. Briefly enumerate the steps required to do so, and provide the resulting Binary Search Tree.



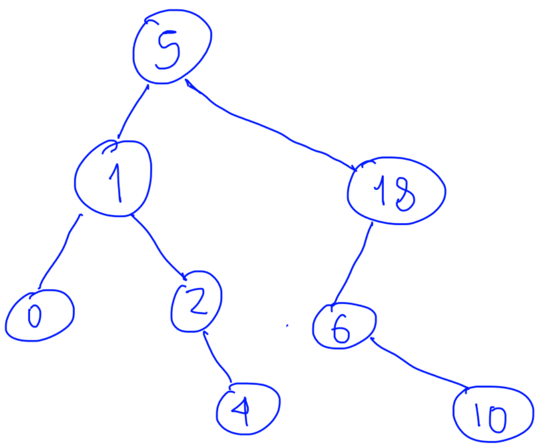
- We first find the predecessor (or successor) of the root, which in this case is 15 (26).
- Node 15 (26) should be the new root.
- This in turns leaves a new single orphan, 12. Because 15 had only one children in the original binary search tree, we point the parent (10) to the new orphan (12) – likewise for other case.



The output is as above.

- 1 marks for correctly explaining how to apply criterion involving *successor* for deleted node with two children,
- 1 marks for applying simpler criterion to resolve resulting orphan after relocating root.
- 1 mark for drawing a correct BST without root.

(c) Consider an empty Binary Search Tree to which you add the integers 5, 1, 18, 6, 2, 0, 4, 10 in that order. Draw the tree obtained after the last insertion.



- 1 marks if solution is correct BST including all elements, or, 2 marks if solution also follows procedure in class (see figure).

Question 4 – MIPS [10 marks = 3 + 7]

(a) Consider the following Python code that constructs and returns a string:

```
def hello(name):
    out = "Hello␣" + name + ",␣great␣to␣meet␣you."
    return out
```

In MIPS, local variables are created (allocated) during function entry and destroyed (deallocated) upon function exit. That is why they are stored in the System Stack. However, the MIPS translation of Python code to `x = hello("Maria"), print(x)` will successfully print `Hello Maria, great to meet you.` How is this possible given variable `out` has been deallocated by the time the string is printed? Briefly explain in terms of memory allocation.

Inside the function a local variable out will be created, pointing to the resulting string. The string is in the heap, but upon return the address is stored in the variable x of the main function. Although the variable out has been destroyed, the address of the resulting string is passed upon function returning.

- 3 marks for evidence of understanding local variables and return value in `$v0`.
- reduce 2 marks or 1 mark if too vague or vague.

3

(b) In the space next to the Python code, complete the MIPS program so that it constitutes a **faithful** translation of the original Python program, where `sec(a)` is a call to a function defined elsewhere. Remember to comment the code!

- 2 marks for set up first function (fp, ra, and sp)
- 2 marks allocate locals of *first* and call *sec*
- 1.5 marks for cleaning after sec (deallocate locals and store result)
- 1.5 for function exit for *first*

```
first:
    addi $sp, $sp, -8    # store $fp and $ra
    sw $fp, ($sp)
    sw $ra, 4($sp)
    addi $fp, $sp, $0    # Move $fp up


    addi $sp, $sp, -4    # store a on the stack
    lw $t0, 8($fp)
    sw $t0, -4($fp)
    jal sec              # sec(a)

    addi $sp, $sp, 4     # deallocate argument
    lw $t0, 12($fp)      # load i
    add $v0, $t0, $v0    # i + sec(a)

    lw $fp, ($sp)        # restore $ra and $fp
    lw $ra, 4($sp)
    addi $sp, $sp, 8

    jr $ra
```

Question 5 – Iterators [10 marks = 8 + 2]

In this question, you will be asked about iterators and their usage.

(a) Write up a class `FactorIterator` that can be used to iterate over all the factors of a given number n. That is, the code

```
x = FactorIterator(8)
it = iter(x)
next(it)
next(it)
next(it)
next(it)
```

will bound `it` to an iterator and print numbers 1, 2, 4 and 8, in that order.

```
class FactorIterator:

    def __init__(self, n):
        self.n = n
        self.i = 0

    def __iter__(self):
        return self

    def __next__(self):
        self.i += 1
        if self.i > self.n:
            raise StopIteration()
        while self.n % self.i != 0:
            self.i += 1
        return self.i
```

- 2 marks if the iterator has correct signatures for the three required methods (`__init__`, `__iter__` and `__next__`)
- 1 mark `__iter__` returns `self`
- 2 marks correctly raising `StopIteration`
- 3 marks for correct logic of `__next__`

8

(b) Write a Python function `odd_factors(n)` that *uses the iterator you just defined* to print all the odd factors of number `n` (e.g.,. for n=900, it would print 3,5,9, and 15)

```
def odd_factors(n):
    p = FactorIterator(n)
    for fact in p:
        if fact%2 ==1:
            print(fact)
```

- 1 mark for using the iterator correctly
- 1 mark logic for correctness

Question 6 – Classes, Objects and Namespaces [10 marks]

Provide next to the Python code below, the result of each of the print statements (marked with comments from #1 to #11). If the result is an error, please explain why.

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

    def print(self):
        print(self.x)

    def a(self):
        self.x = self.x + 1

    def b(self):
        self.x = x + 2

    def c(self):
        x = self.x + 3

def a(x):
    x = x - 1

def b():
    x = x + 1

x = 1
myobject = myclass(x)
myobject.print() #1
x = 2
myobject.print() #2
myobject.x = 1
myobject.a()
myobject.print() #3
myobject.b()
myobject.print() #4
myobject.c()
myobject.print() #5
a(x)
print(x) #6
myclass.x = 1
myclass.print(myobject) #7
myobject.x = 2
myclass.print(myobject) #8
myobject.c()
print(myclass.x) #9
yourobject = myclass(myobject.x)
yourobject.a()
myobject.print() #10
```

5

```
#1:
1
#2:
1
#3:
2
#4:
4
#5:
4
#6:
2
#7:
4
#8:
2
#9:
1
#10:
2
```

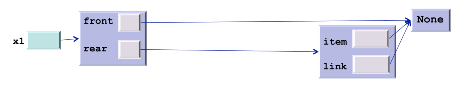
- 1 mark per each correct print.

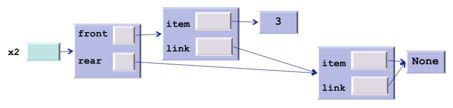
Question 7 – Linked Queues [10 marks = 5 + 5]

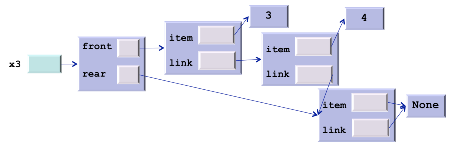
An invariant of the array-based queue implementation we have seen in the lectures, is that `self.rear` always points to the first empty slot at the rear of the queue. To replicate this `self.rear` invariant in the linked queue, consider the partial implementation given below, where the rear of the queue points to an empty but already created node, that is ready to be filled :

```
class Node:

    def __init__(self, item=None, link=None):
        self.item = item
        self.link = link

class LinkedQueue:

    def __init__(self):
        self.front = None
        self.rear = Node()

    def is_empty(self):
        return self.front is None
```

For example, the figures below show an empty queue (`x1`), a queue with one element (`x2`) and one with two elements (`x3`).







(a) Implement the usual queue method `append(self, item)` within the `LinkedQueue` class above.

```
def append(self, item):
    if self.is_empty():
        new_node = Node(item, self.rear)
        self.front = new_node
        self.rear = new_node.link
    else:
        self.rear.item = item
        self.rear.link = Node()
        self.rear = self.rear.link
```

- 2 mark for checking special case empty queue
- 1 marks correctly adding new Node with given item
- 2 marks if rear always ends pointing to empty Node instead of None

4

(b) Likewise, implement the method `serve(self)` within the `LinkedQueue` class above. The method should accommodate all boundary cases including serving the last element in the queue.

```
def serve(self):
    # if queue is empty
    assert not self.is_empty(), "queue␣is␣empty"
    item = self.front.item
    self.front = self.front.next
    # if the queue has only one element to serve
    if self.front.item is None:
      self.front = None
    return item
```

- 1 mark for correctly handling empty case with exception or assertion
- 2 marks for correctly handling the case with more than 1 node
- 2 marks for correctly handling case with one element only, making sure that `self.front` ends up pointing to `None` on completion

Question 8 – Sorting [8 marks = 6 + 2 + 2]

You are a lecturer in charge of a unit with m students and k tutors. Your tutors have marked $\frac{m}{k}$ exam papers each, and you need to sort all papers according to their marks. We suppose that comparing marks (and thus comparing exams) takes constant time.

You propose the following distribution of work: each tutor will sort their $\frac{m}{k}$ papers, and you, the lecturer, will take those k lists of papers and merge them into a single sorted list of m papers. We suppose that each tutor requires exactly p^2 operations to sort p papers, and that you require exactly $k * p$ operations to merge those k parts of p papers into a single sorted list

(a) What is the total number of operations required for this algorithm? Explain.

Each tutor takes $\frac{m}{k}$ papers, which means sorting per tutor takes $\frac{m^2}{k^2}$. This work for sorting is repeated k times, thus total sorting time is $k\frac{m^2}{k^2}$. For merging we take k parts time m/k per part for a total m. So total running time is $\frac{m^2}{k} + m$.

- 3 marks for sorting reasoning
- 1 marks for merging reasoning
- 2 marks for correct answer

(b) For m fixed, what is the number k of tutors that minimises the total number of operations required? Explain.

Running time is minimised when $k \to \infty$. More tutors always improve the running time. Other possible answers are m or m^2, with appropriate reasoning.

- 2 marks for correct reasoning

(c) For m fixed, what is the number k of tutors that gives each tutor the same amount of work (i.e. operations) as the lecturer? Explain.

Work per tutor is $\frac{m^2}{k^2}$, the lecturer work is m, thus the operations are equal when $k^2 = m$. Thus work is the same when $k = \sqrt{m}$

- 2 marks for correct reasoning

6

Question 9 – Priority Queues and Heaps [10 marks]

This question is about priority queues and heaps. Consider a priority queue implemented by a max Heap, where its elements are key value pairs stored using a tuple:

```
class MaxHeap:
        def __init__(self, max_capacity):
              self.count = 0
              self.the_array = build_array(max_capacity+1)
        # code with the rest of the functions provided by the class
```

Assume the methods `rise(self, k)` and `sink(self, k)` are correctly implemented. Write a method `changePriority(self, key, value)` for this class, which first finds the element with the given value v (note that values are unique) and then changes the priority (*key*) of that element to some new value k, raising an exception if not found.

```
def changePriority(self,newKey,oldValue):
     # first, search old key
     pos =0
     for i in range(self.count+1):
          if self.the_array[i][0] == oldValue:
               pos = i
     # if position not found, raise exception
     if pos == 0:
          raise Exception("key␣does␣not␣exist")
     # otherwise, update key and place relocate to right position
     self._array[pos] = (key, newKey)
     self.rise(pos)
     self.sink(pos)
```

- 2 marks for finding the right position by looking at keys
- 2 marks for correct handling of tuples – indices (key, value)
- 2 mark for handling key not found
- 4 marks for correct positioning of new key using rise and sink

Question 10 – Hash Tables [10 marks = 3 + 4 + 3]

Consider a hash table implementation where collisions are handled with quadratic probing. Also, the implementation uses method `rehash(self)` that is called after adding an element to the table if the load factor reaches or exceeds 1/2. The method will double the size of the table and reinsert each key.

(a) Assuming the initial size of the table is 4, what is the size of the table after inserting 12 different keys into the above hash table. Explain.

Initial table size is 4, so load 0.5 is first hit at 2 elements. With table size 8 the target load is reached at 4 elements. With table size 16 the target load is reached at size 8, and the table size doubles to 32. With 12 elements the load is $12/32 \approx 0.3$. Thus, final table size is 32.

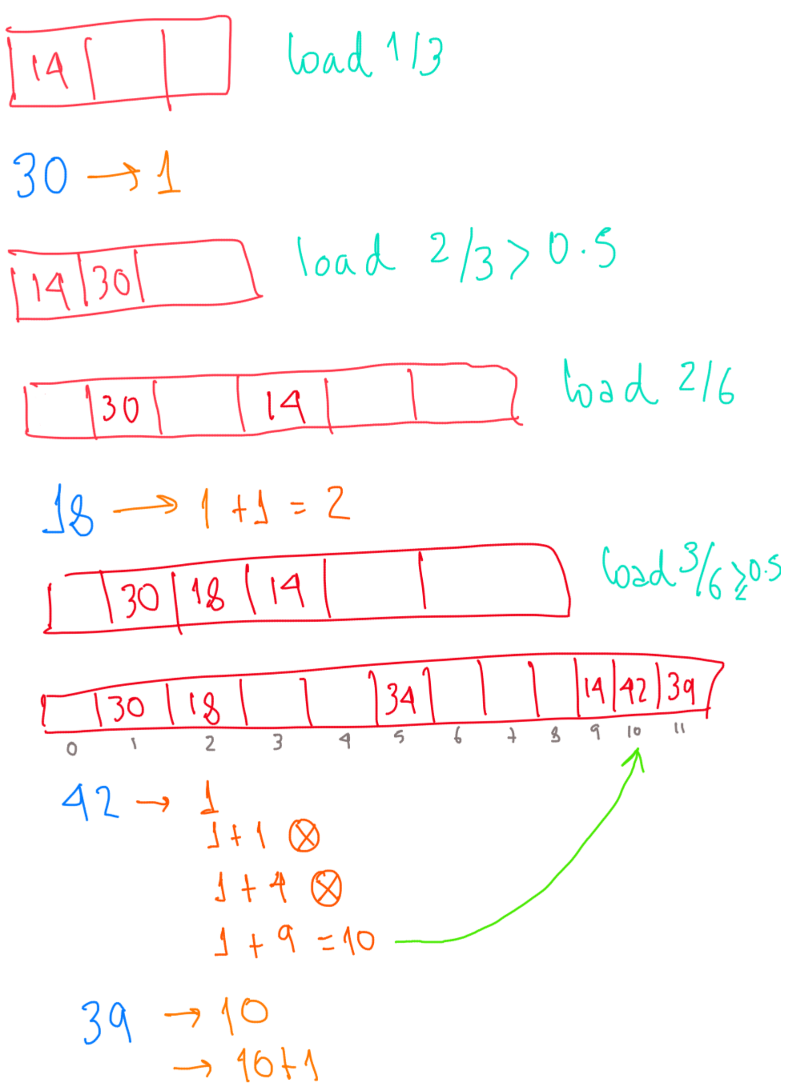
- 1 mark for correct understanding of load
- 2 marks for correct sequence of doubling size, and correct anwer.

(b) Draw the content of the hash table after inserting keys 30, 18, 34, 42, 39 (with associated hashed value given by the table below), assuming the initial size of the table is 3 and there is already one key (14) inside as shown below. Note that the method `rehash(self)` will need to be called while you are adding the keys, as described above.

Table Size	Key	Hashed Value
3	14	0
	30	1
	18	1
6	14	3
	30	1
	18	1
	34	5
	42	1
	39	4
12	14	9
	30	1
	18	1
	34	5
	42	1
	39	10

0	1	2
14		

7



If evidence of process:

- 2 marks for correct handling of collisions using quadratic probing
- 2 mark for correct doubling size with load

Otherwise 4 marks if answer correct or 0 otherwise.

(c) Suppose that, instead of using quadratic probing to handle collision, the hash table uses separate chaining implementation using sorted linked list. What would be the best-case and worst-case time complexity of this implementation during insertion? Explain.

Best case is no collisions, insertion is constant time as space is readily found.
Worst case if collision on location containing all elements, complexity is $O(n)$ finding correct location inside linked list.

- 1.5 mark correct reasoning best case
- 1.5 mark correct reasoning worst case

END OF EXAM.

7