Question 1 [10 marks]

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

One mark per correct answer. No partial marks.

(1) In MIPS, how many bits are required to store a word?
32 bits

(2) In MIPS, how many bytes are required to store an array of 6 integers?
$(6+1)*4 = 28$ bytes $= 224$ bits

(3) Recursion is usually memory intensive because... (**Hint:** Use your MIPS knowledge)
The arguments of the function need to be copied onto the stack every time the function is called.

(4) In the worst-case time complexity scenario, Merge-sort outperforms Quick-sort. However, quick-sort is often a better choice because...
The algorithm is in place and does not require extra memory. Moreover the worst case of Quick-sort is very unlikely.

(5) The two main operations of a Stack ADT are?
Push and Pop.

5

(6) How does a Circular Queue differ from a standard array-based Queue?
The circular Queue does not waste space, by allowing the rear and the front of the Queue to wrap around each other.

(7) List 3 container ADT's covered in the lectures
3 of: Stack, List, Queue, and their variants. Dictionary, Heap.

(8) A class variable is...
A variable that is shared by all instances of a given class.

(9) The instance variables of a simple Node to support a Linked Structure are....
Item to hold what needs to be stored, and a reference *next* to the next node.

(10) By convention, a parameter `self` in a method definition refers to...
A reference to the object that is calling the method.

5

Question 2 [5 marks = 3 + 2]

This question is about sorting. Consider the following implementation of `selection_sort`.

```
def selection_sort(a_list):
    n = len(a_list)
    for k in range(n-1, -1, -1):
        max_position = find_max(a_list, k)
        a_list[k], a_list[max_position] = a_list[max_position], a_list[k]
```

(a) Using Python, define the function `find_max(a_list, limit_index)` that completes the implementation.

```
def find_max(a_list, limit_index):
    max_position = 0
    n = len(a_list)
    for i in range(0, limit_index+1):
        if a_list[i] > a_list[max_position]:
            max_position = i
    return max_position
```

- 1 mark for iterating correctly through the sub-list given by index k
- 1 mark for keeping track of the current max element
- 1 mark for returning the correct answer

(b) Is Selection sort as implemented above stable? Explain your answer. Selection sort is not stable. Selecting the max element and **swapping** with the element at the end does not guarantee that the relative order of the elements is maintained.

- 1 mark for stating that it is unstable
- 1 mark for explaining / evidence of understanding stability

5

Question 3 [8 marks = 2 + 2 + 2 + 2]

This question is about time complexity. For algorithms (a) to (d) express their Big-O notation time-complexity in the best and worst case. Provide a short explanation in each case. No explanation means no marks.

(a)
```
def algorithm_a(a_list):
    n = len(a_list)
    for k in range(n-1):
        a = k
        for i in range(k+1, n):
            if a_list[i] < a_list[a]:
                a = i
        a_list[k], a_list[a] = a_list[a], a_list[k]
```

Best time complexity: $\underline{O(n^2)}$. Worst time complexity: $\underline{O(n^2)}$
Explanation: Two nested loops. There is no way to leave any of the two loops early.

- 1 mark explained best case
- 1 mark explained worst case

(b)
```
def algorithm_b(a_list):
    n = len(a_list)
    for k in range(0, n-1):
        position = 0
        for i in range(k, -1, -1):
            if a_list[i] == a_list[position]:
                break
            else:
                position += 1
        a_list[k], a_list[position] = a_list[position], a_list[k]
```

Best time complexity: $\underline{O(n)}$. Worst time complexity: $\underline{O(n^2)}$
Explanation: Two nested loops. In the best case the second loop is terminated after a constant number of instructions, which yields linear time.

- 1 mark explained best case
- 1 mark explained worst case

4

(c)
```python
def algorithm_c(a_list):
    return a_list[-1]
```

Best time complexity: O(1). Worst time complexity: O(1)

Explanation: Accessing the last element is constant time regardless of the size of the array.

- 1 mark explained best case
- 1 mark explained worst case

(d)
```python
def algorithm_d(a_list, item):
    a = 0
    b = len(a_list) -1
    while a < = b:
        c = (a+b)//2
        if a_list[c] == item:
            return c
        elif a_list[c] > item:
            b = c - 1
        else:
            a = c + 1
    return -1
```

Best time complexity: O(1). Worst time complexity: $O(\log n)$

Explanation: Best case if item is equal to the element in the middle of the list, in which case only a constant number of operations execute. Worst case when item is not in the list, the problem is reduced by half each time, which yields logarithmic time.

- 1 mark explained best case
- 1 mark explained worst case

4

Question 4 [7 marks = 2 + 2 + 3]

This question is about Stacks. Consider the partial implementation of a Stack ADT below:

```
class Stack:
    def __init__(self, size):
        assert size > 0, "size should be positive"
        self.array =  size * [None]
        self.count =  0
        self.top = -1

    def is_full(self):
        return self.count > = len(self.array)

    def is_empty(self):
        return self.count == 0
```

(a) Implement the method `push(self, item)` using an assertion to check for the precondition.

```
def push(self, item):
        assert not self.is_full(), "The stack is full"
        self.array[self.count] = item
        self.count += 1
        self.top += 1
```

- 0.5 using assertion or exception with precondition
- 0.5 insert element in correct array position
- 0.5 keep count correctly
- 0.5 update top correctly

(b) Implement the method `pop(self)` using an assertion to check for the precondition.

```
def pop(self):
    assert not self.is_empty(), "Stack is empty"
    item = self.array[self.top]
    self.top -= 1
    self.count -= 1
    return item
```

- 0.5 using assertion or exception with precondition
- 0.5 return element at the top
- 0.5 keep count correctly
- 0.5 update top correctly

4

(c) Consider the method `factorial` below, which relies on recursion.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

Provide a version of `factorial` which uses the Stack implementation to replace recursion.

```
def factorial_stack(n):
    my_stack = Stack(n)
    for i in range(1, n+1):
        my_stack.push(i)
    ans = 1
    while not my_stack.is_empty():
        ans = ans* my_stack.pop()
    return ans
```

This is one possible solution.

- 1 mark for using the Stack data type interface correctly.
- 2 marks for correctness of the algorithm, partial marks for minor mistakes

3

Question 5 [6 marks = 2 + 2 + 2]

This question is about linked structures. Consider the following partial implementation of a linked SortedList.

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

class SortedList:
    def __init__(self):
        self.head = None
        self.count = 0

    def _getnode(self, index):
        assert 0 <= index <= self.count, "index out of bounds"
        node = self.head
        for _ in range(index):
            node = node.next
        return node
```

(a) Define the method `add(self, item)`, which adds one item to the list keeping the list sorted.

```
def add(self, item):
        # if empty, trivial
        if self.head is None:
            self.head = Node(item)
            self.count +=1
            return
        # find right position
        current = self.head
        previous = None
        while current is not None:
            if current.item < item:
                previous = current
                current = current.next
            else:
                break
        if previous is None:
            self.head = Node(item, self.head)
            self.count +=1
        else:
            previous.next = Node(item, current)
            self.count +=1
```

This is one possible solution.

- 0.5 handling correctly edge cases, empty list or list with one node.
- 0.5 correctly finding the correct position while iterating on the list
- 0.5 handling correctly the pointers once the new location is identified
- 0.5 updating count appropriately.

(b) What is the best and worst-case time complexity of a correct and efficient implementation of `add(self, item)` for this data type. Explain your answer. Best case is $O(1)$, when inserting smallest element. Worst case is $O(n)$, inserting largest element. 1 mark for correct best explained, 1 mark for correct worst explained.

4

(c) Define the method `__next__(self)`, of the Iterator below, which is intended to go through all elements of the Sorted List defined above.

```
class SortedListIterator:
    def __init__(self, head):
        self.current = head
    def __iter__(self):
        return self
```

```
def __next__(self):
        if self.current == None:
            raise StopIteration
        else:
            item_required = self.current.item
            self.current = self.current.next
            return item_required
```

This is one possible solution.

- 0.5 marks for disregarding order during iteration
- 0.5 marks returning an item and not a node
- 0.5 marks for correctly rising a `StopIteration` exception
- 0.5 marks for correctly updating current before returning

2

Question 6 [6 marks = 3 + 3]

This question is about recursion.

(a) The greatest common divisor (GCD) of two integer numbers is the largest positive integer that divides the numbers without a remainder. Convert the following iterative version of the GCD algorithm into a recursive algorithm.

```
def gcd(a, b):
    while(b != 0):
        r = a%b
        a = b
        b = r
    return a
```

```
def recursive_gcd(m, n):
    if m % n == 0:
        return n
    else:
        return recursive_gcd(n, m % n)
```

- 1 mark for using recursion
- 1 mark for base case, and smaller recursive case
- 1 mark for correctness

(b) Provide a tail-recursive version of the following algorithm::

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-2) + fib(n-1)
```

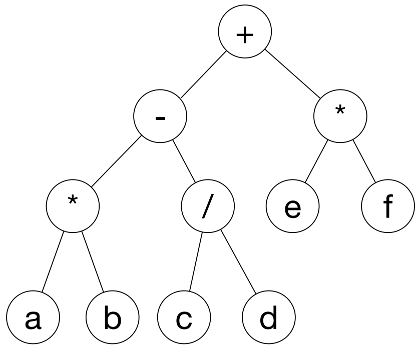
```
def fib(n):
    return fib_aux(n, 0, 1)

def fib_aux(n, before_last, last):
    if n  == 0:
        return before_last
    else:
        return fib_aux(n-1, last, before_last + last)
```

- 1 mark for attempt tail recursive – return a call to function
- 1 mark for correctly using an auxiliary function to pass first call args, using an accumulator argument
- 1 mark for correctness

6

Question 7 [10 = 1 + 2 + 2 + 5 marks]

This question is about Binary Trees and Binary Search Trees. Consider the graph below, which represents an expression tree:



(a) What is the infix arithmetic expression given by this Binary Tree?
$((a * b) - (c/d)) + (e * f)$
1 mark for correct answer.

(b) List the sequence of characters as they occur when you traverse the tree above in pre-order?
$+ - *ab/cd * ef$
2 marks for correct answer. 1 mark for minor mistakes.

(c) List the sequence of characters as they occur when you traverse the tree above in post-order?
$ab * cd/ - ef * +$
2 marks for correct answer. 1 mark for minor mistakes.

5

(d) For the BinarySearchTree data type defined below, write down the recursive method `_insert_aux(self, current, key, item):`

```
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

    def insert(self, key, item):
        self._insert_aux(self.root, key, item)
```

```
def _insert_aux(self, current_node, key, value):
        if current_node is None:
            current_node = BinaryTreeNode(key, value)
        elif key < current_node.key:
            current_node.left =  self._insert_aux(key, value, current_node.left)
        elif key > current_node.key:
            current_node.right = self._insert_aux(key, value, current_node.right)
        elif key == current_node.key:
            current_node.value = value
        return current_node
```

- 1 mark for 4 arguments in aux func, including self.
- 1 mark for handling empty tree edge case
- 1 mark for assigning on return from recursive call
- 1 mark insert on the correct side given comparison of keys
- 1 mark returning correct reference at the end

5

Question 8 [10 marks = 3 + 3 + 4]

This question is about Heaps. Consider the partial implementation of a Max Heap.

```
class Heap:
    def __init__(self):
        self.count = 0
        self.array = [None]

    def __len__(self):
        return self.count

    def add(self, item):
        if self.count + 1 < len(self.array):
            self.array[self.count + 1] = item
        else:
            self.array.append(item)
        self.count += 1
        self.rise(self.count)

    def swap(self, i, j):
        self.array[i], self.array[j] = self.array[j], self.array[i]

    def get_max(self):
        item = self.array[1]
        self.swap(1, self.count)
        self.count -= 1
        self.sink(1)
        return item

    def sink(self, k):
        while 2*k <= self.count:
            child = self.largest_child(k)
            if self.array[k] >= self.array[child]:
                break
            self.swap(child, k)
            k = child
```

(a) Implement the method `rise(self, k)` which complements the `add` function.

```
def rise(self, k):
        while k > 1 and self.array[k//2] < self.array[k]:
            self.swap(k, k//2)
            k //= 2
```

- 1 mark for reducing k by half on every iteration
- 1 mark for correct loop condition
- 1 mark for correct swapping of elements

3

(b) Implement the method `largest_child(self, k)` which complements the `sink` function

```
def largest_child(self, k):
    if 2 * k == self.count or self.array[2 * k] > self.array[2 * k + 1]:
        return 2 * k
    else:
        return 2 * k + 1
```

- 1 mark correctly handle the case where the tree is not full, i.e., node k has only one child
- 1 mark for comparing $2 * k + 1$ and $2 * k$ children
- 1 mark for choosing correct child

(c) Using only the methods defined above define a function `get_minimum(a_max_heap)` that returns the minimum element of the Heap in $O(N)$. The parameter of the function is assumed to be a Max-Heap. Your method **can** modify the Heap if necessary, but only through the operations of the Data Type (i.e., you should not access instance variables directly).

```
def get_minimum(a_max_heap):
    n = len(a_max_heap)
    for i in range(n):
        last = a_max_heap.get_max()
    return last
```

- 2 marks for correctness
- 1 mark for using only defined functions in the type
- **This solution is actually not linear so we will probably disallow this question**

7

Question 9 [**9 marks = 3 + 3 + 3**]

This question is about Hash Tables. Keep answers short using the space provided only.

(a) Explain how quadratic probing is used to resolve collisions in a HashTable.
Quadratic probing defines the sequence of array positions that are visited when there is a collision in position h. After a collision it will first look in position $h+1^2$, then position $h+2^2$, then $h+3^2$ and so on.

- 3 marks for evidence of understanding quadratic probing, partial marks for minor misunderstandings.

(b) Explain how separate chaining is used to resolve collisions in a HashTable.
Separate chaining uses a linked list in each position of the hash table. If a collusion occurs it simply adds the element to the list.

- 3 marks for evidence of understanding separate chaining, partial marks for minor misunderstandings

(c) If you were given a perfect hash function, would collision handling be necessary? Explain why/why not. Perfect hash functions always assign different locations, thus, provided a sufficiently large table no collision handling is necessary.

- 3 marks for evidence of understanding perfect hashing, partial marks for minor misunderstandings.

9

Question 10 [11 marks]

Translate to MIPS faithfully using only the instructions available in the reference sheet and following the function calling convention discussed in the lectures.

Python Code	MIPS Code
a = 5 a_abs = 0	.data a: .word -6 a_abs: .word 0 .text

def my_function(x):
 	
    if x > 0:
        return x
 	
    else:
 	
        return -x
	my_function:
	#CALLEE PREP: 1. save $ra and $fp on stack
	addi $sp, $sp, -8
	sw $ra, 4($sp)
	sw $fp, 0($sp)
	#CALLEE PREP: 2. copy $sp into $fp
	addi $fp, $sp, 0
	#CALLEE PREP: 3 ALLOCATE LOCAL VARIABLES
	# No local variables
	#BUSINESS
	# if x > 0 return x
	lw $t0, 8($fp) #t0 = x
	blt $t0, $0, else
	j end
	else:
	lw $t0, 8($fp) #t0 = x
	addi $t1, $0, -1
	mul $t0, $t1, $t0
	end:
	add $v0, $t0, $0 #CALLEE CLEAN: 1. $v0 to return value
	#CALLE CLEAN: 2. deallocate local variables
	#no local variables
	#CALLEE CLEAN: 3 restore saved $ra
	lw $fp, ($sp)
	lw $ra, 4($sp)
	addi $sp, $sp, 8
	#CALLEE CLEAN: 4 return to caller
	jr $ra

a_abs = my_function(a)
 	
print(a_abs)
	main:
	#CALLER PREP: 1. save temp registers -- none
	
	#CALLER PREP:2. pass arguments on stack
	addi $sp, $sp, -4 # space for one argument
	lw $t0, a # t0 = a
	sw $t0, 0($sp) #copy argument
	
	jal my_function
	
	#CALLER CLEAN: 1 clears arguments off stack
	addi $sp, $sp, 4 # 1 argument
	
	#CALLER CLEAN: 3. use return value in $v0
	sw $v0, a_abs
	
	lw $a0, a_abs # print a_abs
	li $v0, 1
	syscall

- 1 mark for storing global variables in data segment
- 4 marks for applying all steps in function calling/returning convention
- 4 marks for correct business logic (printing, decisions, etc)
- 2 marks for correct interactions between memory and registers

11

Question 11 [12 marks = 4 + 4 + 4]

For each situation described in the rows of the table below, choose one of the following structures: Linked List (1), Array-based List (2), SortedList Array-based (3), or Sorted Linked-List (4). Explain briefly the reason behind your choice in the space provided. **In each row, 1 mark for choosing a reasonable data type and 3 marks for demonstrating knowledge about the data type and reasoning behind choice**

Situation	Choice of Data Type	Explanation
A post office needs to store in a list a record for each packet being processed. Postal demand is volatile so the number of packages arriving each day is unpredictable. It is not necessary to keep track of the order.		
A University needs to keep a list of students. Demand is predictable so the approximate size of the list is known and after enrolments not a lot of changes are needed in terms of additions or deletions. The students do not need to be sorted.		
A University needs to keep a list of employees. The approximate size of the list is known and new items or removals happen infrequently. The critical functionality to be provided is search by name.		

12

Question 12 [6 marks = 1 × 6]

In MIPS, the function calling convention has the following steps. For each step, explain in the space provided **why** this action is required by the convention.

(1) Caller saves temporary registers on the stack.
To avoid over-writing temporary registers that may be needed un-tampered outside of the function.
1 mark

(2) Caller passes arguments on to the stack.
An alternative to using the stack is using registers so that the function can access the arguments, but this would imply a maximum and fixed numger of arguments, therefore the stack is the way to do it.
1 mark

(3) Caller calls function using `jal`.
The function is a piece of code that is reusable so we can give it a label, the `jal` allows us to also keep track of the return address to keep the code going after we are finished with the function .
1 mark

3

(4) Callee saves `ra` and `fp` on the stack.
Because functions can call other functions is important to re-store the stack and the line from which the code is running to the right place. Saving the `ra` and the `fp` on the stack allows us to recover the state when leaving any function .
1 mark

(5) Callee copies `sp` to `fp`.
This step moves the frame pointer to allow for easy access to function's local variables and arguments. This steps defines the stack frame of the function.
1 mark

(6) Callee allocates local variables on the stack.
Local variables are only accessible to the function while the function is running, so they belong in the stack frame of the function.
1 mark

END OF EXAM.

3