Office Use Only

MONASH University

Semester One 2016
Examination Period

Faculty of Information Technology

EXAM CODES: **FIT1008 / FIT2085**

TITLE OF PAPER: **INTRODUCTION TO COMPUTER SCIENCE - PAPER 1**

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 ☐ Monash Extension ☐ Sth Africa
☐ Parkville ☐ Other (specify)

During an exam, you must not have in your possession any item/material that has not been authorised for your exam. This includes books, notes, paper, electronic device/s, mobile phone, smart watch/device, calculator, pencil case, or writing on any part of your body. Any authorised items are listed below. Items/materials on your desk, chair, in your clothing or otherwise on your person will be deemed to be in your possession.

No examination materials are to be removed from the room. This includes retaining, copying, memorising or noting down content of exam material for personal use or to share with any other person by any means following your exam.
Failure to comply with the above instructions, or attempting to cheat or cheating in an exam is a discipline offence under Part 7 of the Monash University (Council) Regulations.

AUTHORISED MATERIALS

OPEN BOOK	☐ **YES**	**X NO**
CALCULATORS	☐ **YES**	**X NO**
SPECIFICALLY PERMITTED ITEMS **if yes, items permitted are:**	☐ **YES**	**X NO**

Page	Marks	Page	Marks
3		21	
5		23	
7		25	
9		27	
11		29	
13		31	
15		33	
17		35	
19		**Total**	

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

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

This page intentionally left blank, use if needed but it will not be marked.

Question 1 [10 marks]

This question is about MIPS programming and function calls. Translate the following Python code faithfully into MIPS assembly language. Make sure you follow the MIPS function calling and memory usage conventions as discussed in the lectures. Use only instructions in the MIPS reference sheet.

Python Code	MIPS Code
`def func(n):`	
`    if n == 1:`	
`        return 1`	
`    else:` `        return n * func(n//2)`	

10

This page intentionally left blank, use if needed but it will not be marked.

Question 2 [7 marks]

This question is about Iterators. Using Python define a `NegativeIterator` iterator class. This Iterator should work with a standard Python list. An instance of this class iterates through all the negative elements of the list without modifying the list.
For example:

```
>>> itr = NegativeIterator([3,-8,-6,0,-11])
>>> next(itr)
-8
>>> next(itr)
-6
>>> next(itr)
-11
>>> next(itr)
Stop Iteration
```

Your class must have the three methods: `__init__`, `__iter__` and `__next__`.

7

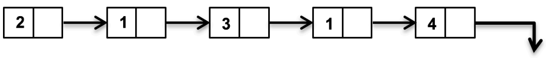
This page intentionally left blank, use if needed but it will not be marked.

Question 3 [8 marks]

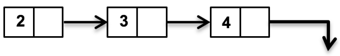
Consider the two classes `Node` and `List` as seen in the lectures, which define a **List ADT** implemented using a **linked structure**:

```
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
```

Define the method `delete_item(self, item)`, which deletes all the items in the list that have the value `item`. For example, assume `alist` is a List and `alist.head` points to the first node in the following structure:



After calling `alist.delete_item(1)`, `alist` should have the following structure with all the items of value `1` removed:



8

This page intentionally left blank, use if needed but it will not be marked.

Question 4 [6 marks = 5 + 1]

This questions is about Heaps.

(a) Suppose a `min-heap` is represented using an array. Write a Python function `def is_valid_heap(array)`, which given an `array` returns `True` if the `array` represents a valid `min-heap`, and returns `False` otherwise.

(b) Using Big-O notation, provide and explain the worst-time complexity of the function you defined in part *a*. No explanation means no marks.

6

This page intentionally left blank, use if needed but it will not be marked.

Question 5 [10 marks = 2 + 3 + 3 + 2]

This question is about sorting algorithms.

(a) What is a stable sorting algorithm?

(b) Is selection sort stable? Explain your answer **and** provide an example to make your case.

5

This page intentionally left blank, use if needed but it will not be marked.

(c) Using Python, write a function `def insertion_sort(a_list)`, which takes as input a list of numbers and sorts this list into increasing order using insertion sort.

(d) What are best and worst-case time complexity for Insertion Sort in Big-O notation. When do they occur? Explain your answer (no explanation means no marks).

5

This page intentionally left blank, use if needed but it will not be marked.

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

This question is about Time complexity. For each of the given Python functions, state and explain the complexity in Big-O notation. If appropriate discuss best and worst cases. No explanation means no marks.

(a)
```
def func_a(the_list):
    total = 0
    for item in the_list:
        for i in range(5):
            total = total + i * item
```

(b)
```
def func_b(a, b):
    while a > b:
        print(str(a-b))
        a = a - 1
        b = b + 1
```


4

This page intentionally left blank, use if needed but it will not be marked.

(c)
```
def func_c(the_list):
    total = 0
    for i in range(len(the_list)):
        total = total + the_list[i]
        if i < 0:
            break
```

(d)
```
def func_d(n):
    while n > 0:
        print(n)
        n = n // 2
```


4

This page intentionally left blank, use if needed but it will not be marked.

Question 7 [10 marks]

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

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

You also have a `Stack` class which implements a Stack ADT using some data structure (you do not need to know which one) and defines the following methods:

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

Using Python, write a function `def magnitude(a_queue)`. This function takes as input a queue of numbers sorted in increasing order. The function then returns a new queue with the numbers sorted according to their absolute value, in increasing order. For example: given a queue [-322, -180, -5, 3, 7, 10, 180, 360]; the function would return a queue with values [3, -5, 7, 10, -180, 180, -322, 360]. You should only interact with the Queue and the Stack through the operations given above.

10

This page intentionally left blank, use if needed but it will not be marked.

Question 8 [10 marks = 3 + 5 + 2]

(a) How can we address collisions in Hash Tables? List at least 3 different approaches, covered in the lectures, and explain how they differ from each other.

3

This page intentionally left blank, use if needed but it will not be marked.

(b) Consider the class `Hash` which has the instance variables `array`, `table_size`, and `count`, and the following methods: `__init__()`, `hash(the_key)` and `rehash()`.

Using Linear Probing, define a method `__setitem__(self, key, data)` which inserts the `data` into the Hash Table at the **position** calculated using the key. If a collision occurs, the method should resolve it using linear probing. If the Hash Table is full, rehash the Hash Table and proceed to insert as above.

(c) List one advantage and one disadvantage of Separate Chaining over Linear Probing? Explain your answer.

7

This page intentionally left blank, use if needed but it will not be marked.

Question 9 [12 marks = 2 + 2 + 3 + 3 + 2]

A Dequeue is a Queue that supports operations to add and serve items to and from the front and the rear of the Queue. This question is about implementing a Dequeue ADT based on an array. **All the method implementations should be circular to avoid wasting space in the underlying array. Use assertions to deal with potential errors.** The constructor and two additional methods are defined as follows:

```
class CircularDeQueue:
    def __init__(self, size):
        assert size > 0, "Size should be positive"
        self.the_array = size*[None]
        self.count = 0
        self.rear = 0
        self.front = 0

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

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

(a) Implement the method `def append_rear(self, new_item)`, which appends a `new_item` at the rear of the Queue.

(b) Implement the method `def serve_front(self)`, which removes from the Queue and returns the object at the front of the Queue.

4

This page intentionally left blank, use if needed but it will not be marked.

(c) Implement the method `def append_front(self)`, which appends a `new_item` at the front of the Queue.

(d) Implement the method `def serve_rear(self)`, which takes the object at the rear of the Queue, removing it from the queue and returning it.

(e) Implement the method `def print_items(self)`, which prints all the objects in the Queue from the front to the rear. Each element should be printed in one line.

8

This page intentionally left blank, use if needed but it will not be marked.

Question 10 [7 marks = 3 + 2 + 2]

Consider the ADT SortedList and the class defined below:

```
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
```

(a) Implement the class method `def _binary_search(self, item)`, which returns the index of `item` if it is in the list, or `-1` if the item is not in the list. Your implementation should have worst-case time complexity $O(\log n)$.

(b) Calling the method `_binary_search` defined in part *a*, implement `def index(self, item)`. This method should return the index of the first occurrence of `item` in the list, or raise a `ValueError` exception if `item` is not in the list.

5

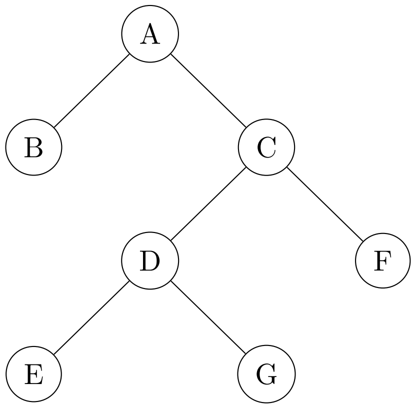
This page intentionally left blank, use if needed but it will not be marked.

(c) What is the best-case time complexity of the method `index(self, item)`, and when does it occur? Explain your answer.

Question 11 [6 marks = 3 + 1 + 1 +1]

This question is about Binary Trees and Binary Search Trees.

(a) State the outcomes of printing the elements of the Binary Tree below in pre-order, in-order and post-order.

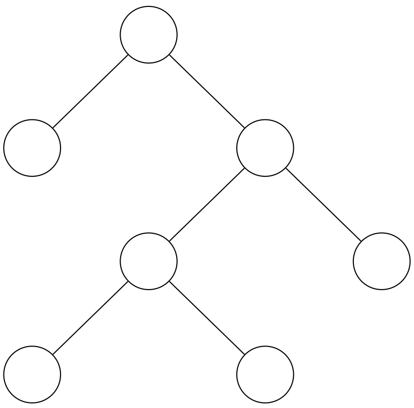


5

This page intentionally left blank, use if needed but it will not be marked.

(b) What are the best-case and worst-case time complexities in terms of the number of nodes, for an algorithm that prints the elements of a Binary Tree in pre-order. Explain your answer. No explanation means no marks.

(c) Fill in the nodes in the Binary Tree below with the elements from the list $[1, 5, 6, 7, 9, 10, 4]$ so that it is a valid Binary Search Tree.



(d) What is the worst-case time complexity for an algorithm that searchs for an item in a Binary Search Tree. Explain your answer. No explanation means no marks.

3

This page intentionally left blank, use if needed but it will not be marked.

Question 12 [6 marks = 4 + 2]

The next two questions are about recursion.

(a) Consider the following partial implementation of a Binary Tree.

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

    def __str__(self):
        return str(self.item)


class BinaryTree:
    def __init__(self):
        self.root = None

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

Using Python write a method `def height(self)` for the class `BinaryTree` that computes the height of a given tree using **recursion**.

(b) Quick-sort is a recursive sorting algorithm that relies on a partition method with linear time complexity. If you had a magic algorithm to do the partition in $O(1)$ time, what would be the best and worst case time complexity of Quick-sort? Explain your answer. No explanation means no marks.

6

This page intentionally left blank, use if needed but it will not be marked.

FIT1008 Supplementary Material - MIPS Reference Sheet

Table 1: **SPIM system calls**

Call code (`$v0`)	Service	Arguments	Returns	Notes
1	Print integer	`$a0` = value to print	-	value is signed
4	Print string	`$a0` = address of string to print	-	string must be terminated with `'\0'`
5	Input integer	-	`$v0` = entered integer	value is signed
8	Input string	`$a0` = address to store string at `$a1` = maximum number of chars	–	returns if `$a1-1` characters or Enter typed, the string is terminated with `'\0'`
9	Allocate memory	`$a0` = number of bytes	`$v0` = address of first byte	-
10	Exit	-	-	ends simulation

Table 2: **General-purpose registers**

Number	Name	Purpose
R00	`$zero`	provides constant zero
R01	`$at`	reserved for assembler
R02, R03	`$v0, $v1`	system call code, return value
R04–R07	`$a0--$a3`	system call and function arguments
R08–R15	`$t0--$t7`	temporary storage (caller-saved)
R16–R23	`$s0--$s7`	temporary storage (callee-saved)
R24, R25	`$t8, $t9`	temporary storage (caller-saved)
R26, R27	`$k0, $k1`	reserved for kernel code
R28	`$gp`	pointer to global area
R29	`$sp`	stack pointer
R30	`$fp`	frame pointer
R31	`$ra`	return address

Table 3: **Assembler directives**

.data	assemble into data segment
.text	assemble into text (code) segment
.byte b1[, b2, ...]	allocate byte(s), with initial value(s)
.half h1[, h2, ...]	allocate halfword(s), with initial value(s)
.word w1[, w2, ...]	allocate word(s) with initial value(s)
.space n	allocate n bytes of uninitialized, unaligned space
.align n	align the next item to a 2^n-byte boundary
.ascii "string"	allocate ASCII string, do not terminate
.asciiz "string"	allocate ASCII string, terminate with `'\0'`

Table 4: **Function calling convention**

On function call:

Caller:	**Callee:**
saves temporary registers on stack passes arguments on stack calls function using `jal fn_label`	saves `$ra` and `$fp` on stack copies `$sp` to `$fp` allocates local variables on stack

On function return:

Callee:	**Caller:**
sets `$v0` to return value clears local variables off stack restores saved `$fp` and `$ra` off stack returns to caller with `jr $ra`	clears arguments off stack restores temporary registers off stack uses return value in `$v0`

Table 5: **Instruction Set**

A partial instruction set is on the next page. The following conventions apply.

Instruction Format

Rsrc, Rsrc1, Rsrc2: source operand(s), - must be a register value(s)

Src2; source operand - may be an immediate value or a register value

Rdest: destination, must be a register

Imm: Immediate value, may be 32 or 16 bits

Imm16: Immediate 16-bit value

Addr: Address in the form: offset(Rsrc) ie. absolute address = Rsrc + offset

label: label of an instruction

$\star$: pseudoinstruction

Immediate Form -: no immediate form, or this is the immediate form

$\star$: immediate form synthesized as pseduoinstruction

Unsigned form (append 'u' to instruction name):

- : no unsigned form, or this is the unsigned form

Table 6: **MIPS instruction set**

Instruction format	Meaning	Operation	Immediate form	Unsigned form(u)
add Rdest, Rsrc1, Rsrc2	Add	Rdest = Rsrc1 + Rsrc2	addi	no overflow trap
sub Rdest, Rsrc1, Rsrc2	Subtract	Rdest = Rsrc1 - Rsrc2	$\star$	no overflow trap
mul Rdest, Rsrc1, Rsrc2 $\star$	Multiply	Rdest = Rsrc1 * Rsrc2	$\star$	unsigned operands
mulo Rdest, Rsrc1, Rsrc2 $\star$	Multiply (with 32-bit overflow)	Rdest = Rsrc1 * Rsrc2	$\star$	unsigned operands
mult Rsrc1, Rsrc2	Multiply (machine instruction)	Hi:Lo = Rsrc1 * Rsrc2	-	unsigned operands
div Rdest, Rsrc1, Rsrc2 $\star$	Divide	Rdest=Rsrc1/Rsrc2	$\star$	unsigned operands
div Rsrc1, Rsrc2	Divide (machine instruction)	Lo = Rsrc1/Rsrc2; Hi = Rsrc1 % Rsrc2	-	unsigned operands
rem Rdest, Rsrc1, Rsrc2 $\star$	Remainder	Rdest = Rsrc1 % Rsrc2	$\star$	unsigned operands
neg Rdest, Rsrc $\star$	Negate	Rdest = -Rsrc1	-	no overflow trap
and Rdest, Rsrc1, Rsrc2	Bitwise AND	Rdest = Rsrc1 & Rsrc2	andi	-
or Rdest, Rsrc1, Rsrc2	Bitwise OR	Rdest = Rsrc1 $\vert$ Rsrc2	ori	-
xor Rdest, Rsrc1, Rsrc2	Bitwise XOR	Rdest = Rsrc1 $\wedge$ Rsrc2	xori	-
nor Rdest, Rsrc1, Rsrc2	Bitwise NOR	Rdest = $\sim$(Rsrc1 $\vert$ Rsrc2)	$\star$	-
not Rdest, Rsrc $\star$	Bitwise NOT	Rdest = $\sim$(Rsrc)	−	-
sll Rdest, Rsrc1, Rsrc2	Shift Left Logical	Rdest = Rsrc1 << Rsrc2	-	-
srl Rdest, Rsrc1, Rsrc2	Shift Right Logical	Rdest = Rsrc1 >> Rsrc2 (MSB=0)	-	-
sra Rdest, Rsrc1, Rsrc2	Shift Right Arithmetic	Rdest = Rsrc1 >> Rsrc2 (MSB preserved)	-	-
move Rdest, Rsrc $\star$	Move	Rdest=Rsrc	-	-
mfhi Rdest	Move from Hi	Rdest = Hi	-	-
mflo Rdest	Move from Lo	Rdest = Lo	-	-
li Rdest, Imm $\star$	Load immediate	Rdest=Imm	-	-
lui Rdest, Imm16	Load upper immediate	Rdest=Imm16 << Imm	-	-
la Rdest, Addr(or label) $\star$	Load Address	Rdest=Addr (or Rdest=label)	-	-
lb Rdest, Addr (or label $\star$)	Load byte	Rdest = mem8[Addr]	-	zero-extends data
lh Rdest, Addr (or label $\star$)	Load halfword	Rdest = mem16[Addr]	-	zero-extends data
lw Rdest, Addr (or label $\star$)	Load word	Rdest = mem32[Addr]	-	-
sb Rsrc2, Addr (or label $\star$)	Store byte	mem8[Addr] = Rsrc2	-	-
sh Rsrc2, Addr (or label $\star$)	Store halfword	mem16[Addr] = Rsrc2	-	-
sw Rsrc2, Addr (or label $\star$)	Store word	mem32[Addr] = Rsrc2	-	-
beq Rsrc1, Rsrc2, label	Branch if equal	if (Rsrc1 == Rsrc2) PC = label	$\star$	-
bne Rsrc1, Rsrc2, label	Branch if not equal	if (Rsrc1 != Rsrc2) PC = label	$\star$	-
blt Rsrc1, Rsrc2, label $\star$	Branch if less than	if (Rsrc1 < Rsrc2) PC = label	$\star$	unsigned operands
ble Rsrc1, Rsrc2, label $\star$	Branch if less than or equal	if (Rsrc1 <= Rsrc2) PC = label	$\star$	unsigned operands
bgt Rsrc1, Rsrc2, label $\star$	Branch if greater than	if (Rsrc1 > Rsrc2) PC = label	$\star$	unsigned operands
bge Rsrc1, Rsrc2, label $\star$	Branch if greater than or equal	if (Rsrc1 >= Rsrc2) PC = label	$\star$	unsigned operands
slt Rdest, Rsrc1, Rsrc2	Set if less than	if (Rsrc1 < Rsrc2) Rdest=1 else Rdest=0	slti	unsigned operands
j label	Jump	PC = label	-	-
jal label	Jump and link	$ra = PC + 4; PC = label	-	-
jr Rsrc	Jump register	PC = Rsrc	-	-
jalr Rsrc	Jump and link register	$ra = PC + 4; PC = Rsrc	-	-
syscall	System call	depends on call code in `$v0`	-	-