University of KwaZulu-Natal
School of Engineering
Lecturer: Mr Raleigh Reddy
Internal Moderator: Prof J Tapamo

ENEL2DS: Data Structures
Date: 12 November 2025
Total marks: 100
Duration: 120 minutes

Main Examination

Instructions:

- All algorithms must be implemented in C.
- Mark allocations are indicated in brackets.
- The examination paper consists of 7 pages, please make sure you have them all.

Question 1 (10 marks, 1 marks each)

1.1. During a Shell sort, which statement is correct?
A) The interval should remain the same.
B) The interval should increase from 1.
C) The interval should decrease down to 1.
D) The interval should increase from 0.

1.2. Stacks are a LIFO structure which means...
A) it is easier to implement with an array than with a linked list.
B) stacks are abstract data types, defined by their operations.
C) the first item added to the stack will be the last one removed.
D) items are added to and removed from different ends of the stack.

1.3. Which statement is correct?
A) Linear search, like binary search, halves the search space with every iteration.
B) Linear search is the only valid searching algorithm if the array is unsorted.
C) Binary search is faster than linear search because it searches two elements at a time.
D) Binary search is faster than linear search because binary search scales linearly with the number of elements of the array.

1.4. In the context of recursion algorithms, which statement is correct?
A) Base cases are where the input to a recursion function is 0 or 1.
B) Recursion algorithms must have a base case.
C) Recursion algorithms only work on positive integers.
D) It is possible for a base case to also call the recursive function.

1.5. Inserting a node at the beginning of a linked list is different to inserting a node anywhere else in the linked list because...
A) the head node is the only node who's location is known.
B) there are more free memory locations at the beginning of the linked list, therefore malloc() runs quicker.
C) it is easier to delete nodes at the end of the linked list.
D) nodes are created dynamically, as and when they are needed, not at compile time.

1.6. Queues are a FIFO structure which means...
A) items are added to and removed from the same part of the queue.
B) queues are abstract data types, defined by their operations.
C) the last item added to the queue will be the last one removed.
D) it is easier to implement with an array than with a linked list.

1.7. Which statement is correct?

A) In both static and dynamic arrays, elements are accessed via indices.
B) Static arrays cannot change their size because they have an internal dynamic array.
C) The elements of static arrays are constant and set when the array is created.
D) When a dynamic array runs out of space, its elements are replaced with NULL.

1.8. Which statement is correct?

A) Algorithm A has complexity $O(n)$ and Algorithm B has complexity $O(n^2)$. The two algorithms are executed one after the other. The resulting time complexity is $O(n^2)$.

B) Algorithm A and Algorithm B have the same complexity. They will take the same amount of time to execute on the same system, given the same input.

C) Algorithm A and Algorithm B take exactly the same amount of time to execute on the same system when the same input is used. Therefore, if the input is doubled, they will take the same amount of time to execute.

D) Algorithm A has complexity $O(n^{1.5})$ and Algorithm B has complexity $O(n^2)$. Therefore, given the same input and executed on the same system, Algorithm A will execute faster.

1.9. Consider the following C code snippet:

```
struct node {
   int i:
   float j;
};
struct node *s[10];
```

Which of the following does the above define s to be:
A) An array, each element of which is a pointer to a structure of type node.
B) A structure of 2 fields, each field being a pointer to an array of 10 elements.
C) A structure of 3 fields: an integer, a float, and an array of 10 elements.
D) An array, each element of which is a structure of type node.

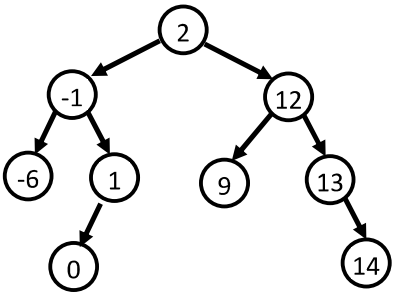
1.10. Consider the following C program.

```
#include < stdio.h >
int main( )
{
    int arr[4][5];
    int  i, j;
    for (i=0; i < 4; i++)
  {
        for (j=0; j < 5; j++)
        {
            arr[i][j] = 10 * i + j;
        }
    }
    printf("%d", *(arr[1]+9));
    return 0;
}
```

What will be the output of the above program?
A) 14
B) 20
C) 24
D) 30

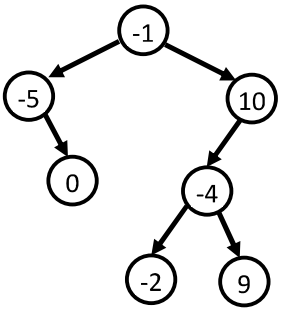
Question 2 (20 marks)

2.1. Consider the tree given below:



2.1.1. Show that the tree is an AVL tree. (2 marks)
2.1.2. Show the new tree, T1, when the node with value -6 is removed. (1 mark)
2.1.3. Show that the tree, T1, is no longer an AVL tree by identifying the type and location of the violation of the AVL property and thereafter specifying and performing the rotations needed to restore the AVL property. (10 marks)

2.2. Consider the tree below.



2.2.1. Give the order in which the nodes are visited if the tree is traversed in a in-order traversal. (4 marks)
2.2.2. Based on this traversal, is the above tree a Binary Search Tree? Provide a reason for your answer. (3 marks)

Question 3 (20 marks)

3.1. A company requires that usernames should start with an uppercase letter and consist only of letters and numbers. Write a function **`bool checkUsername(char s[], int SIZE)`** that takes a string **`s`** with length **`SIZE`** and returns **true** if it is a valid username or **false** if it does not. (10 marks)

You may find some of the following functions helpful.

int isalpha(int c);	Returns a true value if c is a letter and 0 otherwise.
int islower(int c);	Returns a true value if c is a lowercase letter and 0 otherwise.
int isupper(int c);	Returns a true value if c is an uppercase letter and 0 otherwise.
int isblank(int c);	Returns a true value if c is a blank character and 0 otherwise.
int isdigit(int c);	Returns a true value if c is a digit and 0 otherwise.
int isalnum(int c);	Returns a true value if c is a digit or letter and 0 otherwise.
int ispunct(int c);	Returns a true value if c is a character other than a space, digit or letter and 0 otherwise.
int tolower(int c);	If c is an uppercase letter, tolower returns c as a lowercase letter. Otherwise, tolower returns the argument unchanged.
int toupper(int c);	If c is a lowercase letter, toupper returns c as an uppercase letter. Otherwise, toupper returns the argument unchanged.

3.2 Complete the program below which is meant to be an iterative program to reverse a string **str** in linear time and using constant space. The use of the stack data structure is not allowed or any reverse library methods. (10 marks)

```
#include <stdio.h>
#include <string.h>

// Iterative function to reverse a given string.
void reverse(char *str) {

    // ANSWER TO Q3.2

}

int main()
{
    char str[100];
    printf("Enter a line of text: ");
    scanf("%[^\n]", str); // Read characters until a newline is encountered and store in str
    reverse(str);
    printf("Reverse of the entered string is %s", str);

    return 0;
}
```

Question 4 (20 marks)

The median-of-three quicksort involves selecting the median of the first, middle, and last elements of the current subarray being sorted as the pivot element.

This strategy aims to choose a pivot that is more likely to be close to the true median which would be the middle value if the subarray was sorted. This pivot leads to better average-case performance and mitigating the worst-case $O(n^2)$ performance of quicksort where a poorly chosen pivot consistently leads to highly unbalanced partitions.

The median-of-three quicksort can be implemented using the standard partition and core quicksort functions and merely defining a median-of-three function that returns the index of the median of the first, middle, and last elements of the subarray being sorted.

The median-of-three function is then called in the partition function to obtain the index of this median-of-three pivot and swapping the pivot with the last element so that the standard partition logic can be used where the pivot is the last element.

The quicksort function is defined as normal calling the partition function to obtain its index and then separately sorting the elements before and after it.

Complete the program given on the next page by providing the implementations of:

4.1. **medianOfThree(int arr[], int first, int last)**,
4.2. **partition(int arr[], int low, int high)** and
4.3. **quickSort(int arr[], int low, int high)**.

The mark breakdown for the subquestions are furthermore provided in their code commented sections.

```c
#include <stdio.h>

// Helper function to swap two elements
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;

}

// Function to find the median of three elements and return the index of this value
int medianOfThree(int arr[], int first, int last) {

    // ANSWER TO Q4.1. (6 marks)

}

// Function to partition the array around a pivot
int partition(int arr[], int low, int high) {

    // ANSWER TO Q4.2. (10 marks)

}

// Function that implements QuickSort
void quickSort(int arr[], int low, int high) {

    // ANSWER TO Q4.3. (4 marks)

}

// Helper function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }

    printf("\n");
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5, 2, 4, 3, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    quickSort(arr, 0, n - 1);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Question 5 (30 marks)

Timsort is a hybrid stable sorting algorithm, derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data.

It works by finding naturally occurring "runs" (already sorted subarrays) within the data and then merging them.

If a run's length is less than a specified minimum (often 32 or 64), elements are taken from the subsequent part of the array to extend the run to the minimum length.

The run segment that has been extended to minimum size is then sorted using the insertion sort, which is efficient for small arrays.

Timsort can be implemented using the standard merge and core insertion sort functions and then calling the merge function to merge the sorted runs or calling the insertion sort function to sort runs that were extended to minimum length.

Values for the left and right indices are passed in along with the array as paramenters to the insertion sort function instead of just the array and the value n to represent its size. The left and right indices are then used to indicate the location of the sub-array within the array representing the run that is meant to be sorted.

You are to complete the program below by providing the implementations of:

5.1. **insertionSort(int arr[], int left, int right)** and
5.2. **merge(int arr[], int left, int mid, int right)**

The mark breakdown for the subquestions are furthermore provided in their code commented sections.

A reminder to use the values of the left and right paramenters to restrict the insertion sort function to the sub-array that needs be sorted instead of the whole array and also when shifting elements in the sub-array sorted part.

Additionally, the merge function would be just as its standard implementation.

Lastly, the implemention of the function **timsort(int arr[], int n)** has already been provided as shown on the next page.

```
#include <stdio.h>
#include <stdlib.h>

#define MIN_RUN 32 // Minimum run length for Timsort

// Function for Insertion Sort
void insertionSort(int arr[], int left, int right) {

    // ANSWER TO Q5.1. (10 marks)

}

// Function to merge two sorted runs
void merge(int arr[], int left, int mid, int right) {

    // ANSWER TO Q5.2. (20 marks)

}
```

```
// Timsort function implemented
void timsort(int arr[], int n) {

    // Sort individual runs using insertion sort
    for (int i = 0; i < n; i += MIN_RUN) {
        insertionSort(arr, i, ((i + MIN_RUN - 1) < (n - 1)) ?
                              (i + MIN_RUN - 1) : (n - 1));
    }

    // Merge runs
    for (int size = MIN_RUN; size < n; size = 2 * size) {
        for (int left = 0; left < n; left += 2 * size) {
            int mid = left + size - 1;
            int right = ((left + 2 * size - 1) < (n - 1)) ?
                         (left + 2 * size - 1) : (n - 1);

            if (mid < right) { // Ensure there are two runs to merge
                merge(arr, left, mid, right);
            }
        }
    }
}

int main() {
    int arr[] = {5, 2, 8, 1, 9, 4, 7, 3, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    timsort(arr, n);

    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

END OF PAPER