Birla Institute of Technology & Science, Pilani
Work Integrated Learning Programmes Division
First Semester 2021-2022

Comprehensive Examination (EC-3 Make-Up)

Course No.	: SEZG519 / SSZG519
Course Title	: Data Structures and Algorithms Design
Nature of Exam	: Open Book
Weightage	: 50%
Duration	: 2 Hours
Date of Exam	: 27/11/2021 (FN)

No. of Pages = 7
No. of Questions = 7

Note to Students:

1. Please follow all the *Instructions to Candidates* given on the cover page of the answer book.
2. All parts of a question should be answered consecutively. Each answer should start from a fresh page.
3. Assumptions made if any, should be stated clearly at the beginning of your answer.

Q.1 Set. (A) Marks: 6

Arrange the following 10 functions (on n) in descending order of their growth rates:

$6n^4+n^3+1$, $n^{\log(13) / \log(n)}$, $1024^{\log(8n)}$, $(\pi n)^{3.55}$, $1729n^{1/2}\log(n)$,
$7n(\log(n))^5$, $4\log(\log(n))$, $44^{n-4}n^6$, $13n^3+5$, $69\log(n^2)+11$

Q.1 Set. (B) Marks: 6

Arrange the following 10 functions (on n) in descending order of their growth rates:

$5n^6+n^5+2$, $3n^{\log(17) / \log(n)}$, $256^{\log(4\pi n)}$, $(3n)^{5.25}$, $2002n^{1/3}\log(n)$,
$16n(\log(n))^4$, $19\log(\log(n))$, $33^{n+3}n^5$, $7n^5+3$, $8\log(n^5)+22$

Q.1 Set. (C) Marks: 6

Arrange the following 10 functions (on n) in descending order of their growth rates:

$3n^7+n^2+1$, $n^{\log(19) / \log(n)}$, $2096^{\log(2n)}$, $(2n)^{6.75}$, $505n^{1/4}\log(n)$,
$5n(\log(n))^3$, $\pi\log(\log(n))$, $77^{n+7}n^9$, $4n^6+7$, $17\log(n^{51})+9$

Q.2 Set. (A) Marks: 7+1

You have implemented a Queue ADT by using a circular integer array A of size 4. Initially, the queue contains no elements, and therefore you have initialized the two end-pointers as *front=0* and *rear=-1* (as per the convention shown in class). Now, you proceed to perform the following sequence of operations on this array-based queue:

enqueue(3), enqueue(5), enqueue(7), dequeue(), dequeue(), dequeue(), enqueue(1), enqueue(4), enqueue(6), enqueue(8), dequeue(), dequeue(), enqueue(2), dequeue()

(a) Draw a snapshot of the array A after each of the above operations, and mention clearly the values of *front* and *rear* corresponding to each such snapshot.

(b) Does the queue become empty or full at any point (except the beginning) during this sequence of operations? If yes, mention the exact point(s) in the sequence.

Q.2 Set. (B) Marks: 7+1

You have implemented a Queue ADT by using a circular integer array A of size 4. Initially, the queue contains no elements, and therefore you have initialized the two end-pointers as *front=0* and *rear=-1* (as per the convention shown in class). Now, you proceed to perform the following sequence of operations on this array-based queue:

enqueue(4), enqueue(3), enqueue(6), dequeue(), dequeue(), dequeue(), enqueue(1), enqueue(7), enqueue(8), enqueue(5), dequeue(), dequeue(), enqueue(2), dequeue()

(a) Draw a snapshot of the array A after each of the above operations, and mention clearly the values of *front* and *rear* corresponding to each such snapshot.

(b) Does the queue become empty or full at any point (except the beginning) during this sequence of operations? If yes, mention the exact point(s) in the sequence.

Q.2 Set. (C) Marks: 7+1

You have implemented a Queue ADT by using a circular integer array A of size 4. Initially, the queue contains no elements, and therefore you have initialized the two end-pointers as *front=0* and *rear=-1* (as per the convention shown in class). Now, you proceed to perform the following sequence of operations on this array-based queue:

enqueue(8), enqueue(6), enqueue(4), dequeue(), dequeue(), dequeue(), enqueue(2), enqueue(7), enqueue(5), enqueue(3), dequeue(), dequeue(), enqueue(1), dequeue()

(a) Draw a snapshot of the array A after each of the above operations, and mention clearly the values of *front* and *rear* corresponding to each such snapshot.

(b) Does the queue become empty or full at any point (except the beginning) during this sequence of operations? If yes, mention the exact point(s) in the sequence.

Q.3 Set. (A) Marks: 3+3+3

Consider a hashing scheme that maps elements to a hash table T of size 17 by using the hash function *h(k) = (k+5) mod 17*, where *k* represents a key. T was initially empty. Now, 10 records are inserted sequentially into T, with the corresponding keys being as follows: **43, 36, 92, 87, 11, 4, 88, 96, 181, 77**

(a) If the collisions in T are resolved by means of open addressing and separate chaining, then draw and show how T will look after the insertion of all the 10 records.

(b) If the collisions in T are resolved by means of closed addressing and linear probing, then draw and show how T will look after the insertion of all the 10 records.

(c) If the collisions in T are resolved by means of closed addressing and quadratic probing, then draw and show how T will look after the insertion of all the 10 records.

Q.3 Set. (B) Marks: 3+3+3

Consider a hashing scheme that maps elements to a hash table T of size 17 by using the hash function *h(k) = (k+5) mod 17*, where *k* represents a key. T was initially empty. Now, 10 records are inserted sequentially into T, with the corresponding keys being as follows: **45, 38, 94, 89, 13, 6, 56, 98, 166, 79**

(a) If the collisions in T are resolved by means of open addressing and separate chaining, then draw and show how T will look after the insertion of all the 10 records.

(b) If the collisions in T are resolved by means of closed addressing and linear probing, then draw and show how T will look after the insertion of all the 10 records.

(c) If the collisions in T are resolved by means of closed addressing and quadratic probing, then draw and show how T will look after the insertion of all the 10 records.

Q.3 Set. (C) Marks: 3+3+3

Consider a hashing scheme that maps elements to a hash table T of size 17 by using the hash function *h(k) = (k+5) mod 17*, where *k* represents a key. T was initially empty. Now, 10 records are inserted sequentially into T, with the corresponding keys being as follows: **41, 34, 90, 85, 9, 2, 69, 94, 196, 58**

(a) If the collisions in T are resolved by means of open addressing and separate chaining, then draw and show how T will look after the insertion of all the 10 records.

(b) If the collisions in T are resolved by means of closed addressing and linear probing, then draw and show how T will look after the insertion of all the 10 records.

(c) If the collisions in T are resolved by means of closed addressing and quadratic probing, then draw and show how T will look after the insertion of all the 10 records.

Q.4 Set. (A) Marks: 5

Consider the following array: **A = {6, 3, 8, -1, 13, 7, 5}**

Trace all the execution steps of the Insertion Sort algorithm applied on the array A, and accordingly fill up the table shown below.

Pass No.	Intermediate Output	No. of Comparisons
0	{ 6, 3, 8, -1, 13, 7, 5 }	-
1	{ 3, 6, 8, -1, 13, 7, 5 }	1
⋮	⋮	⋮
TOTAL	{ -1, 3, 5, 6, 7, 8, 13 }	...

[NB – You need to follow the exact pseudocode for Insertion Sort discussed in class, where the element originally at index *i*, i.e. the element *A[i]* in the original array, is moved and inserted into its correct position during the *i*th pass of the algorithm.]

Q.4 Set. (B) Marks: 5

Consider the following array: **A = {5, 2, 7, -2, 11, 6, 4}**

Trace all the execution steps of the Insertion Sort algorithm applied on the array A, and accordingly fill up the table shown below.

Pass No.	Intermediate Output	No. of Comparisons
0	{ 5, 2, 7, -2, 11, 6, 4 }	-
1	{ 2, 5, 7, -2, 11, 6, 4 }	1
⋮	⋮	⋮
TOTAL	{ -2, 2, 4, 5, 6, 7, 11 }	...

[NB – You need to follow the exact pseudocode for Insertion Sort discussed in class, where the element originally at index *i*, i.e. the element *A[i]* in the original array, is moved and inserted into its correct position during the *i*th pass of the algorithm.]

Q.4 Set. (C) Marks: 5

Consider the following array: **A = {7, 4, 9, -3, 15, 8, 6}**

Trace all the execution steps of the Insertion Sort algorithm applied on the array A, and accordingly fill up the table shown below.

Pass No.	**Intermediate Output**	**No. of Comparisons**
0	{ 7, 4, 9, -3, 15, 8, 6 }	-
1	{ 4, 7, 9, -3, 15, 8, 6 }	1
⋮	⋮	⋮
TOTAL	{ -2, 2, 4, 5, 6, 7, 11 }	...

[NB – You need to follow the exact pseudocode for Insertion Sort discussed in class, where the element originally at index *i*, i.e. the element *A[i]* in the original array, is moved and inserted into its correct position during the *i*th pass of the algorithm.]

Q.5 Set. (A) Marks: 5+1

Insert the following integers, in the given sequence, into an initially empty BST:
57, 25, 33, 19, 69, 77, 82, 64, 13, 91

Draw and show how the BST looks after the insertion of each of the above integers.

After inserting all these integers, what is the height of the BST at the end?

Q.5 Set. (B) Marks: 5+1

Insert the following integers, in the given sequence, into an initially empty BST:
55, 23, 31, 17, 69, 72, 84, 66, 10, 98

Draw and show how the BST looks after the insertion of each of the above integers.

After inserting all these integers, what is the height of the BST at the end?

Q.5 Set. (C) Marks: 5+1

Insert the following integers, in the given sequence, into an initially empty BST:
59, 27, 35, 21, 66, 78, 80, 62, 14, 93

Draw and show how the BST looks after the insertion of each of the above integers.

After inserting all these integers, what is the height of the BST at the end?

Q.6 Set. (A) Marks: 7+3

Consider the following infix expression: **8 + 2 * 5 ^ (7 - 4 + 1) + 9 / 3**

(a) Convert the above infix expression into its equivalent postfix expression by using a stack, and trace the steps of the conversion algorithm by filling in the table shown below:

Step #	Remaining Input	Stack Trace	Current (Partial) Output
0	8 + 2 * 5 ^ (7 – 4 + 1) + 9 / 3 $	<empty>	<empty>
1	+ 2 * 5 ^ (7 – 4 + 1) + 9 / 3 $	<empty>	8
2	2 * 5 ^ (7 – 4 + 1) + 9 / 3 $	+	8
⋮	⋮	⋮	⋮
Final	<empty>	<empty>	...

[NB – For the stack trace, you should write out from left to right all the elements of the stack as they are placed in sequence from the bottom to the top of the stack. For example, if the stack currently contains the 3 elements x, y and z, with x being at the bottom and z being at the top, then you should write out "x, y, z" in the 'Stack Trace' column.]

(b) What is the result on evaluating the postfix expression obtained above as output? Clearly show all the steps in the evaluation algorithm, along with the stack trace.

Q.6 Set. (B) Marks: 7+3

Consider the following infix expression: **7 + 1 * 4 ^ (6 - 3 + 9) + 8 / 2**

(a) Convert the above infix expression into its equivalent postfix expression by using a stack, and trace the steps of the conversion algorithm by filling in the table shown below:

Step #	Remaining Input	Stack Trace	Current (Partial) Output
0	7 + 1 * 4 ^ (6 – 3 + 9) + 8 / 2 $	<empty>	<empty>
1	+ 1 * 4 ^ (6 – 3 + 9) + 8 / 2 $	<empty>	7
2	1 * 4 ^ (6 – 3 + 9) + 8 / 2 $	+	7
⋮	⋮	⋮	⋮
Final	<empty>	<empty>	...

[NB – For the stack trace, you should write out from left to right all the elements of the stack as they are placed in sequence from the bottom to the top of the stack. For example, if the stack currently contains the 3 elements x, y and z, with x being at the bottom and z being at the top, then you should write out "x, y, z" in the 'Stack Trace' column.]

(b) What is the result on evaluating the postfix expression obtained above as output? Clearly show all the steps in the evaluation algorithm, along with the stack trace.

Q.6 Set. (C) Marks: 7+3

Consider the following infix expression: **6 + 9 * 3 ^ (5 - 2 + 8) + 7 / 1**

(a) Convert the above infix expression into its equivalent postfix expression by using a stack, and trace the steps of the conversion algorithm by filling in the table shown below:

Step #	Remaining Input	Stack Trace	Current (Partial) Output
0	6 + 9 * 3 ^ (5 – 2 + 8) + 7 / 1 $	<empty>	<empty>
1	+ 9 * 3 ^ (5 – 2 + 8) + 7 / 1 $	<empty>	6
2	9 * 3 ^ (5 – 2 + 8) + 7 / 1 $	+	6
⋮	⋮	⋮	⋮
Final	<empty>	<empty>	...

[NB – For the stack trace, you should write out from left to right all the elements of the stack as they are placed in sequence from the bottom to the top of the stack. For example, if the stack currently contains the 3 elements x, y and z, with x being at the bottom and z being at the top, then you should write out "x, y, z" in the 'Stack Trace' column.]

(b) What is the result on evaluating the postfix expression obtained above as output? Clearly show all the steps in the evaluation algorithm, along with the stack trace.

Q.7 Set. (A) Marks: 3+3

In a style similar to the pseudocodes for the Linked List methods done in class, write down a correct implementation (in pseudocode) for the function `modifyDLL(p,k)`, where p is a pointer to a doubly-linked list node containing integer data, and k is a positive integer. The function `modifyDLL(p,k)` is supposed to achieve the following:

(i) Delete the node which lies *k* positions forward (i.e. to the right) from the pointer *p*, and return the data stored in that node, if such a node exists; otherwise, return -1.

(ii) Move the node pointed to by *p* backward (i.e. to the left) by *k* positions, if there exists at least *k* nodes between the *head* and the position *p*; otherwise, simply move this node to the *head* of the list. Note that you are supposed to actually move the node by updating the pointers appropriately, and not simply swap the data between existing nodes.

For example, if *p* points to the 4th node in the list [0, 1, 3, 5, 7, 9], and if *k=2*, then the function `modifyDLL(p,k)` should return the value 9, and the modified list at the end of its execution should be [0, 5, 1, 3, 7].

[NB – While writing out the pseudocode for the function, you need to correctly specify the full function signature, including the return type and the parameter types. You should assume the standard structure for the nodes of a linear doubly-linked list, as discussed in class. You need to write a robust implementation, and do all possible boundary condition checks, in order to prevent any accidental access operations on a NULL pointer.]

Q.7 Set. (B) Marks: 3+3

In a style similar to the pseudocodes for the Linked List methods done in class, write down a correct implementation (in pseudocode) for the function `modifyDLL(p,k)`, where p is a pointer to a doubly-linked list node containing integer data, and k is a positive integer. The function `modifyDLL(p,k)` is supposed to achieve the following:

(i) Delete the node which lies *k* positions backward (i.e. to the left) from the pointer *p*, and return the data stored in that node, if such a node exists; otherwise, return -1.

(ii) Move the node pointed to by *p* forward (i.e. to the right) by *k* positions, if there exists at least *k* nodes between the *tail* and the position *p*; otherwise, simply move this node to the *tail* of the list. Note that you are supposed to actually move the node by updating the pointers appropriately, and not simply swap the data between existing nodes.

For example, if *p* points to the 3rd node in the list [0, 1, 3, 5, 7, 9], and if *k=2*, then the function `modifyDLL(p,k)` should return the value 0, and the modified list at the end of its execution should be [1, 5, 7, 3, 9].

[NB – While writing out the pseudocode for the function, you need to correctly specify the full function signature, including the return type and the parameter types. You should assume the standard structure for the nodes of a linear doubly-linked list, as discussed in class. You need to write a robust implementation, and do all possible boundary condition checks, in order to prevent any accidental access operations on a NULL pointer.]

Q.7 Set. (C) Marks: 3+3

In a style similar to the pseudocodes for the Linked List methods done in class, write down a correct implementation (in pseudocode) for the function `modifyDLL(p,k)`, where p is a pointer to a doubly-linked list node containing integer data, and k is a positive integer. The function `modifyDLL(p,k)` is supposed to achieve the following:

(i) Delete the node which lies *k+1* positions forward (i.e. to the right) from the pointer *p*, and return the data stored in that node, if such a node exists; otherwise, return -1.

(ii) Move the node pointed to by *p* backward (i.e. to the left) by *k* positions, if there exists at least *k* nodes between the *head* and the position *p*; otherwise, simply move this node to the *head* of the list. Note that you are supposed to actually move the node by updating the pointers appropriately, and not simply swap the data between existing nodes.

For example, if *p* points to the 4th node in the list [0, 1, 3, 5, 7, 8, 9], and if *k=2*, then the function `modifyDLL(p,k)` should return the value 9, and the modified list at the end of its execution should be [0, 5, 1, 3, 7, 8].

[NB – While writing out the pseudocode for the function, you need to correctly specify the full function signature, including the return type and the parameter types. You should assume the standard structure for the nodes of a linear doubly-linked list, as discussed in class. You need to write a robust implementation, and do all possible boundary condition checks, in order to prevent any accidental access operations on a NULL pointer.]