Data Structures (CS2001)	
Date: 22nd Sep 2025	
Course Instructor(s)	
Dr. Jawwad, Dr. Anam, Dr. Farrukh, Mr. Basit, Mr. Farooq, Mr. Nouman, Ms. Sania, Ms. Fizza, Ms. Mubashara	
Sessional-I Exam	
Total Time (Hrs):	1
Total Marks:	15
Total Questions:	03

Do not write below this line

Attempt all the questions.

____________ Roll No	____________ Section	________________________ Student Signature

CLO # 1: Use & explain concepts related to basic and advanced data structures and describe their usage in terms of common algorithmic operations

Q1: Assume you are designing a class StudentRecord that stores a student's grades in a dynamically allocated array. Later, you observe the following situations:[Marks 1+1+1]

- Sometimes two StudentRecord objects are copied.
- Sometimes one StudentRecord object is assigned to another.

A. Explain why the default copy constructor and default assignment operator will fail in this scenario.
B. Write only the copy constructor for this class.
C. In one line, explain what problem may occur if you do not implement the assignment operator.

A. Default copy and assignment perform shallow copy both objects share the same pointer leads to double deletion or corrupted data.
B.

```
StudentRecord(const StudentRecord&amp; other) {
size = other.size;
grades = new int[size];
for (int i = 0; i &lt; size; i++)
grades[i] = other.grades[i];
}
```

C.

If the assignment operator isn't implemented assigning one object to another would again cause shallow copy and runtime crash when destructors run.

CLO # 1: Use & explain concepts related to basic and advanced data structures and describe their usage in terms of common algorithmic operations

Q2: Each node of a linked list holds an integer value. Implement a function that takes an integer x as input and splits the linked list into two sublists: [Marks 6]

- The first sublist contains all nodes with values less than x.
- The second sublist contains all nodes with values greater than or equal to x.

Note: Both sublists should preserve the original relative order of the nodes.

```
void splitList(Node* head, int x, Node*& lessHead, Node*& greaterEqualHead) {
    Node* lessTail = NULL;
    Node* greaterEqualTail = NULL;

    while (head) {
        if (head->data < x) {
            if (!lessHead) {
                lessHead = lessTail = head;
            } else {
                lessTail->next = head;
                lessTail = head;
            }
        } else {
            if (!greaterEqualHead) {
                greaterEqualHead = greaterEqualTail = head;
            } else {
                greaterEqualTail->next = head;
                greaterEqualTail = head;
            }
        }
        head = head->next;
    }

    // End both lists properly
    if (lessTail) lessTail->next = NULL;
    if (greaterEqualTail) greaterEqualTail->next = NULL;
}

int main() {
    Node* head = NULL;
```

```
   append(head, 10);
   append(head, 4);
   append(head, 3);
   append(head, 2);
   append(head, 5);
   append(head, 12);

   int x = 5;

   Node* less = NULL;
   Node* greaterEqual = NULL;

   splitList(head, x, less, greaterEqual);

   cout << "Less than " << x << ": ";
   printList(less);

   cout << "Greater than or equal to " << x << ": ";
   printList(greaterEqual);

   return 0;}
```

CLO # 3: Compare different data structures in terms of their relative efficiency and design effective solutions and algorithms that make use of them.
Q3: Imagine you are building a compiler for a programming language. One of the tasks in compilation is symbol table management. The compiler must keep track of variable names (symbols) in sorted order so they can be found quickly later. Instead of sorting the whole symbol table every time a new variable is declared, the compiler can use insertion sort logic: The existing table of symbols is already sorted. Each time a new variable name is declared, you insert it into the right place among the already sorted symbols. To optimize memory consumption, the underlying data structure used will be linked list. [Marks 3+3]

1. Write the recursive algorithm for the above problem of inserting a new node so that the list remains sorted.
2. Compare and contrast the recursive version with the iterative version in terms of performance and practicality by explaining their pros and cons.

SOLUTION:
Part 1:

```
InsertSorted_Recursive(head, newNode)
   if head = NULL OR newNode.symbol < head.symbol then
      newNode.next !   head
      return newNode
   else
      head.next !   InsertSorted_Recursive(head.next, newNode)
      return head
   end if
End Algorithm
```

Note:
Up to the students if they define the structure of the node. It just needs two parts: data,

pointer. Symbol could be any datatype of student's choosing.

Part 2:
Recursive
Pros:
Compact code, easier to understand and read
Naturally matches the recursive definition of the list.
Some compilers automatically optimize so readability and efficiency are both preserved.

Cons:
Extra function call overhead on every recursive step.
Stack depth grows with list size, and may cause stack overflow if the symbol table is very large.
Less memory-efficient due to recursion overhead.

Iterative
Pros:
More efficient in practice (no recursive call overhead).
Can handle very large symbol tables without risk of stack overflow.
Straightforward to integrate in compilers, where performance and memory are critical.

Cons:
Slightly more lenghty code.
Does not always follow the straightforward logic of the problem definition.

——— The End ———