SOLUTION QUIZ 01 PAPER A

MCQs (5 marks)

1. C) To hide implementation details and expose only the necessary operations.
2. B) Heap allocation
3. B) Memory is allocated dynamically as nodes are added.
4. B) It makes reversing the list more efficient.
5. C) Adding 1 to a pointer advances it to the next memory location of its type.

Short Q/A (5 marks, 2.5 each)

Question 01

Answer: A doubly linked list uses more memory per node because each node contains two pointers (next and prev), whereas a singly linked list node contains only one pointer (next).

Question 02

Steps to Insert a Node at Position n in a Doubly Linked List:

1. **Create the New Node:**
   - Allocate memory for the new node and initialize it with the desired data.
   - Set the prev and next pointers of the new node to NULL.
2. **Handle Edge Case - Insert at the Head (Position 0):**
   - If n == 0, you are inserting at the head of the list.
   - Set the next pointer of the new node to the current head.
   - If the list is not empty, update the prev pointer of the old head to point to the new node.
   - Update the head pointer to point to the new node.
3. **Traverse to the (n-1)th Node:**
   - Use a temporary pointer to traverse the list to the (n-1)th node (the node before the insertion point).
   - If the position n is beyond the current length of the list, terminate the operation or handle it as an out-of-bounds insertion.
4. **Insert Between Nodes:**
   - Set the next pointer of the new node to point to the current nth node (the node that was previously in position n).

- Set the prev pointer of the new node to point to the (n-1)th node (where the traversal stopped).
- Update the next pointer of the (n-1)th node to point to the new node.
- If the new node is not inserted at the end of the list, update the prev pointer of the original nth node to point to the new node.

CODE (5 marks, 2.5 each)

Question 01

```
// Function to insert a node at any position

void insertNodeAtAny(int data, int position) {

if (position == 0) {

Node* newNode = createNode(data);

if (head == nullptr) {

head = newNode;

} else {

newNode->next = head;

head->prev = newNode;

head = newNode;

}


return;

}


Node* newNode = createNode(data);

Node* temp = head;
```

```
for (int i = 0; i < position - 1 && temp != nullptr; ++i) {

temp = temp->next;

}


if (temp == nullptr) {

cout << "Position out of bounds" << endl;

delete newNode;

return;

}


newNode->next = temp->next;

if (temp->next != nullptr) {

temp->next->prev = newNode;

}

temp->next = newNode;

newNode->prev = temp;


}
```

Question 02

Error: In the case where the node at position 0 is deleted, the code incorrectly deletes the head pointer instead of the node being removed. This leads to freeing the wrong node and potential memory leaks.

Correction: The deletion should occur for the node that is stored in temp, not head.

```
void deleteAtPosition(Node*& head, int position) {
    if (head == NULL) return; // Check if the list is empty

    Node* temp = head;

    if (position == 0) {
        head = head->next;  // Update head
        delete temp;        // Free the original head node
        return;
    }

    for (int i = 0; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;  // Traverse to the (n-1)th node
    }

    if (temp == NULL || temp->next == NULL) return;  // Invalid position

    Node* toDelete = temp->next;       // Node to be deleted
    temp->next = temp->next->next;     // Bypass the deleted node
    delete toDelete;                   // Free memory
}
```