**Question 1 (Single Choice)**

**What is the time complexity for inserting a node at the beginning of a singly linked list?**

A) O(n)  
B) O(1)  
C) O(log n)  
D) O(n log n)

**Answer:** B) O(1)

**Explanation:**

* **A) O(n)** Incorrect. This would imply that we have to traverse the list to insert the node, which is not the case when inserting at the beginning.
* **B) O(1)** Correct. When inserting a node at the beginning of a singly linked list, we only need to update the head pointer and make the new node point to the previous head. This operation does not depend on the size of the list, so it takes constant time.
* **C)O(logn)**: Incorrect. This time complexity is typical of operations in balanced trees, not linked lists.
* **D)O(nlogn)**: Incorrect. This complexity is irrelevant in the context of linked list operations.

**Question 2 (Single Choice)**

**Which of the following is a valid scenario for choosing a linked list over an array?**

A) When you need fast random access to elements  
B) When the size of the data is fixed and known in advance  
C) When frequent insertions and deletions at arbitrary positions are required  
D) When you need to store data of the same type and size

**Answer:** C) When frequent insertions and deletions at arbitrary positions are required

**Explanation:**

* **A) When you need fast random access to elements**: Incorrect. Linked lists do not support random access efficiently. For accessing an element at a specific index, you have to traverse the list from the beginning, leading to a time complexity of O(n)
* **B) When the size of the data is fixed and known in advance**: Incorrect. Arrays are better suited for this, as they allow constant-time access and a fixed-size memory block.
* **C) When frequent insertions and deletions at arbitrary positions are required**: Correct. Linked lists are ideal when frequent insertions and deletions are necessary because these operations can be done in O(1) time if you have a pointer to the node. There is no need to shift elements like in arrays.
* **D) When you need to store data of the same type and size**: Incorrect. Both arrays and linked lists can store data of the same type and size, but this reason alone is not why a linked list would be chosen over an array.

**Question 3 (Multiple Choice)**

**Which of the following statements about a singly linked list are true? (Choose 2 options)**

A) Each node contains two pointers.  
B) The last node’s next pointer is set to NULL.  
C) Linked lists allow random access to elements.  
D) Traversing a linked list takes O(n) time.

**Answer:** B) The last node’s next pointer is set to NULL.  
D) Traversing a linked list takes O(n) time.

**Explanation:**

* **A) Each node contains two pointers**: Incorrect. In a **singly** linked list, each node contains only one pointer, which points to the next node. A **doubly** linked list contains two pointers (one for the previous and one for the next node).
* **B) The last node’s next pointer is set to NULL**: Correct. In a singly linked list, the last node does not point to any other node, so its next pointer is set to NULL to indicate the end of the list.
* **C) Linked lists allow random access to elements**: Incorrect. Linked lists do not support random access. You must traverse the list node by node to access an element.
* **D) Traversing a linked list takes O(n) time**: Correct. To visit all elements in a singly linked list, you must start from the head and move to each subsequent node, leading to a linear time complexity of O(n).

(Medium Level)

**Question 1 (Single Choice)**

**What will happen if you try to access the next pointer of the last node in a singly linked list?**

A) The program will crash.  
B) You will get a segmentation fault.  
C) It will return NULL.  
D) It will print garbage values.

**Answer:** C) It will return NULL.

**Explanation:**

* **A) The program will crash**: Incorrect. Accessing the next pointer of the last node does not necessarily cause a crash as long as you properly check for NULL.
* **B) You will get a segmentation fault**: Incorrect. A segmentation fault occurs if you try to dereference an invalid pointer, but simply accessing a valid node’s next pointer when it is NULL will not cause this error.
* **C) It will return NULL**: Correct. The last node’s next pointer is set to NULL, so if you try to access it, the program will return NULL, indicating the end of the list.
* **D) It will print garbage values**: Incorrect. If the next pointer is properly set to NULL, you will not see garbage values. Garbage values occur when the memory is uninitialized or if you access invalid memory.

**Question 2 (Single Choice)**

**You have a circular linked list where the last node points to the head. What happens if you try to traverse the list using the same logic as for a singly linked list (i.e., stop when the next pointer is NULL)?**

A) The list will be traversed successfully.  
B) The program will enter an infinite loop.  
C) The program will crash.  
D) The traversal will stop at the last node.

**Answer:** B) The program will enter an infinite loop.

**Explanation:**

* **A) The list will be traversed successfully**: Incorrect. In a circular linked list, the last node points to the head, so the traversal logic that relies on the next pointer being NULL will not work.
* **B) The program will enter an infinite loop**: Correct. Since the next pointer of the last node points to the head, and never becomes NULL, the loop will continue indefinitely.
* **C) The program will crash**: Incorrect. The program won’t crash unless there’s some other error (e.g., a stack overflow after too many iterations), but the issue here is an infinite loop.
* **D) The traversal will stop at the last node**: Incorrect. The traversal won’t stop because there is no NULL in a circular linked list.

**Question 3 (Multiple Choice)**

**In a doubly linked list, what is the primary advantage of having a prev pointer in each node? (Choose 2 options)**

A) It allows traversal of the list in both directions.  
B) It improves the time complexity of searching for an element to O(logn).  
C) It simplifies the deletion of a node when a pointer to that node is given.  
D) It reduces the memory overhead by eliminating the need for a next pointer.

**Answer:** A) It allows traversal of the list in both directions.  
C) It simplifies the deletion of a node when a pointer to that node is given.

**Explanation:**

* **A) It allows traversal of the list in both directions**: Correct. The prev pointer enables traversing the list backward, which is not possible in a singly linked list.
* **B) It improves the time complexity of searching for an element to (logn)**: Incorrect. The time complexity for searching remains O(n)O(n)O(n) in both singly and doubly linked lists. The prev pointer does not change the search complexity.
* **C) It simplifies the deletion of a node when a pointer to that node is given**: Correct. In a doubly linked list, deletion becomes easier because you can access the previous node directly using the prev pointer, without needing to traverse the list from the head.
* **D) It reduces the memory overhead by eliminating the need for a next pointer**: Incorrect. The memory overhead is actually higher in a doubly linked list because each node needs two pointers (next and prev), compared to just one in a singly linked list.

(Hard level)

**Question 1 (Single Choice)**

**What will be the output of the following code?**

struct Node {

int data;

struct Node\* next;

};

int main() {

struct Node\* head = (struct Node\*)malloc(sizeof(struct Node));

head->data = 5;

head->next = head; // Points to itself

printf("%d", head->next->data);

return 0;

}

A) 5  
B) Segmentation fault  
C) Compilation error  
D) Undefined behavior

**Answer:** A) 5

**Explanation:**

* **A) 5**: Correct. Since head->next points to head itself, head->next->data is the same as head->data, which is 5.
* **B) Segmentation fault**: Incorrect. No invalid memory access occurs, so there is no segmentation fault.
* **C) Compilation error**: Incorrect. The code compiles and runs without errors.
* **D) Undefined behavior**: Incorrect. The behavior is well-defined; the node simply points to itself.

**Question 2 (Single Choice)**

**What will be the output of the following code?**

struct Node {

int data;

struct Node\* next;

};

int main() {

struct Node\* head = (struct Node\*)malloc(sizeof(struct Node));

struct Node\* second = (struct Node\*)malloc(sizeof(struct Node));

head->data = 10;

second->data = 20;

head->next = second;

second->next = head;

free(head);

printf("%d", second->next->data);

return 0;

}

A) 10  
B) Segmentation fault  
C) Undefined behavior  
D) Compilation error

**Answer:** C) Undefined behavior

**Explanation:**

* **A) 10**: Incorrect. The pointer second->next points to the freed memory of head, which is invalid to access.
* **B) Segmentation fault**: Incorrect. This may or may not cause a segmentation fault, depending on the system.
* **C) Undefined behavior**: Correct. Accessing second->next->data after freeing head is undefined behavior.
* **D) Compilation error**: Incorrect. The code compiles correctly but exhibits undefined runtime behavior.

**Question 3 (Multiple Choice)**

**What will be the output of the following code?**

struct Node {

int data;

struct Node\* next;

};

int main() {

struct Node\* head = (struct Node\*)malloc(sizeof(struct Node));

struct Node\* second = (struct Node\*)malloc(sizeof(struct Node));

head->data = 5;

head->next = second;

second->data = 10;

second->next = NULL;

free(second);

printf("%d ", head->next->data);

return 0;

}

A) 10  
B) Garbage value  
C) Segmentation fault  
D) Undefined behavior

**Answer:** B) Garbage value  
D) Undefined behavior

**Explanation:**

* **A) 10**: Incorrect. second has been freed, so accessing its data is undefined behavior.
* **B) Garbage value**: Correct. After freeing, head->next->data may print garbage data from unallocated memory.
* **C) Segmentation fault**: Incorrect. This might happen, but not guaranteed.
* **D) Undefined behavior**: Correct. Accessing the data of a freed node leads to undefined behavior.