DATA STRUCTURE

ISE-I(g-4)

Representation, Operation of Doubly and circular Linked List

INTRODUCTION TO DATA STRUCTURES AND IT’S TYPES

**ROLL NO. URN NAME**

1. 2026 1022091026 Anuj .P. Patil
2. 2028 1022091028 Prathmesh .A. Jadhav
3. 2029 1022091029 Rakesh .S. Yadav
4. 2031 1022091031 Shivtej .M. Sul
5. 2033 1022091033 Samadhan .A. Lade
6. 2034 1022091034 Prabhu .R. Badkar
7. 2037 1022091037 Ashitosh .M. Mane

**What Are Data Structures?**

In the realm of computer science and software engineering, \*\*data structures\*\* are essential components that allow us to efficiently store, organize, and manage data. They serve as the building blocks for designing and implementing algorithms and data-driven solutions. Understanding data structures is paramount for creating software that operates effectively and optimally.

**Types of Data Structures**

Data structures come in various types, each tailored to specific use cases:

1. **Arrays**: Simple, fixed-size data structures, used for basic data storage.
2. Linked Lists: Dynamic structures consisting of nodes, offering flexibility and efficient insertions and deletions.
3. **Stacks and Queues**: Specialized structures for managing data in a last-in, first-out (LIFO) or first-in, first-out (FIFO) manner, respectively.
4. **Trees**: Hierarchical structures, including binary trees and search trees, used for efficient data retrieval.
5. **Graphs**: Complex structures used for modelling relationships and connections between data points.
6. **Hash Tables**: Structures that enable fast data retrieval through key-value pairs.

INTRODUCTION TO DOUBLY LINKED LIST

While singly linked lists offer advantages in terms of simplicity and efficiency, they do have limitations. One significant limitation is the lack of backward traversal. This is where \*\*doubly linked lists\*\* come into play.

A **doubly linked list** is a versatile and powerful data structure used in computer science and software engineering. It shares similarities with its simpler counterpart, the singly linked list, but introduces a significant enhancement: bidirectional traversal. This allows for efficient movement in both forward and backward directions within the list.

Contents Of Doubly Linked List

**Structure of a Doubly Linked List**

The primary components that define a doubly linked list's structure are:

1. Data Elements: The data elements are the actual values or information stored within the list. These can be integers, strings, objects, or any other data type relevant to the application.

2. Pointers (Next and Prev): What sets a doubly linked list apart from a singly linked list is the presence of two pointers in each node. These pointers are as follows:

- Next Pointer: The "next" pointer references the following node in the list, enabling forward traversal. This pointer allows efficient navigation from one node to the next.

- Prev Pointer: The "prev" pointer, short for "previous," points to the node that precedes the current node. It's the key to reverse traversal, facilitating efficient movement backward within the list.

**Visualizing the Structure**

A visual representation can provide a clearer understanding of the doubly linked list's structure. Below is a diagram illustrating the components of a doubly linked list:

Node A: [prev] ↔ [data] ↔ [next]

Node B: [prev] ↔ [data] ↔ [next]

Node C: [prev] ↔ [data] ↔ [next]

In this illustration, "Node A" represents the first node in the list. Each node contains a "prev" pointer, a data element, and a "next" pointer. The "next" pointer of "Node A" points to "Node B," enabling forward traversal, while the "prev" pointer remains null. This structure continues throughout the list, forming a bidirectional chain.

Operations in Doubly

|  |  |
| --- | --- |
| // Function to insert a node at the beginning of a doubly linked list.  struct Node\* insertAtBeginning(struct Node\* head, int newData) {  struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));  if (newNode == NULL) {  printf("Memory allocation failed\n");  return head;  }  newNode->data = newData;  newNode->prev = NULL;  newNode->next = head;  if (head != NULL) {  head->prev = newNode;  } | // Function to delete a node at the start of the doubly linked list  void deleteAtStart(struct Node\*\* head) {  if (\*head == NULL) {  printf("List is empty. Deletion not possible.\n");  return;  }    struct Node\* temp = \*head;  \*head = (\*head)->next;    if (\*head != NULL) {  (\*head)->prev = NULL;  }    free(temp);  } |
| // Function to insert a node at the end of the doubly linked list  void insertAtEnd(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);    if (\*head == NULL) {  \*head = newNode;  } else {  struct Node\* current = \*head;    while (current->next != NULL) {  current = current->next;  }    current->next = newNode;  newNode->prev = current;  }  } | // Function to insert a node at the end of the doubly linked list  void insertEnd(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);    if (\*head == NULL) {  \*head = newNode;  } else {  struct Node\* current = \*head;    while (current->next != NULL) {  current = current->next;  }    current->next = newNode;  newNode->prev = current;  }  } |
| // Function to insert a node at a specific position in the doubly linked list  void insertAtPosition(struct Node\*\* head, int data, int position) {  if (position < 0) {  printf("Invalid position. Insertion not possible.\n");  return;  }  struct Node\* newNode = createNode(data);  if (position == 0) {  if (\*head != NULL) {  newNode->next = \*head;  (\*head)->prev = newNode;  }  \*head = newNode;  return;  }  struct Node\* current = \*head;  int count = 0;  while (count < position - 1 && current != NULL) {  current = current->next;  count++;  }  if (current == NULL) {  printf("Position is out of range. Insertion not possible.\n");  return;  }  newNode->next = current->next;  newNode->prev = current;  current->next = newNode;  if (newNode->next != NULL) {  newNode->next->prev = newNode;  }  } | /// Function to delete a node at a specific position in the doubly linked list  void deleteAtPosition(struct Node\*\* head, int position) {  if (\*head == NULL) {  printf("List is empty. Deletion not possible.\n");  return;  }  struct Node\* current = \*head;  int count = 0;  while (current != NULL && count < position) {  current = current->next;  count++;  if (current == NULL) {  printf("Position %d is out of range. Deletion not possible.\n", position);  return;  }  if (current->prev != NULL) {  current->prev->next = current->next;  } else {  \*head = current->next;  }  if (current->next != NULL) {  current->next->prev = current->prev;  }  free(current);  } |

Advantages and Applications

**Advantages:**

**1. Bidirectional Traversal**

Doubly linked lists enable efficient traversal in both forward and backward directions. This bidirectional capability is a standout advantage, as it simplifies data manipulation and access.

**2. Efficient Insertions and Deletions**

The presence of "prev" pointers in doubly linked lists makes insertions and deletions highly efficient. Nodes can be added or removed at both the beginning and end of the list with ease, without the need to traverse the entire list.

**3. Applications in Real-World Scenarios**

Doubly linked lists find application in a wide array of real-world scenarios where bidirectional navigation is crucial. Let's delve into a few such scenarios:

**Use Cases and Examples:**

* Music Playlists

Music playlists can be seen as a series of songs where users may want to skip forward or backward. Doubly linked lists enable the seamless movement between songs in both directions. It's the technology behind the "previous" and "next" buttons on your music player.

* Browser History

Web browsers maintain a history of visited web pages. The "Back" and "Forward" buttons on browsers utilize doubly linked lists to allow users to navigate their browsing history efficiently.

* Undo/Redo Functionality

In applications that require undo/redo functionality, doubly linked lists store the sequence of actions taken by the user. Users can undo previous actions and redo them in both directions, making these operations quick and responsive.

**CIRCULAR DOUBLY LINKED LIST**

A **circular doubly linked list** is an extension of the standard doubly linked list. In a circular doubly linked list, the last node points back to the first node, creating a closed loop. This structure is particularly useful for applications where you need continuous and seamless traversal, such as implementing circular buffers or managing tasks in a round-robin scheduling system.

Example: Circular doubly linked lists are employed in real-time systems to manage tasks, ensuring that each task is executed in a round-robin fashion without the need for a starting or ending point.

**What is a Circular Doubly Linked List?**

A circular doubly linked list is a data structure used in computer science to organize a collection of elements, just like a regular doubly linked list. However, it has a unique feature: the last element is connected to the first element, forming a circular structure. In this structure, each element, or node, contains two references, one pointing to the next element and the other pointing to the previous element. This circular arrangement allows for efficient traversal in both forward and backward directions.

Advantages and Applications

**Advantages**

1. Efficient Traversal: Since the list is circular, you can easily traverse it in both directions without the need to loop back to the beginning.

2. Insertions and Deletions: Insertions and deletions at both ends of the list can be performed in constant time. In a non-circular list, updating the last element requires linear time, which is not the case in a circular list.

3. Looping and Rotating: Circular lists are well-suited for tasks that involve looping through a set of elements, such as implementing certain algorithms.

**APPLICATION**

1. Music and Video Playlists: Circular linked lists can be used to create playlists that continuously loop through a collection of songs or videos.

2. Image Slideshows: They are suitable for implementing image slideshows that cycle through a set of images endlessly.

3. Game Development: In game development, they can be used for various purposes, such as cycling through game levels or managing character inventories.

Basic Structure and Operations

**In a circular doubly linked list, each node has a specific structure:**

**Data**: This is where you store the actual data or payload associated with the node.

**Next Pointer**: This pointer points to the next node in the list, allowing you to traverse the list in a forward direction.

**Previous Pointer**: This pointer points to the previous node in the list, enabling backward traversal.

**Circular Connection**: The last node in the list connects back to the first node, creating a circular structure.

**OPERATIONS ON CIRCULAR DOUBLY LINKED LIST**

1. **Insertion at the Beginning**: To insert a node at the beginning of the list, you need to update the next and previous pointers of the new node, the old first node, and the old last node. This operation takes constant time (O(1)).

2. **Insertion at the End**: Similar to insertion at the beginning, inserting at the end also requires updates to pointers. This operation also takes O(1) time.

3. **Deletion of a Node**: When deleting a node, you update the next and previous pointers of the adjacent nodes to bypass the node being removed. Like insertions, this operation takes O(1) time.

4. **Deletion of the Entire List**: To delete the entire list, you start by breaking the circular connection and then traverse the list to delete each node individually. This operation takes O(n) time, where n is the number of nodes in the list.

Implementing Circular Doubly Linked Lists

|  |  |
| --- | --- |
| // Function to insert a node at the beginning of the circular doubly linked list  void insertAtBeginning(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  } else {  newNode->next = \*head;  newNode->prev = (\*head)->prev;  (\*head)->prev->next = newNode;  (\*head)->prev = newNode;  \*head = newNode;  }  } | // Function to delete a node at the beginning of a circular doubly linked list:  void deleteAtBeginning(struct Node\*\* head) {  if (\*head == NULL) {  printf("List is empty. Deletion not  possible.\n");  return;  }  struct Node\* toDelete = \*head;  if (toDelete->next == \*head) {  \*head = NULL;  } else {  (\*head)->prev->next = toDelete->next;  toDelete->next->prev = (\*head)->prev;  \*head = toDelete->next;  }  free(toDelete);  } |
| // Function to insert a node at the end of the circular doubly linked list  void insertAtEnd(struct Node\*\* head, int data) {  struct Node\* newNode = createNode(data);  if (\*head == NULL) {  \*head = newNode;  } else {  newNode->next = \*head;  newNode->prev = (\*head)->prev;  (\*head)->prev->next = newNode;  (\*head)->prev = newNode;  }  } | // Function to delete a node at the end of a circular doubly linked list  void deleteAtEnd(struct Node\*\* head) {  if (\*head == NULL) {  printf("List is empty. Deletion not possible.\n");  return;  }  struct Node\* toDelete = (\*head)->prev;  if (toDelete == \*head) {  \*head = NULL;  } else {  toDelete->prev->next = \*head;  (\*head)->prev = toDelete->prev;  }  free(toDelete);  } |
| // Function to insert a node after a specific node with a given value  void insertAfterValue(struct Node\*\* head, int value, int data) {  if (\*head == NULL) {  printf("List is empty. Cannot insert after value.\n");  return;  }    struct Node\* current = \*head;  do {  if (current->data == value) {  struct Node\* newNode = createNode(data);  newNode->next = current->next;  newNode->prev = current;  current->next->prev = newNode;  current->next = newNode;  return;  }  current = current->next;  } while (current != \*head); | // Function to delete a node at a specific position in the circular doubly linked list  void deleteAtPosition(struct Node\*\* head, int position) {  if (\*head == NULL) {  printf("List is empty. Deletion not possible.\n");  return;  }  struct Node\* current = \*head;  int count = 0;  do {  if (count == position) {  struct Node\* prevNode = current->prev;  struct Node\* nextNode = current->next;  prevNode->next = nextNode;  nextNode->prev = prevNode;  if (current == \*head) {  \*head = nextNode;  }  free(current);  return;  }  current = current->next;  count++;  } while (current != \*head); |

|  |  |  |  |
| --- | --- | --- | --- |
|  | DATA STRUCTURE | |  |
|  | |
|  | |
|  | QUESTIONS | |  |
|  |  |  |
|  | |  |

Question 1 (Creating): Imagine you are developing a music player application. Which data structure would you choose for managing the playlist, allowing users to efficiently move both forward and backward through the song list?

a. Doubly linked list

b. Singly linked list

c. Circular linked list

d. Binary tree

Answer: a) Doubly linked list

Question 2 (Comprehension): In a doubly linked list, what is the purpose of the "prev" pointer in each node?

a) It points to the next node in the list

b) It points to the previous node in the list

c) It points to the current node's value

d) It is not used in a doubly linked list

Answer: b) It points to the previous node in the list

**Question 3 (Application):** In which real-world application would a circular doubly linked list be most useful?

A. Maintaining a to-do list

B. Implementing a file system

C. Representing a basic queue

D. Managing a simple stack

**Answer:** B. Implementing a file system

Question 4 (Comprehension): In a circular doubly linked list, what is the purpose of making the "next" pointer of the last node point back to the first node?

a) It ensures that the list cannot be modified

b) It allows for easier traversal in both directions

c) It makes the list a singly linked list

d) It improves memory efficiency

Answer: b) It allows for easier traversal in both directions

**Question 5(Comprehension):** Which of the following operations can be performed more efficiently in a doubly linked list compared to a singly linked list?

A. Forward traversal

B. Insertion at the beginning

C. Deletion of a specific node

D. Backward traversal

**Answer:** D. Backward traversal

Question 6: Which of the following is a disadvantage of using a doubly linked list over a singly linked list?

a) Doubly linked lists require less memory.

b) Doubly linked lists are harder to implement.

c) Doubly linked lists use more memory due to the extra "prev" pointers.

d) Doubly linked lists are always faster for traversing

Answer: c) Doubly linked lists use more memory due to the extra "prev" pointers.

Question 7: Which of the following data structures is most suitable for implementing an undo/redo functionality in software applications?

a) Doubly linked list

b) Circular singly linked list

c) Binary tree

d) Stack

Answer: a) Doubly linked list

Question 8: Why is a circular doubly linked list often preferred over a circular singly linked list for certain applications?

) It uses less memory.

b) It allows for faster insertions.

c) It enables efficient traversal in both directions without additional checks.

d) It simplifies the implementation of circular lists.

Answer: c) It enables efficient traversal in both directions without additional checks.

Question 9: You are tasked with implementing a text editor with undo/redo functionality. Which data structure is most suitable for storing the sequence of text changes and managing the undo/redo operations?

a) Singly linked list

b) Doubly linked list

c) Circular singly linked list

d) Circular doubly linked list

Answer: b) Doubly linked list

Question 10: You are given a large doubly linked list containing millions of nodes. You are required to design an efficient algorithm to reverse the entire list. What is the most time-efficient approach for this task?

a) Use an iterative approach to traverse the list and reverse the pointers in place.

b) Utilize a recursive algorithm to reverse the list.

c) Create a new list and insert the nodes in reverse order.

d) Implement an algorithm that uses multithreading for faster reversal.

Answer: a) Use an iterative approach to traverse the list and reverse the pointers in place.

# SHORT QUESTIONS

(Application):  
Question 1: Provide an example of a real-world application where a doubly linked list would be preferred over a singly linked list.  
Answer: A text editor with undo/redo functionality benefits from a doubly linked list for efficient traversal.

Comprehension Level:

Question 2: Why is a doubly linked list preferred over a singly linked list when reverse traversal is required?

Answer: A doubly linked list is preferred because it allows for efficient reverse traversal by using the "prev" pointers in each node.

Application Level:

Question 3: Given a doubly linked list with nodes A ↔ B ↔ C, how would you insert a new node D between nodes B and C?

Answer : To insert node D between B and C, you would update the "next" and "prev" pointers of B, C, and D to link them in the correct order: B ↔ D ↔ C.

(Comprehension)  
Question 4: How does a doubly linked list compare to an array in terms of memory usage and flexibility for inserting elements?  
Answer: A doubly linked list is more flexible for inserting elements but typically consumes more memory due to the additional pointers.

Evaluation Level:

Question 5: Compare the advantages of using a doubly linked list over an array for implementing a dynamic data structure.

Answer: A doubly linked list offers advantages such as efficient insertions and deletions, flexibility in memory allocation, and the ability to easily support data reordering, whereas an array is generally faster for random access but has fixed size limitations and may require more memory overhead.

# LONG QUESTIONS

Question 1 (Knowledge Level):

## Explain the fundamental structure of a doubly linked list. Describe the role and purpose of the "next" and "prev" pointers in each node. Provide an example to illustrate this structure.\*

Answer 1 (Knowledge Level):

A doubly linked list is a versatile data structure used in computer science. It consists of nodes, each of which holds a data element and two pointers. The first pointer, known as the "next" pointer, references the following node in the list, enabling forward traversal. The second pointer, the "prev" pointer, points to the previous node, facilitating reverse traversal. This dual-pointer feature sets doubly linked lists apart from singly linked lists, where each node has only a "next" pointer.

To illustrate, consider the example of a doubly linked list:

```

Node A: [prev] ↔ [data] ↔ [next]

Node B: [prev] ↔ [data] ↔ [next]

Node C: [prev] ↔ [data] ↔ [next]

```

Here, Node A has its "next" pointer directed towards Node B, and Node B's "prev" pointer connects back to Node A. This structure allows bidirectional navigation, making it invaluable in scenarios where both forward and backward traversal are essential.

Question 2 (Comprehension Level):

## Compare and contrast the advantages of a singly linked list and a doubly linked list. Provide examples of scenarios where each type of list is more suitable.

Answer 2 (Comprehension Level):

Singly linked lists and doubly linked lists are two common data structures, each with unique advantages and use cases.

Singly linked lists consist of nodes, each of which has a "next" pointer pointing to the following node, but no "prev" pointer for reverse traversal. They are memory-efficient as they have only one pointer per node, making them suitable for situations where memory is a constraint. They are also relatively straightforward to implement for basic operations like insertion and deletion at the beginning. Singly linked lists are well-suited for applications where forward traversal is the primary requirement, such as task scheduling, simple queues, and stacks.

Doubly linked lists, on the other hand, offer a more sophisticated structure. In addition to the "next" pointer, each node features a "prev" pointer that enables reverse traversal. This is particularly valuable in scenarios that require both forward and backward navigation, like text editors, music playlists, or navigating through large documents. Doubly linked lists excel at operations like insertions and deletions at both ends, providing faster access to the head and tail of the list.

The choice between singly and doubly linked lists depends on the specific requirements of the application. For resource-constrained environments and situations where forward traversal is sufficient, singly linked lists are preferred. For more complex use cases that demand bidirectional traversal and versatile manipulation, doubly linked lists are the superior choice.

Question 3 (Application Level):

## Design an algorithm to insert a new node with data "X" after a given node "N" in a doubly linked list. Provide the pseudocode and describe the steps involved.

Answer 3 (Application Level):

To insert a new node with data "X" after a given node "N" in a doubly linked list, you can follow these steps:

1. Create a new node, "newNode," with the data "X."

2. Update the "next" pointer of "newNode" to point to the node following node "N."

3. Set the "prev" pointer of "newNode" to point to node "N."

4. Adjust the "next" pointer of node "N" to reference "newNode."

5. If "newNode" is not the last node in the list, update the "prev" pointer of the node following "newNode" to point back to "newNode."

Here's the pseudocode that outlines this process:

```

Procedure InsertAfter(N, X):

newNode = CreateNodeWithData(X)

newNode.next = N.next

newNode.prev = N

N.next = newNode

if newNode.next is not null:

newNode.next.prev = newNode

```

This algorithm ensures that the new node is seamlessly inserted after node "N" in the doubly linked list.

Question 4 (Analysis Level):

## Analyze the time complexity of searching for a specific element in a doubly linked list with "n" nodes. How does the time complexity compare to searching in an array?

Answer 4 (Analysis Level):

Searching for a specific element in a doubly linked list with "n" nodes has a time complexity of O(n) in the worst case. To locate the desired element, the list may need to be traversed from the beginning to the end, making it linear in terms of time complexity.

Comparatively, searching in an array also has a worst-case time complexity of O(n). However, it's essential to note that arrays offer constant-time (O(1)) access for elements if you already have their index. In contrast, a linked list, whether singly or doubly linked, lacks direct index-based access, leading to a linear time complexity for searching.

Therefore, while both data structures have a worst-case time complexity of O(n), arrays are more efficient for random access to elements, making them the preferred choice in scenarios that require frequent direct access to specific positions in the data.

Question 5 (Evaluation Level):

## Evaluate the advantages and challenges of using a circular doubly linked list in comparison to a regular doubly linked list. Provide examples of scenarios where a circular doubly linked list would be particularly beneficial.

Answer 5 (Evaluation Level):

Advantages of Circular Doubly Linked List:

Circular doubly linked lists are a specialized form of doubly linked lists that offer specific advantages in certain scenarios:

1. Seamless Looping: Circular doubly linked lists are well-suited for applications that require seamless looping. For instance, in a music playlist, the last song in the list can transition seamlessly to the first song without an abrupt stop or jump. This circular behavior creates a more enjoyable listening experience.

2. Efficient Continuous Traversal: Circular doubly linked lists eliminate the need to check for null pointers during traversal. The last node points to the first, enabling smooth, continuous movement both forward and backward. This feature is particularly beneficial in applications where items need to cycle through a process or sequence endlessly.

3. Space-Efficient Looping: Unlike regular doubly linked lists, circular doubly linked lists do not require an additional pointer to maintain a reference to the head or tail of the list. This results in a more memory-efficient structure, which can be advantageous in resource-constrained environments.

Challenges:

1. Potential for Infinite Loops: While seamless looping is an advantage, it can also lead to challenges. Careful handling is required to avoid infinite loops, especially during operations such as traversal, insertion, or deletion.

2. Additional Complexity: Managing circular links adds complexity to the implementation of circular doubly linked lists. Operations like insertion and deletion need to account for the circular behavior, which can make code more intricate.

Scenarios where Circular Doubly Linked Lists are Beneficial:

- Music Playlists: Circular doubly linked lists are ideal for creating music playlists where songs should seamlessly loop from the last to the first, providing a continuous and enjoyable listening experience.

- Scheduling Algorithms: In scheduling

algorithms, processes can be organized in a circular doubly linked list to simulate cyclic processes, ensuring that each process gets its turn without interruption.

- Circular Buffers: Circular doubly linked lists are used in circular buffers for efficient data storage, where data elements wrap around after reaching the end, ensuring a continuous stream of data without gaps or overwrites.

The choice between a circular and regular doubly linked list depends on the specific application's requirements. Circular doubly linked lists are most valuable in situations where the circular behavior is advantageous and can enhance user experience. However, they require careful implementation to avoid potential pitfalls, such as infinite loops.