DATA STRUCTURES – UNIT 4: LINKED LIST

COMPLETE EXAM-READY QUESTION BANK WITH ANSWERS

SECTION 1: LONG ANSWER QUESTIONS AND ANSWERS (5 MARKS EACH)

Q1) What is a Linked List? Explain its basic structure and components.

Answer:

Linked list is an ordered collection of finite homogeneous data elements called nodes where linear order is maintained by means of links or pointers rather than physical placement.

Node structure components:

(a) Data field (INFO field): Stores the actual information or value

(b) Link field (NEXT pointer): Contains address/reference to the next node in sequence

Head pointer is a special pointer that points to the first node of the linked list; it serves as the entry point to access the entire list.

NULL pointer marks the end of the linked list; the last node's link field contains NULL to indicate no further nodes exist.

Unlike arrays, linked list elements are not stored in contiguous memory locations; each node can be placed anywhere in memory.

Each node is accessed through its predecessor node by following the chain of pointers starting from the head pointer.

Example: In a list storing marks {85, 92, 78}, three separate nodes are created with data values and pointers linking them sequentially, terminated by NULL.

Q2) Explain Singly Linked List (SLL) with its structure and characteristics.

Answer:

Singly Linked List is a linear data structure where each node contains only one link field pointing to the next node in the list.

Structure consists of:

(a) Header node (optional): First node with no data, link field points to first data node

(b) Data nodes: Each contains one data field and one next pointer

Traversal is unidirectional; we can move only in forward direction from head to tail, cannot move backward.

Representation in memory:

```
Code Example

HEAD → [DATA|NEXT] → [DATA|NEXT] → [DATA|NEXT] → NULL
       Node 1          Node 2          Node 3
```

Last node's next pointer contains NULL value indicating end of list; this distinguishes the tail node.

Operations like insertion and deletion require maintaining only one link per node, making implementation simpler than doubly linked lists.

Example: Student roll number list {101, 105, 109} stored as:

HEAD → [101|addr2] → [105|addr3] → [109|NULL]

Q3) What is a Circular Singly Linked List? Explain its characteristics and advantages.

Answer:

Circular Singly Linked List is a variation of singly linked list where the last node points back to the header node instead of NULL, forming a circular chain.

Key characteristic: There are no NULL links in the list; every node has a valid next pointer creating a closed loop.

Structure representation:

```
Code Example

HEAD ←──────────────────────────────────┐
  ↓                                     |
[DATA|NEXT] → [DATA|NEXT] → [DATA|NEXT]
```

Traversal can start from any node and will eventually return to the starting point after visiting all nodes.

Advantages over linear singly linked list:

(a) Any node can be reached from any other node by following the circular path

(b) Useful for applications requiring round-robin scheduling or circular buffer implementation

Detecting end of list requires checking if current node's next pointer equals head pointer rather than checking for NULL.

Example application: Implementation of circular queue, where rear node connects back to front node automatically.

Q4) Explain Doubly Linked List (DLL) with its structure and features.

Answer:

Doubly Linked List is a linear data structure where each node contains two link fields: one pointing to the next node and another pointing to the previous node.

Node structure contains three fields:

(a) Data field: Stores the actual information

(b) PREV pointer: Contains address of previous node

(c) NEXT pointer: Contains address of next node

Structure representation:

```
Code Example

HEAD → [PREV|DATA|NEXT] ⇄ [PREV|DATA|NEXT] ⇄ [PREV|DATA|NEXT] → NULL
       NULL←
```

Bidirectional traversal is possible; we can move both forward (using next pointers) and backward (using prev pointers) through the list.

First node's prev pointer contains NULL; last node's next pointer contains NULL, marking the boundaries of the list.

Advantages over singly linked list:

(a) Quick access to predecessor node without traversing from head

(b) Simpler implementation of deletion operation as previous node is directly accessible

Disadvantage: Requires extra memory for storing additional prev pointer in each node, increasing space complexity.

Q5) What is a Circular Doubly Linked List? Explain its structure and characteristics.

Answer:

Circular Doubly Linked List combines features of both circular list and doubly linked list, where last node points to header and header points to last node.

Structure contains no NULL links; both forward and backward chains form complete circles creating a closed bidirectional loop.

Structure representation:

```
Code Example

       ┌──────────────────────────────────────────┐
       ↓                                          ↑
HEAD ⇄ [PREV|DATA|NEXT] ⇄ [PREV|DATA|NEXT] ⇄ [PREV|DATA|NEXT]
  ↑                                                  ↓
  └──────────────────────────────────────────────────┘
```

Bidirectional circular traversal is possible; can move forward endlessly or backward endlessly through all nodes.

Characteristics:

(a) HEAD→next points to first data node; HEAD→prev points to last data node

(b) Last node→next points back to HEAD; First node→prev points back to HEAD

Advantages: Most flexible linked list structure allowing efficient operations at both ends and easy implementation of deque (double-ended queue).

Detecting list traversal completion requires checking if current node equals head node in either direction, not checking for NULL.

Q6) Differentiate between Singly Linked List (SLL) and Doubly Linked List (DLL).

Answer:

Aspect	Singly Linked List (SLL)	Doubly Linked List (DLL)
Number of pointers	Each node contains only one pointer (next)	Each node contains two pointers (prev and next)
Traversal direction	Unidirectional traversal only, can move forward from head to tail	Bidirectional traversal possible, can move forward and backward
Space complexity	Less memory overhead, requires space for one pointer per node	Higher memory overhead, requires space for two pointers per node
Access to predecessor	Requires traversal from head to find previous node, O(n) time	Direct access to previous node through prev pointer, O(1) time
Insertion/Deletion	Need reference to previous node for certain operations	Simpler operations as both neighbors are directly accessible
Implementation	Simpler to implement and manage with fewer pointer manipulations	More complex with need to maintain consistency of two pointers
Use cases	Suitable when forward-only traversal sufficient and memory is limited	Preferred when frequent bidirectional access or easier deletions needed

---

Q7) Differentiate between Linear Linked List and Circular Linked List.

Answer:

Aspect	Linear Linked List	Circular Linked List
Last node structure	Last node's next pointer contains NULL, indicating end of list	Last node's next pointer points back to header/first node
Traversal	Traversal has definite start (head) and end (NULL), terminates automatically	No natural end point, traversal continues indefinitely unless explicitly stopped
End detection	Check if current node's next pointer is NULL	Check if current node's next pointer equals head pointer
Node reachability	Must always start from head, cannot reach earlier nodes from later nodes	Can reach any node from any other node by following circular path

---

Aspect	Linear Linked List	Circular Linked List
Memory representation	Contains at least one NULL pointer (in last node)	Contains no NULL pointers, all links are valid addresses
Applications	Suitable for simple sequential data processing with clear beginning and end	Ideal for round-robin scheduling, circular buffers, repeated cyclic operations
Example	HEAD→[A]→[B]→[C]→NULL	HEAD→[A]→[B]→[C]→HEAD

Q8) Explain the advantages of Linked Lists over Arrays.

Answer:

Dynamic size: Linked lists can grow or shrink during runtime by allocating/deallocating memory as needed; arrays have fixed size determined at compile time.

Efficient insertion and deletion: Adding or removing elements requires only pointer manipulation without shifting elements; arrays require shifting all subsequent elements.

No contiguous memory requirement: Nodes can be scattered anywhere in memory; arrays require large contiguous memory block which may not be available.

No memory wastage: Memory allocated exactly as per requirement; arrays may waste space if not fully utilized or overflow if size exceeded.

Flexible memory management: Can grow until system memory exhausted; arrays cannot grow beyond declared size without reallocation.

No overflow condition: Only limited by available system memory; arrays have fixed upper limit causing overflow errors.

Example: Inserting element at beginning of linked list takes O(1) time; in array it requires O(n) time to shift all elements.

Q9) Explain the disadvantages of Linked Lists compared to Arrays.

Answer:

No random access: Cannot directly access element by index; must traverse from head taking O(n) time, while arrays provide O(1) index-based access.

Extra memory overhead: Each node requires additional memory for storing pointer(s); arrays store only data without pointer overhead.

Cache performance: Poor cache locality as nodes scattered in memory; arrays have better cache performance due to contiguous storage.

Traversal speed: Sequential access slower due to pointer following; arrays provide faster sequential access through contiguous memory.

Reverse traversal difficulty: SLL cannot traverse backward efficiently; arrays can move in any direction easily using indices.

Complex implementation: Requires careful pointer manipulation and memory management; arrays are simpler to implement and use.

Memory fragmentation: Frequent allocation/deallocation can fragment memory; arrays use single contiguous block avoiding fragmentation.

Example: Binary search impossible on linked list (requires random access); arrays support binary search efficiently.

Q10) List and explain applications of Linked Lists.

Answer:

Polynomial representation and manipulation:

Linked lists store polynomial terms efficiently with coefficient and exponent in each node.

Each term stored in separate node avoids memory waste for sparse polynomials.

Operations like addition and multiplication simplified through list traversal.

Dynamic memory management:

Operating system maintains free memory blocks as linked list (free list).

Memory allocator links freed blocks together for reuse.

Garbage collection uses linked structures to track and reclaim unused memory.

Implementation of other data structures:

Stacks and queues can be implemented using linked lists providing dynamic size.

Graph adjacency lists use linked lists to store neighbors of each vertex.

Hash tables use linked lists for chaining in collision resolution.

Symbol table management in compilers:

Compiler maintains identifiers and their attributes in linked list.

Allows dynamic addition of new symbols during compilation.

Image viewer and music player applications:

Images or songs stored as linked list nodes.

Previous and next operations navigate through list using DLL.

Undo/Redo functionality:

Applications maintain history of actions as linked list.

Undo moves backward, redo moves forward through the list.

Q11) Explain representation of Linked List using sequential organization (array).

Answer:

Sequential representation uses two parallel arrays: DATA[ ] for storing elements and LINK[ ] for storing next node indices.

Implementation details:

DATA[i] stores the actual data element.

LINK[i] stores array index of next node (not memory address).

Special variable HEAD stores index of first node.

LINK value of -1 indicates end of list (equivalent to NULL).

Example: List L = {Jan, Feb, Mar, Apr, May}

Array indices don't follow list order; LINK array maintains logical sequence.

```
Code Example

Index:  1    2    3    4    5
DATA:  [Apr][May][Jan][Feb][Mar]
LINK:  [5]  [-1] [4]  [1]  [2]
HEAD = 3
```

Traversal: Start at HEAD=3 (Jan), follow LINK[3]=4 (Feb), then LINK[4]=1 (Apr), then LINK[1]=5 (Mar), then LINK[5]=2 (May), then LINK[2]=-1 (end).

Advantages: Simple implementation, no dynamic memory allocation needed, useful for educational purposes.

Disadvantages: Fixed size array limitation, memory wastage if array not fully used, pointer arithmetic not as natural.

Q12) Explain representation of Linked List using dynamic organization (pointers).

Answer:

Dynamic representation uses actual memory addresses (pointers) to link nodes, with memory allocated at runtime using malloc/new.

Node structure definition:

```
Code Example

struct node {
    int data;
    struct node *next;
};
```

Memory allocation: Each node dynamically allocated from heap memory when needed using malloc() or new operator.

Pointer operations:

HEAD is pointer variable storing address of first node.

Each node's next field stores actual memory address of next node.

NULL pointer (address 0) indicates end of list.

Example: Creating node and linking:

```
Code Example

struct node *newNode = (struct node *)malloc(sizeof(struct node));
newNode->data = 10;
newNode->next = NULL;
```

Advantages:

(a) True dynamic sizing - grows and shrinks as needed

(b) Efficient memory utilization - allocates only required memory

(c) Natural pointer-based implementation

Memory management: Programmer responsible for freeing memory using free() or delete to avoid memory leaks.

Representation: Nodes scattered throughout heap memory, connected only through pointers creating flexible structure.

Q13) Explain basic operations performed on Linked Lists.

Answer:

Creating a List: Initialize head pointer to NULL for empty list, then repeatedly create nodes and link them together.

List Traversal: Start from head, follow next pointers until NULL reached, visiting each node to display or process data.

Inserting an element:

(a) At beginning: Create new node, set its next to head, update head to new node

(b) At end: Traverse to last node, set its next to new node

(c) At position: Traverse to position-1, adjust pointers to insert new node

Deleting an element:

(a) From beginning: Update head to head->next, free old head node

(b) From end: Traverse to second-last node, set its next to NULL, free last node

(c) From position: Traverse to position-1, adjust pointers to skip target node, free it

Searching a list: Traverse list comparing each node's data with search key until found or list ends.

Reversing a list: Change direction of all next pointers so they point to previous nodes instead of next nodes.

Merging two lists: Combine two sorted lists into single sorted list by comparing elements and adjusting pointers.

Q14) Explain Polynomial representation using Linked List.

Answer:

Polynomial represented as linked list where each node stores one term with coefficient and exponent.

Node structure for polynomial:

```
Code Example

struct polyNode {
    int coef;       // coefficient
    int exp;        // exponent
    struct polyNode *next;
};
```

Storage format: Terms arranged in descending order of exponents for efficient operations.

Example: $P(x) = 5x^3 + 3x^2 + 2x + 7$

Represented as: HEAD → [5|3] → [3|2] → [2|1] → [7|0] → NULL

Advantages over array representation:

(a) No memory waste for sparse polynomials (polynomials with few non-zero terms)

(b) Easy insertion and deletion of terms

(c) Dynamic size - can represent polynomials of any degree

Operations: Addition and multiplication of polynomials implemented by traversing lists and creating result list with appropriate terms.

Example of sparse polynomial: $P(x) = 7x^{1000} + 3x^2 + 2$

Only 3 nodes needed in linked list; array would need 1001 positions (wasteful).

Q15) Explain Polynomial addition using Linked Lists.

Answer:

Polynomial addition combines two polynomials P1(x) and P2(x) to produce result P3(x) by comparing and combining terms.

Algorithm approach: Use two pointers to traverse P1 and P2 simultaneously, comparing exponents at each step.

Case 1 - Exponent P1 > Exponent P2:

Copy term from P1 to result list P3.

Move P1 pointer to next term.

Case 2 - Exponent P1 < Exponent P2:

Copy term from P2 to result list P3.

Move P2 pointer to next term.

Case 3 - Exponent P1 = Exponent P2:

Add coefficients of both terms.

Store result in P3 with same exponent (if sum is non-zero).

Move both P1 and P2 pointers to next terms.

Completion step: After main comparison loop, copy all remaining terms from whichever polynomial (P1 or P2) still has terms left.

Example: P1 = $5x^3 + 3x$, P2 = $4x^3 + 2x^2 + 7$

Result P3 = $9x^3 + 2x^2 + 3x + 7$

Time complexity: O(m + n) where m and n are number of terms in P1 and P2.

END OF QUESTION BANK

Note: This question bank covers all important theory topics from Unit 4. Practice the algorithms and problem-types listed in Section 2 separately by writing pseudocode/code and doing dry runs. Master both theory concepts and algorithmic implementations for complete Unit 4 exam preparation.

Total Questions: 20 Long Answer Questions (5 marks each)

Coverage: Complete Unit 4 - Linked List syllabus based on university notes and previous year patterns.