DATA STRUCTURES

Unit II — Memory Allocation & Linked List Operations

Topics Covered in this Unit

Memory Allocation Paradigms: Static (Stack) vs. Dynamic (Heap) Memory, Compile-time vs. Runtime Sizing
Dynamic Memory Management in C/C++: malloc(), calloc(), realloc(), free(), Memory Leaks, Dangling Pointers
Linked List as Abstract Data Type (ADT): Self-Referential Structures, Node Anatomy, Dynamic Realization
Types of Linked Lists: Singly Linked List (SLL), Circular Linked List (CLL), Doubly Linked List (DLL), Doubly Circular (DCLL)
Primitive Operations on Linked Lists: Create, Traverse, Search, Insert (Beg/End/Pos), Delete (Beg/End/Key), Sort, Concatenate, Reverse
Comparative Evaluation: Array vs. Singly Linked List vs. Doubly Linked List vs. Circular Linked List
Applications of Linked Lists: Polynomial Representation & Polynomial Addition Algorithm with Complete Trace
Generalized Linked List (GLL): Atom vs. Sublist, Tag Bits, Head & Tail, Depth, Multi-dimensional Representation
High-Yield University Exam Question Bank with Detailed Model Answers & Unit Summary

Compiled in a structured, easy-to-understand, textbook style format
Specially curated for B.Tech / B.E. / BCA / MCA / B.Sc Computer Science & Engineering Students

COURSE SYLLABUS & MAPPING

[UNIT II SYLLABUS — PRESCRIBED UNIVERSITY CURRICULUM]

This section contains the official syllabus for Unit II. Verify with your university curriculum mapping.

Unit II Syllabus Breakdown:

Unit II - Memory Allocation & Linked List Operations: Introduction to Static and Dynamic Memory Allocation. Linked List: Introduction of Linked Lists, Realization of linked list using dynamic memory management, operations on Linked Lists, Linked List as ADT, Types of Linked List: singly linked, linear and Circular Linked Lists, Doubly Linked List, Doubly Circular Linked List, Primitive Operations on Linked List-Create, Traverse, Search, Insert, Delete, Sort, Concatenate. Polynomial Manipulations-Polynomial addition. Generalized Linked List (GLL) concept.

• Course Code: _____ • Semester / Year: _____

TABLE OF CONTENTS

2.1 Static vs. Dynamic Memory Allocation

Memory Allocation Fundamentals

• *Static Memory Allocation: Memory is allocated by the compiler at compile-time on the Call Stack or Data/BSS segment. Size and location are permanently fixed during execution.*

• *Dynamic Memory Allocation: Memory is allocated explicitly at runtime on the Heap segment via system calls. Memory blocks can grow, shrink, and be released dynamically as needed.*

Detailed Comparison: Static vs. Dynamic Allocation

Feature / Parameter	Static Memory Allocation	Dynamic Memory Allocation
Allocation Time	Compile time (before program execution).	Run time (during program execution).
Memory Segment	Stack Segment / Data Segment.	Heap Segment.
Size Flexibility	Fixed; cannot be resized during execution.	Flexible; can be reallocated (resized) using realloc().
Deallocation	Automatic when variable goes out of scope.	Manual; programmer must explicitly call free() or delete.
Access Speed	Faster access due to direct stack pointer indexing.	Slightly slower due to pointer indirection.
Memory Wastage	High if allocated buffer is larger than needed; overflow if smaller.	Minimal; allocates exact memory required.
Typical Example	int arr[100]; (Fixed static array)	int *ptr = (int*)malloc(n * sizeof(int));

Dynamic Memory Management Functions in C

- **1. malloc(size_t size):** Allocates a contiguous block of uninitialized raw bytes on the heap. Returns void* pointer to the first byte, or NULL if memory is exhausted. Memory contains garbage values.
- **2. calloc(size_t num, size_t size):** Allocates contiguous memory for an array of num elements, each of size bytes. Initializes all allocated bytes to ZERO.
- **3. realloc(void *ptr, size_t new_size):** Resizes an existing dynamically allocated block pointed to by ptr. May expand in-place or allocate a new block, copy data, free old block, and return new pointer.

- **4. free(void *ptr):** Deallocates heap memory block previously returned by malloc/calloc/realloc, returning memory back to the heap manager.

NOTE / EXAM POINTER

Common Dynamic Memory Pitfalls: • Memory Leak: Forgetting to call free() on allocated heap memory, causing available memory to steadily drain. • Dangling Pointer: A pointer pointing to a memory location that has already been deallocated using free(). Solution: Set ptr = NULL immediately after free(ptr). • Double Free: Calling free() twice on the same pointer, leading to heap corruption.

2.2 Linked List as ADT & Dynamic Realization

Definition: Linked List & Node Anatomy

A Linked List is a linear, dynamic data structure consisting of a sequence of dynamically allocated elements called Nodes, where each node contains two components:

1. Data Field: Stores the actual information/value.

2. Next Pointer (Link): Stores the memory address of the succeeding node in the sequence.

ANATOMY OF A SINGLY LINKED LIST NODE & CHAIN

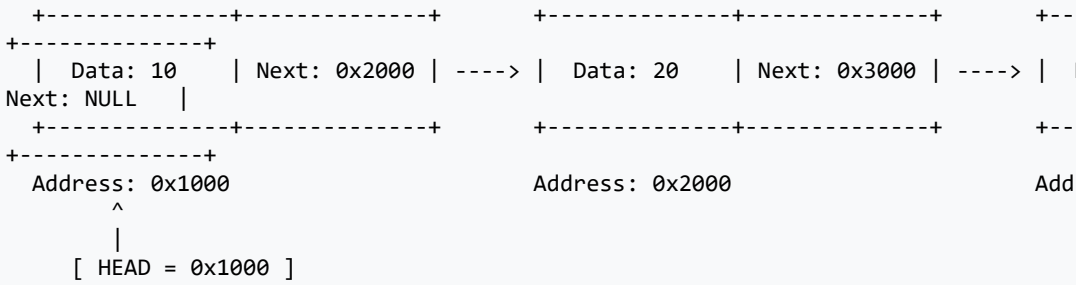

Figure: Self-referential node chaining with Head pointer and terminating NULL pointer

Linked List Abstract Data Type (ADT) Specification

The Linked List ADT defines a mathematical model with data objects and associated primitive operations:

- • **Data:** A sequence of zero or more homogeneous elements <a_1, a_2, ..., a_n>.
- • **Operations:** create(), isEmpty(), length(), traverse(), search(key), insert(pos, val), delete(pos/key), reverse(), concatenate(list2).

C Structure Realization (Self-Referential Structure)

Self-Referential Node Declaration in C

```
struct Node {
   int data;           // Value payload
   struct Node* next;     // Pointer to next node of same type
};
typedef struct Node* NodePtr;
// Creating a new node dynamically:
NodePtr newNode = (NodePtr)malloc(sizeof(struct Node));
newNode->data = value;
newNode->next = NULL;
```

2.3 Types of Linked Lists: Architectures & Comparisons

1. Singly Linked List (SLL)

Each node has a single link pointer pointing to the next node. The last node points to NULL. Traversal is strictly unidirectional (forward only).

2. Circular Singly Linked List (CLL)

The next pointer of the last node points back to the first node (Head), forming a closed ring. No node contains NULL. If maintained with a 'Tail' pointer, insertion at both head (Tail->next) and tail (Tail) takes O(1) time.

3. Doubly Linked List (DLL)

Each node contains two pointers: 'prev' (pointing to predecessor) and 'next' (pointing to successor). The first node's prev is NULL, and last node's next is NULL. Enables bidirectional traversal (forward and backward) and O(1) node deletion if pointer is given.

4. Doubly Circular Linked List (DCLL)

Combines Doubly and Circular list properties: First node's prev points to last node, and last node's next points to first node. Complete circular bidirectional connectivity.

TYPES OF LINKED LIST ARCHITECTURES

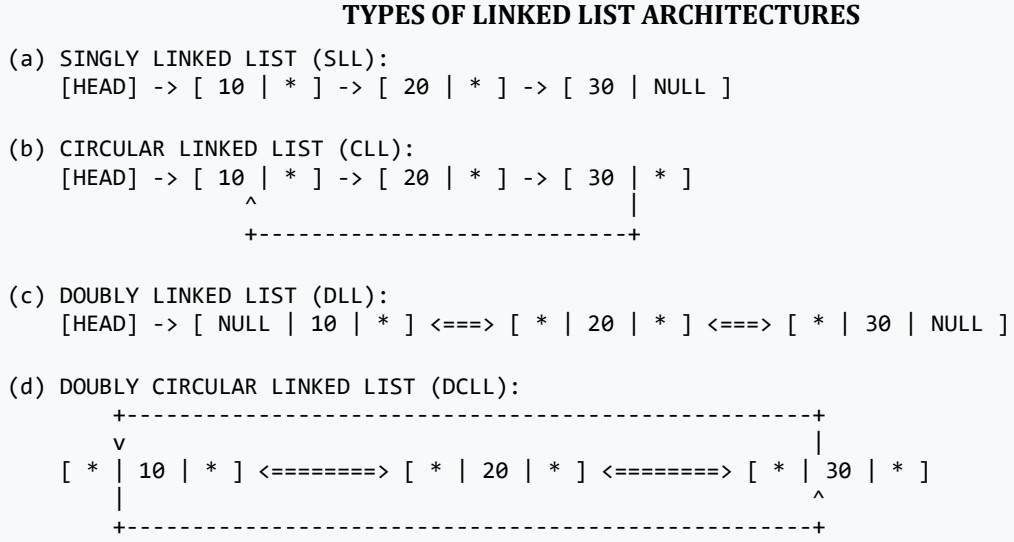

Figure: Architectural schematics of Singly, Circular, Doubly, and Doubly Circular Linked Lists

Comparative Analysis Table of Linked List Variants

List Type	Pointers / Node	Traversal Direction	Insertion at Head	Insertion at Tail	Delete Given Node
Singly Linked (SLL)	1 (next)	Forward only	O(1)	O(n) (or O(1) with tail)	O(n) (needs predecessor)
Circular Singly (CLL)	1 (next)	Forward circular	O(1) (with tail)	O(1) (with tail)	O(n)
Doubly Linked (DLL)	2 (prev, next)	Bidirectional	O(1)	O(n) (or O(1) with tail)	O(1) (prev pointer available)
Doubly Circular (DCLL)	2 (prev, next)	Bidirectional circular	O(1)	O(1)	O(1)

2.4 Primitive Operations on Linked Lists

1. Insertion Operations in Singly Linked List

- **a. Insert at Beginning (Head):** Create newNode. Set newNode->next = head. Update head = newNode. Time: O(1).

- **b. Insert at End (Tail):** Create newNode (next = NULL). If list is empty, head = newNode. Else traverse to last node (temp->next == NULL), set temp->next = newNode. Time: O(n).
- **c. Insert After a Given Node (Key):** Traverse to node with data == key. Set newNode->next = temp->next. Set temp->next = newNode. Time: O(n) search + O(1) insertion.

2. Deletion Operations in Singly Linked List

- **a. Delete from Beginning:** If head == NULL, error underflow. Save temp = head. Update head = head->next. Call free(temp). Time: O(1).
- **b. Delete from End:** If single node, free(head), head = NULL. Else traverse with two pointers (prev and curr) until curr->next == NULL. Set prev->next = NULL, free(curr). Time: O(n).
- **c. Delete Specific Value (Key):** Search for key tracking prev and curr. If found at head, update head = head->next, free(curr). Else prev->next = curr->next, free(curr). Time: O(n).

3. Reversal of Singly Linked List (3-Pointer Iterative Algorithm)

Iterative Reversal Algorithm (In-Place O(n) Time, O(1) Space)

```
struct Node* reverseList(struct Node* head) {
   struct Node *prev = NULL, *curr = head, *next = NULL;
   while (curr != NULL) {
      next = curr->next;   // 1. Store next node
      curr->next = prev;   // 2. Reverse current node pointer
      prev = curr;         // 3. Move prev forward
      curr = next;         // 4. Move curr forward
   }
   return prev;            // New head is prev
}
```

4. Concatenation and Sorting of Linked Lists

- **Concatenation:** Traverse list1 to last node (temp->next == NULL), set temp->next = head2. Time: O(length(list1)).
- **Merge Sort on Linked List:** Find midpoint using slow and fast pointers (O(n)), recursively sort left and right halves, merge two sorted linked lists in O(n) time. Overall time: O(n log n), Space: O(log n) stack. Ideal sorting algorithm for linked lists.

2.5 Polynomial Manipulations Using Linked Lists

A single-variable polynomial $P(x) = c_n x^{en} + c_{n-1} x^{en-1} + ... + c_0 x^0$ is efficiently represented using a singly linked list where each node contains coefficient (float), exponent (int), and next pointer, ordered in descending powers of exponent.

Polynomial Node Structure in C

```
struct PolyNode {
   float coeff;          // Coefficient value (e.g., 5.0)
   int exp;              // Exponent power (e.g., 3 for x^3)
   struct PolyNode* next;   // Link to next term
};
```

Polynomial Addition Algorithm

Let P1 and P2 be two polynomial linked lists. Traverse both lists simultaneously using pointers p1 and p2:

- **Case 1: p1->exp == p2->exp :** Add coefficients: sum = p1->coeff + p2->coeff. If sum != 0, insert term (sum, p1->exp) into result list. Advance both p1 = p1->next and p2 = p2->next.
- **Case 2: p1->exp > p2->exp :** Insert (p1->coeff, p1->exp) into result list. Advance p1 = p1->next.
- **Case 3: p1->exp < p2->exp :** Insert (p2->coeff, p2->exp) into result list. Advance p2 = p2->next.
- **Remaining Terms:** Append remaining nodes of whichever list is non-empty to result list.
- **Time Complexity:** O(m + n), where m and n are the number of terms in P1 and P2.

TRACE OF POLYNOMIAL ADDITION

```
  P1: (5x⁴) -> (3x²) -> (2x⁰) -> NULL
  P2: (4x⁴) -> (6x³) -> (3x²) -> (1x¹) -> NULL

  Step-by-step Execution:
  1. exp(4) == exp(4) ==> coeff = 5+4 = 9  ==> Result: (9x⁴)
  2. exp(P1:2) < exp(P2:3) ==> copy P2    ==> Result: (9x⁴) -> (6x³)
  3. exp(2) == exp(2) ==> coeff = 3+3 = 6  ==> Result: (9x⁴) -> (6x³) -> (6x²)
  4. exp(P1:0) < exp(P2:1) ==> copy P2    ==> Result: (9x⁴) -> (6x³) -> (6x²) -> (1x¹)
  5. P2 empty, append P1 term (2x⁰)        ==> Result: (9x⁴) -> (6x³) -> (6x²) -> (1x¹) ->
(2x⁰)
```

Figure: Step-by-step exponent matching trace during polynomial addition

2.6 Generalized Linked List (GLL) Concept

Definition: Generalized Linked List (GLL)

A Generalized Linked List (GLL) is a finite sequence of zero or more elements <e_1, e_2, ..., e_n> where each element e_i is either:

1. An Atom: A single atomic data item (integer, char, float).

2. A Sublist: Another Generalized Linked List.

- *Head: The first element e_1 of the list (can be an atom or sublist).*
- *Tail: The list of remaining elements <e_2, e_3, ..., e_n> (always a GLL).*
- *Depth of GLL: The maximum level of nested parenthesization.*

Node Representation in GLL

Each node in a GLL utilizes a Tag (Flag) bit to differentiate between an atomic data value and a sublist pointer:

GLL Node Structure Declaration

```
struct GLLNode {
   int tag;              // 0 = Atom node, 1 = Sublist node
   union {
      char data;         // Stored atomic value (when tag == 0)
      struct GLLNode* dlink; // Down pointer to sublist (when tag == 1)
   };
   struct GLLNode* next;    // Next pointer in same level
};
```

GLL VISUALIZATION FOR LIST L = (a, (b, c), d)

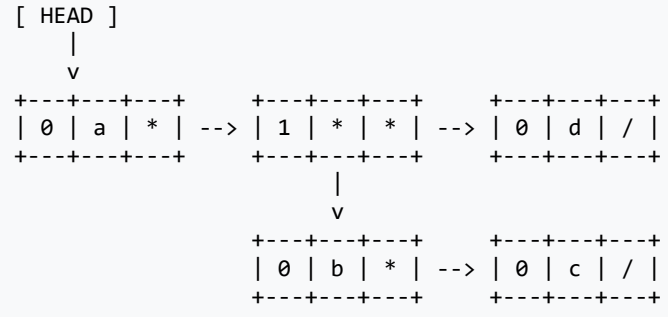

Figure: Box-pointer structural diagram for Generalized Linked List with tag bits and down pointers

Applications & Significance of GLL

- **1. Multi-variable Polynomials & Expressions:** Representation of complex algebraic structures with arbitrary nesting.
- **2. Sparse Matrices & Tensor Representation:** Efficient multidimensional sparse matrix modeling without memory overhead for zero elements.
- **3. LISP Language S-Expressions:** The foundational internal data structure for LISP, Scheme, and symbolic AI programming languages.

2.7 High-Yield University Exam Question Bank

Part A: Short Answer Questions (2 to 5 Marks)

Q1. What is the difference between malloc() and calloc()? [3 Marks]

- **Answer:** 1. Arguments: malloc(size) takes 1 argument (total bytes); calloc(num, size) takes 2 arguments (number of elements and size of each). 2. Initialization: malloc() allocates uninitialized memory containing garbage values; calloc() allocates and initializes every byte to ZERO.

Q2. Define Dangling Pointer and Memory Leak with prevention measures. [4 Marks]

- **Answer:** A Dangling Pointer points to a freed/deallocated memory address. Prevent by setting ptr = NULL after free(ptr). A Memory Leak occurs when dynamically allocated heap memory is no longer referenced but never freed with free(), slowly exhausting available RAM. Prevent by ensuring every malloc() has a corresponding free().

Q3. State the advantage of Circular Linked List with Tail pointer over standard Singly Linked List. [3 Marks]

- **Answer:** In a standard SLL, insertion at tail takes O(n) time unless an extra tail pointer is kept. In a Circular Linked List with Tail pointer, Tail->next gives Head. Thus, insertion at both Beginning (Head = Tail->next) and End (Tail) can be executed in O(1) constant time without traversal.

Q4. Define Head, Tail, and Depth of a Generalized Linked List (GLL). [3 Marks]

- **Answer:** For a GLL L = (e_1, e_2, ..., e_n): • Head is the first element e_1 (atom or sublist). • Tail is the list of remaining elements (e_2, ..., e_n) (always a GLL). • Depth is the maximum nesting level of sublists. Example: Depth of (a, (b, (c))) is 3.

Part B: Long Answer Questions (10 to 15 Marks Outline Guides)

- **Q5. Write complete C functions for Singly Linked List: (a) Insert at position, (b) Delete key, (c) Reverse list iteratively. Analyze time complexity of each. [12 Marks]**
- **Q6. Explain Doubly Linked List. Write algorithms for inserting a node after a given node and deleting an arbitrary node in DLL. Discuss advantages of DLL over SLL. [12 Marks]**
- **Q7. Write an algorithm to perform Polynomial Addition using Singly Linked Lists. Trace the algorithm with P1 = $7x^3 + 4x^2 + 2$ and P2 = $5x^4 + 3x^3 + 2x + 1$. [12 Marks]**
- **Q8. Explain Generalized Linked List (GLL) with node structure and tag representation. Draw box-pointer diagrams for: (a) L_1 = (a, b, (c, d), e), (b) L_2 = ((p, q), (r, (s, t))). Compute Head, Tail, and Depth for both. [14 Marks]**

2.8 Unit Summary & Key Takeaways

- Static allocation happens on Stack at compile time with fixed size; Dynamic allocation happens on Heap at runtime via malloc/calloc/realloc/free.
- Linked List is a dynamic linear structure composed of self-referential nodes connected via memory pointers.
- Types include Singly Linked List (forward only), Circular Linked List (looping back to head), Doubly Linked List (bidirectional prev/next), and Doubly Circular Linked List.
- Primitive operations (insert, delete, traverse, search, reverse, sort, concatenate) manipulate pointers in O(1) for head/tail and O(n) for arbitrary positions.
- Polynomial manipulation represents terms as (coeff, exp, next) ordered descending by exponent, adding terms in linear O(m+n) time.
- Generalized Linked Lists (GLL) handle multi-level nested data and sublists using tag bits, enabling hierarchical representation in LISP, matrices, and symbolic algebra.