

UNIT 5

Linked Lists

(5 Hours) — Data Structures and Algorithms

This unit introduces the linked list, a fundamental dynamic data structure that overcomes the size and memory-shifting limitations of arrays. We study its abstract definition, how it is implemented dynamically using pointers, the algorithms for inserting and deleting nodes at various positions, and two important applications — linked stacks and linked queues. The unit closes with doubly linked lists and the advantages they provide over singly linked lists.

5.1 Introduction to Linked Lists

An array stores a fixed number of elements in contiguous memory locations. This makes array size static, and inserting or deleting an element in the middle requires shifting many other elements — an expensive operation. A linked list solves both problems. It is a linear collection of elements, called nodes, where each node is stored at a possibly non-contiguous memory location and is logically connected to the next node using a pointer (link).

Because nodes are linked by address rather than by physical adjacency, a linked list can grow or shrink at run time, and insertion or deletion requires only a constant number of pointer changes — no shifting of other elements is needed.

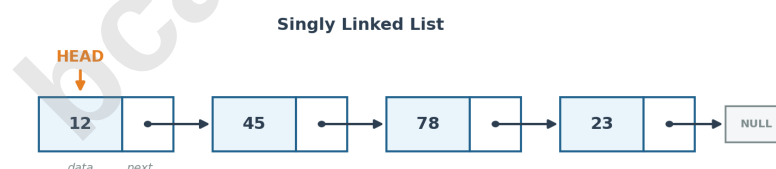


Fig 5.1 — A singly linked list of four nodes, terminated by NULL and accessed through the HEAD pointer.

5.1.1 Why Linked Lists? (Array vs Linked List)

Property	Array	Linked List
Memory allocation	Contiguous, fixed at compile/run time	Non-contiguous, allocated dynamically
Size	Fixed (static)	Grows / shrinks at run time (dynamic)
Access	Direct / random access — $O(1)$	Sequential access only — $O(n)$
Insertion / Deletion	Costly — needs shifting of elements	Efficient — only pointers are updated

Property	Array	Linked List
Memory utilisation	May waste memory if under-filled	Uses exactly as much as needed, plus pointer overhead
Extra memory	None required	Extra memory needed for storing pointers

5.1.2 Types of Linked Lists

- Singly Linked List — each node points only to the next node; traversal is possible in one direction only.
- Doubly Linked List — each node has two pointers, one to the next node and one to the previous node; traversal is possible in both directions.
- Circular Linked List — the last node points back to the first node instead of NULL, forming a ring.
- Circular Doubly Linked List — combines the doubly linked and circular properties.

Note: Unless stated otherwise, “linked list” in this unit refers to the singly linked list; doubly linked lists are covered separately in Section 5.7.

5.2 Linked List as an Abstract Data Type (ADT)

An Abstract Data Type defines a data structure purely in terms of the operations that can be performed on it, without specifying how those operations are implemented. The Linked List ADT can be described as follows:

5.2.1 Structure of a Node

Every node of a linked list has exactly two parts:

- Data field — stores the actual value / information.
- Next (link/pointer) field — stores the address of the next node in the list; the last node stores NULL.

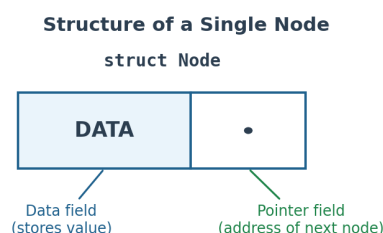


Fig 5.2 — A single node has a data field and a pointer field.

```

struct Node {
    int data;
    struct Node *next;
};
typedef struct Node *NODEPTR;
NODEPTR HEAD = NULL; /* points to the first node of the list */

```

5.2.2 ADT Operations

Operation	Description
create()	Creates an empty linked list
isEmpty()	Returns true if the list has no nodes (HEAD == NULL)
insert(pos, item)	Inserts a new node containing item at the given position
delete(pos)	Removes the node at the given position
search(item)	Traverses the list to find a node containing item
traverse()	Visits every node exactly once, from HEAD to the last node
display()	Prints the data of every node in the list
length()	Counts and returns the total number of nodes

Note: An ADT only specifies what operations are available and what they do — the actual C/C++ code that implements them (using pointers and dynamic memory) is the concrete data structure, discussed next.

5.3 Dynamic Implementation of Linked Lists

A linked list is called a dynamic data structure because its nodes are created and destroyed at run time using dynamic memory allocation, rather than being declared in advance like an array. In C, this is done with malloc() (to create a node) and free() (to destroy a node); in C++, the new and delete operators are used instead.

5.3.1 Creating a New Node

```

NODEPTR getnode() {
    NODEPTR temp;
    temp = (NODEPTR) malloc(sizeof(struct Node));
    return temp;
}

```

The function above allocates memory for exactly one node on the heap and returns a pointer to it. The calling routine then fills in the data field and adjusts the next field to splice the node into

the list.

5.3.2 Freeing a Node

```
void freenode(NODEPTR p) {  
    free(p);  
}
```

Whenever a node is removed from a list, its memory should be released back to the system using `free()`; failing to do so causes a memory leak.

5.3.3 Traversing a Linked List

Traversal means visiting every node of the list exactly once, usually to display, search, or process the data. A temporary pointer is used so that HEAD itself is never disturbed.

```
void display(NODEPTR head) {  
    NODEPTR p = head;  
    while (p != NULL) {  
        printf("%d -> ", p->data);  
        p = p->next;  
    }  
    printf("NULL\n");  
}
```

Example 5.1 — Counting nodes in a linked list

Write a function that returns the number of nodes present in a linked list.

Solution

```
int length(NODEPTR head) {  
    int count = 0;  
    NODEPTR p = head;  
    while (p != NULL) {  
        count++;  
        p = p->next;  
    }  
    return count;  
}
```

5.4 Insertion of a Node Into a Linked List

Inserting a node involves two steps common to every case: (1) obtain a new node using `getnode()` and store the data in it, and (2) adjust pointers so the new node is correctly linked into the list. There are three standard cases.

5.4.1 Insertion at the Beginning

The new node becomes the first node of the list. Its next field is made to point to the old first node (the old HEAD), and HEAD is then updated to point to the new node.

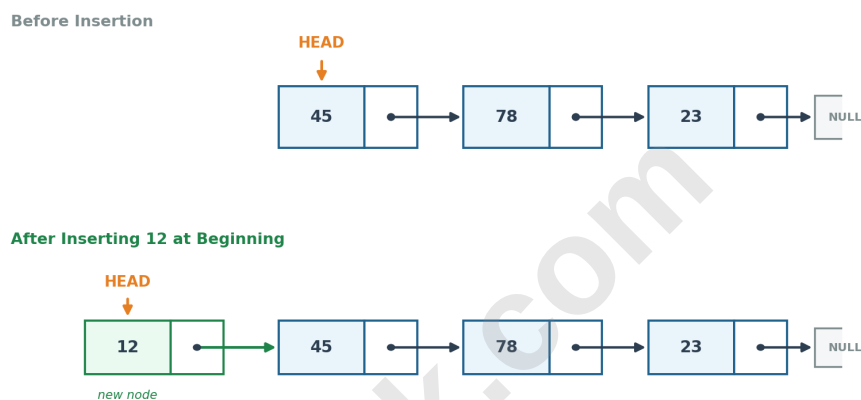


Fig 5.3 — Inserting a new node (12) at the beginning of the list.

```
NODEPTR insertAtBeginning(NODEPTR head, int val) {  
    NODEPTR temp = getnode();  
    temp->data = val;  
    temp->next = head; /* new node points to old first node */  
    head = temp; /* HEAD now points to new node */  
    return head;  
}
```

Note: Insertion at the beginning always runs in $O(1)$ time, because no traversal of the list is required.

5.4.2 Insertion at the End

The list is traversed until the last node (the one whose next field is NULL) is reached. The last node's next field is then made to point to the new node, whose own next field is set to NULL.

```

NODEPTR insertAtEnd(NODEPTR head, int val) {
    NODEPTR temp = getnode();
    temp->data = val;
    temp->next = NULL;
    if (head == NULL) return temp; /* list was empty */
    NODEPTR p = head;
    while (p->next != NULL)
        p = p->next; /* move to last node */
    p->next = temp;
    return head;
}

```

Note: Insertion at the end runs in $O(n)$ time for a singly linked list, since the whole list must be traversed to reach the last node (unless a separate TAIL pointer is maintained).

5.4.3 Insertion at a Given Position

To insert at a specific position k , the list is traversed to the node just before position k . The new node's next field is set to point to the node currently at position k , and the preceding node's next field is updated to point to the new node.

```

NODEPTR insertAtPos(NODEPTR head, int val, int pos) {
    if (pos == 1) return insertAtBeginning(head, val);
    NODEPTR temp = getnode();
    temp->data = val;
    NODEPTR p = head;
    for (int i = 1; i < pos - 1 && p != NULL; i++)
        p = p->next; /* stop at node (pos-1) */
    temp->next = p->next;
    p->next = temp;
    return head;
}

```

5.5 Deletion of a Node From a Linked List

Deletion removes a node from the list and frees the memory it occupied. As with insertion, there are three standard cases, and each requires updating the pointer of the preceding node (or HEAD) to bypass the deleted node.

5.5.1 Deletion of the First Node

```

NODEPTR deleteFirst(NODEPTR head) {
    if (head == NULL) return NULL; /* list already empty */
    NODEPTR temp = head;
    head = head->next;
    free(temp);
    return head;
}

```

5.5.2 Deletion of the Last Node

```

NODEPTR deleteLast(NODEPTR head) {
    if (head == NULL) return NULL;
    if (head->next == NULL) { free(head); return NULL; }
    NODEPTR p = head;
    while (p->next->next != NULL)
        p = p->next; /* stop one node before the last */
    free(p->next);
    p->next = NULL;
    return head;
}

```

5.5.3 Deletion of an Intermediate (Middle) Node

A pointer *p* is used to track the node just before the one to be deleted. Once found, *p*'s next field is redirected to skip over the deleted node, and the deleted node's memory is freed.

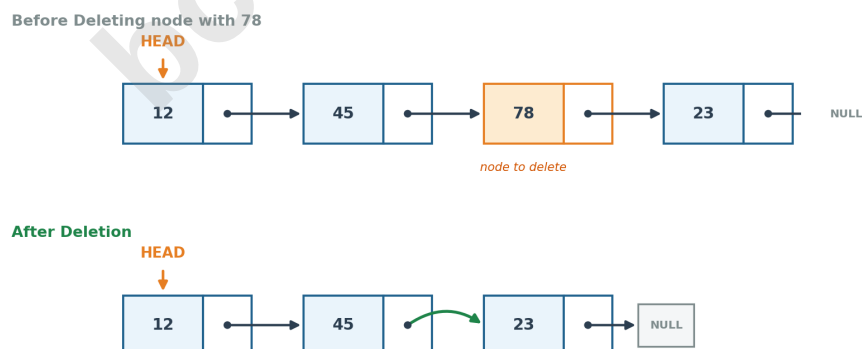


Fig 5.4 — Deleting the node containing 78 from the middle of the list.

```

NODEPTR deleteByValue(NODEPTR head, int val) {
    NODEPTR p = head, prev = NULL;
    while (p != NULL && p->data != val) {
        prev = p;
        p = p->next;
    }
    if (p == NULL) return head; /* value not found */
    if (prev == NULL) head = p->next; /* deleting first node */
    else prev->next = p->next; /* bypass p */
    free(p);
    return head;
}

```

Example 5.2

Given the list $12 \rightarrow 45 \rightarrow 78 \rightarrow 23 \rightarrow \text{NULL}$, trace `deleteByValue(head, 78)`.

Solution

p starts at node 12 (`prev = NULL`); $12 \neq 78$, so `prev = node(12)`, `p = node(45)`.

$45 \neq 78$, so `prev = node(45)`, `p = node(78)`.

$78 == 78$ — loop stops. `prev` is not `NULL`, so `prev->next = p->next`, i.e. `node(45)->next` now points to `node(23)`.

`free(p)` releases `node(78)`. Resulting list: $12 \rightarrow 45 \rightarrow 23 \rightarrow \text{NULL}$.

5.6 Insertion and Deletion After / Before a Given Node

In many applications the position of insertion or deletion is specified relative to an existing node rather than by numeric index — e.g. “insert 60 after the node containing 45”, or “insert 60 before the node containing 78”. Because a singly linked list can only be traversed forward, these two cases require slightly different logic.

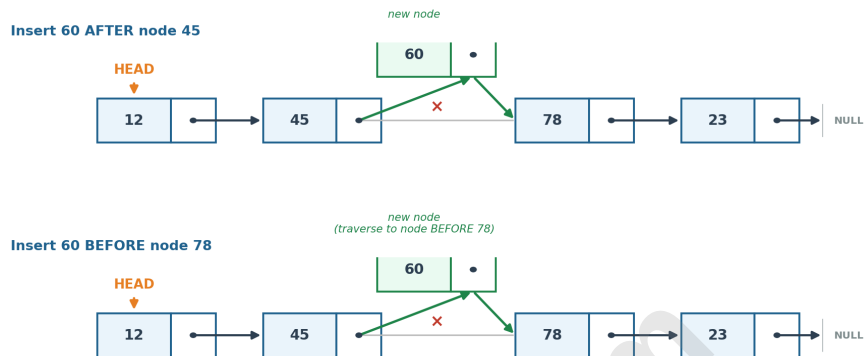


Fig 5.5 — Inserting a node after a given node (top) versus before a given node (bottom). The old link (×) is replaced by two new links (green).

5.6.1 Insertion After a Given Node

This is the simpler case: once a pointer *p* to the given node is obtained (by searching), the new node is spliced in directly after it — no need to know the node that precedes *p*.

```
void insertAfter(NODEPTR p, int val) {  
    if (p == NULL) return;  
    NODEPTR temp = getnode();  
    temp->data = val;  
    temp->next = p->next; /* new node points to what p pointed to */  
    p->next = temp; /* p now points to new node */  
}
```

5.6.2 Insertion Before a Given Node

Since a singly linked list has no “previous” pointer, inserting before a given node *q* requires the list to be traversed from HEAD to find the node that immediately precedes *q* — call it *p*. The new node is then inserted after *p* (which is the same as before *q*).

```

NODEPTR insertBefore(NODEPTR head, NODEPTR q, int val) {
    if (head == q) return insertAtBeginning(head, val);
    NODEPTR p = head;
    while (p != NULL && p->next != q)
        p = p->next; /* find node just before q */
    if (p == NULL) return head; /* q not found in list */
    NODEPTR temp = getnode();
    temp->data = val;
    temp->next = q;
    p->next = temp;
    return head;
}

```

5.6.3 Deletion After / Before a Given Node

Deletion follows the same reasoning. To delete the node after a given pointer *p*, simply bypass *p->next* directly. To delete the node before a given node *q*, the list must again be traversed from HEAD to locate the node two positions before *q*.

```

void deleteAfter(NODEPTR p) {
    if (p == NULL || p->next == NULL) return;
    NODEPTR temp = p->next;
    p->next = temp->next; /* bypass the node after p */
    free(temp);
}

```

Operation	Needs traversal from HEAD?	Time Complexity
Insert / Delete at beginning	No	O(1)
Insert / Delete after a known node <i>p</i>	No (<i>p</i> already known)	O(1)
Insert / Delete before a known node <i>q</i>	Yes — to find node before <i>q</i>	O(<i>n</i>)
Insert / Delete at the end	Yes — to find the last node	O(<i>n</i>)
Insert / Delete at position <i>k</i>	Yes — up to <i>k</i> nodes	O(<i>k</i>)

5.7 Linked Stacks and Queues

Stacks and queues can be implemented either using arrays or using linked lists. The linked (dynamic) implementation removes the fixed-size limitation of an array-based stack/queue — the structure grows and shrinks exactly as needed, limited only by available memory, and there is no risk of “stack overflow” due to a fixed array bound.

5.7.1 Linked Stack

A linked stack is a linked list in which insertions and deletions happen only at one end, called TOP. This makes the stack follow the LIFO (Last-In-First-Out) principle. Using the beginning of the list as TOP is essential, since it is the only position where insertion/deletion can be done in $O(1)$ time without traversal.

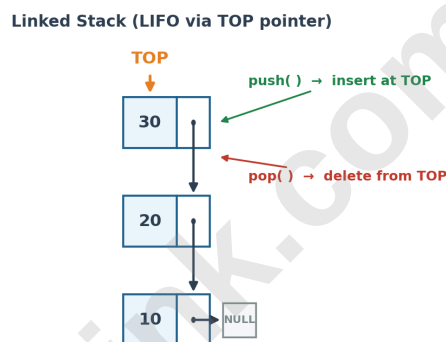


Fig 5.6 — A linked stack: *push()* inserts and *pop()* deletes at TOP.

```
struct Node { int data; struct Node *next; };
typedef struct Node *NODEPTR;
NODEPTR TOP = NULL;
void push(int val) { /* insert at TOP -- same as insertAtBeginning */
    NODEPTR temp = getnode();
    temp->data = val;
    temp->next = TOP;
    TOP = temp;
}
int pop() { /* delete from TOP -- same as deleteFirst */
    if (TOP == NULL) { printf("Stack Underflow"); return -1; }
    NODEPTR temp = TOP;
    int val = temp->data;
    TOP = TOP->next;
    free(temp);
    return val;
}
```

Note: push() and pop() on a linked stack both run in $O(1)$ time and never need to check for a full-stack condition — only the underflow (empty-stack) condition.

5.7.2 Linked Queue

A linked queue is a linked list in which insertion happens at one end (REAR) and deletion happens at the other end (FRONT), giving the FIFO (First-In-First-Out) behaviour. Two pointers are maintained: FRONT (pointing to the first node) and REAR (pointing to the last node), so that both enqueue and dequeue run in $O(1)$ time without any traversal.



Fig 5.7 — A linked queue: enqueue() inserts at REAR, dequeue() deletes at FRONT.

```

NODEPTR FRONT = NULL, REAR = NULL;
void enqueue(int val) { /* insert at REAR */
    NODEPTR temp = getnode();
    temp->data = val;
    temp->next = NULL;
    if (REAR == NULL) { FRONT = REAR = temp; return; }
    REAR->next = temp;
    REAR = temp;
}
int dequeue() { /* delete from FRONT */
    if (FRONT == NULL) { printf("Queue Underflow"); return -1; }
    NODEPTR temp = FRONT;
    int val = temp->data;
    FRONT = FRONT->next;
    if (FRONT == NULL) REAR = NULL; /* queue became empty */
    free(temp);
    return val;
}
  
```

Feature	Linked Stack	Linked Queue
Pointer(s) maintained	TOP only	FRONT and REAR
Insertion point	TOP (beginning)	REAR (end)
Deletion point	TOP (beginning)	FRONT (beginning)
Discipline	LIFO	FIFO
Insert/Delete time complexity	$O(1)$	$O(1)$

5.8 Doubly Linked Lists and Their Advantages

A doubly linked list (DLL) is a linked list in which every node has two pointer fields instead of one:

- prev — address of the previous node (NULL for the first node)
- data — the value stored in the node
- next — address of the next node (NULL for the last node)

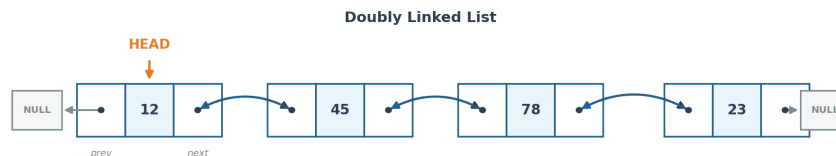


Fig 5.8 — A doubly linked list: every node links both forward (next) and backward (prev).

```
struct DNode {
    struct DNode *prev;
    int data;
    struct DNode *next;
};
```

5.8.1 Insertion at the Beginning of a DLL

```
DNODEPTR insertAtBeginning(DNODEPTR head, int val) {
    DNODEPTR temp = getnode();
    temp->data = val;
    temp->prev = NULL;
    temp->next = head;
    if (head != NULL) head->prev = temp; /* back-link old first node */
    head = temp;
    return head;
}
```

5.8.2 Deletion of a Given Node in a DLL

Because each node already knows its predecessor, a given node can be deleted in $O(1)$ time once a pointer to it is known — there is no need to traverse from HEAD to find the previous node, unlike in a singly linked list.

```
void deleteNode(DNODEPTR *head, DNODEPTR p) {
    if (p->prev != NULL) p->prev->next = p->next;
    else *head = p->next; /* p was the first node */
    if (p->next != NULL) p->next->prev = p->prev;
    free(p);
}
```

5.8.3 Advantages of Doubly Linked Lists

- Bidirectional traversal — the list can be traversed both forward (using next) and backward (using prev), which a singly linked list cannot do.
- Efficient deletion — given only a pointer to the node to be deleted, it can be removed in $O(1)$ time, since its predecessor is directly available via prev; a singly linked list needs an $O(n)$ search for the predecessor.
- Efficient insertion before a given node — inserting before a known node q takes $O(1)$ time (via $q->prev$), instead of the $O(n)$ traversal a singly linked list requires.
- Easier implementation of certain algorithms — e.g. the LRU cache, text editor undo/redo, and browser history are naturally implemented using a DLL.

5.8.4 Disadvantages of Doubly Linked Lists

- Each node requires extra memory for the additional prev pointer.
- Insertion and deletion are slightly more expensive per node, because two pointers must be updated on each side instead of one.
- The implementation is marginally more complex and more error-prone (an incorrect prev update is a common source of bugs).

Feature	Singly Linked List	Doubly Linked List
Pointers per node	1 (next)	2 (prev, next)
Traversal direction	Forward only	Forward and backward
Delete given node p ($O(\cdot)$)	$O(n)$ — must find predecessor	$O(1)$ — predecessor known via $p->prev$
Extra memory per node	Lower	Higher
Typical use	Simple lists, stacks, queues	LRU cache, undo/redo, browser history

Key Takeaways

Unit 5 — Key Takeaways

- ✓ A linked list is a dynamic linear structure of nodes connected by pointers; unlike an array it grows/shrinks at run time and needs no element shifting for insertion/deletion.
- ✓ Each node has a data field and a next (pointer) field; the list is accessed via HEAD and terminated by NULL.
- ✓ Nodes are created and destroyed dynamically at run time using malloc()/free() (or new/delete in C++).
- ✓ Insertion and deletion each have three standard cases: at the beginning, at the end, and at a given position — each requiring careful pointer updates.
- ✓ Inserting/deleting after a known node is $O(1)$; inserting/deleting before a known node needs an $O(n)$ traversal from HEAD in a singly linked list.
- ✓ A linked stack uses a single TOP pointer (LIFO); a linked queue uses FRONT and REAR pointers (FIFO) — both give $O(1)$ insert/delete without a fixed size limit.
- ✓ A doubly linked list adds a prev pointer to every node, enabling bidirectional traversal and $O(1)$ deletion/insertion given a node pointer, at the cost of extra memory per node.

— End of Unit 5 —