Data Structures

Doubly and circular linked lists

The notes for the lesson “Doubly and circular linked lists”: chapters 2.5 to 2.6, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

2 chapters | Chapters 2.5 to 2.6

About these notes

These are the notes for the University free n easy video lesson **Doubly and circular linked lists** on Data Structures: chapters 2.5 to 2.6, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

2.5 Doubly linked lists

A doubly linked list gives every node two pointers, one to the next node and one to the previous. You can walk the list in both directions, and, given a pointer to any node, you can delete it or insert beside it in O(1) without a second pointer trailing behind. The cost is one more pointer per node and twice as many links to update. Browser back/forward history, undo/redo stacks, music playlists and the deque of a later lesson are all doubly linked lists underneath.

> You need: singly linked lists and the two-step insertion rule from chapter 2.4.

The node and its invariant

```
typedef struct DNode {
    struct DNode *prev;
    int data;
    struct DNode *next;
} DNode;
DNode *head = NULL;      /* first node's prev is NULL; last node's next is NULL */
```

The structure keeps one rule at all times: for every pair of neighbouring nodes a and b, `a->next == b` and `b->prev == a`. Every operation below is just "re-establish that rule for the nodes

around the change". If you can state the rule, you can derive the code in the exam instead of memorising it.

Traversal forward is the same loop as chapter 2.4. Traversal backward starts from the last node and follows `prev`; with a `tail` pointer that start is O(1).

Insertion after a node

To insert new node p after node q (q not the last node), four pointers change. The order matters only in that p's pointers are set before q's and q->next's are changed:

1. `p->next = q->next;` p looks forward at q's old successor.
2. `p->prev = q;` p looks back at q.
3. `q->next->prev = p;` the old successor looks back at p (skip this if q was last; set tail = p instead).
4. `q->next = p;` q looks forward at p. Do this last, because step 3 still needs q->next to mean the old successor.

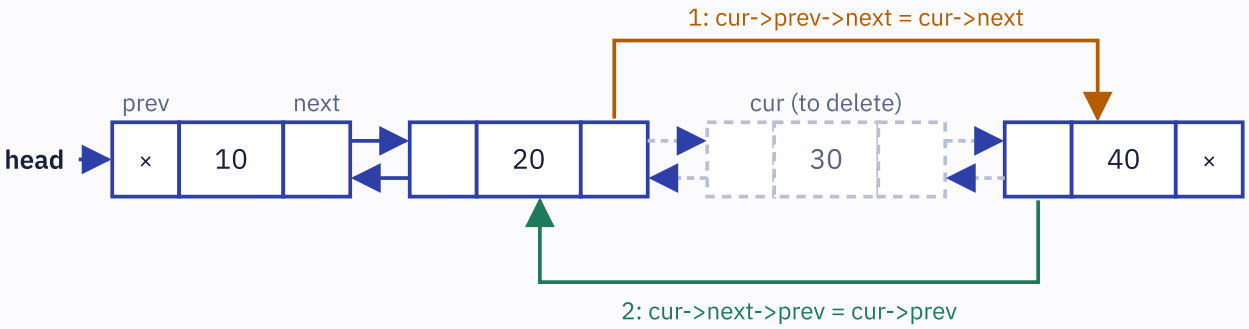
Insertion at the front: `p->next = head; p->prev = NULL; if (head) head->prev = p; head = p;` Insertion before a node is insertion after its `prev`, which is the operation a singly linked list cannot do in O(1).

Deletion of a node

Given a pointer `cur` to the node to remove, two links bypass it:

```
void deleteNode(DNode **head, DNode *cur) {
    if (cur->prev != NULL) cur->prev->next = cur->next;   /* predecessor skips cur */
    else *head = cur->next;                                /* cur was the first node */
    if (cur->next != NULL) cur->next->prev = cur->prev;   /* successor skips cur */
    free(cur);
}
```

No search for the previous node is needed; cur carries it. That is the whole advantage over the singly linked version, where deletion by node pointer is O(n). The two `if`s handle the first and last nodes; in a list with a dummy header and trailer node they disappear, which is why many libraries use sentinels.



× means NULL; dashed links are the ones that disappear when cur is freed

Figure 2.5.1 Deleting the node holding 30. Link 1 (orange) makes 20's next skip to 40; link 2 (green) makes 40's prev skip back to 20. The four dashed links belonged to cur and die with free(cur). Nothing was searched: cur already knew both neighbours.

Cost and applications

Insert and delete beside a known node: O(1). Reaching a node by position or value: O(n), as for any linked list, but a walk can start from whichever end is nearer when the position is known. Memory: three fields per node, typically 24 bytes for an int payload on a 64-bit machine, against 16 for a singly linked node. When a syllabus lists "doubly linked lists and their applications", the usual answers are undo/redo (the current state sits in the middle, undo moves prev, redo moves next, a new action deletes everything after the cursor), browser history, the LRU cache (a hash table pointing into a doubly linked list so that any page can be moved to the front in O(1)), and the deque of a later lesson.

WORKED EXAMPLE

Question. The list is 10 ⇄ 20 ⇄ 40 with head at 10 and a tail pointer at 40. (a) Insert 30 after 20, showing the four assignments and the node fields after each. (b) Then delete the tail node 40. (c) Print the list backward.

Solution (a). q = node 20, p = makeNode(30) with prev = next = NULL. Fields written as (prev, data, next), naming nodes by their data.

STEP	STATEMENT	NODE 20	NODE 30 (P)	NODE 40
0	start	`(10, 20, 40)`	`(×, 30, ×)`	`(20, 40, ×)`
1	p→next = q→next	`(10, 20, 40)`	`(×, 30, 40)`	`(20, 40, ×)`
2	p→prev = q	`(10, 20, 40)`	`(20, 30, 40)`	`(20, 40, ×)`
3	q→next→prev = p	`(10, 20, 40)`	`(20, 30, 40)`	`(30, 40, ×)`
4	q→next = p	`(10, 20, 30)`	`(20, 30, 40)`	`(30, 40, ×)`

List: 10 ⇄ 20 ⇄ 30 ⇄ 40. Check the invariant: 20→next is 30 and 30→prev is 20; 30→next is 40 and 40→prev is 30.

Solution (b). cur = tail = node 40. cur→prev is node 30, not NULL, so `node30->next = cur->next = NULL`. cur→next is NULL, so the second update is skipped and instead `tail = cur->prev = node 30`. free(node 40). List: 10 ⇄ 20 ⇄ 30, tail = 30. One link changed; no walk.

Solution (c). `for (DNode *p = tail; p != NULL; p = p->prev) printf("%d ", p->data);` prints `30 20 10`.

EXAM TRAPS

- Doing `q->next = p` before `q->next->prev = p`: the second line then sets p's own prev, and the old successor keeps pointing back at q. Write the four lines in the order given.
- Forgetting the prev pointer of the node after the insertion or deletion; marks go for updating both directions.
- Dereferencing NULL at the ends: `cur->next->prev` crashes when cur is the last node. Guard with if, or use sentinel nodes.
- Counting pointer updates wrongly: an insertion in the middle changes 4 pointers, a deletion changes 2 (plus head or tail at the ends).
- Favourite question: "Compare singly and doubly linked lists" with a table (memory per node, backward traversal, deletion given a node pointer, insertion before a node, code complexity).

CHECK YOURSELF

– REMEMBER **Name the three fields of a doubly linked list node and the values of the end nodes' outer pointers.** 2 marks

prev, data, next; the first node's prev and the last node's next are NULL.

– UNDERSTAND **Why can a doubly linked list delete a node in O(1) given its pointer while a singly linked list cannot?** 3 marks

Unlinking needs the predecessor's next pointer changed. The doubly linked node stores its predecessor (prev); the singly linked node does not, so the predecessor must be found by walking from the head, O(n).

– UNDERSTAND **What invariant do all doubly linked list operations preserve?** 2 marks

For neighbours a and b: a→next == b exactly when b→prev == a.

– APPLY **Write the statements to insert node p at the front of a non-empty doubly linked list.** 3 marks

```
p->next = head; p->prev = NULL; head->prev = p; head = p;
```

– APPLY **The list is 5 ⇄ 8 ⇄ 12 ⇄ 20. Delete 8 and state which pointers change and their new values.** 3 marks

node5→next = node 12; node12→prev = node 5; free(node 8). Result 5 ⇄ 12 ⇄ 20. Two pointer updates.

– ANALYZE **Design the data structure for an editor's undo/redo and explain each operation's cost.** 4 marks

A doubly linked list of states with a cursor pointer. Undo: cursor = cursor→prev, O(1). Redo: cursor = cursor→next, O(1). New edit: free every node after the cursor (O(k) for k redo states), insert the new state after the cursor, O(1). Singly linked would make undo O(n).

BEYOND THE SYLLABUS

Not for the exam. Java's `LinkedList` and C++'s `std::list` are doubly linked; Python's `collections.OrderedDict` and the standard LRU cache recipe pair a hash map with a doubly linked list. The XOR linked list stores prev XOR next in one field to save memory, a trick worth reading once. Search "LRU cache doubly linked list" and "XOR linked list".

2.6 Circular linked lists

In a circular linked list the last node points back to the first instead of to NULL, so the list has no end: from any node you can reach every other node by walking forward. Keep a single pointer to the *last* node and both ends of the list are one step away, which gives O(1) insertion at the front and at the back with one pointer. Round-robin CPU scheduling, players taking turns in a game, the circular queue of a later lesson and the Josephus problem all live naturally on this structure.

You need: singly linked list insertion and deletion from chapter 2.4.

Structure and the tail pointer

The node is the same as in chapter 2.4: data and next. What changes is the convention: the last node's `next` holds the address of the first node, and NULL appears nowhere except when the list is empty. Textbooks draw the list with a `head` pointer, but the useful choice is a pointer to the last node, usually called `tail` (some syllabi call the structure a "circularly linked list"). Then:

- `tail` is the last node, reached in O(1).
- `tail->next` is the first node, also O(1).
- Empty list: `tail == NULL`. One node: `tail->next == tail`.

Some syllabi list "linked lists with header node": a circular list with a permanent dummy node has no empty case at all (the header points to itself), which simplifies every routine at the cost of one node.

Traversal: why a do-while

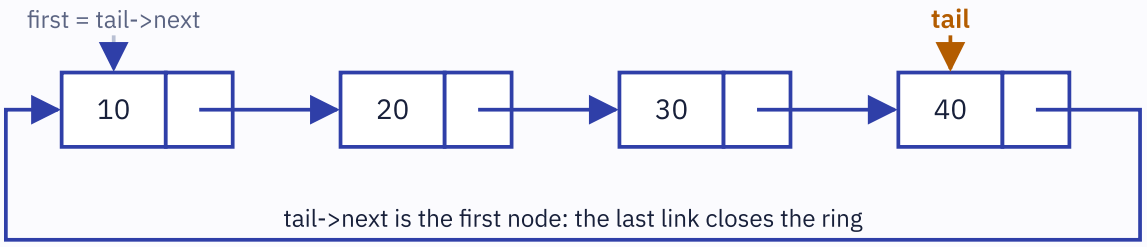
The `while (p != NULL)` loop of chapter 2.4 never ends on a circular list. Start at the first node, process it, and stop when you are back where you started. Because the test is at the end, a `do … while` visits a one-node list correctly.

```
void traverse(Node *tail) {
    if (tail == NULL) return;              /* empty */
    Node *p = tail->next;                  /* first node */
    do { printf("%d -> ", p->data); p = p->next; } while (p != tail->next);
    printf("(back to %d)\n", tail->next->data);
}
```

Insertion and deletion with a tail pointer

1. **Insert at front**, O(1): if empty, `p->next = p; tail = p;` else `p->next = tail->next; tail->next = p;`.
2. **Insert at end**, O(1): same two statements as the front, then `tail = p;`. The node becomes the new last node and the old first node stays first.
3. **Delete the front**, O(1): `Node *f = tail->next;` if `f == tail` the list had one node, so `tail = NULL`; else `tail->next = f->next;`. Then `free(f)`.
4. **Delete the last**, O(n): you need the node before tail, so walk from `tail->next` until `q->next == tail`, then `q->next = tail->next; free(tail); tail = q;`. A circular *doubly* linked list makes this O(1) too.

Insertion after a given node and deletion by value are the same two-pointer walks as chapter 2.4, with the loop stopping when it returns to the first node.



Insert p at front: p->next = tail->next; tail->next = p;
Insert p at end: the same two statements, then tail = p;
Traverse: p = tail->next; do { visit p; p = p->next; } while (p != tail->next);

Figure 2.6.1 A circular singly linked list with the one pointer that matters, tail. The last node's next closes the ring back to 10, so no node holds NULL. Both insertions use the same two statements; insertion at the end just moves tail afterwards. Traversal stops when the walk returns to tail→next.

Applications

Round-robin scheduling: processes form a ring; the scheduler runs the current one for a time slice and moves to `next`, forever, with finished processes deleted in O(1) if the scheduler holds the previous pointer. **Josephus problem**: n people in a circle, every k-th is removed until one remains; a circular list models it directly (worked example). **Circular buffers and queues**: A later lesson implements the circular queue on an array, but the linked version with a tail pointer is the same idea. **Multiplayer turns and image carousels**: "next" from the last item is the first.

WORKED EXAMPLE

Question. Seven people numbered 1 to 7 stand in a circle. Counting starts at person 1; every third person is removed (count 1, 2, 3 → the third leaves) and counting restarts at the next person. Using a circular linked list, show the order of removal and the survivor.

Solution. Build the ring 1 → 2 → 3 → 4 → 5 → 6 → 7 → back to 1. Keep a pointer `prev` at the node before the one being counted (start prev = node 7, so the first count begins at node 1). Each round: advance prev by k − 1 = 2 steps, then delete `prev->next` and leave prev where it is, so counting restarts at the next survivor.

ROUND	RING BEFORE (COUNT STARTS AT FIRST)	COUNTED 1, 2, 3	REMOVED
1	1 2 3 4 5 6 7	1, 2, 3	**3**
2	4 5 6 7 1 2	4, 5, 6	**6**
3	7 1 2 4 5	7, 1, 2	**2**
4	4 5 7 1	4, 5, 7	**7**
5	1 4 5	1, 4, 5	**5**
6	1 4	1, 4, 1	**1**

Order of removal: 3, 6, 2, 7, 5, 1. Survivor: **person 4**. Check with the recurrence J(1) = 0, J(n) = (J(n − 1) + k) mod n (0-based): J(2) = 1, J(3) = 1, J(4) = 0, J(5) = 3, J(6) = 0, J(7) = 3, which is position 3 from 0, person 4. The list simulation costs O(n × k) pointer steps: 6 rounds × 2 steps = 12 advances plus 6 deletions.

```
Node *prev = tail;                    /* node 7; tail->next is person 1 */
while (prev->next != prev) {          /* more than one node left */
    for (int i = 1; i < k; i++) prev = prev->next;   /* advance k-1 */
    Node *victim = prev->next;
    prev->next = victim->next;        /* unlink */
    printf("%d removed\n", victim->data); free(victim);
}
printf("survivor %d\n", prev->data);  /* survivor 4 */
```

EXAM TRAPS

- Traversing with `while (p != NULL)`: it never terminates. Use do-while and stop at the first node.
- Forgetting the one-node and empty cases: with one node, `tail->next == tail`, and deleting it must set tail to NULL.
- Inserting at the end but not moving `tail`; the node is then at the front instead.
- Saying deletion at the end is O(1) for a singly circular list: it is O(n), because the node before tail must be found.
- Favourite question: "Explain the advantages of a circular list over a singly linked list, and write the insertion routines." Advantages: reach any node from any node, O(1) insertion at both ends with one pointer, natural for round-robin; disadvantage: infinite-loop risk and a slightly harder empty case.

CHECK YOURSELF

– REMEMBER **What does the last node's next field hold in a circular linked list?** 1 mark

The address of the first node.

– REMEMBER **How is the empty circular list represented with a tail pointer, and how do you recognise a one-node list?** 2 marks

tail == NULL; a one-node list has tail→next == tail.

– UNDERSTAND **Why does keeping a pointer to the last node rather than the first give O(1) insertion at both ends?** 3 marks

The last node is tail itself, and the first node is tail→next, so both are reachable in one step; inserting between them (after tail) is the same two assignments, and moving tail decides which end the new node counts as.

– APPLY **Starting from an empty circular list, insertEnd(5), insertEnd(8), insertFront(2), deleteFront(). Show the ring and tail after each step.** 4 marks

5 (tail = 5, 5→5); 5 → 8 → 5 (tail = 8); 2 → 5 → 8 → 2 (tail = 8); 5 → 8 → 5 (tail = 8, node 2 freed).

– APPLY **Five people, every second one removed (k = 2), counting from person 1. Who survives?** 3 marks

Removed 2, 4, 1, 5; survivor **3**. (Recurrence: J(2) = 0, J(3) = 2, J(4) = 0, J(5) = 2 → person 3.)

– ANALYZE **An OS scheduler cycles through processes and must remove a finished process in O(1). Which list variant and which pointers would you keep?** 4 marks

A circular doubly linked list with a pointer to the current process: next gives the following process in O(1), and the current node can be unlinked in O(1) because it knows its prev. A singly circular list would need the previous pointer kept alongside.

BEYOND THE SYLLABUS

Not for the exam. The Josephus problem has a closed form for k = 2 (rotate the binary representation of n one bit left) and O(n) or O(k log n) solutions otherwise; it is a standard dynamic-programming example. Circular buffers (ring buffers) are the array-based cousin used in network cards, audio drivers and lock-free single-producer queues. Search "Josephus problem closed form" and "ring buffer".

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
Insert after q (doubly)	p→next = q→next; p→prev = q; q→next→prev = p; q→next = p	2.5
Delete cur (doubly)	cur→prev→next = cur→next; cur→next→prev = cur→prev	2.5
Circular with tail	first = tail→next; insert front/end O(1); traverse with do-while	2.6
Josephus recurrence	J(1) = 0; J(n) = (J(n − 1) + k) mod n, 0-based	2.6

Exam question bank

Long-answer exam questions on these chapters, each with an outline of the points an examiner looks for. Answer one in full on paper before you read its outline.

Remember and understand

- **4. Compare singly, doubly and circular linked lists: structure, memory per node, operations that are O(1) in each, and one application of each. (10 marks)**

  Outline: node layouts; 2, 3 and 2 fields; singly: O(1) front insert; doubly: O(1) delete given node, insert before, backward traversal; circular with tail: O(1) insertion at both ends and deletion at the front (deleting the last node is still O(n)); applications: hash chains, undo/redo or LRU, round-robin or Josephus; traversal conditions (NULL vs back to first).

Apply

- **8. Write the algorithm to insert a node after a given node in a doubly linked list and to delete a given node, and apply both to 1 ⇄ 2 ⇄ 4 (insert 3 after 2, then delete 1). (10 marks)**

  Outline: four assignments in order for insertion with the last-node guard; two bypass assignments plus head/tail cases for deletion; result after insert 1 ⇄ 2 ⇄ 3 ⇄ 4; delete 1: head = node 2, node2→prev = NULL, free; final 2 ⇄ 3 ⇄ 4; state the invariant preserved.

- **10. Six people stand in a circle and every fourth is eliminated starting the count at person 1. Simulate with a circular linked list and find the survivor; verify with the recurrence. (8 marks)**

  Outline: ring 1 → 2 → 3 → 4 → 5 → 6 → back to 1, prev starts at node 6; each round advance prev three steps and delete prev→next. Round 1: count 1, 2, 3, 4 → remove 4. Round 2, from 5: 5, 6, 1, 2 → remove 2. Round 3, from 3: 3, 5, 6, 1 → remove 1. Round 4, from 3: 3, 5, 6, 3 → remove 3. Round 5, from 5: 5, 6, 5, 6 → remove 6. Survivor **person 5**. Recurrence with k = 4: J(1) = 0, J(2) = 0, J(3) = 1, J(4) = 1, J(5) = 0, J(6) = 4, and 0-based position 4 is person 5. Marks for the table, the pointer statements and the agreement of the two methods.

Analyze and evaluate

- **14. (Spans 2.4, 2.5, 2.6.) For each of the following, choose the list variant and the pointers to keep, and justify with complexities: a music player with next/previous and repeat-all; a hash table's collision chain; a round-robin scheduler that removes finished tasks. (12 marks)**

  Outline: music: circular doubly linked list, current pointer, O(1) next/previous, repeat-all is free; chain: singly linked, head only, insert at front O(1), search O(chain length), least memory; scheduler: circular (doubly, or singly with prev kept) with current pointer, O(1) advance and O(1) removal; mention memory per node for each choice.

Glossary (terms used in this lesson)

Array	Fixed number of same-type elements stored contiguously; element address computed from the index.
Node	A block holding one element and one or more pointers to other nodes.
Head	Pointer to the first node of a list; NULL when the list is empty.
Tail	Pointer to the last node; in a circular list, tail→next is the first node.
NULL	The pointer value meaning "no node"; marks the end of a non-circular list.
Header node	A dummy node kept permanently at the front so that insert and delete have no empty-list special case.
Singly linked list	Nodes each with one next pointer; traversal in one direction only.
Doubly linked list	Nodes with prev and next pointers; O(1) deletion given a node pointer.
Circular linked list	List whose last node points back to the first; no NULL links.
Traversal	Visiting every node in order, following pointers from the head.
Josephus problem	Eliminating every k-th person around a circle; modelled by a circular list or the recurrence $J(n) = (J(n-1) + k) \bmod n$.

Sources and further reading

Textbooks

- Ellis Horowitz, Sartaj Sahni and Susan Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd edition. The arrays and sparse-matrix chapter (including the fast transpose) and the linked-lists chapter (polynomials, circular lists, dynamic storage management) are the source of most Indian syllabi for this unit.
- Mark Allen Weiss, *Data Structures and Algorithm Analysis in C*, 2nd edition. Chapter 3, "Lists, Stacks, and Queues": the List ADT, array and linked implementations, polynomial and radix-sort applications.
- Aaron M. Tenenbaum, Yedidyah Langsam and Moshe J. Augenstein, *Data Structures Using C*. Linked lists in C, lists with header nodes, dynamic storage (widely prescribed).
- Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, 2nd edition. Section 8.7, "A Storage Allocator": a free-list allocator with first fit and coalescing in under a page of code.
- Donald E. Knuth, *The Art of Computer Programming, Volume 1*, section 2.5, "Dynamic Storage Allocation": first fit, best fit, boundary tags and garbage collection, the origin of chapter 2.8.

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*, chapter 10, "Elementary Data Structures": linked lists with sentinels and the array-based representation of pointers.

Open web resources (opened while writing)

- Wikipedia: Row- and column-major order: which languages use which layout, and the offset formulas.
- Wikipedia: Josephus problem: history, the k = 2 closed form and the general recurrence.
- cppreference: malloc: the C standard's contract for malloc, including the null return on failure and alignment of the returned pointer.

Syllabi: the universal syllabus and a range of university course documents.

Built for the University free n easy YouTube channel from the universal syllabus of Data Structures (Oct 2026). Written with top universities' syllabi and standard textbooks; check a formula against your prescribed book before an exam, and report mistakes on the channel.