UNIVERSAL SYLLABUS · DATA STRUCTURES

Data Structures

Polynomials and memory management

The notes for the lesson “Polynomials and memory management”: chapters 2.7 to 2.8, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

2 chapters | Chapters 2.7 to 2.8

About these notes

These are the notes for the University free n easy video lesson **Polynomials and memory management** on Data Structures: chapters 2.7 to 2.8, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

2.7 Polynomial manipulation using linked lists

A polynomial such as $5x^3 + 4x^2 + 2$ is a list of terms, each a coefficient and an exponent. Storing it in an array indexed by exponent wastes space the moment the polynomial is sparse ($x^{100} + 1$ needs 101 slots for two terms), so each term becomes a node in a linked list kept in decreasing order of exponent. Addition then becomes a walk down two sorted lists, the same merge pattern used later in merge sort and in sparse-matrix addition. This is the linked-list application examiners ask for most.

You need: singly linked list insertion at the end from chapter 2.4.

Representation

```
typedef struct Term {
    int coef;           /* coefficient, never 0 in a stored term */
    int exp;            /* exponent, strictly decreasing along the list */
    struct Term *next;
} Term;
```

Two rules keep the representation canonical, so that two equal polynomials have identical lists: terms are sorted by **decreasing exponent**, and **no term has coefficient 0**. The zero polynomial is

the empty list. A term with exp 0 is the constant. Under these rules $5x^3 + 4x^2 + 2$ is the list (5,3) → (4,2) → (2,0) → NULL; the missing x^1 term is simply absent.

Building a polynomial means appending terms in the order given; with a tail pointer each append is O(1). An `attach(coef, exp, &tail)` helper that creates a node and links it after the tail keeps the addition code short.

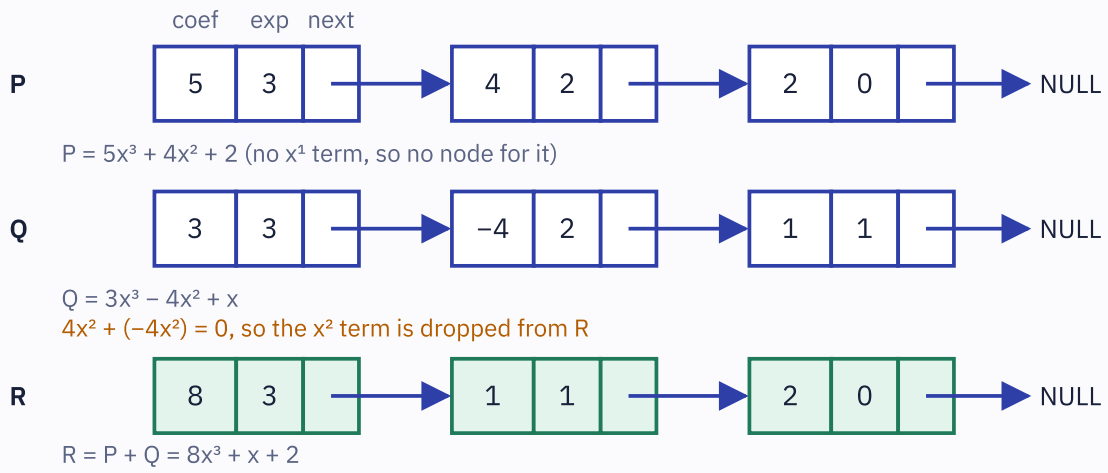

Figure 2.7.1 P and Q stored as linked lists of (coef, exp) nodes in decreasing exponent order, and their sum R. The x^3 terms combine to 8; the x^2 terms cancel and get no node; x and the constant are copied from whichever list has them. R has three nodes though P and Q have six between them.

Addition: the merge walk

Keep a pointer p on P and q on Q, both starting at the first term. Repeat while both are non-NULL:

1. If `p->exp > q->exp`: copy p's term to the result; advance p.
2. If `p->exp < q->exp`: copy q's term; advance q.
3. If the exponents are equal: sum = `p->coef + q->coef`; if sum ≠ 0 attach (sum, exp) to the result; advance both. A zero sum attaches nothing.

When one list runs out, copy the rest of the other. Each step advances at least one pointer, so the time is O(m + n) for polynomials of m and n terms, and the result has at most m + n terms.

```
Term *add(Term *p, Term *q) {
    Term *head = NULL, *tail = NULL;
    while (p && q) {
        if (p->exp > q->exp)      { attach(p->coef, p->exp, &head, &tail); p = p->next; }
        else if (p->exp < q->exp) { attach(q->coef, q->exp, &head, &tail); q = q->next; }
        else {
            int s = p->coef + q->coef;
```

```
            if (s != 0) attach(s, p->exp, &head, &tail);
            p = p->next; q = q->next;
        }
    }
    for (; p; p = p->next) attach(p->coef, p->exp, &head, &tail);   /* leftovers */
    for (; q; q = q->next) attach(q->coef, q->exp, &head, &tail);
    return head;
}
```

Subtraction, multiplication and evaluation

Subtraction (some syllabi list "Polynomial Addition/Subtraction"): P − Q is P + (−Q); either negate every coefficient of a copy of Q and call add, or write the same walk with `p->coef - q->coef` in the equal case and `-q->coef` when copying from Q. The zero-sum rule still applies.

Multiplication: for each term of P, multiply it with every term of Q (coefficients multiply, exponents add) and add the resulting single-term polynomial into the running result with the add routine, which merges like terms. With m and n terms there are m × n products. Inserting each product one at a time into a sorted result of up to m × n terms costs $O(m^2n^2)$; multiplying one term of P by the whole of Q (a sorted n-term list) and merging that list into the running result with add costs O(mn) per term of P, $O(m^2n)$ overall. **Evaluation** at x = a is one walk computing coef × a^{exp} per node, O(n) with a power function, or Horner's rule when exponents are dense.

WORKED EXAMPLE

Question. Represent $P = 5x^3 + 4x^2 + 2$ and $Q = 3x^3 - 4x^2 + x$ as linked lists and compute $P + Q$ using the merge algorithm, showing each step. Then evaluate the result at $x = 2$.

Solution. P: (5,3) → (4,2) → (2,0). Q: (3,3) → (−4,2) → (1,1). Pointers p and q start at the first nodes.

STEP	P TERM	Q TERM	COMPARE EXP	ACTION	RESULT SO FAR
1	(5,3)	(3,3)	3 = 3	sum 5 + 3 = 8 ≠ 0, attach (8,3); advance both	$8x^3$
2	(4,2)	(−4,2)	2 = 2	sum 4 + (−4) = 0, attach nothing; advance both	$8x^3$
3	(2,0)	(1,1)	0 < 1	copy q's (1,1); advance q	$8x^3 + x$
4	(2,0)	NULL	q exhausted	copy rest of P: (2,0)	$8x^3 + x + 2$

R = $8x^3 + x + 2$, three nodes, as in figure 2.7.1. Evaluation at x = 2: $8 \times 2^3 + 1 \times 2^1 + 2 \times 2^0 = 8 \times 8 + 2 + 2 = 64 + 4 =$ **68**. Cross-check with P(2) + Q(2) = (40 + 16 + 2) + (24 − 16 + 2) = 58 + 10 = 68.

Subtraction check. P − Q: step 1 gives 5 − 3 = 2 → (2,3); step 2 gives 4 − (−4) = 8 → (8,2); step 3 copies −(1,1) = (−1,1); step 4 copies (2,0). $P - Q = 2x^3 + 8x^2 - x + 2$.

EXAM TRAPS

- Attaching a node with coefficient 0 when like terms cancel; the result then has a "term" that is not a term and later comparisons misbehave.
- Comparing coefficients instead of exponents in the merge, or storing the terms in increasing exponent order and then merging with the wrong inequality.
- Forgetting the leftover loop: when one polynomial runs out, the remaining terms of the other are copied, not dropped.
- Multiplication: adding exponents and multiplying coefficients is right; students often do the reverse for one of them under time pressure.
- Favourite question: "Write a C function to add two polynomials represented as linked lists and trace it on a given pair." Draw the three lists, then the table of steps.

CHECK YOURSELF

– REMEMBER **What are the fields of a polynomial term node, and in what order are the nodes kept?** 2 marks

coef, exp and next; nodes are kept in decreasing order of exponent with no zero coefficients.

– UNDERSTAND **Why is a linked list preferred over an exponent-indexed array for polynomials?** 3 marks

An array needs one slot per possible exponent up to the degree, so $x^{100} + 1$ wastes 99 slots; the list stores only the terms present, and addition still runs in time proportional to the number of terms.

– UNDERSTAND **What is the time complexity of adding polynomials of m and n terms, and why?** 2 marks

O(m + n): each loop iteration advances at least one of the two pointers, and each pointer moves at most once per term.

– APPLY **Add $A = 3x^4 + 2x^2 + 7$ and $B = x^4 - 2x^2 + 5x + 1$ using the merge walk.** 4 marks

x^4: 3 + 1 = 4 → (4,4). x^2: 2 − 2 = 0 → dropped. B's (5,1) copied. Constants 7 + 1 = 8 → (8,0). Result **$4x^4 + 5x + 8$**.

– APPLY **Multiply $(2x + 3)$ by $(x^2 - 1)$ term by term and give the resulting list.** 4 marks

$2x \times x^2 = 2x^3$, $2x \times (-1) = -2x$, $3 \times x^2 = 3x^2$, $3 \times (-1) = -3$. Merged in decreasing exponent: (2,3) → (3,2) → (−2,1) → (−3,0), i.e. $2x^3 + 3x^2 - 2x - 3$.

– ANALYZE **A student stores terms in increasing exponent order. Which parts of the addition algorithm change, and does the complexity change?** 3 marks

Only the two inequality tests swap (copy the smaller exponent first); the equal case and leftover loops are unchanged. Still O(m + n). Printing in standard form would then need a reverse or a backward walk.

BEYOND THE SYLLABUS

Not for the exam. Computer algebra systems (SymPy, Mathematica, Maxima) use sparse term lists like this for polynomials in many variables, with exponents stored as vectors and terms ordered by a monomial order. Fast multiplication of dense polynomials uses Karatsuba's divide-and-conquer method or the FFT (as in Schönhage–Strassen) to beat O(mn). Search "sparse polynomial representation" and "polynomial multiplication FFT".

2.8 Dynamic storage management and garbage collection

Every `malloc` in this unit took memory from a pool called the heap, and every `free` gave it back. Dynamic storage management is how that pool is organised: which free block to hand out, how to merge freed blocks, and what to do when the free space is scattered in small pieces. Garbage collection is the automatic version, where the system frees the blocks no pointer reaches. Few syllabi list this topic in Data Structures, so this chapter is short.

The free list and allocation strategies

The allocator keeps the unused blocks of the heap on a **free list**, a linked list in which each block records its size and the next free block. On a request for n bytes it picks a block of size ≥ n by one of three policies, **splits** it, and leaves the remainder on the list:

- **First fit**: the first block large enough. Fast; leaves small fragments near the front of the list.
- **Best fit**: the smallest block large enough. Scans the whole list; leaves tiny, often unusable, leftovers.
- **Worst fit**: the largest block. Leaves big remainders; rarely best in practice.

On `free`, the block returns to the list and is **coalesced** (merged) with any free neighbour, found through size fields at both ends of each block (boundary tags). **Fragmentation** is *external* when the free space is large in total but no single block is large enough, and *internal* when a block handed out is bigger than requested.

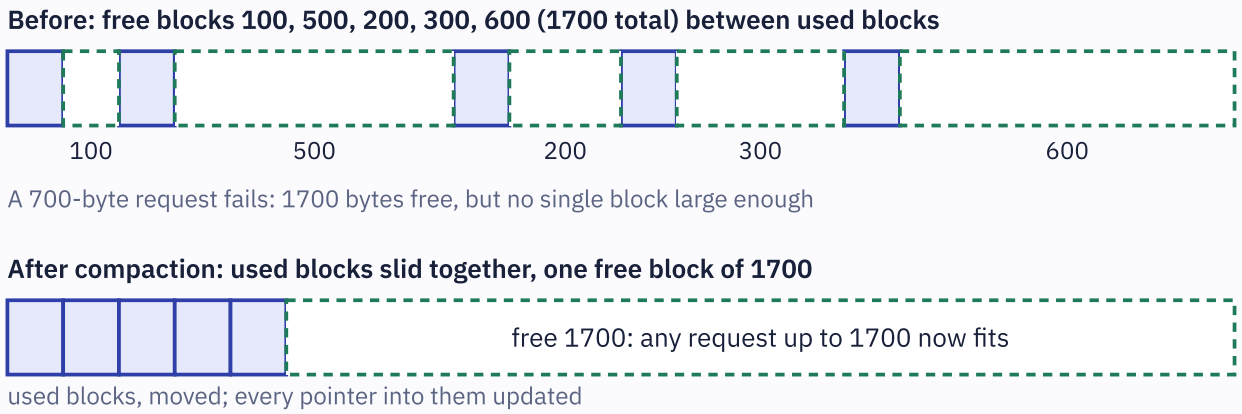

Figure 2.8.1 External fragmentation and its cure. Above, 1700 bytes are free but split into five blocks. Below, compaction has slid the used blocks together and rewritten every pointer into them, leaving one 1700-byte block.

Garbage collection and compaction

In C the programmer calls `free`; forget it and the block leaks. Java, Python and Go instead collect **garbage**: blocks that no live variable can reach. Two classic methods:

1. **Reference counting**: each block stores how many pointers refer to it, updated on every assignment; a block whose count drops to 0 is freed at once. Simple and prompt, but two blocks that point at each other (a cycle) never reach 0, so they leak.
2. **Mark and sweep**: starting from the roots (global variables, the stack), follow every pointer and mark each block reached; then sweep the whole heap, freeing every unmarked block. Cycles are no problem, but the program pauses while the collector runs, and the heap stays fragmented.

Compaction fixes the fragmentation: after marking, the live blocks are slid to one end of the heap and every pointer to them is rewritten (figure 2.8.1).

WORKED EXAMPLE

Question. The free list holds 100, 500, 200, 300 and 600 bytes, in that order. Requests arrive for 212, 417, 112 and 426 bytes. Serve them under first fit and under best fit, showing the free list after each request.

Solution.

REQUEST	FIRST FIT: BLOCK → REMAINDER	FREE LIST AFTER	BEST FIT: BLOCK → REMAINDER	FREE LIS
212	500 → 288	`100, 288, 200, 300, 600`	300 → 88	`100, 50 200, 88`
417	600 → 183	`100, 288, 200, 300, 183`	500 → 83	`100, 83 200, 88`
112	288 → 176	`100, 176, 200, 300, 183`	200 → 88	`100, 83 88, 600`
426	**fails**: largest block is 300	`unchanged; 959 bytes free`	600 → 174	`100, 83 88, 174`

First fit failed on the last request with 959 bytes free in total (1700 − 212 − 417 − 112): external fragmentation. Best fit served all four because it put 212 into the 300 block and 417 into the 500 block, keeping the 600 block whole for the 426 request; first fit had already spent the 600 block on 417.

EXAM TRAPS

- Best fit means the smallest block that fits, found by scanning every block, not the first adequate one.
- Confusing internal (inside an allocated block) and external (between blocks) fragmentation.
- Claiming reference counting handles all garbage: cycles are the standard counterexample.
- Favourite question: "Explain mark-and-sweep and the need for compaction, with a diagram." Draw roots, reachable and unreachable blocks, then the compacted heap.

CHECK YOURSELF

– REMEMBER **Name the three placement policies for a free list.** 1 mark

First fit, best fit, worst fit.

– REMEMBER **What are the two phases of mark-and-sweep?** 2 marks

Mark: follow pointers from the roots and mark every reachable block. Sweep: scan the heap and free every unmarked block.

– UNDERSTAND **Why does coalescing matter when a block is freed?** 2 marks

Unmerged neighbours stay small forever, so large requests fail even when enough total memory is free.

– APPLY **Free list 300, 150, 700; requests 120 and 650. Serve them with worst fit and give the final list.** 3 marks

120 → 700 (largest), remainder 580: list 300, 150, 580. 650 → no block ≥ 650: **fails**.

– ANALYZE **A real-time controller cannot tolerate pauses. Reference counting or mark-and-sweep? Discuss.** 4 marks

Reference counting spreads the work over every assignment with no long pause, which suits real-time, but leaks cycles. Mark-and-sweep stops the program for the whole collection unless an incremental or concurrent collector bounds the pause.

BEYOND THE SYLLABUS

Not for the exam. Real allocators (glibc malloc, jemalloc) keep separate free lists per size class and per thread; Rust avoids garbage collection with ownership rules checked at compile time; Java's ZGC and Go's collector run alongside the program with pauses under a millisecond. Search "boundary tags", "generational garbage collection" and "slab allocator".

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
Polynomial add	merge by decreasing exponent, drop zero sums; O(m + n)	2.7
First / best / worst fit	first adequate block / smallest adequate / largest block	2.8
Mark and sweep	mark reachable from roots, free the rest; compaction slides live blocks together	2.8

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

- **5. Explain reference counting and mark-and-sweep garbage collection, their advantages and limitations, and the purpose of compaction. (8 marks)**

  Outline: reference count per block, freed at zero, cannot free cycles, overhead per pointer write; mark from roots, sweep unmarked, handles cycles, pauses program; external fragmentation after sweep; compaction slides live blocks and rewrites pointers; diagram.

Apply

- **9. Write a C function to add two polynomials stored as linked lists and trace it on $P = 6x^4 + 3x^2 - 5$ and $Q = 2x^4 - 3x^2 + 7x$. (12 marks)**

  Outline: node (coef, exp, next), decreasing exponents, attach helper, merge loop with the three cases and the zero-sum rule, leftover loops; trace: x^4: 8; x^2: 0 dropped; Q's 7x copied; P's −5 copied; result **$8x^4 + 7x - 5$**; O(m + n).

- **11. A free list holds 200, 400, 150, 600 and 300 bytes in that order. Serve requests of 250, 350, 120 and 500 under first fit and best fit, show the list after each request, and state which policy fails. (8 marks)**

  Outline: first fit: 250 → 400 (150 left), 350 → 600 (250 left), 120 → 200 (80 left), 500 → **fails**, no block ≥ 500 although 80 + 150 + 150 + 250 + 300 = 930 bytes are free. Best fit: 250 → 300 (50), 350 → 400 (50), 120 → 150 (30), 500 → 600 (100): all served, final list 200, 50, 30, 100, 50. Name the failure as external fragmentation and note that best fit scans the whole list each time.

Analyze and evaluate

- **13. (Spans 2.2 and 2.7.) Sparse matrices and polynomials are both stored as lists of non-zero terms. Analyse what the two representations share, how their addition algorithms are alike, and where they differ. (10 marks)**

  Outline: both store only non-zero entries with their position (exponent vs row/col), both kept sorted so addition is a merge, both drop zero sums; sparse addition compares (row, col) pairs lexicographically, polynomial compares one exponent; triplets usually in an array (transpose needs re-sorting), polynomials in a linked list (insertion-friendly); complexity $O(t_1 + t_2)$ and $O(m + n)$.

- **16. Analyse why reference counting cannot free a doubly linked list that has become unreachable, and propose a fix. (8 marks)**

  Outline: each node is referenced by its neighbour's prev and next, so counts stay ≥ 1 after the head is dropped; the cycle keeps itself alive; fixes: break links before dropping the head, use weak references for prev, or run a cycle-detecting or mark-and-sweep collector; mention CPython's approach.

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.
Singly linked list	Nodes each with one next pointer; traversal in one direction only.

Merge walk	Advancing two pointers through two sorted lists, taking the smaller key each step; the shape of polynomial and sparse addition.
Free list	Linked list of unused heap blocks kept by the allocator.
Coalescing	Merging a freed block with adjacent free blocks.
Fragmentation	External: free memory split into blocks too small to use; internal: waste inside an allocated block.
Garbage collection	Automatic freeing of blocks no live pointer reaches; by reference counting or mark-and-sweep.
Compaction	Moving live blocks together and updating pointers to leave one large free block.

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.