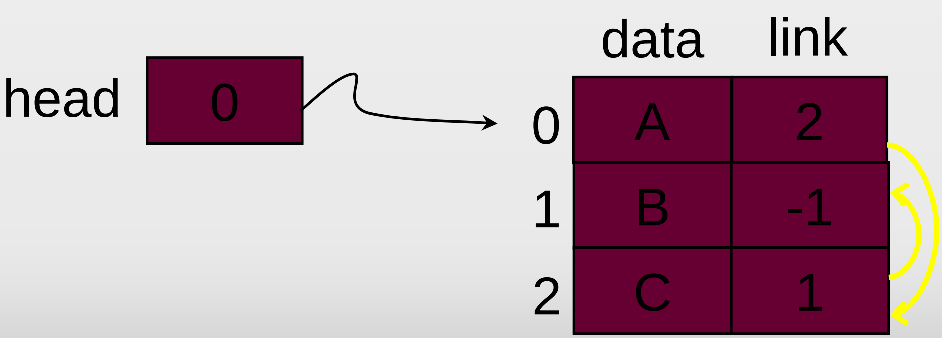
Linked Lists

Data Structures

Ching-Fang Hsu
Department of Computer Science and Information Engineering
National Cheng Kung University

Why Lists?

- Problems of ordered lists implemented by arrays
  - Data movement
  - Although the data movement problem can be avoided by implementing an ordered list by two arrays, memory management problem still exists.

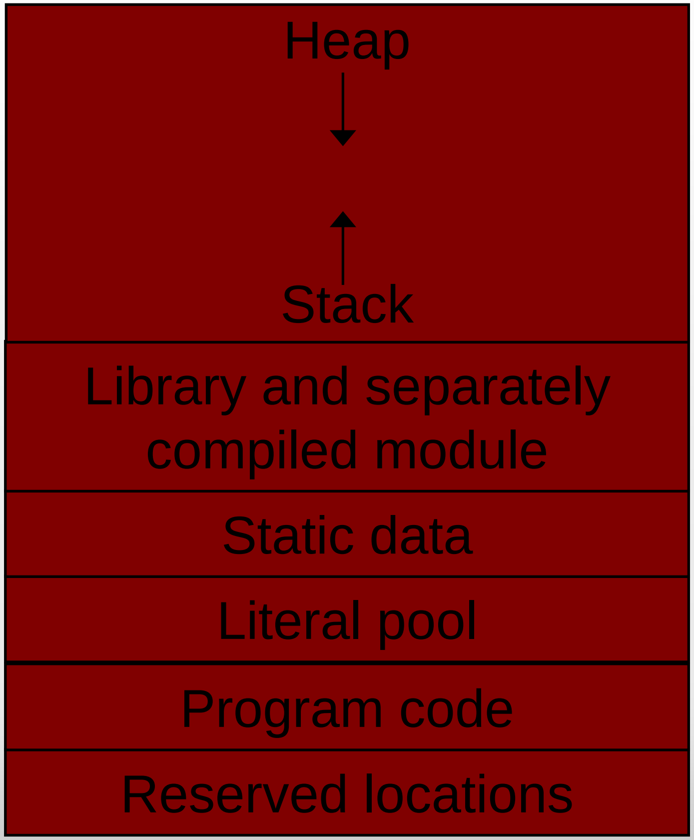


Pointers

- For any type T in C there is a corresponding type pointer-to-T.
- The actual value of a pointer type is an address of memory.
  - &: the address operator
  - $*$: the dereferencing (or indirection) operator
- Accessing dynamically allocated storage (i.e., heap)

Pointers (contd.)

- A typical program layout in memory



Pointers (contd.)

- Pointer problems
  - Dangling pointers problem
    - A dangling pointer is a pointer that contains the address of a heap-dynamic variable that has been deallocated.
  - Memory leakage
    - This problem occurs when an allocated heap-dynamic variable is no longer accessible to the user program.

Pointers (contd.)

Pointer p1 is set to point at a new heap-dynamic variable.

↓

Pointer p2 is assigned p1’s value.

↓

The heap-dynamic variable pointed to by p1 is explicitly deallocated, but p2 is not changed by this operation.

↓

p2 is a dangling pointer.

Pointers (contd.)

Pointer `p1` is set to point to a newly heap-dynamic variable.

↓

`p1` is later set to point to another newly created heap-dynamic variable.

↓

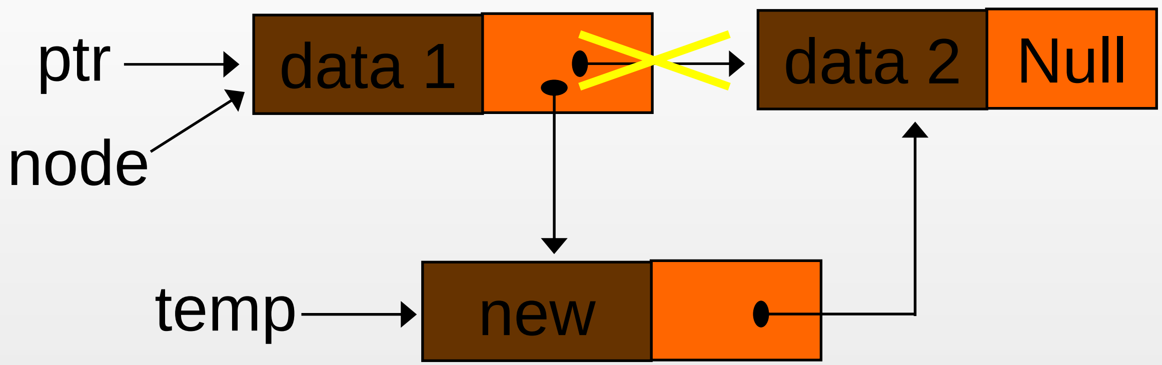
The first heap-dynamic variable is inaccessible, or lost.

Singly Linked Lists

- Each node on a singly linked list consists of exactly one link field and at least one other field (p.147, Fig. 4.2).
- Necessary capabilities to make linked representations possible
  - A mechanism for defining a node's structure
  - A way to create new nodes when we need them
    - *malloc* in C
  - A way to remove nodes that we no longer need
    - *free* in C

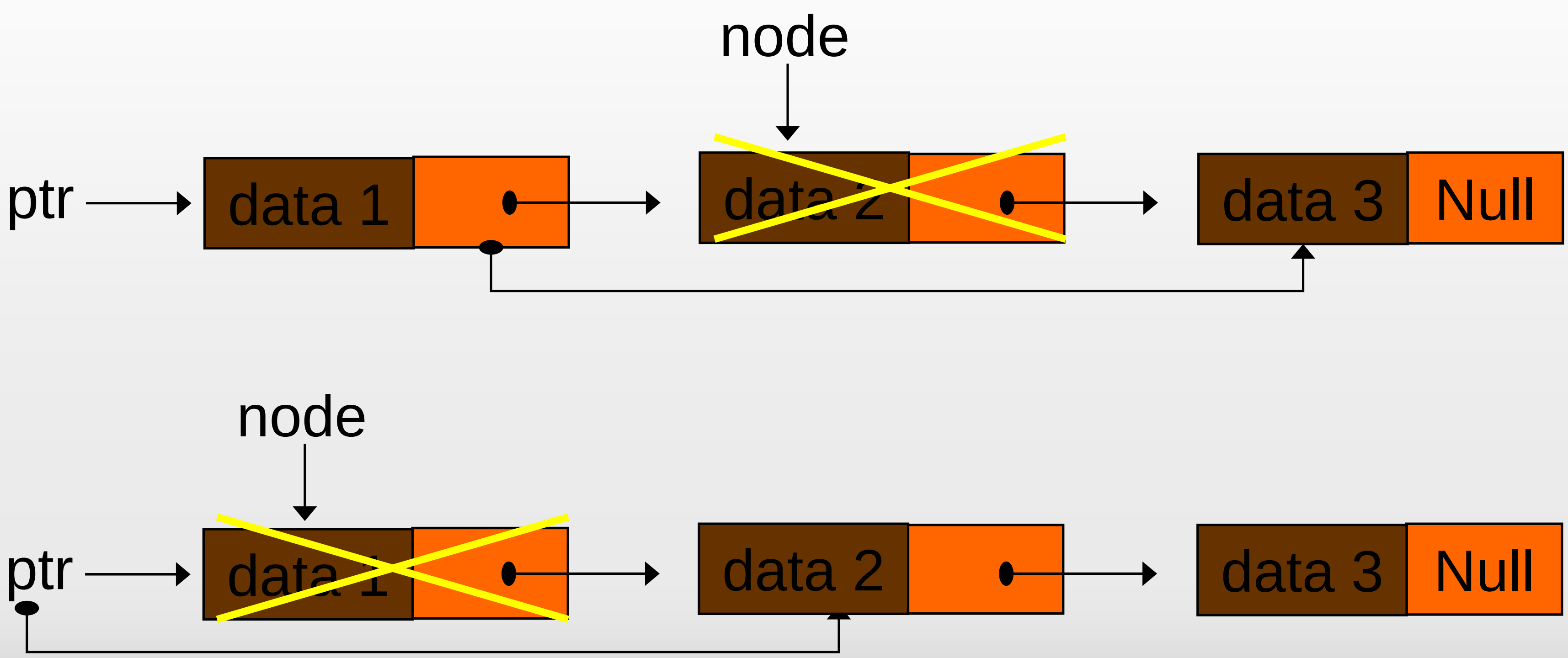
Singly Linked Lists -- Operations

- Insertion (p. 153, Program 4.2)



Singly Linked Lists -- Operations (contd.)

- Deletion (p. 155, Program 4.3)



Dynamically Linked Stacks and Queues

- Stacks
  - Structure definitions (p. 156)
  - The initial condition
    - $top[i] = NULL, 0 \leq i \leq MAX\_STACKS$
  - Boundary conditions
    - $top[i] = NULL$ iff the ith stack is empty, and
    - the memory is full
  - Push operation (p.158, Program 4.5)
  - Pop operation (p.158, Program 4.6)

Dynamically Linked Stacks and Queues (contd.)

- Queues
  - Structure definitions (p. 158)
  - The initial condition
    - $front[i] = NULL$, $0 \le i \le MAX\_QUEUES$
  - Boundary conditions
    - $front[i] = NULL$ iff the ith queue is empty, and
    - the memory is full
  - Insertion (p.159, Program 4.7)
  - Deletion (p.160, Program 4.8)

Polynomials

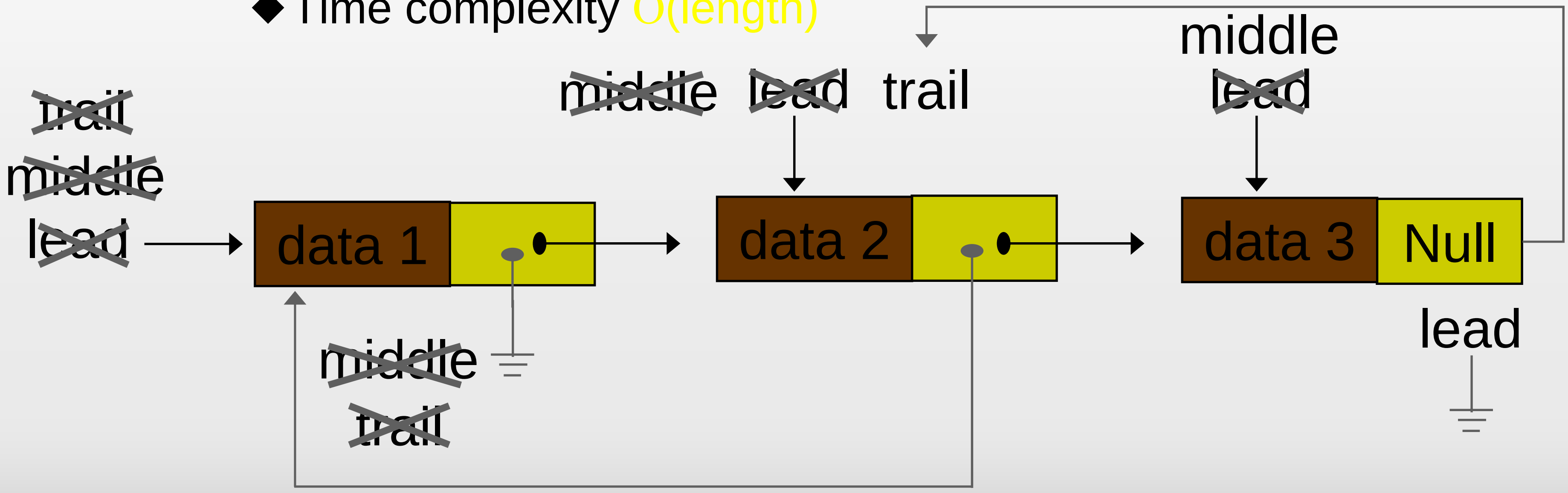
- Each polynomial term can be defined as

coef	expon	link

- Adding polynomials
  - Three cost measures
    - Coefficient additions
    - Exponent comparisons
    - Creation of new nodes
  - Time complexity $O(m + n)$, assuming that the two polynomials have m and n terms, respectively
    - p. 163~164, Program 4.9 and 4.10

Additional List Operations

- Inverting chains
  - "in place" processing if there are three pointers
  - p. 171, Program 4.16
    - Time complexity O(length)



Equivalence Relations

- **Definition**: A relation, $\equiv$, over a set, S, is said to be an equivalence relation over S iff it is symmetric, reflexive, and transitive over S.
  - Reflexive: $x \equiv x$
  - Symmetric: $x \equiv y \rightarrow y \equiv x$
  - Transitive: $x \equiv y$ and $y \equiv z \rightarrow x \equiv z$
- Equivalence classes of a set S
  - Two members x and y of S are in the same equivalence class iff $x \equiv y$.

Equivalence Relations (contd.)

- Equivalence determination
  - Phase 1: Read in and store the equivalence pairs <i, j>
    - p. 176, Fig. 4.16
  - Phase 2: Begin at 0 and find all pairs of the form <0, j>, where 0 and j are in the same equivalence class.
- P. 177~178, Program 4.22
  - Let m and n represent the number of related pairs and the number of objects, respectively.
  - Phase 1: $O(m + n)$
  - Phase 2: $O(m + n)$

The overall computing time is $O(m + n)$.

Sparse Matrices

- Each column of a sparse matrix is represented as a circularly linked list with a head node.
  - A similar representation for each row
- Node structure for sparse matrices (p. 179, Fig. 4.17)
  - A tag field is used to distinguish between head nodes and entry nodes.
  - The *down* field is used to link into a column list and the *right* field to link into a row list.
    - The head node for row i is also the head node for column i.

Sparse Matrices (contd.)

- Each head node is in three lists: a list of rows, a list of columns, and a list of head nodes.
- The list of head nodes also has a head node that has this node to store the matrix dimensions.
  - p. 180, Fig. 4.18; p. 181, Fig. 4.19

Doubly Linked Lists

- Singly linked lists pose problems because we can move easily only in the direction of the links.
  - Doubly linked lists
- A node in a doubly linked list has at least three fields.
  - A left link field
  - A data field
  - A right link field

Doubly Linked Lists -- Doubly Linked Circular Lists (contd.)

- A doubly linked list may or may not be circular.
- A head node allows us to implement our operations more easily.
  - The item field of the head node usually contains no information.
  - An empty list is not really empty. (p. 188, Fig. 4.22)
- Insertion and Deletion
  - In constant time (p. 188~189, Program 4.26 and 4.27)