Linked Lists

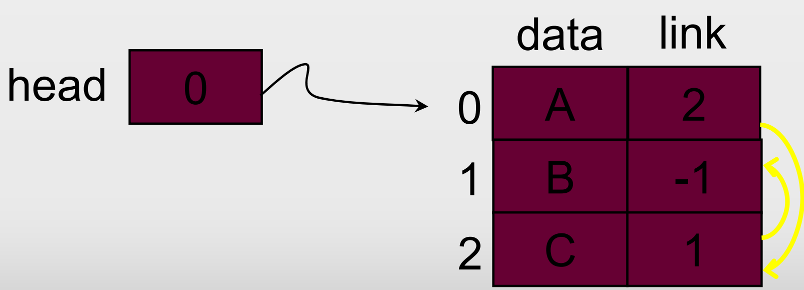
Data Structures

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.

使用array模擬 linked list



只要改link欄位，
不用改到data

Pointers

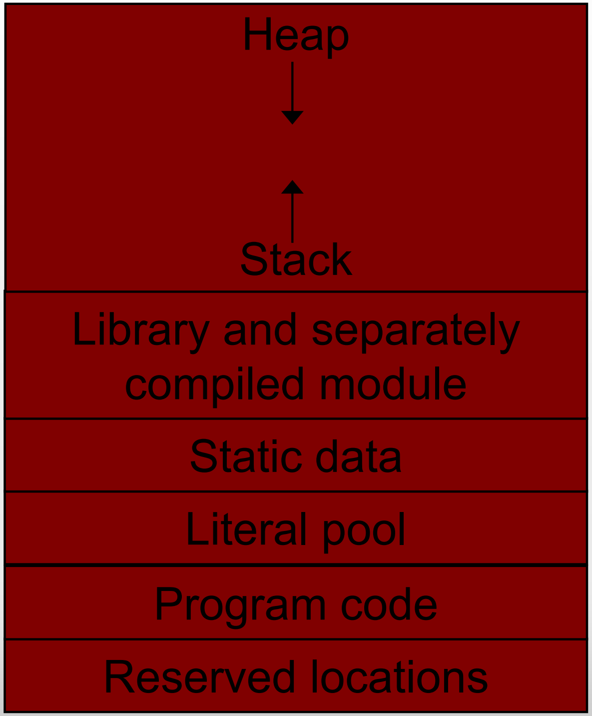
- For any type T in C there is a corresponding type pointer-to-T.
- The actual value of a pointer type is an address of memory.
  - &: the address operator
  - $*$: the dereferencing (or indirection) operator
- Accessing dynamically allocated storage (i.e., heap)

+ variable
取出來是一個位置

+ address
Return出來是
data value

Pointers (contd.)

- A typical program layout in memory



Pointers (contd.)

使用ptr需注意的問題

懸置指標：指標指到一個不存在任何有意義資訊的記憶體空間（不應該再去存取的ptr，下次再做dereferencing 將會取到錯誤的值）

- 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.

memory leakage：
如果 allocate 出來了之後、在沒有指標去指到那塊記憶體空間、又沒有做對應的 free 的情況下，就會產生「記憶體還是佔在那邊，但是卻沒有辦法使用、也沒辦法釋放」的問題

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.

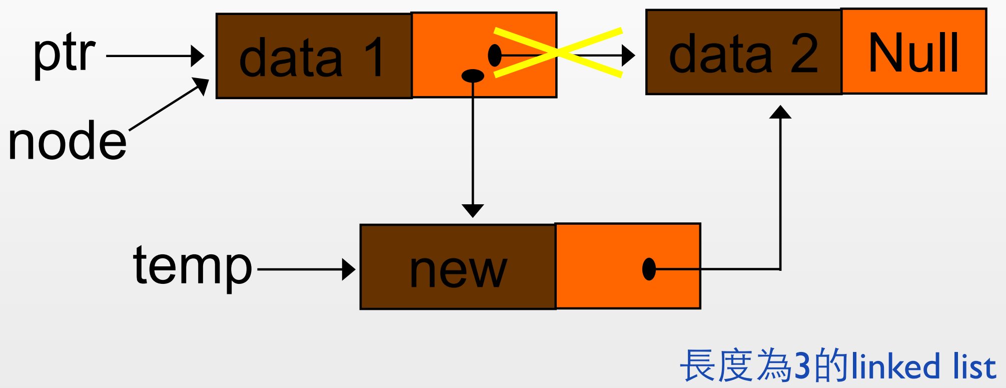
Singly Linked Lists

讓使用者可以去規範的node

- 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 凡是使用node記得做記憶體的釋放
  - A way to remove nodes that we no longer need
    - *free* in C

Singly Linked Lists -- Operations

- Insertion (p. 153, Program 4.2)



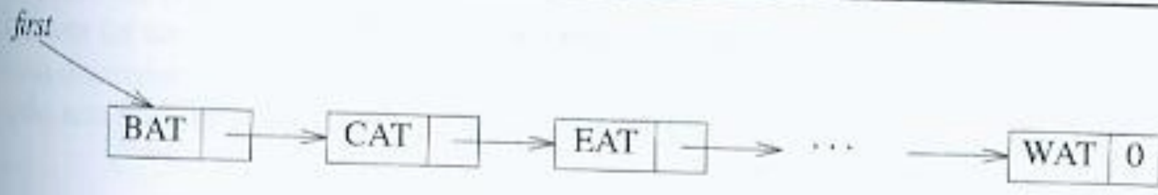


Figure 4.2: Usual way to draw a linked list

```
void insert(listPointer *first, listPointer x)
{/* insert a new node with data = 50 into the chain
   first after node x */
  listPointer temp;
  MALLOC(temp, sizeof(*temp));
  temp→data = 50;
  if (*first) {
    temp→link = x→link;
    x→link = temp;
  }
  else {
    temp→link = NULL;
    *first = temp;
  }
}
```

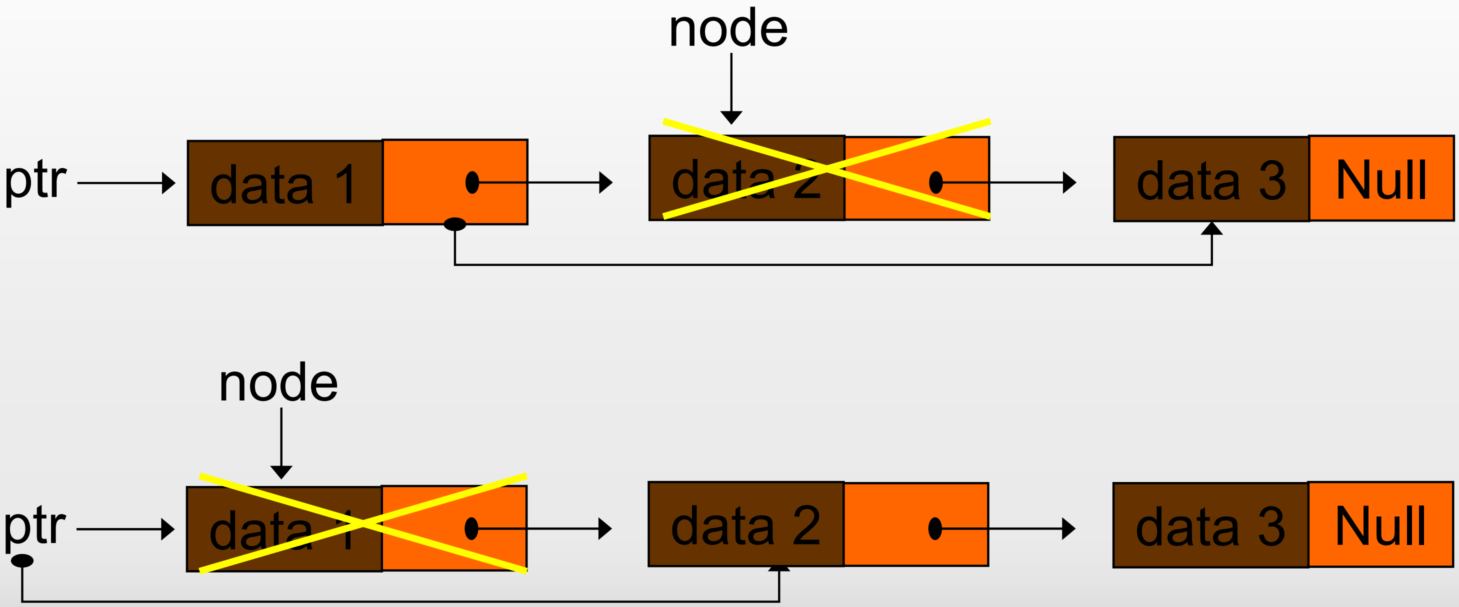
Program 4.2: Simple insert into front of list

```
void delete(listPointer *first, listPointer trail,
                                 listPointer x)
{/* delete x from the list, trail is the preceding node
    and *first is the front of the list */
  if (trail)
    trail→link = x→link;
  else
    *first = (*first)→link;
  free(x);
}
```

Program 4.3: Deletion from a list

Singly Linked Lists -- Operations (contd.)

- Deletion (p. 155, Program 4.3)



Dynamically Linked Stacks and Queues

- Stacks
  - Structure definitions (p. 156)
  - The initial condition 初始化，去做設定
    - $top[i] = NULL, 0 \le i \le MAX\_STACKS$ 比較使用array去做存取stack
  - Boundary conditions 判斷邊界
    - $top[i] = NULL$ iff the ith stack is empty, and
    - the memory is full 嘗試去做memory allocation，如果有free memory會return回ptr，如果滿了會回傳error message
  - Push operation (p.158, Program 4.5)
  - Pop operation (p.158, Program 4.6)

```c
#define MAX_STACKS 10 /* maximum number of stacks */
typedef struct {
        int key;
        /* other fields */
        } element;
typedef struct stack *stackPointer;
typedef struct stack {
        element data;
        stackPointer link;
        };
stackPointer top[MAX_STACKS];
```

```c
void push(int i, element item)
{/* add item to the ith stack */
  stackPointer temp;
  MALLOC(temp, sizeof(*temp));
  temp→data = item;
  temp→link = top[i];
  top[i] = temp;
}
```

Program 4.5: Add to a linked stack

```c
element pop(int i)
{/* remove top element from the ith stack */
  stackPointer temp = top[i];
  element item;
  if (!temp)
    return stackEmpty();
  item = temp→data;
  top[i] = temp→link;
  free(temp);
  return item;
}
```

Program 4.6: Delete from a linked stack

Dynamically Linked Stacks and Queues (contd.)

- Queues
  - Structure definitions (p. 158)
  - The initial condition 初始化
    - $front[i] = NULL, 0 \le i \le MAX\_QUEUES$
  - 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)

```
#define MAX_QUEUES 10 /* maximum number of queues */
typedef struct queue *queuePointer;
typedef struct queue {
        element data;
        queuePointer link;
        };
queuePointer front[MAX_QUEUES], rear[MAX_QUEUES];
```

```
void addq(i, item)
{/* add item to the rear of queue i */
  queuePointer temp;
  MALLOC(temp, sizeof(*temp));
  temp→data = item;
  temp→link = NULL;
  if (front[i])
     rear[i]→link = temp;
  else
     front[i] = temp;
  rear[i] = temp;
}
```

Program 4.7: Add to the rear of a linked queue

```
element deleteq(int i)
{/* delete an element from queue i */
  queuePointer temp = front[i];
  element item;
  if (!temp)
    return queueEmpty();
  item = temp→data;
  front[i]= temp→link;
  free(temp);
  return item;
}
```

Program 4.8: Delete from the front of a linked queue

Polynomials

- Each polynomial term can be defined as

coef	expon	link

- Adding polynomials
  - Three cost measures
    - Coefficient additions
    - Exponent comparisons
    - Creation of new nodes

發生在attach

  - 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

```
polyPointer padd(polyPointer a, polyPointer b)
{/* return a polynomial which is the sum of a and b */
  polyPointer c, rear, temp;
  int sum;
  MALLOC(rear, sizeof(*rear));
  c = rear;
  while (a && b)
    switch (COMPARE(a→expon,b→expon)) {
      case -1: /* a→expon < b→expon */
             attach(b→coef,b→expon,&rear);
             b = b→link;
             break;
      case 0: /* a→expon = b→expon */
             sum = a→coef + b→coef;
             if (sum) attach(sum,a→expon,&rear);
             a = a→link;  b = b→link; break;
      case 1: /* a→expon > b→expon */
             attach(a→coef,a→expon,&rear);
             a = a→link;
    }
  /* copy rest of list a and then list b */
  for (; a; a = a→link) attach(a→coef,a→expon,&rear);
  for (; b; b = b→link) attach(b→coef,b→expon,&rear);
  rear→link = NULL;
  /* delete extra initial node */
  temp = c; c = c→link;  free(temp);
  return c;
}
```

Program 4.9: Add two polynomials

```
void attach(float coefficient, int exponent,
            polyPointer *ptr)
{/* create a new node with coef = coefficient and expon =
    exponent, attach it to the node pointed to by ptr.
    ptr is updated to point to this new node */
  polyPointer temp;
  MALLOC(temp, sizeof(*temp));
  temp→coef = coefficient;
  temp→expon = exponent;
  (*ptr)→link = temp;
  *ptr = temp;
}
```

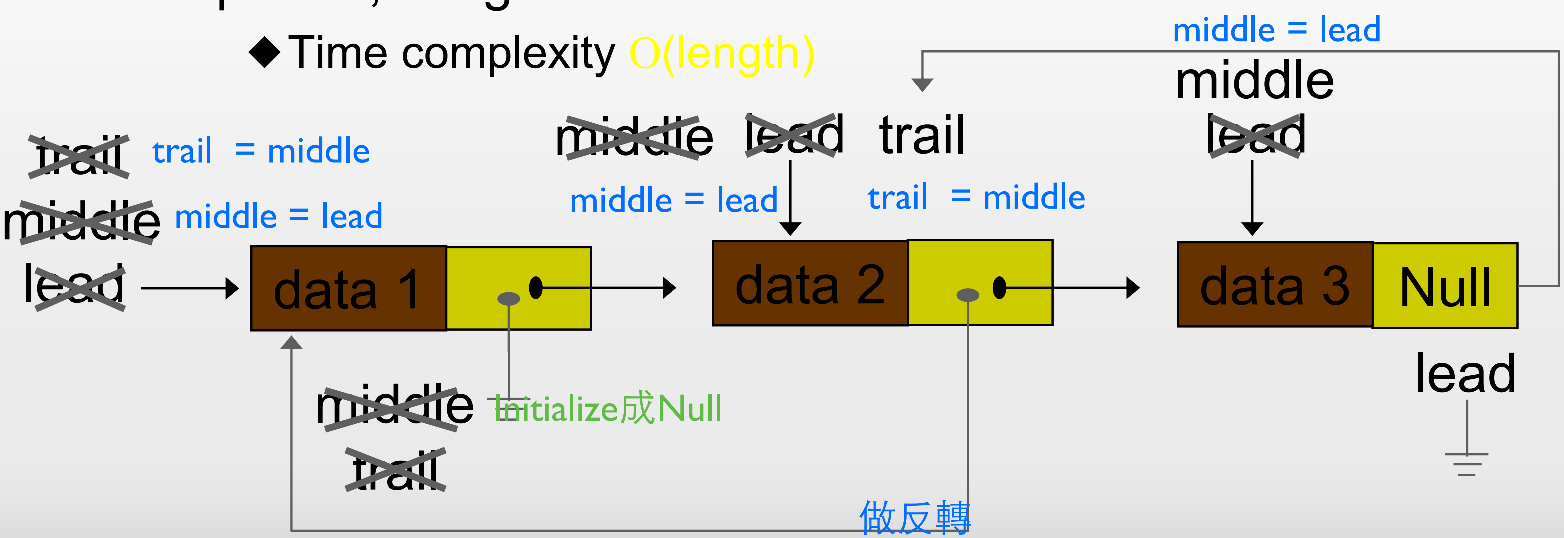
Program 4.10: Attach a node to the end of a list

```
listPointer invert(listPointer lead)
{/* invert the list pointed to by lead */
  listPointer middle,trail;
  middle = NULL;
  while (lead) {
    trail = middle;
    middle = lead;
    lead = lead→link;
    middle→link = trail;
  }
  return middle;
}
```

Program 4.16: Inverting a singly linked list

Additional List Operations

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



Equivalence Relations

同一個族群

- **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: $x \equiv x$ 自身性：自己和自己是同一群
  - Symmetric: $x \equiv y \rightarrow y \equiv x$ 對稱性
  - Transitive: $x \equiv y$ and $y \equiv z \rightarrow x \equiv z$ 遞移性

皆滿足即為 equivalence relation

- 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 分群：決定群數與其成員
  - Phase 1: Read in and store the equivalence pairs <i, j>
    - p. 176, Fig. 4.16 first 表示成seq[ ]
  - 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: $O(m + n)$

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

```
void equivalence()
{
   initialize seq to NULL and out to TRUE;
   while (there are more pairs) {
      read the next pair, <i,j>;
      put j on the seq[i] list;
      put i on the seq[j] list;
   }
   for (i = 0; i < n; i++)
      if (out[i]) {
         out[i] = FALSE;
         output this equivalence class;
      }
}
```

Program 4.21: A more detailed version of the equivalence algorithm

```
#include <stdio.h>
#include <alloc.h>
#define MAX_SIZE 24
#define FALSE 0
#define TRUE 1
typedef struct node *nodePointer;
typedef struct node {
        int data;
        nodePointer link;
        };
void main(void)
{
  short int out[MAX_SIZE];
  nodePointer seq[MAX_SIZE];
  nodePointer x,y,top;
  int i,j,n;

  printf("Enter the size (<= %d) ",MAX_SIZE);
  scanf("%d",&n);
  for (i = 0; i < n; i++) {
  /* initialize seq and out */
    out[i] = TRUE;   seq[i] = NULL;
  }

  /* Phase 1: Input the equivalence pairs: */
  printf("Enter a pair of numbers (-1 -1 to quit): ");
  scanf("%d%d",&i,&j);
  while (i >= 0) {
    MALLOC(x, sizeof(*x));
    x→data = j;  x→link = seq[i];  seq[i] = x;
    MALLOC(x, sizeof(*x));
    x→data = i;  x→link = seq[j];  seq[j] = x;
    printf("Enter a pair of numbers (-1 -1 to quit): ");
    scanf("%d%d",&i,&j);
  }
```

Two node insertions

```
    /* Phase 2: output the equivalence classes */
    for (i = 0; i < n; i++)
       if (out[i]) {
          printf("\nNew class: %5d",i);
          out[i] = FALSE;    /* set class to false */
          x = seq[i];  top = NULL; /* initialize stack */
          for (;;) {      /* find rest of class */
             while (x) {  /* process list */
                j = x→data;
                if (out[j]) {
                   printf("%5d",j);  out[j] = FALSE;
                   y = x→link; x→link = top; top = x; x = y;
                }
                else x = x→link;
             }
             if (!top) break;
             x = seq[top→data]; top = top→link;
                                        /* unstack */
          }
       }
}
```

Preparing to process stack i with top element seq[i]

The initialization of the stack for class i

push z into the stack of class i

Program 4.22: Program to find equivalence classes

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.

Figure 4.17: Node structure for sparse matrices

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

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

Figure 4.18: 4×4 sparse matrix a

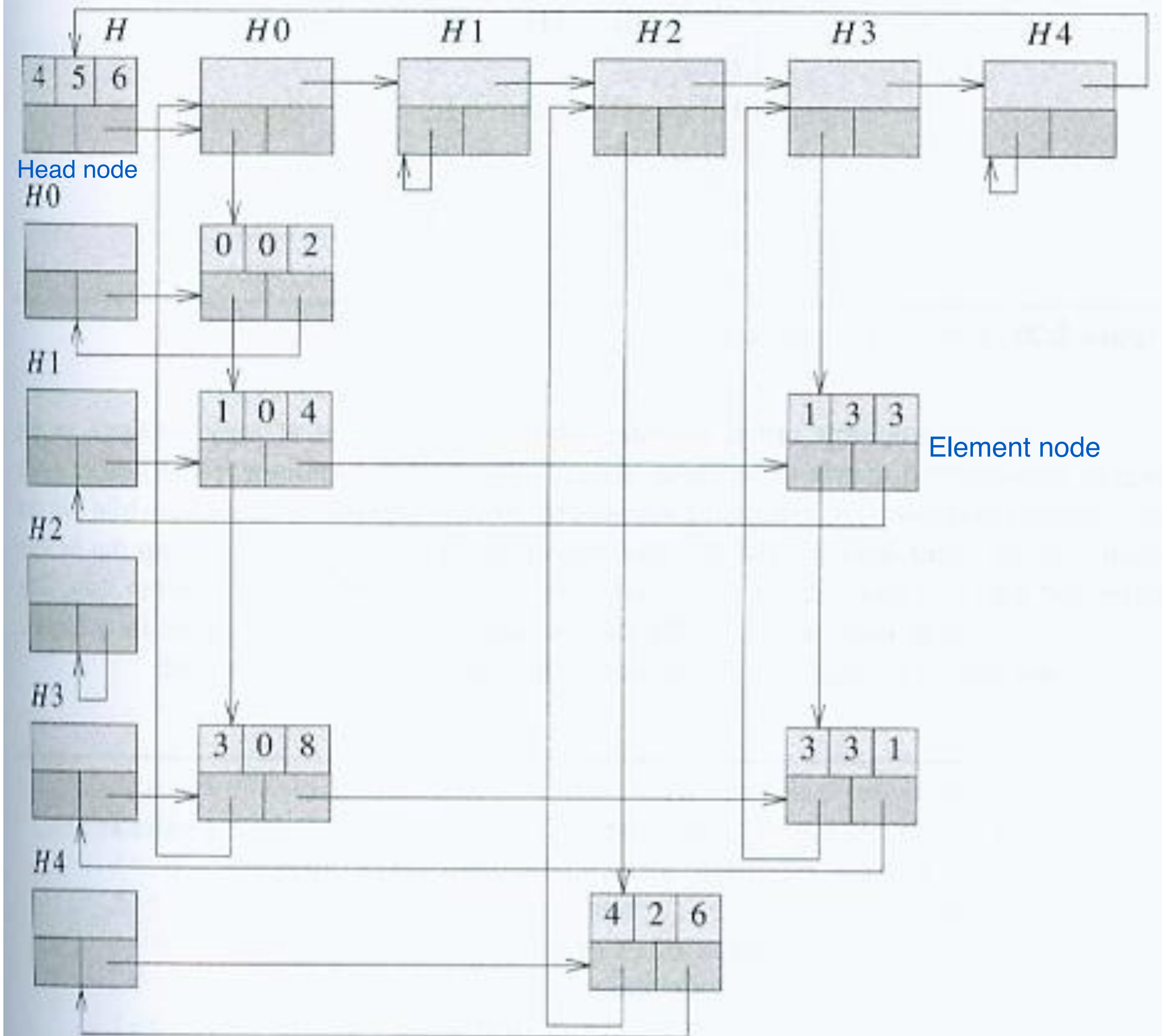


Figure 4.19: Linked representation of the sparse matrix of Figure 4.18 (the *head* field of a node is not shown)

Doubly Linked Lists

安全性較singly linked list佳

- 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)

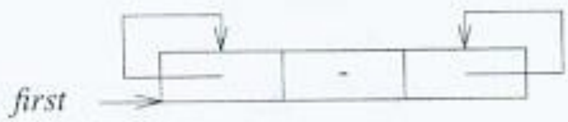


Figure 4.22: Empty doubly linked circular list with header node

```
void dinsert(nodePointer node, nodePointer newnode)
{/* insert newnode to the right of node */
   newnode→llink = node;
   newnode→rlink = node→rlink;
   node→rlink→llink = newnode;
   node→rlink = newnode;
}
```

Program 4.26: Insertion into a doubly linked circular list

```
void ddelete(nodePointer node, nodePointer deleted)
{/* delete from the doubly linked list */
  if (node == deleted)
    printf("Deletion of header node not permitted.\n");
  else {
    deleted→llink→rlink = deleted→rlink;
    deleted→rlink→llink = deleted→llink;
    free(deleted);
  }
}
```

Program 4.27: Deletion from a doubly linked circular list