연결 리스트 예제 2-1

- 0~20사이의 난수 10개를 생성하고 연결 리스트로 연결하여 출력하는 프로그램을 작성하시오.

```
 20  18   4  16  19   3  13   5  20  19
계속하려면 아무 키나 누르십시오 . . .
```

```
 12  14  18   1  11   3  10   6   9   0
계속하려면 아무 키나 누르십시오 . . .
```

```c
typedef struct node* LINK;
typedef struct node {
    int num;
    LINK next;
}NODE;
```

```c
#define NUM 10
```

```c
struct node {
    int num;
    struct node* next;
};

typedef struct node NODE;
typedef NODE* LINK;
```

```c
void printList(LINK);
LINK appendNode(LINK, LINK);
LINK createNode(int);
```

```c
void main() {

    LINK head = NULL, cur;
    srand(time(NULL));
    for (int i = 0; i < NUM; i++) {
        cur = createNode(rand() % 21);
        head = appendNode(head, cur);
    }
    printList(head);
    puts("");
}

LINK appendNode(LINK head, LINK cur) {
    LINK nextnode = head;
    if (!head) {
        head = cur;
        return head;
    }
    while (nextnode->next) {
        nextnode = nextnode->next;
    }
    nextnode->next = cur;
    return head;
}
```

```c
void printList(LINK head) {
	LINK nextnode = head;
	while (nextnode) {
		printf("%4d", nextnode->num);
		nextnode = nextnode->next;
	}
}

LINK createNode(int number) {
	LINK cur = (NODE*)malloc(sizeof(NODE));
	cur->num = number;
	cur->next = NULL;

	return cur;
}
```

연결 리스트 예제 **2-2**

- 2-1 예제에서 입력받은 수 이상의 원소들을 출력하는 프로그램을 작성하시오.

```
  20   17   11   12   18    5    5    6    8   15
기준 숫자 입력:10
10보다 큰 수들은...
  20   17   11   12   18   15입니다.
```

```
  12    0   12   17   18    5    2    7   10   18
기준 숫자 입력:20
20보다 큰 수들은...
없습니다.
```

```c
void printgreatnumber(LINK head, int number) {
	LINK nextnode = head;
	int flag = 0;
	printf("%d보다 큰 수들은...\n", number);
	while (nextnode) {
		if (nextnode->num > number) {
			printf("%4d", nextnode->num);
			flag = 1;
		}
		nextnode = nextnode->next;
	}
	if (flag) {
		printf("입니다.\n");
	}
	else
		printf("없습니다.\n");

}
```

연결 리스트 예제 2-3

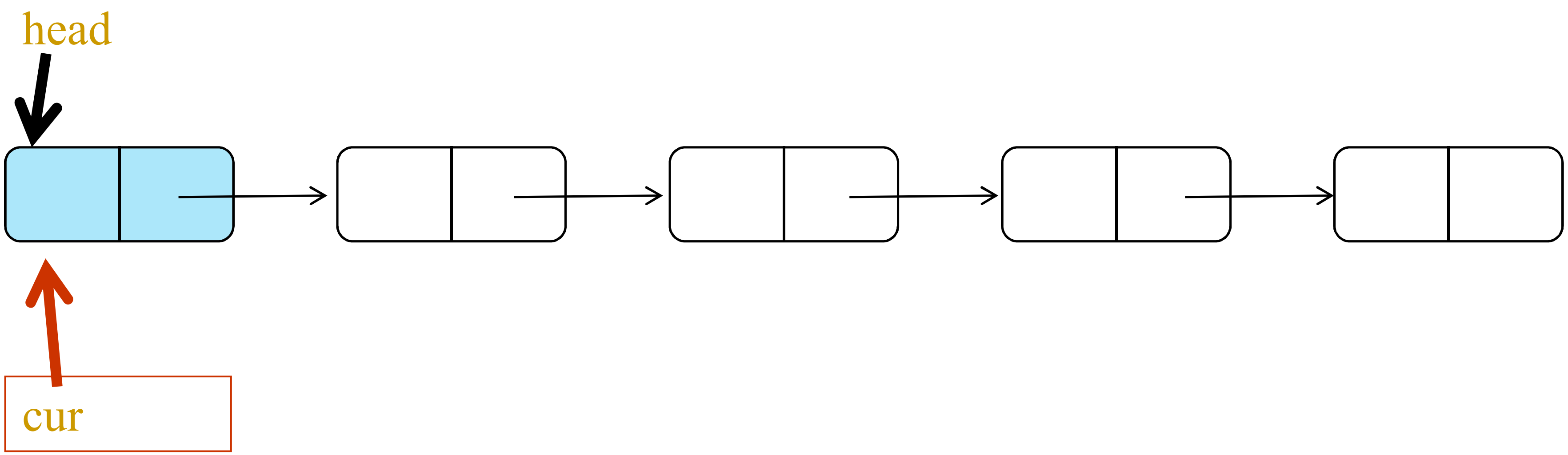
- 2-1 예제에서 지정한 노드를 삭제하고, 지정한 노드를 특정 노드 다음으로 삽입하는 프로그램을 작성하시오.
- delete: 두 개의 node pointer 필요
  - 현재노드, 이전노드
  - 삭제할 노드가 리스트의 head인 경우와 그렇지 않은 경우로 나누어 생각.

```
현재 리스트...
  11   2  19   9  20  13   0  11   4  11
삭제할 데이터 입력:20
삭제한 결과...
  11   2  19   9  13   0  11   4  11
삽입할 위치, 삽입할 데이터 입력:9 19
  11   2  19   9  19  13   0  11   4  11
```

```c
LINK deleteNode(LINK head, int key) {

    LINK cur = head, prev = NULL;

    if (cur->key==key){

        free(cur);
        return head;
    }
```

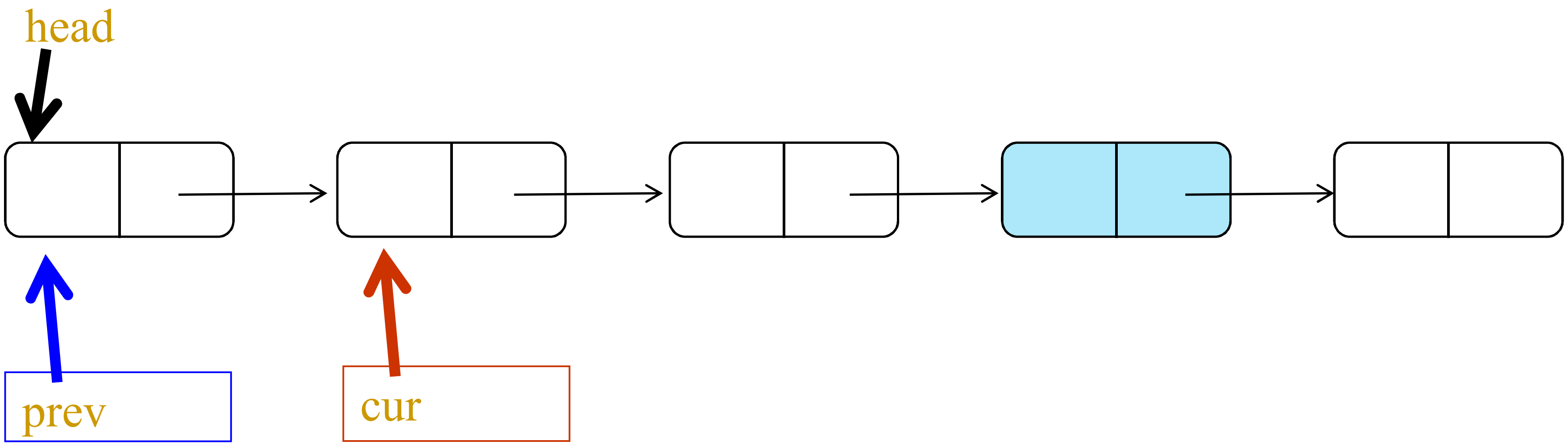
현재 리스트...
15 10 20 19 19 13 18 17 12 14
삭제할 데이터 입력:15

삭제한 결과...
10 20 19 19 13 18 17 12 14



```c
while (cur->next) {
    prev = cur;
    cur = cur->next;
    if (cur->key == key) {
        dd
        free(cur);
        return head;
    }
}
```

```
현재 리스트...
  10    6    7    2   14   14   19    9   12    5
삭제할 데이터 입력:2

삭제한 결과...
  10    6    7   14   14   19    9   12    5
```



```c
printf("₩n삽입할 위치, 삽입할 데이터 입력:");
scanf("%d %d", &prev, &key);
cur = createNode(key);
insertNode(head, cur, prev);
printList(head);
puts("");
```

```
삭제한 결과...
  11   2  19   9  13   0  11   4  11
삽입할 위치, 삽입할 데이터 입력:9 19
  11   2  19   9  19  13   0  11   4  11
```

* 삽입 위치가 가장 앞인 경우는 빠져있음.

```c
LINK insertNode(LINK head, LINK cur, int prev) {
    LINK nextnode = head;
    while (nextnode) {
        if (nextnode->num == prev) {
            cur->next = nextnode->next
            nextnode->next = cur;
            break;
        }
        nextnode = nextnode->next;
    }
    return head;
}
```

Linked Lists

Chapter 4

Contents

4.1 Singly Linked Lists

4.2 Representing Chains in C

4.3 Linked Stacks and Queues

4.4 Polynomials

4.5 Additional List Operations

4.6 Equivalence Classes

4.7 Sparse Matrices

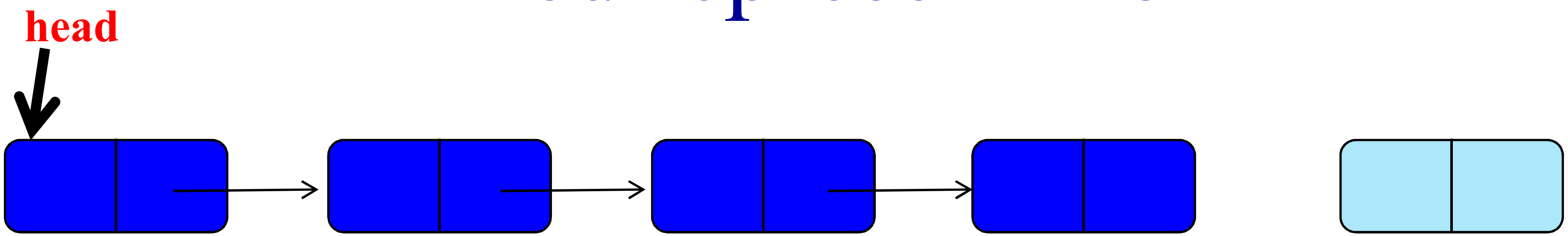
4.8 Doubly Linked Lists

Sequential representation

0	1	2	3	4	5	6	7	8
0	10	20	30	40	50	60	70	80

- successive nodes of the data object are stored a fixed distance apart
- order of elements is the same as in ordered list
- adequate for functions such as accessing an arbitrary node in a table
- operations such as insertion and deletion of arbitrary elements from ordered lists become expensive

Linked representation



- successive items of a list may be placed anywhere in memory
- order of elements need not be the same as order in list
- each data item is associated with a pointer (link) to the next item

Linked List Simulation with **Arrays**

- List of 3-letter words :(BAT, CAT, EAT, ..., VAT, WAT)

	data	*link*
1	HAT	15
2		
3	CAT	4
4	EAT	9
5		
6		
7	WAT	0
8	BAT	3
9	FAT	1
10		
11	VAT	7
	.	.
	.	.
	.	.

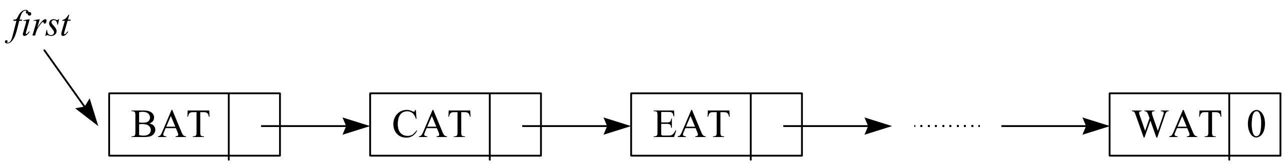
first = 8;

data[8]="BAT";

link[8]=3;

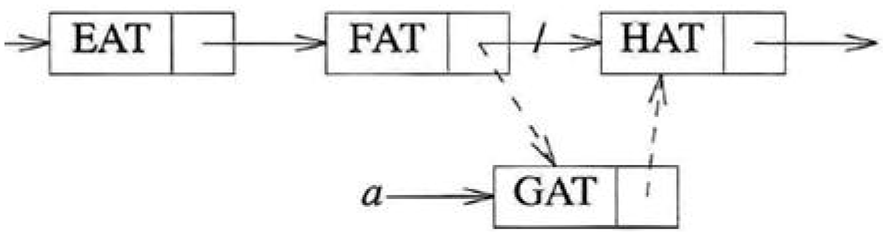
data[3] = "CAT";

….



Insertion (1)

- To insert GAT between FAT and HAT
  - (1) get a node N that is currently unused ; let its address be x
  - (2) set the data field of N to GAT
  - (3) set the link field of N to point to the node after FAT, which contains HAT
  - (4) set the link field of the node containing FAT to x



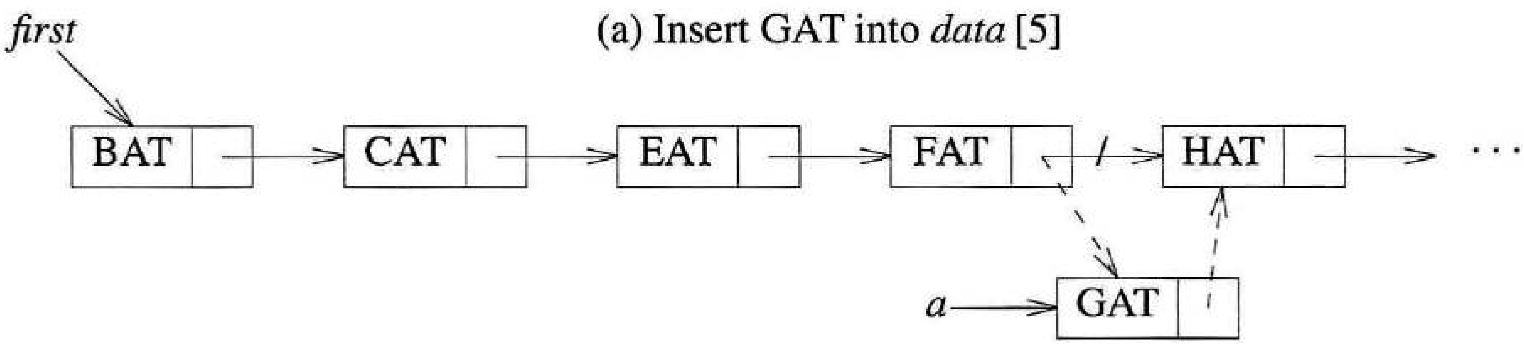
Insertion(2)

	data	*link*
1	HAT	15
2		
3	CAT	4
4	EAT	9
5		
6		
7	WAT	0
8	BAT	3
9	FAT	1
10		
11	VAT	7
	.	.
	.	.
	.	.

before

	data	*link*
1	HAT	15
2		
3	CAT	4
4	EAT	9
5	GAT	1
6		
7	WAT	0
8	BAT	3
9	FAT	5
10		
11	VAT	7

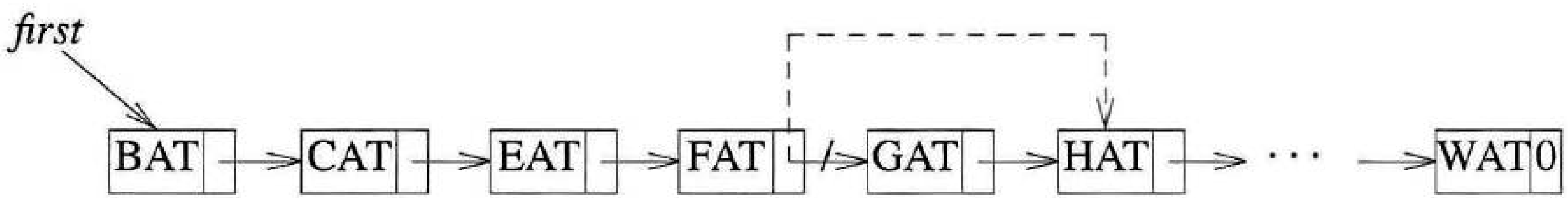
(a) Insert GAT into *data* [5]



(b) Insert node GAT into list

Deletion

- To delete GAT
  - (1) find the element that immediately precedes GAT, which is FAT
  - (2) set the link field of FAT to the position of HAT

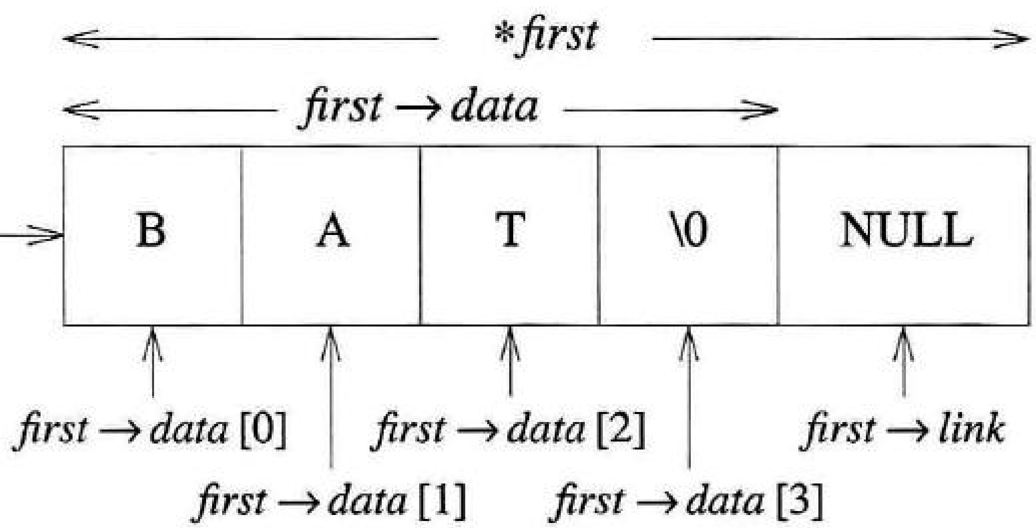


Representing Chains in C

```
typedef struct listNode *listPointer;
typedef struct listNode {
        char data[4];    //  int data;
        listPointer link;
} ;
```

struct listNode*

listNode



Create

```
typedef struct listNode *listPointer;
typedef struct listNode {
        int data;
        listPointer link;
        };
```

```
listPointer create2()
{/* create a linked list with two nodes */
  listPointer first, second;
  MALLOC(first, sizeof(*first));
  MALLOC(second, sizeof(*second));
  second→link = NULL;
  second→data = 20;
  first→data = 10;
  first→link = second;
  return first;
}
```

Program 4.1: Create a two-node list

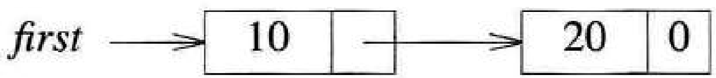


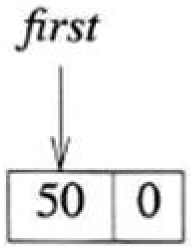
Figure 4.6: A two-node list

Insertion(1)

```
typedef struct listNode *listPointer;
typedef struct listNode {
        int data;
        listPointer link;
        };
```

```
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;
  }
}
```

empty list에 insert



Program 4.2: Simple insert into front of list

Insertion(2)

```
if (*first) {
   temp→link = x→link;
   x→link = temp;
}
```

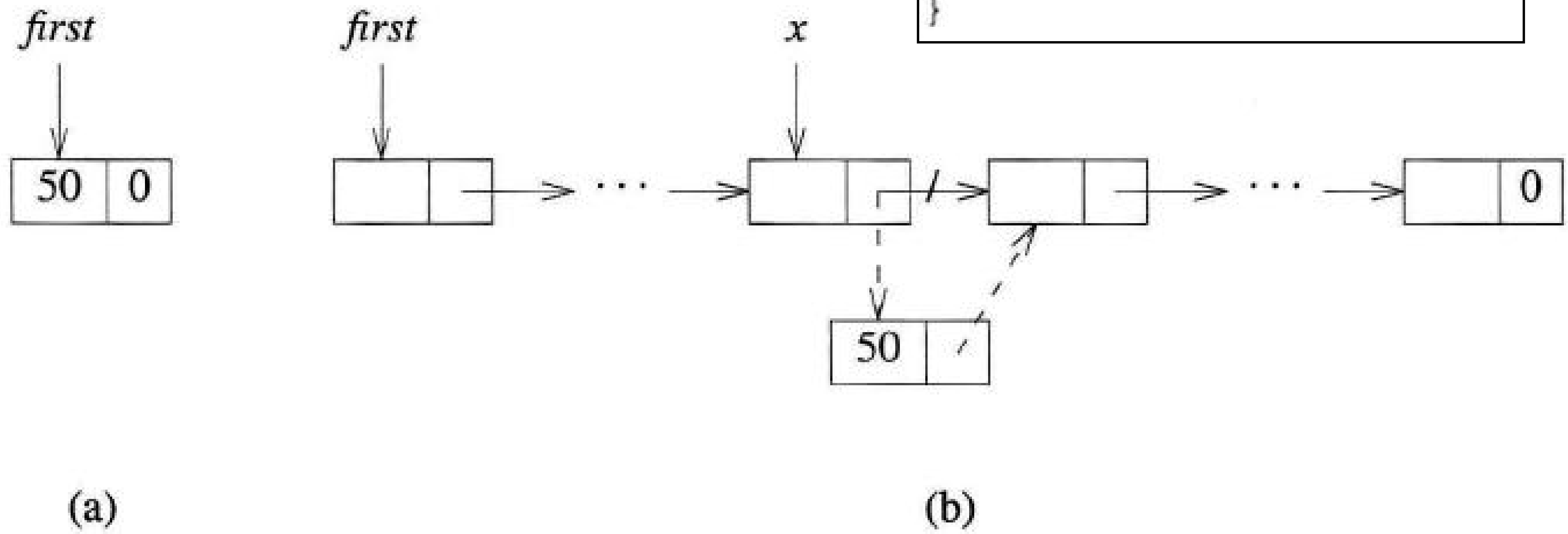


Figure 4.7: Inserting into an empty and nonempty list

Deletion(1)

```
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

Deletion(2)

```
*first = (*first)→link;
```

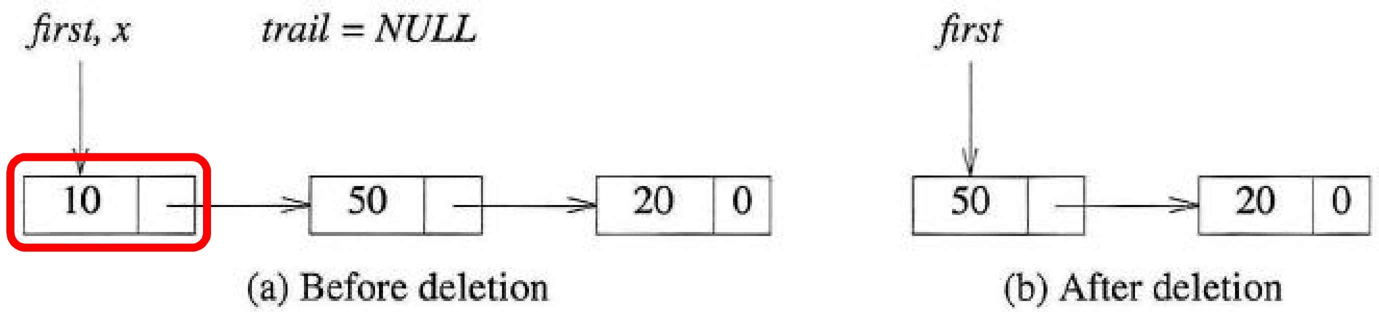


Figure 4.8: List before and after the function call *delete(&first, NULL, first);*

```
if (trail)
   trail→link = x→link;
```

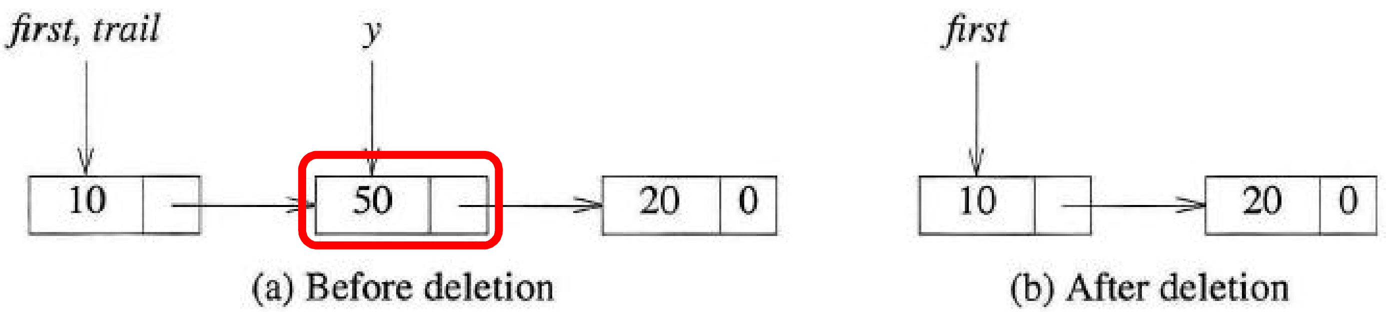


Figure 4.9: List after the function call delete(&first, trail, y);

```
void printList(listPointer first)
{
  printf("The list contains: ");
  for (; first; first = first→link)
    printf("%4d",first→data);
  printf("\n");
}
```

Program 4.4: Printing a list


```
void printList(LINK head) {
    LINK nextnode = head;
    while (nextnode) {
        printf("%4d", nextnode->num);
        nextnode = nextnode->next;
    }
}
```

Polynomials

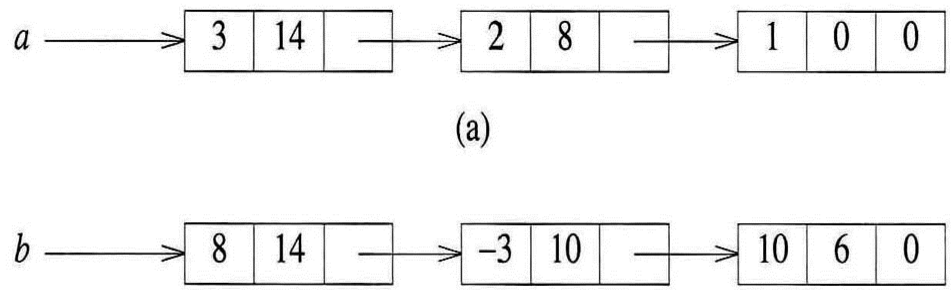
- Now, more efficient representation of Polynomials is possible using linked lists

```
typedef struct polyNode *polyPointer;
typedef struct polyNode {
      int coef;
      int expon;
      polyPointer link;
} ;
polyPointer a,b;
```

Examples:

$$a = 3x^{14} + 2x^{8} + 1$$

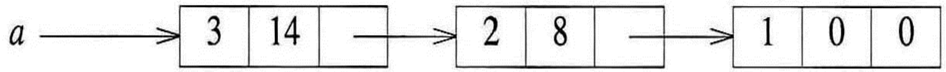
$$b = 8x^{14} - 3x^{10} + 10x^{6}$$



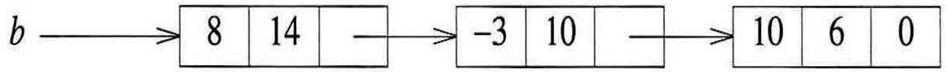
Examples:

$$a = 3x^{14} + 2x^{8} + 1$$

$$b = 8x^{14} - 3x^{10} + 10x^{6}$$



(a)

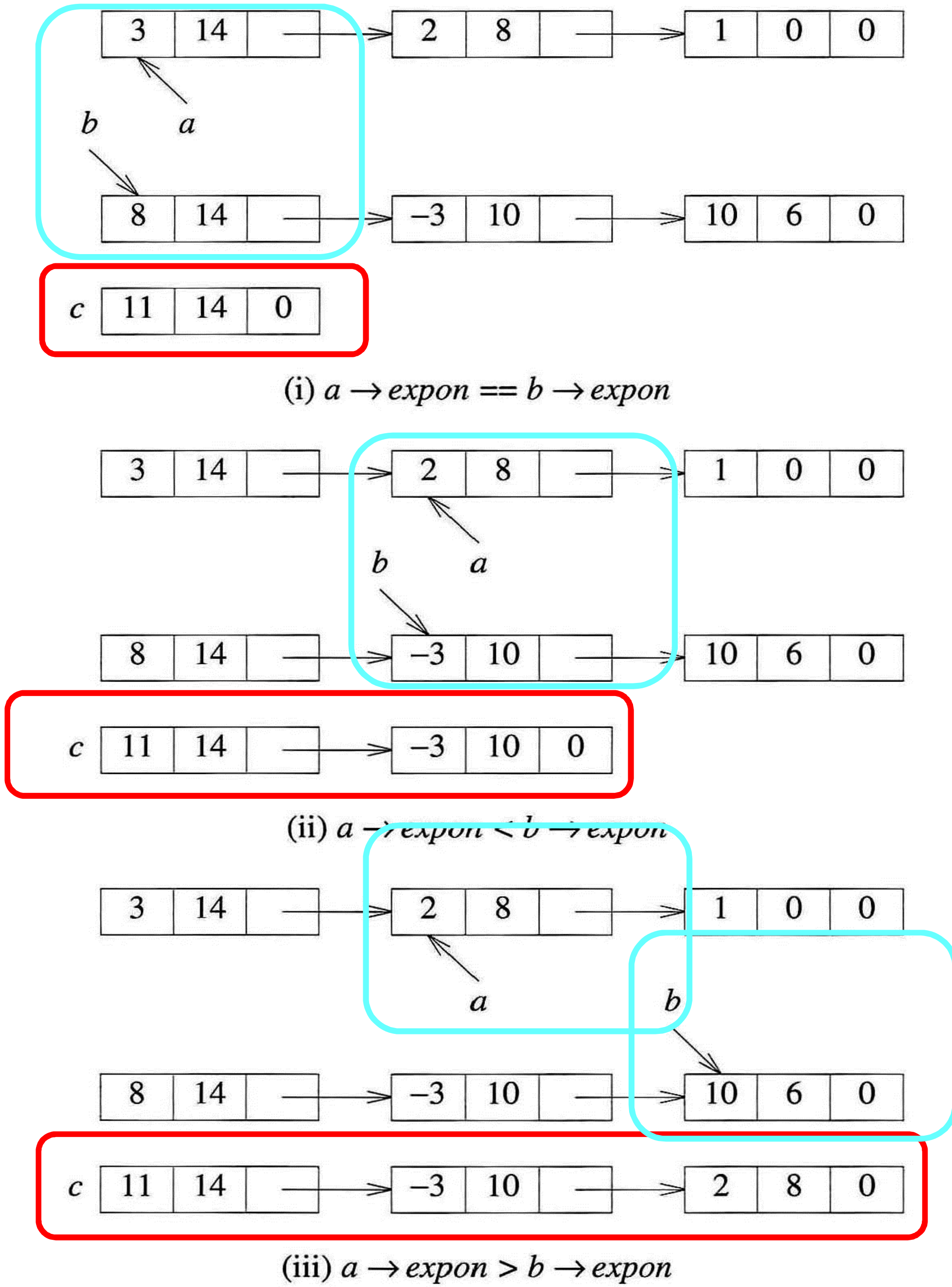


Adding polynomials

$$a = 3x^{14} + 2x^{8} + 1$$

$$b = 8x^{14} - 3x^{10} + 10x^{6}$$

c = a + b



Operations on Polynomials: add

```
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;
}
```

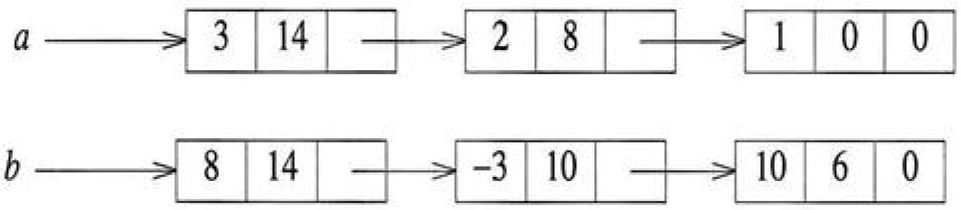
Program 4.9: Add two polynomials

```
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;
}
```

```
polyPointer padd(polyPointer a, polyPointer b)
{/* return a polynomial which is the sum of a and ]
   polyPointer c, rear, temp;
   int sum;
   MALLOC(rear, sizeof(*rear));
   c = rear;
   while (a && b)
```

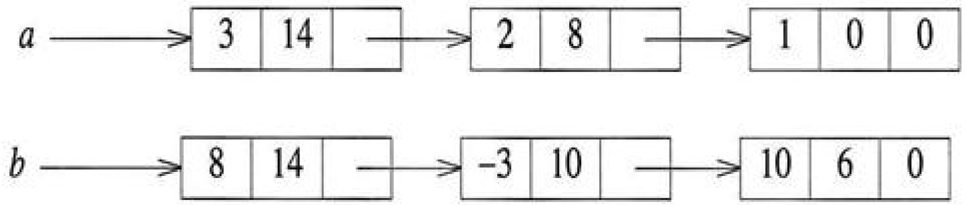
```
while (a && b)
   switch (COMPARE(a→expon,b→expon)) {
      case -1: /* a→expon < b→expon */
            attach(b→coef,b→expon,&rear);
            b = b→link;
            break;
      case 0: /* a→expon = b→expon */
            sum = a→coef + b→coef;
            if (sum) attach(sum,a→expon,&rear);
            a = a→link;  b = b→link; break;
      case 1: /* a→expon > b→expon */
            attach(a→coef,a→expon,&rear);
            a = a→link;
   }
```

```
void main(){
  polyPointer a, b, result;
   …
  result=padd(a, b);
```



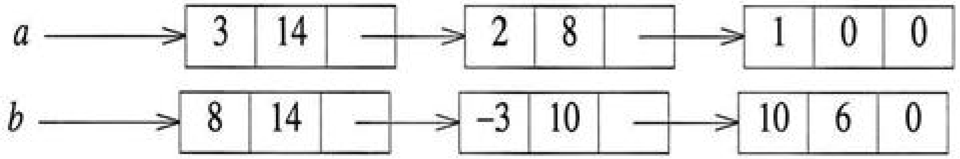

```
  /* copy rest of list a and then list b */
  for (; a; a = a→link) attach(a→coef,a→expon,&rear);
  for (; b; b = b→link) attach(b→coef,b→expon,&rear);
  rear→link = NULL;
  /* delete extra initial node */
  temp = c; c = c→link;  free(temp);
  return c;
}
```

이 부분이 필요한 이유는?




```
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;
   }
```

```
case 0: /* a→expon = b→expon */
     sum = a→coef + b→coef;
     if (sum) attach(sum,a→expon,&rear);
```




```
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

[참고]

- polyPointer attach(float co, int ex, polyPointer C_head){

  ….

  return C_head;

}

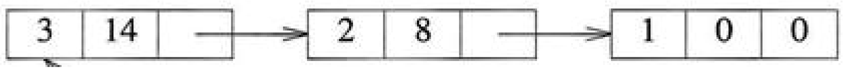
형태로 사용하면 padd의 rear, temp 불필요.

(C예제의 appendNode와 유사하게 구현하면 됨)

Operations on Polynomials: erase

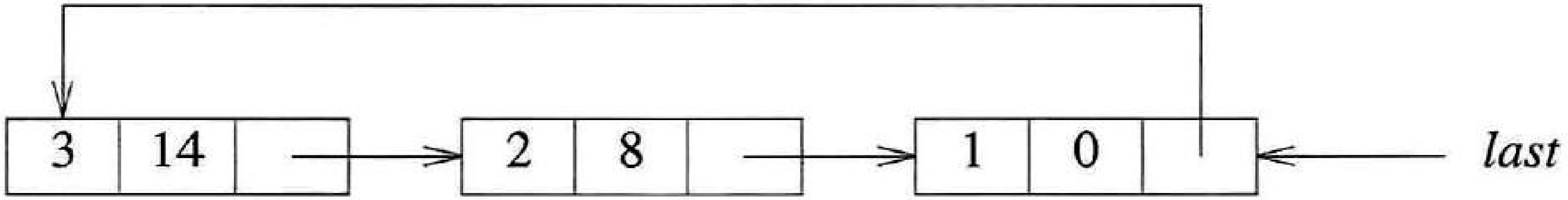
```
void erase(polyPointer *ptr)
{/* erase the polynomial pointed to by ptr */
     polyPointer temp;
     while ( *ptr) {
          temp = *ptr;
          *ptr = (*ptr)->link;
          free (temp);
     }
}
```

erase(head)



Circular List Representation of Polynomials

$3x^{14} + 2x^{8} + 1$



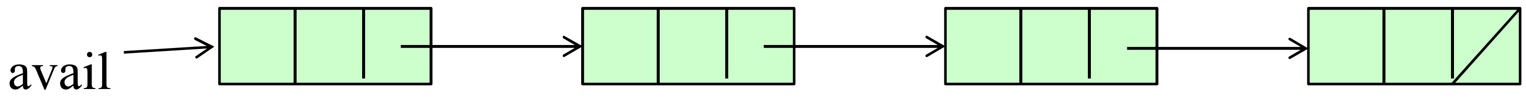
- cf. A singly linked list in which the last node has a null link: *chain*

Circular linked list를 이용한 삭제 – 변수 avail사용

- free된 node를 따로 관리하여 이용.
  - malloc/free의 잦은 이용 자제
- get_node()
  - malloc()대신 사용
  - avail이 가리키는 chain으로부터 하나의 노드 사용(chain이 empty이면 malloc())
- ret_node()
  - free()대신 사용
  - 삭제될 노드를 변수 avail이 가리키는 chain에 추가
- cerase()
  - polynomial삭제
  - 삭제될 circular list 전체를 변수 avail이 가리키는 chain에 추가
  - O(1)

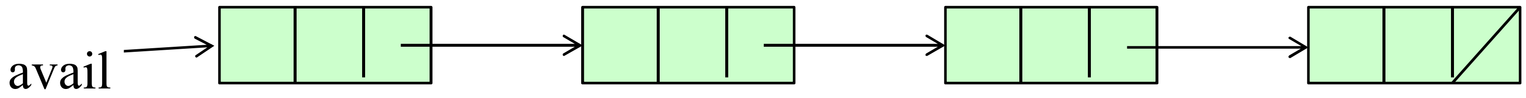
```
polyPointer getNode(void)
{/* provide a node for use */
   polyPointer node;
   if (avail) {
      node = avail;
      avail = avail→link;
   }
   else
      MALLOC(node, sizeof(*node));
   return node;
}
```

Program 4.12: *getNode* function



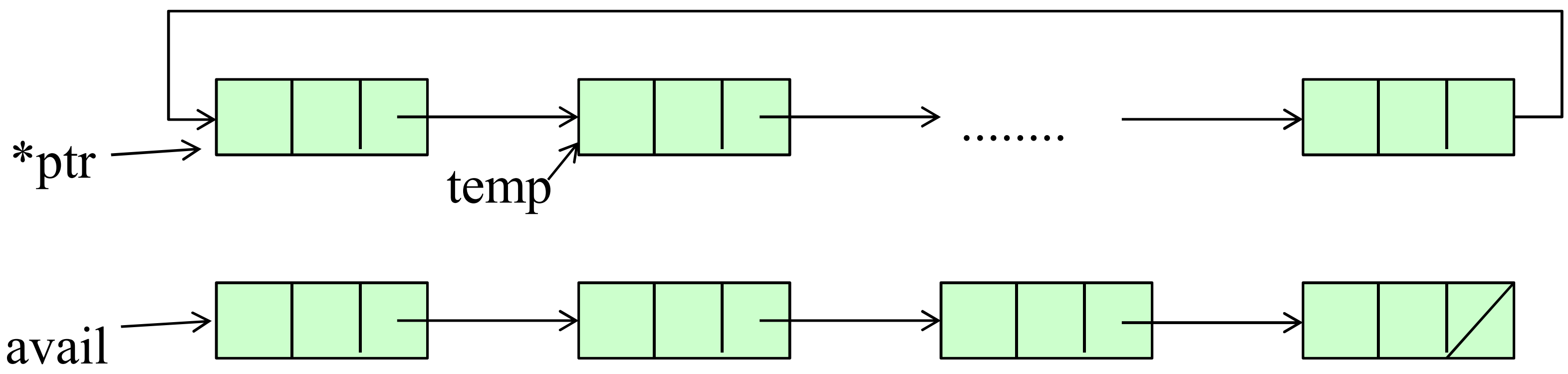
```
void retNode(polyPointer node)
{/* return a node to the available list */
   node→link = avail;
   avail = node;
}
```

Program 4.13: *retNode* function



```
void cerase(polyPointer *ptr)
{/* erase the circular list pointed to by ptr */
   polyPointer temp;
   if (*ptr) {
      temp = (*ptr)→link;
      (*ptr)→link = avail;
      avail = temp;
      *ptr = NULL;
   }
}
```

Program 4.14: Erasing a circular list



Additional list operations

- Operations for chains

```
typedef struct listNode *listPointer;
typedef struct listNode
      char data;
      listPointer link;
} ;
```

```
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;
}
```

```
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;
}
```

trail은 middle 바로 전의 노드를 가리킴.

middle은 invert할 대상인 노드를 가리킴.

lead는 아직 invert되지 않는 리스트의 시작을 가리킴.

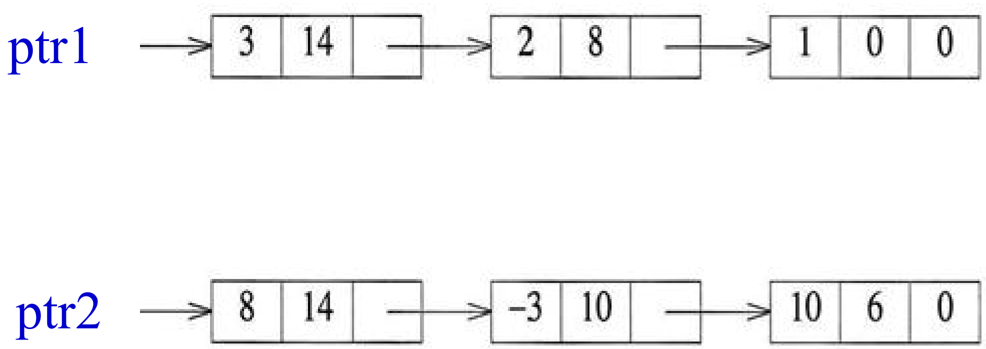
Additional list operations(1)

- Operations for chains: concatenate

```
listPointer concatenate(listPointer ptrl, listPointer ptr2)
{/* produce a new list that contains the list ptrl followed by the list
    ptr2. The list pointed to by ptrl is changed permanently */
    listPointer temp;
    /* check for empty lists */
    if (!ptrl) return ptr2;
    if (!ptr2) return ptrl;
    /* neither list is empty, find end of first list */
    for (temp = ptrl; temp->link; temp = temp->link);
    /* link end of first to start of second */
    temp->link = ptr2;
    return ptr1;
}
```

head=concatenate(ptr1, ptr2);

```
if (!ptrl) return ptr2;
if (!ptr2) return ptrl;
/* neither list is empty, find end of first list */
for (temp = ptrl; temp->link; temp = temp->link);
/* link end of first to start of second */
temp->link = ptr2;
```

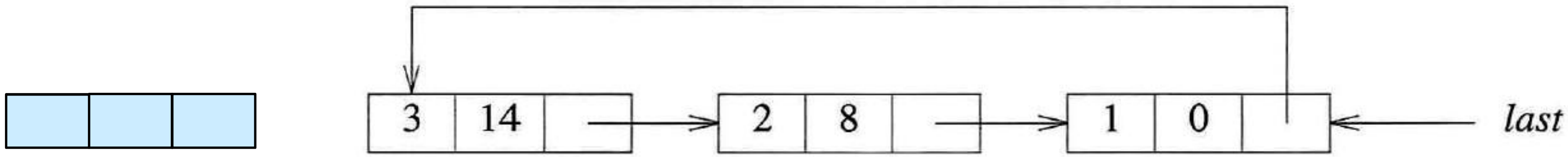


Additional list operations(2)

- Operations for Circularly Linked Lists

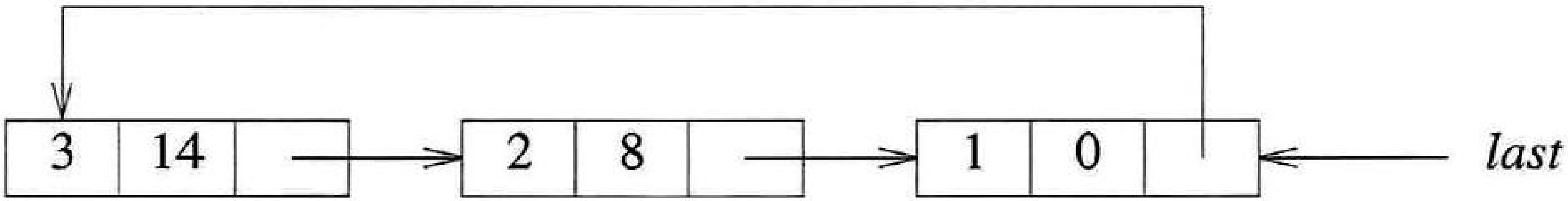
```
void insertFront(listPointer *last,
                                  listPointer node)
{
    if (!(*last)){
         *last = node;
         node->link = node;
    }
   else {
         node->link = (*last)->link;
         (*last)->link = node;
    }
}
```

```
int length(listPointer last)
{
       listPointer temp;
       int count = 0;
       if (last){
            temp = last;
            do {
                 count++;
                 temp = temp->link;
             }while (temp !=last);
        }
        return count;
}
```



insertFront(&mlast, node);

```
void insertFront(listPointer *last,
                                listPointer node)
{
   if (!(*last)){
        *last = node;
         node->link = node;
   }
  else {
         node->link = (*last)->link;
        (*last)->link = node;
   }
}
```




```
int length(listPointer last)
{
      listPointer temp;
      int count = 0;
      if (last){
           temp = last;
           do {
                count++;
               temp = temp->link;
            }while (temp !=last);
        }
       return count;
}
```

Doubly Linked Lists

- Limitation in chains and singly linked circular lists
  - The only way to find the node that precedes a node *p* is to start at the beginning of the list.
  - The same situation when we want to delete an arbitrary node in the list.
- It is useful to have doubly linked lists

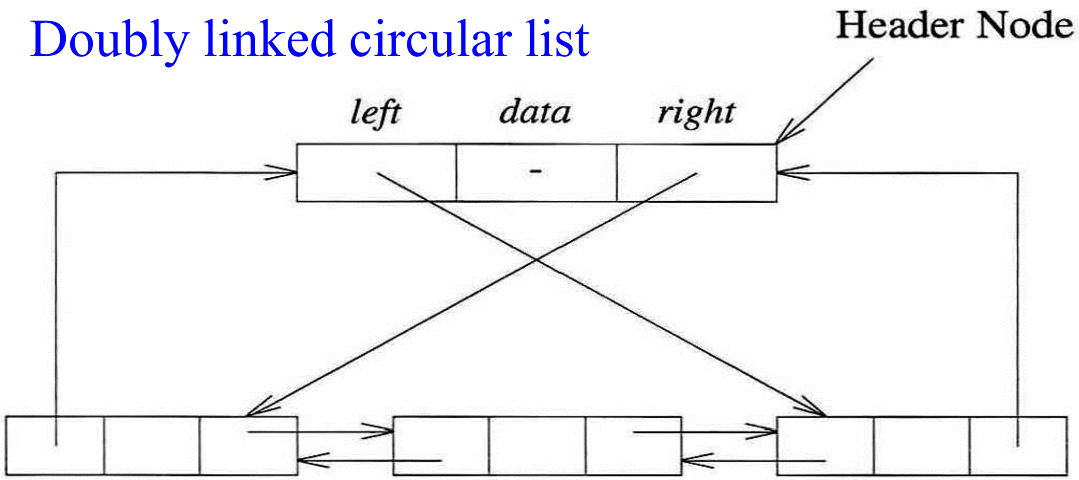
Doubly Linked Lists: node structure

```
typedef struct node *nodePointer;
typedef struct node {
        nodePointer llink;
        element data;
        nodePointer rlink;
        } ;
```

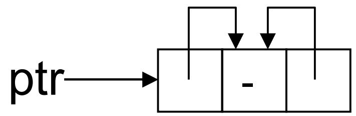
If *ptr* points to any node in a doubly linked list, then

ptr = <u>*ptr->llink->rlink*</u> = <u>*ptr->rlink->llink*</u>

Doubly linked circular list

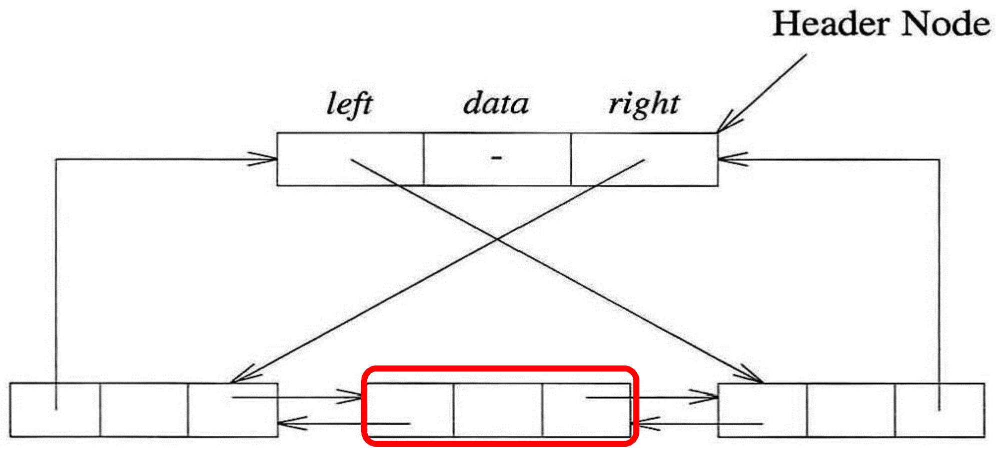


Empty doubly linked circular list



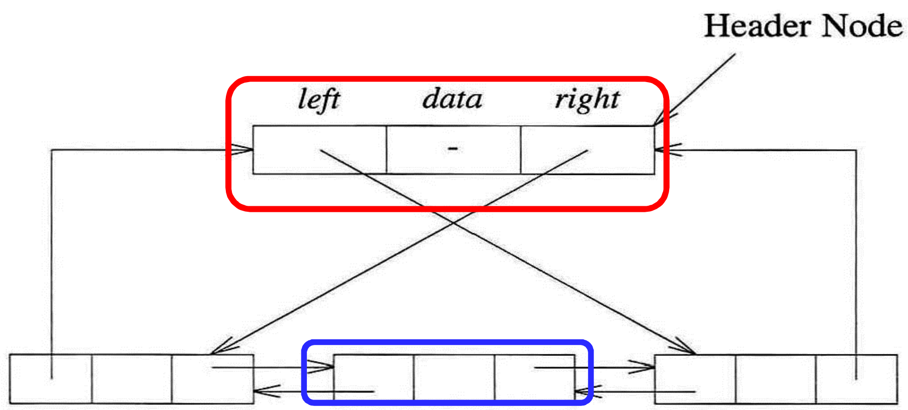
Doubly Linked Lists: operations (1)

```
void dinsert(nodePointer node,
             nodePointer newnode)
{
    newnode->llink = node;
    newnode->rlink = node->rlink;
    node->rlink->llink = newnode;
    node->rlink = newnode;
}
```

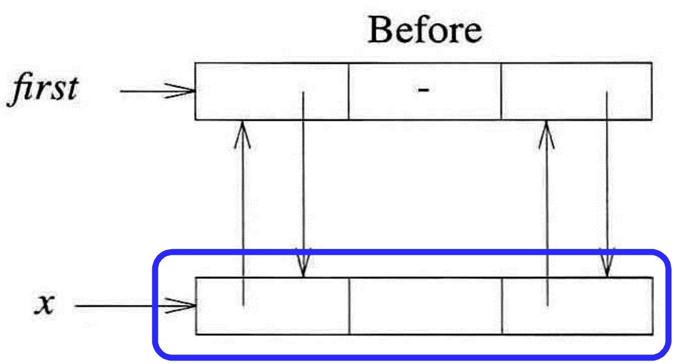


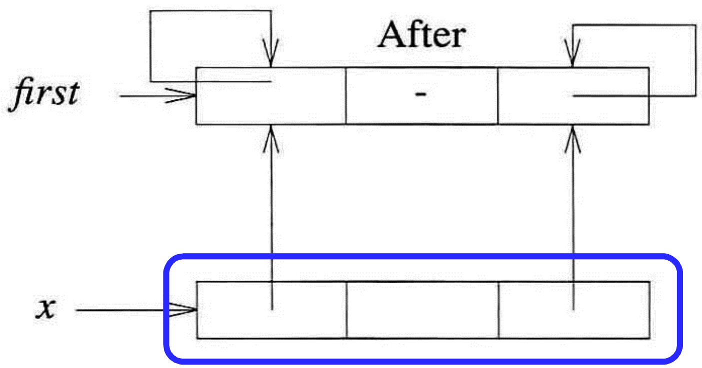
Doubly Linked Lists: operations (2)

```
void ddelete(nodePointer node,
             nodePointer deleted)
{
    if (node == deleted)
    printf("Deletion of header node not
                      permitted.\n");
    else {
        deleted->llink->rlink = deleted->rlink;
        deleted->rlink->llink = deleted->llink
        free(deleted);
    }
}
```



free(deleted)하기 직전





이전 stack, queue구현: array

- 만일 stack과 queue가 여러 개라면?

- stack1개: element stack[MAX]
- stack N개: element stack[N][MAX]??
  - top도 N개: int top[N]

Linked Stacks and Queues

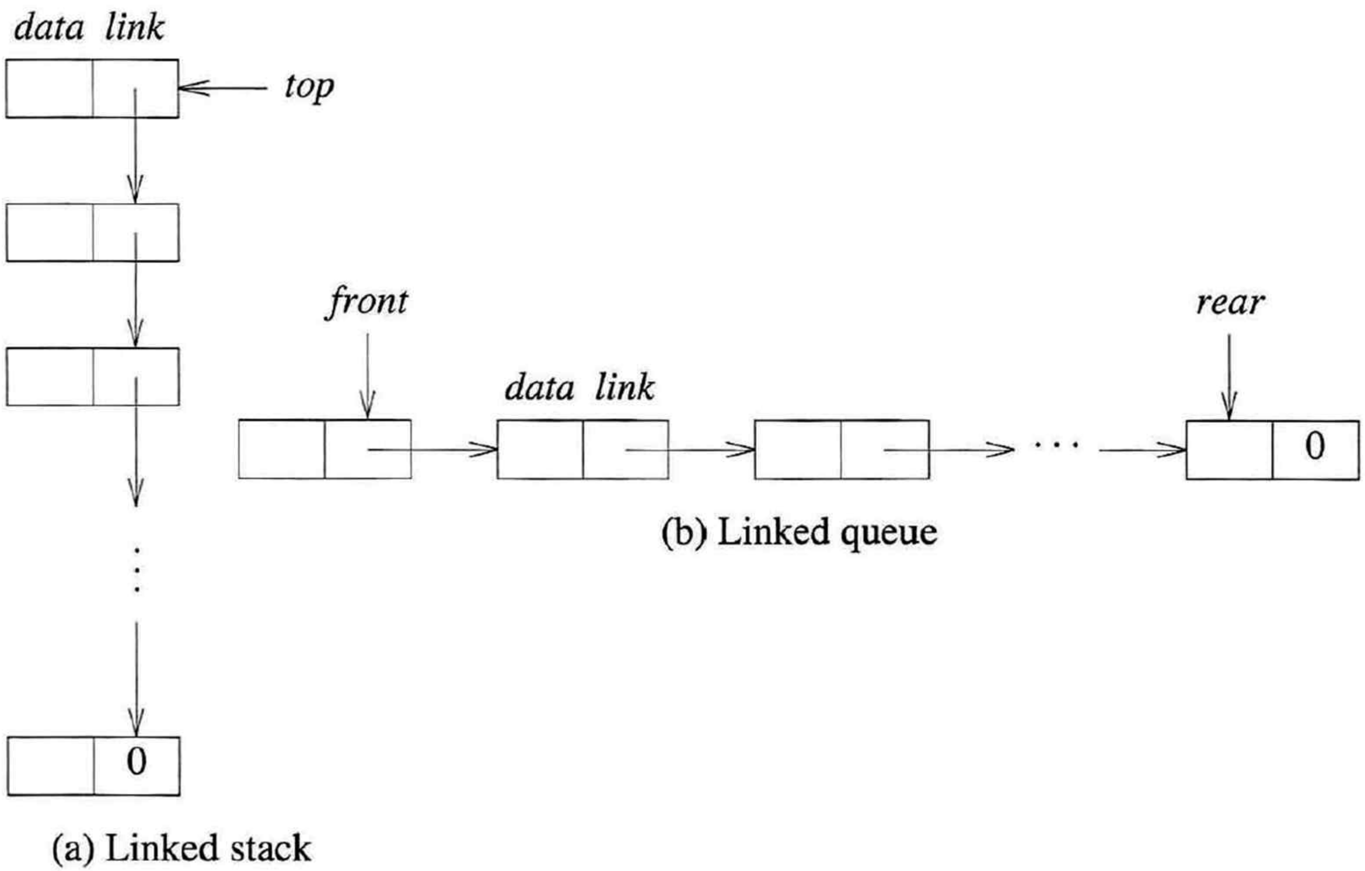


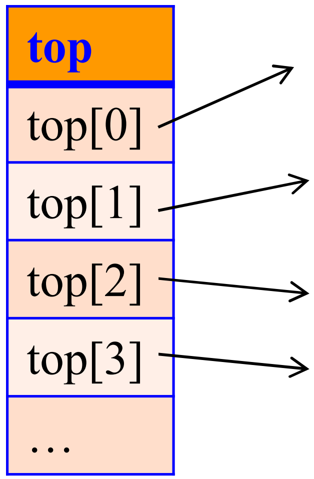
Figure 4.11: Linked stack and queue

Linked Stacks(1)

```
#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];
```

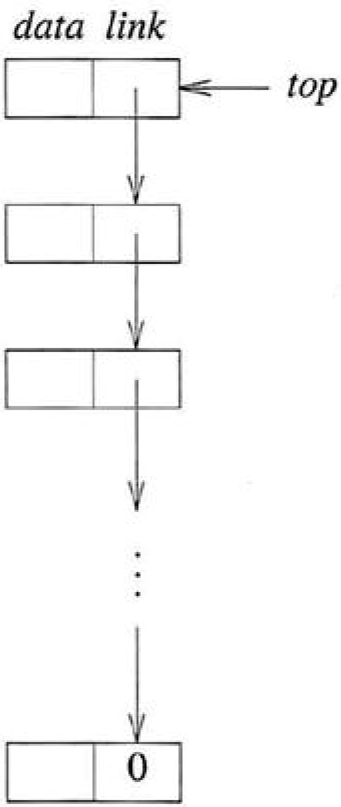
initially, top[i] ➔ NULL

top [*i*] = *NULL* iff the *i*th stack is empty



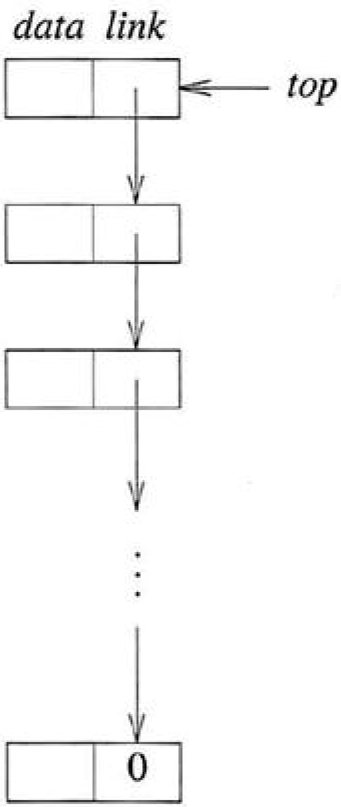
Linked Stacks(2)

```
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;
}
```



Linked Stacks(3)

```
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;
}
```



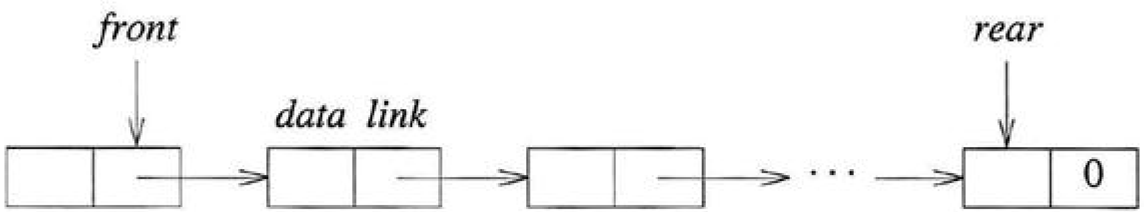
top [*i*] = *NULL* iff the *i*th stack is empty

Linked Queues (1)

```
#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];
```

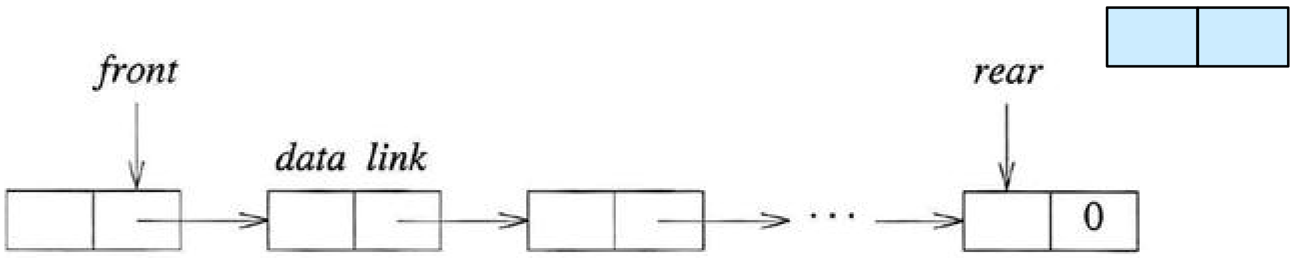
initially, front[i] ➔ NULL

front[*i*] = NULL iff the *i*th queue is empty



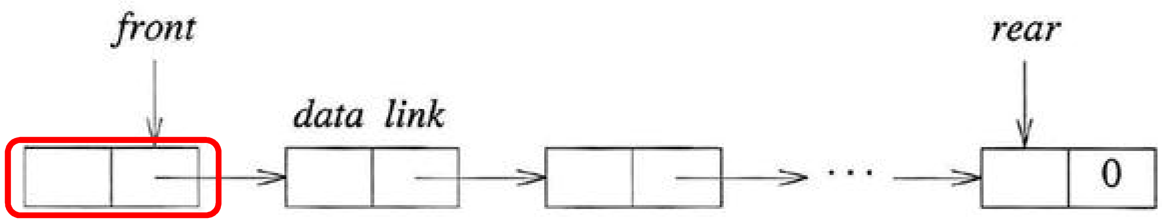
Linked Queues (2)

```
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;
}
```



Linked Queues (3)

```
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;
}
```



Equivalence Class (1)

- A relation '$\equiv$' over a set S is an equivalence relation over S iff it's symmetric, reflexive, and transitive over S
  - $x \equiv x$: reflexive
  - If $x \equiv y$ then $y \equiv x$: symmetric
  - If $x \equiv y$ and $y \equiv z$ then $x \equiv z$: transitive
- How about '$<$ ' (smaller than) and '$=$ ' (equals to)?

Equivalence Class (2)

- $S = \{0, 1, 2, \dots, 11\}$
- Given equivalence pairs

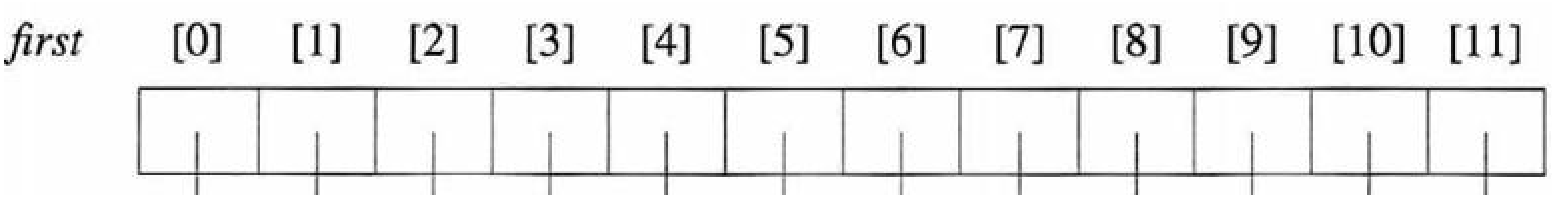
  $0 \equiv 4$, $3 \equiv 1$, $6 \equiv 10$, $8 \equiv 9$, $7 \equiv 4$, $6 \equiv 8$, $3 \equiv 5$, $2 \equiv 11$, $11 \equiv 0$
- Equivalence class:
  - $\{0, 2, 4, 7, 11\}$; $\{1, 3, 5\}$; $\{6, 8, 9, 10\}$

Application

- VLSI circuits
  - When exposing a silicon wafer using a series of masks, each mask consists of several polygons, where polygons that overlap electrically are equivalent

- S = {0, 1, 2, … , 11}
- Given equivalence pairs

  0 ≡ 4, 3 ≡ 1, 6 ≡ 10, 8 ≡ 9, 7 ≡ 4, 6 ≡ 8, 3 ≡ 5, 2 ≡ 11, 11 ≡ 0



Algorithm

- Input: equivalence pairs <i, j>, i, j $\in$S

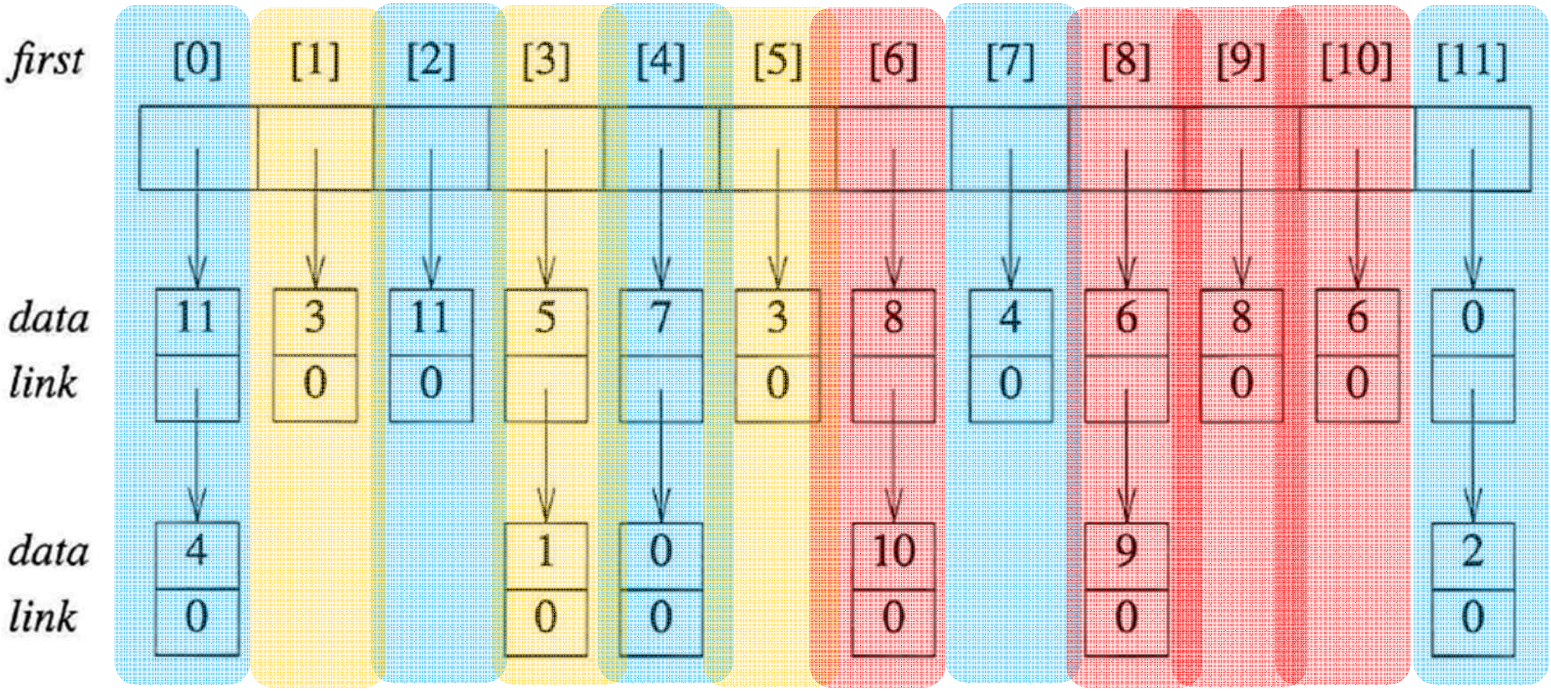
```
void equivalence()
{
   initialize seq to NULL and out to TRUE;
   while (there are more pairs) {
      read the next pair, <i,j>;
      put j on the seq[i] list;
      put i on the seq[j] list;
   }
   for (i = 0; i < n; i++)
      if (out[i]) {
         out[i] = FALSE;
         output this equivalence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

$0 \equiv 4$, $3 \equiv 1$, $6 \equiv 10$, $8 \equiv 9$, $7 \equiv 4$, $6 \equiv 8$, $3 \equiv 5$, $2 \equiv 11$, $11 \equiv 0$

Example

- Input

<0,4>, <3,1>, <6,10>, <8,9>, <7,4>, <6,8>, <3,5>, <2,11>, <11,0>



Program 4.22 (1)

```
#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;
  }
```

0	1	2	3	4	5	6	7	8	9	10	11

nodePointer seq[MAX]

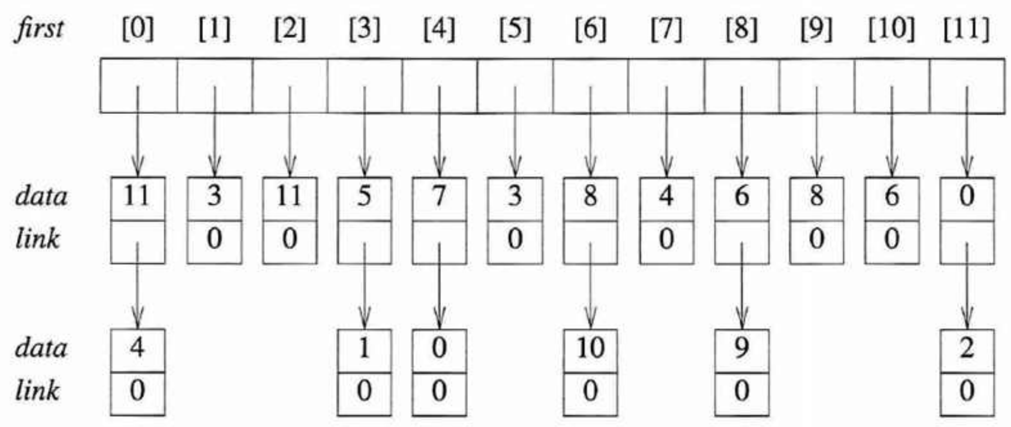
0	1	2	3	4	5	6	7	8	9	10	11

short int out[MAX]

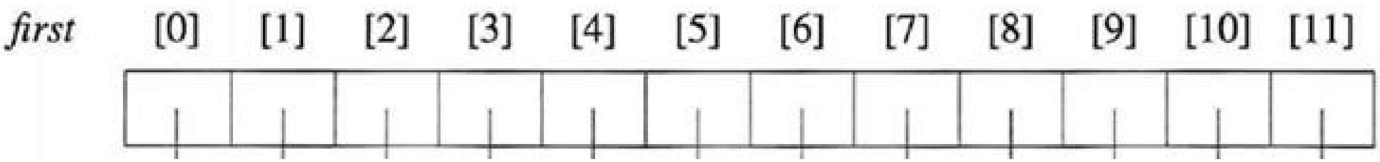
Program 4.22 (2)

```
/* 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);
}
```

<0,4>, <3,1>, <6,10>, <8,9>, <7,4>, <6,8>,
<3,5>, <2,11>, <11,0>

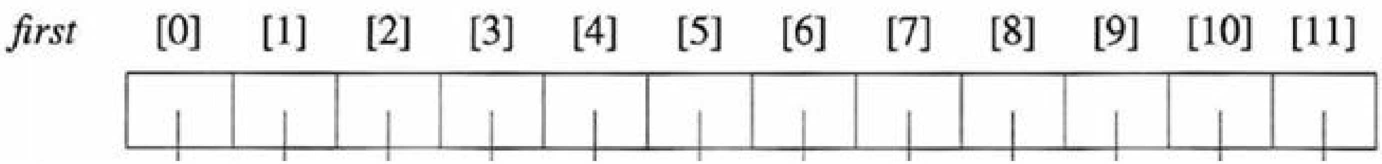


<0,4>, <3,1>, <6,10>, <8,9>, <7,4>, <6,8>,
<3,5>, <2,11>, <11,0>



```c
/* 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);
}
```

<0,4>, <3,1>, <6,10>, <8,9>, <7,4>, <6,8>, <3,5>, <2,11>, <11,0>

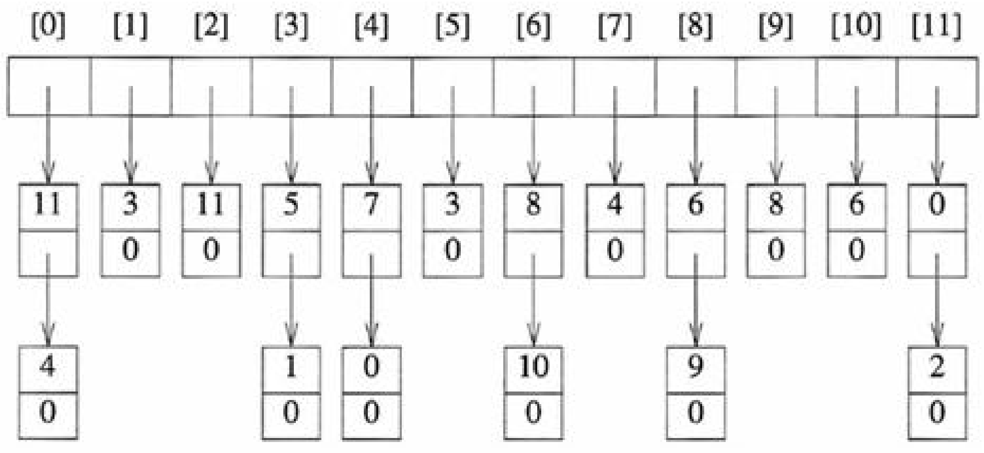


i	j
0	4
3	1
6	10
…	…
11	0

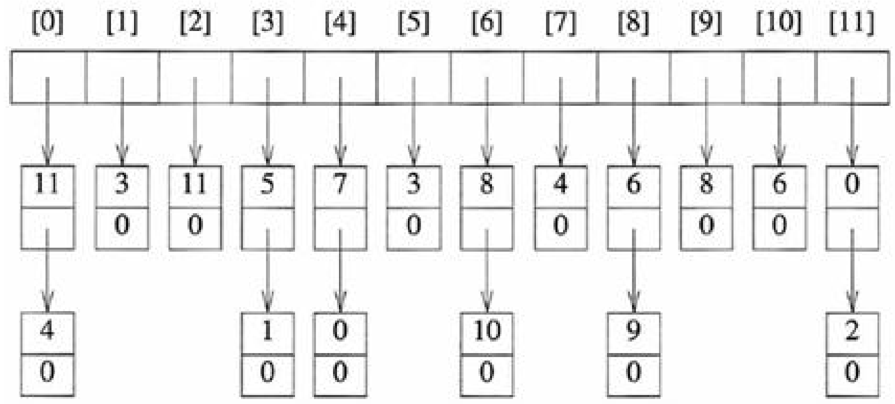
Program 4.22 (3)

```
  /* 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 */
        }
     }
}
```

Phase2 설명



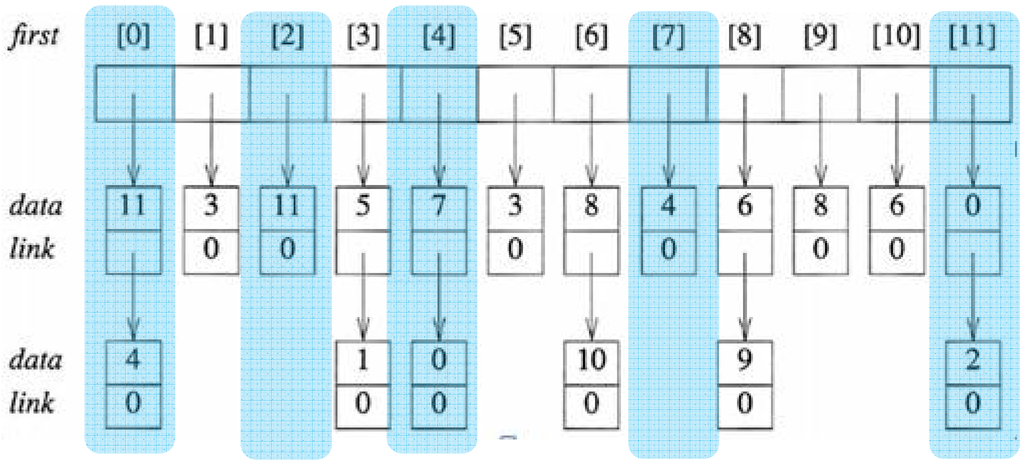
- 0부터 11까지 순차적으로 seq[i]탐색
- (예) seq[0]탐색시에 0->11->4 순서로 프린트하며 매번 각각의 out: out[0], out[11], out[4]를 false로.
  - 스택에 11->4의 순서로 push함.
  - (i) while(x)는, x=seq[i]였을 경우 seq[i]의 모든 연결된 노드 탐색이 끝나면 빠져나오며
  - (ii) 그때 stack에서 pop한 데이터j의 seq[j]를 다시 탐색.
  - (iii) stack이 비면 해당 equiv. class 출력이 완료된 것임.



0	1	2	3	4	5	6	7	8	9	10	11

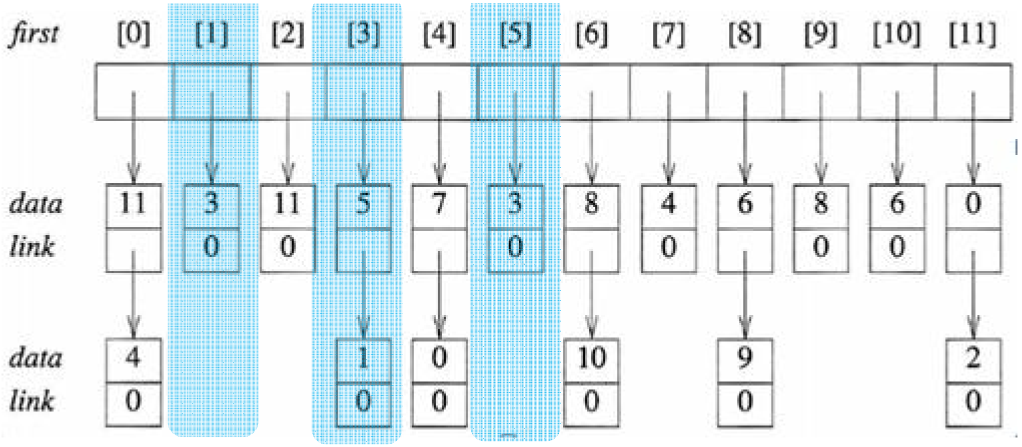
```
/* Phase 2: output the equivalence classes */
for (i = 0; i < n; i++)
  if (out[i]) {
    printf("\nNew class: %5d",i);
    out[i] = FALSE;    /* set class to false */
    x = seq[i];  top = NULL; /* initialize stack */
    for (;;) {        /* find rest of class */
      while (x) {  /* process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

stack에 x가 가리키는 노드 push



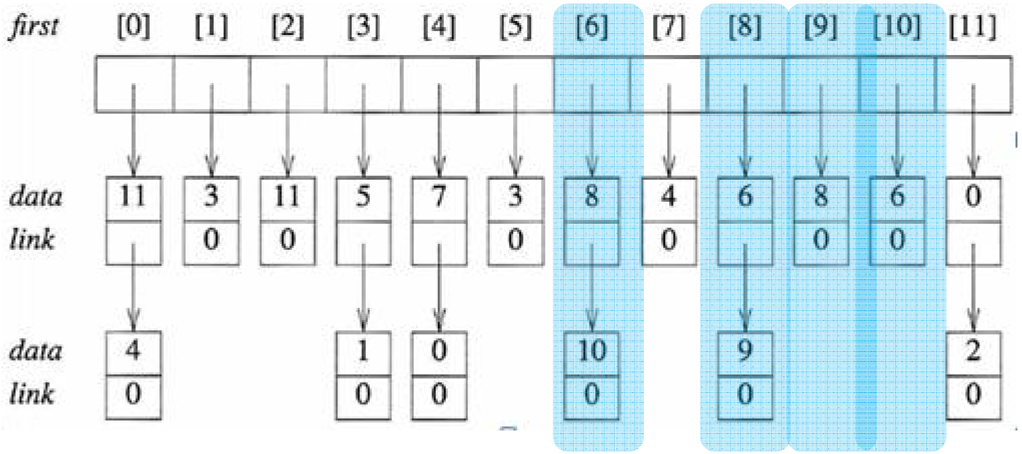
x	y	top

```
/* 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 */
    }
  }
}
```



x	y	top

```
/* 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 */
      }
   }
}
```



x	y	top

```
/* 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 */
      }
   }
}
```

Program 4.22 (4)

- Time complexity:

  O(n+m), where n is the size of data set and m is the number of equivalence pairs

  initialization & phase1: O(m+n)

  phase2: O(m+n)

Sparse Matrices

- Linked lists allow us to efficiently represent structures that vary in size → can be applied to sparse matrices, too.
- Represent each column/each row of a sparse matrix as a circularly linked list with a header node.
- Header node and entry(element) node

Sparse Matrices: Representation

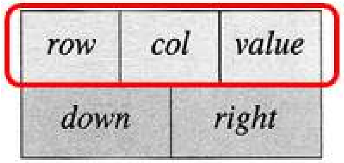
```
#define MAX-SIZE 50
typedef enum {head,entry} tagfield;
typedef struct matrixNode *matrixPointer;
typedef struct entryNode {
            int row;
            int col;
            int value;
            } ;
typedef struct matrixNode {
            matrixPointer down;
            matrixPointer right;
            tagfield tag;
            union {
               matrixPointer next;
               entryNode entry;
            } u;
         };
matrixPointer hdnode[MAX-SIZE];
```

(a) header node

(b) element node

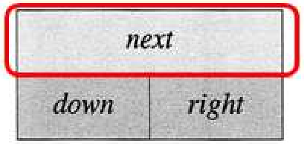
head field is not shown

- 세 종류의 header가 있음.
- list of head
  - [row, col, value]



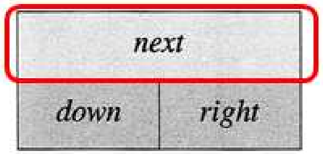
(b) element node

- COL_head

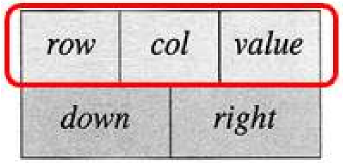


(a) header node

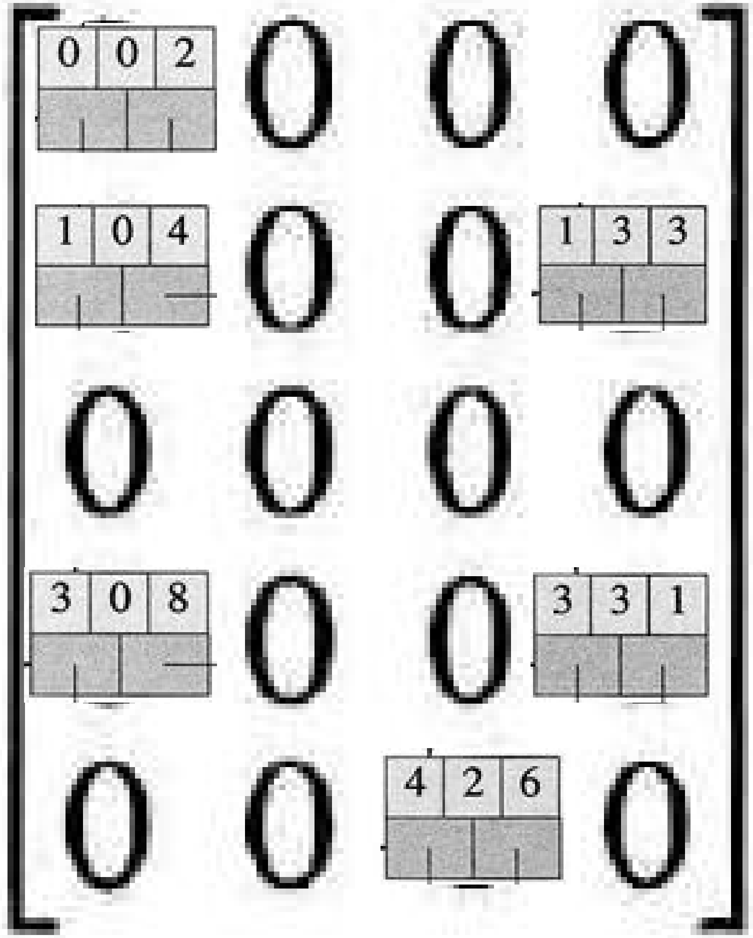
- ROW_head



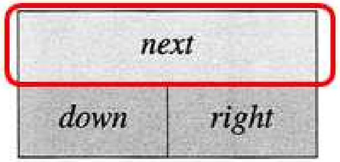
(a) header node



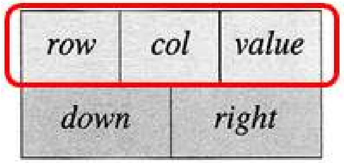
(b) element node



Sparse Matrices: Representation



(a) header node



(b) element node

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

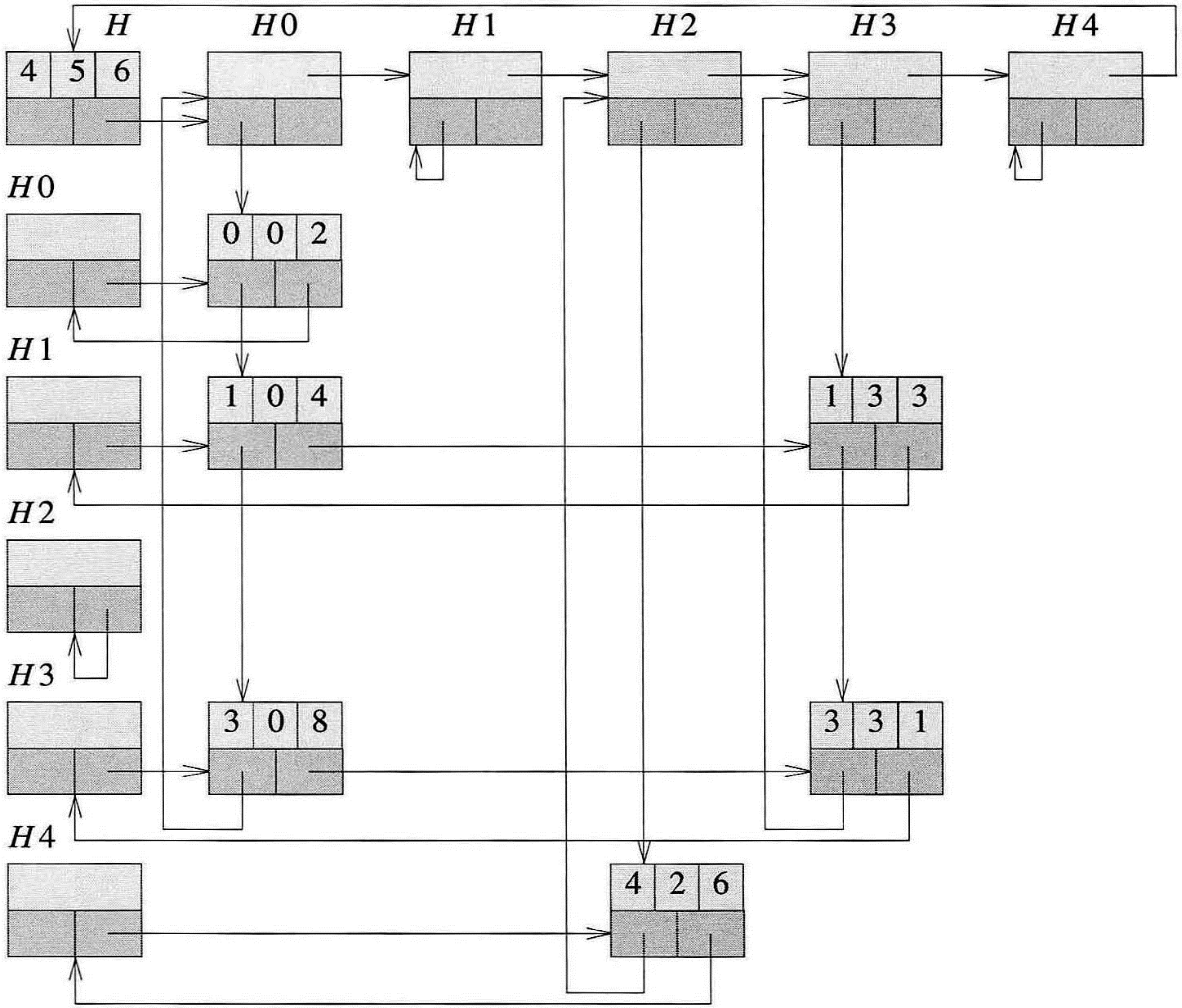


Figure 4.19: Linked representation of the sparse matrix of Figure 4.18 (the *head* field of a node is not shown)