Chap 4. Linked Lists (4)

Contents

Contents

Relation

- Relation on Set A={a, b, c, d, e…}
  - Notation : ${}_aR_b$ (a와 b는 관계 R)
  - Relation R = { (a, b), (c, d) ….} (set으로 나타냄)
  - Relation의 종류
    - 자연수 집합 : =, <, <=, >, >=
    - 사람의 집합 : 배우자, 가족, 동창, 동향, 원수

- Ex)
  - S={“철수”, “영희”, “경자”, “순희”, “용필” …}
    - R={(“철수”, “영희”), (“용필”, “순희”,)…)} : 배우자

4.6 Equivalence Class

Definition : A relation, $\equiv$, over a set, S, is said to be an ***equivalence relation*** over S *iff* it is *reflexive*, *symmetric* and *transitive* over S.

- Reflexive : $x \equiv x$
- Symmetric : $x \equiv y \Rightarrow y \equiv x$
- Transitive : $x \equiv y$ and $y \equiv z \Rightarrow x \equiv z$

※

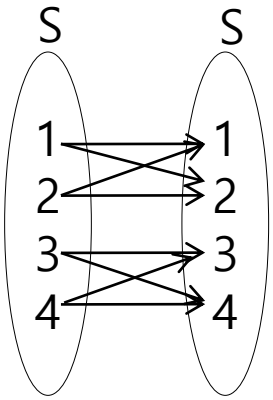
xRx

$xRy \Rightarrow yRx$

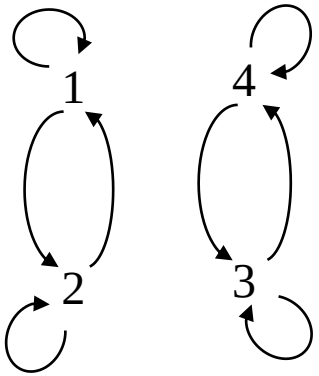
xRy and $yRz \Rightarrow xRz$

- **Example:** A relation R of a set S={1, 2, 3, 4} is an *equivalence relation*, where R = {(1, 1), (1, 2), (2, 1), (2, 2), (3, 3), (3, 4), (4, 3), (4, 4)}.

arrow diagram



directed graph



relation matrix

	1	2	3	4
1	1	1	0	0
2	1	1	0	0
3	0	0	1	1
4	0	0	1	1

Definition : If R is an equivalence relation over a set S, ***equivalence class*** of a is defined as $[a] = \{x \mid (a, x) \in R\}$.

Ex)
S={"Kim", "Park", "Jon", "Marry", "Hong", "Son"}

R(alumnus)={("kim", "Kim"), ……("Son", "Son"), ("Kim", "Park"), ("Park", "Kim"), ("Jon", "Marry"), ("Marry", "Jon"), ("Hong", "Son"), ("Son", "Hong"), ("Kim", "Hong"), ("Hong", "Kim"), ("Kim", "Son"), ("Son", "Kim"), ("Park", "Hong"), ("Hong", "Park"), ("Park", "Son"), ("Son", "Park")}

Equivalance Class ["Kim"] ={"Kim", "Park", "Hong", "Son" }

- **Example:** Consider an equivalence relation R = {(1, 1), (1, 2), (2, 1), (2, 2), (3, 3), (3, 4), (4, 3), (4, 4)} over a set S={1, 2, 3, 4}.

  – The equivalence classes :

    [1] = [2] = { 1, 2 }, [3] = [4] = { 3, 4 }

  – The partition of S :{{1, 2}, {3, 4}}

We can use an equivalence relation to partition a set S into equivalence classes such that *two members x and y of S are in the same equivalence class iff x ≡ y*.

- **Example:**
  - For twelve polygons numbered 0 through 11

    S = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}

  - If the following pairs overlap,

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

  - then, the equivalence classes :
    - { 0, 2, 4, 7, 11 }, { 1, 3, 5}, { 6, 8, 9, 10}

Algorithm to determine equivalence classes

1. Read in and store the equivalence pairs <*i*, *j*>
2. Begin at 0 and find all pairs of the form <0, *j*>
3. *By transitivity*, all pairs of the form <*j*, *k*> imply that *k* is in the same equivalence class as 0.
4. Continue in this way, found, marked, and printed *the entire equivalence class containing 0*.
5. Then, continue on.

```
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 the equivalence class;
     }
}
```

Program 4.21: A more detailed version of the equivalence algorithm

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

initialize

seq	out
NULL	T
NULL	T
NULL	T
NULL	T
NULL	T
NULL	T
NULL	T
NULL	T
NULL	T
NULL	T
NULL	T
NULL	T

Equivalence relation

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

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

Equivalence relation

seq	out	0	1	2	3	4	5	6	7	8	9	10	11
0	F					1							1
1	T				1								
2	T												1
3	T		1				1						
4	F	1							1				
5	T				1								
6	T									1		1	
7	T					1							
8	T							1			1		
9	T									1			
10	T							1					
11	F	1		1									

0 class : stack

11
4

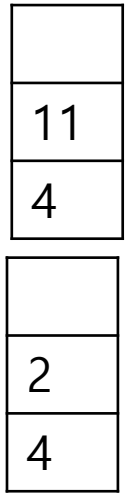
0, 4, 11

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

Equivalence relation

seq	out	0	1	2	3	4	5	6	7	8	9	10	11
0	F					1							1
1	T				1								
2	F												1
3	T		1				1						
4	F	1							1				
5	T				1								
6	T									1		1	
7	T					1							
8	T							1			1		
9	T									1			
10	T							1					
11	F	1		1									

0 class : stack



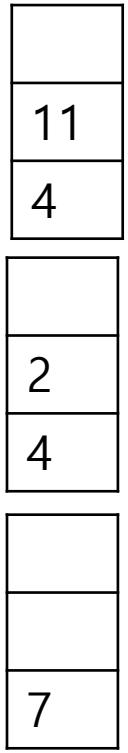
0, 4, 11 , 2

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

Equivalence relation

seq	out	0	1	2	3	4	5	6	7	8	9	10	11
0	F					1							1
1	T				1								
2	F												1
3	T		1				1						
4	F	1							1				
5	T				1								
6	T									1		1	
7	F					1							
8	T							1			1		
9	T									1			
10	T							1					
11	F	1		1									

0 class : stack



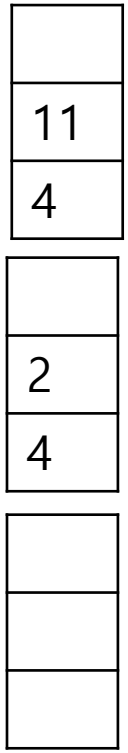
0, 4, 11 , 2, 7

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

Equivalence relation

seq	out	0	1	2	3	4	5	6	7	8	9	10	11
0	F					1							1
1	T				1								
2	F												1
3	T		1				1						
4	F	1							1				
5	T				1								
6	T									1		1	
7	F					1							
8	T							1			1		
9	T									1			
10	T							1					
11	F	1		1									

0 class : stack



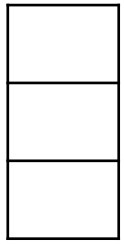
0, 4, 11 , 2, 7

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

Equivalence relation

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

stack



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

Representations for this algorithm

- An 2D *Boolean* array *pairs*[*n*][*n*]
  - *n* : the number of objects
  - *pairs*[*i*][*j*] = *TRUE iff* the pair <*i*, *j*> is in the input.
  - potentially wasteful of space
  - $\Theta(n^2)$ time, just to initialize the array

- A linked representation
  - *seq*[*n*] holds the header nodes of the *n* lists
  - *out*[*n*] tells us whether or not the object *i* has been printed

- Phase one

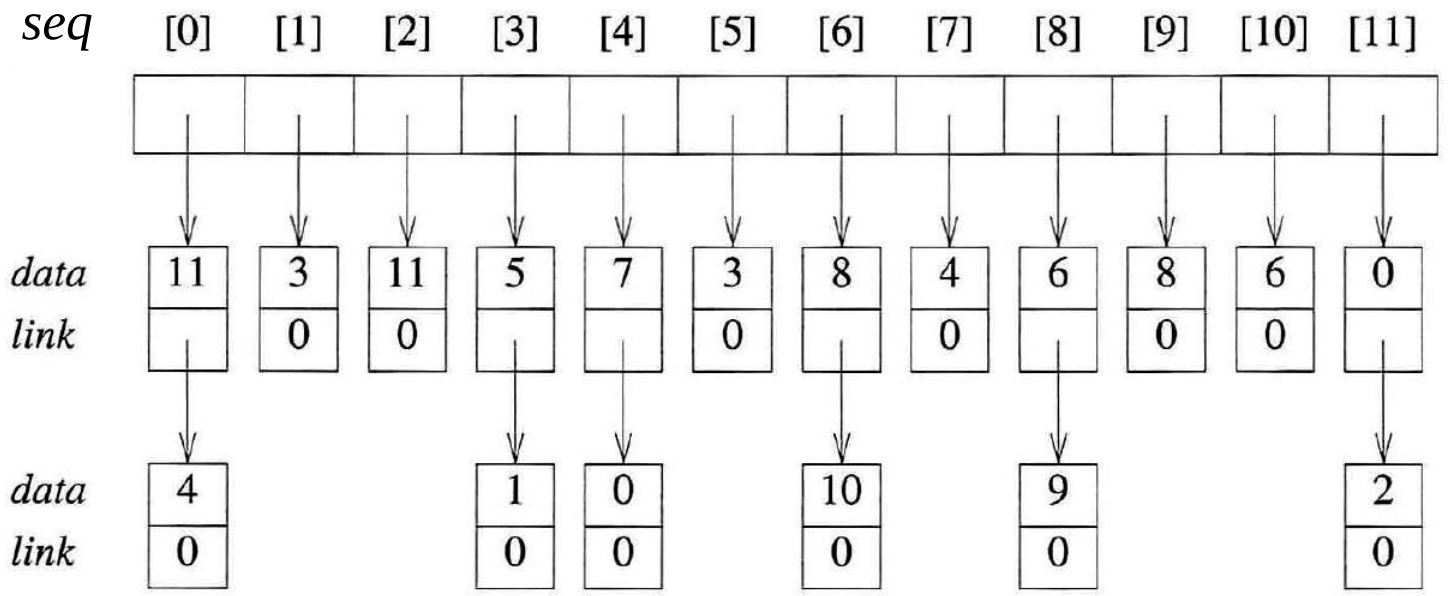


Figure 4.16: Lists after pairs have been input

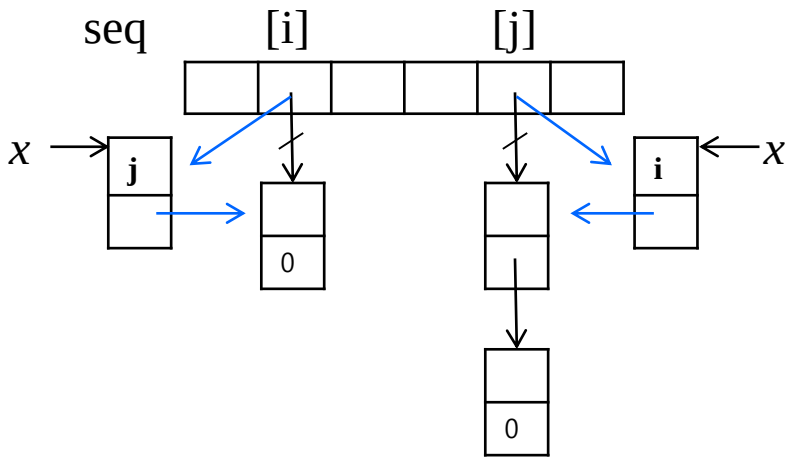
Input pairs : (0, 4), (3, 1), (6, 10), (8, 9), (7, 4), (6, 8), (3, 5), (2, 11), (11, 0)

- Phase two
  - we scan the `seq` array for the first i, $0 \leq i \leq n$, such that `out`[i] = TRUE. Each element in the list `seq`[i] is printed.
  - To process the remaining lists which, by transitivity, belong in the same class as i, we create a stack of their nodes.
  - We do this by changing the link fields so that they point in the reverse direction.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 24
#define FALSE 0
#define TRUE 1
typedef struct node *nodePointer;
typedef struct node {
        int data;
        nodePointer link;
        } node;
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);
}
```



- Phase one

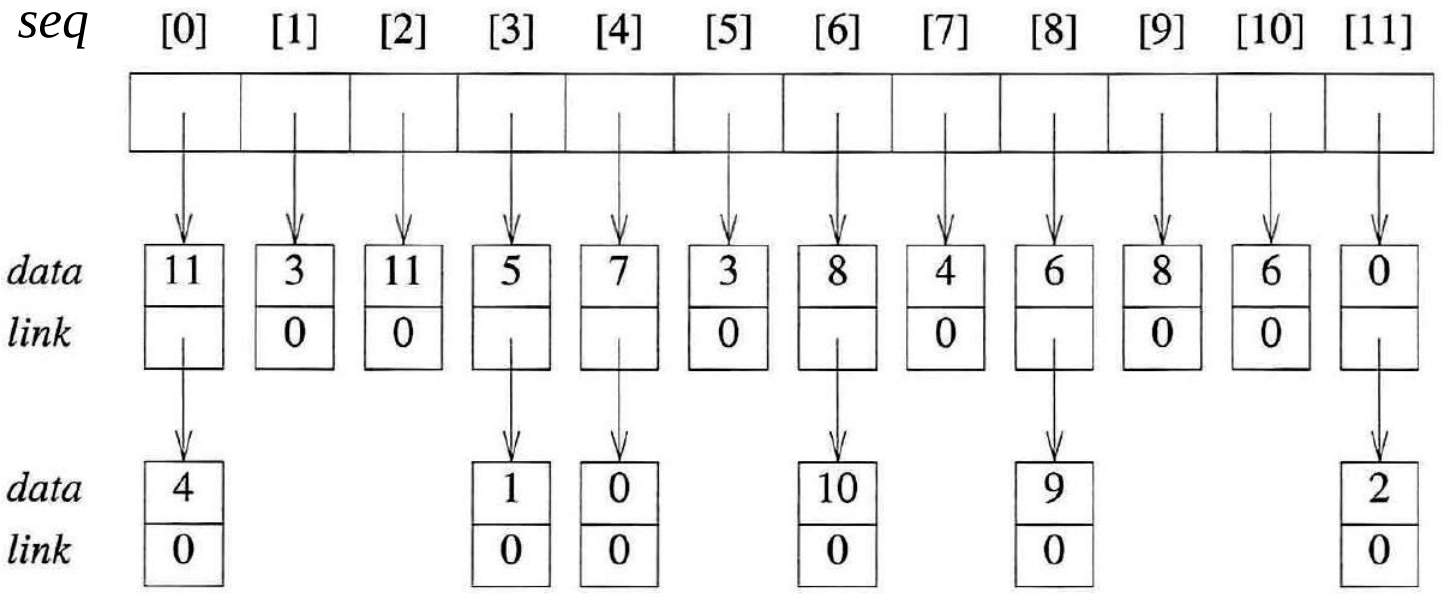
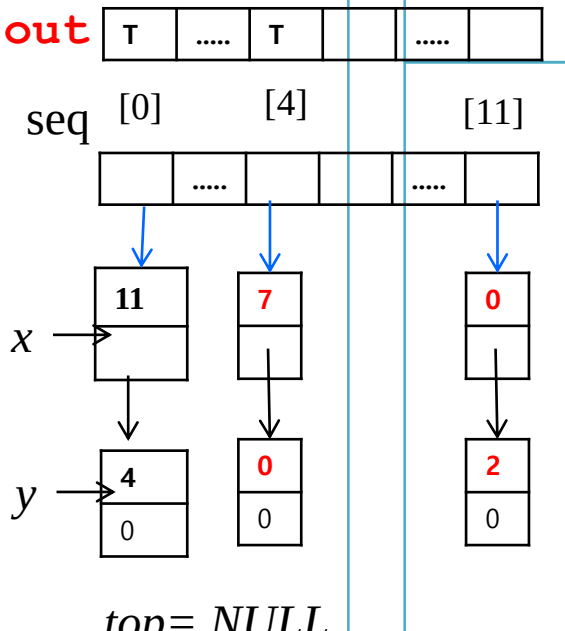


Figure 4.16: Lists after pairs have been input

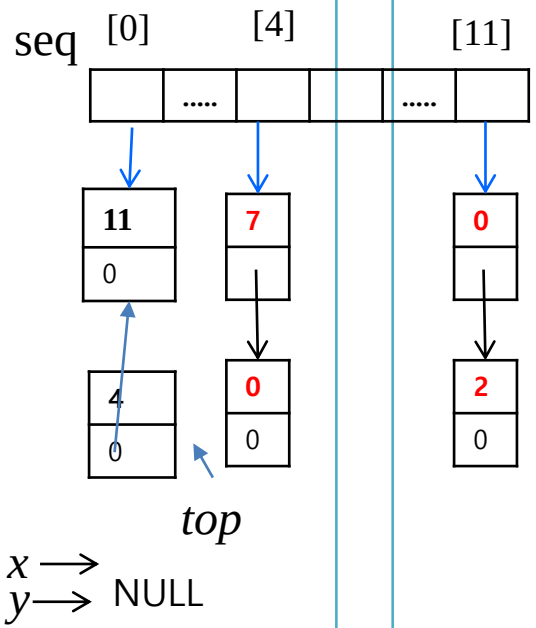
Input pairs : (0, 4), (3, 1), (6, 10), (8, 9), (7, 4), (6, 8), (3, 5), (2, 11), (11, 0)




```
/* 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 true  */
      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;
                                                  // push
            }
            else x = x→link;
         }
         if (!top) break; // empty stack
         x = seq[top→data]; top = top→link;  // pop
                                          /* unstack */
      }
   }
}
```

Program 4.22: Program to find equivalence classes

```
/* 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 true  */
      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;
                                                  // push
            }
            else x = x→link;
         }
         if (!top) break; // empty stack
         x = seq[top→data]; top = top→link;  // pop
                                          /* unstack */
      }
   }
}
```

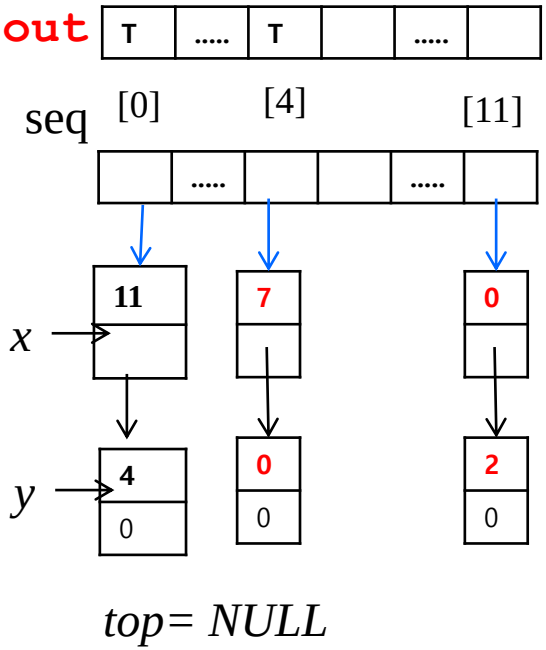


Program 4.22: Program to find equivalence classes

```
/* Phase 2: Output the equivalence classes */
        for ( i = 0; i < n; i++)
        {
                if( out[i] == TRUE )
                {
                        printf("₩nNew class: %5d", i);
                        out[i] = FALSE;                 /* set class to true */
                        x = seq[i]; top = -1;  // top = NULL;   /* initialize stack */
                        for (;;)                        /* find rest of class */
                        {
                            while(x)  /* process list */
                            {
                                    j = x->data;
                                    if(out[j])
                                        {
                                         printf("%5d", j); out[j] = FALE;
                                         push(j);
                                         x = x->link;
                                         //y = x->link; x->link = top; top = x; x = y;
                                          }
                                    else
                                        {
                                         x = x->link;
                                         }
                             } // while

                        if ( top == -1) break;
                        x = seq[pop()];             /* unstack */
                         // x = seq[top->data]; top = top->link;
                        }// for

                }// if
        }// for
```



- Analysis of the equivalence program
  - *m*: the number of input pairs
  - *n*: the number of objects (the size of a set S)
  - Time complexity : O(*m*+*n*)
    - initialize : n(seq, out)+m(node)
    - second path : n + 2m(stack)
    - 2n+3m => O(*m*+*n*)
  - Space complexity : O(*m*+*n*)
    - seq, out : 2n
    - node link to seq : 2m
    - stack : reuse node

4.7 Sparse Matrices

4.7.1 Sparse Matrix Representation

- *Linked lists* allow us to efficiently represent *structures that vary in size*
  - can be applied to sparse matrices, too.

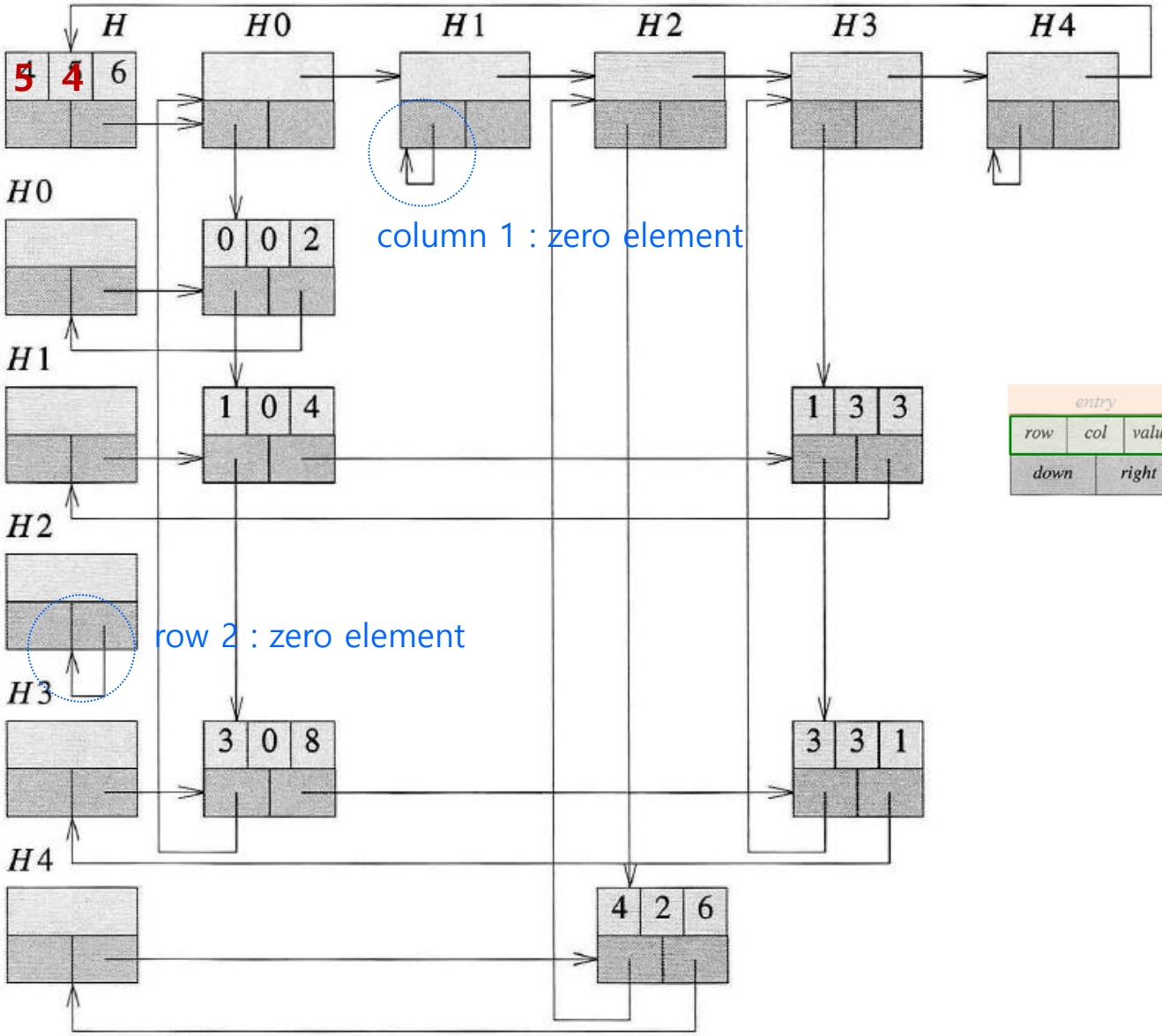
- Represent *each column or each row* of a sparse matrix as *a circularly linked list with a header node*.
  - header node and entry(element) node

```
#define MAX_SIZE 50 /*size of largest matrix*/
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

tag field is not shown

Figure 4.17: Node structure for sparse matrices



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

Figure 4.18: 5 × 4 sparse matrix *a*

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

4.7.2~4 Sparse Matrix Operations

- Sparse Matrix Input
- Sparse Matrix Output
- Erasing a Sparse Matrix

4.8 Doubly Linked Lists

- Limitation in *chains* and *singly linked circular lists*
  - The only way to find a specific node *p* or the node that precedes the node *p* is to start at the beginning of the list.
  - Easy deletion of an arbitrary node requires knowing the preceding node.

- It is useful to have *doubly linked lists,* for a problem that
  - need to move in either directions
  - must delete an arbitrary node

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

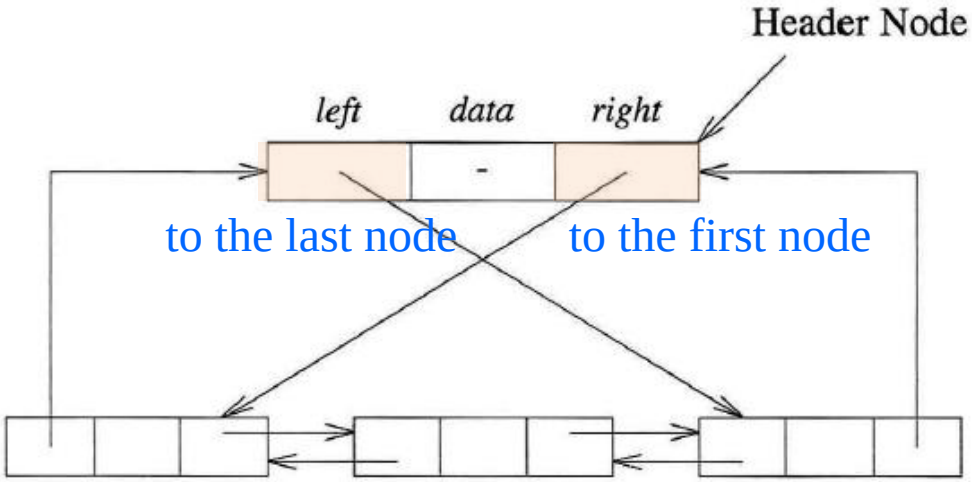


Figure 4.21: Doubly linked circular list with header node

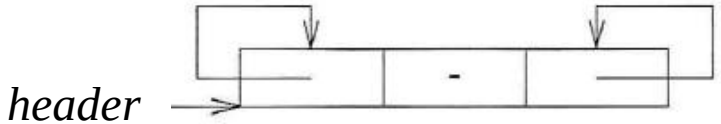
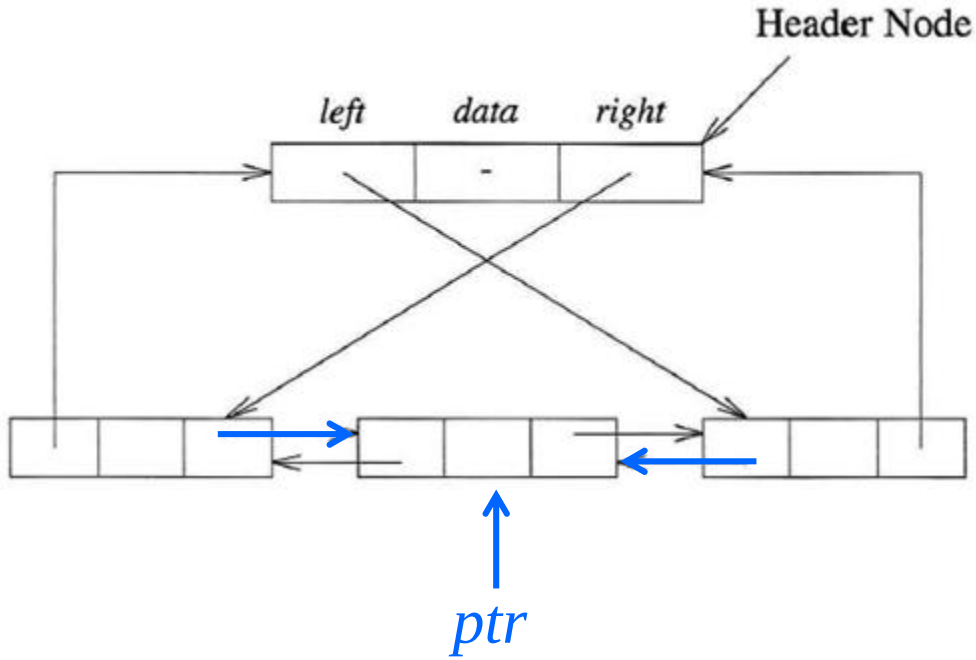


Figure 4.22: Empty doubly linked circular list with header node



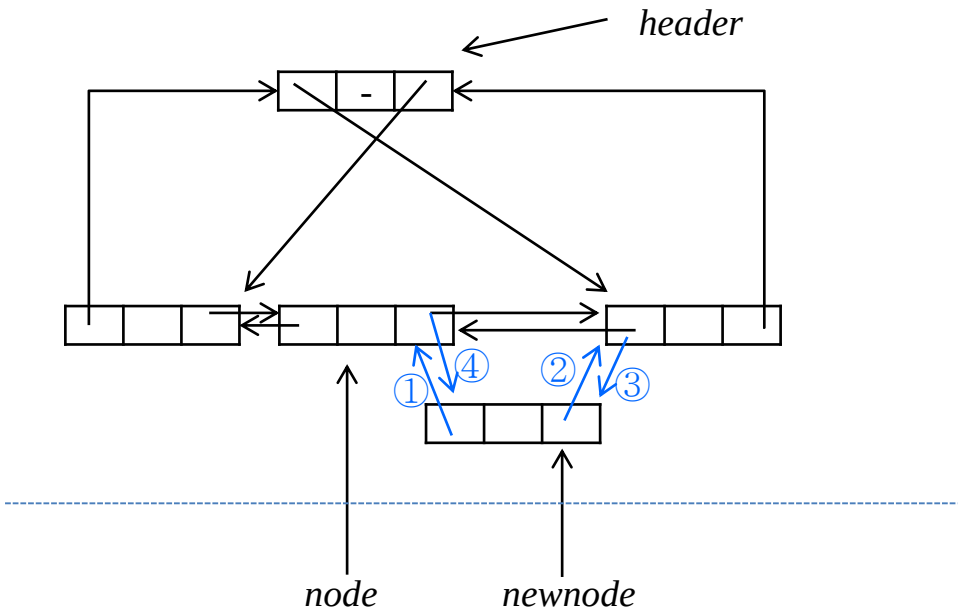
If *ptr* points to any node in a doubly linked list, then

$$ptr = ptr \rightarrow llink \rightarrow rlink = ptr \rightarrow rlink \rightarrow llink$$

This formula reflects that
we can go back and forth with equal ease.

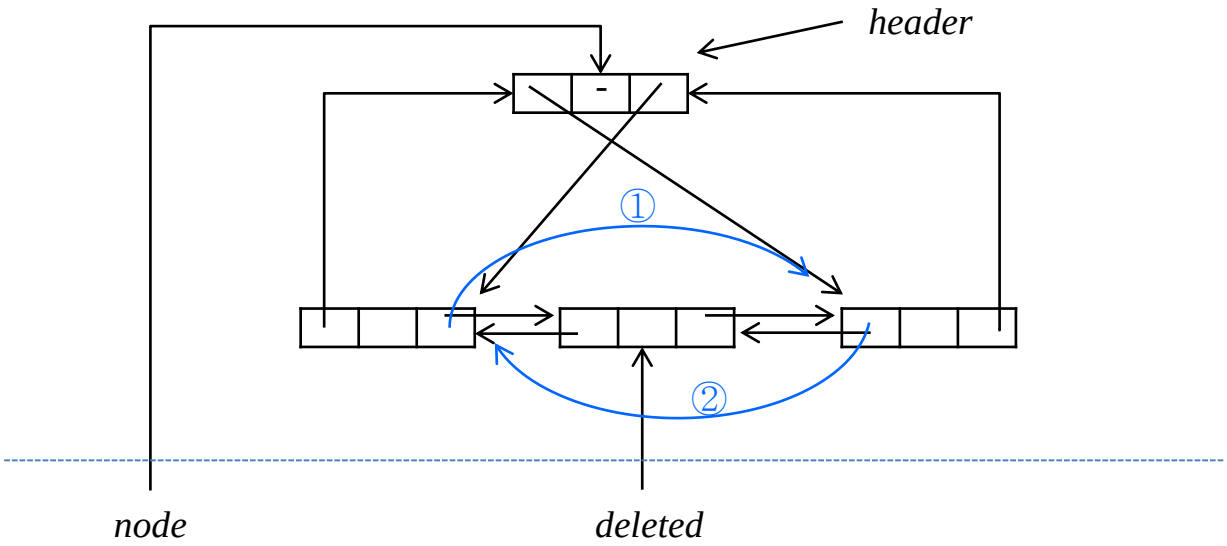
```
void dinsert(nodePointer node, nodePointer newnode)
{/* insert newnode to the right of node */
   newnode→llink = node;                    ①
   newnode→rlink = node→rlink;              ②
   node→rlink→llink = newnode;              ③
   node→rlink = newnode;                    ④
}
```

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;   ①
      deleted→rlink→llink = deleted→llink;   ②
      free(deleted);                         ③
   }
}
```

Program 4.27: Deletion from a doubly linked circular list



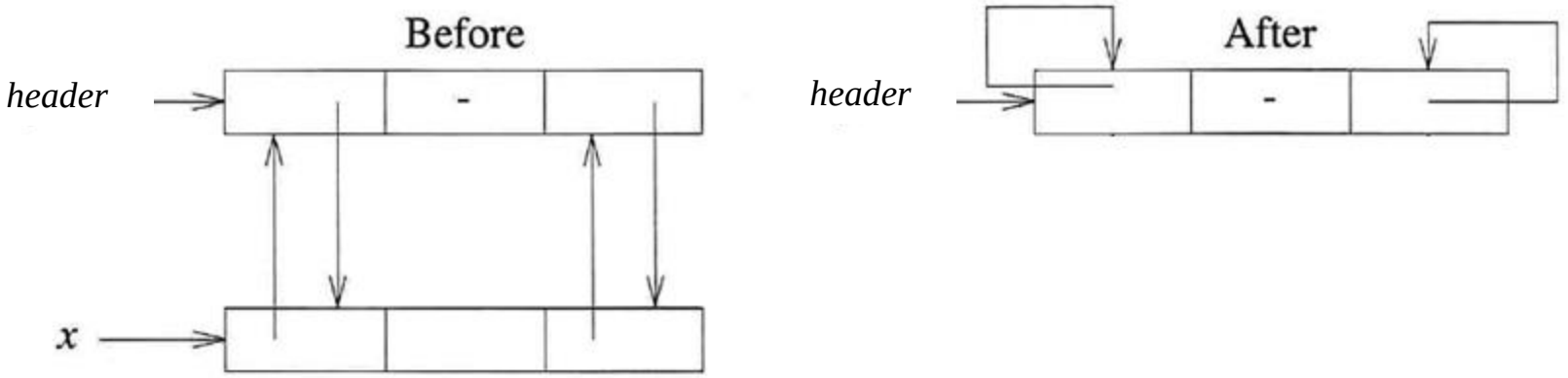


Figure 4.23: Deletion from a doubly linked circular list with a single node