제4장 Linked lists

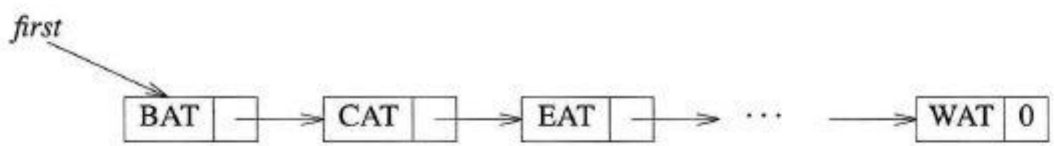
경북대학교 임경식 교수

4.1 Singly linked lists and chains

❖ An elegant solution to this problem of data movement in *sequential* representations is achieved by using *linked* representations

(BAT, CAT, EAT, FAT, HAT, JAT, LAT, MAT, OAT, PAT, RAT, SAT, VAT, WAT)

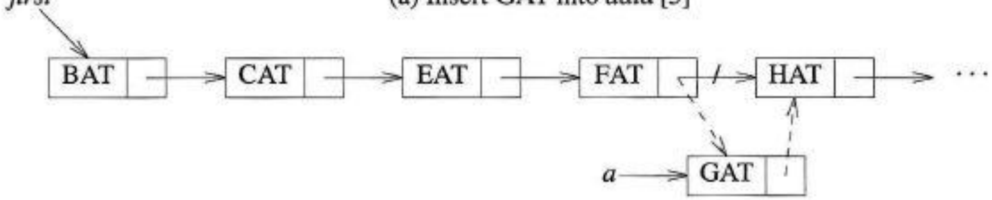
	data	*link*
1	HAT	15
2		
3	CAT	4
4	EAT	9
5		
6		
7	WAT	0
8	BAT	3
9	FAT	1
10		
11	VAT	7
	.	.
	.	.



4.1 Singly linked lists and chains

	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

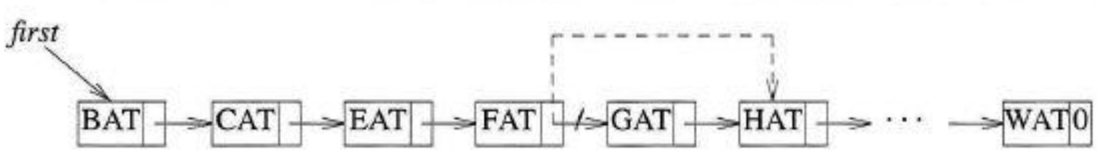
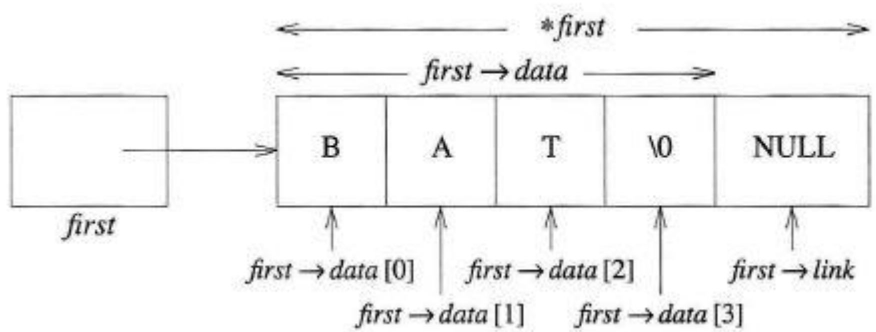


Figure 4.4: Delete GAT

4.2 Representing chains in C

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




```
listPointer first = NULL;
```

```
#define IS_EMPTY(first) (!(first))
```

```
MALLOC(first, sizeof(*first));
```

```
strcpy(first→data,"BAT");
first→link = NULL;
```

4.2 Representing chains in C

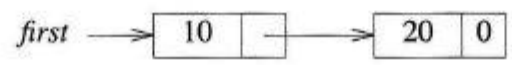


Figure 4.6: A two-node list

```
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;
}
```

4.2 Representing chains in C

```
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

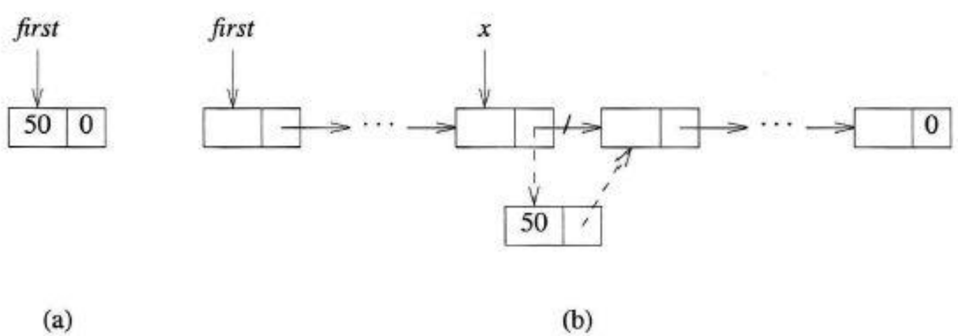


Figure 4.7: Inserting into an empty and nonempty list

4.2 Representing chains in C

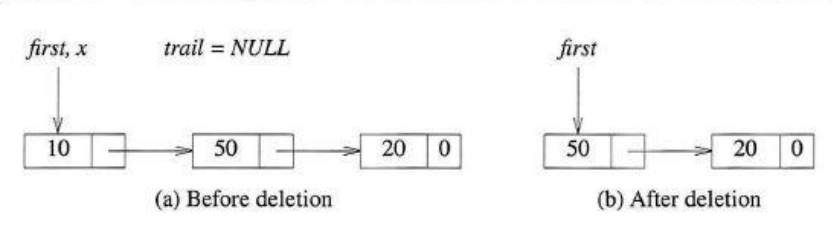


Figure 4.8: List before and after the function call *delete(&first, NULL, first);*

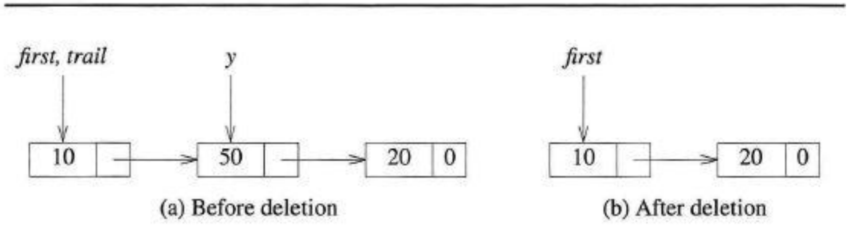


Figure 4.9: List after the function call *delete(&first, y, y→link);*

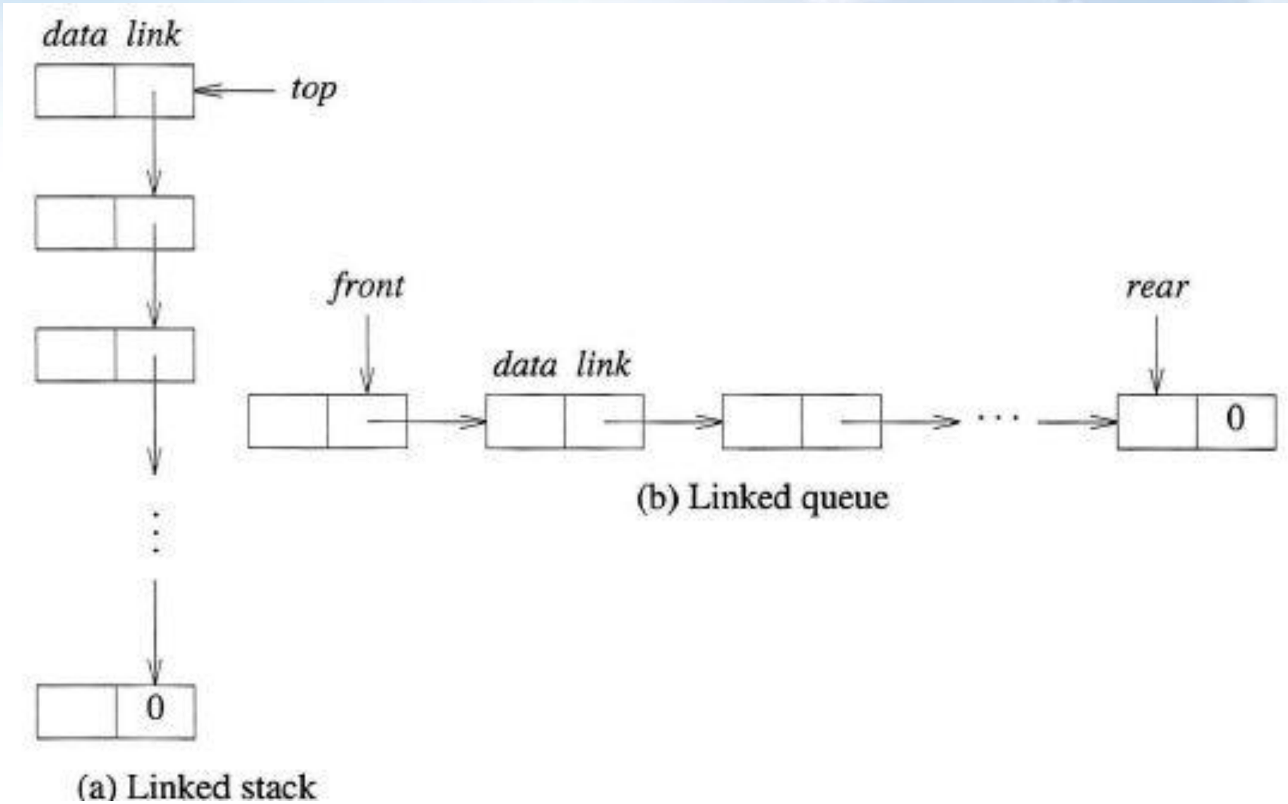
```
void delete(listPointer *first, listPointer trail,
                                listPointer x)
{/* delete x from the list, trail is the preceding node
    and *first is the front of the list */
   if (trail)
      trail→link = x→link;
   else
      *first = (*first)→link;
   free(x);
}
```

Program 4.3: Deletion from a list

```
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

4.3 Linked stacks and queues



4.3 Linked stacks and queues

```
#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];
```

$top[i] = NULL, 0 \le i < MAX\_STACKS$

$top[i] = NULL$ iff the ith stack is empty

```
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;
}
```

```
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;
}
```

4.3 Linked stacks and queues

```
#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];
```

$front[i] = NULL, 0 \leq i < MAX\_QUEUES$

$front[i]$ = NULL iff the ith queue is empty

```
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;
}
```

```
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;
}
```

4.4 Polynomials

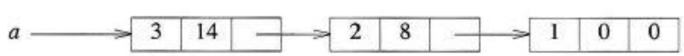
$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0 x^{e_0}$$

coef	expon	link

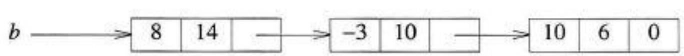
```
typedef struct polyNode *polyPointer;
typedef struct polyNode {
        int coef;
        int expon;
        polyPointer link;
        };
polyPointer a,b;
```

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

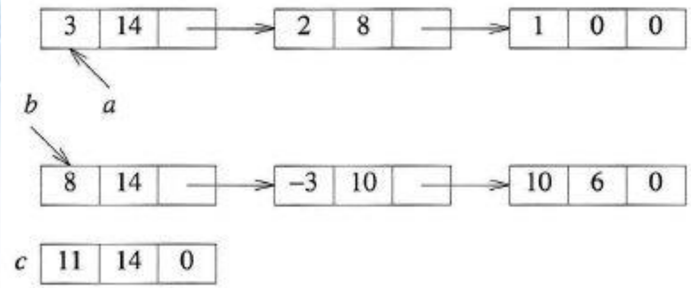
$$b = 8x^{14} - 3x^{10} + 10x^6$$



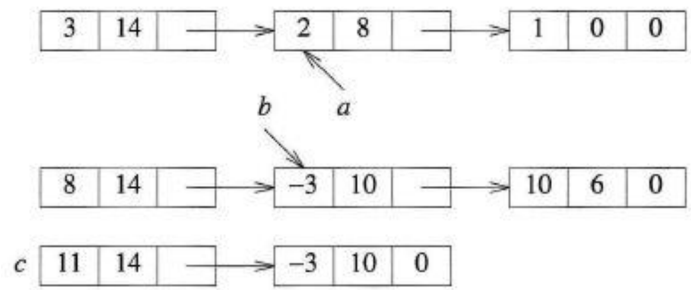
(a)



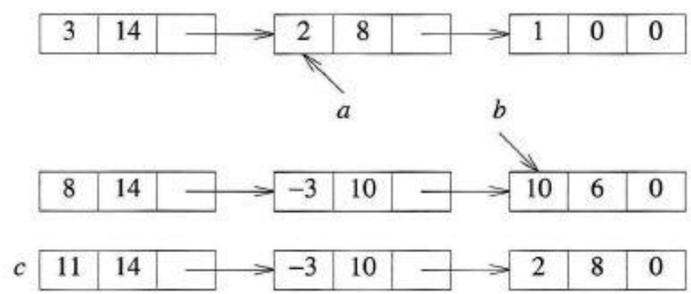
(b)



(i) $a \rightarrow expon == b \rightarrow expon$



(ii) $a \rightarrow expon < b \rightarrow expon$



(iii) $a \rightarrow expon > b \rightarrow expon$

4.4 Polynomials

// Polynomial c = polynomial a + polynomial 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;
```

```
void attach(float coefficient, int exponent,
            polyPointer *ptr)
{/* create a new node with coef = coefficient and expo
    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;
}
```

Time complexity   : O(m+n)
Space complexity : O(m+n)

4.4 Polynomials

// Erasing polynomials

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

❖ Let *avail* be a variable of type *poly Pointer* that points to the first node in our list of freed nodes. Initially, we set *avail* to *NULL.* If the *avail* list is not empty, then we may use one of its nodes. Only when the list is empty do we need to use *malloc* to create a new node. Thus, instead of using *malloc* and *free,* we now use *getNode* and *retNode*.

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

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

4.4 Polynomials

// Circular list representation of polynomials

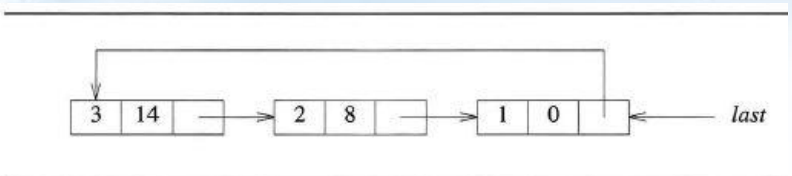
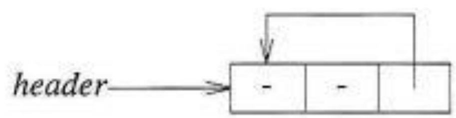


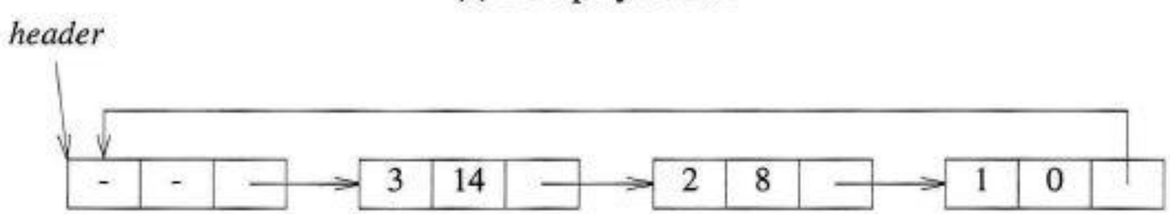
Figure 4.14: Circular representation of $3x^{14} + 2x^8 + 1$

```
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;
   }
}
```

- we must handle the zero polynomial as a special case. To avoid this special case, we introduce a *header node* into each polynomial, that is, each polynomial, zero or nonzero, contains one additional node.



(a) Zero polynomial



(b) $3x^{14} + 2x^8 + 1$

4.4 Polynomials

```
polyPointer startA, c, lastC;
int sum, done = FALSE;
startA = a;           /* record start of a */
a = a→link;           /* skip header node for a and b*/
b = b→link;
c = getNode();        /* get a header node for sum */
c→expon = -1; lastC = c;
do {
  switch (COMPARE(a→expon, b→expon)) {
    case -1: /* a→expon < b→expon */
          attach(b→coef,b→expon,&lastC);
          b = b→link;
          break;
    case 0:  /* a→expon = b→expon */
          if (startA == a)  done = TRUE;
          else {
            sum = a→coef + b→coef;
            if (sum) attach(sum,a→expon,&lastC);
            a = a→link; b = b→link;
          }
          break;
    case 1:  /* a→expon > b→expon */
          attach(a→coef,a→expon,&lastC);
          a = a→link;
  }
} while (!done);
lastC→link = c;
return c;
```

- Singly linked circular lists with a header node
- Poly c = poly a + poly b
- To simplify the addition algorithm for polynomials represented as circular lists, we set the *expon* field of the header node to -1.

4.5 Additional list operations

```
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 concatenate(listPointer ptr1, listPointer ptr2)
{/* produce a new list that contains the list
    ptr1 followed by the list ptr2. The
    list pointed to by ptr1 is changed permanently */
   listPointer temp;
   /* check for empty lists */
   if (!ptr1) return ptr2;
   if (!ptr2) return ptr1;

   /* neither list is empty, find end of first list */
   for (temp = ptr1; temp→link; temp = temp→link) ;

   /* link end of first to start of second */
   temp→link = ptr2;
}
```

4.5 Additional list operations

```
void insertFront(listPointer *last, listPointer node)
{/* insert node at the front of the circular list whose
    last node is last */
   if (!(*last)) {
   /* list is empty, change last to point to new entry */
     *last = node;
     node→link = node;
   }
   else {
   /* list is not empty, add new entry at front */
     node→link = (*last)→link;
     (*last)→link = node;
   }
}
```

```
int length(listPointer last)
{/* find the length of the circular list last */
   listPointer temp;
   int count = 0;
   if (last) {
     temp = last;
     do {
       count++;
       temp = temp→link;
     } while (temp != last);
   }
   return count;
}
```

4.6 Equivalence classes

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

Examples of equivalence relations are numerous. For example, the "equal to" (=) relationship is an equivalence relation since

(1) $x = x$

(2) $x = y$ implies $y = x$

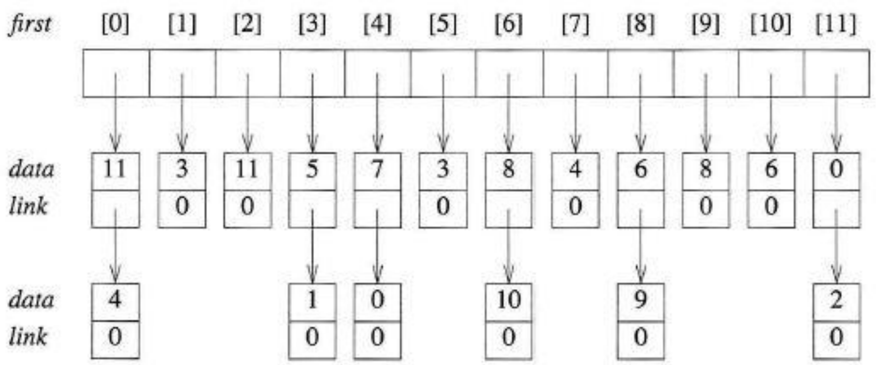
(3) $x = y$ and $y = z$ implies that $x = z$

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

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* $x \equiv y$.

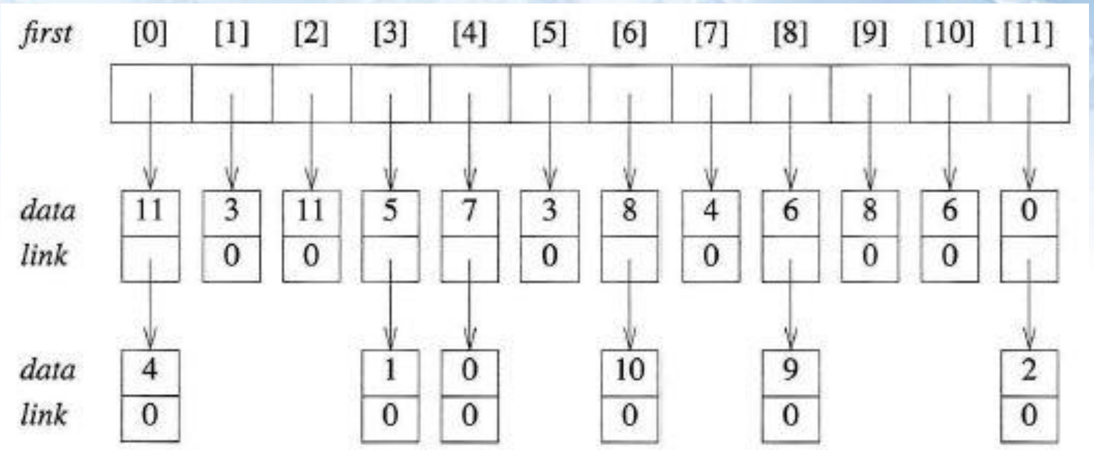
$$\{0, 2, 4, 7, 11\};\ \{1, 3, 5\};\ \{6, 8, 9, 10\}$$

4.6 Equivalence classes



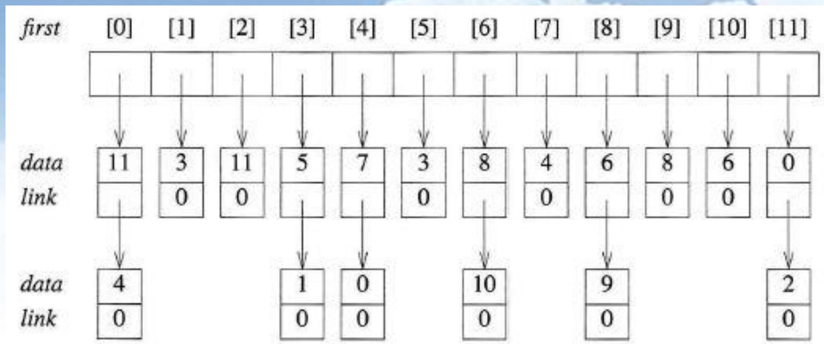

```
#include <stdio.h>
#include <alloc.h>
#define MAX_SIZE 24
#define FALSE 0
#define TRUE 1
typedef struct node *nodePointer;
typedef struct node {
        int data;
        nodePointer link;
        };
void main(void)
{
   short int out[MAX_SIZE];
   nodePointer seq[MAX_SIZE];
   nodePointer x,y,top;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

4.6 Equivalence classes




```
/* 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);
}
```

4.6 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 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 */
      }
   }
}
```

Time complexity : O(m+n)
Space complexity : O(m+n),
Where m and n are the # of related pairs and the # of objects, respectively.

4.8 Doubly linked circular lists

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

```
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

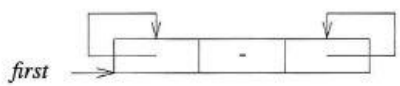


Figure 4.22: Empty doubly linked circular list with header node

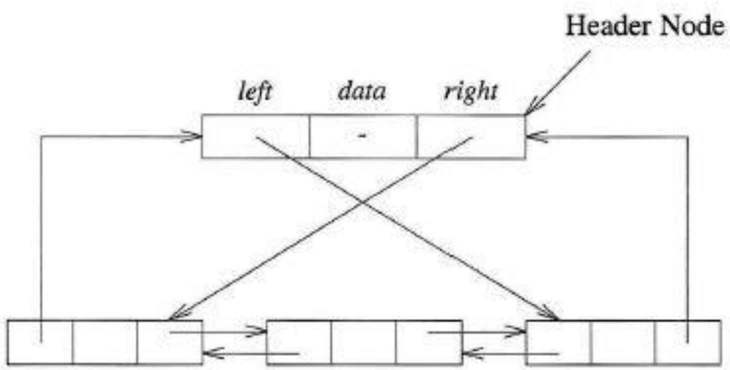


Figure 4.21: Doubly linked circular list with header node

4.8 Doubly linked circular lists

```
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

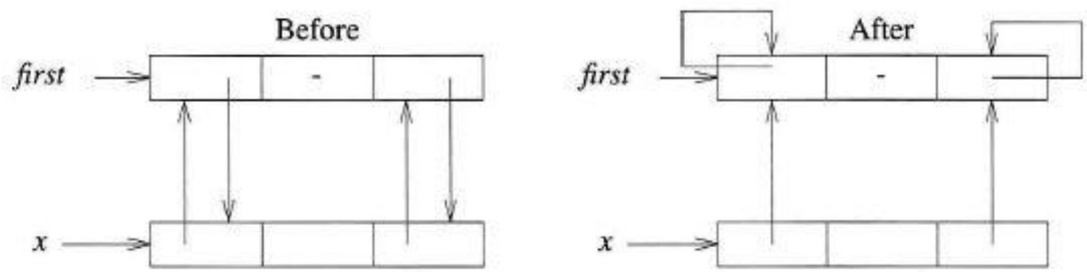


Figure 4.23: Deletion from a doubly linked circular list

Thank You !

▌ 노력 없이 이룰 수 있는 것 아무것도 없다.