Chapter 4 : LISTS

Data Structure Lecture Note
Prof. Jihoon Yang
Data Mining Research Laboratory

4.1 POINTERS

- **Sequential representation**
  - storing successive elements of the data object a fixed distance apart.
  - adequate for many operations.

  But difficulties occurs when
  - insertion and deletion of an arbitrary element (time-consuming)
  - storing several lists of varying sizes in different arrays of maximum size (waste of storage)
  - maintaining the lists in a single array (frequent data movements)

- **Linked representation**
  - A node, associated with an element in the list, contains a *data component* and a *pointer* to the next item in the list. The pointers are often called *links.*

- C provides extensive support for pointers
  - actual value of a pointer type is an address of memory.
  - operators
    - & : the address operator
    - * : the dereferencing (or indirection) operator.

```
int  i, *pi;
pi = &i;
```

- To assign a value to i,

  i = 10; or *pi = 10;

- C allows us to perform arithmetic operations and relational operations on pointers. Also we can convert pointers explicitly to integers.

- The null pointer points to no object or function.
- Typically the null pointer is represented by the integer 0.
- There is a macro called NULL which is defined to be this constant.

- The macro is defined either

  in *stddef.h* for ANSI C or in *stdio.h* for K&R C.

- To test for the null pointer on C

  ```
  if (pi == NULL) or if (!pi)
  ```

4.1.1 Pointers Can Be Dangerous

- By using pointers we can attain a high degree of flexibility and efficiency.
- But pointer can be dangerous: accessing unexpected memory locations
  - Set all pointers to NULL when they are not actually pointing to an object.
  - Explicit *type casts* when converting between pointer types.

```
pi = malloc(sizeof(int));    /* assign to pi a pointer to int */
pf = (float *) pi;     /* casts an int pointer to a float pointer */
```

  - Define explicit return types for functions.

4.1.2 Using Dynamically Allocated Storage

- **malloc**
- **free**

[Program 4.1]

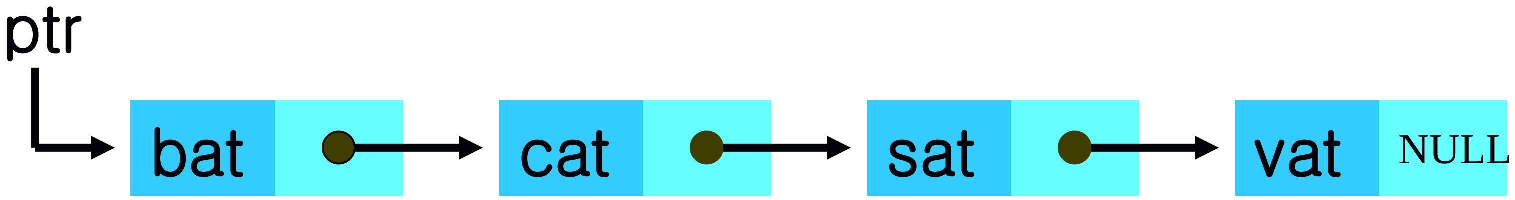
```
int i, *pi;
float f, *pf;
pi = (int *) malloc(sizeof(int));
pf = (float *) malloc(sizeof(float));
*pi = 1024;
*pf = 3.14;
printf("an integer = %d, a float = %f□n", *pi, *pf);
free(pi);
free(pf);
```

inserting pf = (float *) malloc(sizeof(float));

Creates *Garbage, Dangling reference*

4.2 SINGLY LINKED LISTS

[Figure 4.1]



- The name of the pointer to the first node in the list is the name of the list.

- **Note that**

  (1) the nodes do not reside in sequential locations

  (2) the locations of the nodes may change on the different runs.

- When we write a program that works with lists, we almost never look for a specific address except when we test for the end of the list.

- **To insert the word *mat* between *cat* and *sat*, we must :**

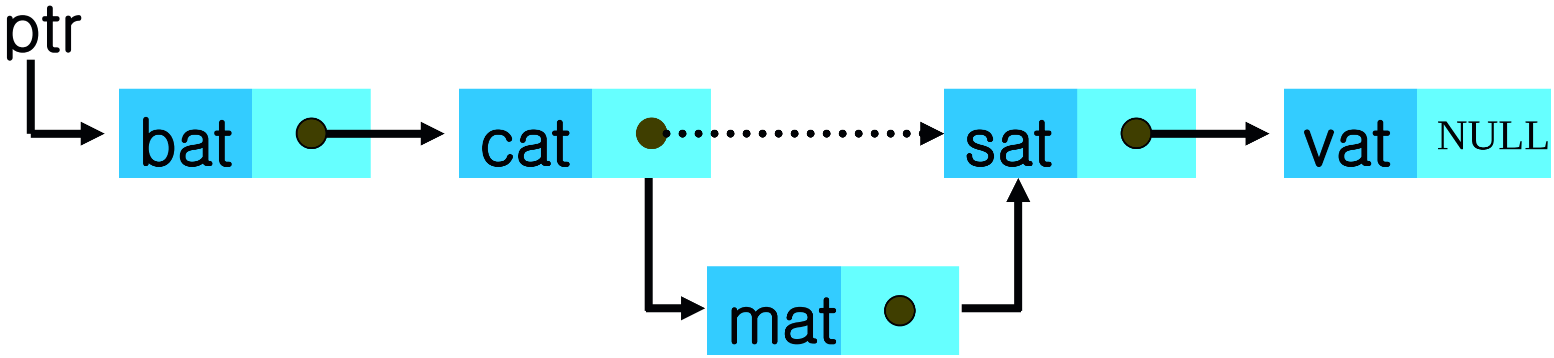
  (1) Get a node that is currently unused; let its address be *paddr*.

  (2) Set the data field of this node to *mat*.

  (3) Set *paddr*'s link field to point to the address found in the link field of the node containing *cat*.

  (4) Set the link field of the node containing *cat* to pointer to *paddr*.

[Figure 4.2]

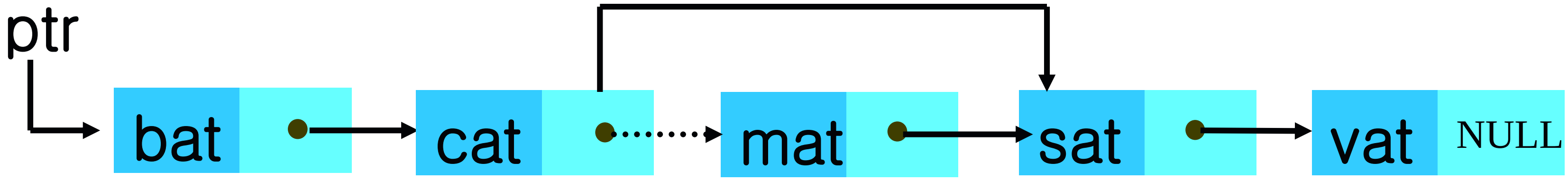


- **To delete *mat* from the list.**

  (1) Find the element (node) that immediately precedes *mat*, which is *cat*.

  (2) Set *cat*'s link field to point to *mat*'s link field.

[Figure 4.3]



- **Necessary capabilities to make linked list possible :**

  (1) A mechanism for defining a node's structure,
  self-referential structures.

  (2) A way to create new nodes when we need them, *malloc*.

  (3) A way to remove nodes that we no longer need, *free.*

- **Example 4.1 [List of words ending in *at*]**

  Necessary declarations are :

```
typedef struct  list_node  *list_pointer;
typedef struct  list_node {
      char  data[4];
      list_pointer  link;
};
list_pointer  ptr = NULL;     /* creating a new empty list */
```

- **A macro to test for an empty list :**

```
#define IS_EMPTY(ptr)  (!(ptr))
```

- **Creating new nodes :**

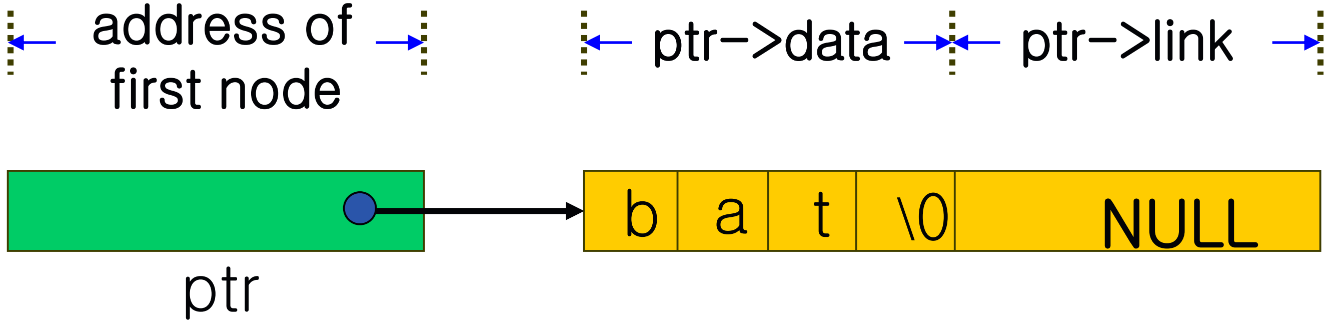
  use the *malloc* function provided in <*stdio.h*>.

```
ptr = (list_pointer) malloc (sizeof(list_node));
```

- **Assigning the values to the fields of the node:**
  - If *e* is a pointer to a structure that contains the field *name*, *e->name* is a shorthand way of writing the expression (**e*).*name*.

- **To place the word bat into the list :**

```
strcpy (ptr->data, "bat");
ptr->link = NULL;
```



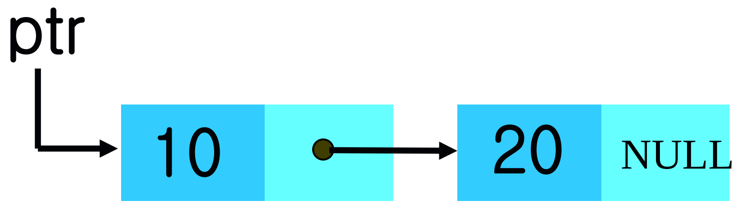
- **Example 4.2 [Two-node linked list] :**

```
typedef struct  list_node  *list_pointer;
typedef struct  list_node {
      int  data;
      list_pointer  link;
};
list_pointer  ptr = NULL;
```

- [Program 4.2]

```
list_pointer create2()
{
/*  create a linked list with two nodes  */
list_pointer first, second;
first = (list_pointer) malloc(sizeof(list_node));
second = (list_pointer) malloc(sizeof(list_node));
second->link = NULL;
second->data = 20;
first->data = 10;
first->link = second;
return first;
}
```

[Figure 4.5]

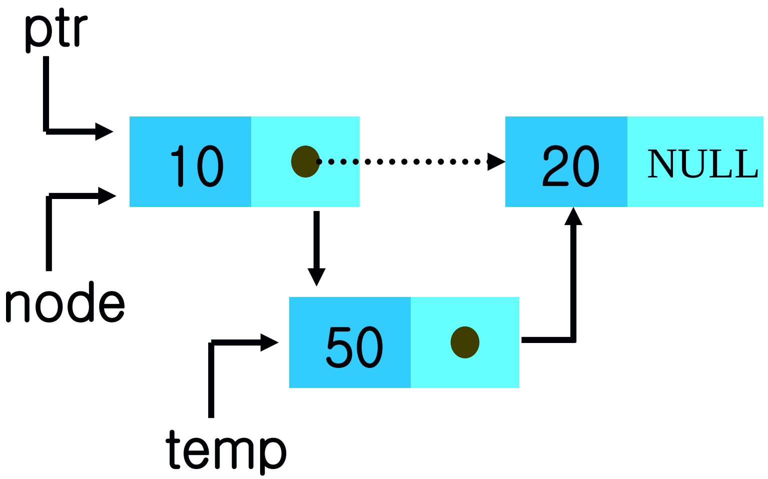


- **Example 4.3 [List insertion] :**
  - To insert a node with data field of 50 after some arbitrary node. Note that we use the parameter declaration *list_pointer *ptr*.
  - We use a new macro, IS_FULL,
    that allows us to determine if we have used all available memory.

```
#define IS_FULL (ptr)  (!(ptr))
```

- **[Program 4.3]**

```
void insert(list_pointer *ptr, list_pointer node)
{
    /* insert a new node with data=50 into the list ptr after node */
    list_pointer temp;
    temp = (list_pointer) malloc(sizeof(list_node));
    if (IS_FULL(temp))  {
        fprintf(stderr, "The memory is full□n");
        exit(1);
     }
    temp->data = 50;
    if (*ptr)  {
        temp->link = node->link;
        node->link = temp;
     }
    else  {
        temp->link = NULL;
        *ptr = temp;
    }
}
```



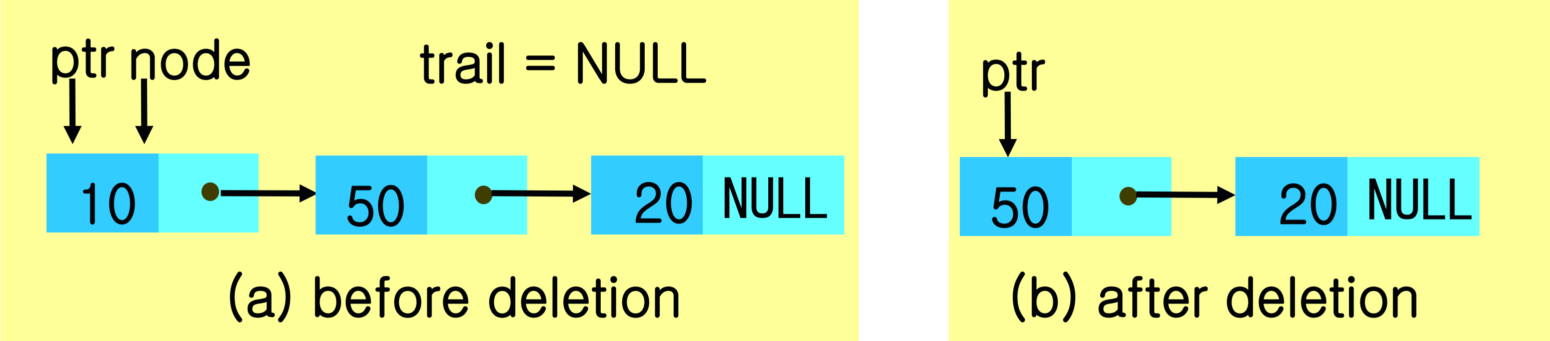
- **Example 4.4 [List deletion] :**
  - Deletion depends on the location of the node to be deleted.
  - Assume three pointers :

    ptr points to the start of the list.

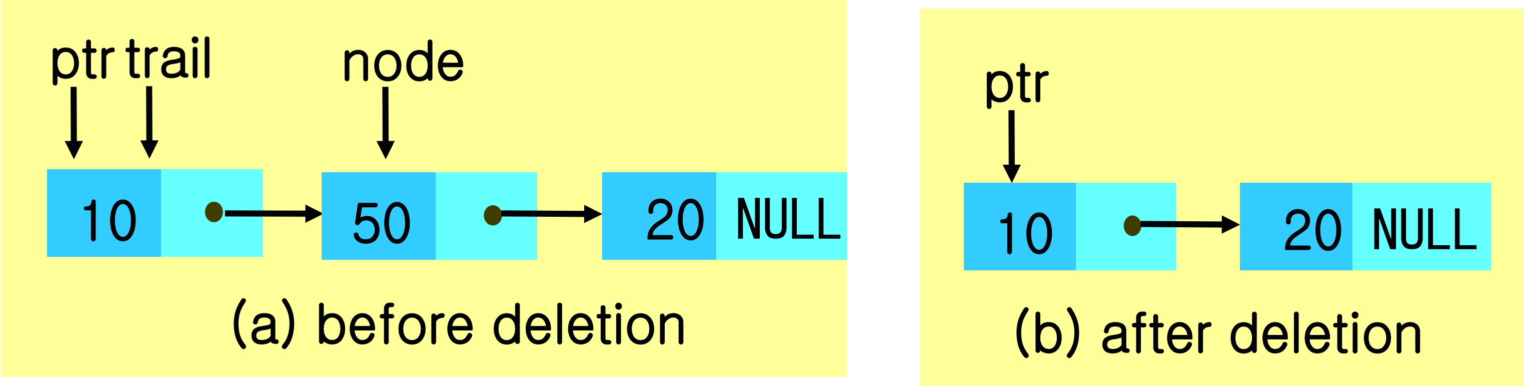
    node points to the node that we wish to delete.

    trail points to the node that precedes the node to be deleted.
- **[Program 4.4]**

```
void delete(list_pointer *ptr, list_pointer trail,  list_pointer node)
{
    /* delete node from the list, trail is the preceding node
    ptr is the head of the list  */
    if (trail)
        trail->link = node->link;
    else
        *ptr = (*ptr)->link;
    free(node);
}
```



[Figure 4.7] ***delete(&ptr, NULL, ptr);***



[Figure 4.8] ***delete(&ptr, ptr, ptr->link);***

- **Example 4.5 [Printing out a list] :**
- **[Program 4.5]**

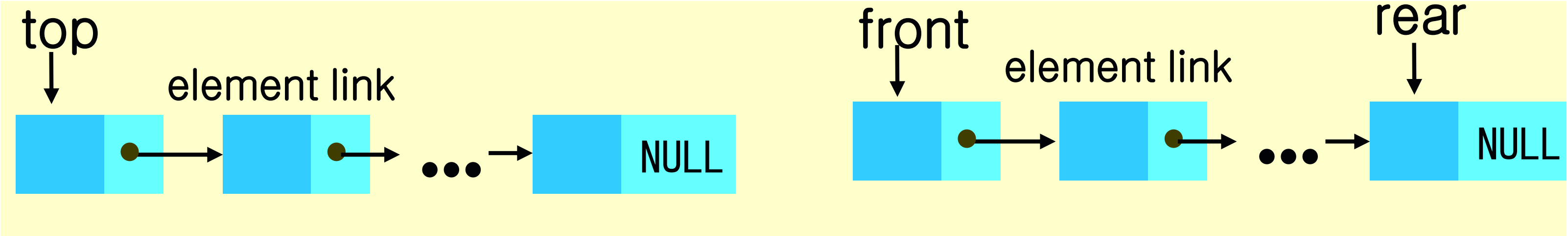
```
void print_list(list_pointer ptr)
{
    printf("The list contains: ");
    for ( ; ptr; ptr = ptr->link)
        printf("%4d", ptr->data);
    printf("□n");
}

list_pointer  search (list_pointer ptr, int num)
{
    for ( ; ptr; ptr = ptr->link)
        if (ptr->data == num)   return  ptr;
return  ptr;
}
```

```
void  merge (list_pointer x,  list_pointer y,  list_pointer *z)
{
      list_pointer  last;
      last = (list_pointer) malloc(sizeof(list_node));
      *z = last;
      while (x && y)  {
            if (x->data <= y->data)  {
                  last->link = x;
                  last = x;
                  x = x->link;
            }
            else  {
                  last->link = y;
                  last = y;
                  y = y->link;
            }
      }
      if (x)   last->link = x;
      if (y)   last->link = y;
      last = *z;  *z = last->link;  free(last);
}
```

4.3 DYNAMICALLY LINKED STACKS AND QUEUES

- Sequential representation is proved efficient if we had only one stack or one queue.
- When several stacks and queues coexisted, there was no efficient way to represent them sequentially.
- Linked stacks and linked queues.



(a) Linked Stack (b) Linked Queue

Notice that the direction of links for both the stack and the queue facilitate easy insertion and deletion of nodes.

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

- **initialize empty stacks :**

  top[*i*] = NULL, 0<=i<MAX_STACKS

- **the boundary conditions :**

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

  and

  IS_FULL(temp) *iff* the memory is full

- **[Program 4.6]**

```
void add(stack_pointer *top, element item)
{
    /*  add an element to the top of the stack  */
    stack_pointer  temp = (stack_pointer) malloc(sizeof(stack));
    if (IS_FULL(temp))  {
        fprintf(stderr, "The memory is full□n");
        exit(1);
    }
    temp->item = item;
    temp->link = *top;
    *top = temp;
}
```

call : *add*(&*top*[*stack_no*], *item*);

- **[Program 4.7]**

```
element delete(stack_pointer *top)
{
      /*  delete an element from the stack */
      stack_pointer  temp = *top;
      element  item;
      if (IS_EMPTY(temp))  {
            fprintf(stderr, "The stack is empty□n");
            exit(1);
      }
      item = temp->item;
      *top = temp->link;
      free(temp);
      return  item;
}
```

call : *item* = *delete*(&*top*[*stack_no*]);

```
#define MAX_QUEUES 10  /* maximum number of queues */
typedef struct  {
      int key;
      /* other fields */
} element;
typedef struct queue *queue_pointer;
typedef struct queue  {
      element  item;
      queue_pointer  link;
};
queue_pointer  front[MAX_QUEUES], rear[MAX_QUEUES];
```

- **initialize empty queues :**

  front[*i*] = NULL, 0<=i<MAX_QUEUES

- **the boundary conditions :**

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

  and

  IS_FULL(temp) *iff* the memory is full

- **[Program 4.8] call : *addq*(&*front[queue_no]*, &*rear[queue_no]*, *item*);**

```
void addq(queue_pointer *front, queue_pointer *rear, element item)
{
    /*  add an element to the rear of the queue  */
    queue_pointer  temp =  (queue_pointer) malloc(sizeof(queue));
    if (IS_FULL(temp))  {
        fprintf(stderr, "The memory is full□n");
        exit(1);
    }
    temp->item = item;
    temp->link = NULL;
    if (*front)  (*rear)->link = temp;
    else  *front = temp;
    *rear = temp;
}
```

- **[Program 4.9]   call : *item* = *deleteq*(&*front[queue_no]*);**

```
element deleteq(queue_pointer *front)
{
    /*  delete an element from the queue */
    queue_pointer  temp = *front;
    element  item;
    if (IS_EMPTY(*front))  {
        fprintf(stderr, "The queue is empty□n");
        exit(1);
    }
    item = temp->item;
    *front = temp->link;
    free(temp);
    return  item;
}
```

4.4 POLYNOMIALS

4.4.1 Representing Polynomials As Singly Linked Lists

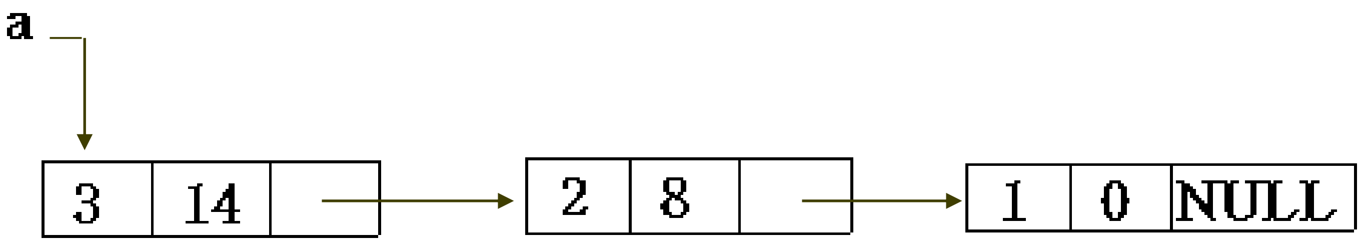
- We want $A(x) = a_{m-1} x^{e_{m-1}} + \cdots + a_0 x^{e_0}$
  - where a_i's are nonzero coefficients and e_i's are nonnegative integer exponents such that e_m*-1* > e_m*-2* > . . . > e_1 > $e_0 \geq 0$.

```
typedef  struct  poly_node  *poly_pointer;
typedef  struct  poly_node  {
      float  coef;
      int  expon;
      poly_pointer  link;
};
poly_pointer  a, b, d;
```

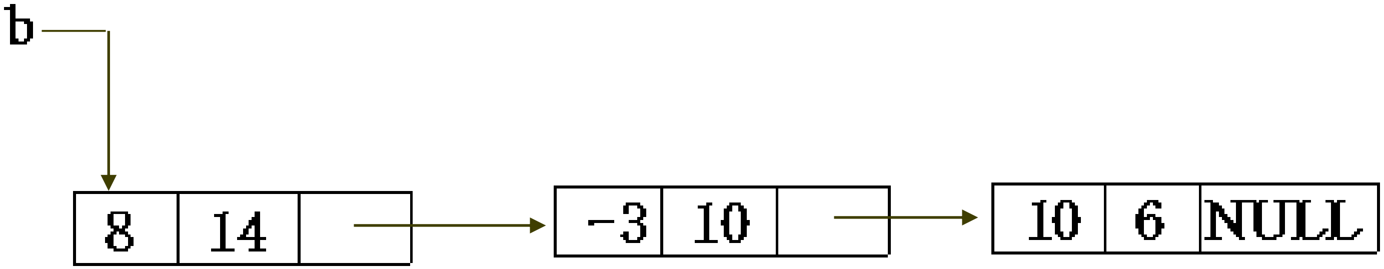
coef	expon	link

- **[Figure 4.11]**

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



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



4.4.2 Adding Polynomials

- Compare Program 4.10 and Program 4.11 with Program 2.5 and Program 2.6.

- **[Program 4.10]**

```
poly_pointer padd(poly_pointer a, poly_pointer b)
{
     /*  return a polynomial which is the sum of a and b  */
     poly_pointer  front, rear, temp;
     float sum;
     rear = (poly_pointer) malloc(sizeof(poly_node));
     if  (IS_FULL(rear))  {
          fprintf(stderr, “The memory is full□n”);
          exit(1);
     }
     front = 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 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 = front;  front = front->link;  free(temp);
    return  front;
}
```

- **[Program 4.11]**

```
void attach(float coefficient, int exponent,   poly_pointer *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  */
    poly_pointer  temp;
    temp = (poly_pointer)malloc(sizeof(poly_node));
    if  (IS_FULL(temp))  {
        fprintf(stderr, "The memory is full□n");
        exit(1);
    }
    temp->coef = coefficient;
    temp->expon = exponent;
    (*ptr)->link = temp;
    *ptr = temp;
}
```

- **Analysis of *padd* :**
  - Similar to the analysis of Program 2.5.

    Three cost measures :

    (1) coefficient additions

    (2) exponent comparisons

    (3) creation of new nodes for d
- Clearly, ≤ 0 number of coefficient additions ≤ min[m, n], number of exponent comparisons and creation of new nodes is at most $m+n$.
- Therefore, its time complexity is O($m + n$).

4.4.3 Erasing Polynomials

- Let's assume that we are writing a collection of functions for input, addition, subtraction, and multiplication of polynomials using linked lists as the means of representation.
- Suppose we wish to compute e(x) = a(x) * b(x) + d(x) :

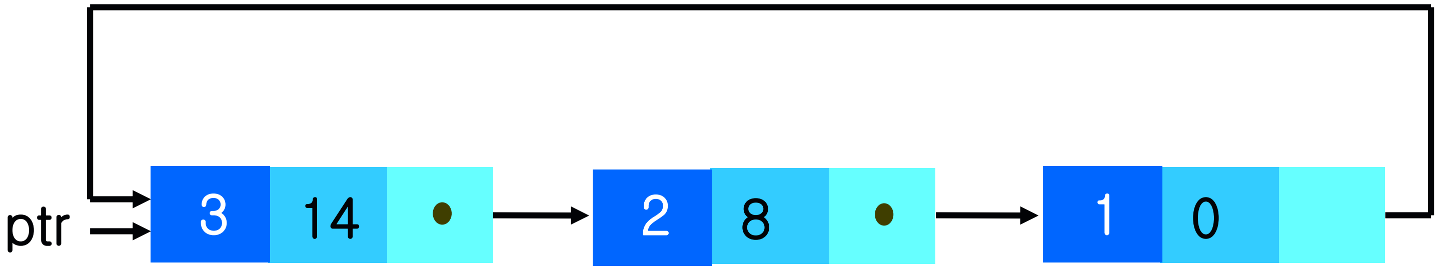
```
poly_pointer  a, b, d, e;
⋮
a = read_poly();
b = read_poly();
d = read_poly();
temp = pmult(a, b);
e = padd(temp, d);
print_poly(e);
```

- Note that we create polynomial *temp(x)* only to hold a partial result for d(x).
- By returning the nodes of *temp(x)*, we may use them to hold other polynomials.
- **[Program 4.12]**

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

4.4.4 Representing Polynomials As Circularly Linked Lists

- To free all the nodes of a polynomial more efficiently, we modify our list structure so that the link field of the last node points to the first node in the list.



- We call this a *circular list*.
- A *chain* : a singly linked list in which the last node has a null link.

- We want to free nodes that are no longer in use so that we may reuse these nodes later.
- We can obtain an efficient erase algorithm for circular lists, by maintaining our own list (as a chain) of nodes that have been "freed".
- When we need a new node, we examine this list.

  If the list is not empty, then we may use one of its nodes.

  Only when the list is empty, use *malloc* to create a new node.
- Let *avail* be a variable of type *poly_pointer* that points to the first node in the list of freed nodes.

- **[Program 4.13]**

```
poly_pointer get_node(void) {
    /* provide a node for use */
    poly_pointer node;
    if (avail)  {
        node = avail;
        avail = avail->link;
    }
    else  {
        node = (poly_pointer) malloc(sizeof(poly_node));
        if (IS_FULL(node))  {
            fprintf(stderr, "The memory is full□n");
            exit(1);
        }
     }
    return node;
}
```

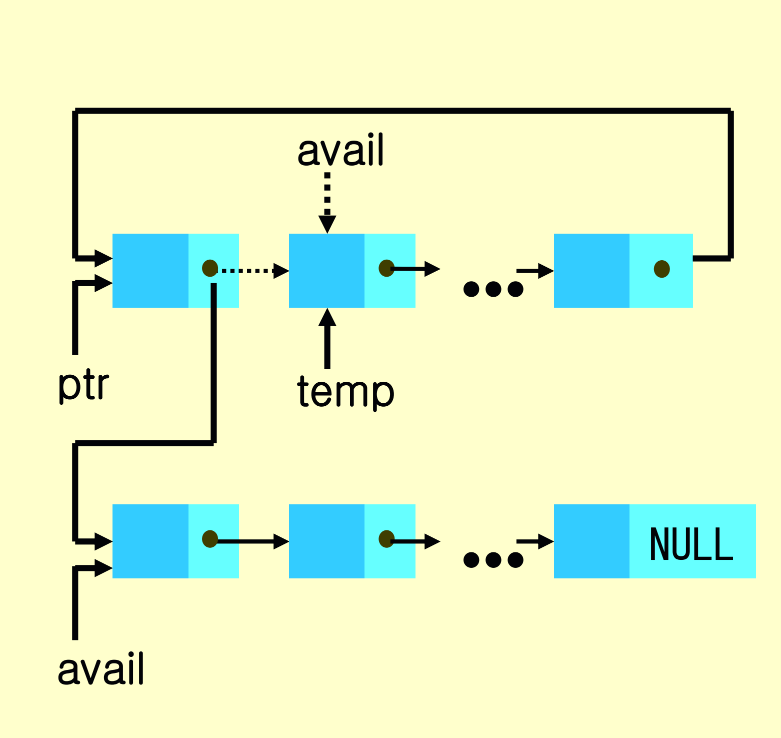
- **[Program 4.14]**

```
void ret_node(poly_pointer ptr)  {
      /*  return a node to the available list */
      ptr->link = avail;
      avail = ptr;
}
```

- **[Program 4.15]**

```
void cerase(poly_pointer *ptr)  {
      /* erase the circular list ptr */
      poly_pointer  temp;
      if (*ptr)  {
            temp = (*ptr)->link;
            (*ptr)->link = avail;
            avail = temp;
            *ptr = NULL;
      }
}
```

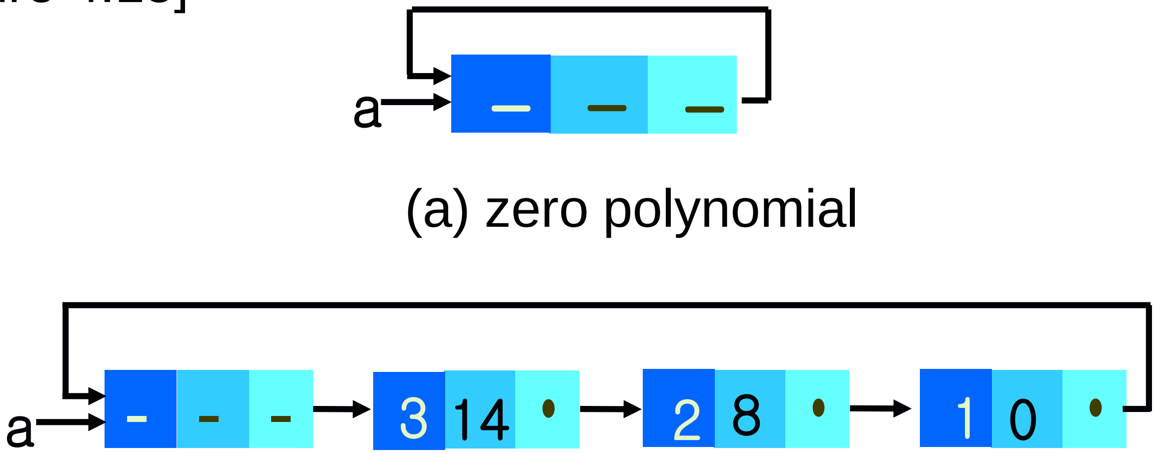
[Figure 4.14]



- A direct changeover to the structure of Figure 4.13 creates problems when we implement the other polynomial operations since we must handle the zero polynomial as a special case.

- We introduce a *head node* into each polynomial.

[Figure 4.15]



(a) zero polynomial

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

- For the circular list with head node representation, we may remove the test for *(*ptr*) from *cerase*.
- The only changes that we need to make to *padd* are :

(1) Add two variables, *starta = a* and *startb =* b.

(2) Prior to the *while* loop, assign *a = a->link* and *b = b->link*.

(3) Change the *while* loop to *while* (*a* != *starta* && *b* != *startb*).

(4) Change the first *for* loop to *for* (; *a* != *starta*; *a* = *a*->*link*).

(5) Change the second *for* loop to *for* (; *b* != *startb*; *b* = *b*->*link*).

(6) Delete the lines :

rear -> link = NULL;

/* delete extra initial node */

(7) Change the lines :

temp = front;

front = front -> link;

free(temp);

to

rear -> link = front;

- We may further simplify the addition algorithm
  if we set the *expon* field of the head node to -1.

- **[Program 4.16]**

```
poly_pointer cpadd(poly_pointer a, poly_pointer b)
{
/* polynomials a and b are singly linked circular lists with  a head
  node.  Return a polynomial which is the sum of  a and b */
      poly_pointer starta, d, lastd;
      int sum,  done = FALSE;
      starta = a;                 /*  record start of a  */
      a = a->link;                /*  skip head node for a and b  */
      b = b->link;
      d = get_node();             /(  get a head node for sum  */
      d->expon = -1;              lastd = d;
```

```
    do  {
      switch (COMPARE(a->expon, b->expon)){
      case –1 :  /* a->expon < b->expon */
        attach (b->coef, b->expon, &lastd);
        b = b->link;      break;
      case  0 : /* a->expon = b->expon */
        if (starta == a)  done = TURE;
        else  {
            sum =  a->coef + b->coef;
            if (sum)   attach(sum, a->expon, &lastd);
            a = a->link;   b = b->link;
        }
        break;
       case  1 :  /* a->expon > b->expon */
        attach (a->coef, a->expon, &lastd);
        a = a->link;
       }
    } while (!done)
        lastd->link = d;
    return d;
}
```

4.5 ADDITIONAL LIST OPERATIONS

4.5.1 Operations For Chains

- It is often necessary, and desirable to build a variety of functions for manipulating singly linked lists. We have seen *get_node* and *ret_node*.

- We use the following declarations :

```
typedef  struct  list_node  *list_pointer;
typedef  struct  list_node  {
        char  data;
        list_pointer  link;
};
```

- ***Inverting a chain*** **:**
    - we can do it "in place" if we use three pointers.
- **[Program 4.17]**

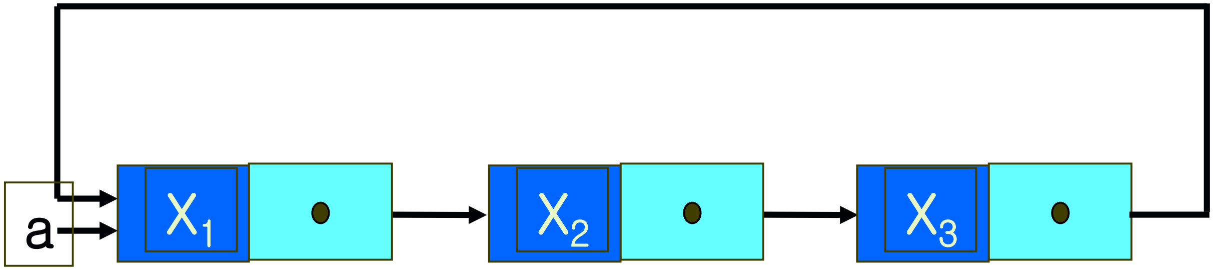
```
list_pointer invert(list_pointer lead)
{
    /* invert the list pointed to by lead  */
    list_pointer  middle, trail;
    middle = NULL;
    while (lead)  {
        trail = middle;
        middle = lead;
        lead = lead->link;
        middle->link = trail;
    }
    return  middle;
}
```

- ***Concatenating two chains*** **:**
- **[Program 4.18]**

```
list_pointer concatenate(list_pointer ptr1, list_pointer ptr2)
{
    /* produce a new list that contains the list ptr1 followed
    by the list ptr2. The list pointed to by ptr1 is changed
    permanently */
    list_pointer  temp;
    if (IS_EMPTY(ptr1))  return ptr2;
    else  {
        if (!IS_EMPTY(ptr2))  {
            for (temp = ptr1; temp->link; temp = temp->link)
                ;
            temp->link = ptr2;
        }
        return  ptr1;
    }
}
```

4.5.2 Operations For Circularly Linked Lists

- ***Inserting a new node at the front of a circular list*** **:**
  - Since we have to change the link field of the last node, we must move down the list until we find the last node.
  - It is more convenient if the name of the circular list points to the last node rather than the first.



- **[Program 4.19]**

```
void insert_front(list_pointer *ptr, list_pointer node)
    /* insert node at the front of the circular list ptr,
    where ptr is the last node in the list.    */
{
    if (IS_EMPTY(*ptr))  {
        /* list is empty, change ptr to point to new entry */
        *ptr = node;
        node->link = node;
    }
    else  {
        /* list is not empty, add new entry at front */
        node->link = (*ptr)->link;
        (*ptr)->link = node;
    }
}
```

- ***Inserting a new node at the rear of a circular list*** **:**
  We only need to add the additional statement **ptr = node* to the *else* clause of *insert_front*.

- **[Program 4.20]**

```
int length(list_pointer ptr)
{
    /* find the length of the circular list ptr */
    list_pointer temp;
    int count = 0;
    if (ptr)  {
        temp = ptr;
        do  {
            count++;
            temp = temp->link;
        } while (temp != ptr);
    }
    return count;
}
```

4.6 EQUIVALENCE RELATIONS

- R is a *binary relation* on a set S if $R \subseteq S \times S$.

  If $(a, b) \in R$ then we may write aRb.

  - R is *reflexive* if aRa for all $a \in S$.
  - R is *symmetric* if aRb implies bRa.
  - R is *transitive* if aRb and bRc implies aRc.

  - R is an *equivalence relation* over S

    if R is reflexive, symmetric and transitive over S.

- **[Example]**
  - One of the steps in the manufacture of a VLSI circuit involves exposing a silicon wafer using a series of masks. Each mask consists of several polygons. Polygons that overlap electrically are equivalent and electrical equivalence specifies an equivalence relation ≡ over the set of mask polygons.

(1) For any polygon x, x≡x, that is, x is electrically equivalent to itself. Thus, ≡ is reflexive.

(2) For any two polygons, x and y, if x≡y then y≡x. Thus, the relation ≡ is symmetric.

(3) For any three polygons, x, y, and z, if x≡y and y≡z then x≡z. For example, if x and y are electrically equivalent and y and z are also equivalent, then x and z are also electrically equivalent.

Thus the relation ≡ is transitive.

- Any equivalence relation R over S can partition the set S into disjoint subsets called *equivalence classes*.

- An *equivalence class* E is a subset of S such that if x is in E then E contains every element which is related to x by R. That is, for any $x \in S$, $[x] = \{y \mid y \in S \text{ and } x \equiv y\}$.

- For any x and y in S, either $[x] = [y]$ or $[x] \cap [y] = \emptyset$.

- **Example :**
  - If we have 12 polygons numbered 0 through 11

    and the following pairs overlap :

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

  - as a result of the reflexivity, symmetry, and transitivity of the relation ≡, we can obtain the following equivalence classes :

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

- **The algorithm to determine equivalence works in two phases :**
  - *First phase* : read in and store the equivalence pairs.
  - *Second phase* : determining equivalence class as follows

    we begin at 0 find all pairs of the form <0, j>.

    By transitivity, find all pairs of the form <j, k>.

    /* <0, j> and <j, k> ⇒ <0, k> i.e, 0≡j and j≡k ⇒ 0≡k */

    We continue in this way until we have found, marked, and printed the entire equivalence class containing 0.

    Then we continue on.

- **<u>Our first design attempt</u> :**
- **[Program 4.21]**

```
void equivalence()
{
    initialize;
    while (there are more pairs)  {
        read the next pair <i, j>;
        process this pair;
    }
    initialize the output;
    do
    output a new equivalence class;
    while  (not done);
}
```

- Let *m* and *n* represent the number of related pairs and the number of objects, respectively.

- We must first figure out which data structure we should use to hold these pairs.

- The pair <i, j> is essentially two random integers in the range 0 to n-1.

- Use an array, *pairs*[n][m], for easy random access.
  this could waste a lot of space and require considerable time
  or use more storage to insert a new pair.

- These considerations lead us to a linked representation for each row.
- Since we still need random access to the *i* -th row, we use a one-dimensional array, *seq*[n], to hold the head nodes of the *n* lists.
- In the second phase of the algorithm, we need to check whether or not the object, *i*, has been printed.
- We use the array *out*[n].

- **[Program 4.22]**

```
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;
        }
}
```

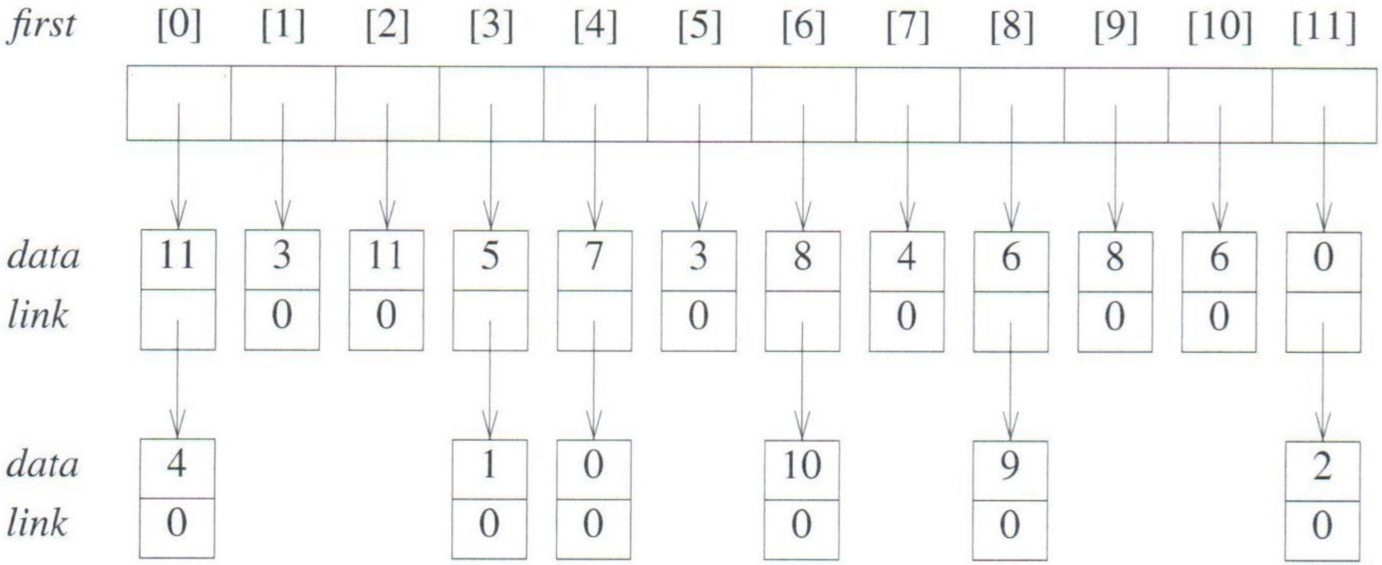


Figure 4.16: Lists after pairs have been input

- In phase two :
  - We scan the *seq* array for the first *i*, $0 \leq i < n$, 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.
- For the complete equivalence algorithm, see the following declaration and Program 4.22.

```
#include <stdio.h>
#include <alloc.h>
#define MAX_SIZE  24
#define IS_FULL (ptr)  (!(ptr))
#define FALSE 0
#define TRUE 1

typedef  struct node *node_pointer;
typedef  struct node {
      int data;
      node_pointer link;
};
```

```
void main(void)
{
      short  int out[MAX_SIZE];
      node_pointer  seq[MAX_SIZE];
      node_pointer 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)   {
        x = (node_pointer)malloc(sizeof(node));
        if (IS_FULL(x))   {
                fprintf(stderr, "The memory is full□n");
                exit(1);
        }
        x->data = j;  x->link = seq[i];  seq[i] = x;
        x = (node_pointer)malloc(sizeof(node));
        if (IS_FULL(x))   {
                fprintf(stderr, "The memory is full□n");
                exit(1);
        }
        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 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 */
            }
        }
    }
}
```

- Analysis of the equivalence program :
  - Initialization of *seq* and *out* takes O(n) time.
  - Each of Phase 1 and 2 takes O($m + n$) time.
  - Time complexity is O(m+n) and space complexity is also O(m+n).
  - In Chapter 5, we will look at an alternate solution that requires only O(n) space.

4.7 SPARSE MATRIX

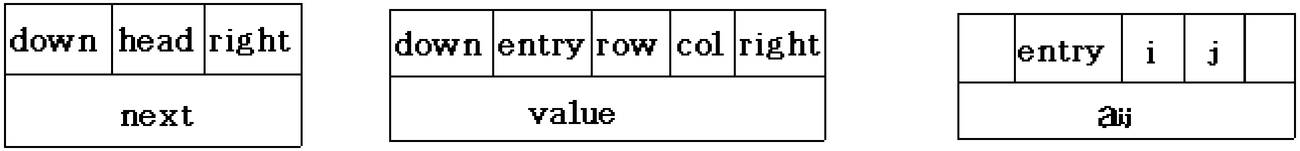
- In Chapter 2, we considered a sequential representation of sparse matrices and implemented matrix operations.

- However we found that the sequential representation of sparse matrices suffered from the same inadequacies as the similar representation of polynomials.

- As we have seen previously, linked lists allow us to efficiently represent structures that vary in size, a benefit that also applies to sparse matrices.

- In our data representation, we represent each column of a sparse matrix as a circularly linked list with a head node. We use a similar representation for each row of a sparse matrix.

[Figure 4.19]



(a) head node (b) entry node (c) set up for a_{ij}

[Figure 4.20]

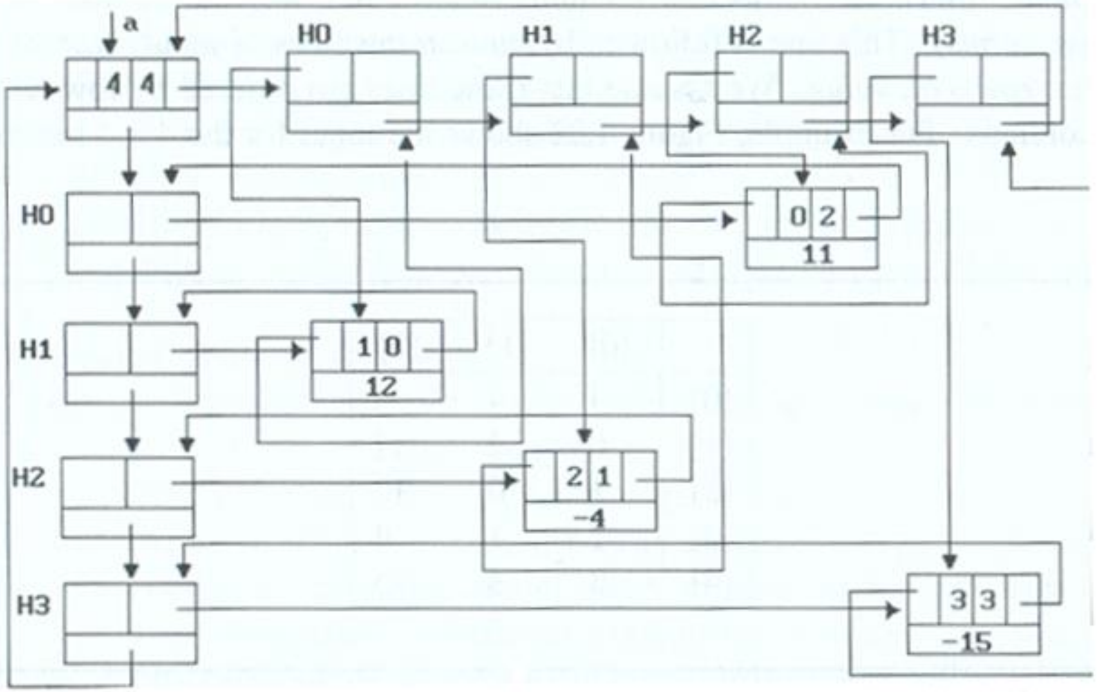
$$\begin{bmatrix} 0 & 0 & 11 & 0 \\ 12 & 0 & 0 & 0 \\ 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & -15 \end{bmatrix}$$

- **[Figure 4.21]**
  - 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 the same structure as an entry node.



NOTE: The tag field of a node is not shown; its value for each node should be clear from the node structure.

Figure 4.21: Linked representation of the sparse matrix *a*

```
#define MAX_SIZE 50
typedef enum {head, entry} tagfield;
typedef  struct  matrix_node  *matrix_pointer;
typedef  struct  entry_node  {
    int row;
    int col;
    int value;
};
typedef struct matrix_node {
    matrix_pointer down;
    matrix_pointer right;
    tagfield tag;
    union {
        matrix_pointer next;
        entry_node entry;
    } u;
    matrix_pointer hdnode[MAX_SIZE];
matrix_pointer mread()
{
    int num_rows, num_cols,num_terms, num_heads,i;
    int row,  col, value, current_row;
    matrix_pointer temp, last, node;

    scanf(&num_rows, &num_cols, &num_terms);
    num_heads = (num_cols > num_rows) ? num_cols : num_rows;
    node = new_node(); node_tag = entry;
    node->u.entry.row = num_rows;
    node->u.entry.col = num_cols;
```

```
if (!num_heads) node->right = node;
else {
      for (i=0; i<num_heads; i++) {
            temp = new_node();
            hdnode[i] = temp; hdnode[i]->tag = head;
            hdnode[i]->right = temp; hdnode[i]->u.next=temp;
      }
      current_row = 0;  last = hdnode[0];
      for (i=0; i<num_terms; i++) {
            scanf(&row, &col, &value);
            if (row > current_row) {
                  last->right = hdnode[current_row];
                  current_row = row;  last = hdnode[row];
            }
            temp = new_node();  temp->tag = entry;
            temp->u.entry.row = row;  temp->u.entry.col = col;
            temp->u.entry.value = value;  last->right = temp;  last = temp;
            hdnode[col]->u.next->down = temp;
            hdnode[col]->u.next = temp;
      }
```

```
    // close last row
    last->right = hdnode[current_row];
    // close all column lists
    for (i=0; i<num_cols; i++)
        hdnode[i]->u.next->down = hdnode[i];
    // link all head nodes together
    for (i=0; i<num_heads-1; i++)
        hdnode[i]->u.next = hdnode[i+1];
    hdnode[num_heads-1]->u.next = node;
    node->right = hdnode[0];
  }
  return node;
}
```

```
// print out the matrix in row major form
void mwrite(matrix_pointer node)
{
    int i;
    matrix_pointer temp, head = node->right;

    for (i=0; i<node->u.entry.row; i++) {
        for (temp = head->right; temp != head;
            temp = temp->right)
            printf(temp->u.entry.row, temp->u.entry.col,
            temp->u.entry.value);
        head = head->u.next;
    }
}
```

```
void merase(matrix_pointer *node)
{
    int i, num_heads;
    matrix_pointer x,y, head = (*node)->right;

    for (i=0; i<(*node)->u.entry.row; i++) {
        y = head->right;
        while (y != head) {
            x = y;  y = y->right;  free(x);
        }
        x = head;  head = head->u.next;  free(x);
    }
    // free remaining head nodes
    y = head;
    while (y != *node) {
        x = y;  y = y->u.next;  free(x);
    }
    free(*node);  *node = NULL;
}
```

- **Analysis of *mread* : [Program 4.24]**
  O(max{*num_rows*, *num_cols*} + *num_terms*)
  = O(*num_rows* + *num_cols* + *num_terms*).

- **Analysis of *mwrite* : [Program 4.26]**
  O(*num_rows* + *num_terms*).

- **Analysis of *merase* : [Program 4.27]**
  O(*num_rows* + *num_cols* + *num_terms*).

4.8 DOUBLY LINKED LISTS

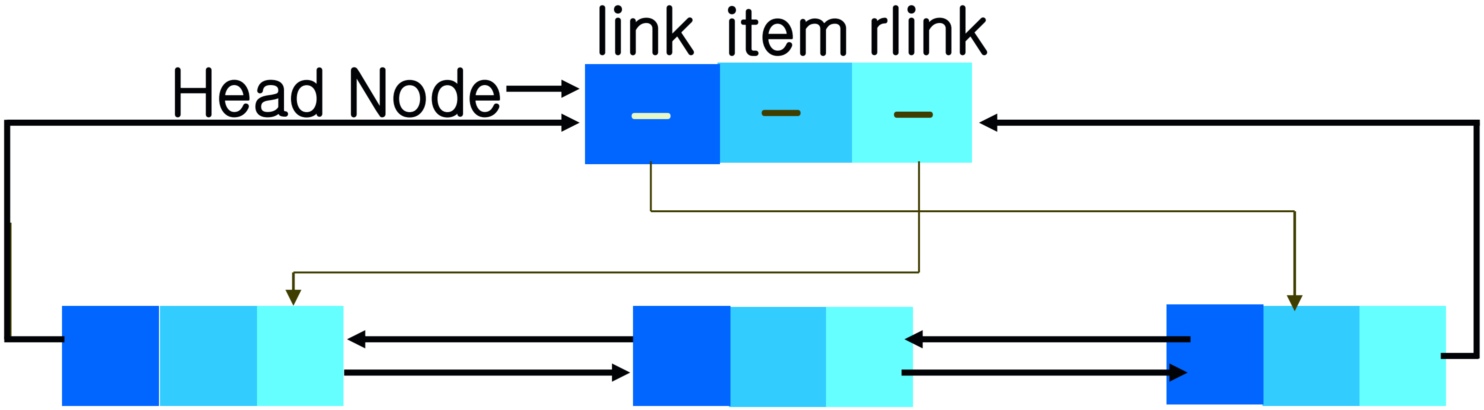
- Singly linked lists pose problems because we can move easily only in the direction of the links.

- Whenever we have a problem that requires us to move in either direction, it is useful to have doubly linked lists.

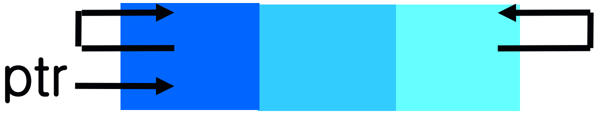
- The necessary declarations are :

```
typedef  struct  node  *node_pointer;
typedef  struct  node  {
     node_pointer  llink;
     element  item;
     node_pointer  rlink;
};
```

- A doubly linked list may or may not be circular.
- **[Figure 4.23] Doubly linked circular list with head node**



- **[Figure 4.24] Empty doubly linked circular list with head node**



- Now suppose that *ptr* points to any node in a doubly linked list.

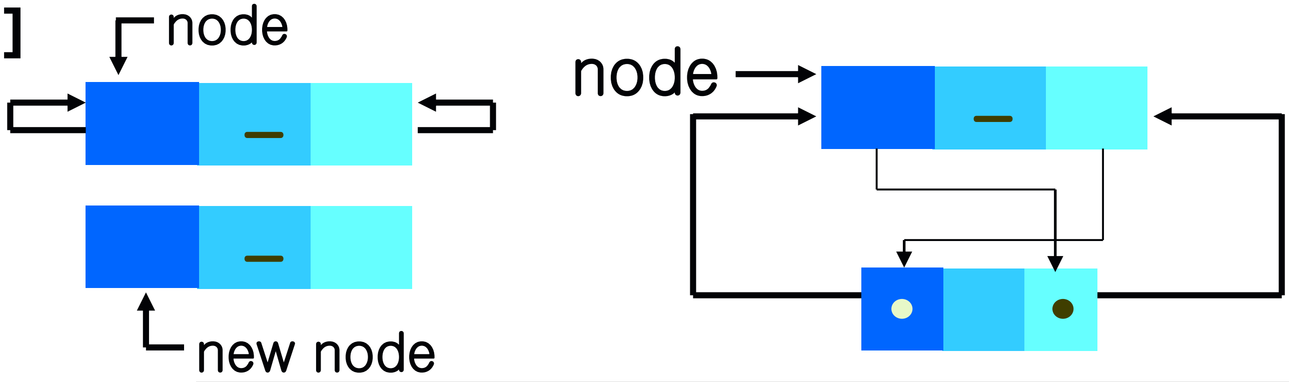
  Then :

  ptr == ptr->llink->rlink == ptr->rlink->llink

- **Insertion into a doubly linked circular list :**
- **[Program 4.28]**

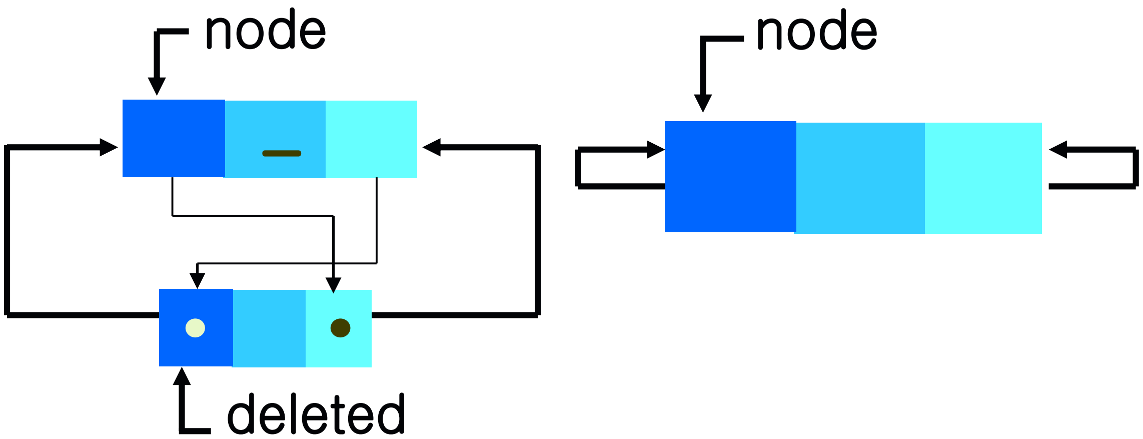
```
void dinsert(node_pointer node, node_pointer newnode)
{
        /* insert newnode to the right of node */
        newnode->llink = node;
        newnode->rlink = node->rlink;
        node->rlink->llink = newnode;
        node->rlink = newnode;
}
```

[Figure 4.25]



- **Deletion from a doubly linked circular list :**
- **[Program 4.29]**

```
void ddelete(node_pointer node,  node_pointer deleted)  {
        /* delete from the doubly linked list  */
        if (node == deleted)
                printf("Deletion of head node not permitted.□n");
        else  {
                deleted->llink->rlink = deleted->rlink;
                deleted->rlink->llink = deleted->llink;
                free(deleted);
        }
}
```



[Figure 4.26]