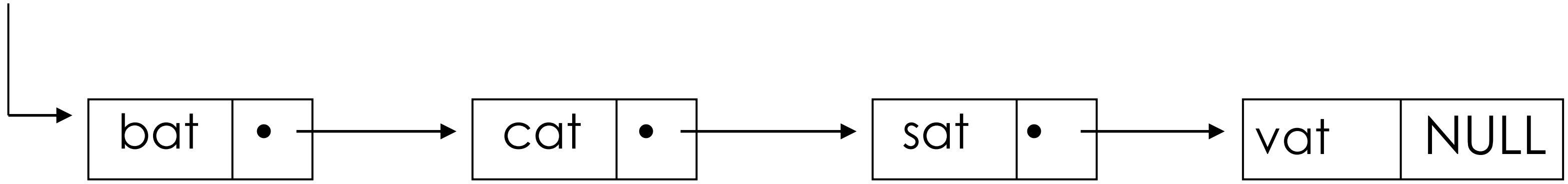
DATA STRUCTURE

Chapter 04: Lists

INTRODUCTION

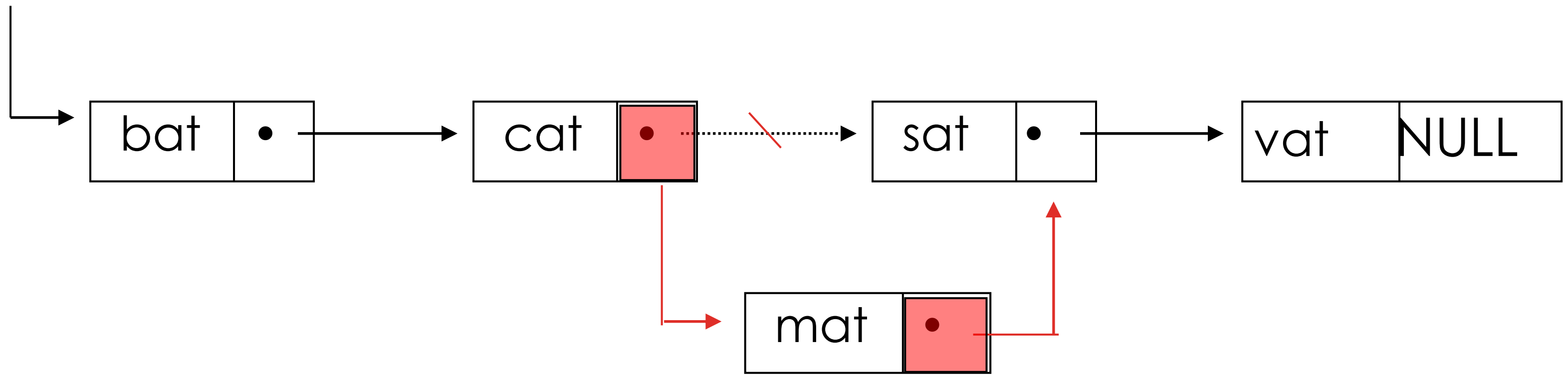
- Array
  successive items locate a fixed distance
- disadvantage
  - data movements during insertion and deletion
  - waste space in storing n ordered lists of varying size
- possible solution
  - linked list
    - Singly
    - Doublly
    - Circularly

Singly-linked list

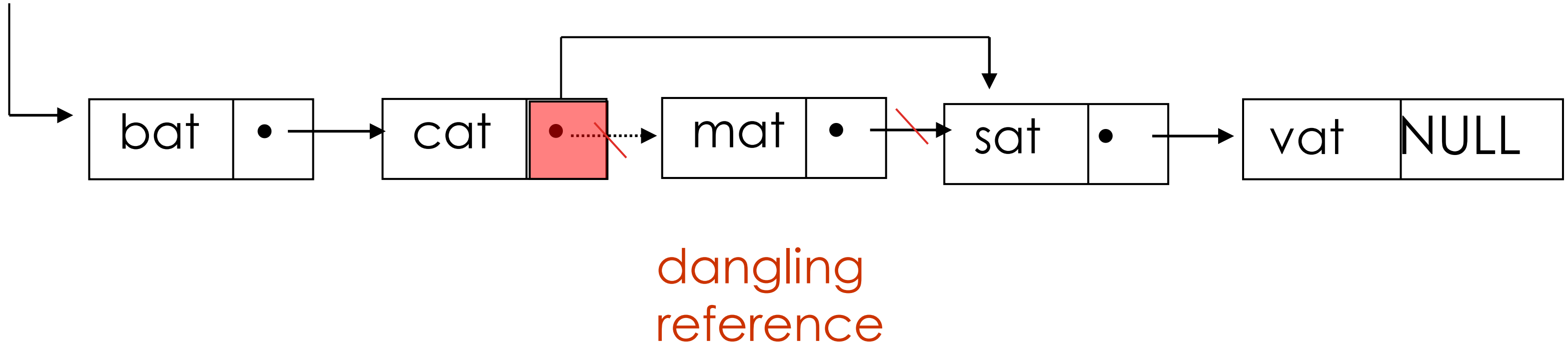


SINGLY-LINKED LIST - OPERATIONS

Insertion



Delete mat from list



EXAMPLE 4.1: CREATE A LINKED LIST OF WORDS

- Declaration

```
typedef struct list_node *list_pointer;
typedef struct list_node {
        char data[4];
        list_pointer link;
        };
```

Creation (a new empty list called ptr)

```
list_pointer ptr = NULL;
```

Testing for an empty list

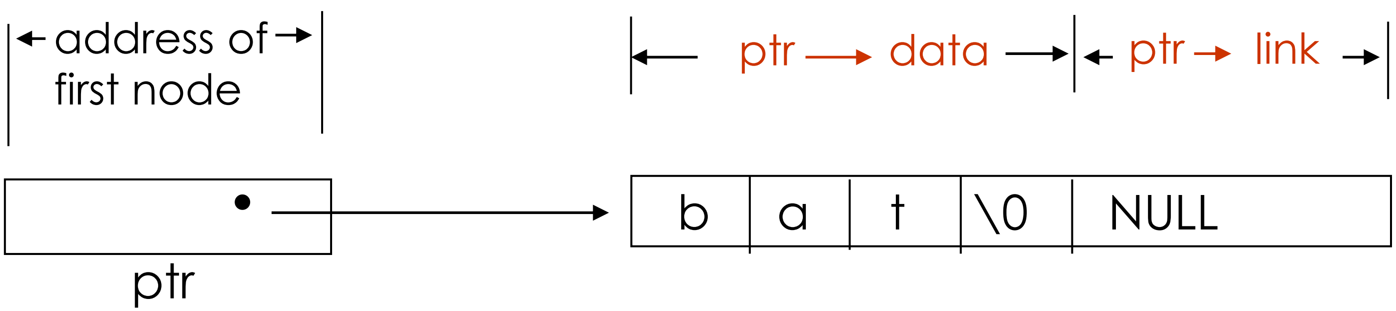
```
#define IS_EMPTY(ptr) (!(ptr))
```

Allocation

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

REFERENCING THE FIELDS OF A NODE

e -> name ← (*e).name
n=4;
strcnpy(ptr -> data, “bat”, n);
ptr -> link = NULL;

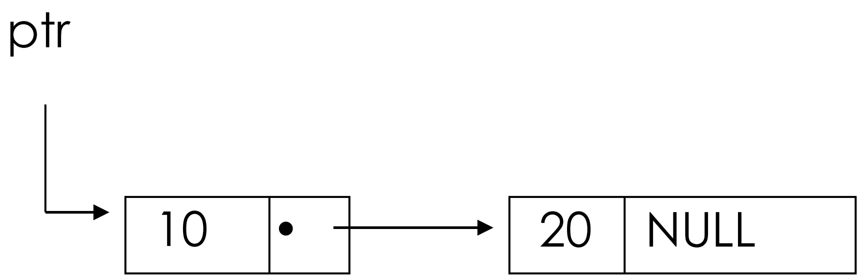


EXAMPLE: CREATE A TWO NODES LIST (STRING DATA)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct ListNode *ListNode;
typedef struct ListNode {
   char data[4];
   ListNode* link;
};

void main() {
   ListNode first, second;
 // Create the first node using malloc()
   first = (ListNode*) malloc(sizeof(struct ListNode));
   strncpy(first->data, "bat", 4);
   first->link = NULL;
 // Create the second node using malloc()
   second = (ListNode*) malloc(sizeof(struct ListNode));
   strncpy(second->data, "cat", 4);
   second->link = NULL;

   // Link the nodes together
   first->link = second;
}
```

```
// Traverse the linked list and print
the data in each node
   // create a listnode pointer
   ListNode current = first;
   while (current != NULL) {
      printf("%s\n", current->data);
      current = current->link;
   }

 // Free the memory used by
 the nodes
    free(first);
    free(second);
```

EXAMPLE: CREATE A TWO NODES LIST (INT DATA)



Example 4.2

```
typedef struct ListNode *ListNode;
typedef struct ListNode {
    int data;
    ListNode* link;
};
```

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

***Program 4.2:**Create a two-node list

USE OF A POINTER (1)

int i, *pi;

	Address	Value
i	1000	?
pi	2000	?

pi = &i;

	Address	Value
i / *pi	1000	?
pi	2000	1000

i = 10 or *pi = 10

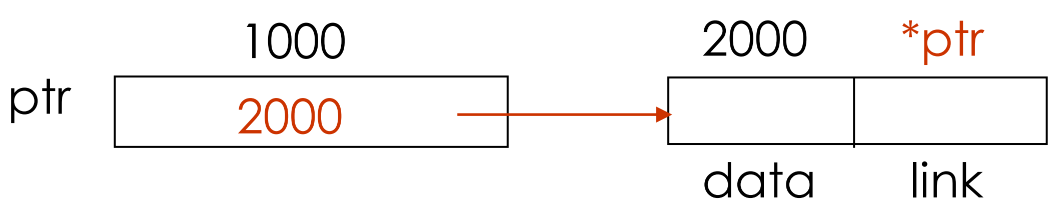
	Address	Value
i / *pi	1000	10
pi	2000	1000

USE OF A POINTER (2)

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

ptr->data ← (*ptr).data

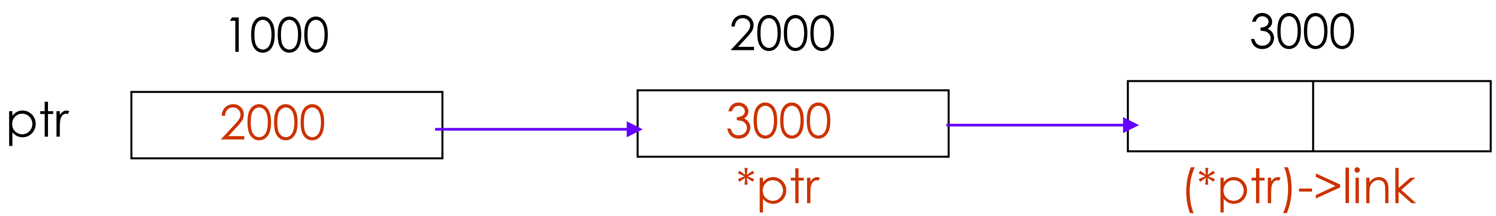
```
ptr = malloc(sizeof(ListNode));
```



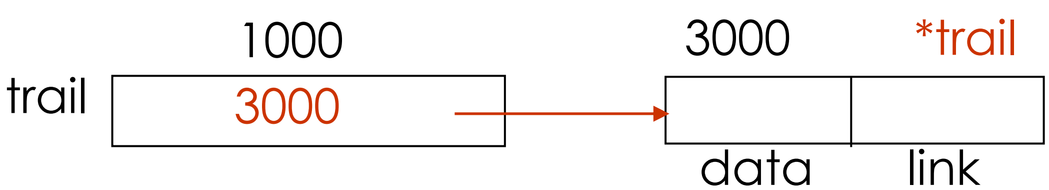
USE OF A POINTER (3)

void delete(ListNode *ptr, ListNode trail, ListNode node)

ptr: a pointer point to a pointer point to a list node



trail (node): a pointer point to a list node



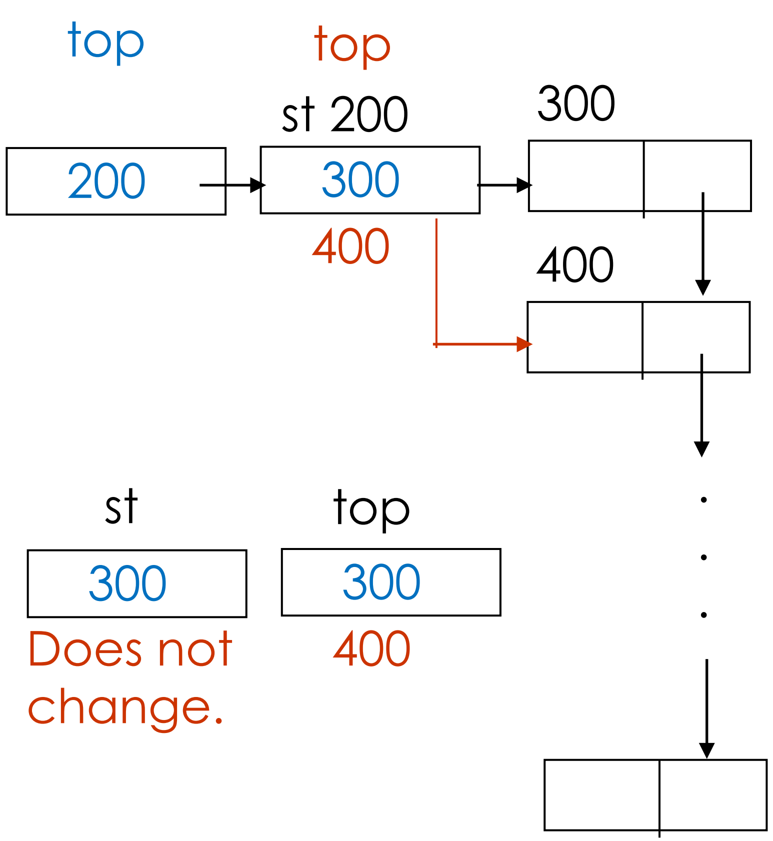
trail->link 🡐 (*trail).link

USE OF A POINTER (4)

element delete(stack_pointer *top)

delete(&st) vs. delete(st)

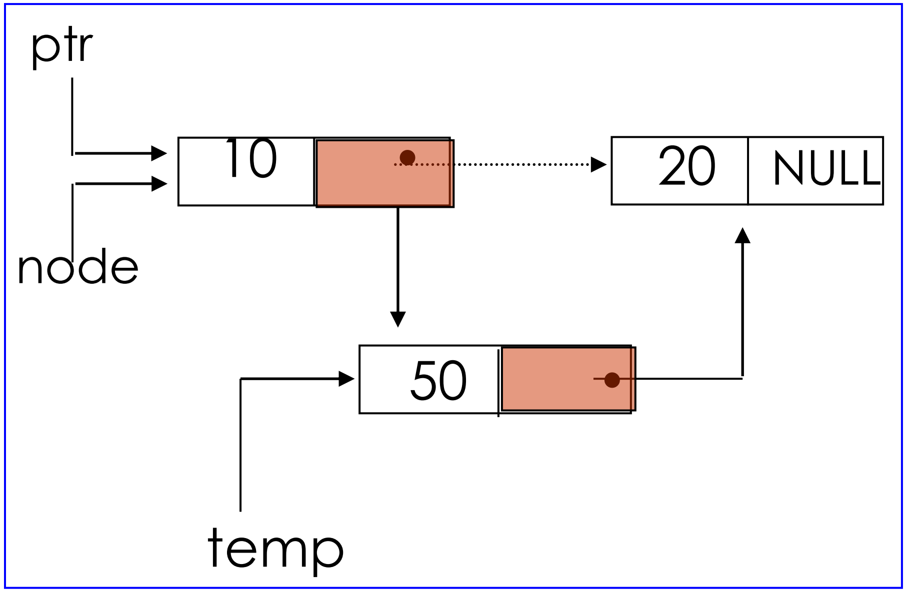
200 300



LIST INSERTION

- Insert a node after a specific node

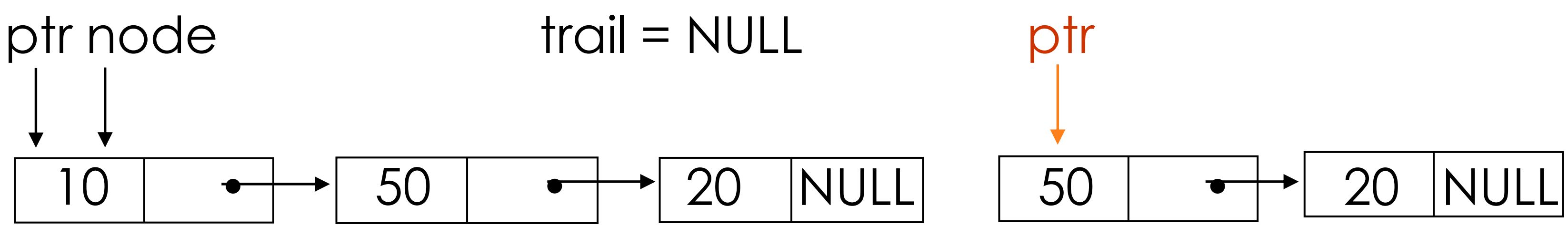
```
void insert(ListNode *ptr, ListNode node)
{
/* insert a new node with data = 50 into the list ptr after node */
    ListNode temp;
    temp = malloc(sizeof(ListNode));

    if (IS_FULL(temp)){
       fprintf(stderr, "The memory is full\n");
       exit (1);
    }

    temp->data = 50;
    if (*ptr) {    nonempty list
        temp->link =node ->link;
        node->link = temp;
    }
    else {    empty list
       temp->link = NULL;
       *ptr =temp;
    }
}
```



***Program 4.3:**Simple insert into front of list

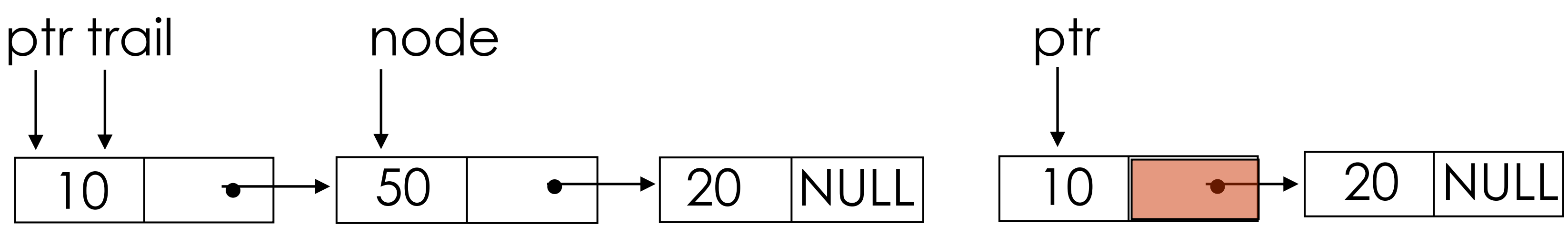
List Deletion

Delete the first node.

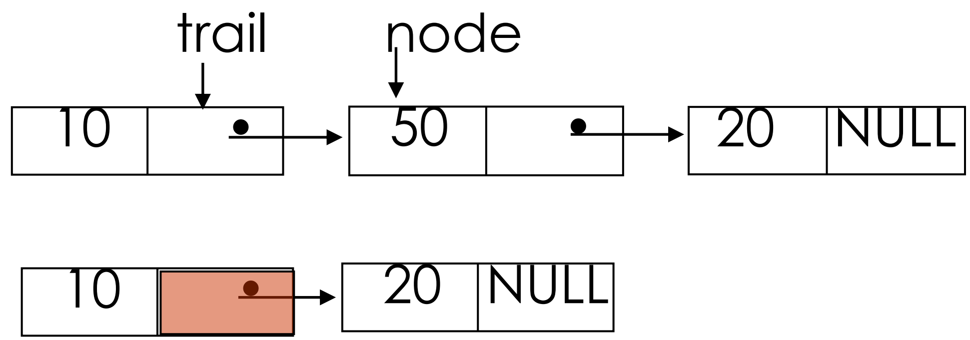


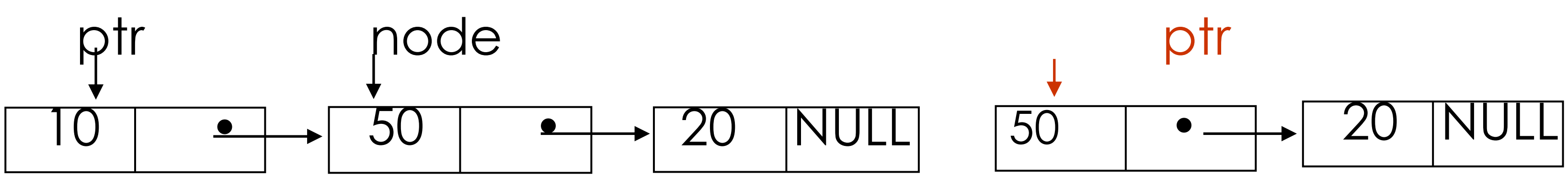
(a) before deletion (b)after deletion

Delete node other than the first node.



```
void delete(ListNode *ptr, ListNode trail, ListNode 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);
}
```





Print out a list (traverse a list)

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

***Program 4.5:** Printing a list

DYNAMICALLY LINKED STACKS AND QUEUES

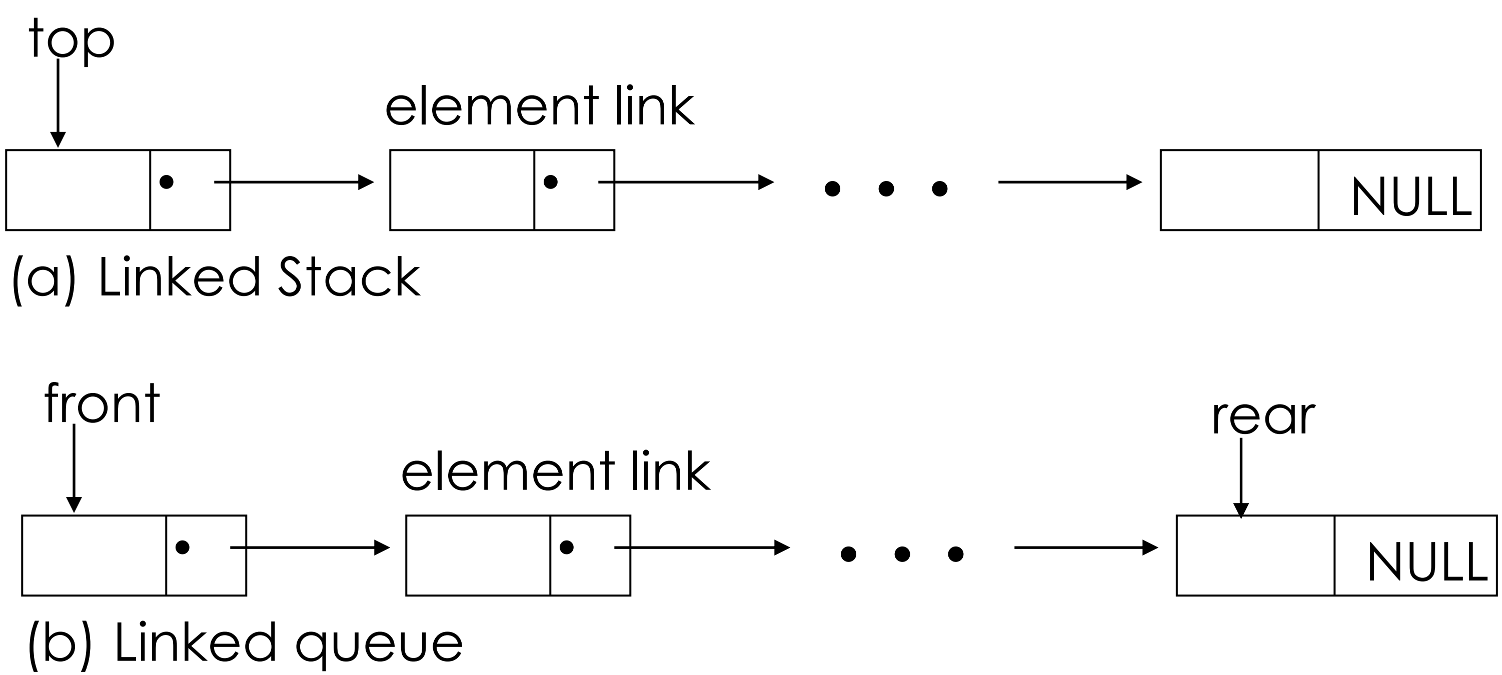


***Figure 4.10:** Linked Stack and queue*

Represent n stacks

```
#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];
```

Represent n queues

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

Push in the linked stack

```
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;          }
```

*__Program 4.6:__ Add to a linked stack

pop from the linked stack

```
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;
}
```

***Program 4.7**: Delete from a linked stack

ENQUEUE IN THE LINKED QUEUE

```
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;   }
```

DEQUEUE FROM THE LINKED QUEUE (SIMILAR TO PUSH)

```
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;
}
```

Polynomials

$$A(x) = a_{m-1}x^{e_{m-1}} + a_{m-2}x^{e_{m-2}} + \ldots + a_0 x^{e_0}$$

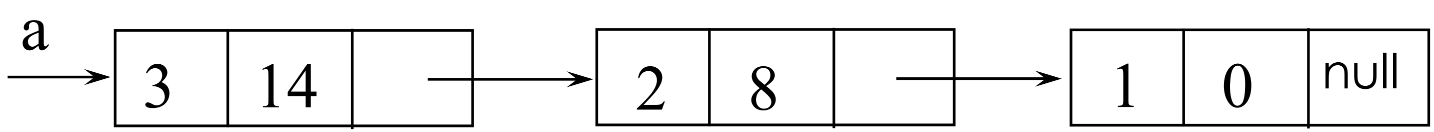
Representation

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

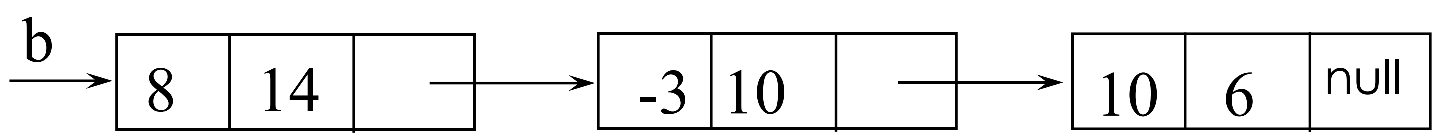
coef	expon	link

Examples

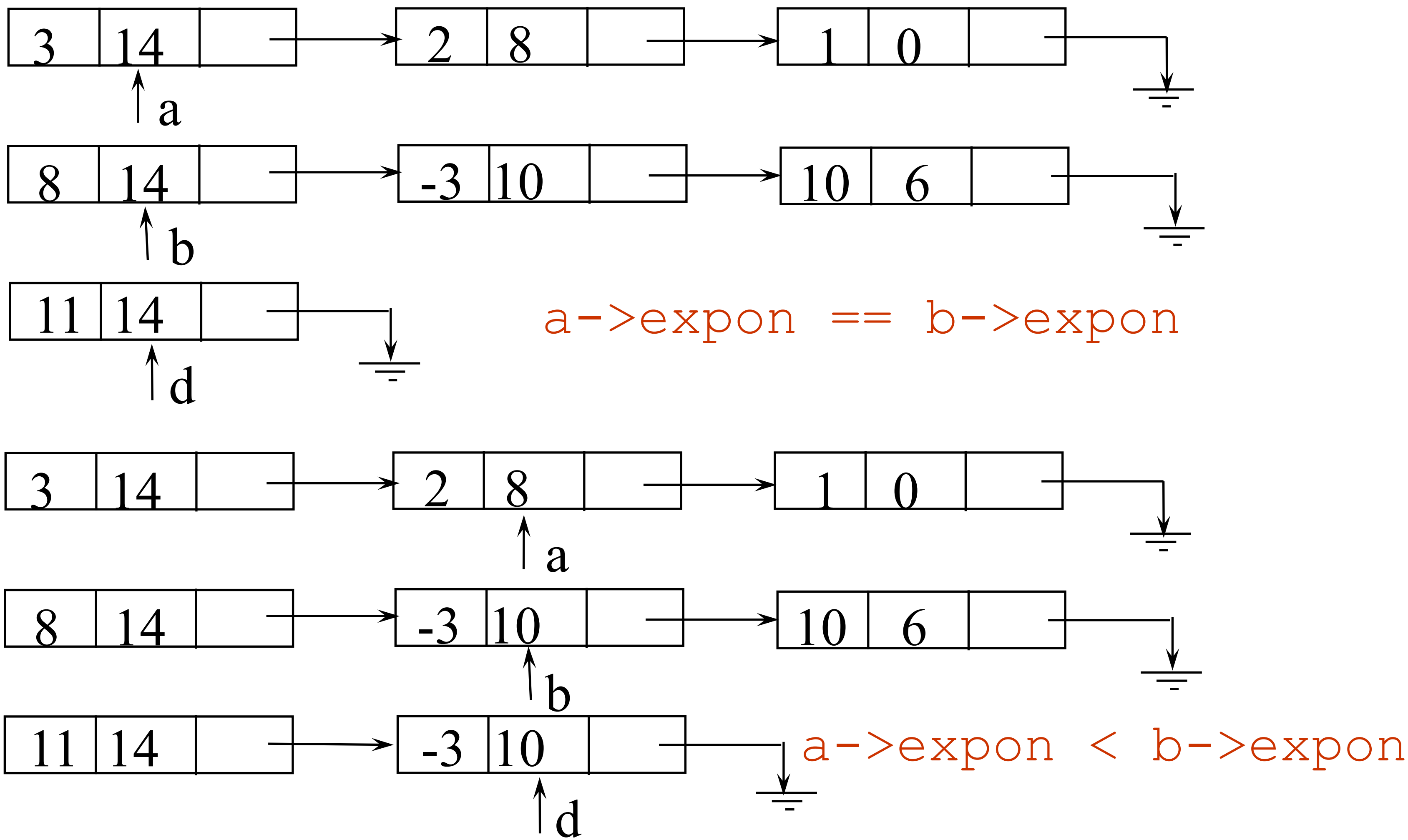
$$a = 3x^{14} + 2x^{8} + 1$$



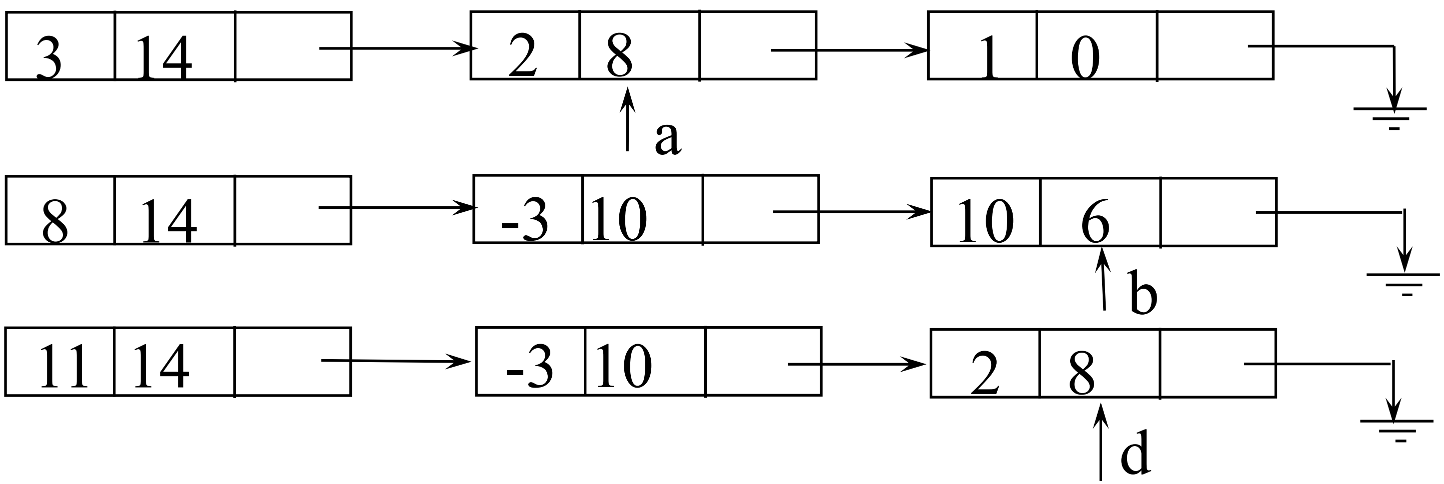
$$b = 8x^{14} - 3x^{10} + 10x^{6}$$



Adding Polynomials



ADDING POLYNOMIALS *(CONTINUED)*



`a->expon > b->expon`

ALOGRITHM FOR ADDING POLYNOMIALS

```
poly_pointer padd(poly_pointer a, poly_pointer b)
{
    poly_pointer front, rear, temp;
    int 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;
        }
    }
    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;
    temp = front;  front = front->link;  free(temp);
    return front;
}
```

Delete extra initial node.

ANALYSIS

(1) coefficient additions
$0 \leq$ additions $\leq \min(m, n)$
where m (n) denotes the number of terms in A (B).

(2) exponent comparisons
extreme case
$e_{m-1} > f_{m-1} > e_{m-2} > f_{m-2} > \ldots > e_0 > f_0$
m+n-1 comparisons

(3) creation of new nodes
extreme case
m + n new nodes
summary O(m+n)

ATTACH A TERM

```
void attach(float coefficient, int exponent,
            poly_pointer *ptr)
{
/* create a new node attaching 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;
}
```

A SUITE FOR POLYNOMIALS

$e(x) = a(x) * b(x) + d(x)$

```
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);
```

```
read_poly()
print_poly()
padd()
psub()
pmult()
```

temp is used to hold a partial result. By returning the nodes of temp, we may use it to hold other polynomials

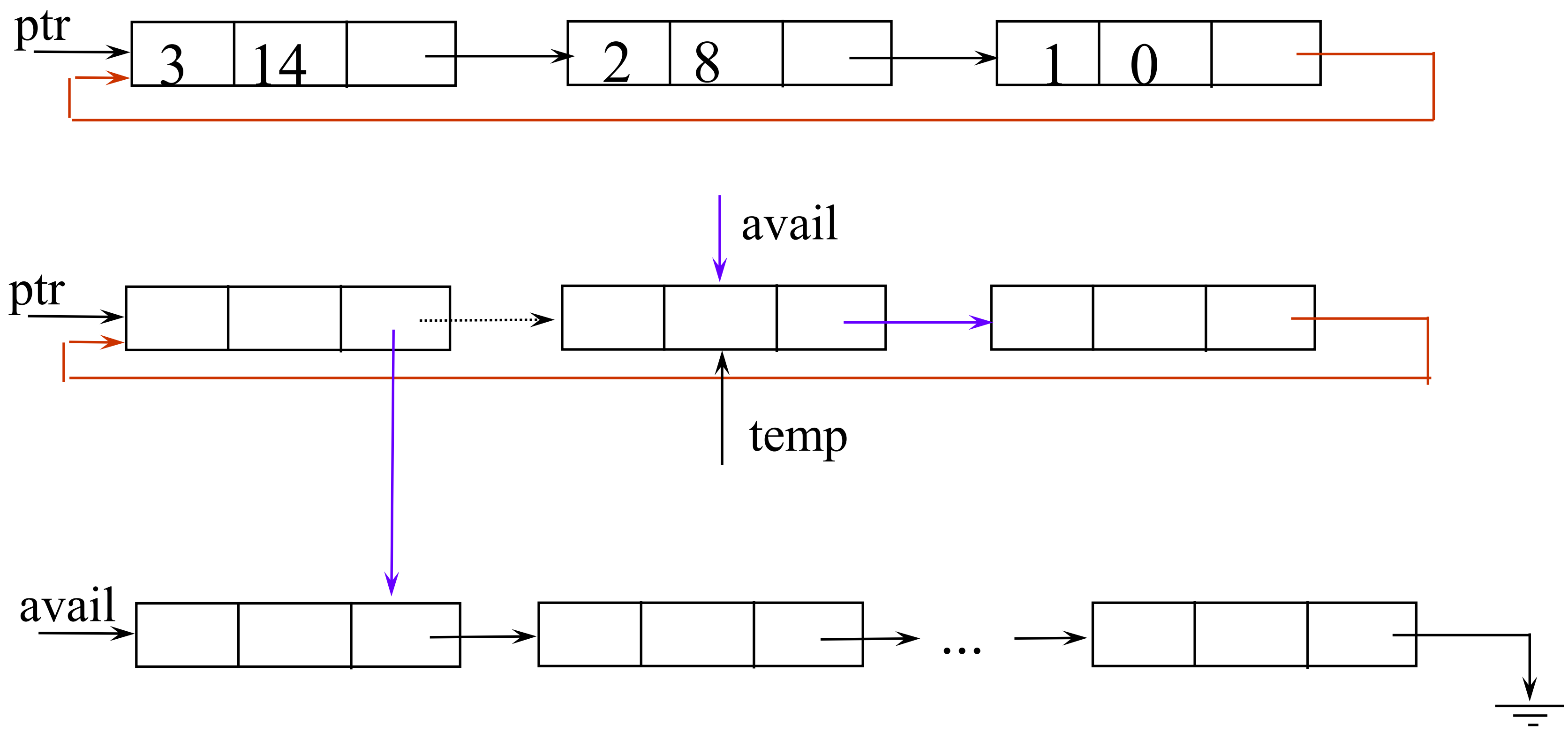
ERASE POLYNOMIALS

```
void earse(poly_pointer *ptr)
{
/* erase the polynomial pointed to by ptr */

    poly_pointer temp;

    while (*ptr) {
        temp = *ptr;
        *ptr = (*ptr)->link;
        free(temp);
    }
}
```

O(n)

CIRCULARLY LINKED LISTS

circular list vs. chain



MAINTAIN AN AVAILABLE LIST

```
poly_pointer get_node(void)
{
  poly_pointer node;
  if (avail) {
      node = avail;
      avail = avail->link:
  }
  else {
      node = (poly_pointer)malloc(sizeof(poly_node));
      if (IS_FULL(node)) {
          printf(stderr, "The memory is full\n");
          exit(1);
      }
  }
  return node;
}
```

MAINTAIN AN AVAILABLE LIST (CONTINUED)

Insert ptr to the front of this list

```
void ret_node(poly_pointer ptr)
{
  ptr->link = avail;
    avail = ptr;
}
```

Erase a circular list (see next page)

```
void cerase(poly_pointer *ptr)
{
    poly_pointer temp;
    if (*ptr) {
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```

(1)

(2)

Independent of # of nodes in a list O(1) constant time

REPRESENTING POLYNOMIALS AS CIRCULARLY LINKED LISTS

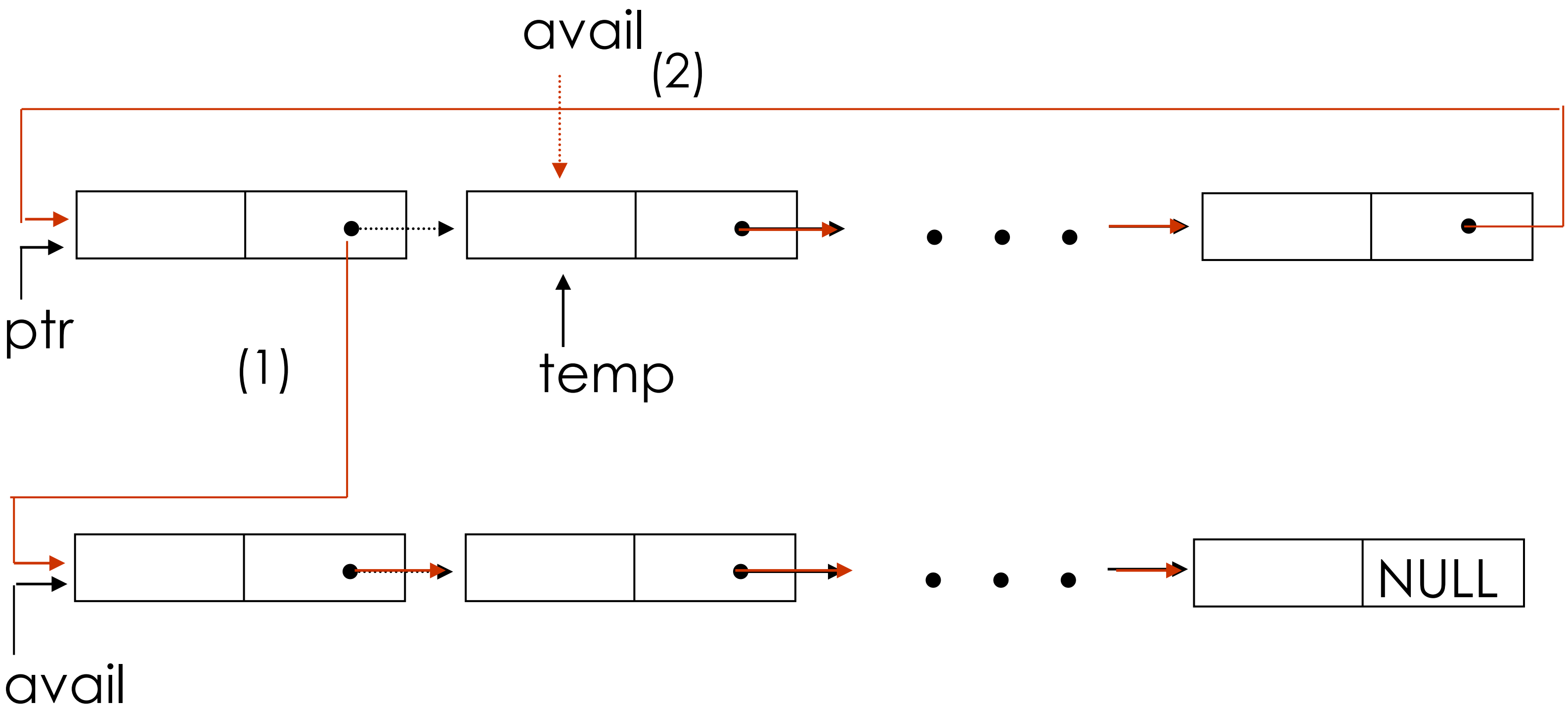
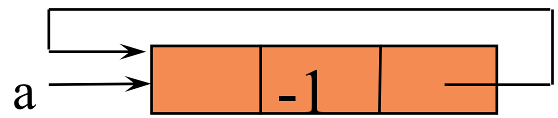


***Figure 4.14:** Returning a circular list to the avail list (p.159)

Head Node

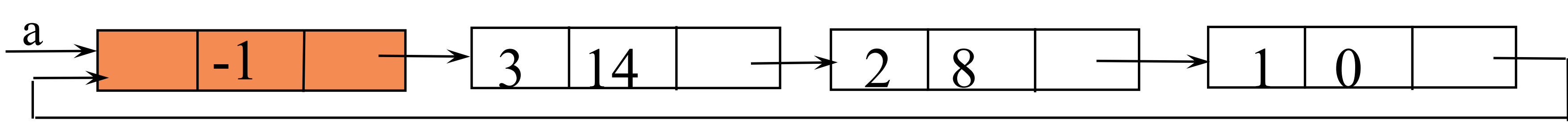
Represent polynomial as circular list.

(1) zero



Zero polynomial

(2) others



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

Another Padd

```
poly_pointer cpadd(poly_pointer a, poly_pointer b)
{
  poly_pointer starta, d, lastd;
  int sum, done = FALSE;
  starta = a;
  a = a->link;
  b = b->link;
  d = get_node();
  d->expon = -1;    lastd = d;
  do {
    switch (COMPARE(a->expon, b->expon)) {
      case -1: attach(b->coef, b->expon, &lastd);
               b = b->link;
               break;
```

Set expon field of head node to -1.

Another Padd *(Continued)*

```
        case 0: if (starta == a) done = TRUE;
                else {
                  sum = a->coef + b->coef;
                  if (sum) attach(sum,a->expon,&lastd);
                  a = a->link;   b = b->link;
                }
                break;
        case 1: attach(a->coef,a->expon,&lastd);
                a = a->link;
      }
    } while (!done);
    lastd->link = d;
    return d;
}
```

Link last node to first

Additional List Operations

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

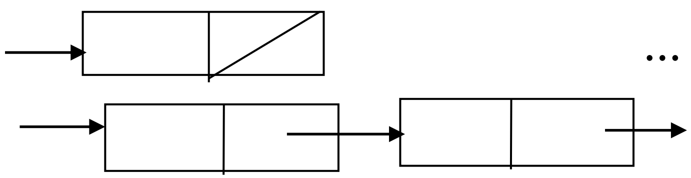
Invert single linked lists
Concatenate two linked lists

INVERT SINGLE LINKED LISTS

Use two extra pointers: middle and trail.

```
list_pointer invert(list_pointer lead)
{
    list_pointer middle, trail;
    middle = NULL;
    while (lead) {
        trail = middle;
        middle = lead;
        lead = lead->link;
        middle->link = trail;
    }
    return middle;
}
```

0: null
1: lead
≥2: lead



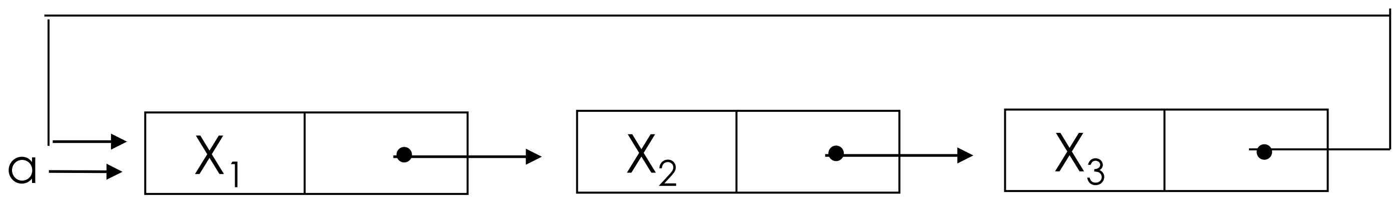
Concatenate Two Lists

```
list_pointer concatenate(list_pointer
             ptr1, list_pointer ptr2)
{
  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;
  }
}
```

O(m) where m is # of elements in the first list

OPERATIONS FOR CIRCULARLY LINKED LIST

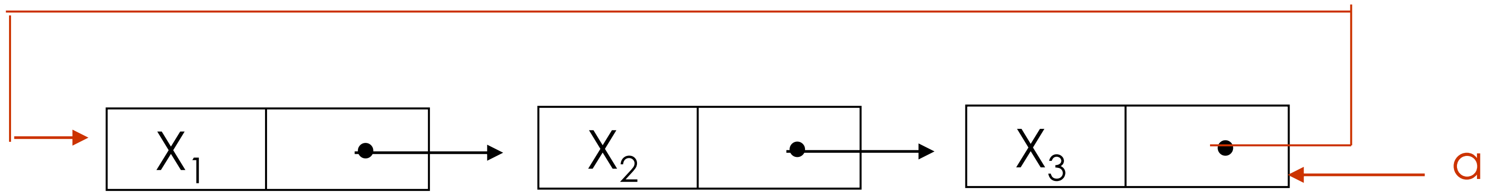
What happens when we insert a node to the front of a circular linked list?



Problem: move down the whole list.

***Figure 4.16:** Example circular list (p.165)*

A possible solution:

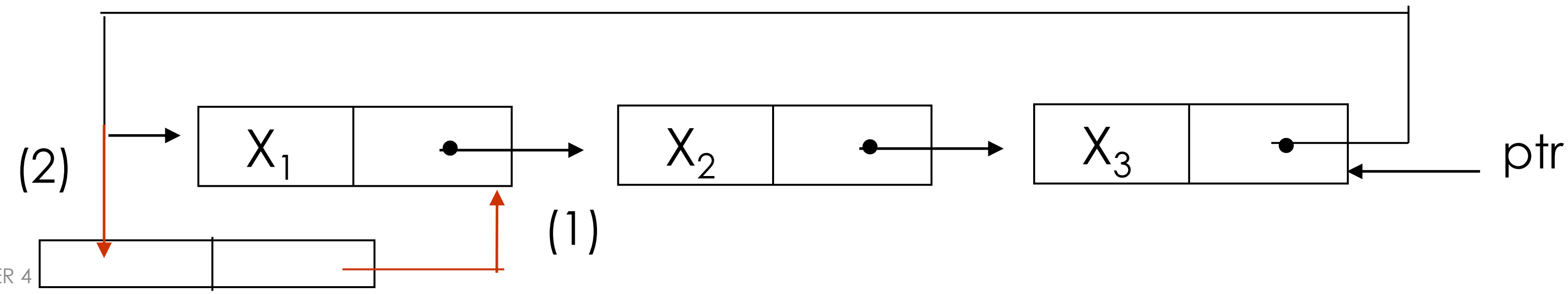


Note a pointer points to the last node.

***Figure 4.17:** Pointing to the last node of a circular list

OPERATIONS FOR CIRCULAR LINKED LISTS

```
void insert_front (list_pointer *ptr, list_pointer
node)
{
    if (IS_EMPTY(*ptr)) {
        *ptr= node;
        node->link = node;
    }
    else {
        node->link = (*ptr)->link;   (1)
        (*ptr)->link = node;         (2)
    }
}
```



Length of Linked List

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

Equivalence Relations

A relation over a set, S, is said to be an *equivalence relation* over S *iff* it is symmertric, reflexive, and transitive over S.

reflexive, x=x

symmetric, if x=y, then y=x

transitive, if x=y and y=z, then x=z

Examples

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

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

A ROUGH ALGORITHM TO FIND EQUIVALENCE CLASSES

```
void equivalenec()
{
     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);
}
```

Phase 1

Phase 2

What kinds of data structures are adopted?

First Refinement

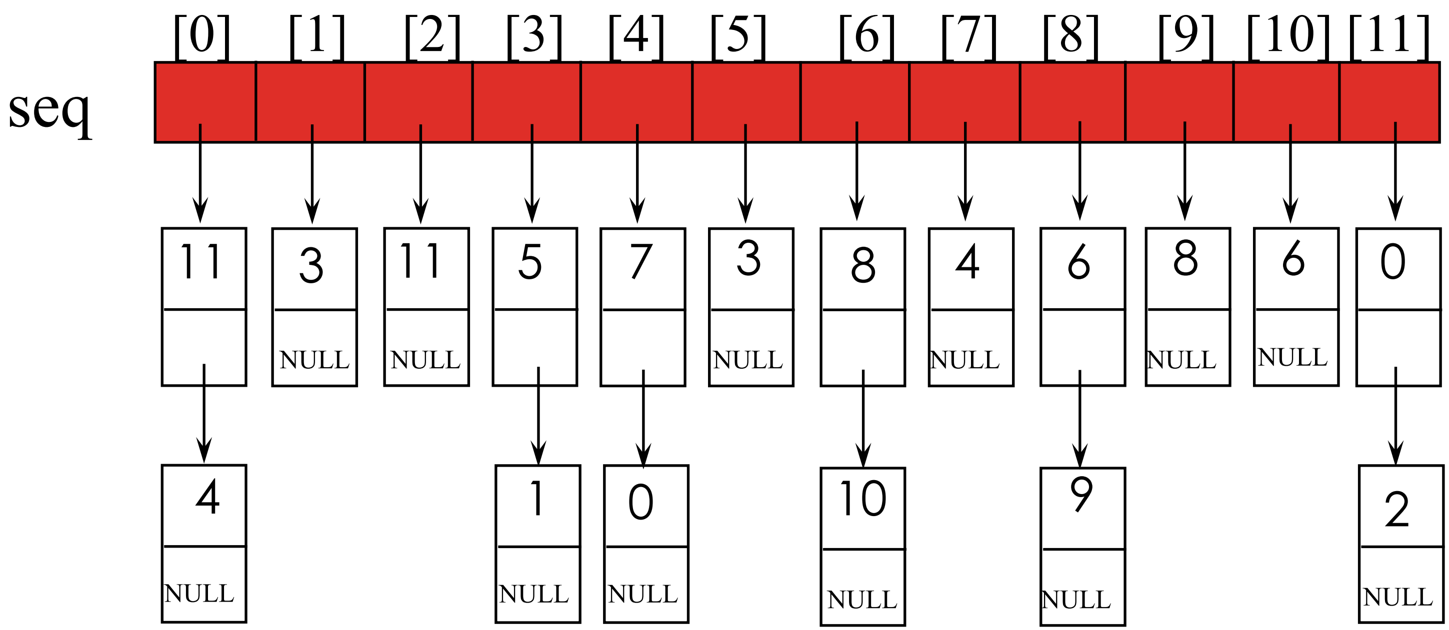
```
#include <stdio.h>
#include <alloc.h>
#define MAX_SIZE 24
#define IS_FULL(ptr)   (!(ptr))
#define FALSE  0
#define TRUE   1
void equivalence()
{
    initialize seq to NULL and out to TRUE
    while (there are more pairs) {
        read the next pair, <i,j>;
        put j on the seq[i] list;
        put i on the seq[j] list;
    }
    for (i=0; i<n; i++)
        if (out[i]) {
            out[i]= FALSE;
            output this equivalence class;
        }
}
```

direct equivalence

Compute indirect equivalence using transitivity

Lists After Pairs are input

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




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

FINAL VERSION FOR FINDING EQUIVALENCE CLASSES

```
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++) {
      out[i]= TRUE;    seq[i]= NULL;
  }
  printf("Enter a pair of numbers (-1 -1 to quit): ");
  scanf("%d%d", &i, &j);
```

Phase 1: input the equivalence pairs:

```
while (i>=0) { Insert x to the top of lists seq[i]
    x = (node_pointer) malloc(sizeof(node));
    if (IS_FULL(x))
      fprintf(stderr, "memory is full\n");
        exit(1);
    } Insert x to the top of lists seq[j]
    x->data= j;  x->link= seq[i];  seq[i]= x;
    if (IS_FULL(x))
      fprintf(stderr, "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;
          x = seq[i];    top = NULL;
          for (;;) {
               while (x) {
                    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;
          }
     }
}
```

push

pop

SPARSE MATRICES

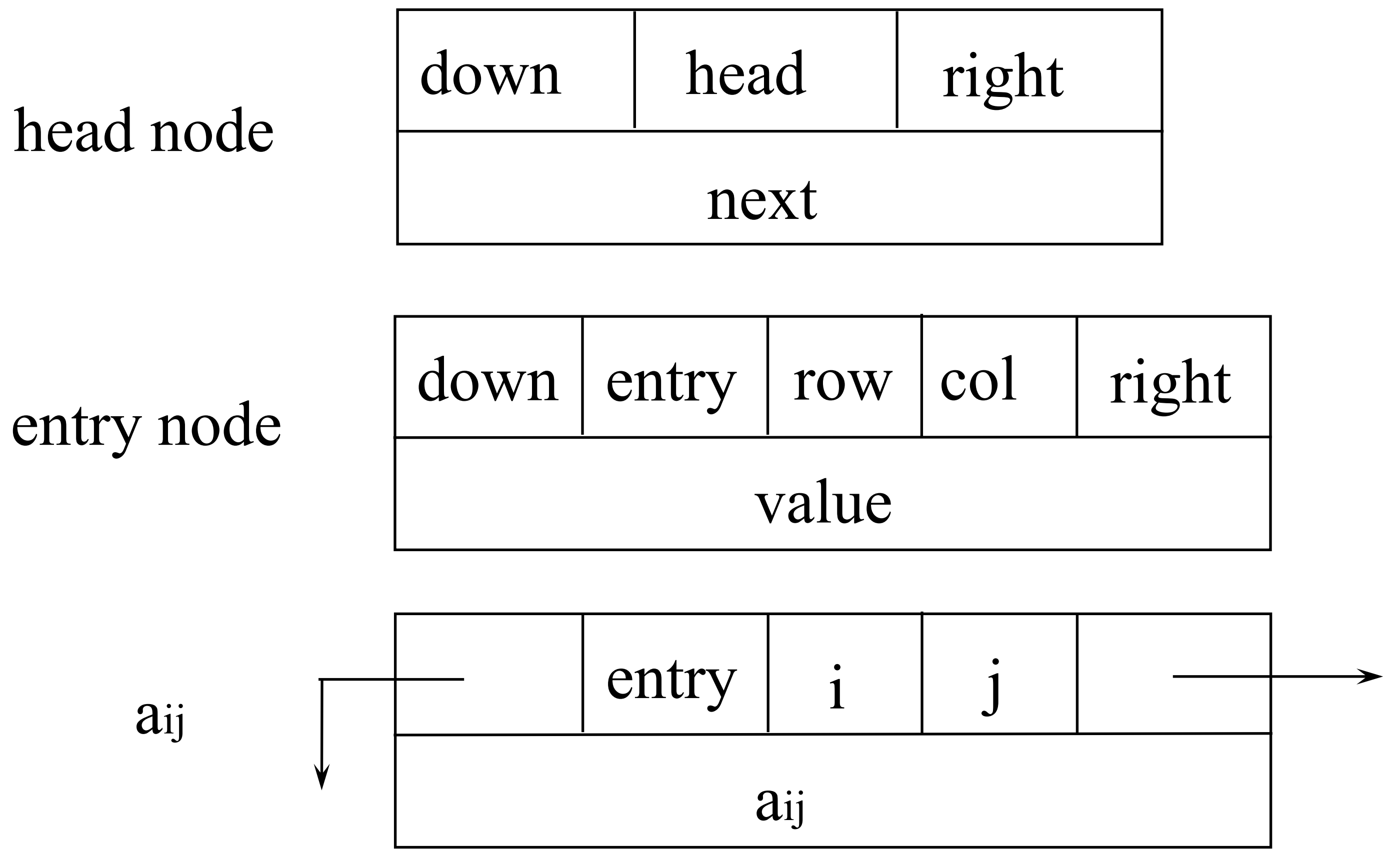
$$\begin{bmatrix} 0 & 0 & 11 & 0 \\ 12 & 0 & 0 & 0 \\ 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & -15 \end{bmatrix}$$

inadequates of sequential schemes
(1) # of nonzero terms will vary after some matrix computation
(2) matrix just represents intermediate results

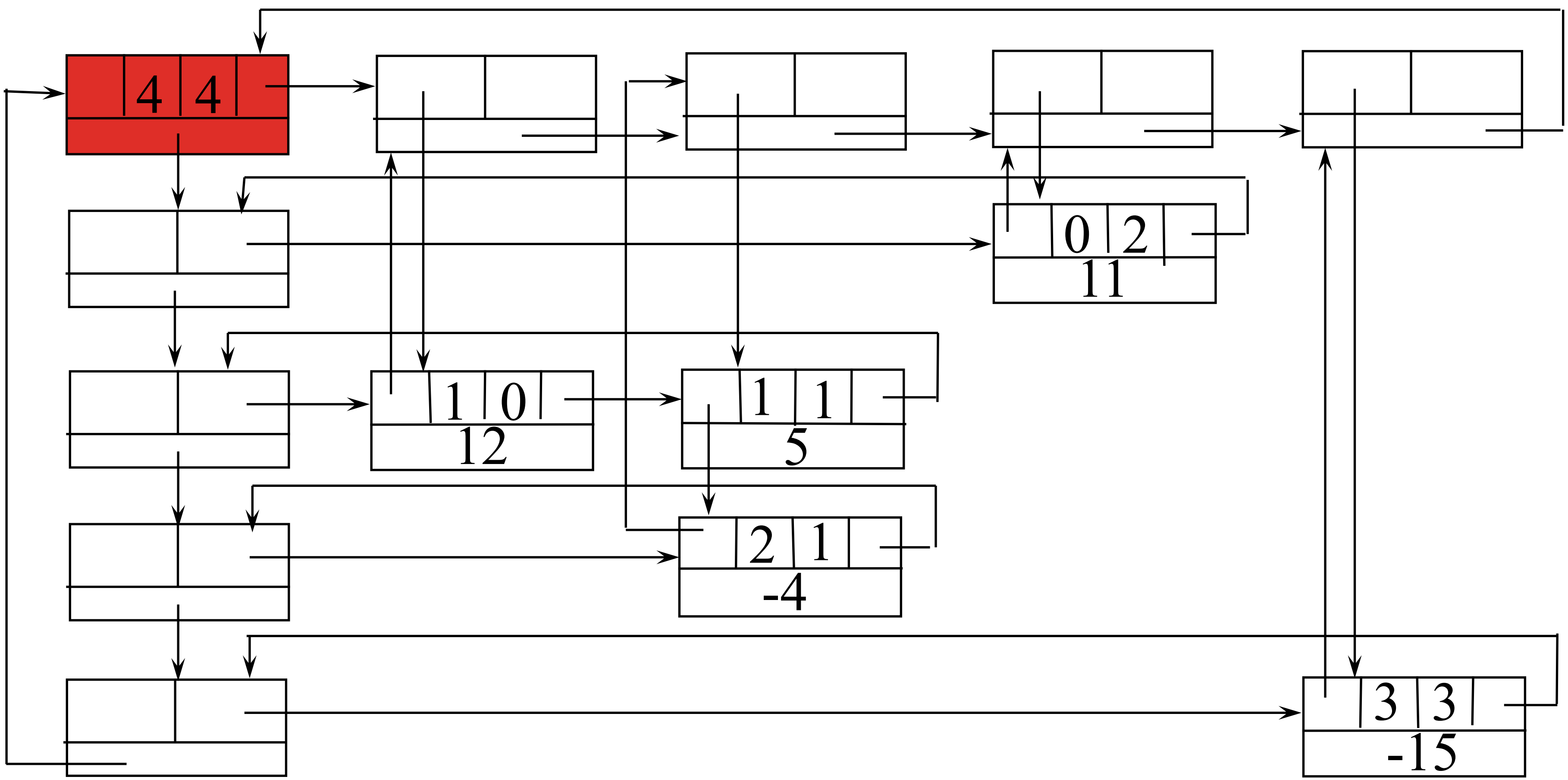
new scheme
Each column (row): a circular linked list with a head node

Revisit Sparse Matrices

of head nodes = max{# of rows, # of columns}



Linked Representation for Matrix



Circular linked list

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

	[0]	[1]	[2]
[0]	4	4	4
[1]	0	2	11
[2]	1	0	12
[3]	2	1	-4
[4]	3	3	-15

***Figure 4.22: Sample input for sparse matrix**

Read in a Matrix

```
matrix_pointer mread(void)
{
/* read in a matrix and set up its linked
 list. An global array hdnode is used */
  int num_rows, num_cols, num_terms;
  int num_heads, i;
  int row, col, value, current_row;
  matrix_pointer temp, last, node;

  printf("Enter the number of rows, columns
         and number of nonzero terms: ");
```

```
scanf("%d%d%d", &num_rows, &num_cols,
      &num_terms);
 num_heads =
 (num_cols>num_rows)? num_cols : num_rows;
 /* set up head node for the list of head
    nodes */
 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 { /* initialize the head nodes */
   for (i=0; i<num_heads; i++) {
     term= new_node();
     hdnode[i] = temp;
     hdnode[i]->tag = head;
     hdnode[i]->right = temp;
     hdnode[i]->u.next = temp;
   }
```

O(max(n,m))

```
current_row= 0;    last= hdnode[0];
for (i=0; i<num_terms; i++) {
  printf(“Enter row, column and value:”);
  scanf(“%d%d%d”, &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;/*link to row list */
  last= temp;
  /* link to column list */
  hdnode[col]->u.next->down = temp;
  hdnode[col]=>u.next = temp;
}
```

Store the last node of column in next field

```
    /*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;
}
```

O(max{#_rows, #_cols}+#_terms)

Write out a Matrix

```
void mwrite(matrix_pointer node)
{ /* print out the matrix in row major form */
  int i;
  matrix_pointer temp, head = node->right;
  printf("\n num_rows = %d, num_cols= %d\n",
         node->u.entry.row,node->u.entry.col);
  printf("The matrix by row, column, and
         value:\n\n");      O(#_rows+#_terms)
  for (i=0; i<node->u.entry.row; i++) {
    for (temp=head->right;temp!=head;temp=temp->right)
      printf("%5d%5d%5d\n", temp->u.entry.row,
           temp->u.entry.col, temp->u.entry.value);
    head= head->u.next; /* next row */
  }
}
```

Free the entry and head nodes by row.

Erase a Matrix

```
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);
  }
  y = head;
  while (y!=*node) {
    x = y;  y = y->u.next;  free(x);
  }
  free(*node);  *node = NULL;
}
```

O(#_rows+#_cols+#_terms)

Doubly Linked List

Move in forward and backward direction.

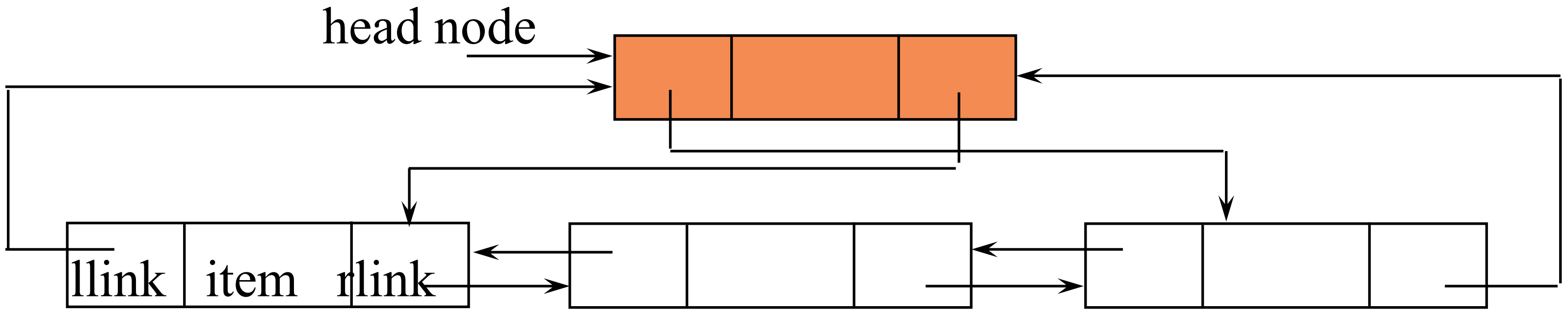
Singly linked list (in one direction only)
How to get the preceding node during deletion or insertion?
Using 2 pointers

Node in doubly linked list
left link field (llink)
data field (item)
right link field (rlink)

Doubly Linked Lists

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

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



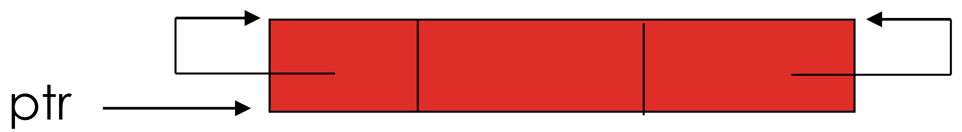


***Figure 4.24:**Empty doubly linked circular list with head node (p.1

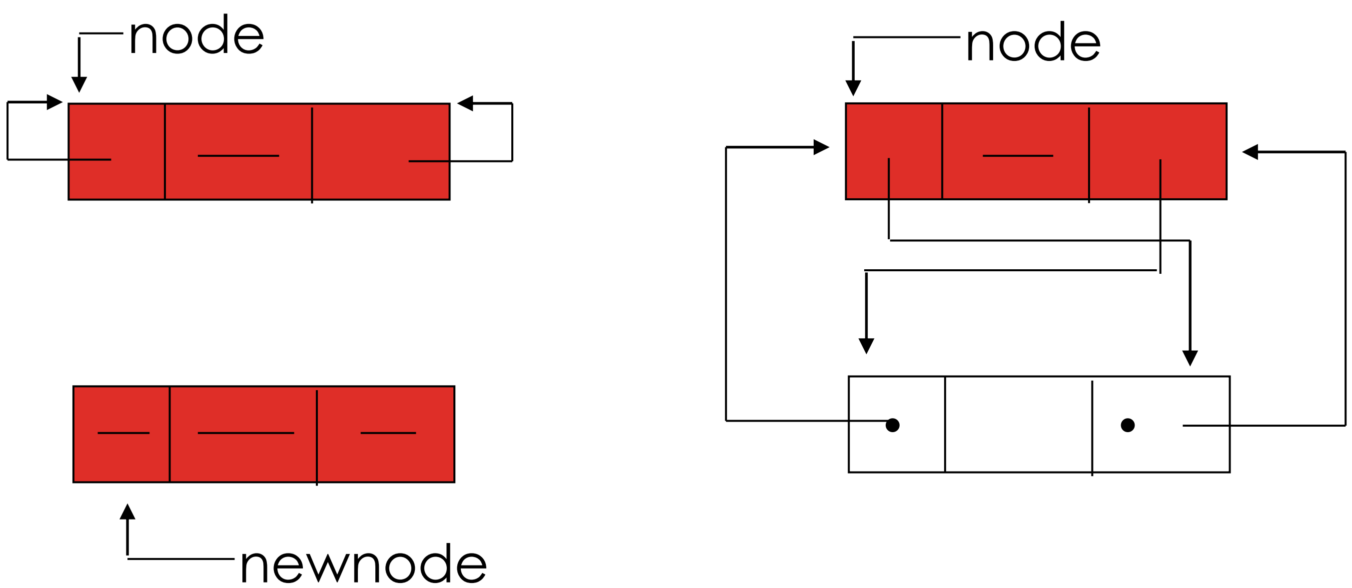
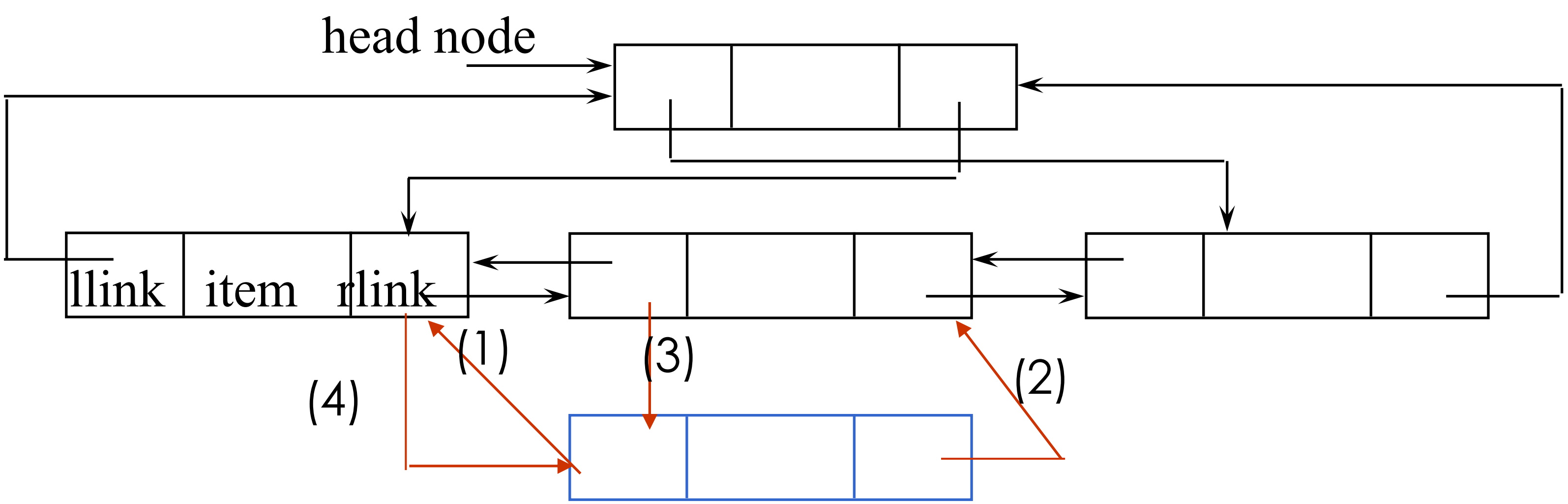


***Figure 4.25:** Insertion into an empty doubly linked circular list (p.18

Insert

```
void dinsert(node_pointer node, node_pointer newnode)
{
      (1) newnode->llink = node;
      (2) newnode->rlink = node->rlink;
      (3) node->rlink->llink = newnode;
      (4) node->rlink = newnode;
}
```



Delete

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