LISTS

Introduction

- Array
  successive items located at fixed distance apart
- disadvantage
  - data movements during insertion and deletion
  - waste space in storing n ordered lists of varying size
- possible solution
  - linked list

Pointer Review (1)

Pointer Can Be Dangerous

int i, *pi;

	1000		2000
i	?	pi	?

pi = &i;

	1000		2000
i *pi	?	pi	1000

i = 10 or *pi = 10

	1000		2000
i *pi	10	pi	1000

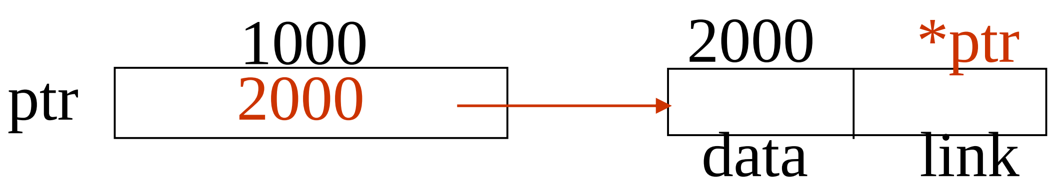
- Set to NULL
- Explicit typecasts

Pointer Review (2)

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

ptr->data□(*ptr).data

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



4.2 SINGLY LINKED LISTS

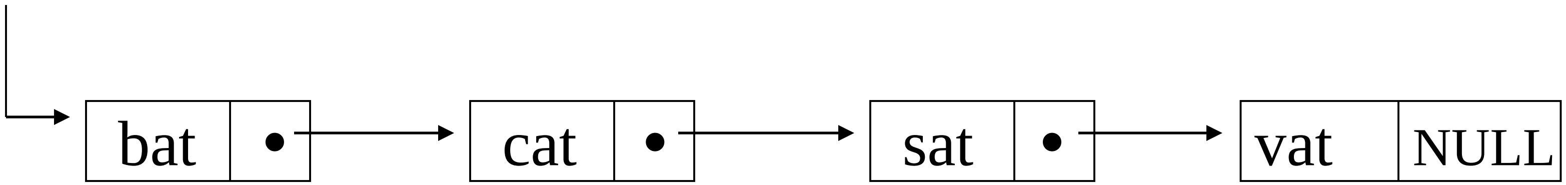


***Figure 4.1:** Usual way to draw a linked list

Insertion

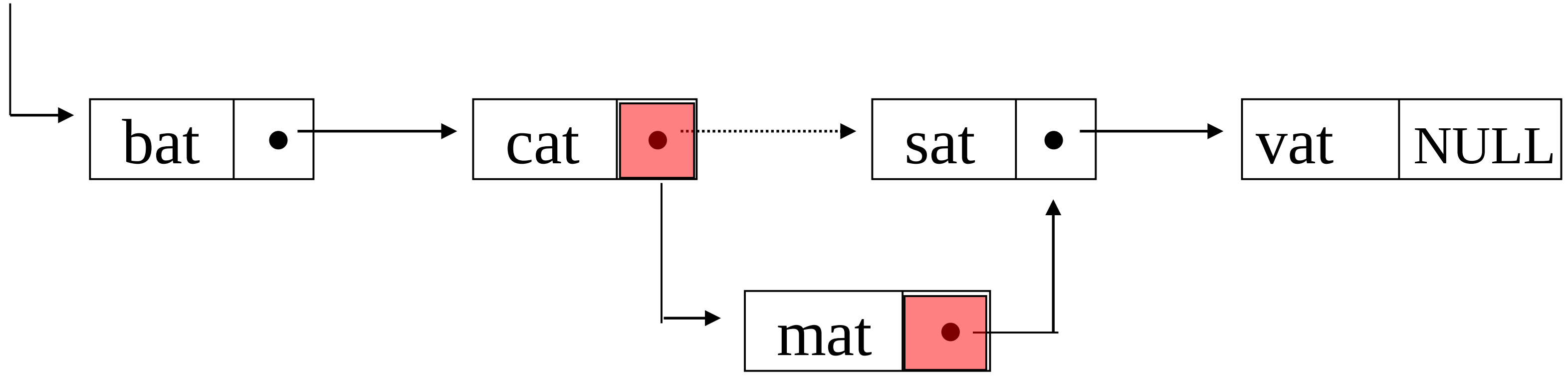


***Figure 4.2:** Insert mat after cat

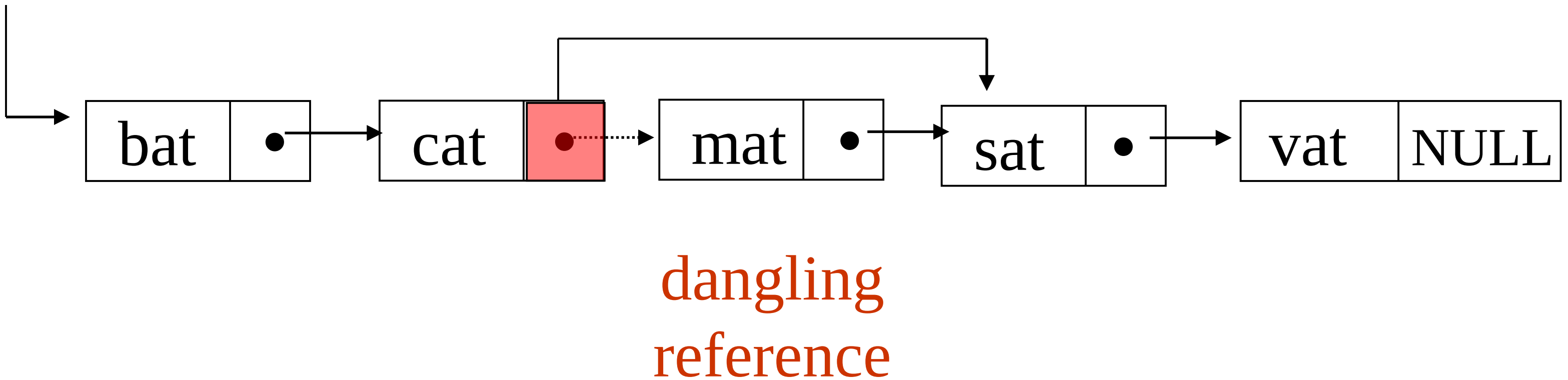


***Figure 4.3:** Delete *mat* from list

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

```
list_pointer ptr =NULL;
```

Testing

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

Allocation

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

strcpy(ptr -> data, “bat”);
ptr -> link = NULL;

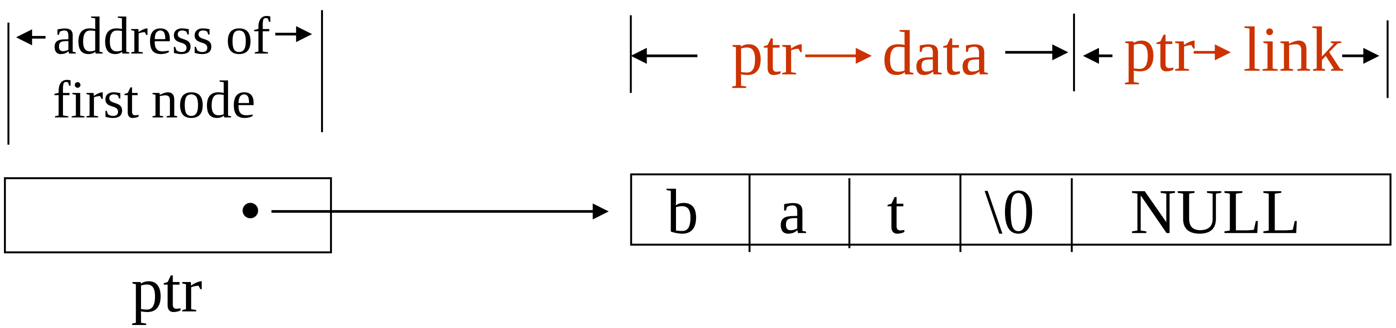
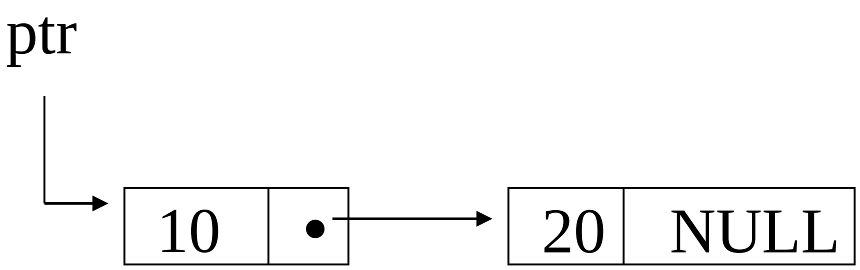


***Figure 4.4:**Referencing the fields of a node(p.142)

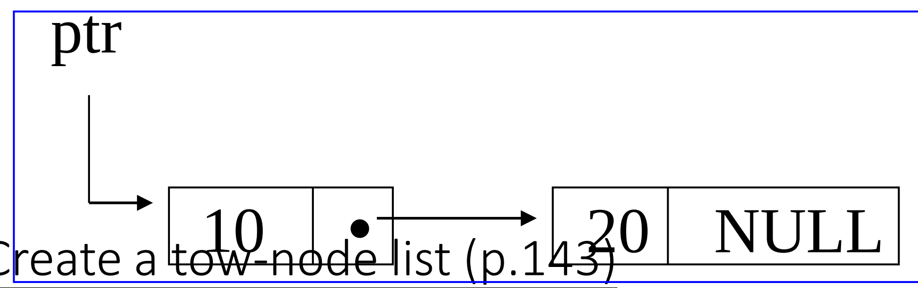
Example: create a two-node list




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

Example 4.2: (p.142)

```
list_pointer create2( )
{
/* create a linked list with two nodes */
   list_pointer first, second;

   first = (list_pointer) malloc(sizeof(list_node)); // new list_node;
   second = ( list_pointer) malloc(sizeof(list_node));//new list_node
   second -> link = NULL;
   second -> data = 20;
   first -> data = 10;
   first ->link = second;
   return first;
}
```



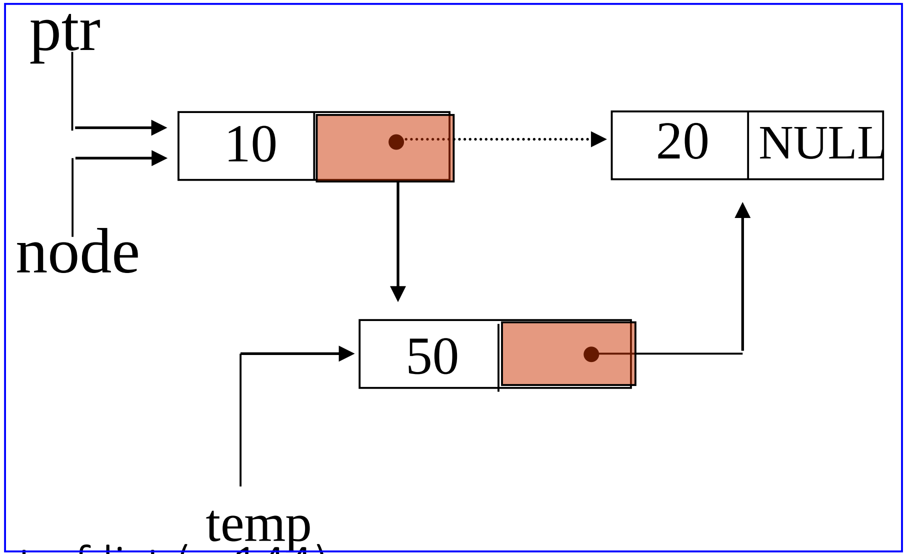
***Program 4.2:**Create a tow-node list (p.143)

List Insertion:

Insert a node after a specific node

```
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)){
       cout<<"The memory is full\n";
       exit (1);
    }
```

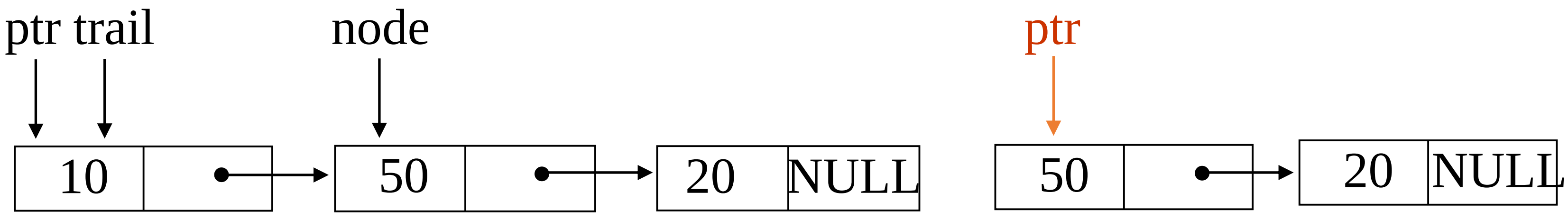
```
    temp->data = 50;
    if (*ptr) {  noempty list
        temp->link =node ->link;
        node->link = temp;
   }
   else {    empty list
       temp->link = NULL;
       *ptr =temp;
    }
}
```



***Program 4.3:**Simple insert into front of list (p.144)

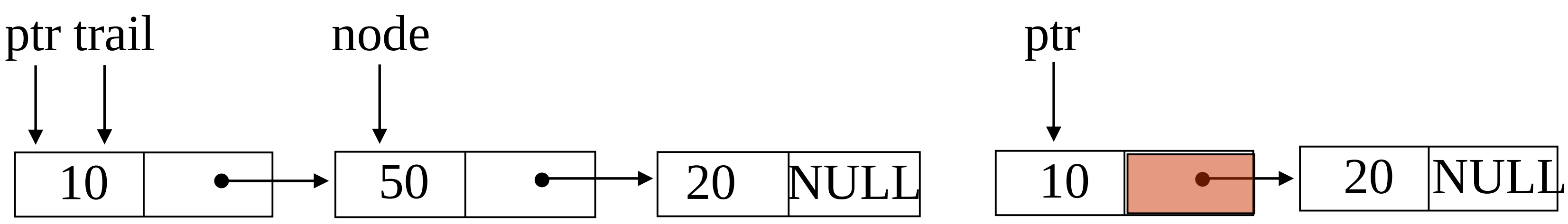
List Deletion

Delete the first node.

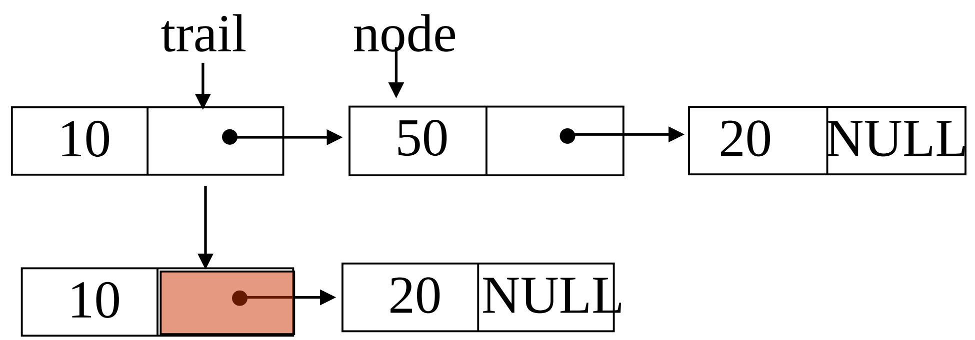


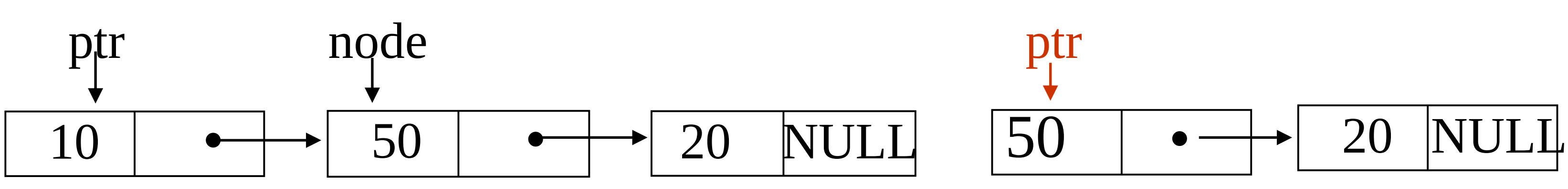
(a) before deletion (b)after deletion

Delete node other than the first node.



```
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);
}
```





Print out a list (traverse a list)

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

***Program 4.5:** Printing a list (p.146)

4.3 DYNAMICALLY LINKED STACKS AND QUEUES

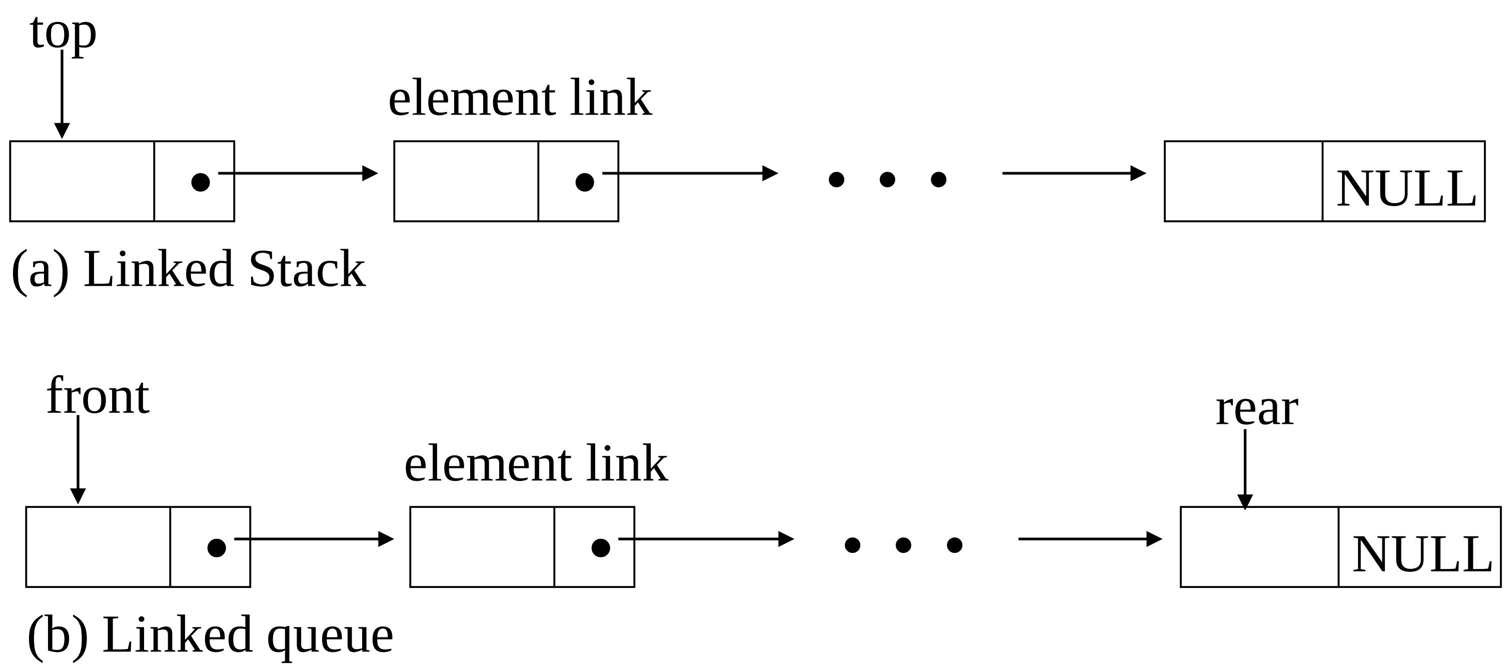


***Figure 4.10:** Linked Stack and queue (p.147)

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 (p.149)

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 (p.149)

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

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

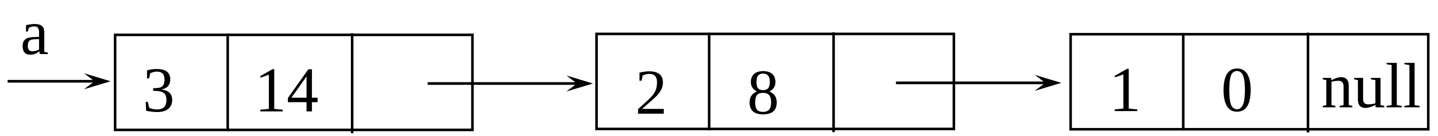
Representation

```
typedef struct poly_node *poly_pointer;

typedef struct poly_node {
      int coef;
      int expon;
      poly_pointer link;
  };

poly_pointer a, b, c;
```

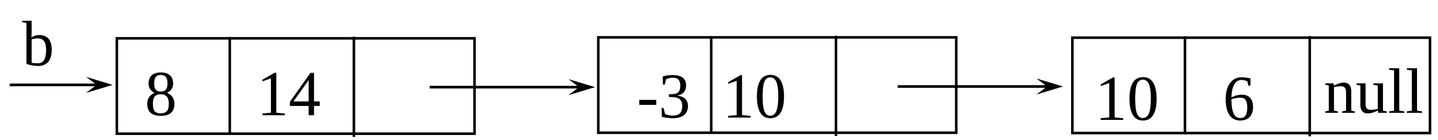
coef	expon	link

Examples

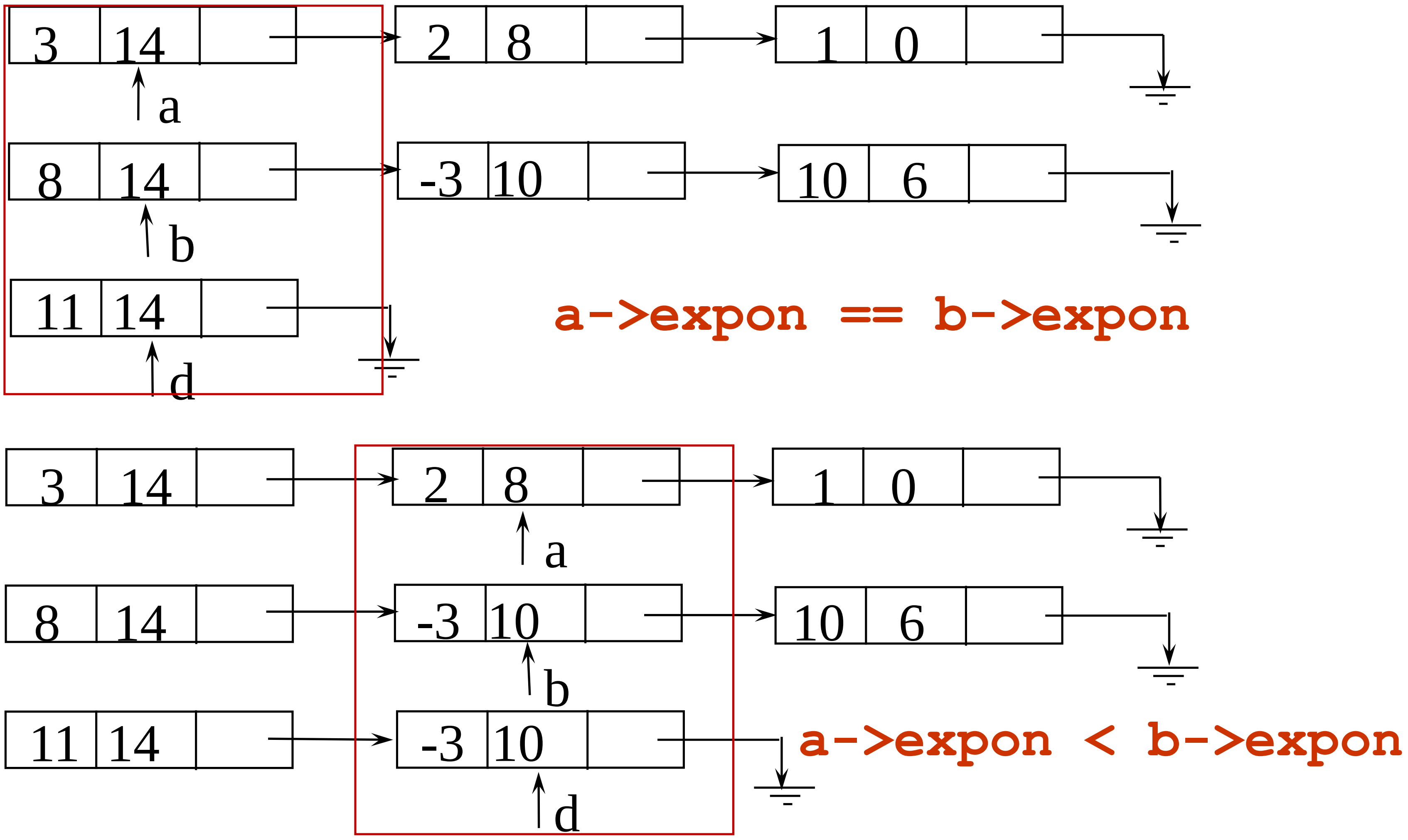
$$a = 3x^{14} + 2x^{8} + 1$$



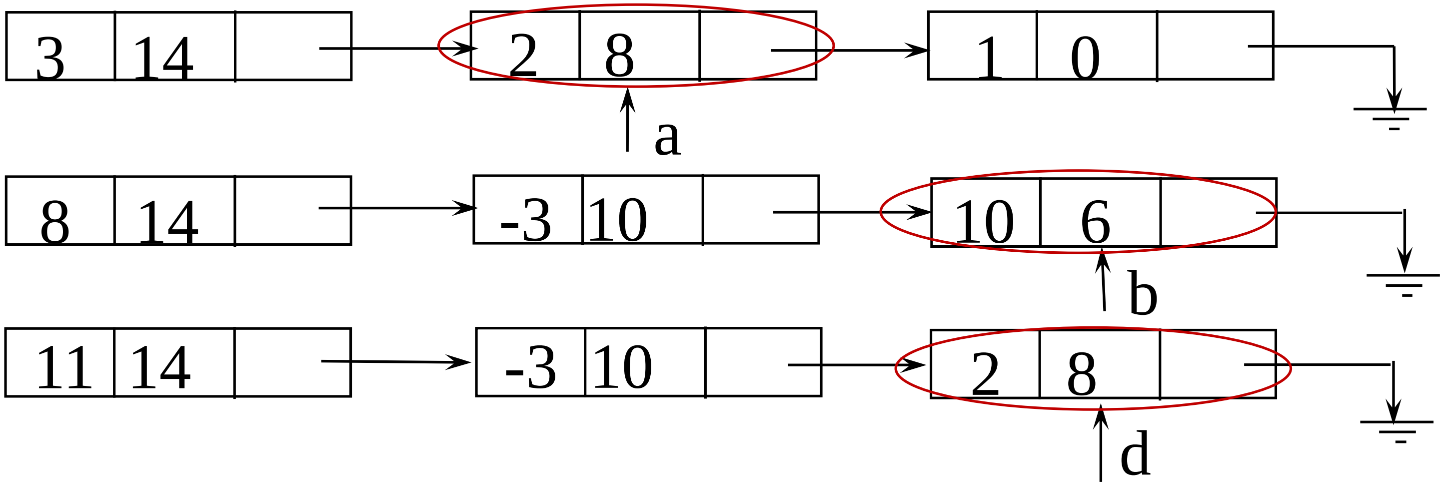
$$b = 8x^{14} - 3x^{10} + 10x^{6}$$



Adding Polynomials



Adding Polynomials (*Continued*)




```
a->expon > b->expon
```

Alogrithm for Adding Polynomials

```
poly_pointer padd(poly_pointer a,
                  poly_pointer b)
{
}

void attach(int coefficient, int exponent,
            poly_pointer *ptr)
{

}
```

Attach a Term

```
void attach(int 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.
*/


}
```

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));


}
```

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));

    temp->coef = coefficient;
    temp->expon = exponent;

}
```

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));

    temp->coef = coefficient;
    temp->expon = exponent;


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

Alogrithm for Adding Polynomials

```
poly_pointer padd(poly_pointer a, poly_pointer b)
{
    poly_pointer c, rear, temp;
    int sum;

     //create a dummy node
    rear =(poly_pointer)malloc(sizeof(poly_node));
    c= rear;
```

```
while (a && b) {
        switch (COMPARE(a->expon, b->expon)) {
            …
            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;
            …
        }
```

```
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;

    …
}
```

```
    /* 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;
}
```

Erasing Polynomials

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

```
…..
polypointer a, b, d, e;
.
.
.
a=readPoly();
/*read b and d polynomials */
….

temp=pmult(a,b);
e=padd(temp, d);
printPoly(e);
..
```

Erasing Polynomials

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

```
…..
polypointer a, b, d, e;
.
.
.
a=readPoly();
/*read b and d polynomials */
….

temp=pmult(a,b);
e=padd(temp, d);
printPoly(e);
..
/* Erase temp */
```

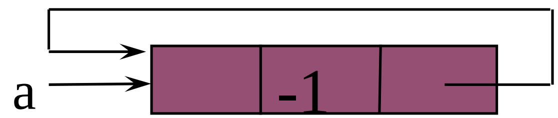
Erasing Polynomials

```
void erase(poly_pointer *ptr)
{
    poly_pointer temp;

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

       free(temp);
     }
}
```

Head Node

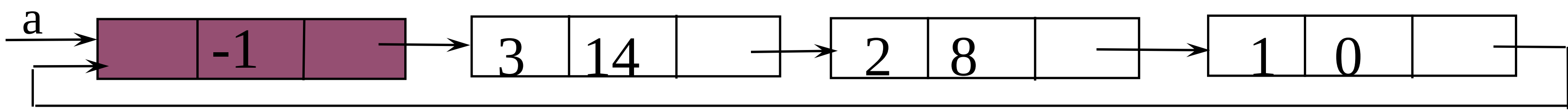
Represent polynomial as circular list.

(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, c, lastC;
  int sum, done = FALSE;

  startA = a;
  a = a->link;
  b = b->link;

  c = get_node();
  c->expon = -1;    lastC = c;
```

Set expon field of head node to -1.

Another Padd *(Continued)*

```
do {
   switch (COMPARE(a->expon, b->expon)) {
     case -1: attach(b->coef, b->expon, &lastC);
              b = b->link;
              break;

     case 0: 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: attach(a->coef,a->expon,&lastC);
             a = a->link;
   }
 } while (!done);
```

Another Padd (*Continued*)

```
  lastC->link = c;
  return c;
}
```

Link last node to first

Additional List Operations

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

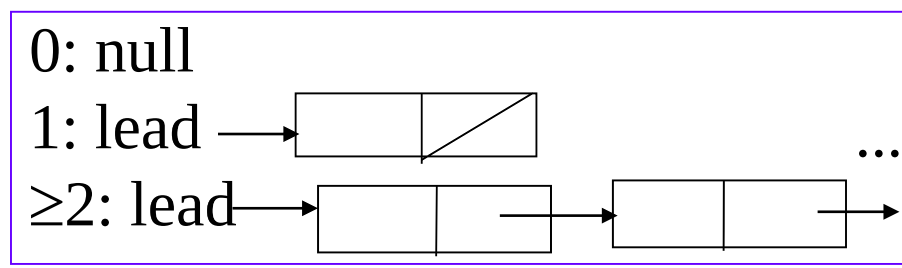
Operations For Chains

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;
}
```



Concatenate Two Lists

```
list_pointer concatenate(list_pointer
             ptr1, list_pointer ptr2)
{
//Note: One or both the pointers may be NULL
 ………



}
```

Concatenate Two Lists

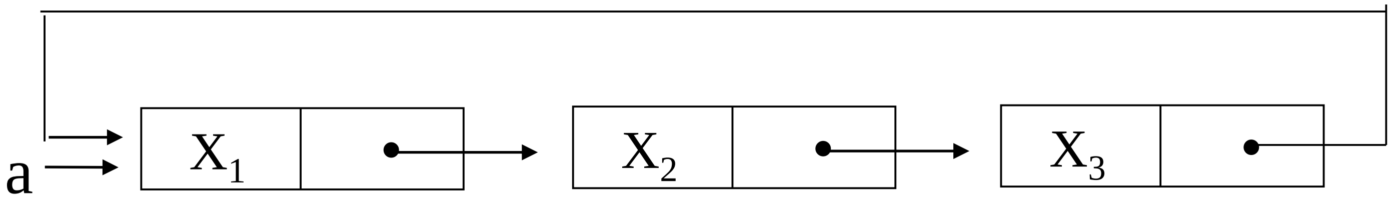
```
list_pointer concatenate(list_pointer
             ptr1, list_pointer ptr2)
{
  list_pointer temp;
  if (IS_EMPTY(ptr1)) return ptr2;
  else {
     .........
    }
    return ptr1;
  }
}
```

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;
  }
}
```

4.5.2 Operations For Circularly Linked List

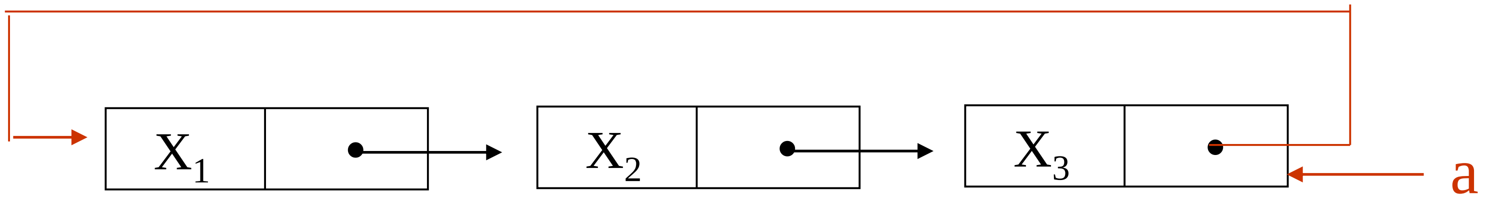
What happens when we insert a node to the front of a circular linked list?



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 (p.165)

Operations for Circular Linked Lists

```
insertFront()
insertRear()
length()
```

Operations for Circular Linked Lists

insertFront()

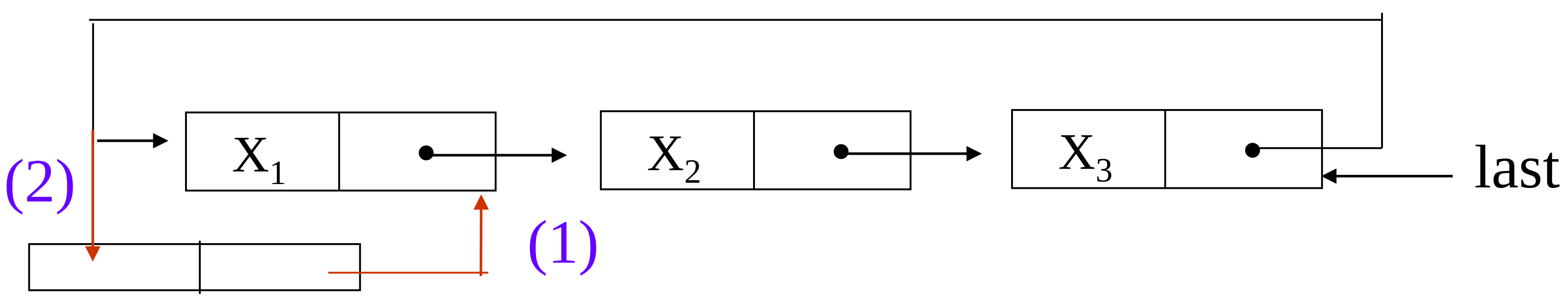
insertRear()

length()

```
void insertFront (list_pointer *last,
list_pointer node)
{
  //Note: *last may be null

}
```

Operations for Circular Linked Lists

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



Operations for Circular Linked Lists

`insertFront()`
`insertRear()`
`length()`

```
void insertRear (list_pointer *last,
list_pointer node)
{
  //Note: *last may be null

}
```

Operations for Circular Linked Lists

insertFront()

insertRear()

length()

```
void insertRear (list_pointer *last,
list_pointer node)
{
  //Note: *last may be null
// same as insertFront except
   *last = node; //in else
}
```

Operations for Circular Linked Lists

```
insertFront()
insertRear()
```
`length()`

```
void length(list_pointer last)
{
  //Note: last may be null

}
```

Length of Linked List

```
int length(list_pointer last)
{   list_pointer temp;
    int count = 0;

    if (last) {
        temp = last;
        do {
            count++;
            temp = temp->link;
        } while (temp!=last);
    }

    return count;
}
```