Chapter 4
Lists

2024 Spring

Ri Yu

Ajou University

Contents

Singly Linked Lists and Chains

Representing Chains in C

Polynomials

Doubly Linked Lists

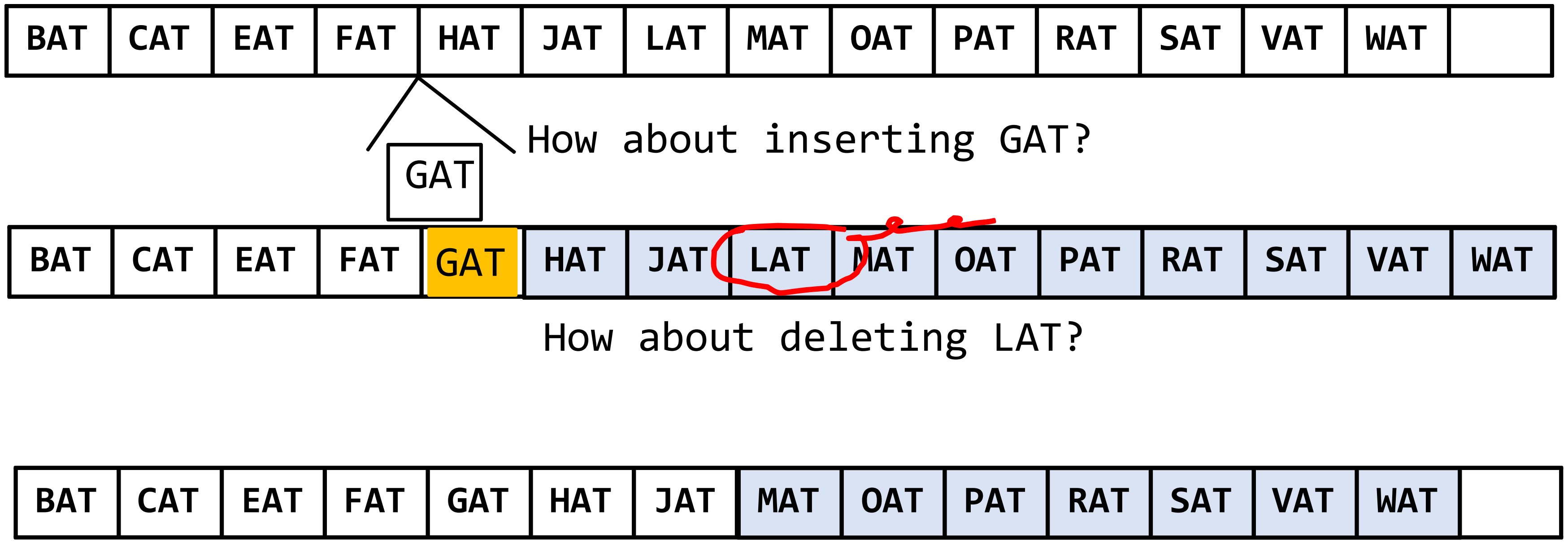
Singly Linked Lists and Chains

- Sequential representation
  - Successive items of a list are located a fixed distance apart
- Linked representation
  - Items may be placed anywhere in memory
  - To access list elements
    - store the address or location of the next element in that list

Why Linked List?

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

If sequential mapping is used,



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

GAT

How about inserting GAT?

BAT	CAT	EAT	FAT	GAT	HAT	JAT	LAT	MAT	OAT	PAT	RAT	SAT	VAT	WAT

How about deleting LAT?

BAT	CAT	EAT	FAT	GAT	HAT	JAT	MAT	OAT	PAT	RAT	SAT	VAT	WAT	

Insertion and deletion of arbitrary elements become expensive!

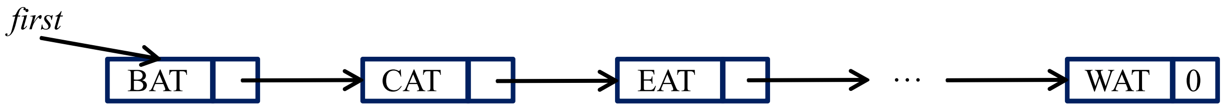
Singly Linked Lists and Chains

❖ *data*[*i*] and *link*[*i*] pair comprise a node

- *Data :* Elements are no longer in sequential order
- *Link :* The values are pointers to elements in the *data* array

- The list starts at *data*[8]=BAT
  - *first*=8
  - *link*[8]=3, which means it points to *data*[3], which contains CAT
- When we have come to the end of the ordered list
  - *link* equals zero

	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

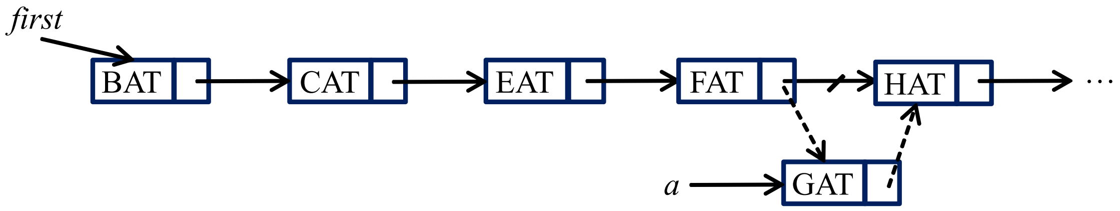


Usual way to draw a linked list

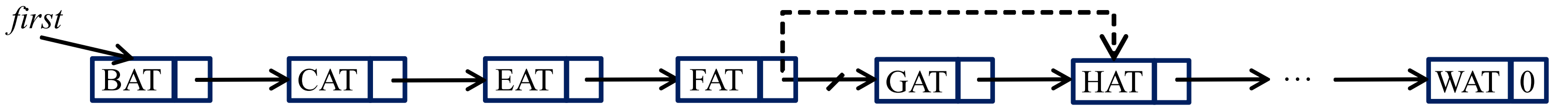
Singly Linked Lists and Chains

	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

- ✋■⬧♏❒♦♓□■
  - Inserting GAT into the list
    - Do not have to move any elements



- Deletion
  - Deleting GAT from the list
    - Even though the link of GAT still contains a pointer to HAT, GAT is no longer in the list as it cannot be reached by starting at the first element of the list



Representing Chains in C

- A chain is a linked list in which each node represents one element.

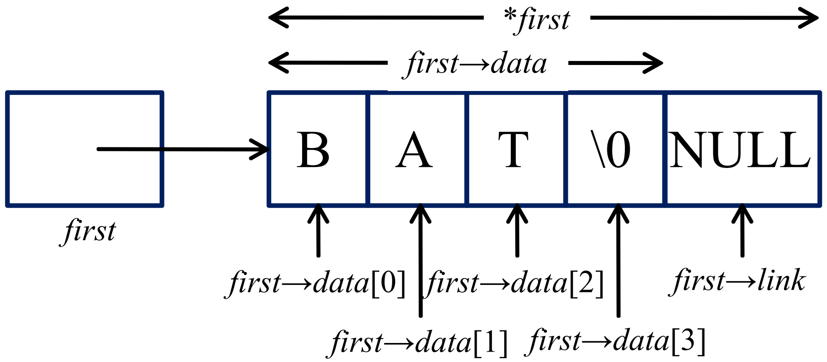
  There is a link or pointer from one element to the next.

```
typedef struct listNode *listPointer;
typedef struct listNode {
                 char data[4];
                 listPointer        link;   /* self-referential structure */
};
```

Representing Chains in C

❖ Example [*List of words*]

- Create a new empty list
  - listPointer first = NULL;   /* contains the address of the start of the list */
- Macro to test for an empty list
  - #define IS_EMPTY(first)   (!(first))
- Create a new node
  - MALLOC(first, sizeof(*first));
- Place the word BAT into the list
  - strcpy(first→data, "BAT");
  - first→link = NULL;



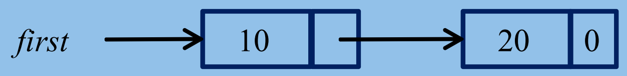
→ :Structure member operator
first →data = (*first).data

Representing Chains in C

- Example [*Two-node linked list*]

```
typedef struct listNode *listPointer;
typedef struct listNode {
                 int      data;
                 listPointer     link;    /* self-referential structure */
};
```

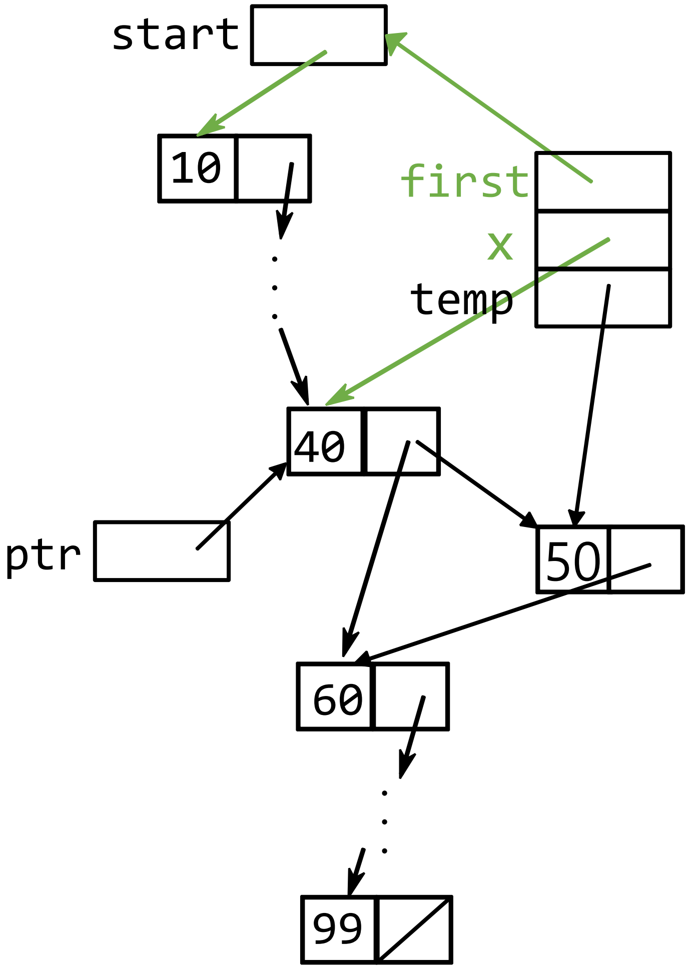
```
listPointer create2()
{       /* create a linked list with two nodes */
        listPointer first, second;
        MALLOC( first, sizeof(*first) );
        MALLOC( second, sizeof(*second) );
        second→link = NULL;
        second→data = 20;
        first→data=10;
        first→link = second;
        return first;
}
```



List Insertion

```
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) {       /* if (*first != NULL), if the chain is not empty */
        temp->link = x->link;
        x->link = temp;
   }
   else {                  /* if (*first == NULL), if the chain is empty */
        temp->link = NULL;
        *first = temp;
   }
}
```

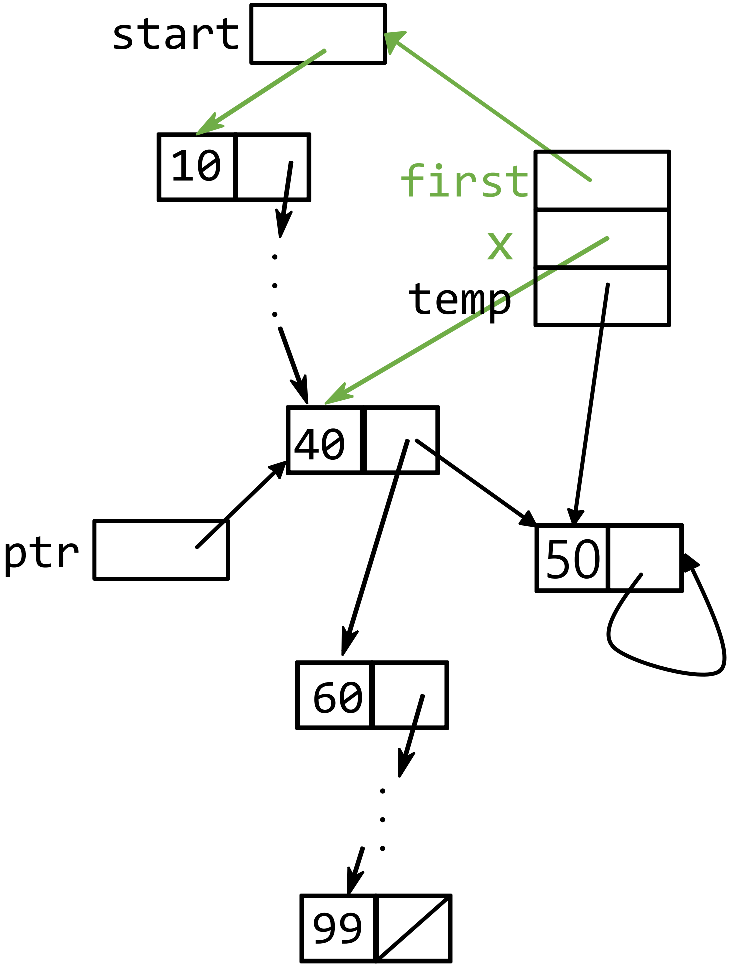
List Insertion Example (1)



```
listPointer start, ptr;
 /* codes for building a list */
 insert(&start,ptr);
```

```
void insert(listPointer *first, listPointer x)
{
    listPointer temp;
    MALLOC(temp, sizeof(*temp));
    temp->data = 50;
    if(*first) {
         temp->link = x->link;
         x->link = temp;
    }
    else {
         /* … */
    }
}
```

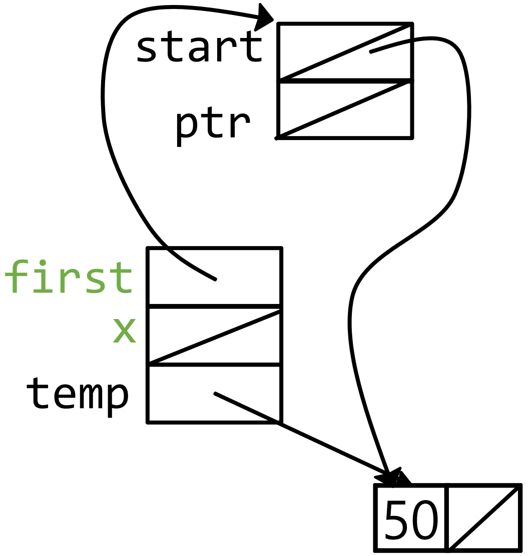
List Insertion Example (1)



```
  listPointer start, ptr;
   /* codes for building a list */
   insert(&start,ptr);
```

```
void insert(listPointer *first, listPointer x)
{
    listPointer temp;
    MALLOC(temp, sizeof(*temp));
    temp->data = 50;
    if(*first) {
        x->link=temp;    What if??
        temp->link=x->link;
    }
    else {
            /* … */
    }
}
```

List Insertion Example (2)



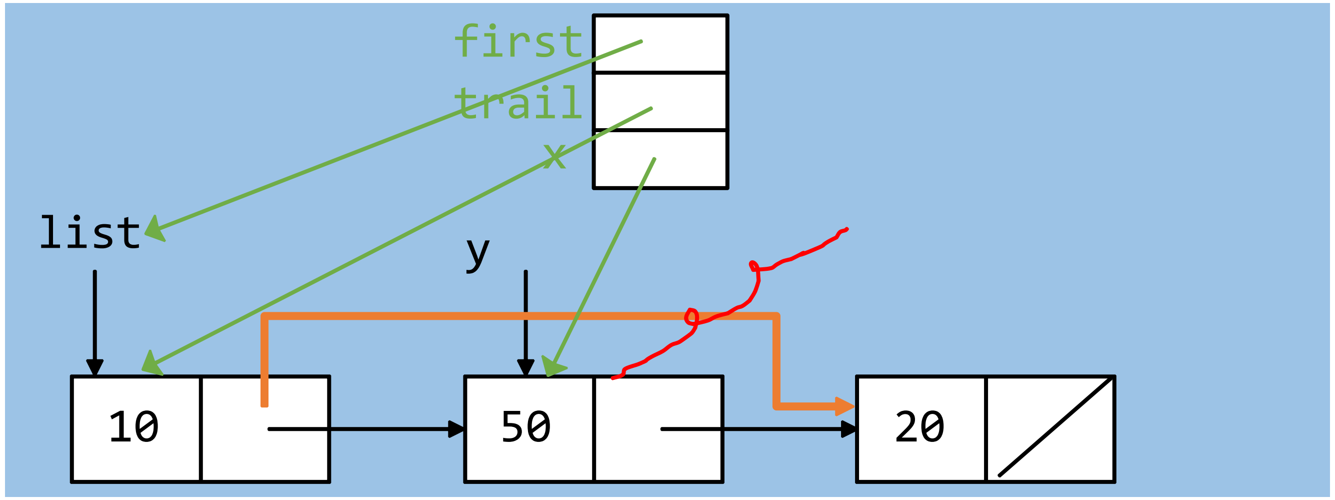
```
  listPointer start, ptr;
  start=ptr=NULL;
   insert(&start,ptr);
```

```
void insert(listPointer *first, listPointer x)
{
   listPointer temp;
   MALLOC(temp, sizeof(*temp));
   temp->data = 50;
   if(*first) {/* … */
   } else {
       temp->link = NULL;
       *first = temp;
   }
}
```

List Delete (1)

```
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);
}
```

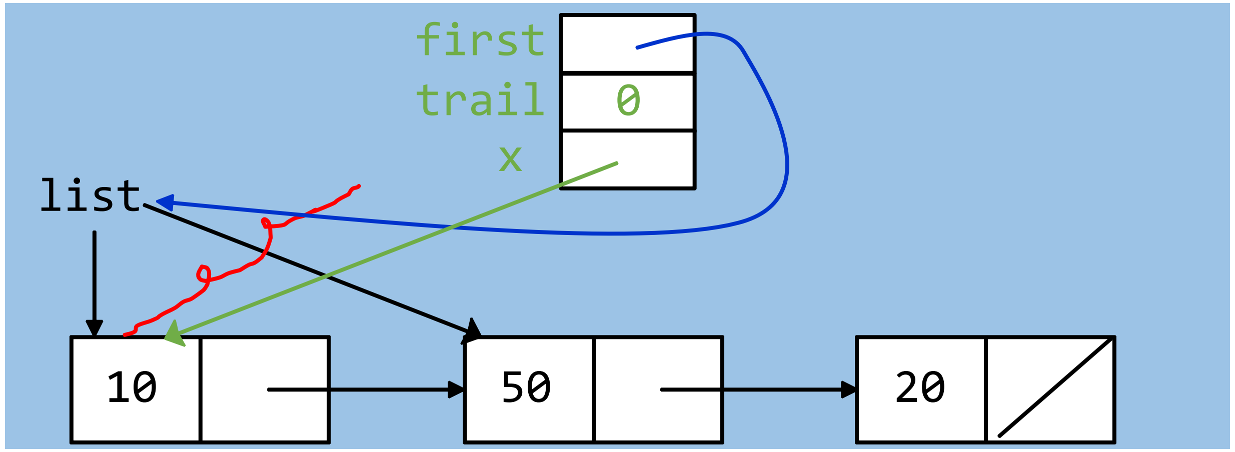
```
delete(&list, list, y);
```



List Delete (2)

```
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);
}
```

```
delete(&list, NULL, list);
```



Polynomials

- Representing polynomials using linked lists

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

where the a_i are nonzero coefficients and the e_i are nonnegative integer exponents such that $e_{m\text{-}1} > e_{m\text{-}2} > \ldots > e_1 > e_0 \geq 0$

- We will represent each term as a node containing coefficient and exponent fields, as well as a pointer to the next term

Representation of Polynomial

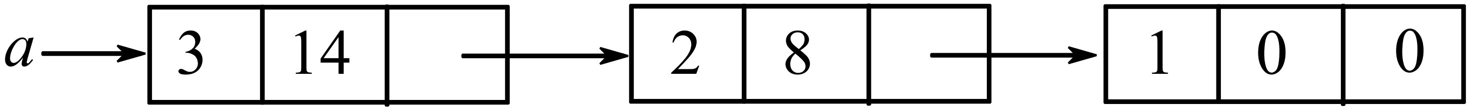
- Declaration of polynomial terms

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

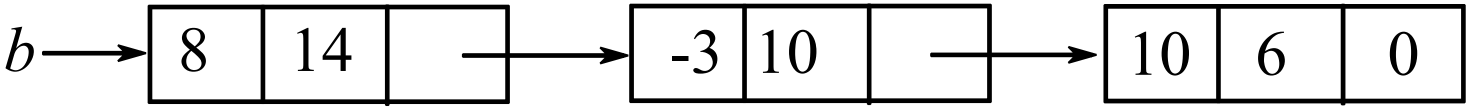
coef	expon	link

Polynomials

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



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



Add Two Polynomials (1)

```
polyPointer padd (polyPointer a, polyPointer b){
    /* return a polynomial which is the sum of a and 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 > a->expon */
                attach(a->coef,a->expon,&rear);
                a = a->link;}
    }
```

Add Two Polynomials (2)

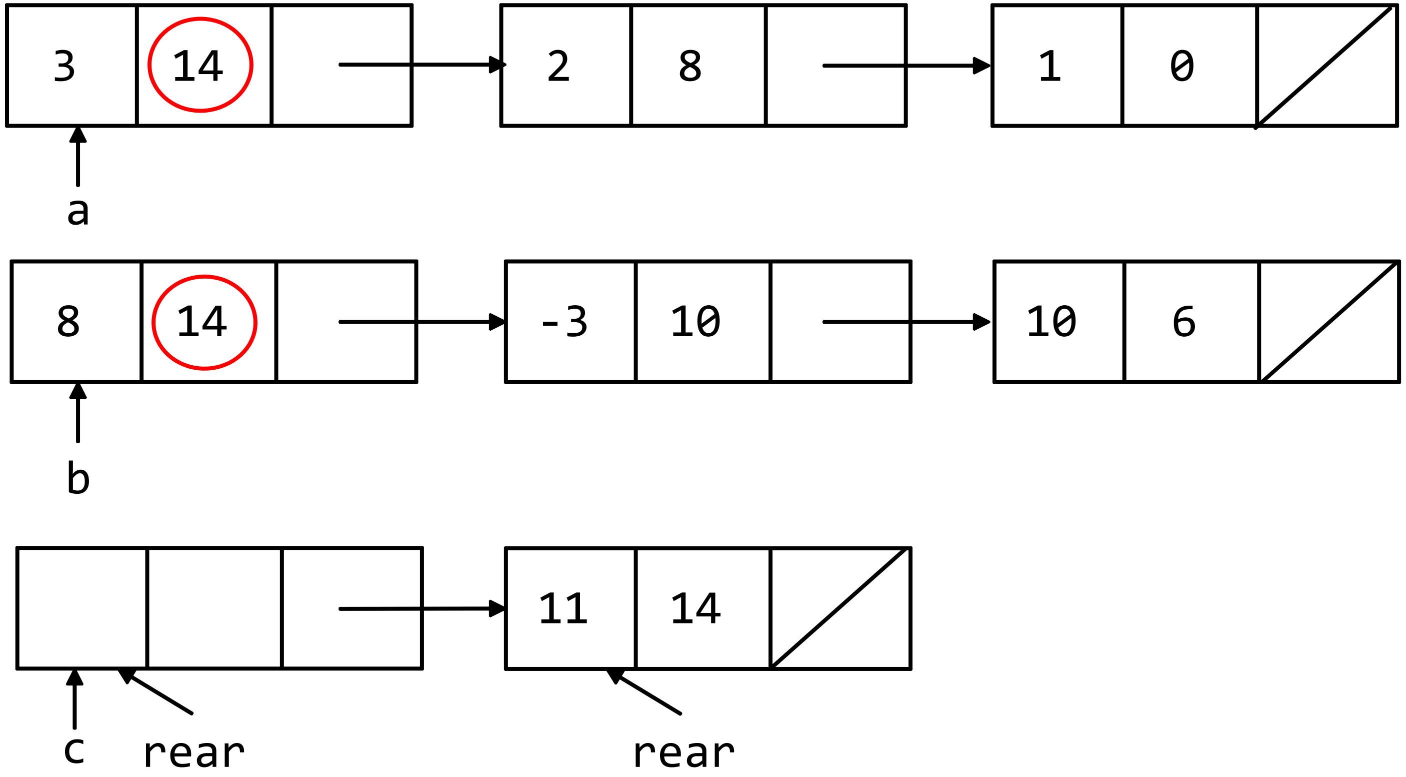
```
    /* copy the rest of list a and list b */
    for (; a; a = a->link) attach(a->coef,a->expon,&rear);
    for (; b; b = b->link) attach(b->coef,b->expon,&rear);
    rear->link = NULL;

    /* delete the useless initial node */
    temp = c;    c = c->link;    free(temp);
    return c;
}
```

Add Two Polynomials (3)

```
void attach(float coefficient, int exponent, polyPointer *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 */

      polyPointer temp;
      MALLOC( temp, sizeof(*temp) );
      temp→coef = coefficient;
      temp→expon = exponent;
      (*ptr)→link = temp;
      *ptr = temp;
}
```

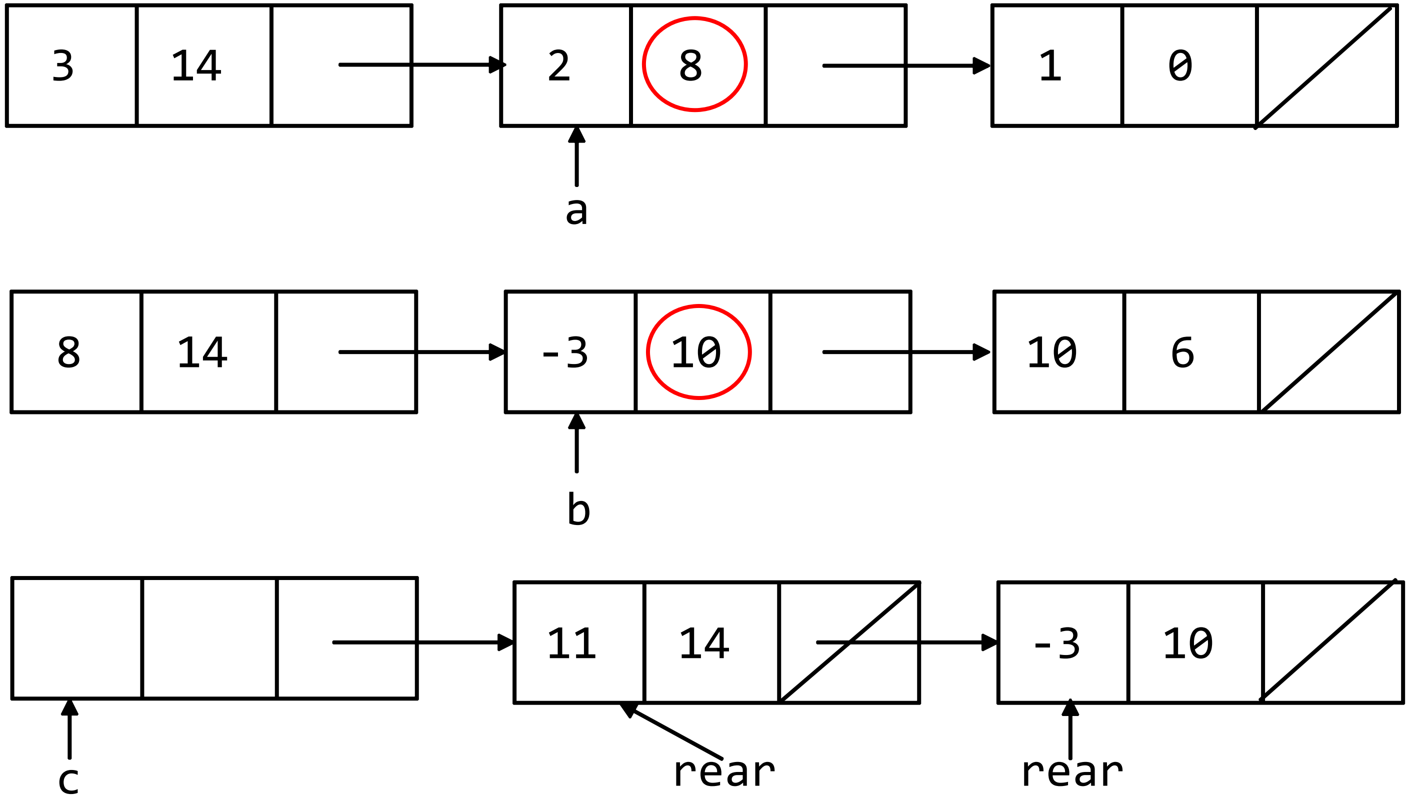
Polynomials – Addition (1)

```
if (a-> expon == b->expon)
    attach(a->coef + b->coef, a->expon, &rear);
```



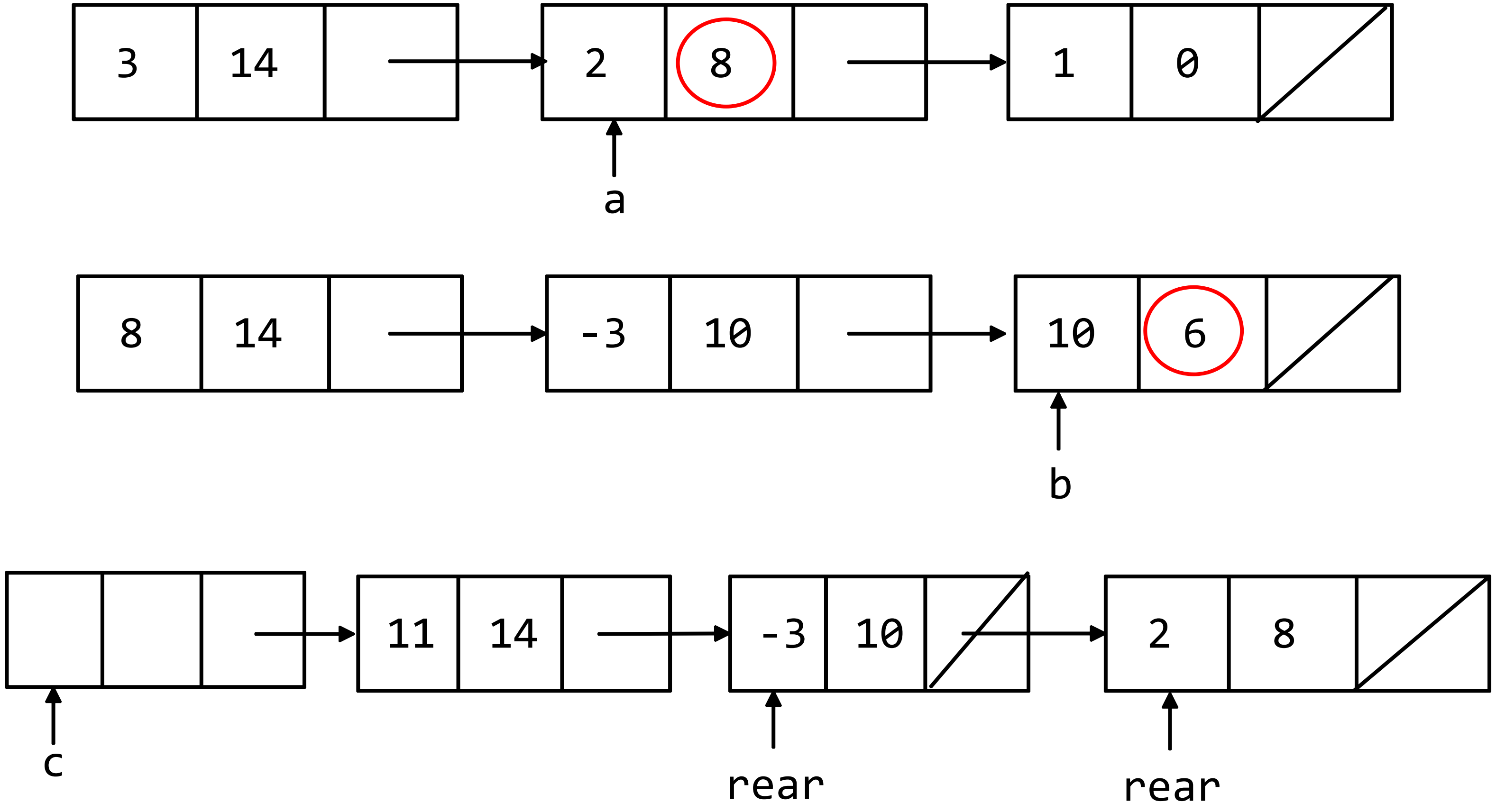
Polynomials – Addition (2)

```
if (a-> expon < b->expon)
      attach(b->coef, b->expon,&rear);
```



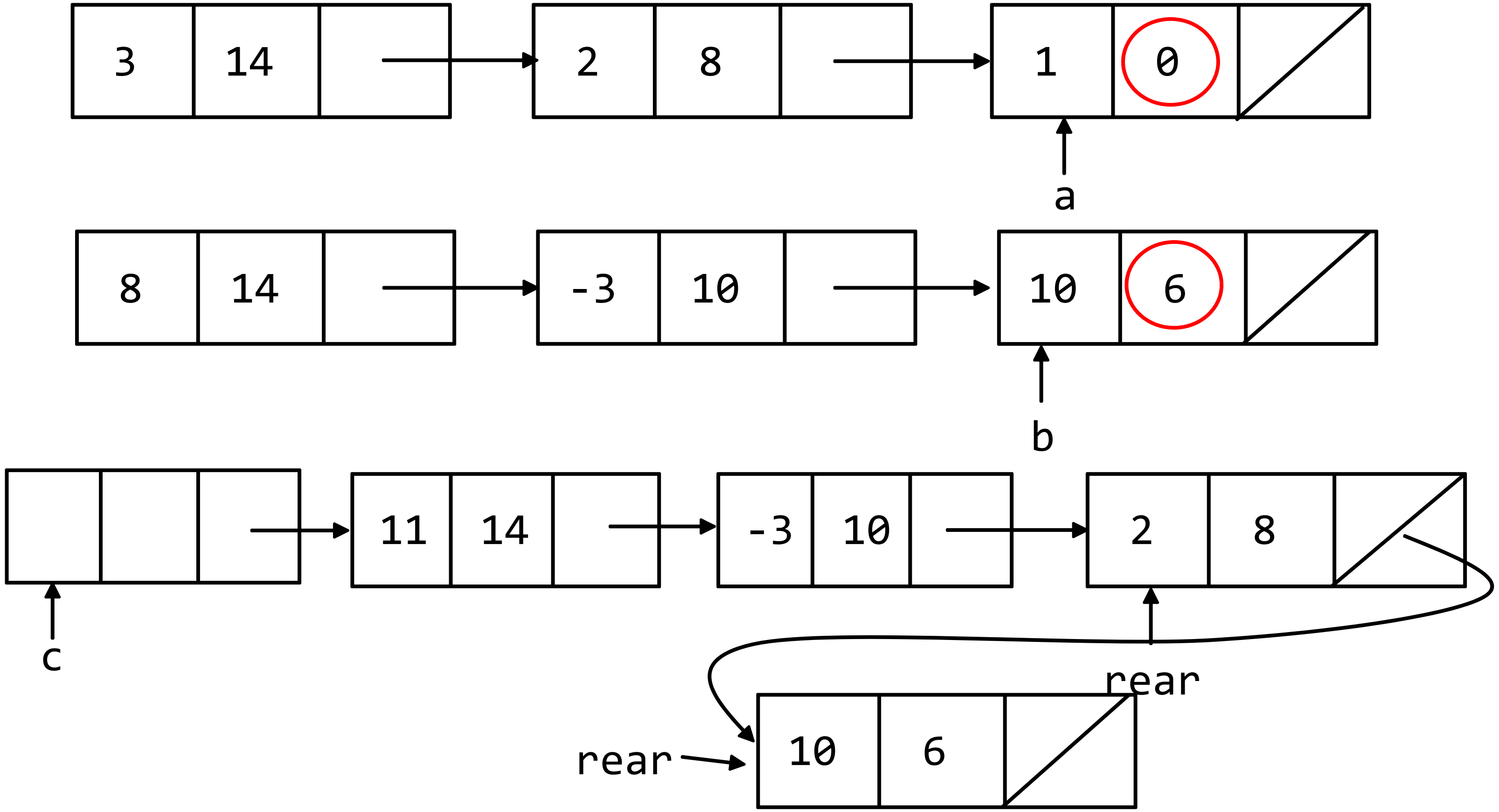
Polynomials – Addition (3)

```
if (a-> expon > b->expon)
   attach(a->coef, a->expon, &rear);
```



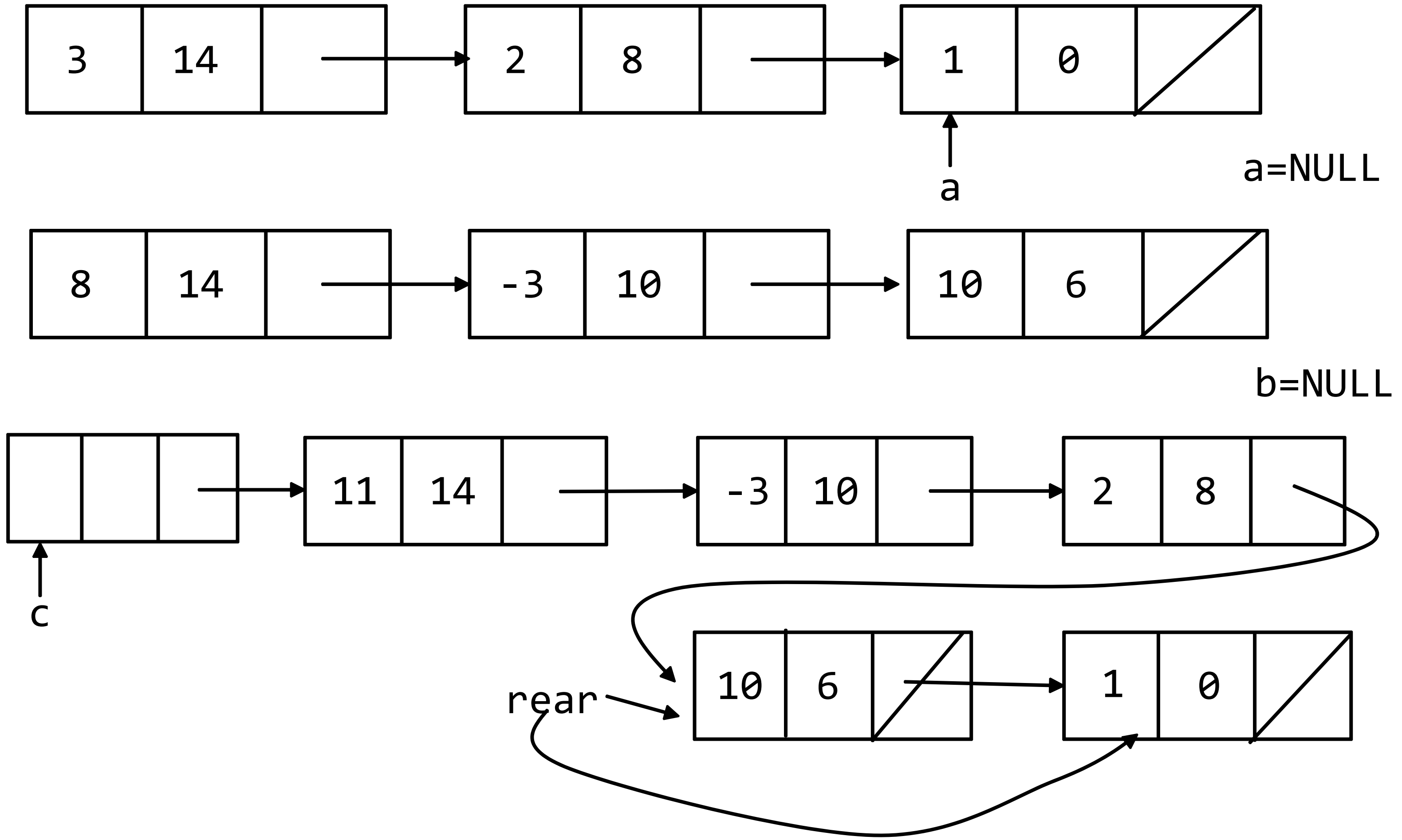
Polynomials – Addition (4)

```
if (a-> expon < b->expon)
   attach(b->coef, b->expon, &rear);
```



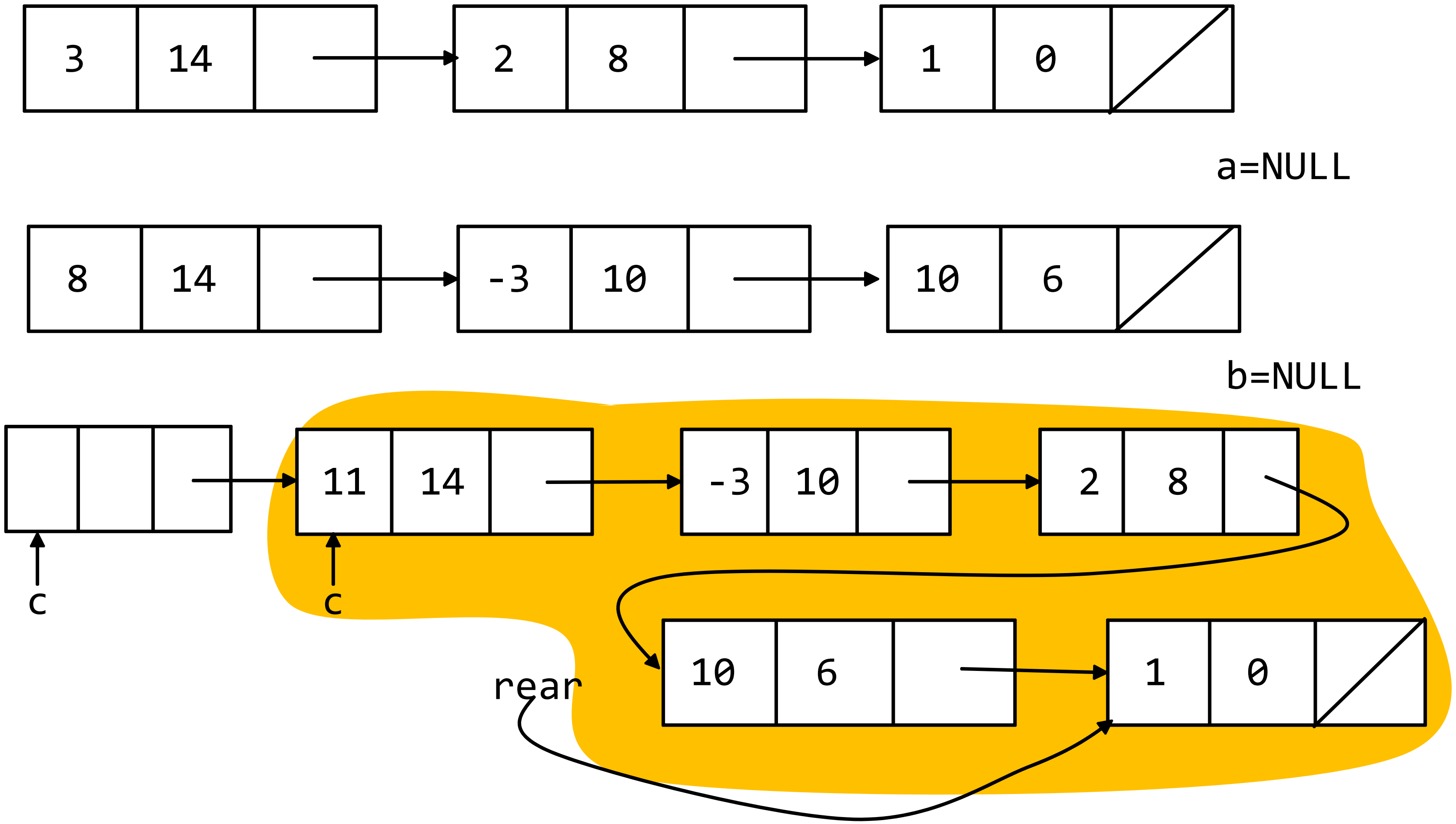
Polynomials – Addition (5)

```
for (; a; a = a->link)
        attach(a->coef,a->expon,&rear);
```



Polynomials – Addition (6)

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



Analysis of padd

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

$$B(x) = b_{n-1}x^{f_{n-1}} + \ldots + b_0 x^{f_0}$$

where $a_i, b_i \neq 0, e_{m-1} > \ldots > e_0 \geq 0, f_{n-1} > \ldots > f_0 \geq 0,$

$0 \leq$ number of coefficient additions $\leq \min\{m, n\}$

number of terms $\leq m + n$

→ number of exponent comparisons $\leq m + n$

→ f(m,n) = O(m+n)

Erasing Polynomials

❖ Compute e(x) = a(x) * b(x) + d(x)

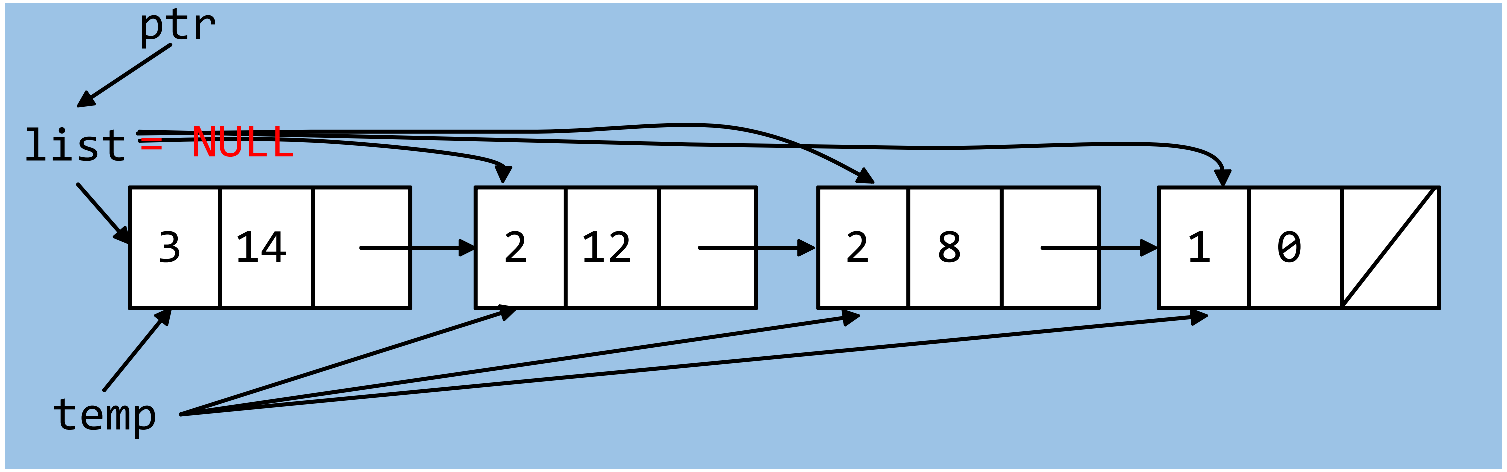
```
polyPointer a, b, d, e;
...
a = readPoly();
b = readPoly();
d = readPoly();
temp = pmult(a, b); /* only hold a partial result for d(x) */
e = padd(temp, d);
printPoly(e);
```

- We created *temp(x)* only to hold a partial result for *d(x)*
  - It would be useful to reclaim the nodes that are being used to represent *temp(x)*

Erasing Polynomials

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

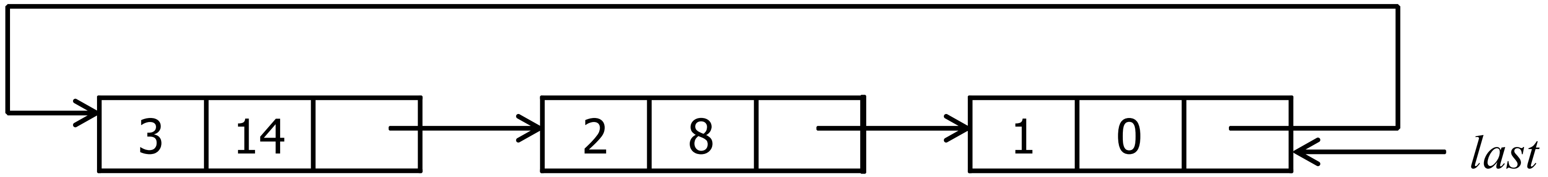
```
erase (&list);
```



Circular List Representation of Polynomials

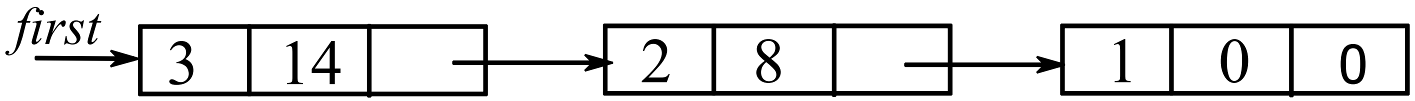
❖ If the link field of the last node points to the first node in the list, all the nodes of a polynomial can be freed more efficiently

❖ Circular representation of $3x^{14} + 2x^{8} + 1$



Available Space List

- Chain
  - A singly linked list in which the last node has a null link



- An efficient erase algorithm for circular lists, by maintaining a list (as a chain) of nodes that have been "freed"
  - When a new node is needed, examine this list
  - If the list is not empty, then we may use one of its nodes
  - Only need to use *malloc* to create a new node when the list is empty

- This list is called the available space list or *avail* list
  - Initially, set *avail* to NULL
- Instead of using *malloc* and *free*, now use ***getNode*** and ***retNode***

getNode Function

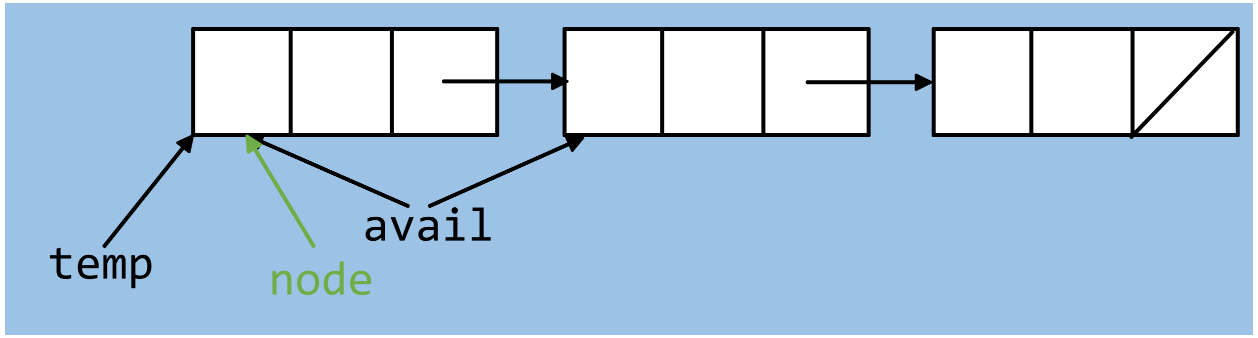
```
polyPointer getNode(void)
{       /* provide a node for use */
        polyPointer node;
        if ( avail ) {
                node = avail;
                avail = avail→link:
        }
        else
                MALLOC( node, sizeof(*node) );

        return node;
}
```

getNode Function

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

```
polyPointer avail; /* a global variable */
```

```
polyPointer temp=getNode();
```

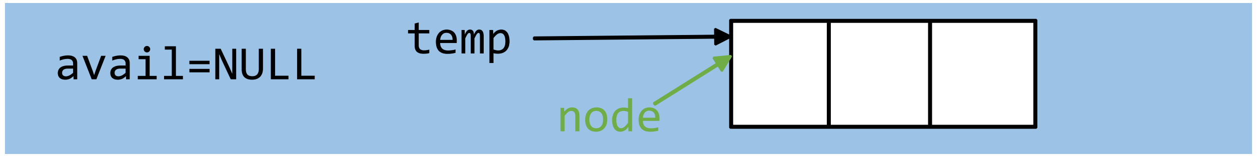


getNode Function

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

```
polyPointer avail; /* a global variable */
```

```
polyPointer temp=getNode();
```



retNode function

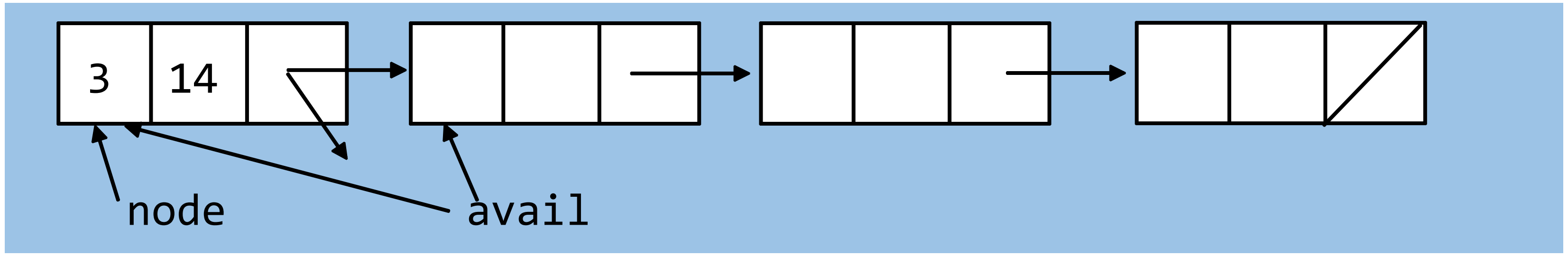
- ❒♏◆☠ □♎♏ Return a node to the available list

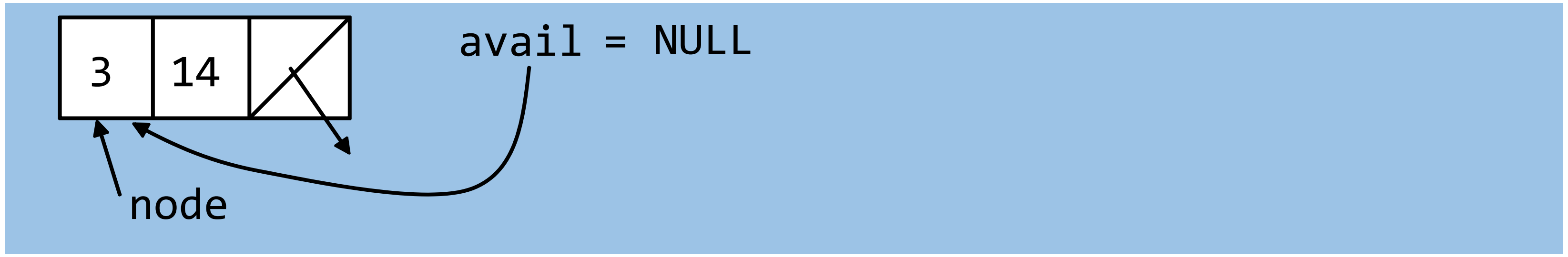
```
void retNode(polyPointer node)
{
        node→link = avail;
        avail = node;
}
```

retNode function

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

```
polyPointer avail;
/* a global variable that points to the first node
of the free nodes list */
```





cerase function

- cerase: Erase a circular list in a fixed amount of time independent of the number of nodes in the list

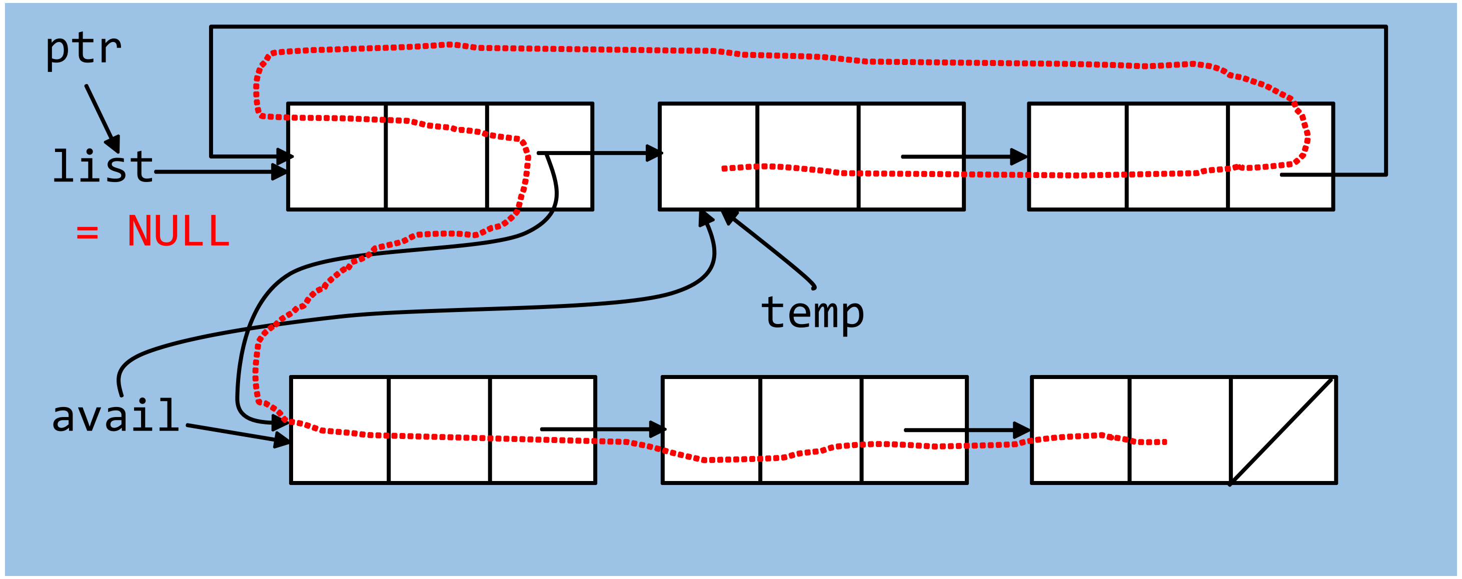
```
void cerase( polyPointer *ptr )
{       /* erase the circular list pointed to by ptr */
        polyPointer temp;
        if (*ptr) {
                temp = (*ptr)→link;
                (*ptr)→link = avail;
                avail = temp;
                *ptr = NULL;
        }
}
```

Erasing a Circular List

```
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;
   }
}
```

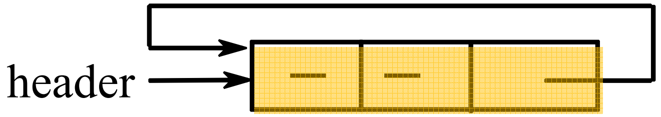
```
polyPointer avail;
/* a global variable that
points to the first node of
the free nodes list */
```

```
cerase(&list);
```

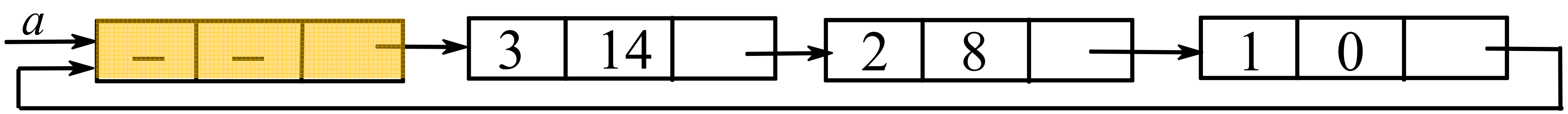


Zero Polynomial

- To avoid the special case of zero polynomial, each polynomial contains one additional node, a header node
  - The *expon* and *coef* fields of this node are irrelevant (expon = -1)
- The representation
  - Zero polynomial



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

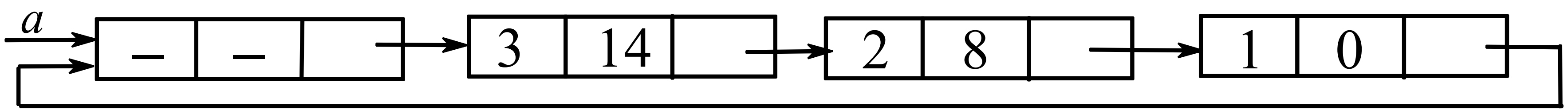


❖ Two polynomials represented as circular lists with header node

```
polyPointer cpadd(polyPointer a, polyPointer b)
{       /* Polynomials a and b are singly linked circular lists with a header node.
            Return a polynomial which is the sum of a and b */
        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;
}
```

Why Doubly Linked Lists

So far, we’ve been working with chains and singly linked circular lists. Too restrictive!



For example, deletion of an arbitrary node requires knowing the preceding node.

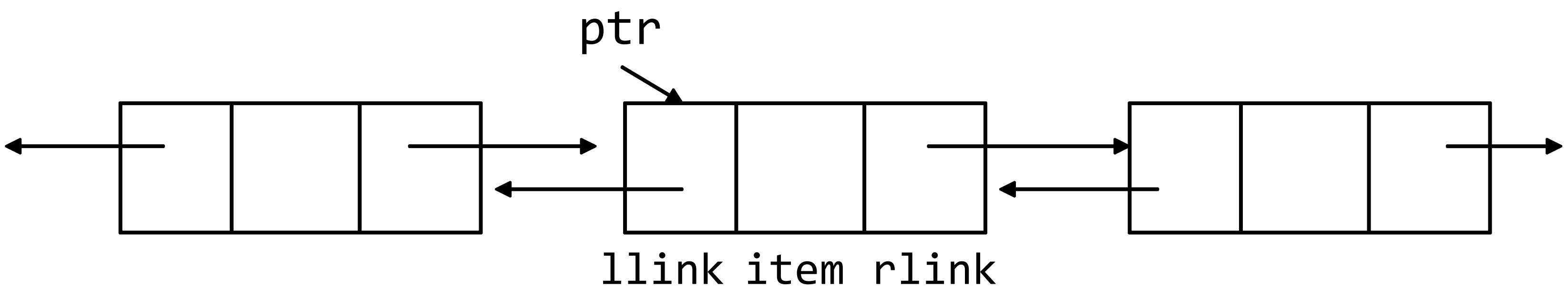
```
void delete(listPointer *first, listPointer trail,
listPointer x)
{ /* delete x from the list, trail is the preceding
node and *first is the front of the list */
     if (trail)
         trail->link = x->link;
     else
         *first = (*first)->link;
     free(x);
}
```

Doubly Linked Lists

- Node structure

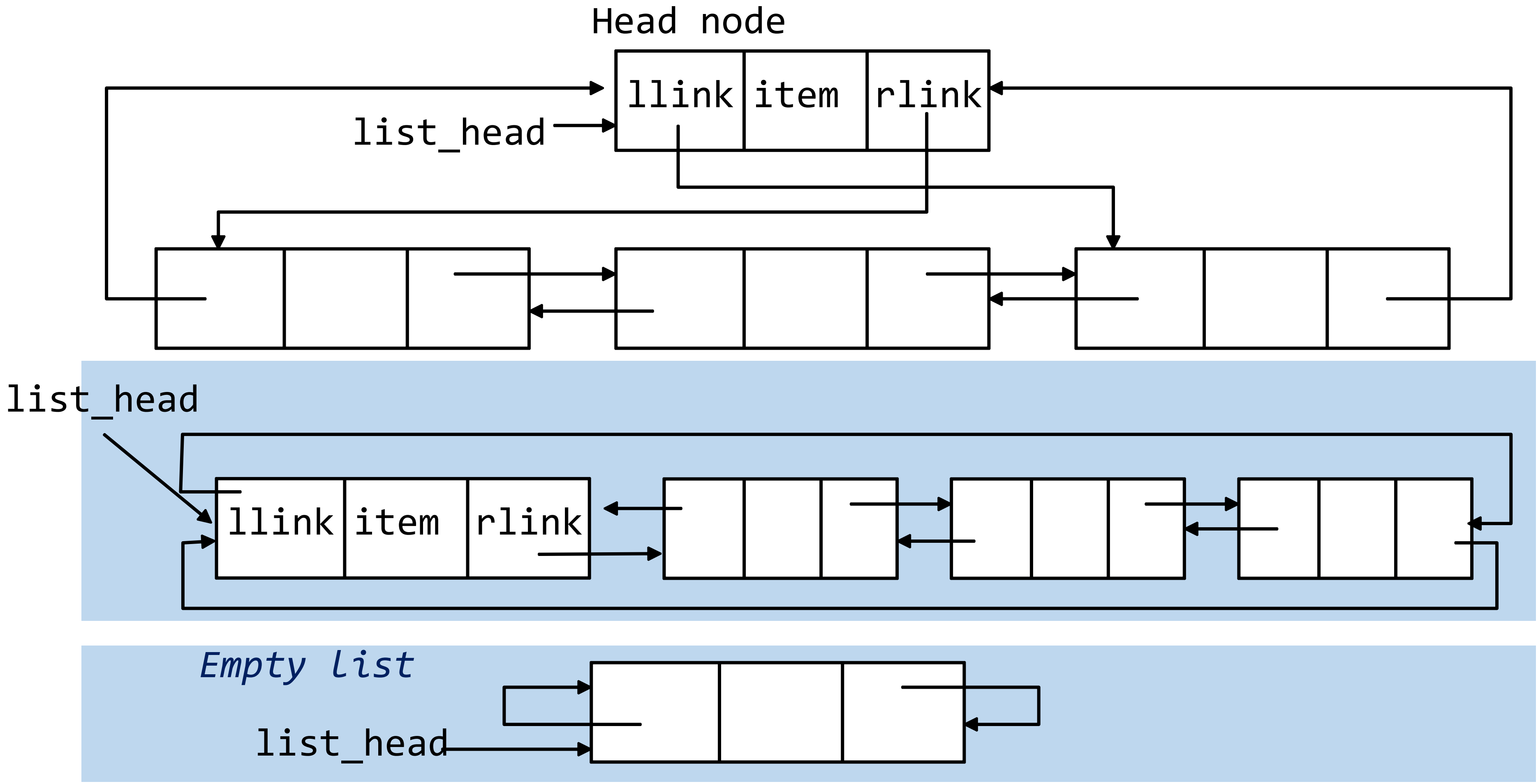
```
typedef struct node* nodePointer;
typedef struct node {
    nodePointer llink;
    element item;
    nodePointer rlink;
}
```

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



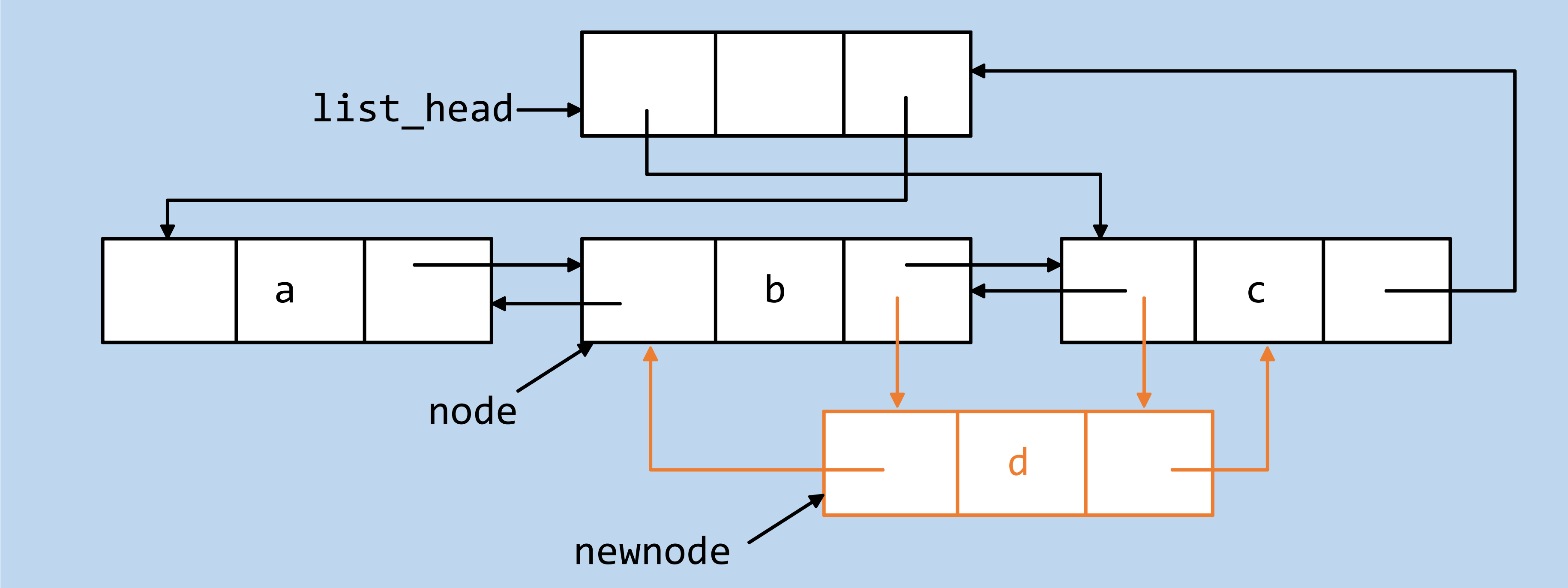
Doubly Linked Circular Lists (DLCL)

- ❖ Doubly linked circular list with head nodes



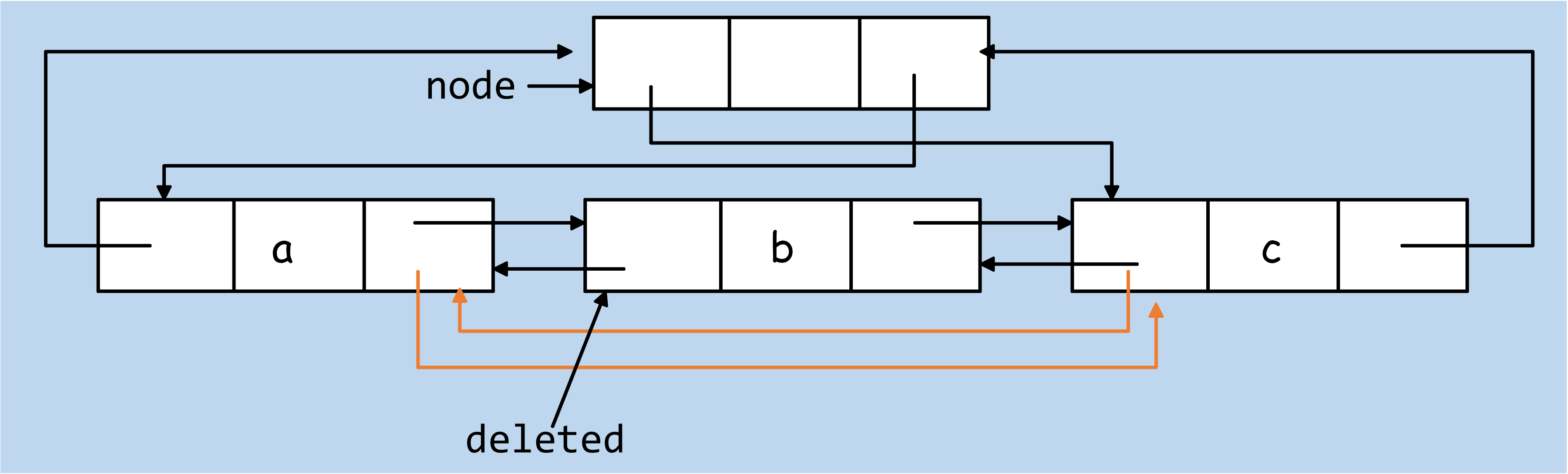
Inserting a Node to a DLCL

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



Deleting a Node from DLCL

```
void  ddelete (nodePointer node, nodePointer deleted)
{ /* delete from the DLCL pointed to by node */
   if (node == deleted)
      printf("Deletion of head node not permitted.\n");
   else {
      deleted->llink->rlink = deleted->rlink;
      deleted->rlink->llink = deleted->llink;
      free(deleted);
   }
}
```


node
a
b
c
deleted