Chap 4. Linked Lists (2)

Contents

4.4 Polynomials

4.4.1 Polynomial Representation

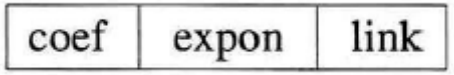
- Polynomial

$A(x) = a_{m-1}x^{e_{m-1}} + \ldots + a_0 x^{e_0}$, *where* $e_{m-1} > e_{m-2} > \ldots > e_1 > e_0 \geq 0$

  - a_i : nonzero coefficients
  - e_i : nonnegative integer exponents

- Representation of Polynomial

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



- Representation of polynomials

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

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

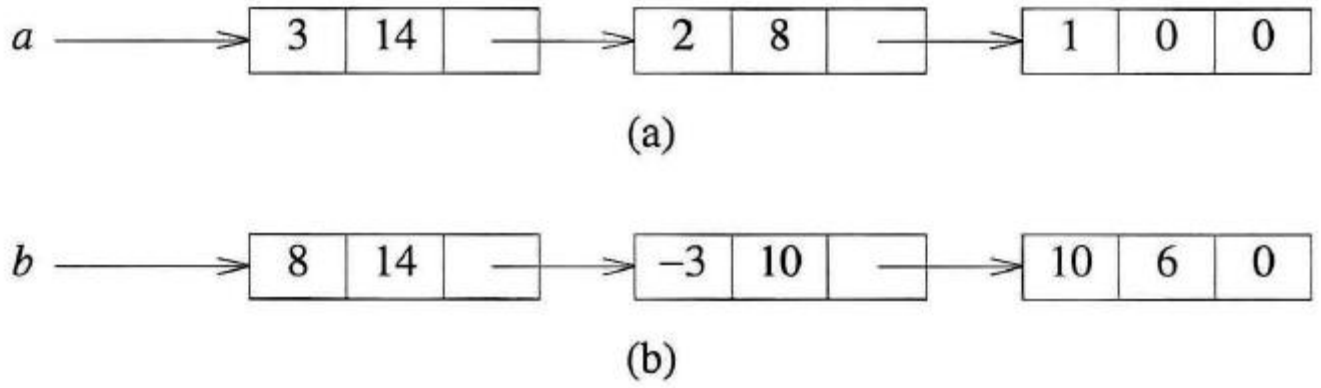


Figure 4.12: Representation of $3x^{14}+2x^8+1$ and $8x^{14}-3x^{10}+10x^6$

4.4.2 Adding Polynomials

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

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

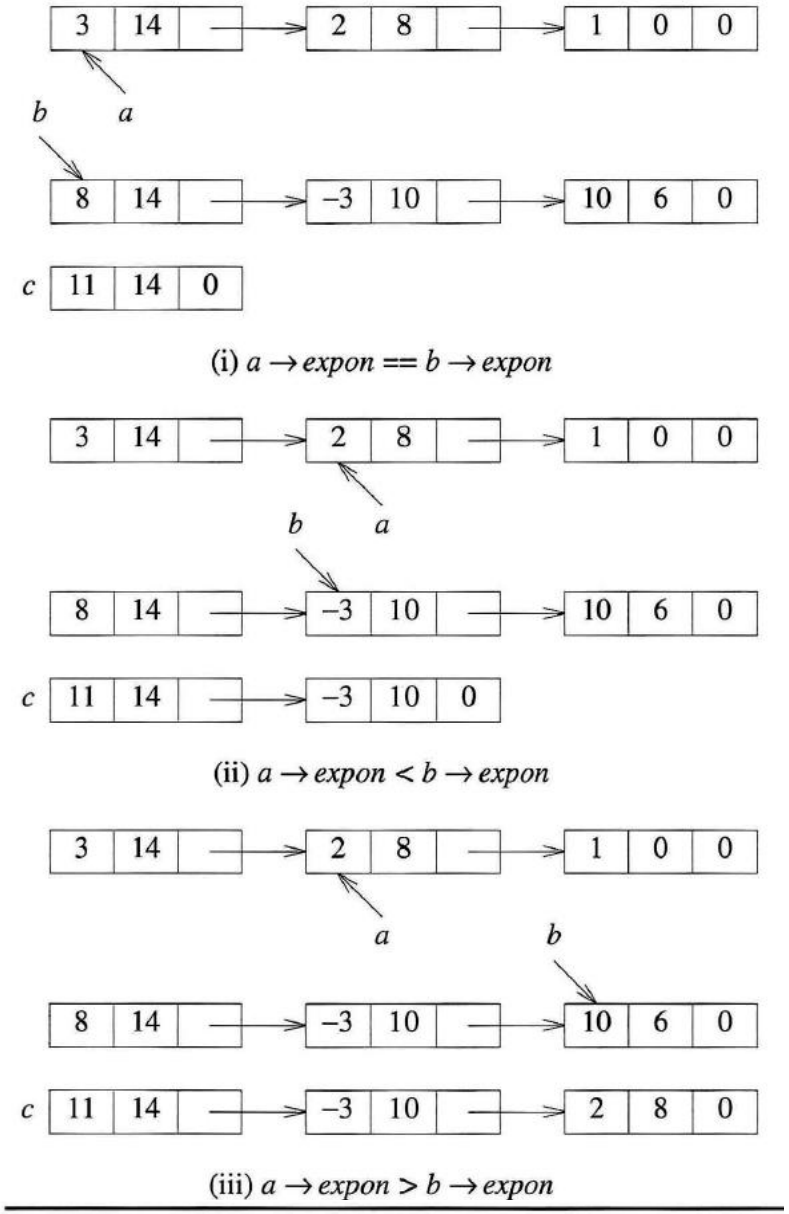


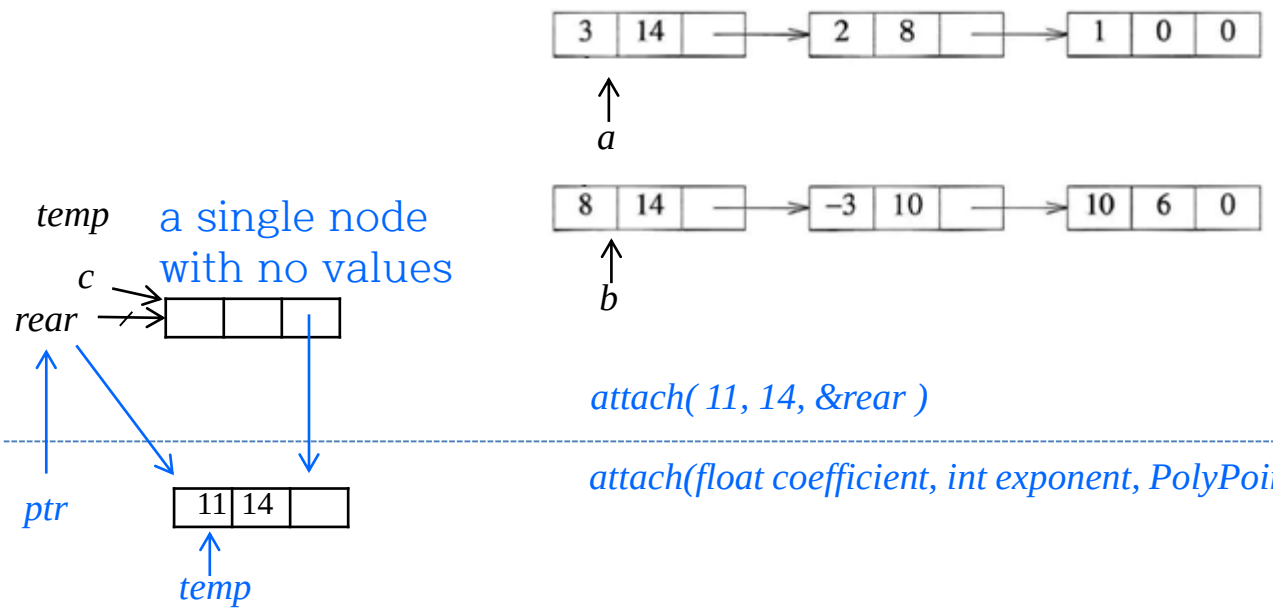
Figure 4.13: Generating the first three terms of $c = a + b$

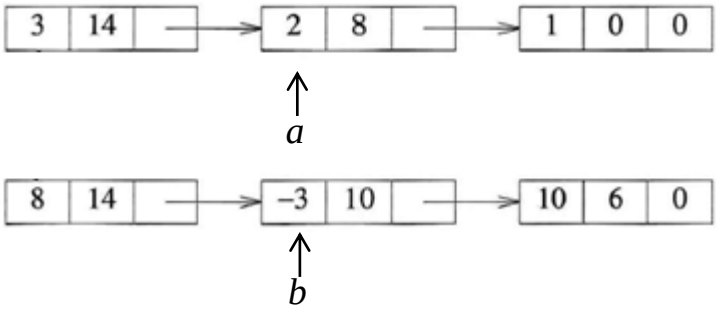
```
polyPointer padd(polyPointer a, polyPointer b)
{/* return a polynomial which is the sum of a and b */
   polyPointer c, rear, temp;
   int sum;
   MALLOC(rear, sizeof(*rear));
   c = rear;
   while (a && b)
      switch (COMPARE(a→expon,b→expon)) {
         case -1: /* a→expon < b→expon */
               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 > b→expon */
               attach(a→coef,a→expon,&rear);
               a = a→link;
      }
   /* copy rest of list a and then list b */
   for (; a; a = a→link) attach(a→coef,a→expon,&rear);
   for (; b; b = b→link) attach(b→coef,b→expon,&rear);
   rear→link = NULL;
   /* delete extra initial node */
   temp = c; c = c→link;  free(temp);
   return c;
}
```

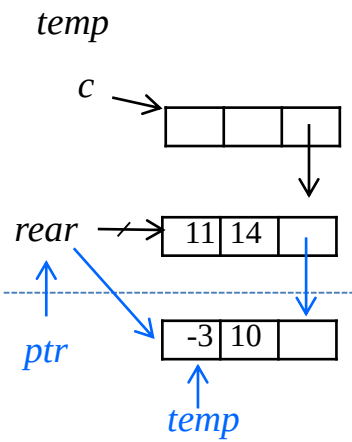
Program 4.9: Add two polynomials

```
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;
}
```

Program 4.10: Attach a node to the end of a list

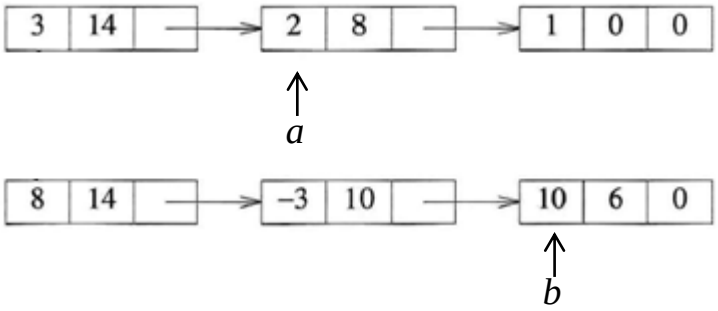


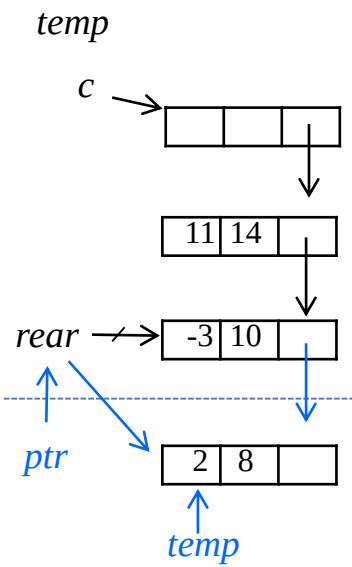




attach(-3, 10, &rear)

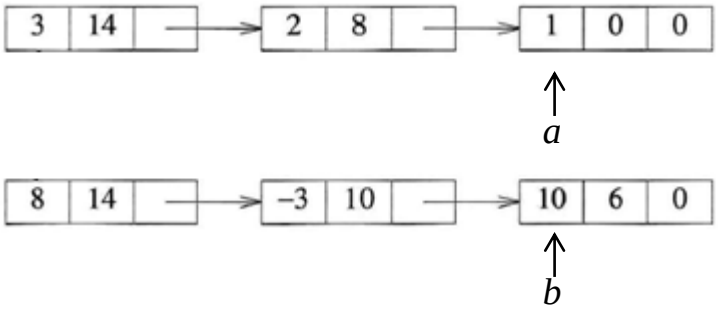
*attach(float coefficient, int exponent, PolyPointer *ptr)*

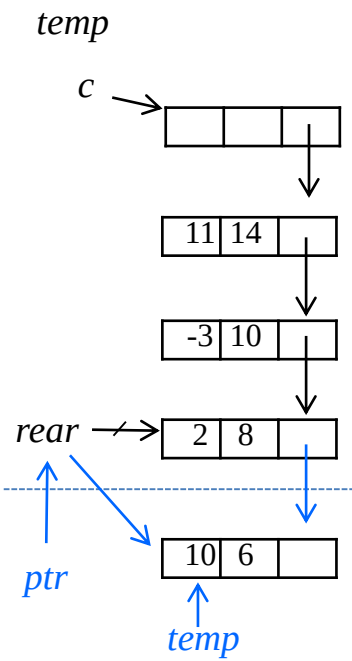




attatch(2, 8, &rear)

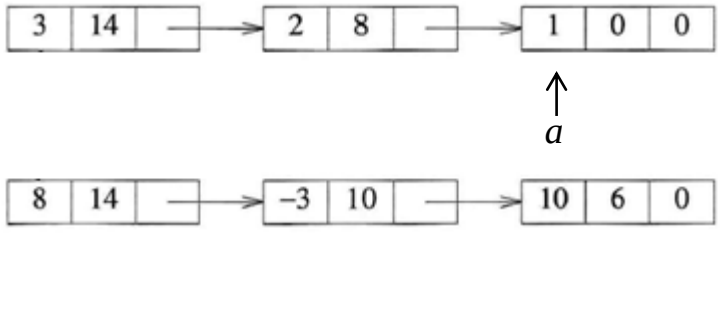
*attatch(float coefficient, int exponent, PolyPointer *ptr)*

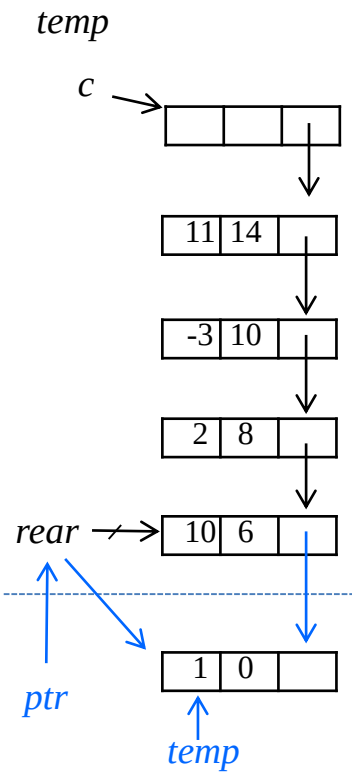




attach(10, 6, &rear)

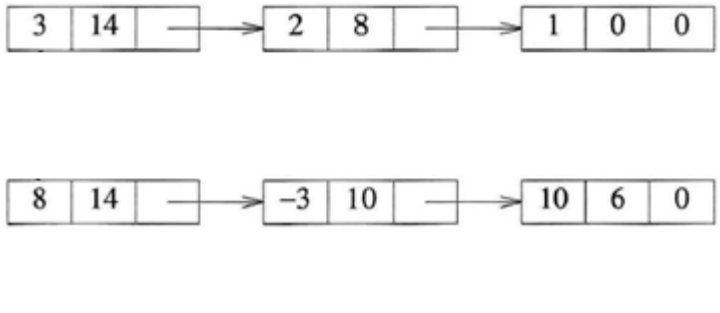
*attach(float coefficient, int exponent, PolyPointer *ptr)*

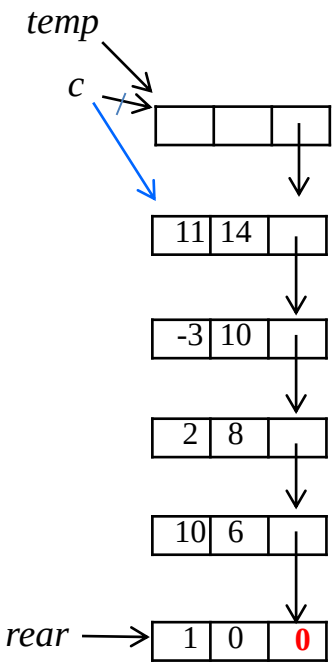




attach(1, 0, &rear)

*attach(float coefficient, int exponent, PolyPointer *ptr)*


3 14
2 8
1 0 0
NULL
a
8 14
-3 10
10 6 0
NULL
b


temp
c
11 14
-3 10
2 8
10 6
rear
1 0 0

- Analysis of ***padd***
  - Three cost measures for this algorithm

    (1) Coefficient additions

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

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

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

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

(2) Exponent comparisons

- One comparison on each iteration of the `while` loop
- The number of iterations is bounded by $m + n$

  ex) $m+n-1$ iterations for $m = n$ and

$$e_{m-1} > f_{m-1} > e_{m-2} > f_{m-2} > \cdots > e_1 > f_1 > e_0 > f_0$$

(3) Creations of new nodes for *c*

- The maximum number of terms in *c* is $m + n$

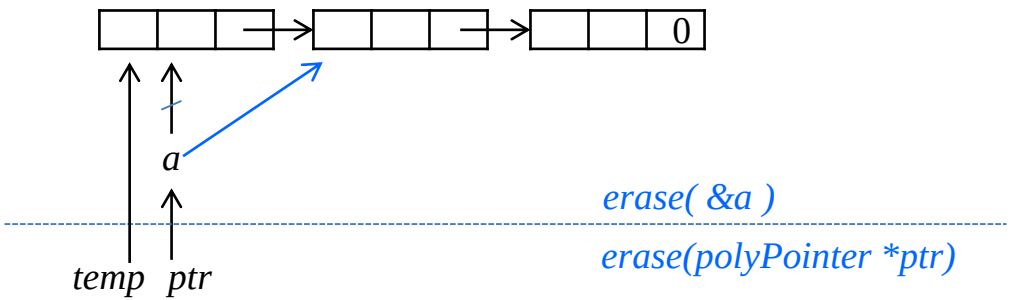
From (1)~(3),

- the total time complexity is **O (*m* + *n*)**

4.4.3 Erasing Polynomials

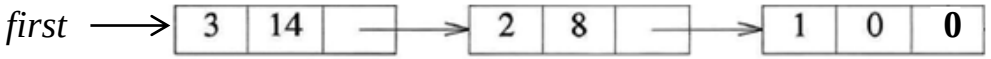
```
void erase(polyPointer *ptr)
{/* erase the polynomial pointed to by ptr */
   polyPointer temp;
   while (*ptr) {
      temp = *ptr;
      *ptr = (*ptr)→link;
      free(temp);
   }
}
```

Program 4.11: Erasing a polynomial

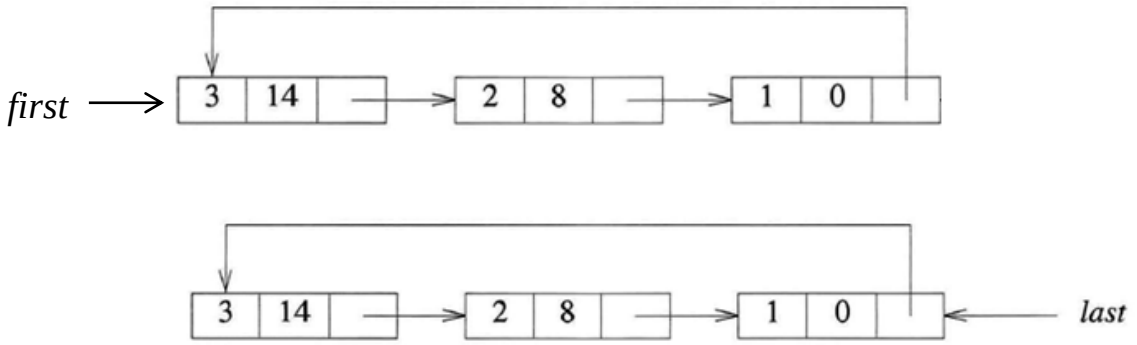


4.4.4 Circular List Representation of Polynomials

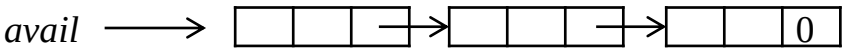
- Chain
  - A singly linked list in which the last node has a null link



- Circular list
  - The link filed of the last node points to the first node in the list

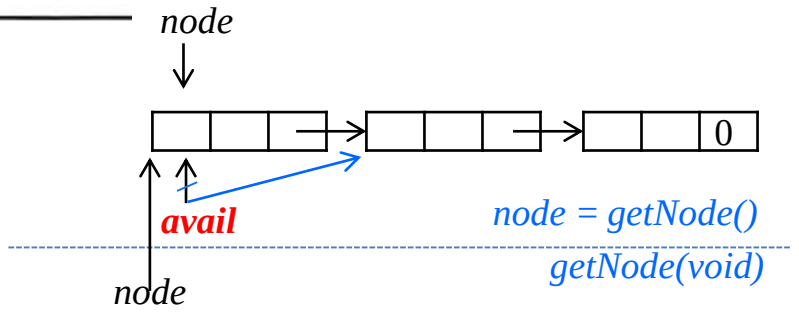


- Available space list
  - *A chain of nodes that have been "freed"*
  - Use *getNode* and *retNode*, instead of *malloc* & *free*



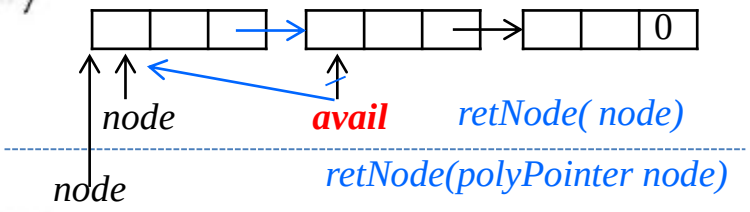
- When maintaining it,
  - we can obtain an efficient erase algorithm for circular list.

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



Program 4.12: *getNode* function

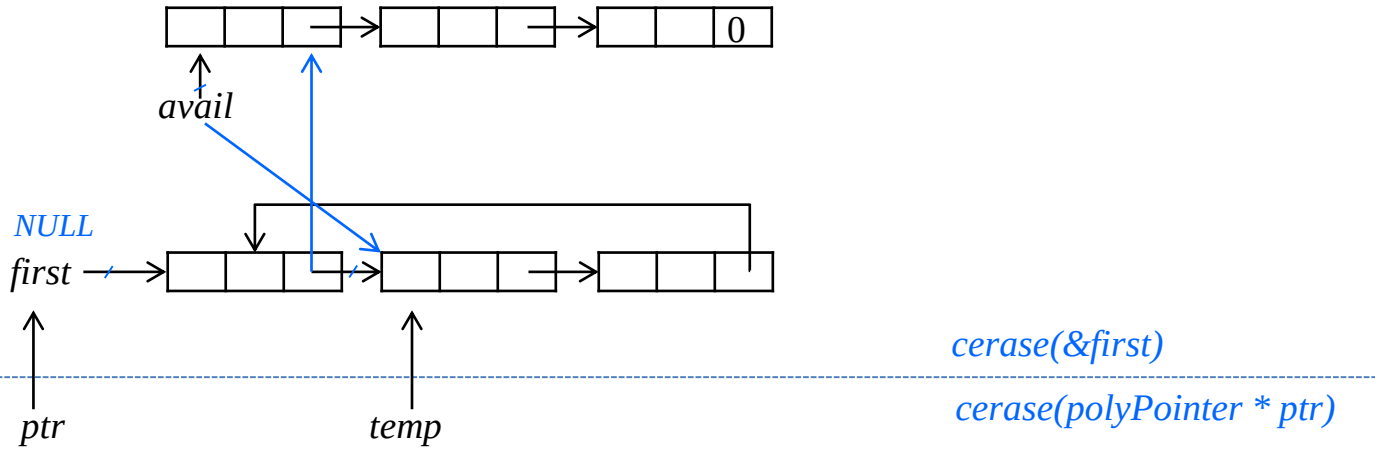
```
void retNode(polyPointer node)
{/* return a node to the available list */
   node→link = avail;
   avail = node;
}
```



Program 4.13: *retNode* function

```
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;
   }
}
```

Program 4.14: Erasing a circular list



- To avoid handling the zero polynomial as a special case, *a header node* is added.

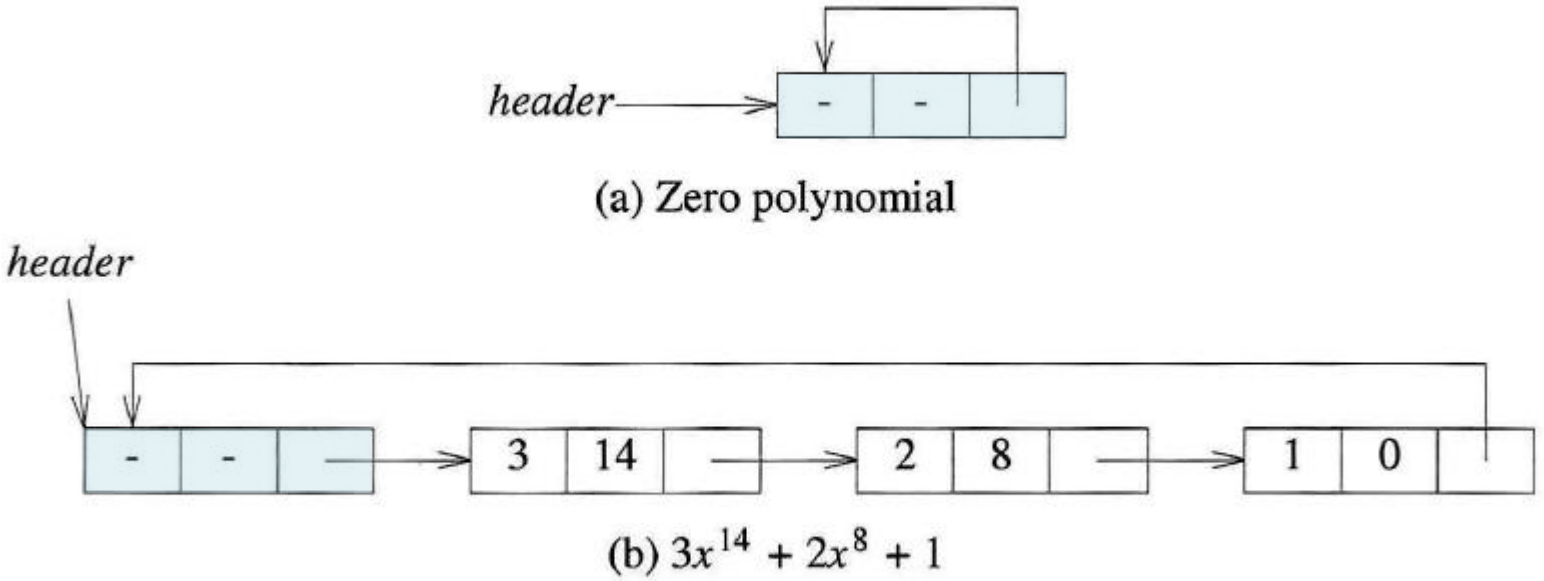
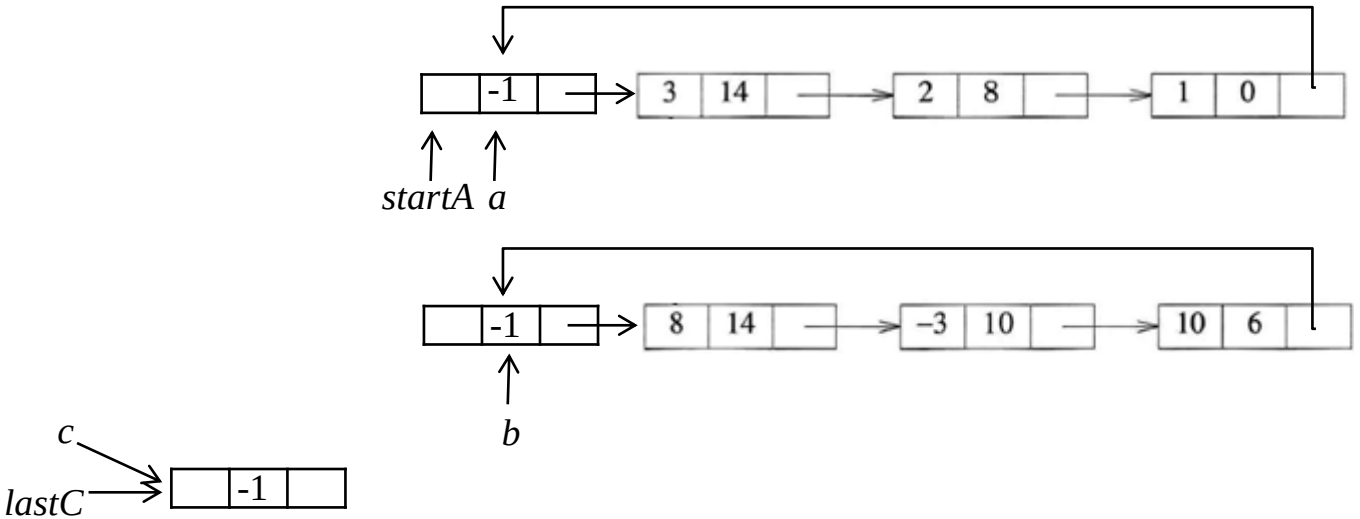
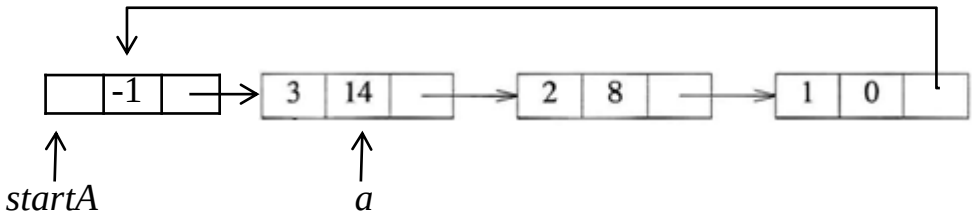


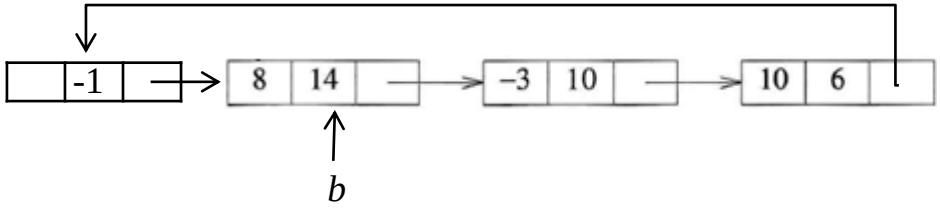
Figure 4.15: Example polynomials with header nodes

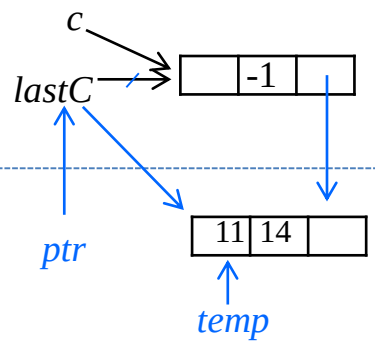
```
polyPointer cpadd(polyPointer a, polyPointer b)
{/* polynomials a and b are singly linked circular lists
    with a header node. Return a polynomial which is
    the sum of a and b */
  polyPointer startA, c, lastC;
  int sum, done = FALSE;
  startA = a;           /* record start of a */
  a = a->link;          /* 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;
}
```

Program 4.15: Adding two polynomials represented as circular lists with header nodes


-1
3
14
2
8
1
0
startA
a
-1
8
14
−3
10
10
6
b
c
lastC
-1

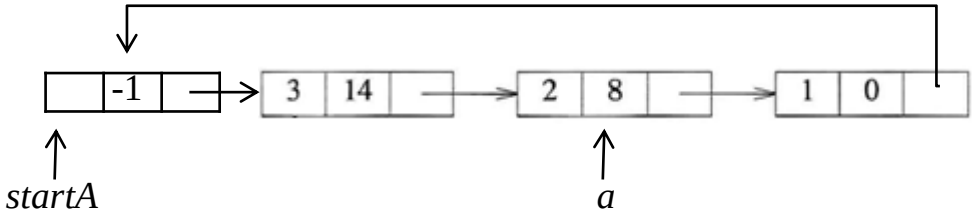


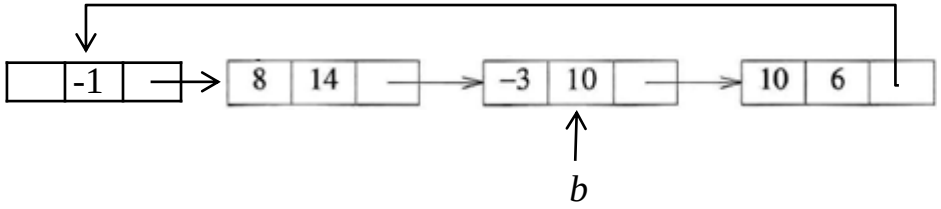


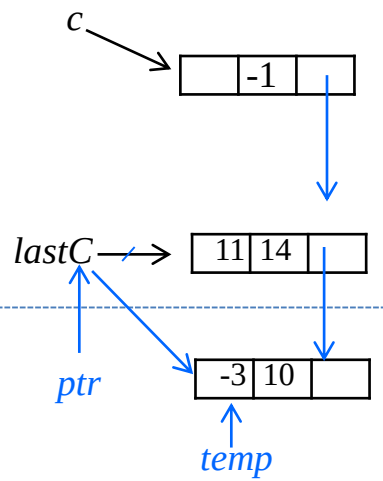


attach(11, 14, &lastC)

*attach(float coefficient, int exponent, PolyPointer *ptr)*

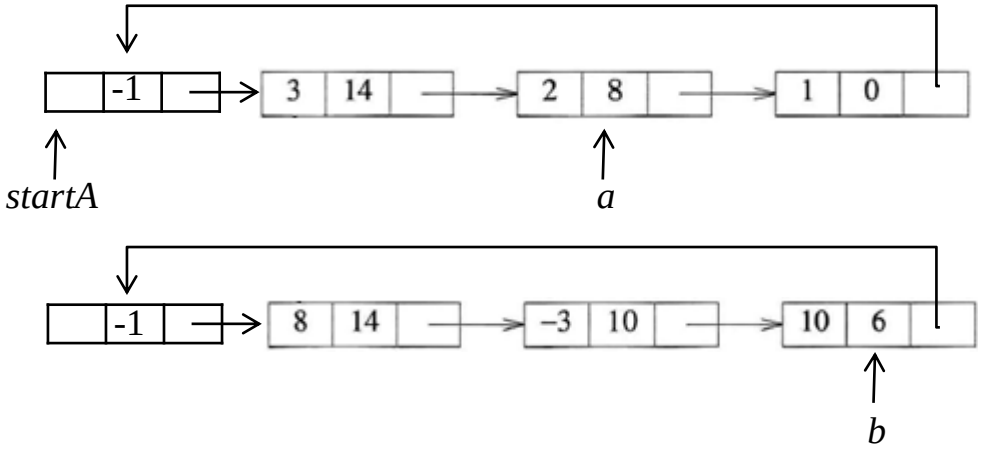


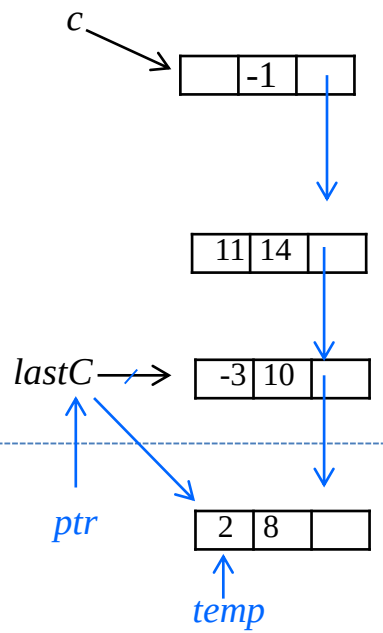




attach(-3, 10, &lastC)

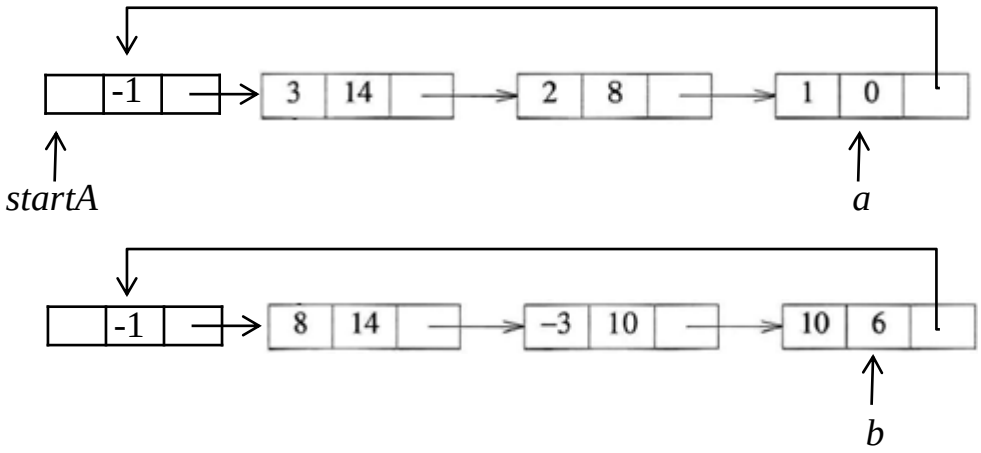
*attach(float coefficient, int exponent, PolyPointer *ptr)*

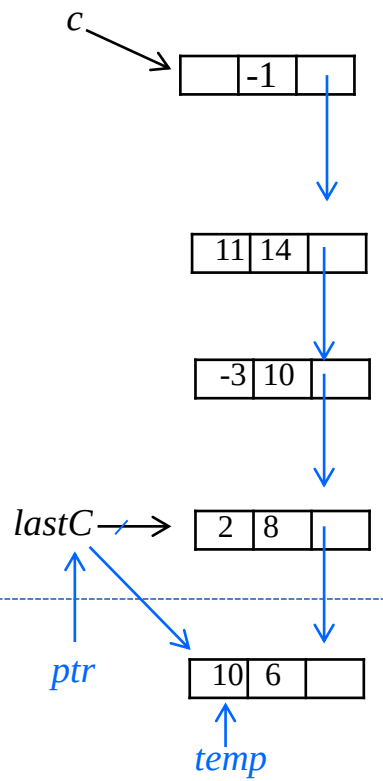




attach(2, 8, &lastC)

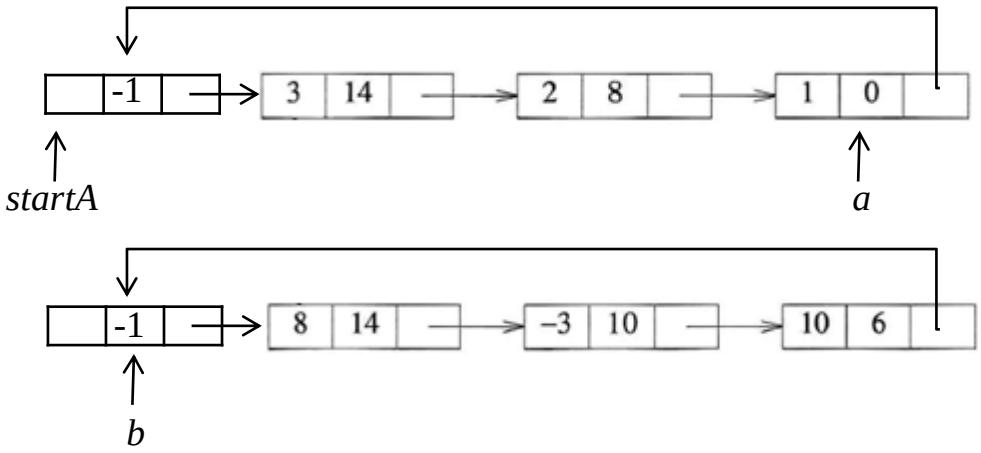
*attach(float coefficient, int exponent, PolyPointer *ptr)*

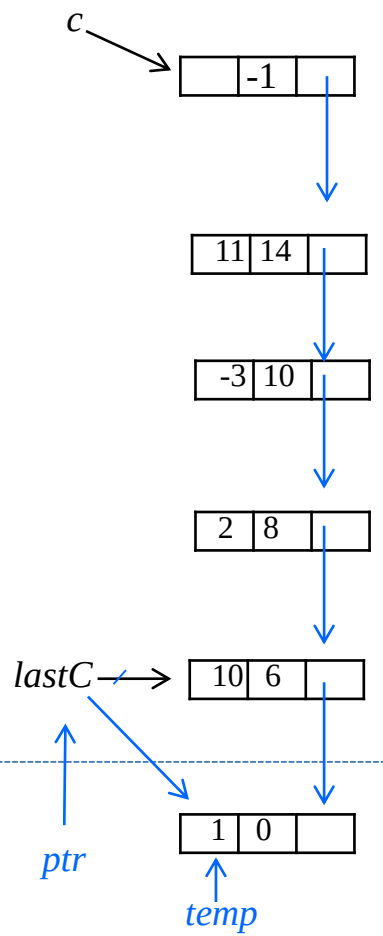




attach(10, 6, &lastC)

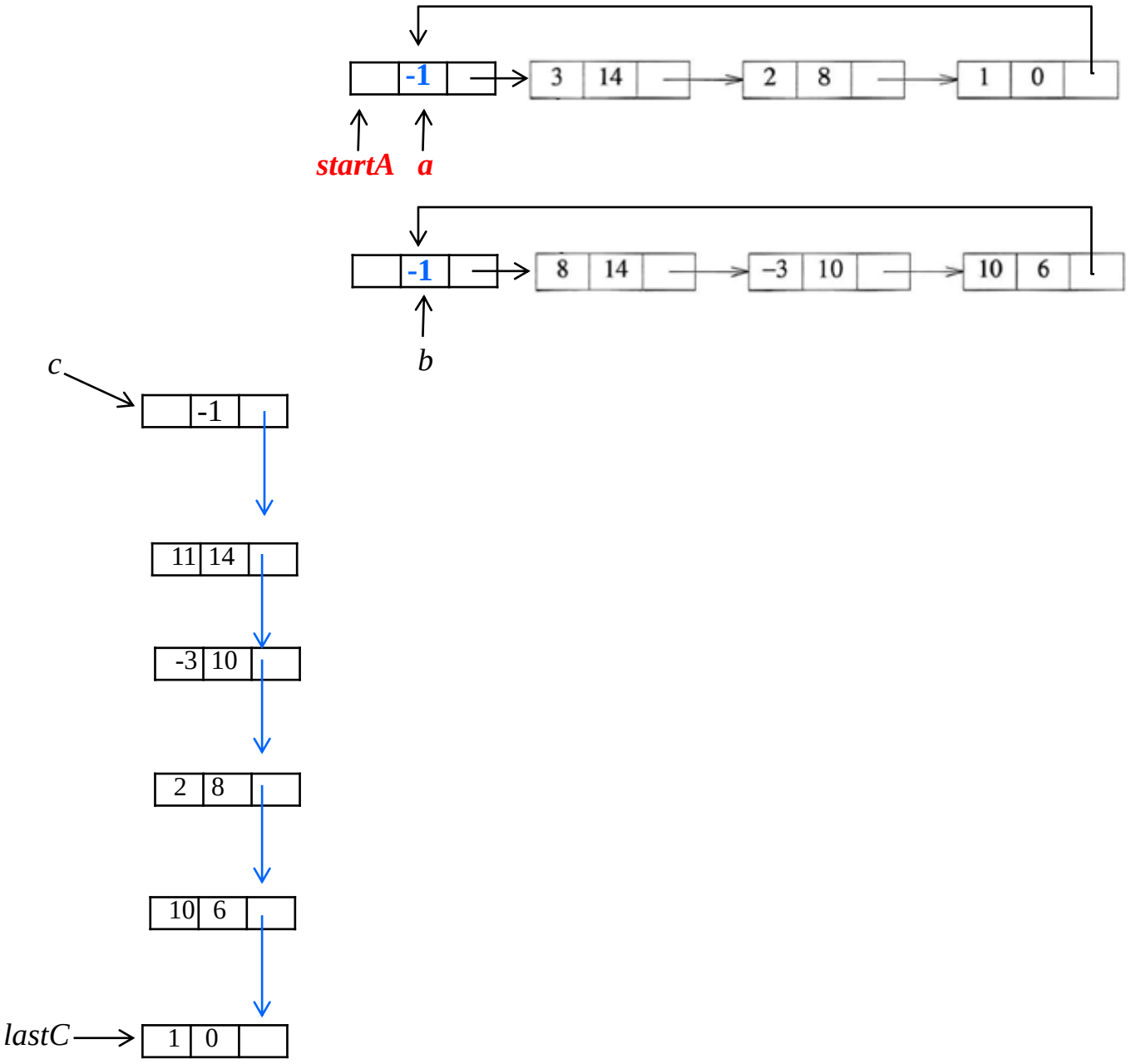
*attach(float coefficient, int exponent, PolyPointer *ptr)*

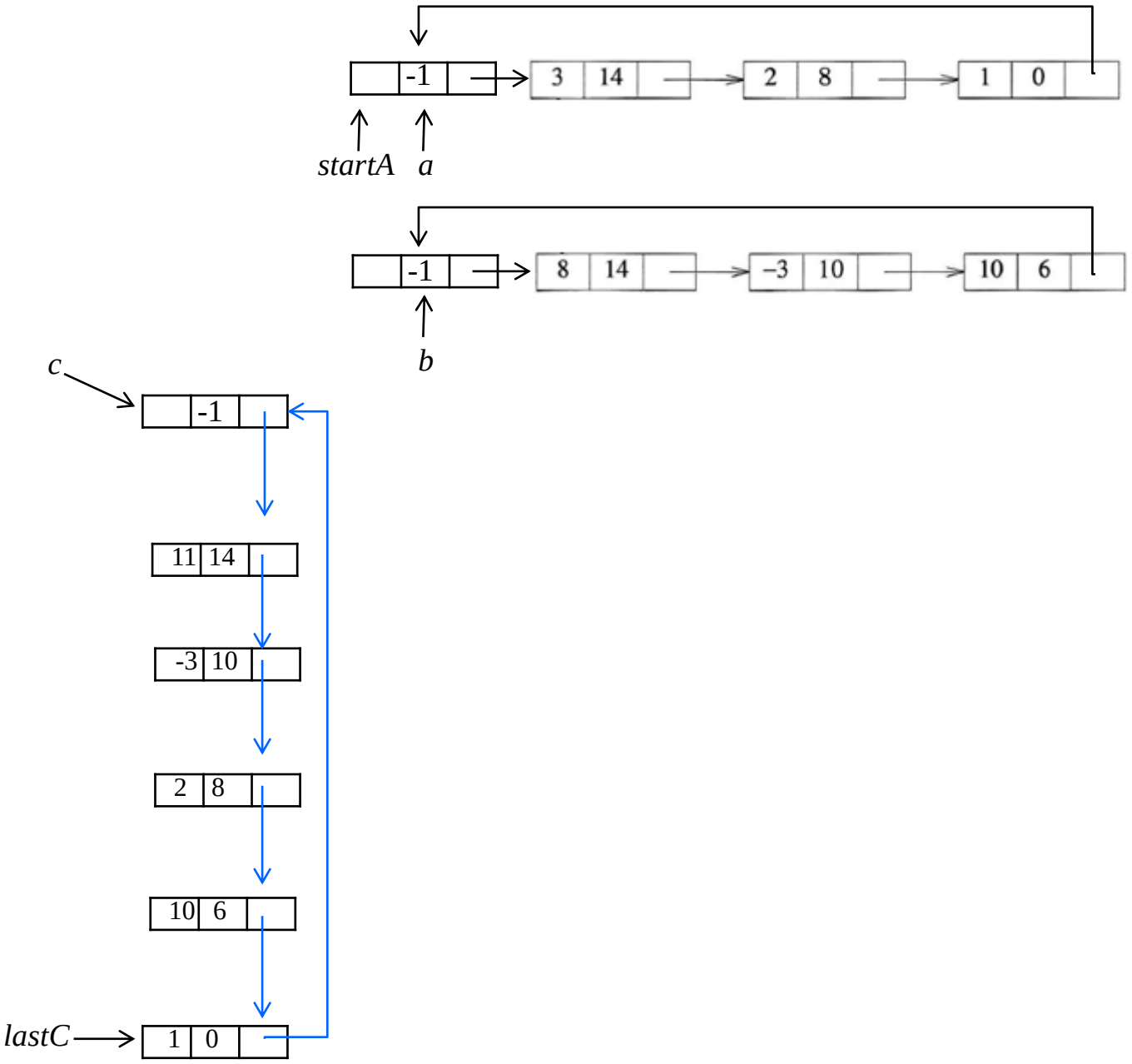




attach(1, 0, &lastC)

*attach(float coefficient, int exponent, PolyPointer *ptr)*


-1
3
14
2
8
1
0
startA
a
-1
8
14
–3
10
10
6
b
c
-1
11
14
-3
10
2
8
10
6
lastC
1
0


-1
3
14
2
8
1
0
startA
a
-1
8
14
−3
10
10
6
b
c
-1
11
14
-3
10
2
8
10
6
lastC
1
0

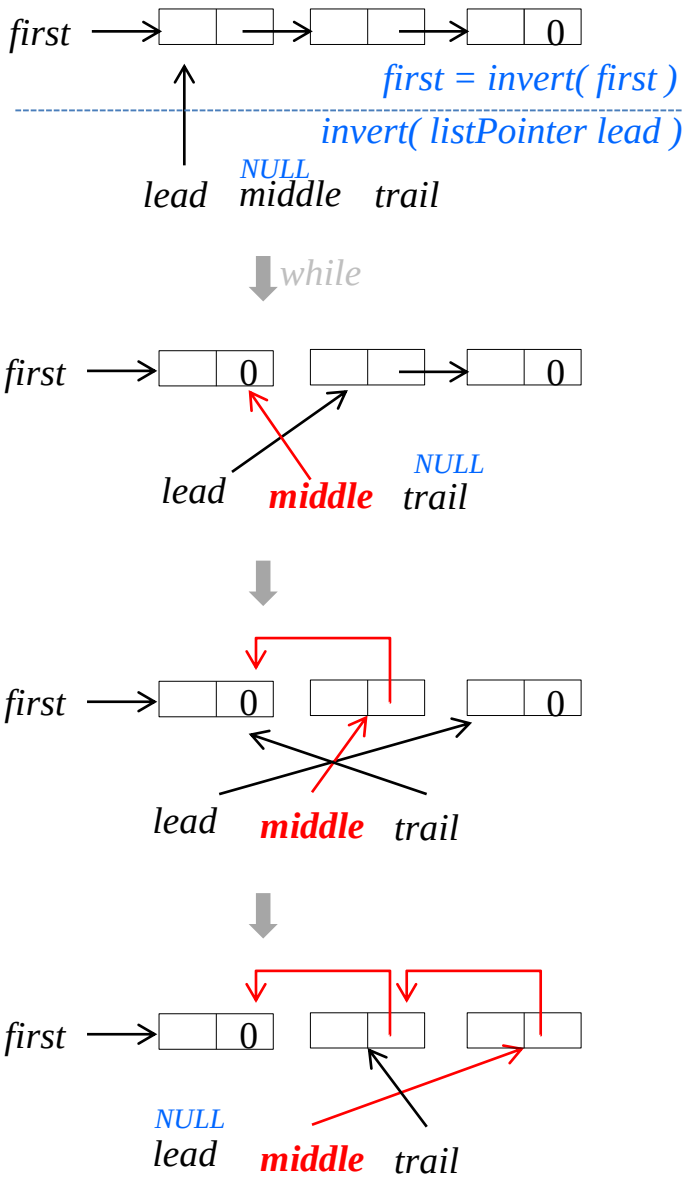
4.5 Additional List Operations

4.5.1 Operations For Chains

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

```
listPointer invert(listPointer lead)
{/* invert the list pointed to by lead */
   listPointer middle,trail;
   middle = NULL;
   while (lead) {
      trail = middle;          ①
      middle = lead;           ②
      lead = lead→link;        ③
      middle→link = trail;     ④
   }
   return middle;
}
```

Program 4.16: Inverting a singly linked list

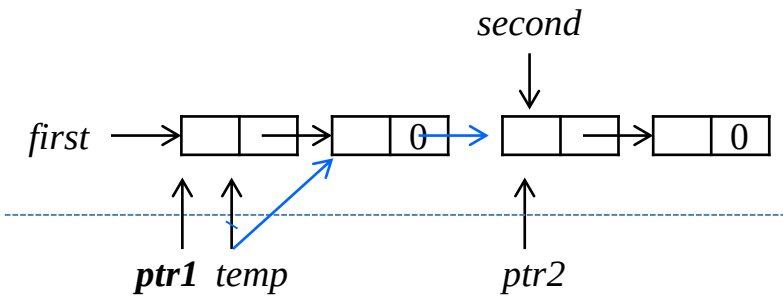


```
listPointer concatenate(listPointer ptr1, listPointer ptr2)
{/* produce a new list that contains the list
    ptr1 followed by the list ptr2. The
    list pointed to by ptr1 is changed permanently */
   listPointer temp;
   /* check for empty lists */
   if (!ptr1) return ptr2;
   if (!ptr2) return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

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

   /* link end of first to start of second */
   temp→link = ptr2; return ptr1;
}
```

Program 4.17: Concatenating singly linked lists



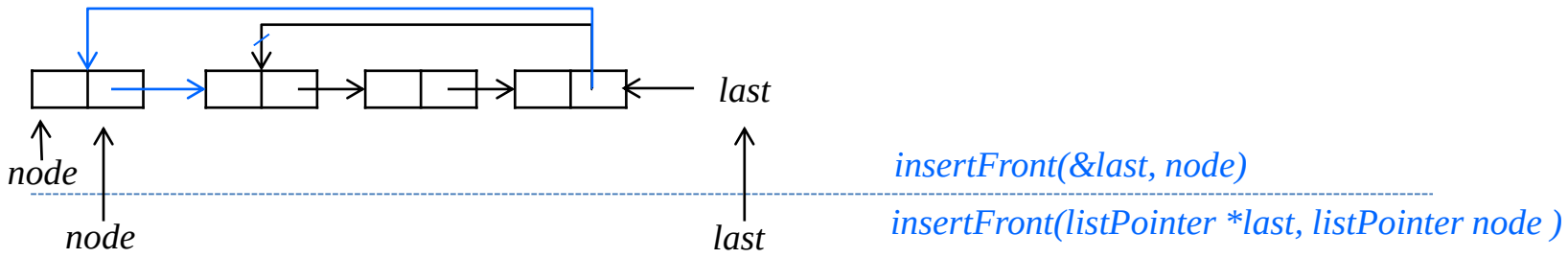
concatList = concatenate(first, second)

concatenate(listPointer ptr1, listPointer ptr2)

4.5.2 Operations For Circularly Linked Lists

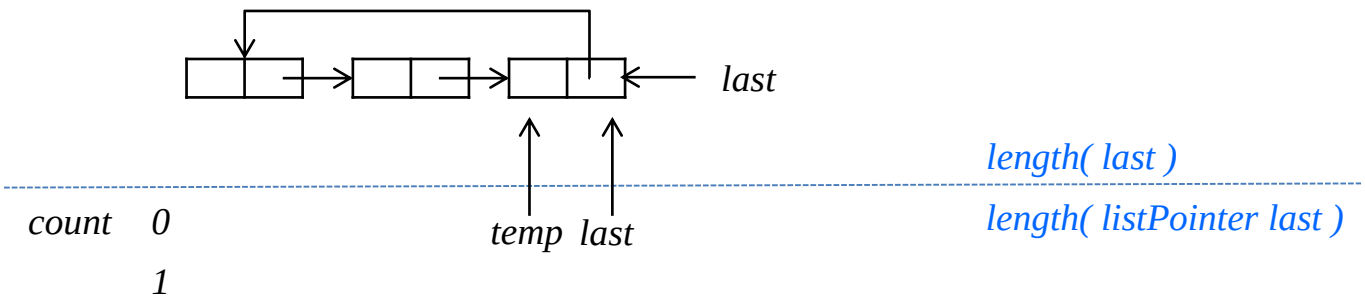
```
void insertFront(listPointer *last, listPointer node)
{/* insert node at the front of the circular list whose
    last node is last */
   if (!(*last)) {
   /* list is empty, change last to point to new entry */
      *last = node;
      node→link = node;
   }
   else {
   /* list is not empty, add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

Program 4.18: Inserting at the front of a list



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

Program 4.19: Finding the length of a circular list



4.8 Doubly Linked Lists

- Limitation in *chains* and *singly linked circular lists*
  - The only way to find a specific node *p* or the node that precedes the node *p* is *to start at the beginning of the list*.
  - Easy deletion of an arbitrary node requires knowing the preceding node.

- It is useful to have *doubly linked lists,* for a problem that
  - need to move in either directions
  - must delete an arbitrary node

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

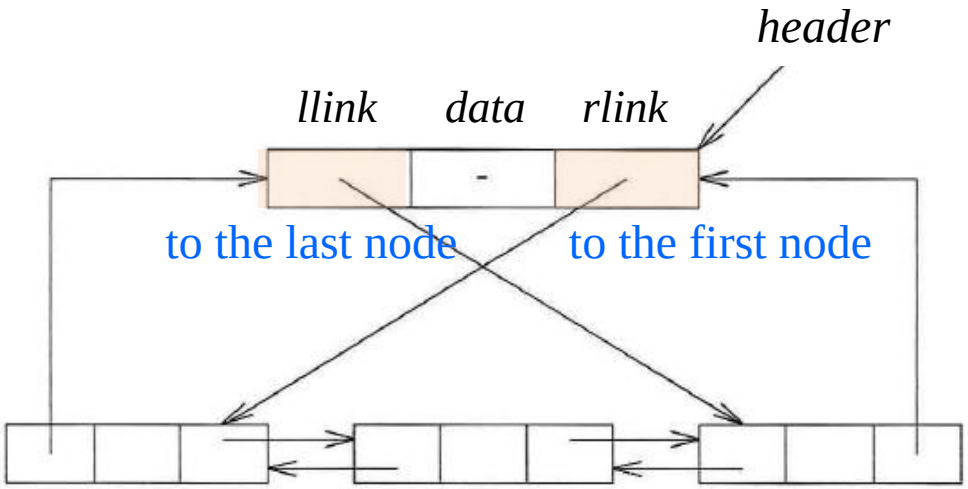


Figure 4.21: Doubly linked circular list with header node

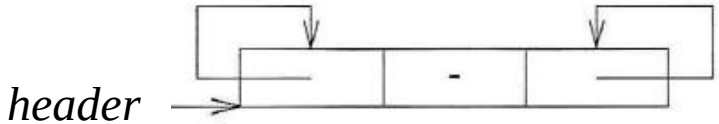
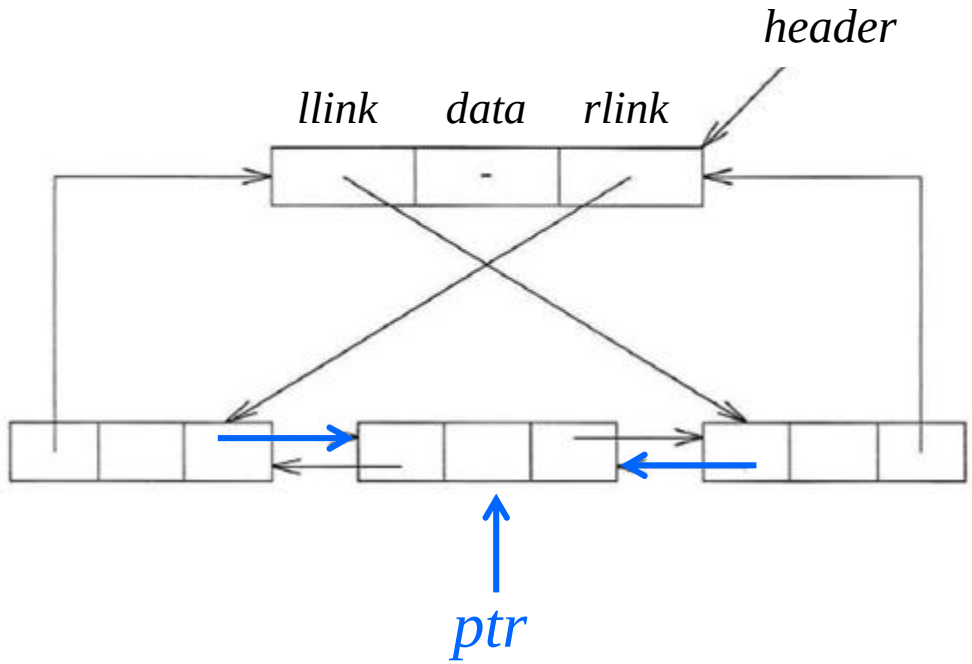


Figure 4.22: Empty doubly linked circular list with header node



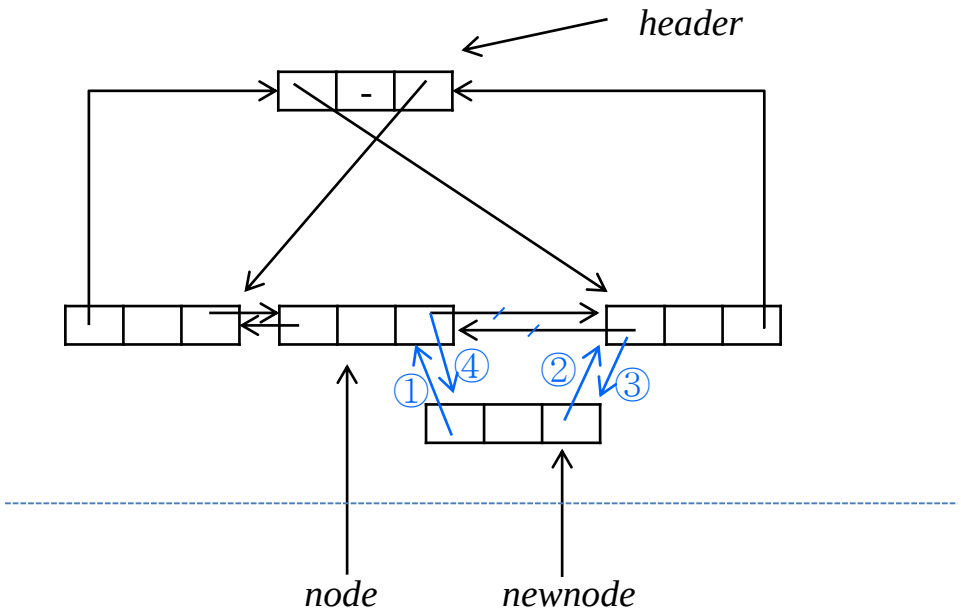
If *ptr* points to any node in a doubly linked list, then

$$ptr = ptr \rightarrow llink \rightarrow rlink = ptr \rightarrow rlink \rightarrow llink$$

This formula reflects that
we can go back and forth with equal ease.

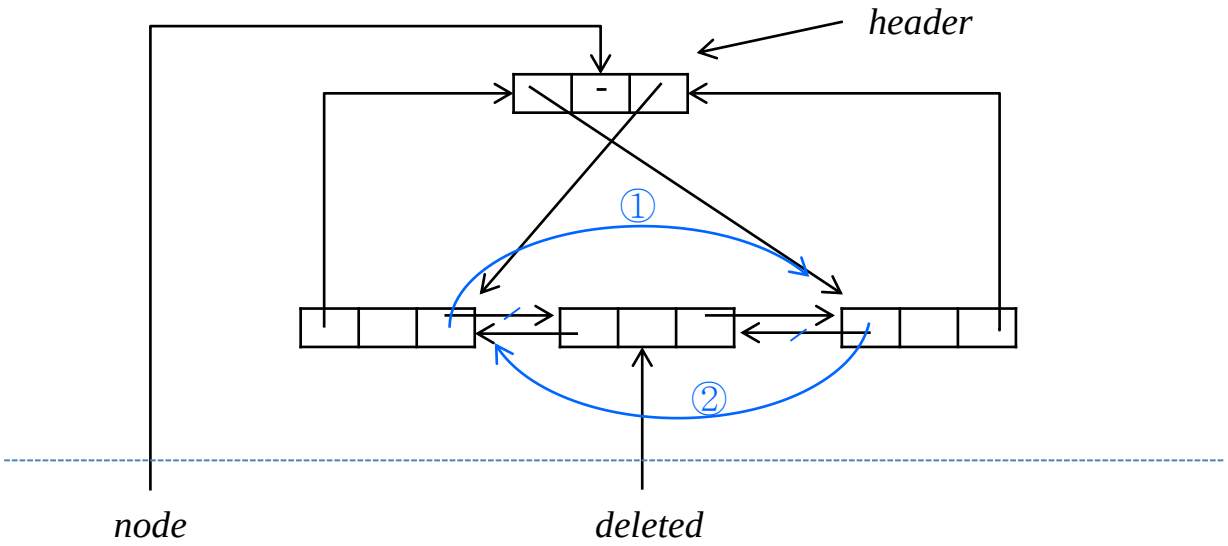
```
void dinsert(nodePointer node, nodePointer newnode)
{/* insert newnode to the right of node */
   newnode→llink = node;                    ①
   newnode→rlink = node→rlink;              ②
   node→rlink→llink = newnode;              ③
   node→rlink = newnode;                    ④
}
```

Program 4.26: Insertion into a doubly linked circular list



```
void ddelete(nodePointer node, nodePointer deleted)
{/* delete from the doubly linked list */
   if (node == deleted)
      printf("Deletion of header node not permitted.\n");
   else {
      deleted→llink→rlink = deleted→rlink;   ①
      deleted→rlink→llink = deleted→llink;   ②
      free(deleted);                         ③
   }
}
```

Program 4.27: Deletion from a doubly linked circular list



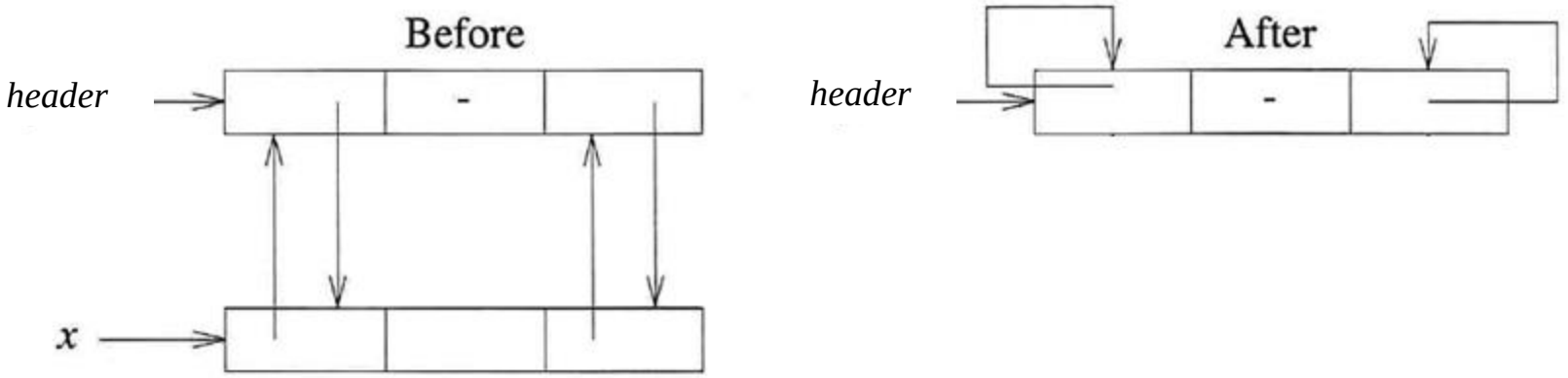


Figure 4.23: Deletion from a doubly linked circular list with a single node