Chapter 3 Lists

- Pointers
- Singly Linked Lists
- Polynomials
- Chain
- Circularly Linked Lists
- Doubly Linked Lists

Pointers (1/5)

- Consider the following alphabetized list of three letter English words ending in *at*:
  (*bat*, *cat*, *sat*, *vat*)
- If we store this list in an array
  - Add the word *mat* to this list
    - move *sat* and *vat* one position to the right before we insert *mat*.
  - Remove the word *cat* from the list
    - move *sat* and *vat* one position to the left
- Problems of a sequence representation (ordered list)
  - Arbitrary insertion and deletion from arrays can be very time-consuming
  - Waste storage

Pointers (2/5)

- An elegant solution: using linked representation
  - Items may be placed anywhere in memory.
  - In a sequential representation the order of elements is the same as in the ordered list, while in a linked representation these two sequences need not be the same.
  - Store the address, or location, of the next element in that list for accessing elements in the correct order with each element.
  - Thus, associated with each list element is a *node* which contains both a data component and a pointer to the next item in the list. The pointers are often called *links*.

Pointers (3/5)

- C provides extensive supports for pointers.
  - Two most important operators used with the pointer type :

    & the address operator

    * the dereferencing (or indirection) operator
    - **Example:**

      If we have the declaration:

      ```
      int i, *pi;
      ```

      then *i* is an integer variable and *pi* is a pointer to an integer.

      If we say:

      ```
      pi = &i;
      ```

      then &*i* returns the address of *i* and assigns it as the value of *pi*.

      To assign a value to *i* we can say:

      ```
      i = 10;
      ```
      or
      ```
      *pi = 10;
      ```

Pointers (4/5)

- Pointers can be dangerous
  - Using pointers: high degree of flexibility and efficiency, but dangerous as well.
    - It is a wise practice to set all pointers to *NULL* when they are not actually pointing to an object.
    - Another: using explicit **type cast** when converting between pointer types.
    - **Example:**

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

    - In many systems, pointers have the same size as type **int**.
      - Since **int** is the default type specifier, some programmers omit the return type when defining a function.
      - The return type defaults to **int** which can later be interpreted as a pointer.

Pointers (5/5)

- Using dynamically allocated storage
  - When programming, you may not know how much space you will need, nor do you wish to allocate some vary large area that may never be required.
    - C provides *heap*, for allocating storage at run-time.
    - You may call a function, *malloc*, and request the amount of memory you need.
    - When you no longer need an area of memory, you may free it by calling another function, *free*, and return the area of memory to the system.
  - Example:

```
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);
```

request memory

return memory

Program 4.1: Allocation and deallocation of pointers

Singly Linked Lists (1/15)

- Linked lists are drawn as an order sequence of nodes with links represented as arrows (Figure 4.1).
  - The name of the pointer to the first node in the list is the name of the list. (the list of Figure 4.1 is called *ptr*.)
    - Notice that we do not explicitly put in the values of pointers, but simply draw allows to indicate that they are there.

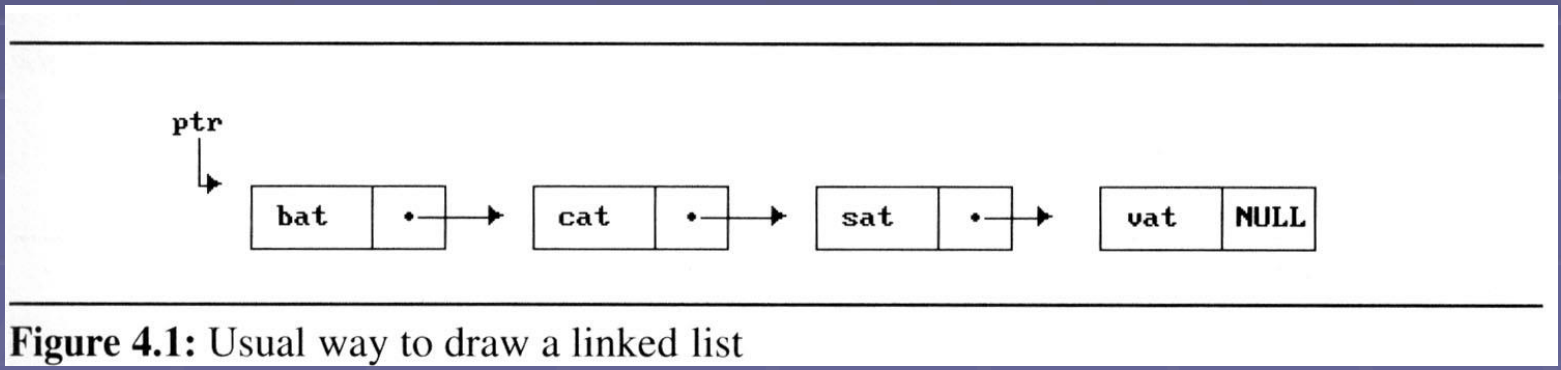
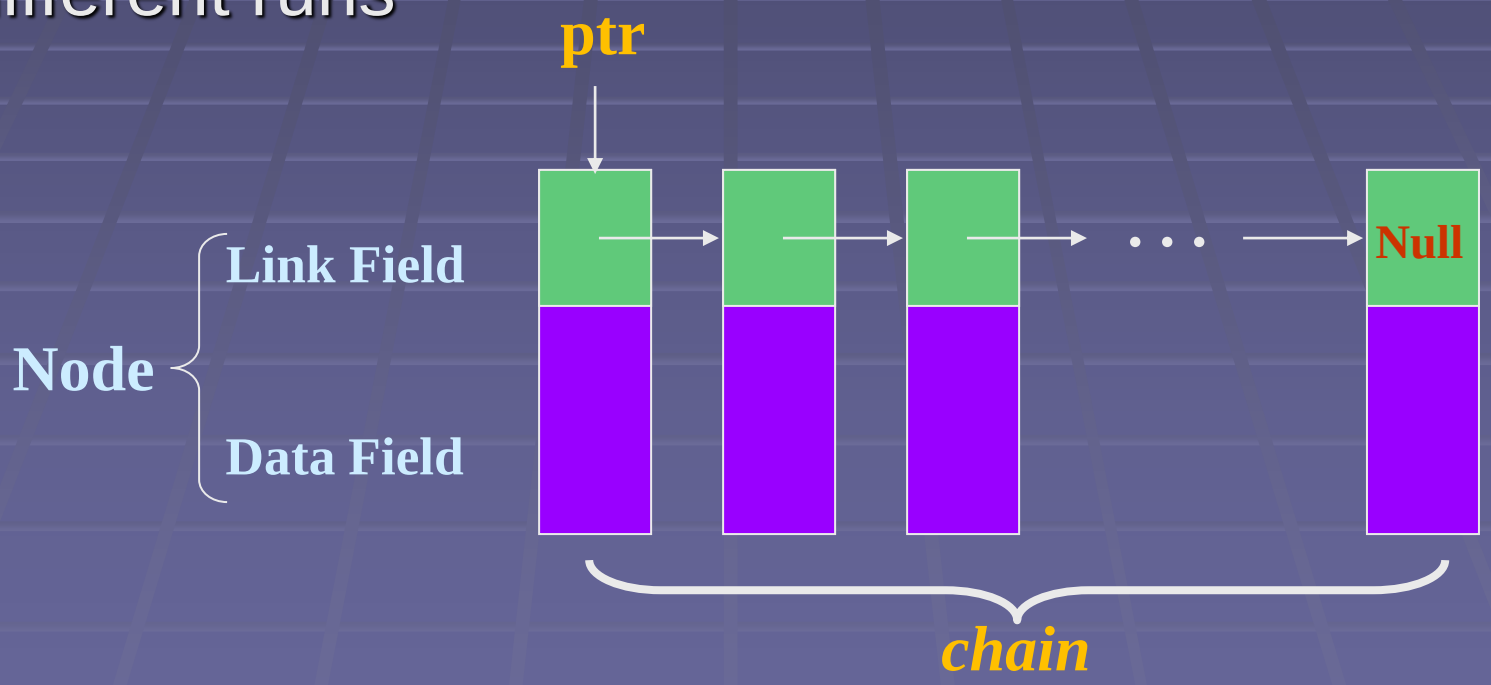


Figure 4.1: Usual way to draw a linked list

Singly Linked Lists (2/15)

- The nodes do not resident in sequential locations
- The locations of the nodes may change on different runs



Singly Linked Lists (3/15)

- Why it is easier to make arbitrary insertions and deletions using a linked list?
  - To insert the word *mat* between *cat* can *sat*, we must:
    - Get a node that is currently unused; let its address be *paddr*.
    - Set the data field of this node to *mat*.
    - Set *paddr*'s link field to point to the address found in the link field of the node containing *cat*.
    - Set the link field of the node containing *cat* to point to *paddr*.

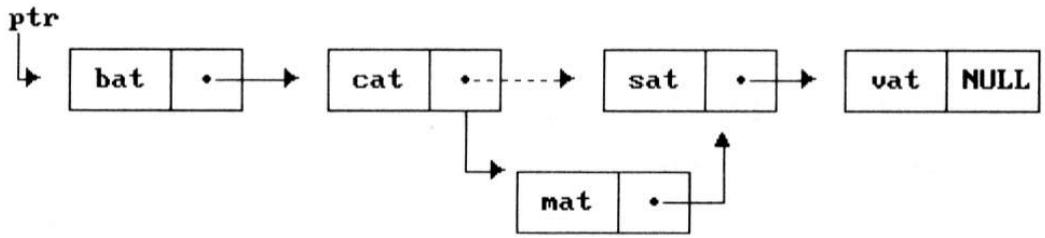


Figure 4.2: Insert *mat* after *cat*

Singly Linked Lists (4/15)

- Delete mat from the list:
  - We only need to find the element that immediately precedes mat, which is cat, and set its link field to point to mat's link (Figure 4.3).
  - We have not moved any data, and although the link field of mat still points to sat, mat is no longer in the list.

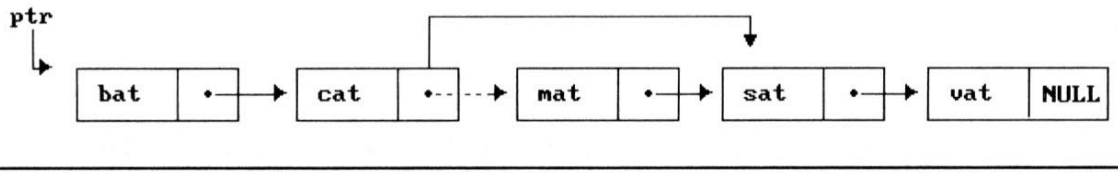


Figure 4.3: Delete *mat* from list

Singly Linked Lists (5/15)

- We need the following capabilities to make linked representations possible:
  - Defining a node's structure, that is, the fields it contains. We use *self-referential structures*, discussed in Section 2.2 to do this.
  - Create new nodes when we need them. (*malloc*)
  - Remove nodes that we no longer need. (*free*)

Singly Linked Lists (6/15)

- 2.2.4 Self-Referential Structures
  - One or more of its components is a pointer to itself.
  - typedef struct list {
    char data;
    list *link;
    }
  - list item1, item2, item3;
    item1.data='a';
    item2.data='b';
    item3.data='c';
    item1.link=item2.link=item3.link=NULL;

Construct a list with three nodes
item1.link=&item2;
item2.link=&item3;
malloc: obtain a node (memory)
free: release memory

Singly Linked Lists (7/15)

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

```
typedef struct list_node *list_pointer;
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));
```

  - Return the spaces:

```
free(ptr);
```

Singly Linked Lists (8/15)

e -> name ⇔ (*e).name

strcpy(ptr -> data, “bat”);

ptr -> link = NULL;

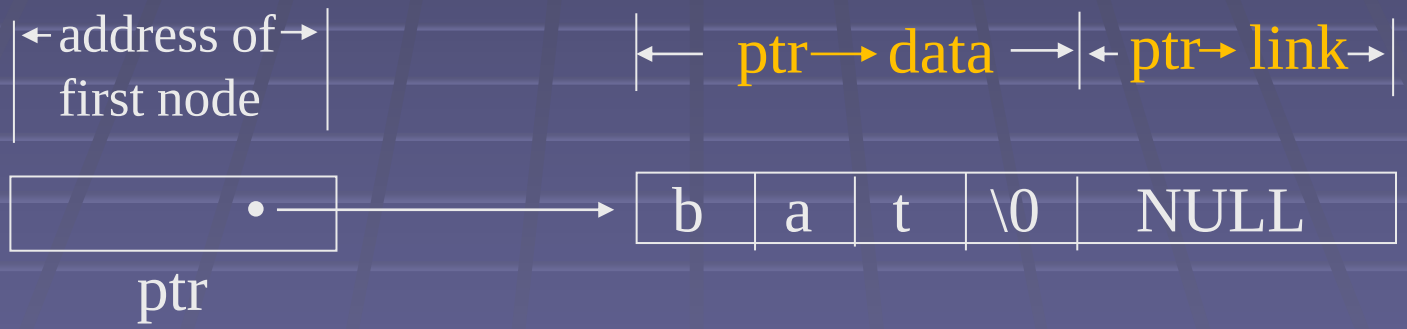


Figure 4.4: Referencing the fields of a node(p.142)

Singly Linked Lists (9/15)

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

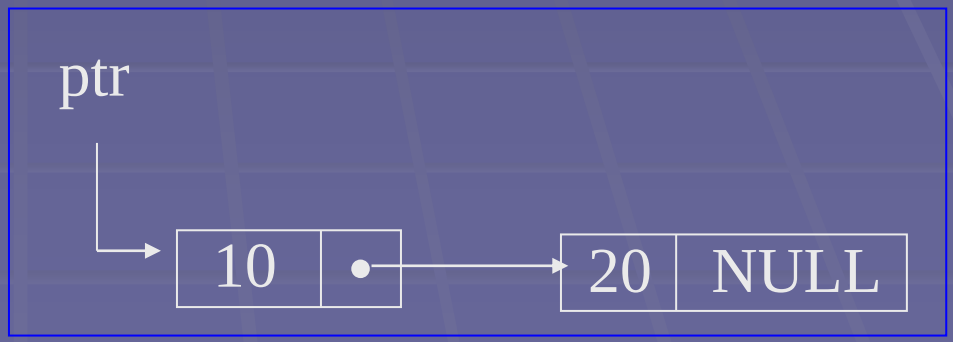
```
typedef struct list_node *list_pointer;
typedef struct list_node {
    int data;
    list_pointer link;
};
list_pointer ptr =NULL;
```

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

```
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;
}
```

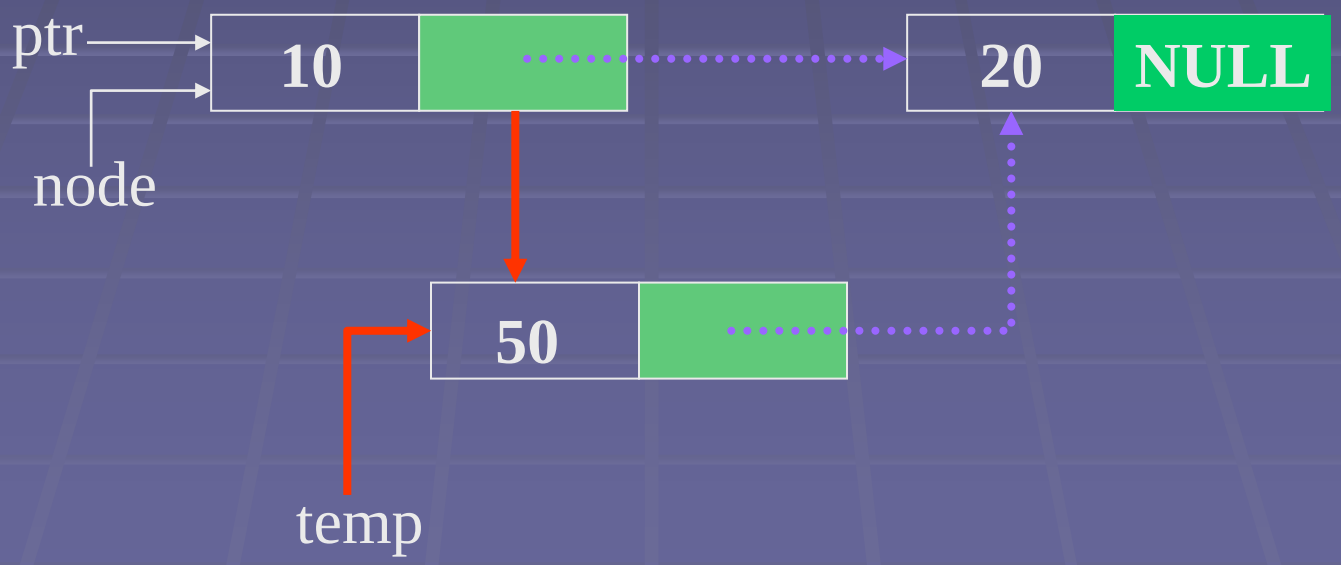
```
#define IS_FULL(ptr) (!(ptr))
```

When returns *NULL* if there is no more memory.



Singly Linked Lists (10/15)

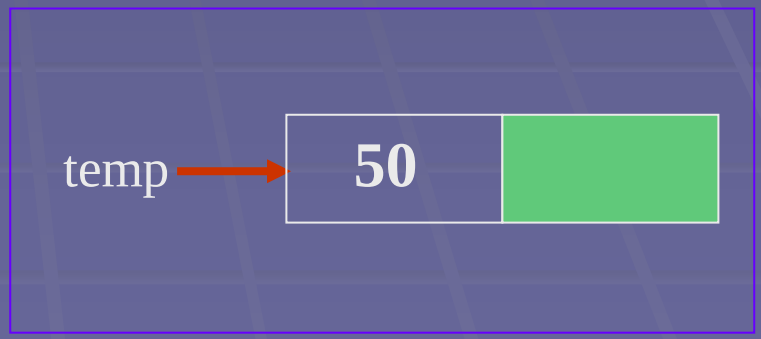
- Insertion
  - Observation
    - insert a new node with data = 50 into the list ptr after node



Singly Linked Lists (11/15)

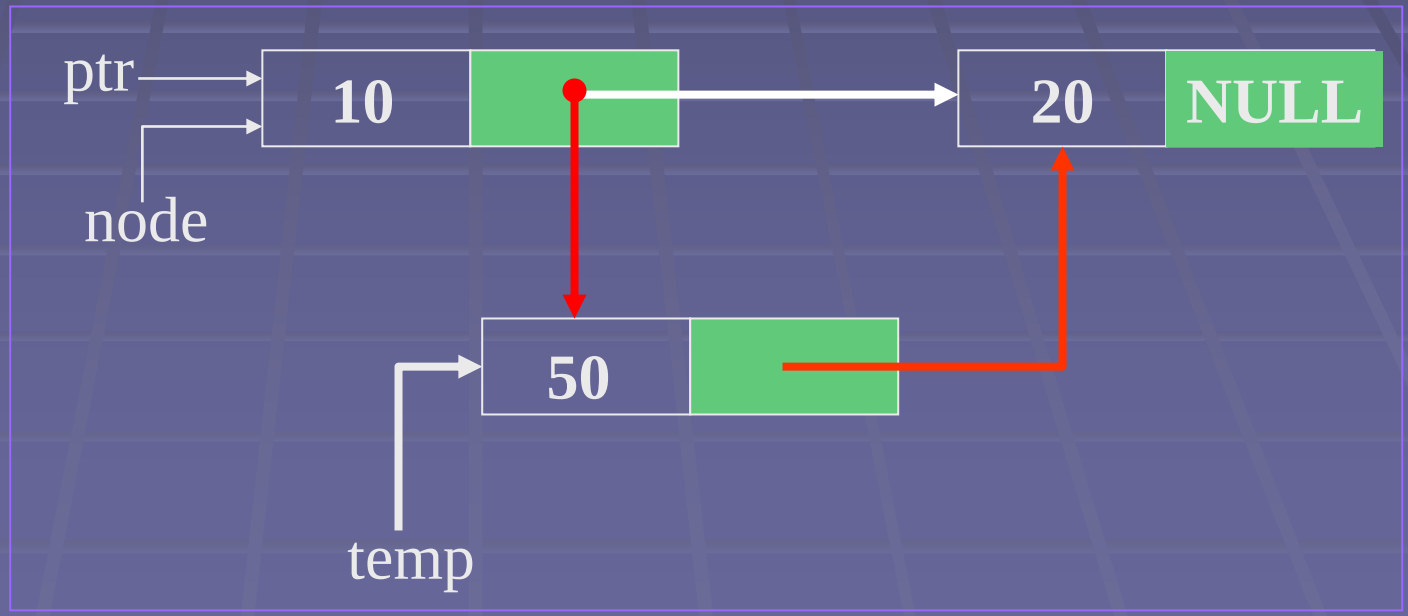
- Implement Insertion:

```
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;
```



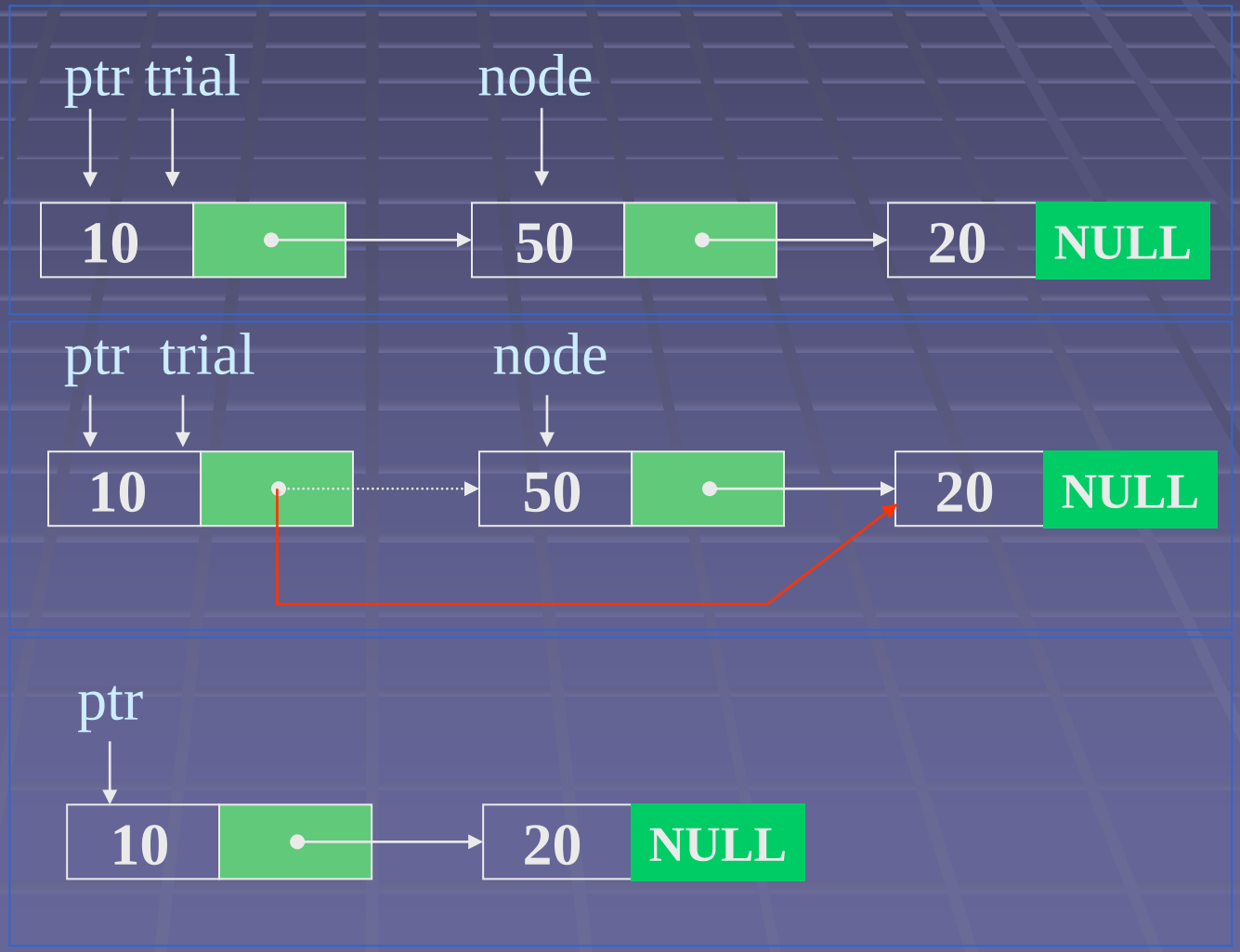
Singly Linked Lists (12/15)

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



Singly Linked Lists (13/15)

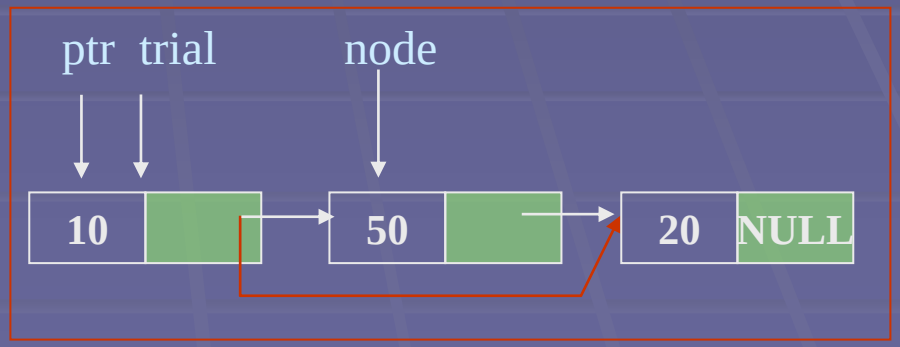
- Deletion
  - Observation: delete node from the list



Singly Linked Lists (14/15)

- Implement Deletion:

```
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);
}
```



Singly Linked Lists (15/15)

- Print out a list (traverse a list)
  - **Program 4.5:** Printing a list

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

Polynomials (1/9)

- Representing Polynomials As Singly Linked Lists
  - The manipulation of symbolic polynomials, has a classic example of list processing.
  - In general, we want to represent the polynomial:

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

    - Where the a_i are nonzero coefficients and the e_i are nonnegative integer exponents such that

      $e_{m-1} > e_{m-2} > \ldots > e_1 > e_0 \geqq 0$.
  - We will represent each term as a node containing coefficient and exponent fields, as well as a pointer to the next term.

Polynomials (2/9)

- Assuming that the coefficients are integers, the type declarations are:

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

- Draw *poly_nodes* as:

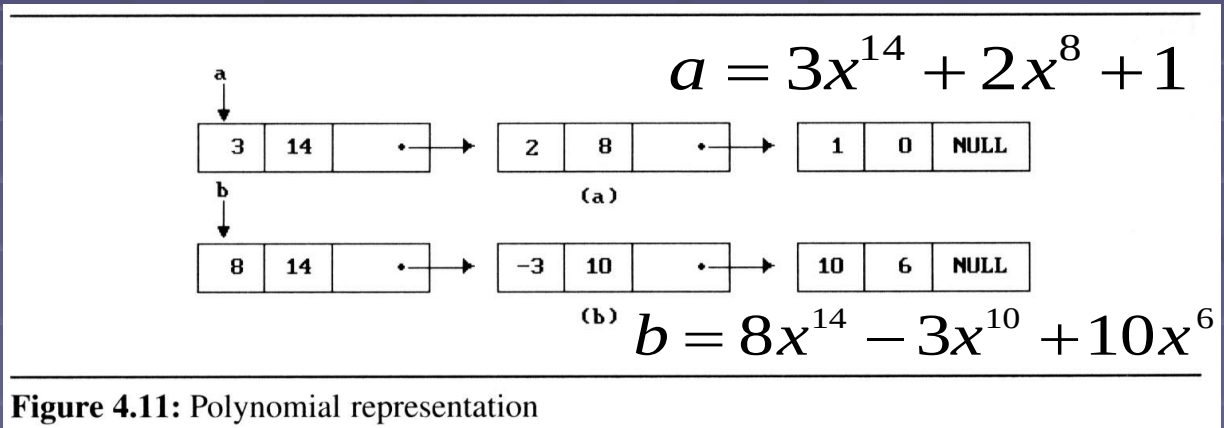


Figure 4.11: Polynomial representation

coef	expon	link

Polynomials (3/9)

- Adding Polynomials
  - To add two polynomials,we examine their terms starting at the nodes pointed to by *a* and *b*.
    - If the exponents of the two terms are equal
      1. add the two coefficients
      2. create a new term for the result.
    - If the exponent of the current term in *a* is less than *b*
      1. create a duplicate term of *b*
      2. attach this term to the result, called *d*
      3. advance the pointer to the next term in *b*.
    - We take a similar action on *a* if *a->expon* > *b->expon.*
  - Figure 4.12 generating the first three term of *d* = *a*+*b* (next page)

Polynomials (4/9)

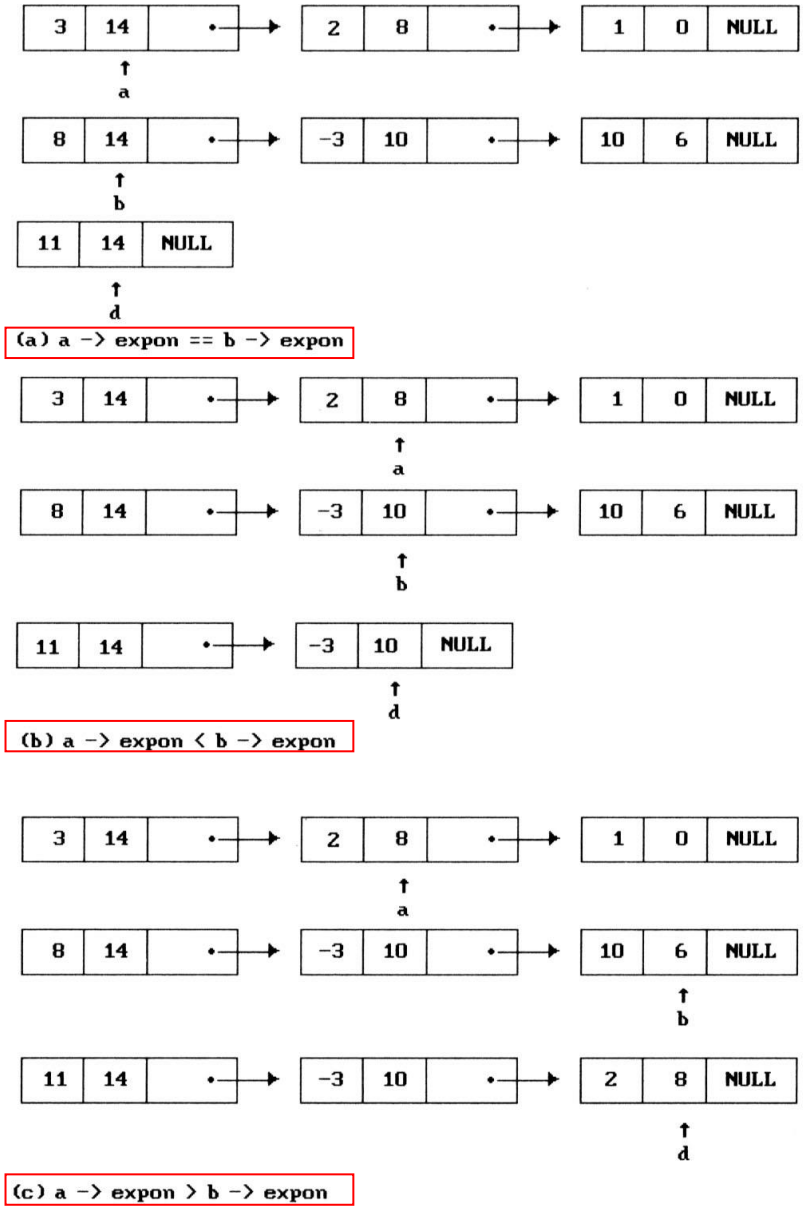


Figure 4.12: Generating the first three terms of $d = a + b$

Polynomials (5/9)

- Add two polynomials

```
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;
   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;
      }
   /* 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 = front; front = front->link;  free(temp);
   return front;
}
```

Program 4.10: Add two polynomials

Polynomials (6/9)

- Attach a node to the end of a list

```
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));
    /* create new node */
    if (IS_FULL(temp)) {
      fprintf(stderr, “The memory is full\n”);
      exit(1);
    }
    temp->coef = coefficient;  /* copy item to the new node */
    temp->expon = exponent;
    (*ptr)->link = temp;           /* attach */
    *ptr = temp;                   /* move ptr to the end of the list */
}
```

Polynomials (7/9)

- Analysis of padd

$$A(x)(=a_{m-1}x^{e_{m-1}}+\cdots+a_0x^{e_0})+B(x)(=b_{n-1}x^{f_{n-1}}+\cdots+b_0x^{f_0})$$

1. coefficient additions
   $0 \le$ additions $\le min(m, n)$
   where m (n) denotes the number of terms in A (B).
2. exponent comparisons
   extreme case:
   $e_{m-1} > f_{m-1} > e_{m-2} > f_{m-2} > \ldots > e_1 > f_1 > e_0 > f_0$
   $m+n-1$ comparisons
3. creation of new nodes
   extreme case: maximum number of terms in d is $m+n$
   $m + n$ new nodes
   summary: $O(m+n)$

Polynomials (8/9)

- 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

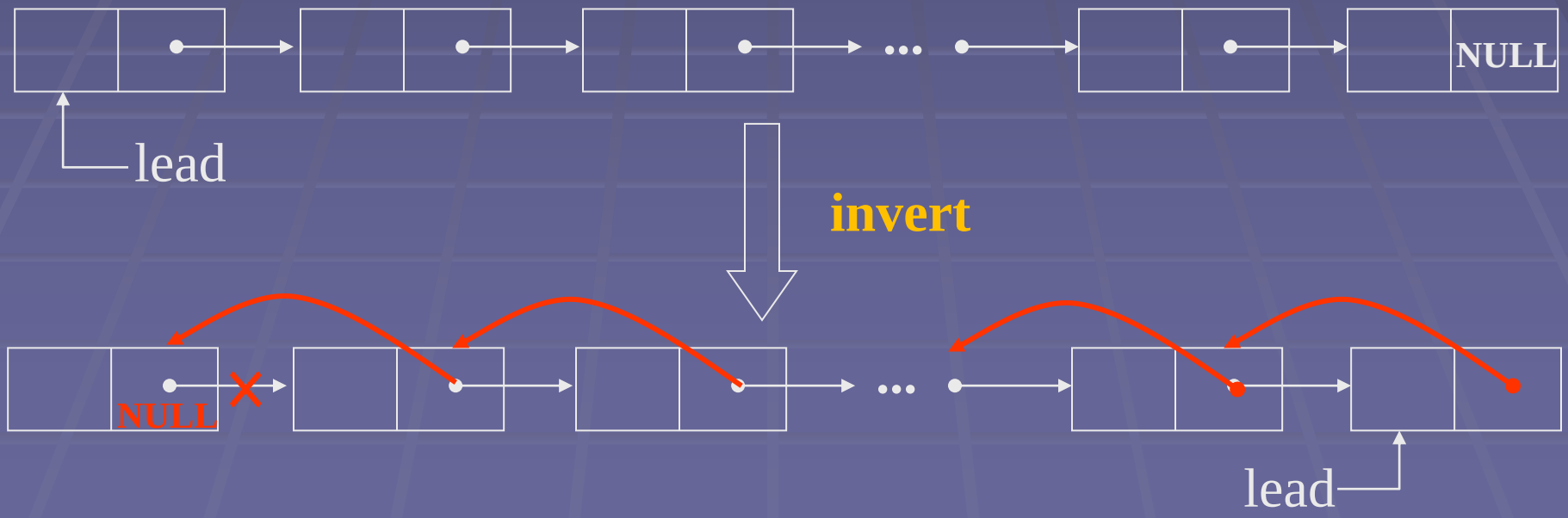
Polynomials (9/9)

- Erase Polynomials
  - erase frees the nodes in *temp*

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

Chain (1/3)

- Chain:
  - A singly linked list in which the last node has a null link
- Operations for chains
  - Inverting a chain
    - For a list of *length* ≧1 nodes, the **while** loop is executed *length* times and so the computing time is linear or O(*length*).

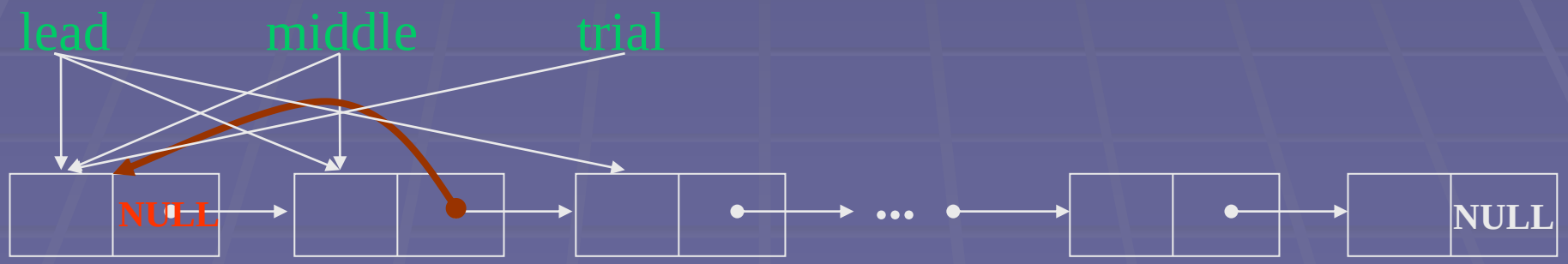


Chain (2/3)

```
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;
}
```

Two extra pointers

Program 4.17: Inverting a singly linked list



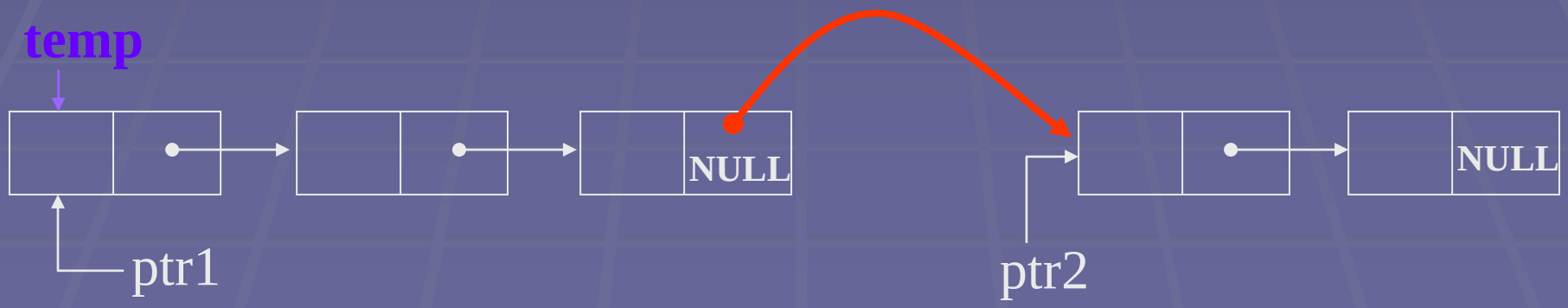
Chain (3/3)

- Concatenates two chains
  - Concatenates two chains, ptr1 and ptr2.
  - Assign the list ptr1 followed by the list ptr2.

```
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;
   }
}
```

O(length of list *ptr1*)

Program 4.18: Concatenating singly linked lists

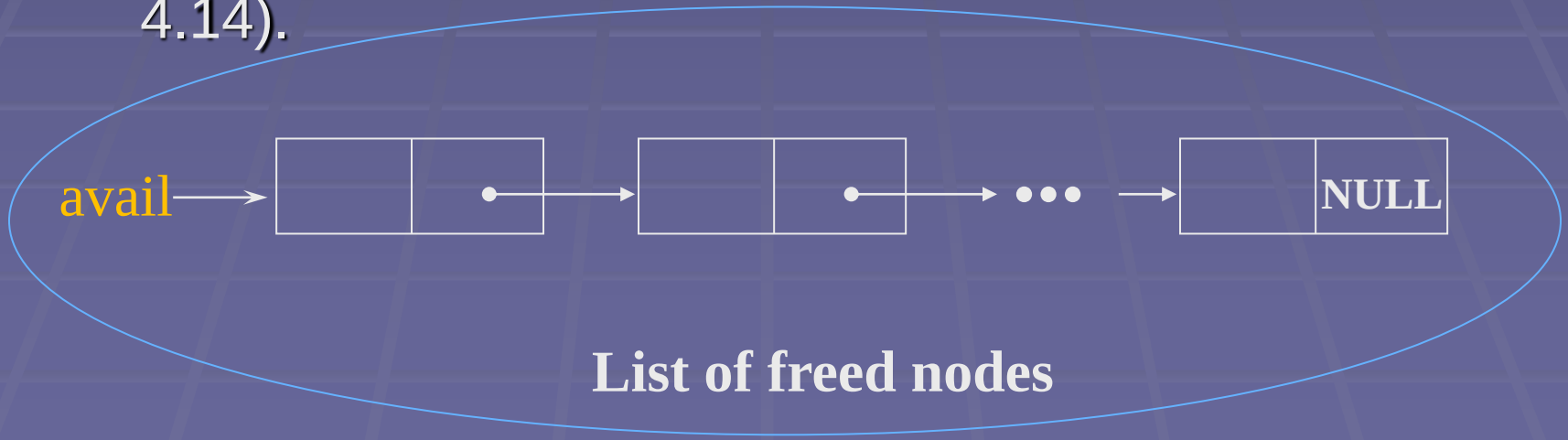


Circularly Linked Lists (1/10)

- Circular Linked list
  - The link field of the last node points to the first node in the list.
- Example
  - Represent a polynomial *ptr* = $3x^{14}+2x^{8}+1$ as a circularly linked list.

Circularly Linked Lists (2/10)

- Maintain an Available List
  - We 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”.
  - Instead of using ***malloc*** and ***free***, we now use get_node (program 4.13) and ret_node (program 4.14).



Circularly Linked Lists (3/10)

- Maintain an Available List (cont'd)
  - 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 we do need to use ***malloc*** to create a new node.

```
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 4.13: *get_node* function

Circularly Linked Lists (4/10)

- Maintain an Available List (cont'd)
- Insert ptr to the front of this list
  - Let ***avail*** be a variable of type poly_pointer that points to the first node in our list of freed nodes.
  - Henceforth, we call this list the available space list or ***avail*** list.
  - Initially, we set ***avail*** to ***NULL***

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

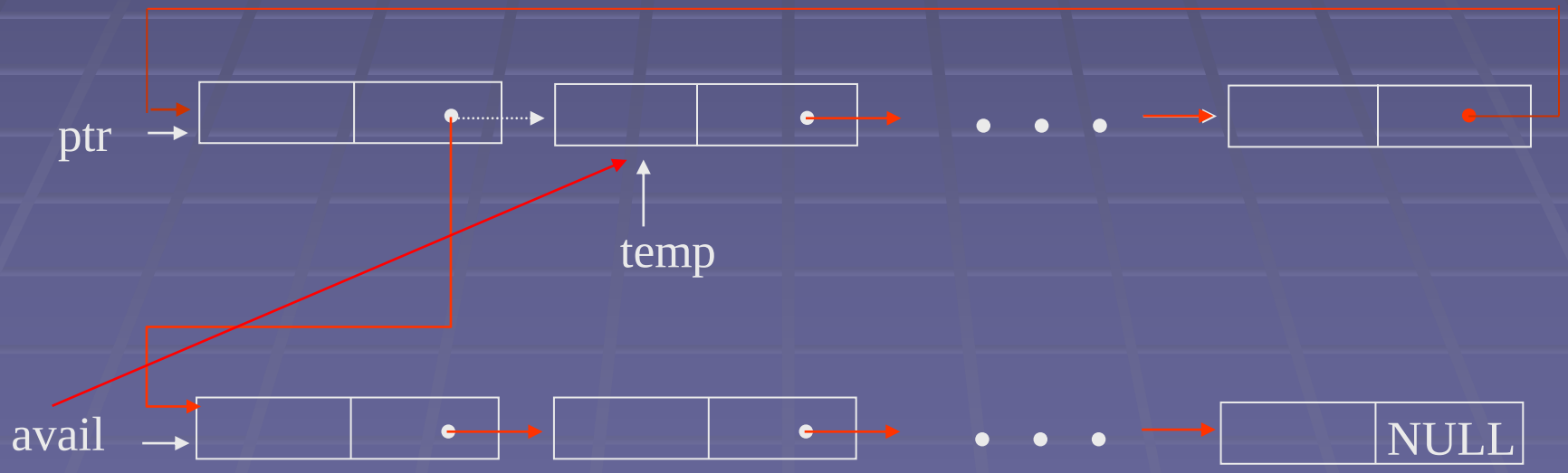
Program 4.14: *ret_node* function

Circularly Linked Lists (5/10)

- Maintain an Available List
  - Erase a circular list in a fixed amount (constant) of time *O*(1) independent of the number of nodes in the list using ***cerase***

```
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;
   }
}
```

Program 4.15: Erasing a circular list



紅色link所連接而成的 chain

Circularly Linked Lists (6/10)

- We must handle the **zero polynomial** as a special case. To avoid it, we introduce a ***head node*** into each polynomial
  - each polynomial, zero or nonzero, contains one additional node.
  - The *expon* and *coef* fields of this node are irrelevant.

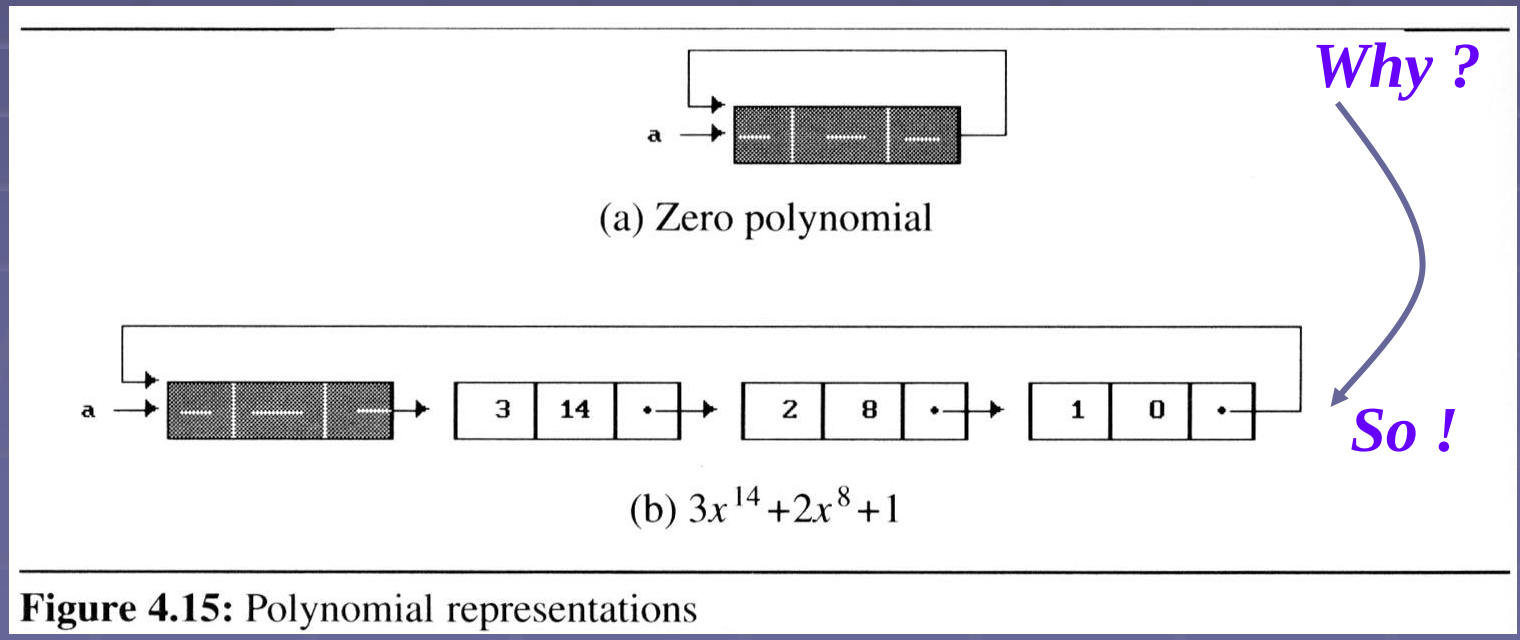


Figure 4.15: Polynomial representations

Circularly Linked Lists (7/10)

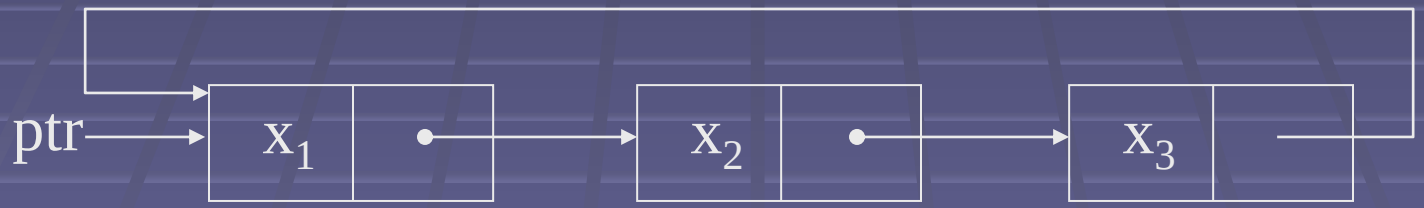
- For fit the circular list with head node representation
  - We may remove the test for (*ptr) from *cerase*
  - Changes the original *padd* to *cpadd*

```
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; /* head node */
   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 = TRUE;
                  else { /*a->expon=-1, so b->expont > -1 */
                     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; /* link to the first node */
   return d;
}
```

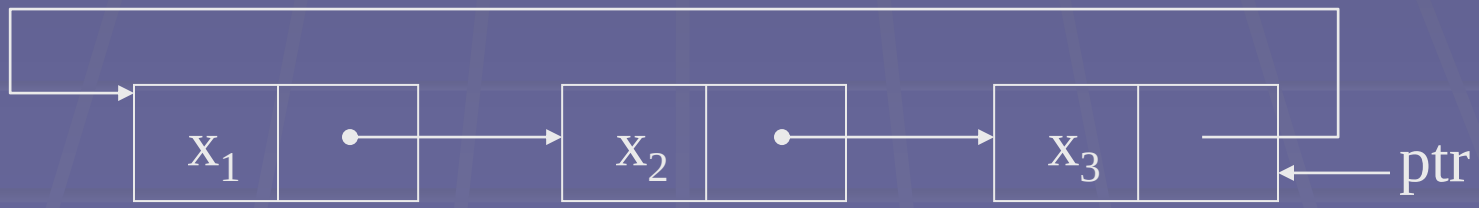
Program 4.16: Adding circularly represented polynomials

Circularly Linked Lists (8/10)

- Operations for circularly linked lists
  - **Question:**
    - What happens when we want to insert a new node at the front of the circular linked list ptr?



  - **Answer:**
    - move down the entire length of ptr.
  - **Possible Solution:**

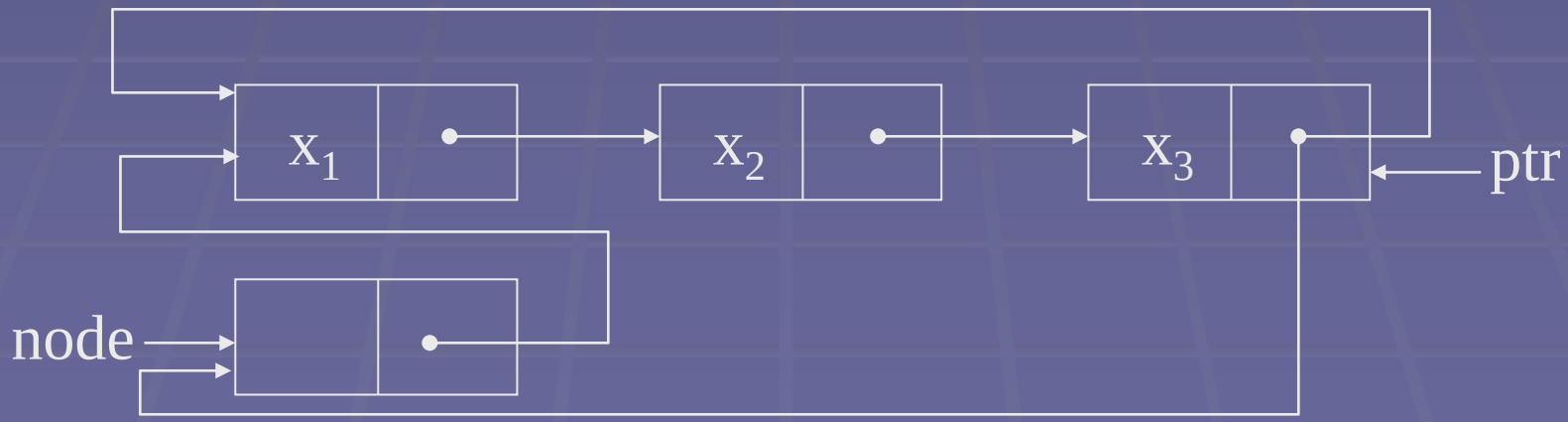


Circularly Linked Lists (9/10)

- Insert a new node at the front of a circular list
- To insert *node* at the rear, we only need to add the additional statement **ptr* = *node* to the else clause of *insert_front*

```
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;
   }
}
```

Program 4.19: Inserting at the front of a list



Circularly Linked Lists (10/10)

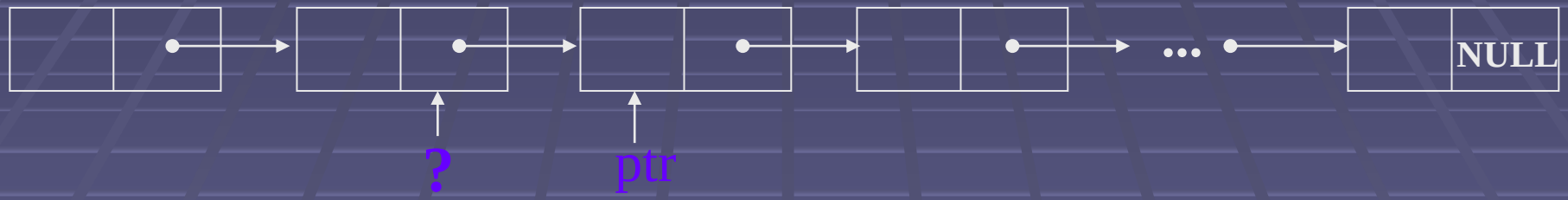
- Finding the length of a circular list

```
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;
}
```

Program 4.20: Finding the length of a circular list

Doubly Linked Lists (1/4)

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



- Doubly linked list has at least three fields
  - left link field(*llink*), data field(*item*), right link field(*rlink*).
  - The necessary declarations:

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

Doubly Linked Lists (2/4)

- Sample
  - doubly linked circular with head node: (Figure 4.23)

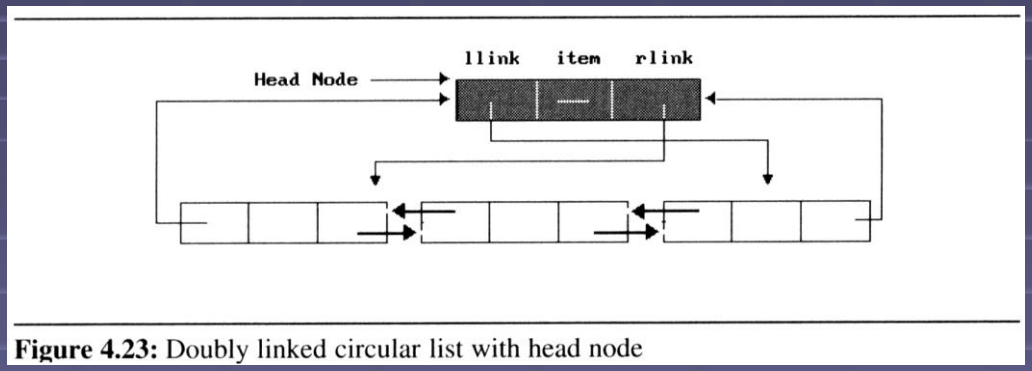


Figure 4.23: Doubly linked circular list with head node

  - empty double linked circular list with head node (Figure 4.24)

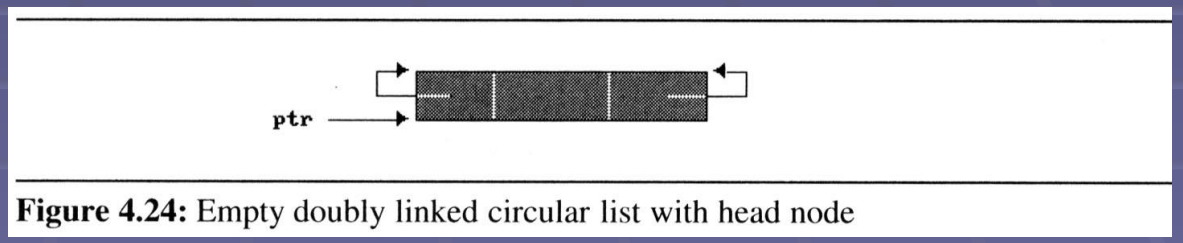


Figure 4.24: Empty doubly linked circular list with head node

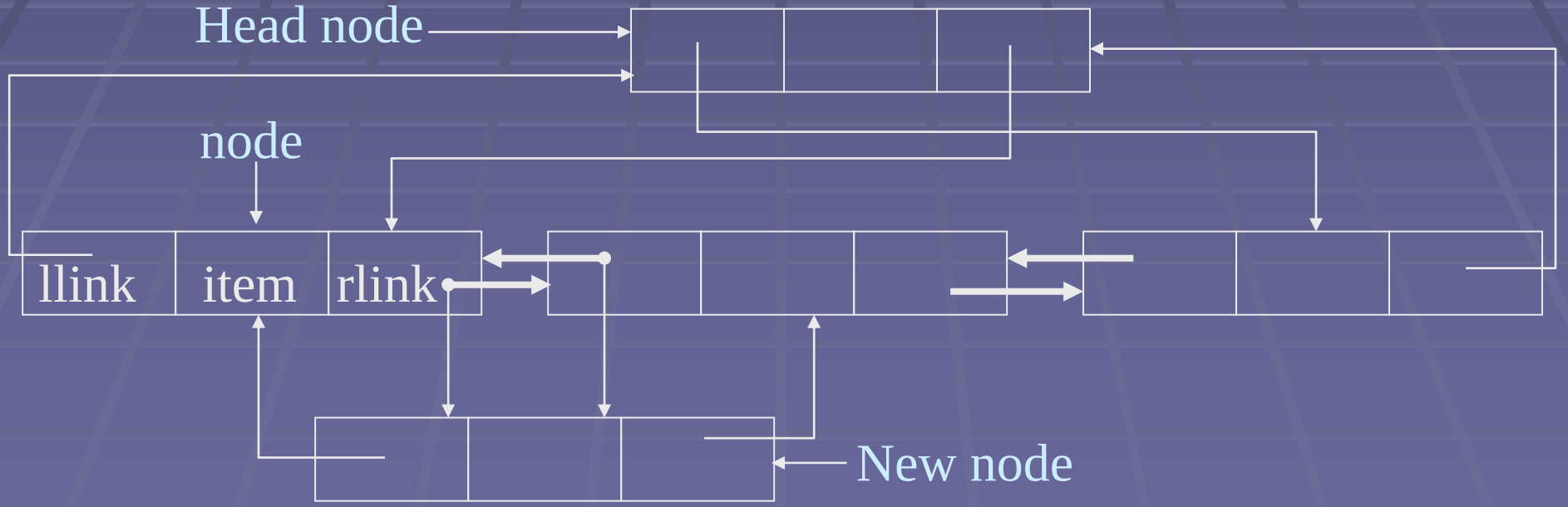
  - suppose that *ptr* points to any node in a doubly linked list, then:
    - ptr = ptr -> llink -> rlink = ptr -> rlink -> llink

Doubly Linked Lists (3/4)

- Insert node

```
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;
}
```

Program 4.28: Insertion into a doubly linked circular list



Doubly Linked Lists (4/4)

- Delete node

```
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);
   }
}
```

Program 4.29: Deletion from a doubly linked circular list