Chapter 4
Linked Lists

Yi-Fen Liu

Department of IECS, FCU

References:
- E. Horowitz, S. Sahni and S. Anderson-Freed, *Fundamentals of Data Structures (2nd Edition)*
- Slides are credited from Prof. Chung, NTHU

Outline

- Pointers
- Singly Linked Lists
- Dynamically Linked Stacks and Queues
- Polynomials
- Chain
- Circularly Linked Lists
- Equivalence Relations
- Doubly Linked Lists

POINTERS

Pointers (1)

- Consider the following alphabetized list
  - (*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)

	0	1	2	3	4	5	6	7	8	9
list:	10	20	30	40	50					

move= 5

Insert 25

	0	1	2	3	4	5	6	7	8	9
list:	10	20	25	30	40	50				

move= 4

Delete 20

	0	1	2	3	4	5	6	7	8	9
list:	10	25	30	40	50					

move = 5

Pointers (3)

- An elegant solution: using linked representation
  - Items may be placed anywhere in memory
- Store the address, or location, of the next element in that list for accessing elements in the correct order with each element
  - The 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 (4)

- C provides extensive supports for pointers
- Two most important operators used with the pointer type
  - & the address operator
  - * the dereferencing (or indirection) operator
- Example
  - `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 (5)

- Pointers can be dangerous
  - It is a wise practice to set all pointers to NULL when they are not actually pointing to an object
  - Using explicit type cast when converting between pointer types
  - Example
    - `pi = (int *) 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 (6)

- 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

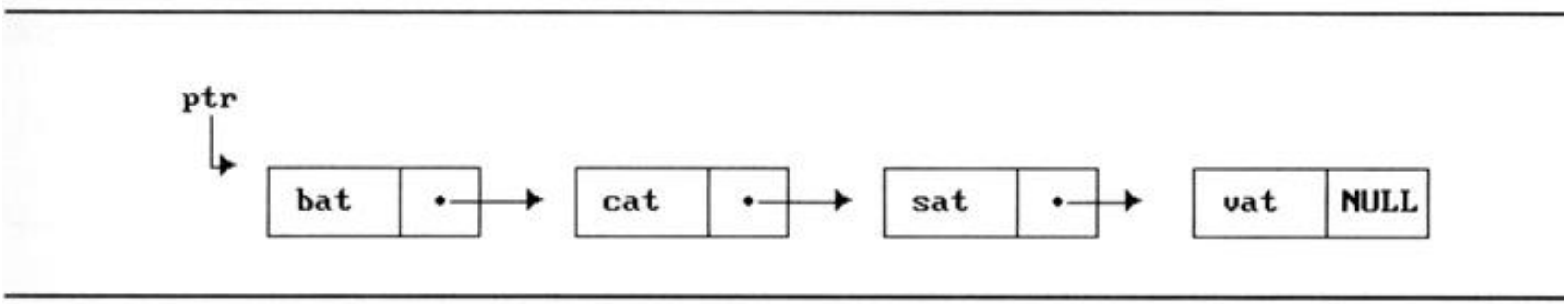
```
int i, *pi;
float f, *pf;
pi = (int *) malloc(sizeof(int));
pf = (float *) malloc(sizeof(float));
*pi = 1024;
*pf = 3.14;
printf("an integer = %d, a float = %f\n", *pi, *pf);
free(pi);
free(pf);
```

Allocation and deallocation of pointers

SINGLY LINKED LIST

Singly Linked Lists (1)

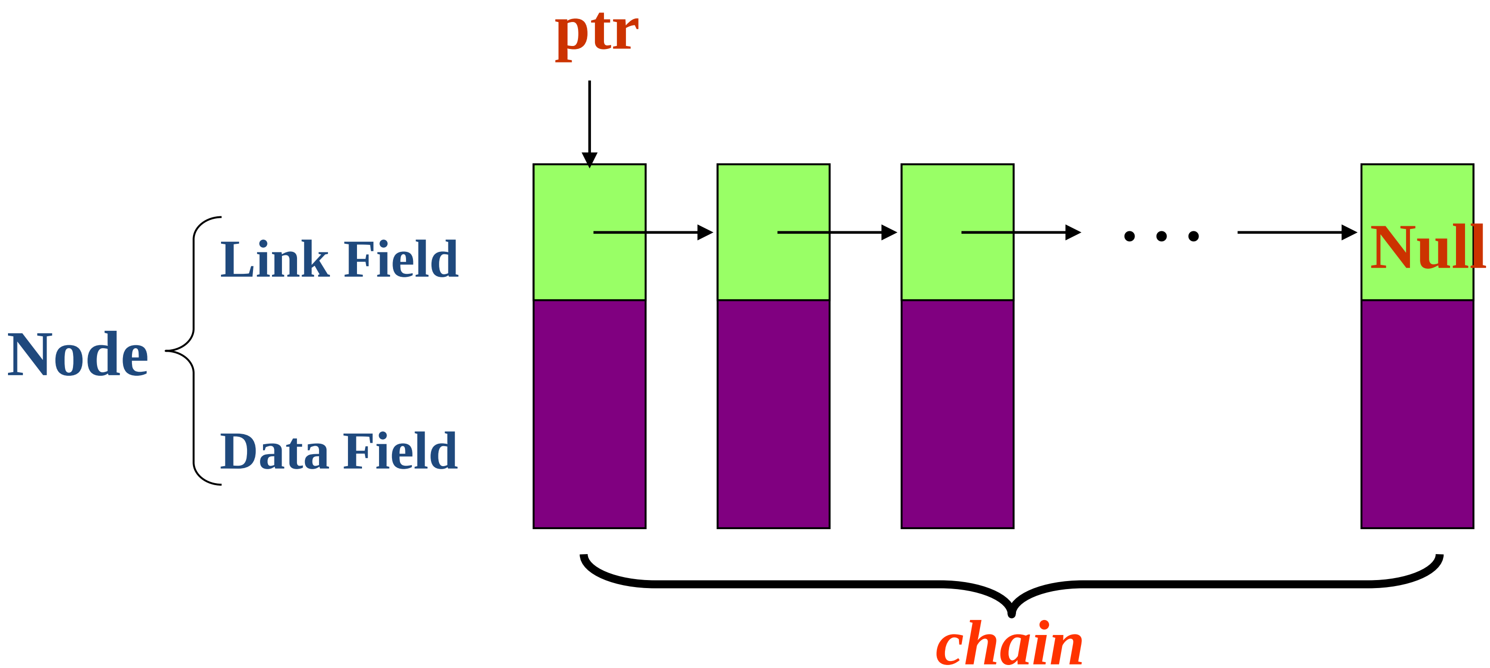
- Linked lists are drawn as an order sequence of nodes with links represented as arrows
  - The name of the pointer to the first node in the list is the name of the list



Usual way to draw a linked list

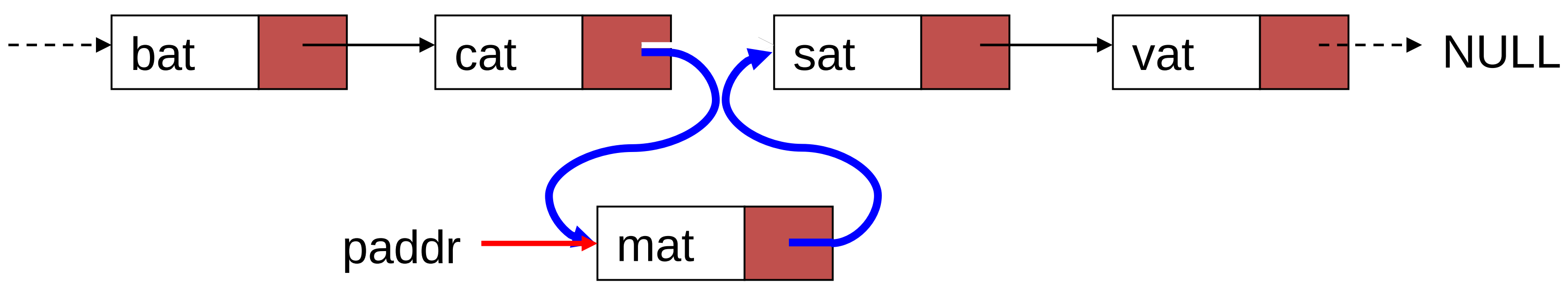
Singly Linked Lists (2)

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



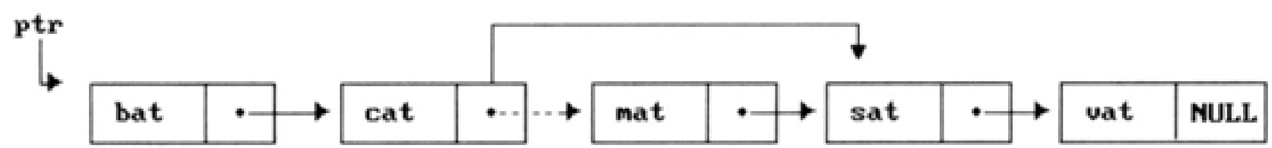
Singly Linked Lists (3)

- 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 *sat*
    - Set the link field of the node containing *cat* to point to *paddr*



Singly Linked Lists (4)

- 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 *sat's* link
  - We have not moved any data, and although the link field of mat still points to *sat*, mat is no longer in the list



Delete *mat* from list

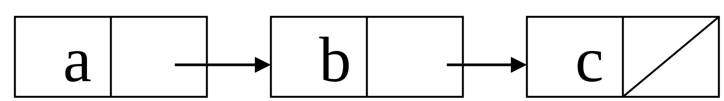
Singly Linked Lists (5)

- 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 Chapter 2 to do this
  - Create new nodes when we need them (*malloc*)
  - Remove nodes that we no longer need (*free*)

Singly Linked Lists (6)

- 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;
    item2.link=&item3;
    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)

- Example: List of words

```
typedef struct listNode *listPointer;
typedef struct listNode {
  char data [4];
  listPointer link;
};
```

  - Creation
    - `listPointer ptr = NULL;`
  - Testing
    - `#define IS_EMPTY(ptr) (!(ptr))`
  - Allocation
    - `ptr = (listPointer) malloc (sizeof(listNode));`
  - Return the spaces
    - `free(ptr);`

Singly Linked Lists (8)

- Example: Two-node linked list

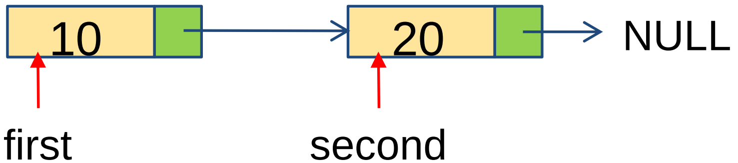
```
typedef struct listNode *listPointer;
typedef struct listNode {
   int data;
   listPointer link;
};
listPointer ptr =NULL;
```

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

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

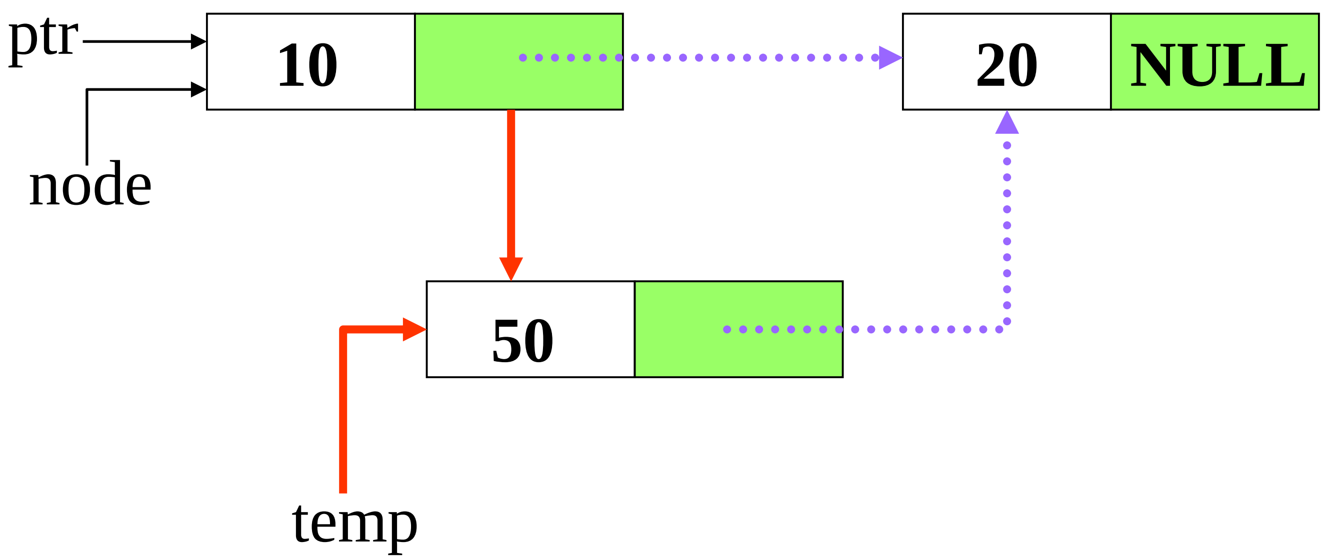
- Program : Create a two-node list

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



Singly Linked Lists (9)

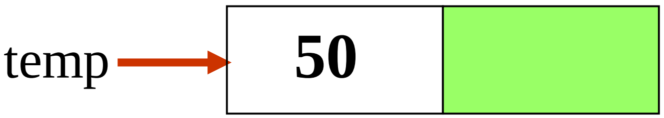
- Insertion
  - Observation
    - insert a new node with data = 50 into the list ptr after node



Singly Linked Lists (10)

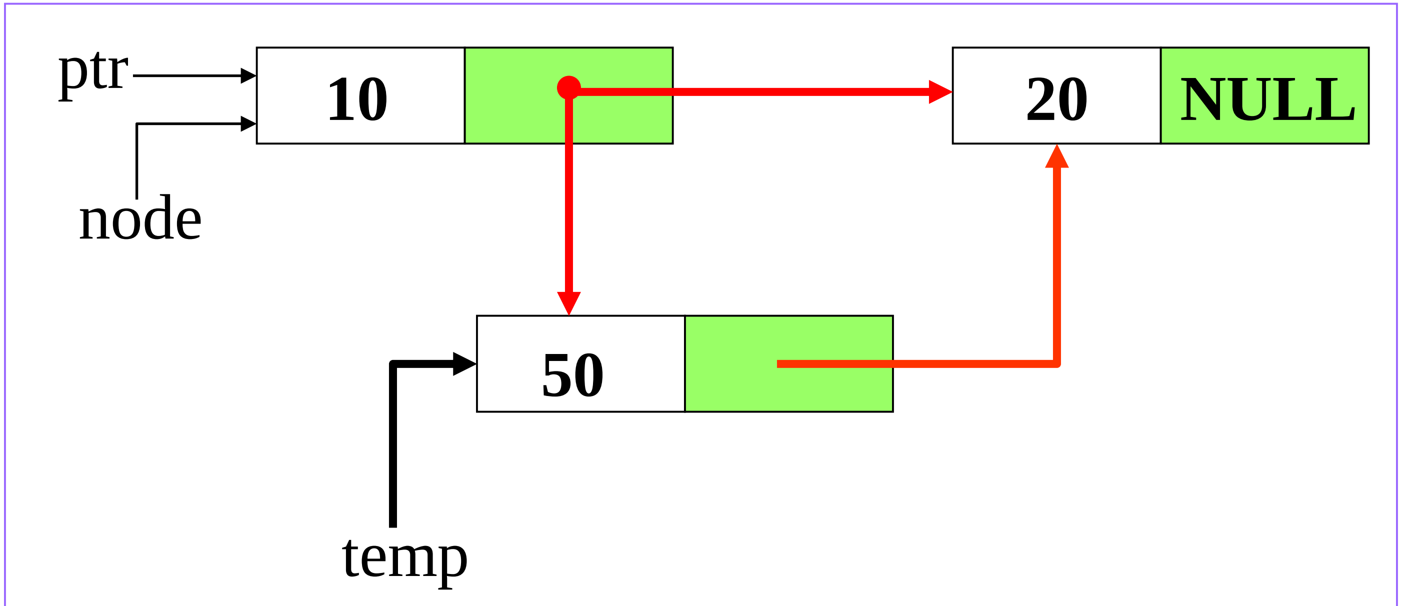
- Implement Insertion

```
void insert(listPointer *ptr, listPointer node)
{
/* insert a new node with data = 50 into the list
  ptr after node */
  listPointer temp;
  temp=(listPointer)malloc(sizeof(listNode));
  if(IS_FULL(temp)){
   fprintf(stderr, “The memory is full\n”);
   exit(1);
  }
  temp->data=50;
```



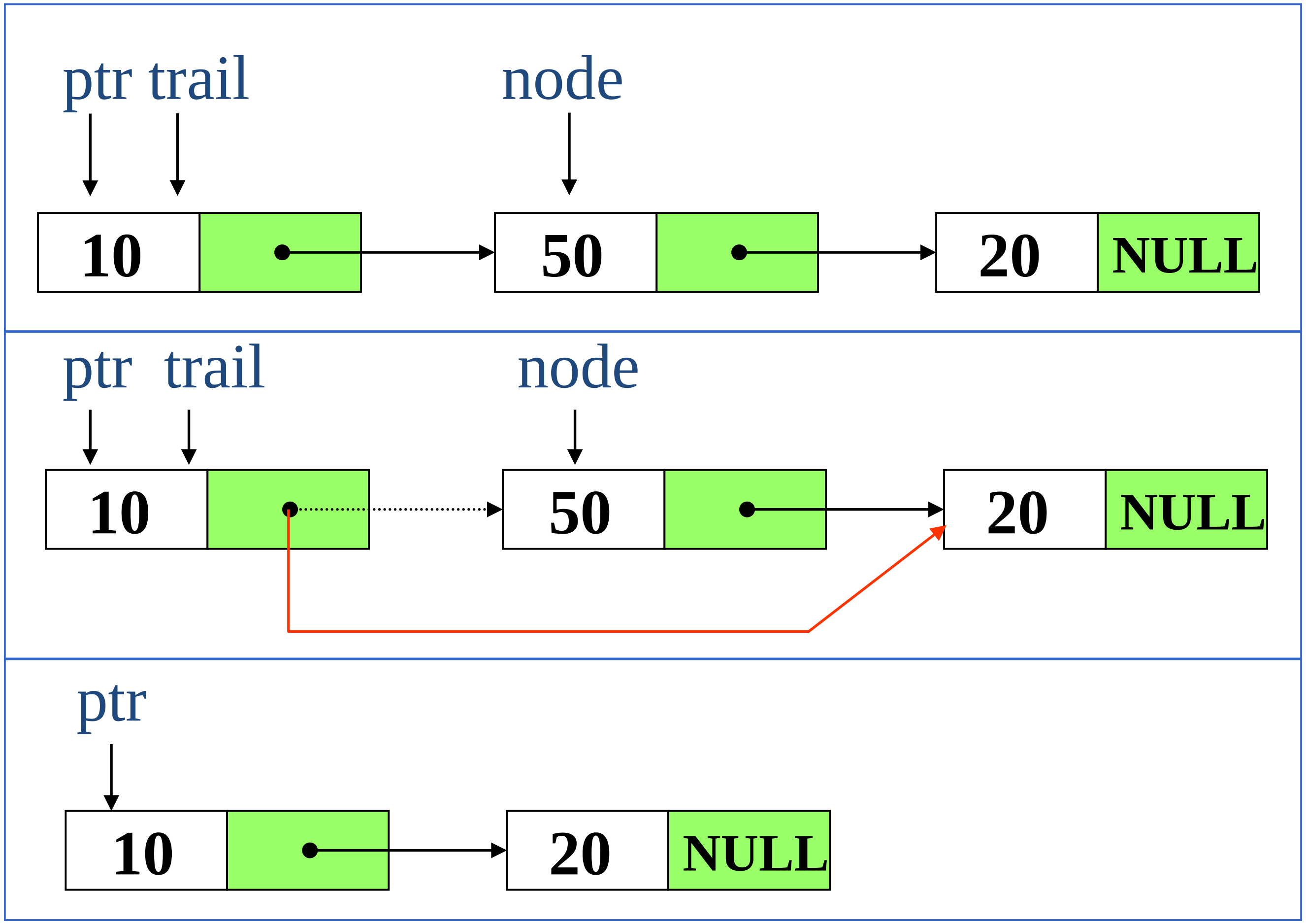
Singly Linked Lists (11)

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



Singly Linked Lists (12)

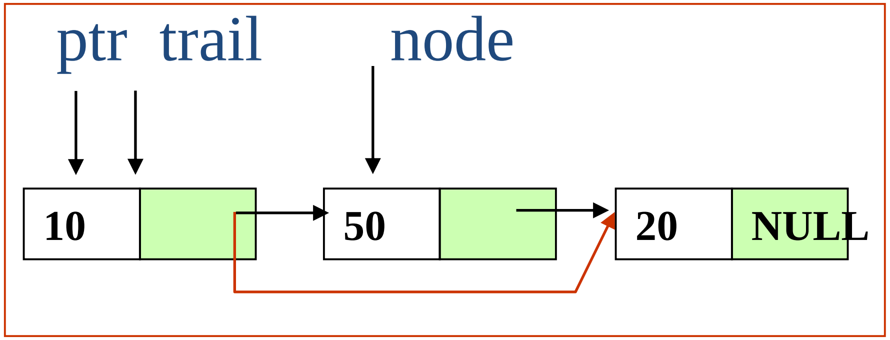
- Deletion
  - Observation: delete node from the list



Singly Linked Lists (13)

- Implement Deletion:

```
void delete(listPointer *ptr, listPointer trail,
  listPointer 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 (14)

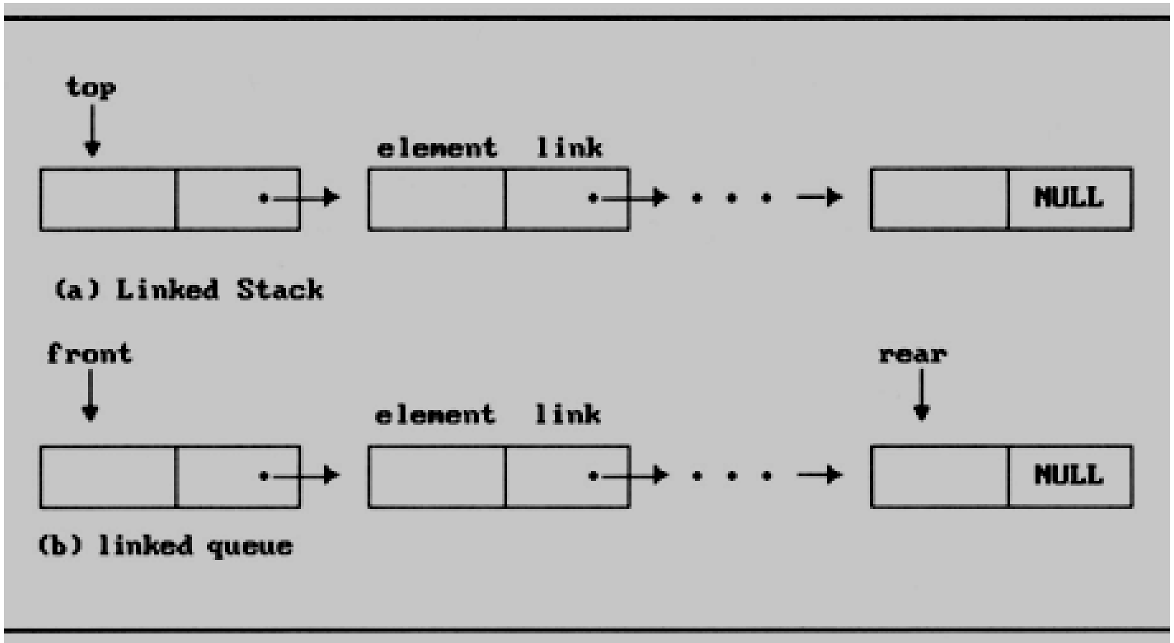
- Print out a list (traverse a list)
  - Program: Printing a list

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

DYNAMICALLY LINKED STACKS AND QUEUES

Dynamically Linked Stacks and Queues

- Both stack and the queue are easy insertion and deletion of nodes by using link list
  - Easily add or delete a node form the top of the stack
  - Easily add a node to the rear of the queue and add or delete a node at the front of a queue

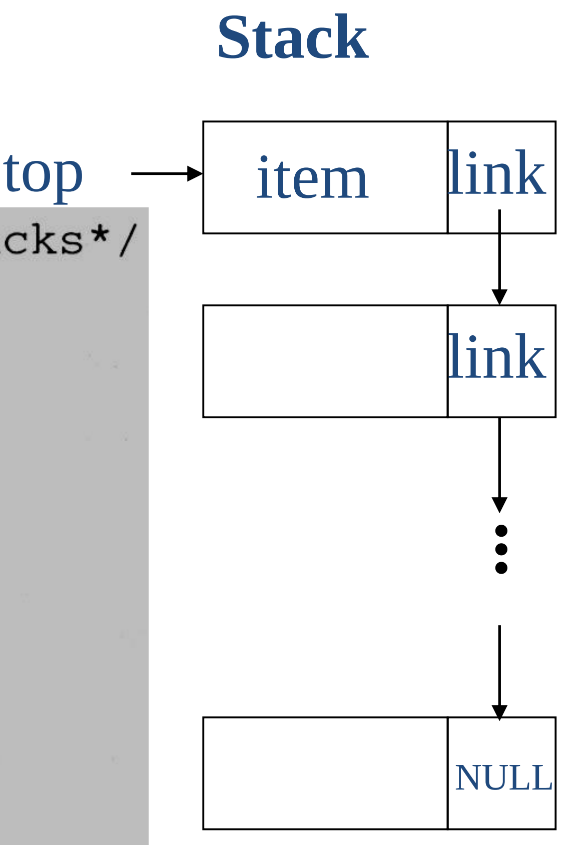


Linked stack and queue

Dynamically Linked Stacks (1)

- Represent *n* stacks

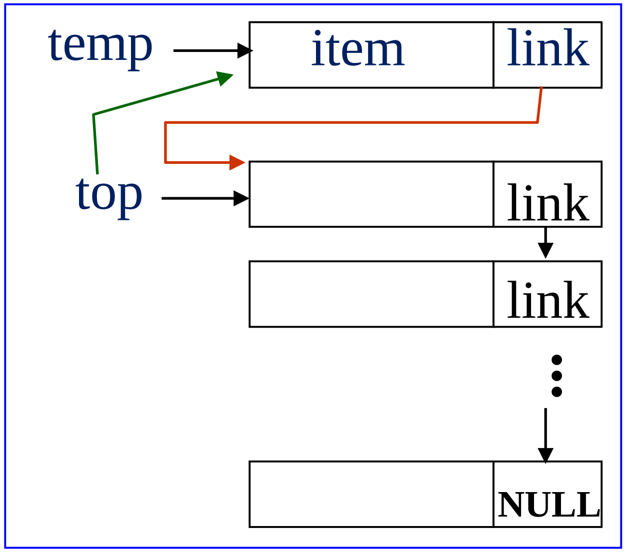
```
#define MAX_STACKS 10 /*maximum number of stacks*/
typedef struct {
        int key;
        /* other fields */
        } element;
typedef struct stack *stack_pointer;
typedef struct stack {
        element item;
        stack_pointer link;
        };
stack_pointer top[MAX_STACKS];
```



Dynamically Linked Stacks (2)

- Push in the linked stack

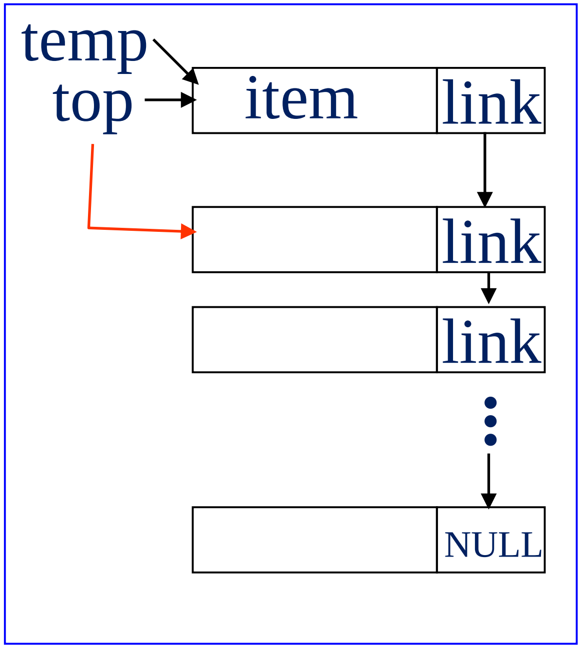
```
void add(stack_pointer *top, element item){
  /* add an element to the top of the stack */
  stack_pointer temp = (stack_pointer) malloc
  (sizeof (stack));
  if (IS_FULL(temp)) {
   fprintf(stderr, " The memory is full\n");
   exit(1);
  }
  temp->item = item;
  temp->link = *top;
  *top= temp;
}
```



Dynamically Linked Stacks (3)

- Pop from the linked stack

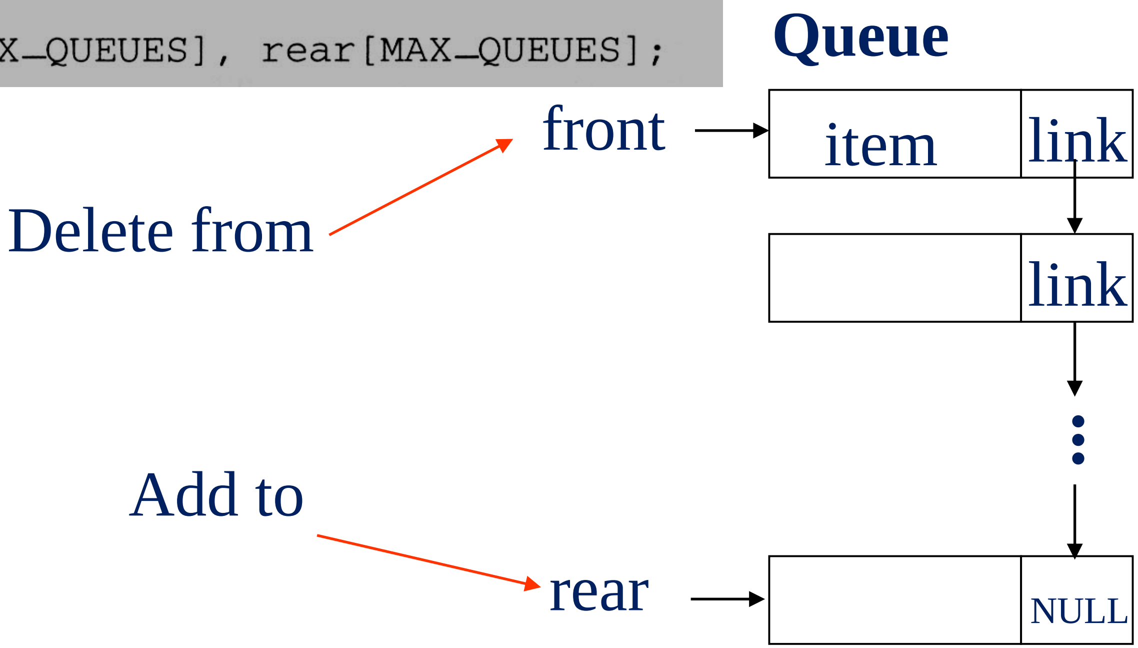
```
element delete(stack_pointer *top) {
/* delete an element from the stack */
  stack_pointer temp = *top;
  element item;
  if (IS_EMPTY(temp))  {
   fprintf(stderr,  "The stack
    is empty\n");
   exit(1);
  }
  item = temp->item;
  *top = temp->link;
  free(temp);
  return item;
}
```



Dynamically Linked Queues (1)

- 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_QUEUES], rear[MAX_QUEUES];
```

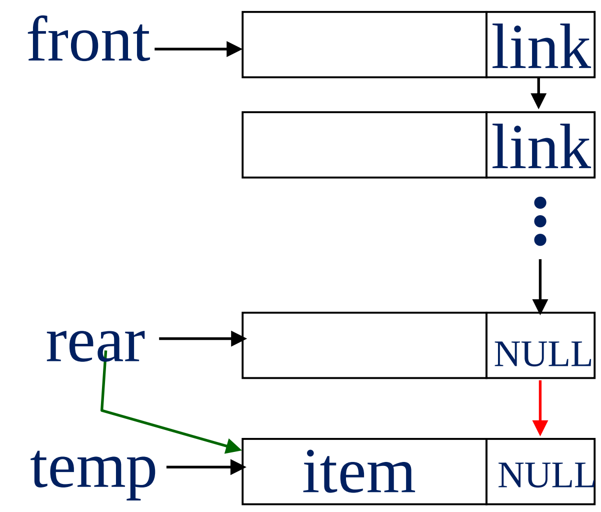


Dynamically Linked Queues (2)

- 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;
}
```

Add to the rear of a linked queue

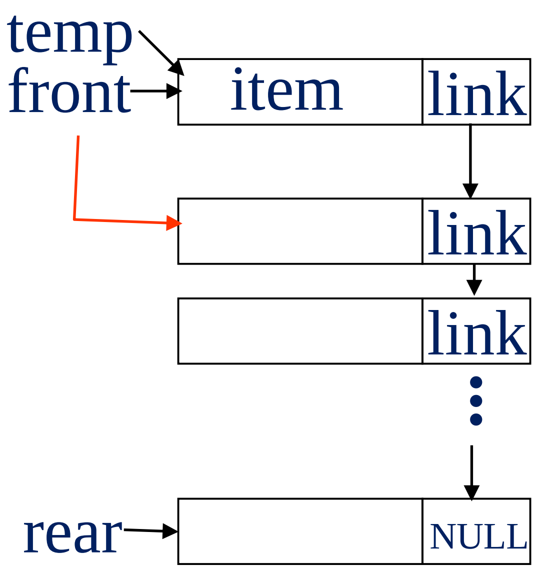


Dynamically Linked Queues (3)

- 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;
}
```

Delete from the front of a linked queue



Dynamically Linked Stacks and Queues

- The solution presented above to the *n*-stack, *m*-queue problem is both computationally and conceptually simple
  - We no longer need to shift stacks or queues to make space
  - Computation can proceed as long as there is memory available

POLYNOMIALS

Polynomials (1)

- Representing polynomials as singly linked lists
  - 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

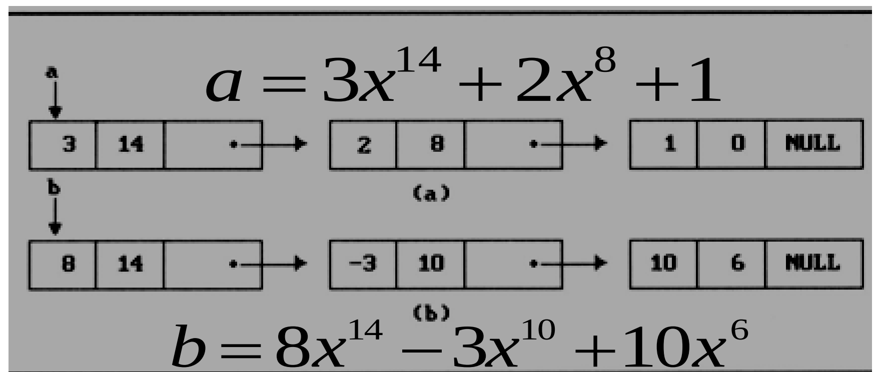
      $e^{m-1} > e^{m-2} > \ldots > e^1 > e^0 \geqq 0$

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

Polynomials (2)

- 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;
```



Polynomial representation

- Draw poly_nodes as

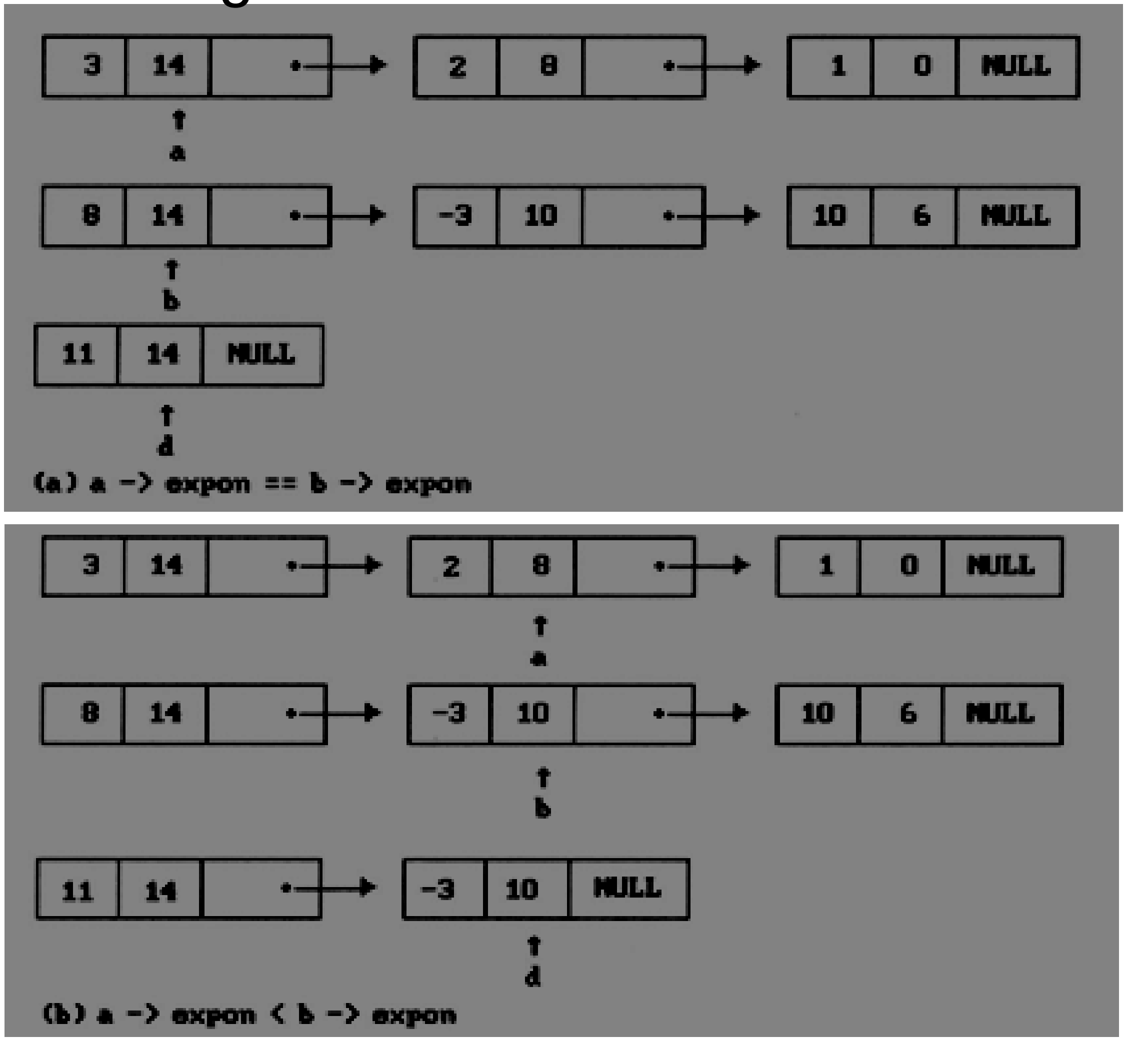
coef	expon	link

Polynomials (3)

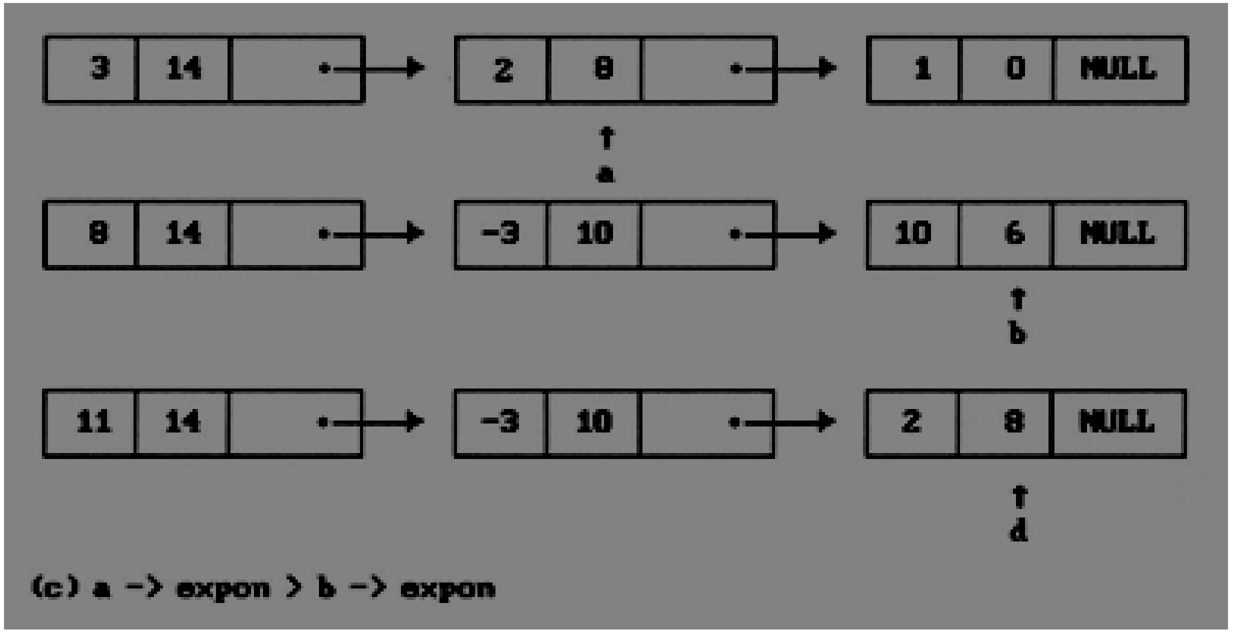
- Adding Polynomials
  - To add two polynomials, we examine their terms starting at the nodes pointed to by a and b
    - If the exponents of the two terms are equal
      - add the two coefficients
      - create a new term for the result
    - If the exponent of the current term in a is less than b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

Polynomials (4)

Generating the first three terms of d = a+b



Polynomials (5)



Polynomials (6)

```
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;
}
```

Add two polynomials

Polynomials (7)

- 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 (8)

- Analysis of padd

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

  - coefficient additions

    $0 \leq$ additions $\leq \min(m, n)$

    where m (n) denotes the number of terms in A (B).

  - 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

  - creation of new nodes

    extreme case:

    maximum number of terms in d is m+n (m + n new nodes)

    summary: O(m+n)

Polynomials (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 (10)

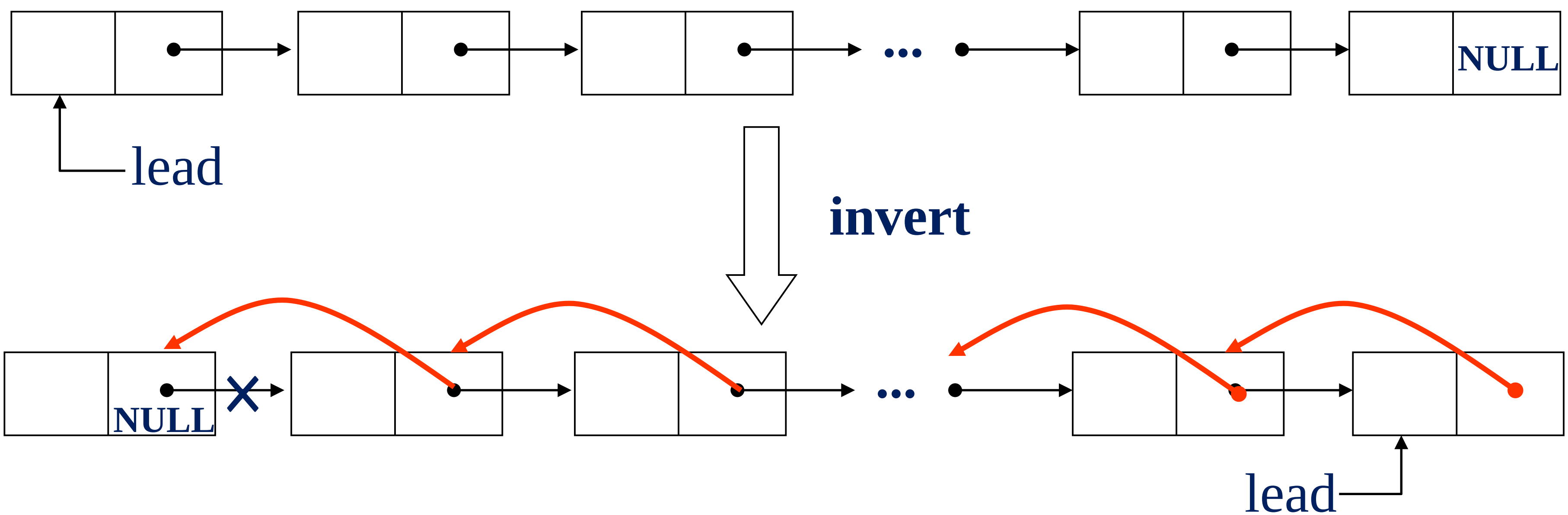
- Erase Polynomials
  - erase frees the nodes in temp

```
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

Chain (1)

- 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*).

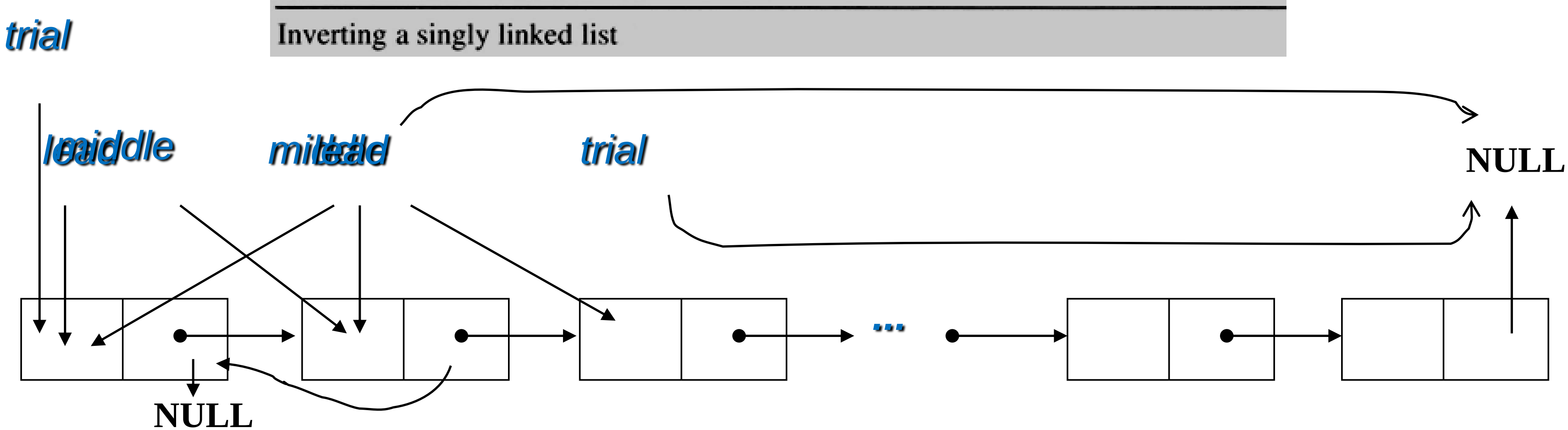


Chain (2)

```
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

Inverting a singly linked list



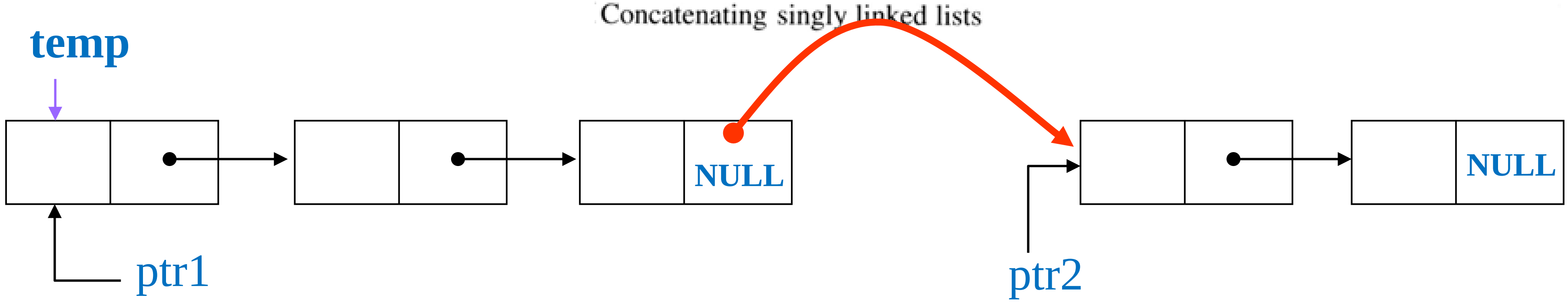
Chain (3)

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

O(length of list *ptr1*)

```
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;
   }
}
```

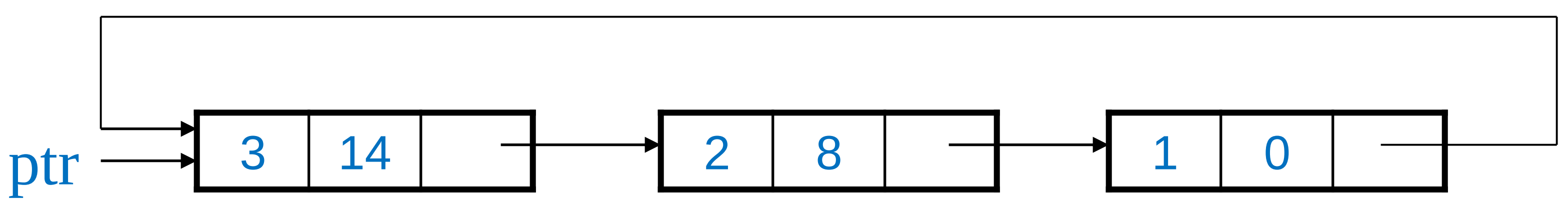
Concatenating singly linked lists



CIRCULARLY LINKED LISTS

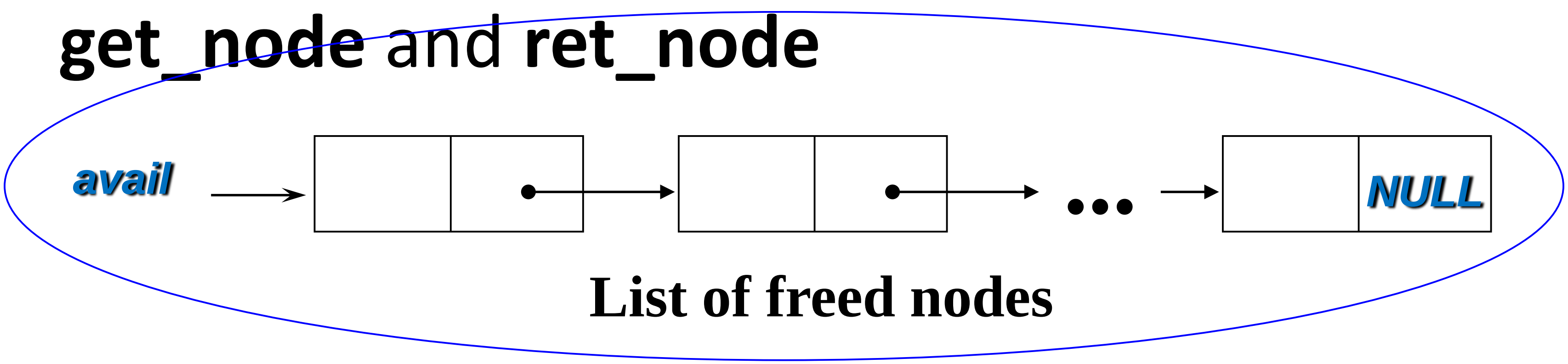
Circularly Linked Lists

- 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



Maintain an Available List (1)

- 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** and **ret_node**



Maintain an Available List (2)

- 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 have 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;
}
```

get_node function

Maintain an Available List (3)

- 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;
}
```

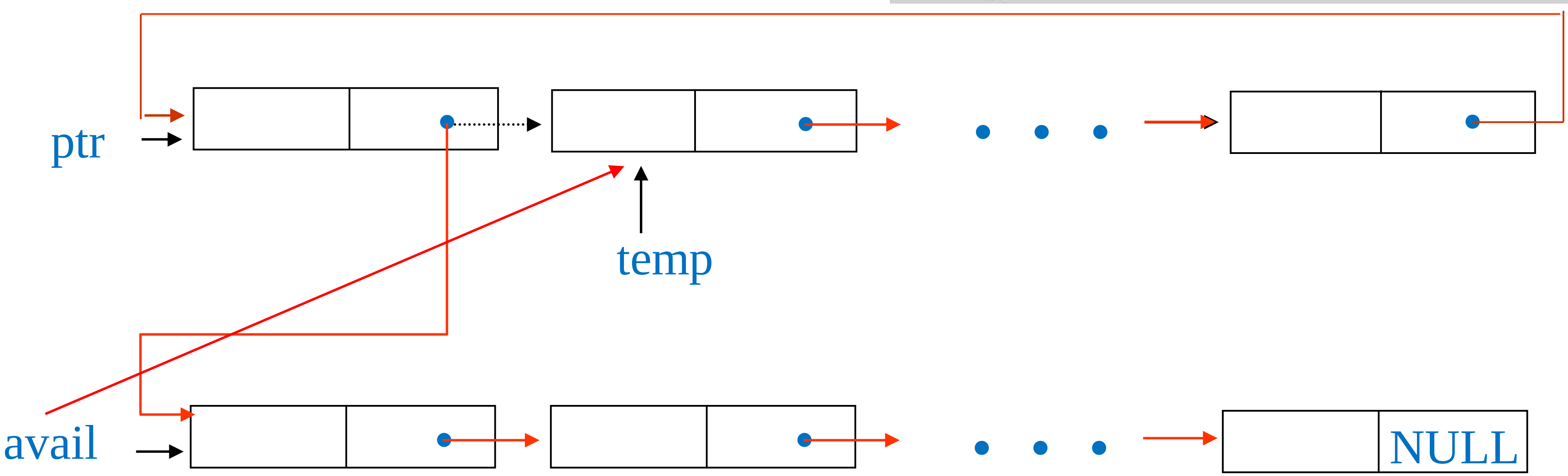
ret_node function

Maintain an Available List (4)

- 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;
  }
}
```

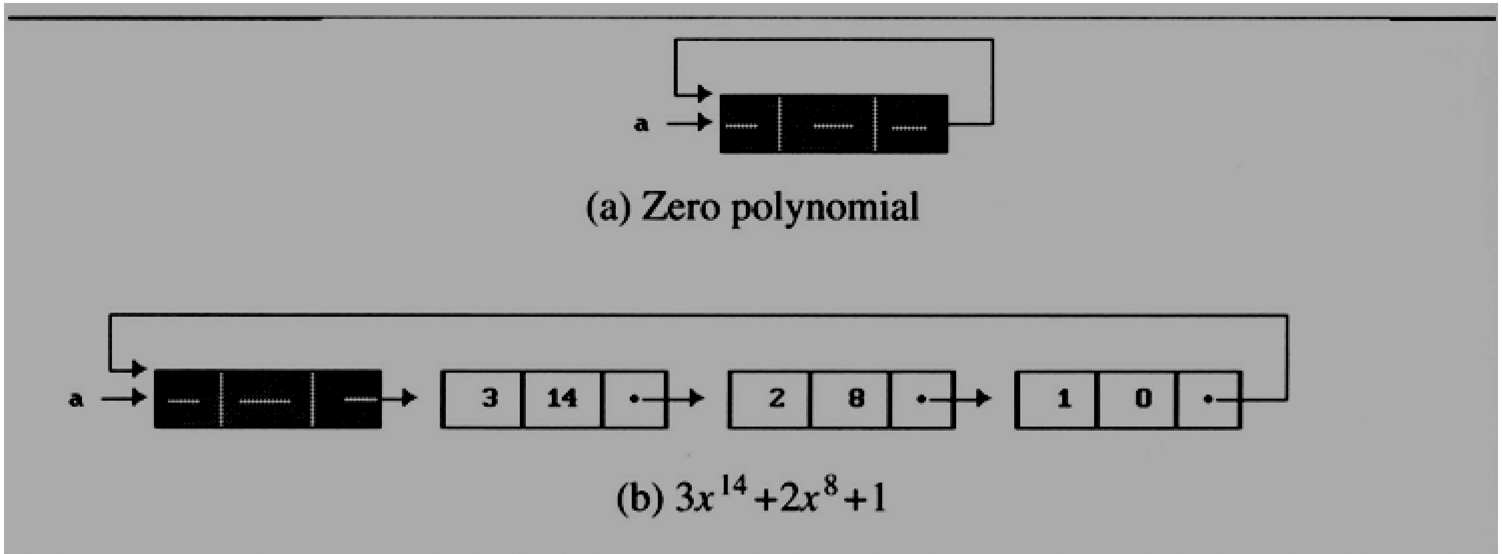
Erasing a circular list



紅色link所連接而成的 chain

Polynomial Representations (1)

- 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



Polynomial representations

Polynomial Representations (2)

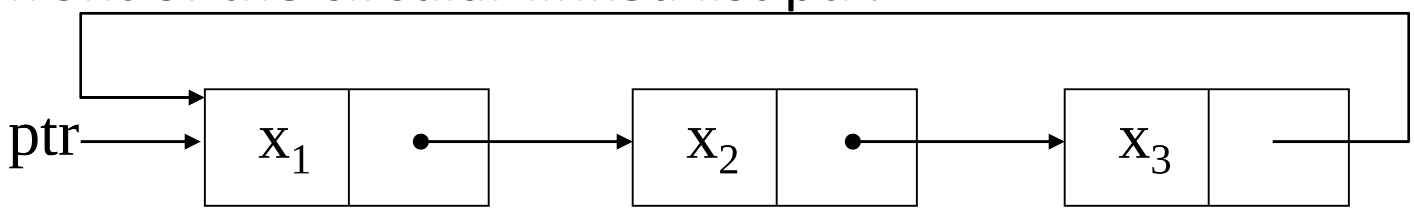
- For fit the circular list with head node representation
  - We may remove the test for (*ptr) from *cerase* (p.168)
  - Changes the original *padd* to *cpadd*

```
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;
}
```

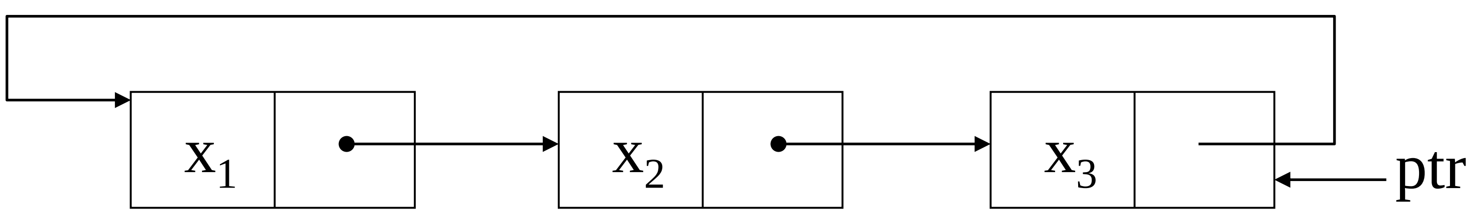
Adding circularly represented polynomials

Circularly Linked Lists (1)

- 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

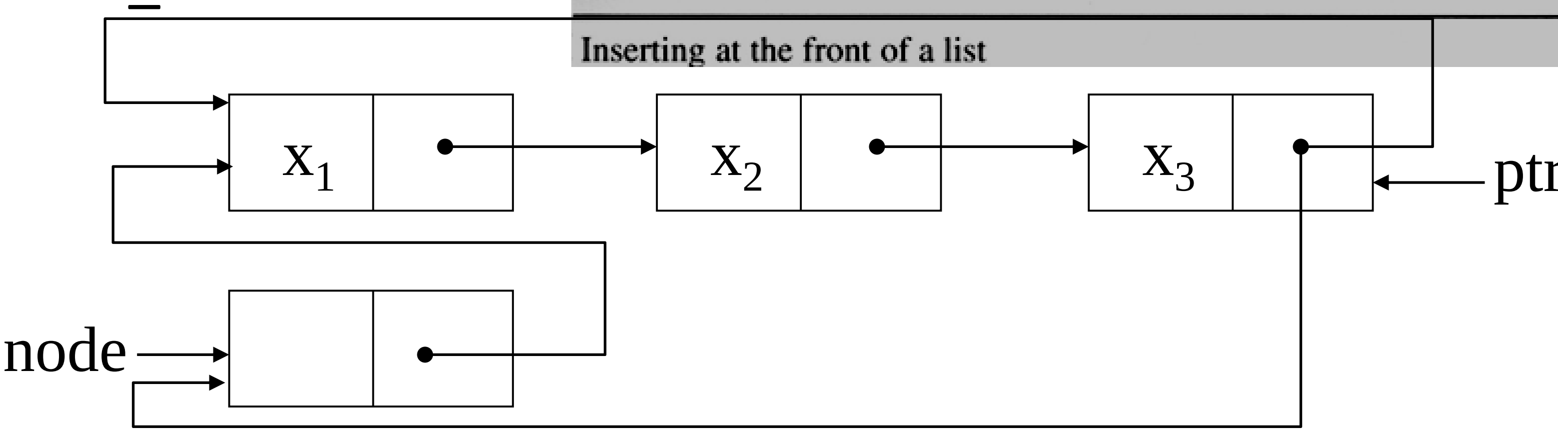


Circularly Linked Lists (2)

- 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;
   }
}
```

Inserting at the front of a list



Circularly Linked Lists (3)

- 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;
}
```

Finding the length of a circular list

EQUIVALENCE RELATIONS

Equivalence Relations (1)

- Reflexive Relation
  - For any polygon *x*, $x \equiv x$ (*e.g.*, *x* is electrically equivalent to itself)
- Symmetric Relation
  - For any two polygons *x* and *y*, if $x \equiv y$, then $y \equiv x$
- Transitive Relation
  - For any three polygons *x*, *y*, and *z*, if $x \equiv y$ and $y \equiv z$, then $x \equiv z$
- Definition
  - A relation over a set, *S*, is said to be an equivalence relation over *S* *iff* it is ***symmetric***, ***reflexive***, and ***transitive*** over *S*
- Example
  - "*equal to*" relationship is an equivalence relation

Equivalence Relations (2)

- Example
  - If we have 12 polygons numbered 0 through 11

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

    we can partition the twelve polygons into the following equivalence classes

    $\{0, 2, 4, 7, 11\}, \{1, 3, 5\}, \{6, 8, 9, 10\}$
- Two phases to determine equivalence
  - First phase: the equivalence pairs (*i*, *j*) are read in and stored
  - Second phase:
    - We begin at 0 and find all pairs of the form (0, j)
      Continue until the entire equivalence class containing 0 has been found, marked, and printed
  - Next find another object not yet output, and repeat the above process

Equivalence Relation (3)

- Program to find equivalence classes

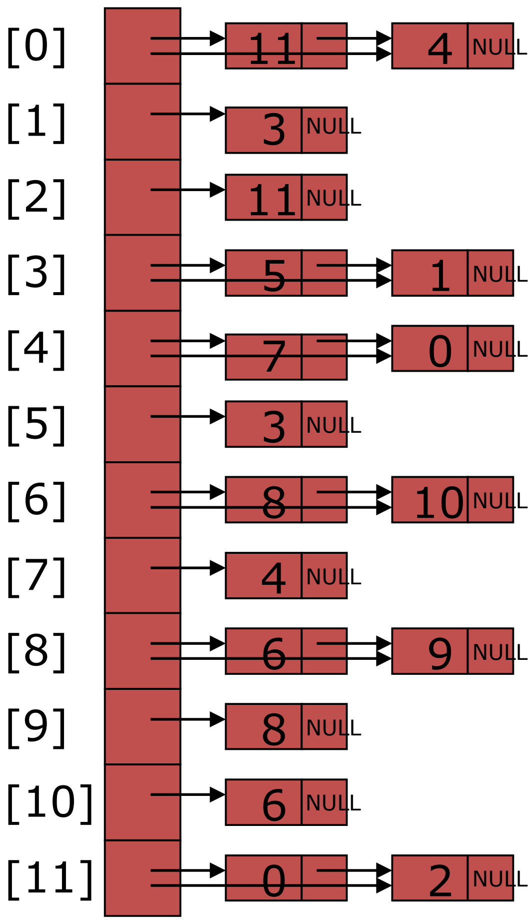
```
#include <stdio.h>
#define MAX_SIZE        24
#define IS_FULL(ptr)    (!(ptr))
#define FALSE           0
#define TRUE            1
```

```
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++){
   /*initialize seq and out */
   out[i] = TRUE;
   seq[i] = NULL;
  }
  /* Phase 1 */
  /* Phase 2 */
}
```

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

Equivalence Relations (4)

- Phase 1: read in and store the equivalence pairs <*i*, *j*>




```c
/* Phase 1: Input the equivalence pairs: */
printf("Enter a pair of numbers (-1 -1 to quit): ");
scanf("%d%d",&i,&j);
while (i >= 0) {
   x = (node_pointer)malloc(sizeof(node));
   if (IS_FULL(x)) {
      fprintf(stderr,"The memory is full\n");
      exit(1);
   }
   x->data = j;  x->link = seq[i];  seq[i] = x;
   x = (node_pointer)malloc(sizeof(node));
   if (IS_FULL(x)) {
      fprintf(stderr, "The 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);
}
```

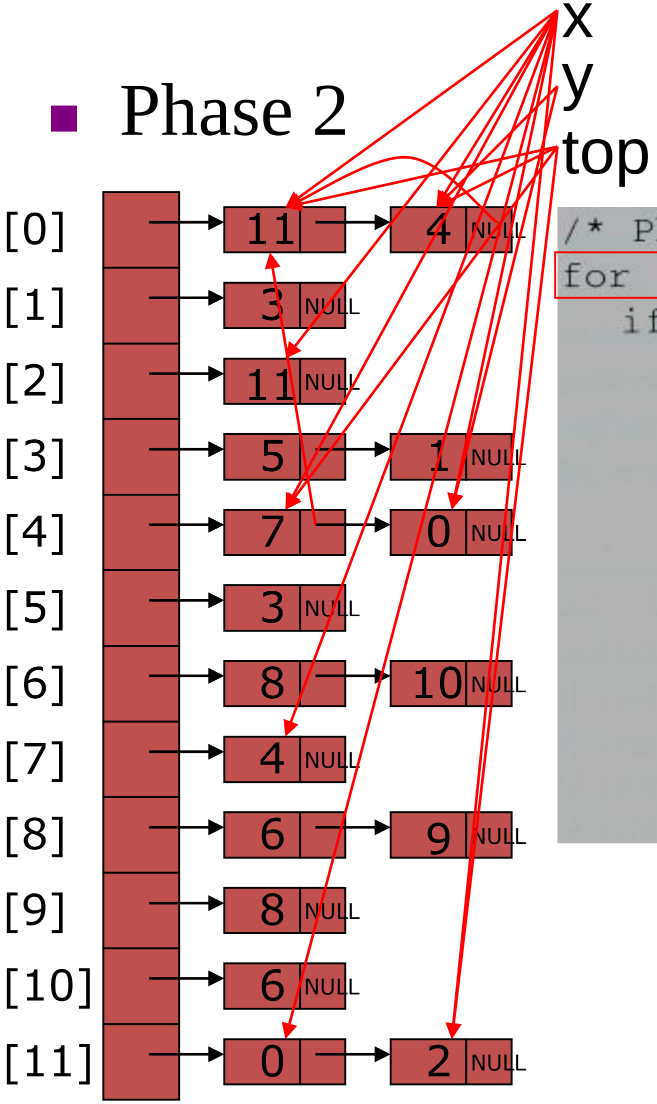
$0 \equiv 4$, $3 \equiv 1$, $6 \equiv 10$, $8 \equiv 9$, $7 \equiv 4$, $6 \equiv 8$, $3 \equiv 5$, $2 \equiv 11$, $11 \equiv 0$

Equivalence Relations (5)

- Phase 2
  - Begin at 0 and find all pairs of the form <0, j>, where 0 and *j* are in the same equivalence class
  - By transitivity, all pairs of the form <*j*, *k*> imply that *k* in the same equivalence class as 0
  - Continue this way until we have found, marked, and printed the entire equivalent class containing 0

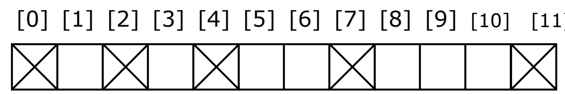
Equivalence Relations (6)

- Phase 2



i= 0 j= 11

out:



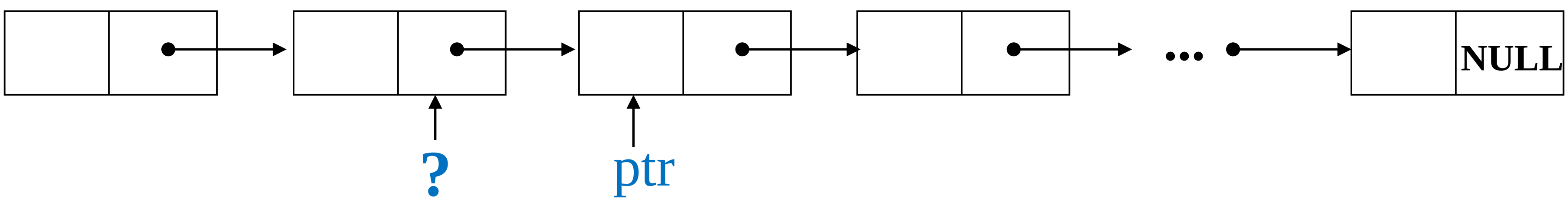

```
/* Phase 2: output the equivalence classes */
for (i = 0; i < n; i++)
  if (out[i]) {
     printf("\nNew class: %5d",i);
     out[i] = FALSE;    /* set class to false */
     x = seq[i]; top = NULL; /* initialize stack */
     for (;;) {        /* find rest of class */
       while (x) {  /* process list */
         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; /*unstack*/
     }
  }
```

New class: 0 11 4 7 2

DOUBLY LINKED LISTS

Doubly Linked Lists (1)

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

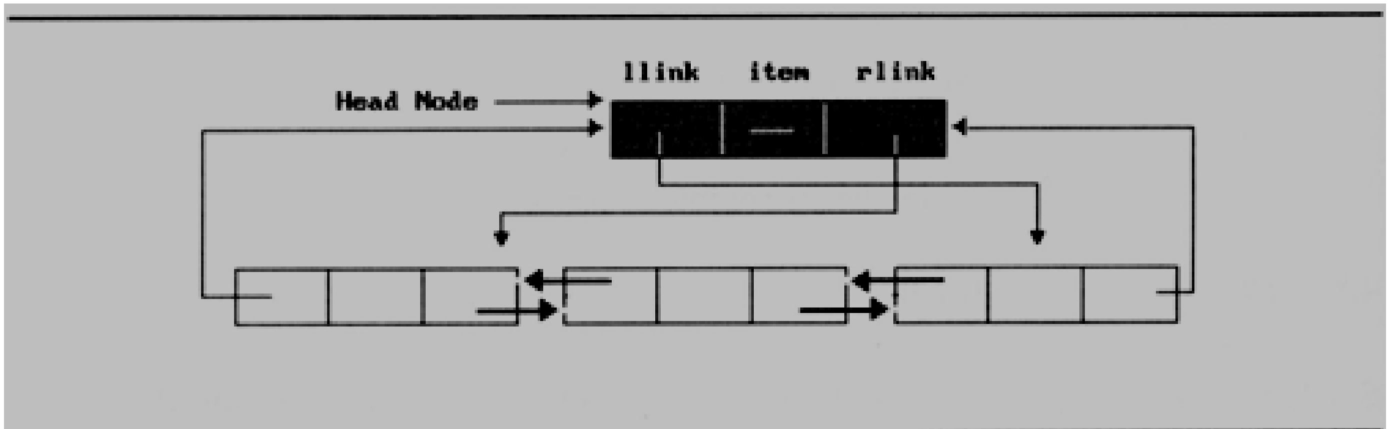


- Doubly linked list has at least three fields
  - left link field (llink), data field (item), right link field (rlink)
  - The necessary declarations

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

Doubly Linked Lists (2)

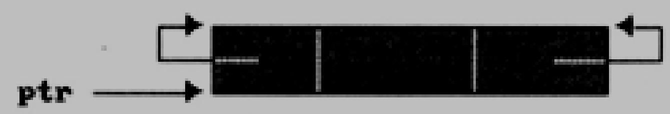
- Sample
  - doubly linked circular with head node



Doubly linked circular list with head node

Doubly Linked Lists (3)

– empty double linked circular list with head node



Empty doubly linked circular list with head node

– suppose that ptr points to any node in a doubly linked list, then:

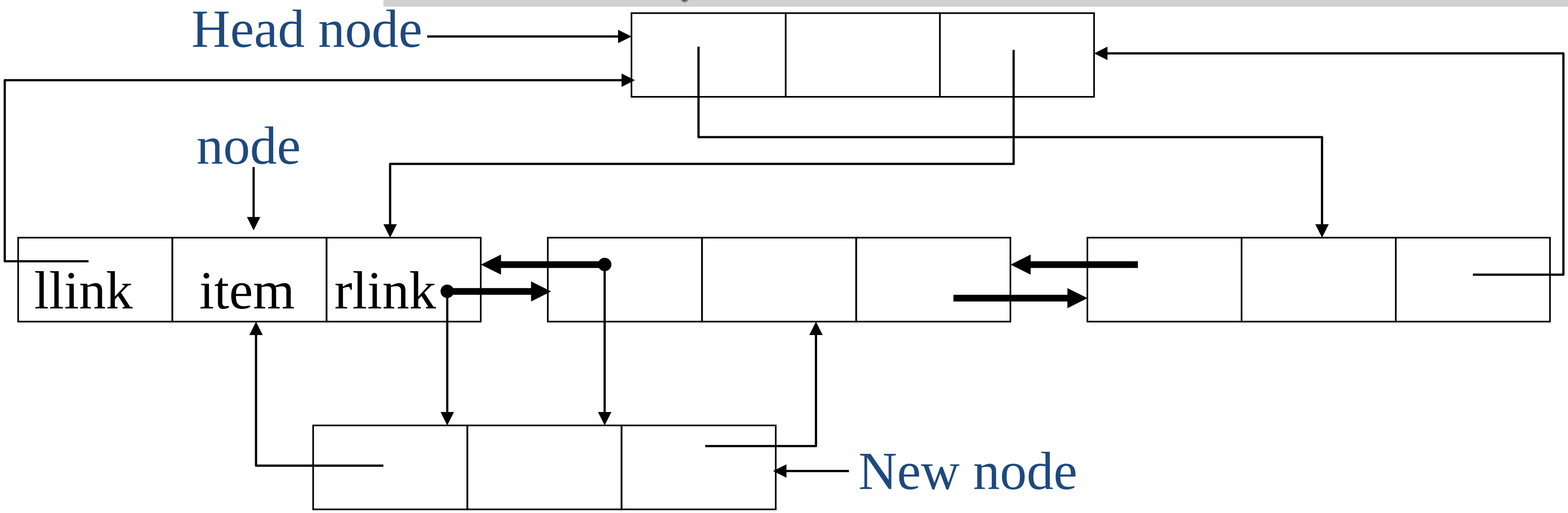
- ptr = ptr -> llink -> rlink = ptr -> rlink -> llink

Doubly Linked Lists (4)

Insert a 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;
}
```

Insertion into a doubly linked circular list



Doubly Linked Lists (5)

Delete a 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);
   }
}
```

Deletion from a doubly linked circular list