CHAPTER 4

LINKED LISTS

All the programs in this file are selected from

Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
"Fundamentals of Data Structures in C /2nd Edition",
Silicon Press, 2008.

Introduction

- Array
  successive items locate a fixed distance
- disadvantage
  - data movements during insertion and deletion
  - waste space in storing n ordered lists of varying size
- possible solution

4.1 Singly Linked Lists and Chains

	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
	.	.
	.	.
	.	.

Figure 4.1: Nonsequential list-representation (p.147)

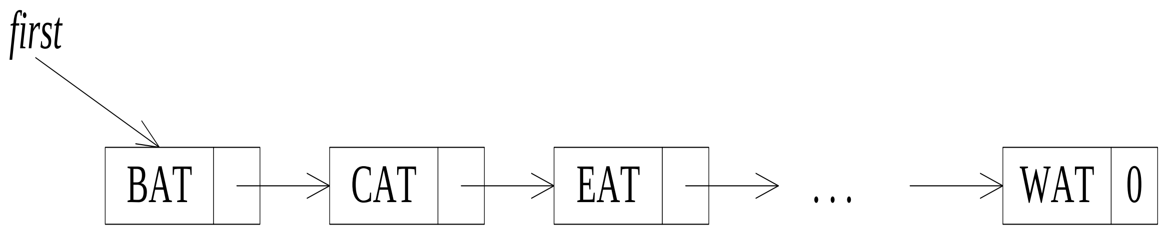
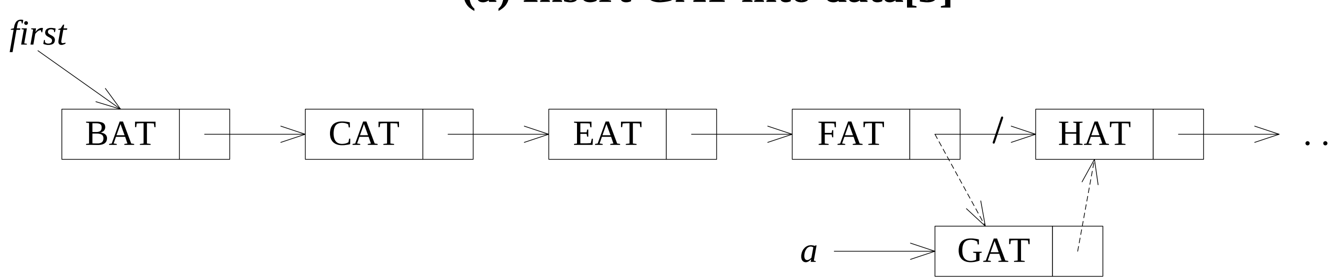


Figure 4.2: Usual way to draw a linked list (p.147)

	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

(a) Insert GAT into data[5]



(b) Insert node GAT into list

<u>**Figure 4.3:** Inserting into a linked list (p.148)</u>

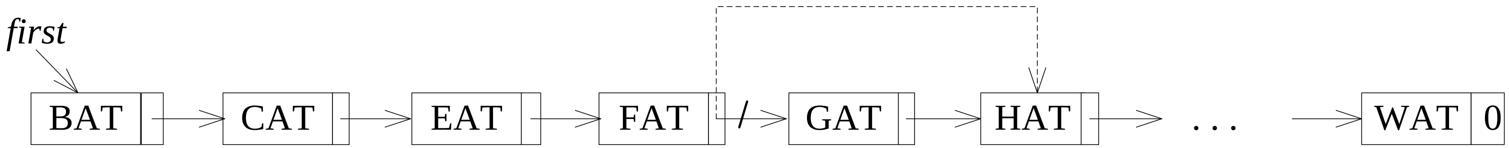


Figure 4.4: Delete GAT (p.149)

4.2 Representing Chains in C

Declaration

```
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));
```

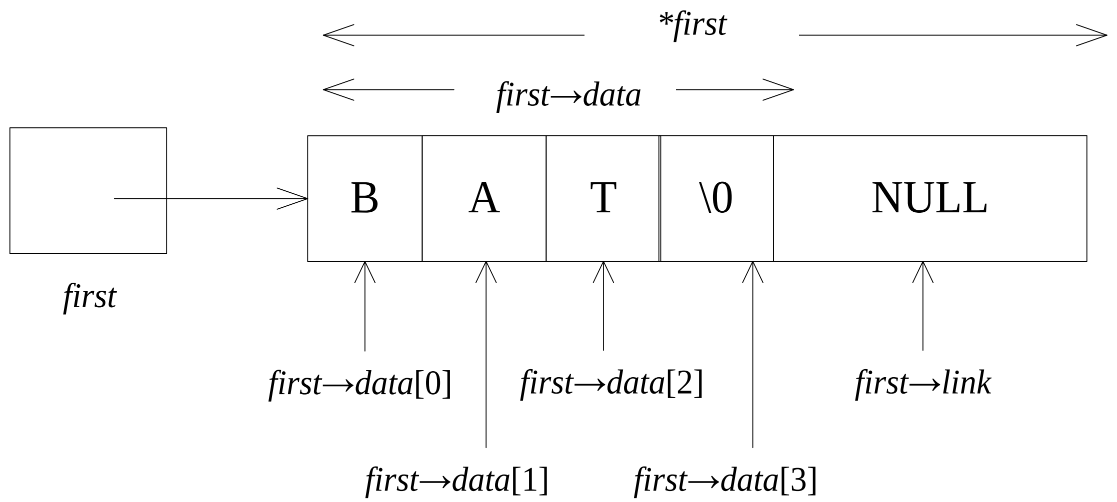
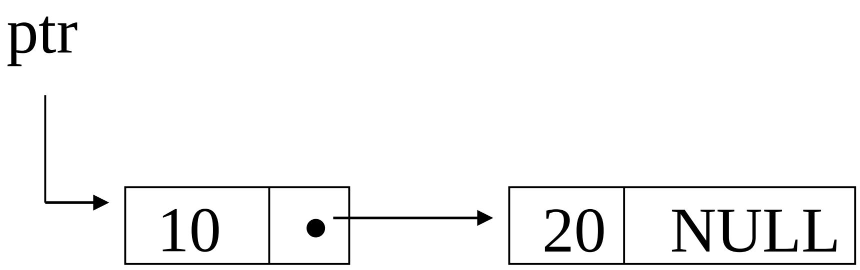


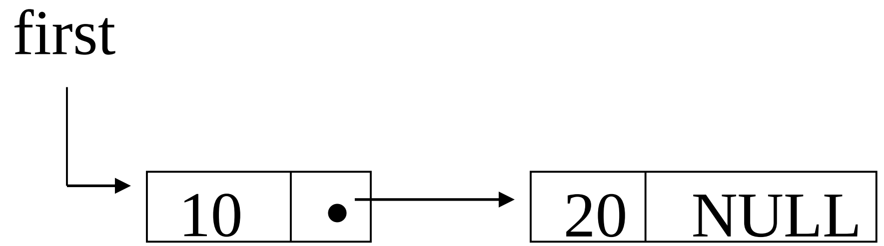
***Figure 4.5:**Referencing the fields of a node(p.151)

Example: create a two-node list




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

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

    second = ( listPointer) malloc(sizeof(listNode));


    first -> data = 10;


    return first;
}
```

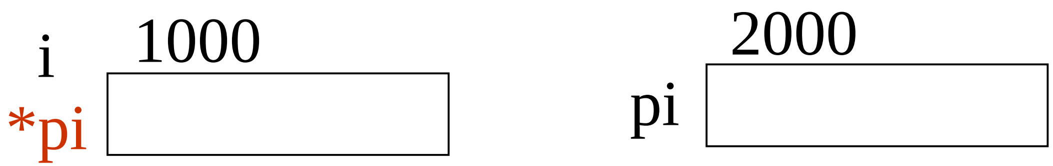


***Program 4.1:**Create a two-node list (p.152)*

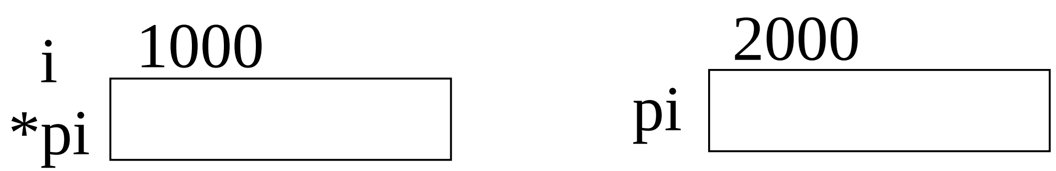
Pointer Review (1/3)

int i, *pi;

pi = &i;



i = 10 or *pi = 10



Pointer Review (2/3)

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

1000

ptr NULL

ptr->data ⇨ (*ptr).data

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

1000 2000 *ptr

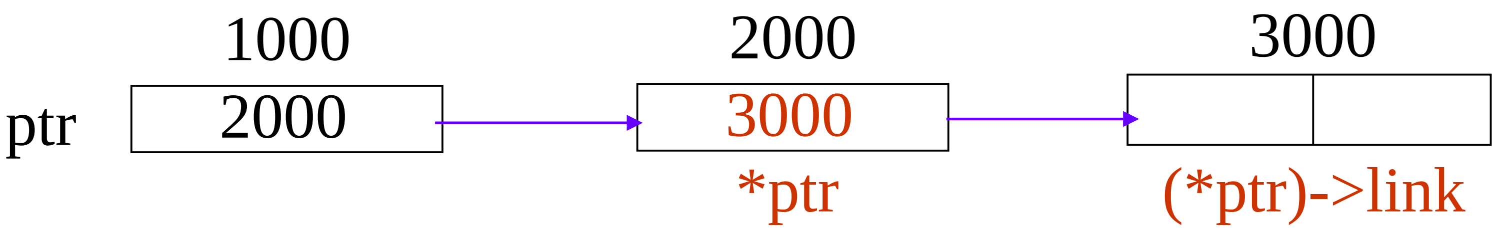
ptr 2000

data link

Pointer Review (3/3)

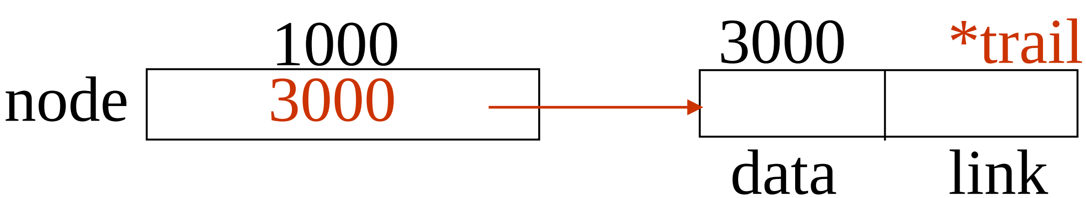
void insert(listPointer *ptr, listPointer node)

ptr: a pointer point to a pointer point to a list node



node: a pointer point to a list node

node->link⇨(*node).link

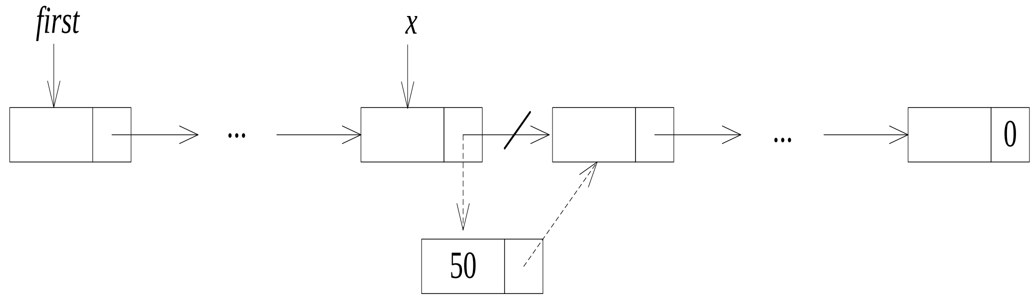


List Insertion:

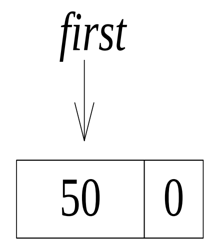
Insert a node after a specific node

```
void insert(                                    )
{
/* insert a new node with data = 50 into the chain first after node x */
    listPointer temp;
    temp = (listPointer) malloc(sizeof(listNode));
    if (IS_FULL(temp)){
      fprintf(stderr, "The memory is full\n");
      exit (1);
    }
```

temp->data = 50;
if (*first) {//noempty list



}
else { //empty list

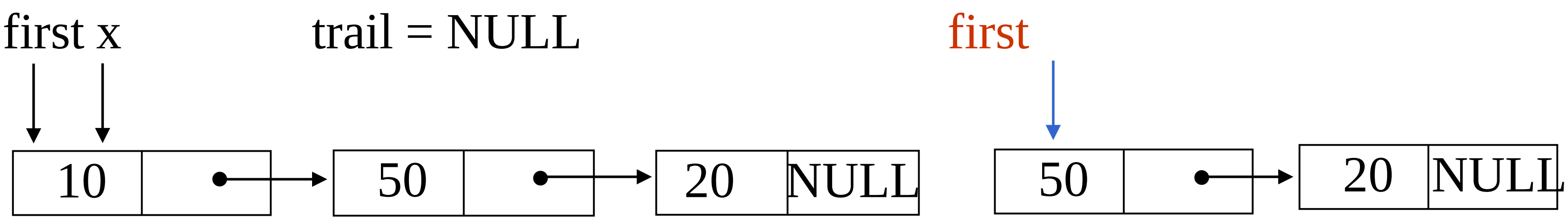


}
}

***Program 4.2:**Simple insert into front of list (p.153)

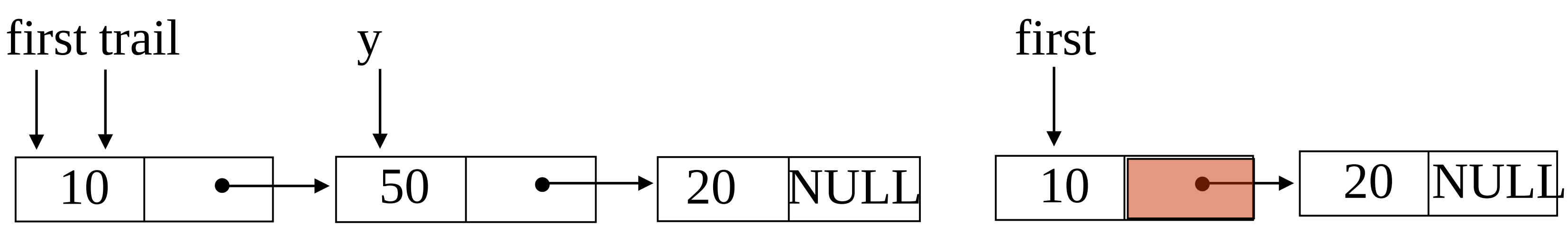
List Deletion

Delete the first node.

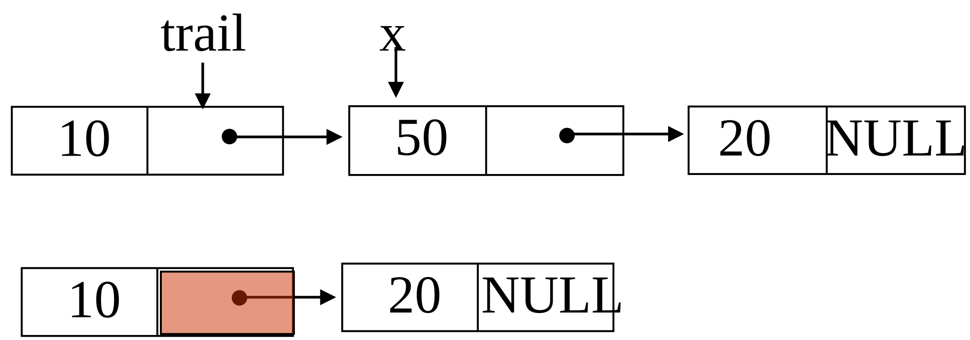


(a) before deletion (b)after deletion

Delete node other than the first node.

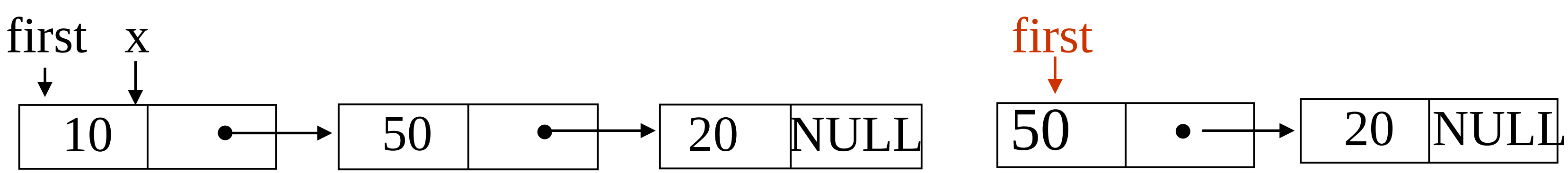


void delete(listPointer *first, listPointer trail, listPointer x)
{/* delete x from the list, trail is the preceding node
    ptr is the head of the list */
    if (trail)



    else

}



Print out a list (traverse a list)

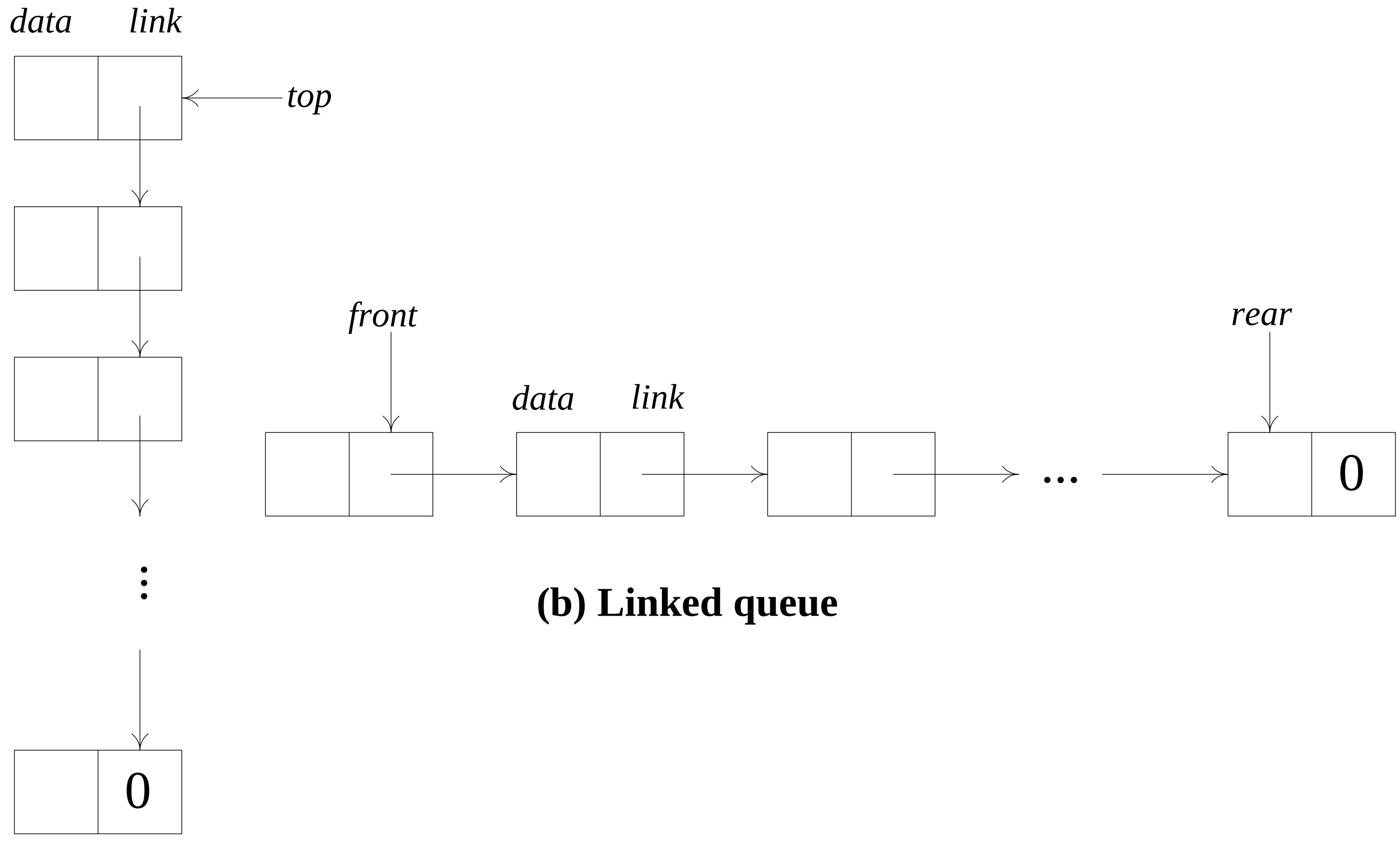
```
void print_list(listPointer first)
{
   printf("The list ocntains: ");
   for ( ; ptr; ptr = first->link)
      printf("%4d", first->data);
   printf("\n");
 }
```

Program 4.4: Printing a list (p.155)

4.3 LINKED STACKS AND QUEUES

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

Represent n stacks



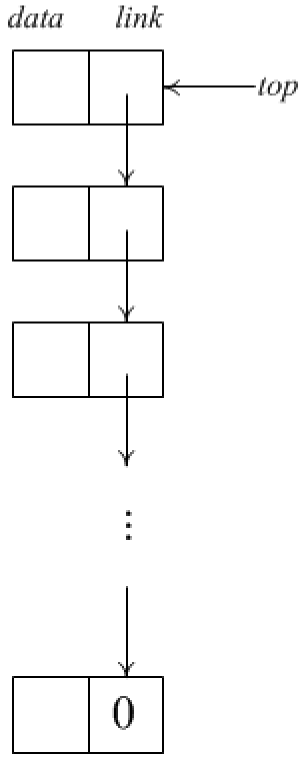
(b) Linked queue

(a) Linked stack

***Figure 4.11:**Linked stack and queue (p.157)

push in the linked stack

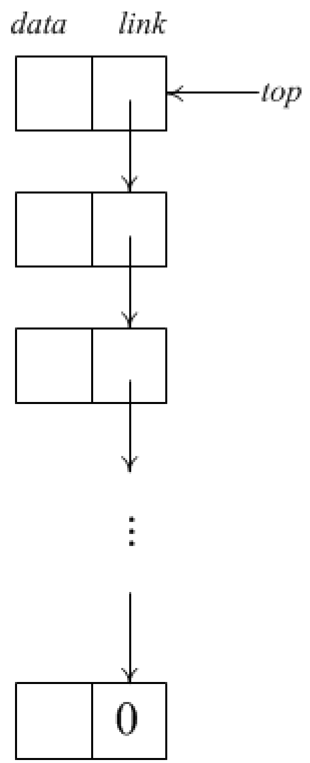
```
void add(stackPointer *top, element item)
{
  /* add an element to the top of the stack */
  stackPointer temp =
                    (stackPointer) malloc (sizeof (stack));
  if (IS_FULL(temp)) {
    fprintf(stderr, " The memory is full\n");
    exit(1);
    }


}
```



***Program 4.5:** Add to a linked stack (p.158)*

pop from the linked stack

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



     return data;
}
```



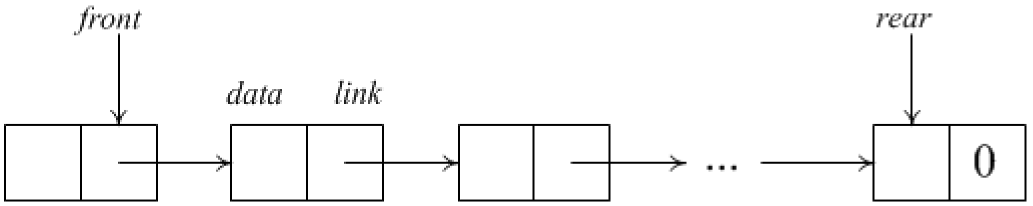
***Program 4.6**: Delete from a linked stack (p.158)

Represent n queues

```
#define MAX_QUEUES 10 /* maximum number of queues */
typedef struct queue *queuePointer;
typedef struct queue {
        element data;
        queuePointer link;
        };
queuePointer front[MAX_QUEUE], rear[MAX_QUEUES];
```

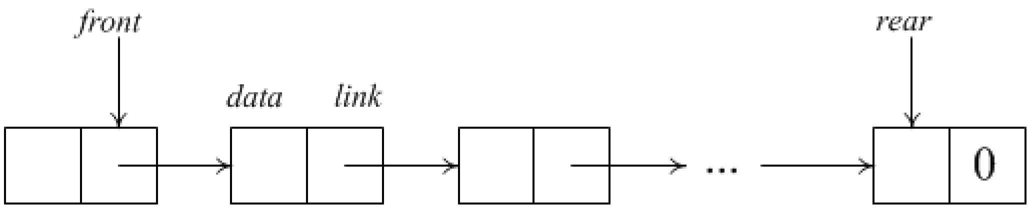
enqueue in the linked queue

```
void addq(queuePointer *front, queuePointer *rear, element item)
{  /* add an element to the rear of the queue */
   queuePointer temp =
                     (queuePointer) malloc(sizeof (queue));
   if (IS_FULL(temp)) {
      fprintf(stderr, " The memory is full\n");
      exit(1);
      }
      temp->data = item;
      temp->link = NULL;
      if (*front
      else

}
```



dequeue from the linked queue (similar to push)

```
element deleteq(queuePointer *front) {
/* delete an element from the queue */
    queuePointer temp = *front;
    element item;
    if (IS_EMPTY(*front))  {
        fprintf(stderr,  “The queue is empty\n”);
        exit(1);
    }


    return item;
}
```



Polynomials

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

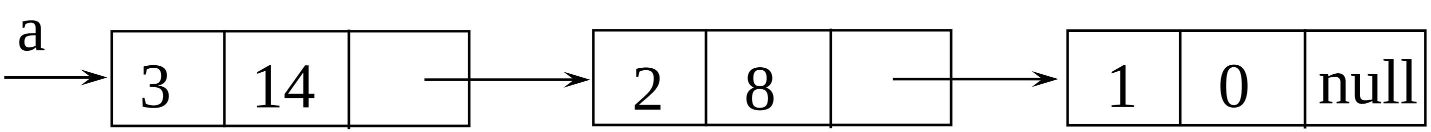
Representation

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

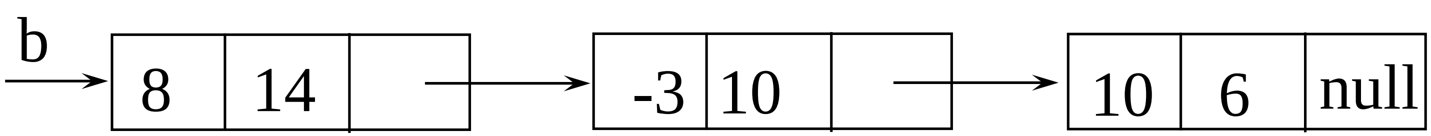
coef	expon	link

Examples

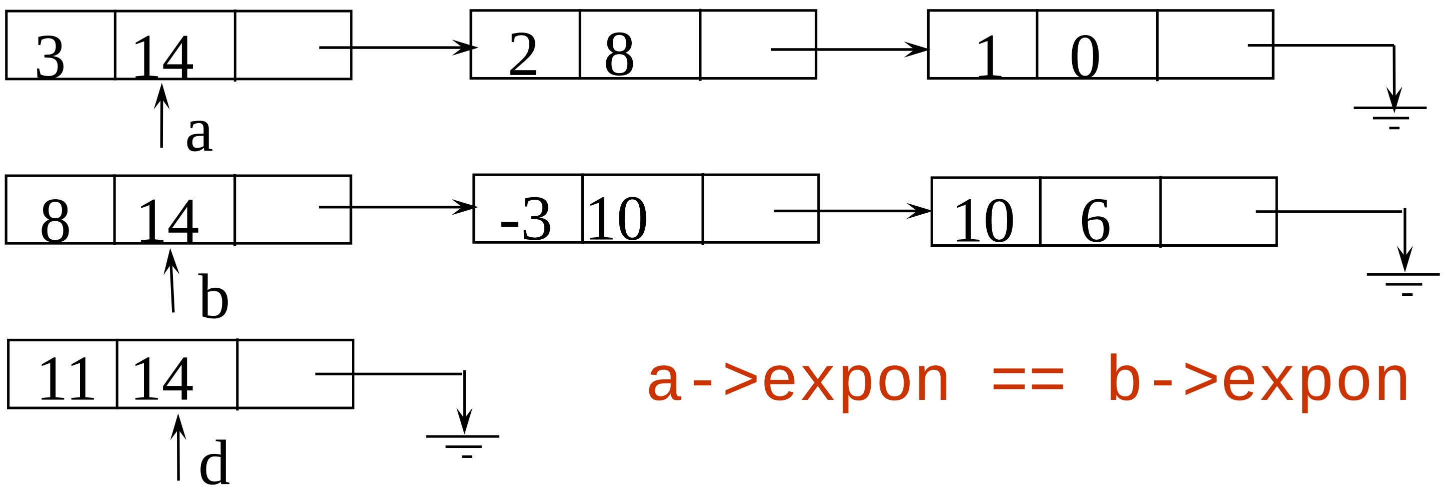
$$a = 3x^{14} + 2x^{8} + 1$$

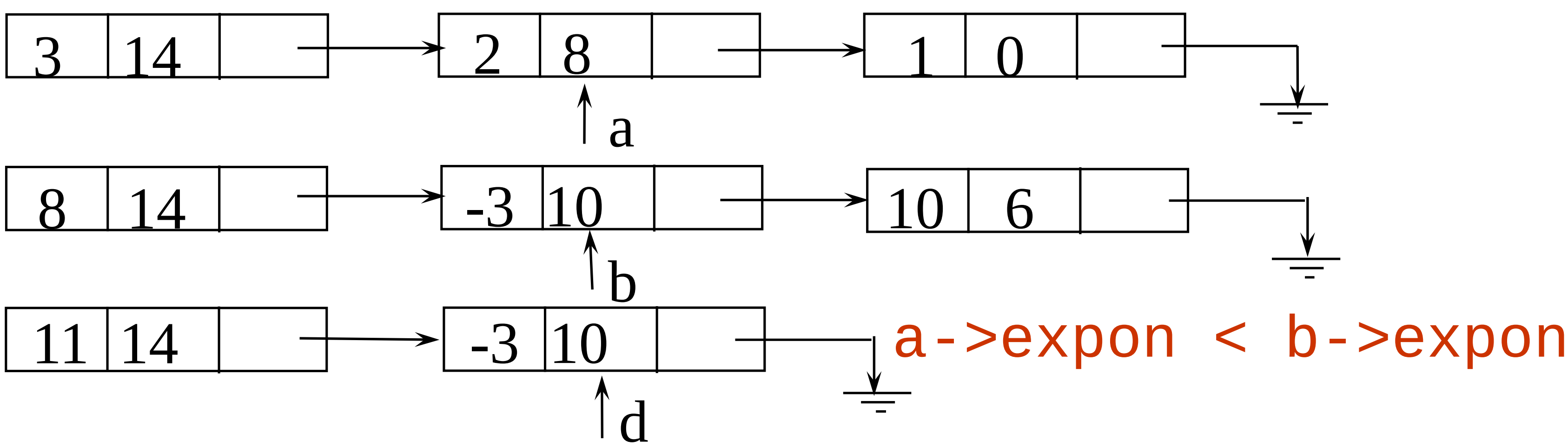


$$b = 8x^{14} - 3x^{10} + 10x^{6}$$

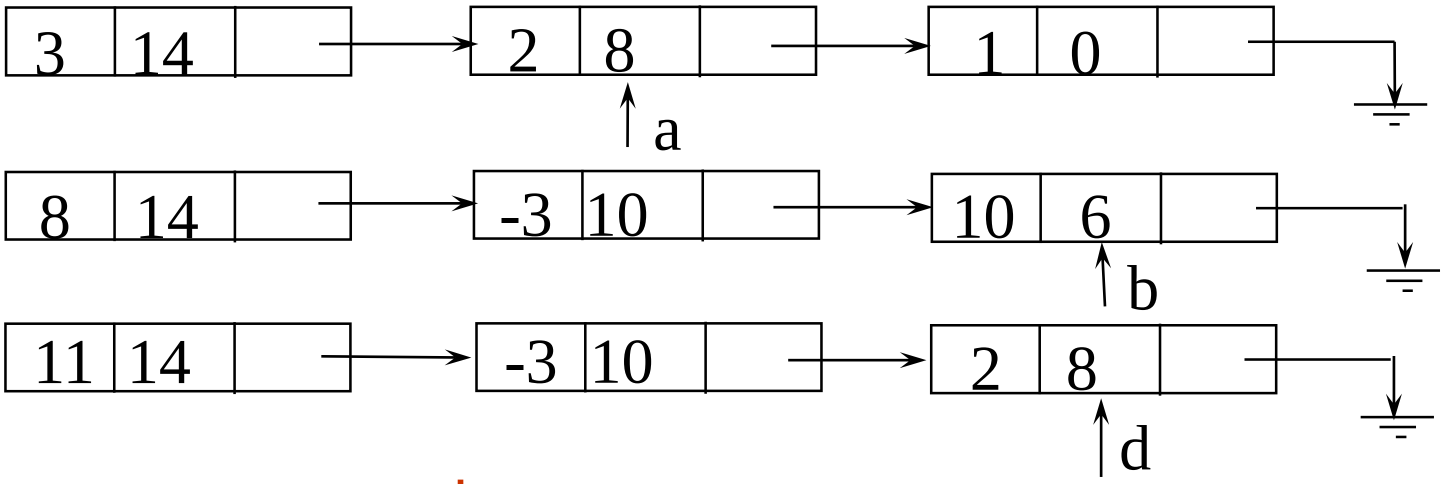


Adding Polynomials





Adding Polynomials (*Continued*)



`a->expon > b->expon`

Alogrithm for Adding Polynomials

```
poly_pointer padd(polyPointer a, polyPointer b)
{
    polyPointer front, rear, temp;
    int sum;
    rear =(polyPointer)malloc(sizeof(polyNode));
    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;
        }
    }
    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;
    temp = front;  front = front->link;  free(temp);
    return front;
}
```

Delete extra initial node.

Attach a Term

```
void attach(float coefficient, int exponent,
            polyPointer *ptr)
{
/* create a new node attaching to the node pointed to
   by ptr. ptr is updated to point to this new node. */
     polyPointer temp;
     temp = (polyPointer) malloc(sizeof(polyNode));
     if (IS_FULL(temp)) {
          fprintf(stderr, “The memory is full\n”);
          exit(1);
     }
     temp->coef = coefficient;
     temp->expon = exponent;
     (*ptr)->link = temp;
     *ptr = temp;
}
```

A Suite for Polynomials

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

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

```
readPoly()
printPoly()
padd()
psub()
pmult()
```

Erase Polynomials

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

O(n)

Circularly Linked Lists

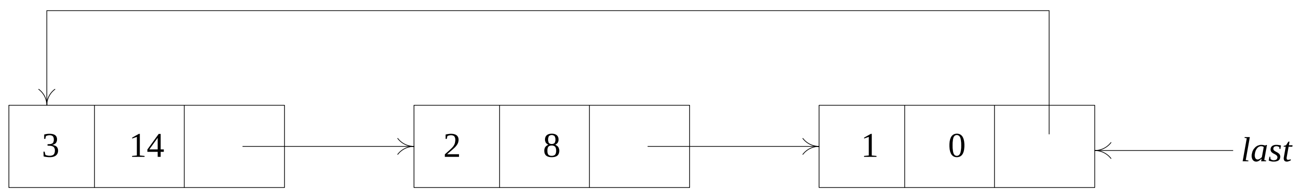
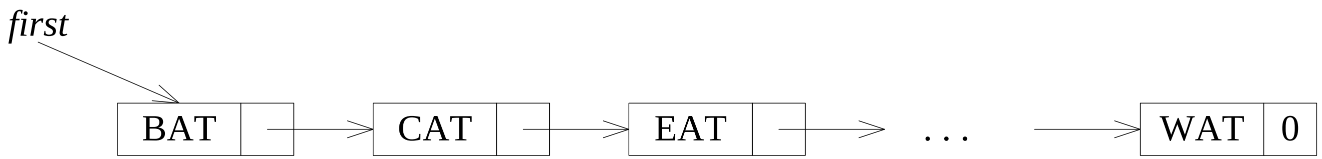


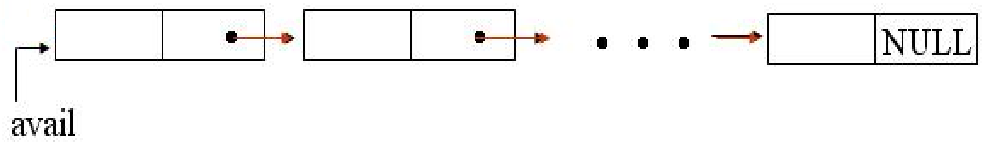
***Figure 4.14:**Circular representation of $3x^{14} + 2x^{8} + 1$ (p.166)

circular list vs. chain



Maintain an Available List

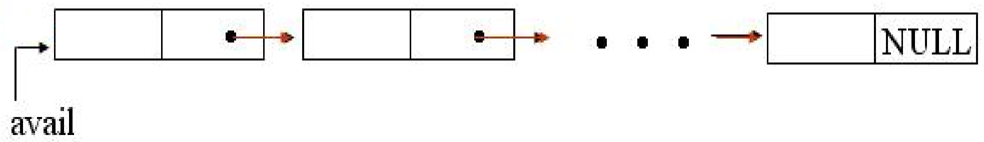
```
polyPointer getNode(void)
{
  polyPointer node;
  if (avail) {



  }
  else {
      node = (polyPointer)malloc(sizeof(polyNode));
      if (IS_FULL(node)) {
          printf(stderr, "The memory is full\n");
          exit(1);
      }
  }


}
```



Maintain an Available List (*Continued*)

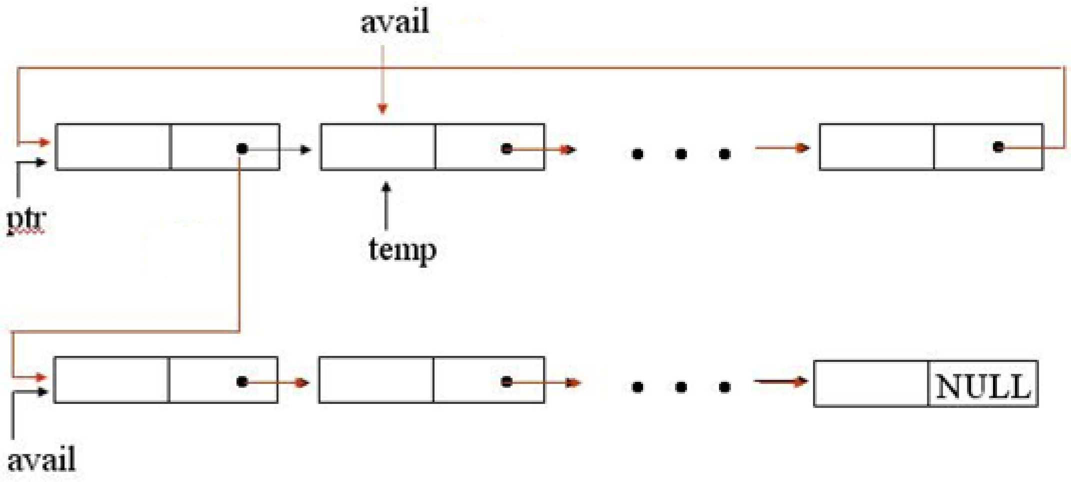
Insert node to the front of this list

```
void retNode(polyPointer node)
{


}
```



Maintain an Available List (*Continued*)

```
void cerase(polyPointer *ptr)
{
    polyPointer temp;
    if




    }
}
```

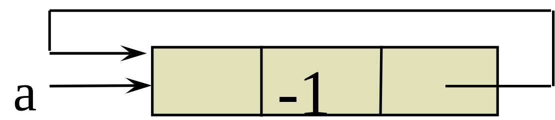


Independent of # of nodes in a list O(1) constant time

Head Node

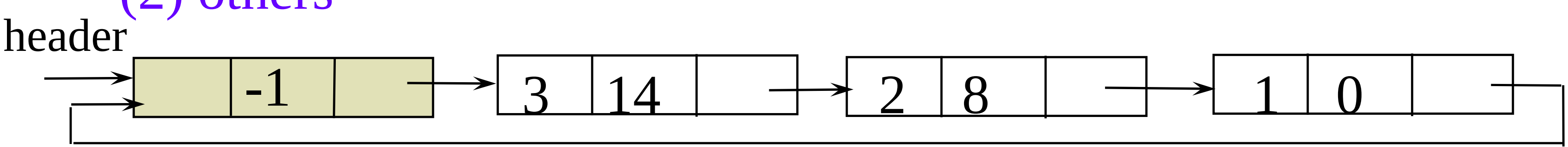
Represent polynomial as circular list.

(1) zero



Zero polynomial

(2) others



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

Another Padd

```
poly_pointer cpadd(polyPointer a, polyPointer b)
{
  polyPointer starta, d, lastd;
  int sum, done = FALSE;
  starta = a;
  a = a->link;
  b = b->link;
  d = get_node();
  d->expon = -1;    lastd = d;
  do {
    switch (COMPARE(a->expon, b->expon)) {
      case -1: attach(b->coef, b->expon, &lastd);
               b = b->link;
               break;
```

Set expon field of head node to -1.

Another Padd (*Continued*)

```
        case 0: if (starta == a) done = TRUE;
                else {
                  sum = a->coef + b->coef;
                  if (sum) attach(sum,a->expon,&lastd);
                  a = a->link;   b = b->link;
                }
                break;
        case 1: attach(a->coef,a->expon,&lastd);
                a = a->link;
      }
    } while (!done);
    lastd->link = d;
    return d;
}
```

Link last node to first

4.5 Additional List Operations

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

- **Invert single linked lists**
- **Concatenate two linked lists**

Invert Single Linked Lists

Use two extra pointers: middle and trail.

```
listPointer invert(listPointer lead)
{
    listPointer middle, trail;
    middle = NULL;
    while (lead) {
        trail = middle;
        middle = lead;
        lead = lead->link;
        middle->link = trail;
    }
    return middle;
}
```

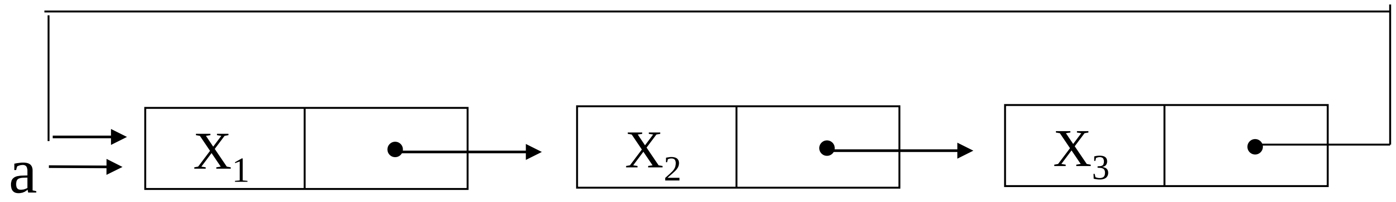
Concatenate Two Lists

```
listPointer concatenate(listPointer ptr1, listPointer ptr2)
{
  listPointer 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(m) where m is # of elements in the first list

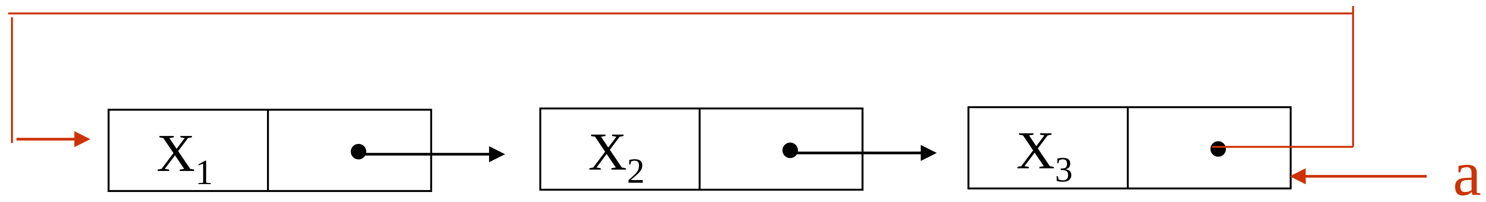
4.5.2 Operations For Circularly Linked List

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



Problem: move down the whole list.

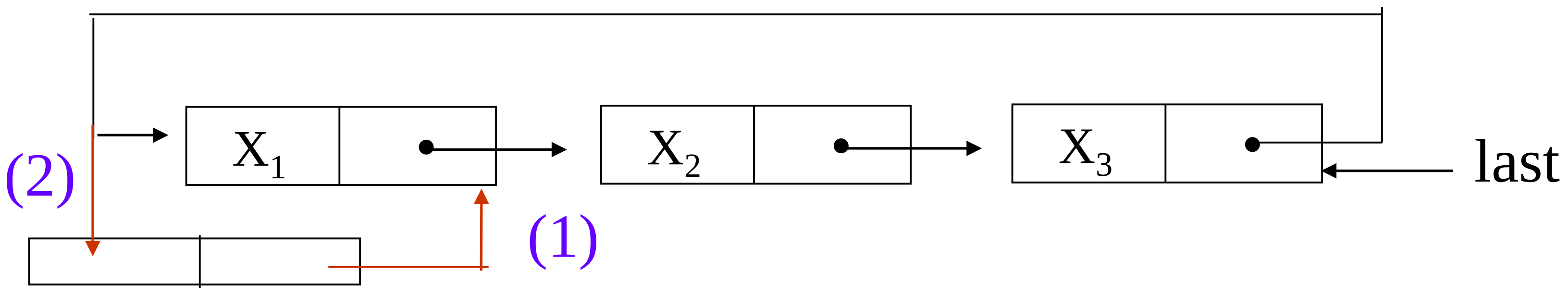
A possible solution:



Note a pointer points to the last node.

Operations for Circular Linked Lists

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



Length of Linked List

```
int length(listPointer last)
{
    list_pointer temp;
    int count = 0;
    if (last) {
        temp = last;
        do {
             count++;
             temp = temp->link;
        } while (temp!=last);
    }
    return count;
}
```

4.7 Sparse Matrices

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

Revisit Sparse Matrices

of head nodes = max{# of rows, # of columns}

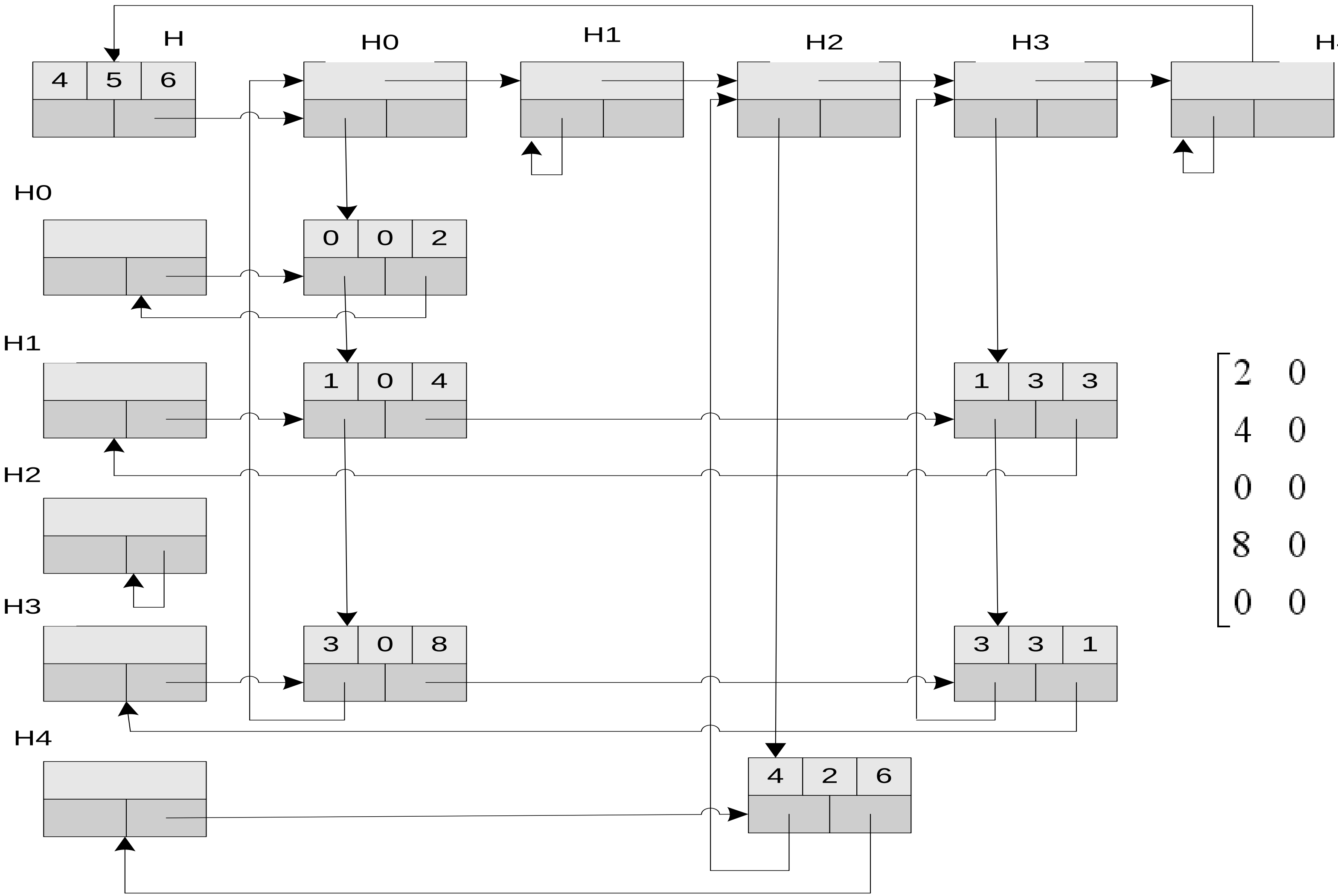
head node

<table>
<tr><td colspan="2">next</td></tr>
<tr><td>down</td><td>right</td></tr>
</table>

entry node

<table>
<tr><td>row</td><td>col</td><td>value</td></tr>
<tr><td>down</td><td colspan="2">right</td></tr>
</table>

Linked Representation for Matrix



$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

```
#define MAX_SIZE 50 /* size of largest matrix */
typedef enum {head, entry} tagfield;
typedef struct matrixNode *matrixPointer;
typedef struct entryNode {



            };
typedef struct matrixNode {
```

```
        union {



            } u;
        };
matrixPointer hdnode[MAX_SIZE];
```

Read in a Matrix

```
matrix_pointer mread(void)
{
/* read in a matrix and set up its linked
 list. An global array hdnode is used */
  int num_rows, num_cols, num_terms;
  int num_heads, i;
  int row, col, value, current_row;
  matrix_pointer temp, last, node;

  printf("Enter the number of rows, columns
        and number of nonzero terms: ");
```

```
scanf("%d%d%d", &num_rows, &num_cols,
      &num_terms);
num_heads =
(num_cols>num_rows)? num_cols : num_rows;
/* set up head node for the list of head
   nodes */
node = new_node();    node->tag = entry;
node->u.entry.row = num_rows;
node->u.entry.col = num_cols;

if (!num_heads) node->right = node;
else { /* initialize the head nodes */
  for (i=0; i<num_heads; i++) {
    term= new_node();
    hdnode[i] = temp;
    hdnode[i]->tag = head;
    hdnode[i]->right = temp;
    hdnode[i]->u.next = temp;
  }
```

O(max(n,m))

```
    current_row= 0;    last= hdnode[0];
    for (i=0; i<num_terms; i++) {
      printf("Enter row, column and value:");
      scanf("%d%d%d", &row, &col, &value);
      if (row>current_row) {
        last->right= hdnode[current_row];
        current_row= row;  last=hdnode[row];
      }
      temp = new_node();
      temp->tag=entry; temp->u.entry.row=row;
      temp->u.entry.col = col;
      temp->u.entry.value = value;
      last->right = temp;/*link to row list
*/
      last= temp;
      /* link to column list */
      hdnode[col]->u.next->down = temp;
      hdnode[col]=>u.next = temp;
    }
```

```
    /*close last row */
    last->right = hdnode[current_row];
    /* close all column lists */
    for (i=0; i<num_cols; i++)
      hdnode[i]->u.next->down = hdnode[i];
    /* link all head nodes together */
    for (i=0; i<num_heads-1; i++)
      hdnode[i]->u.next = hdnode[i+1];
    hdnode[num_heads-1]->u.next= node;
    node->right = hdnode[0];
  }
  return node;
}
```

O(max{#_rows, #_cols}+#_terms)

Write out a Matrix

```
void mwrite(matrix_pointer node)
{ /* print out the matrix in row major form */
  int i;
  matrix_pointer temp, head = node->right;
  printf("\n num_rows = %d, num_cols= %d\n",
         node->u.entry.row,node->u.entry.col);
  printf("The matrix by row, column, and
         value:\n\n");
  for (i=0; i<node->u.entry.row; i++) {
    for (temp=head->right;temp!=head;temp=temp->right)
      printf("%5d%5d%5d\n", temp->u.entry.row,
           temp->u.entry.col, temp->u.entry.value);
    head= head->u.next; /* next row */
  }
}
```

O(#_rows+#_terms)

Erase a Matrix

```
void merase(matrix_pointer *node)
{
  int i, num_heads;
  matrix_pointer x, y, head = (*node)->right;
  for (i=0; i<(*node)->u.entry.row; i++) {
    y=head->right;
    while (y!=head) {
      x = y;  y = y->right;  free(x);
    }
    x= head;  head= head->u.next; free(x);
  }
  y = head;
  while (y!=*node) {
    x = y;  y = y->u.next;  free(x);
  }
  free(*node);  *node = NULL;
}
```

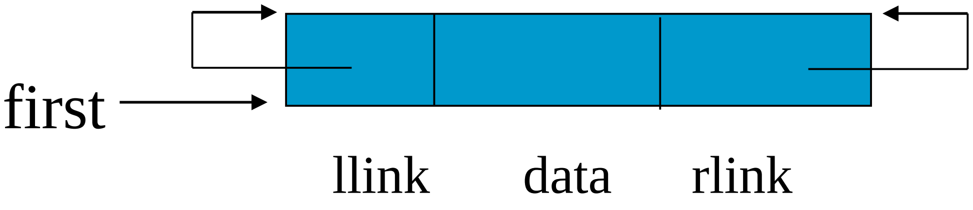
O(#_rows+#_cols+#_terms)

Doubly Linked List

Move in forward and backward direction.

Singly linked list (in one direction only)
How to get the preceding node during deletion or insertion?
Using 2 pointers

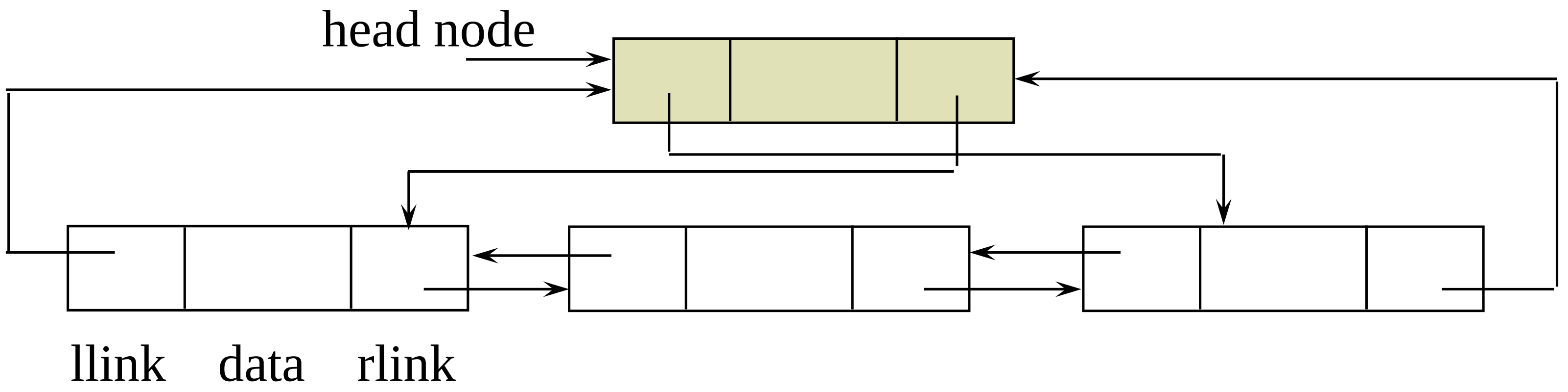
Node in doubly linked list
left link field (llink)
data field (data)
right link field (rlink)



Doubly Linked Lists

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

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



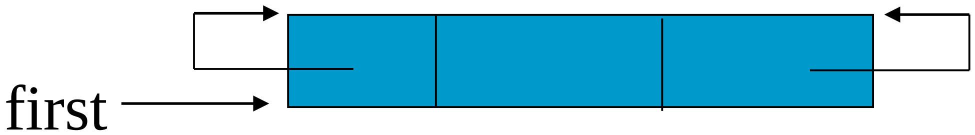
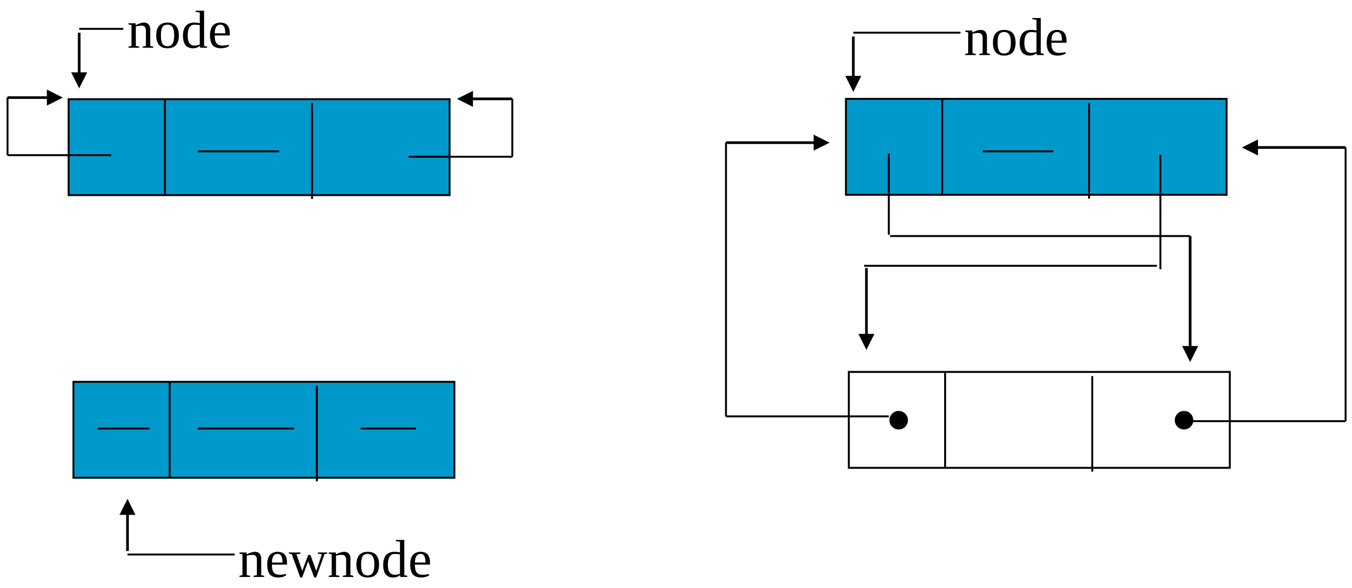


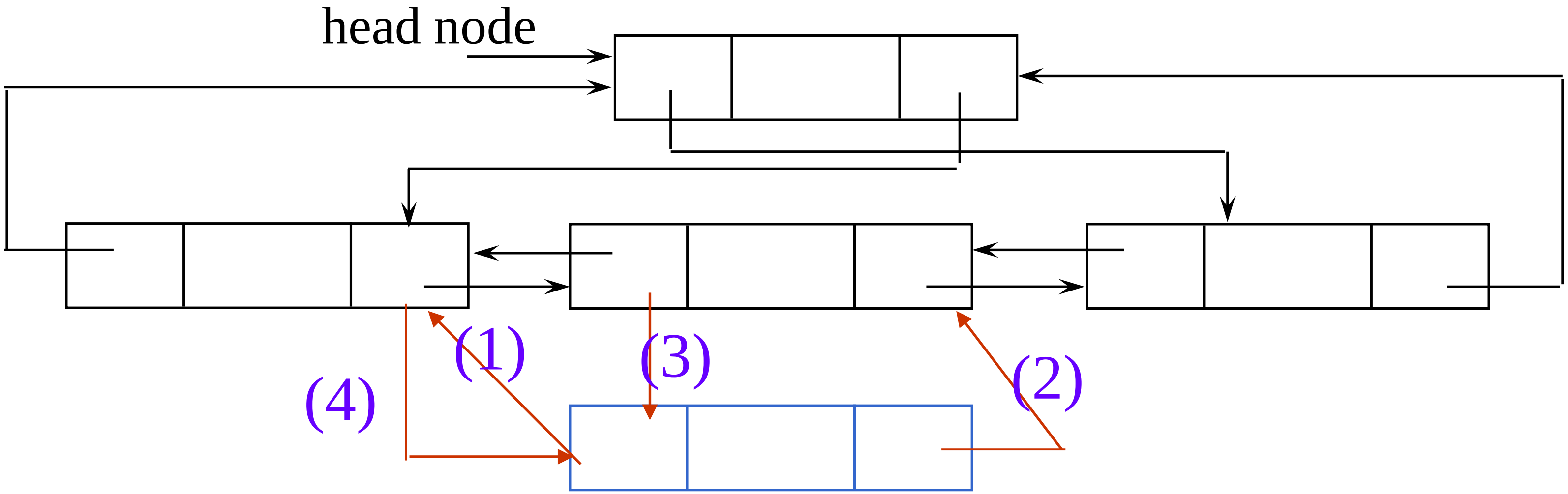
***Figure 4.22:**Empty doubly linked circular list with head node (p.188)

Insertion into an empty doubly linked circular list



Insert

```
void dinsert(nodePointer node, nodePointer newnode)
{
    (1)
    (2)
    (3)
    (4)

}
```



Delete

```
void ddelete(nodePointer node, nodePointer deleted)
{
    if (node==deleted)
       printf("Deletion of head node
                                      not permitted.\n");
    else {
        (1)
        (2)

            free(deleted);
    }
}
```