UNIT-3

Prepared by:

Manjula L, Assistant Professor

Dept. of CSE, MSRIT

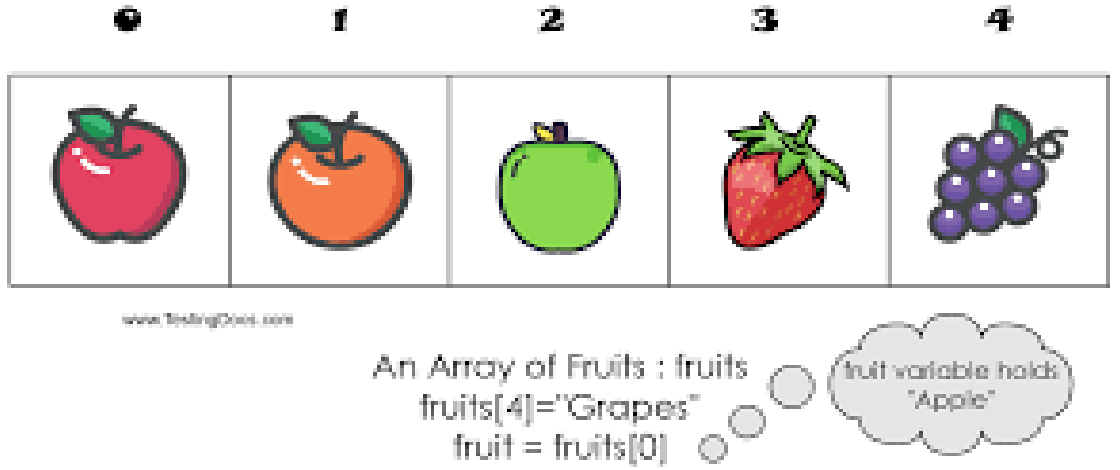
Text Book: Horowitz, Sahni, Anderson-Freed: Fundamentals of Data Structures in C, 2nd Edition, Universities Press, 2008.

Contents

- ***Linked Lists: Singly Linked lists and Chains***
- ***Representing Chains in C***
- ***Linked Stacks and Queues***
- ***Polynomials***
- ***Additional List operations***
- ***Sparse Matrices***
- ***Doubly Linked Lists***

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*
  - *– linked list*



Singly Linked Lists and Chain

	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
	.	.
	.	.
	.	.

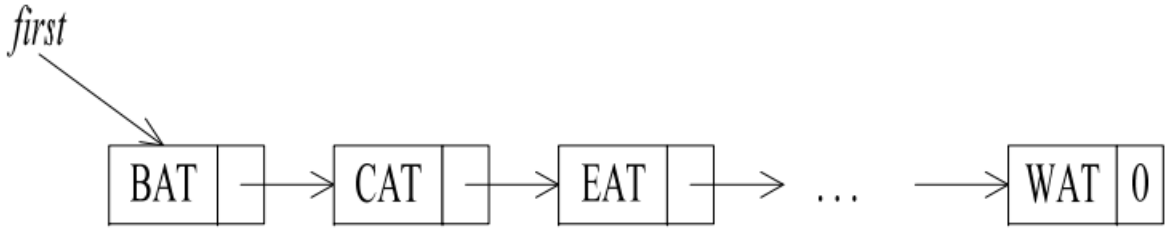
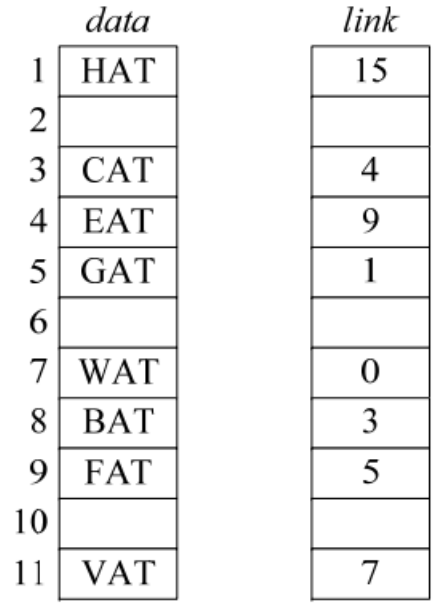
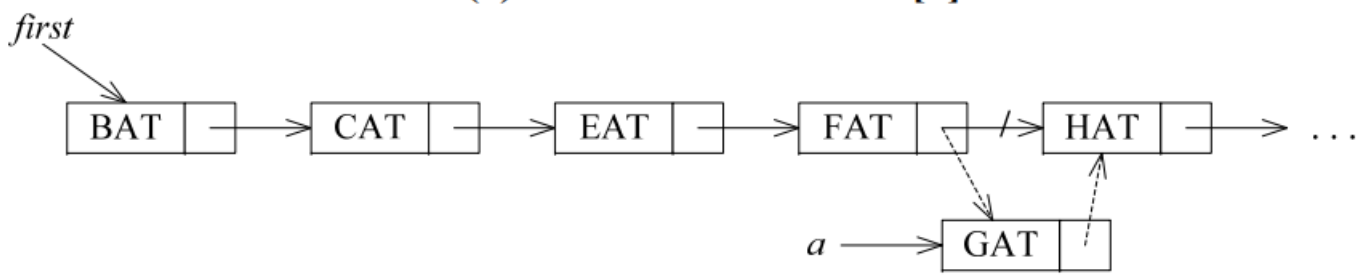


Figure 4.2: Usual way to draw a linked list (p.147)

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



(a) Insert GAT into data[5]



(b) Insert node GAT into list

Figure 4.3: Inserting into a linked list (p.148)

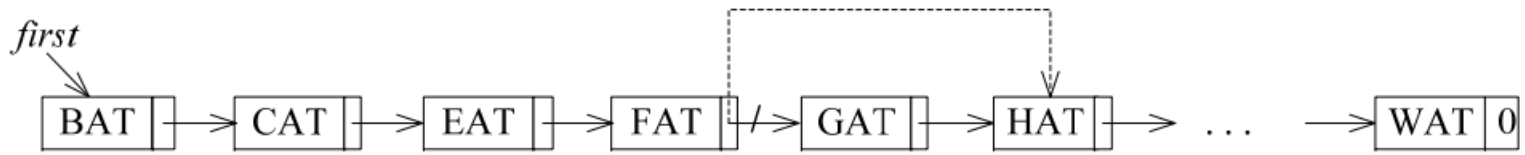
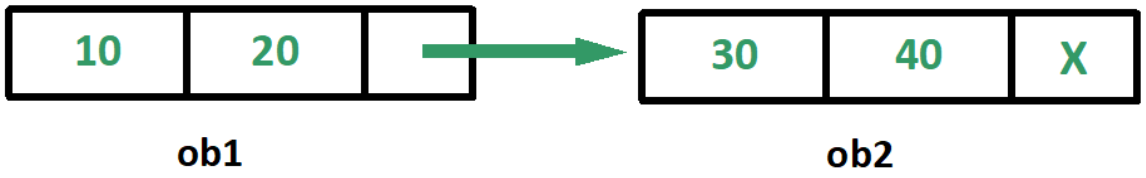
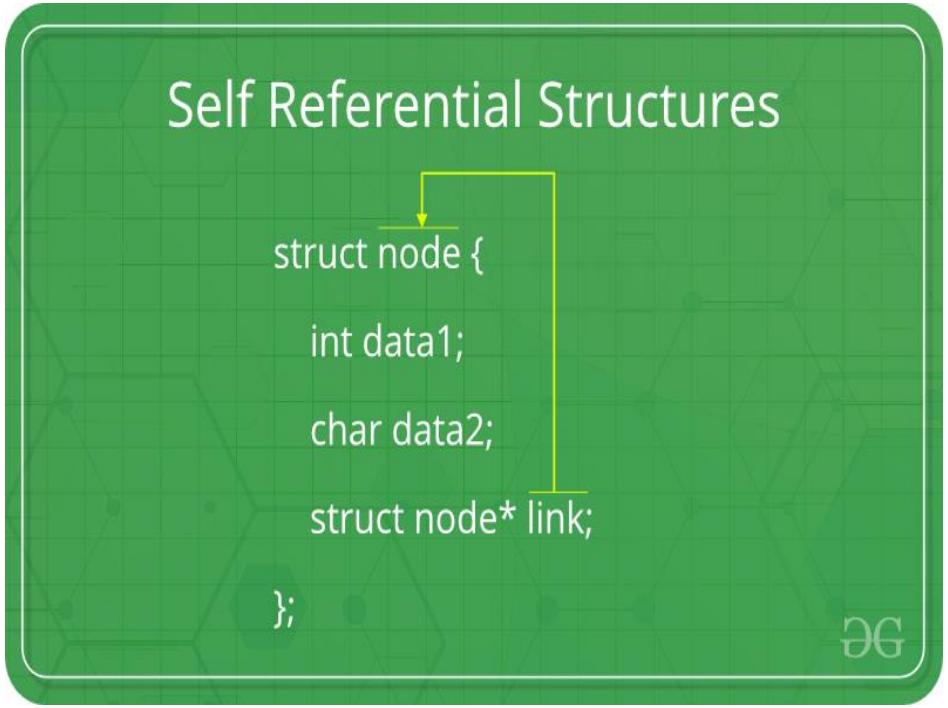


Figure 4.4: Delete GAT (p.149)

Self Referential Structure

Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.





Representing Chains in C

1. A mechanism for defining a node’s structure.
2. Way to create new node(assign memory) – malloc()
3. Remove a node – free()

Node's structure

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

struct node n1;
n1.data=10;
N1.link=NULL;
```

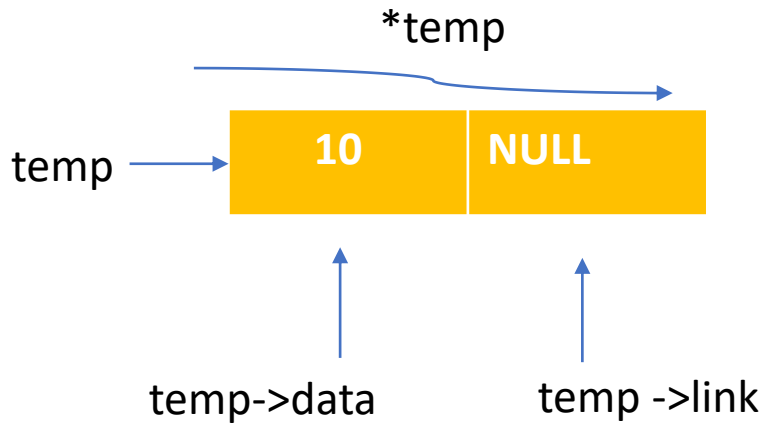
We cannot use structure variables to allocate dynamic memory. So pointer variables must be created. To create Pointers , it must be declared before structure definition

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

Note: Variables created using listPointer will all be pointer, no need to explicitly specify *

Representing Chains in C

```
#include <stdio.h>
#include<stdlib.h>
struct listNode
{
        int data;
        struct listNode *link;
};
int main()
{
        struct listNode *temp;
        temp = (struct listNode*) malloc(sizeof(struct listNode));
        temp -> data=10;
        temp -> link=NULL;
        printf("%d",temp -> data);
        return 0;
}
```



Representing Chains in C

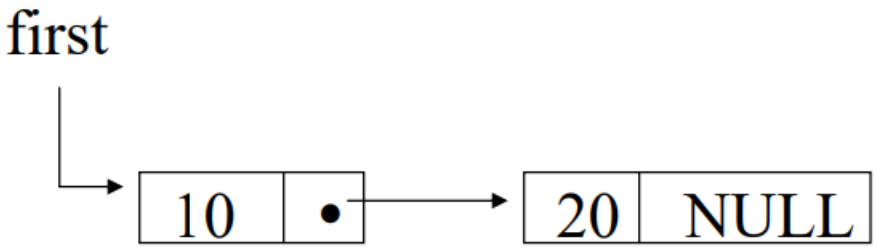
```
#include <stdio.h>
#include<stdlib.h>

typedef struct listNode *lp;
typedef struct listNode
{
        int data;
        lp link;
}Node;
```

```
int main()
{
        lp temp;
        temp = (Node*) malloc(sizeof(Node));
        temp -> data=10;
        temp -> link=NULL;
        printf("%d",temp -> data);
        return 0;
}
```

```c
#include <stdio.h>
#include<stdlib.h>

struct listNode
{
    int data;
    struct listNode *link;
};

int main()
{

    struct listNode *temp;
    temp = (struct listNode*) malloc(sizeof(struct listNode));

    temp -> data=10;
    temp -> link=NULL;

    printf("%d\n",temp -> data);
    printf("%p\n",temp);
    printf("%d\n",*temp);


    return 0;
}
```

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;
}
```



Insert into front of the list

```
void insert_front()     // f is a pointer to a pointer
{
        listPointer temp;
        temp = (Node*) malloc(sizeof(Node));
        temp->data = 50;
        if(first)
        {
                temp->link=first
                 first=temp;
        }
        else
        {
                temp->link=NULL;
                first=temp;
        }
}
```

Insert into end of the list

```
void insert_front()     // f is a pointer to a pointer
{
        listPointer temp, ptr;
        temp = (Node *) malloc(sizeof(Node));
        temp->data = 50;
        if(first)
        {
                ptr = first;
                while (ptr -> link != NULL)
                {
                        ptr = ptr -> link;
                }
                ptr->link=temp;
                temp->link=NULL;
        }
else
        {
                temp->link=NULL;
                first=temp;
        }
}
```

Insert into specified Position of the list

```
void insert_front()     // f is a pointer to a pointer
{
        listPointer ptr;
        int x;
        printf("Enter the position to be inserted");
        scanf("%d",&x)
        temp = (Node*) malloc(sizeof(Node));
        temp->data = 50;
          if(first)
        {
                ptr = first;
                for( int i=0;i<x;i++)
                {
                        ptr = ptr -> link;
                }
                if(ptr==NULL)
                {
                        Printf("Cannot insert ");
                        return;
                }
                        temp->link= ptr->link;
                        ptr->link=temp;
        }

else
        {
                temp->link=NULL;
                first=temp;
                printf("Inserted at pos 1:");
        }
}
```

Delete front of the list

```
void insert_front()
{
        listPointer ptr;
        if(first == NULL)
              printf("\nList is empty\n");
       else
        {
                ptr = first;
                first = first->link;
                free(ptr);
                printf("\nNode deleted from the begining ...\n");
        }
}
```

Delete end of the list

```
void insert_front()     // f is a pointer to
a pointer
{
        listPointer ptr1,ptr2;
        if(first == NULL)
              printf("\nList is empty\n");
         else if(first->link == NULL)
             {
                        first = NULL;
                        free(first);
             }
else
{
          ptr1 = first;
        while(ptr1->next != NULL)
        {
            ptr2 = ptr1;
            ptr1= ptr1 ->link;
        }
        ptr2->link = NULL;
        free(ptr1);
}
```

Random Delete of the list

```
void insert_front()     // f is a pointer to
a pointer
{
        listPointer ptr1,ptr2;
        int loc, i;
         printf("\n Enter the location ");
        scanf("%d",&loc);
        ptr1=first;
        if(first == NULL)
              printf("\nList is empty\n");
         else
for(i=0;i<loc;i++)
  {
        ptr2 = ptr1;
        ptr 1= ptr1->link;

       if(ptr1== NULL)
      {
         printf("\nCan't delete");
         return;
      }
   }
        ptr2 ->next = ptr1 ->next;
        free(ptr1);
}
```

Insert into front of the list

Let first be a pointer to a linked list. If the link is empty first should be changed , so address of it is send .

listPointer first = NULL;

For insert after x node the function call is:

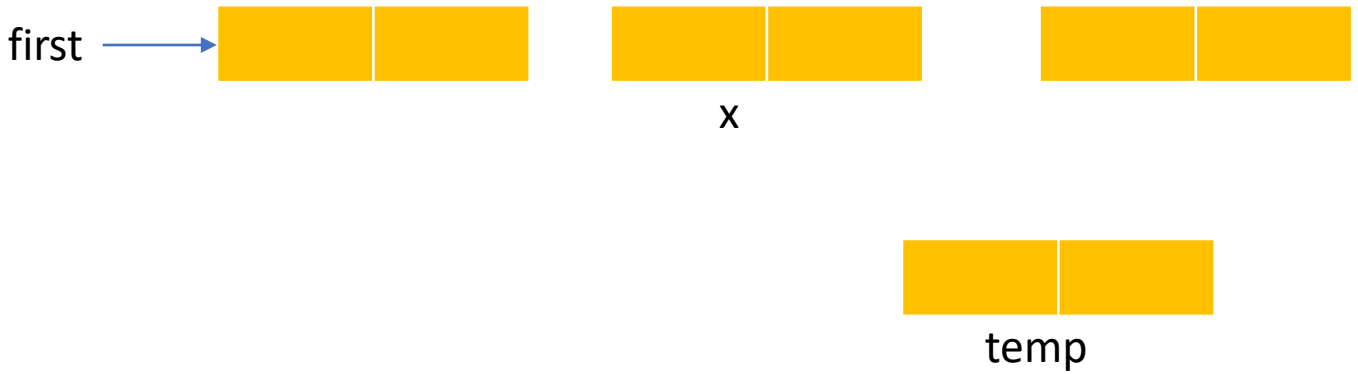
insert(&first, x);

Function definition must have pointer to pointer:

void insert (listPointer *f, listPointer x)

Insert into front of the list

Let first be a pointer to a linked list. If the link is not empty



temp->link=x->link

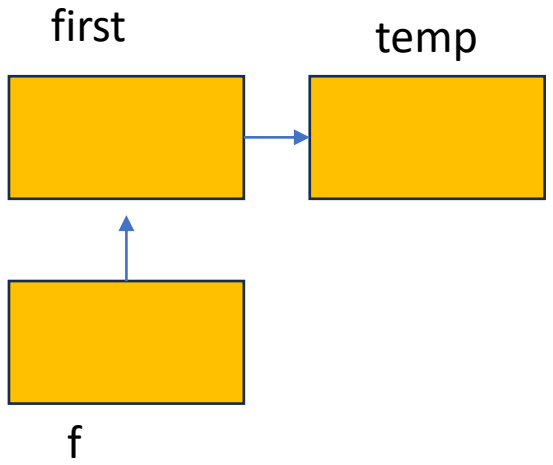
x->link=temp;

Insert into front of the list

If the link is empty , first will be NULL

f= &first;

*f = * (&first) = first



if (*f=NULL)

***f=temp;**

```
void insert(listPointer *f, listPointer x)     // f is a pointer to a pointer
{
        listPointer temp;
        temp = (Node*) malloc(sizeof(Node));
        temp->data = 50;
        if(*f)
        {
                temp->link=x->link
                 x->link=temp;
        }
        else
        {
                temp->link=NULL;
                *f=temp;
        }
}
```

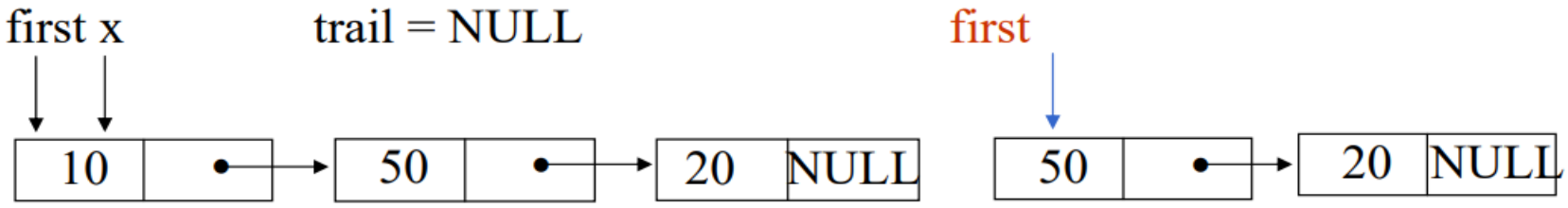
List Deletion

Assume there are three pointers:

- first = start of the list
- trail = Points to the node that precedes it
- x= points the node to be deleted

Delete the first node.

delete (& first, NULL, first)

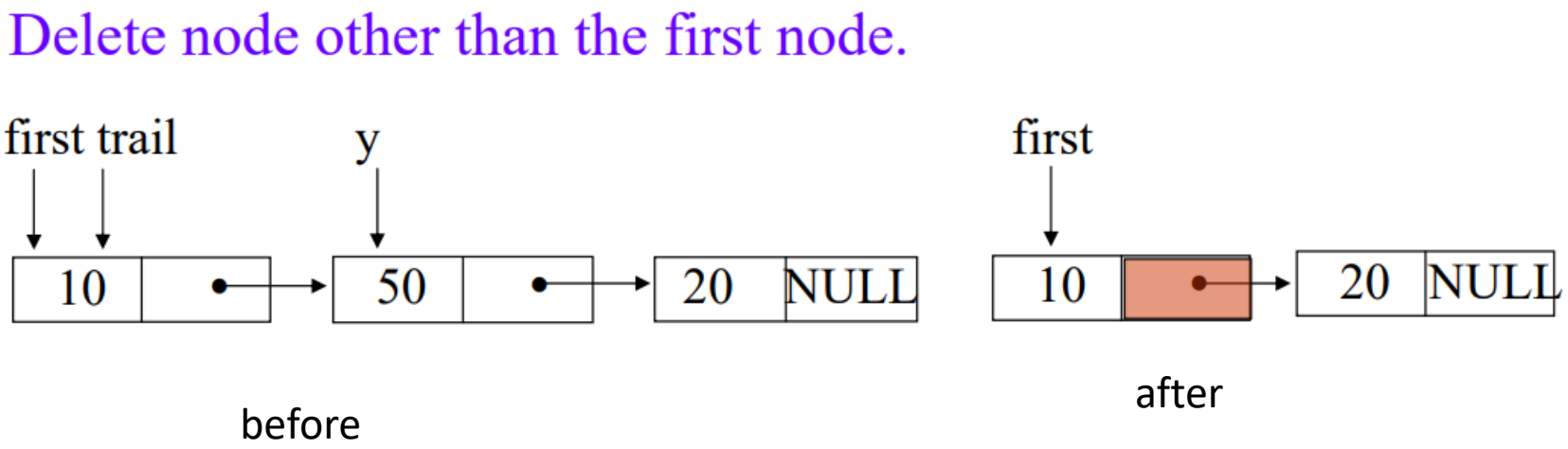


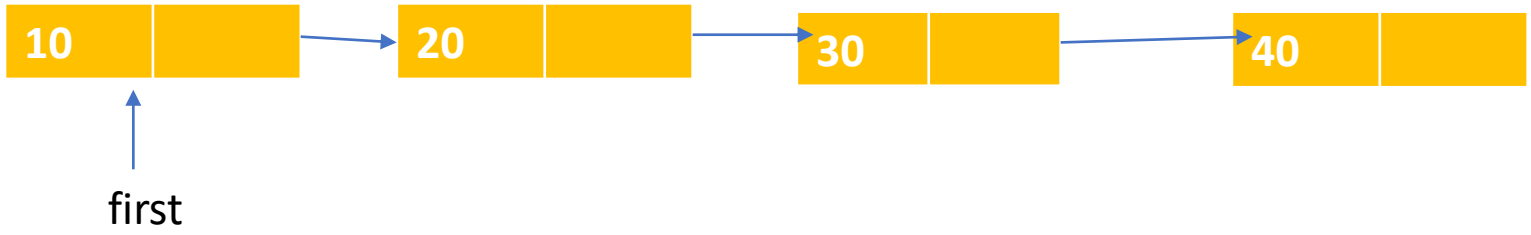
(a) before deletion

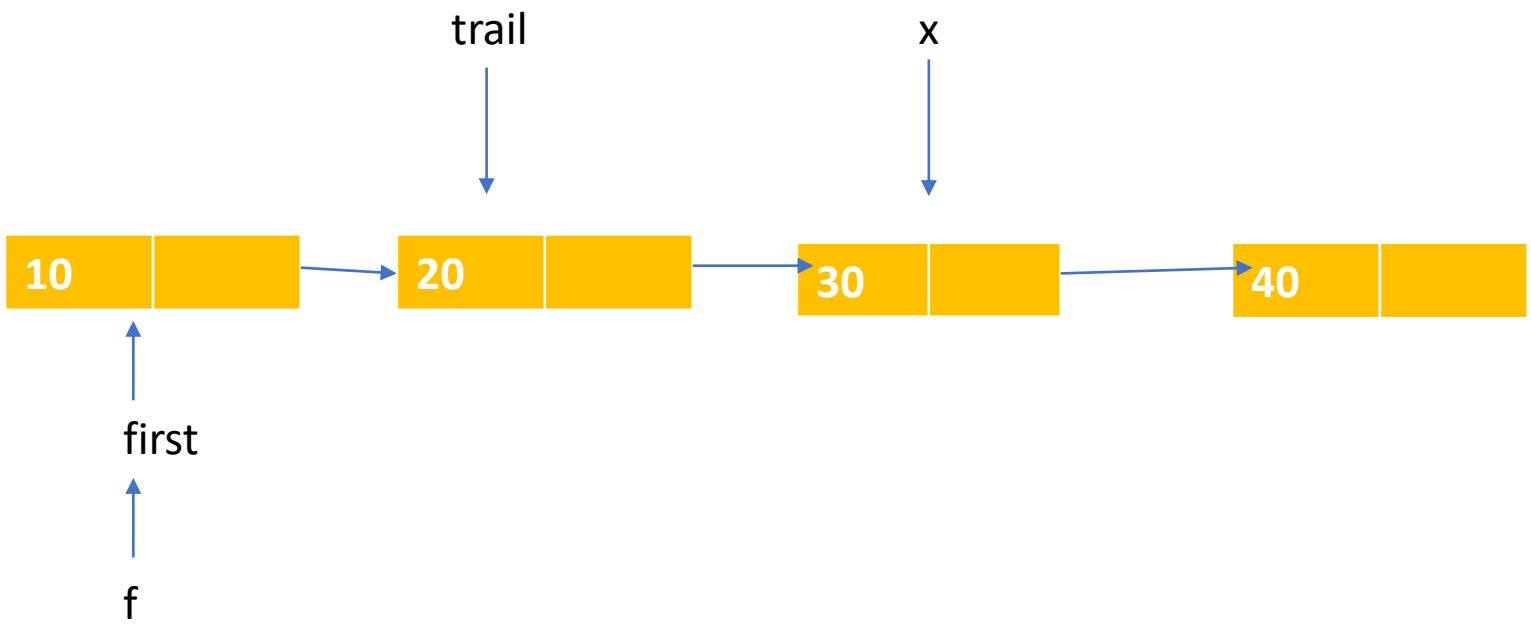
(b)after deletion

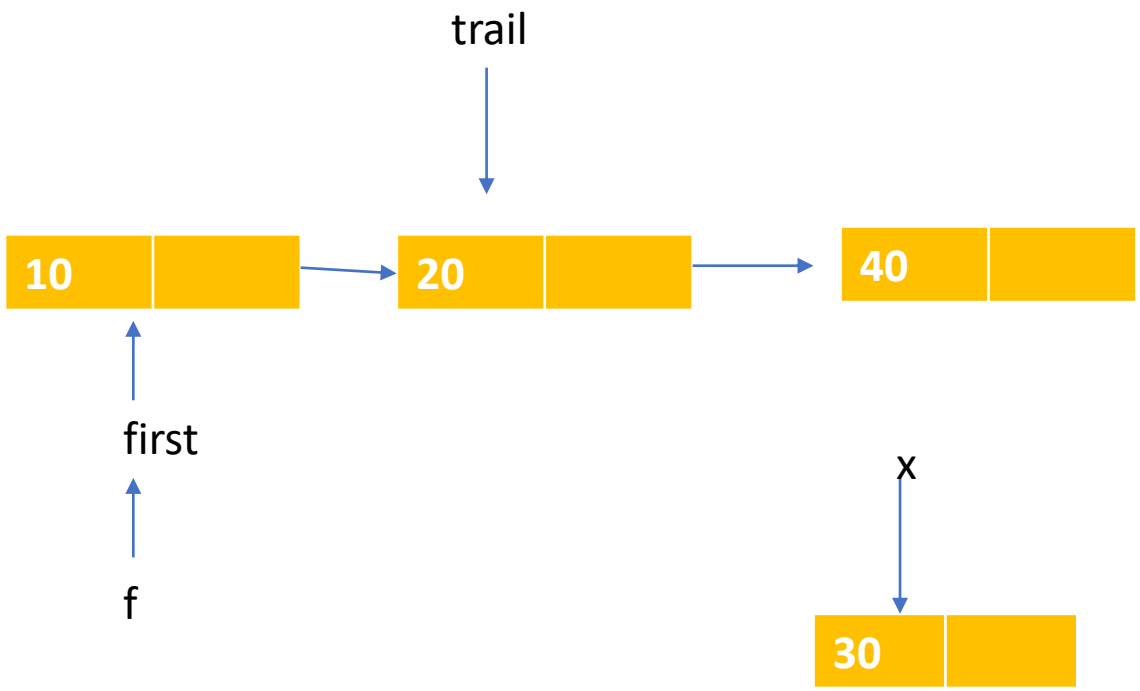
List Deletion

delete(&first, first, y)




10
20
30
40
first

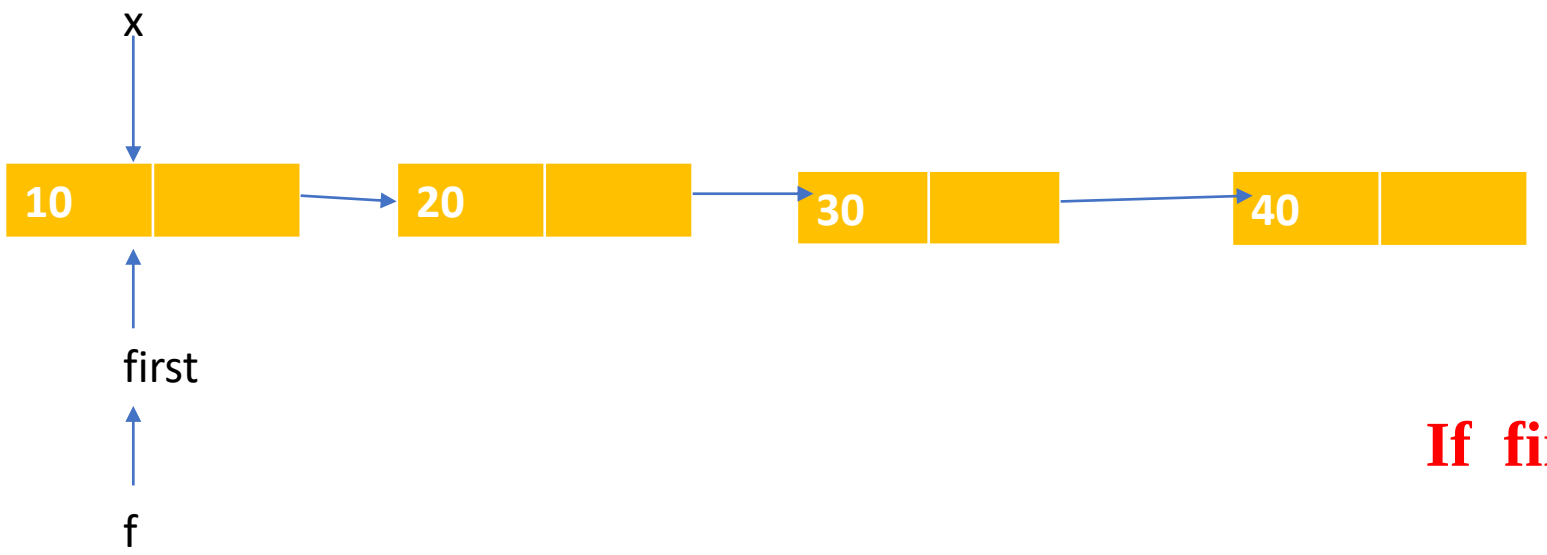

trail
x
10
20
30
40
first
f



If not first node:

trail->link = x->link

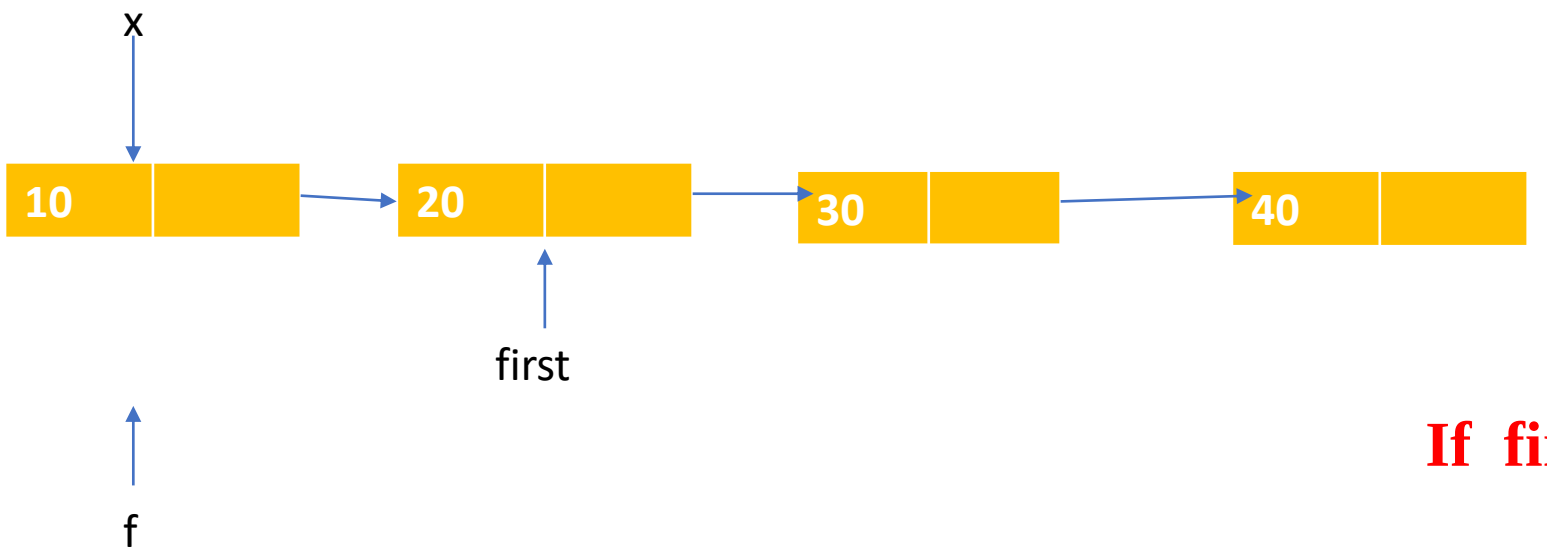
trail = NULL



If first node:

then trail will be NULL

trail = NULL



If first node:

then trail will be NULL

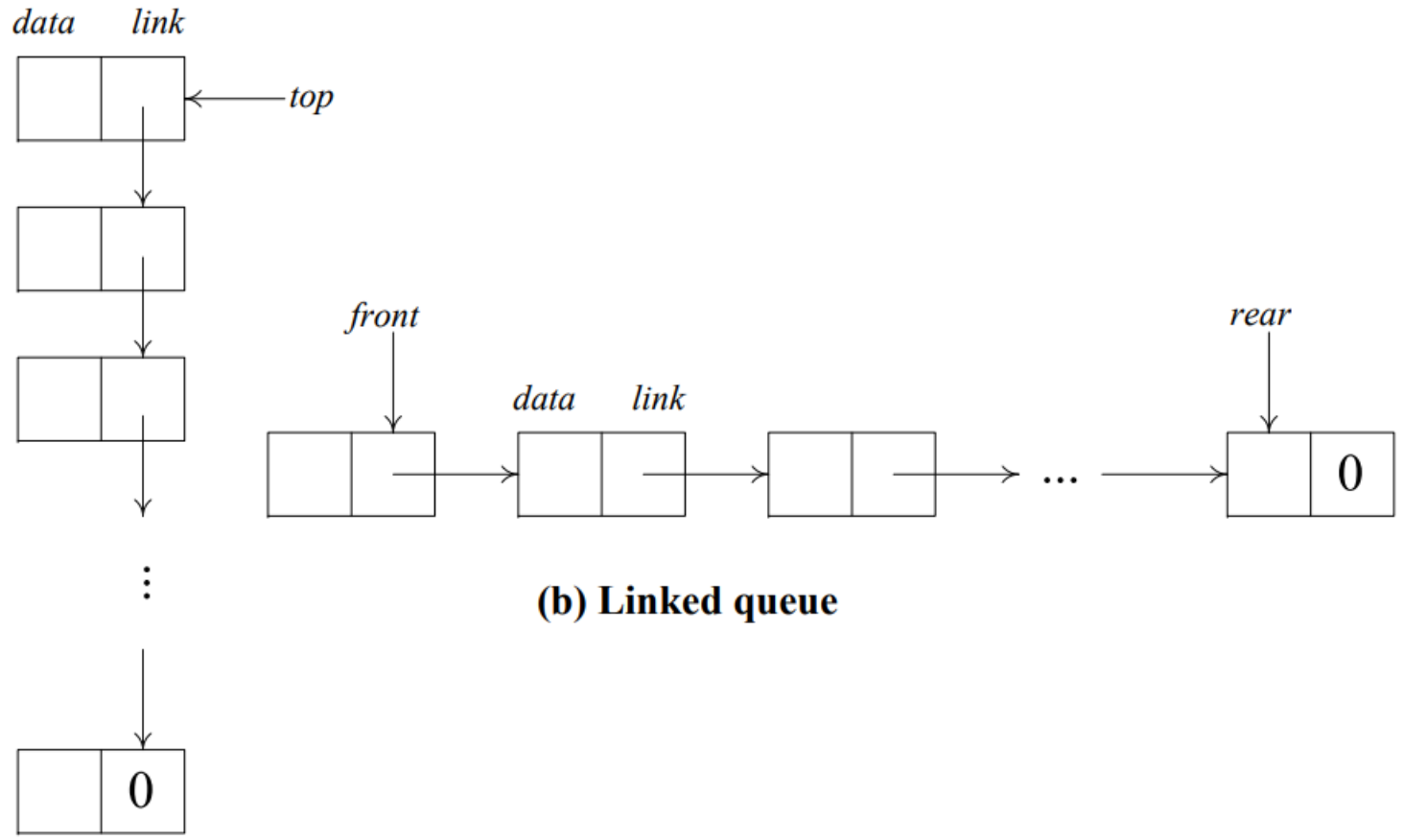
***f = (*f)->link**
***(&first) = (*&first)->link**
first=first ->link

```
void delete(listPointer *f , listPointer trail, listPointer x)
{
        if(trail)
                trail>link=x->link
        else
                *f= (*f)->link;
        free(x);
}
```

List Printing

```
void printList (listPointer f)
{
        printf(“The list contains”);
        for (; f; f= f->link)
                printf(“%d”, f->data)
        printf(“\n”);
}
```

Linked Stacks and Queues



(b) Linked queue

(a) Linked stack

Polynomial Addition : Implementation

```
MAX_TERMS 100 /* size of terms array */
typedef struct {
        float coef;
        int expon;
        } polynomial;
polynomial terms[MAX_TERMS];
int avail = 0;
```

Polynomial Addition : Implementation

```
void padd (int starta, int finisha, int startb, int finishb, int * startd, int *finishd)
{
/* add A(x) and B(x) to obtain D(x) */
    float coefficient;
   *startd = avail;
   while (starta <= finisha && startb <= finishb)
     switch (COMPARE(terms[starta].expon,
                                terms[startb].expon)) {
      case -1: /* a expon < b expon */
               attach(terms[startb].coef, terms[startb].expon);
               startb++;
               break;
```

Polynomial Addition : Implementation

```
case  0: /* equal exponents */
            coefficient = terms[starta].coef + terms[startb].coef;
            if (coefficient)
               attach (coefficient, terms[starta].expon);
            starta++;
            startb++;
            break;
case 1: /* a expon > b expon */
         attach(terms[starta].coef, terms[starta].expon);
         starta++;
}
```

Polynomial Addition : Implementation

```
/* add in remaining terms of  A(x) */
for( ; starta <= finisha; starta++)
    attach(terms[starta].coef, terms[starta].expon);
/* add in remaining terms of B(x) */
for( ; startb <= finishb; startb++)
   attach(terms[startb].coef, terms[startb].expon);
*finishd =avail -1;
}
```

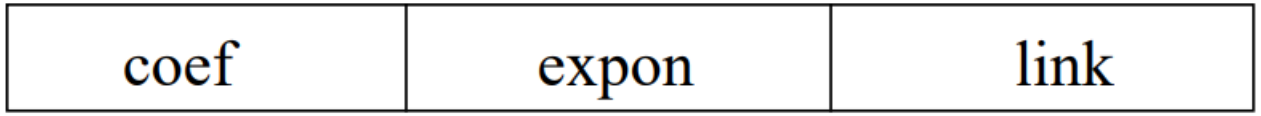
Polynomial Addition : Implementation

```
void attach(float coefficient, int exponent)
{
/* add a new term to the polynomial */
    if (avail >= MAX_TERMS) {
       fprintf(stderr, “Too many terms in the polynomial\n”);
       exit(1);
      }
      terms[avail].coef  = coefficient;
      terms[avail++].expon = exponent;
}
```

Polynomials implementation using linked list

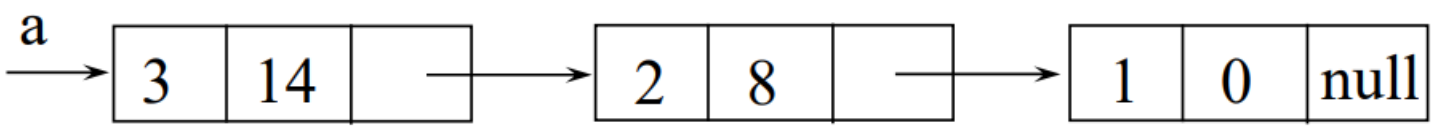
$$A(x) = a_{m-1} x^{e_{m-1}} + a_{m-2} x^{e_{m-2}} + \ldots + a_0 x^{e_0}$$

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

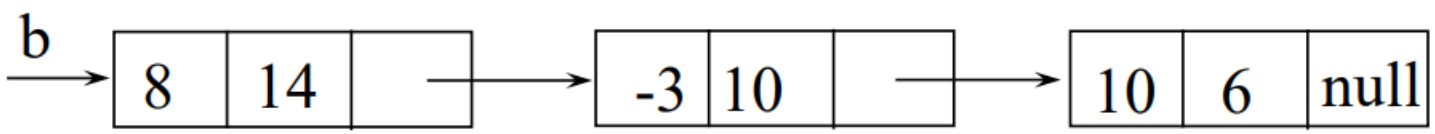


Examples

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



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



```
poly_pointer padd(polyPointer a, polyPointer b)
{
        polyPointer rear, temp;
        int sum;
        rear =(polyPointer)malloc(sizeof(polyNode));
        if (IS_FULL(rear)) {
                fprintf(stderr, “The memory is full\n”);
                exit(1);
        }
        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
```

```
void attach(float coefficient, int exponent, polyPointer *ptr)//insert at end
{
        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;
}
```

Erase Polynomials

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

Invert Single Linked Lists

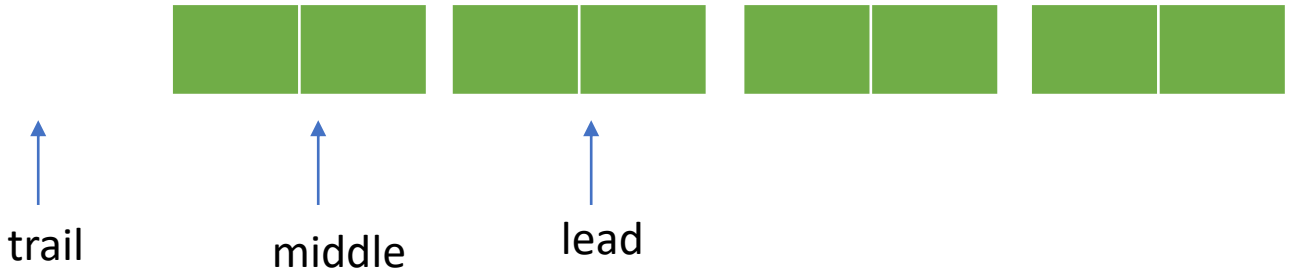
```
listPointer invert(listPointer lead)
{
        listPointer middle, trail;
        middle = NULL;
        while (lead)
        {
                trail = middle;
                middle = lead;
                lead = lead->link;
                middle->link = trail;
        }
        return middle;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

Invert Single Linked Lists

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

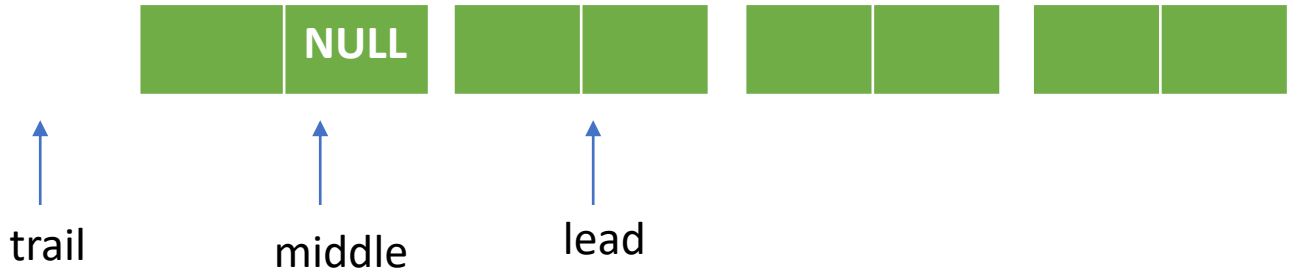
Invert Single Linked Lists

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



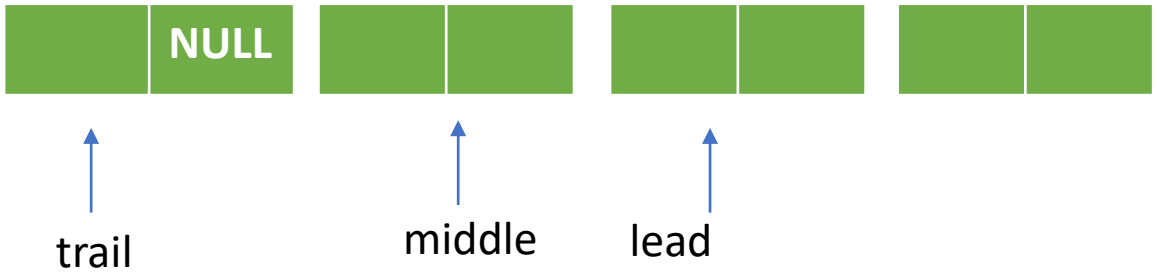
Invert Single Linked Lists

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



Invert Single Linked Lists

```
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;
}
```

Maintain an Available List

```
polyPointer getNode(void)
{
        polyPointer node;
        if (avail)
        {
                node = avail;
                avail = avail->link:
        }
        else
        {
                node = (polyPointer)malloc(sizeof(polyNode));
                if (IS_FULL(node))
                {
                        printf(stderr, “The memory is full\n”); exit(1);
                }
        }
        return node;
}
```

Return node – function

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

Circularly Linked Lists

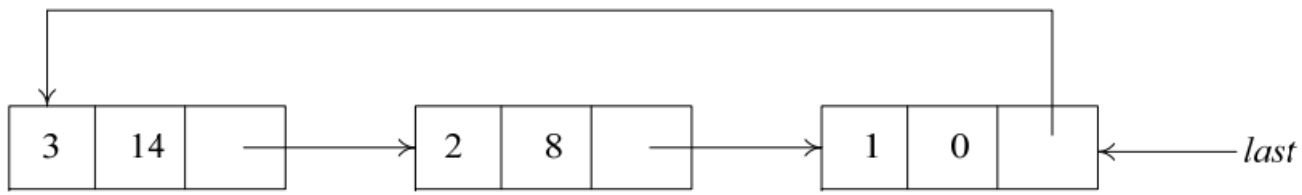
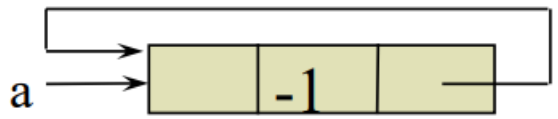


***Figure 4.14:**Circular representation of $3x^{14} + 2x^{8} + 1$ (p.166)

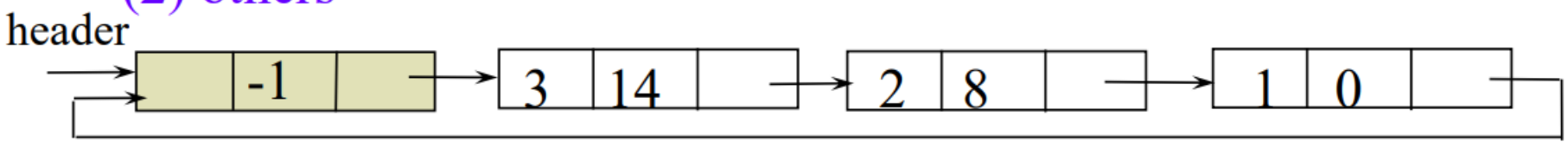
Represent polynomial as circular list.

(1) zero



Zero polynomial

(2) others



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

```
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;
```

```
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;
}
```

Erase the node ptr

```
void cerase(polyPointer *ptr)
{
        polyPointer temp;
        if (*ptr) {
                temp = (*ptr)->link;
                (*ptr)->link = avail;
                avail = temp;
                *ptr = NULL; }
}
```

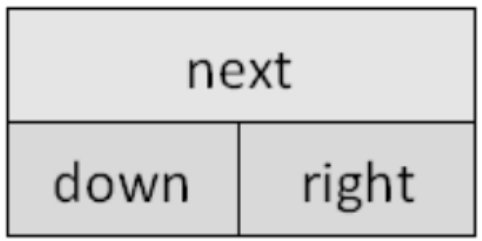
Sparse Matrices

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

Revisit Sparse Matrices

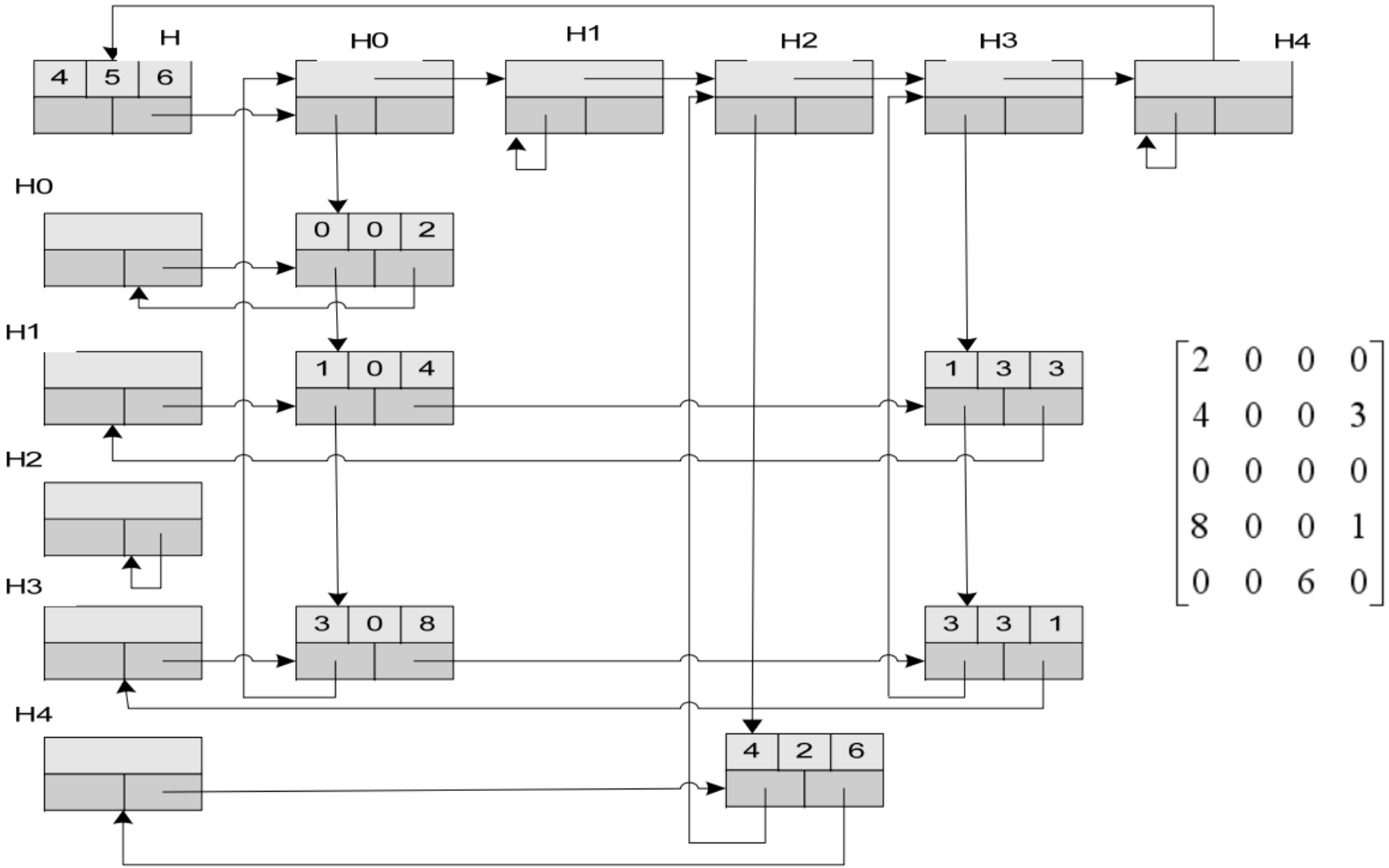
of head nodes = max{# of rows, # of columns}

head node



entry node

Linked Representation for Matrix



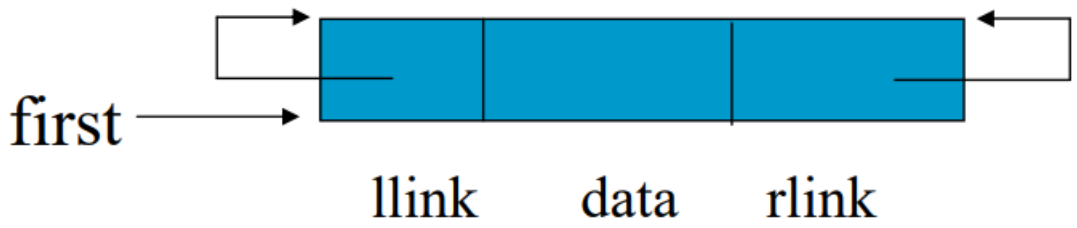
$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

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

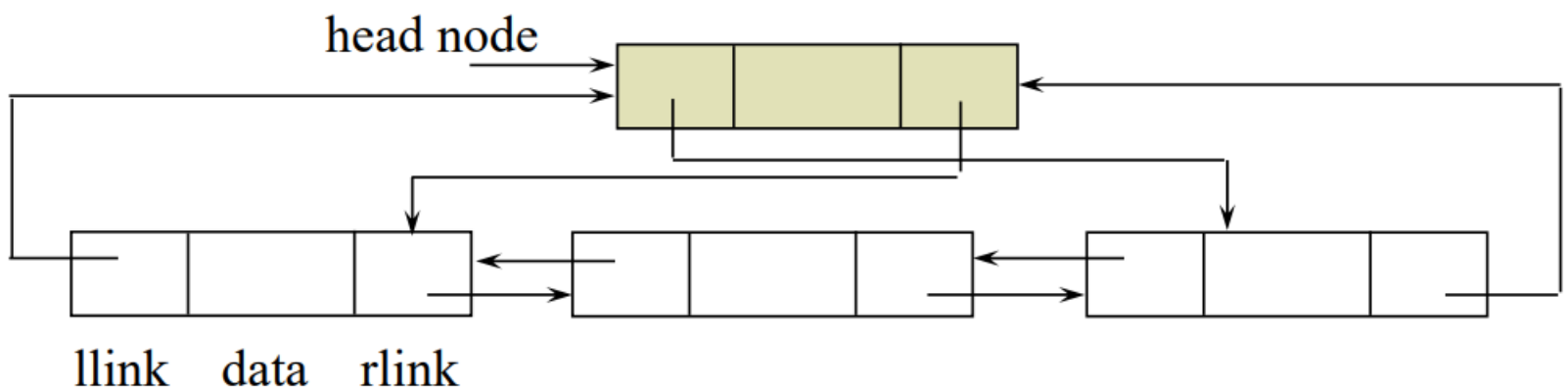
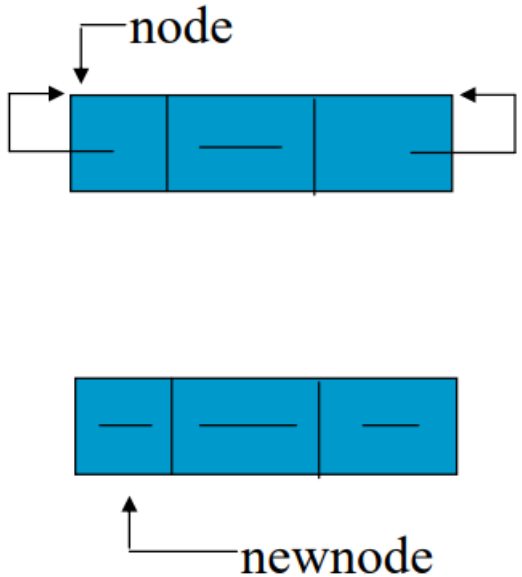
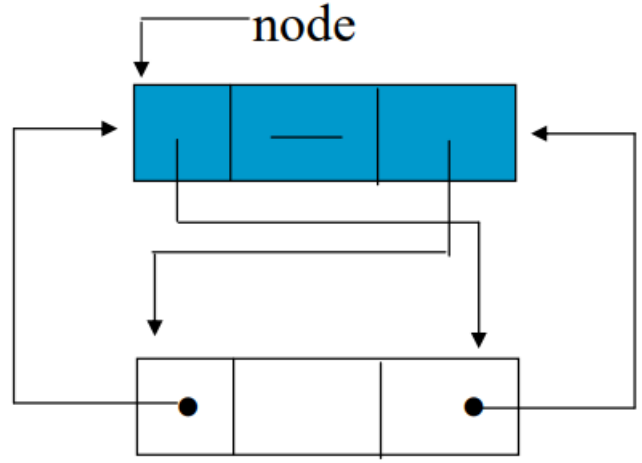


***Figure 4.22:**Empty doubly linked circular list with head node (p.188)

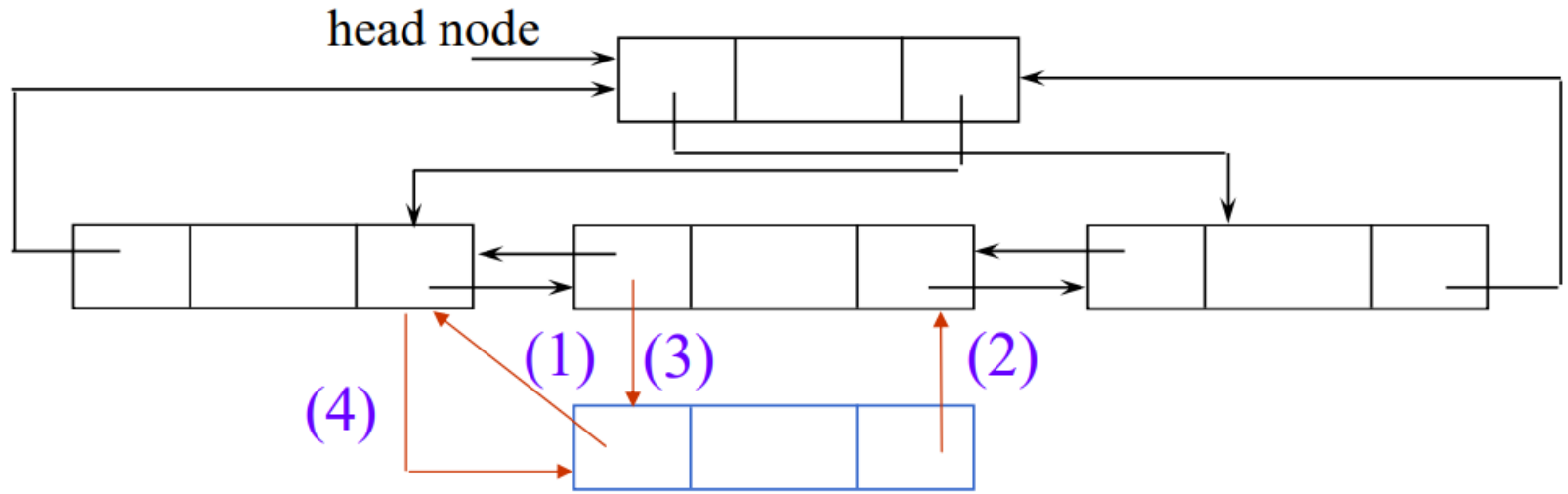
Insertion into an empty doubly linked circular list





Insert

```
void dinsert(nodePointer node, nodePointer newnode)
{
     (1)  newnode->llink = node;
     (2)  newnode->rlink = node->rlink;
     (3)  node->rlink->llink = newnode;
     (4)  node->rlink = newnode;
}
```



Delete

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