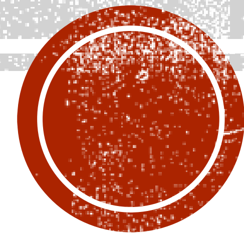
LINKED LISTS

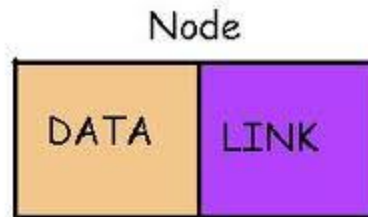


SEQUENTIAL REPRESENTATION

- Array representation : has a fixed size.
- Successive items of a list are located a fixed distance apart.
- Arbitrary insertions and deletions from list can be expensive/time-consuming.
- Additional difficulty when we used several ordered lists of varying size.
  exp. multiple stacks, multiple queues.

LINKED LISTS

- Successive items may be placed anywhere in memory.



INSERTION - DELETION

(a)
20 → 45 → 51 → 76 → 84

(b)
20 → 45 → 51 → 76 → 84
Old value
60
Inserted item

(c)
20 → 45 → 51 → 60 → 76 → 84
Deleted item

POINTER-BASED LINKED LISTS

- A node in a linked list is usually a `struct`

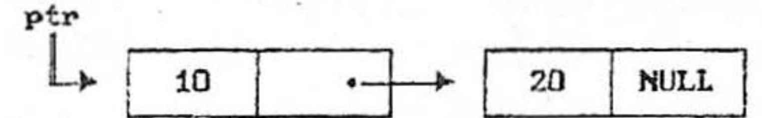
```
typedef struct list_node *list_pointer;
typedef struct list_node{
    int item;
    list_pointer link;
};
list_pointer ptr = NULL;     //the address of the start of the list
```

- A node is dynamically allocated

    ptr = (list_pointer)malloc(sizeof(list_node));

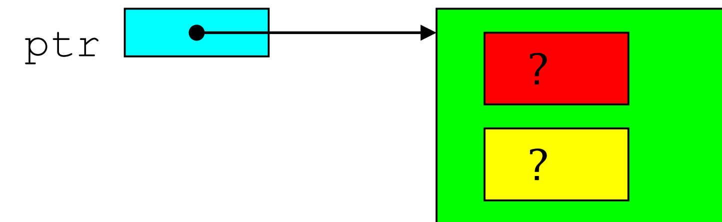
A TWO-NODE LIST

```
list_pointer create2()
{
        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 = 20;
        first->link = second;
        return first;
}
```



DYNAMIC MEMORY OPERATORS IN C

```
struct node{
    int data;
    struct node *next;
};
struct node *ptr;

ptr = (struct node *)     /*type casting */
      malloc(sizeof(struct node));
```



THE FREE OPERATOR IN C

- Function *free* deallocates memory- i.e. the memory is returned to the system so that the memory can be reallocated in the future.

free(ptr);

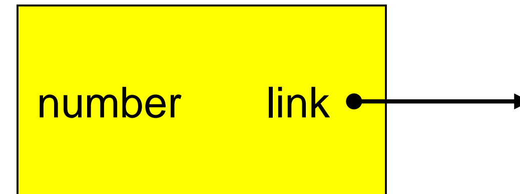
ptr

THE NODES OF A LINKED LIST

- A node in a linked list is a structure that has at least two fields. One of the fields is a data field; the other is a pointer that contains the address of the next node in the sequence.

- A node with one data field:

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



AN EXAMPLE

A node with three data fields:

```
struct student{
   char name[20];
   int id;
   double grdPts;
   struct student *next_student;
}
```

BASIC OPERATIONS

1. Add a node.
2. Delete a node.
3. Search for a node.
4. Traverse (walk) the list. Useful for counting operations or aggregate operations.

ADDING

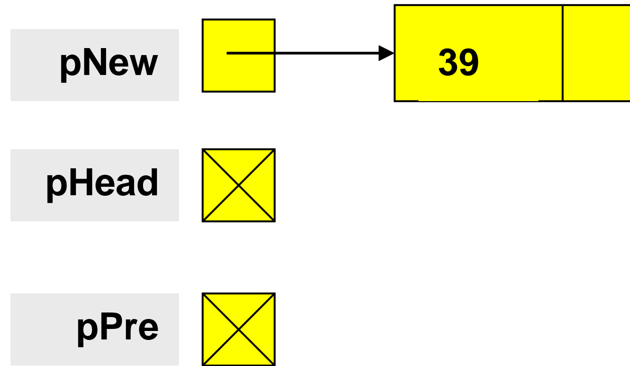
There are four steps to add a node to a linked list:

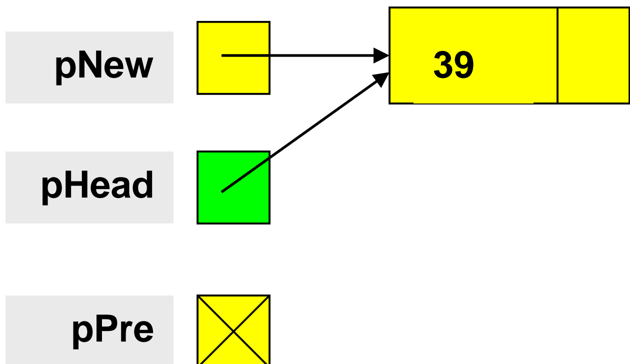
- Allocate memory for the new node.
- Determine the insertion point (you need to know only the new node's predecessor (`pPre`)
- Point the new node to its successor.
- Point the predecessor to the new node.

Pointer to the predecessor (`pPre`) can be in one of two states:

- it can contain the address of a node (i.e. you are adding somewhere after the first node – in the middle or at the end)
- it can be NULL (i.e. you are adding either to an empty list or at the beginning of the list)

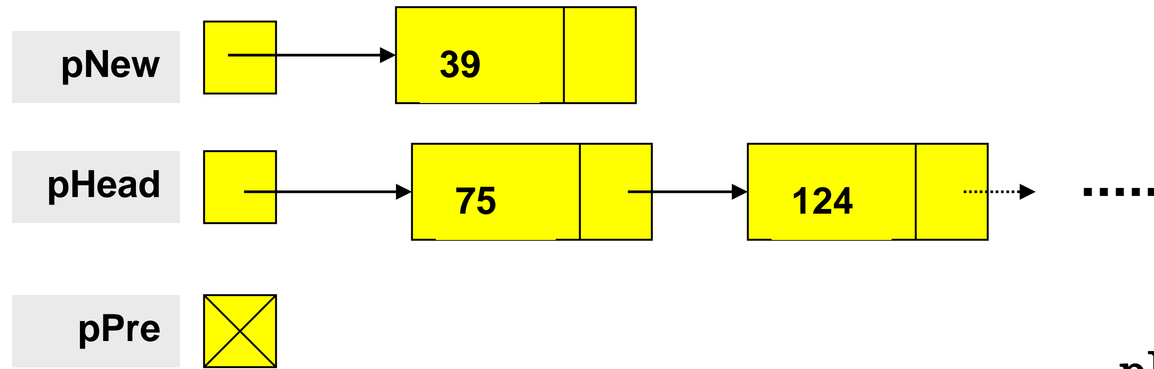
ADDING A NODE TO AN EMPTY LIST



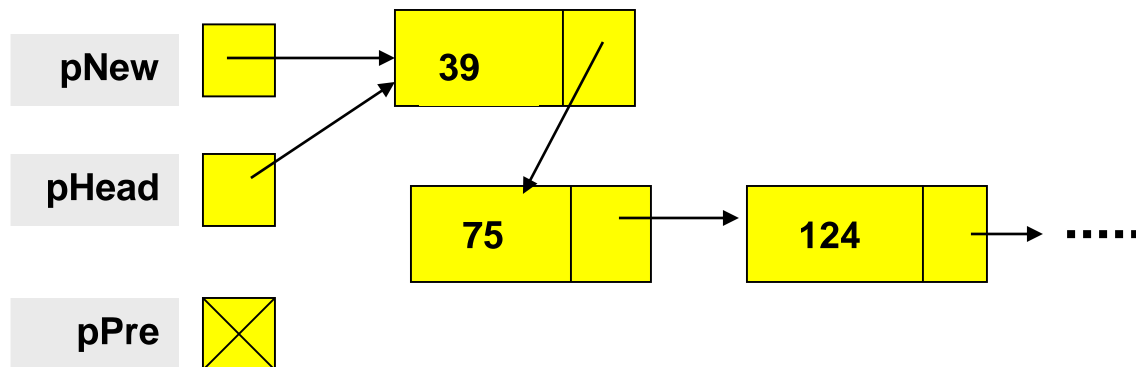


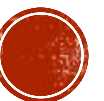
pNew -> next = pHead; // set link to NULL
pHead = pNew;// point list to first node

ADDING A NODE TO THE BEGINNING OF A LIST

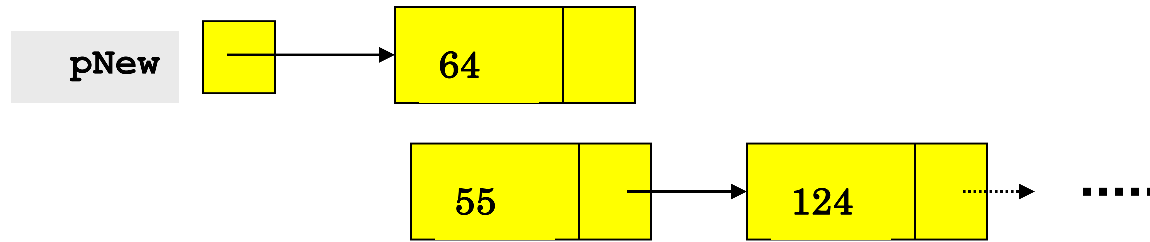


pNew -> next = pHead; // set link to NULL
pHead = pNew;// point list to first node

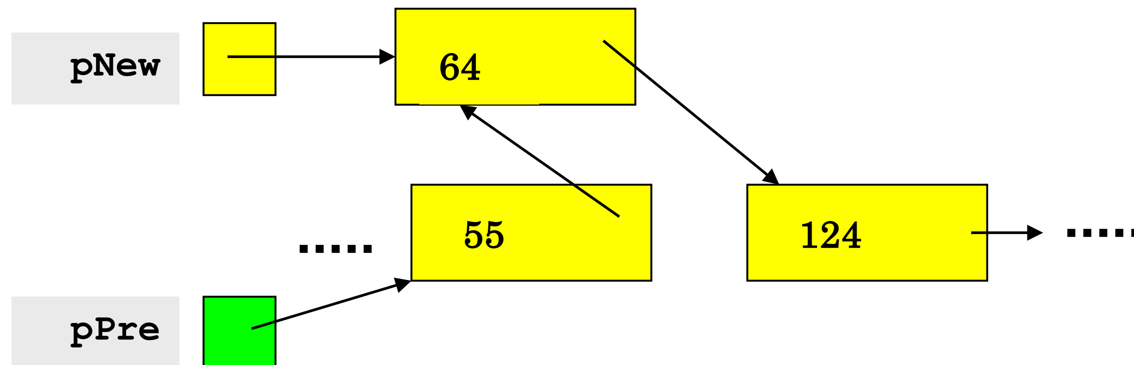


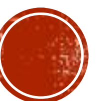


ADDING A NODE TO THE MIDDLE OF A LIST

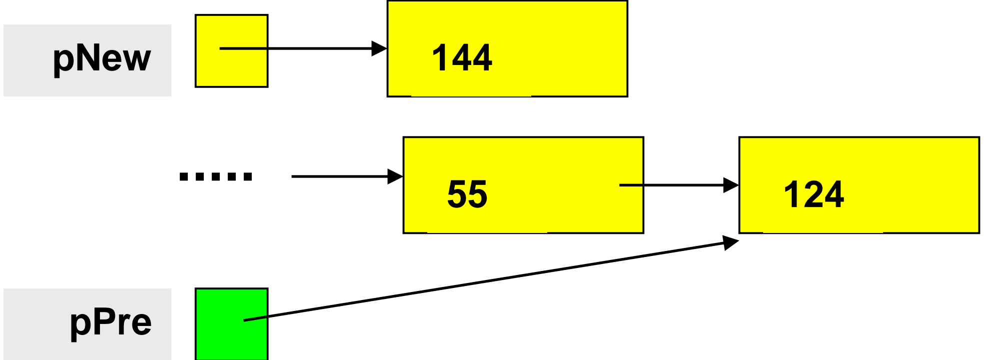


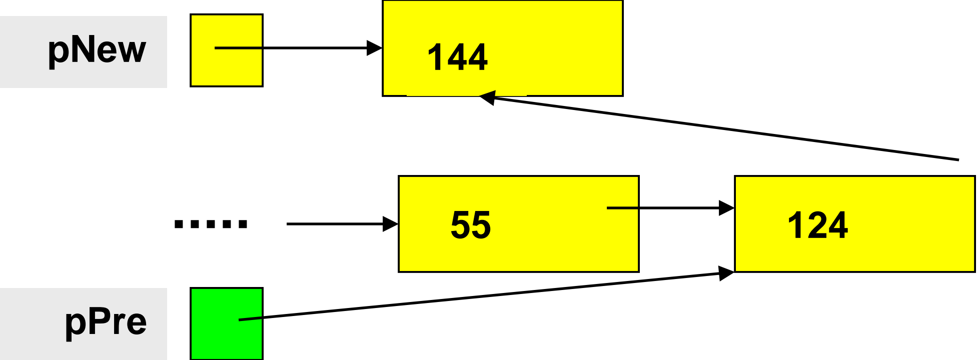
```
pNew -> next = pPre->next;
pPre->next = pNew;
```





ADDING A NODE TO THE END OF A LIST






```
pNew -> next = NULL;
pPre->next = pNew;
```

INSERTING

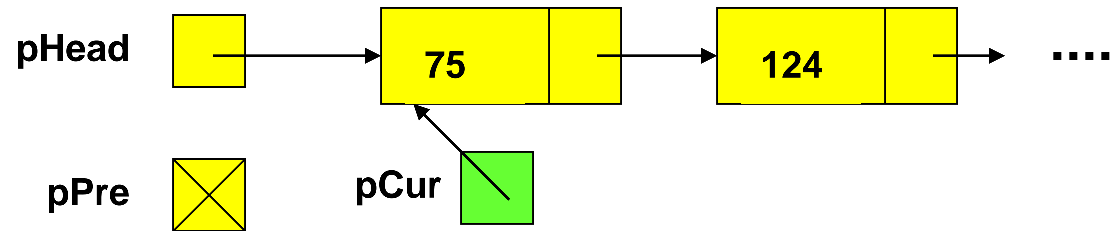
Given the head pointer (pHead), the predecessor (pPre) and the data to be inserted (item). Memory must be allocated for the new node (pNew) and the links properly set.

```
struct node *pNew;
pNew = (struct node *)  malloc(sizeof(struct node));
pNew -> data = item;
if (pPre == NULL){
     //add before first logical node or to an empty list
     pNew -> next = pHead;
     pHead = pNew;
}
else {
      //add in the middle or at the end
     pNew -> next = pPre -> next;
     pPre -> next = pNew;
}
```

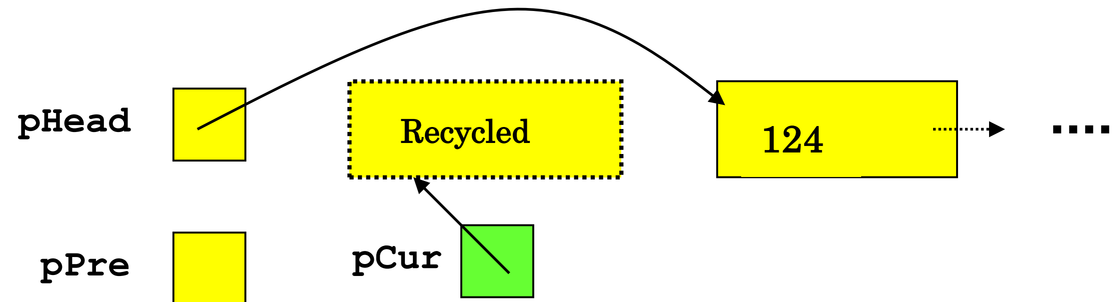
DELETING

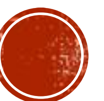
- Deleting a node requires that we logically remove the node from the list by changing various links and then physically deleting the node from the list (i.e., return it to the heap).
- Any node in the list can be deleted. Note that if the only node in the list is to be deleted, an empty list will result. In this case the head pointer will be set to NULL.
- To logically delete a node:
  - First locate the node itself (pCur) and its logical predecessor (pPre).
  - Change the predecessor's link field to point to the deleted node's successor (located at pCur -> next).
  - Recycle the node using the free() function.

DELETING THE FIRST NODE FROM A LINKE LIST

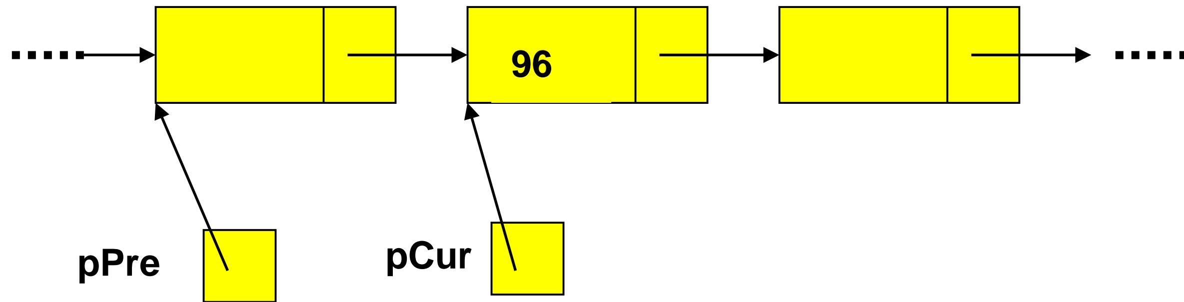


pHead = pCur->next;
free(pCur);

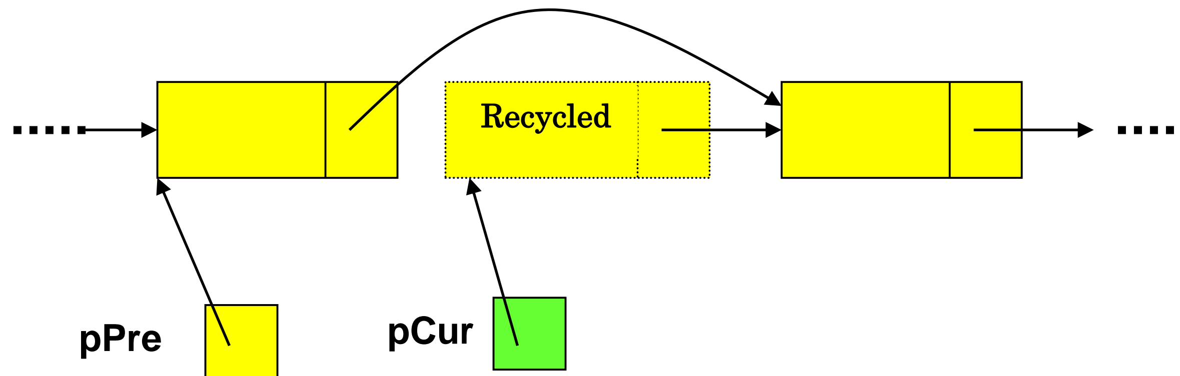




DELETING A NODE FROM A LINKED LIST



pPre->next = pCur->next;
free(pCur);



DELETING

- Given the head pointer (pHead), the node to be deleted (pCur), and its predecessor (pPre), delete pCur and free the memory allocated to it.

```
//delete a node from a linked list
if (pPre == NULL)
     //deletion is on the first node of the list
    pHead = pCur -> next;
else
    //deleting a node other than the first node of the list
    pPre -> next = pCur -> next;
free(pCur)
```

SEARCHING A LINKED LIST

- Notice that both the insert and delete operations on a linked list must search the list for either the proper insertion point or to locate the node corresponding to the logical data value that is to be deleted.

```
//search the nodes in a linked list
pPre = NULL;
pCur = pHead;
//search until the target value is found or the end of the list is reached
while (pCur != NULL && pCur -> data != target) {
    pPre = pCur;
    pCur = pCur -> next;
}
//determine if the target is found or ran off the end of the list
if (pCur != NULL)
    found = 1;
else
    found = 0;
```

TRAVERSING A LINKED LIST

- List traversal requires that all of the data in the list be processed. Thus each node must be visited and the data value examined.

```
//traverse a linked list
Struct node *pWalker;
pWalker = pHead;
printf("List contains:\n");
while (pWalker != NULL){
   printf("%d ", pWalker -> data);
   pWalker = pWalker -> next;
}
```

POINTER-BASED LINKED LISTS

- A node in a linked list is usually a `struct`

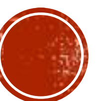
```
typedef struct list_node *list_pointer;
typedef struct list_node{
    int item;
    list_pointer link;
};
list_pointer ptr = NULL;     //the address of the start of the list
```

- A node is dynamically allocated

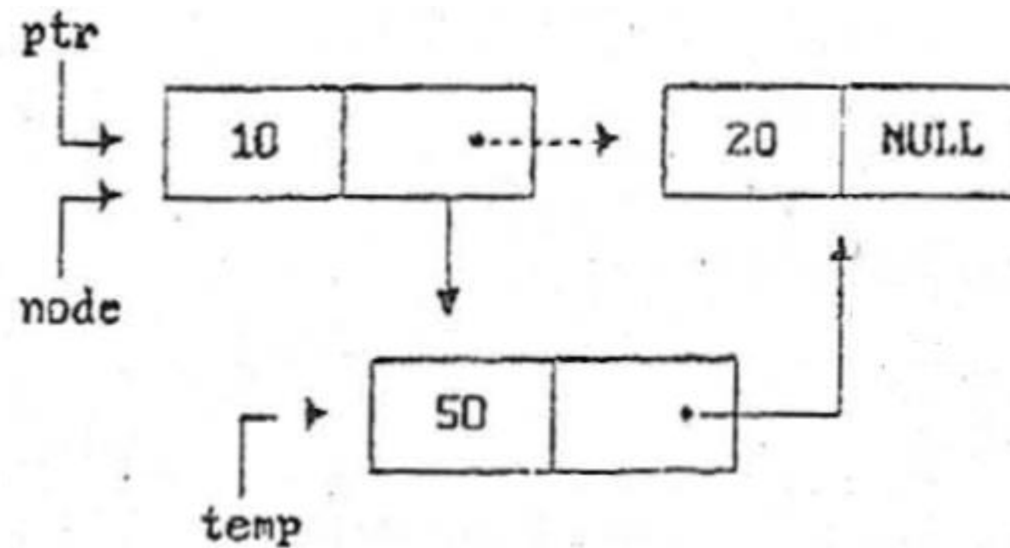
```
ptr = (list_pointer)malloc(sizeof(list_node));
```

INSERT INTO THE FRONT OF A LIST

```
void insert(list_pointer *ptr, list_pointer node)
{
   //insert a new node with data = 50 into the list ptr (after).
   list_pointer temp = (list_pointer)malloc(sizeof(list_node));
   if(IS_FULL(temp))
           exit(1);   //the memory is full
   temp->data = 50;
   if(*ptr){
           temp->link = node->link;
           node->link = temp;
   }
   else{
           temp->link = NULL;
           *ptr = temp;
   }
}
```



AFTER INSERTION

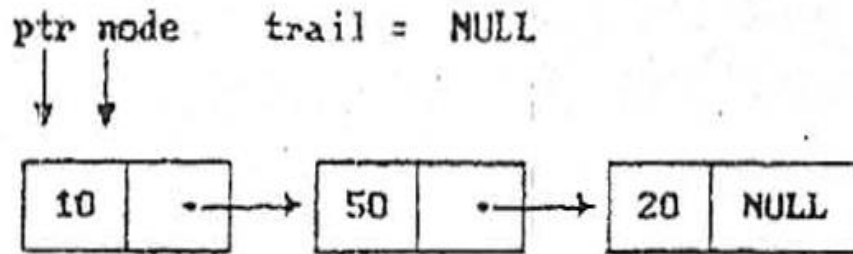


Two node list after the function call ***insert(&ptr, ptr)***;

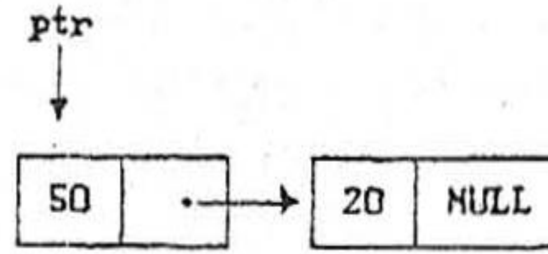
DELETION FROM A LIST

```
void delete(list_pointer *ptr, list_pointer trail, list_pointer node)
{
   //delete node from the list
   //trail is the preceeding node
   //ptr is the head of the list
   if(trail)
              trail->link = node->link;
  else
              *ptr = (*ptr)->link;

   free(node);
}
```

AFTER DELETION

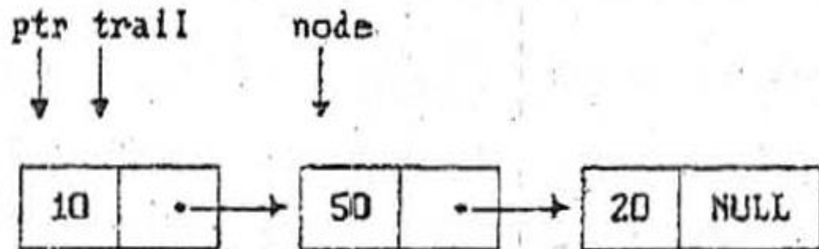


(a) before deletion

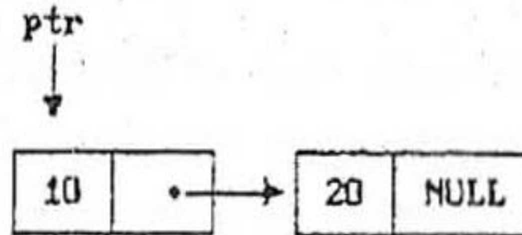


(b) after deletion

List after the function call ***delete(&ptr, NULL, ptr)***;

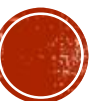


(a) before deletion



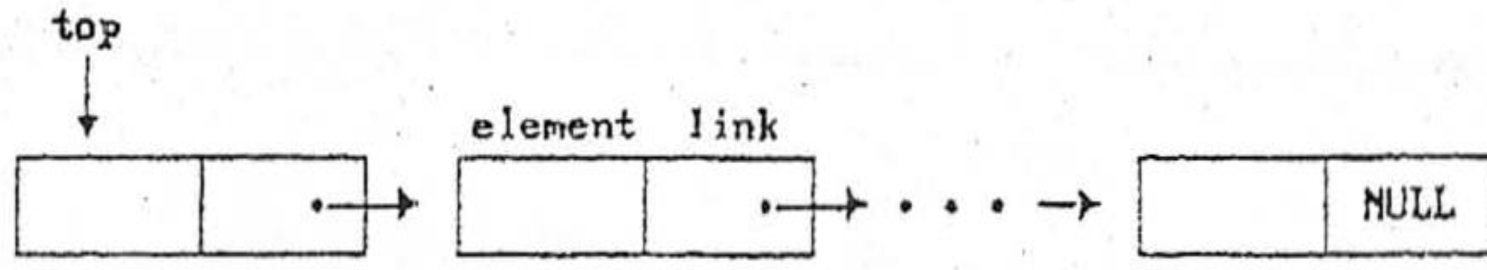
(b) after deletion

List after the function call ***delete(&ptr, ptr, ptr->link)***;

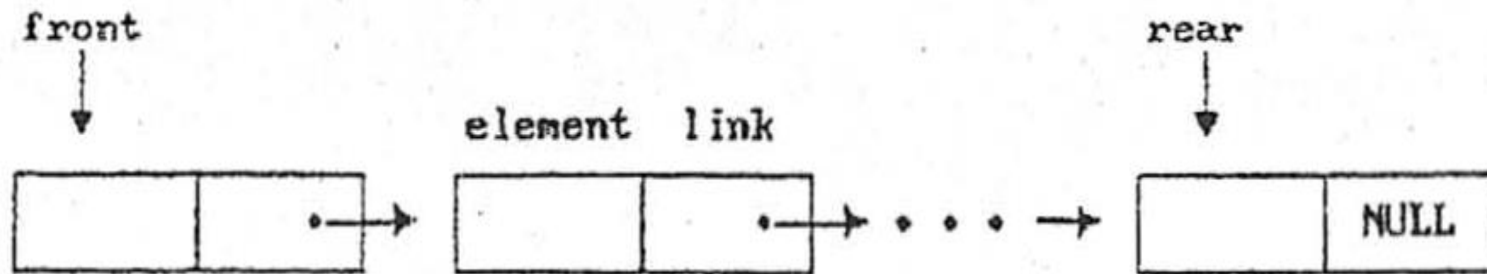


DYNAMICALLY LINKED STACKS AND QUEUES

LINKED STACK AND QUEUE



(a) Linked Stack



(b) linked queue

MULTIPLE STACKS

```
#define MAX_STACKS 10
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];

top[i] = NULL, 0 <= i < MAX_STACKS //the inital condition
```

INSERTION

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

A typical function call would be ***add(&top[stack_no], item);***

DELETION

```
element delete(stack_pointer *top)
{
   //delete an element from the stack
   stack_pointer temp = *top;
   element item;
   if(isEmpty(temp))
               exit(1);   //the stack is empty
   item = temp->item;
   *top = temp->link;
   free(temp);
   return item;
}
```

A typical function call would be item = ***delete(&top[stack_no]);***

MULTIPLE QUEUES

```
#define MAX_QUEUES 10
typedef struct{
        int key;
        /*other fields*/
}element;
typedef struct queue *queue_pointer;
typedef struct queue{
    element item;
    queue_pointer link;
};
queue_pointer front[MAX_QUEUES], rear[MAX_QUEUES];

front[i] = NULL, 0 <= i < MAX_QUEUES       //the inital condition
```

INSERTION

```
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(size_of(queue));
   if(IS_FULL(temp))
               exit(1);   //the memory is full
   temp->item = item;
   temp->link = NULL;
   if(*front)          (*rear)->link = temp;
   else                *front = temp;
   *rear = temp;
}
```

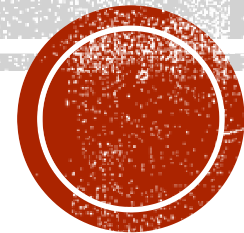
A typical function call would be ***addq(&front, &rear, item);***

DELETION

```
element deleteq(queue_pointer *front)
{
  //delete an element from the queue
  queue_pointer temp = *front;
  element item;
  if(isEmpty(*front))
          exit(1);   //the queue is empty
  item = temp->item;
  *front = temp->link;
  free(temp);
  return item;
}
```

A typical function call would be item = ***delete(&front);***

POLYNOMIALS AS SINGLY LINKED LISTS



REPRESENTATION

$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0x^{e_0}$$

Where the a_i are non-zero coefficients and the e_i are nonnegative integer exponents such that $e_{m-1} > e_{m-2} > \ldots > e_1 > e_0 >= 0$

LINKED LIST REPRESENTATION

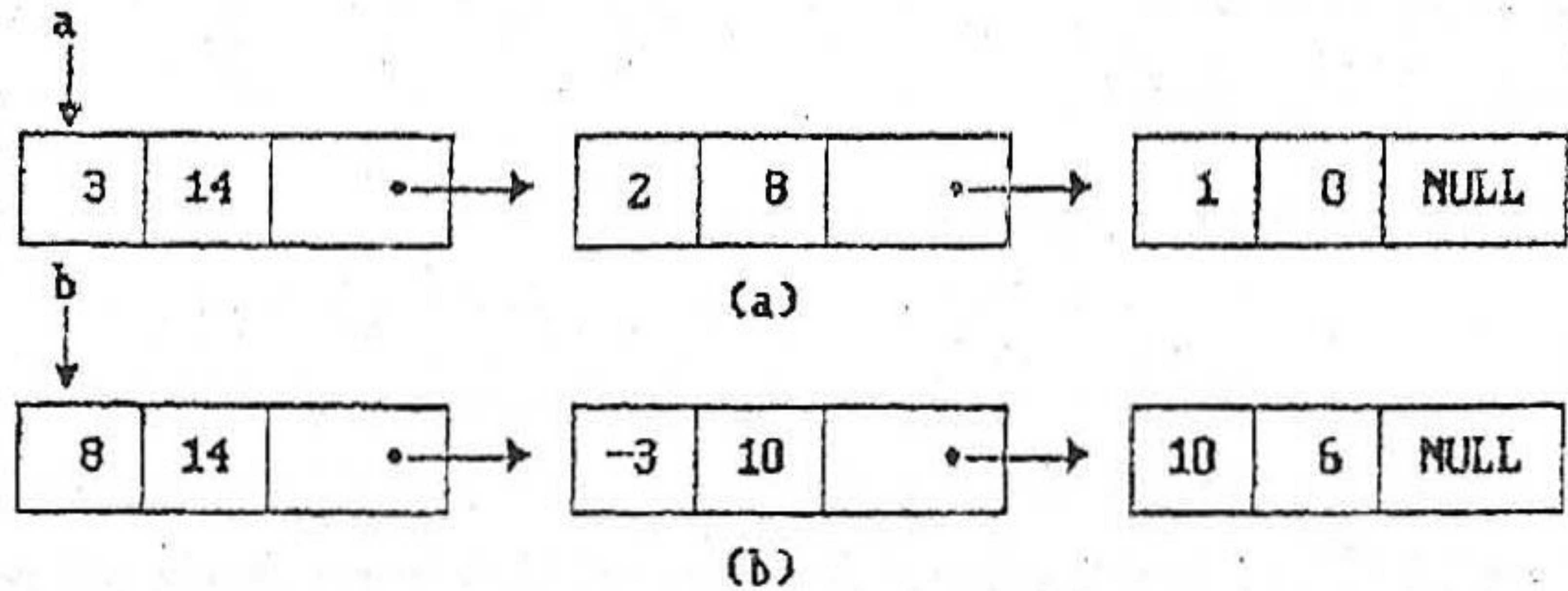
```
typedef struct poly_node *poly_pointer;

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

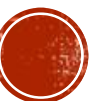
coef	expon	link

EXAMPLE

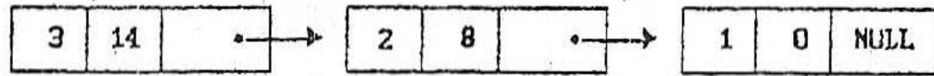
$a = 3x^{14} + 2x^{8} + 1$

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

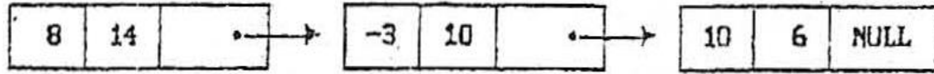




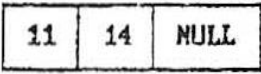
ADDING POLYNOMIALS



a

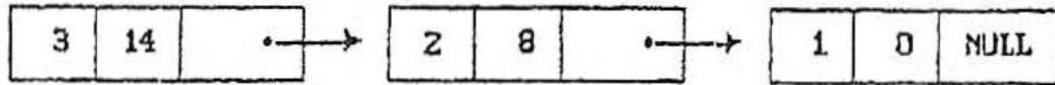


b

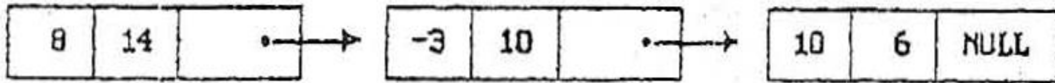


d

(a) a -> expon == b -> expon



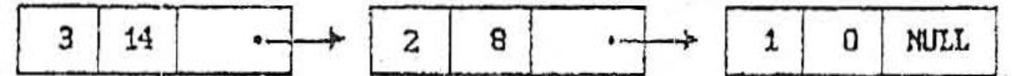
a



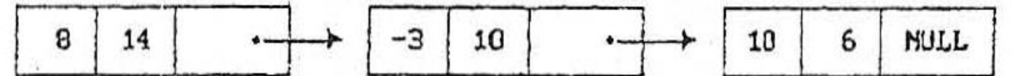
b

d

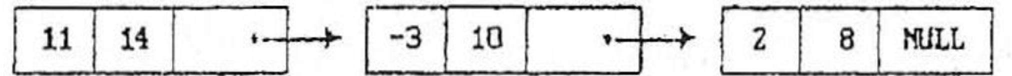
(b) a -> expon < b -> expon



a



b

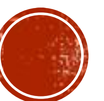


d

(c) a -> expon > b -> expon

```
poly_pointer padd(poly_pointer a, poly_pointer b)
{
        poly_pointer front, rear, temp;
        int sum;
        rear = (poly_pointer)malloc(sizeof(poly_node));
        if(isFull(rear))
                exit(1);   //the memory is full
        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;
                                  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;
        //delete extra initial node
        temp = front; front = front ->link; free(temp);
        return front;
}
```

a has m terms
b has n terms
O(m+n)

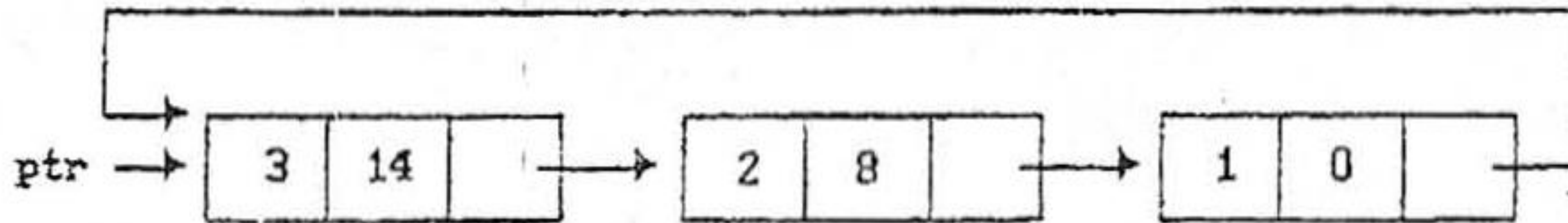


```
void attach(float coefficient, int exponent, poly_pointer *ptr)
{
        poly_pointer temp;
        temp = (poly_pointer)malloc(sizeof(poly_node));
        if(IS_FULL(temp))
                exit(1);   //the memory is full
        temp->coef = coefficient;
        temp->expon = exponent;
        (*ptr)->link = temp;
        *ptr = temp;
}
```

ERASING A POLYNOMIAL

```
void erase(poly_pointer *ptr)
{
        poly_pointer temp;

        while(*ptr){
                temp = *ptr;
                *ptr = (*ptr)->link;
                free(temp);
        }
}
```

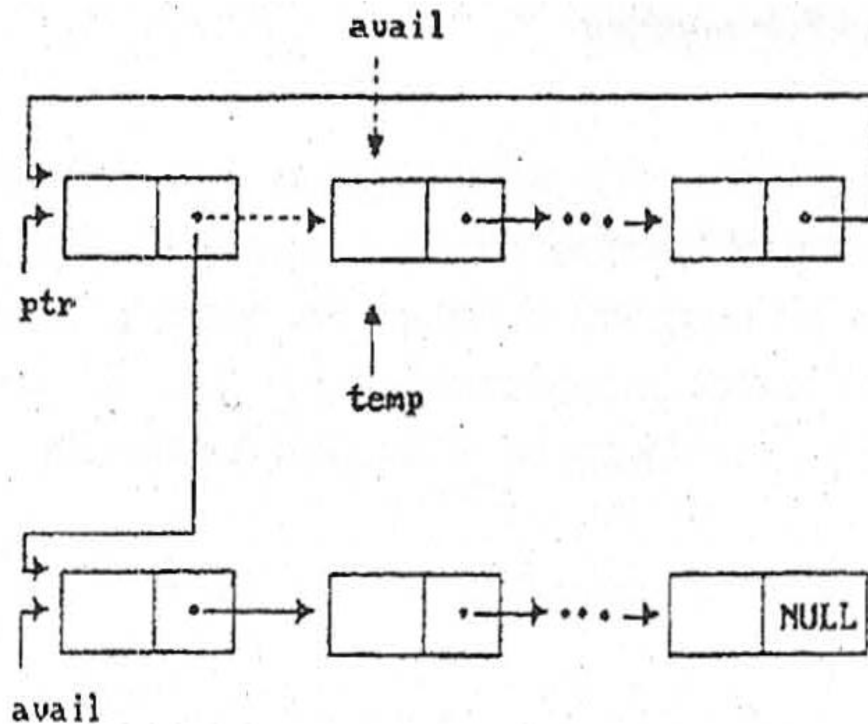
CIRCULAR LINKED LIST REPRESENTATION



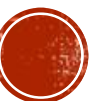
One more list ***avail*** which keeps the available nodes for use.

```
//instead of using malloc
poly_pointer get_node(void)
{
        poly_pointer node;
        if(avail){
                node = avail;
                avail = avail->link;
        }
        else{
                node = (poly-pointer) malloc(sizeof(poly_node));
                if(IS_FULL(node))
                        exit(1); //the memory is full
        }
        return node;
}

//instead of using free
void ret_node(poly_pointer ptr)
{
        ptr->link = avail;
        avail = ptr;
}
```

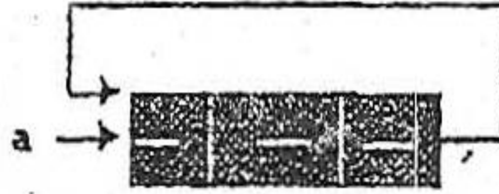
```
void cearse(poly_pointer *ptr)
{
        poly_pointer temp;
        if(*ptr){
                temp = (*ptr)->link;
                (*ptr)->link = avail;
                avail = temp;
                *ptr = NULL;
        }

}
```



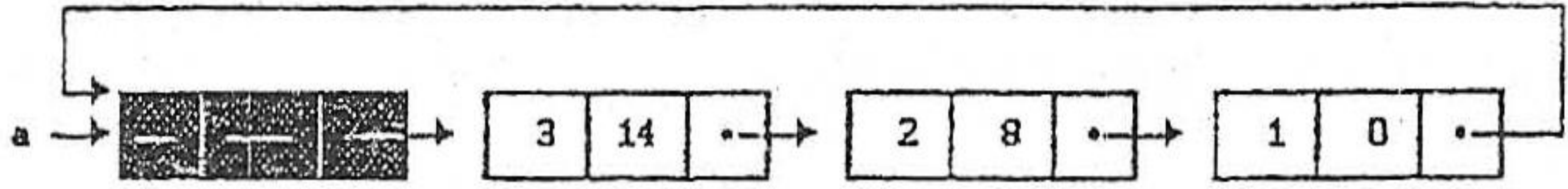
- independent of the number of nodes in the list.
- a fixed amount of time.



POLYNOMIAL REPRESENTATIONS

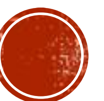


(a) Zero polynomial



(b) $3x^{14}+2x^{8}+1$

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



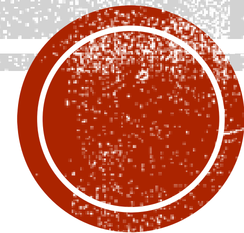
OTHER OPERATIONS - INVERT

```
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;
}
```

OTHER OPERATIONS - CONCATENATE

```
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;
        }

}
```

CIRCULARLY LINKED LISTS

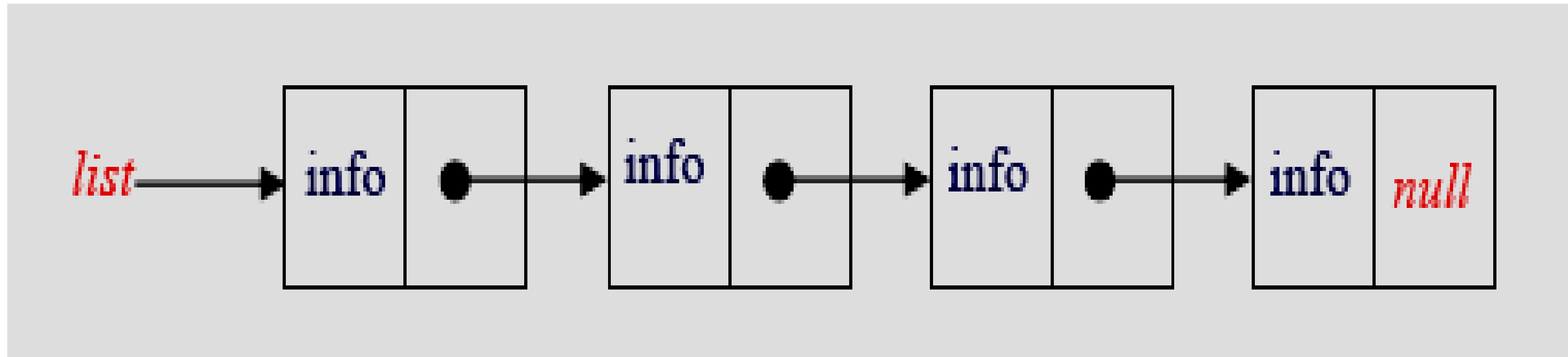


some slides are taken from http://faraday.ee.emu.edu.tr/eeng212/

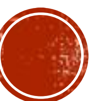
CIRCULAR LINKED LISTS

- In linear linked lists if a list is traversed (all the elements visited) an external pointer to the list must be preserved in order to be able to reference the list again.
- Circular linked lists can be used to help the traverse the same list again and again if needed. A circular list is very similar to the linear list where in the circular list the pointer of the last node points not NULL but the first node.

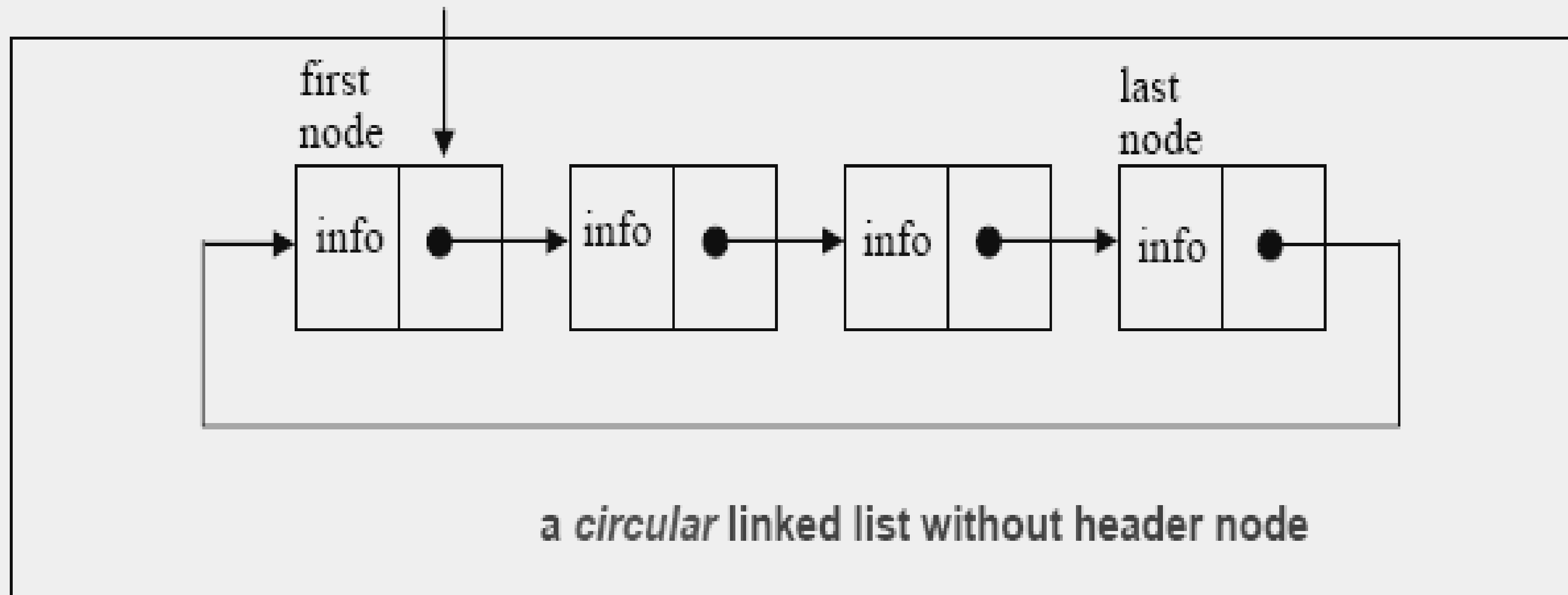
CIRCULAR LINKED LISTS

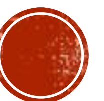


A Linear Linked List

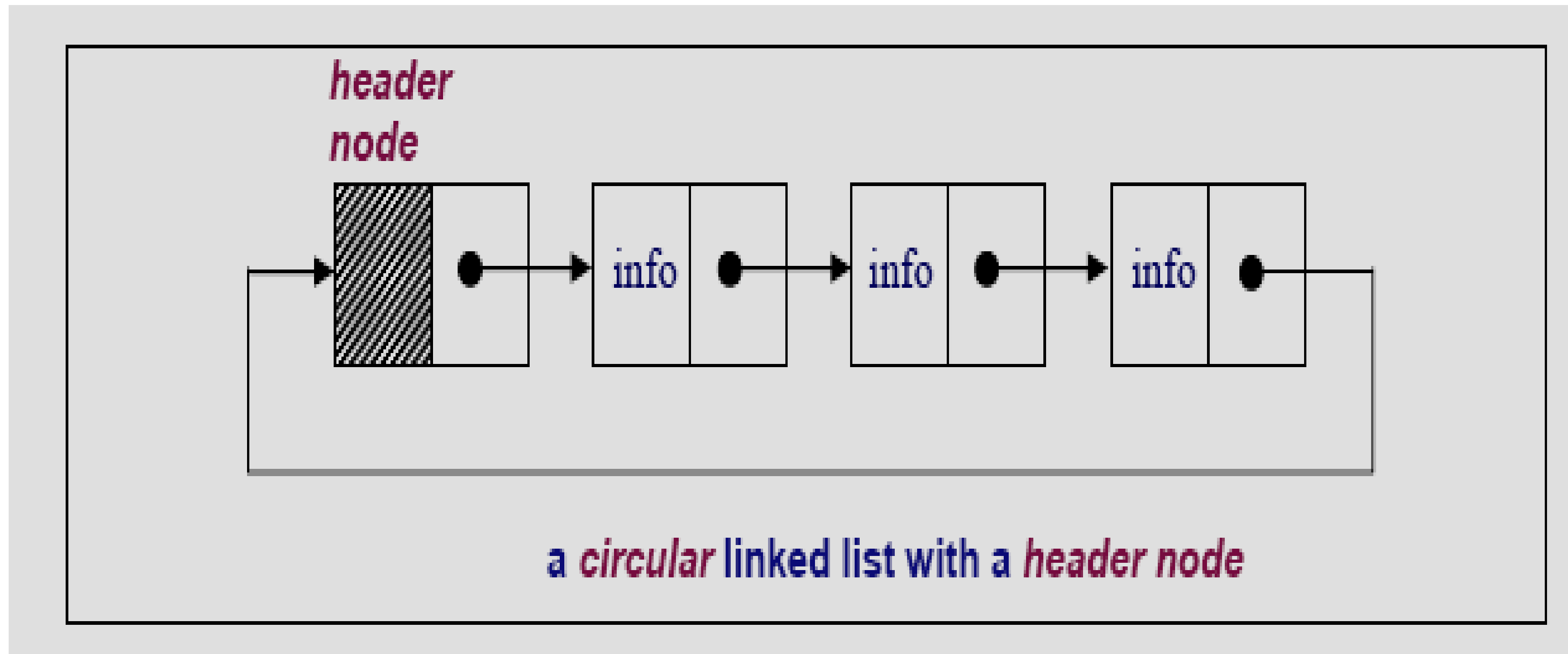


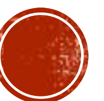
CIRCULAR LINKED LISTS





CIRCULAR LINKED LISTS





CIRCULAR LINKED LISTS

- In a circular linked list there are two methods to know if a node is the first node or not.
  - Either a external pointer, ***list***, points the first node or
  - A ***header node*** is placed as the first node of the circular list.
- The header node can be separated from the others by either heaving a ***sentinel value*** as the info part or having a dedicated ***flag*** variable to specify if the node is a header node or not.

PRIMITIVE FUNCTIONS IN CIRCULAR LISTS

- The structure definition of the circular linked lists and the linear linked list is the same:

```
struct node{
   int info;
   struct node *next;
};
typedef struct node *NODEPTR;
```

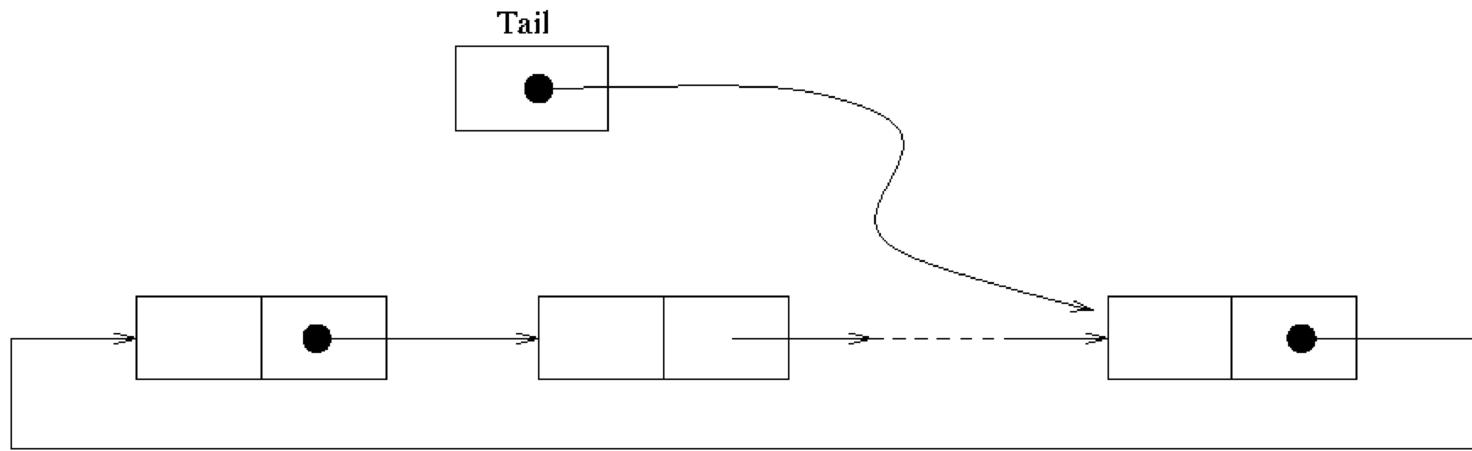
DELETE A NODE

The delete after and insert after functions of the linear lists and the circular lists are almost the same.

```
void delafter(NODEPTR p, int *px)
{
        NODEPTR q;
        if((p == NULL) || (p == p->next)){ /*the empty list
                        contains a single node and may be pointing itself*/
                printf("void deletion\n");
                exit(1);
        }
        q = p->next;
        *px = q->info; /*the data of the deleted node*/
        p->next = q->next;
        freenode(q);
}
```

INSERT A NODE

```
void insafter(NODEPTR p, int x)
{
        NODEPTR q;
        if(p == NULL){
                printf(“void insertion\n”);
                exit(1);
        }
        q = getnode();
        q->info = x; /*the data of the inserted node*/
        q->next = p->next;
        p->next = q;
}
```




```
typedef struct list_node *list_pointer;

typedef struct list_node{
        int item;
        list_pointer link;
      };
```

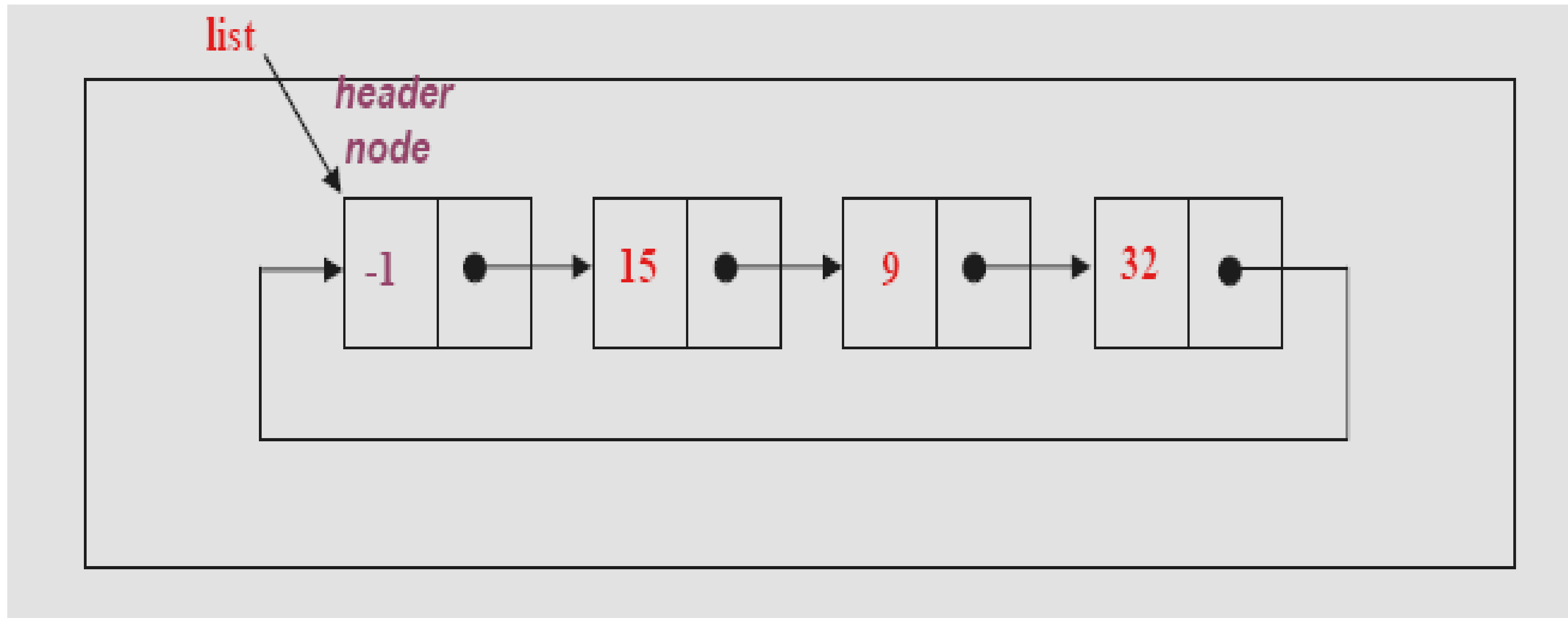
```
void insert_front(list_pointer *ptr, list_pointer node)
{ /*insert node at the front of the circular list ptr, where
ptr is the last node in the list*/
        if(IS_EMPTY(*ptr)){
                *ptr = node;
                node ->link = node;
        }
        else{
                node->link = (*ptr)->link;
                (*ptr)->link = node;
        }
}
```

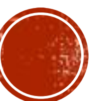
```
int length(list_pointer ptr)
{
        /*find the length of the circular list ptr*/
        list_pointer temp;
        int count = 0;
        if(ptr) {
                temp = ptr;
                do {
                        count++;
                        temp = temp->link;
                } while (temp != ptr);
        }
        return count;
}
```

CIRC. LIST WITH HEADER NODE

- The header node in a circular list can be specified by a ***sentinel value*** or a dedicated ***flag***:
- ***Header Node with Sentinel:*** Assume that info part contains integers. Therefore the info part of a header node can be -1. The following circular list is an example for a sentinel used to represent the header node:

CIRCULAR LIST WITH HEADER NODE





CIRCULAR LIST WITH HEADER NODE

- *Header Node with Flag:* In this case a extra variable called flag can be used to represent the header node. For example flag in the header header node can be 1, where the flag is 0 for the other nodes.

```
struct node{
        int flag;
        int info;
struct node *next;
};
typedef struct node *NODEPTR;
```

CIRCULAR LIST WITH HEADER NODE

list

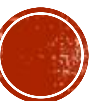
header node

flag=1 Info=27

flag=0 Info=15

flag=0 Info=3

flag=0 Info=7



EXAMPLE

- Consider a circular linked list with a header node, where each node contains the name, account number and the balance of a bank customer. The header node contains a sentinel account number to be -99.
- **(a)** Write an appropriate node structure definition for the circular linked
- **(b)** Write a function to display the full records of the customers with balance.

```c
struct node{
	char Name[15];
	int AccNo;
	float Balance;
	struct node *next;
};
typedef struct node
*NODEPTR;
```

```c
//Assume that the list pointer points the header with the sentinel account number -99.


void DispNegBalanca(NODEPTR *plist)
{
		NODEPTR p;
		p=*plist;
		if(p == NULL){
				printf("There is no list!\n");
				exit(1);
		}
		p=p->next;
		while(p->AccNo!=-99)
		{
				if(p->Balance > 0.0)
				printf("The Customer Name:%s\nThe Account No:%d\nThe
						Balance:%.2f\n", p->Name, p->AccNo, p->Balance);
				p=p->next;
		}
}
```

EXAMPLE

- Write a function that returns the average of the numbers in a circular list. Assume that the following node structure is used, where the flag variable is 1 for the header node and 0 for all the other nodes.

```
struct node{
    int flag;
    float info;
    struct node *next;
};
typedef struct node *NODEPTR;
```

```
float avList(NODEPTR *plist)/*assume that plist points the header node*/
{
        int count=0;
        float sum =0.0;
        NODEPTR p;
        p=*plist;
        if((p == NULL)){
                printf("Empty list\n");
                exit(1);
        }
        do{
                sum=sum + p->info;
                p =p->next;
                count++;
        }while(p->flag !=1);
        return sum/count;
}
```