

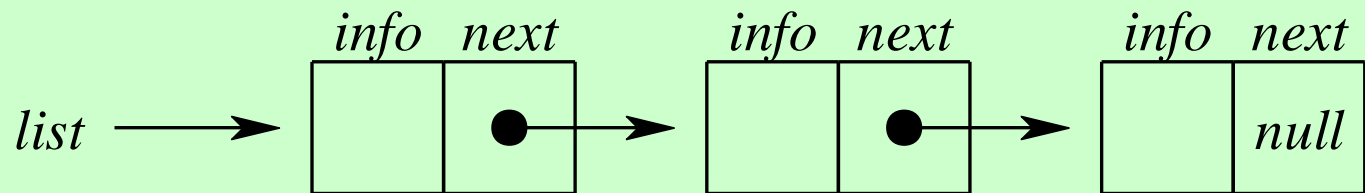
Linked List

Content

- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Applications

Singly Linked List

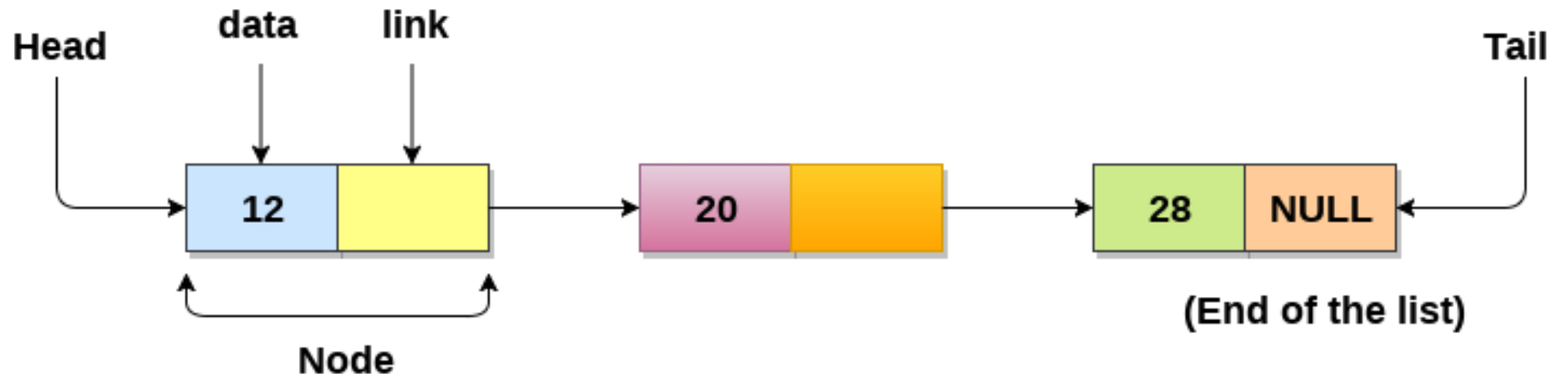
- A ***linked list*** is a linear collection of data elements, called ***nodes***, where the linear order is given by means of ***pointers***.
- Each ***node*** is divided into two parts:
 - The first part contains the ***information*** of the element and
 - The second part contains the address of the next node (***link /next pointer field***) in the list.



Linear linked list

Contd.

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.



Properties

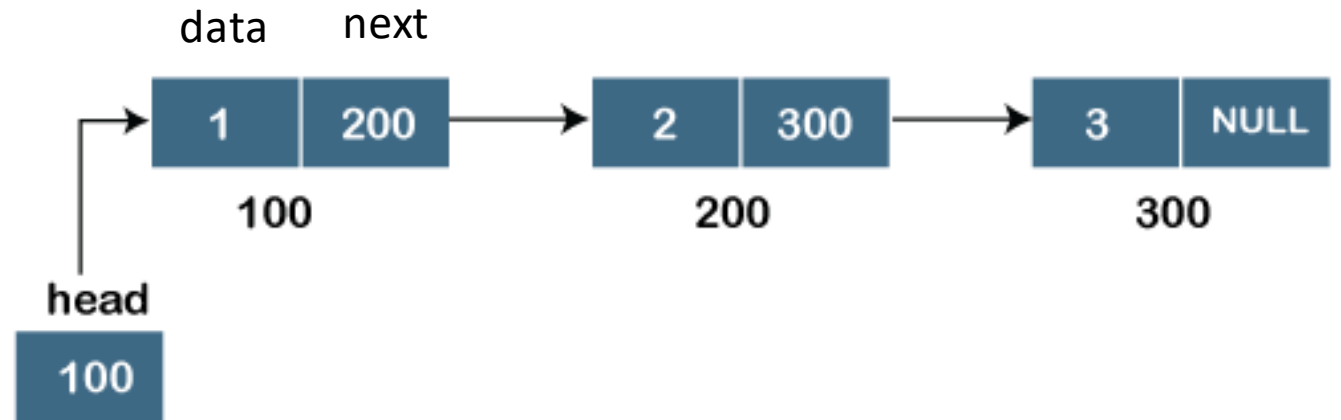
- The list is not required to be contiguously present in the memory. The node can reside anywhere in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can not be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Disadvantage of Arrays

- 1.The size of array must be known in advance before using it in the program.
- 2.Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
- 3.All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

Representation

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



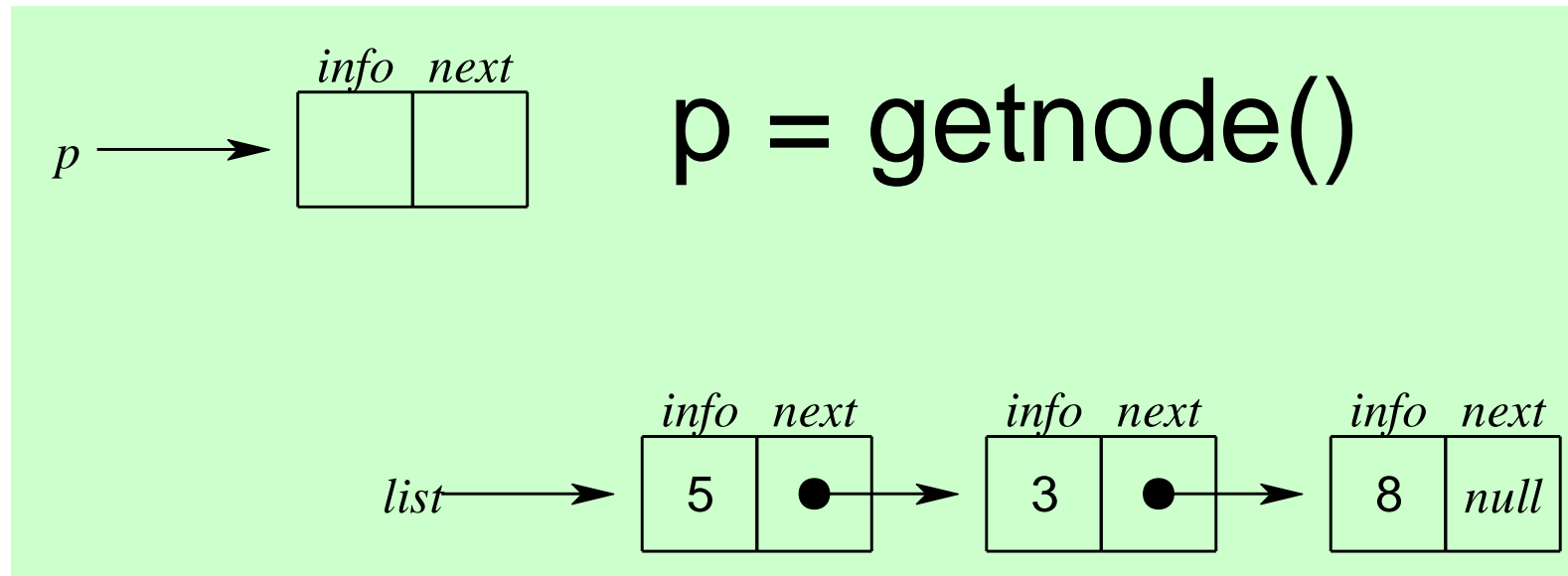
Operations

- Insertion
 - at beginning
 - at end
 - after specified node
- Deletion
 - at beginning
 - at end
 - after specified node
- Traversing
- Searching

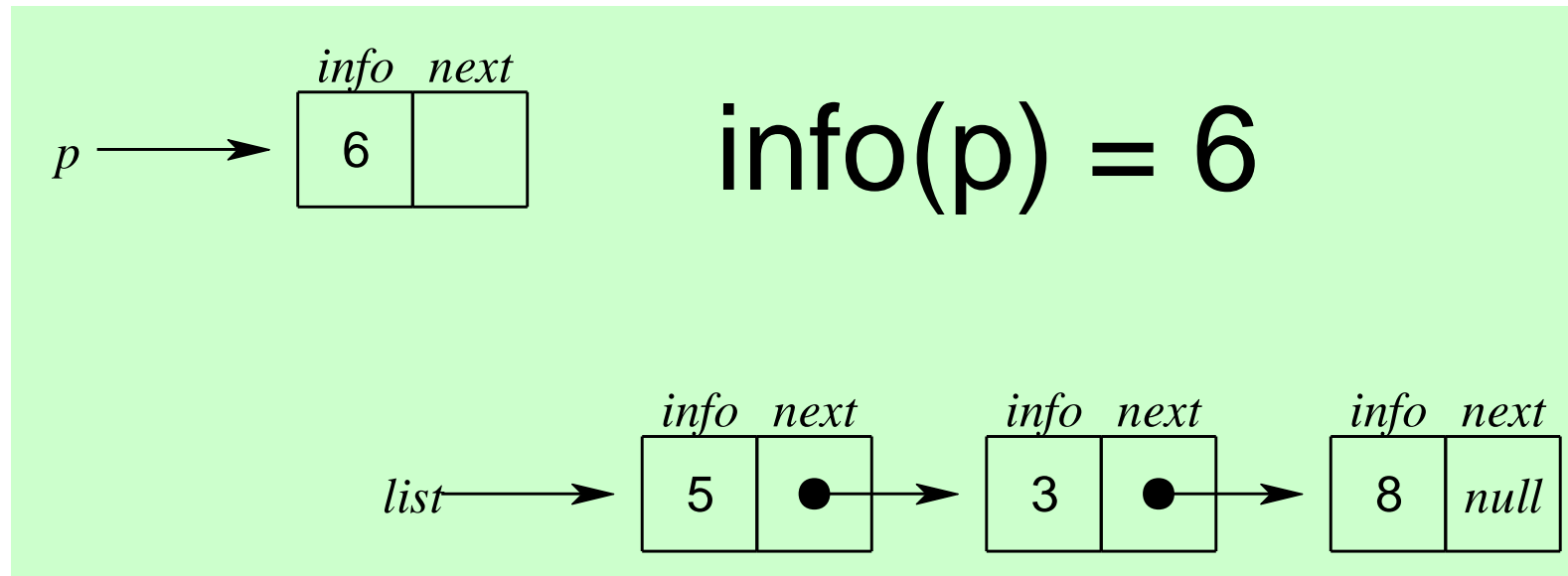
Basic Code

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *head;
```

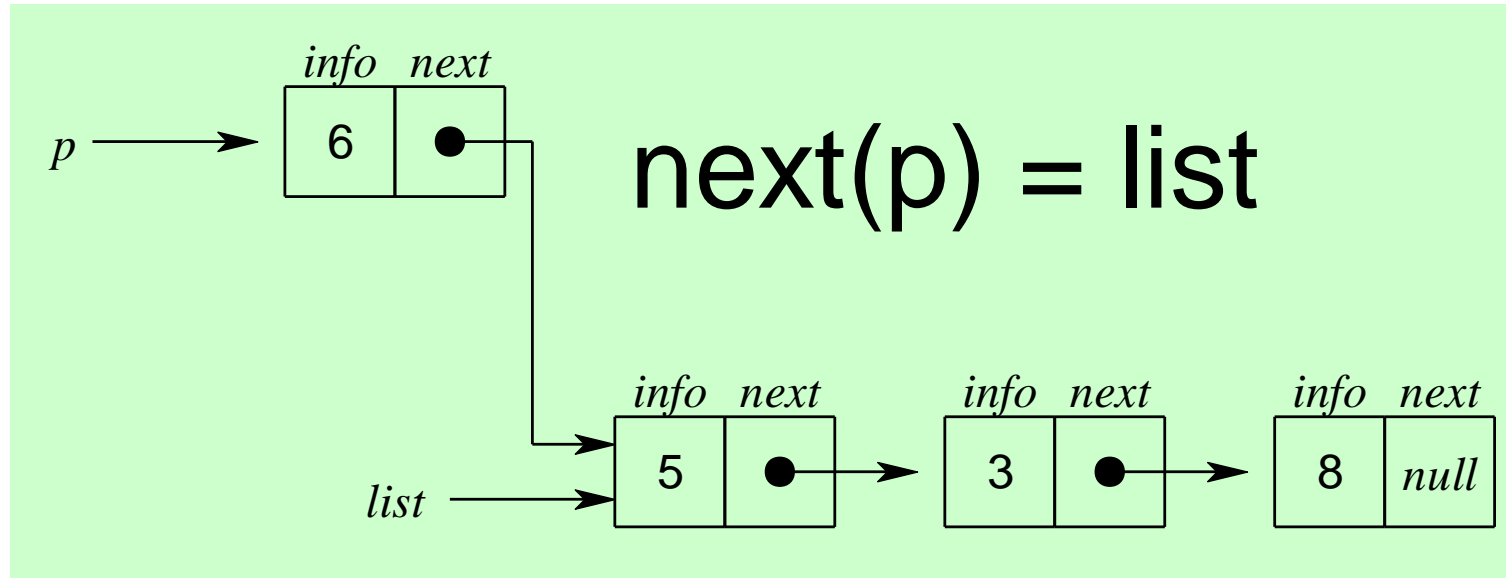
Insert at the beginning



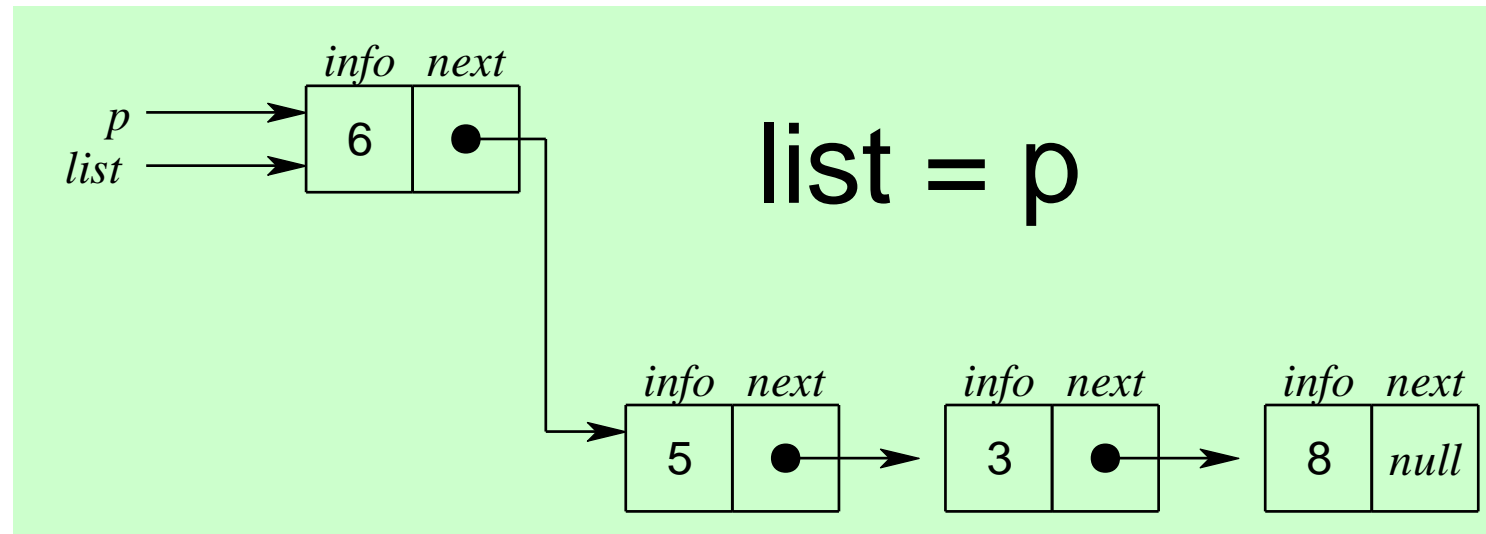
Contd.



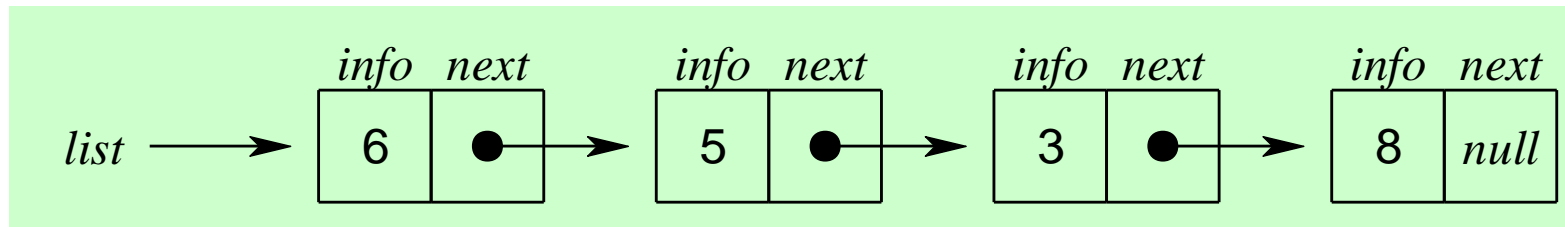
Contd.



Contd.



Contd.

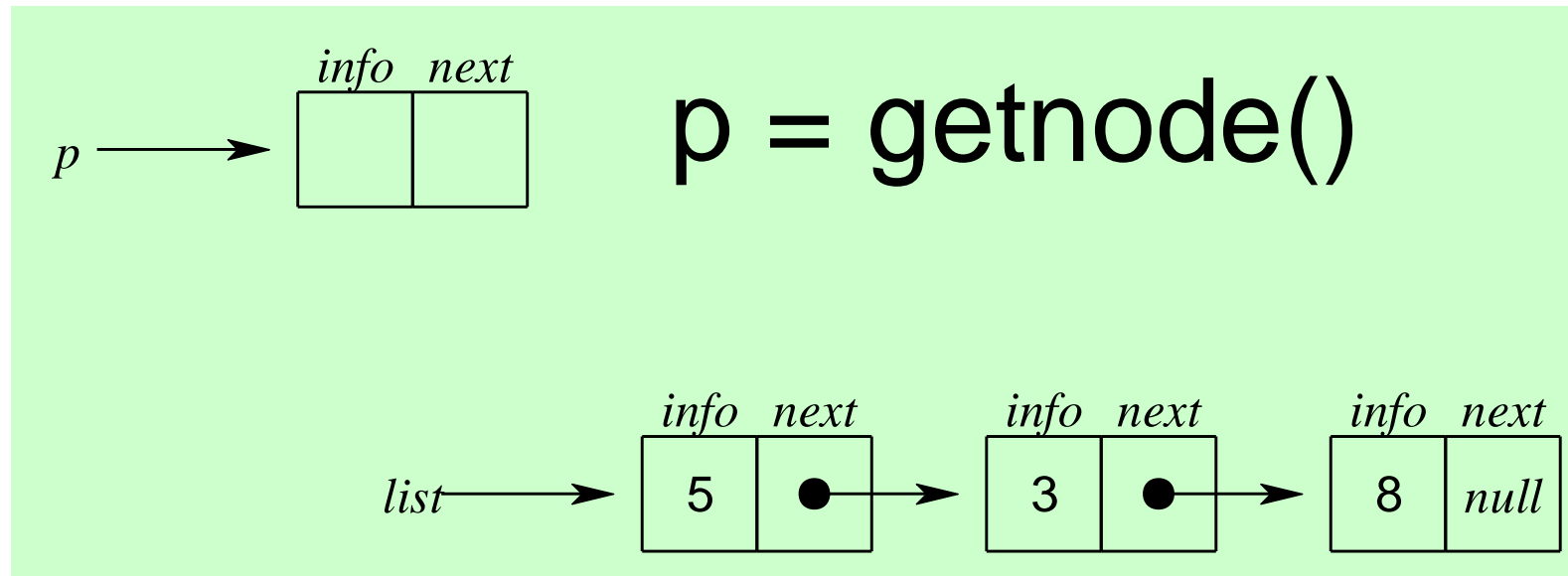


Code

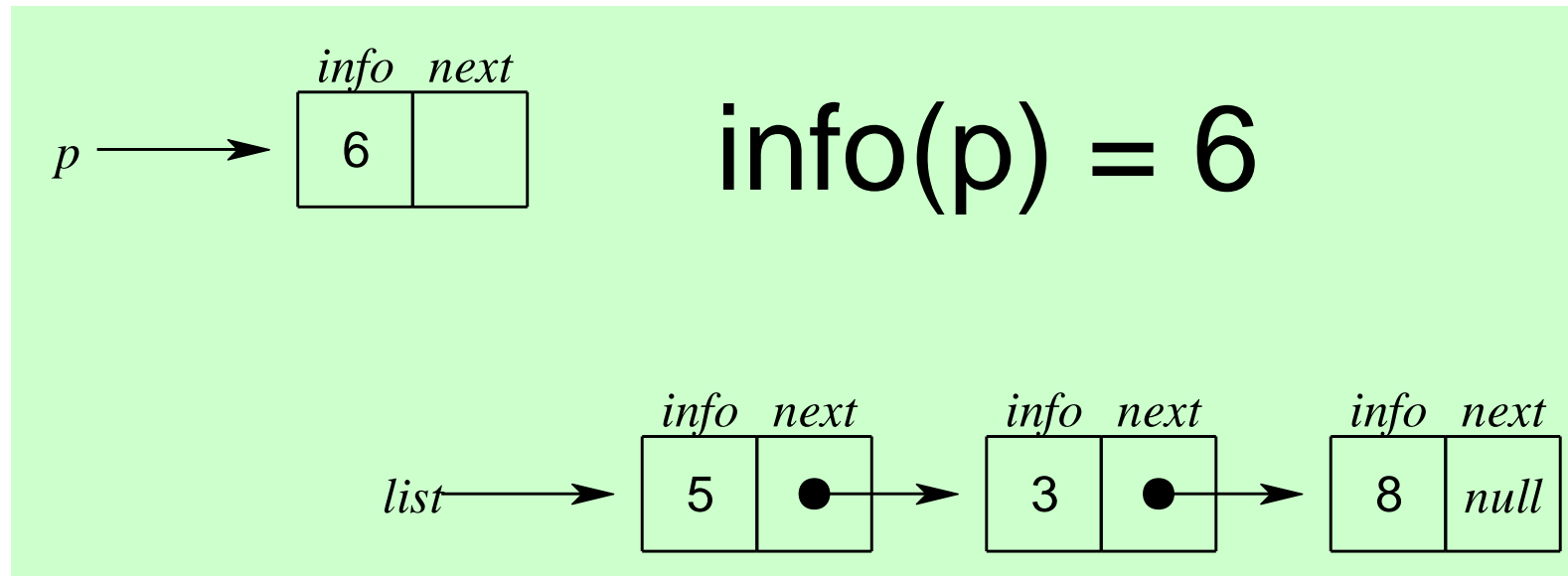
```
void begininsert()
{
    struct node *ptr;
    int item;
    ptr = (struct node *)
    malloc(sizeof(struct node *));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
}
```

```
else
{
    printf("\nEnter value\n");
    scanf("%d",&item);
    ptr->data = item;
    ptr->next = head;
    head = ptr;
    printf("\nNode inserted");
}
}
```

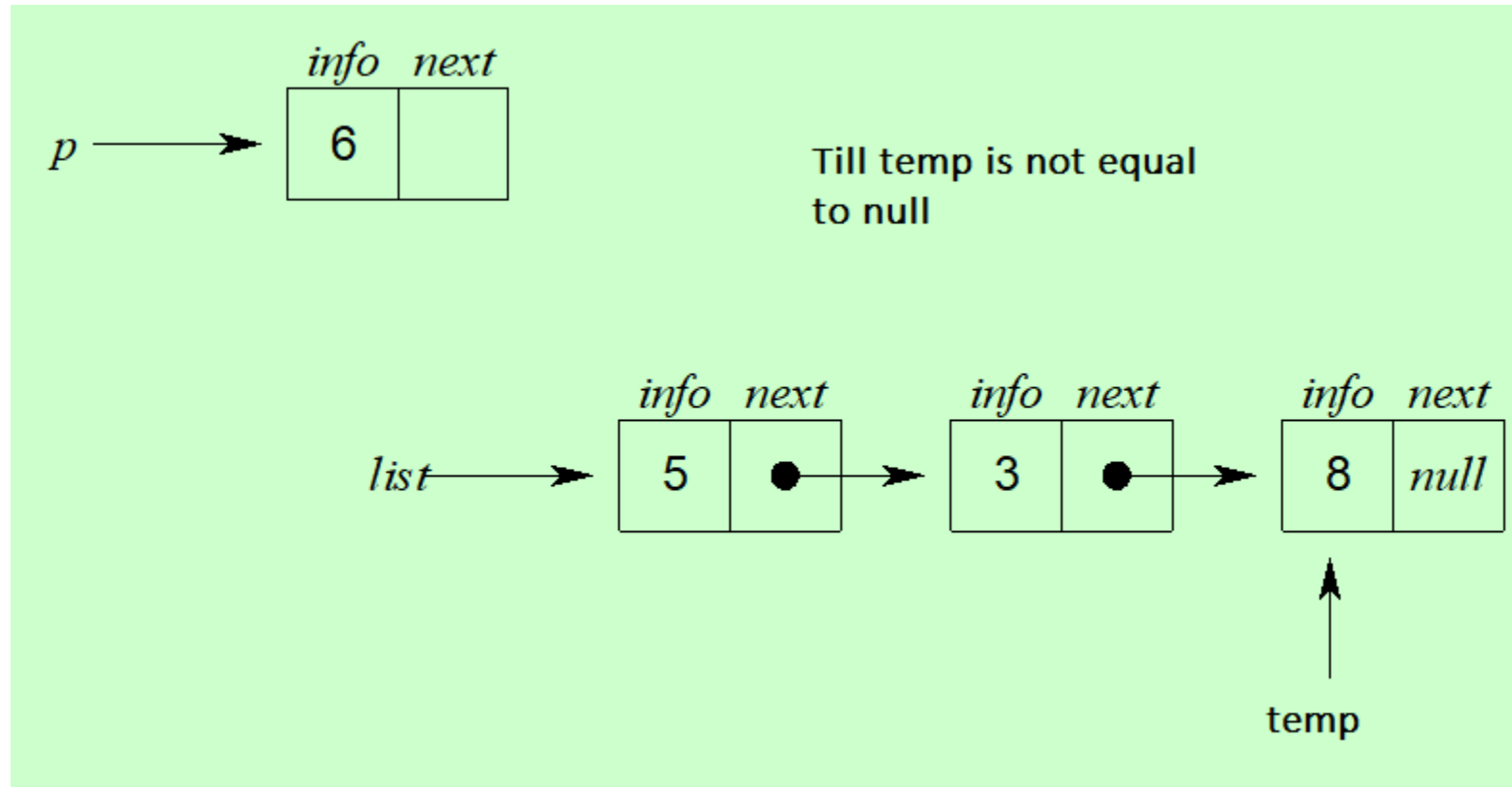
Insert at the end



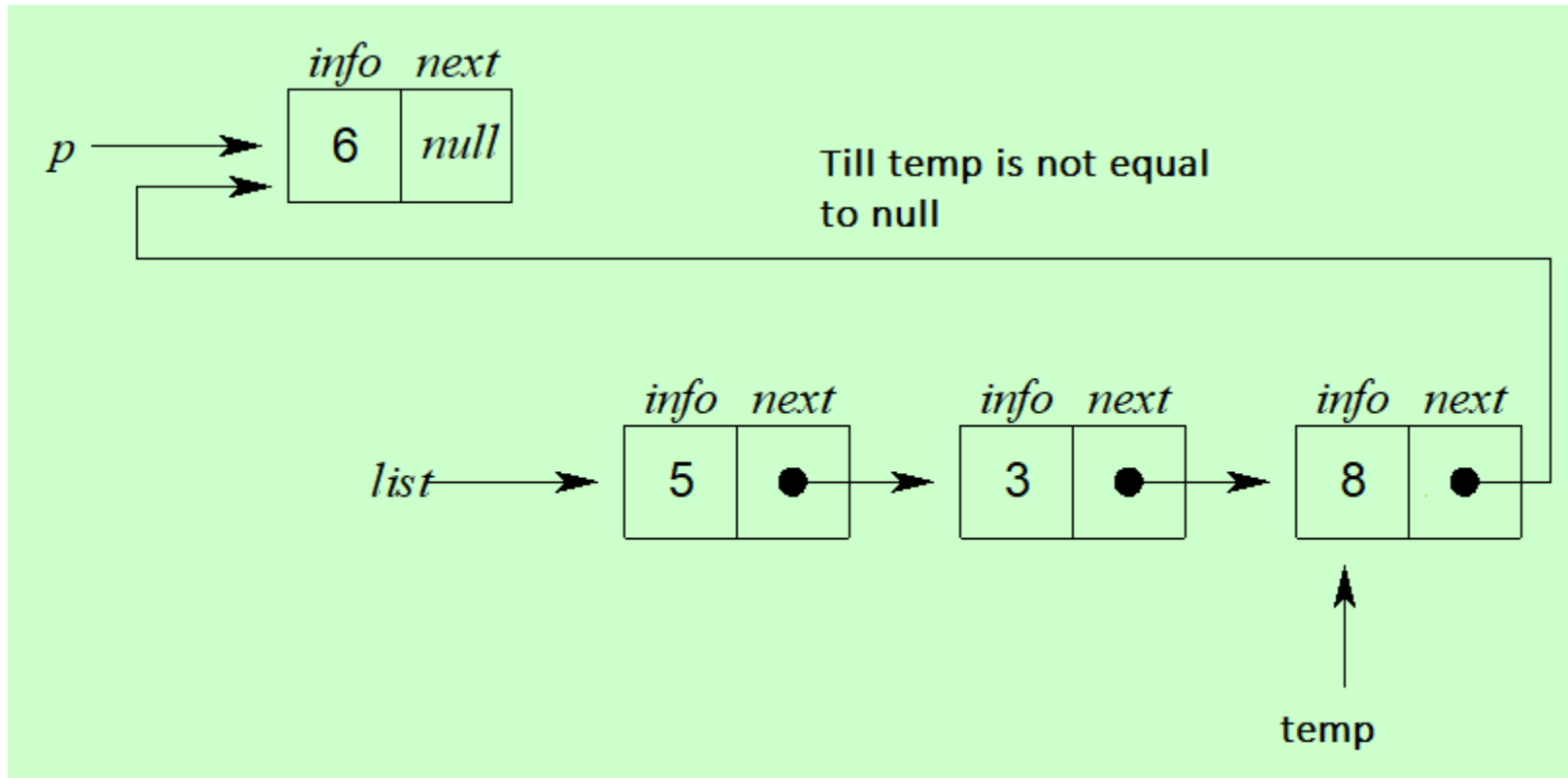
Contd.



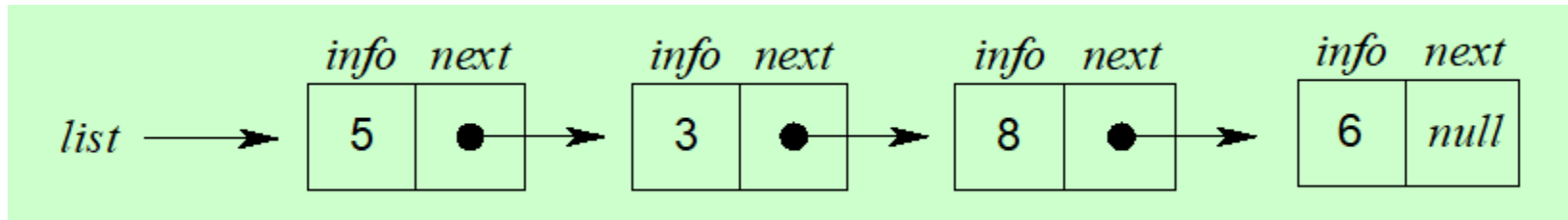
Contd.



Contd.



Contd.



Code

```
void lastinsert()  
{  
    struct node *ptr,*temp;  
    int item;  
    ptr = (struct node*)malloc(sizeof(struct node));  
    if(ptr == NULL)  
    {  
        printf("\nOVERFLOW");  
    }
```

Code

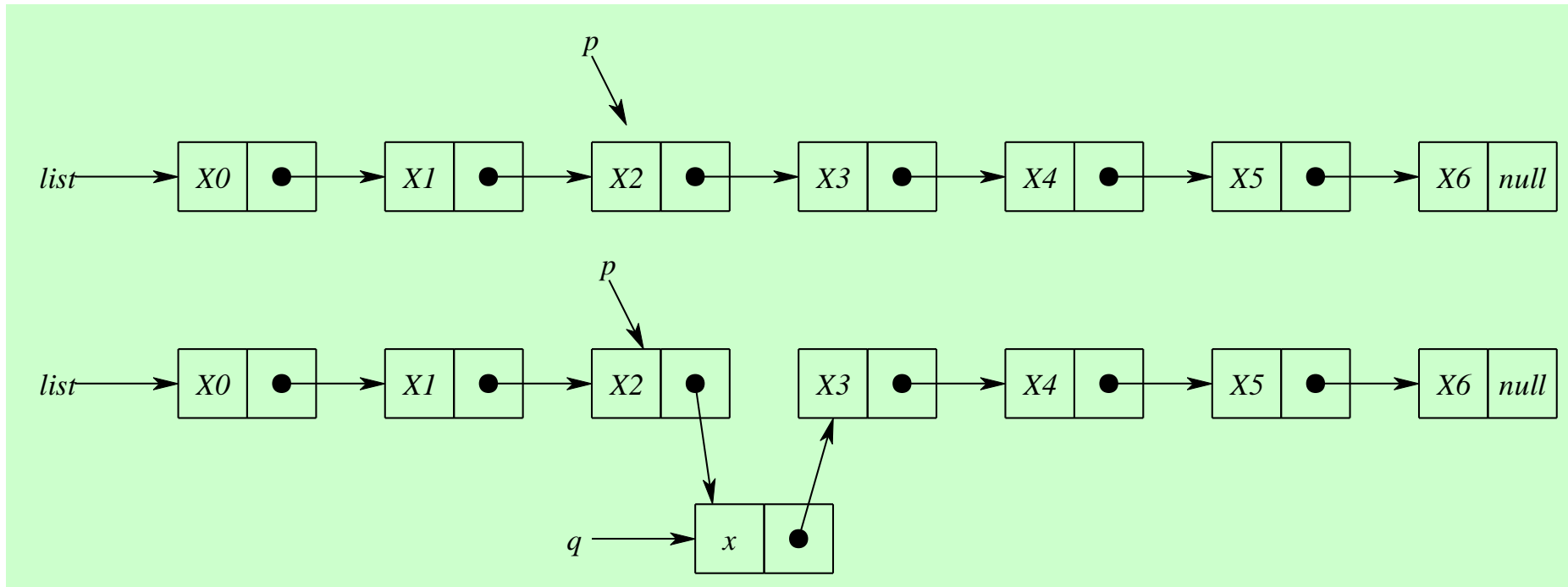
else

```
{
    printf("\nEnter value?\n");
    scanf("%d",&item);
    ptr->data = item;
    if(head == NULL)
    {
        ptr -> next = NULL;
        head = ptr;
        printf("\nNode inserted");
    }
}
```

else

```
{
    temp = head;
    while (temp -> next != NULL)
    {
        temp = temp -> next;
    }
    temp->next = ptr;
    ptr->next = NULL;
    printf("\nNode inserted");
}
}
```

Insert after location



Code

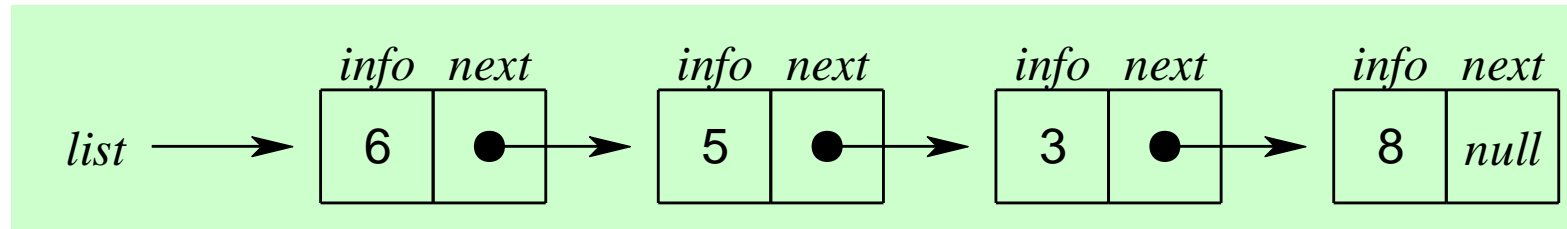
```
void randominsert()
{
    int i,loc,item;
    struct node *ptr, *temp;
    ptr = (struct node *) malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter element value");
        scanf("%d",&item);
```

```
        ptr->data = item;
        printf("\nEnter the location after which you want to insert ");
        scanf("\n%d",&loc);
        temp=head;
        for(i=0;i<loc;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
                printf("\ncan't insert\n");
                return;
            }
        }
```


Contd.

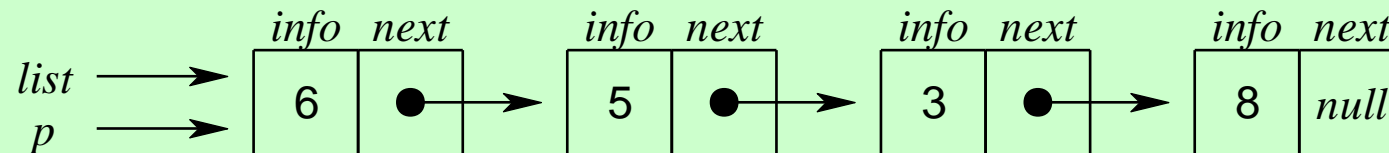
```
}  
    ptr ->next = temp ->next;  
    temp ->next = ptr;  
    printf("\nNode inserted");  
}  
}
```

Deletion from beginning

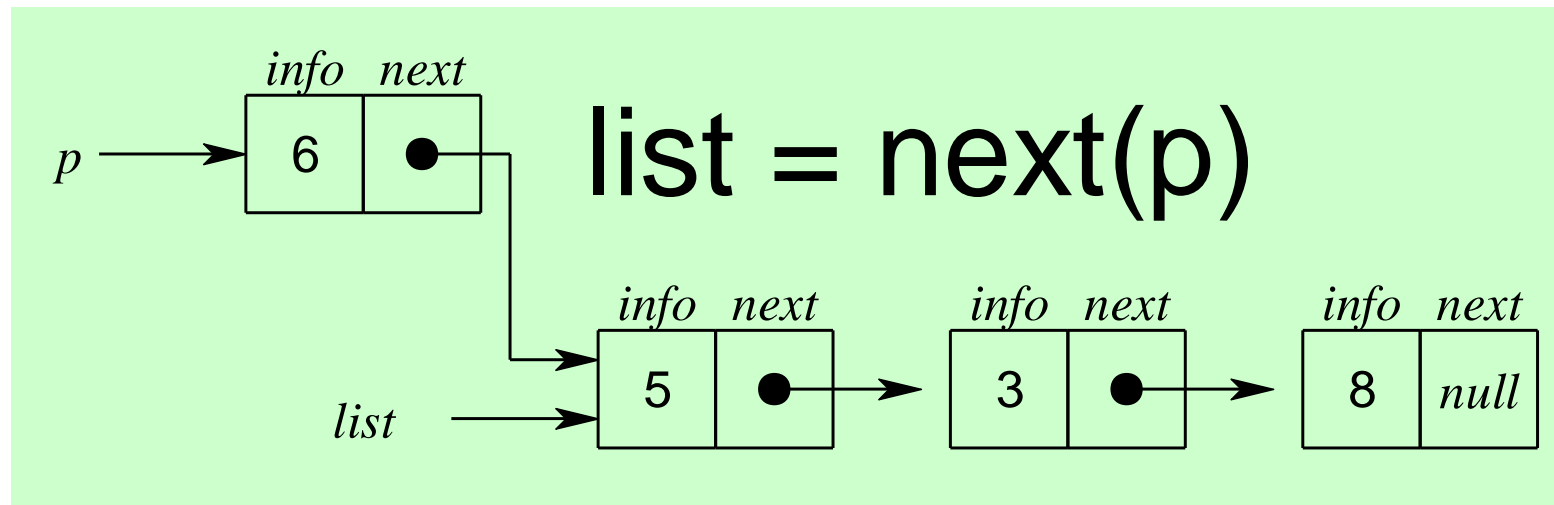


Contd.

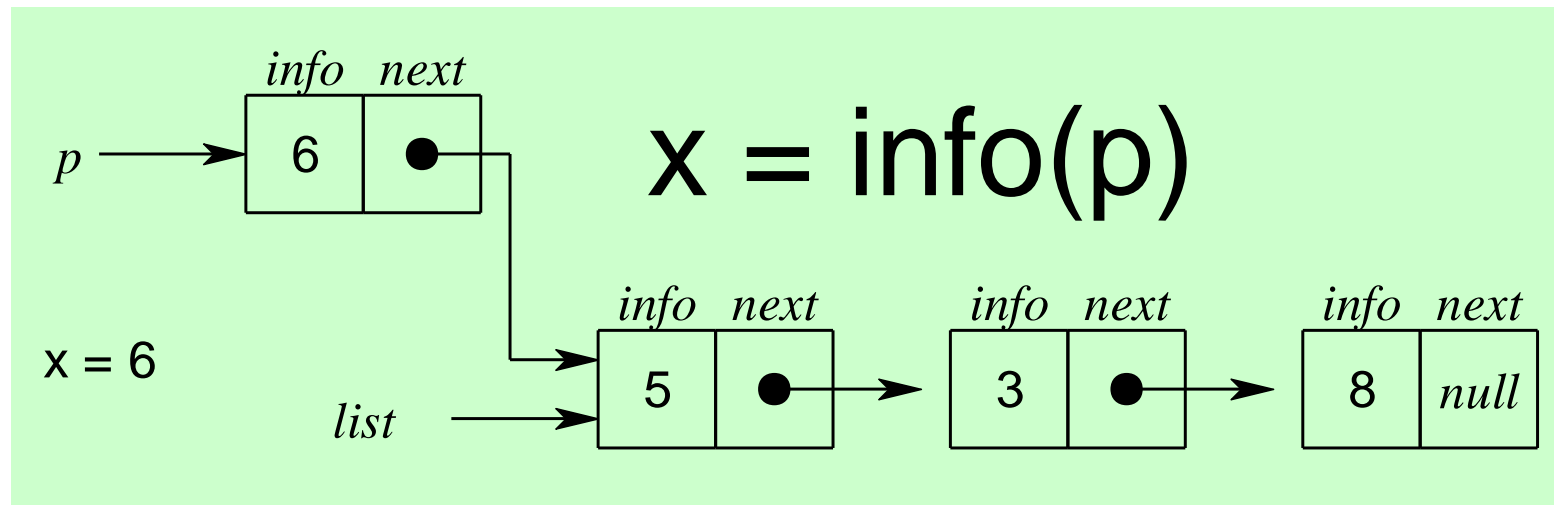
$p = \text{list}$



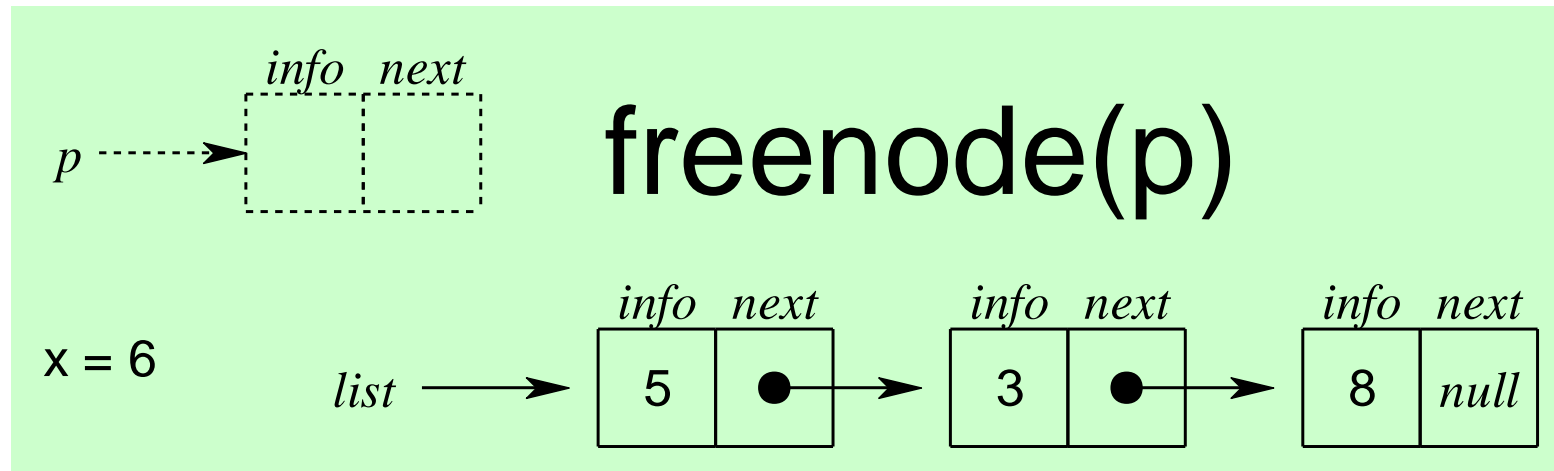
Contd.



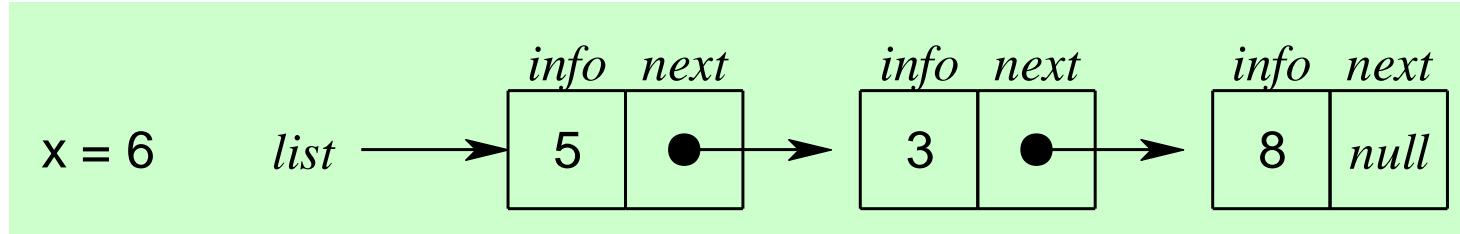
Contd.



Contd.



Contd.

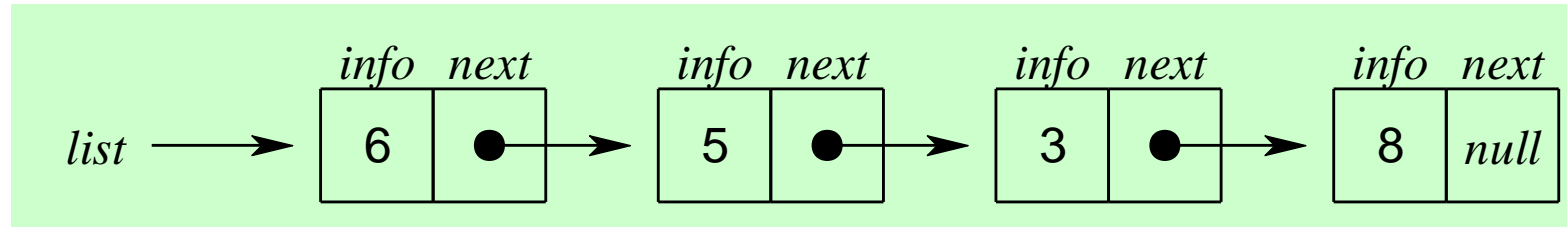


Code

```
void begin_delete()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\nList is empty\n");
    }
    else
    {
```

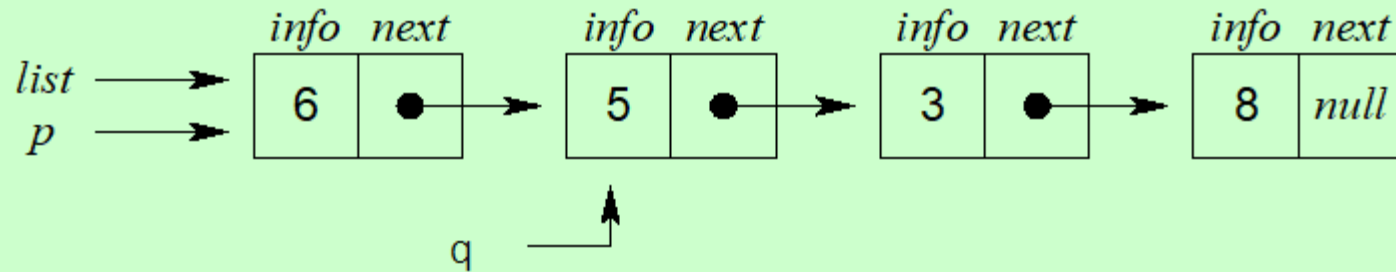
```
        ptr = head;
        head = ptr->next;
        free(ptr);
        printf("\nNode deleted from
the begining ...\n");
    }
}
```


Deletion from End

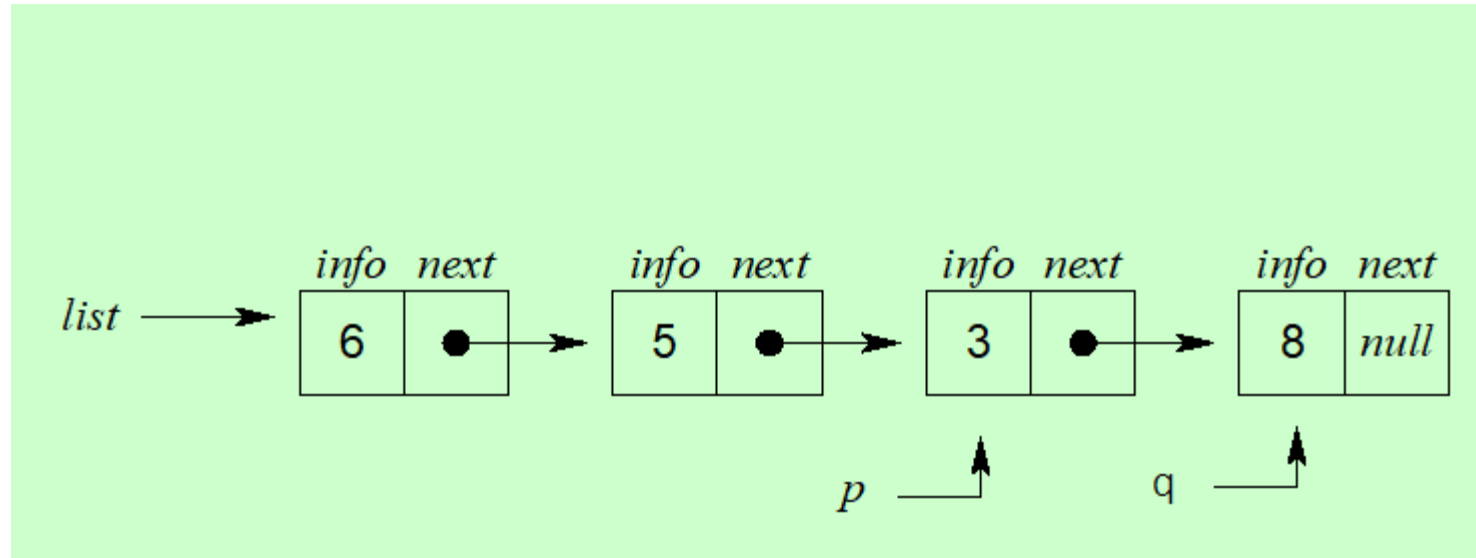


Contd.

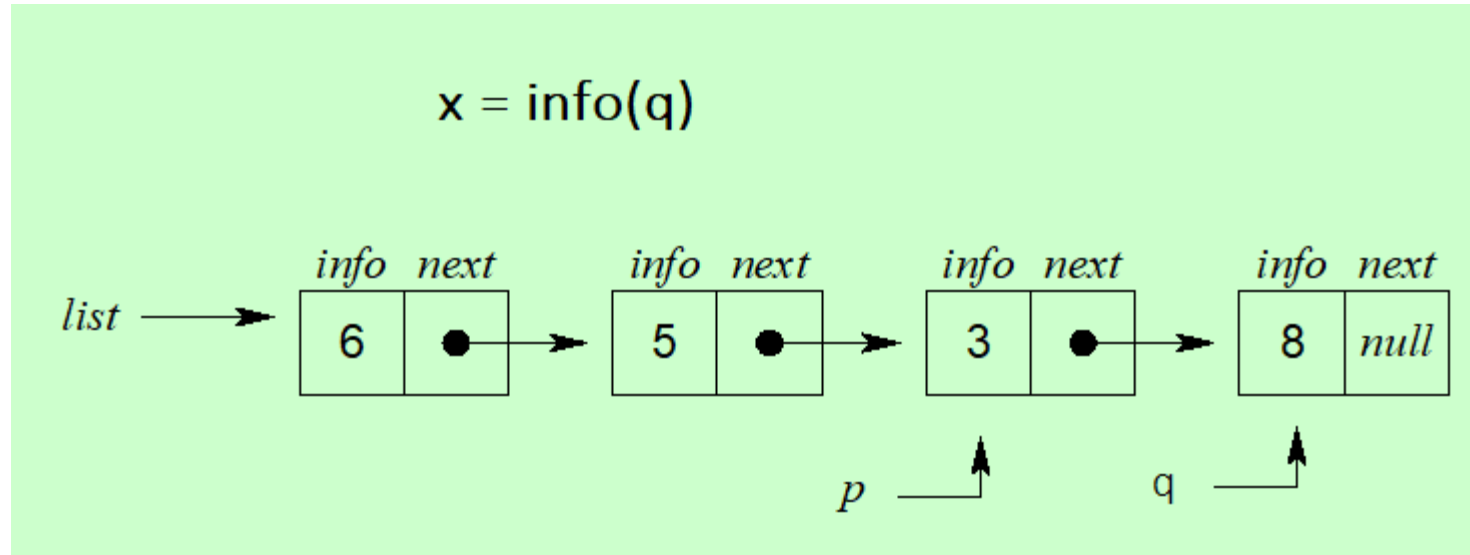
$p = \text{list}$ $q = \text{list} \rightarrow \text{next}$



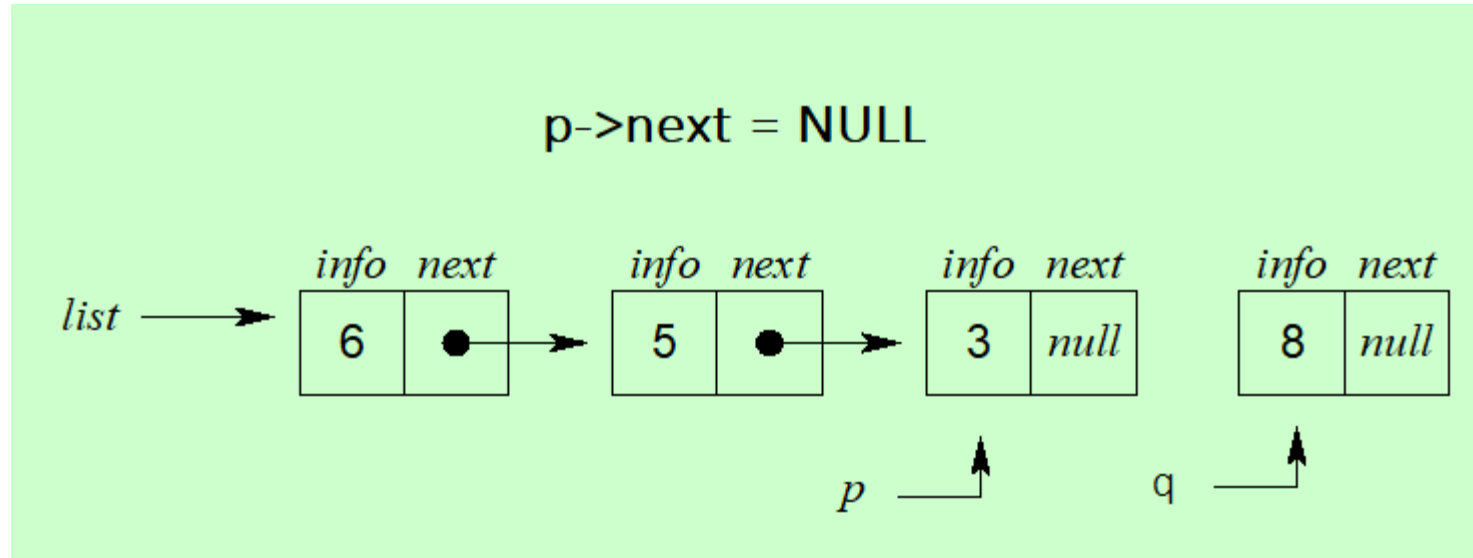
Contd.



Contd.



Contd.

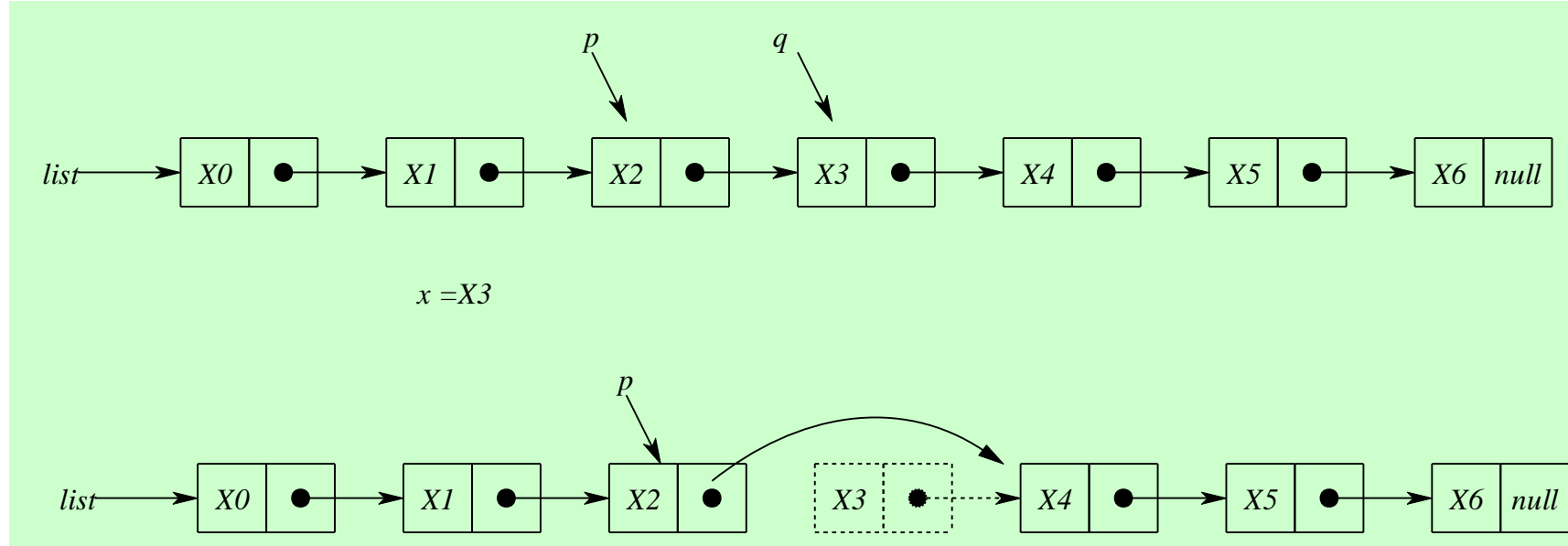


Code

```
void last_delete()
{
    struct node *ptr,*ptr1;
    if(head == NULL)
    {
        printf("\nlist is empty");
    }
    else if(head -> next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nOnly node of the list deleted ...\n");
    }
}
```

```
else
{
    ptr = head;
    while(ptr->next != NULL)
    {
        ptr1 = ptr;
        ptr = ptr ->next;
    }
    ptr1->next = NULL;
    free(ptr);
    printf("\nDeleted Node from the last ...\n");
}
```

Delete after location



Code

```
void random_delete()
{
    struct node *ptr,*ptr1;
    int loc,i;
    printf("\n Enter the location of the node after
which you want to perform deletion \n");
    scanf("%d",&loc);
    ptr=head;
    for(i=0;i<loc;i++)
    {
        ptr1 = ptr;
        ptr = ptr->next;
```

```
        if(ptr == NULL)
        {
            printf("\nCan't delete");
            return;
        }
    }
    ptr1 ->next = ptr ->next;
    free(ptr);
    printf("\nDeleted node %d ",loc+1);
}
```


Searching

```
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
```

```
            if(ptr->data == item)
            {
                printf("item found at location %d ",i+1);
                flag=0;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("Item not found\n");
        }
    }
}
```

Doubly Linked List: Representation

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



Doubly Linked List

Operations

- Insertion
 - at beginning
 - at end
 - after specified node
- Deletion
 - at beginning
 - at end
 - after specified node
- Traversing
- Searching

Codes



Insertion_at_beg.txt



Insertion_at_end.txt



Insertion_specified.txt



Deletion_beg.txt



Deletion_end.txt



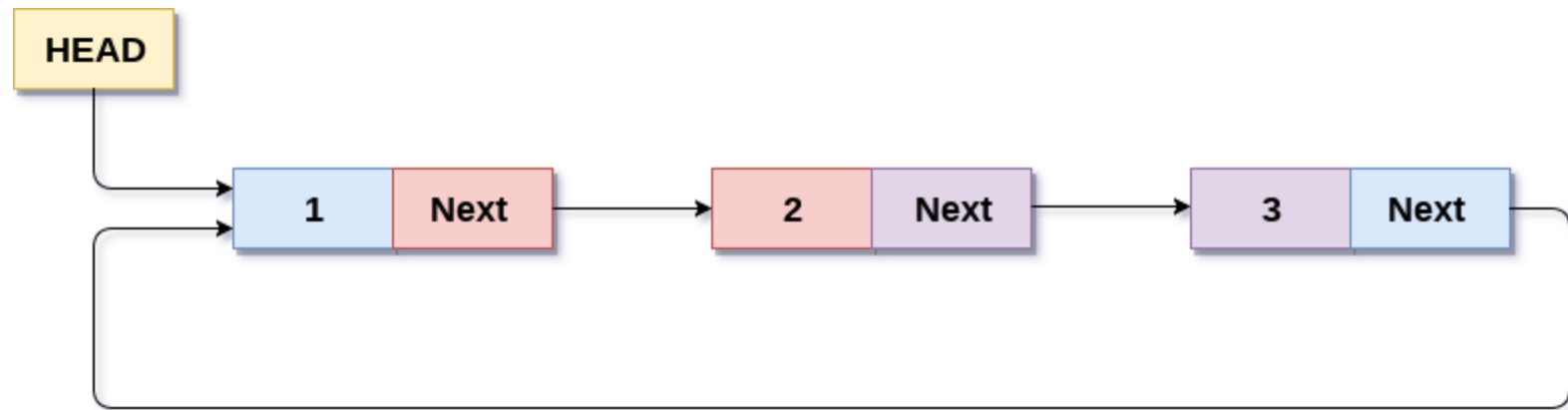
Deletion_specified.txt



Search.txt

Circular Linked List: Representation

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



Circular Singly Linked List

Operations

- Insertion
 - at beginning
 - at end
- Deletion
 - at beginning
 - at end
- Traversing
- Searching

Codes



Insertion_circular_at_beg.txt



Insertion_circular_at_end.txt



Deletion_circular_at_beg.txt

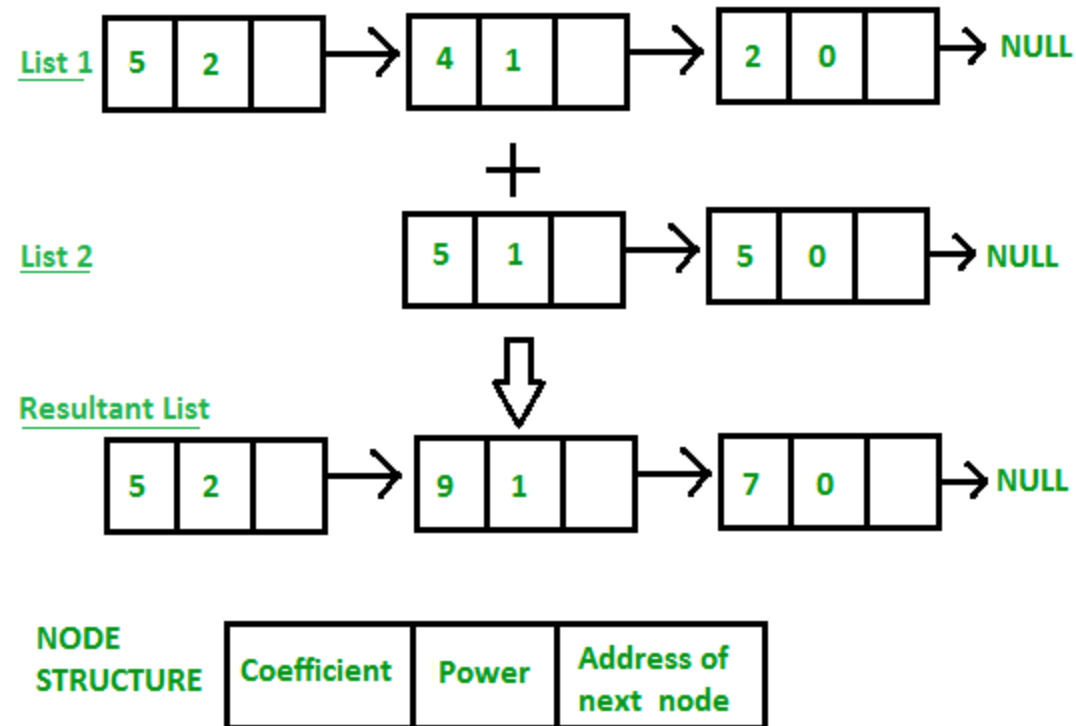


Deletion_circular_at_end.txt



Search_circular.txt

Adding two polynomials



Code

```
// Node structure containing power and coefficient of  
// variable  
struct Node {  
    int coeff;  
    int pow;  
    struct Node* next;  
};
```

Node Creation

```
// Function to create new node
void create_node(int x, int y, struct Node** temp)
{
    struct Node *r, *z;
    z = *temp;
    if (z == NULL) {
        r = (struct Node*)malloc(sizeof(struct Node));
        r->coeff = x;
        r->pow = y;
        *temp = r;
        r->next = (struct Node*)malloc(sizeof(struct
Node));
```

```
        r = r->next;
        r->next = NULL;
    }
    else {
        r->coeff = x;
        r->pow = y;
        r->next = (struct
Node*)malloc(sizeof(struct Node));
        r = r->next;
        r->next = NULL;
    }
}
```

Polynomial Add

```
// Function Adding two polynomial numbers
void polyadd(struct Node* poly1, struct Node* poly2,
             struct Node* poly)
{
    while (poly1->next && poly2->next) {
        // If power of 1st polynomial is greater then 2nd,
        // then store 1st as it is and move its pointer
        if (poly1->pow > poly2->pow) {
            poly->pow = poly1->pow;
            poly->coeff = poly1->coeff;
            poly1 = poly1->next;
        }
    }
}
```

Contd.

```
        // If power of 2nd polynomial  
is greater than 1st,
```

```
        // then store 2nd as it is and  
move its pointer
```

```
        else if (poly1->pow < poly2->  
pow) {
```

```
            poly->pow = poly2->pow;
```

```
            poly->coeff = poly2->coeff;
```

```
            poly2 = poly2->next;
```

```
        }
```

```
        // If power of both polynomial  
numbers is same then
```

```
            // add their coefficients
```

```
            else {
```

```
                poly->pow = poly1->pow;
```

```
                poly->coeff = poly1->coeff +  
poly2->coeff;
```

```
                poly1 = poly1->next;
```

```
                poly2 = poly2->next;
```

```
            }
```

Contd.

```
// Dynamically create new node
```

```
    poly->next
```

```
        = (struct Node*)malloc(sizeof(struct Node));
```

```
    poly = poly->next;
```

```
    poly->next = NULL;
```

```
}
```

Contd.

```
while (poly1->next || poly2->next) {  
    if (poly1->next) {  
        poly->pow = poly1->pow;  
        poly->coeff = poly1->coeff;  
        poly1 = poly1->next;  
    }  
    if (poly2->next) {  
        poly->pow = poly2->pow;
```

```
        poly->coeff = poly2->coeff;  
        poly2 = poly2->next;  
    }  
    poly->next  
        = (struct  
Node*)malloc(sizeof(struct Node));  
    poly = poly->next;  
    poly->next = NULL;  
    }  
}
```

Contd.

```
// Display Linked list
void show(struct Node* node)
{
    while (node->next != NULL) {
        printf("%dx^%d", node->coeff, node->pow);
        node = node->next;
        if (node->coeff >= 0) {
            if (node->next != NULL)
                printf("+");
        }
    }
}
```

Driver Code

```
// Driver code
int main()
{
    struct Node *poly1 = NULL, *poly2 = NULL, *poly = NULL;

    // Create first list of  $5x^2 + 4x^1 + 2x^0$ 
    create_node(5, 2, &poly1);
    create_node(4, 1, &poly1);
    create_node(2, 0, &poly1);

    // Create second list of  $-5x^1 - 5x^0$ 
    create_node(-5, 1, &poly2);
    create_node(-5, 0, &poly2);

    printf("1st Number: ");
    show(poly1);
```

```
    printf("\n2nd Number: ");
    show(poly2);

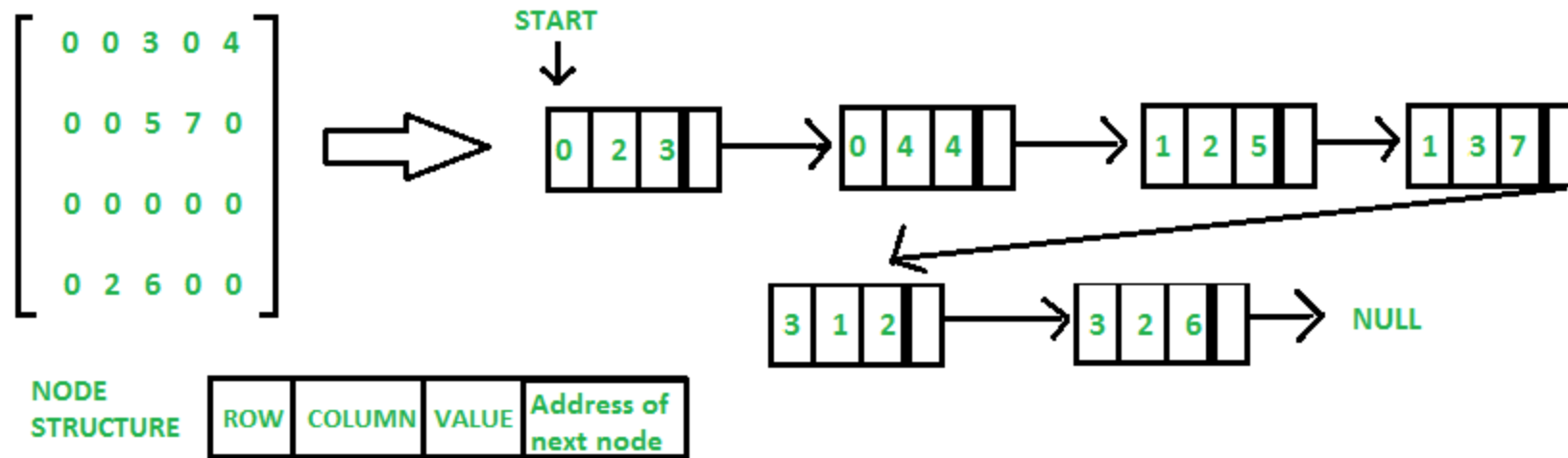
    poly = (struct Node*)malloc(sizeof(struct Node));

    // Function add two polynomial numbers
    polyadd(poly1, poly2, poly);

    // Display resultant List
    printf("\nAdded polynomial: ");
    show(poly);

    return 0;
}
```


Sparse Matrix Representation



Code

```
// C program for Sparse Matrix Representation
// using Linked Lists
#include<stdio.h>
#include<stdlib.h>

// Node to represent sparse matrix
struct Node
{
    int value;
    int row_position;
    int column_postion;
    struct Node *next;
};
```

Contd.

```
// Function to create new node
void create_new_node(struct Node** start, int non_zero_element,
                    int row_index, int column_index)
{
    struct Node *temp, *r;
    temp = *start;
    if (temp == NULL)
    {
        // Create new node dynamically
        temp = (struct Node *) malloc(sizeof(struct Node));
        temp->value = non_zero_element;
        temp->row_position = row_index;
        temp->column_position = column_index;
        temp->next = NULL;
        *start = temp;
    }
}
```

```
else
{
    while (temp->next != NULL)
        temp = temp->next;

    // Create new node dynamically
    r = (struct Node *) malloc(sizeof(struct Node));
    r->value = non_zero_element;
    r->row_position = row_index;
    r->column_position = column_index;
    r->next = NULL;
    temp->next = r;
}
}
```

Contd.

```
// This function prints contents of linked list
// starting from start
void PrintList(struct Node* start)
{
    struct Node *temp, *r, *s;
    temp = r = s = start;

    printf("row_position:");
    while(temp != NULL)
    {

        printf("%d ", temp->row_position);
        temp = temp->next;
    }
    printf("\n");
```

```
printf("column_postion:");
while(r != NULL)
{
    printf("%d ", r->column_postion);
    r = r->next;
}
printf("\n");
printf("Value:");
while(s != NULL)
{
    printf("%d ", s->value);
    s = s->next;
}
printf("\n");
}
```

Contd.

```
// Driver of the program
int main()
{
    // Assume 4x5 sparse matrix
    int sparseMatric[4][5] =
    {
        {0 , 0 , 3 , 0 , 4 },
        {0 , 0 , 5 , 7 , 0 },
        {0 , 0 , 0 , 0 , 0 },
        {0 , 2 , 6 , 0 , 0 }
    };

    /* Start with the empty list */
    struct Node* start = NULL;
```

```
        for (int i = 0; i < 4; i++)
            for (int j = 0; j < 5; j++)

                // Pass only those values which
                are non - zero
                if (sparseMatric[i][j] != 0)
                    create_new_node(&start,
                    sparseMatric[i][j], i, j);

        PrintList(start);

        return 0;
    }
```