

Self-Referencing Structures

Recursive Structure

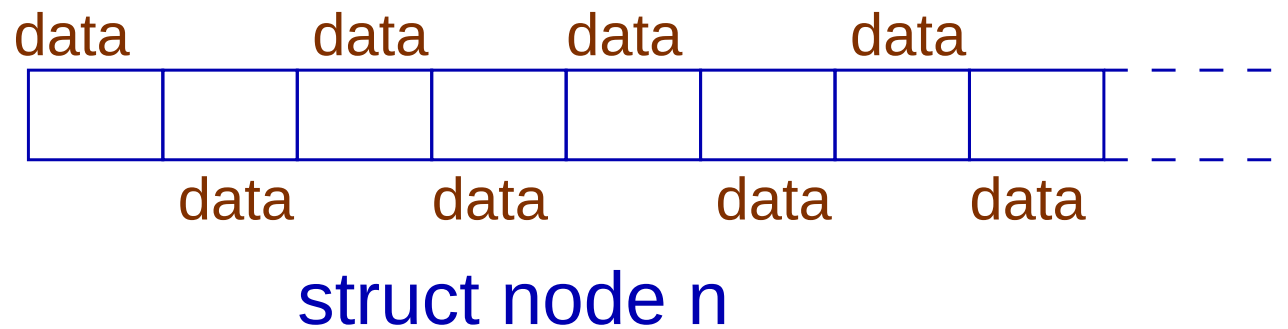
We cannot define the following recursive structure.

```
#include <stdio.h>
struct node {
    int data;
    struct node next;
};
int main() { // selfRef1.c
    struct node n;
    return 0;
}
```

Compilation

```
$ cc -Wall selfRef1.c  
selfRef1.c:7: error: field 'next' has  
incomplete type  
selfRef1.c: In function 'main':  
selfRef1.c:11: warning: unused  
variable 'n'
```

n Demands an Indefinite Amount of Memory



Self-Referencing Pointer

But we can make the allocation **lazy** by introducing a pointer to the same structure.

```
#include <stdio.h>
struct node {
    int data ;
    struct node *next ;
};
int main() { // selfRef2.c
    struct node n;
    n.next = &n ;
    printf("&n: %p\tn.next: %p\n", &n, n.next);
    return 0; }
```

Output

```
$ cc -Wall selfRef2.c
```

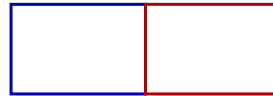
```
$ a.out
```

```
&n:  0xbff52f70 n.next:  0xbff52f70
```

Memory Allocation **n**

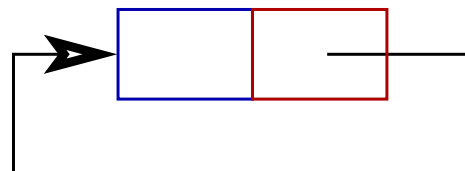
struct node n;

data



n.next = &n;

data next



New Type Names

```
typedef struct node {  
    int data ;  
    struct node *next ;  
} node, *list ;
```

`node` is a type name equivalent to `struct node`. Similarly `list` is also a type name, equivalent to `struct node *`.

New Type Name

```
#include <stdio.h>
typedef struct node {
    int data ;
    struct node *next ;
} node, *list ;
int main() { // selfRef3.c
    node n;
    list l;
    l = n.next = &n ;
    printf("&n: %p\t n.next: %p\t l: %p\n", &n, n.next, l);
    return 0;
}
```

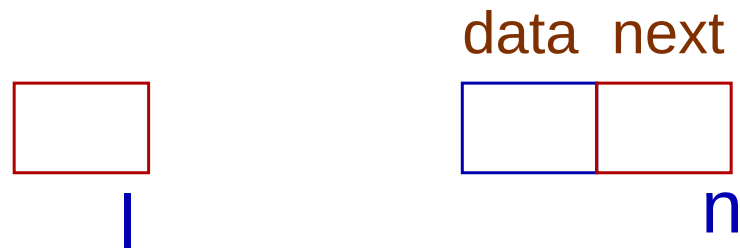
Output

```
$ cc -Wall selfRef3.c
```

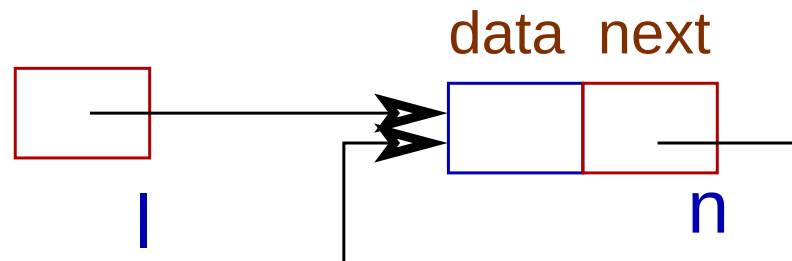
```
$ a.out
```

```
&n:  0xbff8f7e0 n.next:  0xbff8f7e0 l:  
0xbff8f7e0
```

node n; list l;



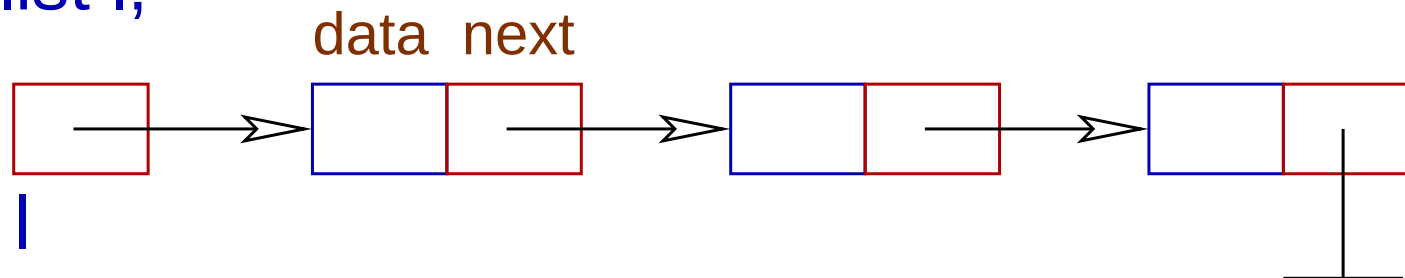
`l = n.next = &n ;`



Linked List

We can create a **linked list** of data by dynamically creating nodes of self-referencing structure and connecting them by the **next** pointer. The next pointer of the last node is **NULL**.

list l;



A Example

Following C program creates a **singly linked-list** where the last data is at the head of the list.

We can travel the list, iteratively or recursively, and print data in **last-in-first-out (LIFO)** or **first-in-first-out (FIFO)** order.

C Program

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node {
    int data ;
    struct node *next ;
} node, *list ;
int printList(list);
int printListR(list);
int printList0(list);

int main() { // selfRef4.c
    list l=NULL, current=NULL;
```

```
int n, data;

printf("Enter data, terminate by Ctrl+D\n");
while(scanf("%d", &data) != EOF){
    l = (list)malloc(sizeof(node));
    l->data = data; l->next = current;
    current = l; // Last data at the head, LIFO
}
printf("Input data are: ");
n = printList(l);
//    n = printListR(l);
//    n = printListO(l);
putchar('\n');
printf("Data count: %d\n", n);
```

```
        return 0;
    }
    int printList(list l){
        int count=0;
        while(l) {
            printf("%d ", l->data);
            l = l -> next;
            ++count;
        }
        return count;
    }
    int printListR(list l){
        if(l){
            printf("%d ", l->data);
```

```
        return printListR(l->next) + 1;
    }
    return 0;
}

int printList0(list l){
    if(l){
        int temp = printList0(l->next) + 1;
        printf("%d ", l->data);
        return temp;
    }
    return 0;
} // selfRef4.c
```

A Example

Following C program creates a **singly linked-list** where the first data is at the head of the list. We can travel the list, iteratively or recursively, and print data in **first-in-first-out (FIFO)** or **last-in-first-out (LIFO)** order.

C Program

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node {
    int data ;
    struct node *next ;
} node, *list ;
int printList(list);
int printListR(list);
int printList0(list);

int main() { // selfRef5.c
    list l=NULL, current=NULL;
```

```
int n, data;

printf("Enter data, terminate by Ctrl+D\n");
if(scanf("%d", &data) != EOF){
    l = (list)malloc(sizeof(node));
    l->data = data;
    current = l;
    while(scanf("%d", &data) != EOF){
        current -> next = (list)malloc(sizeof(node));
        current = current -> next;
        current->data = data;
    } // First data is at the head of the list
}
if(current) current -> next = NULL;
```

```
        printf("Input data are: ");  
//      n = printList(l);  
//      n = printListR(l);  
n = printListO(l);  
putchar('\n');  
printf("Data count: %d\n", n);  
return 0;  
}  
  
int printList(list l){  
    int count=0;  
    while(l) {  
        printf("%d ", l->data);  
        l = l -> next;  
    }
```

```
        ++count;
    }
    return count;
}

int printListR(list l){
    if(l){
        printf("%d ", l->data);
        return printListR(l->next) + 1;
    }
    return 0;
}

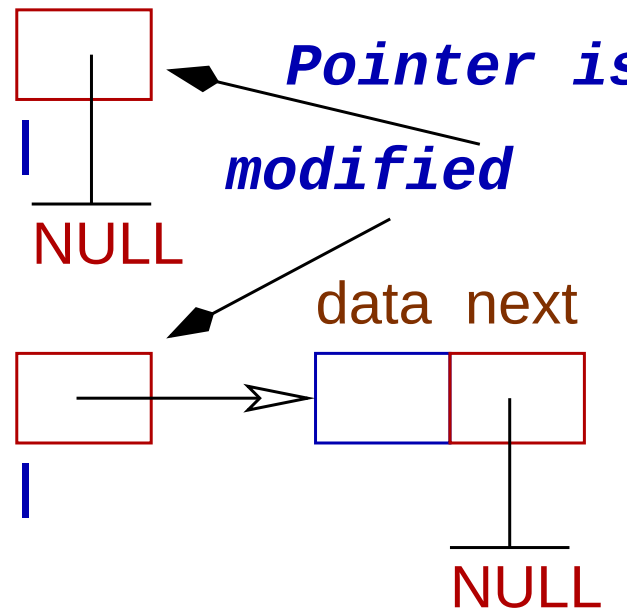
int printList0(list l){
    if(l){
        int temp = printList0(l->next) + 1;
```

```
        printf("%d ", l->data);  
        return temp;  
    }  
    return 0;  
} // selfRef5.c
```

Insert Function

We can write function to insert data in a list. But there is one important point to remember. Our **list** is a variable **l** of type pointer to **node**. When we insert a new node with data, the pointer may get modified.

list l;



Insert Function

The parameter to the *insert* function cannot be **1** but a pointer to it (call by value). Note that **1** itself is a pointer and we have to deal with **pointer to a pointer** inside the insert function.

Insert Function

The other option is to get the modified address returned as a value. There is a third option, we can have a **dummy node** and the whole list starts after the **dummy node**.

Insert at Head I

```
#include <stdio.h>
#include <stdlib.h>
#define ERROR -1
#define OK 0
typedef struct node {
    int data ;
    struct node *next ;
} node, *list ;
int printList(list);
int insertAtHead1(list *, int);
int main() { // insertList1.c
    list l=NULL;
```

```
int err, n, data;

printf("Enter data, terminate by Ctrl+D\n");
while(scanf("%d", &data) != EOF){
    err = insertAtHead1(&l, data);
    if(err == ERROR) printf("malloc error\n");
}
printf("Input data are: ");
n = printList(l);
putchar('\n');
printf("Data count: %d\n", n);
return 0;
}

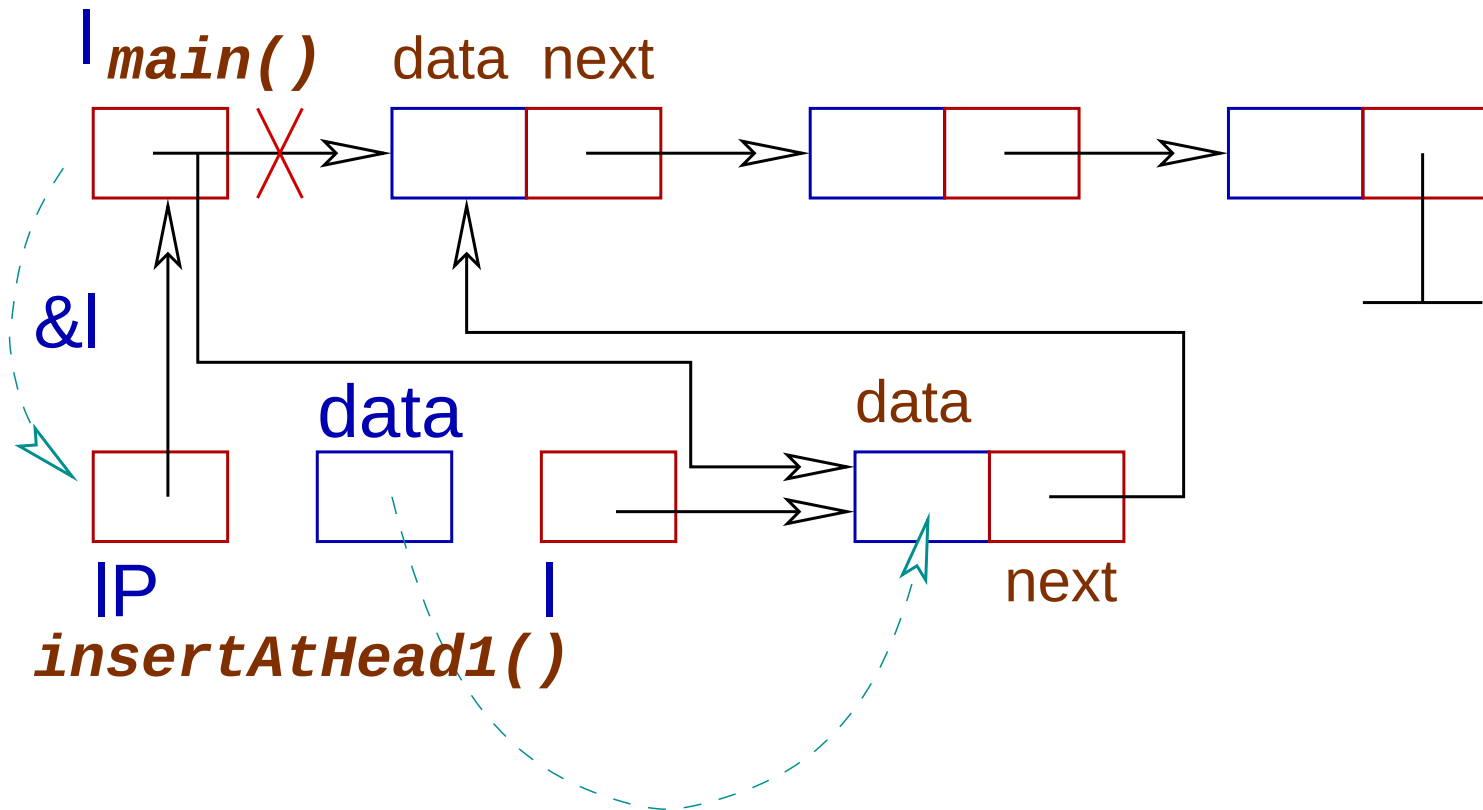
int insertAtHead1(list *lP, int data){
```

```
list l ;

l = (list)malloc(sizeof(node));
if(l == NULL) return ERROR;
l->data = data; l -> next = *lP;
*lP = l;
return OK;
} // insertList1.c

int printList(list l){
    int count=0;
    while(l) {
        printf("%d ", l->data);
        l = l -> next;
        ++count;
    }
}
```

```
    }  
    return count;  
} // insertList1.c
```



Insert at Head II

```
list insertAtHead2(list l, int data, int *eP){  
    list t ;  
    t = (list)malloc(sizeof(node));  
    if(t == NULL) {*eP = ERROR; return NULL;}  
    t->data = data; t -> next = l;  
    *eP = OK; return t;  
} // insertList2.c
```

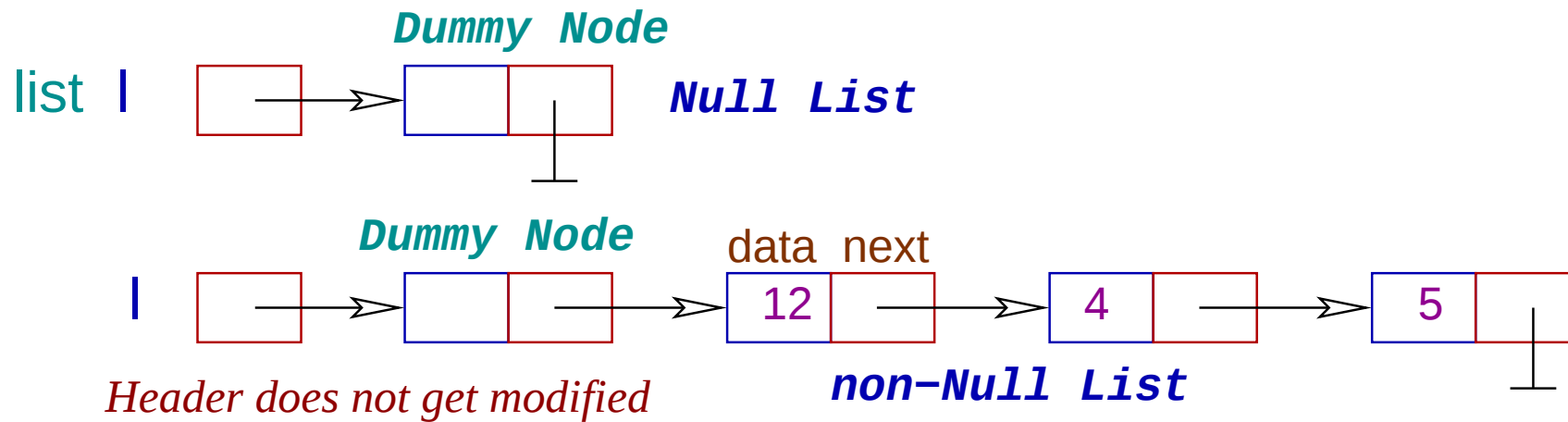
Called As

```
int main() { // insertList2.c
    list l=NULL;
    int err, n, data;
    printf("Enter data, terminate by Ctrl+D\n");
    while(scanf("%d", &data) != EOF){
        l = insertAtHead2(l, data, &err);
        if(err == ERROR) printf("malloc error\n");
    }
}
```

Insert at Head III

```
int insertAtHead3(list l, int data){  
    list t ;  
    t = (list)malloc(sizeof(node));  
    if(t == NULL) return ERROR;  
    t->data = data; t -> next = l->next;  
    l->next = t; return OK;  
} // insertList3.c
```

Dummy Node



CreateList()

```
list createList(){  
    list l = (list)malloc(sizeof(node));  
    l->next=NULL;  
    return l;  
} // Creating dummy node
```

Called As

```
int main() { // insertList3.c
    list l;
    int err, n, data;
    l = createList();
    printf("Enter data, terminate by Ctrl+D\n");
    while(scanf("%d", &data) != EOF){
        err = insertAtHead3(l, data);
        if(err == ERROR) printf("malloc error\n");
    }
}
```

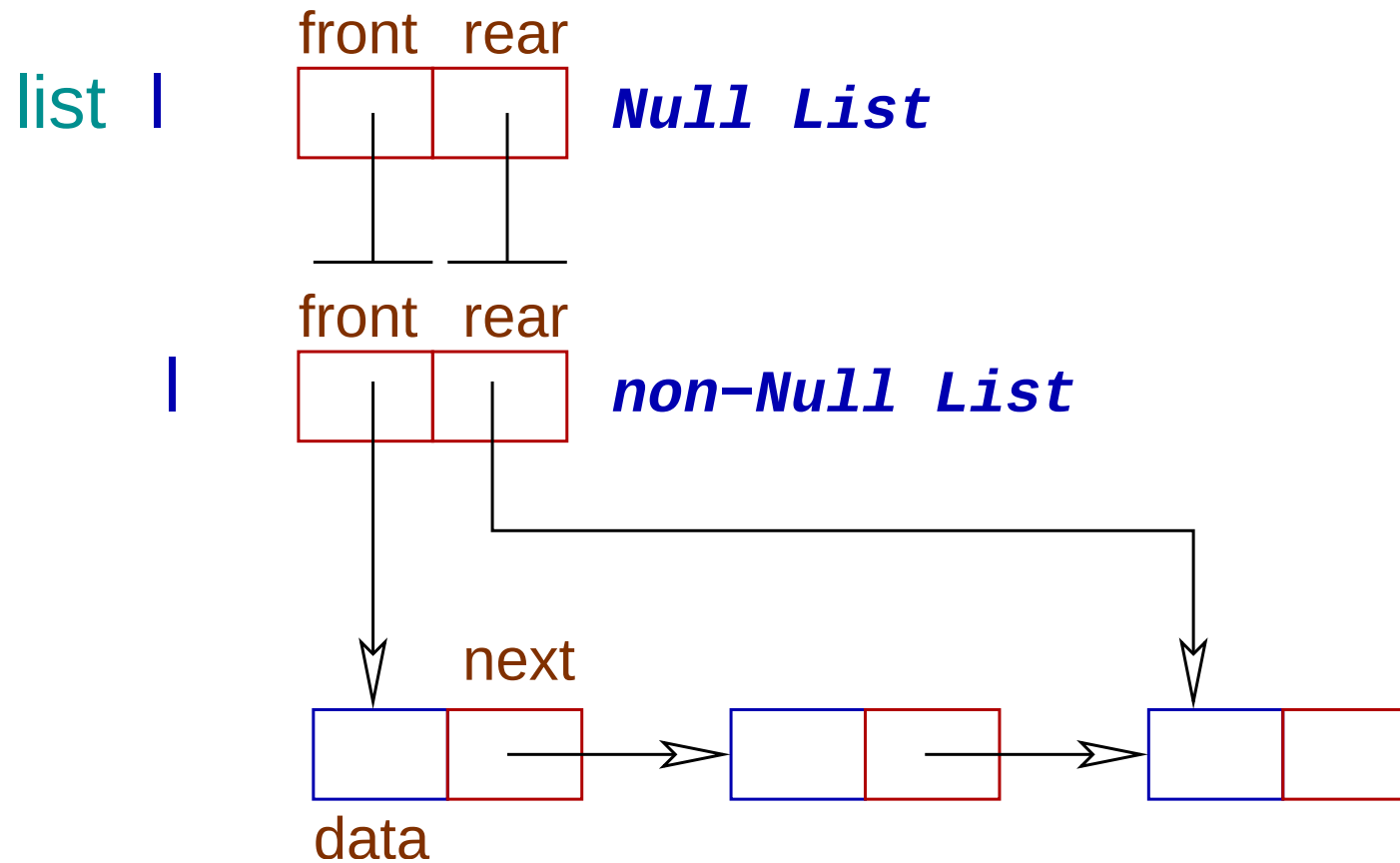
Note

Insertion at the head costs $O(1)$ when there are n data present in the list. But insertion at the tail costs $O(n)$ unless there is another pointer pointing at the tail.

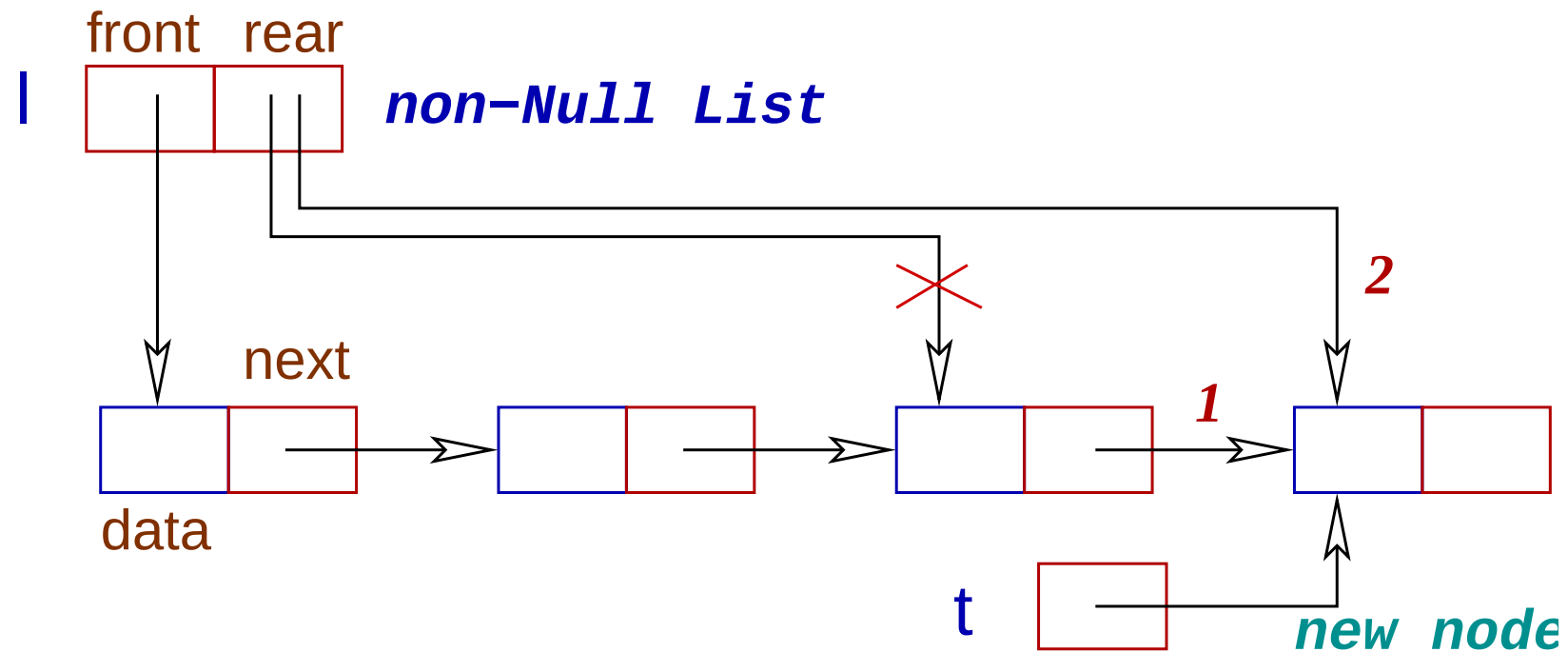
Insert at Tail I

```
int insertAtTail1(list *lP, int data){
    list t ;
    t = (list)malloc(sizeof(node));
    if(t == NULL) return ERROR;
    t->data = data; t -> next = NULL;
    if(*lP == NULL) *lP = t;
    else{
        list curr = *lP;
        while(curr->next != NULL) curr=curr->next;
        curr -> next = t;
    } return OK;
} // insertList4.c
```

Front and Rear Pointers



Insert at the End



Insert at Tail II

```
#include <stdio.h>
#include <stdlib.h>
#define ERROR -1
#define OK 0
typedef struct node {
    int data ;
    struct node *next ;
} node ;
typedef struct {node *front, *rear;} list;
int printList(list);
int insertAtTail2(list *, int);
```

```
int main() { // insertList5.c
    list l={NULL, NULL};
    int err, n, data;

    printf("Enter data, terminate by Ctrl+D\n");
    while(scanf("%d", &data) != EOF){
        err = insertAtTail2(&l, data);
        if(err == ERROR) printf("malloc error\n");
    }
    printf("Input data are: ");
    n = printList(l);
    putchar('\n');
    printf("Data count: %d\n", n);
    return 0;
```

```
}  
int insertAtTail2(list *lP, int data){  
    node *t ;  
  
    t = (node *)malloc(sizeof(node));  
    if(t == NULL) return ERROR;  
    t->data = data; t->next = NULL;  
    if(lP->front == NULL && lP->rear == NULL)  
        lP->front = lP->rear = t;  
    else {  
        lP->rear->next = t;  
        lP->rear = t;  
    }  
    return OK;  
}
```

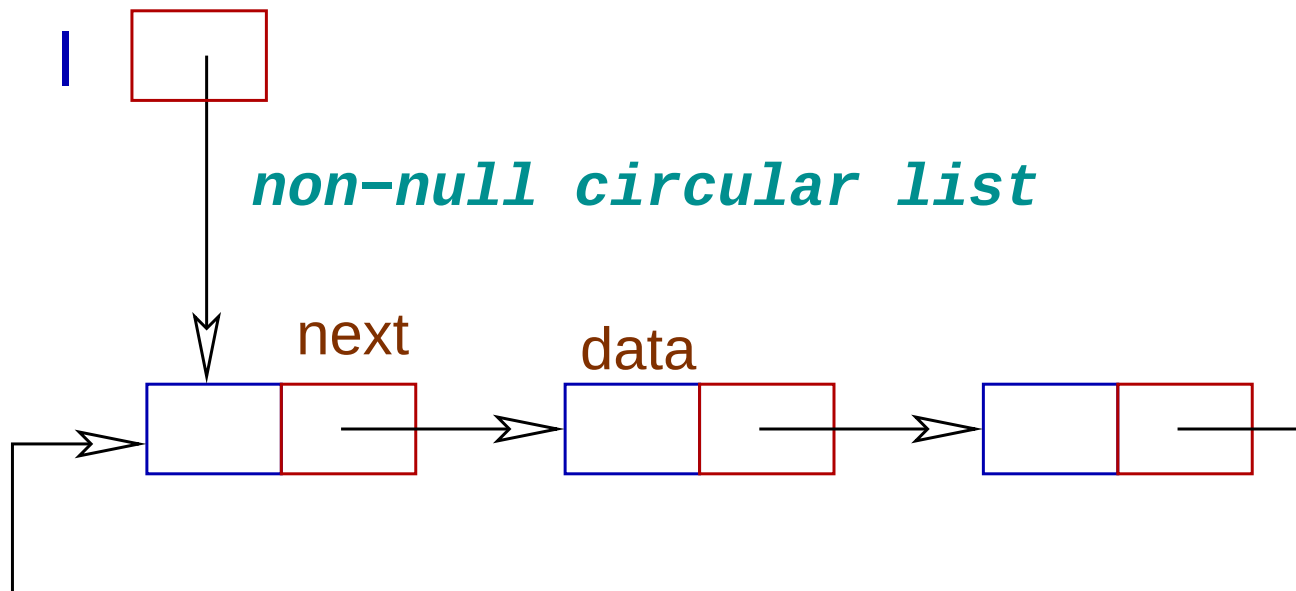
```
} // insertList5.c  
int printList(list l){  
    int count=0;  
    node *t = l.front;  
    while(t) {  
        printf("%d ", t->data);  
        t = t -> next;  
        ++count;  
    }  
    return count;  
} // insertList5.c
```

Circular List

A list may be circular so that the last node points to the first node. In effect there is no beginning or end.

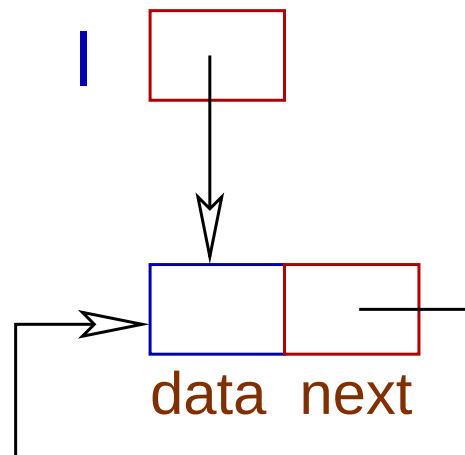
Circular List

list |  *null list*

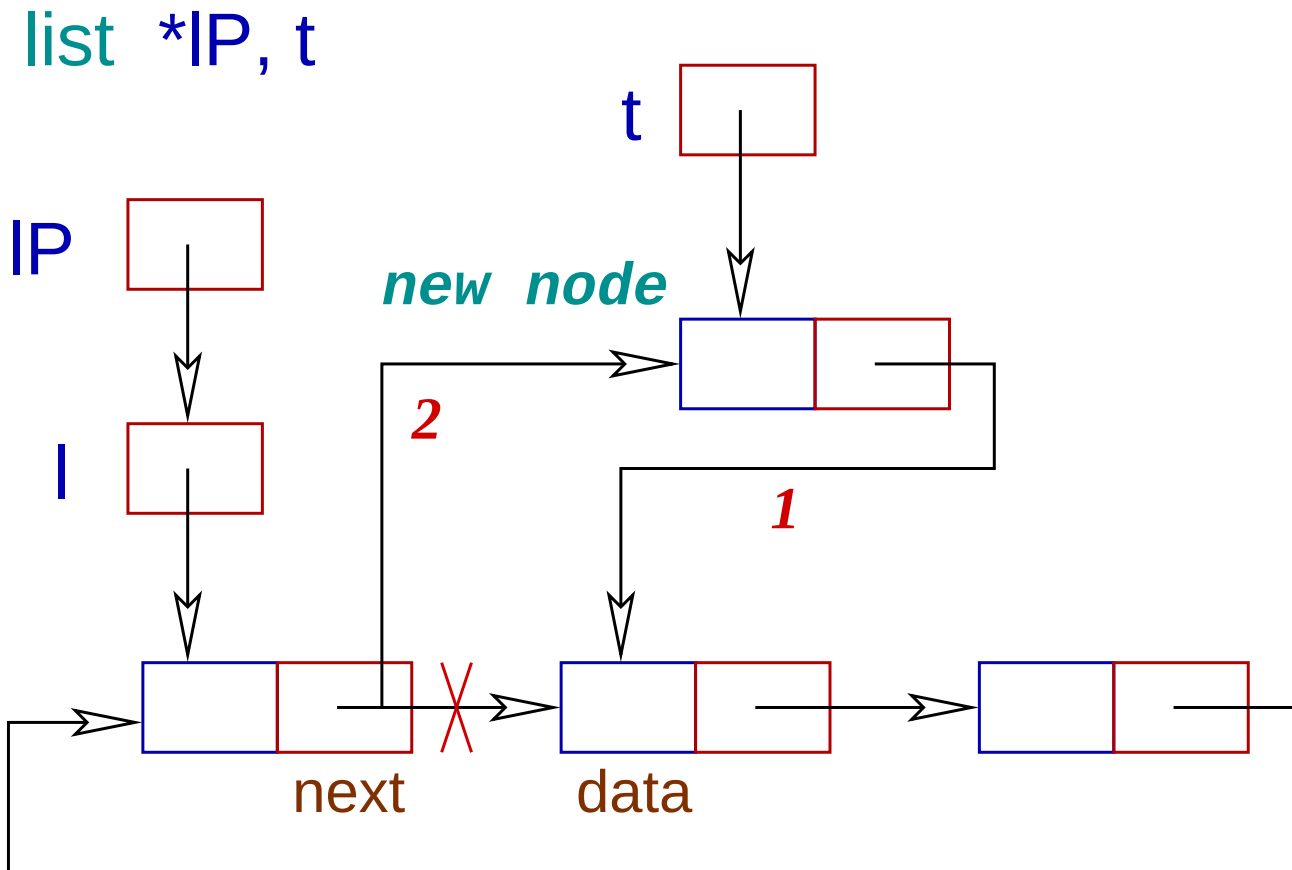


Insert in NULL List

list | 
null list



Insert in a non-NULL List



Circular List

```
int insertHCircular(list *lP, int data){
    list t ;

    t = (list)malloc(sizeof(node));
    if(t == NULL) return ERROR;
    t->data = data;
    if(*lP == NULL) t->next = t;
    else{
        t->next = (*lP)->next;
        (*lP)->next = t;
    }
    *lP = t;
}
```

```
    return OK;  
} // insertList6.c
```

Print Circular List

```
int printList(list l){
    list t;
    int count=0;
    if(l != NULL) {
        t = l->next; // FIFO order
        do {
            printf("%d ", t->data);
            t = t -> next;
            ++count;
        } while (t != l->next);
    }
    return count;
}
```

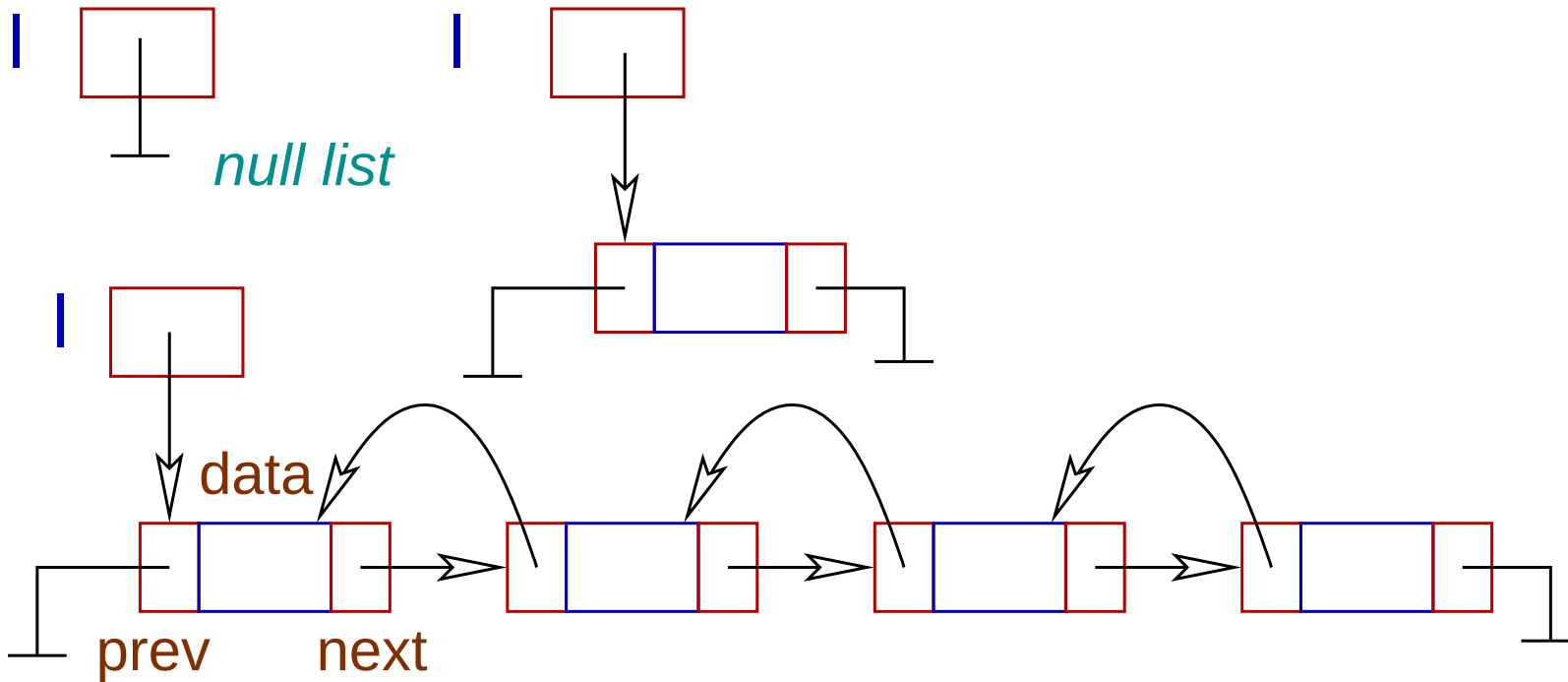
```
} // insertList6.c
```

Doubly Linked List

There are two pointers in a doubly linked list (two-way list). If a node is a terminal node one of the pointers is null (for single node both are null). For all other nodes one pointer points to the previous node and the other pointer points to the next node.

Doubly Linked List

list |

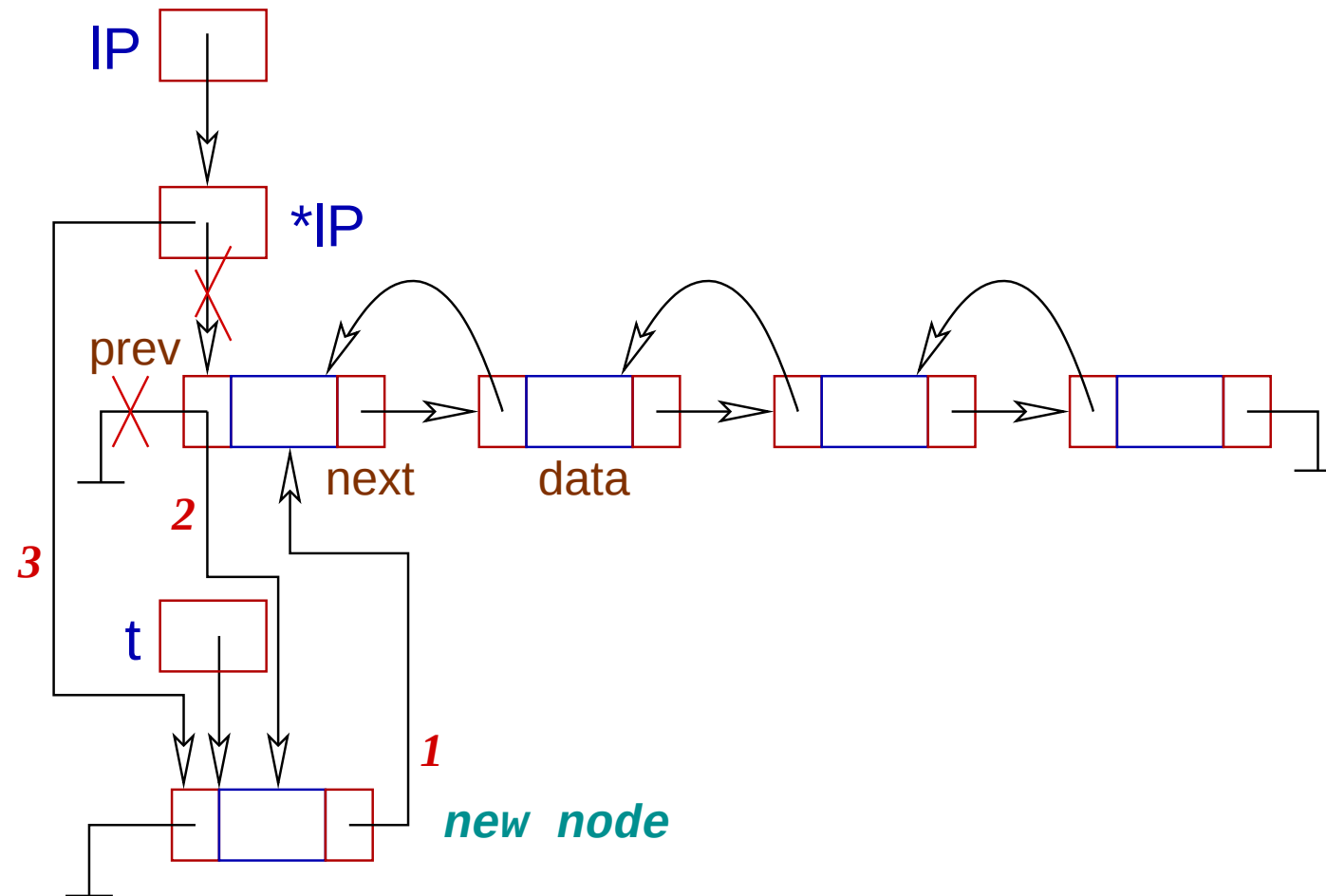


Insert in Doubly Linked List

```
int insertDLList(list *lP, int data){  
    list t ;  
  
    t = (list)malloc(sizeof(node));  
    if(t == NULL) return ERROR;  
    t->data = data; t->prev = NULL;  
    t->next = *lP ;  
    if(*lP) (*lP)->prev = t;  
    *lP = t;  
    return OK;  
} // insertList7.c
```

Insert in Doubly Linked List

list *IP, t



Print Doubly Linked List

```
int printList(list l){
    list t;
    int count=0;
    while(l) {
        printf("%d ", l->data);
        t = l;
        l = l -> next;
        ++count;
    }
    putchar('\n');
    while(t) {
        printf("%d ", t->data);
```

```
        t = t -> prev;  
    }  
    return count;  
} // insertList7.c
```

Ordered List

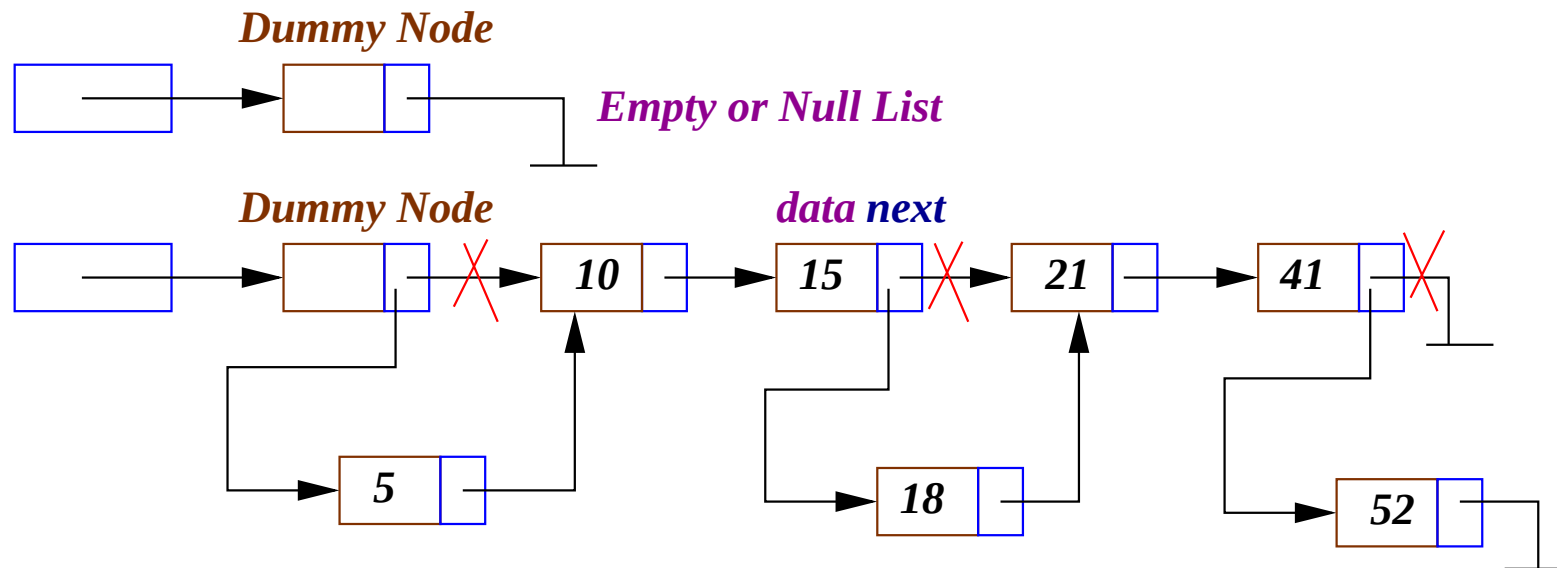
We consider an **ordered-list**, where data items are kept in order (say ascending). Following are important operations on such a list.

- Insertion of a new data (order is maintained),
- Search for a data,
- Deletion of a data,
- Printing the data present in the list.

Ordered List: Implementation

We consider a singly-linked list with a dummy node to store the ordered list.

Insert in an Ordered List



Note

We try to locate the **successor** from the **predecessor** node. If no successor is found, the new node is inserted at the end. If no predecessor is there, it is inserted after the dummy node. The **next** field of the inserted node always take the value of **next** field of its predecessor.

Insert Code

```
int insertOrdered(list l, int data){
    list t ;

    t = (list)malloc(sizeof(node));
    if(t == NULL) return ERROR;
    t->data = data;
    while(l->next != NULL &&
           l->next->data <= data) l=l->next;
    t->next = l->next;
    l->next = t; return OK;
} // insertList8.c
```

Search for a Key

When a **key** is searched in an ordered-list and the first node with the key value is found, the address of the node is returned. If no such node is found, the function returns **null**.

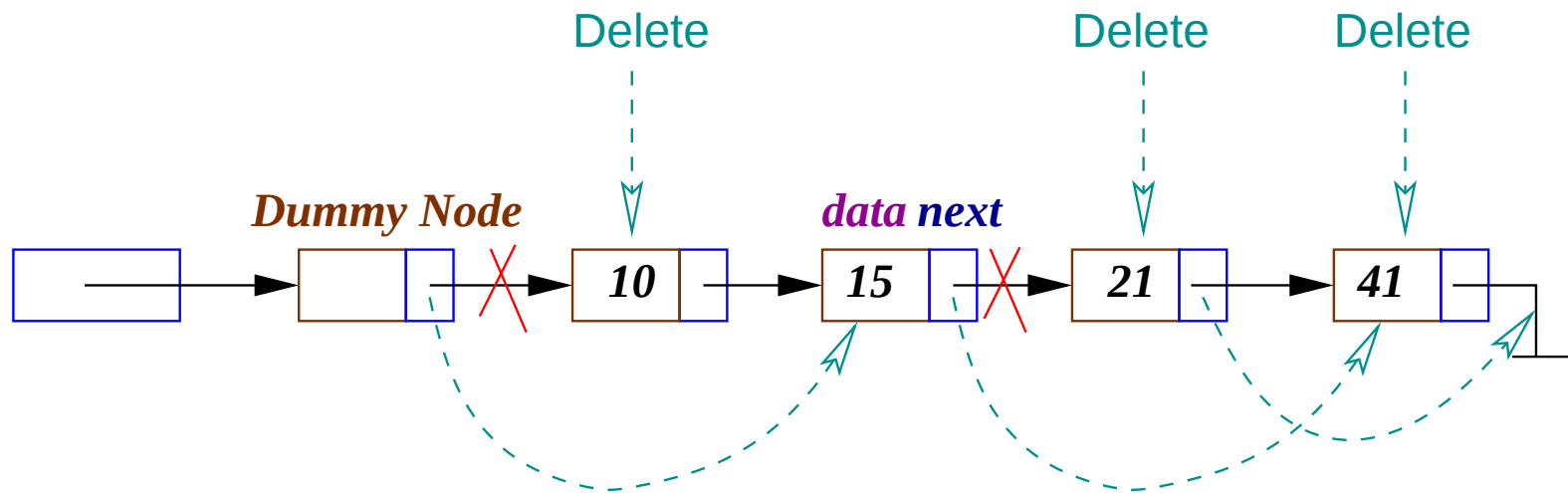
Search Code

```
list searchList(list l, int k){  
    l=l->next;  
    while(l){  
        if(l->data == k) return l;  
        l=l->next;  
    }  
    return NULL;  
}
```

Delete a Key

To delete a key (node having the key) from an ordered list, we first search for the key in the list. If the search fails, delete also fails. But if the search returns a valid address of a node, we locate its predecessor and make it point to the successor of the node. We also free the deleted node. The **next** field of the predecessor node (or the dummy node) takes the **next** field value of the deleted node.

Delete From an Ordered List



Delete Code

```
int deleteData(list l, int k){  
    list l1 = searchList(l,k);  
    if(l1 == NULL) return 0;  
    while(l->next != l1) l=l->next;  
    l->next = l1->next;  
    free(l1);  
    return 1;  
}
```

Recursive Functions

- A recursive function for `insert` should not have `malloc()` within it. We create a node and pass it in `insert`.

Recursive Insert Code

```
void insertOrderedR(list l, list dl){  
    if(l->next == NULL ||  
        l->next->data > dl->data) {  
        dl->next=l->next;  
        l->next=dl;  
        return;  
    }  
    insertOrderedR(l->next, dl);  
} // insertList9.c
```

Calling insertOrderedR

```
while(scanf("%d", &data) != EOF){  
    l1 = (list)malloc(sizeof(node));  
    l1->data = data;  
    insertOrderedR(l, l1);  
}
```

Recursive Search Code

```
list searchListR(list l, int k){  
    if(l->next == NULL) return NULL;  
    if(l->next->data == k) return l->next;  
    return searchListR(l->next, k);  
}
```

Recursive Delete Code

```
int deleteDataR(list l, int k){  
    list l1 = searchListR(l,k);  
    if(l1 == NULL) return 0;  
    if(l->next == l1) {  
        l->next = l1->next;  
        free(l1);  
        return 1;  
    }  
    return deleteDataR(l->next, k);  
}
```

Another Recursive Delete Code

```
int deleteDataR(list l, int k){  
    if(l->next == NULL) return 0;  
    if(l->next->data == k){  
        list t = l->next ;  
        l->next = t->next;  
        free(t); return 1;  
    }  
    return deleteDataR(l->next, k);  
}
```