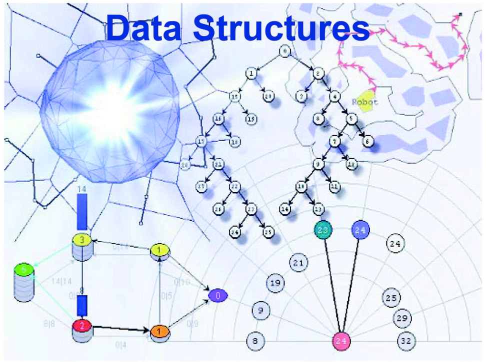
BBM 201
DATA STRUCTURES

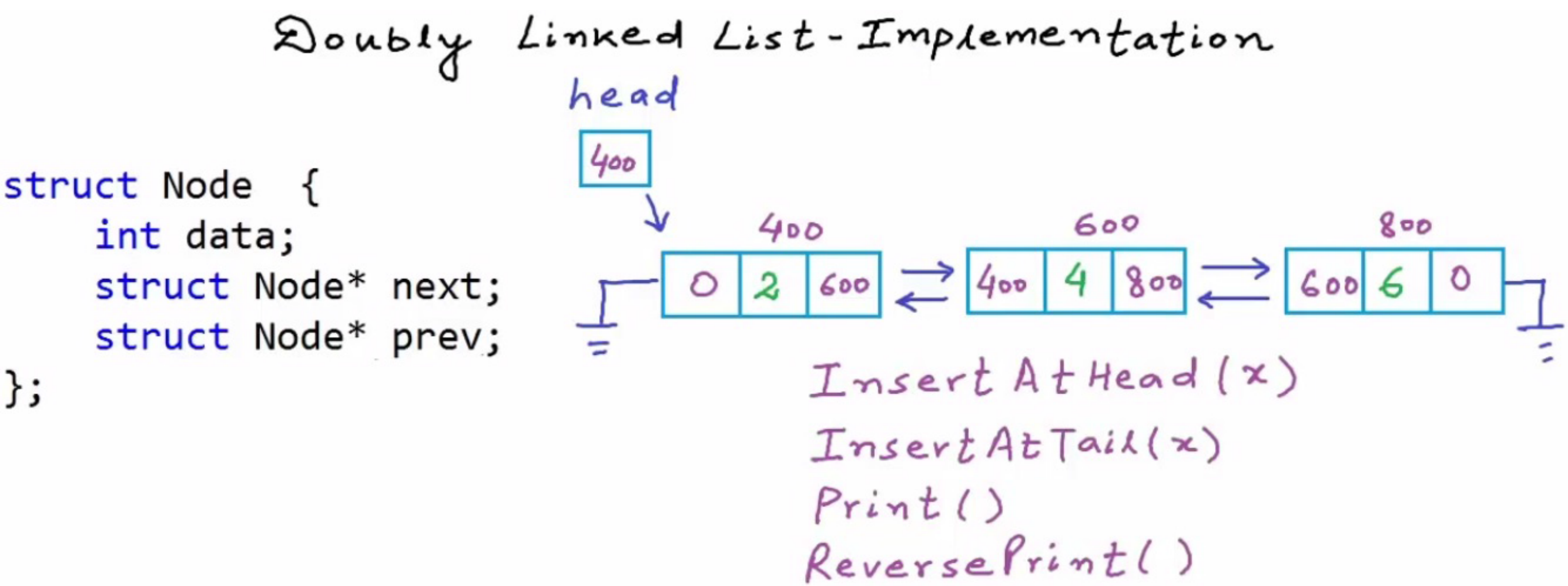
Lecture 10:

Doubly Linked Lists

2018-2019 Fall



Doubly Linked Lists



Each node stores not only the address of the next node, but also the address of the previous node. So, each node stores three fields.

For temp being 600, temp-> next points to the address 800 and temp->prev points to the address 400.

```c
/* Doubly Linked List implementation */
#include<stdio.h>
#include<stdlib.h>
struct Node  {
    int data;
    struct Node* next;
    struct Node* prev;
};
struct Node* head; // global variable - pointer to head node.
void InsertAtHead(int x) {
    // local variable
    // Will be cleared from memory when function call will finish
    struct Node myNode;
    myNode.data = x;
    myNode.prev = NULL;
    myNode.next = NULL;
}
```

Note: head is a global variable. Each node inside the InsertAtHead function is created locally and the node myNode does not exist after the function is executed.
Therefore, local node allocation is **NOT** preferred.

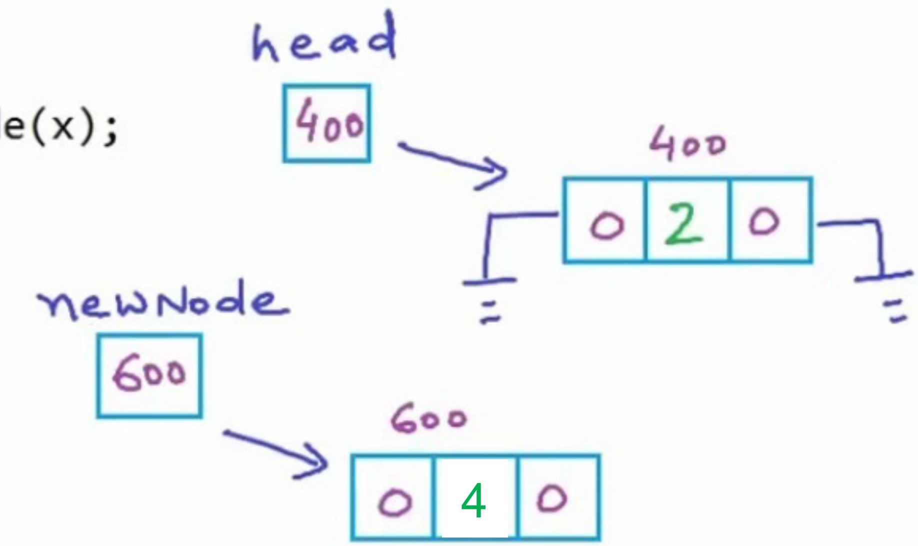
```c
struct Node  {
    int data;
    struct Node* next;
    struct Node* prev;
};
struct Node* head; // global variable - pointer to head node.
struct Node* GetNewNode(int x) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = x;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}
void InsertAtHead(int x) {

}
```

Now, we create a new node in a separate function, called GetNewNode.

Preferred Method:
Each "newNode" is created in the dynamic memory and the node exists after the function is executed.

Doubly Linked List - Implementation

```c
void InsertAtHead(int x) {
    struct Node* newNode = GetNewNode(x);
    if(head == NULL) {
        head = newNode;
        return;
    }
    head->prev = newNode;
    newNode->next = head;
    head = newNode;
}
```



Insert AtHead (2)

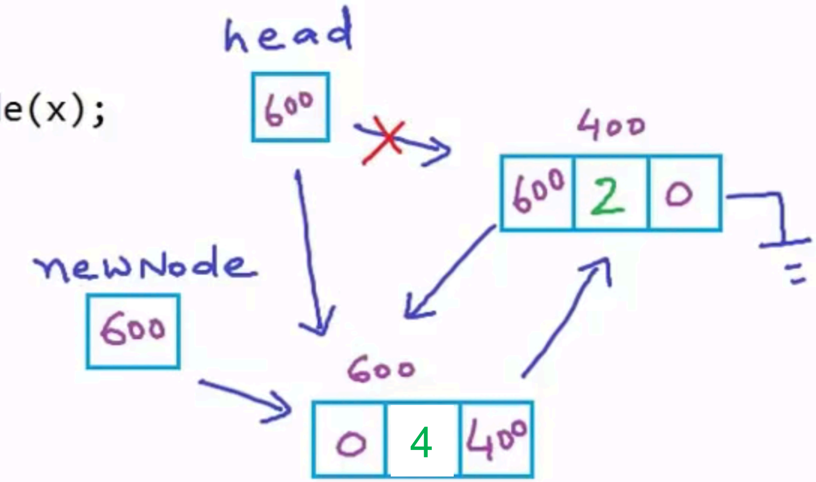
Insert AtHead (4)

Now, one node is created in the list with head pointing to it using the line head = newNode.

We have two nodes, head is pointing to the node at address 400 and newNode is pointing to the node at address 600.

Doubly Linked List - Implementation

```
void InsertAtHead(int x) {
    struct Node* newNode = GetNewNode(x);
    if(head == NULL) {
        head = newNode;
        return;
    }
    head->prev = newNode;
    newNode->next = head;
    head = newNode;
}
```



InsertAtHead(2)

InsertAtHead(4)

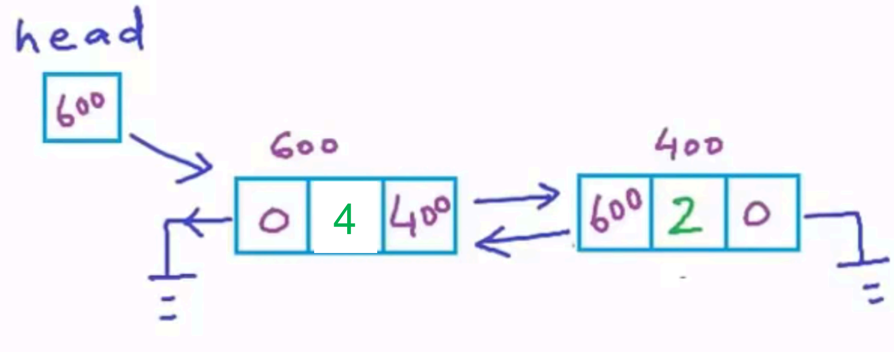
Set the prev-field of the head node as 600 (address of the new node).
Then, set the next-field of the new node as 400 (the address of the head node).
And now, head can point to 600, that is the address of the final head node.

Doubly Linked List - Implementation

```
void InsertAtHead(int x) {
    struct Node* newNode = GetNewNode(x);
    if(head == NULL) {
        head = newNode;
        return;
    }
    head->prev = newNode;
    newNode->next = head;
    head = newNode;
}
```

InsertAtHead(2)

InsertAtHead(4)

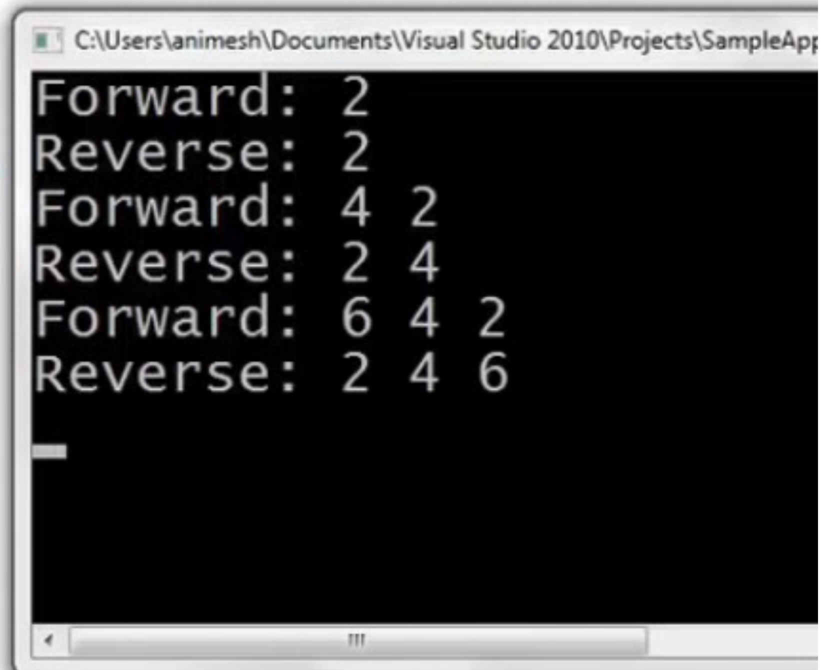


Reverse Printing

```c
void ReversePrint() {
    struct Node* temp = head;
    if(temp == NULL) return; // empty list, exit
    // Going to last Node
    while(temp->next != NULL) {
        temp = temp->next;
    }
    // Traversing backward using prev pointer
    printf("Reverse: ");
    while(temp != NULL) {
        printf("%d ",temp->data);
        temp = temp->prev;
    }
    printf("\n");
}
```

the code first goes to the end of the list and then traverses backwards.

```c
void ReversePrint() {
    struct Node* temp = head;
    if(temp == NULL) return; // empty list, exit
    // Going to last Node
    while(temp->next != NULL) {
        temp = temp->next;
    }
    // Traversing backward using prev pointer
    printf("Reverse: ");
    while(temp != NULL) {
        printf("%d ",temp->data);
        temp = temp->prev;
    }
    printf("\n");
}

int main() {
    head = NULL; // empty list.
    InsertAtHead(2); Print(); ReversePrint();
    InsertAtHead(4); Print(); ReversePrint();
    InsertAtHead(6); Print(); ReversePrint();
}
```



Doubly vs. Singly Linked List

Doubly Linked List

- Uses extra space for previous pointers.
- Requires extra work for Insertion/Deletion.
- Has ready Access/Insert on oth ends.
- Can work as a Queue and a Stack at the same time.
- Does not require additional pointers for Node Deletion.

Reverse a doubly linked list

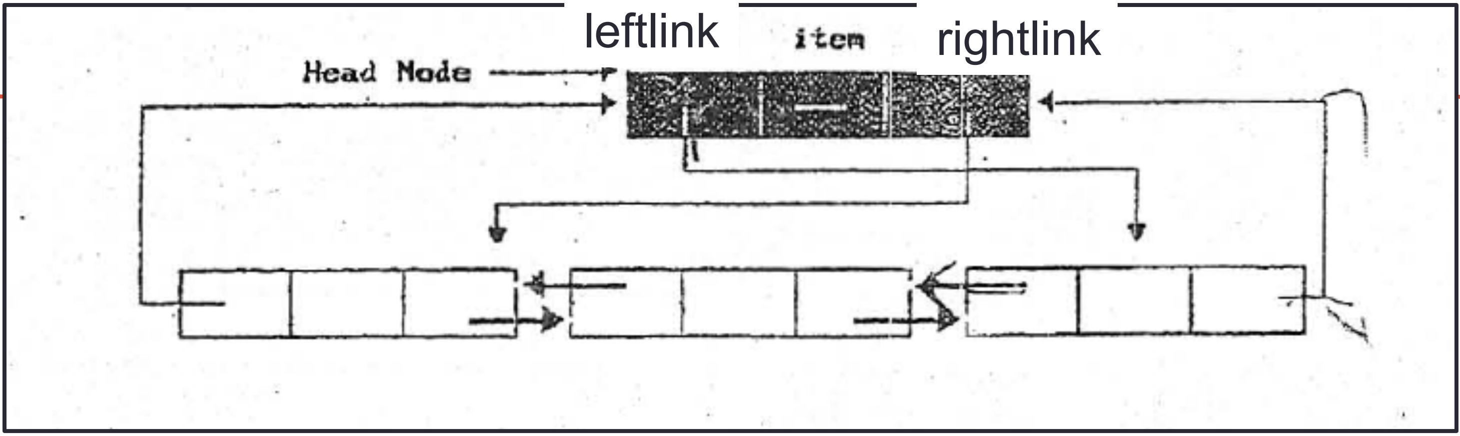
```
struct Node* reverse(struct Node* head)
{
    struct Node* n = head, next;
    //running till the last node
    while(n->next != NULL){
        next = n->next;
        n->next = n->prev;
        n->prev = next;
        n = next;
    }
   //for the last node
   n->next = n->prev;
   n->prev = NULL;
    // n is the new head.
    return n;
}
```

Doubly Circular Linked List

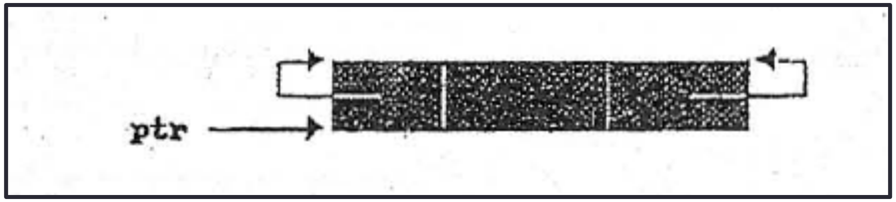
```
typedef struct node *node_pointer;
typedef struct node{
    node_pointer leftlink;
    element item;
    node_pointer rightlink;};
```

```
ptr = ptr->leftlink->rightlink = ptr->rightlink->leftlink
```

Doubly linked circular linked list with head node:

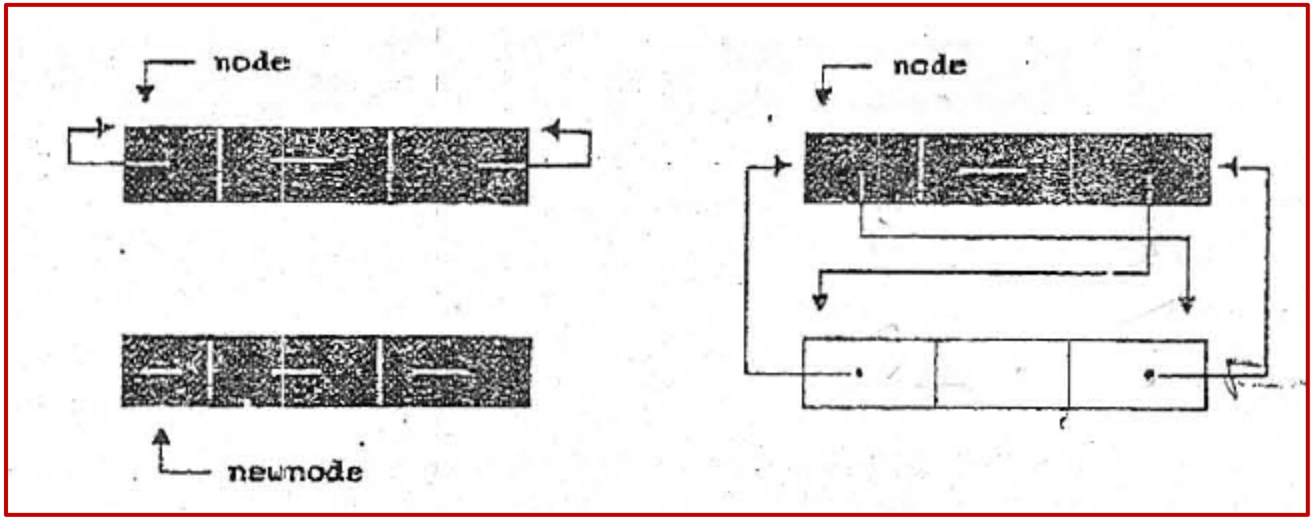


Empty doubly linked circular linked list with head node:



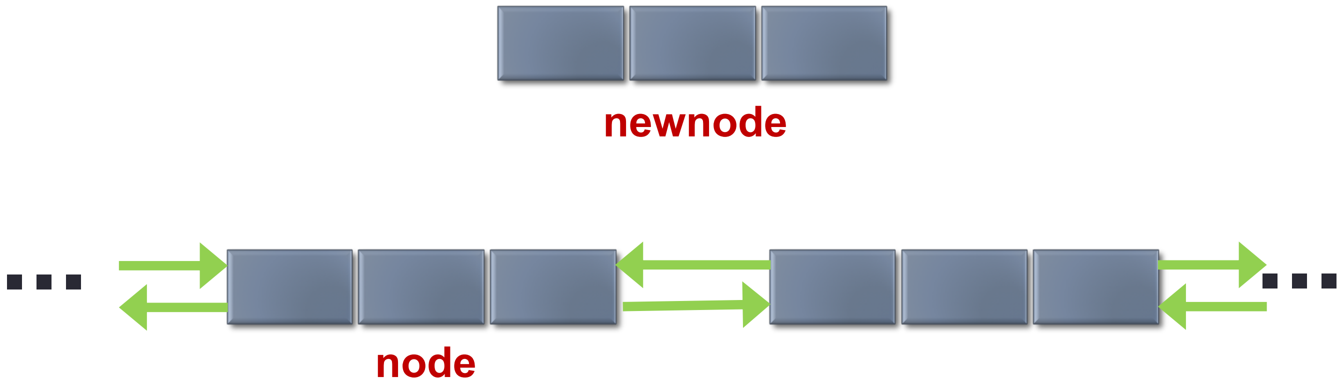
```
Inserting into a doubly-linked circular list:
void dinsert(node_pointer node, node_pointer newnode)
{
    /* insert newnode to the right of node */
    newnode->leftlink = node;
    newnode->rightlink = node->rightlink;
    node->rightlink->leftlink = newnode;
    node->rightlink = newnode;}
```

Insertion into an empty doubly linked circular linked list:

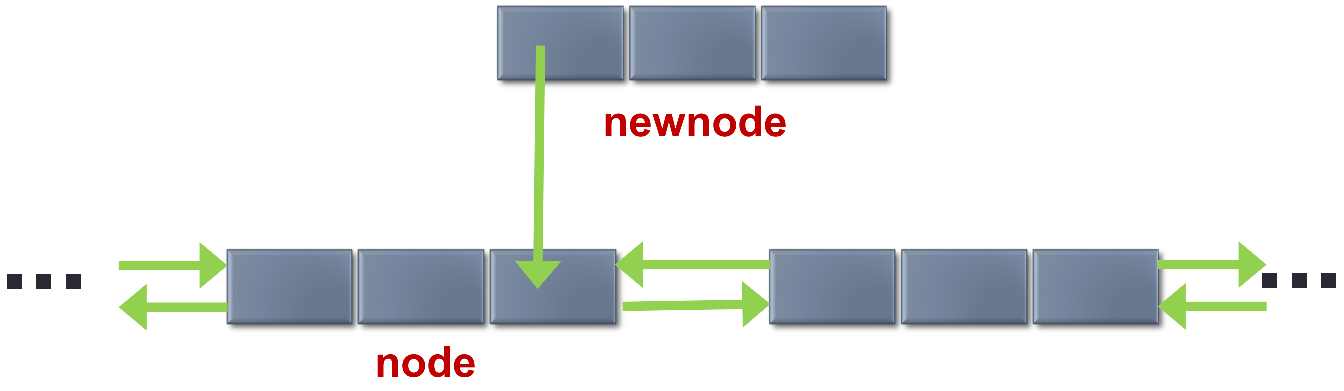


Inserting into a doubly-linked circular list:

```
void dinsert(node_pointer node, node_pointer newnode)
{
    /* insert newnode to the right of node */
    newnode->leftlink = node;
    newnode->rightlink = node->rightlink;
    node->rightlink->leftlink = newnode;
    node->rightlink = newnode;}
```

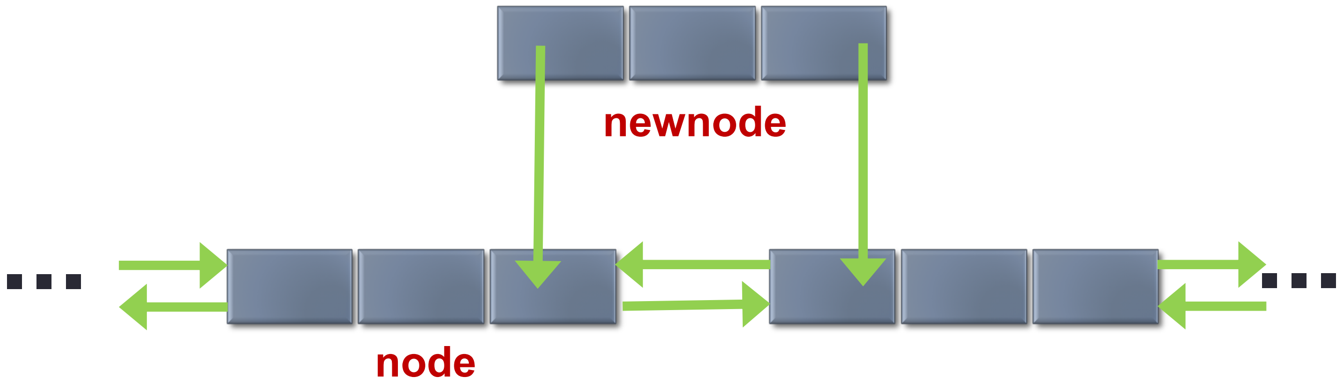


```
Inserting into a doubly-linked circular list:
void dinsert(node_pointer node, node_pointer newnode)
{
    /* insert newnode to the right of node */
    newnode->leftlink = node;
    newnode->rightlink = node->rightlink;
    node->rightlink->leftlink = newnode;
    node->rightlink = newnode;}
```



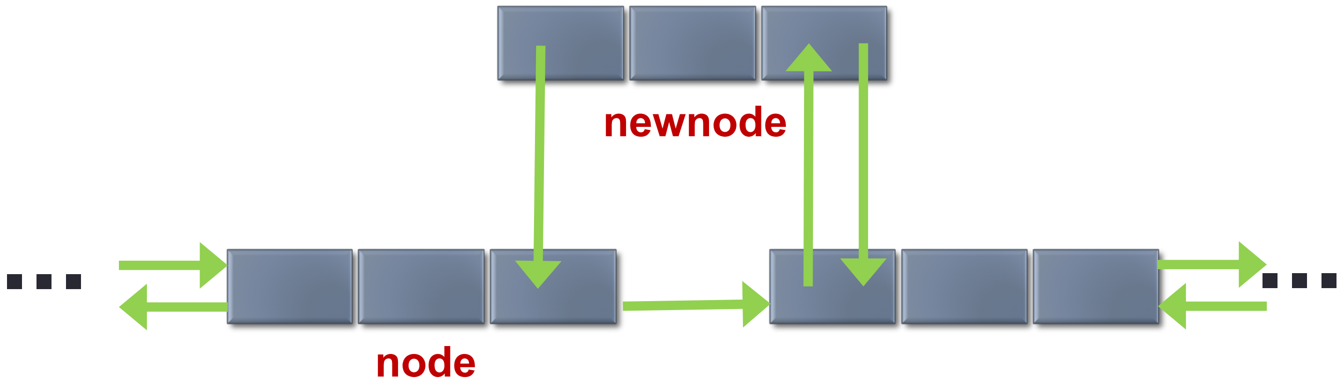
Inserting into a doubly-linked circular list:

```
void dinsert(node_pointer node, node_pointer newnode)
{
    /* insert newnode to the right of node */
    newnode->leftlink = node;
    newnode->rightlink = node->rightlink;
    node->rightlink->leftlink = newnode;
    node->rightlink = newnode;}
```



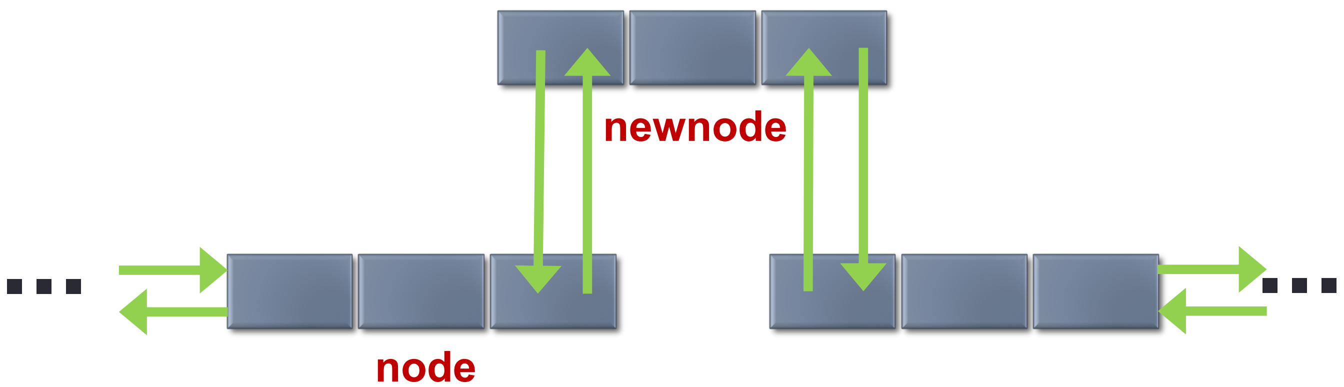
Inserting into a doubly-linked circular list:

```
void dinsert(node_pointer node, node_pointer newnode)
{
    /* insert newnode to the right of node */
    newnode->leftlink = node;
    newnode->rightlink = node->rightlink;
    node->rightlink->leftlink = newnode;
    node->rightlink = newnode;}
```



Inserting into a doubly-linked circular list:

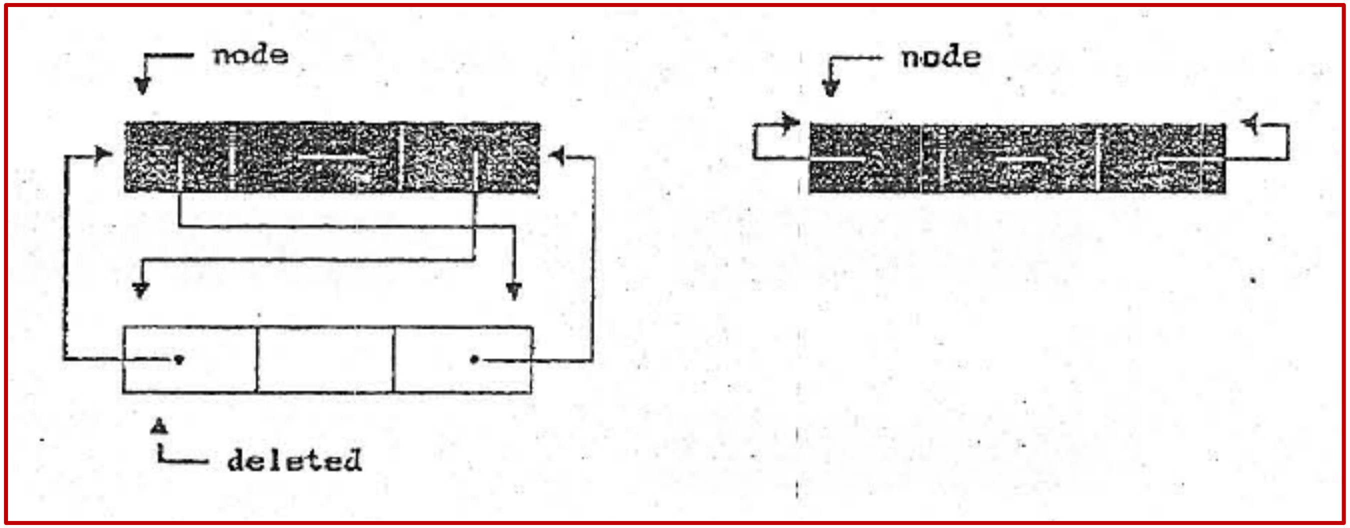
```
void dinsert(node_pointer node, node_pointer newnode)
{
    /* insert newnode to the right of node */
    newnode->leftlink = node;
    newnode->rightlink = node->rightlink;
    node->rightlink->leftlink = newnode;
    node->rightlink = newnode;}
```



Deletion from a doubly-linked circular list:

```
void ddelete(node_pointer node, node_pointer deleted)
{
    // node points to the head node of the list
    if(node == deleted)
        printf("Deletion of head node not permitted.\n");
    else
        deleted->leftlink->rightlink = deleted->rightlink;
        deleted->rightlink->leftlink = deleted->leftlink;
        free(deleted);}
}
```

Deletion from a doubly linked circular linked list:

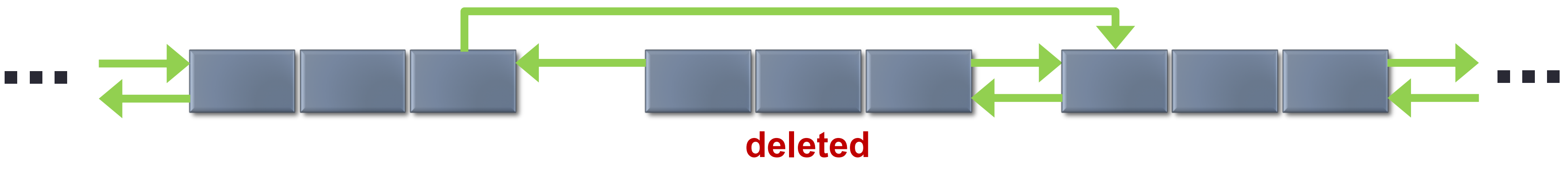


Deletion from a doubly-linked circular list:

```
void ddelete(node_pointer node, node_pointer deleted)
{
    // node points to the head node of the list
    if(node == deleted)
        printf("Deletion of head node not permitted.\n");
    else
        deleted->leftlink->rightlink = deleted->rightlink;
        deleted->rightlink->leftlink = deleted->leftlink;
        free(deleted);}
}
```

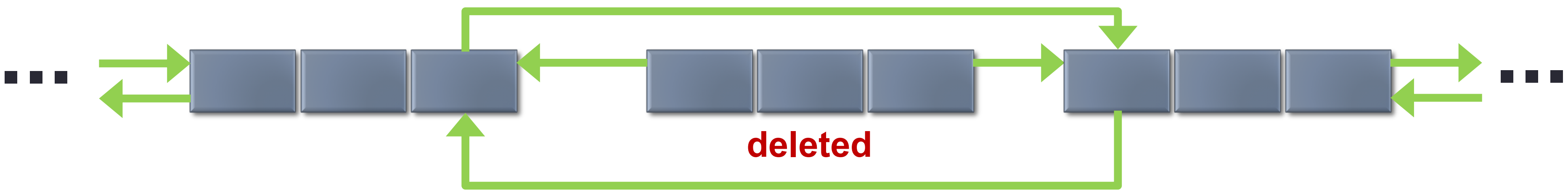
Deletion from a doubly-linked circular list:

```
void ddelete(node_pointer node, node_pointer deleted)
{
    // node points to the head node of the list
    if(node == deleted)
        printf("Deletion of head node not permitted.\n");
    else
        deleted->leftlink->rightlink = deleted->rightlink;
        deleted->rightlink->leftlink = deleted->leftlink;
        free(deleted);}
}
```



Deletion from a doubly-linked circular list:

```
void ddelete(node_pointer node, node_pointer deleted)
{
    // node points to the head node of the list
    if(node == deleted)
        printf("Deletion of head node not permitted.\n");
    else
        deleted->leftlink->rightlink = deleted->rightlink;
        deleted->rightlink->leftlink = deleted->leftlink;
        free(deleted);}
}
```



Deletion from a doubly-linked circular list:

```
void ddelete(node_pointer node, node_pointer deleted)
{
    // node points to the head node of the list
    if(node == deleted)
        printf("Deletion of head node not permitted.\n");
    else
        deleted->leftlink->rightlink = deleted->rightlink;
        deleted->rightlink->leftlink = deleted->leftlink;
        free(deleted);}
}
```