Chapter 5

Lists

Lists

- A list has sequence of elements with a fixed order.
- We can add, remove, inspect, and update elements anywhere in a list. Hence, lists are extremely versatile abstract data types (ADTs). Lists are more general than stacks or queues.
- The **length** of the list is the number of elements it contains. The empty list has length zero.
- We can implement lists using both arrays and linked lists.

Linked Lists

- Using sequential storage (such as array) to represent stacks and queues has following drawbacks:
  - Fixed amount of storage remains allocated to the stack or queue.
  - No more than fixed amount of storage may be allocated, thus introducing the possibility of overflow.
- A **linked list** is a dynamic data structure in which each item (called a **node**) within itself contains the address of the next item. Such data structure is also called **singly linked list (SLL)**.
- Each node contains two fields: an **information** field and a **next address** field.

Linked Lists

- The information field holds the actual element on the list and the next address field contains the address of the next node in the list.
- The address which is used to access a particular node is known as a **pointer**.
- The entire linked list is accessed from an external pointer called **list** that points to (contains the address of) the first node in the list.
- The next address field of the last node in the list contains a special value known as **null**, which is not a valid address and is used to signal the end of a list.

Linked Lists

- The list with no nodes on it is called the empty list or the null list. The value of the external pointer to such list is the null pointer.

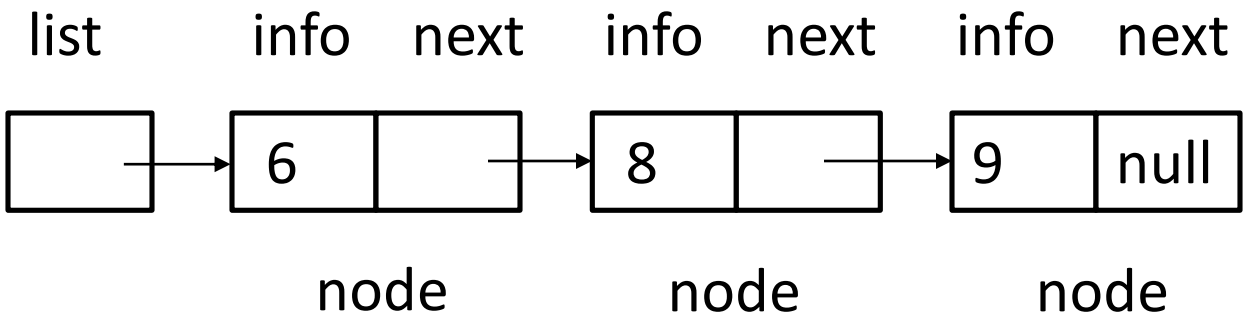


Fig: Singly linked list

Inserting and Removing Nodes in SLL

- A linked list is a dynamic data structure. We can insert and remove any number of nodes in a linked list.
- The dynamic nature of a linked list may be contrasted with the static nature of an array, whose size remains constant.
- To insert a node at a given point in the linked list, we create a new node with info (set to given information) and next (set to null) fields. Then we insert this new node at the given point.
- To delete a node at a given point in a nonempty linked list, we set next field of its predecessor node (if any) to the address of its successor node (if any).

Inserting and Removing Nodes in SLL

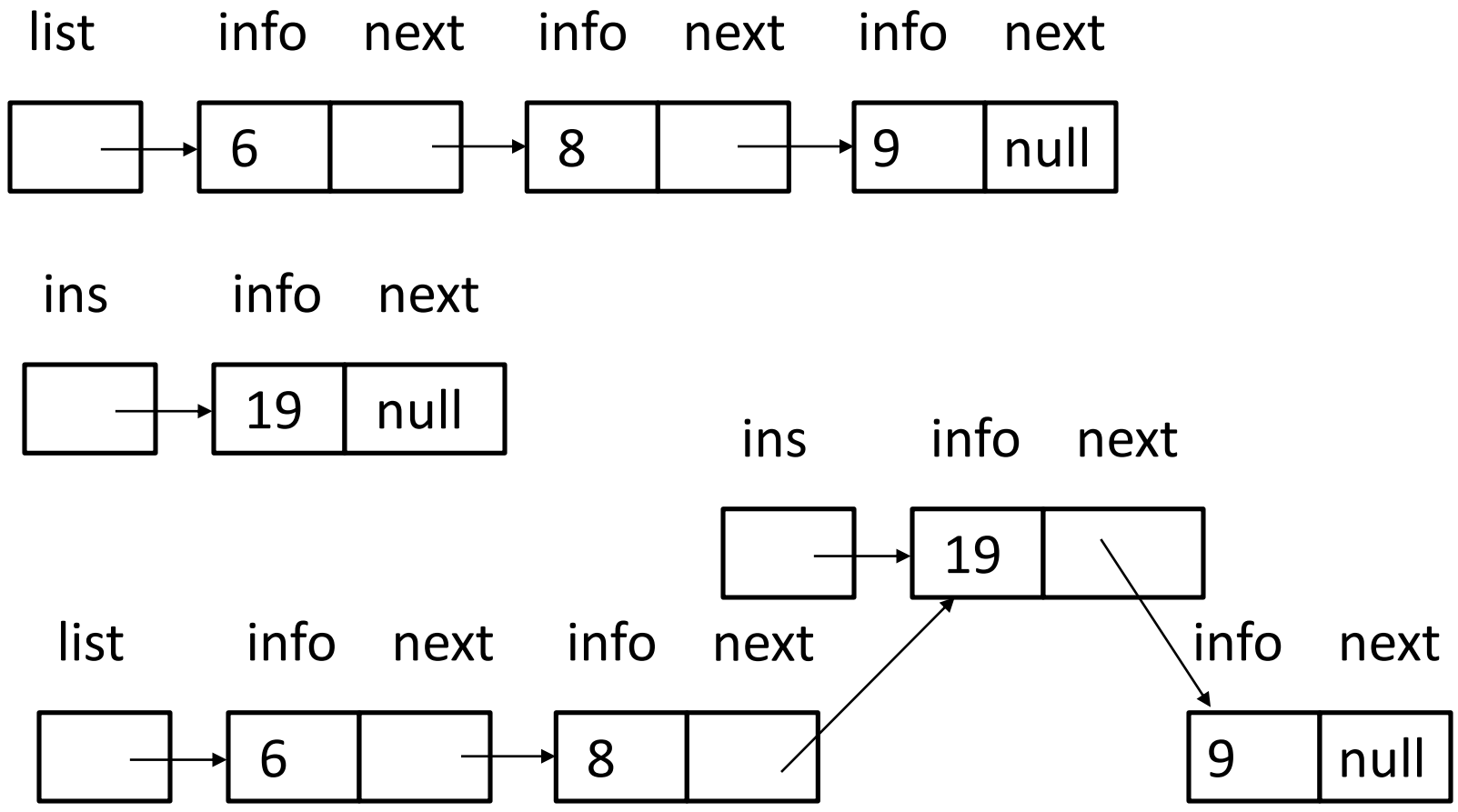


Fig: Inserting a node with information 19 after second node

Inserting and Removing Nodes in SLL

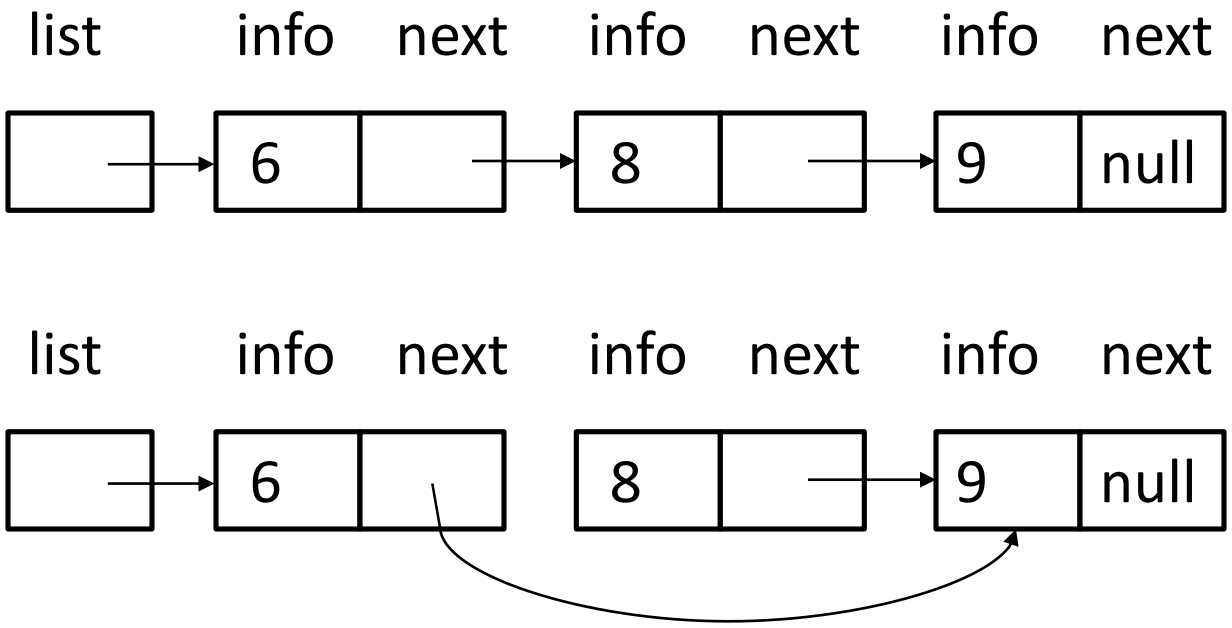


Fig: Deleting second node

C-implementation of SLL

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node* next;
};
struct node* list = NULL;
void insertnode(struct node* pred, int val)
{
        struct node* ins = (struct node*) malloc(sizeof(struct node));
        ins->info = val;
        ins->next = NULL;
        if(pred == NULL)
```

C-implementation of SLL

```
        {
                ins->next = list;
                list = ins;
        }
        else
        {
                ins->next = pred->next;
                pred->next = ins;
        }
}
void deletenode(struct node* del)
{
        if(del == list)
        {
                list  = del->next;
        }
        else
        {
                struct node* pred = list;
                while (pred->next != del)
                {
                        pred = pred->next;
                }
                pred->next   =   del->next;

        }
        free(del);
}
```

C-implementation of SLL

```
void display()
{
        struct node* temp = list;
        while(temp != NULL)
        {
                printf("%d\t", temp->info);
                temp = temp->next;
        }
}
```

```
int main()
{
        insertnode(NULL, 7);
        insertnode(list, 10);
        insertnode(list, 16);
        deletenode(list->next);
        display();
}
```

Doubly Linked Lists

- Singly linked list cannot be traversed backward
- In case where we need backward traversal, the appropriate data structure is a **doubly linked list (DLL)**
- Each node in a **doubly linked list** contains two pointers, one to its predecessor and another to its successor
- So, a node in a doubly linked list consists of three fields: an **information** field that contains the information stored in the node, and **left** and **right** fields that contain pointers to the nodes on either side

Doubly Linked Lists

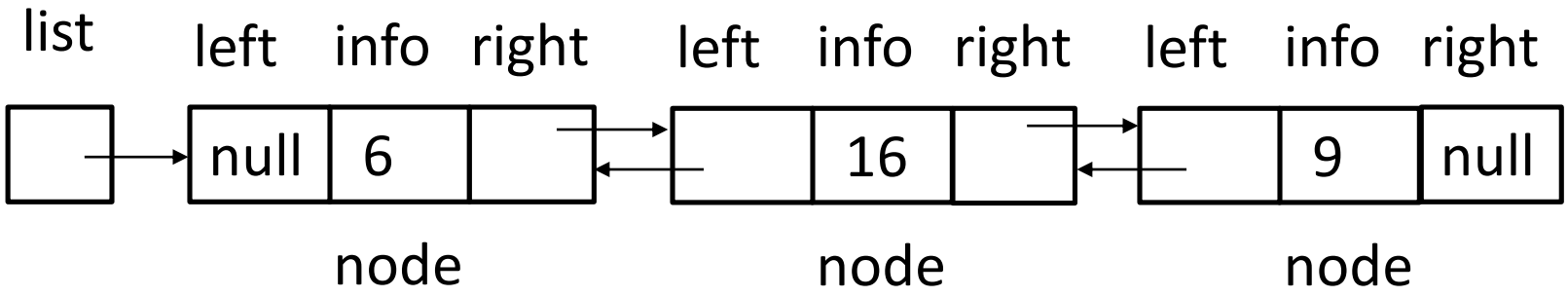


Fig: Doubly linked list

Inserting and Removing Nodes in DLL

- To insert a node at a given point in the linked list, we create a new node with info (set to given information), left (set to null) and right (set to null) fields; Then we insert this new node at the given point by adjusting links in both direction
- To delete a node at a given point in a nonempty linked list, we set right field of its predecessor node (if any) to the address of its successor node (if any); We also set left field of its successor node (if any) to the address of its predecessor node (if any)

Inserting and Removing Nodes in SLL

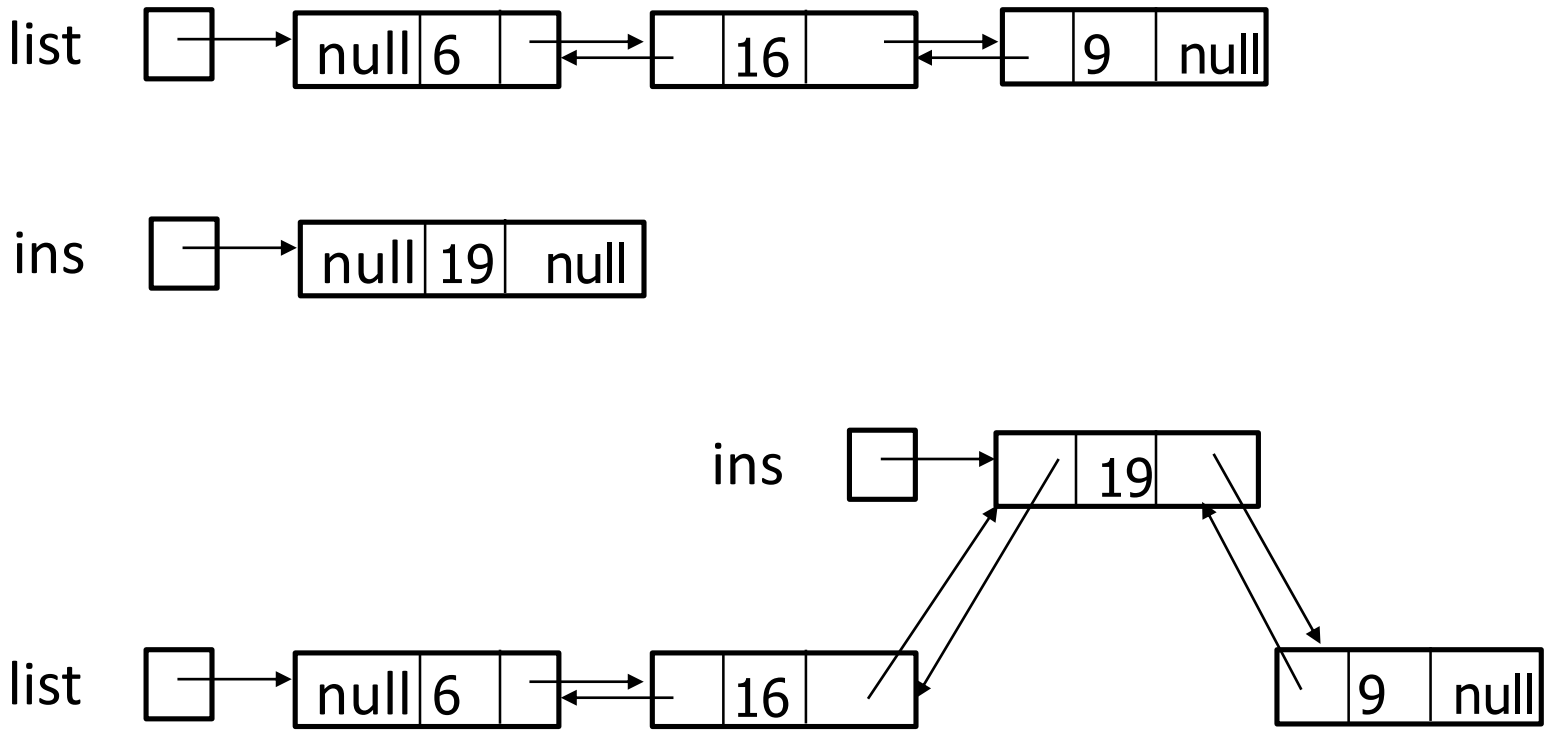


Fig: Inserting a node with information 19 after second node

Inserting and Removing Nodes in SLL

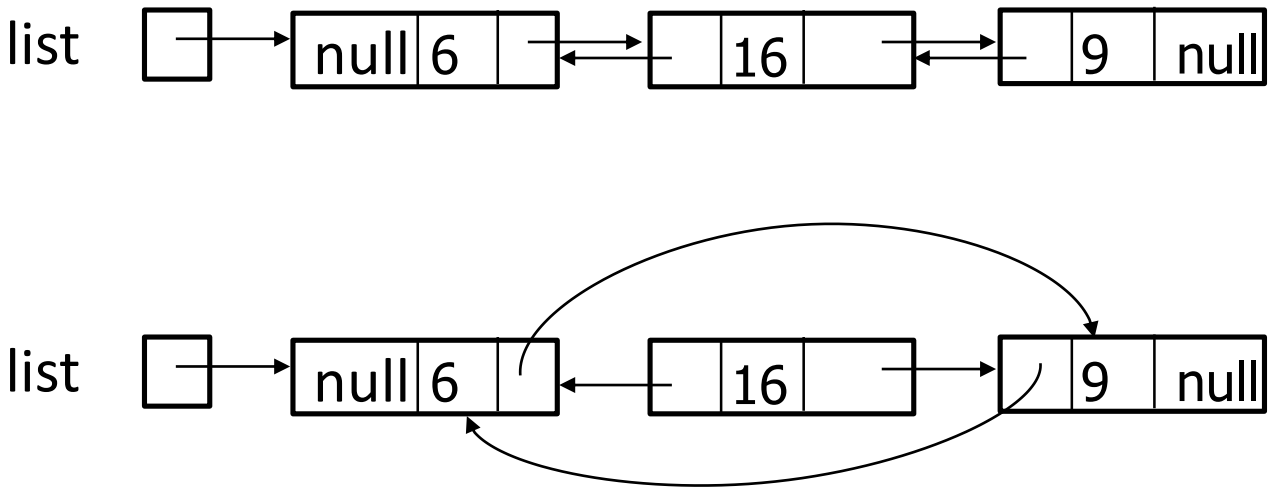


Fig: Deleting second node

C-implementation of DLL

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *left, *right;
};
struct node* list = NULL;
void insertnode(struct node* pred, int val)
{
        struct node* ins = (struct node*) malloc(sizeof(struct node));
        ins->info = val;
        ins->left = NULL;
        ins->right = NULL;
```

C-implementation of DLL

```
if(pred == NULL)
{
        ins->right = list;
        list = ins;
        struct node *succ = ins->right;
        if(succ != NULL)
                succ->left = ins;
}
else
{
   ins->right = pred->right;
   pred->right = ins;
   struct node *succ = ins->right;
```

C-implementation of DLL

```
            if(succ != NULL)
                        succ->left = ins;
            ins->left = pred;
        }
}
void deletenode(struct node* del)
{
        if(del == list)
        {
                struct node *succ = del->right;
                list  = succ;
                if(succ != NULL)
                        succ->left = NULL;
        }
```

C-implementation of DLL

```
        else
        {
                struct node *pred = del->left;
                pred->right = del->right;
                struct node *succ = del->right;
                if(succ != NULL)
                        succ->left = pred;
        }
        free(del);
}
void display()
{
        struct node* temp = list;
```

C-implementation of DLL

```
        while(temp != NULL)
        {
                printf("%d\t", temp->info);
                temp = temp->right;
        }
}
int main()
{
        insertnode(NULL, 7);
        insertnode(list, 10);
        insertnode(list, 16);
        deletenode(list->right);
        display();
}
```

Implementing Stack using Linked List

- Stack can be implemented using a singly linked lists instead of an array
- The two operations that we apply on a stack are push (inserting item on the top) and pop (deleting item at the top).
- The **push** operation is similar to adding a new node at the beginning of the linked list.
- The **pop** operation is similar to removing the first node from a linked list.
- The first node of the linked list is **top** of the stack.

Implementing Stack using Linked List

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *next;
};
struct node *top = NULL;
int isempty()
{
        if(top == NULL)
                return 1;
        else
                return 0;
}
```

```
void push(int val)
{
        struct node* ins = (struct node*) malloc(sizeof(struct node));
        ins->info = val;
        ins->next = NULL;
        ins->next = top;
        top = ins;
}
int peek()
{
        if(isempty())
                printf("Stack is empty.\n");
        else
                return top->info;
}
```

Implementing Stack using Linked List

```
int pop()
{
        if(isempty())
                printf("Stack is
empty.\n");
        else
        {
                struct node *temp = top;
                int data = top->info;
                top = top->next;
                free(temp);
                return data;
        }
}
```

```
int main()
{
        push(1);
        push(12);
        push(15);
        push(20);
        printf("Element at top of the stack:
%d\n", peek());
        printf("Popped element: %d\n",
pop());
        printf("Popped element: %d\n",
pop());
        printf("Element at top of the stack:
%d\n", peek());
}
```

Implementing Queue using Linked List

- Queue can be implemented using a singly linked list with two external pointers **front** and **rear**.
- We make the first node of the list as the **front** and the last node as the **rear**.
- The **front** will point to the beginning of the list while **rear** will point to the last node of the list.
- Initially, when the queue is empty, both **front** and **rear** will be NULL.
- To enqueue a new item, we insert it after the rear node and to dequeue an item, we remove the front node from the linked list.

Implementing Queue using Linked List

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *next;
};
struct node *front = NULL, *rear = NULL;
int isempty()
{
        if(front == NULL)
                return 1;
        else
                return 0;
}
```

```
void enqueue(int val)
{
        struct node* ins = (struct node*) malloc(sizeof(struct node));
        ins->info = val;
        ins->next = NULL;
        if (rear != NULL)
                rear->next = ins;
        else
                front = ins;
        rear = ins;
}
```

Implementing Queue using Linked List

```
int peek()
{
        if(isempty())
                printf("Queue is empty.\n");
        else
                return front->info;
}
```

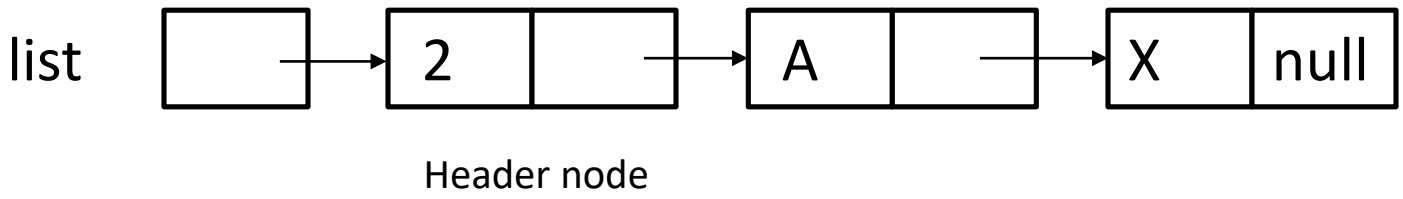
```
int dequeue()
{
        if(isempty())
                printf("Queue is empty.\n");
        else
        {
                struct node *temp = front;
                int data = front->info;
                front = front->next;
                if(front == NULL)
                        rear = NULL;
                free(temp);
                return data;
        }
}
```

Implementing Queue using Linked List

```
int main()
{
        enqueue(1);
        enqueue(12);
        enqueue(15);
        enqueue(20);
        printf("Front element : %d\n", peek());
        printf("Removed element: %d\n", dequeue());
        printf("Removed element: %d\n", dequeue());
        printf(" Front element: %d\n", peek());
}
```

Header Node

- Sometimes it is desirable to keep an extra node at the the front of a list. Such a node does not represent an item in the list and is called a **header node** or a **list header**.
- The *info* portion of such a header node might be unused. More often, the info portion of such node could be used to keep global information (such as, number of nodes except the header in the list) about the entire list

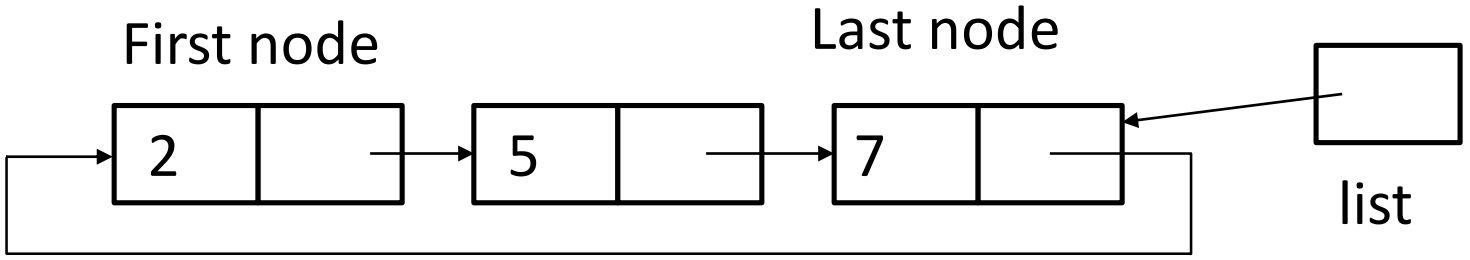


Circular Liked List

- In a singly linked list, we cannot reach any of the nodes that precede a node n. To reach to the preceding node, the external pointer to the list must be preserved.
- In a circular linked list, the next field in the last node contains a pointer back to the first node rather than the null pointer.
- From any point in such a list it is possible to reach any other point in the list. If we begin at a given node and traverse the entire list, we ultimately end up at the starting point.

Circular Linked List

- Note that a circular list does not have a natural “first" or "last" node. We must, therefore, establish a first and last node by convention. One useful convention is to let the external pointer to the circular list point to the last node, and to allow the following node to be the first node.



Doubly Circular Linked List

- In such a linked list, the right link of last node points to the first node while the left link of first node points to the last node.