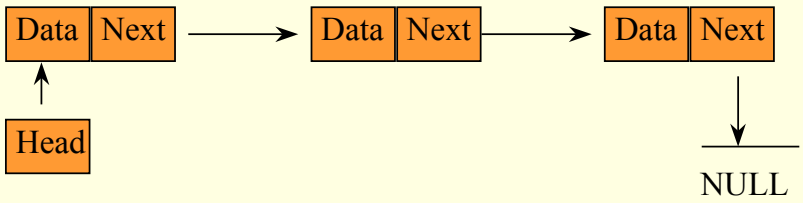
Linked Lists

Liste linkate (1)

- La combinazione di class/struct e puntatori porta alla creazioni di interessanti Abstract Data Type
  - liste linkate (stack, queues), trees
- Liste linkate come strutture

```
struct ListNode {
    int data;
    struct ListNode * ptr_Next;}
struct ListNode ListaNomi;
```

Liste linkate (2)



- due elementi
  - nodi
  - puntatore al primo nodo (Head list) -> se NULL lista vuota
- operazioni possibili
  - Inserimento nuovo nodo (creazione)
  - Cancellazione nuovo nodo
  - Display

Progettazione: `struct`

```
struct ListNode;

struct ListNode{
   int         Item;
   ListNode*   ptr_Next;

};
     CreateList(..)
     DestroyList(..)
     ListIsEmpty(..)
     InsertItem(..)
     RemoveItem(..)
     GetItem(..)
```

- implementazione del C
  - *struct -> insieme di dati*
- dati e funzioni definite separatamente:
  - *occorre passare una variabile di tipo struct ListNode* ad ogni procedura*
  - *NB se occorre cambiare la "testa" della lista occorre passare una struct ListNode ***
- protezione dati
  - *i campi Item e Next sono visibili a tutte le procedure*

Progettazione: `class`

```
struct (class) ListNode;

class List {
 private:
      ListNode* ptr_Testa
 public:
      List(..)costruttore
      ~List(..)distruttore
      ListIsEmpty(..)
      Insert(..)
      Remove(..)
      Get(..)
};
```

- implementazione del C++
  - ***class*** *-> dati + metodi per l'uso+ meccanismo di controllo per l'accesso*
- uso

```
class List LaMiaLista(..);
…
LaMiaLista.Insert(..);
LaMiaLista.Insert(..);
LaMiaLista.Remove(..);
LaMiaLista.Get(..);
```


una classe List più protetta

```
class List {
 private:
   class ListNode {
   public:
      int Item;
   private:
       ListNode* ptr_Next;
   friend class List;
   };
    ListNode* ptr_Testa
 public:
   List(..)  costruttore
   ~List(..) distruttore
   … // altri metodi
          };
```

definizione privata della classe ListNode. Gli attributi sono messi in comune (friend) con la classe List: SOLO la classe List può accedere agli attributi della classe ListNode

puntatore al primo elemento dell'oggetto List:
ptr_Testa = NULL; lista vuota
ptr_Testa = "indirizzo" lista con qualcosa

Implementazione metodi lista linkata

- operazioni su di una lista

```
List(..)              // crea una lista vuota
~List(..)             // distruggi una lista (anche il contenuto)
ListIsEmpty (..)      // determina se la lista è vuota
int Insert (int NewItem)
```

Aggiungi un nuovo elemento (NewItem) alla lista (ALL'INIZIO) e ritorna 0 (bool TRUE) se l'inserimento è andato bene.

```
int Remove( )
```

Rimuovi l'ultimo elemento aggiunto e ritorna 0 (bool TRUE) se la rimozione è andata a buon fine.

```
int Get (int& ptr_ListItem)
```

Recupera l'ultimo elemento aggiunto e lo salva in ListItem, SENZA modificare la lista. Ritorna 0 (bool TRUE) se l'operazione è andata a buon fine.

Some methods..

- **`List()`** deve solo inizializzare a NULL il puntatore alla cima della lista

```
ptr_Testa = NULL;
```

- **`~List()`** deve ripetere n-volte l'operazione **`Remove()`** fino a quando il ritorno dalla funzione non diventi FALSE

```
while (Remove()) ;
```

- **`int ListIsEmpty ()`** controlla se ptr_Testa == NULL

```
if (ptr_Testa == NULL) return (0);
return(1);
```

Insert(NewItem)

La funzione (metodo):

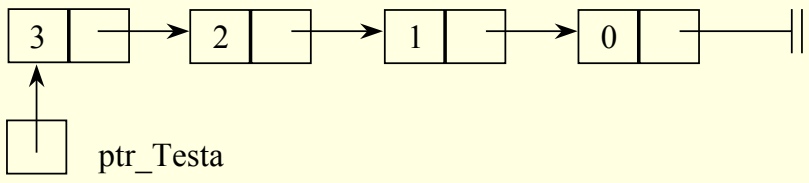
```
int Insert(int NewItem)
```

deve:

- creare un nuovo nodo di Lista e controllare che l'allocazione dinamica sia andata bene
- inserirlo in cima alla lista linkata
- ritornare il successo (TRUE) o meno (FALSE) dell'operazione

Esempio lista linkata

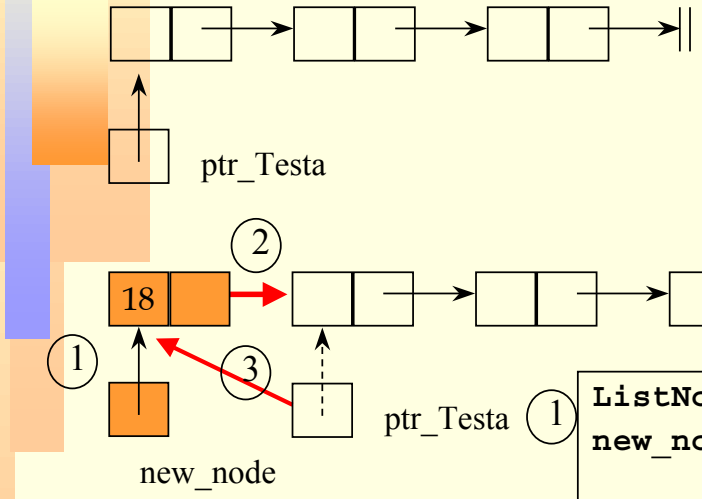
Esempio: lista linkata di interi




```
List LaMiaLista;
…
LaMiaLista.Insert(0);
LaMiaLista.Insert(1);
LaMiaLista.Insert(2);
LaMiaLista.Insert(3);

…
```

int Insert(int NewItem)–in cima



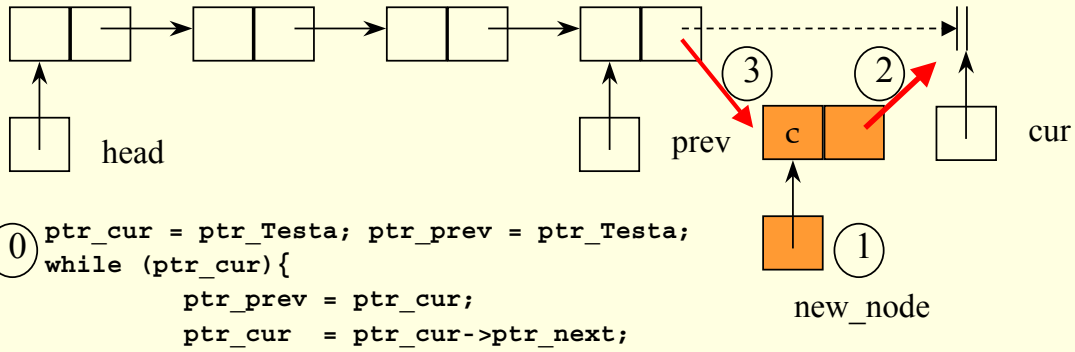

```
1  ListNode *new_node =new ListNode;
   new_node->Item = NewItem;

2  new_node->ptr_Next = ptr_Testa;

3  ptr_Testa = new_node;
```


Insert – non ordinato, in coda

inserimento elemento (non ordinato, in coda) - caso generale




```
0  ptr_cur = ptr_Testa; ptr_prev = ptr_Testa;
   while (ptr_cur){
            ptr_prev = ptr_cur;
            ptr_cur  = ptr_cur->ptr_next;
         }
1  ListNode *new_node = new ListNode;
   new_node->Item = NewItem;

   new_node->ptr_next=ptr_cur; 2

   ptr_prev->ptr_next=new_node; 3
```

Remove e Get

Il metodo **`int Remove ()`** deve:

- controllare che la lista non sia vuota (nel caso ritorna FALSE)
- rimuovere il nodo in cima (basta modificare il ptr_Testa)
- liberare la memoria relativa all'elemento distrutto e ritornare TRUE

Il metodo **`int Get (int& ptr_ListItem)`** deve:

- controllare che la lista non sia vuota (nel caso ritorna FALSE)
- copiare il contenuto del campo Item del nodo in cima su *ptr_ListItem

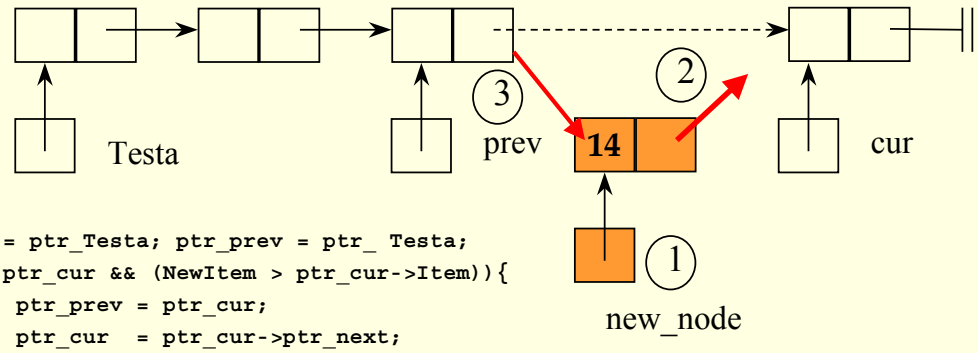
Inserimento Ordinato

- Insert (Data)
  Inserimento (*ORDINATO*) in una lista; per trovare la posizione corretta usare due puntatori (oltre a **ptr_Testa**):
  (1) **ptr_Cur**: all'elemento in uso e
  (2) **ptr_Prev** al precedente.

  Considerare i casi particolari:
  - inserimento in una lista vuota
  - inserimento in cima alla lista

Inserimento ordinato (1)

inserimento ordinato elemento - caso generale



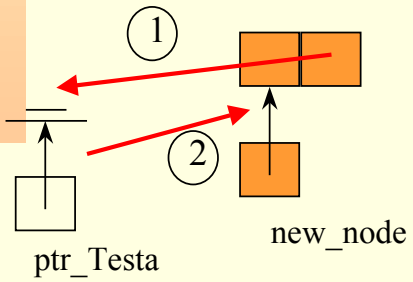

```
0  ptr_cur = ptr_Testa; ptr_prev = ptr_ Testa;
   while ( ptr_cur && (NewItem > ptr_cur->Item)){
            ptr_prev = ptr_cur;
            ptr_cur  = ptr_cur->ptr_next;
          }
1  ListNode *new_node = new ListNode;
   new_node->Item = NewItem;

   new_node->ptr_next=ptr_cur;  2

3  ptr_prev->ptr_next=new_node;
```


Inserimento ordinato (2)

caso particolare : `ptr_Testa = NULL` oppure
`NewItem <= ptr_Testa ->Item`



ATTENZIONE:
Viene modificato il puntatore al primo elemento

```
ptr_cur = ptr_Testa;
if ( (ptr_cur==NULL) ||
          (NewItem <= ptr_cur->Item)){
   ListNode *new_node = new ListNode;
   new_node->Item = NewItem;
   new_node->ptr_next=ptr_cur;  1
   ptr_head = new_node;         2
   return
   }
```

Liste Linkate vs Arrays

- pro Liste Linkate
  - non hanno una grandezza definita all'inizio
  - allocazione migliore della memoria
  - inserimento e cancellazione non necessitano di spostare/copiare i dati
- pro Arrays
  - ordine implicito A[i] < A[i+1]; necessita di minore memoria
  - accesso diretto all'elemento i-esimo