leggibilità attraverso precondizioni e postcondizioni

```
struct nodo {
 int elem;
 struct nodo * next;
};
typedef struct nodo  Lista;
typedef  Lista* ListaPtr;
```

```
 void insert (ListaPtr *L, int val)
{ListaPtr new;
if (*L == NULL || (*L)->elem >= val)
    {new = malloc(sizeof(Lista));
     new->elem = val;
     new->next = *L;
     *L = new;
     }
   else insert (&(*L)->next, val);
}
```

leggibilità attraverso precondizioni e postcondizioni

```
 void insert (ListaPtr *L, int val)
{/*prec: La lista deve essere ordinata in ordine crescente*/
 /*postc: inserisce val nella prima posizione consistente con l'ordine */
Lista new;
if (*L == NULL || (*L)->elem >= val)
    {new = malloc(sizeof(Lista));
     new->elem = val;
     new->next = *L;
     *L = new;
     }
   else insert (&(*L)->next, val);
}
```

OBIETTIVO LEGGIBILITA': precondizioni e postcondizioni, ma anche NOMI AUTOESPLICATIVI.

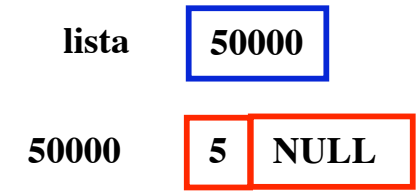
```
 void insListOrd (ListaPtr *L, int val)
{/*prec: La lista deve essere ordinata in ordine crescente*/
 /*postc: inserisce val nella prima posizione consistente con l'ordine */
Lista new;
if (*L == NULL || (*L)->elem >= val)
    {new = malloc(sizeof(Nodo));
     new->elem = val;
     new->next = *L;
     *L = new;
     }
   else insListOrd(&(*L)->next, val);
}
```

Un programma si scrive una volta, ma si legge e si usa molte volte!

```
 void insTestaListSb (ListaPtr L, int val)
/*postc: lascia la lista invariata, ma vorrebbe inserire val in testa alla lista */
{ListaPtr new;
   {new = malloc(sizeof(Lista));
    new->elem = val;
    new->next = L;
    L = new;
    printf( ” dopo l’inserimento, ma nella chiamata \n");
stampaListaInt(L);}
}
main()
{/* facciamo vedere perchè insTestaListSb è un'implementazione sbagliata */
int es = 5, es1 = 8;
ListaPtr lista;
lista = malloc(sizeof(lista));
lista->elem = es;
lista>next = NULL;
printf( ”prima di un qualsiasi inserimento \n” );
stampaListaInt(lista);
insTestaListSb(lista, es);
printf( ”dopo l’inserimento \n");
stampaListaInt(lista);
return 0;}
```

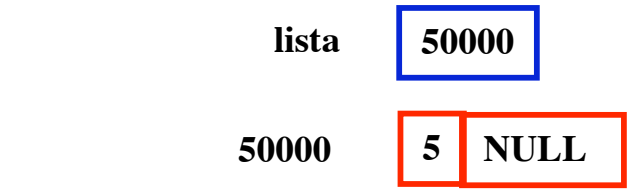
L'effetto delle seguenti istruzioni in memoria è:

```
lista = malloc(sizeof(lista));
lista->elem = es;
lista>next = NULL;
printf(" prima di un qualsiasi inserimento \n");
stampaListaInt(lista);
```



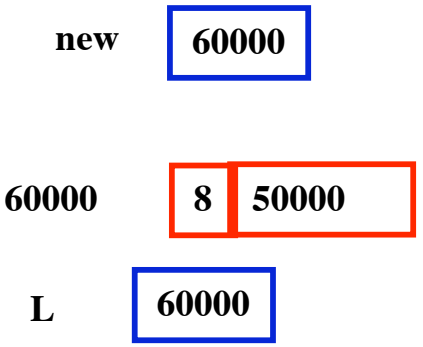
OUTPUT:
prima di un qualsiasi inserimento
la lista é
5 --> NULL

La chiamata
insTestaListsb(lista, es);
produce la copia del valore di lista
nel parametro formale L



E l'esecuzione del seguente codice

```
new = malloc(sizeof(L));
new->elem = val;
new->next = L;
L = new;
```



dopo l'inserimento, ma nella chiamata la lista é
8 --> 5 --> NULL

dopo l'inserimento, fuori chiamata la lista é
5 --> NULL

Perché le modifiche sono state fatte su L e non su lista!

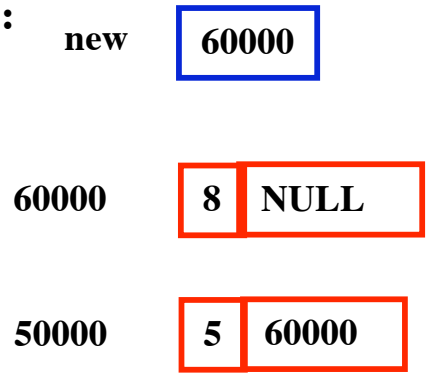
```
void insCancList(ListaPtr L, int val)
/*prec: L !=NULL
postc:  sostituisce gli elementi successivi al secondo di L con  un unico elemento
contenente val.  */
{ListaPtr new;
   {new = malloc(sizeof(Lista));
    new->elem = val;
    new->next = NULL;
    L ->next = new;
    printf( ”dopo l’inserimento, ma nella chiamata \n");
stampaListaInt(L);}
}
main()
{/* facciamo vedere perchè insCancList funziona */
int es = 5, es1 = 8;
ListaPtr lista;
lista = malloc(sizeof(lista));
lista->elem = es;
lista>next = NULL;
printf( ”prima di un qualsiasi inserimento \n” );
stampaListaInt(lista);
insSecList(lista, es);
printf( ” dopo l’inserimento \n");
stampaListaInt(lista);
return 0;}
```

La chiamata
insCanccList(lista, es);
produce la copia del valore di lista
nel parametro formale L

---

l'esecuzione del seguente codice produce le modifiche:

```
new = malloc(sizeof(L));
new->elem = val;
new->next = NULL;
L ->next = new;
```



dopo l'inserimento, ma nella chiamata
la lista é
5 --> 8 --> NULL

dopo l'inserimento, fuori chiamata
la lista é
5 --> 8 --> NULL

Perché le modifiche sono nella memoria!

L'inserimento ricorsivo in una lista ordinata, avviene sempre come se si inserisse in testa a una lista, perché in ogni chiamata il valore del parametro è il puntatore al successivo elemento della lista. Quindi se si scrive:

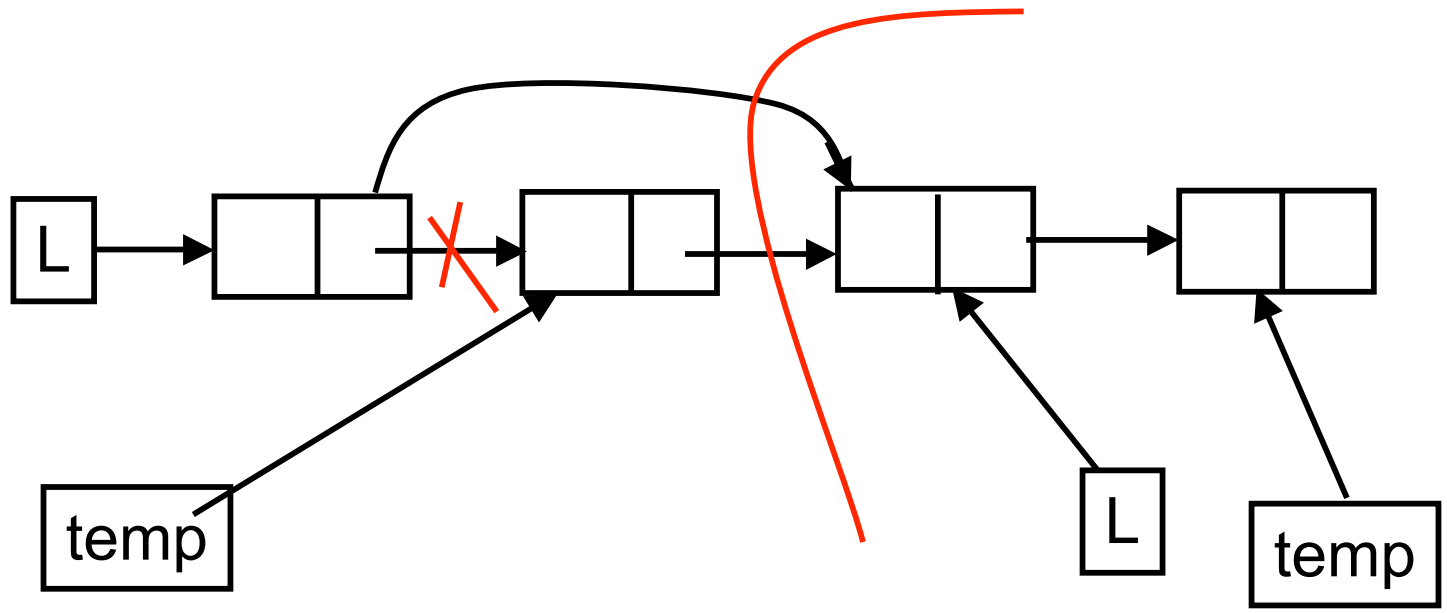
```
void insListOrdSb (ListaPtr L, int val)
{ListaPtr new;
  if (L == NULL || L ->elem >= val)
  {new = malloc(sizeof(Lista));
     new->elem = val;
     new->next = L;
     L = new;
     printf(" dopo l'inserimento, ma nella chiamata \n");
     stampaListaInt(L);}
else insListOrdSb(L->next, val);
}
```

non si ottiene affatto l'effetto voluto, anzi eseguendo la funzione si otterrebbe se la lista inizialmenente é
5 --> 8 --> NULL e val = 9:

```
dopo l'inserimento, ma nella chiamata
la lista  é
9 --> NULL
all'uscita dalla chiamata di insListOrdSb.
la lista  é
5 --> 8 --> NULL
```

Esercizio: data una lista produrne due con circa la metà degli elementi senza duplicare la lista.

Sol1. la lista temp conterra' tutti gli elementi di posto pari, mentre nella lista originaria resteranno quelli di posto dispari.



Soluzione 1:

```
ListaPtr dividi(ListaPtr L)
/* divide una lista in due metà
postc: se L è non vuota  conterrà gli elementi di
posto dispari e la lista restituita quelli di
posto pari*/
{ListaPtr temp;
if (!L || !L-> next) return NULL;
temp = L-> next;
L->next = temp -> next;
temp -> next = dividi(temp -> next);
return temp;
}
```

Soluzione alternativa:

```
ListaPtr  dividi2lt( ListaPtr L)
/* divide la lista L in due metà
postc: se L è non vuota  conterrà la prima metà dei
suoi elementi, viene restituita la lista con i
rimanenti (n elementi se la lista ne contiene 2n o
2n+1) */
{ListaPtr L2 = L;
if (!L || !L->next) return NULL;
while (L2-> next)
       {L2 = L2->next;
       L = L -> next;
       if (L2) L2 = L2 -> next;
       }
L2 = L -> next;
L-> next = NULL;
return L2;}
```