Liste semplici

Obiettivi:

- Discutere la realizzazione collegata (puntatori a strutture) di liste semplici
- Introdurre l'ADT lista semplice e le operazioni tipiche su essa

IL CONCETTO DI LISTA

Una lista semplice è una sequenza (*multi-insieme finito e ordinato*) di elementi dello ***stesso tipo***

Multi-insieme: insieme in cui un medesimo elemento può comparire più volte

Notazione: $\mathtt{L = [el_1, el_2, \ldots, el_N]}$

[‘a’ , ‘b’ , ‘c’] denota la lista dei caratteri ‘a’ , ‘b’ , ‘c’
[5, 8, 5, 21, 8] denota una lista di 5 interi

Come ogni ***ADT***, la lista è definita in termini di:

- ***dominio*** dei suoi elementi (dominio-base)
- operazioni di ***costruzione*** sul tipo lista, operazioni di ***selezione*** sul tipo lista
- ***predicati***

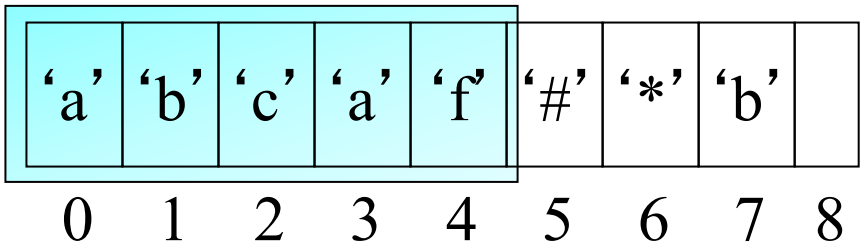
RAPPRESENTAZIONE CONCRETA DI LISTE

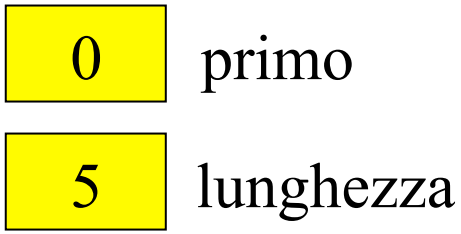
1) RAPPRESENTAZIONE SEQUENZIALE (STATICA)

Utilizzare un ***vettore*** per memorizzare gli elementi della lista uno dopo l'altro (***rappresentazione sequenziale***)

- ***primo*** memorizza l'indice del vettore in cui è inserito primo elemento
- ***lunghezza*** indica da quanti elementi è composta la lista (dimensione logica)

ESEMPIO: ['a' , 'b' , 'c' , 'a' , 'f']





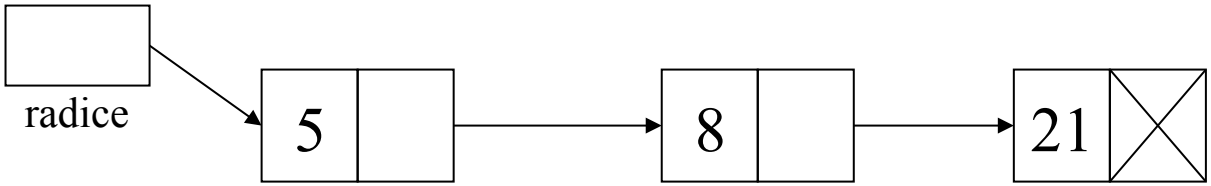
RAPPRESENTAZIONE COLLEGATA

Le liste sono realizzabili anche con array (rappresentazione sequenziale): la successione degli elementi della lista è realizzata dall'adiacenza delle locazioni; non la trattiamo.

Nella **rappresentazione collegata**, a ogni elemento si associa l'indirizzo ***(puntatore)*** che individua la posizione dell'elemento successivo

NOTAZIONE GRAFICA
- elementi della lista come ***nodi***
- riferimenti (indici) come ***archi***

ESEMPIO
[5, 8, 21]

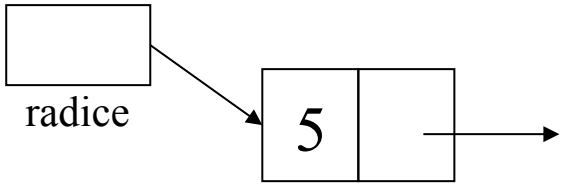


Liste - RAPPRESENTAZIONE COLLEGATA

IMPLEMENTAZIONE MEDIANTE PUNTATORI

Ciascun nodo della lista è una struttura di due campi:

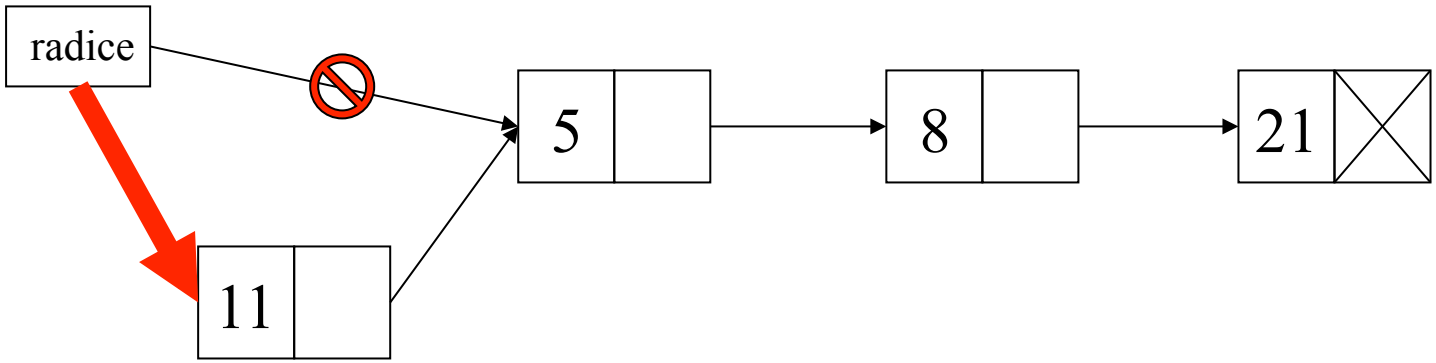
- ***valore*** dell' elemento
- un ***puntatore*** nodo successivo lista (NULL se ultimo elemento)



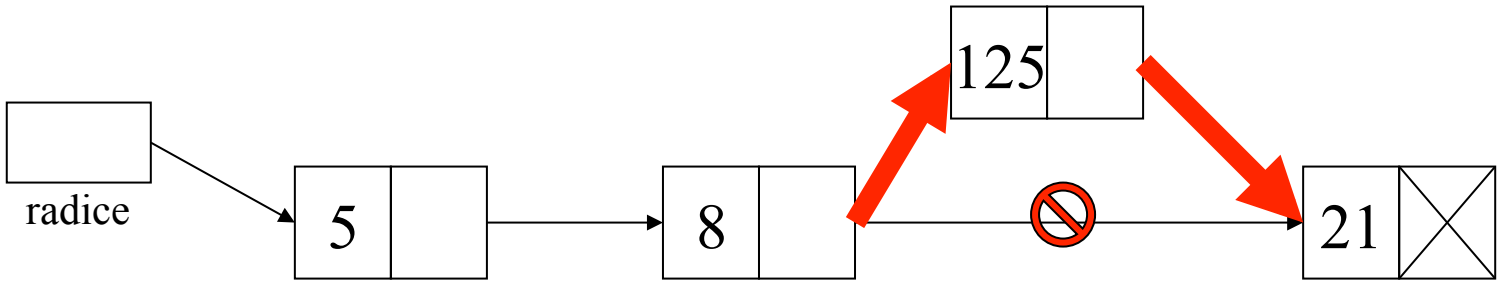

```
typedef struct list_element {
      int value;
      struct list_element *next;} item;
typedef item *list;
list radice;
```

RAPPRESENTAZIONE COLLEGATA

INSERIMENTO (in testa o …)

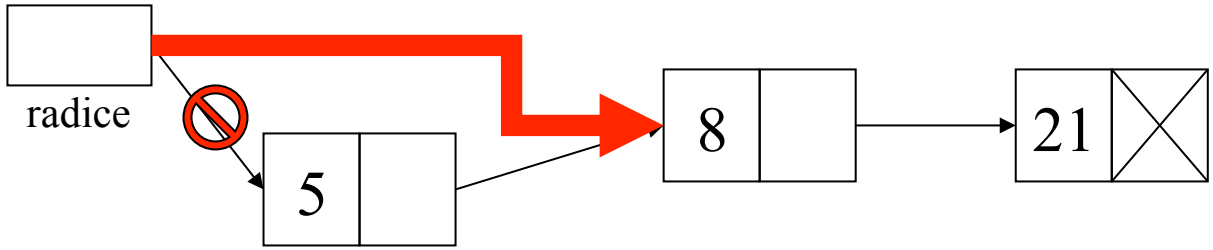


(… in un punto intermedio)

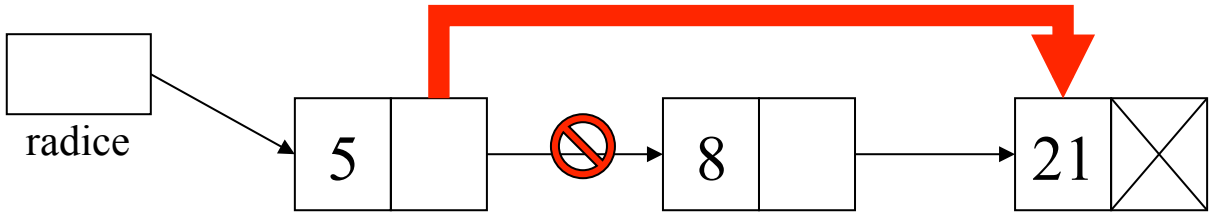


RAPPRESENTAZIONE COLLEGATA

ELIMINAZIONE (dell'elemento in testa o …)



(di un elemento intermedio)



Liste semplici - laboratorio

Obiettivi:

- Sperimentare la realizzazione collegata (puntatori a strutture) per la lista semplice
- Definire funzioni (primitive) per la creazione di liste
- Definire funzioni di elaborazione sequenziale su liste, iterative e ricorsive

Liste - RAPPRESENTAZIONE COLLEGATA

IMPLEMENTAZIONE MEDIANTE PUNTATORI

Ciascun nodo della lista è una struttura di due campi:

- ***valore*** dell' elemento
- un ***puntatore*** nodo successivo lista (NULL se ultimo elemento)

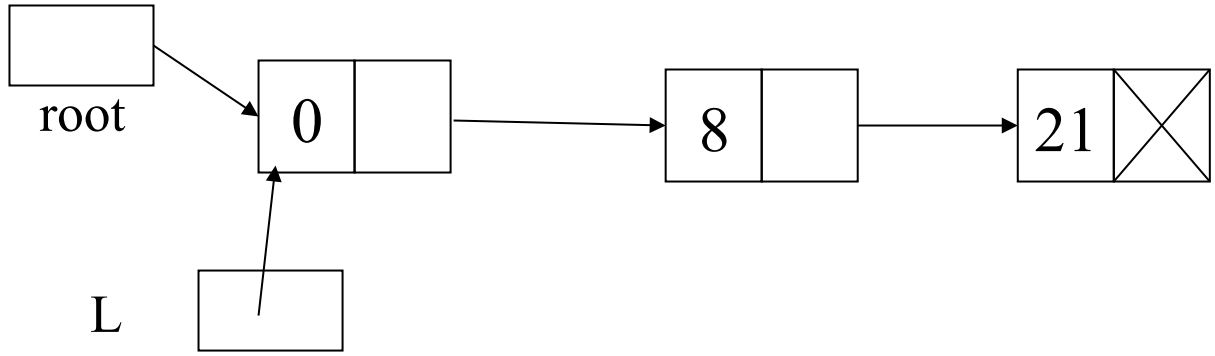
```
typedef struct list_element {
      int value;
      struct list_element *next;} item;
typedef item *list;
```

NOTA: ***etichetta list_element*** in dichiarazione struct è ***indispensabile***, altrimenti sarebbe impossibile definire un tipo ricorsivamente

DO IT!

Il ***main*** da realizzare deve leggere la sequenza di interi e inserire ogni elemento letto in una ***lista***

Per ogni inserimento, **`malloc`** e aggiustamento dei puntatori



root inizialmente NULL

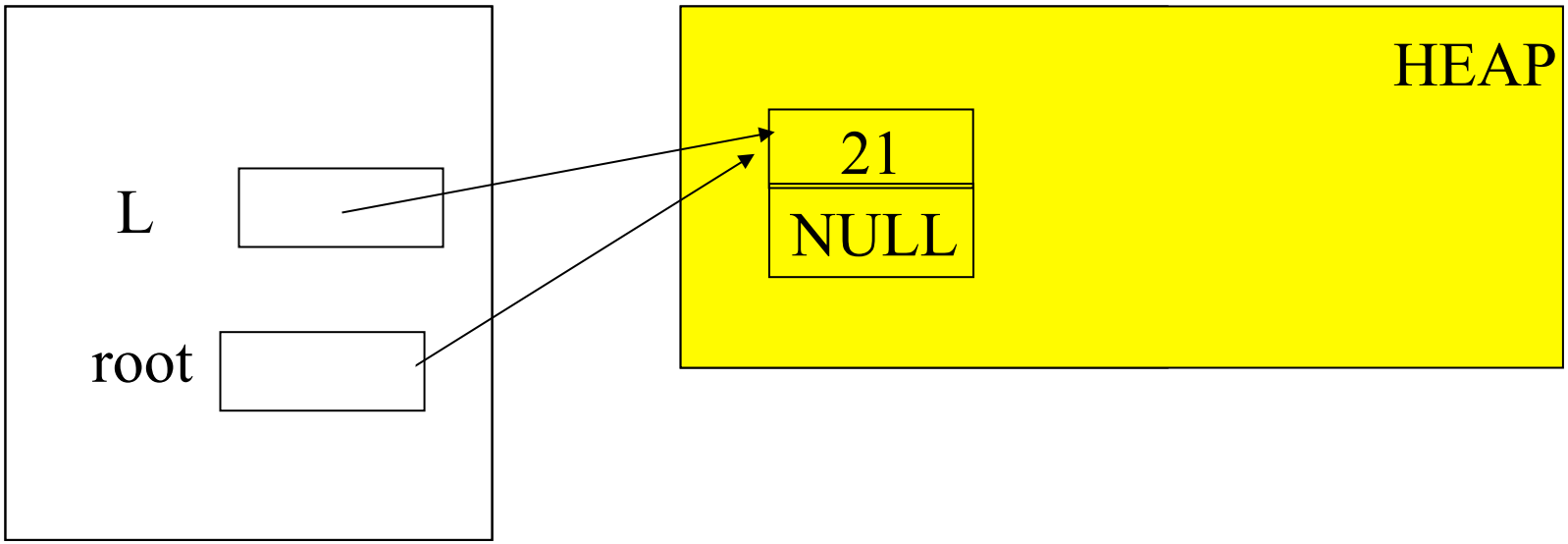
Creazione di una lista semplice

```
#include <stdlib.h>
#include <stdio.h>
typedef struct list_element { int value;
        struct list_element *next; } item;
typedef item *list;

main() { list L; int i;
   list root = NULL;
   do { printf("\nIntrodurre valore: \t");
        scanf("%d", &i);
```

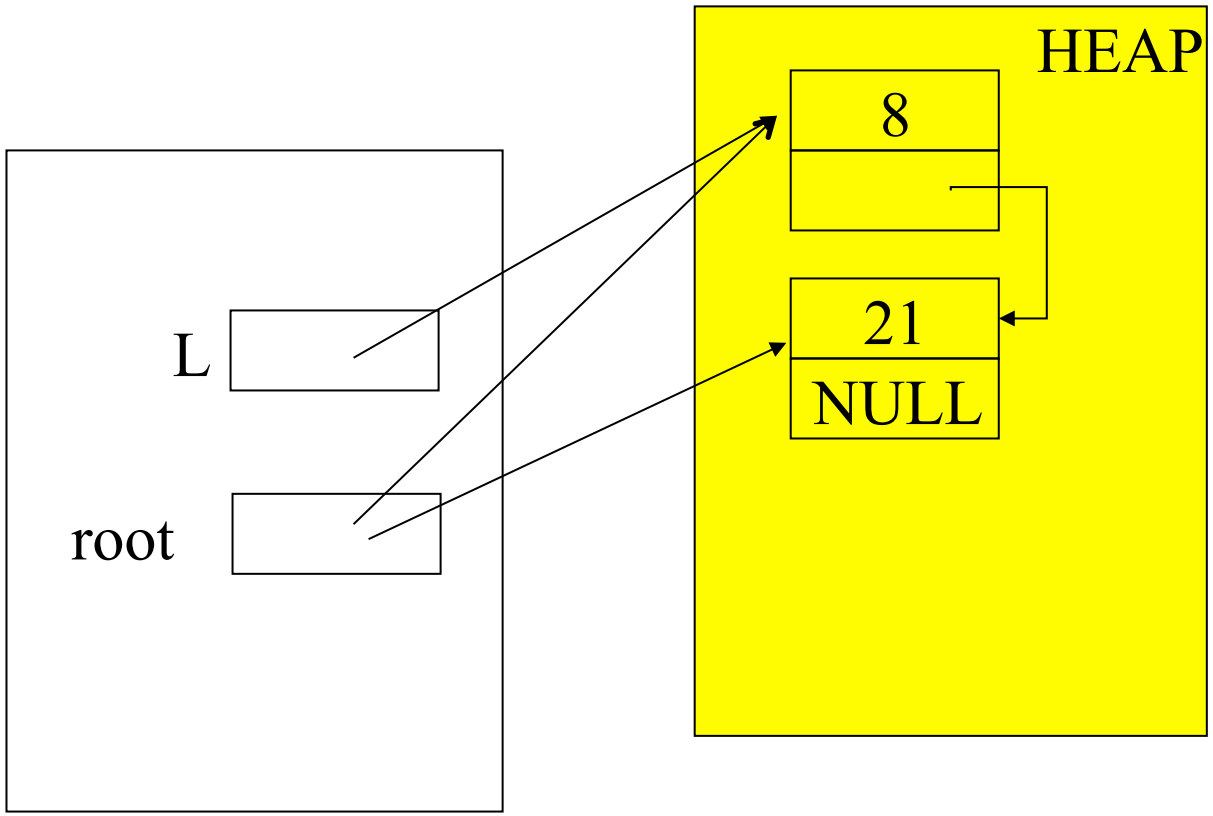
Inserisci **i** nella lista puntata da **root**

```
        } while (i!=0);
}
```

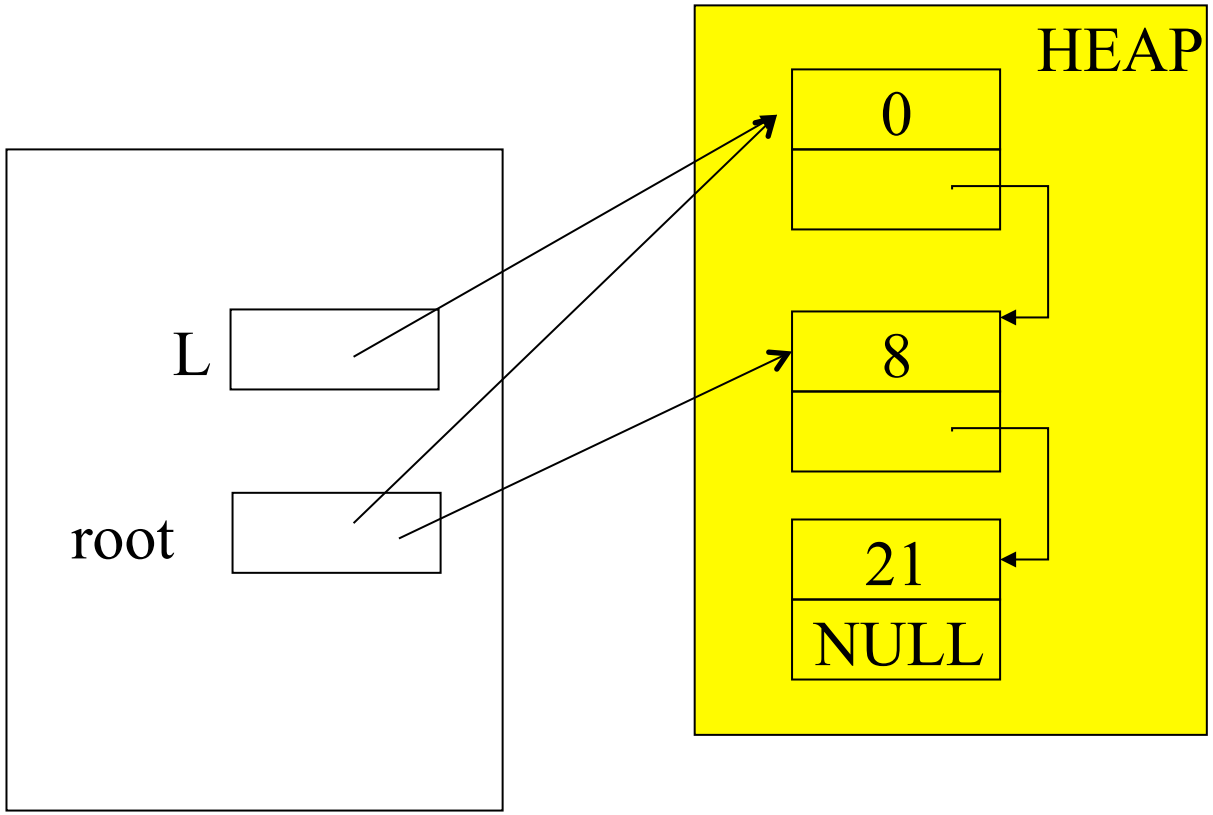
Prima iterazione: leggo 21



Seconda iterazione: leggo 8



Terza iterazione: leggo 0



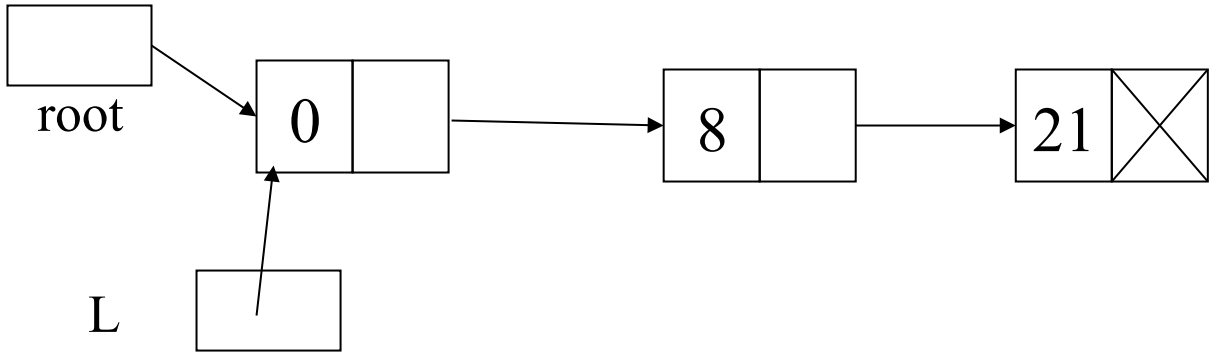
Creazione di una lista semplice

```
#include <stdlib.h>
#include <stdio.h>
typedef struct list_element { int value;
        struct list_element *next; } item;
typedef item *list;

main() { list L; int i;
  list root = NULL;
  do { printf("\nIntrodurre valore: \t");
       scanf("%d", &i);
       L = (list) malloc(sizeof(item));
       L->value = i;
       L->next = root;
       root = L;
      } while (i!=0);
/* stampa della lista */  }
```

DO IT!

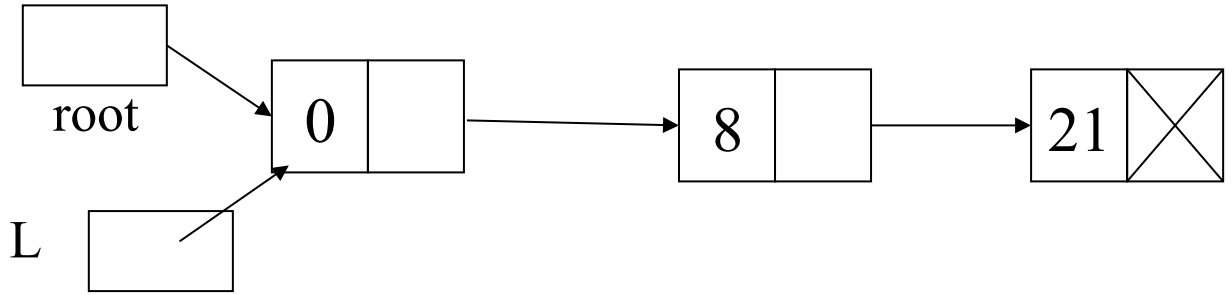
Il ***main*** da realizzare deve leggere la sequenza di interi e inserire ogni elemento letto in una ***lista***

Poi stampiamo la sequenza …
(facciamo insieme anche questo secondo passo)



Non c'è un "indice" da incrementare come per vettori

Stampa di una lista di interi (segue)

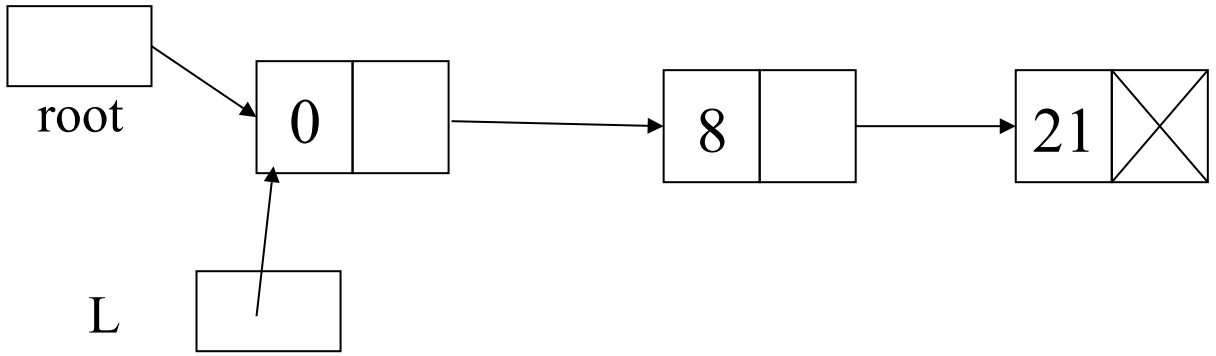



```
i=0;
while (i<N) {
 printf("\nValore : \t%d", V[i]);
 i++; }
}          /* stampa degli elementi di un vettore */
```

```
L=root;
while (L!=NULL) {
 printf("\nValore: \t%d", L->value);
 L = L->next; }
}
```

Stampa di una lista di interi (segue)

```
L = root;

while (L!=NULL) {
 printf("\nValore estratto: \t%d", L->value);
 L = L->next; }
}
```

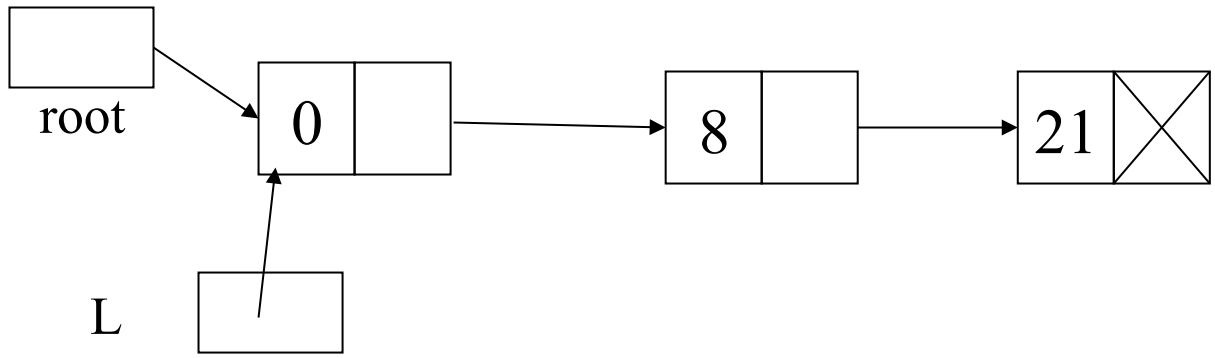


Per "non perdere" il puntatore al primo elemento della lista, la lista va scandita con un puntatore ausiliario (ad esempio L)

FUNZIONI cons E showlist

Prepariamo opportune funzioni di inserimento in testa alla lista e di stampa della lista

```
list cons(int, list);
void showlist (list);
```

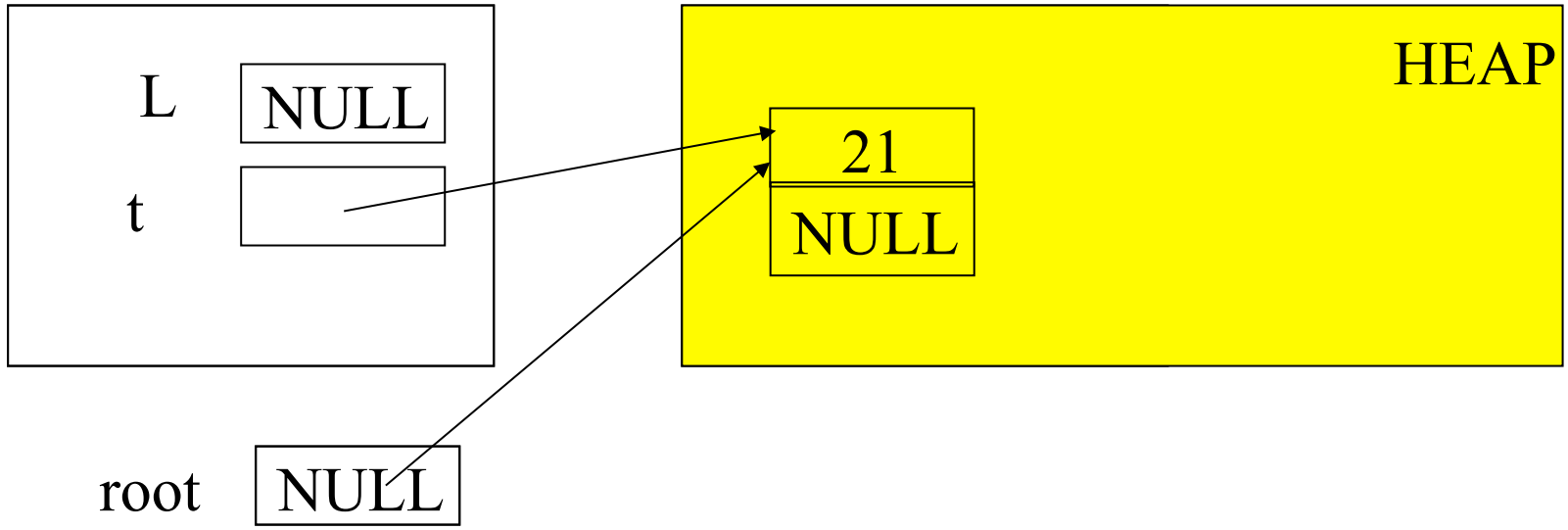


Creazione di una lista semplice

```
#include <stdlib.h>
#include <stdio.h>
typedef struct list_element
    { int value;
      struct list_element *next; } item;
typedef item *list;

main() { list L; int i;
  list root = NULL;
  do { printf("\nIntrodurre valore: \t");
       scanf("%d", &i);

    root = cons( i, root);

       } while (i!=0);
/*stampa lista*/ }
```

Inserimento in testa: *cons*

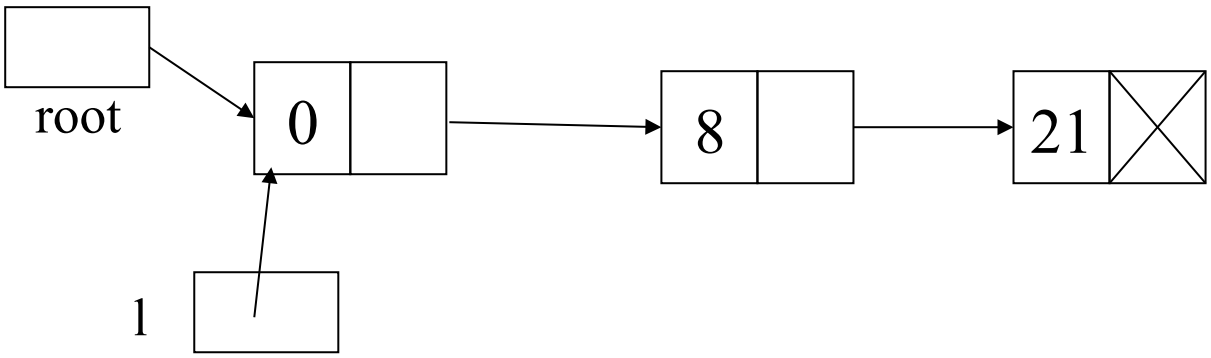
```
list cons(int e, list L) {
     list t;
     t = (list) malloc(sizeof(item));
     t->value = e; t->next = L;
     return t; }
```



Stampa di una lista: `showList`

```
void showList(list l) {
     while (l!=NULL)
          { printf("%d", l->value);
            l=l->next;}
}
```

0 8 21



Iterativa o ricorsiva?

DO IT!

Abbiamo visto queste due funzioni per inserire un elemento i in testa ad una lista L:

```
root = cons( i, root);
```

e per stampare il contenuto di una lista L:

```
showList(root);
```

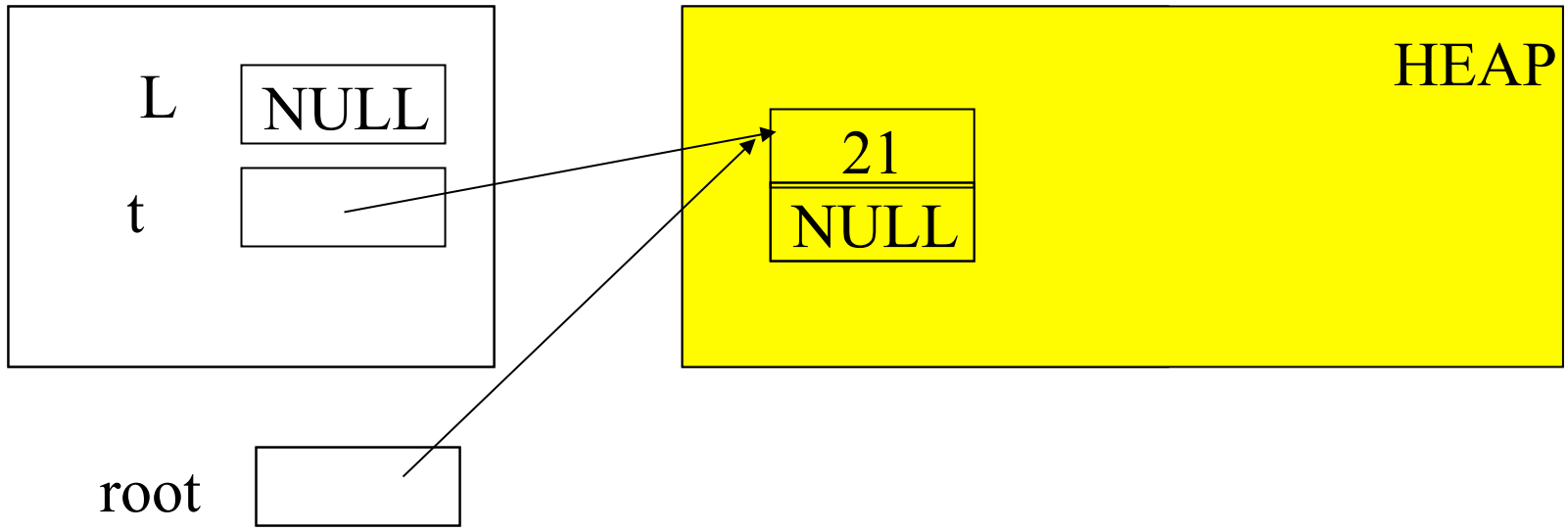
Riscriviamo il ***main*** utilizzando chiamate a ***cons*** e ***showList***

Creazione e stampa di una lista di interi

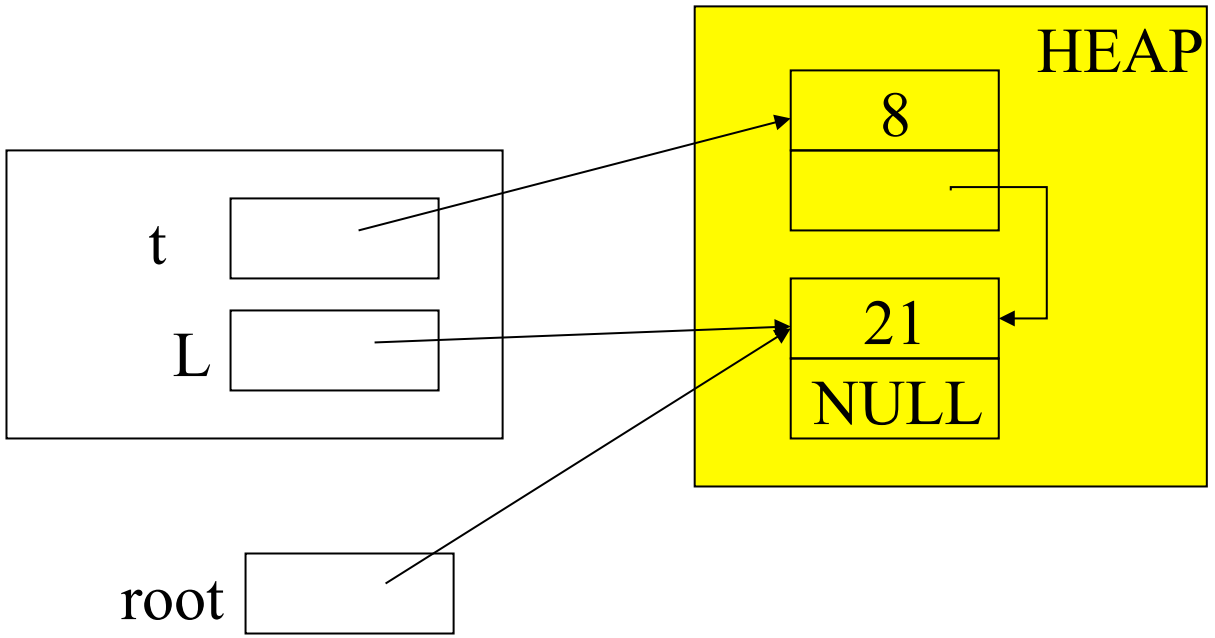
```
#include <stdio.h>
#include <stdlib.h>
typedef struct list_element { int value;
              struct list_element *next; } item;
typedef item *list;
void showList(list l);

list cons(int e, list l) { list t;
       t = (list) malloc(sizeof(item));
       t->value = e; t->next = l;
       return t; }

main() {
list root = NULL; int i;
do {   printf("\nIntrodurre valore: \t");
       scanf("%d", &i);
       root = cons(i, root);
  } while (i!=0);
showList(root); }
```

Prima chiamata: root=cons(i,root)

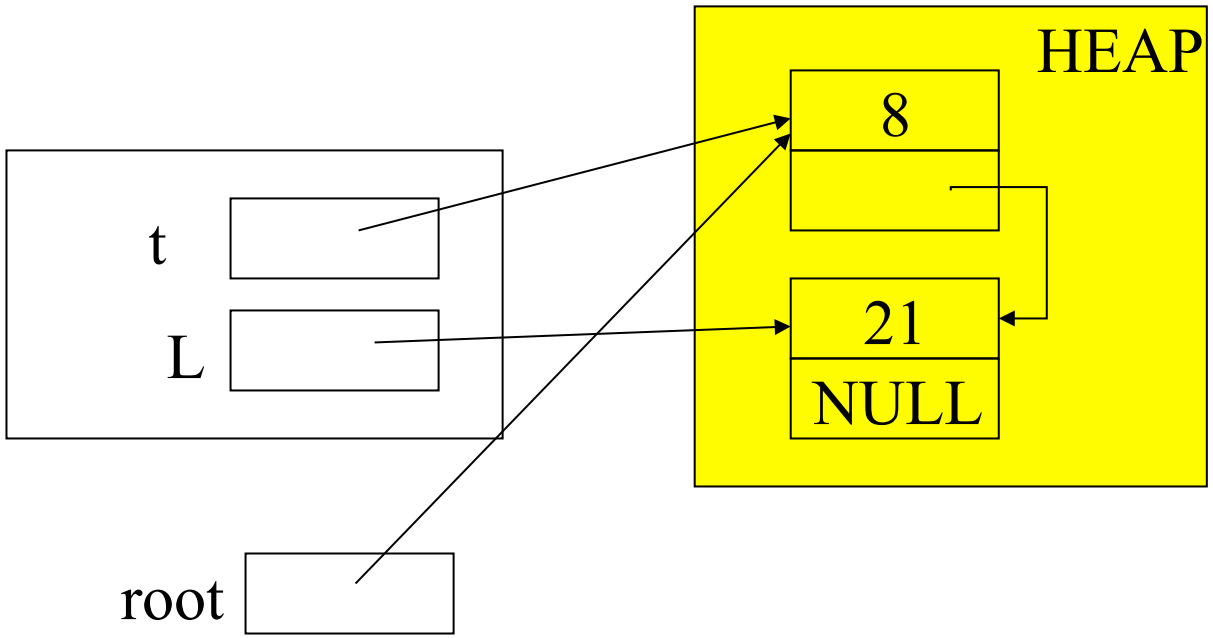
```
list cons(int e, list L) {
        list t;
        t = (list) malloc(sizeof(item));
        t->value = e; t->next = L;
        return t; }
```



Seconda chiamata: root=cons(i,root)



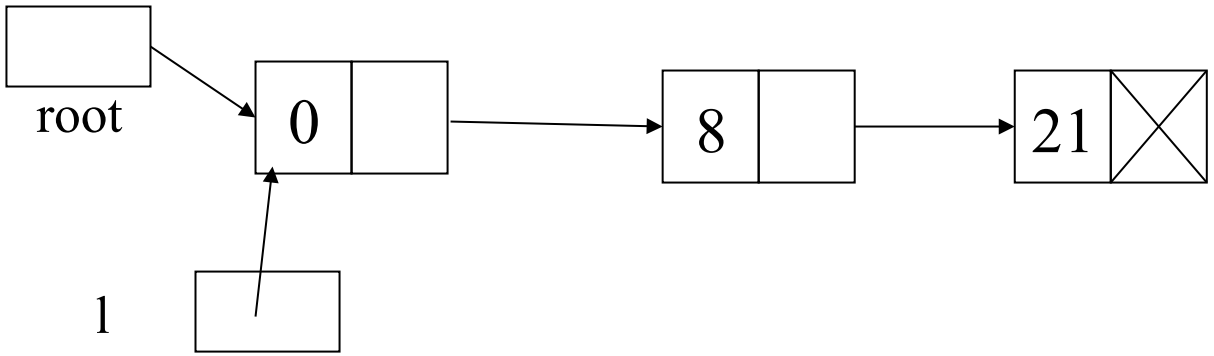
Seconda chiamata: root=cons(i,root)



ESERCIZIO: `showList`

Stampa di una lista (iterativa), la lista va ***scandita (sequenzialmente) con un puntatore:***

```
void showList(list l) {
     while ( l!=NULL )
           {   printf("%d", l->value);
               l=l->next;  }
}
```



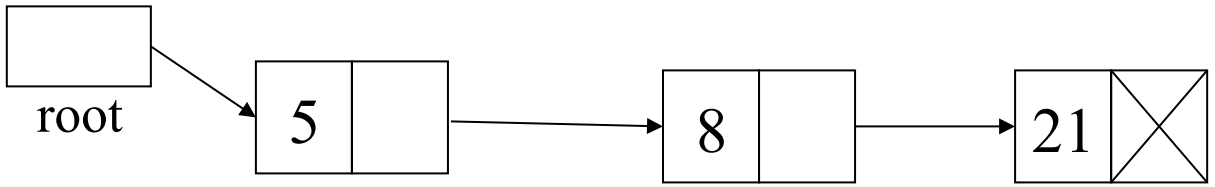
ESERCIZIO: `showListr` ricorsiva

… oppure ***versioni ricorsive***

Stampa di una lista (ricorsiva):

```
void showListr(list l) {
     if( l!=NULL )
          { printf("%d", l->value);
            showListr( l->next ); }
}
```

Operazioni su liste



Tipiche operazioni:

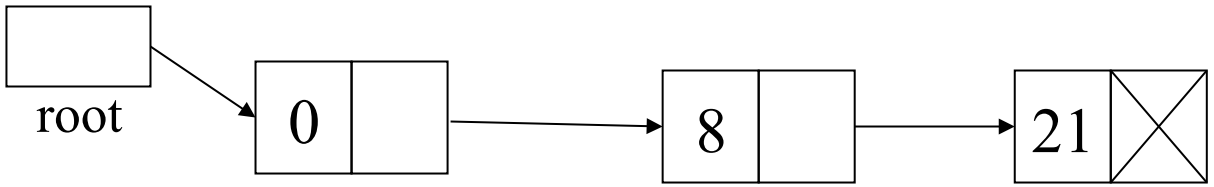
- stampa degli elementi
- ricerca di un elemento
- conteggio degli elementi (lunghezza)
- etc.

Non abbiamo un "indice" da incrementare per scandire la lista, come succedeva per la rappresentazione sequenziale mediante vettori.

La lista va ***scandita (sequenzialmente) con un puntatore,*** *oppure* ***versioni ricorsive***

Ricerca in una lista

```
…
printf(“\nIntrodurre un valore da cercare: \t”);
      scanf(“%d”, &i);
      if (ricerca(i, root)) printf(“Trovato\n”);
       else printf(“Non trovato\n”);

}
```

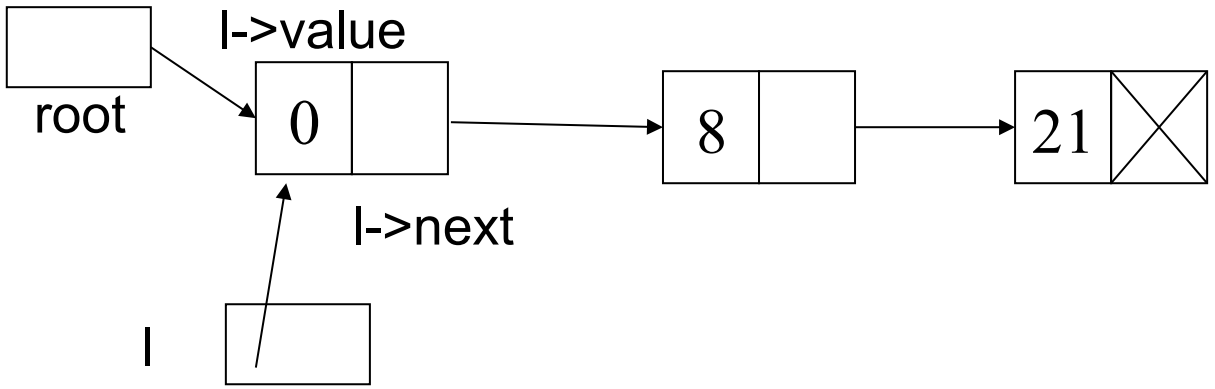


Non abbiamo un “indice” da incrementare per scandire la lista, come succedeva per la rappresentazione sequenziale mediante vettori.

La lista va ***scandita (sequenzialmente) con un puntatore***

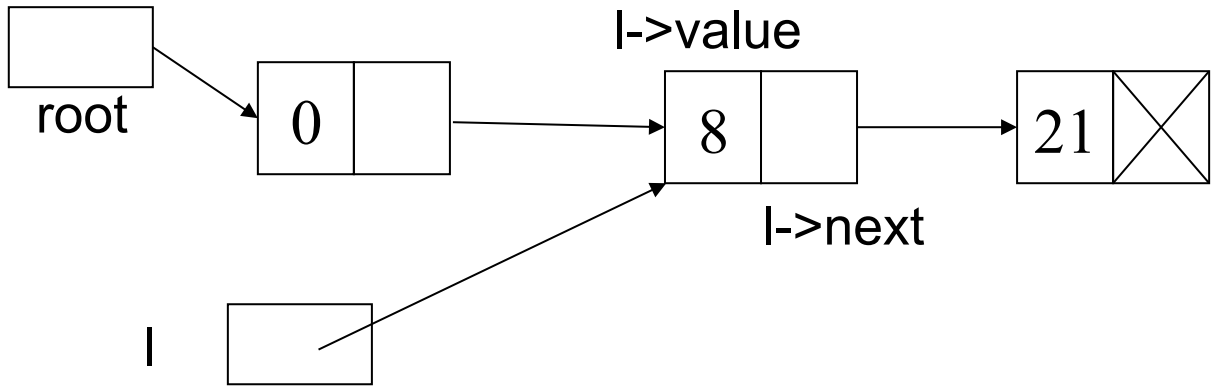
Ricerca in una lista

```
int ricerca(int e, list l) {
      while (l!=NULL)
            if (l->value == e) return 1;
            else l = l->next;
      return 0;
}
```



Ricerca in una lista

```
int ricerca(int e, list l) {
      int trovato = 0;
      while ((l!=NULL) && !trovato)
            if (l->value == e) trovato = 1;
            else l = l->next;
      return trovato;
}
```



DO IT!

Aggiungiamo al ***main*** che utilizza già chiamate a ***cons*** e ***showList*** anche

la lettura da input di un valore intero

che viene poi cercato in lista con la funzione ***ricerca***

e visualizziamo a video se trovato o meno
if (ricerca(i,root)) printf(“%s”, “trovato”);

ESERCIZIO : ricerca in una lista

```
int ricerca(int e, list l) {
        int trovato = 0;
        while ((l!=NULL) && !trovato)
                if (l->value == e) trovato = 1;
                else l = l->next;
        return trovato;
}
```

È una ***scansione sequenziale,*** **quale complessità?**

- nel caso ***peggiore***, occorre scandire l'intera lista *O(N)*
- nel caso ***migliore***, è il primo elemento *costante*
- nel caso ***medio***, proporzionale a N/2 *O(N)*

Esercizio proposto: progettare e implementare una soluzione ricorsiva (in Laboratorio – **Tutorato**)

Inserimento di un elemento in una lista

- In testa a una lista

  facile da realizzare

  l'ordine in lista è esattamente inverso rispetto all'ordine di inserimento

- In fondo a una lista

  facile da realizzare?

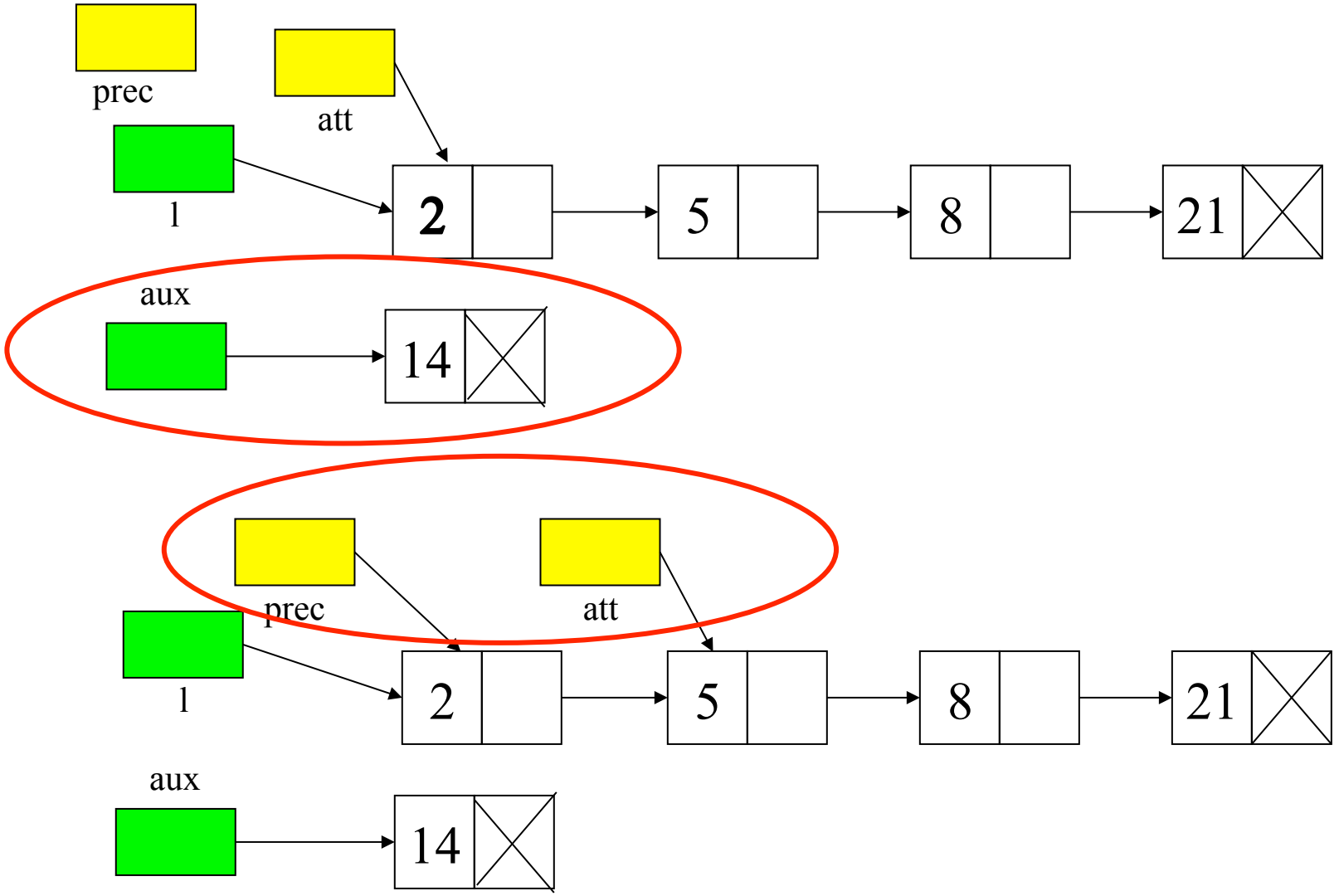
  l'ordine in lista è esattamente identico all'ordine di inserimento

- Inserimento ordinato

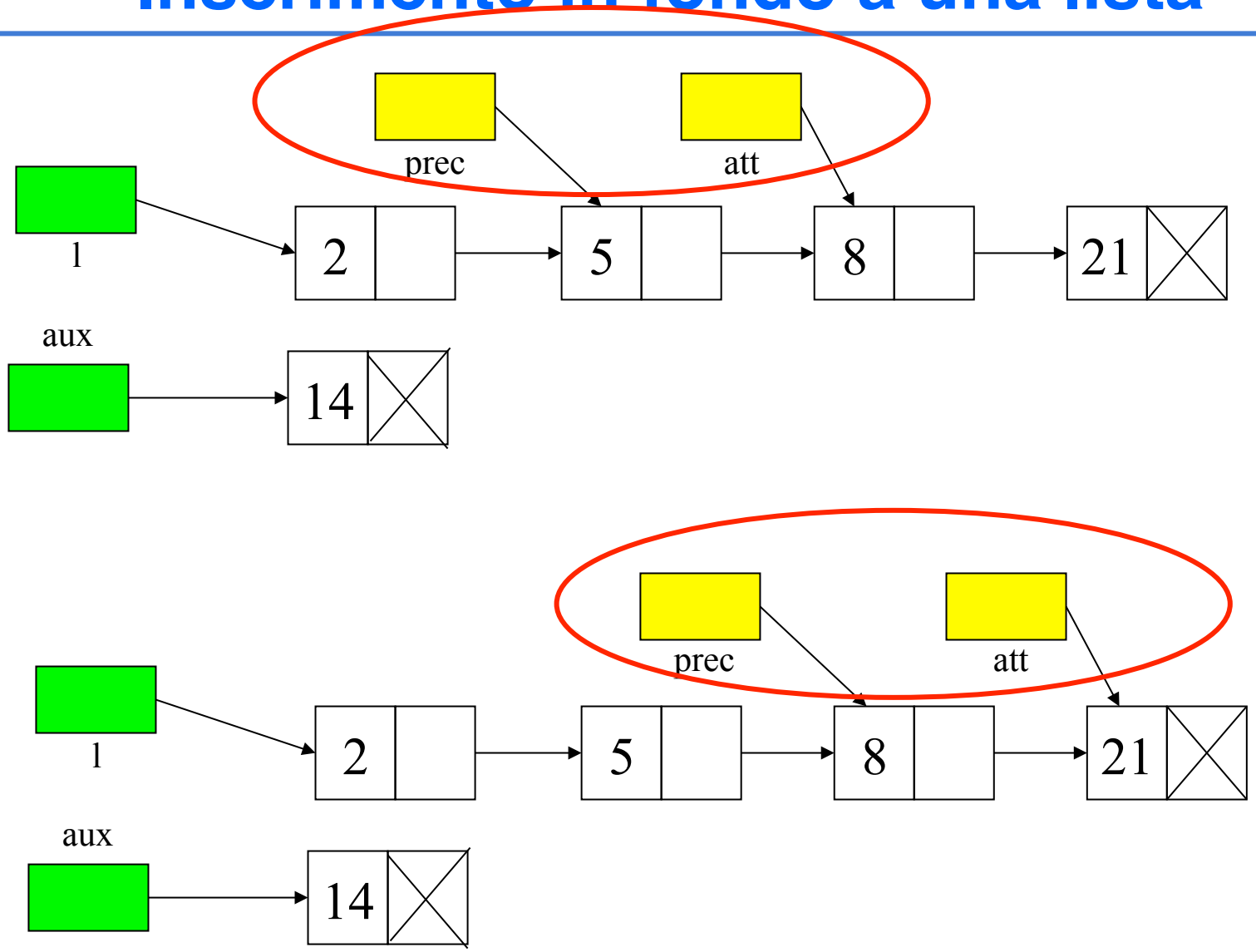
  facile da realizzare?

  l'ordine in lista rispetta la relazione d'ordine tra elementi utilizzata (crescente / decrescente / etc)

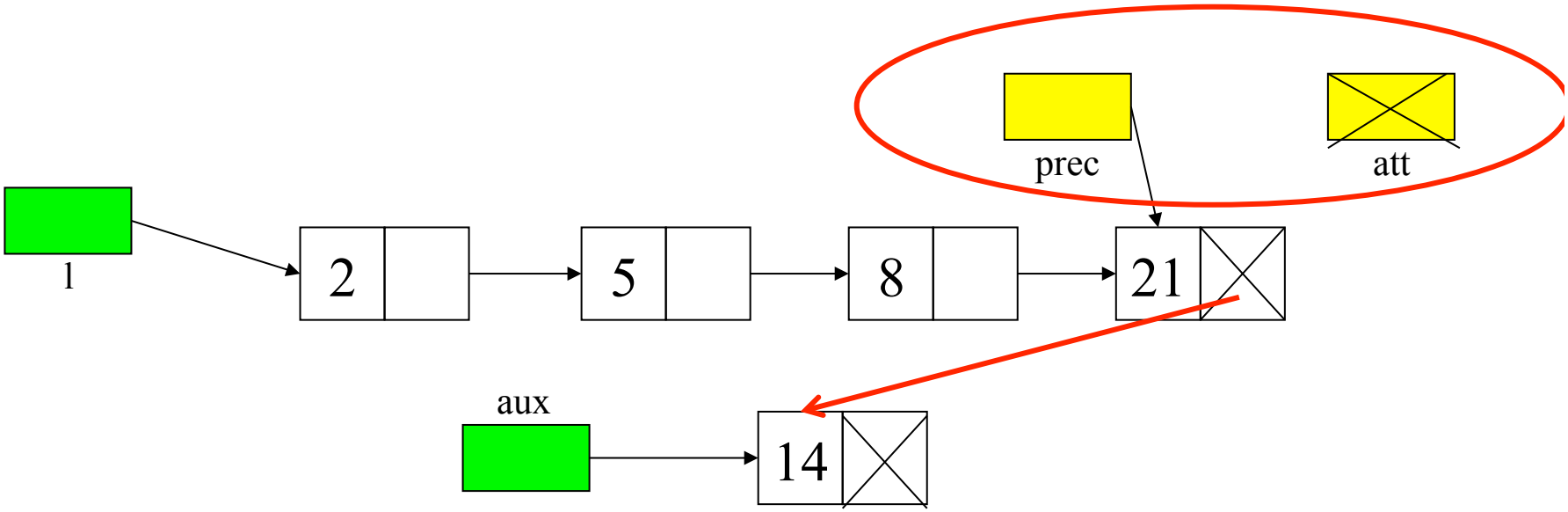
Inserimento in fondo a una lista



Inserimento in fondo a una lista



Inserimento in fondo a una lista

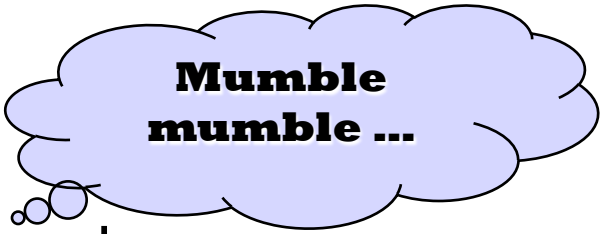


Inserimento in fondo

```
// FUNZIONE CHE INSERISCE IN FONDO - PRIMITIVA ITERATIVA

list  cons_tail(int e, list l) {
  list prec, aux;
  list patt=l;

  aux=(list)malloc(sizeof(item));          // ALLOCA NODO
  aux->value=e;
  aux->next=NULL;
  if (l==NULL) return aux;     // INSERISCE IN LISTA VUOTA
  else
    { while (patt!=NULL)              // NON FINE LISTA
        { prec=patt ;
          patt=patt->next; }

      prec->next=aux;                 // AGGIUNGE IN FONDO
      return l;                       // RESTITUISCE RADICE l
    }
}
```

DO IT!

Aggiungiamo al ***main*** che utilizza già chiamate a ***cons*** e ***showList*** anche ***cons_tail***



Lettura di una sequenza da input e costruzione di due liste: root con inserimento in testa e L2 in fondo

Stampa di root e L2 con ***showlist***

Come si presentano le due sequenze visualizzate?

Creazione e stampa di una lista di interi

```
#include <stdio.h>
#include <stdlib.h>
typedef struct list_element { int value;
              struct list_element *next; } item;
typedef item *list;
void showList(list l);
list cons(int e, list l);
list cons_tail(int e, list l);

main() {
list root = NULL; int i; list L2=NULL;
do {   printf("\nIntrodurre valore: \t");
       scanf("%d", &i);
       root = cons(i, root);
       L2=cons_tail(i, L2)
  } while (i!=0);
showList(root);
showList(L2);
 }
```

Inserimento di un elemento in una lista

- In testa a una lista

  facile da realizzare

  l'ordine in lista è esattamente inverso rispetto all'ordine di inserimento

- In fondo a una lista

  facile da realizzare?

  l'ordine in lista è esattamente identico all'ordine di inserimento

- Inserimento ordinato

  facile da realizzare?

  l'ordine in lista rispetta la relazione d'ordine tra elementi utilizzata (crescente / decrescente / etc)

LISTE ORDINATE

Deve essere definita una ***relazione d'ordine*** sul ***dominio-base*** degli elementi della lista

NOTA: criterio di ordinamento dipende da ***dominio base*** e dalla specifica ***necessità applicativa***

Ad esempio:

- interi ordinati in senso crescente, decrescente, …
- stringhe ordinate in ordine alfabetico, in base alla loro lunghezza, …
- persone ordinate in base all'ordinamento alfabetico del loro cognome, all'età, al codice fiscale, …

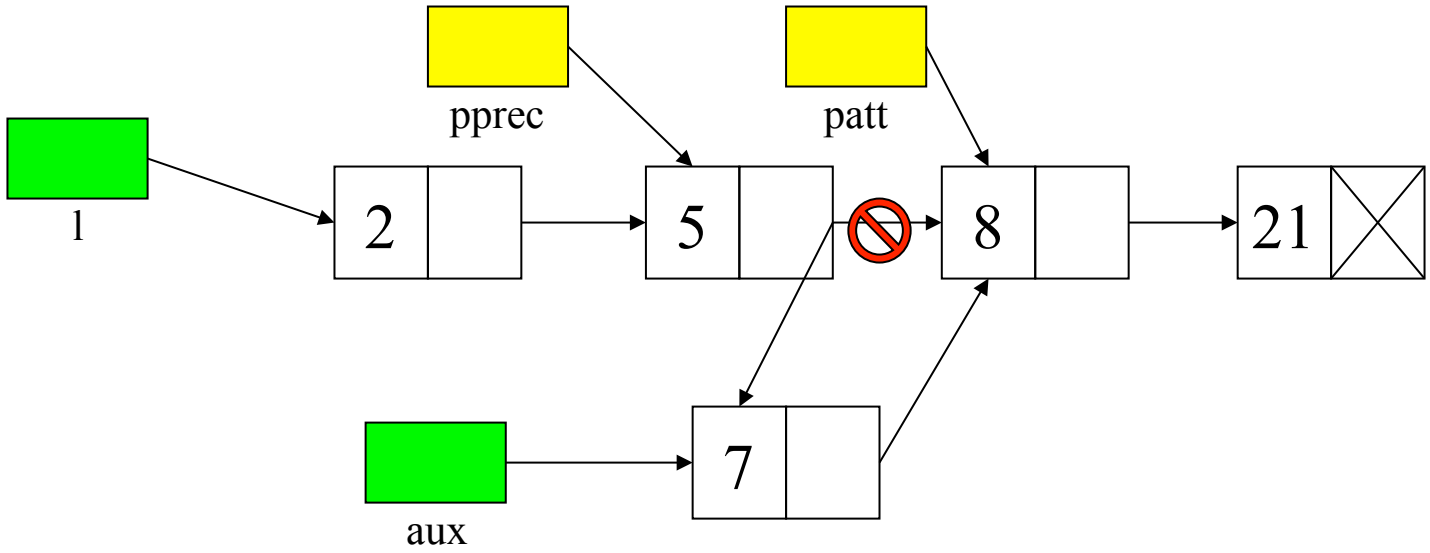
La funzione insord iterativa (vers. 3)

Versione primitiva e ***iterativa*** della funzione ***insord***, sapendo che la lista di partenza è ordinata

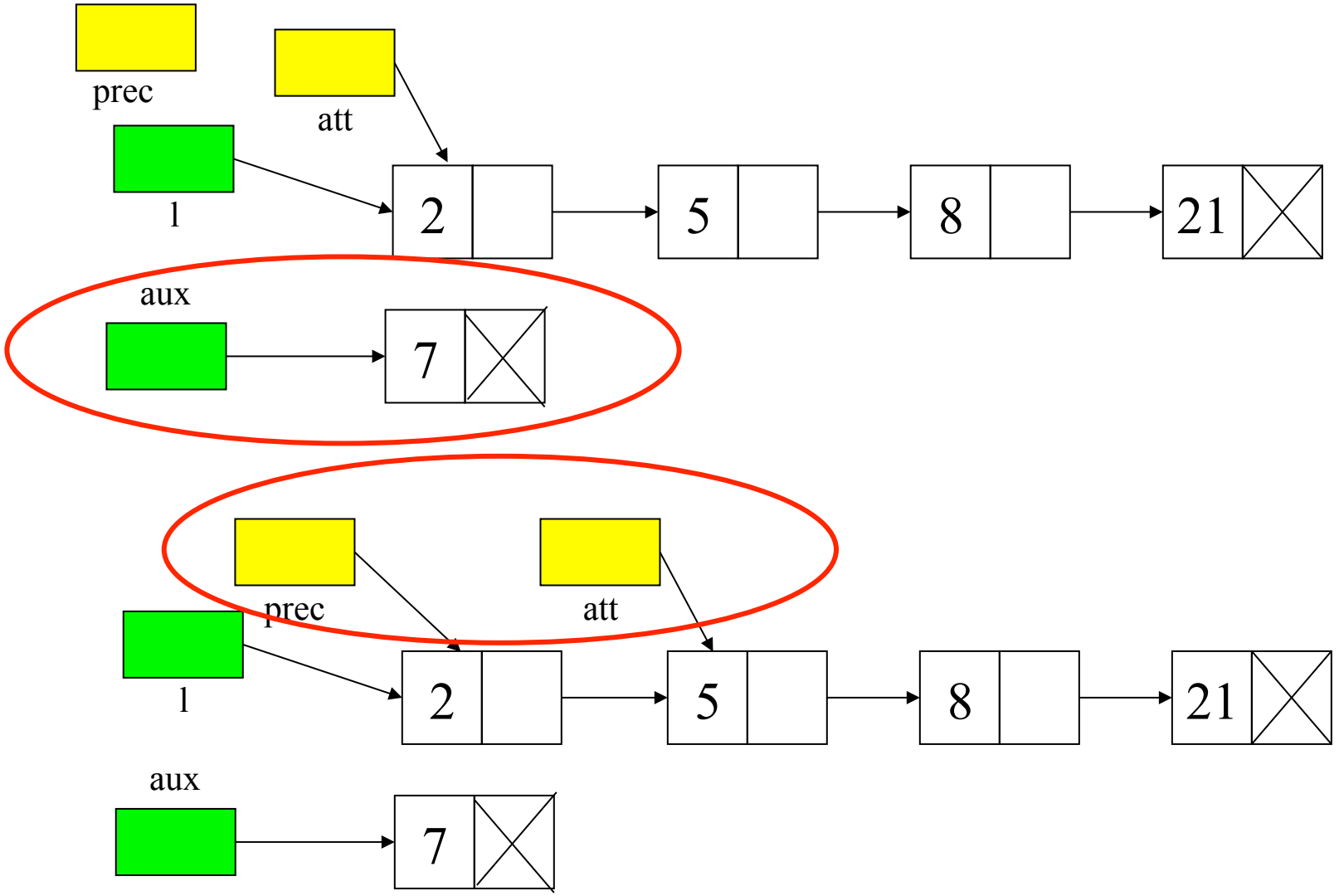
Algoritmo:

- Scandire la lista finché si incontra un nodo contenente un elemento maggiore di quello da inserire
- Allocare un nuovo nodo, con l'elemento da inserire
- Collegare il nuovo nodo ai due adiacenti (vedi figura)

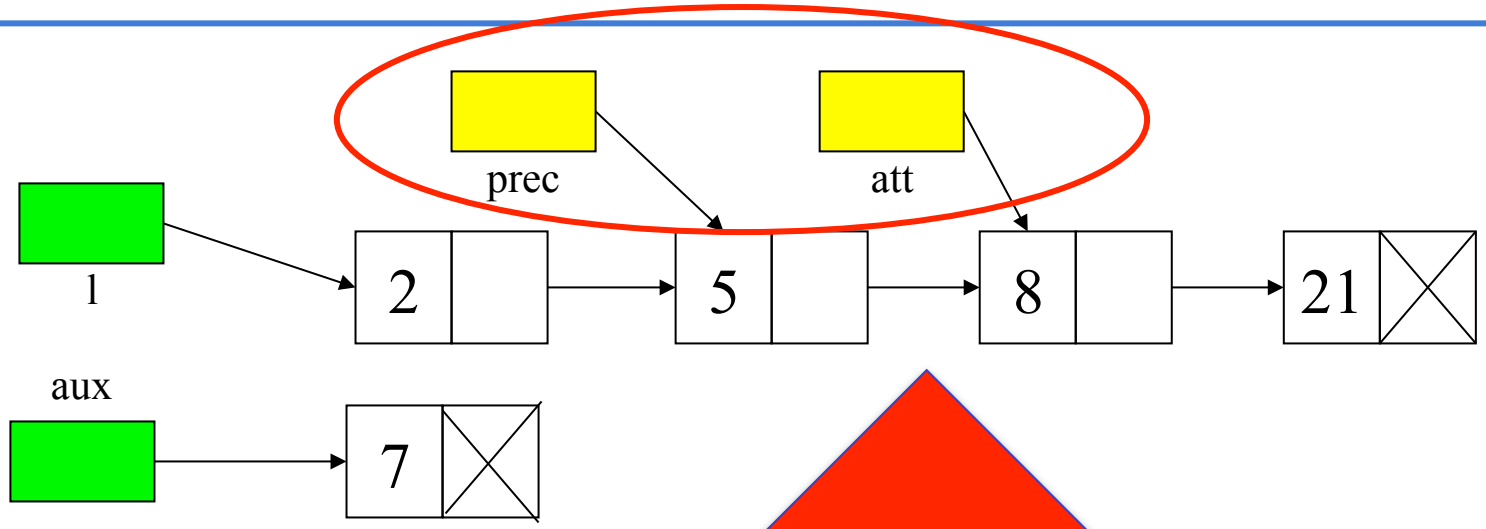
Somiglia all'inserimento in fondo visto?



Inserimento ordinato iterativo



Inserimento in fondo a una lista

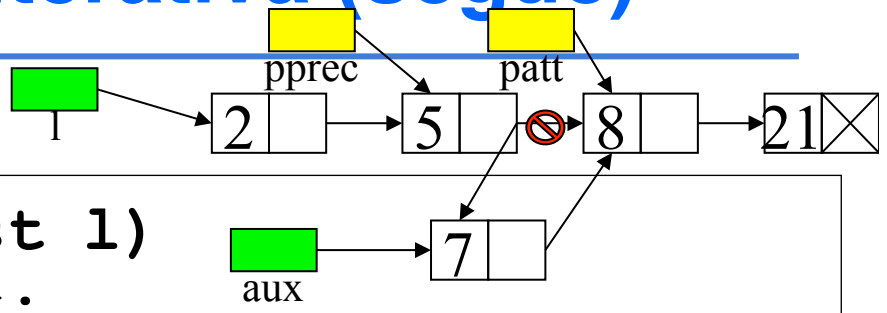


Il posto giusto del nuovo nodo è ***prima*** del nodo attuale: occorre anche il riferimento al ***nodo precedente***

Inserimento in fondo

```
// FUNZIONE CHE INSERISCE IN FONDO - PRIMITIVA ITERATIVA

list  cons_tail(int e, list l) {
  list prec, aux;
  list patt=l;

  aux=(list)malloc(sizeof(item));        // ALLOCA NODO
  aux->value=e;
  aux->next=NULL;
  if (l==NULL) return aux;     // INSERISCE IN LISTA VUOTA
  else
    { while (patt!=NULL)           // NON FINE LISTA
        { prec=patt ;
          patt=patt->next; }

      prec->next=aux;              // AGGIUNGE IN FONDO
      return l;                    // RESTITUISCE RADICE l
    }
}
```

La funzione insord iterativa (segue)




```
list insord_p(int el, list l)
{ list pprec, patt=l, aux;
  int trovato=0;
  while (patt!=NULL && !trovato)
  { if (el < patt->value) trovato = 1;
    else {pprec=patt; patt=patt->next; }  }

  aux=(list) malloc(sizeof(item));
  aux->value = el;
  aux->next = patt;
  if (patt==l) return aux;
  else { pprec->next = aux; return l; }
}
```

DO IT!

Riscriviamo il ***main*** creando tre liste, L1, L2, e L3, usando ***cons***, ***cons_tail*** e ***insord_p*** per inserire l'elemento letto:

```
L1= cons( i, L1);
L2= cons_tail( i, L2);
L3= insord_p( i, L3);
```

e stampiamole poi con tre chiamate a:

```
showList( );
```

Leggiamo un intero e cerchiamolo nelle tre liste:

```
ricerca(i, L );
```

Per la lista ordinata L3, cambia la complessità della **`ricerca`**?

ESERCIZIO : ricerca in una lista

```
int ricerca(int e, list l) {
        int trovato = 0;
        while ((l!=NULL) && !trovato)
                if (l->value == e) trovato = 1;
                else l = l->next;
        return trovato;
}
```

È sempre una ***scansione sequenziale,*** **anche se la lista è ordinata !**

- nel caso ***peggiore***, occorre scandire l'intera lista *O(N)*
- nel caso ***medio***, proporzionale a N/2 *O(N)*

Nota Bene: L'accesso agli elementi della lista è sempre sequenziale (dal primo all'ultimo), non c'è accesso diretto a elementi intermedi!

RICORSIONE: `showListr` ricorsiva

La lista va ***scandita (sequenzialmente) con un puntatore,*** *oppure* ***versioni ricorsive***

Stampa di una lista (ricorsiva):

```
void showListr(list l) {
      if( l!=NULL )
            { printf("%d", l->value);
              showListr( l->next ); }
}
```

LISTE ORDINATE: la funzione insord (1)

Per inserire un elemento in modo ordinato in una lista supposta ordinata:

- se la lista è vuota, costruire una ***nuova lista*** contenente il nuovo elemento, *altrimenti*
- se l'elemento da inserire è minore della testa della lista, aggiungere il ***nuovo elemento in testa*** alla lista data, *altrimenti*
- l'elemento andrà ***inserito più avanti*** della lista data

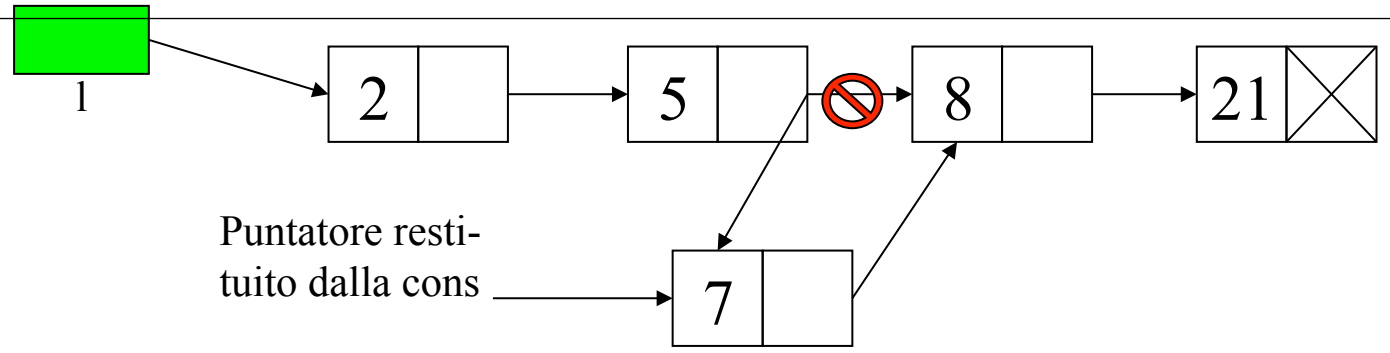
I primi due casi sono operazioni elementari.

Il terzo caso può essere trattato ***in modo iterativo*** o riconducendosi allo ***stesso problema ricorsivamente, ma su un caso più semplice***:

La funzione insord ricorsiva (segue)

Il terzo caso può essere trattato riconducendosi allo ***stesso problema ricorsivamente, ma su un caso più semplice***: alla fine si potrà effettuare o un inserimento in testa o ci si ricondurrà alla lista vuota

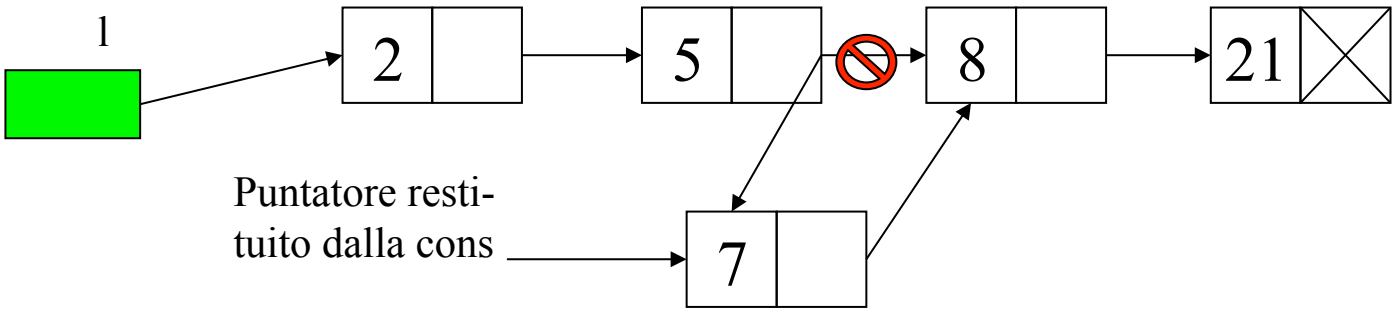
```
list insord(int el, list l) {
  if (l==NULL) return cons(el, l);
  else if (el<=l->value) return cons(el, l);
     else <inserisci el nel resto di l, e
           restituisci la lista così modificata>
}
```



La funzione insord ricorsiva

Il terzo caso può essere trattato riconducendosi allo ***stesso problema ricorsivamente, ma su un caso più semplice***: alla fine si potrà effettuare o un inserimento in testa o ci si ricondurrà alla lista vuota

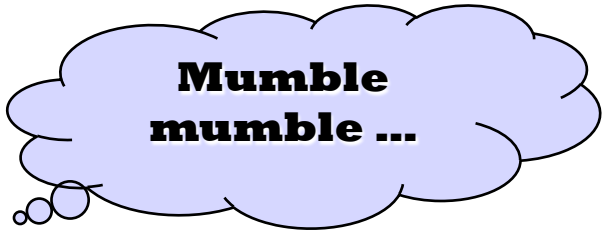
```
list insord(int el, list l) {
  if (l==NULL) return cons(el, l);
  else if (el<=l->value) return cons(el, l);
     else {l->next = insord(el,l->next); }
          return l; }
}
```



DO IT!

Riscriviamo il ***main*** creando tre liste, L1, L2, e L3, usando ***cons***, ***cons_tail*** e ***insord (insord_p)*** per inserire l'elemento letto:

```
L1= cons( i, L1);
L2= cons_tail( i, L2);
L3= insord( i, L3);
```



e stampiamole poi con tre chiamate a:

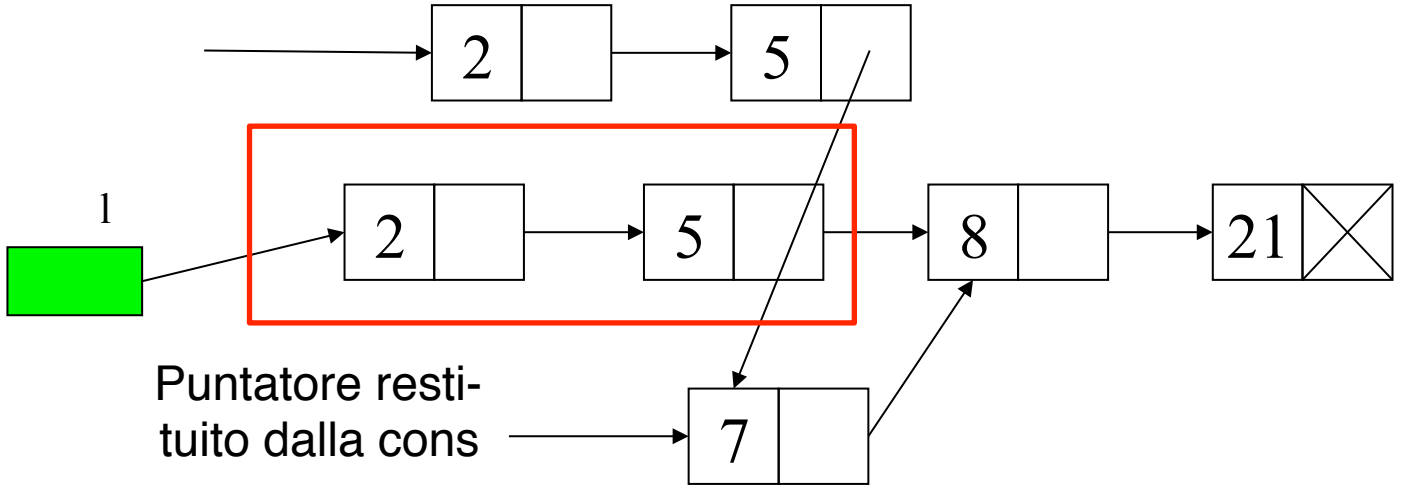
```
showList( );
```

Come si presentano le tre sequenze visualizzate?

La funzione insord ricorsiva (*v*ers. 2)

```
list insord(int el, list l) {
 if (l==NULL) return cons(el, l);
 else if (el<=l->value) return cons(el, l);
  else return cons(l->value,insord(el,l->next));
}
```

Structure copying: parte della struttura dati è replicata (dispendiosa come uso dell'heap!)



Tutorato

- Scrivere una versione ricorsiva della funzione **`showList`**
- Scrivere una versione ricorsiva della funzione **`ricerca`**
- Scrivere una versione iterativa e una ricorsiva della funzione **`lenght`** che calcoli la lunghezza di una lista
- Definire una funzione **`subList`** che, dato un intero positivo n e una lista l, restituisca una lista che rappresenti *la sotto-lista di quella data a partire dall'elemento n-esimo*

ESEMPIO: l = [1, 13, 7, 9, 10, 1]
subList(2, l) = [7, 9, 10, 1]