Liste

Il tipo astratto *Lista*

- Una ***lista*** è una sequenza di elementi di un determinato tipo, in cui è possibile aggiungere o togliere elementi. Una sequenza vuota è detta *lista vuota*.

- A differenza dell'array, che è una struttura a dimensione fissa dove è possibile accedere *direttamente* ad *ogni* elemento specificandone l'indice, la lista è a *dimensione variabile* e si può accedere ***direttamente*** solo al ***primo*** elemento della lista.

- Per accedere ad un generico elemento, occorre ***scandire sequenzialmente*** gli elementi della lista.

ADT *Lista*: Specifica sintattica

- **Tipo di riferimento**: list
- **Tipi usati**: item, boolean

- **Operatori**
  - newList() → list
  - emptyList(list) → boolean
  - consList(item, list) → list
  - tailList(list) → list
  - getFirst(list) → item

Usiamo un tipo generico item in quanto l'ADT Lista è indipendente dal tipo degli elementi contenuti nella lista

ADT *Lista*: Specifica semantica

- **Tipo di riferimento list**
  - list è l'insieme delle sequenze *L=a1,a2,…,an* di tipo ***item***
  - L'insieme list contiene inoltre un elemento ***nil*** che rappresenta la lista vuota (priva di elementi)

ADT *Lista*: Specifica semantica

- **Operatori**
  - newList() → l
    - Post: l = nil
  - emptyList(l) → b
    - Post: se l=nil allora b = true altrimenti b = false
  - consList(e, l) → l'
    - Post: l = <a1, a2, … an> AND l' = <e, a1, a2, …, an>
  - tailList(l) → l'
    - Pre: l = <a1, a2, …, an> n>0
    - Post: l' = <a2, …, an>
  - getFirst(l) → e
    - Pre: l = <a1, a2, …, an> n>0
    - Post: e = a1

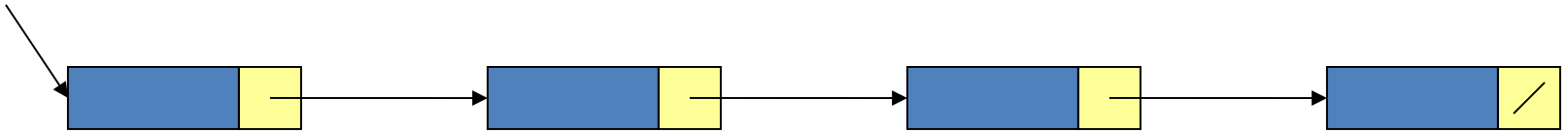
Il tipo astratto *Lista*

- … e adesso una possibile implementazione

- le **liste concatenate**

Le Liste Concatenate

- Ogni elemento di una lista concatenata è un nodo con un riferimento che serve da collegamento per il nodo successivo.



- Si accede alla struttura attraverso il riferimento al primo nodo.
- Il riferimenti dell'ultimo nodo contiene un valore nullo

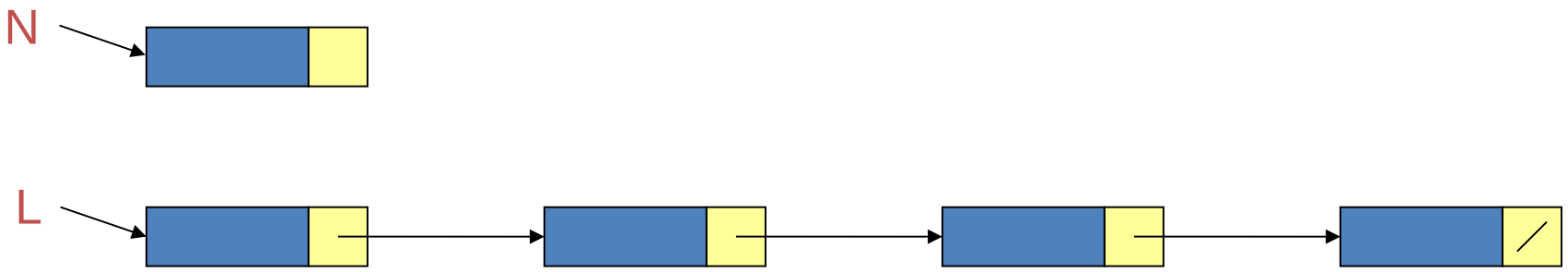
Liste concatenate

- Una lista concatenata è più flessibile di un array; è più facile inserire o cancellare un elemento, utilizzando solo la memoria strettamente necessaria
- C'è uno svantaggio rispetto agli array: si perde la capacità di accedere in modo diretto agli elementi della lista, usando l'indice dell'array:
  - Ogni elemento di un array è accessibile direttamente usando il suo indice (tempo costante)
  - Per accedere all'*i*-esimo elemento di una lista occorre scorrere la lista dal primo all'*i*-esimo elemento (tempo proporzionale ad *i*)

Inserire un elemento in una lista concatenata

- Il modo più semplice per inserire un nuovo elemento in una lista concatenata L è inserire l'elemento in un nuovo nodo da aggiungere in testa alla lista

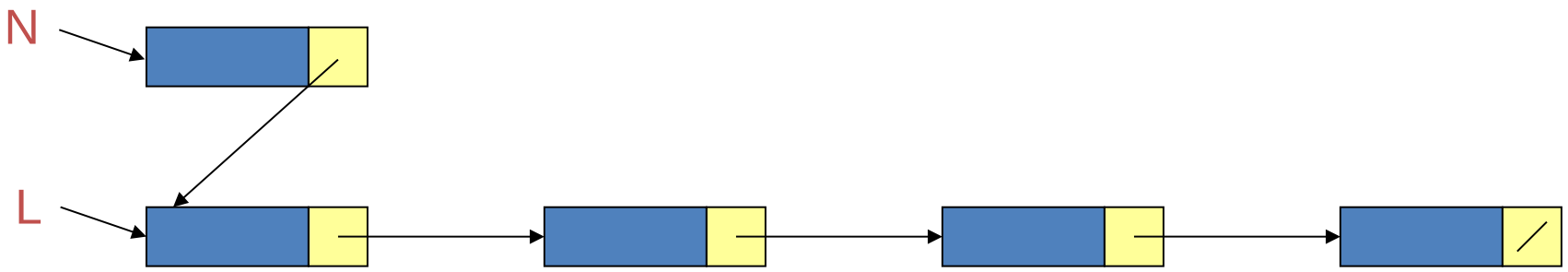
1. per prima cosa si crea il nuovo nodo, poi si aggiunge il collegamento con il record iniziale della lista



Inserire un elemento in una lista concatenata

- Il modo più semplice per inserire un nuovo elemento in una lista concatenata L è inserire l'elemento in un nuovo nodo da aggiungere in testa alla lista

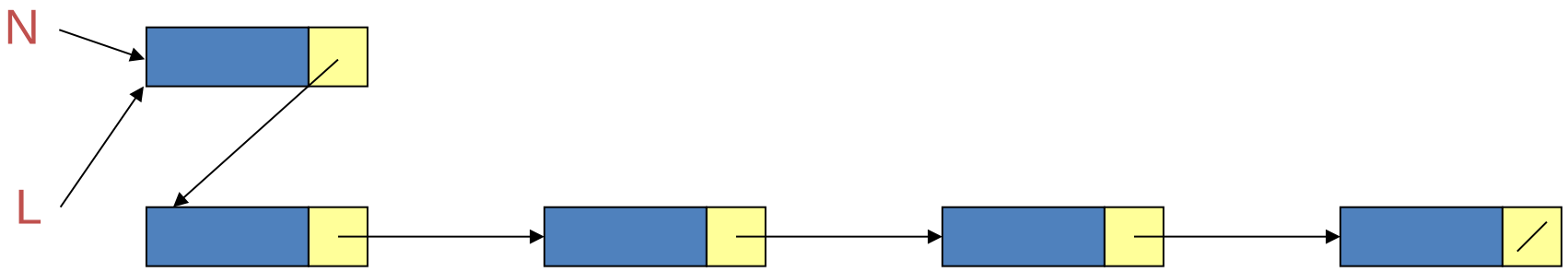
1. per prima cosa si crea il nuovo nodo, poi si aggiunge il collegamento con il record iniziale della lista



Inserire un elemento in una lista concatenata

- Il modo più semplice per inserire un nuovo elemento in una lista concatenata L è inserire l'elemento in un nuovo nodo da aggiungere in testa alla lista

1. per prima cosa si crea il nuovo nodo, poi si aggiunge il collegamento con il nodo iniziale della lista



2. Poi si aggiorna L facendolo puntare al nodo appena aggiunto

… e adesso un po’ di codice C

Dichiarazione del tipo di un nodo

- Per usare una lista concatenata serve una struttura che rappresenti i nodi
- La struttura conterrà i dati necessari (un intero nel seguente esempio) ed un puntatore al prossimo elemento della lista:

```
struct node {
  item value;                  /* data stored in the node  */
  struct node *next;     /* pointer to the next node */
};
```

Struttura auto-referenziata

```
struct node {
  item value;                  /* data stored in the node  */
  struct node *next;     /* pointer to the next node */
};
```

- node definisce una struttura che contiene un campo che punta ad un altra struttura di tipo node

Dichiarazione del tipo lista

- Il passo successivo è quello di dichiarare il tipo lista

```
typedef struct node *list;
```

- una variabile di tipo lista punterà al primo nodo della lista:

```
list L = NULL;  // in questo caso, L rappresenta la lista vuota
```

- Assegnare a l il valore NULL indica che la lista è inizialmente vuota

Creare un nodo della lista

- Man mano che costruiamo la lista, creiamo dei nuovi nodi da aggiungere alla lista
- I passi per creare un nodo sono:
  1. Allocare la memoria necessaria
  2. Memorizzare i dati nel nodo
  3. Inserire il nodo nella lista
- Vediamo i primi due passi per adesso

Creare un nodo della lista

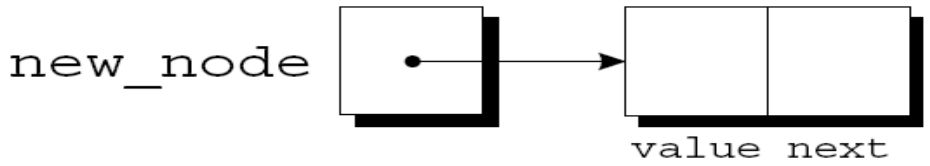
- Per creare un nodo ci serve un puntatore temporaneo che punti al nodo:

  struct node *new_node;

- Possiamo usare malloc per allocare la memoria necessaria e salvare l'indirizzo in new_node:

  new_node = malloc(sizeof(struct node));

- new_node adesso punta ad un blocco di memoria che contiene la struttura di tipo node:



Implementare il tipo astratto *Lista*: header file list.h

```
// file list.h

typedef struct node *list;

// prototipi

list newList(void);

int emptyList(list l);

list tailList(list l);

list consList(item val, list l);

item getFirst (list l);
```

L'ADT lista è indipendente dal tipo degli elementi contenuti.

Definiamo quindi un tipo generico item in un file item.h

I file item.h e item.c

```
// file item.h

typedef int item;

#define NULLITEM 0

/* per semplicità in questo es.
   il nostro tipo item è l'insieme
   degli interi positivi
   NULLITEM è un elemento che
   viene restituito quando la
   precondizione di getFirst
   viene violata */

int eq(item x, item y);
void input_item(item *x);
void output_item(item x);
```

```
// file item.c

/* il modulo contiene per ora tre
operatori che useremo nel seguito.
Aggiungerne altri all'occorrenza */

#include <stdio.h>
#include "item.h"

int eq(item x, item y) {
   return x == y;
}

void input_item(item *x) {
     scanf("%d", x);
}

void output_item(item x) {
      printf("%d", x);
}
```

Implementare il tipo astratto *Lista*:

file list.c

```
#include <stdio.h>
#include <stdlib.h>
#include “item.h”
#include “list.h”


struct node {
    item value;
    struct node *next;
};



list newList(void)
{
    return NULL;
}
```

```
int emptyList(list l)
{
    return l == NULL;
}


list consList(item val, list l)
{
     struct node *nuovo;
     nuovo = malloc (sizeof(struct node));
     if (nuovo != NULL) {
          nuovo->value = val;
          nuovo->next = l;
          l = nuovo;
          }
     return l;
}
```

Implementare il tipo astratto *Lista*:

file list.c

Osservazione: per modificare una lista *L* con l'aggiunta di un nuovo elemento *val*, l'operatore *consList()* va usato nel modo seguente:

```
list L;
……….
……….

L = consList(val, list L);
```

file list.c

```
list tailList(list l)
{
      list temp;
      if (l != NULL)
            temp = l->next;
      else
            temp = NULL;
      return temp;
}

// uso tipico dell'operatore:
//   L = tailList(L);
```

```
item getFirst (list l)
{
      item e;
      if(l != NULL)
            e = l->value;
      else
            e = NULLITEM;
      return e;
}
```

Alcune note sull'ADT lista

- L'insieme degli operatori così definiti costituisce l'insieme degli operatori di base (il minimo insieme di operatori) di una lista
- Ogni altro operatore che si volesse aggiungere all'ADT lista potrebbe essere implementato utilizzando gli operatori dell'insieme di base
- Alcuni esempi
  - calcolo della lunghezza di una lista
  - ricerca di un elemento in una lista
  - inserimento/cancellazione in una posizione intermedia
  - …

Aggiungiamo operatori alla lista

- **Specifica sintattica**
  - sizeList(list) → integer
  - posItem(list, item) → integer
  - searchItem(list, item) → boolean
  - reverseList(list) → list
  - removeItem(list, item) → list
  - getItem(list, integer) → item
  - insertList(list, integer, item) → list
  - removeList(list, integer) → list
  - …

Aggiungiamo operatori alla lista

- **Specifica semantica**
  - sizeList(l) → n
    - Post: l = <a1, a2, …, an> AND n ≥ 0
  - searchItem(l, e) → b
    - Post: se e è contenuto in l allora b = true, se no b = false
  - posItem(l, e) → p
    - Post: se e è contenuto in l allora p è la posizione della prima occorrenza di e in l, altrimenti p = -1
  - reverseList(l) → l'
    - Post: l = <a1, a2, …, an> AND l' = <an, …, a2, a1>
  - removeItem(l, e) → l'
    - Post: se e è contenuto il l, allora l' si ottiene da l eliminando la prima occorrenza di e in l, altrimenti l' = l

Aggiungiamo operatori alla lista

- **Specifica semantica**
  - getItem(l, pos) → e
    - Pre: pos >= 0 AND sizeList(l) > pos
      // assumiamo 0 come prima posizione
    - Post: e è l'elemento che occupa in l la posizione pos
  - insertList(l, p, e) → l'
    - Pre: pos >=0 AND sizeList(l) ≥ p
      // assumiamo 0 come prima posizione
    - Post: l' si ottiene da l inserendo e in posizione p
  - removeList(l, p) → l'
    - Pre: pos >=0 AND sizeList(l) > p
    - Post: l' si ottiene da l eliminando l'elemento in posizione p

NB: ***getItem(l, 0) = getFirst*** ***insert(l, 0, val) = consList(val, l)*** ***remove(l, 0) = tailList(l)***

Vediamo alcune implementazioni

Implementiamo alcuni dei nuovi operatori mediante l'uso degli operatori di base …

Implementare i rimanenti come esercizio …

Implementiamo anche due funzioni inputList e outputList utili per realizzare programmi con le liste …

Implementazione di sizeList

- Realizziamo la funzione **sizeList(*l*)** che restituisce il numero di elementi di una lista *l*
- L'algoritmo richiede una ***visita totale*** della lista: il ciclo si arresta quando la lista *l* termina
- Ad ogni iterazione aggiungiamo 1 ad un contatore n a cui inizialmente assegnamo il valore 0
- In generale, per visitare una lista usiamo le operazioni getFirst(*l*) per accedere al primo elemento e tailList(*l*) per accedere alla parte restante della lista

- Schema di visita totale:

```
while(!emptyList(l)) {
        operazioni specifiche sul primo elemento
        l = tailList(l); // continuiamo la visita
}
```

Implementazione di sizeList

```
int sizeList (list l)
{
    int n=0;
    while (!emptyList(l)) // finché non raggiungiamo la fine della lista
    {
        n++;                    // il primo elemento contribuisce al conteggio
        l = tailList(l);        // continuiamo la visita degli elementi successivi
    }
    return n;
}
```

Esempio di chiamata:

```
int size = sizeList (L1);     // la lista L1 non viene modificata
```

Implementazione di posItem

- Realizziamo la funzione **posItem (l, val)** che, dati una lista l e un elemento val, restituisce la posizione della lista in cui appare la prima occorrenza dell'elemento, oppure -1 se l'elemento non è presente
- Richiede una ***visita finalizzata*** della lista: usciamo da ciclo quando troviamo l'elemento cercato oppure quando raggiungiamo la fine della lista (usiamo una variabile booleana found che viene settata a 1 (true) quando troviamo l'elemento)
- Schema di visita finalizzata:

```
found = 0;        // all'inizio l'elemento non è stato trovato
while (! emptyList(l) && ! found) {
        if(getFirst(l) è l'elemento cercato)
                found = 1;
        else {
                operazioni specifiche sul primo elemento
                l = tailList(l);  // continuiamo la visita
        }
}
```

Implementazione di posItem

```
int posItem (list l, item val)
{
    int pos =0; // contatore di posizione
    int found =0;

    while (!emptyList(l) && !found) {
       if (eq(getFirst(l), val))
          found =1;
       else {
          pos++; // incrementa il contatore di posizione
          l = tailList(l);  // continuiamo la visita degli elementi della lista
          }
     }

    if(!found)
          pos = -1;  // se non trovato restituiamo come posizione -1

    return pos;
}
```

Implementazione di posItem (vers. Ricorsiva)

```
int posItem (list l, item val)
{
    if emptyList(l) return -1;
    if (eq(getFirst(l), val))  return 0;
    else
    {
        int ris =  posItem(tailList(l), val);
        if (ris== -1)
            return -1;
        else
            return 1 + ris;
    }
}
```

Implementazione di getItem

Realizziamo la funzione **getItem(l, pos)** che, dati una lista l e un intero pos, restituisce l'elemento in l di posizione pos, oppure l'elemento nullo se la lista ha meno di pos+1 elementi

```
item getItem (list l, int pos)
{
      item e;
      int i =0;
     // prima scorriamo la lista fino alla posizione pos … se esiste
      while (i < pos && !emptyList(l)) {
          i++;
          l = tailList(l);
      }

      if (!emptyList(l)) // se la lista ha almeno pos+1 elementi
          e = getFirst(l);  // elemento di posizione pos
      else e = NULLITEM;

      return e;
}
```

Implementazione di reverseList

Realizziamo la funzione **reverseList(l)** che, dati una lista restituisce una nuova lista che ha gli elementi della lista in ordine inverso (***schema di visita totale della lista***)

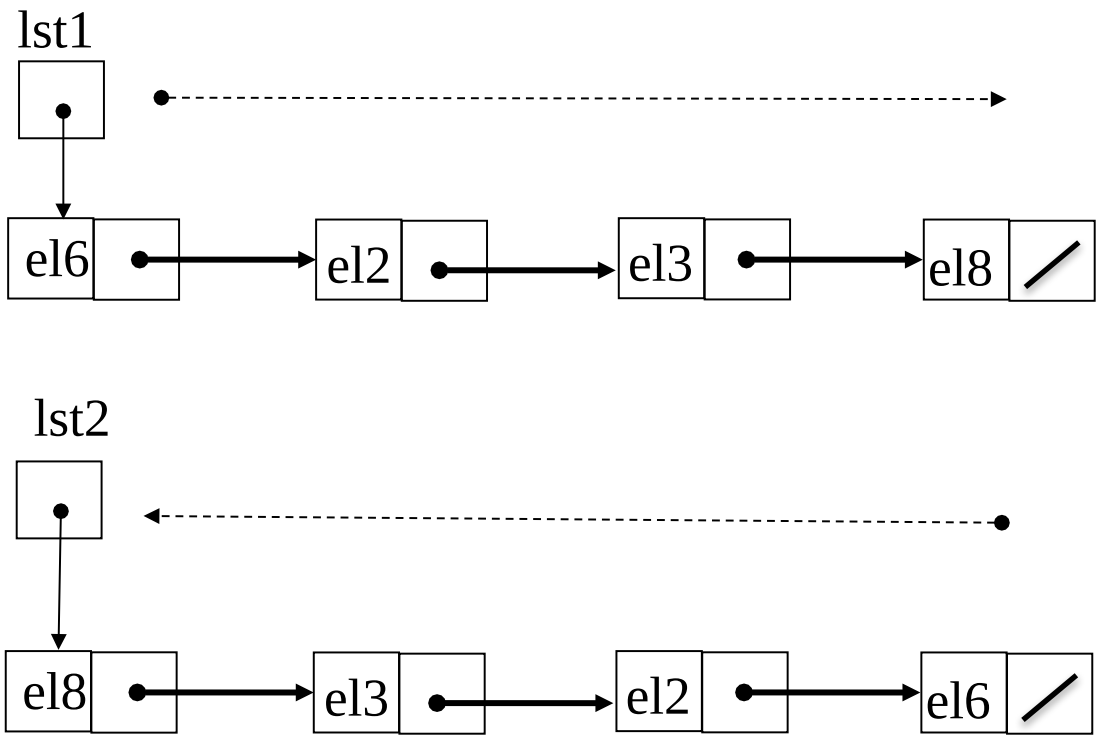
```
list reverseList (list l)
{
      list rev;
      item val;

      rev = newList();

      while (!emptyList(l)) {
         val = getFirst(l);
         rev = consList(val, rev);
         l = tailList(l);
       }

      return rev;
}
```

lst2 = reverseList(lst1);



Implementazione di outputList

Realizziamo la funzione **outputList(l)** che prende visualizza in output gli elementi di una lista l (***visita totale della lista***)

```
void outputList (list l)
{
      int i =0;
      item val;

      while(!emptyList(l)) {
          val = getFirst(l);
          printf("Elemento di posizione %d: ", i);
          output_item(val);
          printf("\n");
          l = tailList(l);
          i++;
       }
}
```

Implementazione di outputList (vers. Ricorsiva)

Realizziamo la funzione **outputList(l)** che prende visualizza in output gli elementi di una lista l (***visita totale della lista***)

```
void outputList (list l)
{
    out_List(l, 0);
}
```

```
void out_List (list l, int i)
{
      item val;

      if (emptyList(l)) return;
      val = getFirst(l);
     printf("Elemento di posizione %d: ", i);
     output_item(val);
      printf("\n");
      out_List(tailList(l), i+1);
}
```

Implementazione di inputList

Realizziamo la funzione **inputList(n)** che prende in ingresso un intero n e restituisce una lista di n elementi inseriti da standard input

```
list inputList (int n)
{
    item val;
    list l = newList();

    for(int i = 0; i < n; i++) {
        printf("Elemento di posizione %d: ", i);
        input_item(&val);
        l = consList(val, l);
    }

// alla fine del ciclo l contiene gli elementi della lista al contrario

    return reverseList(l);
}
```

Fine prima lezione sulle liste

- Prossima lezione: operatori di inserimento/rimozione