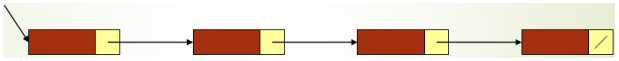
L04. ADT LISTA

Il tipo astratto **Lista** è una sequenza di elementi di un *determinato tipo (anche Generics)*, in cui è possibile aggiungere o togliere elementi, è possibile specificare la posizione relativa nella quale l'elemento va aggiunto o tolto.

Liste concatenate	***VS***	***Array***
Dimensione variabile, accesso diretto solo al primo elemento della lista. Per accedere ad un generico elemento, occorre **scandire sequenzialmente** gli elementi della lista: ▪ Per accedere all'i-esimo elemento occorre scorrere la lista dal primo all'i-esimo elemento (*tempo max proporzionale ad n*); ▪ Dato un elemento, è possibile eliminarlo o aggiungerne uno dopo direttamente (*tempo costante*).		Dimensione fissa, **accesso diretto** ad ogni elemento (con indice): ▪ Ogni elemento di un array è accessibile direttamente usando il suo indice (*tempo costante*); ▪ Per l'eliminazione occorre effettuare delle operazioni di shift (*tempo max proporzionale ad n*).

Per quanto riguarda la progettazione di liste concatenate, ogni elemento di una lista concatenata è un **record** con un campo puntatore che serve da collegamento per il record successivo.



Si accede alla struttura attraverso il puntatore al primo record ed il campo puntatore dell'ultimo record contiene il valore NULL.

Sintattica	Semantica
Nome del tipo: List Tipi usati: Item, boolean	Dominio: insieme di sequenze L=$a_1,\ldots,a_n$ di tipo Item L'elemento **nil** rappresenta la lista vuota
newList() → List	newList() → l • Post: l = nil
isEmpty(List) → boolean	isEmpty(l) → b • Post: se l=nil allora b = true altrimenti b = false
addHead(List, Item) → List	addHead(l, e) → l' • Post: l = <a1, a2, … an> AND l' = <e, a1, …, an>
removeHead(List) → List	removeHead(l) → l' • Pre: l = <a1, a2, …, an> n>0 • Post: l' = <a2, …, an>
getHead(List) → Item	getHead(l) → e • Pre: l = <a1, a2, …, an> n>0 • Post: e = a1

Vediamo come implementare il tipo **Lista**:

Per prima cosa occorre dichiarare il tipo lista (*rispettando l'inf. hiding*):

```
typedef struct list *List;      //NOTA: si troverà nel file .h
struct list{                    //NOTA: si troverà nel file .c
   int size;                    //Aggiornata di volta in volta
   struct node *head;           //Puntatore al nodo testa
};
```

Per istanziare la lista occorre riservare memoria e inizializzarla vuota:

```
List list = malloc(sizeof(struct list));
list->size = 0;
list->head = NULL;
```

Per usare una lista concatenata serve una struttura che rappresenti i **nodi**:

Si usa una struttura auto-referenziale:

```
struct node {                   //NOTA: si troverà nel file .c
   Item value;                  //dati nel nodo
   struct node *next;           //puntatore al prossimo nodo
};
```

La struttura conterrà i dati necessari ed un puntatore al prossimo elemento della lista.

Per iterare sui nodi si può usare un ciclo for:

```
for(p = list->head; p != NULL; p = p->next)
    outputItem(p->value);
```

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.

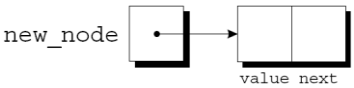
Per creare un nodo ci serve un puntatore temporaneo che punti al nodo:

```
struct node *new_node;
```

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

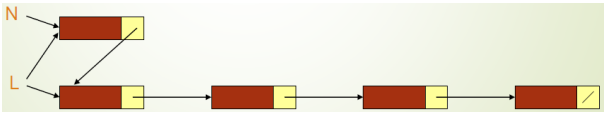
```
new_node = malloc(sizeof(struct node));
```

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



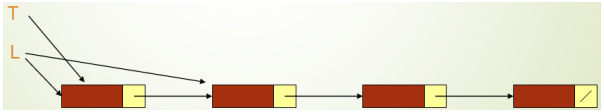
Per **inserire un nuovo elemento in una lista concatenata *semplice* L** è inserire l'elemento in un nuovo nodo da aggiungere in testa alla lista:

1. Si alloca il nuovo nodo N;
2. Si aggiunge il collegamento con il record iniziale della lista;
3. Si aggiorna L facendolo puntare a N.



Per **eliminare un elemento in una lista concatenata *semplice* L** è eliminarlo in testa alla lista:

1. Si crea un puntatore temporaneo T, copia di L
2. Si aggiorna L facendolo puntare al successivo di L
3. Si elimina il nodo puntato da T, liberando la memoria



Alcuni operatori richiedono una **visita parziale** o **totale della lista**, ad esempio:

- **Visita totale**: usata per calcolare la size, dove scorriamo i nodi incrementando un contatore (*ottimizzazione*: mantenere un contatore da aggiornare), oppure la stampa degli elementi;
- **Visita parziale**: inserimento o rimozione in una posizione i oppure la ricerca di un elemento.

Adesso implementiamo alcune funzioni per una **lista concatenata *più complessa***.

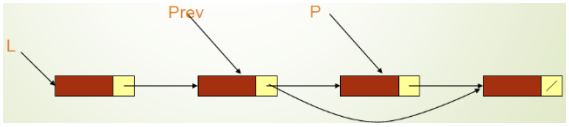
Sintattica	Semantica
Nome del tipo: List Tipi usati: Item, boolean	Dominio: insieme di sequenze $L=a_1,\dots,a_n$ di tipo Item L'elemento **nil** rappresenta la lista vuota
searchItem(List, Item) → int	searchItem(l, i) → pos • Post: se i in l allora pos = pos. di i in l else pos = -1
removeItem(List, Item) → List	removeItem(l, e) → l' • Pre: l = <a1, a2, …, e, …, an> n>0 • Post: l' = l - <e>
removeItem(List, int) → List	removeItem(l, pos) → l' • Pre: l = <a1, a2, …, a_{pos}, …, an> 1 <= pos <= n • Post: l' = l - <a_{pos}>
insertItem(List, Item, int) → List	insertItem(l, e, pos) → l' • Pre: l = <a1, …, an> & 1 <= pos <= n+1 • Post: l' = <a1, …, a_{pos}, …, a_{n+1}> & a_{pos}=e
insertTail(List, Item) → List	insertTail(l, e) → l' • Post: l = <a1, …, an> & l' = <a1, …, an, e>
reverseList(List) → List	reverseList(l) → l' • Post: l = <a1, a2, …, an> AND l' = <an, …, a2, a1>
cloneList(List) → List	cloneList(l) → l' • Post: l = <a1, a2, …, an> AND l' = <a1, a2, …, an>

Per quanto riguarda la **ricerca di un elemento**, 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, cioè usciamo da ciclo quando troviamo l'elemento cercato oppure quando raggiungiamo la fine della lista e possiamo ottenere sia il riferimento all'Item ricercato, sia la sua posizione, implementando la funzione: Item searchItem(List list, Item item, int*pos)

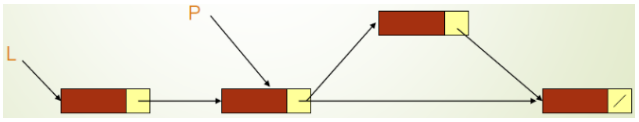
Per **eliminare un elemento** (esempio il terzo) in una lista concatenata L:

1. Si fanno avanzare due puntatori Prev e P fino a che P punta al nodo da eliminare;
2. Si aggiorna il nodo puntato da Prev, facendolo puntare al successivo di P;
3. Si elimina il nodo puntato da P, liberando la memoria.



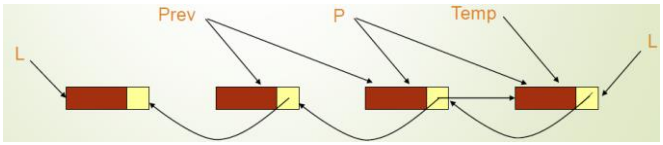
Per **inserire un elemento** (esempio in terza posizione) in una lista concatenata L:

1. Si fa avanzare un puntatore P, fino a che punta al nodo precedente alla posizione dell'inserimento;
2. Si crea il nuovo nodo e lo si fa puntare al successivo di P;
3. Si fa puntare P al nuovo nodo.



Per **invertire una lista concatenata L**:

1. ***Per ogni nodo*** (con i nodi fino a Prev già invertiti) si utilizzano due puntatori Prev e P;
   a) Si salva il successivo di P in un puntatore temporaneo Temp
   b) Si aggiorna il nodo puntato da P, facendolo puntare a Prev
   c) Si fanno avanzare P e Prev
2. Si aggiorna la testa facendola puntare all'ultimo nodo

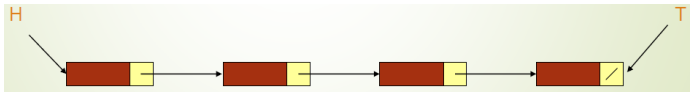


Per **clonare una lista concatenata** esistono diverse scelte progettuali:

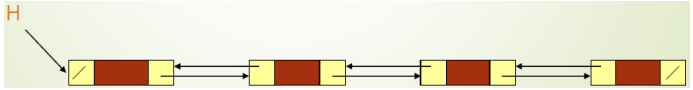
- **Clonare solo la struct list**, i nodi sono gli stessi ed una modifica su una lista viene riflessa nell'altra;
- **Clonare i nodi ma non gli item**, modifiche alla struttura di una lista non vengono riflesse nell'altra, ma se si modifica un item, risulta modificato in entrambe le liste;
- **Clonare i nodi e gli item**, le liste una volta clonate sono totalmente indipendenti.

Per concludere, esistono implementazioni alternative della struttura Lista:

- **Lista a collegamento singolo**: manteniamo la struttura lista vista precedentemente, aggiungendo un puntatore **tail** alla fine della lista, questo ci consente di ottimizzare l'operazione di inserimento in coda;



- **Lista circolare**: l'ultimo nodo ha un collegamento al primo nodo della lista, il vantaggio è di poter iterare lungo tutta la lista partendo da un nodo qualsiasi;
- **Lista a collegamento doppio**: ogni nodo ha due puntatori, uno al nodo precedente ed uno al successivo, il vantaggio è che possiamo effettuare operazioni di cancellazione e inserimento in tempo costante, lo svantaggio è che il codice diventa complesso.



Tutte queste possibili implementazioni possono essere combinate.

Item.h

```
typedef void* Item;

Item inputItem();
void outputItem(Item);
int cmpItem(Item,Item);
Item cloneItem(Item);
```

lista.h

```
#include "item.h"

typedef struct list *List;

List newList();
int isEmpty(List);
void addHead(List, Item);
Item removeHead(List);
Item getHead(List);
int sizeList(List);
void printList(List);
void sortList(List);
Item searchList(List, Item, int *);
Item removeListItem(List, Item);
Item removeListItemPos(List, int );
int addListItem(List, Item, int);
int addListTail(List, Item);
void reverseList(List);
List cloneList(List);
```

lista.c

```
#include <stdio.h>
#include <stdlib.h>
#include "item.h"
#include "list.h"
#include "utils.h"

struct list {
  int size;
  struct node *head;
};

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

struct node * minimo (struct node *p);

List newList(){
  List list = malloc(sizeof(struct list));
  list->size = 0;
  list->head = NULL;
  return list;
}

int isEmpty(List list){
  return list->head == NULL;
}

void addHead(List list, Item item){
  struct node *x = malloc(sizeof(struct node));
  x->next = list->head;
  x->item = item;
  list->head = x;                 //Aggiorna head
  list->size++;                   //Aggiorna size
}

Item removeHead(List list){
  Item app;
  if(isEmpty(list)==1){
    fprintf(stderr,"Lista vuota");
    return NULL;
  }
  struct node *temp = list->head;
  list->head = temp->next;
  app=temp->item;
  free(temp);
  list->size--;
  return app;
}

Item getHead(List list){
  if(isEmpty(list)==1){
    fprintf(stderr,"Lista vuota");
    return NULL;
  }
  return list->head->item;
}

int sizeList(List list){
  return list->size;
}

void printList(List list){
  struct node *p;
  for(p = list->head; p != NULL; p = p->next)
    outputItem(p->item);
  printf("\n");
}

void sortList(List list){
  struct node *p, *pos_minimo;
  for (p=list->head; p != NULL; p = p-> next){
    pos_minimo = minimo(p);
    swap(&(pos_minimo->item), &(p->item));
  }
}

struct node * minimo (struct node *p){
  struct node *i, *min = p;
  for (i = p; i != NULL; i = i->next){
    if ((cmpItem(min->item, i->item)) > 0)
      min = i;
  }
  return min;
}

Item searchList(List list, Item item, int *pos){
  struct node *p;
  *pos=0;
  for (p=list->head; p != NULL; p = p-> next){
      if(cmpItem(item,p->item) == 0)
        return p->item;
      ++*pos;
  }
  *pos=-1;
  return NULL;
}

Item removeListItem(List list, Item item){
  struct node *prev, *p;
  Item i;

  if(isEmpty(list)==1){
    fprintf(stderr,"Lista vuota");
    return NULL;
  }
  for (p=list->head; p != NULL; prev = p, p = p-> next){
    if(cmpItem(item,p->item) == 0) {
      if(p == list->head)
        return removeHead(list);
      else {
        prev->next = p->next;
        i = p->item;
        free(p);
        list->size--;
        return i;
      }
    }
  }
  return NULL;
}

Item removeListItemPos(List list, int pos){
  struct node *prev, *p;
  Item i;
  int j;
  if(isEmpty(list)==1){
    fprintf(stderr,"Lista vuota");
    return NULL;
  }
  j = 0;
  for (p=list->head; p != NULL; prev = p, p = p-> next, j++){
    if(j==pos) {
      if(pos == 0)
        return removeHead(list);
      else {
        prev->next = p->next;
        i = p->item;
        free(p);
        list->size--;
        return i;
      }
    }
  }
  return NULL;
}

int addListItem(List list, Item item, int pos){
  if (pos == 0){
    addHead(list, item);
    return 1;
  }
  if (pos > sizeList(list))
    return 0;
  struct node *p;
  int i;
  for (p = list -> head, i = 0; p; i++, p = p -> next){
    if (i == pos-1){
      struct node *new = malloc(sizeof(struct node));
      new -> next = p -> next;
      p -> next = new;
      new -> item = item;
      list -> size++;
      return 1;
    }
  }
  return 0;
}

int addListTail(List list, Item item){
  return addListItem(list, item, sizeList(list));
}

void reverseList(List list){
  struct node *prev=NULL, *p, *temp;
  for(p=list->head; p; prev=p, p=temp){
    temp= p->next;
    p->next=prev;
  }
  list->head=prev;
}

List cloneList(List list){
  List clone= newList();
  struct node *p;
  for (p = list -> head; p; p = p -> next){
    Item item=cloneItem(p->item);
    addListTail(clone,item);
  }
  return clone;
}
```