LISTE

Struttura dati:

1. Modo sintetico per organizzare i dati
2. Insieme di operatori o primitive per la manipolazione degli elementi. Sono una serie di azioni effettuabili sulla struttura dati tra cui, per esempio, la lettura e la modifica.

Array: struttura dati che memorizza elementi omogenei in *sequenza*, ad *accesso diretto*, e che ha *dimensione fissa*.

Accesso diretto: dato un indice i, si accede in modo diretto all'elemento in posizione i-esima senza scorrere quelli precedenti
Dimensione fissa: la dimensione non può cambiare
Operatori: inserimento/cancellazione/ricerca di un elemento; ci permettono di leggere o modificare l'array.

Strutture dati dinamiche

Vengono usate su un problema che opera su una sequenza di valori non nota a priori.

Limiti degli array:

- *Occupazione della memoria*: la dimensione dell'array deve essere definita nella parte dichiarativa del programma (o al momento dell'allocazione se dinamico). In caso non fosse nota la dimensione dell'array si potrebbe sovrastimarla ma, potrebbe capitare, che il vettore saturi oppure si sprechi una quantità di memoria notevole.
- *Velocità di esecuzione*: in caso di inserimento in posizione i-esima occorrerebbe shiftare in avanti tutti gli elementi uguali o successivi alla posizione i-esima. In caso, invece, di estrazione in posizione [..
- *Scarsa flessibilità*

La soluzione risiede nelle liste.

Lista

Sequenza a dimensione variabile di elementi omogenei ad accesso sequenziale (il C++ non ha il dato primitivo lista).
Caratteristiche delle liste:

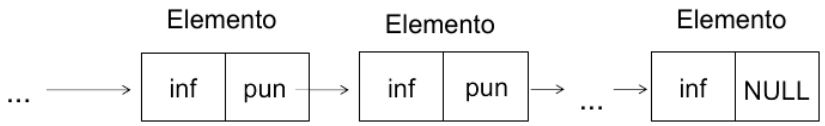
- *Dimensione variabile:* la dimensione può variare nel tempo
- *Accesso sequenziale*: per accedere a un elemento devo prima accedere a tutti gli elementi che lo precedono nella sequenza

Implementazione lista: avviene con l'allocazione dinamica e i puntatori.

Linked list:

- Ogni volta che dobbiamo aggiungere un elemento alla lista allochiamo un nuovo elemento
- Ogni volta che dobbiamo estrarre un elemento dalla lista, lo deallochiamo
- Gli elementi della lista non occupano posizioni contigue in memoria
- Usiamo i puntatori per collegare i vari elementi

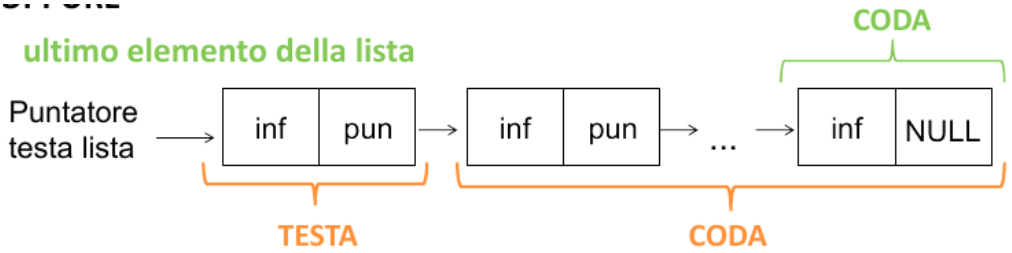
Ciascun elemento contiene un campo informazione (di qualsiasi tipo) ed un campo puntatore all'elemento successivo.
Mi accorgo che la lista è finita quando il piuntatore dell'ultimo



elemento della lista ha valore NULL (non fa riferimento a niente).

Implementazione

- *Testa (o head) della lista:* primo elemento della lista. È fondamentale in quanto l'accesso alla lista avviene attraverso il puntatore alla testa della lista (se perso ho perso la lista).
- *Coda (o tail):* resto della lista.



Elemento implementato attraverso una struct

```
struct elem {
int inf;      // o qualsiasi tipo semplice
elem* pun;    // definizione ricorsiva
};

elem *testa; // puntatore alla testa della
                lista
// oppure:
typedef elem* lista;
lista testa;
```

Nota bene: elem *testa non crea un oggetto dinamico ma solo il puntatore. Deve essere eseguita un'allocazione mediante l'operatore new (p = new elem).

Tipi di liste

- *Lista semplice o singolarmente concatenata (singly linked list):* ogni elemento contiene un unico puntatore che lo collega all'elemento successivo.
- *Lista doppia o doppiamente concatenata (doubly linked list)*: ogni elemento contiene due puntatori, uno all'elemento precedente e uno al successivo
- *Lista vuota (empty list):* lista senza elementi identificata da un puntatore alla testa della lista avente valore NULL

LISTE SEMPLICI

Caratteristiche:

Gli elementi occupano posizioni NON sequenziali
L'ordine è determinato da un puntatore in ogni elemento

Per determinare la fine della lista è necessario il segnale di fine lista *NULL*
Non si può risalire avendo un elemento al suo antecedente; si può solo avere il suo successivo.

Accesso alla lista:
Occorre avere la primitiva (come se fossero variabili) *head* e *tail* implementate come funzioni.

Il tipo restituito corrisponde al tipo di inf

```
int head(lista l){return l->inf;}
lista tail(lista l){return l->pun;}
```

Stampa di una lista:
Scandire una lista esistente fino alla fine (usando head e tail).

stampalista() prende in ingresso il puntatore alla testa della lista

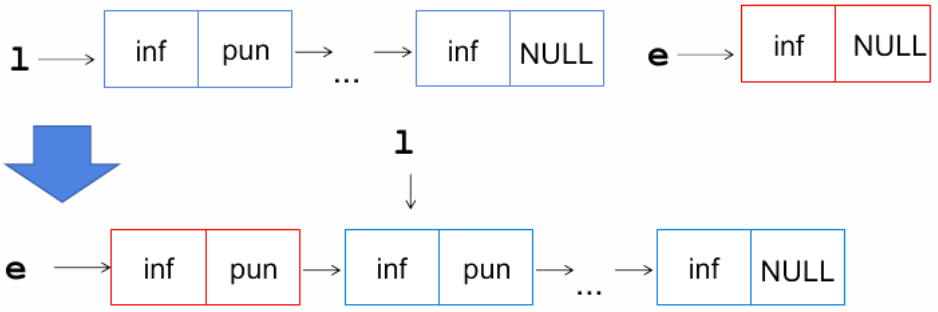
```
void stampalista(lista p)
{ while (p != NULL) {
  cout << head(p) << " " ; // stampa valore in testa
  p = tail(p);   // spostamento sulla coda di p
  }
  cout << endl ;
}
```

Inserimento di un elemento:

- *in testa* → semplice ed efficiente, gli elementi nella lista compariranno nella lista in ordine inverso rispetto all'ordine di inserimento.
- *In fondo (in coda)* → richiede di scorrere tutta la lista, gli elementi nella lista compariranno in ordine diretto rispetto all'ordine di inserimento.

Salvo eccezioni, l'inserimento avviene normalmente in testa.

lista insert_elem(lista l, elem* e)
Funzione che aggiunge **e** a **l** e restituisce la lista aggionata




```
lista insert_elem(lista l, elem* e){
      e->pun=l;
      return e;
}
```

La funzione richiede un puntatore elem* e all'elemento da aggiungere che deve essere allocato prima di essere passato alla funzione.

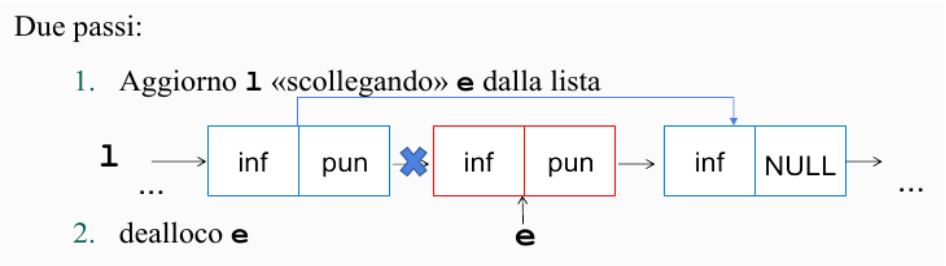
Esempio

```
 lista a;
elem* ele;
…
 ele=new elem;
 cin>>ele->inf;
 ele->pun = NULL;
 a=insert_elem(a,ele);
…
```

Cancellazione di un elemento: **`lista delete_elem(lista l, elem* e)`**

Funzione che cancella l'elemento **`e`** da **`l`** e restituisce la lista aggiornata

- Se *e* coincide con la testa della lista *l* è necessario modificare *l* affinché punto all'elemento puntato da *e*.
- Altrimenti è necessario aggiornare l'elemento che punta a *e* affinché punti all'elemento puntato da *e*.



Nell'istruzione *delete* e viene passato il valore memorizzato dentro e ossia l'indirizzo dell'oggetto da deallocare dalla memoria dinamica.

Si utilizza la variabile di appoggio L1 e non L perchè perderei il riferimento alla lista.

```
lista delete_elem(lista l, elem* e){
      if (l==e)
          l=tail(l);
      else{
         lista l1=l;
         while (tail(l1)!=e)
            l1=tail(l1);
         l1->pun=tail(e);}
      delete e;
      return l;
}
```

e è la testa della lista

Localizzo l'elemento che punta a e

Aggiorno l'elemento che punta a e

Cancellazione di tutta la lista

```
void eliminalista(lista &testa)
{
     while (testa != NULL)
           testa=delete_elem(testa,testa);
}
```

Ricerca di un elemento

Il tipo di valore cercato int v corrisponde al tipo del contenuto nel campo informativo di elem.

`elem* search(lista l, int v)`

Funzione che cerca nella lista **`l`** il valore **`v`** e restituisce il puntatore all'elemento che contiene la *prima occorrenza* di **`v`**, se esiste, **`NULL`**, altrimenti

```
elem* search(lista l, int v){
     while(l!=NULL)
          if(head(l)==v)
               return l;
          else
               l=tail(l);
     return NULL;}
```

Si scorre la lista l con le primitive head e tail fino a trovare l'elemento richiesto (se esite).

Copia di una lista

Esegue la copia di una lista.

1. Inizializzo la lista l a NULL
2. Per ogni valore in l1 genero un nuovo elemento curr e lo appendo alla lista l

```
lista copy(lista l1){
    lista l=NULL;
    elem* curr;
    elem* prev=NULL;          → Puntatore ausiliario: punta all'ultimo elemento inserito

    while(l1!=NULL){
        curr = new elem ;
        curr->inf = head(l1);
        curr->pun=NULL;
        if(prev==NULL) /* sto creando la testa */
            l=curr;
        else
            prev->pun=curr;   → Inserimento in fondo alla lista

        prev=curr;            → Aggiornamento di prev
        l1=tail(l1);
    }
    return l;
}
```