Lista

Una **lista** è una sequenza finita, anche vuota, di elementi dello stesso tipo. Nella lista uno stesso elemento può comparire più volte, in posizioni diverse. Gli elementi della lista, cui sono associate delle informazioni, sono definiti **atomi** o **nodi**.
Indichiamo la lista con la notazione:

$$l = \ < a_1, a_2, \ldots, a_n > \ con\, n \geq 0$$

A ciascun elemento di una lista viene associata una **posizione** $pos(i)$ e un **valore** $a(i)$.

Accesso

Si può accedere direttamente solo al primo elemento della sequenza; per accedere al generico elemento occorre scandire sequenzialmente gli elementi della lista che lo precedono.

Operazioni

Le operazioni consentite sono di **inserimento** e **rimozione** degli elementi; poiché la lista è una struttura a dimensione variabile, queste operazioni alterano la dimensione. La lista è dunque una **struttura** dati **dinamica**.

Specifica sintattica

Tipi: `lista, posizione, boolean, tipoelem`
Operatori:

- `crealista: () → lista`
- `listavuota: (lista) → boolean`
- `leggilista: (posizione, lista) → tipoelem`
- `scrivilista: (tipoelem, posizione, lista) → lista`
- `primolista: (lista) → posizione`
- `finelista: (posizione, lista) → boolean`
- `succlista: (posizione, lista) → posizione`
- `predlista: (posizione, lista) → posizione`
- `inslista: (tipoelem, posizione, lista) → lista`
- `canclista: (posizione, lista) → lista`

Specifica semantica

Tipi:

- `lista`: insieme delle sequenze $l = < a_1, a_2, \ldots, a_n >$ con $n \geq 0$, di elementi di tipo tipoelem dove l'elemento i-esimo ha valore $a(i)$ e posizione $pos(i)$
- `boolean`: insieme dei valori di verità

Operatori:

- `crealista = l'`
  - Post: $l' = \ < \ >$
- `listavuota(l) = b`
  - Post: $b = true$ se $l = \ < \ >$; $b = false$ altrimenti
- `leggilista(p, l) = a`
  - Pre: $p = pos(i)$ con $1 \leq i \leq n$
  - Post: $a = a(i)$
- `scrivilista(a, p, l) = l'`
  - Pre: $p = pos(i)$ con $1 \leq i \leq n$
  - Post: $l' = < a_1, a_2, \ldots, a_{i-1}, a, a_{i+1}, \ldots, a_n >$
- `primolista(l) = p`
  - Pre: listavuota(l) = false
  - Post: $b = true$ se $p = pos(1)$; $b = false$ altrimenti
- `finelista(p, l) = b`
  - Pre: $p = pos(i)$ con $1 \leq i \leq n + 1$
  - Post: $b = true$ se $p = pos(n + 1)$; $b = false$ altrimenti
- `succlista(p, l) = q`
  - Pre: $p = pos(i)$ con $1 \leq i \leq n$
  - Post: $q = pos(i + 1)$
- `predlista(p, l) = q`
  - Pre: $p = pos(i)$ con $2 \leq i \leq n$
  - Post: $q = pos(i - 1)$
- `inslista(a, p, l) = l'`

- Pre: $p = pos(i)$ con $1 \leq i \leq n + 1$
- Post: $l' = < a_1, a_2, \ldots, a_{i-1}, a, a_i, a_{i+1}, \ldots, a_n >$ se $1 \leq i \leq n$

  $l' = < a_1, a_2, \ldots, a_n, a >$ se $i = n + 1$

  $l' = < a >$ se $i = 1$ e $l = < >$

- `canclista(p, l) = l'`
  - Pre: $p = pos(i)$ con $1 \leq i \leq n$
  - Post: $l' = < a_1, a_2, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n >$

Dalla specifica semantica emerge che per accedere ad un elemento occorre conoscerne la posizione. L'unico operatore che dà per risultato la posizione, senza altre informazioni, è primolista. Per gli altri la posizione si ottiene conoscendo a priori la posizione dell'elemento precedente (o seguente) e applicando l'operazione succlista (o predlista). Dunque, per accedere ad un generico elemento occorre scandire la lista a partire dal primo elemento.

Eliminazione di duplicati

Ci poniamo il seguente problema:
Data una lista l, i cui elementi siano interi, eliminare da l gli elementi che sono duplicati.

```
epurazione(lista l)
      p = primolista(l)
      while not finelista(p, l) do
            q = succlista(p, l)
            while not finelista(q, l) do
                  if leggilista(q, l) == leggilista(p, l)
                        canclista(q, l)
                  q = succlista(q, l)
            p = succlista(p, l)
```

Realizzazione sequenziale con vettore

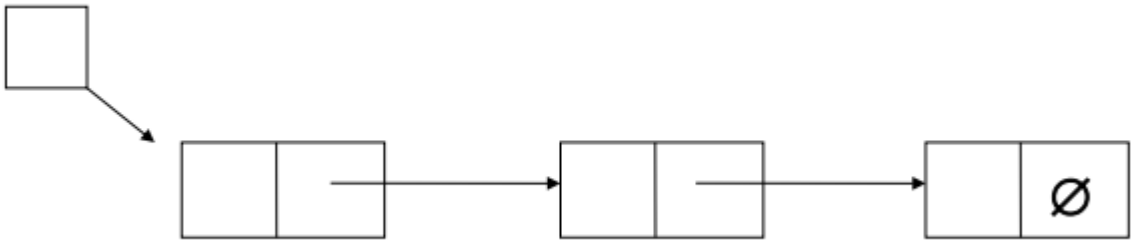
Una lista può essere rappresentata usando un **vettore** (array monodimensionale). Poiché il numero di elementi che compongono la lista può variare, si utilizza una variabile $primo$ per il valore dell'indice della componente del vettore in cui è memorizzato il primo elemento della lista. Si utilizza una variabile $lunghezza$ per indicare il numero di elementi di cui è composta la lista rappresentata. Questa rappresentazione consente di realizzare molto semplicemente alcune delle operazioni definite per la lista. Il vero problema riguarda l'inserzione e la rimozione di componenti. La banale inserzione di un nuovo elemento **causa lo spostamento** verso il basso degli altri elementi.

Rappresentazione collegata

L'idea fondamentale della rappresentazione collegata di una lista è quella di memorizzare i suoi elementi associando ad ognuno di essi una particolare informazione (detta **riferimento**)

che permetta di individuare la locazione in cui è memorizzato l'elemento successivo. Per visualizzare tale rappresentazione si usa una notazione grafica in cui:

- Gli **elementi** sono rappresentati mediante **nodi**;
- I **riferimenti** mediante **archi** che collegano nodi.



Si può notare che si usa un riferimento al primo elemento della lista e un simbolo speciale ∅ come riferimento associato all'ultimo nodo. Nel caso la lista sia vuota, tale simbolo compare direttamente nel riferimento iniziale.

Realizzazione con cursori

Viene utilizzato un **vettore** (array monodimensionale) per l'implementazione della lista, ma si riesce a superare, attraverso i riferimenti, il problema dell'aggiornamento (inserimento o cancellazione di un elemento). Si realizzano i riferimenti mediante cursori, cioè variabili interne o enumerative, il cui valore è interpretato come indice di un vettore. Si definisce un vettore $spazio$ che:

- Contiene **tutte le liste**, ognuna individuata da un proprio cursore iniziale;
- Contiene **tutte le celle libere**, organizzate in una lista, detta "listalibera".

Esempio

Disponiamo di tre diverse liste l, m, s:

$$l = \ < 7, 2 >$$
$$m = \ < 4, 9, 13 >$$
$$s = \ < 13, 4, 8, 13 >$$

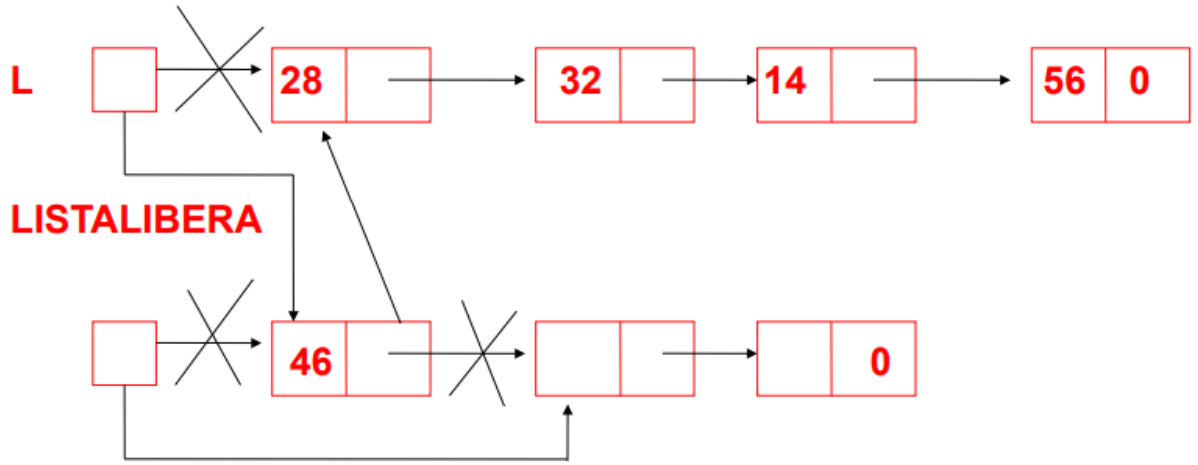
Possiamo usare un **unico vettore** $spazio$ per rappresentare le tre liste. La componente di spazio ha due campi:

1. Nel campo "**elemento**" è memorizzato il **contenuto** del nodo,
2. Nel campo "**successivo**" è memorizzato il **riferimento al prossimo** nodo.

La sequenza degli elementi che formano la lista è ricostruibile iniziando dalla componente dell'array corrispondente al valore di inizio, nel cui campo "elemento" è memorizzato il valore del primo elemento. Seguendo i cursori si trovano gli elementi successivi della lista. Infatti, si può definire la posizione $pos(i)$ dell'elemento i-esimo di l uguale al valore del cursore alla cella del vettore spazio che contiene l'elemento $(i-1)$-esimo, se $2 \leq i \leq n+1$, uguale a 0 se $i = 1$. Analogamente si definisce $pos(n+1)$ uguale al cursore alla cella di spazio che contiene l'elemento n-esimo se $n \geq 1$, o uguale a 0 altrimenti. La lista vuota si indica con $l = \ \emptyset$.

Per poter aggiornare una lista così realizzata sorge il problema di individuare la posizione di una componente libera nell'array. Si usa una `listalibera` memorizzata nello stesso

array per raccogliere in modo collegato le componenti libere dell'array, dalla quale si prendono (o rilasciano) posizioni libere quando sono necessarie.
La `listalibera` rappresenta un serbatoio da cui prelevare componenti libere dell'array e in cui riversare le componenti dell'array che non sono più utilizzate per la lista. Se si vuole inserire un nuovo elemento in testa alla lista, si fa uso della prima componente della lista libera.



È immediato verificare, prima di definire le operazioni di inslista e canclista, che con questa rappresentazione l'inserimento e l'eliminazione di un elemento **non richiedono lo spostamento** di altri elementi della lista, grazie alla `listalibera`. Rimangono i problemi connessi all'uso dell'array, cioè all'esigenza di definire una dimensione.

Realizzazione con puntatori

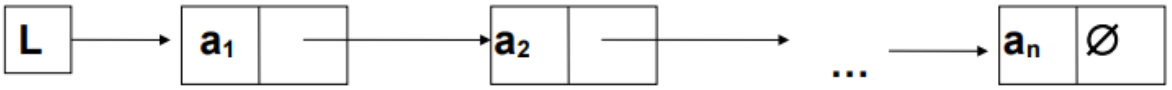
Un'altra possibile realizzazione di una lista è quella mediante l'uso congiunto del tipo **puntatore** e del tipo **record**. Le operazioni usualmente disponibili su una variabile di tipo puntatore p sono:

- L'accesso alla locazione il cui indirizzo è memorizzato in p;
- La richiesta di una nuova locazione di memoria e la memorizzazione dell'indirizzo in p (new);
- Il rilascio della locazione di memoria il cui indirizzo è memorizzato in p (delete).

Rappresentazione collegata realizzata mediante puntatori

Una possibile realizzazione è quella di una **lista monodirezionale semplificata**. Vi è una struttura di n elementi o "**celle**", tale che l'i-esima cella contiene l'i-esimo elemento della lista e l'indirizzo della cella che contiene l'elemento successivo.

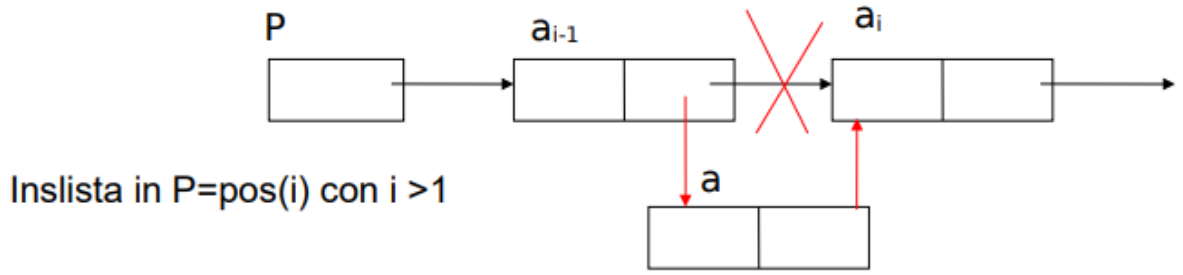
- La prima cella è indirizzata da una variabile di tipo puntatore;
- L'ultima cella punta ad un valore convenzionale nil;
- Gli indirizzi sono noti alla macchina ma non al programmatore;
- La posizione $pos(i)$ è uguale al valore del puntatore alla cella che contiene l'i-esimo elemento $1 \leq i \leq n$;
- Per numero di elementi diverso da zero, $pos(i)$ è il puntatore alla cella contenente l'i -esimo elemento, mentre quando la lista è vuota, $L = null$.



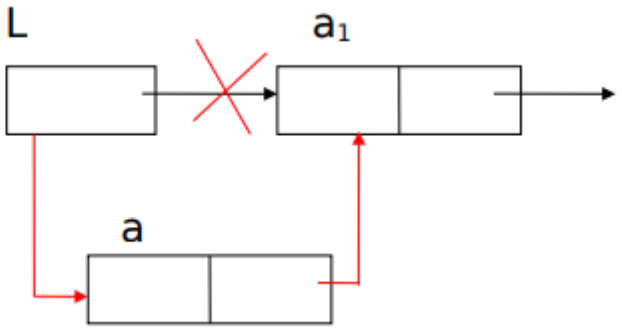
Una possibile dichiarazione di tipo sarebbe la seguente:

- **Posizione**: tipo puntatore a cella
- **Cella**: tipo strutturato con componenti
    - **Elemento** di tipo tipoelem
    - **Successivo** di tipo posizione
- **Lista**: alias per il tipo posizione

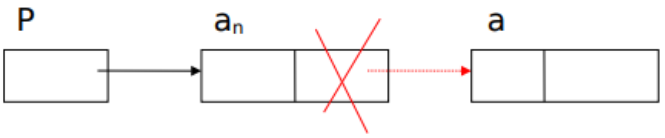
L'aggiornamento dei puntatori avviene nel seguente modo:



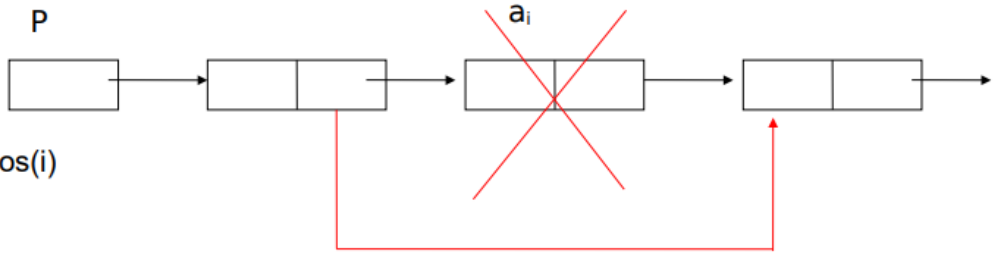
Inslista in P=pos(i) con i >1



Inslista in P=pos(1)



Inslista in P=pos(n+1)



Canclista P=pos(i)

Variante della rappresentazione collegata: a doppi puntatori o simmetrica

Ogni elemento contiene, oltre al riferimento al nodo successivo, anche il riferimento al precedente. La lista può essere “espansa” con una cella in più per la **realizzazione circolare**.
Con questa rappresentazione si ha il vantaggio di:

- Poter scandire la lista in **entrambe le direzioni**;
- Poter individuare facilmente l’elemento che **precede**;
- Poter realizzare le operazioni di **inserimento** senza dover usare variabili aggiuntive.

Esercizi

Ricerca in una lista lineare ordinata

$Progettare\ e\ implementare\ un\ algoritmo\ per\ ricercare\ in\ una\ lista\ lineare\ e\ ordinata\ per\ chiave\ alfabetic$

Una lista lineare ordinata è un particolare tipo di lista in cui l’ordine sequenziale degli elementi è legato ad una relazione d’ordine definita sugli elementi. Pertanto, se

$$l = < a_1, a_2, ..., a_n >$$

Allora

$$a_1 \leq a_2 \leq ... \leq a_n$$

Quando si devono effettuare operazioni di inserimento e di cancellazione di elementi in una lista lineare ordinata è necessario stabilire prima la posizione in cui va fatta la variazione. Consideriamo una lista di nomi messi in ordine alfabetico nella quale intendiamo inserire un nuovo nome, ad esempio

Posizione	**Nome**
1	Anna
2	Antonio
3	Carlo
4	Davide
5	Elena
6	Filippo
7	Giulio

Per prima cosa è necessario **trovare la posizione** in cui il nuovo elemento va inserito. La struttura sequenziale della lista impone che si effettui una scansione sia che si voglia inserire un nome, sia che si voglia cancellarlo. Dunque è necessario scandire la lista finchè non si trovi un nome che segue il nome dato in ordine alfabetico (per l’inserimento) oppure non si trovi un nome uguale al nome dato (per la cancellazione).

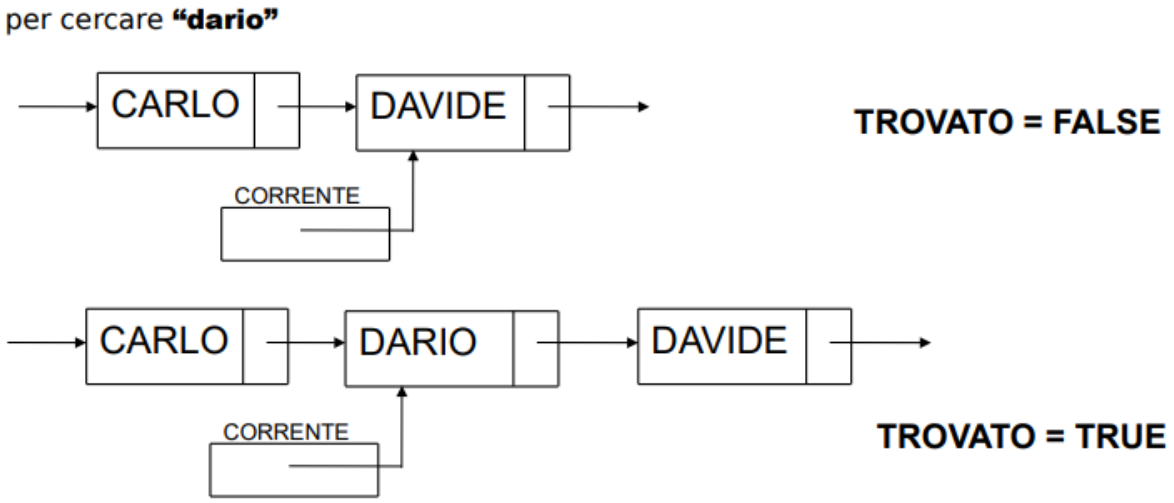
La parte centrale dell'algoritmo di ricerca sarà:

Mentre il nome cercato segue nell'ordine alfabetico il nome corrente di lista, passa al nome successivo nella lista.

Nel condurre una ricerca in una lista va seguito l'ordine logico e non quello fisico. Se la struttura è realizzata con puntatori

- **Cella**: tipo strutturato con componenti
  - **Nome**: di tipo tiponome
  - **Successivo**: di tipo puntatore a cella
- **Lista**: puntatore a cella
- **Posizione**: puntatore a cella

Allora useremo l'informazione sulla connessione logica data da successivo per procedere lungo la lista.
Se il nodo corrente punta all'elemento corrente allora basterà effettuare l'assegnazione corrente = succlista(corrente, l) per scorrere la catena finché non si trovi la posizione in cui il nome va inserito (in caso di inserimento) o è presente (in caso di ricerca).
Alla procedura chiamante possiamo restituire una variabile booleana trovato che aiuti a sapere se l'elemento ricercato c'è o non c'è.
Può essere utile restituire il valore del puntatore corrente che punterà al primo elemento della lista come nome in ordine alfabetico "superiore" al nome cercato o all'elemento col medesimo nome di quello cercato.



Potrebbe essere utile restituire anche il valore del puntatore memorizzato nell'elemento contenente il nome che cerchiamo, in tal caso va modificato il meccanismo di aggiornamento dei puntatori:

- `precedente = corrente`
- `corrente = succlista(corrente, l)`

Se l'elemento cercato è l'ultimo elemento nella lista, la ricerca termina quando si trova finelista(l) = true (oppure quando il puntatore ha valore nil). Nel caso in cui il nome cercato sia il primo nell'elenco, va restituita la posizione attraverso primolista(l).

definisci il nome cercato e l'inizio lista
inizializza precedente e corrente a inizio lista

poni trovato a falso
mentre la ricerca continua e non è finita la lista, esegui:

a. se il nome cercato precede alfabeticamente il nome corrente
    i. interrompi la ricerca
b. altrimenti
    i. poni precedente a corrente
    ii. aggiorna corrente puntando all'elemento successivo

stabilisci se il nome cercato è trovato
restituisci precedente, corrente e trovato

Fusione di liste ordinate

Date due liste ordinate, produrre una terza lista dalla fusione delle prime due e che rispetta l'ordinament

mentre non è finita lista1 e non è finita lista2
    confronta i due elementi correnti di lista1 e lista2
    memorizza l'elemento minore in lista3
    aggiorna l'elemento corrente della lista interessata

```
fusione(Lista1 di tipo Lista, Lista2 di tipo Lista, Lista3 di tipo
Lista per riferimento)
crealista(Lista3)
p1 = primolista(Lista1)
p2 = primolista(Lista2)
p3 = primolista(Lista3)
while (not finelista(p1, Lista1) and not finelista(p2, Lista2)) do
     elem1 = leggilista(p1, Lista1)
     elem2 = leggilista(p2, Lista2)
     if elem1 < elem2 then
          inslista(elem1, p3, Lista3)
          p1 = succlista(p1, Lista1)
     else
          inslista(elem2, p3, Lista3)
          p2 = succlista(p2, Lista2)
     p3 = succlista(p3, Lista3)
while not finelista(p1, Lista1) do
     inslista(leggilista(p1, Lista1), p3, Lista3)
     p1 = succlista(p1, Lista1)
     p3 = succlista(p3, Lista3)
while not fineslista(p2, Lista2) do
     inslista(leggilista(p2, Lista2), p3, Lista3)
     p2 = succlista(p2, Lista2)
     p3 = succlista(p3, Lista3)
```

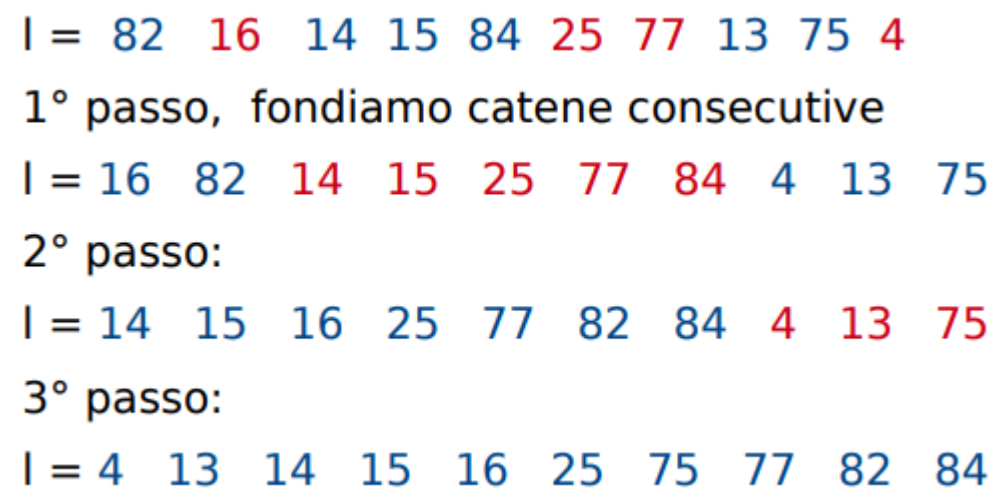
Ordinamento di una lista

Data una lista, ordinare gli elementi in ordine crescente rispetto ad una relazione d'ordine $\geq$.

Utilizzeremo a tale scopo il metodo noto come **ordinamento naturale**, o **natural merge sort**.
Data una lista $l = a_1 a_2 \ldots a_n$, diremo che una sottosequenza a_i, a_{i+1}, …, a_k costituisce una catena (o run) se accade che:

$$a_{i-1} > a_i$$
$$a_j \leq a_{j+1} \; \forall j = i, i+1, \ldots, k-1$$
$$a_k > a_{k+1}$$

Una lista è quindi una **sequenza di catene**.
Una proprietà interessante è che la fusione di due sequenze di n catene produce una singola sequenza di n catene. Così, fondendo le catene che compongono una lista si ottiene una nuova lista con un numero dimezzato di catene. Ripetendo questa operazione ad ogni passo, dopo al più $log_2(n)$ passi la lista **sarà completamente ordinata**:



Per determinare le sequenze di catene da fondere si può pensare ad una **fase di distribuzione** in cui le catene sono distribuite alternativamente a due liste ausiliarie. Al termine di questa fase le due liste conterranno ognuna $n/2$ catene originarie della lista iniziale.
L'algoritmo di ordinamento alterna fasi di distribuzione a **fasi di fusione**, fino a quando si ottiene una lista con un'unica catena.

```
OrdinamentoNaturale(L di tipo lista per riferimento)
repeat
      crealista(A)
      crealista(B)
      distribuisci(L, A, B)
      numero_catene = 0
      crealista(L)
      merge(A, B, L, numero_catene)
until numero_catene = 1
```

```
distribuisci(l di tipo lista, a di tipo lista per riferimento, b di
tipo lista per riferimento)
pl = primolista(l)
```

```
pa = primolista(a)
pb = primolista(b)
repeat
     copiacatena(pl, l, pa, a)
     if not finelista(pl, l)
          copiacatena(pl, l, pb, b)
until finelista(pl, l)
```

merge(A di tipo lista, B di tipo lista, L di tipo lista per riferimento, numero_catene int)

```
pa = primolista(a)
pb = primolista(b)
pl = primolista(L)
while not finelista(pa, A) and not finelista(pb, B) do
     fondiCatena(pa, A, pb, B, pl, L)
     numero_catene = numero_catene + 1
while not finelista(pa, A) do
     copiaCatena(pa, A, pl, L)
     numero_catene = numero_catene + 1
while not finelista(pb, B) do
     copiaCatena(pb, B, pl, L)
     numero_catene = numero_catene + 1
```

fondiCatena(pa di tipo posizione per riferimento; A, B di tipo lista;
pb, pa, pl di tipo posizione per riferimento;
L di tipo lista per riferimento)

```
finecatena = false
repeat
     if leggilista(pa, A) < leggilista(pb, B) then
          copia(pa, A, pl, L, finecatena)
          if finecatena then
               copiaCatena(pb, B, pl, L)
     else
          copia(pb, B, pl, L, finecatena)
          if finecatena then
               copiaCatena(pa, A, pl, L)
until finecatena
```

copiaCatena(ph, pk posizione per riferimento; H, K lista)

```
finecatena = false
repeat
     copia(ph, H, pk, K, finecatena)
until finecatena
```

copia(px, pl posizione per riferimento; X lista; L lista per riferimento; finecatena boolean rif)

```
elemento = leggilista(px, X)
```

```
inslista(elemento, pl, L)
px = succlista(px, X)
pl = succlista(pl, L)
if finelista(px, X) then
     finecatena = true
else
     finecatena = (elemento > leggilista(px, X))
```