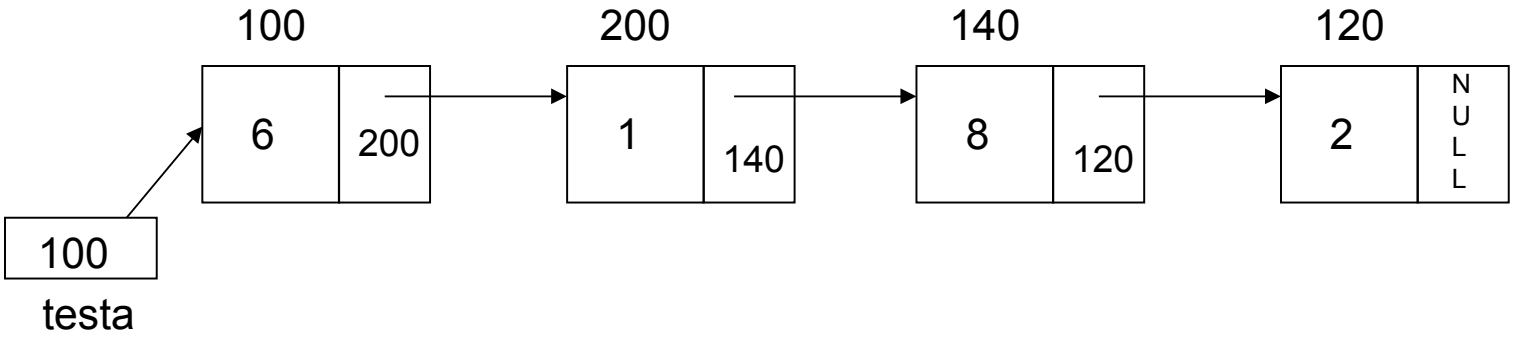
Rappresentazione collegata mediante puntatori di una lista

Rappresentazione collegata di una lista

- Gli elementi di una lista possono essere rappresentati mediante delle struct contenenti, oltre al dato associato all'elemento, un un campo che contiene l'indirizzo dell'elemento successivo della lista.
- Da un punto di vista grafico la lista L=(6,1,8,2) può essere rappresentata nel seguente modo:



dove la variabile **testa** contiene l'indirizzo del primo elemento della lista

- Se la lista è vuota la variabile testa ha valore NULL
- L'ultimo elemento della lista ha il campo relativo al successivo elemento a NULL.

Rappresentazione collegata di una lista

- Una lista di interi all'interno di un programma C può essere rappresentata definendo la struct atomo come:

```
struct atomo
{ int dato;
  struct atomo *prossimo;
};
```

e all'interno del programma una variabile testa_lista nel seguente modo:

```
struct atomo *testa_lista=NULL;
```

Funzioni lista_vuota e testa

- La funzione lista_vuota restitusce 1 se il puntatore alla testa della lista ha valore 1, altrimenti 0.

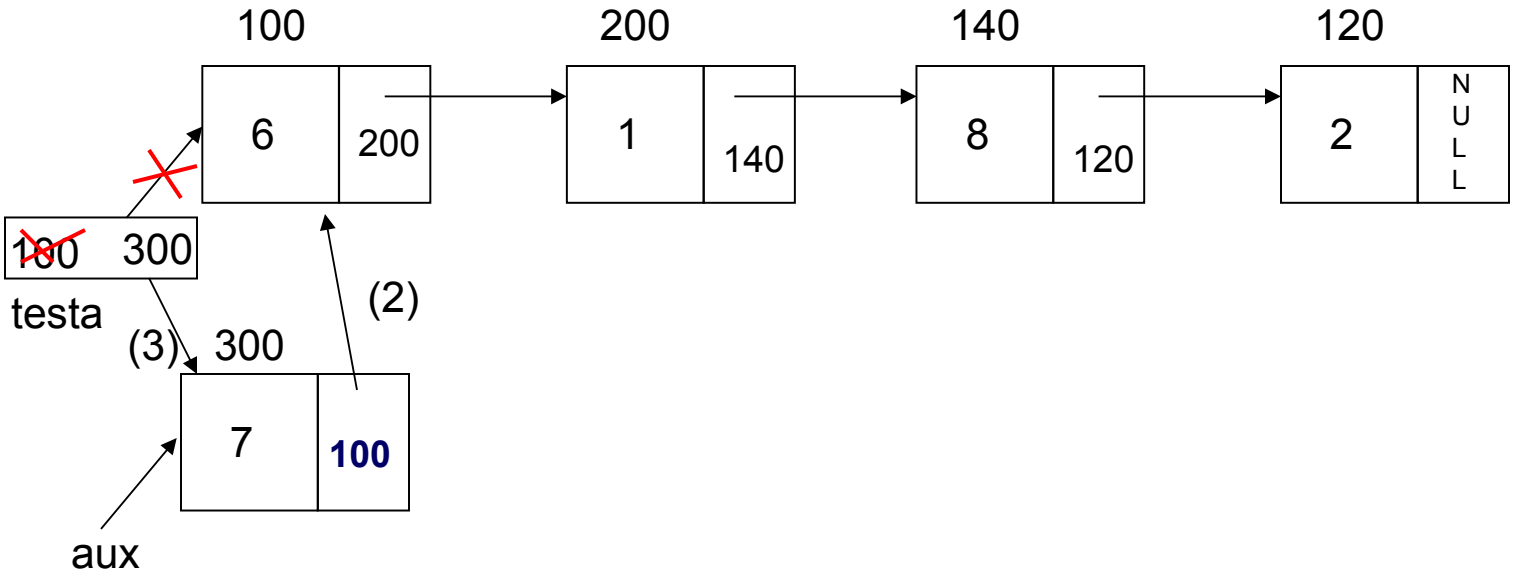
```
int lista_vuota( struct atomo *tl)
{
  if(tl==NULL) return 1;
  else return 0;
}
```

- La funzione testa restituisce il puntatore al primo elemento della lista.

```
struct atomo * testa (struct atomo *tl)
{
   return tl;
}
```

Inserimento in testa

Prof. G. Ascia



1) Viene creato il nuovo elemento (mediante la malloc()) il cui indirizzo è assegnato ad **aux;** `(aux=malloc(sizeof(struct atomo);)`
2) Viene assegnato ad aux->prossimo l'indirizzo del vecchio primo elemento (testa); `(aux->prossimo=testa)`
3) Viene assegnato a testa l'indirizzo del nuovo primo elemento. `(testa=aux)`

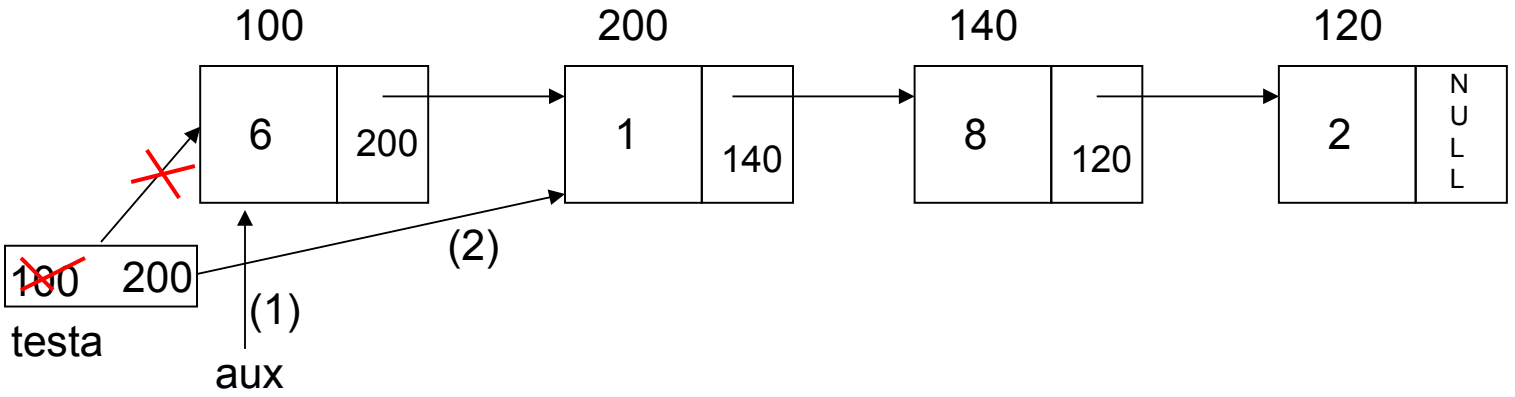
Inserimento in testa

- La funzione in_testa inserisce l'elemento e in testa alla lista.
- Come primo parametro viene passato l'indirizzo della variabile testa_lista (che è il puntatore al primo elemento della lista), poiché il suo valore viene modificato all'interno della funzione.

```
void in_testa (struct atomo **ptl, int e)
{ struct atomo *aux;

 aux=malloc(sizeof(struct atomo));
 if(aux)  {    aux->dato=e;
               aux->prossimo=*ptl;
               *ptl=aux;
          }
 else printf("Memoria esaurita\n");
}
```

Eliminazione dalla testa della lista

Prof. G. Ascia



1) Viene copiato nella variabile aux l'indirizzo dell'elemento da eliminare;

```
aux=testa;
```

2) Viene assegnato ad testa l'indirizzo dell'elemento successivo al primo;

```
testa=testa->prossimo;
```

3) Viene liberata lamemoria occupata dall'elemento puntato da aux.

```
free(aux);
```

Eliminazione dalla testa della lista

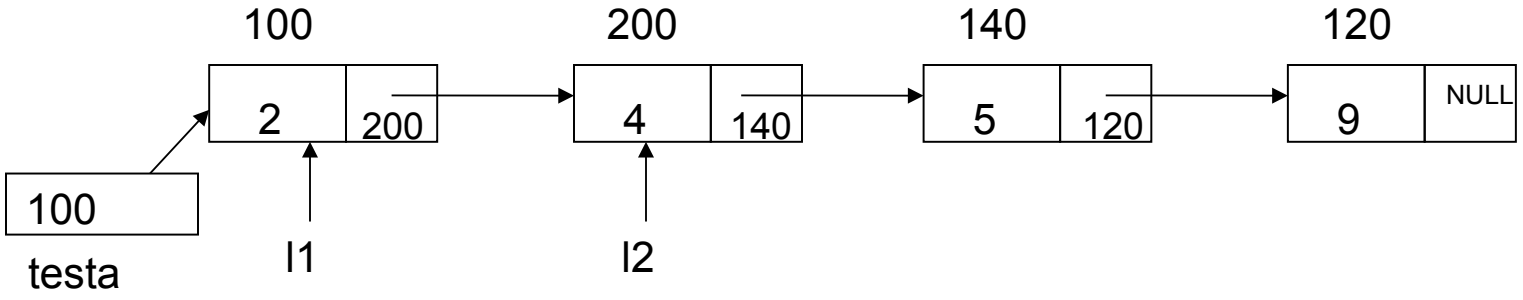

```
void da_testa (struct atomo **ptl)
{ struct atomo *aux;

 if(!lista_vuota(*ptl))
  {aux=*ptl;
   *ptl=aux->prossimo;
   free(aux);
  }
 else printf("La lista e' gia' vuota\n");
}
```

Inserimento ordinato in una lista

Prof. G. Ascia

Nell'inserimento ordinato bisogna distinguere i seguenti casi:

1. lista vuota: l'inserimento ordinato coincide con l'inserimento in testa;
2. L'elemento da inserire è più piccolo del primo elemento: l'inserimento è un inserimento in testa;
3. L'elemento da inserire è più grande del primo elemento: è necessario trovare la posizione all'interno della lista dove inserire il nuovo elemento;



Nel caso 3, volendo inserire un elemento nella lista in modo ordinato, possiamo usare due puntatori: l1 e l2.

Inizialmente l1 punta alla testa della lista e l2 al secondo elemento

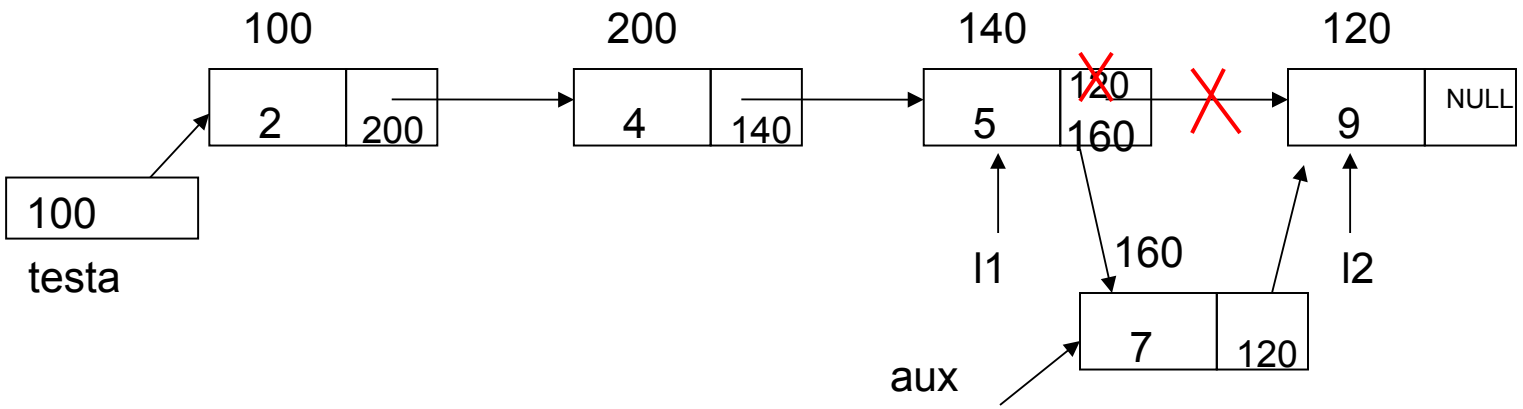
Inserimento ordinato in una lista

- La posizione corretta dove inserire il nuovo elemento è quella in corrispondenza della quale **e** è minore del campo dato puntato da l2 ovvero se `e < l2->dato;`
- Fino a quando tale condizione non è soddisfatta e l2 è diverso da NULL bisogna fare avanzare l1 e l2.

- La ricerca della posizione corretta nel caso 3 pertanto può essere codificato nel seguente modo:

```
l1=*ptl;
l2=l1->prossimo;
 while(l2)
   if(e< l2->dato)
         break;
   else {       l1=l1->prossimo;
         l2=l2->prossimo;
         }
```

Inserimento ordinato in una lista



- Una volta trovata la posizione dove effettuare l'inserimento, il nuovo elemento, il cui indirizzo è nella variabile aux, viene inserito:
  - ponendo il suo campo prossimo a l2 ovvero

    ```
    aux->prossimo=l2;
    ```
  - assegnando al campo prossimo di l1 il valore di aux ovvero:

    ```
    l1->prossimo=aux,
    ```

Inserimento ordinato in una lista


```
void inserimento_ordinato(struct atomo **ptl, int e)
{ struct atomo *aux,*l1,*l2;

  if(lista_vuota(*ptl) || e<(*ptl)->dato)
     in_testa(ptl,e);
  else
  { aux=malloc(sizeof(struct atomo));
    if(aux)
    { aux->dato=e;

      l1=*ptl;
      l2=l1->prossimo;
      while(l2)
        if(e< l2->dato)
             break;
        else { l1=l1->prossimo;
               l2=l2->prossimo;
             }
      aux->prossimo=l2;
      l1->prossimo=aux;
    }
    else printf("Memoria esaurita");
  }
}
```

Inserimento ordinato in una lista

- Anziché usare l1 e l2 possiamo usare una sola variabile l usando l al posto di l1 e l->prossimo al posto di l2.

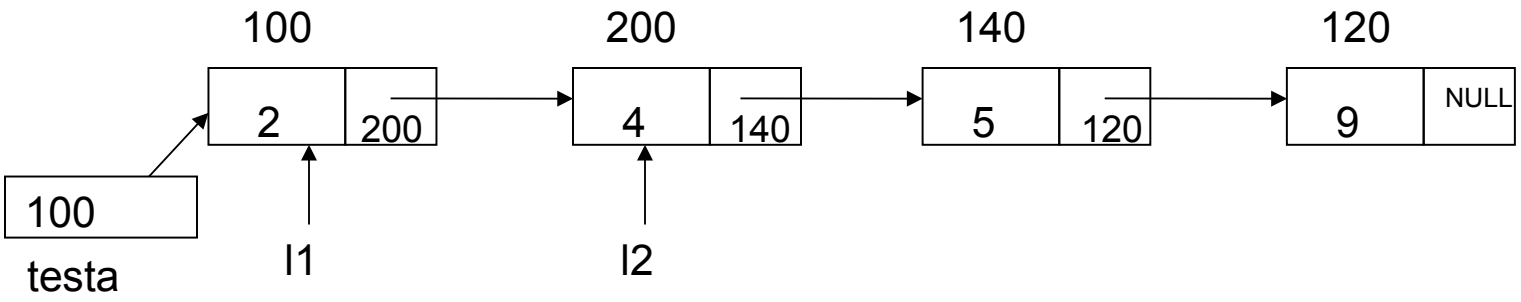
```
void inserimento_ordinato(struct atomo **ptl, int e)
{ struct atomo *aux,*l;

  if(lista_vuota(*ptl) || e<(*ptl)->dato)
     in_testa(ptl,e);
  else
  { aux=malloc(sizeof(struct atomo));
    if(aux)
    { aux->dato=e;
      l=*ptl;
      while(l->prossimo)
        if(e< l->prossimo->dato)
             break;
        else l=l->prossimo;

      aux->prossimo=l->prossimo;
      l->prossimo=aux;
    }
    else printf("Memoria esaurita");
  }
}
```

Cancellazione da una lista

- Possiamo distinguere tre casi:

1. Lista vuota: nulla viene fatto;
2. Cancellazione del primo elemento: coincide con la cancellazione dalla testa di una lista;
3. Cancellazione di un elemento in posizione successiva: viene fatta la scansione della lista alla ricerca della posizione dell'elemento da eliminare.

- La ricerca del caso 3. può essere fatta usando due puntatori: l2 che punta all'elemento da cercare nella lista e l1 che punta all'elemento che precede quello puntato da l2.



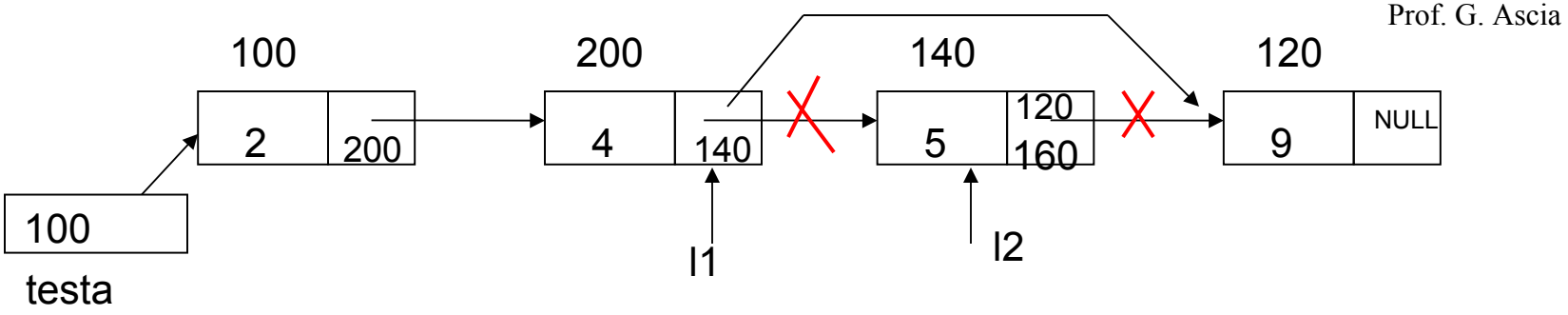
Cancellazione da una lista

- La posizione dell'elemento da cancellare è quella in corrispondenza della quale `e == l2->dato;`
- Fino a quando tale condizione non è soddisfatta e l2 è diverso da NULL bisogna fare avanzare l1 e l2.

- La ricerca della posizione corretta nel caso 3 pertanto può essere codificato nel seguente modo:

```
l1=*ptl;
l2=l1->prossimo;
 while(l2)
   if(e== l2->dato)
         break;
   else {l1=l1->prossimo;
         l2=l2->prossimo;
         }
```

Cancellazione da una lista



- Una volta trovata la posizione dell'elemento da eliminare, è necessario aggiornare il campo l1->pros a l2->pros ed eliminare l'elemento puntato da l2 ovvero:

```
l->prossimo=l2->prossimo;
free(l2);
```

Cancellazione di un elemento da una lista


```
void elimina (struct atomo **ptl, int e)
{ struct atomo *l1,*l2;

  if(lista_vuota(*ptl))   printf("Lista vuota\n");
  else if((*ptl)->dato==e) da_testa(ptl);
  else
     {  l1=*ptl;
        l2=l1->prossimo;
        while(l2)
         if(l2->dato==e)
           { l->prossimo=l2->prossimo;
             free(l2);
             printf("\nEliminato\n");
             return;
           }
         else {l1=l1->prossimo;
               l2=l2->prossimo;
              }
        printf("\nNon e' presente\n");
    }
}
```

Cancellazione di un elemento da una lista

- Anziché usare l1 e l2 possiamo usare una sola variabile l usando l al posto di l1 e l->prossimo al posto di l2

```
void elimina (struct atomo **ptl, int e)
{ struct atomo *aux,*l;

  if(lista_vuota(*ptl))   printf("Lista vuota\n");
  else if((*ptl)->dato==e) da_testa(ptl);
  else
     {  l=*ptl;
        while(l->prossimo)
         if(l->prossimo->dato==e)
           { aux=l->prossimo;
             l->prossimo=aux->prossimo;
             free(aux);
             printf("\nEliminato\n");
             return;
           }
         else l=l->prossimo;
        printf("\nNon e' presente\n");
     }
}
```