Rappresentazione di liste di interi

Per rappresentare un elemento della lista usiamo la struttura

```
struct element {
    int info;  // valore dell'elemento
    struct element *next;   // prossimo elemento
};
```


Rappresentazione di liste di interi

Per rappresentare un elemento della lista usiamo la struttura

```
struct element {
    int info;  // valore dell'elemento
    struct element *next;   // prossimo elemento
};
```

Conviene porre la definizione

```
typedef  struct element  element;
```

che definisce il nuovo tipo `element` come sinonimo del tipo `struct element`.

Rappresentazione di liste di interi

Per rappresentare un elemento della lista usiamo la struttura

```
struct element {
    int info;  // valore dell'elemento
    struct element *next;   // prossimo elemento
};
```

Conviene porre la definizione

```
typedef  struct element  element;
```

che definisce il nuovo tipo `element` come sinonimo del tipo `struct element`.

La memoria occupata da un elemento di tipo `struct element` è

```
sizeof(struct element) =  sizeof(int) + sizeof(int*)
```

(si ricordi che l'occupazione in memoria di una variabile di tipo `T*` è la stessa per tutti i tipi `T`).

Sia E un elemento della lista.

- Il campo `info` di E contiene il valore dell'elemento rappresentato da E (un intero).

Sia E un elemento della lista.

- Il campo `info` di E contiene il valore dell'elemento rappresentato da E (un intero).
- Per quanto riguarda il campo `next`:
  - Se E è l'ultimo elemento della lista, il campo `next` di E vale `NULL`;

Sia E un elemento della lista.

- Il campo `info` di E contiene il valore dell'elemento rappresentato da E (un intero).
- Per quanto riguarda il campo `next`:
  - Se E è l'ultimo elemento della lista, il campo `next` di E vale `NULL`;
  - altrimenti, il campo `next` di E è l'indirizzo del prossimo elemento della lista.

Sia E un elemento della lista.

- Il campo `info` di E contiene il valore dell'elemento rappresentato da E (un intero).
- Per quanto riguarda il campo `next`:
  - Se E è l'ultimo elemento della lista, il campo `next` di E vale `NULL`;
  - altrimenti, il campo `next` di E è l'indirizzo del prossimo elemento della lista.

Occorre controllare che, al termine di una operazione di alto livello (es., inserimento di un elemento) la proprietà sia rispettata.

Sia E un elemento della lista.

- Il campo `info` di E contiene il valore dell'elemento rappresentato da E (un intero).
- Per quanto riguarda il campo `next`:
  - Se E è l'ultimo elemento della lista, il campo `next` di E vale `NULL`;
  - altrimenti, il campo `next` di E è l'indirizzo del prossimo elemento della lista.

Occorre controllare che, al termine di una operazione di alto livello (es., inserimento di un elemento) la proprietà sia rispettata.

Per rappresentare una lista occorre definire una variabile `head` di tipo `element*` (*testa* della lista).

Sia E un elemento della lista.

- Il campo `info` di E contiene il valore dell'elemento rappresentato da E (un intero).
- Per quanto riguarda il campo `next`:
  - Se E è l'ultimo elemento della lista, il campo `next` di E vale `NULL`;
  - altrimenti, il campo `next` di E è l'indirizzo del prossimo elemento della lista.

Occorre controllare che, al termine di una operazione di alto livello (es., inserimento di un elemento) la proprietà sia rispettata.

Per rappresentare una lista occorre definire una variabile `head` di tipo `element*` (*testa* della lista).

- Se `head` vale `NULL`, `head` rappresenta la lista vuota.
- Se `head` è diverso da `NULL`, `head` è l'indirizzo al primo elemento (testa) della lista.

Operazioni su liste

- `element* newList()`

  Restituisce la lista vuota.

Operazioni su liste

- `element* newList()`

  Restituisce la lista vuota.

- `element* insert(int n, element *h)`

  Inserisce un nuovo elemento contenente `n` in testa alla lista `h`. Restituisce la testa della nuova lista.
- `element* find(int n, element *h)`

  Cerca l'intero `n` nella lista `h`. Se `n` è nella lista restituisce l'indirizzo dell'elemento che lo contiene, altrimenti restituisce `NULL`.

Operazioni su liste

- `element* newList()`

  Restituisce la lista vuota.

- `element* insert(int n, element *h)`

  Inserisce un nuovo elemento contenente `n` in testa alla lista `h`. Restituisce la testa della nuova lista.
- `element* find(int n, element *h)`

  Cerca l'intero `n` nella lista `h`. Se `n` è nella lista restituisce l'indirizzo dell'elemento che lo contiene, altrimenti restituisce `NULL`.

- `element* delete(element *x, element *h)`

  Cancella l'elemento di indirizzo `x` nella lista `h` e restituisce la lista ottenuta. Si assume che `x` sia l'indirizzo di un elemento della lista `h`.

Operazioni su liste

- `element* newList()`

  Restituisce la lista vuota.

- `element* insert(int n, element *h)`

  Inserisce un nuovo elemento contenente `n` in testa alla lista `h`. Restituisce la testa della nuova lista.

- `element* find(int n, element *h)`

  Cerca l'intero `n` nella lista `h`. Se `n` è nella lista restituisce l'indirizzo dell'elemento che lo contiene, altrimenti restituisce `NULL`.

- `element* delete(element *x, element *h)`

  Cancella l'elemento di indirizzo `x` nella lista `h` e restituisce la lista ottenuta. Si assume che `x` sia l'indirizzo di un elemento della lista `h`.

- `void print(element *h)`

  Stampa gli elementi contenuti nella lista `h`.

Creazione di una lista vuota

La funzione `newList()` restituisce la lista vuota.

In questo caso occorre restituire `NULL`, ma in rappresentazioni diverse (ad esempio, le liste con sentinella) la lista vuota può avere una struttura più complessa.

Creazione di una lista vuota

La funzione `newList()` restituisce la lista vuota.

In questo caso occorre restituire `NULL`, ma in rappresentazioni diverse (ad esempio, le liste con sentinella) la lista vuota può avere una struttura più complessa.

```
element  *newList(){
    return NULL;
}
```


Inserimento di un elemento

La funzione `insert()` inserisce un nuovo elemento il cui valore è `n` in testa alla lista `h`.

Restituisce la lista ottenuta.

Inserimento di un elemento

La funzione `insert()` inserisce un nuovo elemento il cui valore è `n` in testa alla lista `h`.

Restituisce la lista ottenuta.

```
element *insert(int n, element *h){
  element *new = malloc(sizeof(element)); // (A)
  new->info = n;
  new->next = h;  // (B)
  return new;
}
```


Con l'istruzione

```
element *new = malloc(sizeof(element)); // (A)
```

viene dichiarata una variabile `new` di tipo `element*` a cui viene assegnato come valore iniziale l'indirizzo dell'area di memoria allocata da `malloc()`.

Con l'istruzione

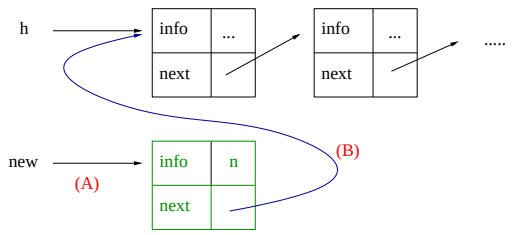
```
element *new = malloc(sizeof(element)); // (A)
```

viene dichiarata una variabile `new` di tipo `element*` a cui viene assegnato come valore iniziale l'indirizzo dell'area di memoria allocata da `malloc()`.

Dopo

```
  new->info = n;
  new->next = h;  // (B)
```

si ha:



Poiché il primo elemento della nuova lista è quello di indirizzo `new`, la funzione deve restituire il valore di `new`. Verificare che la funzione è corretta anche nel caso in cui il valore di `h` è `NULL` (ossia, `h` rappresenta la lista vuota).

Poiché il primo elemento della nuova lista è quello di indirizzo `new`, la funzione deve restituire il valore di `new`. Verificare che la funzione è corretta anche nel caso in cui il valore di `h` è `NULL` (ossia, `h` rappresenta la lista vuota).

La complessità dell'inserimento in testa è $O(1)$ in quanto il numero di operazioni da compiere per inserire un nuovo elemento non dipende dalla lunghezza della lista.

Poiché il primo elemento della nuova lista è quello di indirizzo `new`, la funzione deve restituire il valore di `new`. Verificare che la funzione è corretta anche nel caso in cui il valore di `h` è `NULL` (ossia, `h` rappresenta la lista vuota).

La complessità dell'inserimento in testa è $O(1)$ in quanto il numero di operazioni da compiere per inserire un nuovo elemento non dipende dalla lunghezza della lista.

Esempio 1

Per costruire la lista che rappresenta l'insieme $\{10, 20, 30\}$:

```
int main(){
    element *head;
    head = newList();
    head = insert(10, head);
    head = insert(20, head);
    head = insert(30, head);
    return 0;
}
```

Si noti che dopo ogni chiamata il valore della variabile `head` cambia.

Nota

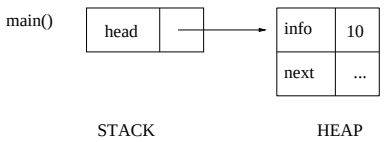
Ricordarsi che all'inizio la variabile `head` va inizializzata facendo

```
head = newList();  // OPPURE: head = NULL;
```

Infatti, dopo la prima istruzione di inserimento

```
head = insert(10, head);
```

si ha:



Il valore di `head->next` è uguale al valore di `head` passato come primo parametro (valore di `head` prima della chiamata). Se tale valore è diverso da `NULL`, la lista ottenuta non è corretta (non vengono rispettate le convenzioni sulla rappresentazione di liste) e si hanno errori in esecuzione.

Il valore di `head->next` è uguale al valore di `head` passato come primo parametro (valore di `head` prima della chiamata). Se tale valore è diverso da `NULL`, la lista ottenuta non è corretta (non vengono rispettate le convenzioni sulla rappresentazione di liste) e si hanno errori in esecuzione.

Se si vuole trattare il caso in cui `malloc()` fallisce restituendo `NULL`, subito dopo la chiamata a `malloc()` occorre aggiungere un'istruzione del tipo

```
...
new = malloc(...);
if(new == NULL) { // malloc() e' fallita
    printf("Errore malloc() \n"); // stampa messaggio
    exit(-1); /* il programma termina restituendo -1
                 all'ambiente chiamante */
  }
```

In C non esistono meccanismi per gestire le eccezioni.

Altra soluzione

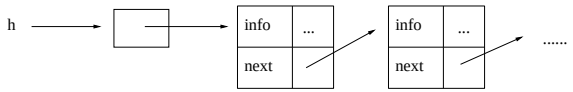
L'inserimento in una lista modifica la testa della lista. Nella soluzione vista, l'indirizzo della lista ottenuta è restituito dalla funzione.

Alternativamente, si può passare la testa della lista "per indirizzo". In questo caso l'intestazione della funzione per l'inserimento è

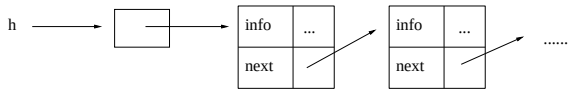
```
void insert1(int n, element **h)
```

dove `h` è l'indirizzo della locazione di memoria contenente l'indirizzo al primo elemento della lista.

Dopo il passaggio dei parametri si ha:

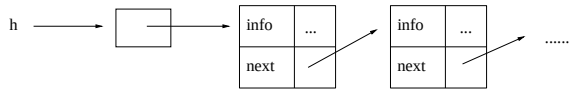


Dopo il passaggio dei parametri si ha:



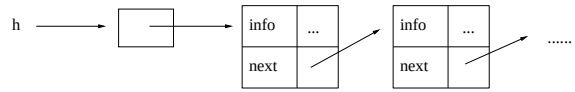
- Il valore di `h` (che ha tipo `element**`) è l'indirizzo della locazione di memoria in cui è contenuto l'indirizzo del primo elemento della lista.

Dopo il passaggio dei parametri si ha:



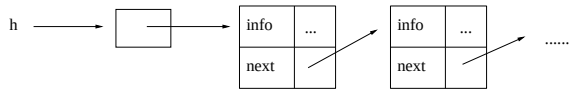
- Il valore di `h` (che ha tipo `element**`) è l'indirizzo della locazione di memoria in cui è contenuto l'indirizzo del primo elemento della lista.
- Il valore di `*h` (che ha tipo `element*`) è l'indirizzo del primo elemento della lista (quindi, `*h` denota la testa della lista).

Dopo il passaggio dei parametri si ha:



- Il valore di `h` (che ha tipo `element**`) è l'indirizzo della locazione di memoria in cui è contenuto l'indirizzo del primo elemento della lista.
- Il valore di `*h` (che ha tipo `element*`) è l'indirizzo del primo elemento della lista (quindi, `*h` denota la testa della lista).
- L'espressione `**h` (che ha tipo `element`) denota il primo elemento della lista (è quindi una struttura).

Dopo il passaggio dei parametri si ha:



- Il valore di `h` (che ha tipo `element**`) è l'indirizzo della locazione di memoria in cui è contenuto l'indirizzo del primo elemento della lista.
- Il valore di `*h` (che ha tipo `element*`) è l'indirizzo del primo elemento della lista (quindi, `*h` denota la testa della lista).
- L'espressione `**h` (che ha tipo `element`) denota il primo elemento della lista (è quindi una struttura).

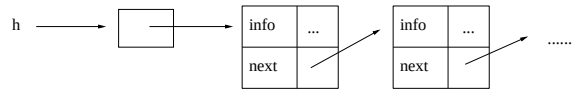
Le seguenti espressioni sono sintatticamente corrette

`(**h).info` `(**h).next` `(*h)->info` `(*h)->next`

Le seguenti espressioni invece contengono errori sintattici

`(*h).info` `h->info` `*h->info` `**h.info`

Dopo il passaggio dei parametri si ha:



- Il valore di `h` (che ha tipo `element**`) è l'indirizzo della locazione di memoria in cui è contenuto l'indirizzo del primo elemento della lista.
- Il valore di `*h` (che ha tipo `element*`) è l'indirizzo del primo elemento della lista (quindi, `*h` denota la testa della lista).
- L'espressione `**h` (che ha tipo `element`) denota il primo elemento della lista (è quindi una struttura).

Le seguenti espressioni sono sintatticamente corrette

`(**h).info` `(**h).next` `(*h)->info` `(*h)->next`

Le seguenti espressioni invece contengono errori sintattici

`(*h).info` `h->info` `*h->info` `**h.info`

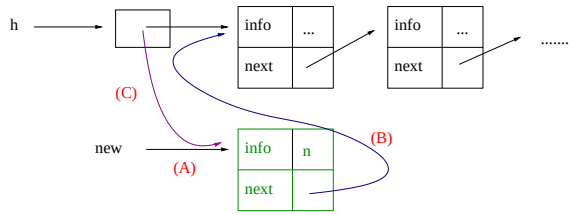
Le ultime due espressioni sono equivalenti rispettivamente a

`*(h->info)` `*(*(h.info))`

```
void insert1(int n, element **h){
 element *new;
 new = malloc(sizeof(element)); // (A)
 new->info = n;
 new->next = *h; // (B)
 *h = new;  //  (C)
}
```



```
void insert1(int n, element **h){
 element *new;
 new = malloc(sizeof(element)); // (A)
 new->info = n;
 new->next = *h; // (B)
 *h = new;  //  (C)
}
```



Esempio 2

Per costruire l'insieme $\{10, 20, 30\}$:

```
int main(){
 element *head;
 head = newList(); // head = NULL
 insert1(10, &head);
 insert1(20, &head);
 insert1(30, &head);
 return 0;
}
```


Esempio 2

Per costruire l'insieme $\{10, 20, 30\}$:

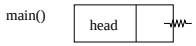
```
int main(){
 element *head;
 head = newList(); // head = NULL
 insert1(10, &head);
 insert1(20, &head);
 insert1(30, &head);
 return 0;
}
```

Esecuzione Esempio 2

(1) Dopo l'istruzione

```
head = newList();
```

la configurazione della memoria è:



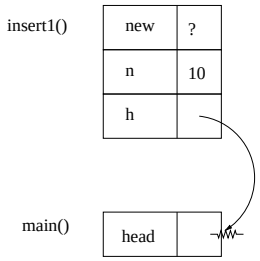
STACK

(2) Viene ora eseguita l'istruzione

```
insert1(10, &head);
```

Viene allocato il record di attivazione di `insert1()`.

Dopo il passaggio dei parametri si ha:



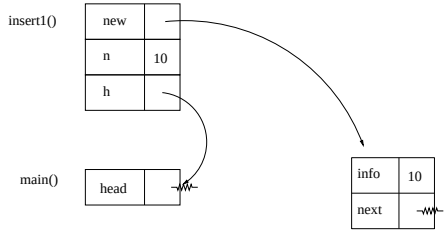
STACK

(3) Il controllo passa a `insert1()`.

Dopo aver eseguito le istruzioni

```
new = malloc(sizeof(element)); // nuovo elemento
new->info = n;
new->next = *h;
```

si ha:



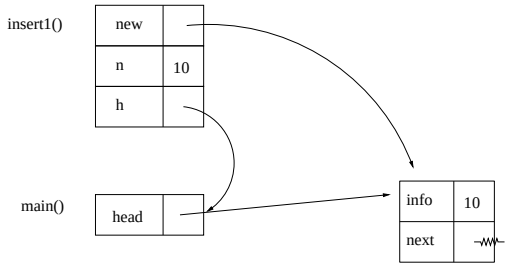
STACK HEAP

Si noti che `*h` vale `NULL`.

(4) Eseguendo ora

```
*h = new;
```

alla variabile `head` viene assegnato il valore di `new` (indirizzo dell'elemento di tipo `element` contenente 10).

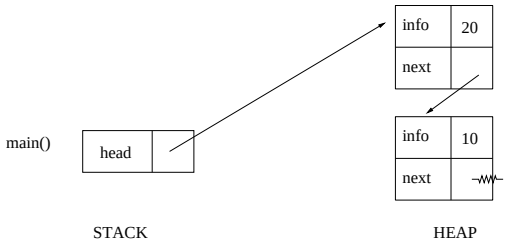


STACK HEAP

(5) Supponiamo di aver eseguito le chiamate

```
insert1(10, &head);
insert1(20, &head);
```

In memoria si ha:

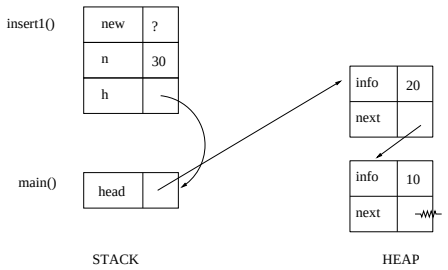


STACK HEAP

(6) Eseguendo ora la chiamata

```
insert1(30, &head);
```

dopo il passaggio dei parametri si ha:

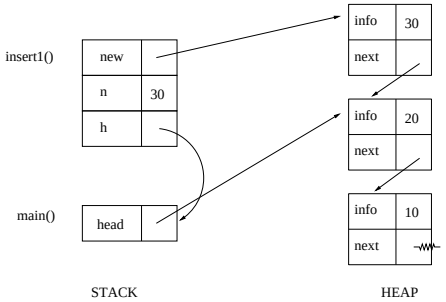


(7) Viene ora eseguita `insert1()`.

Dopo le istruzioni

```
new = malloc(sizeof(element)); // nuovo elemento
new->info = n;
new->next = *h;
```

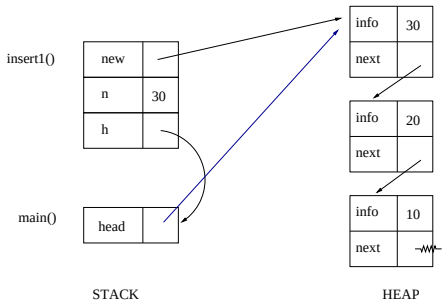
si ha:



(8) Dopo l'istruzione

```
*h = new;
```

la variabile `head` è la testa della nuova lista.



Quando il record di attivazione di `insert1()` è tolto, lista `head` rappresenta l'insieme $\{10, 20, 30\}$.

Ricerca di un elemento

La funzione `find()` cerca nella lista `h` l'intero `n`.

Se `n` è nella lista restituisce l'indirizzo dell'elemento che lo contiene, altrimenti restituisce `NULL`.

Ricerca di un elemento

La funzione `find()` cerca nella lista `h` l'intero `n`.

Se `n` è nella lista restituisce l'indirizzo dell'elemento che lo contiene, altrimenti restituisce `NULL`.

```
element* find(int n, element *h){
    for( ; h != NULL; h = h->next)
        if(h->info == n)
            return h;
//   n non e' nella lista
    return NULL;
}
```


All'interno del ciclo, la variabile `h` viene usata come "indice" per scorrere gli elementi della lista.

Al termine di ogni iterazione, con l'assegnamento

```
h = h->next
```

`h` è l'indirizzo del prossimo elemento della lista (controllare cosa succede quando `h` è l'ultimo elemento della lista).

Si ricordi che `h` è una variabile *locale* a `find()` e quando il suo valore è modificato il valore del corrispondente parametro attuale non cambia.

All'interno del ciclo, la variabile `h` viene usata come "indice" per scorrere gli elementi della lista.

Al termine di ogni iterazione, con l'assegnamento

```
h = h->next
```

`h` è l'indirizzo del prossimo elemento della lista (controllare cosa succede quando `h` è l'ultimo elemento della lista).

Si ricordi che `h` è una variabile *locale* a `find()` e quando il suo valore è modificato il valore del corrispondente parametro attuale non cambia.

Quando l'espressione

```
h->info == n
```

è verificata, l'elemento di indirizzo `h` contiene `n`.

Complessità

La complessità della funzione è proporzionale al numero di elementi della lista a cui occorre accedere. Supponiamo che la lista contenga n elementi.

Complessità

La complessità della funzione è proporzionale al numero di elementi della lista a cui occorre accedere. Supponiamo che la lista contenga n elementi.

- Il caso migliore si ha quando l'elemento cercato è il primo della lista (complessità $O(1)$).

Complessità

La complessità della funzione è proporzionale al numero di elementi della lista a cui occorre accedere. Supponiamo che la lista contenga n elementi.

- Il caso migliore si ha quando l'elemento cercato è il primo della lista (complessità $O(1)$).
- il caso peggiore quando l'elemento cercato è l'ultimo della lista o non è nella lista (complessità $\Theta(n)$).

Caso medio

Occorre fare la media delle complessità di *tutti* i casi possibili. Indichiamo con C_k, con $k = 1, \ldots, n$, la complessità nel caso in cui l'elemento cercato sia il k-esimo della lista e con C' la complessità nel caso in cui l'elemento non sia nella lista. Si ha:

$$C_M = \frac{\sum_{k=1}^{n} C_k + C'}{n+1}$$

Caso medio

Occorre fare la media delle complessità di *tutti* i casi possibili. Indichiamo con C_k, con $k = 1, \ldots, n$, la complessità nel caso in cui l'elemento cercato sia il k-esimo della lista e con C' la complessità nel caso in cui l'elemento non sia nella lista. Si ha:

$$C_M = \frac{\sum_{k=1}^{n} C_k + C'}{n+1}$$

Poiché $C_k = k$ per ogni $1 \leq k \leq n$ e $C' = n$, si ha:

$$C_M = \frac{\sum_{k=1}^{n} k + n}{n+1} = \frac{\frac{n.(n+1)}{2} + n}{n+1} = \frac{n^2 + 3n}{2 \cdot (n+1)} = \Theta(n)$$

La complessità del caso medio è la stessa del caso peggiore.

Caso medio

Occorre fare la media delle complessità di *tutti* i casi possibili. Indichiamo con C_k, con $k = 1, \ldots, n$, la complessità nel caso in cui l'elemento cercato sia il k-esimo della lista e con C' la complessità nel caso in cui l'elemento non sia nella lista. Si ha:

$$C_M = \frac{\sum_{k=1}^{n} C_k + C'}{n+1}$$

Poiché $C_k = k$ per ogni $1 \leq k \leq n$ e $C' = n$, si ha:

$$C_M = \frac{\sum_{k=1}^{n} k + n}{n+1} = \frac{\frac{n.(n+1)}{2} + n}{n+1} = \frac{n^2 + 3n}{2 \cdot (n+1)} = \Theta(n)$$

La complessità del caso medio è la stessa del caso peggiore.

Quindi:

> Se sono richieste frequenti operazioni di *ricerca*, implementare un insieme dinamico con una lista NON è una soluzione efficiente.

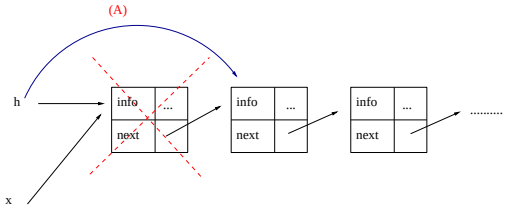
Cancellazione di un elemento

```
element *delete(element *x, element *h)
```

cancella l'elemento di indirizzo `x` nella lista `h` e restituisce la lista ottenuta. Si assume che `x` sia l'indirizzo di un elemento della lista `h`.

1. L'elemento da cancellare è il primo della lista

   Prima di eliminare l'elemento `x`, a `h` deve essere assegnato l'indirizzo del secondo elemento della lista, che diventerà il primo elemento della lista ottenuta (valore restituito dalla funzione).



   Il codice è corretto anche nel caso in cui `h` contiene un solo elemento (infatti viene restituito `NULL`).

Cancellazione di un elemento

```
element *delete(element *x, element *h)
```

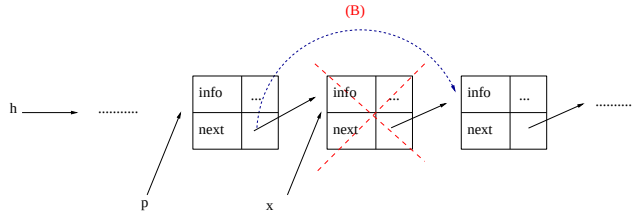
cancella l'elemento di indirizzo `x` nella lista `h` e restituisce la lista ottenuta. Si assume che `x` sia l'indirizzo di un elemento della lista `h`.

2. L'elemento da cancellare non è il primo della lista

   Occorre determinare l'indirizzo `p` dell'elemento che precede quello da cancellare. Per fare questo, `p` scorre gli elementi della lista a partire dal primo fino a quando

   ```
   p->next == x
   ```

   è verificato. Prima di eliminare l'elemento `x`, occorre collegare l'elemento `p` con quello successivo a `x` La testa della nuova lista è ancora `h`.



```
element *delete(element *x, element *h){
   element *p;
   if(x == h){  // l'elemento da cancellare e' il primo
           h = h->next;  // (A)
           free(x);
           return h;
       }
   else{  // l'elemento da cancellare non  e' il primo
           for(p = h;  p->next != x; p = p->next);
           // p e' l'elemento che precede x
           p->next = x->next; // (B)
           free(x);
           return h; // h non e' cambiata
       }
}
```


Le funzioni `find()` e `delete()` vanno usate insieme, in quanto il primo argomento di `delete()` deve essere l'indirizzo dell'elemento da cancellare.

Ad esempio, per cancellare (se c' è) l'elemento 10 dalla lista `head`:

```
element *head, *p;
...
if( (p = find(10, head)) != NULL )
    head = delete(p, head);
```

Quando la condizione in `if` è valutata, a `p` viene assegnato il valore restituito dalla chiamata a `find()` e la cancellazione viene eseguita solo se il valore di `p` è diverso da `NULL`.

Complessità

La parte più dispendiosa è la ricerca dell'elemento precedente a quello da eliminare. Infatti, diversamente da quanto avviene per gli array, non c'è alcun modo per accedere direttamente a uno specifico elemento della lista (a meno che il suo indirizzo non sia noto). Quindi, se l'elemento da cancellare è il k-esimo della lista, occorrono almeno $k-1$ accessi per trovare l'indirizzo dell'elemento che lo precede.

Complessità

La parte più dispendiosa è la ricerca dell'elemento precedente a quello da eliminare. Infatti, diversamente da quanto avviene per gli array, non c'è alcun modo per accedere direttamente a uno specifico elemento della lista (a meno che il suo indirizzo non sia noto). Quindi, se l'elemento da cancellare è il k-esimo della lista, occorrono almeno $k-1$ accessi per trovare l'indirizzo dell'elemento che lo precede.

Ragionando come nel caso dell'inserimento, si dimostra facilmente che la complessità media della cancellazione di un elemento in una lista di n elementi è $\Theta(n)$.

Complessità

La parte più dispendiosa è la ricerca dell'elemento precedente a quello da eliminare. Infatti, diversamente da quanto avviene per gli array, non c'è alcun modo per accedere direttamente a uno specifico elemento della lista (a meno che il suo indirizzo non sia noto). Quindi, se l'elemento da cancellare è il k-esimo della lista, occorrono almeno $k-1$ accessi per trovare l'indirizzo dell'elemento che lo precede.

Ragionando come nel caso dell'inserimento, si dimostra facilmente che la complessità media della cancellazione di un elemento in una lista di n elementi è $\Theta(n)$.

Esercizio

Scrivere una funzione per la cancellazione avente prototipo

```
void delete1(element *x, element **h)
```

in cui il significato dei parametri è analogo a quello di `insert1()`.

Cancellazione e stampa di tutti gli elementi

La funzione `destroy()` cancella tutti gli elementi della lista `h`.

```
void destroy(element *h){
   element *x = h;
   while(x != NULL){
        h = x->next;
        free(x);
        x = h;
       }
}
```

All'interno del ciclo, `x` è l'elemento da cancellare, `h` è l'elemento successivo a `x`. Attenzione all'ordine delle istruzioni.

Cancellazione e stampa di tutti gli elementi

La funzione `destroy()` cancella tutti gli elementi della lista `h`.

```
void destroy(element *h){
   element *x = h;
   while(x != NULL){
        h = x->next;
        free(x);
        x = h;
       }
}
```

All'interno del ciclo, `x` è l'elemento da cancellare, `h` è l'elemento successivo a `x`. Attenzione all'ordine delle istruzioni.

La funzione `print()` stampa tutti gli elementi della lista `h`.

```
void print(element *h){
    for( ; h !=  NULL; h = h->next)
        printf("%d ", h->info);
   }
```


Rappresentazione di insiemi dinamici mediante liste

Un insieme dinamico può essere rappresentato da una lista (si veda il testo dell'esercizio assegnato in laboratorio).

- L'ordine in cui gli elementi dell'insieme si trovano nella lista è irrilevante.
- Ogni elemento dell'insieme deve comparire *una sola volta* nella lista, pertanto prima di inserire un nuovo elemento occorre controllare se è già nella lista.

Rappresentazione di insiemi dinamici mediante liste

Un insieme dinamico può essere rappresentato da una lista (si veda il testo dell'esercizio assegnato in laboratorio).

- L'ordine in cui gli elementi dell'insieme si trovano nella lista è irrilevante.
- Ogni elemento dell'insieme deve comparire *una sola volta* nella lista, pertanto prima di inserire un nuovo elemento occorre controllare se è già nella lista.

La complessità in spazio è ottimale. Infatti, per rappresentare un insieme di n elementi lo spazio richiesto è $O(n)$, in quanto ogni elemento dell'insieme occupa la stessa quantità di spazio; non si ha spreco di memoria e non è necessario conoscere in anticipo la cardinalità massima dell'insieme.

Per quanto riguarda la complessità in tempo, supponiamo che l'insieme contenga n elementi.

Per quanto riguarda la complessità in tempo, supponiamo che l'insieme contenga n elementi.

- La complessità dell'inserimento nel caso medio è $\Theta(n)$. Infatti l'inserimento ha complessità $O(1)$, ma occorre fare una ricerca per controllare se l'elemento è già nella lista.

Per quanto riguarda la complessità in tempo, supponiamo che l'insieme contenga n elementi.

- La complessità dell'inserimento nel caso medio è $\Theta(n)$. Infatti l'inserimento ha complessità $O(1)$, ma occorre fare una ricerca per controllare se l'elemento è già nella lista.
- Anche la eliminazione di un elemento richiede mediamente $\Theta(n)$ (anche in questo caso nella valutazione della complessità il contributo maggiore è dato dal tempo di ricerca).

Per quanto riguarda la complessità in tempo, supponiamo che l'insieme contenga n elementi.

- La complessità dell'inserimento nel caso medio è $\Theta(n)$. Infatti l'inserimento ha complessità $O(1)$, ma occorre fare una ricerca per controllare se l'elemento è già nella lista.
- Anche la eliminazione di un elemento richiede mediamente $\Theta(n)$ (anche in questo caso nella valutazione della complessità il contributo maggiore è dato dal tempo di ricerca).
- La complessità della stampa e della cancellazione dell'insieme è $\Theta(n)$ (e ovviamente non può essere migliorata).

Formato istruzioni in input

Istruzione in input	Operazione
`+ n`	Se n non appartiena all'insieme lo inserisce, altrimenti non compie alcuna operazione
`- n`	Se n appartiene all'insieme lo elimina, altrimenti non compie alcuna operazione
`? n`	Stampa un messaggio che dichiara se n appartiene all'insieme
`c`	Stampa il numero di elementi dell'insieme
`p`	Stampa gli elementi dell'insieme
`o`	Stampa gli elementi dell'insieme in ordine crescente
`d`	Cancella tutti gli elementi dell'insieme
`f`	Termina l'esecuzione

Struttura `main()`

```
int main(){
  element *head; //  testa della lista
  int count;  // contatore elementi nella lista
  ...
   head = newList(); //  crea lista vuota
  // OPPURE:  head = NULL;
  count = 0;
  while((c=getchar()) != 'f'){
     switch(c){
      case '+':  // aggiunta di un elemento
        ...
        break;
      case '-':  // eliminazione di un elemento
        ...
        break;
     // ALTRI CASI
       } // end switch
  } // end while
  destroy(head);
  return 0;
}
```


Alcuni casi

```
...
case '+':  // aggiunta di un elemento
   scanf("%d", &n);
   if(find(n,head) == NULL){// n non e' nella lista
       head = insert(n, head);
       count++;
   }
break;
case '-':  // eliminazione di un elemento
   scanf("%d", &n);
   if((p=find(n, head)) != NULL){
      head = delete(p, head);
      count--;
   }
break;
```

Alcuni casi

```
case 'o':   // stampa in ordine crescente
   a = listToArray(head, count);
   qksort(a,count);  // ordina array a[] con quicksort
   printArray(a,count);
   free(a); // l'array a non serve piu'
   break;
case 'd':  // distrugge la lista
   destroy(head);
   head = newList();  //  crea lista vuota
   count = 0;
  break;
...
```


Liste bidirezionali

Si può migliorare l'efficienza della cancellazione di un elemento facendo in modo che, dato l'elemento da cancellare, sia possibile risalire in modo immediato all'elemento che lo precede.

Liste bidirezionali

Si può migliorare l'efficienza della cancellazione di un elemento facendo in modo che, dato l'elemento da cancellare, sia possibile risalire in modo immediato all'elemento che lo precede.

Questo può essere ottenuto con le liste bidirezionali, così chiamate perché ogni elemento della lista contiene, oltre all'indirizzo dell'elemento successivo (campo `next`), anche l'indirizzo dell'elemento che lo precede (campo `prev`).

Liste bidirezionali

Si può migliorare l'efficienza della cancellazione di un elemento facendo in modo che, dato l'elemento da cancellare, sia possibile risalire in modo immediato all'elemento che lo precede.

Questo può essere ottenuto con le liste bidirezionali, così chiamate perché ogni elemento della lista contiene, oltre all'indirizzo dell'elemento successivo (campo `next`), anche l'indirizzo dell'elemento che lo precede (campo `prev`).

Il campo `prev` del primo elemento della lista vale `NULL`

Il campo `next` dell'ultimo elemento della lista vale `NULL`

Rispetto alle liste semplici:

Rispetto alle liste semplici:

- Vantaggi:

  La cancellazione di un elemento di cui si conosce l'indirizzo avviene in tempo costante.
- Svantaggi:

  Ogni elemento della lista occupa più spazio (un puntatore in più); nelle operazioni di inserimento e cancellazione occorre aggiornare i campi `prev`.

Rispetto alle liste semplici:

- Vantaggi:

  La cancellazione di un elemento di cui si conosce l'indirizzo avviene in tempo costante.
- Svantaggi:

  Ogni elemento della lista occupa più spazio (un puntatore in più); nelle operazioni di inserimento e cancellazione occorre aggiornare i campi `prev`.

Per rappresentare un elemento della lista definiamo la struttura

```
struct elementBid{
  int info;
  struct elementBid* next; // indirizzo el. successivo
  struct elementBid* prev; // indirizzo el. precedente
};
```

e il tipo

```
typedef struct elementBid   elementBid;
```


Operazioni su liste bidirezionali

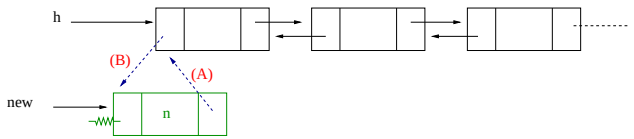
Occorre modificare le funzioni di inserimento e cancellazione di un elemento (le altre operazioni sono sostanzialmente uguali).

Operazioni su liste bidirezionali

Occorre modificare le funzioni di inserimento e cancellazione di un elemento (le altre operazioni sono sostanzialmente uguali).

Inserimento

La funzione `insert()` inserisce un elemento il cui valore è `n` in testa alla lista `h` e restituisce la lista ottenuta.
Se la lista `h` non è vuota, occorre collegare il primo elemento della vecchia lista al nuovo elemento, come schematizzato nella figura.




```
elementBid *insert(int n, elementBid *h){
  elementBid *new = malloc(sizeof(elementBid));
  new->info = n;
  new->next = h;  // (A)
  new->prev = NULL;
  if(h != NULL)  // h non e' vuota
        h->prev = new;   // (B)
  return new;
}
```

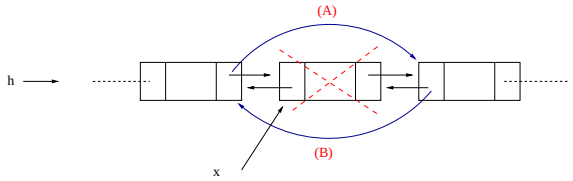
Dire cosa succede se si omette l'istruzione

```
new->prev = NULL;
```


Cancellazione

La funzione `delete()` cancella dalla lista `h` l'elemento di indirizzo `x` e restituisce la testa della lista ottenuta.

Occorre collegare l'elemento che precede `x` con quello che segue `x`, distinguendo alcuni casi particolari.




```
elementBid *delete(elementBid *x, elementBid *h){
    if(x->prev != NULL) // x non e' il primo elemento
        x->prev->next = x->next;  // (A)
    else  // x  e' il primo elemento
        h = x->next;  // la testa della lista cambia
    if(x->next != NULL) // x non e' l'ultimo elemento
        x->next->prev = x->prev; // (B)
    free(x);
    return h;
}
```

```
elementBid *delete(elementBid *x, elementBid *h){
    if(x->prev != NULL) // x non e' il primo elemento
        x->prev->next = x->next;  // (A)
    else  // x  e' il primo elemento
        h = x->next;  // la testa della lista cambia
    if(x->next != NULL) // x non e' l'ultimo elemento
        x->next->prev = x->prev; // (B)
    free(x);
    return h;
}
```

La complessità in tempo è $O(1)$ in quanto il numero di operazioni da compiere è indipendente dalla lunghezza della lista.

Considerazioni finali

Nella valutazione della complessità in tempo delle operazioni sugli insiemi dinamici il fattore che incide maggiormente è il *tempo di ricerca* di un elemento, che nel caso medio è $\Theta(n)$ (come nel caso peggiore).

Considerazioni finali

Nella valutazione della complessità in tempo delle operazioni sugli insiemi dinamici il fattore che incide maggiormente è il *tempo di ricerca* di un elemento, che nel caso medio è $\Theta(n)$ (come nel caso peggiore).

Questo significa che l'implementazione proposta non è efficiente.

Si noti che usando altre varianti delle liste (liste circolari, liste con sentinella, ...) non si hanno miglioramenti, perché in tutte queste strutture dati il tempo di ricerca è uguale a quello delle liste semplici.

Considerazioni finali

Nella valutazione della complessità in tempo delle operazioni sugli insiemi dinamici il fattore che incide maggiormente è il *tempo di ricerca* di un elemento, che nel caso medio è $\Theta(n)$ (come nel caso peggiore).

Questo significa che l'implementazione proposta non è efficiente.

Si noti che usando altre varianti delle liste (liste circolari, liste con sentinella, ...) non si hanno miglioramenti, perché in tutte queste strutture dati il tempo di ricerca è uguale a quello delle liste semplici.

Per ottenere miglioramenti significativi, bisogna usare strutture dati in cui *la ricerca è efficiente*, ad esempio alberi binari di ricerca, RB alberi, tabelle hash,....

Esercizi

1. Scrivere una funzione avente intestazione

```
element *reverse(element *h)
```

che inverte la lista `h` e restituisce la lista ottenuta.

2. Scrivere una funzione ricorsiva avente intestazione

```
element *printRev(element *h)
```

che stampa gli elementi della lista `h` in ordine inverso.

3. Scrivere una funzione avente intestazione

```
elementBid *concatenate(elementBid *h1, elementBid *h2)
```

che collega la lista bidirezionale `h2` in coda alla lista bidirezionale `h1` e restituisce la testa della lista bidirezionale ottenuta. Le liste `h1` e `h2` possono essere vuote.

4. Scrivere un programma per gestire insiemi di stringhe analogo al programma sugli interi. Per rappresentare un elemento della lista usare la struttura

```
struct element {
    char* word;  //  indirizzo della stringa
    struct element *next;
};
```

Il campo `word` è allocato dinamicamente.

5. Una lista di interi è *ordinata* se ogni elemento della lista è minore o uguale all'elemento successivo.
Scrivere le funzioni per gestire una lista ordinata di interi.
Si supponga di rappresentare un insieme dinamico di interi usando una lista ordinata. Discutere la complessità delle operazioni confrontando con il caso delle liste semplici.