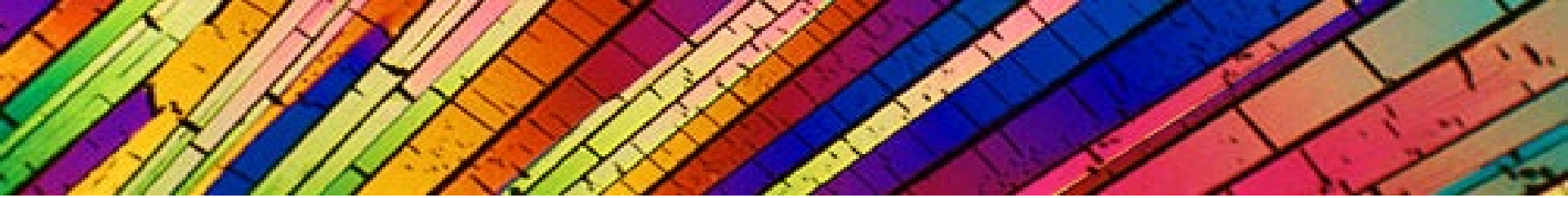
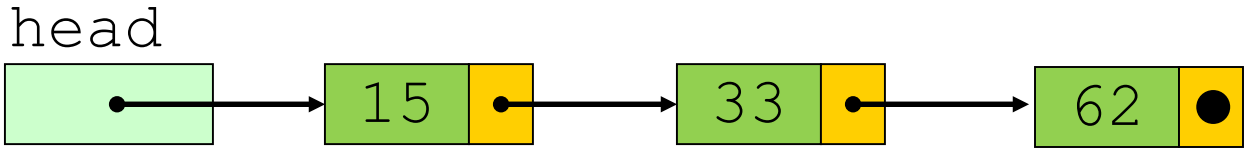


Liste concatenate
(introduzione)

Ver. 3

Lista concatenata

- Struttura dati in cui ogni oggetto ha un collegamento ad un altro oggetto (*linked list*)
- Il collegamento è un puntatore con l'indirizzo di memoria dell'altro oggetto



Strutture autoreferenzianti

- ```
  struct nodo
  {
      int valore;
      struct nodo *next;
  };
  ```

`next` è un puntatore al tipo di oggetto che si sta dichiarando in questo stesso momento, non essendo la dichiarazione della `struct` ancora completa si dice che ha un *tipo incompleto* (non è possibile stabilirne l'occupazione in memoria in questo momento). Il nome del tag qui è indispensabile
  ```

Strutture autoreferenzianti

- È comunque possibile definire un puntatore a un tipo incompleto, quindi ad esempio a una struttura che verrà definita successivamente, come nel caso precedente
- Per lo stesso motivo è lecito utilizzare la dichiarazione con `typedef` seguente:

```
typedef struct nodo * NodoP;
typedef struct nodo
{
   int valore;
   NodoP next;
} Nodo;
```

Strutture autoreferenzianti

- Per lo stesso motivo, è possibile definire strutture mutuamente referenzianti:

```
struct due
{
   int datoDue;
   struct uno *puntUno;
};
struct uno
{
   int datoUno;
   struct due *puntDue;
};
```

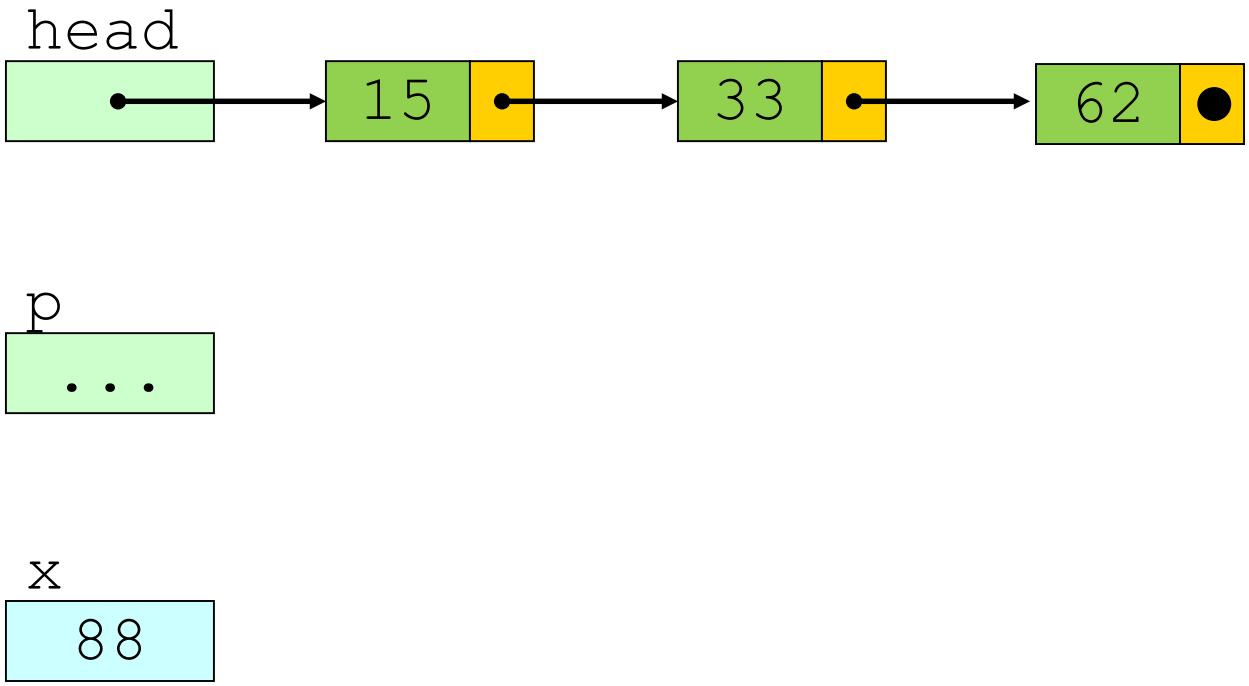
Testa e coda della lista

```
struct nodo
{
    int valore;
    struct nodo *next;
} *head = NULL;
```

- `head` è un puntatore a `struct nodo`, tutta la lista inizierà da questo puntatore
- La lista termina con un elemento in cui il membro `next` vale `NULL`
- La lista è inizialmente vuota, quindi `head` viene inizializzato a `NULL`

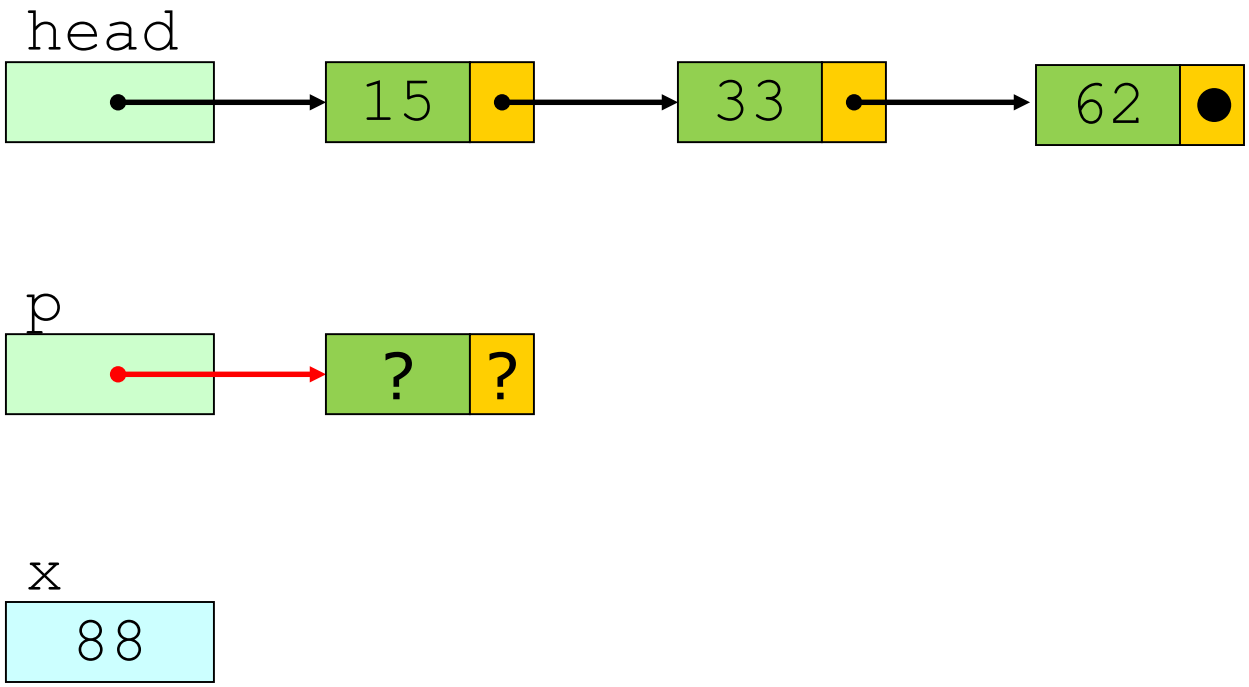
Inserimento in testa

- Situazione iniziale
  Caso generico: la lista non è vuota



Inserimento in testa

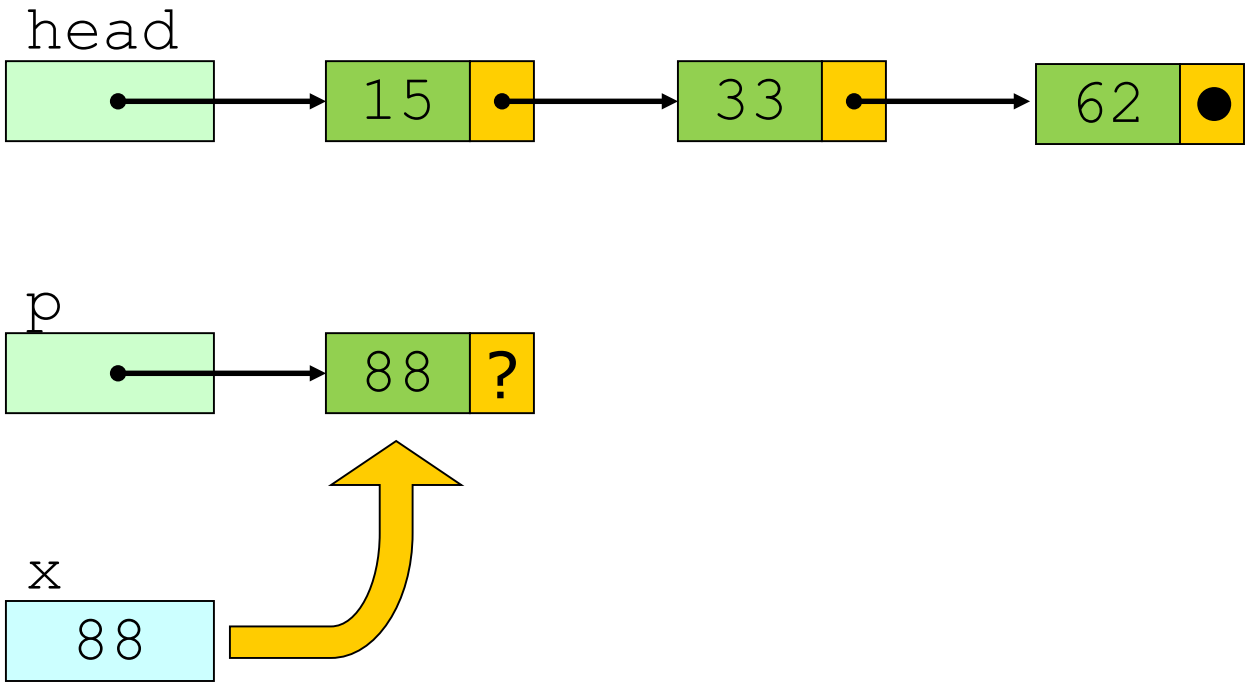
- Si crea la nuova `struct nodo` da inserire:
  `p=malloc(sizeof(struct nodo))`



Inserimento in testa

- Si immette il valore da memorizzare nella lista:

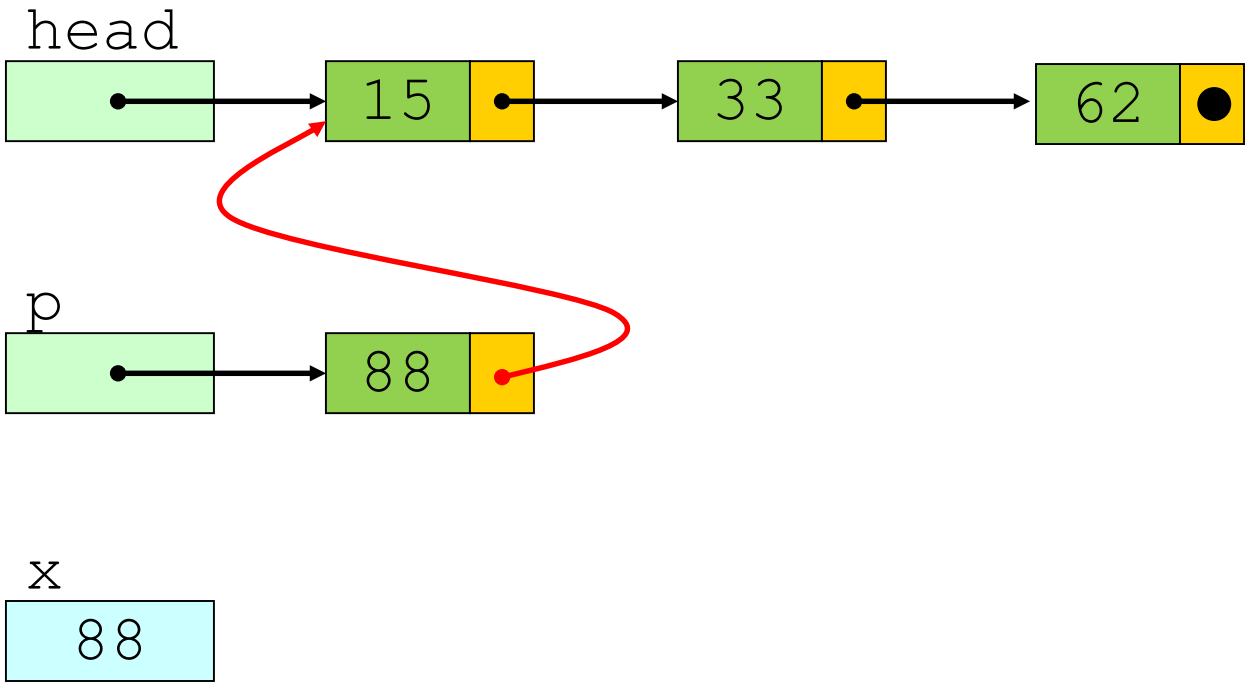
```
p->valore = x;
```



Inserimento in testa

- Si fa puntare il nuovo elemento al 1º:

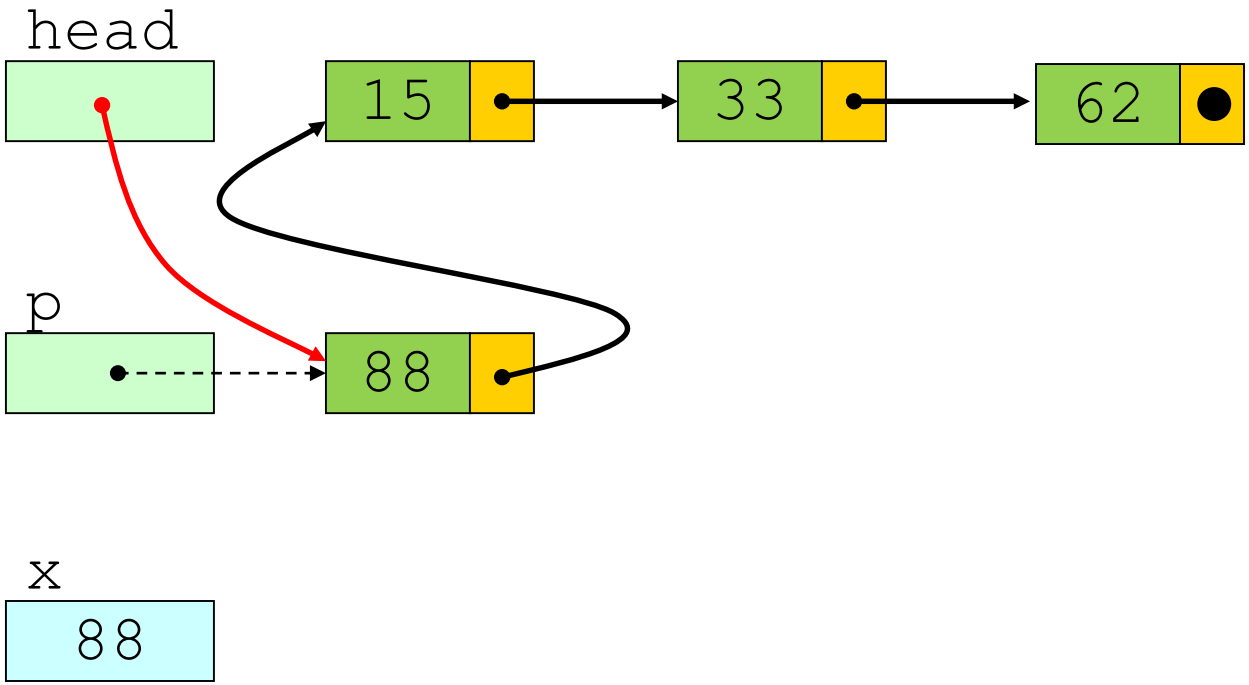
```
p->next = head;
```



Inserimento in testa

- Si fa puntare `head` al nuovo elemento:

```
head = p;
```



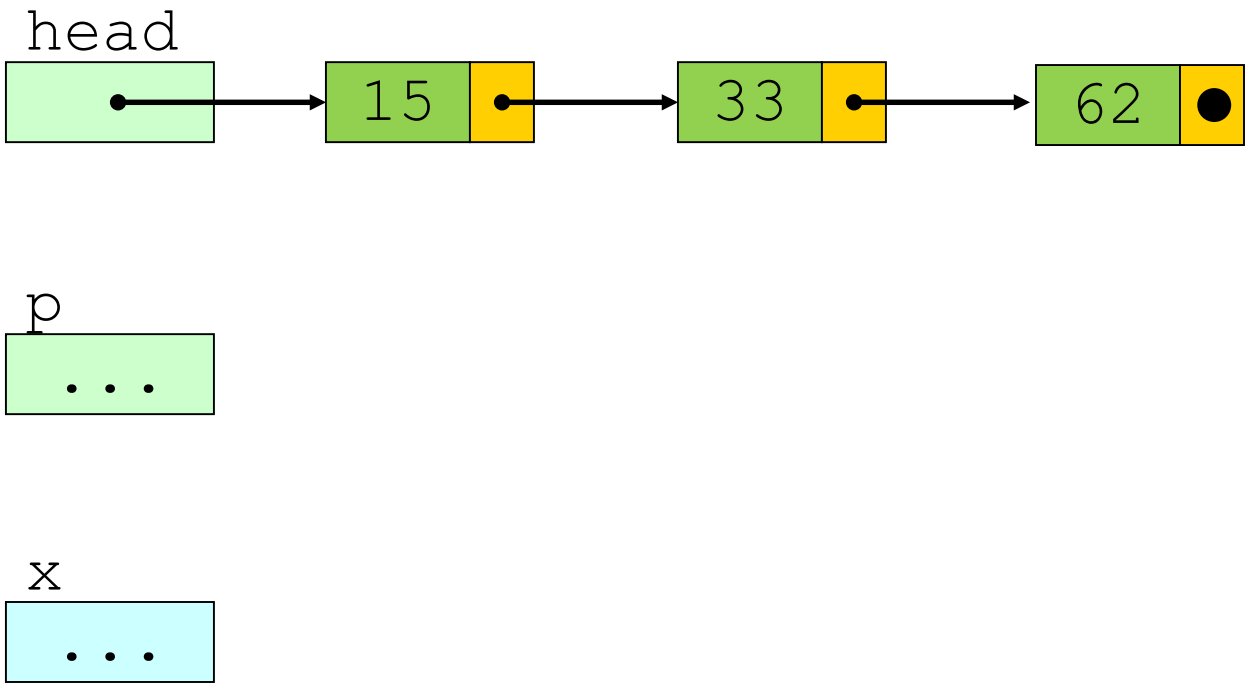
Inserimento in testa

Riassumendo:

```
if ((p=malloc(sizeof(struct nodo)))!= NULL)
{
      p->value = x;
      p->next = head;
      head = p;
}
```

Rimozione dalla testa

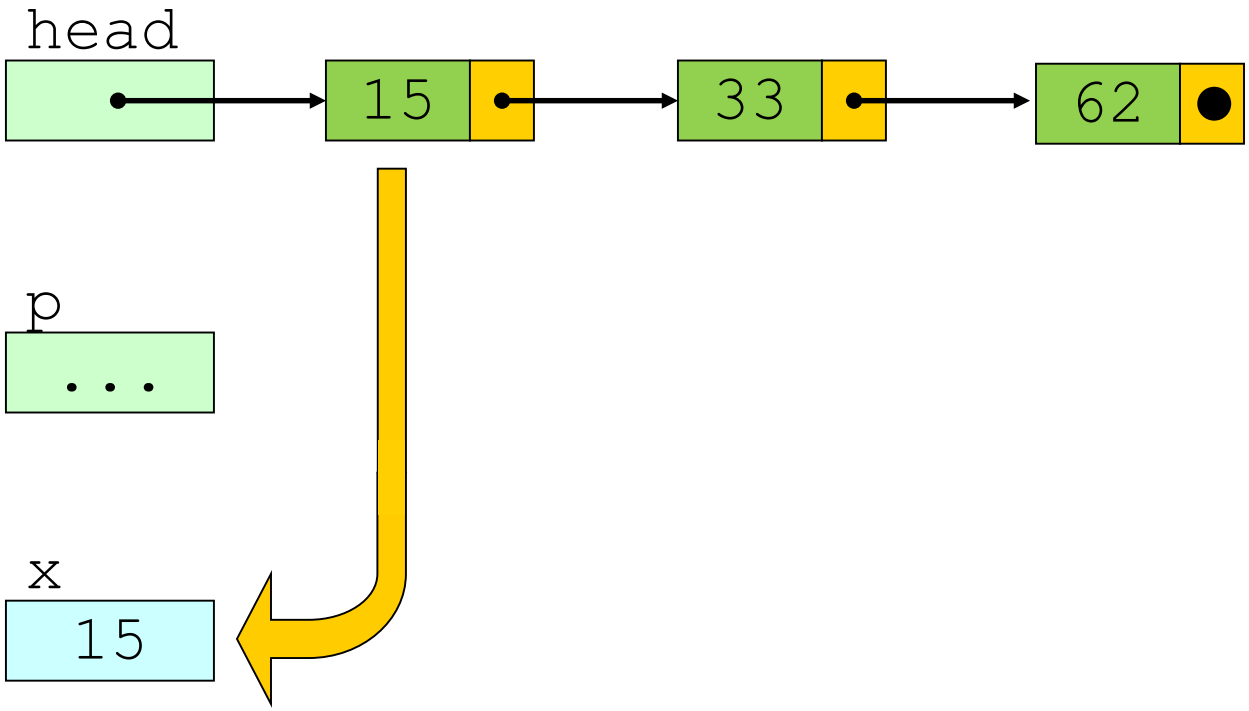
- Situazione iniziale



Rimozione dalla testa

- Si preleva il valore dal 1° elemento:

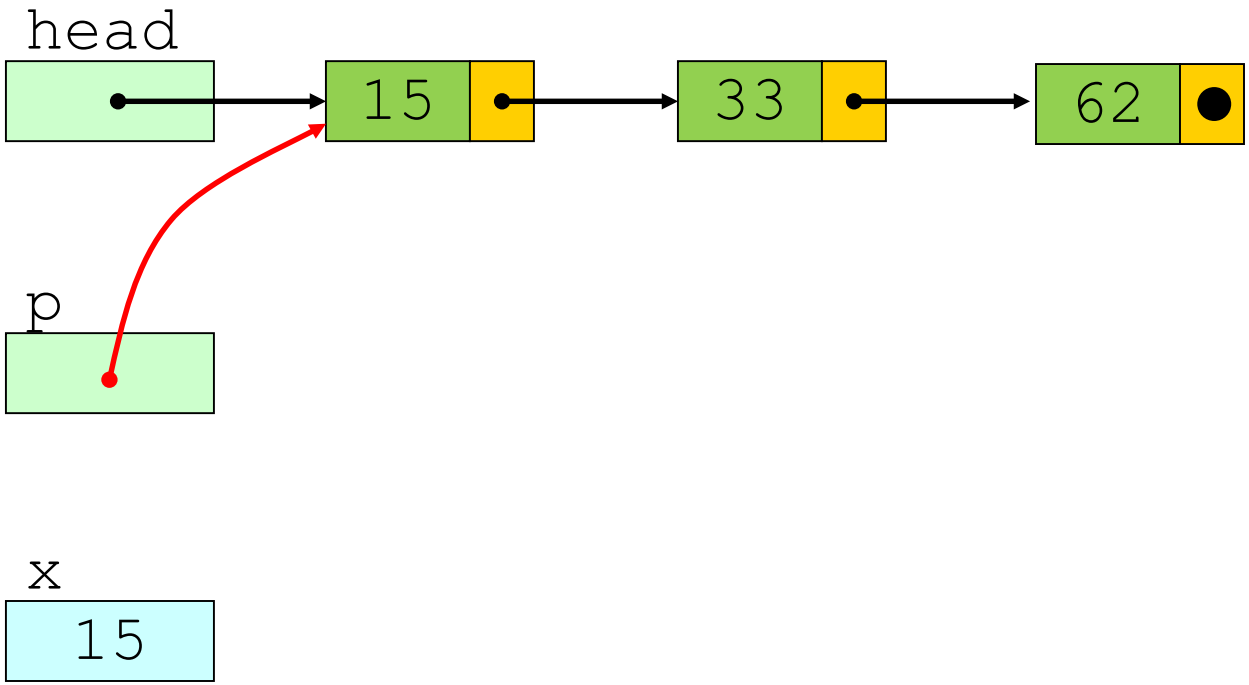
```
x = head->valore;
```



Rimozione dalla testa

- Si salva il puntatore all'elemento da rimuovere:

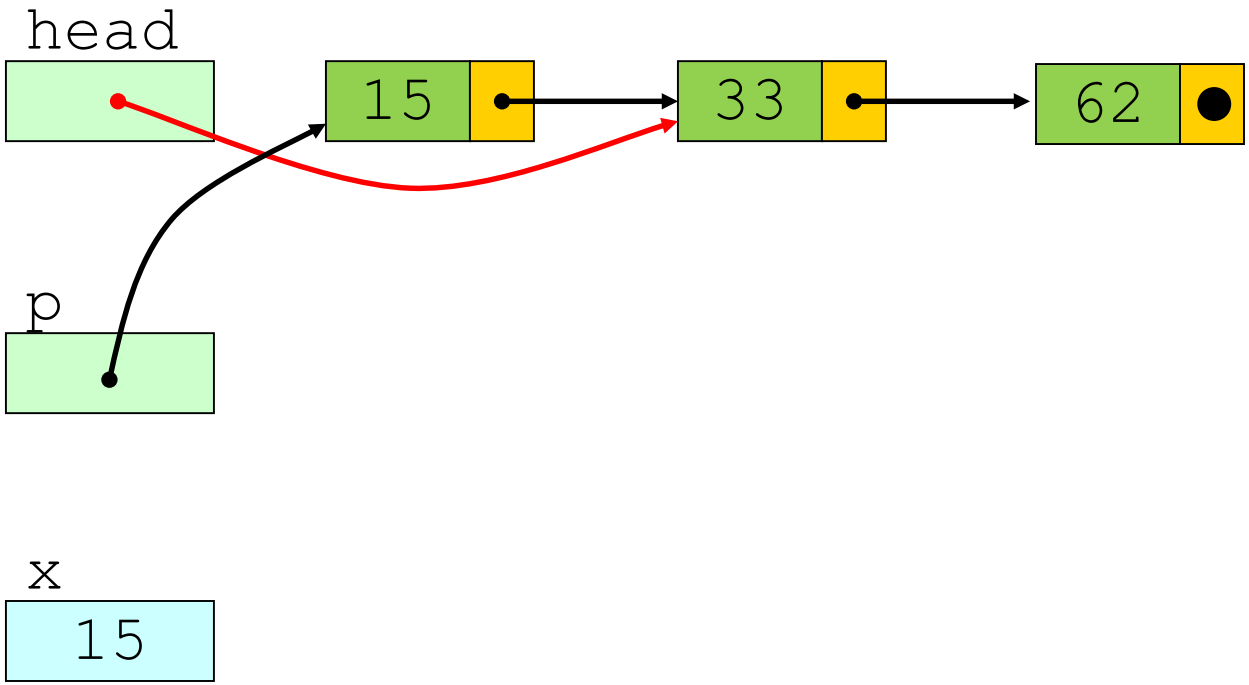
```
p = head;
```



Rimozione dalla testa

- Si fa puntare `head` al 2º elemento:

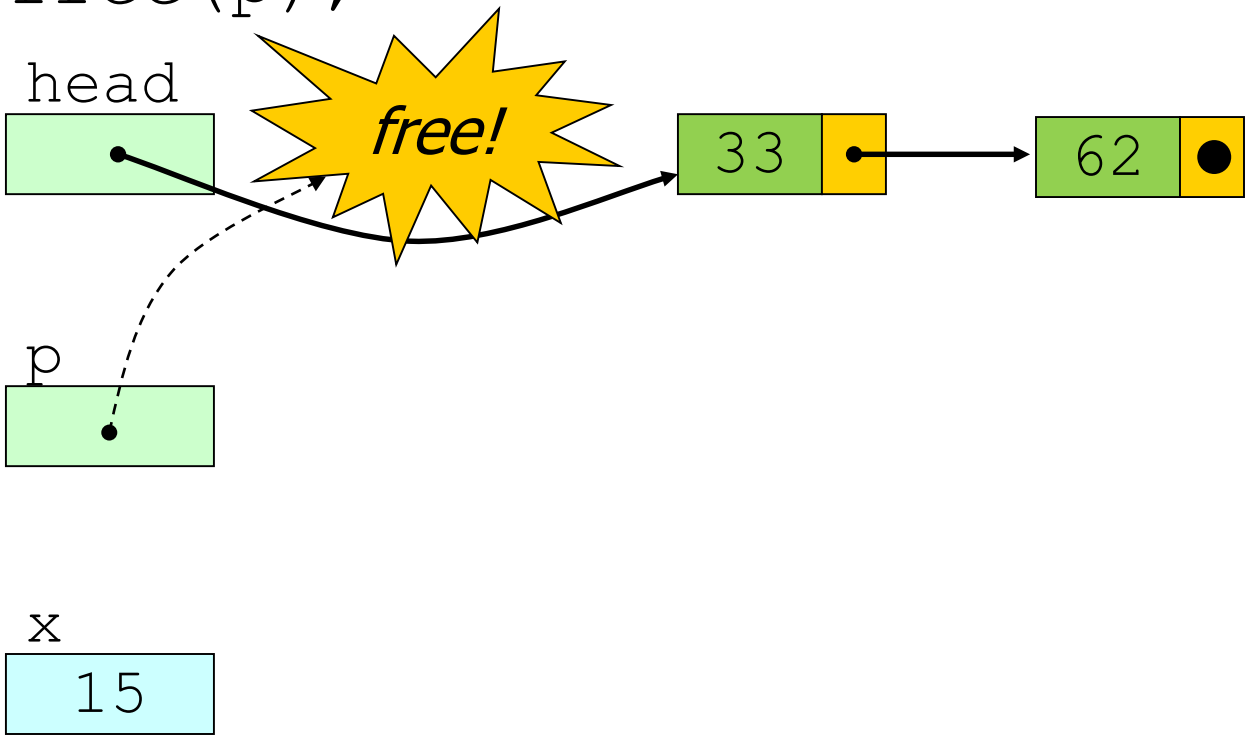
```
head = head->next;
```



Rimozione dalla testa

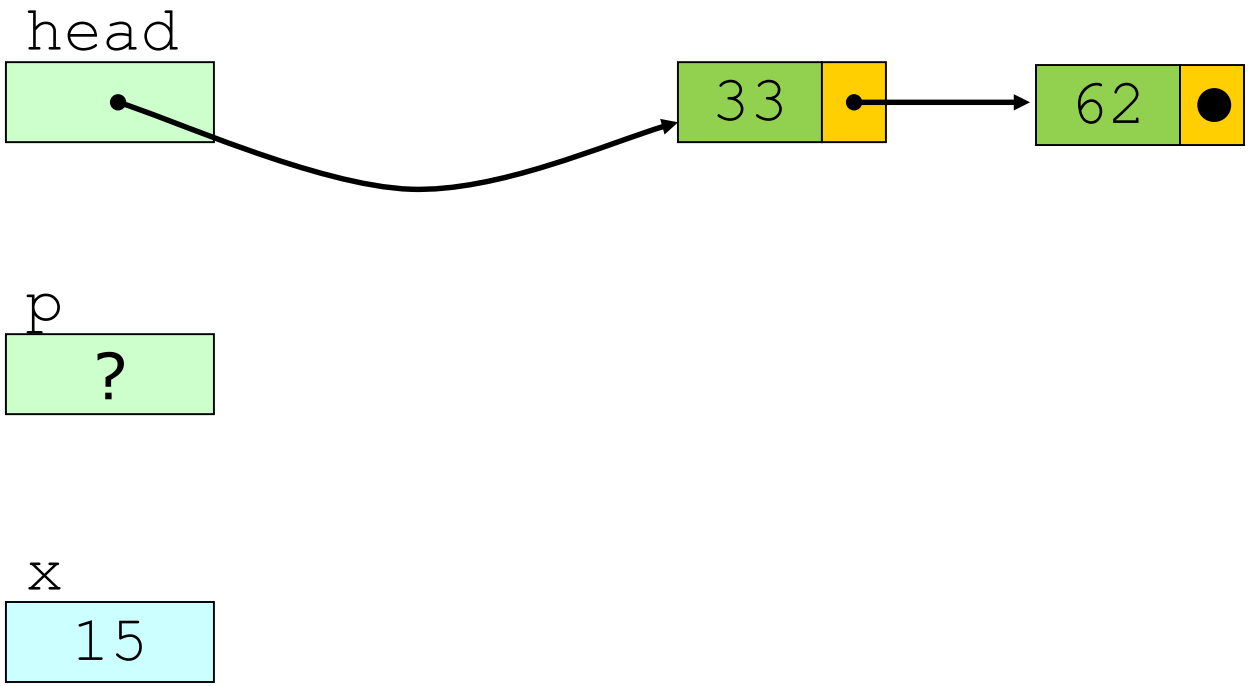
- Si dealloca lo spazio del 1° elemento:

```
free(p);
```



Rimozione dalla testa

- Situazione finale



Rimozione dalla testa

Riassumendo:

```
if (head != NULL)
{
   x = head->valore;
   p = head;
   head = head->next;
   free(p);
}
```

Considerazioni

- Una lista di strutture, rispetto a un vettore di strutture:
  - Non ha problemi di dimensioni: varia all'occorrenza
  - Lo scambio di elementi è molto veloce (copia di puntatori)
  - L'accesso è più lento (deve percorrere la lista dall'inizio fino all'elemento voluto)

Esercizi

1. Scrivere in un file separato (con controlli di errore) le seguenti funzioni di gestione di una lista dinamica di `int` stabilendo per esse appropriati argomenti e valori restituiti. La struttura dati sia `static`. Produrre un `main` a menu per il test.
   - `AddToHead()`
   - `AddToTail()`
   - `RemoveFromHead()`
   - `RemoveFromTail()`
   - `ClearAll()` → *svuota la lista*

Esercizi

2. Utilizzando le funzioni dell'esercizio 1 (senza modificare il file), scrivere le funzioni `push`/`pop` ed `enqueue`/`dequeue` con gli stessi identici prototipi definiti per stack e coda realizzata con vettori. Il `main` deve essere lo stesso identico usato con le realizzazioni con i vettori (e quindi compatibile con la nuova realizzazione).
   Per evitare l'overhead della doppia chiamata a funzione (es. la chiamata a `push` che a sua volta chiama `AddToHead`), le funzioni richieste vengano realizzate come macro (nel file `.h` da includere nel `main`).

Variazioni

- **Lista con valori fittizi (*dummy*)**
  Per semplificare il codice di alcune operazioni (avere meno casi particolari) può essere utile che la lista abbia uno o più elementi *fittizi* aggiuntivi (non contengono valori significativi):
  - come primo elemento fisso puntato da `head`
  - come ultimo elemento fisso della lista (*sentinella*)
- **Lista circolare**
  L'ultimo elemento punta al primo
- **Multi-lista**
  Ogni elemento della lista ha due o più puntatori ai nodi successivi, ci possono essere quindi più modi di percorrerla (es. diversi ordinamenti)

Homework 8

Scrivere in un file separato le funzioni di gestione di una lista di `int`, passando anche la testa della lista (per avere più liste).

La lista potrà avere due stati: *ordinata* o *non ordinata*; alcune funzioni riordineranno preventivamente la lista (se necessario), altre la lasceranno non più ordinata; alcune funzioni lavoreranno in modo diverso a seconda che la lista sia ordinata o no.

- Può essere utile avere un puntatore all'ultimo elemento e un contatore degli elementi
- Ricordarsi di eliminare (`free`) tutti gli elementi che non servono più

Si scriva un main di test

Homework 8

(Continuazione)

- Funzioni di base
  - `listAddToHead` (rende la lista non ordinata)
  - `listAddToTail` (rende la lista non ordinata)
  - `listRemoveFromHead` (e ne restituisce il valore)
  - `listRemoveFromTail` (e ne restituisce il valore)
  - `listClearAll` (svuota la lista)
  - `listCount` (restituisce il numero degli elementi della lista)
  - `listTestIsEmpty` (test se la lista è vuota)

Homework 8

(Continuazione)

- Funzioni di ordinamento e ricerca
  - `listSort` (ordina la lista in senso ascendente o discendente in base a un parametro passato)
  - `listAddOrdered` (aggiunge in lista ordinata al posto giusto, se necessario la lista venga ordinata)
  - `listFind` (cerca il valore dato nella lista a partire dall'elemento di indice indicato, restituisce l'indice della prima occorrenza, se la lista è ordinata deve fermarsi appena scopre che l'elemento non c'è)
  - `listRemoveValue` (rimuove dalla lista l'elemento contenente la prima occorrenza del valore dato e lo restituisce)

Homework 8

(Continuazione)

- Funzioni assolute di scansione della lista
  - `listGetValueAt` (restituisce il valore dell'elemento di indice i, il primo abbia indice 1)
  - `listSetValueAt` (cambia il valore dell'elemento di indice i)
  - `listAddAt` (aggiunge un elemento in modo che occupi la posizione di indice i, facendo avanzare gli altri elementi)
  - `listRemoveAt` (elimina e restituisce l'elemento di indice i)

Homework 9

Per accedere ai singoli elementi della lista si può definire un "cursore" (un puntatore) che si riferisca (punti) ai singoli elementi della lista

Il cursore può essere spostato avanti e indietro nella lista per identificare un particolare elemento.

Si continui l'Homework precedente aggiungendo le seguenti funzioni per l'uso dei cursori. Si faccia attenzione a gestire correttamente i cursori quando la lista viene modificata con le altre funzioni e viceversa.
Il main di test sia identico a quello dell'Hw 8

Homework 9

(Continuazione)

- Funzioni di scansione della lista con cursore
  - `listGetCursor` (restituisce l'indice dell'elemento puntato dal cursore, il primo abbia indice 1)
  - `listSetCursor` (posiziona il cursore all'elemento specificato dall'indice fornito)
  - `listMoveCursor` (muove il cursore al primo elemento, al successivo, al precedente, all'ultimo – utilizzare una `enum` per dare i nomi delle posizioni `FIRST`, `NEXT`, `PREVIOUS`, `LAST`)
  - `listGetValueAtCursor` (restituisce il valore dell'elemento puntato dal cursore)
  - `listSetValueAtCursor` (cambia il valore dell'elemento puntato dal cursore)

Homework 9

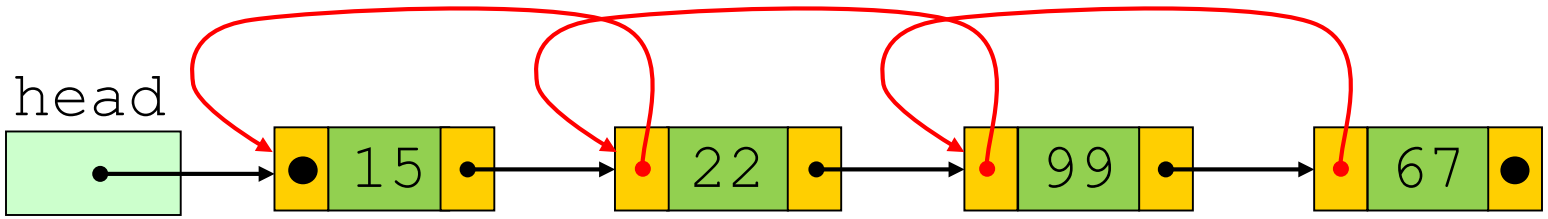
(Continuazione)

- Funzioni di scansione della lista con cursore
  - `listAddAtCursor` (aggiunge un elemento alla posizione del cursore, facendo avanzare gli altri elementi)
  - `listRemoveAtCursor` (elimina l'elemento puntato dal cursore)
  - `listTestCursorAtFirst` (test se punta al primo elemento)
  - `listTestCursorAtLast` (test se punta all'ultimo elemento)

Liste bidirezionali

- La bidirezionalità si ottiene mediante un link al nodo precedente (*doubly-linked list*)

```
struct nodo
{
   int valore;
   struct nodo *prev;
   struct nodo *next;
} *head=NULL;
```



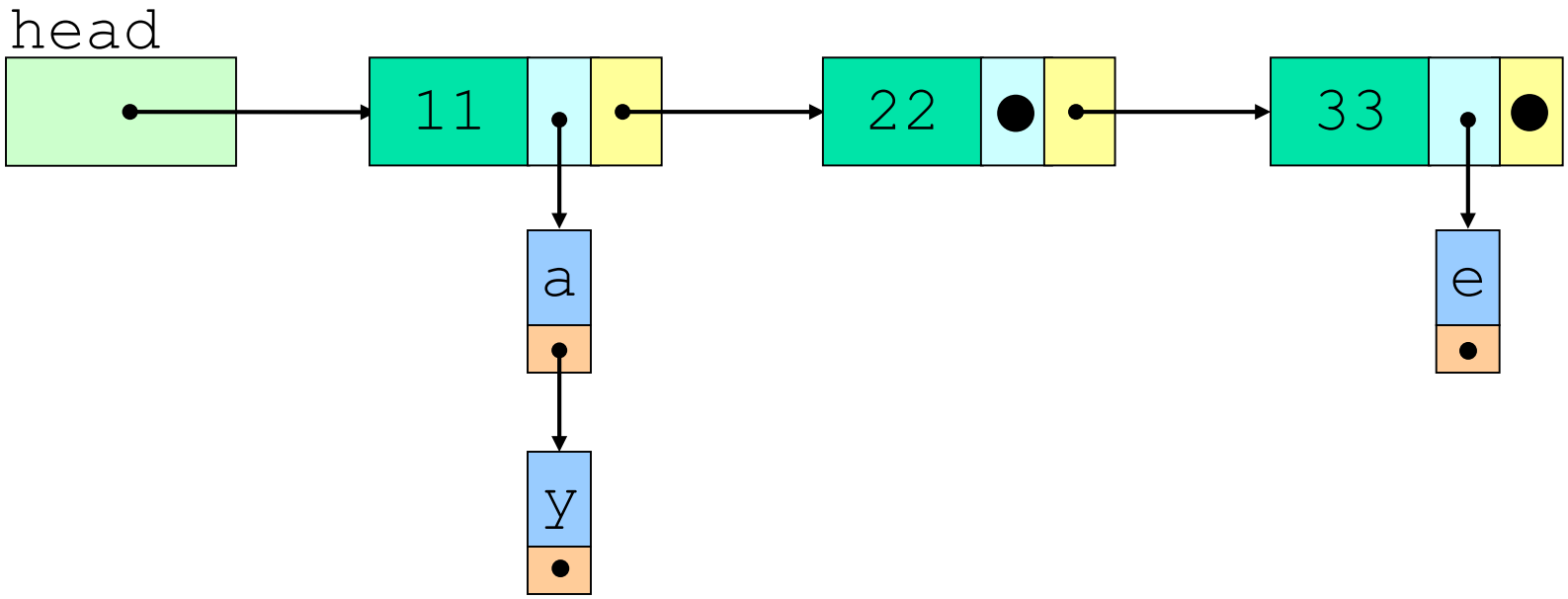
Homework 10

Come gli Homework 8 e 9, ma si usino liste bidirezionali.

Il `main` deve essere identico a quello degli Homework 8 e 9.

Liste di liste

- Alcuni dei membri dei nodi di una lista principale sono le teste di liste secondarie



Considerazioni

- Problema: in uno stesso programma servono liste con nodi di tipo diverso
- Ad es., in una lista di liste la lista principale e quelle secondarie hanno nodi di tipo diverso
- Soluzione dei linguaggi procedurali (C): una serie di funzioni diverse per ciascun tipo di lista
- Soluzione ideale: una serie di funzioni *generiche* che gestiscano qualsiasi tipo di elemento (oggetto)

Si usano allora ***linguaggi ad oggetti***

Tabelle di hash

- Per velocizzare la ricerca di un elemento in una lista, *si distribuiscono gli elementi su più liste* (più corte quindi più veloci da utilizzare)
- Le teste di queste liste sono in un vettore di puntatori
- La scelta di quale lista debba ospitare i singoli elementi viene stabilito da un opportuno calcolo (*funzione di hash*) sull'elemento stesso, il risultato è l'indice che in quel vettore indica quale lista usare

Tabelle di hash
Esempio

- Esempio
  Suddivisione dei valori in base alla cifra delle unità

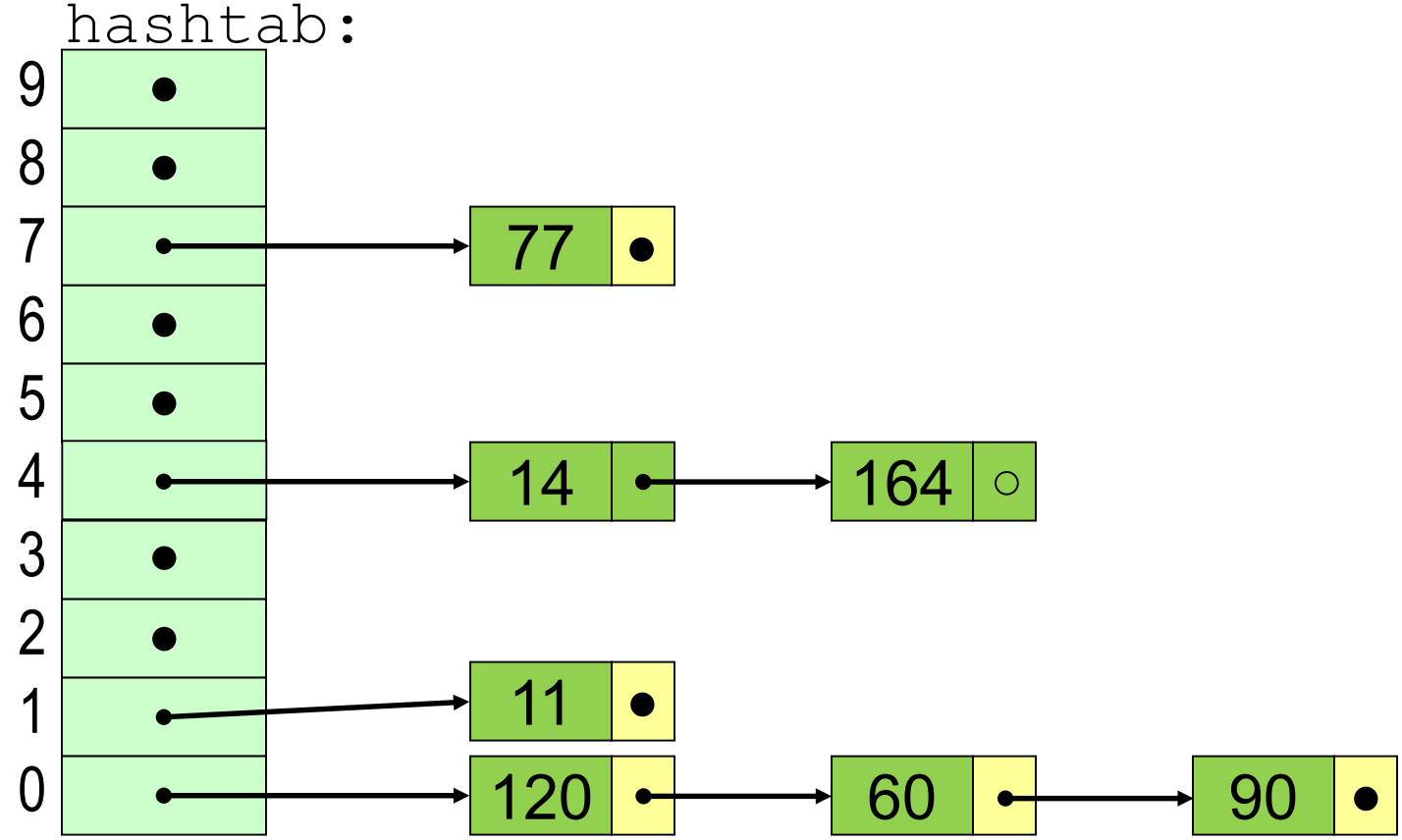


Tabelle di hash
Esempio

- La funzione di hash è:
```
int hash(int n){
   return abs(n) % 10;}
```
- Il vettore di puntatori è:
```
static struct nodo *hashtab[10];
```
- La testa della lista relativa a x è:
```
hashtab[hash(x)]
```
- Per trovare `x` nella lista corretta:
```
listFind(hashtab[hash(x)], x);
```
- Per inserire `x` nella lista corretta:
```
listAdd(hashtab[hash(x)], x);
```

Tabelle di hash

- Una buona funzione di hash genera valori con distribuzione uniforme, ossia le liste prodotte hanno lunghezze più o meno uguali
- Esempio di funzione di hash per stringhe:

```
unsigned hash(char *s)
{
  unsigned hashval = 0;
  for ( ; *s != '\0'; s++)
    hashval = *s + 31 * hashval;
  return hashval % HASHSIZE;
}
```

Homework 11

Si scriva un programma per generare l'indice analitico di un (lungo) file di testo. Il programma deve memorizzare le singole parole (si usi una lista ordinata per ciascuna delle lettere iniziali) e per ciascuna deve memorizzare i numeri di tutte le righe dove è presente (sotto-lista ordinata). Un secondo file di testo elenca le parole da non tenere in considerazione (es. articoli, preposizioni, etc.), per questa si usi un opportuna funzione hash. Il risultato della generazione dell'indice analitico sia scritto in un file.