Strutture Dati per Inserimento Ordinato

Luca Abeni

Esempio: Ordinamento di Numeri

- A cosa servono i tipi di dato strutturati? Non bastano i tipi scalari?
  - Capiamolo con un esempio...
- Problema: dato un insieme di numeri naturali, ordinarli
- Possibile soluzione: Ordinamento per Inserimento Diretto
  1. Scorrere tutti i numeri dal secondo all'ultimo
  2. Per ogni numero considerato, inserirlo nel giusto ordine fra i precedenti
- Descrizione molto informale...
  - "*Scorrere tutti i numeri*"?
  - "*Per ogni numero considerato*"?
  - "*Inserire nel giusto ordine*"?

Ordinamento per Inserimento - Esempio

- Cerchiamo di capire meglio l'ordinamento per inserimento...
- ...tramite un esempio!

44	55	12	42	94	18	6	67
44	55	12	42	94	18	6	67
44	55	12	42	94	18	6	67
12	44	55	42	94	18	6	67
12	42	44	55	94	18	6	67
12	42	44	55	94	18	6	67
12	18	42	44	55	94	6	67
6	12	18	42	44	55	94	67
6	12	18	42	44	55	67	94

Formalizziamo l'Algoritmo - 2

- Contatore `i`: posizione del numero che stiamo considerando
- Per ogni valore di `i` (da 2 a n), considera i numeri con posizione $<$ `i`
  - Altro contatore `j`, che va da `i - 1` a 1 (conta indietro)
- Se il numero in posizione `j` è $>$ del numero in posizione `i`, sposta tale numero a destra
- Se il numero in posizione `j` è $<$ del numero in posizione `i`, ferma il contatore `j` ed inserisci in posizione `j + 1` il numero che era in posizione `i`

Formalizziamo l'Algoritmo - 1

- Variabili: i, j. Insieme di definizione: numeri naturali
- Inoltre, appoggio (intero) e numeri da ordinare...
- `i = 2`
- mentre `i` $\leq n$
  - `appoggio` = numero in posizione `i`
  - `j = i - 1`
  - mentre `j` > 0 e numero in posizione `j` $>$ `appoggio`
    - numero in posizione `j + 1` = numero in posizione `j`
    - `j = j - 1`
  - `i = i + 1`
  - numero in posizione `j + 1` = `appoggio`

Formalizziamo l'Algoritmo - 3

- "numero in posizione `i` / `j`"?
  - Formalizziamo un po' meglio questa cosa...
- Usiamo un nome per identificare tutti i numeri...
  - Se un numero ha insieme di definizione $\mathcal{Z}$, questa variabile ha insieme di definizione $\mathcal{Z}^n$
- ...ed identifichiamo il singolo numero tramite un indice
  - In matematica, $\texttt{numero}_i$, $\texttt{numero}_j$
  - Ma usiamo qualche simbolo che sia meglio rappresentabile sui computer
  - `numero[i]`, `numero[j]`

Formalizziamo l'Algoritmo - 4

- Variabili: `i`, `j` $\in \mathcal{N}$; `appoggio` $\in \mathcal{Z}$ e `numero` $\in \mathcal{Z}^n$
- `i = 2`
- mentre `i` $\leq n$
  - `appoggio` $=$ `numero[i]`
  - `j = i - 1`
  - mentre `j` > 0 e `numero[j]` $>$ `appoggio`
    - `numero[j + 1] = numero[j]`
    - `j = j - 1`
  - `i = i + 1`
  - `numero[j + 1] = appoggio`

Array

- `numero` $\in \mathcal{Z}^n$
  - Variabile con insieme di definizione $\mathcal{Z}^n$
  - Composta da n variabili con insieme di definizione $\mathcal{Z}$ (variabili intere)
  - Singoli elementi accessibili come `numero[i]`
- Abbiamo appena scoperto gli Array!
- Array: tipo *strutturato* composto da variabili di un tipo semplice ripetute più volte
- Tutti gli elementi dell'array hanno lo stesso tipo

Rivisitiamo L'Esempio

- Abbiamo visto come ordinare un insieme di numeri preesistente
- Se i numeri vengono inseriti uno dopo l'altro, è possibile usare un algoritmo più furbo?
- Algoritmo stupido:
  1. Immettere n numeri uno dopo l'altro ed inserirli in un array
  2. Ordinare l'array usando inserimento diretto, selezione diretta, scambio diretto, ...
- Ma forse i numeri possono essere inseriti nell'array già in ordine?

Inserimento Ordinato in un Array

- Problema: dato un numero `val` $\in n$, inserirlo nella posizione corretta di un array `numeri` $\in N^n$
- Possibile soluzione:
  - Scorrere l'array da sinistra verso destra fino a che non si trova un numero più grande di `val`
  - Spostare a sinistra tale numero e tutti i numeri successivi nell'array
  - Inserire `val` nel posto vuoto che si è creato

Formalizziamo l'Algoritmo

- Variabili: `i`, `j` $\in \mathcal{N}$; `val` $\in Z$ e `numero` $\in \mathcal{Z}^n$
- `i = 0`
- mentre `i` $< n$ e `numero[i] < val`
  - `i = i + 1`
- `j = n`
- mentre `j > i`
  - `numero[j] = numero[j - 1]`
  - `j = j - 1`
- `numero[i] = val`

Problemi con L'Algoritmo

- L'algoritmo mostrato nella slide precedente funziona...
- Il problema iniziale è risolubile ripetendo n volte
  - Leggi `val`
  - `inserimento_ordinato(val, numero, i)`
- Ma ogni volta l'algoritmo può dover "spostare a destra" dei numeri nell'array
  - Se `val` è più piccolo di tutti i numeri dell'array, tutti i numeri vanno spostati!!!
  - Perdita di tempo...
- Si può fare un inserimento ordinato senza dover spostare numeri?

Array e Inserimento Ordinato

- L'inserimento ordinato di elementi può essere fatto in modo più efficiente usando strutture dati appropriate
  - Array: possono richiedere sempre di spostare elementi per inserirne uno in ordine
  - Che struttura dati utilizzare?
- Problema con gli array: ordine degli elementi ed ordine di memorizzazione sono legati fra loro
  - Serve una struttura dati che renda l'ordine dei componenti indipendente dalla loro posizione in memoria

Ordine e Posizione in un Array

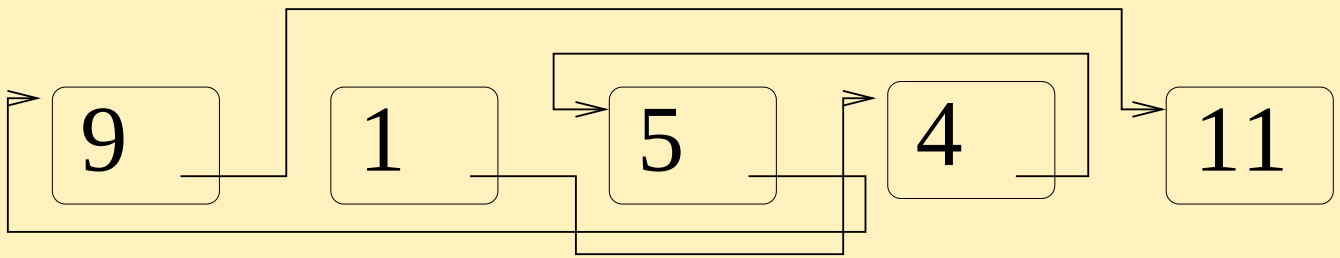
- Array: collezione di variabili omogenee ordinate
- Indice: numero intero che identifica un elemento dell'array
  - L'indice stabilisce l'ordine delle variabili dell'array (`v[i]` è l'`i`-esimo elemento dell'array `v`)
  - Stabilisce la posizione in memoria di un elemento dell'array

$$\mathtt{v} \in \mathcal{N}^d\colon\ \mathtt{v} \rightarrow \boxed{\mathtt{v1}}\boxed{\mathtt{v2}}\boxed{\mathtt{v3}}\cdots\boxed{\mathtt{vd}}$$

- Serve un insieme di variabili omogenee con un ordinamento diverso da quello stabilito dalla posizione in memoria...
  - Idea: ad ogni elemento dell'array associamo un valore e l'indice del prossimo elemento secondo un ordinamento indipendente dalla posizione in memoria

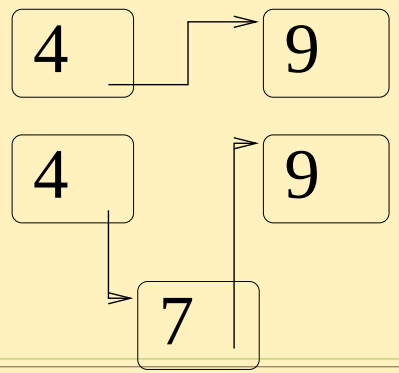
Indice del Prossimo Elemento

- Posizione in memoria arbitraria
  - Posizione nell'array: ordine di inserimento
  - No necessità di spostare elementi per inserirne uno in ordine!
- Ordinamento dato dagli indici al prossimo elemento
- Indice al prossimo elemento: rappresentato graficamente con una freccia
  - `v[j]` viene dopo `v[i]` $\Rightarrow$ il "prossimo indice" di `v[i]` è `j`
  - Freccia da `v[i]` a `v[j]`



Inserimento Ordinato

- Inserimento: basta aggiornare gli indici
- Se ho due elementi consecutivi con valore 4 e 9, il "prossimo indice" del primo punta al secondo
- Se inserico un elemento con valore 7, deve andare fra questi due...
  - Modifico il "prossimo indice" del primo elemento per puntare al nuovo elemento
  - Il "prossimo indice" del nuovo elemento punta all'elemento con valore 9



Implementazione - Struttura "elemento"

- Ogni elemento è caratterizzato da:
  - Un valore ($\in \mathcal{Z}$)
  - Un indice del prossimo elemento ($\in \mathcal{N}$)
- È un tipo di dato "complesso"!!!
  - Usiamo una struttura
  - Due campi: `val` $\in \mathcal{Z}$ e `prossimo` $\in \mathcal{N} \cup \{-1\}$
- Perché `prossimo` $\in \mathcal{N} \cup \{-1\}$?
  - Serve un valore di `prossimo` per indicare "non esiste un prossimo elemento" (questo è l'ultimo elemento)
  - Scegliamo arbitrariamente -1

Implementazione - Continua

- Array di N strutture "elemento"
  - N: numero massimo di numeri da ordinare
- Serve inoltre una variabile `primo` $\in \mathcal{N} \cup \{-1\}$
  - Inizializzata a `primo = -1` (nessun numero inserito)
- Ogni volta che inserisco un numero:
  - Lo memorizzo nel campo `val` del primo elemento libero dell'array
    - L'n-esimo numero è memorizzato in `numero[n].val`
  - Aggiusto la variabile `primo` ed i campi `prossimo` per inserire l'elemento nel giusto ordine

Implementazione - Inserimento

- Se non ho ancora inserito numeri o il numero che sto inserendo è il più piccolo, modifico `primo`, altrimenti, devo "seguire" i vari campi `prossimo`
- Se `primo == -1` o `numero[primo].val` $>$ `numero[n].val`
  - `numero[n].prossimo = primo`
  - `primo = n`
- Altrimenti,
  - `i = primo`
  - Mentre `i` $\neq$ `-1` e `numero[i].val < numero[n].val`
    - `i = numero[i].prossimo`
  - Per aggiornare correttamente i campi `prossimo`, devo ricordare l'indice dell'elemento precedente a `numero[i]`

Descrizione Formale

- `numero[n].val = val`
- Se `primo == -1` o `numero[primo].val` $>$ `val`
  - `numero[n].prossimo = primo`
  - `primo = n`
- Altrimenti,
  - `i = primo`
  - Mentre `i` $\neq -1$ e `numero[i].val` $<$ `val`
    - `prev = i`
    - `i = numero[i].prossimo`
  - `numero[n].prossimo = i`
  - `numero[prev].prossimo = n`

Liste Ordinate

- Sequenza ordinata di strutture contenenti un campo valore ed un indice del prossimo elemento della sequenza
  - In teoria, anche più di un campo valore!!!
  - L'indice del prossimo delemento definisce l'ordinamento
  - "Collegamento" (link) fra due elementi consecutivi
- Abbiamo appena re-inventato le Liste!!!
  - In particolare, liste ordinate

Codice C Rivisitato

```
numero[i].val = val;
if ((primo == -1) || (numero[primo].val > val)) {
  numero[i].prossimo = primo;
  primo = i;
} else {
  int j;
  unsigned int prev;

  j = primo;
  while ((j != -1) && (numero[j].val <= val)) {
    prev = j;
    j = numero[j].prossimo;
  }
  numero[i].prossimo = j;
  numero[prev].prossimo = i;
```

- Ogni singolo *nodo* della lista è acceduto tramite l'array `numero`...
- Esempio: numero[i].prossimo = j; numero[prev].prossimo = i
- Perché non identificare i nodi semplicemente tramite il loro indirizzo in memoria???

Indirizzo di un Nodo - 1

- Idea: invece di riferirsi ad ogni nodo usando un indice in un array...
- ...Usiamo l'indirizzo di memoria dove tale nodo è memorizzato!!!
- Si può fare?
- Abbiamo bisogno di:
  1. Avere un modo per conoscere l'indirizzo dove una variabile è memorizzata
  2. Avere un tipo di variabile che possa contenere indirizzi di memoria
  3. Avere un modo per conoscere il contenuto della variabile memorizzata ad un indirizzo specificato

Indirizzo di un Nodo - 2

- Supponiamo di poter usare gli indirizzi in memoria dei nodi:
  1. Indirizzo in memoria del nodo `numero[i]`: *i*`(numero[i])`
  2. Esiste un tipo di variabili atto a contenere gli indirizzi di un nodo. Il valore `NULL` indica un indirizzo non valido (come `-1` per gli indici)
  3. Nodo memorizzato all'indirizzo `p`: *v*`(p)`
- L'algoritmo è riscrivibile:
  - Usando variabili di tipo indirizzo invece che indici interi
    - Introducendo una variabile `attuale` = *i*`(numero[n])` contenente l'indirizzo di `numero[n]`
    - `primo` e `prossimo` non sono più indici ma variabili di tipo indirizzo
  - Sostituendo `numero[k]` con *v*`(k)`

Modifichiamo L'Algoritmo...

- `n` $\in \mathcal{N}$; `i`, `prev` $\in \mathcal{N} \cup \{-1\}$
- `numero[n].val = val`
- Se `primo == -1` o `numero[primo].val` $>$ `val`
  - `numero[n].prossimo = primo`
  - `primo = n`
- Altrimenti,
  - `i = primo`
  - Mentre `i` $\neq -1$ e `numero[i].val` $<$ `val`
    - `prev = i`
    - `i = numero[i].prossimo`
  - `numero[n].prossimo = i`
  - `numero[prev].prossimo = n`

- `n` $\in \mathcal{N}$; `i`, `prev` $\in$ {indirizzi} $\cup$ {NULL}
- attuale $=$ *i*(numero[n]), *v*`(attuale).val = val`
- Se `primo == NULL` o *v*`(primo).val` $>$ `val`
  - *v*`(attuale).prossimo = primo`
  - `primo = attuale`
- Altrimenti,
  - `i = primo`
  - Mentre `i` $\neq$ `NULL` e *v*`(i).val` $<$ `val`
    - `prev = i`
    - `i =` *v*`(i).prossimo`
  - *v*`(attuale).prossimo = i`
  - *v*`(prev).prossimo = attuale`

Algoritmo Modificato

- attuale = *i*(numero[n]), *v*(attuale).val = val
- Se primo == NULL o *v*(primo).val > val
  - *v*(attuale).prossimo = primo
  - primo = attuale
- Altrimenti,
  - i = primo
  - Mentre i ≠ NULL e *v*(i).val < val
    - prev = i
    - i = *v*(i).prossimo
  - *v*(attuale).prossimo = i
  - *v*(prev).prossimo = attuale

Indirizzo di una Variabile

- L'operatore & può essere usato per risalire dal nome di una variabile al suo indirizzo in memoria
- Esempio di utilizzo: `scanf()`
  - `scanf()` riceve come parametri gli indirizzi delle variabili dove memorizzare i dati immessi, non le variabili stesse

```
int main()
{
  int i = 10;

  printf("Valore di i: %d\n", i);
  printf("Indirizzo di i: %p\n", &i);

  return 0;
}
```

Operatore &

- `&` si applica a nomi di variabili
  - No costanti o espressioni
- `&` non si applica a sinistra di un =
  - No cose tipo `&i = 666;`
- L'effetto di `&` è di trasformare un nome di variabile in un valore (che rappresenta l'indirizzo a cui la variabile è memorizzata)
- Si può accedere all'indirizzo di una variabile in sola lettura
  - Non è possibile cambiare l'indirizzo a cui una variabile è memorizzata

Puntatori

- È spesso necessario memorizzare l'indirizzo di una variabile (risultato dell'operatore &) da qualche parte...
- Dove? In una variabile!!! Di che tipo???
  - I tipi di dato visti fino ad ora non sono appropriati...
  - Ci vuole un nuovo tipo di dato! Insieme di definizione: tutti i possibili indirizzi di memoria
  - Tipo puntatore
- Puntatore: indirizzo di una variabile ("punta" a tale variabile)

Tipi Puntatore - 1

- Un tipo puntatore per ogni "tipo puntato"
  - Puntatore che rappresenta indirizzi di variabili di un tipo base: nome del tipo base seguito da "*"
  - Esempio: `int *p;` definisce una variabile `p` che può contenere indirizzi di variabili di tipo `int`
- Non si può assegnare ad un puntatore un indirizzo generico
  - Deve essere un indirizzo di variabile "puntabile" di puntatore
  - Esempio: il codice seguente non è corretto

```
int *p;
float f;

p = &f;
```

Tipi Puntatore - 2

- Ancora una volta le cose sarebbero più complesse
  - Tipo "`void *`"
  - Possibilità di cast fra puntatori
    - `p = (int *)&f;`
  - ...
- Ma noi ignoreremo queste complicazioni
  - "Giocare" con i puntatori può essere pericoloso...
  - Manteniamoci ad un uso limitato che è più sicuro

Dereferenziazione di Puntatori

- Se `p` è una variabile puntatore a variabile di tipo T...
- ...Come accedo al valore della variabile puntata?
  - Operatore `*`: data una variabile puntatore, la "trasforma" nella variabile puntata
  - Se `p` è definita come `int *p` e `p = &i`, allora `*p` è sinonimo di `i`

```
int *p;
int i;

p = &i;
*p = 5;
```

  - Ora il valore memorizzato in `i` è 5
- Il passaggio da indirizzo a variabile si chiama dereferenziazione

Ancora su Definizione di Puntatori

- Se `p` è un puntatore a variabili di tipo T, `*p` rappresenta una variabile di tipo T
  - Tale puntatore è definito come T `*p;`
  - Esempio: `int *p;`
- Il tipo di dato contenuto nella definizione di un puntatore `p` indica il tipo del risultato dell'operazione di dereferenziazione `*p`
  - Esempio: `float *fp;` significa che `*fp` rappresenta una variabile di tipo float
- Due possibili modi di leggere una definizione di puntatore
  - "`float *fp;`" significa `fp` è di tipo "`float *`"
  - "`float *fp;`" significa `*fp` è di tipo "`float`"

Inizializzazione di Puntatori

- I puntatori sono variabili come tutte le altre
  - Se definiti come variabili locali, devono essere inizializzati
  - Se definiti come variabili globali, sono automagicamente inizializzati a `NULL`
- Come tutte le variabili, sono inizializzabili nella definizione:

```
int main()
{
   int i;
   int *p = &i;
   [...]
```

```
int main()
{
   /* Errore!!! */
   int *p = &i;
   int i;
   [...]
```