Strutture dati PILA e CODA

Le strutture dati si dividono in interne e astratte.
Quelle astratte sono rappresentazioni dei dati di un problema che rispecchiano le proprietà dei dati e le relazioni usate nella stesura dell'algoritmo. Quelle interne, invece, si presentano quando gli algoritmi, e quindi le strutture astratte, vengono tradotti in una forma eseguibile dal calcolatore. I dati, allora, devono essere memorizzati in una qualche forma all'interno della memoria.

Il vettore e la lista concatenate (o catena) sono due esempi di strutture interne.

Due strutture astratte molto importanti sono la pila e la coda.

PILA:
è una lista lineare (che a sua volta è una struttura astratta molto semplice, ossia un insieme ordinato di oggetti) a lunghezza variabile in cui inserimenti ed estrazioni avvengono dalla stessa parte, detta testa della pila. Si realizza il principio LIFO: Last In First Out. Le pile sono usate in moltissime applicazioni.

CODA:
è una lista lineare a lunghezza variabile in cui l'inserimento viene effettuato ad un estremo (fondo) e l'estrazione all'altro estremo (testa). Si realizza il principio FIFO: First IN First Out.

Un'applicazione della pila è la valutazione di un'espressione in *notazione postfissa*. La notazione a cui siamo abituati per le espressioni si dice "infissa" ed è caratterizzata dal fatto che l'operatore binario è fra i due operandi: 7 + 5. Nella notazione postfissa invece l'operatore si trova dopo i due operandi: 7 5 + (esiste anche la notazione prefissa).

Ad esempio, l'espressione (7 + 10)* 5, in notazione infissa, si trasforma in notazione postfissa nella seguente espressione: 7 10 + 5 * (l'asterisco è il simbolo per la moltiplicazione). Per scoprire quale è il risultato si procede così: leggo il 7, leggo il 10, poi incontro il +, quindi eseguo la somma fra 10 e 7, il risultato lo considero come l'operando che sostituisce 7 10 +, poi trovo il 5, poi trovo l'asterisco, quindi eseguo la moltiplicazione fra il 5 e l'operando di prima.

Una pila può essere utilizzata per calcolare il risultato di un'espressione in forma postfissa (supponendo che sia corretta). L'algoritmo è il seguente:

1. se l'espressione è non vuota, leggere il prossimo simbolo dell'espressione; *(la prima volta sarà letto il primo simbolo)*
2. se l'espressione è vuota, vai a 8;
3. se è un operando, inserirlo nella pila e tornare a 1.;
4. eseguire due estrazioni dalla pila; *(se siamo qui il simbolo letto è un operatore)*
5. calcolare il risultato applicando l'operatore appena letto;
6. inserire il risultato nella pila;
7. torna a 1;
8. la pila contiene il risultato finale. STOP.

Come esempio di altra applicazione della pila, descriviamo una procedura per controllare se un'espressione in notazione infissa risulta o meno ben parentesizzata.
Supponiamo che *a* sia il vettore che contiene le parentesi dell'espressione nell'ordine con cui esse si trovano da sinistra verso destra e che esso contenga *n* elementi. La procedura prevede l'utilizzo di una pila *S.*

La procedura può essere descritta con un ciclo `for` che prende in esame tutti i simboli contenuti in *a.* Se il simbolo processato è una parentesi aperta allora esso viene inserito in *S,* altrimenti si analizza la pila. Se essa è vuota, l'algoritmo termina segnalando un errore (si è trovata una parentesi chiusa di troppo rispetto a quelle aperte incontrate fino ad ora), altrimenti si esegue una estrazione della pila.
Alla fine del ciclo `for` si analizza di nuovo la pila: se essa è vuota allora l'algoritmo termina segnalando che l'espressione è ben parentesizzata, altrimenti l'algoritmo termina segnalando un errore (ci sono troppe parentesi aperte).

```
for i = 1 to n step 1 do{

     if a_i è una parentesi aperta inseriscila in S;

     else{
          if S è vuota segnala errore e termina;
          esegui una estrazione dalla pila;
     }
}

if S è vuota segnala OK e termina;

segnala errore e termina.
```

Infine, presentiamo un algoritmo per trasformare un'espressione infissa nella sua equivalente postfissa. Prima di tutto, nell'espressione infissa ogni operazione binaria deve essere racchiusa tra parentesi. Ad esempio: l'espressione 5 * 4 + 1 deve essere scritta nella forma ((5 * 4) + 1), anche se le parentesi non sarebbero necessarie per la sua comprensione. Vedremo però che sono fondamentali per il funzionamento dell'algoritmo.

- finché l'espressione infissa non è vuota, leggerla da sinistra verso destra e:
  - se si legge un numero (operando), metterlo in output;
  - se si legge un operatore, inserirlo in una pila (PUSH);
  - se si legge una parentesi aperta, ignorarla;
  - se si legge una parentesi chiusa, fare un'estrazione dalla pila (POP) e mettere il risultato dell'estrazione in output.

Alla fine, in output si ottiene l'espressione in notazione postfissa equivalente.

Altre interessanti applicazioni della pile e della coda si hanno nella visita di un albero.

Algoritmo per la visita generica di un albero.

Sia T un albero. Sia S un insieme inizialmente vuoto in cui vengono inseriti i nodi dell'albero.
Un possibile algoritmo per la visita generica di T è il seguente:

1. inserire la radice di T in S,
2. finché S è non vuoto,
    2.1 estrarre un elemento da S;
    2.2 scrivere il contenuto dell'elemento in output;
    2.3 inserire in S tutti i figli dell'elemento estratto;
3. STOP

Notare che nell'algoritmo sopra non si specifica quale elemento estrarre né in che modo inserire i figli in S.
Se S è una pila, l'elemento estratto sarà per forza quello che è stato inserito per ultimo. Si ottiene quella che si chiama *visita in profondità.*
Se S è una coda, invece si ottiene la *visita in ampiezza*.

Implementazione di una pila tramite un vettore

Sia *a* il vettore che ospita la pila, di dimensione *n.*
Sia *top* una variabile intera che memorizza l'indice del primo posto libero nel vettore.
Notare che possono essere memorizzati al massimo *n - 1* elementi (perché?).

- Inserimento di un elemento *x* in *a* (**push**):

```
if (top == n)  //si controlla se il valore di top sia o meno n.
     segnala "pila piena" e termina;

a_top = x;        // si assegna alla cella di indice top il valore x.
top = top + 1; // si incrementa il valore di top di 1.
```

- estrazione dalla pila (**pop**):

```
if (top == 1)
     segnala "pila vuota" e termina;

sia estr una variabile intera; //dichiarazione di variabile.
estr = a_top - 1 ;
top = top – 1;   // il valore di top deve essere decrementato di 1.
return estr;    /* il valore di estr deve essere comunicato
all'esterno dell'algoritmo, o meglio, restituito al chiamante.*/
```

- lettura del prossimo estratto, senza eseguire l'estrazione (**top**):

```
if (top == 1);
     segnala "pila vuota" e termina;

return  a_top – 1 ;
```

Implementazione di una coda tramite un vettore.

In questa versione il vettore viene ricompattato dopo ogni estrazione. Sia N la dimensione del vettore. Gli elementi vengono mantenuti nel vettore da sinistra verso destra, in maniera consecutiva, Sia R l'indice nel vettore dove avverrà il prossimo inserimento. Sia *a* il vettore che ospita la coda.

- inserimento di un elemento *x* in *a* (**enQueue**):

```
if (R == N)
     segnala coda piena e termina;

a[R] = x;
R = R + 1;
```

estrazione di un elemento dalla coda (**deQueue**):

```
if(R == 1)
     segnala pila vuota e termina;

estr = a[1];

/*
con il seguente ciclo for gli elementi, tolto il primo, vengono
spostati di una posizione verso sinistra in modo da ricompattare
il vettore.
*/
for(i = 1 to R - 1 by 1)
     a[i]=a[i + 1];

R = R - 1;
return estr;
```

Il ciclo for rende il costo dell'operazione di estrazione dipendente dal numero di elementi presenti nella coda. Per evitare ciò, nella prossima versione, invece, si simula la coda con un vettore ma non viene eseguito il ricompattamento. Oltre ad N e R, con lo stesso significato di prima, è necessaria un'altra variabile F che memorizza l'indice in cui avverrà la prossima estrazione.

- inserimento di un elemento *x* in *a* (**enQueue**):

```
if (R == N)
     segnala coda piena e termina;

a[R] = x;
R = R + 1;
```

- estrazione di un elemento dalla coda (**deQueue**):

```
if(R == F)
     segnala pila vuota e termina;

estr = a[F];
```

```
F=F+1;
return estr;
```

La versione sopra ha il difetto che può presentarsi il caso in cui F e R siano indici prossimi ad N (cioè verso la fine del vettore) e quindi sono permessi soltanto altri pochi inserimenti poiché quando F diviene uguale ad R la coda viene segnalata piena. A ben guardare, però la prima parte del vettore (da 1 a F – 1) risulta libera e potrebbe essere sfruttata per ulteriori inserimenti.
Nella versione seguente si sfrutta l'osservazione sopra e si considera il vettore come una struttura circolare, ossia tale che all'ultima casella segua logicamente la prima. Quindi la variabile R, una volta che assume il valore N, se viene incrementata di 1 assumerà il valore 1. Lo stesso vale per F.

- inserimento di un elemento *x* in *a* (**enQueue**):

```
if (R == N)
     segnala coda piena e termina;

a[R] = x;
R = (R + 1) mod N;
/*
mod è l'operatore che calcola il resto della divisione fra due
interi positivi.
*/
```

- estrazione di un elemento dalla coda (**deQueue**):

```
if(R == F)
     segnala pila vuota e termina;

estr = a[F];
F = (F + 1) mod N;
return estr;
```