Azzolini Riccardo 2018-12-11

Strutture dati dinamiche

1 Strutture dati

Le **strutture dati** sono strutture per memorizzare e organizzare dati.

Una struttura dati si dice **dinamica** se la sua organizzazione evolve dinamicamente.

Esistono varie strutture dati (pile, code, insiemi, liste, alberi, ecc.). La scelta di quella da utilizzare dipende principalmente dalle operazioni che dovranno essere effettuate.

2 Pila

Una **pila**, o **stack**, è una struttura **LIFO** (*Last In First Out*): è possibile aggiungere o eliminare un elemento solo in cima.

Le sue operazioni sono:

push: aggiunge un elemento in cima

pop: preleva ed elimina l'elemento in cima

empty: controlla se la pila è vuota

2.1 Metodi fondamentali

- `public void push(E o)`
  Aggiunge `o` in cima alla pila.
- `public E pop()`
  Restituisce l'elemento in cima e lo elimina dalla pila. Se la pila è vuota, solleva un'eccezione non controllata `EmptyStackException`.
- `public boolean empty()`
  Restituisce `true` se la pila è vuota, altrimenti `false`.

2.2 Definizione di `EmptyStackException`

```
public class EmptyStackException extends RuntimeException {}
```

2.3 Implementazione di una pila di `Object` con un array

```
public class Pila {
    private static final int SIZE = 10;
    private Object dati;
    private int top; // Indica la prima posizione libera

    public Pila() {
        dati = null;
        top = 0;
    }

    public void push(Object o) {
        dati[top++] = o;
    }

    public Object pop() {
        if (top == 0)
            throw new EmptyStackException();
        else
            return dati[--top];
    }

    public boolean empty() {
        return top == 0;
    }
}
```

2.4 Implementazione generica con una struttura dati dinamica

L'implementazione basata su un array ha una dimensione massima fissa.

Un'implementazione che non abbia questo limite deve creare spazio per la pila dinamicamente, man mano che è necessario, cioè a ogni push, e rilasciarlo quando non è più richiesto, dopo ogni pop.

Una struttura con questo comportamento è la **lista concatenata**: un insieme di **nodi**, collegati tra loro da dei riferimenti. Ogni nodo contiene:

- un'**informazione** (elemento della lista)
- un **riferimento** (**puntatore**) al nodo successivo, o `null` se è l'ultimo

La classe che rappresenta un nodo è definita in modo ricorsivo (contiene un riferimento a un oggetto della stessa classe):

```
class NodoStack {
    E dato;
    NodoStack pros;
}
```

Per implementare una pila con una lista concatenata, è sufficiente inserire e rimuovere sempre il primo nodo.

```
public class Stack<E> {
    private NodoStack cima;

    private class NodoStack {
        E dato;
        NodoStack pros;
    }

    public Stack() {
        cima = null;
    }

    public boolean empty() {
        return cima == null;
    }

    public void push(E o) {
        NodoStack t = new NodoStack();
        t.dato = o;
        t.pros = cima;
        cima = t;
    }

    public E pop() {
        if (cima == null) {
            throw new EmptyStackException();
        } else {
            E risultato = cima.dato;
            cima = cima.pros;
            // Il nodo rimosso diventa inaccessibile
            // e sarà poi cancellato dal garbace collector
            return risultato;
        }
    }
}
```

3 Coda

Una **coda** è una struttura **FIFO** (*First In First Out*): il primo elemento che può essere prelevato è quello inserito per primo.

Come la pila, una coda può essere implementata in modo dinamico con una lista concatenata, effettuando però gli inserimenti alla fine della lista (ma sempre prelevandoli dall'inizio).

Il modo più semplice per inserire un elemento alla fine è percorrere l'intera lista, fino ad arrivare all'ultimo nodo, e modificare il riferimento di quest'ultimo in modo che punti a un nuovo nodo. Questa tecnica, però, richiede molto tempo se la lista è lunga. Una soluzione migliore è memorizzare due riferimenti: uno al primo e uno all'ultimo nodo, che può così essere raggiunto direttamente. Se la coda è vuota, entrambi i riferimenti sono `null`.

```
public class CodaVuotaException extends RuntimeException {}

public class Coda<E> {
    private NodoCoda primo, ultimo;

    private class NodoCoda {
        E dato;
        NodoCoda pros;
    }

    public Coda() {
        primo = ultimo = null;
    }

    public void aggiungi(E x) {
        NodoCoda t = new NodoCoda();
        t.dato = x;
        t.pros = null;
        if (primo == null) {
            primo = ultimo = t;
        } else {
            ultimo.pros = t;
            ultimo = t;
        }
    }

    public E preleva() {
        if (primo == null) {
            throw new CodaVuotaException();
```

```
        } else {
            E risultato = primo.dato;
            primo = primo.pros;
            if (primo == null)
                ultimo = null;
            return risultato;
        }
    }

    public boolean vuota() {
        return primo == null;
    }
}
```