PROGETTO ALBERO BINARIO DI RICERCA

Gaia Assunta Bertolino
Mat. 209507 A.A. 2020/2021
Corso di Programmazione orientata agli oggetti

Organizzazione del progetto:
Il progetto è stato realizzato attraverso una sola classe che implementa l'interfaccia CollezioneOrdinata<T> e in cui vengono ripresi vari metodi della classe ABR implementata a lezione. Di conseguenza, nella seguente relazione saranno trattati solamente i metodi modificati o di nuova implementazione.

class AlberoBinarioDiRicerca

La classe implementa, diversamente dalla classe ABR, uno Stack<Nodo<T>> dove vengono raccolti i nodi sbilanciati a seguito di una operazione di add o remove e, inoltre, i seguenti metodi e classi:

- **private static class Nodo<E> ->** classe statica che definisce il tipo di oggetto nodo. Al suo interno sono definite le variabili d'istanza che identificano il contenuto di tipo E, i nodi figli e le cardinalità destra e sinistra (cioè i numeri di nodi che si possono contare percorrendo i sottoalberi destro e sinistro rispettivamente)
- **public boolean bilanciato(Nodo<T> radice) ->** metodo che verifica se un albero è bilanciato confrontando le cardinalità destra e sinistra; se differiscono per più di una unità, l'albero non è bilanciato. Il meccanismo è penato per funzionare se richiamato ricorsivamente a partire dalle radici agli ultimi livelli di un albero
- **private Nodo<T> insert(Nodo<T> radice, List<T> lista, int inizio, int fine) ->** metodo che viene richiamato dalla add e dalla remove per ribilanciare l'albero. Infatti, una volta aggiunto o rimosso un elemento, se risulta sbilanciato l'albero, viene richiamato questo metodo che ha come uno degli argomenti la lista ordinata dell'albero stesso. Il metodo applica il meccanismo della ricerca binaria dividendo la lista e facendo corrispondere ad ogni elemento a metà la radice di un sottoalbero, richiamando poi in maniera ricorsiva il metodo stesso sia sulla metà destra che sinistra della lista e valutando opportunamente le cardinalità di ciascun nodo in base alle cardinalità dei propri figli.
- **public void add(T x) ->** metodo pubblico che richiama a sua volta il metodo privato *add(Nodo<T> radice, T x)* e verifica poi se l'albero ottenuto è sbilanciato controllando se lo stack dedicato alle radici di sottoalberi sbilanciati non sia vuoto; in caso affermativo, richiama il metodo *insert(Nodo<T> radice, List<T> lista, int inizio, int fine)*
- **private Nodo<T> add(Noto<T> radice, T x) ->** metodo privato richiamato da *add(T x)* e che segue lo stesso meccanismo dell'addizione implementata a lezione. L'unica differenza consiste nel conteggio della cardinalità di ciascun nodo che viene incrementata ad ogni aggiunta rispettivamente nel contatore del ramo sinistro o destro. Inoltre, differisce gestisce i nodi di sottoalberi sbilanciati (valutati ogni volta che si è conclusa una chiamata ricorsiva) per cui ogni radice di un sottoalbero sbilanciato viene aggiunta allo stack creato appositamente. Al termine di questo metodo, lo sbilanciamento viene gestito come spiegato prima dalla parte restante del codice del metodo privato
- **public void remove(T x) ->** metodo pubblico che ha funziona come *add(T x)* e cioè richiama il metodo privato *remove(Nodo<T> radice, T x)* e gestisce l'eventuale sbilanciamento richiamando *insert(Nodo<T> radice, List<T> lista, int inizio, int fine)*

- **private Nodo<T> remove(Nodo<T> radice, T x) ->** metodo privato che viene richiamato da *remove(T x)* e differisce dal metodo sviluppato a lezione solo per il conteggio della cardinalità (questa volta gestita decrementando quella relativa al ramo in cui viene operata la remove) e per l'aggiunta nello stack dedicato di tutti i nodi che sono radici di alberi sbilanciati. Come per l'add, lo sbilanciamento è gestito dalla parte finale del metodo pubblico
- **private class ABRIterator implements Iterator<T> ->** la classe fornisce una implementazione dell'iteratore basata su uno stack in cui all'inizio vengono salvati i nodi del ramo sinistro della radice. L'implementazione è tale che l'elemento in cima è il minore di tutto l'albero e, nel momento in cui viene fatta una *next()* per prelevarlo, viene aggiunto allo stack il ramo sinistro del figlio destro, così da mettere in atto un meccanismo per cui si fa la *pop()* sempre dell'elemento minore dell'albero. La *remove()* è implementata in modo da richiamare la *remove(T x)* della classe AlberoBinarioDiRicerca; poiché tale chiamata potrebbe comportare uno sbilanciamento e, dunque, che l'albero venga modificato per ribilanciarlo, allora viene ricalcolato lo stack dell'iteratore sulla base dell'albero ottenuto in modo che l'iteratore stesso non diventi inutilizzabile.
- **public AlberoBinarioDiRicerca<T> copy() ->** metodo pubblico che opera la copia di un albero invocando un metodo privato. Il metodo privato opera la copia ricorsiva di un albero aggiungendo, volta per volta, la info della radice di un sottoalbero all'albero copia
- **public void build(T[] a) ->** metodo che riempie un array passato come argomento con la lista ordinata degli elementi dell'albero. Una implementazione senza l'uso dell'ArrayList e del metodo *inOrder()* potrebbe implementare un meccanismo di ordinamento simile all'*inOrder()* ma basato appunto su array
- **public int altezza() ->** metodo ricorsivo che richiama un metodo privato per il calcolo del numero di archi lungo il percorso più lungo dell'albero. Ricordando comunque che l'altezza ha un valore massimo pari a log base 2 del numero di nodi, il metodo lo calcola percorrendo la dove la cardinalità è maggiore e opera questa scelta in maniera ricorsiva ad ogni nodo
- **public static void main(String[] args) ->** il metodo main mostra un semplice uso della classe aggiungendo e rimuovendo dei numeri (dunque T = Integer). Al termine è mostrato anche come, tramite l'iteratore e un ciclo di while, sia possibile svuotare l'albero