Strutture Dati, Algoritmi e Complessità

Esercitazione del 21 Maggio 2024

Scopo di questa esercitazione è realizzare una struttura dati per gestire un albero binario di ricerca. E' possibile svolgere l'esercitazione sia in C che Java.

Rappresentazione di un BST

Un albero binario di ricerca (Binary Search Tree o BST) contiene un insieme di coppie (chiave, valore), e supporta sia inserimento di un elemento arbitrario sia ricerca di elementi per chiave, ed eventuale rimozione. L'insieme delle chiavi è totalmente ordinato. In questa esercitazione assumeremo per semplicità che le chiavi siano degli interi positivi, mentre i valori possono assumere un tipo generico (di tipo void * se
C).
Un BST soddisfa la seguente proprietà: dati tre nodi u, v e w tali che u è nel sottoalbero sinistro di v e w è nel sottoalbero destro di v, allora $key(u) \leq key(v) \leq key(w)$. L'attraversamento in ordine (un nodo è visitato dopo il suo sottoalbero sinistro e prima del suo sottoalbero destro), di conseguenza, visita le chiavi in ordine non decrescente. Utilizzeremo il BST come una mappa, ovvero *non si hanno mai due valori distinti con la stessa chiave*.

Per quanto riguarda l'albero binario, utilizzeremo una (variante della) rappresentazione basata su riferimenti. In questa rappresentazione, il generico nodo dell'albero memorizza:

- la chiave (di tipo intero) associata al nodo
- il valore (generico) associato al nodo
- il puntatore (riferimento) al sottoalbero sinistro
- il puntatore (riferimento) al sottoalbero destro

Task 1

Specifiche. Scrivere un modulo C con intestazione `bst.h` (rispettivamente una classe generica Java `BST.java`) contenente le seguenti funzioni (metodi di classe se Java):

- una funzione/costruttore che costruisce un BST che contiene un unico nodo (k, v):

```
bst * bst_new ( int k , void * v );

public BST ( int k , V v );
```

- una funzione/metodo che restituisce l'unico valore associato ad una chiave k:

```
void * bst_find ( bst * t , int k );

public V find ( int k );
```

  La ricerca non deve visitare tutto l'albero ma seguire lo schema della ricerca binaria. Il metodo deve avere costo $O(h)$, dove h è l'altezza corrente del BST.

- una funzione/metodo che inserisce la coppia $(k, value)$ nel punto appropriato del BST:

  ```
  void bst_insert ( bst * t , int k , void * value );

  public void insert ( int k , V value );
  ```

  Se esiste già un valore associato alla chiave k, il metodo sovrascrive il vecchio valore. Il metodo deve avere costo $O(h)$, dove h è l'altezza corrente del BST.
- una funzione/metodo che restituisce la chiave minima contenuta dal BST:

  ```
  int bst_find_min ( bst * n );

  public int findMin ();
  ```

- una funzione che elimina il nodo con la chiave minima contenuta dal BST:

  ```
  void bst_remove_min ( bst * n );

  public void removeMin ();
  ```

- una funzione/metodo che elimina dal BST il nodo con chiave k (se esiste):

  ```
  void bst_remove ( bst * t , int k );

  public void remove ( int k );
  ```

  Il metodo deve avere costo $O(h)$, dove h è l'altezza corrente del BST.
- (solo in C) una funzione che elimina il BST:

  ```
  void bst_delete ( bst * h );
  ```

- una funzione/metodo che stampa in stdout una *qualsiasi* rappresentazione dell'albero:

  ```
  void bst_print ( bst * h );

  public void printf ();
  ```

Si proceda a testare il codice sviluppato utilizzando `driver.c` (rispettivamente `Driver.java`) passando come argomento `bst`. Si consiglia di implementare i nodi come classe interna di `BST.java` (analogamente in C):

```
    private class Node<V> {
        private int key;
        private V value;
        private Node<V> left;
        private Node<V> right;

        public Node(int key, V value) {
            this.key = key;
            this.value = value;
        }
    }
```

```
typedef struct _bst_node {
    int key;
    void * value;
    struct _bst_node * left;
    struct _bst_node * right;
} _bst_node;
```

Task 2. Trovare il predecessore di una chiave

Un'operazione ausiliaria utile nel tipo di dati BST consiste nel trovare, data una chiave, il nodo che più si "avvicina" a tale chiave, senza superarla. Più precisamente, si vuole considerare la funzione `predecessor`, definita in questo modo: `predecessor(k)` è il nodo del BST avente la chiave maggiore possibile tra tutte
quelle minori di k. Il calcolo di `predecessor(k)` in un BST può essere effettuato sfruttando le seguenti proprietà:

- se k è minore o uguale della chiave nella radice, `predecessor(k)` è nel sottoalbero sinistro.
- se k è maggiore della chiave nella radice, `predecessor(k)` è nel sottoalbero destro se il sottoalbero destro contiene qualche chiave $\leq k$; altrimenti `predecessor(k)` è la radice.

Specifiche. Estendere il codice sviluppato nel precedente esercizio aggiungendo la funzione (metodo):

```
int bst_predecessor ( bst * t , int k );

public int predecessor ( int k );
```

Si proceda a testare il codice sviluppato utilizzando `driver.c` (rispettivamente `Driver.java`) passando come argomento `predecessor`.

Task 3. Testo albero BST

Dato un albero binario, questo esercizio richiede di sviluppare un algoritmo per verificare se esso soddisfi la proprietà di BST.

Specifiche. Estendere il codice presente in `tree.c` (`Tree.java`) implementando la funzione (metodo):

```
int tree_is_bst ( bst * t );

public boolean isBST ();
```

Tale funzione restuisce `1` (`true`) se l'albero binario è un BST , `0` (`false`) altrimenti.

Si proceda a testare il codice sviluppato utilizzando `driver.c` (rispettivamente `Driver.java`) passando come argomento `tree`.

Task 4. Test albero bilanciato

Un albero binario si dice *bilanciato* se *per ogni* nodo, le altezze dei sotto-alberi sinistro e destro differiscono di *al più di 1. Dato un albero binario, questo esercizio richiede di sviluppare un algoritmo per testare se tale risulta bilanciato in altezza.

Specifiche. Estendere il codice sviluppato nel precedente esercizio aggiungendo la funzione (metodo):

```
int tree_is_balanced ( bst * t );

public boolean isBalanced ();
```

Tale funzione restuisce `1` (`true`) se l'albero binario è bilanciato, `0` (`false`) altrimenti.

Si proceda a testare il codice sviluppato utilizzando `driver.c` (rispettivamente `Driver.java`) passando come argomento `balanced`.

Task 5. Test albero AVL (per casa)

Un albero binario di ricerca è un AVL (dai nomi di Adelson-Velsky e Landis che lo proposero) se è bilanciato. Dato un albero binario, questo esercizio richiede di sviluppare un algoritmo per testare se tale albero soddisfi le proprietà di un AVL. Si noti che devono essere verificate due proprietà: (1) l'albero è un BST, (2) l'albero è bilanciato. *Tuttavia occorre implementare questo algoritmo effettuando un'unica visita sull'albero.*

Specifiche. Estendere il codice sviluppato nel precedente esercizio aggiungendo la funzione (metodo):

```
int tree_is_avl ( bst * t );

public boolean isAVL ();
```

Tale funzione restuisce `1` (`true`) se l'albero binario è un AVL, `0` (`false`) altrimenti.

Si proceda a testare il codice sviluppato utilizzando `driver.c` (rispettivamente `Driver.java`) passando come argomento `avl`.