UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dispense del Corso di Laboratorio di Fondamenti di Informatica II e Lab

Esercitazione 09: Alberi

Ultimo aggiornamento: 15/05/2019

Introduzione 1/2

- Date le seguenti definizioni di tipi:

```
typedef int element;
typedef struct tree_element {
        element value;
        struct tree_element *left, *right;
} node;
typedef node* tree;
```

- N.B. normalmente non è una buona idea (anzi è proprio pessima!) mascherare dei puntatori dietro ad alias, come viene fatto in questo caso con la definizione del tipo `typedef node* tree`. Durante il corso utilizzeremo comunque questa sintassi a scopo puramente didattico.

Introduzione 2/2

- E date le implementazioni delle seguenti primitive:
  - `tree EmptyTree();`
  - `tree ConsTree(const element *e, tree l, tree r);`
  - `bool IsEmpty(tree t);`
  - `element *GetRoot(tree t);`
  - `tree Left(tree t);`
  - `tree Right(tree t);`
  - `bool IsLeaf(tree t);`
- Trovate le dichiarazioni e le rispettive implementazioni delle primitive sopra elencate nei file `tree_int.h` e `tree_int.c` su dolly. Al link http://imagelab.ing.unimore.it/alberi/html/tree__int_8c.html trovate anche la loro documentazione, che vi aiuterà nella risoluzione degli esercizi.

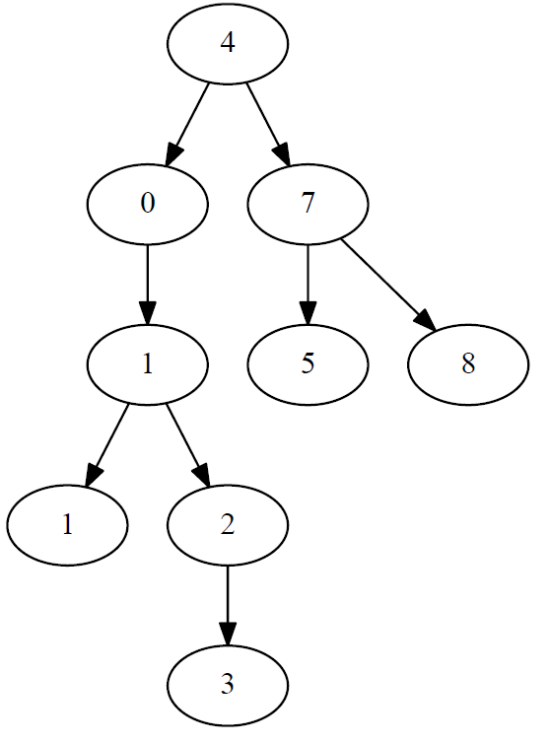
Ripasso: Alberi BST 1/2

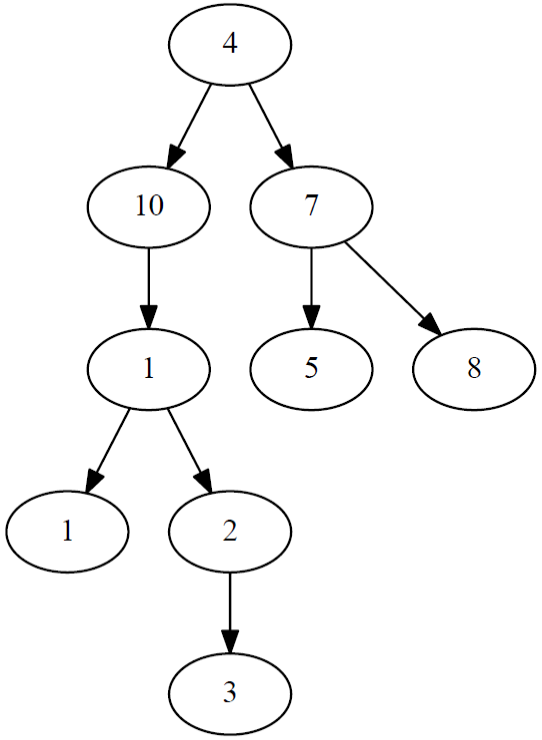
Un albero binario di ricerca (o Binary Search Tree BST) deve soddisfare le seguenti tre proprietà:

- Il sottoalbero sinistro di un nodo contiene soltanto i nodi con valori (chiavi) minori o uguali della chiave del nodo
- Il sottoalbero destro di un nodo contiene soltanto i nodi con valori (chiavi) maggiori o uguali della chiave del nodo
- Il sottoalbero destro e il sottoalbero sinistro devono essere entrambi due alberi binari di ricerca.

Ripasso: Alberi BST 2/2

L'albero di sinistra, ad esempio, è un albero BST perché rispetta tutti i vincoli elencati nella slide precedente. Quello di destra, invece, è un albero binario non BST in quanto la chiave 10 viola le proprietà.





Alberi: Inserimento in un BST

- Esercizio 1 (BstInsert):

Nel file `alberi.c` si implementi la definizione della seguente funzione:

```
tree BstInsert(const element *e, tree t);
```

La funzione prende in input l'indirizzo di un nuovo elemento `e` un albero binario di ricerca `t`, `e` deve aggiungere un nuovo nodo all'albero (con chiave uguale ad e), facendo in modo che siano rispettate le proprietà BST. La funzione deve quindi ritornare l'albero risultante. Se l'albero t è vuoto, la funzione deve costruire un albero costituito da un solo nodo (di valore `e`) e ritornarlo. Si testi la funzione con un opportuno `main` di prova.

- Esercizio 2 (BstInsertRec):

Si implementi la funzione precedente in maniera ricorsiva.

Alberi: Conta dominanti

- Esercizio 3 (ContaDominanti):

Un nodo di un albero è detto dominante se non è una foglia e se contiene un valore maggiore della somma dei valori contenuti nei suoi due figli (destro e sinistro). Nel file `alberi.c` si implementi la definizione della funzione `ContaDomininanti`:

```
int ContaDominanti(tree t);
```

La funzione prende in input un albero binario `t` e deve restituire il numero di nodi dominanti in esso contenuti. Se l'albero è vuoto o non contiene nodi dominanti la funzione deve ritornare `0`.

Si scriva un opportuno `main` di prova per testare la funzione.

Quanti sono i nodi dominanti in un albero BST?

Alberi: Massimo

- Esercizio 4 (BstMax):

Nel file `alberi.c` si implementi la definizione della seguente funzione:

```
element* BstMax(tree t)
```

La funzione prende in input un albero BST e ritorna l'indirizzo dell'elemento di valore massimo. Se l'albero è vuoto la funzione deve ritornare NULL. Si scriva un opportuno `main` di prova per testare la funzione.

- Esercizio 5 (Max):

Nel file `alberi.c` si implementi la definizione della seguente funzione:

```
element* Max(tree t)
```

La funzione prende in input un albero binario qualunque e ritorna l'indirizzo dell'elemento di valore massimo. Se l'albero è vuoto la funzione deve ritornare NULL. Si scriva un opportuno `main` di prova per testare la funzione.

Alberi: Minimo

- Esercizio 6 (BstMin):

Nel file `alberi.c` si implementi la definizione della seguente funzione:

```
element* BstMin(tree t)
```

La funzione prende in input un albero BST e ritorna l'indirizzo dell'elemento di valore minimo. Se l'albero è vuoto la funzione deve ritornare `NULL`. Si scriva un opportuno `main` di prova per testare la funzione.

- Esercizio 7 (Min):

Nel file `alberi.c` si implementi la definizione della seguente funzione:

```
element* Min(tree t)
```

La funzione prende in input un albero binario qualunque e ritorna l'indirizzo dell'elemento di valore minimo. Se l'albero è vuoto la funzione deve ritornare `NULL`. Si scriva un opportuno `main` di prova per testare la funzione.

Alberi: Eliminazione 1/2

- Esercizio 8 (Delete):

Nel file `alberi.c` si implementi la definizione della seguente funzione:

```
tree Delete(tree t, const element *e)
```

La funzione prende in input un albero binario qualunque e un elemento, e deve eliminare tutti i nodi dell'albero contenenti la stessa chiave dell'elemento `e` (se presente). Se l'albero `t` è vuoto la funzione deve ritornare un albero vuoto.

Alberi: Eliminazione 2/2

- Esercizio 9 (BstDelete):

Nel file `alberi.c` si implementi la definizione della seguente funzione:

```
tree BstDelete(tree t, const element *e)
```

La funzione prende in input un albero BST e un elemento e deve eliminare tutti i nodi dell'albero contenenti la stessa chiave dell'elemento `e` (se presente), assicurando che le proprietà BST siano rispettate. Se l'albero `t` è vuoto la funzione deve ritornare un albero vuoto.