UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

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

Federico Bolelli, Silvia Cascianelli

Esercitazione 09: Alberi

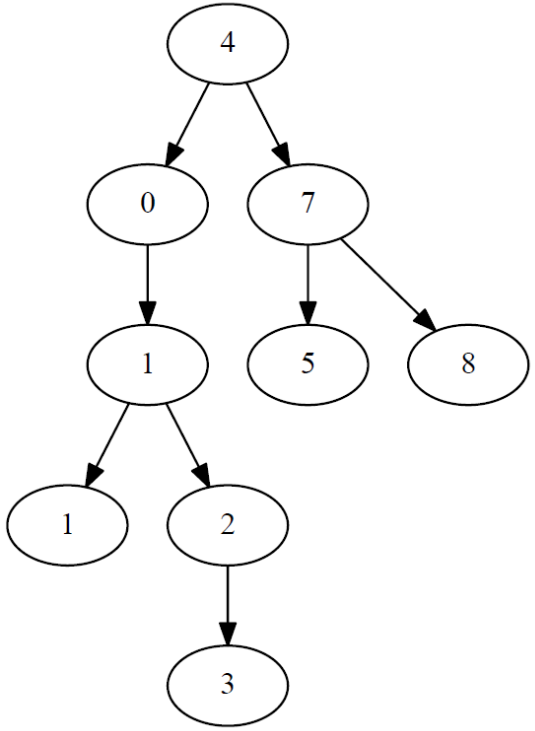
Ultimo aggiornamento: 28/05/2020

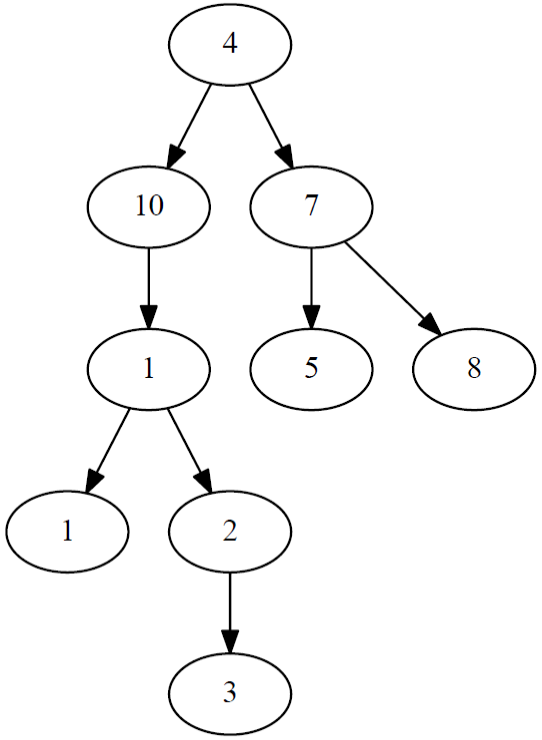
Ripasso: Alberi BST

- Un albero binario di ricerca (o Binary Search Tree BST) deve soddisfare le seguenti tre proprietà:
  1. Il sottoalbero sinistro di un nodo contiene soltanto i nodi con valori (chiavi) minori o uguali della chiave del nodo;
  2. Il sottoalbero destro di un nodo contiene soltanto i nodi con valori (chiavi) maggiori o uguali della chiave del nodo;
  3. Il sottoalbero destro e il sottoalbero sinistro devono essere entrambi due alberi binari di ricerca.

Ripasso: Alberi BST

- 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.





Introduzione

- Siano date le seguenti definizioni:

```
typedef int ElemType;

struct Node
{
    ElemType value;
    struct Node *left;
    struct Node *right;
};
typedef struct Node Node;
```

Introduzione

- Siano date le implementazioni delle seguenti funzioni specifiche per la creazione, eliminazione, confronto, acquisizione da file e scrittura su file di `ElemType` di tipo `int`:

```
- int ElemCompare(const ElemType *e1, const ElemType *e2);
- ElemType ElemCopy(const ElemType *e);
- void ElemDelete(ElemType *e);
- int ReadElem(FILE *f, ElemType *e);
- int ReadStdinElem(ElemType *e);
- void WriteElem(const ElemType *e, FILE *f);
- void WriteStdoutElem(const ElemType *e);
```

Introduzione

- Siano inoltre date le implementazioni delle seguenti funzioni primitive (e non):

```
- Node* CreateEmptyTree(void);
- Node* CreateRootTree(const ElemType *e, Node* l, Node* r);
- bool IsEmptyTree(const Node *n);
- const ElemType* GetRootValueTree(const Node *n);
- Node* LeftTree(const Node *n);
- Node* RightTree(const Node *n);
- bool IsLeafTree(const Node *n);
- void DeleteTree(Node *n);

- void WritePreOrderTree(const Node *n, FILE *f);
- void WriteStdoutPreOrderTree(const Node *n);
- void WriteInOrderTree(const Node *n, FILE *f);
- void WriteStdoutInOrderTree(const Node *n);
- void WritePostOrderTree(const Node *n, FILE *f);
- void WriteStdoutPostOrderTree(const Node *n);
```

Introduzione

- Trovate le dichiarazioni e le definizioni dei tipi di dato e delle funzioni descritte nelle slide precedenti nel repository GitHub al link: https://github.com/prittt/Fondamenti-II
- Leggete attentamente il README, il quale vi spiega come scaricare lo zip contenente i file `tree_int.h` e `tree_int.c` con le implementazioni che vi serviranno per l'esercitazione, e dove trovare la documentazione delle suddette funzioni.
- Consultate la documentazione prima di utilizzare una funzione di cui non conoscete con esattezza il comportamento.
- Attenzione: non dovete implementare voi le funzioni primitive, ma dovete utilizzare quelle che vi vengono fornite.

Alberi: Inserimento in un BST

- Esercizio 1 (BstInsert):

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

```
extern Node* BstInsert(const ElemType *e, Node *n)
```

La funzione deve aggiungere all'albero avente radice nel nodo puntato da n un nuovo nodo di chiave e. La funzione deve garantire che siano rispettate le proprietà BST dopo l'inserimento e ritornare l'albero risultante, ovvero il puntatore al nodo radice. Se l'albero di input è vuoto, la funzione deve ritornare un nuovo albero costituito da un solo nodo con chiave uguale ad e.

- Esercizio 2 (BstInsertRec):

Nel file `insert.c` si implementi la funzione `BstInsertRec()`. La funzione deve avere lo stesso comportamento `BstInsert()` ma deve essere implementata in maniera ricorsiva.

Si scriva un opportuno `main()` di prova per testare le funzioni.

Alberi: Conta dominanti

- Esercizio 3 (CountDominant):

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 `dominant.c si` implementi la definizione della seguente funzione:

```
extern int CountDominant(const Node* t)
```

La funzione prende in input un albero binario (puntatore al nodo radice) 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 (BstTreeMax):

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

```
extern const ElemType* BstTreeMax(const Node *n);
```

La funzione prende in input un albero BST (puntatore al nodo radice) e ritorna l'indirizzo dell'elemento di valore massimo. Se l'albero è vuoto la funzione deve ritornare NULL.

- Esercizio 5 (TreeMax):

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

```
extern const ElemType* TreeMax(const Node *n);
```

La funzione prende in input un albero binario qualunque (puntatore al nodo radice) 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 le funzioni.

Alberi: Minimo

- Esercizio 6 (BstTreeMin):

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

```
extern const ElemType* BstTreeMin(const Node *n);
```

La funzione prende in input un albero BST (puntatore al nodo radice) e ritorna l'indirizzo dell'elemento di valore minimo. Se l'albero è vuoto la funzione deve ritornare NULL.

- Esercizio 7 (TreeMin):

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

```
extern const ElemType* TreeMin(const Node *n);
```

La funzione prende in input un albero binario qualunque (puntatore al nodo radice) 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 le funzioni.

Alberi: Eliminazione

- Esercizio 8 (DeleteBstNode):

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

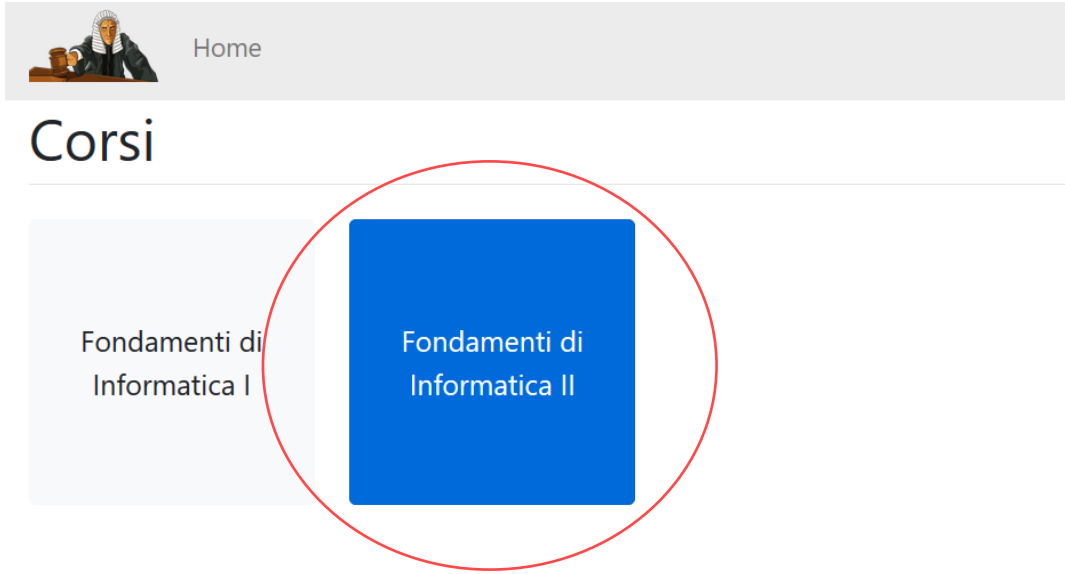
```
extern Node* DeleteBstNode(Node *n, const ElemType *key);
```

La funzione prende in input un albero BST (puntatore al nodo radice) e una chiave. La funzione deve eliminare dell'albero il nodo avente la chiave specificate (se presente), assicurando che le proprietà BST siano rispettate. La funzione deve ritornare l'albero ottenuto dopo l'eventuale modifica.

Si scriva un opportuno `main()` di prova per testare le funzioni.

Modalità di Consegna

- Per questa esercitazione dovrete consegnare gli esercizi 1, 3, 4, 7 e 8 utilizzando un **nuovo** sistema di sottomissione online.
- **Collegatevi al sito https://olj.ing.unimore.it/ e fate il login (in alto a destra) utilizzando le vostre credenziali shibboleth di UNIMORE.**
- Selezionate quindi il corso «Fondamenti di Informatica II»:



Modalità di Consegna

- Selezionate *Esercitazione Alberi,* aprite il link dell'esercizio di cui volete fare la sottomissione e incollate il codice nei box relativi ai rispettivi file. **Non dovete caricare il `main()`.**
- Assicuratevi tuttavia di scrivere il `main()` in Visual Studio per verificare se quello che avete fatto funziona, prima di caricare la soluzione!
- Quando necessario, il codice che sottomettete dovrà opportunamente includere il file delle primitive (`#include "tree_int.h"`).
- **Non caricate sul sistema l'implementazione delle primitive.**
- Il funzionamento del nuovo sistema di sottomissione è, lato utente, del tutto simile a quello vecchio, quindi non dovreste trovare difficoltà.
- **Attenzione: il sistema è ancora in fase di sviluppo/test quindi vi chiediamo di segnalare a federico.bolelli@unimore.it eventuali errori o incongruenze riscontrate.**