Fondamenti di Programmazione 2
(Corso di Laurea in Informatica)
Esame del 13/06/2025

UNIVERSITÀ DELLA CALABRIA

Esercizio 1

Data la seguente porzione di programma rispondere alle domande corrispondenti:

```
void f (int* a) {
    int b = (*a+3);
    int c = *(a+3);
    cout << b << " " << c << endl;
}

int main() {
    int* matricola = new int[6] {//Inserire qui la tua matricola};

    // 1. La seguente istruzione è corretta? Se si, che cosa stampa?
    f(matricola);

    // 2. La seguente istruzione è corretta? Se si, che cosa stampa?
    f(matricola+3);

    // 3. Le seguenti istruzioni sono corrette? Motivare la risposta
    while (!matricola.empty()) matricola.pop_back();

    // 4. Scrivere sul foglio le istruzioni per deallocare la memoria dinamica allocata
    nel programma. Scrivere "NIENTE" se non c'è bisogno di deallocare memoria

    …

    return 0;

}
```

Esercizio 2

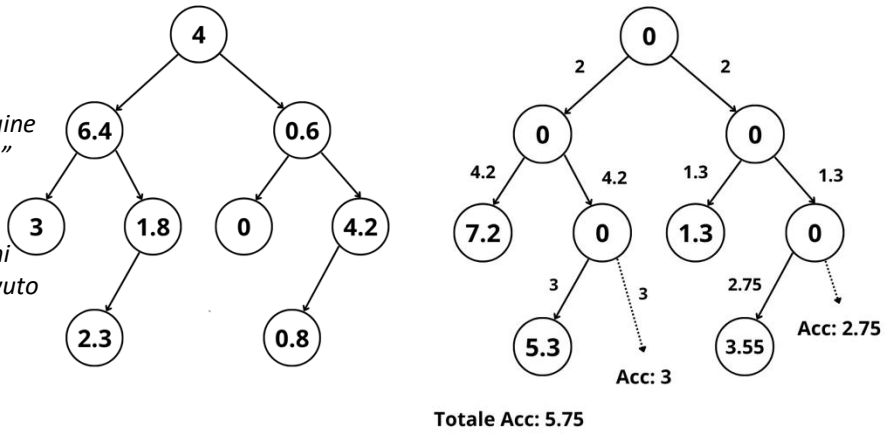
Una compagnia di amici è rappresentata da un vettore di `n` stringhe (che si possono supporre tutte diverse tra loro). Ogni stringa rappresenta il nome di una persona (e contiene solo lettere minuscole dell'alfabeto latino). Il gruppo di amici in questione è solito organizzare serate e cene. Come spesso accade, quando si organizzano questo tipo di attività, alcuni componenti del gruppo anticipano le spese di organizzazione (per qualcuno nello specifico o per tutto il gruppo) per poi riscuotere il denaro dagli amici in un secondo momento. Bisogna tenere traccia di tutte le entrate e uscite per una data serata. A tal fine vogliamo implementare una classe `GestioneDebiti` che tiene traccia dei debiti e dei crediti che ognuno ha con tutti gli altri. La classe riceve in input il vettore degli "amici" e deve esporre almeno i seguenti metodi:

- `void anticipa(string x, string y, float k)` con il quale stiamo dicendo che `x` ha anticipato una spesa per conto di `y` di `k` euro. Se **`y=="*"`** significa che la spesa sostenuta è da attribuire a tutti (l'importo `a` deve quindi essere diviso equamente per tutti gli amici, compreso `x` stesso). Ogni debito generato "verso se stessi" verrà poi automaticamente azzerato.
- `float saldo (string x, string y) const` che restituisce l'importo che `x` deve restituire ad `y` in quel dato momento, considerando anche eventuali debiti che `y` ha con `x`, calcolandone la differenza. Se tale differenza è minore di `0`, `saldo` deve restituire `0`.
- `void paga (string x, string y, float k)` tiene traccia del fatto che `x` ha restituito `k` euro ad `y`. Questa operazione può essere effettuata solo se `saldo(x,y)` è positivo e `k` è minore o uguale a `saldo(x,y)`. Pertanto, il debito residuo di `x` verso `y` sarà `saldo(x,y)-k`, e il debito residuo di `y` verso `x` sarà conseguentemente azzerato.

Esercizio 3

Scrivere una funzione `spremi` che, ricevuto in input un `AlberoB<float> tree`, imposti il valore di ogni nodo non foglia a 0, spostandolo verso i suoi figli, dividendolo equamente tra sinistra e destra. Qualora un nodo non foglia abbia un figlio nullo, la quantità ad esso destinata viene "raccolta" in una variabile che deve essere restituita in output. I nodi foglia conserveranno quindi la somma delle quantità che arrivano dai suoi antenati.

*Esempio: l'input raffigurato nell'immagine a sinistra modifica l'albero come nell'immagine a destra e produce come output "Totale Acc" (che in questo caso equivale a **5.75**).*
Le quantità trasferite sono raffigurate sugli archi nell'immagine a destra. Si noti che ogni nodo trasferisce il suo valore + il valore ricevuto dal suo genitore



La classe `AlberoB<T>` mette a disposizione la seguente interfaccia pubblica (`tree` istanza di `AlberoB<T>`):

- `tree.radice()` restituisce il valore informativo di `tree` (di tipo `T`)
- `tree.figlio(DIR)` restituisce il sottoalbero sinistro (`DIR=SIN`) e destro (`DIR=DES`) di `tree`
- `tree.nullo()` restituisce `true` se `tree` è un albero nullo e `false` altrimenti
- `tree.foglia()` restituisce `true` se `tree` è una foglia e `false` altrimenti
- `tree.modRadice(const T& a)` per modificare il valore informativo di `tree` in `a`

Esercizio 4

Dato un grafo orientato `g`, scrivere una funzione `void esercizio4(…)` che controlli se esiste un cammino che, partendo da un qualsiasi nodo di `g`, passi una sola volta per tutti i nodi del grafo e ritorni nel nodo di partenza tramite un arco diretto dall'ultimo nodo del cammino trovato. Se tale cammino esiste, la funzione deve stampare, in ordine, i nodi che lo compongono. In tutti gli altri casi, la funzione deve stampare "`Impossibile`".

La classe `Grafo` mette a disposizione la seguente interfaccia di metodi costanti (con `g` istanza della classe grafo e `i` e `j` nodi di `g`):

- Il metodo `g.n()` restituisce il numero di nodi di `g`
- Il metodo `g.m()` restituisce il numero di archi di `g`
- Il metodo `g(i,j)` restituisce `true` se esiste un arco dal nodo `i` al nodo `j` e `false` altrimenti