Esame di	
	Algoritmi e strutture dati (parte di Fondamenti di informatica II 12 CFU)
	Algoritmi e strutture dati (V.O., 5 CFU)
	Algoritmi e strutture dati (Nettuno, 6 CFU)

Appello del 5-9-2018 – a.a. 2017-18 – Tempo a disposizione: *4 ore* – somma punti: 35

Istruzioni

Lanciare la macchina virtuale Oracle VirtualBox e lavorare all'interno della cartella `ESAME`, avendo cura di creare all'interno della cartella stessa:

- un file `studente.txt` contenente, una stringa per riga, cognome, nome, matricola, email; in tutto quattro righe, memorizzando il file nella cartella `esame`; è possibile aggiungere una quinta riga contenente eventuali informazioni addizionali che intendi porre all'attenzione del docente;
- una cartella `java.<matricola>`, o `c.<matricola>`, ove al posto di `<matricola>` occorrerà scrivere il proprio numero di matricola, **contenente i file prodotti per risolvere il Problema 2** (in tale cartella si copi il contenuto dell'archivio `c-aux.zip` o `java-aux.zip`); tale cartella va posizionata nella cartella `esame`;
- tre altri file `probl1.<matricola>.txt`, `probl3.<matricola>.txt` e `probl4.<matricola>.txt`, contenenti, rispettivamente, gli svolgimenti dei problemi 1, 3 e 4; i tre file vanno posti nella cartella `esame`.

Attenzione: i simboli < e > usati nei nomi dei file fanno parte del linguaggio dei metadati e **non debbono essere inclusi nei nomi reali**. *È possibile consegnare **materiale cartaceo integrativo** solo contenente **equazioni e disegni**, che verrà esaminato solo a condizione che risulti ben leggibile e a completamento di quanto digitalmente redatto.*

Per l'esercizio di programmazione (Problema 2) è possibile usare qualsiasi ambiente di sviluppo disponibile sulla macchina virtuale. Si raccomanda però di controllare che i file vengano salvati nella cartella `java.<matricola>`, o `c.<matricola>`. Si consiglia inoltre per chi sviluppa in C di compilare da shell eseguendo il comando `make` e poi eseguire `driver` per verificare la correttezza dell'implementazione. Analogamente si raccomanda per chi sviluppa in Java di compilare da shell eseguendo il comando `javac *.java` e poi eseguire `java Driver` per verificare la correttezza dell'algoritmo.

N.B. Le implementazioni debbono essere compilabili. In caso contrario, l'esame non è superato.

Problema 1 Analisi algoritmo

Si considerino i metodi Java di seguito illustrati.

```
// n >= k >= 0
static int bin(int n, int k) {
    if(k == 0 || k == n) return 1;
    if(k == 1 || k == n-1) return n;
    if(k > n-k) k = n-k;
    return prod(n, n-1, 2, k-1);
}
```

```
static int prod(int acc, int p, int q, int c) {
    int r = (acc*p)/q;
    if(c == 1) return r;
    return prod(r, p-1, q+1, c-1);
}
```

Sviluppare, *argomentando adeguatamente* (il 50% del punteggio dell'esercizio sarà sulle argomentazioni addotte), quanto segue:

(a) Determinare il costo temporale asintotico dell'algoritmo descritto da `bin(int,int)` in funzione della dimensione dell'input. [4/30]

(b) Definire il significato di algoritmo *in-place* e, trascurando la memoria impiegata per gestire le ricorsioni runtime, spiegare se `bin(int,int)` è in-place oppure no. [2/30]

Problema 2 Progetto algoritmi C/Java [soglia minima: 5/30]

In questo problema si fa riferimento a *grafi* con archi pesati. I grafi sono rappresentati attraverso *liste di incidenza*, ovvero liste di adiacenza in cui appaiono, per ogni nodo, gli archi incidenti invece dei nodi adiacenti. A ciascun nodo u è dunque associata una lista collegata contenente degree(u) elementi, ciascuno dei quali descrive un arco (incidente u); inoltre ad ogni nodo sono associati un numero progressivo (a iniziare da 0) automaticamente assegnato dalla primitiva di inserimento nodo ed un nome (stringa) fornita come input alla

stessa primitiva. Similmente, a ciascun arco è automaticamente assegnato un numero progressivo (a iniziare da 0). Nodi ed archi sono rappresentati dalle classi/strutture `GraphNode`/`graph_node` e `GraphEdge`/`graph_edge`; il grafo è rappresentato dalla classe/struttura `Graph`/`graph`.

La gestione delle liste (di archi e/o di nodi) fa uso del tipo `linked_list` (C) o la classe `java.util.LinkedList<E>` (Java); per tali tipi sono già disponibili, nel codice sorgente fornito, o nella classe `java.util.LinkedList<E>`, le primitive di manipolazione. Si noti che gli elementi delle liste sono dei contenitori che includono puntatori ai nodi/archi del grafo.

(A fine testo è disponibile un disegno che mostra la rappresentazione di un semplice grafo di esempio).

Sono inoltre già disponibili le primitive di manipolazione grafo: creazione di grafo vuoto, get lista nodi, inserimento di nuovo nodo, get lista archi incidenti un dato nodo, inserimento nuovo arco, get chiave (progressivo) di un dato nodo/arco, get nome (stringa) di un dato nodo, get opposite di un nodo per un dato arco, cancellazione arco, cancellazione nodo (e relativi archi incidenti), cancellazione grafo, stampa grafo. Per dettagli sulle signature di tali primitive, così come per dettagli su quali porzioni di memoria allocata vengono liberate dalle varie primitive di cancellazione, si rimanda ai file sorgente distribuiti. Si noti che le primitive forniscono un insieme base per la manipolazione di grafi, ma i problemi proposti possono essere risolti usandone solo un sottoinsieme.

Tutto ciò premesso, risolvere al computer quanto segue, in Java o in C.

La ditta ACME Inc. deve realizzare il cablaggio delle città italiane, impiegando un cavo ad alte prestazioni, che è molto costoso. Per questo, ACME Inc. sta cercando un ingegnere in grado di realizzare il lavoro nel modo più efficiente possibile. Date le ottime prestazioni del cavo, è ammissibile che due città siano collegate non direttamente, ma attraverso un numero indefinito di città intermedie. Per aiutare la ACME Inc., è necessario risolvere i seguenti tre quesiti:

2.1 Realizzare una funzione/metodo `getCompleteGraph` che, data una lista di città, ritorna un puntatore al grafo completo avente un nodo per ogni città della lista e, per ogni coppia di nodi del grafo, un arco pesato con la distanza tra le due città. Si noti che una città è rappresentata da un nome (`name`, stringa), ed una coppia di coordinate `x` e `y` (double). Si noti inoltre che nel modulo/classe `GraphServices` viene fornita una funzione/metodo di supporto per calcolare la distanza euclidea di due città.[3.5/30]

2.2 Realizzare una funzione/metodo `getMinHeapEdges` che, dato un grafo pesato, ritorna un puntatore al MinHeap costruito sugli archi del grafo. La priorità è dettata dal peso degli archi. Si noti che è disponibile il modulo/classe `MinHeap` a supporto della gestione del MinHeap. [3/30]

2.3 Realizzare una funzione/metodo `getMST` che, dato un grafo pesato, ritorna una lista degli archi contenuti in un opportuno Spanning Tree del grafo, selezionato con l'obiettivo di minimizzare la quantità di cavo impiegato. Si noti che per la risoluzione di questo punto, è possibile utilizzare i prodotti degli esercizi precedenti. Inoltre, è disponibile il modulo/classe `Partition` che implementa le primitive di Union-Find. [3.5/30]

È necessario implementare le funzioni in `graph_services.c` o i metodi in `GraphServices.java`, identificati dal commento `/*DA IMPLEMENTARE*/`. In tali file è permesso sviluppare nuovi metodi/funzioni ausiliari, se utile. *Non è assolutamente consentito modificare metodi e strutture già implementati.* È invece possibile modificare il file `driver.c`/`Driver.java` per poter effettuare ulteriori test.

Per il superamento della prova al calcolatore è necessario conseguire almeno 5 punti. Si prega di non fare usi di package nel codice Java.

Problema 3 Algoritmi

(a) Descrivere accuratamente (codice o pseudo-codice) l'algoritmo di Horner per determinare il valore di un polinomio in un punto [3/30] e dimostrarne il *costo lineare* [1/30]. [4/30]

(b) Progettare un algoritmo (codice o pseudo-codice) che dati n array di interi già ordinati in senso non decrescente, costruisca e restituisca un nuovo array ordinato in senso non decrescente contenente tutti e soli i valori presenti negli array in input. Non sono ammessi side-effects. Assumere che l'input sia fornito attraverso un array di n riferimenti agli array ordinati; l'output sarà fornito attraverso un riferimento al nuovo array. Analizzare il costo dell'algoritmo.[1] [5/30]

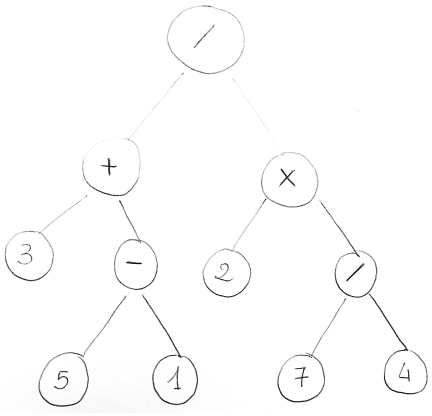
(c) Per descrivere la struttura statica di un programma P viene costruito un grafo diretto i cui nodi sono associati alle funzioni (o metodi) di P e che contiene un arco diretto (u, v) se la funzione u contiene almeno una chiamata alla funzione v. Il main viene associato a un nodo denominato s. Il grafo ammette cappi, ovvero archi del tipo (w, w), nel caso la funzione w sia ricorsiva.

[1] Si noti che in generale trovare il minimo fra n elementi costa $n - 1$ confronti.

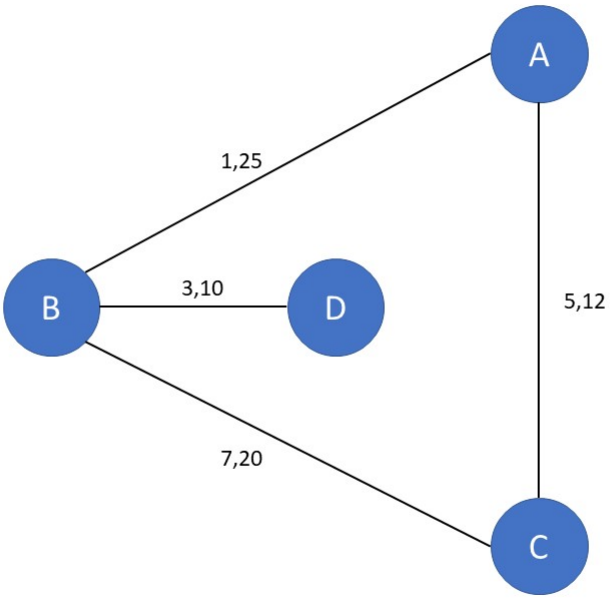
Ciò premesso, progettare un algoritmo (codice o pseudo-codice) che, dato un grafo diretto che descrive un programma P, decide se P è privo di ricorsioni di qualunque tipo: sia quelle dirette (le classiche) che le indirette, vale a dire schemi ricorsivi in cui una funzione t_1 può chiamare una funzione t_2, che a sua volta può chiamare t_3, ecc., fino a chiamare $t_h = t_1$, per $h > 1$. [4/30]

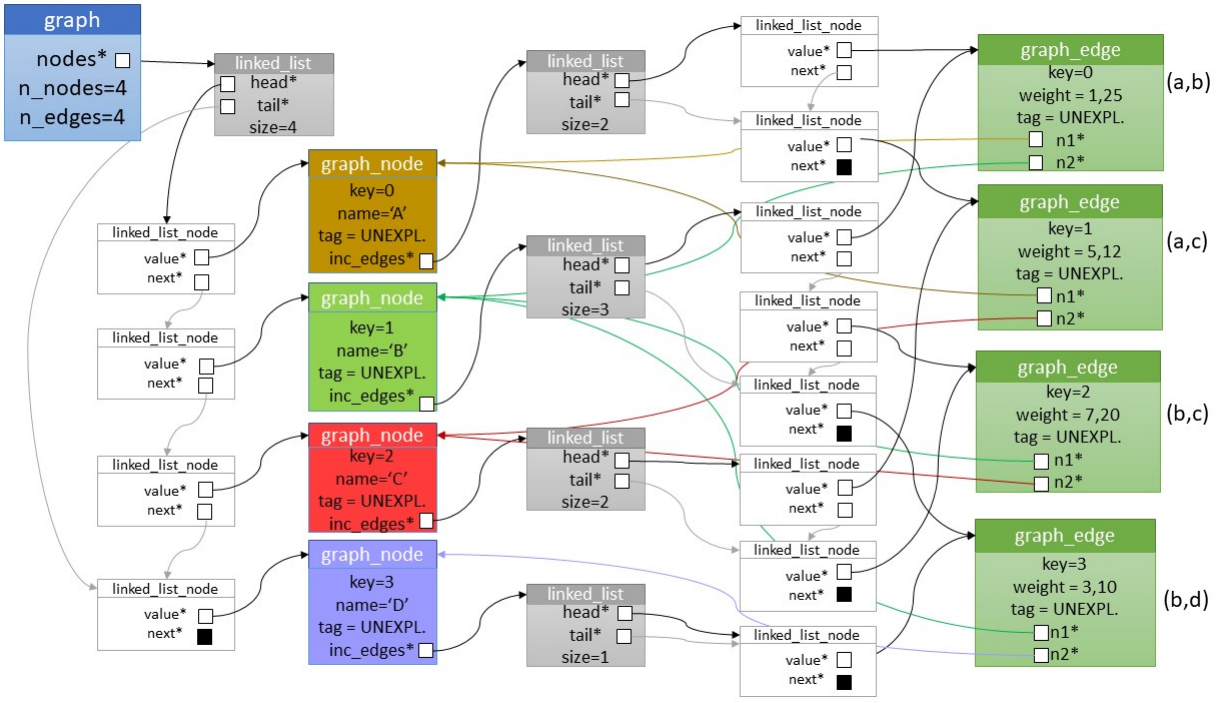
Problema 4 Calcolo e stampa di expression trees

Un expression tree è un albero binario che rappresenta una espressione aritmetica su un insieme di valori. Nel caso in esame prendiamo in considerazione expression trees in cui ciascun nodo interno è associato a un'operazione binaria in $\{+, -, \times, /\}$ su operandi memorizzati nelle foglie come costanti in virgola mobile. Nell'esempio in figura l'expression tree rappresenta l'espressione $(3 + (5 - 1))/(2 \times (7/4))$.



Ciò premesso, si chiede di scrivere un algoritmo (codice o pseudo-codice) che dato un expression tree, stampi su standard output l'espressione aritmetica rappresentata, restituendo attraverso `return` il valore dell'espressione stessa. [6/30]


A
B
C
D
1,25
3,10
5,12
7,20


graph
nodes*
n_nodes=4
n_edges=4
linked_list
head*
tail*
size=4
graph_node
key=0
name='A'
tag = UNEXPL.
inc_edges*
graph_node
key=1
name='B'
tag = UNEXPL.
inc_edges*
graph_node
key=2
name='C'
tag = UNEXPL.
inc_edges*
graph_node
key=3
name='D'
tag = UNEXPL.
inc_edges*
linked_list_node
value*
next*
linked_list
head*
tail*
size=2
linked_list
head*
tail*
size=3
linked_list
head*
tail*
size=2
linked_list
head*
tail*
size=1
graph_edge
key=0
weight = 1,25
tag = UNEXPL.
n1*
n2*
(a,b)
graph_edge
key=1
weight = 5,12
tag = UNEXPL.
n1*
n2*
(a,c)
graph_edge
key=2
weight = 7,20
tag = UNEXPL.
n1*
n2*
(b,c)
graph_edge
key=3
weight = 3,10
tag = UNEXPL.
n1*
n2*
(b,d)