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 17-02-2023 – a.a. 2021-22 – Tempo: 100 minuti – somma punti: 32 - Compito A

Istruzioni

Lanciare la macchina virtuale Oracle VirtualBox e lavorare all'interno della cartella `Esame`. Per prima cosa compilare il file `studente.txt` con le informazioni richieste. In particolare, cognome, nome, matricola, email, esame (vecchio o nuovo), linguaggio in cui si svolge l'esercizio 1 (Java o C). Per quanto riguarda il campo esame (vecchio o nuovo), possono optare per il vecchio esame gli studenti che nel periodo che va dal 2014-15 al 2017-18 (estremi inclusi) sono stati iscritti al II anno.

Nota bene. Per ritirarsi è sufficiente rinominare il file `studente.txt` in `studenteritirato.txt`, *senza* cancellarlo.

Come procedere. Nella cartella `Esame` trovate i) il testo del compito, ii) il file `studente.txt` sopra citato, iii) due sottocartelle `C-aux` e `java-aux`, contenenti il codice di supporto e lo scheletro delle soluzioni per il quesito di programmazione (quesito 1). Svolgere il compito nel modo seguente:

- Per il quesito 1, alla fine la cartella `C-aux` (o `java-aux`, a seconda del linguaggio usato) dovrà contenere le implementazioni delle soluzioni al quesito 1, ottenute completando i relativi metodi (o le relative funzioni) contenuti nei file forniti dai docenti; si ricorda ancora una volta che la cartella `java-aux` (o `c-aux`) deve trovarsi all'interno della cartella `Esame`.
- Per i quesiti 2 e 3, creare due file `probl2.txt` e `probl3.txt` contenenti, rispettivamente, gli svolgimenti dei problemi proposti nei quesiti 2 e 3; i tre file devono trovarsi nella cartella `Esame`. È possibile integrare i contenuti di tali file con eventuale materiale cartaceo *unicamente per disegni, grafici, tabelle, calcoli*.

Attenzione: i simboli < e > usati nei nomi dei file fanno parte del linguaggio dei metadati e non debbono essere inclusi nei nomi reali.

Avviso importante 1. Per svolgere il quesito di programmazione *si consiglia caldamente l'uso di un editor di testo (ad esempio Geany) e la compilazione a riga di comando*, strumenti più che sufficienti per lo svolgimento del compito. La macchina virtuale mette a disposizione diversi ambienti di sviluppo, quali Eclipse e Javabeans. *Gli studenti che li usano lo fanno a proprio rischio*. In particolare, *se ne sconsiglia l'uso qualora non se ne abbia il pieno controllo e certamente se non si è già in grado di sviluppare servendosi unicamente di un editor e della compilazione a riga di comando*. Qualora lo studente intenda comunque usare un ambiente di sviluppo integrato, si raccomanda di controllare che i file vengano effettivamente salvati nella cartella `java-aux` (o `c-aux`, a seconda del linguaggio usato), in quanto vi è il rischio concreto di perdere il proprio lavoro. In generale, file salvati esternamente alla cartella `Esame` andranno persi al termine della prova e quindi non saranno corretti.

Avviso importante 2. Il codice *deve compilare* correttamente (warning possono andare, ma niente errori di compilazione), altrimenti riceverà 0 punti. *Si raccomanda quindi di scrivere il programma in modo incrementale, ricompilando il tutto di volta in volta*.

Quesito 1: Progetto algoritmi C/Java [soglia minima per superare l'esame: 5/30]

1. In questo problema si fa riferimento a grafi semplici *diretti*. In particolare, ogni nodo ha associata la lista dei vicini uscenti (vicini ai quali il nodo è connesso da archi uscenti) e dei vicini entranti (nodi che hanno archi diretti verso il nodo considerato). Il grafo è descritto nella classe `Graph.java` (Java) e nel modulo `graph.c` (C). Il generico nodo è descritto nella classe `GraphNode` (Java) e in `struct graph_node` (C) .

   Sono inoltre già disponibili le primitive di manipolazione del grafo: creazione di grafo vuoto, lista dei vicini uscenti ed entranti di ciascun nodo (quest'ultima non necessaria per lo svolgimento di questo esercizio), inserimento di un nuovo nodo, inserimento di un nuovo arco, get label/valore (stringa) di un dato nodo, cancellazione arco, cancellazione nodo (e relativi archi incidenti), cancellazione grafo, stampa grafo. Per dettagli sulle segnature 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.

   In particolare, per Java si rimanda alle classi `Graph` e `GraphNode` (quest'ultima classe interna di e contenuta nel file `Graph.java`) e ai commenti contenuti in `Graph.java`. Per C si rimanda agli header file `graph.h`, `graph_services.h` e `linked_list.h` (per le liste).

   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 calcolatore quanto segue, in Java o C, con l'avvertenza che gli esempi dati nel testo che segue fanno riferimento alla figura seguente. che corrisponde al grafo usato nel programma di prova (`Driver.java` o `driver.c`):

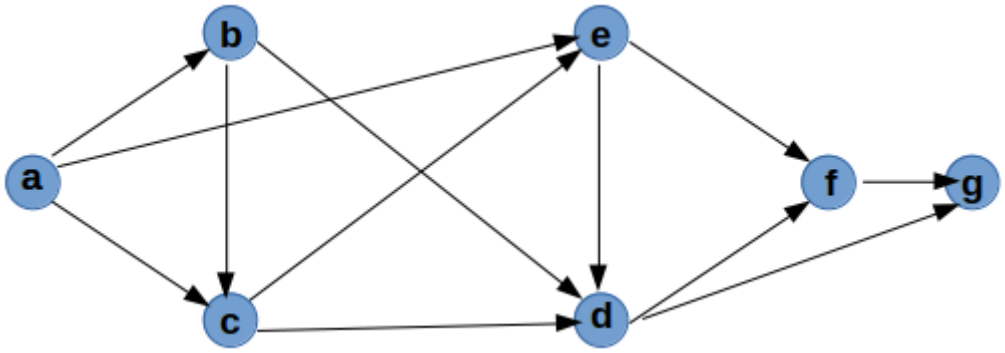

   Figura 1. Esempio di grafo diretto non pesato. Le distanze dal nodo **a** sono [**a**: 0, **b**: 1, **c**: 1, **e**: 1, **d**: 2, **f**: 2, **g**: 3], mentre le distanze dal nodo **e** sono [**a**: -, **b**: -, **c**: -, **d**: 1, **e**: 0, **f**: 1, **g:** 2], dove il simbolo "-" indica che un nodo non è raggiungibile a partire dal nodo considerato. Se `source` è il nodo **a** e $k = 2$ la funzione deve stampare la stringa "a b c d e f" (l'ordine non è importante), mentre se `source` fosse il nodo **e** e $k = 1$ la funzione dovrebbe stampare la stringa "e d f". Infine, se `source` corrispondesse al nodo **g**, la funzione dovrebbe stampare la stringa "g", indipendentemente dal valore di k, in quanto non vi sono nodi raggiungibili da **g** oltre **g** stesso.

1. Implementare la funzione/metodo `static <V> void kdist(Graph<V> g, Node<V> source), int k` della classe `GraphServices` (o `void kdist(graph* g, graph_node* source, int k)` del modulo `graph_services` in C) che, dato un grafo `g`, un (oggetto) nodo `source` e un intero `k`, stampa le etichette dei nodi del grafo (incluso `source`) che si trovano a distanza *al più* k da `source` in termini del *numero minimo di archi da attraversare*. Si considerino con attenzione la Figura 1 e gli esempi in essa contenuti.

   La valutazione delle risposte degli studenti terrà conto dell'efficienza delle soluzioni proposte.

   Nota: Per tenere traccia della distanza dei vari nodi dalla sorgente si consiglia di usare il campo `int timestamp` della classe `GraphNode` (`int timestamp` di `struct graph_node`) .

   Punteggio: [10/30]

Quesito 2: Algoritmi

1. Si consideri un grafo $G = (V, E)$ avente n vertici, *non diretto e completo* (ossia, avente un arco non diretto per ogni possibile coppia (u, v) di vertici). Si caratterizzino le proprietà dell'albero di visita DFS (Depth First Search) per questo grafo a partire dal generico vertice $s \in V$. In particolare, caratterizzare, *motivando* brevemente ogni risposta:
   - L'altezza dell'albero di visita
   - Il numero di foglie dell'albero di visita
   - Il grado (numero di archi incidenti in un nodo) minimo e il grado massimo dell'albero di visita

   Punteggio: [4/30]

   Rispondere alle stesse domande per la visita BFS (Breadth First Search).

   Punteggio: [4/30]

2. Sia T un albero binario con chiavi intere associate ai nodi. Si indichi con `T.root` la radice dell'albero. Dato un nodo `v`, siano `v.key`, `v.left` e `v.right` rispettivamente la chiave associata a `v` e i suoi figli sinistro e destro. Scrivere lo pseudo-codice di un algoritmo che, dato T, restituisca `true` se T le chiavi associate ai nodi di T soddisfano la proprietà di ordinamento di un heap minimale, `false` altrimenti. Si noti che non è richiesto di verificare che T sia un albero binario completo.

   Occorre descrivere l'algoritmo con uno pseudo-codice chiaro. Il punteggio dipenderà dalla correttezza e dall'efficienza dell'algoritmo proposto, nonché dalla chiarezza della sua descrizione.

   Punteggio: [7/30]

Quesito 3:

Si supponga di avere due liste ℓ_1 ed ℓ_2, *ordinate* in modo crescente, contenenti rispettivamente n_1 e n_2 stringhe (l'ordinamento è ovviamente quello lessicografico). Si supponga che una stringa possa apparire *al più una volta* in ciascuna lista.

- Si proponga e *si scriva lo pseudo-codice* del più efficiente algoritmo possibile che, prese in ingresso le due liste, restituisca una lista ℓ *ordinata*, contenente le stringhe che compaiono in ℓ_1 ma non in ℓ_2. Ad esempio, se $\ell_1 = \{a, ac, b, bc, d, eh\}$ e $\ell_2 = \{a, ad, bc, eh\}$, allora $\ell = \{ac, b, d\}$.

  Nota: a parte variabili scalari di appoggio, non si possono usare ulteriori strutture dati oltre alle liste.

Punteggio: [4/30]

- Si calcoli il costo asintotico dell'algoritmo proposto.

  Punteggio: [3/30]

Occorre dare un'argomentazione quantitativa per il costo asintotico. Il punteggio dipenderà molto dalla chiarezza espositiva e dalla solidità delle argomentazioni proposte, nonché dalla semplicità dell'algoritmo proposto.

Appendice: interfacce dei moduli/classi per il quesito 1

Di seguito sono descritti i campi/moduli/funzioni delle classi Java o moduli C che si suppone siano utilizzabili.

Interfacce Java

In Java sono già implementate e disponibili le classi `GraphNode` e `Graph`, che rispettivamente implementano il generico nodo e un grafo non diretto. Ovviamente, in Java si assumono disponibili tutte le classi delle librerie standard, come ad esempio `java.util.LinkedList` ecc. In C sono implementati già i moduli che realizzano un grafo e le primitive per la sua manipolazione. Si vedano anche i sorgenti (che sono commentati) delle rispettive classi o moduli C.

Classe GraphNode

```
public class GraphNode<V> implements Cloneable{
    public static enum Status {UNEXPLORED, EXPLORED, EXPLORING}

    protected V value; // Valore associato al nodo
    protected LinkedList<GraphNode<V>> outEdges; // Lista dei vicini uscenti
    protected LinkedList<GraphNode<V>> inEdges; // Lista dei vicini entranti

    // keep track status
    protected Status state; // Stato del nodo
    protected int timestamp; // Campo intero utilizzabile per vari scopi

    @Override
    public String toString() {
        return "GraphNode [value=" + value + ", state=" + state + "]";
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return (GraphNode<V>) this;
    }
}
```

Metodi messi a disposizione dalla classe Graph. Di seguito si descrivono le interfacce dei metodi utili alla risoluzione degli esercizi e messi a disposizione dalla classe `Graph`.

```java
// Restituiscee una lista di riferimenti ai nodi del grafo
public List<GraphNode<V>> getNodes();

// Restituisce una lista con i riferimenti dei vicini uscenti del nodo n
(outEdges nella classe GraphNode)
public List<GraphNode<V>> getOutNeighbors(GraphNode<V> n);

// Restituisce una lista con i riferimenti dei vicini entranti del nodo n
(inEdges nella classe GraphNode)
public List<GraphNode<V>> getInNeighbors(GraphNode<V> n);
```

Metodi potenzialmente utili della classe LinkedList.

```java
// Appende e alla fine della lista. Restituisce true
boolean add(E e);

// Rimuove e restituisce l'elemento in testa alla lista.
E remove(); // E indica il tipo generico dell'elemento
```

Scheletro della classe GraphServices

Di seguito, lo scheletro della classe `GraphServices` con le segnature dei metodi che essa contiene.

```java
public class GraphServices<V>{

  public static <V> void kdist(Graph<V> g, Graph.GraphNode<V> source, int k) {
        /* DA IMPLEMENTARE */
      System.out.println(" ");
  }

}
```

Interfacce C

graph.h (solo tipi principali)

```c
#include "linked_list.h"

typedef enum { UNEXPLORED, EXPLORED, EXPLORING } STATUS;

typedef struct graph_node {
    void *value;
    linked_list *out_edges;
    linked_list *in_edges;

    STATUS status;
    int timestamp;
} graph_node;
```

```c
typedef struct graph_prop {
    int n_vertices;
    int n_edges;
} graph_prop;

typedef struct graph {
    linked_list* nodes;
    graph_prop* properties;
} graph;
```

Scheletro del modulo C graph_services

Di seguito lo scheletro del metodo `graph_services.c` e le segnature delle funzioni da implementare.

```c
#include "graph.h"

int void kdist(graph* g, graph_node* source, int k) {
        /* DA IMPLEMENTARE */
    printf(" ");
}
```

linked_list.h (solo parte)

```c
typedef struct linked_list_node {
    void *value;
    struct linked_list_node *next;
    struct linked_list_node *pred;
} linked_list_node;

typedef struct linked_list {
    linked_list_node *head;
    linked_list_node *tail;
    int size;
} linked_list;

typedef struct linked_list_iterator linked_list_iterator;

/**********************
      linked_list
**********************/
/**
Crea una nuova lista.
*/
linked_list * linked_list_new();

/**
Aggiunge in testa alla lista ll, un nodo che punta a value.
*/
void linked_list_insert_head(linked_list* ll, void* value);

/**
Aggiunge in coda alla lista ll, un nodo che punta a value.
```

```
*/
void linked_list_insert_tail(linked_list* ll, void* value);

/**
Come linked_list_insert_tail(linked_list* ll, void* value).
*/
void linked_list_add(linked_list * ll, void * value);


/**
Aggiunge alla lista ll un nodo che punta a value, subito dopo predec
*/
void linked_list_insert_after(linked_list * ll, linked_list_node *predec, void
* value);


/**
Rimuove dalla lista ll il nodo in testa e ritorna il valore puntato da tale
nodo.
*/
void *linked_list_remove_head(linked_list* ll);

/**
Rimuove dalla lista ll il nodo in coda e ritorna il valore puntato da tale
nodo.
*/
void* linked_list_remove_tail(linked_list * ll);

/**
Ritorna un puntatore al valore puntato dal nodo in input.
*/
void *linked_list_node_getvalue(linked_list_node* node);

/**
Ritorna la dimensione della lista ll.
*/
int linked_list_size(linked_list *ll);


/**
Ritorna 1 se la linked list contiene value, 0 altrimenti.
*/
int linked_list_contains(linked_list *ll, void *value);


/**
Stampa a video una rappresentazione della lista ll.
*/
void linked_list_print(linked_list *ll);

/**
Distrugge la lista ll e libera la memoria allocata per i suoi nodi. Nota che la
funzione
non libera eventuale memoria riservata per i valori puntati dai nodi della
lista.
```

```
*/
void linked_list_delete(linked_list *ll);


/**********************
  linked_list_iterator
***********************/
/**
Crea un nuovo iteratore posizionato sul primo elemento della lista ll.
*/
linked_list_iterator * linked_list_iterator_new(linked_list *ll);

/**
Ritorna 1 se l'iteratore iter ha un successivo, 0 altrimenti.
*/
int linked_list_iterator_hasnext(linked_list_iterator* iter);


/**
Muove l'iteratore un nodo avanti nella lista e ritorna il valore puntato dal
nodo
appena oltrepassato, o NULL se l'iteratore ha raggiunto la fine della lista.
*/
//void * linked_list_iterator_next(linked_list_iterator * iter);
linked_list_node * linked_list_iterator_next(linked_list_iterator * iter);

/**
Rimuove dalla lista il nodo ritornato dall'ultima occorrenza della funzione
linked_list_iterator_next.
*/
void *linked_list_iterator_remove(linked_list_iterator * iter);

/**
Ritorna il valore puntato dal nodo su cui si trova attualmente l'iteratore
iter.
*/
//void * linked_list_iterator_getvalue(linked_list_iterator *iter);

/**
Distrugge l'iteratore e libera la memoria riservata. Nota che questa operazione
non ha nessun effetto sulla lista puntata dall'iteratore.
*/
void linked_list_iterator_delete(linked_list_iterator* iter);
```