ESERCIZI DA SVOLGERE

Laboratorio di Algoritmi e Strutture Dati

ISTRUZIONI GENERALI

- Il progetto di laboratorio va inizializzato "clonando" il repository su git

- l'uso corretto di git (commit adeguate da parte di tutti i componenti del gruppo) è parte della valutazione

- su Git dovrà essere caricato solamente il codice:
  nessun file dati dovrà essere oggetto di commit!!!!

- Relazione (circa una pagina) sui risultati dell'Esercizio 2

ISTRUZIONI GENERALI

- Uso di librerie esterne e/o native del linguaggio scelto:
    - in Java, possibile l'uso di ArrayList
    - È lecito (ma non obbligatorio) avvalersi di strutture dati più complesse quando la loro realizzazione non è richiesta da uno degli esercizi proposti
        - Es. HashTable OK, libreria per i grafi NO (vedi esercizio 4)

ISTRUZIONI GENERALI

- Tutti gli esercizi richiedono almeno di sviluppare una struttura dati e/o un algoritmo

- Si deve assumere di stare sviluppando una libreria generica intesa come fondamento di futuri programmi

    - Non è lecito fare assunzioni semplificative sugli usi

    - implementazione della libreria generica non influenzata dagli usi eventualmente richiesti negli esercizi

ISTRUZIONI GENERALI

- Esempio: esercizio che richiede
    - l'implementazione della struttura dati "grafo"
    - l'implementazione di un algoritmo per il calcolo delle componenti connesse di un grafo
        - l'implementazione della struttura dati non deve contenere elementi (es. variabili, procedure) utili per il calcolo delle componenti connesse, ma non essenziali alla struttura dati
- Esempio: esercizio che richiede di operare su grafi con nodi di tipo stringa
    - l'implementazione della struttura dati grafo deve restare generica
        - non deve assumere che i nodi contengano solo dati di tipo stringa

ISTRUZIONI GENERALI

- In alcuni esercizi si ribadisce la necessità di implementare una versione generale della libreria
    - **Non implica che dove non specificato esplicitamente sia lecita una implementazione meno generale**
- Tutti gli esercizi chiedono di implementare un programma che sfrutta la libreria realizzata
    - Questa parte degli esercizi (e solo questa) può fare leva sulle caratteristiche particolari del problema
        - Esempio si può assumere che i dati siano di un particolare tipo

UNIT TEST

- Implementare gli unit-test di **tutti** gli algoritmi implementati

DATASET

- Per alcuni esercizi sono forniti files con dataset per effettuare opportune prove
  - /usr/NFS/Linux/labalgoritmi/datasets/
  - in laboratorio von Neumann, selezionare il disco Y
- **NOTA**: questi files non devono essere oggetto di commit su Git!

ESERCIZIO 1

- Si consideri il tipo di dato astratto **Lista**, definito nei termini delle seguenti operazioni:
  - verifica se la lista è vuota in O(1)
  - inserimento in coda alla lista in O(1)
  - inserimento di un elemento nella posizione i-esima della lista in O(n)
  - cancellazione dell'elemento in coda alla lista in O(1)
  - cancellazione dell'elemento in posizione i-esima nella lista in O(n)
  - recupero dell'elemento in posizione i-esima nella lista (senza cancellare l'elemento dalla lista) in O(n)
  - recupero del numero di elementi della lista in O(1)
  - creazione di un iteratore per la lista in O(1)

ESERCIZIO 1

- La lista può contenere oggetti di tipo qualunque e non noto a priori
- Iteratore:
    - tipo di dato astratto che permette di iterare su un container di qualche tipo.
    - deve mettere a disposizione le seguenti operazioni:
        - Verifica se l'iteratore è ancora valido in O(1) (un iteratore è inizializzato in modo da fare riferimento alla testa della lista e diventa invalido quando viene spostato oltre la fine della lista)
        - Recupera l'elemento corrente in O(1)
        - Sposta l'iteratore all'elemento successivo in O(1)

ESERCIZIO 1

- Si realizzino in C due implementazioni alternative per il tipo di dato astratto Lista (e, conseguentemente per l'iteratore su di essa) basate su:
    - array dinamici (ridimensionabili)
    - record collegati.
- Entrambe le implementazioni devono offrire:
    - funzione per creare una lista vuota;
    - funzione per distruggere la lista (con conseguente deallocazione della memoria associata)
    - una funzione per distruggere un iteratore (con conseguente deallocazione della memoria associata)
    - tutte e sole le operazioni specificate, realizzate tramite funzioni aventi la stessa signature in entrambe le librerie.

ESERCIZIO 1 - USO

- Implementare un algoritmo **merge**
- Input:
    - criterio di ordinamento
        - due liste ordinate secondo tale criterio di ordinamento
- Restituisce in output una nuova lista, corrispondente alla fusione delle due liste di input e ordinata secondo lo stesso criterio
- L'algoritmo implementato deve poter essere eseguito ***senza modifiche*** su ciascuna delle due implementazioni per il tipo di dato astratto Lista prodotte

ESERCIZIO 1 - UNIT TESTING

- Implementare gli unit-test per gli algoritmi che implementano le funzioni del tipo di dato astratto Lista

- Implementare gli unit-test per la funzione che implementa merge

ESERCIZIO 2

- Si consideri il problema di determinare la distanza di edit tra due stringhe (Edit distance)
    - date due stringhe s1 e s2, non necessariamente della stessa lunghezza, determinare il minimo numero di operazioni necessarie per trasformare la stringa s2 in s1
- Operazioni disponibili:
        - cancellazione di un carattere
        - inserimento di un carattere
        - rimpiazzamento di un carattere

ESERCIZIO 2

- Esempi:
  - "casa" e "cassa" edit distance =1 (1 cancellazione)
  - "casa" e "cara" edit distance =1 (1 rimpiazzamento)
  - “vinaio” e “vino” edit distance=2 (2 inserimenti)
  - "tassa" e "passato" edit distance =3 (2 cancellazioni + 1 rimpiazzamento)
  - "pioppo" e "pioppo" edit distance =0

ESERCIZIO 2

- Si implementi una versione ricorsiva della funzione **edit_distance**

- Sia $|s|$ la lunghezza di una stringa

- Sia $\mathrm{rest}(s)$ la sottostringa di s ottenuta ignorando il primo carattere di s

  - se $|s1| = 0$, allora $\mathrm{edit\_distance}(s1, s2) = |s2|$

  - se $|s2| = 0$, allora $\mathrm{edit\_distance}(s1, s2) = |s1|$

ESERCIZIO 2

- Altrimenti siano:
  - $d_{\text{no-op}} = \begin{cases} \text{edit\_distance}(\text{rest}(s1), \text{rest}(s2)) & \text{se} \quad s1[0] = s2[0] \\ \infty \quad \text{altrimenti} \end{cases}$
  - $d_{\text{canc}} = 1 + \text{edit\_distance}(s1, \text{rest}(s2))$
  - $d_{\text{ins}} = 1 + \text{edit\_distance}(\text{rest}(s1), s2)$
  - $d_{\text{replace}} = 1 + \text{edit\_distance}(\text{rest}(s1), \text{rest}(s2))$
- Allora:

$$\text{edit\_distance}(s1, s2) = \min\{d_{\text{no-op}}, d_{\text{canc}}, d_{\text{ins}}, d_{\text{replace}}\}$$

ESERCIZIO 2

- Si implementi una versione **edit_distance_dyn** della funzione, adottando una strategia di programmazione dinamica

- Questa versione deve essere a sua volta ricorsiva

- **Nota**: Le definizioni alle slides precedenti non corrispondono al modo usuale di definire la distanza di edit. Sono del tutto sufficienti però per risolvere l'esercizio e sono quelle su cui dovrà essere basato il codice prodotto

ESERCIZIO 2 - USO DELLE FUNZIONI

- File dictionary.txt
    - elenco delle parole italiane (molte)
        - Parole scritte di seguito, ciascuna su una riga
- File correctme.txt
    - citazione di John Lennon
    - presenti alcuni errori di battitura

ESERCIZIO 2 - USO DELLE FUNZIONI

- Si implementi un'applicazione che usa la funzione edit_distance_dyn per determinare, per ogni parola w in correctme.txt, la lista di parole in dictionary.txt con edit distance minima da w

- Si sperimenti il funzionamento dell'applicazione e si riporti in una breve relazione (circa una pagina) i risultati degli esperimenti

ESERCIZIO 2- UNIT TESTING

- Implementare gli unit-test degli algoritmi

ESERCIZIO 3

- Si implementi la struttura dati **Union-Find Set**
- La struttura dati deve permettere di inserire oggetti di tipo generico e non deve prevedere alcuna cardinalità massima per l'insieme iniziale di elementi

ESERCIZIO 3- UNIT TESTING

- Implementare gli unit-test degli algoritmi

ESERCIZIO 4

- Si implementi una libreria che realizza la struttura dati **Grafo** in modo che sia ottimale per dati sparsi

- La struttura deve consentire di rappresentare sia grafi diretti che grafi non diretti

  - *suggerimento*: un grafo non diretto può essere rappresentato usando un'implementazione per grafi diretti modificata per garantire che, per ogni arco (a,b), etichettato w, presente nel grafo, sia presente nel grafo anche l'arco (b,a), etichettato w

  - il grafo dovrà mantenere l'informazione che specifica se esso è un grafo diretto o non diretto

ESERCIZIO 4

- L'implementazione deve essere generica
  - sia per quanto riguarda il tipo dei nodi
  - sia per quanto riguarda le etichette degli archi

ESERCIZIO 4

- Offrire (almeno) le seguenti operazioni (n=numero di nodi o di archi, a seconda del contesto):
  - Creazione di un grafo vuoto in O(1)
  - Aggiunta di un nodo in O(1)
  - Aggiunta di un arco in O(1)
  - Verifica se il grafo è diretto in O(1)
  - Verifica se il grafo contiene un dato nodo in O(1)
  - Verifica se il grafo contiene un dato arco in O(1) ✳
  - Cancellazione di un nodo in O(n)
  - Cancellazione di un arco in O(1) ✳
  - Determinazione del numero di nodi in O(1)
  - Determinazione del numero di archi in O(n)
  - Recupero dei nodi del grafo in O(n)
  - Recupero degli archi del grafo in O(n)
  - Recupero nodi adiacenti di un dato nodo in O(1) ✳
  - Recupero etichetta associata a una coppia di nodi in O(1) ✳

✳ **: quando il grafo è veramente sparso, assumendo che l'operazione venga effettuata su un nodo la cui lista di adiacenza ha una lunghezza in O(1)**

ESERCIZIO 4- UNIT TESTING

- Implementare gli unit-test degli algoritmi

ESERCIZIO 4- USO DELLA LIBRERIA

- Si implementi l'algoritmo di **Kruskal** per la determinazione della minima foresta ricoprente di un grafo
- L'implementazione dell'algoritmo di Kruskal dovrà utilizzare la struttura dati Union-Find Set implementata nell'Esercizio 3
    - NOTA BENE:
        - Nel caso in cui il grafo sia costituito da una sola componente connessa, l'algoritmo restituirà un albero
        - Nel caso in cui vi siano più componenti connesse, l'algoritmo restituirà una foresta costituita dai minimi alberi ricoprenti di ciascuna componente connessa.

ESERCIZIO 4- USO DELLA LIBRERIA E DELL'ALGORITMO DI KRUSKAL

- La struttura dati Grafo e l'algoritmo di Kruskal dovranno essere utilizzati con i dati contenuti nel file italian_dist_graph.csv
- Contiene le distanze in metri tra varie località italiane e una frazione delle località a loro più vicine
- Formato: CSV standard
    - campi separati da virgole
    - record separati dal carattere di fine riga (\n).
- Ogni record contiene:
    - località 1: (tipo stringa) nome della località "sorgente". Può contenere spazi, non virgole
    - località 2: (tipo stringa) nome della località "destinazione". Può contenere spazi, non virgole
    - distanza: (tipo float) distanza in metri tra le due località

ESERCIZIO 4- USO DELLA LIBRERIA E DELL'ALGORITMO DI KRUSKAL

- potete interpretare le informazioni presenti nelle righe del file come archi non diretti
    - *suggerimento:* inserire nel grafo sia l'arco di andata che quello di ritorno per ogni riga
- il file è stato creato a partire da un dataset poco accurato
    - i dati contengono inesattezze e imprecisioni
- Un'implementazione corretta dell'algoritmo di Kruskal, eseguita sui dati contenuti nel file italian_dist_graph.csv, dovrebbe determinare una minima foresta ricoprente con:
    - 18.640 nodi
    - 18.637 archi (non orientati)
    - peso complessivo di circa 89.939,913 Km

ESERCIZIO 4- UNIT TESTING

- Implementare gli unit-test degli algoritmi