MANUALE TECNICO

CAPITOLI

1) PRESENTAZIONE APPLICAZIONE

2) SCELTE TECNICHE E ALGORITMICHE CON ANALISI COMPLESSITÀ

3) PRESTAZIONI DELL'APLICAZIONE CON UN MILIONE E CENTOMILA DATI INSERITI

1) CLASSI COIVOLTE: PRESENTAZIONE

➔ **EmotionalState:**

Il fine di questa classe è quello di definire una struttura dati che renda gli stati immutabili e riconoscibili, senza doverli rappresentare come stringhe.
Questa classe è, quindi, di tipo **ENUM** e rappresenta tutti i possibili stati emozionali dell'aplicazione: "ARRABBIATO, FELICE, TRISTE, NEUTRO, SORPRESO, NEUTRO".

Essa ha due metodi statici:

- *GetEmotionalState (String emotionalStateValue):* ritorna un elemento della classe presa in input una stringa con un valore emozionale tra "A, F, T, S, N". Viene usato in fase di parsing delle stringhe di eventi che l' utente importa, al fine di creare un evento con un campo "statoEmozionale" con uno dei valori di questa classe.

- *ToArray():* ritorna un array contenente tutti gli stati emozionali. Viene usata dalla classe manager di dati per popolare le strutture dati contenenti l' ammontare di ogni operazione.

➔ **Event**:

Questa classe è stata scelta come struttura data apposita per modellare un evento.
Essa ha vari parametri tra cui

- *statoRegistrazione:* booleano con valore true se l' utente era registrato durante l' evento, false altrimenti

- *statoLogin:* booleano con valore true se l' utente era acceduto al sistema durante l' evento, false altrimenti

- *date:* Hashmap che rappresenta la data di un evento con chiave String che può assumere valori "ANNO, MESE, GIORNO" e valore in forma Integer dell' anno del mese o del giorno. Utilizzata per avere ***O(1)*** durante l' accesso ai dati

- *poi:* Stringa che può assumere valori "POI1, POI2, POI3, UNDEFINED" in base al point of interest.

- *userId:* intero che rappresenta l' id dell'utente associato all' evento

- *EmotionalState*: Utilizza la classe enum sopra citata per rappresentare lo stato emozionale dell'utente associato all'evento.

➔ **StringParser:**

Questa classe è artefice delle operazioni di parsing delle Stringhe che vengono lette dai file di testo degli eventi. Essa infatti si occupa di tagliare e rendere accessibili alle altre classi dell'applicazione le opeazioni di import, e definizione degli eventi.

Essa ha i seguenti metodi tutti statici:

- Un metodo privato *cutCommands()* che taglia tutte le parti inutili dei comandi come gli spazi o i trattini.

- *parseStringToEvent (String eventValue):* prende in input una stringa contenente il valore dell' evento e ne ritorna un oggetto della classe Event sopra citata.

- *parseCoordinates(String coordinatesValue):* prende in input una stringa contenente il valore delle coordinate di un evento e ritorna un array di float[] contente al punto 0 dell' array le x e al punto 1 le y.

- *parseStringToUpperLowerDate(String uppLowValue):* prende in input una stirnga rappresentante le date di lower bound e upper bound richieste per creare una mappa emozionale e ne ritorna due eventi con le date corrispondenti.

➔ **TxtReader:**

Questa classe viene utilizzata in combinazione con la classe string parser e le classi `java.io.BufferedReader, java.io.File, java.io.FileReader,` per leggere i dati del file comandi txt , tagliare le parti del file utili alla creazione di eventi ed al loro salvataggio, aggiungere al data manager gli eventi. Essa ha i seguenti metodi: (spiegati meglio nel capitolo successivo):

- *readCommands(String file)*

- *execCommands()*

➔ **DataManager:**

Questa classe è stata creata al fine di fare lo storage dei dati che l' utente richiede di importare nel modo più efficiente possibile. Le scelte tecniche saranno meglio discusse nella seconda parte.

Essa ha un parametro: dataCollector, ovvero un HashMap con **chiave** l'**anno** di un insieme di eventi avvenuti nello stesso anno e **valore** , per ogni anno, un oggetto della classe `java.util.TreeSet` utilizzato per salvare gli eventi: infatti i TreeSet prendono come tipo parametro la nostra classe Event.

Essa ha i seguenti metodi:

- *addEvent(Event):* aggiunge un evento alla struttura dati degli eventi nell' anno corrispondente. Se l'anno non è presente verrà gestita l'eccezione (NullPointerException) che solleva l'HashMap contente i TreeSet e l' anno verrà aggiunto alla struttra dati.

- *createMap(LowerBoundEvent, UpperBoundEvent): presi in input la data più piccola e quella più grande, crea una mappa emozionale e la stampa sullo standard output.*

➔ ***RandomEventGenerator:***

Questa classe è stata usata per creare centomila o un milione di eventi in maniera casuale nel pc di test dell'applicazione con una data compresa tra 2000 e 2030 e un POI con 1/3 di possibilità.

2) SCELTE TECNICHE E ALGORITMICHE

➔ Gestione dei dati:

Per gestire i dati, come già descritto, nella classe DataManager, la tecnica scelta è stata di utilizzare un HashMap con chiave corrispondente all' anno di un insieme di eventi e con valore un TreeSet contenente gli eventi corrispondenti a quell'anno.

Motivazione hashmap:
la scelta dell' HashMap è stata presa tenendo in considerazione le operazioni da svolgere in caso sia richiesto di aggiungere un evento o creare una mappa emozionale. Infatti durante queste operazioni è necessario andare a prendere gli eventi che partono da una data e terminano con un altra . Se avessimo utilizzato una Lista, un Vettore, o una qualsiasi altra struttura dati per rappresentare gli eventi, per trovare l'anno di partenza, o di fine sarebbero servite delle operazioni con costo computazionale **O(K)** dove **k** è il numero di eventi avvenuti prima della data di lower bound. Utilizzando un HashMap invece, per trovare l' anno di appartenenza della data di partenza serve un operazione con costo **O(1)** mentre per trovare il giorno solo il costo **O(log(z))** dove **z** è il numero di eventi precedenti, come giorno, a quello di inizio. Il costo complessivo è di **O(log(z))**.

Motivazione *java.util.TreeSet:*
Come specificato dalla documentazione ufficiale da java 7 in poi, un TreeSet è un implementazione navigabile di *java.util.TreeMap,* ovvero una classe che modella alberi **red black**. Questi alberi, come visto a lezione di algoritmi e strutture dati, hanno costo **O(log(n))** nelle operazioni di ricerca, aggiunta e rimozione, ovvero le operazioni più richieste dal sistema.

In questo modo:

- Se l' utente richiede di **aggiungere** un evento il costo dell 'operazione sarà **O(1)** per trovare l' anno dell HashMap corrispondente data e **O(log(z))** per l' aggiunta dell'evento, per un costo complessivo di **O(log(z)),** dove **z** NON è il numero di tutti gli eventi ma solo il numero degli eventi in un determinato anno (per quanto detto nella motivazione del HashMap).

- Se l' utente richiede di **creare una mappa emozionale** il costo dell' operazione dipenderà dall' anno di inzio creazione mappa e termine creazione mappa:

  - **Anno inzio creazione = Anno termine creazione:**

    In questo caso il costo della creazione della mappa sara corrispondente a:

    → ricerca dell' anno di inzio (o fine perché e lo stesso) per evitare di creare mappe di anni che non esistono nel sistema. **O(1)** per motivazione HashMap.

→ Iterazione del TreeSet mediante l' interfaccia `java.util.Iterator` e aggiunta degli eventi alla struttura dati divisi per POI e stati emozionali *Costo* ***O(n)*** dove **n** è compreso tra il primo evento dell' anno e l' evento di confine.

→ Calcolo delle percentuali e stampa a schermo. **O(3 * 5) = O(1)**. In quanto i valori di ogni stato emozionale sono in un hasmap.

La complessità totale è quindi **O(n)**.

- **Anno inizio creazione < Anno termine creazione:**

  In questo caso il costo della creazione della mappa sara corrispondente a:

  → ricerca dell' anno di inzio e fine per evitare di creare mappe di anni che non esistono nel sistema. **O(1)** per motivazione HashMap.

  → **Per ogni anno** coinvolto: iterazione del TreeSet mediante l' interfaccia `java.util.Iterator` e aggiunta degli eventi alla struttura dati divisi per POI e stati emozionali. *Costo* ***O(n + k),*** dove **n** è compreso tra il primo evento dell' anno e l' evento di confine, e **k** è il numero di anni.

  → Calcolo delle percentuali e stampa a schermo. **O(3 * 5) = O(1)**. In quanto i valori di ogni stato emozionale sono in un hasmap.

  La complessità totale è quindi **O(n + k)**

Precisazioni:

- Usando una lista di Eventi si sarebbe ottenuto **O(n + k)** in fase di creazione della mappa (dove **n** è il numero di eventi in un anno, e **k** è il numero di anni coinvolti), ma si otterrebbe **O(n)** anche in fase di aggiunta (l' operazione più richiesta dal sistema) e **O(n)** per eliminazione di eventi. Una lista sarebbe quindi meno efficiente.

- Piuttosto che iterare su tutti i dati di un anno, per i**terare solo su una parte** di essi sarebbe bastato creare un **SubSet** del TreeSet di partenza mediante il metodo ***subSet(lower, upper)*** della classe TreeSet, con costo uguale a **O(m*log(n)),** in quanto è necessario attraversare il TreeSet m volte, dove **m** è il numero di dati compresi tra la data di partenza e di fine e **n** il numero totale di dati. Quindi a livello asintontico la creazione di una mappa sarebbe costata **O(m*log(n) + m).** Tuttavia **m*log(n) + m != O(n)** e quindi riuslta più efficiente iterare su tutti i dati.

➔ **Gestione dell' aggiunta di eventi in TreeSet:**

La classe *java.util.TreeSet<T>* prende come parametro solo classi che implementano l'interfaccia *Comparable.* Siccome il parametro dei TreeSet del sistema è la classe Event, da noi creata, essa implementa *Comparable,* e il metodo *compareTo()* è stato definito da noi.
La strategia di definizione del metodo *CompareTo(Event otherEvent)* è stata la seguente:

- Confronto dell' anno
  Se l'evento *this* ha l'**anno** maggiore di *otherEvent* allora ritorna 1.
  Se l'evento *this* ha l'**anno** minore di *otherEvent* allora ritorna -1.
  Altrimenti ritorna il confronto del **mese.**

- Confronto del mese
  Se l'evento *this* ha il **mese** maggiore di *otherEvent* allora ritorna 1.
  Se l'evento *this* ha il **mese** minore di *otherEvent* allora ritorna -1.
  Altrimenti ritorna il confronto del **giorno.**

- Confronto del giorno
  Se l'evento *this* ha il **giorno** maggiore di *otherEvent* allora ritorna 1.
  Se l'evento *this* ha il **giorno** minore O UGUALE di *otherEvent* allora ritorna -1.

Con questa tecnica, quindi, se due eventi sono avvenuti lo stesso giorno, verrà considerato più recente, e quindi maggiore, un evento inserito successivamente nel file di import degli eventi. Inoltre è opportuno considerare che se gli eventi avvenuti lo stesso giorno fossero considerati uguali allora il TreeSet ne importerebbe solo uno e scarterebbe tutti quelli uguali.

Di seguito il codice:

```
@Override
public int compareTo(Event otherEvent) {
    //Se gli anni sono uguali ritorno compare del mese. Se i mesi sono uguali ritorno compare del giorno.
    if (compareYear(otherEvent) != 0)
        return compareYear(otherEvent);
    else
        if (compareMonth(otherEvent) != 0)
            return compareMonth(otherEvent);
        else
            return compareDay(otherEvent);
}

//METODI PER COMPARARE ANNI MESI E GIORNI
private int compareYear(Event otherEvent){
    if(this.getYear() > otherEvent.getYear())
        return 1;
    else if(this.getYear() < otherEvent.getYear())
        return -1;
    else

 return 0;
}

private int compareMonth(Event otherEvent){
    if(this.getMonth() > otherEvent.getMonth())
        return 1;
    else if(this.getMonth() < otherEvent.getMonth())
        return -1;
    else
        return 0;
}

//Questo metodo ritorna 1 anche se l'evento è uguale altrimenti il treeset lo scarterebbe
private int compareDay(Event otherEvent){
    if(this.getDay() >= otherEvent.getDay())
        return 1;
    else
        return -1;
}
```

➔ Gestione delle stringhe nel file comandi.txt:

Come sopra descritto per gestire i file che gli utenti andranno ad inserire sono state utilizzate due classi:

- **TxtReader:**
  Questa classe ha i seguenti metodi:

  Un metodo chiamato *readCommands(String path)* che prende come input il path del file comandi.txt e sfrutta la classe *java.io.BufferedReader* per leggere i comandi riga dopo riga e salvarli in un ArrayList come stringhe.

  Un metodo privato *execCommands()* che esegue tutti i comandi importati e tagliati nell'ArrayList, e stampa a schermo un errore con *System.err.println* in caso un comando non sia riconosciuto dal sistema. Il costo di questo metodo è **O(n),** dove **n** è il numero di comandi da eseguire, sommato alla complessità dell'operazione di aggiunta degli eventi in caso di import o la complessità di creazione della mappa (descritte nel capitolo gestione dati), per ogni richiesta.

- **StringParser**:

  I metodi sono stati descritti sopra.
  In particolare il metodo *parseStringToUpperLowerDate(String uppLowValue)*, ritorna un HashMap di eventi con chiave "LOWERBOUND" per l' evento di partenza e "UPPERBOUND" per l' evento di termine e valore l' evento con la data di upper bound e quello con data di lower bound.

➔ Gestione delle date degli eventi:

Anche per la gestione delle date, che come sopra descritto sono un campo della classe Event, è stato scelto di utilizzare un HashMap. In questo modo si ha **O(1)** per la richiesta di Anno, Mese e Giorno di un evento. Questa scelta è stata presa in quanto la richiesta di questi parametri è molto frequente nel sistema e si è voluto renderla il più efficiente possibile.

➔ **Determinazione del Point Of Interest (POI):**

La determinazione del poi viene fatta dalla classe Event al momento della chiamata del proprio costruttore. In questo modo infatti il costruttore di Event chiamerà il metodo setPoi *(String coordinates),* che presa in input la Stringa con le relative coordinata va a creare un Array di float con il valore delle coordinate mediante la classe StringParser e va a determinare il POI con la seguente logica:

Si creano dei range quadrati immaginari intorno ad ogni POI, ognuno con un lato di 10.0 di cui 5.0 a sinitra e 5.0 a destra della x e della y di ogni POI.

Si tenta di far entrare il punto in uno di questi range: se il punto entra in uno dei range viene considerato di quel POI, altrimenti viene considerato “UNDEFINED” e verrà scartato durante la valutazione delle mappe emozionali.

Di seguito il codice:

```
//Metodo usato per determinare il point of interest
public void setPoi(String coordinatesValue) {

    boolean hasPoi = false;
    float[] coordinates = StringParser.parseCoordinates(coordinatesValue);

    //Range quadrato POI1
    if ((coordinates[0] >= 45.459 && coordinates[0] <= 45.469) &&  (coordinates[1] >= 9.185 && coordinates[1] <= 9.195 )) {
        this.poi = "POI1";
        hasPoi = true;
    }

    //Range quadrato POI2
    if ((coordinates[0] >= 45.468 && coordinates[0] <= 45.478) &&  (coordinates[1] >= 9.168 && coordinates[1] <= 9.178)) {
        this.poi = "POI2";
        hasPoi = true;
    }

    //Range quadrato POI3
    if((coordinates[0] >= 45.453 && coordinates[0] <= 45.463) && (coordinates[1] >= 9.176 && coordinates[1] <= 9.185)) {
        this.poi = "POI3";
        hasPoi = true;
    }

    //Se non ha POI sarà considerato UNDEFINED sarà successivamente scartato
    if(!hasPoi)
        this.poi = "UNDEFINED";
}
```

➔ **Calcolo delle percentuali:**

Per calcolare le percentuali si utilizza una variabile della classe Wapper ***Double*** per ogni stato emozionale. Il suo funzionamento è:

1) Si crea un ciclo for per ogni POI

2) Si pone il valore delle variabili Double a: (numero di eventi totali di un POI) **diviso** (ammontare dello stato emozionale in un determinato POI).

3) Si controlla che il valore trovato non corrisponda a *Double.Nan* (divisione di 0/0):

   - Se è NaN allora si modifica il valore a 0.

   - Se non è NaN si divide 100 per il valore precendentemente trovato e si otterrà la percentuale desiderata.

Di seguito il codice:

```
»   /*CREO PERCENTUALI: trovo il 100% degli eventi e lo divido per il valore dello stato emozionale:
»   //Se divido 100 per il valore trovato avrò la percentuale*/

»   Double arrabbiato, felice, sorpreso, triste, neutro;
»   for(int i = 1; i <= 3; i++){

»   »   //ARRABBIATO
»   »   arrabbiato = (double)totalEvents[i-1]/(double)emotionalData[i-1].get(EmotionalStates.ARRABBIATO);
»   »   //FELICE
»   »   felice = (double)totalEvents[i-1]/(double)emotionalData[i-1].get(EmotionalStates.FELICE);
»   »   //SORPRESO
»   »   sorpreso = (double)totalEvents[i-1]/(double)emotionalData[i-1].get(EmotionalStates.SORPRESO);
»   »   //TRSITE
»   »   triste = (double)totalEvents[i-1]/(double)emotionalData[i-1].get(EmotionalStates.TRISTE);
»   »   //NEUTRO
»   »   neutro = (double)totalEvents[i-1]/(double)emotionalData[i-1].get(EmotionalStates.NEUTRO);

»   »   String result = "POI"+ i + " ---> ";
»   »   result += arrabbiato.equals(Double.NaN) ?   0 + " A " : (100/arrabbiato) + " A " ;
»   »   result += felice.equals(Double.NaN)     ?   0 + " F " : (100/felice) + " F " ;
»   »   result += sorpreso.equals(Double.NaN)   ?   0 + " S " : (100/sorpreso) + " S ";
»   »   result += triste.equals(Double.NaN)     ?   0 + " T " : (100/triste) + " T ";
»   »   result += neutro.equals(Double.NaN)     ?   0 + " N " : (100/neutro) + " N " ;

»   »   System.out.println(result);

»   }

»   long endTime = System.currentTimeMillis();

»   System.err.println("Tempo totale (in millisecondi) di creazione mappa: " + (startTime - endTime)/1000);
»   System.err.println("Tempo totale (in secondi) di creazione mappa: " + (startTime - endTime)/1000);


»   System.out.println("tot " + totalEvents[0] + " arr " + emotionalData[0].get(EmotionalStates.ARRABBIATO) );

}
```

3) PRESTAZIONI DELL' APPLICAZIONE

L' applicazione è stata testata nella funzione create_map().

È stata usata la classe RandomEventGenerator per creare centomila e un milione di dati:

- Centomila dati: mappa contenente 95294 valori (somma dei totalEvents): 68 millisecondi di runtime per creazione mappa.

```
comando tagliato: create_map
valore comando tagliato: 01012000-25112028

total events di POI1: 28088
total events di POI2: 33322
total events di POI3: 33884

POI1 ---> 20% A 20% F 20% S 20% T 20% N
POI2 ---> 20% A 20% F 19% S 20% T 20% N
POI3 ---> 20% A 20% F 20% S 20% T 20% N

start time: 1580555870109
end time: 1580555870177
Tempo totale (in millisecondi) di creazione mappa: 68
```

- Un milione di dati: mappa contenente 953699 eventi (somma dei totalEvents): 372 millisecondi di runtime per creazione mappa.

```
comando tagliato: create_map
valore comando tagliato: 01012000-25112028

total events di POI1: 280586
total events di POI2: 333514
total events di POI3: 339599

POI1 ---> 20% A 20% F 20% S 20% T 20% N
POI2 ---> 20% A 20% F 20% S 20% T 20% N
POI3 ---> 20% A 20% F 20% S 20% T 20% N

start time: 1580556405750
end time: 1580556406122
Tempo totale (in millisecondi) di creazione mappa: 372
```