Insiemi implementati con alberi Red Black

Traccia 1 – Quesito 1

0124002062
Caruso Denny

Indice

Descrizione problema

La traccia del quesito numero uno chiede di progettare e implementare una struttura dati basata su alberi Red Black in cui ogni albero rappresenta un insieme generico. Inoltre, è richiesto che la struttura dati a partire da questi insiemi, permetta di eseguire le operazioni canoniche di unione, intersezione e differenza tra due insiemi.

Un insieme in Matematica è un raggruppamento di elementi di qualsiasi tipo: di tipo numerico, logico o concettuale, che può essere individuato mediante una caratteristica comune agli elementi che gli appartengono oppure per semplice elencazione degli elementi dell'insieme.

Prima di procedere con l'analisi approfondita del problema, è necessario ricordare che da un punto di vista matematico un insieme è un raggruppamento di elementi per il quale è possibile decidere senza dubbio se un elemento vi appartiene o meno. Inoltre i suoi elementi devono essere tutti distinti tra loro.
Di conseguenza negli insiemi risultanti che si ottengono dalle operazioni di unione, intersezione e differenza, non vi saranno elementi duplicati. Per quanto riguarda gli insiemi di partenza (forniti in input), si assume che questi potrebbero contenere dei duplicati. La presenza di duplicati all'interno degli insiemi forniti in input non influisce sui risultati delle operazioni; in quanto è presente una particolare gestione degli elementi duplicati che, come vedremo nella sezione "Descrizione Algoritmo", permette di eliminarli tutti.

Nella descrizione delle seguenti operazioni, si assume che non vi siano duplicati all'interno dell'insieme risultante. Inoltre, si considera l'insieme E come l'insieme universo.
Un insieme universo di un insieme A, detto anche insieme ambiente, è un qualsiasi insieme che possa contenere A come sottoinsieme. Il concetto di insieme universo permette di individuare un ambiente all'interno del quale possono essere definiti altri insiemi, intesi come suoi sottoinsiemi.

Per semplicità non sono riportate tutte le proprietà annesse e connesse a queste tre operazioni insiemistiche. Infine, si precisa che la traccia ben chiarisce la natura degli elementi presenti nei singoli insiemi. Gli insiemi sui quali si andrà ad operare sono insiemi numerici e in particolare, sono costituiti soltanto da numeri interi.

Le operazioni insiemistiche canoniche richieste dal problema sono definite come di seguito:

Unione
L'unione di due insiemi è un'operazione che restituisce l'insieme contenente tutti gli elementi del primo insieme e del secondo insieme. In termini rigorosi l'unione di due insiemi,
$A \subseteq E,\ B \subseteq E$ è l'insieme definito da:

$$A \cup B = \{x \in E: x \in A \vee x \in B\}$$

Intersezione
L'intersezione di due insiemi è un'operazione che permette di individuare l'insieme degli elementi che appartengono ad entrambi gli insiemi dati. In termini rigorosi l'intersezione di due insiemi, $A \subseteq E,\ B \subseteq E$ è l'insieme definito da:

$$A \cap B = \{x \in E: x \in A \wedge x \in B\}$$

Differenza
La differenza di due insiemi è un'operazione che restituisce l'insieme contenente tutti gli elementi del primo insieme, dal quale eliminiamo gli elementi del secondo insieme. In termini rigorosi la differenza di due insiemi, $A \subseteq E,\ B \subseteq E$ è l'insieme definito da:

$$A - B = \{x \in E: x \in A \wedge x \notin B\}$$

Come si può notare, la differenza insiemistica non è commutativa e dunque l'ordine degli insiemi nella differenza è fondamentale.

Ora che sono state definite le operazioni insiemistiche richieste dalla traccia, si può passare all'analisi della rappresentazione che la traccia chiede sia data a tali insiemi.

Descrizione strutture dati

È richiesto che gli insiemi siano rappresentati mediante Alberi Red Black.
Prima di iniziare, si precisa che da questo momento si utilizzerà la seguente notazione:

- BST (Binary Search Tree), per indicare un albero binario di ricerca
- RBT (Red Black Tree), per indicare un albero binario Red Black
- Set (Insieme), per indicare la struttura dati "astratta" che rappresenta l'insieme

L'approccio per risolvere il problema posto dalla traccia dal punto di vista delle strutture dati, è stato quello di realizzare la struttura dati "astratta" Set. La struttura dati "concreta" usata è RBT. È necessario precisare che un RBT, è a sua volta un BST.

BST
È necessario definire cosa sia un BST. Un BST è un albero binario che può essere rappresentato da una struttura dati concatenata in cui ogni nodo è un oggetto. Oltre a una chiave (key) e ai dati satellite, ogni nodo dell'albero contiene gli attributi left, right e p che puntano ai nodi che corrispondono rispettivamente, al figlio sinistro, al figlio destro e al padre del nodo. Se manca un figlio o il padre, i corrispondenti attributi contengono il valore NIL. Inoltre, la radice è l'unico nodo che ha padre NIL.

Le chiavi in un albero binario di ricerca sono sempre memorizzate in modo da soddisfare la proprietà dei BST. Nello specifico, se x è un nodo del BST e y è un nodo del sottoalbero sinistro di x, allora $y.key \leq x.key$. Se y è un nodo del sottoalbero destro di x, allora $y.key > x.key$. Dicendo ciò, si precisa che nodi con chiavi uguali sono memorizzati per definizione sempre nel sottoalbero sinistro.

Una delle proprietà fondamentali di un BST e che viene usata ampiamente per risolvere il problema posto dal quesito, è la visita in-order (o simmetrica) del BST. Una visita in-order di un BST permette di elencare ordinatamente tutte le chiavi dell'albero.

Un BST però, può non essere sempre bilanciato. Ciò significa che le operazioni su un BST richiedono un tempo $\Theta(h)$ dove h è l'altezza dell'albero. Per tale motivo, un RBT è visto come un miglioramento del BST. Infatti, il bilanciamento approssimativo che possiede un RBT, permette di garantire che le operazioni elementari richiedano un tempo $\Theta(\log n)$ nel caso peggiore (dove n è il numero di nodi memorizzati nel RBT).

RBT

Un RBT memorizza un bit aggiuntivo di memoria per ogni nodo al fine di definire il suo colore che può essere RED o BLACK. Assegnando dei vincoli al modo in cui i nodi possono essere colorati lungo qualsiasi cammino semplice che va dalla radice a una foglia, questi tipi di alberi RBT garantiscono che nessuno di tali cammini sia lungo più del doppio di qualsiasi altro. Quindi l'albero è approssimativamente bilanciato.

Un RBT soddisfa le seguenti proprietà:

1. Ogni nodo è rosso o nero
2. La radice è nera
3. Ogni foglia è NIL ed è nera
4. Se un nodo è rosso, allora entrambi i suoi figli sono neri
5. Per ogni nodo, tutti i cammini semplici che vanno dal nodo alle foglie sue discendenti contengono lo stesso numero di nodi neri

Siccome ogni nodo foglia è NIL, è possibile ridurre lo spazio occupato in memoria dal RBT utilizzando un nodo sentinella (anche chiamato Nil_T). Tale nodo sentinella sostituirà tutti i nodi foglia NIL e sarà anche il padre del nodo radice dell'albero.

Un'importante nozione relativa al RBT è quella di altezza nera di un nodo x. Cioè il numero di nodi neri lungo un cammino semplice che inizia dal nodo x (ma non lo include) e finisce in una foglia. Per la proprietà 5 vista poc'anzi, l'altezza nera in ogni cammino dal nodo x verso qualsiasi foglia sua discendente, è sempre la stessa. L'altezza nera è indicata con $bh(x)$.

Qui di seguito alcune proprietà e conseguenze di un RBT:

a) Un RBT con radice x, ha almeno $2^{bh(x)} - 1$ nodi interni. È dimostrata per induzione. Supponendo che l'altezza nera di x sia 0, allora x è una foglia e il sottoalbero ha 0 nodi interni. Per il passo induttivo, si considera che ogni nodo figlio ha almeno $2^{bh(x)-1} - 1$ nodi interni. Pertanto, sommando i nodi interni che ha ogni figlio, otteniamo $2^{bh(x)} - 1$ nodi interni per la radice x.

b) In un RBT almeno la metà dei nodi dalla radice ad una foglia deve essere nera. Questa proprietà vale per la proprietà 4 vista precedentemente.

c) In un RBT, nessun percorso da un nodo v ad una foglia, è lungo più del doppio del percorso da v ad un'altra foglia. Questa proprietà vale per la proprietà 4 e 5, viste precedentemente.

d) Un RBT con n nodi interni ha altezza massima pari a $2log(n+1)$. Questa proprietà vale per la proprietà 4 vista precedentemente. Infatti proprio per quest'ultima proprietà, se consideriamo l'altezza nera della radice, questa è almeno $\frac{h}{2}$.
Quindi, se $bh(x) = \frac{h}{2}$ ne consegue che $n \geq 2^{\frac{h}{2}} - 1$. Da cui: $h \leq 2log\,(n+1)$.

Tutte queste proprietà appena viste permettono di dire che le operazioni "classiche" di un RBT possono essere implementate in un tempo $O(log(n))$ in quanto l'albero risulta essere approssimativamente bilanciato.

A questo punto analizziamo a fondo come sono state organizzate le strutture dati.
Nel caso specifico del quesito:

- Ogni insieme è rappresentato mediante un'istanza della classe Set
- Un Set è un RBT e possiede un codice identificativo univoco
- Un RBT è un BST
- Un BST ha un nodo radice e un nodo Nil_T. A partire dal nodo radice, si può poi accedere ai vari discendenti e di conseguenza a tutto l'albero. Ogni nodo è rappresentato mediante un'istanza della classe Node
- Infine, vi è "Sets" che è la classe "problema" che viene istanziata per gestire l'apertura dello stream di input, la lettura da esso, la creazione dei vari insiemi e le operazioni che l'utente richiede di eseguire tra i vari Set

Nell'interfaccia privata di Node, sono presenti i seguenti attributi:

- Una chiave, ovvero un intero
- Il colore, rappresentato mediante un valore booleano (FALSE = BLACK e TRUE = RED)
- Un puntatore a Node parametrizzato dal parametro T, rappresentante il nodo padre
- Un puntatore a Node parametrizzato dal parametro T, rappresentante il figlio sinistro, che a sua volta è radice del sottoalbero sinistro
- Un puntatore a Node parametrizzato dal parametro T, rappresentante il figlio destro, che a sua volta è radice del sottoalbero destro
- Dati satellite parametrizzati dal parametro T

Nell'interfaccia pubblica di Node, sono presenti i seguenti metodi (per semplicità vengono omessi i parametri):

- Setters degli attributi di Node
- Getters degli attributi di Node

Nell'interfaccia privata di BST, sono presenti i seguenti attributi:

- Un puntatore a Node parametrizzato dal parametro T, rappresentante il nodo radice
- Un puntatore a Node parametrizzato dal parametro T, rappresentante il nodo Nil_T

Nell'interfaccia privata di BST, sono presenti i seguenti metodi (per semplicità vengono omessi i parametri):

- transplant(...), un metodo che permette di trapiantare e sostituire il sottoalbero con radice nel nodo u con il sottoalbero con radice nel nodo v. Dove u e v sono radici di sottoalberi radicati passate come parametro al metodo
- inorderVisitBuildArray(...), un metodo che permette di ottenere un puntatore a un array di Node parametrizzati dal parametro T. L'array contiene i nodi del BST ordinati per chiave in senso non decrescente. Questo metodo non è altro che una variante della visita in-order dove l'unica modifica è sostituire la stampa della chiave del nodo con l'istruzione che permette di aggiungere il nodo a un array

Nell'interfaccia protetta di BST, sono presenti i seguenti metodi (per semplicità vengono omessi i parametri):

- setRoot(...), per impostare la radice del BST
- resetNilTNode() per reimpostare il valore della chiave del nodo Nil_T

Nell'interfaccia pubblica di BST, sono presenti i seguenti metodi (per semplicità vengono omessi i parametri):

- advancedInsertNode(...), un metodo che permette di inserire un nodo all'interno del BST e di ricevere come valore di ritorno, un puntatore al Node appena inserito parametrizzato dal parametro T
- insertNode(...) per eseguire il classico inserimento di un Node all'interno del BST
- deleteNode(...) per eseguire la classica cancellazione di un Node dal BST
- Getters degli attributi di BST
- getMin(...), getMax(...), getPredecessor(...), getSuccessor(...) per ottenere un puntatore a Node (parametrizzato dal parametro T) a partire dal nodo x passato come parametro. Come si può evincere il primo metodo, restituisce un puntatore a un Node y tale che y è il nodo con chiave minima del sottoalbero radicato in x.
  Il secondo metodo, analogamente per quello con chiave massima. Invece, il terzo e il quarto metodo restituiscono rispettivamente il predecessore e il successore (se esiste) del nodo x passato come parametro. Infatti, per definizione il successore e il predecessore di un nodo x, in alcuni casi non esiste. In tali casi i metodi restituiscono un puntatore al nodo Nil_T
- searchNode(...), per ricercare un nodo all'interno del BST a partire da un nodo x passato come parametro
- Le classiche visite pre-order, in-order e post-order
- buildSortedArray(...), che fa uso di inorderVisitBuildArray(...) per costruire l'array di nodi del BST ordinati per chiave in senso non decrescente

La scelta di porre setRoot(...) e resetNilTNode() come metodi protetti è motivata dal fatto che volevo costruire la classe BST in maniera tale che le sottoclassi di BST potessero accedere a tali metodi e nel frattempo mantenere "oscurati" tali metodi all'esterno della classe.

Nell'interfaccia di RBT, non vi sono attributi aggiunti. Nell'interfaccia privata di RBT, sono presenti i seguenti metodi (per semplicità vengono omessi i parametri):

- leftRotate(...), per eseguire una rotazione verso sinistra del sottoalbero radicato nel nodo x passato come parametro
- rightRotate(...), per eseguire una rotazione verso destra del sottoalbero radicato nel nodo x passato come parametro
- transplantRB(...), un metodo che permette di trapiantare e sostituire il sottoalbero con radice nel nodo u con il sottoalbero con radice nel nodo v. La differenza rispetto alla transplant(...) di BST, è che si usa il nodo sentinella Nil_T al posto di NIL e che l'assegnazione finale viene sempre eseguita ($v.p = u.p$)
- deleteFixUp(...), un metodo che permette di andare a rispristinare le proprietà di RBT in seguito a una cancellazione (qualora non dovessero essere più verificate)

Nell'interfaccia pubblica di RBT, sono presenti i seguenti metodi (per semplicità vengono omessi i parametri):

- insertNodeRB(...), per inserire un nodo nel RBT
- deleteNodeRB(...), per cancellare un nodo dal RBT

Nell'interfaccia privata di Set, sono presenti i seguenti attributi:

- Codice identificativo univoco per il Set

Nell'interfaccia privata di Set, sono presenti i seguenti metodi (per semplicità vengono omessi i parametri):

- setID(...)
- merge(...) per eseguire la fusione di due array di Node parametrizzati dal parametro T ordinati in base al loro campo chiave in senso non decrescente. Il metodo restituisce un puntatore a un array di Node parametrizzati dal parametro T e contenente tutti i nodi dei due array
- removeDuplicate(...), un metodo per rimuovere i Node che risultano essere duplicati in base alla loro chiave, da un array di Node parametrizzati dal parametro T passato sotto forma di puntatore

Nell'interfaccia pubblica di Set, sono presenti i seguenti metodi (per semplicità vengono omessi i parametri):

- getID() e getCounterID() per restituire rispettivamente il codice identificativo univoco del Set e il numero totale di Set istanziati
- unionOperation(...), intersectOperation(...) e differenceOperation(...). I 3 metodi che permettono di eseguire le operazioni insiemistiche richieste dal quesito.
  Ognuno di essi prende come parametro il secondo Set sotto forma di puntatore a Set parametrizzato dal parametro T. Ognuno di essi fa la relativa operazione con il Set sul quale il metodo è invocato. Ognuno di essi restituisce un puntatore a un nuovo Set parametrizzato dal parametro T
- printSet() per stampare le chiavi dei Node appartenenti al Set in ordine non decrescente
- insertElement(...), un metodo per inserire un nuovo Node all'interno di un Set. Il Node è creato in base alla chiave e ai dati satellite passati come parametro

Nell'interfaccia privata di Sets, sono presenti i seguenti attributi:

- Un puntatore a un array di puntatori a oggetti di tipo Set parametrizzati dal parametro T
- Una stringa indicante il path del file di input
- Un puntatore a uno stream di input per gestire la lettura, l'apertura e la chiusura del file contenente gli insiemi

Nell'interfaccia privata di Sets, sono presenti i seguenti metodi (per semplicità vengono omessi i parametri):

- Setters degli attributi di Sets
- Getters degli attributi di Sets
- printError(...), per stampare gli errori generati dall'utente a run-time
- openInputFileStream() e closeInputFileStream() rispettivamente per aprire e chiudere lo stream di input
- buildSets() che costruisce i Set a partire dal file di input prescelto e li inserisce all'interno dell'array di puntatori a Set parametrizzati dal parametro T
- checkInputSetID(...) che verifica se l'insieme scelto dall'utente (mediante codice identificativo), sia valido o meno. In caso positivo restituisce TRUE, altrimenti FALSE

Nell'interfaccia pubblica di Sets, sono presenti i seguenti metodi (per semplicità vengono omessi i parametri):

- printSets(), per stampare le chiavi di tutti i Set in ordine non decrescente
- printSetWithID(...) per stampare le chiavi del Set passato come parametro (sotto forma di codice identificativo) in ordine non decrescente. Questo solo se è presente un Set con tale codice identificativo
- requestSetOperation(setOperationID, firstSetID, secondSetID), permette di eseguire le operazioni insiemistiche canoniche. In particolar modo, esegue l'operazione identificata da setOperationID, tra l'insieme identificato da firstSetID e l'insieme identificato da secondSetID (se presenti dei Set con tali codici identificativi e se presente un'operazione insiemistica con tale codice identificativo). L'insieme risultante (se generato), viene restituito sotto forma di puntatore a un oggetto di tipo Set parametrizzato dal parametro T.

Per quanto riguarda i distruttori, il loro unico ruolo è quello di deallocare la memoria dinamicamente allocata precedentemente per nodi, vector di puntatori a Set e stream di input.

Per quanto riguarda i costruttori di Node, BST, RBT e Set non hanno un particolare comportamento da segnalare, oltre a quello dell'assegnazione e inizializzazione degli eventuali attributi presenti. Invece, il costruttore di Sets si occupa anche dell'apertura di uno stream per la lettura da file e la creazione dei vari Set (in base a quelli forniti nel file di input).

Formato dati in input/output

Gli elementi degli insiemi (numeri interi) sono memorizzati all'interno di un file di testo. Gli elementi appartenenti ad uno stesso insieme si trovano su una stessa riga separati da uno spazio. Righe successive corrispondono ai diversi insiemi. Ogni insieme fornito in input verrà rappresentato mediante un RBT.

Di conseguenza su tale input è possibile fare soltanto l'assunzione che gli elementi degli insiemi siano interi. Altri dati di input forniti al programma consistono nelle varie scelte effettuate dall'utente mediante menù. L'utente una volta avviato il programma, si ritroverà a prendere alcune decisioni come:

- scegliere un primo insieme da considerare per eseguire l'operazione insiemistica
- scegliere un secondo insieme da considerare per eseguire l'operazione insiemistica
- scegliere l'operazione insiemistica da eseguire su i due insiemi appena scelti
- scegliere se continuare l'esecuzione del programma eseguendo altre operazioni insiemistiche o meno

In particolar modo nelle prime 3 scelte, l'utente inserisce dei numeri interi positivi per scegliere i Set e l'operazione insiemistica da svolgere, rispettivamente mediante codice identificativo univoco degli insiemi e mediante codice identificativo univoco dell'operazione insiemistica.
Per quanto riguarda l'ultima scelta, l'utente può scegliere di continuare digitando 'Y' o 'y'; altrimenti qualsiasi altro input permetterà di terminare l'esecuzione.

In output viene mostrato un menù che permette all'utente di eseguire le varie scelte di cui sopra. Inoltre, in output si ha di volta in volta il risultato dell'operazione insiemistica sotto forma di insieme risultante che viene stampato a video con i suoi elementi ordinati per chiave in senso non decrescente. Prima di inserire i vari codici per compiere le scelte, l'utente ha a disposizione anche la stampa di tutti gli insiemi, i codici identificativi univoci e gli elementi per ognuno di essi (anche in questo caso ordinati per chiave in senso non decrescente).

Descrizione Algoritmo

Per la risoluzione del problema, assumiamo che il file contenente i dati di input sopra descritti sia già stato aperto, letto e preso in considerazione. Di conseguenza abbiamo già a disposizione un'istanza della classe "Sets" contenente i "Set" sui quali andare ad operare ed eseguire le operazioni insiemistiche. L'obiettivo dell'algoritmo è quello di poter compiere le operazioni insiemistiche di unione, intersezione e differenza su alberi Red Black che vanno a rappresentare gli insiemi.
L'operazione insiemistica da fare viene identificata dal primo intero positivo passato come parametro al metodo requestOperation(...). Viene eseguita l'operazione insiemistica tra l'insieme identificato dal secondo intero positivo passato come parametro al metodo requestOperation(...) e l'insieme identificato dal terzo intero positivo passato come parametro al metodo requestOperation(...). Il risultato sotto forma di puntatore a insieme è memorizzato in una variabile che verrà poi restituita all'utente.

Vi sono alcuni aspetti in comune tra i tre algoritmi che eseguono le operazioni insiemistiche. Innanzitutto, si istanzia un nuovo puntatore a Set che sarà il Set risultante che verrà poi restituito al chiamante. Si esegue una variante della visita in-order dei due Set coinvolti nell'operazione insiemistica. Infatti, si ricorda che un Set è un RBT e quindi per la proprietà analizzata nella sezione "Descrizione strutture dati" relativa alla visita in-order su un BST, si riescono ad elencare gli elementi appartenenti al Set in maniera ordinata per chiave in senso non decrescente. Viene usata una variante della visita in-order in quanto, l'ordinamento degli elementi del Set permette di eseguire le operazioni insiemistiche di seguito riportate, in maniera efficiente.

Un altro aspetto in comune è la rimozione dei duplicati. Infatti, come analizzato nella sezione "Descrizione problema" gli insiemi risultanti non conterranno elementi con chiavi uguali. La rimozione dei duplicati grazie al fatto che gli elementi dell'insieme sono ordinati per chiave, può essere implementata in maniera semplice mediante un metodo che va a controllare di volta in volta se l'elemento corrente ha la stessa chiave di quello precedente. In tal caso, si provvede a rimuovere l'elemento corrente. Questo viene fatto scorrendo tutto l'elenco degli elementi appartenenti all'insieme ordinati per chiave.

Analizziamo singolarmente i vari algoritmi che permettono di ottenere il risultato richiesto, una volta che l'utente ha inserito gli input validi per eseguire la relativa operazione insiemistica.

Unione

Qualora esistano due insiemi identificati dai codici identificativi univoci inseriti dall'utente, allora si procede alla chiamata del metodo union(...) analizzato nella sezione "Descrizione strutture dati". Questo metodo è invocato su un Set e riceve come parametro un altro Set col quale eseguire l'unione. Il Set risultante è restituito come valore di ritorno sotto forma di puntatore a un oggetto di tipo Set parametrizzato dal parametro T.

Si esegue una variante della visita in-order sul primo e sul secondo insieme coinvolto nell'operazione insiemistica. È una variante perché, invece di stampare le chiavi degli elementi in maniera ordinata, memorizza tali elementi in un array che verrà poi restituito al chiamante. In tal modo alla fine della visita, l'array conterrà tutti gli elementi dell'insieme ordinati per chiave. Tale variante della visita consiste nella chiamata eseguita a buildSortedArray().

A questo punto, viene invocato il metodo merge(...) che restituirà l'insieme unione risultante sotto forma di puntatore ad array di nodi parametrizzati dal parametro T. L'union è eseguita fra i due insiemi che gli sono stati passati come parametro sotto forma di puntatori ad array di nodi parametrizzati dal parametro T. L'idea è quella di andare a realizzare un ciclo che viene eseguito $m + n$ volte dove m è il numero di elementi del primo array e n è il numero di elementi del secondo array.
All'interno di questo ciclo, si verifica ogni volta se sono stati inseriti tutti gli elementi del primo array. In tal caso si deve soltanto procedere a inserire tutti gli elementi del secondo array. Poi, si verifica ogni volta se sono stati inseriti tutti gli elementi del secondo array. In tal caso si deve soltanto procedere a inserire tutti gli elementi del primo array.
Altrimenti se ci sono ancora elementi sia nel primo che nel secondo array, si procede a inserire nell'insieme risultante l'elemento che ha chiave più piccola fra i due elementi attualmente presi in considerazione dei due array.

A questo punto si ritorna all'interno del metodo union(...) che ora procederà a rimuovere i duplicati dall'array risultante fornitogli dal metodo merge(...). Una volta fatto ciò, basta inserire tutti gli elementi presenti in questo array risultante senza duplicati, all'interno del nuovo insieme istanziato inizialmente. Inserire questi elementi all'interno del nuovo insieme, vuol dire eseguire degli inserimenti nella struttura dati RBT.

Una volta terminati gli inserimenti all'interno dell'insieme, si procede a restituire tale insieme al chiamante: requestOperation(...). All'interno di requestOperation(...) si inserirà questo nuovo "Set" all'interno dell'array di Set memorizzati dall'istanza della classe "Sets" che si sta utilizzando per eseguire tali operazioni insiemistiche. Inoltre, si restituisce il Set risultante dalla precedente operazione di union(...) all'utente. Infine, si procederà alla stampa del nuovo insieme ottenuto.

La correttezza dell'algoritmo è dimostrata in base alla definizione dell'operazione insiemistica di unione presente nella sezione "Descrizione problema".

Lo pseudo-codice seguente implementa la suddetta idea:

```
1.  merge(firstSetArray, secondSetArray)
2.          thirdSetArray = ∅
3.          i = 0
4.          j = 0
5.          for k = 0 to (firstSetArray.size + secondSetArray.size)
6.                  if i == firstSetArray.size
7.                          thirdSetArray = thirdSetArray ∪ secondSetArray[j]
8.                          j = j + 1
9.                          continue
10.                 if j == secondSetArray.size
11.                         thirdSetArray = thirdSetArray ∪ firstSetArray[i]
12.                         i = i + 1
13.                         continue
14.
15.                 if firstSetArray[i].key < secondSetArray[i].key
16.                         thirdSetArray = thirdSetArray ∪ firstSetArray[i]
17.                         i = i + 1
18.                 else
19.                         thirdSetArray = thirdSetArray ∪ secondSetArray[j]
20.                         j = j + 1
21.
22.         return thirdSetArray
23.
24. removeDuplicate(array)
25.         for i = 1 to array.size
26.                 if array[i – 1].key == array[i].key
27.                         array.erase(array[i])
28.                         i = i -1
29.
30.         return array
31.
32. union(secondSet)
33.         resultSet = ∅
34.         firstSetArray = firstSet.buildSortedArray()
35.         secondSetArray = secondSet.buildSortedArray()
36.         resultSetArray = merge(firstSetArray, secondSetArray)
37.
38.         removeDuplicate(resultSetArray)
39.
40.         for i = 0 to resultSetArray.size
41.                 resultSet.insertNodeRB(resultSetArray[i].key, resultSetArray[i].data)
42.
43.         return resultSet
```

Dalla riga 1 alla riga 22, è presente il metodo merge(...) che per eseguire la fusione di due array ordinati in base al loro campo chiave in senso non decrescente. In particolare, alla riga 2 - 4 vengono inizializzati rispettivamente un nuovo array risultante, la variabile i e la variabile j. A questo punto alla riga 5 si procede con il ciclo che di volta in volta inserisce l'elemento j-esimo del secondo array, se sono terminati gli elementi da inserire del primo array. Oppure inserisce l'elemento i-esimo del primo array, se sono terminati gli elementi da inserire del secondo array. Il tutto incrementando opportunamente i e j.

Alla riga 15, qualora dovessero essere presenti sia elementi nel primo array che nel secondo array, allora si inserisce quello che tra i due ha chiave minore, incrementando opportunamente i e j. Alla riga 22 si restituisce l'array risultante.

Alla riga 24 si trova il metodo che permette di rimuovere i duplicati da un array che è stato già descritto precedentemente.
Infine dalla riga 32 alla riga 43, vi è il metodo union(...) che una volta ottenuti gli array con gli elementi ordinati per chiave in senso non decrescente del primo e del secondo Set, procede ad invocare la merge(...). A questo punto si invoca il metodo per rimuovere i duplicati e poi si passa all'inserimento di ogni elemento presente nell'array risultante privo di elementi con chiavi uguali, all'interno di un nuovo Set che viene poi restituito all'utente.

Intersezione

Qualora esistano due insiemi identificati dai codici identificativi univoci inseriti dall'utente, allora si procede alla chiamata del metodo intersect(...) analizzato nella sezione "Descrizione strutture dati". Questo metodo è invocato su un Set e riceve come parametro un puntatore a un altro Set col quale eseguire l'intersezione. Il Set risultante è restituito come valore di ritorno sotto forma di puntatore a un oggetto di tipo Set parametrizzato dal parametro T.

Si esegue una variante della visita in-order sul primo e sul secondo insieme coinvolto nell'operazione insiemistica. È una variante perché, invece di stampare le chiavi degli elementi in maniera ordinata, memorizza tali elementi in un array che verrà poi restituito al chiamante. In tal modo alla fine della visita, l'array conterrà tutti gli elementi dell'insieme ordinati per chiave. Tale variante della visita consiste nella chiamata eseguita a buildSortedArray().

A questo punto, l'idea è quella di andare a realizzare un ciclo che viene eseguito fintanto che ci sono altri elementi presenti all'interno del primo e del secondo array che contengono gli elementi dei due insiemi ordinati per chiave in senso non decrescente. All'interno di questo ciclo, si confronta di volta in volta se l'elemento i-esimo del primo array ha chiave minore dell'elemento j-esimo del secondo array. In tal caso vuol dire che è necessario incrementare la variabile i, in maniera tale da andare a verificare se l'elemento successivo all'interno del primo array sia corrispondente all'elemento j-esimo del secondo array.
Altrimenti se l'elemento i-esimo del primo array ha chiave maggiore dell'elemento j-esimo del secondo array, allora è necessario incrementare la variabile j, in maniera tale da andare a verificare se l'elemento successivo all'interno del secondo array sia corrispondente all'elemento i-esimo del primo array. Altrimenti se l'elemento i-esimo del primo array ha chiave uguale all'elemento j-esimo del secondo array, allora vuol dire che tale elemento deve far parte dell'insieme intersezione risultante. In questo caso si incrementa sia la variabile i, che la variabile j.

Nel momento in cui il ciclo termina vuol dire che sono terminati gli elementi da prendere in considerazione nel primo array o nel secondo array. La correttezza è dimostrata. Infatti, qualora dovessero terminare gli elementi del secondo array ed esservi ancora altri elementi nel primo array o viceversa, l'intersezione termina dal momento che in entrambi i casi un ulteriore elemento non potrebbe appartenere a entrambi gli array. Questo perché, per la proprietà dell'ordinamento, si ha la certezza che un ulteriore elemento avrà una chiave non inferiore agli elementi finora analizzati dal momento che i due array risultano essere ordinati per chiave degli elementi.

A questo punto si ritorna all'interno del metodo intersect(...) che ora procederà a rimuovere i duplicati dall'array risultante. Una volta fatto ciò, basta inserire tutti gli elementi presenti in questo array risultante senza duplicati, all'interno del nuovo insieme istanziato inizialmente. Inserire questi elementi vuol dire eseguire degli inserimenti nella struttura dati RBT.

Una volta terminati gli inserimenti all'interno dell'insieme, si procede a restituire tale insieme al chiamante: requestOperation(...). All'interno di requestOperation(...) si inserirà questo nuovo "Set" all'interno dell'array di Set memorizzati dall'istanza della classe "Sets" che si sta utilizzando per eseguire tali operazioni insiemistiche. Inoltre, si restituisce il Set risultante dalla precedente operazione di intersect(...) all'utente. Infine, si procederà alla stampa del nuovo insieme ottenuto.

La correttezza dell'algoritmo è dimostrata in base alla definizione dell'operazione insiemistica di intersezione presente nella sezione "Descrizione problema".

Lo pseudo-codice seguente implementa la suddetta idea:

```
1.  removeDuplicate(array)
2.      for i = 1 to array.size
3.          if array[i – 1].key == array[i].key
4.              array.erase(array[i])
5.              i = i -1
6.
7.      return array
8.
9.  intersect(secondSet)
10.     resultSet = ∅
11.     firstSetArray = firstSet.buildSortedArray()
12.     secondSetArray = secondSet.buildSortedArray()
13.     resultSetArray = ∅
14.     i = 0
15.     j = 0
16.
17.     while (i < firstSetArray.size AND j < secondSetArray.size)
18.         if (firstSetArray[i].key < secondSetArray[j].key)
19.             i = i + 1
20.         else if (firstSetArray[i].key > secondSetArray[j].key)
21.             j = j + 1
22.         else
23.             resultSetArray = resultSetArray ∪ firstSetArray[i]
24.             i = i + 1
25.             j = j + 1
26.
27.     removeDuplicate(resultSetArray)
28.
29.     for i = 0 to resultSetArray.size
30.         resultSet.insertNodeRB(resultSetArray[i].key, resultSetArray[i].data)
31.
32.     return resultSet
```

Dalla riga 1 alla riga 7, vi è il metodo che permette di rimuovere i duplicati da un array che è stato già descritto precedentemente.

Dalla riga 9 alla riga 32, vi è il metodo intersect(...) che inizia generando gli array con gli elementi ordinati per chiave in senso non decrescente del primo e del secondo Set, inizializzando un nuovo Set, il rispettivo array e le variabili i e j. Successivamente, si esegue il ciclo while che permette di prelevare dall'array contenente gli elementi del primo insieme ordinati per chiave in ordine non decrescente, quello che è presente anche nel secondo array contenente gli elementi del secondo insieme ordinati per chiave in ordine non decrescente. A questo punto si invoca il metodo per rimuovere i duplicati e poi si passa all'inserimento di ogni elemento presente nell'array risultante privo di elementi con chiavi uguali, all'interno di un nuovo Set che viene poi restituito all'utente.

Differenza

Qualora esistano due insiemi identificati dai codici identificativi univoci inseriti dall'utente, allora si procede alla chiamata del metodo difference(...) analizzato nella sezione "Descrizione strutture dati". Questo metodo è invocato su un Set e riceve come parametro un puntatore a un altro Set col quale eseguire la differenza. Il Set risultante è restituito come valore di ritorno sotto forma di puntatore a un oggetto di tipo Set parametrizzato dal parametro T.

Si esegue una variante della visita in-order sul primo e sul secondo insieme coinvolto nell'operazione insiemistica. È una variante perché, invece di stampare le chiavi degli elementi in maniera ordinata, memorizza tali elementi in un array che verrà poi restituito al chiamante. In tal modo alla fine della visita, l'array conterrà tutti gli elementi dell'insieme ordinati per chiave. Tale variante della visita consiste nella chiamata eseguita a buildSortedArray().

A questo punto l'idea è quella di andare a realizzare un ciclo che viene eseguito fintanto che ci sono altri elementi presenti all'interno del primo e del secondo array che contengono gli elementi dei due insiemi ordinati per chiave in senso non decrescente. All'interno di questo ciclo, si confronta di volta in volta se l'elemento i-esimo del primo array ha chiave minore dell'elemento j-esimo del secondo array. In tal caso è possibile quindi inserire l'elemento i-esimo del primo array all'interno dell'array differenza risultante. Dopodiché si incrementa i.
È possibile inserire l'elemento dal momento che gli array sono ordinati per chiave e quindi se è più piccolo dell'elemento j-esimo del secondo array, sarà anche più piccolo di tutti gli altri elementi che saranno presenti dopo l'elemento j-esimo all'interno del secondo array.
Altrimenti, se l'elemento i-esimo del primo array ha chiave maggiore dell'elemento j-esimo del secondo array, è necessario incrementare la variabile j. Questo perché un elemento con la stessa chiave di quello i-esimo del primo array potrebbe trovarsi in una locazione successiva del secondo array. Altrimenti se l'elemento i-esimo del primo array ha chiave uguale all'elemento j-esimo del secondo array, allora per la definizione dell'operazione insiemistica di differenza l'elemento non deve essere preso. Vengono incrementate i e j.

Nel momento in cui il ciclo termina vuol dire che sono terminati gli elementi da prendere in considerazione nel primo array o nel secondo array. A differenza dell'intersezione però, non è possibile fermarsi qui. Infatti, qualora dovessero terminare gli elementi del secondo array ed esservi ancora altri elementi nel primo array, è necessario inserire gli elementi restanti del primo array all'interno dell'insieme risultante, per la definizione di operazione insiemistica di differenza. Invece, qualora dovessero terminare gli elementi del primo array ed esservi ancora altri elementi nel secondo array, è possibile passare al passo successivo per la definizione di operazione insiemistica di differenza.

A questo punto si ritorna all'interno del metodo difference(...) che ora procederà a rimuovere i duplicati dall'array risultante. Una volta fatto ciò, basta inserire tutti gli elementi presenti in questo array risultante senza duplicati, all'interno del nuovo insieme istanziato inizialmente. Inserire questi elementi vuol dire eseguire degli inserimenti nella struttura dati RBT.

Una volta terminati gli inserimenti all'interno dell'insieme, si procede a restituire tale insieme al chiamante: requestOperation(...). All'interno di requestOperation(...) si inserirà questo nuovo "Set" all'interno dell'array di Set memorizzati dall'istanza della classe "Sets" che si sta utilizzando per eseguire tali operazioni insiemistiche. Inoltre, si restituisce il Set risultante dalla precedente operazione di difference(...) all'utente. Infine, si procederà alla stampa del nuovo insieme ottenuto.

La correttezza dell'algoritmo è dimostrata in base alla definizione dell'operazione insiemistica di differenza presente nella sezione "Descrizione problema".

Lo pseudo-codice seguente implementa la suddetta idea:

```
1.  removeDuplicate(array)
2.      for i = 1 to array.size
3.          if array[i – 1].key == array[i].key
4.              array.erase(array[i])
5.              i = i -1
6.
7.      return array
8.
9.  difference(secondSet)
10.     resultSet = ∅
11.     firstSetArray = firstSet.buildSortedArray()
12.     secondSetArray = secondSet.buildSortedArray()
13.     resultSetArray = ∅
14.     i = 0   j = 0
15.
16.     while (i < firstSetArray.size AND j < secondSetArray.size)
17.         if (firstSetArray[i].key < secondSetArray[j].key)
18.             resultSetArray = resultSetArray ∪ firstSetArray[i]
19.             i = i + 1
20.         else if (firstSetArray[i].key > secondSetArray[j].key)
21.             j = j + 1
22.         else
23.             i = i + 1
24.             j = j + 1
25.
26.     while (i < firstSetArray.size)
27.         resultSetArray = resultSetArray ∪ firstSetArray[i]
28.         i = i + 1
29.
30.     removeDuplicate(resultSetArray)
31.     for i = 0 to resultSetArray.size
32.         resultSet.insertNodeRB(resultSetArray[i].key, resultSetArray[i].data)
33.
34.     return resultSet
```

Dalla riga 1 alla riga 7, vi è il metodo che permette di rimuovere i duplicati da un array che è stato già descritto precedentemente.

Dalla riga 9 alla riga 34, vi è il metodo difference(...) che inizia generando gli array con gli elementi ordinati per chiave in senso non decrescente del primo e del secondo Set, inizializzando un nuovo Set, il rispettivo array e le variabili i e j. Successivamente, si esegue il ciclo while che permette di prelevare dall'array contenente gli elementi del primo insieme ordinati per chiave in ordine non decrescente, quello che non è presente nel secondo array contenente gli elementi del secondo insieme ordinati per chiave in ordine non decrescente. Ora, a differenza dell'intersect, si procede con un ulteriore ciclo while nel quale si prelevano tutti gli elementi rimanenti all'interno del primo array (cioè quelli presenti dopo firstSetArray[i]), e si inseriscono all'interno dell'array differenza risultante.

A questo punto si invoca il metodo per rimuovere i duplicati e poi si passa all'inserimento di ogni elemento presente nell'array risultante privo di elementi con chiavi uguali, all'interno di un nuovo Set che viene poi restituito all'utente.

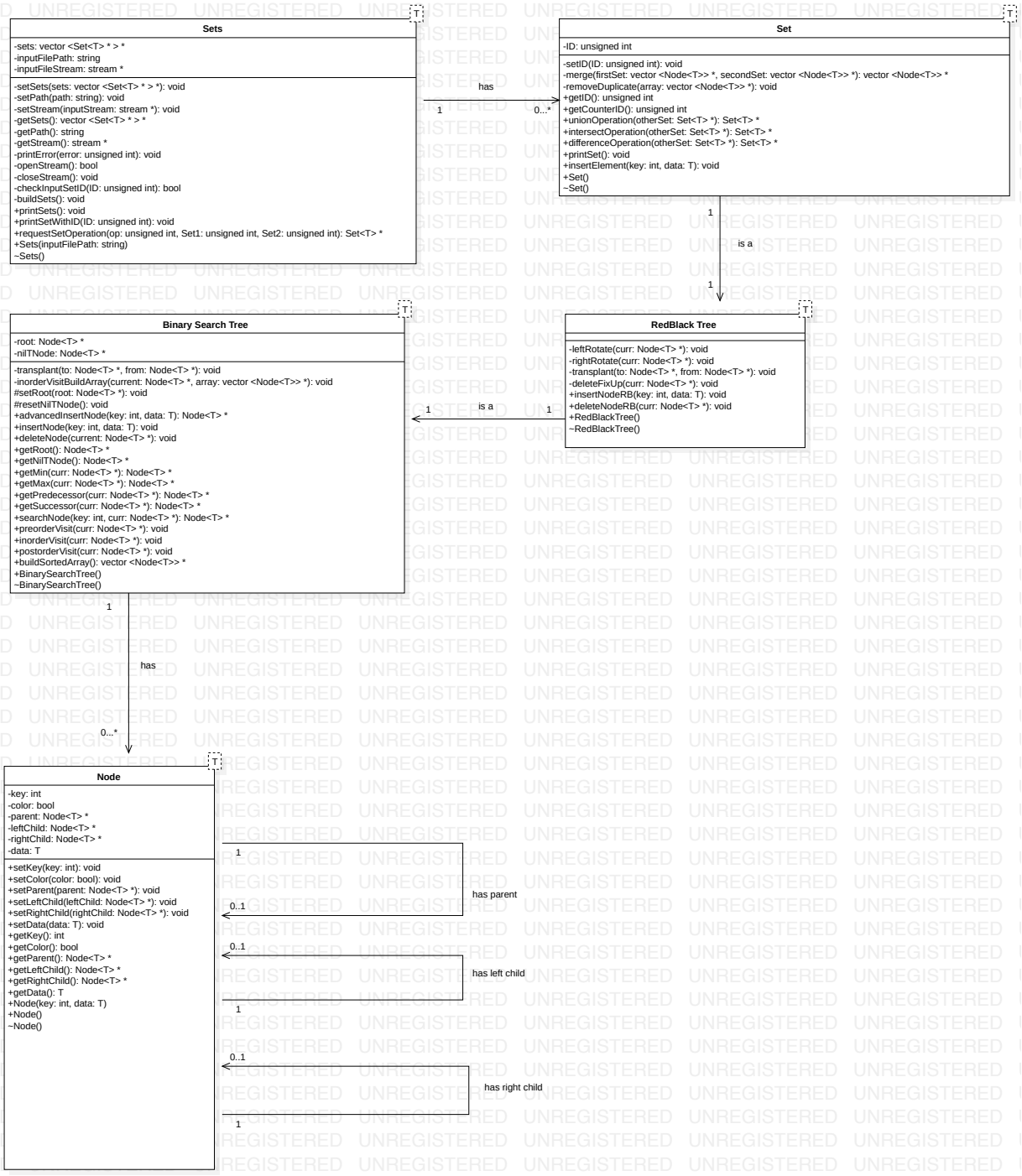
buildSortedArray()

Infine, è necessario un particolare approfondimento per quella che è la variante della visita in-order che permette di costruire un array contenente gli elementi di un insieme ordinati per chiave in senso non decrescente.
Ecco lo pseudo-codice:

```
1.  buildSortedArray()
2.      sortedArray = ∅
3.      inorderVisitBuildArray(root, sortedArray, Nil_T)
4.      return sortedArray
5.
6.  inorderVisitBuildArray(curr, sortedArray, Nil_T)
7.      if curr != Nil_T
8.          inorderVisitBuildArray(curr.left, sortedArray, Nil_T)
9.          sortedArray = sortedArray ∪ curr
10.         inorderVisitBuildArray(curr, sortedArray, Nil_T)
```

Alla riga 2 si inizializza un nuovo array. Alla riga 3 si invoca la inorderVisitBuildArray a partire dalla radice che, ricorsivamente, memorizza gli elementi in ordine simmetrico.
Alla riga 4 si restituisce l'array ordinato.

Class Diagram



La classe Sets ha al suo interno un puntatore a un vector di puntatori a Set parametrizzati dal parametro T. Quindi, la classe Sets ha dei Set. La classe Sets è vista come la classe "problema" così come descritto nella sezione "Descrizione strutture dati" a pagina 7. Infatti, permette di contenere gli insiemi forniti in input e quelli generati dalle operazioni insiemistiche richieste dall'utente. Inoltre, la classe Sets ha come attributi un puntatore a uno stream di input e una stringa indicante il path del file come indicato nella sezione "Descrizione strutture dati".

La classe Sets ha una relazione 1 ... 0...* con Set, ad indicare che un'istanza della classe Sets può avere all'interno del vector a cui il primo dei suoi attributi punta, zero o più Set. La classe Sets non presenta altre relazioni.

La classe Sets svolge buona parte del lavoro nel momento in cui viene istanziato un oggetto di tipo Sets. Infatti, richiede un inputFilePath cioè "nome.estensione" del file da aprire
(es. "input.txt"), avvia lo stream, apre il file e costruisce i vari Set. A questo punto l'utente può iniziare a interagire con il menù, richiedendo delle operazioni insiemistiche fra gli insiemi stampati a video.
Gli altri metodi sono per lo più setters, getters, metodi di gestione dello stream, per la stampa degli errori, per la verifica degli ID relativi agli insiemi inseriti dall'utente, per le stampe di un insieme in particolare o di tutti gli insiemi e quello per richiedere un'operazione insiemistica tra due insiemi ben specificati mediante ID.

La classe Set eredita da RBT. Un Set è un RBT. Un RBT è un BST. Quindi Set ha una relazione con molteplicità 1 ... 1 di tipo "is a" con RBT.
La classe Set eredita (in maniera privata così come specificato nel codice implementativo) da RBT che a sua volta eredita (in maniera pubblica così come specificato nel codice implementativo) da BST. Set eredita "in privato" da RBT perché ho preferito fare in maniera tale che, eventuali sottoclassi di Set non avrebbero poi ereditato metodi e attributi rispettivamente della classe RBT e BST. Cioè, Set smette di propagare alle classi discendenti l'interfaccia che ha ereditato a sua volta dai suoi "ancestor". Allo stesso tempo però, ho ritenuto necessario che altre classi avessero comunque modo di vedere Set per quello che effettivamente è, cioè un RBT. Quindi "Sets" ha comunque modo di accedere all'interfaccia pubblica di RBT e di BST. Inoltre, ho fatto questo tipo di scelta in maniera tale che dal classico "main", istanziando un oggetto di tipo Set o istanziando un oggetto di tipo Sets, non si potesse comunque accedere alla struttura dati concreta che c'è dietro Set. In questo modo si nasconde il funzionamento e l'implementazione, si realizza l'information hiding e si preserva la sicurezza dei dati membro.

La classe Set oltre a setters e getters, ha il metodo per il merge di due array di Node parametrizzati dal parametro T, il metodo per la rimozione dei duplicati da un array dello stesso tipo e i metodi descritti nella sezione "Descrizione Algoritmo", cioè union(...), intersect(...), difference(...), un metodo per la stampa dell'insieme e uno per l'inserimento di un nuovo elemento all'interno dell'insieme.

La classe RBT eredita “in pubblico” da BST. RBT è un BST. Quindi RBT ha una relazione con molteplicità 1 ... 1 di tipo “is a” con BST.
La classe RBT presenta i classici metodi per le rotazioni a sinistra e a destra, per il trapianto di sottoalberi, per l’inserimento, la cancellazione e il fix up che può avvenire successivamente alla cancellazione per ripristinare le proprietà di un RBT.

RBT eredita “in pubblico” da BST perché così facendo:

- Le classi derivate da RBT (come Set), hanno modo di accedere all’interfaccia pubblica di BST.
- Le classi che hanno una relazione di tipo “has” con RBT, sono in grado di accedere anche all’interfaccia pubblica di BST.

Con un’ereditarietà privata o protetta, non sarebbe stato possibile implementare i punti suddetti.

La classe BST “ha” dei Node perché un BST è composto da 0 o più nodi e infatti la relazione tra BST e Node ha molteplicità 1 ... 0...* ed è di tipo “has”. BST ha un puntatore a Node parametrizzato da T e uno al nodo sentinella Nil_T dello stesso tipo. Per quanto riguarda i metodi, la classe BST possiede il metodo per eseguire il trapianto di sottoalberi in un BST, le classiche 3 visite di un BST, più una variante di quella in-order per costruire un array di Node ordinati per chiave in senso non decrescente, setters, getters degli attributi di BST, e i metodi canonici per trovare il minimo, il massimo, il predecessore, il successore in un BST come descritto nella sezione “Descrizione strutture dati”. Inoltre, vi è il metodo per la ricerca di un Node in un BST con una data chiave a partire da un certo nodo del BST, il metodo per l’inserimento e quello per la cancellazione di un Node dal BST.

La classe Node presenta diverse relazioni. Un Node può:

- avere o meno un padre
- avere o meno un figlio sinistro
- avere o meno un figlio destro

Quindi Node ha una relazione con sé stesso con molteplicità 1 ... 0...1 ed è denominata “has parent” e altre due relazioni con la stessa molteplicità ma denominate rispettivamente “has left child” e “has right child”. Un Node ha diversi attributi che lo caratterizzano come è stato menzionato anche in precedenza nella sezione “Descrizione strutture dati”. Per quanto riguarda i metodi dell’interfaccia di Node, non vi sono particolari metodi da segnalare oltre ai classici setters e getters.

Tutte le classi su menzionate sono parametrizzate dal parametro T. La classe template T in questo specifico caso, non è sfruttata al meglio delle sue potenzialità. Una possibile applicazione generalizzata della classe template T è quella di avere nel file di input, oltre a interi intesi come elementi dei vari Set, anche dei dati satellite per ogni elemento. Però, per come è stato progettato il metodo buildSets(), attualmente non è in grado di gestire tale funzionalità.
Attualmente gli interi presenti in un insieme vengono memorizzati nel Node sia all’interno del campo key che all’interno del campo data.

Studio complessità

Per il seguente studio della complessità di tempo e di spazio si assume che m è il numero di Node del primo Set considerato e n è il numero di Node del secondo Set considerato.
Analizziamo la complessità di tempo di union(...):

- Siccome buildSortedArray(...) è una visita in-order e siccome vi sono due chiamate a questo metodo, allora si impiega un tempo di $\Theta(m+n)$
- La funzione merge(...) impiega un tempo $\Theta(m+n)$ per eseguire la fusione di due array ordinati
- La funzione removeDuplicate(...) impiega un tempo $\Theta(m+n)$ per eseguire la rimozione dei duplicati da un array di $m+n$ elementi
- L'ultimo for permette di inserire al più $m+n$ nodi all'interno di un RBT. Siccome l'inserimento in un RBT impiega un tempo $\Theta(log(n))$ e siccome vengono inseriti al più $m+n$ nodi, allora quest'ultimo passaggio impiega un tempo di

$$\Theta((m+n)(log(m+n))$$

A livello asintotico se consideriamo che $m+n$ può essere assimilato a n, allora si nota che in ogni caso la complessità risulta essere sempre dominata dall'ultimo passaggio di inserimento di n nodi. Quindi in totale la complessità della union(...) a livello asintotico è di: $\Theta(n(log(n))$.

Analizziamo la complessità di tempo di intersect(...):

- Siccome buildSortedArray(...) è una visita in-order e siccome vi sono due chiamate a questo metodo, allora si impiega un tempo di $\Theta(m+n)$
- Il ciclo while che permette di prelevare gli elementi idonei a far parte dell'insieme intersezione risultante, impiega un tempo $\Theta(n)$ nel caso peggiore, cioè quando entrambi gli array contengono gli stessi elementi e lo stesso numero di elementi
- La funzione removeDuplicate(...) impiega un tempo $\Theta(m+n)$ per eseguire la rimozione dei duplicati da un array di $m+n$ elementi
- L'ultimo for permette di inserire al più $m+n$ nodi all'interno di un RBT. Siccome l'inserimento in un RBT impiega un tempo $\Theta(log(n))$ e siccome vengono inseriti al più $m+n$ nodi, allora quest'ultimo passaggio impiega un tempo di

$$\Theta((m+n)(log(m+n))$$

A livello asintotico se consideriamo che $m+n$ può essere assimilato a n, allora si nota che in ogni caso la complessità risulta essere sempre dominata dall'ultimo passaggio di inserimento di n nodi. Quindi in totale la complessità della intersect(...) a livello asintotico è di: $\Theta(n(log(n))$.

Analizziamo la complessità di tempo di difference(...):

- Siccome buildSortedArray(...) è una visita in-order e siccome vi sono due chiamate a questo metodo, allora si impiega un tempo di $\Theta(m+n)$
- Il ciclo while che permette di prelevare gli elementi idonei a far parte dell'insieme differenza risultante, impiega un tempo $\Theta(m+n)$ nel caso peggiore
- La funzione removeDuplicate(...) impiega un tempo $\Theta(m+n)$ per eseguire la rimozione dei duplicati da un array di $m+n$ elementi.
- L'ultimo for permette di inserire al più $m+n$ nodi all'interno di un RBT. Siccome l'inserimento in un RBT impiega un tempo $\Theta(log(n))$ e siccome vengono inseriti al più $m+n$ nodi, allora quest'ultimo passaggio impiega un tempo di

$$\Theta((m+n)(log(m+n))$$

A livello asintotico se consideriamo che $m+n$ può essere assimilato a n, allora si nota che in ogni caso la complessità risulta essere sempre dominata dall'ultimo passaggio di inserimento di n nodi. Quindi in totale la complessità di difference(...) a livello asintotico è di: $\Theta(n(log(n))$.

A questo punto consideriamo la complessità di spazio. Sia per union(...), che per intersect(...), che per difference(...), è necessario allocare tre array che conterranno rispettivamente m, n e $m+n$ elementi. Inoltre, per ognuno dei 3 metodi è necessario allocare lo spazio di un nuovo Set che nel caso peggiore conterrà $m+n$ Node. Quindi a livello asintotico la complessità di spazio risulta essere: $\Theta(m+n)$.

Test e risultati

Il programma permette all'utente di scegliere gli insiemi sui quali svolgere l'operazione insiemistica e l'operazione insiemistica da svolgere. Se le scelte dell'utente sono valide, si otterrà una corretta elaborazione dell'operazione.
Qualora sia possibile determinare l'insieme risultante dall'operazione, si procede con la stampa di esso e dei suoi elementi. Successivamente viene chiesto all'utente se vuole eseguire un'ulteriore operazione insiemistica. Il programma termina nel momento in cui l'utente esprime la propria volontà di non voler eseguire altre operazioni insiemistiche.
È presente un controllo sull'input inserito dall'utente sia sull'ID operazione che sceglie di fare, sia sulla scelta degli ID relativi agli insiemi, sia sulla scelta da lui effettuata quando gli viene chiesto di continuare o meno con l'esecuzione del programma.
Alcuni test effettuati sono riportati qui di seguito.

```
Benvenuti a 'Insiemi implementati con alberi Red  Black' ...

Gli insiemi presenti nel file caricato sono i seguenti. Inserisci l'ID di un primo
insieme, poi l'ID di un secondo insieme e infine l'operazione che vuoi eseguire
fra questi due insiemi.

Sets...

Set # 1:
4 8 22 24 32 35 45 49 52 57 61 66 74 75 76 85 92 96 98 99

Set # 2:
11 14 30 34 47 63 66 68 70 73 81 83

Set # 3:
3 4 15 17 20 22 42 47 49 52 58 73 74 78 80 86 94


Digita qui l'ID del primo insieme: 1

Digita qui l'ID del secondo insieme: 2

Scegli il codice dell'operazione insiemistica:
1) Unione
2) Intersezione
3) Differenza

Digita qui: 1

Il risultato della tua operazione insiemistica, è il seguente:

Set # 4:
4 8 11 14 22 24 30 32 34 35 45 47 49 52 57 61 63 66 68 70 73 74 75 76 81 83 85 92 96 98 99
```

```
Benvenuti a 'Insiemi implementati con alberi Red  Black' ...

Gli insiemi presenti nel file caricato sono i seguenti. Inserisci l'ID di un primo
insieme, poi l'ID di un secondo insieme e infine l'operazione che vuoi eseguire
fra questi due insiemi.

Sets...

Set # 1:
4 8 22 24 32 35 45 49 52 57 61 66 74 75 76 85 92 96 98 99

Set # 2:
11 14 30 34 47 63 66 68 70 73 81 83

Set # 3:
3 4 15 17 20 22 42 47 49 52 58 73 74 78 80 86 94

Set # 4:
4 8 11 14 22 24 30 32 34 35 45 47 49 52 57 61 63 66 68 70 73 74 75 76 81 83 85 92 96 98 99


Digita qui l'ID del primo insieme: 1

Digita qui l'ID del secondo insieme: 2

Scegli il codice dell'operazione insiemistica:
1) Unione
2) Intersezione
3) Differenza

Digita qui: 2

Il risultato della tua operazione insiemistica, è il seguente:

Set # 5:
66
Sets...

Set # 1:
4 8 22 24 32 35 45 49 52 57 61 66 74 75 76 85 92 96 98 99

Set # 2:
11 14 30 34 47 63 66 68 70 73 81 83

Set # 3:
3 4 15 17 20 22 42 47 49 52 58 73 74 78 80 86 94

Set # 4:
4 8 11 14 22 24 30 32 34 35 45 47 49 52 57 61 63 66 68 70 73 74 75 76 81 83 85 92 96 98 99

Set # 5:
66


Digita qui l'ID del primo insieme: 1

Digita qui l'ID del secondo insieme: 2

Scegli il codice dell'operazione insiemistica:
1) Unione
2) Intersezione
3) Differenza

Digita qui: 3

Il risultato della tua operazione insiemistica, è il seguente:

Set # 6:
4 8 22 24 32 35 45 49 52 57 61 74 75 76 85 92 96 98 99


Vuoi eseguire un'altra operazione insiemistica? (Y = YES / N = NO).
Scegli:
```

```
Benvenuti a 'Insiemi implementati con alberi Red  Black' ...

Gli insiemi presenti nel file caricato sono i seguenti. Inserisci l'ID di un primo
insieme, poi l'ID di un secondo insieme e infine l'operazione che vuoi eseguire
fra questi due insiemi.

Sets...

Set # 1:
4 8 22 24 32 35 45 49 52 57 61 66 74 75 76 85 92 96 98 99

Set # 2:
11 14 30 34 47 63 66 68 70 73 81 83

Set # 3:
3 4 15 17 20 22 42 47 49 52 58 73 74 78 80 86 94


Digita qui l'ID del primo insieme: 2

Digita qui l'ID del secondo insieme: 1

Scegli il codice dell'operazione insiemistica:
1) Unione
2) Intersezione
3) Differenza

Digita qui: 3

Il risultato della tua operazione insiemistica, è il seguente:

Set # 4:
11 14 30 34 47 63 68 70 73 81 83


Vuoi eseguire un'altra operazione insiemistica? (Y = YES / N = NO).
Scegli:
```