Traccia 1

Quesito 1

Descrizione del problema:

Il problema posto dal quesito 1 è quello di creare e gestire degli insiemi generici implementati sulle strutture dati degli alberi red-black, possiamo quindi suddividere il problema in due macro-problemi: l'implementazione della struttura dati albero red-black e il trattamento di questi come se fossero degli insiemi matematici.

In particolare, quando ci viene richiesto di basare gli insiemi su gli alberi red-black significa rappresentare questi tramite gli alberi e permettere di compiere le canoniche operazioni insiemistiche, ovvero:

- **Unione**: dati due insiemi *A* e *B* l'unione *(A U B)* consiste nel creare un terzo *U* insieme che contiene gli elementi di *A* e di *B* presi una sola volta.
- **Intersezione**: dati due insiemi *A* e *B* l'intersezione *(A ∩ B)* consiste nel creare un terzo insieme I che contiene gli elementi in comune tra i *A* e *B*.
- **Differenza**: dati due insiemi *A* e *B* la differenza di *A* con *B (A-B)* di consiste nel creare un terzo insieme *D* contenente gli elementi dell'insieme *A* meno quelli appartenenti a *B*, la differenza al contrario dell'unione e dell'intersezione non gode della proprietà commutativa, per cui $A\text{-}B \neq B\text{-}A$.

Come specificato nella traccia gli elementi su cui si andrà ad operare e quindi contenuti dagli insiemi saranno numeri interi.

Descrizione struttura dati:

La principale struttura dati utilizzata è l'albero red-black. Un albero red-black è un albero binario di ricerca, dato un nodo *x* dell'albero e dato un nodo *y* nel sottoalbero sinistro di *x*, abbiamo $key[y] \leq key[x]$, se invece *y* è un nodo nel sottoalbero destro di *x*, allora $key[y] \geq key[x]$.

Ciò che differenzia l'albero red-black da l'albero binario di ricerca è la presenza di un bit aggiuntivo di memoria per ogni nodo: il colore del nodo può essere rosso (red) o nero (black). Unendo questa caratteristica ad alcune limitazioni riguardanti le colorazioni dei nodi si può affermare che un albero binario di ricerca è un albero red-black se soddisfa le seguenti proprietà:

- Ogni nodo è rosso o nero
- La radice è nera
- Ogni foglia è nera
- Se un nodo è rosso, entrambi i suoi figli sono neri

- Per ogni nodo, tutti i percorsi che vanno dal nodo alle foglie discendenti contengono lo stesso numero di nodi neri, ovvero l'altezza nera (b-altezza).

La caratteristica dell'albero red-black è di essere pressoché bilanciato come conseguenza alla proprietà della b-altezza, in particolare il ramo dell'albero red-black più lungo avrà altezza al massimo due volte maggiore rispetto a quella ramo più corto.

Per risparmiare spazio invece di avere come nodi foglia tanti nodi vuoti *NIL*, vengono fatti puntare i nodi che dovrebbero puntare alle foglie, ad un unico nodo sentinella *nil*.

L'altezza massima di un albero red-black con *n* nodi interni è: *2*log(n + 1)*. Questo implica che a differenza di un albero binario di ricerca le operazioni di *search, minimum, maximum, successor, predecessor, insert e delete* impiegano un tempo $\theta(log\ n)$ dove *n* è il numero dei nodi. La risoluzione del problema non necessita di modifiche né alla struttura "classica" dell'albero red-black.

Formato dati in input/output:

Gli input sono numeri interi forniti al programma tramite lettura da file. All'interno del file gli elementi appartenenti ad uno stesso insieme, si trovano sulla stessa riga separati da uno spazio, le righe successive corrispondono ai diversi insiemi.

L'utente interagisce attraverso un menù che gli permette di scegliere quali operazioni svolgere con gli insiemi di cui dispone. Per poter usare il menù l'utente fornisce da tastiera come input il valore di alcune variabili, a seconda del quale viene eseguita un'operazione.

L'output consiste degli insiemi letti dal file e da quelli che vengono a crearsi in seguito alle operazioni di unione, intersezione e sottrazione che saranno memorizzati insieme a quelli di partenza del file.

Tutti gli insiemi verranno visualizzati tramite una stampa a video.

Viene stampato a video anche il menù il quale dà la possibilità all'utente di compiere le operazioni sopra citate sugli insiemi, di visualizzare gli insiemi memorizzati fino a quel momento e di uscire dal programma.

Descrizione algoritmo:

L'obbiettivo dell'algoritmo è quello di eseguire le operazioni insiemistiche di unione, intersezione e differenza su gli alberi red-black che vanno a rappresentare gli insiemi.

Intersezione

Alla base dell'idea dell'algoritmo dell'intersezione vi è una visita in ordine simmetrico su entrambi gli alberi contemporaneamente mediante due stack, che vanno a "simulare" le chiamate ricorsive che avvengono all'interno della visita "classica" ricorsiva in ordine simmetrico.

Questo tipo di visita permette di visitare gli elementi di un albero red-black in ordine crescente di chiave.

Applicandola in modo iterativo è possibile confrontare direttamente, ad ogni passo, il nodo visitato nel primo albero e nel secondo albero, riuscendo a determinare se il nodo in questione appartenga o meno all'insieme intersezione.

Dato *x* appartenente al primo insieme e *y* appartenente al secondo insieme visitati al passo *i-esimo*, ipotizzando che $key[x] < key[y]$, la visita del secondo insieme riprenderà solo quando verrà visitato un nodo x^1 del primo insieme tale che $key[x^1] \geq key[y]$, oppure quando non ci saranno più elementi da visitare nel primo insieme. Questo farà sì che terminata la visita di un elemento, la chiave del successivo sarà maggiore di quello appena visitato.

Partendo dalla radice si scende nel figlio sinistro finché non si arriva al nodo sentinella *nil*, inserendo i nodi che si visitano man mano in uno stack. Facendo ciò in ambedue gli alberi considerati si effettua la visita completa del ramo sinistro più esterno di ogni albero. Si confronta il nodo inserito per ultimo nei due stack, ovvero quello in cima, se la chiave dei nodi è uguale viene inserito solo uno dei due nodi, per non creare duplicati, in un nuovo albero red-black il quale andrà a rappresentare l'insieme intersezione, in questo modo viene inserito nell'insieme intersezione un elemento che è presente in entrambi gli insiemi di partenza, proprietà che indica un elemento fa parte dell'insieme intersezione. Si procede scendendo nel figlio destro sia del nodo del primo stack che del secondo stack e da questi ripartirà la visita dei rispettivi rami nei corrispettivi insiemi di partenza rappresentati da due alberi red-black. Nel caso in cui uno dei due nodi estratti dalla cima dello stack sia minore, si scenderà nel suo figlio destro inserendolo nel relativo stack e avendo come conseguenza la visita di un nuovo ramo in uno solo dei due insiemi di partenza.

Ovviamente verrà fatta l'operazione simmetrica per quanto riguarda il caso opposto.

L'algoritmo termina la sua esecuzione quando i due stack saranno vuoti e non vi saranno più elementi da poter inserire al loro interno, il che significa che sono stati visitati tutti i rami di entrambi gli alberi ed analizzati i rispettivi nodi.

L'albero red-black risultante rappresenterà l'insieme intersezione dei due insiemi di partenza.

Nello pseudo-codice rappresentante l'algoritmo gli elementi vengono inseriti prima all'interno di un array in modo da eliminare i duplicati derivanti da possibili elementi uguali presenti nello stesso insieme.

La funzione *deleteDuplicate* che si occupa di svolgere questa operazione verrà descritta durante l'unione.

Di seguito viene riportato lo pseudo-codice rappresentante l'algoritmo appena descritto con l'analisi delle rispettive righe.

```
1.  intersection(tree1, tree2)
2.  s1 ← ∅                          // stack che visiterà il primo albero
3.  s2 ← ∅                          // stack che visiterà il secondo albero
4.  Intersect ← ∅                   //Viene dichiarato ed inizializzato l'array che costituirà l'intersezione
5.  node1 ← root[tree1]
6.  node2 ← root[tree2]
7.  while ((s1 ≠ ∅ and s2 ≠ ∅) or (node1 ≠ nil[tree1] or node2 ≠ nil[tree2]))
8.    if(node1 ≠ nil[tree1])
9.                  push(s1, node1)
10.                 node1 ← left[node1]
11.   else if (node2 ≠ nil[tree2])
12.       push(s2, node2)
13.       node2 ← left[node2]
14.   else
15.       top(s1, node1)
16.                 top(s2, node2)
17.       if (key[node1] = key[node2])
18.           insert(Intersect, key[node1])      //Inserisci elementi nell'insieme intersezione
19.           pop(s1)
20.           pop(s2)
21.           node1 ← right[node1]
22.           node2 ← right[node2]
23.       else if (key[node1] < key[node2])
24.         pop(s1)
25.         node1 ← right[node1]
26.         node2 ← nil[tree2]
27.       else
28.        pop(s2)
29.         node2 ← right[node2]
30.         Node1 ← nil[tree1]
31. TreeIntersect ← ∅
32. TreeIntersect ← deleteDuplicate(Interserct) //Inserisce gli elementi meno duplicati nel'albero
                                                  intersezione
33. return TreeIntersect
```

La funzione *intersection* riceve in input i due alberi red-black *tree1 e tree2* i quali vanno a rappresentare i due insiemi sui cui verrà compiuta l'intersezione, l'output di questa funzione corrisponde all'albero red-black contenente l'intersezione di *tree1 e tree2*.

Dalla riga 2 alla riga 6 vengono inizializzati i due stack *s1* e *s2* che andranno a simulare le chiamate ricorsive che si avverrebbero all'interno dello stesso tipo di visita ma ricorsiva e che salvano gli elementi visitati ma non ancora analizzati, sono dichiarate due variabili *node1* e *node2* alle quali viene assegnato rispettivamente il valore della radice di *tree1* e *tree2*.

Si dichiara un array *intersect* che sarà costituito dagli elementi dell'insieme intersezione.

Nella riga 7 si entra all'interno del ciclo *while* dal quale si uscirà una volta che almeno uno dei due stack sarà vuoto ed il valore di entrambe le variabili *node* sia uguale a *nil*. L'uscita dal ciclo *while* implica che almeno uno dei due alberi è stato interamente visitato e sono stati controllati tutti i possibili nodi e determinato quelli facente parte dell'intersezione.

Dalla riga 8 alla riga 13 si va a scendere il più possibile verso sinistra nel ramo del nodo salvato nella variabile *node1 e node2*, inserendo i nodi che man mano vengono visitati nel rispettivo stack, fin quando che non si arriva al nodo sentinella *nil* il quale indica la fine del ramo considerato. La visita del ramo avviene prima per un albero e poi per l'altro, cosa rispettivamente determinata se si entra nel blocco dell'*if* o dell'*else if*. Non si entrerà nel costrutto dell'*else* fin quando entrambi le variabili *node* non saranno uguali al nodo sentinella *nil* e indicando di aver visitato interamente il ramo preso in esame.

Dalla riga 14 alla riga 30 viene assegnato al parametro *key* di *node1* e *node2* il valore dell'ultimo elemento aggiunto nei due stack *s1* e *s2*, ovvero il più piccolo degli elementi visitati nei rispettivi alberi fino a quel momento, ma anche l'elemento con il valore minore presente nello stack.

Se *key[node1] = key[node2]* significa che l'elemento è presente sia in *tree1* che in *tree2,* di conseguenza fanno entrambi parte dell'insieme intersezione, per cui viene inserito solo *key[node1]* in *intersect* in modo da evitare duplicati. A questo punto vengono decrementati gli stack in modo da non ritrovare in seguito gli elementi appena analizzati e viene assegnato a *node1* e *node2* il valore del loro figlio destro, cosa che permetterà una volta ricominciato il ciclo di visitare anche gli altri rami che non sono stati considerati nelle precedenti visite.

Nel caso in cui *key[node1] < key[node2]* significa che il valore *key[node1]* non fa parte dell'intersezione, viene quindi decrementato *s1* ed assegnato a *node1* il valore del suo figlio destro e a *node2* il nodo sentinella *nil[tree2]* in modo da continuare, ricominciando il ciclo, la visita dell'albero sui rami non ancora controllati solamente nell'albero *tree1*. Avviene l'operazione simmetrica nel caso di *key[node1] > key[node2]*. In questo modo è garantita una visita completa dell'albero in ordine simmetrico grazie al quale vengono determinati quali elementi facciano parte dell'intersezione per quanto detto nel preambolo allo pseudocodice.

Nella riga dalla 31 alla 33 viene richiamata la funzione *deleteDuplicate* (che verrà spiegata nell'unione) sull'array *Intersect,* in modo da eliminare possibili duplicati derivanti da elementi uguali nello stesso insieme. Gli elementi vengono inseriti in un nuovo albero *TreeIntersect* il quale viene ritornato dalla funzione.

Unione

L'algoritmo dell'unione si basa su tre funzioni: *inorder, merge e deleteDuplicate*.

La funzione *inorder* visita banalmente in ordine simmetrico prima un albero e poi l'altro.

Durante la visita i nodi dei due alberi vengono memorizzati in due array differenti *vett1* e *vett2*.

Gli elementi all'interno di dei due array saranno posizionati in ordine crescente grazie alla proprietà della visita in ordine simmetrico riguardo gli alberi red-black.

Quando viene effettuata la visita in ordine simmetrico su un albero binario di ricerca, quindi anche su un albero red-black, gli elementi sono visualizzati in ordine crescente di parametro *key*.

I due array *vett1* e *vett2* diventano l'input della funzione *merge.*

La funzione *merge* prende in input due array ordinati e restituisce un terzo array ordinato, questo array è la fusione dei due presi in input. Il meccanismo della fusione consiste al passo *i-esimo* nel controllare l'elemento *a[i]* del primo array e *b[j]* del secondo array con i *e* $j \in \mathbb{N}$.

Se $a[i] \leq b[j]$ allora inseriremo nell'array della fusione l'elemento *a[i]* ed incrementeremo l'indice *i*, altrimenti compiremo l'operazione simmetrica. Visto che i due array di partenza sono ordinati, scegliendo sempre l'elemento più piccolo dei due ad ogni passo, l'array derivante dalla fusione sarà ordinato.

L'array uscente dalla funzione *merge* conterrà tutti gli elementi del primo array e del secondo array compresi i duplicati.

Per epurare l'array dai duplicati ed ottenere così l'unione dei due insiemi di partenza viene richiamata la funzione *deleteDuplicate*.

Questa funzione si basa sulla caratteristica dell'array di essere ordinato, in questo modo elementi uguali si trovano in posizioni consecutive. Basterà quindi scorrere l'array e controllare al passo *i-esimo* se alla posizione *c[i] e c[i+1]* dell'array sono presenti elementi uguali.

Nel caso in cui i due elementi siano diversi vengono inseriti entrambi all'interno di un nuovo albero red-black rappresentante un nuovo insieme, altrimenti viene inserito solo l'elemento alla posizione *c[i]* .

In questo modo il nuovo insieme rappresenterà l'unione dei due insiemi di partenza.

```
1.  Union(tree1, tree2)
2.  left ← ∅              //primo array contenente gli elementi ordinati del primo insieme
3.  right ← ∅             //primo array contenente gli elementi ordinati del primo insieme
4.  inorder(root[tree1], left)
5.  inorder(root[tree2], right)
6.  newVet ← ∅            //array contenente la fusione dei due array left e right
7.  newVet ← merge(left, right)
8.  newSet ← ∅            //nuovo albero red-black rappresentante l'insieme unione
9.  newSet ← deleteDuplicate(merge)
10. return newSet

11. inorder(x, vett)
12. if (x != Nil)
13.     inorder(left[x], vett)
14.     Insert(vett, x)
15.     Inorder(right[x], vett)

16. merge(left, right)
17. i ← 1
18. j ← 1
19. k← 1
20. vett ← ∅
21. while i ≤ size[left] and j ≤ size[right]
22.         if left[i] ≤ right[j]
23.             vett[k] ← left[i]
24.             i = i+1
25.             k = k+1
26.         else
27.             vett[k] ← right[j]
28.             j = j+1
29.             k = k+1
30. while i ≤ size[left]
31.         vett[k] ←left[i]
32.         i++
```

```
33.     k++
34. while i ≤ size[left]
35.     vett[k] ←left[i]
36.     i++
37.     k++
38. return vett


39. deleteDuplicate(vett)
40. newSet ← ∅
41. for i ← 1 to size[vett]
42.   if vett[i] ≠ vett[i+1]
43.     Insert(newSet, vett[i])
44. return newSet
```

Dalla riga 1 alla riga 10 viene illustrata la funzione *unionSet* la quale richiama le funzioni *inorder*, *merge* e *deleteDuplicate*, istanzia inoltre i due array *left* e *right* che poi verranno fusi in *newVet* e il nuovo insieme *newSet*, questo nuovo insieme è rappresentato da un albero red-black, questo conterrà gli elementi dell'insieme unione dei due alberi tree1 e tree2 che rappresentano gli insiemi in input.

Dalla riga 11 alla 15 viene illustrata la funzione *inorder* la quale effettua una visita in ordine simmetrico, i nodi invece di essere stampati vengono inseriti all'interno dell'array *vett*.

Questo array per la proprietà della visita in ordine simmetrico sarà ordinato.

Questa visita verrà effettuata sia su tree1 che su tree2, i nodi di questi vengono memorizzati in due array *left* e *right*.

Gli array *left* e *right* diventano l'input della funzione merge rappresentata dallo pseudo-codice tra le righe 16 e 38.

Nella funzione merge viene dichiarato un terzo array *vett* che conterrà gli elementi dell'unione.

Gli indici i, j e k serviranno a scorrere gli array *left*, *right* e *vett*.

Successivamente si entra nel ciclo while dal quale si uscirà una volta che uno dei due array in input è stato completamente visitato.

Se *left[i]* ≤ *right[j]*, viene inserito *left[i]* in posizione *k* di *vett* e si incrementano *i* e *k*, in caso contrario viene effettuata l'operazione simmetrica.

Terminati gli elementi di un array la visita continua esclusivamente su quello contenente ancora elementi, tramite uno dei due while alla fine della funzione. In questo modo l'array *vett* di ritorno dalla funzione merge sarà ordinato in maniera crescente.

L'array derivante dalla fusione viene assegnato nella funzione *unionSet* alla variabile *newSet*. Questo array diventa l'input della funzione *deleteDuplicate* la quale viene illustrata dalla riga 39 alla riga 44.

Questa funzione prende in input un array ordinato come *newSet*, essendo quindi ordinato gli elementi uguali si trovano in posizioni consecutive. A questo punto non resta che scorrere l'array e controllare se al passo *i-esimo* l'elemento *c[i]* e *c[i+1]* dell'array sono uguali.

In tal caso verrà inserito solo l'elemento *c[i]* all'interno di un nuovo insieme rappresentato dal codice da *newSet*, altrimenti ad essere inseriti in *newSet* saranno sia *c[i]* e *c[i+1]*.

La funzione *deleteDuplicate* ritornerà quindi l'albero red-black rappresentante l'insieme unione tra i due insiemi di partenza tree1 e tree2.

Differenza

Prima di spiegare l'algoritmo alla base della differenza bisogna ricordare che tale operazione, a differenza dell'unione e dell'intersezione, non gode della proprietà commutativa.

L'algoritmo della differenza si basa su alcune delle funzioni illustrate precedentemente: *inorder*, *merge*, *intersect* e una variante della *deleteDuplicate*.

La funzione *inorder* viene applicata sul primo insieme in modo da ottenere un array ordinato in maniera crescente.

Viene richiamata la funzione *intersect* la quale restituisce l'intersezione fra i due insiemi di partenza , questa però non viene inserita nell'albero red-black ma all'interno di un array, a questa funzione *intersect* non viene applicata la *deleteDuplicate*. Questo perché se al primo insieme viene sottratta l'intersezione con il secondo insieme si ottiene la differenza.

Viene richiamata quindi la funzione merge avente come argomento in input sia l'array derivante da *inorder* che da *intersect* entrambi ordinati.

Sull'array uscente dalla funzione merge viene quindi applicata la *deleteInt* (variante della *deleteDuplicate*) la quale epura questo array dall'intersezione e quindi ottenendo la differenza.

```
1.  difference(tree1, tree2)
2.  left ← ∅              //primo array contenente gli elementi ordinati del primo insieme
3.  right ← ∅             //primo array contenente gli elementi ordinati del primo insieme
4.  inorder(root[tree1], left)
5.  right ← intersect(tree1, tree2)
6.  newVet ← ∅            //array contenente la fusione dei due array left e right
7.  newVet ← merge(left, right)
8.  newSet ← ∅            //nuovo albero red-black rappresentante l'insieme differenza
9.  newSet ← deleteInt(merge)
10. return newSet

11. intersection(tree1, tree2)
12. s1 ← ∅                  // stack che visiterà il primo albero
13. s2 ← ∅                  // stack che visiterà il secondo albero
14. Intersect ← ∅           //Viene dichiarato ed inizializzato l'array che costituirà
    l'intersezione
15. node1 ← root[tree1]
16. node2 ← root[tree2]
17. while ((s1 ≠ ∅ and s2 ≠ ∅) or (node1 ≠ nil[tree1] or node2 ≠ nil[tree2]))
18.   if(node1 ≠ nil[tree1])
19.       push(s1, node1)
20.       node1 ← left[node1]
21.   else if (node2 ≠ nil[tree2])
22.       push(s2, node2)
23.       node2 ← left[node2]
24.   else
25.       top(s1, node1)
26.       top(s2, node2)
27.       if (key[node1] = key[node2])
28.           insert(Intersect, key[node1])     //Inserisci elementi nell'insieme intersezione
29.           pop(s1)
30.           pop(s2)
31.           node1 ← right[node1]
32.           node2 ← right[node2]
33.       else if (key[node1] < key[node2])
34.           pop(s1)
```

```
35.            node1 ← right[node1]
36.            node2 ← nil[tree2]
37.        else
38.            pop(s2)
39.            node2 ← right[node2]
40.            Node1 ← nil[tree1]
41.  return Intersect

42. inorder(x, vett)
43. if (x != Nil)
44.     inorder(left[x], vett)
45.     Insert(vett, x)
46.     Inorder(right[x], vett)

47. merge(left, right)
48. i ← 1
49. j ← 1
50. k← 1
51.  vett ← ∅
52. while i ≤ size[left] and j ≤ size[right]
53.         if left[i] ≤ right[j]
54.             vett[k] ← left[i]
55.             i = i+1
56.             k = k+1
57.         else
58.             vett[k] ← right[j]
59.             j = j+1
60.             k = k+1
61.  while i ≤ size[left]
62.         vett[k] ←left[i]
63.         i++
64.         k++
65. while i ≤ size[left]
66.         vett[k] ←left[i]
67.         i++
68.         k++
```

```
69. return vett


70. deleteInt(vett)
71. newSet ← ∅
72. for i ← 1 to size[vett]
73.     if vett[i] ≠ vett[i+1] and vett[i] ≠ vett[i-1]
74.         Insert(newSet, vett[i])
75. return newSet
```

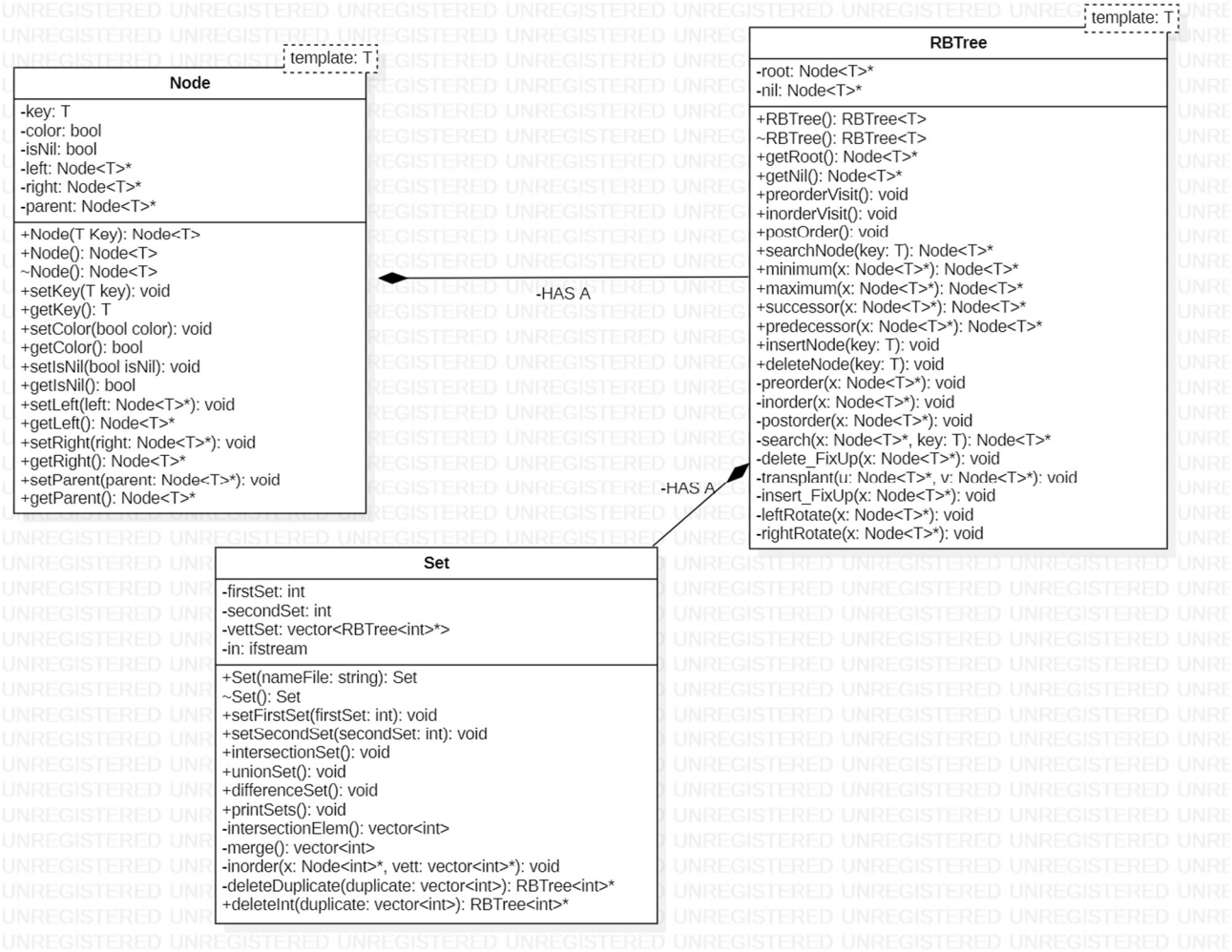
Tra le righe 1 e 10 viene illustrato lo pseudo-codice della funzione *difference*, questa prende in input i due insiemi rappresentati da due alberi red-black *tree1 e tree2*. All'interno di questa funzione vengono richiamate le stesse funzioni già spiegate precedentemente, fatta eccezione per alcuni per particolari della *intersection*.

Questa si differenza dalla precedente in quanto gli elementi non vengono inseriti in un array e non viene richiamata la *deleteDuplicate,* come illustrato anche dalle righe comprese tra la 11 e la 41.

La funzione *deleteInt*, illustrata tra le righe 70 e 75 invece risulta essere una variante della *deleteDuplicate.*

Un elemento in posizione i dell'array viene inserito nell'insieme differenza se in posizione *i-1* e *i+1* non sono presenti gli elementi. In tal modo l'insieme che viene ritornato alla riga 10 della funzione *difference* conterrà gli elementi appartenenti all'insieme differenza di tree1 *e* tree2.

Diagram class:



Studio complessità:

Sebbene esistano delle funzioni conosciute che riescano ad ottenere lo stesso risultato ma con complessità di tempo minore, queste richiedono delle assunzioni precise fatte sugli input le quali non possono essere applicate per il problema preso in considerazione.

Intersezione

La funzione *intersection* si basa su una visita di due alberi red-black in contemporanea composti rispettivamente da m ed n nodi, il che comporta una complessità $\theta(m+n)$.

All'interno di questa visita in base a determinate condizioni vengono inseriti dei nodi all'interno di un nuovo albero red-black, operazione che richiede una complessità $\theta(log(n))$ dove n è il numero di nodi presenti all'interno del nuovo albero.

Possiamo quindi affermare l'esistenza di un caso migliore e di un caso peggiore.

Nel caso in cui l'intersezione sia uguale a ∅ (ovvero quando tutti gli elementi del primo insieme sono diversi da quelli del secondo) possiamo affermare che la complessità è data da $\theta(m+n)$ ovvero solamente dalla visita, poiché non viene effettuato nessun inserimento.

Il caso peggiore si verifica quando i due insiemi sono uguali, in tal caso viene effettuata la visita su due alberi con lo stesso numero di nodi *n* impiegando un tempo $\theta(2n)$, nella *deleteDuplicate* si scorre un array contenente *n* nodi effettuando *n* inserimenti nel nuovo insieme, per una complessità di $\theta((n) * log(n))$, per cui la complessità del caso peggiore è di $\theta((n) * log(n))$.

Possiamo quindi affermare che a livello asintotico l' *intersection* ha un costo $\theta((n) * log(n))$ nel caso peggiore ed $\theta(m+n)$ nel caso migliore.

Per quanto riguarda la complessità di spazio questa è determinata dal numero di nodi massimi contenuti all'interno dei due stack, che corrisponde a $\theta(h1 + h2)$ ovvero l'altezza del primo albero più quella del secondo, al quale deve essere aggiunto un $\theta(k)$ corrispondente al numero di nodi inseriti nel nuovo albero dove $0 \leq k \leq m+n$.

Unione

La funzione *union* richiama due volte la funzione *inorder*, la funzione *merge* e la *deleteDuplicate*.

La funzione *inorder* essendo una visita dell'albero impiega un tempo $\theta(n)$ dove *n* è il numero dei nodi del primo albero.

Il costo delle due *inorder* impiega quindi $\theta(n)$ e $\theta(m)$ dove *n* è il numero di nodi del primo albero ed *m* quello del secondo.

La funzione *merge* impiega un tempo $\theta(n+m)$ poiché deve essere visitato interamente l'array contenente *n* nodi l'altro array contenente *m* nodi.

La funzione *deleteDuplicate* scorre un array contenente *m+n* nodi, mentre l'array viene visitato vengono effettuati degli inserimenti nell'albero red-black che impiegano un costo $\theta(log(n))$, dove *n* è il numero degli elementi nell'albero. Il costo totale della funzione è $\theta(log\ (n+m) * (n+m))$.

In conclusione, analizzando tutte le complessità delle funzioni richiamate da *union,* si ha che a livello asintotico la funzione impiega un tempo $\theta(log\ (n+m) * (n+m))$.

La complessità di spazio è data dal numero di elementi del primo insieme e del secondo insieme ovvero $\theta(n+m)$.

Differenza

La funzione richiama due volte la funzione *inorder*, la funzione *merge*, *intersection* e la *deleteInt*.

La funzione *inorder* essendo una visita dell'albero impiega un tempo $\theta(n)$ dove *n* è il numero dei nodi del primo albero.

La funzione *intersection* per quanto detto prima ha un costo $\theta(n+m)$ nel caso migliore, nel caso peggiore $\theta((n) * log(n))$.

La funzione *merge* impiega un tempo $\theta(n+m)$ poiché deve essere visitato interamente l'array contenente *n* nodi l'altro array contenente *m* nodi.

La funzione *deleteInt* scorre un array contenente *m+n* nodi provenienti da intersection ed inorder nel caso peggiore ed *n* nel caso migliore, durante lo scorrimento effettua degli inserimenti in un albero red-black, per cui il costo di questa funzione è $\theta((n+m) * log(n+m))$ nel caso peggiore ed $\theta((n) * log(n))$ nel caso migliore.

In conclusione, analizzando tutte le complessità appena ottenute si ha che la funzione *union* a livello asintotico impiega un tempo $\theta((n+m) * log(n+m))$ nel caso peggiore ed $\theta((n) * log(n))$ nel caso migliore.

La complessità di spazio è data $\theta(n+m)$ dove *m* e il numero di elementi del primo insieme e *n* del secondo.

Test e risultati:

Il programma stamperà tutti gli insiemi con i rispettivi elementi ottenuti tramite la lettura del file in input.

Successivamente un menù informerà l'utente sulle possibili operazioni a sua disposizione, le quali vengono selezionate dal valore indicato dal menù che si vuole digitare.

In particolare, all'utente si presentano 5 possibili scelte alle quali sono associati i corrispettivi comandi da digitare:

- Digitando 1 sarà possibile compiere l'operazione di unione
- Digitando 2 sarà possibile compiere l'operazione di intersezione
- Digitando 3 sarà possibile compiere l'operazione di differenza
- Digitando 4 sarà possibile stampare gli insiemi presenti memorizzati fino a quel momento
- Digitando 0 si uscirà dal programma.

Nel caso in cui l'utente digiti un comando non consentito dal menù verrà riproposto nuovamente quest'ultimo.

Dopo che l'utente avrà effettuato la scelta dell'operazione, gli sarà richiesto di indicare i due insiemi sul quale vuole operare digitando un numero per indentificare ciascuno di essi, di fatto nella stampa ad ogni insieme viene affiancato relativo numero.

Se viene inserito un numero non consentito il programma restituisce un messaggio diagnostico di errore, altrimenti l'operazione avviene con successo.

Una volta effettuata l'operazione viene stampato l'insieme derivante da quest'ultima e viene chiesto poi all'utente di digitare *0* nel caso voglia uscire dal programma e qualsiasi altro carattere per continuare a eseguire ancora il programma.

Per quanto riguarda i casi testati, il programma è stato eseguito sia nei casi "normali" quindi insiemi variegati fra loro con risposte positive, sia nei casi "limite".

Per esempio, nel caso dell'intersezione e dell'unione di un insieme con se stesso o con un insieme contenente elementi uguali ma in minor numero viene prodotto un insieme con un unico elemento.

Tra gli altri casi limite è si può segnalare anche il caso in cui venga prodotto un insieme vuoto, come la differenza di un insieme con se stesso.

Si noti che al gruppo degli insiemi in questo caso viene aggiunto un nuovo albero red-black contenente unicamente il nodo sentinella *nil*.

Di seguito vengono mostrarti alcuni test che confermano la validità di ciò che è stato descritto.

Stampa menù e insiemi

```
6 sets were produced from the file

Set number: 1

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

Set number: 2

11 14 30 34 47 63 66 68 70 73 81 83

Set number: 3

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

Set number: 4

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Set number: 5

1

Set number: 6

2 2 2 2 2

Choose an action:
Type 1 for union
Type 2 for intersection
Type 3 for subtraction
Type 4 to show the sets
Type 0 to exit
1
Choose the sets number you want to operate on:
1
```

Casi normali

```
Choose an action:
Type 1 for union
Type 2 for intersection
Type 3 for subtraction
Type 4 to show the sets
Type 0 to exit
1
Choose the sets number you want to operate on:
1
2
union:
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 99
New set was add at number 7
Type 0 to exit otherwise type any other simbol to continue 2


Choose an action:
Type 1 for union
Type 2 for intersection
Type 3 for subtraction
Type 4 to show the sets
Type 0 to exit
2
Choose the sets number you want to operate on:
1
2
intersection:
66
New set was add at number 8
Type 0 to exit otherwise type any other simbol to continue 3


Choose an action:
Type 1 for union
Type 2 for intersection
Type 3 for subtraction
```

```
Choose an action:
Type 1 for union
Type 2 for intersection
Type 3 for subtraction
Type 4 to show the sets
Type 0 to exit
3
Choose the sets number you want to operate on:
1
2
difference:
4 8 22 24 32 35 45 49 52 57 61 74 75 76 85 92 96 98 99
New set was add at number 9
Type 0 to exit otherwise type any other simbol to continue 3


Choose an action:
Type 1 for union
Type 2 for intersection
Type 3 for subtraction
Type 4 to show the sets
Type 0 to exit
3
Choose the sets number you want to operate on:
2
1
difference:
11 14 30 34 47 63 68 70 73 81 83
New set was add at number 10
Type 0 to exit otherwise type any other simbol to continue 3


Choose an action:
Type 1 for union
Type 2 for intersection
```

Casi limite

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

  Set number: 2

  11 14 30 34 47 63 66 68 70 73 81 83

  Set number: 3

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

  Set number: 4

  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

  Set number: 5

  1

  Set number: 6

  2 2 2 2 2

  Choose an action:
  Type 1 for union
  Type 2 for intersection
  Type 3 for subtraction
  Type 4 to show the sets
  Type 0 to exit
  1
  Choose the sets number you want to operate on:
  4
  5
  union:
  1
  New set was add at number 7
  Type 0 to exit otherwise type any other simbol to continue
```

```
Set number: 4

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Set number: 5

1

Set number: 6

2 2 2 2 2

Choose an action:
Type 1 for union
Type 2 for intersection
Type 3 for subtraction
Type 4 to show the sets
Type 0 to exit
2
Choose the sets number you want to operate on:
5
6
intersection:

New set was add at number 7
Type 0 to exit otherwise type any other simbol to continue
```