Corso di Laurea
Informatica e Tecnologie per la Produzione del Software

Corso di Algoritmi e Strutture Dati + Laboratorio
Docente: Ing. Stefano Ferilli

STUDIO DI UN CASO (CASE STUDY):

Problema:

- Acquisire in un albero binario una formula logica (rappresentata come stringa) contenente delle stringhe combinate solo tramite gli operatori AND, OR e NOT, comunque nidificati, e comprensiva di tutte le parentesi (non e' specificata alcuna priorità implicita).
- Stampare l'albero e consentire all'utente di applicare (anche più volte) ad un nodo a scelta una delle seguenti trasformazioni:
  • leggi di De Morgan
  • distributività
- Ritrasformare, su richiesta dell'utente, la formula rappresentata dall'albero binario in una stringa corrispondente.

Ulteriore funzionalità (facoltativa):

- Trasformare la formula in forma normale congiuntiva, secondo la seguente strategia:
- Ridurre il raggio di azione del NOT affinché si applichi al più ad una formula atomica, sfruttando le leggi di De Morgan e quella della doppia negazione:
  - NOT (NOT p) = p
  - NOT (p AND q) = NOT p OR NOT q
  - NOT (p OR q) = NOT p AND NOT q
- Mettere la formula in forma normale congiuntiva (disgiunzione di congiunzioni di atomi negati oppure no), sfruttando la proprietà distributiva:
  - p OR (q AND r) = (p OR q) AND (p OR r)
- Rimpiazzare le espressioni ottenute come segue:
  - (p11 AND ... AND P1n) OR ... OR (Pm1 AND ... AND Pmnm) diventa
  - [p11, ..., P1n]
  - ...
  - [Pm1, ..., Pmnm]

Sommario:

Svolto da:
Maiellaro Andrea
Domenico Pio Novelli

1. FASE DI ANALISI

Data un'espressione booleana, di lunghezza limitata, si vuole permette all'utente di scegliere una delle operazioni presenti nell'espressione e applicare a scelta uno dei seguenti punti:

- Leggi di De Morgan
- Proprietà distributiva degli operatori booleani (AND e OR)

L'espressione deve essere del tipo: *(espressione AND / OR espressione),*
dove l'espressione può includere anche il NOT

Nell'espressione i tre operatori logici: AND, OR e NOT, dovranno essere inseriti nel seguente modo:

- & -> AND
- | -> OR
- ! -> NOT

Esempio di espressione: "((Gino | !Pino) & (!(Mino & Rino) | Dino))"

L'albero che contiene l'espressione ricevuta in input dovrà essere visualizzata per consentire all'utente di scegliere il nodo su cui apportare la modifica.
Per applicare le trasformazioni, sono possibili due modalità:

- Apportare una modifica, visualizzare su monitor la nuova espressione booleana corrispondente alla prima e ripresentare la scelta all'utente di una ulteriore modifica.
- Poter applicare più trasformazioni ad un nodo consecutivamente e solo al termine mostrare su monitor la nuova espressione booleana

Inoltre si presuppone che l'espressione booleana sia dotata di parentesi tonde per indicare la precedenza delle singole operazioni logiche che compongono l'espressione stessa.

Legge della Doppia Negazione:

- NOT (NOT a) = a

Leggi di De Morgan:

- Prima: NOT (a AND b) = NOT a OR NOT b
- Seconda: NOT (a OR b) = NOT a AND NOT b

Le leggi di De Morgan possono essere applicati a tutte le espressioni.
Il caso in cui l'operatore NOT si trovi prima di una parentesi, applicata ad un fattore composto da una operazione biunivoca, verrà tradotto in una espressione, che segue la sintassi: FATTORE - NOT OPERATORE - FATTORE; in questo modo l'applicazione di una delle Leggi di De Morgan porterà al seguente risultato:

1) NOT(a AND b) -> NOT(AND) a b ⇔ NOT a OR NOT b
2) NOT (a OR b)-> NOT(OR) a b ⇔ NOT a AND NOT b

All'espressioni con Operatore AND applichiamo solo la (2.<=)
All'espressioni con Operatore OR applichiamo solo la (1.<=)
All'espressioni con Operatore NOT-AND applichiamo solo la (1.=>)
All'espressioni con Operatore NOT-OR applichiamo solo la (2.=>)

Proprietà Distributive:
Le proprietà distributive possono essere applicate all'espressioni del tipo:

(Distributività dell' AND rispetto all' OR)
1) a AND (b OR c) => (a AND b) OR (a AND c)
2) (b OR c) AND a => (b AND a) OR (c AND a)

(Distributività dell'OR rispetto all'AND)
3) a OR (b AND c) => (a OR b) AND (a OR c)
4) (b AND c) a OR => (b OR a) AND (c OR a)

La distributive può essere applicata anche inversamente
All'espressioni con Operatore OR applichiamo solo la 1 o la 2
1) a AND (b OR c) <= (a AND b) OR (a AND c)
2) (b OR c) AND a <= (b AND a) OR (c AND a)

All'espressioni con Operatore OR applichiamo solo la 3 o la 4
Dell'OR rispetto all'AND:
3) a OR (b AND c) <= (a OR b) AND (a OR c)
4) (b AND c) a OR <= (b OR a) AND (c OR a)

I dati in Input che la traccia offre sono:
- Stringa dell'espressione booleana
- Scelta del nodo da modificare

I dati in Output da fornire sono:
- Albero Binario dell'espressione booleana
- Stringa modificata (a video)

Errori che possono avvenire in fase di input sono:
- Inserimento di una Stringa Errata (vuota o non corretta)
- Scelta di applicare una modifica su un nodo errato
- Eccezioni di Input

2. ALGEBRA DELLA STRUTTURA
2.1 Albero Binario

LA SPECIFICA SINTATTICA
TIPI: ALBEROBIN, BOOLEANO, NODO, TIPOELEM

- CREABINALBERO : ( ) -> ALBEROBIN
- BINALBEROVUOTO : (ALBEROBIN) -> BOOLEANO
- BINRADICE : (ALBEROBIN) -> NODO
- BINPADRE : (NODO,ALBEROBIN) ->NODO
- FIGLIOSINISTRO : (NODO,ALBEROBIN) -> NODO
- FIGLIODESTRO : (NODO,ALBEROBIN) -> NODO
- SINISTROVUOTO : (NODO,ALBEROBIN) ->BOOLEANO
- DESTROVUOTO : (NODO,ALBEROBIN) -> BOOLEANO
- COSTRBINALBERO : (ALBEROBIN,ALBEROBIN) -> ALBEROBIN
- CANCSOTTOBINALBERO : (NODO,ALBEROBIN) -> ALBEROBIN
- LEGGINODO: (NODO,ALBEROBIN) -> TIPOELEM
- SCRIVINODO: (TIPOELEM,NODO,ALBEROBIN) -> ALBEROBIN
- INSBINRADICE: (NODO, ALBEROBIN)-> ALBEROBIN
- INSFIGLIOSINISTRO: (NODO, ALBEROBIN)-> ALBEROBIN
- INSFIGLIODESTRO: (NODO, ALBEROBIN)-> ALBEROBIN

LA SPECIFICA SEMANTICA
TIPI: ALBEROBIN: insieme degli alberi binari T=(N,A), nei quali ad ogni nodo è associato un LIVELLO, BOOLEANO, NODO. TIPOELEM del tipo dell'etichetta

- CREABINALBERO = T'

POST: T' = (∅,∅) = Λ

- BINALBEROVUOTO(T) = b

POST: b=VERO SE T = Λ ; b=FALSO ALTRIMENTI

- BINRADICE(T) = u

PRE: T ≠ Λ
POST: u->RADICE DI T->LIVELLO(u) = 0

- BINPADRE(u,T) = v

PRE: T ≠ Λ, u ∈ N, LIVELLO(u) > 0
POST: v E' PADRE DI u->(v,u) ∈ A->LIVELLO(u)=LIVELLO(v)+1

- FIGLIOSINISTRO(u,T) = v

PRE: T ≠ Λ, u ∈ N, u HA UN FIGLIO SINISTRO
POST: v E' IL FIGLIO SINISTRO DI u IN T

- FIGLIODESTRO(u,T) = v

PRE: T ≠ Λ, u ∈ N, u HA UN FIGLIO DESTRO
POST: v E' IL FIGLIO DESTRO DI u IN T

- SINISTROVUOTO(u,T) = b

PRE: $T \neq \Lambda$, $u \in N$
POST: b=VERO SE u NON HA UN FIGLIO SINISTRO; b=FALSO ALTRIMENTI

- DESTROVUOTO(u,T) = b

PRE: $T \neq \Lambda$, $u \in N$
POST: b=VERO SE u NON HA UN FIGLIO DESTRO; b=FALSO ALTRIMENTI

- COSTRBINALBERO(T,T') = T"

POST: T" SI OTTIENE DA T E DA T' INTRODUCENDO AUTOMATICAMENTE UN NUOVO NODO r" (RADICE DI T") CHE AVRA' COME SOTTOALBERO SINISTRO T E SOTTOALBERO DESTRO T' (*SE $T = \Lambda$ E $T' = \Lambda$, L'OPERATORE INSERISCE LA SOLA RADICE r"; SE T =* L, *r" NON HA FIGLIO SINISTRO; SE $T' = \Lambda$, r" NON HA FIGLIO DESTRO*)

- CANCSOTTOBINALBERO(u,T) = T'

PRE: $T \neq \Lambda$, $u \in N$
POST: T' E' OTTENUTO DA T ELIMINANDO IL SOTTOALBERO DI
RADICE u, CON TUTTI I SUOI DISCENDENTI
VALIDA PER ALBERI DI OGNI ORDINE AGISCE POTANDO DAL
NODO u.

- LEGGINODO (n,T) = a

PRE: n E' UN NODO DI T, $n \in N$
POST: a E' IL VALORE ASSOCIATO AL NODO n IN T

- SCRIVINODO (a,n,T) = T'

PRE: n E' UN NODO DI T, $n \in N$
POST: T' E' IL NUOVO ALBERO CORRISPONDENTE AL VECCHIO T CON IL VALORE a ASSEGNATO AL NODO n

- SETBINRADICE (n,T) = T'

POST: se T=Λ, T'= (N,A) N={n }, A=∅, LIVELLO(n)=0
Altrimenti T'= (N,A) N'= N U {n }-{v}, A=∅, LIVELLO(n)=0, dove v è la vecchia radice che viene sostituita da n

- SETFIGLIOSINISTRO (n,T) = T'

PRE: $T \neq \Lambda$, $n \in N$
POST: se SINISTROVUOTO(n)= Vero, N'= N U {v }, T' è uguale a T con v come figlio sinistro di n, altrimenti N'= N U {v}-{z}, dove z è il vecchio figlio sinistro di n viene sostituita da v

- SETFIGLIODESTRO (n,T) = T'

PRE: $T \neq \Lambda$, $n \in N$
POST: se DESTROVUOTO(n)= Vero, N'= N U {v }, T' è uguale a T con v come figlio destro di n, altrimenti N'= N U {v}-{z}, dove z è il vecchio figlio destro di n viene sostituita da v

1.2 Coda

Specifica Sintattica

TIPI: CODA, BOOLEAN, TIPOELEM

- CREACODA: () -> CODA
- CODAVUOTA: (CODA) -> BOOLEAN
- INCODA: (CODA,TIPOELEM) ->
- LEGGICODA: (CODA) -> TIPOELEM
- FUORICODA: (CODA)-> CODA

Specifica Semantica

- CREACODA: ()= C

POST: C= Λ

- CODAVUOTA: (C)= b

POST: b=Vero se C= Λ, b=Falso altrimenti

- INCODA: (C,a)= C'

PRE: C=<a1,a2,a3,a4,an> n>=0

POST: C'=<a1,a2,a3,a4,an,a>

- LEGGICODA: (C)= a

PRE: C=<a1,a2,a3,an> n>0

POST: a=a1

- FUORICODA: (C)=C'

PRE: C=<a1,a2,a3,an> n>0

POST: C'=<a2,a3,an>, C'= Λ se n=1

3. STRATEGIA SOLUTIVA

E' necessario suddividere il problema principale in sottoproblemi:

- 3.0 Main di Gestione
- 3.1 Acquisizione dell'Espressione Booleana
- 3.2. Trasformazione dell'Espressione Booleana in Albero Binario
- 3.3 Modifica dell'Espressione Booleana
  - 3.4.0 Gestione Modifica dell'Espressione Booleana
  - 3.4.1 Visualizzazione a Video dell'Albero
  - 3.4.2 Scelta del Nodo
  - 3.4.3 Applicazione della Distributiva
  - 3.4.4 Applicazione della Distributiva Inversa
  - 3.4.5 Applicazione di DeMorgan
- 3.5 Visualizzazione a Video dell'Espressione

Alcuni problemi possono essere risolti indipendentemente da altri, altri invece hanno bisogno del/dei sottoproblema/i corrispondenti per poter essere completati.
Inoltre è necessario prendere decisioni in fase di sviluppo riguardo a ciò che è stato risaltato in fase di analisi.

Scelte generali di Progettazione:

- Il Not può essere legato solo agli operandi semplici in fase di input, mentre è possibile legarlo anche agli operatori in fase di modifica dell'espressione,
- La doppia negazione è utilizzata durante l'applicazione delle leggi di De Morgan, ma non è possibile utilizzarla indipendentemente
- La distributiva e l'inversa della distributiva possono essere applicate solo ad operatori a cui non è legato il Not
- L'espressione booleana finale è del tipo (x OP y) oppure !(x OP y)

RISOLUZIONE DEI SOTTOPROBLEMI:

- 3.0. Main di Gestione

Linguaggio Lineare
Inizio
Ripeti
Acquisizione dell'Espressione Booleana *[vedere sottoproblema #3.1#]*
Fintantoché (stringa Errata)
Trasformazione dell'Espressione in Albero Binario *[vedere sottoproblema #3.2#]*
Modifica dell'Espressione Booleana *[vedere sottoproblema #3.3#]*
Visualizzazione a Video dell'Espressione Modificata *[vedere sottoproblema #3.4#]*
Fine

- 3.1. Acquisizione dell'Espressione Booleana

Considerazioni:
La Stringa viene acquisita da tastiera. Al termine vengono eliminati tutti gli spazi
Input:
- Input da Tastiera

Output:
- Stringa dell'espressione booleana

Linguaggio Lineare:
Inizio
Leggi input da Tastiera
Inserisci input in una Stringa
Fintantoché ci sono spazi nella stringa
Elimina spazio
Fine_Fintantochè
Fine

- 3.2. Trasformazione dell'Espressione Booleana in Albero Binario

Considerazioni:
Le espressioni sono valide se del tipo (a OP b), dove
- OP è AND or OR
- a e b possono essere a loro volta espressioni booleane
- è possibile legare l'operatore NOT solo ad operandi semplici, ma non agli operatori

E' un *Algoritmo Ricorsivo*

Input:
- stringa -> Stringa dell'espressione booleana

Output:
- alberoFinale -> Albero Binario dell'espressione booleana

Variabili utilizzate
- alberoSinistro, alberoDestro –> Albero Binario
- stringaSinistra, stringaDestra –> Stringa
- operatore -> Stringa

Linguaggio Lineare:
LogicalTree (stringa)
Inizio
creaAlbero (alberoFinale)
creaAlbero (alberoDestro)
creaAlbero (alberoSinistro);
Se (primo carattere è "(") *allora*
Assegna alberoSinistro = LogicalTree (stringa senza primo carattere)
Altrimenti
Fintantoché (non trova un operatore "&" o "|")
Scorri la stringa un carattere alla volta
Concatena a stringaSinistra il carattere letto

Fine_Fintantochè
alberoSinistro = creaBinAlbero(null, null)
alberoSinistro.scriviNodo(alberoSinistro.binRadice, stringaSinistra)
Fine_Se
*Se (*non è entrato nel ramo ricorsivo dell'If)
Assegna a operatore il carattere corrente della stringa
Altrimenti
Trova l'operatore corrispondente
Assegna a operatore il carattere corrente della stringa
Fine_Se
Leggi la stringa dal carattere dopo l'operatore
Se (carattere è "(") *allora*
Assegna alberoDestro = LogicalTree (stringa senza primo carattere)
Altrimenti
Fintantoché (non trova ")")
Scorri la stringa un carattere alla volta
Concatena a stringaDestra il carattere letto
Fine_Fintantochè
alberoDestro = creaBinAlbero(null, null)
alberoDestro.scriviNodo(alberoDestro.binRadice, stringaDestra)
Fine_Se
alberoFinale = creaBinAlbero(alberoSinistro, AlberoDestro)
alberoFinale.scriviNodo(alberoFinale.binRadice, operatore)
Fine

- 3.3. Modifica dell'Espressione Booleana

Input:
- Albero Binario dell'Espressione Booleana
- Scelta dell'utente

Output:
- Albero Binario dell'Espressione Booleana modificata

Liguaggio Lineare:
Inizio
Ripeti
Visualizzazione a Video della Stringa [vedere sottoproblema #3.3.0#]
Leggere il nodo da modificare *[vedere sottoproblema #3.3.1#]*
Leggere quale modifica effettuare
Passare al relativo sottoproblema [#3.3.3 / #3.3.4]
Fintantoché (l'utente non sceglie di uscire)
Fine

o 3.3.1. Visualizzazione a Video dell'Albero

Considerazioni:
L'algoritmo è una modifica dell'algoritmo della Visita di un Albero Binario in Ampiezza.

Input:
- albero -> Albero Binario dell'Espressione Booleana

Output:
- Caratteri a video

Variabili Utilizzate:
- coda -> Coda di Nodi
- indice -> valore intero
- NodoT -> Nodo dell'albero

Liguaggio Lineare:
Inizio
Visualizza Etichetta della Radice
Incoda Radice
Assegna indice=2
*Fintantoché (*la coda non è vuota)
leggiCoda e *Assegnalo* a NodoT
fuoriCoda
Se (il figlio sinistro non è vuoto AND il figlio sinistro è un operatore)
Visualizza valore di i a Video
Visualizza a Video del SottoAlbero generato dal Figlio sinistro
Incoda figlio Sinistro del NodoT
Incrementa i di 1
Fine_Se
Se (il figlio sinistro non è vuoto AND il figlio sinistro è un operatore)
Visualizza valore di i a Video
Visualizza a Video del SottoAlbero generato dal Figlio sinistro
Incoda figlio Destro del NodoT
Incrementa i di 1
Fine_Se
Fine_Fintantochè
Fine

o 3.3.2. Scelta del Nodo

Considerazioni:
Questo è una rielaborazione dell'algoritmo di Visualizzazione a video dell'Albero (algoritmo #3.3.0)

Input:
- albero -> Albero Binario dell'Espressione Booleana
- scelta nodo

Output:
- Nodo T -> Nodo selezionato

Variabili Utilizzate:
- coda -> Coda di Nodi
- indice -> valore intero

Liguaggio Lineare:
Inizio

```
Se (scelta=1)
        T=Etichetta della Radice
Altrimenti
        Incoda Radice
        Assegna indice=1
        Fintantoché (la coda non è vuota)
                leggiCoda e Assegnalo a NodoT
                fuoriCoda
                Se (il figlio sinistro non è vuoto AND il figlio sinistro è un operatore)
                        Se (il valore di scelta è uguale al valore di i)
                                Termina l'algoritmo restituendo T
                        Altrimenti
                                Incoda figlio Sinistro del NodoT
                                Incrementa i di 1
                        Fine_Se
                Se (il figlio sinistro non è vuoto AND il figlio sinistro è un operatore)
                        Se (il valore di scelta è uguale al valore di i)
                                Termina l'algoritmo restituendo T
                        Altrimenti
                                Incoda figlio Destro del NodoT
                                Incrementa i di 1
                        Fine_Se
                Fine_Se
        Fine_Fintantochè
Fine_Se
Fine
```

o 3.3.3. Applicazione della Distributiva

Considerazioni:
Come notato in analisi, la Distributiva include quattro casi differenti, ma tutti identici nella loro risoluzione. Di seguito è riportato solo il caso del nodo del tipo *(p OR (q AND r)*. Per gli altri è necessario solo invertire Destro con Sinistro o AND con OR a seconda dei casi:

1) *(p OR (q AND r) -> caso analizzato*
2) *((q AND r) OR p) -> invertire destro con sinistro rispetto al caso analizzato*
3) *(p AND (q OR r) -> invertire AND con OR e viceversa*
4) *((q OR r) AND p) -> invertire AND con OR e viceversa, invertire destro con sinistro rispetto al caso analizzato*

Input:
- albero -> Albero Binario dell'Espressione Booleana
- nodo -> Nodo a cui applicare la modifica

Output:
- albero -> Albero Binario dell'Espressione Booleana modificata

Variabili utilizzate
- alberoSinistro, alberoDestro, p, q, r –> Albero Binario

Liguaggio Lineare:
Inizio
Se (il contenudo del nodo è & AND il contenudo del figlio destro del nodo è |)
Radice di p =figlioSinistro del nodo
Radice di q =figlioSinistro del figlio Destro del nodo
Radice di r = figlioDestro del figlio Destro del nodo
scriviNodo (albero, nodo, "&")
creaAlbero(alberoSinistro)
alberoSinistro=costrBinAlbero(p, q)
creaAlbero(alberoDestro)
alberoDestro=costrBinAlbero(p, r)
albero=costrBinAlbero(alberoSinistro, alberoDestro)
Assegna al nodo iniziale l'albero appena creato
Altrimenti
Nessuna modifica effettuata
Fine_Se
Fine

o 3.3.4. Applicazione della Distributiva Inversa

Considerazioni:
E' molto simile all'algoritmo della Distributiva. Anche qui, come notato in analisi, la Distributiva include quattro casi differenti, ma tutti identici nella loro risoluzione. Di seguito è riportato solo il caso del nodo del tipo *(p OR (q AND r).* Per gli altri è necessario solo invertire Destro con Sinistro o AND con OR a seconda dei casi:

1) *(p AND q) OR (p AND r)* -> caso analizzato
2) *(q AND p) OR (r AND p) -> invertire destro con sinistro rispetto al caso analizzato*
3) *(p OR q) AND (p OR r) -> invertire AND con OR e viceversa*
4) *(q OR p) AND (r OR p) -> invertire AND con OR e viceversa, invertire destro con sinistro rispetto al caso analizzato*

Input:
- albero -> Albero Binario dell'Espressione Booleana
- nodo -> Nodo a cui applicare la modifica

Output:
- albero -> Albero Binario dell'Espressione Booleana modificata

Variabili utilizzate
- alberoSinistro, alberoDestro, p, q, r –> Albero Binario

Liguaggio Lineare:
Inizio
Se (il contenudo del nodo è | AND il contenudo di entrambi i figli è destro del nodo è &)
Se (il figli sinistri di entrambi i figli del nodo sono uguali)
Radice di p =figlioSinistro del figlio sinistro del nodo
Radice di q =figlioDestro del figlio Sinistro del nodo
Radice di r = figlioDestro del figlio Sinistro del nodo

```
                    scriviNodo (albero, nodo, "|")
                    creaAlbero(alberoSinistro)
                    alberoSinistro=p
                    creaAlbero(alberoDestro)
                    alberoDestro=costrBinAlbero(q, r)
                    albero=costrBinAlbero(alberoSinistro, alberoDestro)
                    Assegna al nodo iniziale l'albero appena creato
                Altrimenti
                    Nessuna modifica effettuata
                Fine_Se
        Fine_Se
Fine
```

o 3.3.5. Applicazione delle Leggi di DeMorgan

Considerazioni:
Vedere le considerazioni sulle leggi di DeMorgan fatte in analisi.

Input:
- Albero Binario dell'Espressione Booleana
- Nodo a cui applicare la trasformazione

Output:
- Albero Binario dell'Espressione Booleana modificata

Liguaggio Lineare:

```
Inizio
        Leggi Nodo
        Se (Nodo==”OR”)
                Nodo = NOT AND
        Fine_Se

        Se (Nodo==”NOT OR”)
                Nodo = AND
        Fine_Se

        Se (Nodo==”AND”)
                Nodo = NOT OR
        Fine_Se

        Se (Nodo==”NOT AND”)
                Nodo = OR
        Fine_Se

        FiglioSinistro del Nodo = ! FiglioSinistro del Nodo
        FiglioDestro del Nodo = ! FiglioDestro del Nodo
Fine
```

- 3.4. Visualizzazione a Video della Stringa

Considerazioni:
Le stringhe sono valide se del tipo (a OP b) oppure NOT(a OP B), dove
- OP è AND or OR
- a e b possono essere a loro volta espressioni booleane
- è possibile legare l'operatore NOT ad operandi semplici

Questo algoritmo è una modifica della BinInVisita di un albero: sono state aggiunti due controlli sulla radice all'inizio e alla fine dell'algoritmo, per visualizzare le parentesi
E' un *Algoritmo Ricorsivo*

Input:
- Albero Binario dell'Espressione Booleana -> albero
- Radice dell'Albero -> nodo

Output:
- Caratteri su schermo

Liguaggio Lineare:

```
InVisitaModificata(albero, nodo)
Inizio
    Se (il contenudo del nodo è un operatore & o |)
      Se (al nodo è legato NOT)
            Visualizza a video "|("
      Altrimenti
            Visualizza a video "("
      Fine_Se

      Se (Figlio Sinistro del nodo non è vuoto)
          InVisitaModificata(albero, figlio Sinistro del nodo)
      Fine_Se

      Se (il contenudo del nodo è un operatore & o |)
            Visualizza a video l'etichetta
      Altrimenti
          Se (al nodo è legato NOT)
              Visualizza a video l'etichetta preceduta dal carattere "!"
          Altrimenti
              Visualizza a video l'etichetta
         Fine_Se
      Fine_Se

      Se (Figlio Destro del nodo non è vuoto)
          InVisitaModificata(albero, figlio Destro del nodo)
      Fine_Se

      Se (il contenudo del nodo è un operatore & o |)
            Visualizza a video ")"
      Fine_Se
Fine
```

4. CODIFICA

4.1 Considerazioni

Considerazioni generali:

Le implementazioni degli algoritmi nei due linguaggi forniscono lo stesso risultato, ma con delle piccole variazioni:

- Le espressioni booleane che è possibile acquisire con il programma creato con C++ sono limitate a *256* caratteri, mentre in Java sono limitate secondo l'API java.lang.String
- Entrambi i programmi vanno in errore con stringhe errate, ma la gestione delle stringhe errate da parte del programma creato con Java è maggiore. Questo è dovuto al diverso utilizzo delle try-catch nei due linguaggi. In Java le eccezioni catturate sono maggiori e ben definite, a differenza del C++
    - In C++, l'errore di stringa vuota viene gestito, mentre non vengono gestiti errori di altro tipo
    - In Java vengono gestiti errori del tipo:
        - stringa vuota
        - errori di assenza del primo o del secondo operando in espressioni semplici
- Dopo aver eliminato un Nodo da un Albero con il programma Java (ovvero vengono cancellati tutti i riferimenti al nodo), la memoria viene liberata automaticamente dalla GarbaceCollection. Nell'implementazione in C++, la liberazione della memoria viene fatta con una dolete all'interno del metodo CancBinAlbero
- In C++ il programma finale è contenuto nel file "*LogicalTree.exe*"
- In Java le classi sono suddivise in 3 package:
    - "*AlberoBinario*", contenente i file: AlberoBinario.class, Nodo.class e BinServizio.class
    - "*Coda*", contenente i file: "Coda.class" e "Nodoc.Class"
    - "*logicalTree*", contenente il file "Main.class"

Considerazioni sull'implementazione delle strutture:

- Con entrambi i linguaggi, l'Albero Binario è stato implementato attraverso due file
    - C++: Java: "*Nodo.h" e "AlberoBinario.h*"
    - Java: "*Nodo.java*" e "*AlberoBinario.java*"

  e utilizzando i puntatori. In C++ l'utilizzo dei puntatori è diretto, mentre in Java è "nascosto", poiché quando si crea un oggetto, quello che si gestisce non ì l'oggetto stesso, ma il valore del riferimento all'oggetto (che concettualmente è un puntatore)
  Inoltre all'Albero Binario sono state aggiunte delle funzioni di servizio contenute nei file:
    - C++: Java: "*BinServizio.h*"
    - Java: "*BinServizio.java"*

  Il file "BinServizio.java" implementa 3 metodi in più rispetto alla corrispondente versione C++:
    - BinPreVisita
    - BinInVisita
    - BinPostVisita

- Anche per la Coda, l'implementazione è avvenuta nello stesso modo nei due linguaggi e attraverso la stessa idea base, sfruttando i puntatori.

I file sorgenti sono:

- C++: Java: "*NodoC.h" e "Coda.h*"
- Java: "*NodoC.java*" e "*Coda.java*"

Considerazioni sull'implementazione del Main:

- La gestione del main e dei metodi necessari per risolvere i vari algoritmi risolti in fase di progettazione è uguale in entrambi i linguaggi.
- L'unico metodo che cambia nei due linguaggi e il "creaLogicalTree", che permette la trasformazione dell'Espressione booleana in Albero Binario, dovuta sostanzialmente alla diversa gestione delle stringe. In Java avviene completamente attraverso i metodi messi a disposizione dalla classe String, mentre in C++ avviene in parte utilizzando la libreria "string", in altre usando puntatori a caratteri.
- L'altra differenza è quella già citata del diverso utilizzo delle try-catch, ottenuta in C++, inserendo cactch che catturano un intero che indica il codice dell'errore.

4.2 Codifica C++

Main.ccp

```
/*
 * main.cpp
 *
 * @author Andrea Maiellaro & Domenico Pio Novelli
 * Data Creazione: 19 dicembre 2005
 * Data Ultima Modifica: 20 Dicembre 2005
 *
 */

#include <cstdlib>
#include <iostream>
#include <string>
#include "AlberoBinario.h"
#include "BinServizio.h"
#include "Coda.h"

using namespace std;

AlberoBinario* creaLogicalTree(char*);
AlberoBinario *modificaAlbero(AlberoBinario*);
void distributivaInv (AlberoBinario *albero, Nodo *nd);
//Per gli altri metodi i prototipi non sono necessari perchè sono
//dichiarati così come in sequenza di utilizzo

//Grandezza massima della stringa
const int MAX=256;

int main(int argc, char *argv[])
{

    //Dichiarazione della Stringa
    char str[MAX];
    char *strPunt=(char*)&str;
    AlberoBinario *alberoPunt;

        cout<<"\nSOFTWARE ACQUISIZIONE E GESTIONE ESPRESSIONI BOOLEANE\n";

        cout<<"\nIl metodo accetta espressioni del tipo";
        cout<<"\n\t(x&y) oppure (x|y)" ;
        cout<< "\ndove x e y possono a loro volta essere complesse";
        cout<< "\ne a cui puo' essere legato anche l'operatore ! (NOT)";
        cout<< "\nEsempio di espressioni valide: (A&B) oppure (!a&(B&!C))\n";

        //Ripeti fintantochè stringa non valida
        do{

            //Ripeti se stringa vuota
            do{
                //Acquisizione della stringa da Tastiera
                 cout << "\nInserire una espressione booleana: ";
                 cin.getline(strPunt, MAX, '\n');


                if (*str==NULL){
                    cout<<"\nStringa vuota\n";
```

```
            }
        }while (*str==NULL);

            //Eliminazione degli spazi all'interno della Stringa
            char *strPuntTemp=strPunt;
            while (*strPuntTemp!='\0'){
                  if (*strPuntTemp==' '){ //Se trova uno spazio
                     char *strPuntT=strPuntTemp;
                     while (*strPuntT!='\0'){
                           *strPuntT=*(strPuntT+1);
                           strPuntT++;
                     }
                  }
                  strPuntTemp++;
               }

        cout<<"\nStringa acquisita: "<<str;

        //Trasformazione della Stringa in Albero Binario
        alberoPunt=new AlberoBinario;
        alberoPunt = creaLogicalTree(strPunt+1);

        //Stampa in profondità
        if (alberoPunt!=NULL){
           cout << "\nAlbero Iniziale: ";
           binVisitaFin (alberoPunt, alberoPunt->binRadice());
        }
        else {
            cout<<"Stringa Errata";
        }
    }while (alberoPunt==NULL);

    alberoPunt=modificaAlbero (alberoPunt);

    //Stampa in profondità
    cout<<"\nAlbero Finale:";
    binVisitaFin(alberoPunt, alberoPunt->binRadice());
    cout<<"\n";

cout << "\n\n";
system("PAUSE");
return EXIT_SUCCESS;
}

/**
 * Metodo di acquisizione della Stringa e di creazione dell'Albero Binario
 *
 * Il metodo accetta stringhe del tipo
 *         (x&y) oppure (x|y)
 * dove x e y possono a loro volta essere complesse
 * e a cui può essere legato anche l'operatore !
 *
 * @param str Stringa da esaminare
 * @return Albero Binario della Stringa
 */
AlberoBinario* creaLogicalTree(char *str){

        //cout<<"Entrato nella funzione con la stringa: "+str);
```

```
            //Albero Padre
            AlberoBinario *logicalTree=new AlberoBinario();

            //Alberi: Figlio Sinistro e Figlio Destro
            AlberoBinario *logicalTreeLeft = new AlberoBinario;
            AlberoBinario *logicalTreeRight = new AlberoBinario;

            //Operatore
            string operatore="";
            char *operatoreP=NULL;

            //Stringa Primo Operando
            string str1="";

            //Stringa Secondo Operando
            string str2="";

            //Stringa temporane;
            char *strP=str;

            //Varibaili temporanea di scorrimento nella Stringa
            int i=0, j=0;


/* INIZIO AQUISIZIONE DEL PRIMO OPERANDO / FIGLIO SINISTRO */

            bool negato=false;

            // Se trova una Parentesi Tonda Aperta all'inizio
            if (*str=='('){
                    logicalTreeLeft=creaLogicalTree (str+1);
                } // fine if (parTonApB==0)
            else {

                //Scorri nella stringa
                while (*strP!='&' && *strP!='|'){

                    //Aggiorna operando di sinistra
                    if (i==0 && *strP=='!') {//Se c'è il NOT all'inizio
                       negato=true;
                    }
                    else {
                         str1=str1+(*strP);
                    }
                    strP++;
                    i++;
                    // cout<<"Posizione i= "+i);
                }

                //Memorizzo Posizione Operatore
                operatoreP = strP;

                logicalTreeLeft->costrBinAlbero(NULL, NULL);

                logicalTreeLeft->scriviNodo(logicalTreeLeft->binRadice(), str1,
negato);

              //cout << "FiglioSinistro= " << logicalTreeLeft-
>leggiNodo(logicalTreeLeft->binRadice())<<"\n";
```

```
                // cout << "Operando di sinistra= "<< str1<<"\n";

            }

    /* FINE AQUISIZIONE DEL PRIMO OPERANDO / FIGLIO SINISTRO */

        /* INIZIO AQUISIZIONE DELL'OPERATORE */

            if (i!=0) { //Se non è entrato nella chiamata ricorsiva dell'albero
sinistro

                //Acquisisci operatore
                operatore=*operatoreP;
                //cout << "Operatore = "<<operatore<<"\n";
                strP++;

            }
            else {
             /* Se entrato nell'if con ricorsione, è necessario recuperare
             * la posizione dell'operatore corrispondente
             */

             //operatore='o';
             //cout << "Operatore = "<<operatore<<"\n";


                //cout<<"SONO QUI: "+strP);

                char* strk=strP; //Stringa temporanea

                int k=0;

                //Ciclo per contare le parentesi tonde aperte iniziali

                //cout << "\nstrP prima del while = "<<strP<<"\n";

                while (*strk=='(') {
                    k++;
                    //cout<<"Posizione k= "<<k<<"\n";
                    strk++;
                }

                int h=0, l=0;
                char *strL=strP;

                while (h!=k){
                    //Ogni parentesi aperta, corrisponde ad una tonda chiusa

                    //Scorri nella stringa, fino a trovare la posizione
dell'operatore corrispondente
                    while ( (*strL!=')' && *strL+1!='&') && (*strL!=')' &&
*strL+1!='|') ){

                        l++;
                        strL++;
                        //cout<<"Posizione L= "+l);
                        //cout<<"strL= "+strL);
                    }
```

```
                    h++; //Trovata una parentesi chiusa
                    //cout<<"Posizione H= "+h);
                    strL=strL+2; //Elimina
                    l++; //Posizione eliminate
                }

                //cout<<"H ="<<h<<"\n";

                //cout<<"Posizione dopo while di L= "+l);

                //La Stringa inizia dall'operatore corrispondente
                if (h==0){
                     while ( (*strP!=')' && *strP+1!='&') || (*strP!=')' &&
*strP+1!='|') ){
                           strP++;
                           //cout<<"Nel Ciclo\n";
                     }
                     //cout<<"Operatore ELSE= "<<strP<<"\n";
                }
                else if (h>=1){
                    strP=str+l+h-1;
                    //cout<<"Operatore IF1= "<<strP<<"\n";
                }

                //cout << "strP dopo operatore = "<<strP<<"\n";

                //Acquisisci operatore
                operatore=*strP;
                strP++;

            }

    /* FINE  AQUISIZIONE DELL'OPERATORE */

    /* INIZIO AQUISIZIONE DEL SECONDO OPERANDO / FIGLIO DESTRO */

            //Quello di prima viene modificato
            negato=false;

            // Se trova una Parentesi Tonda Aperta all'inizio
            if (*strP=='('){
                    logicalTreeRight=creaLogicalTree (strP+1);
                } //
            else {

                //Scorri nella stringa
                while (*strP!=')'){

                    //Aggiorna operando di sinistra
                    if (j==0 && *strP=='!') {//Se c'è il NOT all'inizio
                       negato=true;
                    }
                    else {
                         str2=str2+(*strP);
                    }
                    strP++;
                    j++;
                    // cout<<"Posizione i= "+i);
                }
```

```
                //Memorizzo Posizione Operatore
                operatoreP = strP;

                logicalTreeRight->costrBinAlbero(NULL, NULL);

                logicalTreeRight->scriviNodo(logicalTreeRight->binRadice(), str2,
negato);

               //cout << "FiglioDestro= " << logicalTreeRight-
>leggiNodo(logicalTreeRight->binRadice())<<"\n";
               // cout << "Operando di destra= "<< str2<<"\n";

            }

    /* FINE AQUISIZIONE DEL SECONDO OPERANDO / FIGLIO DESTRO */

        /* CREAZIONE DELL'ALBERO
            /* Condizioni per una stringa valida:
             * Albero Sinistro non vuoto
             * Primo operatore lungo più di 0 caratteri
             * Albero Destro non vuoto
             * Secondo operatore lungo più di 0 caratteri
             */

            //Se la stringa non è valida
            if (logicalTreeLeft!=NULL && logicalTreeRight!=NULL){
                logicalTree->costrBinAlbero(logicalTreeLeft, logicalTreeRight);
                logicalTree->scriviNodo(logicalTree->binRadice(), operatore,
false);

                /**  Prove della costruzione dell'Albero:
                cout << "Radice Albero Intermedio Sinistro:";
                cout << logicalTreeLeft->leggiNodo(logicalTreeLeft-
>binRadice())<<"\n";
                cout << "Radice Albero Intermedio Destro:";
                cout << logicalTreeRight->leggiNodo(logicalTreeRight-
>binRadice())<<"\n";
                cout << "Operatore= "+operatore+"\n";
                cout << "Albero Intermedio: \n";
                binVisitaFin(logicalTree, logicalTree->binRadice());
                cout<<"\n\n";

                //**/

                /** Prove della Costruzione dell'Albero
                cout<<"Albero Intermedio Sinistro:");
                BinServizio.binVisitaFin(logicalTreeLeft,
logicalTreeLeft.binRadice());
                cout<<);
                cout<<"Albero Intermedio Destro:");
                BinServizio.binVisitaFin(logicalTreeRight,
logicalTreeRight.binRadice());
                cout<<);
                cout<<"Operatore= "+operatore+"\n");
                **/

                return logicalTree;
```

```
            }
            else //Se la Stringa è valida
                return NULL;
}

void visualizzaAmpiezza(AlberoBinario* albero){
        cout << "\n\nVisualizzazione dell'Albero in Ampiezza\n";
        cout << "Nodo 1= ";
        binVisitaFin(albero, albero->binRadice());
        cout << "\n";
        Coda* codaT = new Coda;
        codaT->inCoda(albero->binRadice());
        int i=2;
        while (!codaT->codaVuota()){
            //cout<<"SONO QUI");
            Nodo *N=codaT->leggiCoda();
            codaT->fuoriCoda();
            if ( !albero->sinistroVuoto(N) ){
                if (albero->leggiNodo(albero->figlioSinistro(N))=="&" || albero-
>leggiNodo(albero->figlioSinistro(N))=="|" || albero->leggiNodo(albero-
>figlioSinistro(N))=="!&" || albero->leggiNodo(albero->figlioSinistro(N))=="!|"){
                    //cout<<""+i+"= "+albero->leggiNodo(albero-
>figlioSinistro(N))+"\t");
                    cout << "\nNodo" << i << "= ";
                    binVisitaFin(albero, albero->figlioSinistro(N));
                    cout << "\n";
                    codaT->inCoda(albero->figlioSinistro(N));
                    i++;
                }
            }
            if ( !albero->destroVuoto(N) ){
                if (albero->leggiNodo(albero->figlioDestro(N))=="&" || albero-
>leggiNodo(albero->figlioDestro(N))=="|" || albero->leggiNodo(albero-
>figlioDestro(N))=="!&" || albero->leggiNodo(albero->figlioDestro(N))=="!|"){
                    //cout<<""+i+"= "+albero->leggiNodo(albero-
>figlioDestro(N))+"\t");
                    cout << "\nNodo"<<i<<"= ";
                    binVisitaFin(albero, albero->figlioDestro(N));
                    cout << "\n";
                    codaT->inCoda(albero->figlioDestro(N));
                    i++;
                }
            }

        }
    }

Nodo* scegliNodo (AlberoBinario* albero, int j){
      if (j!=1){
        Coda* codaT = new Coda;
        codaT->inCoda(albero->binRadice());
        int i=1;
        while (!codaT->codaVuota()){
            //cout<<"SONO QUI");
            Nodo *N=codaT->leggiCoda();
            codaT->fuoriCoda();
            if ( !albero->sinistroVuoto(N) ){
                if (albero->leggiNodo(albero->figlioSinistro(N))=="&" || albero-
>leggiNodo(albero->figlioSinistro(N))=="|" || albero->leggiNodo(albero-
```

```
>figlioSinistro(N))=="!&" || albero->leggiNodo(albero->figlioSinistro(N))=="!|"){
                    //cout<<""+i+"= "+albero->leggiNodo(albero-
>figlioSinistro(N))+"\t");
                    codaT->inCoda(albero->figlioSinistro(N));
                    i++;
                    if (j==i) //se il valore è lo stesso di quello richiesto
                       return albero->figlioSinistro(N);
                }
            }
            if ( !albero->destroVuoto(N) ){
                if (albero->leggiNodo(albero->figlioDestro(N))=="&" || albero-
>leggiNodo(albero->figlioDestro(N))=="|" || albero->leggiNodo(albero-
>figlioDestro(N))=="!&" || albero->leggiNodo(albero->figlioDestro(N))=="!|"){
                    //cout<<""+i+"= "+albero->leggiNodo(albero-
>figlioDestro(N))+"\t");
                    codaT->inCoda(albero->figlioDestro(N));
                    i++;
                    if (j==i) //se il valore è lo stesso di quello richiesto
                       return albero->figlioDestro(N);
                }
            }

        }
      }
      else{
           return albero->binRadice();
      }

      //restituisce NULL se j non è valido
      return NULL;
    }

AlberoBinario* deMorgan (AlberoBinario *albero, Nodo *nd){

        cout << "Nodo Vecchio= "+albero->leggiNodo(nd)<<"\n";

        //Se (Nodo=="OR")
        if (albero->leggiNodo(nd)=="|"){
            //Nodo = NOT AND
            albero->scriviNodo(nd, "&", true);
            cout << "Nuovo Nodo = " << albero->leggiNodo(nd);
        }

        //Se (Nodo=="NOT OR")
        else if (albero->leggiNodo(nd)=="!|") {
            //Nodo = AND
            albero->scriviNodo(nd, "&", false);
            cout << "Nuovo Nodo = "+albero->leggiNodo(nd);
        }

        //Se (Nodo=="AND")
        else if (albero->leggiNodo(nd)=="&"){
            //Nodo = NOT OR
            albero->scriviNodo(nd, "|", true);
            cout << "Nuovo Nodo = " << albero->leggiNodo(nd);
        }

        //Se (Nodo=="NOT AND")
        else if (albero->leggiNodo(nd)=="!&"){
```

```
            //Nodo = OR
            albero->scriviNodo(nd, "|", false);
            cout << "Nuovo Nodo = " << albero->leggiNodo(nd);
        }

        //FiglioSinistro del Nodo = ! FiglioSinistro del Nodo
        albero->setNot(albero->figlioSinistro(nd), !albero->getNot(albero-
>figlioSinistro(nd)));
        //FiglioDestro del Nodo = ! FiglioDestro del Nodo
        albero->setNot(albero->figlioDestro(nd), !albero->getNot(albero-
>figlioDestro(nd)));

        return albero;


    }

     /**
      * Applicazione della distributiva su un nodo:
      * 1) p OR (q AND r) => (p OR q) AND (p OR q)
      * 2) (q AND r) OR p => (q OR p) AND (r OR p)
      * 3) p AND (q OR r) => (p AND q) OR (p AND q)
      * 4) (q AND r) OR p => (q AND p) OR (r AND p)
      *
      *     NON accetta stringhe con NOT legato all'operatore
     **/
AlberoBinario* distributiva (AlberoBinario* albero, Nodo* nd){

        //cout<<"Radice Nodo= "+albero->leggiNodo(nd));
        //cout<<"Figlio destro del Nodo= "+albero->leggiNodo(albero-
>figlioDestro(nd)));

        AlberoBinario* alberoTemp=new AlberoBinario();

        // 1) p OR (q AND r) => (p OR q) AND (p OR q)
        if (albero->leggiNodo(nd)=="|" && albero->leggiNodo(albero-
>figlioDestro(nd))=="&"){

            //Memorizzazione di p
            AlberoBinario* p=new AlberoBinario();
            p->costrBinAlbero(NULL, NULL);
            p->setRadice(albero->figlioSinistro(nd));


            Nodo *figlioDestro=albero->figlioDestro(nd);

            //Memorizzazione di q
            AlberoBinario *q=new AlberoBinario();
            q->costrBinAlbero(NULL, NULL);
            q->setRadice(albero->figlioSinistro(figlioDestro));


            //Memorizzazione di r
            AlberoBinario *r=new AlberoBinario();
            r->costrBinAlbero(NULL, NULL);
            r->setRadice(albero->figlioDestro(figlioDestro));
            //r->binRadice()=albero->figlioDestro(figlioDestro); NON FUNZIONA

            //Costruzione del figlio Sinistro
```

```
            AlberoBinario *alberoSinistro=new AlberoBinario();
            alberoSinistro->costrBinAlbero(p, q);
            //Assegna OR alla radice del figlio sinistro
            alberoSinistro->scriviNodo(alberoSinistro->binRadice(), "|", false);

            //Costruzione del figlio Destro
            AlberoBinario *alberoDestro=new AlberoBinario();
            alberoDestro->costrBinAlbero(p, r);
            //Assegna OR alla radice del figlio destro
            alberoDestro->scriviNodo(alberoDestro->binRadice(), "|", false);

            //Ricostruzione del SottoAlbero
            AlberoBinario *sottoAlbero=new AlberoBinario();
            sottoAlbero->costrBinAlbero(alberoSinistro, alberoDestro);
            sottoAlbero->scriviNodo(sottoAlbero->binRadice(), "&", false);

            //cout<<"Nuovo SottoAlbero");
            //BinServizio->binVisitaFin(sottoAlbero, sottoAlbero->binRadice());
            //cout<<"\n");

            if (albero->binPadre(nd)!=NULL) { //Se non è la radice dell'albero
iniziale
                //Aggiorna il padre del nodo
                if (albero->figlioDestro(albero->binPadre(nd))==nd){
                    albero->setFiglioDestro(albero->binPadre(nd), sottoAlbero-
>binRadice());
                }
                if (albero->figlioSinistro(albero->binPadre(nd))==nd){
                    albero->setFiglioSinistro(albero->binPadre(nd), sottoAlbero-
>binRadice());
                }
                nd=sottoAlbero->binRadice();
            }
            else {//Se il nodo è la radice dell'Albero iniziale
                //Aggiornamento del nodo
                nd=sottoAlbero->binRadice();
                albero->setRadice(nd);
            }
        }

        // 2) (q AND r) OR p => (q OR p) AND (r OR p)
        else if (albero->leggiNodo(nd)=="|" && albero->leggiNodo(albero-
>figlioSinistro(nd))=="&"){

            //Memorizzazione di p
            AlberoBinario *p=new AlberoBinario();
            p->costrBinAlbero(NULL, NULL);
            p->setRadice(albero->figlioDestro(nd));

            Nodo *figlioSinistro=albero->figlioSinistro(nd);

            //Memorizzazione di q
            AlberoBinario *q=new AlberoBinario();
            q->costrBinAlbero(NULL, NULL);
            q->setRadice(albero->figlioSinistro(figlioSinistro));

            //Memorizzazione di r
            AlberoBinario *r=new AlberoBinario();
            r->costrBinAlbero(NULL, NULL);
```

```
            r->setRadice(albero->figlioDestro(figlioSinistro));

            //Costruzione del figlio Sinistro
            AlberoBinario *alberoSinistro=new AlberoBinario();
            alberoSinistro->costrBinAlbero(q, p);
            //Assegna OR alla radice del figlio sinistro
            alberoSinistro->scriviNodo(alberoSinistro->binRadice(), "|", false);

            //Costruzione del figlio Destro
            AlberoBinario *alberoDestro=new AlberoBinario();
            alberoDestro->costrBinAlbero(r, p);
            //Assegna OR alla radice del figlio destro
            alberoDestro->scriviNodo(alberoDestro->binRadice(), "|", false);

            //Ricostruzione del SottoAlbero
            AlberoBinario *sottoAlbero=new AlberoBinario();
            sottoAlbero->costrBinAlbero(alberoSinistro, alberoDestro);
            sottoAlbero->scriviNodo(sottoAlbero->binRadice(), "&", false);

            //cout<<"Nuovo SottoAlbero");
            //BinServizio->binVisitaFin(sottoAlbero, sottoAlbero->binRadice());
            //cout<<"\n");

            if (albero->binPadre(nd)!=NULL) { //Se non è la radice dell'albero
iniziale
                //Aggiorna il padre del nodo
                if (albero->figlioDestro(albero->binPadre(nd))==nd){
                    albero->setFiglioDestro(albero->binPadre(nd), sottoAlbero-
>binRadice());
                }
                if (albero->figlioSinistro(albero->binPadre(nd))==nd){
                    albero->setFiglioSinistro(albero->binPadre(nd), sottoAlbero-
>binRadice());
                }
                nd=sottoAlbero->binRadice();
            }
            else {//Se il nodo è la radice dell'Albero iniziale
                //Aggiornamento del nodo
                nd=sottoAlbero->binRadice();
                albero->setRadice(nd);
            }
        }

        // 3) p AND (q OR r) => (p AND q) OR (p AND q)
        else if (albero->leggiNodo(nd)=="&" && albero->leggiNodo(albero-
>figlioDestro(nd))=="|"){

            //Memorizzazione di p
            AlberoBinario *p=new AlberoBinario();
            p->costrBinAlbero(NULL, NULL);
            p->setRadice(albero->figlioSinistro(nd));

            Nodo *figlioDestro=albero->figlioDestro(nd);

            //Memorizzazione di q
            AlberoBinario *q=new AlberoBinario();
            q->costrBinAlbero(NULL, NULL);
            q->setRadice(albero->figlioSinistro(figlioDestro));
```

```
            //Memorizzazione di r
            AlberoBinario *r=new AlberoBinario();
            r->costrBinAlbero(NULL, NULL);
            r->setRadice(albero->figlioDestro(figlioDestro));

            //Costruzione del figlio Sinistro
            AlberoBinario *alberoSinistro=new AlberoBinario();
            alberoSinistro->costrBinAlbero(p, q);
            //Assegna OR alla radice del figlio sinistro
            alberoSinistro->scriviNodo(alberoSinistro->binRadice(), "&", false);

            //Costruzione del figlio Destro
            AlberoBinario *alberoDestro=new AlberoBinario();
            alberoDestro->costrBinAlbero(p, r);
            //Assegna OR alla radice del figlio destro
            alberoDestro->scriviNodo(alberoDestro->binRadice(), "&", false);

            //Ricostruzione del SottoAlbero
            AlberoBinario *sottoAlbero=new AlberoBinario();
            sottoAlbero->costrBinAlbero(alberoSinistro, alberoDestro);
            sottoAlbero->scriviNodo(sottoAlbero->binRadice(), "|", false);

            //cout<<"Nuovo SottoAlbero");
            //BinServizio->binVisitaFin(sottoAlbero, sottoAlbero->binRadice());
            //cout<<"\n");

            if (albero->binPadre(nd)!=NULL) { //Se non è la radice dell'albero
iniziale
                //Aggiorna il padre del nodo
                if (albero->figlioDestro(albero->binPadre(nd))==nd){
                    albero->setFiglioDestro(albero->binPadre(nd), sottoAlbero-
>binRadice());
                }
                if (albero->figlioSinistro(albero->binPadre(nd))==nd){
                    albero->setFiglioSinistro(albero->binPadre(nd), sottoAlbero-
>binRadice());
                }
                nd=sottoAlbero->binRadice();
            }
            else {//Se il nodo è la radice dell'Albero iniziale
                //Aggiornamento del nodo
                nd=sottoAlbero->binRadice();
                albero->setRadice(nd);
            }
        }

        // 4) (q AND r) OR p => (q AND p) OR (r AND p)
        else if (albero->leggiNodo(nd)=="&" && albero->leggiNodo(albero-
>figlioSinistro(nd))=="|"){

            //Memorizzazione di p
            AlberoBinario *p=new AlberoBinario();
            p->costrBinAlbero(NULL, NULL);
            p->setRadice(albero->figlioDestro(nd));

            Nodo *figlioSinistro=albero->figlioSinistro(nd);

            //Memorizzazione di q
            AlberoBinario *q=new AlberoBinario();
```

```
            q->costrBinAlbero(NULL, NULL);
            q->setRadice(albero->figlioSinistro(figlioSinistro));

            //Memorizzazione di r
            AlberoBinario *r=new AlberoBinario();
            r->costrBinAlbero(NULL, NULL);
            r->setRadice(albero->figlioDestro(figlioSinistro));

            //Costruzione del figlio Sinistro
            AlberoBinario *alberoSinistro=new AlberoBinario();
            alberoSinistro->costrBinAlbero(q, p);
            //Assegna OR alla radice del figlio sinistro
            alberoSinistro->scriviNodo(alberoSinistro->binRadice(), "&", false);

            //Costruzione del figlio Destro
            AlberoBinario *alberoDestro=new AlberoBinario();
            alberoDestro->costrBinAlbero(r, p);
            //Assegna OR alla radice del figlio destro
            alberoDestro->scriviNodo(alberoDestro->binRadice(), "&", false);

            //Ricostruzione del SottoAlbero
            AlberoBinario *sottoAlbero=new AlberoBinario();
            sottoAlbero->costrBinAlbero(alberoSinistro, alberoDestro);
            sottoAlbero->scriviNodo(sottoAlbero->binRadice(), "|", false);

            //cout<<"Nuovo SottoAlbero");
            //BinServizio->binVisitaFin(sottoAlbero, sottoAlbero->binRadice());
            //cout<<"\n");

            if (albero->binPadre(nd)!=NULL) { //Se non è la radice dell'albero
iniziale
                //Aggiorna il padre del nodo
                if (albero->figlioDestro(albero->binPadre(nd))==nd){
                    albero->setFiglioDestro(albero->binPadre(nd), sottoAlbero-
>binRadice());
                }
                if (albero->figlioSinistro(albero->binPadre(nd))==nd){
                    albero->setFiglioSinistro(albero->binPadre(nd), sottoAlbero-
>binRadice());
                 }
                 nd=sottoAlbero->binRadice();
            }
            else {//Se il nodo è la radice dell'Albero inizial(e
                //Aggiornamento del nodo
                nd=sottoAlbero->binRadice();
                albero->setRadice(nd);
            }
        }


        // Se non viene effettuata nessuna modifica
        else{
            cout << "Nessuna modifica effettuata";
        }

        //cout<<"Nuovo Albero");
        //BinServizio->binVisitaFin(albero, albero->binRadice());
        //cout<<"\n");
```

```
        return albero;
    }

AlberoBinario* modificaAlbero (AlberoBinario *albero){

        int scelta=-1;

        do{

            // Visualizzazione a Video della Stringa
            visualizzaAmpiezza(albero);


            try{

            //Necessaria altrimenti rimane memorizzata la scelta precedente
                scelta=-1;

                //Leggere il nodo da modificare
                        cout << "\nScegliere il Nodo da modificare (0 per uscire):
";
                        cin >> scelta;
            }
            catch (int x){
                        //Digitazione di un carattere che non è un numero
                        cout << "Inserito valore errato";
            }

            //Nodo da Modificare
            //Nodo N = albero.figlioDestro(albero.binRadice());
            Nodo *N = scegliNodo(albero, scelta);

            //Controllare se il numero del Nodo è valido
            // !!! INSERIRE CONTROLLO !!!
            bool valido = false;

            if (scelta!=0){

                //Controllare se il numero del Nodo è valido
                if (N!=NULL){

                int scelta2=-1;

                //Chiedere quale trasformazione applicare
                do{
                    try{

                        //Necessaria altrimenti rimane memorizzata la scelta
precedente
                        scelta2=-1;

                        //Leggere il nodo da modificare
                        cout << "Modifiche possibili: ";
                        cout << "\n\t1) Applicazione della Legge di DeMorgan ";
                        cout << "\n\t2) Applicazione della Distributiva ";
                        cout << "\n\t3  Applicazione della Distributiva Inversa:";
                        cout << "\nInserire scelta (0 per uscire):";
                        cin >> scelta2;
                    }
```

```
                    catch (int i){
                  //Digitazione di un carattere che non è un numero
                    cout << "Inserito valore errato";
                    }
                }while(scelta2<0 || scelta2>3);

                //Passare al relativo sottoproblema
                if (scelta2!=0){

                    if (scelta2==1){
                        //Applicazione di Demorgan al nodo N
                        albero=deMorgan (albero, N);
                    }
                    else if (scelta2==2){
                        //Applicazione della Distributiva al nodo N
                        albero=distributiva(albero,N);
                    }
                    else if (scelta2==3)
                        distributivaInv (albero, N);
                    else{
                         cout << "Inserito numero errato\n";
                    }


                  } //if (N!=null)

                  } // if (scelta!=0)

            }
            //System.out.println ("\nFINE CICLO\n");

        } while (scelta!=0);

        return albero;
    }

    /**
      * Applicazione della distributiva Inversa su un nodo
      * 1) p OR (q AND r) <= (p OR q) AND (p OR q)
      * 2) (q AND r) OR p <= (q OR p) AND (r OR p)
      * 3) p AND (q OR r) <= (p AND q) OR (p AND q)
      * 4) (q AND r) OR p <= (q AND p) OR (r AND p)
      *
      *     NON accetta stringhe con NOT legato all'operatore
     **/
     void distributivaInv (AlberoBinario *albero, Nodo *nd){

        //System.out.println("Radice Nodo= "+albero.leggiNodo(nd));
        //System.out.println("Figlio destro del Nodo=
"+albero.leggiNodo(albero.figlioDestro(nd)));

        AlberoBinario *alberoTemp=new AlberoBinario;

        if (albero->leggiNodo(nd)=="&") {
            // 1) p OR (q AND r) <= (p OR q) AND (p OR q)
            if (albero->leggiNodo(albero->figlioDestro(nd))=="|" && albero-
>leggiNodo(albero->figlioSinistro(nd))=="|"){
                  Nodo *p1N = albero->figlioSinistro(albero->figlioSinistro(nd));
                  string p1S=binVisitaRet(albero, p1N);
```

```
                    Nodo *p2N = albero->figlioSinistro(albero->figlioDestro(nd));
                    string p2S=binVisitaRet(albero, p2N);
                    //cout<<"P1 = "+p1S+"\nP2 = "+p2S);
                    if (p1S==p2S) {

                          //Memorizzazione di p
                          AlberoBinario *p=new AlberoBinario;
                          p->costrBinAlbero(NULL, NULL);
                          p->setRadice(p1N);

                          //Memorizzazione di q
                          AlberoBinario *q=new AlberoBinario;
                          q->costrBinAlbero(NULL, NULL);
                          q->setRadice(albero->figlioDestro(albero-
>figlioSinistro(nd)));

                          //Memorizzazione di r
                          AlberoBinario *r=new AlberoBinario;
                          r->costrBinAlbero(NULL, NULL);
                          r->setRadice(albero->figlioDestro(albero-
>figlioDestro(nd)));

                          //Costruzione del figlio Sinistro
                          AlberoBinario *alberoSinistro=new AlberoBinario;
                          alberoSinistro=p;

                          //Costruzione del figlio Destro
                          AlberoBinario *alberoDestro=new AlberoBinario;
                          alberoDestro->costrBinAlbero(q, r);
                          //Assegna OR alla radice del figlio destro
                          alberoDestro->scriviNodo(alberoDestro->binRadice(), "&",
false);

                          //Ricostruzione del SottoAlbero
                          AlberoBinario *sottoAlbero=new AlberoBinario;
                          sottoAlbero->costrBinAlbero(alberoSinistro, alberoDestro);
                          sottoAlbero->scriviNodo(sottoAlbero->binRadice(), "|",
false);

                          //cout<<"Nuovo SottoAlbero");
                          //BinServizio->binVisitaFin(sottoAlbero, sottoAlbero-
>binRadice());
                          //cout<<"\n");

                          if (albero->binPadre(nd)!=NULL) { //Se non è la radice
dell'albero iniziale
                              //Aggiorna il padre del nodo
                              if (albero->figlioDestro(albero->binPadre(nd))==nd){
                                  albero->setFiglioDestro(albero->binPadre(nd),
sottoAlbero->binRadice());
                              }
                              if (albero->figlioSinistro(albero->binPadre(nd))==nd){
                                  albero->setFiglioSinistro(albero->binPadre(nd),
sottoAlbero->binRadice());
                              }
                              nd=sottoAlbero->binRadice();
                          }
                          else {//Se il nodo è la radice dell'Albero iniziale
                              //Aggiornamento del nodo
```

```
                            nd=sottoAlbero->binRadice();
                            albero->setRadice(nd);
                        }
                    }
                else {
                //2) (q AND r) OR p <= (q OR p) AND (r OR p)
                p1N = albero->figlioDestro(albero->figlioSinistro(nd));
                  string p1S=binVisitaRet(albero, p1N);
                  p2N = albero->figlioDestro(albero->figlioDestro(nd));
                  string p2S=binVisitaRet(albero, p2N);
                  //cout<<"P1 = "+p1S+"\nP2 = "+p2S);
                  if (p1S==p2S) {

                        //Memorizzazione di p
                        AlberoBinario *p=new AlberoBinario;
                        p->costrBinAlbero(NULL, NULL);
                        p->setRadice(p1N);

                        //Memorizzazione di q
                        AlberoBinario *q=new AlberoBinario;
                        q->costrBinAlbero(NULL, NULL);
                        q->setRadice(albero->figlioSinistro(albero-
>figlioSinistro(nd)));

                        //Memorizzazione di r
                        AlberoBinario *r=new AlberoBinario;
                        r->costrBinAlbero(NULL, NULL);
                        r->setRadice(albero->figlioSinistro(albero-
>figlioDestro(nd)));

                        //Costruzione del figlio Sinistro
                        AlberoBinario *alberoSinistro=new AlberoBinario;
                        alberoSinistro->costrBinAlbero(q, r);
                        //Assegna AND alla radice del figlio destro
                        alberoSinistro->scriviNodo(alberoSinistro->binRadice(),
"&", false);

                        //Costruzione del figlio Destro
                        AlberoBinario *alberoDestro=new AlberoBinario;
                        alberoDestro=p;

                        //Ricostruzione del SottoAlbero
                        AlberoBinario *sottoAlbero=new AlberoBinario;
                        sottoAlbero->costrBinAlbero(alberoSinistro, alberoDestro);
                        sottoAlbero->scriviNodo(sottoAlbero->binRadice(), "|",
false);

                        //cout<<"Nuovo SottoAlbero");
                        //BinServizio->binVisitaFin(sottoAlbero, sottoAlbero-
>binRadice());
                        //cout<<"\n");

                        if (albero->binPadre(nd)!=NULL) { //Se non è la radice
dell'albero iniziale
                            //Aggiorna il padre del nodo
                            if (albero->figlioDestro(albero->binPadre(nd))==nd){
                                albero->setFiglioDestro(albero->binPadre(nd),
sottoAlbero->binRadice());
                            }
```

```
                            if (albero->figlioSinistro(albero->binPadre(nd))==nd){
                                albero->setFiglioSinistro(albero->binPadre(nd),
sottoAlbero->binRadice());
                            }
                            nd=sottoAlbero->binRadice();
                        }
                        else {//Se il nodo è la radice dell'Albero iniziale
                            //Aggiornamento del nodo
                            nd=sottoAlbero->binRadice();
                            albero->setRadice(nd);
                        }
                    }
                        }
            }
            }
            else if (albero->leggiNodo(nd)=="|") {
            // 3) p AND (q OR r) <= (p AND q) OR (p AND q)
            if (albero->leggiNodo(albero->figlioDestro(nd))=="&" && albero-
>leggiNodo(albero->figlioSinistro(nd))=="&"){
                Nodo *p1N = albero->figlioSinistro(albero->figlioSinistro(nd));
                string p1S=binVisitaRet(albero, p1N);
                Nodo *p2N = albero->figlioSinistro(albero->figlioDestro(nd));
                string p2S=binVisitaRet(albero, p2N);
                //cout<<"P1 = "+p1S+"\nP2 = "+p2S);
                if (p1S==p2S) {

                        //Memorizzazione di p
                        AlberoBinario *p=new AlberoBinario;
                        p->costrBinAlbero(NULL, NULL);
                        p->setRadice(p1N);

                        //Memorizzazione di q
                        AlberoBinario *q=new AlberoBinario;
                        q->costrBinAlbero(NULL, NULL);
                        q->setRadice(albero->figlioDestro(albero-
>figlioSinistro(nd)));

                        //Memorizzazione di r
                        AlberoBinario *r=new AlberoBinario;
                        r->costrBinAlbero(NULL, NULL);
                        r->setRadice(albero->figlioDestro(albero-
>figlioDestro(nd)));

                        //Costruzione del figlio Sinistro
                        AlberoBinario *alberoSinistro=new AlberoBinario;
                        alberoSinistro=p;

                        //Costruzione del figlio Destro
                        AlberoBinario *alberoDestro=new AlberoBinario;
                        alberoDestro->costrBinAlbero(q, r);
                        //Assegna OR alla radice del figlio destro
                        alberoDestro->scriviNodo(alberoDestro->binRadice(), "|",
false);

                        //Ricostruzione del SottoAlbero
                        AlberoBinario *sottoAlbero=new AlberoBinario;
                        sottoAlbero->costrBinAlbero(alberoSinistro, alberoDestro);
                        sottoAlbero->scriviNodo(sottoAlbero->binRadice(), "&",
false);
```

```
                        //cout<<"Nuovo SottoAlbero");
                        //BinServizio->binVisitaFin(sottoAlbero, sottoAlbero-
>binRadice());
                        //cout<<"\n");

                        if (albero->binPadre(nd)!=NULL) { //Se non è la radice
dell'albero iniziale
                            //Aggiorna il padre del nodo
                            if (albero->figlioDestro(albero->binPadre(nd))==nd){
                                albero->setFiglioDestro(albero->binPadre(nd),
sottoAlbero->binRadice());
                            }
                            if (albero->figlioSinistro(albero->binPadre(nd))==nd){
                                albero->setFiglioSinistro(albero->binPadre(nd),
sottoAlbero->binRadice());
                            }
                            nd=sottoAlbero->binRadice();
                        }
                        else {//Se il nodo è la radice dell'Albero iniziale
                            //Aggiornamento del nodo
                            nd=sottoAlbero->binRadice();
                            albero->setRadice(nd);
                        }
                    }
                else {
                //4) (q or r) AND p <= (q AND p) OR (r AND p)
                p1N = albero->figlioDestro(albero->figlioSinistro(nd));
                  string p1S=binVisitaRet(albero, p1N);
                  p2N = albero->figlioDestro(albero->figlioDestro(nd));
                  string p2S=binVisitaRet(albero, p2N);
                  //cout<<"P1 = "+p1S+"\nP2 = "+p2S);
                  if (p1S==p2S) {

                        //Memorizzazione di p
                        AlberoBinario *p=new AlberoBinario;
                        p->costrBinAlbero(NULL, NULL);
                        p->setRadice(p1N);

                        //Memorizzazione di q
                        AlberoBinario *q=new AlberoBinario;
                        q->costrBinAlbero(NULL, NULL);
                        q->setRadice(albero->figlioSinistro(albero-
>figlioSinistro(nd)));

                        //Memorizzazione di r
                        AlberoBinario *r=new AlberoBinario;
                        r->costrBinAlbero(NULL, NULL);
                        r->setRadice(albero->figlioSinistro(albero-
>figlioDestro(nd)));

                        //Costruzione del figlio Sinistro
                        AlberoBinario *alberoSinistro=new AlberoBinario;
                        alberoSinistro->costrBinAlbero(q, r);
                        //Assegna AND alla radice del figlio destro
                        alberoSinistro->scriviNodo(alberoSinistro->binRadice(),
"|", false);

                        //Costruzione del figlio Destro
```

```
                        AlberoBinario *alberoDestro=new AlberoBinario;
                        alberoDestro=p;

                        //Ricostruzione del SottoAlbero
                        AlberoBinario *sottoAlbero=new AlberoBinario;
                        sottoAlbero->costrBinAlbero(alberoSinistro, alberoDestro);
                        sottoAlbero->scriviNodo(sottoAlbero->binRadice(), "&",
false);

                        //cout<<"Nuovo SottoAlbero");
                        //BinServizio->binVisitaFin(sottoAlbero, sottoAlbero-
>binRadice());
                        //cout<<"\n");

                        if (albero->binPadre(nd)!=NULL) { //Se non è la radice
dell'albero iniziale
                            //Aggiorna il padre del nodo
                            if (albero->figlioDestro(albero->binPadre(nd))==nd){
                                albero->setFiglioDestro(albero->binPadre(nd),
sottoAlbero->binRadice());
                            }
                            if (albero->figlioSinistro(albero->binPadre(nd))==nd){
                                albero->setFiglioSinistro(albero->binPadre(nd),
sottoAlbero->binRadice());
                            }
                            nd=sottoAlbero->binRadice();
                        }
                        else {//Se il nodo è la radice dell'Albero iniziale
                            //Aggiornamento del nodo
                            nd=sottoAlbero->binRadice();
                            albero->setRadice(nd);
                        }
                    }
                        }
                }
            }
            }
            else
                  cout<<"Nessuna Modifica effettuata";
    }
```

Nodo.h

```
/*
 * Nodo.h
 *
 * @author Andrea Maiellaro & Domenico Pio Novelli
 * Data Creazione: 18 dicembre 2005
 * Data Ultima Modifica: 18 Dicembre 2005
 *
 */

#ifndef _NODO_H
#define _NODO_H

#include <string>
```

```
#include <iostream>

using namespace std;

class Nodo {
    private:
           Nodo* figlioDestro;
           Nodo* figlioSinistro;
           Nodo* genitore;
           string etichetta;
           bool flagNot;
    public:
           Nodo();
           //~Nodo();
           Nodo* getFiglioDestro() const;
           Nodo* getFiglioSinistro() const;
           Nodo* getGenitore() const;
           string getEtichetta() const;
           bool getNot() const;
           void setFiglioDestro(Nodo *fd);
           void setFiglioSinistro(Nodo *fs);
           void setGenitore(Nodo *gn);
           void setEtichetta(string etc);
           void setNot(bool bn);
};

    /** Creates a new instance of Nodo */
    Nodo::Nodo() {
           figlioDestro=NULL;
           figlioSinistro=NULL;
           genitore=NULL;
           etichetta="";
           flagNot=false;
    }

//    Nodo::~Nodo(){}

    Nodo* Nodo::getFiglioDestro() const{
        return figlioDestro;
    }

    Nodo* Nodo::getFiglioSinistro() const{
        return figlioSinistro;
    }

    Nodo* Nodo::getGenitore() const{
        return genitore;
    }

    string Nodo::getEtichetta() const{
        return etichetta;
    }

    bool Nodo::getNot() const{
        return flagNot;
    }

    void Nodo::setFiglioDestro(Nodo *fd){
        figlioDestro = fd;
```

```
    }

    void Nodo::setFiglioSinistro(Nodo *fs){
        figlioSinistro = fs;
    }

    void Nodo::setGenitore(Nodo *gn){
        genitore = gn;
    }

    void Nodo::setEtichetta(string etc){
        etichetta = etc;
    }

    void Nodo::setNot(bool bn){
        flagNot=bn;
    }

#endif
```

AlberoBinario.h

```
/*
 * AlberoBinario.h
 *
 * @author Andrea Maiellaro & Domenico Pio Novelli
 * Data Creazione: 19 dicembre 2005
 * Data Ultima Modifica: 20 Dicembre 2005
 *
 */

#ifndef _ALBBIN_H
#define _ALBBIN_H

#include "Nodo.h"

class AlberoBinario {

    private:
        Nodo *radiceAlbero;
    public:
        AlberoBinario::AlberoBinario();
        bool AlberoBinario::binAlberoVuoto() const;
        Nodo* binRadice() const;
        Nodo* binPadre(Nodo *nd) const;
        Nodo* figlioSinistro (Nodo *nd) const;
        Nodo* figlioDestro (Nodo *nd) const;
        bool AlberoBinario::sinistroVuoto (Nodo *nd) const;
        bool destroVuoto (Nodo *nd) const;
        void cancSottoBinAlbero (Nodo* nd);
        string leggiNodo (Nodo* nd) const;
        void scriviNodo (Nodo *nd, string etc, bool flagNot);
        void setRadice(Nodo* nd);
        void setFiglioDestro(Nodo *nd, Nodo *nuovo);
        void setFiglioSinistro(Nodo *nd, Nodo *nuovo);
        void setNot(Nodo *nd, bool flag);
        bool getNot(Nodo *nd) const;
```

```
        void costrBinAlbero (AlberoBinario *a1, AlberoBinario *a2);
};

    /* Seguendo le specifiche:
     *
     * public AlberoBinario(){
     *      creaBinAlbero()
     * }
     *
     * private creaBinAlbero(){}
     *
     * SPRECO DI MEMORIA Stack: Due chiamate a metodi che non fanno nulla
     */

    /**
     * Corrispondente a CreaBinAlbero delle specifiche
     */
    AlberoBinario::AlberoBinario() {
           radiceAlbero=NULL;
    }

    /**
     * @return boolean
     * POST: b=VERO SE Albero Vuoto
     *       b=FALSO ALTRIMENTI
     */
    bool AlberoBinario::binAlberoVuoto() const{
        //POST: b=VERO SE Albero Vuoto
        if (radiceAlbero==NULL)
            return true;
        // POST: b=FALSO ALTRIMENTI
        else
            return false;
    }

    /**
     * @return Nodo Radice dell'Albero
     * PRE: Albero non vuoto
     * POST: RADICE DI T LIVELLO(u) = 0
     */
    Nodo* AlberoBinario::binRadice() const{
        //Pre condizione non inserita: se l'albero è vuoto restituisce null
        return radiceAlbero;
    }

    /**
     * PRE: T non vuoto, nd non appartenente a N, LIVELLO(nd) > 0
     * POST: v E' PADRE DI nd -> (v,nd) ? A -> LIVELLO(nd)=LIVELLO(nd)+1
     */
    Nodo* AlberoBinario::binPadre(Nodo *nd) const{
        //Pre condizione non inserita:
      //se l'albero è vuoto o il nodo passato è la radice restituisce null
        return nd->getGenitore();
    }

    Nodo* AlberoBinario::figlioSinistro (Nodo *nd) const{
        //Pre condizione non inserita:
      //se il nodo passato non ha figlio sinistro restituisce null
        return nd->getFiglioSinistro();
```

```
}

Nodo* AlberoBinario::figlioDestro (Nodo *nd) const{
    //Pre condizione non inserita:
  //se il nodo passato non ha figlio destro restituisce null
    return nd->getFiglioDestro();
}

bool AlberoBinario::sinistroVuoto (Nodo *nd) const{
    //Pre condizione non inserita
    if (nd->getFiglioSinistro()==NULL)
        return true;
    else
        return false;
}

bool AlberoBinario::destroVuoto (Nodo *nd) const{
    //Pre condizione non inserita
    if (nd->getFiglioDestro()==NULL)
        return true;
    else
        return false;
}

void AlberoBinario::cancSottoBinAlbero (Nodo* nd){
    //Pre condizione: albero non vuoto
    if (radiceAlbero!=NULL){

        // Cancellazzione ricorsiva dei sottoalberi
            if (nd->getFiglioSinistro()!=NULL)
                cancSottoBinAlbero (nd->getFiglioSinistro());
            if (nd->getFiglioDestro()!=NULL)
                cancSottoBinAlbero (nd->getFiglioDestro());

        // Se è la radice dell'albero
        if (nd->getGenitore()==NULL)
            radiceAlbero=NULL;
        else
        {
            Nodo *padre=new Nodo;
            padre = nd->getGenitore();
            if (nd->getGenitore()->getFiglioSinistro()==nd){
                padre->setFiglioSinistro(NULL);
            }
            else
                padre->setFiglioDestro(NULL);

        }
        delete(nd);
     }
}

string AlberoBinario::leggiNodo (Nodo* nd) const{
    //Pre condizione non inserita
    string str=nd->getEtichetta();
    if (nd->getNot())
        str="!"+str;
    return str;
}
```

```
    void AlberoBinario::scriviNodo (Nodo *nd, string etc, bool flagNot){
        //Pre condizione non inserita
        nd->setEtichetta(etc);
        nd->setNot(flagNot);
    }

    void AlberoBinario::setRadice(Nodo* nd){
         //Non è un InsRadice
        radiceAlbero=nd;
    }

    void AlberoBinario::setFiglioDestro(Nodo *nd, Nodo *nuovo){
         //Non è un InsFiglioDestro
        nd->setFiglioDestro(nuovo);
    }

    void AlberoBinario::setFiglioSinistro(Nodo *nd, Nodo *nuovo){
         //Non è un InsFiglioSinistro
        nd->setFiglioSinistro(nuovo);
    }

    //aggiunta a scriviNodo
    void AlberoBinario::setNot(Nodo *nd, bool flag){
        nd->setNot(flag);
    }

    //aggiunta a LeggiNodo
    bool AlberoBinario::getNot(Nodo *nd) const{
        return nd->getNot();
    }

    void AlberoBinario::costrBinAlbero (AlberoBinario *a1, AlberoBinario *a2){
        Nodo *radice = new Nodo();
        if (a1!=NULL){
            radice->setFiglioSinistro(a1->binRadice());
            a1->binRadice()->setGenitore(radice);
        }
        if (a2!=NULL){
            radice->setFiglioDestro(a2->binRadice());
            a2->binRadice()->setGenitore(radice);
        }
        radiceAlbero=radice;
    }


#endif
```

BinServizio.h

```
/*
 * BinServizio.java
 *
 * @author Andrea Maiellaro & Domenico Pio Novelli
 * Data Creazione: 19 dicembre 2005
 * Data Ultima Modifica: 19 Dicembre 2005
```

```
 *
 */


#ifndef _BINSERV_H
#define _BINSERV_H

#include "AlberoBinario.h"
#include "Nodo.h"
#include <string>

void binVisitaFin (AlberoBinario *aB, Nodo *nd);
string binVisitaRet (AlberoBinario *aB, Nodo *nd);

void binVisitaFin(AlberoBinario *aB, Nodo *nd){

        if (nd!=NULL){

            if (nd->getEtichetta()=="&" || nd->getEtichetta()=="|" ){
                if (nd->getNot())
                    cout<<"!(";
                else
                    cout<<"(";
            }

            if (!aB->sinistroVuoto(nd)){
                binVisitaFin (aB, nd->getFiglioSinistro());
            }

            if (nd->getEtichetta()=="&" || nd->getEtichetta()=="|" )
                    cout<<nd->getEtichetta();
            else
                if (nd->getNot())
                    cout<<"!"+nd->getEtichetta();
                else
                    cout<<nd->getEtichetta();

            if (!aB->destroVuoto(nd)){
                binVisitaFin (aB, nd->getFiglioDestro());
            }

            if (nd->getEtichetta()=="&" || nd->getEtichetta()=="|" )
                cout<<")";
        }
    }

    string binVisitaRet(AlberoBinario *aB, Nodo *nd){

        string str="";

        if (nd!=NULL){

            if (nd->getEtichetta()=="&" || nd->getEtichetta()=="|" ){
                if (nd->getNot())
                    str=str+"!(";
                else
                    str=str+"(";
            }
```

```
            if (!aB->sinistroVuoto(nd)){
               str=str+binVisitaRet (aB, nd->getFiglioSinistro());
            }

            if (nd->getEtichetta()=="&" || nd->getEtichetta()=="|" )
                 str=str+nd->getEtichetta();
            else
                 if (nd->getNot())
                     str=str+"!"+nd->getEtichetta();
                 else
                     str=str+nd->getEtichetta();

            if (!aB->destroVuoto(nd)){
               str=str+binVisitaRet (aB, nd->getFiglioDestro());
            }

            if (nd->getEtichetta()=="&" || nd->getEtichetta()=="|" )
                 str=str+")";
        }
        return str;
    }


#endif
```

NodoC.h

```
/*
 * NodoC.h
 *
 * @author Andrea Maiellaro & Domenico Pio Novelli
 * Data Creazione: 19 dicembre 2005
 * Data Ultima Modifica: 19 Dicembre 2005
 *
 */

#ifndef _NODOC_H
#define _NODOC_H

#include "AlberoBinario.h"

class NodoC {
    private:
            Nodo *elemento;
            NodoC *succ;

    public:
           NodoC();
           void setElemento(Nodo *nd);
           Nodo* getElemento() const;
           void setSucc(NodoC *nd);
           NodoC* getSucc() const;

};

    /** Creates a new instance of NodoC */
    NodoC::NodoC() {
```

```
        elemento=NULL;
        succ=NULL;

    }

    void NodoC::setElemento(Nodo *nd){
        elemento=nd;
    }

    Nodo* NodoC::getElemento() const{
        return elemento;
    }

    void NodoC::setSucc(NodoC *nd){
        succ=nd;
    }

    NodoC* NodoC::getSucc() const{
        return succ;
    }


#endif
```

Coda.h

```
/*
 * Coda.h
 *
 * @author Andrea Maiellaro & Domenico Pio Novelli
 * Data Creazione: 19 dicembre 2005
 * Data Ultima Modifica: 19 Dicembre 2005
 *
 */

#ifndef _CODA_H
#define _CODA_H

#include  "AlberoBinario.h"
#include  "NodoC.h"

class Coda {
        public:
        Coda();
           bool codaVuota() const;
           void inCoda(Nodo *nd);
           void fuoriCoda();
           Nodo* leggiCoda() const;
        private:
           NodoC *testa;
           NodoC *fondo;
           };


    /** Creates a new instance of Coda */
    Coda::Coda() {
        testa=NULL;
```

```
        fondo=NULL;
    }

    bool Coda::codaVuota() const {
        if (testa==NULL)
            return true;
        else
            return false;
    }

    Nodo* Coda::leggiCoda() const{
        return testa->getElemento();
    }

     void Coda::fuoriCoda(){
        testa=testa->getSucc();
    }

    void Coda::inCoda (Nodo *nd){
        NodoC *nuovo=new NodoC;
        nuovo->setElemento(nd);
        if (testa==NULL)
                testa=nuovo;
        else
            fondo->setSucc(nuovo);
        fondo=nuovo;
    }

#endif
```

4.2 Codifica Java

Main.java

```
  1 /*
  2  * Main.java
  3  *
  4  * @author Andrea Maiellaro & Domenico Pio Novelli
  5  * Data Creazione: 13 dicembre 2005
  6  * Data Ultima Modifica: 20 Dicembre 2005
  7  *
  8  */
  9
 10 package logicaltree;
 11
 12 import java.io.*;
 13 import AlberoBinario.*;
 14 import Coda.*;
 15
 16 public class Main {
 17
 18     /**
 19      * @param args the command line arguments
 20      */
 21     public static void main (String args[]) throws IOException{
 22
 23         //Apertura Buffer per Input da Tastiera
 24         BufferedReader stdin = new BufferedReader (new InputStreamReader
(System.in));
 25
 26         AlberoBinario logicalTree=null;
 27         String str=null;
 28
 29         System.out.println("\nSOFTWARE ACQUISIZIONE E GESTIONE ESPRESSIONI
BOOLEANE\n");
 30
 31         System.out.println("Il metodo accetta espressioni del tipo"
 32                         +"\n\t(x&y) oppure (x|y)"
 33                         + "\ndove x e y possono a loro volta essere complesse"
 34                                 + "\ne a cui puo" essere legato anche
l"operatore ! (NOT)"
 35                                 + "\nEsempio di espressioni valide: (A&B)
oppure (!a&(B&!C))");
 36
 37         //Ripeti fintantochè stringa non valida
 38         do{
 39
 40             //Ripeti se stringa vuota
 41             do{
 42                                 System.out.print("\nInserire Espressione
Booleana: ");
 43
 44                 //Acquisizione della stringa da Tastiera
 45                 str=stdin.readLine().trim();
 46
 47                 if (str.equals("")){
```

```
48                      System.out.println ("\nStringa vuota");
49                  }
50              }while (str.equals(""));
51
52                  //Eliminazione degli spazi all"interno della Stringa
53                  while (str.indexOf(" ")!=-1){
54                      int spazioP=str.indexOf(" ");
55                      str=(str.substring(0, spazioP))+str.substring(spazioP+1);
56                      //System.out.println ("Stringa senza spazio: "+str);
57                  }
58
59              System.out.println("\nStringa acquisita: "+str);
60
61              //Trasformazione della Stringa in Albero Binario
62              logicalTree = creaLogicalTree(str.substring(1));
63
64              //Stampa in profondità
65              if (logicalTree!=null) {
66                  System.out.println("Albero Iniziale:");
67                  //visualizzaAmpiezza(logicalTree);
68                  BinServizio.binVisitaFin(logicalTree, logicalTree.binRadice());
69                  System.out.println();
70              }
71              else {
72                  System.out.println("Stringa Errata");
73              }
74          }while (logicalTree==null);
75
76          modificaAlbero (logicalTree);
77
78          //Stampa in profondità
79          System.out.println("\nAlbero Finale:");
80          BinServizio.binVisitaFin(logicalTree, logicalTree.binRadice());
81          System.out.println();
82
83      }
84
85      /**
86       * Metodo di acquisizione della Stringa e di creazione dell"Albero Binario
87       *
88       * Il metodo accetta stringhe del tipo
89       *          (x&y) oppure (x|y)
90       * dove x e y possono a loro volta essere complesse
91       * e a cui può essere legato anche l"operatore !
92       *
93       * @param str Stringa da esaminare
94       * @return Albero Binario della Stringa
95       */
96      private static AlberoBinario creaLogicalTree(String str){
97
98          try {
99              //System.out.println("Entrato nella funzione con la stringa:
"+str);
100
101             //Albero Padre
102             AlberoBinario logicalTree=new AlberoBinario();
103
```

```
104             //Alberi: Figlio Sinistro e Figlio Destro
105             AlberoBinario logicalTreeLeft=new AlberoBinario(),
logicalTreeRight=new AlberoBinario();
106
107             //Operatore
108             String operatore=null;
109
110             //Stringa Primo Operando
111             String str1=null;
112
113             //Stringa Secondo Operando
114             String str2=null;
115
116             //Posizione Parentesi Tonda Aperta
117             int parTonApB = str.indexOf("(");
118             int parTonChB = str.lastIndexOf(")");
119
120             //Varibaili temporanea di scorrimento nella Stringa
121             int i=0, j=0;
122
123     /* INIZIO AQUISIZIONE DEL PRIMO OPERANDO / FIGLIO SINISTRO */
124
125             // Se trova una Parentesi Tonda Aperta all"inizio
126             if (parTonApB==0){
127                     logicalTreeLeft=creaLogicalTree (str.substring(1));
128                 } // fine if (parTonApB==0)
129             else {
130                 str1=str; //Stringa temporanea
131
132                 //Scorri nella stringa
133                 while (str1.indexOf("&")!=0 && str1.indexOf("|")!=0){
134                     i++;
135                     // System.out.println("Posizione i= "+i);
136                     str1=str.substring (i);
137                 }
138
139                 //Stringa dell"operando
140                 str1=str.substring(0, i);
141
142                 // System.out.println("Operando di sinistra= "+str1);
143
144                 logicalTreeLeft=AlberoBinario.costrBinAlbero(null, null);
145
146                 if (str1.indexOf("!")==0) //Se c"è il NOT
147                     logicalTreeLeft.scriviNodo(logicalTreeLeft.binRadice(),
str1.substring(1), true);
148                 else
149                     logicalTreeLeft.scriviNodo(logicalTreeLeft.binRadice(), str1,
false);
150
151             } //Fine else if (parTonApB==0)
152
153     /* FINE AQUISIZIONE DEL PRIMO OPERANDO / FIGLIO SINISTRO */
154
155
156     /* INIZIO AQUISIZIONE DELL"OPERATORE */
157
```

```
158             if (i!=0) { //Se non è entrato nella chiamata ricorsiva dell"albero
sinistro
159
160                 //Acquisisci operatore
161                 operatore=str.substring(i, i+1);
162                 //System.out.println ("Operatore= "+operatore);
163                 str=str.substring(i);
164
165             }
166
167             else{
168             /* Se entrato nell"if con ricorsione, è necessario recuperare
169              * la posizione dell"operatore corrispondente
170              */
171
172                 //System.out.println("SONO QUI: "+str);
173
174                 String strk=str; //Stringa temporanea
175
176                 int k=0;
177
178                 //Ciclo per contare le parentesi tonde aperte iniziali
179                 while (strk.indexOf("(")==0){
180                     k++;
181                     strk=strk.substring(1);
182                     //System.out.println("Posizione k= "+k);
183                 }
184
185                 int h=0, l=0;
186                 String strL=str;
187
188                 while (h!=k){
189                     //Ogni parentesi aperta, corrisponde ad una tonda chiusa
190
191                     //Scorri nella stringa, fino a trovare la posizione
dell"operatore corrispondente
192                     while (strL.indexOf(")&")!=0 && strL.indexOf(")|")!=0){
193                         l++;
194                         strL=strL.substring(1);
195                         //System.out.println("Posizione L= "+l);
196                         //System.out.println("strL= "+strL);
197                     }
198
199                     h++; //Trovata una parentesi chiusa
200                     //System.out.println("Posizione H= "+h);
201                     strL=strL.substring(2); //Elimina
202                     l++; //Posizione eliminate
203                 }
204
205                 //System.out.println("Posizione dopo while di L= "+l);
206
207                 //La Stringa inizia dall"operatore corrispondente
208                 if (h>1)
209                     str=str.substring(l+h-1);
210                 else
211                     str=str.substring(l);
212                 //System.out.println("str dopo "+h+" while = "+str);
213
```

```
214                 //Acquisisci operatore
215                 operatore=str.substring(0, 1);
216                 //System.out.println ("Operatore= "+operatore);
217
218             } // fine if (parTonApB==0)
219
220     /* FINE  AQUISIZIONE DELL"OPERATORE */
221
222
223     /* INIZIO AQUISIZIONE DEL SECONDO OPERANDO / FIGLIO DESTRO */
224
225                 str=str.substring(1);
226                 parTonApB = str.indexOf("(");
227                 // System.out.println ("Stringa Restante= "+str);
228
229                 // Se trova una Parentesi Tonda Aperta all"inizio
230                 if (parTonApB==0){
231                         logicalTreeRight=creaLogicalTree (str.substring(1));
232                 } // fine if (parTonApB==0)
233                 else {
234                     str2=str; //Stringa temporanea
235
236                     //Scorri nella stringa
237                     while (str2.indexOf(")")!=0){
238                         j++;
239                         // System.out.println("Posizione j= "+j);
240                         str2=str.substring (j);
241                     }
242
243                     str2=str.substring(0, j);
244                     // System.out.println("Operando di destra= "+str2);
245
246                     logicalTreeRight=AlberoBinario.costrBinAlbero(null, null);
247                     if (str2.indexOf("!")==0) //Se c"è il NOT
248                         logicalTreeRight.scriviNodo(logicalTreeRight.binRadice(),
str2.substring(1), true);
249                     else
250                         logicalTreeRight.scriviNodo(logicalTreeRight.binRadice(),
str2, false);
251                 }
252
253     /* FINE AQUISIZIONE DEL SECONDO OPERANDO / FIGLIO DESTRO */
254
255     /* CREAZIONE DELL"ALBERO
256             /* Condizioni per una stringa valida:
257              * Albero Sinistro non vuoto
258              * Primo operatore lungo più di 0 caratteri
259              * Albero Destro non vuoto
260              * Secondo operatore lungo più di 0 caratteri
261              */
262
263             //Se la stringa non è valida
264             if (logicalTreeLeft!=null && logicalTreeRight!=null){
265                 logicalTree=AlberoBinario.costrBinAlbero(logicalTreeLeft,
logicalTreeRight);
266                 logicalTree.scriviNodo(logicalTree.binRadice(), operatore, false);
267
268                 /**  Prove della costruzione dell"Albero:
```

```
 269                 System.out.print("Radice Albero Intermedio Sinistro: \n");
 270 
System.out.println(logicalTreeLeft.leggiNodo(logicalTreeLeft.binRadice()));
 271                 System.out.print("Radice Albero Intermedio Destro: \n");
 272 
System.out.println(logicalTreeRight.leggiNodo(logicalTreeRight.binRadice()));
 273                 System.out.println("Operatore= "+operatore+"\n");
 274                 System.out.println("Albero Intermedio: \n");
 275                 BinServizio.binInVisita(logicalTree, logicalTree.binRadice());
 276                 **/
 277 
 278                 /** Prove della Costruzione dell"Albero
 279                 System.out.println("Albero Intermedio Sinistro:");
 280                 BinServizio.binVisitaFin(logicalTreeLeft,
logicalTreeLeft.binRadice());
 281                 System.out.println();
 282                 System.out.println("Albero Intermedio Destro:");
 283                 BinServizio.binVisitaFin(logicalTreeRight,
logicalTreeRight.binRadice());
 284                 System.out.println();
 285                 System.out.println("Operatore= "+operatore+"\n");
 286                 **/
 287 
 288                 return logicalTree;
 289             }
 290             else //Se la Stringa è valida
 291                 return null;
 292       }
 293       catch (StringIndexOutOfBoundsException errSt){
 294           return null;
 295       }
 296     }
 297 
 298     private static void visualizzaAmpiezza(AlberoBinario albero){
 299         System.out.println("\nVisualizzazione dell"Albero in ampiezza");
 300         System.out.print("Nodo 1= ");
 301         BinServizio.binVisitaFin(albero, albero.binRadice());
 302         System.out.println();
 303         Coda codaT = new Coda();
 304         codaT.inCoda(albero.binRadice());
 305         int i=2;
 306         while (!codaT.codaVuota()){
 307             //System.out.println("SONO QUI");
 308             Nodo N=codaT.leggiCoda();
 309             codaT.fuoriCoda();
 310             if ( !albero.sinistroVuoto(N) ){
 311                 if (albero.leggiNodo(albero.figlioSinistro(N)).equals("&") ||
albero.leggiNodo(albero.figlioSinistro(N)).equals("|") ||
albero.leggiNodo(albero.figlioSinistro(N)).equals("!&") ||
albero.leggiNodo(albero.figlioSinistro(N)).equals("!|")){
 312                     //System.out.println(""+i+"=
"+albero.leggiNodo(albero.figlioSinistro(N))+"\t");
 313                     System.out.print("\nNodo"+i+"= ");
 314                     BinServizio.binVisitaFin(albero, albero.figlioSinistro(N));
 315                     System.out.println();
 316                     codaT.inCoda(albero.figlioSinistro(N));
 317                     i++;
```

```
318                     }
319                 }
320                 if ( !albero.destroVuoto(N) ){
321                     if (albero.leggiNodo(albero.figlioDestro(N)).equals("&") ||
albero.leggiNodo(albero.figlioDestro(N)).equals("|") ||
albero.leggiNodo(albero.figlioDestro(N)).equals("!&") ||
albero.leggiNodo(albero.figlioDestro(N)).equals("!|")){
322                         //System.out.println(""+i+"=
"+albero.leggiNodo(albero.figlioDestro(N))+"\t");
323                         System.out.print("\nNodo"+i+"= ");
324                         BinServizio.binVisitaFin(albero, albero.figlioDestro(N));
325                         System.out.println();
326                         codaT.inCoda(albero.figlioDestro(N));
327                         i++;
328                     }
329                 }
330 
331             }
332         }
333 
334         /**
335          * Metodo per la modifica della Stringa da parte dell"utente
336          */
337         private static void modificaAlbero (AlberoBinario albero) throws IOException{
338 
339             int scelta=-1;
340 
341             do{
342 
343                 // Visualizzazione a Video della Stringa
344                 visualizzaAmpiezza(albero);
345 
346                 //Apertura Buffer per Input da Tastiera
347                 BufferedReader stdin = new BufferedReader (new InputStreamReader
(System.in));
348 
349                 try{
350 
351                     //Necessaria altrimenti rimane memorizzata la scelta
precedente
352                     scelta=-1;
353 
354                     //Leggere il nodo da modificare
355                                         System.out.print ("\nScegliere il Nodo da
modificare (0 per uscire): ");
356                                         scelta = Integer.parseInt(stdin.readLine());
357                 }
358                 catch (NumberFormatException StringaErrata){
359                                         //Digitazione di un carattere che non è un
numero
360                                         System.out.println("Inserito valore errato");
361                 }
362 
363                 //Nodo da Modificare
364                 //Nodo N = albero.figlioDestro(albero.binRadice());
365                 Nodo N = scegliNodo(albero, scelta);
366 
```

```
367             if (scelta!=0){
368             //Controllare se il numero del Nodo è valido
369             if (N!=null){
370
371                 int scelta2=-1;
372
373                 //Chiedere quale trasformazione applicare
374                 do{
375                     try{
376
377                         //Necessaria altrimenti rimane memorizzata la scelta
precedente
378                         scelta2=-1;
379
380                         //Leggere il nodo da modificare
381                         System.out.println ("Modifiche possibili: ");
382                         System.out.println ("\t1) Applicazione della Legge di
DeMorgan ");
383                         System.out.println ("\t2) Applicazione della
Distributiva ");
384                         System.out.println ("\t3) Applicazione della
Distributiva Inversa ");
385
386                         System.out.print ("Inserire scelta (0 per uscire):");
387                         scelta2 = Integer.parseInt(stdin.readLine());
388                     }
389                     catch (NumberFormatException StringaErrata){
390                             //Digitazione di un carattere che non è un
numero
391                         System.out.println("Inserito valore errato");
392                     }
393                 }while(scelta2<0 || scelta2>3);
394
395                 //Passare al relativo sottoproblema
396                 if (scelta2!=0){
397
398                     if (scelta2==1){
399                         //Applicazione di Demorgan al nodo N
400                         deMorgan (albero, N);
401                     }
402                     else if (scelta2==2){
403                         //Applicazione della Distributiva al nodo N
404                         distributiva(albero,N);
405                     }
406                     else if (scelta2==3)
407                     //Applicazione della Associativa al nodo N
408                         distributivaInv (albero, N);
409                     else {
410                             System.out.println("Inserito numero errato\n");
411                         }
412                 }
413
414                 } //if (N!=null)
415
416                 } // if (scelta!=0)
417
418             //System.out.println ("\nFINE CICLO\n");
```

```
419
420             } while (scelta!=0);
421
422         }
423
424          /**
425           * Applicazione della distributiva su un nodo:
426           * 1) p OR (q AND r) => (p OR q) AND (p OR q)
427           * 2) (q AND r) OR p => (q OR p) AND (r OR p)
428           * 3) p AND (q OR r) => (p AND q) OR (p AND q)
429           * 4) (q AND r) OR p => (q AND p) OR (r AND p)
430           *
431           * NON accetta stringhe con NOT legato all"operatore
432          **/
433         private static void distributiva (AlberoBinario albero, Nodo nd){
434
435             //System.out.println("Radice Nodo= "+albero.leggiNodo(nd));
436             //System.out.println("Figlio destro del Nodo=
"+albero.leggiNodo(albero.figlioDestro(nd)));
437
438             AlberoBinario alberoTemp=new AlberoBinario();
439
440             // 1) p OR (q AND r) => (p OR q) AND (p OR q)
441             if (albero.leggiNodo(nd).equals("|") &&
albero.leggiNodo(albero.figlioDestro(nd)).equals("&")){
442
443                 //Memorizzazione di p
444                 AlberoBinario p=new AlberoBinario();
445                 p=AlberoBinario.costrBinAlbero(null, null);
446                 p.setRadice(albero.figlioSinistro(nd));
447                 //p.binRadice()=albero.figlioSinistro(nd); NON FUNZIONA
448
449                 Nodo figlioDestro=albero.figlioDestro(nd);
450
451                 //Memorizzazione di q
452                 AlberoBinario q=new AlberoBinario();
453                 q=AlberoBinario.costrBinAlbero(null, null);
454                 q.setRadice(albero.figlioSinistro(figlioDestro));
455                 //q.binRadice()=albero.figlioSinistro(figlioDestro); NON FUNZIONA
456
457                 //Memorizzazione di r
458                 AlberoBinario r=new AlberoBinario();
459                 r=AlberoBinario.costrBinAlbero(null, null);
460                 r.setRadice(albero.figlioDestro(figlioDestro));
461                 //r.binRadice()=albero.figlioDestro(figlioDestro); NON FUNZIONA
462
463                 //Costruzione del figlio Sinistro
464                 AlberoBinario alberoSinistro=new AlberoBinario();
465                 alberoSinistro=AlberoBinario.costrBinAlbero(p, q);
466                 //Assegna OR alla radice del figlio sinistro
467                 alberoSinistro.scriviNodo(alberoSinistro.binRadice(), "|", false);
468
469                 //Costruzione del figlio Destro
470                 AlberoBinario alberoDestro=new AlberoBinario();
471                 alberoDestro=AlberoBinario.costrBinAlbero(p, r);
472                 //Assegna OR alla radice del figlio destro
473                 alberoDestro.scriviNodo(alberoDestro.binRadice(), "|", false);
```

```
474
475             //Ricostruzione del SottoAlbero
476             AlberoBinario sottoAlbero=new AlberoBinario();
477             sottoAlbero=AlberoBinario.costrBinAlbero(alberoSinistro,
alberoDestro);
478             sottoAlbero.scriviNodo(sottoAlbero.binRadice(), "&", false);
479
480             //System.out.println("Nuovo SottoAlbero");
481             //BinServizio.binVisitaFin(sottoAlbero, sottoAlbero.binRadice());
482             //System.out.println("\n");
483
484             if (albero.binPadre(nd)!=null) { //Se non è la radice dell"albero
iniziale
485                 //Aggiorna il padre del nodo
486                 if (albero.figlioDestro(albero.binPadre(nd))==nd){
487                     albero.setFiglioDestro(albero.binPadre(nd),
sottoAlbero.binRadice());
488                 }
489                 if (albero.figlioSinistro(albero.binPadre(nd))==nd){
490                     albero.setFiglioSinistro(albero.binPadre(nd),
sottoAlbero.binRadice());
491                 }
492                 nd=sottoAlbero.binRadice();
493             }
494             else {//Se il nodo è la radice dell"Albero iniziale
495                 //Aggiornamento del nodo
496                 nd=sottoAlbero.binRadice();
497                 albero.setRadice(nd);
498             }
499         }
500
501         // 2) (q AND r) OR p => (q OR p) AND (r OR p)
502         else if (albero.leggiNodo(nd).equals("|") &&
albero.leggiNodo(albero.figlioSinistro(nd)).equals("&")){
503
504             //Memorizzazione di p
505             AlberoBinario p=new AlberoBinario();
506             p=AlberoBinario.costrBinAlbero(null, null);
507             p.setRadice(albero.figlioDestro(nd));
508
509             Nodo figlioSinistro=albero.figlioSinistro(nd);
510
511             //Memorizzazione di q
512             AlberoBinario q=new AlberoBinario();
513             q=AlberoBinario.costrBinAlbero(null, null);
514             q.setRadice(albero.figlioSinistro(figlioSinistro));
515
516             //Memorizzazione di r
517             AlberoBinario r=new AlberoBinario();
518             r=AlberoBinario.costrBinAlbero(null, null);
519             r.setRadice(albero.figlioDestro(figlioSinistro));
520
521             //Costruzione del figlio Sinistro
522             AlberoBinario alberoSinistro=new AlberoBinario();
523             alberoSinistro=AlberoBinario.costrBinAlbero(q, p);
524             //Assegna OR alla radice del figlio sinistro
525             alberoSinistro.scriviNodo(alberoSinistro.binRadice(), "|", false);
```

```
526
527             //Costruzione del figlio Destro
528             AlberoBinario alberoDestro=new AlberoBinario();
529             alberoDestro=AlberoBinario.costrBinAlbero(r, p);
530             //Assegna OR alla radice del figlio destro
531             alberoDestro.scriviNodo(alberoDestro.binRadice(), "|", false);
532
533             //Ricostruzione del SottoAlbero
534             AlberoBinario sottoAlbero=new AlberoBinario();
535             sottoAlbero=AlberoBinario.costrBinAlbero(alberoSinistro,
alberoDestro);
536             sottoAlbero.scriviNodo(sottoAlbero.binRadice(), "&", false);
537
538             //System.out.println("Nuovo SottoAlbero");
539             //BinServizio.binVisitaFin(sottoAlbero, sottoAlbero.binRadice());
540             //System.out.println("\n");
541
542             if (albero.binPadre(nd)!=null) { //Se non è la radice dell"albero
iniziale
543                 //Aggiorna il padre del nodo
544                 if (albero.figlioDestro(albero.binPadre(nd))==nd){
545                     albero.setFiglioDestro(albero.binPadre(nd),
sottoAlbero.binRadice());
546                 }
547                 if (albero.figlioSinistro(albero.binPadre(nd))==nd){
548                     albero.setFiglioSinistro(albero.binPadre(nd),
sottoAlbero.binRadice());
549                 }
550                 nd=sottoAlbero.binRadice();
551             }
552             else {//Se il nodo è la radice dell"Albero iniziale
553                 //Aggiornamento del nodo
554                 nd=sottoAlbero.binRadice();
555                 albero.setRadice(nd);
556             }
557         }
558
559         // 3) p AND (q OR r) => (p AND q) OR (p AND q)
560         else if (albero.leggiNodo(nd).equals("&") &&
albero.leggiNodo(albero.figlioDestro(nd)).equals("|")){
561
562             //Memorizzazione di p
563             AlberoBinario p=new AlberoBinario();
564             p=AlberoBinario.costrBinAlbero(null, null);
565             p.setRadice(albero.figlioSinistro(nd));
566
567             Nodo figlioDestro=albero.figlioDestro(nd);
568
569             //Memorizzazione di q
570             AlberoBinario q=new AlberoBinario();
571             q=AlberoBinario.costrBinAlbero(null, null);
572             q.setRadice(albero.figlioSinistro(figlioDestro));
573
574             //Memorizzazione di r
575             AlberoBinario r=new AlberoBinario();
576             r=AlberoBinario.costrBinAlbero(null, null);
577             r.setRadice(albero.figlioDestro(figlioDestro));
```

```
578
579             //Costruzione del figlio Sinistro
580             AlberoBinario alberoSinistro=new AlberoBinario();
581             alberoSinistro=AlberoBinario.costrBinAlbero(p, q);
582             //Assegna OR alla radice del figlio sinistro
583             alberoSinistro.scriviNodo(alberoSinistro.binRadice(), "&", false);
584
585             //Costruzione del figlio Destro
586             AlberoBinario alberoDestro=new AlberoBinario();
587             alberoDestro=AlberoBinario.costrBinAlbero(p, r);
588             //Assegna OR alla radice del figlio destro
589             alberoDestro.scriviNodo(alberoDestro.binRadice(), "&", false);
590
591             //Ricostruzione del SottoAlbero
592             AlberoBinario sottoAlbero=new AlberoBinario();
593             sottoAlbero=AlberoBinario.costrBinAlbero(alberoSinistro,
alberoDestro);
594             sottoAlbero.scriviNodo(sottoAlbero.binRadice(), "|", false);
595
596             //System.out.println("Nuovo SottoAlbero");
597             //BinServizio.binVisitaFin(sottoAlbero, sottoAlbero.binRadice());
598             //System.out.println("\n");
599
600             if (albero.binPadre(nd)!=null) { //Se non è la radice dell"albero
iniziale
601                 //Aggiorna il padre del nodo
602                 if (albero.figlioDestro(albero.binPadre(nd))==nd){
603                     albero.setFiglioDestro(albero.binPadre(nd),
sottoAlbero.binRadice());
604                 }
605                 if (albero.figlioSinistro(albero.binPadre(nd))==nd){
606                     albero.setFiglioSinistro(albero.binPadre(nd),
sottoAlbero.binRadice());
607                 }
608                 nd=sottoAlbero.binRadice();
609             }
610             else {//Se il nodo è la radice dell"Albero iniziale
611                 //Aggiornamento del nodo
612                 nd=sottoAlbero.binRadice();
613                 albero.setRadice(nd);
614             }
615         }
616
617         // 4) (q AND r) OR p => (q AND p) OR (r AND p)
618         else if (albero.leggiNodo(nd).equals("&") &&
albero.leggiNodo(albero.figlioSinistro(nd)).equals("|")){
619
620             //Memorizzazione di p
621             AlberoBinario p=new AlberoBinario();
622             p=AlberoBinario.costrBinAlbero(null, null);
623             p.setRadice(albero.figlioDestro(nd));
624
625             Nodo figlioSinistro=albero.figlioSinistro(nd);
626
627             //Memorizzazione di q
628             AlberoBinario q=new AlberoBinario();
629             q=AlberoBinario.costrBinAlbero(null, null);
```

```
630                 q.setRadice(albero.figlioSinistro(figlioSinistro));
631
632                 //Memorizzazione di r
633                 AlberoBinario r=new AlberoBinario();
634                 r=AlberoBinario.costrBinAlbero(null, null);
635                 r.setRadice(albero.figlioDestro(figlioSinistro));
636
637                 //Costruzione del figlio Sinistro
638                 AlberoBinario alberoSinistro=new AlberoBinario();
639                 alberoSinistro=AlberoBinario.costrBinAlbero(q, p);
640                 //Assegna OR alla radice del figlio sinistro
641                 alberoSinistro.scriviNodo(alberoSinistro.binRadice(), "&", false);
642
643                 //Costruzione del figlio Destro
644                 AlberoBinario alberoDestro=new AlberoBinario();
645                 alberoDestro=AlberoBinario.costrBinAlbero(r, p);
646                 //Assegna OR alla radice del figlio destro
647                 alberoDestro.scriviNodo(alberoDestro.binRadice(), "&", false);
648
649                 //Ricostruzione del SottoAlbero
650                 AlberoBinario sottoAlbero=new AlberoBinario();
651                 sottoAlbero=AlberoBinario.costrBinAlbero(alberoSinistro,
alberoDestro);
652                 sottoAlbero.scriviNodo(sottoAlbero.binRadice(), "|", false);
653
654                 //System.out.println("Nuovo SottoAlbero");
655                 //BinServizio.binVisitaFin(sottoAlbero, sottoAlbero.binRadice());
656                 //System.out.println("\n");
657
658                 if (albero.binPadre(nd)!=null) { //Se non è la radice dell"albero
iniziale
659                     //Aggiorna il padre del nodo
660                     if (albero.figlioDestro(albero.binPadre(nd))==nd){
661                         albero.setFiglioDestro(albero.binPadre(nd),
sottoAlbero.binRadice());
662                     }
663                     if (albero.figlioSinistro(albero.binPadre(nd))==nd){
664                         albero.setFiglioSinistro(albero.binPadre(nd),
sottoAlbero.binRadice());
665                     }
666                     nd=sottoAlbero.binRadice();
667                 }
668                 else {//Se il nodo è la radice dell"Albero inizial(e
669                     //Aggiornamento del nodo
670                     nd=sottoAlbero.binRadice();
671                     albero.setRadice(nd);
672                 }
673             }
674
675
676             // Se non viene effettuata nessuna modifica
677             else{
678                 System.out.println("Nessuna modifica effettuata");
679             }
680
681             //System.out.println("Nuovo Albero");
682             //BinServizio.binVisitaFin(albero, albero.binRadice());
```

```
683           //System.out.println("\n");
684
685       }
686
687
688       private static void deMorgan (AlberoBinario albero, Nodo nd){
689           //Se (Nodo=="OR")
690           if (albero.leggiNodo(nd).equals("|")){
691               //Nodo = NOT AND
692               albero.scriviNodo(nd, "&", true);
693               System.out.println("Nuovo Nodo = "+albero.leggiNodo(nd));
694           }
695
696           //Se (Nodo=="NOT OR")
697           else if (albero.leggiNodo(nd).equals("!|")){
698               //Nodo = AND
699               albero.scriviNodo(nd, "&", false);
700               System.out.println("Nuovo Nodo = "+albero.leggiNodo(nd));
701           }
702
703           //Se (Nodo=="AND")
704           else if (albero.leggiNodo(nd).equals("&")){
705               //Nodo = NOT OR
706               albero.scriviNodo(nd, "|", true);
707               System.out.println("Nuovo Nodo = "+albero.leggiNodo(nd));
708           }
709
710           //Se (Nodo=="NOT AND")
711           else if (albero.leggiNodo(nd).equals("!&")){
712               //Nodo = OR
713               albero.scriviNodo(nd, "|", false);
714               System.out.println("Nuovo Nodo = "+albero.leggiNodo(nd));
715           }
716           //FiglioSinistro del Nodo = ! FiglioSinistro del Nodo
717           albero.setNot(albero.figlioSinistro(nd), !
albero.getNot(albero.figlioSinistro(nd)));
718           //FiglioDestro del Nodo = ! FiglioDestro del Nodo
719           albero.setNot(albero.figlioDestro(nd), !albero.getNot(albero.figlioDestro(nd)));
720
721       }
722
723
724       private static Nodo scegliNodo(AlberoBinario albero, int j){
725           if (j!=1) {
726               Coda codaT = new Coda();
727               codaT.inCoda(albero.binRadice());
728               int i=1;
729               while (!codaT.codaVuota()){
730                   //System.out.println("SONO QUI");
731                   Nodo N=codaT.leggiCoda();
732                   codaT.fuoriCoda();
733                   if ( !albero.sinistroVuoto(N) ){
734                                                   if
(albero.leggiNodo(albero.figlioSinistro(N)).equals("&") ||
albero.leggiNodo(albero.figlioSinistro(N)).equals("|") ||
albero.leggiNodo(albero.figlioSinistro(N)).equals("!&") ||
albero.leggiNodo(albero.figlioSinistro(N)).equals("!|")){
```

```
 735                          //System.out.println(""+i+"=
"+albero.leggiNodo(albero.figlioSinistro(N))+"\t");
 736                          codaT.inCoda(albero.figlioSinistro(N));
 737                          i++;
 738                          if (j==i) //se il valore è lo stesso di quello
richiesto
 739                              return albero.figlioSinistro(N);
 740                      }
 741                  }
 742                  if ( !albero.destroVuoto(N) ){
 743                          if (albero.leggiNodo(albero.figlioDestro(N)).equals("&") ||
albero.leggiNodo(albero.figlioDestro(N)).equals("|") ||
albero.leggiNodo(albero.figlioDestro(N)).equals("!&") ||
albero.leggiNodo(albero.figlioDestro(N)).equals("!|")){
 744                          //System.out.println(""+i+"=
"+albero.leggiNodo(albero.figlioDestro(N))+"\t");
 745                          codaT.inCoda(albero.figlioDestro(N));
 746                          i++;
 747                          if (j==i) //se il valore è lo stesso di quello
richiesto
 748                              return albero.figlioDestro(N);
 749                      }
 750                  }
 751
 752              }
 753          }
 754          else {
 755               return albero.binRadice();
 756          }
 757
 758          // restituisce null se j non è un valore valido
 759          return null;
 760      }
 761
 762
 763          /**
 764        * Applicazione della distributiva Inversa su un nodo
 765        * 1) p OR (q AND r) <= (p OR q) AND (p OR q)
 766        * 2) (q AND r) OR p <= (q OR p) AND (r OR p)
 767        * 3) p AND (q OR r) <= (p AND q) OR (p AND q)
 768        * 4) (q AND r) OR p <= (q AND p) OR (r AND p)
 769        *
 770        * NON accetta stringhe con NOT legato all"operatore
 771       **/
 772      private static void distributivaInv (AlberoBinario albero, Nodo nd){
 773
 774          //System.out.println("Radice Nodo= "+albero.leggiNodo(nd));
 775          //System.out.println("Figlio destro del Nodo=
"+albero.leggiNodo(albero.figlioDestro(nd)));
 776
 777          AlberoBinario alberoTemp=new AlberoBinario();
 778
 779          if (albero.leggiNodo(nd).equals("&")) {
 780                  // 1) p OR (q AND r) <= (p OR q) AND (p OR q)
 781                  if (albero.leggiNodo(albero.figlioDestro(nd)).equals("|")
&&albero.leggiNodo(albero.figlioSinistro(nd)).equals("|")){
 782                          Nodo p1N = albero.figlioSinistro(albero.figlioSinistro(nd));
```

```
 783                             String p1S=BinServizio.binVisitaRet(albero, p1N);
 784                             Nodo p2N = albero.figlioSinistro(albero.figlioDestro(nd));
 785                             String p2S=BinServizio.binVisitaRet(albero, p2N);
 786                             //System.out.println("P1 = "+p1S+"\nP2 = "+p2S);
 787                             if (p1S.equals(p2S)) {
 788
 789                                 //Memorizzazione di p
 790                                 AlberoBinario p=new AlberoBinario();
 791                                 p=AlberoBinario.costrBinAlbero(null, null);
 792                                 p.setRadice(p1N);
 793                                 //p.binRadice()=albero.figlioSinistro(nd); NON
FUNZIONA
 794
 795                                 //Memorizzazione di q
 796                                 AlberoBinario q=new AlberoBinario();
 797                                 q=AlberoBinario.costrBinAlbero(null, null);
 798                                 q.setRadice(albero.figlioDestro(albero.figlioSinistro(nd)));
 799
//q.binRadice()=albero.figlioSinistro(figlioDestro); NON FUNZIONA
 800
 801                                 //Memorizzazione di r
 802                                 AlberoBinario r=new AlberoBinario();
 803                                 r=AlberoBinario.costrBinAlbero(null, null);
 804                                 r.setRadice(albero.figlioDestro(albero.figlioDestro(nd)));
 805                                 //r.binRadice()=albero.figlioDestro(figlioDestro);
NON FUNZIONA
 806
 807                                 //Costruzione del figlio Sinistro
 808                                 AlberoBinario alberoSinistro=new AlberoBinario();
 809                                 alberoSinistro=p;
 810
 811                                 //Costruzione del figlio Destro
 812                                 AlberoBinario alberoDestro=new AlberoBinario();
 813                                 alberoDestro=AlberoBinario.costrBinAlbero(q, r);
 814                                 //Assegna OR alla radice del figlio destro
 815                                 alberoDestro.scriviNodo(alberoDestro.binRadice(), "&",
false);
 816
 817                                 //Ricostruzione del SottoAlbero
 818                                 AlberoBinario sottoAlbero=new AlberoBinario();
 819
sottoAlbero=AlberoBinario.costrBinAlbero(alberoSinistro, alberoDestro);
 820                                 sottoAlbero.scriviNodo(sottoAlbero.binRadice(), "|",
false);
 821
 822                                 //System.out.println("Nuovo SottoAlbero");
 823                                 //BinServizio.binVisitaFin(sottoAlbero,
sottoAlbero.binRadice());
 824                                 //System.out.println("\n");
 825
 826                                 if (albero.binPadre(nd)!=null) { //Se non è la radice
dell"albero iniziale
 827                                     //Aggiorna il padre del nodo
 828                                     if (albero.figlioDestro(albero.binPadre(nd))==nd){
 829                                         albero.setFiglioDestro(albero.binPadre(nd),
sottoAlbero.binRadice());
 830                                     }
```

```
 831                                     if (albero.figlioSinistro(albero.binPadre(nd))==nd){
 832                                         albero.setFiglioSinistro(albero.binPadre(nd),
sottoAlbero.binRadice());
 833                                     }
 834                                     nd=sottoAlbero.binRadice();
 835                                 }
 836                                 else {//Se il nodo è la radice dell"Albero iniziale
 837                                     //Aggiornamento del nodo
 838                                     nd=sottoAlbero.binRadice();
 839                                     albero.setRadice(nd);
 840                                 }
 841                                 }
 842                                 // 2) (q AND r) OR p <= (q OR r) AND (r OR p)
 843                             else if (albero.leggiNodo(albero.figlioDestro(nd)).equals("|")
&& albero.leggiNodo(albero.figlioSinistro(nd)).equals("|")){
 844                                     p1N = albero.figlioDestro(albero.figlioSinistro(nd));
 845                                     p1S=BinServizio.binVisitaRet(albero, p1N);
 846                                     p2N = albero.figlioDestro(albero.figlioDestro(nd));
 847                                     p2S=BinServizio.binVisitaRet(albero, p2N);
 848                                     if (p1S.equals(p2S)) {
 849
 850
 851                                         //Memorizzazione di p
 852                                         AlberoBinario p=new AlberoBinario();
 853                                         p=AlberoBinario.costrBinAlbero(null, null);
 854                                         p.setRadice(p1N);
 855                                         //p.binRadice()=albero.figlioSinistro(nd);
NON FUNZIONA
 856
 857                                         //Memorizzazione di q
 858                                         AlberoBinario q=new AlberoBinario();
 859                                         q=AlberoBinario.costrBinAlbero(null, null);
 860
q.setRadice(albero.figlioSinistro(albero.figlioSinistro(nd)));
 861
//q.binRadice()=albero.figlioSinistro(figlioDestro); NON FUNZIONA
 862
 863                                         //Memorizzazione di r
 864                                         AlberoBinario r=new AlberoBinario();
 865                                         r=AlberoBinario.costrBinAlbero(null, null);
 866
r.setRadice(albero.figlioSinistro(albero.figlioDestro(nd)));
 867
//r.binRadice()=albero.figlioDestro(figlioDestro); NON FUNZIONA
 868
 869                                         //Costruzione del figlio Sinistro
 870                                         AlberoBinario alberoSinistro=new
AlberoBinario();
 871
alberoSinistro=AlberoBinario.costrBinAlbero(q, r);
 872                                         //Assegna AND alla radice del figlio
sinistro
 873
alberoSinistro.scriviNodo(alberoSinistro.binRadice(), "&", false);
 874
 875                                         //Costruzione del figlio Destro
```

```
 876                                         AlberoBinario alberoDestro=new
AlberoBinario();
 877                                         alberoDestro=p;
 878
 879                                         //Ricostruzione del SottoAlbero
 880                                         AlberoBinario sottoAlbero=new
AlberoBinario();
 881
sottoAlbero=AlberoBinario.costrBinAlbero(alberoSinistro, alberoDestro);
 882
sottoAlbero.scriviNodo(sottoAlbero.binRadice(), "|", false);
 883
 884                                         //System.out.println("Nuovo SottoAlbero");
 885                                         //BinServizio.binVisitaFin(sottoAlbero,
sottoAlbero.binRadice());
 886                                         //System.out.println("\n");
 887
 888                                         if (albero.binPadre(nd)!=null) { //Se non è
la radice dell"albero iniziale
 889                                             //Aggiorna il padre del nodo
 890                                             if
(albero.figlioDestro(albero.binPadre(nd))==nd){
 891
albero.setFiglioDestro(albero.binPadre(nd), sottoAlbero.binRadice());
 892                                             }
 893                                             if
(albero.figlioSinistro(albero.binPadre(nd))==nd){
 894
albero.setFiglioSinistro(albero.binPadre(nd), sottoAlbero.binRadice());
 895                                             }
 896                                             nd=sottoAlbero.binRadice();
 897                                         }
 898                                         else {//Se il nodo è la radice dell"Albero
iniziale
 899                                             //Aggiornamento del nodo
 900                                             nd=sottoAlbero.binRadice();
 901                                             albero.setRadice(nd);
 902                                         }
 903                                     }
 904                                 }
 905                                 }
 906                                 else
 907                                         System.out.println("Nessuna Modifica
effettuata");
 908                     }
 909                     else if (albero.leggiNodo(nd).equals("|")) {
 910                     // 3) p AND (q OR r) <= (p AND q) OR (p AND q)
 911                     if (albero.leggiNodo(albero.figlioDestro(nd)).equals("&")
&&albero.leggiNodo(albero.figlioSinistro(nd)).equals("&")){
 912                             Nodo p1N = albero.figlioSinistro(albero.figlioSinistro(nd));
 913                             String p1S=BinServizio.binVisitaRet(albero, p1N);
 914                             Nodo p2N = albero.figlioSinistro(albero.figlioDestro(nd));
 915                             String p2S=BinServizio.binVisitaRet(albero, p2N);
 916                             //System.out.println("P1 = "+p1S+"\nP2 = "+p2S);
 917                             if (p1S.equals(p2S)) {
 918
 919                                 //Memorizzazione di p
```

```
 920                                 AlberoBinario p=new AlberoBinario();
 921                                 p=AlberoBinario.costrBinAlbero(null, null);
 922                                 p.setRadice(p1N);
 923                                 //p.binRadice()=albero.figlioSinistro(nd); NON
FUNZIONA
 924
 925                                 //Memorizzazione di q
 926                                 AlberoBinario q=new AlberoBinario();
 927                                 q=AlberoBinario.costrBinAlbero(null, null);
 928                                 q.setRadice(albero.figlioDestro(albero.figlioSinistro(nd)));
 929
//q.binRadice()=albero.figlioSinistro(figlioDestro); NON FUNZIONA
 930
 931                                 //Memorizzazione di r
 932                                 AlberoBinario r=new AlberoBinario();
 933                                 r=AlberoBinario.costrBinAlbero(null, null);
 934                                 r.setRadice(albero.figlioDestro(albero.figlioDestro(nd)));
 935                                 //r.binRadice()=albero.figlioDestro(figlioDestro);
NON FUNZIONA
 936
 937                                 //Costruzione del figlio Sinistro
 938                                 AlberoBinario alberoSinistro=new AlberoBinario();
 939                                 alberoSinistro=p;
 940
 941                                 //Costruzione del figlio Destro
 942                                 AlberoBinario alberoDestro=new AlberoBinario();
 943                                 alberoDestro=AlberoBinario.costrBinAlbero(q, r);
 944                                 //Assegna OR alla radice del figlio destro
 945                                 alberoDestro.scriviNodo(alberoDestro.binRadice(), "|",
false);
 946
 947                                 //Ricostruzione del SottoAlbero
 948                                 AlberoBinario sottoAlbero=new AlberoBinario();
 949
sottoAlbero=AlberoBinario.costrBinAlbero(alberoSinistro, alberoDestro);
 950                                 sottoAlbero.scriviNodo(sottoAlbero.binRadice(), "&",
false);
 951
 952                                 //System.out.println("Nuovo SottoAlbero");
 953                                 //BinServizio.binVisitaFin(sottoAlbero,
sottoAlbero.binRadice());
 954                                 //System.out.println("\n");
 955
 956                                 if (albero.binPadre(nd)!=null) { //Se non è la radice
dell"albero iniziale
 957                                     //Aggiorna il padre del nodo
 958                                     if (albero.figlioDestro(albero.binPadre(nd))==nd){
 959                                         albero.setFiglioDestro(albero.binPadre(nd),
sottoAlbero.binRadice());
 960                                     }
 961                                     if (albero.figlioSinistro(albero.binPadre(nd))==nd){
 962                                         albero.setFiglioSinistro(albero.binPadre(nd),
sottoAlbero.binRadice());
 963                                     }
 964                                     nd=sottoAlbero.binRadice();
 965                                 }
 966                                 else {//Se il nodo è la radice dell"Albero iniziale
```

```
 967                                    //Aggiornamento del nodo
 968                                    nd=sottoAlbero.binRadice();
 969                                    albero.setRadice(nd);
 970                                }
 971                                }
 972                                // 4) (q OR r) AND p <= (q AND r) OR (r AND p)
 973                            else if (albero.leggiNodo(albero.figlioDestro(nd)).equals("&")
&& albero.leggiNodo(albero.figlioSinistro(nd)).equals("&")){
 974                                    p1N = albero.figlioDestro(albero.figlioSinistro(nd));
 975                                    p1S=BinServizio.binVisitaRet(albero, p1N);
 976                                    p2N = albero.figlioDestro(albero.figlioDestro(nd));
 977                                    p2S=BinServizio.binVisitaRet(albero, p2N);
 978                                    if (p1S.equals(p2S)) {
 979
 980
 981                                        //Memorizzazione di p
 982                                        AlberoBinario p=new AlberoBinario();
 983                                        p=AlberoBinario.costrBinAlbero(null, null);
 984                                        p.setRadice(p1N);
 985                                        //p.binRadice()=albero.figlioSinistro(nd);
NON FUNZIONA
 986
 987                                        //Memorizzazione di q
 988                                        AlberoBinario q=new AlberoBinario();
 989                                        q=AlberoBinario.costrBinAlbero(null, null);
 990
q.setRadice(albero.figlioSinistro(albero.figlioSinistro(nd)));
 991
//q.binRadice()=albero.figlioSinistro(figlioDestro); NON FUNZIONA
 992
 993                                        //Memorizzazione di r
 994                                        AlberoBinario r=new AlberoBinario();
 995                                        r=AlberoBinario.costrBinAlbero(null, null);
 996
r.setRadice(albero.figlioSinistro(albero.figlioDestro(nd)));
 997
//r.binRadice()=albero.figlioDestro(figlioDestro); NON FUNZIONA
 998
 999                                        //Costruzione del figlio Sinistro
1000                                        AlberoBinario alberoSinistro=new
AlberoBinario();
1001
alberoSinistro=AlberoBinario.costrBinAlbero(q, r);
1002                                        //Assegna AND alla radice del figlio
sinistro
1003
alberoSinistro.scriviNodo(alberoSinistro.binRadice(), "|", false);
1004
1005                                        //Costruzione del figlio Destro
1006                                        AlberoBinario alberoDestro=new
AlberoBinario();
1007                                        alberoDestro=p;
1008
1009                                        //Ricostruzione del SottoAlbero
1010                                        AlberoBinario sottoAlbero=new
AlberoBinario();
```

```
1011
sottoAlbero=AlberoBinario.costrBinAlbero(alberoSinistro, alberoDestro);
1012
sottoAlbero.scriviNodo(sottoAlbero.binRadice(), "&", false);
1013
1014                                        //System.out.println("Nuovo SottoAlbero");
1015                                        //BinServizio.binVisitaFin(sottoAlbero,
sottoAlbero.binRadice());
1016                                        //System.out.println("\n");
1017
1018                                        if (albero.binPadre(nd)!=null) { //Se non è
la radice dell"albero iniziale
1019                                            //Aggiorna il padre del nodo
1020                                            if
(albero.figlioDestro(albero.binPadre(nd))==nd){
1021
albero.setFiglioDestro(albero.binPadre(nd), sottoAlbero.binRadice());
1022                                            }
1023                                            if
(albero.figlioSinistro(albero.binPadre(nd))==nd){
1024
albero.setFiglioSinistro(albero.binPadre(nd), sottoAlbero.binRadice());
1025                                            }
1026                                            nd=sottoAlbero.binRadice();
1027                                        }
1028                                        else {//Se il nodo è la radice dell"Albero
iniziale
1029                                            //Aggiornamento del nodo
1030                                            nd=sottoAlbero.binRadice();
1031                                            albero.setRadice(nd);
1032                                        }
1033                                    }
1034                                }
1035                                }
1036                                else
1037                                        System.out.println("Nessuna Modifica
effettuata");
1038                    }
1039                    else
1040                                System.out.println("Nessuna Modifica effettuata");
1041      }
1042
1043 }
```

Nodo.java

```
1 /*
2  * Nodo.java
3  *
4  * @author Andrea Maiellaro & Domenico Pio Novelli
5  * Data Creazione: 13 dicembre 2005
6  * Data Ultima Modifica: 16 Dicembre 2005
7  *
8  */
```

```
 9
10 package AlberoBinario;
11
12 public class Nodo {
13
14     private Nodo figlioDestro=null;
15     private Nodo figlioSinistro=null;
16     private Nodo genitore=null;
17     private String etichetta="";
18     private boolean flagNot=false;
19
20     /** Creates a new instance of Nodo */
21     public Nodo() {
22     }
23
24     public Nodo getFiglioDestro(){
25         return figlioDestro;
26     }
27
28     public Nodo getFiglioSinistro(){
29         return figlioSinistro;
30     }
31
32     public Nodo getGenitore(){
33         return genitore;
34     }
35
36     public String getEtichetta(){
37         return etichetta;
38     }
39
40     public boolean getNot(){
41         return flagNot;
42     }
43
44     public void setFiglioDestro(Nodo fd){
45         figlioDestro = fd;
46     }
47
48     public void setFiglioSinistro(Nodo fs){
49         figlioSinistro = fs;
50     }
51
52     public void setGenitore(Nodo gn){
53         genitore = gn;
54     }
55
56     public void setEtichetta(String etc){
57         etichetta = etc;
58     }
59
60     public void setNot(boolean bn){
61         flagNot=bn;
62     }
63 }
```

AlberoBinario.java

```
1  /*
2   * AlberoBinario.java
3   *
4   * @author Andrea Maiellaro & Domenico Pio Novelli
5   * Data Creazione: 13 dicembre 2005
6   * Data Ultima Modifica: 20 Dicembre 2005
7   *
8   */
9
10 package AlberoBinario;
11
12 public class AlberoBinario {
13
14     private Nodo radiceAlbero=null;
15
16     /* Seguendo le specifiche:
17      *
18      * public AlberoBinario(){
19      *       creaBinAlbero()
20      * }
21      *
22      * private creaBinAlbero(){}
23      *
24      * SPRECO DI MEMORIA Stack: Due chiamate a metodi che non fanno nulla
25      */
26
27     /**
28      * Corrispondente a CreaBinAlbero delle specifiche
29      */
30     public AlberoBinario() {
31     }
32
33     /**
34      * @return boolean
35      * POST: b=VERO SE Albero Vuoto
36      *        b=FALSO ALTRIMENTI
37      */
38     public boolean binAlberoVuoto(){
39         //POST: b=VERO SE Albero Vuoto
40         if (radiceAlbero==null)
41             return true;
42         // POST: b=FALSO ALTRIMENTI
43         else
44             return false;
45     }
46
47     /**
48      * @return Nodo Radice dell"Albero
49      * PRE: Albero non vuoto
50      * POST: RADICE DI T LIVELLO(u) = 0
51      */
52     public Nodo binRadice(){
53         //Pre condizione non inserita: se l"albero è vuoto restituisce null
54         return radiceAlbero;
```

```
55      }
56
57      /**
58       * PRE: T non vuoto, nd non appartenente a N, LIVELLO(nd) > 0
59       * POST: v E' PADRE DI nd -> (v,nd) ? A -> LIVELLO(nd)=LIVELLO(nd)+1
60       */
61      public Nodo binPadre(Nodo nd){
62          //Pre condizione non inserita:
63          //se l"albero è vuoto o il nodo passato è la radice restituisce null
64          return nd.getGenitore();
65      }
66
67      public Nodo figlioSinistro (Nodo nd){
68          //Pre condizione non inserita:
69          //se il nodo passato non ha figlio sinistro restituisce null
70          return nd.getFiglioSinistro();
71      }
72
73      public Nodo figlioDestro (Nodo nd){
74          //Pre condizione non inserita:
75          //se il nodo passato non ha figlio destro restituisce null
76          return nd.getFiglioDestro();
77      }
78
79      public boolean sinistroVuoto (Nodo nd){
80          //Pre condizione non inserita
81          if (nd.getFiglioSinistro()==null)
82              return true;
83          else
84              return false;
85      }
86
87      public boolean destroVuoto (Nodo nd){
88          //Pre condizione non inserita
89          if (nd.getFiglioDestro()==null)
90              return true;
91          else
92              return false;
93      }
94
95      public static AlberoBinario costrBinAlbero (AlberoBinario a1, AlberoBinario a2){
96          Nodo radice = new Nodo();
97          AlberoBinario a3 = new AlberoBinario();
98          if (a1!=null){
99              radice.setFiglioSinistro(a1.binRadice());
100             a1.binRadice().setGenitore(radice);
101         }
102         if (a2!=null){
103             radice.setFiglioDestro(a2.binRadice());
104             a2.binRadice().setGenitore(radice);
105         }
106         a3.radiceAlbero=radice;
107         return a3;
108     }
109
110     public void cancSottoBinAlbero (Nodo nd){
111         //Pre condizione: albero non vuoto
```

```
112         if (radiceAlbero!=null)
113
114                     // Cancellazzione ricorsiva dei sottoalberi
115                     if (nd.getFiglioSinistro()!=null)
116                         cancSottoBinAlbero (nd.getFiglioSinistro());
117                     if (nd.getFiglioDestro()!=null)
118                         cancSottoBinAlbero (nd.getFiglioDestro());
119
120                         //Se il nodo e" la radice dell"albero
121                 if (nd.getGenitore()==null)
122                     //Se è la radice dell"albero
123                     radiceAlbero=null;
124                 else
125                 {
126                     Nodo padre=nd.getGenitore();
127                     //Faccio perdere il riferimento da parte del padre
128                     if (nd.getGenitore().getFiglioSinistro()==nd){
129                         padre.setFiglioSinistro(null);
130                     }
131                     else
132                         padre.setFiglioDestro(null);
133                 }
134     }
135
136     public String leggiNodo (Nodo nd){
137                 //Pre condizione non inserita
138         String str=nd.getEtichetta();
139         if (nd.getNot())
140             str="!"+str;
141         return str;
142     }
143
144     public void scriviNodo (Nodo nd, String etc, boolean flagNot){
145         //Pre condizione non inserita
146         nd.setEtichetta(etc);
147         nd.setNot(flagNot);
148     }
149
150     public void setRadice(Nodo nd){
151         //Non è un InsRadice
152         radiceAlbero=nd;
153     }
154
155     public void setFiglioDestro(Nodo nd, Nodo nuovo){
156         //Non è un InsFiglioDestro
157         nd.setFiglioDestro(nuovo);
158     }
159
160     public void setFiglioSinistro(Nodo nd, Nodo nuovo){
161         //Non è un InsFiglioSinistro
162         nd.setFiglioSinistro(nuovo);
163     }
164
165     //aggiunta a scriviNodo
166     public void setNot(Nodo nd, boolean flag){
167         nd.setNot(flag);
168     }
```

```
169
170     //aggiunta a LeggiNodo
171     public boolean getNot(Nodo nd){
172         return nd.getNot();
173     }
174
175 }
176
```

BinServizio.java

```
 1 /*
 2  * BinServizio.java
 3  *
 4  * @author Andrea Maiellaro & Domenico Pio Novelli
 5  * Data Creazione: 15 dicembre 2005
 6  * Data Ultima Modifica: 15 Dicembre 2005
 7  *
 8  */
 9
10 package AlberoBinario;
11
12 public class BinServizio {
13
14     public static void binInVisita(AlberoBinario aB, Nodo nd){
15         if (nd!=null){
16             if (!aB.sinistroVuoto(nd)){
17                 binInVisita (aB, nd.getFiglioSinistro());
18             }
19
20             //InVisita
21             System.out.println(nd.getEtichetta());
22
23             if (!aB.destroVuoto(nd)){
24                 binInVisita (aB, nd.getFiglioDestro());
25             }
26         }
27     }
28
29     public static void binPreVisita(AlberoBinario aB, Nodo nd){
30         if (nd!=null){
31
32             //PreVisita
33             System.out.println(nd.getEtichetta());
34
35             if (!aB.sinistroVuoto(nd)){
36                 binPreVisita (aB, nd.getFiglioSinistro());
37             }
38
39             if (!aB.destroVuoto(nd)){
40                 binPreVisita (aB, nd.getFiglioDestro());
41             }
42         }
43     }
44
```

```
45      public static void binPostVisita(AlberoBinario aB, Nodo nd){
46          if (nd!=null){
47
48              if (!aB.sinistroVuoto(nd)){
49                  binPostVisita (aB, nd.getFiglioSinistro());
50              }
51
52              if (!aB.destroVuoto(nd)){
53                  binPostVisita (aB, nd.getFiglioDestro());
54              }
55
56              //PostVisita
57              System.out.println(nd.getEtichetta());
58          }
59      }
60
61          public static void binVisitaFin(AlberoBinario aB, Nodo nd){
62          if (nd!=null){
63
64              if (nd.getEtichetta().equals("&") || nd.getEtichetta().equals("|") ){
65                  if (nd.getNot())
66                      System.out.print("!(");
67                  else
68                      System.out.print("(");
69              }
70
71              if (!aB.sinistroVuoto(nd)){
72                  binVisitaFin (aB, nd.getFiglioSinistro());
73              }
74
75              if (nd.getEtichetta().equals("&") || nd.getEtichetta().equals("|") )
76                      System.out.print(nd.getEtichetta());
77              else
78                  if (nd.getNot())
79                      System.out.print("!"+nd.getEtichetta());
80                  else
81                      System.out.print(nd.getEtichetta());
82
83              if (!aB.destroVuoto(nd)){
84                  binVisitaFin (aB, nd.getFiglioDestro());
85              }
86
87              if (nd.getEtichetta().equals("&") || nd.getEtichetta().equals("|") )
88                  System.out.print(")");
89          }
90      }
91
92          public static String binVisitaRet(AlberoBinario aB, Nodo nd){
93
94          String str=null;
95
96          if (nd!=null){
97
98              if (nd.getEtichetta().equals("&") || nd.getEtichetta().equals("|") ){
99                  if (nd.getNot())
100                     str=str+"!(";
```

```
101                 else
102                     str=str+"(";
103             }
104
105             if (!aB.sinistroVuoto(nd)){
106                 str=str+binVisitaRet (aB, nd.getFiglioSinistro());
107             }
108
109             if (nd.getEtichetta().equals("&") || nd.getEtichetta().equals("|") )
110                     str=str+nd.getEtichetta();
111             else
112                 if (nd.getNot())
113                     str=str+"!"+nd.getEtichetta();
114                 else
115                     str=str+nd.getEtichetta();
116
117             if (!aB.destroVuoto(nd)){
118                 str=str+binVisitaRet (aB, nd.getFiglioDestro());
119             }
120
121             if (nd.getEtichetta().equals("&") || nd.getEtichetta().equals("|") )
122                 str=str+")";
123         }
124         return str;
125     }
126
127 }
128
```

NodoC.java

```
1  /*
2   * NodoC.java
3   *
4   * @author Andrea Maiellaro & Domenico Pio Novelli
5   * Data Creazione: 16 dicembre 2005
6   * Data Ultima Modifica: 16 Dicembre 2005
7   *
8   */
9
10 package Coda;
11
12 import AlberoBinario.*;
13
14 public class NodoC {
15     private Nodo elemento=null;
16     private NodoC succ=null;
17
18     /** Creates a new instance of NodoC */
19     public NodoC() {
20     }
21
22     public void setElemento(Nodo nd){
23         elemento=nd;
24     }
```

```
25
26      public Nodo getElemento(){
27          return elemento;
28      }
29
30      public void setSucc(NodoC nd){
31          succ=nd;
32      }
33
34      public NodoC getSucc(){
35          return succ;
36      }
37  }
38
39
```

Coda.java

```
1   /*
2   * Coda.java
3   *
4   * @author Andrea Maiellaro & Domenico Pio Novelli
5   * Data Creazione: 16 dicembre 2005
6   * Data Ultima Modifica: 16 Dicembre 2005
7   *
8   */
9
10  package Coda;
11
12  import AlberoBinario.*;
13
14  public class Coda {
15          private NodoC testa=null;
16          private NodoC fondo=null;
17
18      /** Creates a new instance of Coda */
19      public Coda() {
20      }
21
22      public boolean codaVuota() {
23          if (testa==null)
24              return true;
25          else
26              return false;
27      }
28
29      public Nodo leggiCoda(){
30          return testa.getElemento();
31      }
32
33      public void fuoriCoda(){
34          testa=testa.getSucc();
35      }
36
37      public void inCoda (Nodo nd){
```

```
38          NodoC nuovo=new NodoC();
39          nuovo.setElemento(nd);
40          if (testa==null)
41                  testa=nuovo;
42          else
43               fondo.setSucc(nuovo);
44          fondo=nuovo;
45      }
46
47  }
48
```

5. TEST

Sono stati effettuati dei test sul programma per verificarne il funzionamento.

Per provare l'**Acqusizione della Stringa e Creazione dell'Albero** sono state fatte le seguenti prove:

- Implementazione **C++**
  - Stringa vuota -> Messaggio di Errore
  - Stringa errata senza operatori -> Uscita dal Programma *(Errore critico)*
  - Stringa errata -> Prosecuzione del programma e Costruzione di un Albero Errato
  - Stringa esatta -> Prosecuzione del programma e Costruzione dell'Albero corrispondente
- Implementazione **Java**
  - Stringa vuota -> Messaggio di Errore
  - Stringa errata senza operatori -> Messaggio di Errore
  - Stringa errata -> Prosecuzione del programma e Costruzione di un Albero Errato
  - Stringa esatta -> Prosecuzione del programma e Costruzione dell'Albero corrispondente

Per provare la **Costruzione** dell'Albero Binario, sono state provate le seguenti espressioni ben poste:

- (A&b)
- ((A&B)&C)
- (a&(!B&C))
- (((A&B)&C)&D)
- ((((!A&B)&C)&D)&E)
- (((A&B)&!C)&((A|!B)|(C&D)))
- (((A&B)&(!C&D))|E)

In tutti è casi è stato hanno costruito l'albero corrispondente

Per provare le **Leggi di De Morgan** su un nodo di un Albero sono state fatte le seguenti prove:

- (a&b) -> Costruzione dell'esatto albero corrispondente a !(!a|!b)
- !(a|b) -> Costruzione dell'esatto albero corrispondente a (!a&!b)
- Applicazione delle leggi di De Morgan due volte consecutivamente allo stesso nodo -> ritorno all'espressione di partenza

Per provare la **Distributiva** su un nodo di un Albero sono state fatte le seguenti prove:

- (c|(a&!b)) -> Costruzione dell'esatto albero corrispondente a ((c&a) | (c&!b))
- (a&(c|(a&b))) -> Costruzione dell'esatto albero corrispondente a (a&((c&a) | (c&b)))
- !((a|b)&c) -> nessuna modifica effettuata
- (a&!(b|c)) -> nessuna modifica effettuata
- (a&b) -> nessuna modifica effettuata

Per provare la **Distributiva Inversa** su un nodo di un Albero sono state fatte le seguenti prove:

- (((a|b)&c)|((a|b)&d)) -> Costruzione dell'esatto albero corrispondente a (a|b)&(c|d)
- ((a|b)&c) -> nessuna modifica effettuata
- (a&b) -> nessuna modifica effettuata
- ((!(a|b)&c)|(!(a|b)&d)) -> nessuna modifica effettuata
- !(!((a|b)&c)|!((a|b)&d)) -> nessuna modifica effettuata