Università degli Studi di Napoli “Parthenope”

Relazione progetto di Lab. Di Algoritmi e Strutture dati

Traccia 2: *“Codifica di Huffman implementata con coda Red-Black”*

Studente: *Vigliotti Clemente*
Matricola: *0124000822*

Indice

Testo della Traccia

Costruire un encoder (codificatore) ed un decoder (decodificatore) che utilizzi la codifica di Huffman. L' implementazione deve far uso di una coda di priorità, realizzata mediante un albero binario RED BLACK. Definire il formato (od i formati) dei dati accettati in input e definire il formato del file codificato. Calcolare il tasso di compressione ottenuto. Il codificatore deve essere almeno in grado di codificare qualunque file di testo (.txt) e, facoltativamente, anche altri tipi di formati (bmp, wav, …).

Cenni Teorici

Introduzione

Sin dagli albori dell'informatica, ci si è posti il problema della capacità limitata della memoria delle macchine. Infatti, anche se oggi con le capacità raggiunte dai dischi e dalle memorie può sembrare banale, per un informatico limitare gli sprechi di memoria è fondamentale.
Un notevole passo avanti nell'ottimizzazione dello spazio è stato fatto grazie all'algoritmo che prende il nome dal suo inventore:
"La Codifica di Huffam".
Esso è un algoritmo di ***compressione dei dati****, nato inizialmente come compressore di testi, che fonda la sua logica sulla frequenza delle lettere all'interno di una stringa.*
L'idea di base è quella di assegnare codici binari più brevi alle lettere più frequenti, e codici più lunghi alle lettere meno frequenti.
Per generare questi codici, l'algoritmo fa uso di alcune strutture dati complesse.
In particolare sfrutta una ***Coda di priorità****, che a sua volta è implementata facendo uso di una struttura dati di tipo albero. Andremo ad analizzare queste strutture una ad una nei prossimi paragrafi, così da poter analizzare poi più nello specifico l'algoritmo.*

Albero binario di ricerca

L'ABR è una struttura dati rappresentata come un grafo, ossia un insieme di elementi detti nodi che sono collegati tra loro da uno e un solo cammino. Ogni nodo di un ABR ha un grado compreso tra 0 e 2, esso contiene una ***chiave*** *e dei* ***puntatori*** *ai suoi nodi adiacenti.*
La sua particolarità è che mantiene i nodi ordinati secondo una logica precisa, cioè:

> *preso un nodo X, il suo figlio sinistro avrà una chiave minore della chiave del padre e il suo figlio destro avrà una chiave maggiore del padre.*

Quindi risulta evidente che la ricerca del min, del max o di un elemento generico diviene molto più veloce in questo tipo di albero.

Le operazioni principali che si compiono sugli ABR, sono: l'inserimento, la cancellazione, la ricerca di una chiave, la ricerca del successore/predecessore, la ricerca del Min/Max e le visite.
Sugli ABR esistono 3 principali algoritmi di visita,

- *l'attraversamento anticipato (PREORDER)*
- *l'attravresamento simmetrico (INORDER)*
- *l'attraversamento posticipato (POSTORDER)*

Il costo delle operazioni su un ABR risulta essere O(h) dove h è l'altezza dell'albero, nel caso peggiore, cioè quando i nodi vengono inseriti ordinatamente, può salire a O(n), dove n è il numero di nodi. Risulta perciò fondamentale mantenere l'albero bilanciato. Ecco l'importanza dei Red-Black.

Albero Red-Black

Un ARB è un tipo particolare di Albero Binario di Ricerca, la sua particolarità, come abbiamo accennato nel paragrafo precedente, è che si autobilancia.
Per permettere all'ARB di autobilanciarsi, si aggiunge un ulteriore attributo al nodo, il colore, che può assumere solo due valori (per convenzione Rosso e Nero).
Oltre a questo un ARB deve soddisfare alcune proprietà fondamentali:

1. *Un nodo può essere solo Rosso o Nero.*
2. *I nodi NIL sono neri.*
3. *La Radice è nera.*
4. *Un nodo rosso ha tutti i figli neri.*
5. *Ogni percorso da un nodo ad una foglia contiene lo stesso numero di nodi neri.*

Rispettando questi 5 vincoli, viene garantito l'autobilanciamento, infatti un altra proprietà che deriva da questi 5 vincoli è che ***il cammino più lungo dalla radice ad una foglia è al massimo lungo il doppio del cammino più breve.***
Sugli alberi Red-Black, poiché sono una specializzazione degli ABR, è possibile eseguire le medesime operazioni, però a differenza di questi le operazioni di inserimento e cancellazione prevedono delle chiamate a ulteriori funzioni, stiamo parlando di INSERT-FIXUP e DELETE-FIXUP.

Insert-FixUP

Questa funzione provvede al ripristino delle proprietà dell'albero Red-Black. Viene richiamata dopo il normale inserimento degli ABR. Il tempo totale richiesto per eseguirla è pari a O(log n) dove n è il numero di nodi.

```
void ARB::InsertFixUP(Nodo* x) {

while ( x->GetParent()->GetColor() == RED) {

        if(x->GetParent() == x->GetParent()->GetParent()->GetSx()) {

        Nodo* y = x->GetParent()->GetParent()->GetDx();

            if(y->GetColor() == RED) {

                x->GetParent()->SetColor(BLACK);
                y->SetColor(BLACK);
                x->GetParent()->GetParent()->SetColor(RED);
                x=x->GetParent()->GetParent();
                }
            else
            {
                if(x==x->GetParent()->GetDx()){
                        x=x->GetParent();
                        rotate_sx(x);
                }

                x->GetParent()->SetColor(BLACK);
                x->GetParent()->GetParent()->SetColor(RED);
                x->GetParent()->GetDx()->SetColor(BLACK);
                rotate_dx(x->GetParent()->GetParent());
            }
        }

        else  //COME SOPRA MA CON DX E SX SCAMBIATI
```

Delete-FixUP

Questa funzione provvede al ripristino delle proprietà dell'albero Red-Black con rotazioni e cambiamenti di colore, dopo la cancellazione effettuata tramite il metodo dell'ABR.
Il tempo totale richiesto per eseguirla è pari a ***O(log n)*** *dove n è il numero di nodi. Esso quindi viene richiamato dopo la cancellazione del nodo solo se quest'ultimo è nero, infatti se il nodo cancellato è rosso, non viene violata alcuna proprietà dell'albero.*
Questo metodo prevere 4 casi possibili, più altri 4 casi che sono simmetrici con i primi:

1. *Fratello rosso, padre nero*
2. *fratello nero con figli neri*
3. *fratello nero con figlio sinistro rosso*
4. *fratello nero con figlio destro rosso*

```
void ARB::DeleteFixUP(Nodo* &x) {
    while(x != root && x->GetColor() == BLACK )
    {
        Nodo* n;
        if( x->GetParent()->GetSx() == x )
        {
            n = x->GetParent()->GetDx();

            if( n->GetColor() == RED)
            {
                n->SetColor(BLACK);
                x->GetParent()->SetColor(RED);
                rotate_sx(x->GetParent());
                n = x->GetParent()->GetDx();
            }

            if(n->GetSx()->GetColor() == BLACK && n->GetDx()->GetColor() == BLACK )
            {
                n->SetColor(RED);                x = x->GetParent();
            }
            else
            {
                if( n->GetDx()->GetColor() == BLACK )
                {
                    n->GetSx()->SetColor(BLACK);
                    n->SetColor(RED);
                    rotate_dx(n);
                    n = x->GetParent()->GetDx();

                }

                n->SetColor(x->GetParent()->GetColor());
                x->GetParent()->SetColor(BLACK);
                n->GetDx()->SetColor(BLACK);
                rotate_sx(x->GetParent());
                x = root;
```

//COME SOPRA CON DX E SX SCAMBIATI

Rotazione

Quest'operazione in realtà appartiene agli alberi binari in generale, ma prende particolare importaza nel caso dei Red-Black.

Infatti essa consente di ripristinare le proprietà dell'albero RB in seguito alle operazioni di inserimento e cancellazione (come si può notare dal codice dei metodi Insert-FixUP e DeleteFixUP).

Questa operazione provoca una rotazione in senso antiorario (rotazione sinistra) o orario (rotazione destra) e coinvolge due nodi, il pivot e il padre, essi vengono scambiati preservando le proprietà dell'albero.

```
void ARB::rotate_dx(Nodo* x) {
    Nodo* y = x->GetSx();
     x->SetSx(y->GetDx());
     if (y->GetDx() != NIL ) y->GetDx()->SetParent(x);

     y->SetParent(x->GetParent());


     if( x->GetParent() == NIL )
         root = y;
     else
     {
         if( x == x->GetParent()->GetDx())
             x->GetParent()->SetDx(y);
         else
             x->GetParent()->SetSx(y);
     }


     y->SetDx(x);
     x->SetParent(y);
}
```

Coda di Priorità

Una coda di priorità rappresenta una struttura dati che identifica una serie di elementi che sono interconnessi da una relazione d'ordine espressa tramite un parametro definito appunto ***priorità dell'elemento****. Le sue applicazioni sono molteplici, anche se può sembrare una struttura dati relativamente semplice, essa viene impiegata per molte operazioni complesse, ad esempio: Ottimizzazione dell'accesso a risorse condivise, Scheduling, Ordinamento, gestione attività ecc..*

Le operazioni principali che possono essere eseguite su una coda di Priorità sono:
Insert: *Inserimento di un elemento nella posizione opportuna.*
Delete: *Cancellazione di un elemento e ripristino dell'ordine degli stessi.*
EstraiMIN: *Estrazione e cancellazione del minimo.*
EstraiMax: *Estrazione e cancellazione del massimo.*
IncreasePriority: *Incremento della priorità di un elemento.*

In generale possiamo dire che queste operazioni possono essere implementate diversamente a secoda della struttura dati utilizzata per la sua rappresentazione. Infatti, una coda di Priorità può essere rappresentata facendo uso di diverse strutture dati, nel nostro caso, si è fatto uso di un albero ***Red-Black****. Quindi ad esempio l'operazione di inserimento farà uso del metodo di* ***insert*** *dei red-black.*

La codifica di Huffman

Come detto nell'introduzione di questo paragrafo, la Codifica di Huffman è un metodo di compressione dei dati, a seconda delle caratteristiche del file che si vuole comprimere, questo algoritmo garantisce un tasso di compressione dal 20% al 90%.
*Una particolarità dell'algoritmo è che esso genera codici che non sono mai **prefissi** di altri codici, questo fa si che durante la decompressione non si sovrappone mai un codice con un altro.*
*La complessità dell'algoritmo è **O(n log n)** dove **n** è il numero di simboli, ed esso gode della scelta **greedy**, che si realizza pescando ogni volta i due elementi di frequenza minima dalla coda, e della **sottostruttura ottima**, poiché la risoluzione ottimale di un sottoproblema permette di ricostruire efficientemente la soluzione ottimale.*

Il suo funzionamento è molto semplice, essa consiste in 4 passaggi principali:

1. *Ordina i simboli in base alla loro priorità, ossia il loro numero di occorrenze.*
2. *Itera i seguenti passi finché la coda non contiene un unico elemento:*
   (a)Estrae i due simboli con priorità minima e crea un nodo genitore di questi due
   (b)Assegna al nodo padre la somma delle priorità dei nodi figli
   (c)Inserisce il padre nella coda
3. *Assegna ad ogni elmento un codice sulla base del percorso dalla radice dell'albero creato al nodo foglia corrispondente al simbolo*

La codifica quindi viene effettuata assegnando ad ogni simbolo il proprio codice, e poiché i simboli con frequenza maggiore verranno estratti per ultimi, essi avranno un codice più breve (trovandosi in cima all'albero).
La decodifica poi viene effettuata ripercorrendo l'albero in base ai codici presenti nel file compresso.

Il numero di bit necessari per codificare un file può essere calcolato con la formula:

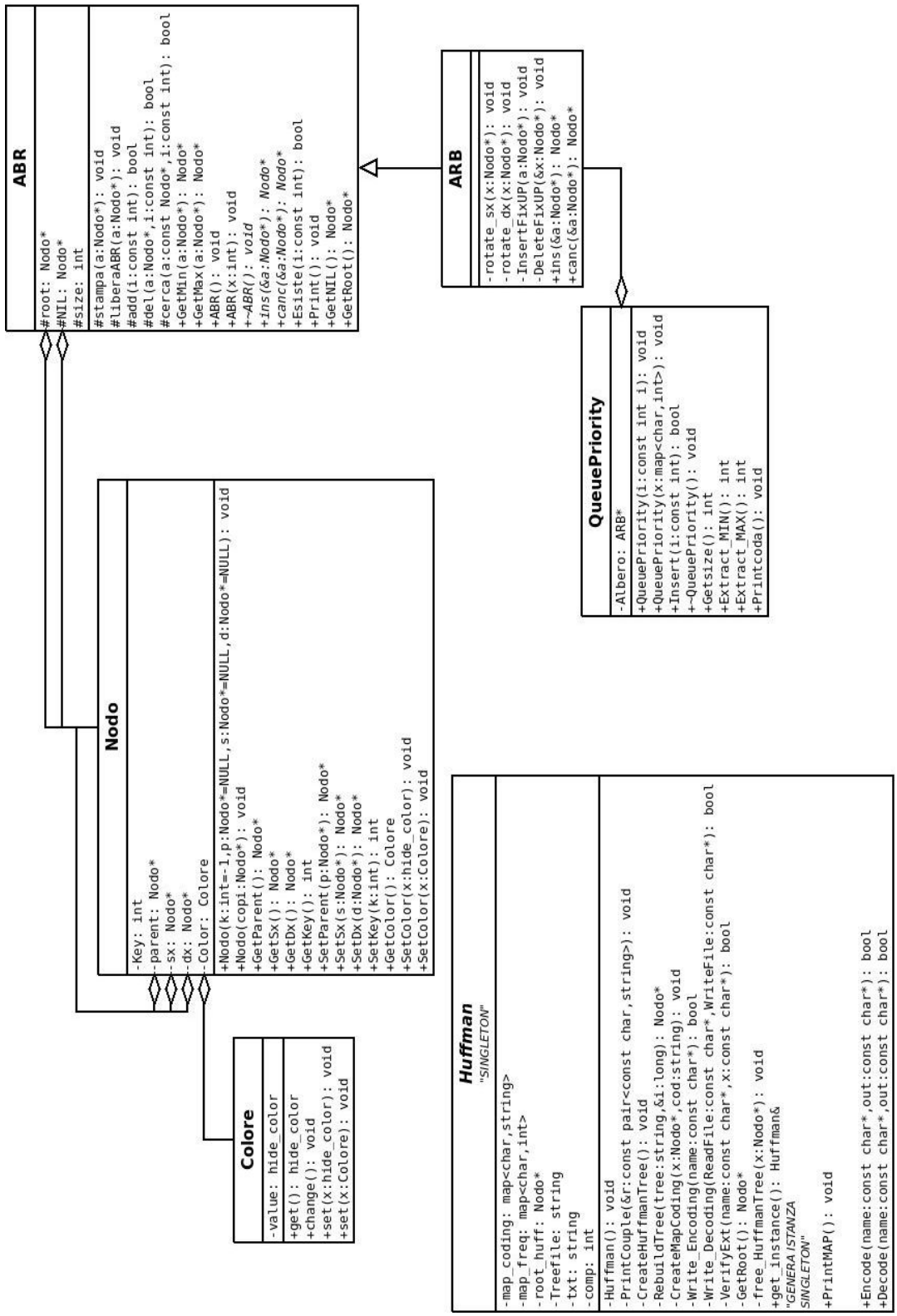
$$B(T) = \sum_{[\forall c \in C]} f(c) * d_T(c)$$

Dove:

- $B(T)$ *rappresenta il costo dell'albero T (quindi il numero di bit necessari per codificarlo).*
- $f(c)$ *rappresenta la frequenza assoluta del carattere c.*
- $d_T(c)$ *rappresenta la profondità della foglia con carattere c.*

Struttura del programma (UML)

Diagramma delle classi

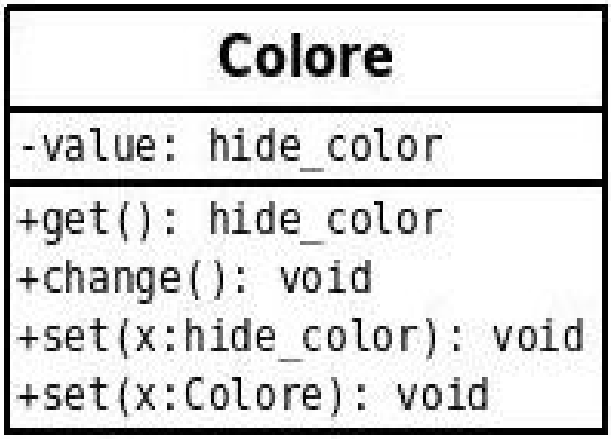


Organizzazione dei file

Il programma è composto da più header file, uno per classe, e ogni header ha associato il suo file cpp che contiene l'implementazione dei vari metodi.

Implementazione

Classe Colore

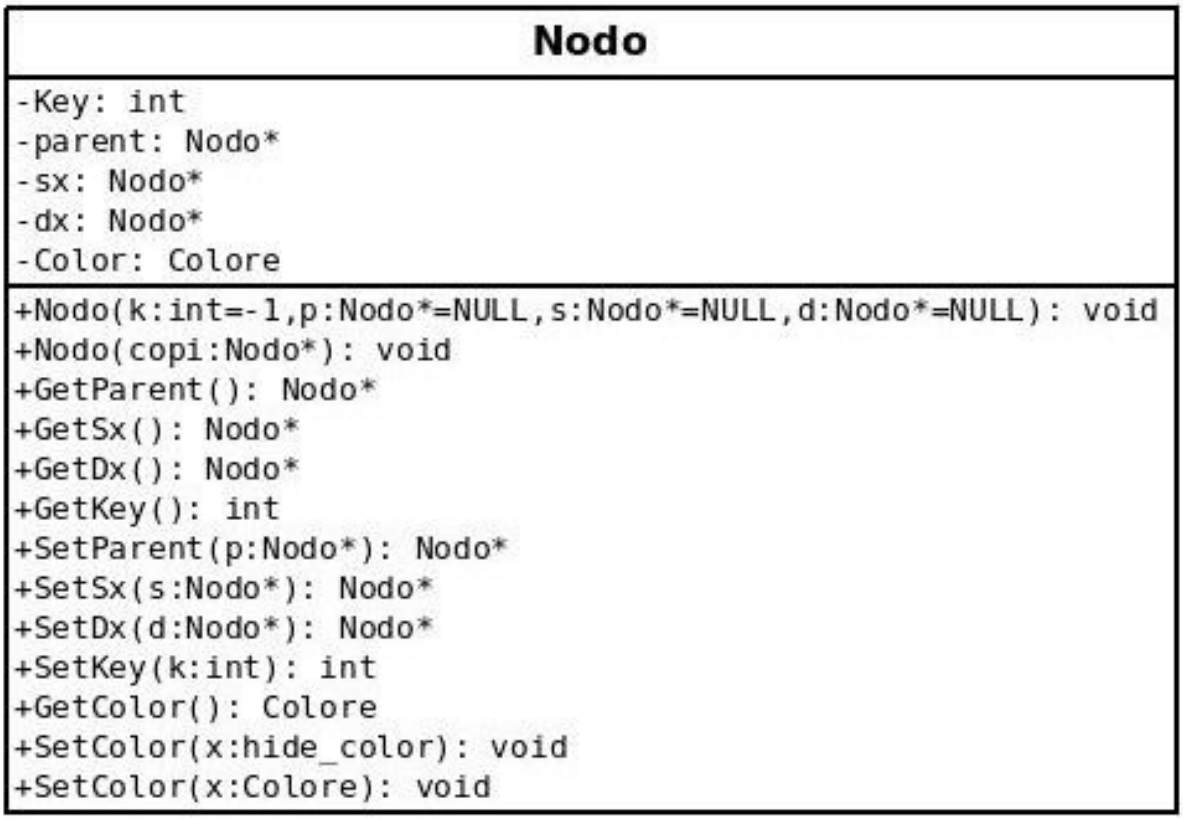


La classe Colore rappresenta l'entità usata per identificare il colore di un nodo. Essa fa uso di un tipo enumerativo ***hide_color****, che può assumere 3 valori (RED, BLACK, NULLO). Si è scelto di implementare una classe a parte piuttosto che usare direttamente il tipo hide_color, poiché quest'ultimo, essendo enumerativo soffre di alcune debolezze proprie del linguaggio, come ad esempio la possibilità di assegnare al tipo hide_color il valore di un altro tipo enumerativo equivalente. Implementando questa classe si evita ciò.*
La classe contiene oltre ai metodi classici get() e set().
Anche un metodo change() che setta il colore a RED se è BLACK e viceversa, infine è stato implementato l'overload degli operatori.

```
enum hide_color
  { RED, BLACK, NULLO } ;

class Colore
{
    private:
        hide_color value;

    public:
        Colore(hide_color c = NULLO) { value=c; }

        hide_color get() { return value; }
        void change() { if (value!=NULLO) { value = ((value == RED) ? BLACK : RED); } else { cout<<"COLORE NULLO"<<endl; }
        void set(hide_color x) { value = x;}
        void set(Colore x) { value = x.get();}


        bool operator ==( hide_color x ) { return( value == x ) ; }
        bool operator !=( hide_color x ) { return( value != x ) ; }



};
#endif
```

Classe Nodo

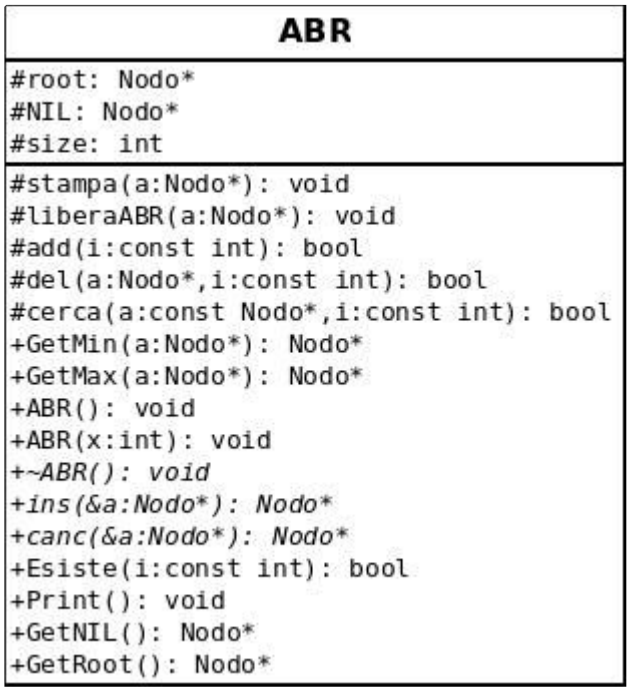
*La classe Nodo, rappresenta l'entità usata per rappresentare i Nodi degli alberi. Essa fa uso di due costruttori, uno che imposta i vari attributi, e l'altro che prende in input un altra istanza della stessa classe e ne genera una copia (**costruttore di copia**).*




```
class Nodo {
    private:

       int key;
       Nodo* parent;
       Nodo* sx;
       Nodo* dx;
       Colore color;


    public:

       Nodo (int k = -1, Nodo* p = NULL, Nodo* s = NULL, Nodo* d = NULL, Colore c=NULLO): key (k), parent(p), sx (s), dx (d), color(c) {}
       Nodo (Nodo* copia): key(copia->key), parent(copia->parent), sx(copia->sx), dx(copia->dx){color=copia->GetColor().get();}

        //METODI GET
        Nodo* GetParent() const {return parent;}
        Nodo* GetSx() const {return sx;}
        Nodo* GetDx() const {return dx;}
        int   GetKey() const {return key;}
        Colore GetColor() const {return color;}
        void SetColor(hide_color x) {color = x;}
        void SetColor(Colore x) {color = x;}
        //METODI SET
        Nodo* SetParent(Nodo* p) {return parent=p;}
        Nodo* SetSx(Nodo* s) {return sx=s;}
        Nodo* SetDx(Nodo* d) {return dx=d;}
        int   SetKey(int k) {return key=k;}


            };
```

Classe ABR

La classe ABR è l'implementazione dell'albero binario di ricerca, struttura dati descritta in precedenza.
Essa continene alcuni attributi protected poiché verrà ereditata dalla classe ARB(Albero red-black).

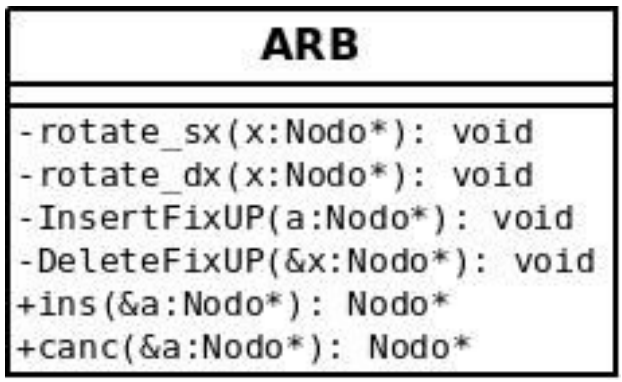


*Tra gli attributi oltre al Nodo radice, c'è un **Nodo NIL**, esso rappresenta il nodo sentinella, puntato da tutte le foglie (figlio sx e figlio dx) e dalla radice (padre).*

```
class ABR {

    protected:

       Nodo* root;
       Nodo* NIL; //NODO SENTINELLA

       int size;

       Nodo* add (Nodo* x);
       Nodo* del (Nodo* a);
       Nodo* cerca (Nodo* a, const int i);


    public:
       void stampa ( Nodo* a);
       ABR () {NIL = new Nodo(); root = NIL; size = 0;}
       ABR (int x) { NIL = new Nodo(); root = new Nodo(x,NIL,NIL,NIL); size=1; }
       void liberaABR (Nodo* a);
       virtual ~ABR() {liberaABR(root);}
       Nodo* GetRoot() {return root;}
       Nodo* GetNIL() {return NIL;}
       Nodo* GetMin (Nodo* a);
       Nodo* GetMax (Nodo* a);
       int GetSize() {return size;}
       virtual Nodo* ins (Nodo* &a) { return add (a);}
       virtual Nodo* canc (Nodo* &a) {  return del (a);}


       bool esiste (const int i) { bool value = ((cerca (root, i) == NIL) ? 0 : 1); return value; }
       void Print () {stampa(root);}

};
```

*Oltre ai vari metodi **Get()** e **Set()** abbiamo un metodo **ins()** che serve ad aggiungere un Nodo, il metodo **canc()** cha fa l'opposto, un metodo pubblico **esiste()** che fa uso del metodo protected **cerca()** e restituisce 1 nel caso in cui esiste un nodo con chiave uguale all'intero passato in input, un metodo **print()** che stampa l'albero a partire dalla radice, esso fa uso del metodo protected **stampa()**, e infine un metodo **liberaABR()** richiamato dal distruttore e che dealloca i nodi delll'ABR.*

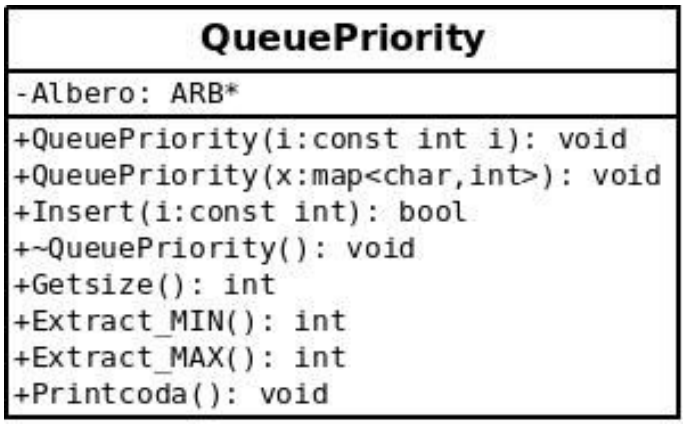
Classe ARB



La classe ARB è l'entità che rappresenta la struttura dati Albero Red-Black. Essa eredita metodi e attributi pubblici e protetti di ABR, e in più contiene i metodi descritti in precedenza nel paragrafo dedicato. Inoltre sono stati reimplementati i metodi ***push()*** *e* ***pop()****, sfruttando il* ***polimorfismo*** *offerto dal C++, la reimplementazione è dovuta al fatto che l'inserimento e la cancellazione di Nodi in un albero red-Black avviene in modo diverso (cioè richiamando anche insertFixUp e DeleteFixUP).*

```
class ARB : public ABR
{   private:
      void rotate_sx(Nodo* x);
      void rotate_dx(Nodo* x);
      void InsertFixUP(Nodo* x);
      Nodo* del(Nodo* &a);
      void DeleteFixUP(Nodo* &x);

    public:
       ARB():ABR() {root->SetColor(BLACK); NIL->SetColor(NULLO); }
       ARB(int x):ABR(x) { root->SetColor(BLACK); NIL->SetColor(NULLO);}
       Nodo* ins(Nodo* &a);
       Nodo* canc (Nodo* &a);
};
```

Classe QueuePriority



La classe QueuePriority è la rappresentazione della struttura dati coda di priorità. Essa fa uso di un attributo privato di tipo Albero Red-Black(ARB) e di due costruttori (oltre quello di default) uno che prende in input un intero e l'altro che prende in input una mappa contenente char (i simboli della codifica) e int (le frequenze dei caratteri, ossia la priorità).

L'uso di questo costruttore agevola l'implementazione dell'Algoritmo di Huffman, poiché ci permette di instanziare un oggetto della classe passandogli direttamente un'unica struttura dati contenente tutti i dati necessari.

Infine possiede i metodi classici di questa struttura dati, Insert e Extract_MIN/MAX.

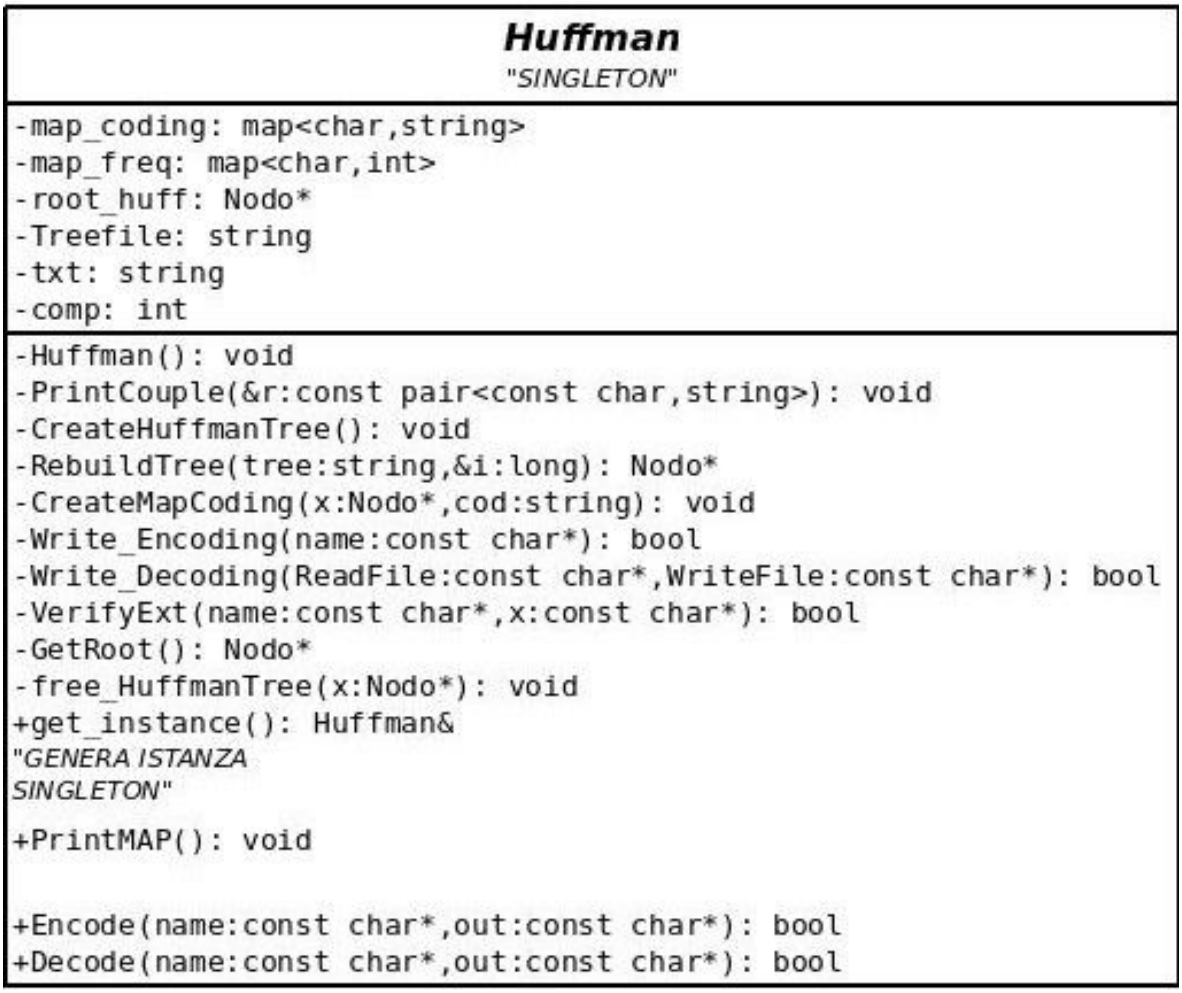
Gran parte dei metodi di questa classe fanno uso dei metodi della classe ARB, ma a differenza di quest'ultima, che lavora direttamente su istanze della classe nodo, la classe QueuePriority dispone di metodi che lavorano direttamente con numeri interi e provvede automaticamente ad allocare e deallocare nodi all'interno della struttura dati Albero RB utilizzata, in modo da evitare che si lascino allocate risorse accidentalmente.

```
#ifndef QueuePriority_H_INCLUDED
#define QueuePriority_H_INCLUDED

#include <map>

#include "ARB.h"

class QueuePriority
{
    private:
        ARB* Albero;
    public:
        QueuePriority(const int i){ Albero = new ARB(i); }
        QueuePriority(map<char,int> x);
        QueuePriority(){ Albero = new ARB(); }
        ~QueuePriority(){ delete Albero;};


        bool Insert(const int i);

        int Getsize() { return Albero->GetSize(); }

        int Extract_MIN();
        int Extract_MAX();

        void Printcoda() { Albero->Print(); }


};
```

Classe Huffman

La classe Huffman è stata implementata come ***Singleton****, ossia una classe in cui, in ogni momento dell'esecuzione, può esistere una sola istanza della classe.*

Per fare ciò, è stato dichiarato un unico costruttore dandogli una visibilità privata, e un metodo statico pubblico che ne istanzia un oggetto di tipo Huffman alla prima chiamata, richiamando il costruttore privato dichiarato in precedenza.



Questa classe si occupa sia della codifica che della decodifica. Per capire meglio come funziona andiamo ad analizzare nei dettagli la sua struttura.

Attributi

map_coding *→Mappa che contiene per ogni carattere la sua codifica di Huffman.*

map_freq *→ Mappa che contiene per ogni charattere la sua frequenza.*

root_huff *→ Nodo radice dell'albero di Huffman.*

Treefile *→ Stringa che rappresenta la codifica dell'albero di Huffman da scrivere su file.*

txt *→ Stringa che contiene il testo del file dato in input.*

Comp *→ Intero che contiene il tasso di compressione.*

Metodi:CODIFICA

Encode

```
bool Huffman::Encode(const char* name, const char* out){

if( VerifyExt(name,"txt") && VerifyExt(out,"huff") ) {
        ifstream file(name);
            if (file) {
                char ch = file.get();

                cout<<"LETTURA FILE..."<<endl;
                while(!file.eof()){
                    txt.push_back(ch);
                    map_freq[ch]++;
                    file.get(ch);
                }

                file.close();

            cout<<"\nCOSTRUZIONE ALBERO DI HUFFMAN"<<endl;
            root_huff = CreateHuffmanTree();

            string codd;
            cout<<"\nCOSTRUZIONE MAPPA CODIFICHE"<<endl;
            CreateMapCoding(root_huff,codd);

            cout<<"\nINIZIO SCRITTURA CODIFICA"<<endl;
            Write_Encoding(out);

            cout<<"\nSCRITTURA ESEGUITA CON SUCCESSO"<<endl;

            return 1;}

            else {cout<<"FILE NOT FOUND"<<endl; return 0;}

    } else return 0;
}
```

Il metodo Encode, è il metodo che viene richiamato per codificare un file. Esso prende in input il nome del file da codificare e il nome che si vuole dare al file di output con estensione '.huff'.
*Sulle prime righe di codice viene richiamata il metodo **VerifiExt()**, che verifica la correttezza delle estensioni dei nomi dati in input. Poi viene creata un istanza del file di input e se questa viene trovata, partirà l'algoritmo di creazione della **mappa delle frequenze**. Successivamente viene chiuso il file e richiamato il metodo **CreateHuffmanTree()**, che restituirà il nodo dell'albero di Huffman....*

CreateHuffmanTree

```
Nodo* Huffman::CreateHuffmanTree() {
QueuePriority* coda = new QueuePriority(map_freq);
vector <Nodo*> vet;
vector <Nodo*>::iterator i;
Nodo* parent = new Nodo();
int s,d;
   while (coda->Getsize()>1) {


        s = coda->Extract_MIN();
        d = coda->Extract_MIN();
        Nodo* sx = new Nodo();
        Nodo* dx = new Nodo();

        //CONTROLLO SE ESISTE GIA IL MINIMO TRA I NODI REINSERITI
        i=vet.begin();
        while(i != vet.end()) {
              if  ((*i)->GetKey() == s ) { sx = *i; vet.erase(i); break;}
            i++;
        }

        i=vet.begin();
        while(i != vet.end()) {

            if  ((*i)->GetKey() == d ) {  dx = *i; vet.erase(i); break; }
            i++;
        }

        //SE IL MINIMO NON VIENE TROVATO VUOL DIRE CHE E' UN CARATTERE
        //QUINDI UNA FOGLIA, DI CONSEGUENZA GLI IMPOSTO LA FREQUENZA
        if (sx->GetKey()==-1) { sx->SetKey(s);  }
        if (dx->GetKey()==-1) { dx->SetKey(d);  }

        //CREO LA FREQUENZA DEL PADRE E LO ALLOCO
        int freq = sx->GetKey()+dx->GetKey();
        parent = new Nodo(freq,NULL,sx,dx,NULLO);
        //LO INSERISCO NEL VETTORE DEI NODI PADRI
        vet.push_back(parent);

        //INSERISCO LA FREQUENZA DEL PADRE NELLA CODA
        coda->Insert(freq);
       }
delete coda;
return parent;
}
```

Questo metodo viene richiamato dopo la costruzione della mappa delle frequenze. Prima di tutto viene creata un istanza della ***classe Queue****, che conterrà tutte le frequenze dei caratteri.*
Successivamente partirà un ciclo while che si fermerà solo quando la coda conterrà un solo elemento. In questo ciclo, verranno estratte le frequenze minime, creato un nodo padre con chiave uguale alla somma delle frequenze dei due minimi e reinserito nella coda.
*Da notare il vettore '****vet****', esso è usato come vettore di appoggio per tenere traccia dei figli dei nodi padre già estratti e poi reinseriti nella coda, altrimenti verrebbero persi creando un nuovo nodo ogni volta. Infine il metodo restituisce il nodo padre, radice dell'****albero di Huffman****.*

Il secondo metodo richiamato dal metodo Encode() è il metodo CreateMapCoding() che genera la mappa delle codifiche, vediamo com'è strutturato...

CreateMapCoding

```
void Huffman::CreateMapCoding(Nodo* x, string cod) {

        map<char, int>::iterator i;

    if ( x->GetSx() == NULL && x->GetDx() == NULL) {

        //CERCA FREQUENZA SU MAPPA
          i = map_freq.begin();
       while( i!=map_freq.end()) {
       if(i->second == x->GetKey())
       {
        map_coding[i->first] = cod;

        Treefile.push_back(1);
        Treefile.push_back((int)i->first);

       break;
       }
       i++; }

    } else
        {
            if(x->GetSx() != NULL){
                        cod.append("0");
                        Treefile.push_back(0);
                        CreateMapCoding(x->GetSx(),cod); }

            cod.erase(cod.end()-1);
            if(x->GetDx() != NULL){
                        cod.append("1");
                        Treefile.push_back(0);
                        CreateMapCoding(x->GetDx(),cod); }
            cod.erase(cod.end()-1);
        }

}
```

Questoo metodo fa uso dell'albero di Huffman per generare una mappa che conterrà la codifica di ogni carattere contenuto nel testo.
Il funzionamente è abbastanza banale:
esso controlla innanzitutto se il nodo corrente passato in input ***è una foglia****, se è così:*

> *trova il carattere con la frequenza corrispondente a alla frequenza del nodo foglia sulla mappa delle frequenze, e viene aggiunto alla mappa delle codifiche. La codifica è contenuta in una stringa passata in input ricorsivamente, a cui viene aggiunto 1 quando si scende a destra e 0 quando si scende a sinistra.*

Altrimenti, se ***non è una foglia****:*

> *Scendiamo sia a sinistra che a destra (solo se esiste) e aggiungiamo rispettivamente 0 e 1 alla stringa* ***cod****, in modo da salvarci ricorsivamente la codifica del nodo foglia in cui giungeremo.*

Da notare che in contemporanea a cod, viene costruita la stringa ***Treefile****, che conterrà la codifica dell'albero di Huffman, questa stringa registra ogni chiamata ricorsiva con uno 0 e quando incontra una foglia aggiunge un 1.*
Successivamente viene richiamato il metodo ***WriteEncoding****.*

WriteEncoding

```
bool Huffman::Write_Encoding(const char * name){
string bitcode;
long i = 0, ibit = 8, nbit = 0;
unsigned char ch = 0 , cc = 0;
//CONTINUA FINCHE L'I-ESIMO CARATTERE SARA DIVERSO DALL'ULTIMO
while (txt[i] != *txt.end()){

   ch = txt[i];//CH E' UGUALE ALL'I-ESIMO CARATTERE

   string tmp = map_coding[ch];//tmp contiene il char ch codificato sottoforma di stringa di 1 e 0

            //PER OGNI CARATTERE DI TMP
            for (unsigned int j=0; j<tmp.size(); j++) {
                unsigned char x;
                x = ((tmp[j] == '0') ? 0 : 1);
                cc = (cc | (x << --ibit));
                nbit++;
                if (ibit<=0) { ibit = 8; bitcode.push_back(cc); cc = 0;}
            }
   i++;
}

if(ibit>0) {bitcode.push_back(cc);}

comp = 100-((nbit*100)/(i*8));//nbit->numero di bit realmente scritti i->numero di byte del file di origine

ofstream SaveFile(name);
SaveFile <<nbit<<char(20)<<Treefile<<char(20)<<bitcode;
SaveFile.close();
cout<<"FILE CODIFICATO"<<endl;
cout<<"TASSO DI COMPRESSIONE -> "<<comp<<"%"<<endl;

return 1;
}
```

Questo è il metodo che si occuperà di scrivere su file (.huff) la codifica del file di testo dato in input. Sfrutta un ciclo while che scorre tutti i caratteri del file di testo, e per ogni carattere ne preleva la codifica dalla mappa delle codifiche e poi genera un char scrivendo bit per bit la sua rappresentazione. Grazie ad un indice 'ibit', che tiene traccia della posizione del bit corrente, ogni 8 bit (quindi ogni byte) viene scritto il carattere e resettato l'indice. Infine viene calcolato il tasso di compressione, come la differenza tra 100 e il numero di bit effettivamente scritti moltiplicato per 100 e diviso l'effettiva dimensione della stringa compressa scritta (che equivale al numro di char(1 byte) scritti per 8).

Nella parte finale del codice, si scrive sul file in ordine, prima il numero di bit scritti, poi si usa il 20esimo carattere del codice ASCII, cioè un "carattere non visualizzabile", come separatore, poi si aggiunge l'albero codificato in precedenza, un altro separatore e poi il testo codificato.

Metodi:DECODIFICA

Decode

```
bool Huffman::Decode(const char* name, const char* out){

if( VerifyExt(name,"huff") && VerifyExt(out,"txt") ){

return Write_Decoding(name,out); } else {return 0;}

}
```

Questo metodo pubblico non fa altro che verificare la correttezza dell'estensione dei nomi dei file dati in input e in caso affermativo richiama il metodo Write_Decoding.

Write_Decoding...(Prima parte)

```
bool Huffman::Write_Decoding(const char * ReadFile, const char* WriteFile){

    ifstream Rfile(ReadFile);

if (Rfile) {

    Rfile.seekg (0, Rfile.beg);

        //RECUPERO SIZE DEI BIT SCRITTI NELLA CODIFICA
        Rfile.ignore(1024,char(20));
        long y = Rfile.tellg();
        Rfile.seekg(Rfile.beg);
        char* size_txt = new char [y];
        Rfile.get(size_txt,y); long size = atol(size_txt);
        delete size_txt;

        cout<<"IL SIZE DEI BIT CODIFICATI E = "<<size<<endl;

        Rfile.ignore(1024,char(20)); Rfile.ignore(1024,char(20));//mi posizioni alla fine della codifica dell'albero
        long x = Rfile.tellg();//mi salvo la posizione
        char* tree = new char [x];//preparo un puntatore a char per salvare l'albero
        Rfile.seekg(y);//mi posiziono all'inizio dell'albero
        Rfile.read(tree,x-y-1);
        string treeS;
        for(long tmp=0; tmp<=(x-y-2);tmp++) treeS.push_back(int(tree[tmp]));
        delete tree;

        long ii = -1;
        root_huff=RebuildTree(treeS,ii);
        cout<<"ALBERO RICOSTRUITO..."<<endl;

        Rfile.seekg(x);
        char ch;
```

*In questa prima parte di codice del metodo Write_deconding, si verifica l'esistenza del file codificato con un if, se questo esiste ci si posiziona subito dopo il carattere 20 dell'ASCII, carattere usato come separatore dei dati all'interno del file di testo, per recuperare il size dei bit scritti precedentemente con il metodo **write_coding**. Allo stesso modo si prelevano i caratteri che rappresentano l'albero codificato, viene inserito in una stringa e passato al metodo **RebuildTree**, che si occuperà di ricostruire l'albero e restituire la sua radice.*

RebuildTree

```
Nodo* Huffman::RebuildTree (string tree, long &ix) {

 ix++;

 unsigned char ch = tree[ix];

  Nodo* n;
  Nodo* sx= NULL;
  Nodo* dx= NULL;

     if (ch)   return n = new Nodo(tree[++ix]);

     else { sx = RebuildTree(tree, ix); ch = tree[++ix];

      dx = RebuildTree(tree, ix); }


     return n = new Nodo(ix,NULL,sx,dx);
}
```

Questo metodo si occupa di ricostruire l'albero di Huffman a partire dalla sua stringa codificata. Esso fa uso di un contatore che useremo per estrarre l'ix-esimo carattere dalla stringa, se questo è 1 vuol dire che successivamente avremo un carattere, quindi istanziamo un nuovo nodo con chiave uguale al carattere. Al contrario, quando incontriamo 0, istanziamo due nodi, uno sinistro e uno destro, chiamando ricorsivamente il metodo e incrementando l'indice a dovere, all'uscita ritorneremo un nodo che avrà come figlio sinistro e figlio destro i nodi creati ricorsivamente.

WriteDecoding...(Seconda Parte)

```
        ////////////////////SCRIVO SU FILE////////////////////////////
        ofstream Wfile(WriteFile);

        long j = 0;
        const Nodo* nod = root_huff;

        while (!Rfile.eof()){

            Rfile.get(ch);

        /*PER OGNI BIT*/
        for (int i=7; i>=0 ; i--)

                    if (j<size) {


                        unsigned char bit = ((ch>>i) & 1);

                        if (bit) {nod = nod->GetDx();} else {nod = nod->GetSx();}

                        if (nod->GetSx() == NULL && nod->GetDx() == NULL) { Wfile<<char(nod->GetKey());
                                                                            nod = root_huff;}
                      j++;
                    }else continue;
        }
cout<<"FINE DECODIFICA"<<endl;

Rfile.close();
Wfile.close();

return 1;

  } else { cout<< "FILE NOT FOUND." <<endl; return 0; }

  }
```

Nella seconda parte del metodo WriteDecoding, viene creata un istanza del file su cui scrivere, e con un ciclo while che scorre tutti i caratteri codificati, viene eseguito un ciclo for che scorre tutti i bit del carattere i-esimo.
Partendo dalla radice dell'albero di Huffman, se il bit è 1 cambiamo il nodo corrente con il figlio destro, al contrario se il bit è 0 lo cambiamo con il figlio sinistro. Quando i figli del nodo corrente sono entrambi NULL, e quindi siamo su una foglia, scrive il carattere contenuto dal nodo sul file e reimposta il nodo corrente uguale alla radice dell'albero.
Il contenuto di questo ciclo for, verrà eseguito solo finché un indice j sarà minore o uguale del size dei bit effettivamente scritti. Questo comportamento dell'algoritmo è dovuto al fatto che l'ultimo carattere codificato, potrebbe essere stato scritto 'per metà' , nel senso che se i bit scritti non sono un multiplo di 8, l'ultimo carattere non rappresenterà una codifica per intero, ma solo una sua parte lo sarà. Nelle ultime righe di codice, si liberano/deallocano le varie strutture dati.

Analisi del main

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

{
    if (argc != 4) {
                                cout<<"\n\nINSERIRE UN NUMERO DI ARGOMENTI PARI A 4\nSintassi: "<< argv[0];
                                cout<<" operazione nomefileinput nomefileoutput\nOperazioni disponibili: ";
                                cout<<"-c per compressione\t-d per decompressione\n"<<endl;
                                exit(-1); }

    if ( strcmp(argv[1],"-c") == 0 ){
                cout<<"\n\ncodifica in corso del file "<<argv[2]<<"...\n"<<endl;
                Huffman::get_instance().Encode(argv[2],argv[3]);
            }
    else if ( strcmp(argv[1], "-d") == 0) {
                cout<<"\n\ndecodifica in corso del file "<<argv[2]<<"...\n"<<endl;
                Huffman::get_instance().Decode(argv[2],argv[3]);
            }
    else {
            cout<<"\n\nERRORE! IL PRIMO ARGOMENTO DEVE ESSERE:\n-c per la compressione\t-d per la decompressione\n"<<endl;
            exit(-1);
            }


 return 0;

}
```

Nel main viene creata un interfaccia minimale per gestire il programma.
Il programma prende in input 3 parametri da linea di comando:

- *la modalità (-**c** per la codifica e -**d** per la decodifica)*
- *nome del file in input*
- *nome del file in output*

*Inoltre viene effettuato un controllo sull'input per evitare di dare un input sbagliato. Si noti anche l'uso della classe **Singleton Huffman**, usando il metodo **get_instance()**, che restituisce l'unica istanza della classe generata durante l'intera esecuzione del programma.*

Esempio di codifica

Mostriamo un semplice test che rappresenta l'esecuzione del programma.

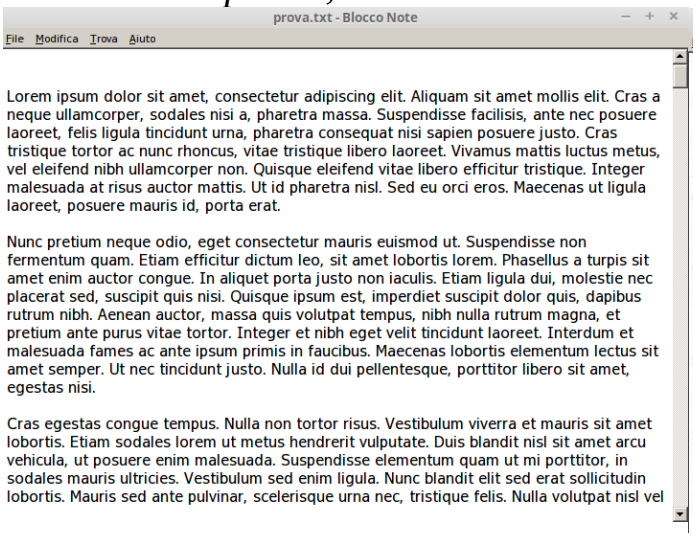
```
max@max-Satellite-A350 ~ $ ./progettohuff -c prova.txt prova.huff


codifica in corso del file prova.txt...

LETTURA FILE...

COSTRUZIONE ALBERO DI HUFFMAN

COSTRUZIONE MAPPA CODIFICHE

INIZIO SCRITTURA CODIFICA
FILE CODIFICATO
TASSO DI COMPRESSIONE -> 47%

SCRITTURA ESEGUITA CON SUCCESSO
```

Viene dato in input il file prova.txt, un file contenente 500 paragrafi, per un totale di 61,8kB.

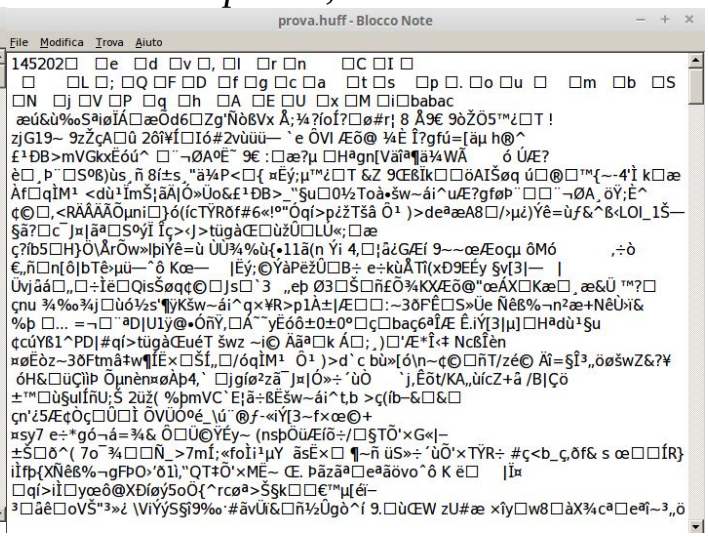
Il file restituito in output si chiamerà prova.huff.

Come si evince dallo screen, il tasso di compressione ottenuto in questo caso è del 47%.

File di input 61,8kB



File di output 33,3kB



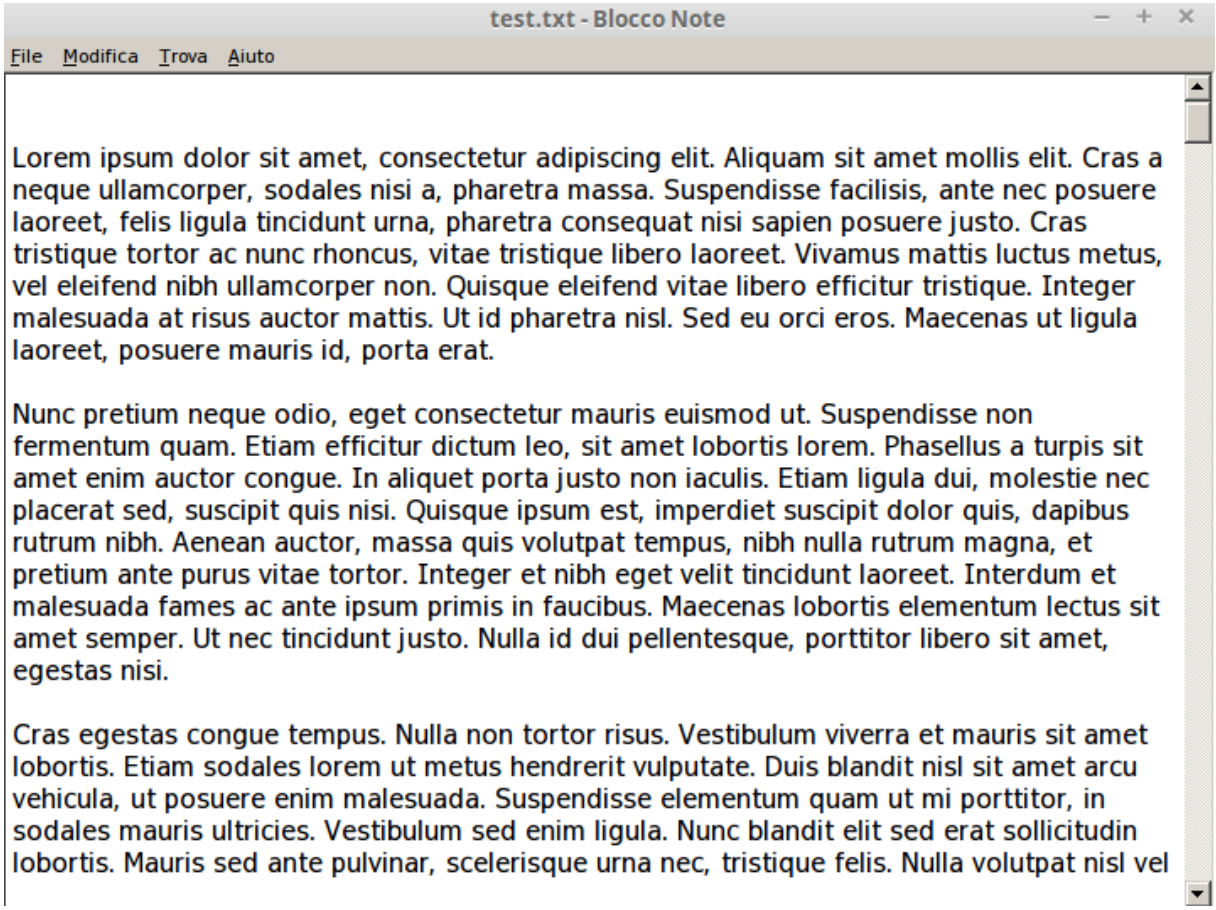
Potrebbe risultare strano che la proporzione delle dimensioni del file di output con quello di input non rispecchia un tasso del 47%. Questo perché l'effettivo tasso di compressione è calcolato solo sui bit che realmente rappresentano la codifica, a questi poi vanno aggiunti i bit per rappresentare l'albero e i caratteri usati per tenere il conteggio dei bit scritti.

Esempio di decodifica

Il processo inverso del precedente è la decodifica, vediamolo brevemente.

```
max@max-Satellite-A350 ~ $ ./progettohuff -d prova.huff test.txt


decodifica in corso del file prova.huff...

IL SIZE DEI BIT CODIFICATI E = 265117
ALBERO RICOSTRUITO...
FINE DECODIFICA
```

File di output



Codice

Colore.h

```
#ifndef COLORE_H_INCLUDED
#define COLORE_H_INCLUDED

#include <iostream>

using namespace std;

enum hide_color
 { RED, BLACK, NULLO } ;

class Colore
{
   private:
      hide_color value;

   public:
      Colore(hide_color c = NULLO) { value=c; }

      hide_color get() { return value; }
        void change() { if (value!=NULLO) { value = ((value == RED) ?
BLACK : RED); } else { cout<<"COLORE NULLO"<<endl; }}
      void set(hide_color x) { value = x;}
      void set(Colore x) { value = x.get();}


      bool operator ==( hide_color x ) { return( value == x ) ; }
      bool operator !=( hide_color x ) { return( value != x ) ; }



};
#endif
```

Nodo.h

```
#ifndef NODO_H_INCLUDED
#define NODO_H_INCLUDED

using namespace std;


#include "Colore.h"

class Nodo {
   private:

      int key;
      Nodo* parent;
      Nodo* sx;
      Nodo* dx;
      Colore color;

   public:

       Nodo (int k = -1, Nodo* p = NULL, Nodo* s = NULL, Nodo* d = NULL, Colore c=NULLO): key (k), parent(p), sx (s), dx (d), color(c) {}
           Nodo (Nodo* copia): key(copia->key), parent(copia->parent), sx(copia->sx), dx(copia->dx){color=copia->GetColor().get();}

      //METODI GET
      Nodo* GetParent() const {return parent;}
      Nodo* GetSx() const {return sx;}
      Nodo* GetDx() const {return dx;}
      int   GetKey() const {return key;}
      Colore GetColor() const {return color;}
      void SetColor(hide_color x) {color = x;}
      void SetColor(Colore x) {color = x;}
      //METODI SET
      Nodo* SetParent(Nodo* p) {return parent=p;}
      Nodo* SetSx(Nodo* s) {return sx=s;}
      Nodo* SetDx(Nodo* d) {return dx=d;}
      int   SetKey(int k) {return key=k;}


          };

#endif // NODO_H_INCLUDED
```

ABR.h

```
#ifndef ABR_H_INCLUDED
#define ABR_H_INCLUDED

#include <vector>
#include "Nodo.h"

class ABR {

   protected:

      Nodo* root;
      Nodo* NIL; //NODO SENTINELLA

      int size;

      Nodo* add (Nodo* x);
      Nodo* del (Nodo* a);
      Nodo* cerca (Nodo* a, const int i);

   public:
      void stampa ( Nodo* a);
      ABR () {NIL = new Nodo(); root = NIL; size = 0;}
        ABR (int x) { NIL = new Nodo(); root = new Nodo(x,NIL,NIL,NIL);
size=1; }
      void liberaABR (Nodo* a);
      virtual ~ABR() {liberaABR(root);}
      Nodo* GetRoot() {return root;}
      Nodo* GetNIL() {return NIL;}
      Nodo* GetMin (Nodo* a);
      Nodo* GetMax (Nodo* a);
      int GetSize() {return size;}
      virtual Nodo* ins (Nodo* &a) { return add (a);}
      virtual Nodo* canc (Nodo* &a) {  return del (a);}

    bool esiste (const int i) { bool value = ((cerca (root, i) == NIL) ? 0 : 1);
return value; }
      void Print () {stampa(root);}
};


#endif // ABR_H_INCLUDED
```

ABR.cpp

```
#include "ABR.h"

Nodo* ABR::GetMin(Nodo* a){
        if (a == NIL)
         return NULL;
        else if (a->GetSx() == NIL) return a;
        else return GetMin(a->GetSx());
                                }
Nodo* ABR::GetMax (Nodo* a){
        if (a == NIL)
         return NULL;
        else if (a->GetDx() == NIL) return a;
        else return GetMax(a->GetDx());
                                }

Nodo* ABR::add(Nodo* a){
size ++;
   Nodo* x = root;
   Nodo* y = NIL;

   while ( x!= NIL) {

      y=x;

      if ( a->GetKey() < x->GetKey() )
         x=x->GetSx();
      else
         x=x->GetDx();
   }


   a->SetParent(y);

   if ( y == NIL )
      root = a;
   else
      if ( a->GetKey() < y->GetKey() )
         y->SetSx(a);
      else
```

```
        y->SetDx(a);



      return a;
        }

Nodo* ABR::cerca(Nodo* a, const int i) {

    if (a == NIL) return NIL;

    else if (i == a->GetKey()) return a;

    else if (i < a->GetKey()) return cerca (a->GetSx(), i);

    else  return cerca (a->GetDx(), i);
 }

 Nodo* ABR::del(Nodo* a){
                size--;
                Nodo* x = new Nodo();
                Nodo* y = new Nodo();
                /* UNO DEI DUE FIGLI O ENTRAMBI SONO NIL */
                if (a->GetSx() == NIL || a->GetDx() == NIL)
                    y = a;
                else
                        y = GetMin(a->GetDx());//ENTRAMBI I FIGLI
ESISTONO, QUINDI TROVA SUCCESSORE

                if (y->GetSx() != NIL)//SE IL SX ESISTE
                    x = y->GetSx();
                else
                   x = y->GetDx();


                     if (x != NIL)//SE X ESISTE, IMPOSTA IL PADRE
UGUALE AL PADRE DI Y
                     x->SetParent(y->GetParent());


                    if (y->GetParent() == NIL) //SE IL PADRE DI Y E'
UGUALE A NIL X E' LA RADICE
                   root = x;
```

```
                else {
                    /*CASO DI UN SOLO FIGLIO A NIL*/
                    if (y==y->GetParent()->GetSx())
                        y->GetParent()->SetSx(x);
                    else
                            y->GetParent()->SetDx(x);
                if (y!=a)
                 a->SetKey(y->GetKey());


                }

            return a;
}

void ABR::liberaABR (Nodo* a){
    if (a == NIL)
      delete a;
    else {
     if (a->GetSx() != NIL)
      liberaABR(a->GetSx());
     if (a->GetDx() != NIL)
      liberaABR(a->GetDx());
      delete a;
       a = NIL;
            }
}

void ABR::stampa ( Nodo* a){

   if ( a != NIL ) {

   vector <Nodo*> coda;

   Nodo* x = new Nodo(a);

   coda.push_back(x);
   cout<<"%%%%%%%%%%%%%%%%%%%%"<<endl;
   cout<<"%%%%%%STAMPA ALBERO%%%%%%%%"<<endl;
   cout<<"%%%%%%%%%%%%%%%%%%%%"<<endl;

    cout<<"SIZE DELL'ALBERO: "<<size<<endl;
```

```
        while (coda.size()>0) {

        x = coda[0];
        coda.erase(coda.begin());


         string value = ((x->GetColor().get() == 0) ? "RED" : "BLACK");
        cout<<"NODO: "<<x->GetKey()<<"  "<<char(x->GetKey())<<" --
COLORE: "<<value<<endl;

         if (x->GetSx() != NULL) {cout<<"nodo sinistra = "<< x->GetSx()-
>GetKey()<<"                    "<<char(x->GetSx()->GetKey())<<endl;
coda.push_back(x->GetSx());}


         if (x->GetDx() != NULL) {cout<<"nodo destra  = "<< x->GetDx()-
>GetKey()<<"                    "<<char(x->GetDx()->GetKey())<<endl;
coda.push_back(x->GetDx());}
        cout<<"====================="<<endl;
        cout<<"====================="<<endl;
        }
        }
        else
           { cout<<"ALBERO VUOTO"<<endl;}
    }
```

ARB.h

```
#ifndef ARB_H_INCLUDED
#define ARB_H_INCLUDED

#include "ABR.h"

class ARB : public ABR
{   private:
      void rotate_sx(Nodo* x);
      void rotate_dx(Nodo* x);
      void InsertFixUP(Nodo* x);
      Nodo* del(Nodo* &a);
      void DeleteFixUP(Nodo* &x);

    public:
```

```
    ARB():ABR() {root->SetColor(BLACK); NIL->SetColor(NULLO); }
                ARB(int x):ABR(x) { root->SetColor(BLACK); NIL->SetColor(NULLO);}
    Nodo* ins(Nodo* &a);
    Nodo* canc (Nodo* &a);
};

#endif // ARB_H_INCLUDED
```

ARB.cpp

```
#include "ARB.h"

void ARB::rotate_sx(Nodo* x) {

   Nodo* y = x->GetDx();
   x->SetDx(y->GetSx());
   if (y->GetSx() != NIL ) y->GetSx()->SetParent(x);

   y->SetParent(x->GetParent());


       if( x->GetParent() == NIL )
             root = y;
       else
   {

             if( x == x->GetParent()->GetSx())
                   x->GetParent()->SetSx(y);
             else
                   x->GetParent()->SetDx(y);
       }


       y->SetSx(x);
       x->SetParent(y);
}

void ARB::rotate_dx(Nodo* x) {
  Nodo* y = x->GetSx();
   x->SetSx(y->GetDx());
   if (y->GetDx() != NIL ) y->GetDx()->SetParent(x);
```

```
    y->SetParent(x->GetParent());


        if( x->GetParent() == NIL )
                root = y;
        else
    {
       if( x == x->GetParent()->GetDx())
                        x->GetParent()->SetDx(y);
                else
                        x->GetParent()->SetSx(y);
        }


        y->SetDx(x);
        x->SetParent(y);
}


Nodo* ARB::ins(Nodo* &x){
//RESETTO NODO DA INSERIRE
x->SetColor(RED);
x->SetSx(NIL);
x->SetDx(NIL);
x->SetParent(NIL);

add(x);

InsertFixUP(x);

return x;
}


void ARB::InsertFixUP(Nodo* x) {

while ( x->GetParent()->GetColor() == RED) {

        if(x->GetParent() == x->GetParent()->GetParent()->GetSx()) {

        Nodo* y = x->GetParent()->GetParent()->GetDx();

           if(y->GetColor() == RED) {
```

```
            x->GetParent()->SetColor(BLACK);
            y->SetColor(BLACK);
            x->GetParent()->GetParent()->SetColor(RED);
            x=x->GetParent()->GetParent();
            }
         else
         {
            if(x==x->GetParent()->GetDx()){
                  x=x->GetParent();
                  rotate_sx(x);
            }

            x->GetParent()->SetColor(BLACK);
            x->GetParent()->GetParent()->SetColor(RED);
            x->GetParent()->GetDx()->SetColor(BLACK);
            rotate_dx(x->GetParent()->GetParent());
         }
      }

      else

      {

        Nodo* y=x->GetParent()->GetParent()->GetSx();

         if( y->GetColor()== RED){
            x->GetParent()->SetColor(BLACK);
            y->SetColor(BLACK);
            x->GetParent()->GetParent()->SetColor(RED);
            x=x->GetParent()->GetParent();
            }
         else{
            if(x==x->GetParent()->GetSx()){
               x=x->GetParent();
               rotate_dx(x);}
               x->GetParent()->SetColor(BLACK);
               x->GetParent()->GetParent()->SetColor(RED);
               x->GetParent()->GetSx()->SetColor(BLACK);
               rotate_sx(x->GetParent()->GetParent());
            }
      }
   }
```

```
root->SetColor(BLACK);

}

Nodo* ARB::canc (Nodo* &a) {

if (a != NIL)
   { return del(a);}
else
      {cout<<"IL NODO CON CHIAVE "<< a->GetKey() <<" NON
ESISTE"<<endl; return NIL;}
}

void ARB::DeleteFixUP(Nodo* &x) {

{
   while(x != root && x->GetColor() == BLACK )
   {

      Nodo* n;
            if( x->GetParent()->GetSx() == x )
            {
                  n = x->GetParent()->GetDx();


                  if( n->GetColor() == RED)
                  {
                        n->SetColor(BLACK);
                        x->GetParent()->SetColor(RED);
                        rotate_sx(x->GetParent());
                        n = x->GetParent()->GetDx();
                  }


                  if(n->GetSx()->GetColor() == BLACK && n->GetDx()-
>GetColor() == BLACK )
                  {
                        n->SetColor(RED);
                        x = x->GetParent();
                  }
                  else
                  {
                        if( n->GetDx()->GetColor() == BLACK )
```

```
                {
                        n->GetSx()->SetColor(BLACK);
                        n->SetColor(RED);
                        rotate_dx(n);
                        n = x->GetParent()->GetDx();

                }

                n->SetColor(x->GetParent()->GetColor());
                x->GetParent()->SetColor(BLACK);
                n->GetDx()->SetColor(BLACK);
                rotate_sx(x->GetParent());
                x = root;
            }
        }

        else
    {   n = x->GetParent()->GetSx();

            if( n->GetColor() == RED )
            {
                n->SetColor(BLACK);
                x->GetParent()->SetColor(RED);
                rotate_dx(x->GetParent());
                n = x->GetParent()->GetSx();
            }

            if(n->GetDx()->GetColor() == BLACK && n->GetSx()-
>GetColor() == BLACK)
            {
                n->SetColor(RED);
                x = x->GetParent();
            }
            else
            {
                if( n->GetSx()->GetColor() == BLACK )
                {
                    n->GetDx()->SetColor(BLACK);
                    n->SetColor(RED);
                    rotate_sx(n);
                    n = x->GetParent()->GetSx();
                }
                n->SetColor(x->GetParent()->GetColor());
```

```
                        x->GetParent()->SetColor(BLACK);


                        n->GetSx()->SetColor(BLACK);

                        rotate_dx(x->GetParent());
                        x = root;
                    }
                }
            }
            x->SetColor(BLACK);
}

}

Nodo* ARB::del(Nodo* &a){

                    size--;
                    Nodo* x = new Nodo();
                    Nodo* y = new Nodo();

                    /* UNO DEI DUE FIGLI O ENTRAMBI SONO NIL */
                    if (a->GetSx() == NIL || a->GetDx() == NIL)
                        y = a;
                    else
                            y = GetMin(a->GetDx());//ENTRAMBI I FIGLI
ESISTONO, QUINDI TROVA SUCCESSORE

                    if (y->GetSx() != NIL)//SE IL SX ESISTE
                        x = y->GetSx();
                    else
                       x = y->GetDx();


                   //IMPOSTA IL PADRE UGUALE AL PADRE DI Y
                    x->SetParent(y->GetParent());


                        if (y->GetParent() == NIL) //SE IL PADRE DI Y E'
UGUALE A NIL X E' LA RADICE
                       root = x;
```

```
                else {
                    /*CASO DI UN SOLO FIGLIO A NIL*/
                    if (y==y->GetParent()->GetSx())
                        y->GetParent()->SetSx(x);
                    else
                        y->GetParent()->SetDx(x);

                    if (y!=a)
                       a->SetKey(y->GetKey());


                }
                    //SE IL NODO CANCELLATO ERA NERO, CHIAMA
DELETEFIXUP PER RIPRISTINARE LE PROPRIETA' DELL'ALBERO
                if (y->GetColor()==BLACK ) { DeleteFixUP(x);}

            return a;
    }
```

QueuePriority.h

```
#ifndef QueuePriority_H_INCLUDED
#define QueuePriority_H_INCLUDED

#include <map>

#include "ARB.h"

class QueuePriority
{
   private:
      ARB* Albero;
   public:
```

```
    QueuePriority(const int i){ Albero = new ARB(i); }
    QueuePriority(map<char,int> x);
    QueuePriority(){ Albero = new ARB(); }
    ~QueuePriority(){ delete Albero;};


    bool Insert(const int i);

    int Getsize() { return Albero->GetSize(); }

    int Extract_MIN();
    int Extract_MAX();

    void Printcoda() { Albero->Print(); }


};

#endif // QueuePriority_H_INCLUDED
```

Queue.cpp

```
#include "QueuePriority.h"

QueuePriority::QueuePriority(map < char, int > x)
{
   map<char, int>::iterator i = x.begin();
   Albero = new ARB(i->second); i++;
   while(i!=x.end()) {
   Insert(i->second);
   i++;
   }


}

bool QueuePriority::Insert(const int i)
{
Nodo* x = new Nodo(i, Albero->GetNIL(),Albero->GetNIL(),Albero-
>GetNIL(),RED);
if (x->GetKey()<1)
```

```
        {cout<<"IMPOSSIBILE INSERIRE ELEMENTO CON CHIAVE MINORE DI 1"<<endl; return 0;}

Albero->ins(x);

return 1;
}

int QueuePriority::Extract_MIN()
{

   Nodo* tmp = Albero->GetMin(Albero->GetRoot());
   int x = tmp->GetKey();
   Albero->canc(tmp);
   delete tmp;
   return x;
}

int QueuePriority::Extract_MAX()
{
   Nodo* tmp = Albero->GetMax(Albero->GetRoot());
   int x = tmp->GetKey();
   Albero->canc(tmp);

   return x;
}
```

Huffman.h

```
#ifndef HUFFMAN_H_INCLUDED
#define HUFFMAN_H_INCLUDED

#include <vector>
#include <stdlib.h>
#include <fstream>

#include "QueuePriority.h"

class Huffman
{
   private:

      Huffman(){};
      ~Huffman();

         map<char, string> map_coding; //MAPPA CHE CONTIENE LA CODIFICA DI OGNI CARATTERE
            map<char, int> map_freq; //MAPPA CHE CONTIENE LA FREQUENZA DI OGNI CARATTERE
      Nodo* root_huff; //NODO RADICE DELL'ALBERO DI HUFFMAN

      string Treefile; //STRINGA CHE CONTIENE LA SERIALIZZAZIONE DELL'ALBERO
      string txt; //STRINGA CHE CONTIENE IL TESTO DEL FILE
      int comp; //TASSO DI COMPRESSIONE

      //METODI

      //STAMPA UNA COPPIA DI VALORI DI UNA MAPPA
        void PrintCouple(const pair<const char,string>& r) {cout<<" char = "<<r.first<<"  cod = "<<r.second<<endl;}

      Nodo* CreateHuffmanTree();//CREA ALBERO DI HUFFMAN
      void CreateMapCoding(Nodo* x, string cod);//CREA MAPPA DELLE CODIFICHE

            bool Write_Encoding(const char* name);//SCRIVE IL FILE CODIFICATO
             bool Write_Decoding(const char * ReadFile, const char* WriteFile);//SCRIVE IL FILE DECODIFICATO
```

```
        Nodo* RebuildTree (string tree, long &i);//RICOSTRUISCE L'ALBERO A PARTIRE DALLA SUA STRINGA CODIFICATA

        bool VerifyExt(const char* name, const char* x);//VERIFICA L'ESTENSIONE DEI NOMI DEI FILE PASSATI IN INPUT

        Nodo* GetRoot() { return root_huff;}//RESTITUISCE LA RADICE DELL'ALBERO DI HUFFMAN

        void free_HuffmanTree(Nodo* x);



    public:

        //CREA UN ISTANZA STATICA DELLA CLASSE AL SUO PRIMO RICHIAMO. SARA' LA STESSA PER TUTTA LA DURATA DEL PROGRAMMA
        static Huffman& get_instance() { static Huffman instance;  return instance;  }

        //STAMPA MAPPA DELLE CODIFICHE
                void printMAP() { map<char, string>::const_iterator i; for(i=map_coding.begin(); i!=map_coding.end(); i++) PrintCouple(*i); }

        bool Encode(const char* name, const char* out);
        bool Decode(const char* name, const char* out);


};


#endif // HUFFMAN_H_INCLUDED
```

Huffman.cpp

```
#include "Huffman.h"


Huffman::~Huffman(){
        map_freq.clear();
        map_coding.clear();
        Treefile.clear();
        txt.clear();
        free_HuffmanTree(root_huff);
        }

bool Huffman::VerifyExt(const char* name, const char* x){
string  str = name;

size_t pos = str.rfind('.');
   if(pos != std::string::npos)
   { std::string ext = str.substr(++pos);
      if(ext != x){
              cout << "ERRORE, IL NOME "<<name<<" NON E' NEL
FORMATO CORRETTO.\nINSERIRE NOME CON
ESTENSIONE ."<<x<<endl; return 0;}

    } else { cout << "ERRORE, IL NOME "<<name<<" NON E' NEL
FORMATO CORRETTO.\nINSERIRE NOME CON
ESTENSIONE ."<<x<<endl; return 0;}

   return 1;
}

void Huffman::free_HuffmanTree (Nodo* a){
       if (a == NULL)
         delete a;
       else {
        if (a->GetSx() != NULL)
         free_HuffmanTree(a->GetSx());
        if (a->GetDx() != NULL)
         free_HuffmanTree(a->GetDx());
         delete a;
          a = NULL;
              }
   }
```

```
bool Huffman::Encode(const char* name, const char* out){

if( VerifyExt(name,"txt") && VerifyExt(out,"huff") ) {
      ifstream file(name);
         if (file) {
            char ch = file.get();

            cout<<"LETTURA FILE..."<<endl;
            while(!file.eof()){
               txt.push_back(ch);
               map_freq[ch]++;
               file.get(ch);
            }

            file.close();

         cout<<"\nCOSTRUZIONE ALBERO DI HUFFMAN"<<endl;
         root_huff = CreateHuffmanTree();

         string codd;
         cout<<"\nCOSTRUZIONE MAPPA CODIFICHE"<<endl;
         CreateMapCoding(root_huff,codd);

         cout<<"\nINIZIO SCRITTURA CODIFICA"<<endl;
         Write_Encoding(out);

         cout<<"\nSCRITTURA ESEGUITA CON SUCCESSO"<<endl;

         return 1;}

         else {cout<<"FILE NOT FOUND"<<endl; return 0;}

    } else return 0;
}

bool Huffman::Decode(const char* name, const char* out){

if( VerifyExt(name,"huff") && VerifyExt(out,"txt") ){

return Write_Decoding(name,out); } else {return 0;}

}
```

```
Nodo* Huffman::CreateHuffmanTree() {
QueuePriority* coda = new QueuePriority(map_freq);
vector <Nodo*> vet;
vector <Nodo*>::iterator i;
Nodo* parent = new Nodo();
int s,d;
  while (coda->Getsize()>1) {


      s = coda->Extract_MIN();
      d = coda->Extract_MIN();
      Nodo* sx = new Nodo();
      Nodo* dx = new Nodo();

            //CONTROLLO SE ESISTE GIA IL MINIMO TRA I NODI
REINSERITI
      i=vet.begin();
      while(i != vet.end()) {
          if  ((*i)->GetKey() == s ) { sx = *i; vet.erase(i); break;}
        i++;
      }

      i=vet.begin();
      while(i != vet.end()) {

        if  ((*i)->GetKey() == d ) {  dx = *i; vet.erase(i); break; }
        i++;
      }

       //SE IL MINIMO NON VIENE TROVATO VUOL DIRE CHE E' UN
CARATTERE
       //QUINDI UNA FOGLIA, DI CONSEGUENZA GLI IMPOSTO LA
FREQUENZA
      if (sx->GetKey()==-1) { sx->SetKey(s);  }
      if (dx->GetKey()==-1) { dx->SetKey(d);  }

      //CREO LA FREQUENZA DEL PADRE E LO ALLOCO
      int freq = sx->GetKey()+dx->GetKey();
      parent = new Nodo(freq,NULL,sx,dx,NULLO);
      //LO INSERISCO NEL VETTORE DEI NODI PADRI
      vet.push_back(parent);

      //INSERISCO LA FREQUENZA DEL PADRE NELLA CODA
```

```
        coda->Insert(freq);
       }
delete coda;
return parent;
}

void Huffman::CreateMapCoding(Nodo* x, string cod) {

       map<char, int>::iterator i;

   if ( x->GetSx() == NULL && x->GetDx() == NULL) {

       //CERCA FREQUENZA SU MAPPA
         i = map_freq.begin();
      while( i!=map_freq.end()) {
      if(i->second == x->GetKey())
      {
              //cout<<" TROVATO -> "<<i->first<<" FREQ -> "<<i-
>second<<" COD -> "<<cod<<endl;
       map_coding[i->first] = cod;

       Treefile.push_back(1);
       Treefile.push_back((int)i->first);

      // cout<<"TREEFILE -> "<<Treefile<<endl;
       map_freq.erase(i);

       break;
      }
      i++; }

      //ASSEGNA PUNTATORE A CHAR

   } else
      {
         if(x->GetSx() != NULL){
                cod.append("0");
                Treefile.push_back(0);
                CreateMapCoding(x->GetSx(),cod); }

         cod.erase(cod.end()-1);
         if(x->GetDx() != NULL){
                cod.append("1");
```

```
                Treefile.push_back(0);
                CreateMapCoding(x->GetDx(),cod); }
        cod.erase(cod.end()-1);
    }


}

Nodo* Huffman::RebuildTree (string tree, long &ix) {

 ix++;

 unsigned char ch = tree[ix];

  Nodo* n;
  Nodo* sx= NULL;
  Nodo* dx= NULL;

    if (ch)  return n = new Nodo(tree[++ix]);

    else { sx = RebuildTree(tree, ix); ch = tree[++ix];

     dx = RebuildTree(tree, ix); }


    return n = new Nodo(ix,NULL,sx,dx);
}

bool Huffman::Write_Encoding(const char * name){
string bitcode;
long i = 0, ibit = 8, nbit = 0;
unsigned char ch = 0 , cc = 0;
//CONTINUA FINCHE L'I-ESIMO CARATTERE SARA DIVERSO DALL'ULTIMO
while (txt[i] != *txt.end()){

   ch = txt[i];//CH E' UGUALE ALL'I-ESIMO CARATTERE

      string tmp = map_coding[ch];//tmp contiene il char ch codificato sottoforma di stringa di 1 e 0

          //PER OGNI CARATTERE DI TMP
          for (unsigned int j=0; j<tmp.size(); j++) {
```

```
            unsigned char x;
            x = ((tmp[j] == '0') ? 0 : 1);
            cc = (cc | (x << --ibit));
            nbit++;
            if (ibit<=0) { ibit = 8; bitcode.push_back(cc); cc = 0;}
          }
  i++;
}

if(ibit>0) {bitcode.push_back(cc);}

comp = 100-((nbit*100)/(i*8));//nbit->numero di bit realmente scritti i->numero di byte del file di origine

ofstream SaveFile(name);
SaveFile <<nbit<<char(20)<<Treefile<<char(20)<<bitcode;
SaveFile.close();
cout<<"FILE CODIFICATO"<<endl;
cout<<"TASSO DI COMPRESSIONE -> "<<comp<<"%"<<endl;

return 1;
}

bool Huffman::Write_Decoding(const char * ReadFile, const char* WriteFile){

   ifstream Rfile(ReadFile);

if (Rfile) {


   Rfile.seekg (0, Rfile.beg);


      //RECUPERO SIZE DEI BIT SCRITTI NELLA CODIFICA
      Rfile.ignore(1024,char(20));
      long y = Rfile.tellg();
      Rfile.seekg(Rfile.beg);
      char* size_txt = new char [y];
      Rfile.get(size_txt,y); long size = atol(size_txt);
      delete size_txt;

      cout<<"IL SIZE DEI BIT CODIFICATI E = "<<size<<endl;
```

```
        Rfile.ignore(1024,char(20));  Rfile.ignore(1024,char(20));//mi posizioni alla fine della codifica dell'albero
    long x = Rfile.tellg();//mi salvo la posizione
     char* tree = new char [x];//preparo un puntatore a char per salvare l'albero
    Rfile.seekg(y);//mi posiziono all'inizio dell'albero
    Rfile.read(tree,x-y-1);
    string treeS;
                                for(long  tmp=0;  tmp<=(x-y-2);tmp++) treeS.push_back(int(tree[tmp]));
    delete tree;


    long ii = -1;
    root_huff=RebuildTree(treeS,ii);
    cout<<"ALBERO RICOSTRUITO..."<<endl;

    Rfile.seekg(x);
    char ch;

    ////////////////////SCRIVO SU FILE//////////////////////////
    ofstream Wfile(WriteFile);

    long j = 0;
    const Nodo* nod = root_huff;


    while (!Rfile.eof()){

       Rfile.get(ch);

    /*PER OGNI BIT*/
    for (int i=7; i>=0 ; i--)

            if (j<size) {


               unsigned char bit = ((ch>>i) & 1);

               if (bit) {nod = nod->GetDx();} else {nod = nod->GetSx();}
```

```
                    if (nod->GetSx() == NULL && nod->GetDx() == NULL)
{ Wfile<<char(nod->GetKey());
                                                    nod = root_huff;}
                 j++;
                }else continue;

        }
        ////////////////////////////////////////////
cout<<"FINE DECODIFICA"<<endl;

Rfile.close();
Wfile.close();
map_freq.clear();
map_coding.clear();
Treefile.clear();
txt.clear();
free_HuffmanTree(root_huff);
return 1;

  } else { cout<< "FILE NOT FOUND." <<endl; return 0; }

  }
```

Main.cpp

```
#include "Huffman.h"
#include "string.h"

using namespace std;

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

{
   if (argc != 4) {
  cout<<"\n\nINSERIRE UN NUMERO DI ARGOMENTI PARI A 4\nSintassi: "<< argv[0];
cout<<" operazione nomefileinput nomefileoutput\nOperazioni disponibili: ";
 cout<<"-c per compressione\t-d per decompressione\n"<<endl;
                        exit(-1); }
```

```
   if ( strcmp(argv[1],"-c") == 0 ){
                                    cout<<"\n\ncodifica in corso del file "<<argv[2]<<"...\n"<<endl;
            Huffman::get_instance().Encode(argv[2],argv[3]);
         }
   else if ( strcmp(argv[1], "-d") == 0) {
                                    cout<<"\n\ndecodifica in corso del file "<<argv[2]<<"...\n"<<endl;
            Huffman::get_instance().Decode(argv[2],argv[3]);
         }
   else {
                  cout<<"\n\nERRORE! IL PRIMO ARGOMENTO DEVE ESSERE:\n-c per la compressione\t-d per la decompressione\n"<<endl;
         exit(-1);
         }


 return 0;

}
```