UNIVERSITÀ DEGLI STUDI DI UDINE

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

RELAZIONE PROGETTO DI ALGORITMI E STRUTTURE DATI

A cura di:

- Raffaele Marseu, matricola 133879
  marseu.raffaele@spes.uniud.it
- Maksym Sernyak, matricola 143087
  sernyak.maksym@spes.uniud.it
- Danjel Stefani, matricola 142863
  stefani.danjel@spes.uniud.it
- Marco Venir, matricola 144762
  venir.marco@spes.uniud.it

Anno Accademico 2019/2020

Indice

1. Introduzione Pag. 1

2. Presentazione algoritmi Pag. 1
   - 2.1. Metodi in comune Pag. 1
   - 2.2. Alberi binari di ricerca semplici Pag. 4
   - 2.3. Alberi binari di ricerca di tipo AVL Pag. 6
   - 2.4. Alberi binari di ricerca di tipo Red-Black Pag. 8

3. La funzione "*Creatore*" Pag. 13

4. La funzione *"Generatore"* Pag. 14

5. Il metodo per misurare il tempo Pag. 14

6. Presentazione dei risultati e considerazioni Pag. 20

1.Introduzione

Abbiamo realizzato i programmi in Java sul programma Intellij, come ambiente di sviluppo e di run dei programmi. Abbiamo usato un computer con uno Xeon W3680 (3.33 GHz di base, 3.60 GHz Intel TurboBoost 1.0) con 6 core e 12 threads, equipaggiato con 20gb di ram DDR3 1333 MHz di velocità. Il sistema operativo usato e' Windows 10 Pro. Come JDK abbiamo usato la versione 14.0.0.

2. Presentazione algoritmi

2.1 Metodi in comune:

I seguenti tre metodi sono utilizzati in tutti e tre gli algoritmi con delle piccole modifiche in base al tipo di albero utilizzato.

1. Nodo
2. Ricerca
3. Pre Order
4. Generatore interattivo

```
public Nodo ricerca(int valore, Nodo radice)
{
    if (radice == null || valore == radice.getKey())
    {
        return radice;
    }
    else if (valore < radice.getKey())
    {
        return ricerca(valore, radice.getNs());
    }
    else
    {
        return ricerca(valore, radice.getNd());
    }
}
```

La ricerca è da manuale.

```java
public static class Nodo
{
    public int key;
    public String num ;
    private Nodo gen;
    private Nodo ns;
    private Nodo nd;

    public Nodo(int key, String num)
    {
        this.key = key;
        this.num = num;
        gen = null;
        ns = null;
        nd = null;
    }
```

La classe "Nodo" utilizzata in tutti e 3 gli algoritmi, serve a inizializzare tutte le seguenti variabili:

key = chiave
num = nome del nodo
altezza = altezza dell'albero impostata di default come 1
ns = Nodo Sinistro
nd = Nodo Destro

```java
public void preorder(Nodo n)
{
    if (n == null)
    {
        System.out.print("NULL ");
    }
    else
    {
        System.out.print(n.getKey() + ":" + n.getNum() + " ");
        preorder(n.getNs());
        preorder(n.getNd());
    }
}
```

PreOrder è da manuale

Generatore interattivo:
Metodo usato per rendere gli algoritmi interattivi e fornire l'output. Utilizza la classe Scanner con il "hasNextLine" per leggere i comandi in input. Il while termina quando si inserisce la parola "exit". L'array "b" rappresenta una linea di input dove in prima posizione c'è il comando da eseguire, in seconda il valore, mentre la terza posizione rappresenta il nome. Gli if successivi determinano i comandi da eseguire in base all'istruzione desiderata inserita in input (che possono essere "insert", "show", "find" e "exit").

```
public static void generatore(AlberoBinario albero)
{
    Scanner Int = new Scanner(System.in);

    {
        while (Int.hasNextLine())
        {
            String a = Int.nextLine();
            String[] b = a.split( regex: " ",  limit: 3);

            comando = b[0];

            if (b.length >= 2)
            {
                valore = Integer.valueOf(b[1]);
            }

            if (b.length >= 3)
            {
                nome = b[2];
            }

            if (comando.equals("insert"))
            {
                albero.agValore(albero, valore, nome);
            }
            else if (comando.equals("show"))
            {
                albero.preorder(albero);
                System.out.print("\n");
            }
            else if (comando.equals("find"))
            {
                System.out.println(albero.ricerca(albero, valore).getNum());
            }
            else if (comando.equals("exit"))
            {
                break;
            }
        }
    }
}
```

2.2 Alberi binari di ricerca semplici

Dato che la classe "nodo" e le funzioni "ricerca" e "preorder", sono le stesse per tutti, approfondiamo solo le funzioni "inserimento" e "agValore", contenute nella classe "AlberoBinario".

La funzione "agValore" serve per aggiungere all'albero un nodo, tramite la funzione "inserimento". Il nodo da inserire è "daIns".

```
public static class AlberoBinario
{
    Nodo radice;

    public AlberoBinario()
    {
        radice = null;
    }

    public void agValore(AlberoBinario albero, int valore, String nome)
    {
        Nodo daIns = new Nodo(valore, nome);
        inserimento(radice, daIns);
    }
```

Nella funzione "inserimento", il primo if verifica se l'albero è vuoto. In questo caso, il nodo da inserire ("daIns"), viene inserito come radice. La condizione successiva, controlla se il valore del nodo da inserire è minore o uguale dell'ultimo nodo preso in considerazione ("attuale", nel primo caso è la radice, se esiste). In caso affermativo, si agisce sulla parte sinistra dell'albero, in caso contrario, sulla parte destra. Le istruzioni per l'aggiunta dei nodi figli sono le stesse per entrambe le parti dell'albero.

Si verifica se è presente il figlio (sinistro o destro) e in questo caso si richiama ricorsivamente la funzione "inserimento" utilizzando quest'ultimo come nodo "attuale". Se il figlio (sinistro o destro) non è presente, si setta "daIns" come nodo "attuale" e di conseguenza quest'ultimo diventa genitore di "daIns".

```java
public void inserimento(Nodo attuale, Nodo daIns)
{
    if (attuale == null)
    {
        radice = daIns;
        return;
    }

    if (daIns.getKey() <= attuale.getKey())
    {
        if (attuale.getNs() == null)
        {
            attuale.setNs(daIns);
            daIns.setGen(attuale);
        }
        else
        {
            inserimento(attuale.getNs(), daIns);
        }
    }
    else
    {
        if (attuale.getNd() == null)
        {
            attuale.setNd(daIns);
            daIns.setGen(attuale);
        }
        else
        {
            inserimento(attuale.getNd(), daIns);
        }
    }
}
```

2.3 Alberi binari di ricerca di tipo AVL

```
public static class AVL
{
    Nodo radice;

    public int altezza(Nodo N)
    {
        if (N == null)
            return 0;

        return N.altezza;
    }

    public int max(int a, int b)
    {
        return (a > b) ? a : b;
    }

    public Nodo rotazioneDestra(Nodo y)
    {
        Nodo x = y.ns;
        Nodo T2 = x.nd;

        x.nd = y;
        y.ns = T2;

        y.altezza = max(altezza(y.ns), altezza(y.nd)) + 1;
        x.altezza = max(altezza(x.ns), altezza(x.nd)) + 1;

        return x;
    }

    public Nodo rotazioneSinistra(Nodo x)
    {
        Nodo y = x.nd;
        Nodo T2 = y.ns;

        y.ns = x;
        x.nd = T2;

        x.altezza = max(altezza(x.ns), altezza(x.nd)) + 1;
        y.altezza = max(altezza(y.ns), altezza(y.nd)) + 1;

        return y;
    }
```

Nella classe AVL, “altezza” ritorna l’altezza dell’albero, invece “max” è usato per trovare l’elemento più grande nelle due rotazioni.
“rotazioneSinistra” e “rotazioneDestra” eseguono semplicemente le rotazioni per ribilanciare l’albero e ricalcolano l’altezza del nodo.

```java
public Nodo inserimento(Nodo nodo, int key, String nome)
{
    if (nodo == null)
    {
        return (new Nodo(key, nome));
    }

    if (key < nodo.key)
    {
        nodo.ns = inserimento(nodo.ns, key, nome);
    }
    else if (key > nodo.key)
    {
        nodo.nd = inserimento(nodo.nd, key, nome);
    }
    else
    {
        return nodo;
    }
    nodo.altezza = 1 + max(altezza(nodo.ns), altezza(nodo.nd));
    int bilancio = bilancio(nodo);

    if (bilancio > 1 && key < nodo.ns.key)
    {
        return rotazioneDestra(nodo);
    }

    if (bilancio < -1 && key > nodo.nd.key)
    {
        return rotazioneSinistra(nodo);
    }

    if (bilancio > 1 && key > nodo.ns.key)
    {
        nodo.ns = rotazioneSinistra(nodo.ns);
        return rotazioneDestra(nodo);
    }

    if (bilancio < -1 && key < nodo.nd.key)
    {
        nodo.nd = rotazioneDestra(nodo.nd);
        return rotazioneSinistra(nodo);
    }
    return nodo;
}
```

Nella funzione “inserimento”, il primo if, quando trova un nodo “null”, inserisce il nuovo nodo in quella posizione. Durante la prima iterazione, viene creata la radice. I successivi due if, confrontano la key del nodo che si sta per inserire e quella del nodo di riferimento. Quindi settano ricorsivamente il figlio sinistro, se la key del nodo di riferimento è maggiore, il figlio destro altrimenti. Se le key sono uguali, ritorna il nodo.
A questo punto viene calcolata l’altezza del nodo, sommando 1 all’altezza massima tra i due figli. Successivamente si usa “bilancio”.
Gli if successivi stabiliscono, se ci sia bisogno e quale tipo di rotazione effettuare (se rotazione sinistra o rotazione destra).

```
public int bilancio(Nodo N)
{
    if (N == null)
    {
        return 0;
    }

    return altezza(N.ns) - altezza(N.nd);
}
```

Il metodo “bilancio”, calcola la differenza di altezza fra il figlio sinistro e il figlio destro.

2.4 Alberi binari di ricerca di tipo Red-Black

Il nodo “nil” è un nodo con key -1. Nella funzione inserimento, al primo inserimento la radice corrisponde al valore nil. Se la radice ha valore nil, il nodo da inserire (ovvero nodo), diventa radice, il colore del nodo viene settato a black e il parente del nodo diventa nil. Altrimenti, il colore del nodo viene settato a red e il suo valore viene comparato con gli altri nodi e viene inserito in una posizione vuota a sinistra o a destra in base al suo valore, come succede negli altri tipi di albero binario. Infine viene chiamata la funzione “fixTree”, che sistema l’albero.

```java
public static final Nodo nil = new Nodo( key: -1,  nome: "");

public static class RBT
{
    Nodo radice = nil;
    public void inserimento(Nodo nodo)
    {
        Nodo temp = radice;
        if (radice == nil)
        {
            radice = nodo;
            nodo.colore = BLACK;
            nodo.parent = nil;
        } else
        {
            nodo.colore = RED;
            while (true)
            {
                if (nodo.key < temp.key)
                {
                    if (temp.sinistro == nil)
                    {
                        temp.sinistro = nodo;
                        nodo.parent = temp;
                        break;
                    } else
                    {
                        temp = temp.sinistro;
                    }
                } else if (nodo.key >= temp.key)
                {
                    if (temp.destro == nil)
                    {
                        temp.destro = nodo;
                        nodo.parent = temp;
                        break;
                    } else
                    {
                        temp = temp.destro;
                    }
                }
            }
            fixTree(nodo);
        }
    }
```

La funzione “fixTree” fa rispettare le 5 proprietà che valgono per i Red Black tree. Finchè il colore del parent del nodo (padre) è rosso, la variabile “zio” viene inizializzata a nil. Se il valore del nodo padre è uguale al nodo zio sinistro (figlio sinistro del nodo nonno), ridefiniamo la variabile “zio” come zio destro (figlio destro del nodo nonno). In seguito controlliamo se lo zio non è di colore rosso e non ha valore nil e in questo caso, vengono settati i colori: “parent” e “zio” a nero e il nodo genitore del genitore (nonno) a rosso. La variabile nodo, viene ridefinita come genitore del genitore (nonno). Se la variabile “nodo” equivale a parent.destro (al fratello destro), il nodo genitore diventa nodo e si esegue la funzione “rotazioneSinistra”. Infine, il colore del genitore viene settato a nero, mentre quello del genitore del genitore (nonno) a rosso e si esegue “rotazioneDestra” del nonno.
Se il primo if è falso, viene eseguita la stessa procedura in modo speculare (quindi con lo zio sinistro al posto dello zio destro ecc.).
Alla fine della funzione, la radice viene colorata di nero.

```
public void fixTree(Nodo nodo)
{
    while (nodo.parent.colore == RED)
    {
        Nodo zio = nil;
        if (nodo.parent == nodo.parent.parent.sinistro)
        {
            zio = nodo.parent.parent.destro;

            if (zio != nil && zio.colore == RED)
            {
                nodo.parent.colore = BLACK;
                zio.colore = BLACK;
                nodo.parent.parent.colore = RED;
                nodo = nodo.parent.parent;
                continue;
            }

            if (nodo == nodo.parent.destro)
            {
                nodo = nodo.parent;
                rotazioneSinistra(nodo);
            }

            nodo.parent.colore = BLACK;
            nodo.parent.parent.colore = RED;
            rotazioneDestra(nodo.parent.parent);
```

```
        } else
        {
            zio = nodo.parent.parent.sinistro;

            if (zio != nil && zio.colore == RED)
            {
                nodo.parent.colore = BLACK;
                zio.colore = BLACK;
                nodo.parent.parent.colore = RED;
                nodo = nodo.parent.parent;
                continue;
            }

            if (nodo == nodo.parent.sinistro)
            {
                nodo = nodo.parent;
                rotazioneDestra(nodo);
            }

            nodo.parent.colore = BLACK;
            nodo.parent.parent.colore = RED;
            rotazioneSinistra(nodo.parent.parent);
        }
    }

    radice.colore = BLACK;
}
```

```java
public void rotazioneSinistra(Nodo nodo)
{
    if (nodo.parent != nil)
    {
        if (nodo == nodo.parent.sinistro)
        {
            nodo.parent.sinistro = nodo.destro;
        } else
        {
            nodo.parent.destro = nodo.destro;
        }

        nodo.destro.parent = nodo.parent;
        nodo.parent = nodo.destro;

        if (nodo.destro.sinistro != nil)
        {
            nodo.destro.sinistro.parent = nodo;
        }

        nodo.destro = nodo.destro.sinistro;
        nodo.parent.sinistro = nodo;
    } else
    {
        Nodo destro = radice.destro;
        radice.destro = destro.sinistro;
        destro.sinistro.parent = radice;
        radice.parent = destro;
        destro.sinistro = radice;
        destro.parent = nil;
        radice = destro;
    }
}
```

```java
public void rotazioneDestra(Nodo nodo)
{
    if (nodo.parent != nil)
    {
        if (nodo == nodo.parent.sinistro)
        {
            nodo.parent.sinistro = nodo.sinistro;
        } else
        {
            nodo.parent.destro = nodo.sinistro;
        }

        nodo.sinistro.parent = nodo.parent;
        nodo.parent = nodo.sinistro;

        if (nodo.sinistro.destro != nil)
        {
            nodo.sinistro.destro.parent = nodo;
        }
        nodo.sinistro = nodo.sinistro.destro;
        nodo.parent.destro = nodo;
    } else
    {
        Nodo sinistro = radice.sinistro;
        radice.sinistro = radice.sinistro.destro;
        sinistro.destro.parent = radice;
        radice.parent = sinistro;
        sinistro.destro = radice;
        sinistro.parent = nil;
        radice = sinistro;
    }
}
```

La funzione “rotazioneSinistra” è da manuale e la funzione “rotazioneDestra” è speculare alla precedente. Perciò, le spiegazioni sono state omesse.

```java
public static void preorder(Nodo nodo)
{
    if (nodo == nil)
    {
        System.out.print("NULL ");
    } else if (nodo != null && nodo.colore == BLACK)
    {
        System.out.print(nodo.key + ":" + nodo.nome + ":" + "black ");
        preorder(nodo.sinistro);
        preorder(nodo.destro);
    } else if (nodo != null && nodo.colore == RED)
    {
        System.out.print(nodo.key + ":" + nodo.nome + ":" + "red ");
        preorder(nodo.sinistro);
        preorder(nodo.destro);
    }
}
```

La funzione “preorder” è da manuale, con i dovuti adattamenti affinché funzioni con i colori.

3. La funzione "*Creatore*"

Per utilizzare il "*Random*" predisposto dalla libreria di Java utilizziamo la stessa funzione che abbiamo usato nella prima Relazione. Impostiamo i limiti inferiore e superiore che rappresentano il minimo e massimo numero che il *"Random"* può selezionare. Per fare in modo che possano essere presenti anche numeri negativi, utilizziamo la variabile *"a"* in modo che l'intervallo di scelta sia *"y"*. Nella variabile *"y"*, il "+1" viene aggiunto, perché la funzione "nextInt()" non considera l'ultimo elemento del range. Sottraendo il limite inferiore, sarà possibile avere anche numeri negativi nell'array.

```java
public static ArrayList creatore(int n)
{
    Random random = new Random();
    int limiteI = -100000000;
    int limiteS = 100000000;
    int cicli = n;
    int y = limiteS - limiteI + 1;

    ArrayList<Integer> array = new ArrayList<>();

    for (int i = 0; i < cicli; i++)
    {
        int a = random.nextInt(y) + limiteI;
        array.add(a);
    }
    return array;
}
```

Ad esempio:

Limite Inferiore = -100;

Limite superiore = 100;

Range Y = Limite Superiore - Limite Inferiore + 1 = 100 + (-100) + 1 = 201;

Random Y = 10;

Numero finale = Random Y + Limite inferiore = 10 + (-100) = -90;

Nell'implementazione di tutti gli algoritmi, abbiamo utilizzato un range che va da

-100.000.000 a 100.000.000.

4 La funzione *"Generatore"*

```java
public static void generatore(ArrayList lista)
{

    for (int i = 0; i < lista.size(); i++)
    {
        Nodo cercato = ricerca((int) lista.get(i), radice);

        if (cercato == null)
        {
            Nodo nodo = new Nodo((int) lista.get(i),  nome: "");
            inserimento(nodo);
        }
    }
}
```

La funzione "generatore" cerca i valori della lista che è data in input. Se non vengono trovati, li inserisce come nuovi nodi nell'albero con la funzione "inserimento".

```java
public static Nodo ricerca(int valore, Nodo radice)
{
    if (radice == null || valore == radice.key)
    {
        return radice;
    }
    else if (valore < radice.key)
    {
        return ricerca(valore, radice.sinistro);
    }
    else
    {
        return ricerca(valore, radice.destro);
    }
}
```

La funzione "ricerca" è da manuale, perciò non ci soffermiamo su questa spiegazione.

5 Il metodo per misurare il tempo

Per granularità (o risoluzione) si intende l'intervallo di misurazione necessario al sistema per caricare una singola istruzione, in questo caso noi carichiamo e tentiamo di ottenere il tempo minimo registrabile e ritorniamo la differenza tra i due risultati.

```java
public static long Granularità() {

    long t0 = System.nanoTime();
    long t1 = System.nanoTime();

    while (t1 == t0){

        t1 = System.nanoTime();

    }
    return (t1 - t0);
}
```

In "*ripCalculateTare*" calcoliamo il numero di ripetizioni minimo per far si che l'esecuzione di *"prepare"* sia registrato come tempo di preparazione e venga registrato come dato per conoscere il tempo per il caricamento dati tenendo conto anche dei cicli errati. Il tempo di *"prepare"* viene registrato per poterlo sottrarre al tempo totale ed avere il tempo di esecuzione vero e proprio dell'algoritmo.

```java
public static double ripCalculateTare(int d, long tMin){

    long t0 = 0;
    long t1 = 0;
    long rip = 1;

    while (t1 - t0 <= tMin){
        rip = rip * 2;
        t0 = System.currentTimeMillis();
        for (int i = 1; i <= rip; i++){
            prepare(d);
        }
        t1 = System.currentTimeMillis();
    }

    long max=rip;
    long min=rip/2;
    int failedCycles=5;
    while((max-min)>=failedCycles) {
        rip= (max+min)/2; //medium value
        t0=System.currentTimeMillis() ;
        for(int i=0; i<=rip; i++) {
            prepare(d);
        }
        t1=System.currentTimeMillis();
        if(t1-t0<= tMin) {
            min=rip;
        }else {
            max=rip;
        }
    }
    return max;
}
```

Nel primo while verifichiamo ogni quante ripetizioni la differenza dei due tempi (t0 e t1) continua ad essere minore del tMin (ovvero la granularità fratto la percentuale di errore dei dati, nel nostro caso 1%). La crescita di *"rip"* è esponenziale. Nel secondo while invece, considerando le ripetizioni, troviamo il tempo minore senza cicli falliti tramite il metodo numerico di bisezione.

```
private static long ripCalculateGross(int d, long tMin){
    long t0=0;
    long t1=0;
    long rip=1;
    while(t1-t0 <= tMin) {
        rip = rip*2;
        t0 = System.currentTimeMillis();
        for(int i = 0; i <= rip; i++) {
            execute(prepare(d), d);
        }
        t1= System.currentTimeMillis();
    }
    long max = rip;
    long min = rip / 2 ;
    int cicliErrati = 5 ;

    while (max-min >= cicliErrati) {
        rip = (max+min) / 2 ;
        t0 = System.currentTimeMillis();
        for(int i = 1; i<=rip; i++) {
            execute (prepare(d), d);
        }
        t1= System.currentTimeMillis();
        if(t1-t0 <= tMin) {
            min = rip;
        } else {
            max = rip;
        }
    }
    return max;
}
```

"ripCalculateGross" calcola il tempo di esecuzione del programma in totale, compreso del tempo di esecuzione dell'algoritmo, il tempo per il caricamento dei dati e come per *"ripCalculateTare"*, vengono tenuti in considerazione anche i cicli errati che possono essere presenti. In questo caso abbiamo inserito un numero di al più 5 cicli errati.

La stessa metodologia viene usata sia in *"ripCalculateTare"* sia in *"ripCalculateGross"*. Nella prima viene considerato il tempo di esecuzione della preparazione dei dati che andranno inseriti nell'algoritmo, mentre in *"ripCalculateGross"* viene considerato tutto il processo, ovvero dalla creazione dei dati allo svolgimento dell'algoritmo.

La funzione *"prepare"* inizializza l'algoritmo che successivamente andrà' ad inserire i dati nell'array per lo svolgimento dell'algoritmo. *"execute*" invece esegue effettivamente l'algoritmo, senza contare come siano stati prodotti i dati in input.

```
private static ArrayList<Integer> prepare (int d){

    return creatore(d);
}

private static long execute(ArrayList<Integer> array, int d){

    return QuickSelect(array,  inizio: 0,  fine: array.size() - 1,  k: d / 2);
}
```

In questa parte implementiamo il calcolo del tempo caricando il valore minimo (esecuzione della parte di caricamento dati) e il valore massimo, ovvero sfruttiamo il tempo impiegato dall'algoritmo per calcolare i risultati finali, prendendo in input i dati creati randomicamente.

In "mediumNetTime", usiamo i vari metodi già realizzati (*"RipCalculateGross"*, *"ripCalculateTare"*, *"prepare"* ed *"execute").* Con il primo for controlliamo nuovamente il tempo di preparazione dei dati mentre nel secondo for, calcoliamo i tempi dell'esecuzione dell'algoritmo. Tutto ciò serve per calcolare il tempo medio sottraendo i tempi messi a rapporto con i valori precedentemente calcolati.

```
private static double mediumNetTime(int d, long tMin) {

    double ripTara = ripCalculateTare(d, tMin);
    double ripLordo = ripCalculateGross(d, tMin);
    long t0 = System.currentTimeMillis();
    for(int i = 1; i<=ripTara; i++){
        prepare(d);
    }
    long t1 = System.currentTimeMillis();
    long tTara = t1-t0;
    t0 = System.currentTimeMillis();
    for(int i = 1; i<=ripLordo; i++){
        execute(prepare(d), d);
    }
    t1 = System.currentTimeMillis();
    double tLordo = t1 - t0;
    double tMedio = (tLordo / ripLordo) - (tTara / ripTara);
    return tMedio;
}
```

"Misurate" è la parte più importante dell'intero nucleo dell'algoritmo per la misurazione dei tempi, poiché vengono caricate tutte le variabili necessarie per

inserire i valori delle ripetizioni, del *"delta"* e dell'errore della ripartizione della gaussiana (za). Tutto ciò serve proprio per collocare in un unico punto tutti gli input necessari alla ricerca dei tempi.

```
private static double[] misurate(int d, int c, double za,
                                 long tMin, double DELTA) {
    double t=0;
    double sum2 = 0;
    double cn = 0;
    double e;
    double s;
    double delta;
    double m;
    do {
        for(int i=1; i<=c; i++) {
            m = mediumNetTime(d, tMin);
            t+=m;
            sum2 = sum2 +(m*m);
        }
        cn= cn+c;
        e=t/cn;
        s= Math.sqrt((sum2/cn - (e*e)));
        delta= (1/Math.sqrt(cn)) *za *s;
    }
    while(delta>DELTA);
    double[] result = new double[2];
    result[0] = e;
    result[1] = delta;
    return result;
}
```

Con il do-while verifichiamo quando l'errore, ovvero la variabile *"DELTA"*, è maggiore del valore calcolato (variabile *"delta"*). "*delta*" è il parametro che serve a calcolare l'intervallo di confidenza. Se i valori superano l'1%, viene caricato il valore successivo. Nel for invece, viene tenuto il conto dei cicli di misurazioni da effettuare, ovvero tutti i risultati del tempo medio, tramite la funzione *"mediumNetTime"*.

In output sono riportate le variabili *"e"* ed *"s"* che sono rispettivamente la media dei tempi e la deviazione standard.

Per calcolare la varianza campionaria abbiamo seguito questa formula. Mettendola sotto radice, troviamo la deviazione standard:

Nella formula, la prima parte ($X(n,i)^2$) della sommatoria è rappresentata dalla variabile *"sum2",* la seconda parte (dopo il segno meno), è rappresentata dalla variabile *"e"* al quadrato.

$$s(n)^2 = \frac{1}{c_n} \sum_{i=1}^{c_n} (X(n,i) - E'[X(n)])^2$$

Nel main abbiamo inizializzato le variabili principali, tra cui *"c"* che rappresenta il numero di misurazioni dell'algoritmo. Aumentando il valore, il numero di ripetizioni aumenta e di conseguenza migliora la precisione e aumenta il tempo totale di calcolo dei tempi.

```
public static void main(String[] args) throws IOException {

    int c = 5;
    double za = 2.32;
    double percent = 0.01;
    long tMin = (long) (Granularita() / percent);
    double DELTA = 0.01;
    int contatore = 0;
    double[] mis;
    double[] t = new double[1000];

    for (int i = 100; i <= 6000000; i = i + ((i * 10) / 100)) {
        System.out.println(i);
        mis = misurate(i, c, za, tMin, DELTA);
        if (mis[0] < 100000) {
            t[contatore] = mis[0];
            contatore++;
        }
    }
```

Nel for memorizziamo tutti i tempi nell'array "t". Il "contatore * 2", serve per inserire i dati che verranno trascritti sul foglio Excel. I dati verranno caricati con un ordine tale che il primo dei due risultati sia sempre il tempo e il secondo la deviazione standard. Abbiamo quindi inserito questo contatore solo per ordinare gli elementi dell'array "t".

```
XSSFWorkbook workbook = new XSSFWorkbook();
OutputStream os = new FileOutputStream( name: "TempiABR.xlsx");
Sheet sheet = workbook.createSheet();
Row row = sheet.createRow( rownum: 1);
Cell cell = row.createCell( column: 1);
cell.setCellValue("n");
Cell cell1 = row.createCell( column: 2);
cell1.setCellValue("Tempo");
Cell cell0 = row.createCell( column: 3);
cell0.setCellValue("delta");
Cell cell01 = row.createCell( column: 4);
cell01.setCellValue("ds");
Cell cell02 = row.createCell( column: 5);
cell02.setCellValue("Tempo ammortizzato");
int cont = 0;
for (int nn = 100; nn <= 6000000; nn = nn + ((nn * 10) / 100)) {
    Row row1 = sheet.createRow( rownum: cont + 3);
    Cell cell2 = row1.createCell( column: 1);
    cell2.setCellValue(nn);
    Cell cell3 = row1.createCell( column: 2);
    cell3.setCellValue(sd[(cont * 3)]);
    Cell cell4 = row1.createCell( column: 3);
    cell4.setCellValue(sd[(cont * 3) + 1]);
    Cell cell5 = row1.createCell( column: 4);
    cell5.setCellValue(sd[cont * 3] + 2);
    Cell cell6 = row1.createCell( column: 5);
    cell6.setCellValue(sd[cont * 3] / nn);
    cont++;
}
workbook.write(os);
```

Calcoliamo il tempo ammortizzato direttamente in output, usando i tempi salvati, dividendo ogni tempo per il numero di operazioni.

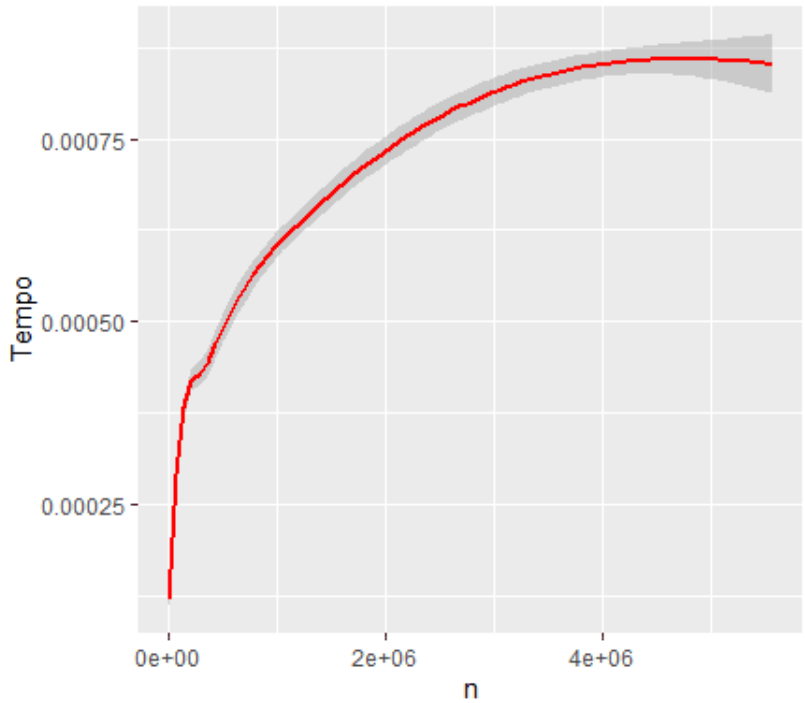
$$\text{tempo ammortizzato} = \frac{\text{tempo totale}}{n}.$$

Una volta che l'algoritmo ha finito, importiamo i dati in output direttamente su Excel. La libreria che abbiamo usato per farlo è "hu.blackbelt.bundles.poi:org.apache.poi:4.0.1_1".

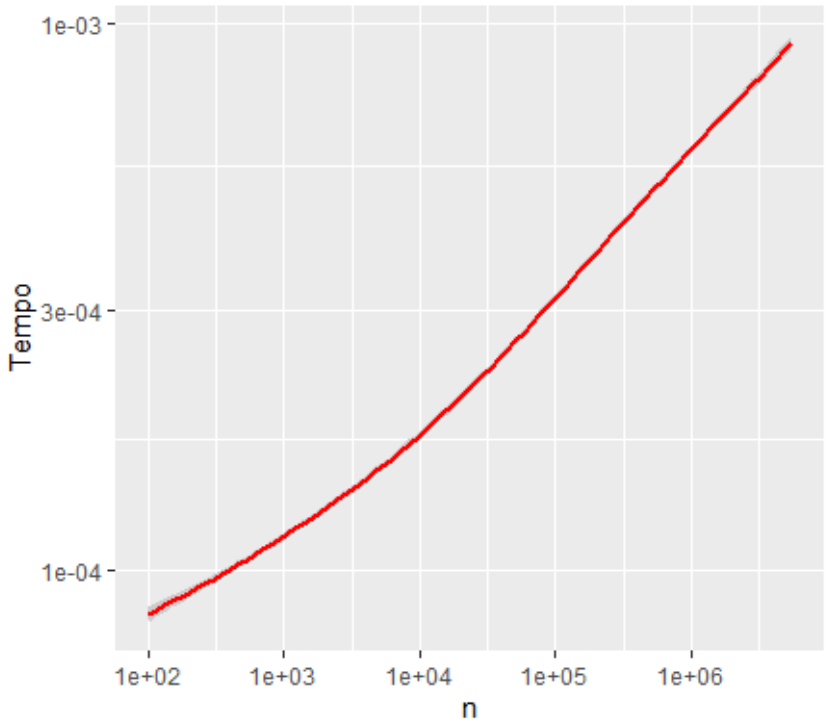
6 Presentazione dei risultati e considerazioni

Per fare i grafici abbiamo usato R. Nei grafici seguenti il Tempo è sempre rappresentato in millisecondi e collocato sull'asse y mentre *"n"* rappresenta il numero di operazioni di inserimento e di ricerca ed è collocato sull'asse x. I tempi sono stati

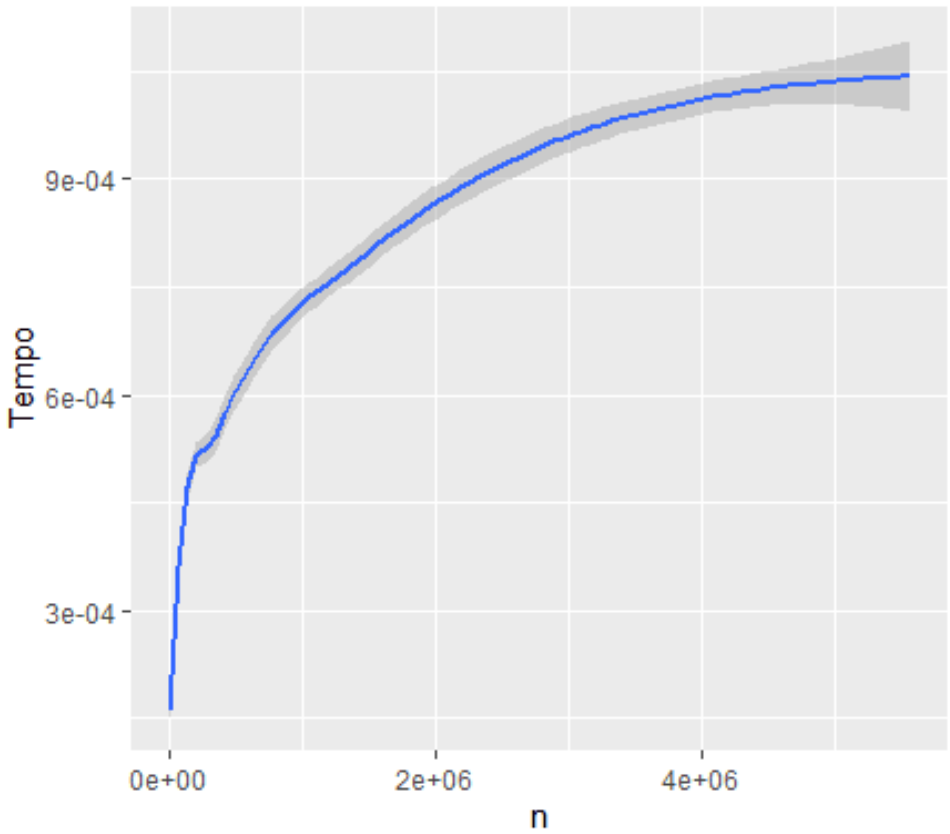
presi con un aumento del 10% della lunghezza dell'array (il valore successivo è maggiore del 10% rispetto al precedente). Il limite superiore della grandezza del vettore è stato fissato a 6 milioni di elementi.
Nei grafici la linea in azzurro rappresenta regressione lineare dei dati mentre la parte in grigio esprime l'errore standard.



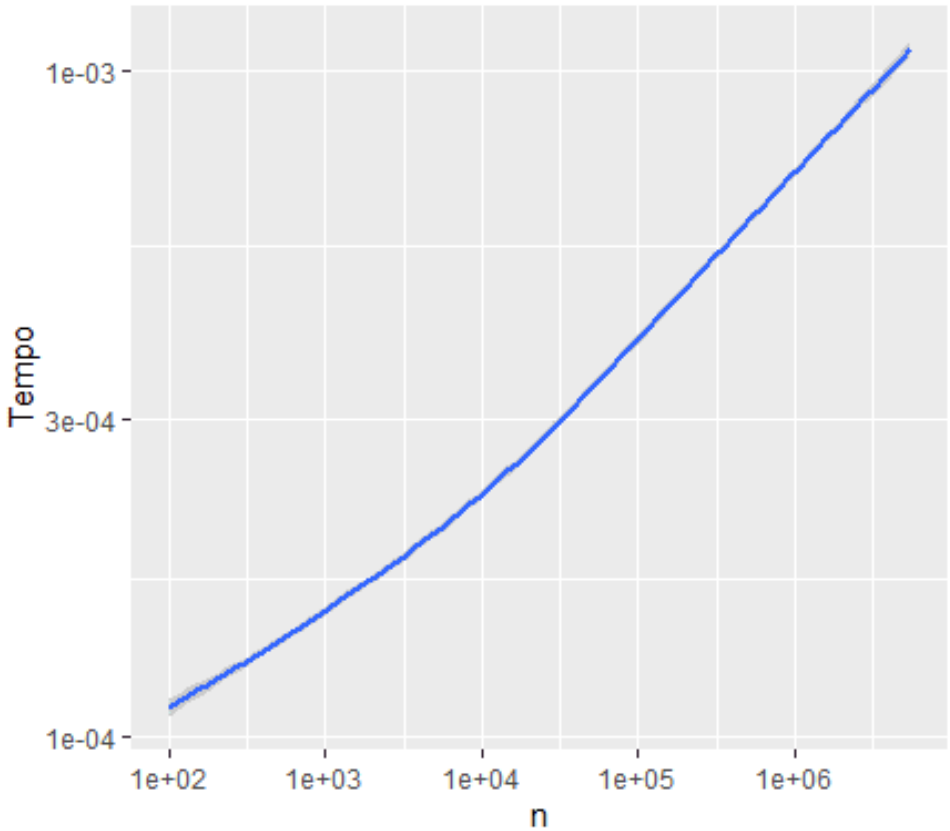
ABR con scala lineare (Albero binario di ricerca semplice)



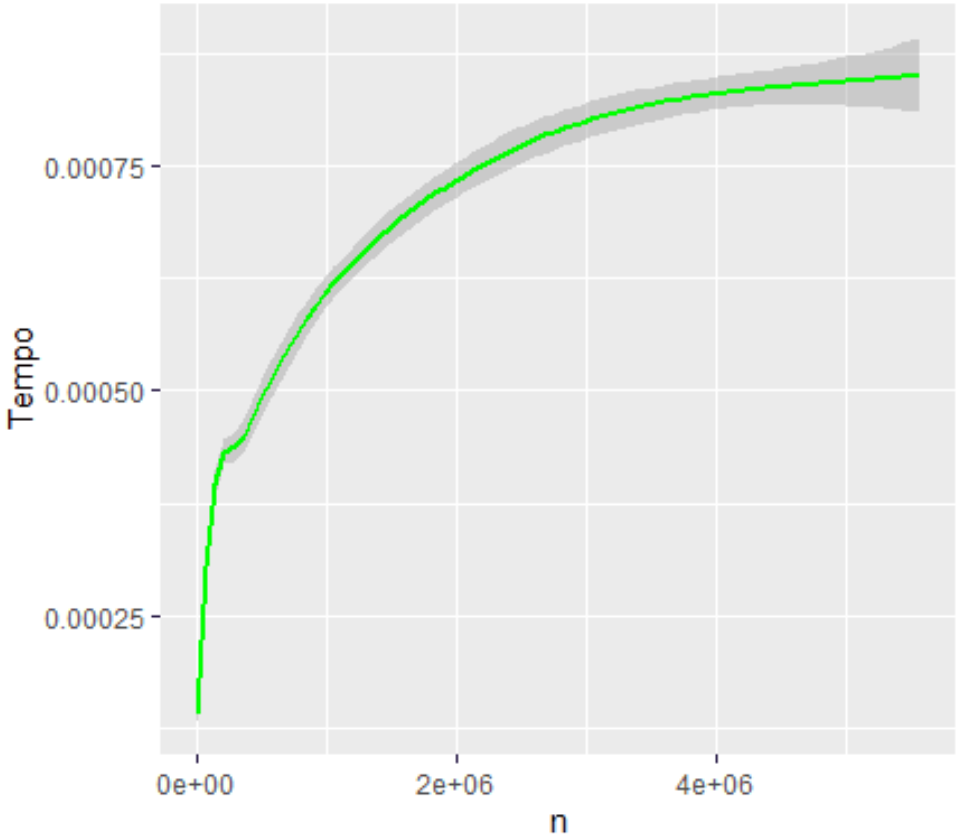
ABR con scala logaritmica (Albero binario di ricerca semplice)



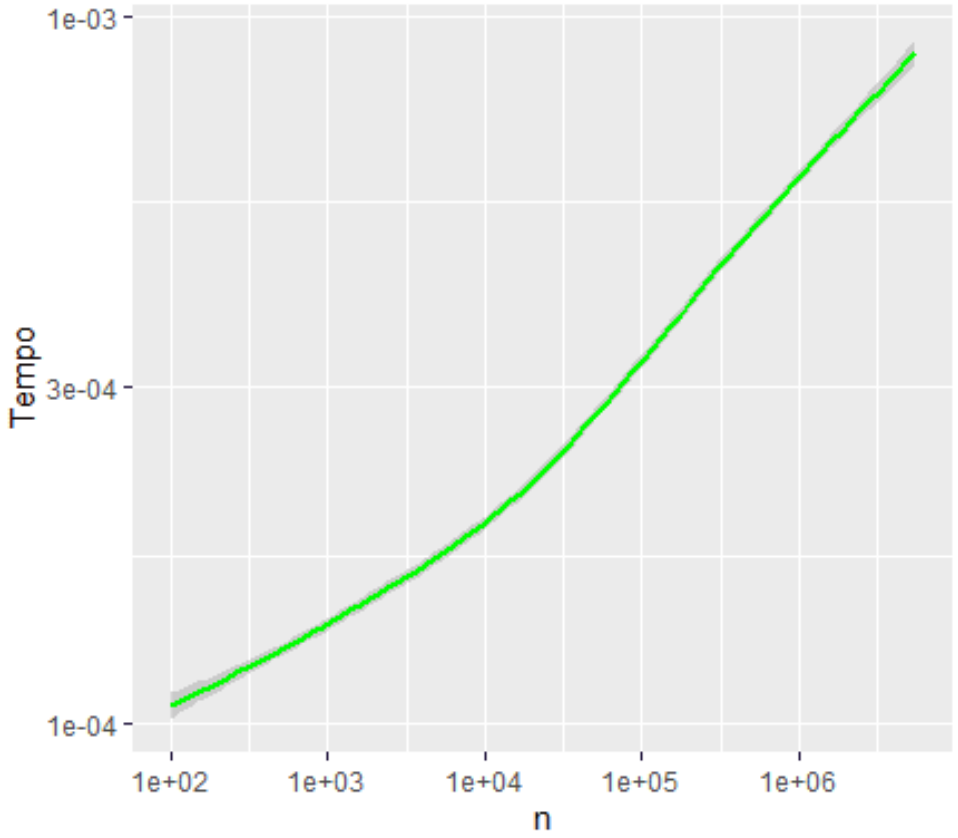
AVL con scala lineare



AVL con scala logaritmica



Red-Black con scala lineare

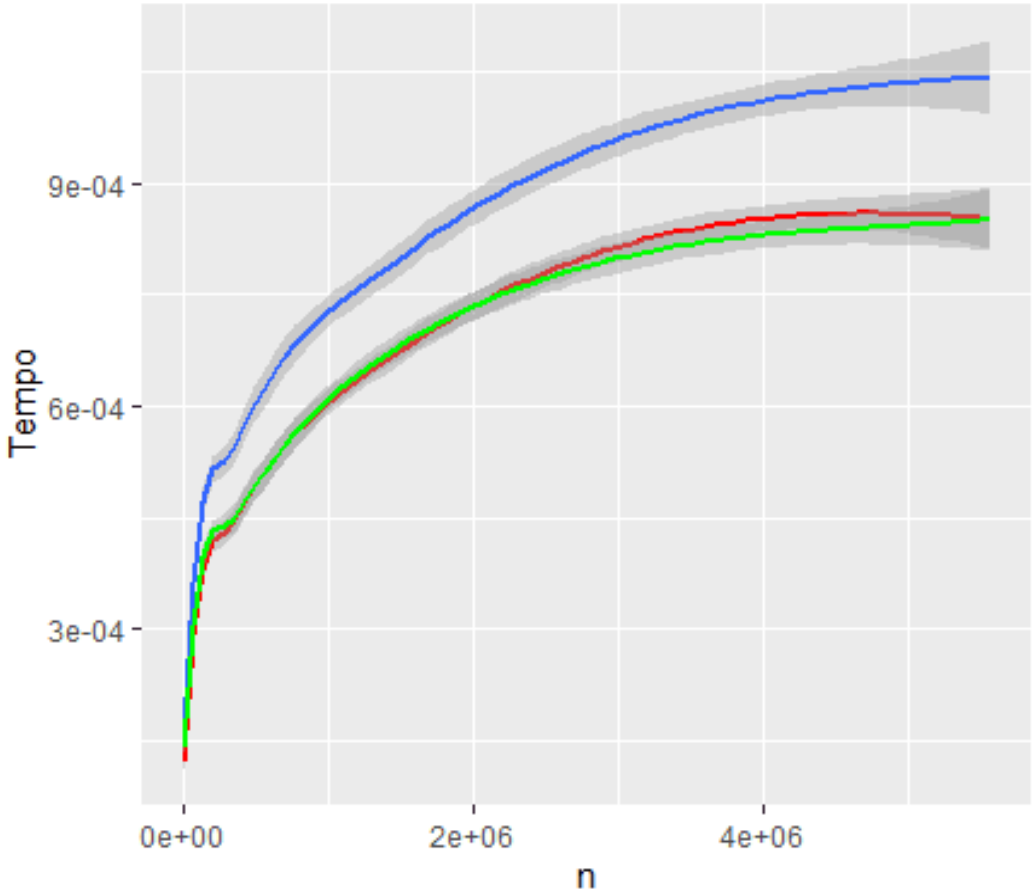


Red-Black con scala logaritmica

Sostanzialmente i grafici risultano coerenti con le nostre aspettative:

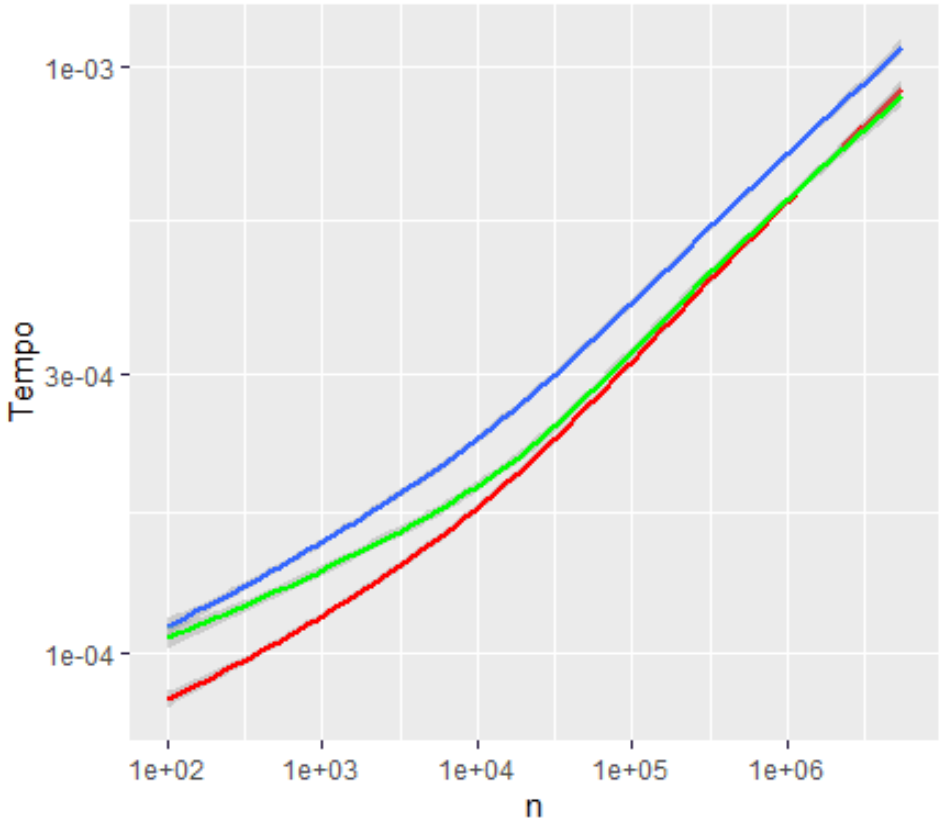
a)ABR ha una complessità temporale della ricerca e dell'inserimento di O(log n) nel caso medio e di O(n) nel caso peggiore (visto che i numeri sono generati in modo pseudo-randomico, il caso peggiore ha una bassa possibilità di presentarsi). Dal grafico si evince che abbiamo un andamento logaritmico, il che soddisfa le nostre aspettative sui tempi.
b)AVL ha una complessità temporale della ricerca e dell'inserimento di O(log n), sia nel caso medio sia nel caso peggiore. Dal grafico si può osservare che abbiamo un andamento logaritmico, il che rispetta le nostre aspettative.
c)RedBlack ha una complessità temporale della ricerca e dell'inserimento di O(log n), sia nel caso medio sia nel caso peggiore. Dal grafico si può osservare che abbiamo un andamento logaritmico, il che rispetta le nostre aspettative.

Tutti gli algoritmi con scala lineare



(in rosso: ABR, in blu: AVL, in verde: RB)

Tutti gli algoritmi con scala logaritmica



(in rosso: ABR, in blu: AVL, in verde: RB)

Come si può vedere l'algoritmo più veloce fino a un milione è ABR. Dopo il milione, le prestazioni di Red Black e ABR sono molto simili, ma l'algoritmo RB è leggermente più veloce. AVL è l'algoritmo più lento dei tre.