UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea in Ingegneria Informatica

Alberi RB vs Alberi BST

Professore:

Ch.mo Prof.

Simone Marinai

Candidato:

Hamidreza Hashemi

Mat. 5996260

Corso:

Algoritmi e Strutture Dati

ANNO ACCADEMICO 2020/2021

Dedicata a coloro che
ci provano ogni giorno.

INDICE

Introduzione **3**

1 Alberi Binari di Ricerca **4**

1.1 Costo e Complessità 5

1.2 Implementazione 6

2 Alberi Rosso Neri **8**

2.1 Costo e Complessità 9

2.2 Implementazione 10

3 Analisi di complessità **12**

3.1 Tempi medi con i dati fissi 14

3.2 Analisi di alberi binari di ricerca 16

3.3 Analisi di alberi Rosso neri 17

INTRODUZIONE

In questo eserizio facciamo un confronto tra due strutture dati che sono : **Alberi rosso neri** e **Alberi binari di ricerca**. Vediamo il codice scritto in python e facciamo un confronto sulle funzionalità di base (*inserimento, ricerca*). Per la semplicità non è stato implementato la funzione di cancellazione.

CAPITOLO 1

ALBERI BINARI DI RICERCA

Alberi binari di ricerca (*BST*) sono un tipo di strutture dati che consentono di Salvare, cancellare e cercare il dato con un tempo inferiore rispetto a tante altre strutture. Ogni albero è costruito da una radice e tanti nodi figli e ogni nodo ha un riferimento al nodo sinistro e destro. Il sotto albero sinistro di un nodo è costruito dai valori inferiori rispetto al nodo padre e viceversa il sotto albero destro è costruito dai valori superiori. Altezza di un albero è la massima distanza tra la radice e qualsiasi nodo figlio.

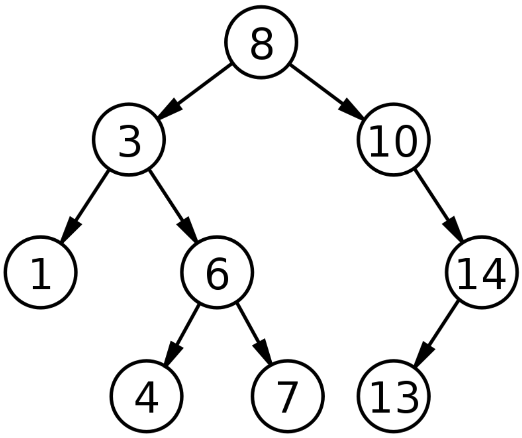
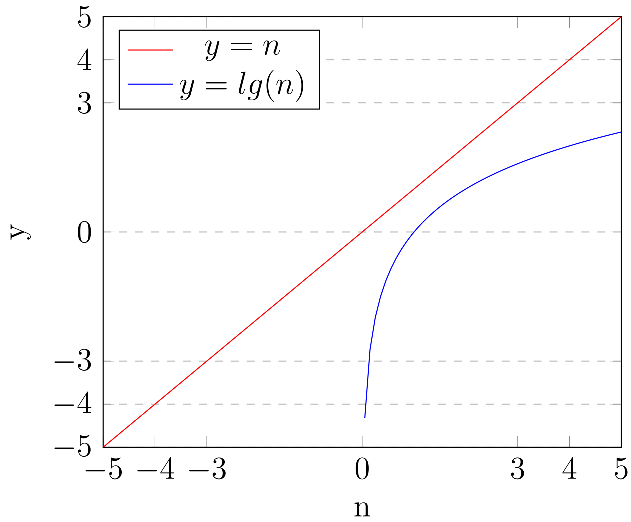


Figura 1.1: BST con l'altezza 3

1.1 Costo e Complessità

Il costo temporale di inserimento, cancellazione e la ricerca dipende dall'altezza dell'albero. le complessità di queste strutture dati sono $\mathcal{O}(h)$ per le operazioni di base. l'altezza dell'albero nel migliore caso può essere $\boldsymbol{n}$ e nel peggiore caso può essere $\mathbf{lg(n)}$. Questo dipende dall'ordine di inserimento.



Nella figura di sopra sono state importate le funzioni utili per il confronto asintotico. Il costo della nostra funzione potrebbe essere :

$$\mathcal{O}(lg(n)) \leq \mathcal{O}(f(n)) \leq \mathcal{O}(n)$$

1.2 Implementazione

Per implementare questa struttura di dati è stato utilizzato il linguaggio python. Ogni nodo è stato realizzato con la classe **Node**.

```
class Node:

    def __init__(self, data):

        self.left = None
        self.right = None
        self.data = data

    # Il metodo insert per inserire un dato.
    def insert(self, data):
        if self.data:
            if data < self.data:
                if self.left is None:
                    self.left = Node(data)
                else:
                    self.left.insert(data)
            elif data > self.data:
                if self.right is None:
                    self.right = Node(data)
                else:
                    self.right.insert(data)
        else:
            self.data = data

    # Il metodo search per trovare un dato.
    def search(self, lkpval):
        if lkpval < self.data:
```

Figura 1.2: Implementazione BST

I metodi implementati sono :

- `search` : Per trovare un dato
- `insert` : Per inserire un dato
- `printTree` : per stampare un dato
- `maxDepth` : Per trovare l'altezza

Per creare un albero binario di ricerca bisogna creare un nodo root e aggiungere con il metodo `insert` altri nodi al nodo root. Questo è stato realizzato per avere un codice più leggibile e corto.

CAPITOLO 2

ALBERI ROSSO NERI

Alberi rosso neri (RBT) sono un tipo esteso da alberi binari di ricerca che consentono di avere un albero bilanciato. Avere un albero bilanciato consente di ridurre i costi temporali. Ogni nodo sull'albero rosso nero ha tutte le proprietà di BST ma ha anche un colore che può variare tra rosso e nero. Ci sono alcune proprietà sui alberi rosso neri che devono essere sempre rispettate :

- Ogni nodo è rosso o nero
- La radice è nera
- Ogni foglia (T.nil) è nera
- Se un nodo è rosso, allora entrambi i suoi figli sono neri(No due rossi consecutivi in un cammino semplice da radice a foglia)
- Tutti i cammini da ogni nodo alle foglie contengono lo stesso numero di nodi neri

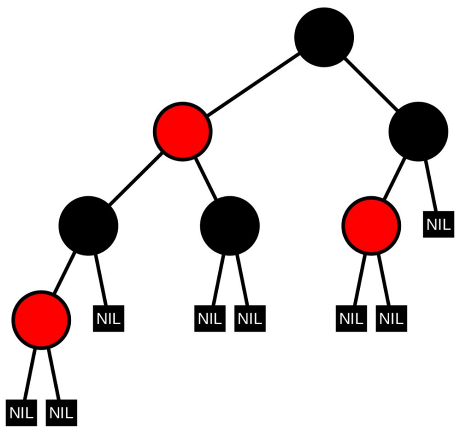
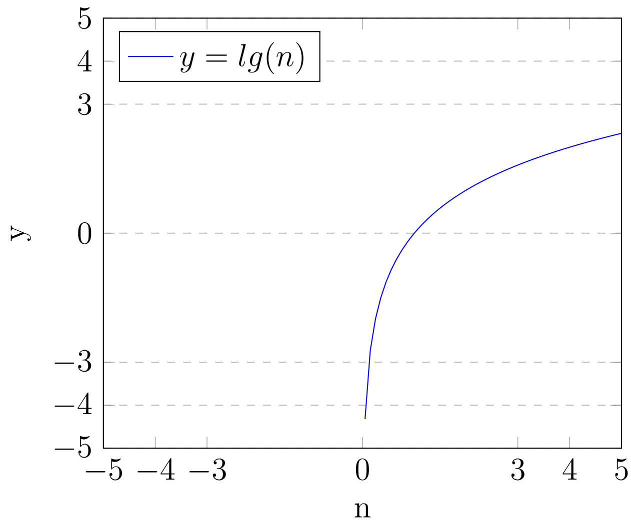


Figura 2.1: Esempio di RBT

2.1 Costo e Complessità

Tutte le operazioni di base[1] hanno un costo temporale di $\mathcal{O}(lg(n))$. Questo è per il fatto che l'altezza massima può essere $2lg(n+1)$.



Nella figura di sopra è stato importato la funzione $lg(n)$. Il costo della nostra funzione potrebbe essere :

$$\mathcal{O}(f(n)) \leq \mathcal{O}(lg(n))$$

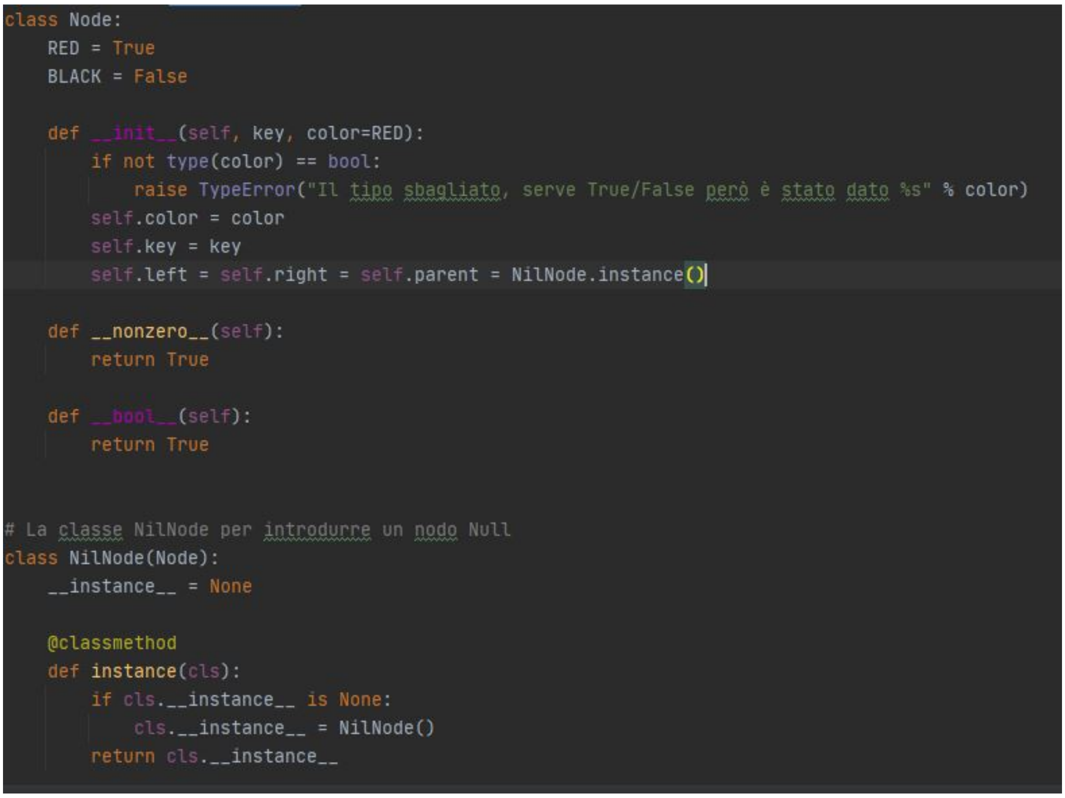
[1]L' operazione di cancellazione ha il costo di $\Theta(log(n))$

2.2 Implementazione

Per implementare questa struttura è stato creato il nodo che può essere rosso o nero. c'è anche il nodo NIL che è stato esteso da un nodo normale. Poi c'è l'albero che ha gli attributi root e size. Si potrebbe estendere BST per creare un RBT però è stato implementato separato in questo esercizio. I metodi importanti sono :

- Node
  - `init` : Initzializzare un nodo e settaggio del colore
- NilNode
  - `init` : Initzializzare un nodo e settaggio le proprietà a None
- RedBlackTree
  - `add` : Per inserire un dato
  - `insert` : Per inserire un nodo
  - `minimum` : Per trovare il minimo
  - `maximum` : Per trovare il massimo
  - `successor` : Per trovare il successore di un nodo
  - `predecessor` : Per trovare il predecessore di un nodo
  - `inorder_walk` : Il cammino inorder sull'albero
  - `search` : Per cercare un dato

Ci sono altri metodi privati non descritti nella lista che servono ad altri metodi della classe (*es. left_rotate, right_rotate, ...*)



```python
class Node:
    RED = True
    BLACK = False

    def __init__(self, key, color=RED):
        if not type(color) == bool:
            raise TypeError("Il tipo sbagliato, serve True/False però è stato dato %s" % color)
        self.color = color
        self.key = key
        self.left = self.right = self.parent = NilNode.instance()

    def __nonzero__(self):
        return True

    def __bool__(self):
        return True


# La classe NilNode per introdurre un nodo Null
class NilNode(Node):
    __instance__ = None

    @classmethod
    def instance(cls):
        if cls.__instance__ is None:
            cls.__instance__ = NilNode()
        return cls.__instance__
```

Figura 2.2: Esempio del codice

Nella figura di sopra è possibile vedere un pezzo del codice dell'implementazione di RBT scritto in python

CAPITOLO 3

ANALISI DI COMPLESSITÀ

In questo capitolo facciamo un'analisi sul costo e complessità di queste strutture dati. Usiamo un programma per fare i test necessari sulle nostre strutture dati.

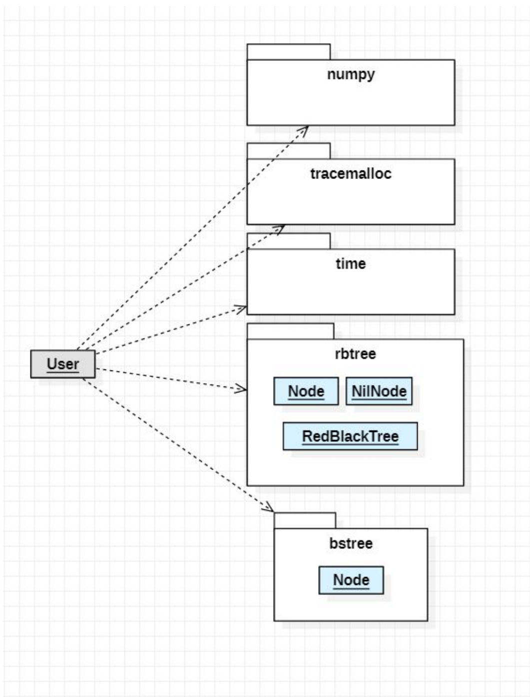


Figura 3.1: Lo schema di UML per testing

Come abbiamo visto nella figura di sopra il programma main usa i moduli descritti per fare un confronto temporale e spaziale tra le due strutture dati.

```
if __name__ == '__main__':
    randArray = np.random.randint(1, 200001, 20000)
    root = bstree.Node(10)
    beforeInsert = time.time()
    tracemalloc.start()
    for number in randArray:
        root.insert(number)
    print(f" BST RAM usato in Bytes (current, peak){tracemalloc.get_traced_memory()}")
    tracemalloc.stop()
    afterInsert = time.time()
    beforeSearch = time.time()
    root.search(2)
    time.sleep(0.01)
    afterSearch = time.time()
    spentInsert = afterInsert - beforeInsert
    spentSearch = afterSearch - beforeSearch
    print("---------------------------------------")
    print(f"tempo trascorso per inserire in BST [{spentInsert}] seconds ")
    print(f"tempo trascorso per cercare in BST [{spentSearch}] seconds ")
    print("---------------------------------------")
    rb = rbtree.RedBlackTree()
    beforeInsert = time.time()
    tracemalloc.start()
    for number in randArray:
        rb.add(number)
    print(f" RB RAM usato in Bytes (current, peak){tracemalloc.get_traced_memory()}")
    tracemalloc.stop()
    afterInsert = time.time()
    beforeSearch = time.time()
    rb.search(2)
    time.sleep(0.01)
    afterSearch = time.time()
    spentInsert = afterInsert - beforeInsert
    spentSearch = afterSearch - beforeSearch
    print("---------------------------------------")
    print(f"tempo trascorso per inserire in RBT [{spentInsert}] seonds ")
    print(f"tempo trascorso per cercare in RBT[{spentSearch}] seonds ")
    print("---------------------------------------")
    print(f"Altezza BST : {bstree.maxDepth(root)}")
```

Figura 3.2: Codice Main

Nella figura di sopra si vede il codice main. Questo programma crea un array di dimensione specificata con tutti i numeri diversi tra loro. Poi va a mischiare tutti i numeri e prende un numero a caso e lo mette come il root dell'albero. Poi va a inserire i dati in un albero binario di ricerca e vede il tempo trascorso e lo spazio occupato. Poi fa la stessa procedura con un albero rosso e nero.

3.1 Tempi medi con i dati fissi

È stato provato questo programma 10 volte su 500000 dati. In seguito vediamo i risultati ottenuti.

Risultato ottenuto con 500000 dati			
N. Esperimento	Altezza albero BST	T. Inserimento BST — RBT (sec.)	T. Ricerca BST — RBT (nanosec.)
1	47	23.557 — 14.395	38300 — 37400
2	46	11.671 — 13.304	52300 — 40600
3	45	10.845 — 10.408	46300 — 48400
4	50	11.216 — 12.969	54900 — 41900
5	44	10.503 — 12.471	39400 — 38700
6	46	11.442 — 13.872	41200 — 40000
7	49	11.361 — 13.849	38800 — 47200
8	52	11.580 — 13.058	45100 — 40100
9	46	11.120 — 12.177	39700 — 39900
10	47	10.878 — 12.755	50000 — 42800

Dalla tabella di sopra si vede che il tempo trascorso di inserimento nell'albero rosso e nero è di più rispetto all'albero binario di ricerca. Questo potrebbe stare per il fatto che la struttura dell'albero è più complicato. Però il tempo di ricerca è di meno rispetto all'albero binario di ricerca.

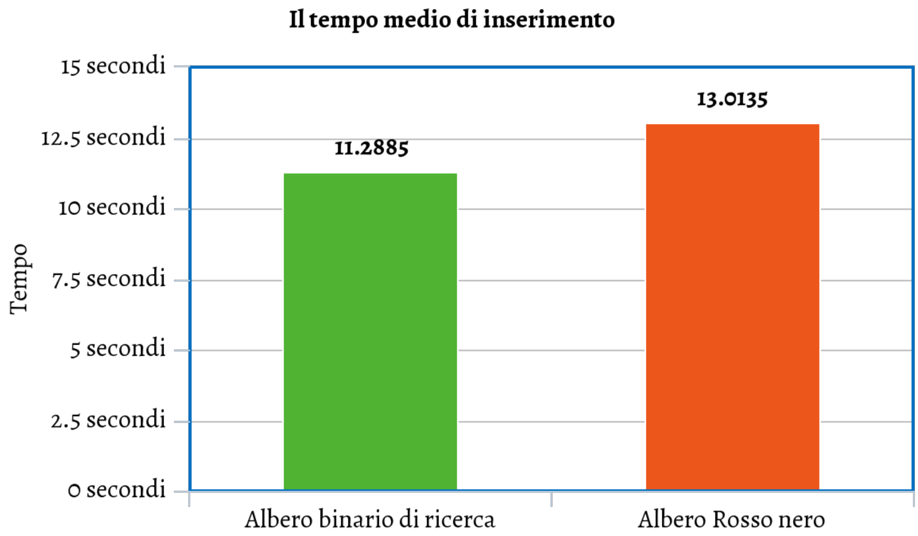


Figura 3.3: Bargraph di inserimento

Nella figura di sopra possiamo vedere il bargraph del tempo medio di inserimento. Si nota che un albero rosso nero ha un tempo maggiore.

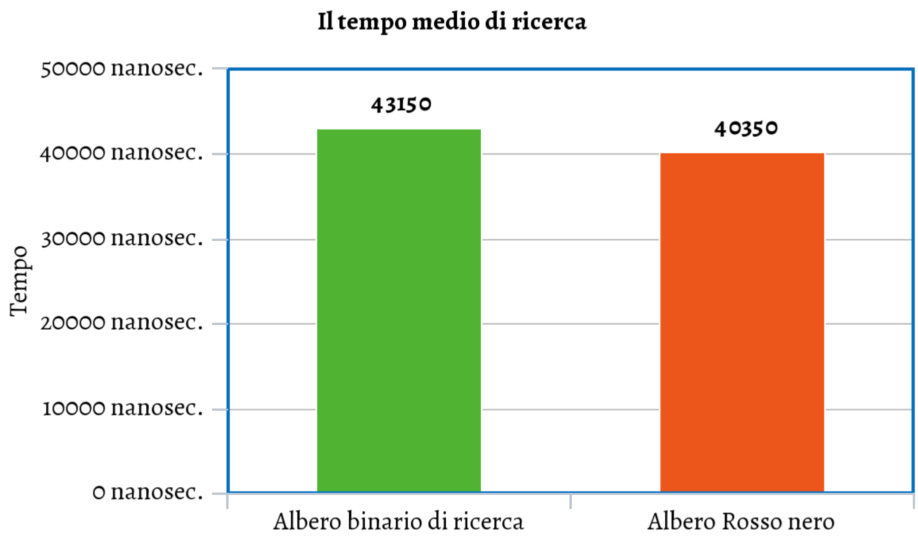


Figura 3.4: Bargraph di ricerca senza successo

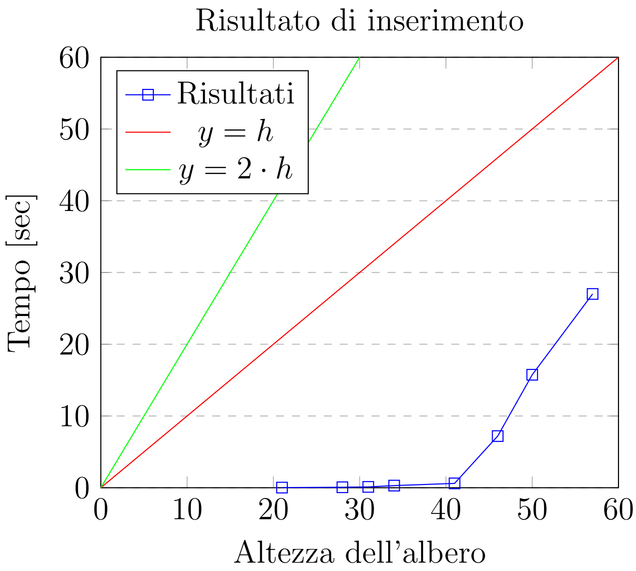
Invece da questo bargraph si capisce che il tempo di ricerca di un albero rosso nero è minore rispetto ad un albero binario di ricerca

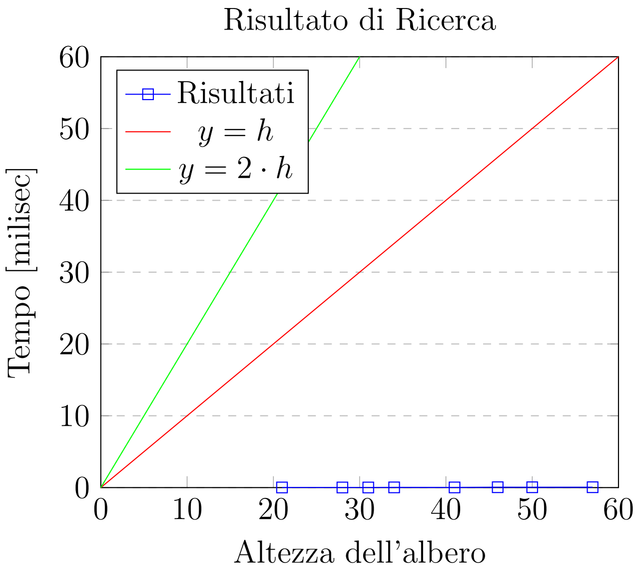
3.2 Analisi di alberi binari di ricerca

Abbiamo visto un confronto riguarda ai tempi di 2 strutture dati. Ora la domanda è da cosa dipende questo tempo. Teoricamente sappiamo che in un BST il tempo dipende dall'altezza di albero e in un RBT il tempo dipende dal numero di dati inseriti. A questo punto inseriamo un numero variabile di dati per verificare la teoria.

Risultato di BST				
Numero Esp.	N. Dati	Altezza albero BST	T. Inserimento BST (sec.)	T. Ricerca BST (nanosec.)
1	1000	21	0.0180	17600
2	5000	28	0.0670	15700
3	10000	31	0.1162	14700
4	20000	34	0.2965	29000
5	40000	41	0.6048	23900
6	300000	46	7.1981	51600
7	600000	50	15.7456	47200
8	1000000	57	27.0089	51500

Come vediamo il risultato ottenuto potrebbe verificare la teoria ma per capire meglio disegniamo il grafico





Come vediamo dal grafico il risultato ottenuto potrebbe essere un O grande di (*altezza del'albero*).

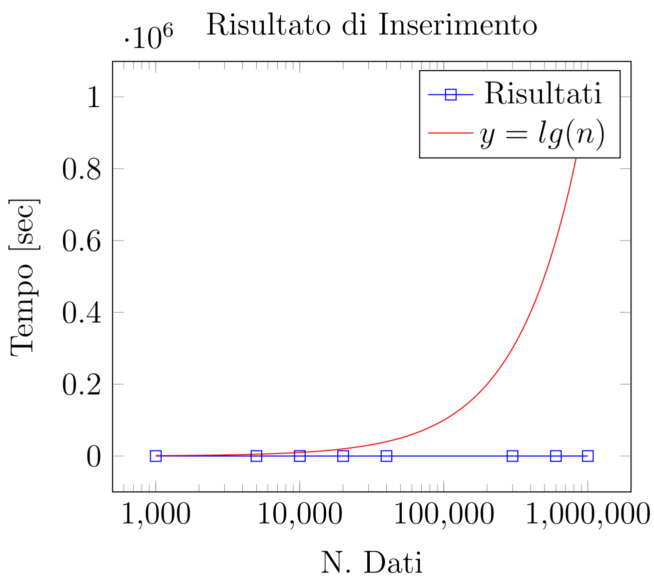
3.3 Analisi di alberi Rosso neri

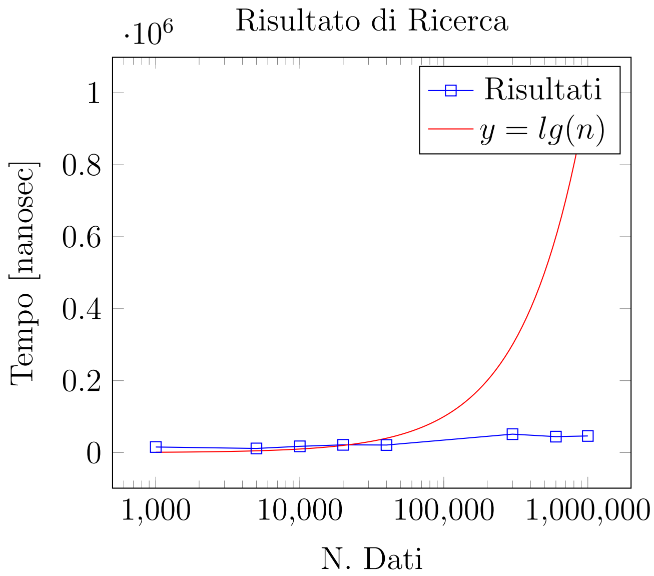
In questa sezione facciamo un'analisi sugli alberi rosso neri come avevamo fatto per BST. inseriamo un numero variabile di dati e cerchiamo un dato nell'albero per verificare la teoria.

Risultato di RBT				
Numero Esp.	N. Dati	Altezza albero RBT	T. Inserimento RBT (sec.)	T. Ricerca RBT (nanosec.)
1	1000	21	0.0275	15600
2	5000	28	0.0731	11600
3	10000	31	0.1350	17500
4	20000	34	0.2953	21600
5	40000	41	0.6470	21100
6	300000	46	7.9200	51300
7	600000	50	14.7611	44400
8	1000000	57	28.0601	46300

Questo risultato è ottenuto dagli stessi dati dell'esperimento su BST.

Vediamo il grafico dell'inserimento e la ricerca.





Come vediamo nelle figure di sopra, il risultato potrebbe essere un O grande di lg(n)