Alberi Binari di Ricerca

Confronto di diverse implementazioni

Silviu Leonard Vatamanelu

April 2023

Indice

1 Introduzione

Si vogliono confrontare vari modi per gestire chiavi duplicate negli Alberi Binari di Ricerca:

- Implementazione senza accorgimenti particolari;
- Implementazione mediante flag booleano;
- Implementazione mediante liste.

Per raggiungere lo scopo prefissato verranno implementate le strutture dati necessarie nel linguaggio Python.

Di seguito vengono indicate come riferimento le specifiche della macchina usata per svolgere gli esperimenti:

- CPU: Ryzen 5 4600U
- RAM: 16GB 3200MHZ
- OS: Windows 11 Pro 64-bit
- Ambiente: WSL con Ubuntu 20.04

2 Cenni Teorici

Un Albero binario di Ricerca è una struttura dati collegata composta da nodi. Ogni nodo ha 4 campi principali:

- Key: il valore rispetto al quale viene deciso il posizionamento di un nodo;
- Left: un puntatore al figlio sinistro;
- Right: un puntatore al figlio destro;
- Parent: un puntatore al padre.

Per memorizzare un Albero binario di Ricerca è quindi necessario mantenere il del nodo radice (Root) il quale avrà il campo Parent impostato a NIL. La proprietà principale dell'Albero binario di Ricerca è l'ordinamento: all'interno del sottoalbero sinistro di un qualsiasi nodo si hanno Key con valore inferiore o uguale a quello del nodo, mentre all'interno del sottoalbero destro si trovano solo Key con valore superiore o uguale a quello del nodo. L'inserimento all'interno di un Albero binario di Ricerca avviene quindi confrontando la propria Key con quella della Root, se il valore è minore o uguale si ripete la procedura nel sottoalbero sinistro altrimenti in quello destro, fino a che non si trova un NIL. Il punto di critico di questa procedura è l'inserimento delle chiavi duplicate. Esistono varie strategie per la loro gestione, vengono riportate solo quelle di interesse:

- Normale: si sceglie arbitrariamente un sottoalbero;

- Flag booleano: ogni nodo ha un campo aggiuntivo Flag che tiene traccia dell'ultimo sottoalbero visitato per l'inserimento;
- Liste: ogni nodo contiene una lista dove verrano inseriti tutti gli elementi con chiave uguale.

Ognuna delle implementazioni presenta vabtaggi e criticità: l'implementazione normale rende l'albero sbilanciato perché tutti i nodi uguali finiscono nello stesso sottoalbero, e nel caso in cui tutte le chiavi siano uguali si ha una lunga catena, in compenso è di semplice implementazione; l'implementazione mediante liste asintoticamente perde i vantaggi dell'Albero binario di Ricerca e somiglia più a una lista concatenata, in compenso le operazioni di cancellazione e inserimento sono estremamente semplici; mentre l'implementazione con flag garantisce un migliore bilanciamento dell'albero al costo di complicare l'operazione di rimozione di un nodo. Per apprezzare quindi le differenze tra le possibili implementazioni degli Alberi Binari di Ricerca bisogna concentrarsi su 3 aspetti fondamentali: Inserimento, Cancellazione e Bilanciamento dell'Albero.

3 Descrizione degli esperimenti

In seguito alle considerazioni svolte nella sezione 2 si ritiene necessario eseguire degli esperimenti sui seguenti metodi e sulle seguenti proprietà per confrontare le prestazioni delle 3 implementazioni:

1. Inserimento di nodi
2. Rimozione di nodi
3. Ricerca di un nodo
4. Profondità dell'albero

Gli esperimenti saranno quindi strutturati come segue: Per ogni implementazione di Albero binario di ricerca viene effettuato l'inserimento di un numero compreso tra $[10, 10000]$ nodi con passo 50 e con chiavi estratte casualmente all'interno degli intervalli $[0, 100]$ e $[0, 500]$. Tali intervalli vengono scelti in modo da avere sempre una buona probabilità di ottenere chiavi duplicate. Viene calcolata la profondità dell'albero, e successivamente si effettua la ricerca di un numero di chiavi pari a $\lfloor len(ABR.length)/4 \rfloor$ scelte casualmente dai valori inseriti nell'albero e la rimozione di tali elementi. Al termine viene calcolata nuovamente la profondità dell'albero.

Gli esperimenti vengono ripetuti 100 volte per ogni intervallo e lunghezza dell'albero; dei dati con stesso tipo di implementazione e lunghezza dell'albero viene fatta la media e poi i dati risultanti vengono mostrati sotto forma di grafico.

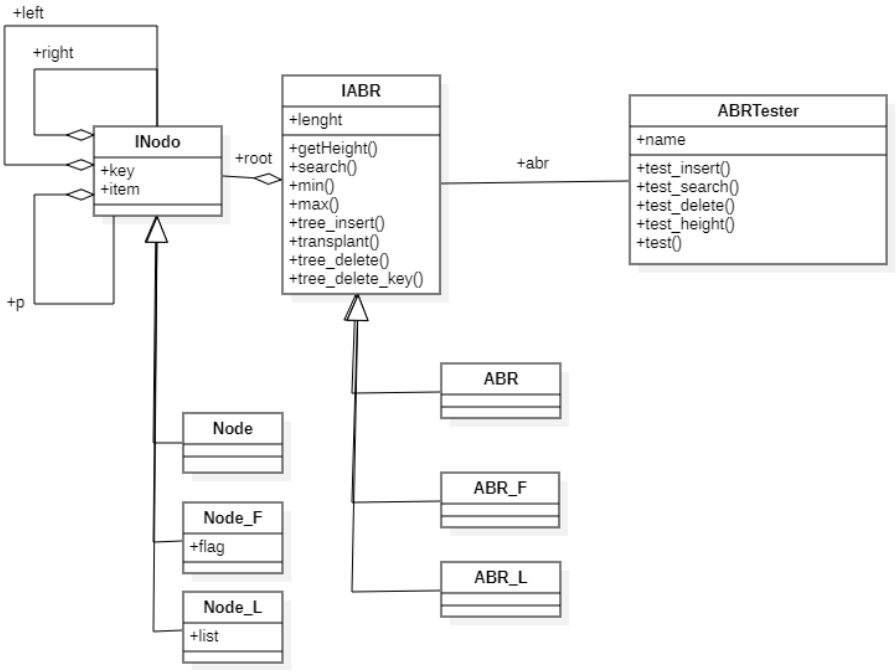

Fig. 1: Diagramma UML

4 Documentazione del codice

La figura 4 rappresenta il diagramma delle classi utilizzato per la soluzione. È importante notare che nell'implementazione del codice, ogni derivazione di IABR implementa la relativa derivazione di INode a lui compatibile, quindi ABR utilizzerà la classe Node, ABR_F utilizzerà Node_F e ABR_L utilizzerà Node_L. Nelle varie implementazioni cambiano soltanto le operazioni effettuate dai metodi `tree_insert(), tree_search(), tree_delete_key()` nei quali cambia la gestione del nodo di riferimento.

4.1 Inserimento

Ogni implementazione comprende i metodi `tree_insert() e __tree_insert()`. Il primo si ocupa di creare un nuovo nodo a partire dai dati in input e richiamare il secondo. L'algoritmo di inserimento, infatti, è interamente contenuto in `__tree_insert()` di cui si riporta l'implementazione nel frammento 5.

```
1     def __tree_insert(self, z: Node):
2         # Inserisce un nodo z all'interno dell'albero
3         y = None
4         x = self.root
5         while x is not None:
6             y = x
7             if z.key < x.key:
8                 x = x.left
9             else:
10                x = x.right
11        z.p = y
12        if y is None:
13            self.root = z
14        elif z.key < y.key:
15            y.left = z
16        else:
17            y.right = z
18        self.length += 1
```

Listing 1: Algoritmo di inserimento base

Tale algoritmo è implementato all'interno della classe ABR. ABR_F e ABR_L, rispetto alla variante base presentano delle variazioni alla prima della riga 7 e della riga 14 , indicate nei frammenti seguenti con dei commenti.

```
1     ## riga 7
2     if z.key == x.key:
3         break
4     ##riga 14
5     elif z.key == y.key:
6         y.list.append(z)
```

Listing 2: Algoritmo di inserimento nell'implementazione con Lista

Nel frammento 2, prima della riga 7 del frammento **??** viene inserito un controllo di uguaglianza per fermare la discesa all'interno dell'albero per la ricerca di un figlio vuoto per l'inserimento, in quanto ogni nodo mantiene una lista di tutti i nodi con chiave uguale. Ciò significa che dobbiamo inserire all'interno della lista del nodo y l'elemento z appena creato, come mostrato dal codice da inserire prima della riga 14.

```
1     ##riga 7
2     if z.key == x.key:
3         # Controllo il flag con logica inversa per decidere se
      inserire il nodo nel sottoalbero sinistro o destro
4         if x.flag:
5             x = x.right
6         else:
7             x = x.left
8         # se l'inserimento non avviene al prossimo livello, cambio
      il flag in modo da bilanciare i molti valori ripetuti
9         if x is not None:
10            y.flag = not y.flag
11    ##riga 14
12    elif z.key == y.key:
13        # Controllo il flag per decidere se inserire il nodo nel
      sottoalbero sinistro o destro del padre
```

```
14          if y.flag:
15              y.right = z
16          else:
17              y.left = z
18          # Cambio il flag
19          y.flag = not y.flag
```

Listing 3: Algoritmo di inserimento nell'implementazione con flag

Nel frammento 3, prima della riga 7 del frammento **??** viene inserito un controllo di uguaglianza perdecidere in quale sottoalbero proseguire la ricerca, in quanto ogni nodo mantiene un flag che indica la direzione. Viene poi inserito un controllo per evitare di scendere nell'albero e trovare all'altezza successiva un 'posto' per l'inserimento in modo da non cambiare 2 volte il flag del padre. Nel codice da inserire prima della riga 14 viene controllato il flag del padre e viene deciso il figlio in cui eseguire l'inserimento.

4.2 Ricerca

Ogni implementazione della ricerca comprende i metodi `search() e __search()`. Il primo si occupa chiamare il secondo a partire dalla radice dell'albero con la chiave da cercare. L'algoritmo di ricerca, infatti, è interamente contenuto in `__search()` di cui si riporta l'implementazione nel frammento 5.

```
1       def __search(self, x: Node, k: int) -> Node:
2           # Metodo ricorsivo che cerca un nodo con chiave k nell'
        albero
3           if x is None or x.key == k:
4               return x
5           if k < x.key:
6               return self.__search(x.left, k)
7           else:
8               return self.__search(x.right, k)
```

Listing 4: Algoritmo di ricerca base

Tale algoritmo è implementato all'interno della classe ABR e ritorna il primo nodo con chiave k. ABR_F e ABR_L, rispetto alla variante base non presentano variazioni in quanto la ricerca si ferma alla prima occorrenza quindi non è necessario prevedere accorgimenti particolari. Diverso sarebbe per una ricerca completa di tutte le chiave, in quanto l'algoritmo base dovrebbe tenere traccia dei nodi incontrati e continuare la ricerca in un singolo sottoalbero dato che l'inserimento avviene sempre nel sottoalbero destro. Per quanto riguarda l'implementazione con flag la ricerca deve avvenire in entrambi i sottoalberi in quanto non siamo a conoscenza del numero di occorrenze della chiave duplicata, mentre la variante con lista non richiede accorgimenti aggiuntivi per laricerca del nodo, ma la ricerca di un elemento specifico deve poi avvenire nella lista.

4.3 Cancellazione

Ogni implementazione comprende i metodi `tree_delete(), transplant()` e `tree_delete_key()`. I primi due cancellano un nodo ed evitano l'inconsistenza

dell'albero, mentre l'ultimo effettua l'eliminazione di tutti i nodi con chiave uguale. Di seguito l'implementazione per l'albero normale.

```
1       def transplant(self, u: Node, v: Node):
2        # Sostituisce il nodo u con il nodo v
3        if u.p is None:
4            self.root = v
5        elif u == u.p.left:
6            u.p.left = v
7        else:
8            u.p.right = v
9        if v is not None:
10           v.p = u.p
11
12   def tree_delete(self, z: Node):
13       # Elimina il nodo z dall'albero
14       if z.left is None:
15           self.transplant(z, z.right)
16       elif z.right is None:
17           self.transplant(z, z.left)
18       else:
19           y = self.min(z.right)
20           if y.p != z:
21               self.transplant(y, y.right)
22               y.right = z.right
23               y.right.p = y
24           self.transplant(z, y)
25           y.left = z.left
26           y.left.p = y
27
28       self.length -= 1
29
30   def tree_delete_key(self, key: int):
31       z = self.search(key)
32       self.__tree_delete_key(key, z)
33
34   def __tree_delete_key(self, key: int):
35       v = node.right
36       if v is not None and v.key == key:
37           self.__tree_delete_key(key, v)
38       self.tree_delete(node)
```

Listing 5: Algoritmo di eliminazione base

In `__tree_delete_key()` per eliminare tutte le occorrenze di una chiave è sufficiente controllare il sottoalbero destro. ABR_F e ABR_L, rispetto alla variante base presentano delle variazioni soltanto nel metodo `tree_delete_key() di cui vengono riportate le`

```
1    def tree_delete_key(self, key: int):
2        z = self.search(key)
3        if z is not None:
4            self.tree_delete(z)
```

Listing 6: Algoritmo di eliminazione nell'implementazione con Lista

Nel frammento 6, non è necessario il metodo ausiliario ricorsivo per cercare tutte le occorrenze dato che si trovano tutte all'interno dello stesso nodo. È sufficiente quindi cancellare solo il nodo z.

```
    def __tree_delete_key(self, key: int, node: None) -> bool:
        # Elimina i nodo con chiave k dall'albero, per ini
        if node is None:
            return
        else:
            if node.flag:
                u = node.left
                v = node.right
            else:
                u = node.left
                v = node.right

            if u is not None and u.key == key:
                self.__tree_delete_key(key, u)
            if v is not None and v.key == key:
                self.__tree_delete_key(key, v)
            self.tree_delete(node)
```

Listing 7: Algoritmo di eliminazione nell'implementazione con flag

Nel frammento 8, come nel caso della ricerca devono essere controllati entrambi i sottoalberi in quanto non abbiamo informazioni riguardo all'occorrenza dei duplicati. Possiamo velocizzare l'algoritmo iniziando l'eliminazione dal sottoalbero in cui è stato effettuato l'ultimo inserimento.

Per brevità non si riportano le implementazioni di `max()`, `min()` e `getHeight()`.

4.4 Test

La classe ABRTester implementa le procedure per registrare i tempi e l'altezza dell'albero in seguito alle varie operazioni. Si riporta il codice usato nel metodo test:

```
    def test(self, values: list) -> pd.DataFrame:
        # colonne: n, abr_type, height_before, height_after,
    insert_time, search_time, delete_time
        # righe: 1 riga per ogni test
        n = len(values)
        # inserimento dei valori nell'albero
        insert_times = []
        for v in values:
            insert_times.append(self.test_insert(v, chr(v)))
        insert_df = pd.DataFrame(insert_times)
        height_before = self.test_height()
        # ricerca dei valori nell'albero
        search_times = []
        values2 = np.random.choice(values, int(np.floor(n / 4)))
        for v in values2:
            search_times.append(self.test_search(v))
        search_df = pd.DataFrame(search_times)
        # eliminazione dei valori dall'albero
        delete_times = []
        for v in values2:
            delete_times.append(self.test_delete(v))
        delete_df = pd.DataFrame(delete_times)
        height_after = self.test_height()
        # creazione del dataframe
```

```
24          return pd.DataFrame({
25              'n': n,
26              'abr_type': self.type,
27              'height_before': height_before,
28              'height_after': height_after,
29              'insert_time': insert_df.mean(),
30              'search_time': search_df.mean(),
31              'delete_time': delete_df.mean()
32          })
33
34      def test_delete(self, key: int) -> float:
35          start = perf_counter()
36          self.ABR.tree_delete_key(key)
37          end = perf_counter()
38          return end - start
39
40      def test_height(self) -> int:
41          return self.ABR.getHeight(self.ABR.root)
```

Listing 8: Algoritmo di test

I metodi `test_insert()` e `test_search()` non vengono riportati in quanto del tutto simili a `test_delete()`. Il codice fa quanto descritto nella sezione 3: inserisce n valori e registra il tempo di ognuno attraverso dei `perf_counter()`, viene poi calcolata l'altezza dell'albero. Vengono scelti dei valori casuali tra quelli inseriti per effettuare la ricerca e poi la cancellazione, e viene registrata l'altezza dell'albero in seguito alla cancellazione. I dati vengono restituiti sotto forma di DataFrame contente la media dei tempi. Viene scelta la media in modo da poter analizzare gli algoritmi nel loro complesso in quanto la scelta dei massimi avrebbe portato a tempi influenzati dalle ultime operazioni eseguite in inserimento e dalla posizione nell'albero per l'eliminazione e la ricerca. Viceversa per il minimo: Il primo inserimento è quello con tempo minimo mentre la ricerca e l'eliminazione hanno il tempo minimo se il valore si trova "in alto" nell'albero e non ci sono altri valori.

L'operazione di test viene ripetuta dal main del programma che viene riportato nel frammento numero 9.

```
1  RANDOM_RANGES = [(0, 100), (0, 500)]
2  DATA_MEAN = 100
3  ITERATIONS = (10, 10000)
4  STEP = 50
5  for random_range in RANDOM_RANGES:
6      df_list: list = []
7      for d in range(DATA_MEAN):
8          df_app: list = []
9          value_list: list = []
10         for i in range(ITERATIONS[0], ITERATIONS[1], STEP):
11             value_list.append(np.random.randint(random_range[0],
       random_range[1], i))
12         for i in range(ITERATIONS[0], ITERATIONS[1], STEP):
13             df_app.append(ABRTester(lista.ABR(), 'lista').test(
       value_list[(i - ITERATIONS[0]) // STEP]))
14         for i in range(ITERATIONS[0], ITERATIONS[1], STEP):
15             df_app.append(ABRTester(flag.ABR(), 'flag').test(
       value_list[(i - ITERATIONS[0]) // STEP]))
```

```
16          for i in range(ITERATIONS[0], ITERATIONS[1], STEP):
17              df_app.append(ABRTester(abr.ABR(), 'abr').test(
        value_list[(i - ITERATIONS[0]) // STEP]))
18
19          print(f'{d} - {random_range[1]}')
20          df_list.append(pd.concat(df_app, ignore_index=True))
21
22      df_concat = pd.concat(df_list)
23      by_row_index = df_concat.groupby([df_concat.index, 'n', '
        abr_type'])
24      df = by_row_index.mean(numeric_only=True).reset_index()
25      # salvataggio dei grafici omesso
```

Listing 9: Algoritmo di test

Viene riportato il codice per puntualizzare l'otternimento dei dati: Il ciclo esterno viene eseguito su ogni intervallo da cui vengono scelti i numeri casuali; il ciclo interno esegue `DATA-MEAN` volte la generazione dei dati, e il test di ogni implementazione su quei dati. I cicli più interni sono separati per una più semplice lettura del DataFrame risultante, ogni test però lavora sugli stessi dati in ingresso generati prima dei test. I dati risultanti vengono concatenati e raggruppati per dimensione e tipo di albero e viene effettuata una media dei valori così ottenuti. Il DataFrame finale viene poi rappresentato sottoforma di grafico in tutte le sue parti con l'asse y in scala logaritmica.

5 Risultati degli esperimenti

In questa sezione verranno riportati i grafici per tutte le informazioni ottenute dall'esecuzione degli algoritmi secondo le modalità descritte nelle sezioni 3 e 4.4.

5.1 Inserimento

Da questo esperimento ci aspettiamo un comportamento simile per le implementazione normale e con flag, proporzianale all'altezza dell'albero i.e. $O(h)$. Mentre un comportamento inizialmente simile per la rimanente implementazione ma poi peggiore dovuto ai tanti duplicati e all'inseriemento sequenziale nella lista. Ci aspettiamo dall'intervallo più ampio un minore calo di prestazioni per la lista.

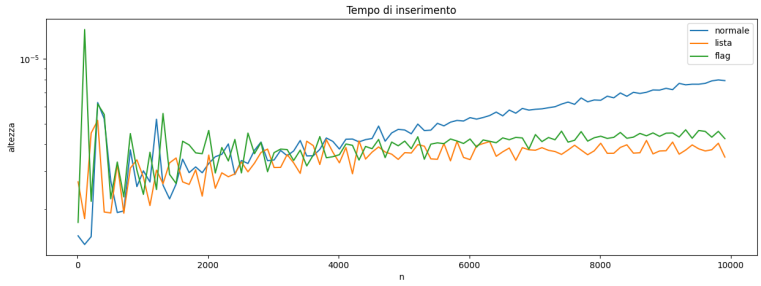

Fig. 2: Tempo di inserimento con intervallo $[0, 100]$

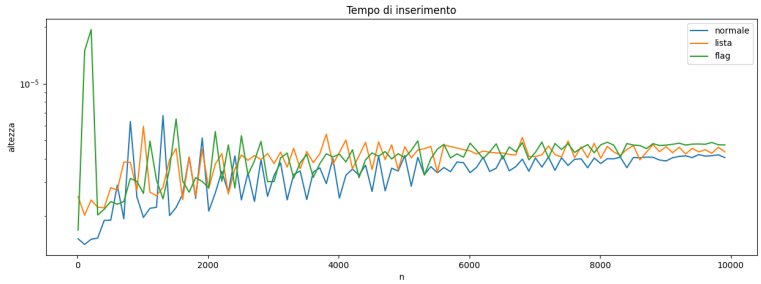

Fig. 3: Tempo di inserimento con intervallo $[0, 500]$

I risultati rappresentati nelle figure 2 e 3 confermano parzialmente quanto intuito: l'implemezione mediante liste non peggiora all'aumentare della dimensione dell'albero; ciò e probabilmente dovuto all'implementazione del metodo `append()` in Python. Le tempistiche per le implementazioni con flag e normale, invece, seguono un andamento molto simile.

5.2 Altezza prima dell'eliminazione

Da questo esperimento ci aspettiamo di vedere i risultati peggiori per l'implementazione normale, mentre i migliori per l'implementazione con le liste. Questo perchè l'altezza dell'albero, per inserimenti casuali, va come $O(\log n)$ dove n è il numero di chiavi uniche, quindi l'implementazione a liste è avvantaggiata in quanto tratta le chiavi come univoche e le duplicate risiedono nello stesso nodo. Tra l'implementazione normale e quella con i flag, possiamo aspettarci risultati migliori dalla seconda in quanto cerca di bilanciare i valori duplicati.

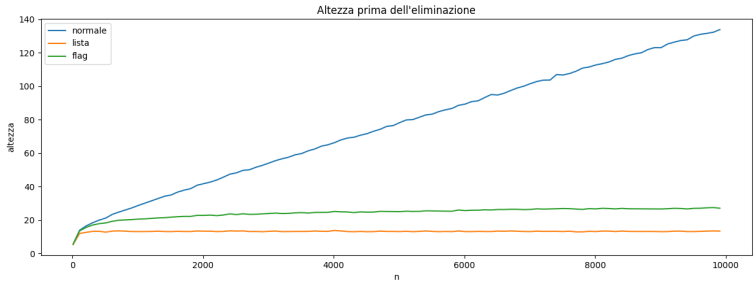

Fig. 4: Altezza con intervallo $[0, 100]$

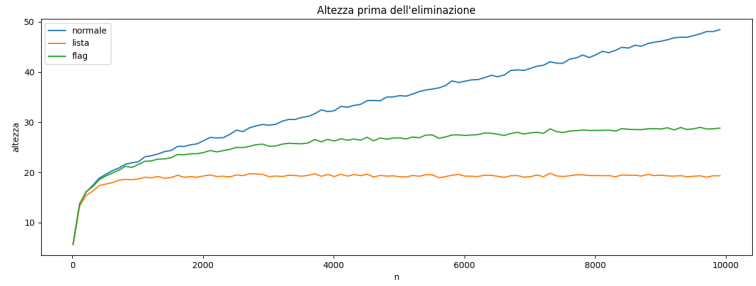

Fig. 5: Altezza con intervallo $[0, 500]$

I risultati rappresentati nelle figure 4 e 5 confermano quanto intuito: l'implemezione mediante liste risulta piu efficace in quanto ha un altezza al piu pari al logaritmo dell'estremo superiore dell'intervallo utilizzato. Invece, le altre due implementazioni vengono influenzate dalle numerose chiavi duplicate presenti.

5.3 Ricerca

Dato che la ricerca viene effettuata sul primo elemento con chiave uguale, ci aspettiamo un andamneto molto simile per tutte le implementazioni, diverso caso per la ricerca di tutti gli elementi. Dato che la ricerca viene effettuata per chiave e non per valore, non si riescono a riscontrare gli svantaggi dell'implementazione con liste.

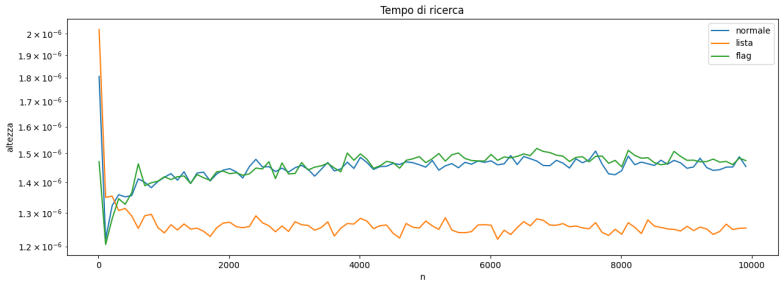

Fig. 6: Tempo di ricerca con intervallo $[0, 100]$

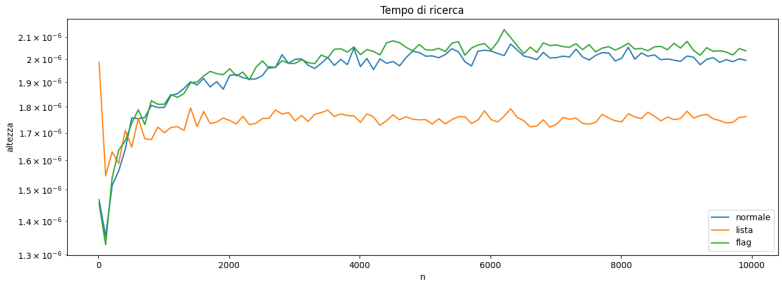

Fig. 7: Tempo di ricerca con intervallo $[0, 500]$

I risultati rappresentati nelle figure 6 e 7 confermano quanto intuito: le implementazioni condividono tempistiche molto simili. L'implementazione mediante liste risulta però leggermente migliore in quanto l'albero che ne risulta è piu basso.

5.4 Eliminazione

Da questo esperimento ci aspettiamo un comportamento simile per le implementazione normale e con flag, proporzianale all'altezza dell'albero i.e. $O(h)$. Mentre risultati simili alla ricerca per l'implemetazione con liste. Anche in questo caso è da notare che l'eliminazione è per chiave e non per valore quindi viene eliminato il costo della ricerca del valore corretto nella lista dei nodi duplicati.

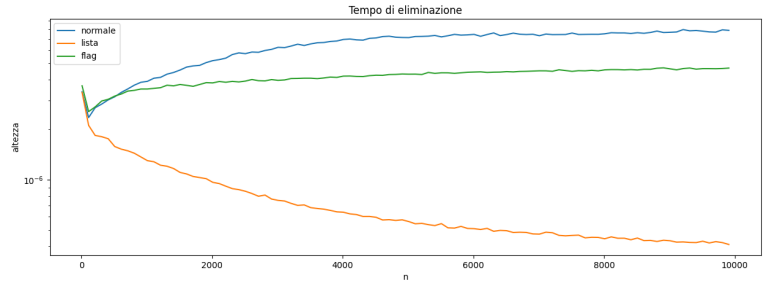

Fig. 8: Tempo di eliminazione con intervallo $[0, 100]$

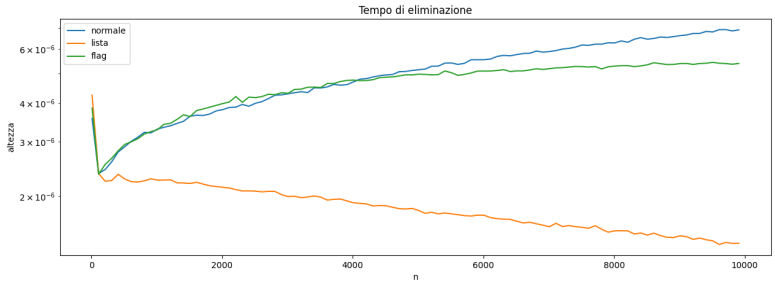

Fig. 9: Tempo di eliminazione con intervallo $[0, 500]$

I risultati rappresentati nelle figure 8 e 9 confermano quanto intuito: l'implemezione risulta migliore delle altre due per la minore altezza dell'albero risultante. La componente costante verso la coda di tutte e tre le curve è dovuta allo svuotamento dell'albero e al conseguente tempo costante per la ricerca della chiave da eliminare.

5.5 Altezza dopo l'eliminazione

Questo esperimento viene eseguito per ottenere delle informazioni riguardo uno scenario tipico di utilizzo della struttura dati in cui un utente effettua degli inserimenti e poi delle cancellazioni. L'altezza in seguito a questo scenario è rilevante perchè influenza le operazioni successive. Da questo esperimento ci aspettiamo di vedere una decrescita lineare per l'implementazione a liste, mentre un andamento più irregolare per le altre due implementazioni.

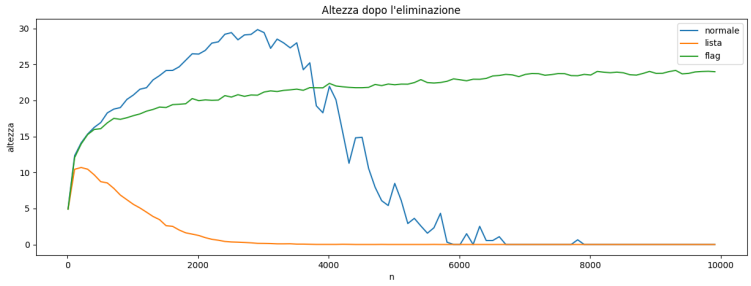

Fig. 10: Altezza con intervallo $[0, 100]$

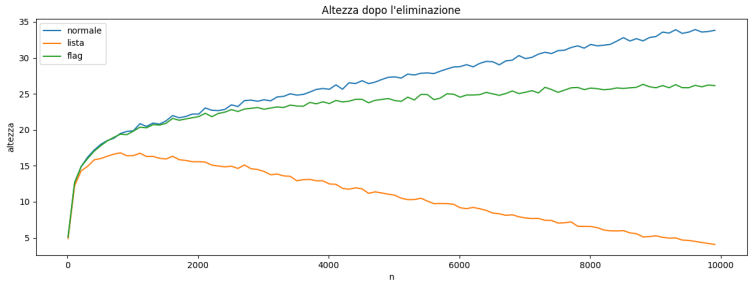

Fig. 11: Altezza con intervallo $[0, 500]$

I risultati rappresentati nelle figure 10 e 11 confermano parzialmente quanto intuito: l'implemezione mediante liste decresce velocemente. I valori tra 0 e 1 sono dovuti alla casualità dei numeri generati e sono chiaramente dovuti all'intervallo usato e al metodo di scelta dei valori da cancellare. Risulta poco comprensibile l'andamento i figura 10 della variante con flag il quale sembra non effettuare la cancellazione.

6 Conclusioni

In seguito agli esperimenti svolti e alle considerazioni fatte, possiamo affermare che l'implemetazione con le liste, per le operazioni svolte è la piu vantaggiosa. La ricerca può essere migliorata includendo anche le prestazioni sulla ricerca e sulla cancellazione del singolo valore.