UNIVERSITÀ
DEGLI STUDI
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laboratorio di Algoritmi e Strutture Dati

Autore:
Lucian Alexandru Topliceanu

N° Matricola:
7003550

Corso principale:
Algoritmi e Strutture Dati

Docente corso:
Simone Marinai

Indice

1 Introduzione

1.1 Obiettivo da raggiungere

Vogliamo confrontare varie implementazioni per un dizionario, in particolare come struttura dati alla base dei dizionari avremmo:

- Lista concatenata
- ABR
- HashHeap

Dovremmo analizzare in termini di prestazioni le funzionalità principali di un dizionario, confronteremo per ogni struttura dati:

- L'inserimento
- La ricerca
- La rimozione

Ciò, al fine di capire in quali condizioni un dizionario implementato con una certa struttura dati ottiene dei risultati migliori in confronto ad altre implementazioni.

1.2 Specifiche del ambiente di test

I dati ottenuti durante i test sono dei dati che dipendono drasticamente dal ambiente di esecuzione, in particolare l'hardware e il compilatore possono influenzare sui valori dei dati. Se l'hardware e il compilatore sono migliori, migliori saranno i valori dei dati. I test saranno eseguito su un dispositivo avente il seguente hardware:

- Processore Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, 2808 Mhz, 4 core, 8 processori logici
- Scheda madre ASUSTeK COMPUTER INC. X580VN
- Memoria Samsung M471A2K43CB1-CRC DDR4 16GB 2400MHz Single channel
- Disco WD BLACK SN850X 1000GB

Come abbiente di programmazione è stato utilizzato Visual Studio Code version 1.87 con Python v2024.2.1

2 Analisi a priori

2.1 Introduzione

In questo sezione tratteremo i aspetti più teorici delle strutture dati al fine di definire un analisi a priori sul set di dati forniti, successivamente, dai test. Analizzeremo le caratteristiche principali di ogni struttura dati e il funzionamento del inserimento, la ricerca e l'eliminazione di un elemento.

2.2 Lista concatenata

Una lista concatenata è una struttura dati i cui oggetti sono disposti in ordine lineare. Diversamente da un array in cui l'ordine lineare è determinato dagli indici dell'array, l'ordine di una lista concatenata è determinato da un puntatore in ogni oggetto. Ogni elemento di una lista concatenata contiene un attributo valore e un puntatore al prossimo elemento della lista, nel nostro caso non avremo solo un puntatore al elemento successivo ma anche uno a quello precedente rendendo la lista doppiamente concatenata.

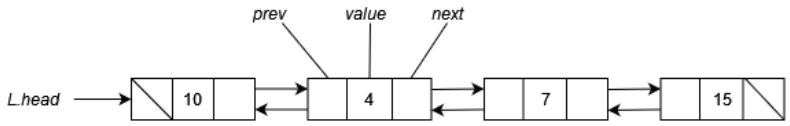


Figura 1: Una lista concatenata L contenente l'insieme 10,4,7,15. Dove l'attributo $L.head$ è un puntatore al primo elemento della lista se nullo implica che la lista è vuota, $prev$ è un puntatore al elemento precedente, $value$ contiene il valore da salvare e $next$ è un puntatore al elemento successivo.

Inserimento

Sia dato un elemento x il cui attributo $value$ sia stato già inserito, l'elemento x verrà inserito d'avanti alla lista concatenata. $L.head$ punterà al elemento x, $x.next$ punterà al elemento precedentemente puntato da $L.head$ e l'attributo $prev$ di questo ultimo punterà a x. Su una lista di n elementi il tempo di inserimento è $O(1)$.

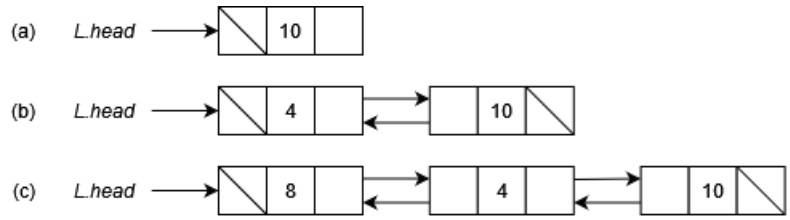


Figura 2: (a) Una lista concatenata L avente l'insieme 10, (b) dopo l'inserimento di un nuovo valore 4 $L.head$ punta al nuovo valore aggiunto in testa il quale avrà il puntatore $next$ che punta al elemento precedentemente in testa e 10 avrà il puntatore $prev$ che punta al nuovo elemento inserito, la lista è composta da 4,10. (c) Come nella figura (b) c'è l'inserimento di un nuovo valore 8 in testa che avrà $next$ sul valore 4 rendendo l'insieme di valori di L 8,4,10.

Ricerca

La procedura di ricerca trova il primo elemento con il valore k nella lista L mediante una ricerca lineare, restituisce un puntatore a questo elemento e in caso cui il valore non è presente restituisce un valore nullo. Per effettuare una ricerca su una lista di n elementi, la procedura impiega il tempo $\Theta(n)$

Rimozione

Per rimuovere un elemento x dalla lista concatenata L di n elementi, avendo un puntatore a x, bisogna rimuovere l'elemento e aggiornare i puntatori del elemento precedente e di quello successivo. Il tempo impiegato nella rimozione è $O(1)$ ma per cancellare un elemento dato un valore bisogna prima effettuare una ricerca che può impiegare nel caso peggiore $\Theta(n)$.

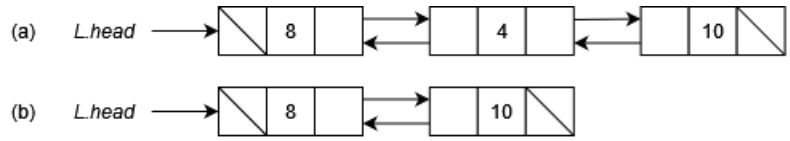


Figura 3: (a) Una lista concatenata L avente 3 valori 8,4,10. (b) La lista L dopo la rimozione del valore 4, il puntatore $next$ del elemento 8 punta al elemento 10 e il $prev$ di 10 punta a 8

2.3 ABR

Un ABR è organizzato in un albero binario, può essere rappresentato da una struttura dati concatenata in cui ogni nodo è un oggetto. Oltre a una chiave e ai suoi dati satelliti, ogni nodo del aero contiene gli attributi $left$, $right$ e p che puntano rispettivamente a:

- $left$ punta al figlio sinistro
- $right$ punta al figlio destro
- p punta al padre

Nel caso in cui a un nodo manca un figlio o il padre i puntatori risulteranno nulli, il nodo radice è l'unico nodo che ha l'attributo padre nullo. Le chiavi di un albero binario di ricerca sono sempre memorizzate soddisfacendo la seguente proprietà:

Sia x un nodo di un albero binario di ricerca. Se y è un nodo nel sotto albero sinistro di x, allora $y.key \leq x.key$.Se y è un nodo del sotto albero destro di x, allora $y.key \geq x.key$.

Le operazioni di inserimento, ricerca e rimozione impiegano $\Theta(h)$ nel caso peggiore, dove h è l'altezza del albero e nel caso in cui l'albero è degenere impiegano $O(n)$. Se si costruisce un albero binario di ricerca in modo casuale avremo che l'altezza attesa h è lgn ciò ci evita il caso degenere.

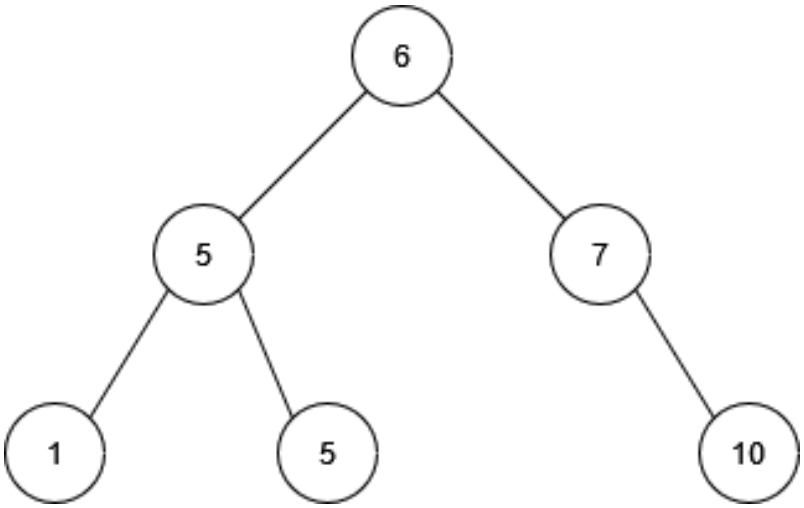


Figura 4: Un ABR contenente l'insieme di chiavi 6,5,1,5,7,10

Inserimento

Per inserire un elemento z in un albero binario di ricerca bisogna stare attenti a rispettare la proprietà dei alberi binari di ricerca, z sarà inserito in una posizione dove il figlio sinistro è minore e quello destro è maggiore. Deve anche risultare che se z è minore del padre allora si trova nel sotto albero sinistro, se maggiore in quello destro. Tale procedura viene eseguita nel tempo $O(h)$ in un albero di altezza h.

Ricerca

La ricerca di un elemento, data una chiave k, viene eseguita partendo dalla radice e verificando se la chiave k e minore o maggiore della chiave del nodo in esaminazione. Se minore, si ricerca la chiave nel sotto albero sinistro altrimenti in quello destro fino a trovare la chiave richiesta. I nodi incontrati durante l'iterazione formano un cammino semplice dal basso verso la radice, quindi il tempo d'esecuzione della ricerca è $O(h)$ dove h è l'altezza del albero binario di ricerca.

Rimozione

La rimozione di un nodo z da un albero binario di ricerca considera tre casi base:

- Se z non ha figli, modifichiamo suo padre $p[z]$ per sostituire z con un valore nullo come suo figlio.
- Se il nodo z ha un solo figlio, eleviamo questo figlio in modo da occupare la posizione di z nell'albero, modifichiamo il padre di z per sostituire z con il figlio di z.
- Se il nodo z ha due figli, troviamo il successore y di z (che deve trovarsi nel sotto albero destro di z) e facciamo in modo che y assuma la posizione di z nell'albero. La parte restante del sotto albero destro originale diventa il nuovo sotto albero destro di y e il sotto albero sinistro di z diventa il nuovo sotto albero sinistro di y.Questa operazione implica la necessita d'introdurre una nuova funzionalità che ci permetta di sostituire un sotto albero con un altro. Nel complesso la rimozione di un nodo z impiega un tempo $O(h)$ dove h è l'altezza del albero binario di ricerca.

2.4 HashHeap

HashHeap è una struttura dati che combina una tabella hash con un heap.

- Tabella hash: Una tabella hash è una struttura dati che permette di memorizzare coppie chiave-valore, consentendo di accedere rapidamente ai valori tramite chiave.
- Heap: Un heap è un albero binario che soddisfa alcune proprietà d'ordinamento, ad esempio se prendiamo in considerazione un max heap allora avremo che il padre di ogni nodo è maggiore del figlio.

L'idea base di un HashHeap è quella di mantenere sia una tabella hash che un heap binario. Gli elementi vengono inseriti nella tabella hash e nell'heap contemperarmene, mantenendo un collegamento tra i due. Ciò consente un accesso rapido agli elementi del heap tramite la tabella hash e un ordinamento efficiente tramite l'heap.

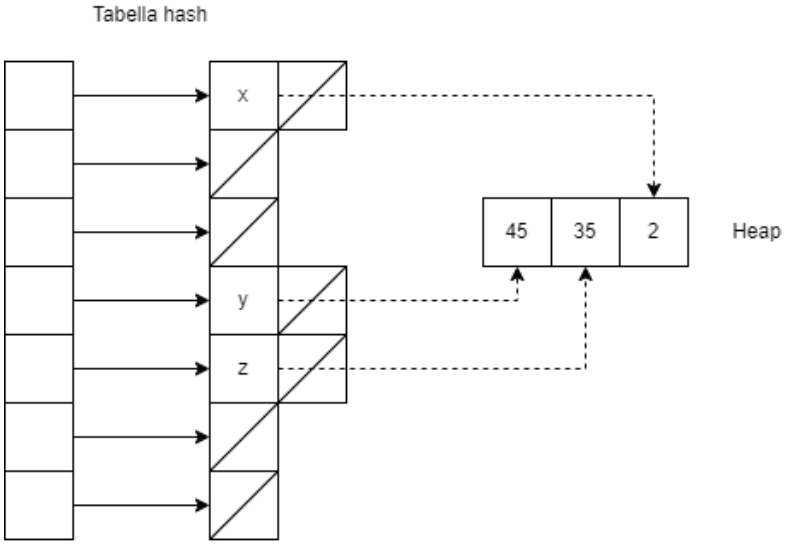


Figura 5: Un hashheap contente una tabella hash con gestione delle collisioni tramite lista concatenata e un heap. Nella tabella hash vengono memorizzate nella posizione $h(chiave)$, dove $h(x)$ è la funzione di hashing, un riferimento alla posizione del elemento corrispondente nel heap. In questo caso vengo memorizzate le coppie $(x, 2)$, $(y, 45)$ e $(z, 35)$

.

Inserimento

L'inserimento in un hashheap avviene mediante l'inserimento in una tabella hash e l'inserimento in un heap.

- HashHeap: L'inserimento di un elemento x nella tabella hash T avviene accedendo alla tabella nella posizione i, dove i è dato dal hashing della chiave $x.chiave$. Per la funzione di hashing viene utilizzato il metodo delle moltiplicazioni e ciò implica che $h(k) = \lfloor m(kAmod1) \rfloor$ dove k è la chiave, m la lunghezza della tabella e A una costante definita come $A : 0 < A < 1$. Alla posizione della tabella $T[h(x.chiave)]$ se non vi è stato inserito nessun altro elemento verrà inserito un puntatore alla testa di una lista concatenata su cui faremmo l'inserimento della nostra chiave a cui assoceremo un valore corrispondente alla posizione del elemento al interno del heap, altrimenti verrà eseguito direttamente un inserimento sulla lista concatenata puntata da $T[h(x.chiave)]$. La procedura d'inserimento dalla parte tabella hash impiega un tempo di hashing e accesso alla tabella pari a $O(1)$ più un tempo di inserimento in una lista concatenata pari a $O(1)$ essendo un inserimento in testa, ciò porta ad avere un tempo di inserimento di $O(1)$.
- Heap: Supponendo che il nostro heap sia memorizzato in un array A dove gli elementi sono memorizzati in modo da che il padre di un nodo i si trova nella posizione $i/2$, il figlio sinistro di un nodo i si trova nella posizione $i \times 2$ e il figlio destro nella posizione $i \times 2 + 1$. L'inserimento del nuovo nodo avviene in una posizione che rispetti la struttura del heap, se l'heap è un max heap il nodo dovrà essere maggiore dei figli altrimenti se la struttura è un min heap il nodo dovrà essere minore dei figli. La procedura per trovare la giusta posizione del nodo viene chiamata heapify, a seconda della struttura può essere max heapify o min heapify e impiega un tempo $O(lgn)$.

L'Inserimento di un elemento in un hashheap impiega un tempo dato dal inserimento di un elemento in una tabella hash gestita con liste concatenate per le collisioni $O(1)$ più il tempo di inserimento di un elemento in un heap $O(lgn)$ ciò implica che il tempo di inserimento totale è $O(lgn)$. Si può notare che se gli elementi da inserire sono ordinati la procedura di heapify non viene seguita essendo già verificata la proprietà di max o min heap questo porta l'inserimento a un tempo $O(1)$.

Ricerca

La ricerca di un elemento data una certa chiave k avviene accedendo alla tabella hash T nella posizione $h(k)$ e facendo una ricerca della chiave nella lista concatenata puntata da $T[h(k)]$. Nella lista concatenata avremo un riferimento alla posizione del elemento associato alla chiave k nel heap, l'accesso nel heap sarà immediato. Il tempo della ricerca è dovuto principalmente dalla ricerca della chiave nella lista concatenata, questa ricerca impiega $O(1 + \alpha)$ dove α è il fattore di carico definito come $\alpha = n/m$.

Rimozione

La rimozione di un elemento in un hashheap è composta dalla rimozione di un elemento da un tabella hash e dalla rimozione di un nodo da un heap. La rimozione dalla tabella hash implica una ricerca del elemento desiderato che impiegherà un tempo $O(1+\alpha)$, sfruttando la ricerca effettuata sulla tabella hash per la rimozione del elemento possiamo accedere in tempo $O(1)$ al elemento da eliminare nel heap ma la sua rimozione comporterà una violazione delle proprietà del heap perciò dovremmo eseguire un heapify che impiegherà un tempo $O(lgn)$. Il tempo impiegato dalla rimozione di un elemento in un hashheap è $O(1 + \alpha) + O(lgn)$.

2.5 Conclusioni teoriche

Date le analisi teoriche delle sezioni precedenti possiamo riassumere per ogni struttura dati i tempi di esecuzione che ci aspettiamo in una tabella.

	Lista concatenata	**ABR**	**HashHeap**
Inserimento	$O(1)$	$O(lgh)$	$O(lgn)$
Ricerca	$O(n)$	$O(lgh)$	$O(1+\alpha)$
Rimozione	$O(n)$	$O(lgh)$	$O(1+\alpha)+O(lgn)$

Figura 6: Tempi di esecuzione del inserimento, ricerca e rimozione per la lista concatenata, ABR e HashHeap

Possiamo dedurre vantaggi e svantaggi per ogni implementazione:

- Lista concatenata: La lista concatenata ha un implementazione meno complessa in confronto alle altre strutture dati ed eccelle nel inserimento ma la ricerca e la rimozione hanno un tempo lineare che porta a tempi lunghi al crescere del numero degli elementi perciò l'implementazione è adatta per una quantità ridotto di elementi. L'ordine degli elementi non ne influenza i tempi di esecuzione.
- ABR: L'albero binario di ricerca ha una dipendenza dal ordine di inserimento degli elementi che influenza il tempo di esecuzione essendo composto dall'altezza del albero. Se gli elementi sono inseriti in modo casuale possiamo dire che l'albero ha altezza lgn e stabilisce tempi di esecuzione migliori della lista concatenata ma se gli elementi vengono inseriti in ordine ordinato l'altezza del albero risulta uguale a n portando a dei tempi di esecuzione analoghi alla lista concatenata, escludendo l'inserimento che diventerebbe $\theta(n)$ perciò in caso di elementi ordinati l'albero binario di ricerca non ottiene dei risultati migliori della lista concatenata.
- HashHeap: L'hashheap ha un implementazione più complessa in confronto alle altre strutture dati ma pur avendo un costo d'inserimento simile al albero binario di ricerca non ha un sequenza di elementi che ne peggiora il tempo di esecuzione. Possiamo dedurre che nel caso in cui i valori da inserire nel heap sono ordinati, conformi al max heap o min heap, heapify non viene eseguito riducendo il costo a $O(1)$. I tempi di esecuzione della ricerca e della rimozione sono legati al fattore di carico ma con una buona scelta della funzione di hashing e della dimensione della tabella ne possiamo ridurne l'impatto eccellendo sulla lista concatenata e sull'albero binario di ricerca nella maggior parte dei casi. L'hashheap a differenze delle altre strutture dati ha un impatto sulla memoria nettamente maggiore.

3 Documentazione del codice e scelte implementative

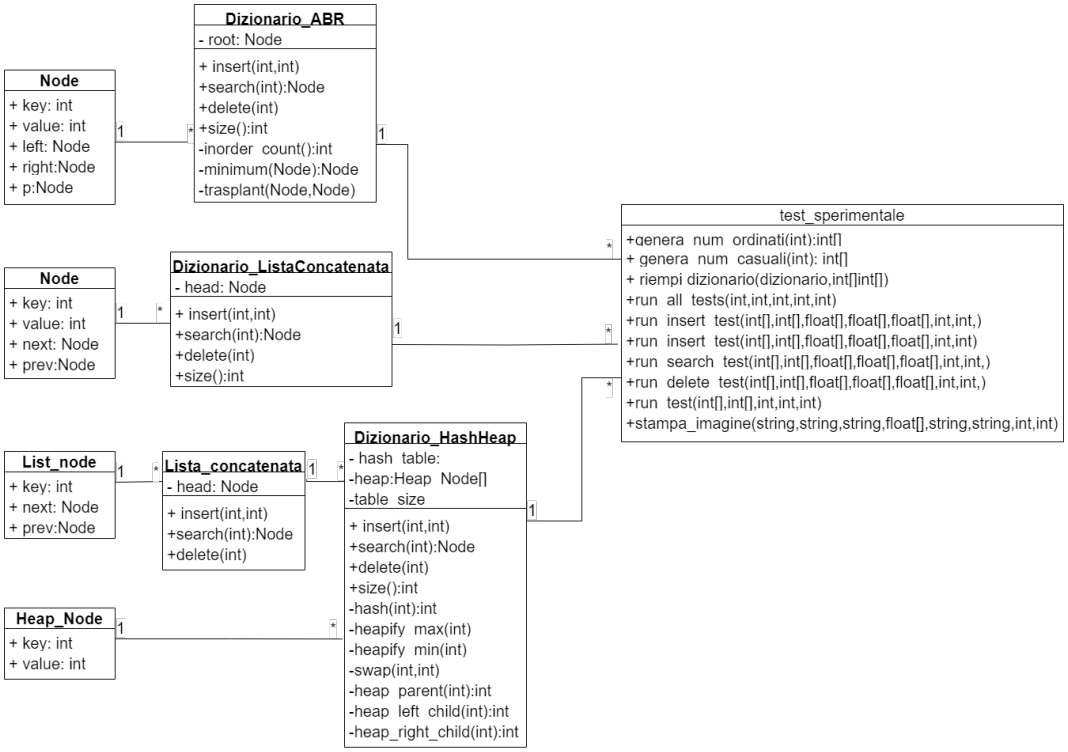


Figura 7: Diagramma delle classi implementate

Le classi che implementano i dizionari sono semplicemente delle implementazioni della loro struttura dati a differenza della classe test sperimentale che implementa il vero test. In particolare le funzioni implementate hanno lo scopo di:

- $genera\_num\_ordinati(numero\_elementi)$: La funzione genera una lista lunga $numero\_elementi$ con interi ordinati.
- $genera\_num\_casuali(numero\_elementi)$: La funzione genera una lista lunga $numero\_elementi$ con interi casuali.
- $riempi\_dizionario(dizionario, chiavi, valori)$: Dato un dizionario come riferimento e due liste di interi rispettivamente chiave-valore, la funzione inserisce tagli elementi nel dizionario.
- $run\_all\_tests(numero\_elementi, hash\_table\_size)$: Inizializza 3 test per ogni funzionalità dei dizionari. La funzione per ogni test genera due liste chiavi-valori e chiama l'esecuzione dei test per ogni dizionario preso in considerazione.
- $run\_test(chiavi, valori, hash\_table\_size, numero\_test, numero\_elementi)$: Esegue il test sui dizionari, ciò che differenzia un test da un altro sono i ingressi. La funzione è incaricata di chiamare i test specifici per i dizionari.
- $run\_insert\_test(chiavi, valori, t\_lista, t\_abr, t\_hash\_heap, numero\_iterazioni, hash\_table\_size)$: La funzione esegue e cronometra la chiamata del inserimento sul dizionario al crescere n fino ad arrivare al ultimo elemento.
- $run\_search\_test(chiavi, valori, t\_lista, t\_abr, t\_hash\_heap, numero\_iterazioni, hash\_table\_size)$: La funzione esegue e cronometra la chiamata della ricerca di un elemento sul dizionario al crescere n fino ad arrivare al ultimo elemento.
- $run\_delete\_test(chiavi, valori, t\_lista, t\_abr, t\_hash\_heap, numero\_iterazioni, hash\_table\_size)$: La funzione esegue e cronometra la chiamata della rimozione di un elemento sul dizionario al crescere n fino ad arrivare al ultimo elemento.
- $stampa\_imagine(title, x\_lab, y\_lab, tempi, nome\_struttura, colore, numero\_test, numero\_elementi)$: La funzione mediante la libreria matplotlib salva l'immagine di un grafico contenente i tempi di esecuzione per il numero di elementi.

Per l'hashheap come funzione di hashing è stato scelto il metodo delle moltiplicazioni con A definita come la costante consigliata da Knuth $(\sqrt{5}–1)/2$ e come dimensione della tabella di hash $m = 51$. Nel eseguire i test, la scelta di non eseguirne un quarto test con chiavi ordinate e valori ordinati è stata presa perché non avrebbe portato dati non già presenti nei altri test.

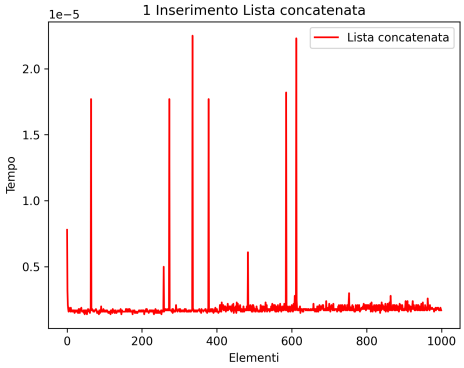
4 Test

Di seguito verranno riportati i risultati dei test, verranno eseguiti 3 su un massimo di 1000 elementi. I test differiscono per l'ordinamento delle chiavi o valori. Durante l'esaminazione dei risultati si potrà notare dei dati incongruenti, questi dati possono essere il risultato del compimento di altri task con priorità maggiore di Windows che influiscono sul tempo di completamento di un test.

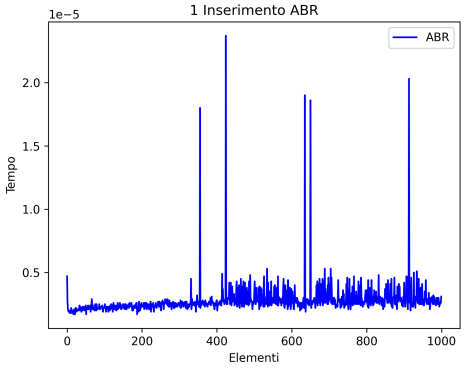
- Test 1: Il test verrà eseguito con un lista di chiavi casuali e una lista di valori casuali.
- Test 2: Il test verrà eseguito con un lista di chiavi ordinate e una lista di valori casuali.
- Test 3: Il test verrà eseguito con un lista di chiavi casuali e una lista di valori ordinati.

4.1 Test 1

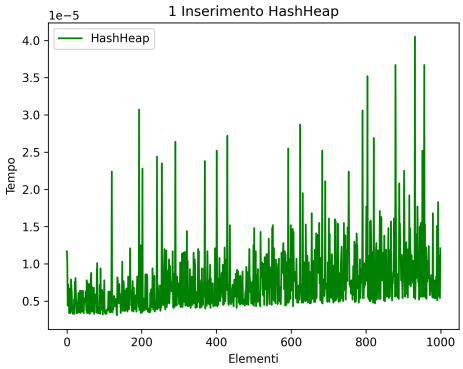
Nel test i dati sono stati raccolti su un set di dati chiave-valore con ordinamento casuale.



(a)

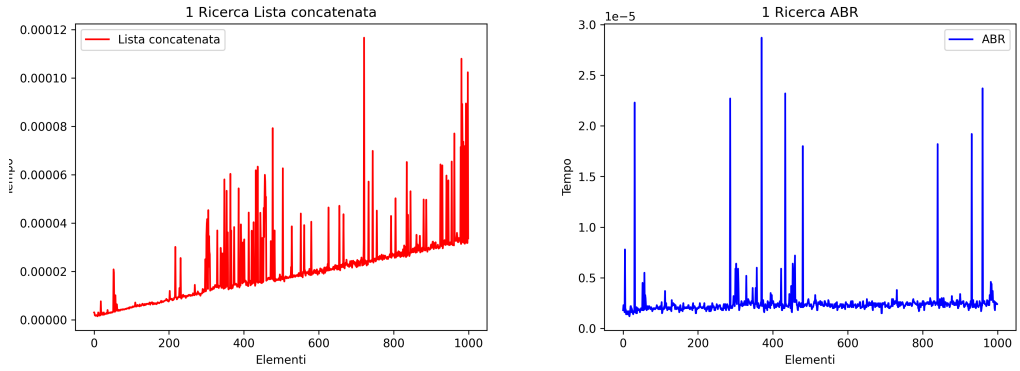


(b)



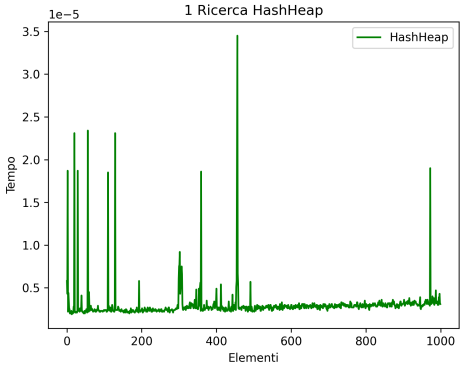
(c)

Figura 8: Grafici basati sui tempi di esecuzione per l'inserimento di un elemento con n elementi precedentemente inseriti. (a) Inserimento per la lista concatenata. (b) Inserimento per l'albero binario di ricerca. (c) Inserimento per l'hasheap



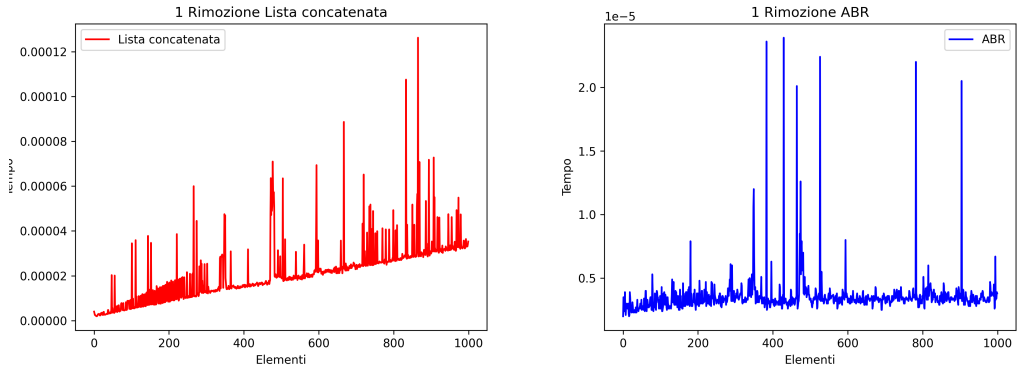
(a)

(b)



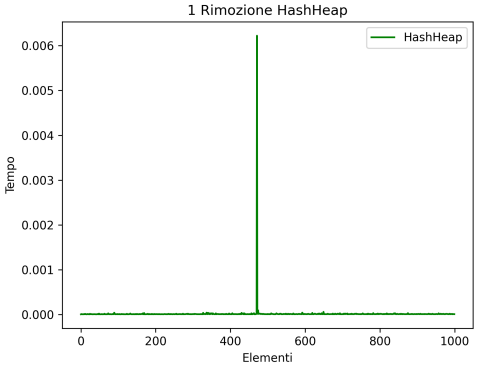
(c)

Figura 9: Grafici basati sui tempi di esecuzione per la ricerca di un elemento con n elementi precedentemente inseriti. (a) Ricerca per la lista concatenata. (b) Ricerca per l'albero binario di ricerca. (c) Ricerca per l'hasheap



(a)

(b)

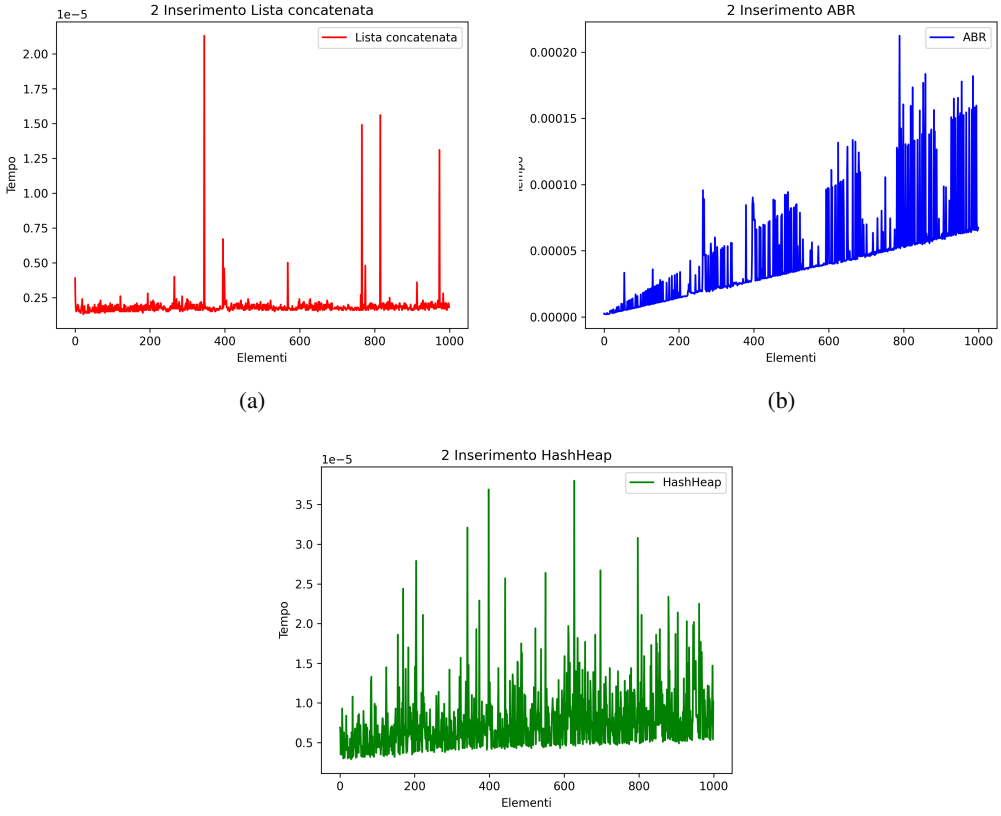


(c)

Figura 10: Grafici basati sui tempi di esecuzione per la rimozione di un elemento con n elementi precedentemente inseriti. (a) Rimozione per la lista concatenata. (b) Rimozione per l'albero binario di ricerca. (c) Rimozione per l'hasheap

4.2 Test 2

Nel test i dati sono stati raccolti su un set di dati chiave-valore con ordinamento ordinato per le chiavi e casuale per i valori.



(a) (b)

(c)

Figura 11: Grafici basati sui tempi di esecuzione per l'inserimento di un elemento con n elementi precedentemente inseriti. (a) Inserimento per la lista concatenata. (b) Inserimento per l'albero binario di ricerca. (c) Inserimento per l'hasheap

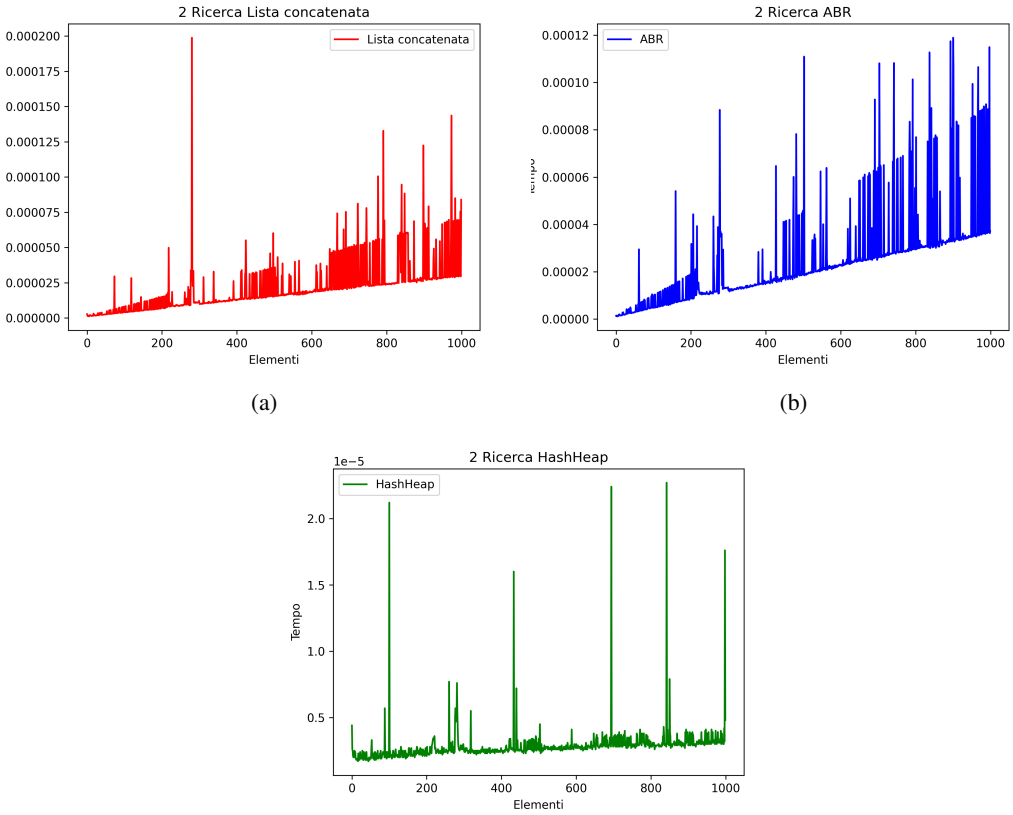
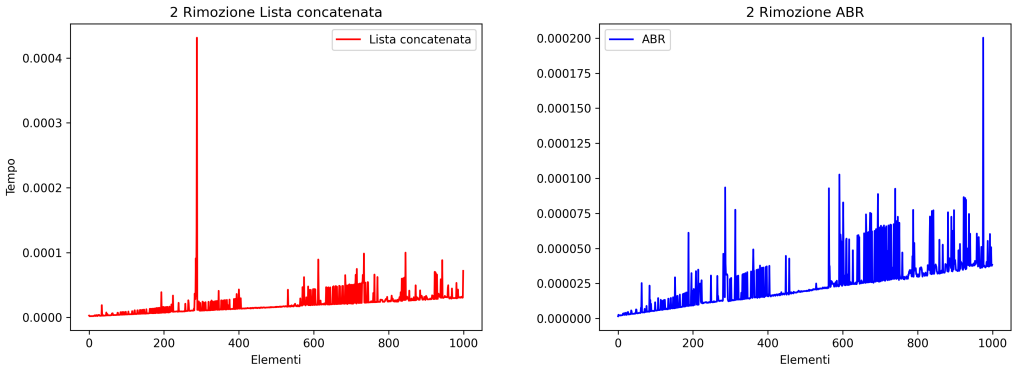
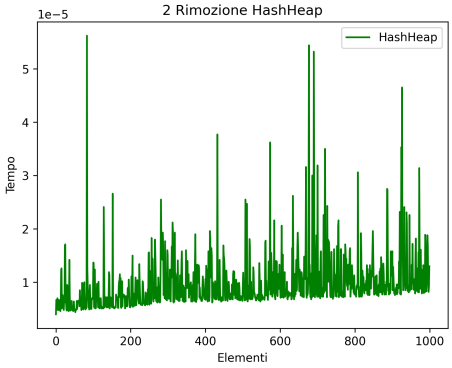


Figura 12: Grafici basati sui tempi di esecuzione per la ricerca di un elemento con n elementi precedentemente inseriti. (a) Ricerca per la lista concatenata. (b) Ricerca per l'albero binario di ricerca. (c) Ricerca per l'hasheap



(a)

(b)

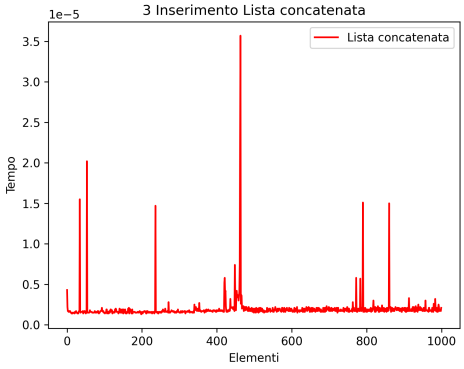


(c)

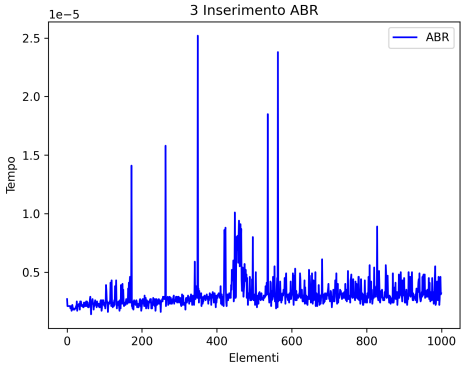
Figura 13: Grafici basati sui tempi di esecuzione per la rimozione di un elemento con n elementi precedentemente inseriti. (a) Rimozione per la lista concatenata. (b) Rimozione per l'albero binario di ricerca. (c) Rimozione per l'hasheap

4.3 Test 3

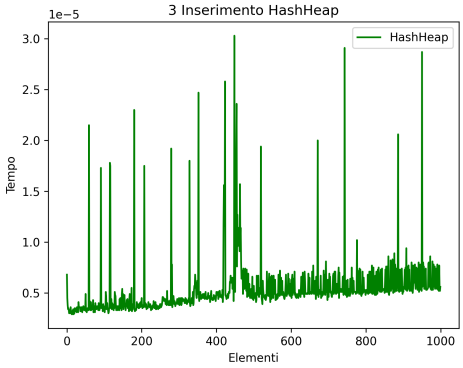
Nel test i dati sono stati raccolti su un set di dati chiave-valore con ordinamento casuale per le chiavi e ordinato per i valori.



(a)

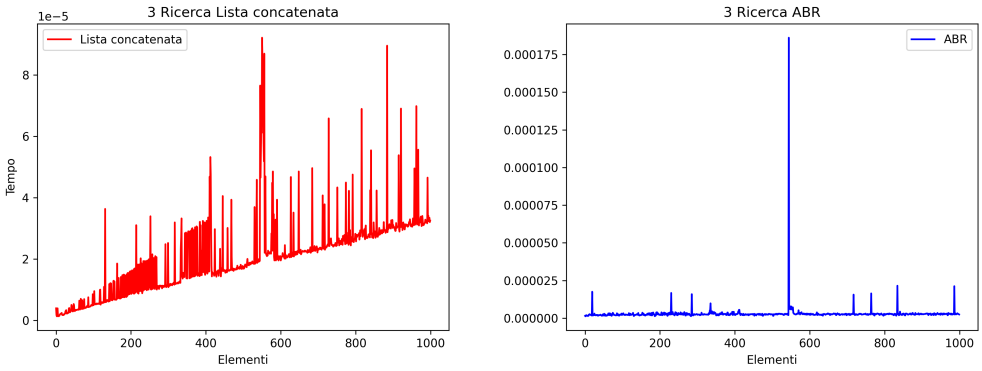


(b)



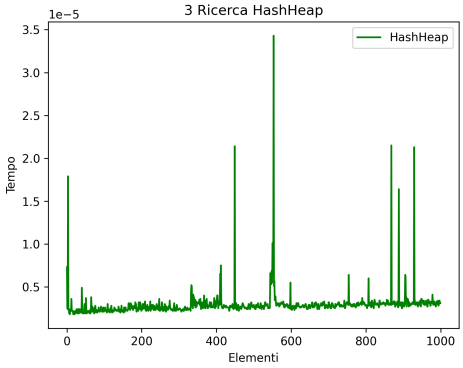
(c)

Figura 14: Grafici basati sui tempi di esecuzione per l'inserimento di un elemento con n elementi precedentemente inseriti. (a) Inserimento per la lista concatenata. (b) Inserimento per l'albero binario di ricerca. (c) Inserimento per l'hasheap



(a)

(b)



(c)

Figura 15: Grafici basati sui tempi di esecuzione per la ricerca di un elemento con n elementi precedentemente inseriti. (a) Ricerca per la lista concatenata. (b) Ricerca per l'albero binario di ricerca. (c) Ricerca per l'hasheap

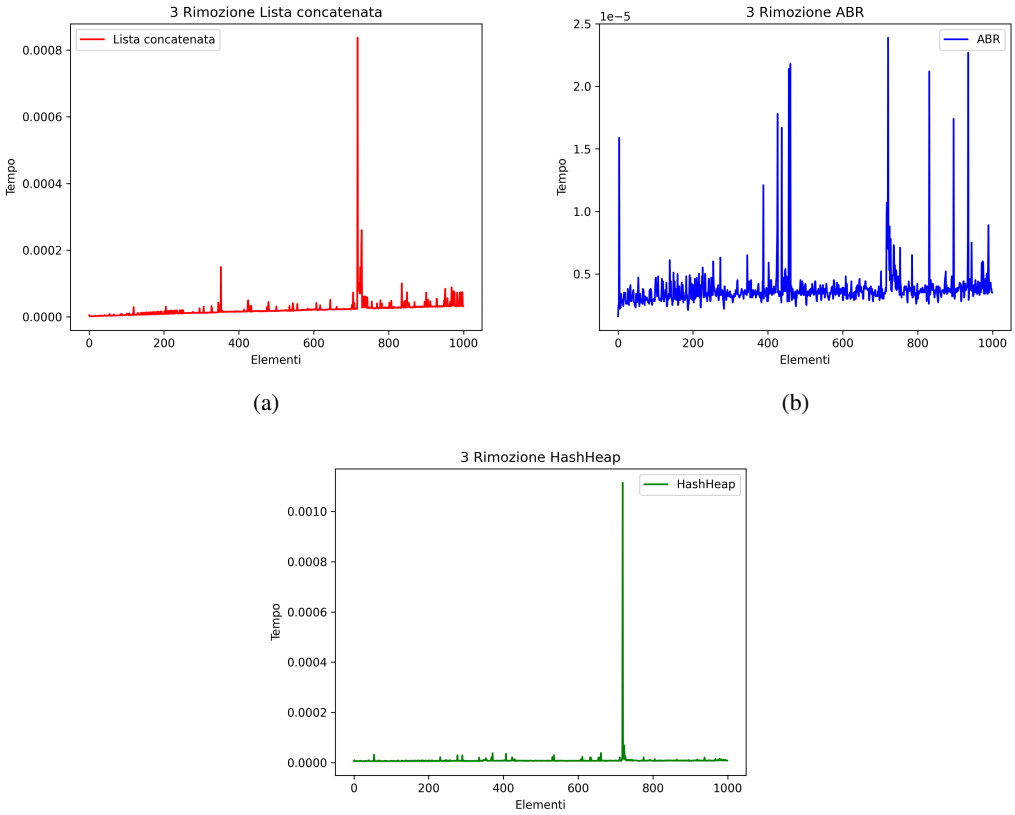


Figura 16: Grafici basati sui tempi di esecuzione per la rimozione di un elemento con n elementi precedentemente inseriti. (a) Rimozione per la lista concatenata. (b) Rimozione per l'albero binario di ricerca. (c) Rimozione per l'hasheap

Mediamente i tempi di esecuzione del primo test sono:

- Lista concatenata:
  - Inserimento: 0.00000188
  - Ricerca: 0.00002014
  - Rimozione: 0.00002078
- ABR:
  - Inserimento: 0.00000279
  - Ricerca: 0.00000250
  - Rimozione: 0.00000355
- HashHeap:
  - Inserimento: 0.00000766
  - Ricerca: 0.00000298
  - Rimozione: 0.00001628

Mediamente i tempi di esecuzione del secondo test sono:

- Lista concatenata:
  - Inserimento: 0.00000186
  - Ricerca: 0.00002163
  - Rimozione: 0.00002075
- ABR:
  - Inserimento: 0.00004395
  - Ricerca: 0.00002438
  - Rimozione: 0.00002509
- HashHeap:
  - Inserimento: 0.00000757
  - Ricerca: 0.00000286
  - Rimozione: 0.00000985

Mediamente i tempi di esecuzione del terzo test sono:

- Lista concatenata:
  - Inserimento: 0.00000195
  - Ricerca: 0.00002030
  - Rimozione: 0.00002247
- ABR:
  - Inserimento: 0.00000310
  - Ricerca: 0.00000289
  - Rimozione: 0.00000379
- HashHeap:
  - Inserimento: 0.00000536
  - Ricerca: 0.00000293
  - Rimozione: 0.00000896

4.4 Conclusioni

Nella realtà i dati posso differire dalla stima teorica ma notiamo una somiglianza. La lista concatenata a seconda dei valori dei elementi non varia il suo tempo di esecuzione a differenza dell'albero binario di ricerca che nel secondo test con chiavi ordinate ha avuto un tempo nella ricerca e rimozione simile a quello della lista peggiorando i suoi tempi. L'hashheap ha avuto dei miglioramenti nel terzo test con le chiavi casuali e i valori ordinati riducendo il tempo di esecuzione al tempo di esecuzione delle funzionalità della tabella hash. A seconda delle esigenze una certa implementazione conviene ad altre ma bisogna stare attenti a vantaggi e svantaggi. La lista concatenata è resistente al ordine delle chiavi e valori, non ne influiscono i tempi di esecuzione. Pur avendo una semplice implementazione se il numero di elementi trattato è grande ci aspettiamo dei tempi sempre più maggiori. L'albero binario di ricerca ci offre mediamente dei tempi stabili a meno di non avere un set di input ordinati ma è evitabile inserendo i elementi in ordine casuale che può portare anche ad avere mediamente un altezza uguale a lgn. L'hashheap tra le strutture dati precedenti è la più complessa da implementare e occupa molta più memoria ma con una buona implementazione della funzione di hashing possiamo ridurre la memoria occupata e le collisioni. Se i nostri elementi subiscono poche rimozioni o inserimenti l'hashheap è la struttura dati più veloce in termini di tempo per la ricerca di un elemento.