ΑΝΑΦΟΡΑ 2ου PROJECT ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΑΛΓΟΡΙΘΜΟΙ

Αλέξια Σούλα: 2021030089
Ιωάννης Μπουρίτης: 2021030173

Θεωρητική ανάλυση για την πολυπλοκότητα χειρότερης περίπτωσης (worst case) των αλγορίθμων εισαγωγής και αναζήτησης για κάθε περίπτωση

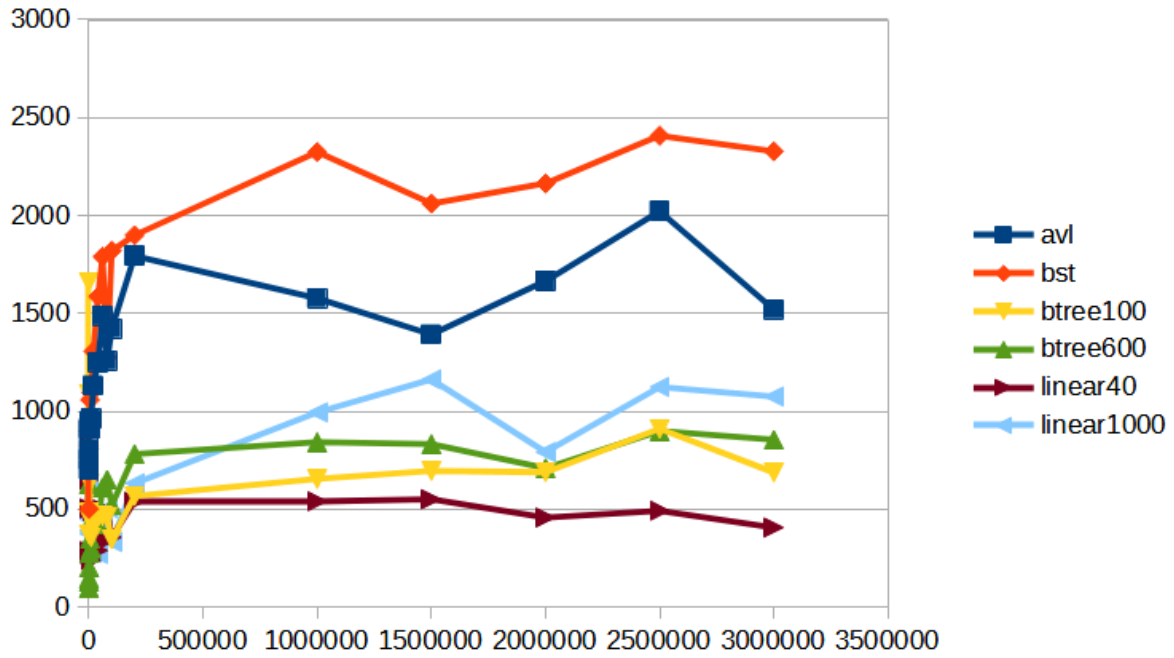
	BST	AVL	BTree100	Btree600	Linear40	Linear1000
insert()	O(n)	O(log n)	O(log n)	O(log n)	O(1)	O(1)
search()	O(n)	O(log n)	O(log n)	O(log n)	O(1)	O(1)
Search levels	O(n)	O(log n)	O(log n)	O(log n)	O(1)	O(1)

Σύντομη τεκμηρίωση

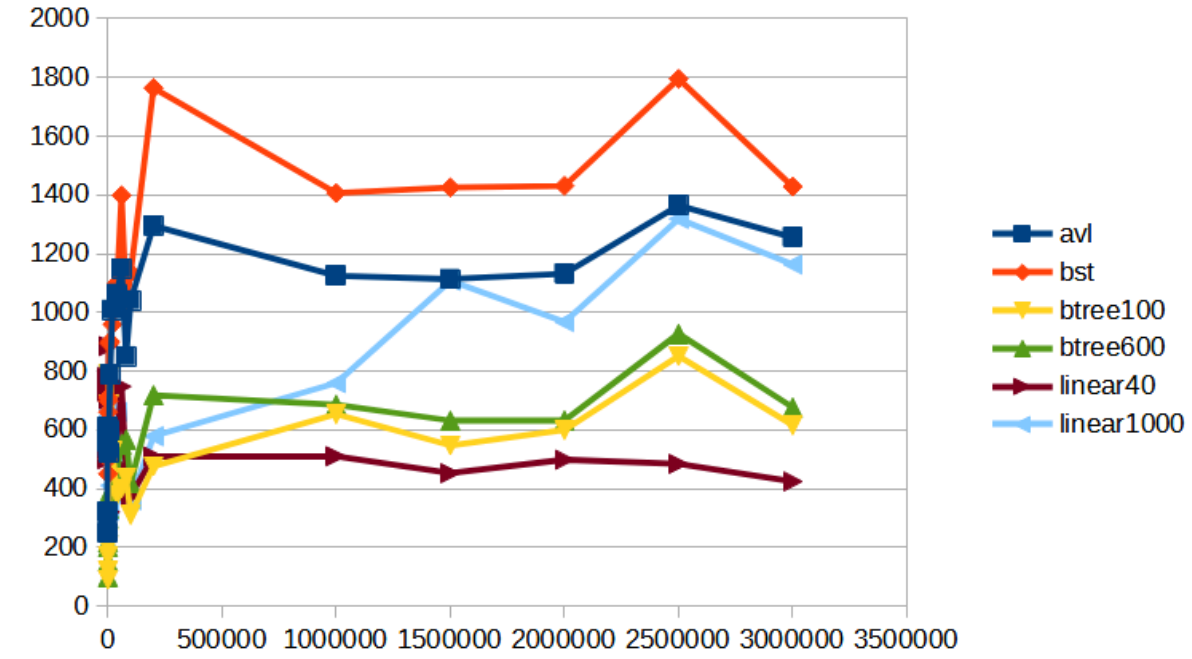
- BSTree
  - insert(): Αν το δέντρο είναι μη ισορροπημένο (π.χ. στη χειρότερη περίπτωση μοιάζει με συνδεδεμένη λίστα), η εισαγωγή μπορεί να απαιτεί O(n) συγκρίσεις.
  - search(): Σε μη ισορροπημένα BST, η αναζήτηση μπορεί να απαιτεί O(n) συγκρίσεις αν το δέντρο έχει το μέγιστο ύψος
  - search levels: Η αναζήτηση μπορεί να διαρκέσει O(n) επίπεδα αν το δέντρο είναι πολύ μη ισορροπημένο.
- AVLTree
  - insert(): Η δομή είναι ισορροπημένη, έτσι κάθε εισαγωγή απαιτεί το πολύ O(log n) περιστροφές για να διατηρήσει την ισορροπία.
  - search(): Η ισορροπημένη φύση του AVL Tree εξασφαλίζει ότι το ύψος του δέντρου είναι O(log n), οπότε η αναζήτηση ενός κλειδιού διαρκεί O(log n) βήματα.
  - search levels: Το μέγιστο ύψος του δέντρου είναι O(log n), που σημαίνει ότι η αναζήτηση διαρκεί το πολύ O(log n) επίπεδα.
- BTree(100, 600)
  - insert(): Τα B-Trees είναι σχεδιασμένα να παραμένουν ισορροπημένα, έτσι η εισαγωγή απαιτεί το πολύ log n συγκρίσεις και ανακατανομές.
  - search(): Η αναζήτηση απαιτεί log n συγκρίσεις καθώς το ύψος του B-Tree είναι O(log n) λόγω της ισορροπίας του δέντρου.
  - search levels: Η αναζήτηση διαρκεί log n επίπεδα καθώς το δέντρο παραμένει ισορροπημένο

- Linear(0 buckets, 1000 buckets)
    - insert(): Οι εισαγωγές είναι συνήθως O(1) λόγω της άμεσης πρόσβασης μέσω της hash συνάρτησης, αν και περιστασιακά μπορεί να απαιτείται ανακατανομή (rehashing).
    - search(): Οι αναζητήσεις είναι συνήθως O(1) λόγω της άμεσης πρόσβασης μέσω της hash συνάρτησης, αν και σε περιπτώσεις συγκρούσεων (collisions) μπορεί να χρειαστεί επιπλέον χρόνος.
    - search levels(): Στη μέση περίπτωση, η αναζήτηση διαρκεί ένα επίπεδο (άμεση πρόσβαση). Σε περιπτώσεις συγκρούσεων μπορεί να χρειαστεί πρόσβαση σε επιπλέον επίπεδα, αλλά αυτό είναι σπάνιο.

Δημιουργήστε ένα γράφημα με τους χρόνους εισαγωγής για όλες τις δομές (4 δομές, 6 καμπύλες) και ένα αντίστοιχο ξεχωριστό γράφημα για τους χρόνους αναζήτησης, όπου στο X άξονα έχετε τις διάφορες τιμές του N.



Γράφημα 1: Χρόνοι εισαγωγής



Γράφημα 2: Χρόνοι αναζήτησης

Τί μορφή έχουν οι καμπύλες

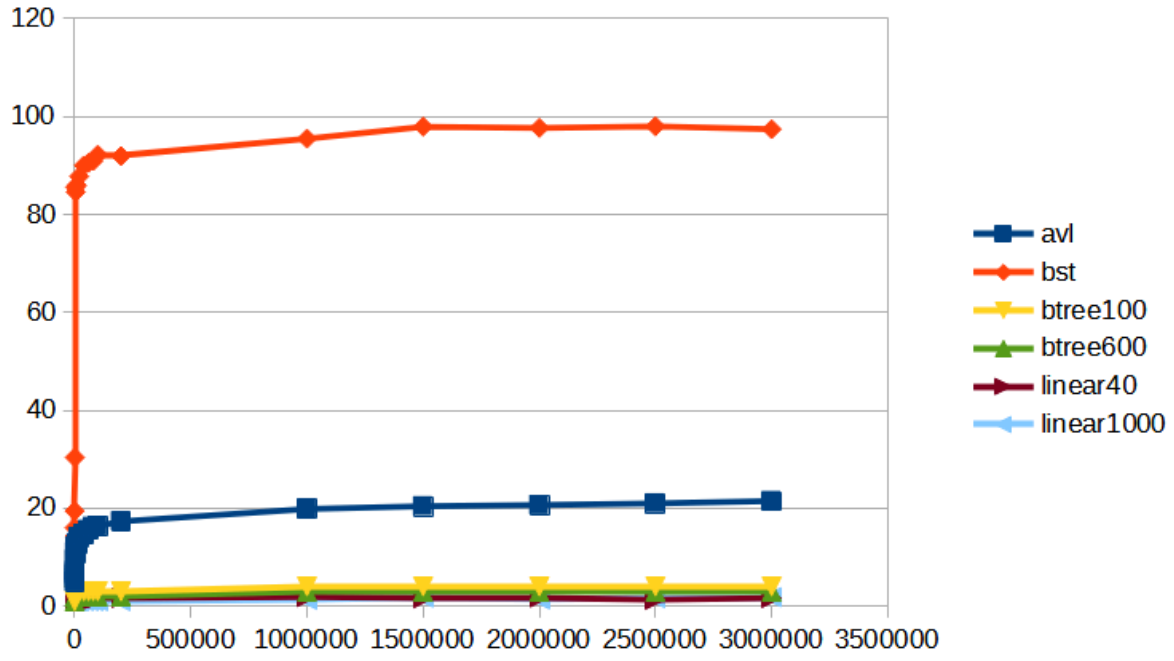
INSERT():

- AVL: Με βάση το γράφημα, παρατηρούμε ότι η καμπύλη είναι λογαριθμική, το οποίο συμφωνεί με τη θεωρητική ανάλυση που έγινε στο (1). Ακραίες τιμές για τα μικρά N εξαιρέθηκαν έτσι ώστε να φαίνεται καθαρότερα η μορφή της καμπύλης. Επίσης, διακρίνουμε ορισμένα καρφιά (spikes) στις γραφικές, πράγμα που μπορεί να οφείλεται στο μοντέλο του υπολογιστή μας ή στο συγκεκριμένο παράδειγμα.
- BST: Παρατηρούμε ότι η καμπύλη είναι επίσης λογαριθμική, πράγμα που συμφωνεί με τη θεωρητική μας μέτρηση, καθώς το worst case του BST έχει O(n) πολυπλοκότητα αν το δέντρο είναι τελείως μη ισορροπημένο. Αυτό δεν συμβαίνει στην περίπτωση μας, και γι' αυτό η καμπύλη είναι λογαριθμική. Παρόλα αυτά, παρατηρούμε ότι η καμπύλη έχει μεγαλύτερο ύψος από τις υπόλοιπες, οπότε αποδεικνύεται ότι το BST είναι πιο αργή υλοποίηση από τις υπόλοιπες.
- BTree(100, 600): Οι καμπύλες βγήκαν επίσης λογαριθμικές, όπως και η πολυπλοκότητά τους στη θεωρητική μας ανάλυση, αλλά επίσης έχουν μικρότερο ύψος από την AVL και BST. Αυτό αποδεικνύει ότι τα B-Trees είναι πιο γρήγορα από τις προηγούμενες υλοποιήσεις. Παρατηρουμε επισης οτι το Btree για t = 100 ειναι σαφως λιγο πιο γρηγορο απο οτι για t = 400.
- Linear Hashing(40, 1000): Η καμπύλη δείχνει σταθερό χρόνο εισαγωγής για μικρό αριθμό στοιχείων, αλλά μερικές αυξήσεις στον χρόνο εισαγωγής καθώς πραγματοποιούνται επεκτάσεις και συγχωνεύσεις. Η καμπύλη έχει σταθερά διαστήματα με μικρά "σκαλοπάτια" όταν πραγματοποιούνται επεκτάσεις. Επισης παρατηρουμε οτι για 1000 Buckets ειναι προφανως πιο αργη απο 40 Buckets.

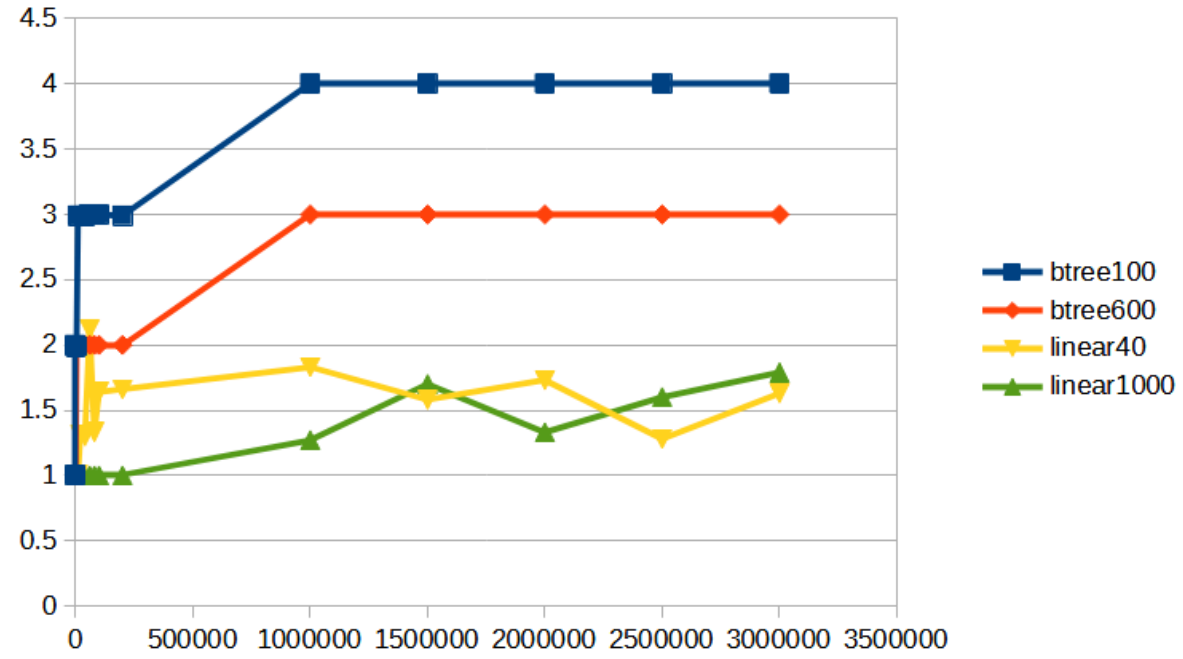
SEARCH():

- AVL search(): Η καμπύλη είναι σχεδόν λογαριθμική. Η αύξηση στον χρόνο αναζήτησης είναι σταθερή και λογαριθμική με τον αριθμό των στοιχείων.
- BST search(): Η καμπύλη είναι λογαριθμική, αλλά έχει μεγαλύτερες διακυμάνσεις, δείχνοντας τις περιπτώσεις όπου το δέντρο δεν είναι ισορροπημένο. Επίσης έχει πάλι το μεγαλύτερο ύψος, που δείχνει ότι δεν είναι τόσο αποδοτικό σε σχέση με τις υπόλοιπες υλοποιήσεις.
- Btree(100,600) search(): Οι καμπύλες για τα B-trees είναι σχεδόν λογαρυθμικες αλλά με χαμηλότερη κλίση σε σχέση με τα AVL, και μικρότερη κλίση για μεγαλύτερο t.
- Linear Hashing(40,1000) search():Οι καμπύλες για τα B-trees είναι σχεδόν γραμμικές αλλά με χαμηλότερη κλίση σε σχέση με τα AVL, και μικρότερη κλίση για μεγαλύτερο t.

Δημιουργήστε ένα γράφημα με τα επίπεδα ανά αναζήτηση για όλες τις δομές (4 δομές, 6 καμπύλες), όπου στο X άξονα έχετε τις διάφορες τιμές του N.



Γράφημα 3: Επίπεδα αναζήτησης για όλες τις υλοποιήσεις



Γράφημα 4: Επίπεδα αναζήτησης για όλες τις υλοποιήσεις εκτός των AVL και BST (για πιο εμφανή διάκριση σε χαμηλές τιμές)

Τι μορφή έχουν οι καμπύλες;

- Η καμπύλη του AVLTree ειναι λογαριθμικη το οποίο συμβαδίζει με την παραπάνω θεωρητική μας προσέγγιση.
- Η καμπύλη του BSTree στην συγκεκριμένη περίπτωση είναι λογαριθμική. Παρόλο που στη θεωρητική προσέγγιση αναφέραμε ότι η πολυπλοκότητα του είναι O(N), αυτό ισχύει στη χειρότερη περίπτωση που το δέντρο είναι πολύ μη ισορροπημένο. Στο συγκεκριμένο, το Binary Search Tree μας έχει γραφική λογαριθμικού χαρακτήρα. Σε ένα ισορροπημένο BST, το ύψος είναι περίπου $\log_2(n)$.
- Η καμπύλες των BTrees όπως είναι αναμενόμενο για τα επίπεδα αναζήτησης παρουσιάζουν λογαριθμική μορφή (O(log n)) καθώς διατηρούνται ισορροπημένα με την προσθήκη ή αφαίρεση κόμβων δηλαδή το ύψος του δέντρου παραμένει λογαριθμικό ως προς τον αριθμό των στοιχείων που περιέχει.
- Οι γραφικές των Linear έχουν πολυπλοκότητα O(1) και ως αποτέλεσμα συμβαδίζουν με τη θεωρητική μας προσέγγιση. Στο συγκεκριμένο διάγραμμα οι δύο γραφικές επικαλύπτουν η μία την άλλη, καθώς αν ελέγξουμε τις τιμές για κάθε N, όλες είναι ανάμεσα στο 1 και το 2.

Τι διαφορά βλέπετε (αν βλέπετε), μεταξύ του AVL δέντρου και του BST δέντρου (είτε σε χρόνο είτε σε επίπεδα); Αιτιολογήστε τη διαφορά (αν υπάρχει).

Η βασική διαφορά μεταξύ του AVL δέντρου και του BST έγκειται στη διατήρηση της ισορροπίας. Το AVL δέντρο διατηρεί τον χρόνο αναζήτησης και εισαγωγής σταθερά O(log n) και το ύψος του παραμένει μικρό, ανεξάρτητα από τη σειρά εισαγωγής των στοιχείων. Το

BST, από την άλλη, μπορεί να γίνει ανισόρροπο, οδηγώντας σε σημαντικά χειρότερες επιδόσεις σε χρόνο και ύψος, ειδικά σε περιπτώσεις χειρότερης περίπτωσης (worst-case). Αυτές οι διαφορές αιτιολογούν γιατί το AVL δέντρο είναι προτιμότερο για σενάρια όπου η ταχύτητα και η σταθερότητα είναι κρίσιμες. Συγκεκριμένα, με βάση τις μετρήσεις παρατηρούμε ότι το average search time του AVL Tree κυμαίνεται γύρω στο log n και το average levels αυξάνεται λογαριθμικά με τον χρόνο, ενώ το average search time του BSTree ποικίλλει ανάλογα με το δέντρο (μπορεί να είναι μεγαλύτερο του log n σε ανισόρροπα δέντρα) και το average levels μπορεί να είναι πολύ υψηλό, φτάνοντας και στο O(n) στη χειρότερη περίπτωση.

Σε ποια περίπτωση (αν υπάρχει) θα προτιμούσατε τη χρήση του BST δέντρου έναντι των άλλων δομών (εισαγωγή και αναζήτηση στη RAM); Αιτιολογείστε την απάντησή σας βάσει των μετρήσεων που πήρατε.

Παρά την απλότητα της υλοποίησης, το BST δεν προσφέρει την καλύτερη απόδοση στις περισσότερες περιπτώσεις σύμφωνα με τις μετρήσεις. Προτείνεται η χρήση του κυρίως σε περιπτώσεις όπου: Η απλότητα και η ευκολία υλοποίησης είναι πιο σημαντικές από την απόδοση.Η εφαρμογή χειρίζεται μικρό όγκο δεδομένων. Η απόδοση της εισαγωγής και της αναζήτησης δεν είναι κρίσιμη για την εφαρμογή.Για εφαρμογές που απαιτούν υψηλή απόδοση, οι ισορροπημένες δομές όπως το AVL δέντρο ή οι δομές κατακερματισμού είναι γενικά προτιμότερες.

Αν αντί στη RAM, χρειάζεται να υλοποιήσετε τις ίδιες δομές σε σκληρό δίσκο, ποια δομή θα προτιμήσετε για αναζήτηση τυχαίου κλειδιού;

Αν υλοποιούσαμε τις ίδιες δομές σε σκληρό δίσκο, θα προτιμούσαμε τη δομή Linear Hashing, καθώς οι μετρήσεις δείχνουν ότι η μέση αναζήτηση στον δίσκο διακειμενεται στις 1 με 1.69 προσβάσεις επιπέδων, κάτι που είναι πολύ κοντά στο O(1). Αυτό σημαίνει ότι απαιτούνται πολύ λίγες προσβάσεις στον δίσκο, μειώνοντας τον χρόνο αναζήτησης. Έχει επίσης τον χαμηλότερο μέσο χρόνο εισαγωγής. Αυτό μειώνει την ανάγκη για συχνές εγγραφές στον δίσκο, κάτι που βελτιώνει την απόδοση και την ταχύτητα της δομής. Επίσης, έχει σταθερό και χαμηλό αριθμό επιπέδων. Αυτό σημαίνει ότι οι αναζητήσεις και οι εισαγωγές παραμένουν αποδοτικές, ακόμη και όταν ο όγκος των δεδομένων αυξάνεται.
Σε περιβάλλον σκληρού δίσκου, η απόδοση καθορίζεται κυρίως από τον αριθμό των προσβάσεων στον δίσκο. Το Linear Hashing υπερέχει σε αυτό το σημείο, καθώς προσφέρει σχεδόν σταθερή πολυπλοκότητα αναζήτησης και εισαγωγής, με ελάχιστο αριθμό επιπέδων.

ΔΕΝ ΕΠΙΛΕΓΟΥΜΕ:

AVLTree: Παρόλο που είναι ισορροπημένο και έχει καλή πολυπλοκότητα O(log n), δεν έχει όσο καλή αποδοτικότητα έχει το Linear Hashing στον αριθμό των προσβάσεων στον δίσκο. Ο μέσος αριθμός επιπέδων είναι μεγαλύτερος και η αναζήτηση είναι πιο αργή από το Linear Hashing.

BSTree: Είναι η λιγότερο προτιμητέα δομή για υλοποίηση σε σκληρό δίσκο λόγω του υψηλού αριθμού επιπέδων και της μη ισορροπημένης δομής του, η οποία αυξάνει τον αριθμό προσβάσεων στον δίσκο και μειώνει την απόδοση.

Αν χρειαζόμασταν ερωτήσεις εύρους τιμών (όλα τα κλειδιά μεταξύ 2 τιμών), ποια (ή ποιές) δομή(ές) θα προτιμούσατε για τη RAM, και ποια (ή ποιες) για το δίσκο;

Δίσκος:

Για την αναζήτηση τυχαίου κλειδιού σε σκληρό δίσκο, τα B-Trees είναι συνήθως προτιμότερα λόγω της αποδοτικότητας τους στις προσπελάσεις δίσκου και της ικανότητάς τους να διατηρούν τον χρόνο αναζήτησης σταθερά O(log n) με ελάχιστες αναγνώσεις σελίδων. Επισης χρησιμοποιειται και το Linear Hashing λογω την O(1) πολυπλοκοτητας του, αλλα στην πράξη, οι προσπελάσεις στο δίσκο εξαρτώνται από το πώς είναι κατανεμημένα τα κλειδιά και από το πόσο γεμάτος είναι ο πίνακας κατακερματισμού. Σε περιπτώσεις σύγκρουσης, μπορεί να χρειαστούν πολλαπλές προσπελάσεις σελίδων για να βρεθεί το κλειδί.

RAM:

Τα AVL Trees είναι οι καλύτερες επιλογές για ερωτήσεις εύρους τιμών, λόγω της ισορροπημένης τους δομής και της καλής απόδοσης στην αναζήτηση εύρους. Χρησιμοποιούν O(log N) χρόνο για εισαγωγή, διαγραφή και αναζήτηση. Για range queries, μπορούμε να κάνουμε αναζήτηση O(log N) για να βρούμε το κατώτατο όριο και στη συνέχεια να διασχίσουμε το δέντρο σε χρόνο O(K), όπου K είναι ο αριθμός των στοιχείων στο εύρος. Ένα ισορροπημένο BSTree έχει παρόμοια απόδοση, αλλά μπορεί να μην είναι τόσο αποδοτικό όσο το AVL Tree αν δεν είναι αυστηρά ισορροπημένο.