Παραδοτέο 1

Μάθημα: Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Μέλη Ομάδας:

Ραμαντάν Κονόμι - ΑΜ: 1115201800281

Θεμιστοκλής Παπαθεοφάνους - ΑΜ: 1115202100227

Μάριος Γιαννόπουλος - ΑΜ: 1115202000032

Πίνακας Περιεχομένων

Ανάλυση της Κλάσης **HopscotchTable**

HopscotchTable

Δομές Δεδομένων

Συναρτήσεις Κατακερματισμού

Ανάλυση της Κλάσης **RobinHoodTable**

RobinHoodTable

Δομές Δεδομένων

Συναρτήσεις Κατακερματισμού

Ανάλυση της Κλάσης **CuckooTable**

CuckooTable

Η κλάση `CuckooTable<Key>` υλοποιεί τον αλγόριθμο Κατακερματισμού Cuckoo (Cuckoo Hashing), μια προηγμένη τεχνική κατακερματισμού που εγγυάται σταθερό χρόνο αναζήτησης $O(1)$ στην **χειρότερη** περίπτωση. Η υλοποίηση είναι βελτιστοποιημένη για αποδοτικότητα μνήμης και κρυφής μνήμης (Cache Efficiency), χρησιμοποιώντας έναν **μηχανισμό κοινόχρηστης αποθήκευσης** για πολλαπλές τιμές και **inline value optimization** για μοναδικές τιμές. Σε αντίθεση με τις μεθόδους ανοικτής διευθυνσιοδότησης που βασίζονται σε chaining ή γραμμική διερεύνηση (linear probing), το Cuckoo Hashing χρησιμοποιεί δύο πίνακες και δύο συναρτήσεις κατακερματισμού για να εξασφαλίσει ότι κάθε στοιχείο βρίσκεται ακριβώς σε μία από τις δύο πιθανές θέσεις του.

Δομές Δεδομένων

Ο πίνακας Cuckoo αποτελείται από δύο πίνακες, `table1` και `table2`, ίσου `capacity`. Επιπλέον, χρησιμοποιεί κοινόχρηστη αποθήκευση για τις τιμές:

`value_store`: Αποθηκεύει τις τιμές (indices/items) που σχετίζονται με τα κλειδιά. `segments`: Αποθηκεύει τους δείκτες για την πρόσβαση σε πολλαπλές τιμές εντός του `value_store`.

CuckooBucket

Κάθε θέση στους πίνακες περιέχει ένα `CuckooBucket`, το οποίο αντικαθιστά την ανάγκη για `std::optional` μέσω του πεδίου `occupied`. Η δομή αυτή υποστηρίζει την inline βελτιστοποίηση:

```
template<typename Key>
struct CuckooBucket {
    Key key;
    uint32_t first_segment; // Δείκτης για την αλυσίδα στοιχείων στο value_store (αν
count > 1)
    uint32_t last_segment;  // Αποθηκεύει την τιμή (item) αν count = 1 (inline
optimization)
    uint16_t count;         // Πλήθος τιμών
    bool occupied;          // Αντικαθιστά το std::optional
};
```

Πίνακες και Χωρητικότητα

Ο πίνακας διαχειρίζεται δύο εσωτερικούς πίνακες table1 και table2, καθένας με χωρητικότητα capacity.

```
std::vector<CuckooBucket<Key>> table1; // Χρήση CuckooBucket, όχι std::optional
std::vector<CuckooBucket<Key>> table2;
size_t capacity;
```

Συναρτήσεις Κατακερματισμού

Χρησιμοποιούνται δύο ανεξάρτητες συναρτήσεις κατακερματισμού, h1 και h2, για την αντιστοίχιση ενός κλειδιού σε μια θέση στους table1 και table2 αντίστοιχα.

Συνάρτηση h1

Η h1 είναι η τυπική συνάρτηση κατακερματισμού, χρησιμοποιώντας την std::hash.

```
size_t h1(const Key& key) const {
    return key_hasher(key) % capacity;
}
```

Συνάρτηση h2

Η h2 προκύπτει από μια απλή κυκλική μετατόπιση (rotation) του αρχικού hash value, εξασφαλίζοντας μια δεύτερη, ανεξάρτητη διεύθυνση.

```
size_t h2(const Key& key) const {
    size_t h = key_hasher(key);
```

```
    // Κυκλική μετατόπιση αριστερά (e.g., κατά 1 bit)
    return ((h << 1) | (h >> (sizeof(size_t) * 8 - 1))) % capacity;
}
```

Μηχανισμός Εισαγωγής (The Kick Process)

Η εισαγωγή ενός νέου στοιχείου γίνεται μέσω της διαδικασίας «εκτόπισης» (kicking) που υλοποιείται στη μέθοδο insert_internal.

Ανάλυση της Κλάσης **CuckooTable**

CuckooTable

Η κλάση `CuckooTable<Key>` υλοποιεί τον αλγόριθμο Κατακερματισμού Cuckoo (Cuckoo Hashing), μια προηγμένη τεχνική κατακερματισμού που εγγυάται σταθερό χρόνο αναζήτησης $O(1)$ στην **χειρότερη** περίπτωση. Η υλοποίηση είναι βελτιστοποιημένη για αποδοτικότητα μνήμης και κρυφής μνήμης (Cache Efficiency), χρησιμοποιώντας έναν **μηχανισμό κοινόχρηστης αποθήκευσης** για πολλαπλές τιμές και **inline value optimization** για μοναδικές τιμές. Σε αντίθεση με τις μεθόδους ανοικτής διευθυνσιοδότησης που βασίζονται σε chaining ή γραμμική διερεύνηση (linear probing), το Cuckoo Hashing χρησιμοποιεί δύο πίνακες και δύο συναρτήσεις κατακερματισμού για να εξασφαλίσει ότι κάθε στοιχείο βρίσκεται ακριβώς σε μία από τις δύο πιθανές θέσεις του.

Δομές Δεδομένων

Ο πίνακας Cuckoo αποτελείται από δύο πίνακες, `table1` και `table2`, ίσου `capacity`. Επιπλέον, χρησιμοποιεί κοινόχρηστη αποθήκευση για τις τιμές:

`value_store`: Αποθηκεύει τις τιμές (indices/items) που σχετίζονται με τα κλειδιά. **`segments`**: Αποθηκεύει τους δείκτες για την πρόσβαση σε πολλαπλές τιμές εντός του `value_store`.

CuckooBucket

Κάθε θέση στους πίνακες περιέχει ένα `CuckooBucket`, το οποίο αντικαθιστά την ανάγκη για `std::optional` μέσω του πεδίου `occupied`. Η δομή αυτή υποστηρίζει την inline βελτιστοποίηση:

```
template<typename Key>
struct CuckooBucket {
    Key key;
    uint32_t first_segment; // Δείκτης για την αλυσίδα στοιχείων στο value_store (αν
count > 1)
    uint32_t last_segment;  // Αποθηκεύει την τιμή (item) αν count = 1 (inline
optimization)
    uint16_t count;         // Πλήθος τιμών
    bool occupied;          // Αντικαθιστά το std::optional
};
```

Πίνακες και Χωρητικότητα

Ο πίνακας διαχειρίζεται δύο εσωτερικούς πίνακες table1 και table2, καθένας με χωρητικότητα capacity.

```
std::vector<CuckooBucket<Key>> table1; // Χρήση CuckooBucket, όχι std::optional
std::vector<CuckooBucket<Key>> table2;
size_t capacity;
```

Συναρτήσεις Κατακερματισμού

Χρησιμοποιούνται δύο ανεξάρτητες συναρτήσεις κατακερματισμού, h1 και h2, για την αντιστοίχιση ενός κλειδιού σε μια θέση στους table1 και table2 αντίστοιχα.

Συνάρτηση h1

Η h1 είναι η τυπική συνάρτηση κατακερματισμού, χρησιμοποιώντας την std::hash.

```
size_t h1(const Key& key) const {
    return key_hasher(key) % capacity;
}
```

Συνάρτηση h2

Η h2 προκύπτει από μια απλή κυκλική μετατόπιση (rotation) του αρχικού hash value, εξασφαλίζοντας μια δεύτερη, ανεξάρτητη διεύθυνση.

```
size_t h2(const Key& key) const {
    size_t h = key_hasher(key);
    // Κυκλική μετατόπιση αριστερά (e.g., κατά 1 bit)
    return ((h << 1) | (h >> (sizeof(size_t) * 8 - 1))) % capacity;
}
```

Μηχανισμός Εισαγωγής (The Kick Process)

Η εισαγωγή ενός νέου στοιχείου γίνεται μέσω της διαδικασίας «εκτόπισης» (kicking) που υλοποιείται στη μέθοδο insert_internal.

Βήματα Εισαγωγής

Η διαδικασία εισαγωγής ακολουθεί τους εξής κανόνες:

- **Έλεγχος Υπάρχοντος:** Πριν την εκτόπιση, ελέγχεται αν το κλειδί υπάρχει ήδη στις δύο πιθανές θέσεις του. Αν ναι, η νέα τιμή **απλώς προστίθεται** στο υπάρχον CuckooBucket (μέσω της insert_duplicate).
- **Ανταλλαγή/Εκτόπιση (Kick):** Εάν η θέση προορισμού είναι κατειλημμένη, το νέο στοιχείο εισάγεται και το υπάρχον στοιχείο εκτοπίζεται. Η ανταλλαγή πραγματοποιείται με

την std::swap(bucket, table[idx]), όπου η μεταβλητή bucket περιέχει πάντα το στοιχείο που είναι «στον αέρα».
- **Μετάβαση:** Το εκτοπισμένο στοιχείο αναζητά την εναλλακτική του θέση στον άλλο πίνακα (από h1 σε h2 και αντίστροφα).

Η διαδικασία αυτή συνεχίζεται έως ότου βρεθεί μια κενή θέση.

Όριο Εκτοπίσεων (MAX_KICKS)

Για να αποφευχθεί ο ατέρμονος βρόχος (cycle) που μπορεί να προκληθεί από την κυκλική εκτόπιση στοιχείων, η υλοποίηση θέτει ένα όριο MAX_KICKS (σταθερά 500). Αν το όριο αυτό ξεπεραστεί, θεωρείται ότι έχει εντοπιστεί ένας κύκλος και απαιτείται ανακατακερματισμός.

Ανακατακερματισμός (Rehash)

Συνθήκη Ανακατακερματισμού

Ο ανακατακερματισμός ενεργοποιείται όταν η εισαγωγή ενός στοιχείου αποτύχει να βρει μια κενή θέση εντός του ορίου MAX_KICKS.

Διαδικασία

Η μέθοδος rehash() εκτελεί τα εξής:
- **Διπλασιασμός Χωρητικότητας:** Ο capacity διπλασιάζεται.
- **Αποδοτική Μεταφορά Παλιών Πινάκων:** Οι παλιοί πίνακες μεταφέρονται με **`std::move`** σε τοπικές μεταβλητές, επιτυγχάνοντας $O(1)$ μεταφορά των πόρων (χωρίς αντιγραφή) πριν την εκκαθάριση των κύριων πινάκων.
- **Επαναφορά Πινάκων:** Δημιουργούνται νέοι, κενοί πίνακες table1 και table2 με τη νέα χωρητικότητα.
- **Επανεισαγωγή:** Όλα τα στοιχεία από τους παλιούς πίνακες επανεισάγονται στους νέους πίνακες.

Αναζήτηση (Search)

Η αναζήτηση (find / search) είναι η απλούστερη λειτουργία του Cuckoo Hashing, καθώς το στοιχείο μπορεί να βρίσκεται μόνο σε δύο πιθανές θέσεις, εγγυώμενη $O(1)$ χρόνο αναζήτησης στην χειρότερη περίπτωση: Στη θέση h1(key) του table1. Στη θέση h2(key) του table2. Η συνάρτηση επιστρέφει ένα **`ValueSpan<Key>`**. Εάν βρεθεί το κλειδί: Αν **`count == 1`** (Inline Optimization), η τιμή διαβάζεται απευθείας από το πεδίο `last_segment` του bucket. Αν **`count > 1`**, το span χρησιμοποιεί τους δείκτες `first_segment` και `segments` για να ανακτήσει την αλυσίδα τιμών από το κοινόχρηστο `value_store`.
