**Ülesanne 1**

1. Räsimine on andmestruktuuride ja algoritmide valdkonnas kasutatav meetod, mis võimaldab kiiret andmete haldamist ja otsingut. Põhikontseptsioon seisneb selles, et antakse sisendandmetele funktsioon, mida nimetatakse räsifunktsiooniks, mis määrab kindlaks unikaalse identifikaatori, mida nimetatakse räsiteks. Räsiväärtus on tavaliselt fikseeritud pikkusega.
2. a) Üks ühele vastavus: Iga sisendi korral peaks räsifunktsioon genereerima ainult ühe konkreetse räsi väärtuse.

b) Unikaalsus: Hea räsifunktsioon peaks genereerima erinevaid räsi väärtusi erinevate sisendandmete jaoks, tagades, et erinevad sisendid ei anna sama räsi.

c) Kiirus: Räsifunktsioon peaks olema kiire ja efektiivne, et tagada tõhus räsimine suurte andmemahtude puhul.

1. a) Eraldi aheldamine (Separate Chaining): Selles tehnikas hoitakse igas räsiväärtuse pesas listi (ahelat) erinevatest sisenditest, mis jagavad sama räsiväärtust. Kokkupõrge ei ole probleem, sest erinevad sisendid võivad jagada sama pesa.

b) Avatud aadressimine (Open Addressing): Siin proovitakse kõikides pesades (aadressides) leida vaba koht räsiväärtuse jaoks. Kui soovitud pesa on juba hõivatud, siis proovitakse järgmist vaba pesa vastavalt teatud reeglitele, nagu lineaarne proovimine või kvadratuur proovimine.

**Ülesanne 2**

1. Python files githubis olemas.
2. a) Aja komplekssus: Triviaalse räsimise aja komplekssus on O(n), kus n on sisendmassiivi pikkus. Iga sisendi jaoks teeme konstantse arvu operatsioone, sõltumata sisendi suurusest.

b) Ruumikomplekssus: Ruumikomplekssus on O(m), kus m on maksimaalne sisendi väärtus. Loome massiivi (index\_map), mille suurus on m + 1. Seega sõltub meie lisaruum ainult sisendi suurimast väärtusest.

1. Indeksi kaardistamine massiividega on kasulik mitmetes olukordades: Andmebaaside indeksid: Andmebaasides kasutatakse sageli indekseid kiirete otsingute jaoks, räsimine failide haldamisel: failide haldamisel võib indeksi kaardistamine aidata kiiresti leida ja muuta konkreetseid kirjeid suurtes failides, kus otsimine võib olla aeganõudev, Sõnastiku implementatsioonid: keeleprotsessimise rakendustes või muude tekstianalüüsi ülesannete jaoks võib indeksi kaardistamine olla efektiivne meetod sõnastike loomiseks ja haldamiseks.

**Ülesanne 3**

1. Näide kasutamisest:

hash\_table = SeparateChainingHashTable(size=5)

hash\_table.insert(5, "A")

hash\_table.insert(10, "B")

hash\_table.insert(15, "C")

print("Search 10:", hash\_table.search(10))

print("Search 20:", hash\_table.search(20))

2. Efektiivsuse võrdlus:

Aeg: Separate Chaining: O(1) keskmine otsimisaeg, sest linked-listi pikkus ei ole liiga suur. Open Addressing: O(1) keskmine otsimisaeg, kui kokkupõrked on haruldased.

Ruum: Separate Chaining: O(n + m), kus n on tabeli suurus ja m on elementide arv. Open Addressing: O(n), kus n on tabeli suurus, kuid võib olla suurem kui Separate Chainingu puhul.

3. Separate Chaining Plusse ja Miinuseid:

Plussid:

Lihtne implementeerida ja mõista.

Võimaldab dünaamilise suurusega muutmist.

Tõhus, kui kokkupõrked on haruldased.

Miinused:

Lisakulu seotud iga linked-listi sõlmega.

Võib olla vähem tõhus kui Open Addressing, kui kokkupõrked on sagedased.

Halb mälu kasutamine, kui tabel on hõredalt täidetud või kui linked-listid on pikad.

**Ülesanne 4**

Avatud aadressimine on meetod räsimises kokkupõrgete lahendamiseks, kus kui kokkupõrge tekib, proovitakse leida järgmine vaba pesa tabelist. See erineb separate chainingust, kus iga tabeli pesa võib sisaldada mitut elementi. Avatud aadressimise meetodid hõlmavad lineaarset otsingut, ruudulist otsingut, ja topelträsimist.

2. Tehnikate Võrdlus:

Lineaarne Otsing:

Kui tekib kokkupõrge, proovitakse järgmist vaba pesa, kuni leitakse vaba koht. Lihtne implementeerida, kuid võib tekitada klasterdamist.

Ruuduline Otsing (Quadratic Probing): Proovib pesi, mis on ruutude arvu kaugusel algsest asukohast, kuni leitakse vaba koht. Vähendab klasterdamist võrreldes lineaarse otsinguga, kuid võib tekitada klastreid mujal tabelis.

Topelträsimine (Double Hashing):

Kasutab teist räsifunktsiooni, et arvutada sammude pikkus järgmise vaba pesa leidmiseks. Võib vähendada klastreid ja on üldiselt tõhus, kuid võib olla keeruline valida sobiv teine räsifunktsioon.

3. Efektiivsuse Erinevused Olukordades:

Lineaarne Otsing:

Efektiivsus: Tõhus, kui tabel on harva täidetud ja kokkupõrked on haruldased.

Sobivus: Sobib hästi juhul, kui sisendväärtused on järjestikused.

Ruuduline Otsing:

Efektiivsus: Vähendab klasterdamist võrreldes lineaarse otsinguga.

Sobivus: Sobib juhtudeks, kus on tõenäoline, et tekivad pikad klastreid.

**Ülesanne 5**

1. Näide

hash\_table = DoubleHashingHashTable(size=5)

hash\_table.insert(5, "A")

hash\_table.insert(10, "B")

hash\_table.insert(15, "C")

print("Search 10:", hash\_table.search(10))

print("Search 20:", hash\_table.search(20))

1. Analüüs: Aja- ja Ruumikomplekssus:

Aja komplekssus: O(1) keskmine otsimise aeg, kui tabel ei ole liiga täis, ja kollisioonid on haruldased. O(n) halvim juht, kui tabel on täis ja kollisioonid on sagedased.

Ruumikomplekssus: O(n), kus n on tabeli suurus.

1. Stsenaarium, kus topelt räsimine on kasulik, võib olla mitme korraga töötavate andmete haldamine, kus võtmed genereerivad sarnaseid räsi väärtusi. Topelt räsimine võib parandada üldist jõudlust sellistes olukordades.