1. Otsimine: paiskmeetod

Kas on otsimiseks võimalik leida paremat ajalist keerukust kui `O( log n)` (sellise keerukuse tagasid kahendotsimispuu ja kahendotsimine sorteeritud massiivis)? Parem saab olla konstantne keerukus `O(1)`, mis tähendaks seda, et on teada, kust õiget kirjet otsida ja andmete mahu muutumine ei mõjuta oluliselt otsmise aega. Kui andmestik, millega tegeletakse, ei pea võimaldama muud kui lisamist, otsimist ja kustutamist, on **paisktabel** mõistlik lahendus.

Otsimise jaoks on kirjes fikseeritud võtmeväärtus, mis peab olema unikaalne - arvete numbrid, isikukoodid jne. Kui arvete numbrid oleksid vahemikus 1..100, siis saaks teha tabeli (massiivi) ja paigutada andmed tabeli lahtritesse 1..100 vastavalt arve numbrile (arve number on indeksiks). Kirjeldatud salvestusviis sobib juhul, kui võtmeid on vähe ja nad on mõistlikult piiratud vahemikus. Sellist meetodit kutsutakse **otsene adresseerimine** *(direct-addressing)*. Tabelit, kuhu andmed salvestatakse, nimetatakse **otseadresseerimisega tabeliks** *(direct-address table)*. Kirjed võivad olla salvestatud otse tabelisse või on tabelis kirje aadress.

Kui aga arveid on küll 100, aga arve numbrid on vahemikus 1..10000, siis nende paigutamine 10000 lahtriga tabelisse on raiskamine. Arve numbritele tuleks rakendada mingit arvutust, et tegelikest väärtustest saaks väärtused vahemikus 1..100. Leitud väärtuse järgi paigutatakse kirje tabelisse.

Kui võtmete väärtused oleksid 100, 200, 300, ...9900, 10000, siis rakendades tehet `key div 100` saame väärtused vahemikus `1 .. 100` ja andmed paigutatavad tabeli lahtritesse 1..100.

> **Paiskmeetod** (ka **räsimeetod**) *(hashing)* on algoritm, mis paneb suvalise pikkusega andmehulga vastavusse fikseeritud pikkusega andmehulgaga. Tavaliselt on vasteks täisarv, mida saab kasutada massiivi indeksina.
>
> **Paiskfunktsioon** (**räsifunktsioon**) *(hash function)* on algoritm, mis arvutab suvalisele väärtusele vasteks täisarvu nii, et see mahub etteantud vahemikku. Paiskmeetodi kontekstis on vahemikuks paisktabeli pikkus ehk leitud täisarv peab sobima tabeli (massiivi) indeksiks.

Paiskfunktsioon `h(k)` vastendab võtme tabeli lahtrile, leides selleks sobiva indeksi.

Teisisõnu: igas kirjes eraldatakse üks väli, mis on **võtmeks** *(key)*. Sellele võtmele rakendatakse **paiskfunktsiooni** *(hash function)*, mis võtme väärtusele järgi arvutab massiivi indeksi (tabeli lahtri aadressi). Paiskfunktsioon ehk arvutuseeskiri tuleb valida selliselt, et arvutuse tulemus mahuks tabeli indeksite vahemikku.

> Leitud indeksit (aadressi) nimetatakse **paiskväärtuseks** *(hash value)*.

Ei tohi unustada kogu meetodi eesmärki: **otsimist**. Kirje otsimiseks võtmega `k` rakendatakse võtmele paiskfunktsioon (`h(k)`), saadakse indeks ja vaadatakse indeksi järgi tabeli lahtrisse. Kui seal paikneb otsitav kirje, oli otsimine edukas. Kui ei paikne, on otsimine mitteedukas.

Võib tekkida olukord, kus mitu erinevat võtit arvutatakse samaks indeksiks. Näiteks on võtmed 500 ja 588. Kui mõlemat võtit jagada täisarvuliselt 100ga, on tulemuseks 5, st mõlemad kirjed tuleks paigutada lahtrisse indeksiga 5, mis pole võimalik. Sellist olukorda kutsutakse **kollisiooniks** ehk **põrkeks** *(collision)*.

Kollisioonide lahendamiseks tuleb midagi ette võtta, sest tavaliselt pole saabuvate võtmete väärtused täpselt teada ja seetõttu on raske leida paiskfunktsiooni, mis garanteeritult kollisioonivabalt töötaks. Ülihea funktsioon võib väga keeruline olla ja arvutamine kaua aega võtta.

Paisksalvestuse realiseerimisel on seega kaks võtmeprobleemi:

1. Milline on hea paiskfunktsioon?
2. Kuidas lahendada kollisioone?

1.1. Paiskfunktsioon

Hea paiskfunktsioon peab olema:

- **lihtne** - kiirelt ja kergelt arvutatav
- **determineeritud** – sama sisend peab alati andma sama väljundi
- **ühetaoline** *(uniform)* - jagama kirjed tabelisse võimalikult ühtlaselt (arvutama indeksid, mis ühtlaselt jaotuvad).

Paiskfunktsiooni `h(k)` väärtused peavad paiknema vahemikus: $0 \le h(k) < M$ kõigi võimalike võtmete `k` jaoks (`M` on tabeli pikkus).

Võti, mille järgi andmeid tabelisse paiskama hakatakse, ei pruugi algselt olla arv. Kui on tegemist tekstiga, siis saab seda teisendada arvuks kooditabeli järgi, kasutada arvu leidmiseks sõnas olevaid sümboleid või siis muutes kogu sõna tekitava bitijada kahendarvuks.

Eraldi teooriad tegelevad küsimusega, kuidas valida sobivaid teisendusi tekstist arvuks.

On leitud, et kollisioonid hakkavad tekkima tõenäosusega üle 0,5, kui tabelisse pikkusega `N` on paigutatud `sqrt(Pi*n/2)` kirjet. (Sünnipäevade paradoks – 23 juhuslikult valitud inimese hulgast on kahel inimesel ühel kuupäeval sünnipäev tõenäosusega 0,5; 43 inimese puhul on tõenäosuseks 0,9; tabeli pikkuseks on 365).

Järgnevalt kaks näidet paiskfunktsioonidest.

1.1.1. Jäägimeetod

Jäägimeetodil *(division method)* leitakse jääk, mis tekib võtme täisarvulisel jagamisel tabeli pikkusega. Tekkiv jääk mahub garanteeritult tabelisse. Funktsioon on lihtne ja kiiresti arvutatav. Seega paiskfunktsioon selle meetodi jaoks on:

`h(k)=k % M`,

kus `k` on võti ja `M` on paisktabeli pikkus. Täpsemalt on paisktabelis aadressid / indeksid `0..M-1`.

Oluline on tabeli suuruse `M` valik, sest see võib mõjutada kollisioonide teket. Halvaks tabeli pikkuseks loetakse arvu, mis on võtmeks oleva arvu arvusüsteemi alus (näit 10, ka 2), samuti arvu, mis sellest arvust väga vähe erineb, paarisarve, kahe astmeid jms.

Sobivaks tabeli pikkuseks on algarvud, mis aga vastavad ka eelmistele tingimustele (ei ole liiga lähedal arvusüsteemi alusele, kahe astmele).

Jäägimeetod võib olla ka üks osa paiskväärtuse leidmisest, kus eelnevalt on näiteks välja arvutatud tekstilise võtme arvuline väärtus ning seejärel viiakse see jäägimeetodi abil tabeli pikkuse jaoks sobivale kujule.

1.1.2. Korrutamise meetod

Korrutamise meetodi *(multiplication method)* puhul korrutatakse võti mingi irratsionaalarvuga $(0 < A < 1)$ ja täisosa võetakse ära. Järgi jääb arv vahemikus 0 kuni 1. Leitud arv korrutatakse tabeli pikkusega `M`, tulemusest jäetakse alles täisosa. See täisosa mahub alati `0` ja `M-1` vahele. Milline on hea arv, millega korrutada? Jättes kõrvale kõik tõestused, mida targad inimesed on teinud, võib väita, et leidub üks tore arv, millega korrutades head tulemused saavutatavad on, selleks arvuks on nn. kuldlõige, mis vahemiku pikkusega 1 jaoks on:

$$T=\frac{\sqrt{5}-1}{2}=0{,}618033$$

Paiskfunktsioon ise on järgmine:

`h(k)=[M*(k*T - [k*T])]`

kus kandilised sulud tähistavad täisosa sulgudes olevast arvust.

Kui näiteks võtmed on 1..10 ja tabeli pikkus on 10, siis rakendades vastavat funktsiooni saame järgmised arvud, nagu näha tabelis 1.

Tabel 1. Korrutamismeetodil leitud paiskväärtused võtmetele 1 .. 10.

Võti	Paiskfunktsiooni rakendamine	Paiskväärtus
1	[10*(1*T - [1*T])]	6
2	[10*(2*T - [2*T])]	2
3	[10*(3*T - [3*T])]	8
4	[10*(4*T - [4*T])]	4
5	[10*(5*T - [5*T])]	0
6	[10*(6*T - [6*T])]	7
7	[10*(7*T - [7*T])]	3
8	[10*(8*T - [8*T])]	9
9	[10*(9*T - [9*T])]	5
10	[10*(10*T - [10*T])]	1

1.2. Lõplik paisksalvestus

Parimal juhul on algoritmi kiiruseks nii võtme lisamisel, kustutamisel kui ka otsimisel O(1). Halvim olukord tekib siis, kui kõigi võtmete jaoks arvutatakse sama paiskväärtus. Sel juhul kiiruseks O(N), sõltumata kollisioonide lahendamise meetodist.

Lõplik paisksalvestus *(perfect hashing)* on olukord, kus kõik võtmed paigutuvad paisktabelisse oma positsioonidele kohe ja lõplikult ilma et tekiks mingeid kollisioone.

See olukord oleks võimalik juhul, kui kõik võtmed k on eelnevalt teada ja sellest lähtuvalt saaks koostada sobiva paiskfunktsiooni. Tavaliselt kõik võtmed kohe teada ei ole.

Erinevaid paiskfunktsioone kasutades tekivad erinevad tabelid. Kui andmed (võtmed) on erinevatel töötlemisetel sarnased (näit. sarnased muutujate nimed programmis), siis sama paiskfunktsiooni kasutades tekivad alati samasugused kollisioonid ja pole välistatud ka halvima olukorra tekkimine. Selle olukorra lahendamiseks nähakse võimalust mitme paiskfunktsiooni kasutusse võtmises, millest iga kord valitakse juhuslikult üks funktsioon. Nii peaks tekkima olukord, et kui ükskord nende võtmete puhul tekkis palju kollisioone, siis teine kord teist paiskfunktsiooni kasutades seda ei teki (st. üks kord läheb halvemini ja teine kord paremini).

1.3. Kollisioonide lahendamine

Enne kollisioonide lahendamist tuleb vähendada kollisiooniohtu. Selleks on võtted järgmised:

- Paisktabel kirjeldatakse 1/4 .. 1/3 võrra suurem, kui tegelikult vaja. Täiendavad positsioonid tabelis vähendavad kollisioonide tekkimise ohtu.
- Tuleb leida selline paiskfunktsioon, mis indeksid/paiskväärtused võimalikult ühtlaselt üle kogu tabeli arvutaks.

Ükskõik kui kavalalt me paiskfunktsiooni ka ei valiks või kui suurt tabelit ei teeks, kollisioonidest täielikku pääsu loota ei tasu. Seega tuleks uurida, mida võtta ette siis, kui kollisioonid tekivad. On põhimõtteliselt kaks lahendust:

- Paigutada kollisiooni läinud kirjed paisktabelist välja nn kollisiooniahelatesse e. moodustada ahelad, mis algavad tabeli lahtritest ja kuhu paigutatakse sama paiskaadressiga kirjed.
- Otsida “kavala” valemi abil tabelis uus koht ja panna kirje sinna; sel juhul peavad kõik kirjed mahtuma tabeli piiresse.

1.3.1. Kollisioonis olevate kirjete paigutamine ahelasse

Kollisiooniahelate meetod seisneb selles, et kirjed, millel tekib kollisioon (sama paiskväärtusega kirjed), seotakse ahelasse. Paisktabeli lahtris indeksiga `h` on aadress selle ahela esimesele elemendile, kuhu paigutatakse kõik paiskväärtust `h` omavad kirjed. Selle meetodi puhul ei pea tabel ka alati 1/4 .. 1/3 võrra suurem olema. Ja pole hullu, kui elemente on tegelikult rohkem, kui tabelis lahtreid.

Näide.

Nüüd ja edaspidisteks näideteks lepime kokku:

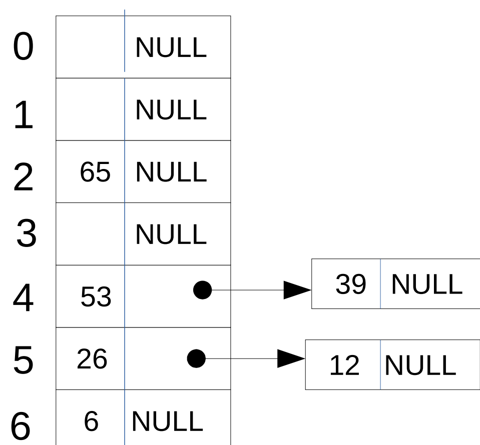
- On väike 7 lahtriga paisktabel (indeksid 0 .. 6).
- On võtmed, mis on täisarvud kümnendsüsteemis. Muu kirjes paiknev info meid ei huvita.
- Võtmed lisatakse tabelisse samas järjekorras, nagu allpool tabelis näha.
- On paiskfunktsioon, mis leiab paiskväärtuse jäägimeetodil `h(k) = k % m`.

Järgnevas tabelis 2 on võtmed ja jäägimeetodil arvutatud paiskaadressid. Nagu näha, tekib kahe võtme puhul kollisioon :

Tabel 2. Võtmed ja paiskväärtused.

Võti	**Paiskväärtus**
26	5
53	4
12	5 (kollisioon)
65	2
39	4 (kollisioon)
6	6

Järgnevalt jooniselt (Joonis 1) on näha võtmete paigutus tabelisse, kasutades kollisiooniahelaid ning kollisioonis kirjete **eraldi seostamist** *(separate linking)*.



Joonis 1: Paisktabel kollisiooniahelatega.

Põhitegevused paisktabeliga on **otsimine**, **lisamine** ja **kustutamine**.

Otsimine võtme `k` järgi:

Arvuta paiskväärtus `h(k)`

Kontrolli, kas võti `k` on tabelis `t` kohal `t[h(k)]`

Kui lahter on tühi, siis otsimine oli mittedukas,

vastasel juhul (lahter täidetud):

> Kui võti selles lahtris ei ole `k`, siis läbi vastav kollisiooniahel.
>
> Kui võtit `k` ei leitud, siis mitteedukas otsimine.
>
> Kui võti `k` oli lahtris `t[h(k)]` või ahelas, siis oli otsimine edukas.

Lisamine võtme `k` järgi (eeldusel, et kirjet võtmega `k` tabelis ei ole.)
Arvuta paiskväärtus `h(k)`

Kui tabelis vastava indeksiga lahter on tühi, siis pane andmed sinna,

vastasel juhul lisa uus element kollisiooniahelasse.

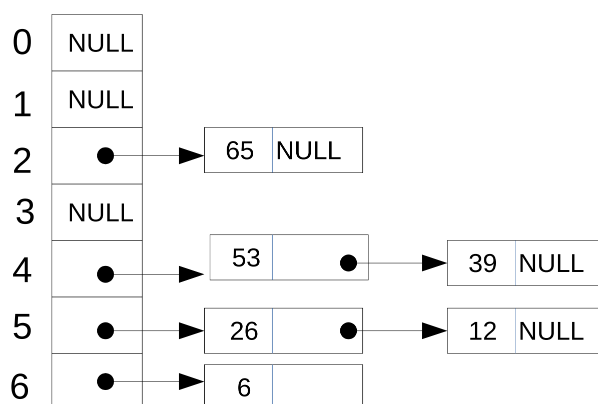
Kirje kustutamine võtme k järgi:
Otsi võtme `k` järgi.

Kui otsimine oli edukas, siis saab kustutada, selleks:

> Kui `k` on tabelis, kirjuta tema asemele esimene element kollisiooniahelast (kui ahel olemas).
>
> Kui `k` on ahelas, siis kustuta sealt vastav element (nagu ühe viidaga ahelast)

Kirjete **otsese seostamine** korral *(direct linking)* on tabelis ainult viidaväljad, kuhu kirjutatakse aadressid ahelate algusele ja kõik kirjed paiknevad tabelist väljas ahelates.

Joonisel (Joonis 2) on eespoolt tuttavate kirjetega paisktabel.



Joonis 2: Paisktabel kollisiooniahelatega – otsene seostamine

Otsimise, lisamise ja eemaldamise jaoks pole vaja erinevalt kontrollida elemendi paiknemist tabelis või ahelas. Sisuliselt on tegemist ahela operatsioonidega.

1.3.2. Vaba paisksalvestus

Erinevalt ahelatega salvestusmeetodist tuleb **vaba paisksalvestuse** *(open hashing)* puhul kõik võtmed tabelisse ära mahutada. Eelduseks uue võtme paigutamisel tabelisse on, et vähemalt üks tabeli lahter on vaba. Kui uue võtme jaoks arvutatakse juba hõivatud tabeli aadress `h(k)`, tuleb leida talle uus **vaba aadress** *(open adress)*. Selleks kirjeldatakse iga võtme jaoks mingi lahtriaadresside järgnevus, kuhu võtmeid paigutada proovitakse, seni kuni vaba koht leitakse. Vaadeldatavate aadresside järgnevust nimetatakse **sondeerimise (proovimise) järjekorraks** *(probing sequence)*.

Võtmega `k` sama paiskväärtust omavaid võtmeid kutsutakse `k` sünonüümideks ja tähistatakse `k'`.

Kustutamine sellise meetodi puhul on veidi kahtlane tegevus. Kui võti `k` on kustutatud, kuidas leida tabelist tema sünonüüm `k'`?. Seega tuleks lahter kustutamise asemel märgistada kustutatuks, kuid info (võti) peab sinna esialgu alles jääma. Kui tekib vajadus uute kirjete / võtmete lisamiseks sellesse lahtrisse, siis saab vana võtme üle kirjutada.

Kirje tegelik aadress sõltub sellest, mitmendat korda vastava võtmega kirjet tabelisse lisada püütakse. Sondeerimisjärjekorra leidmiseks kirjeldatakse funktsioon `s(j,k)`, kus `j` kasvab `0..m-1` ja näitab, mitmendat korda antud võtit uude lahtrisse paigutada proovitakse. Lahtrite aadresside jada, kuhu andmeid paigutada, arvutatakse välja valemiga:

`(h(k) - s(j,k)) mod m`.

Uus aadress sõltub esialgsest aadressist, sellest mitmendat korda proovitakse ja võtmest `k`. Põhilised tegevused (tingimuseks on, et vähemalt üks tabeli lahter on vaba) on endiselt otsimine, lisamine ja kustutamine:

Otsimine võtme k järgi

Alusta lahtri aadressist `i = h(k)`.

Otsi seni kuni leiad `k` või kuni tabeli `t` lahter `t[i]` on vaba.

Järgmine `i` arvuta valemist `i=(h(k) - s(j,k)) mod m` selliselt, et `j` kasvab `0 .. m-1`.

Kui `t[i]` on vaba või kustutatud, siis oli otsimine edutu, vastasel juhul edukas.

Lisamine võtme k järgi

Eeldame, et võtit `k` tabelis ei ole. Alusta aadressist `i=h(k)`.

Jätka lahtriaadresside leidmist valemiga `i=(h(k) - s(j,k)) mod m` kuni lahter aadressiga `i` on vaba või kustutatud.

Lisa sinna uus võti `k.`

Kustutamine võtme k järgi

Otsi võtme `k` järgi.

Kui otsimine oli edukas märgista `t[i]` kustutatuks.

Milline oleks hea sondeerimisfunktsioon `s(j,k)`? Tavaliselt kasutatakse kolme erinevat võimalust: lineaarne sondeerimine, ruutsondeerimine või topeltpaisksalvestus.

1.3.2.1. Lineaarne sondeerimine

Lineaarse sondeerimise *(linear probing)* jaoks on uue koha otsimise järjekord määratud nii:

`h(k), h(k)-1, h(k)-2, ...`

Seega funktsioon `s(j,k) = j`.

Esimesest paiskväärtusest lahutatakse proovimise järjekorra number ja seejärel leitakse uuesti jääk. Uus jäägi leidmine tagab, et tabelis üle serva ei minda (näit. `-1 mod 7 = 6`). Tabelis proovitakse

lahtreid selliselt: kui esimese proovimise indeks oli `i`, siis järgmine on `i-1`, edasi `i-2` jne kuni jõutakse indeksini 0. Edasi jätkub proovimine tabeli lõpust. Sama meetodit kirjeldatakse ka liitmisega.

Lineaarse sorteerimise valem üldisemalt näeb välja järgmine:

`(h(k) - s(j,k)) mod m`, kus `s(j,k) = j`

Võtmete `26, 53, 12, 65, 39, 6` toodud järjekorras lisamisel tekib järgmine tabel (vt tabel 3). Iga rida kujutab tabeli seisu peale järgmise võtme lisamist - st tekkiv tabel ei ole kahemõõtmeline, vaid ikka ühemõõtmeline. Lihtsalt veidi kergem oli seda ühes tükkis joonistada.

Tabel 3 Lineaarse sondeerimise abil lahendatud kollisioonid – tabeli seisundid peale iga võtme lisamist

Indeks Lisatav võti	0	1	2	3	4	5	6
26						26	
53					53	26	
12				12	53	26	
65			65	12	53	26	
39		39	65	12	53	26	
6		39	65	12	53	26	6

Probleemid

Sel meetodil tekib kirjete / võtmete **lineaarne klasterdumine** ehk **esmane kuhjumine** *(primary clustering)*. Mida pikemad tükid on tabelis järjest hõivatud, seda rohkem need tükid ka edasi kasvavad, sest ükskõik millise võtmega sellest hulgast kollisioon tekib, igal juhul pannakse uus võti juba olemasoleva hõivatud tüki kõrvale. Lineaarse sondeerimise efektiivsus kahaneb tunduvalt, kui tabeli täidetus läheneb 100 protsendile, sest pikalt on vaja otsida nii tühja kohta lisamiseks kui ka võtit (mida ei ole).

Hinnang

Vastavalt R. Sedgewick'i raamatule *“Algorithms in C”* on lineaarsel sondeerimisel sammude (katsete) arvud võtme otsimisel näha tabelis 4. Tulemused on leitud statistilisi meetodeid kasutades.

Tabel 4. Lineaarse sondeerimise sammude arv (R. Sedgewick)

Tabeli täidetus:	'1/2	2/3	'3/4	9/10
Edukas otsing	1,5	2,0	3,0	5,5
Edutu otsing	2,5	5,0	8,5	55,5

1.3.2.2. Ruutsondeerimine

Ruutsondeerimisega *(quatratic probing)* välditakse esmast kuhjumist, kuid selle asemel tekib **teisane kuhjumine** *(secundary clustering)*, mis siiski pole nii tugev. Meetod seisneb selles, et uut kohta kirjele võtmega `k` otsitakse üha kaugemalt võtme paiskväärtuses. Kaugus kasvab kord ühele ja kord teisele poole `h(k)`-d `j` (proovimise järjenumbri) ruudu võrra. Võimalik on kasutada ka teisi konstante uute positsioonide arvutamiseks, kuid põhiidee kauguse kiirema kasvamise osas on sarnane.

Sondeerimiste jada kujuneb järgmiseks:

`h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, ...`

Sondeerimisfunktsioon on $s(j,k) = ([j/2])^2 * (-1)^j$

$(-1)^j$-ga saavutatakse + ja – märkide vaheldumine.

Sarnaselt lineaarse sondeerimisega tuleb iga kord uue aadressi arvutamisel rakendada jäägi leidmist, muidu võib kiiresti sattuda tabelis "üle serva".

Teeme jälle näite võtmetest `26, 53, 12, 65, 39, 6`. Järgnevas tabelis 5 on kokkuvõte - tabeli seis peale viimase võtme 6 lisamist.

Tabel 5. Ruutsondeerimise rakendamise tulemus.

Indeks:	0	1	2	3	4	5	6
Võti:	6		65	34	53	26	12

Nagu juba öeldud, tekib siin teisane kuhjumine (väikese tabeli peal see ei avaldu), sest paratamatult sünonüümide `k` ja `k'` korral läbitakse sama aadresside proovimisjada.

1.3.2.3. Topelt paisksalvestus

Topelt paisksalvestuse *(double hashing)* idee seisneb järgnevas: võtame kasutusse teise paiskfunktsiooni `h'(k)`, mida samale võtmele rakendades leiame sammu võtme uue aadressi leidmiseks.

Tekkiv sondeerimisjada võib olla näiteks järgmine:

```
h(k), (h(k) - 1*h'(k)) mod m,
(h(k) - 2*h'(k)) mod m,
(h(k) - 3*h'(k)) mod m, ...
```

Sondeerimisfunktsioon, mis seekord sõltub nii võtmest `k` kui ka proovimiste arvust, on seega:
`s(j,k) = j * h'(k)`

Mis sobib teiseks paiskfunktsiooniks? Funktsioon ei tohi sõltuda esimesest funktsioonist, peab jaotama sammupikkused ühtlaselt ja mitte võrduma 0-ga (muidu aadress ei muutuks). Üks võimalus on järgmine:

`h'(k) = 1 + k mod (m-2)` (Või ka `m-1`)

Seega võttes meie tuttava tabeli pikkusega 7, on teiseks funktsiooniks:

`h'(k) = 1 + k mod (7-2) = 1 + k mod 5`

Teeme taas näite võtmetest `26, 53, 12, 65, 39, 6`. Järgnevas tabelis on kujutatud igas reas paisktabeli seisu peale järgmise võtme lisamist. Viimases tabeli veerus on tehe, millega lisatavale võtmele paiskväärtus leitakse. Nagu näha, siis osa võtmeid sobitub tabelisse esimesel katsel, kuid võtme 6 paigutamiseks tehakse kokku 4 katset.

Tabel 6. Topeltpaisksalvestusel tekkiv tabel.

Indeks võti	0	1	2	3	4	5	6	
26						26		26 mod 7=5
53					53	26		53 mod 7=4
12			12		53	26		(5-(1+12 mod 5)) mod 7=2
65		65	12		53	26		(2-(1+65 mod 5)) mod 7=1
39		65	12		53	26	39	(4-(1+39 mod 5)) mod 7=6
6	6	65	12		53	26	39	(6-(1+6 mod 5)) mod 7=4 (6-2*(1+6 mod 5)) mod 7=2 (6-3*(1+6 mod 5)) mod 7=0

Hinnang

Vastavalt R. Sedgewick'i raamatule "Algorithms in C" on topelt paisksalvestusel sammude (katse) arvud ühe võtme kohta järgmised. Tulemused on leitud statistilisi meetodeid kasutades.

Tabel 7. Hinnang topeltpaisksalvestusele (R. Sedgewick)

Tabeli täidetus:	'1/2	2/3	'3/4	9/10
Edukas otsing	1,4	1,6	1,8	2,6
Edutu otsing	1,5	2,0	3,0	5,5

Võrdlusest lineaarse meetodiga võib järeldada ka seda, et topeltpaisksalvestuse jaoks on võimalik tabelit hoida väiksemana kui lineaarse proovimise korral.

1.4. Paiskmeetodi kiirusest ja võrdlusest kahendotsimispuuga

Eeldades, et paisktabelisse paigutatavad võtmed on juhuslikud, saab tõestada, et tabeli osalise täidetuse korral toimub tuvastamine, et elementi pole keskmiselt ajaga O(1). Kui näiteks tabelist on täidetud pool, ei ole keskmine arv katseid suurem kui 2 ja kui tabel on täidetud 90%, pole keskmine katsete arv suurem kui 10.

Eduka otsimise korral saadakse otsitav kätte pooleldi täidetud tabelist keskmiselt 1,387 katsega ja 90% täidetud tabelist 2,559 katsega. Toodud arvude tõestuseks on mõned teoreemid ja valemid, mida T. Cormen jt raamatust *"Introduction to Algorithms"* täpsemalt uurida saab.

Paiskmeetodit eelistatakse tavaliselt kahendotsimispuule, sest tema realiseerimine on puust lihtsam (ikkagi tavaline massiiv) ja lisaks annab ka reeglina parema aja (konstantne keerukus ja aeg). Kuid eeldus on see, et võtmed peavad olema mingit standartset lihtsamat andmetüüpi, millele on võimalik leida ja rakendada lihtsat ja kiiret paiskfunktsiooni.

Samas on puu paisktabelist parem juhul, kui lisatavate võtmete arvu ei ole võimalik (või on raske) ette ennustada, sest puu on dünaamiline. Viimasest sõltub aga tabeli pikkus. Puud tagavad ka kindla ajalise keerukuse kõige halvemal juhul. Paiskmeetodil võib aga ka parima funktsiooni korral juhtuda, et kõik võtmed arvutatakse üheks paiskväärtuseks (indeksiks). Otsmispuud annavad lisaks näiteks võimaluse võtmetest sorteeritud järjekorra saamiseks (kahendotsimispuus on võtmed sisuliselt sorteeritud) ning samal ajal ka naabervõtmete leidmiseks. Viimast paisktabel ei toeta.

Järgnevalt väike võrdlus paiskmeetodi ja kahendotsimispuu keerukusklassidest.

Tabel 8. Paiskmeetodi ja kahendotsimispuu keerukusklasside võrdlus.

Paiskmeetod	**Kahendotsimispuu**
Lisamine O(1)	Lisamine O(log N)
Kustutamine O(1)	Kustutamine O(log N)
Otsimine O(1)	Otsimine O(log N)
Min O(N)	Min O(log N)
Maks O(N)	Maks O(log N)

1.5. Paiskmeetodi kasutusalad

Paiskmeetodit kasutatakse nii strktuursete andmetüüpide moodustamisel kui rakendustes, kus vaja tagada kiire otsimine. Mõned näited:

- andmebaasidele moodustatavate indeksite üks võimalik ülesehitus,
- kompilaatorid paigutava muutujate nimed muutujate tabelisse,

- struktuursed andemtüübid erinevates keeltes: *dictionary*, *associative memory*, *associative array* (nn vastendustüüp - *mapping type*)

(Pythoni *distionary* ülesehitusest võid vaadata ja kuulata siit: https://youtu.be/npw4s1QTmPg)

Sisukord

1. Otsimine: paiskmeetod....1
  1.1. Paiskfunktsioon....1
    1.1.1. Jäägimeetod....2
    1.1.2. Korrutamise meetod....2
  1.2. Lõplik paisksalvestus....3
  1.3. Kollisioonide lahendamine....3
    1.3.1. Kollisioonis olevate kirjete paigutamine ahelasse....4
    1.3.2. Vaba paisksalvestus....6
      1.3.2.1. Lineaarne sondeerimine....6
      1.3.2.2. Ruutsondeerimine....7
      1.3.2.3. Topelt paisksalvestus....8
  1.4. Paiskmeetodi kiirusest ja võrdlusest kahendotsimispuuga....9
  1.5. Paiskmeetodi kasutusalad....9