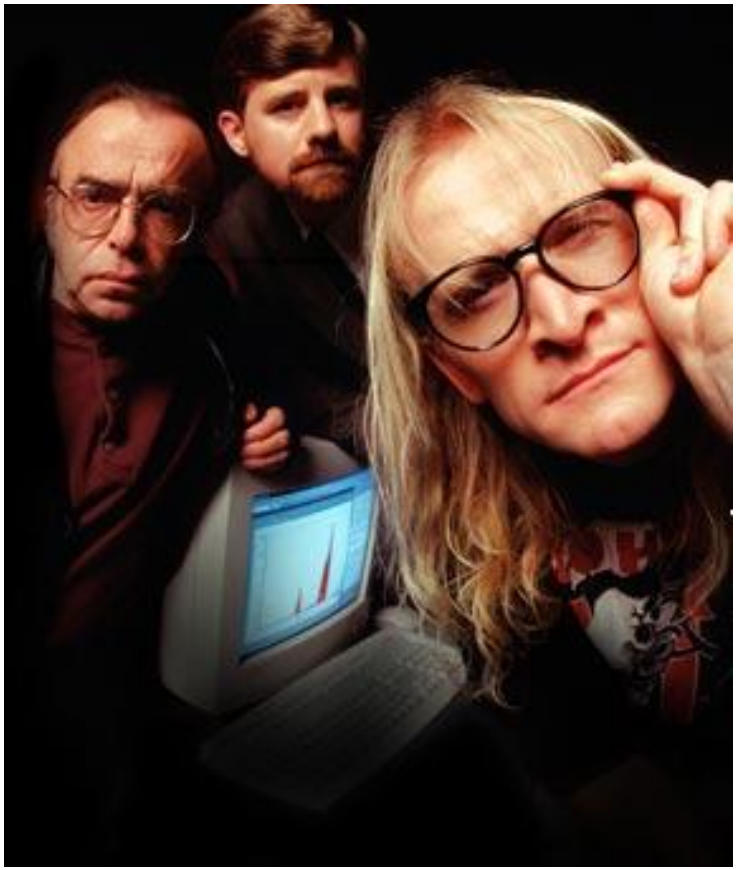
Algoritmid ja andmestruktuurid

- Lingitud andmestruktuurid
- *Dictionary, map, associative array*
- Paisksalvestus (*hashing*)

TTÜ Programmeerimisolümpiaad



IEEExtreme 24h võistlus
- **14-15. okt**

ACM olümpiaad
- **10. okt kell 16.45-22.00**
- **31. okt – 3. nov Minsk**

Registreerige kuni
- 3-liikmeline meeskond
- aadressil
- olympiaad@cs.ttu.ee

https://courses.cs.ttu.ee/pages/ProgComp

Mõisteid

- Andmetüüp - primitiivne andmestruktuur
  - int, string, massiiv, struktuur
- Abstraktne andmetüüp (ADT):
  - hulk objekte
  - nendele rakendatavad operatsioonid
  - ei kirjelda ega määra kuidas need on implementeeritud
  - vastab Java *interface* konstruktsioonile
  - Näiteks
    - täisarvud: liitmine, korrutamine, võrdlemine jne
    - hulk: ühisosa, ühend, täiend, elemendi kuuluvus
    - lingitud list: lisa, kustuta, leia
    - kahendpuu: lisa, kustuta, leia
    - järjekord: lisa, võta välja, kas on tühi
- Andmestruktuur - konkreetne implementatsioon

Lingitud andmestruktuurid

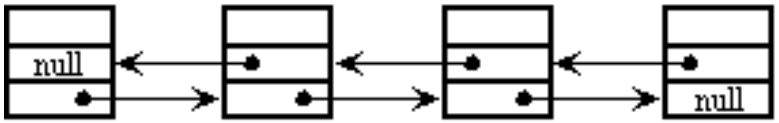
- **Objektid ja struktuurid võimaldavad viidata teisele objektile, mis on tihti sama tüüpi viitava objektiga. Sellist tegevust nimetatakse linkimiseks või viitamiseks.**
- **Dünaamilisus – objekte saab lihtsalt lisada ja eemaldada**
- **Lingitud andmestruktuuridena luuakse palju huvitavaid ja vajalikke andmestruktuure**
  - **lingitud list**
  - **mitmesugused puud**

Lingitud andmestruktuurid

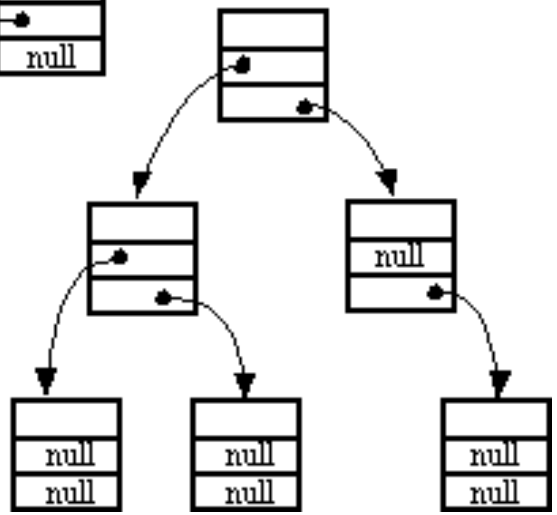
```
class Node {
   Data item;
   Node next;
}
```

When an object contains a reference to an object of the same type, then several objects can be linked together into a list. Each object in the list refers to the next.

```
class Node {
   Data item;
   Node left;
   Node right;
}
```



Things get even more interesting when an object contains two references to objects of the same type. In that case, more compliated data structures can be constructed.



Elementaarsed listi operatsioonid

```
class Node {
  Data item;
  Node next;
}
```

```
class List {
  Node first;
}
```

```
DoFooList(List list)
  Node node := list.first;
  while node != null {
    DoFooNode(node.item);
    node := node.next;
  }
}
```

```
Insert(List list, Node node){
  node.next := list.first;
  list.first := node
}
```

Konteiner

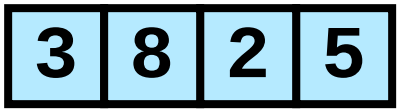
- Objekt, mis võimaldab hoida mingit tüüpi objekte ja meetodid nende objektidega ümber käimiseks:
  - lisa objekt
  - kustuta objekt
  - leia objekt
  - anna järgmine objekt

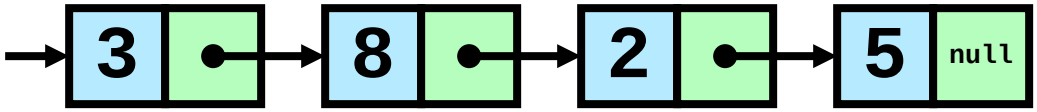
- objektid on identifitseeritavad indeksi - mingi unikaalse atribuudi abil
  - indeks on tihti midagi muud kui täisarv (nimi, atribuutide kombinatsioon, konstantne objekt)

Erinevaid konteinereid

- massiiv, dünaamiline massiiv
- lingitud list
- pinu, järjekord
- paisktabel (*hash table*)
- sõnastik (*dictionary, map, assosiated array, lookup table*)
- otsingupuu
  - binaarne
  - tasakaalustatud
  - trie
- hulk (*set, multiset/bag*)

Massiiv ja list





Massiiv

+ kiire pöördumine
- piiratud maht
- aeglane *add/delete*

List

- aeglane pöördumine
+ piiramatu maht
+ kiire *add/delete*

Dünaamiline massiiv

+ piiramatu maht

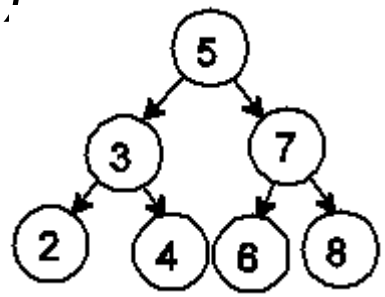
muu samamoodi

Tabel

- Hulk andmeid, mida võib ette kujutada tabelina
  - Eesnimi, perenimi, ruum, telefon
  - Toote nimetus, toote kood, hind
  - Lähtekoht, sihtkoht, väljumise, saabumise aeg, liini nr
- Ühte tabeli veergu (või veergude kombinatsiooni) käsitletakse võtmena, mis on igal real (kirjel) unikaalne
  - Eesnimi+Perenimi, arve number, toote kood, liini nr
- Vajalikud efektiivsed operatsioonid
  - lisada rida
  - kustutada rida
  - leida võtmele vastav rida

Tabeli implementatsioone

- Lingitud list
  - Otsimine ebaefektiivne – O(n)
- Võtme järgi sorteeritud massiiv
  - Otsimine (binaarotsing) O(log n)
  - Lisamine ja kustutamine ebaefektiivne – O(n)
- Otsingupuud (binaarne, tasakaalustatud)
  - Kõik operatsioonid O(log n)!



- Kas saaks veel väiksema keerukusega?
  - Otsingupuu võimaldab Min(), Max(), Next() jt teisi järjestatusel põhinevaid operatsioone, mida me ei vaja

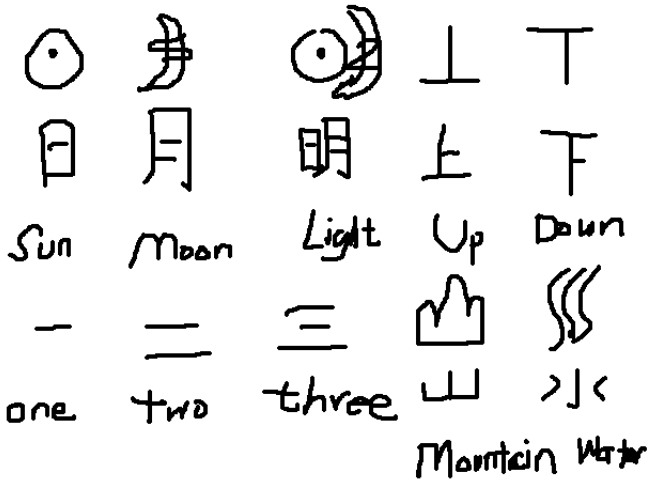
Dictionary, Map, Associative Array

- Mitu nime – sama idee

  Data[key]

  – Andmeid indekseerib võti

  – Sarnane massiivile, aga indeks ei pea olema täisarv



- Andmestruktuur, mis võimaldab

  – Andmeid lisada - *insert*

  – Andmeid võtme väärtuse järgi otsida - *find*

  – Andmeid kustutada – *delete*

- Mitteolulised operatsioonid

  – itereerimine, järjestusega seotud operatsioonid

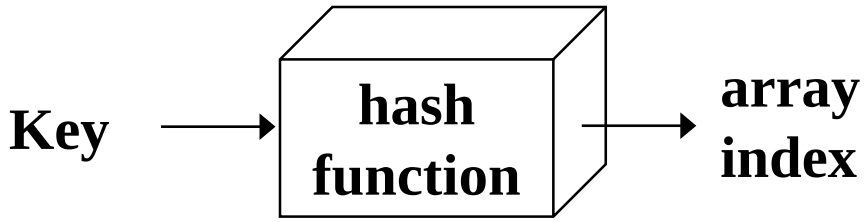
- Näiteks muutuja nimede-mäluaadresside tabel

Dictionary implementatsioon – Otsepöördustabel (*direct-access table*)

- Tavaline massiiv
- Kasutatav kui
  - Võtmeteks täisarvud vahemikus 0 … K
  - Saame reserveerida mälu A[K]
- Operatsioonid O(1)
  - Pöördume elemendi A[key] poole
- Mälukasutus võib olla ebaotstarbekas
  - Enamus massiivist on tühi
  - Itereerimine ebaefektiivne

Dictionary implementatsioon – paisksalvestus (*hashing*)

- Võtmed teisendatakse asukohaks *hash* funktsiooniga

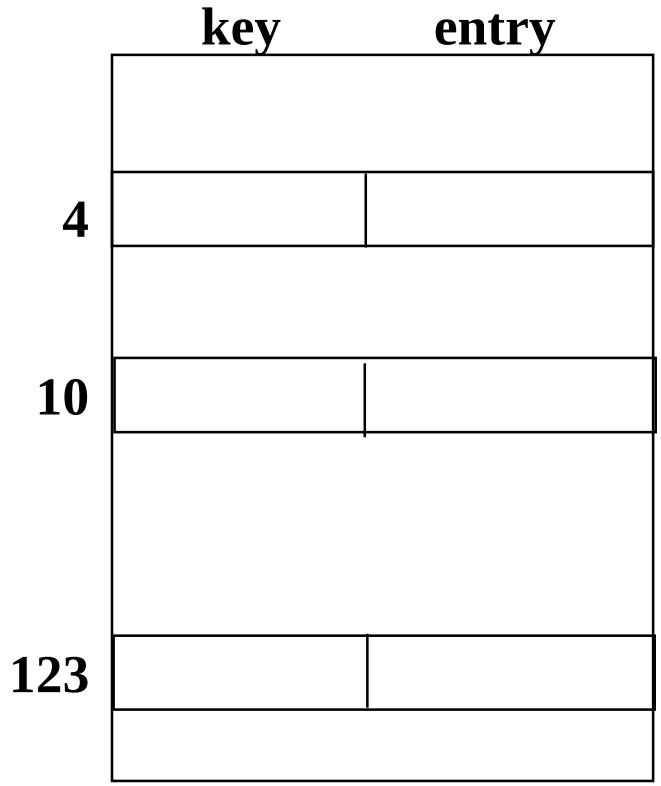


- Võtmed on hajutatud üle massiivi
  - Massiiv – A[key]
  - Hash tabel – A[hash(key)]

Dictionary implementatsioon – paisksalvestus (*hashing*)

- Massiiv, kus kirjed ei ole järjest ega järjestatud. Asukoha määrab *hash* funktsioon

- **Insert(x)**: Otsi asukoht ja salvesta - O(1)
- **Find(x)**: Otsi asukoht ja väljasta tulemus - O(1)
- **Remove(x)**: Otsi asukoht ja nulli selle sisu- O(1)



Operatsioonide keskmine keerukus O(1) !

Hash tabeli näide

10 toodet, 10 tabeli positsiooni

Toote koodid on vahemikust 0 .. 1000

Hash funktsioon h(key) = key / 100

Mis juhtub, kui lisame toote koodiga 350?

Positsioon 3 on hõivatud: *kollosioon*

Kollosiooni lahendamise strateegia (*linear probing*): kasutame järgmist vaba kohta

Toote koodi järgi leiame toote kasutades uuesti *hash* funktsiooni vajadusel kollosiooni lahendamist

	key	data
0	85	piim
1		
2		
3	323	leib
4	462	seep
5	350	makaronid
6		
7		
8		
9	912	sool

Paisksalvestus

Paisksalvestuseks on vajalik

- *Hash* funktsioon
- *Kollosioonide* lahendamise strateegia
- Sobiva suurusega *massiiv*
  - Piisav andmete hulga mahutamiseks
  - Liiga suur – mälu kadu
  - Liiga väike – palju kollosioone
  - Suuruse nõuded (algarv, kahe aste vms) sõltuvalt *hash* funktsioonist ja kollosioonide lahendamise strateegiast

Kollosioone ei saa vältida

- Sünnipäevaparadoks

  Kui suur peab olema inimeste hulk, et vähemalt kahel oleks sünnipäev samal päeval 50% tõenäosusega?

  99% tõenäosusega?

- 50% - 23
- 99% - 57

Hash funktsioon

- Teisendab võtme (suureks) täisarvuks
- Peab olema efektiivselt arvutatav
- Jaotama kirjed (võtmed) ühtlaselt
  - Peaks minimiseerima kollosioonide arvu
    - Võtmete jaotus ei pruugi olla ühtlane
  - Kui kõik võtmed on ette teada, siis on võimalik luua *täiuslik hash funktsioon*, mis väldib kollosioone
- Hash funktsioon krüptograafias
  - Hash väärtus ei tohi anda mingit infot originaali kohta
  - Efektiivne, aga keerukam kui paisksalvestuse korral

Stringi *hash*

- Lihtne meetod – liida sümbolite ASCII koodid
  - Ei pruugi anda ühtlast jaotust – *aias sadas saia*
  - Tähteda ASCII koodid on lähestikku ja summa jääb kindlasse vahemikku
- Parem meetod
  - Kui stringi esitab S[i] ja hash tabeli suurus on m, siis

$$\text{Hash}(S) = (\sum_i S[i]*2^i) \bmod m$$

  - Vahepealsed tulemused võib arvutada üle mooduli *m*

$$\sum_i (S[i]*2^i) \bmod m = \sum_i (S[i]*2^i \bmod m) \bmod m$$

  moodul – täisarvulise jagamise jääk

$$x \bmod m = x \,\%\, m$$

Suure täisarvu *hash*

- Võib arvutada mooduli üle tabeli suuruse m ($k \bmod m$)

  Ei anna head jaotust kui *k* ja *m* on ühisteguriga

  – *k* ja *m* on paarisarvud, lõppevad mingi hulga 0-dega

  – *m* võiks olla algarv

- k^2 keskmised bitid

  Kui tabeli suurus on 2^r, siis

  – Võta k^2 binaaresituse *r* keskmist bitti

  – Keskmised bitid sõltuvad *k* kõigist bittidest

- Korrutusmeetod

  – Konstandiga reaalarvulise korrutamise murdosa

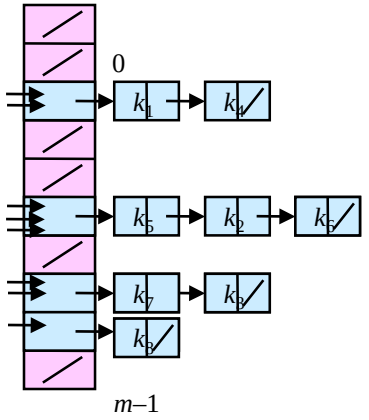
  $0 < A < 1$, $h(k) = \lfloor m\,(kA - \lfloor kA \rfloor)\,\rfloor$

  – *m* võib olla suvaline, hea *A* väärtus on $(\sqrt{5} - 1)/2$

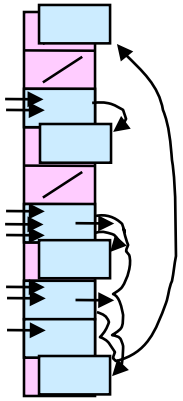
- Jenkins jt *multi-byte* hash lühendamise algoritmid

Kollosioonide (kokkupõrgete) lahendamine

- Aheldamine (*chaining*):
  - Hoiame kirjeid, mille võtme hash on võrdne lingitud listis.
  - Hash tabel hoiab lingitud listi pead.

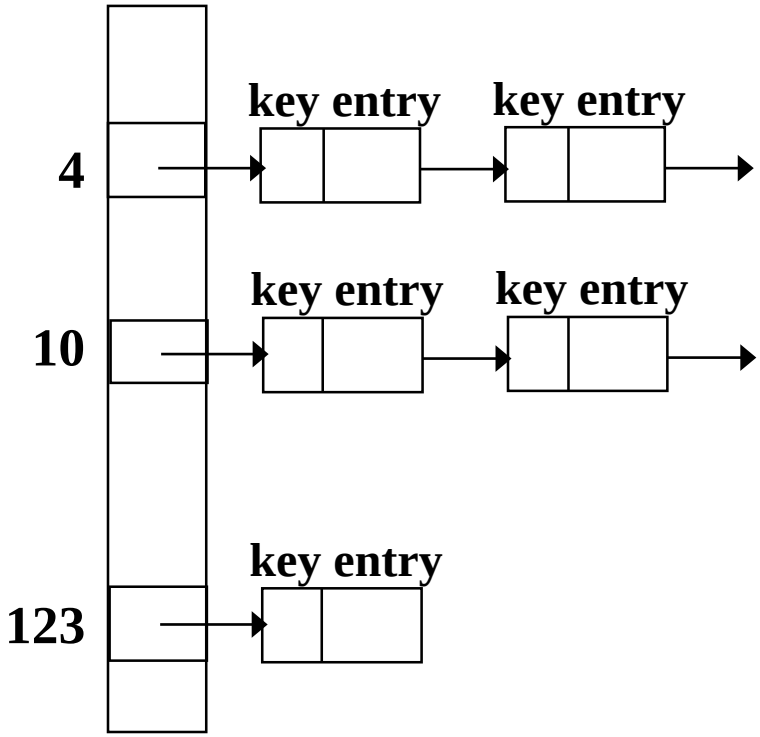


- Avatud adresseerimine (*open addressing*):
  - Kõik elemendid on hash tabelis.
  - Kollosiooni korral valitakse mingi eeskirja järgi järgmine koht.



Kollosioonide lahendamine: *aheldamine (chaining)*

- Tabeli igas positsioonis on list
- Lisa võtmed listi algusesse
- Eelised:
  - Lihtne lisamine ja eemaldamine
  - Massiivi suurus pole piiranguks
    Massiivi suurus peaks olema umbes sama suur, kui võtmete arv
- Puudused
  - Mälu lisakulu listi viitade hoidmiseks
  - Mäluhalduse lisakulu (listi elementide loomiseks ja kustutamiseks)



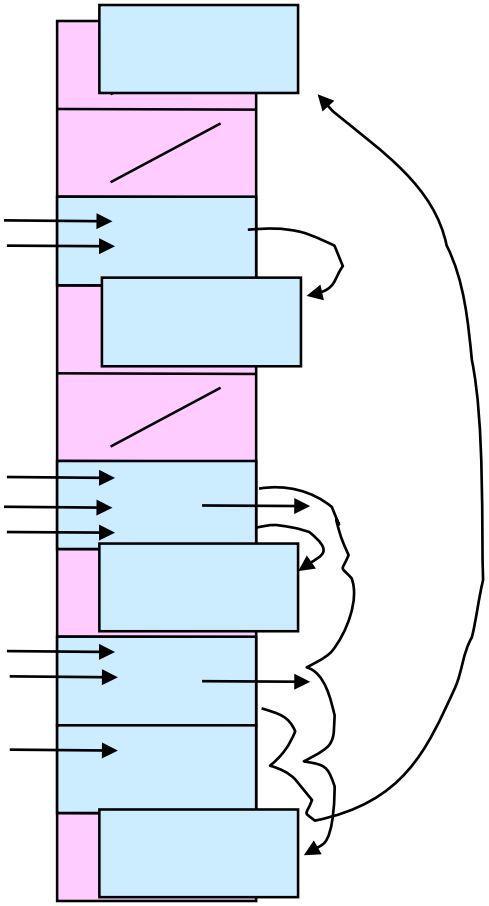
Kollosioonide lahendamine: avatud adresseerimine (*open adressing*)

- Kasutab sondeerimist (*probing*) – kui hash funktsiooni poolt määratud koht on hõivatud, siis hüppa edasi ja proovi seda kohta, kuni leidad vaba koha
  - Tabeli lõpust hüpatakse algusesse

    $pos \bmod m$

- Sama strateegiat tuleb kasutada nii lisamise kui otsimise korral

- Positsiooni määrab $\mathrm{Hash}(k,i)$, kus k on võti ja i on sondeerimise number



Lineaarne sondeerimine (*Linear Probing*)

$h(k, i) = (h'(k)+i) \bmod m.$

võti — sondeerimise number — algne hash funktsioon

- Sondeerimiste järjekord on
  - $T[h'(k)], T[h'(k)+1], \ldots, T[m-1], T[0], T[1], \ldots, T[h'(k)-1]$
- Puuduseks ***primaarne klasterdumine***:
  - Pikad järjestikku hõivatud alad.
  - Hõivatud alad kasvavad järjest pikemaks, kuna alale pikkusega i järgnev koht hõivatakse tõenäosusega $(i+1)/m$.
  - Operatsioonide keskmine aeg kasvab.

Ruutsondeerimine (*Quadratic Probing*)

$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \quad c_1 \neq c_2$

võti sondeerimise number algne hash funktsioon

- Esmane positsioon on $T[h'(k)]$, järgnevad positsiooni sõltuvad i^2-st.
  - Kõige lihtsamal juhul $c_1 = 0$, $c_2 = 1$ on hüpete suuruseks 1, 4, 9, 16 jne.
- c_1, c_2, and m peavad olema hästi valitud, et hüpped annaksid kõikvõimalikud positsioonid $\langle 0, 1, \ldots, m-1 \rangle$.
- Puuduseks ***sekundaarne klasterdumine***:

  Kui kahel võtmel on sama algpositsioon, siis nende järgnevad sondeerimise positsioonid langevad ka kokku.

Topelthash (*Double Hashing*)

$$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$$

võti sondeerimise nuber hash funktsioonid

- Kaks erinevat hash funktsiooni.
  - h_1 annab algpositsiooni. h_2 annab hüppe pikkuse.
  - h_2 ei kasutata h_1 asemel vaid koos h_1-ga!
- $h_2(k)$ ja m ei tohi omada ühistegureid, et sondeerimisjärjestus oleks permutatsioon $\langle 0, 1, \ldots, m-1 \rangle$.
  - m on 2 aste ja $h_2(k)$ annab alati paaritu numbri.
  - m on algarv ja $1 < h_2(k) < m$.
- $\Theta(m^2)$ erinevat sondeerimisjärjekorda.
  - Üks iga $h_1(k)$ ja $h_2(k)$ kombinatsiooni kohta.
  - Lähedane ideaalsele ühtlasele hash-ile.

Lisamise ja otsimise operatsioon

- Lisamine on nagu otsimine, ainult NIL-I korral omistatakse.

Hash-Insert(T, k)

1. $i \leftarrow 0$
2. **repeat** $j \leftarrow h(k, i)$
3. **if** $T[j]$ = NIL
4. **then** $T[j] \leftarrow k$
5. **return** j
6. **else** $i \leftarrow i + 1$
7. **until** $i = m$
8. **error** “hash table overflow”

Hash-Search (T, k)

1. $i \leftarrow 0$
2. **repeat** $j \leftarrow h(k, i)$
3. **if** $T[j] = k$
4. **then return** j
5. $i \leftarrow i + 1$
6. **until** $T[j]$ = NIL **or** $i = m$
7. **return** NIL

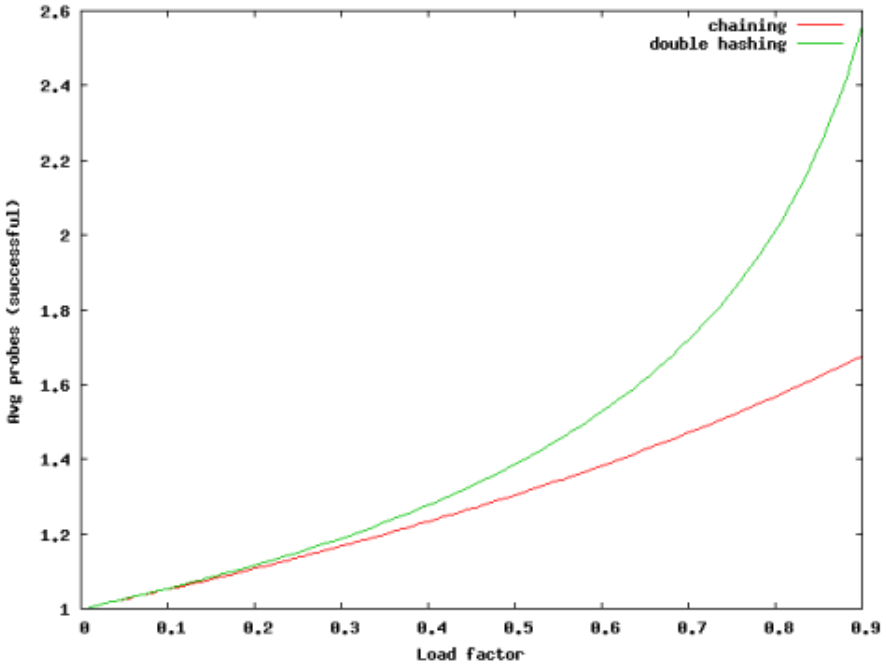
Kustutamine

- Kustutamisel ei saa kirjet asendada lihtsal NIL-iga. Miks?
- Kasutatakse spetsiaalset märgendit DEL NIL-i asemel.
  - ***Search*** peab võtma DEL kirjet nagu võtit, mis ei vasta otsitavale võtmele.
  - ***Insert*** peab võtma DEL kirjet nagu tühja kohta, mida saab uuesti kasutada.
- **Puudus:** Otsingu aeg ei sõltu enam ainult täituvuse faktorist α.
  - Keerukus kasvab, kui on palju kustutamisi

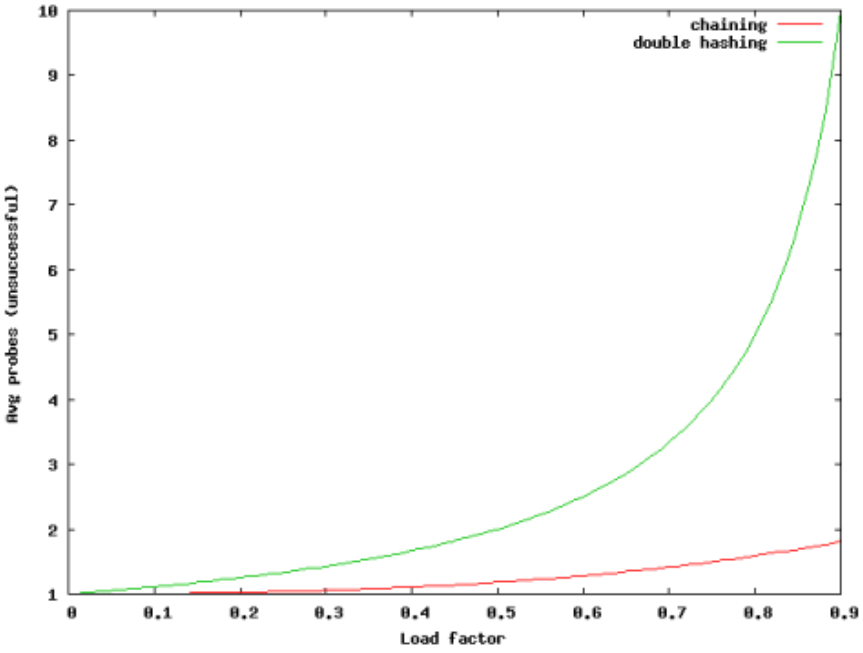
Paisksalvestuse keerukus

- Saame analüüsida keskmist juhtu
  - Halvimal juhul (kõik võtmed on sama hash väärtusega) on kõik operatsioonid lineaarsed
  - Head hash funktsioonid tagavad keskmise keerukuse
- Sondeerimiste arv sõltub täituvuse faktorist (tähistatakse α) mis näitab tabelis olevate võtmete arvu ja tabeli suuruse suhet
- Õnnestunud ja ebaõnnestunud otsinguid tuleb käsitleda eraldi ja eeldada ebaõnnestunud päringute suhet kogu päringute arvusse
- Aheldatud hash tabelis on ebaõnnestunud otsingu keskmine sondeerimiste arv α ja õnnestunud sondeerimiste arv 1 + α /2

Sondeerimiste arv sõltuvalt α-st



Õnnestunud päring



Ebaõnnestunud päring

Rehashing: tabeli suurendamine

- Suurendamiseks:
  - Loo 2 korda suurem massiiv (+ suurenda algarvuni)
  - Kanna andmed ühekaupa üle kasutades Insert() funktsiooni
- Millal suurendada?
  - Kui avatud adresseerimisel on täituvus ≈ 0.7
  - Kui aheldamisel on täituvus ≈1
  - Ruutsondeerimisel, kui tabeli täituvus on 0.5 või kui Insert ei õnnestu
- Miks suurendada 2 korda?
  - Argument sarnane dünaamilisele massiivile
  - Iga Insert(x)-I kohta ei tehta rohkem kui 3 hash-i: x algne insert, x rehash ja massivis enne olnud n/2 liikme rehash

Paisksalvestuse rakendusi

- Kompilaatorid tuvastavad märksõnu ja muutujaid
- On-line õigekirjakontroll — iga sõna õigekirja saab kontrollida konstantse ajaga, kui kogu sõnastik on hashitud.
- Mängu ja otsingualgoritmid salvestavad läbitud seisude hash-I, et teada kas seis on läbitud ja käituda vastavalt
- Kiire objektide/kirjete mittevastavuse kontroll – kui hash on erinev, siis on ka objektid erinevad (aga mitte tingimata vastupidi)
- Väga hõredate andmete hoidmine
  - Võtmete väärtused on väga suurest vahemikust

Millal on teised esitused paremad?

Otsingupuud võivad olla efektiivsemad kui

- on vaja itereerida üle andmete
- on vaja sorteeritust – min, max, suuruselt järgmine
- andmed on väga dünaamilised - on palju lisamisi ja kustutamisi
- Andmeid on rohkem kui mahub operatiivmällu. Siis võib kasutada B-puid

Võrdlus otsingupuudega

- Ei sobi, kui on vaja sorteeritust
  - **FindMax**: O(n) O(log n) Balanced binary tree
  - **FindMin**: O(n) O(log n) Balanced binary tree
  - **PrintSorted**: O(n log n) O(n) Balanced binary tree
- Efektiivne muutmisel ja otsingul
  - **Insert**: O(1) O(log n) Balanced binary tree
  - **Delete**: O(1) O(log n) Balanced binary tree
  - **Find**: O(1) O(log n) Balanced binary tree

Kokkuvõtteks *dictionary* implementatsioonist

- Hash tabelil põhinevad andmestruktuurid – O(1)
  - puudub järjestus
- Puudel põhinevad andmestruktuurid – O(log *n*)
  - võrdlusoperatsiooni põhine järjestus
- Massiiv, dünaamiline massiiv, lingitud list
  - mõned lihtsamad operatsioonid O(1)
  - teised operatsioonid O(*n*)

Oluline on osata leida sobiv andmestruktuur