AVL

AVL (veți avea la examen)

- Video (MIT ultimele 8 minute).
- Lecture Notes

SKIP LISTS

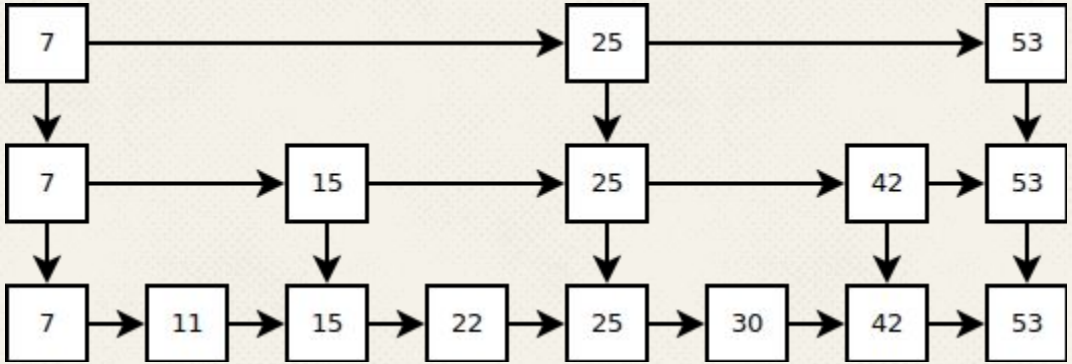
Skip Lists

- Sunt structuri de date echilibrate
- Alte structuri de date eficiente (**log n** sau mai bun):
  - Tabele de dispersie (hash tables) - nu sunt sortate
  - Heap-uri - nu putem căuta în ei
  - Arbori binari echilibrați (AVL, Red Black Trees)

- Ajută la o căutare rapidă
- Elementele sunt sortate!

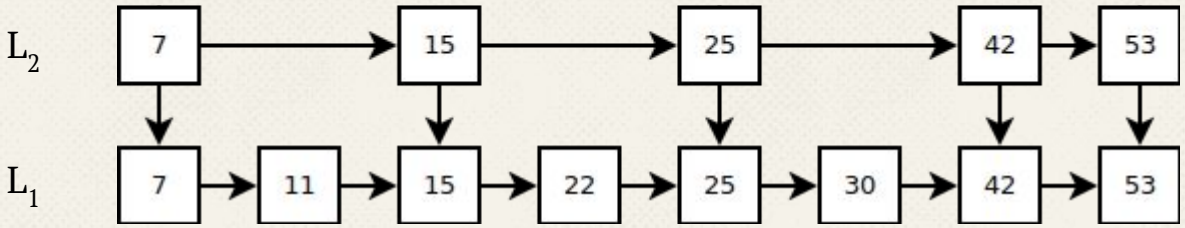
Skip Lists

- Sunt implementate ca liste înlănțuite
- Ideea de implementare:
  - este extinsă pe mai multe nivele (mai multe liste înlănțuite)
  - la fiecare nivel adăugat, **sărim peste o serie de elemente** față de nivelul anterior
  - nivelele au legături între ele



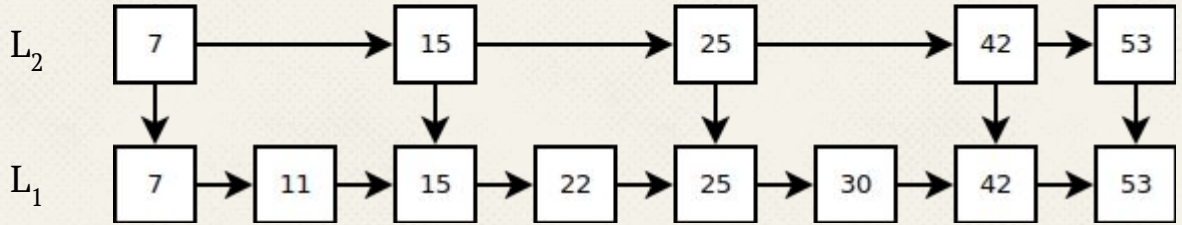
Skip Lists

- Să presupunem că avem doar 2 liste
  - Cum putem alege ce elemente ar trebui transferate în nivelul următor?



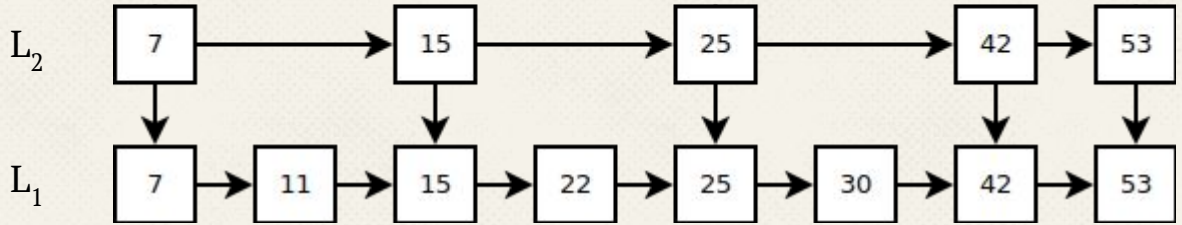
Skip Lists - 2 liste

- Cum putem alege ce elemente ar trebui transferate în nivelul următor?
  - Cea mai bună metodă: elemente egal depărtate
  - Costul căutării = $|L_2| + (|L_1| / |L_2|) = |L_2| + (n / |L_2|)$
  - Când este minim acest cost?



Skip Lists - 2 liste

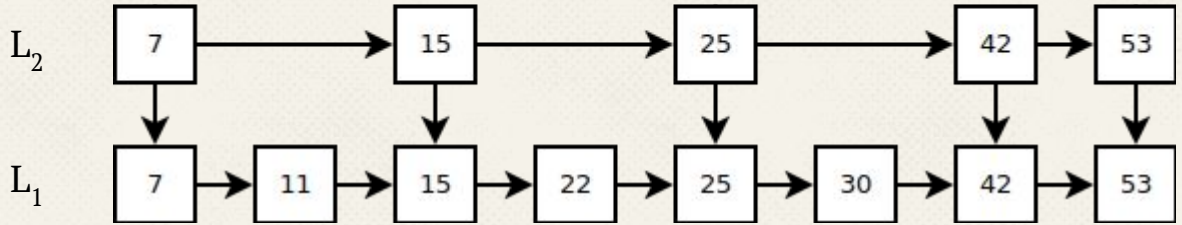
- Cum putem alege ce elemente ar trebui transferate în nivelul următor?
  - Cea mai bună metodă: elemente egal depărtate
  - Costul căutării = $|L_2| + (|L_1| / |L_2|) = |L_2| + (n / |L_2|)$
  - Când este minim acest cost?
    - Când $|L_2| = n / |L_2| \quad \Rightarrow \quad \mathbf{|L_2| = sqrt(n)}$



Skip Lists - 2 liste

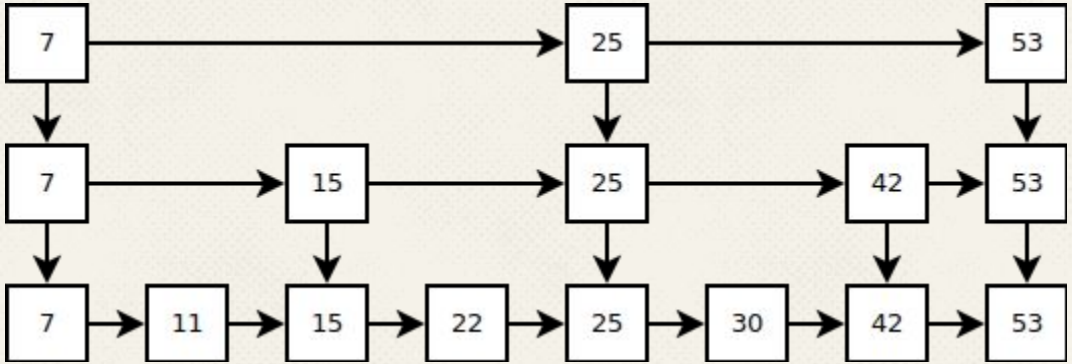
- Cum putem alege ce elemente ar trebui transferate în nivelul următor?
  - Cea mai bună metodă: elemente egal depărtate
  - Costul căutării = $|L_2| + (|L_1| / |L_2|) = |L_2| + (n / |L_2|)$
  - Când este minim acest cost?
    - Când $|L_2| = n / |L_2| \quad \Rightarrow \quad \mathbf{|L_2| = sqrt(n)}$
  - Deci, costul minim pentru căutare este sqrt(n) + n / sqrt(n) = 2*sqrt(n)

  - Complexitate: O(sqrt(n)) → seamănă un pic cu **Batog**



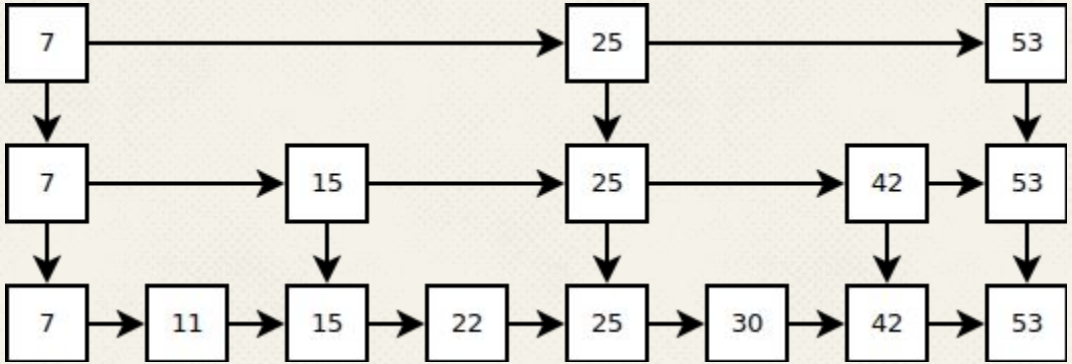
Skip Lists

- Ce se întâmplă când avem mai mult de 2 liste înlănțuite?



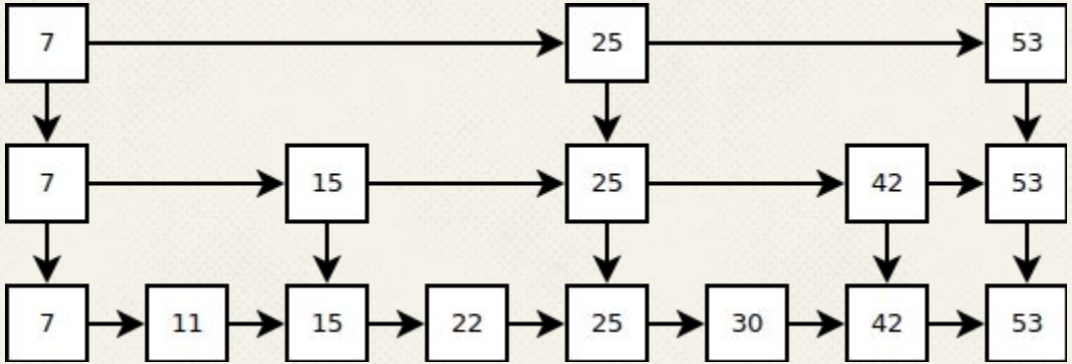
Skip Lists

- Ce se întâmplă când avem mai mult de 2 liste înlănțuite?
  - Costul căutării se modifică
  - 2 liste: $2 * \sqrt{n}$
  - 3 liste: ?



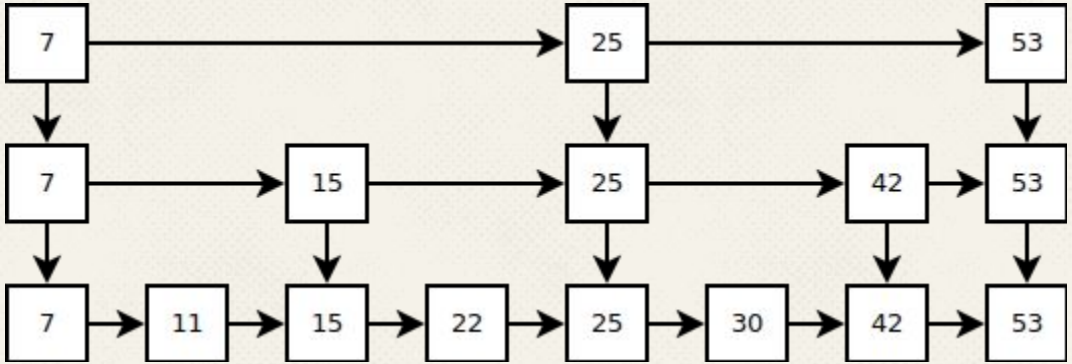
Skip Lists

- Ce se întâmplă când avem mai mult de 2 liste înlănțuite?
  - Costul căutării se modifică
  - 2 liste: $2 * \sqrt{n}$
  - 3 liste: $3 * \sqrt[3]{n}$
  - k liste: $k * \sqrt[k]{n}$



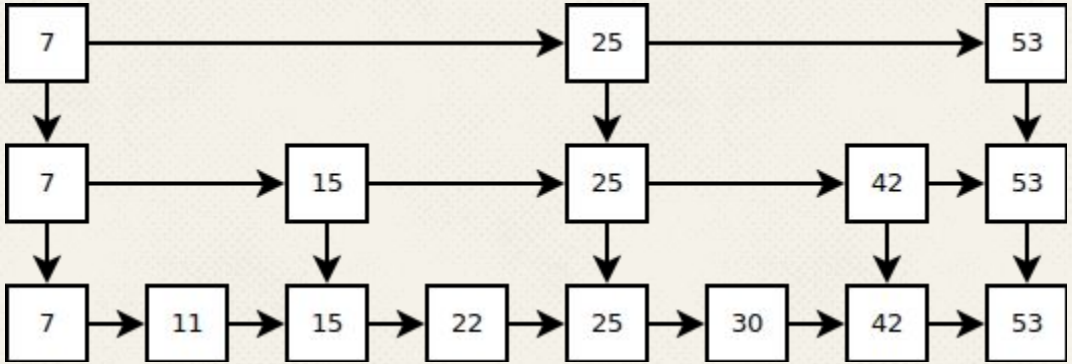
Skip Lists

- Ce se întâmplă când avem mai mult de 2 liste înlănțuite?
  - Costul căutării se modifică
  - 2 liste: $2 * \sqrt{n}$
  - 3 liste: $3 * \sqrt[3]{n}$
  - k liste: $k * \sqrt[k]{n}$
  - logn liste: $logn * \sqrt[logn]{n}$



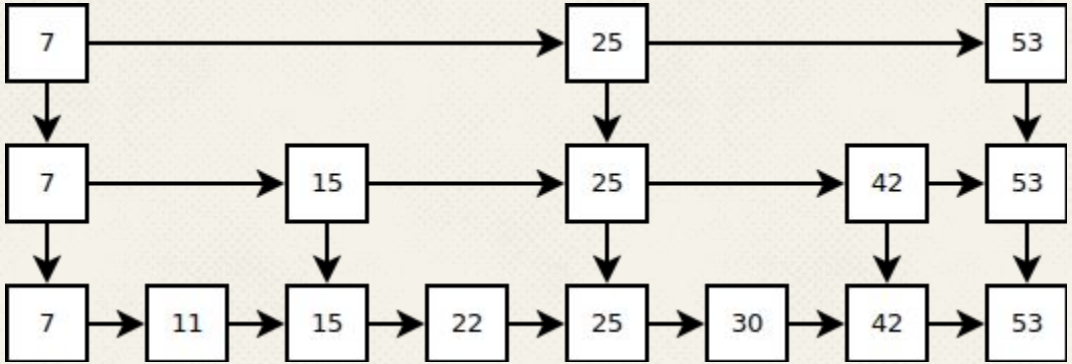
Skip Lists

- Ce se întâmplă când avem mai mult de 2 liste înlănțuite?
  - Costul căutării se modifică
  - 2 liste: $2 * \sqrt{n}$
  - 3 liste: $3 * \sqrt[3]{n}$
  - k liste: $k * \sqrt[k]{n}$
  - logn liste: $logn * \sqrt[logn]{n}$ = ? Cu cât este egal $\sqrt[logn]{n}$?



Skip Lists

- Ce se întâmplă când avem mai mult de 2 liste înlănțuite?
  - Costul căutării se modifică
  - 2 liste: $2 * \sqrt{n}$
  - 3 liste: $3 * \sqrt[3]{n}$
  - k liste: $k * \sqrt[k]{n}$
  - logn liste: $logn * \sqrt[logn]{n} = 2 * logn$ $\Rightarrow$ Complexitate: **O(logn)** !



Skip Lists - Căutare

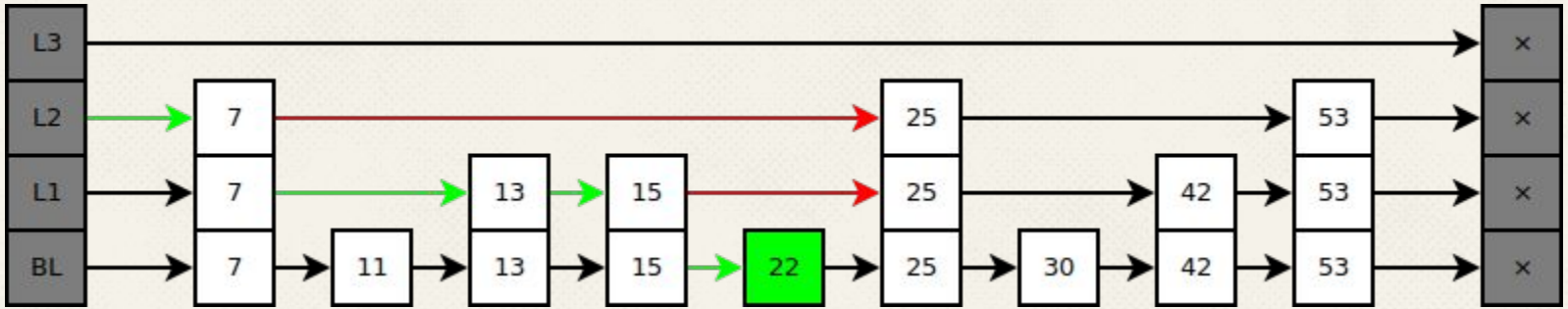
1) Începem căutarea cu primul nivel (cel mai de sus)
2) Avansăm în dreapta, până când, dacă am mai avansa, am merge prea departe (adică elementul următor este prea mare)
3) Ne mutăm în următoarea listă (mergem în jos)
4) Reluăm algoritmul de la pasul 2)

Skip Lists - Căutare

1) Începem căutarea cu primul nivel (cel mai de sus)
2) Avansăm în dreapta, până când, dacă am mai avansa, am merge prea departe (adică elementul următor este prea mare)
3) Ne mutăm în următoarea listă (mergem în jos)
4) Reluăm algoritmul de la pasul 2)

Exemplu: search(22)

Complexitate: O(logn)



Skip Lists - Inserare

- Vrem să inserăm elementul x
- **Observație:** Lista de jos trebuie să conțină toate elementele!

- x trebuie să fie inserat cu siguranță în nivelul cel mai de jos
  - căutăm locul lui x în lista de jos → search(x)
  - adăugăm x în locul găsit în lista cea mai de jos
- Cum alegem în câte liste să fie adăugat?

Skip Lists - Inserare

- Vrem să inserăm elementul x
- x trebuie să fie inserat cu siguranță în nivelul cel mai de jos
- Cum alegem în ce altă listă să fie adăugat?
  - Alegem metoda probabilistică:
    - aruncăm o monedă
      - dacă pică Stema - o adăugăm în lista următoare și aruncăm din nou moneda
      - dacă pică Banul - ne oprim
    - probabilitatea să fie inserat și la nivelul următor: ½
- În medie:
  - ½ elemente nepromovate
  - ¼ elemente promovate 1 nivel
  - ⅛ elemente promovate 2 nivele
  - etc.
- Complexitate: O(logn)

Skip Lists - Ștergere

- Ștergem elementul x din toate listele care îl conțin
- Complexitate: O(logn)

Skip Lists

- Articol
- Video MIT
- Notes

Bibliografie

http://ticki.github.io/blog/skip-lists-done-right/

https://www.guru99.com/avl-tree.html

https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/

MIT lecture notes on skip lists

Esoteric Data Structures: Skip Lists and Bloom Filters - Stanford University

???

```
sol = 0;
for (t = 1 << 30; t > 0; t>>=1) {
    if (sol + t < v.size() && v[sol + t] <= x)
        sol += t;
}
```

???

```
sol = 0;
for (t = 1 << 30; t > 0; t>>=1) {
    if (sol + t < v.size() && v[sol + t] <= x)
        sol += t;
}
```

x= 32

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	7	11	20	24	28	30	32	44	49	62	68	82	84	93	97

???

```
sol = 0; x = 32;
for (t = 1 << 30; t > 0; t>>=1) {
    if (sol + t < v.size() && v[sol + t] <= x)
        sol += t;
}
```

0	1	2	3	**4**	5	**6**	**7**	8	9	10	11	12	13	14	15
3	7	11	20	24	28	30	**32**	44	49	62	68	82	84	93	97

Căutare binară

```
sol = 0; x = 32;
for (t = 1 << 30; t > 0; t>>=1) {
    if (sol + t < v.size() && v[sol + t] <= x)
        sol += t;
}
```

Complexitate?

0	1	2	3	**4**	5	**6**	**7**	8	9	10	11	12	13	14	15
3	7	11	20	24	28	30	**32**	44	49	62	68	82	84	93	97

Căutare binară

```
sol = 0; x = 32;
for (t = 1 << 30; t > 0; t>>=1) {
    if (sol + t < v.size() && v[sol + t] <= x)
        sol += t;
}
```

Complexitate **O(log n)** - recomand cu căldură :)

0	1	2	3	**4**	5	**6**	**7**	8	9	10	11	12	13	14	15
3	7	11	20	24	28	30	**32**	44	49	62	68	82	84	93	97

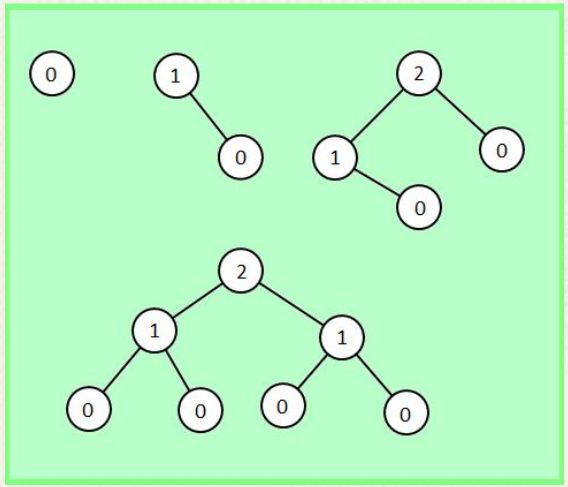
B-Arbori

Arbori echilibrați

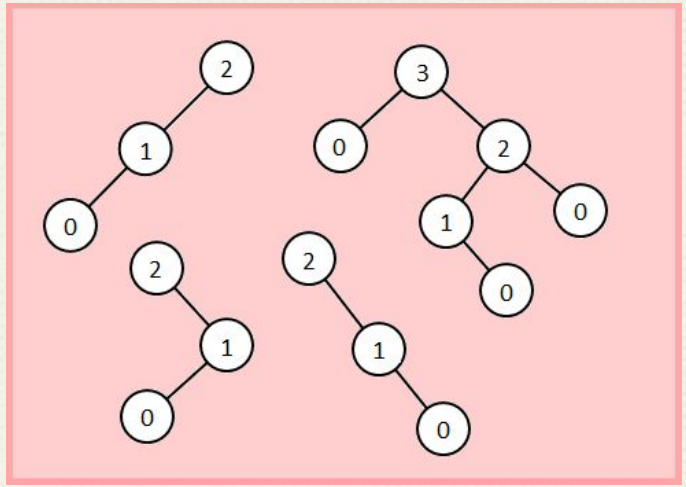
Un **arbore echilibrat** este un arbore în care, **pentru orice nod**, diferența dintre înălțimile subarborilor stâng și drept este de maxim 1.

Exemple:

Echilibrați



Neechilibrați

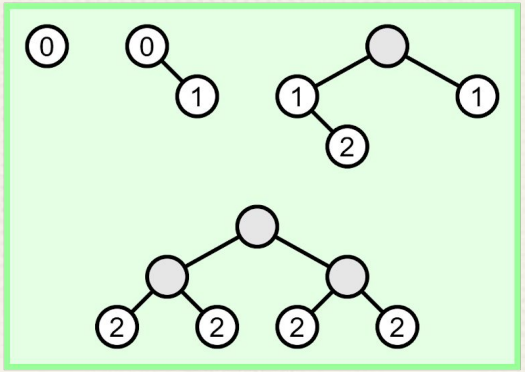


Arbori echilibrați

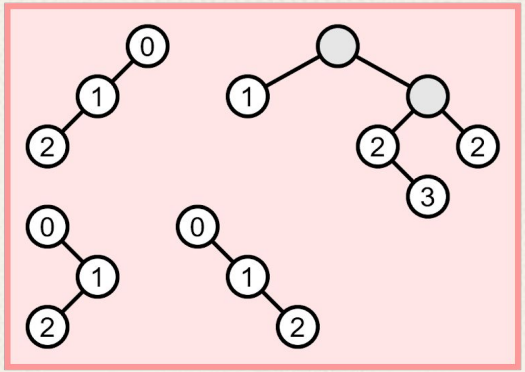
Un **arbore echilibrat** este un arbore în care, **pentru orice nod**, diferența dintre înălțimile subarborilor stâng și drept este de maxim 1.

Exemple:

Echilibrați



Neechilibrați



B-Arbori

Un **B-Arbore** este un arbore echilibrat, destinat căutării eficiente de informație.

Un B-Arbore poate avea mai mult de 2 fii pentru un nod (se poate ajunge și la ordinul sutelor).

Totuși, înălțimea arborelui rămâne O(log n), datorită unei baze a logaritmului convenabilă.

În practică, B-Arborii sunt folosiți pentru baze de date și sisteme de fișiere, pentru citirea și scrierea eficientă pe discul de memorie.
O proprietate importantă a lor este faptul că rețin multă informație. De aceea, B-Arborii reduc numărul de accesări ale discului (accesarea discului este o operație costisitoare).

B-Arbori

Un **B-Arbore** este un arbore echilibrat, destinat căutării eficiente de informație.

Un B-Arbore poate avea mai mult de 2 fii pentru un nod (se poate ajunge și la ordinul sutelor).

Totuși, înălțimea arborelui rămâne O(log n), datorită unei baze a logaritmului convenabilă.

Proprietăți:

1. Un nod poate să conțină mai mult de o cheie
2. Numărul de chei ale unui nod este
3. Un nod are fii
4. Toate frunzele unui B-Arbore se află pe același nivel

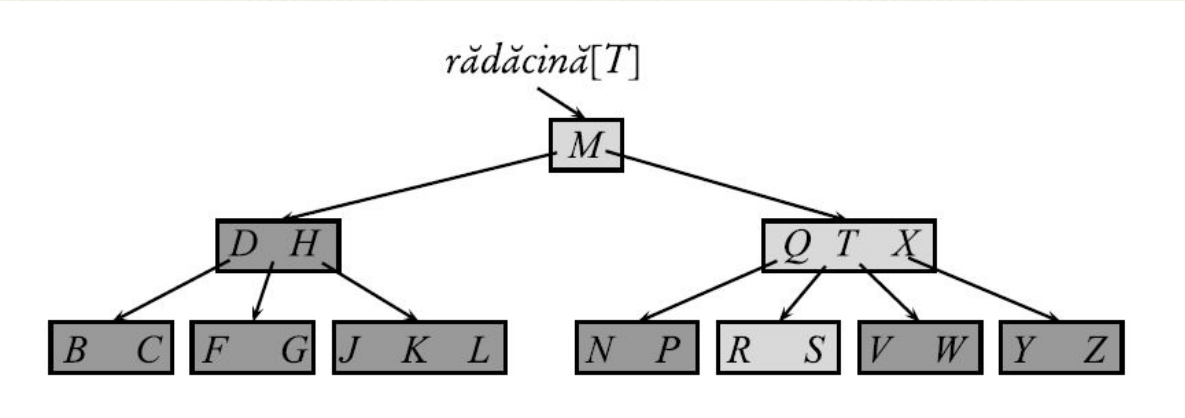
B-Arbori

Proprietăți:

1. Un nod poate să conțină mai mult de o cheie
2. Numărul de chei ale unui nod este
3. Un nod are fii
4. Toate frunzele unui B-Arbore se află pe același nivel

B-Arbori

Exemplu:



Cheile acestui arbore sunt consoanele din alfabetul latin.
Observăm că un nod poate avea mai multe chei, iar fiecare nod care are valori va avea fii.

Căutarea literei ”R” este exemplificată pe traseul hașurat cu o culoare mai deschisă.

B-Arbori

Câmpurile unui nod:

1. – numărul de chei memorate în nodul
2. cele chei, memorate în ordine crescătoare:
$$cheie_1[x] \leq cheie_2[x] \leq cheie_3[x] \leq ... \leq cheie_{n[x]}[x]$$
3. o valoare booleană – **True**, dacă nodul este frunză, **False**, dacă nodul este nod intern

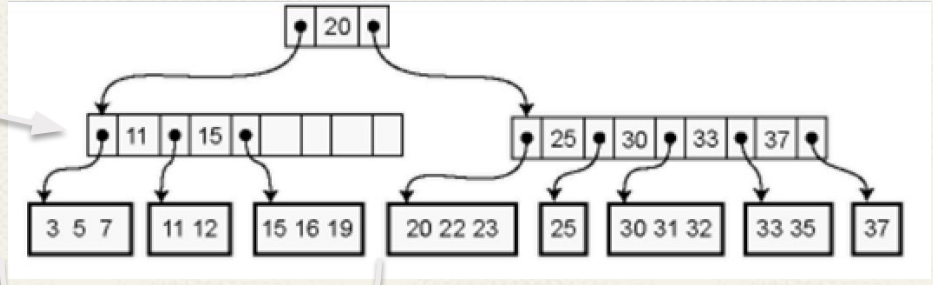
Dacă este un nod intern, atunci el conține pointeri către fiii săi. Nodurile frunză nu au fii, deci nu au aceste câmpuri definite.

B-Arbori

Cheile nodului separă domeniile de chei aflate în fiecare subarbore astfel:

- dacă este o cheie oarecare memorată într-un subarbore cu rădăcina , atunci
  $k_1 \le cheie_1[x] \le k_2 \le cheie_2[x] \le ... \le cheie_{n[x]}[x] \le k_{n[x]+1}$

Nodul x



$fiu_1[x], ..., fiu_{n[x]+1}[x]$

3, 5, 7 ≤ 11
11 ≤ 11, 12
11, 12 ≤ 15
15 ≤ 15, 16, 19

B-Arbori

Gradul unui B-Arbore

Există o limitare inferioară și una superioară a numărului de chei ce pot fi conținute într-un nod. Exprimăm aceste margini printr-un întreg fixat $t \geq 2$, numit al B-Arborelui.

B-Arbori

Restricții:

1. Fiecare nod, cu excepția rădăcinii, trebuie să aibă .
   Consecință: fiecare nod intern trebuie să aibă **cel puțin *t*** fii.

2. Dacă arborele este nevid, atunci rădăcina trebuie să aibă cel puțin o cheie.

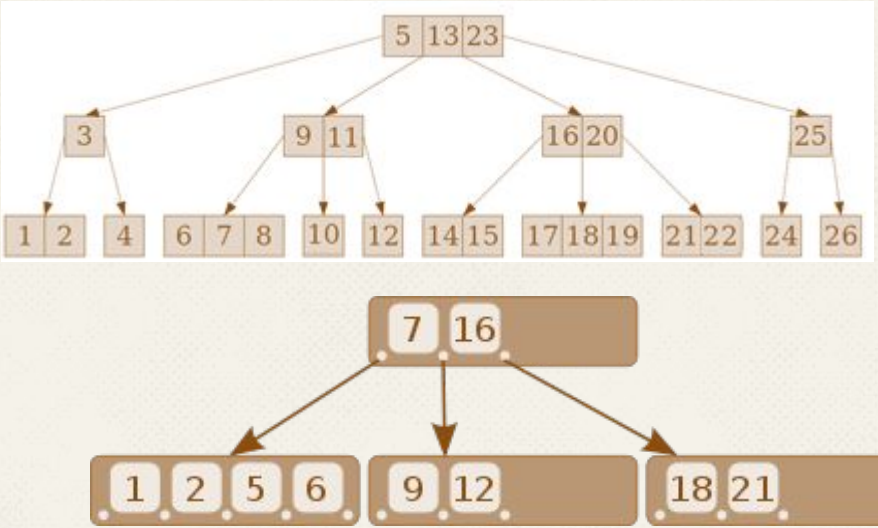
3. Fiecare nod poate să aibă .
   Consecință: orice nod intern poate să aibă **cel mult *2t*** fii.

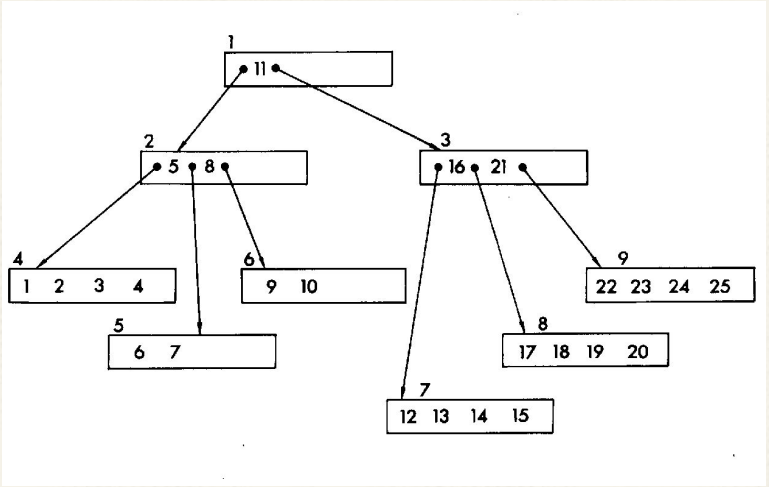
Un nod cu *2t – 1* chei se numește .

B-Arbori

Exemple de B-Arbori:

B-Arbore de ordin 2 (arbore 2-3-4)





Înălțimea unui B-Arbore

Teoremă: Dacă n ≥ 1, atunci, pentru orice B-Arbore T cu **n** chei și grad minim *t ≥ 2*:

$$h \leq \log_t \frac{n+1}{2}$$ - înălțimea arborelui

Demonstrație: Dacă un B-Arbore are înălțimea h, atunci va avea număr minim de chei dacă rădăcina conține o singură cheie, iar toate celelalte noduri câte t – 1 chei.
În acest caz, există:

- pe nivelul 1: 2 noduri
- pe nivelul 2: 2t noduri
- pe nivelul 3: $2t^2$ noduri
- ...
- pe nivelul h: $2t^{h-1}$

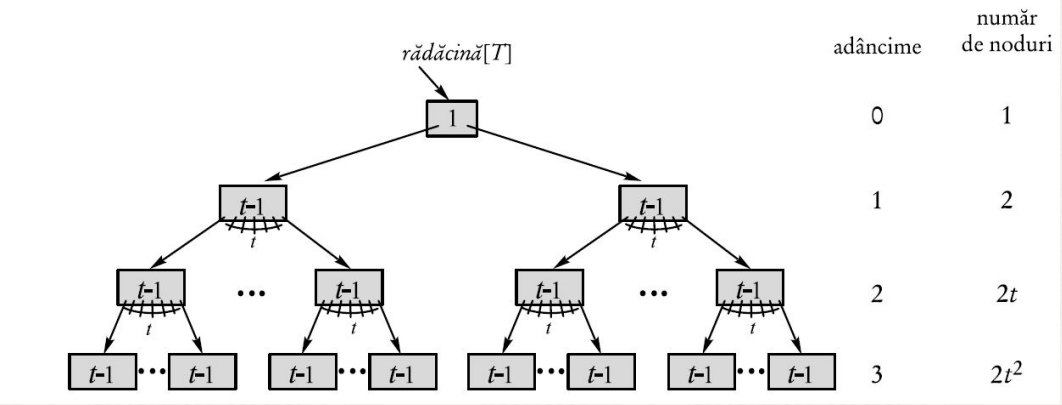
Deci:

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) = 2t^h - 1$$

Înălțimea unui B-Arbore

Exemplu:

Pentru un B-Arbore de înălțime 3, cu număr minim de chei, care are, în fiecare nod, numărul de chei reținute , avem următorul desen:



Înălțimea unui B-Arbore

1. **Concluzie:**

Înălțimea unui arbore este

$$h \leq \log_t \frac{n+1}{2}$$

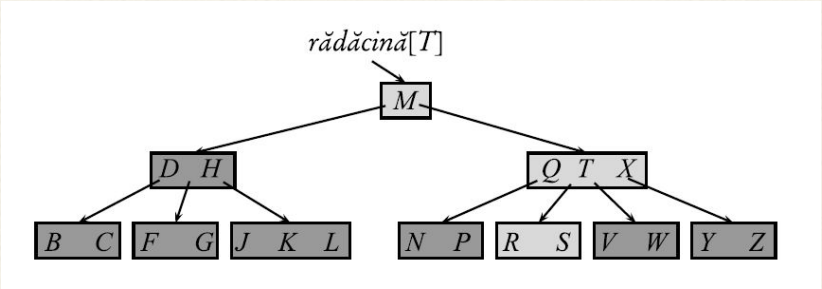
Deci, înălțimea unui B-Arbore crește proporțional cu **O(log n)**.

Discuție

Exerciții:

1. De ce nu putem permite gradul minim $t = 1$?

2. Pentru ce valori ale lui t, arborele de mai jos este un B-Arbore, conform definiției?



3. Desenați toti B-Arborii corecți cu grad minim 2 care să reprezinte mulțimea {1, 2, 3, 4, 5}.

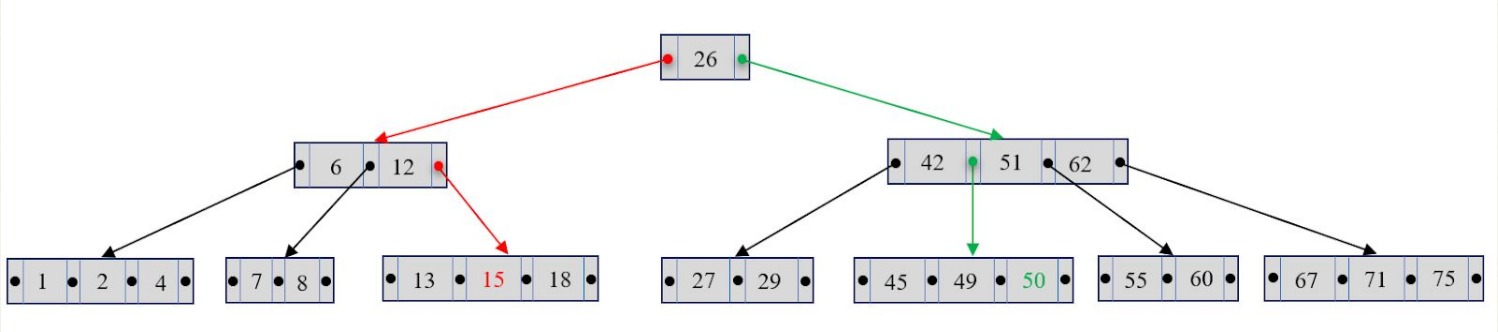
Operații de bază

Căutarea în B-Arbore

Căutarea într-un B-Arbore este asemănătoare cu o căutare într-un arbore binar.

Într-un B-Arbore, căutarea se realizează comparând cheia căutată cu cheile nodului curent, plecând de la nodul rădăcină.

Căutare reușită pentru 50 și nereușită pentru 17.

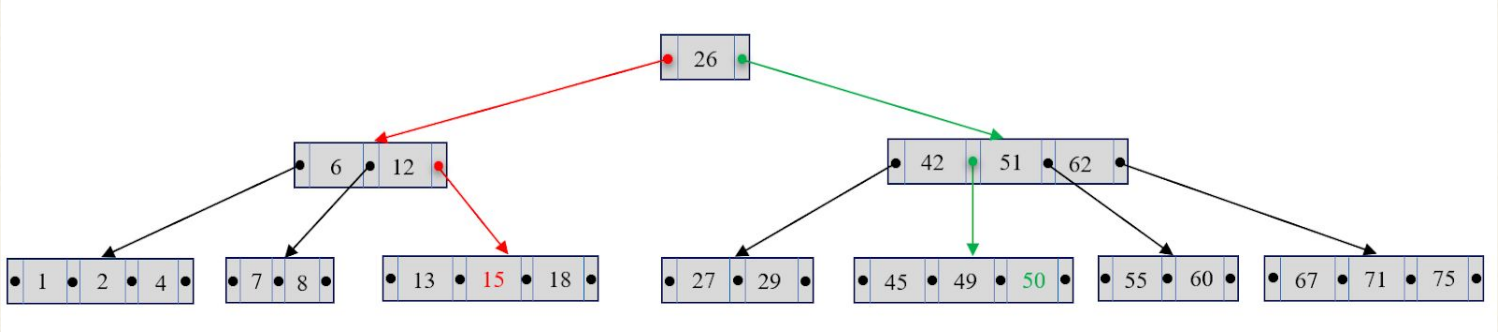


Căutarea în B-Arbore

Algoritm:

1. Căutăm cheia în rădăcină
2. Dacă nu o găsim, atunci continuăm căutarea în fiul corespunzător valorii
3. Dacă găsim cheia, returnăm perechea de valori (y, i), reprezentând nodul, respectiv poziția în nod pe care s-a găsit valoarea .

Putem afla indicele fiului care trebuie explorat în continuare la pasul 2 folosind **căutarea binară dacă numărul de valori din fiecare nod este mare**.



Căutarea în B-Arbore

Căutarea într-un B-Arbore este asemănătoare cu o căutare într-un arbore binar.

Într-un B-Arbore, căutarea se realizează comparând cheia căutată cu cheile nodului curent, plecând de la nodul rădăcină.

Algoritm:

1. Căutăm cheia în rădăcină
2. Dacă nu o găsim, atunci continuăm căutarea în fiul corespunzător valorii
3. Dacă găsim cheia, returnăm perechea de valori (y, i), reprezentând nodul, respectiv poziția în nod pe care s-a găsit valoarea .

Putem afla indicele fiului care trebuie explorat în continuare la pasul 2 folosind căutarea binară.

Căutarea în B-Arbore

Procesul se repetă de cel mult ori, în cazul în care valoarea căutată se află într-o frunză.

Căutarea valorii într-un nod se realizează (folosind căutarea binară) în .

Complexitate finală:
$O(h * \log t) = O(\log t * \log_t n)$
$= O(\log t * (\log n / \log t))$
=

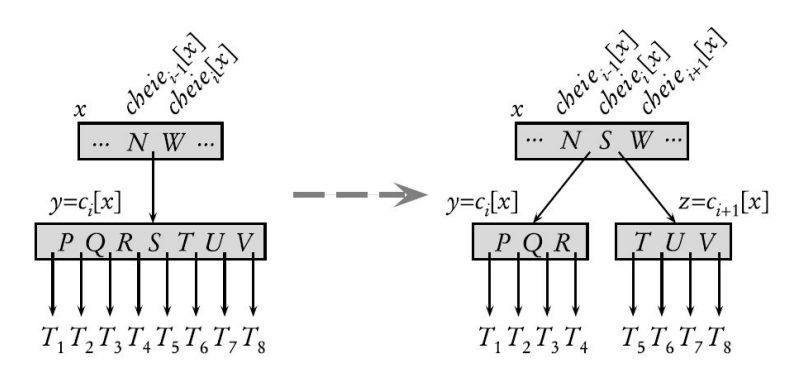
Inserarea în B-Arbore

Pentru a insera o cheie într-un B-Arbore, trebuie distinse două cazuri: când nodul unde trebuie introdus are mai puțin de *2t-1* chei, respectiv când are *2t-1* chei.

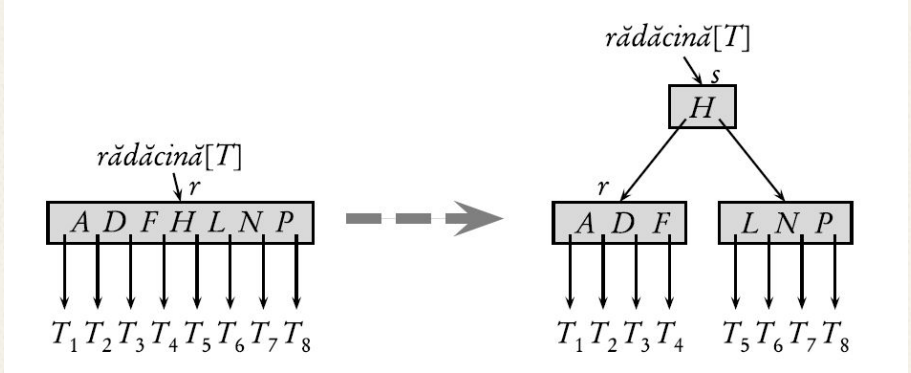
Algoritm:

1. Aplicăm operația de căutare pentru a găsi nodul unde trebuie introdusă cheia. Notăm acest nod cu și va fi o .
2. Dacă are mai puțin de 2t-1 chei, atunci inserarea se efectuează fără a modifica structura arborelui.
3. Dacă are 2t-1 chei, atunci acesta trebuie divizat. Rezultă, astfel, două noduri noi, (fiul din stânga) și (fiul din dreapta).
4. Eliminăm cea mai mare cheie din (cheia mediană). O notăm cu .
5. devine părintele celor două noduri și .
6. Se încearcă (recursiv) adăugarea lui în părintele lui .

Inserarea în B-Arbore



Nod intermediar



Rădăcină

Inserarea în B-Arbore

Căutarea nodului în care trebuie introdusă cheia: **$O(\log_t n)$**
Pentru un nivel: **O(t)**
Recursivitatea: O(h) **= $O(\log_t n)$**

Complexitatea finală: