Structuri de date avansate pentru multimi

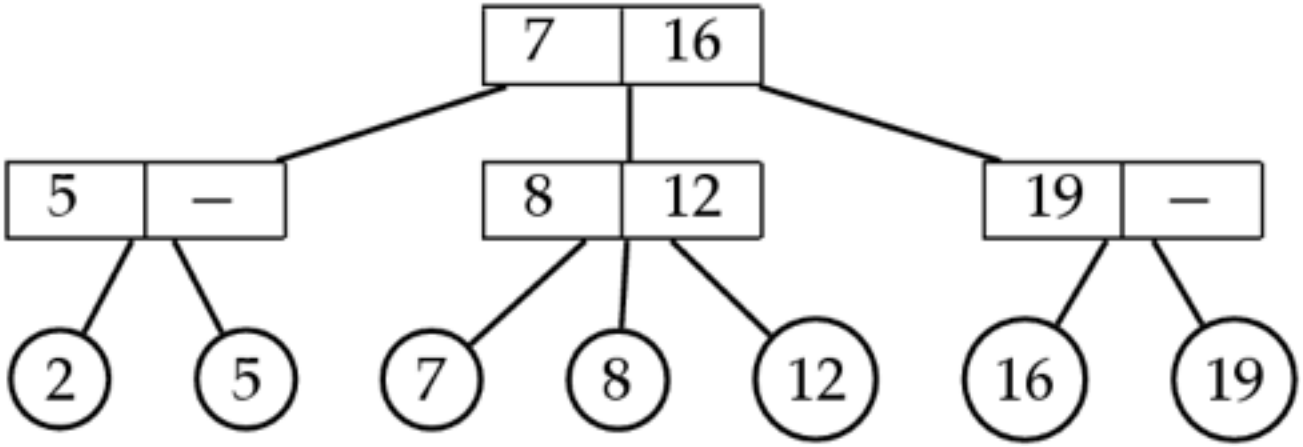
Arbori 2-3. Structuri de date pentru multimi disjuncte

Arbori 2-3

- Proprietati:
  - fiecare nod intern are 2 sau 3 copii (i.e. 1 sau 2 chei)
  - fiecare cale de la radacina la frunza are aceeasi lungime (toate frunzele la acelasi nivel)
  - arbore cu 0/1 noduri - caz special
- *ADT multime* reprezentat ca arbore 2-3:
  - Elementele sunt stocate la frunze
  - Daca elementul **a** se afla la stanga lui **b**, atunci **a<b**
  - Ordonare pe baza de chei
  - In fiecare nod interior, se stocheaza:
    - cheia celui mai mic descendent al celui de-al doilea copil
    - (eventual) cheia celui mai mic descendent al celui de-al treilea copil (daca acest copil exista)

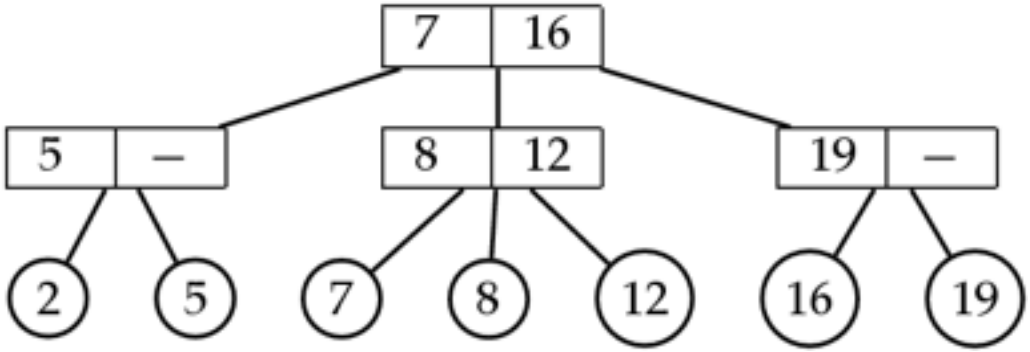
Arbori 2-3

- Un arbore 2-3 cu k nivele are intre 2^{k-1} si 3^{k-1} frunze
  - => pt a reprezenta o multime de n elemente, avem nevoie de cel putin $1+log_3 n$ nivele, dar nu mai mult de $1+log_2 n$
- Costul de mentinere a echilibrului este relativ redus

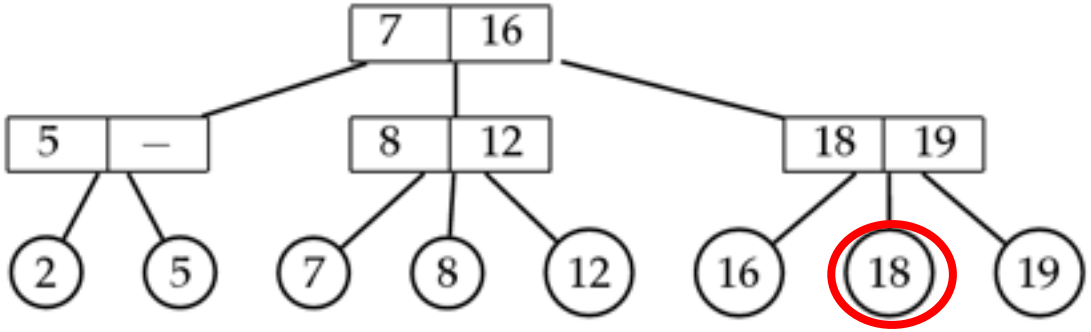


Arbori 2-3: Insert - 2 copii

- Se insereaza ca frunza, respectand conditia de ordine; nu se “adanceste” arborele
- Se updateaza cheile din nodul intern



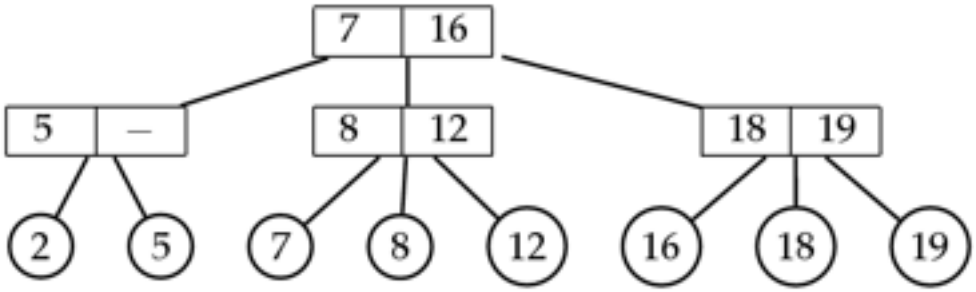
Insert(18)



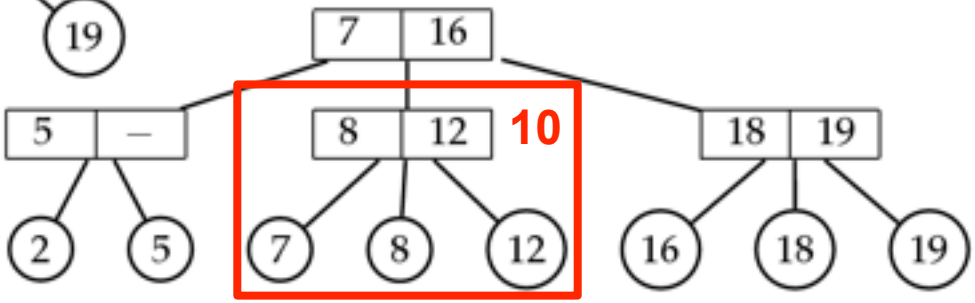
Arbori 2-3: Insert - 3 copii (2 chei)

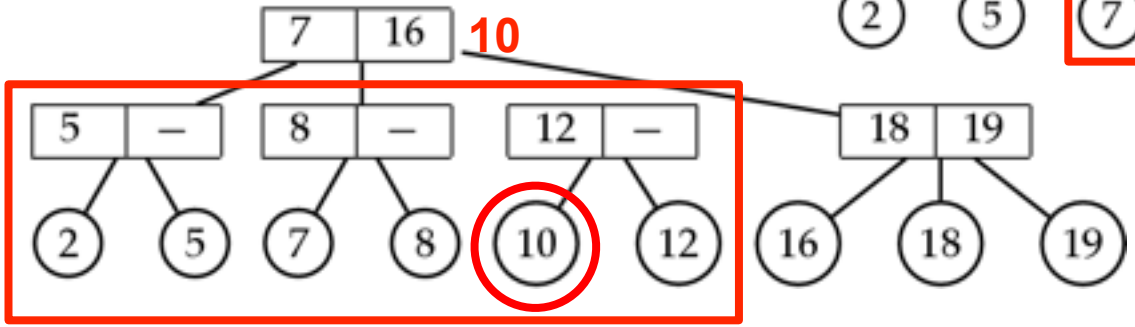
- se imparte parintele in 2 noduri: ***node*** si ***node'***
- 2 cele mai mici elemente din cei 4 copii raman cu ***node*** (i.e. prima cheie)
- celelalte 2 elemente mai mari devin copiii lui ***node'*** (i.e. a doua cheie)
- cheia din mijloc este "promovata" spre nodul parinte, impreuna cu un pointer catre *node'*
- Procesul continua in sus in arbore - i.e. ce se intampla daca parintele avea deja 3 copii?
- caz special cand se sparge radacina

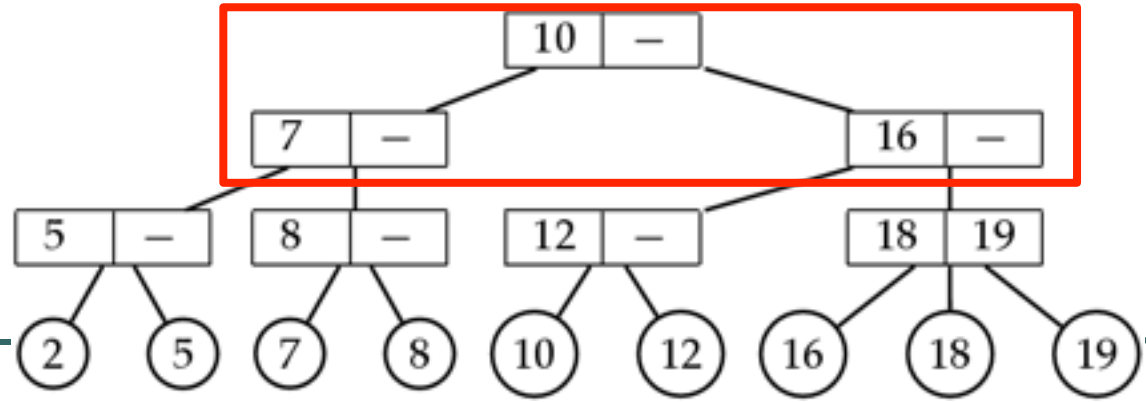
Arbori 2-3: Insert - Radacina



Insert(10)



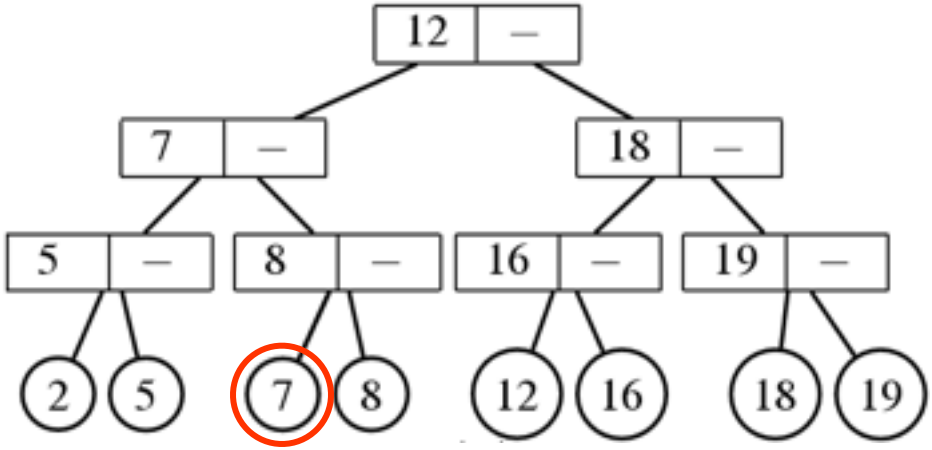




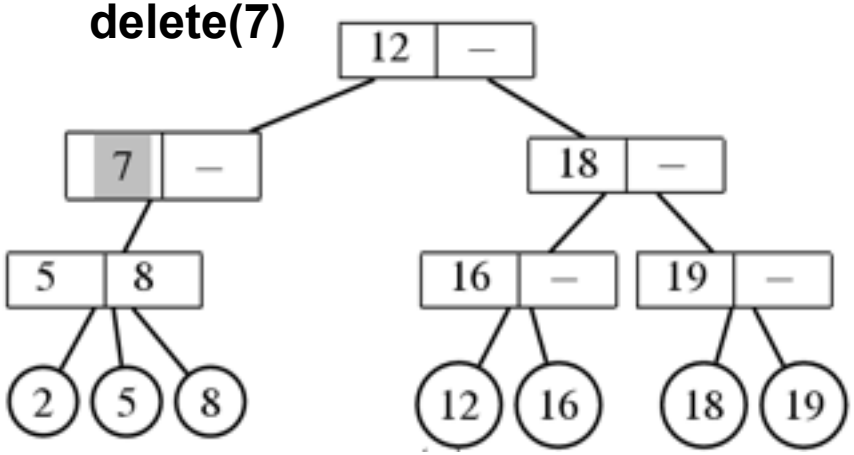
Arbori 2-3: Delete

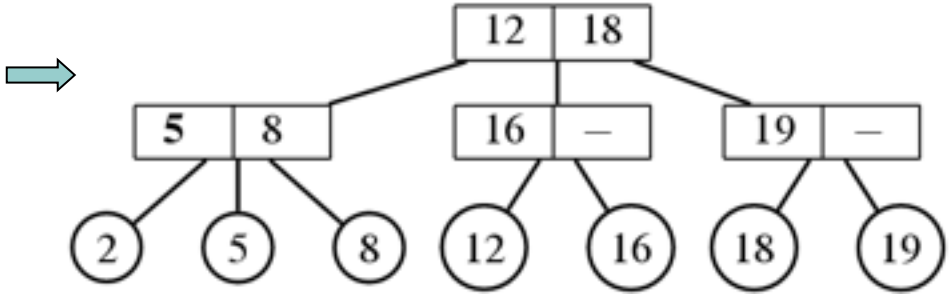
- stergerea unei frunze ar putea lasa parintele cu doar 1 copil
  - daca parintele e radacina: se sterge nodul si copilul ramas devine radacina
  - altfel: fie ***p*** parintele nodului ***node***
    1. daca ***p*** mai are alt copil si acel copil are 3 copii se transfera un copil la ***node***
    2. daca toti copiii lui ***p*** au doar 2 copii, se transfera copilul singur al lui ***node*** unui frate adiacent de-al lui ***node***, si se sterge ***node***
- ***p*** are doar 1 copil, se repeta cele de deasupra recursiv, cu ***p*** in loc de ***node*** (i.e. se repeta de la parinte)

Arbori 2-3: Delete



delete(7)



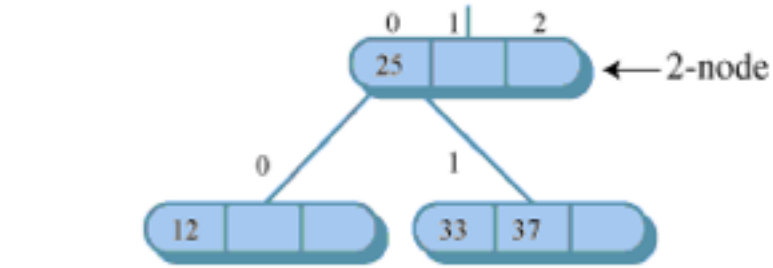


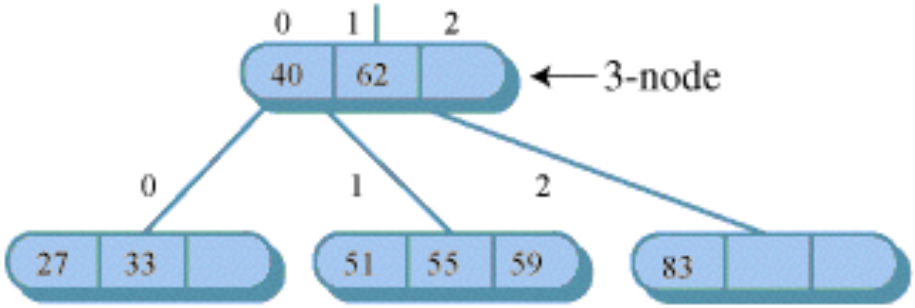
Arbori 2-3: Analiza eficientei

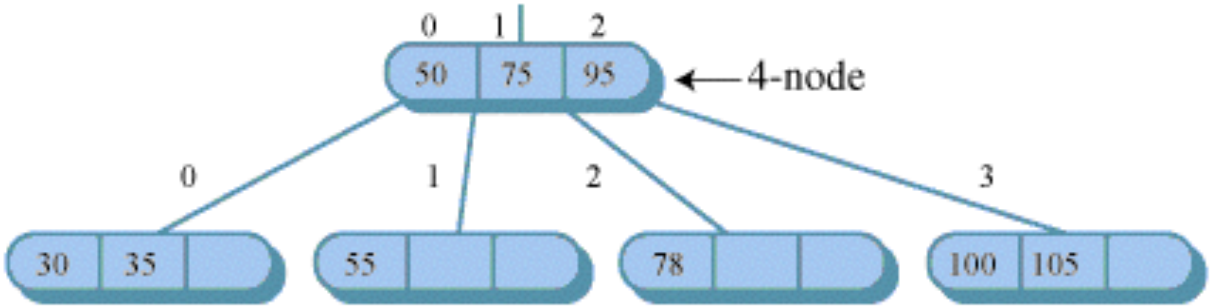
- Search
- Insert
- Delete
- *O(logn)*

Arbori 2-3-4

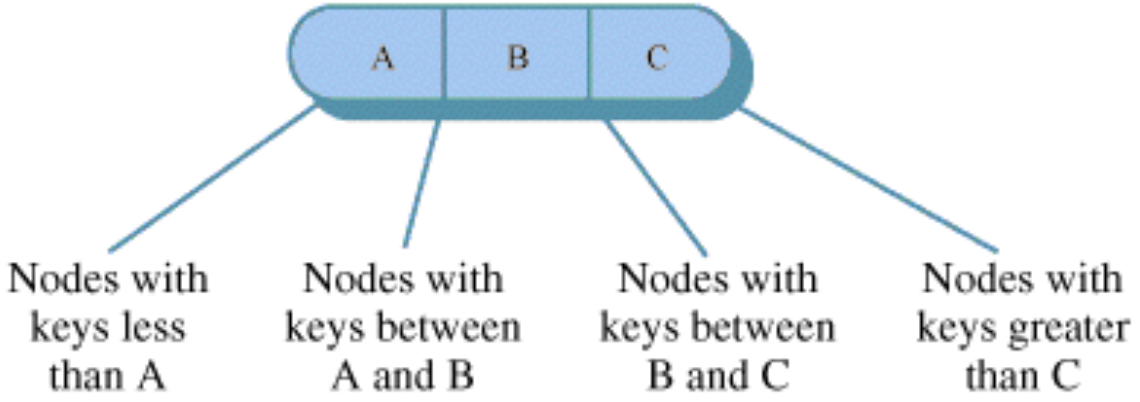
- Nodurile interne: 2, 3 sau 4 copii (1,2, sau 3 chei)
- D - nr de chei
- L - nr. copii
- L = D + 1

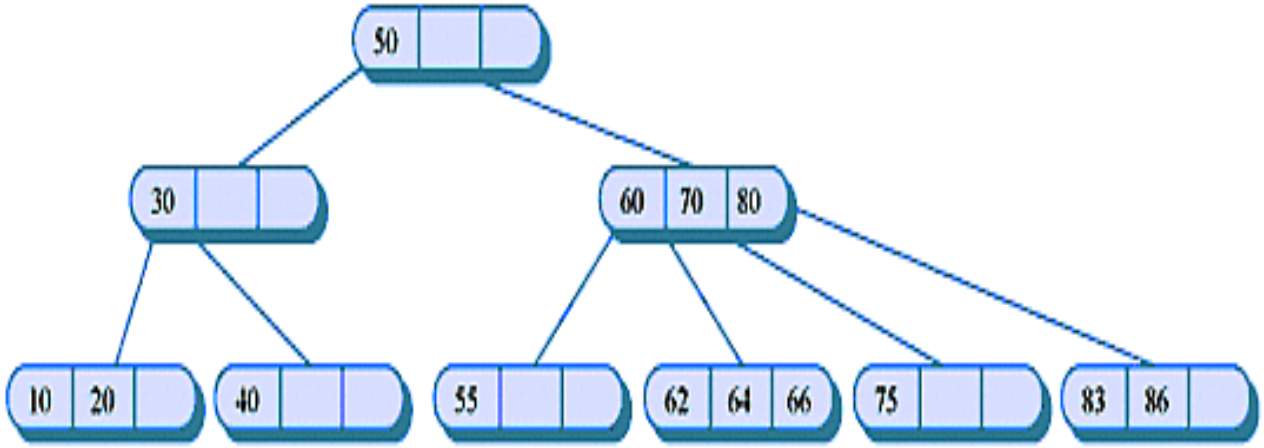






Arbori 2-3-4 (doar cheile)





Arbori 2-3-4: Insert

- similar cu arborii 2-3
- elementele sunt inserate la frunze
- intrucat un nod are maxim 4 copii, 4-nodes sunt “divizate” in timpul inserarii
- Strategie:
  - posibilitate - divizarea pe cautare: “on the way” (nu pe revenire, ca la arborii 2-3)
  - -> *inserarea poate fi realizata intr-o singura trecere*

Arborii 2-3-4: Stergere

- similar cu arborii 2-3
- elementele sunt sterse la frunza
  - interschimbare cheie nod intern cu succesorul
  - nota: dificultatea apare la frunzele de tip *2-node*
- Strategie (mai multe posibilitati):
  - pe calea de la radacina inspre frunza - se transforma *2-node* in *3-node*
  - *-> stergerea se poate efectua intr-o trecere*

Structuri de date pentru multimi disjuncte

- Aplicabilitate
  - se porneste de la o colectie de obiecte, initial fiecare in multimea proprie
  - pe baza unei relatii de echivalenta intre elemente, multimile se reunesc 2 cate doua
  - relatia de apartenenta e importanta

Structuri de date pentru multimi disjuncte

- Clase de echivalenta
  - Daca multimea S are o relatie de echivalenta definita pe elementele ei (reflexiva, tranzitiva, simetrica), atunci ea se poate partitiona in multimile $S_1, S_2, \ldots S_n$, a.i.
    - $\bigcup_k S_k = S$ si $\forall i, j \; S_i \cap S_j = \phi$
- Problema de echivalenta
  - se da S si o secventa de relatii de forma $a \equiv b$
  - se proceseaza relatiile in ordine, astfel incat la orice moment sa se poata specifica clasa de echivalenta de care apartine un anumit element

ADT Multimi disjuncte (Union-Find ADT)

- Exemplu
  - S={1,2,…,7}
  - $1 \equiv 2$, $5 \equiv 6$, $3 \equiv 4$, $1 \equiv 4$
  - $1 \equiv 2$: {1,2}{3}{4}{5}{6}{7}
  - $5 \equiv 6$: {1,2}{3}{4}{5,6}{7}
  - $3 \equiv 4$: {1,2}{3,4}{5,6}{7}
  - $1 \equiv 4$: {1,2,3,4}{5,6}{7}

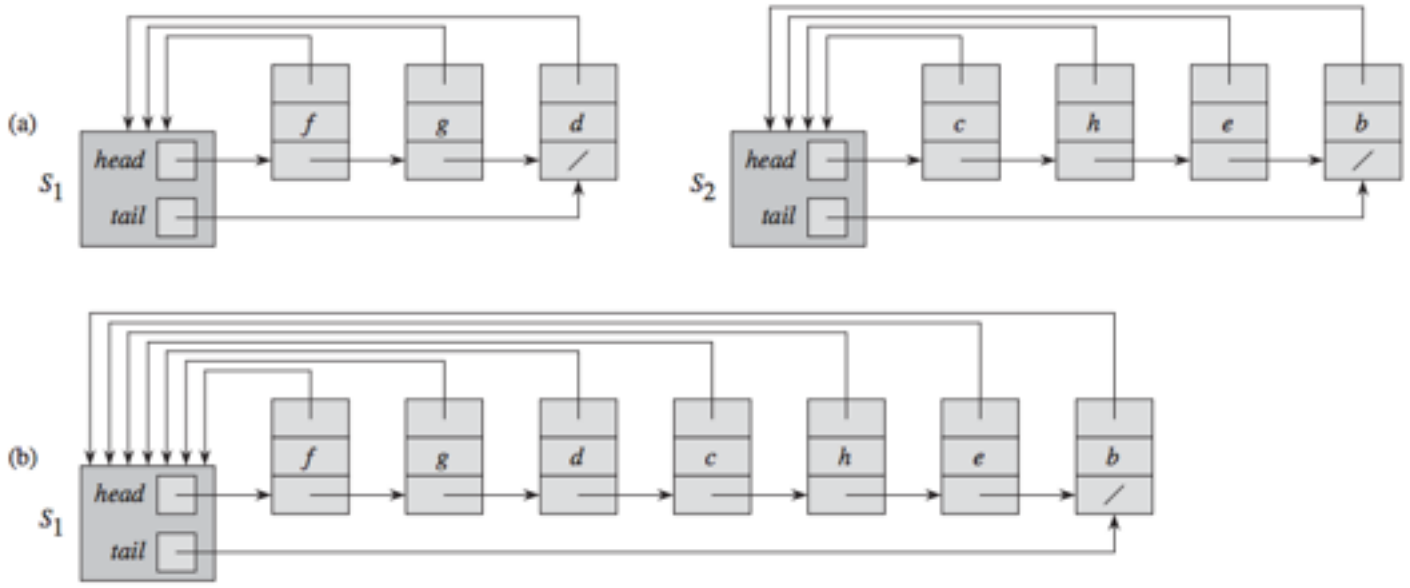
ADT Union-Find: Operatii

- union(*A, B*) genereaza reuniunea multimilor A si B; rezultatul este stocatfie in A, fie in B; cealalta multime dispare
- find(*x*), returneaza componenta de care apartine x (nume, referinta, etc)
- makeSet(*A, x*) creeaza componenta *A* care contine elementul x
- Fiecare multime are un element reprezentativ, prin care se identifica multimea

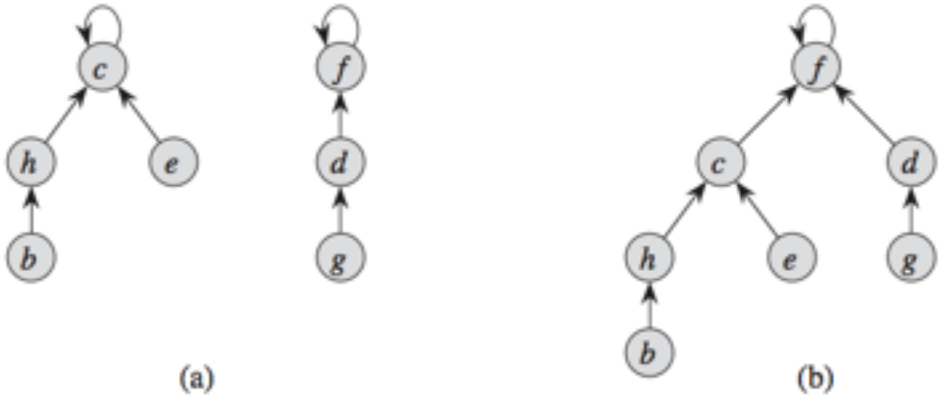
ADT Union-Find: Implementari

- Lista
  - Multimea - stocata ca o secventa reprezentata printr-o lista inlantuita (pointer la first, eventual si la last)
  - Fiecare nod - obiect de multime; contine elementul, adresa urmatorului element si referinta la identificatorul multimii
- Arbore
  - Multimea - arbore, radacina arborelui identifica multimea
  - Fiecare nod - obiect de multime; contine elementul si referinta la un nod parinte
  - Radacina - identificatorul multimii; legatura parinte pointeaza catre acelasi nod (self-reference)

Exemple



Implementarea cu lista

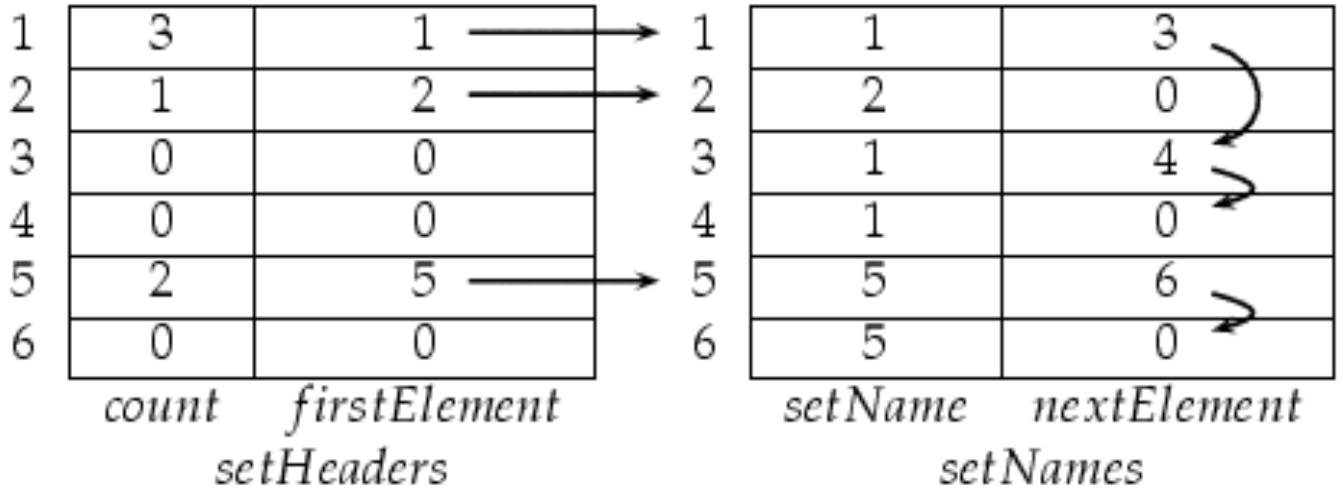


Implementarea cu arbore

Union-find ADT. Implementarea cu lista

```
const int n = //appropriate value
typedef int NameT;
typedef int ElementT;
typedef struct ufset
{
  struct // headers for set lists
  {
    int count:;
    int firstElement ;
  } setHeaders[ n ] ;
```

```
struct
/* table giving set
   containing each member
   */
  {
    NameT setName ;
    int nextElement ;
  } setNames[ n ] ;
} UFSetT;
```



Union-find ADT. Implementarea cu arbore

- S - padure de arbori, cate 1 arbore/partitie
- Initial - ***n*** arbori, fiecare continand cate 1 element
- ***find(x)*** - returneaza radacina arborelui care contine elementul ***x***
- ***union(x,y)*** combina arborii care contin elementele ***x*** si ***y***
- Avem nevoie de acces in sus in arbore (de la element - identificare radacina)
- se poate utiliza reprezentarea de vectori de parinti pentru arbore: *parent[i]* (-1 pt radacina)

Union-find ADT. Implementarea cu arbore

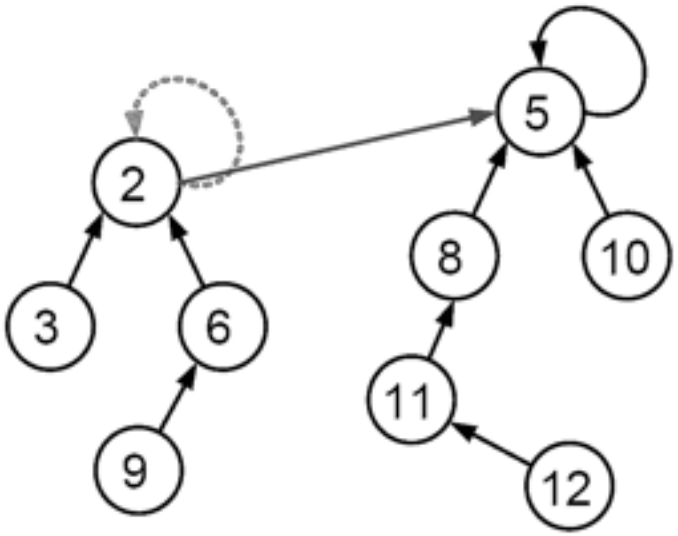
```
INIT()
1  for i ← 1 to n
2       do parent[i] ← 0
```

```
FIND(i)
1  j ← i
2  while parent[j] > 0
3       do j ← parent[j]
4  return j
```

```
UNION(i, j)
1  root_1 ← FIND(i)
2  root_2 ← FIND(j)
3  if root_1 ≠ root_2
4       then parent[root_2] ← root_1
```

Union-find ADT. Implementarea cu arbore. Imbunatatiri

- Stocam in fiecare nod informatie legata de limita superioara a inaltimii - ***rang***
- ***Union by rank:***
  - La o operatie de union, devine radacina arborele cu rangul minim

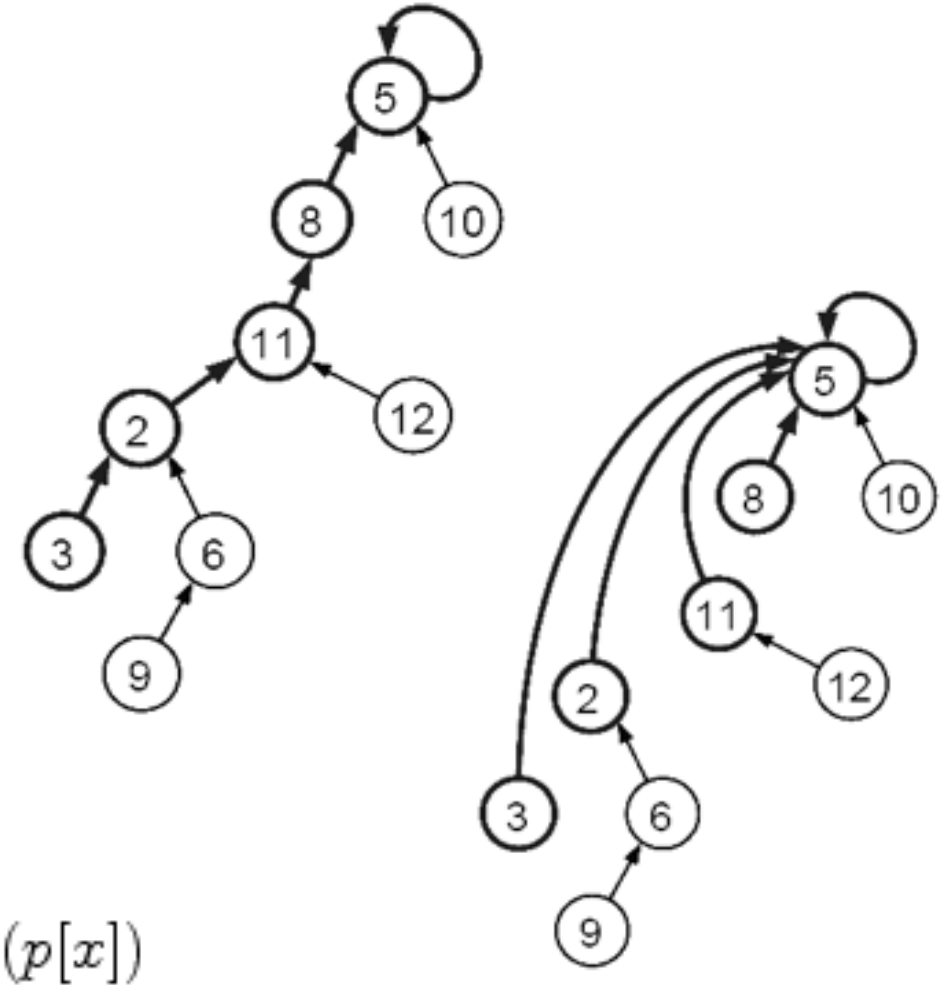



```
LINK(x, y)
1  if r[x] > r[y]
2      then p[y] ← x
3      else p[x] ← y
4           if r[x] = r[y]
5              then r[y] ← r[y] + 1
```

```
UNION(x, y)
1  LINK(FINDSET(x), FINDSET(y))
```

Union-find ADT. Implementarea cu arbore. Imbunatatiri

- ***Path compression***
  - dupa efectuarea unui find, se compreseaza toti pointerii de pe calea traversata astfel incat sa pointeze catre radacina




```
FINDSET(x)
1  if x ≠ p[x]
2      then p[x] ← FINDSET(p[x])
3  return p[x]
```

Heuristici de imbunatatire a timpului: analiza

- Union by rank: implica timp $O(n \log n)$ pentru n operatii union-find:
  - de fiecare cand urmam un pointer parinte, trecem intr-un sub-arbore de dimensiune cel putin dubla fata de sub-arborele initial
  - prin urmare, o operatie de ***find*** parcurge cel mult $\boldsymbol{O(\log n)}$ ponteri
- Path compression: implica imp $O(n \log n)$ pentru n operatii union-find

- $O(m\alpha(n))$, $\alpha(n)$ o functie care creste f incet
  - n - numarul de operatii MAKE-SET
  - m - numarul total de operatii (MAKE-SET, FIND-SET, UNION)

Bibliografie

- http://algoviz.org/OpenDSA/Books/CS3114/html/TwoThreeTree.html#
- https://en.wikipedia.org/wiki/2%E2%80%933%E2%80%934_tree
- CLR - capitolul 21 (Data Structures for Disjoint Sets)