MOVILE

Șl. Dr. Ing. Șerban Radu
Departamentul de Calculatoare
Facultatea de Automatică și Calculatoare

Introducere

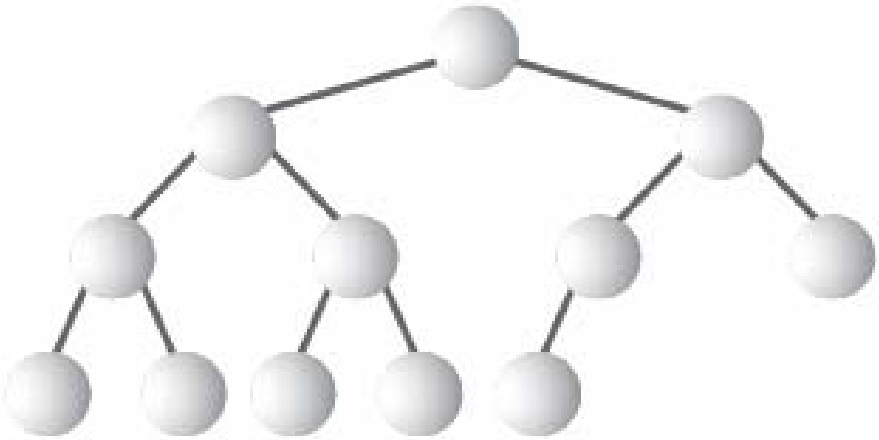
- Movila reprezintă un tip special de arbore
- Inserarea și ștergerea dintr-o movilă se efectuează într-un timp de ordinul O(logN)
- Ștergerea nu mai este la fel de rapidă, dar inserarea devine mult mai rapidă decât în cazul arborilor
- Movila este utilizată la implementarea cozilor cu priorități, când viteza este importantă și e necesară efectuarea multor inserări

Definiție

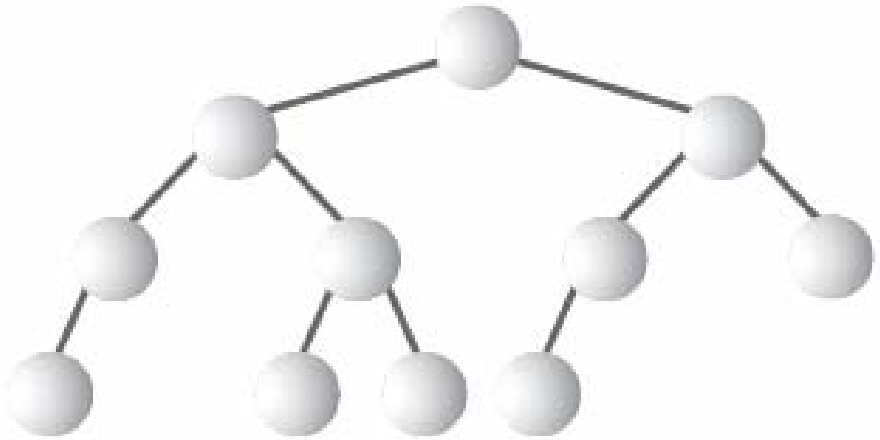
- Movila este un arbore binar cu următoarele proprietăți:
- 1. Este complet
- Arborele este integral completat dacă, atunci când este parcurs pe fiecare nivel, de la stânga la dreapta, nu prezintă locuri libere; ultimul nivel poate să nu fie complet

Definiție

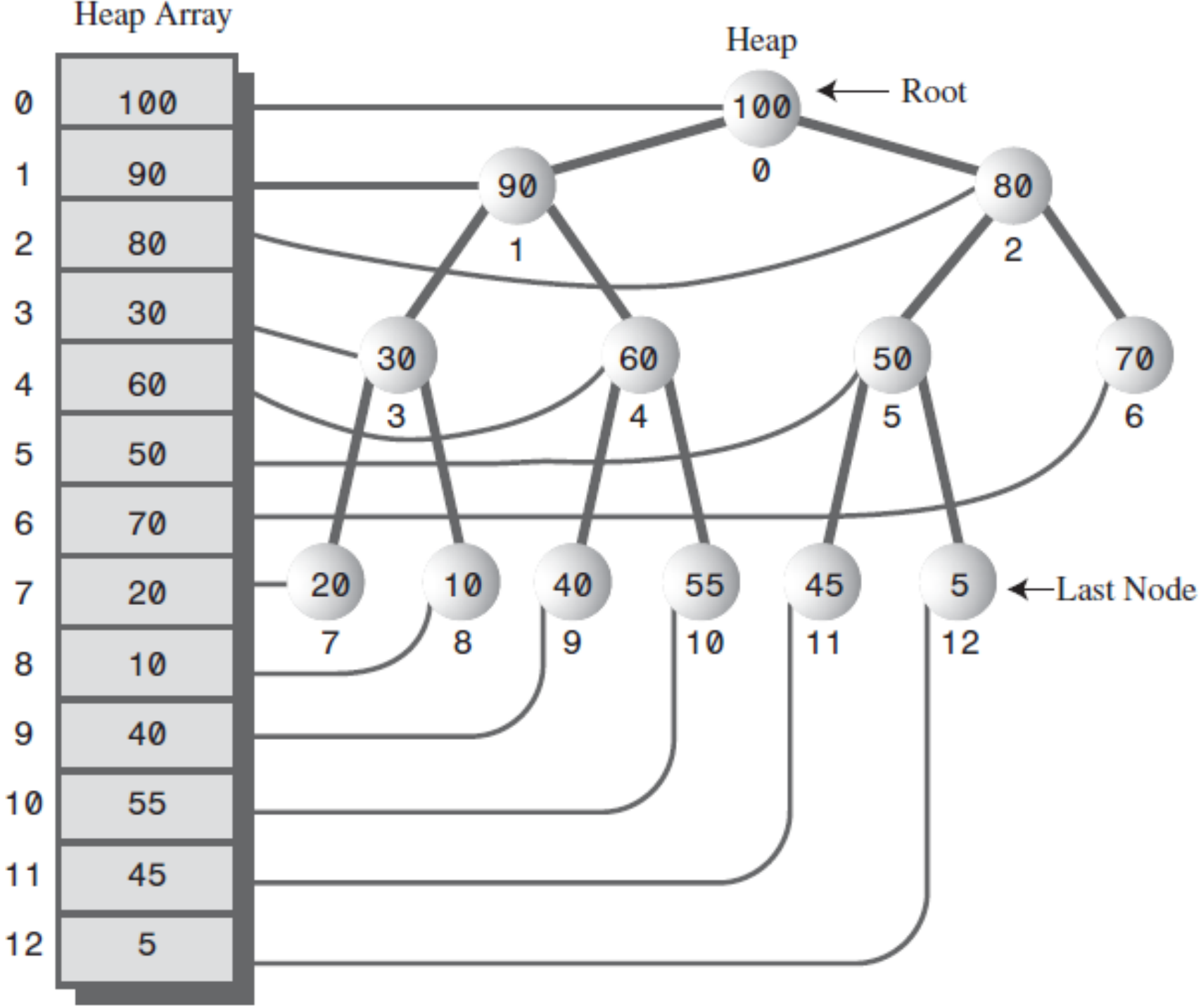
- 2. Arborele este implementat printr-un tablou
- 3. Fiecare nod dintr-o movilă îndeplinește **condiția de movilă**, care precizează că acel nod trebuie să aibă o cheie mai mare (sau egală) decât oricare dintre fiii săi



a) Complete



b) Incomplete


Heap Array
Heap
0 100
1 90
2 80
3 30
4 60
5 50
6 70
7 20
8 10
9 40
10 55
11 45
12 5
100
0
Root
90
1
80
2
30
3
60
4
50
5
70
6
20
7
10
8
40
9
55
10
45
11
5
12
Last Node

Observații

- Tabloul reprezintă structura care se găsește fizic în memorie
- Movila în sine este doar o reprezentare conceptuală
- Se observă completitudinea arborelui și verificarea condiției de movilă pentru fiecare nod

Observații

- Completitudinea arborelui asigură lipsa golurilor din tabloul utilizat în implementare
- Tabloul are toate celulele ocupate, începând cu cea cu indicele 0 și terminând cu cea cu indicele N - 1

Observații

- Se consideră că rădăcina conține cheia maximă
- Cu ajutorul acestei movile, se poate implementa o coadă cu ordine descrescătoare a priorităților

Observații

- Rădăcina are indicele **0** (este primul element din tablou)
- Pentru nodul din pozitia **k**, nodurile vecine sunt:
  - Fiul stânga în poziția **2*k + 1**
  - Fiul dreapta în poziția **2*k + 2**
  - Părintele în poziția **k/2**, dacă k impar
  - Părintele în poziția **k/2 - 1**, dacă k par

Movilă de numere întregi

- typedef struct {
- int v[M];
- int n;
- } heap;
- Se definește o movilă cu maxim **M** numere și dimensiunea efectivă a tabloului **n**

Observații

- Movila este slab ordonată, în comparație cu un arbore binar de căutare, în care toți descendenții stângi ai unui nod au chei mai mici decât descendenții drepți
- Într-o movilă, traversarea nodurilor în ordine este dificilă, deoarece principiul de organizare (condiția de movilă) nu este la fel de puternic ca în cazul unui arbore binar

Observații

- Într-o movilă, pe fiecare cale de la rădăcină către o frunză, nodurile sunt ordonate descrescător
- Nodurile din stânga sau din dreapta unui nod dat, de pe niveluri mai joase sau mai înalte, cu excepția celor situate pe aceeași cale cu nodul dat, pot avea chei mai mari sau mai mici decât nodul dat

Observații

- Ordonarea specifică movilei este suficientă pentru a asigura ștergerea rapidă a nodului cu valoarea maximă și inserarea rapidă a unui nod nou
- Aceste operații sunt suficiente atunci când se utilizează movila la implementarea unei cozi cu priorități

Ștergerea

- Prin ștergere se înțelege ștergerea elementului cu cheia maximă
- Acest nod este întotdeauna rădăcina arborelui, deci ștergerea sa este simplă
- Rădăcina este elementul cu indicele 0 al tabloului:
- `maxNode = h.v[0];`

Observații

- După ștergerea rădăcinii, arborele nu rămâne complet
- Se pot deplasa toate elementele din tablou cu o celulă înapoi, dar există o soluție mult mai rapidă

Ștergerea nodului rădăcină

- 1. Ștergerea propriu-zisă a rădăcinii
- 2. Deplasarea ultimului nod din arbore în nodul rădăcină
- 3. “Filtrarea” acestui nod în jos, până când ajunge dedesubtul unui nod mai mare decât el și deasupra unuia mai mic

Observații

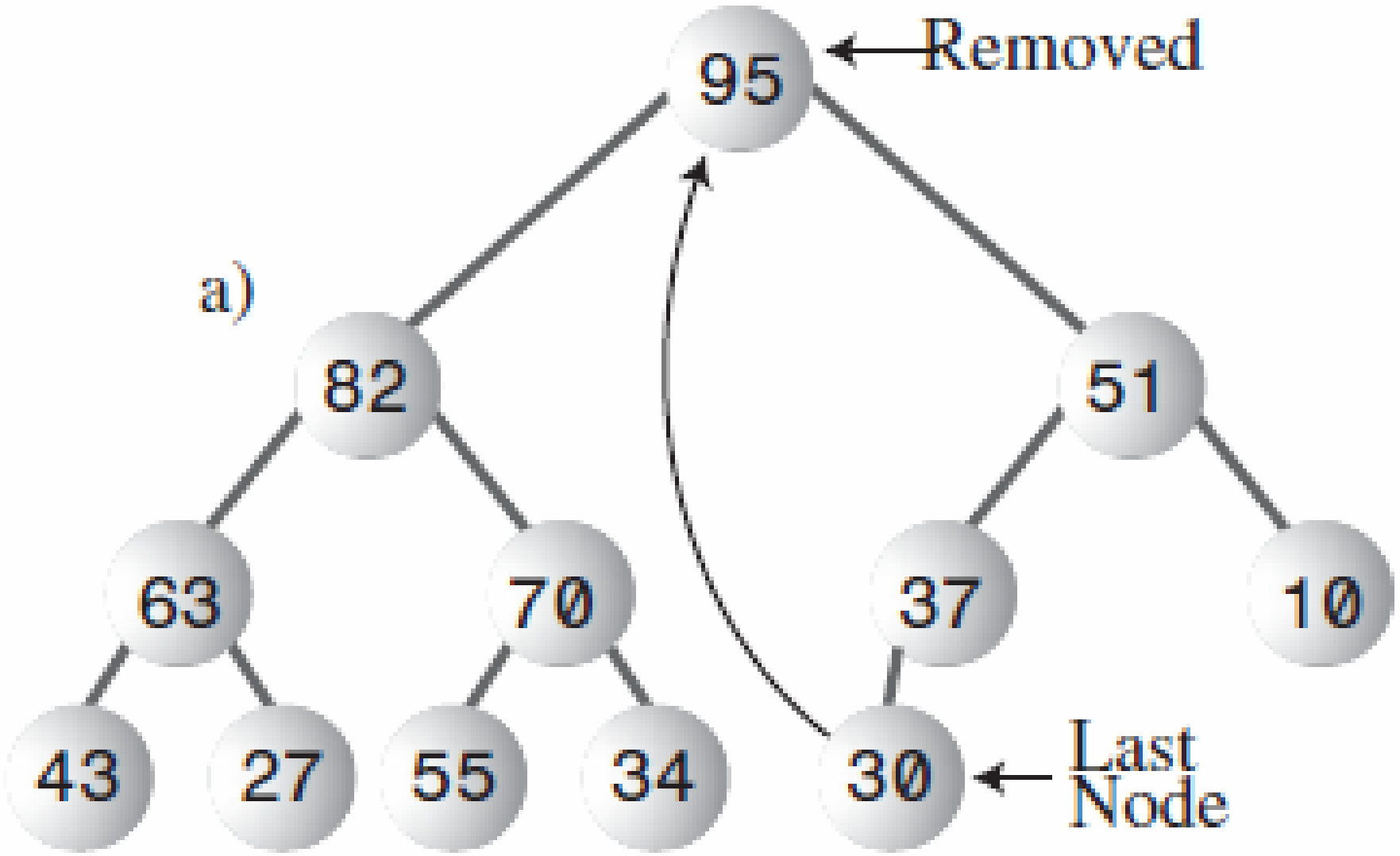
- Ultimul nod = nodul cel mai din dreapta de pe ultimul nivel din arbore
- Acest nod corespunde ultimei celule ocupate din tablou
- Copierea acestui nod în rădăcină se face prin instrucțiunile:
- `h.v[0] = h.v[h.n-1];`
- `h.n--;`

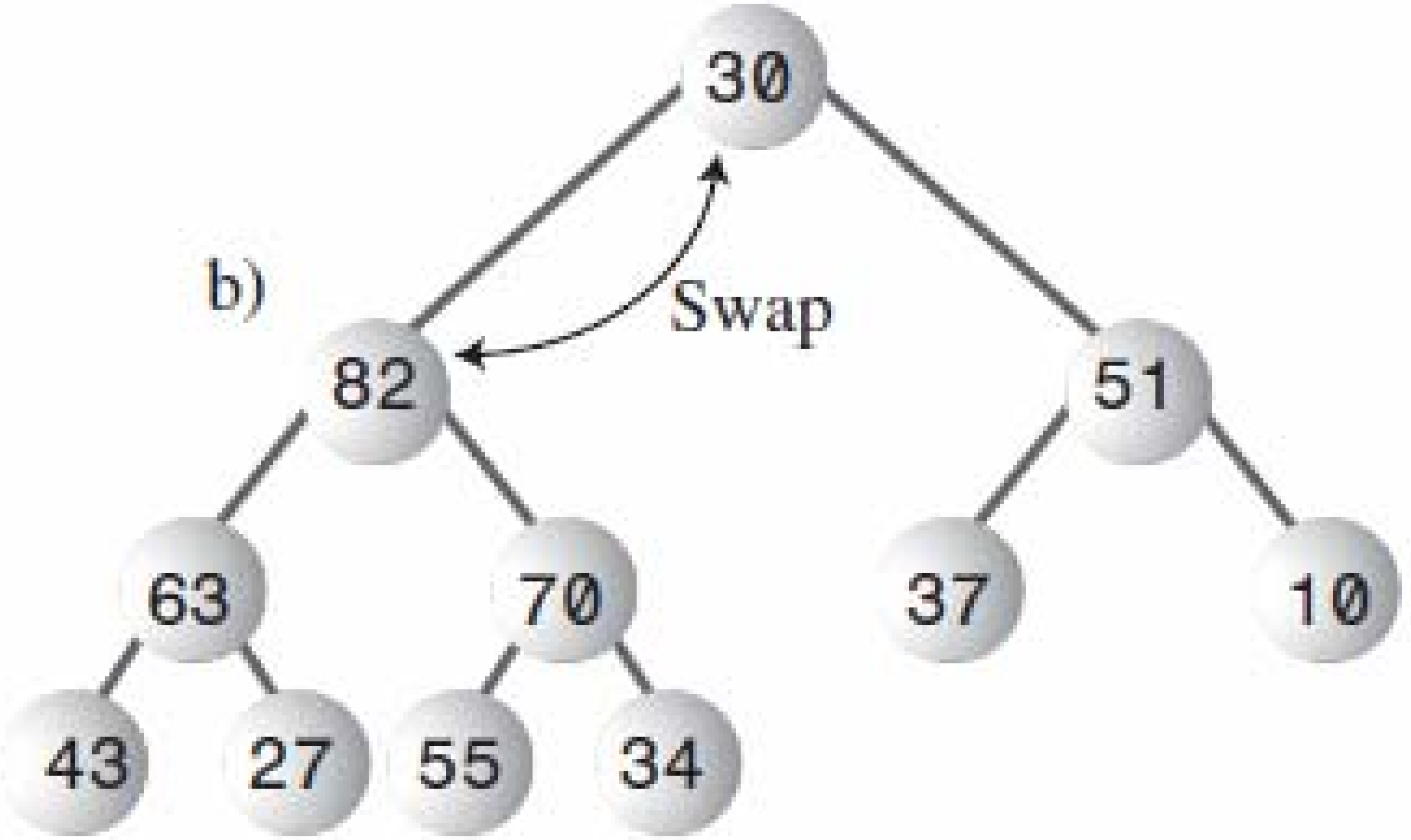
Observații

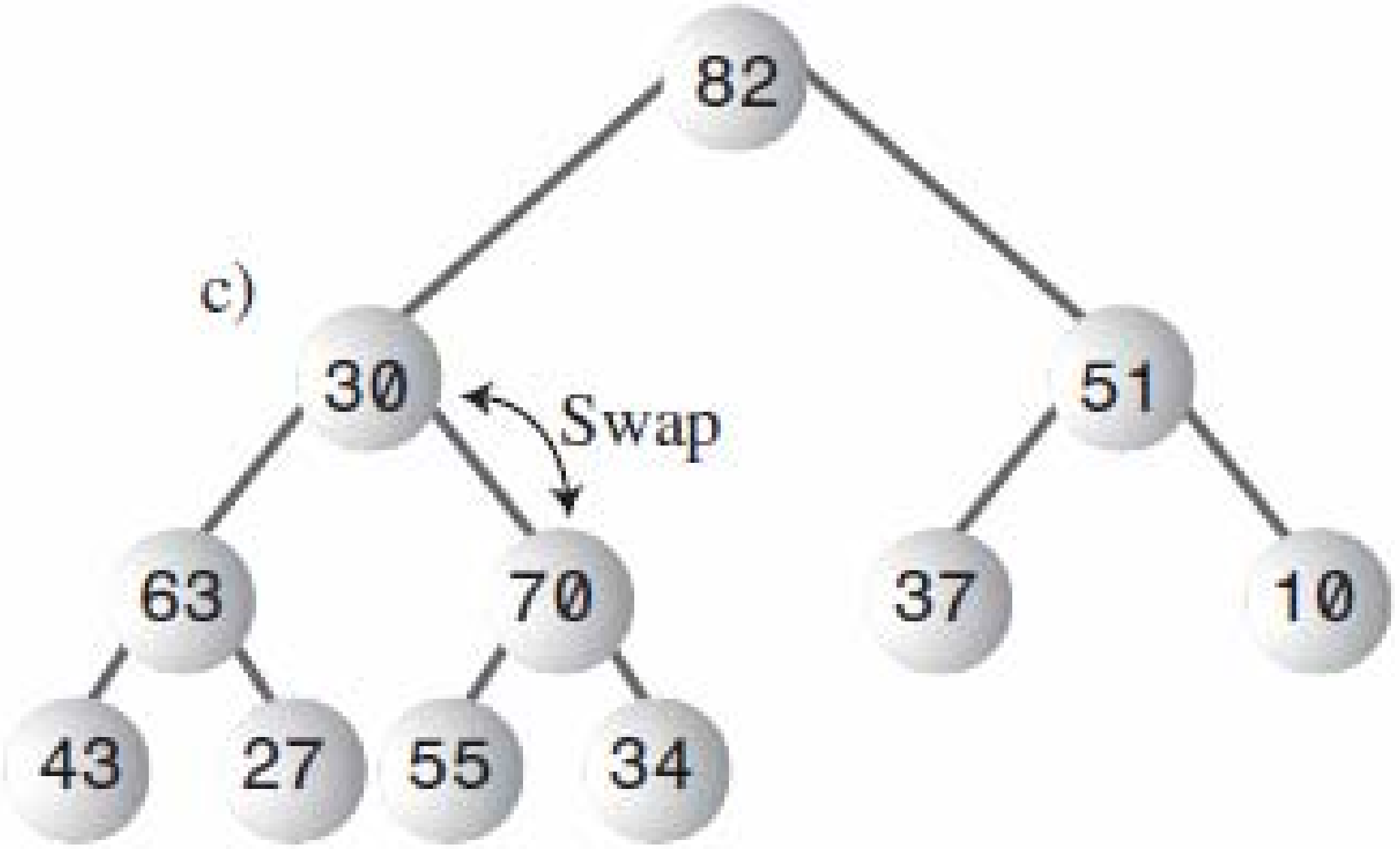
- Ștergerea rădăcinii determină scăderea dimensiunii tabloului cu o unitate
- Filtrarea unui nod în sus sau în jos presupune deplasarea nodului de-a lungul unei căi, pas cu pas, permutând nodul cu nodul următor și verificând dacă se află în poziția corespunzătoare

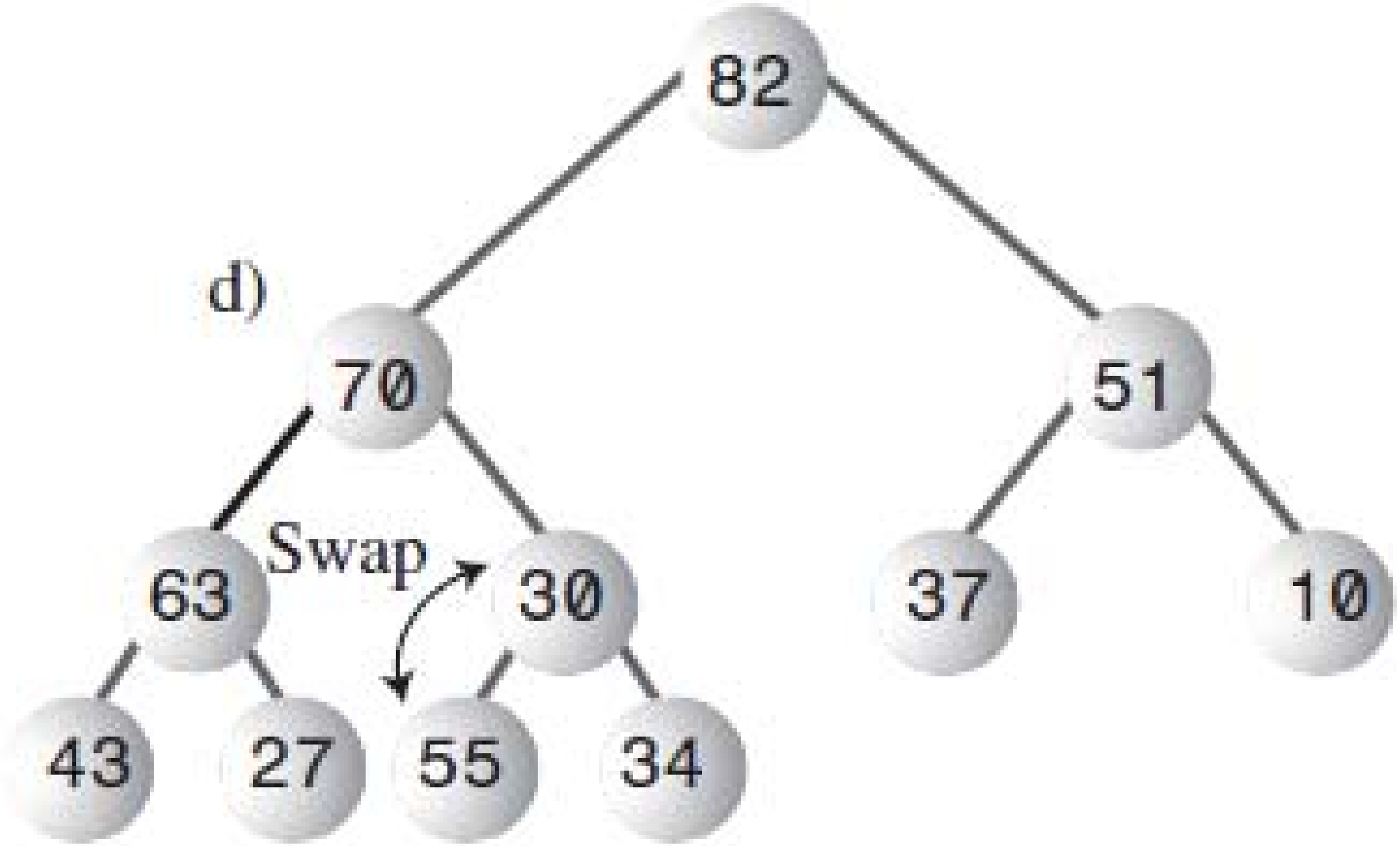
Observații

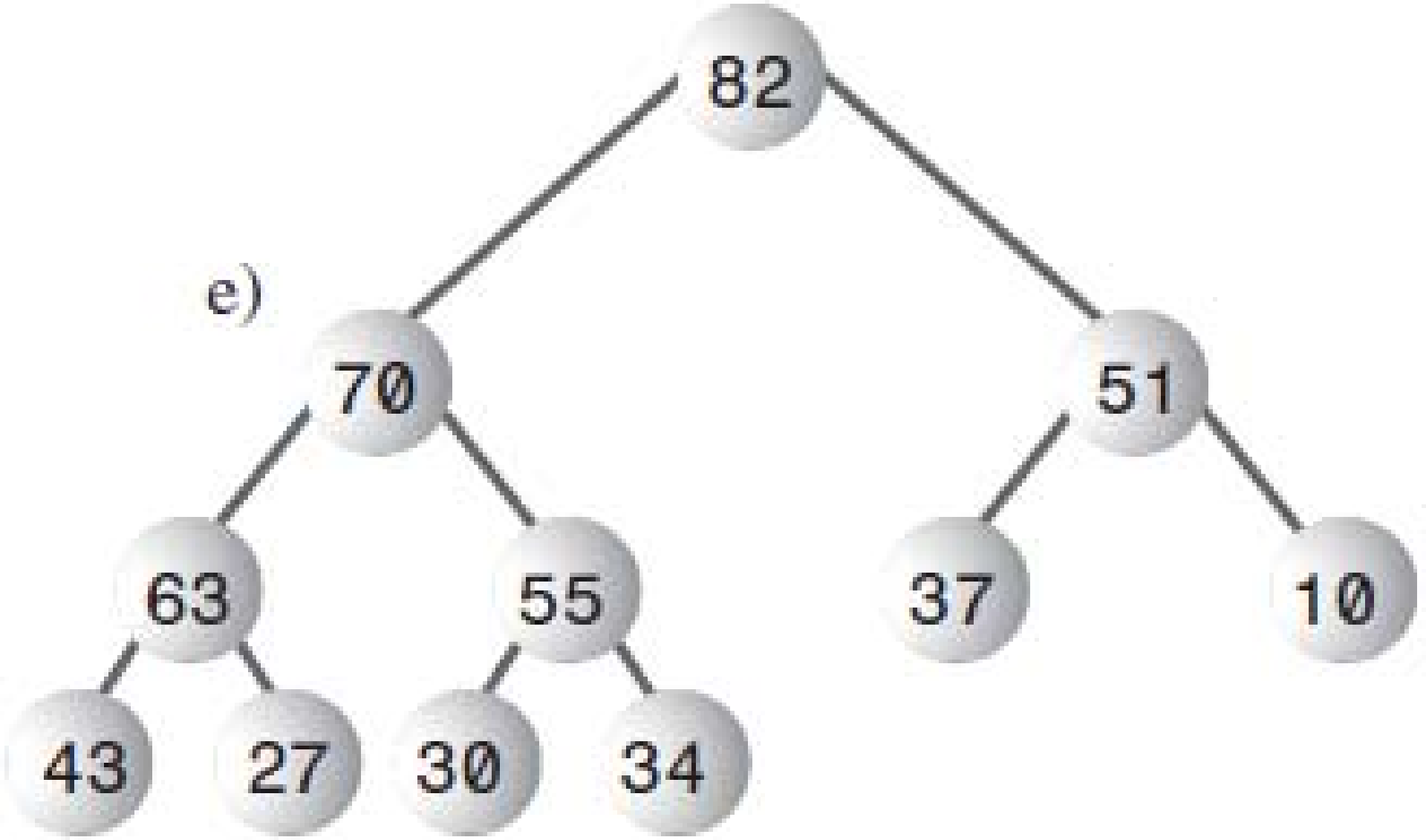
- În pasul 3 al algoritmului, noua rădăcină a arborelui are o valoare prea mică, deci va fi filtrată în jos, prin movilă, până când va ajunge la locul potrivit
- Pasul 2 asigură refacerea completitudinii arborelui, iar pasul 3 asigură verificarea condiției de movilă (fiecare nod părinte trebuie să fie mai mare decât fiii săi)


a)
95
Removed
82
51
63
70
37
10
43
27
55
34
30
Last
Node


b)
Swap
30
82
51
63
70
37
10
43
27
55
34


c)
82
30
51
Swap
63
70
37
10
43
27
55
34


d)
82
70
51
63
Swap
30
37
10
43
27
55
34


e)
82
70
51
63
55
37
10
43
27
30
34

Algoritmul de ștergere

```
// extragere valoare maximă din coada de priorităţi
int delmax ( heap & h) {
int hmax;
  if (h.n <= 0) return -1;
  hmax = h.v[0];   // maxim în prima poziţie din tablou
  h.v[0] = h.v[h.n - 1];
        // se aduce ultimul element în prima poziţie
  h.n--;                  // scade dimensiunea tabloului
  heapify (h, 0);
        // ajustare pt menţinere condiţie de movilă
  return hmax;
}
```

Operația “heapify”

- Operația “heapify” reface o movilă dintr-un arbore, la care elementul **k** nu respectă condiția de movilă, dar subarborii săi respectă această condiție
- La această situație se ajunge după ștergerea sau după modificarea valorii din rădăcina unei movile

Operația “heapify”

- Aplicată asupra unui tablou oarecare, funcția “heapify” nu creează o movilă, dar aduce în poziția **k** cea mai mare dintre valorile subarborelui cu rădăcina în **k**
- Se mută succesiv în jos în arbore valoarea **v[k]**, dacă nu este mai mare decât fii săi
- Funcția recursivă "heapify" propagă în jos valoarea din nodul **i**, astfel încât arborele cu rădăcina în **i** să fie o movilă

Operația “heapify”

- Se determină valoarea maximă dintre **v[i]**, **v[st]** și **v[dr]** și se aduce în poziția **i**, pentru a avea **v[i] >= v[st]** și **v[i] >= v[dr]**, unde **st** și **dr** sunt adresele (indicii) succesorilor la stânga și la dreapta ai nodului din poziția **i**
- Valoarea coborâtă din poziția **i** în **st** sau **dr** va fi din nou comparată cu succesorii săi, la un nou apel al funcției "heapify"

Funcția de interschimbare valori

```
// schimbă între ele valorile v[i] și v[j]
void swap (heap & h, int i, int j) {
   int t;
   t = h.v[i];
   h.v[i] = h.v[j];
   h.v[j] = t;
}
```

Funcția recursivă “heapify”

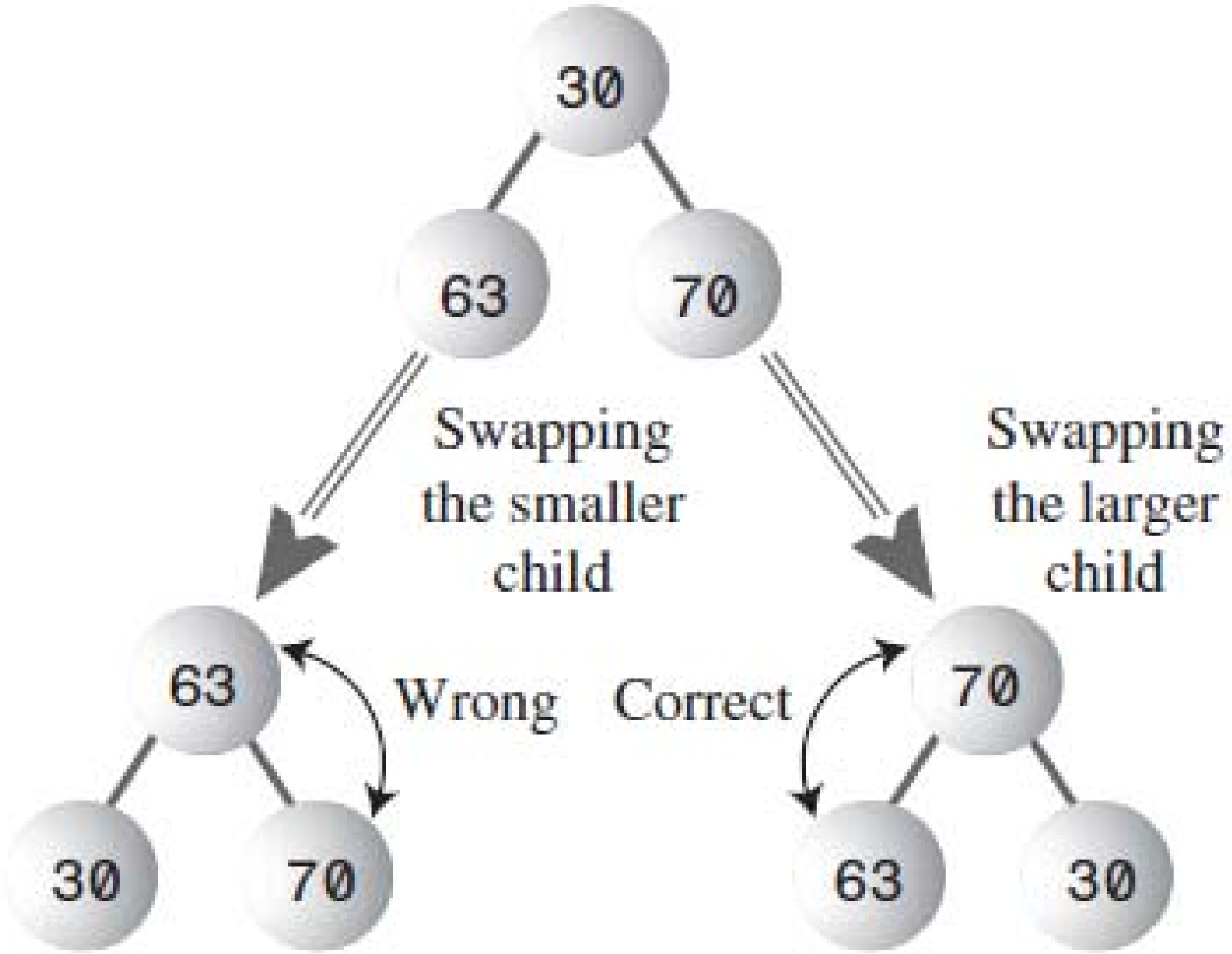
```
void heapify (heap & h,int i) {
  int st, dr, m, aux;
  st = 2 * i; dr = st + 1;            // succesorii nodului i
   // determină maxim dintre valorile din pozițiile i, st, dr
  if (st <= h.n - 1 && h.v[st] > h.v[i])
     m = st;                          // maxim în stânga lui i
  else m = i;                         // maxim în poziția i
  if (dr <= h.n - 1 && h.v[dr] > h.v[m])
     m = dr;                          // maxim în dreapta lui i
     if (m != i) {                    // dacă e necesar
       swap(h, i, m);                 // schimbă maxim cu v[i]
       heapify(h, m);                 // ajustare din poziția m
  }   }
```

Funcția iterativă “heapify”

```
void heapify (heap & h, int i) {
  int st, dr, m = i;            // m = indice valoare maximă
  while (2 * i <= h.n - 1) {
    st = 2 * i; dr = st + 1;          // succesori nod i
    if (st <= h.n - 1 && h.v[st] > h.v[m] )  // dacă v[m] < v[st]
      m = st;
    if (dr <= h.n - 1 && h.v[dr] > h.v[m])  // dacă v[m] < v[dr]
      m = dr;
    if ( i == m) break;               // gata dacă v[i] nemodificat
    swap (h, i, m);                   // interschimb v[i] cu v[m]
    i = m;
  }   }
```

Observații

- În fiecare poziție a nodului, algoritmul de filtrare în jos determină care dintre fiii acestuia este mai mare
- Algoritmul permută apoi nodul cu cel mai mare dintre fii
- Dacă algoritmul ar încerca permutarea cu celălalt fiu, acel fiu va deveni părintele fostului său frate, care este mai mare decât el, iar condiția de movilă ar fi încălcată


30
63
70
Swapping the smaller child
Swapping the larger child
63
Wrong
Correct
70
30
70
63
30

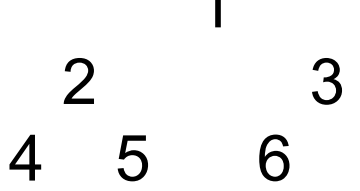
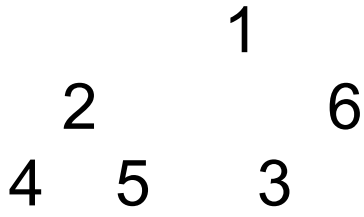
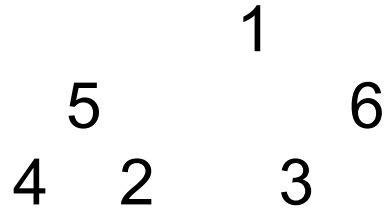
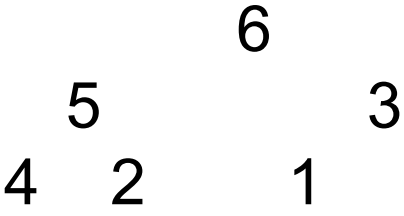
Transformarea tablou $\rightarrow$ movilă

- Transformarea unui tablou dat într-o movilă se face treptat, pornind de la frunze spre rădăcină, cu ajustare la fiecare element

Transformarea tablou $\rightarrow$ movilă

```
void makeheap (heap & h) {
  int i;
  for (i = h.n/2 - 1 ; i >= 0; i--)
        // părintele ultimului element este în poziția n/2 - 1
        heapify (h, i);
}
```

Exemplu de transformare

Operație	**Tablou**	**Arbore**
inițializare	1 2 3 4 5 6	
heapify(3)	1 2 6 4 5 3	
heapify(2)	1 5 6 4 2 3	
heapify(1)	6 5 3 4 2 1	

Inserarea

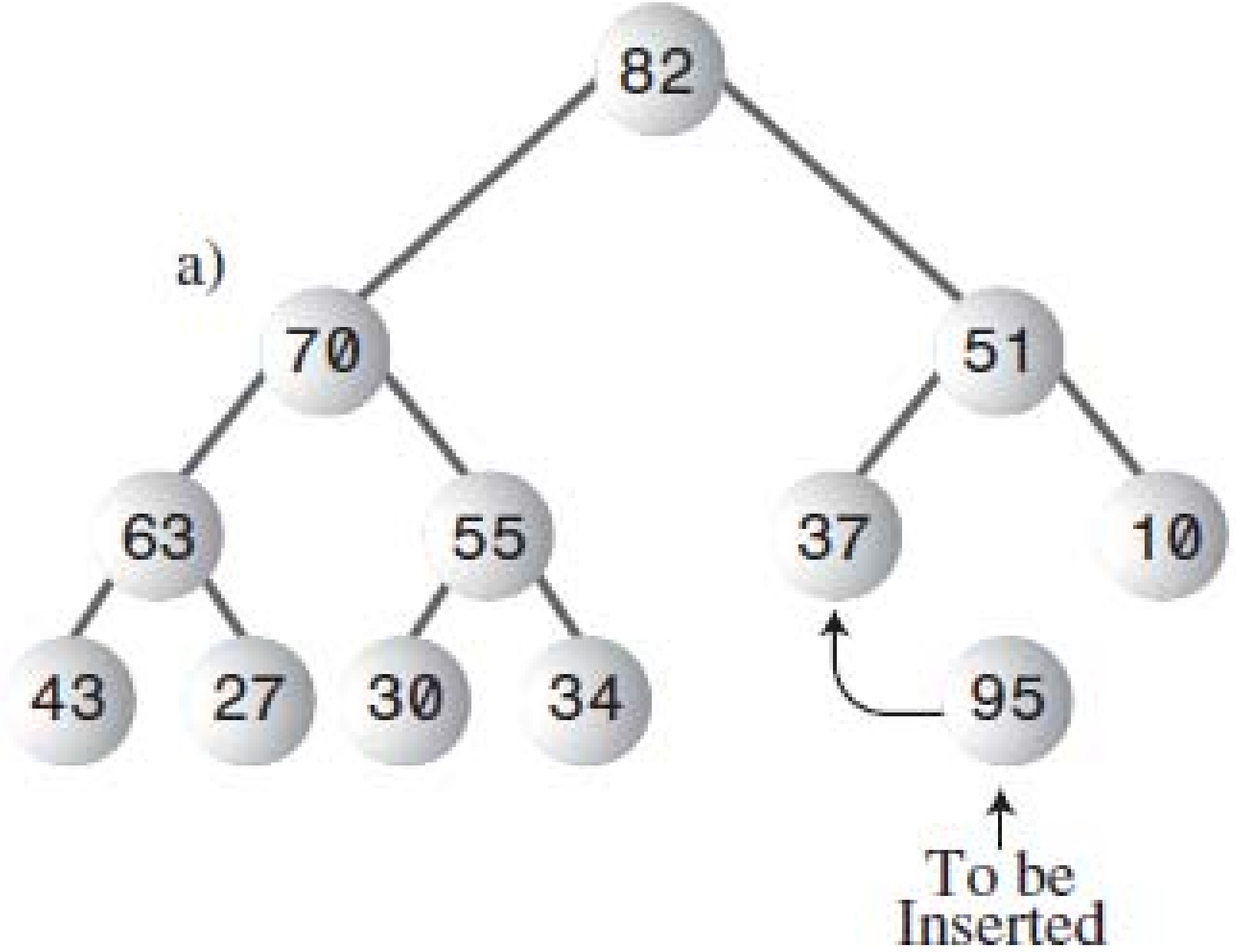
- Inserarea unui nod utilizează o filtrare în sus în locul uneia în jos
- Inițial, nodul este inserat în prima poziție disponibilă de la sfârșitul tabloului, incrementând dimensiunea acestuia:
- ++h.n;
- h.v[h.n] = newNode;

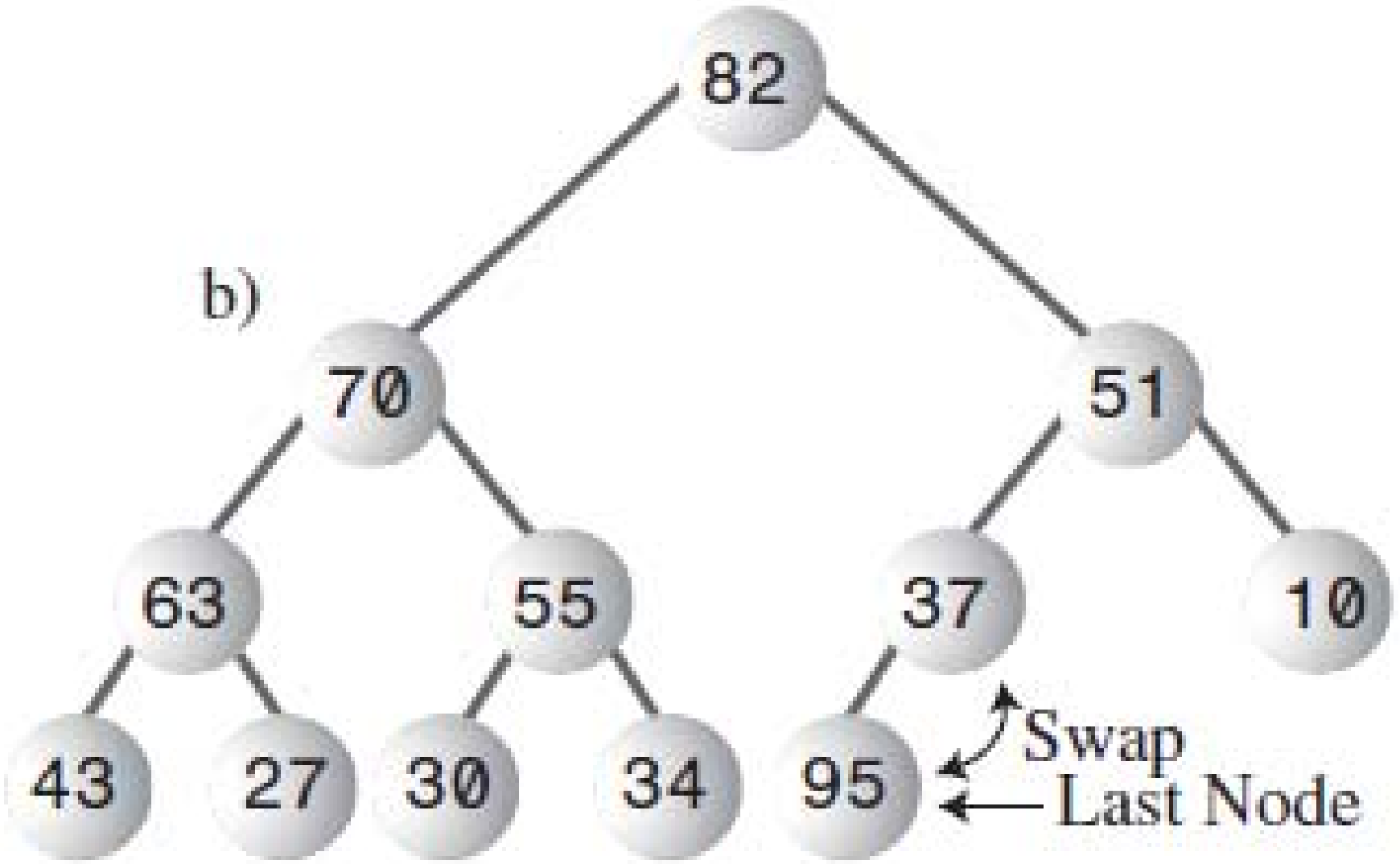
Observații

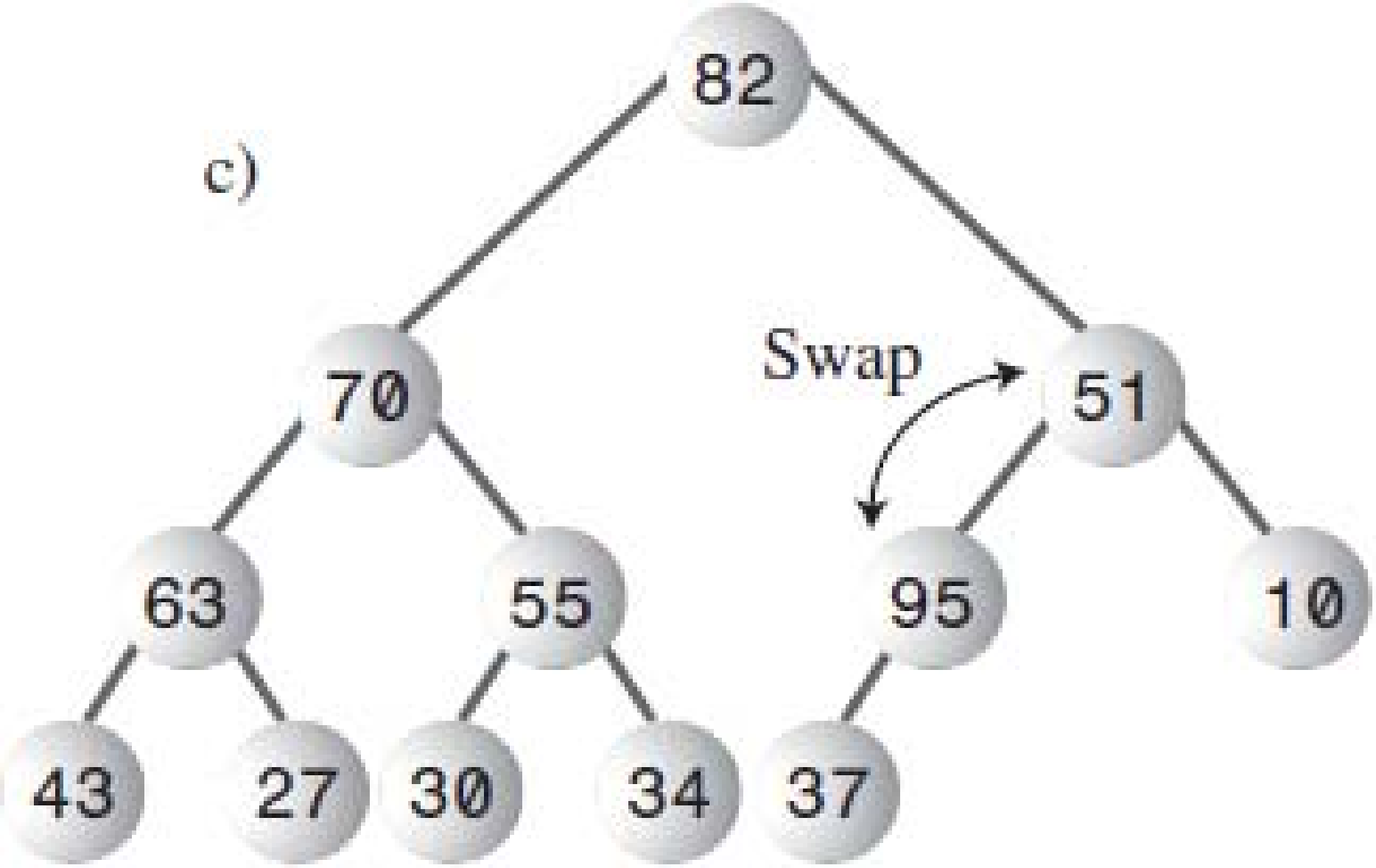
- Problema care apare după inserarea în această poziție este nerespectarea condiției de movilă
- Aceasta se întâmplă dacă nodul nou este mai mare decât nodul care i-a devenit părinte

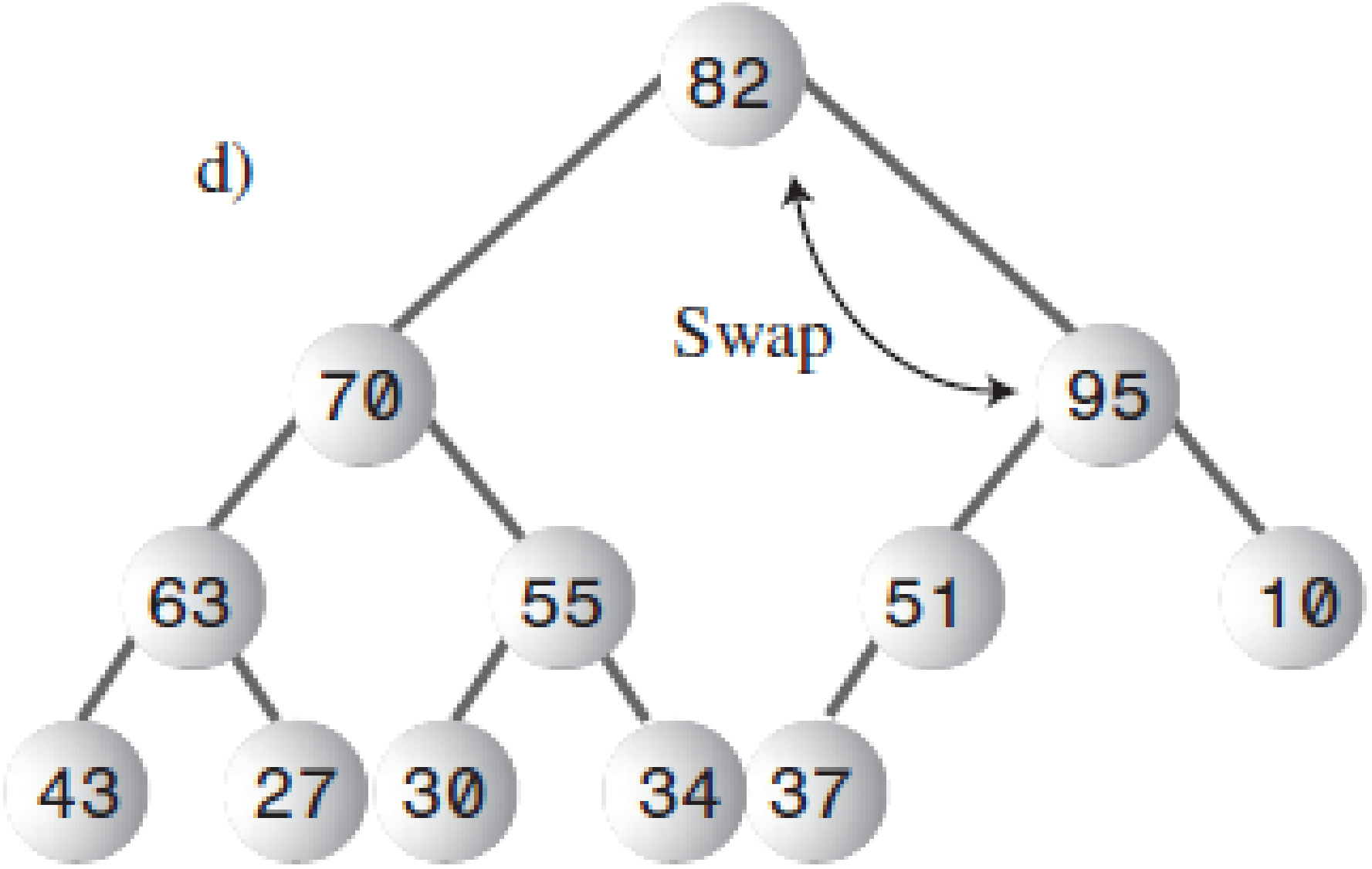
Observații

- Din cauză că nodul părinte se află la baza movilei, nodul nou va fi, în majoritatea situațiilor, mai mare decât acesta
- În consecință, nodul nou va fi filtrat în sus, până când va ajunge sub un nod cu o cheie mai mare și deasupra altui nod, cu o cheie mai mică decât a sa

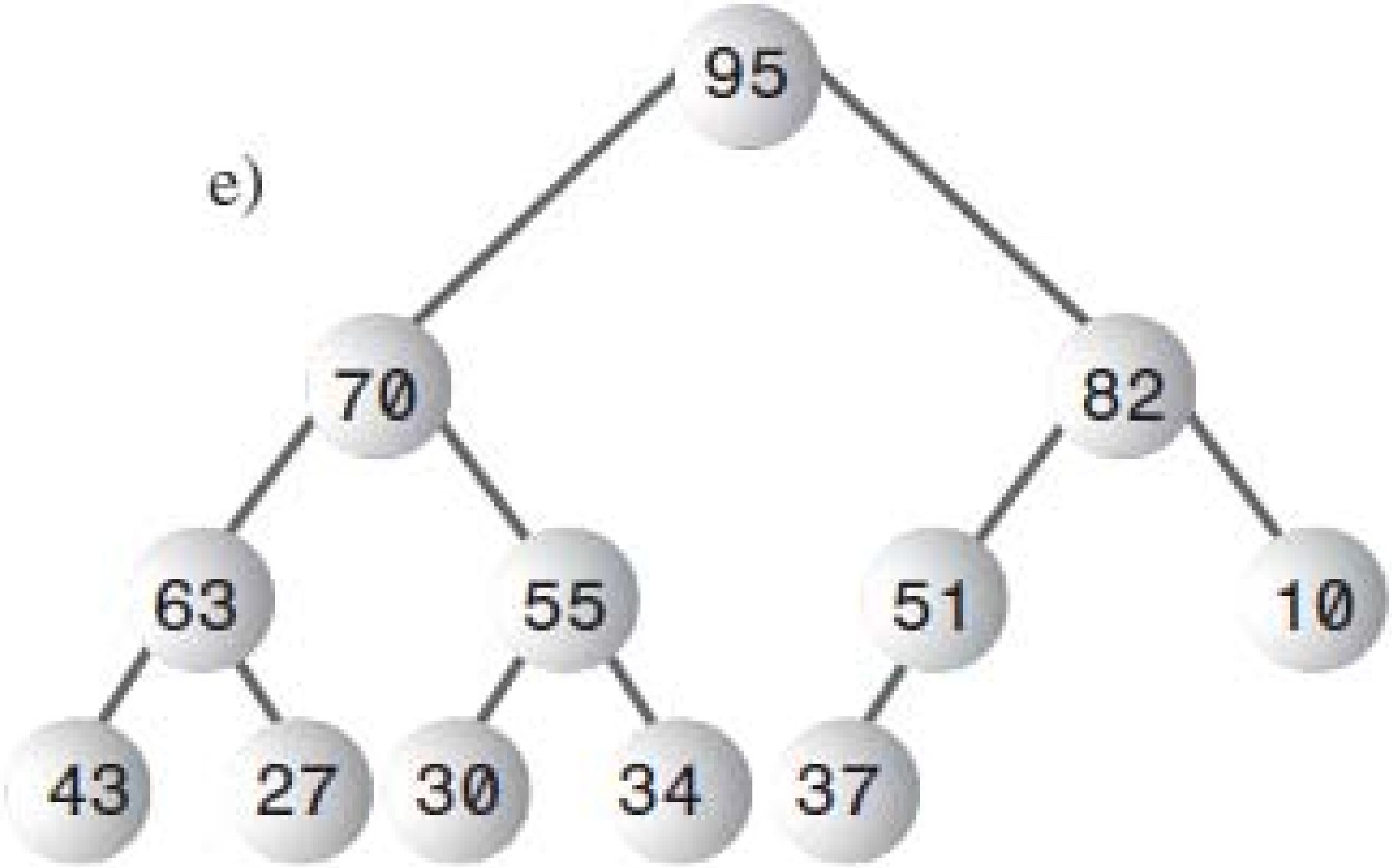

a)
82
70
51
63
55
37
10
43
27
30
34
95
To be
Inserted


b)
82
70
51
63
55
37
10
43
27
30
34
95
Swap
Last Node


c)
82
70
51
Swap
63
55
95
10
43
27
30
34
37


d)
82
Swap
70
95
63
55
51
10
43
27
30
34
37

e)



Algoritmul de inserare

```
void inserare (heap & h, int x ) {
  int i;
  i = ++h.n;                    // prima poziție liberă din tablou
  h.v[i] = x;                   // adaugă noua valoare la sfârșit
  while (i > 0 && h.v[i/2 - 1] < x ) {
                  // cât timp x este prea mare pentru poziția sa
          swap (h, i, i/2 - 1); // se schimbă cu parintele său
          i = i/2 - 1;  // și se continuă din noua poziție a lui x
  }
}
```

Exemplu de inserare

Se inserează valoarea val=7 în tabloul a=[8,5,6,3,2,4,1]

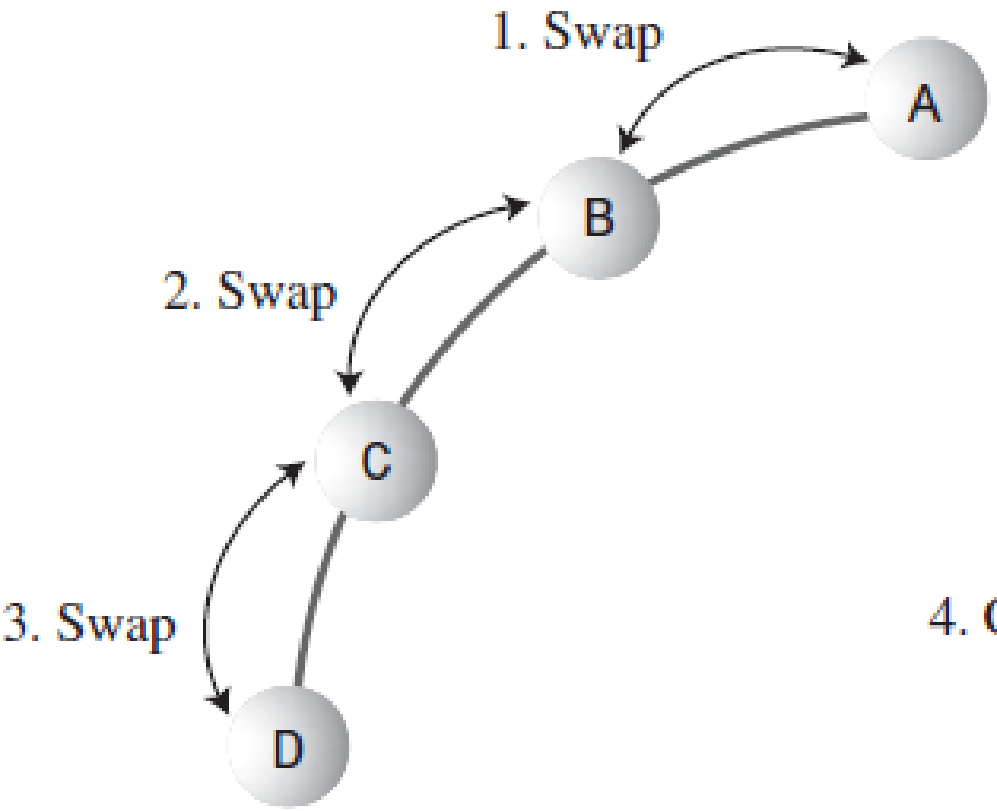
i=8, a[8]=7	a= [8,5,6,3,2,4,1,7]
i=8, a[4]=3 < 7 , a[8] cu a[4]	a= [8,5,6,7,2,4,1,3]
i=4, a[2]=5 < 7 , a[4] cu a[2]	a= [8,7,6,5,2,4,1,3]
i=2, a[1]=8 > 7	a= [8,7,6,5,2,4,1,3]

Observații

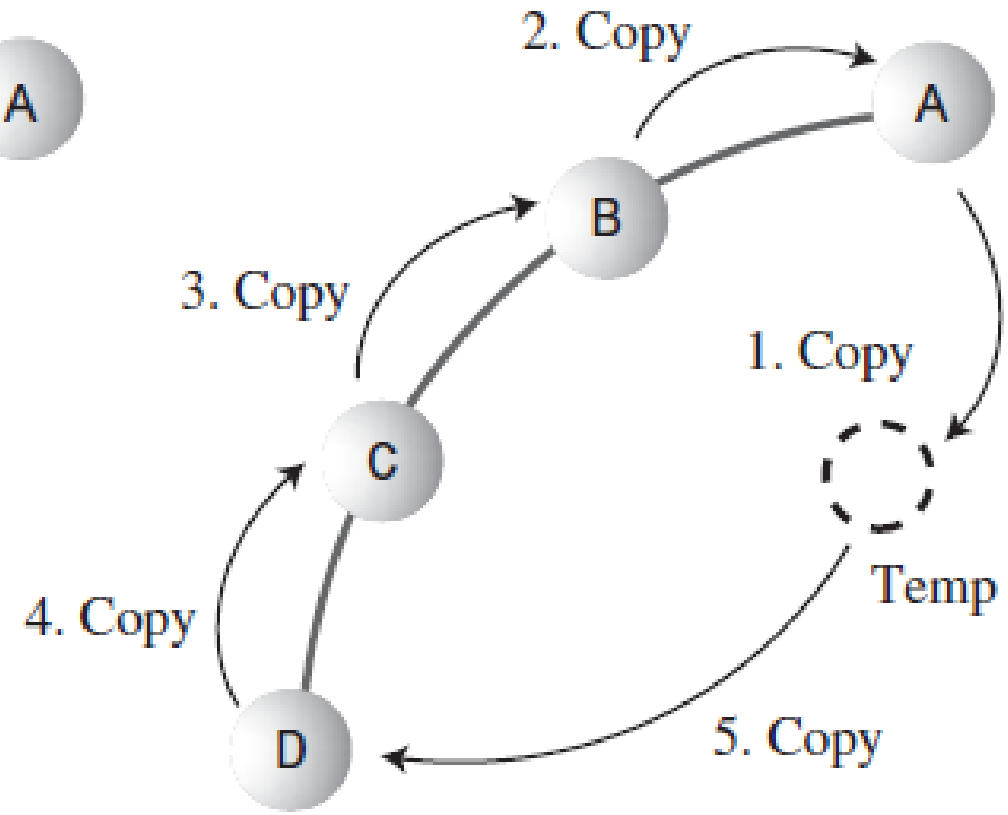
- Algoritmul de filtrare în sus este mai simplu decât cel de filtrare în jos, deoarece nu presupune comparația a doi fii
- Un nod are un singur părinte, iar nodul țintă va fi permutat cu părintele său

Observații

- Ștergând un anumit nod și apoi reinserându-l, nu se obține neapărat movila inițială
- O mulțime dată de noduri poate fi dispusă ca movilă în mai multe moduri, în funcție de ordinea în care nodurile sunt inserate



a) Swaps



b) Copies

Observații

- O permutare necesită trei copieri, deci trei permutări vor presupune efectuarea a nouă copieri
- Numărul total de copieri necesar într-o operație de filtrare poate fi redus, înlocuind permutările cu copieri

Observații

- Se pot efectua trei permutări, utilizând numai cinci copieri
- Mai întâi, nodul A este salvat temporar
- B este apoi copiat in A, C în B, iar D în locul lui C
- În final, A este copiat din variabila temporară, în locul lui D

Observații

- Timpul economisit utilizând copieri în locul permutărilor crește la fiecare nivel, datorită celor două copieri mai puțin
- La un număr mare de niveluri, numărul de copieri scade de aproape trei ori

Algoritmul HeapSort

- Eficiența structurii de movilă conduce la posibilitatea de a implementa un algoritm de sortare eficient, numit **heapsort**
- Se inserează elementele neordonate într-o movilă
- La final, se obține un tablou sortat crescător

Algoritmul HeapSort

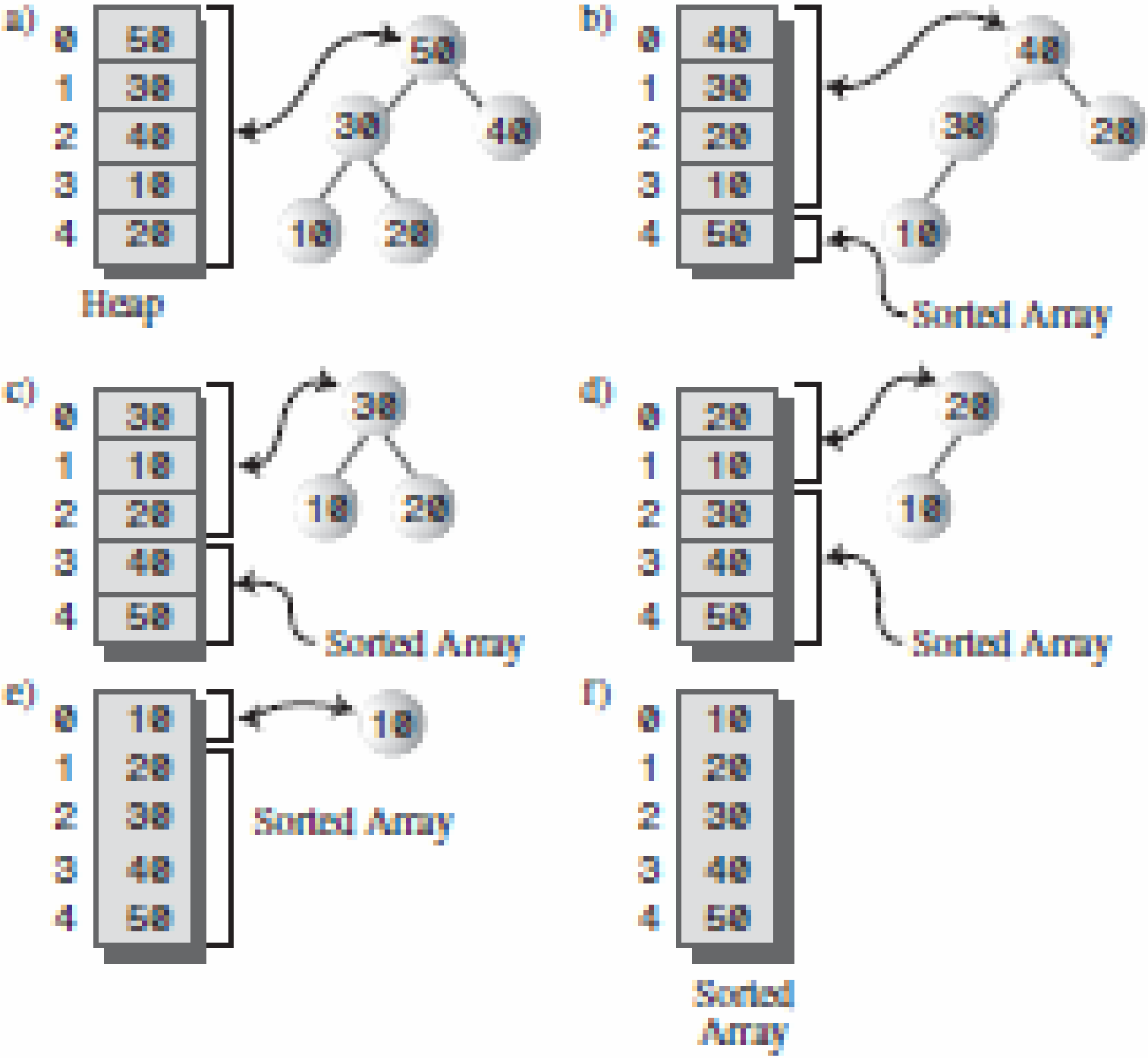
- Algoritmul pornește cu un tablou neordonat, după care elementele din tablou sunt inserate într-o movilă
- Apoi elementele sunt eliminate din movilă și scrise la loc în tablou, în ordine sortată

Algoritmul HeapSort

- De fiecare dată când se șterge un element din movilă, celula de la sfârșitul tabloului se eliberează
- Movila se diminuează cu o poziție
- Se poate pune elementul șters în această celulă eliberată recent

Algoritmul HeapSort

- Pe măsură ce se șterg elemente, movila devine din ce în ce mai mică, iar porțiunea ordonată din tablou devine din ce în ce mai mare
- Este posibil ca tabloul ordonat și movila să fie memorate în același spațiu


a)
0 50
1 38
2 48
3 18
4 28
50
38
48
18
28
Heap
b)
0 48
1 38
2 28
3 18
4 50
48
38
28
18
Sorted Array
c)
0 38
1 18
2 28
3 48
4 50
38
18
28
Sorted Array
d)
0 28
1 18
2 38
3 48
4 50
28
18
Sorted Array
e)
0 18
1 28
2 38
3 48
4 50
18
Sorted Array
f)
0 18
1 28
2 38
3 48
4 50
Sorted Array

Algoritmul HeapSort

```
void heapsort (int a[],int n) {
 int i, t;  heap h;
 h.n = n;      // copiază în movilă valorile din tabloul a
 for (i = 0;i < n; i++)
   h.v[i] = a[i];
 makeheap(h);  // aducere tablou la structura de movilă
 for (i = h.n - 1; i >= 1; i--) {
   t = h.v[0];  h.v[0]=h.v[h.n - 1];  h.v[h.n - 1] = t;
   h.n--;
   heapify(h, 0);
 }
 for (i = 0; i < n; i++)        // scoate din movilă în tabloul a
    a[i] = h.v[i];   }
```

Exemplu

După citire tablou	4,2,6,1,5,3
După makeheap	6,5,4,1,2,3
După schimbare 6 cu 3	3,5,4,1,2,6
După heapify(1,5)	5,3,4,1,2,6
După schimbare 5 cu 2	2,3,4,1,5,6
După heapify(1,4)	4,3,2,1,5,6
După schimbare 4 cu 1	1,3,2,4,5,6
După heapify(1,3)	3,1,2,4,5,6
După schimbare 3 cu 2	2,1,3,4,5,6
După heapify(1,2)	2,1,3,4,5,6
După schimbare 2 cu 1	1,2,3,4,5,6

Eficiența algoritmului HeapSort

- Algoritmul heapsort se execută într-un timp O(N*logN)
- Deși este, de regulă, ceva mai lent decât sortarea rapidă (quicksort), un avantaj este cel al unei relative independențe față de distribuția inițială a elementelor

Eficiența algoritmului HeapSort

- Există anumite aranjamente ale elementelor, pentru care sortarea rapidă (quicksort) are un timp de execuție de ordinul $O(N^2)$, în timp ce algoritmul heapsort rulează într-un timp $O(N*logN)$, indiferent de distribuția inițială a elementelor

Concluzii

- Într-o coadă cu priorități ordonată crescător, elementul cu cheia maximă are cea mai mare prioritate
- Movilele asigură o implementare eficientă pentru cozile cu priorități
- Într-o movilă, ștergerea elementului maxim și inserarea unui element nou se pot executa într-un timp de ordinul O(logN)

Concluzii

- Elementul maxim se găsește întotdeauna în rădăcină
- Movilele nu oferă posibilitatea de parcurgere a elementelor în ordine, de căutare a unui element cu o cheie dată și de ștergere a unui astfel de element

Concluzii

- O movilă se implementează printr-un tablou, care reprezintă un arbore binar complet
- Rădăcina are indicele 0, iar ultimul element are indicele N-1
- Fiecare nod are o cheie mai mică decât părinții săi și mai mare decât fiii săi

Concluzii

- Un element care este inserat este plasat întotdeauna în prima celulă liberă din tablou, după care este filtrat în sus, până ajunge în poziția corectă
- La ștergerea unui element care reprezintă rădăcina, elementul șters este înlocuit de ultimul nod din tablou, care este apoi filtrat în jos, până în poziția curentă

Concluzii

- Operațiile de filtrare în sus sau în jos pot fi gândite ca secvențe de permutări, dar se implementează mult mai eficient prin câte o secvență de copieri

Concluzii

- Prioritatea unui element arbitrar se poate modifica
- Mai întâi, se modifică cheia elementului
- Dacă valoarea a crescut, se filtrează elementul în sus
- Dacă valoarea a scăzut, se filtrează elementul în jos

Concluzii

- Algoritmul heapsort reprezintă o metodă de sortare eficientă, cu timpul de ordinul O(N*logN)
- Din punct de vedere conceptual, algoritmul heapsort efectuează N inserări într-o movilă, urmate de N extrageri