Structuri de Date și Algoritmi
Arbori binari

Mihai Nan

Departamentul de Calculatoare
Facultatea de Automatică și Calculatoare
Universitatea POLITEHNICA din București

Anul Universitar 2022–2023

Conținutul cursului

Structuri ierarhice

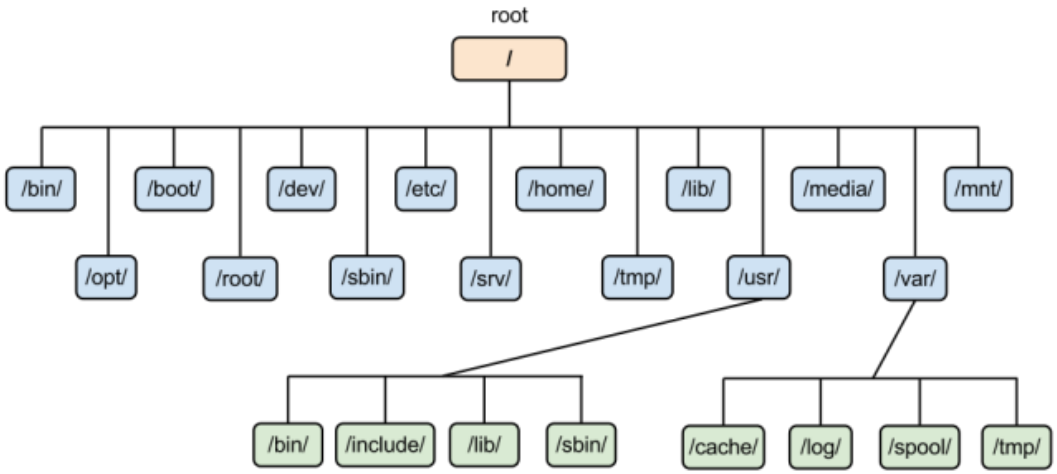
O structură de date este considerată liniară dacă pentru elementele stocate poate fi definită o relație de tip succesor / predecesor (sau ambele) cu proprietatea că fiecare element al structurii are un unic succesor / predecesor direct.

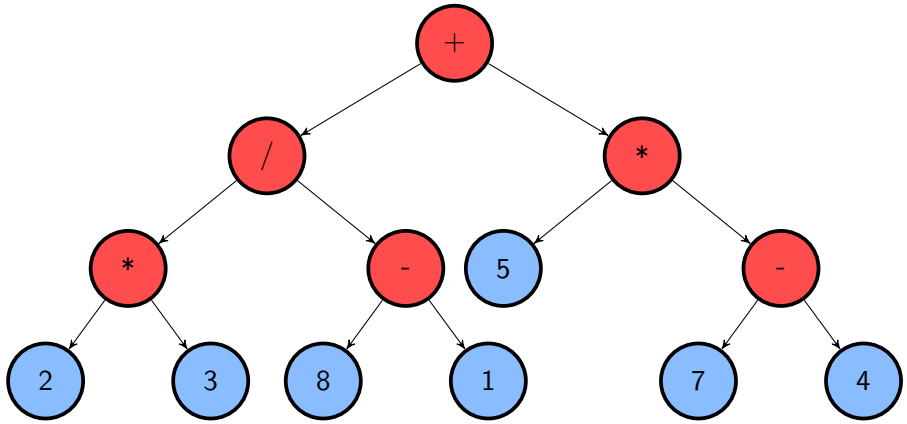
- Exemple de structuri de date liniare: vectori, liste, stive, cozi

Arborele este o structură de date neliniară. De această dată avem o structură ierarhică / arborescentă.

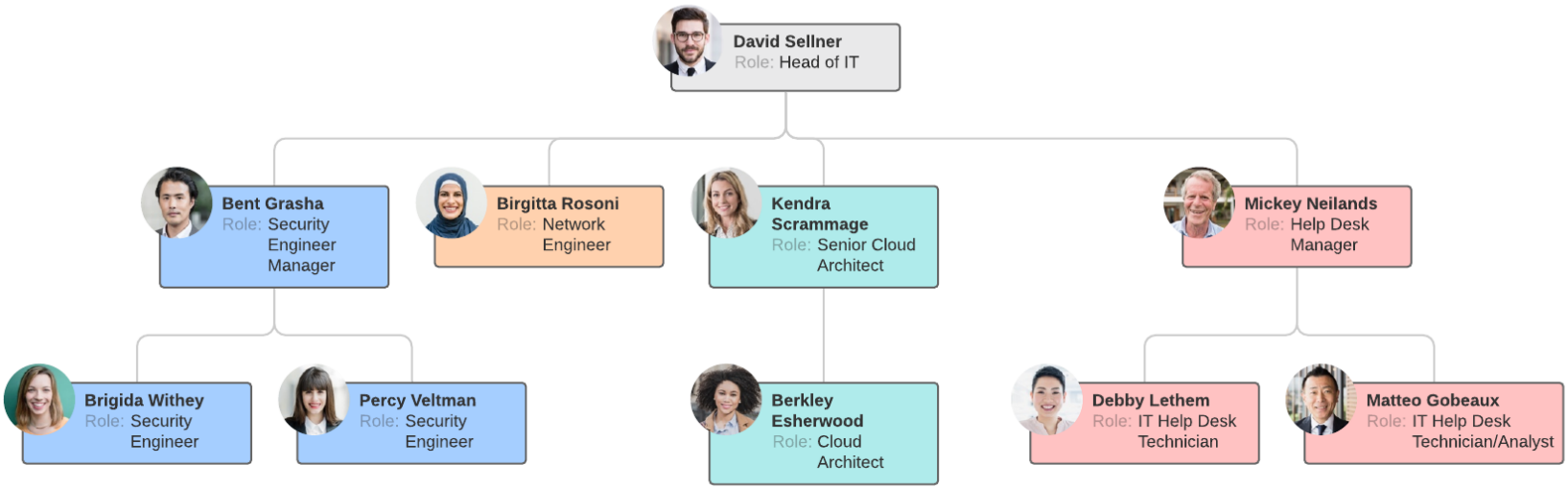
- În practică avem multe exemple de date care pot fi structurate convenabil sub formă de arbore:
  - organizarea administrativă a societăților comerciale, ministerelor, universităților etc.
  - programul meciurilor dintr-un turneu eliminatoriu;
  - directoare și subdirectoare;
  - arborele unei expresii aritmetice.

Exemple de date modelate ca arbori





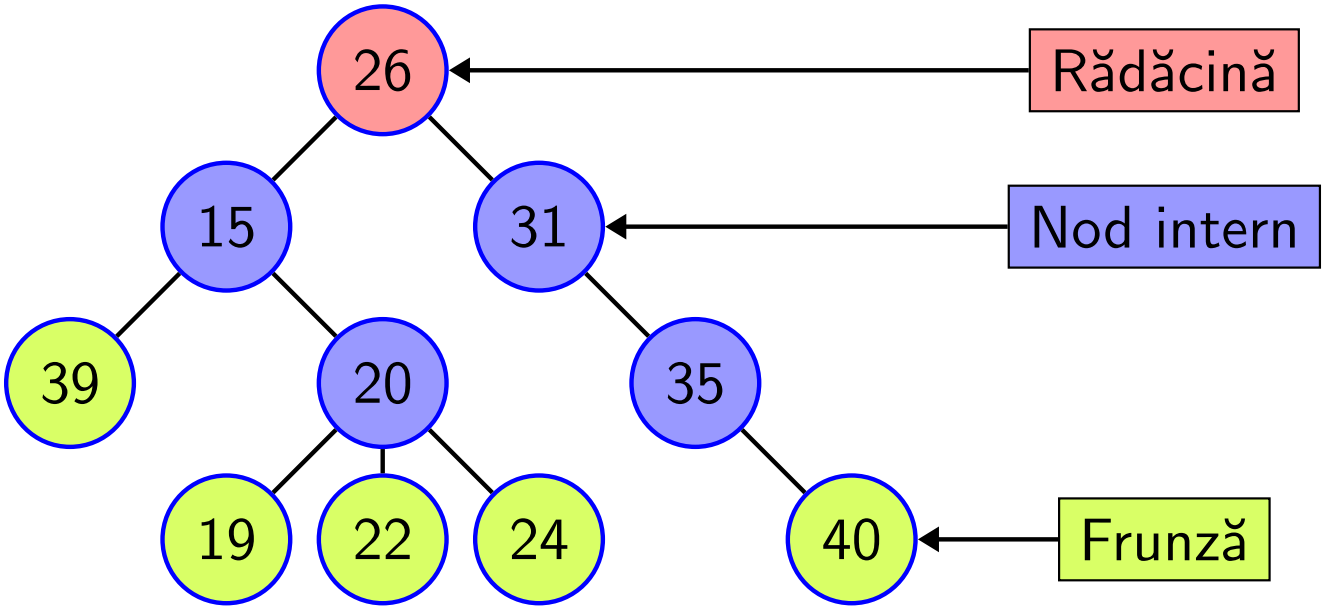
$$(2 * 3)/(8 - 1)+5*(7-4)$$



Arbori – Introducere

Un **arbore** este o colecție de noduri și arce (legături între noduri) în care există o singură cale între oricare două noduri.

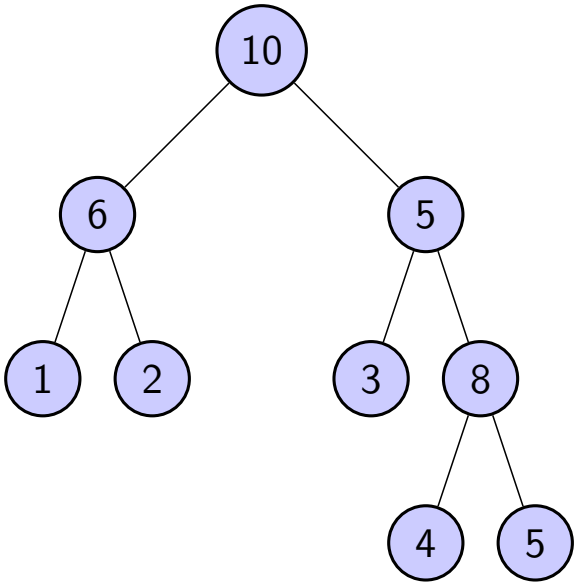
- **Cale** – secvență de noduri și arce care conectează aceste noduri.
- **Părintele** unui nod este acel nod care se află imediat deasupra nodului.
- **Rădăcina** unui arbore este acel nod unic, care nu are niciun părinte.
- **Frunzele** unui arbore sunt acele noduri care nu au copii.



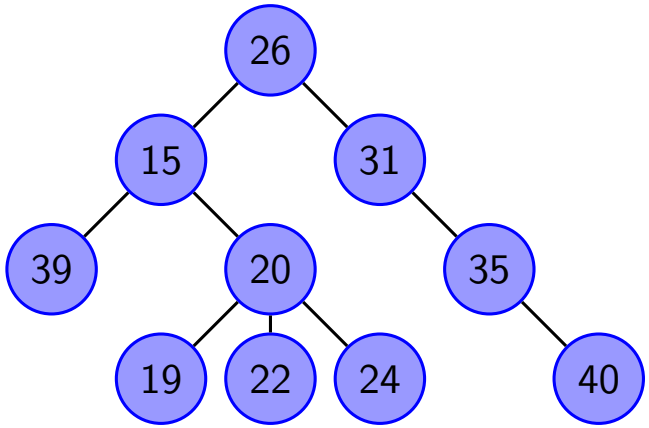
Tipuri de arbori

- Există mai multe tipuri de arbori:
  - Arbori binari
  - Arbor multicăi sau n-ari
  - Arbori sortați sau de căutare

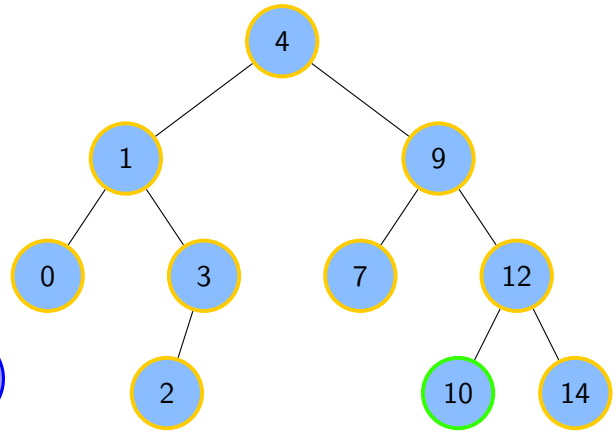
Arbore binar



Arbore multicăi



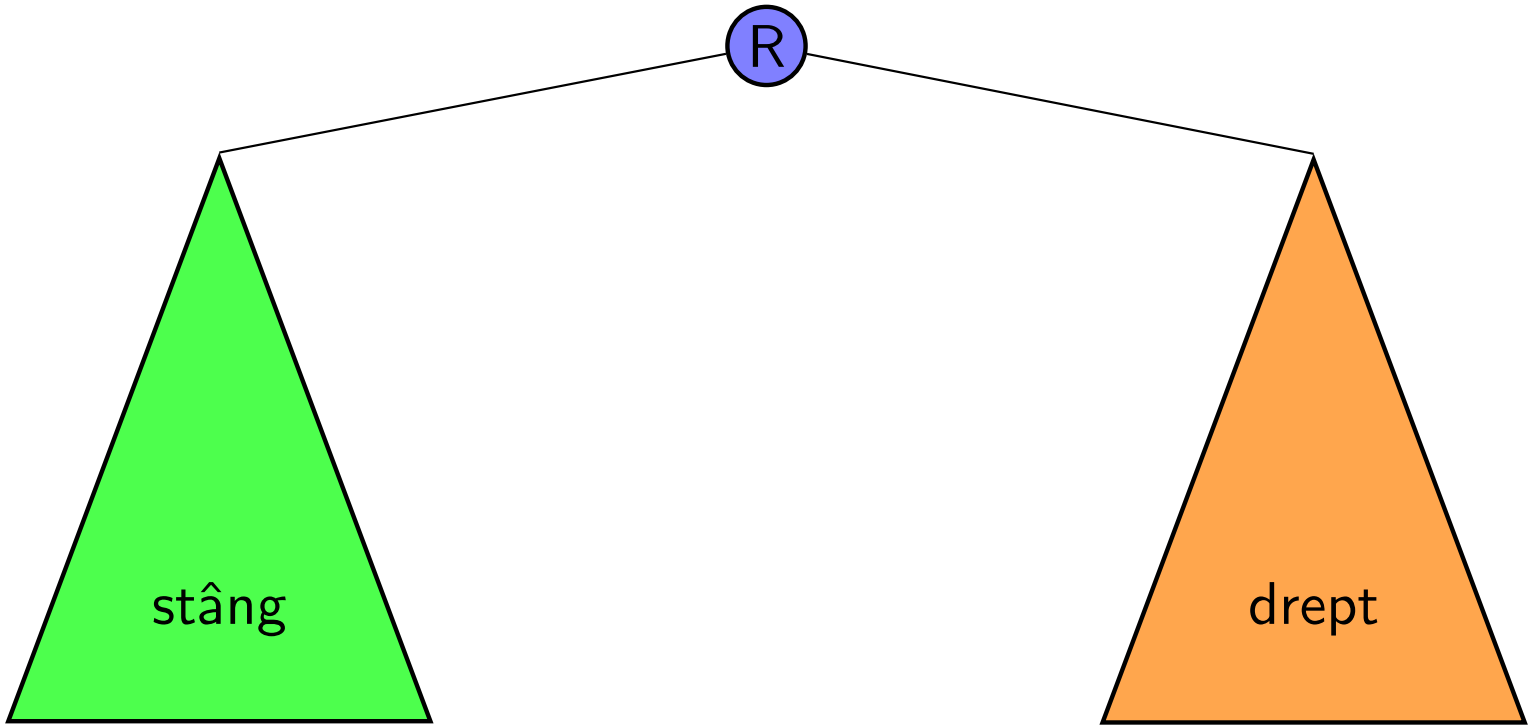
Arbore de căutare



- Un arbore este format dintr-un nod rădăcină, căruia îi este atașat un număr finit de arbori. Pentru a evidenția relația ierarhică, aceștia sunt numiți sub-arbori.

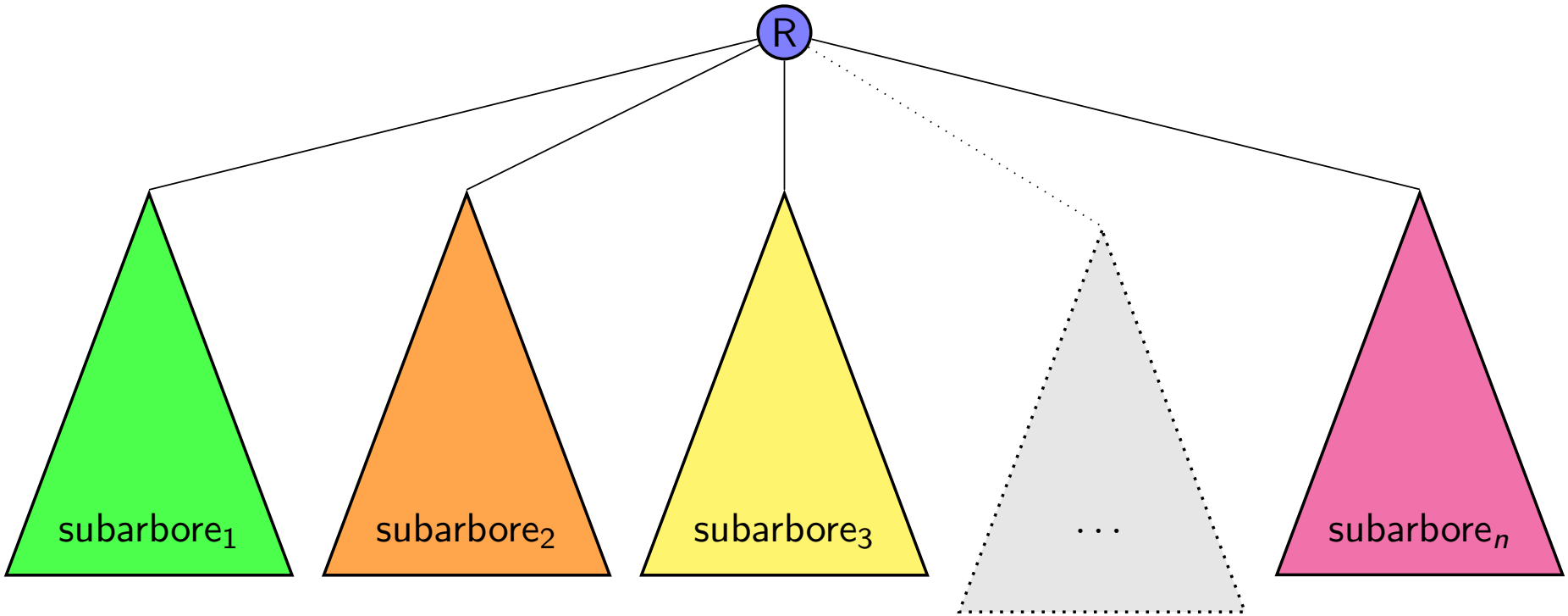
Arbori binari

1. **Arborele vid** – nu conține nici un nod
2. **Arbore binar** – nodurile au cel mult 2 sub-arbori
   1. sub-arbore stâng
   2. sub-arbore drept



Arbori multicăi

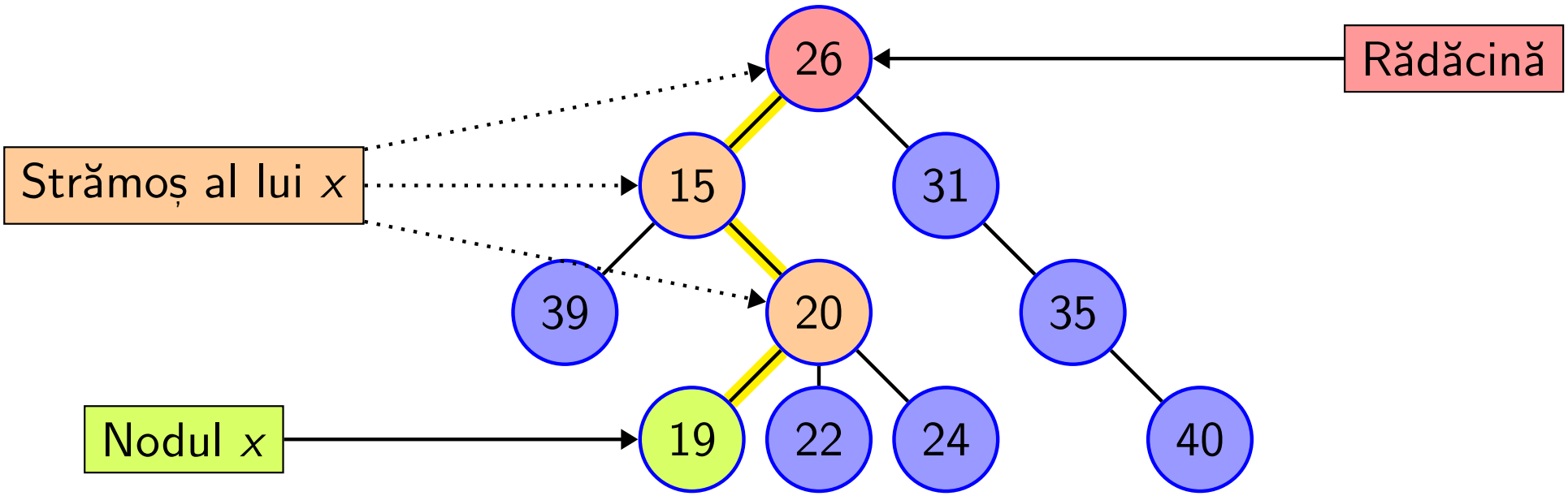
Arbore multicăi – arbore în care nodurile pot avea mai mult de 2 sub-arbori



Arbori – Terminologie

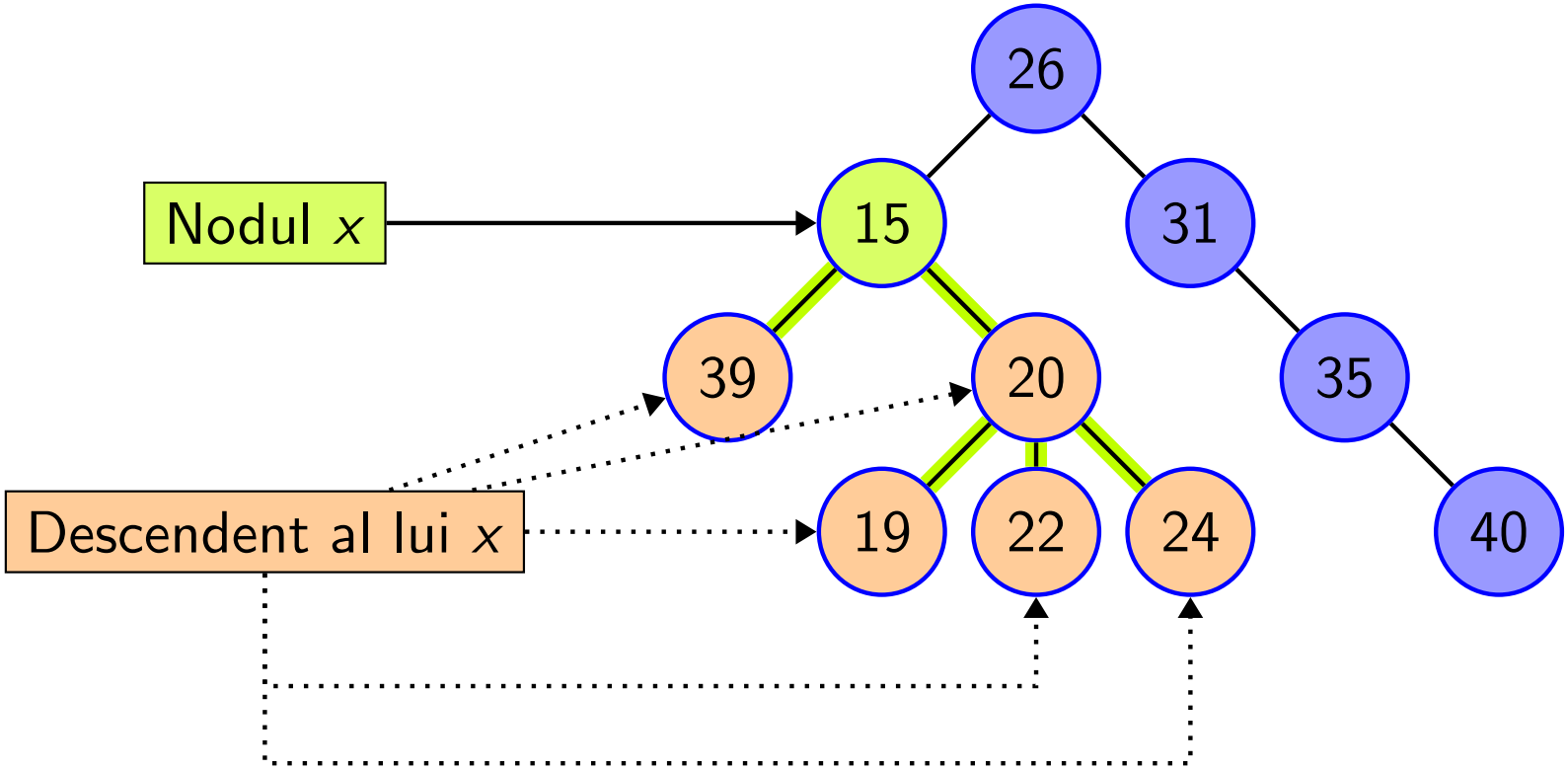
- **Strămoș** al unui nod x – orice nod aflat pe calea (unică) de la rădăcină până la nodul x.

Rădăcina este strămoșul tuturor celorlalte noduri!



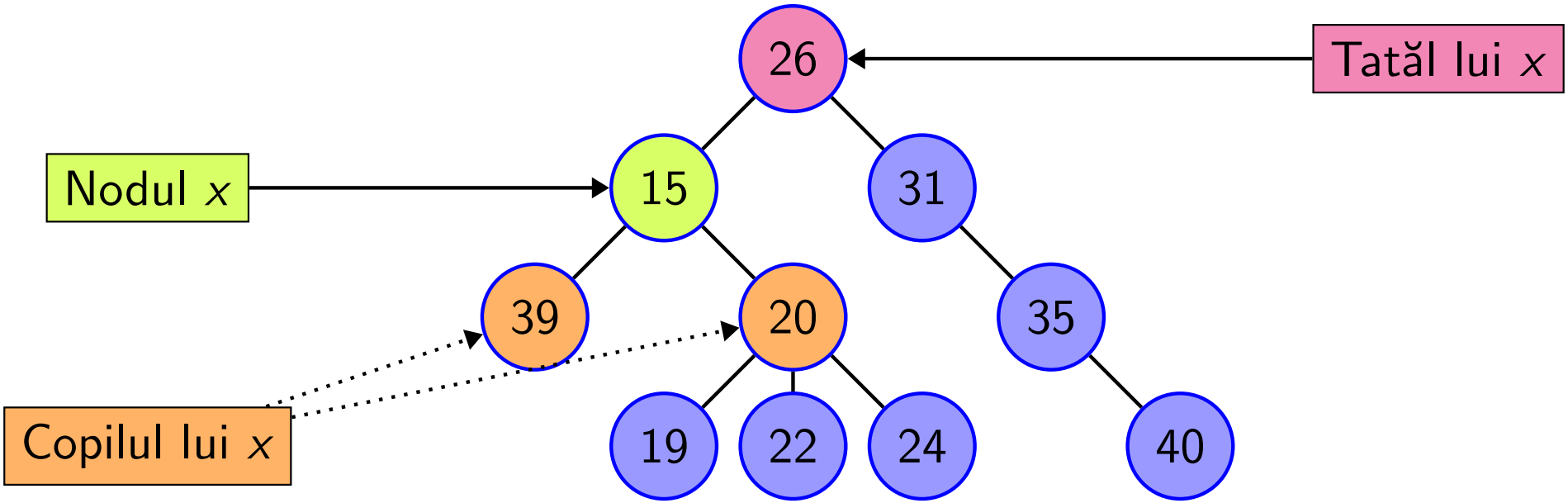
Arbori – Terminologie

- **Descendent** al unui nod x — orice nod aflat în arborele cu rădăcina x.



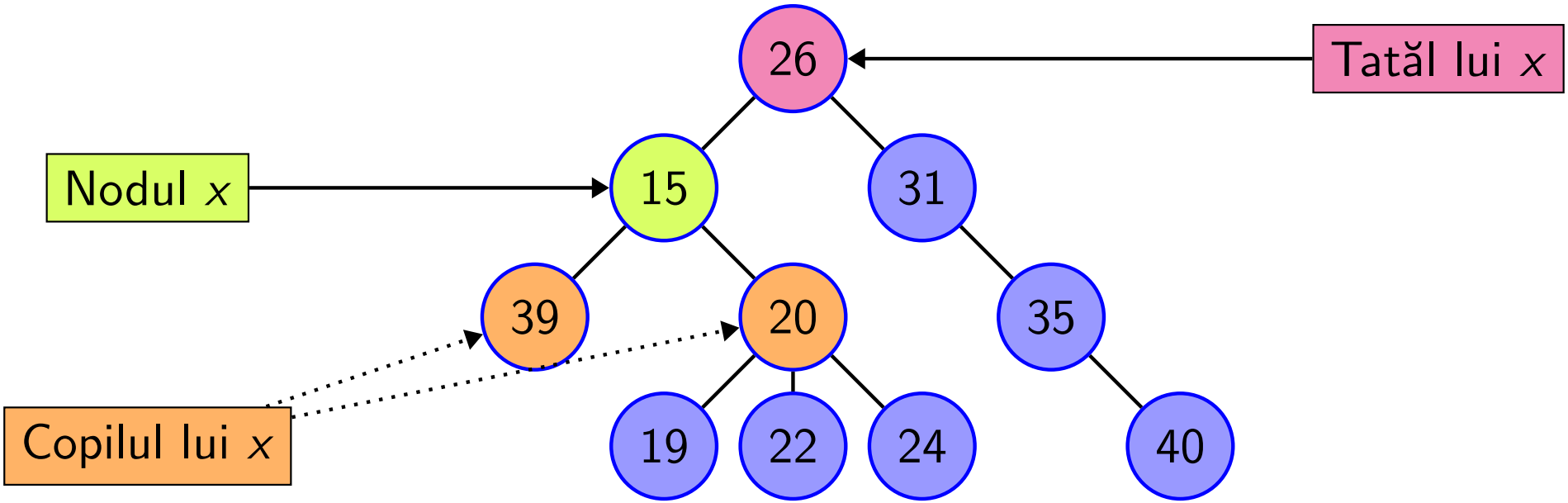
Arbori – Terminologie

- **Tată** / **Fiu** (copil) – noduri aflate la distanța 1 (numite și strămoș / descendent direct).



Arbori – Terminologie

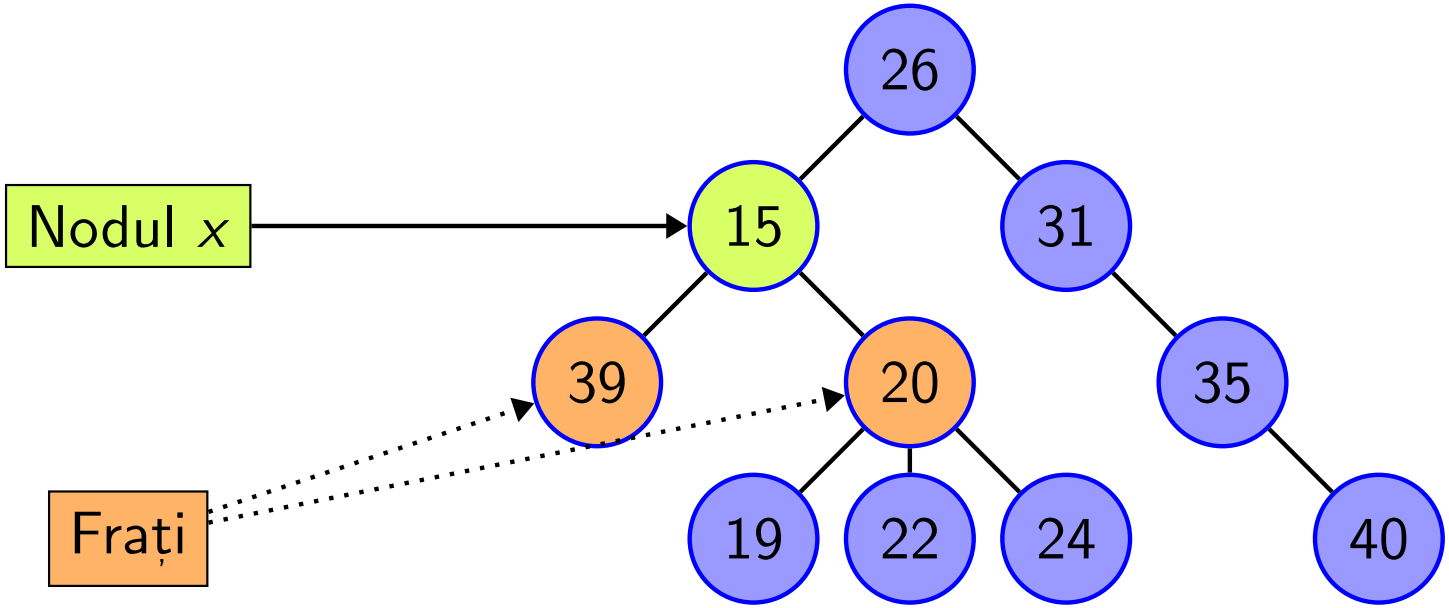
- **Tată** / **Fiu** (copil) – noduri aflate la distanța 1 (numite și strămoș / descendent direct).



Toate nodurile din arbore, cu excepția rădăcinii, au un unic tată!

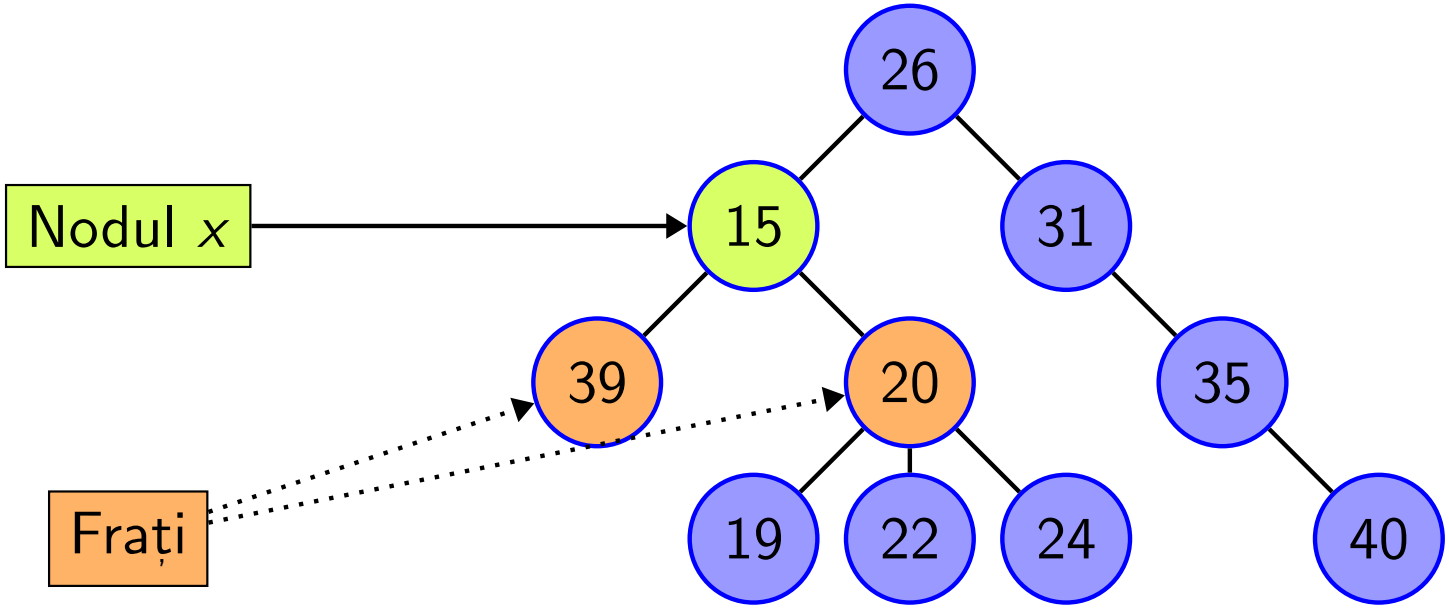
Arbori – Terminologie

- **Frați** (siblings) – fiii aceluiași nod.



Arbori – Terminologie

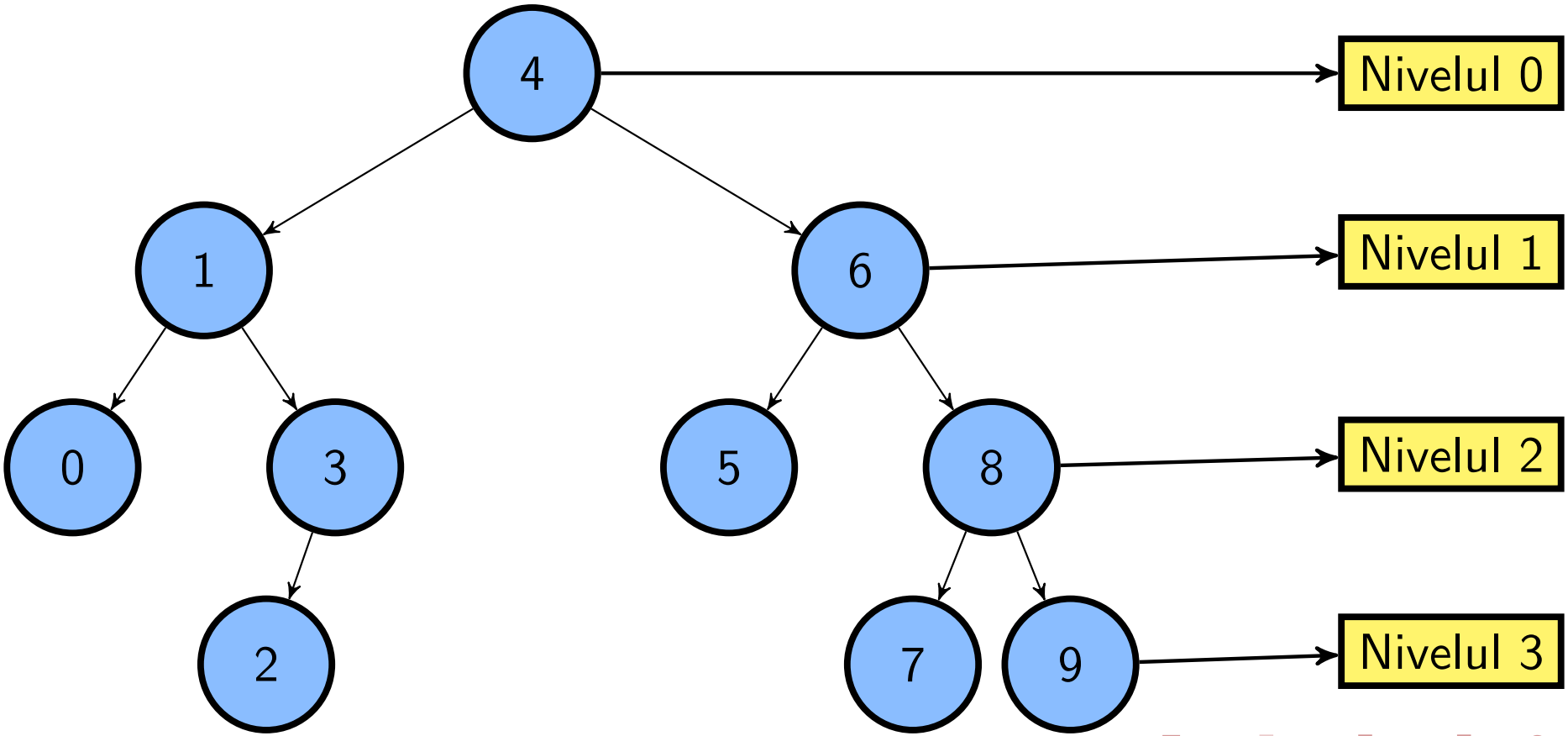
- **Frați** (siblings) – fiii aceluiași nod.



- **Ordinul** (gradul) unui nod – numărul de sub-arbori atașați nodului.
- **Noduri interne** – nodurile care au sub-arbori.
- **Noduri externe** (noduri terminale sau **frunze**) – nodurile fără sub-arbori (cu ordin 0).

Arbori – Terminologie

- **Nivelul** sau **adâncimea** unui nod – numărul de arce de la rădăcină la nod (rădăcina are nivel 0).
- **Înălțimea** unui arbore – nivelul maxim din arbore (adâncimea frunzei de nivel maxim).



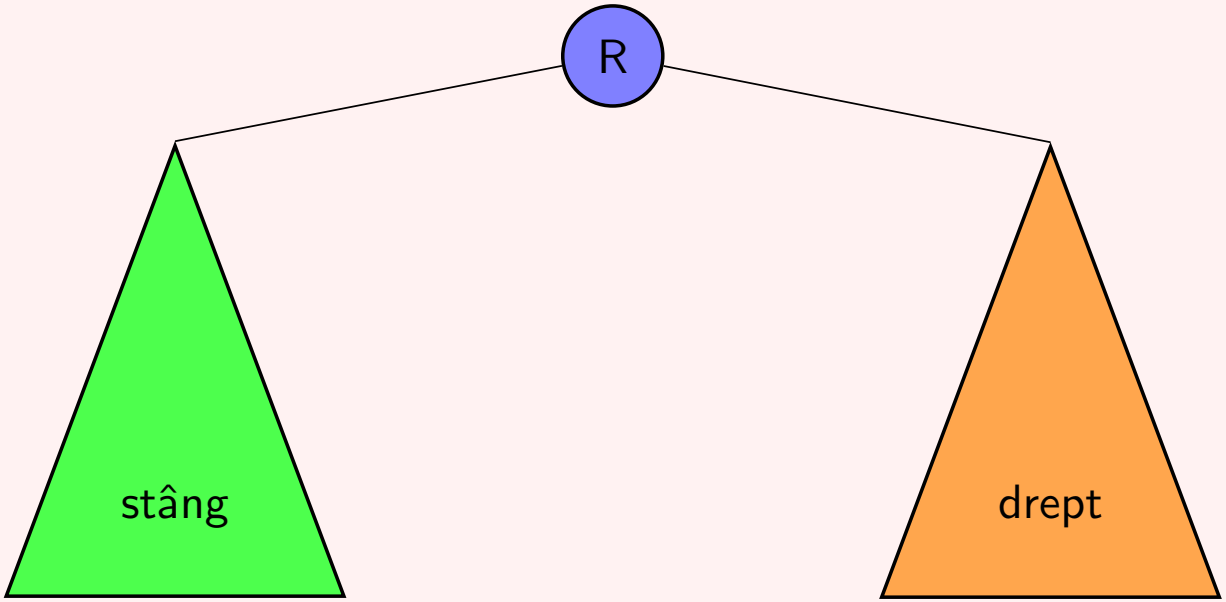
Arbori binari – Definiție

Definiția recursivă

Un **arbore binar** nevid este:

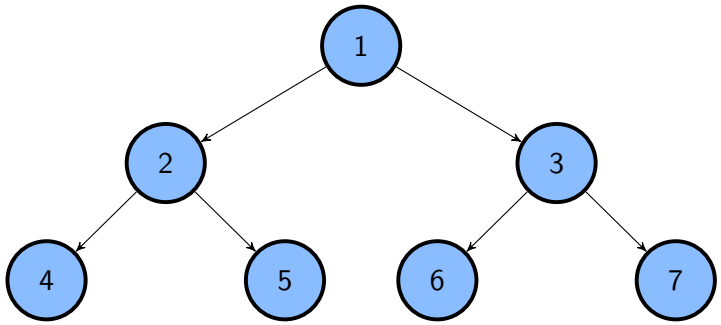
1. un nod extern;

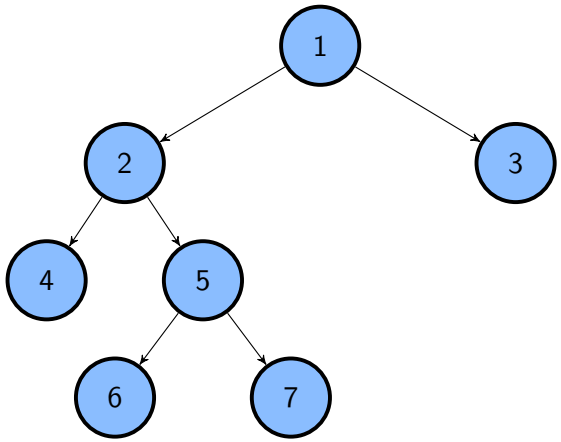
2. un nod intern conectat la o pereche de sub-arbori numiți sub-arbore stâng și sub-arbore drept.

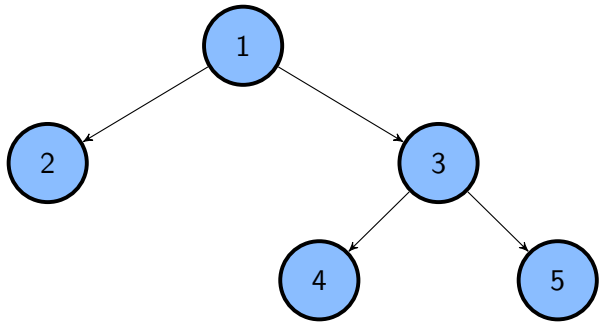


Arbori binari – Proprietăți

1. **Arbore binar plin** – fiecare nod are exact 0 sau 2 fii.

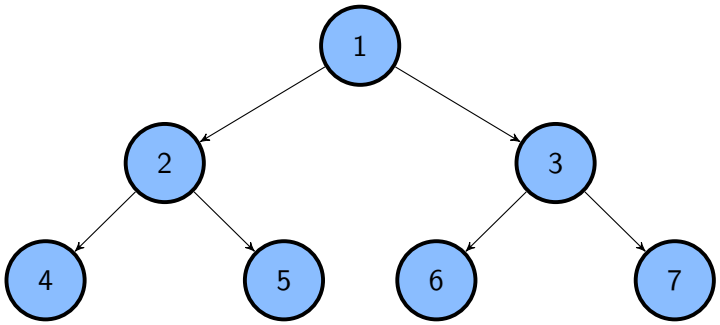


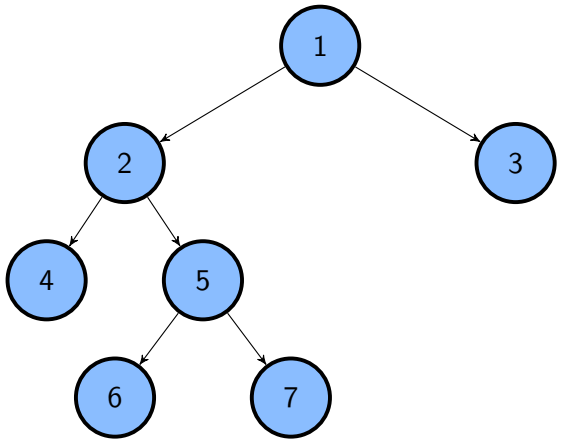


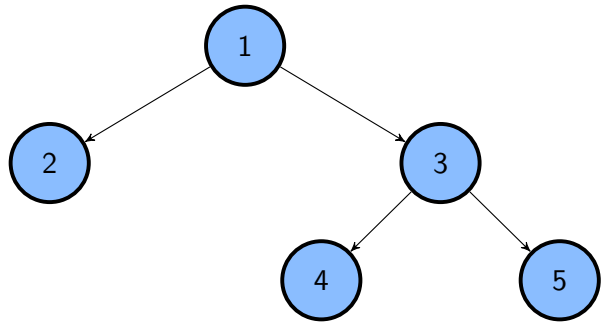


Arbori binari – Proprietăți

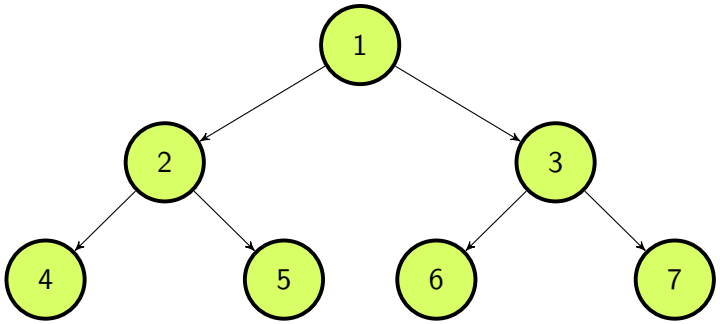
1. **Arbore binar plin** – fiecare nod are exact 0 sau 2 fii.

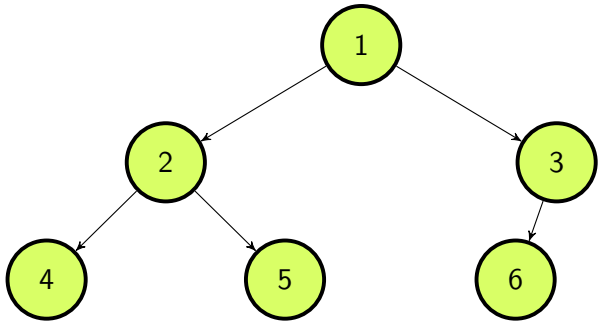


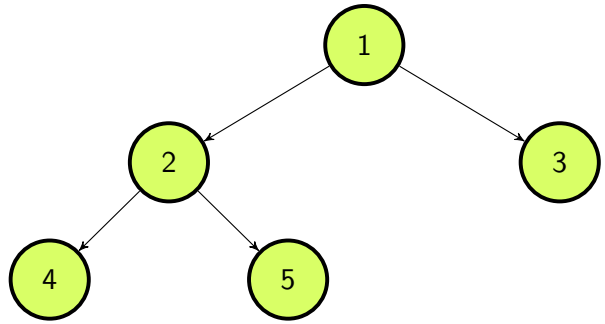




2. **Arbore binar complet** – un arbore binar care este complet umplut cu posibila excepție a ultimului nivel care este umplut de la stânga la dreapta.







Arbori binari – Proprietăți

Teoremă

Fie **T** un arbore binar complet nevid. Atunci:

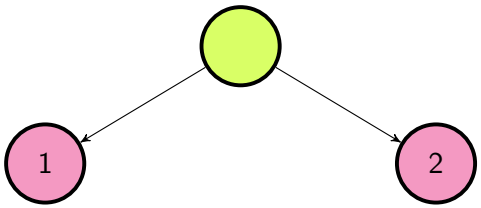
1. Dacă **T** are I noduri interne, numărul de frunze este $L = I + 1$.
2. Dacă **T** are I noduri interne, numărul total de noduri este $N = 2 \cdot I + 1$.
3. Dacă **T** are un total de N noduri, numărul de noduri interne este $I = (N–1)/2$.
4. Dacă **T** are un total de N noduri, numărul de frunze este $L = (N + 1)/2$.
5. Dacă **T** are L frunze, numărul total de noduri este $N = 2 \cdot L–1$.
6. Dacă **T** are L frunze, numărul de noduri interne este $I = L–1$.

Arbori binari – Proprietăți

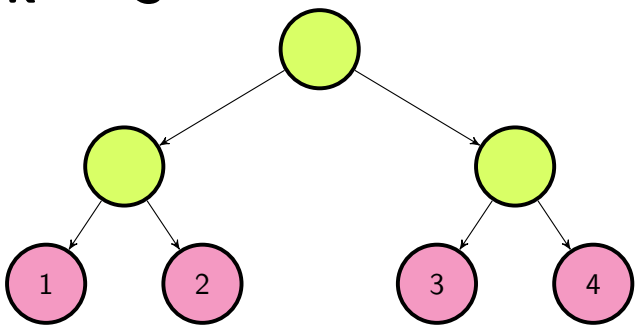
Teoremă

Fie **T** un arbore binar cu k niveluri. Atunci numărul de frunze este de cel mult 2^{k-1}.

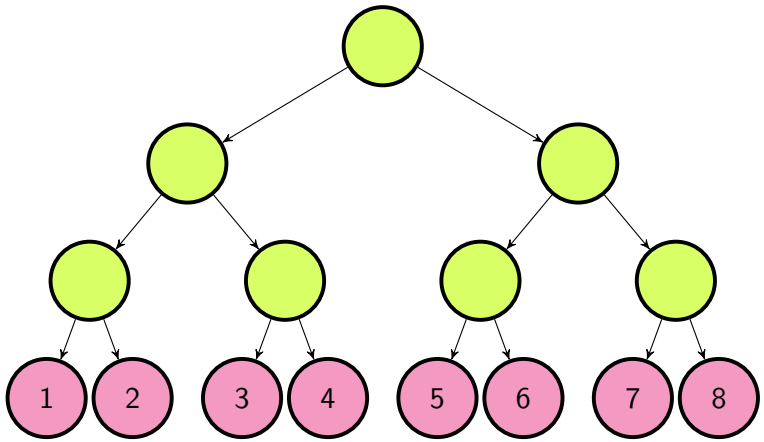
k = 2



k = 3



k = 4

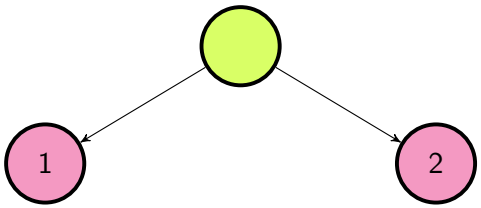


Arbori binari – Proprietăți

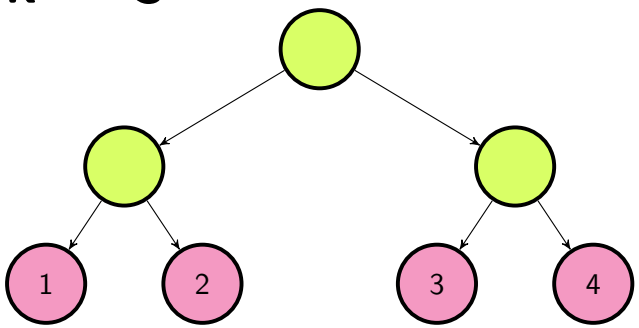
Teoremă

Fie **T** un arbore binar cu k niveluri. Atunci numărul de frunze este de cel mult 2^{k-1}.

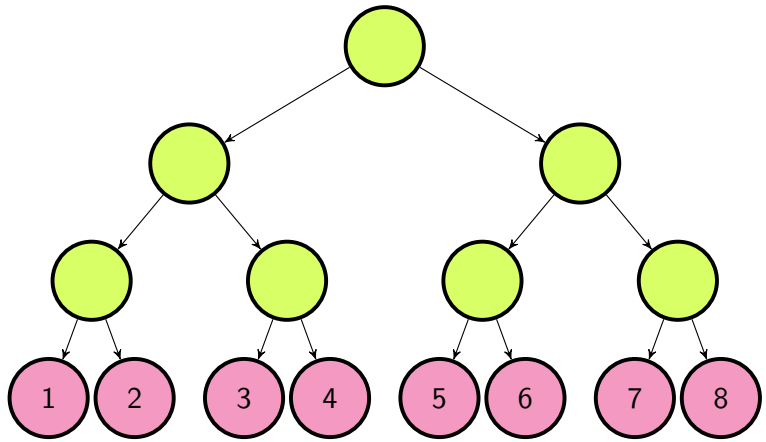
k = 2



k = 3



k = 4



Teoremă

Fie **T** un arbore binar. Pentru fiecare $k \geq 0$, nu există mai mult de 2^k noduri la nivelul k.

Arbori binari – Proprietăți

Teoremă

Fie **T** un arbore binar cu k niveluri. Atunci **T** nu are mai mult de $2^k - 1$ noduri.

Arbori binari – Proprietăți

Teoremă

Fie **T** un arbore binar cu k niveluri. Atunci **T** nu are mai mult de $2^k - 1$ noduri.

Teoremă

Fie **T** un arbore binar cu N noduri. Atunci numărul de niveluri este cel puțin $\lceil \log(N+1) \rceil$.

Arbori binari – Proprietăți

Teoremă

Fie **T** un arbore binar cu k niveluri. Atunci **T** nu are mai mult de $2^k - 1$ noduri.

Teoremă

Fie **T** un arbore binar cu N noduri. Atunci numărul de niveluri este cel puțin $\lceil \log(N+1) \rceil$.

Teoremă

Fie **T** un arbore binar cu L frunze. Atunci numărul de niveluri este cel puțin $\lceil \log L \rceil + 1$.

Arbori binari – Proprietăți

Teoremă

Un **arbore binar plin** cu N noduri interne are $N + 1$ noduri externe.

Arbori binari – Proprietăți

Teoremă

Un **arbore binar plin** cu N noduri interne are $N+1$ noduri externe.

Teoremă

Un arbore binar plin cu N noduri interne are $2N$ arce/legături: $N-1$ legături între nodurile interne și $N+1$ legături la nodurile externe.

Arbori binari – Proprietăți

Teoremă

Un **arbore binar plin** cu N noduri interne are $N+1$ noduri externe.

Teoremă

Un arbore binar plin cu N noduri interne are $2N$ arce/legături: $N-1$ legături între nodurile interne și $N+1$ legături la nodurile externe.

Teoremă

Înălțimea unui arbore binar cu N noduri interne este minim $\log N$ și maxim $N-1$.

Arbori binari – Proprietăți

Teoremă

Înălțimea unui arbore binar complet cu M noduri este cel mult $O(\log M)$.

- Un arbore binar complet este un arbore special deoarece oferă raportul optim între numărul de noduri și înălțimea arborelui.
- Înălțimea h a unui arbore binar complet cu M noduri este cel mult $O(\log M)$

$$M = 1 + 2 + 4 + \cdots + 2^{h-1} + 2^h = 2^{h+1} - 1$$

- Rezolvând pentru h avem $h = O(\log M)$.

Definirea TAD-ului pentru un arbore binar

Constructori

- Aceștia au ca rezultat un arbore binar cu elemente de tip T.
- Considerăm ca nume pentru TAD `TBinaryTree`.
  1. Inițializarea arborelui: `initBinaryTree:` $\rightarrow$ `TBinaryTree`
  2. Construcția unui arbore cu un singur nod (ce are asociată o valoare): `createBinaryTree:` $T \rightarrow$ `TBinaryTree`
  3. Inserarea unui element: `insert:` $T \times$ `TBinaryTree` $\rightarrow$ `TBinaryTree`
  4. Eliminarea unui element: `remove:` $T \times$ `TBinaryTree` $\rightarrow$ `TBinaryTree`

Funcții

- Operați care furnizează informații despre un arbore binar.
  1. Verificarea arbore vid: `isEmpty:` `TBinaryTree` $\rightarrow \{0, 1\}$
  2. Determinarea înălțimii: `height:` `TBinaryTree` $\rightarrow$ `Int`
  3. Determinarea numărului de noduri: `size:` `TBinaryTree` $\rightarrow$ `Int`

Definirea TAD-ului pentru un arbore binar

Constructori

- Aceștia au ca rezultat un arbore binar cu elemente de tip T.
- Considerăm ca nume pentru TAD `TBinaryTree`.
  1. Inițializarea arborelui: `initBinaryTree:` $\rightarrow$ `TBinaryTree`
  2. Construcția unui arbore cu un singur nod (ce are asociată o valoare): `createBinaryTree:` $T \rightarrow$ `TBinaryTree`
  3. Inserarea unui element: `insert:` $T \times$ `TBinaryTree` $\rightarrow$ `TBinaryTree`
  4. Eliminarea unui element: `remove:` $T \times$ `TBinaryTree` $\rightarrow$ `TBinaryTree`

Funcții

- Operați care furnizează informații despre un arbore binar.
  1. Verificarea arbore vid: `isEmpty:` `TBinaryTree` $\rightarrow \{0, 1\}$
  2. Determinarea înălțimii: `height:` `TBinaryTree` $\rightarrow$ `Int`
  3. Determinarea numărului de noduri: `size:` `TBinaryTree` $\rightarrow$ `Int`

Ce putem utiliza pentru a reține elementele unui arbore binar?

Reprezentarea TAD-ului prin doi vectori

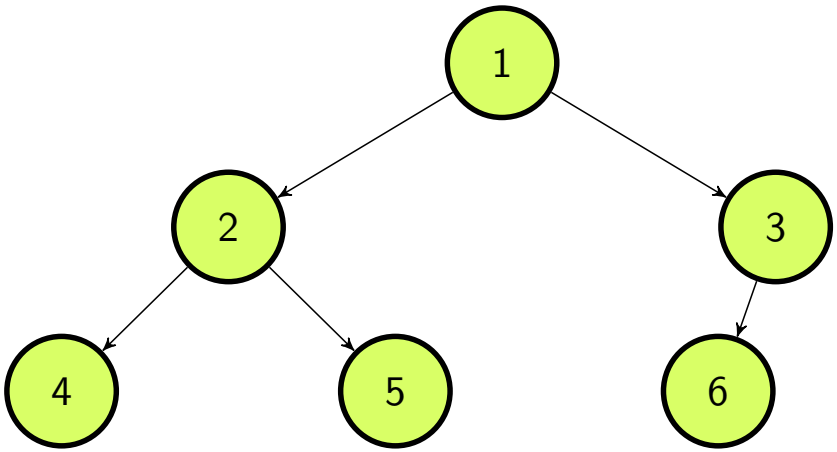
Un arbore binar poate fi reprezentat în memorie prin doi vectori (`S` si `D` cu N elemente).

1. `S[i]` – succesorul stâng al nodului `i`
2. `D[i]` – succesorul drept al nodului `i`

Reprezentarea TAD-ului prin doi vectori

Un arbore binar poate fi reprezentat în memorie prin doi vectori (`S` si `D` cu N elemente).

1. `S[i]` – succesorul stâng al nodului `i`
2. `D[i]` – succesorul drept al nodului `i`



S	0	1	2	3	4	5	6
	0	2	4	6	0	0	0

D	0	1	2	3	4	5	6
	0	3	5	0	0	0	0

Reprezentarea TAD-ului prin vector de tați

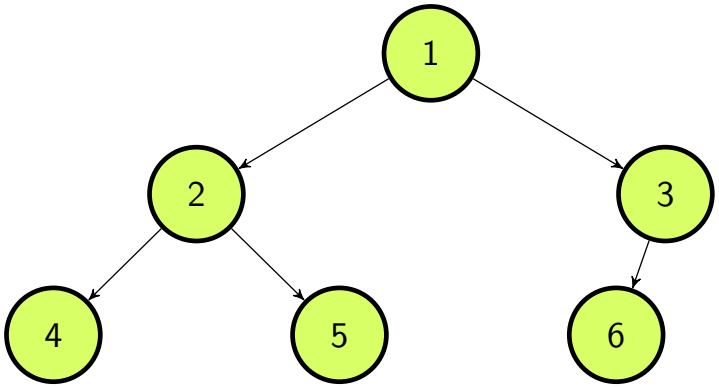
Pentru fiecare nod se memorează informații despre ascendenții direcți. Vom obține un **vector de tați**, în care:

1. `T[r] = 0` – unde `r` este rădăcina arborelui
2. `T[i]` = tatăl nodului `i`

Reprezentarea TAD-ului prin vector de tați

Pentru fiecare nod se memorează informații despre ascendenții direcți. Vom obține un **vector de tați**, în care:

1. `T[r] = 0` – unde `r` este rădăcina arborelui
2. `T[i]` = tatăl nodului `i`

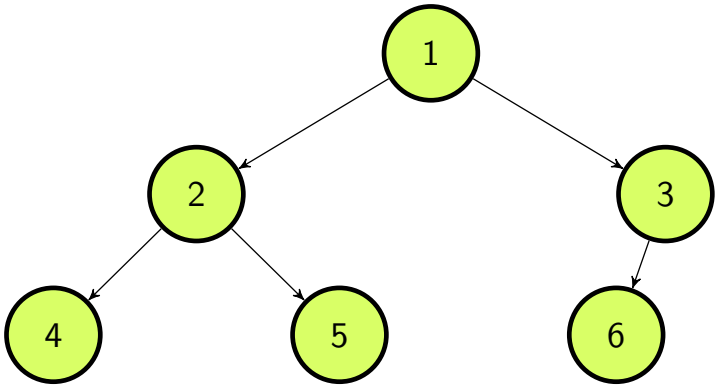


T	1	2	3	4	5	6
	0	1	1	2	2	3

Reprezentarea TAD-ului prin vector de tați

Pentru fiecare nod se memorează informații despre ascendenții direcți. Vom obține un **vector de tați**, în care:

1. `T[r] = 0` – unde `r` este rădăcina arborelui
2. `T[i]` = tatăl nodului `i`



T	1	2	3	4	5	6
	0	1	1	2	2	3

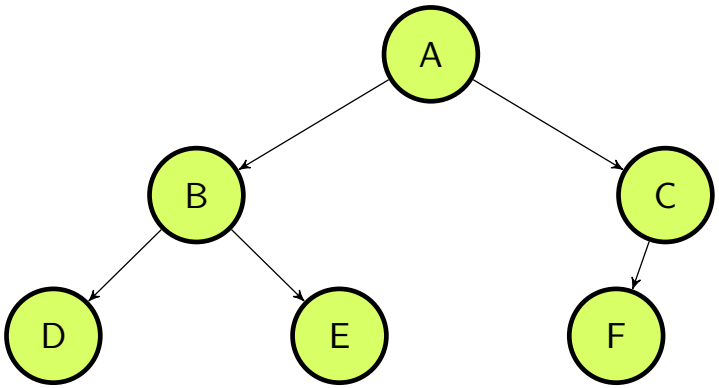
În vectorul de tați există o singură valoare 0, corespunzătoare rădăcinii, iar frunzele corespund valorilor care nu apar în vectorul de tați.

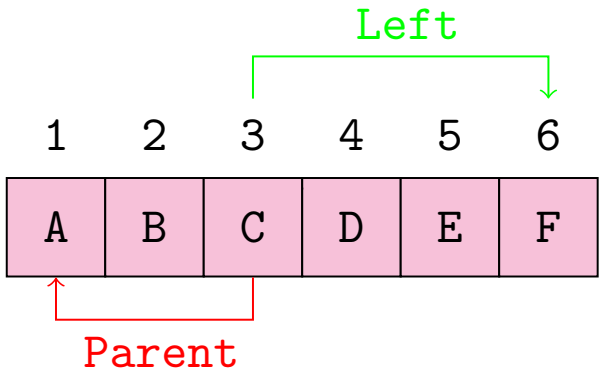
Reprezentarea TAD-ului utilizând un vector

Considerăm pentru rădăcină indicele $i = 1$. Nodul cu indice i are subarborii la pozițiile $2 \cdot i$ și $2 \cdot i + 1$.
Arborele de înălțime h are $2^{h+1} - 1$ celule în vector.
Dacă știu indexul unui nod i, obțin indexul tatălui i **div** 2.

Reprezentarea TAD-ului utilizând un vector

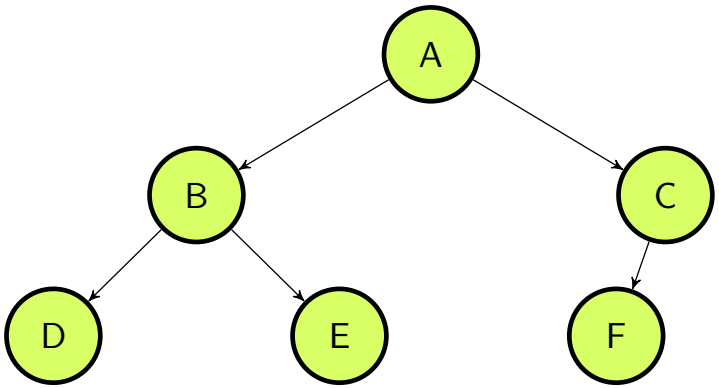
Considerăm pentru rădăcină indicele $i = 1$. Nodul cu indice i are subarborii la pozițiile $2 \cdot i$ și $2 \cdot i + 1$.
Arborele de înălțime h are $2^{h+1} - 1$ celule în vector.
Dacă știu indexul unui nod i, obțin indexul tatălui i **div** 2.

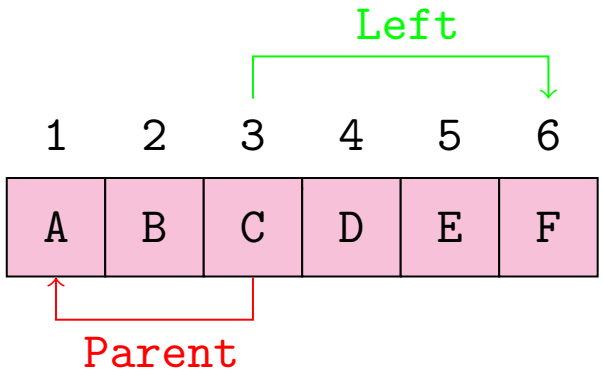




Reprezentarea TAD-ului utilizând un vector

Considerăm pentru rădăcină indicele $i = 1$. Nodul cu indice i are subarborii la pozițiile $2 \cdot i$ și $2 \cdot i + 1$.
Arborele de înălțime h are $2^{h+1} - 1$ celule în vector.
Dacă știu indexul unui nod i, obțin indexul tatălui i **div** 2.





Reprezentare potrivită în cazul în care arborele este construit de la început echilibrat și nu se mai modifică.

Reprezentarea arborilor binari

- Amintim reprezentările utilizate pentru liste
  1. Lista simplu înlănțuită

```
typedef int T;
typedef struct node {
        T value;
        struct node* next;
} Node, *TList;
```

  2. Lista dublu înlănțuită

```
1  typedef int T;
2  typedef struct node {
3      T value;
4      struct node* prev;
5      struct node* next;
6  } Node, *TList;
```

Reprezentarea arborilor binari

- Amintim reprezentările utilizate pentru liste
  1. Lista simplu înlănțuită

```
typedef int T;
typedef struct node {
        T value;
        struct node* next;
} Node, *TList;
```

  2. Lista dublu înlănțuită

```
1  typedef int T;
2  typedef struct node {
3      T value;
4      struct node* prev;
5      struct node* next;
6  } Node, *TList;
```

Un arbore binar este o generalizare a unei liste! Anumiti arbori (degenerati) devin chiar liste.

Reprezentarea arborilor binari

```
1  typedef int T;
2  typedef struct node {
3      T value;
4      struct node* left;
5      struct node* right;
6  } Node, *TBinaryTree;
```

Reprezentarea arborilor binari

```
1  typedef int T;
2  typedef struct node {
3      T value;
4      struct node* left;
5      struct node* right;
6  } Node, *TBinaryTree;
```

Cum putea reprezenta următorul arbore?

Reprezentarea arborilor binari

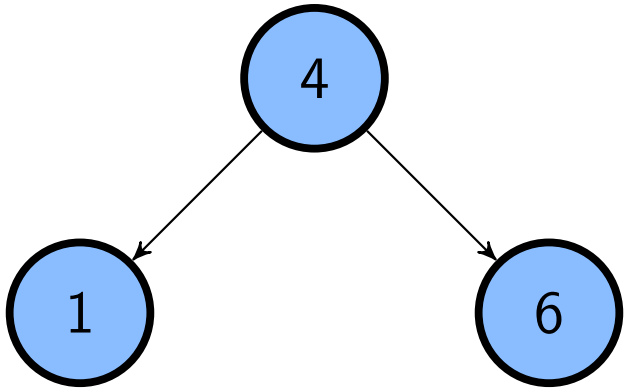
```
1  typedef int T;
2  typedef struct node {
3      T value;
4      struct node* left;
5      struct node* right;
6  } Node, *TBinaryTree;
```

Cum putea reprezenta următorul arbore?


```
TBinaryTree root = malloc(sizeof(Node));
root->value = 10;
root->left = NULL;
root->right = NULL;
```

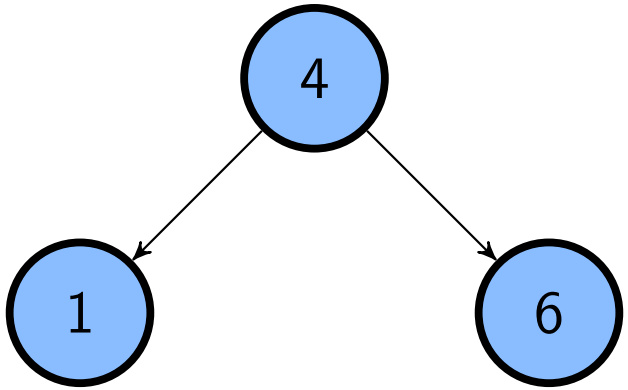
Reprezentarea arborilor binari

- Cum putem reprezenta următorul arbore?



Reprezentarea arborilor binari

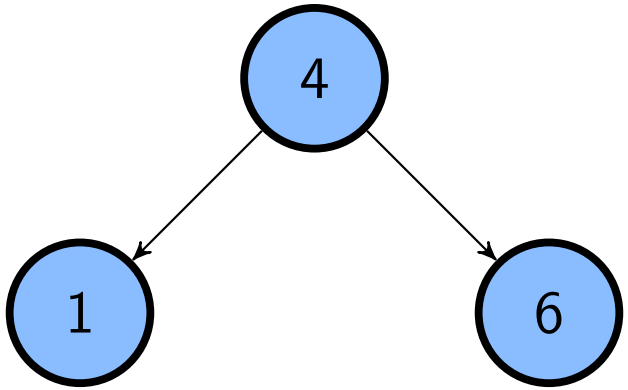
- Cum putem reprezenta următorul arbore?




```
1  TBinaryTree root = malloc(sizeof(Node));
2  root->value = 4;
```

Reprezentarea arborilor binari

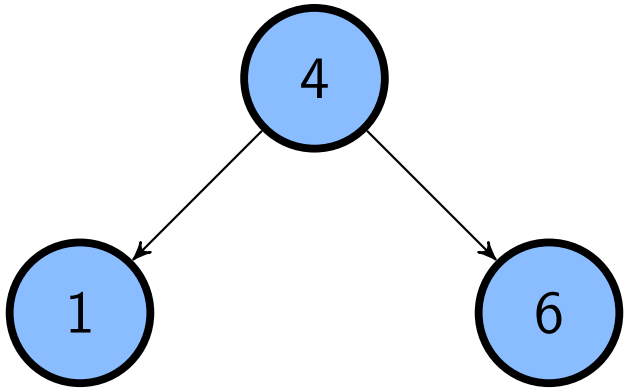
- Cum putem reprezenta următorul arbore?




```
1  TBinaryTree root = malloc(sizeof(Node));
2  root->value = 4;
3  TBinaryTree left, right;
4  left = calloc(1, sizeof(Node));
5  left->value = 1;
```

Reprezentarea arborilor binari

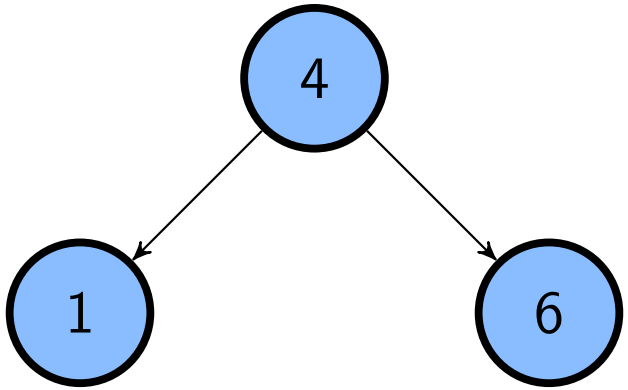
- Cum putem reprezenta următorul arbore?




```
1  TBinaryTree root = malloc(sizeof(Node));
2  root->value = 4;
3  TBinaryTree left, right;
4  left = calloc(1, sizeof(Node));
5  left->value = 1;
6  right = calloc(1, sizeof(Node));
7  right->value = 6;
```

Reprezentarea arborilor binari

- Cum putem reprezenta următorul arbore?



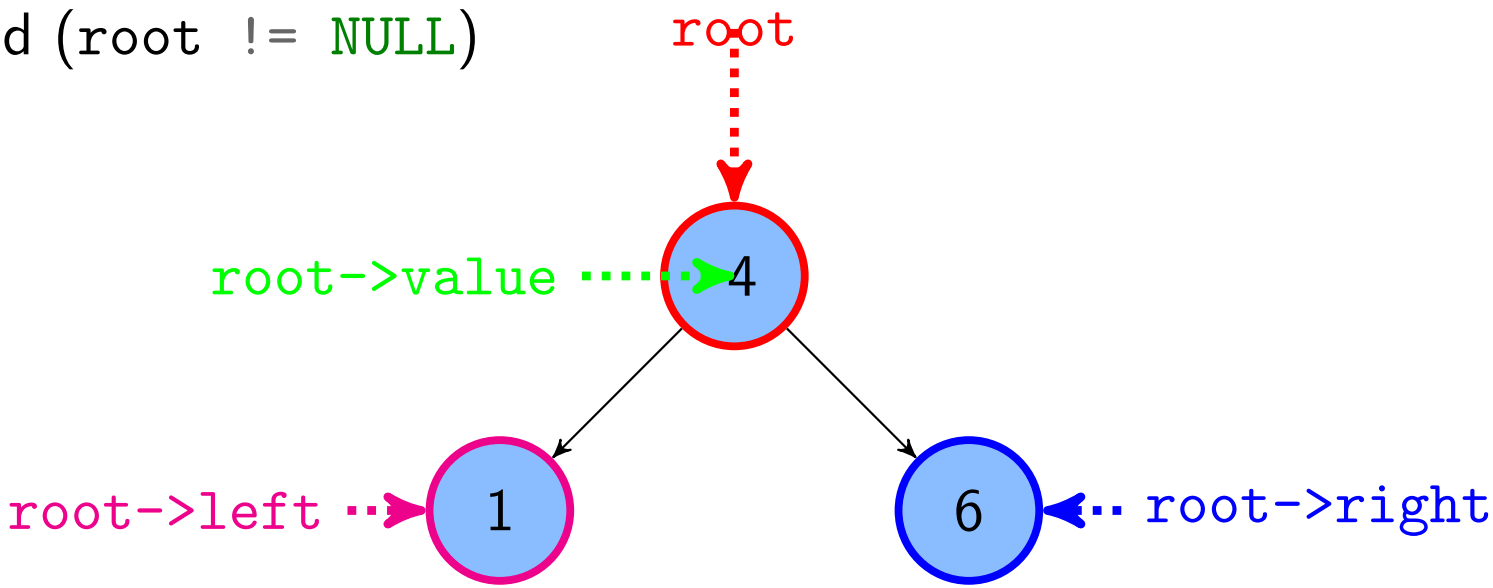

```
1  TBinaryTree root = malloc(sizeof(Node));
2  root->value = 4;
3  TBinaryTree left, right;
4  left = calloc(1, sizeof(Node));
5  left->value = 1;
6  right = calloc(1, sizeof(Node));
7  right->value = 6;
8  root->left = left;
9  root->right = right;
```

Arbori binari – Accesări

1. Arborele vid
   - Inițializare: `TBinaryTree root = NULL;`
   - Testare arbore vid: `root == NULL`

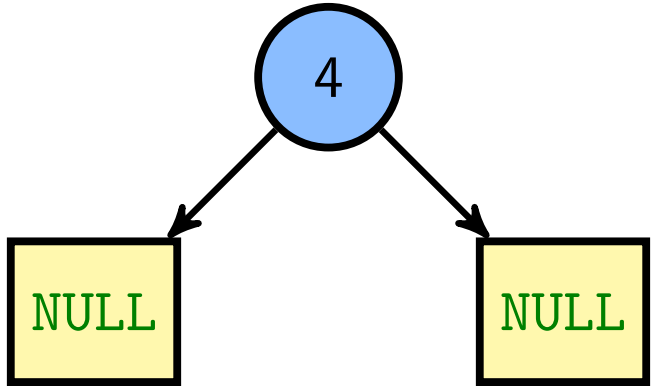
Arbori binari – Accesări

1. Arborele vid
   - Inițializare: `TBinaryTree root = NULL;`
   - Testare arbore vid: `root == NULL`
2. Arborele nevid (`root != NULL`)



- Condiția pentru frunză:
  `node->left == NULL && node->right = NULL`
- Valoarea din rădăcină `root->value`
- Copilul stâng al rădăcinii: `root->left`
- Valoarea copilului stâng al rădăcinii: `root->left->value`

Arbori binari – Inițializare



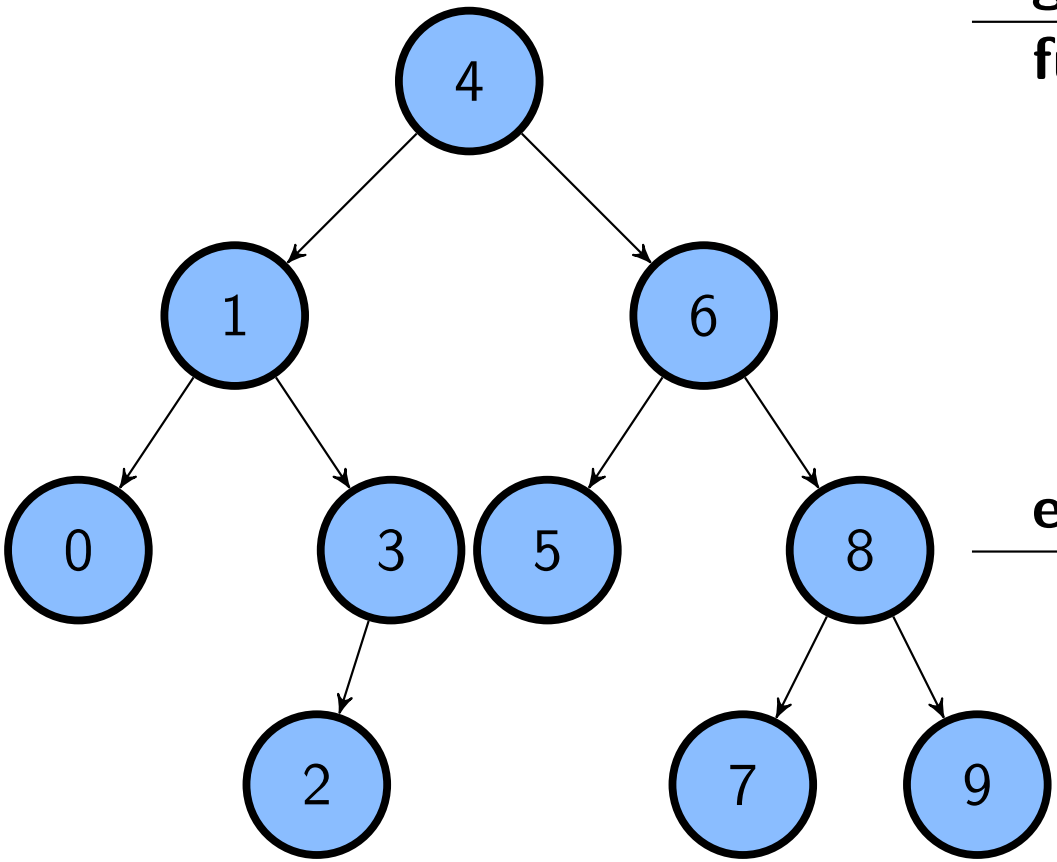

```
1  TBinaryTree initBinaryTree() {
2      return NULL;
3  }
4
5  TBinaryTree createBinaryTree(T value) {
6      TBinaryTree root = malloc(sizeof(Node));
7      root->value = value;
8      root->left = NULL;
9      root->right = NULL;
10     return root;
11 }
```

Parcurgerea arborilor binari

Parcurgerea unui arbore implică **vizitarea tuturor nodurilor dintr-un arbore**. Deoarece arborele este o structură de date neliniară, nu există o unică parcurgere.

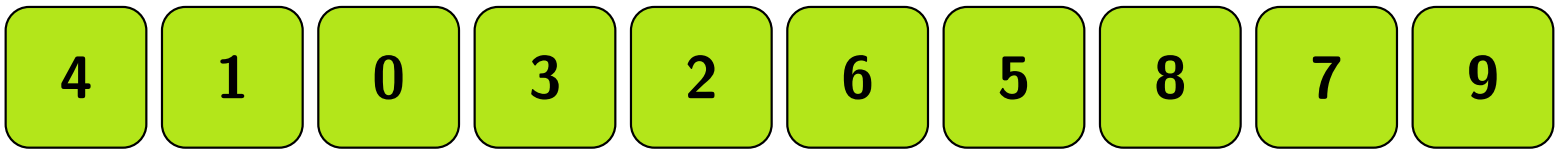
- Avem două tipuri de parcurgere:
  1. Parcurgere în adâncime
  2. Parcurgere pe nivel
- Pentru parcurgerea în adâncime, avem mai multe variante pentru arborii binari
  1. Parcurgere în preordine
  2. Parcurgere în inordine
  3. Parcurgere în postordine

Parcurgerea în preordine

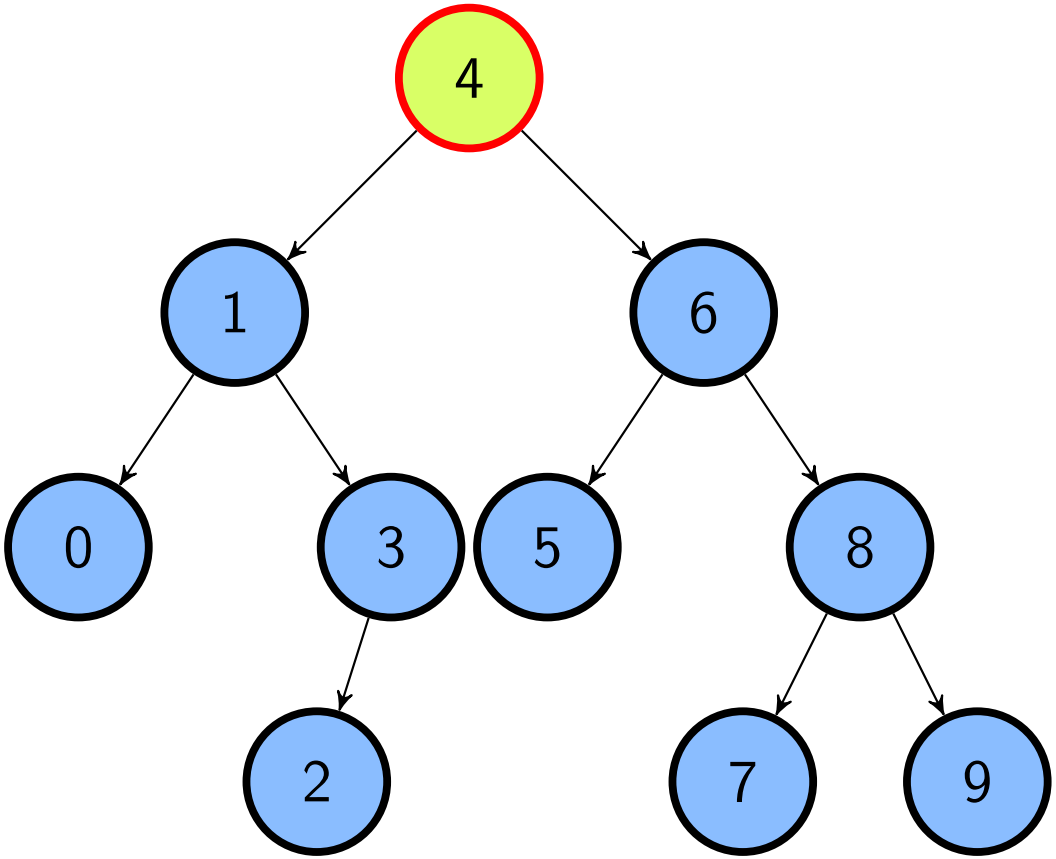


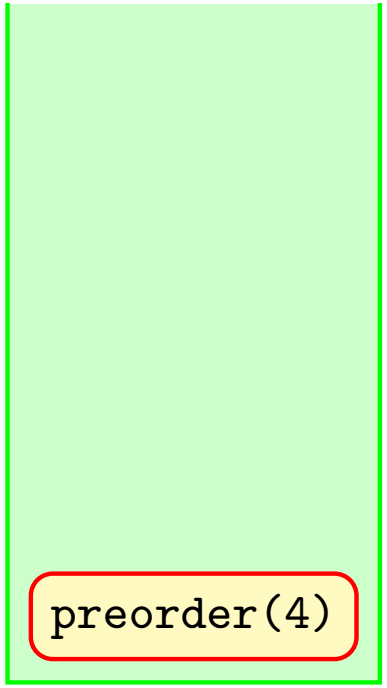
Algorithm 1 Preordine

```
function PREORDER(root)
    if root ≠ NULL then
        print(root → value)
        PREORDER(root → left)
        PREORDER(root → right)
    end if
end function
```



Parcurgerea în preordine

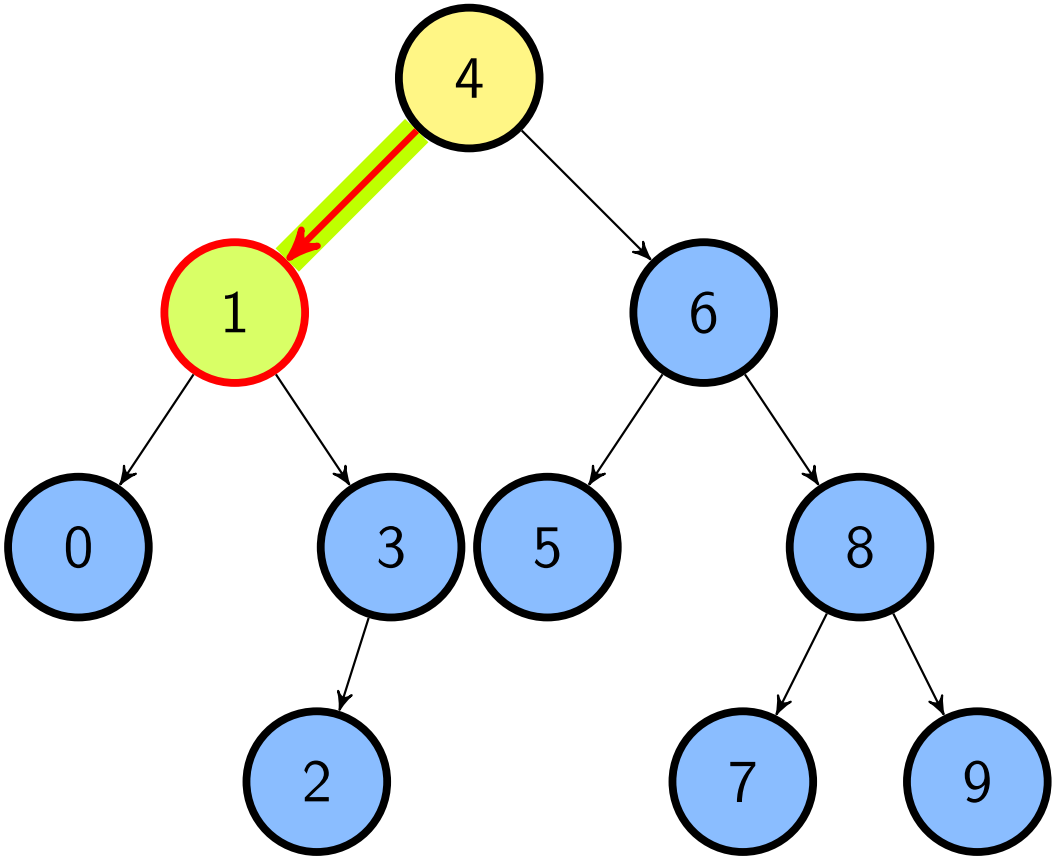


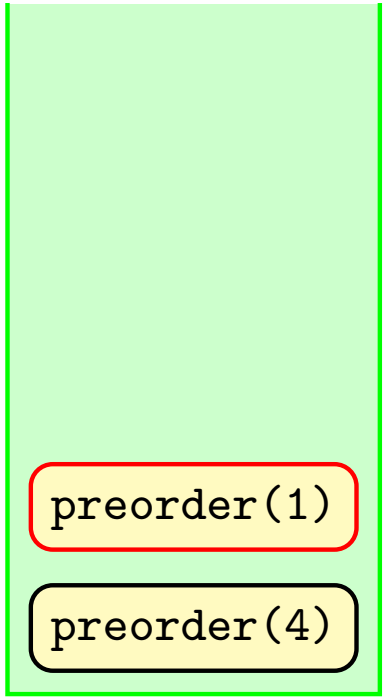


Stiva de apeluri recursive

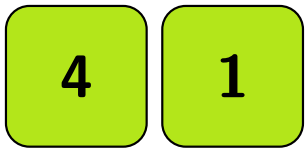
4

Parcurgerea în preordine

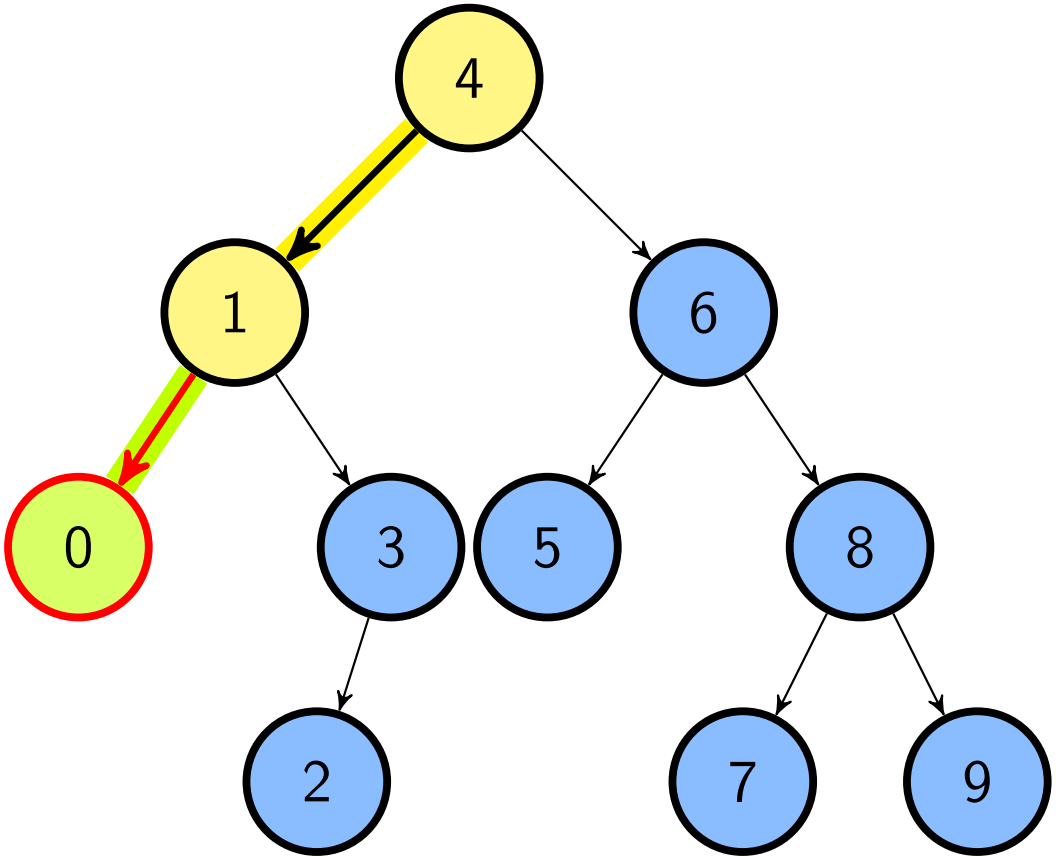


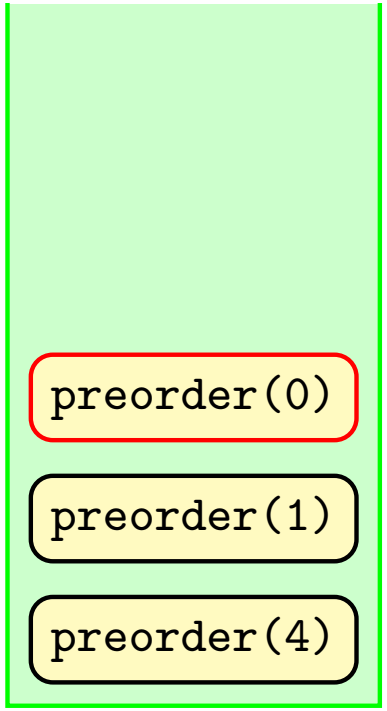


Stiva de apeluri recursive



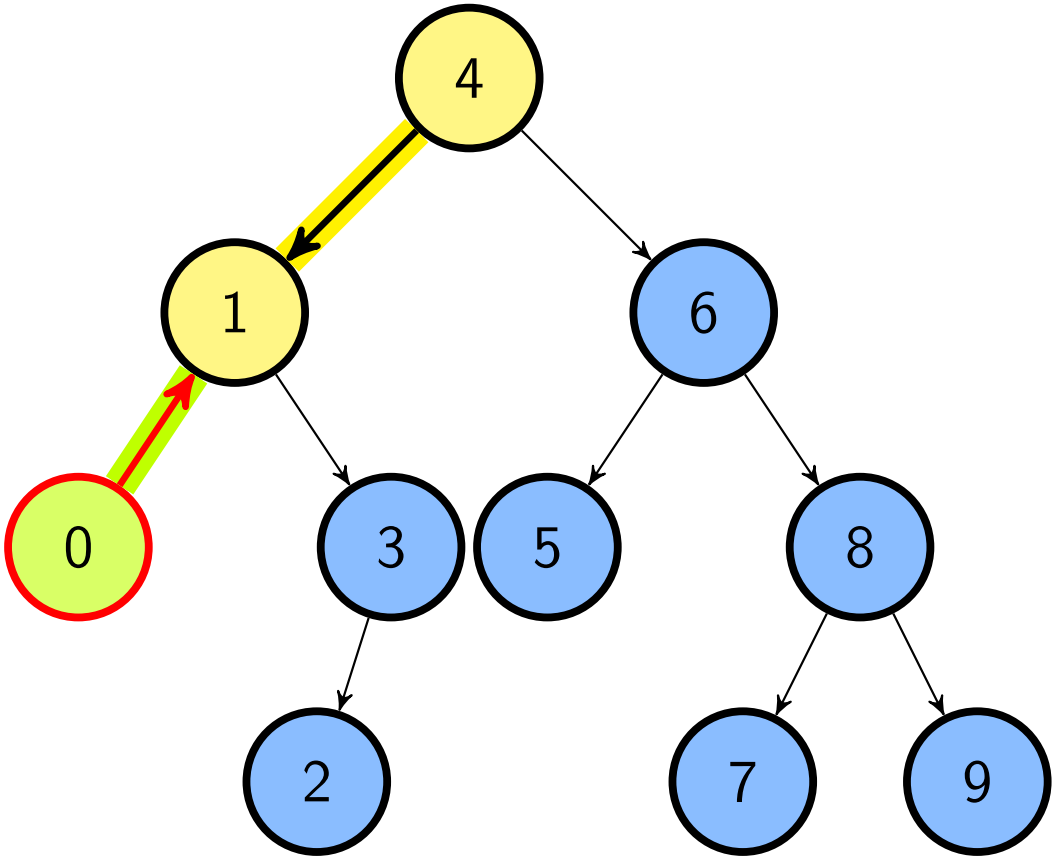
Parcurgerea în preordine

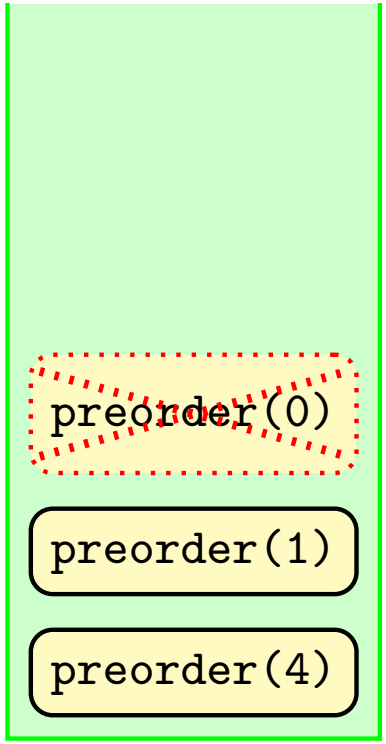




Stiva de apeluri recursive

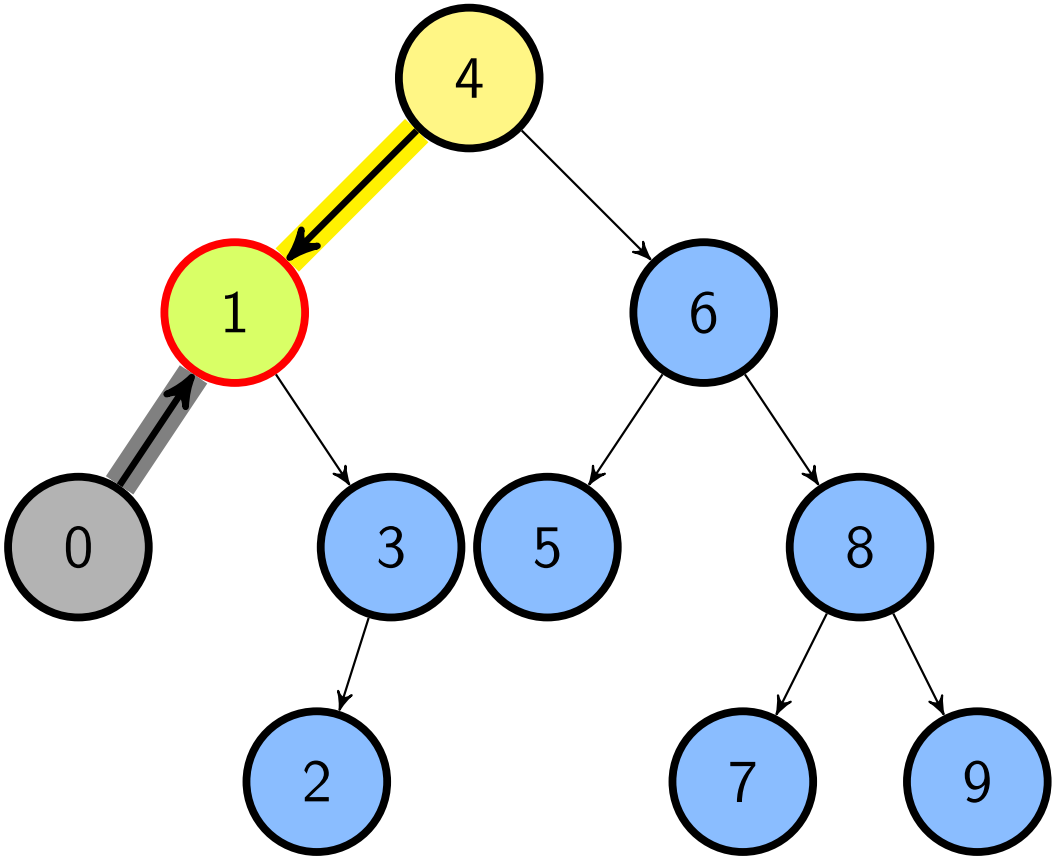
Parcurgerea în preordine

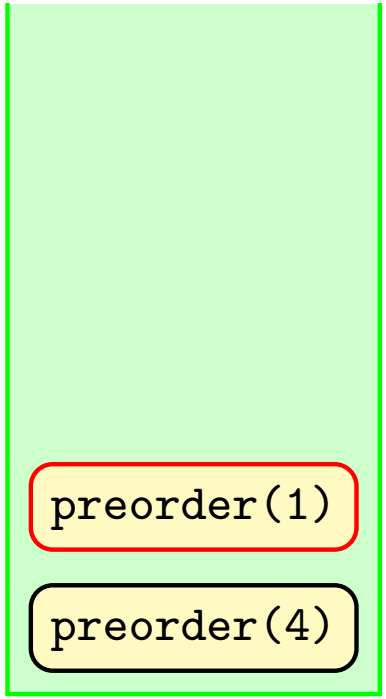




Stiva de apeluri recursive

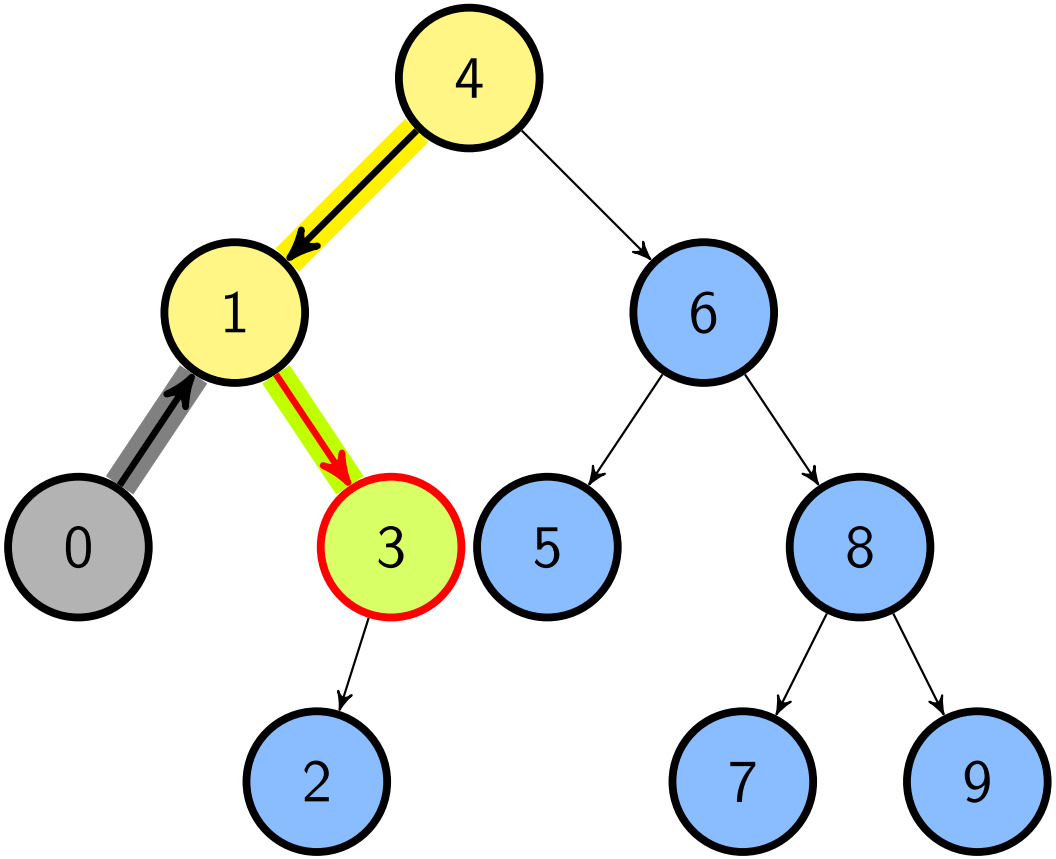
Parcurgerea în preordine

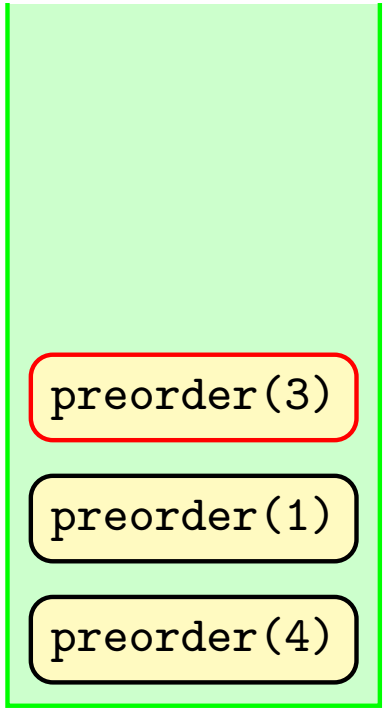




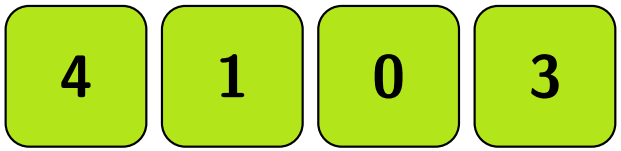
Stiva de apeluri recursive

Parcurgerea în preordine

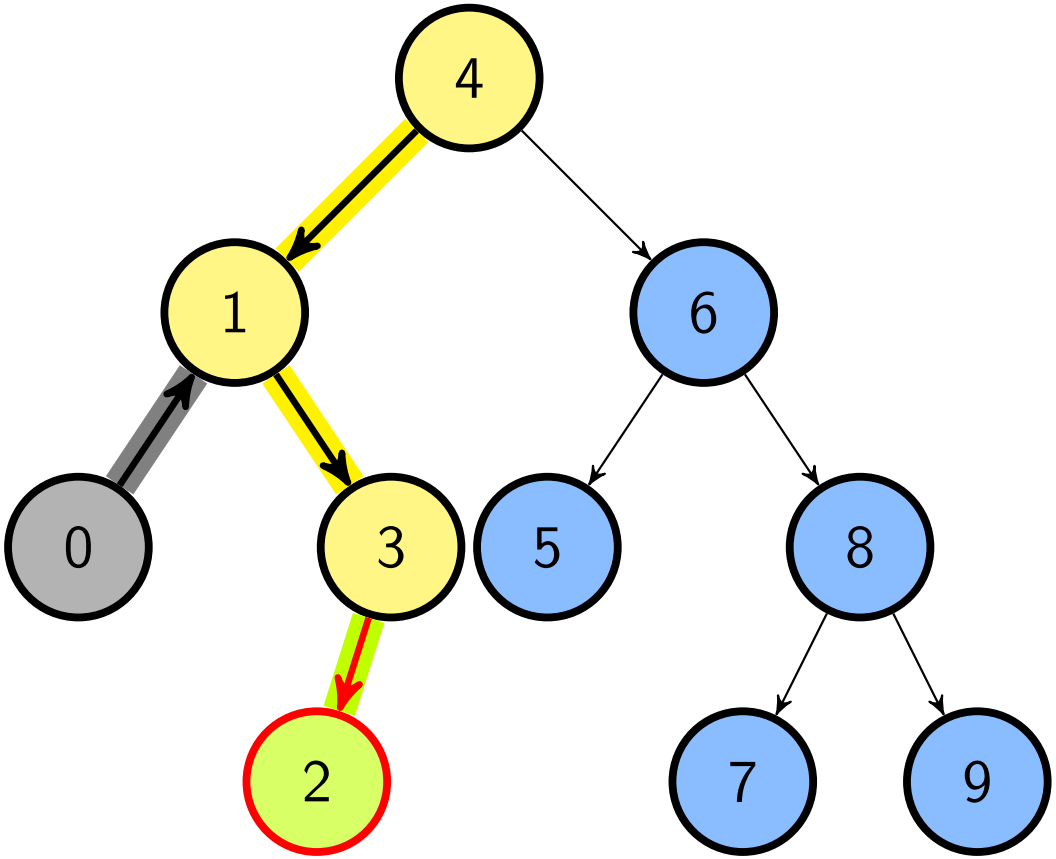


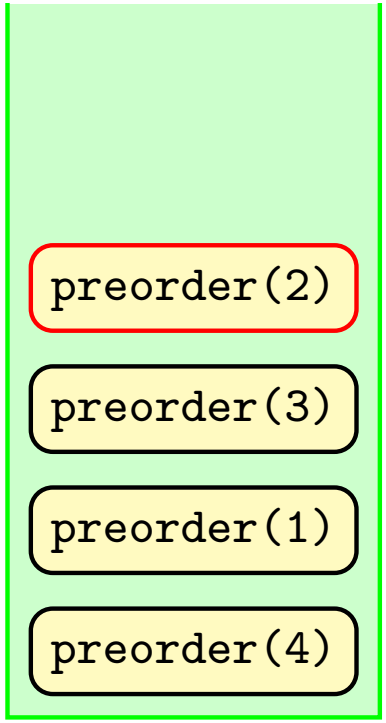


Stiva de apeluri recursive



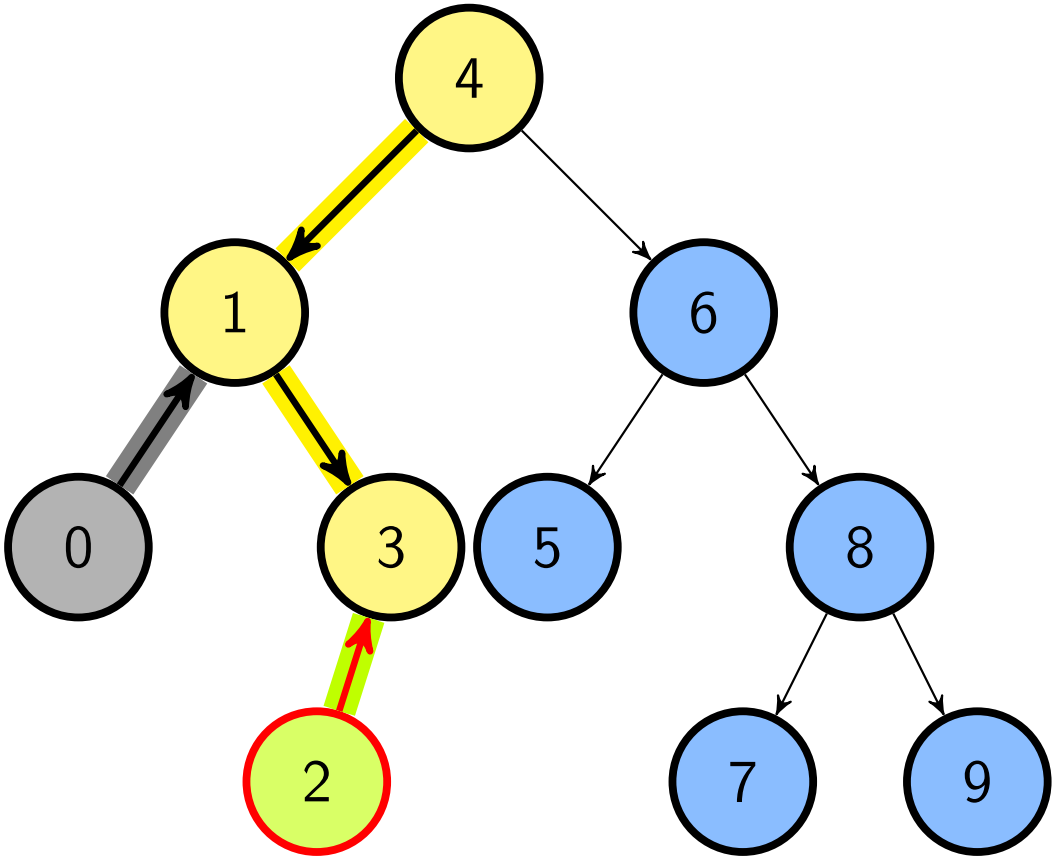
Parcurgerea în preordine

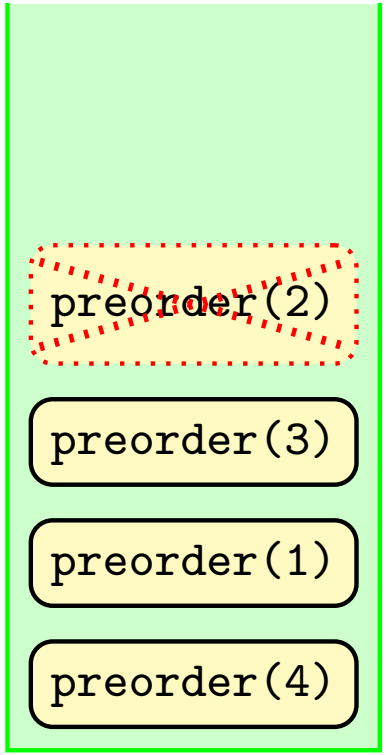




Stiva de apeluri recursive

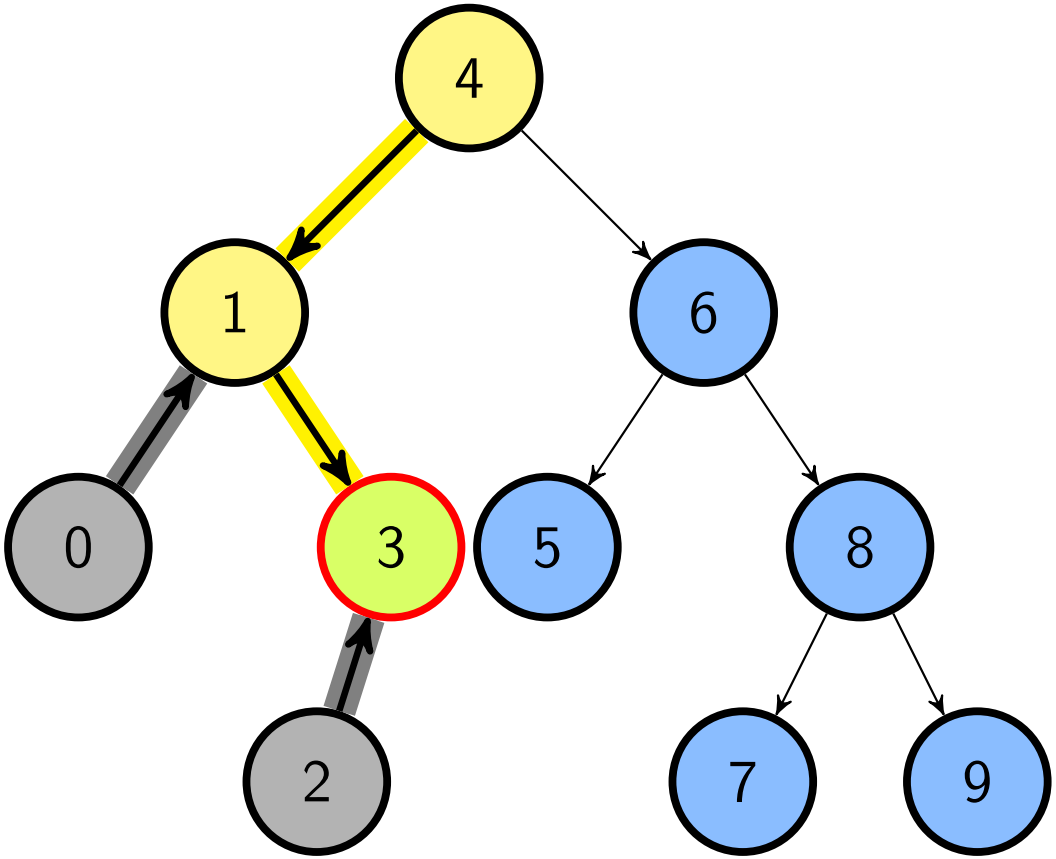
Parcurgerea în preordine

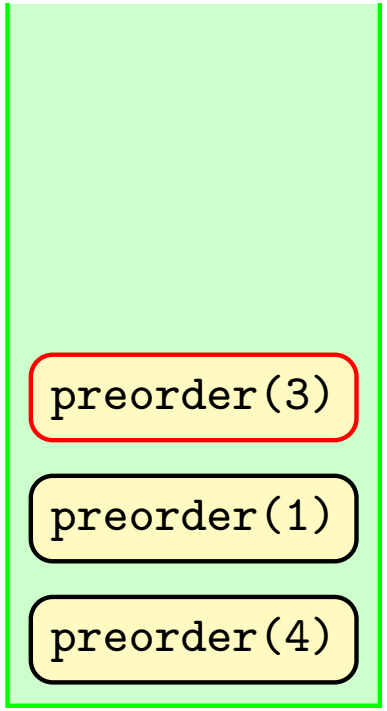




Stiva de apeluri recursive

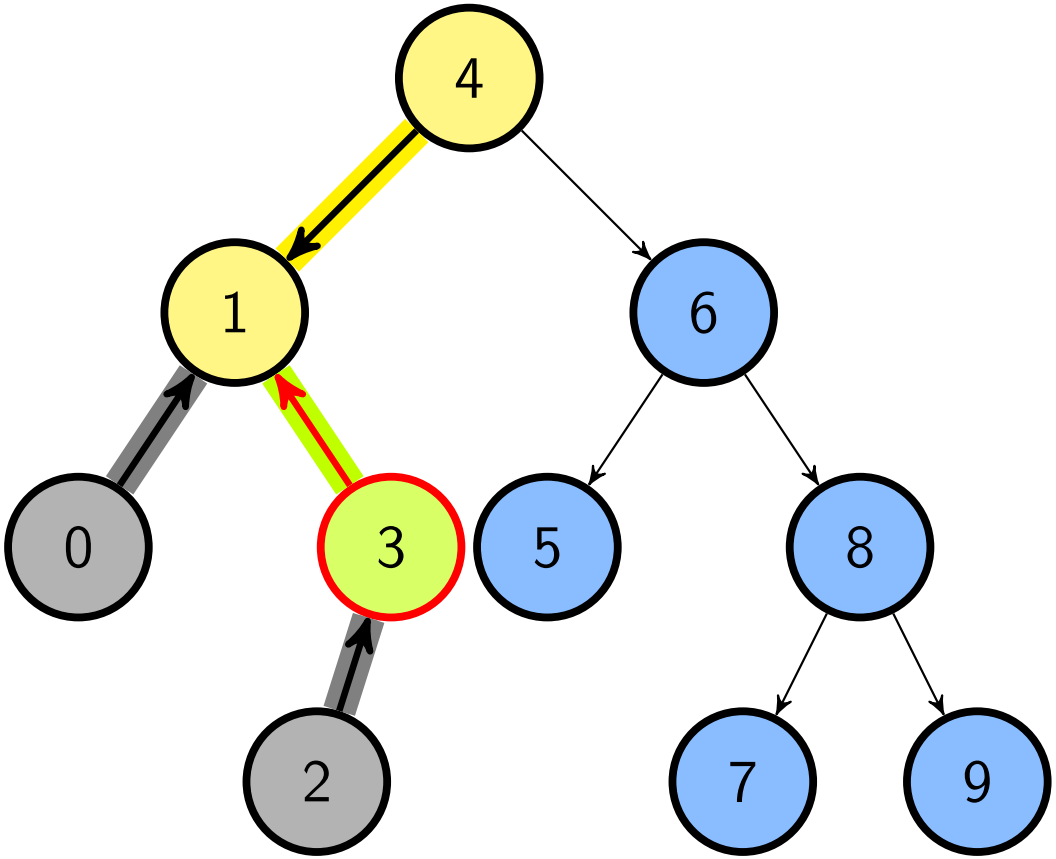
Parcurgerea în preordine

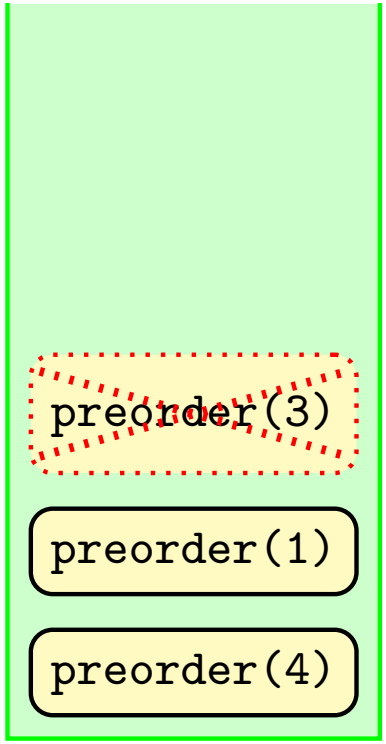




Stiva de apeluri recursive

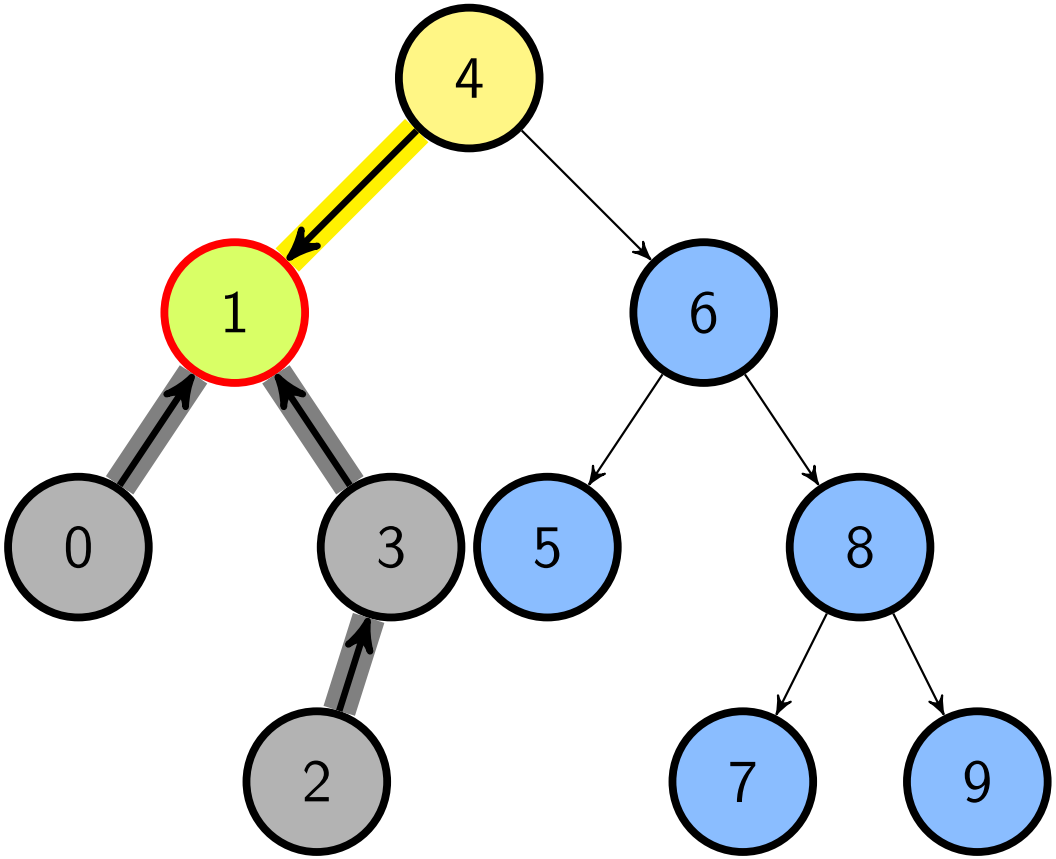
Parcurgerea în preordine

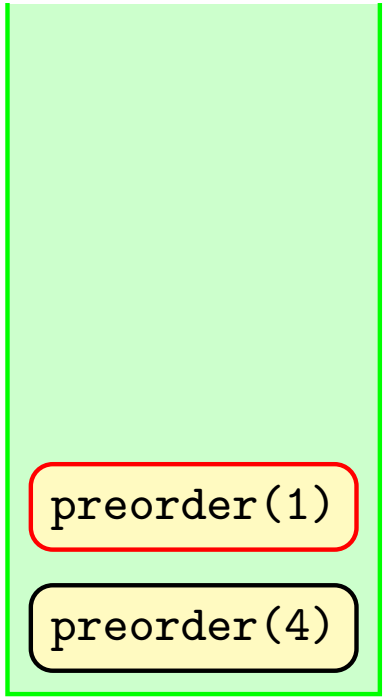




Stiva de apeluri recursive

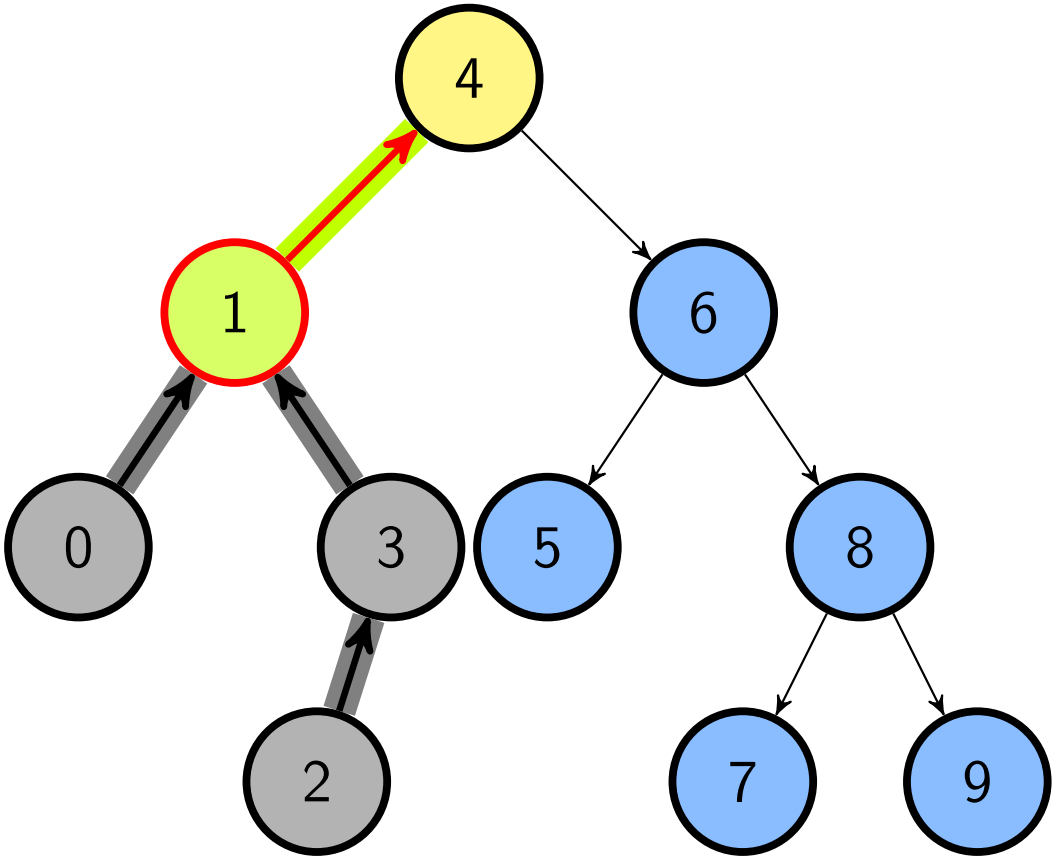
Parcurgerea în preordine

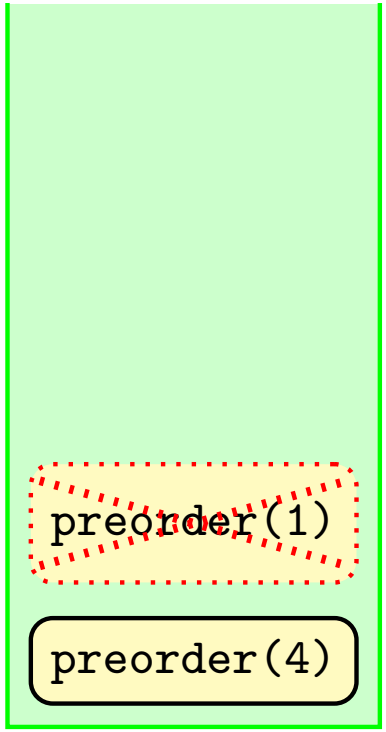




Stiva de apeluri recursive

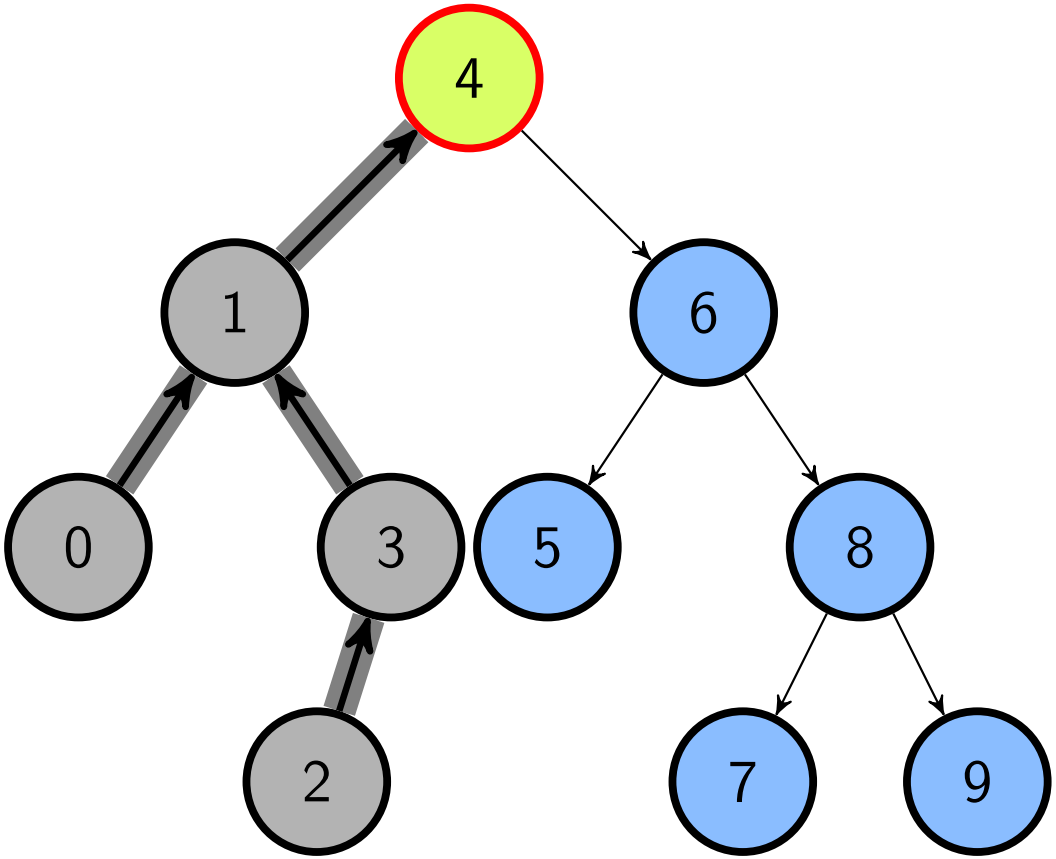
Parcurgerea în preordine

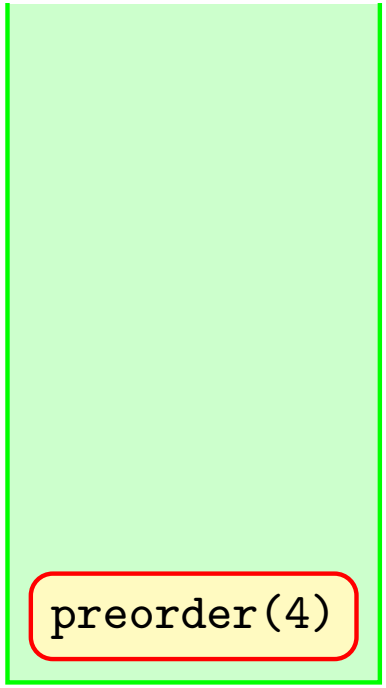




Stiva de apeluri recursive

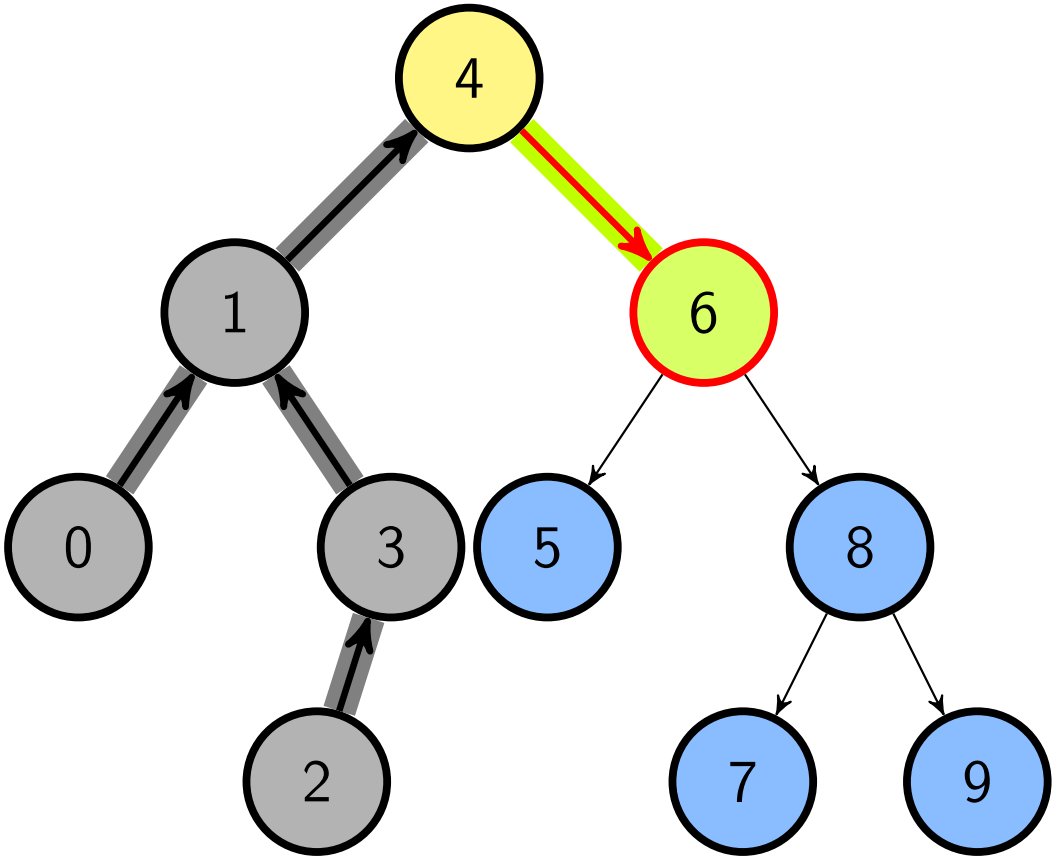
Parcurgerea în preordine

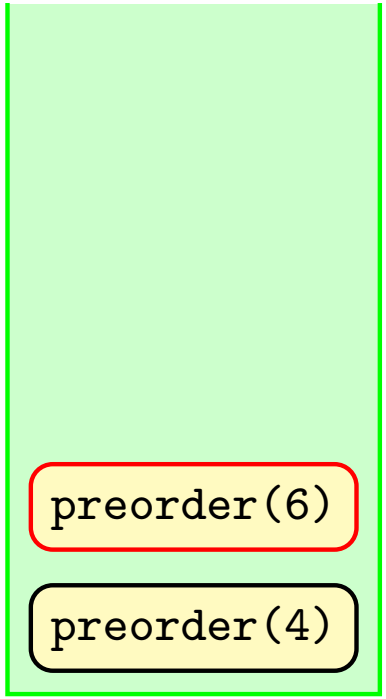




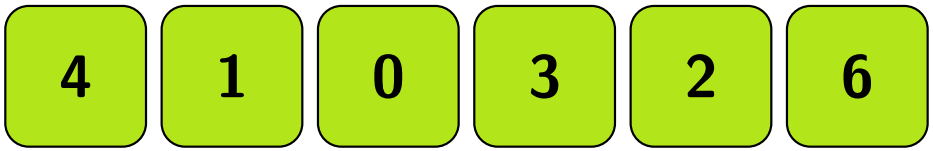
Stiva de apeluri recursive

Parcurgerea în preordine

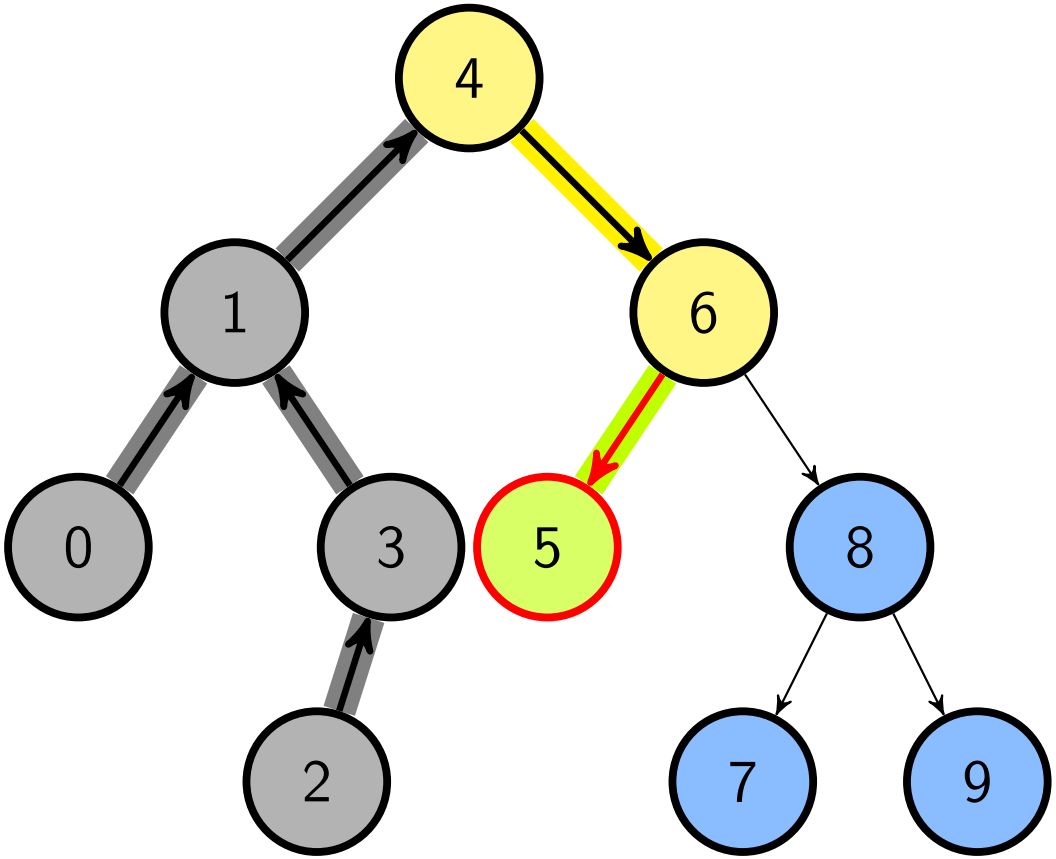


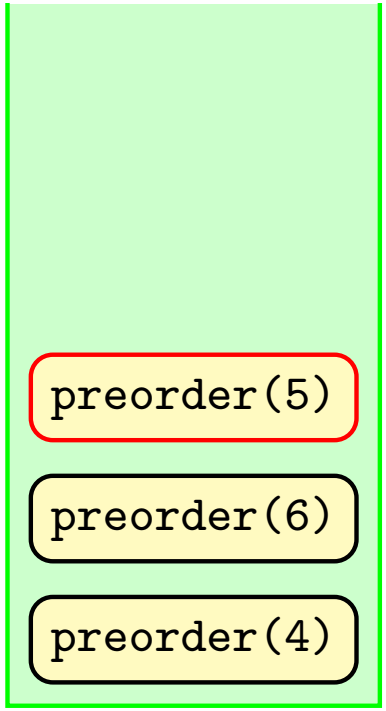


Stiva de apeluri recursive



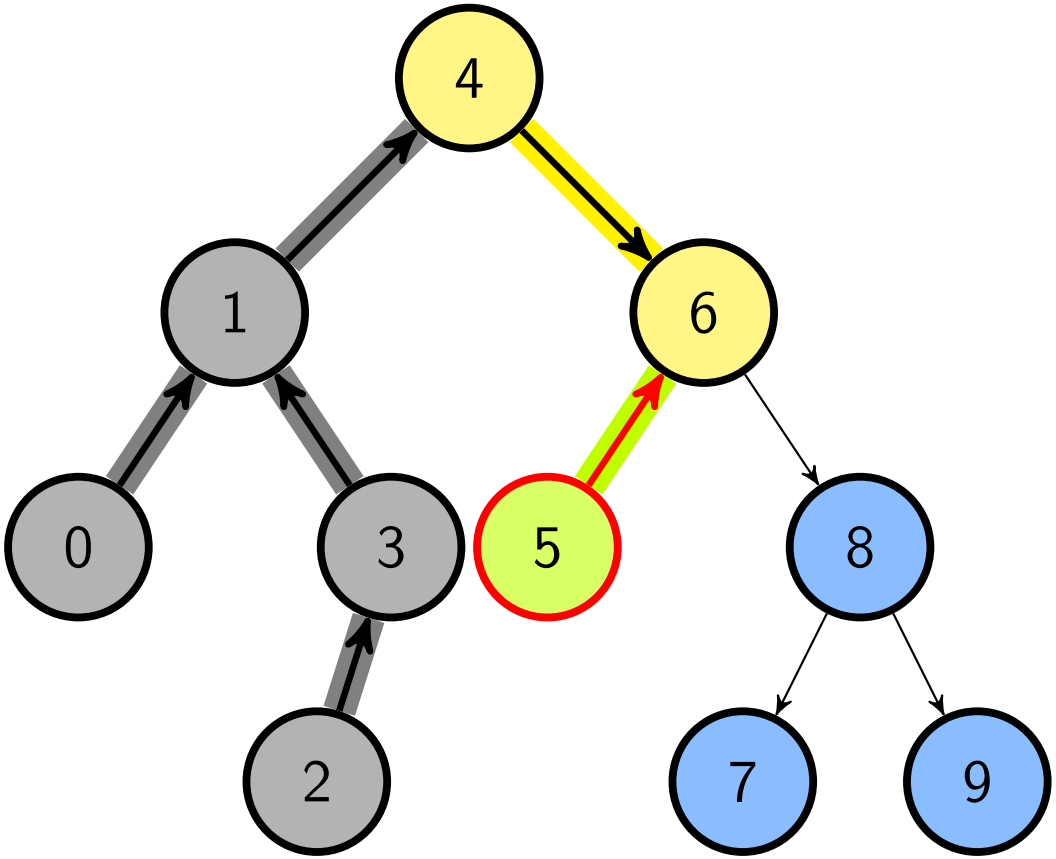
Parcurgerea în preordine

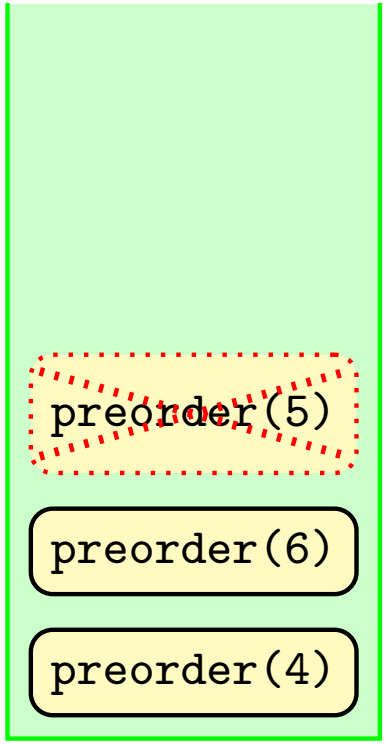




Stiva de apeluri recursive

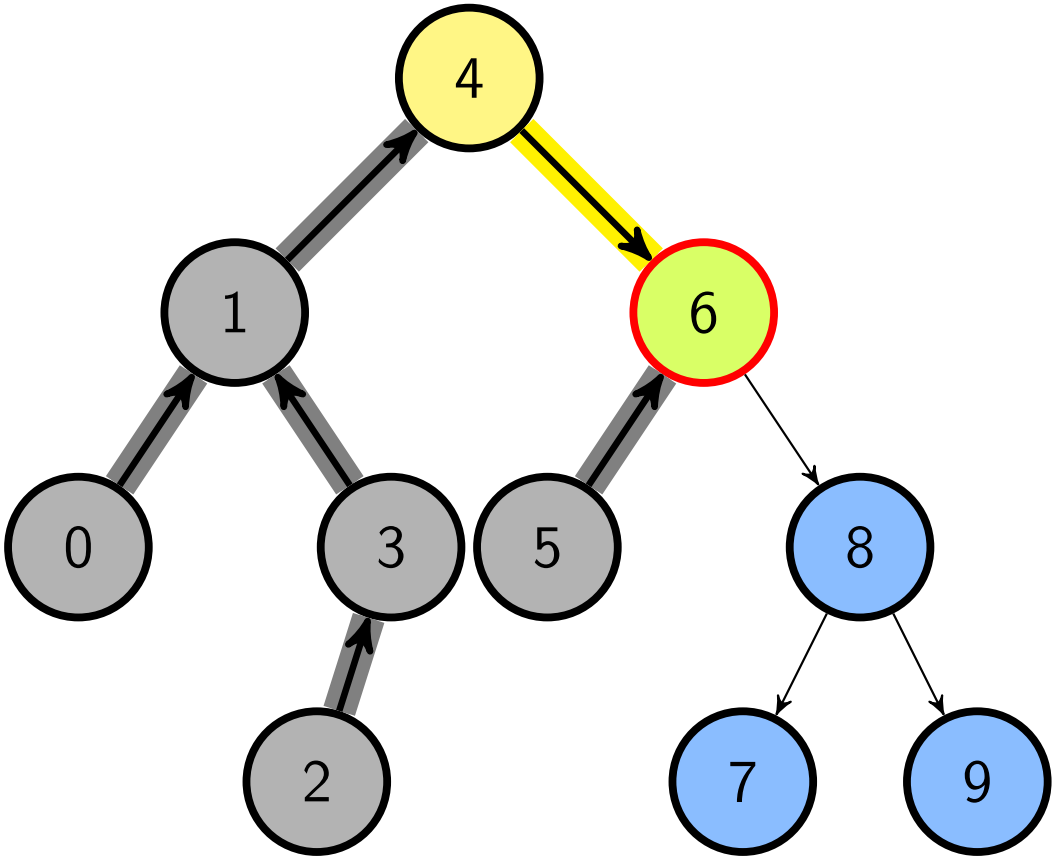
Parcurgerea în preordine

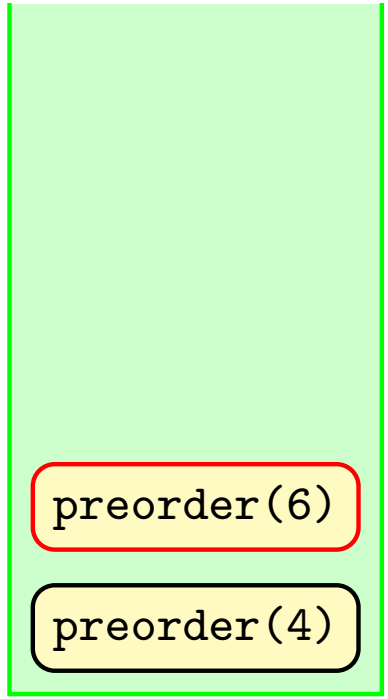




Stiva de apeluri recursive

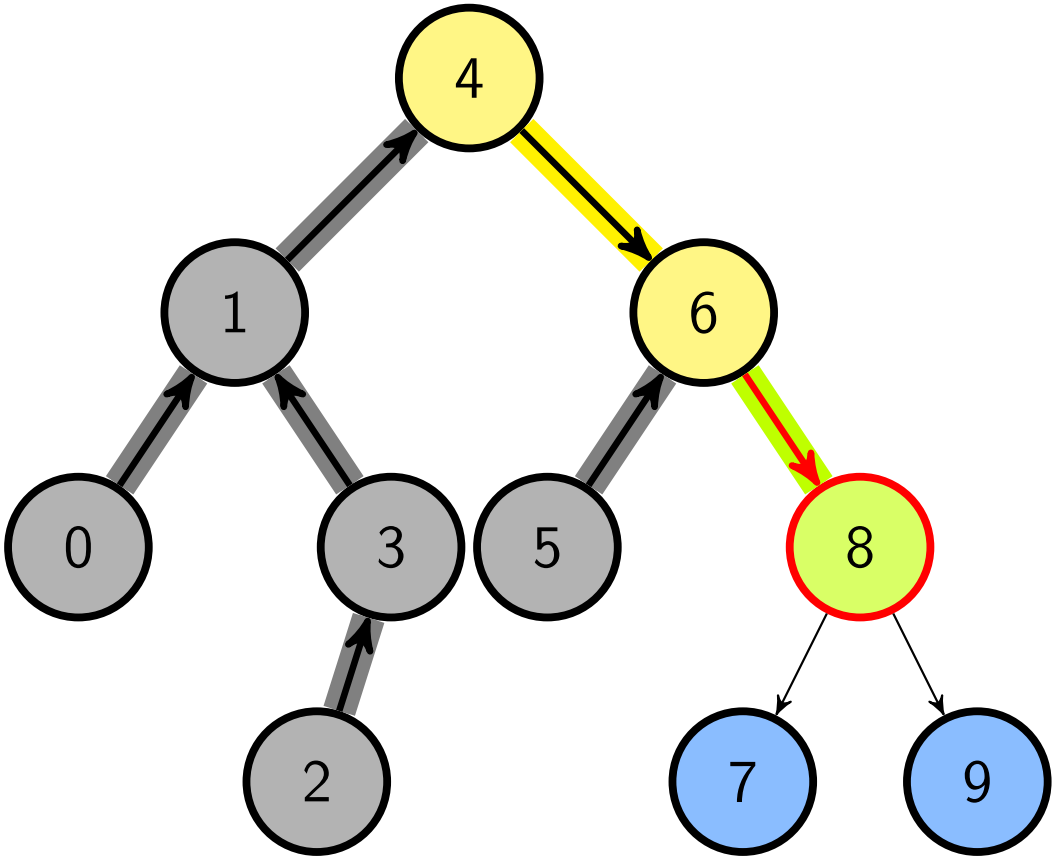
Parcurgerea în preordine

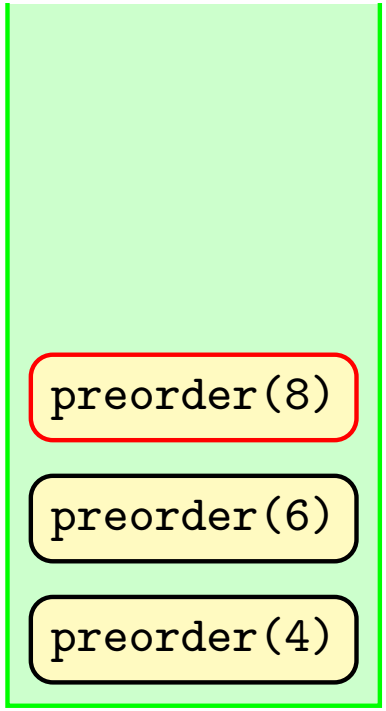




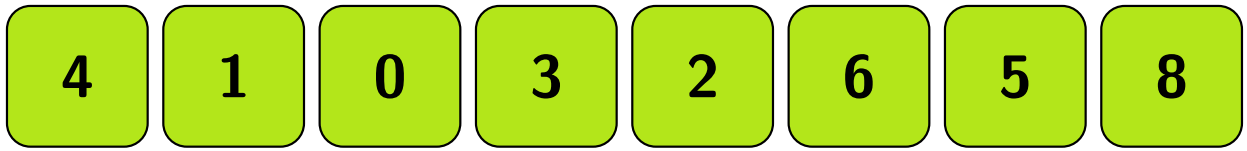
Stiva de apeluri recursive

Parcurgerea în preordine

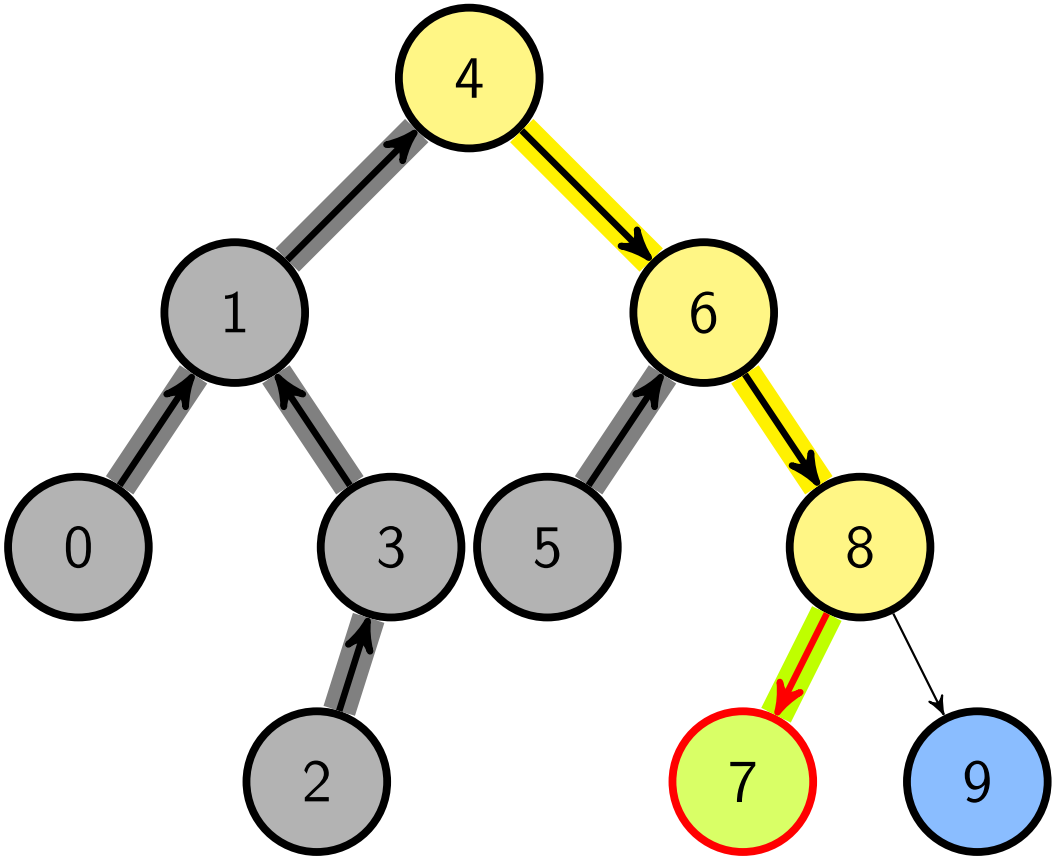


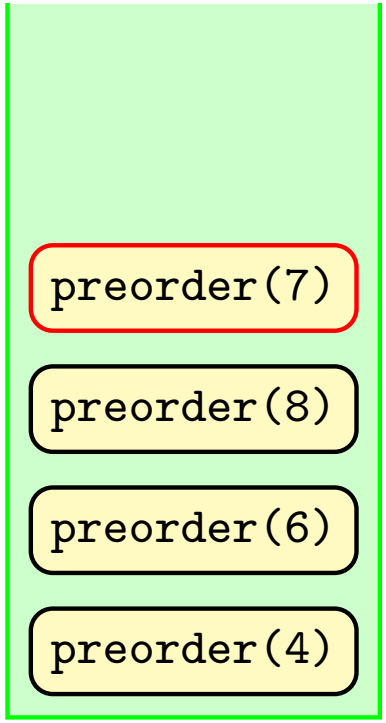


Stiva de apeluri recursive



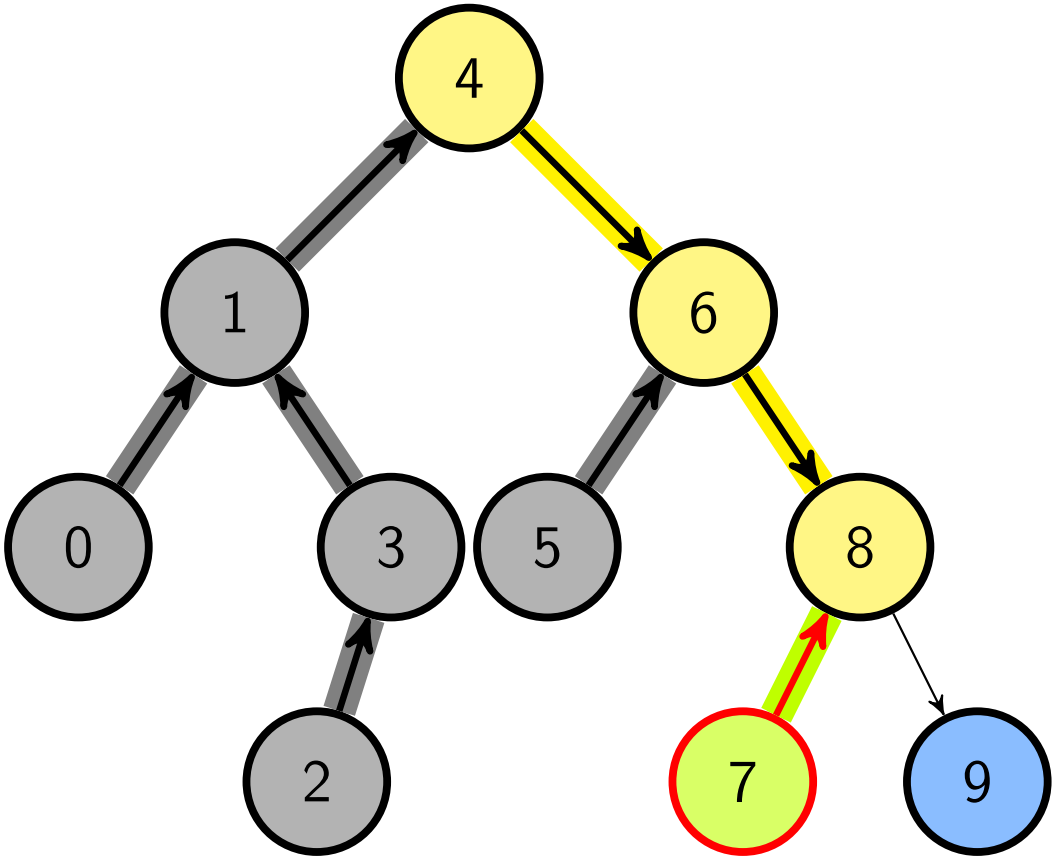
Parcurgerea în preordine

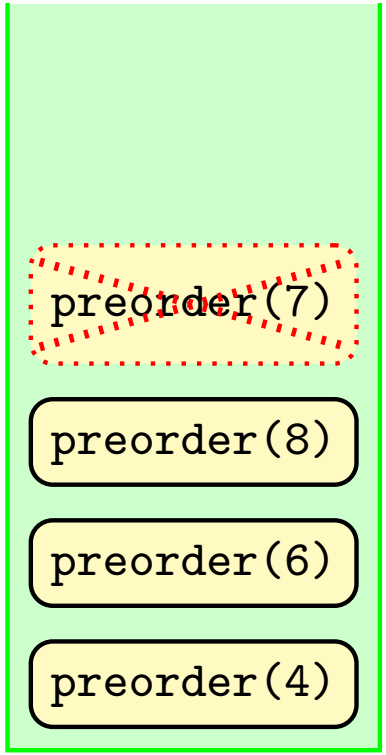




Stiva de apeluri recursive

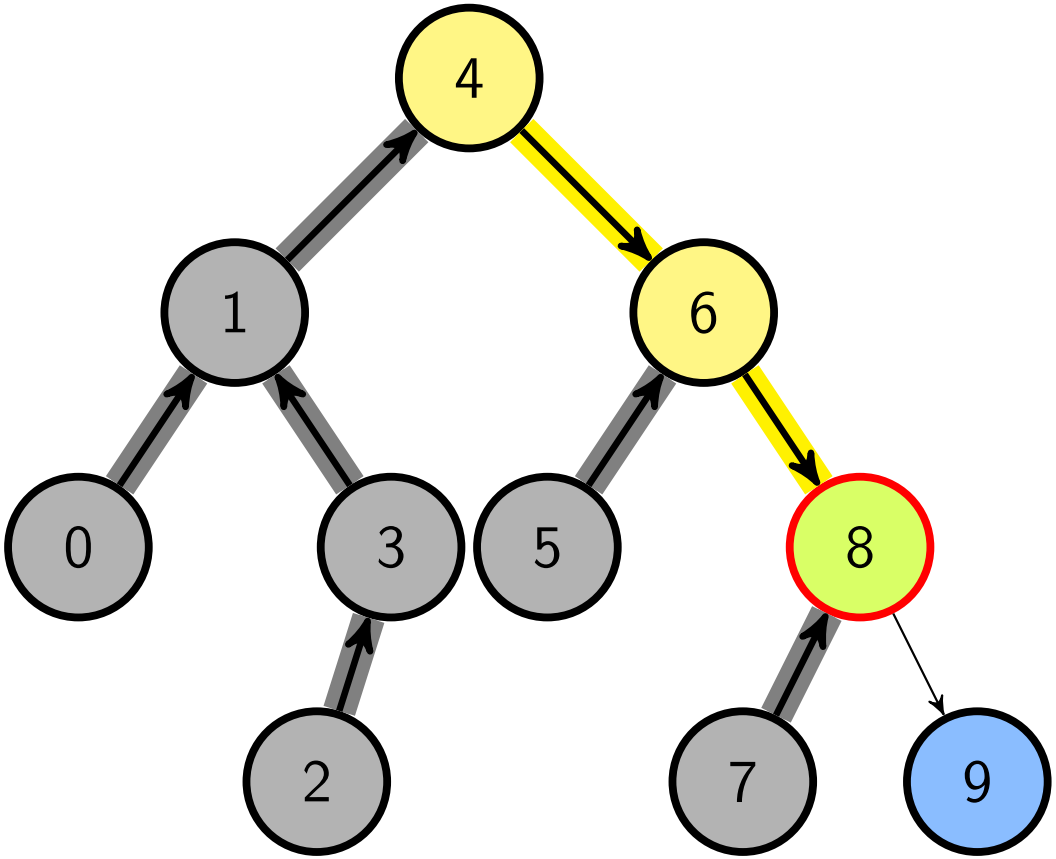
Parcurgerea în preordine

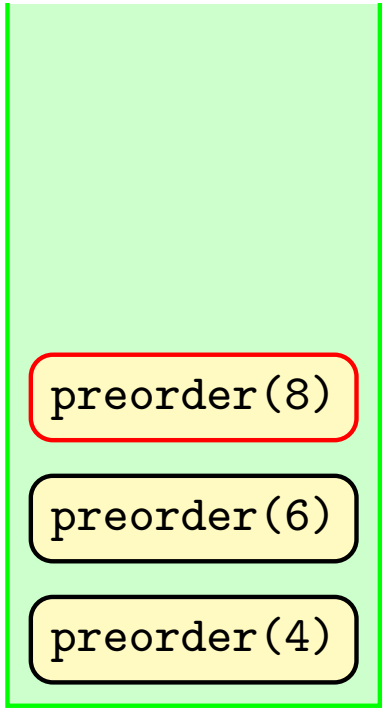




Stiva de apeluri recursive

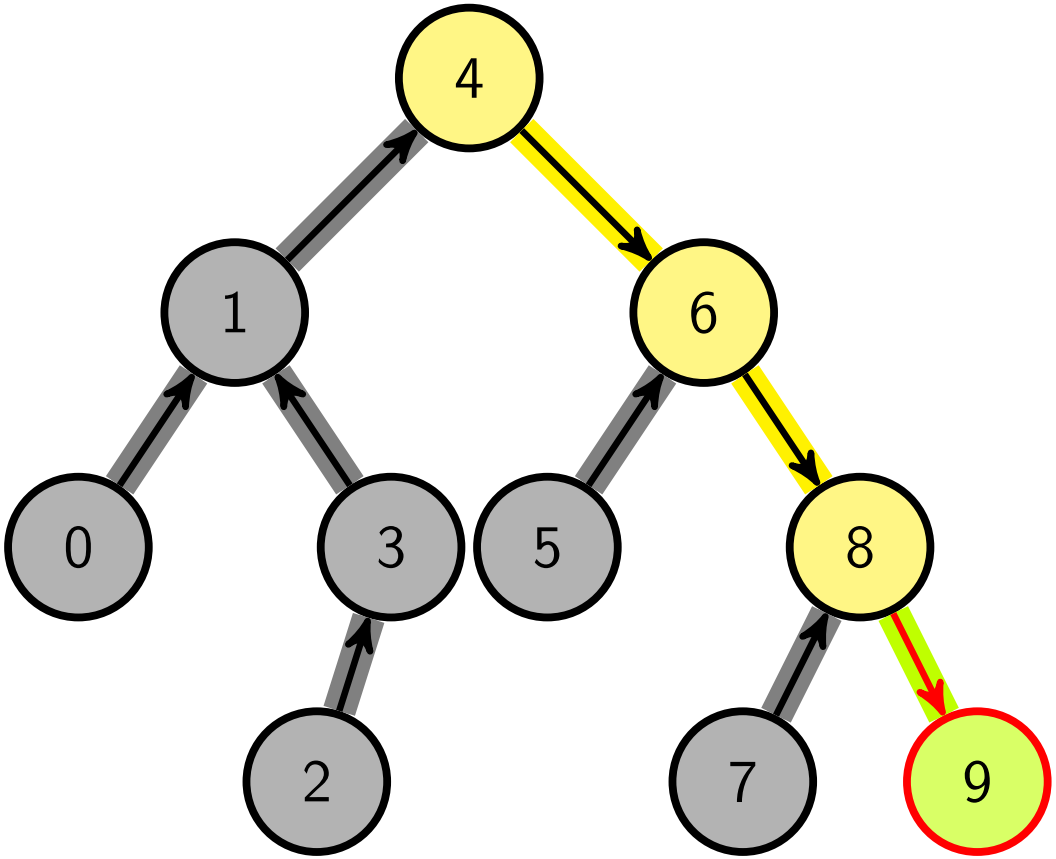
Parcurgerea în preordine

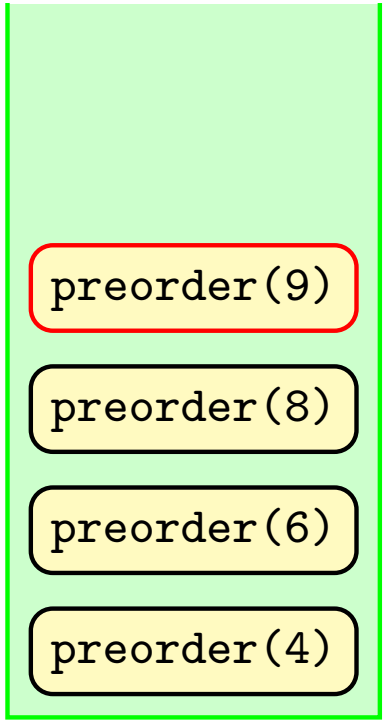




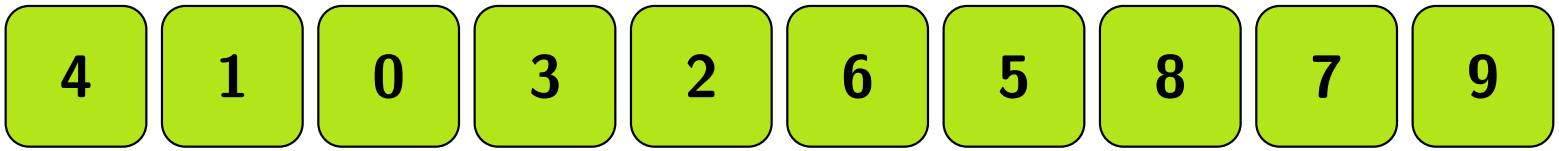
Stiva de apeluri recursive

Parcurgerea în preordine

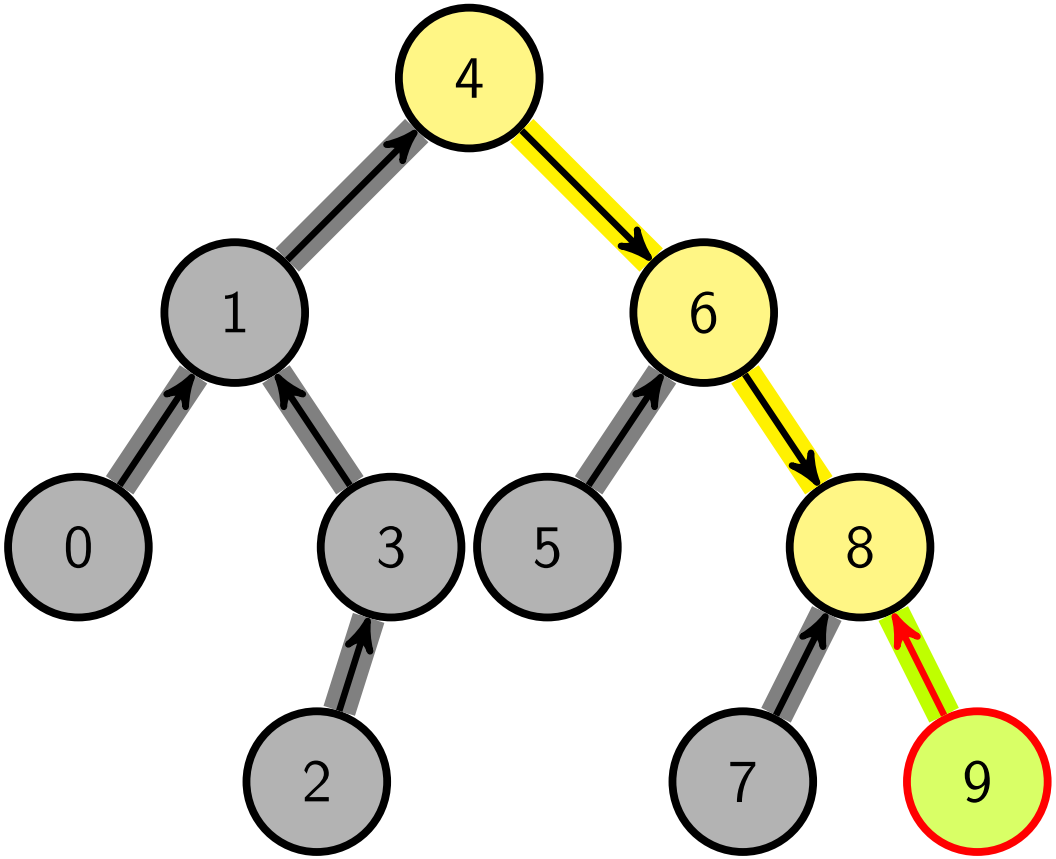


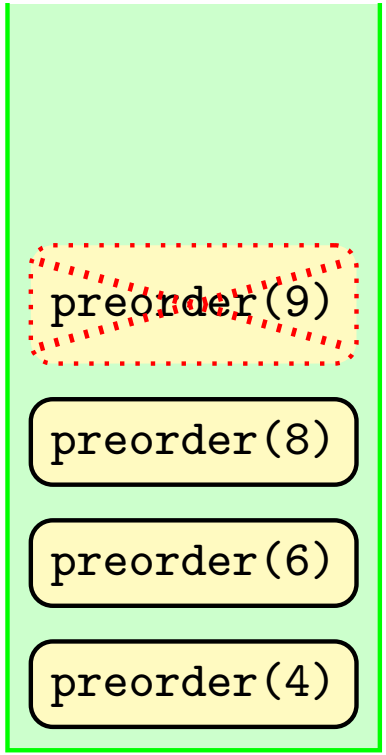


Stiva de apeluri recursive

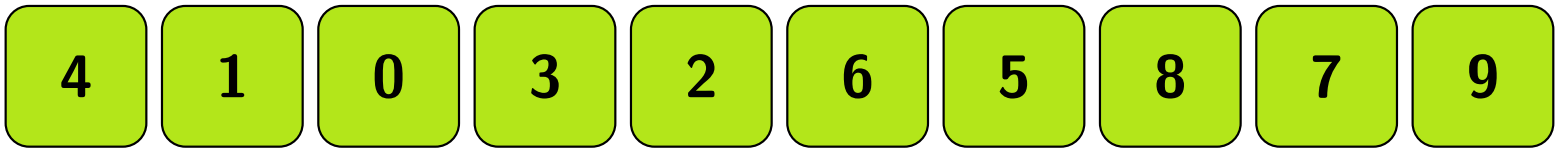


Parcurgerea în preordine

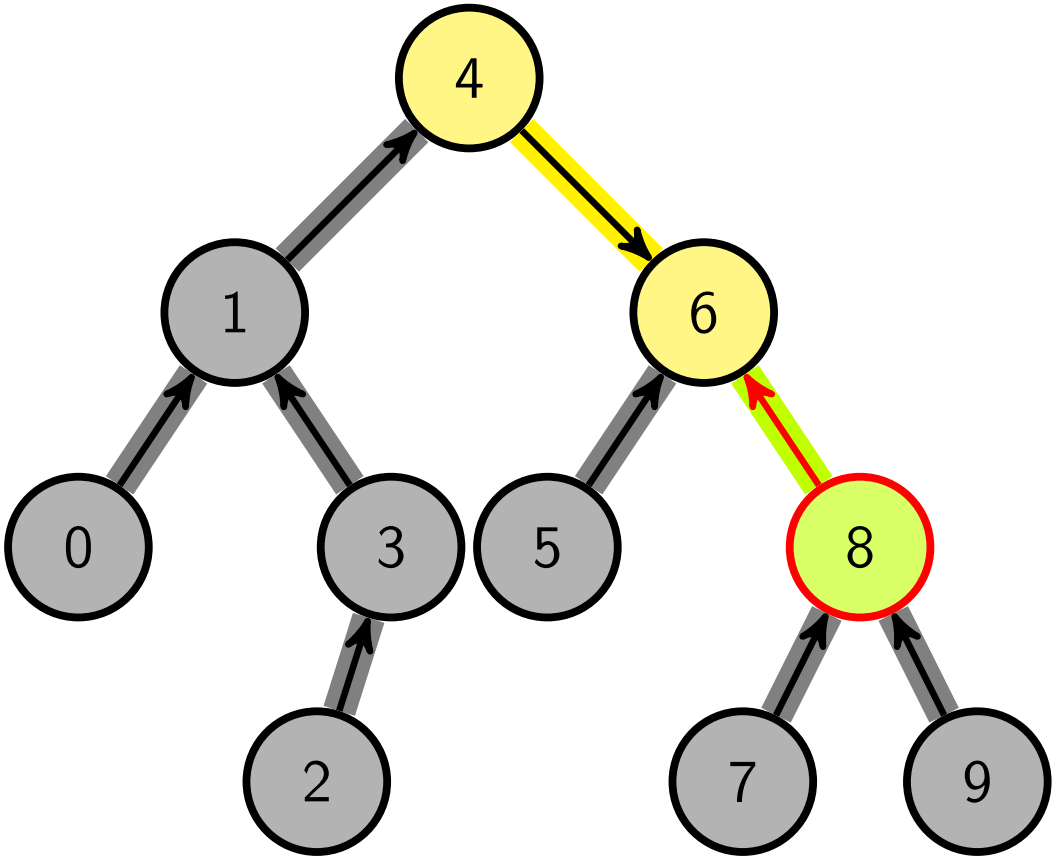


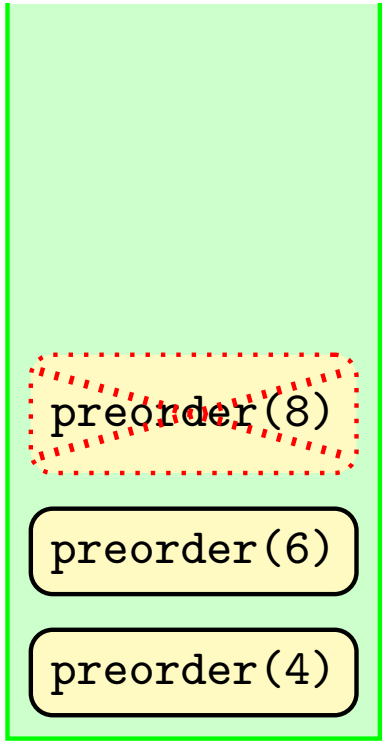


Stiva de apeluri recursive

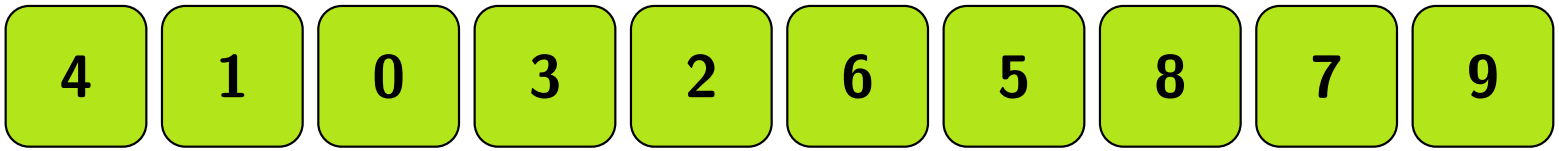


Parcurgerea în preordine

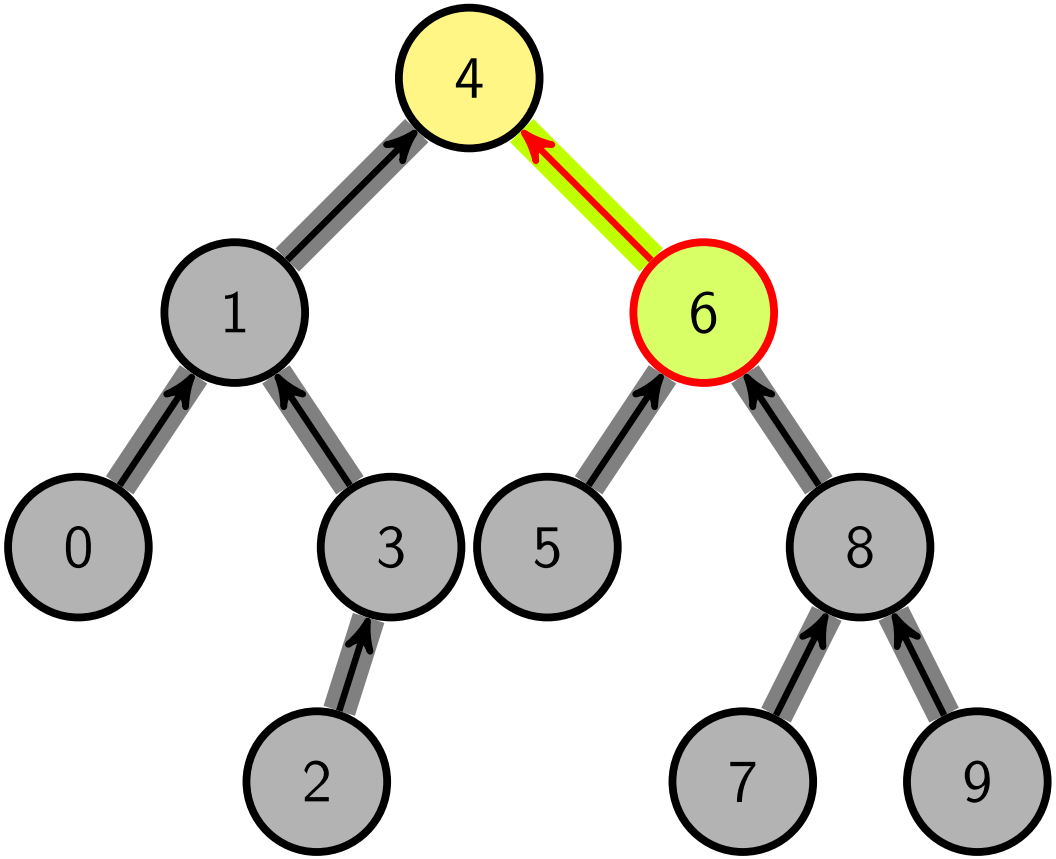


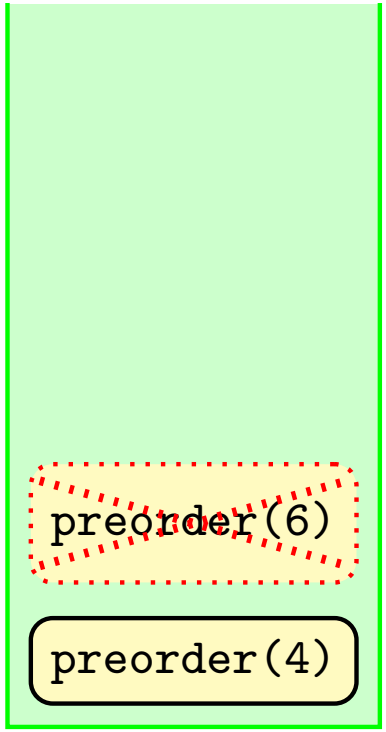


Stiva de apeluri recursive

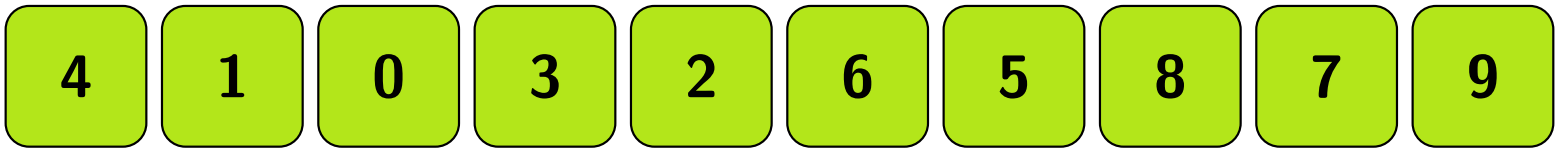


Parcurgerea în preordine

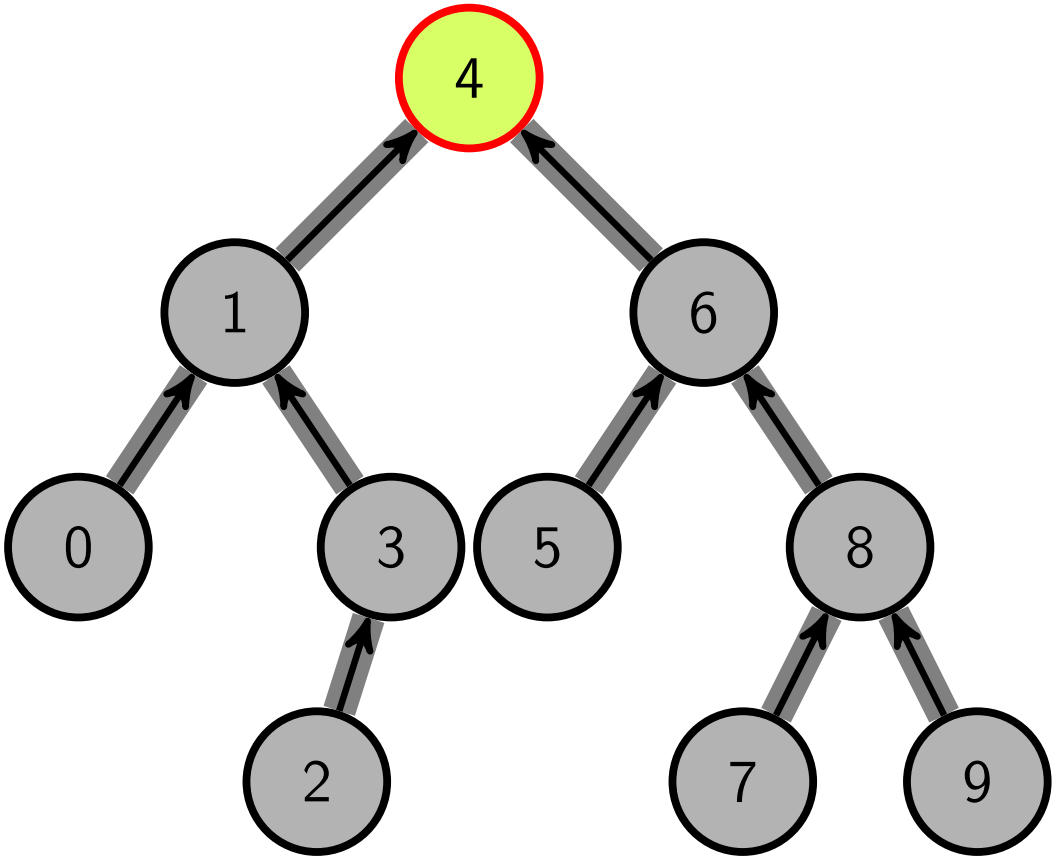


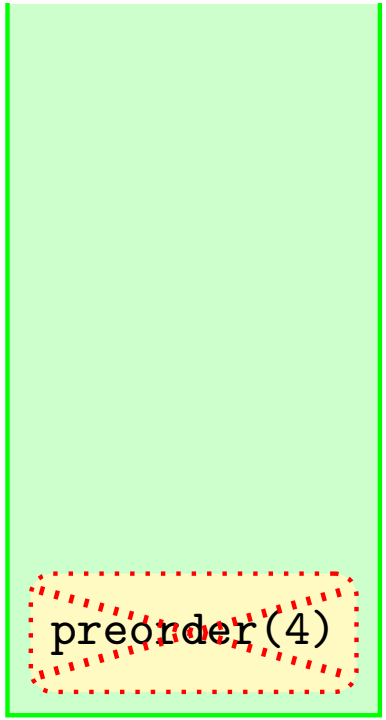


Stiva de apeluri recursive

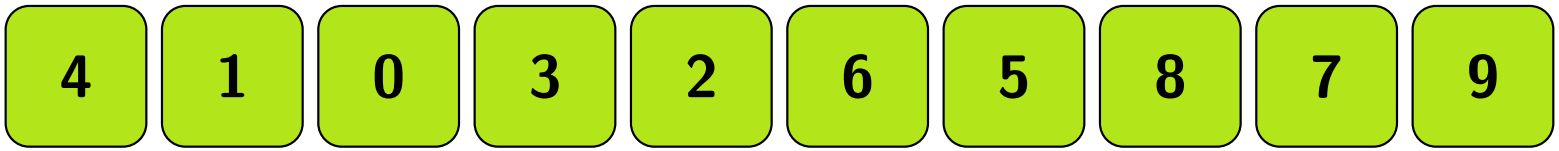


Parcurgerea în preordine

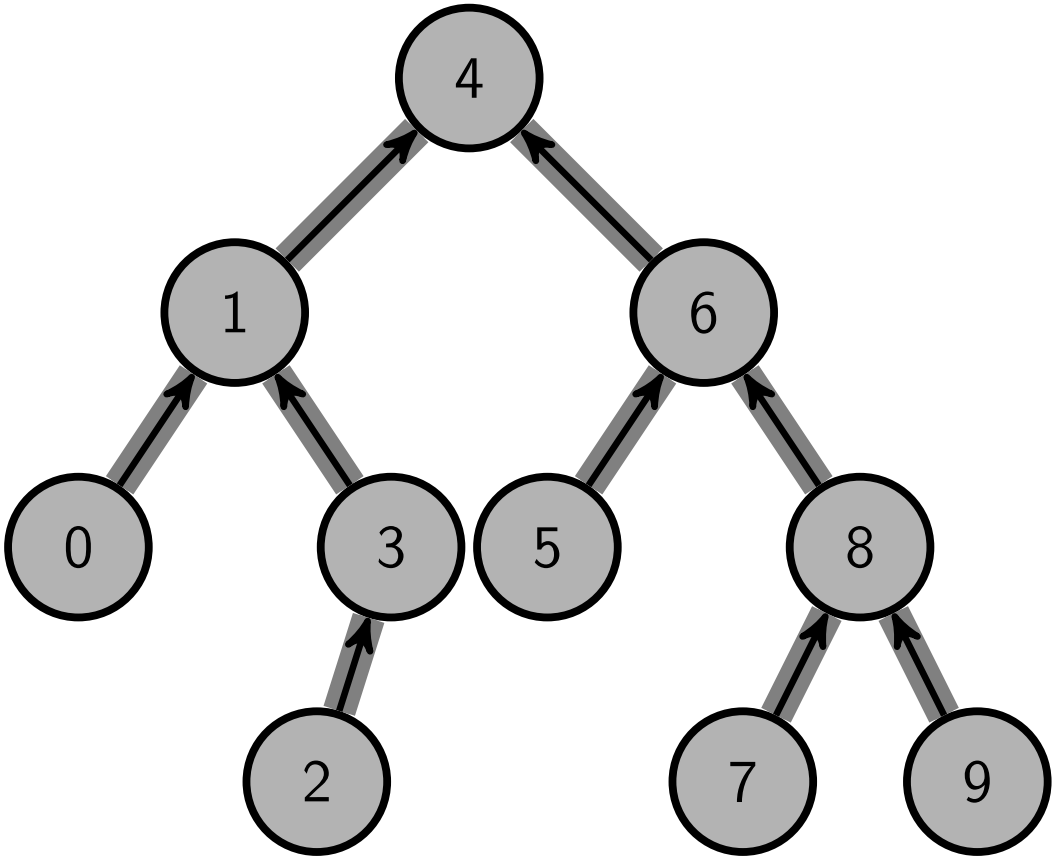


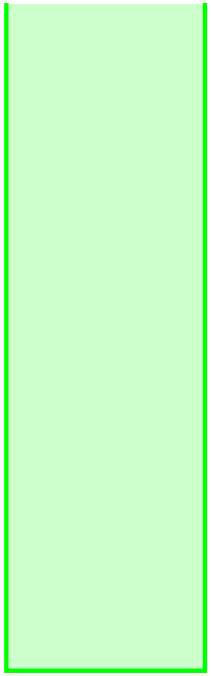


Stiva de apeluri recursive

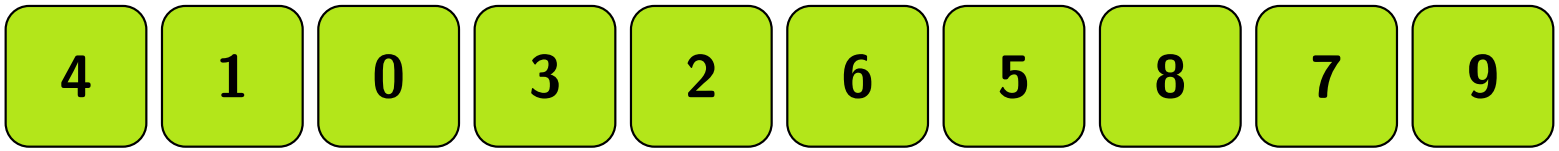


Parcurgerea în preordine

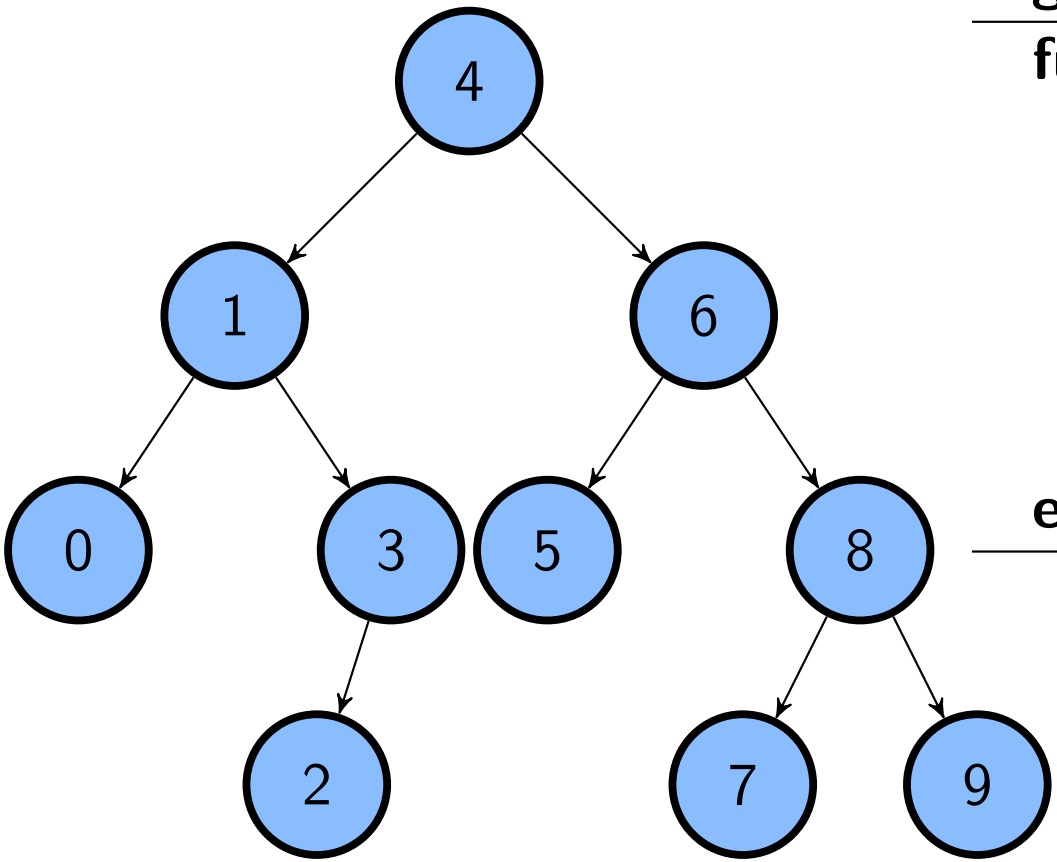




Stiva de apeluri recursive

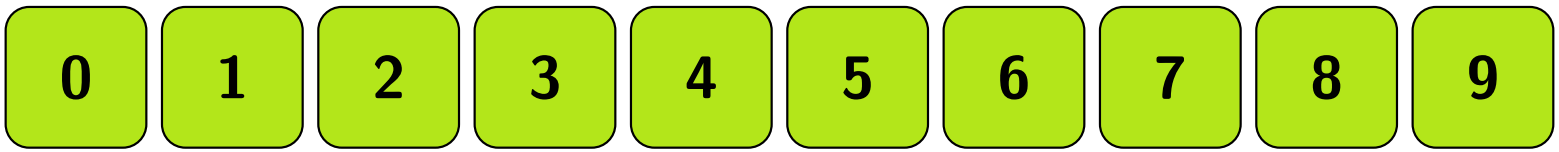


Parcurgerea în inordine

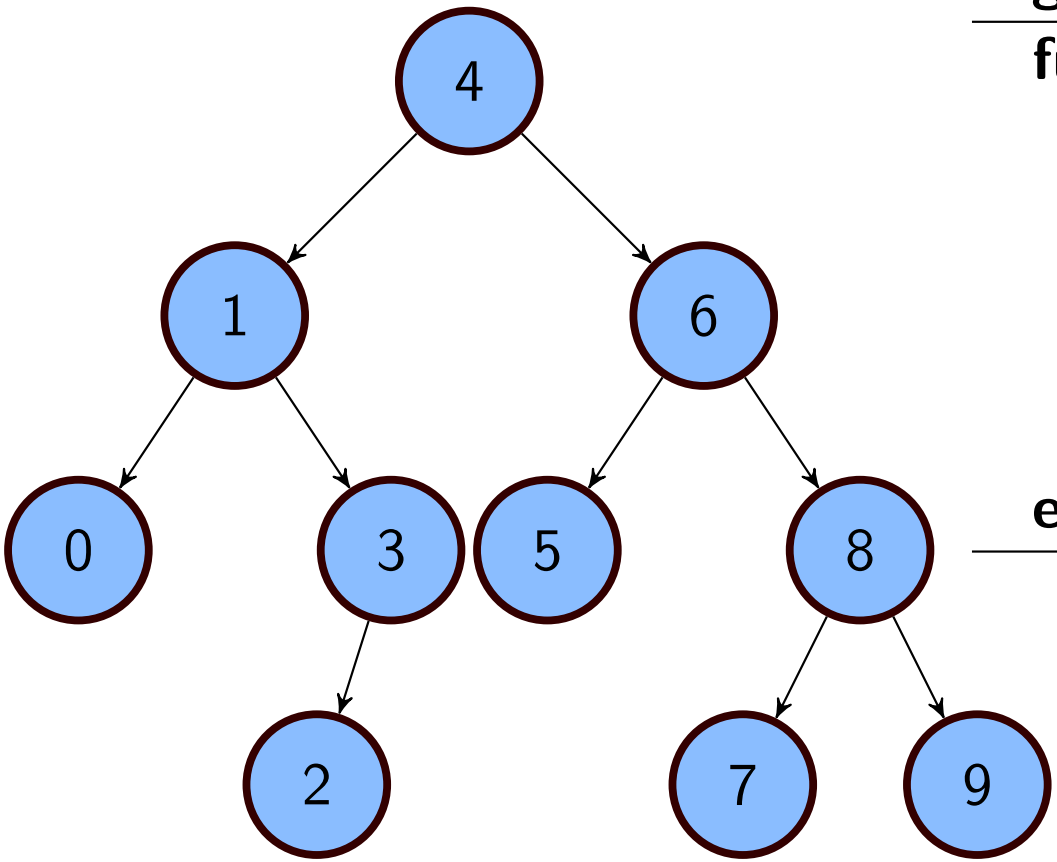


Algorithm 2 Inordine

function INORDER(root)
  if *root* $\neq$ **NULL then**
    INORDER(*root* $\rightarrow$ *left*)
    print(*root* $\rightarrow$ *value*)
    INORDER(*root* $\rightarrow$ *right*)
  end if
end function

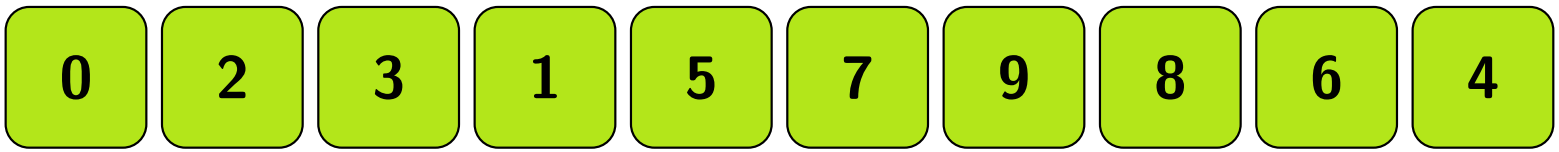


Parcurgerea în postordine

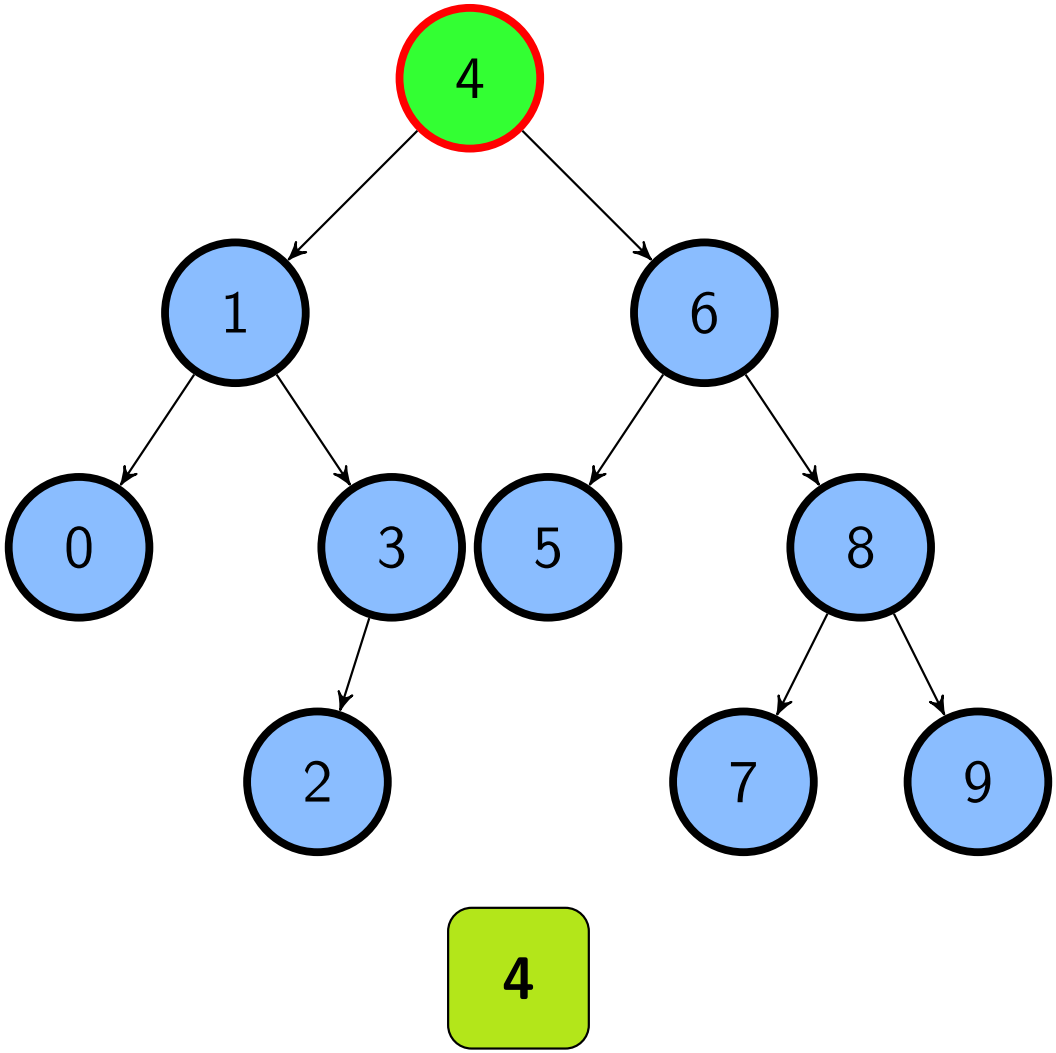


Algorithm 3 Postordine

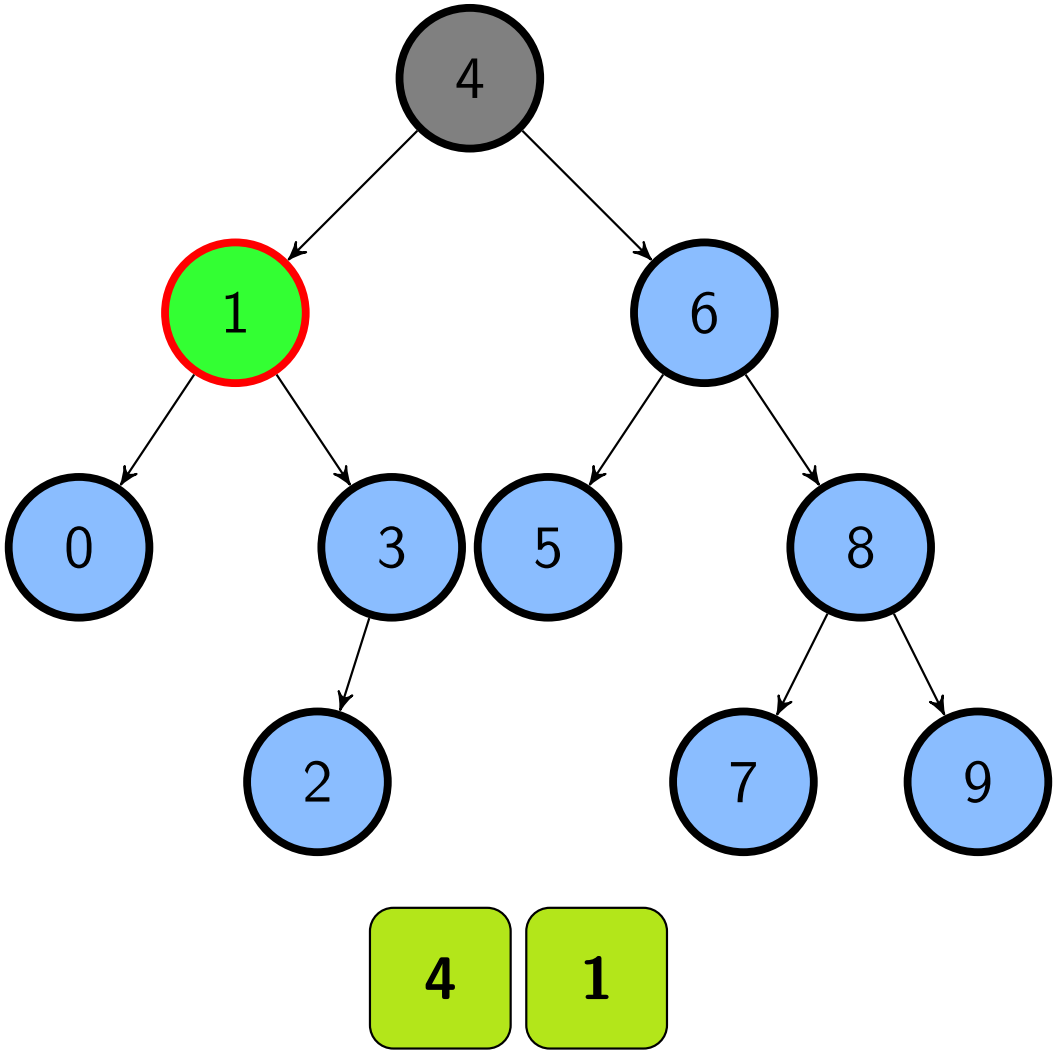
```
function POSTORDER(r)
    if r ≠ NULL then
        POSTORDER(r → left)
        POSTORDER(r → right)
        print(r → value)
    end if
end function
```



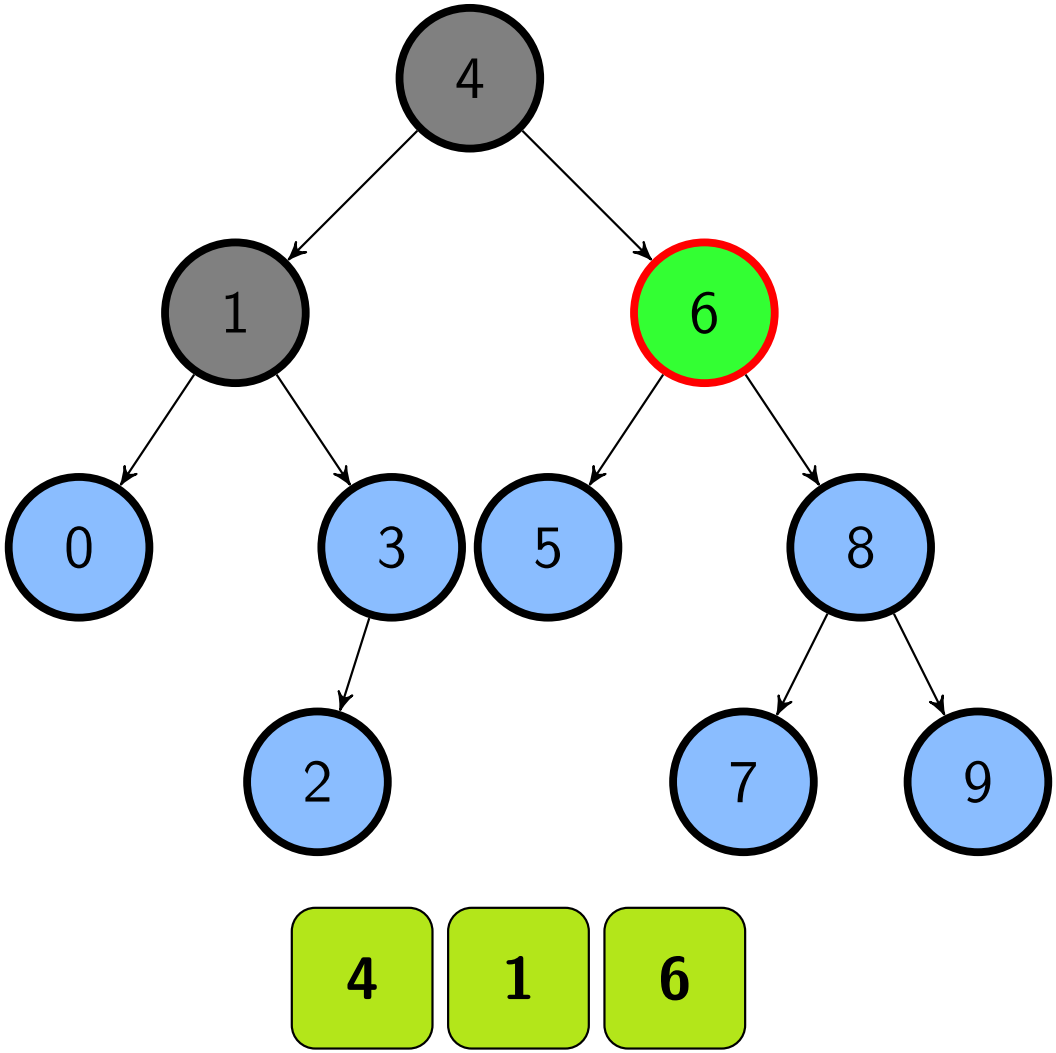
Parcurgerea pe nivel



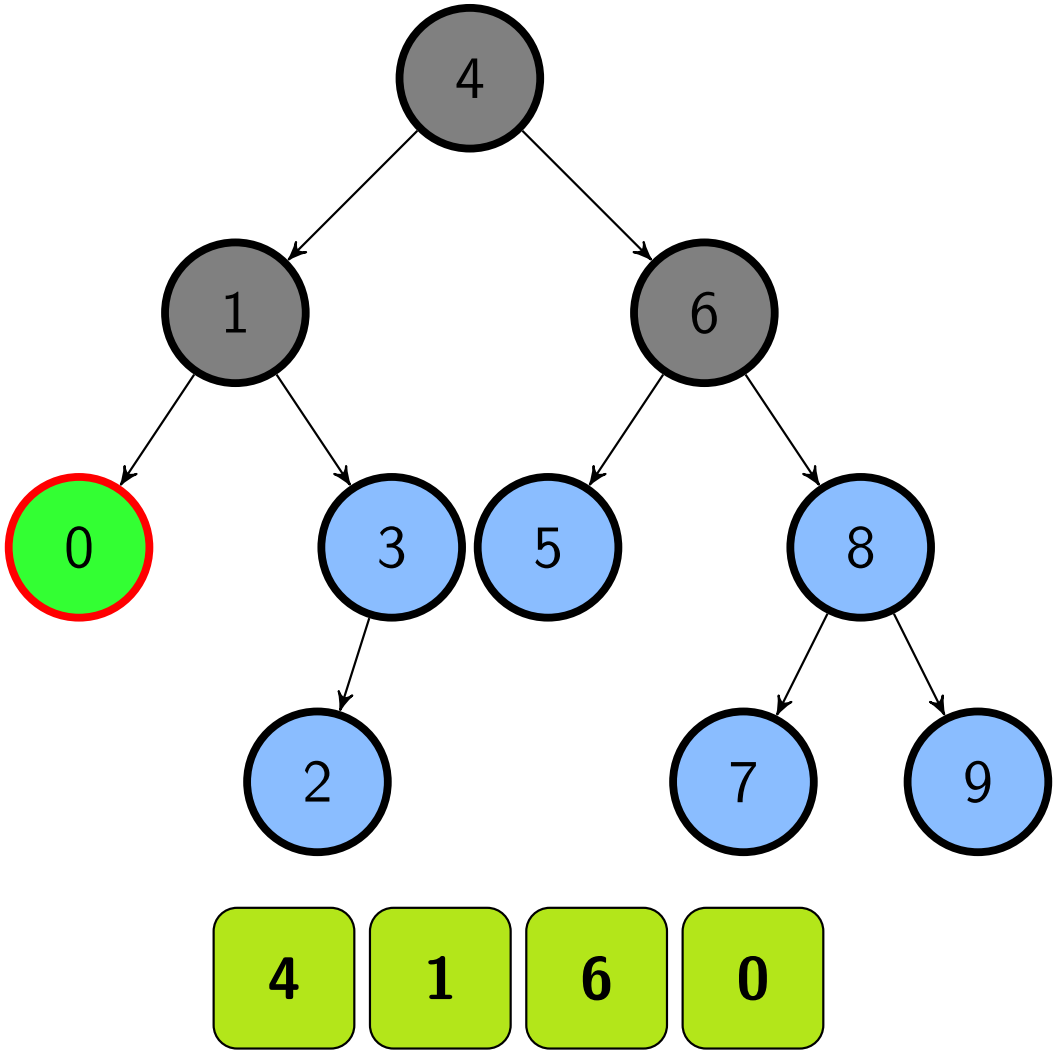
Parcurgerea pe nivel



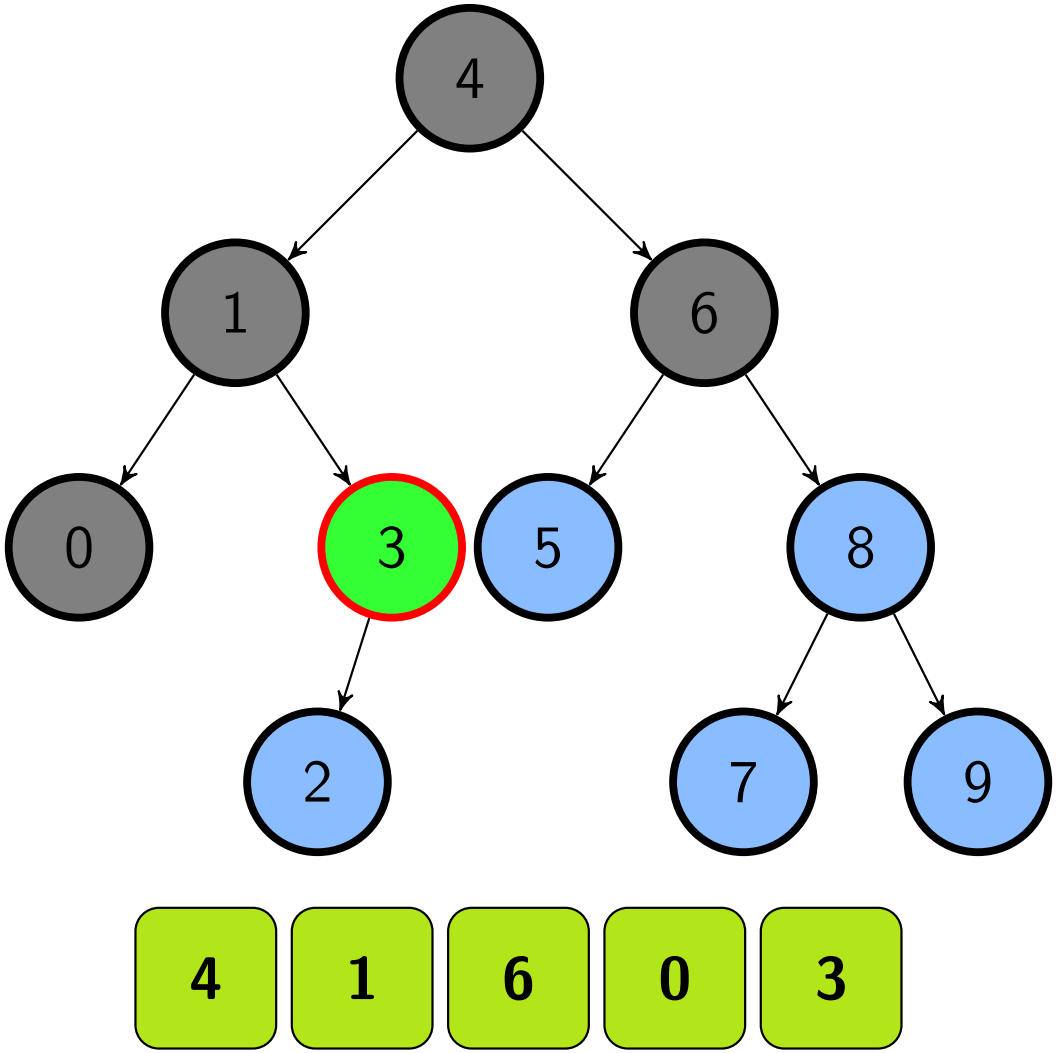
Parcurgerea pe nivel



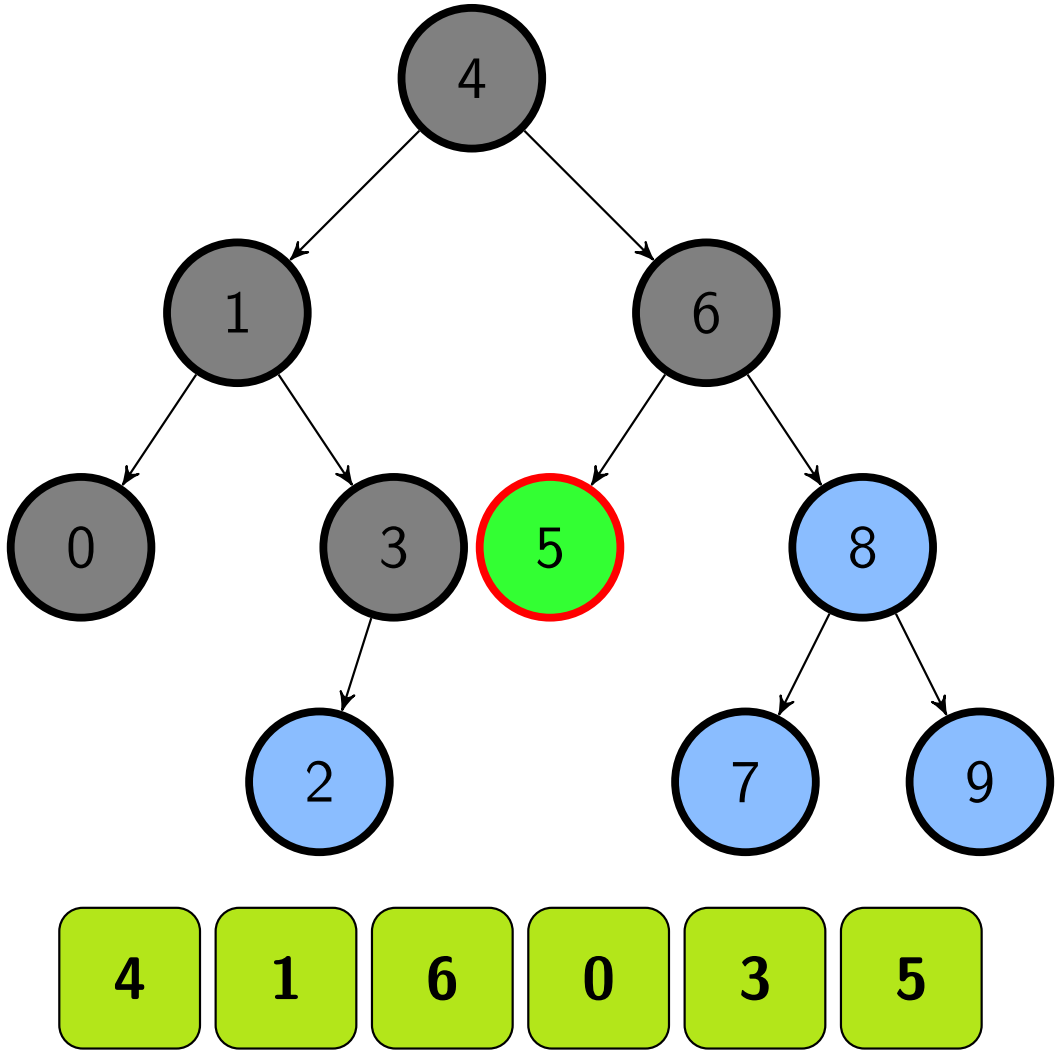
Parcurgerea pe nivel



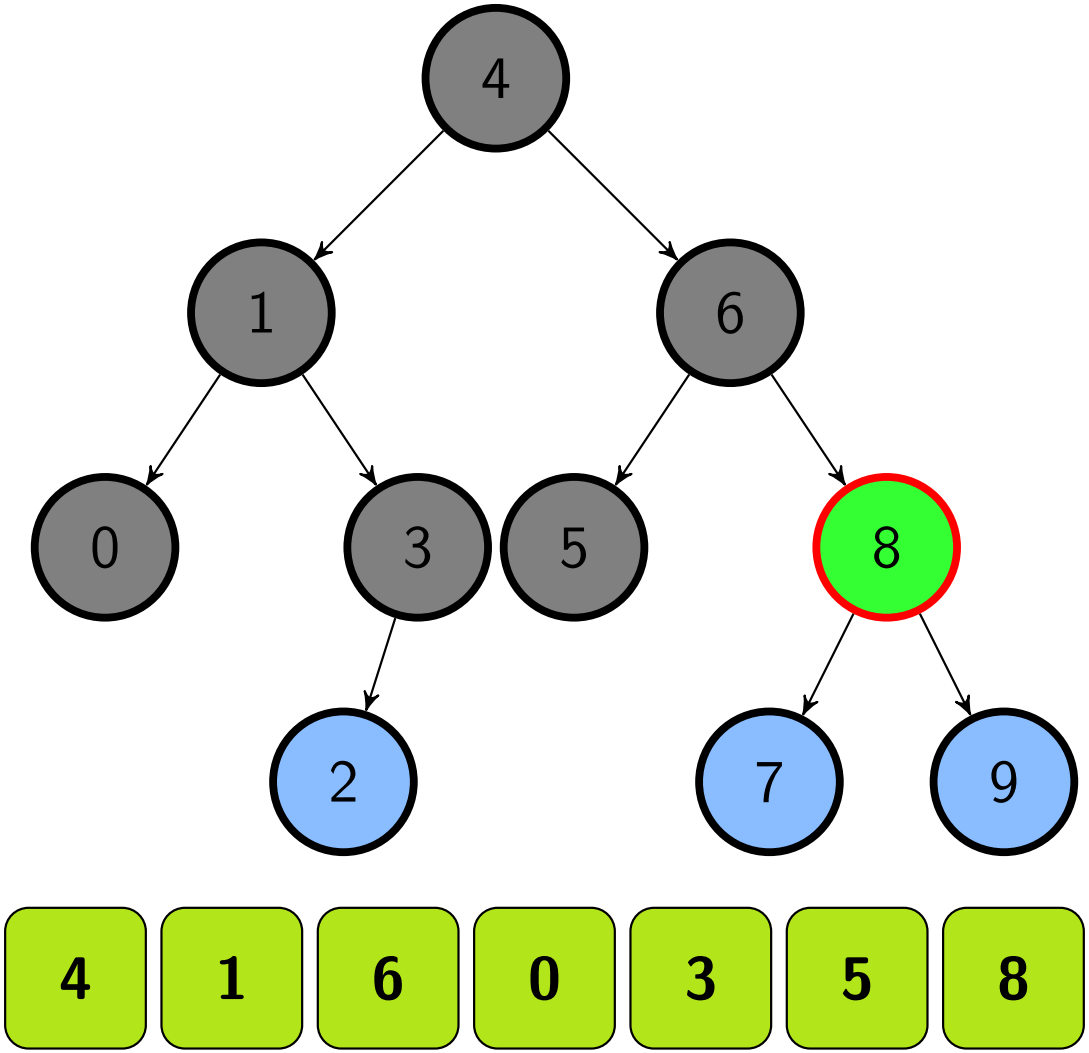
Parcurgerea pe nivel



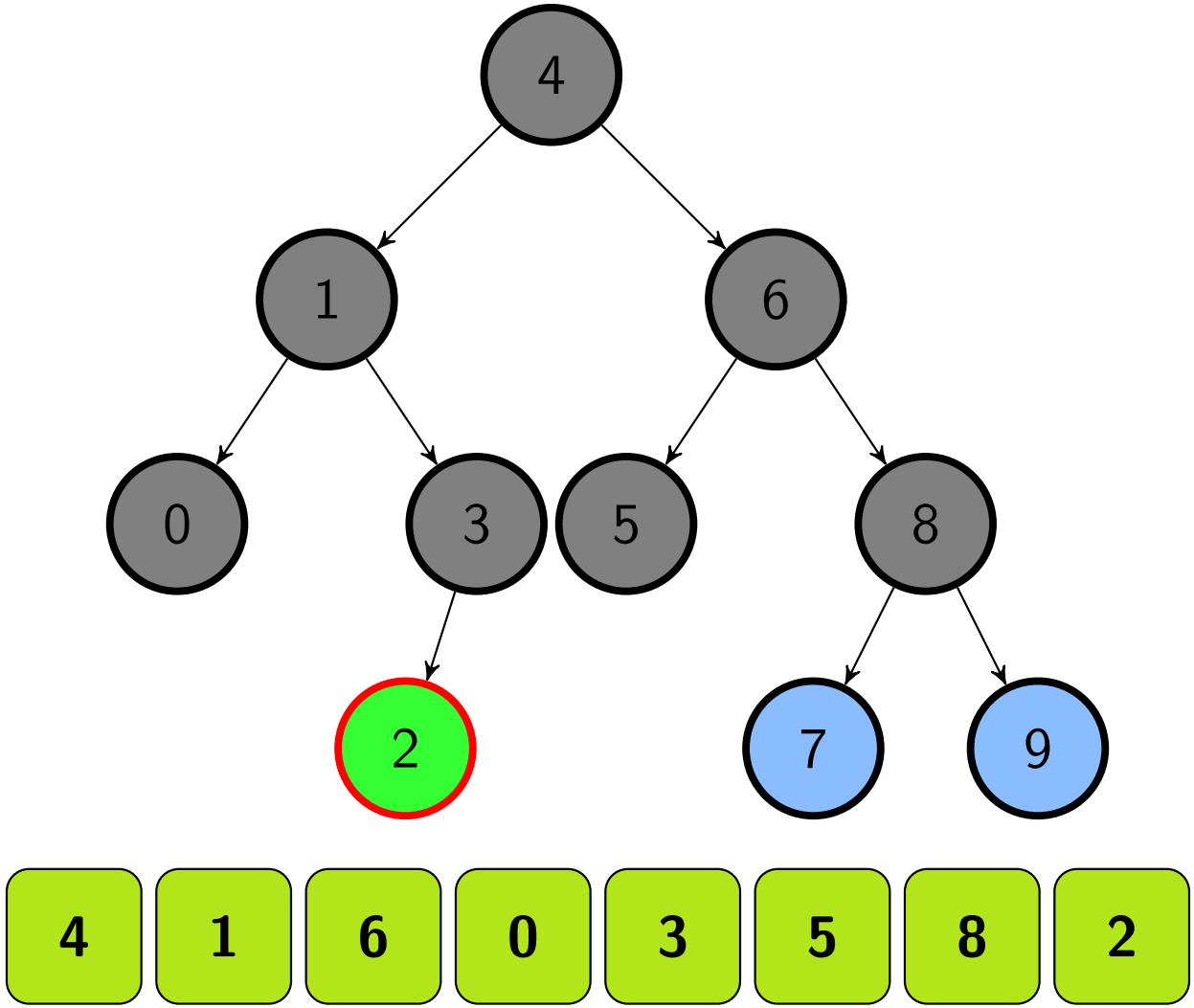
Parcurgerea pe nivel



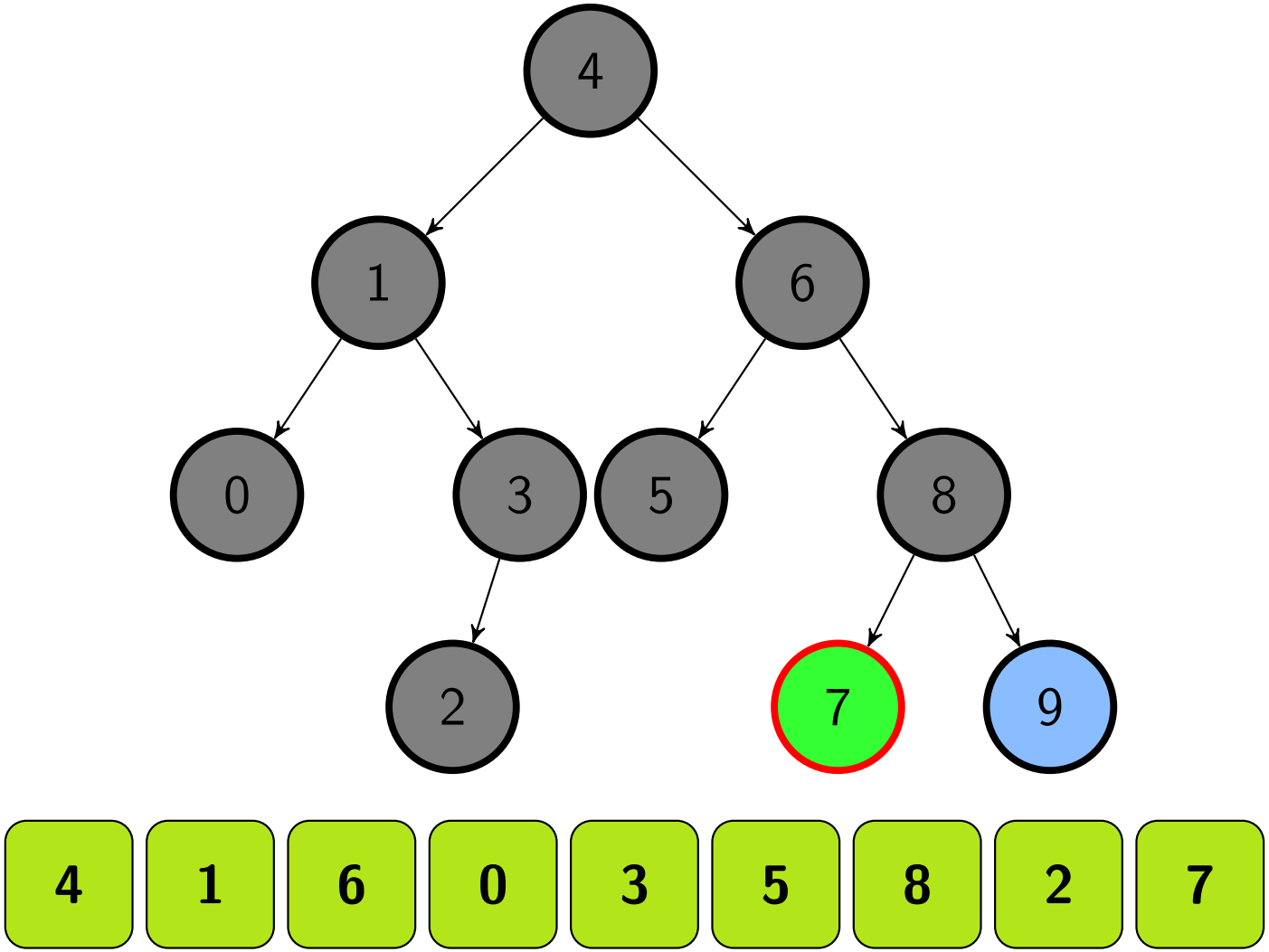
Parcurgerea pe nivel



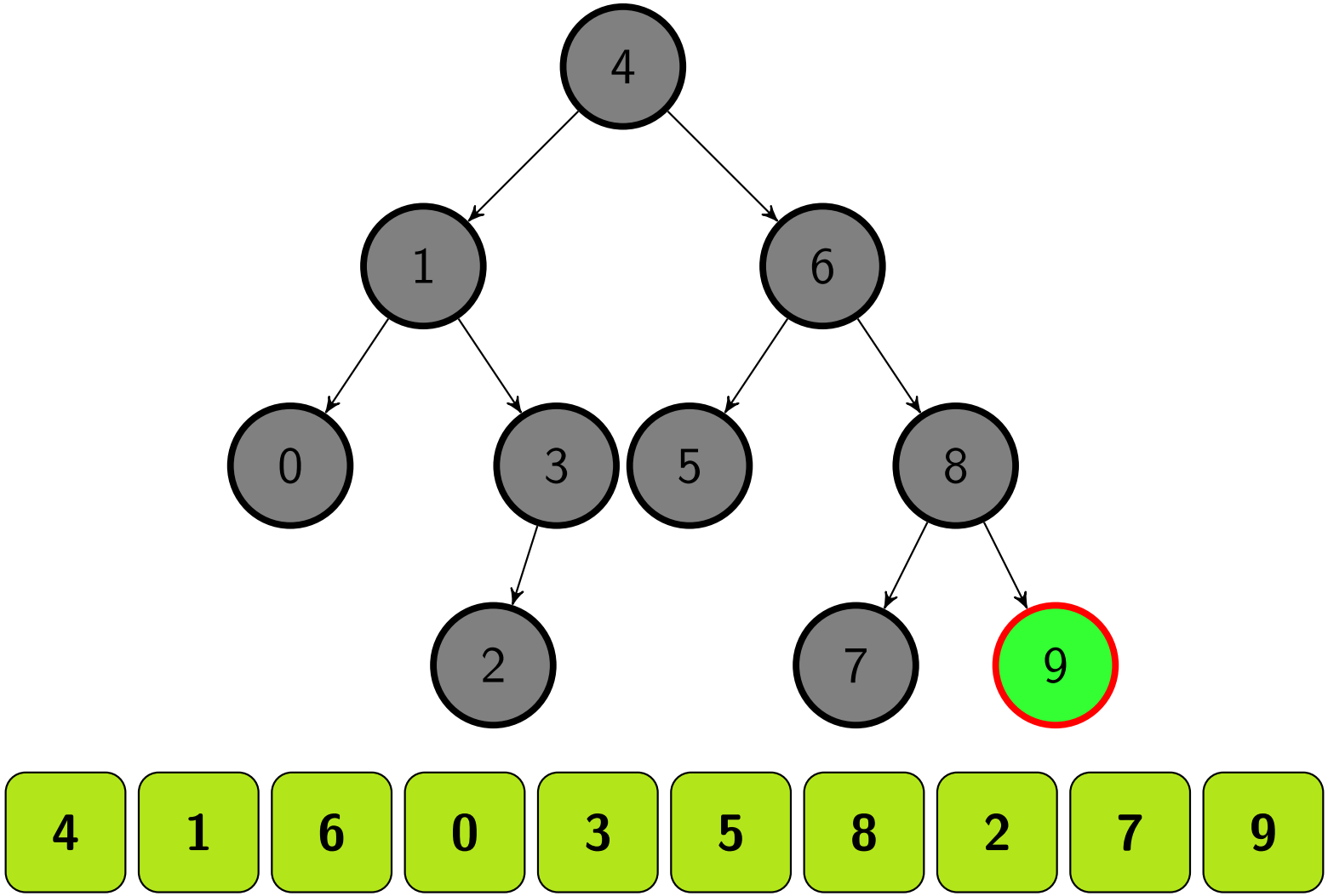
Parcurgerea pe nivel



Parcurgerea pe nivel



Parcurgerea pe nivel



Parcurgerea pe nivel

Cum am putea să implementăm această parcurgere?

Parcurgerea pe nivel

Cum am putea să implementăm această parcurgere?

Putem implementa o funcție recursivă care să afișeze toate nodurile care se află pe un anumit nivel.

Parcurgerea pe nivel

Cum am putea să implementăm această parcurgere?

Putem implementa o funcție recursivă care să afișeze toate nodurile care se află pe un anumit nivel.

```
1  void printLevel(TBinaryTree root, int level) {
2      if (root == NULL)
3          return;
4      if (level == 0) {
5          printf("%d ", root->value);
6          return;
7      }
8      printLevel(root->left, level - 1);
9      printLevel(root->right, level - 1);
10 }
```

Parcurgerea pe nivel

Ce facem cu această funcție?

Parcurgerea pe nivel

Ce facem cu această funcție?

Determinăm înălțimea arborelui și apoi apelăm această funcție pentru toate valorile din mulțimea $\{0, 1, 2, \ldots, h\}$.

Parcurgerea pe nivel

Ce facem cu această funcție?

Determinăm înălțimea arborelui și apoi apelăm această funcție pentru toate valorile din mulțimea $\{0, 1, 2, \ldots, h\}$.

Cum putem determina înălțimea unui arbore binar?

Parcurgerea pe nivel

Ce facem cu această funcție?

Determinăm înălțimea arborelui și apoi apelăm această funcție pentru toate valorile din mulțimea $\{0, 1, 2, \ldots, h\}$.

Cum putem determina înălțimea unui arbore binar?

Vom implementa o funcție recursivă care va calcula maximul dintre înalțimile celor doi sub-arbori.

Parcurgerea pe nivel

Înălțimea unui arbore reprezintă numărul maxim de arce care formează calea de la nodul rădăcină până la cea mai îndepărtată frunză.

```
height(r) = 1 + max(height(r->left), height(r->right))
```

Parcurgerea pe nivel

Înălțimea unui arbore reprezintă numărul maxim de arce care formează calea de la nodul rădăcină până la cea mai îndepărtată frunză.

`height(r) = 1 + max(height(r->left), height(r->right))`

```
11  int height(TBinaryTree root) {
12      if (root==NULL||(root->left==NULL&&root->right==NULL))
13          return 0;
14      int left, right;
15      left = height(root->left);
16      right = height(root->right);
17      if (left < right)
18          return right + 1;
19      return left + 1;
20  }
```

Parcurgerea pe nivel

```
21  void printLevelOrder(TBinaryTree root) {
22      int i, h = height(root);
23      for (i = 0; i <= h; i++)
24          printLevel(root, i);
25      printf("\n");
26  }
```

Parcurgerea pe nivel

```
21  void printLevelOrder(TBinaryTree root) {
22      int i, h = height(root);
23      for (i = 0; i <= h; i++)
24          printLevel(root, i);
25      printf("\n");
26  }
```

De câte ori ajungem să traversăm arborele?

Parcurgerea pe nivel

```
21  void printLevelOrder(TBinaryTree root) {
22      int i, h = height(root);
23      for (i = 0; i <= h; i++)
24          printLevel(root, i);
25      printf("\n");
26  }
```

De câte ori ajungem să traversăm arborele?

Traversăm arborele de foarte multe ori, iar acest lucru face ca funcția să fie ineficientă.

Parcurgerea pe nivel – Varianta eficientă

Cum am putea să implementăm această parcurgere fără a fi nevoie sa traversăm de mai multe ori arborele?

Parcurgerea pe nivel – Varianta eficientă

Cum am putea să implementăm această parcurgere fără a fi nevoie sa traversăm de mai multe ori arborele?

Putem să ne folosim de o coadă în care să inserăm nodurile.

1. Inițial, introducem nodul rădăcină în coadă.
2. Cât timp mai avem noduri în coadă:
   - extragem un nod din coadă;
   - îl prelucrăm;
   - îi adăugăm copiii în coadă (dacă are).

Parcurgerea pe nivel – Varianta eficientă

Avem nevoie de implementarea unei cozi.

```
1  typedef struct node {
2      int value;
3      struct node *left, *right;
4  } *TBinaryTree;
5
6  typedef struct list {
7      TBinaryTree value;
8      struct list *next;
9  } *List;
10
11 typedef struct queue {
12     List front, rear;
13 } Queue;
```

Parcurgerea pe nivel – Varianta eficientă

```
14  Queue initQueue() {
15      Queue q;
16      q.front = NULL;
17      q.rear = NULL;
18      return q;
19  }
20
21  List createList(TBinaryTree value) {
22      List head = malloc(sizeof(struct list));
23      head->value = value;
24      head->next = NULL;
25      return head;
26  }
```

Parcurgerea pe nivel – Varianta eficientă

```
27  int isEmptyQueue(Queue *q) {
28      return q->front == NULL;
29  }
30
31  void enqueue(Queue *q, TBinaryTree node) {
32      if (q->front == NULL)
33          q->front = q->rear = createList(node);
34      else {
35          List list = createList(node);
36          q->rear->next = list;
37          q->rear = list;
38      }
39  }
```

Parcurgerea pe nivel – Varianta eficientă

```
40  TBinaryTree dequeue(Queue *q) {
41      if (q->front == NULL)
42          return NULL;
43      TBinaryTree result = q->front->value;
44      List tmp = q->front;
45      q->front = q->front->next;
46      if (tmp->next == NULL)
47          q->rear = NULL;
48      free(tmp);
49      return result;
50  }
```

Parcurgerea pe nivel – Varianta eficientă

```
51  void printLevelOrder(TBinaryTree root) {
52      if (root == NULL)
53          return;
54      Queue q = initQueue();
55      TBinaryTree current;
56      enqueue(&q, root);
57      while (!isEmptyQueue(&q)) {
58          current = dequeue(&q);
59          if (current != NULL) {
60              printf("%d ", current->value);
61              enqueue(&q, current->left);
62              enqueue(&q, current->right);
63          }
64      }
65      printf("\n");
66  }
```

Parcurgerea în adâncime – Nerecursiv

Cum putem implementa **nerecursiv** parcurgerea în adâncime?

Parcurgerea în adâncime – Nerecursiv

Cum putem implementa **nerecursiv** parcurgerea în adâncime?

Putem utiliza o stivă în care să inserăm nodurile.

1. Inițial, introducem nodul rădăcină în stivă.
2. Cât timp mai avem noduri în stivă:
   - extragem un nod din stivă;
   - îl prelucrăm;
   - îi adăugăm copiii în stivă (dacă are).

Parcurgerea în adâncime – Nerecursiv

Avem nevoie de implementarea unei stive.

```
1  typedef struct node {
2      int value;
3      struct node *left, *right;
4  } *TBinaryTree;
5
6  typedef struct list {
7      TBinaryTree value;
8      struct list *next;
9  } *List;
10
11 typedef struct stack {
12     List head;
13 } Stack;
```

Parcurgerea în adâncime – Nerecursiv

```
14  Stack initStack() {
15      Stack s;
16      s.head = NULL;
17      return s;
18  }
19
20  List createList(TBinaryTree value) {
21      List head = malloc(sizeof(struct list));
22      head->value = value;
23      head->next = NULL;
24      return head;
25  }
```

Parcurgerea în adâncime – Nerecursiv

```
26  int isEmptyStack(Stack *s) {
27      return s->head == NULL;
28  }
29  void push(Stack *s, TBinaryTree node) {
30      List list = createList(node);
31      list->next = s->head;
32      s->head = list;
33  }
34  TBinaryTree pop(Stack *s) {
35      if (s->head == NULL)
36          return NULL;
37      TBinaryTree result = s->head->value;
38      List tmp = s->head;
39      s->head = s->head->next;
40      free(tmp);
41      return result;
42  }
```

Parcurgerea în adâncime – Nerecursiv

```
43  void DepthFirstSearch(TBinaryTree root) {
44      if (root == NULL)
45          return;
46      Stack s = initStack();
47      TBinaryTree current;
48      push(&s, root);
49      while (!isEmptyStack(&s)) {
50          current = pop(&s);
51          if (current != NULL) {
52              printf("%d ", current->value);
53              push(&s, current->right);
54              push(&s, current->left);
55          }
56      }
57      printf("\n");
58  }
```

Vă mulțumesc pentru atenție!