

Structuri de Date și Algoritmi

Arbori binari de căutare

Mihai Nan

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



Anul Universitar 2022–2023

Conținutul cursului

- 1 **Arbore binar de căutare**
- 2 **Implementarea arbore binar de căutare**
 - Inserarea unui element
 - Căutarea unui element
 - Determinarea elementului minim
 - Determinarea elementului maxim
 - Ștergerea unui element
 - Dealocarea memoriei pentru arbore
- 3 **Analizarea eficienței operațiilor**

Arbore binar de căutare – Definiție

Definiție

Un **arbore binar de căutare** este un arbore binar în care pentru fiecare nod intern sunt îndeplinite următoarele proprietăți:

- 1 Cheia din nod este **mai mare** decât **orice cheie din sub-arborele stâng**;
- 2 Cheia din nod este **mai mică** decât **orice cheie din sub-arborele drept**.

Arbore binar de căutare – Definiție

Definiție

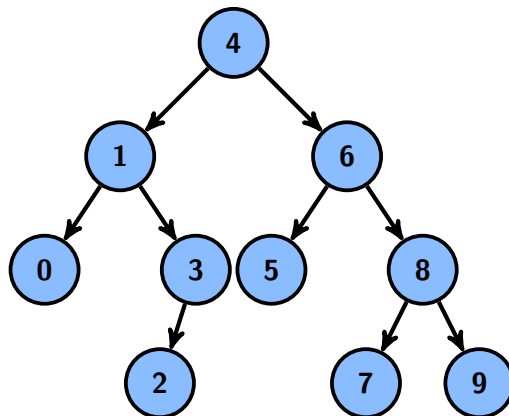
Un **arbore binar de căutare** este un arbore binar în care pentru fiecare nod intern sunt îndeplinite următoarele proprietăți:

- 1 Cheia din nod este **mai mare** decât **orice cheie din sub-arborele stâng**;
- 2 Cheia din nod este **mai mică** decât **orice cheie din sub-arborele drept**.

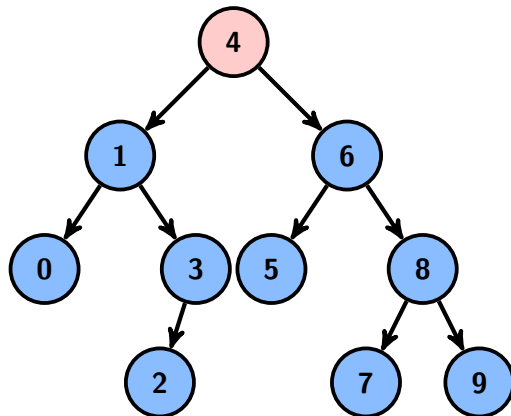
Observație

În general, nu are rost să adăugăm valori duplicate într-un arbore binar de căutare.

Arbore binar de căutare – Exemplu

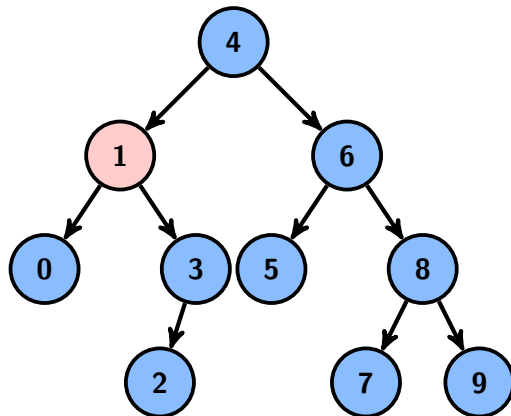


Arbore binar de căutare – Exemplu



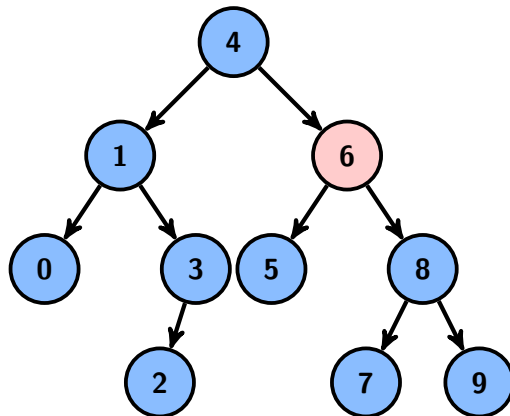
- În stânga lui 4 avem valorile: 0, 1, 2, 3.
- În dreapta lui 4 avem valorile: 5, 6, 7, 8, 9.

Arbore binar de căutare – Exemplu



- În stânga lui 1 avem valorile: 0.
- În dreapta lui 1 avem valorile: 2, 3.

Arbore binar de căutare – Exemplu



- În stânga lui 6 avem valorile: 5.
- În dreapta lui 5 avem valorile: 7, 8, 9.

Arbore binar de căutare – Motivație

1 Vector sortat









- **Adăugarea / Ștergerea unui element:** $O(N)$
- **Căutarea unui element:** $O(\log N)$ – folosind căutare binară

2 Listă sortată



- **Adăugarea / Ștergerea unui element:** $O(1)$ – presupunând că avem un pointer la locația respectivă
- **Căutarea unui element:** $O(N)$

Arbore binar de căutare – Motivație

	Sorted Arrays	Linked Lists	Binary Search Trees*
Search	$O(\log(n))$ 	$O(n)$ 	$O(\log(n))$ 
Insert/Delete	$O(n)$ 	$O(1)$ 	$O(\log(n))$ 

Arbore binar de căutare – Reprezentare



Cum putem reprezenta un arbore binar de căutare? Trebuie să aducem vreo modificare reprezentării din cursul trecut pentru un arbore binar simplu?

Arbore binar de căutare – Reprezentare



Cum putem reprezenta un arbore binar de căutare? Trebuie să aducem vreo modificare reprezentării din cursul trecut pentru un arbore binar simplu?



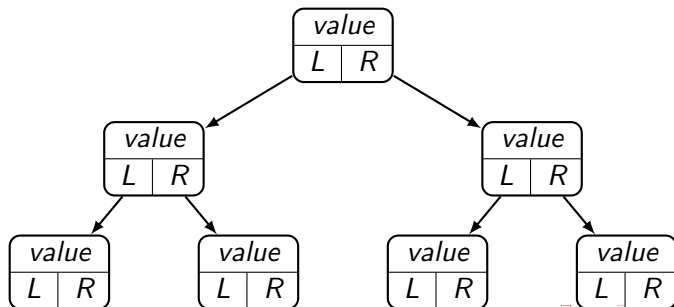
Putem utiliza aceeași reprezentare.

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

Arbore binar de căutare – Reprezentare

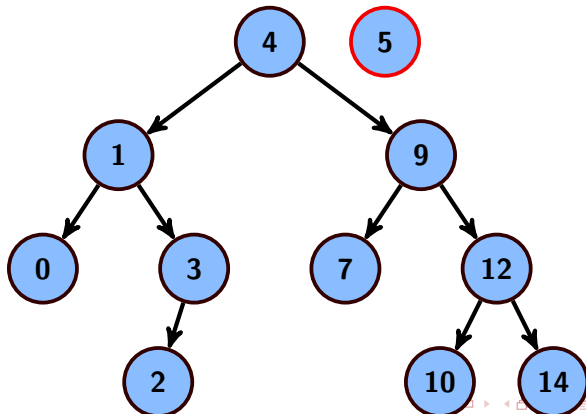
```
1  typedef int T;  
2  typedef struct node {  
3      T value;  
4      struct node* left;  
5      struct node* right;  
6  } Node, *TBinaryTree;
```

Reprezentarea grafică pentru această definiție:



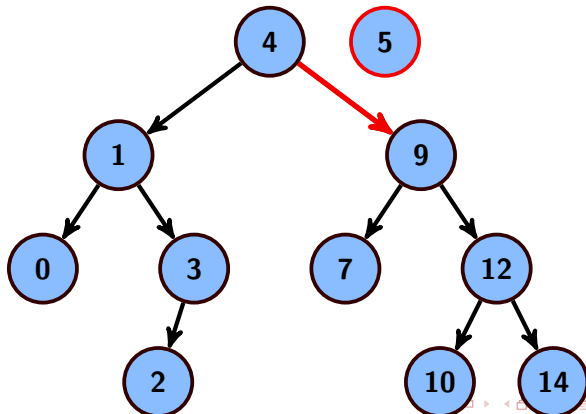
Inserarea unui element

- Pornind de la următorul arbore în care dorim să introducem nodul cu valoarea 5.
- Acesta se va insera ca nod frunză. Pentru a-l insera, va trebui să căutăm o poziție în arbore care respectă regula de integritate a arborilor binari de căutare.



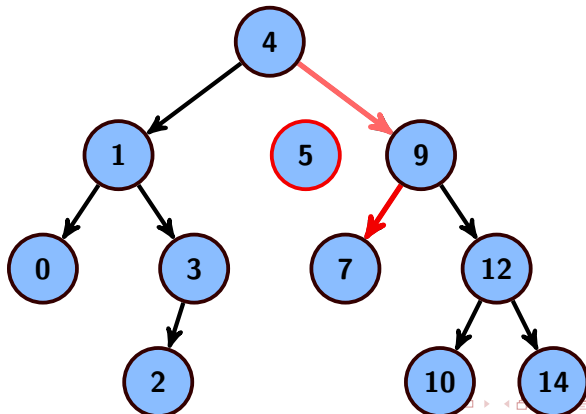
Inserarea unui element

- Pornind de la următorul arbore în care dorim să introducem nodul cu valoarea 5.
- Acesta se va insera ca nod frunză. Pentru a-l insera, va trebui să căutăm o poziție în arbore care respectă regula de integritate a arborilor binari de căutare.



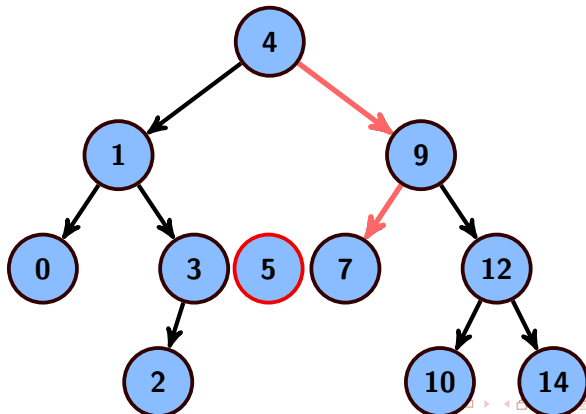
Inserarea unui element

- Pornind de la următorul arbore în care dorim să introducem nodul cu valoarea 5.
- Acesta se va insera ca nod frunză. Pentru a-l insera, va trebui să căutăm o poziție în arbore care respectă regula de integritate a arborilor binari de căutare.



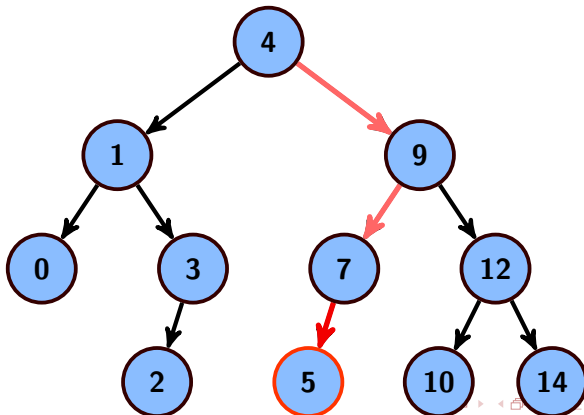
Inserarea unui element

- Pornind de la următorul arbore în care dorim să introducem nodul cu valoarea 5.
- Acesta se va insera ca nod frunză. Pentru a-l insera, va trebui să căutăm o poziție în arbore care respectă regula de integritate a arborilor binari de căutare.



Inserarea unui element

- Pornind de la următorul arbore în care dorim să introducem nodul cu valoarea 5.
- Acesta se va insera ca nod frunză. Pentru a-l insera, va trebui să căutăm o poziție în arbore care respectă regula de integritate a arborilor binari de căutare.



Inserarea unui element



Cum putem implementa această operație de inserare?

Inserarea unui element



Cum putem implementa această operație de inserare?



Putem considera o funcție recursivă care caută poziția corespunzătoare din arbore și apoi inserează un nod frunză în poziția respectivă.

Inserarea unui element



Cum putem implementa această operație de inserare?



Putem considera o funcție recursivă care caută poziția corespunzătoare din arbore și apoi inserează un nod frunză în poziția respectivă.



Când știm că trebuie să oprim căutarea?

Inserarea unui element



Cum putem implementa această operație de inserare?



Putem considera o funcție recursivă care caută poziția corespunzătoare din arbore și apoi inserează un nod frunză în poziția respectivă.



Când știm că trebuie să oprim căutarea?



Când întâlnim **NULL**!

Inserarea unui element

```
1  TBinaryTree createTree(T value) {
2      TBinaryTree root = malloc(sizeof(Node));
3      root->value = value;
4      root->left = root->right = NULL;
5      return root;
6  }
7  TBinaryTree insert(TBinaryTree root, T value) {
8      if (root == NULL)
9          return createTree(value);
10     if (root->value == value)
11         return root;
12     else if (root->value > value)
13         root->left = insert(root->left, value);
14     else
15         root->right = insert(root->right, value);
16     return root;
17 }
```

Căutarea unui element



Cum putem implementa operația de căutare a unui element?

Căutarea unui element



Cum putem implementa operația de căutare a unui element?



Putem considera o funcție recursivă în care avem de tratat următoarele cazuri:

- ❶ Dacă ajungem la **NULL** → valoarea căutată nu există;
- ❷ Dacă ajungem la un nod ce conține o valoare egală cu elementul căutat → oprim căutarea;
- ❸ Dacă ajungem la un nod ce conține o valoare mai mare decât elementul căutat → apelăm recursiv funcția de căutare pentru sub-arborele stâng;
- ❹ Dacă ajungem la un nod ce conține o valoare mai mică decât elementul căutat → apelăm recursiv funcția de căutare pentru sub-arborele drept.

Căutarea unui element

```
18  int contains(TBinaryTree root, T value) {
19      if (root == NULL)
20          return 0;
21      if (root->value == value)
22          return 1;
23      else if (root->value > value)
24          return contains(root->left, value);
25      else
26          return contains(root->right, value);
27  }
```

Căutarea unui element

```
18  int contains(TBinaryTree root, T value) {  
19      if (root == NULL)  
20          return 0;  
21      if (root->value == value)  
22          return 1;  
23      else if (root->value > value)  
24          return contains(root->left, value);  
25      else  
26          return contains(root->right, value);  
27  }
```



Ce complexitate are această funcție?

Căutarea unui element

```
18  int contains(TBinaryTree root, T value) {  
19      if (root == NULL)  
20          return 0;  
21      if (root->value == value)  
22          return 1;  
23      else if (root->value > value)  
24          return contains(root->left, value);  
25      else  
26          return contains(root->right, value);  
27  }
```



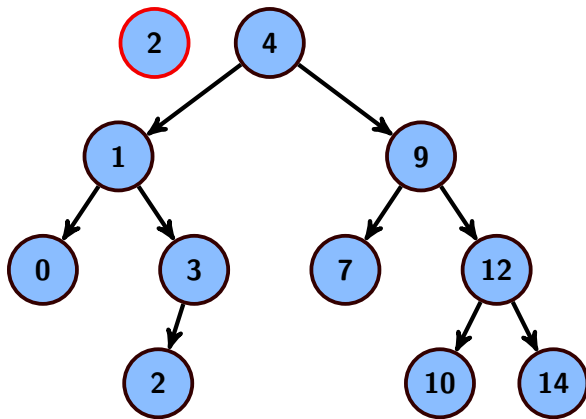
Ce complexitate are această funcție?



$O(H)$ unde H este înălțimea arborelui binar de căutare

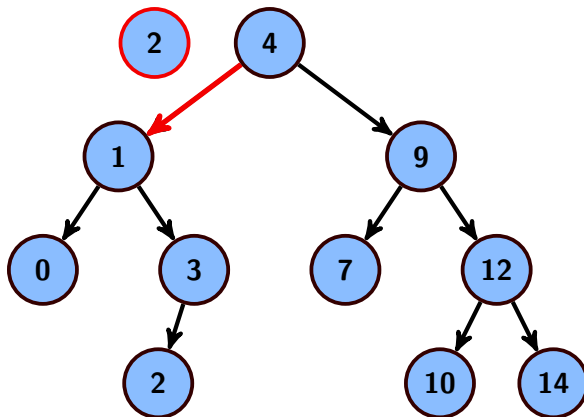
Căutarea unui element

- Pornim de la următorul arbore și căutăm valoarea 2.



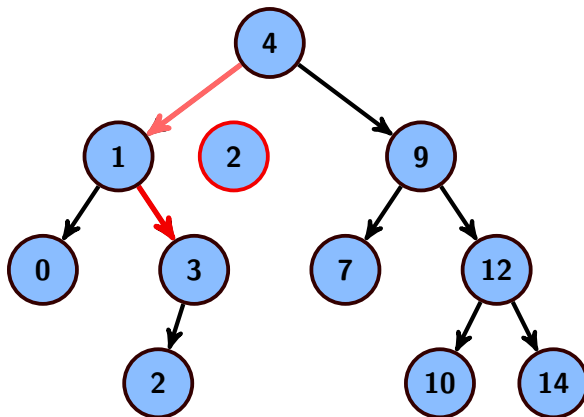
Căutarea unui element

- Pornim de la următorul arbore și căutăm valoarea 2.



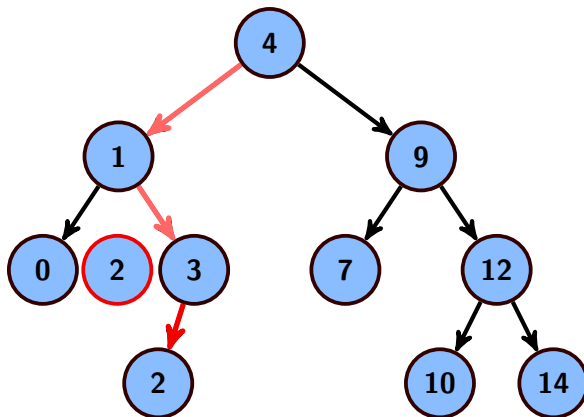
Căutarea unui element

- Pornim de la următorul arbore și căutăm valoarea 2.



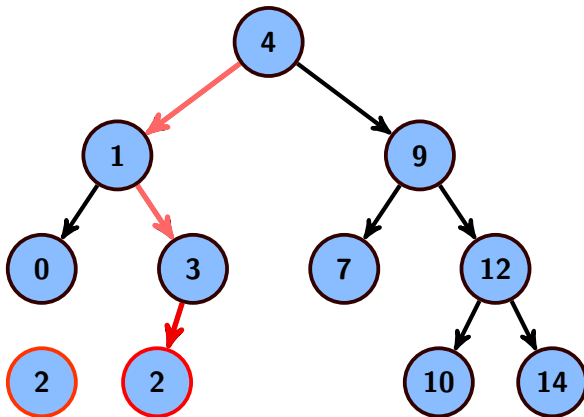
Căutarea unui element

- Pornim de la următorul arbore și căutăm valoarea 2.



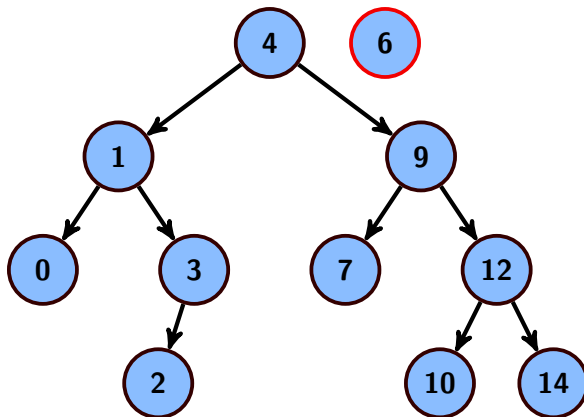
Căutarea unui element

- Pornim de la următorul arbore și căutăm valoarea 2.



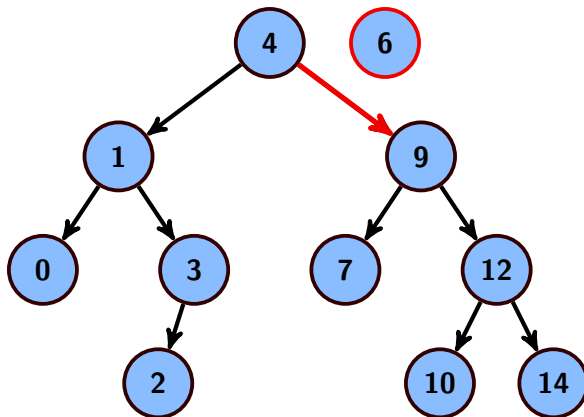
Căutarea unui element

- Pornim de la următorul arbore și căutăm valoarea 6.



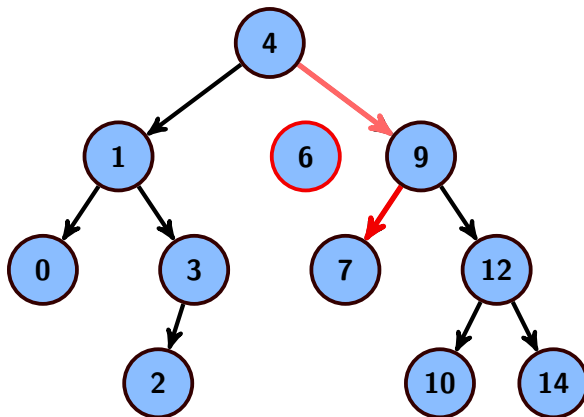
Căutarea unui element

- Pornim de la următorul arbore și căutăm valoarea 6.



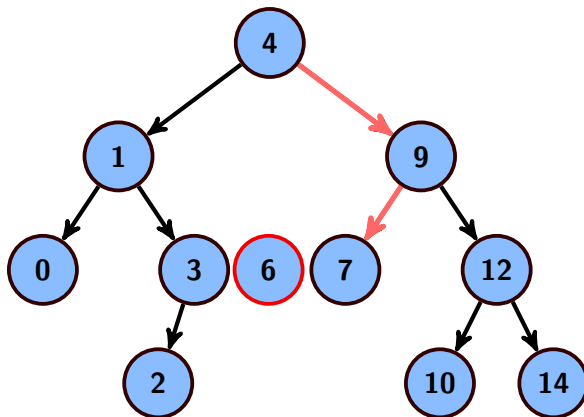
Căutarea unui element

- Pornim de la următorul arbore și căutăm valoarea 6.



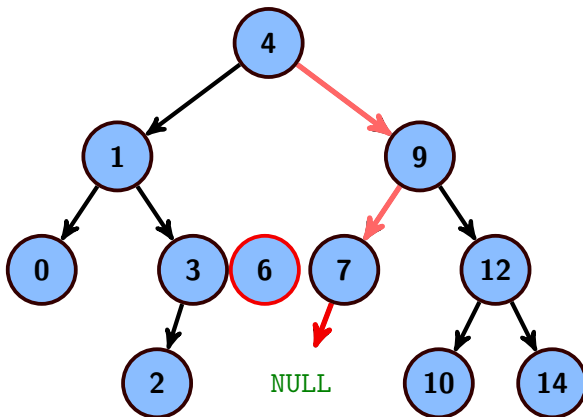
Căutarea unui element

- Pornim de la următorul arbore și căutăm valoarea 6.



Căutarea unui element

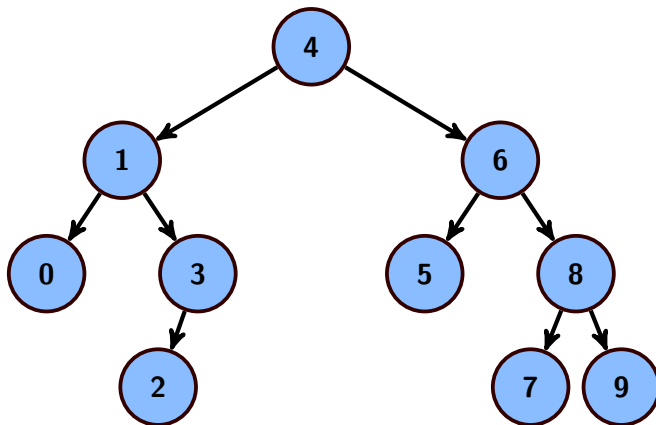
- Pornim de la următorul arbore și căutăm valoarea 6.



Determinarea elementului minim



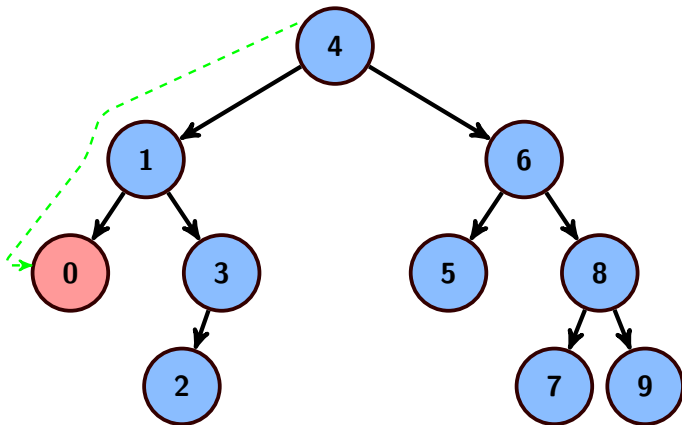
Cum putem determina elementul minim dintr-un arbore binar de căutare?



Determinarea elementului minim



Elementul minim se găsește în nodul cel mai din stânga.



Determinarea elementului minim

```
28 TBinaryTree minimum(TBinaryTree root) {  
29     if (root == NULL || root->left == NULL)  
30         return root;  
31     return minimum(root->left);  
32 }
```

Determinarea elementului minim

```
28 TBinaryTree minimum(TBinaryTree root) {  
29     if (root == NULL || root->left == NULL)  
30         return root;  
31     return minimum(root->left);  
32 }
```



Putem implementa această funcție și în varianta iterativă?

Determinarea elementului minim

```
28 TBinaryTree minimum(TBinaryTree root) {  
29     if (root == NULL || root->left == NULL)  
30         return root;  
31     return minimum(root->left);  
32 }
```



Putem implementa această funcție și în varianta iterativă?



Da, putem folosi `while` pentru a determina nodul cel mai din stânga.

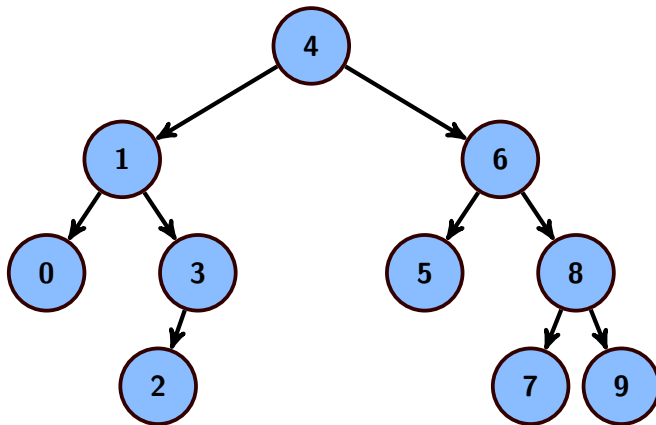
Determinarea elementului minim

```
28 TBinaryTree minimum(TBinaryTree root) {  
29     if (root == NULL)  
30         return root;  
31     while (root->left != NULL)  
32         root = root->left;  
33     return root;  
34 }
```

Determinarea elementului maxim



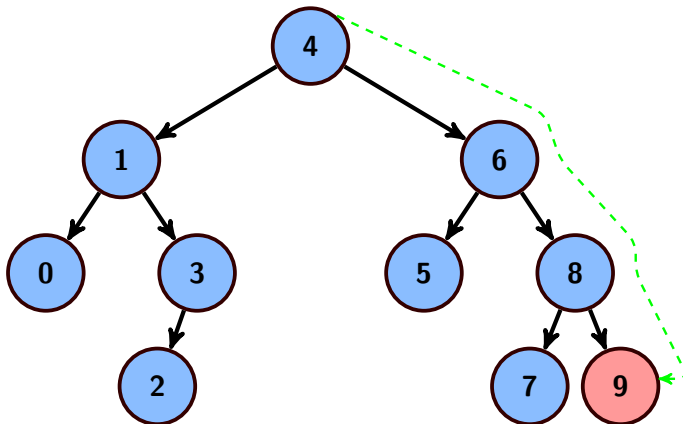
Cum putem determina elementul maxim dintr-un arbore binar de căutare?



Determinarea elementului maxim



Elementul maxim se găsește în nodul cel mai din dreapta.



Determinarea elementului maxim

```
33 TBinaryTree maximum(TBinaryTree root) {  
34     if (root == NULL || root->right == NULL)  
35         return root;  
36     return maximum(root->right);  
37 }
```

Determinarea elementului maxim

```
33 TBinaryTree maximum(TBinaryTree root) {  
34     if (root == NULL || root->right == NULL)  
35         return root;  
36     return maximum(root->right);  
37 }
```

- Sau varianta iterativă

```
33 TBinaryTree maximum(TBinaryTree root) {  
34     if (root == NULL)  
35         return root;  
36     while (root->right != NULL)  
37         root = root->right;  
38     return root;  
39 }
```


Ștergerea unui element

- Operația de ștergere a unui nod presupune următoarele etape:
 - se caută nodul care va fi șters;
 - după ce am găsit nodul, apar trei cazuri pe care le analizăm separat.
- Cele trei cazuri posibile sunt următoarele:
 - 1 nodul care urmează să fie șters este o frunză (nu are fii);
 - 2 nodul are un singur fiu;
 - 3 nodul are doi fii.

Ștergerea unui element

- Operația de ștergere a unui nod presupune următoarele etape:
 - se caută nodul care va fi șters;
 - după ce am găsit nodul, apar trei cazuri pe care le analizăm separat.
- Cele trei cazuri posibile sunt următoarele:
 - 1 nodul care urmează să fie șters este o frunză (nu are fii);
 - 2 nodul are un singur fiu;
 - 3 nodul are doi fii.

Important

Eliminarea unui element (nod) trebuie să se facă astfel încât să se respecte condiția de **arbore binar de căutare** după eliminare.

Ștergerea unui element – Cazul I

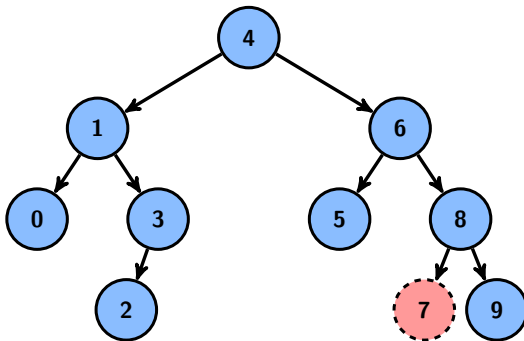


Pentru a șterge un nod frunză se modifică pointerul reținut în nodul părinte, atribuindu-i acestuia valoarea **NULL**. Trebuie să eliminăm explicit nodul din memorie, apelând funcția `free`.

Ștergerea unui element – Cazul I



Pentru a șterge un nod frunză se modifică pointerul reținut în nodul părinte, atribuindu-i acestuia valoarea **NULL**. Trebuie să eliminăm explicit nodul din memorie, apelând funcția `free`.



Eliminarea nodului cu valoare 7

Ștergerea unui element – Cazul I

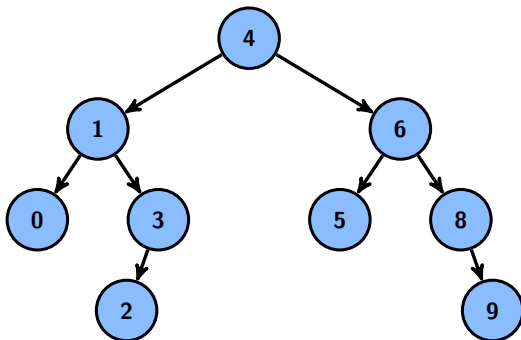


Pentru a șterge un nod frunză se modifică pointerul reținut în nodul părinte, atribuindu-i acestuia valoarea **NULL**. Trebuie să eliminăm explicit nodul din memorie, apelând funcția `free`.

Ștergerea unui element – Cazul I



Pentru a șterge un nod frunză se modifică pointerul reținut în nodul părinte, atribuindu-i acestuia valoarea **NULL**. Trebuie să eliminăm explicit nodul din memorie, apelând funcția `free`.

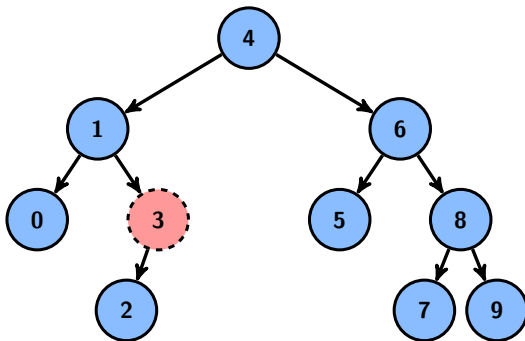


Arborele rezultat în urma operației de ștergere

Ștergerea unui element – Cazul II



Nodul șters are un unic descendent. Vom tăia nodul din aceasta secvență, conectând direct singurul fiu al nodului cu nodul părinte (modificăm pointerul reținut în părinte).

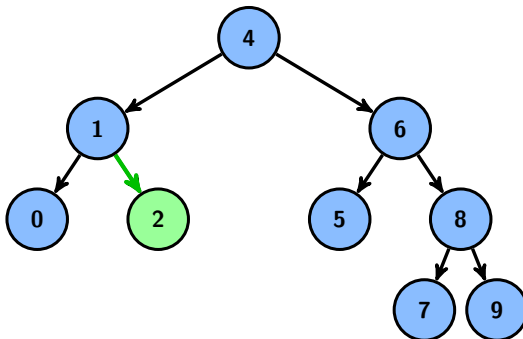


Eliminarea nodului cu valoare 3

Ștergerea unui element – Cazul II

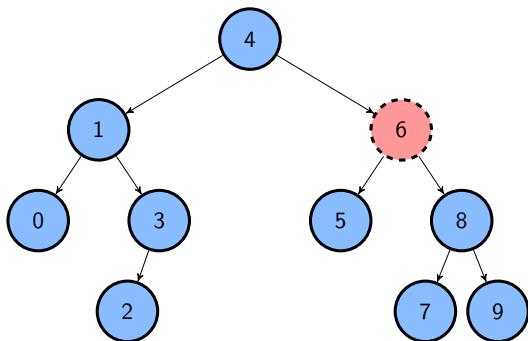


Nodul șters are un unic descendent. Vom tăia nodul din aceasta secvență, conectând direct singurul fiu al nodului cu nodul părinte (modificăm pointerul reținut în părinte).



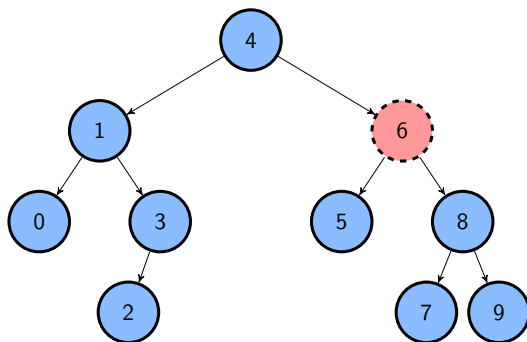
Arborele rezultat în urma operației de ștergere

Ștergerea unui element – Cazul III



Eliminarea nodului cu valoare 6

Ștergerea unui element – Cazul III

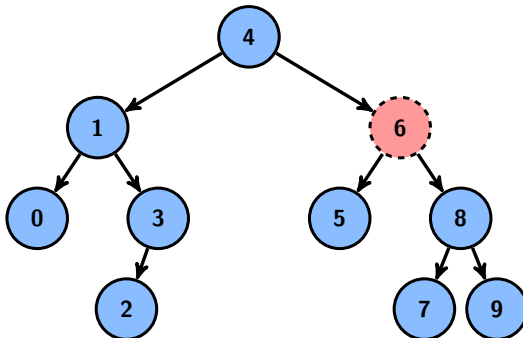


Eliminarea nodului cu valoare 6



Ce am putea face în această situație pentru a păstra proprietatea de arbore binar de căutare?

Ștergerea unui element – Cazul III

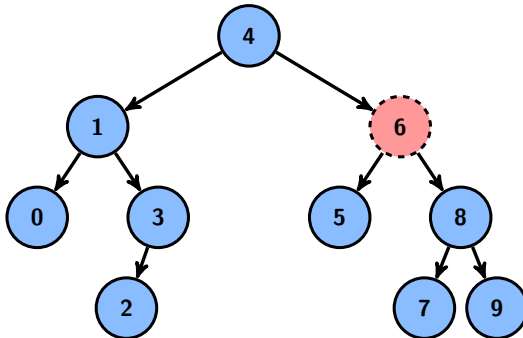


Eliminarea nodului cu valoare 6



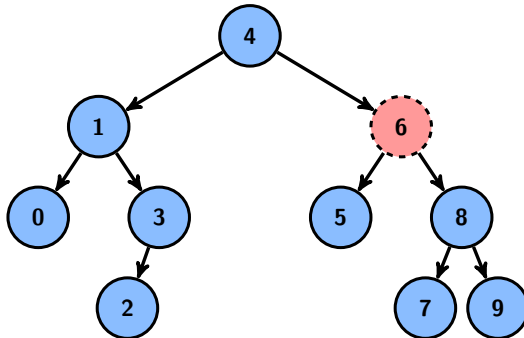
Valoarea pe care o adăugăm în locul lui 6 trebuie să fie mai mare decât orice valoare din sub-arborele stâng și mai mică decât orice valoare din sub-arborele drept.

Ștergerea unui element – Cazul III



Ce valoare este mai mare decât toate din sub-arborele stâng?

Ștergerea unui element – Cazul III

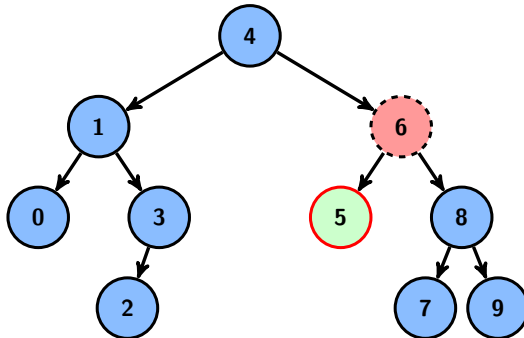


Ce valoare este mai mare decât toate din sub-arborele stâng?



Maximul din sub-arborele stâng!

Ștergerea unui element – Cazul III

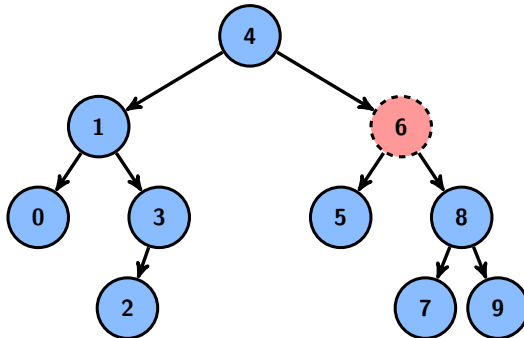


Ce valoare este mai mare decât toate din sub-arborele stâng?



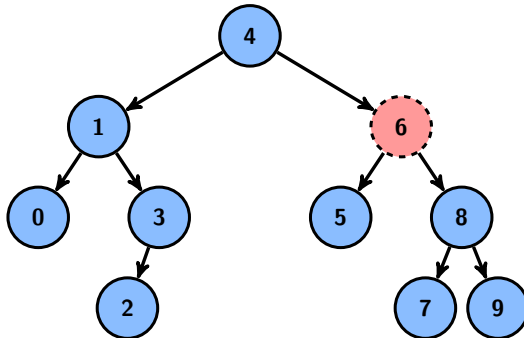
Maximul din sub-arborele stâng!

Ștergerea unui element – Cazul III



Ce valoare este mai mică decât toate din sub-arborele drept?

Ștergerea unui element – Cazul III

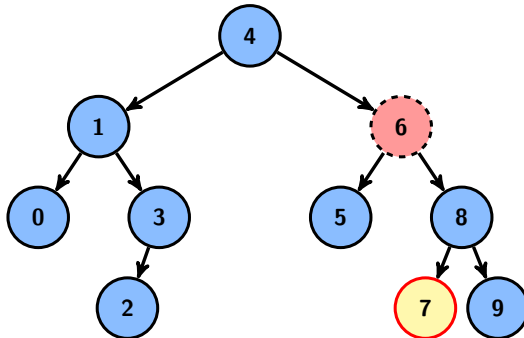


Ce valoare este mai mică decât toate din sub-arborele drept?



Minimul din sub-arborele drept!

Ștergerea unui element – Cazul III



Ce valoare este mai mică decât toate din sub-arborele drept?



Minimul din sub-arborele drept!

Ștergerea unui element

```
1: function DELETE(root, x)
2:   if root == NULL then
3:     print(Nodul nu a fost gasit)
4:     return root
5:   end if
6:   if root → value > value then
7:     root → left ← delete(root → left, x)
8:   else if root → value < value then
9:     root → right ← delete(root → right, x)
10:  else
11:    if root → left ≠ NULL and root → right ≠ NULL then
12:      tmp ← findMinimum(root → right)
13:      root → value ← tmp → value
14:      root → right ← delete(root → right, tmp → value)
15:    else
16:      tmp ← root
17:      if root → left ≠ NULL then
18:        root ← root → left
19:      else
20:        root ← root → right
21:      end if
22:      free(tmp)
23:    end if
24:  end if
25:  return root
26: end function
```

▷ Am gasit nodul cautat

▷ Nodul are 2 fii - cazul III

▷ Nodul are un fiu sau este frunza - cazurile I si II

Ștergerea unui element (I)

```
38 TBinaryTree delete(TBinaryTree root, T value) {
39     if (root == NULL)
40         return root;
41     // Caut nodul
42     if (root->value > value)
43         root->left = delete(root->left, value);
44     else if (root->value < value)
45         root->right = delete(root->right, value);
46     else {
47         // Am găsit nodul căutat
48         if (root->left != NULL && root->right != NULL) {
49             // Nodul are 2 fii
50             TBinaryTree temp = minimum(root->right);
51             root->value = temp->value;
52             root->right = delete(root->right, temp->value);
53         }
```

Ștergerea unui element (II)

```
54     else {  
55         // Nodul are un fiu sau este frunză  
56         TBinaryTree temp = root;  
57         if (root->left != NULL)  
58             root = root->left;  
59         else  
60             root = root->right;  
61         free(temp);  
62     }  
63 }  
64 return root;  
65 }
```

Dealocarea memoriei

```
66 TBinaryTree freeTree(TBinaryTree root) {  
67     if (root == NULL)  
68         return NULL;  
69     root->left = freeTree(root->left);  
70     root->right = freeTree(root->right);  
71     free(root);  
72     return NULL;  
73 }
```

Eficiența operației de inserare

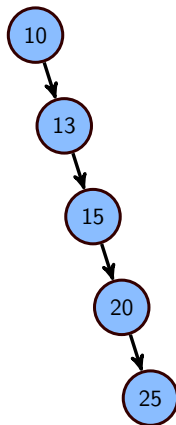


Cum o să arate arborele binar de căutare rezultat în urma inserării valorilor: 10, 13, 15, 20, 25?

Eficiența operației de inserare



Cum o să arate arborele binar de căutare rezultat în urma inserării valorilor: 10, 13, 15, 20, 25?



Eficiența operației de ștergere

- Eliminarea unui element (nod) trebuie să se facă astfel încât să se respecte condiția de **arbore binar de căutare** după eliminare.
- Există mai multe cazuri care trebuie tratate:
 - ❶ Eliminare frunză
 - ❷ Eliminare nod de ordinul 1 (are un singur subarbore)
 - ❸ Eliminare nod de ordinul 2 (are ambii subarbori)

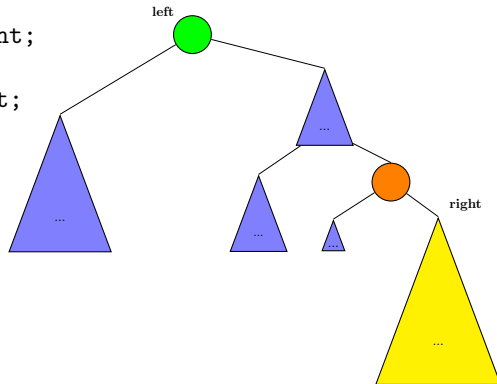
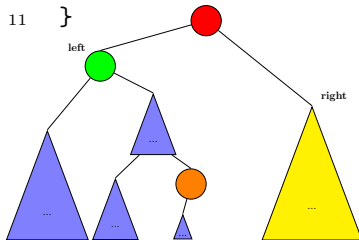
Important

După aplicarea operației de eliminare a unui nod poate rezulta un arbore dezechilibrat.

Pentru nodurile de ordinul 2 putem aplica eliminare cu pivot care nu dezechilibrează arborele.

Eliminare element – Iterativ

```
1  TBinaryTree BuildTree(TBinaryTree left, TBinaryTree right) {
2      TBinaryTree temp=left, parent;
3      if (left == NULL) return right;
4      if (right == NULL) return left;
5      while (temp != NULL) {
6          parent = temp;
7          temp = temp->right;
8      }
9      parent->right = right;
10     return left;
11 }
```



Eliminare element – Iterativ (I)

```
1  TBinaryTree delete_iterative(TBinaryTree root, T value) {
2      TBinaryTree parent=NULL, iter=root, x;
3      if (iter == NULL) {
4          printf("arbore vid\n");
5          return NULL;
6      }
7      while (iter != NULL) {
8          if (iter->value == value)
9              break;
10         parent = iter;
11         if (value < iter->value)
12             iter = iter->left;
13         else iter = iter->right;
14     }
15     if (iter == NULL) {
16         printf("cheia nu exista\n");
17         return NULL;
18     }
```

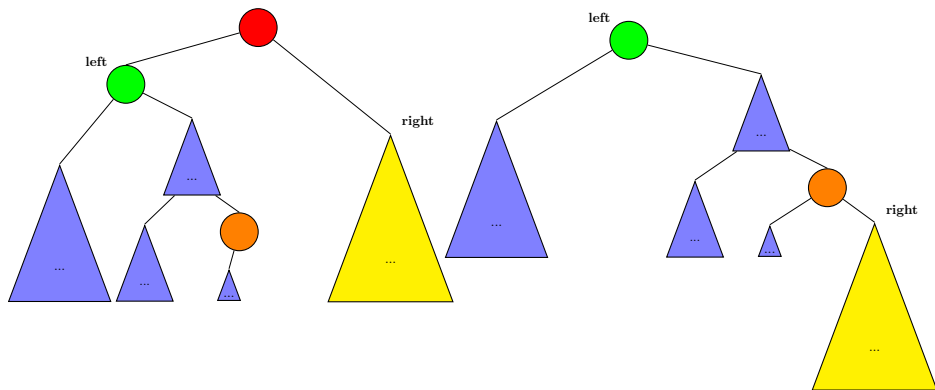
Eliminare element – Iterativ (II)

```
19     if(parent == NULL) { /* cheia gasită este în rădăcină */
20         x = BuildTree(iter->left, iter->right);
21         free(iter);
22         return x;
23     }
24     if(iter->left == NULL && iter->right == NULL) /*cheia este c
25         x = NULL;
26     else
27         x = BuildTree(iter->left, iter->right);
28     if (parent->right->value == value)
29         parent->right = x;
30     else
31         parent->left = x;
32     free(iter);
33     return root;
34 }
```

Eliminare element – Iterativ (exemplu)

Important

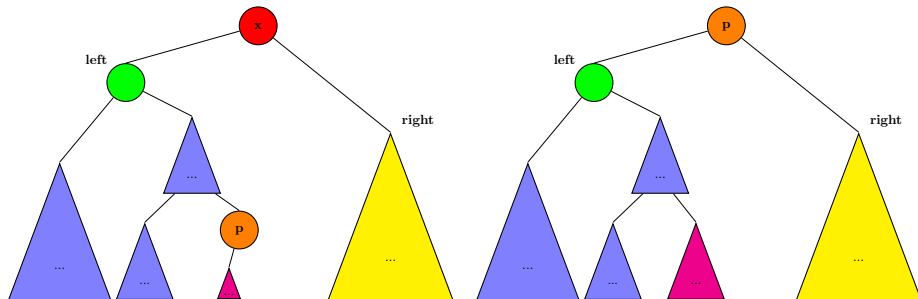
Pentru cazul 2 este posibil să apară o dezechilibrare!



Eliminare element cu pivot

Eliminare nod de ordin 2

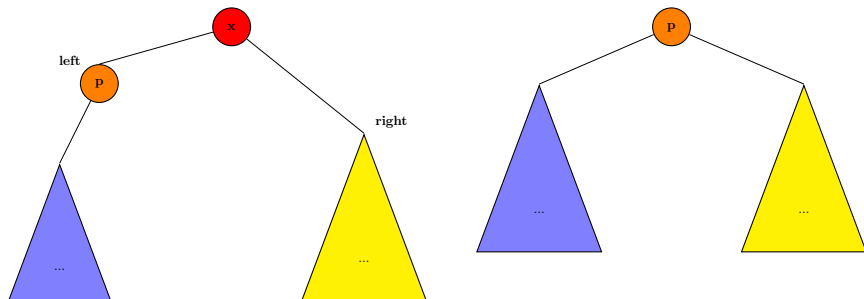
Eliminare cu pivot stânga (cazul general)



Eliminare element cu pivot

Eliminare nod de ordin 2

Eliminare cu pivot stânga (cazul particular)



Eliminare element cu pivot - Funcție construcție

```
1  TBinaryTree BuildTreePivot(TBinaryTree left, TBinaryTree right) {
2      TBinaryTree iter = left, parent;
3      if (left == NULL)
4          return right;
5      if (right == NULL)
6          return left;
7      if (left->right == NULL) {
8          left->right = right;
9          return left;
10     }
11     /* aici left și pivot sunt aceeași */
12     while (iter->right != NULL) { /* cu iter caut pivot */
13         parent = iter;
14         iter = iter->right;
15     } /* la ieșirea din while am găsit pivot referit de iter */
16     parent->right = iter->left;
17     iter->left = left; iter->right = right;
18     return iter;
19 }
```

Observații

Eliminare element cu pivot

Vom folosi funcția auxiliară (`BuildTreePivot`) în funcția `delete_iterative` pentru a elimina nod fără dezechilibrare.

Eliminare element recursiv

În funcția recursivă de eliminare a unui element a fost determinat minimul din subarborele drept. Puteam determina și maximul din subarborele stâng.

Indiferent de situație, nodul respectiv (minim din dreapta / maxim din stânga) va avea cel mult un copil!

Comparație

Funcția recursivă ajunge să încarce stiva procesului care o rulează, deoarece fiecare apel recursiv se va adăuga pe stivă.

Aplicație pentru Arbori Binari de Căutare

De obicei prelucrăm structuri mai complicate decât întregi sau caractere

Varianta I

```
1  typedef struct client {
2      int id_client;
3      char *nume;
4      char *adresa;
5      char *nr_tel;
6  } Item;
7
8  typedef struct node *TLink;
9
10 typedef struct node {
11     Item elem;
12     TLink lt, rt;
13 } TreeNode;
```

Aplicație pentru Arbori Binari de Căutare

De obicei prelucrăm structuri mai complicate decât întregi sau caractere

Varianta I

```
1  typedef struct client {
2      int id_client; // cheia folosită pentru căutare
3      char *nume;
4      char *adresa;
5      char *nr_tel;
6  } Item;
7
8  typedef struct node *TLink;
9
10 typedef struct node {
11     Item elem;
12     TLink lt, rt;
13 } TreeNode;
```

Varianta I

```
1  typedef struct client {
2      int id_client; // cheia folosită pentru căutare
3      char *nume;
4      char *adresa;
5      char *nr_tel;
6  } Item;
7  #define Key(pers) pers.id_client
8  TLink BTSearch(TLink t, int cheie) {
9      if(t == NULL)
10         return NULL;
11      if (Key(t->elem) == cheie)
12         return t;
13      if(cheie < Key(t->elem))
14         return BTSearch(t->lt,cheie);
15      return BTSearch(t->rt,cheie);
16  }
```

Varianta II

```
1  typedef struct client {
2      char *nume; // cheia folosită pentru căutare
3      char *adresa;
4      char *nr_tel;
5  } Item;
6  #define Key(pers) pers.nume
7  #define Equal(key1, key2) strcmp(key1, key2) == 0
8  #define Less(key1, key2) strcmp(key1, key2) < 0
9
10 TLink BTSearch(TLink t, char *cheie) {
11     if (t == NULL)
12         return NULL;
13     if (Equal(cheie, Key(t->elem))
14         return t;
15     if (Less(cheie, Key(t->elem)))
16         return BTSearch(t->lt, cheie);
17     return BTSearch(t->rt, cheie);
18 }
```

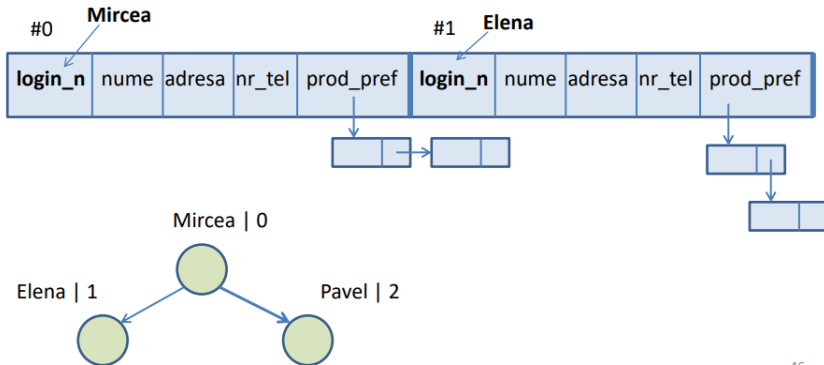
Căutare bazată pe index

- Pentru multe aplicații dorim o structură de date care să ne ajute la căutare dar care să nu ne oblige să mutăm efectiv elementele.
- Utilă și în cazul elementelor de dimensiune mare care ar trebui alocate.

```
1  typedef struct client {
2      char login_n[10];
3      char *nume;
4      char *adresa;
5      char *nr_tel;
6      TList produse_pref
7  } Item;
```

Căutare bazată pe index

Elementele care conțin informație sunt organizate într-o structură separată, chiar statică, și folosim indexul acestora ca elemente din nodurile arborelui.



46

Vă mulțumesc pentru atenție!

