Stive. Cozi. Liste înlănțuite

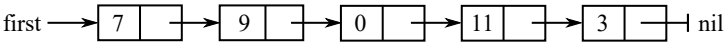
Universitatea "Transilvania" din Brașov

3 martie 2022

Liste simplu înlănțuite

Listă simplu înlănțuită:

- fiecare element *node* este o structură cu două câmpuri
  - *key* = cheia
  - *next* = adresa următorului element
- accesul se realizează prin primul element *first*



Observații:

- elementele nu sunt stocate în zone de memorie adiacente

Observații:

- elementele nu sunt stocate în zone de memorie adiacente
- pentru fiecare nod nou se alocă memorie

Observații:

- elementele nu sunt stocate în zone de memorie adiacente
- pentru fiecare nod nou se alocă memorie
- listele permit inserarea respectiv ștergerea în orice poziție a listei.

Observații:

- elementele nu sunt stocate în zone de memorie adiacente
- pentru fiecare nod nou se alocă memorie
- listele permit inserarea respectiv ștergerea în orice poziție a listei.
- listele suportă operația de căutare a unei chei.

Observații:

- elementele nu sunt stocate în zone de memorie adiacente
- pentru fiecare nod nou se alocă memorie
- listele permit inserarea respectiv ștergerea în orice poziție a listei.
- listele suportă operația de căutare a unei chei.
- accesul la elemente - prin capul listei, reprezentând primul element (uneori și prin ultimul element)

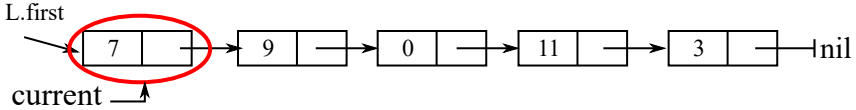
Algoritm: Căutarea unui element cu cheia *value* într-o listă L

```
Functie FIND(L, value)
    current ← L.first
    cat_timp current ≠ nil și current.key ≠ value executa
        current ← current.next
    sfarsit_cat_timp
    return current
end
```

Algoritm: Căutarea unui element cu cheia *value* într-o listă L

Functie *FIND(L, value)*
    current ← *L.first*
    cat_timp *current* ≠ *nil* și *current.key* ≠ *value* **executa**
        current ← *current.next*
    sfarsit_cat_timp
    return *current*
end

Exemplu: caut în lista de mai jos cheia 11. Se pornește din capul listei cu *current* = *L.first*

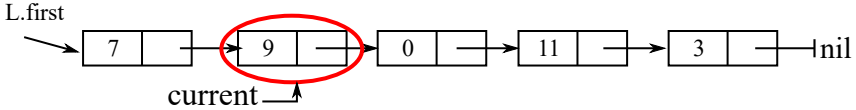


current.key ≠ 11 ⇒ *current* ← *current.next*

Algoritm: Căutarea unui element cu cheia *value* într-o listă L

Functie *FIND(L, value)*
    current $\leftarrow$ *L.first*
    cat_timp *current* $\neq$ *nil și current.key* $\neq$ *value* **executa**
        current $\leftarrow$ *current.next*
    sfarsit_cat_timp
    return *current*
end

Exemplu: caut în lista de mai jos cheia 11. Se pornește din capul listei cu *current* = *L.first*

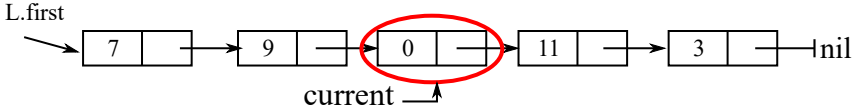


current.key $\neq$ 11 $\Rightarrow$ *current* $\leftarrow$ *current.next*

Algoritm: Căutarea unui element cu cheia *value* într-o listă L

Functie *FIND(L, value)*
    current ← *L.first*
    cat_timp *current* ≠ *nil* și *current.key* ≠ *value* **executa**
        current ← *current.next*
    sfarsit_cat_timp
    return *current*
end

Exemplu: caut în lista de mai jos cheia 11. Se pornește din capul listei cu *current* = *L.first*

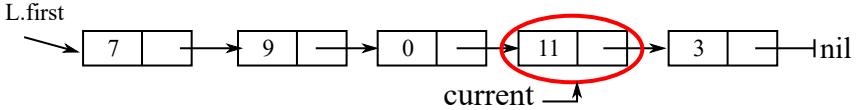


current.key ≠ 11 ⇒ *current* ← *current.next*

Algoritm: Căutarea unui element cu cheia *value* într-o listă L

Functie *FIND(L, value)*
    current $\leftarrow$ *L.first*
    cat_timp *current* $\neq$ *nil și current.key* $\neq$ *value* **executa**
        current $\leftarrow$ *current.next*
    sfarsit_cat_timp
    return *current*
end

Exemplu: caut în lista de mai jos cheia 11. Se pornește din capul listei cu *current* = *L.first*



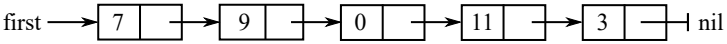
currnet.key = 11 $\Rightarrow$ return *current*

Algoritm: Căutarea unui element cu cheia *value* într-o listă L

```
Functie FIND(L, value)
    current ← L.first
    cat_timp current ≠ nil și current.key ≠ value executa
        current ← current.next
    sfarsit_cat_timp
    return current
end
```

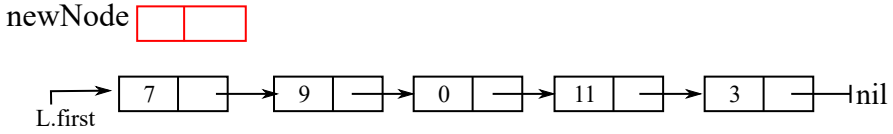
Complexitate: În cel mai defavorabil caz, atunci când *value* nu se găsește în listă, complexitatea este $\Theta(n)$, unde n = lungimea listei.

Adăugarea unui element nou - **Exemplu:** Se adaugă elementul cu cheia 21 în lista L



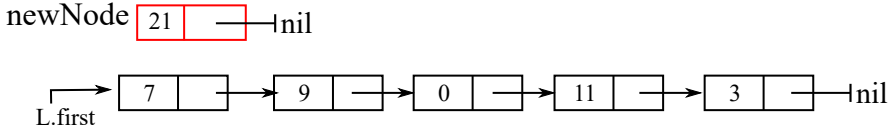
Adăugarea unui element nou - **Exemplu:** Se adaugă elementul cu cheia 21 în lista *L*

1. Se alocă memorie pentru nodul nou: *nodNou* = *newnode*



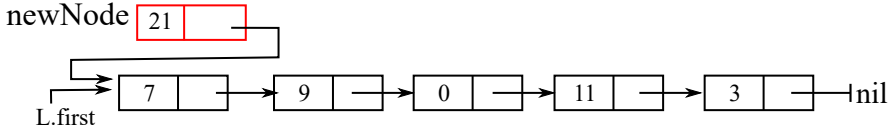
Adăugarea unui element nou - **Exemplu:** Se adaugă elementul cu cheia 21 în lista *L*

2. Se inițializează câmpurile lui *newNode*



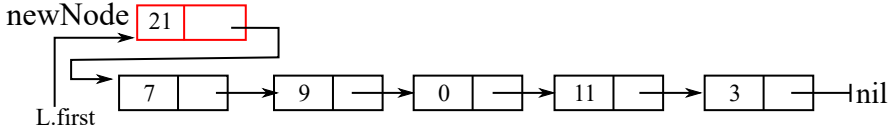
Adăugarea unui element nou - **Exemplu:** Se adaugă elementul cu cheia 21 în lista *L*

3. Se leagă *newNode* de capul listei



Adăugarea unui element nou - **Exemplu:** Se adaugă elementul cu cheia 21 în lista L

4. Se reinițializează capul listei cu noul element.



Liste simplu înlănțuite - Operații

Algoritm: push_front

Intrare: Lista simplu înlănțuită L la care se adauga un element cu cheia *value*

aloca memorie pentru nodul *newNode*

$newNode.key \leftarrow value$

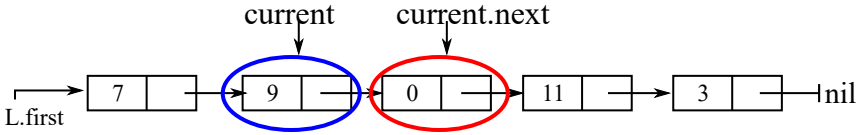
$newNode.next \leftarrow L.first$

$L.first \leftarrow newNode$

Complexitate: $\Theta(1)$

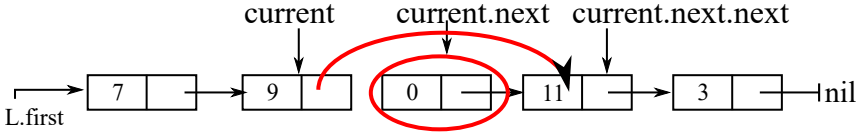
Ștergerea unuei valori din listă - **Exemplul 1:** ștergem nodul cu cheia 0 din lista.

1. Avansăm în listă până când nodul curent are ca nod următor elementul cu cheia 0.



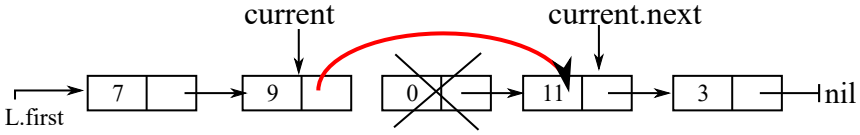
Ștergerea unui element din listă - **Exemplul 1:** ștergem nodul cu cheia 0 din lista

2. Legăm nodul *current* de nodul care îi urmează celui, pe care dorim să îl ștergem, adică *current.next.next*.



Ștergerea unuei valori din listă - **Exemplul 1:** ștergem nodul cu cheia 0 din lista

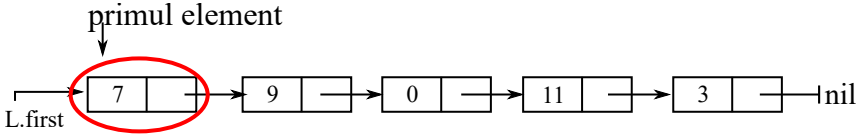
3. Eliberăm memoroia pentru nodul cu cheia 0.



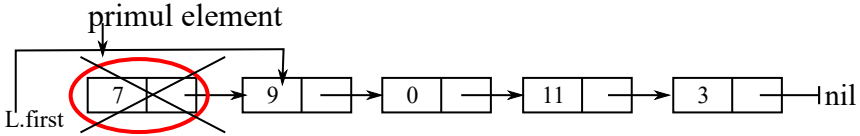
Ștergerea unui element din listă - **Exemplul 1:** ștergem nodul *node* cu cheia 0 din lista

4. Obținem:

Ștergerea unui element din listă - **Exemplul 2:** ștergem nodul cu cheia 7 din listă. Este primul element.



Ștergerea unui element din listă - **Exemplul 2:** ștergem nodul *node* cu cheia 7 din lista



Se modifică doar capul listei și apoi se șterge elementul!

Liste simplu înlănțuite - Operații

Algoritm: REMOVE

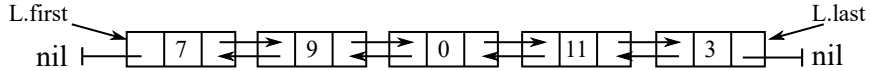
Intrare: Lista simplu înlănțuită L din care se șterge elementul *value*

daca $L.head = nil$ **atunci**
    return
sfarsit_daca
daca $L.head.key = value$ **atunci**
    $node \leftarrow L.head$
    $L.head \leftarrow L.head.next$
    elibereaza memoria pentru *node*
    return
sfarsit_daca
$current \leftarrow L.head$
cat_timp $current.next \neq nil$ *si* $current.next.key \neq value$ **executa**
    $current \leftarrow current.next$
sfarsit_cat_timp
$node = current.next$
$current.next \leftarrow current.next.next$
elibereaza memoria pentru *node*

Liste dublu înlănțuite

Listă simplu înlănțuită:

- fiecare element *node* este o structură cu trei câmpuri
  - *key* = cheia
  - *prev* = adresa elementului precesent
  - *next* = adresa următorului element
- accesul se realizează prin primul element *first* și prin ultimul element *last*

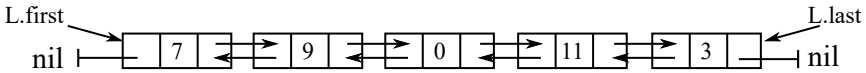


Liste dublu înlănțuite - Operații

Algoritm: Căutarea unei chei *key* în lista *L*

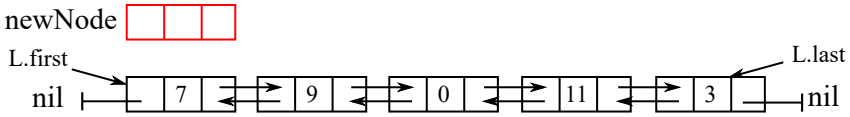
Functie *SEARCH(key)*
    current $\leftarrow$ *L.first*
    cat_timp *current* $\neq$ *NULL* și *current.info* $\neq$ *key* **executa**
        current $\leftarrow$ *current.next*
    sfarsit_cat_timp
    return *current*
end

Adăugarea unui nod nou - **Exemplu**: Adăugarea unui nod cu cheia 21 în lista L:



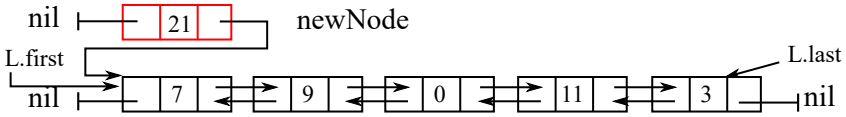
Adăugarea unui nod nou - **Exemplu**: Adăugarea unui nod cu cheia 21 în lista *L*:

1. Se alocă memorie pentru noul nod: *newNode* = *newnode*



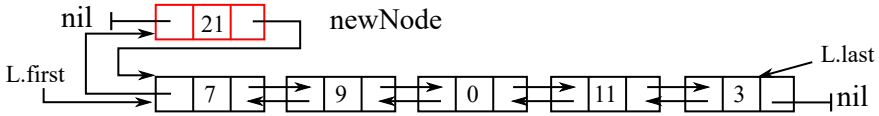
Adăugarea unui nod nou - **Exemplu**: Adăugarea unui nod cu cheia 21 în lista L:

2. Se completează câmpurile noului nod



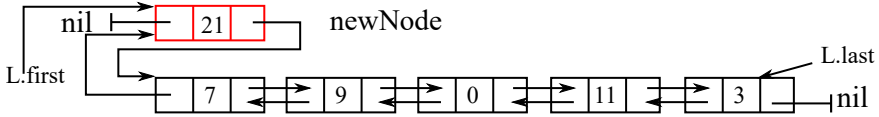
Adăugarea unui nod nou - **Exemplu**: Adăugarea unui nod cu cheia 21 în lista L:

3. Se leagă nodul din capul listei, de noul nod



Adăugarea unui nod nou - **Exemplu**: Adăugarea unui nod cu cheia 21 în lista L:

4. Se reinițializează capul listei cu noul nod

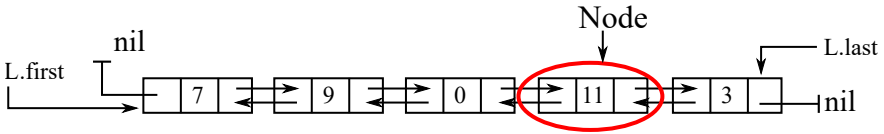


Algoritm: Adăugarea unui element la începutul listei L

Functie *push_front(value)*
    aloca memorie pentru *newNode*
    $newNode.next \leftarrow L.first$
    $newNode.prev \leftarrow$ nil
    daca $L.first \neq nil$ **atunci**
        $L.first.prev \leftarrow newNode$
    sfarsit_daca
    $L.first \leftarrow newNode$
end

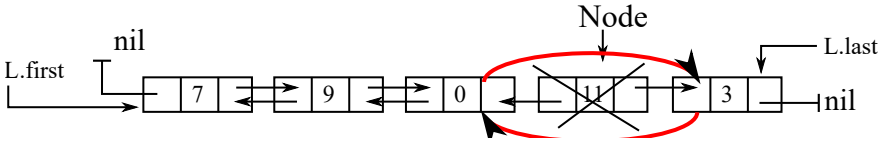
Ștergerea unui element *node* **în listă**: elementul *node* a fost întâi gasit prin funcția SEARCH.

Exemplu: Se șterge *Node* cu cheia 11 din listă:



Ștergerea unui element x în listă: elementul *node* a fost întâi gasit prin funcția SEARCH.

Exemplu: Se șterge *Node* cu cheia 11 din lista:



Se leagă *Node.prev* de *Node.next* și *Node.next* de *Node.prev*. Apoi se șterge *Node*.

Algoritm: Ștergerea unui element *node* din lista *L*

Functie *ERASE(Node)*

```
daca Node.prev ≠ nil atunci
    Node.prev.next ← Node.next
sfarsit_daca
altfel
    L.first = Node.next
sfarsit_daca
daca Node.next ≠ nil atunci
    Node.next.prev ← Node.prev
sfarsit_daca
```

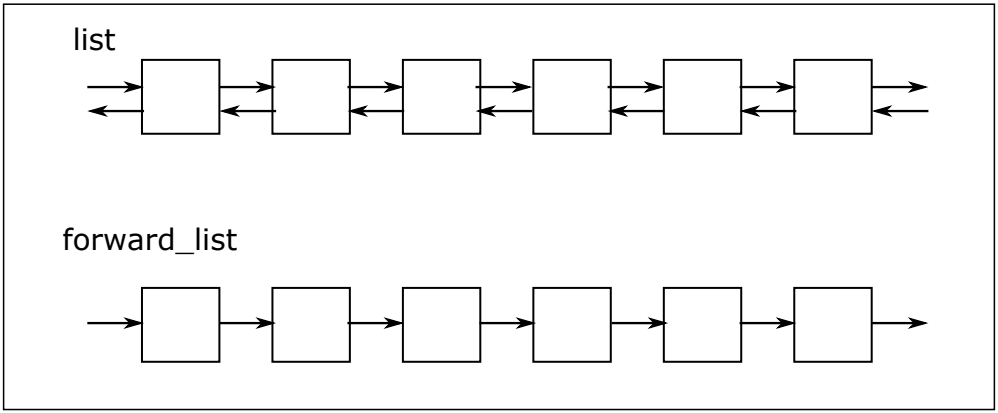
end

Complexitate: Ștergerea efectiva: $\Theta(1)$. Dacă elementul *Node* trebuie întâi căutat $\Rightarrow$ căutarea este $\Theta(n)$

Utilizarea listelor

- atunci când sunt necesare operații fecvente de adăugare / ștergere de elemente din listă
- nu permit acces prin poziție
- dacă operația principală e de acces prin poziție și adăugările sunt preponderent la final $\Rightarrow$ se preferă vectori.

Liste în STL



List:

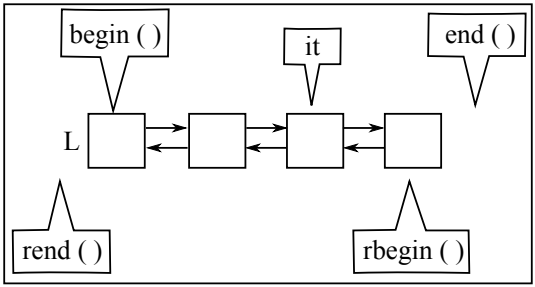
- elementele nu sunt memorate în zone de memorie alăturate
- accesul elementelor în timp liniar, cu iteratori.
- inserție și ștergere de elemente în timp constant.

Liste în STL

```cpp
#include<iostream>
#include<vector>
#include<list>


int main()
{

    std::list<int> lista;

    std::list<int> lista2(4, 50); //contine 4 elem egale cu 50

    std::list<int> lista3(lista2); //copie a listei 2

    std::list<int> lista4 = lista3; //copie a listei 3

    std::list<int> lista5 = { 1, 2, 3, 4, 5 }; // initializare

    std::vector<int> v = { 2, 4, 6, 8, 10 };
    std::list<int> lista6(v.begin(), v.end()); //initializare pe baza unui vector;

    return 0;
}
```

Observații

- Accesul la elemente NU se poate face prin indicele poziției, ca la vector!
- Pentru a parcurge o lista se folosesc ITERATORI



Liste în STL - parcurgere

```cpp
#include<iostream>
#include<list>


int main()
{
    std::list<int> lista;
    int num = 5, elem;
    for (int index = 0; index < num; index++)
    {
        std::cin >> elem;
        lista.push_back(elem);
        lista.push_front(elem);
    }
    std::list<int>::iterator it = lista.begin();
    for (; it != lista.end(); it++)
    {
        std::cout << *it << " ";
    }
    std::cout << std::endl;
    //sau
    for (int element : lista)
    {
        std::cout << element << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Liste în STL - Funcția **Advance**

advance - are ca param un iterator și nr de poziții cu care se avansează:

```
int nr = 5;
auto it = lista.begin();
if (nr < lista.size() && nr > 0)
{
    advance(it, nr);
    std::cout << *it;
}
```

Funcții membre - List

de acces	dimensiune	modificare
cu iteratori front() / back()	size( ) empty( )	push_back(val) / pop_back( ) push_front(val) / pop_front( ) insert / remove clear()

Liste în STL - inserție / ștergere de elemente

lista.insert(it, value)

- are ca param un iterator și o valoare.
- inserează valoarea înainte de elementul indicat prin iterator

lista.remove(value)

- are ca parametru o valoare
- elimină toate aoarițiile alorii din listă

lista.erase(it) sau **lista.erase(it1, it2)**

- are ca parametri unul sau doi iteratori
- elimină din listă elementul indicat de iterator (dacă are un singur parametru), elementele începând de la primul iterator pâna la al doilea (exclusiv pe cel indicat de al doilea parametru)

Liste în STL - Exercițiu

```cpp
#include<iostream>
#include<list>

int main()
{
    std::list<int> lista;
    std::list<int>::iterator it;
    for (int index = 1; index < 6; index++)
        lista.push_back(index);
    it = lista.begin(); it++;
    lista.insert(it, 10);
    lista.insert(it, 2, 20);
    it = lista.erase(it);
    //it se va plasa pe elementul care ii urmeaza
    //elementului care a fost sters
    //https://www.cplusplus.com/reference/list/list/erase/
    --it;
    for (it; it != lista.end(); it++)
    {
        std::cout << *it<<" ";
    }
    return 0;
}
```

Liste în STL - Căutarea unui element

Căutarea unui element: - cu funcția *std* :: *find* - returnează un iterator.

```
#include<iostream>
#include<list>

int main()
{
    std::list<std::string> listOfStrs = {"is", "of", "the", "Hi", "Hello", "from" };
    std::list<std::string>::iterator it;

    // Fetch the iterator of element with value 'the'
    it = std::find(listOfStrs.begin(), listOfStrs.end(), "the");

    // Check if iterator points to end or not
    if (it != listOfStrs.end())//it means element exists in list
           std::cout << "'the' exists in list " << std::endl;
    else // It points to end, it means element does not exists in list
        std::cout << "'the' does not exists in list" << std::endl;
}
```

Liste în STL - Căutarea unui element

Există și alți algoritmi de căutare implementați in STL - vezi documentație online.