Structuri de Date și Algoritmi
Liste simplu înlănțuite

Mihai Nan

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

Anul Universitar 2022–2023

Conținutul cursului

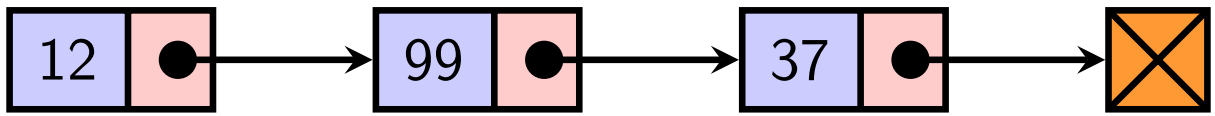
1. **Definiția structurii de date**
2. **Liste versus Vectori**
3. **Operații elementare**
   - Reprezentarea structurii de date
   - Accesări în liste
   - Inițializarea listei
   - Parcurgerea listei
   - Adăugarea la începutul listei
   - Adăugarea la finalul listei
   - Adăugarea în interiorul listei
   - Ștergerea de la începutul listei
   - Ștergerea de la finalul listei
4. **Recapitulare recursivitate**

Liste înlănțuite – Definiție

Definiția unui liste

Lista reprezintă o structură de date care modelează o colecție, alocată dinamic, de elemente care sunt reținute dispersat în memorie și sunt legate între ele prin intermediul pointerilor.

- Este o structură de date liniară și alocată dinamic.
- Lista este cea mai comună structură de date liniară, fiind utilizată în foarte multe aplicații.
- Este o structură de date propusă pentru a reduce dezavantajele pe care le prezintă vectorii pentru **inserarea unui element** sau **ștergerea unui element**.

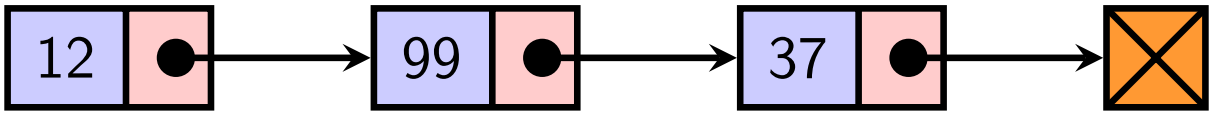


- Pentru această structură de date avem acces secvențial la elemente.

Liste simplu înlănțuite – Definiție

Definiția unei liste simplu înlănțuite

Lista simplu înlănțuită este un caz particular de listă caracterizată printr-o relație de ordine definită explicit prin intermediul unui câmp care pointează către elementul următor sau către `NULL`.



Observație

Structura de date este **recursivă**, deoarece câmpul informației de legătură este de tip pointer la tipul nodului. Astfel, listele simplu înlănțuite sunt structuri obținute prin concatenarea de noduri de tipul descris în declararea structurii prin intermediul câmpului de legătură.

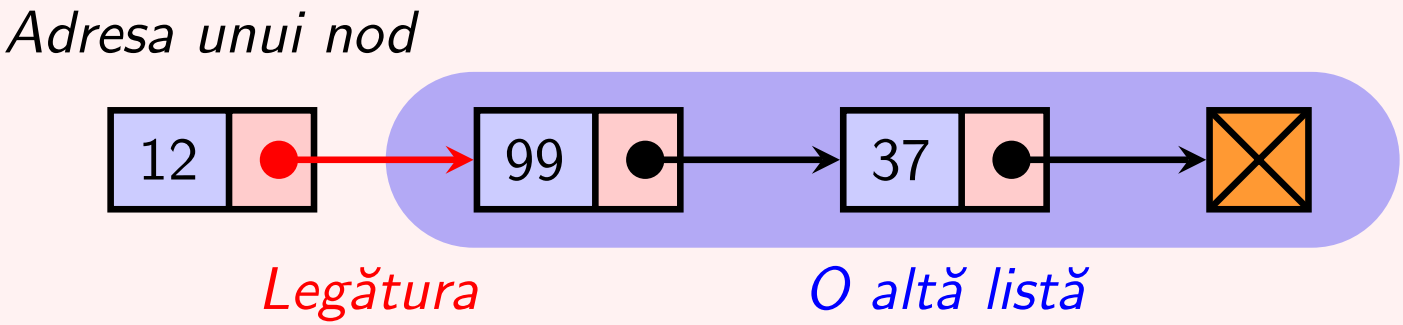
Liste simplu înlănțuite – Definiție recursivă

Definiția recursivă

O listă este:

1. O listă vidă (**cazul de bază**)

2. Adresa unui nod care conține o valoare și o legătură la o altă listă.



Liste simplu înlănțuite – Terminologie

- Elementele dintr-o listă se numesc **celule** sau **noduri**.
- Pentru fiecare element, vom reține o valoare utilă și o legătură către elementul următor din listă.
- Pentru implementarea acestei structuri vom utiliza **alocarea dinamică** a memoriei la nivelul unui nod. Cu alte cuvinte, atunci când avem nevoie de un nod suplimentar, alocăm inițial memoria pentru el.
- Din moment ce folosim alocare dinamică, listele vor fi reținute pe heap.
- O variabilă de tip listă poate avea ca valoare:
  1. `NULL` pentru a reprezenta lista vidă;
  2. adresa primei celule (dacă lista este nevidă).

Liste versus Vectori

Operația	Liste	Vectori	Vectori alocați dinamic
1) Indexarea	$O(n)$	$O(1)$	$O(1)$
2) Inserarea / ștergerea (început)	$O(1)$	$O(n)$ [1]	$O(n)$
3) Inserarea la final	$O(n)$ sau $O(1)$ [2]	$O(1)$ [1]	$O(1)$ [3] sau $O(n)$
4) Ștergerea de la final	$O(n)$ sau $O(1)$ [2]	$O(1)$	$O(1)$
5) Inserarea în mijloc	$O(n)$	$O(n)$ [1]	$O(n)$
6) Ștergerea din mijloc	$O(n)$	$O(n)$	$O(n)$

[1] Dacă mai există memorie să putem realiza inserarea noului element.
[2] Dacă reținem un pointer la ultimul element din listă.
[3] Dacă mai există memorie alocată.
[4] Dacă nu există memorie alocată și trebuie realocată.

Liste simplu înlănțuite – Reprezentarea

- Reprezentarea structurii de date cu care vom lucra este următoarea:

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

unde `T` este un tip formal ce poate fi înlocuit, spre exemplu, cu un tip primitiv din limbajul **C** (`int`, `char` etc.) sau cu un tip definit de către utilizator.

Liste simplu înlănțuite – Reprezentarea

- Reprezentarea structurii de date cu care vom lucra este următoarea:

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

  unde `T` este un tip formal ce poate fi înlocuit, spre exemplu, cu un tip primitiv din limbajul **C** (`int`, `char` etc.) sau cu un tip definit de către utilizator.
  Exemplu: `typedef int T;`

- Care este relația dintre `struct node` și `Node`?

Liste simplu înlănțuite – Reprezentarea

- Reprezentarea structurii de date cu care vom lucra este următoarea:

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

unde `T` este un tip formal ce poate fi înlocuit, spre exemplu, cu un tip primitiv din limbajul **C** (`int`, `char` etc.) sau cu un tip definit de către utilizator.
Exemplu: `typedef int T;`

- Care este relația dintre `struct node` și `Node`?
`Node` ≡ `struct node`
- Care este relația dintre `struct node` și `TList`?

Liste simplu înlănțuite – Reprezentarea

- Reprezentarea structurii de date cu care vom lucra este următoarea:

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

unde `T` este un tip formal ce poate fi înlocuit, spre exemplu, cu un tip primitiv din limbajul **C** (`int`, `char` etc.) sau cu un tip definit de către utilizator.
Exemplu: `typedef int T;`

- Care este relația dintre `struct node` și `Node`?
  `Node` ≡ `struct node`
- Care este relația dintre `struct node` și `TList`?
  `TList` ≡ `struct node *` ≡ `Node*`

Liste simplu înlănțuite – Reprezentarea

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

- Cum putem reprezenta lista vidă?

Liste simplu înlănțuite – Reprezentarea

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

- Cum putem reprezenta lista vidă?

```
TList head = NULL;
```

- Cum putem aloca memorie pentru o celulă?

Liste simplu înlănțuite – Reprezentarea

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

- Cum putem reprezenta lista vidă?

```
TList head = NULL;
```

- Cum putem aloca memorie pentru o celulă?

```
TList head = (TList) malloc(sizeof(struct node));
```

Liste simplu înlănțuite – Reprezentarea

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

- Cum putem reprezenta lista vidă?

```
TList head = NULL;
```

- Cum putem aloca memorie pentru o celulă?

```
TList head = (TList) malloc(sizeof(struct node));
```

sau

```
TList head = (TList) calloc(1, sizeof(struct node));
```

Liste simplu înlănțuite – Reprezentarea

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

- Cum putem reprezenta lista vidă?

```
TList head = NULL;
```

- Cum putem aloca memorie pentru o celulă?

```
TList head = (TList) malloc(sizeof(struct node));
```

sau

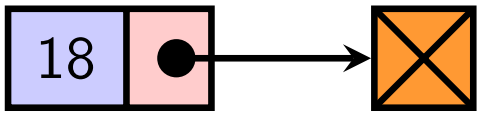
```
TList head = (TList) calloc(1, sizeof(struct node));
```

- Există diferențe între cele două variante?

Liste simplu înlănțuite – Reprezentarea

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

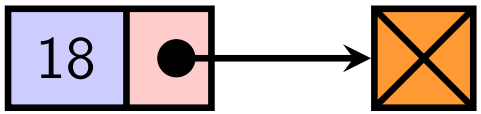
- Cum putem reprezenta următoarea listă?



Liste simplu înlănțuite – Reprezentarea

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

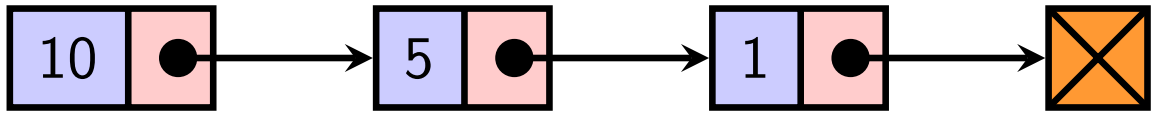
- Cum putem reprezenta următoarea listă?




```
TList head = (TList) malloc(sizeof(struct node));
head->value = 18;
head->next = NULL;
```

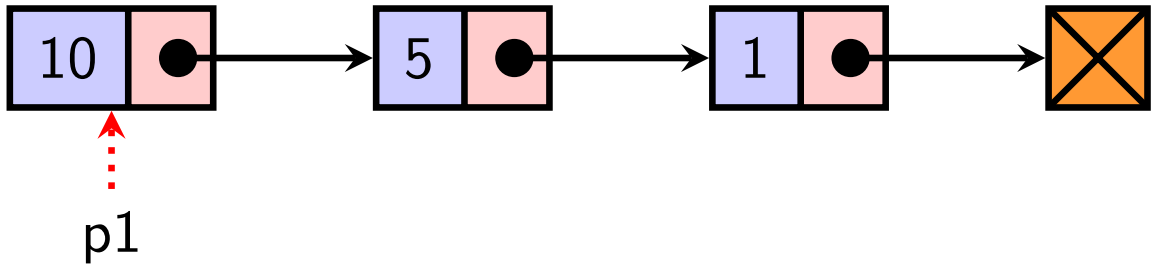
Liste simplu înlănțuite – Reprezentarea

- Cum putem reprezenta următoarea listă?



Liste simplu înlănțuite – Reprezentarea

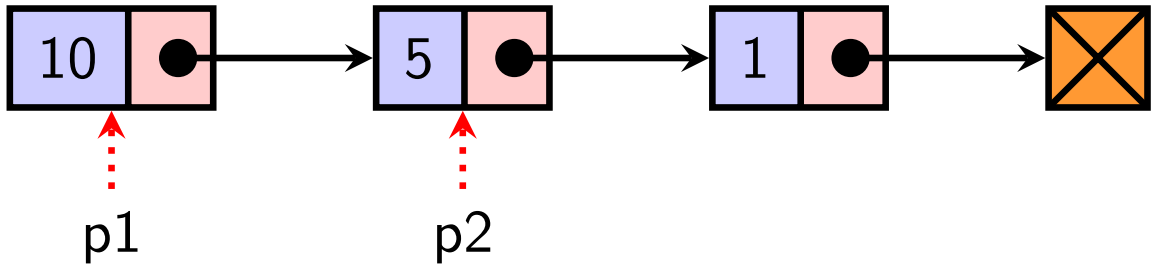
- Cum putem reprezenta următoarea listă?




```
TList p1 = (TList) malloc(sizeof(struct node));
p1->value = 10;
```

Liste simplu înlănțuite – Reprezentarea

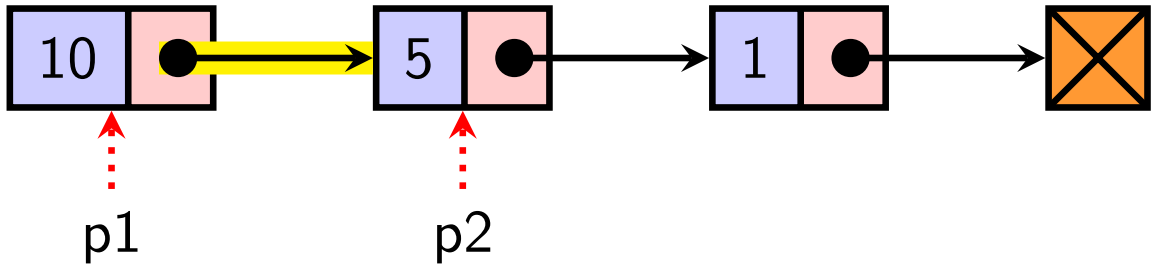
- Cum putem reprezenta următoarea listă?




```
TList p1 = (TList) malloc(sizeof(struct node));
p1->value = 10;
TList p2 = (TList) malloc(sizeof(struct node));
p2->value = 5;
```

Liste simplu înlănțuite – Reprezentarea

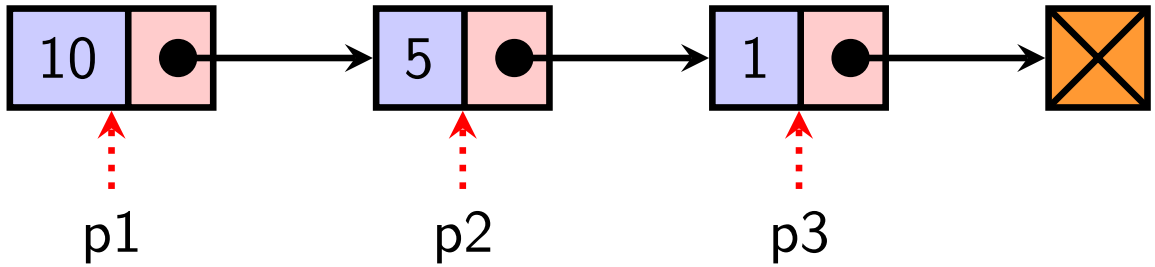
- Cum putem reprezenta următoarea listă?




```
TList p1 = (TList) malloc(sizeof(struct node));
p1->value = 10;
TList p2 = (TList) malloc(sizeof(struct node));
p2->value = 5;
p1->next = p2;
```

Liste simplu înlănțuite – Reprezentarea

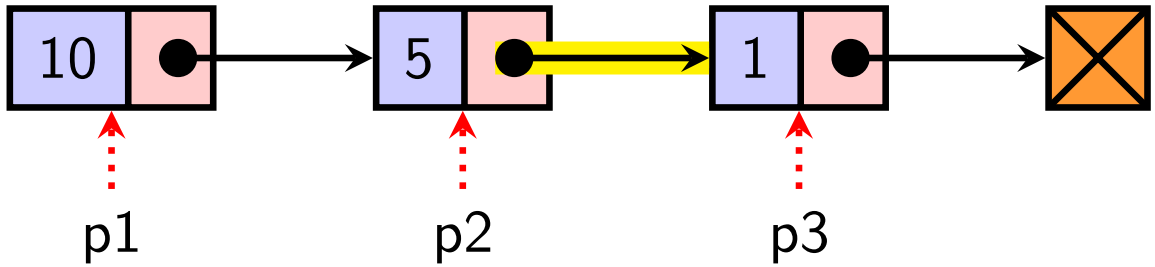
- Cum putem reprezenta următoarea listă?




```
TList p1 = (TList) malloc(sizeof(struct node));
p1->value = 10;
TList p2 = (TList) malloc(sizeof(struct node));
p2->value = 5;
p1->next = p2;
TList p3 = (TList) malloc(sizeof(struct node));
p3->value = 1;
```

Liste simplu înlănțuite – Reprezentarea

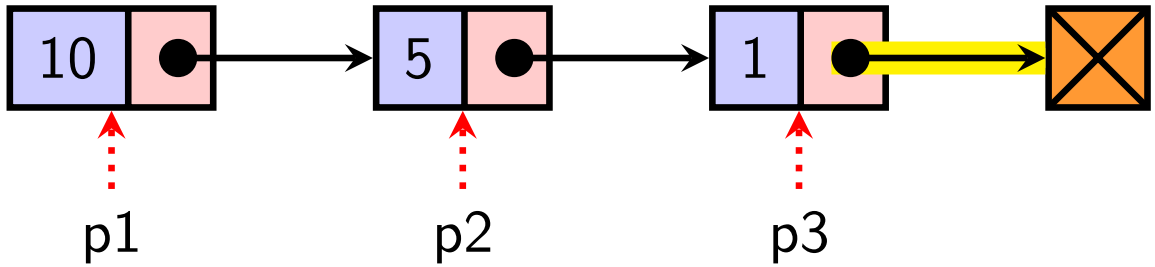
- Cum putem reprezenta următoarea listă?




```
TList p1 = (TList) malloc(sizeof(struct node));
p1->value = 10;
TList p2 = (TList) malloc(sizeof(struct node));
p2->value = 5;
p1->next = p2;
TList p3 = (TList) malloc(sizeof(struct node));
p3->value = 1;
p2->next = p3;
```

Liste simplu înlănțuite – Reprezentarea

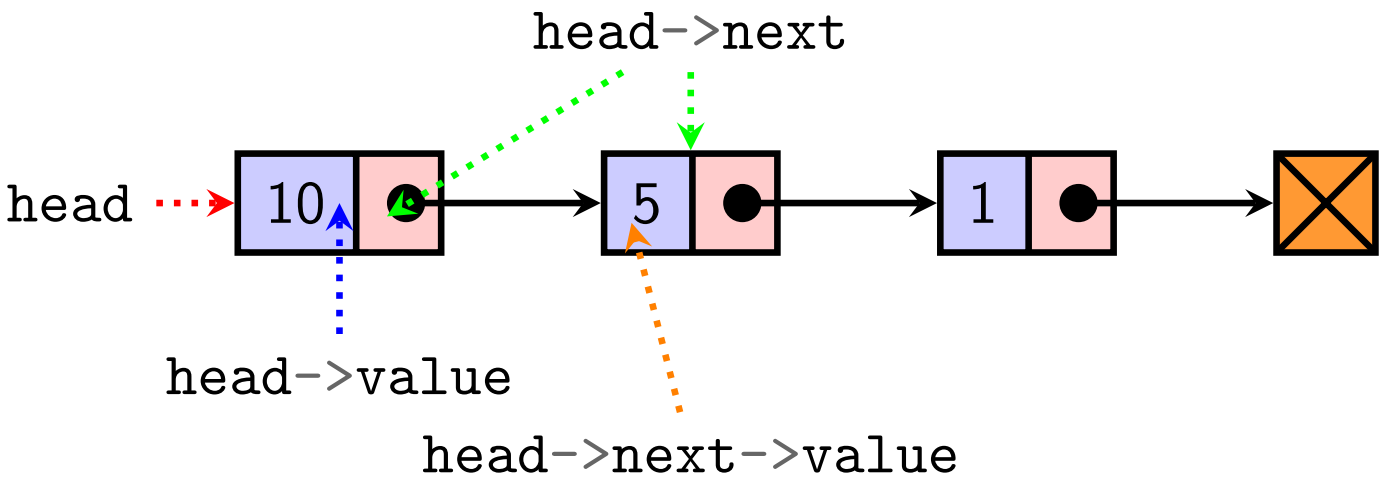
- Cum putem reprezenta următoarea listă?




```
TList p1 = (TList) malloc(sizeof(struct node));
p1->value = 10;
TList p2 = (TList) malloc(sizeof(struct node));
p2->value = 5;
p1->next = p2;
TList p3 = (TList) malloc(sizeof(struct node));
p3->value = 1;
p2->next = p3;
p3->next = NULL;
```

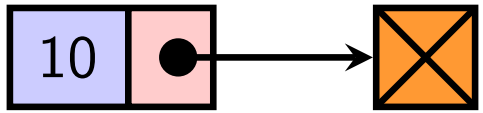
Liste simplu înlănțuite – Accesări

1. Lista vidă
   - Inițializare: `TList p = NULL;`
   - Testare listă vidă: `p == NULL`
2. Lista nevidă (`head != NULL`)



- Adresa primei celule: `head`
- Valoarea primului element din listă: `head->value`
- Adresa următoarei celule: `head->next`
- Test continuare listă: `head->next != NULL`
- Valoarea celui de-al doilea element din listă: `head->next->value`

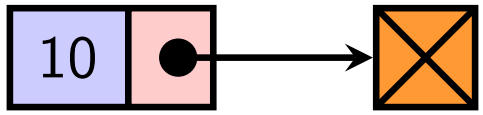
Liste simplu înlănțuite – Inițializare



Varianta I

```
1  TList createList(T value)
2  {
3      TList result = (TList) malloc(sizeof(Node));
4      result->value = value;
5      result->next = NULL;
6      return result;
7  }
8  // ...
9  TList head = createList(10);
```

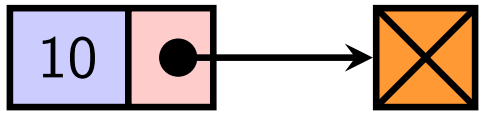
Liste simplu înlănțuite – Inițializare



Varianta I – cu verificarea alocării

```
1  TList createList(T value)
2  {
3      TList result = (TList) malloc(sizeof(Node));
4      if (result == NULL)
5          return result;
6      result->value = value;
7      result->next = NULL;
8      return result;
9  }
10 // ...
11 TList head = createList(10);
```

Liste simplu înlănțuite – Inițializare



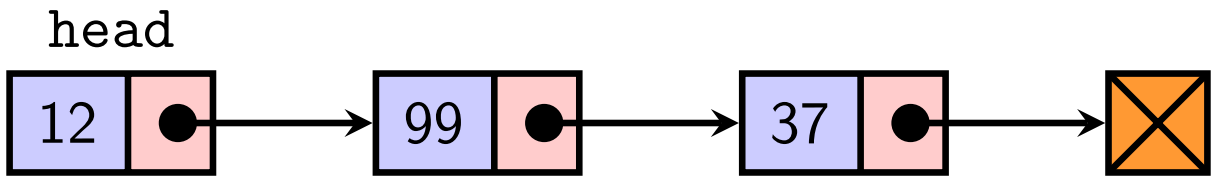
Varianta II – cu verificarea alocării

```
1  void initList(TList *list, T value)
2  {
3      *list = (TList) malloc(sizeof(Node));
4      if (*list == NULL)
5          return;
6      (*list)->value = value;
7      (*list)->next = NULL;
8  }
9  // ...
10 TList head;
11 initList(&head, 10);
```

Liste simplu înlănțuite – Parcurgerea

Important

Pentru a ne asigura că nu pierdem capul listei, va fi nevoie să folosim o variabilă auxiliară pentru a putea parcurge lista.



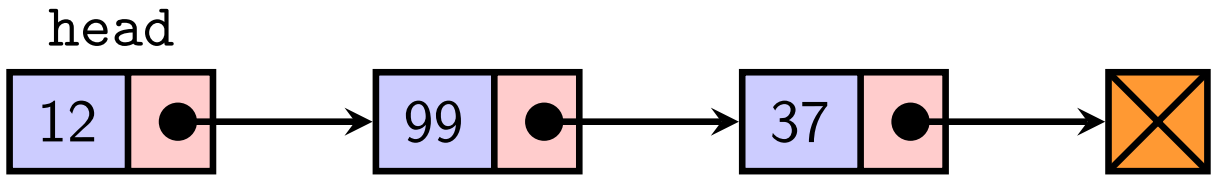
Varianta I – utilizând `for`

```
1  void printList(TList head)
2  {
3      TList iter;
4      for (iter = head; iter != NULL; iter = iter->next)
5          printf("%d ", iter->value);
6      printf("\n");
7  }
```

Liste simplu înlănțuite – Parcurgerea

Important

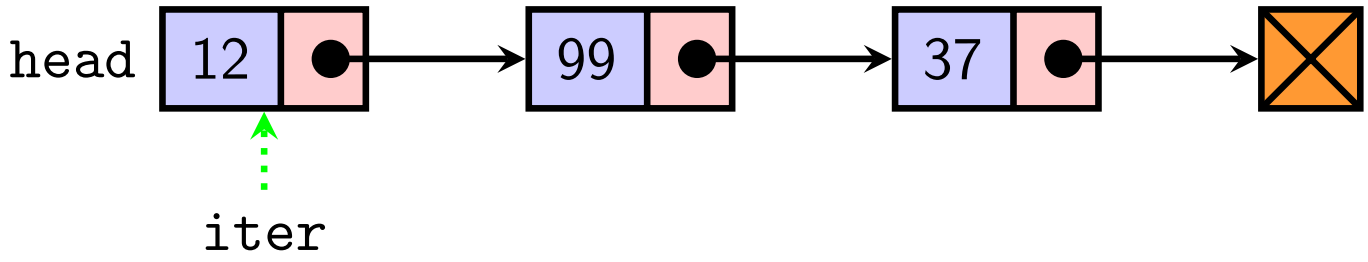
Pentru a ne asigura că nu pierdem capul listei, va fi nevoie să folosim o variabilă auxiliară pentru a putea parcurge lista.



Varianta II – utilizând `while`

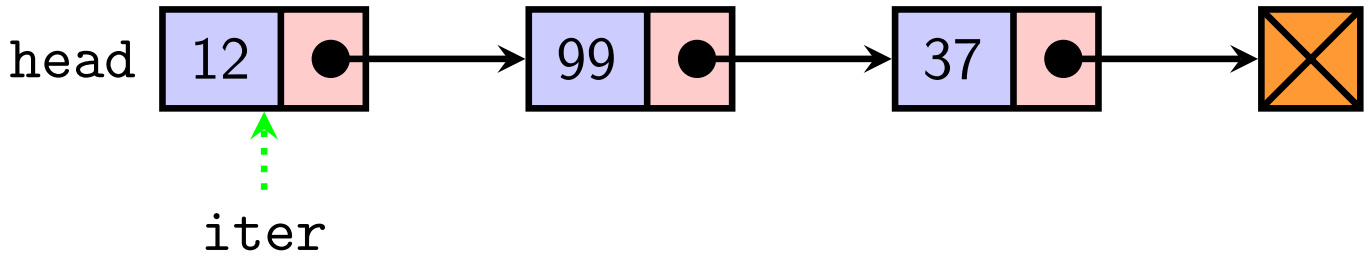
```
1  void printList(TList head)
2  {
3      TList iter = head;
4      while (iter != NULL) {
5          printf("%d ", iter->value);
6          iter = iter->next;
7      }
8      printf("\n");
9  }
```

Liste simplu înlănțuite – Parcurgerea



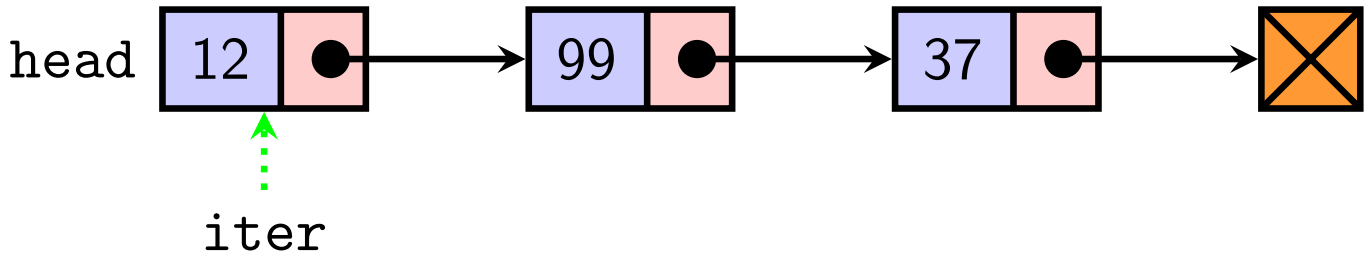

```
1  void printList(TList head)
2  {
3      TList iter = head;
4      while (iter != NULL) {
5          printf("%d ", iter->value);
6          iter = iter->next;
7      }
8      printf("\n");
9  }
```

Liste simplu înlănțuite – Parcurgerea




```
1  void printList(TList head)
2  {
3      TList iter = head;
4      while (iter != NULL) {
5          printf("%d ", iter->value);
6          iter = iter->next;
7      }
8      printf("\n");
9  }
```

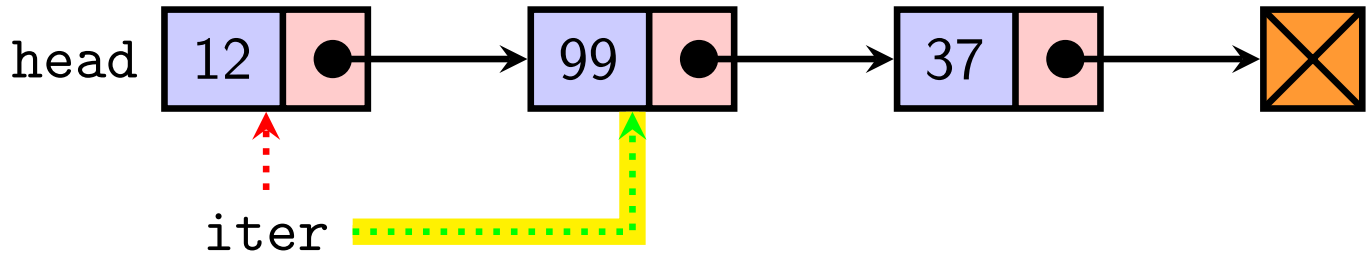
Liste simplu înlănțuite – Parcurgerea




```
1  void printList(TList head)
2  {
3      TList iter = head;
4      while (iter != NULL) {
5          printf("%d ", iter->value);
6          iter = iter->next;
7      }
8      printf("\n");
9  }
```

12

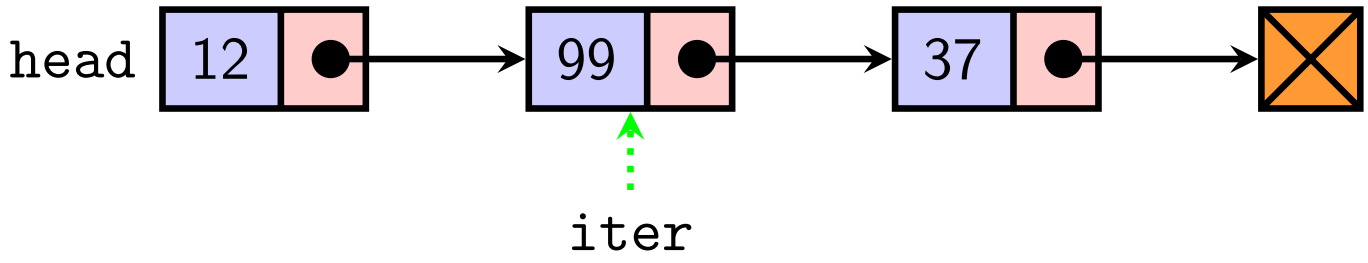
Liste simplu înlănțuite – Parcurgerea




```
1  void printList(TList head)
2  {
3      TList iter = head;
4      while (iter != NULL) {
5          printf("%d ", iter->value);
6          iter = iter->next;
7      }
8      printf("\n");
9  }
```

12

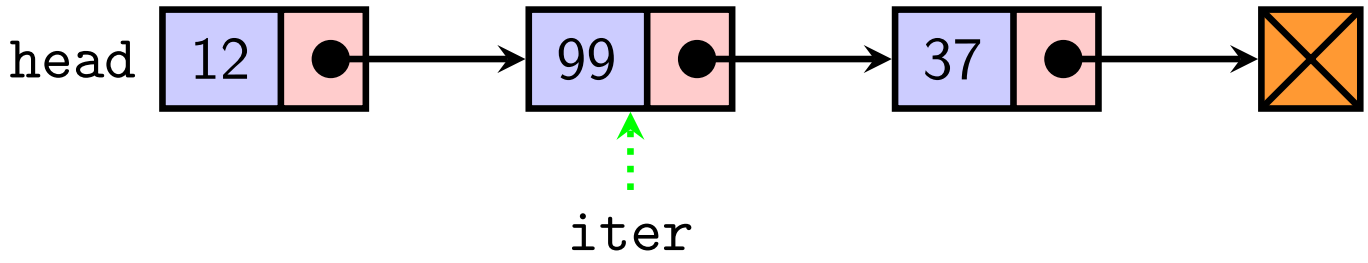
Liste simplu înlănțuite – Parcurgerea




```
1  void printList(TList head)
2  {
3      TList iter = head;
4      while (iter != NULL) {
5          printf("%d ", iter->value);
6          iter = iter->next;
7      }
8      printf("\n");
9  }
```

12

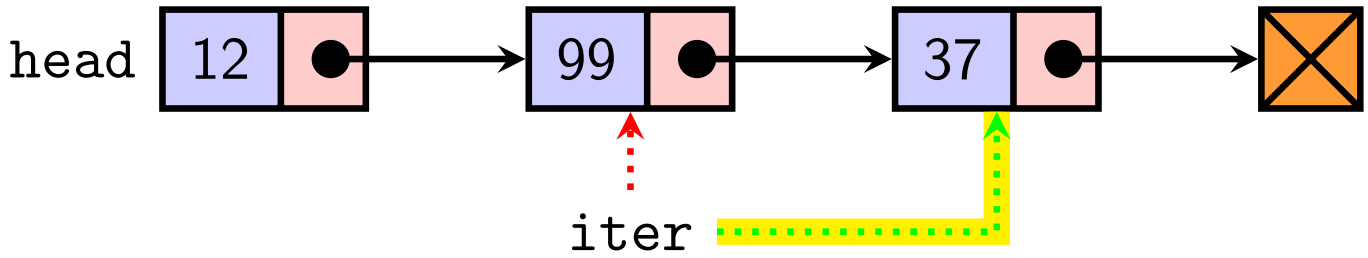
Liste simplu înlănțuite – Parcurgerea




```
1  void printList(TList head)
2  {
3      TList iter = head;
4      while (iter != NULL) {
5          printf("%d ", iter->value);
6          iter = iter->next;
7      }
8      printf("\n");
9  }
```

```
12 99
```

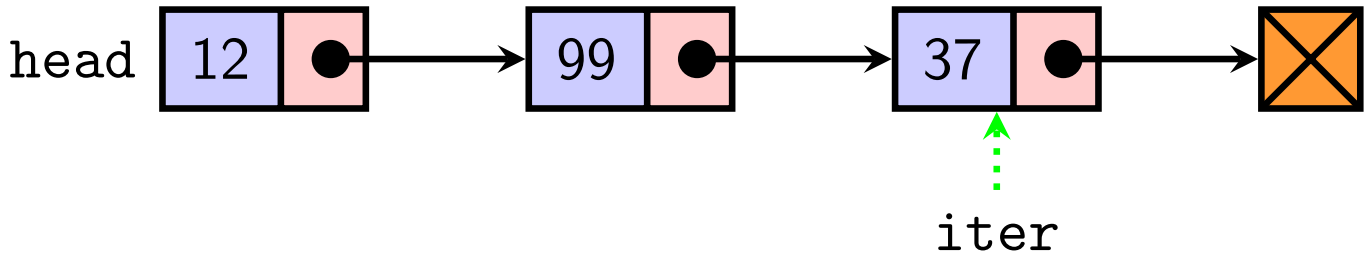
Liste simplu înlănțuite – Parcurgerea




```
1  void printList(TList head)
2  {
3      TList iter = head;
4      while (iter != NULL) {
5          printf("%d ", iter->value);
6          iter = iter->next;
7      }
8      printf("\n");
9  }
```

```
12 99
```

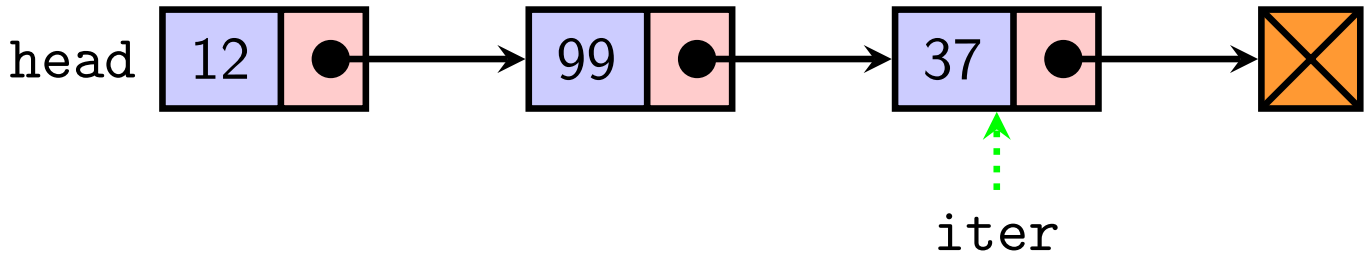
Liste simplu înlănțuite – Parcurgerea




```
1  void printList(TList head)
2  {
3      TList iter = head;
4      while (iter != NULL) {
5          printf("%d ", iter->value);
6          iter = iter->next;
7      }
8      printf("\n");
9  }
```

```
12 99
```

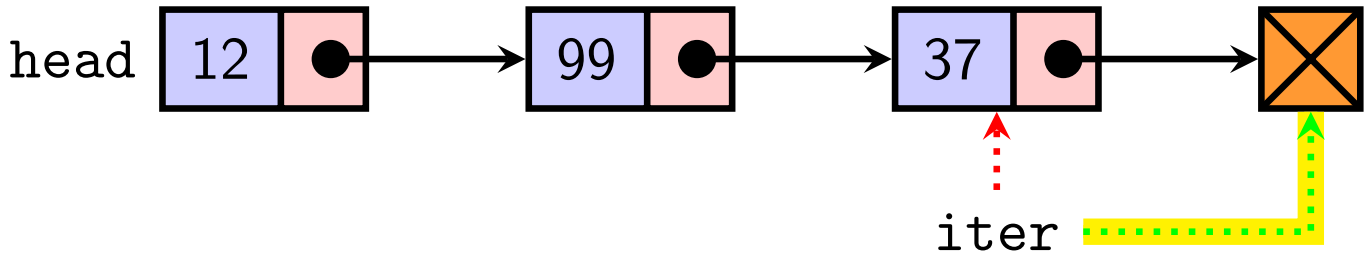
Liste simplu înlănțuite – Parcurgerea




```
1  void printList(TList head)
2  {
3      TList iter = head;
4      while (iter != NULL) {
5          printf("%d ", iter->value);
6          iter = iter->next;
7      }
8      printf("\n");
9  }
```

```
12 99 37
```

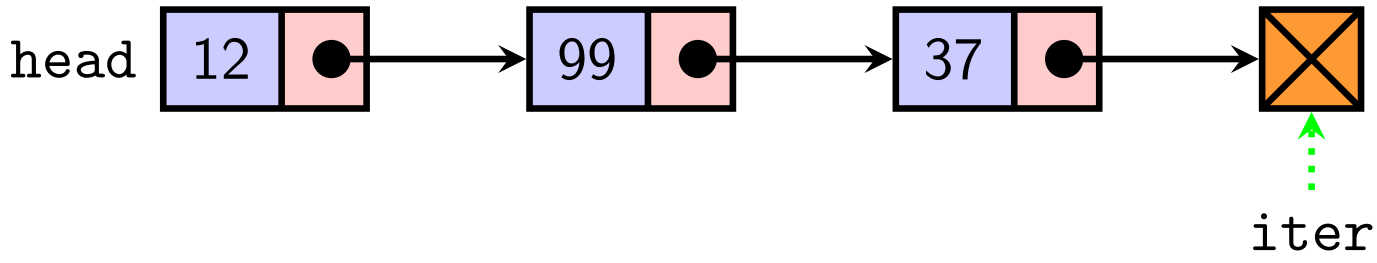
Liste simplu înlănțuite – Parcurgerea




```
1  void printList(TList head)
2  {
3      TList iter = head;
4      while (iter != NULL) {
5          printf("%d ", iter->value);
6          iter = iter->next;
7      }
8      printf("\n");
9  }
```

```
12 99 37
```

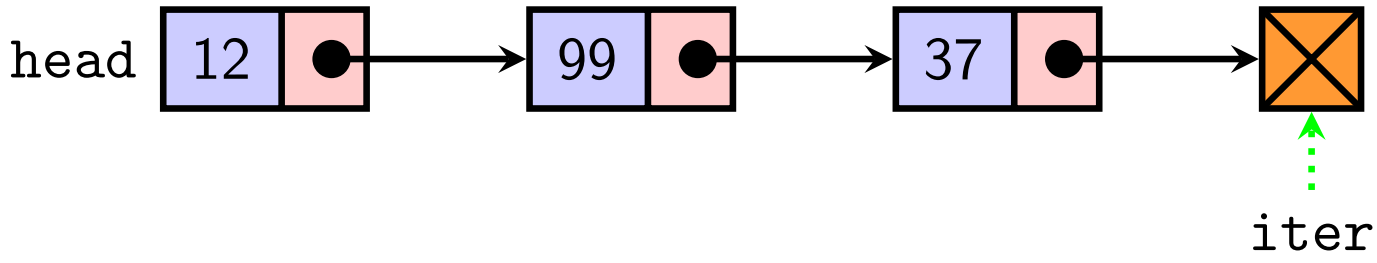
Liste simplu înlănțuite – Parcurgerea




```
1  void printList(TList head)
2  {
3      TList iter = head;
4      while (iter != NULL) {
5          printf("%d ", iter->value);
6          iter = iter->next;
7      }
8      printf("\n");
9  }
```

```
12 99 37
```

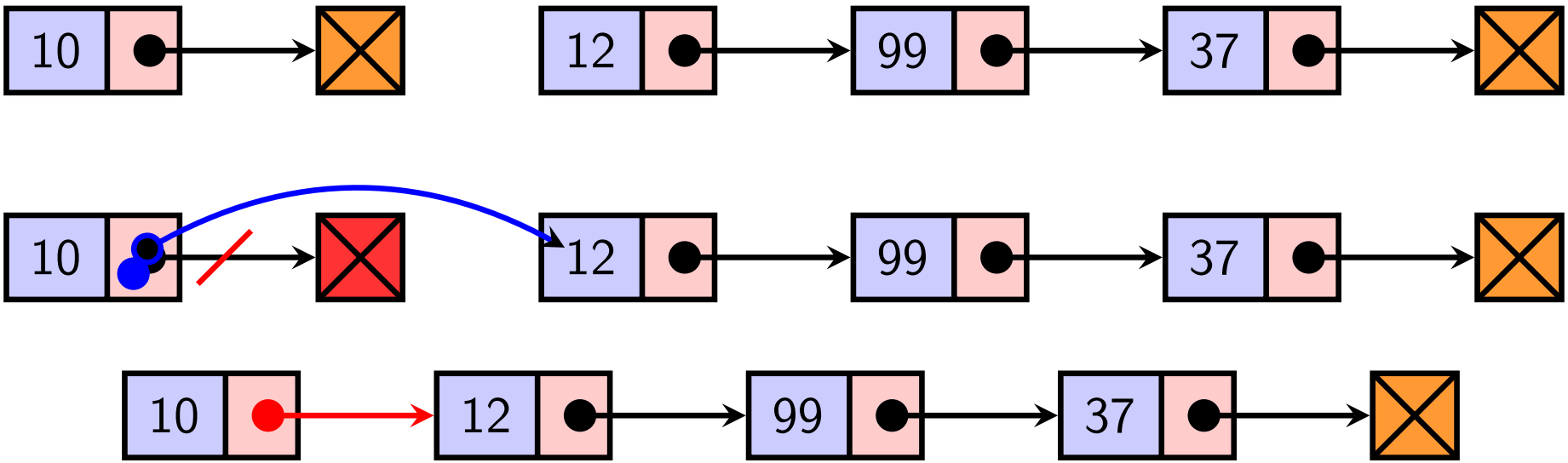
Liste simplu înlănțuite – Parcurgerea




```
1  void printList(TList head)
2  {
3      TList iter = head;
4      while (iter != NULL) {
5          printf("%d ", iter->value);
6          iter = iter->next;
7      }
8      printf("\n");
9  }
```

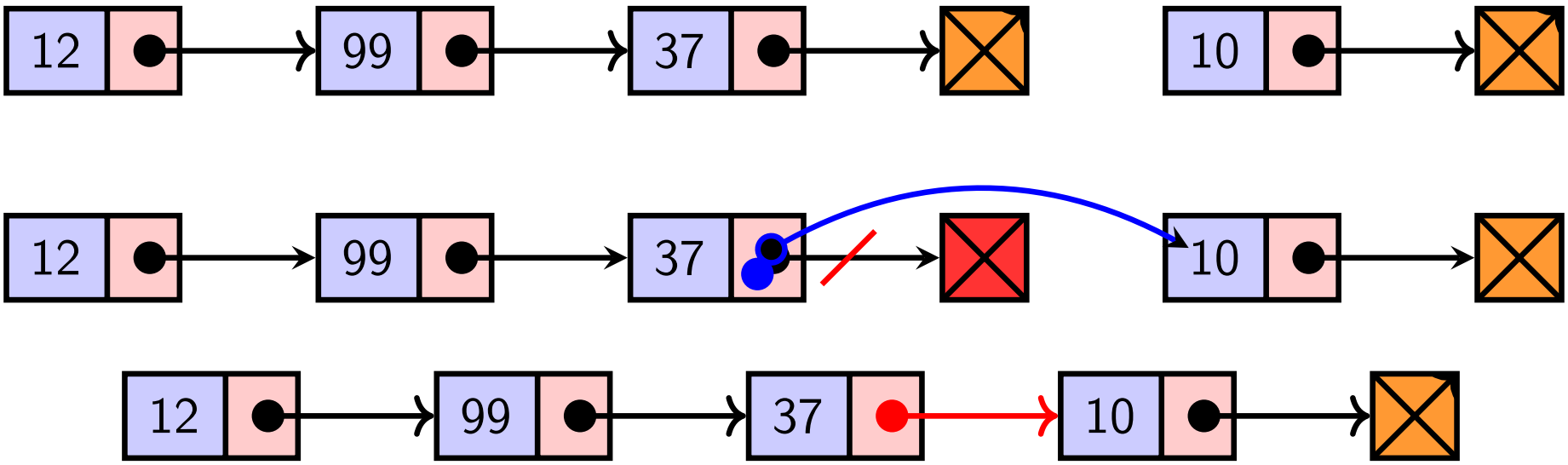
```
12 99 37
```

Liste simplu înlănțuite – Adăugare început



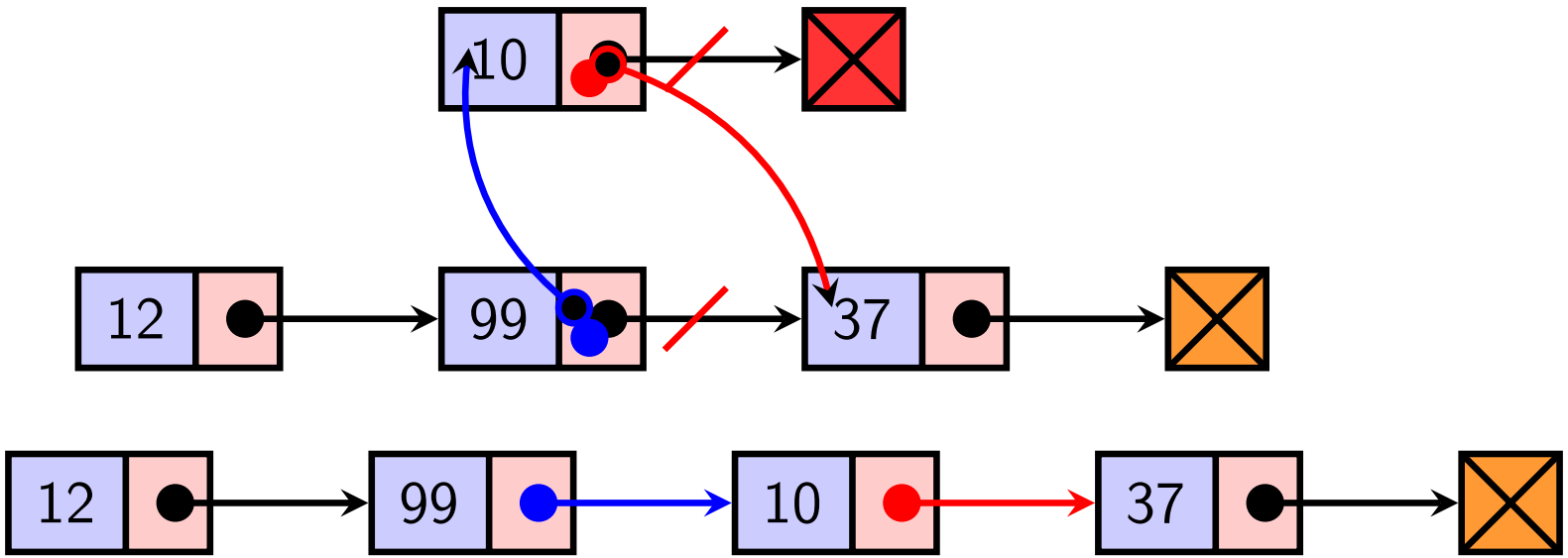

```
1  TList insertFront(TList l, T value)
2  {
3      TList node = createList(value);
4      node->next = l;
5      return node;
6  }
```

Liste simplu înlănțuite – Adăugare la final




```
1   TList insertRear(TList head, T value)
2   {
3       TList iter = head, node;
4       node = createList(value);
5       if (iter == NULL)
6           return node;
7       while (iter->next != NULL)
8           iter = iter->next;
9       iter->next = node;
10      return head;
11  }
```

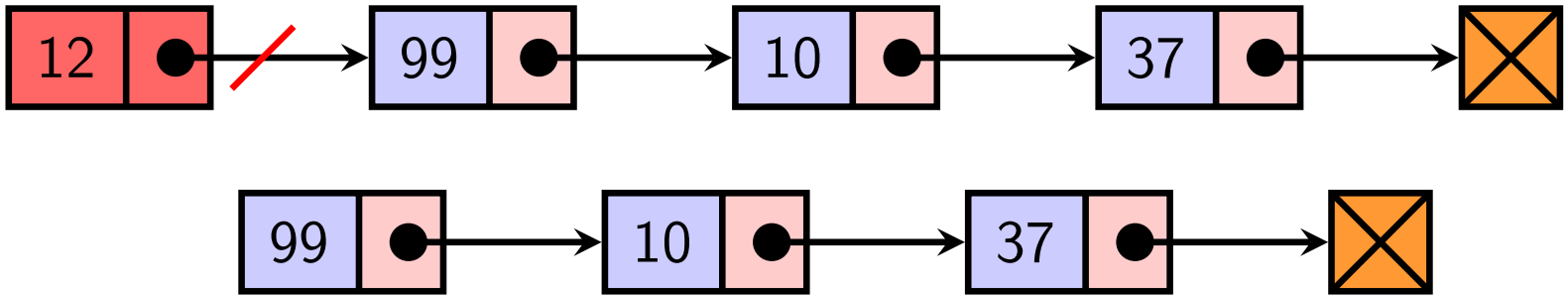
Liste simplu înlănțuite – Adăugare interior



Liste simplu înlănțuite – Adăugare interior

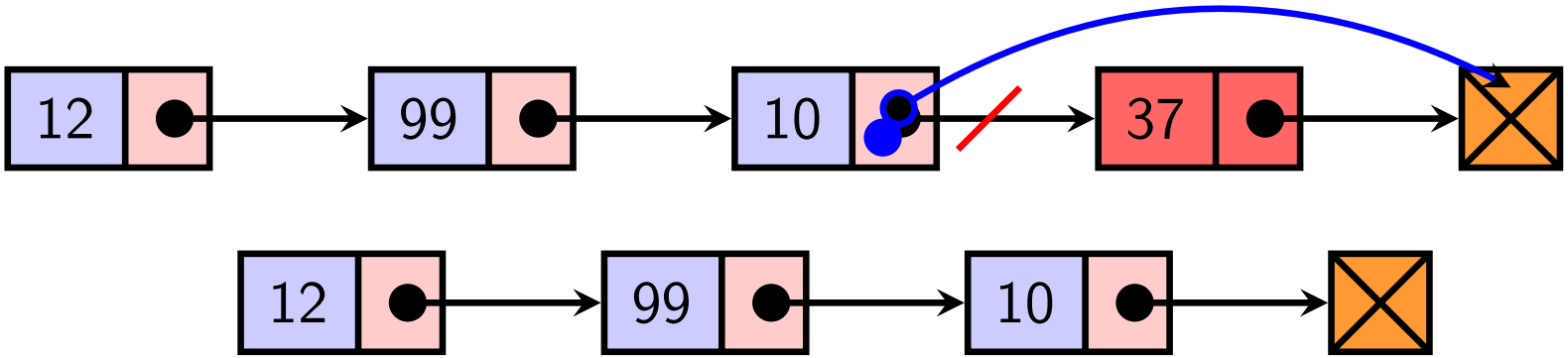
```
1  TList insertAfter(TList head, T x, T value)
2  {
3      TList iter, next, node;
4      if (head == NULL)
5          return head;
6      iter = head;
7      while (iter != NULL) {
8          if (iter->value == x) {
9              node = createList(value);
10             next = iter->next;
11             iter->next = node;
12             node->next = next;
13             return head;
14         }
15         iter = iter->next;
16     }
17     return head;
18 }
```

Liste simplu înlănțuite – Ștergere început

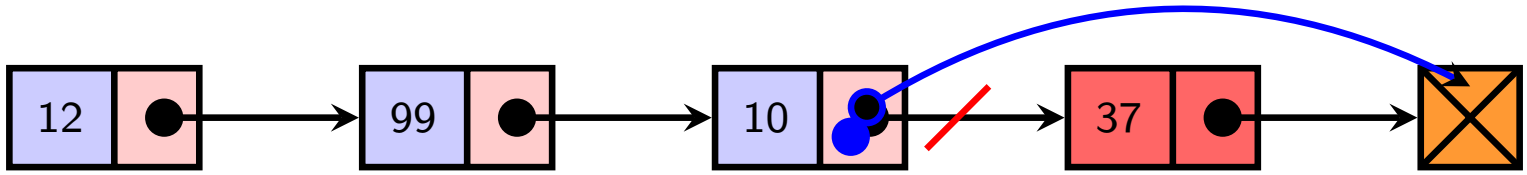



```
1  TList removeFront(TList head)
2  {
3      TList tmp;
4      if (head == NULL)
5          return head;
6      tmp = head;
7      head = head->next;
8      free(tmp);
9      return head;
10 }
```

Liste simplu înlănțuite – Ștergere final



Liste simplu înlănțuite – Ștergere final




```
1   TList removeRear(TList head)
2   {
3       TList itr, prev;
4       if (head == NULL)
5           return head;
6       if (head->next == NULL) {
7           free(head);
8           return NULL;
9       }
10      itr = head->next;
11      prev = head;
12      while (itr->next != NULL) {
13          prev = itr;
14          itr = itr->next;
15      }
16      prev->next = NULL;
17      free(itr);
18      return head;
19  }
```

Recursivitate

1. După ce variabilă(e) fac recursivitatea? (ce variabilă(e) se schimbă de la un apel la altul?)
2. Care sunt condițiile de oprire în funcție de aceste variabile? (cazurile "de bază")
3. Ce se întâmplă când problema nu este încă elementară? (Obligatoriu aici cel puțin un apel recursiv.)

Recursivitate – Exemple

- Recursivitate pe stivă

```
1  int factorial1(int nr) {
2      if (nr == 0) {
3          return 1;
4      }
5      return nr * factorial1(nr - 1);
6  }
```

- Recursivitate pe coadă

```
1  int factorial2(int nr, int acc) {
2      if (nr == 0)
3          return acc;
4      return factorial2(nr - 1, nr * acc);
5  }
```

Terminarea recursivității

Important

O funcție recursivă trebuie să aibă în construcția ei cel puțin o condiție de terminare.

Observație

Altfel, apelul recursiv poate să ducă la o buclă infinită (asemănătoare structurilor repetitive în care condiția de continuare este întotdeauna adevărată și care iterează la nesfârșit)!

Important

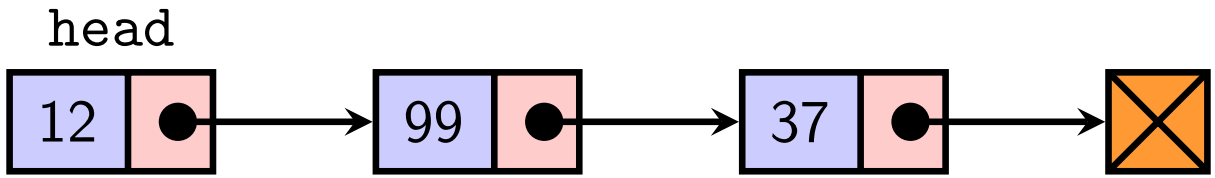
Adâncimea recursivității trebuie să nu fie una foarte mare.

Observație

În cazul depășirii dimensiunii maxime a stivei, programul se termină subit în urma unei erori – **stack overflow**.


Learn to program
Make recursive function
No exit condition
Learn to program
Make recursive function
No exit condition
Learn to program
Make recursive function

Determinarea lungimii unei liste



Pentru a putea determina lungimea listei, este nevoie să o parcurgem și să-i numărăm nodurile.

Varianta recursivă pe stivă

```
int length(TList head)
{
    if (head == NULL)
        return 0;
    return 1 + length(head->next);
}
// ...
int size = length(head);
```

Varianta recursivă pe coadă

```
int length(TList head, int size)
{
    if (head == NULL)
        return size;
    return length(head->next, size + 1);
}
// ...
int size = length(head, 0);
```

Ștergerea unui element dintr-o listă

```
1  TList delete(TList head, T value) {
2      if (head == NULL)
3          return head;
4      if (head->value == value)
5          return removeFront(head);
6      TList prev, iter;
7      prev = head;
8      iter = head->next;
9      while (iter != NULL) {
10         if (iter->value == value) {
11             prev->next = iter->next;
12             free(iter);
13             return head;
14         }
```

List of References

```
15          prev = iter;
16          iter = iter->next;
17      }
18      return head;
19  }
```

Dealocarea memoriei pentru o listă

```
1  TList freeList(TList l) {
2      TList tmp;
3      while (l != NULL) {
4          tmp = l;
5          l = l->next;
6          free(tmp);
7      }
8      return NULL;
9  }
```

- De ce avem nevoie de `tmp`?

Vă mulțumesc pentru atenție!