# Structuri de Date și Algoritmi
# Liste dublu înlănțuite

**Mihai Nan**

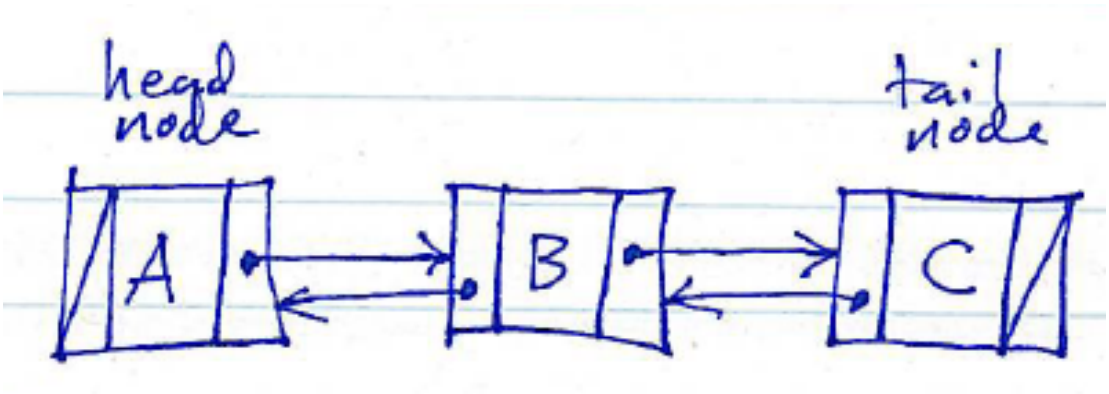
Departamentul de Calculatoare
Facultatea de Automatică și Calculatoare
Universitatea POLITEHNICA din București


**Anul Universitar 2022–2023**

# Conținutul cursului

# Introducere

- Listele dublu înlănțuite sunt structuri de date dinamice omogene.
- Fiecare nod al listei conține, în afară de informația utilă, adresa următorului element și adresa precedentului element.
- În continuare avem numai acces secvențial la elementele listei.

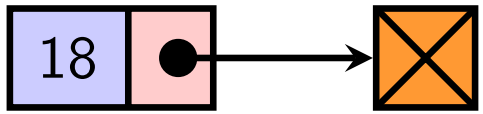



Pentru parcurgerea eficientă a listei, indiferent de sens (de la **început** sau de la **sfârșit**), este necesar accesul la cele două extremități.

# Liste dublu înlănțuite – Reprezentarea

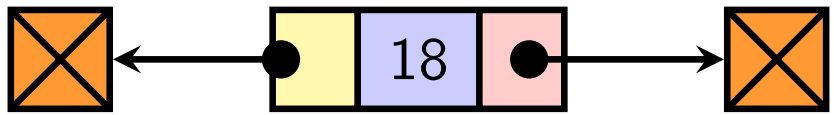
1. Lista simplu înlănțuită

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




2. Lista dublu înlănțuită

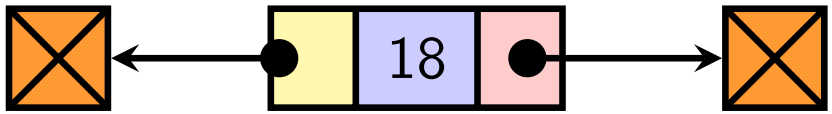
```
1  typedef int T;
2  typedef struct node {
3      T value;
4      struct node* prev;
5      struct node* next;
6  } Node, *TList;
```



# Liste dublu înlănțuite – Reprezentarea

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

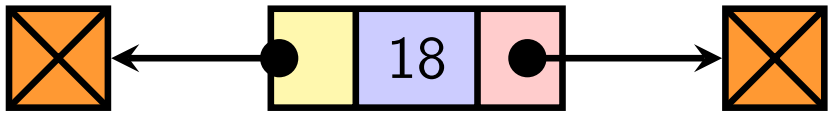
- Cum putem reprezenta următoarea listă?



# Liste dublu înlănțuite – Reprezentarea

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

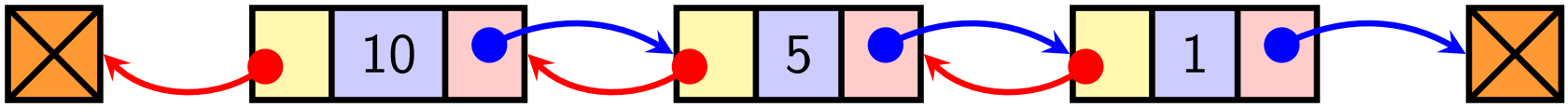
- Cum putem reprezenta următoarea listă?




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

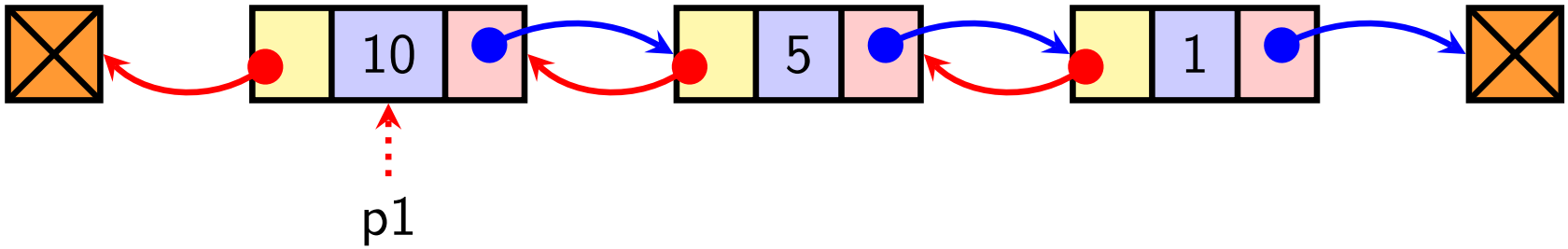
# Liste dublu înlănțuite – Reprezentarea

- Cum putem reprezenta următoarea listă?



# Liste dublu înlănțuite – Reprezentarea

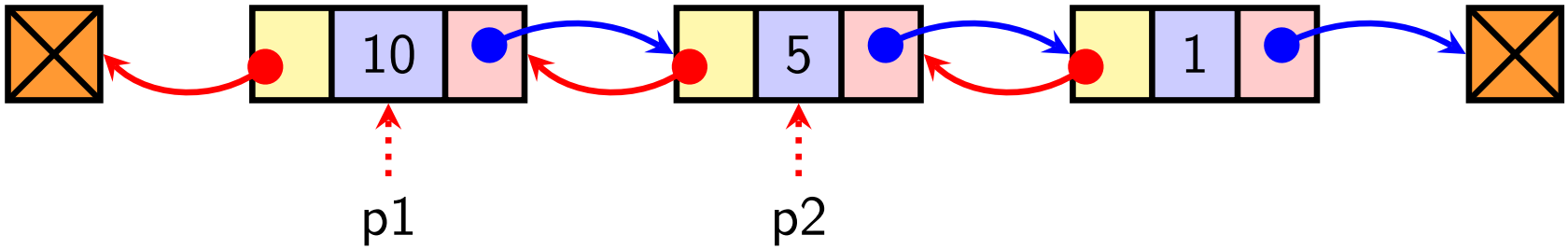
- Cum putem reprezenta următoarea listă?




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

# Liste dublu înlănțuite – Reprezentarea

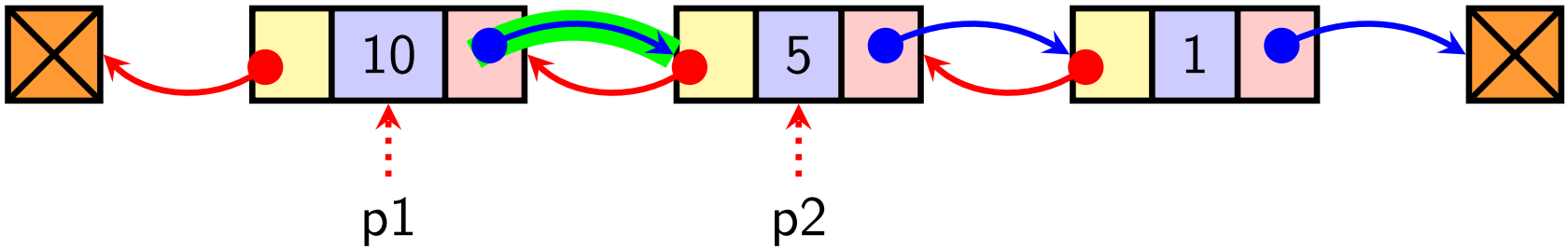
- Cum putem reprezenta următoarea listă?




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

# Liste dublu înlănțuite – Reprezentarea

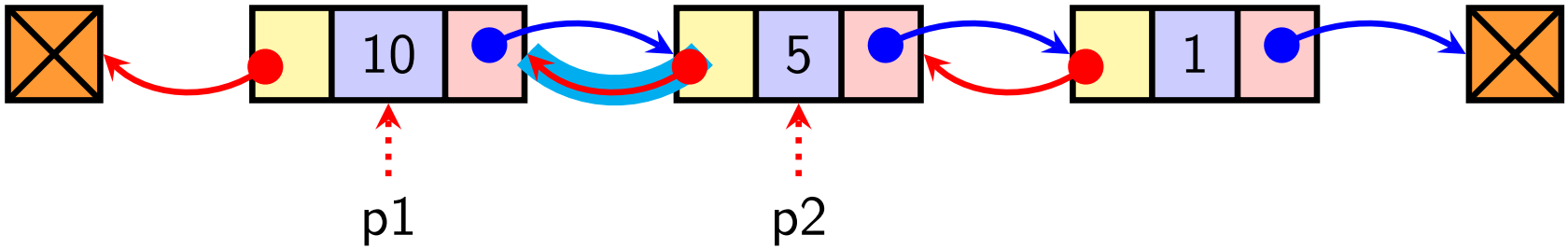
- Cum putem reprezenta următoarea listă?




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

# Liste dublu înlănțuite – Reprezentarea

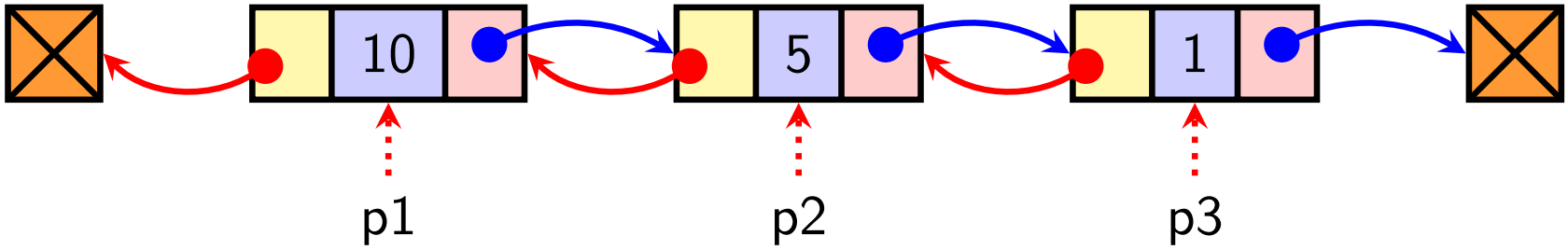
- Cum putem reprezenta următoarea listă?




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

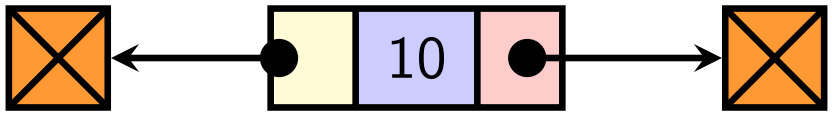
# Liste dublu înlănțuite – Reprezentarea

- Cum putem reprezenta următoarea listă?




```
1  TList p1 = (TList) malloc(sizeof(struct node));
2  p1->value = 10;
3  p1->prev = NULL;
4  TList p2 = (TList) malloc(sizeof(struct node));
5  p2->value = 5;
6  p1->next = p2;
7  p2->prev = p1;
8  TList p3 = (TList) malloc(sizeof(struct node));
9  p3->value = 1;
10 p2->next = p3;
11 p3->prev = p2;
12 p3->next = NULL;
```

# Liste simplu înlănțuite – Inițializare




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

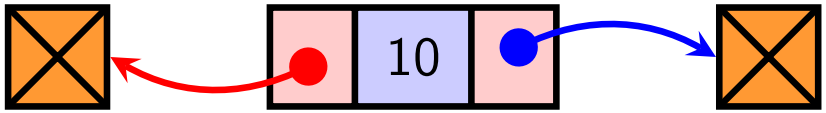
# Liste dublu înlănțuite – Adăugare la început

- Inserarea valorii 10 la începutul listei

# Liste dublu înlănțuite – Adăugare la început

- Inserarea valorii 10 la începutul listei

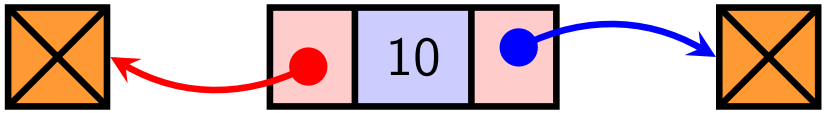

1. Inițializez o listă cu un singur nod ce conține valoarea 10



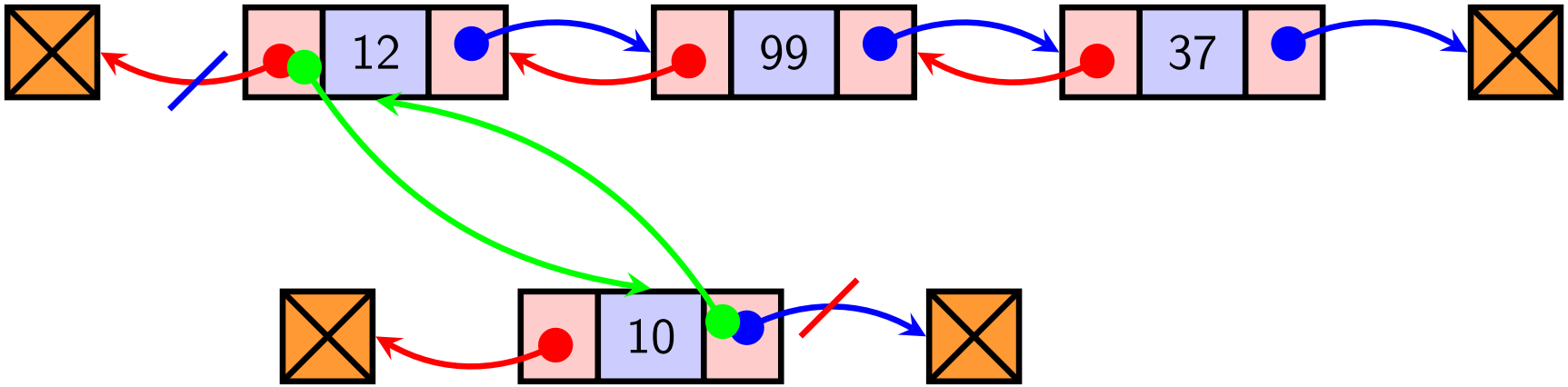
# Liste dublu înlănțuite – Adăugare la început

- Inserarea valorii 10 la începutul listei


1. Inițializez o listă cu un singur nod ce conține valoarea 10



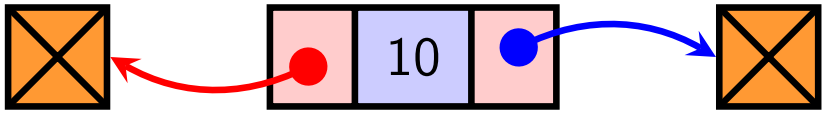

2. Refac legăturile



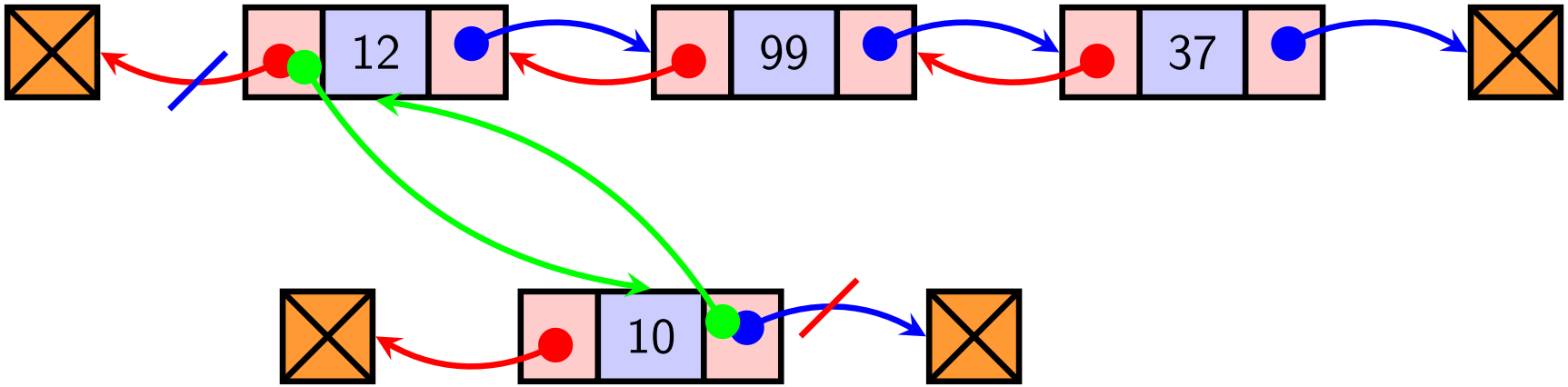
# Liste dublu înlănțuite – Adăugare la început

- Inserarea valorii 10 la începutul listei

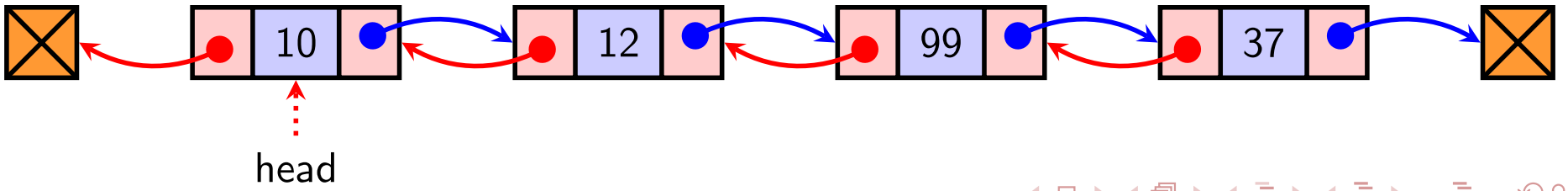

1. Inițializez o listă cu un singur nod ce conține valoarea 10




2. Refac legăturile




3. Lista rezultată



# Liste dublu înlănțuite – Adăugare la început

```
1  TList insertFront(TList head, T value) {
2      TList new = createList(value);
3      new->next = head;
4      if (head != NULL) {
5          head->prev = new;
6      }
7      return new;
8  }
```

# Liste dublu înlănțuite – Adăugare la început

```
1  TList insertFront(TList head, T value) {
2      TList new = createList(value);
3      new->next = head;
4      if (head != NULL) {
5          head->prev = new;
6      }
7      return new;
8  }
```

- Ce complexitate are această funcție?

# Liste dublu înlănțuite – Adăugare la început

```
1  TList insertFront(TList head, T value) {
2      TList new = createList(value);
3      new->next = head;
4      if (head != NULL) {
5          head->prev = new;
6      }
7      return new;
8  }
```

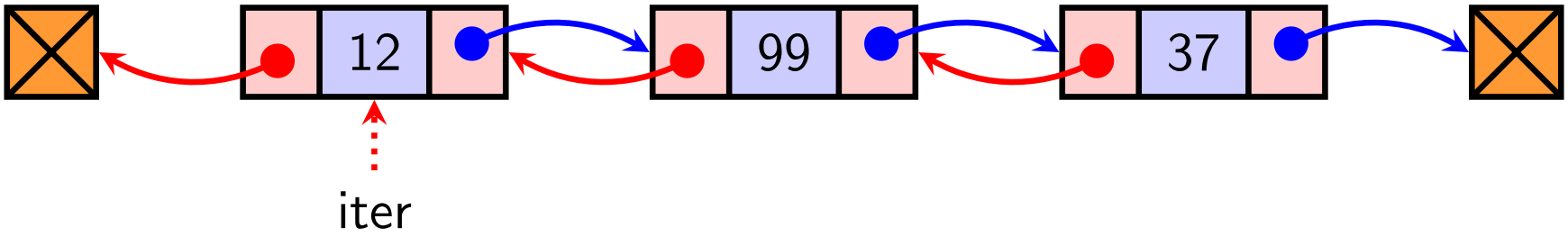
- Ce complexitate are această funcție?

**Răspuns**

$O(1)$ – se execută în timp constant (nu depinde de numărul de elemente din listă)

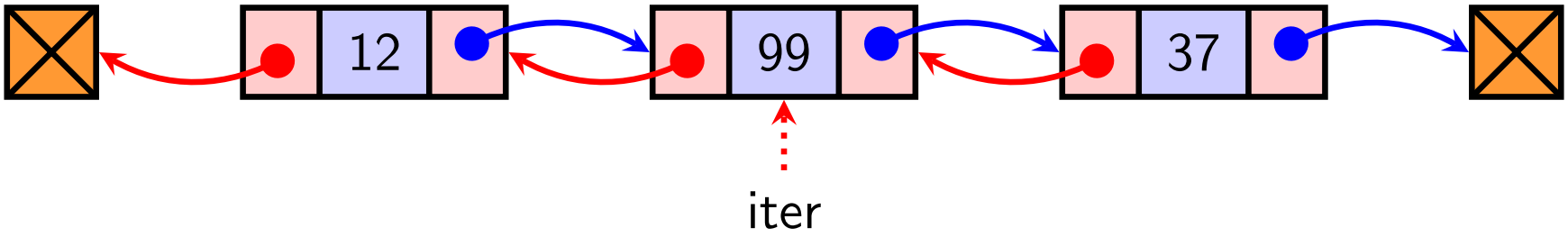
# Liste dublu înlănțuite – Adăugare la final

- Inserarea valorii 10 la finalul listei



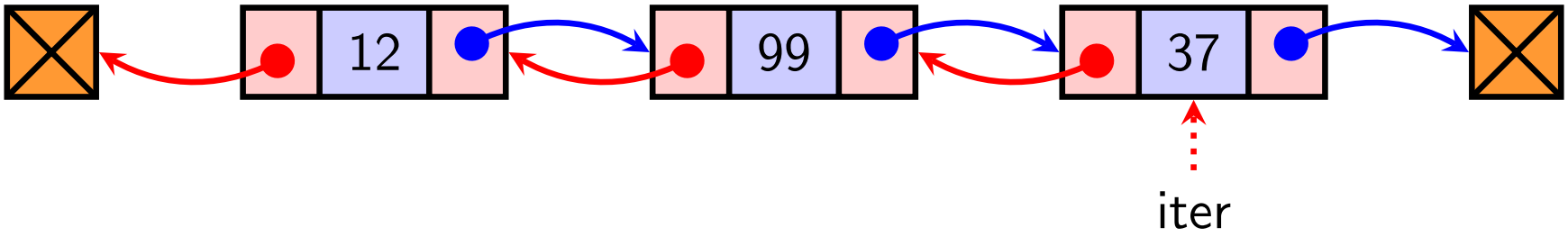
# Liste dublu înlănțuite – Adăugare la final

- Inserarea valorii 10 la finalul listei



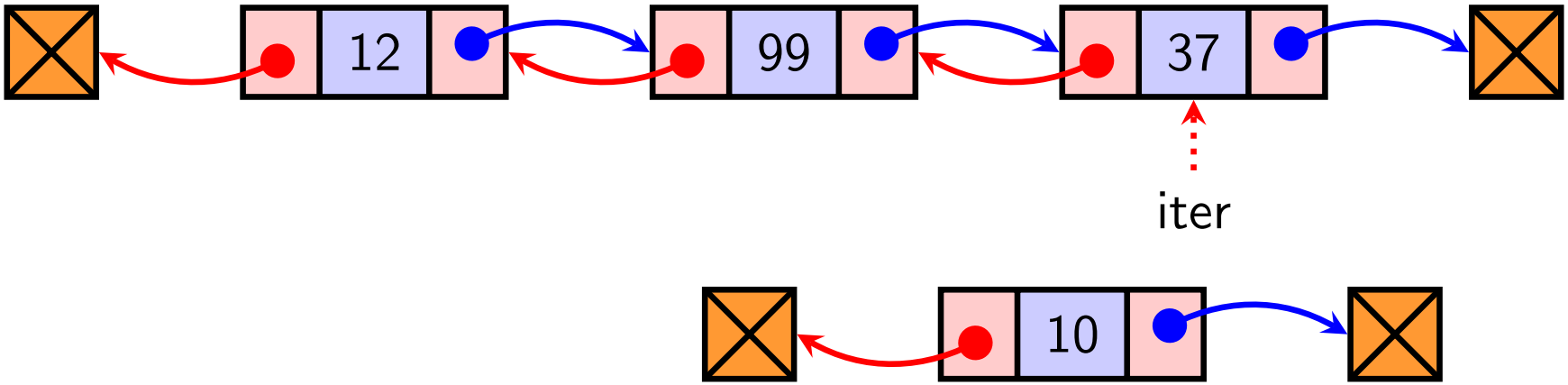
# Liste dublu înlănțuite – Adăugare la final

- Inserarea valorii 10 la finalul listei



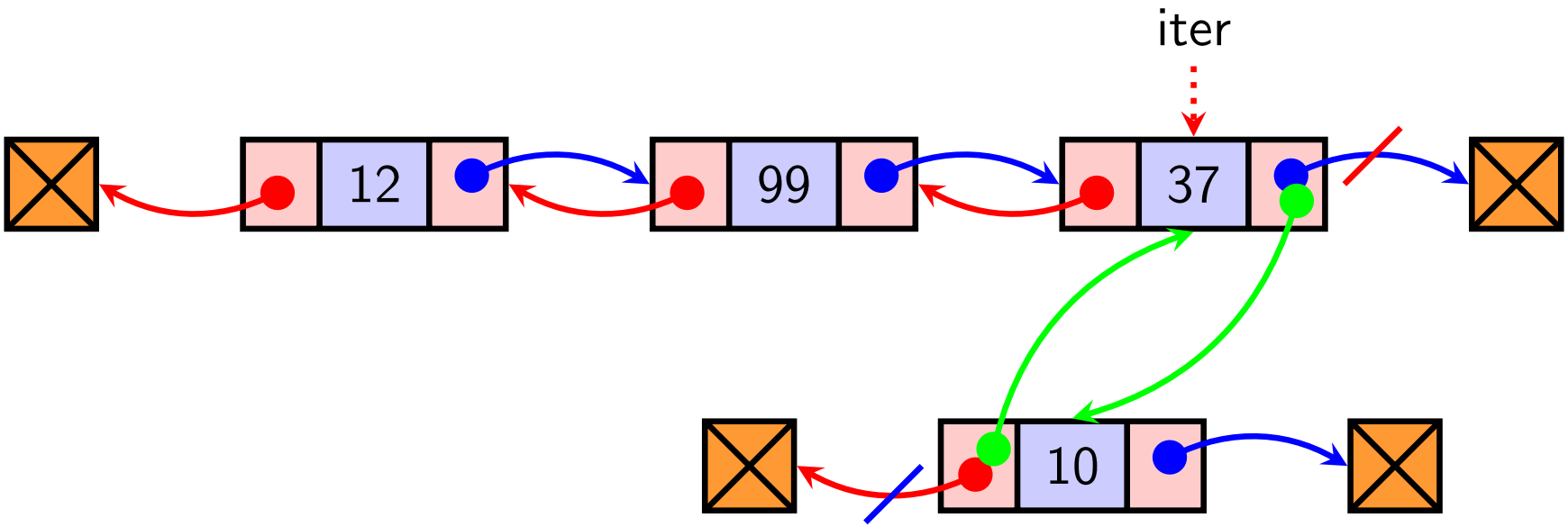
# Liste dublu înlănțuite – Adăugare la final

- Inserarea valorii 10 la finalul listei



# Liste dublu înlănțuite – Adăugare la final

- Inserarea valorii 10 la finalul listei



# Liste dublu înlănțuite – Adăugare la final

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

# Liste dublu înlănțuite – Adăugare la final

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

- Ce complexitate are această funcție?

# Liste dublu înlănțuite – Adăugare la final

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

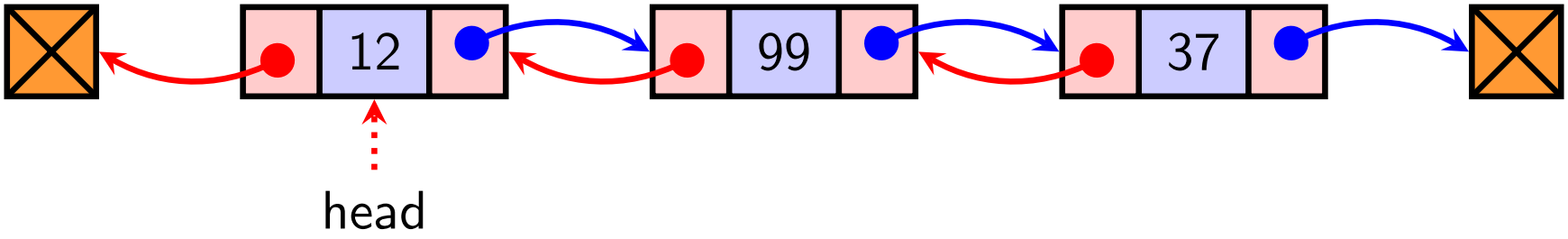
- Ce complexitate are această funcție?

**Răspuns**

$O(N)$ – unde $N$ reprezintă numărul de elemente din listă (trebuie să parcurgem elementele din listă pentru a-l accesa pe ultimul)

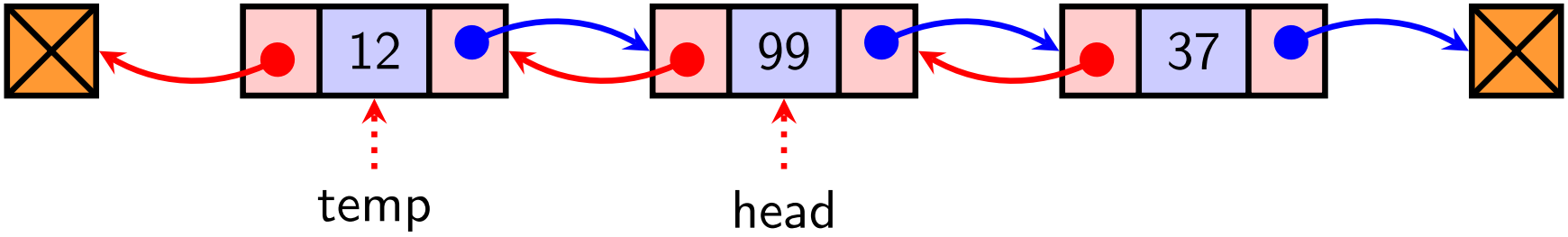
# Liste dublu înlănțuite – Ștergere început

- Dorim să ștergem primul element din următoarea listă



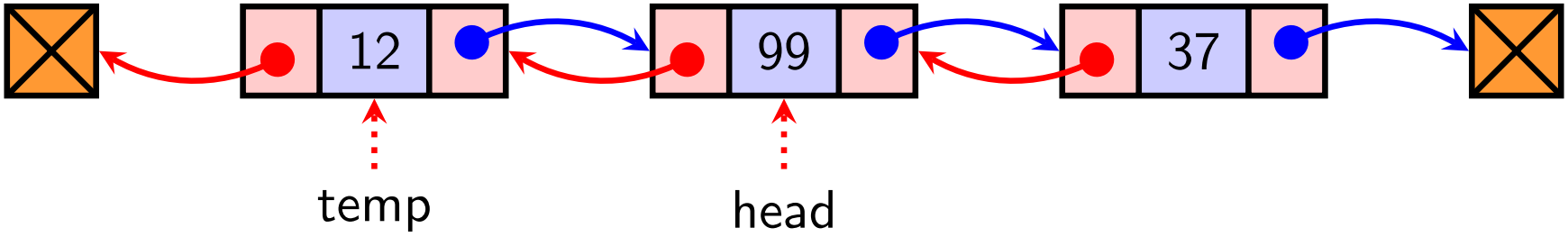
# Liste dublu înlănțuite – Ștergere început

- Dorim să ștergem primul element din următoarea listă

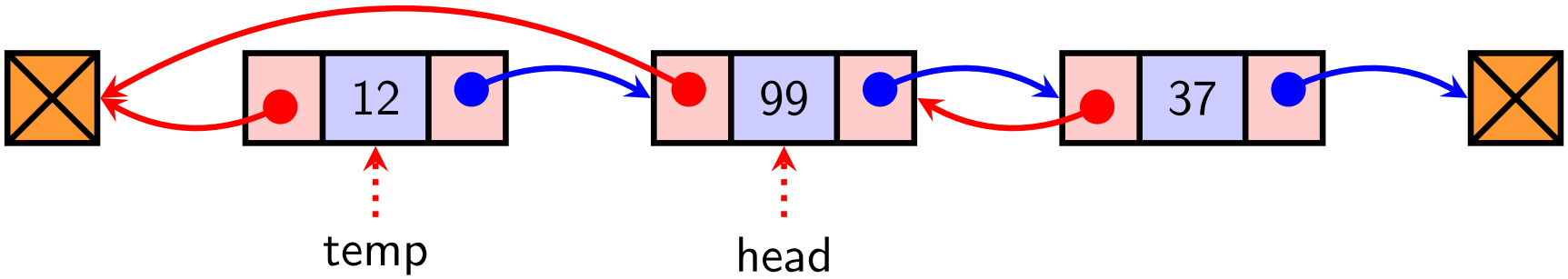


# Liste dublu înlănțuite – Ștergere început

- Dorim să ștergem primul element din următoarea listă

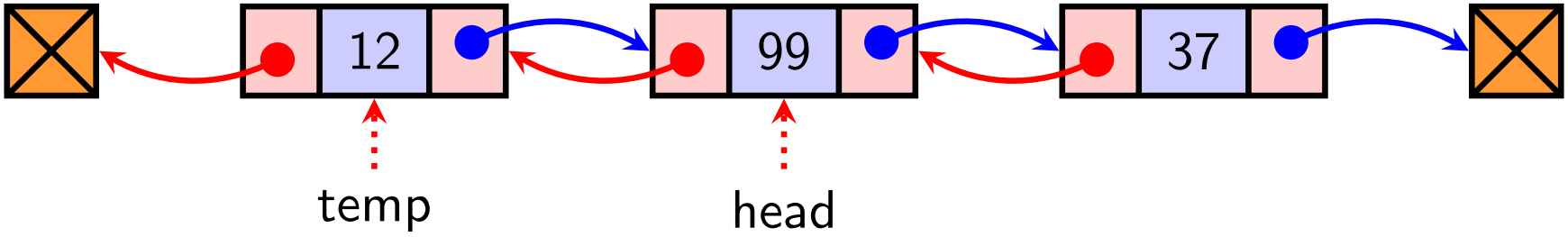



1. Refac legăturile pentru noul head

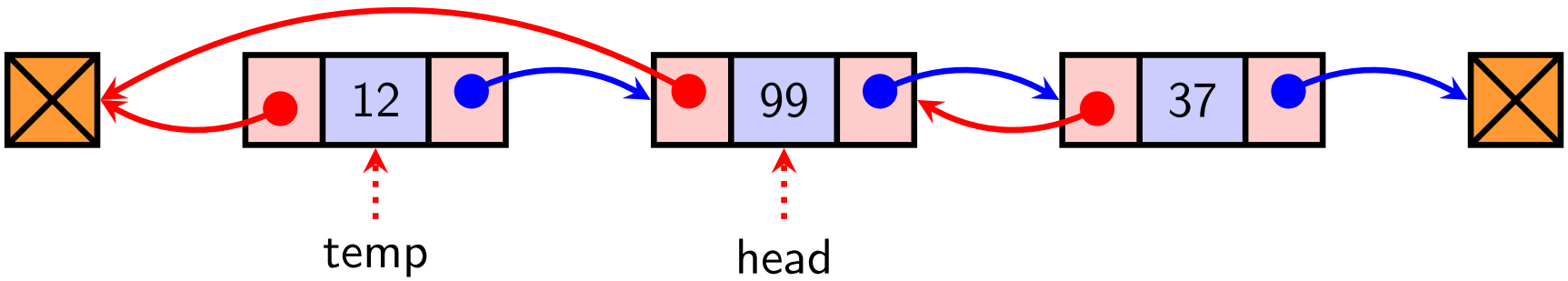


# Liste dublu înlănțuite – Ștergere început

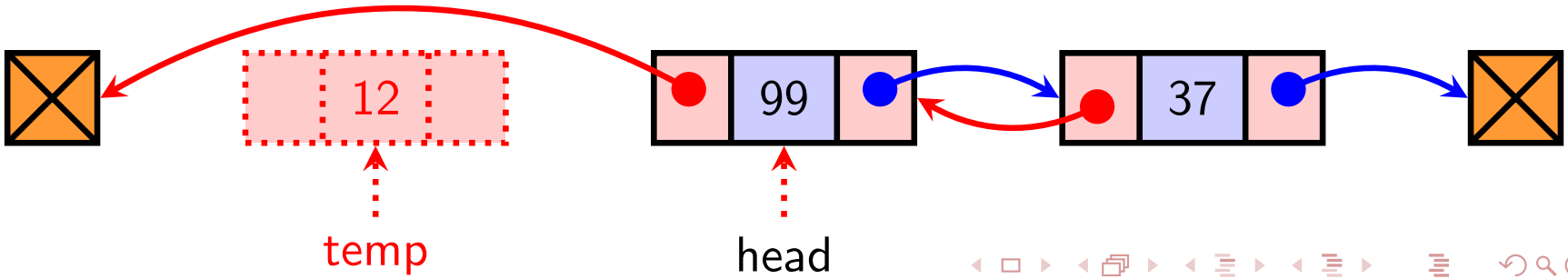
- Dorim să ștergem primul element din următoarea listă




1. Refac legăturile pentru noul head




2. Dealoc memoria pentru temp



# Liste dublu înlănțuite – Ștergere început

```
1  TList removeFront(TList head) {
2      TList temp;
3      if (head == NULL)
4          return head;
5      temp = head;
6      head = head->next;
7      if (head != NULL)
8          head->prev = NULL;
9      free(temp);
10     return head;
11 }
```

# Liste dublu înlănțuite – Ștergere început

```
1  TList removeFront(TList head) {
2      TList temp;
3      if (head == NULL)
4          return head;
5      temp = head;
6      head = head->next;
7      if (head != NULL)
8          head->prev = NULL;
9      free(temp);
10     return head;
11 }
```

- Ce complexitate are această funcție?

# Liste dublu înlănțuite – Ștergere început

```
1  TList removeFront(TList head) {
2      TList temp;
3      if (head == NULL)
4          return head;
5      temp = head;
6      head = head->next;
7      if (head != NULL)
8          head->prev = NULL;
9      free(temp);
10     return head;
11 }
```

- Ce complexitate are această funcție?

**Răspuns**

$O(1)$ – se execută în timp constant (nu depinde de numărul de elemente din listă)

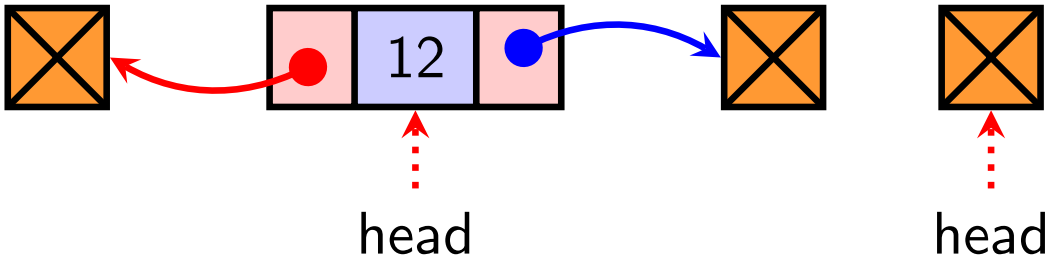
# Liste dublu înlănțuite – Ștergere final

1. Dacă lista este vidă, nu avem ce șterge.

# Liste dublu înlănțuite – Ștergere final

1. Dacă lista este vidă, nu avem ce șterge.

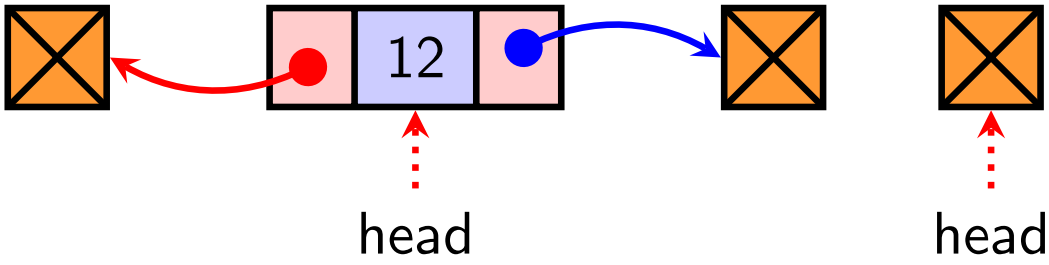
2. Dacă lista conține un singur nod, obținem lista vidă.



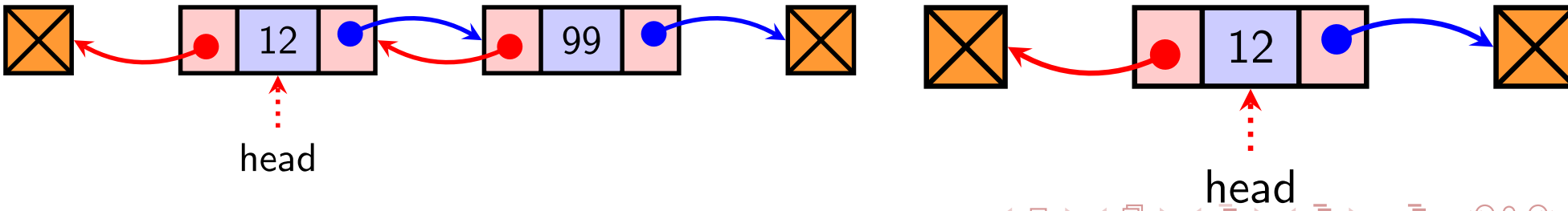
# Liste dublu înlănțuite – Ștergere final

1. Dacă lista este vidă, nu avem ce șterge.

2. Dacă lista conține un singur nod, obținem lista vidă.




3. Dacă lista conține cel puțin două noduri, eliminăm ultimul nod și `head` rămâne nemodificat.



# Liste dublu înlănțuite – Ștergere final

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

# Liste dublu înlănțuite – Recapitulare

- Modalitate de reprezentare

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

# Liste dublu înlănțuite – Recapitulare

- Modalitate de reprezentare

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

- Adăugare la începutul listei – Complexitatea: $O(1)$

# Liste dublu înlănțuite – Recapitulare

- Modalitate de reprezentare

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

- Adăugare la începutul listei – Complexitatea: $O(1)$
- Adăugare la finalul listei – Complexitatea: $O(N)$

# Liste dublu înlănțuite – Recapitulare

- Modalitate de reprezentare

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

- Adăugare la începutul listei – Complexitatea: $O(1)$
- Adăugare la finalul listei – Complexitatea: $O(N)$
- Ștergerea de la începutul listei – Complexitatea: $O(1)$

# Liste dublu înlănțuite – Recapitulare

- Modalitate de reprezentare

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

- Adăugare la începutul listei – Complexitatea: $O(1)$
- Adăugare la finalul listei – Complexitatea: $O(N)$
- Ștergerea de la începutul listei – Complexitatea: $O(1)$
- Ștergerea de la finalul listei – Complexitatea $O(N)$

# Liste dublu înlănțuite – Recapitulare

- Modalitate de reprezentare

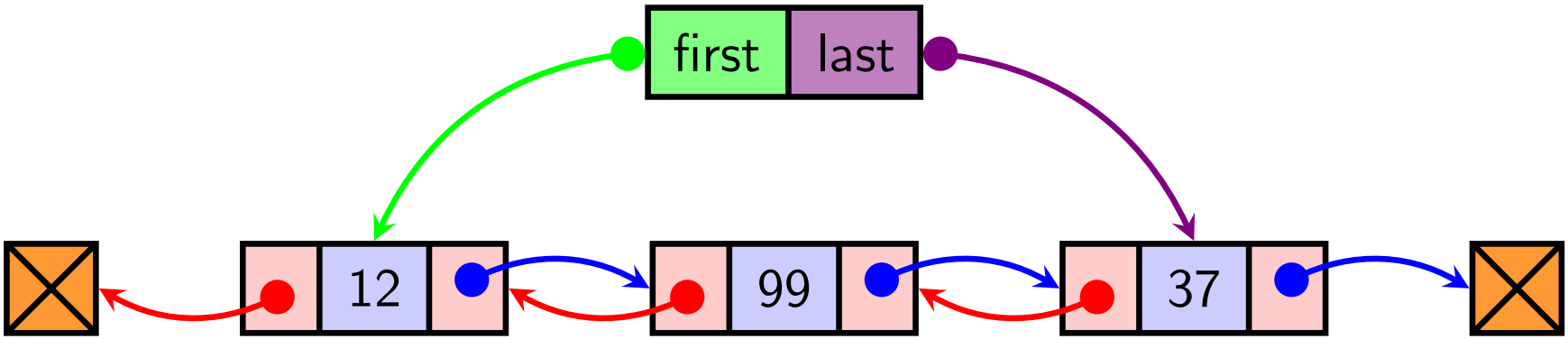
```
1  typedef int T;
2  typedef struct node {
3      T value;
4      struct node* prev;
5      struct node* next;
6  } Node, *TList;
```

- Adăugare la începutul listei – Complexitatea: $O(1)$
- Adăugare la finalul listei – Complexitatea: $O(N)$
- Ștergerea de la începutul listei – Complexitatea: $O(1)$
- Ștergerea de la finalul listei – Complexitatea $O(N)$


Putem optimiza cumva operațiile care nu au complexitatea $O(1)$?

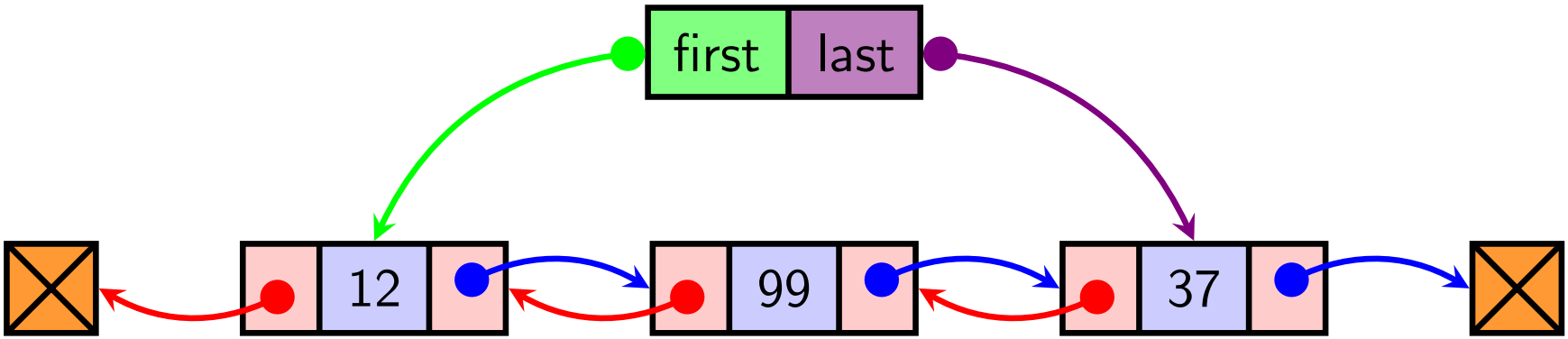
# Liste dublu înlănțuite – Varianta optimizată

Reținem un pointer la ultimul nod din listă!



# Liste dublu înlănțuite – Varianta optimizată

Reținem un pointer la ultimul nod din listă!




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

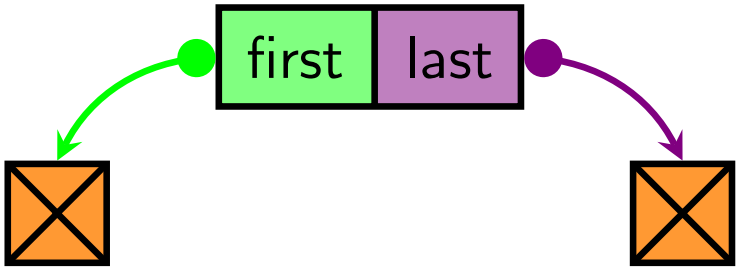
```
1  typedef struct list {
2    Node* first;
3    Node* last;
4  } List;
```

# Liste dublu înlănțuite – Inițializare

```
1  Node *createNode(T value) {
2      Node *node = malloc(sizeof(struct node));
3      node->value = value;
4      node->prev = node->next = NULL;
5      return node;
6  }
7  List *initList() {
8      List *list = malloc(sizeof(struct list));
9      list->first = list->last = NULL;
10     return list;
11 }
12 List *createList(T value) {
13     List *list = malloc(sizeof(struct list));
14     Node *head = createNode(value);
15     list->first = list->last = head;
16     return list;
17 }
```

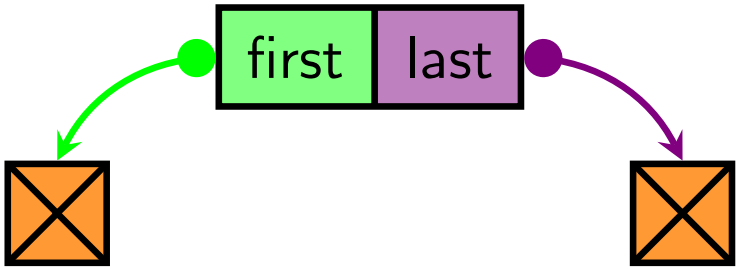
# Liste dublu înlănțuite – Inițializare

1. Lista vidă

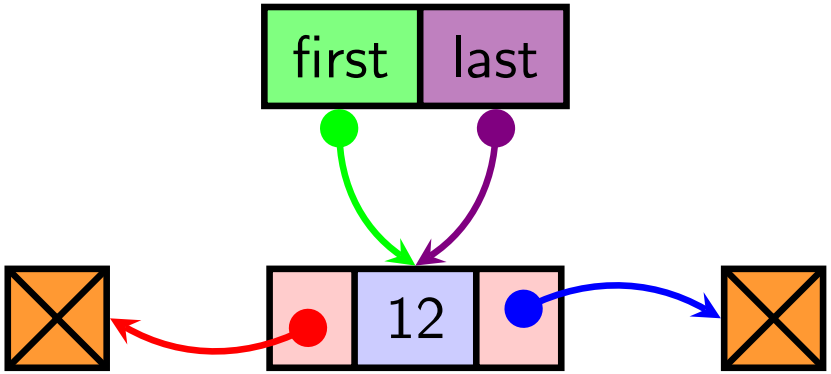


# Liste dublu înlănțuite – Inițializare

1. Lista vidă

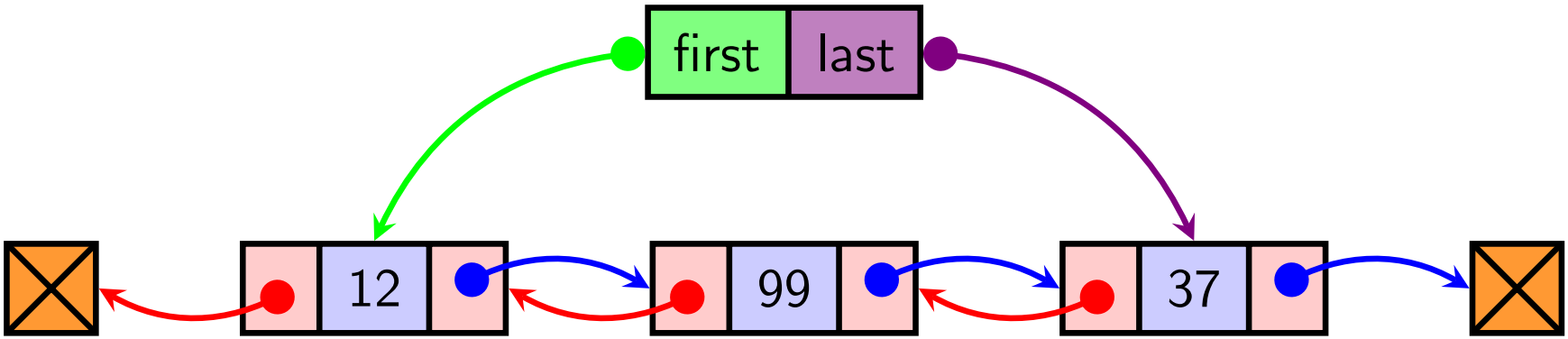



2. Lista cu un singur nod



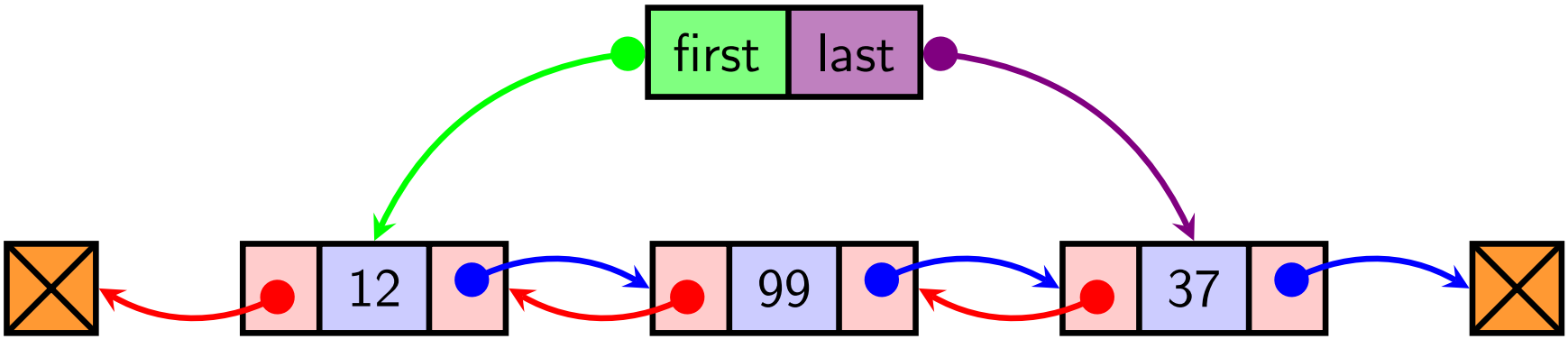
# Liste dublu înlănțuite – Adăugare la început

- Vrem să adăugam valoarea 10 la începutul listei:

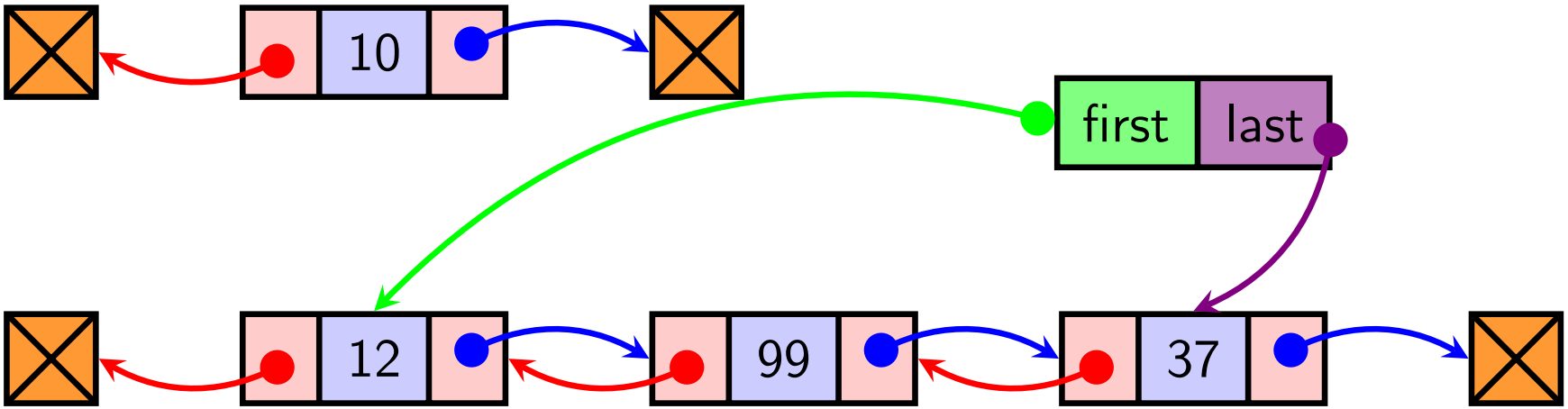


# Liste dublu înlănțuite – Adăugare la început

- Vrem să adăugam valoarea 10 la începutul listei:

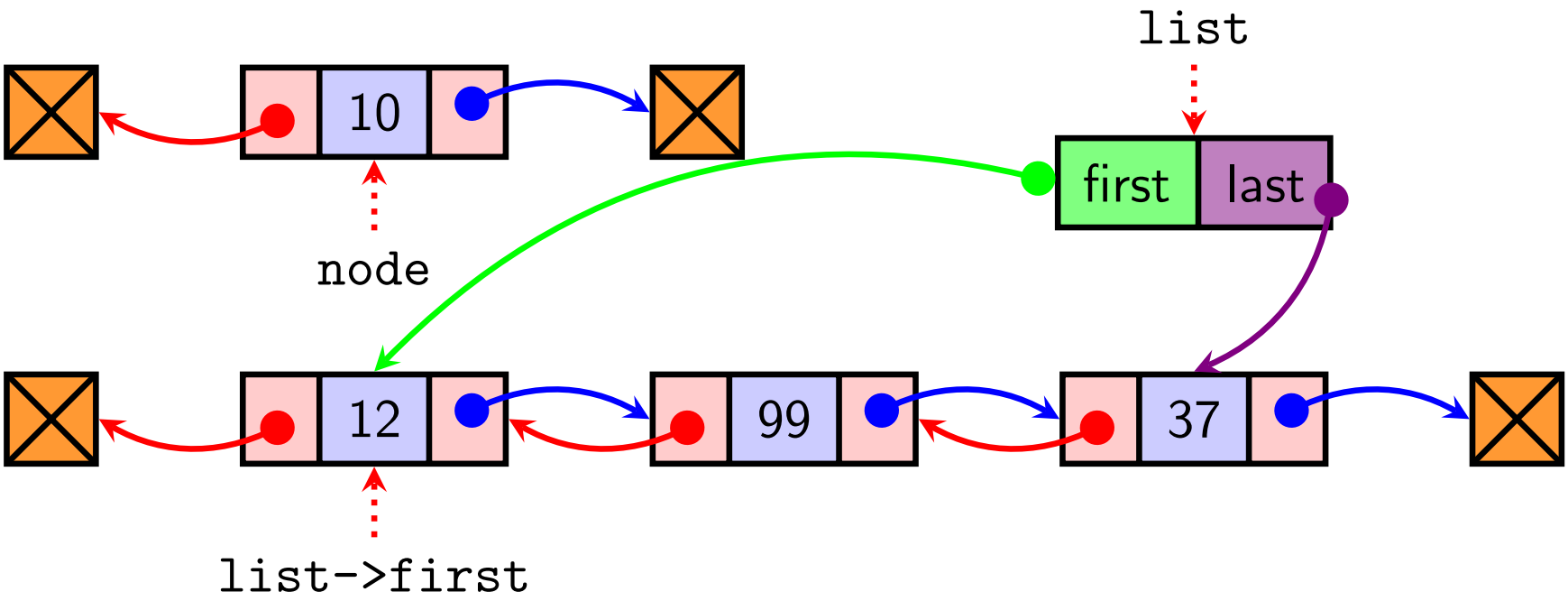



- Construim un nod ce conține valoarea 10



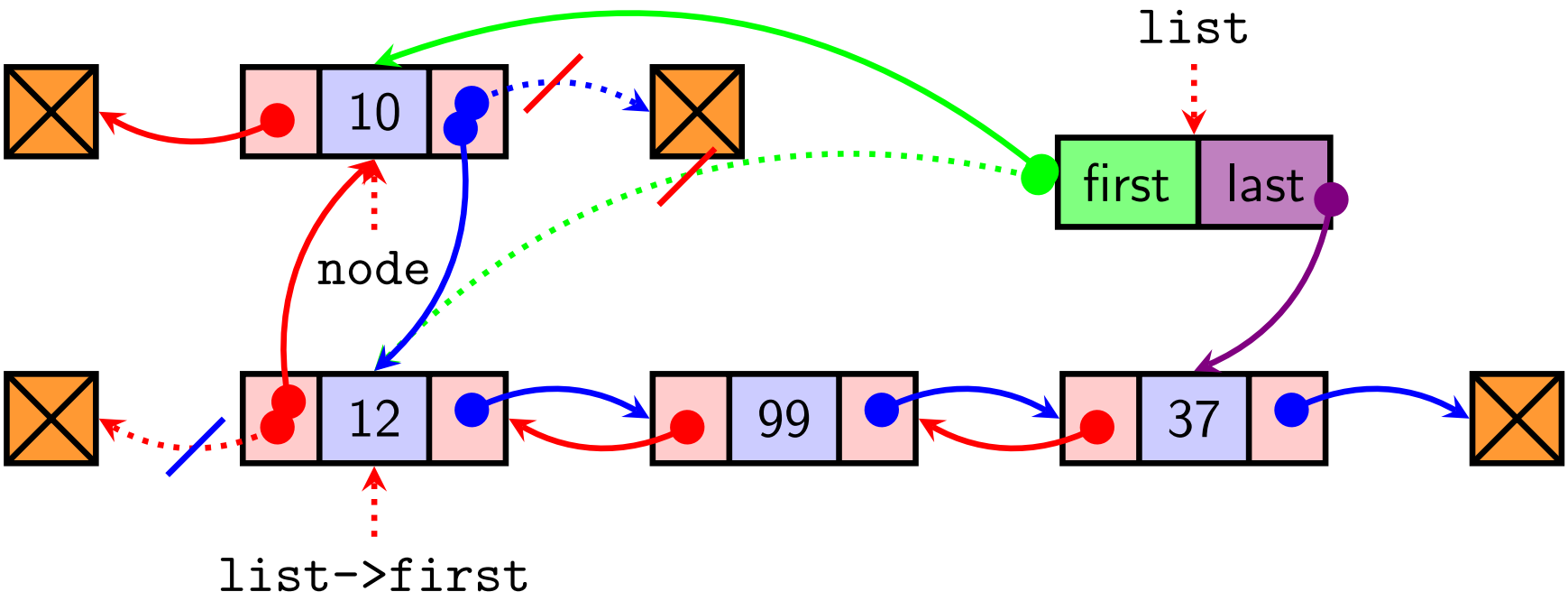
# Liste dublu înlănțuite – Adăugare la început

Ce legături trebuie să refacem?



# Liste dublu înlănțuite – Adăugare la început

```
node->next = list->first;
list->first->prev = node;
list->first = node;
```



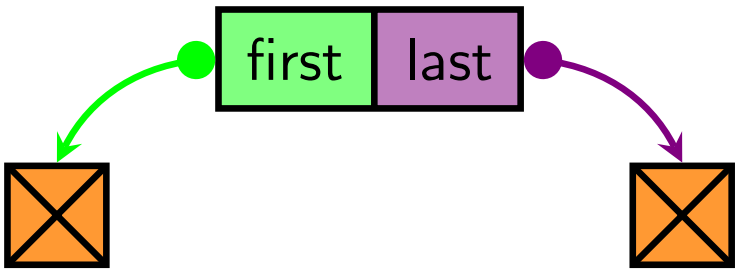
# Liste dublu înlănțuite – Adăugare la început

Există vreun caz în care operația de adăugare la începutul listei modifică câmpul `list->last`?

# Liste dublu înlănțuite – Adăugare la început

Există vreun caz în care operația de adăugare la începutul listei modifică câmpul `list->last`?

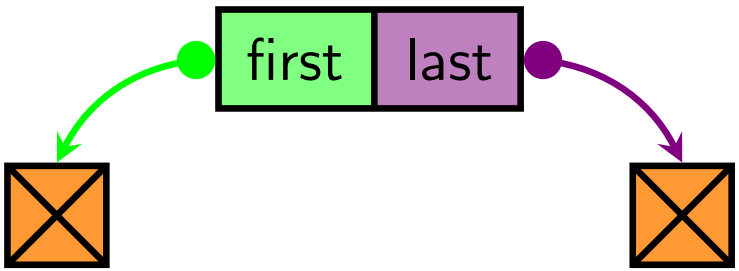
Dacă vrem să realizăm inserare în lista vidă!

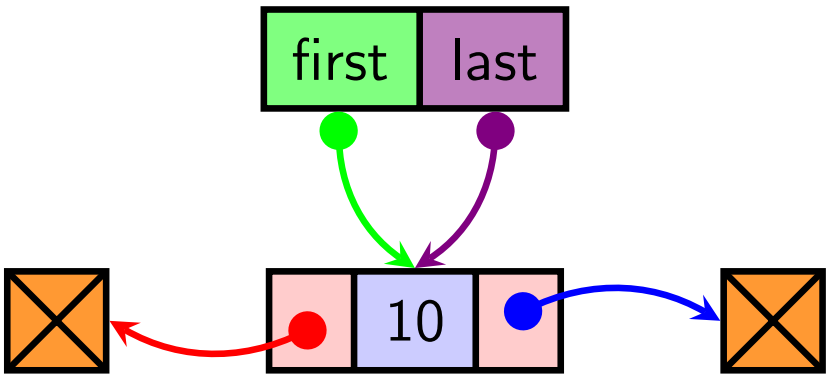


# Liste dublu înlănțuite – Adăugare la început

Există vreun caz în care operația de adăugare la începutul listei modifică câmpul `list->last`?

Dacă vrem să realizăm inserare în lista vidă!



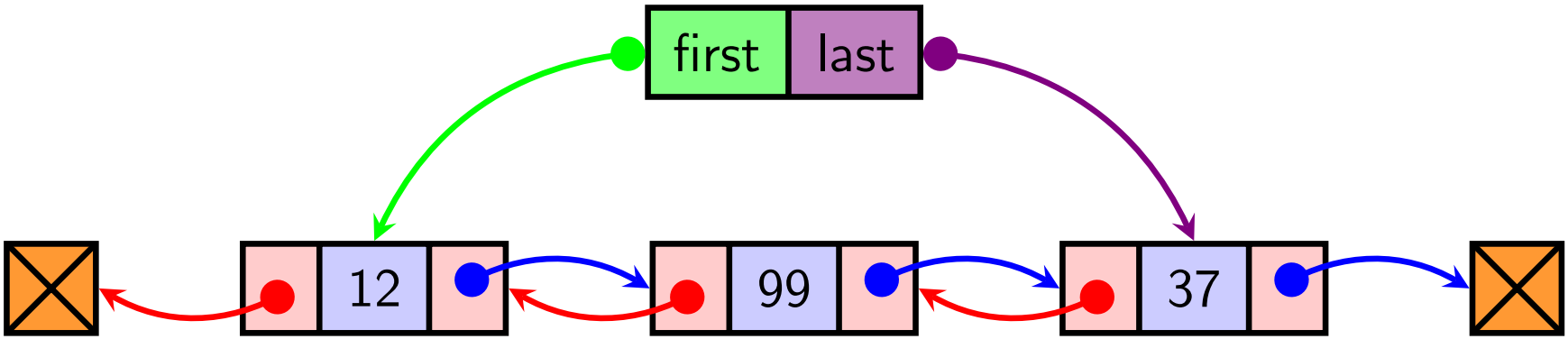



# Liste dublu înlănțuite – Adăugare la început

```
1  List *insertFront(List *list, T value) {
2      if (list == NULL)
3          return createList(value);
4      if (list->first == NULL) {
5          Node *node = createNode(value);
6          list->first = list->last = node;
7          return list;
8      }
9      Node *node = createNode(value);
10     node->next = list->first;
11     list->first->prev = node;
12     list->first = node;
13     return list;
14 }
```

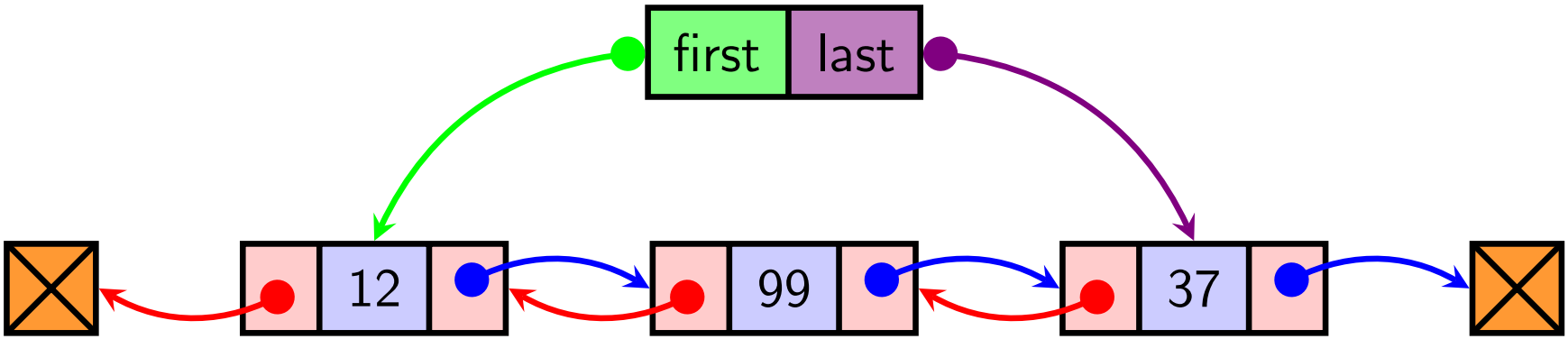
# Liste dublu înlănțuite – Adăugare la final

- Vrem să adăugam valoarea 10 la finalul listei:

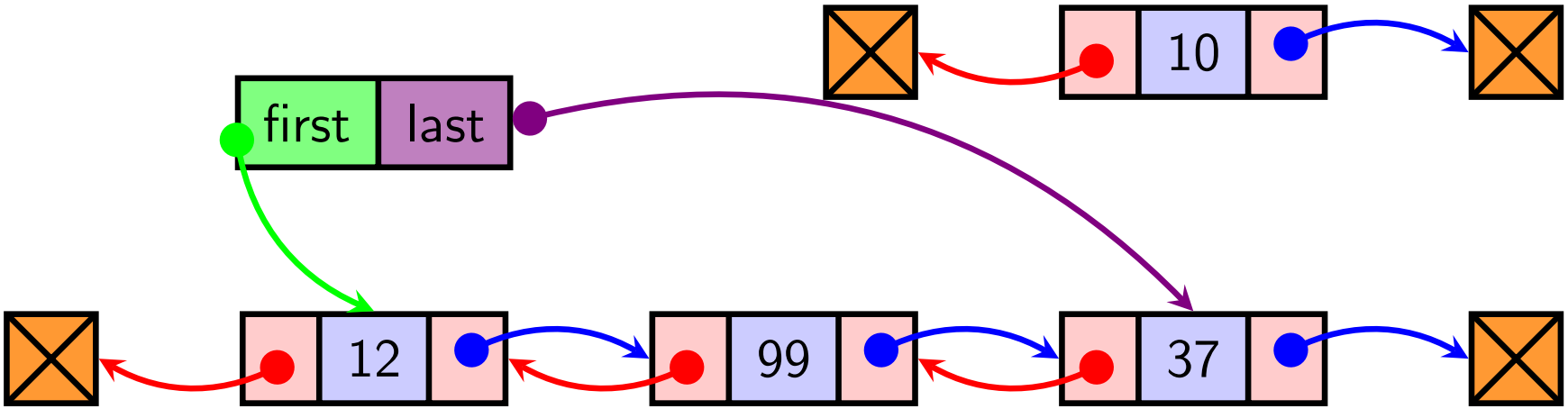


# Liste dublu înlănțuite – Adăugare la final

- Vrem să adăugam valoarea 10 la finalul listei:

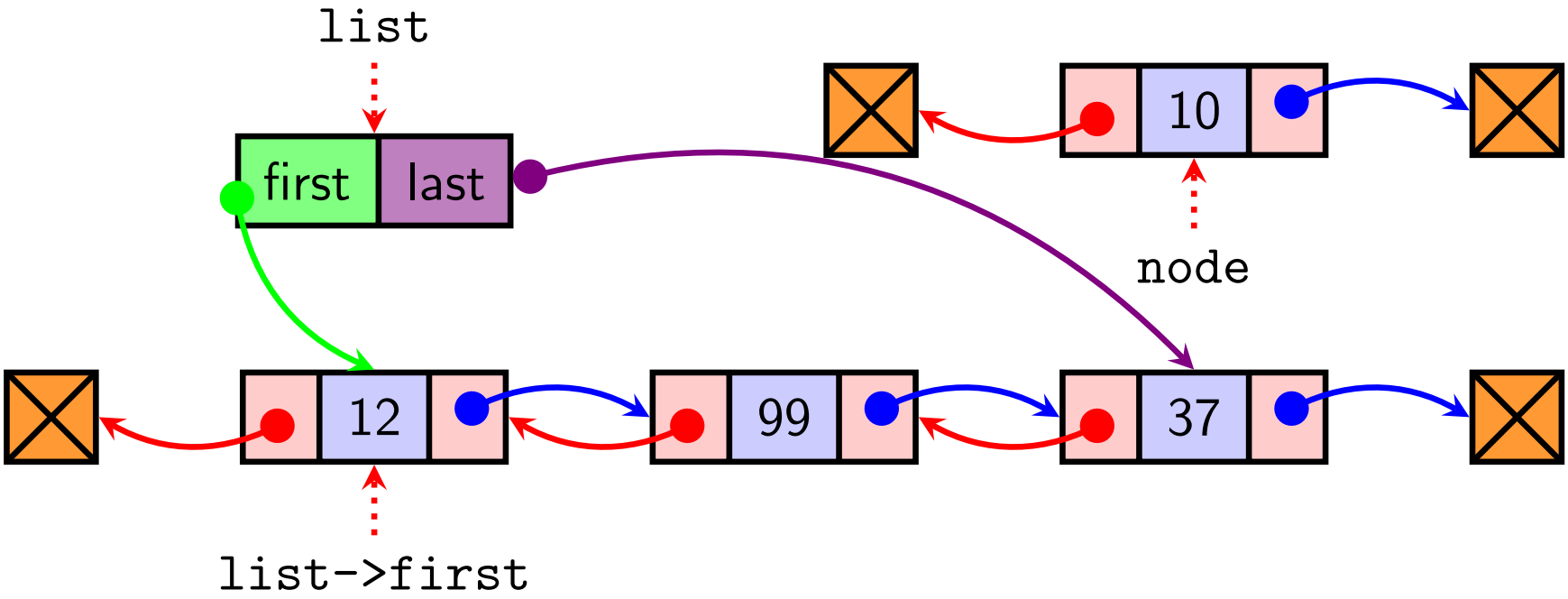



- Construim un nod ce conține valoarea 10



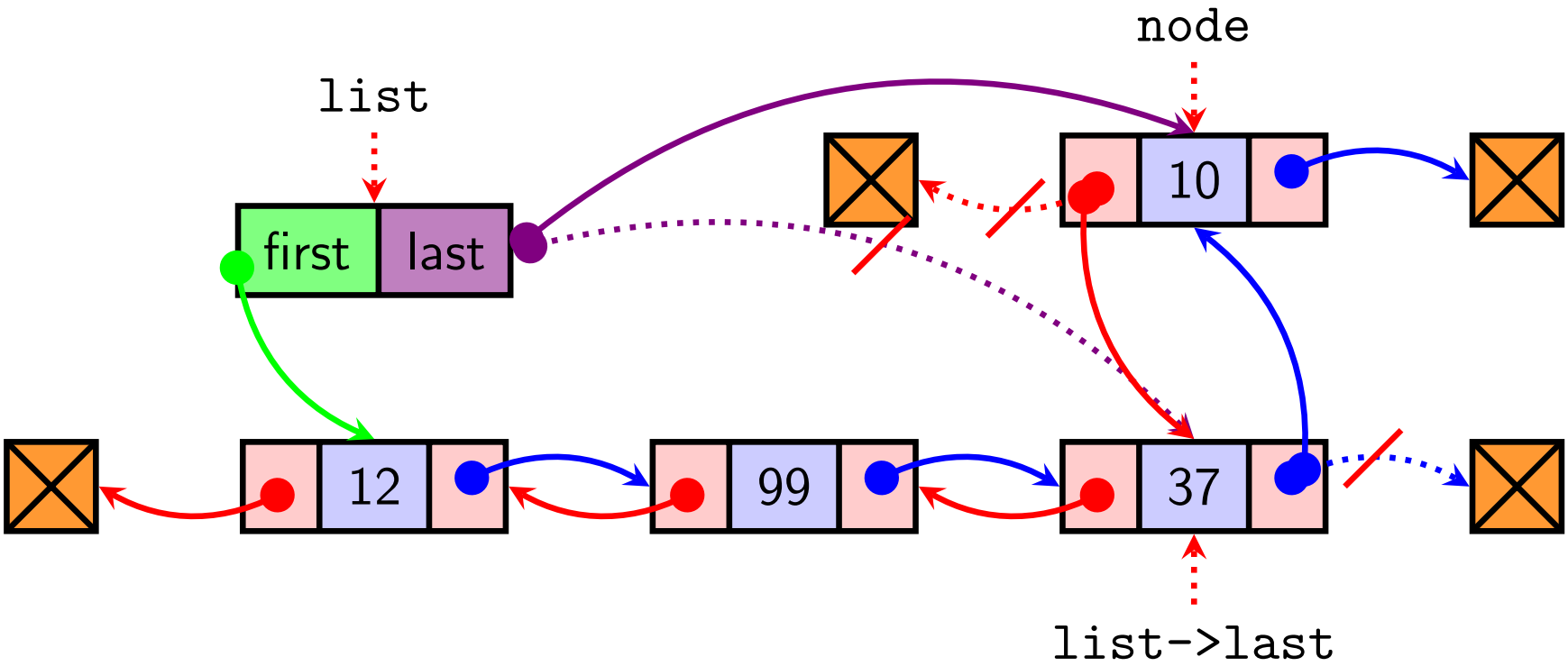
# Liste dublu înlănțuite – Adăugare la final

Ce legături trebuie să refacem?



# Liste dublu înlănțuite – Adăugare la final

```
node->prev = list->last;
list->last->next = node;
list->last = node;
```



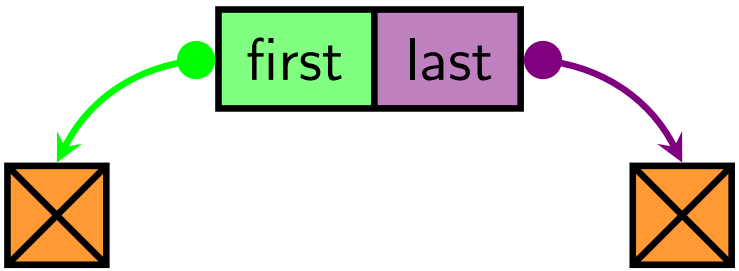
# Liste dublu înlănțuite – Adăugare la final

Există vreun caz în care operația de adăugare la finalul listei modifică câmpul `list->first`?

# Liste dublu înlănțuite – Adăugare la final

Există vreun caz în care operația de adăugare la finalul listei modifică câmpul `list->first`?

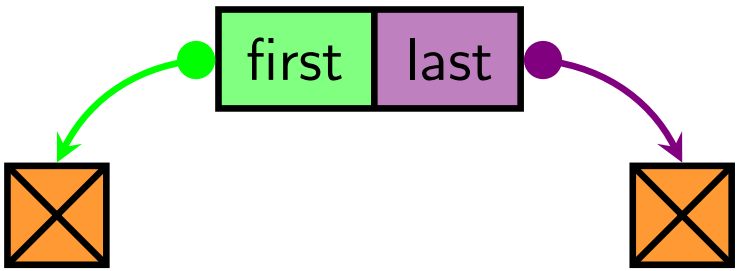
Dacă vrem să realizăm inserare în lista vidă!

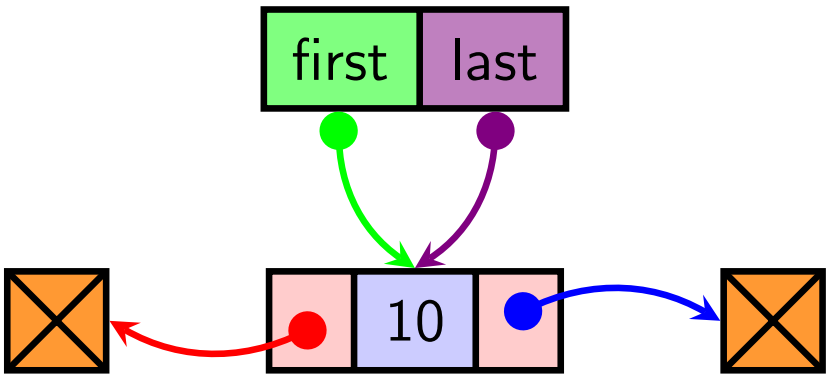


# Liste dublu înlănțuite – Adăugare la final

Există vreun caz în care operația de adăugare la finalul listei modifică câmpul `list->first`?

Dacă vrem să realizăm inserare în lista vidă!



# Liste dublu înlănțuite – Adăugare la final

```
1  List *insertRear(List *list, T value) {
2      if (list == NULL)
3          return createList(value);
4      if (list->first == NULL) {
5          Node *node = createNode(value);
6          list->first = list->last = node;
7          return list;
8      }
9      Node *node = createNode(value);
10     list->last->next = node;
11     node->prev = list->last;
12     list->last = node;
13     return list;
14 }
```

# Liste dublu înlănțuite – Adăugare la final

```
1  List *insertRear(List *list, T value) {
2      if (list == NULL)
3          return createList(value);
4      if (list->first == NULL) {
5          Node *node = createNode(value);
6          list->first = list->last = node;
7          return list;
8      }
9      Node *node = createNode(value);
10     list->last->next = node;
11     node->prev = list->last;
12     list->last = node;
13     return list;
14 }
```

Ce complexitate are cum operația de adăugare la final?

# Liste dublu înlănțuite – Ștergere început

```
1  List *removeFront(List *list) {
2      Node *temp;
3      if (list == NULL || list->first == NULL)
4          return list;
5      if (list->first == list->last) {
6          temp = list->first;
7          list->first = list->last = NULL;
8          free(temp);
9          return list;
10     }
11     temp = list->first;
12     list->first = list->first->next;
13     list->first->prev = NULL;
14     free(temp);
15     return list;
16 }
```

# Liste dublu înlănțuite – Ștergere final

```
1  List *removeRear(List *list) {
2      Node *temp;
3      if (list == NULL || list->first == NULL)
4          return list;
5      if (list->first == list->last) {
6          temp = list->first;
7          list->first = list->last = NULL;
8          free(temp);
9          return list;
10     }
11     temp = list->last;
12     list->last = list->last->prev;
13     list->last->next = NULL;
14     free(temp);
15     return list;
16 }
```

# Liste dublu înlănțuite – Afișarea listei

```
1  void print(List *list) {
2      if (list == NULL) {
3          printf("NULL\n");
4          return;
5      }
6      Node *iter = list->first;
7      while (iter != NULL) {
8          printf("%d ", iter->value);
9          iter = iter->next;
10     }
11     printf("\n");
12 }
```

# Liste dublu înlănțuite – Afișarea listei în ordine inversă

```
1  void printReverse(List *list) {
2      if (list == NULL)
3          return;
4      Node *iter = list->last;
5      while (iter != NULL) {
6          printf("%d ", iter->value);
7          iter = iter->prev;
8      }
9      printf("\n");
10 }
```

# Liste dublu înlănțuite – Dealocarea memoriei

```
1  List *freeList(List *list) {
2      if (list == NULL)
3          return list;
4      Node *iter, *temp;
5      iter = list->first;
6      while (iter != NULL) {
7          temp = iter;
8          iter = iter->next;
9          free(temp);
10     }
11     free(list);
12     return NULL;
13 }
```

# Liste dublu înlănțuite – Dealocarea memoriei

```
1  List *freeList(List *list) {
2      if (list == NULL)
3          return list;
4      Node *iter, *temp;
5      iter = list->first;
6      while (iter != NULL) {
7          temp = iter;
8          iter = iter->next;
9          free(temp);
10     }
11     free(list);
12     return NULL;
13 }
```


De ce este nevoie de operația `free(list)`?

# *Vă mulțumesc pentru atenție!*