Programozás 2

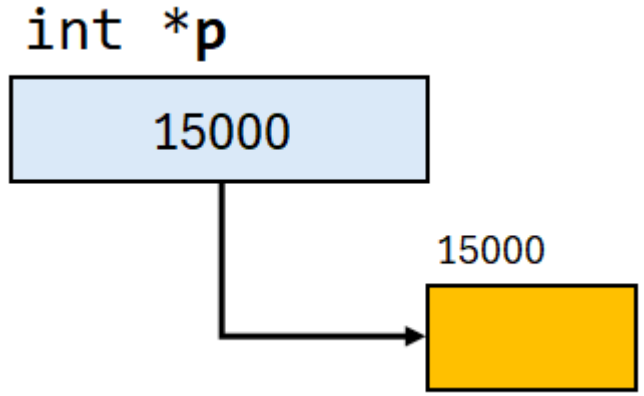
Dinamikus adattípusok és struktúrák. Egyirányú láncolt lista, verem és sor megvalósítása. Ciklikus láncolt lista, kétirányú láncolt lista. Bináris keresőfa. Dinamikus irányított súlyozott gráf.

Dinamikus memóriafoglalás

A dinamikus memóriafoglalás lehetővé teszi, hogy **futásidőben hozzunk létre változókat**. Ez különösen akkor hasznos, ha nem tudjuk előre, mennyi adatot kell tárolnunk.

- Memóriafoglalás:
  void *malloc(size_t size)

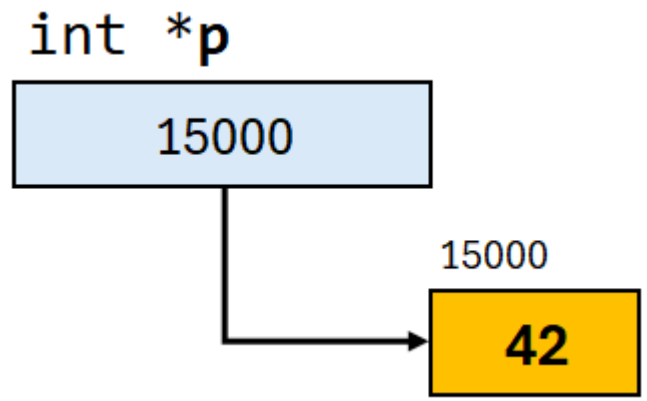
Pl.:
```
int *p = (int*) malloc(sizeof(int));
if (p == NULL) {
    printf("Memoriafoglalas sikertelen\n");
    return 1;
}
```



- Memória felszabadítása:
  void free(void *ptr)

Pl.:
```
free(p);
```

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    // memoriafoglalas int tipusu valtozonak
    int *p = (int*) malloc(sizeof(int));
    if (p == NULL) {
        printf("Memoriafoglalas sikertelen\n");
        return 1;
    }
    // valtozo ertekenek megadasa
    *p = 42;
    // valtozo kiirasa
    printf("A valtozo erteke: %d\n", *p);
    // memoria felszabaditasa
    free(p);
}
```



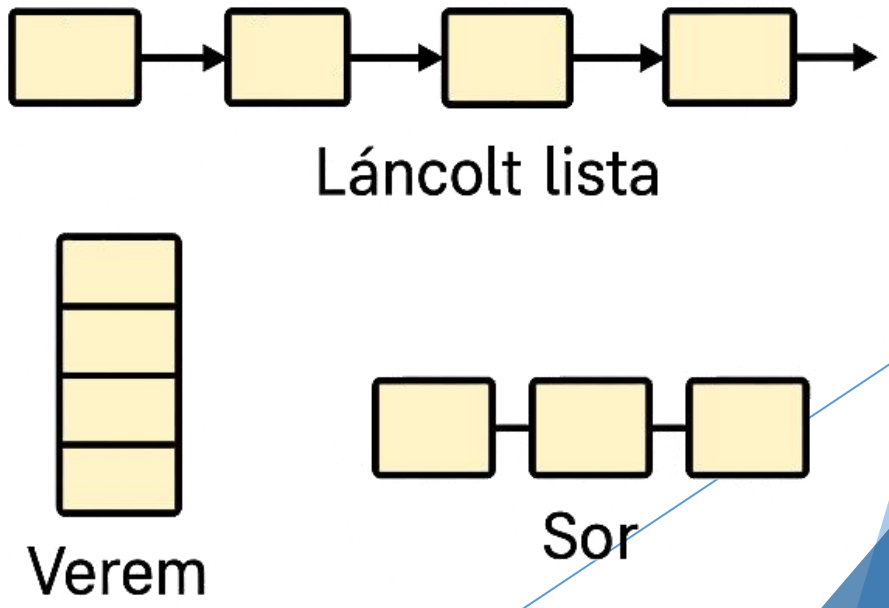
Dinamikus adatszerkezetek

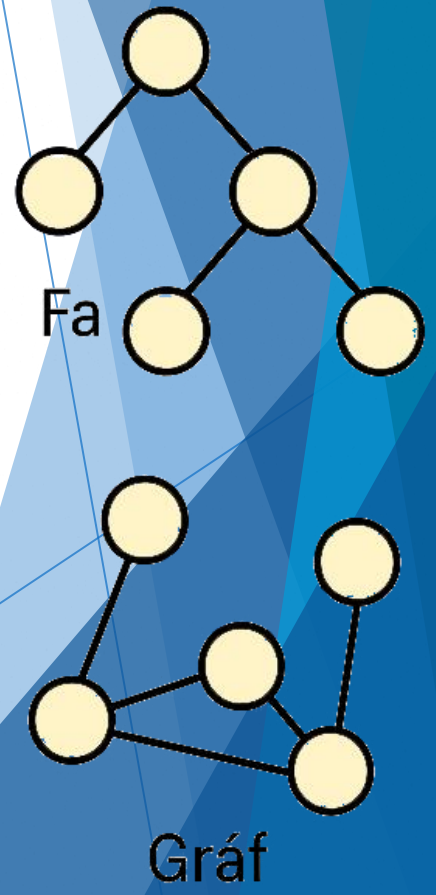
A **dinamikus adatszerkezetek** olyan adattárolási formák, amelyek **mérete futás közben változhat** – azaz az elemek számát nem kell előre meghatározni.

A dinamikus adatszerkezetek jellemzően **mutatók segítségével** valósulnak meg, és lehetővé teszik az elemek rugalmas hozzáadását és törlését.

Pl.:

- Láncolt lista (linked list)
- Verem (stack)
- Sor (queue)
- Fa (tree)
- Gráf (graph)

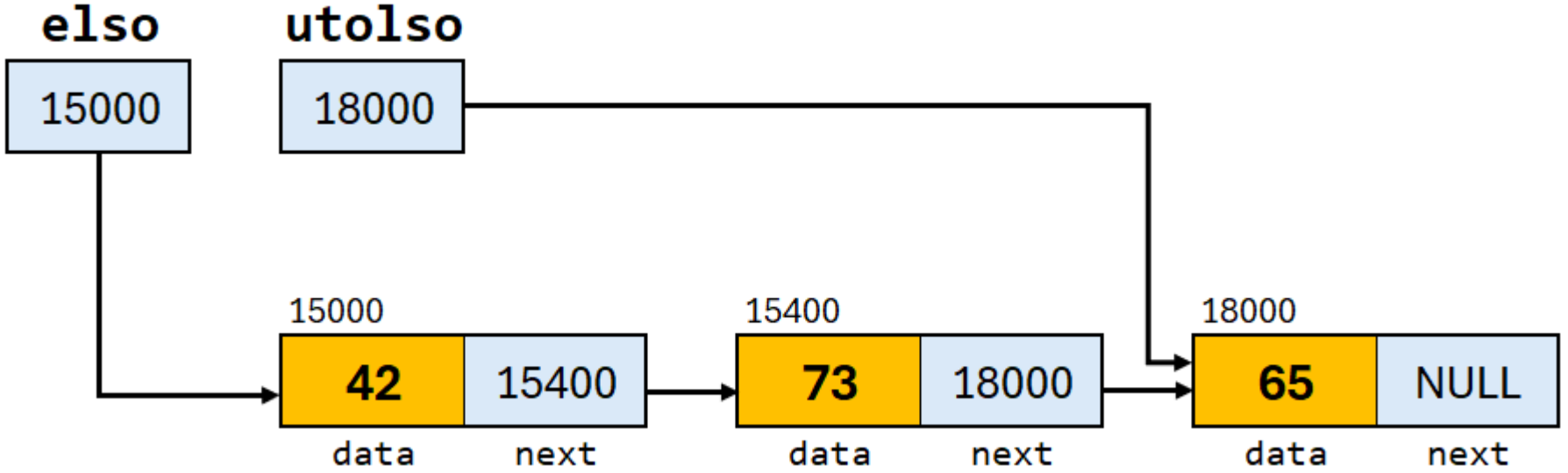




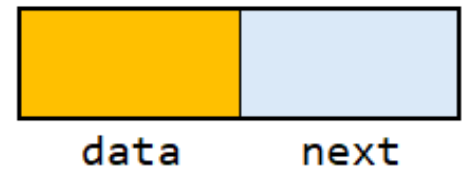
Egyirányú láncolt lista

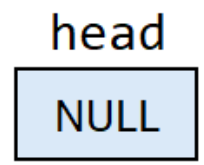
Az **egyirányú láncolt lista** egy dinamikus adatszerkezet, amelyben az elemek (csomópontok) sorrendben kapcsolódnak egymáshoz, és mindegyik csak a következő elemre mutató hivatkozást (mutatót) tartalmaz.

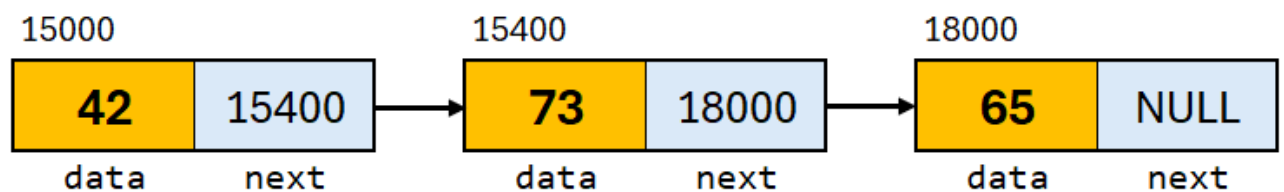
Előnye, hogy könnyen bővíthető vagy módosítható anélkül, hogy a teljes szerkezetet át kellene rendezni.



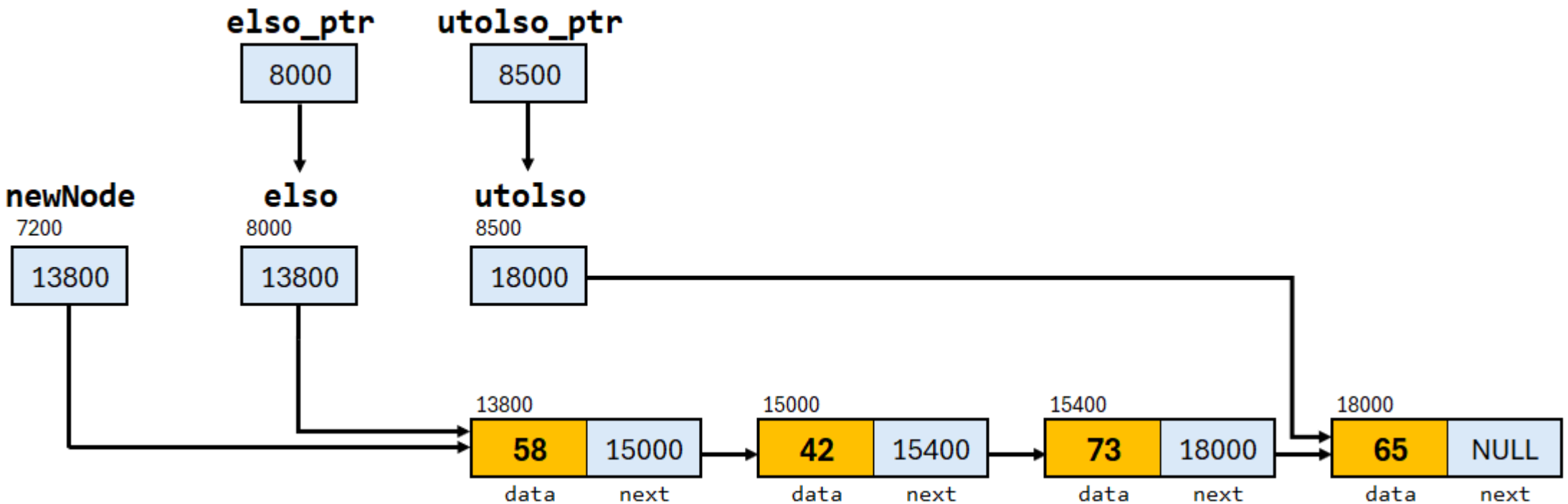
```c
#include <stdio.h>
#include <stdlib.h>

// csomopont tipus (Node) definialasa
typedef struct Node {
    int data;             // tarolt adat (egesz szam)
    struct Node* next;    // mutato a kovetkezo Node-ra
} Node;

// a lancolt lista kiirasa
void printList(Node* head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}
```



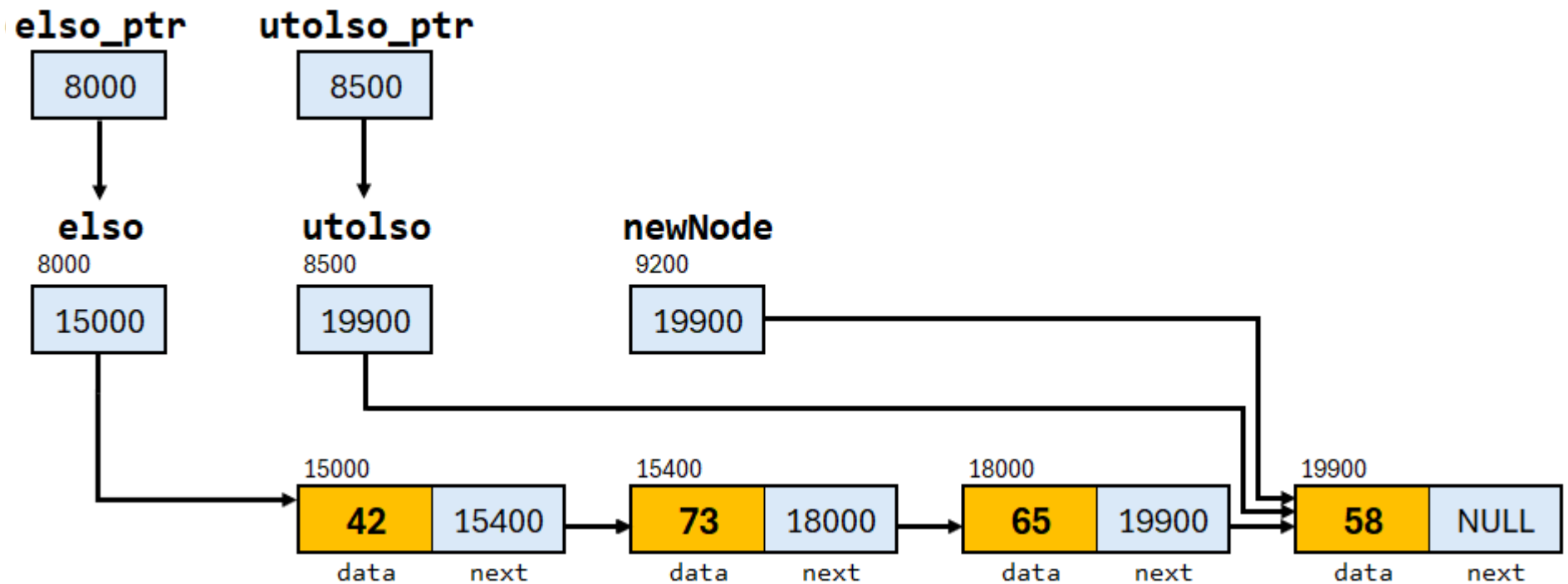




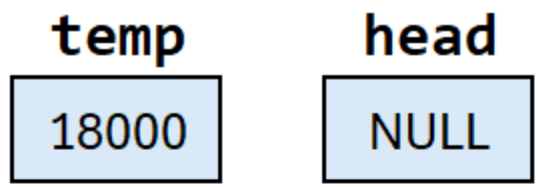
```
// lista elejere szuras
void addFirst(Node** elso_ptr, Node** utolso_ptr, int value) {
    Node* newNode = (Node*) malloc(sizeof(Node));
    newNode->data = value;
    newNode->next = *elso_ptr;
    *elso_ptr = newNode;
    if (*utolso_ptr == NULL) {
        *utolso_ptr = newNode;
    }
}
```



```c
// lista vegere szuras
void addLast(Node** elso_ptr, Node** utolso_ptr, int value) {
    Node* newNode = (Node*) malloc(sizeof(Node));
    newNode->data = value;
    newNode->next = NULL;
    if (*elso_ptr == NULL) {
        *elso_ptr = *utolso_ptr = newNode;
    } else {
        (*utolso_ptr)->next = newNode;
        *utolso_ptr = newNode;
    }
}
```



```c
// memoria felszabaditasa
void freeList(Node* head) {
    while (head != NULL) {
        Node* temp = head;
        head = head->next;
        free(temp);
    }
}
```

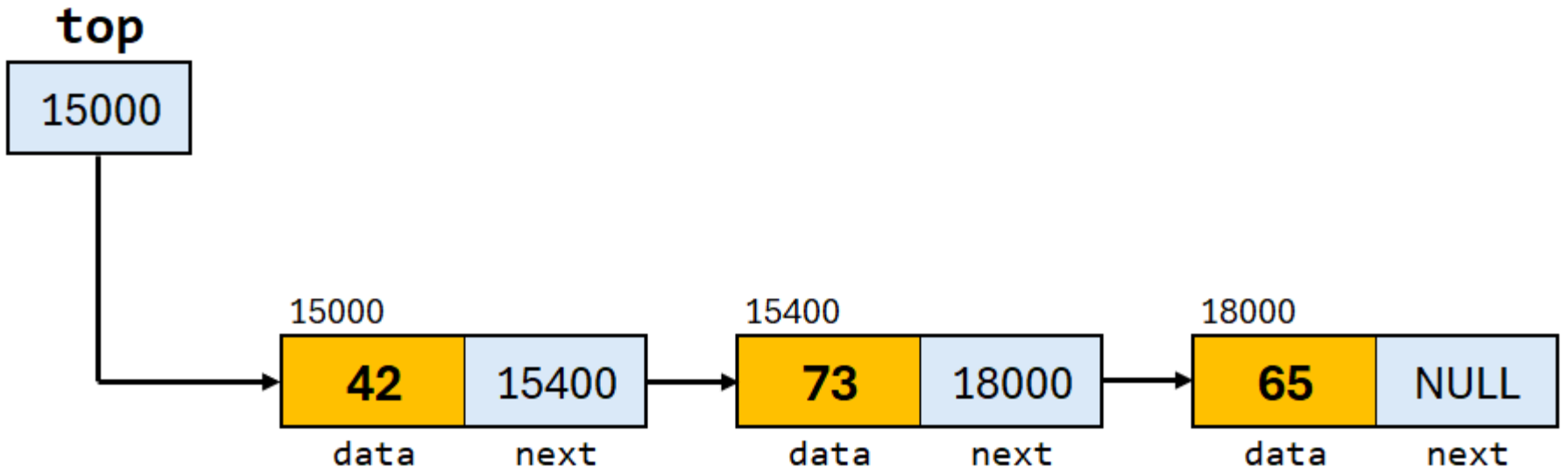


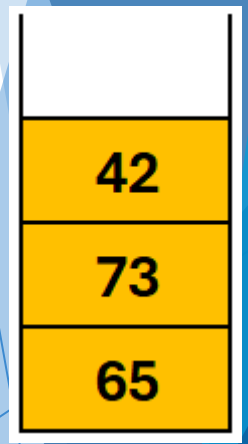
```c
int main() {
    // pointerek az elso es utolso elemre
    Node* elso = NULL;
    Node* utolso = NULL;
    // elejere beszuras
    addFirst(&elso, &utolso, 3);  // 3
    addFirst(&elso, &utolso, 2);  // 2 -> 3
    addFirst(&elso, &utolso, 1);  // 1 -> 2 -> 3
    // vegere beszuras
    addLast(&elso, &utolso, 4);   // 1 -> 2 -> 3 -> 4
    addLast(&elso, &utolso, 5);   // 1 -> 2 -> 3 -> 4 -> 5
    // lista kiirasa
    printList(elso);
    // memoria felszabaditasa
    freeList(elso);
}
```

Verem megvalósítása láncolt listával

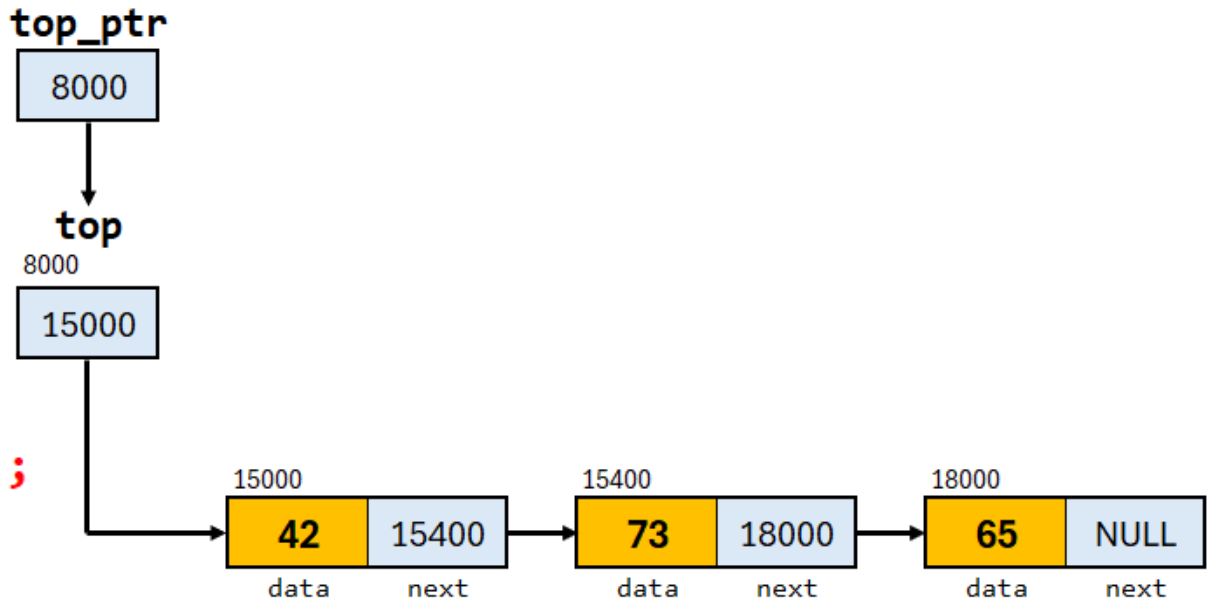
A **verem láncolt listával is megvalósítható**, ahol az új elemek mindig a lista elejére kerülnek.

Működése **LIFO** (Last In, First Out) elvű: mindig a legutóbb betett elemet vesszük ki először.





```c
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;
    struct Node* next;
} Node;

// uj elem hozzaadasa a verem tetejere
void push(Node** top_ptr, int value) {
    Node* newNode = (Node*) malloc(sizeof(Node));
    newNode->data = value;
    newNode->next = *top_ptr;
    *top_ptr = newNode;
}

// a verem tetejen levo elem kiszedese
int pop(Node** top_ptr) {
    if (*top_ptr == NULL) return -1;
    Node* temp = *top_ptr;
    int value = temp->data;
    *top_ptr = temp->next;
    free(temp);
    return value;
}
```

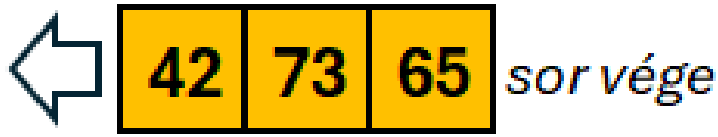


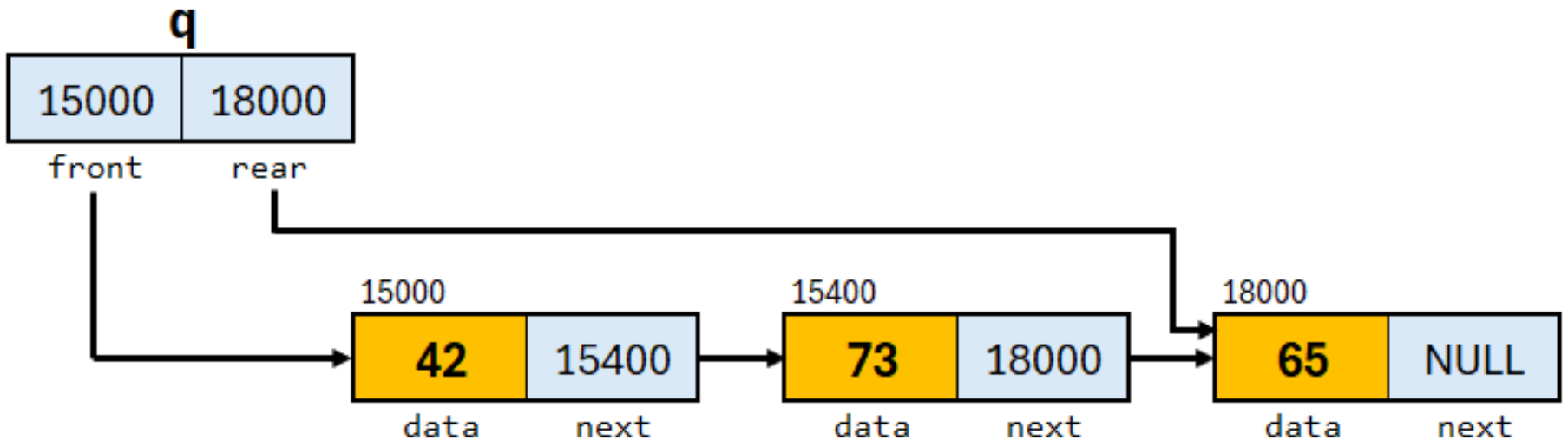

```c
int main() {
    Node* top = NULL;
    push(&top, 10);
    push(&top, 20);
    push(&top, 30);
    printf("Pop: %d\n", pop(&top));
    printf("Pop: %d\n", pop(&top));
    printf("Pop: %d\n", pop(&top));
}
```

Sor megvalósítása láncolt listával

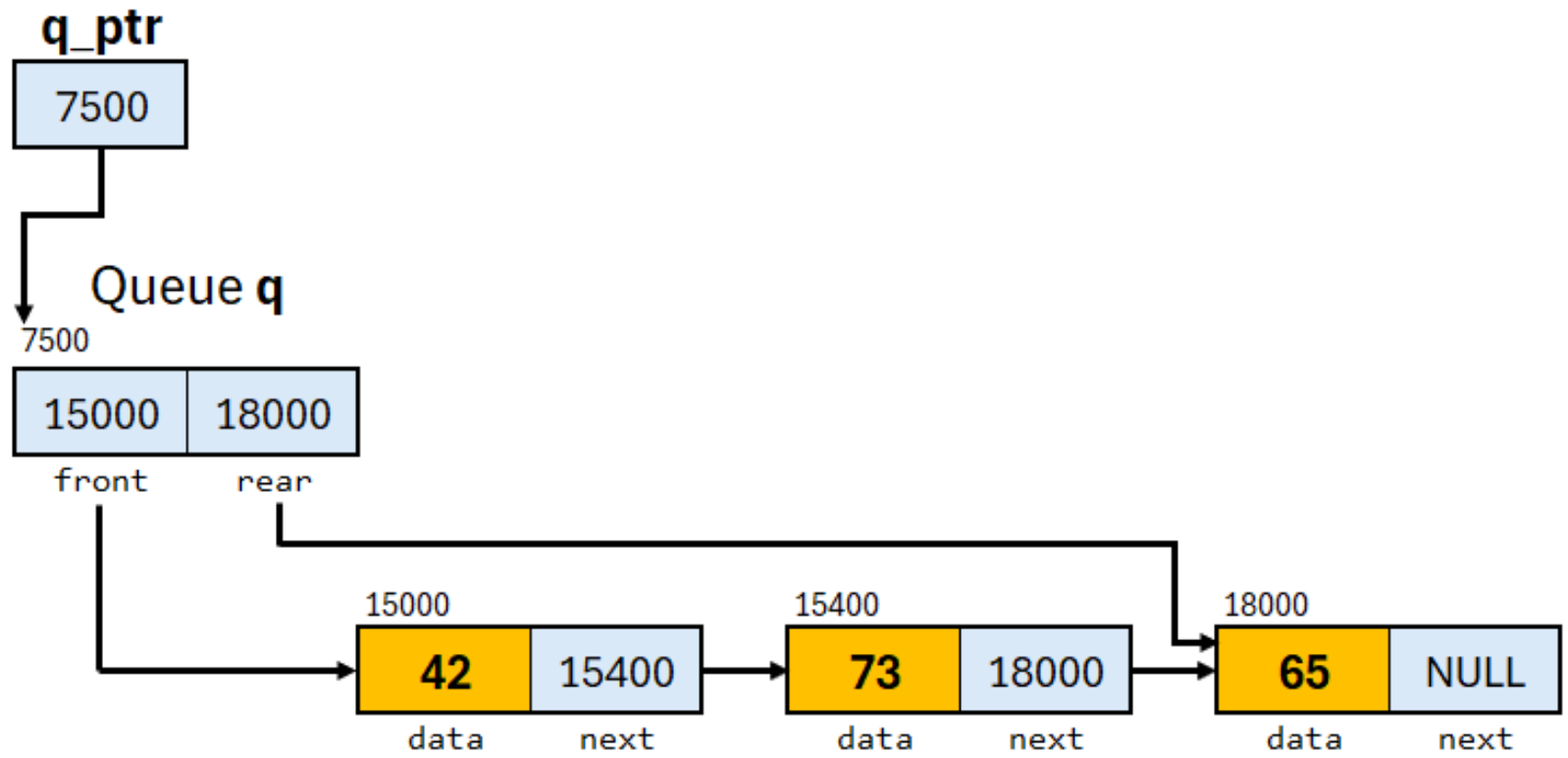
A **sor láncolt listával** úgy valósítható meg, hogy az új elemeket a lista végére fűzzük, és az eltávolítás mindig az elejéről történik.

Működése **FIFO** (First In, First Out) elvű: a legrégebben betett elem kerül ki először.



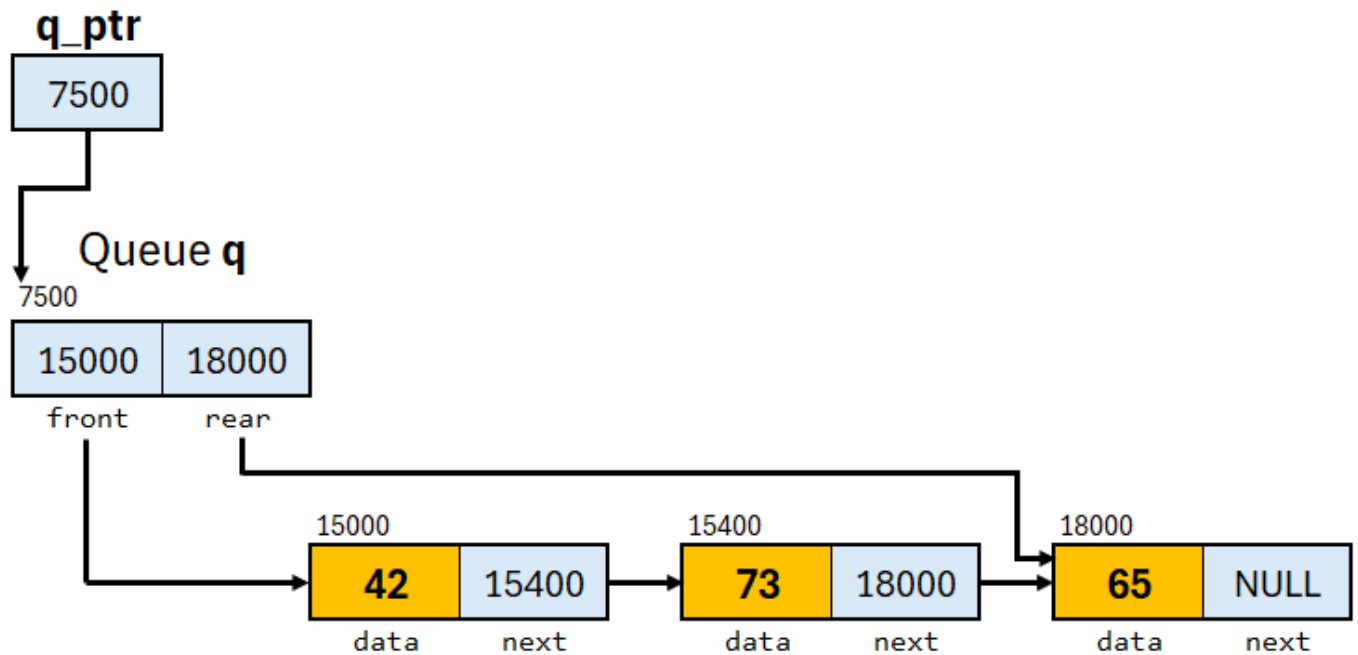


```c
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;
    struct Node* next;
} Node;

typedef struct Queue {
    Node* front;    // elso
    Node* rear;     // utolso
} Queue;
```




```c
// uj elem hozzaadasa a sor vegere
void enqueue(Queue* q_ptr, int value) {
    Node* newNode = (Node*) malloc(sizeof(Node));
    newNode->data = value;
    newNode->next = NULL;
    if (q_ptr->rear != NULL)
        { q_ptr->rear->next = newNode; }
    else
        { q_ptr->front = newNode; }
    q_ptr->rear = newNode;
}
```

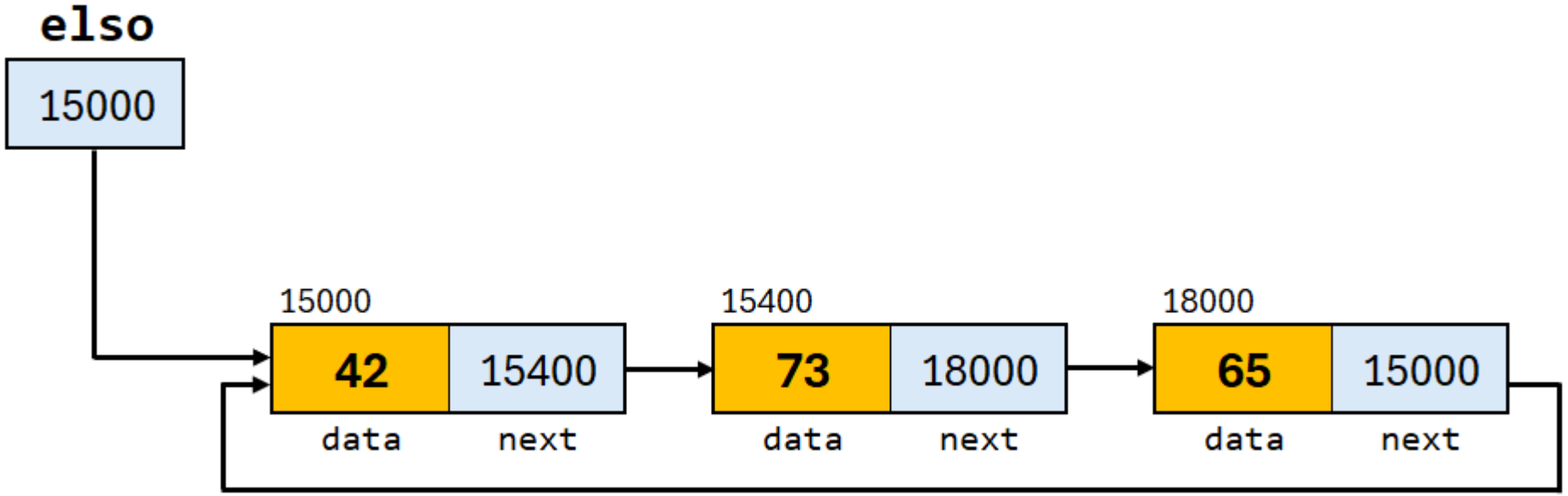
```c
// elem kiszedese a sor elejerol
int dequeue(Queue* q_ptr) {
    if (q_ptr->front == NULL) return -1;
    Node* temp = q_ptr->front;
    int value = temp->data;
    q_ptr->front = temp->next;
    if (q_ptr->front == NULL) q_ptr->rear = NULL;
    free(temp);
    return value;
}

int main() {
    Queue q = {NULL, NULL};
    enqueue(&q, 1);
    enqueue(&q, 2);
    enqueue(&q, 3);
    printf("Dequeue: %d\n", dequeue(&q));
    printf("Dequeue: %d\n", dequeue(&q));
    printf("Dequeue: %d\n", dequeue(&q));
}
```

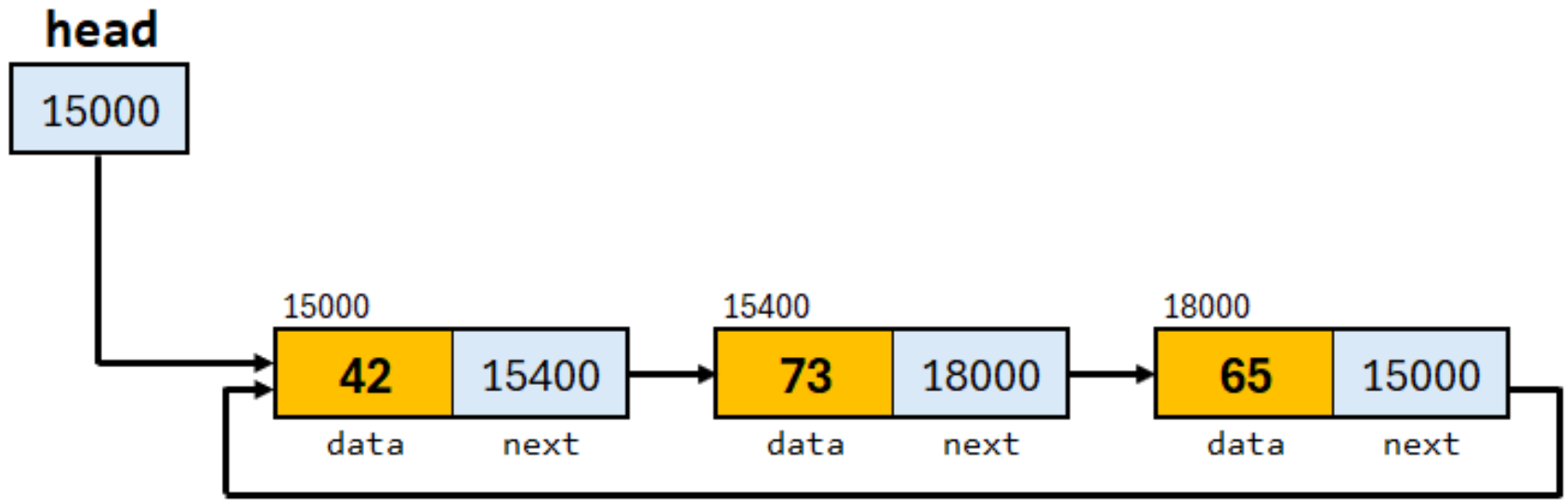


Ciklikus egyirányú láncolt lista

A **ciklikus egyirányú láncolt lista** egy olyan láncolt lista, ahol az utolsó elem nem **NULL**-ra, hanem az első elemre mutat.

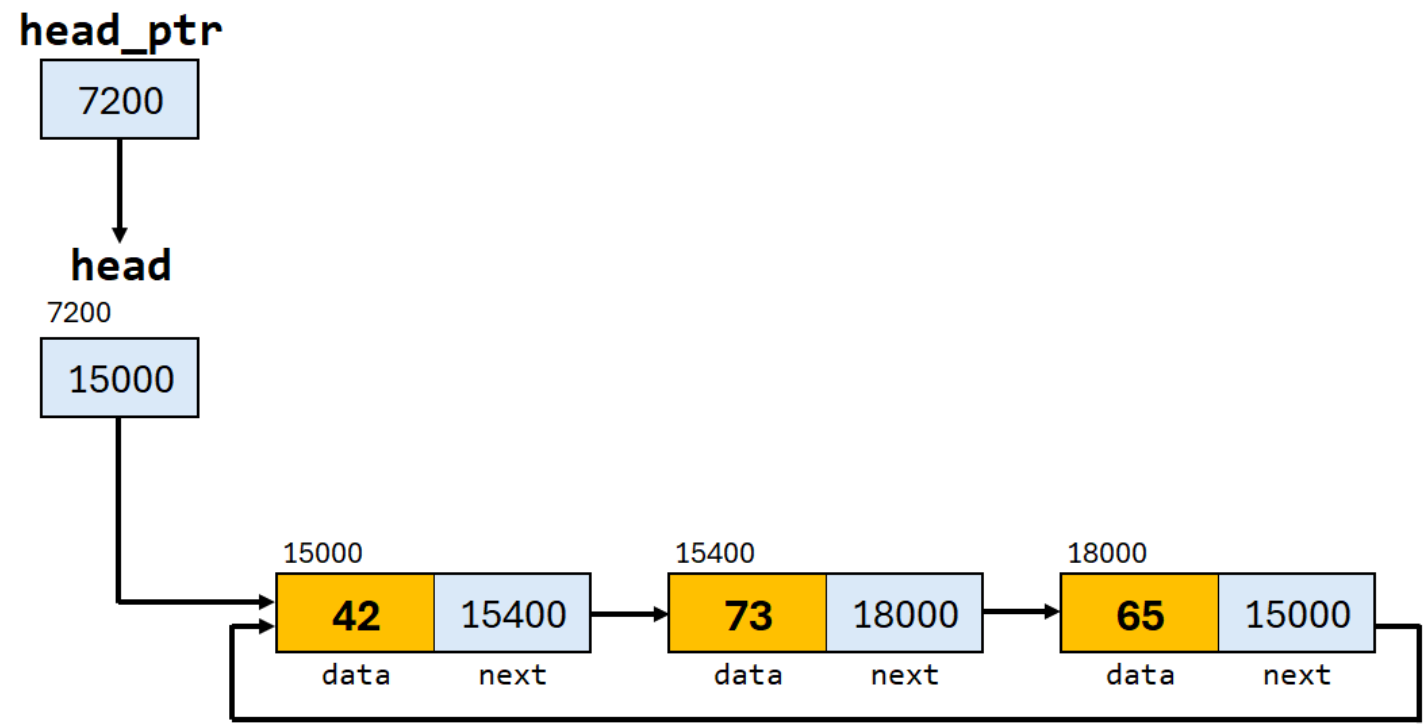
Így a lista körbejárható, ami hasznos lehet például körkörös feldolgozási feladatoknál.



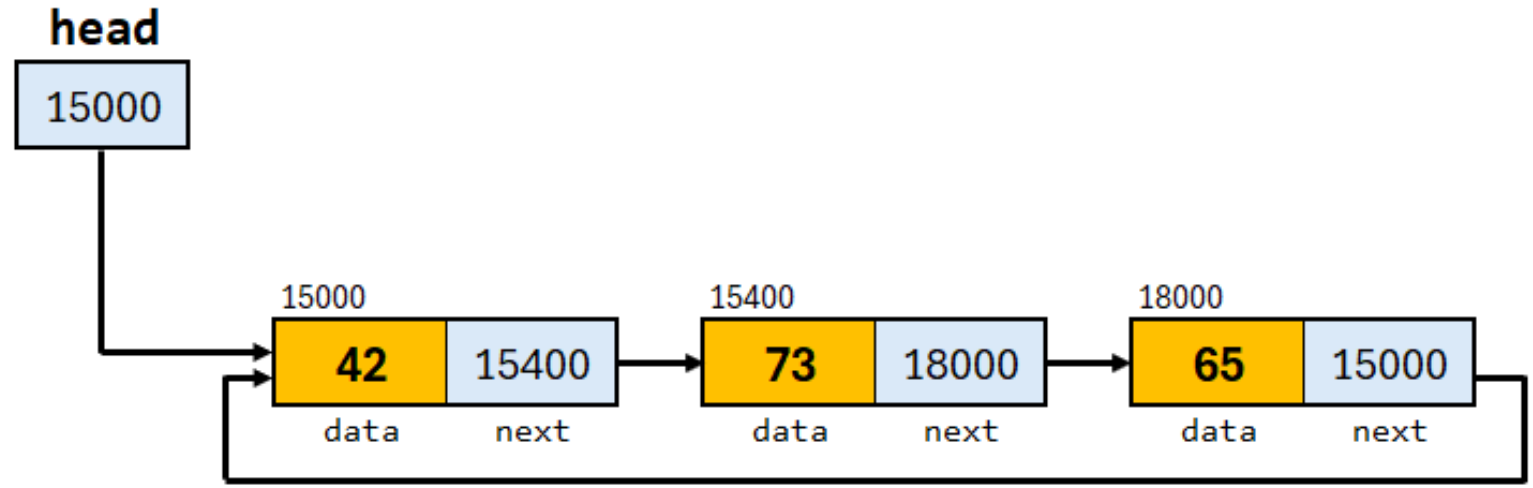


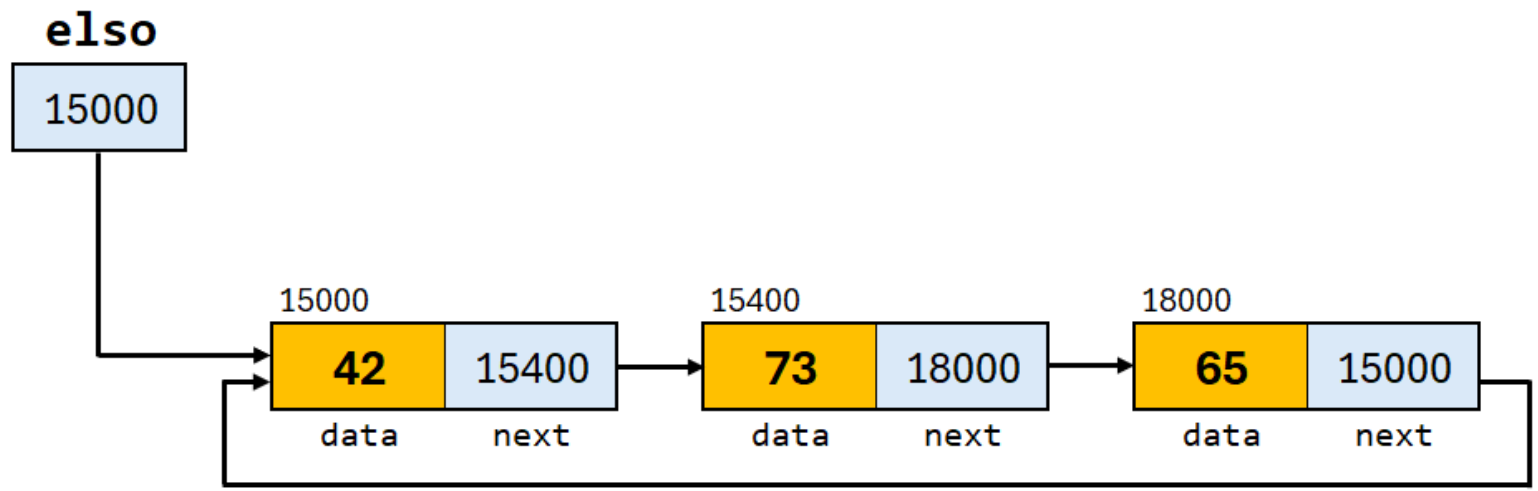

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;
    struct Node* next;
} Node;

// Lista kiiratasa
void printCircular(Node* head) {
    if (head == NULL) return;
    Node* start = head;
    do {
        printf("%d -> ", head->data);
        head = head->next;
    } while (head != start);
    printf("(vissza az elejere)\n");
}
```

```c
// elem hozzaadasa a ciklikus lista vegere
void addLast(Node** head_ptr, int value) {
    Node* newNode = (Node*) malloc(sizeof(Node));
    newNode->data = value;
    if (*head_ptr == NULL) {
        newNode->next = newNode;  // onmagara mutat
        *head_ptr = newNode;
    } else {
        Node* temp = *head_ptr;
        while (temp->next != *head_ptr) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = *head_ptr;
    }
}
```



```c
// Ciklikus lista felszabaditasa
void freeCircular(Node* head) {
    if (head == NULL) return;
    Node* current = head->next;
    while (current != head) {
        Node* temp = current;
        current = current->next;
        free(temp);
    }
    free(head);
}

int main() {
    // elso elemre mutato pointer
    Node* elso = NULL;
    // lista dinamikus bovitese
    addLast(&elso, 1);
    addLast(&elso, 2);
    addLast(&elso, 3);
    addLast(&elso, 4);
    // lista kiirasa
    printCircular(elso);
    // memoria felszabaditasa
    freeCircular(elso);
}
```

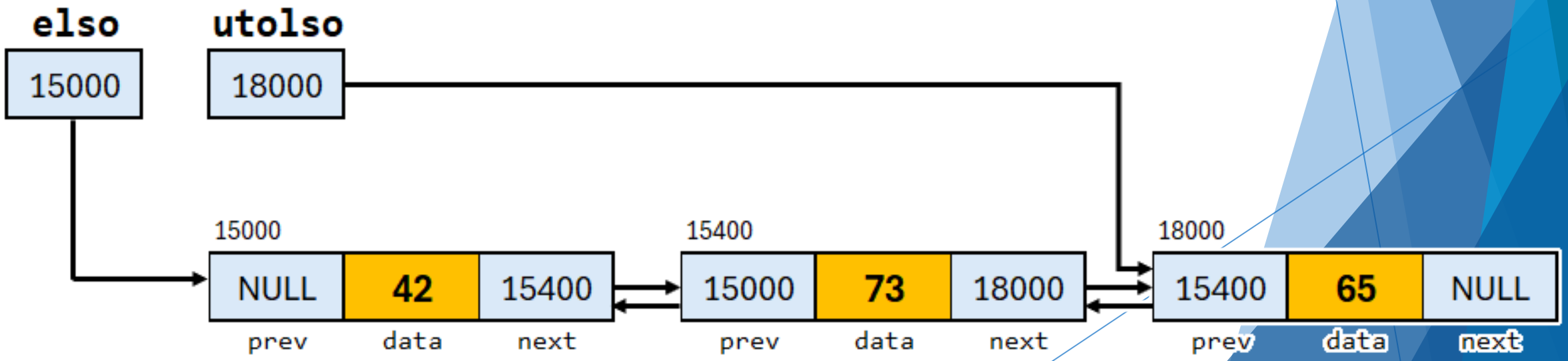




Kétirányú láncolt lista

A **kétirányú láncolt lista** minden csomópontja két hivatkozást tartalmaz: az egyik az előző, a másik a következő elemre mutat.

Ez lehetővé teszi az adatok előre és hátra történő bejárását is, rugalmasabb adatkezelést biztosítva.

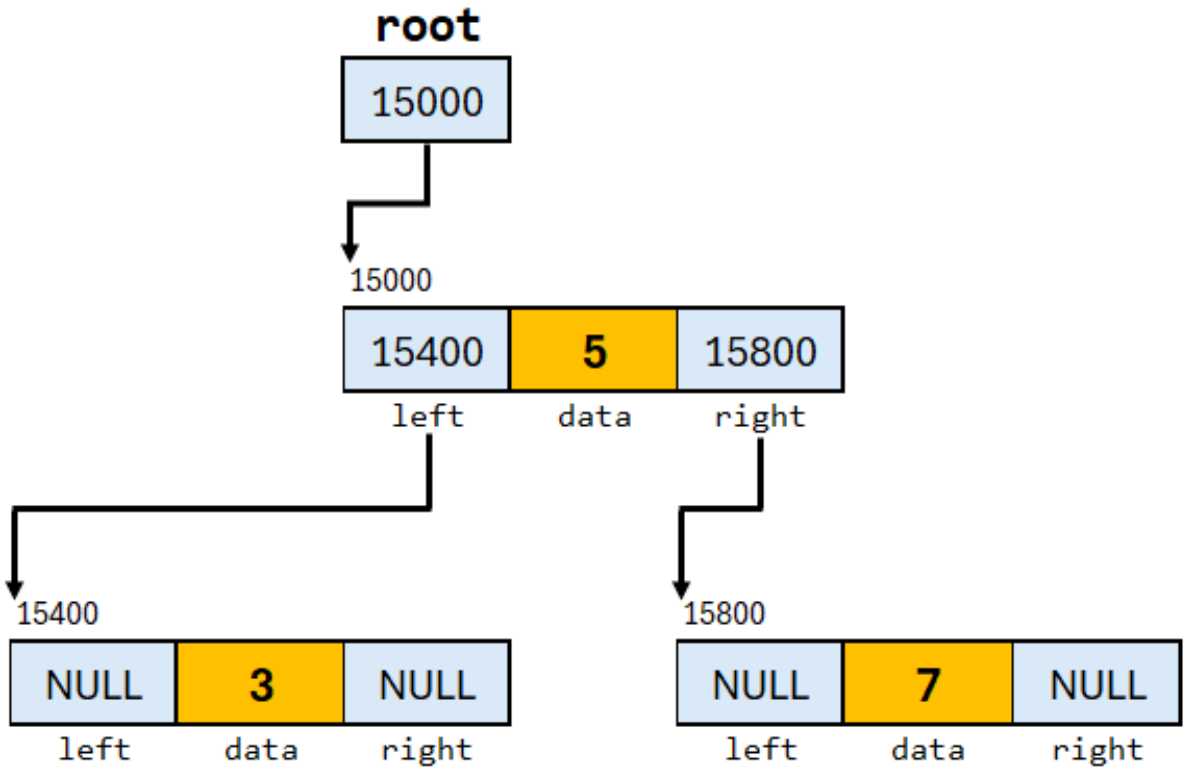


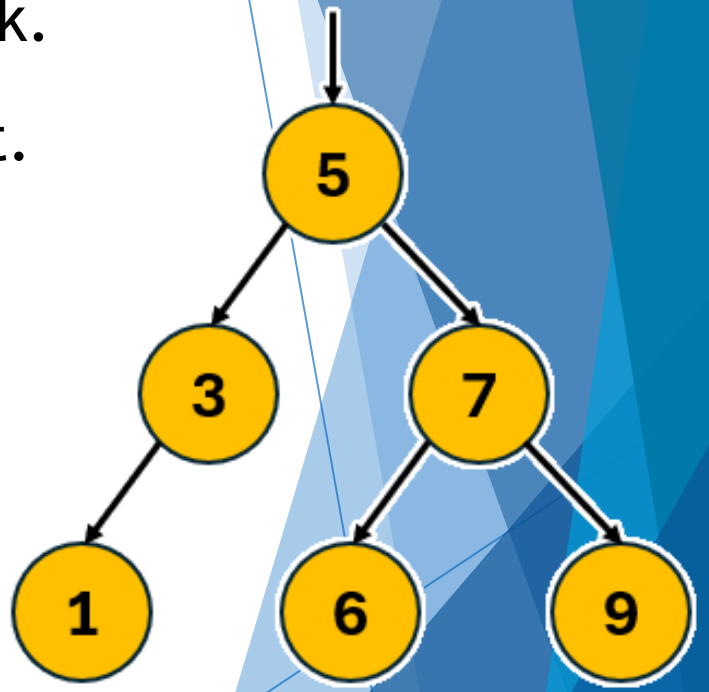
```c
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;           // tarolt adat
    struct Node* prev;  // mutato az elozo Node-ra
    struct Node* next;  // mutato a kovetkezo Node-ra
} Node;

// kiiras elejetol vegeig
void printForward(Node* head) {
    while (head != NULL) {
        printf("%d <-> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

// beszuras az elejere
void addFirst(Node** head_ptr, Node** tail_ptr, int value) {
    Node* newNode = (Node*) malloc(sizeof(Node));
    newNode->data = value;
    newNode->prev = NULL;
    newNode->next = *head_ptr;
    if (*head_ptr != NULL)
        (*head_ptr)->prev = newNode;
    else
        *tail_ptr = newNode;  // ha ures a lista
    *head_ptr = newNode;
}

// beszuras a vegere
void addLast(Node** head_ptr, Node** tail_ptr, int value) {
    Node* newNode = (Node*) malloc(sizeof(Node));
    newNode->data = value;
    newNode->next = NULL;
    newNode->prev = *tail_ptr;
    if (*tail_ptr != NULL)
        (*tail_ptr)->next = newNode;
    else
        *head_ptr = newNode;  // ha ures a lista
    *tail_ptr = newNode;
}

// memoria felszabaditasa
void freeList(Node* head) {
    while (head != NULL) {
        Node* temp = head;
        head = head->next;
        free(temp);
    }
}

int main() {
    // elso es utolso elemre mutato pointerek
    Node* elso = NULL;
    Node* utolso = NULL;
    // beszurasok
    addFirst(&elso, &utolso, 3); // 3
    addFirst(&elso, &utolso, 2); // 2 <-> 3
    addLast(&elso, &utolso, 4);  // 2 <-> 3 <-> 4
    addFirst(&elso, &utolso, 1); // 1 <-> 2 <-> 3 <-> 4
    addLast(&elso, &utolso, 5);  // 1 <-> 2 <-> 3 <-> 4 <-> 5
    // lista kiiratasa
    printForward(elso);
    // memoria felszabaditasa
    freeList(elso);
}
```

Bináris keresőfa

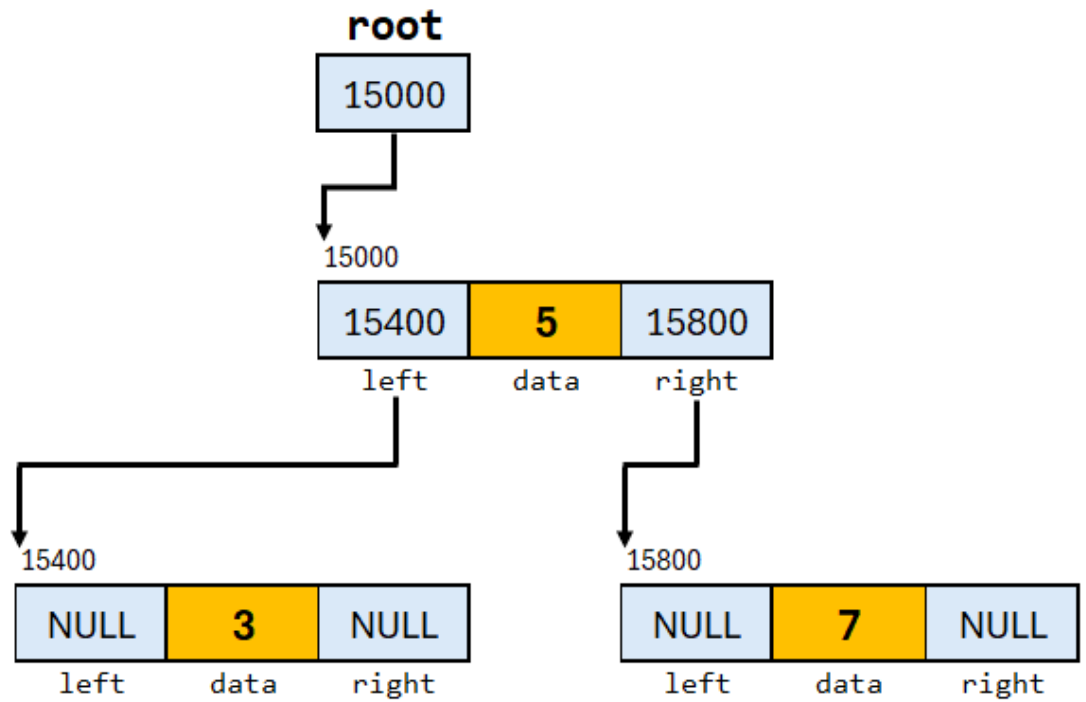
A **bináris keresőfa** olyan faalapú adatszerkezet, ahol minden csomópont legfeljebb két gyerekkel rendelkezik: a bal oldali részfában kisebb, a jobb oldaliban nagyobb értékek vannak.

Ez lehetővé teszi a hatékony keresést, beszúrást és törlést.

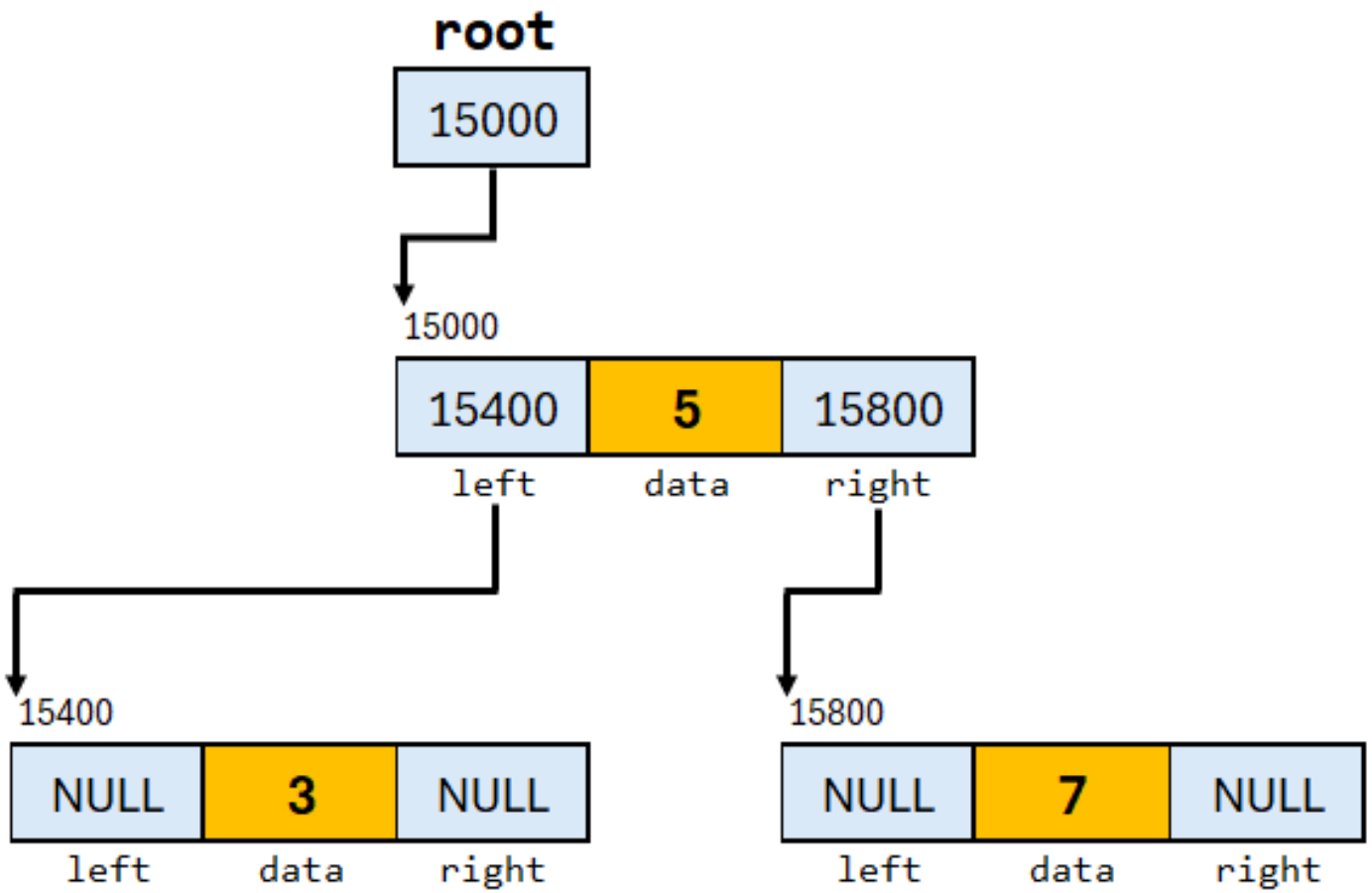




```c
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;              // tarolt adat
    struct Node* left;     // mutato a bal oldali agra
    struct Node* right;    // mutato a jobb oldali agra
} Node;

// uj elem beszurasa
Node* insert(Node* root, int value) {
    if (root == NULL) {
        root = (Node*) malloc(sizeof(Node));
        root->data = value;
        root->left = root->right = NULL;
    } else if (value < root->data) {
        root->left = insert(root->left, value);
    } else {
        root->right = insert(root->right, value);
    }
    return root;
}
```

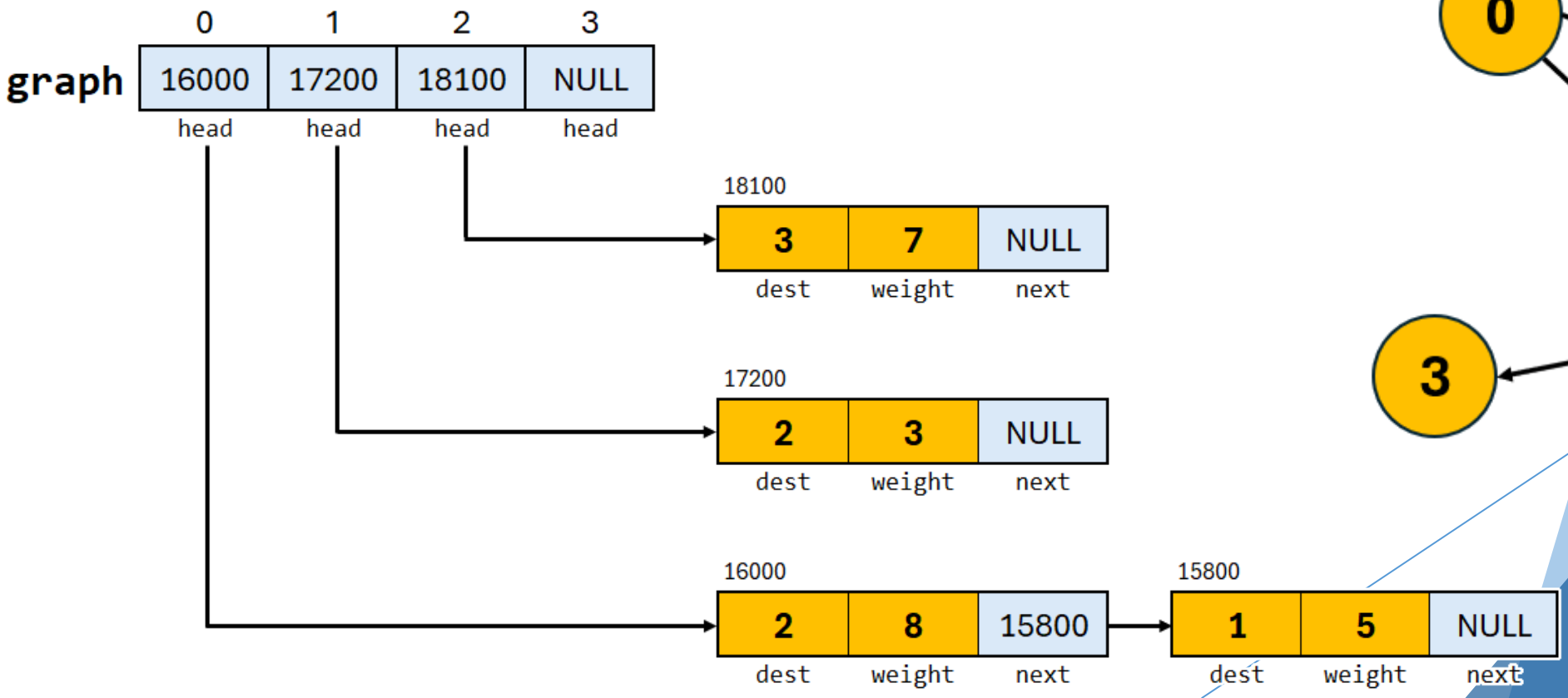


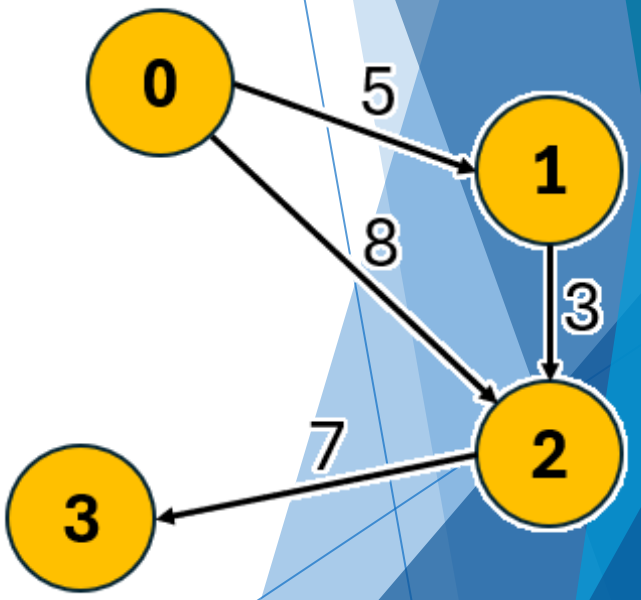
```c
// fa in-order bejarasa
void inorder(Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

int main() {
    Node* root = NULL;
    // fa letrehozasa
    root = insert(root, 5);
    insert(root, 3);
    insert(root, 7);
    // fa kiirasa
    inorder(root);  // 3 5 7
}
```



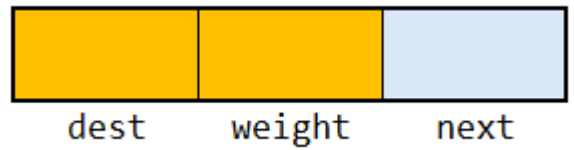
Irányított súlyozott gráf

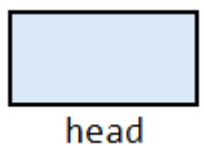
Az **irányított súlyozott gráf** olyan gráf, ahol az élek irányítottak (tehát egyik csúcsból a másikba mutatnak), és minden élhez egy számérték (súly) tartozik. Ez az érték például távolságot, költséget vagy időtartamot jelenthet.



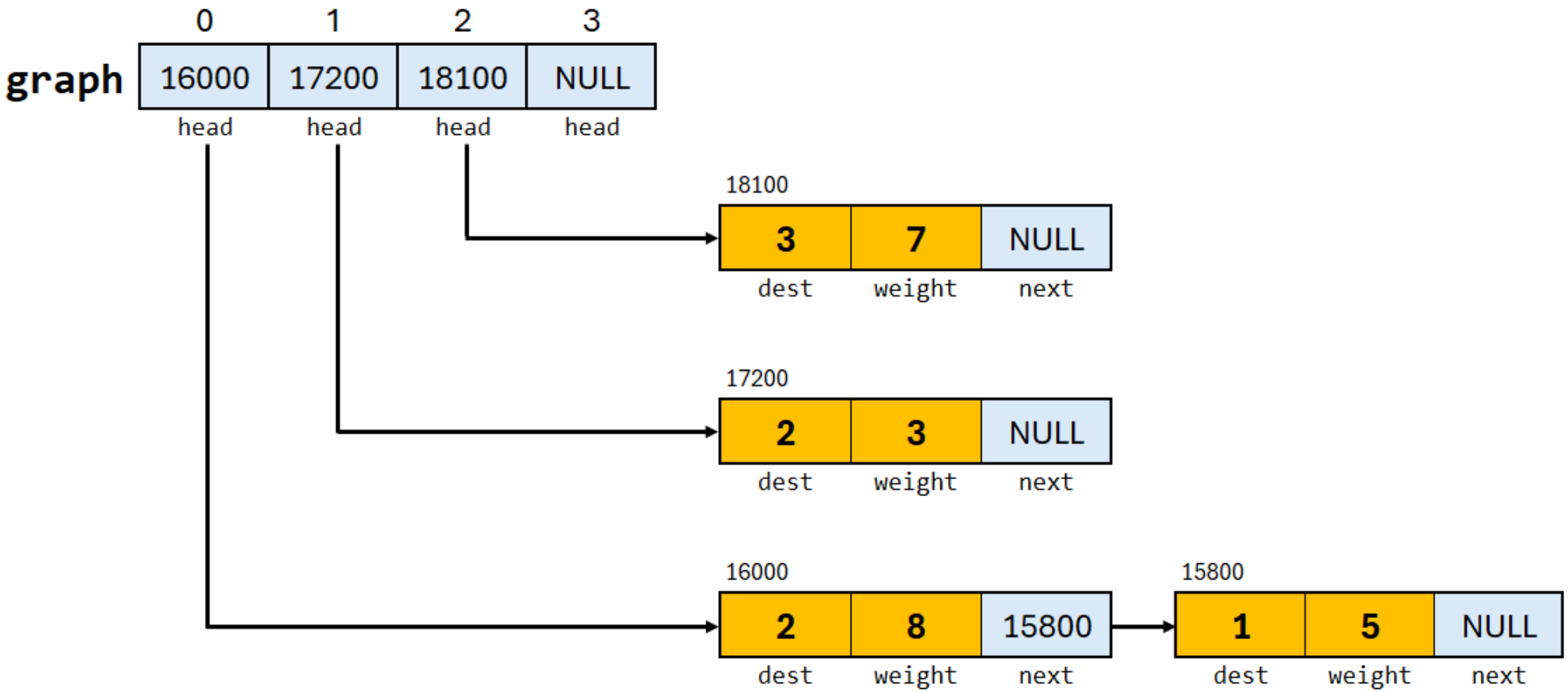


```c
#include <stdio.h>
#include <stdlib.h>

#define N 4  // csucsok szama a grafban

typedef struct Edge {
    int dest;           // hova megy az el (a celcsucs sorszama)
    int weight;         // az el sulya (pl. tavolsag, ido stb.)
    struct Edge* next;  // a kovetkezo el ugyanebbol a csucsbol
} Edge;

typedef struct {
    Edge* head;   // az adott csucsbol az elso elre mutato pointer
} AdjList;
```

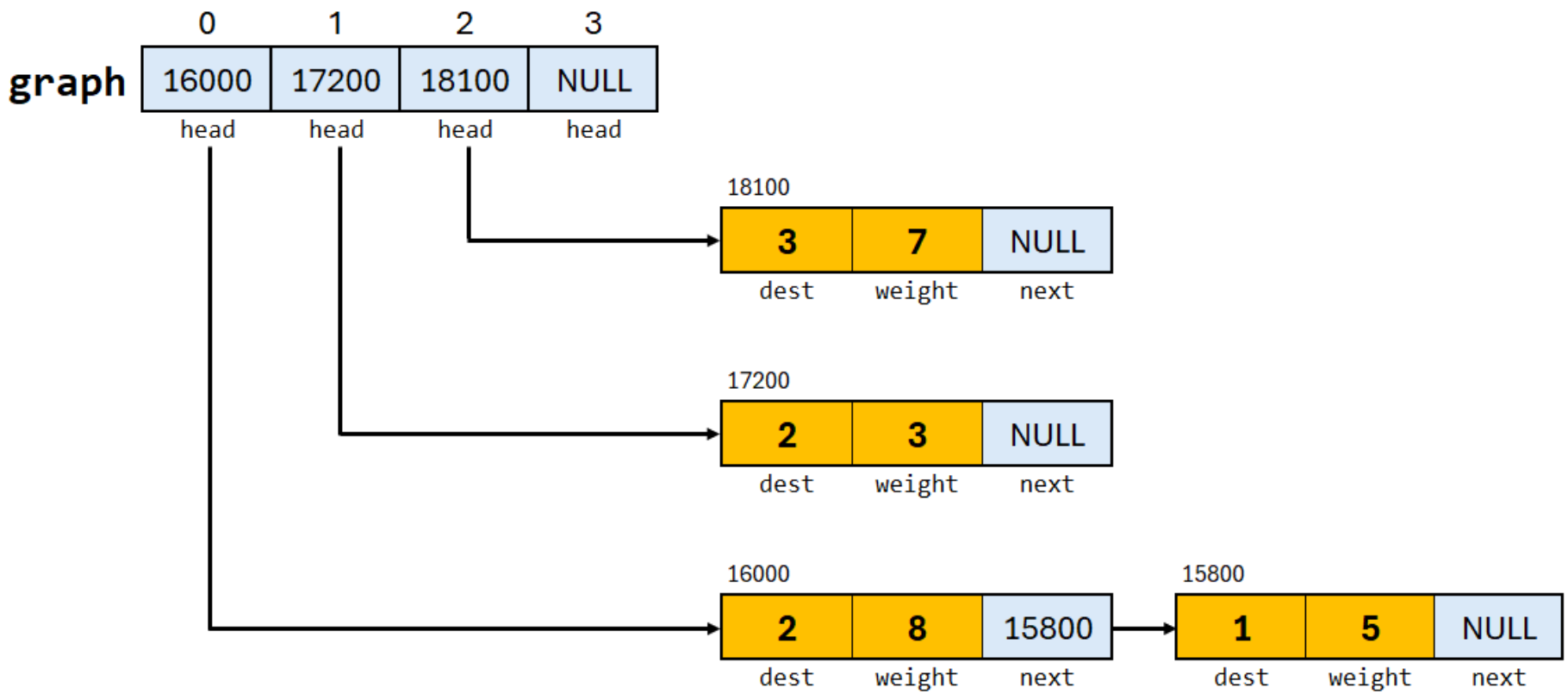




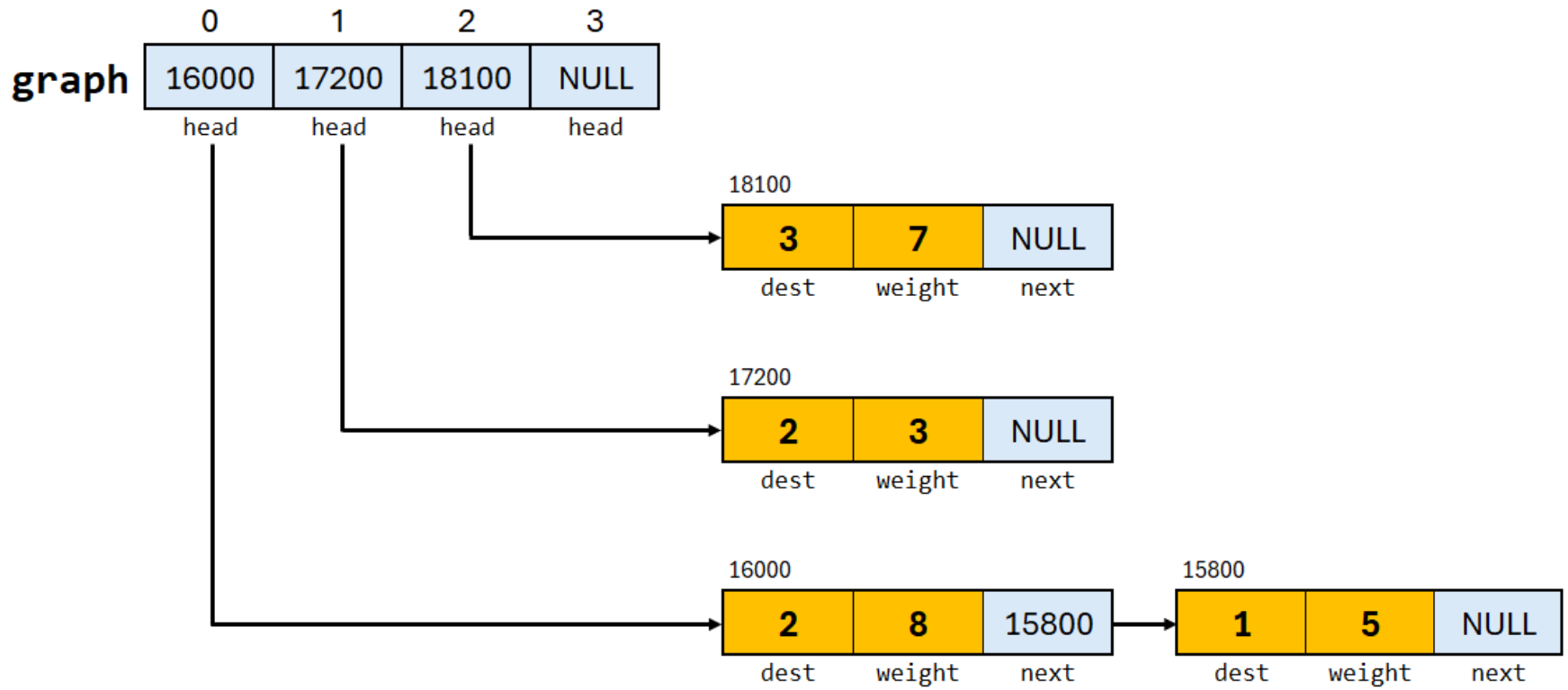
```
// uj el hozzaadasa a grafhoz
void addEdge(AdjList graph[], int from, int to, int weight) {
    Edge* newEdge = (Edge*) malloc(sizeof(Edge));
    newEdge->dest = to;
    newEdge->weight = weight;
    newEdge->next = graph[from].head;
    graph[from].head = newEdge;
}
```



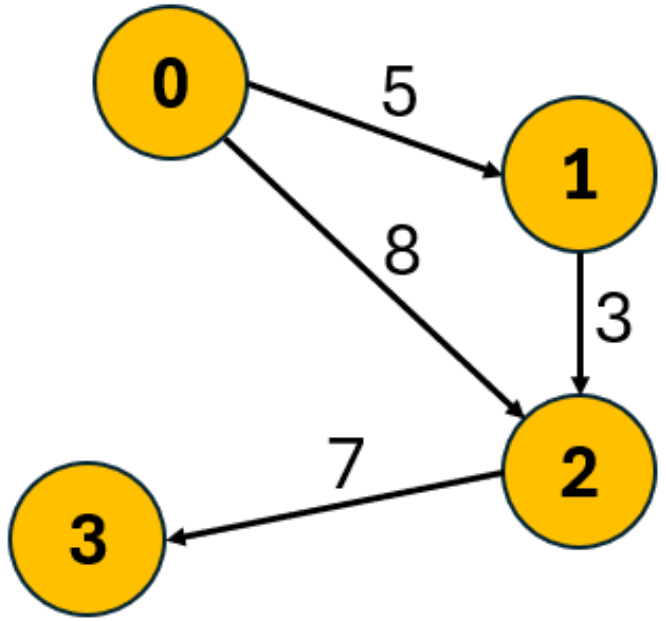
```c
// graf kiirasa a kepernyore
void printGraph(AdjList graph[]) {
    for (int i = 0; i < N; i++) {
        printf("Csucs %d: ", i);
        Edge* current = graph[i].head;
        while (current != NULL) {
            printf("-> %d (suly: %d) ", current->dest, current->weight);
            current = current->next;
        }
        printf("\n");
    }
}
```



```c
// dinamikusan lefoglalt memoria felszabaditasa
void freeGraph(AdjList graph[]) {
    for (int i = 0; i < N; i++) {
        Edge* current = graph[i].head;
        while (current != NULL) {
            Edge* tmp = current;
            current = current->next;
            free(tmp);
        }
        graph[i].head = NULL;
    }
}
```



```c
int main() {
    // N darab csucsot tartalmazo tomb
    // (szomszedsagi listas reprezentacio)
    AdjList graph[N] = {0};  // inicializalas NULL fejjel
    // elek hozzaadasa a grafhoz
    addEdge(graph, 0, 1, 5);
    addEdge(graph, 0, 2, 8);
    addEdge(graph, 1, 2, 3);
    addEdge(graph, 2, 3, 7);
    // graf kiiratasa
    printGraph(graph);
    // dinamikusan lefoglalt memoria felszabaditasa
    freeGraph(graph);
}
```




```
Csucs 0: -> 2 (suly: 8) -> 1 (suly: 5)
Csucs 1: -> 2 (suly: 3)
Csucs 2: -> 3 (suly: 7)
Csucs 3:
```

Köszönöm a figyelmet!