List

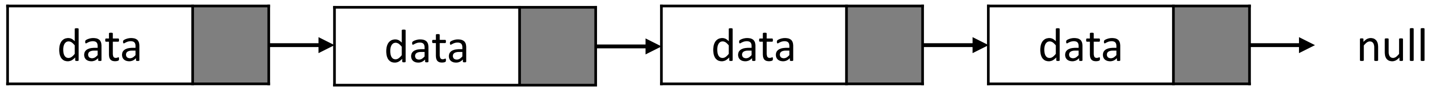
Data structures: List

- **List** is a linear data structure:
  - Order is not given by their physical placement in memory.
  - Each element points to the next.

- Example:
  - Train.

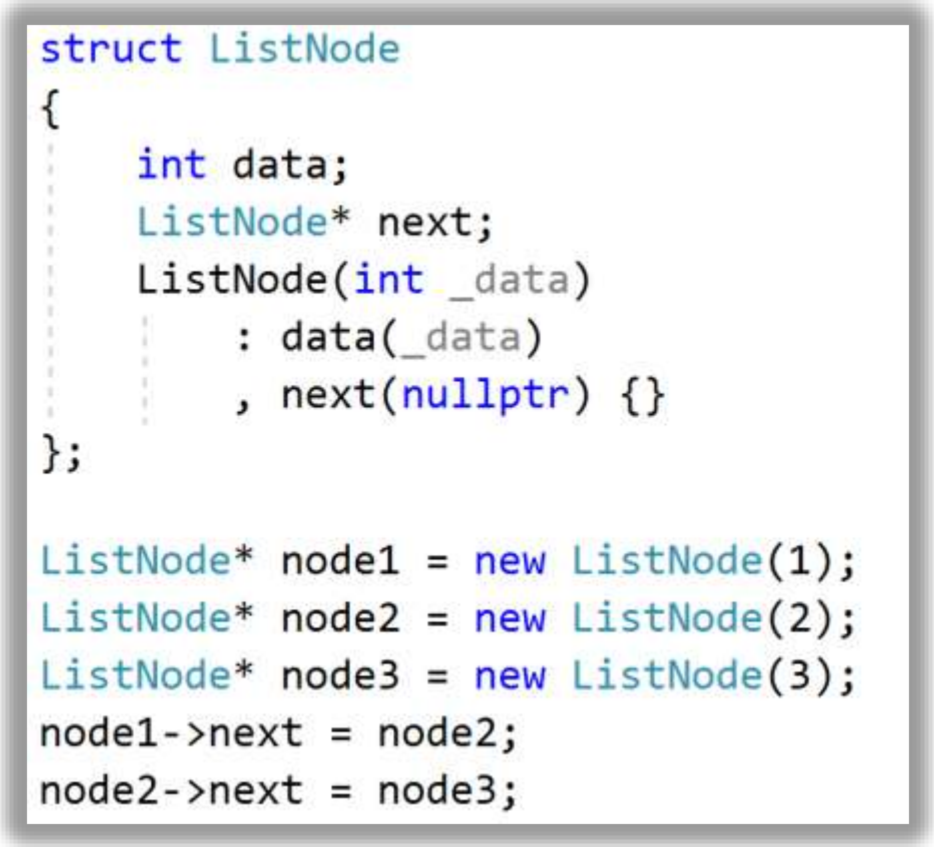
Data structures: List Example

- Here is the structure of list:
  - *data* – is a section for custom data, that list must hold.
  - *pointer* (grey part) – is a section that points to the next node. Last node points to the **null**.



Data structures: List Structure

- The following struct could be used to create a list from the last example:



```
struct ListNode
{
    int data;
    ListNode* next;
    ListNode(int _data)
        : data(_data)
        , next(nullptr) {}
};

ListNode* node1 = new ListNode(1);
ListNode* node2 = new ListNode(2);
ListNode* node3 = new ListNode(3);
node1->next = node2;
node2->next = node3;
```

Data structures: List Operations

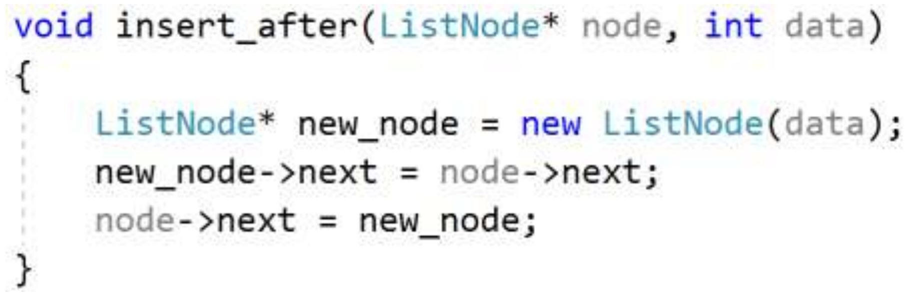
- **insert:** Inserts item to the specified position in list. Time complexity is O(1).

- **remove:** Removes item from the specified position in list. Time complexity is O(1).

- **access:** Element accessing is performs by iteration to that element. Time complexity is O(n).

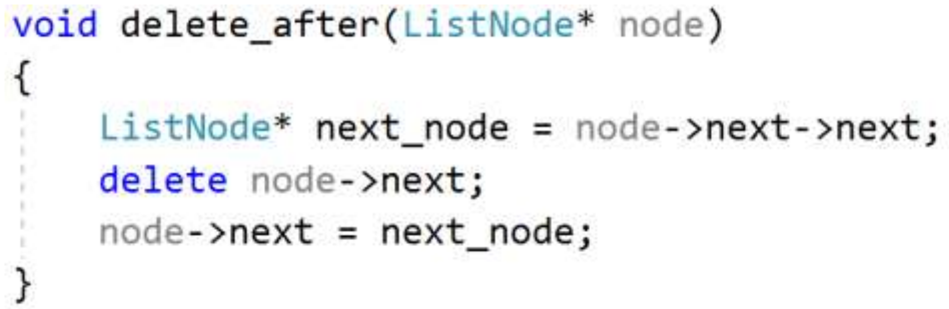
Data structures: List Problems

- For good understanding why insert and delete is O(1), let’s solve the following problems
  - Given a node, you need to insert given you value after this node.



```
void insert_after(ListNode* node, int data)
{
    ListNode* new_node = new ListNode(data);
    new_node->next = node->next;
    node->next = new_node;
}
```

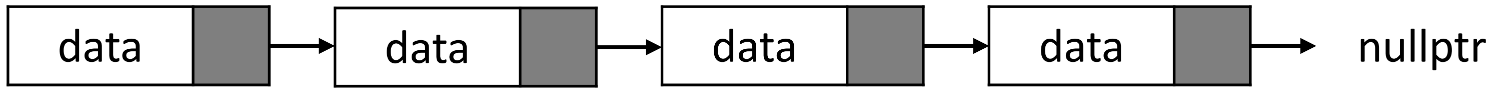
  - Given a node, you need to delete the node located next to it.



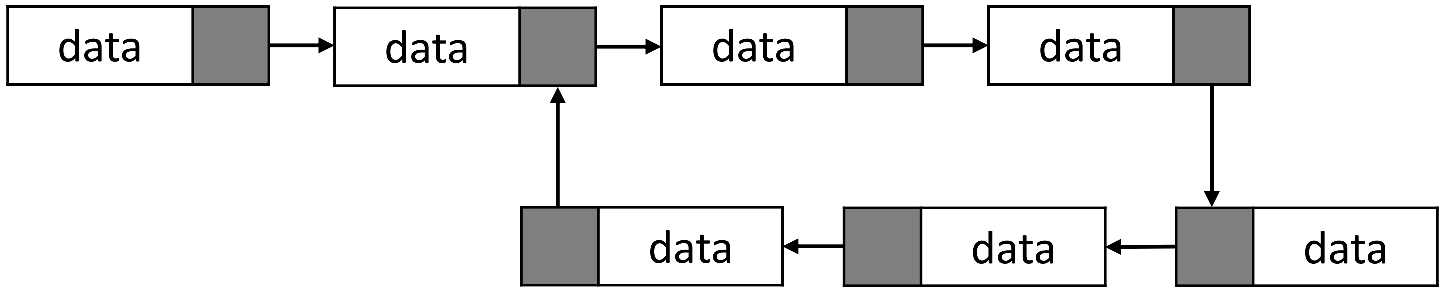
```
void delete_after(ListNode* node)
{
    ListNode* next_node = node->next->next;
    delete node->next;
    node->next = next_node;
}
```

Data structures: List
Problem: Cycle

- You are given a list by its first node, need to answer if there is a cycle in that list:
    - No cycle:



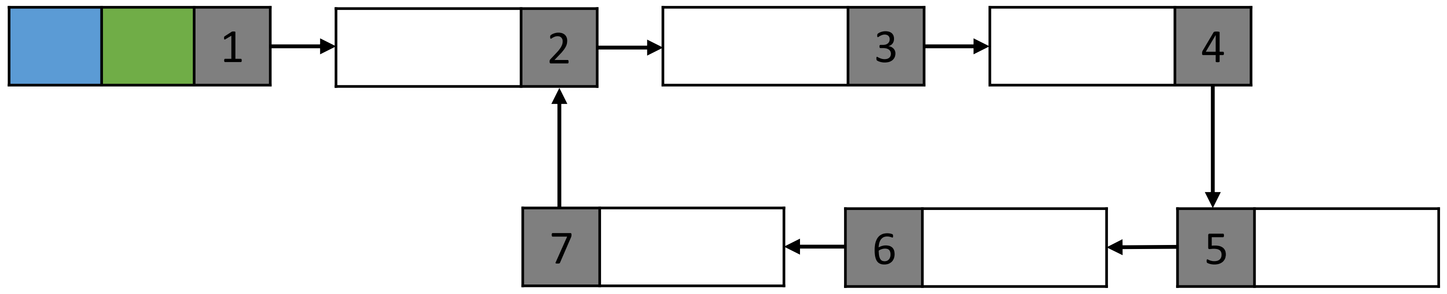
    - Has cycle:



Data structures: List
Problem: Cycle

- Here is visualization of mentioned algorithm:
    - First pointer is visualized as blue node and second as green.

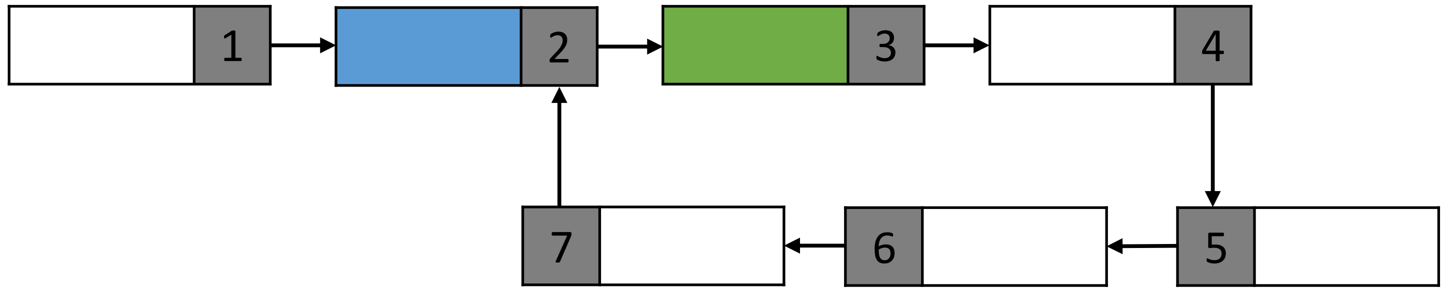
- At the start both pointers points to the node 1



Data structures: List Problem: Cycle

- Here is visualization of mentioned algorithm:
  - First pointer is visualized as blue node and second as green.

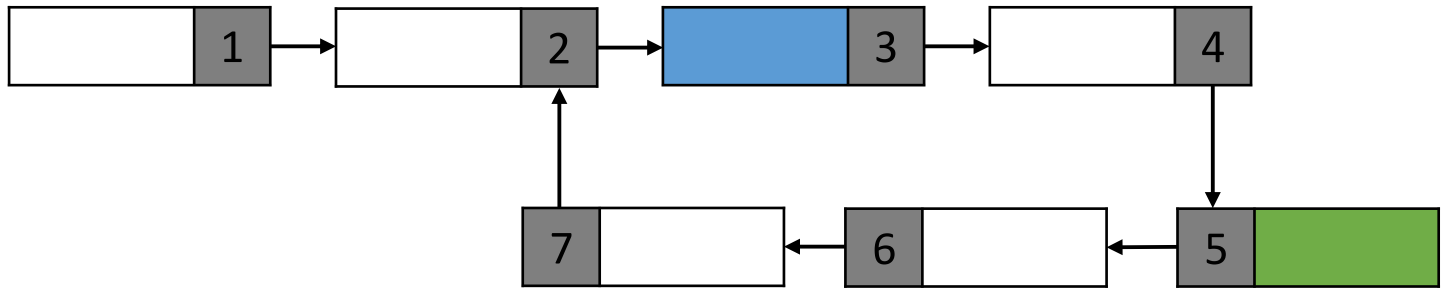
- On the step 1, first pointer start point to the node 2 and second to the node 3



Data structures: List
Problem: Cycle

- Here is visualization of mentioned algorithm:
    - First pointer is visualized as blue node and second as green.

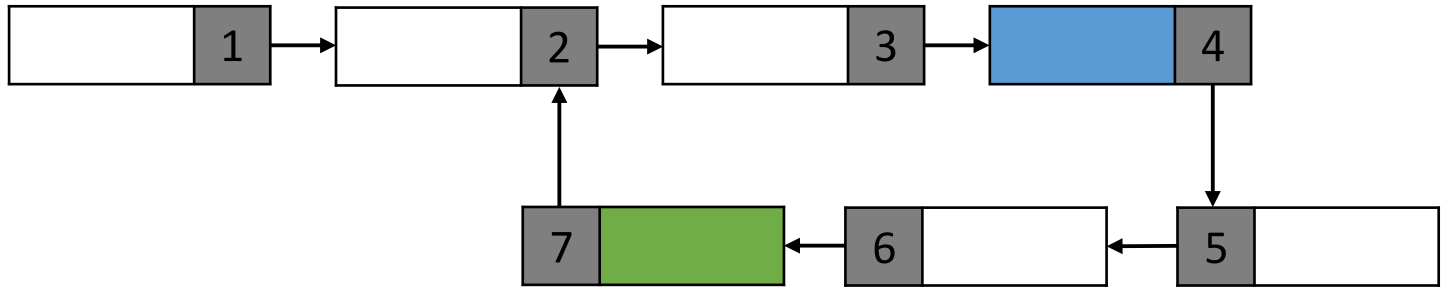
- On the step 2, first pointer start point to the node 3 and second to the node 5



Data structures: List
Problem: Cycle

- Here is visualization of mentioned algorithm:
  - First pointer is visualized as blue node and second as green.

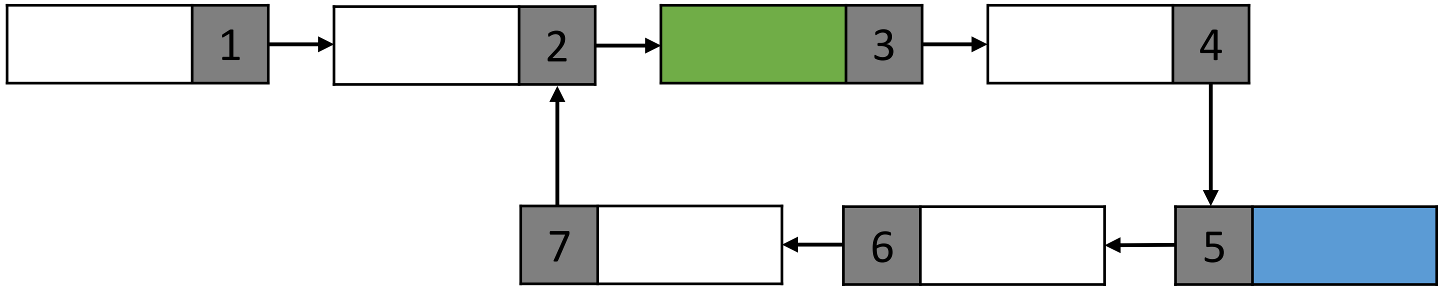
- On the step 3, first pointer start point to the node 4 and second to the node 7



Data structures: List
Problem: Cycle

- Here is visualization of mentioned algorithm:
  - First pointer is visualized as blue node and second as green.

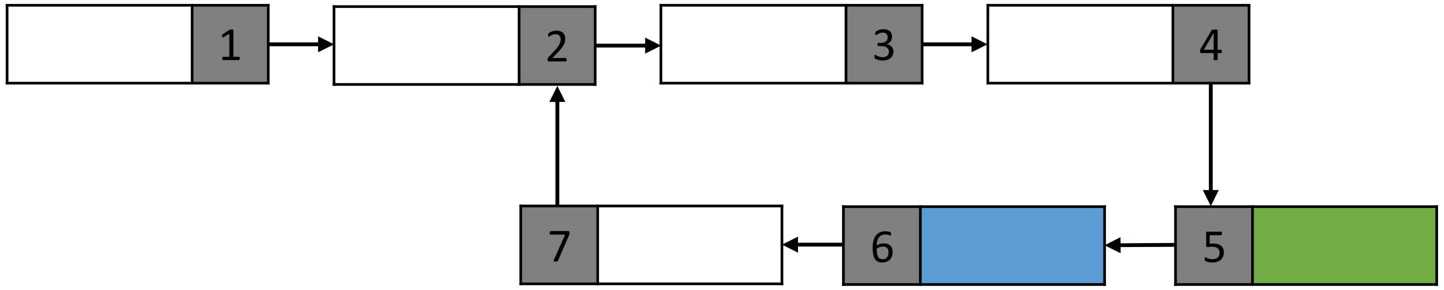
- On the step 4, first pointer start point to the node 5 and second to the node 3



Data structures: List Problem: Cycle

- Here is visualization of mentioned algorithm:
    - First pointer is visualized as blue node and second as green.

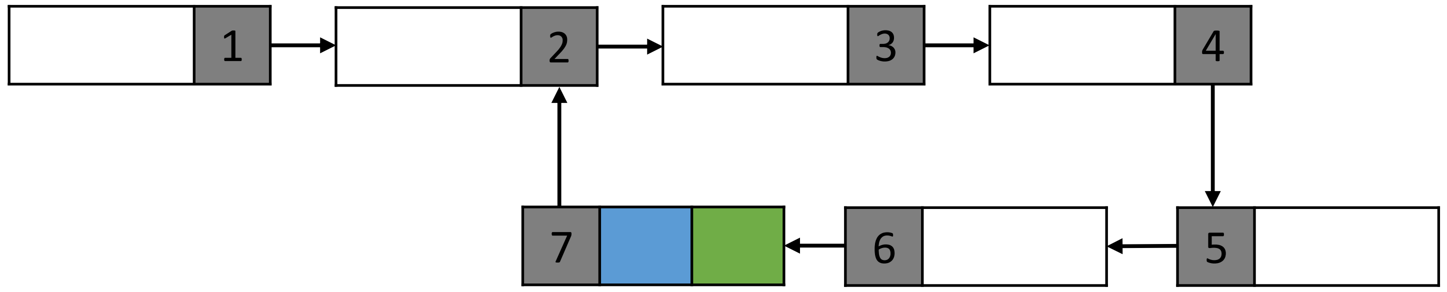
- On the step 5, first pointer start point to the node 6 and second to the node 5



Data structures: List
Problem: Cycle

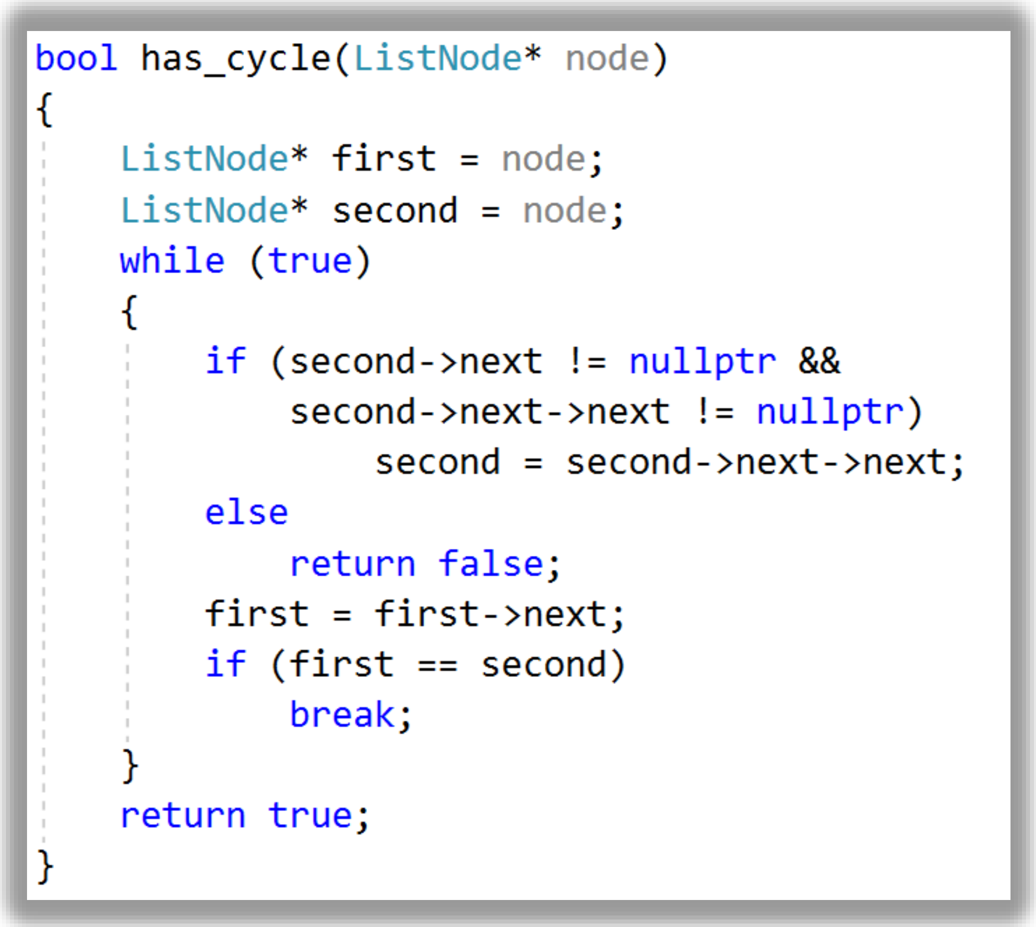
- Here is visualization of mentioned algorithm:
  - First pointer is visualized as blue node and second as green.

- And on the step 6, both pointers will point on the same node 7, and algorithm will return true.



Data structures: List Problem: Cycle

- Solution:
  - Let’s take 2 pointers on the first node. On each step move one pointer to the next node, and second pointer to the next next node. If there was a cycle in some moment these 2 pointers will point on the same node. If not, then the second pointer will reach null.



```
bool has_cycle(ListNode* node)
{
    ListNode* first = node;
    ListNode* second = node;
    while (true)
    {
        if (second->next != nullptr &&
            second->next->next != nullptr)
                second = second->next->next;
        else
            return false;
        first = first->next;
        if (first == second)
            break;
    }
    return true;
}
```