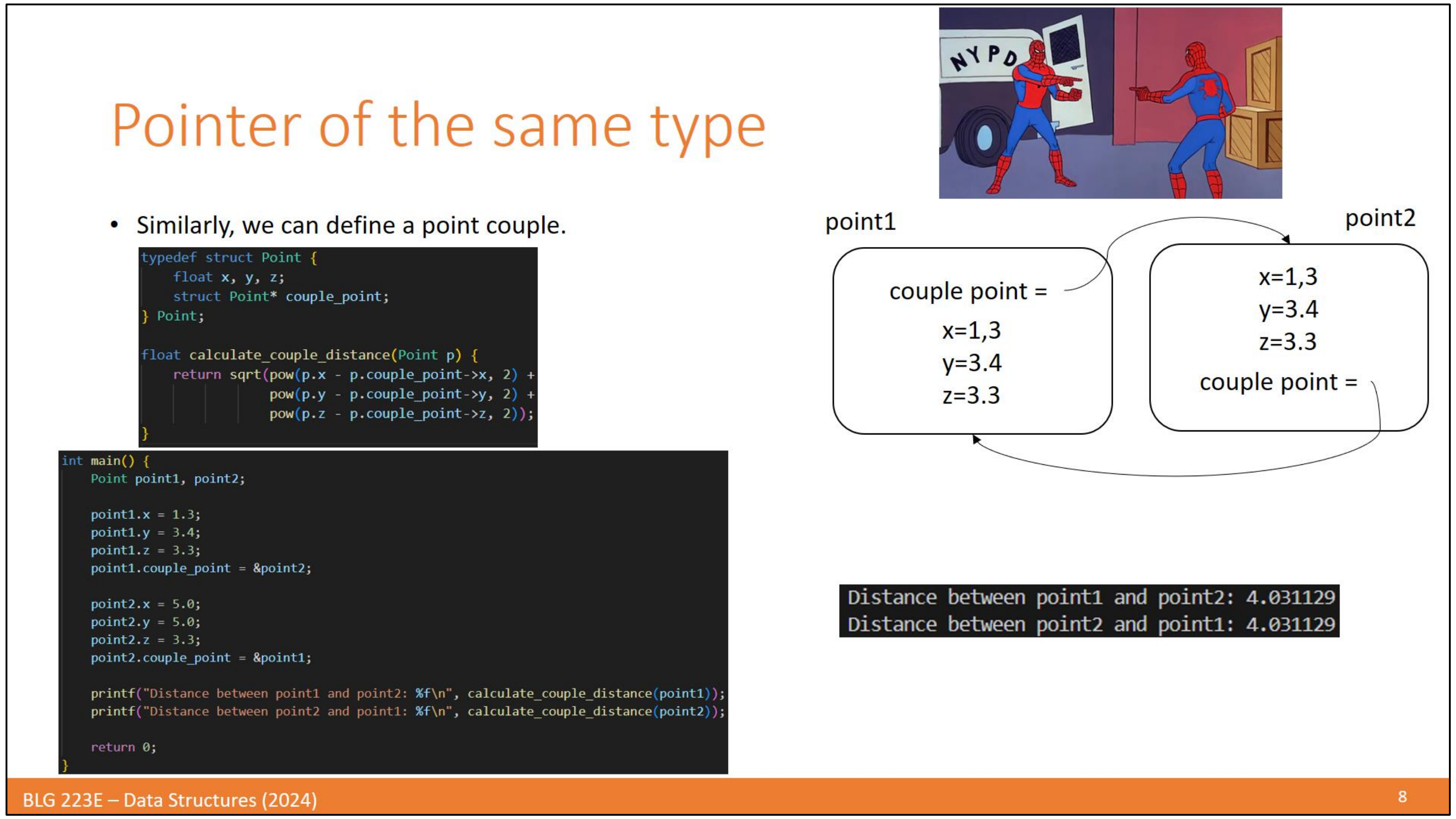
Lecture 4

Linked Lists and Linked List Operations

Dr. Yusuf H. Sahin
Istanbul Technical University

sahinyu@itu.edu.tr

A gentle reminder from previous week

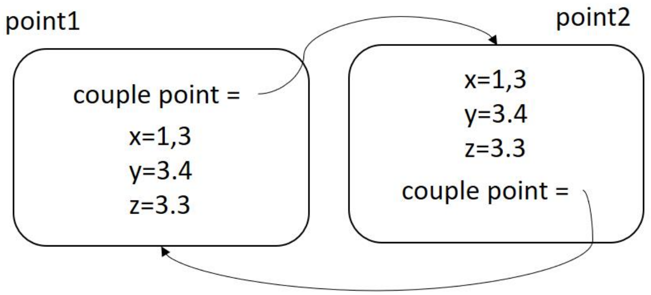


Pointer of the same type

- Similarly, we can define a point couple.

```c
typedef struct Point {
    float x, y, z;
    struct Point* couple_point;
} Point;

float calculate_couple_distance(Point p) {
    return sqrt(pow(p.x - p.couple_point->x, 2) +
                pow(p.y - p.couple_point->y, 2) +
                pow(p.z - p.couple_point->z, 2));
}
```

```c
int main() {
    Point point1, point2;

    point1.x = 1.3;
    point1.y = 3.4;
    point1.z = 3.3;
    point1.couple_point = &point2;

    point2.x = 5.0;
    point2.y = 5.0;
    point2.z = 3.3;
    point2.couple_point = &point1;

    printf("Distance between point1 and point2: %f\n", calculate_couple_distance(point1));
    printf("Distance between point2 and point1: %f\n", calculate_couple_distance(point2));

    return 0;
}
```



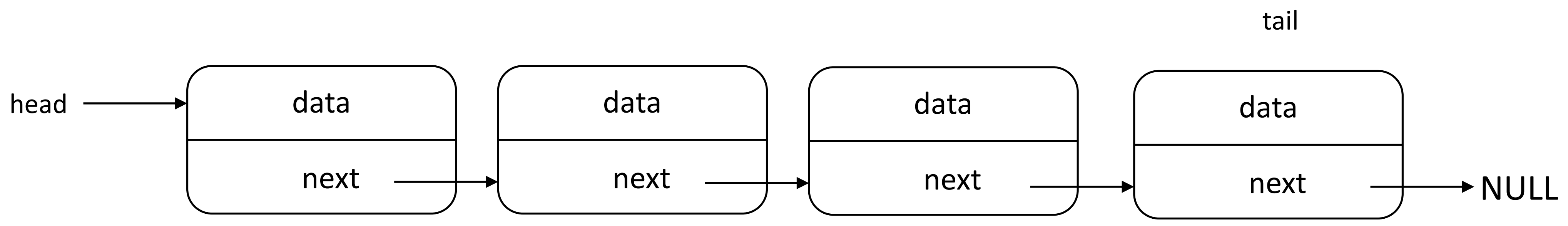
```
Distance between point1 and point2: 4.031129
Distance between point2 and point1: 4.031129
```

BLG 223E – Data Structures (2024) 8

- In C, a struct can have a pointer to its own type. Which could be used to generate many different data structures.

Singly Linked Lists

- A linked list is a collection of elements (nodes) in a linear ordering.
- A singly linked list contains a head pointer which points to the «head» of the list.
- Each object of the list is connected to its next element via a pointer.
- The final element could point to «null» as a common practise.



Singly Linked Lists

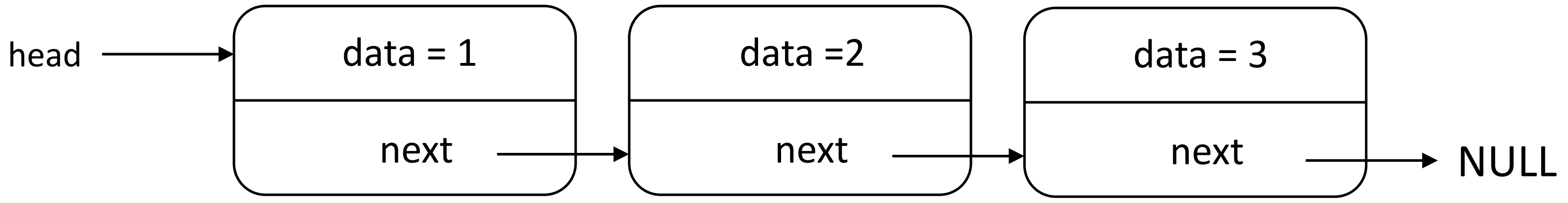
- Using pointer mechanism, we can directly chain the nodes to obtain a linked list.

```c
typedef struct intNode {
    int data;
    struct intNode* next;
} intNode;

typedef struct intList {
    intNode* head;
    int elemcount;
} intList;
```

```c
intNode* Node1 = (intNode*)malloc(sizeof(intNode));
intNode* Node2 = (intNode*)malloc(sizeof(intNode));
intNode* Node3 = (intNode*)malloc(sizeof(intNode));

Node1->data = 1;
Node2->data = 2;
Node3->data = 3;

Node1->next = Node2;
Node2->next = Node3;
Node3->next = NULL;

intList mylist;
mylist.head = Node1;
mylist.elemcount = 3;
```

```c
intNode* ptr = mylist.head;
while (ptr != NULL) {
    printf("%d\n", ptr->data);
    ptr = ptr->next;
}

free(Node1);
free(Node2);
free(Node3);
```

```
1
2
3
```



- Yet, it is more efficient to use member functions to add, delete, update, search and sort operations.

Add a new node to the front

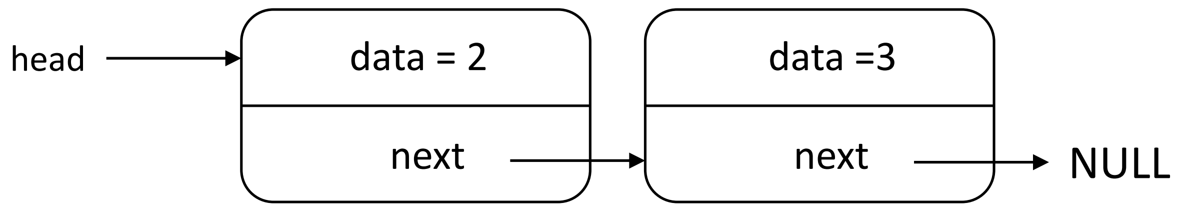
```c
typedef struct intNode {
    int data;
    struct intNode* next;
} intNode;
```

```c
typedef struct intList {
    intNode* head;
    int elemcount;
} intList;
```

- Easiest case to manipulate the linked list is to add and remove frm the front.

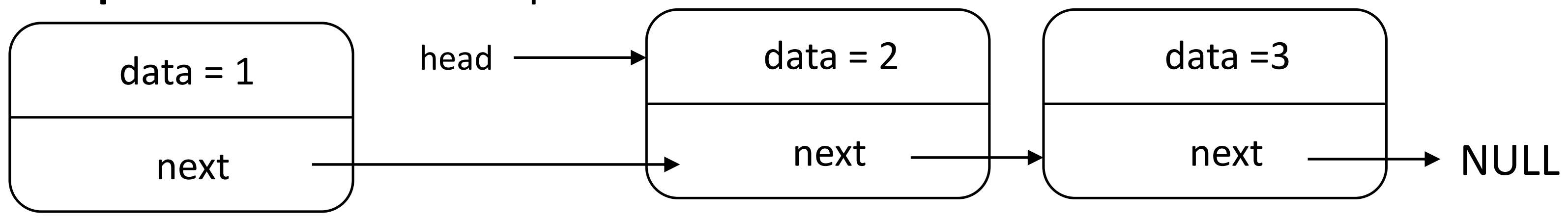
```c
void initList(intList *list) {
    list->head = NULL;
    list->elemcount = 0;
}
```

Initial State

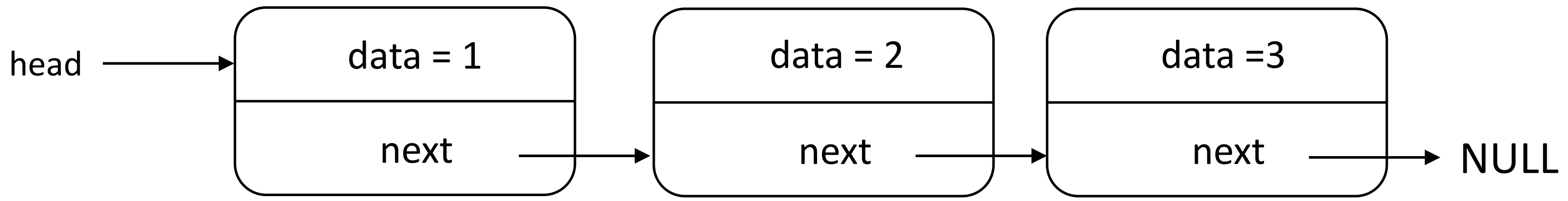


```c
void addFront(intList *list, intNode *new_node) {
    new_node->next = list->head;
    list->head = new_node;
    list->elemcount++;
}
```

Step 1: New node should point to the old head.



Step 2: Head should point to the new node.

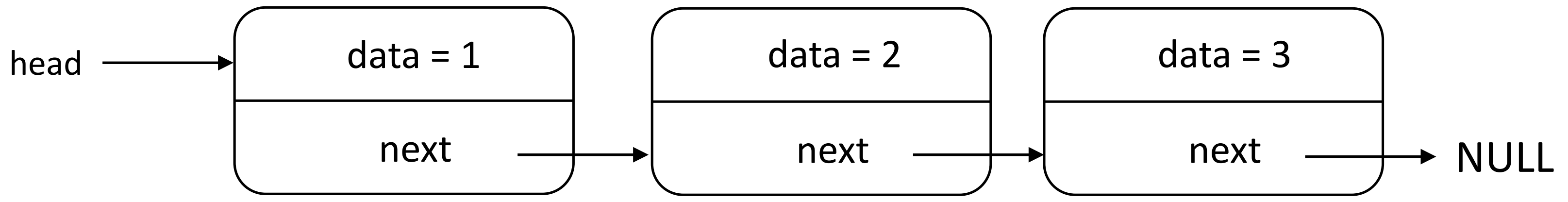


```c
intList mylist;
initList(&mylist);
for (int i = 3; i > 0; i--) {
    intNode *new_node = (intNode*)malloc(sizeof(intNode));
    new_node->data = i;
    addFront(&mylist, new_node);
}
intNode *temp = mylist.head;
while (temp != NULL) {
    printf("%d -> ", temp->data);
    temp = temp->next;
}
printf("NULL\n");
```

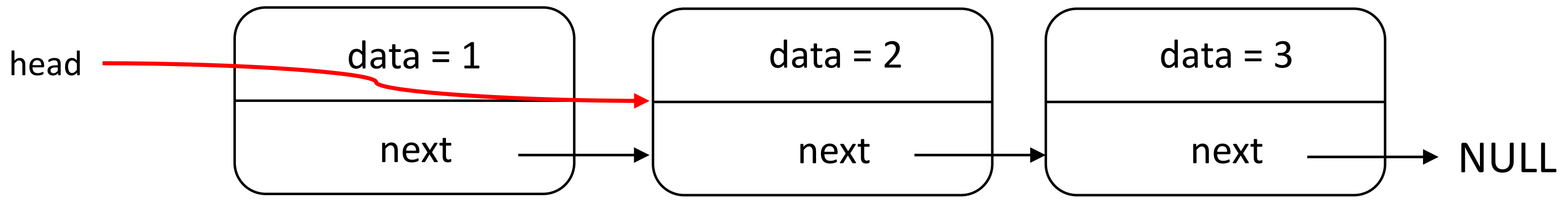
Remove a node from the front

- Easiest case to manipulate the linked list is to add and remove frm the front.

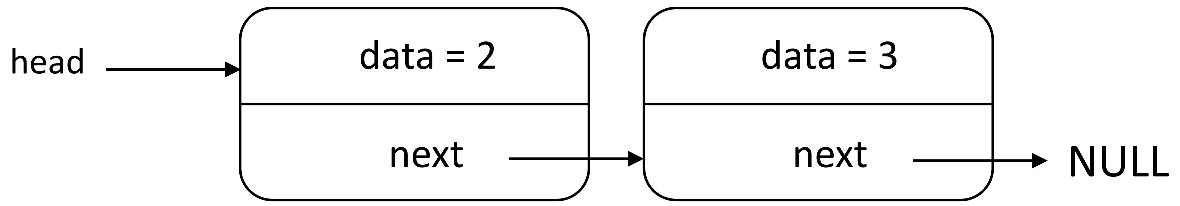
Initial State

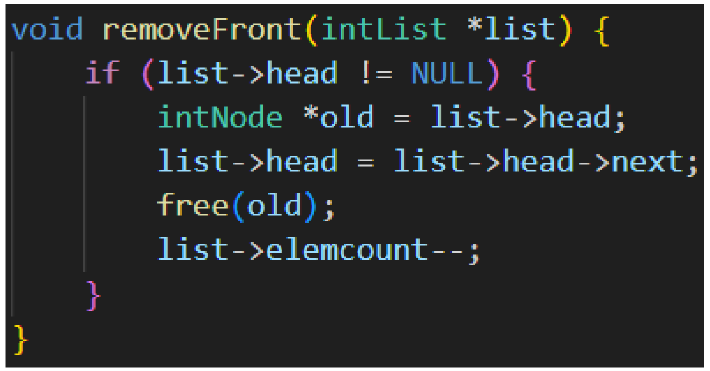


Step 1: Head should point to the next node.

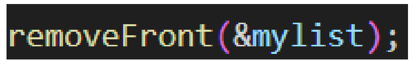


Step 2: Old head should be deleted.





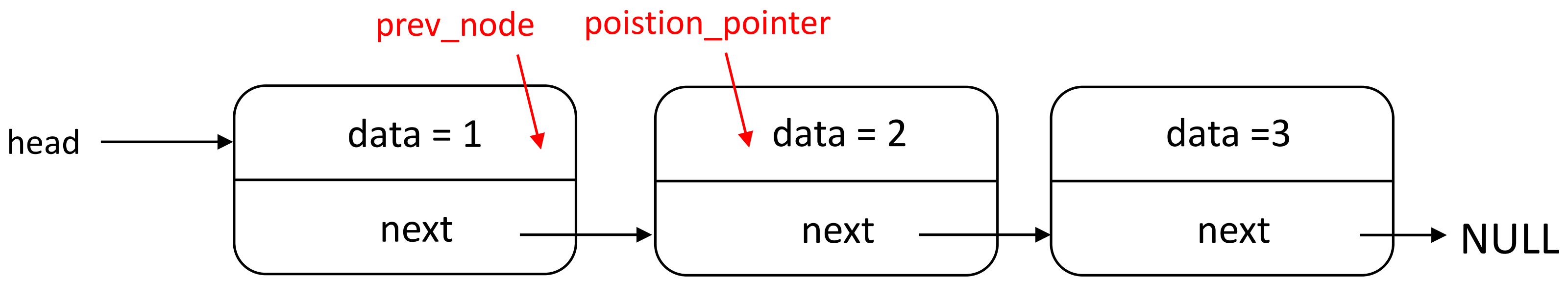
```
void removeFront(intList *list) {
    if (list->head != NULL) {
        intNode *old = list->head;
        list->head = list->head->next;
        free(old);
        list->elemcount--;
    }
}
```



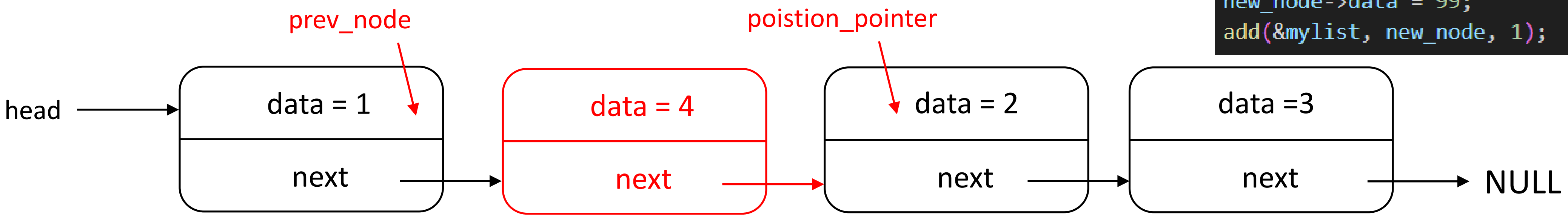
```
removeFront(&mylist);
```

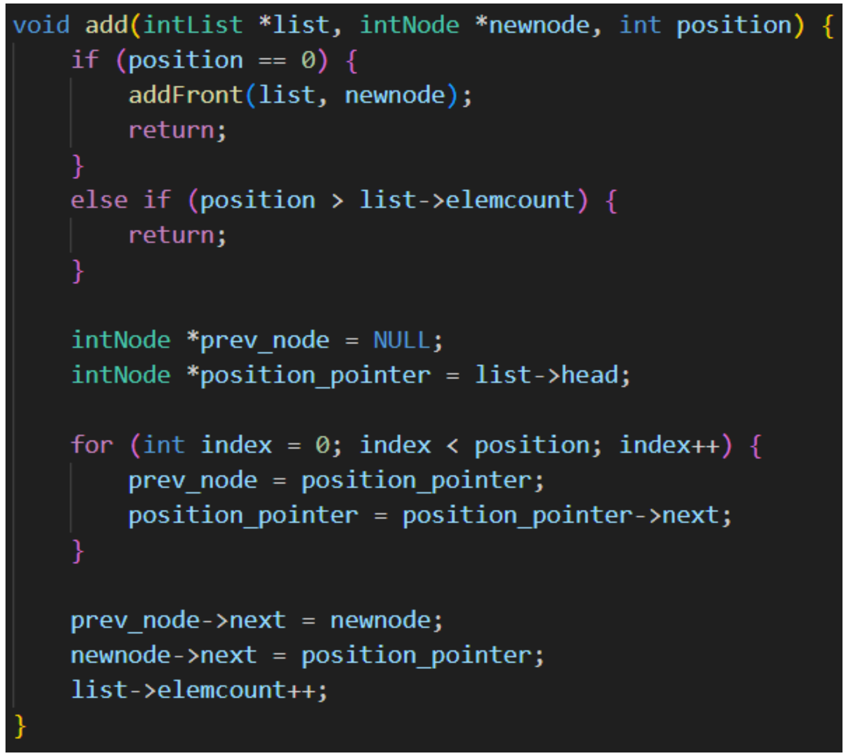
Place a node to a given place

- By iterating a pointer we can find the position to place a specific node.

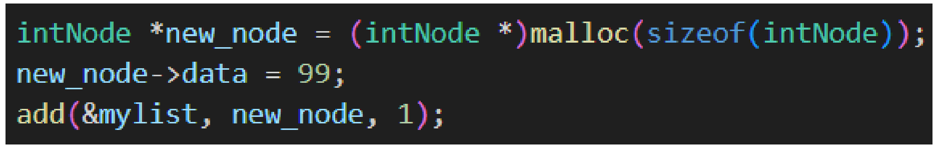


- Node at the previous position should point to the new node.
- New node should point to the node at the specific position.





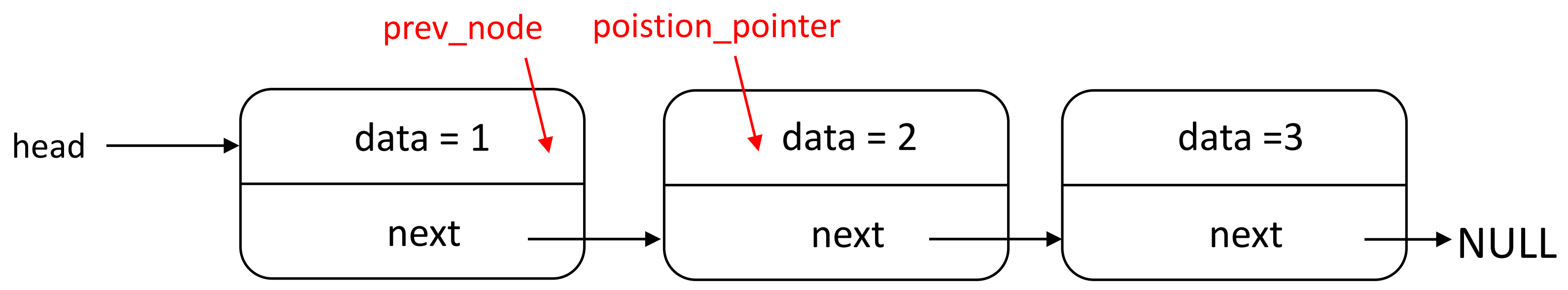
```c
void add(intList *list, intNode *newnode, int position) {
    if (position == 0) {
        addFront(list, newnode);
        return;
    }
    else if (position > list->elemcount) {
        return;
    }

    intNode *prev_node = NULL;
    intNode *position_pointer = list->head;

    for (int index = 0; index < position; index++) {
        prev_node = position_pointer;
        position_pointer = position_pointer->next;
    }

    prev_node->next = newnode;
    newnode->next = position_pointer;
    list->elemcount++;
}
```



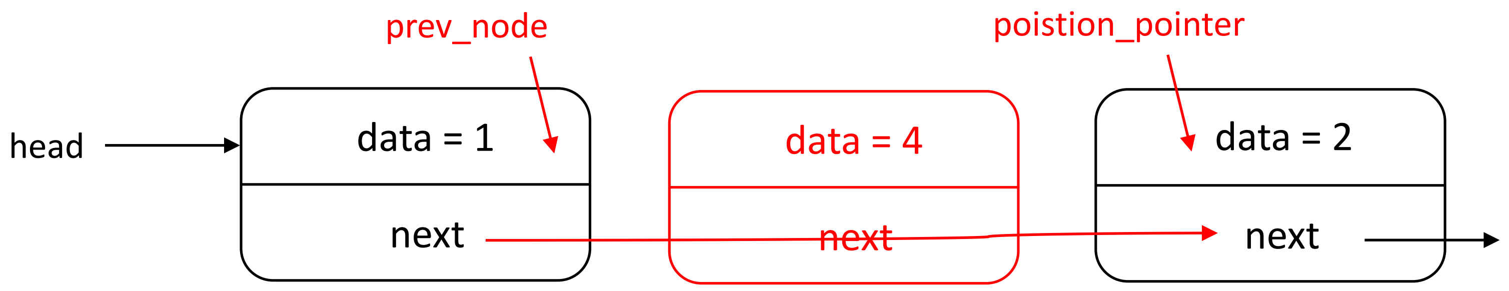
```c
intNode *new_node = (intNode *)malloc(sizeof(intNode));
new_node->data = 99;
add(&mylist, new_node, 1);
```

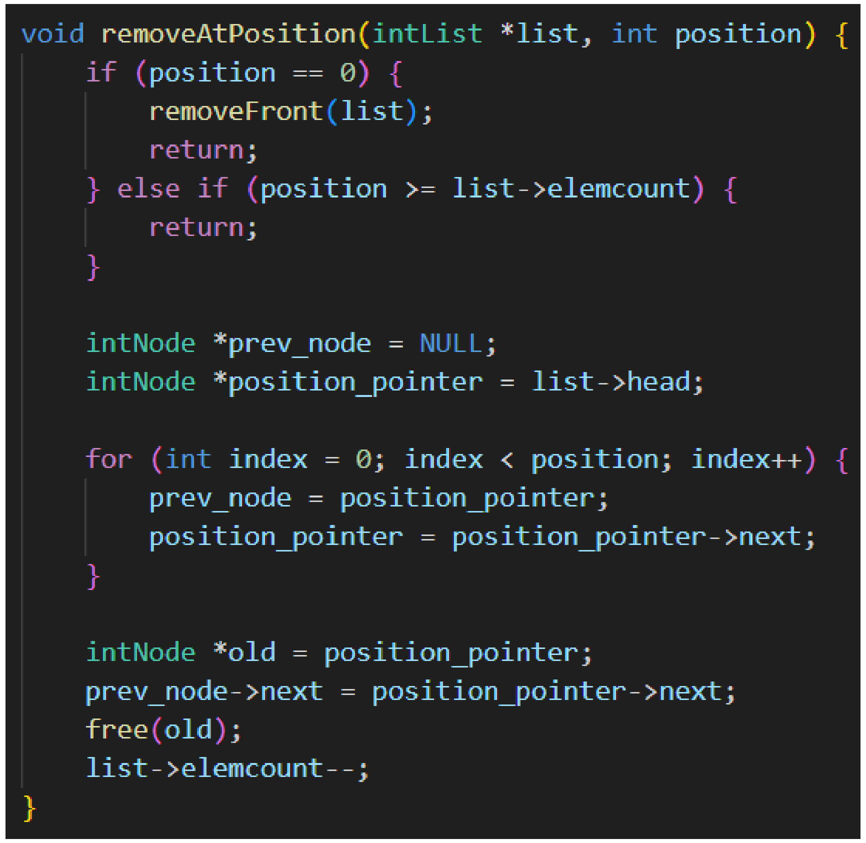
Remove a node from a given index

- By iterating a pointer we can find the position to place a specific node.

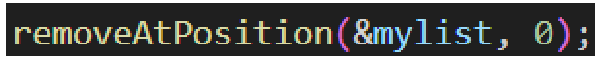


- Node at the previous position should point to the next node.
- The node at the exact position should be deleted.





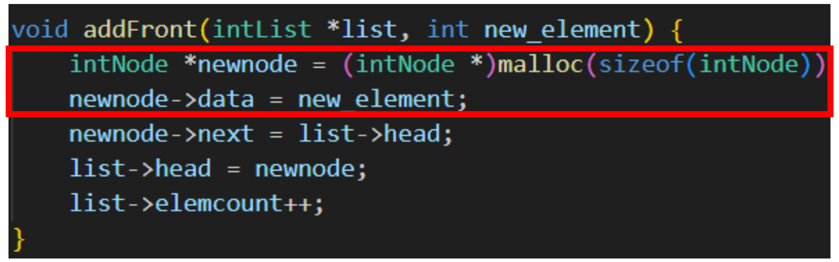
```c
void removeAtPosition(intList *list, int position) {
    if (position == 0) {
        removeFront(list);
        return;
    } else if (position >= list->elemcount) {
        return;
    }

    intNode *prev_node = NULL;
    intNode *position_pointer = list->head;

    for (int index = 0; index < position; index++) {
        prev_node = position_pointer;
        position_pointer = position_pointer->next;
    }

    intNode *old = position_pointer;
    prev_node->next = position_pointer->next;
    free(old);
    list->elemcount--;
}
```



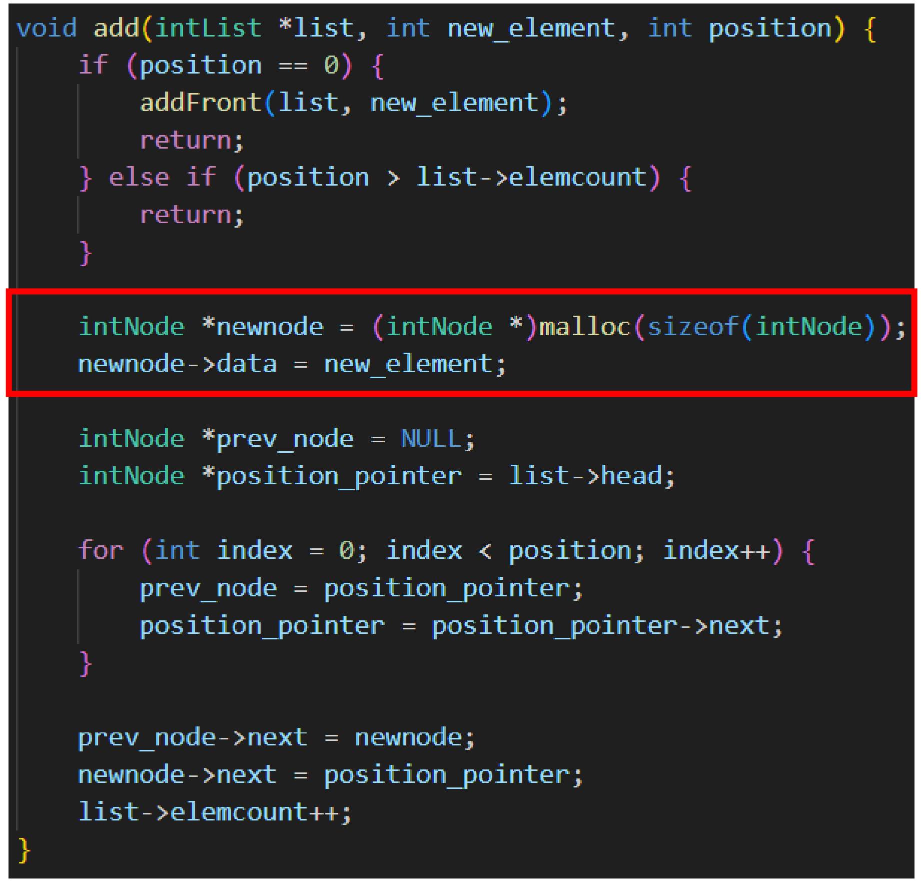
```c
removeAtPosition(&mylist, 0);
```

Hiding the Node

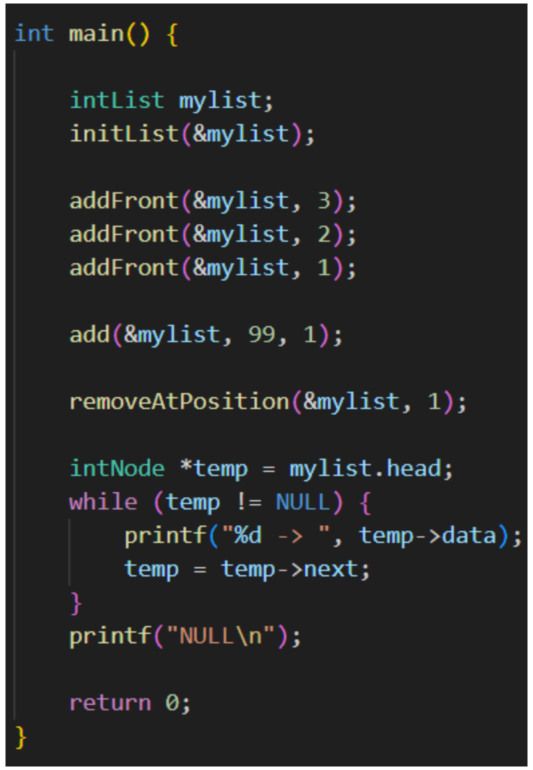
- It is safer to create and use the Node struct only in the helper functions.



```
void addFront(intList *list, int new_element) {
    intNode *newnode = (intNode *)malloc(sizeof(intNode))
    newnode->data = new_element;
    newnode->next = list->head;
    list->head = newnode;
    list->elemcount++;
}
```



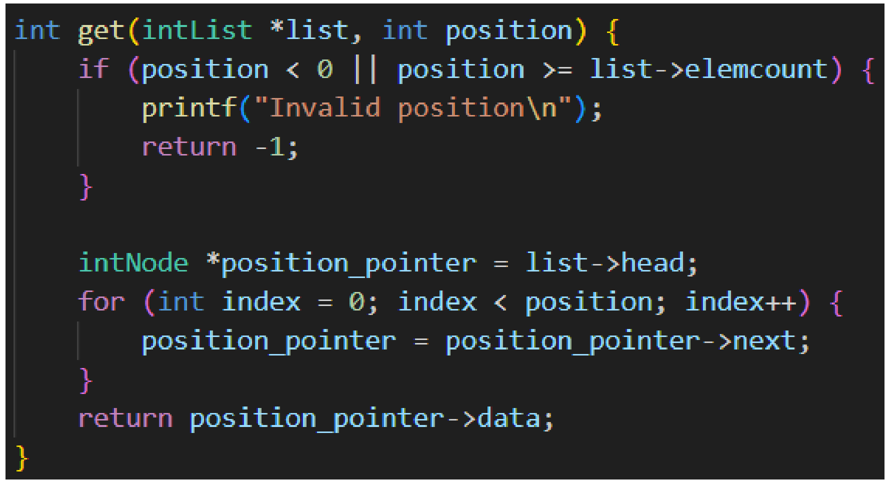
```
void add(intList *list, int new_element, int position) {
    if (position == 0) {
        addFront(list, new_element);
        return;
    } else if (position > list->elemcount) {
        return;
    }

    intNode *newnode = (intNode *)malloc(sizeof(intNode));
    newnode->data = new_element;

    intNode *prev_node = NULL;
    intNode *position_pointer = list->head;

    for (int index = 0; index < position; index++) {
        prev_node = position_pointer;
        position_pointer = position_pointer->next;
    }

    prev_node->next = newnode;
    newnode->next = position_pointer;
    list->elemcount++;
}
```



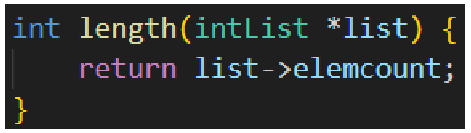
```
int main() {

    intList mylist;
    initList(&mylist);

    addFront(&mylist, 3);
    addFront(&mylist, 2);
    addFront(&mylist, 1);

    add(&mylist, 99, 1);

    removeAtPosition(&mylist, 1);

    intNode *temp = mylist.head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");

    return 0;
}
```

Reaching the elements

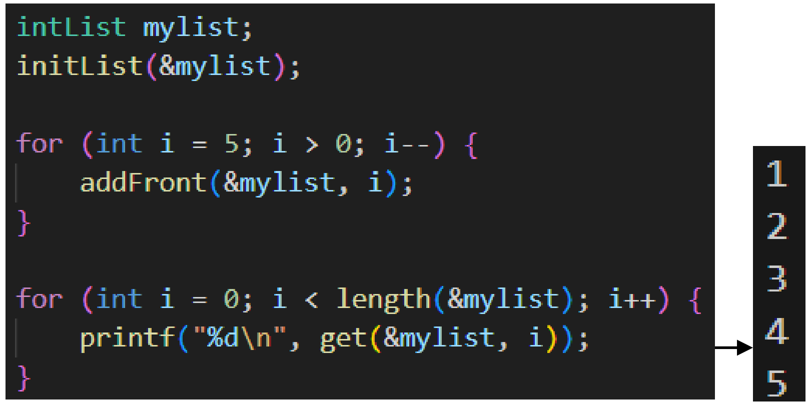
- Using a «get» function, we have the opportunity to iterate through the elements.
- Using a length function, a chance to an outer intervention could be avoided.



```c
int get(intList *list, int position) {
    if (position < 0 || position >= list->elemcount) {
        printf("Invalid position\n");
        return -1;
    }

    intNode *position_pointer = list->head;
    for (int index = 0; index < position; index++) {
        position_pointer = position_pointer->next;
    }
    return position_pointer->data;
}
```



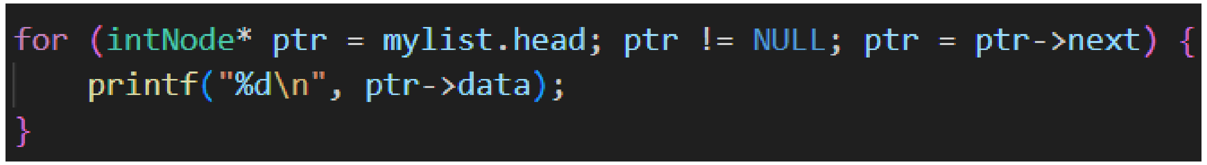
```c
int length(intList *list) {
    return list->elemcount;
}
```



```c
intList mylist;
initList(&mylist);

for (int i = 5; i > 0; i--) {
    addFront(&mylist, i);
}

for (int i = 0; i < length(&mylist); i++) {
    printf("%d\n", get(&mylist, i));
}
```

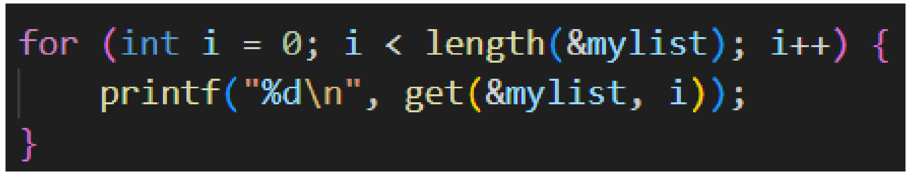
```
1
2
3
4
5
```

- However, the complexity for a traversal is increased dramatically!



```c
for (intNode* ptr = mylist.head; ptr != NULL; ptr = ptr->next) {
    printf("%d\n", ptr->data);
}
```

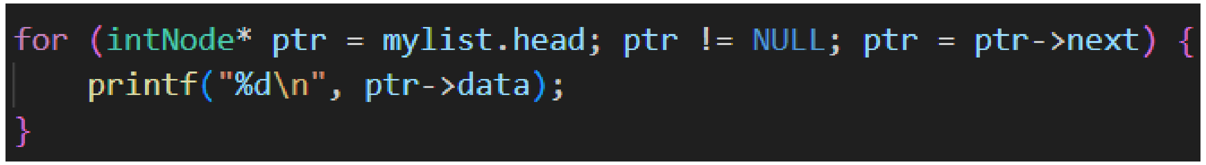
vs.



```c
for (int i = 0; i < length(&mylist); i++) {
    printf("%d\n", get(&mylist, i));
}
```

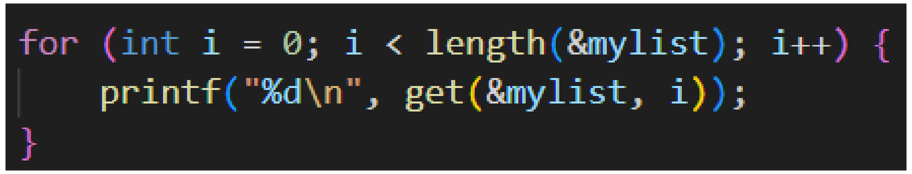
Increased cost for the «get» function

- Suppose that our list has five elements: 0,1,2,3,4.



```c
for (intNode* ptr = mylist.head; ptr != NULL; ptr = ptr->next) {
    printf("%d\n", ptr->data);
}
```

vs.



```c
for (int i = 0; i < length(&mylist); i++) {
    printf("%d\n", get(&mylist, i));
}
```

Operations

Get the 0th element, jump to next node.
Get the 1st element, jump to next node.
Get the 2nd element, jump to next node.
Get the 3rd element, jump to next node.
Get the 4th element, jump to next node.

Linear time

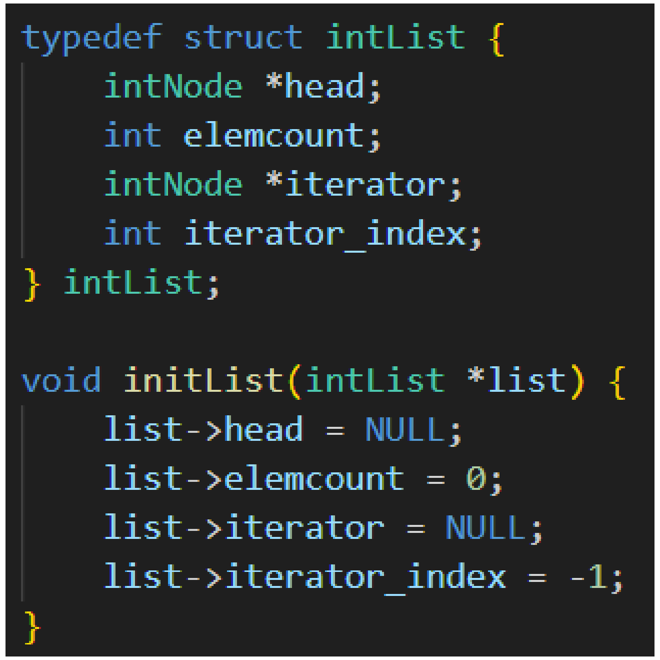
Operations

Get(0): Get the 0th element
Get(1): Jump 1 node. Get the 1st element.
Get(2): Jump 2 nodes. Get the 2nd element.
Get(3): Jump 3 nodes. Get the 3rd element.
Get(4): Jump 4 nodes. Get the 4th element.

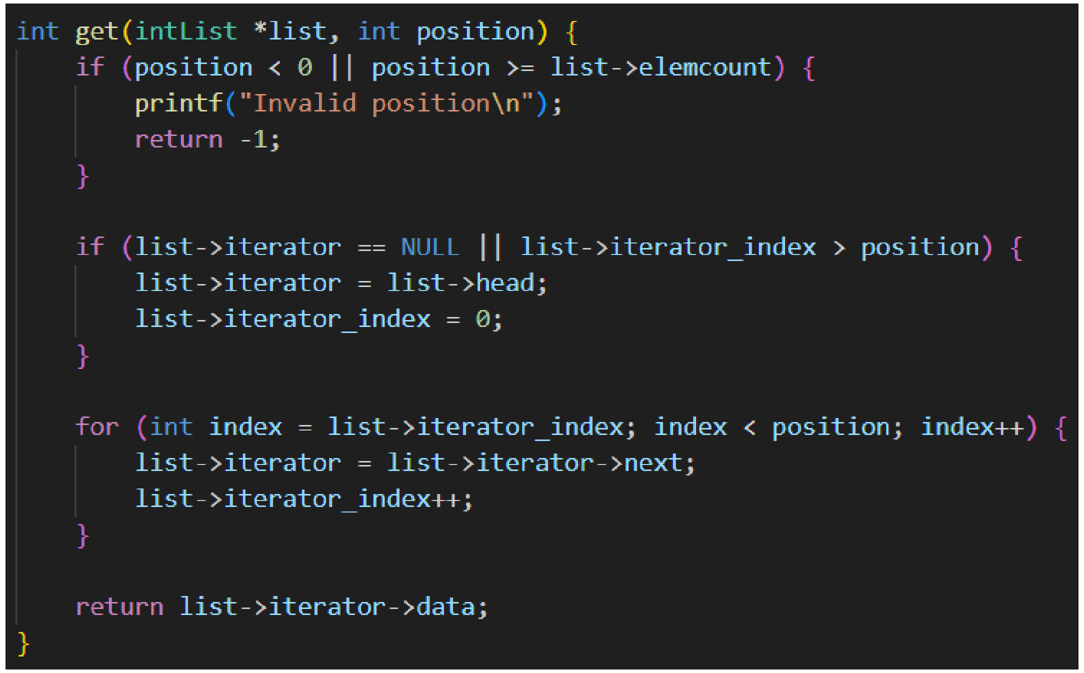
Unnecessary operations!

A simple solution

- By storing an iterator of the lastly checked index, the previous operation could be done in linear time.



```c
typedef struct intList {
    intNode *head;
    int elemcount;
    intNode *iterator;
    int iterator_index;
} intList;

void initList(intList *list) {
    list->head = NULL;
    list->elemcount = 0;
    list->iterator = NULL;
    list->iterator_index = -1;
}
```



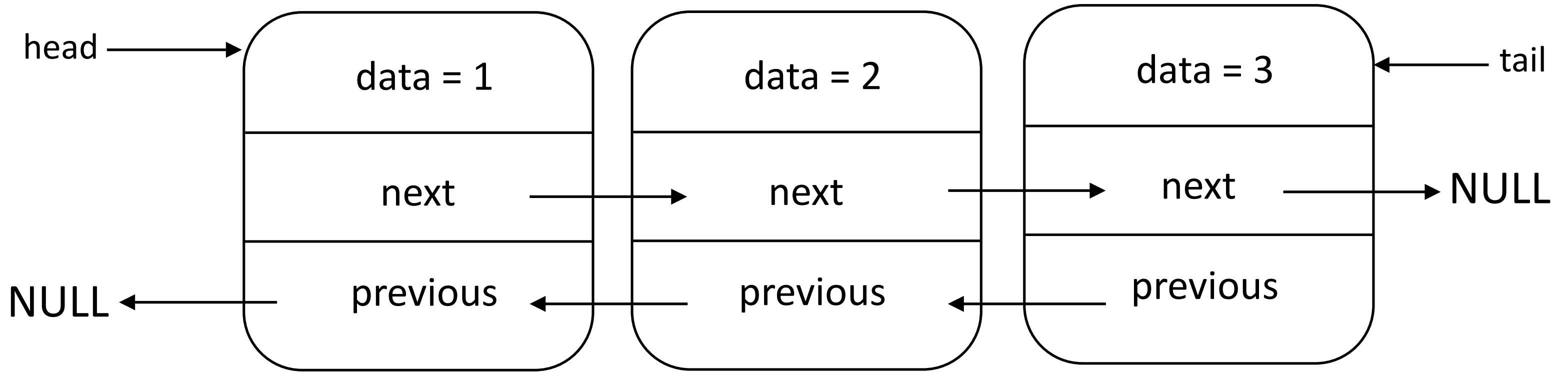
```c
int get(intList *list, int position) {
    if (position < 0 || position >= list->elemcount) {
        printf("Invalid position\n");
        return -1;
    }

    if (list->iterator == NULL || list->iterator_index > position) {
        list->iterator = list->head;
        list->iterator_index = 0;
    }

    for (int index = list->iterator_index; index < position; index++) {
        list->iterator = list->iterator->next;
        list->iterator_index++;
    }

    return list->iterator->data;
}
```

- If we could reach the element at the specified position using the iterator, we may use the iterator.
- Traversing the list changes the position of the iterator.
- Only useful for ordered indices.

Doubly Linked List

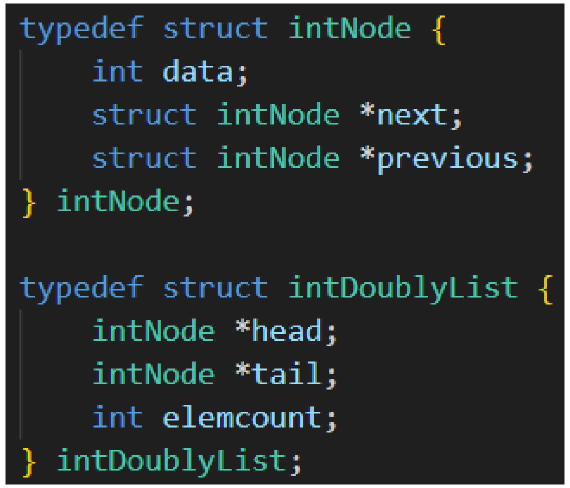
- In a singly linked list, removing any element other than the head is time consuming. To reach any element, we may start from the head node in each case.
- Even that we may use an iterator, we may start from the head over an over to reach the elements.

- Thus, it is useful to point both next and previous nodes.

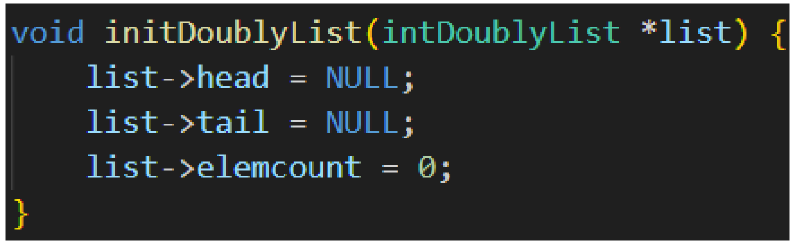


Doubly Linked List

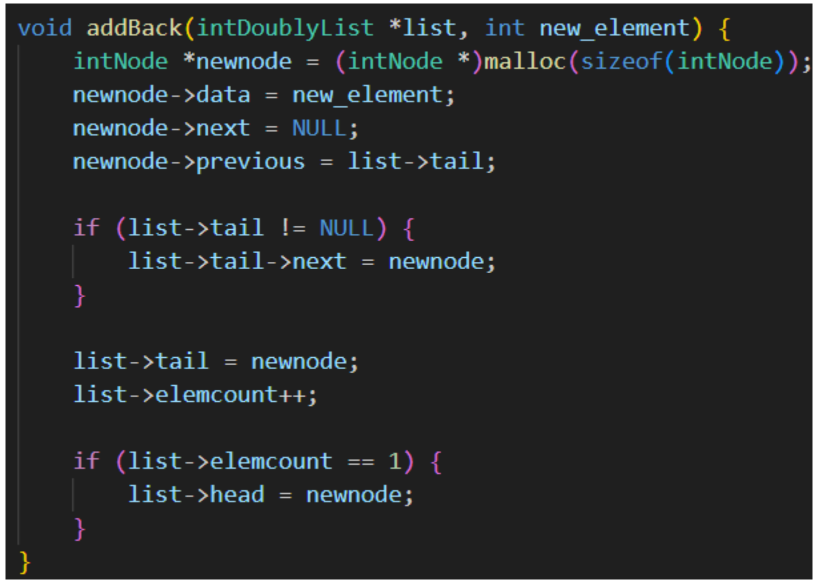
- Without considering the access limitations, we could create the doubly linked list using new functions: removeBack and addBack.



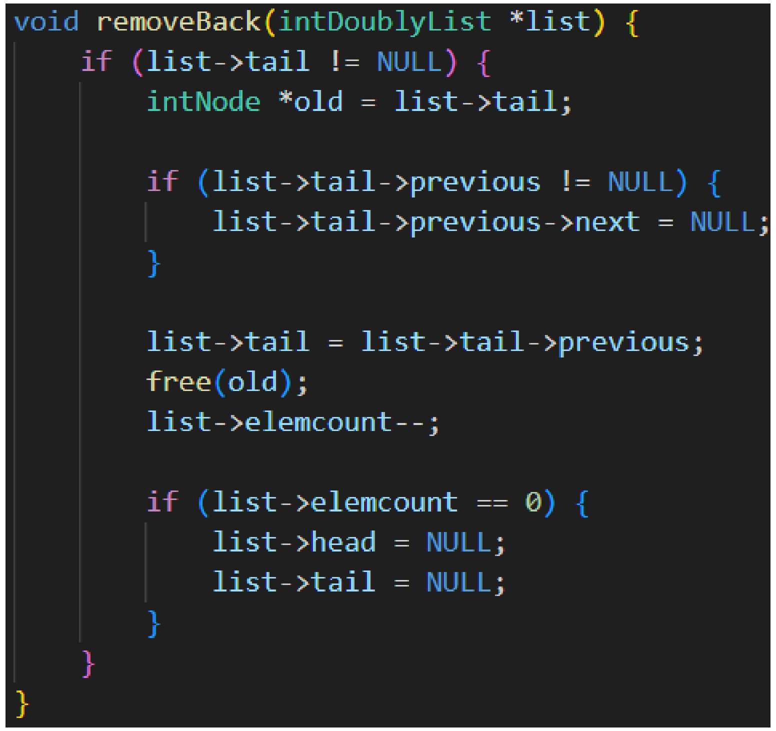
```c
typedef struct intNode {
    int data;
    struct intNode *next;
    struct intNode *previous;
} intNode;

typedef struct intDoublyList {
    intNode *head;
    intNode *tail;
    int elemcount;
} intDoublyList;
```



```c
void initDoublyList(intDoublyList *list) {
    list->head = NULL;
    list->tail = NULL;
    list->elemcount = 0;
}
```

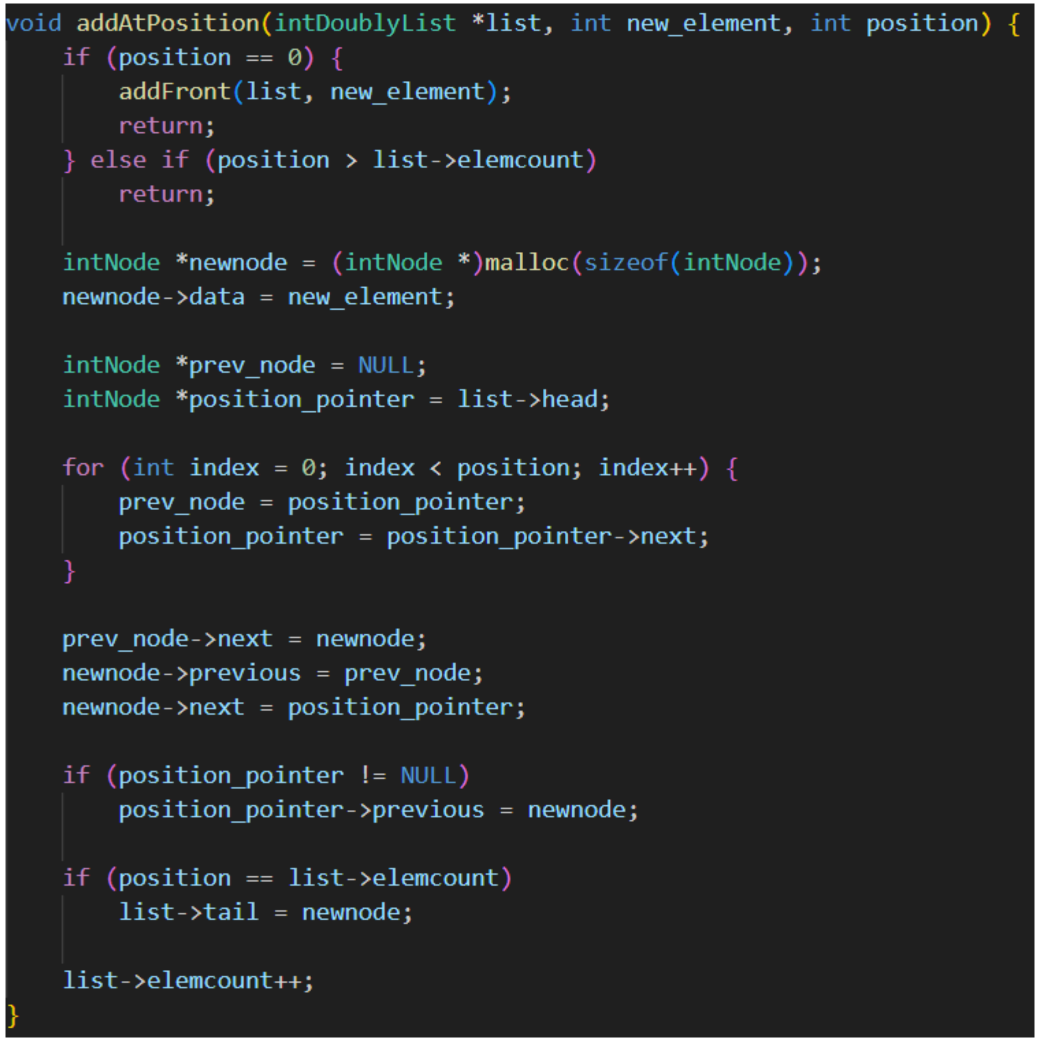


```c
void addBack(intDoublyList *list, int new_element) {
    intNode *newnode = (intNode *)malloc(sizeof(intNode));
    newnode->data = new_element;
    newnode->next = NULL;
    newnode->previous = list->tail;

    if (list->tail != NULL) {
        list->tail->next = newnode;
    }

    list->tail = newnode;
    list->elemcount++;

    if (list->elemcount == 1) {
        list->head = newnode;
    }
}
```

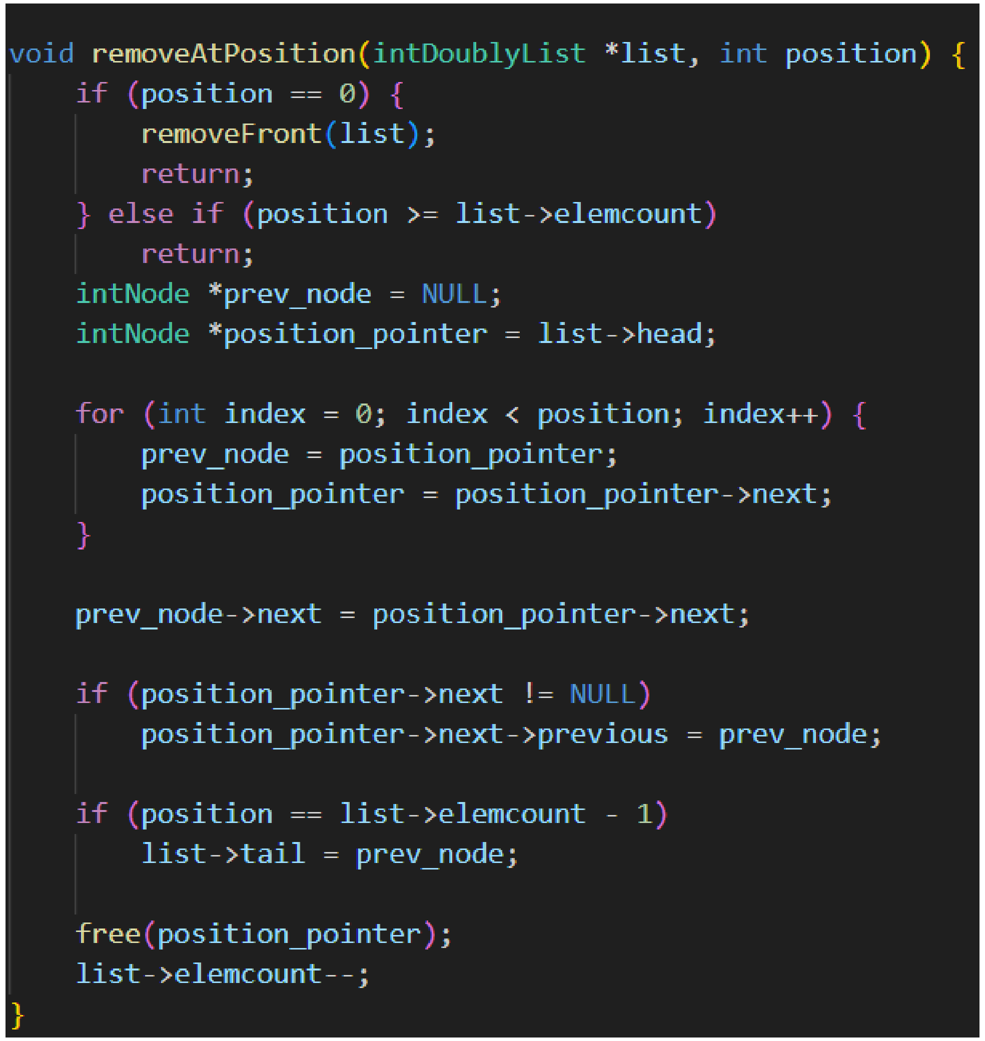


```c
void removeBack(intDoublyList *list) {
    if (list->tail != NULL) {
        intNode *old = list->tail;

        if (list->tail->previous != NULL) {
            list->tail->previous->next = NULL;
        }

        list->tail = list->tail->previous;
        free(old);
        list->elemcount--;

        if (list->elemcount == 0) {
            list->head = NULL;
            list->tail = NULL;
        }
    }
}
```

Doubly Linked List



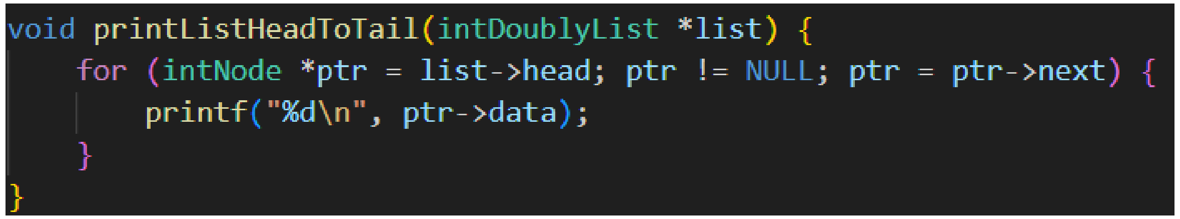
```c
void addAtPosition(intDoublyList *list, int new_element, int position) {
    if (position == 0) {
        addFront(list, new_element);
        return;
    } else if (position > list->elemcount)
        return;

    intNode *newnode = (intNode *)malloc(sizeof(intNode));
    newnode->data = new_element;

    intNode *prev_node = NULL;
    intNode *position_pointer = list->head;

    for (int index = 0; index < position; index++) {
        prev_node = position_pointer;
        position_pointer = position_pointer->next;
    }

    prev_node->next = newnode;
    newnode->previous = prev_node;
    newnode->next = position_pointer;

    if (position_pointer != NULL)
        position_pointer->previous = newnode;

    if (position == list->elemcount)
        list->tail = newnode;

    list->elemcount++;
}
```



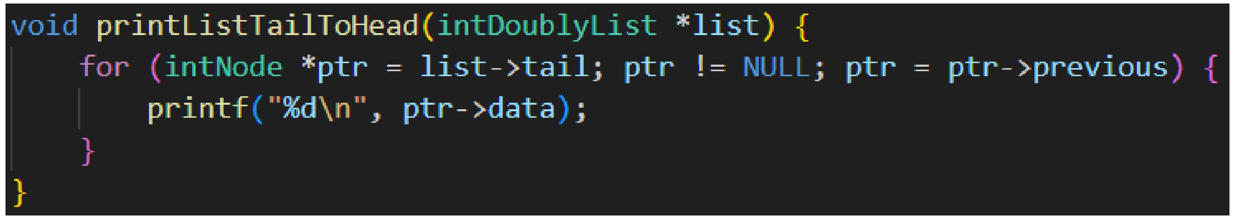
```c
void removeAtPosition(intDoublyList *list, int position) {
    if (position == 0) {
        removeFront(list);
        return;
    } else if (position >= list->elemcount)
        return;
    intNode *prev_node = NULL;
    intNode *position_pointer = list->head;

    for (int index = 0; index < position; index++) {
        prev_node = position_pointer;
        position_pointer = position_pointer->next;
    }

    prev_node->next = position_pointer->next;

    if (position_pointer->next != NULL)
        position_pointer->next->previous = prev_node;

    if (position == list->elemcount - 1)
        list->tail = prev_node;

    free(position_pointer);
    list->elemcount--;
}
```

Iterating a Doubly Linked List

- Because of having double connections, the following two code blocks both processed in linear time.



```c
void printListHeadToTail(intDoublyList *list) {
    for (intNode *ptr = list->head; ptr != NULL; ptr = ptr->next) {
        printf("%d\n", ptr->data);
    }
}
```



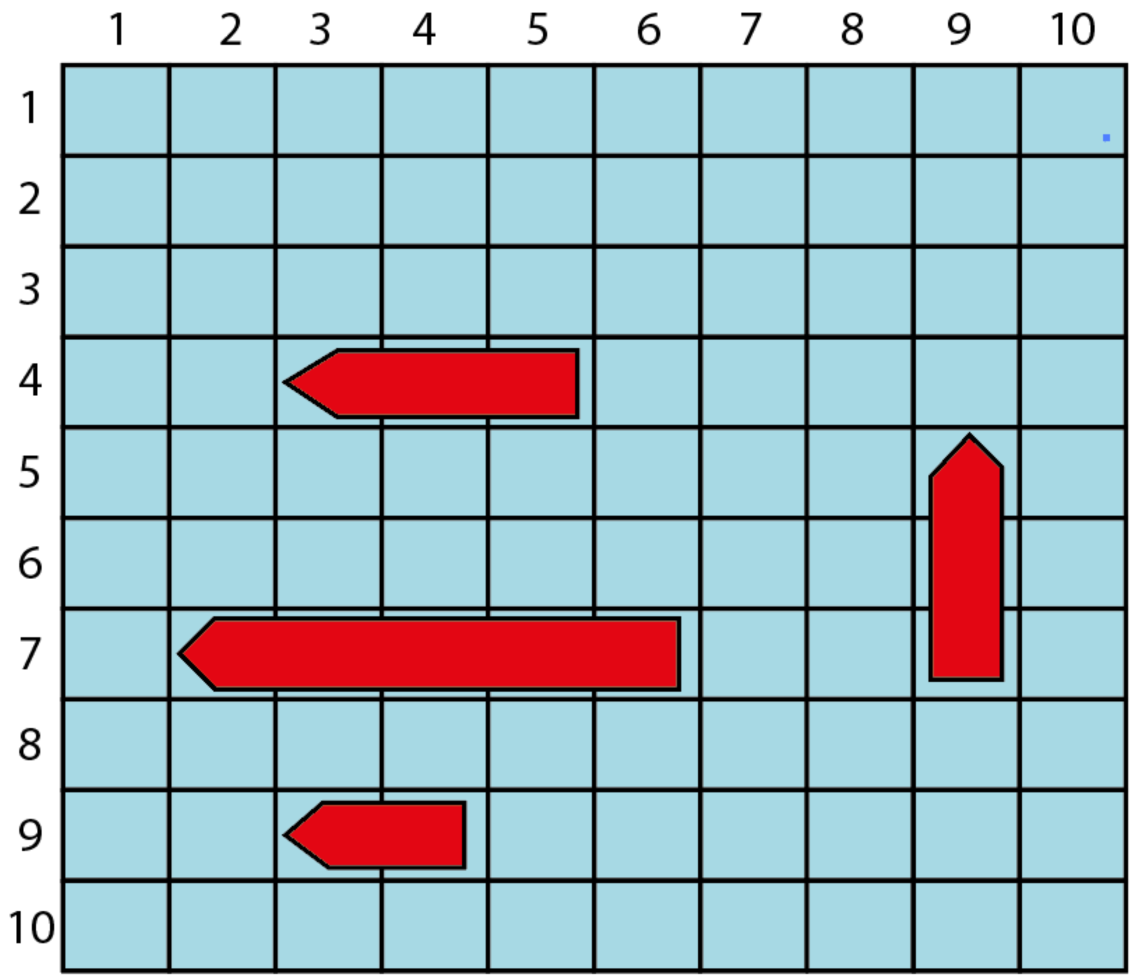
```c
void printListTailToHead(intDoublyList *list) {
    for (intNode *ptr = list->tail; ptr != NULL; ptr = ptr->previous) {
        printf("%d\n", ptr->data);
    }
}
```

Ex: Battleship Game

- In the single-player mode of the battleship game, you play as a general with a cannon and limited ammunition, tasked with attacking enemy ships positioned on tiles of a 2D map.
- If you hit a battleship, it sinks. If all the ships are sunk on the map, the player wins the game.

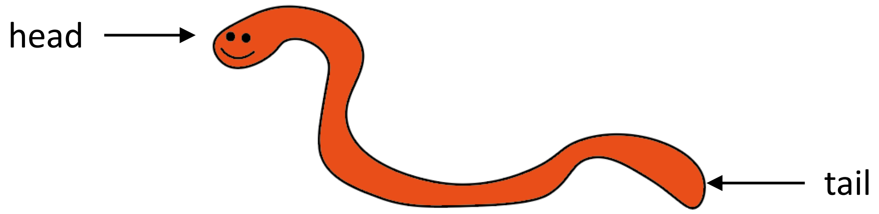
For the given map:

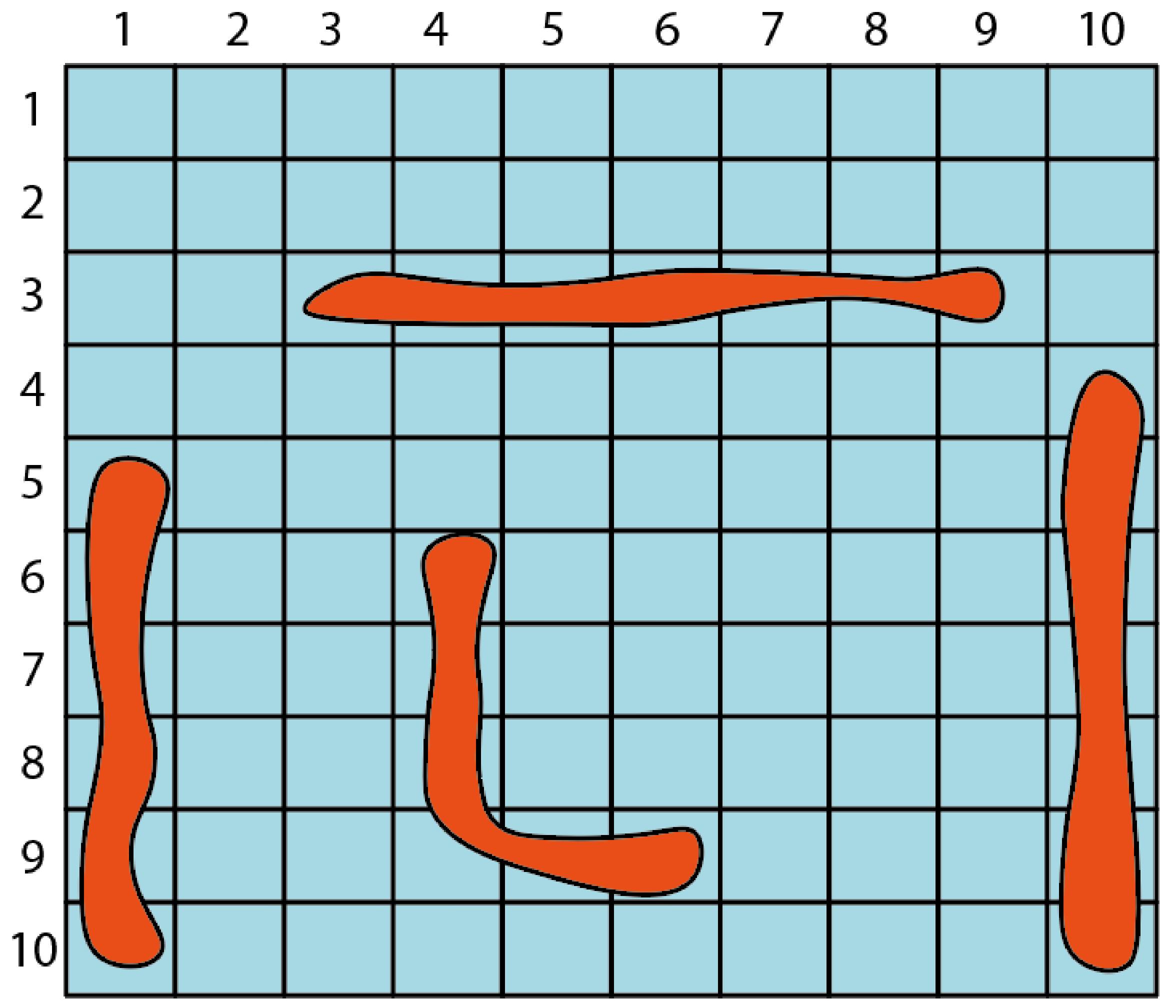
```
>(4,2)
Miss. Remaining ships: 4.
>(4,3)
Hit! Remaining ships: 3.
…
```



Battleworm Game!

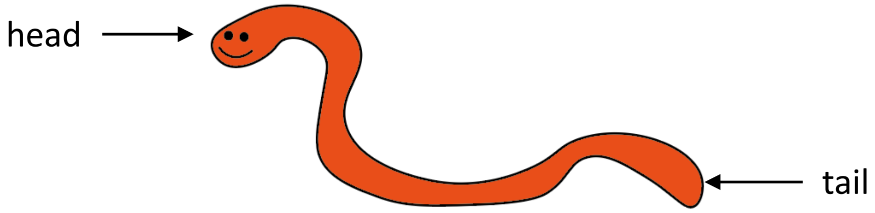
- We may change the game and the rules.
- In the BattleWorm™ game, you play as a farmer attacking the worms in the garden.

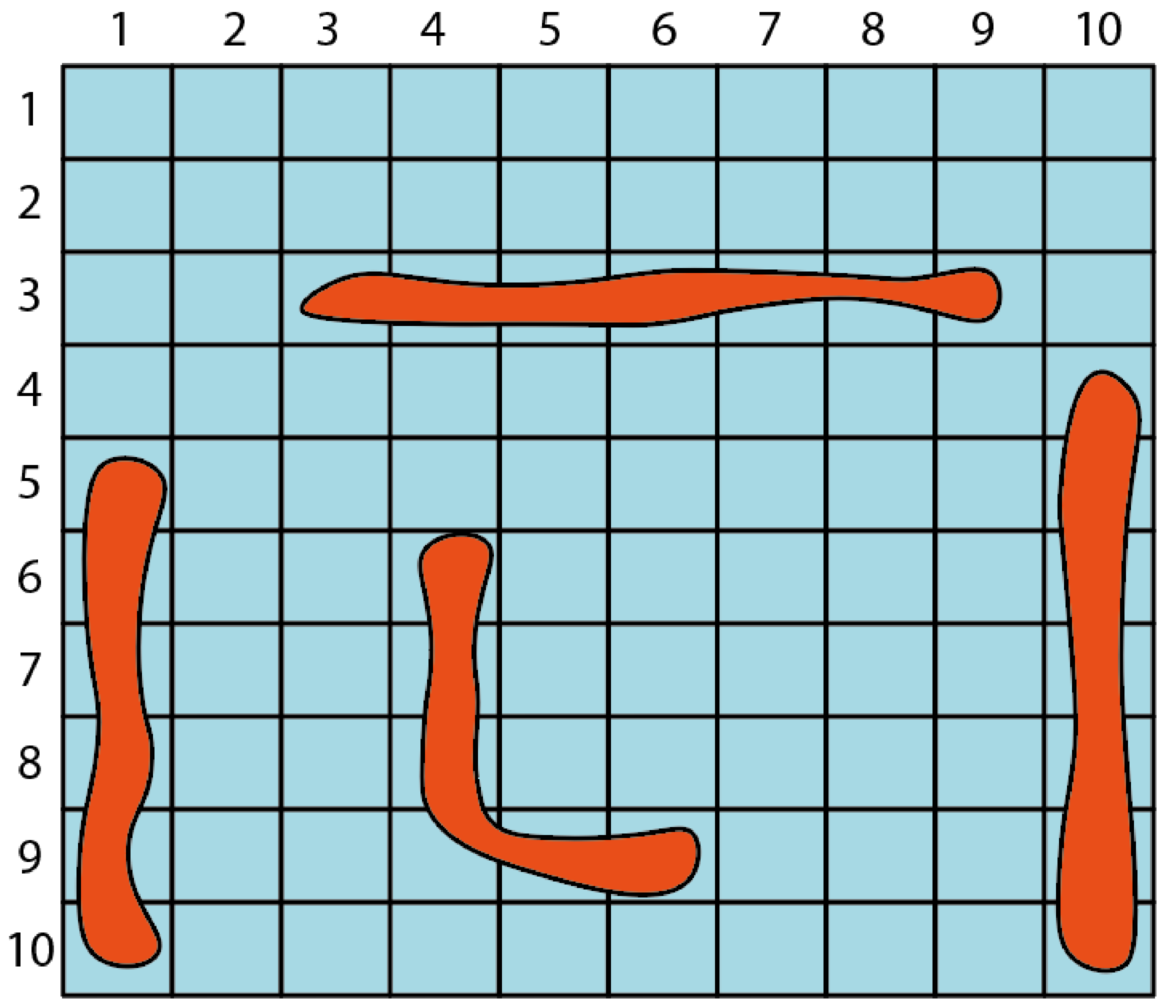




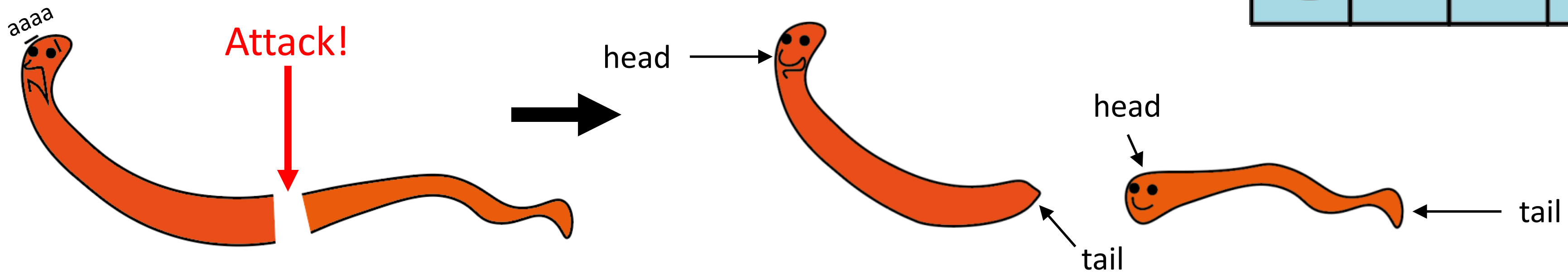
Battleworm Game!

- We may change the game and the rules.
- In the BattleWorm™ game, you play as a farmer attacking the worms in the garden.



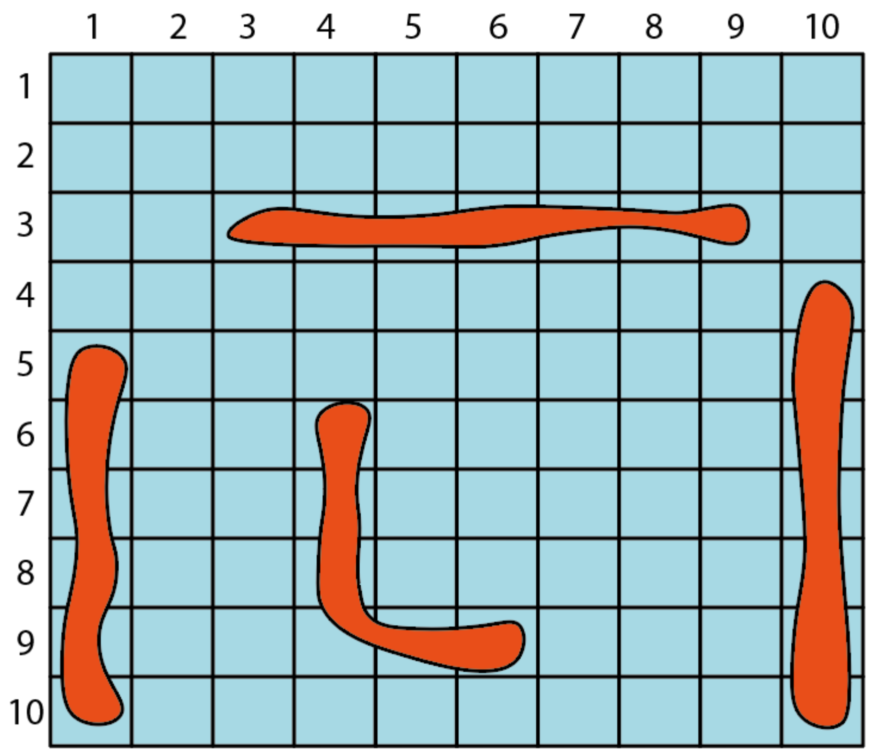


Attack on a worm may create two new worms!

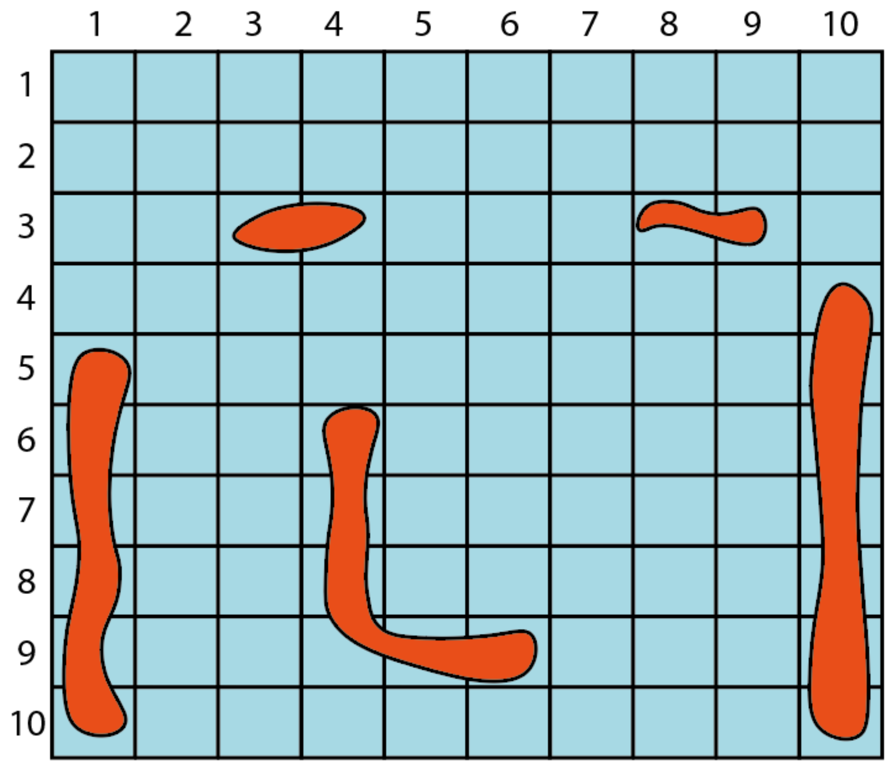


Battleworm Game!

- Attacking a tile of a worm will destroy its parts from previous and next tiles.

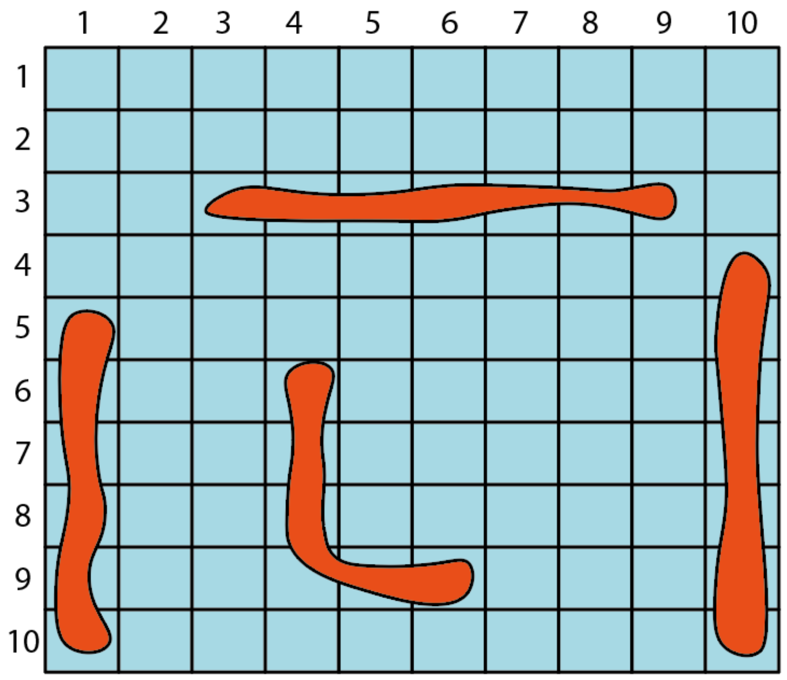


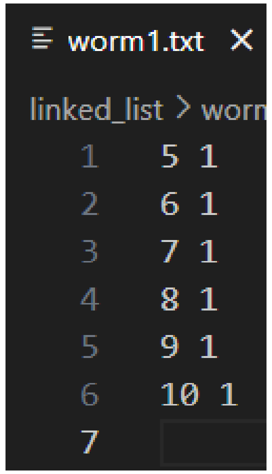
Attack (3,6)!



Battleworm Game!

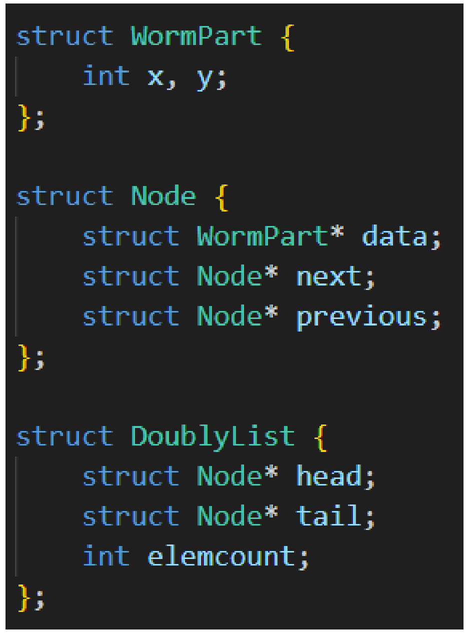
- With the skeleton code, text files for worm coordinates are also given.





```
5 1
6 1
7 1
8 1
9 1
10 1
```

- A tile of a worm could be stored in a WormPart data structure.
- A worm is a list of WormParts.



```c
struct WormPart {
    int x, y;
};

struct Node {
    struct WormPart* data;
    struct Node* next;
    struct Node* previous;
};

struct DoublyList {
    struct Node* head;
    struct Node* tail;
    int elemcount;
};
```