CMPT 125: Introduction to Computing Science and Programming II
Fall 2023

Week 7: Data structures, revisiting recursion

Instructor: Victor Cheung, PhD

School of Computing Science, Simon Fraser University

Fact of the day

The world's first webcam was used to watch a coffee pot to see if it is empty

https://www.bbc.com/news/technology-20439301

The webcam takes images 3 times a minute and uploads them

Recap from Last Lecture

- Implementation details
  - Stacks – making them resizable using realloc
  - Queues – using all the available spaces in a static array, making them resizable using malloc
  - Linked Lists – making use of memory allocation to create a data structure that stores items flexibly, with O(1) time complexity for many operations
    There is one case where the operation is O(N) given the current implementation:
    when we want to remove a node at the end (need to access the second-last node to update its next)

Review from Last Lecture

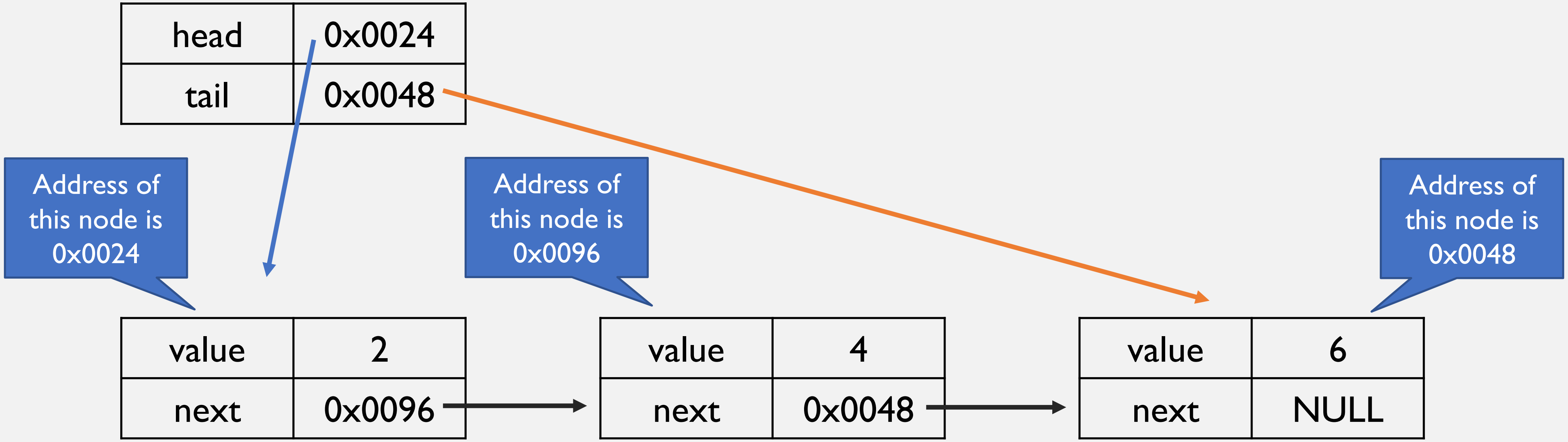
- Implement the queue ADT in C using an int array of size 10 (Size10Queue)
  - **General idea**: careful with the indexing and select a remove strategy that makes the most sense
  - Don't forget to insert a check for null pointers for the create/isEmpty/enqueue/dequeue functions

Today

- Linked Lists (cont’d)
  - Some implementation details (cases need to be considered)
- Live code demo
  - Linked list ADT

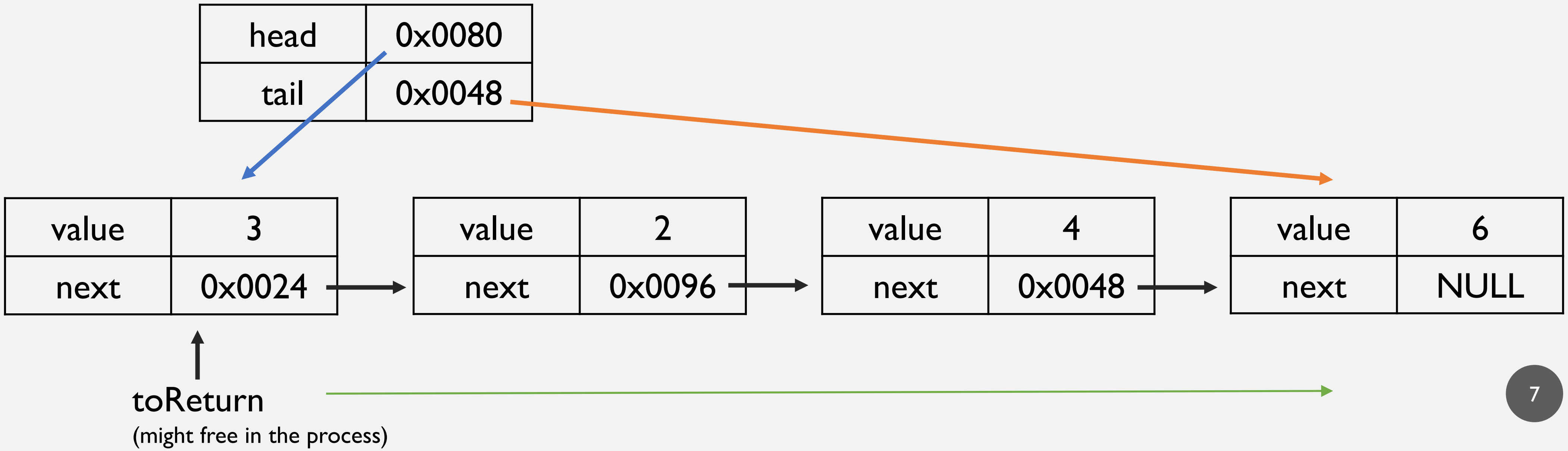
Linked Lists Structure (from Last Lecture)

- To store a linked list in C, we add an extra composite data type that works as a header that remembers the first and last node of the list (we call them head & tail)
  - Some implementations will only have the head, which is sufficient. But also having tail makes some functions faster



Linked Lists – Basic Operations (6)

- Search: Given the key value, do a linear search (*linked lists are not sorted by default*)
  - Create a node pointer called visitor, traverse from head and check each value using a while-loop to advance
  - When found, return the address of that node; if goes to the end of the list (visitor becomes NULL), not found, return NULL

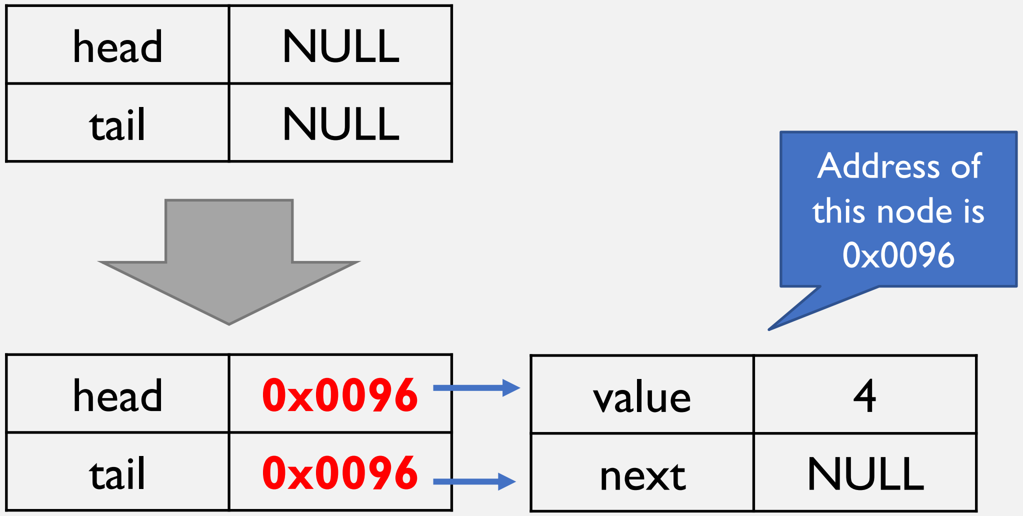


Implementation Details of Linked List

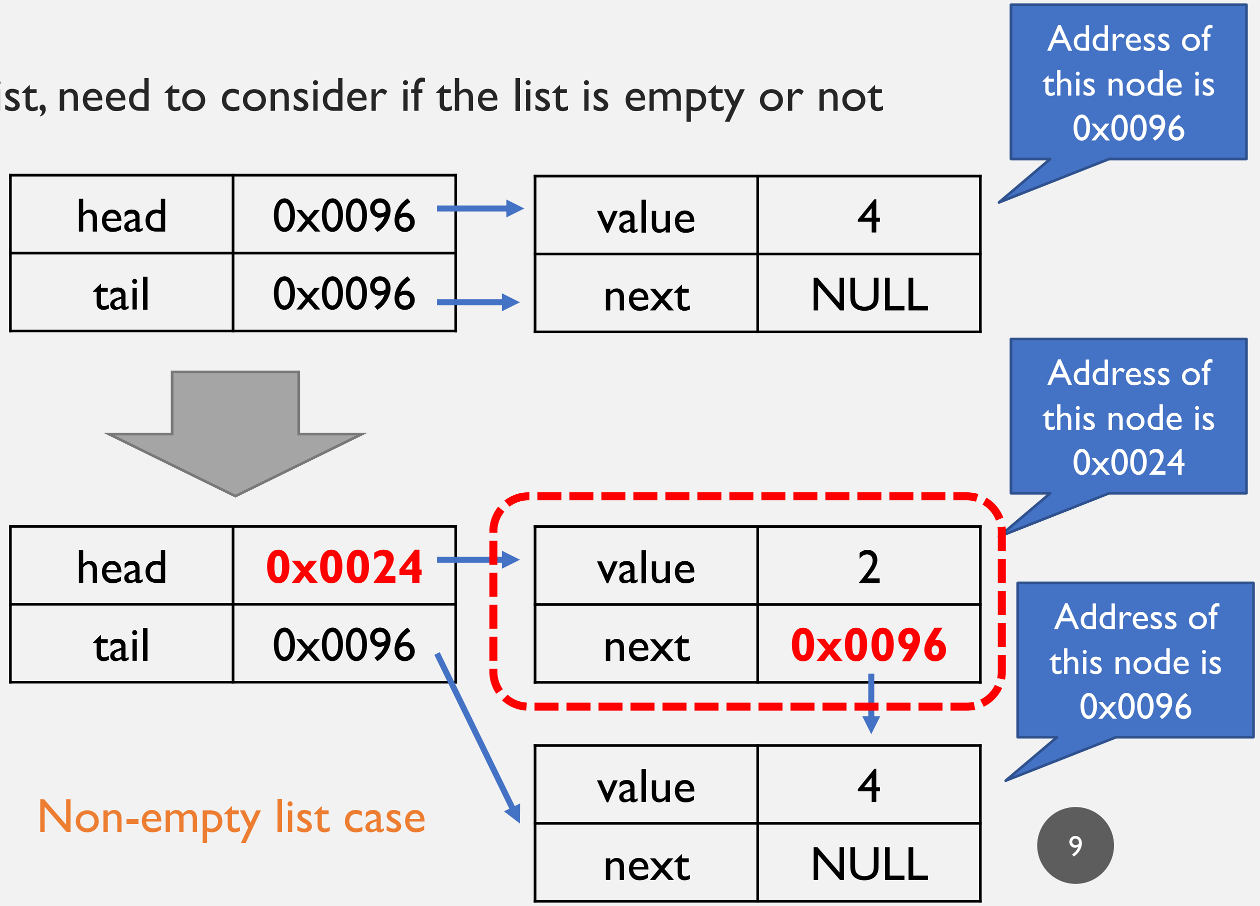
Cases to consider…

2 Cases for InsertFront

- When inserting a node into the front of the linked list, need to consider if the list is empty or not



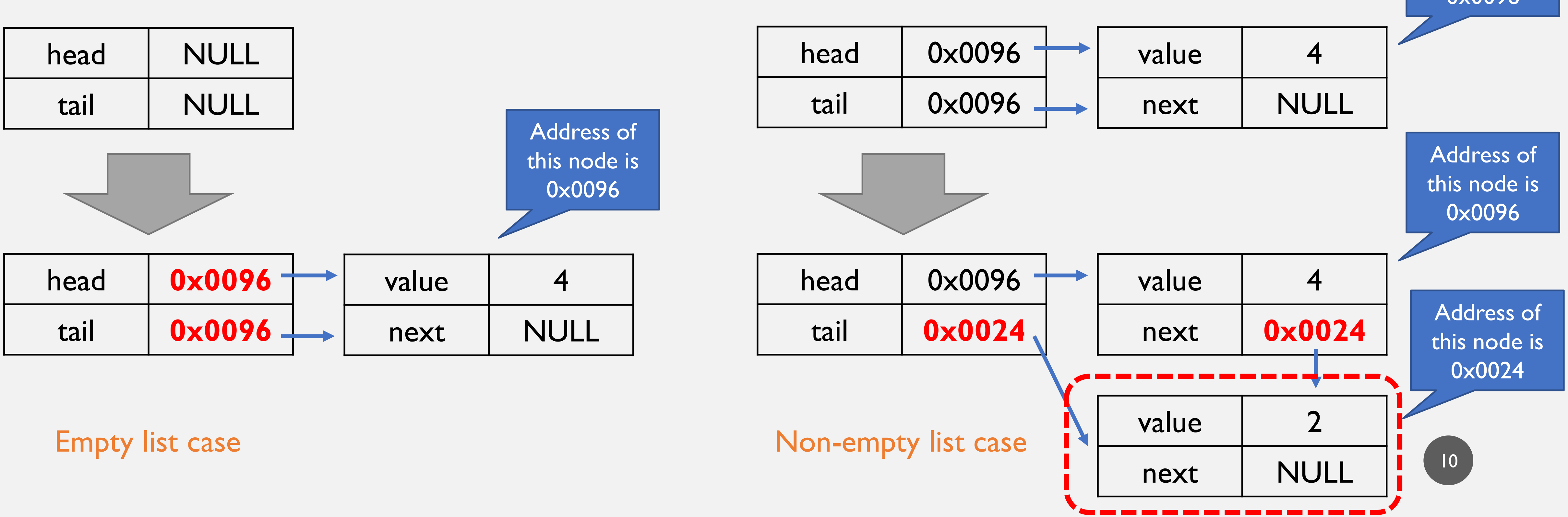
Empty list case



Non-empty list case

2 Cases for InsertEnd

- When inserting a node into the end of the linked list, also need to consider if the list is empty or not

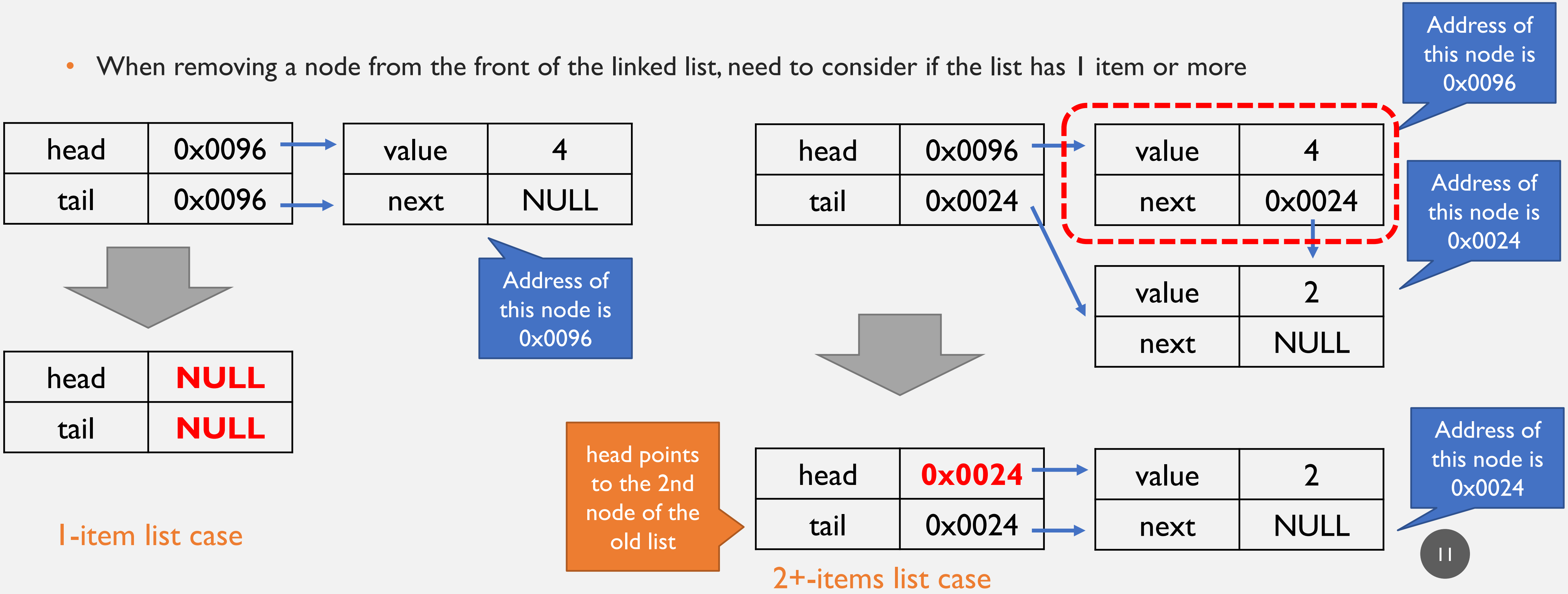


Empty list case

Non-empty list case

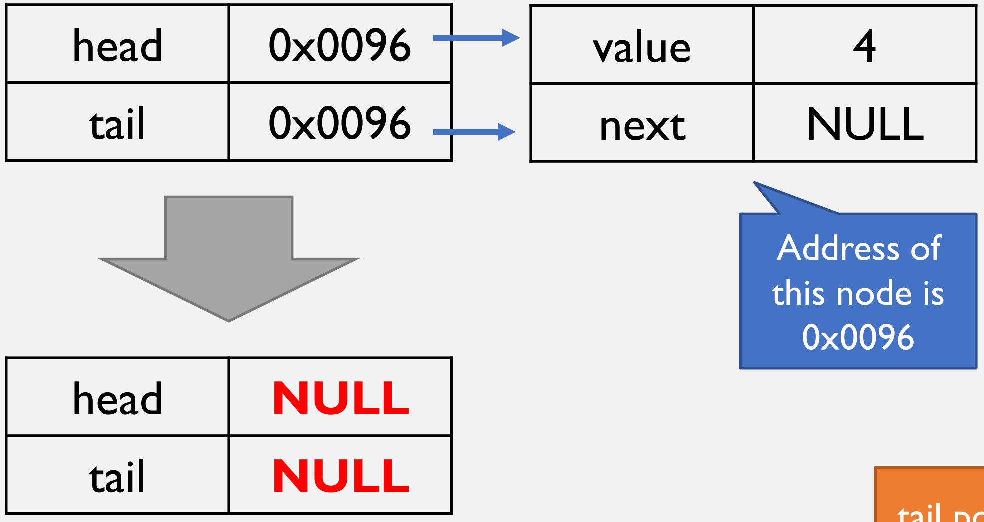
2 Cases for RemoveFront

- When removing a node from the front of the linked list, need to consider if the list has 1 item or more

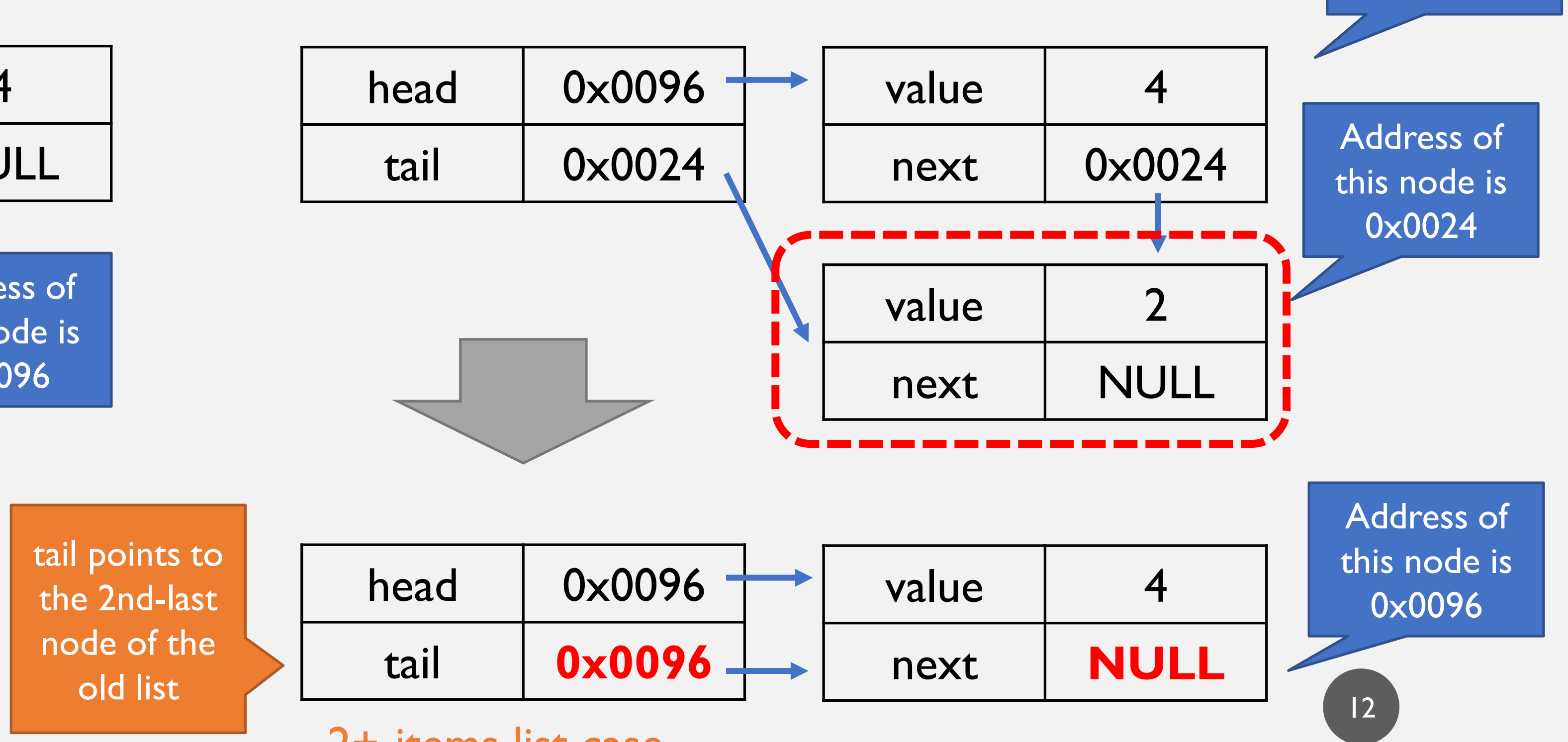


2 Cases for RemoveEnd

- When removing a node from the end of the linked list, also need to consider if the list has 1 item or more



1-item list case



2+-items list case

How to Find The 2nd-Last Node?

- Main idea: consider the fact that the next node of the 2nd-last node is the last node
  - **Option 1**: check if a node's next is the last node
  - **Option 2**: check if the next pointer of a node's next node is NULL

- This takes O(n) time. Some variations of linked lists allow you to find it faster

```
//myList points to the header of a linked list
if (myList->head == NULL) {//empty list
  //do something for empty list
} else if (myList->head == myList->tail) {//1 node
  //do something for a 1-node list
} else {//2 or more nodes
  //start from the front
  Node_t* secondLast = myList->head;

  while (secondLast->next != myList->tail) {
    //follow the trail
    secondLast = secondLast->next;
  }

  //do something with the secondLast node
}
```

Performance of Linked List Operations

- Most operations with linked list is either constant time (a fixed number of steps), or linear time (number of steps proportional to the size of the list)
  - create – O(1) because there is a fixed amount of steps
  - adding/removing elements at the front or end – O(1)
  - search – O(n) because you are doing a linear search
- When compared to arrays:
  - adding/removing elements anywhere – O(n) since you might have to resize and copy all the values
  - search – O(n) because you are also doing a linear search

- In general traversing though a linked list takes more time because the program needs to trace the address of the nodes, where with arrays it is just pointer arithmetic due to its continuous memory allocation
- Linked lists can grow (and shrink) in size by inserting (and removing) more nodes and no realloc is needed

Typical Linked Lists Implementation in C

- The node is a struct

```
typedef struct Node {
  int value;
  struct Node* next;
} Node_t;
```

- The header is also a struct

```
typedef struct {
  Node_t* head;
  Node_t* tail;
} LList_t;
```

Careful: NOT the size of a node

```
//start with a list
LList_t* myList = malloc(sizeof(LList_t));
myList->head = NULL;
myList->tail = NULL;

//add a node, typically done in a function
myList->head = malloc(sizeof(Node_t));
myList->head->value = 0; //access field of the first node
myList->head->next = NULL; //access field of the first node
myList->tail = myList->head;

//don't forget to free the memory created if not used
free(myList->head);
free(myList);
```

Commonly Used Code in Linked Lists

- Very often you'll traverse a linked list, for example:
  - to find a node storing a certain value (search)
  - to add a node some where in-between
  - to remove a node some where in-between
  - to print its content
  - …etc.

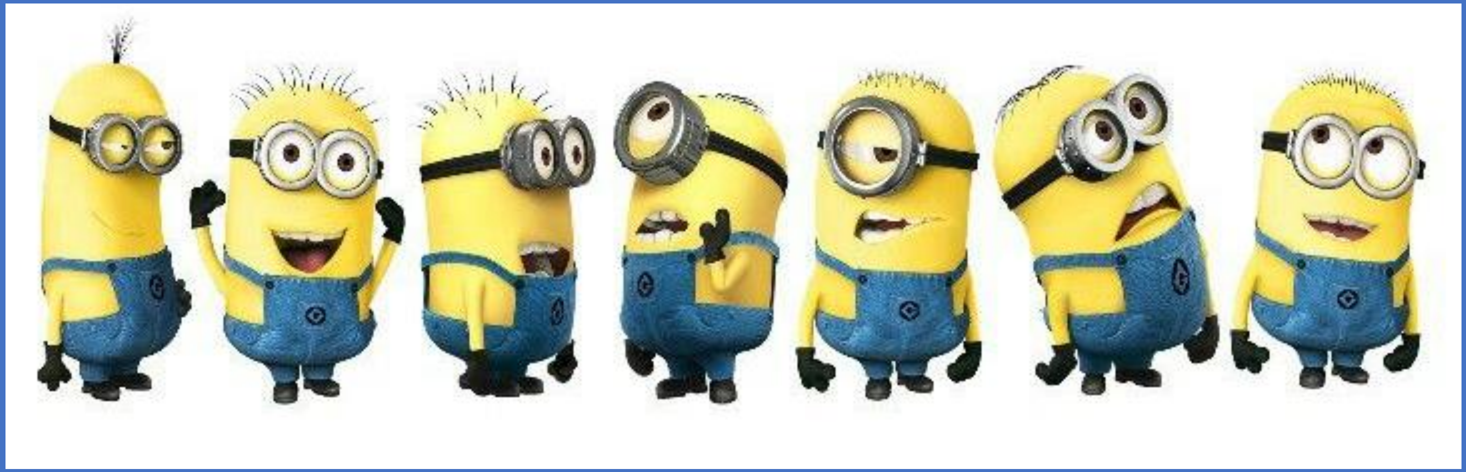
```
//start from the front
Node_t* current = myList->head;

while (current != NULL) {//stop when there is nothing

    //do something with the current value, e.g., print
    printf("%d ", current->value);

    //follow the trail
    current = current->next;
}
```

Break for 10 Minutes



Putting It All Together

Live coding!

Things We Need

- A struct for Node (Node_t) + a struct for LinkedList (LList_t)
- A function to create an empty list (LL_create), returns the address of the list
- A function to free up the list (LL_free)
- A function to add an int value to the front of the list as a node (LL_insertFront)
- A function to remove the node in the front of the list (LL_removeFront), returns the value
- A function to tell if the list is empty (LL_isEmpty), returns 1 if empty, 0 otherwise

- Bonus 1: A function to print the content of the list (LL_print)
- Bonus 2: A function to get the length of the list (LL_length), returns the number of nodes in the list

```c
typedef struct Node {
    int value;
    struct Node* next;
} Node_t;
```

```c
typedef struct {
    Node_t* head;
    Node_t* tail;
} LList_t;
```

```c
LList_t* LL_create() {
    LList_t* theList = malloc(sizeof(LList_t));
    theList->head = NULL;
    theList->tail = NULL;
}
```

It should also do a null test and return theList in the end

```c
void LL_free(LList_t* theList) {
    if (theList == NULL) { return; }

    Node_t* current = theList->head;
    while(current != NULL) {
        Node_t* toFree = current;
        current = current->next;
        free(toFree);
    }
    free(theList);
}
```

```c
void LL_insertFront(LList_t* theList, int value) {
    if (theList == NULL) { return; }

    Node_t* newNode = malloc(sizeof(Node_t));
    newNode->value = value;
    newNode->next = NULL;

    if (theList->head == NULL) {//empty list
        theList->head = newNode;
        theList->tail = newNode;
    } else {//non-empty list
        newNode->next = theList->head;
        theList->head = newNode;
    }
}
```

```c
int LL_removeFront(LList_t* theList) {
    if (theList == NULL || theList->head == NULL) {
        return -99999; //this means error
    }

    Node_t* toRemove = theList->head;
    if (theList->head == theList->tail) {//1 item
        theList->head = NULL;
        theList->tail = NULL;
    } else {//2+ items
        theList->head = theList->head->next;
    }
    int theValue = toRemove->value;
    free(toRemove); //for this implementation
    return theValue;
}
```

```c
int LL_isEmpty(LList_t* theList) {
    if (theList == NULL || theList->head == NULL) {
        return 1;
    } else {
        return 0;
    }
}
```

We'll do LL_print and LL_length if there's time otherwise they'll be an exercise ☺

```c
void LL_insertEnd(LList_t* theList, int value) {
    if (theList == NULL) { return; }

    Node_t* newNode = malloc(sizeof(Node_t));
    newNode->value = value;
    newNode->next = NULL;

    if (theList->head == NULL) {//empty list
        theList->head = newNode;
        theList->tail = newNode;
    } else {//non-empty list
        theList->tail->next = newNode;
        theList->tail = newNode;
    }
}
```

```c
int LL_removeEnd(LList_t* theList) {
    if (theList == NULL || theList->head == NULL) {
        return -99999; //this means error
    }

    Node_t* toRemove = theList->tail;
    if (theList->head == theList->tail) {//1 item
        theList->head = NULL;
        theList->tail = NULL;
    } else {//2+ items
        Node_t* secondLast = theList->head;
        while (secondLast->next != theList->tail) {
            secondLast = secondLast->next;
            printf("on the move\n");
        }
        theList->tail = secondLast;
        secondLast->next = NULL;
    }
    int theValue = toRemove->value;
    free(toRemove); //for this implementation
    return theValue;
}
```

Find the 2nd-last node

Today's Review

- Linked Lists
  - A data structure that stores items in a specific way to provide flexible sizes with O(1) time complexity for many operations
  - More efficient implementation of ADTs – using insertFront/insertEnd/removeFront/removeEnd
    - Need to consider the state of the linked lists for different cases (empty/1-item/2+-items)
- Code live demo
  - Using structs and pointers to implement the list
  - Different functions accessing the list via a pointer to it
    - check for null, check for empty/1-item/2+-items
    - use while-loop to traverse

Homework!

- Implement the rest of the linked list data structure
  - insertEnd, removeEnd (*you'll need the trick to find the 2nd-last node*)
  - move the code to LinkedList.h and LinkedList.c
- Write some testing code to use the linked list
- Think about how to insert/delete a node to/from the inside (not head/tail) of a linked list
  - We'll talk about this next week
- Think about how linked list is related to Stacks & Queues
  - We'll also talk about this next week