CMPT 125: Introduction to Computing Science and Programming II

Fall 2023

Week 7: Data structures: Stacks, Queues, & Linked Lists

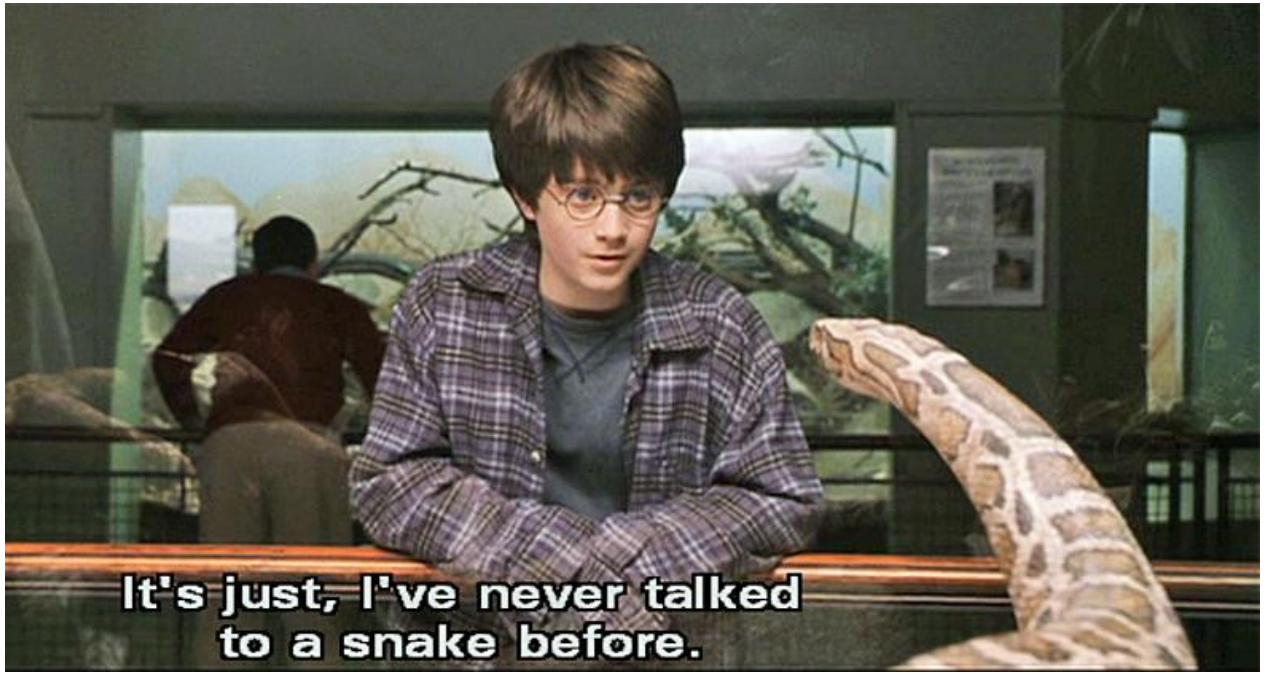
Instructor: Victor Cheung, PhD

School of Computing Science, Simon Fraser University

Joke of the day

Why would Harry Potter make a good programmer?

Because he speaks Python.



Harry Potter and the Sorcerer's Stone (2001)

Recap from Last Lecture

- Theoretical lower bound of comparison-based sorting algorithms – minimal height of a binary decision tree: O(NlogN)
- Radix Sort – a non-comparison-based sorting algorithm that can run faster than O(NlogN) under some conditions
- Abstract Data Types (ADTs) – defined by the data it can represent + the operations being supported
- The use of interfaces in ADTs makes implementation easier and follows principles in software engineering
- Examples of commonly used ADTs
  - Stack – provides access to data in a LIFO manner
  - Queue – provides access to data in a FIFO manner
  - Dynamic Array – provides access to data like an array but has no limit in size
  - Set – provides access to data without limit in size but provides no positional information

Review from Last Lecture (1)

- Perform a Radix Sort on the following array [23, 1, 11, 125, 3, 99, 41, 1000]

	41								
	11		3						
1000	1		23		125				99

[1000, 1, 11, 41, 23, 3, 125, 99]

3									
1		125							
1000	11	23		41					99

[1000, 1, 3, 11, 23, 125, 41, 99]

Review from Last Lecture (1 cont'd)

99									
41									
23									
11									
3									
1									
1000	125								

→ [1000, 1, 3, 11, 23, 41, 99, 125]

?

→ [1, 3, 11, 23, 41, 99, 125, 1000]

Review from Last Lecture (2)

- We mentioned that stacks can be used by a text editor to check if brackets are balanced. We can do further by checking if the types of brackets match. For example, [()] has the brackets balanced and matched by type, while [(]) has the brackets balanced but not matched by type. Think about how you can implement this feature
  - General idea:
    - Use the same strategy of pushing opening brackets into the stack and popping items when a closing bracket is detected
    - When an item is popped, check if they are of the same type
      - For example, [and then (are pushed into the stack, and when) is detected, (is popped and it matches with)
      - But if] is detected instead and (is popped, the brackets don't match by type

What Is Segmentation Fault?

- Also known as access violation, happens when a program attempts to
  - access a memory location that it is not allowed to access, or
  - access a memory location in a way that is not allowed
- Typical causes:
  - Writing to a read-only memory (e.g., setting a character in a constant C string variable: char* s = "hello")
  - Accessing out-of-bound array item (e.g., using index 10 to access an array with size 10)
  - Buffer/stack overflow (e.g., infinite loops/recursive function calls)
- Typical ways to find the causes:
  - print something in a loop/function, for example, the index/iterator
  - use a debugger

Reference: https://en.wikipedia.org/wiki/Segmentation_fault

Using Valgrind to Check Memory Usage

- Valgrind is a framework for building dynamic analysis tools, one such tool is to check for memory management, and in particular if there is any memory leak (e.g., memory allocated by malloc but not freed anywhere)
- For our assignments, we check for memory leak using the --leak-check=yes option, for example:
  `valgrind --leak-check=yes ./test3`
- What we want to see is in the bottom of the output of the command is "**in use at exit: 0 bytes in 0 blocks**"

```
==349326==
==349326== HEAP SUMMARY:
==349326==     in use at exit: 0 bytes in 0 blocks
==349326==   total heap usage: 392 allocs, 392 frees, 83,276 bytes allocated
==349326==
==349326== All heap blocks were freed -- no leaks are possible
```

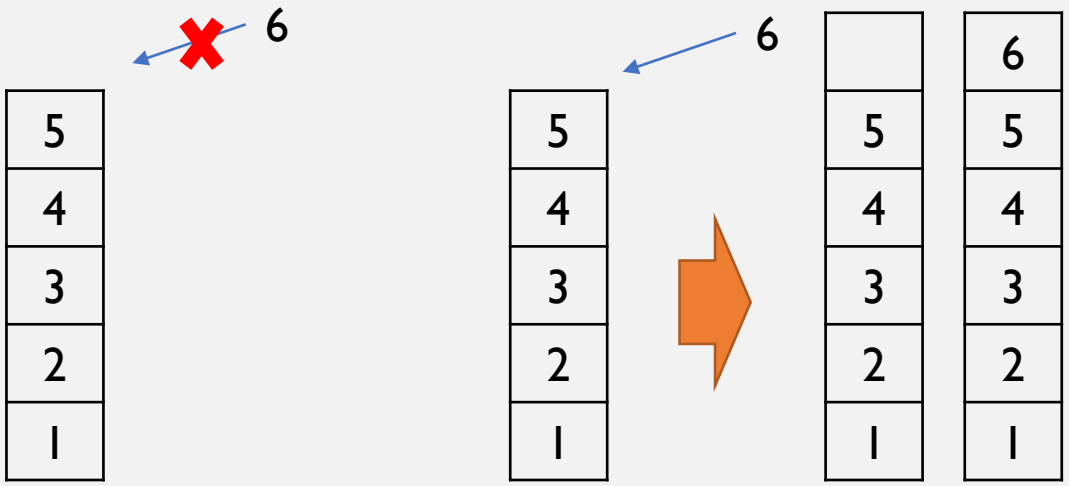
- To see more in-depth analysis, compile your code using the -g option (to generate debugging symbols), then run Valgrind on the program with both --leak-check=yes and --track-origins=yes included in the command

Today

- Implementation details
  - Stacks
  - Queues
  - Linked Lists

Revisiting Stack

- Our previous implementation of stack (Size5Stack) is limited to only 5 items because internally the storage is achieved by a **static array**
- We can replace the internal storage mechanism using malloc/realloc so when we have more items we can "grow" the stack capacity




```
Size5Stack* create() {
  Size5Stack* stack = malloc(sizeof(Size5Stack));
  if (stack == NULL) {
    return NULL;
  } else {
    stack->end = -1;
    return stack;
  }
}
```

```
typedef struct {
  int end;
  int data[5];
} Size5Stack;
```

```
int push(Size5Stack* stack, int item) {
  if (stack->end == 4) {//stack is full
    return -1; //-1 means fail
  } else {
    stack->data[++stack->end] = item;
    return 0; //0 means success
  }
}
```

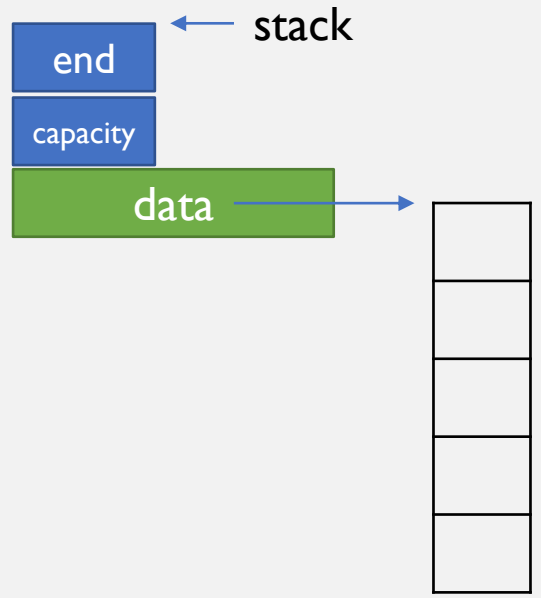
Implementing A Resizable Stack (1)

- Main idea: use a pointer to a dynamic array (not the same as the "Dynamic Array" ADT) to store items, begin with a certain size and keep track of it as capacity

```
typedef struct {
  int end; //index of the latest item
  int capacity; //num of max. items
  int* data; //point to array
} Stack_t;
```

```
Stack_t* create() {
  Stack_t* stack = malloc(sizeof(Stack_t));
  if (stack == NULL) {
    return NULL;
  } else {
    stack->end = -1;
    stack->capacity = 5; //give an initial size
    stack->data = malloc(sizeof(int)*stack->capacity);
    return stack;
  }
}
```

A better way to do this is to define a constant at the top



Implementing A Resizable Stack (2)

- Main idea: use a pointer to a dynamic array (not the same as the "Dynamic Array" ADT) to store the items, when capacity is reached, grow the dynamic array
  - The function realloc will first expand the memory space to store for the extra size, if not it'll find another place and copy the content over (refer to https://www.cplusplus.com/reference/cstdlib/realloc/ for details)

```
int push(Stack_t* stack, int item) {
  if (stack->end == stack->capacity-1) {//stack is full
    stack->capacity += 1;
    stack->data = realloc(stack->data,
                          sizeof(int)*stack->capacity);
  }
  stack->data[++stack->end] = item;
  return 0; //0 means success
}
```

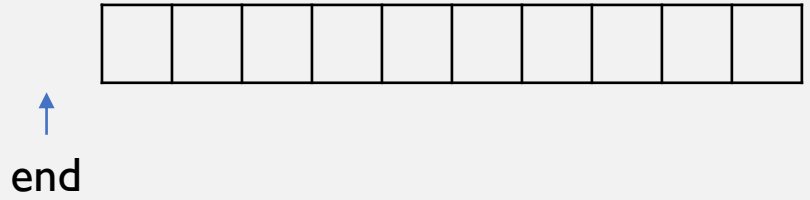
```
int pop(Stack_t* stack) {
  if (isEmpty(stack)) {//nothing to pop
    return -99999; //-99999 means error
  } else {
    return stack->data[stack->end--];
  }
}
```

Implementing A Resizable Stack (3)

- There are different strategies to resize the dynamic array during push & pop, including:
  - during push, if capacity is exceeded increase the array size by 1
  - during push, if capacity is exceeded double the array size
  - during pop, reduce the array size by 1
  - during pop, reduce the array size by half only when half of the array is used
  - during pop, keep the array size
- Different strategies have different strengths & weaknesses

Implementing A Queue (Version 1) from Last Lecture

- Supports create, enqueue, dequeue, isEmpty
- Removal of item follows the First-In-First-Out (FIFO) order



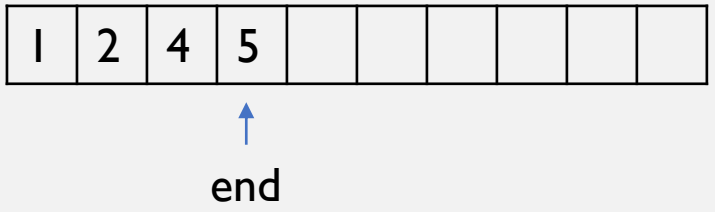

```
typedef struct {
  int end; //index of the latest item
  int data[10];
} Size10Queue;
```

```
bool isEmpty(Size10Queue* queue) {
  return queue->end == -1;
}
```

```
Size10Queue* create() {
  Size10Queue* queue = malloc(sizeof(Size10Queue));
  if (queue == NULL) {
    return NULL;
  } else {
    queue->end = -1;
    return queue;
  }
}
```

Implementing A Queue (Version 1 cont'd) from Last Lecture

- Enqueue: add the item to the end (first slot available)
- Dequeue: remove (and return) the item with index 0 (first item)
  - Very inefficient because need to shift everything up by 1

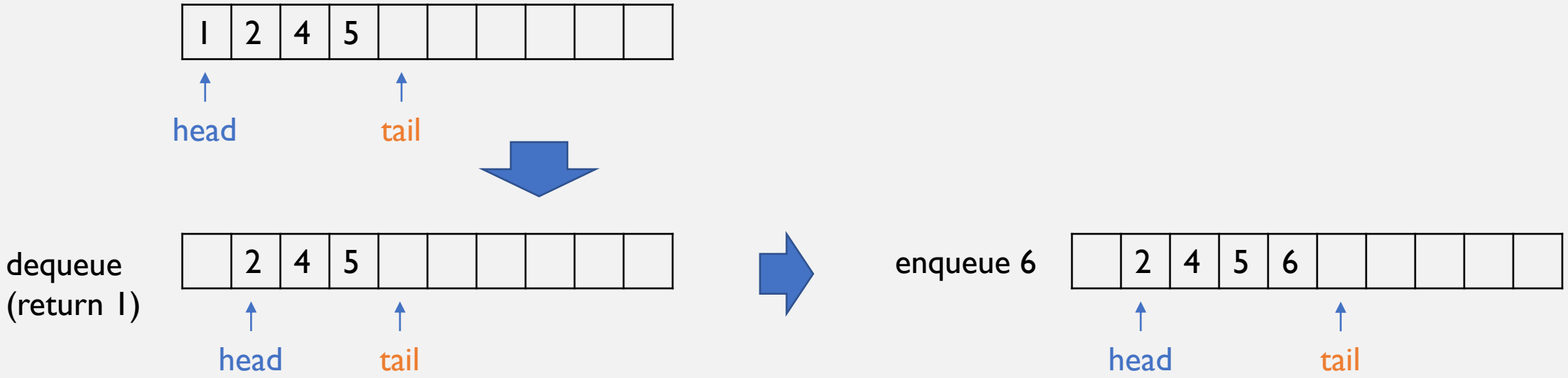



```
int enqueue(Size10Queue* queue, int item) {
  if (queue->end == 9) {//queue is full
    return -1; //-1 means fail
  } else {
    queue->data[++queue->end] = item;
    return 0; //0 means success
  }
}
```

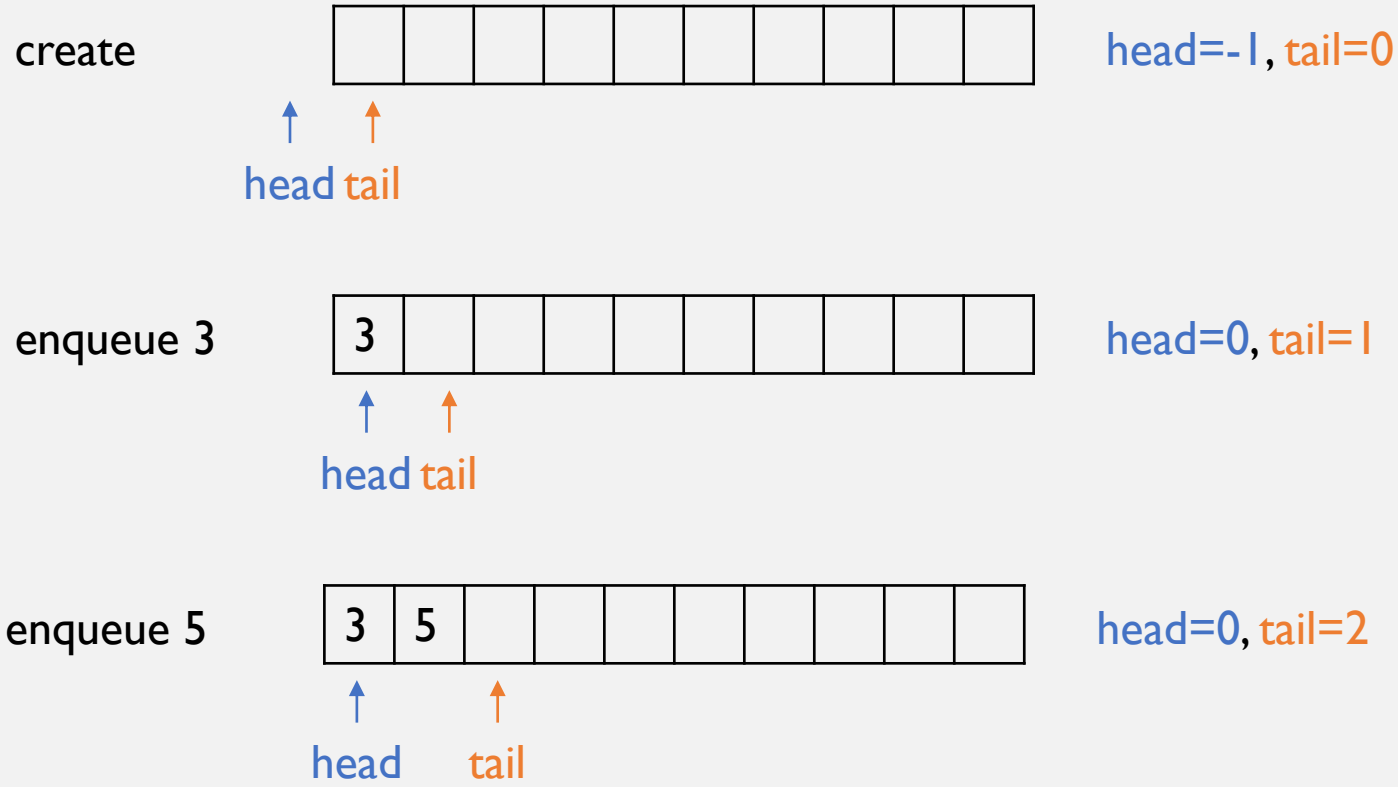
```
int dequeue(Size10Queue* queue) {
  if (isEmpty(queue)) {//nothing to dequeue
    return -99999; //-99999 means error
  } else {
    int toReturn = queue->data[0];
    for (int i=1; i<end; i++) {
      queue->data[i-1] = queue->data[i];
    }
    queue->end--;
    return toReturn;
  }
}
```

Implementing A Better Queue (1)

- A better way to implement a queue is to use 2 indexes (head & tail) to remember where the items are in the array
  - head remembers where the first item (oldest) is, tail remembers where the next available spot is

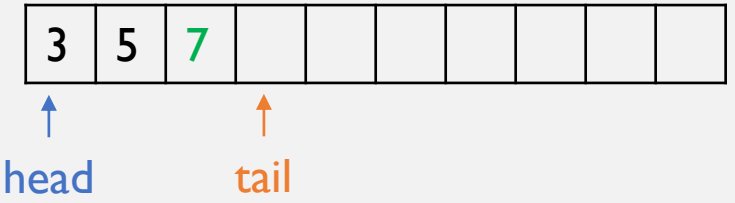


Implementing A Better Queue (2)



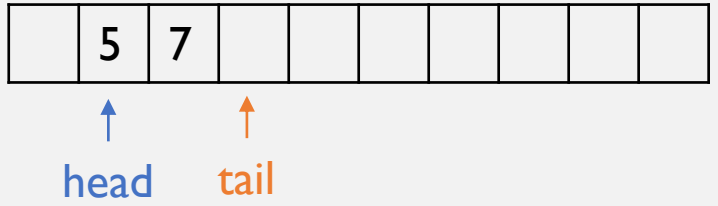
Implementing A Better Queue (3)

enqueue 7



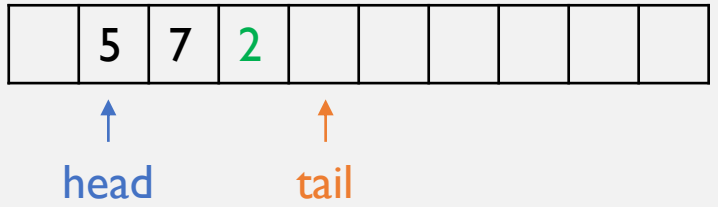
head=0, tail=3

dequeue
(return 3)



head=1, tail=3

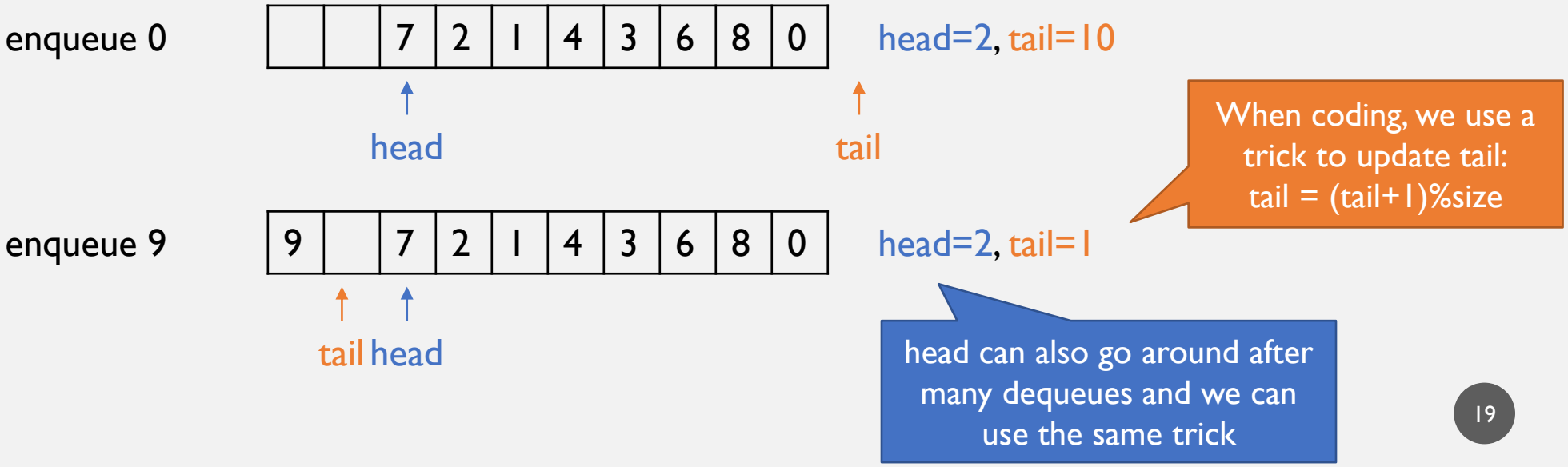
enqueue 2



head=1, tail=4

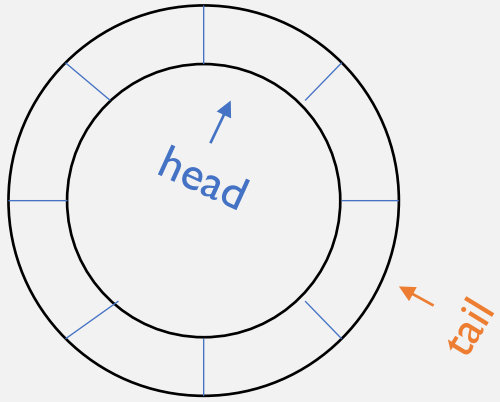
Implementing A Better Queue (4)

- All the enqueue & operations only require a constant amount of time → very efficient
- Question: what if we want to enqueue another item and tail is already at the end of the array (tail=10)?
  - If there is space in the front of the array (before head), use it to store the new item



Queue Implemented by A Circular Array

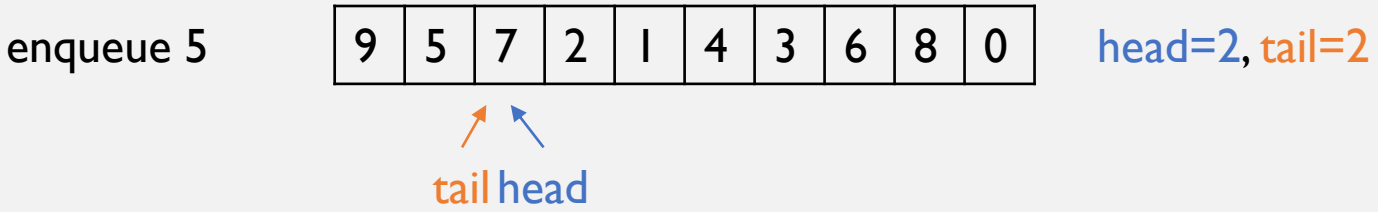
- You can imagine the head & tail indexes go around like a circle, hence this technique is also called the circular array technique



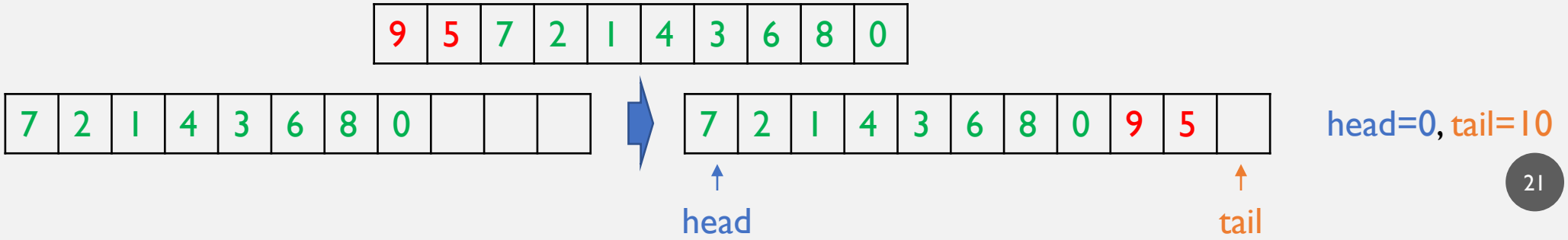
- Think about this:
  - Can you tell if the array is full by just looking at the values of head & tail? If so, how?

Implementing A Better Resizable Queue

- Question: what happens when head == tail?
  - It means the queue is full



- To make the queue resizable, we need to increase the capacity using a similar strategy as in stack
  - Use malloc to request more space, then copy from the head and follow till where the tail is. Finally update head & tail



Can We Do Better?

- In order to make the stack & queue ADTs resizable, we incorporate a dynamically growing array to store the items
  - This process **might** take O(n) of time because they might need to copy the items from the old array to the new one
- Now we introduce a way to store the items to provide flexible sizes and O(1) time complexity for both the adding and removing operations of items in stacks & queues

Linked List

Linked Lists

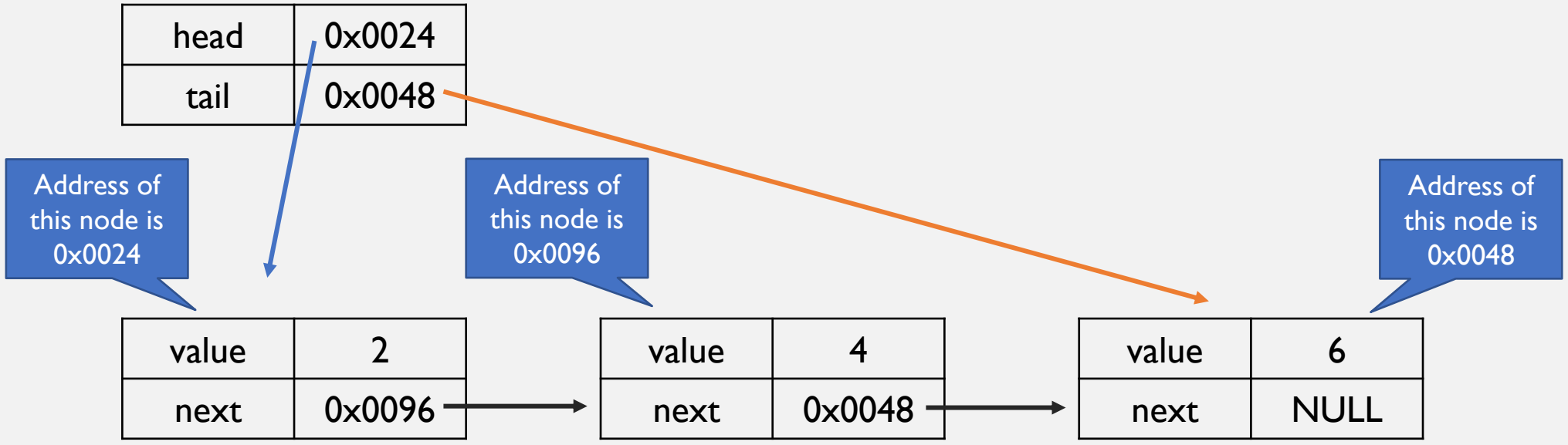
- **Linked lists** are data structures that allows storage of a large number of data of the same type, with fast inserts/deletes, and without size limitations (except system memory limitations)
- They are based on the concept of nodes which store **2 things** in each node:
  - the value of 1 item (can be int, double, boolean, string, a struct, …etc., even pointer)
  - the memory address of the next node

value	2
next	0x0024

- This means a linked list can be "scattered" in the heap memory (places where run-time variables are stored) of the program, as opposed to being continuous like an array

Linked Lists Structure

- 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

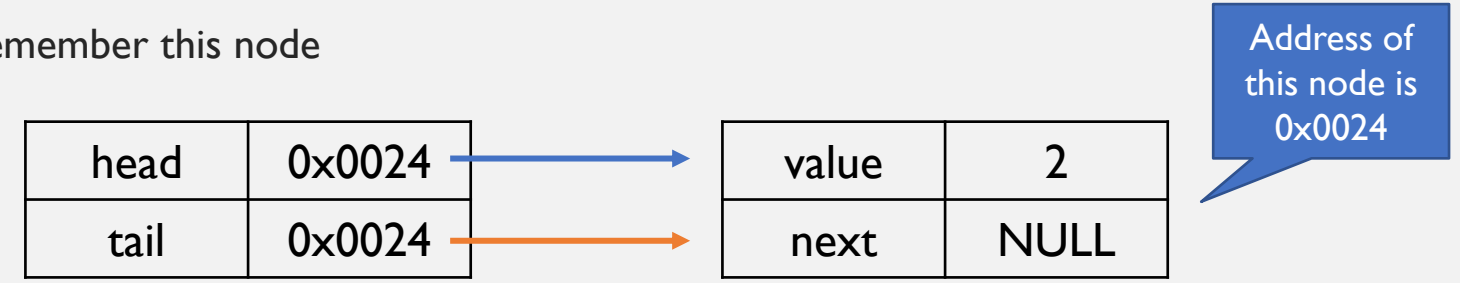


Linked Lists – Basic Operations (1)

- Create: start with an empty list, which has no node
  - When there is no address to store, we use NULL instead

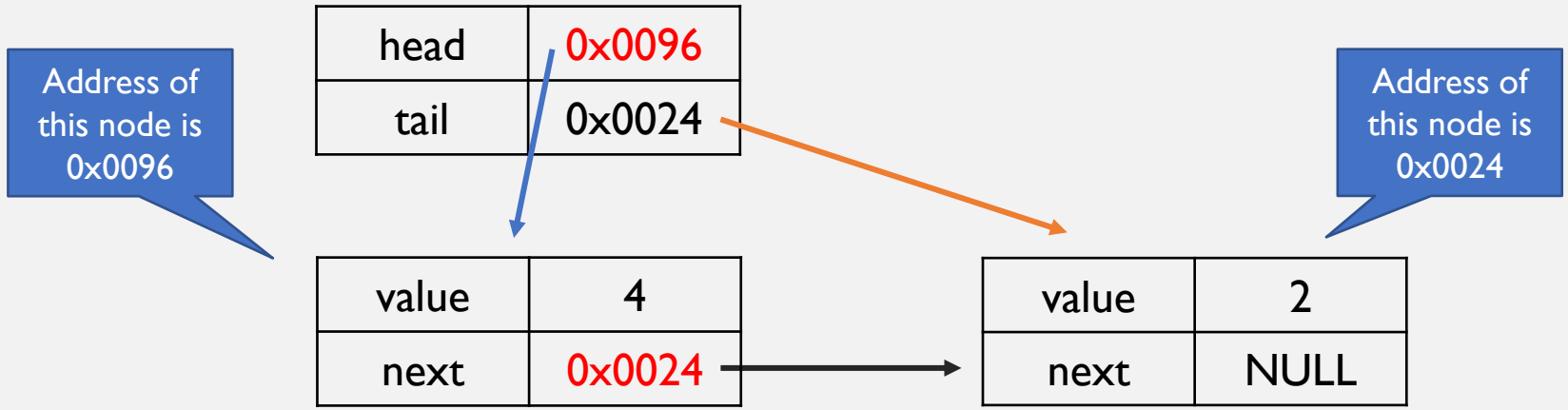
head	NULL
tail	NULL

- Insert first node: create a new node storing the value of the inserted item, next pointer set to NULL
  - Since there is no next node yet, the next pointer remains NULL
  - Update head & tail to remember this node



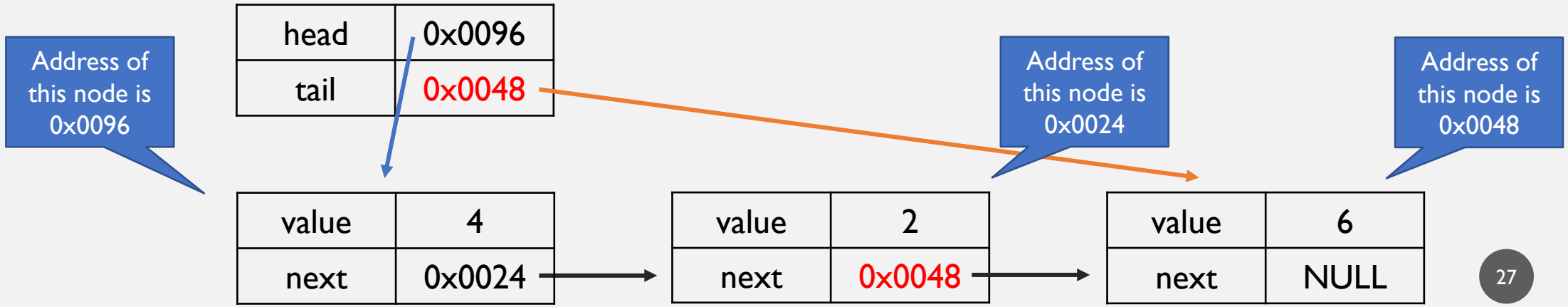
Linked Lists – Basic Operations (2)

- Insert another node (**version 1** – insert to the front):
  - Create a new node storing the value of the inserted item, next pointer set to NULL
  - Since the new node is now the new front, its next node will be the original front, so update its next pointer accordingly
  - Update head to remember this node (tail remains the same)



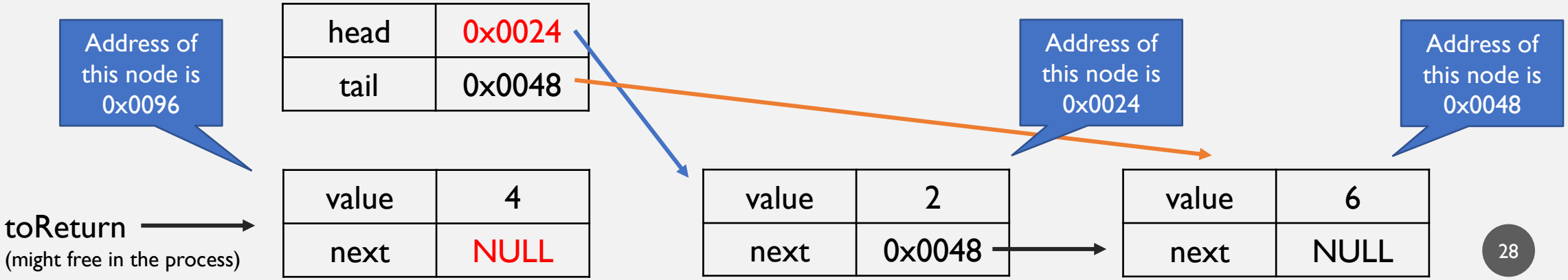
Linked Lists – Basic Operations (3)

- Insert another node (**version 2** – insert to the end):
  - Create a new node storing the value of the inserted item, next pointer set to NULL
  - Since the new node is now the new tail, its previous node will be the original tail, so update the tail's next pointer accordingly
  - Update tail to remember this node (head remains the same)



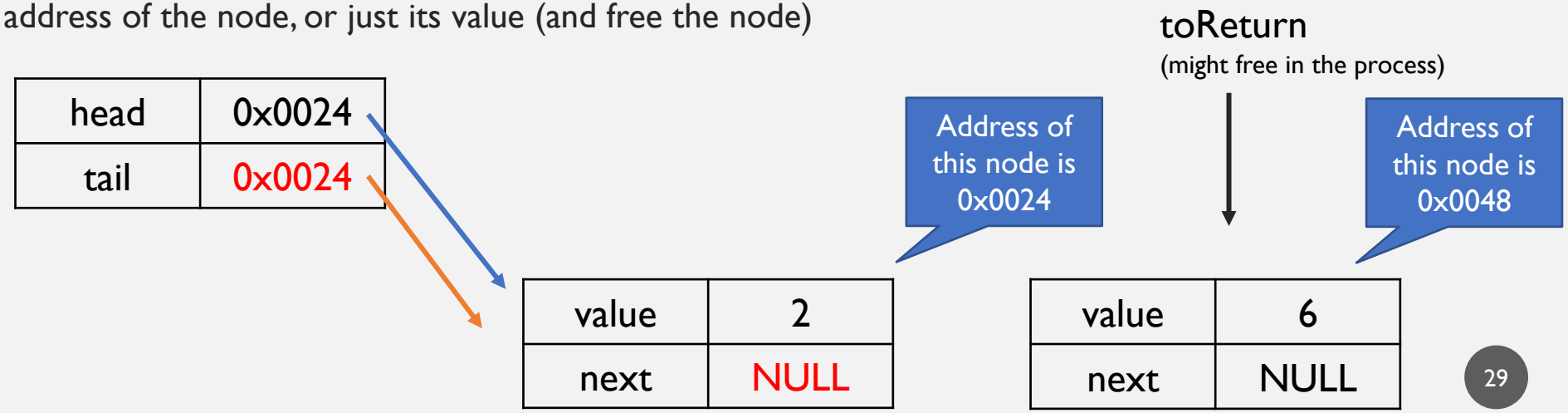
Linked Lists – Basic Operations (4)

- Remove a node (**version 1** – remove from the front):
  - Use an extra node pointer to remember where the node is (*once we update head no one else will know where it is!*)
  - Update the next pointer of this node to NULL since it is not in the list anymore
  - Update head to remember the new front (tail remains the same, unless the list becomes empty, then update both to NULL)
    - return the address of the node, or just its value (and free the node)



Linked Lists – Basic Operations (5)

- Remove a node (**version 2** – remove from the end):
  - Use an extra node pointer to remember where the node is (*once we update tail no one will know where it is!*)
  - Update the next pointer of the previous node of this node to NULL since it is not in the list anymore
  - Update tail to remember the new end (head remains the same, unless the list becomes empty, then update both to NULL)
    - return the address of the node, or just its value (and free the node)



Today's Review

- Implementation details
  - **Stacks** – making them resizable using realloc
  - **Queues** – using a circular approach to use all the space, 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

Homework!

- Implement the queue ADT in C using an int array of size 10 (Size10Queue)
  - Use the better way (circular array method with head & tail)
  - To challenge yourself, make it resizable with malloc