Queue Lab

Shinwoo Kim
Teaching Assistant
shinwookim@pitt.edu
https://sites.pitt.edu/~shk148/CS0449-2234/

Spring 2023, Term 2234
Friday 12 PM Recitation
5502 Sennott Square
Feb 17th, 2023

- ➢ *queue lab intro*
- ➢ *the queue data structure*
- ➢ *lab logistics*
- ➢ *queue lab walkthrough*

Course News!

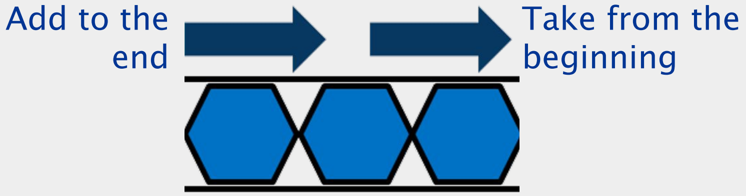
- Queue Lab
  - See Handout on Course Web
  - Today’s recitation topic!
  - Due: 11:59 PM Thursday, February 23rd
- Project 2: Malloc Released!
  - Due: 11:59 PM Friday, March 3rd, 2023.
  - See Handout on Course Web
  - ***START EARLY!!!***
  - Help Session: Next Wednesday 5:45 PM (5502 SENSQ)

Queue

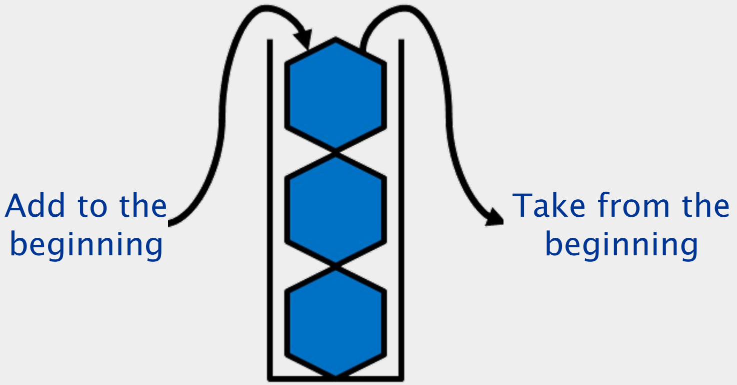
A linear collection of objects that are inserted and removed according to the FIFO principle.

Basic Principles

- First In First Out (FIFO) ← Queue



- Last In First Out (LIFO) ← Stack



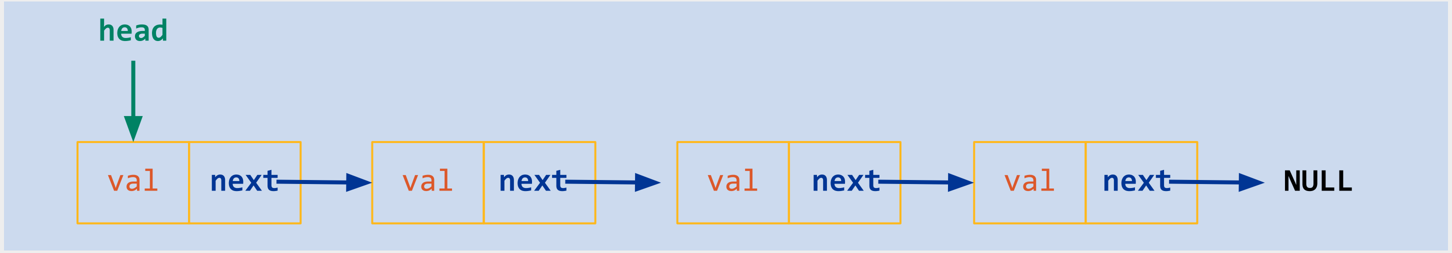
- First-come-first-served resource allocation.
- Dispensing requests on a shared resource.
- Simulations of the real world.

Stacks and queue both arise naturally in countless applications.

- Last-come-first-served resource allocation.
- Function calls in programming languages.
- Fundamental abstractions in computing

Implementation

- How do we connect the elements?
    - Array based data structure
        - But we have to shift every time we 'pop' from the front
        - Also size is finite (we don't know how much space we will need when we *initialize the array)*
    - Linked list data structure
        - No need for shifting ⇒ *simply move the 'head' to the next node*
        - No need to no size at initialization ⇒ If we need more space, simply *allocate* and *link*



Queue Lab

Manipulating linked lists in C

About the Queue Lab

Purpose

To provide practice in the style of C programming which will be required for the next project: *Malloc Project*.

Required Skills

- ➢ Explicit memory management, as required in C.
- ➢ Creating and manipulating pointer-based data structures.
- ➢ Working with strings
- ➢ Enhancing the execution performance of key operations by implementing data structures that trade-off storage for execution speed.
- ➢ Implementing robust code that handles error conditions and operates correctly with invalid arguments, including NULL pointers.

Todo

Implement a *double-ended queue* (*deque*) which supports both *LIFO* and *FIFO* principles using a *singly-linked list* as the underlying data structure. Make the necessary modifications to *enhance the performance* of the queue operations.

Setting up the environment

Handout: https://cs0449.gitlab.io/sp2023/labs/03/

- ➢ Download the lab: `$wget [LINK] -O queuelab-handout.zip`
  - ○ `[LINK]: https://cs0449.gitlab.io/sp2023/labs/03/queuelab-handout.zip`
- ➢ Extract the files: `$unzip queuelab-handout.zip`
  - ○ **DO NOT UNCOMPRESS THE ZIP ON YOUR LOCAL MACHINE**. Doing so may corrupt the permissions and will not allow you to compile and run your program!
  - ○ Please move your files to your *private directory* on Thoth (see Academic Integrity Policies).
- ➢ Move into the new directory: `$cd queuelab-handout`
  - ○ You will modify `queue.h` and `queue.c`
  - ○ A makefile is provided to aid your testing. Run it via: `$make`

Testing

- Tester Program `./qtest`
  - Interactive prompt for testing your implementation
  - You can use the built-in `help` command to see available commands
  - You can individually manipulate which of your functions you call
    - Use `-f traces/trace-xx.cmd` to run a specific trace
    - ⚠️ Write your code to pass the tests in order
      - The later tests intentionally try to break things and without a solid foundation it won't make sense to tackle them

- Autograder: `driver.py` (Used by GradeScope)
  - Runs `./qtest` with all traces
  - Check your code via: `$ make test`

Queue Lab

- Start early
  - This gives you plenty of time to think, try, and seek help
  - *As you may have noticed,* Thoth gets buggy around deadlines
- Draw diagrams
  - This can help you understand what's going on
- Simplify your code
  - Focus on being succinct and clear
- Comment your code!!
  - This helps you remember what you were doing when you come back later
- Don't copy code or try to look up solution
  - It's really easy to detect if you cheat
  - You need to fully understand implementing queues in C
  - Project 2 builds on top of Queue Lab!
- Understanding `malloc()` & `free()` and how `struct`s work is necessary

Supported Operations

Functions	Description
`q_new()`	Create a new, empty queue
`q_free()`	Free all storage used by a queue
`q_insert_head()`	Attempt to insert a new element at the head of the queue (LIFO)
`q_insert_tail()`	Attempt to insert a new element at the tail of the queue (FIFO)
`q_remove_head()`	Attempt to remove the element at the head of the queue
`q_size()`	Compute the number of elements in the queue
`q_reverse()`	Reorder the list so that the queue elements are reversed in order without using malloc or free

queue.h

- You are provided a header file with two struct definitions to implement a linked list
  - **Header file**: contains C code to be shared between several C source files (e.g., `stdio.h`)

Element (Node)

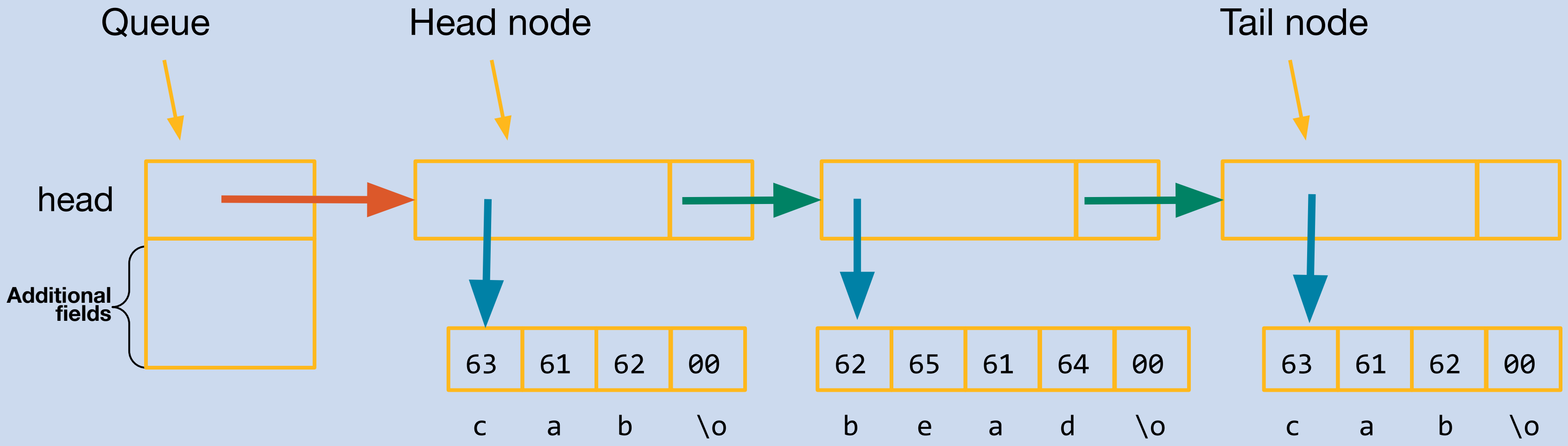
```
/* Linked list element */
typedef struct ELE {
    char *value;
    struct ELE *next;
} list_ele_t;
```

Queue (Linked list)

```
/* Queue structure */
typedef struct {
    list_ele_t *head;
/* Linked list of elements */

} queue_t;
```

queue.c



Points to the head node/element

Points to the next node in the list

Points to the char array value contained in this node

`q_new` creates a new, empty queue

- Allocate memory on the heap
    - `malloc` a `queue_t`
        - What do we do if `malloc()` returns `NULL`?
- Return a pointer to the queue

code snippet from `q_new()`

```
queue_t *q = malloc(sizeof(queue_t));
```

q is a pointer to the beginning of the block in memory where the queue was allocated

Introducing `malloc()`

- Unlike Java, C has no new keyword
- Instead, to denote that a struct, array, or other data is being placed in the heap where it can be accessed via pointers, we request memory using malloc
    - `thing_t thing = malloc(sizeof(thing_t));`
        - Note that we must specify size in bytes
            - `malloc()` returns a block with at least that size
        - Returns a pointer to that memory!

⚠️ Every single object that is malloced must also be **freed** before the program ends or you leak memory

- `free(thing);`

- Project 2 will have you implement your own `malloc()`

The glibc `malloc()` package

```
#include <stdlib.h>
void *malloc(size_t size)
```

- Successful:
  - Returns a pointer to a memory block of at least size bytes
  - aligned to a 16-byte boundary (on x86-64)
  - If **`size == 0`**, returns **`NULL`**
- Unsuccessful: returns `NULL` (0) and sets `errno`

```
void free(void *p)
```

- Returns the block pointed at by `p` to pool of available memory
- `p` must come from a previous call to `malloc()`, `calloc()`, or `realloc()`

Other functions

- `calloc()`: Version of `malloc()` that initializes allocated block to zero
- `realloc()`: Changes the size of a previously allocated block

A `malloc()` example

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

void foo(long n)
{
   long i, *p;

   /* Allocate a block of n longs */
   p = (long *) malloc(n * sizeof(long));
   if (p == NULL) { // always check return value of malloc()
       perror("malloc"); // print error message
       exit(0);
   }

   /* Initialize allocated block */
   for (i=0; i<n; i++)
   p[i] = i;
   /* Do something with p */
   . . .
   /* Return allocated block to the heap */
   free(p); // Always free after malloc()
}
```

Inserting to the queue means more allocations

code snippet from `q_insert_head()`

Declare and allocate space for a new element

```
list_ele_t *newh = malloc(sizeof(list_ele_t));

newh->value = some kind of malloc
strcpy/strdup/etc s into newh->value
```

Allocate space for string in the `value` field of the struct and then copy the given string into it

NOTE:
You can't do `newh->value = s` directly. In C, you have to use string manipulation functions from `string.h`

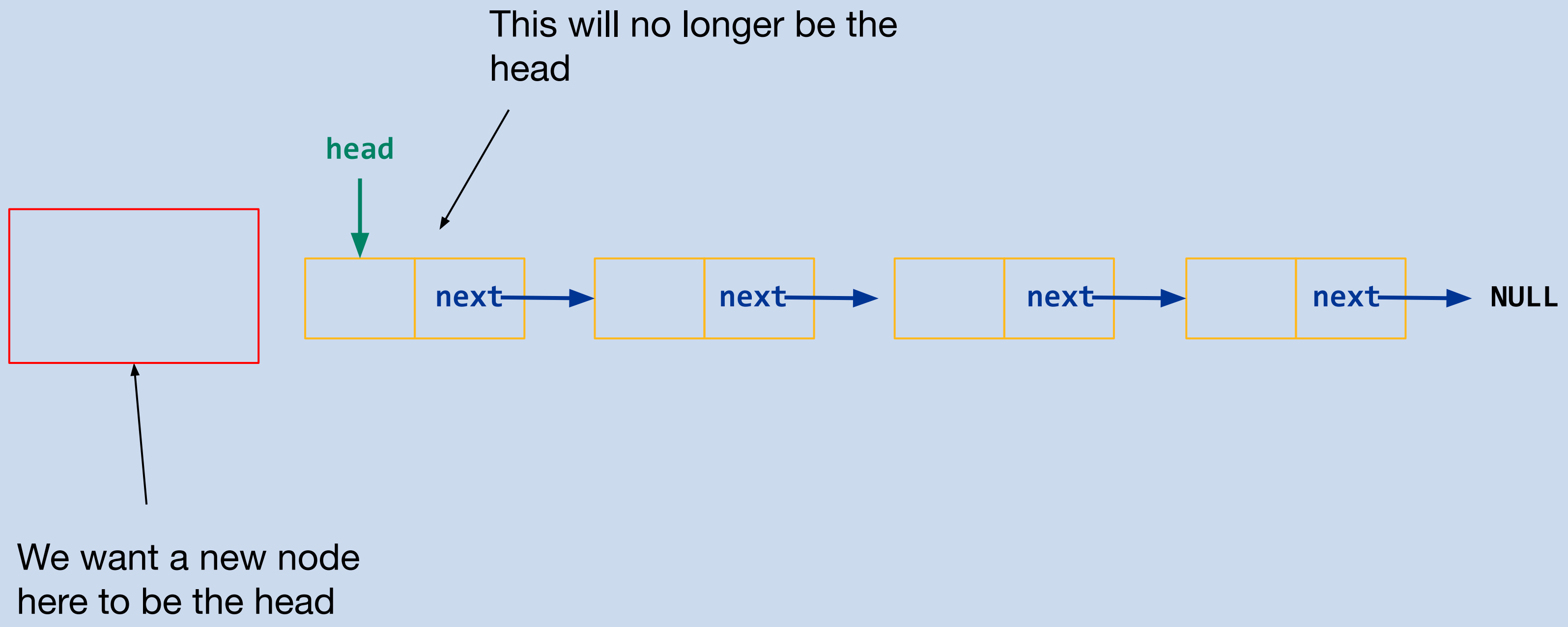
String Operations

function	description
copying strings	
`strcpy()`	copies one string to another
`strncpy()`	copies a certain amount of characters from one string to another
`strdup()`	allocates a copy of a string
`strndup()`	allocates a copy of a string of a specified size
comparing strings	
`strlen()`	returns length of a given string
`strcmp()`	compares two strings
`strncmp()`	compares a certain amount of characters of two strings
"*n*" functions basically just mean "*do this up to at most n*" characters and are often safer	

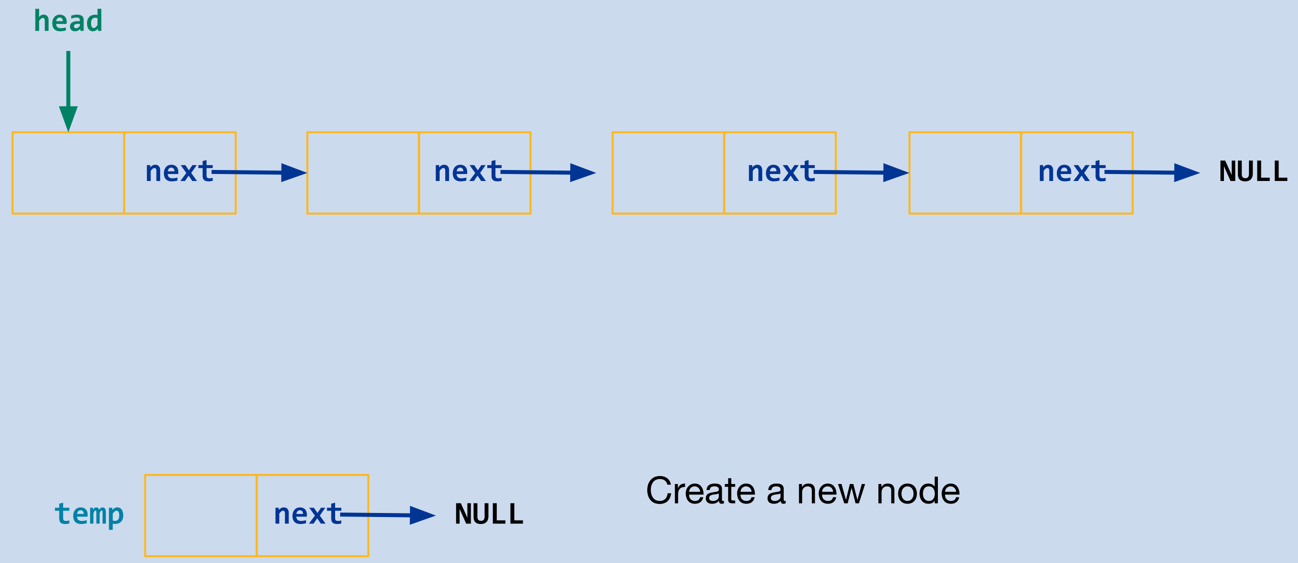
Queue Walkthrough

Insertion using LIFO principles

`q_insert_head()` - insert element at head (LIFO)

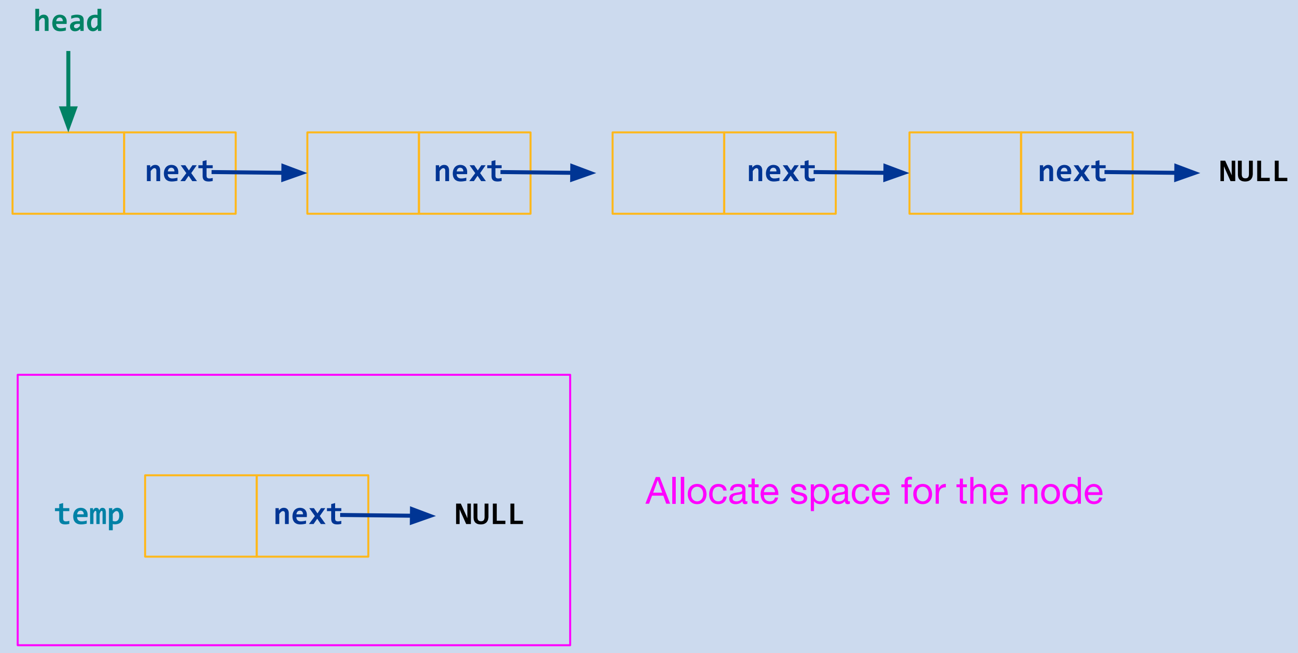

This will no longer be the head
head
next
next
next
next
NULL
We want a new node here to be the head

`q_insert_head()` - insert element at head (LIFO)



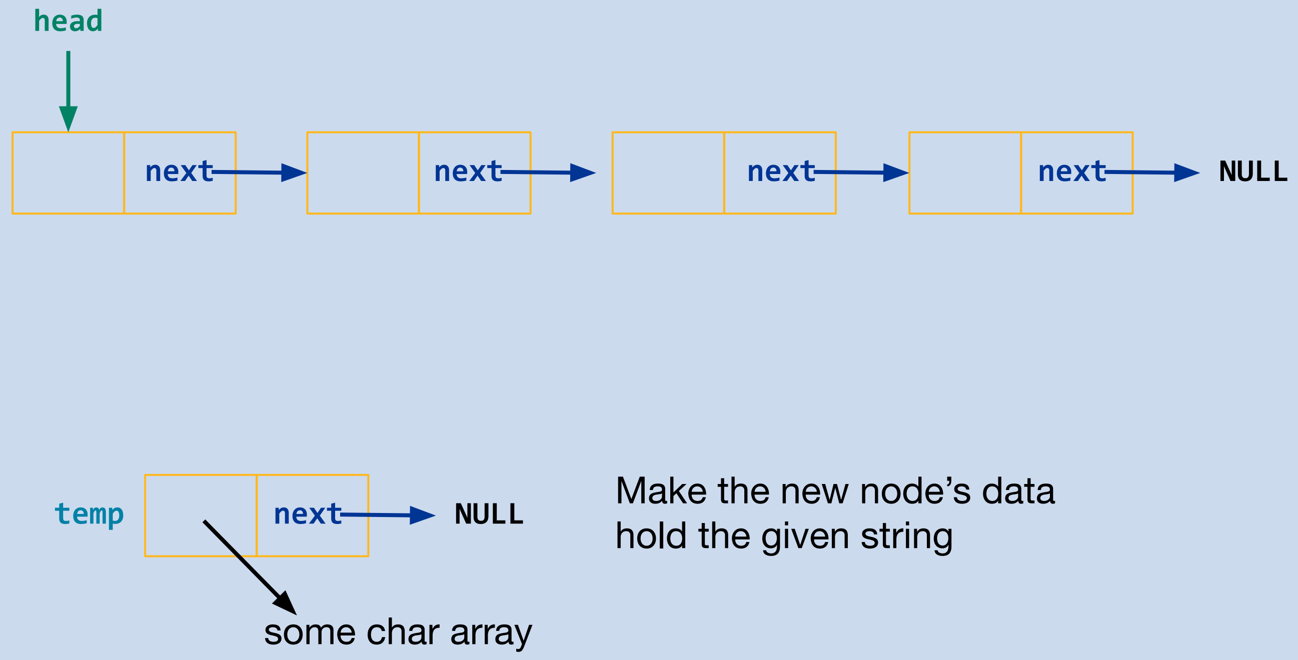
Create a new node

`q_insert_head()` - insert element at head (LIFO)



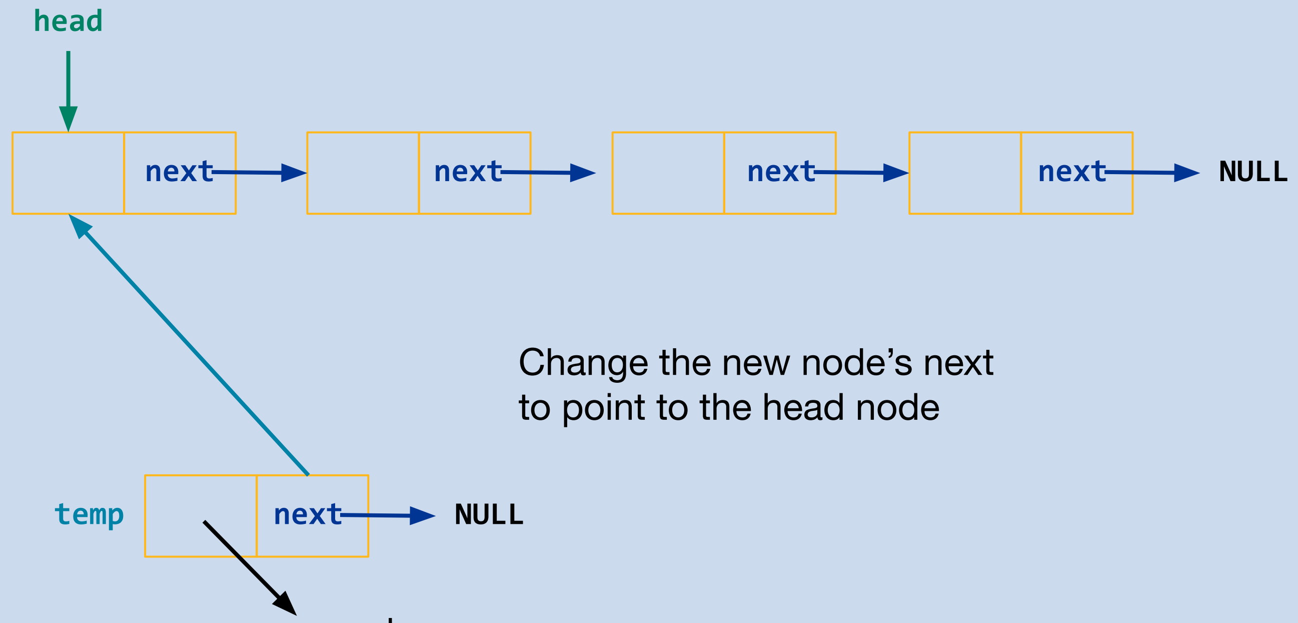
Allocate space for the node

`q_insert_head()` - insert element at head (LIFO)



Make the new node's data hold the given string

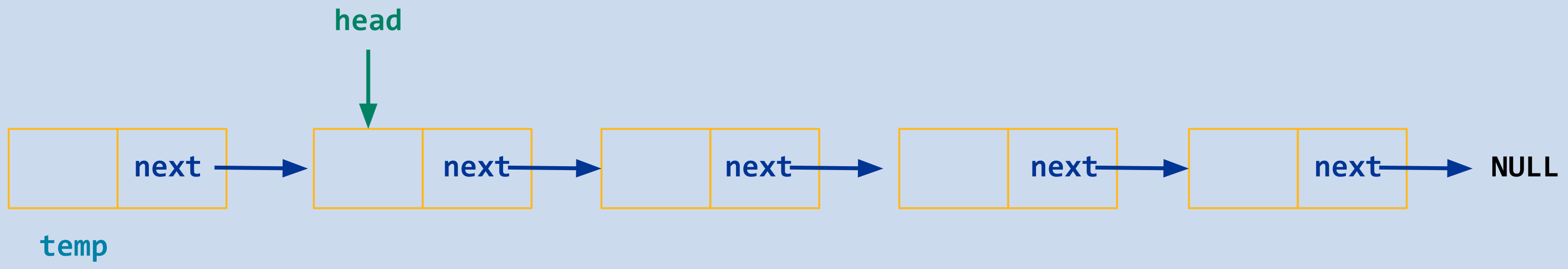
`q_insert_head()` - insert element at head (LIFO)



Change the new node's next
to point to the head node

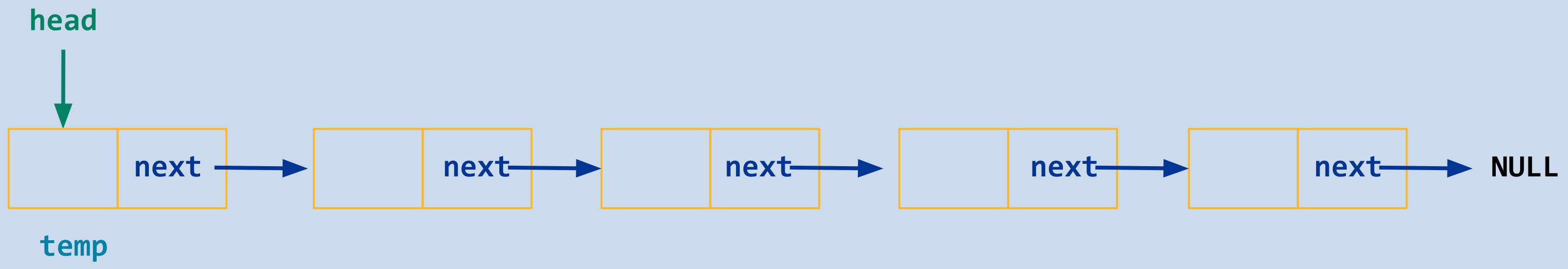
some char array

`q_insert_head()` - insert element at head (LIFO)



`q_insert_head()` - insert element at head (LIFO)

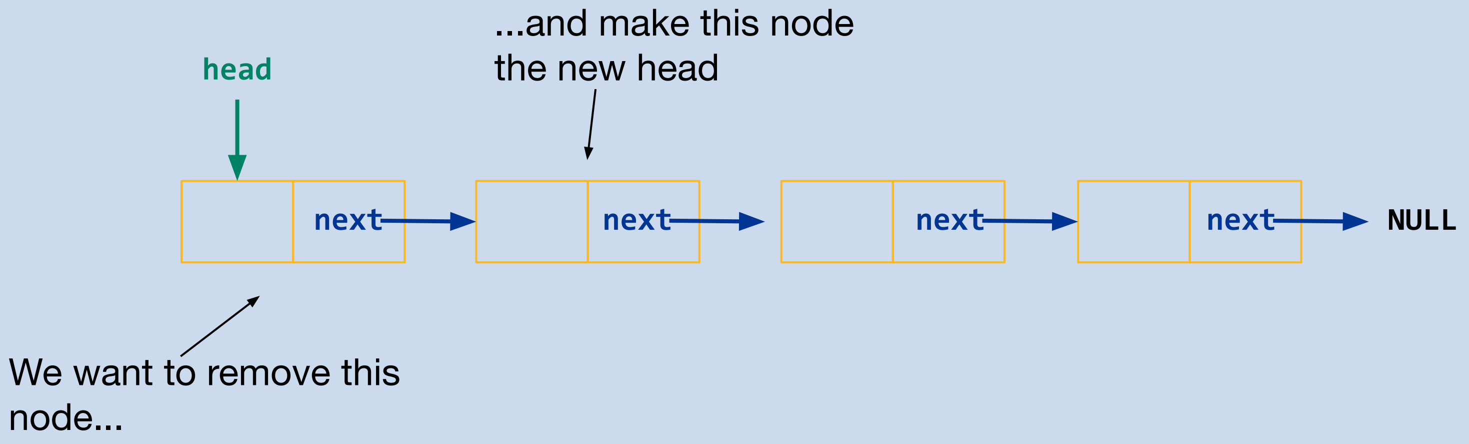
Make the head pointer point to the new node



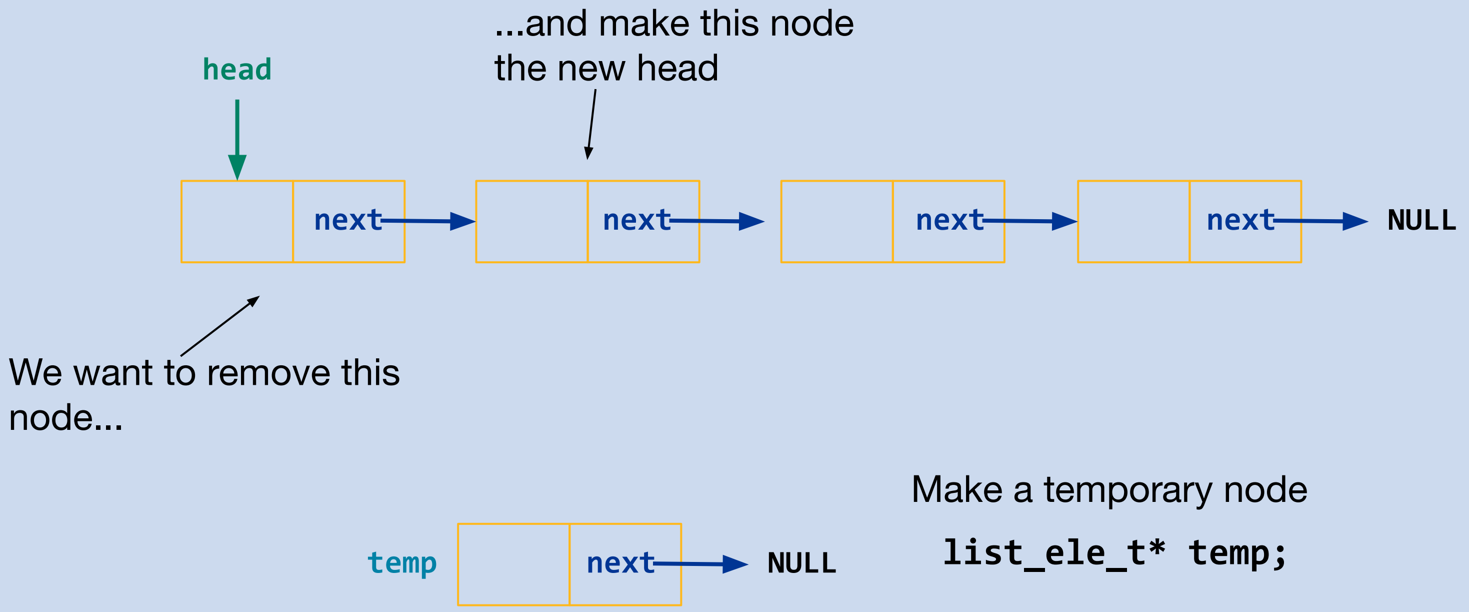
Queue Walkthrough

Removal using FIFO principles

`q_remove_head()` - remove element at head



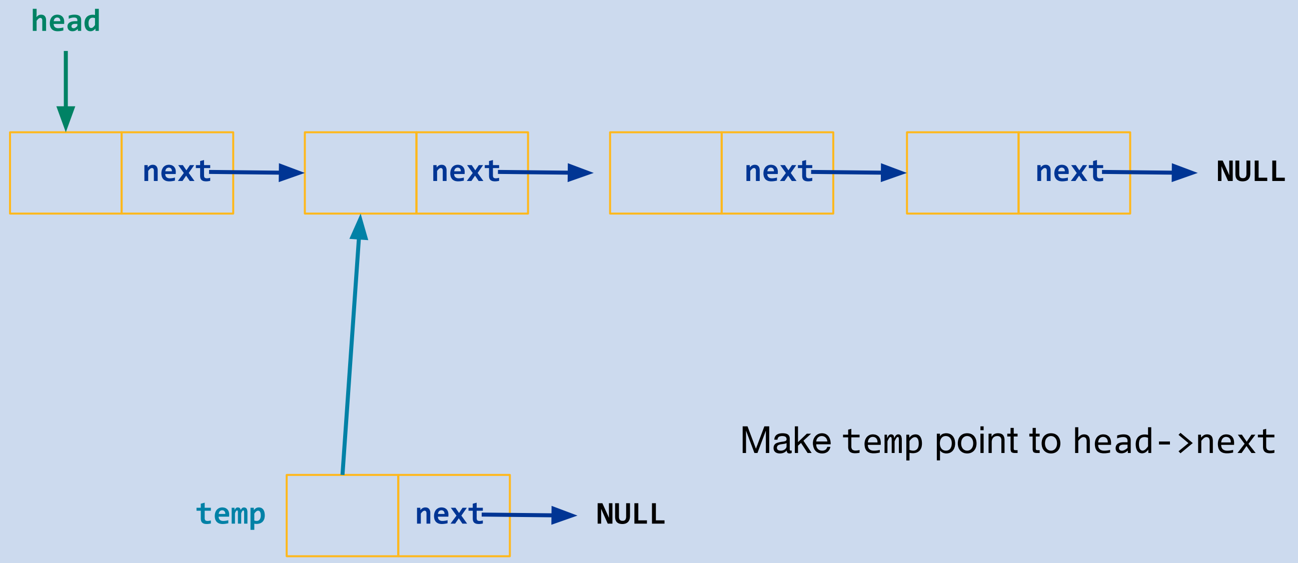
q_remove_head() - remove element at head


...and make this node the new head
head
next
next
next
next
NULL
We want to remove this node...
temp
next
NULL

Make a temporary node

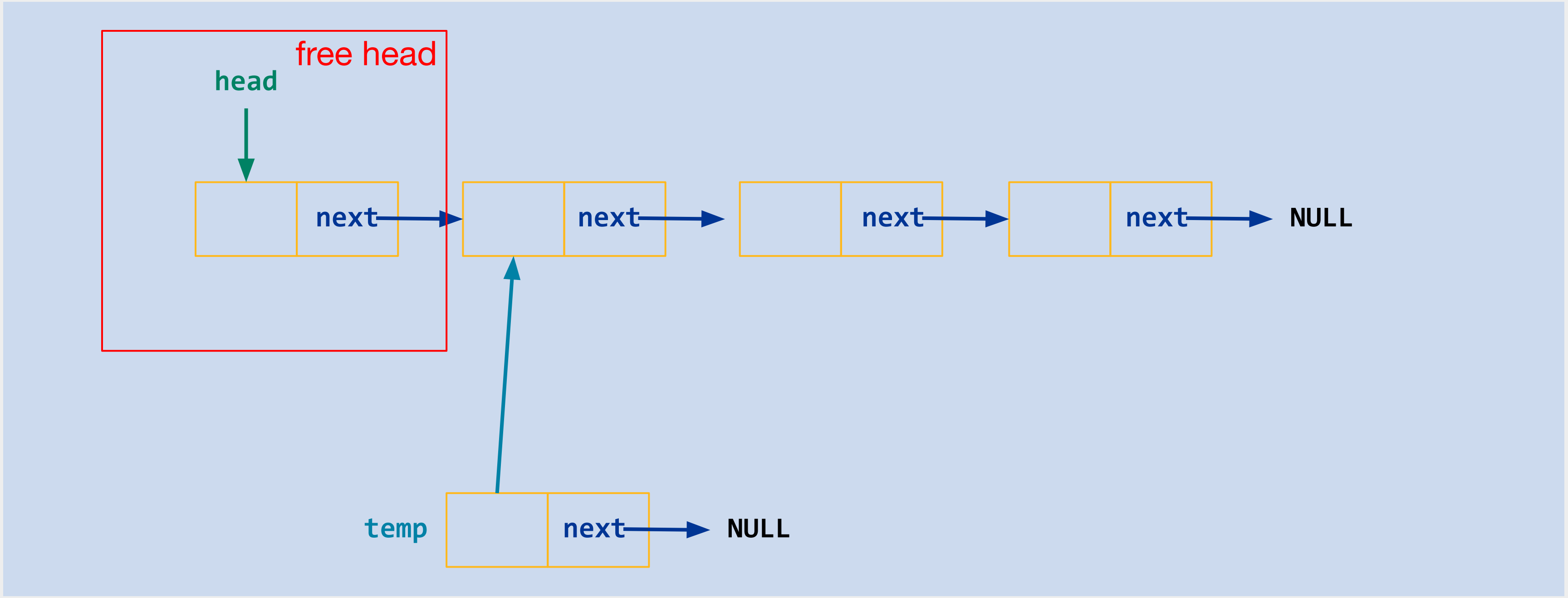
```
list_ele_t* temp;
```

`q_remove_head()` - remove element at head

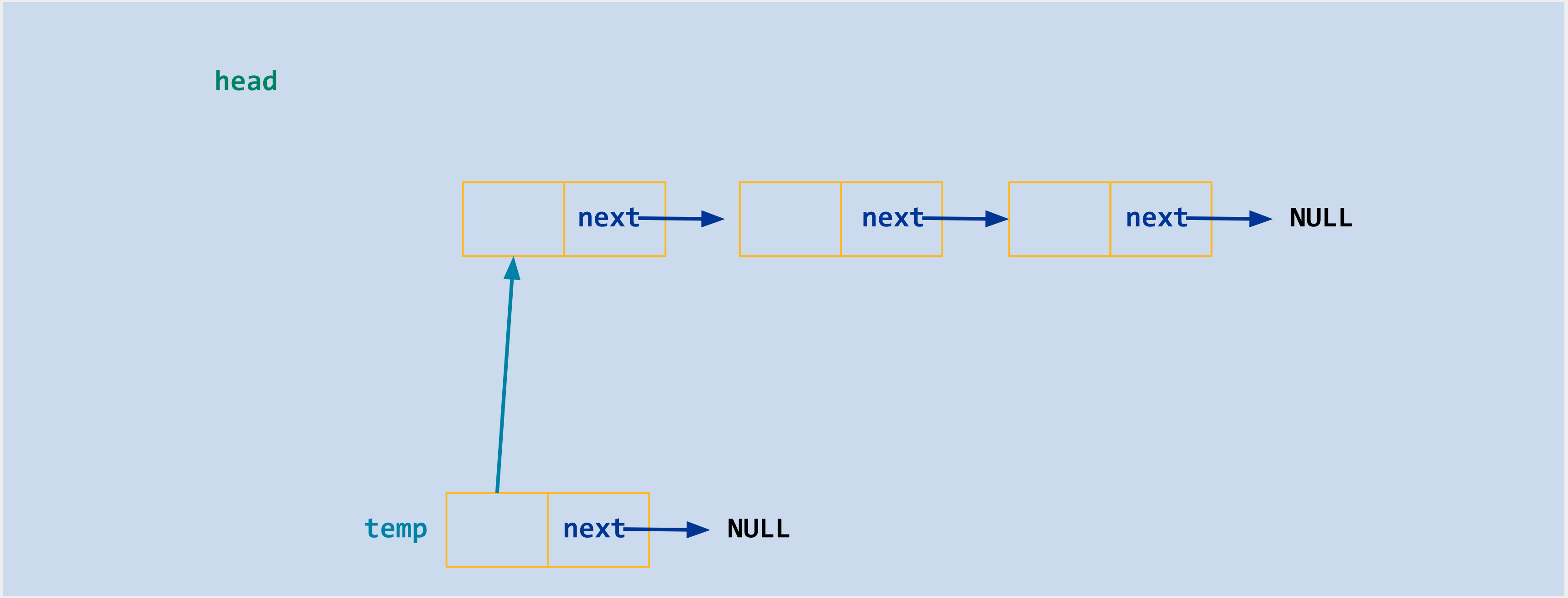


Make `temp` point to `head->next`

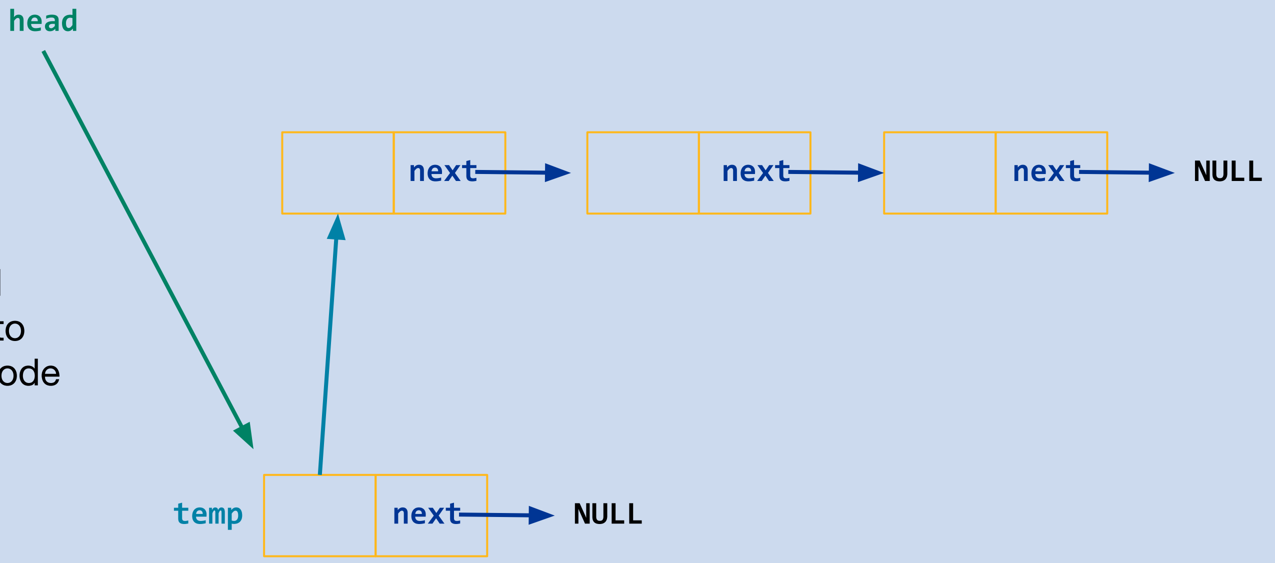
q_remove_head() - remove element at head



q_remove_head() - remove element at head



`q_remove_head()` - remove element at head

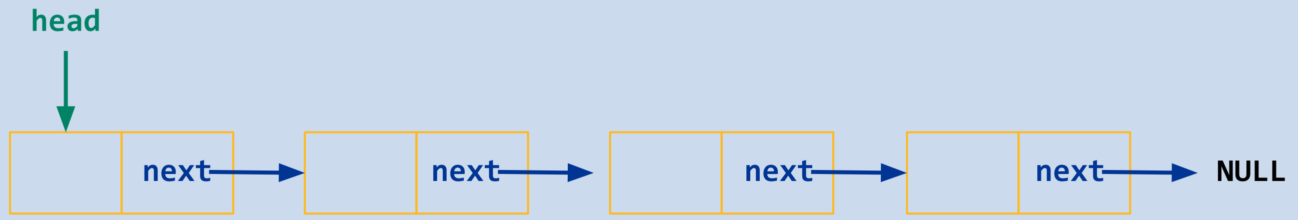


Make head
now point to
the temp node

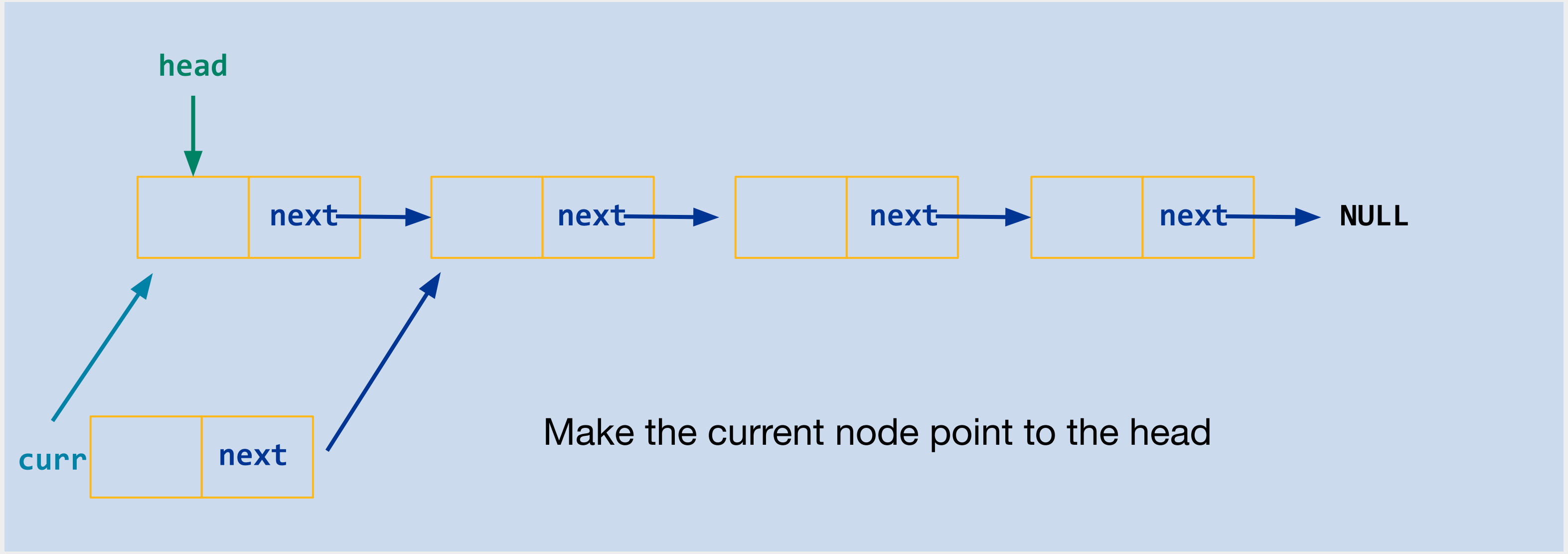
Queue Walkthrough

Iterations and Traversals

Iterating through a linked structure



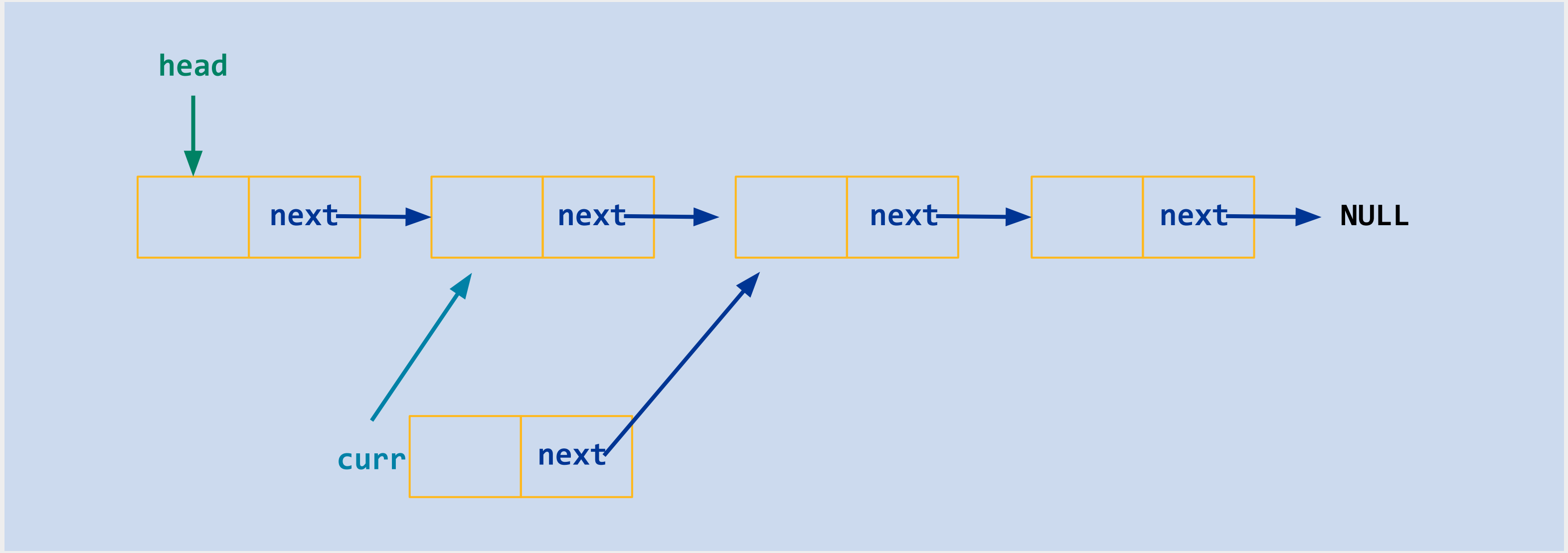
Iterating through a linked structure



Make the current node point to the head

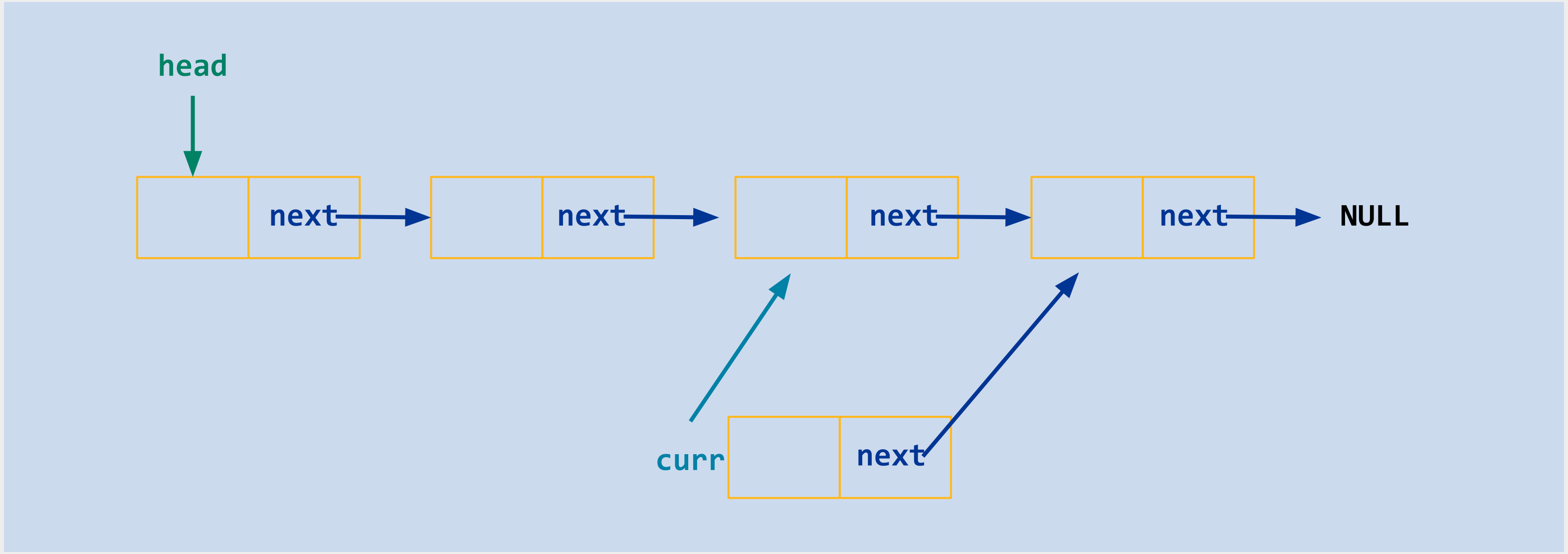
Is `curr->next NULL`?

Iterating through a linked structure



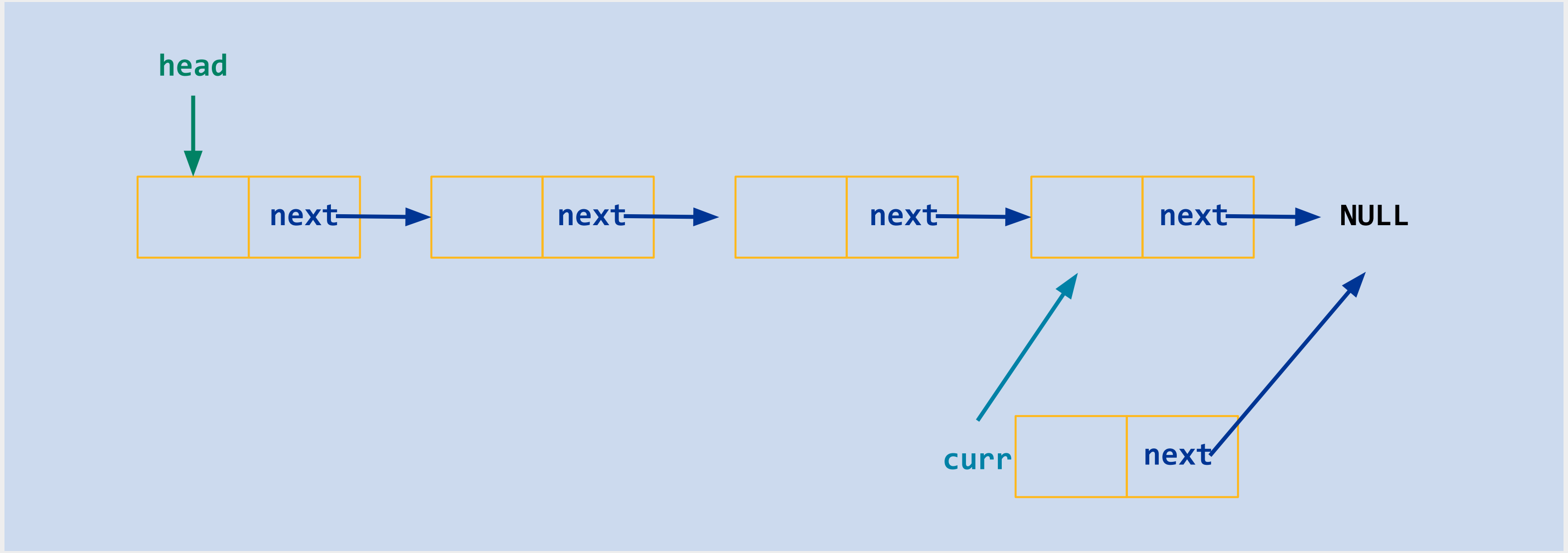
Is `curr->next NULL?`

Iterating through a linked structure



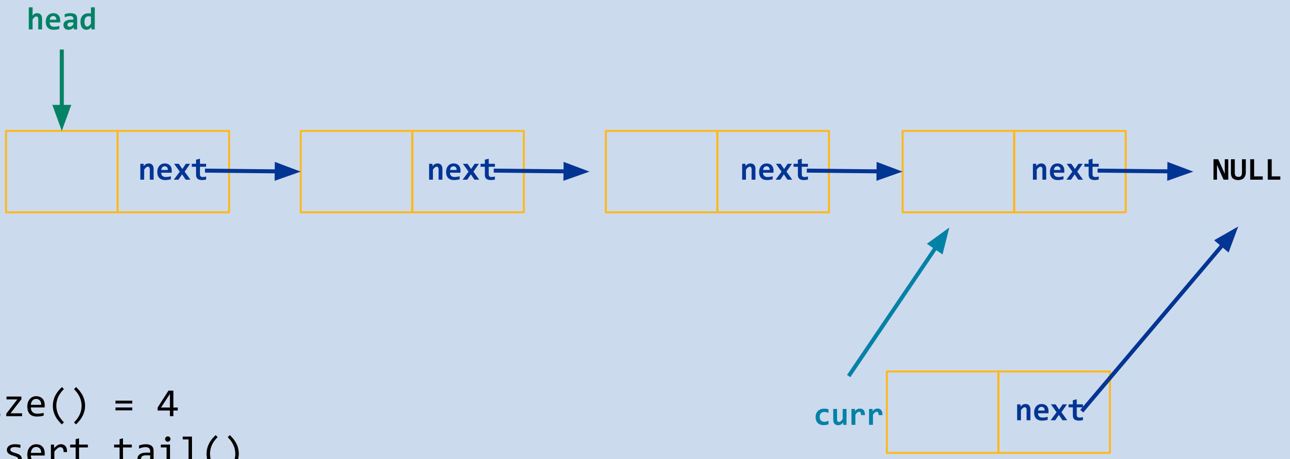
Is `curr->next NULL?`

Iterating through a linked structure



Is `curr->next NULL?`

Iterating through a linked structure



```
q_size() = 4
q_insert_tail()
```

Traversing the queue is slow

- A naïve approach to…
  - `q_insert_tail()` - Iterate through queue and insert a new node at the end
  - `q_size()` - Iterate and count of the number of *links* we traverse
    - Just like `stringLength()` from Pointer Lab
- Run-time?
  - For ‘*n*’ elements, we need to traverse ‘*n*’ elements ⇒ *O(n)*
  - This is fine for a few elements, but what if our queue held 1 billion elements?
    - Do we really want to traverse the whole list, just to ‘add’ 1 element?
    - Do we really want to traverse the entire list every time we want size?
- Can we do better?

Keeping track of important details

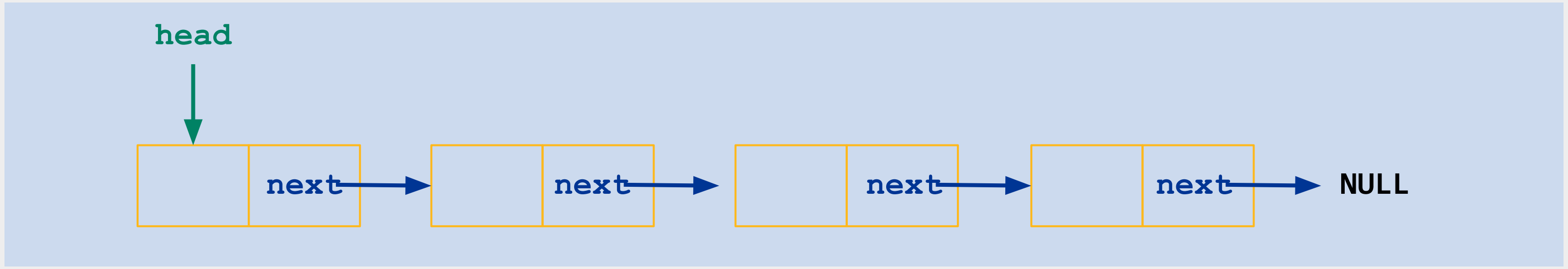
- Queue operations can be implemented in *O(1)* time
  - The runtime should not depend on the number of elements present in the list.
  - You should've seen this in CS445 (in Java).
- Could we store some information about our queue that we can use in our operations?
  - What if we keep a counter of the number of elements?
    - Need to update this counter every time we add/remove.
    - But accessing this value(`int`) can be done in *O(1)* time!
  - What if we keep a reference that allows us to access the last element?
    - We could add a new element to the tail rather easily
    - But we would need to update this reference every time we add/remove to the back
- But where can we store these values?

The space-time tradeoff

- By storing ‘extra’ information about our data structure, we gain a boost in performance…but we need ‘extra’ more memory to store this information
  - This is called a **Space-time tradeoff**
    - or *Time–memory trade-off*
    - or *Algorithmic space-time continuum*

- ***Dynamic Programming*** (*Memoization*) is a algorithmic optimization in which we significantly *reduce time complexity* by *using more memory*.
  - Take CS1501 (*Data Structures and Algorithms II*) to learn more.

`q_free()` - free all storage used by queue



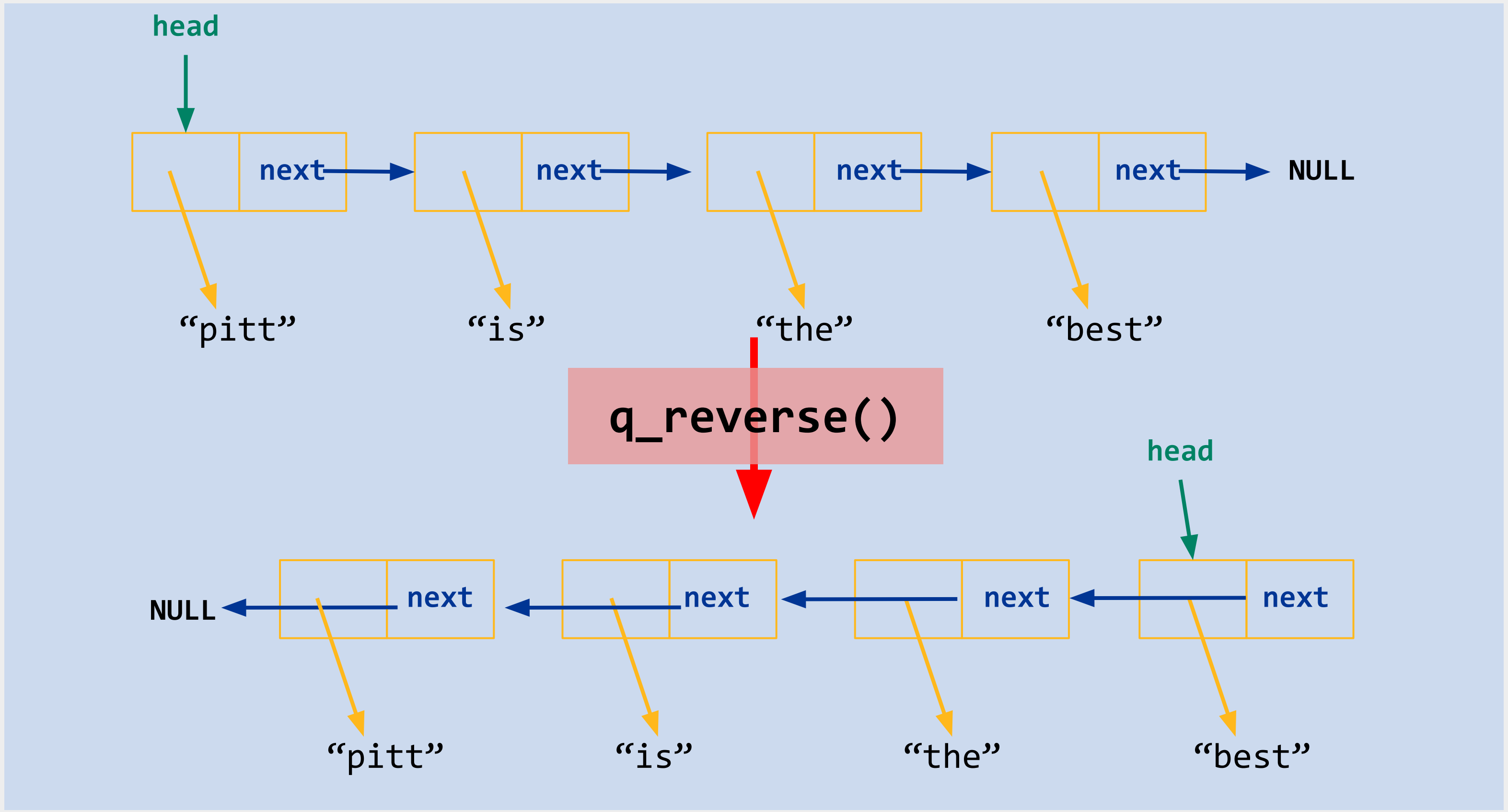
Iterate through the array and free each node

What happens if you *miss* a node?

Queue Walkthrough

Reversing the linked structure

`q_reverse()` - reorder so the queue is reversed



Queue Lab

Resources

Resources

1. C programming
    a. B. W. Kernighan and D. M. Ritchie, “***The C Programming Language, 2nd ed***.” Prentice-Hall, 1988. (Chapter 5 and 6)
    b. Wikibooks “***C Programming***”. Available here: https://en.wikibooks.org/wiki/C_Programming

2. Linked lists
    a. CS 0445 Notes (Data Structures in Java)
        i. Professor Lipschultz (CS 0445): notes on Linked Data Structures
        ii. Dr. Ramirez's CS 0445 Course Website (see lectures section)

3. Asymptotic (big-O) notation.
    a. Review of Asymptotic Complexity (Cornell University)

4. C reference
    a. C Reference @ cppreference.com
        i. Dynamic memory management
        ii. C Strings Library

Per the Academic Integrity Policy, you should not search the web or ask others for solutions or advice on the specifics of the lab.

That means that search queries such as “linked-list implementation in C” are off limits.

Works Referred

Gavin Heinrichs-Majetich's CS 0449 Recitation Slides (Fall 2022)

Martha Dixon's CS 0449 Recitation Slides (Fall 2020)

Robert Sedgewick & Kevin Wayne's *Computer Science: An Interdisciplinary Approach*

Randal Bryant's Computer Systems: A Programmer's Perspective

Carnegie Mellon University's 15-213: *Introduction to Computer Systems*

University of Pittsburgh's CS/COE 445: *Data Structures and Algorithms I*
with materials developed by Michael Lipschultz and Dr. John Ramirez