CMPT 125

Introduction to Computing Science and Programming II

November 1, 2021

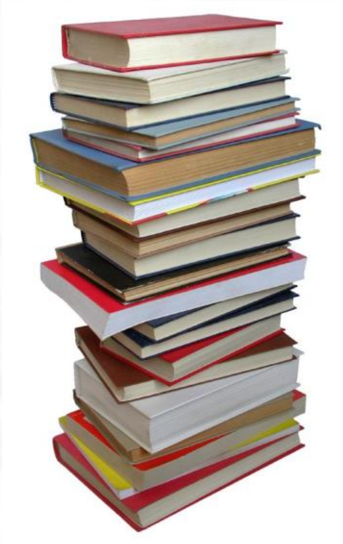
In lab exam: Wednesday, Nov 17

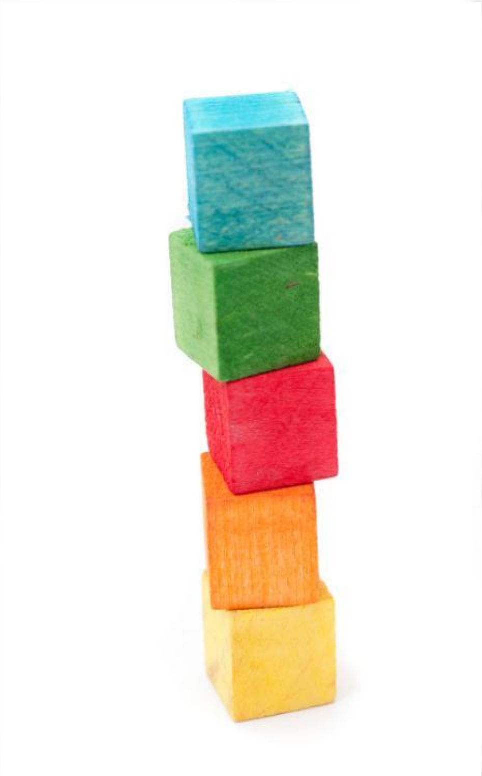
- Wednesday, Nov 17, 2021.
- 50 minutes during your lab section
- Open books, internet (I don’t recommend this, takes too much time)
- No talking to each other.
- You can use your laptop if you want
- The grading will be done on CSIL machine
- You will need to write 3 functions, and submit your code.
- Similar format to homework assignments
- Practice problems on piazza

Stack

Stack

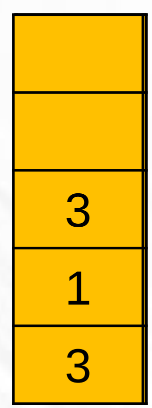
- A stack: an ordered collection of items with the following operations:
  - push(item): add an item to the stack
  - pop(): remove an item from the stack, and return its value
  - isEmpty(): checks if the stack is empty
- Removal follows a last-in-first-out order (LIFO)





Stack

- <u>A stack</u>: an ordered collection of items with the following operations:
  - push(item): add an item to the stack
  - pop(): remove an item from the stack, and return its value
  - isEmpty(): checks if the stack is empty
- Removal follows a last-in-first-out order (LIFO)
  - *init()*
  - *push(3)*
  - *push(5)*
  - *pop() -- returns 5*
  - *push(1)*
  - *push(3)*



Implementing Stack

- Question: How would you implement Stack?

- We saw an implementation using arrays:
  - arr[0] is the bottom
  - int size – points to the top of the stack
  - need to realloc when reaching capacity

Queue

Queue

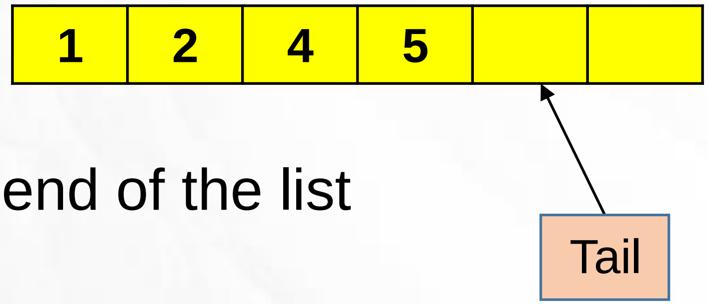
- A queue: an ordered collection of items with the following operations:
  - create(): creates a new queue
  - enqueue(item): add an item to the queue
  - dequeue(): remove an item from the queue
  - isEmpty(): checks if the stack is queue
- Removal follows a first-in-first-out order (FIFO)

- There is no bound on the number of element in the set

- Question: How would you implement Queue?

Queue - implementation

- Use array:
  - Create Queue:
    - Create an array + pointer to end of the list
  - Enqueue
    - Add the element in the end of the list
  - Dequeue
    - Remove the element from the 0 position

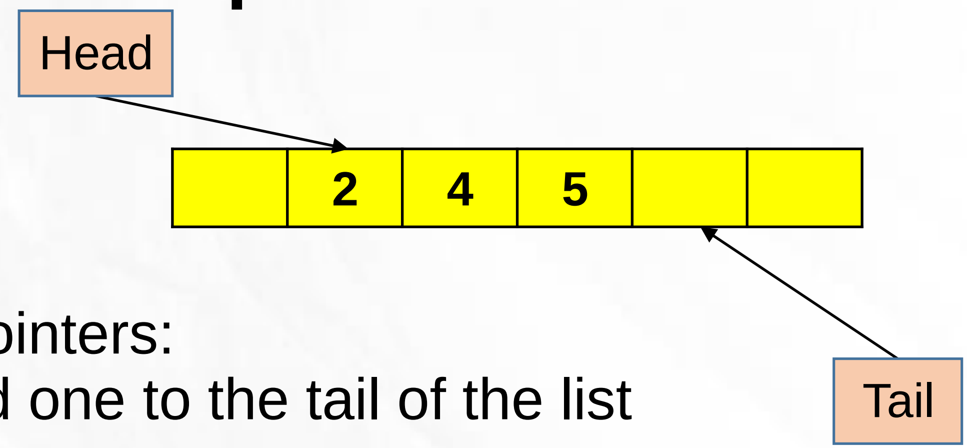


Note that dequeue() is very inefficient!

why?

because need to shift everything to the left

Queue – a better implementation



- Use array:
  - Create Queue:
    - Create an array + 2 pointers:
      one to the head, and one to the tail of the list
  - Enqueue
    - Add the element in the tail, update the tail
  - Dequeue
    - Remove the element from the head, update the head

Queue – a better implementation

Init() [_, _, _, _, _, _, _, _] head=0, tail=0

Enqueue(1) [1, _, _, _, _, _, _, _] head=0, tail=1

Enqueue(2) [1,2, _, _, _, _, _, _] head=0, tail=2

Enqueue(2) [1,2, 3, _, _, _, _, _] head=0, tail=3

Dequeue() [1, 2, 3, _, _, _, _, _] head=1, tail=2

Returns 1

Queue – a better implementation

Q1: What if the tail reaches the end of the array?

Should we increase capacity?

Q2: What if part of the array *is not used* because the head moved too?

A: Move the indices cyclically

Q3: What happens when tail = head-1?

A: We should increase capacity.

Queue – a better implementation

Example: [_, _, _, _, _, 6, 7,_] head=5, tail=7

Enqueue(8) [_, _, _, _, _, 6, 7,8] head=5, tail=0

Enqueue(9) [9, _, _, _, _, 6, 7,8] head=5, tail=1

…

Enqueue(13) [9,10, 11,12, 13, 6, 7,8] head=5, tail=5

Enqueue(14) ---- Increase capacity

Queue

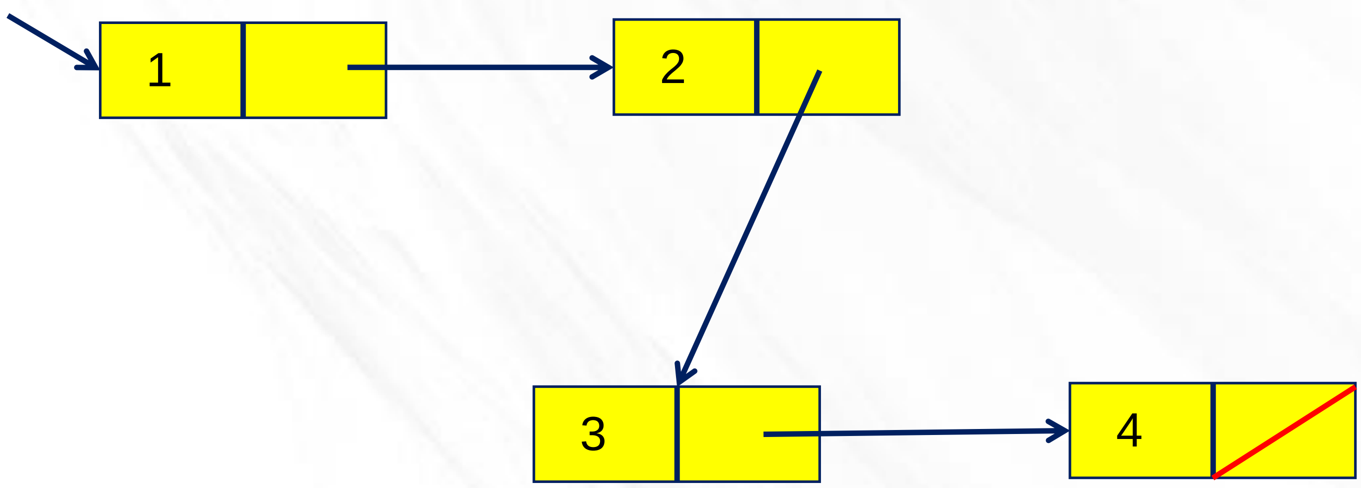
- Homework: Implement Queue using each of the two ideas.

- Idea 1 (returning arr[0]) is not for grading
- Idea with two pointers is in your HW3

Linked List

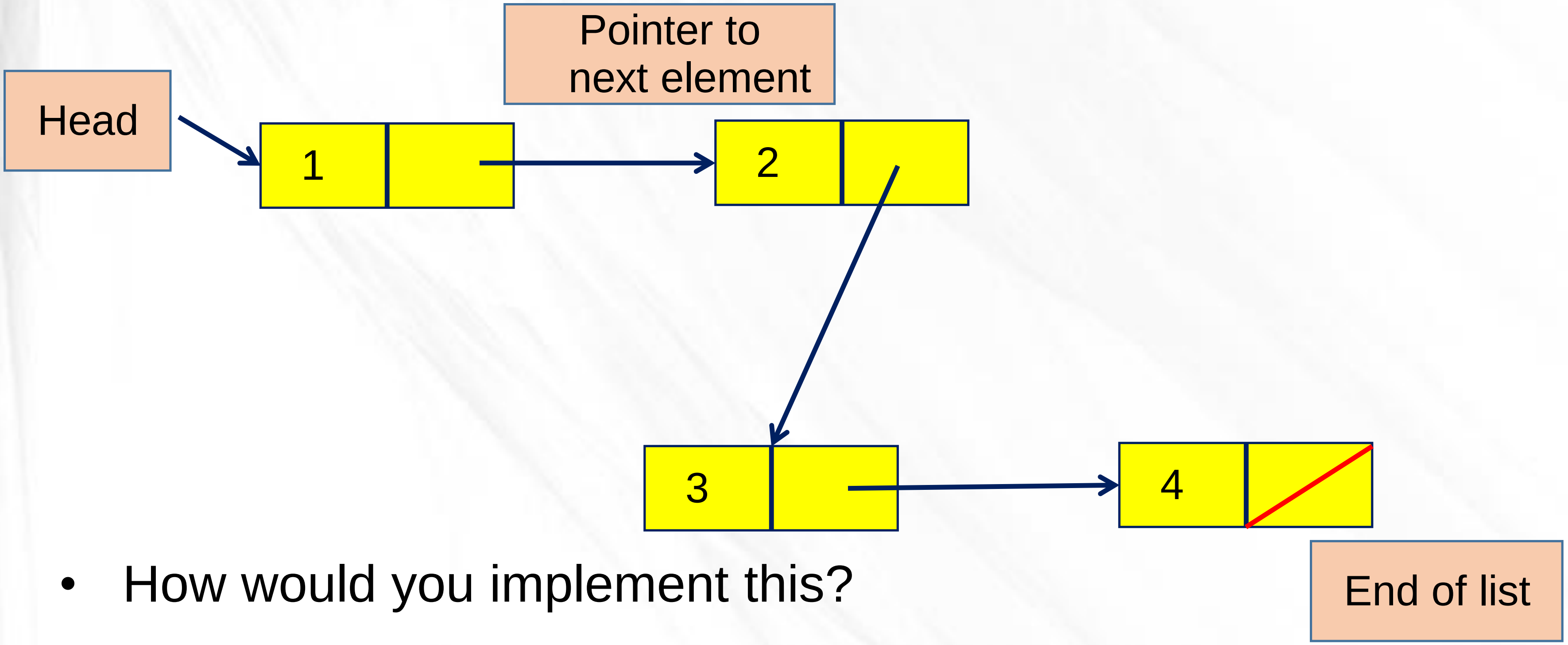
Linked List

- Linked List is a collection of separate elements, where each element is linked to the one following it in the list.
- Think of it as a chain of elements.



Linked List

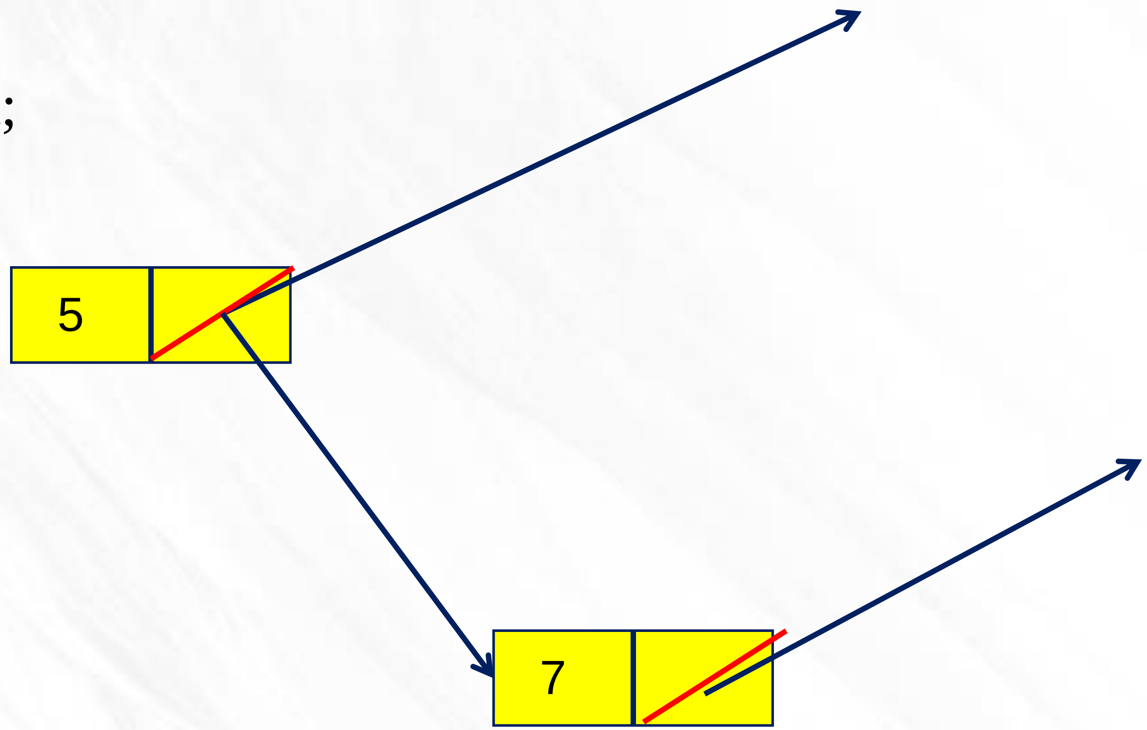
- The list has a head. This is just the pointer to the first element.
- Each element is linked to the following one.
- The last element points to NULL.



- How would you implement this?

Linked List - implementation

```
struct node{
      int data;
      struct node * next;
};

typedef struct node node_t;


node_t x1;
x1.data = 5;

x1.next = NULL;

node_t x2;
x2.data = 7;

x1.next = &x2;

x2.next=NULL;
```

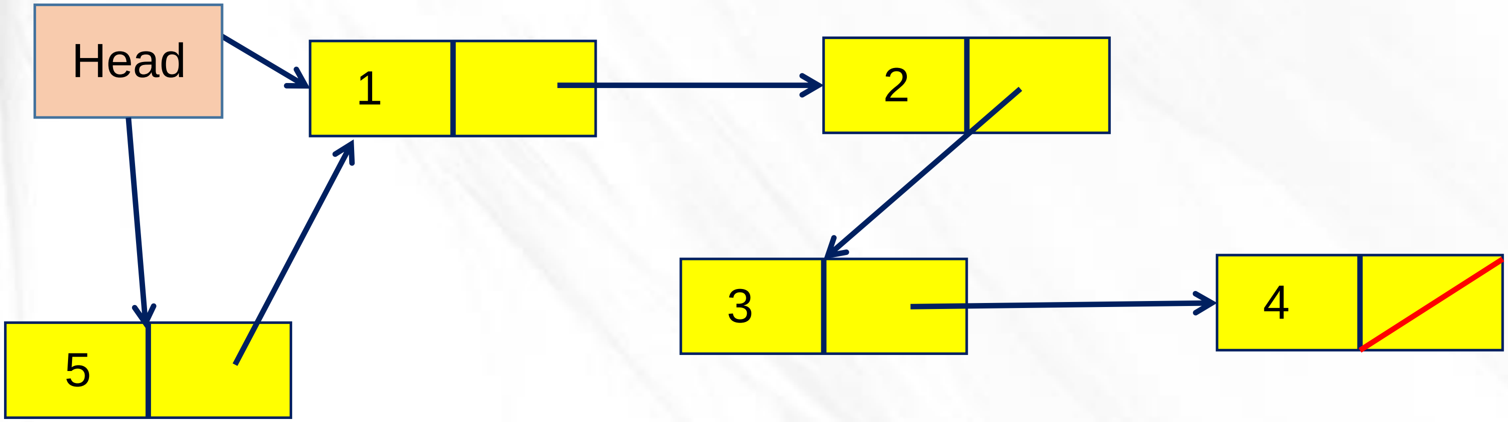


Linked List - implementation

```
struct node{
      int data;
      struct node * next;
};

typedef struct node node_t;


typedef struct {
      struct node_t *head;
} LL_t;


LL_t* LLcreate();

void LL_add_to_head(LL_t* list, int data);

…
```

Linked List – add to head

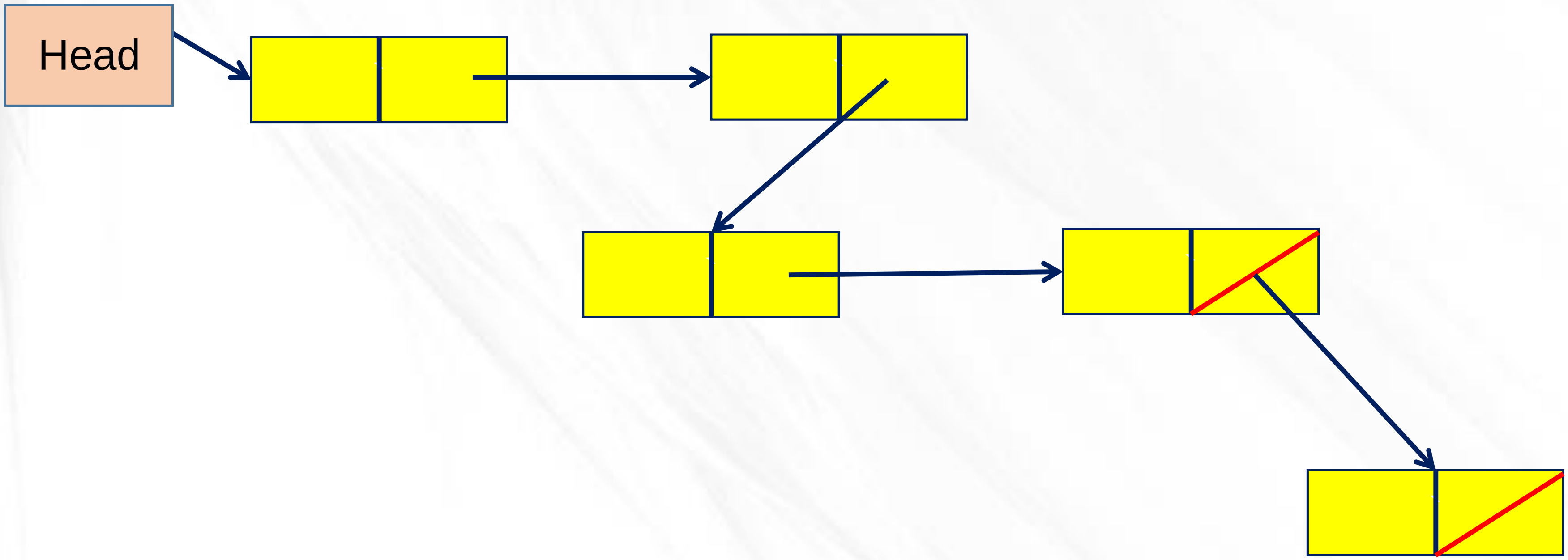
```
LL_add_to_head(LL_t* list, int value) {
    node_t* newNode = create_new_node(value);
    newNode->next = list->head;
    list->head = newNode;
```



Linked List

```
typedef struct {
	struct node_t *head;
} LL_t;
```

Q: What if we want to add an element to the tail of the list?



Linked List

- What operations can we perform on LinkedList?
  - Add a new element at the front
  - Add a new element at the end
  - Insert an element after a specific node
  - Remove a given node
  - Find element
  - Return element in a specified index
  - Get size

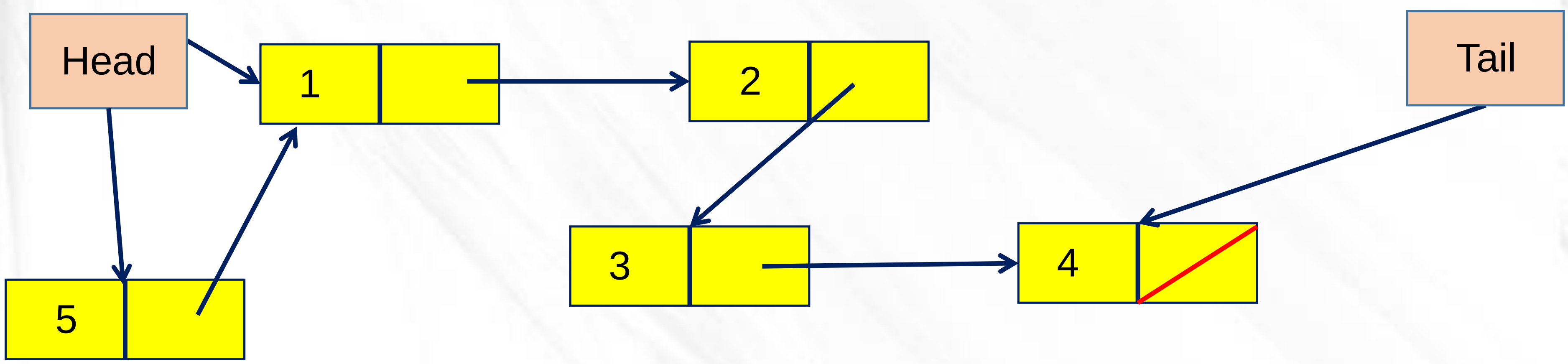
Linked List - implementation

```
typedef struct {
	struct node_t *head;
	struct node_t *tail;
} LL_t;

void LL_add_to_head(LL_t* list, int data);

void LL_add_to_tail(LL_t* list, int data);

int LL_remove_from_head(LL_t* list);

int LL_remove_from_tail(LL_t* list);

int LL_size(LL_t* list);
```

Linked List – add to head

```
LL_add_to_head(LL_t* list, int value) {
    node_t* newNode = create_new_node(value);
    newNode->next = list->head;
    list->head = newNode;
```

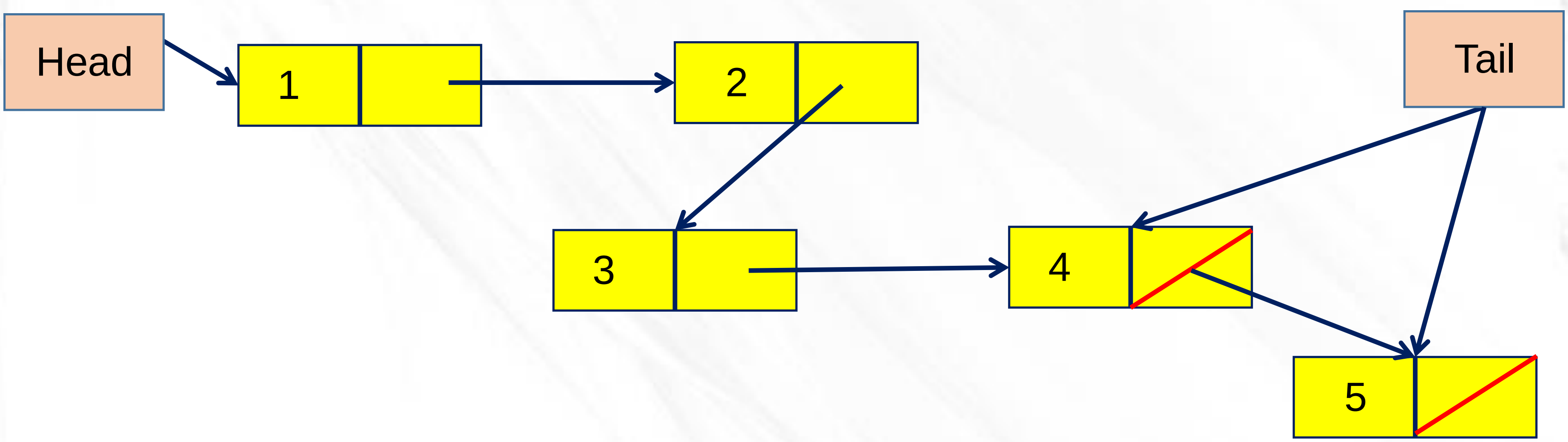
Are we done?



Linked List – add to tail

```
LL_add_to_tail(LL_t* list, int value) {
    node_t* newNode = create_new_node(value);
    newNode->next = NULL;
    list->tail->next = newNode;
    list->tail = newNode;
```

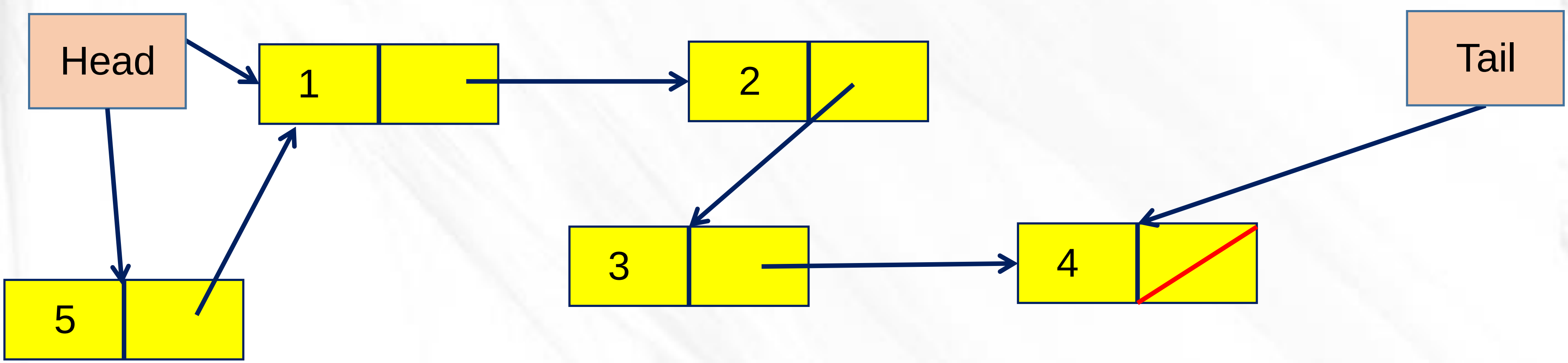
Are we done?



Linked List – remove from head

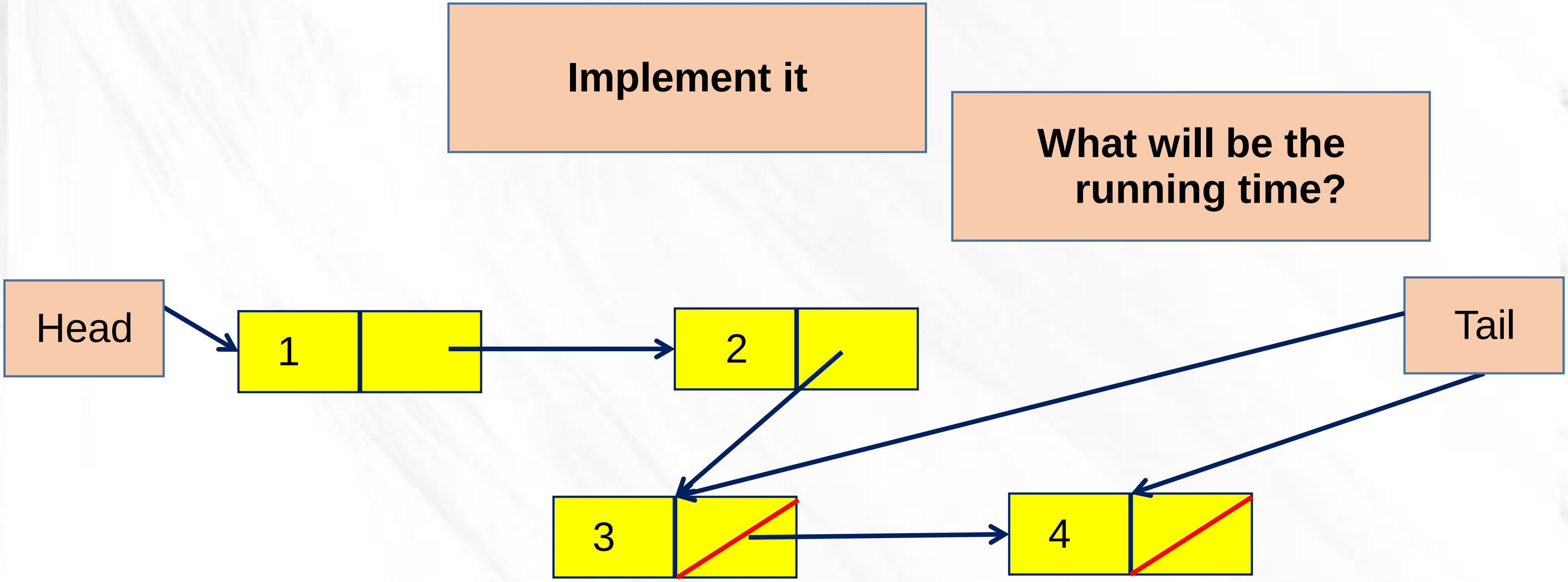
```
LL_remove_from_head(LL_t* list) {
    list->head = list->head->next;
```

Are we done?



Linked List – remove from tail

LL_remove_from_tail(LL_t* list) {



Linked List

- More operations can we perform on LinkedList:
    - Add a new element at the front
    - Add a new element at the end
    - Insert an element after a specific node
    - Remove a given node
    - Find element
    - Return element in a specified index

Back to Stack

Implementing Stack

- Question: Implement Stack so that each operation takes O(1) time.

- Remember: the implementation using arrays required resizing, which took linear time (in some push operations)

Implementing Stack

- Use linked list:
  - Create Stack:
    - Create a linked list
  - Push(item)
    - Add the item to the head of the list
  - Pop()
    - Remove an item from the head of the list

What is the running time of each operation?

Back to Queue

Implementing Queue

- <u>Question:</u> Implement Queue so that each operation takes O(1) time.

- Remember: the implementation using arrays required resizing, which took linear time (in some enqueue operations)

Implementing Queue

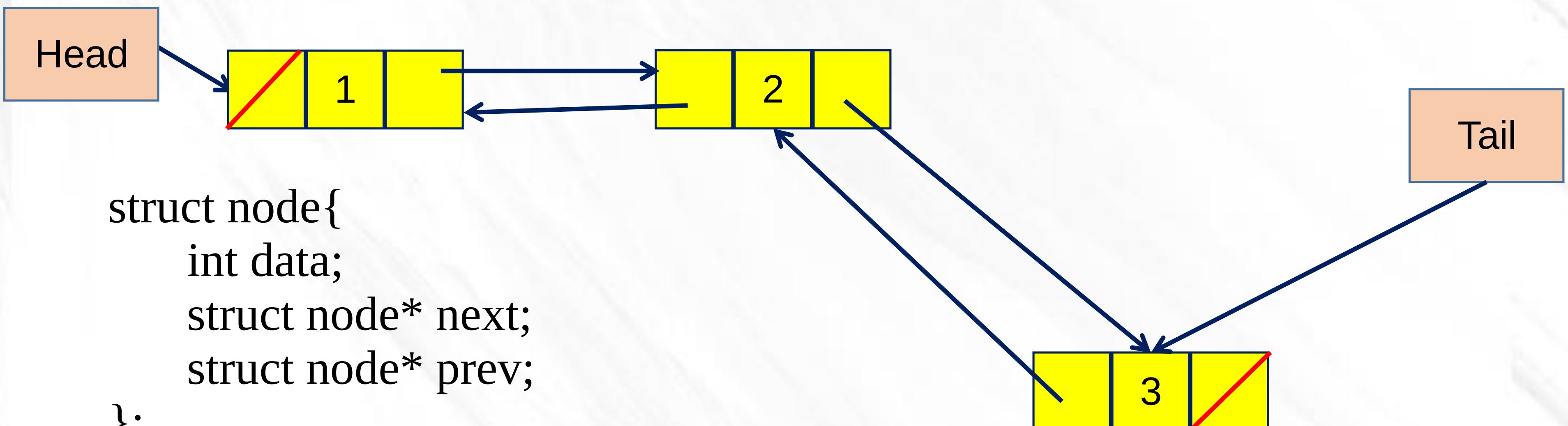
- Use linked list:
  - Create Queue:
    - Create a linked list
  - Enqueue(item)
    - TAIL
    - Add the item to the ~~head~~ of the list
    - **O(1)**
  - Dequeue()
    - HEAD
    - Remove an item from the ~~tail~~ of the list
    - **O(1)**
    - **of queue) !!!**

What is the running time of each operation?

Doubly Linked List

Doubly Linked List

- <u>Double Linked List</u> - similar to linked list.
  Each element is linked to the next element and to the previous element.
- Can traverse the list forward and backward!

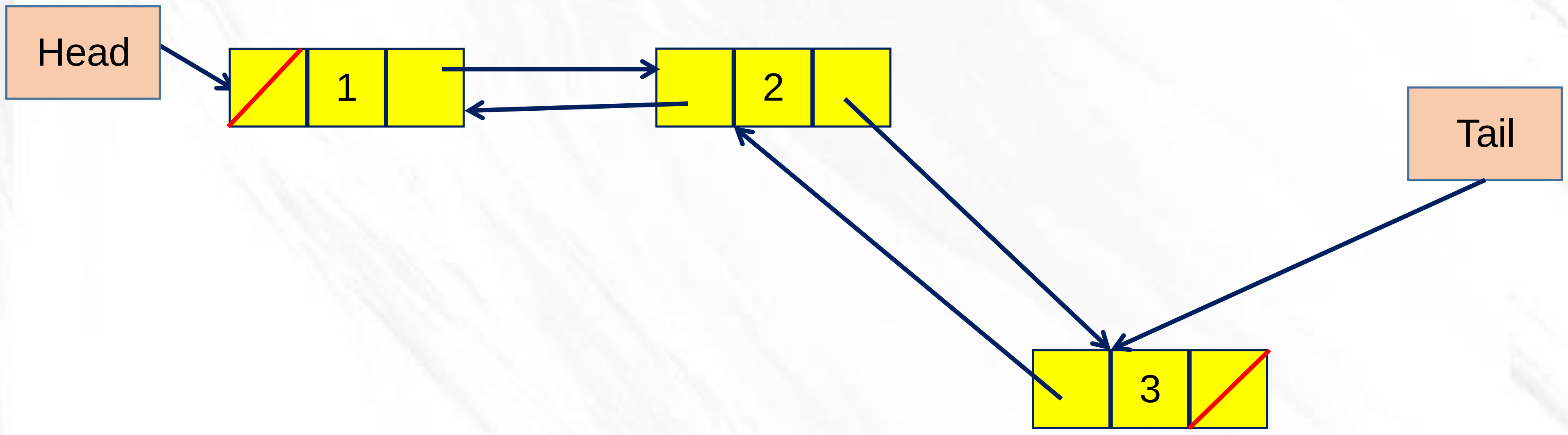



```
struct node{
      int data;
      struct node* next;
      struct node* prev;
};
```

- Homework: Implement this!

Doubly Linked List

```
struct node{
        int data;
        struct node* next;
        struct node* prev;
};
```



Doubly Linked List

- More operations can we perform on LinkedList:
  - Add a new element at the front
  - Add a new element at the end
  - Remove from the front
  - Remove from the tail
  - Insert an element after a specific node
  - Remove a given node

A comment on recursion

Comment on recursion

- How is recursion really implemented inside?
  - What happens when we make a recursive call?

- Wrong answer: my laptop delegates the subtask to other laptops.

- Correct answer: the subtask is added on execution stack:
  - The order in which the functions are called,
  - Where to return after the function ends
  - All the local variables of the function

- In particular, any recursive function can be implemented *non-recursively* using stack.

Comment on recursion

- Thumb rules for writing recursion:
  - Base case: if the problem minimal/not decomposable, solve the problem directly. Checking it should be (typically) the first line of the function.
  - *Examples*:
    empty array / n=0 / index out of bounds
  - Induction case: decompose the problem into one or more similar, STRICTLY smaller sub-problems.
  - *Examples*:
    apply induction on first half of the array, then on the second half of the array
    apply induction on n-1…
  - **BAD IDEAS**: do not use static/global variables in recursion.
  - These are bad practice, and very hard to follow
  - Often fails if the function is invoked several times.

While we are on the subject

While we are on the subject

Do not use **global** variables.

- Only exceptions: global constants

Do not use **static** variables.

- Only exceptions: when we need a shared state for objects/functions

In particular, do not use strtok()

- Ever!
- strtok() uses static variable inside.
- Why is the first call *strtok(str, s)* and then *strtok(NULL, s)*?
- Looks very suspicious.
- It uses static variable to remember the previous call.

- *In general, only use library functions you can write yourself.*

Comment on recursion

Any recursive function can be implemented *non-recursively* using stack.

Example:

Quick sort(A[0… N-1])

pivot_ind = rearrange(A, N)

Quick sort(A[0…pivot_ind-1])

Quick sort(A[pivot_ind+1… N-1])

Implement this without using recursion.

Quick sort without using recursion

```
Quick sort( A[0… N-1] )  {

    // s will contain the indices of subarrays that we need to be sorted
    s = create_stack();  // stack of pairs of indices

    stack_push(s, (pair){0 , N-1});

    while (s is not empty)  {

        ( i , j ) = stack_pop(s);

        pivot_ind = rearrange( A[i…j], pivot_ind); //

        if (pivot_ind+1 < j)
            stack_push(s, (pair){pivot_ind+1 , j});

        if (i < pivot_ind-1)
            stack_push(s,  (pair){i , pivot_ind-1} );

    }

}
```

Homework: run this algorithm on several examples of arrays
In each step follow the state of the array and the state of the stack

Questions?

Comments?