

ECE220

Lecture x0012 - 03/26

Linked lists - stacks & queues

Recap

Recap

- Last week - Tuesday

Recap

- Last week - Tuesday
 - Dynamic memory allocation:
`malloc`, `calloc`,
`realloc`, `free`

Recap

- Last week - Tuesday
 - Dynamic memory allocation:
`malloc`, `calloc`,
`realloc`, `free`
 - Two dimensional arrays

Recap

- Last week - Tuesday
 - Dynamic memory allocation:
`malloc`, `calloc`,
`realloc`, `free`
 - Two dimensional arrays
 - Reading/writing `structs` to
files

Recap

- Last week - Tuesday
 - Dynamic memory allocation:
`malloc`, `calloc`,
`realloc`, `free`
 - Two dimensional arrays
 - Reading/writing `structs` to
files
 - Examples

Recap

- Last week - Tuesday
 - Dynamic memory allocation:
`malloc`, `calloc`,
`realloc`, `free`
 - Two dimensional arrays
 - Reading/writing `structs` to
files
 - Examples
- Last week - Thursday

Recap

- Last week - Tuesday
 - Dynamic memory allocation:
`malloc`, `calloc`,
`realloc`, `free`
 - Two dimensional arrays
 - Reading/writing `structs` to
files
 - Examples
- Last week - Thursday
 - Linked lists

Recap

- Last week - Tuesday
 - Dynamic memory allocation:
`malloc`, `calloc`,
`realloc`, `free`
 - Two dimensional arrays
 - Reading/writing `structs` to
files
 - Examples
- Last week - Thursday
 - Linked lists
 - Traversal

Recap

- Last week - Tuesday
 - Dynamic memory allocation: `malloc`, `calloc`, `realloc`, `free`
 - Two dimensional arrays
 - Reading/writing `structs` to files
 - Examples
- Last week - Thursday
 - Linked lists
 - Traversal
 - Insertion - head, tail, sorted

Recap

- Last week - Tuesday
 - Dynamic memory allocation: `malloc`, `calloc`, `realloc`, `free`
 - Two dimensional arrays
 - Reading/writing `structs` to files
 - Examples
- Last week - Thursday
 - Linked lists
 - Traversal
 - Insertion - head, tail, sorted
 - Deletion - head, tail, middle

Reminders

Reminders

- **Exam on 03/28, study study!**

Reminders

- **Exam on 03/28**, study study!
 - Study material has been posted

Reminders

- **Exam on 03/28**, study study!
 - Study material has been posted
 - Lectures 7 through 15 inclusive

Reminders

- **Exam on 03/28**, study study!
 - Study material has been posted
 - Lectures 7 through 15 inclusive
 - HKN review session material is available

Reminders

- **Exam on 03/28**, study study!
- About the exam
 - Study material has been posted
 - Lectures 7 through 15 inclusive
 - HKN review session material is available

Reminders

- **Exam on 03/28**, study study!
 - Study material has been posted
 - Lectures 7 through 15 inclusive
 - HKN review session material is available
- About the exam
 - Paper-format (same as last time)

Reminders

- **Exam on 03/28**, study study!
 - Study material has been posted
 - Lectures 7 through 15 inclusive
 - HKN review session material is available
- About the exam
 - Paper-format (same as last time)
 - Four questions

Reminders

- **Exam on 03/28**, study study!
 - Study material has been posted
 - Lectures 7 through 15 inclusive
 - HKN review session material is available
- About the exam
 - Paper-format (same as last time)
 - Four questions
 - Arrays & recursion (in C)

Reminders

- **Exam on 03/28**, study study!
 - Study material has been posted
 - Lectures 7 through 15 inclusive
 - HKN review session material is available
- About the exam
 - Paper-format (same as last time)
 - Four questions
 - Arrays & recursion (in C)
 - C2LC3 conversion, concept questions

Review - singly linked lists (plain)

Review - singly linked lists (plain)

```
add_at_head
```

```
if head==NULL
```

```
...
```

```
else
```

```
....
```


Review - singly linked lists (plain)

add_at_head

if head==NULL

...

else

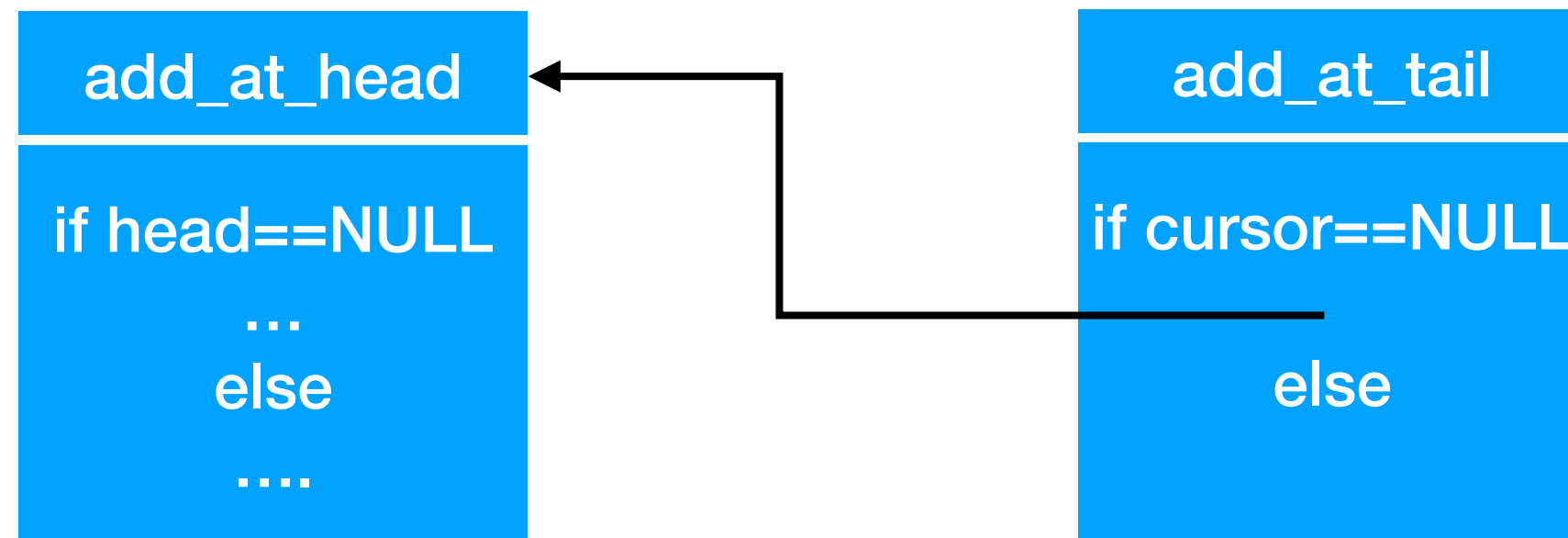
....

add_at_tail

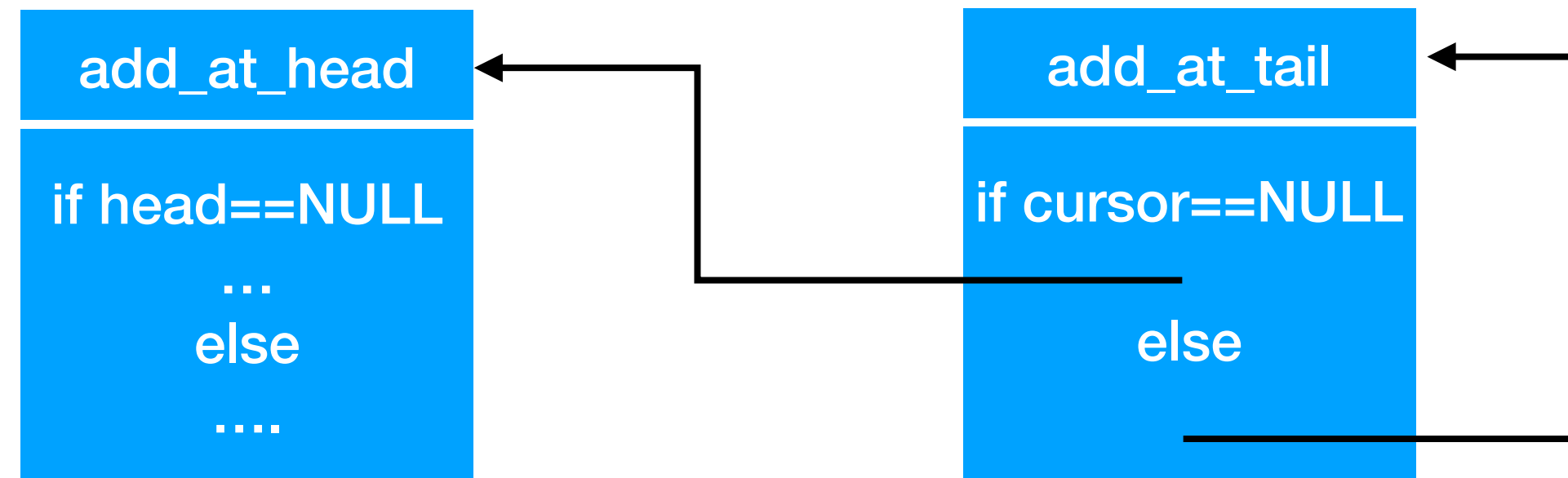
if cursor==NULL

else

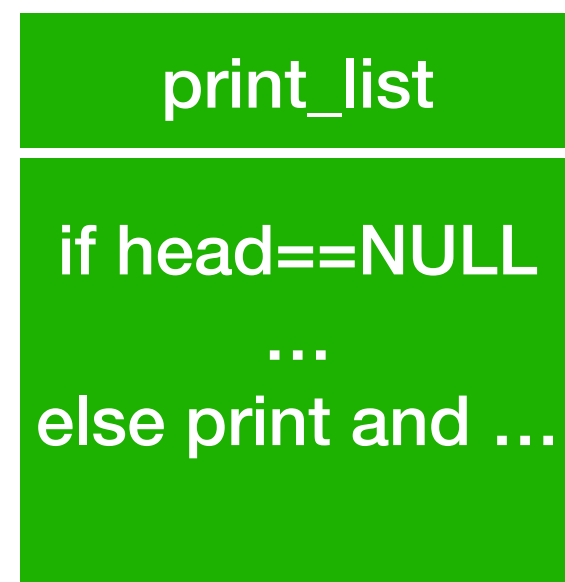
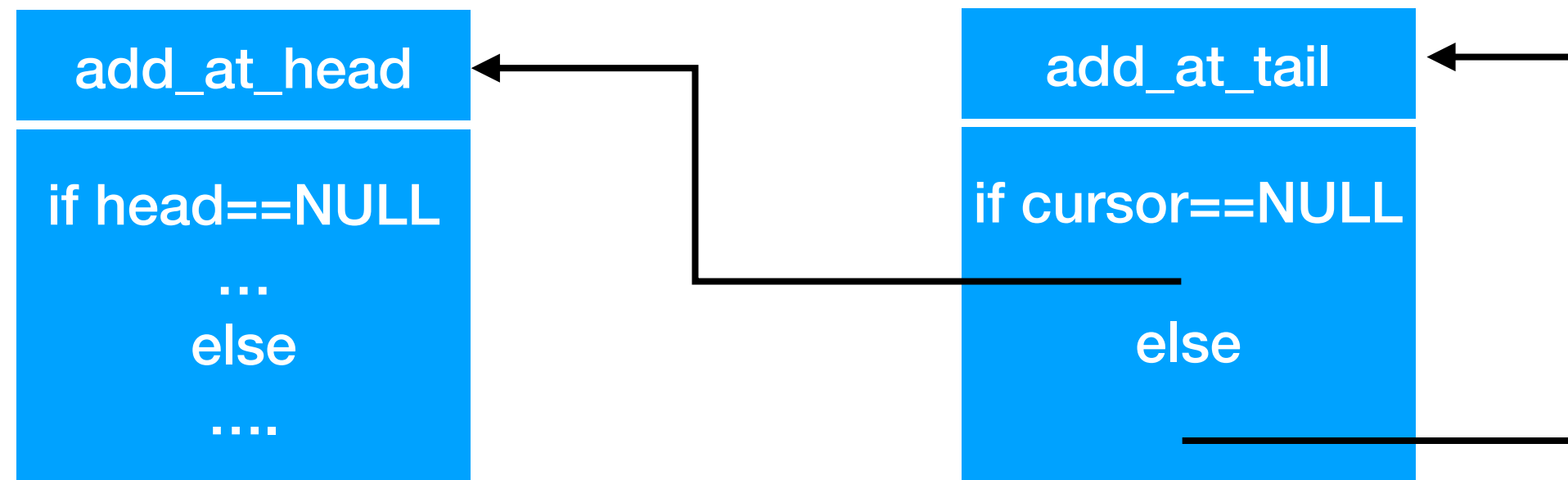
Review - singly linked lists (plain)



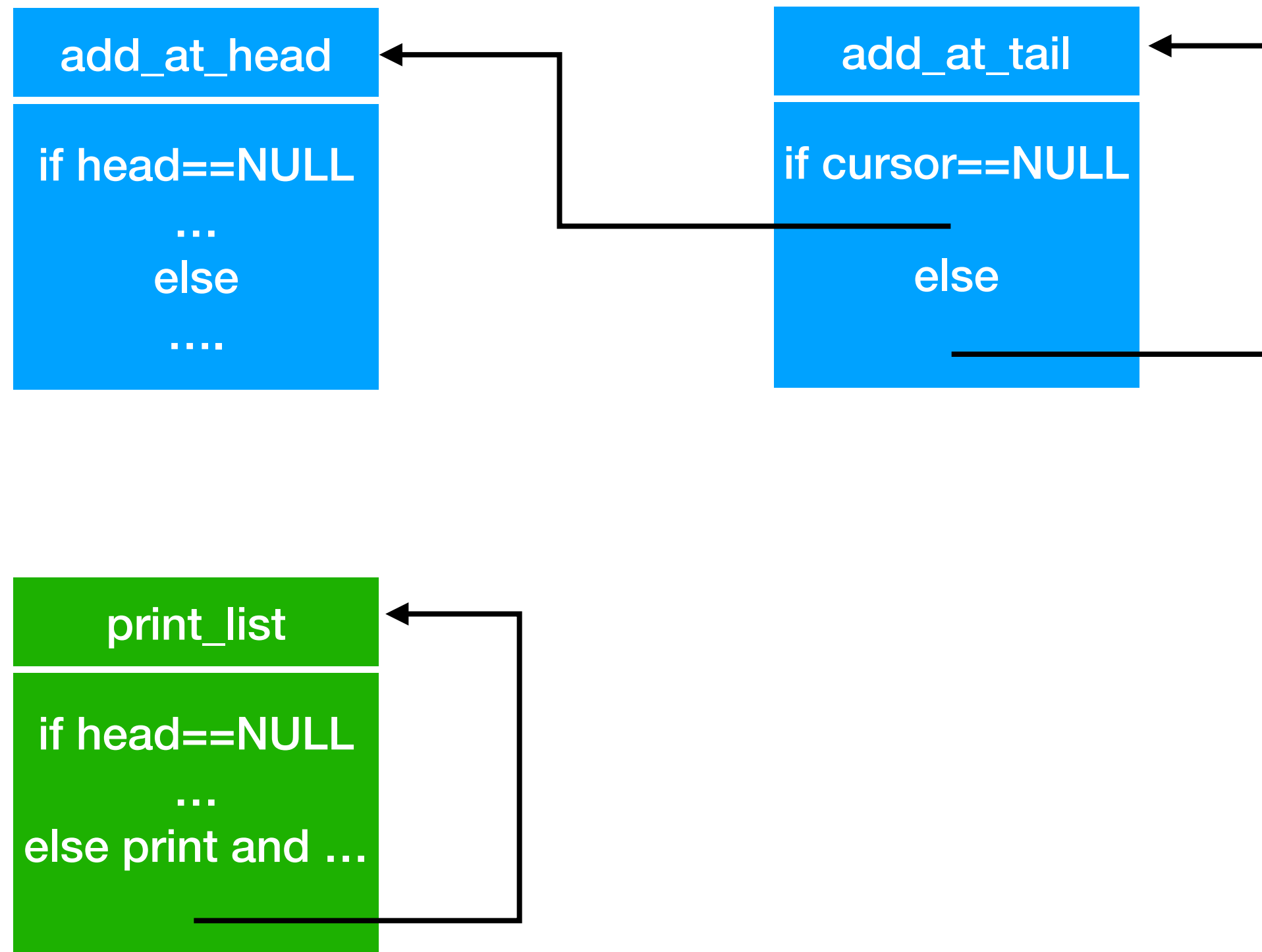
Review - singly linked lists (plain)



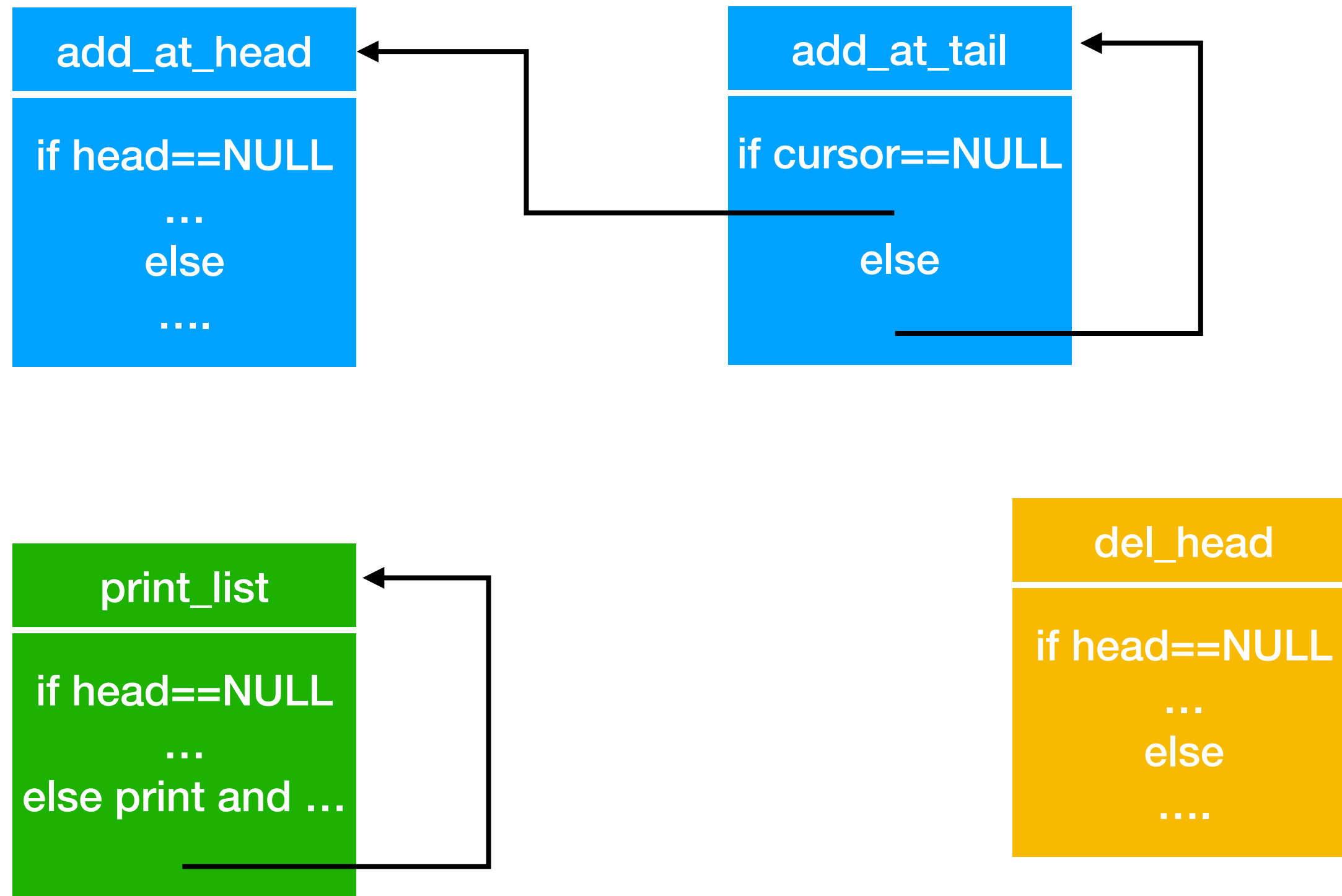
Review - singly linked lists (plain)



Review - singly linked lists (plain)



Review - singly linked lists (plain)



Review - singly linked lists (plain)

```
add_at_head
if head==NULL
...
else
....
```

```
add_at_tail
if cursor==NULL
else
....
```

```
print_list
if head==NULL
...
else print and ...
```


```
del_head
if head==NULL
...
else
....
```

```
del_tail
if cursor==NULL
...
elif cursor->next==NULL
....
else .... while loop to
second_last
```

Review - singly linked lists (sorted)

```
add_at_head
if head==NULL
...
else
....
```

```
print_list
if head==NULL
...
else print and ...
```




A diagram consisting of a black line that starts from the bottom of the 'print_list' block, extends horizontally to the right, then vertically upwards, and finally horizontally to the left, ending with an arrowhead pointing to the left side of the 'add_at_head' block.

Review - singly linked lists (sorted)

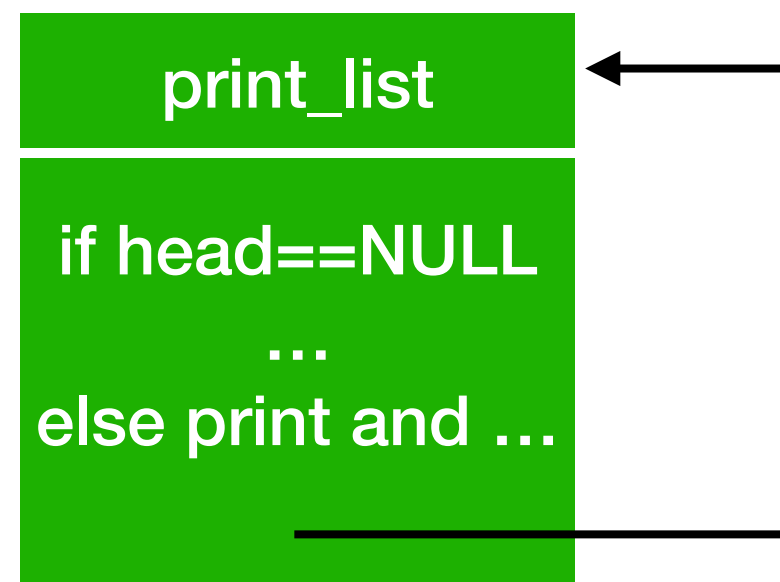
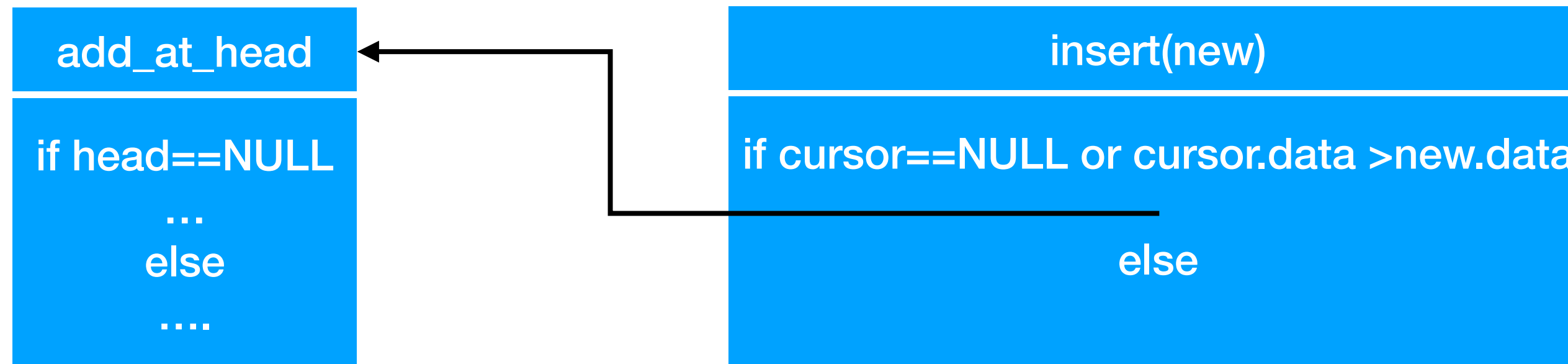
```
add_at_head
if head==NULL
...
else
....
```

```
insert(new)
if cursor==NULL or cursor.data >new.data
else
```

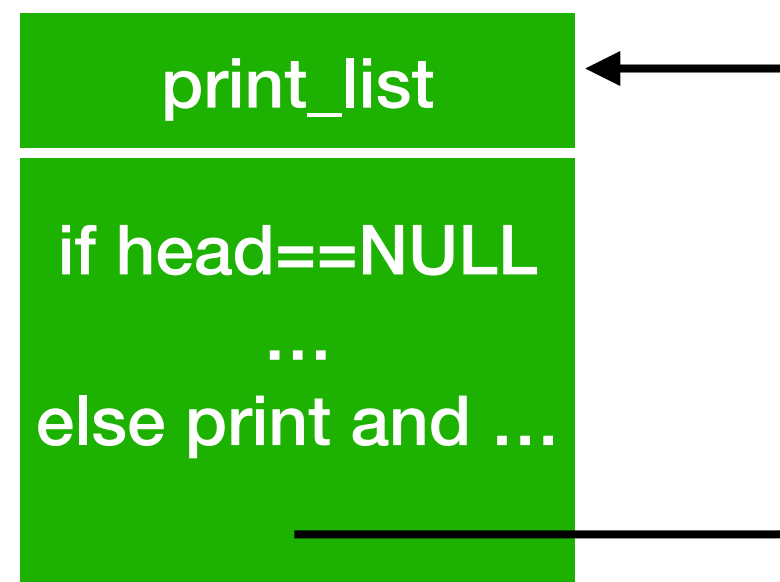
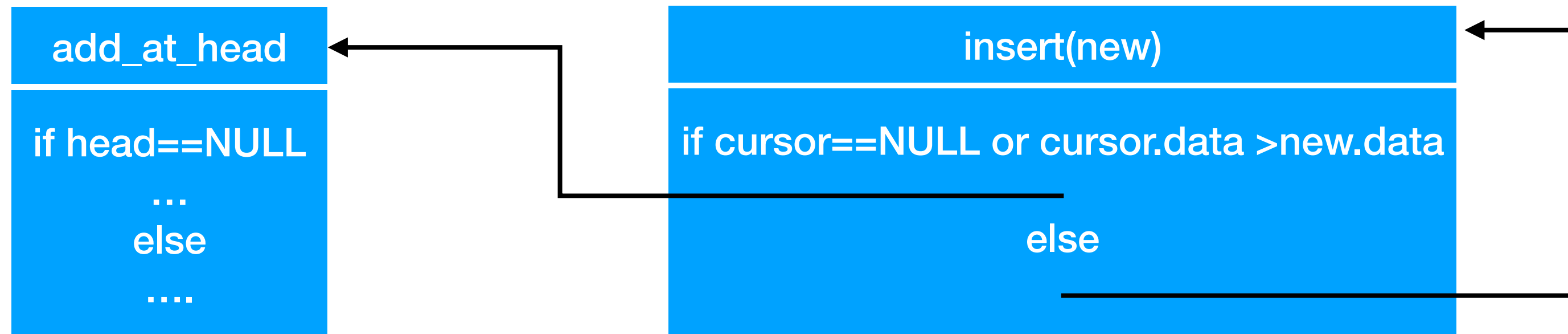
```
print_list
if head==NULL
...
else print and ...
```



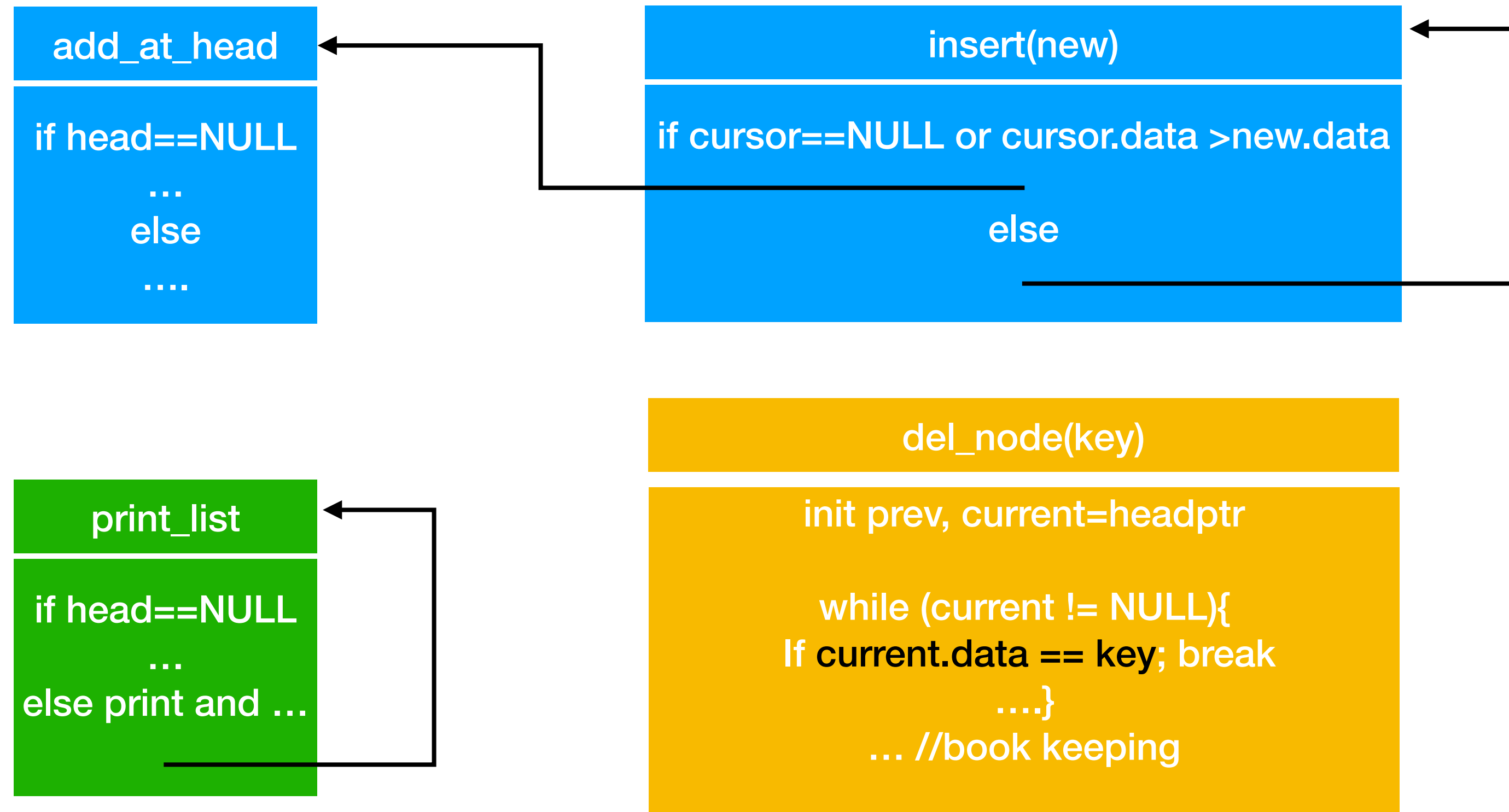
Review - singly linked lists (sorted)



Review - singly linked lists (sorted)



Review - singly linked lists (sorted)




Cons cells

- Covering last few slides from 03/21...

Well time didn't permit ... the next three slides can be skipped. I just think cons cells are cool 🐾

Time permitting ...

- Cons cells - the original take on linked lists
 - First introduced in 1960 by the Lisp programming language
 - A cons cell, by default, adds items to the beginning of the list:
 - `cons(1, cons(2, (cons 3)))` gives 1->2->3
 - The function `cons` takes a value and pointer to head of list as input
 - The function `cons` always returns pointer to head of list

ECE 220 - Spring 2024 Dr. Ivan Abraham 31 Optional reading 

Stack using linked lists

Stack using linked lists

Stack using linked lists

- First item in is the last item out - FILO

Stack using linked lists

- First item in is the last item out - FILO
- Two operations for data movement: **Push & Pop**

Stack using linked lists

- First item in is the last item out - FILO
- Two operations for data movement: **Push & Pop**
- Stack top ~ head pointer/head

Stack using linked lists

- First item in is the last item out - FILO
- Two operations for data movement: **Push & Pop**
- Stack top ~ head pointer/head
- Push ~ add at head

Stack using linked lists

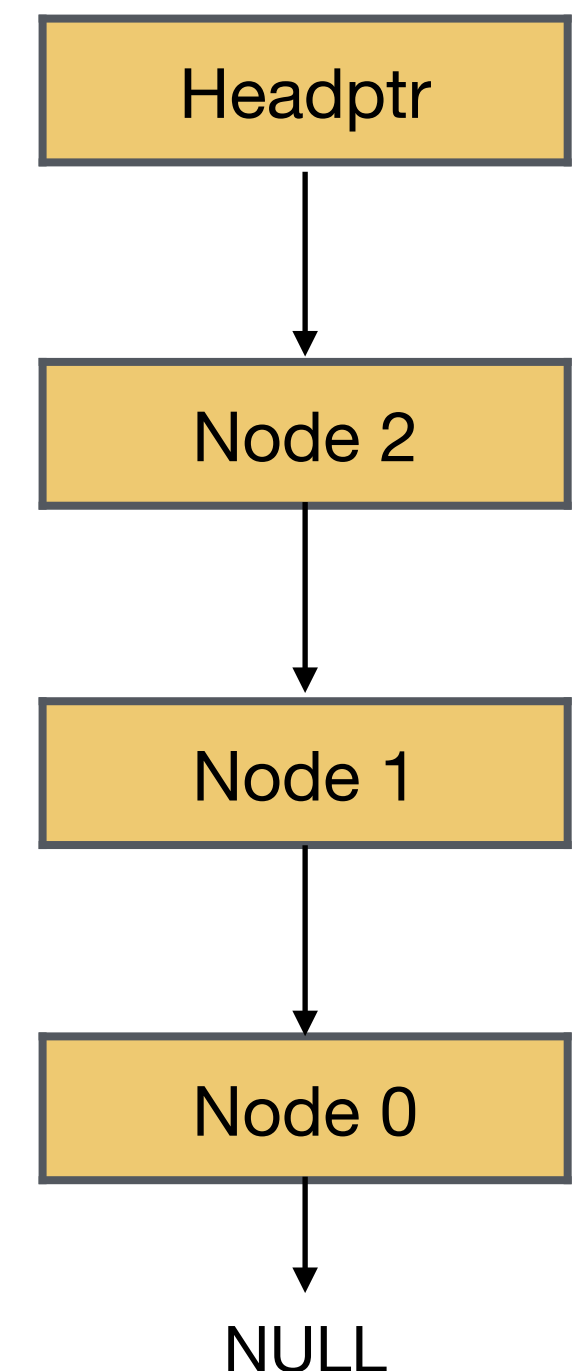
- First item in is the last item out - FILO
- Two operations for data movement: **Push & Pop**
- Stack top ~ head pointer/head
- Push ~ add at head
- Pop ~ remove from head

Stack using linked lists

- First item in is the last item out - FILO
- Two operations for data movement: **Push & Pop**
- Stack top ~ head pointer/head
- Push ~ add at head
- Pop ~ remove from head
 - Need to give popped value to caller

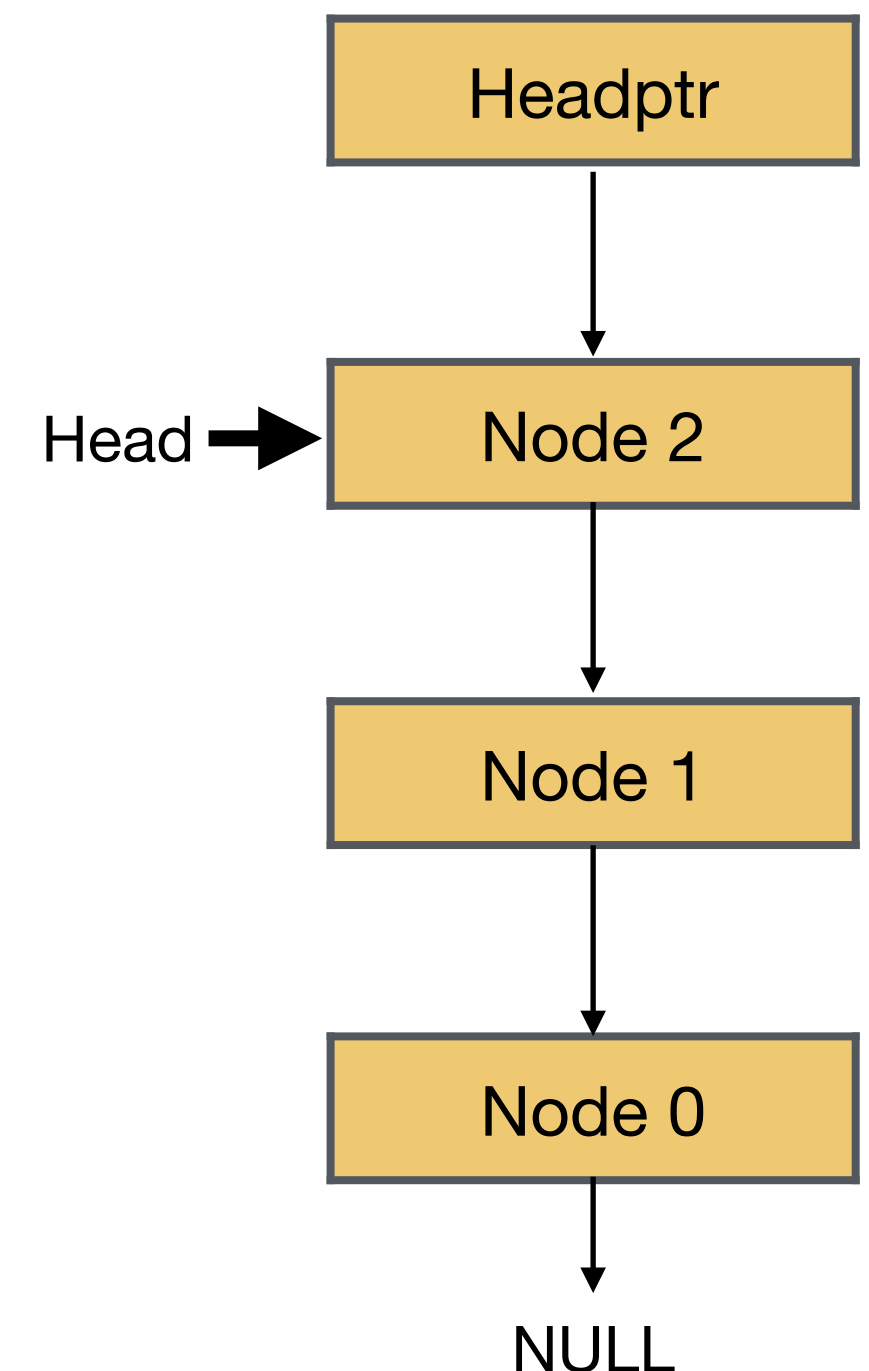
Stack using linked lists

- First item in is the last item out - FILO
- Two operations for data movement: **Push & Pop**
- Stack top ~ head pointer/head
- Push ~ add at head
- Pop ~ remove from head
 - Need to give popped value to caller



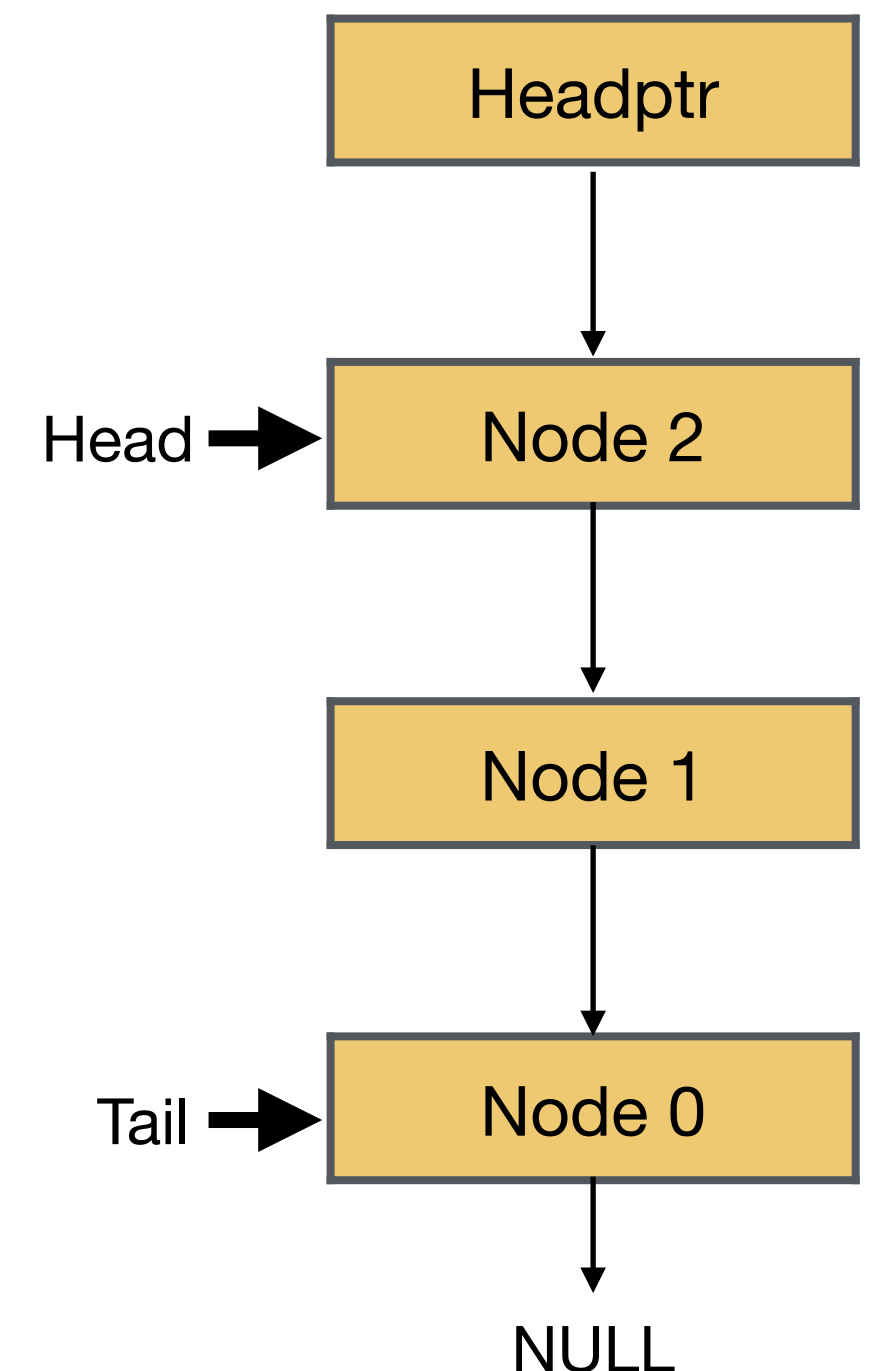
Stack using linked lists

- First item in is the last item out - FILO
- Two operations for data movement: **Push & Pop**
- Stack top ~ head pointer/head
- Push ~ add at head
- Pop ~ remove from head
 - Need to give popped value to caller



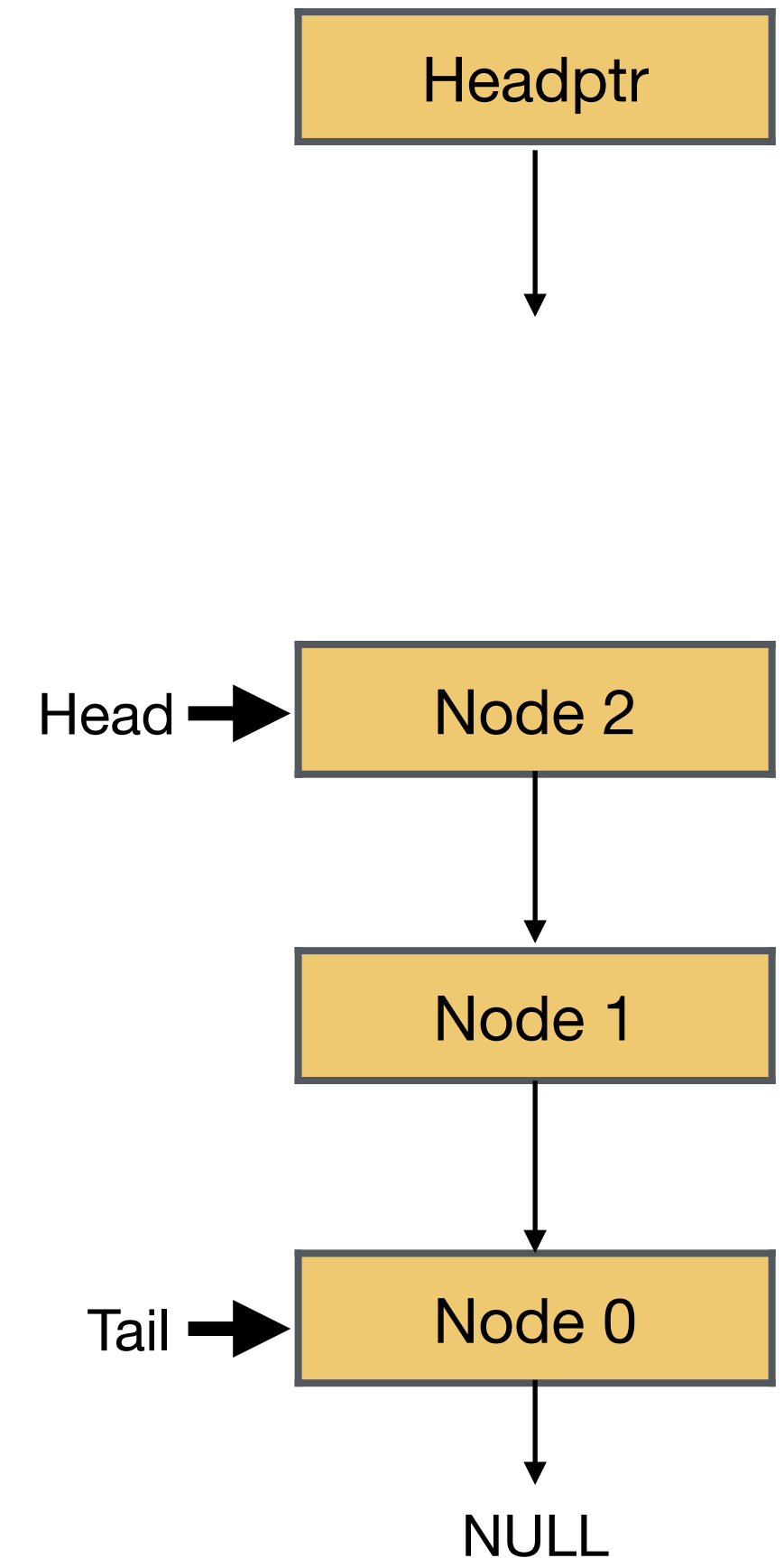
Stack using linked lists

- First item in is the last item out - FILO
- Two operations for data movement: **Push & Pop**
- Stack top ~ head pointer/head
- Push ~ add at head
- Pop ~ remove from head
 - Need to give popped value to caller



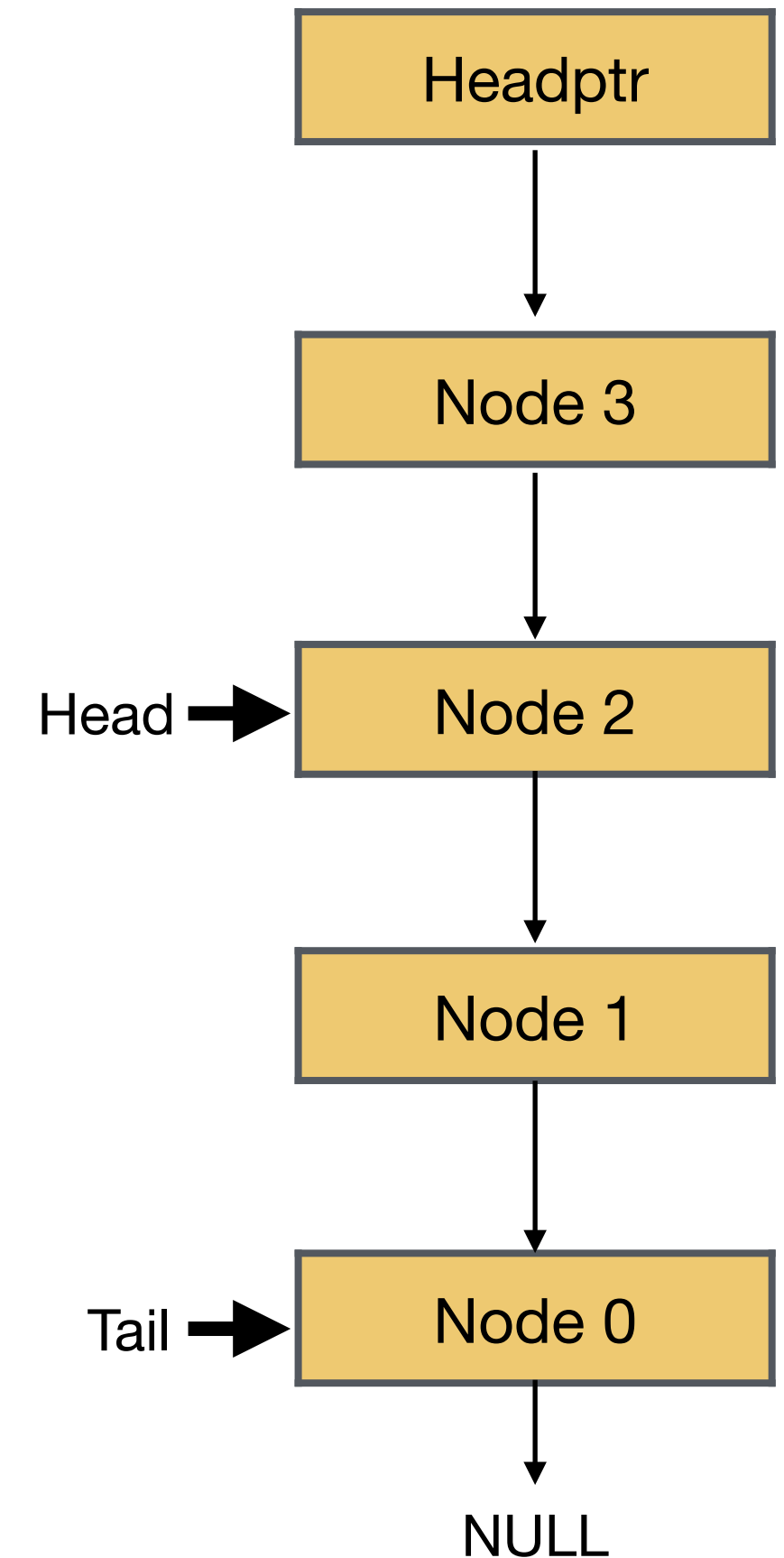
Stack using linked lists

- First item in is the last item out - FILO
- Two operations for data movement: **Push & Pop**
- Stack top ~ head pointer/head
- Push ~ add at head
- Pop ~ remove from head
 - Need to give popped value to caller



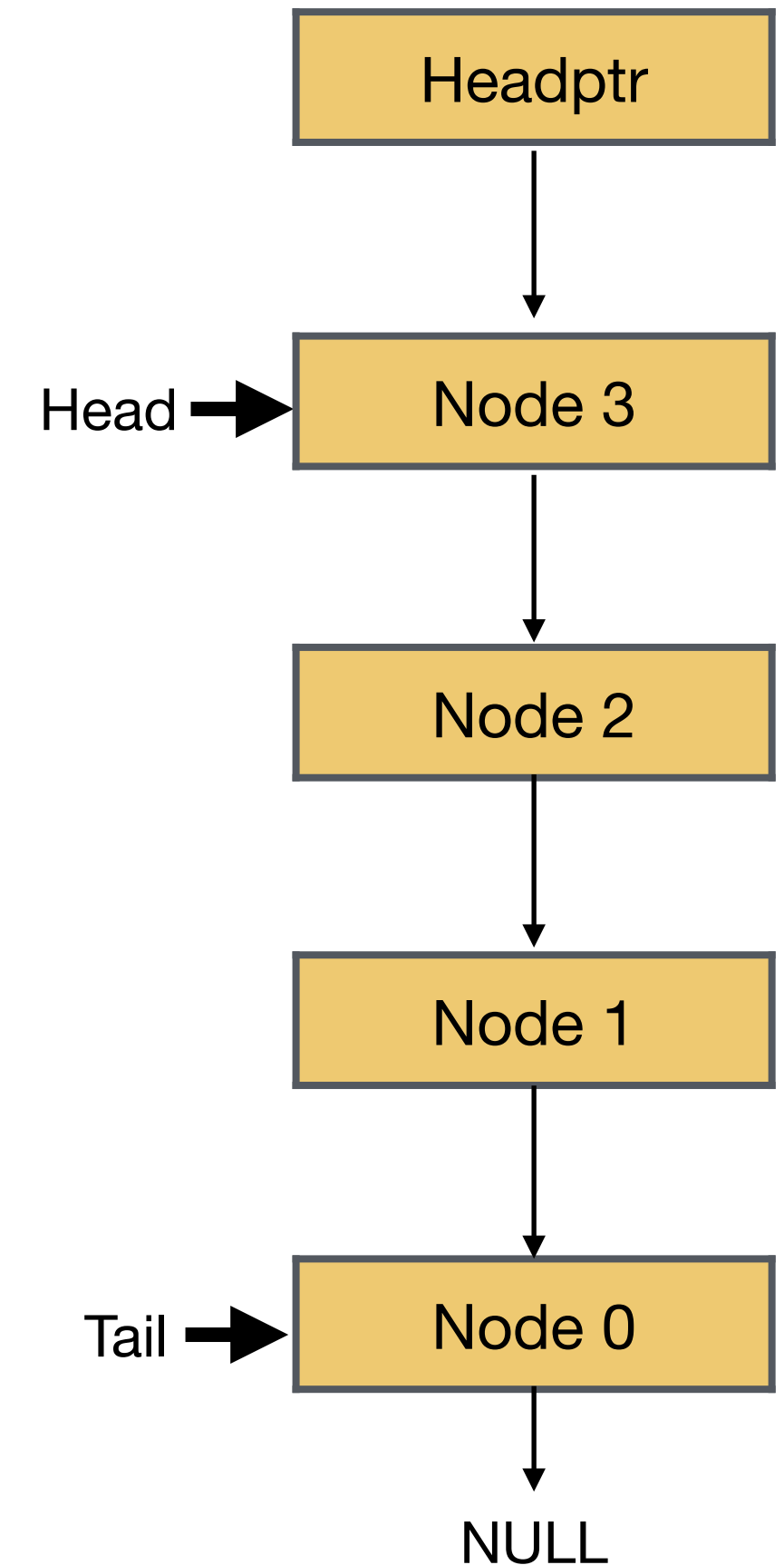
Stack using linked lists

- First item in is the last item out - FILO
- Two operations for data movement: **Push & Pop**
- Stack top ~ head pointer/head
- Push ~ add at head
- Pop ~ remove from head
 - Need to give popped value to caller



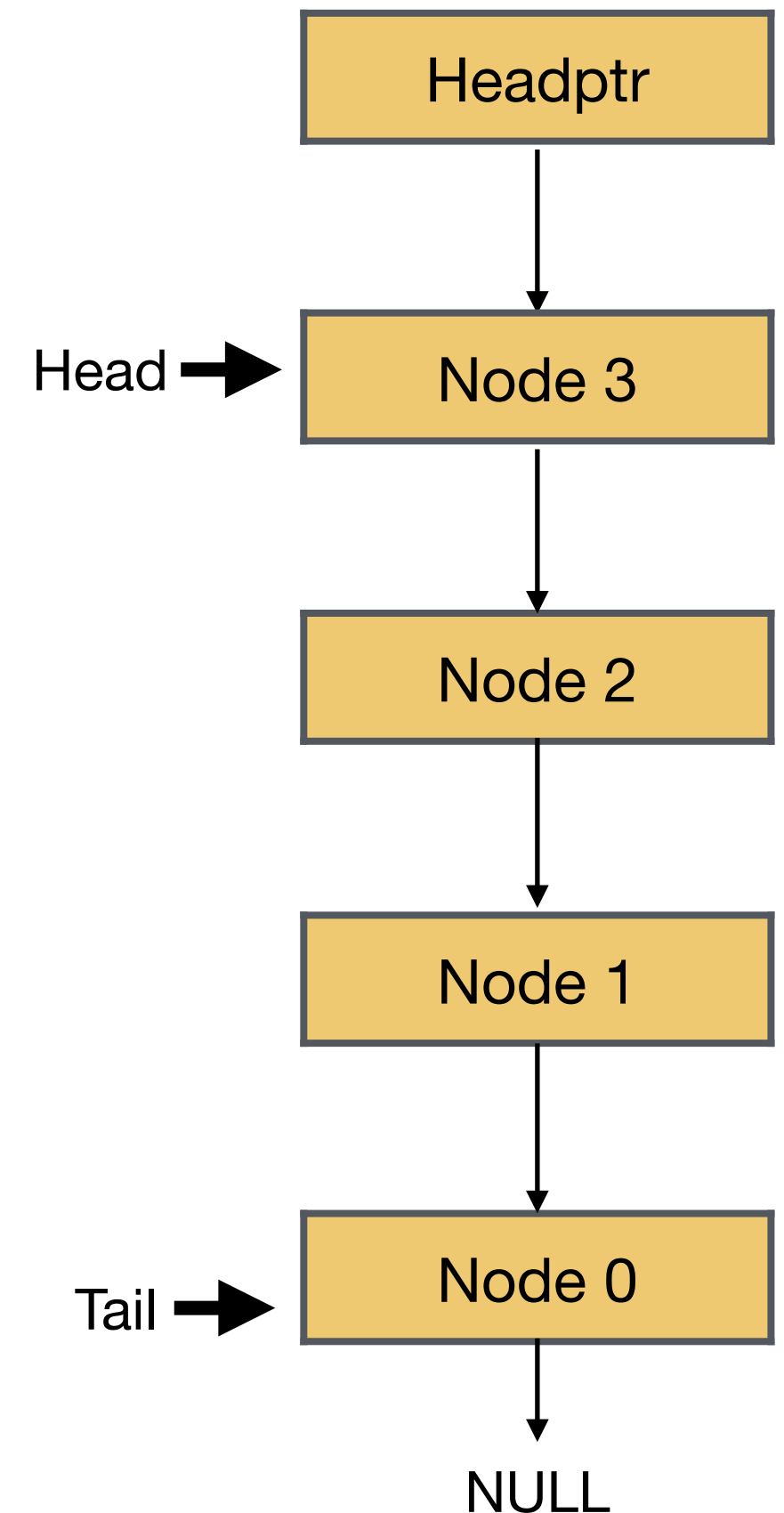
Stack using linked lists

- First item in is the last item out - FILO
- Two operations for data movement: **Push & Pop**
- Stack top ~ head pointer/head
- Push ~ add at head
- Pop ~ remove from head
 - Need to give popped value to caller



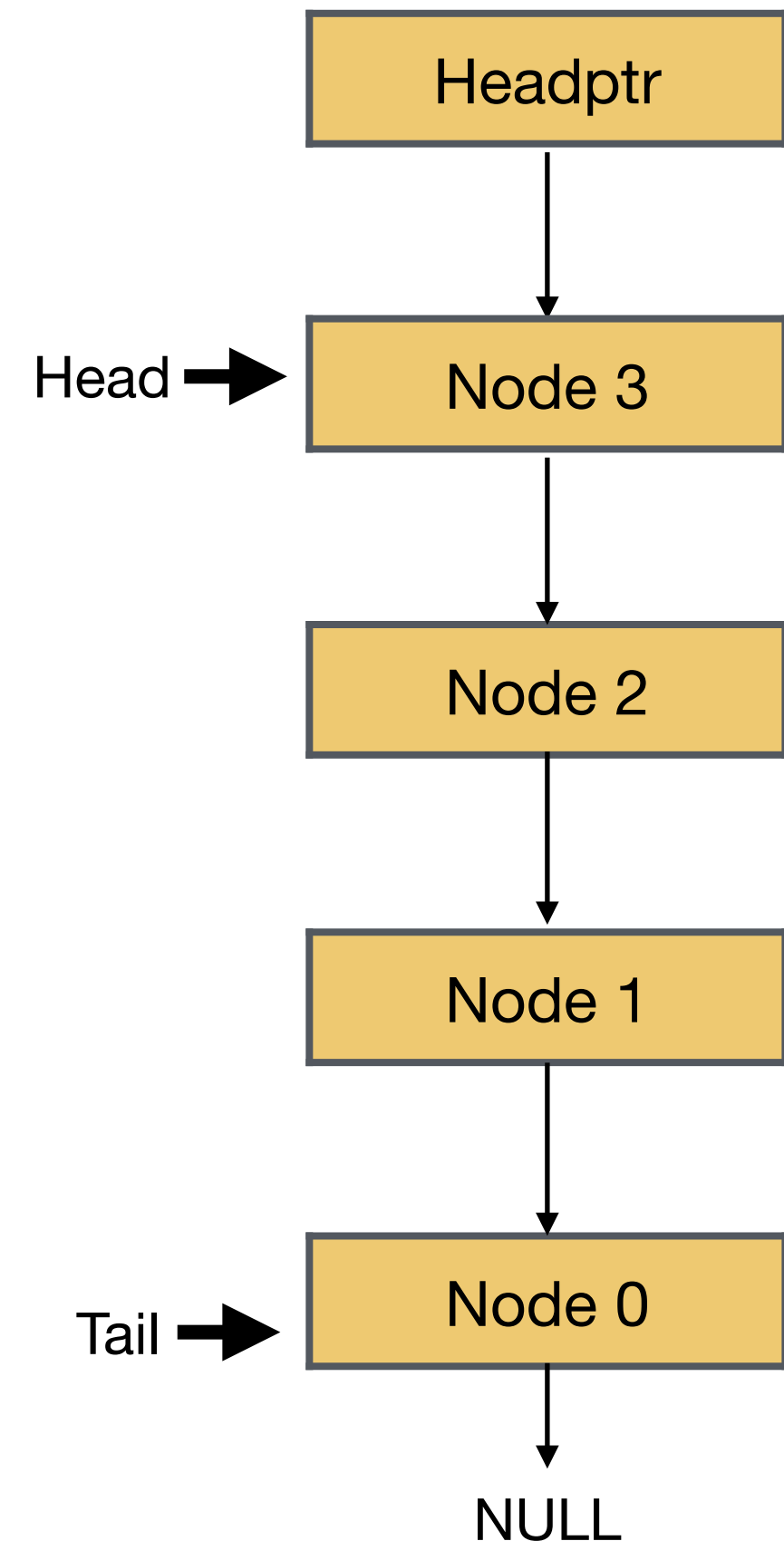
Stack using linked lists

- First item in is the last item out - FILO
- Two operations for data movement: **Push & Pop**
- Stack top ~ head pointer/head
- Push ~ add at head
- Pop ~ remove from head
 - Need to give popped value to caller



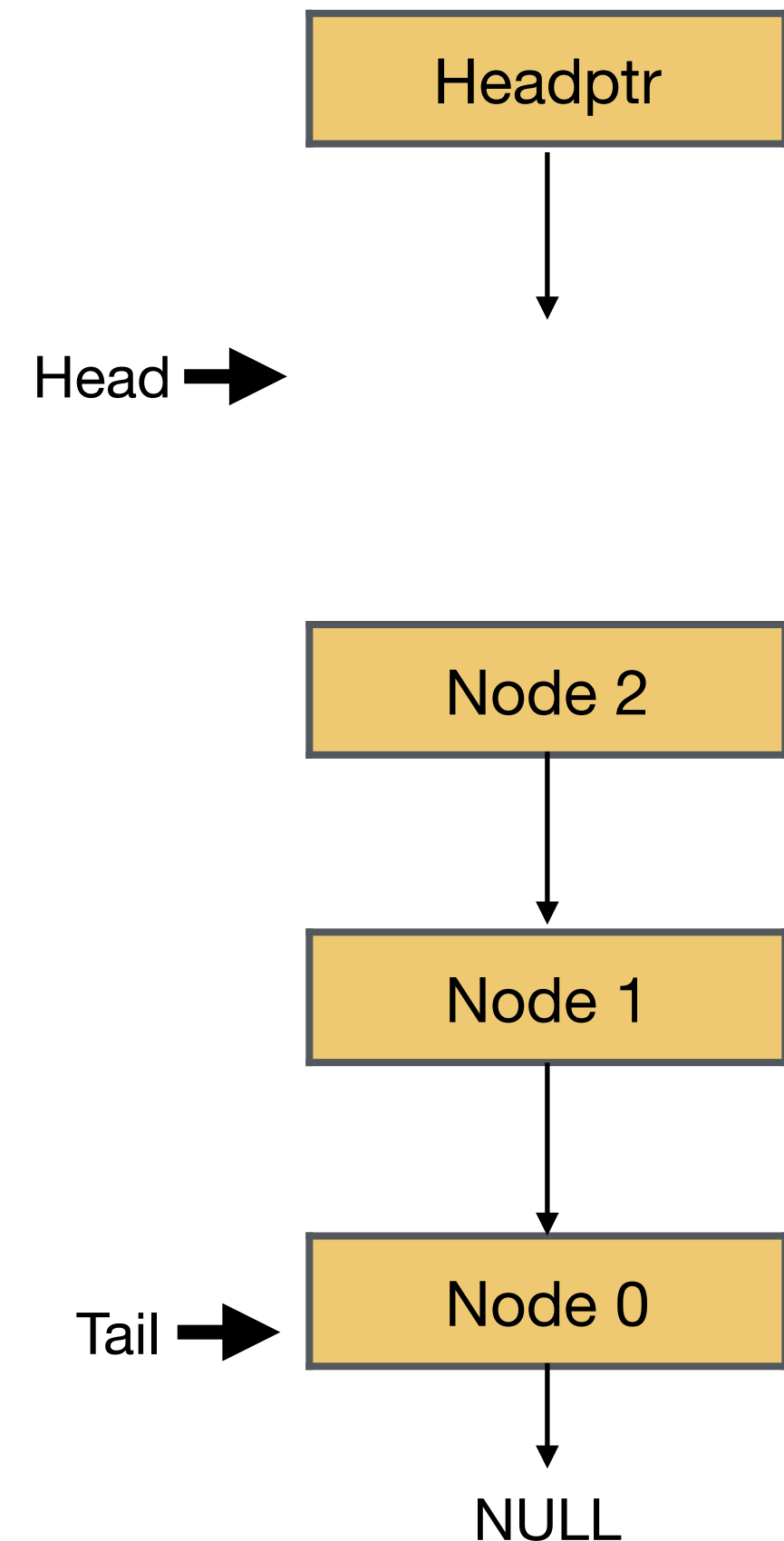
Stack using linked lists

- First item in is the last item out - FILO
- Two operations for data movement: **Push & Pop**
- Stack top ~ head pointer/head
- Push ~ add at head
- Pop ~ remove from head
 - Need to give popped value to caller



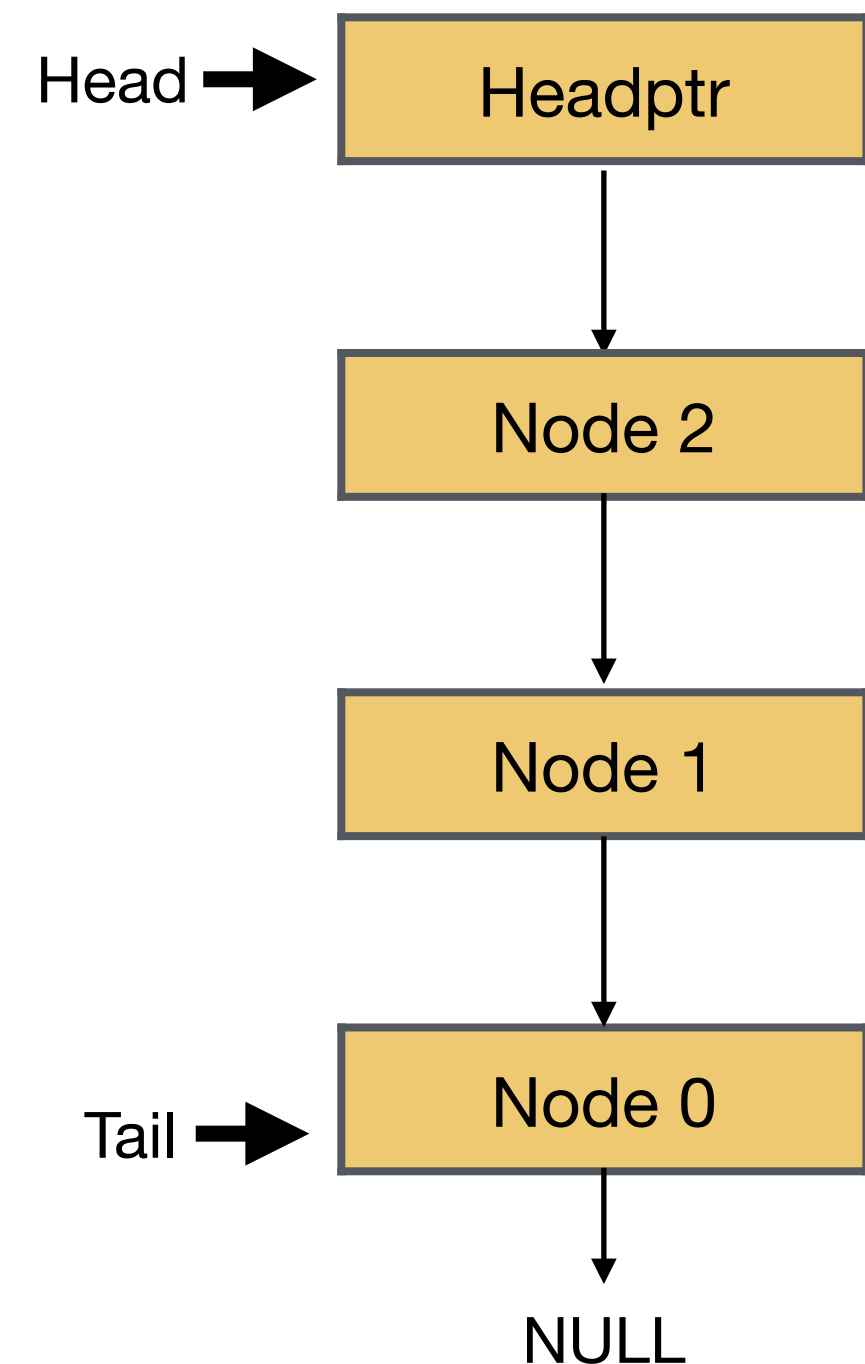
Stack using linked lists

- First item in is the last item out - FILO
- Two operations for data movement: **Push & Pop**
- Stack top ~ head pointer/head
- Push ~ add at head
- Pop ~ remove from head
 - Need to give popped value to caller



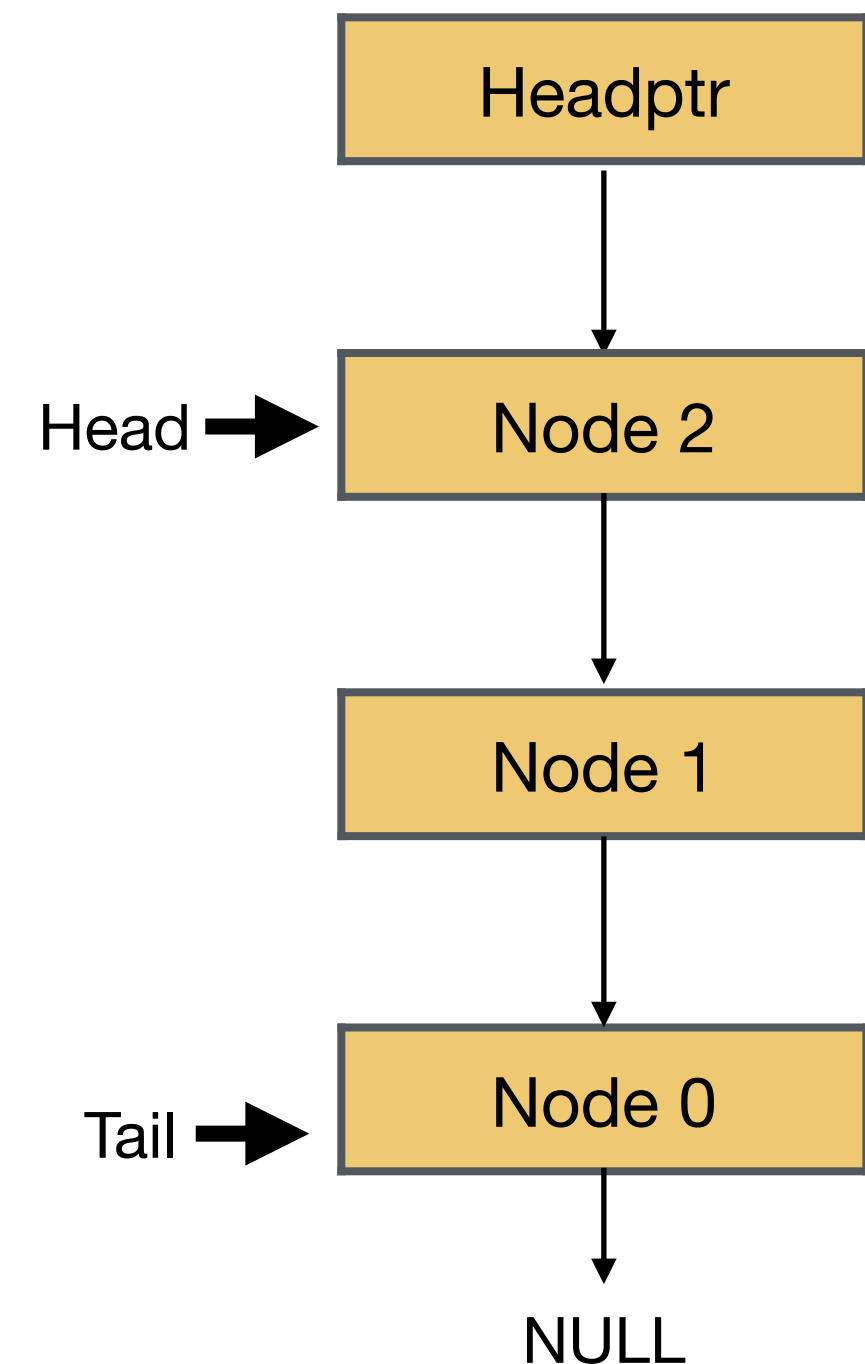
Stack using linked lists

- First item in is the last item out - FILO
- Two operations for data movement: **Push & Pop**
- Stack top ~ head pointer/head
- Push ~ add at head
- Pop ~ remove from head
 - Need to give popped value to caller



Stack using linked lists

- First item in is the last item out - FILO
- Two operations for data movement: **Push & Pop**
- Stack top ~ head pointer/head
- Push ~ add at head
- Pop ~ remove from head
 - Need to give popped value to caller



Stack push using linked lists

Same as insert at head

```
void push(node **cursor, node *new){  
  
    node* temp=(node*) malloc(sizeof(node));  
    temp->name=new->name;  
    temp->byear=new->byear;  
    temp->next=new->next;  
  
    if (cursor == NULL)  
        *cursor = temp;  
    else{  
        temp->next = *cursor;  
        *cursor = temp;  
    }  
}
```

Stack push using linked lists

Same as insert at head

- Suppose we want to **push a node onto stack**.

```
void push(node **cursor, node *new){  
  
    node* temp=(node*) malloc(sizeof(node));  
    temp->name=new->name;  
    temp->byear=new->byear;  
    temp->next=new->next;  
  
    if (cursor == NULL)  
        *cursor = temp;  
    else{  
        temp->next = *cursor;  
        *cursor = temp;  
    }  
}
```

Stack push using linked lists

Same as insert at head

- Suppose we want to **push a node onto stack**.
- What needs to be done?

```
void push(node **cursor, node *new){  
  
    node* temp=(node*) malloc(sizeof(node));  
    temp->name=new->name;  
    temp->byear=new->byear;  
    temp->next=new->next;  
  
    if (cursor == NULL)  
        *cursor = temp;  
    else{  
        temp->next = *cursor;  
        *cursor = temp;  
    }  
}
```

Stack push using linked lists

Same as insert at head

- Suppose we want to **push a node onto stack**.
- What needs to be done?
 - New node should point to current head.

```
void push(node **cursor, node *new){  
  
    node* temp=(node*) malloc(sizeof(node));  
    temp->name=new->name;  
    temp->byear=new->byear;  
    temp->next=new->next;  
  
    if (cursor == NULL)  
        *cursor = temp;  
    else{  
        temp->next = *cursor;  
        *cursor = temp;  
    }  
}
```

Stack push using linked lists

Same as insert at head

- Suppose we want to **push a node onto stack**.
- What needs to be done?
 - New node should point to current head.
 - Current head should be updated to new node.

```
void push(node **cursor, node *new){  
  
    node* temp=(node*) malloc(sizeof(node));  
    temp->name=new->name;  
    temp->byear=new->byear;  
    temp->next=new->next;  
  
    if (cursor == NULL)  
        *cursor = temp;  
    else{  
        temp->next = *cursor;  
        *cursor = temp;  
    }  
}
```

Stack pop using linked lists

Similar to delete at head

- To **pop** a node from stack, we have to delete node from head
 - Save the data of head node
 - Make a copy of the head pointer
 - Shift the head pointer to its next item
 - Call `free` on a copy of the head pointer
 - Return the popped/saved node to caller

```
node * pop(node **headptr) {
    if (*headptr==NULL)
        return NULL;
    else{
        node * new=(node*) malloc(sizeof(node));
        new->name=(*headptr)->name;
        new->byear=(*headptr)->byear;
        new->next = NULL;

        node *old_head = *headptr;
        *headptr = (*headptr)->next;
        free(old_head);

        return new;
    }
}
```

Queue using linked lists

Queue using linked lists

- First item in is the first item out - FIFO

Queue using linked lists

- First item in is the first item out - FIFO
- Two operations for data movement: **enqueue & dequeue**

Queue using linked lists

- First item in is the first item out - FIFO
- Two operations for data movement: **enqueue & dequeue**
 - Dequeued item must be available for use by caller

Queue using linked lists

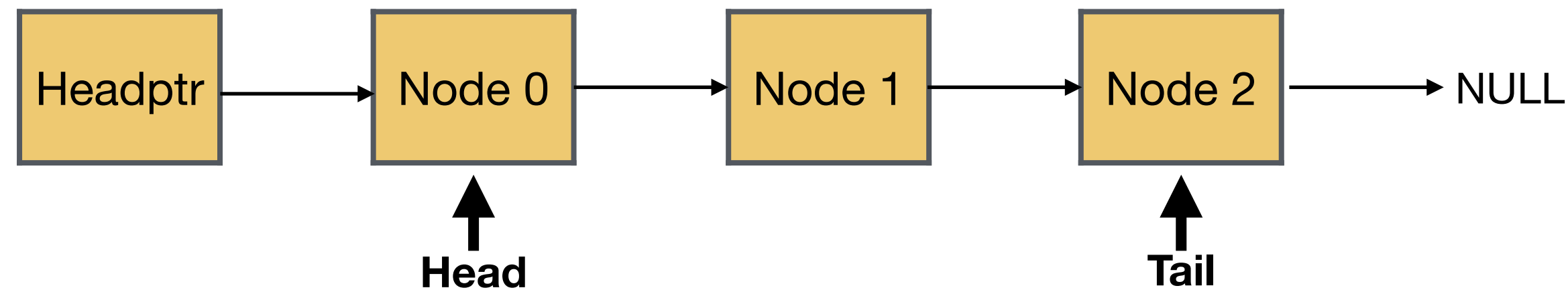
- First item in is the first item out - FIFO
- Two operations for data movement: **enqueue & dequeue**
 - Dequeued item must be available for use by caller

Enqueue

Queue using linked lists

- First item in is the first item out - FIFO
- Two operations for data movement: **enqueue & dequeue**
 - Dequeued item must be available for use by caller

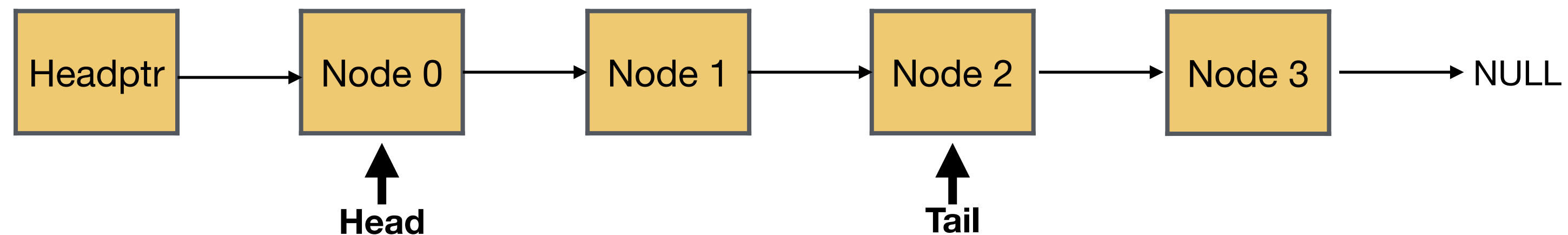
Enqueue



Queue using linked lists

- First item in is the first item out - FIFO
- Two operations for data movement: **enqueue & dequeue**
 - Dequeued item must be available for use by caller

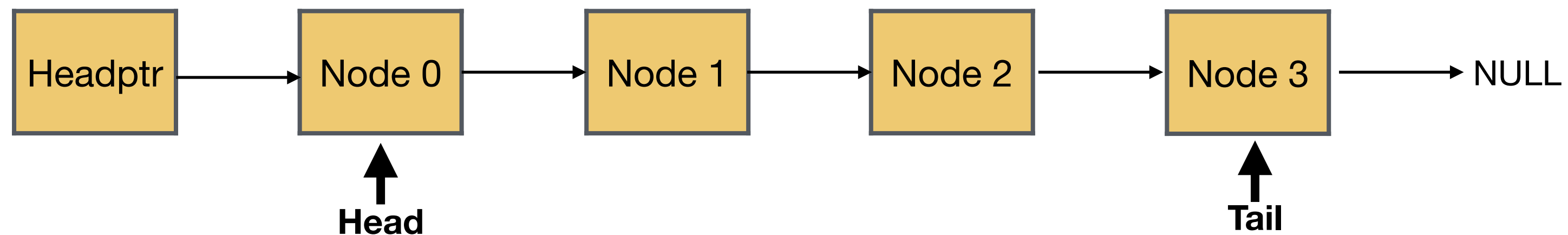
Enqueue



Queue using linked lists

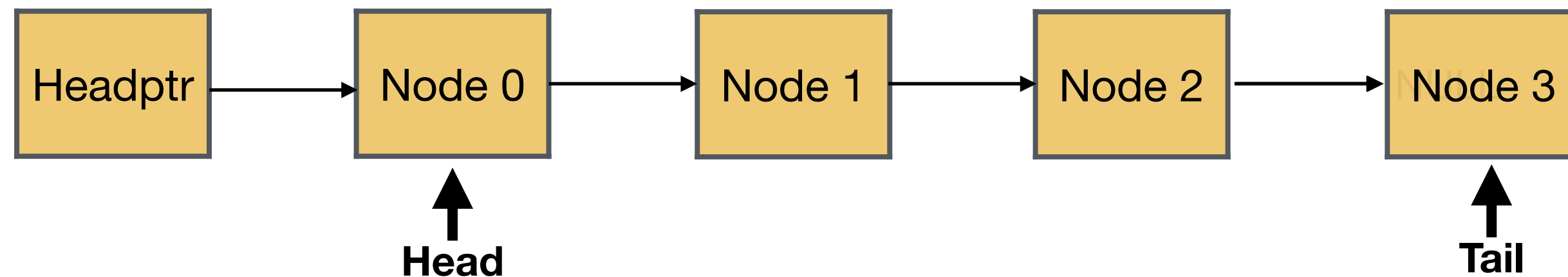
- First item in is the first item out - FIFO
- Two operations for data movement: **enqueue & dequeue**
 - Dequeued item must be available for use by caller

Enqueue



Queue using linked lists

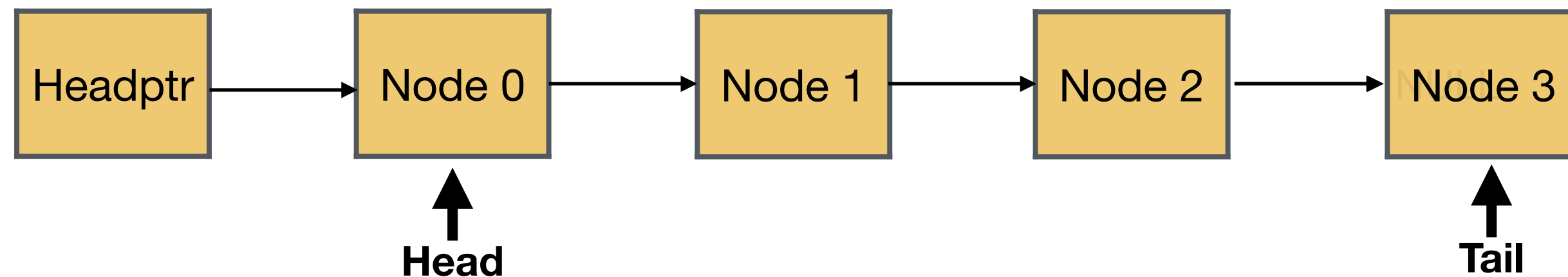
- First item in is the first item out - FIFO
- Two operations for data movement: **enqueue & dequeue**
 - Dequeued item must be available for use by caller



Queue using linked lists

- First item in is the first item out - FIFO
- Two operations for data movement: **enqueue & dequeue**
 - Dequeued item must be available for use by caller

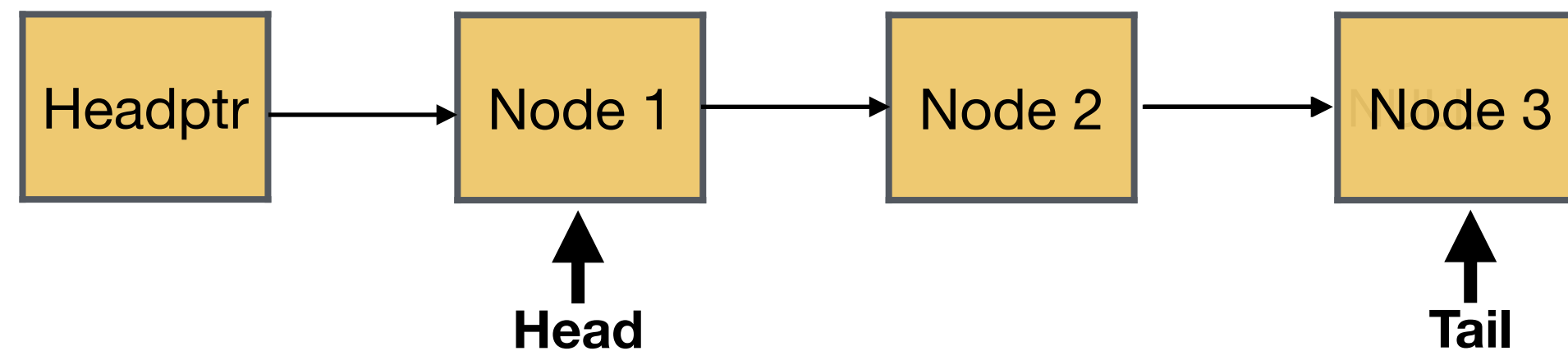
Dequeue



Queue using linked lists

- First item in is the first item out - FIFO
- Two operations for data movement: **enqueue & dequeue**
 - Dequeued item must be available for use by caller

Dequeue



Enqueue using linked lists

Enqueue using linked lists

- To add (enqueue) a **node** to a queue

Enqueue using linked lists

- To add (enqueue) a **node** to a queue
 - We need to first find the tail

Enqueue using linked lists

- To add (enqueue) a **node** to a queue
 - We need to first find the tail

How? The only element in the list whose next is NULL is the tail element.

Enqueue using linked lists

- To add (enqueue) a **node** to a queue
 - We need to first find the tail

How? The only element in the list whose next is NULL is the tail element.

```
void enqueue(node **cursor, node *new){
    if (*cursor == NULL){
        node * temp = (node *)
malloc(sizeof(node));
        temp->name = new->name;
        temp->byear = new->byear;
        temp->next = new->next;
        *cursor = temp;
    }
    else
        enqueue(&(*cursor)->next, new);
}
```

Enqueue using linked lists

- To add (enqueue) a **node** to a queue
 - We need to first find the tail

How? The only element in the list whose next is NULL is the tail element.

```
void enqueue(node **cursor, node *new){
    if (*cursor == NULL){
        node * temp = (node *)
malloc(sizeof(node));
        temp->name = new->name;
        temp->byear = new->byear;
        temp->next = new->next;
        *cursor = temp;
    }
    else
        enqueue(&(*cursor)->next, new);
}
```

Same as insert at tail

Deque using linked lists

Dequeue using linked lists

- To delete (dequeue) a **node** from the queue

Dequeue using linked lists

- To delete (dequeue) a **node** from the queue
 - If head empty do nothing, else,

Dequeue using linked lists

- To delete (dequeue) a **node** from the queue
 - If head empty do nothing, else,
 - Save copy of current head

Dequeue using linked lists

- To delete (dequeue) a **node** from the queue
 - If head empty do nothing, else,
 - Save copy of current head
 - Advance head pointer and free the memory used by old head

Dequeue using linked lists

- To delete (dequeue) a **node** from the queue
 - If head empty do nothing, else,
 - Save copy of current head
 - Advance head pointer and free the memory used by old head
 - Pass/return dequeued item to caller

Dequeue using linked lists

- To delete (dequeue) a **node** from the queue
 - If head empty do nothing, else,
 - Save copy of current head
 - Advance head pointer and free the memory used by old head
 - Pass/return dequeued item to caller

Same as delete at head!

Dequeue using linked lists

- To delete (dequeue) a **node** from the queue
 - If head empty do nothing, else,
 - Save copy of current head
 - Advance head pointer and free the memory used by old head
 - Pass/return dequeued item to caller

```
node * dequeue(node **headptr) {  
    if (*headptr==NULL)  
        return NULL;  
    else{  
        node* new=(node*)  
malloc(sizeof(node));  
new->name=(*headptr)->name;  
new->byear=(*headptr)->byear;  
  
node *old_head = *headptr;  
*headptr = (*headptr)->next;  
free(old_head);  
  
return new;  
    }  
}
```

Same as delete at head!

Exercise(s)

- Given a *sorted* linked list, implement binary search on the list

```
node * binary_search(*headptr, char * key)
```

- Return a NULL pointer if the element is not found
 - Otherwise return a pointer to the element.
- Hint: Write a function to get the middle element in a linked list

How do you find the middle element in a linked list?

Finding middle of a linked list

```
#include <stdio.h>

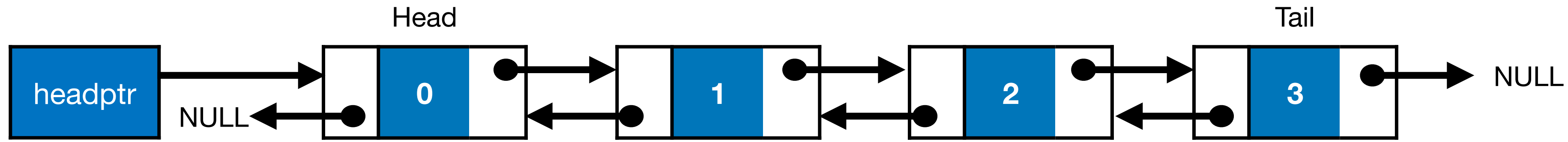
int main(void){
    int i, target, j;
    printf("Enter a target number:\t");
    scanf("%d", &target);
    for (j=0, i=0; j<target; i++, j++)
        j++;
    printf("Midway to target is %d", i);
}
```

Exercise(s)

- Given two ***sorted*** linked lists write a function that takes the two head pointers and returns a pointer to a ***merged*** list
- Sort order **must be maintained**. Basic idea ...
 - Traverse both lists until one of them ends, then copy over the remaining list
 - During traversal add new nodes in sorted order

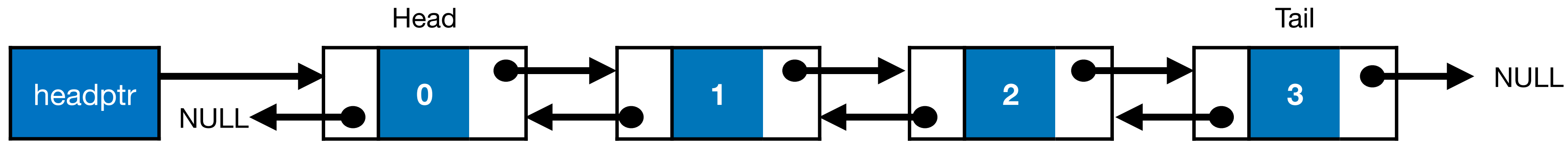
See (click): [Full Code on Gitlab!](#)

Doubly linked list



*A **doubly linked** list maintains a pointer to both the previous as well as the next element.*

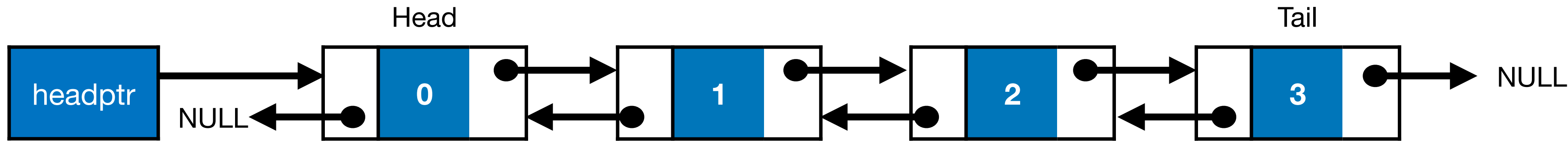
Doubly linked list



*A **doubly linked** list maintains a pointer to both the previous as well as the next element.*

- Advantages:

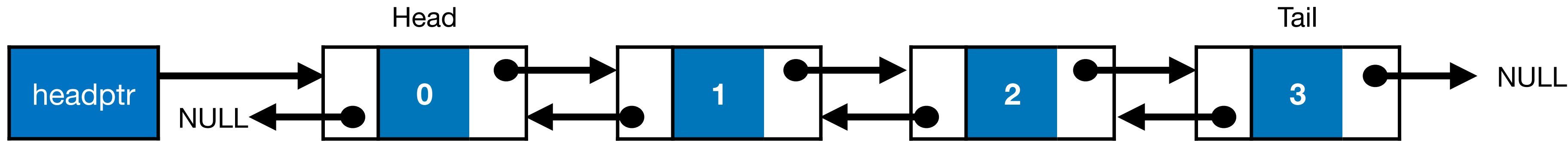
Doubly linked list



*A **doubly linked** list maintains a pointer to both the previous as well as the next element.*

- Advantages:
 - Allows backward and forward traversal

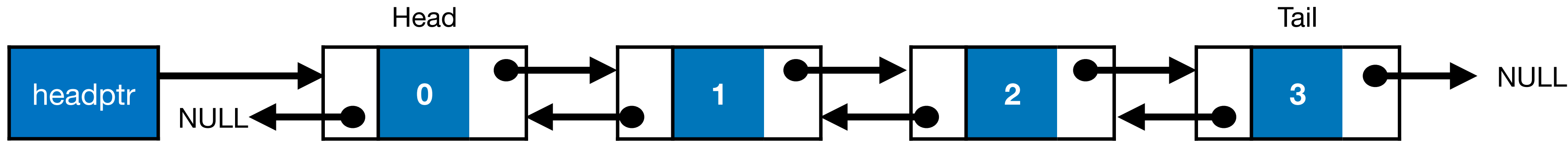
Doubly linked list



*A **doubly linked** list maintains a pointer to both the previous as well as the next element.*

- Advantages:
 - Allows backward and forward traversal
 - Easier to delete a node - why?

Doubly linked list



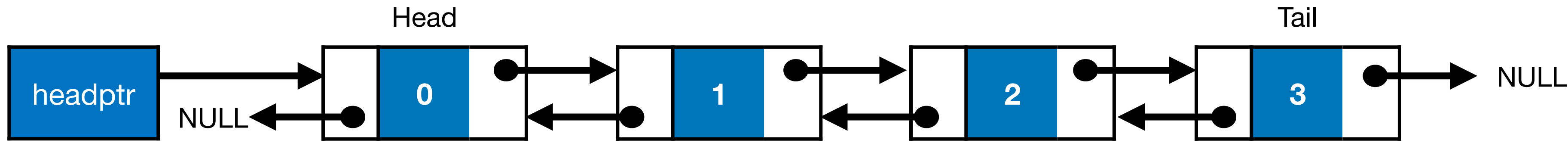
*A **doubly linked** list maintains a pointer to both the previous as well as the next element.*

- Advantages:

- Allows backward and forward traversal
- Easier to delete a node - why?

- Disadvantages

Doubly linked list



*A **doubly linked** list maintains a pointer to both the previous as well as the next element.*

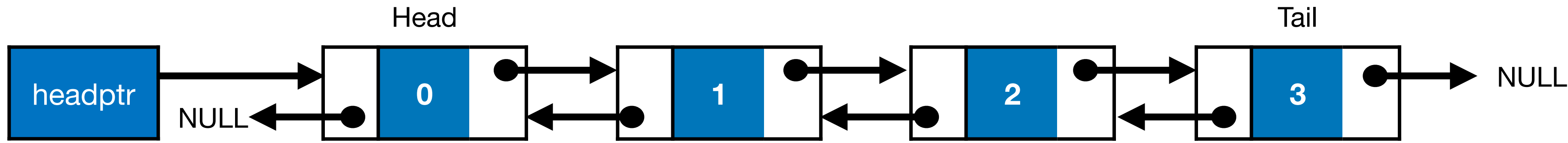
- Advantages:

- Allows backward and forward traversal
- Easier to delete a node - why?

- Disadvantages

- Takes up more memory.

Doubly linked list



A **doubly linked** list maintains a pointer to both the previous as well as the next element.

- Advantages:

- Allows backward and forward traversal
- Easier to delete a node - why?

- Disadvantages

- Takes up more memory.
- Increased bookkeeping, therefore performance overhead

Doubly linked lists

- First there will be a change to the struct definition
- Need to modify insertion/deletion functions so that prev and next are maintained.
 - Insert at head
 - Insert at tail
 - ...

```
typedef struct person{  
    char *name;  
    unsigned int byear;  
    struct person *next;  
    struct person *prev;  
}node;
```

To be continued next time ...