ECE220

Lecture x0012 - 04/01

Linked lists - stacks & queues

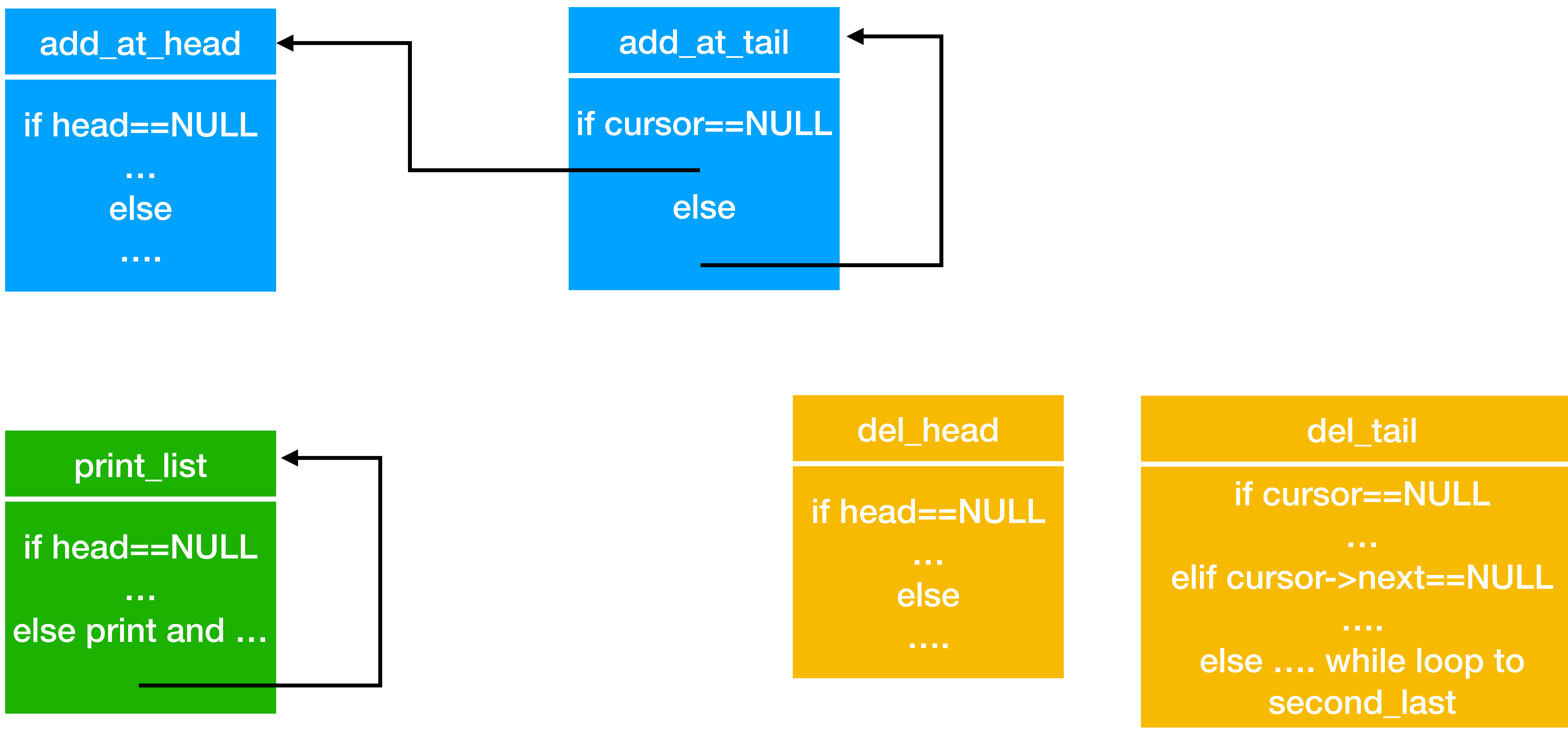
Announcements

- **Quiz #3 ongoing**:
    - 04/02 is last day
- **Exam on 4/10:**
    - Study material has been posted
    - Lectures 7 through 15 inclusive
    - Conflict exam sign up live (**deadline is 04/06**)

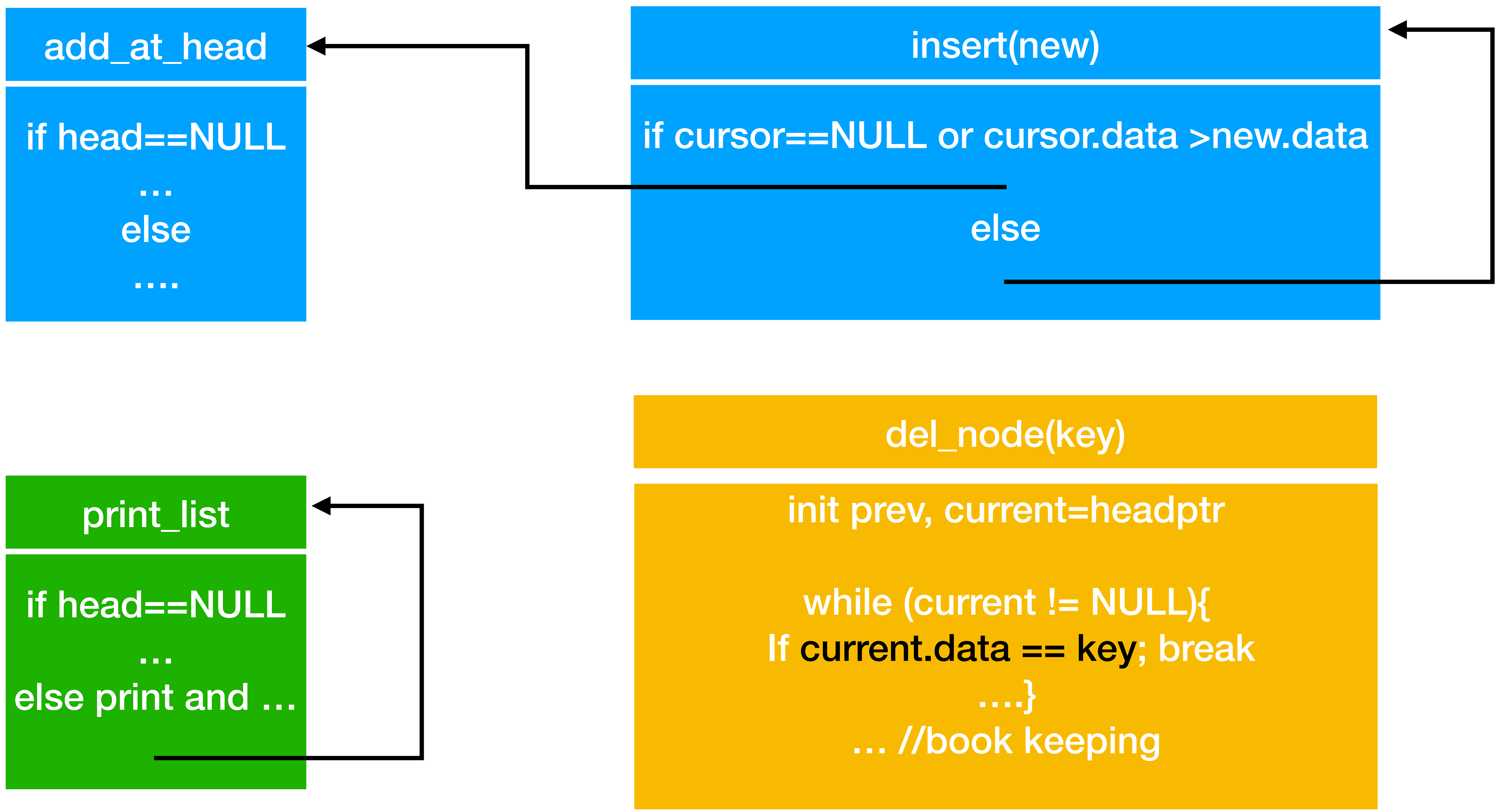
Recap

- 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

Review - singly linked lists (plain)

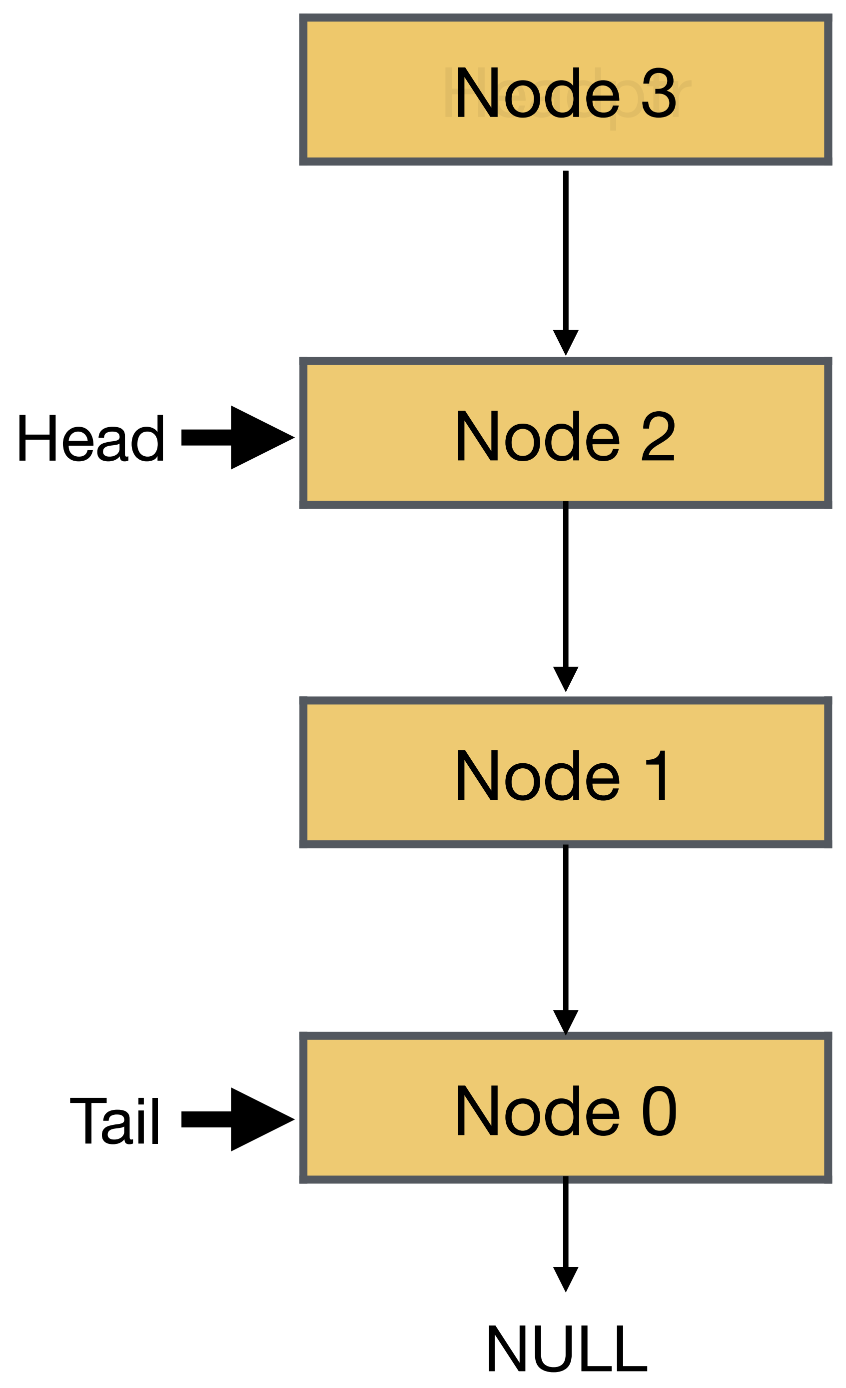


Review - singly linked lists (sorted)



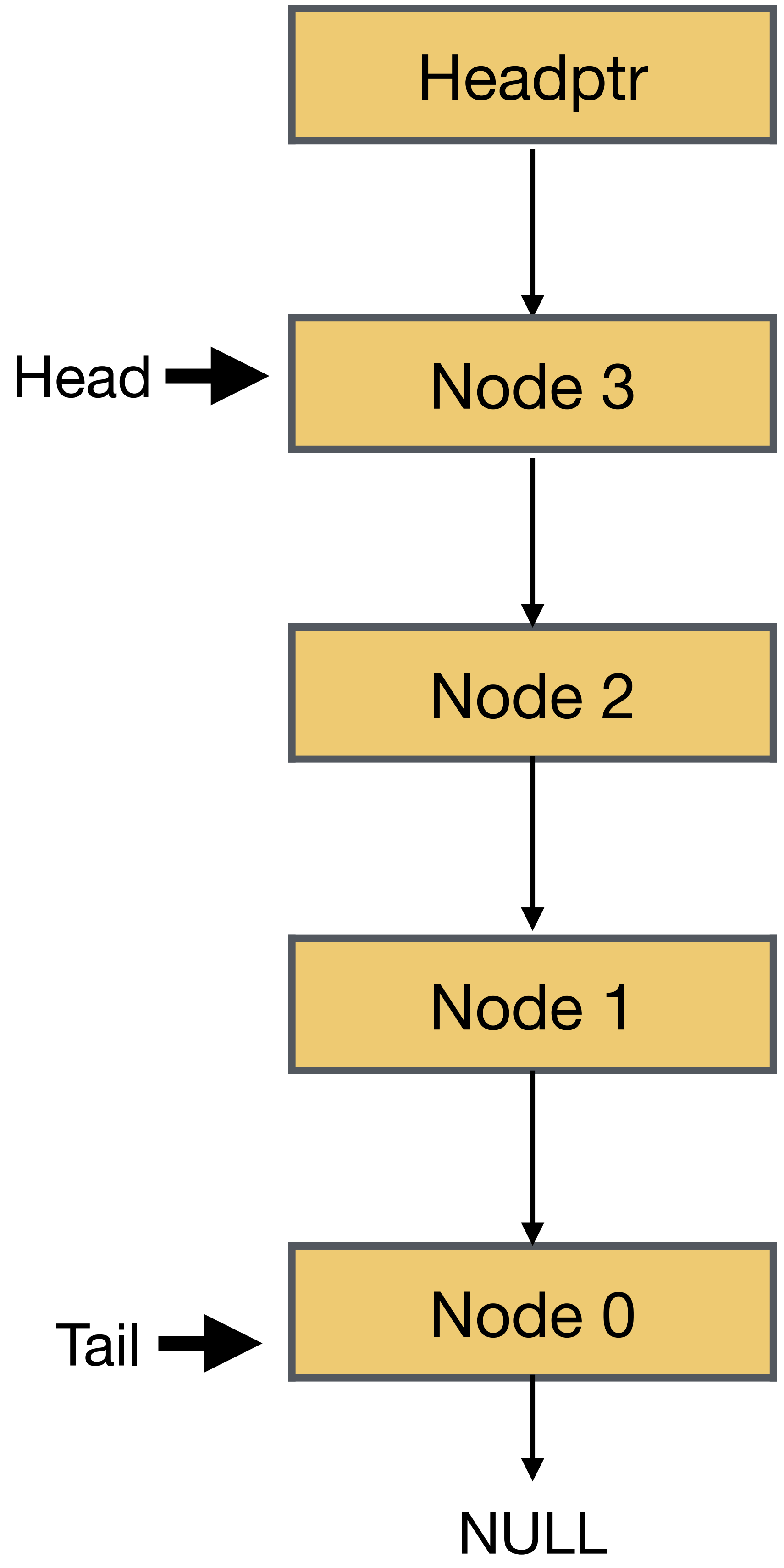
Stack using linked lists

- 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

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

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;






  }
}
```

Stack pop using linked lists

Similar to delete at head

- To **pop** a node from stack, we have to delete node from head:
  - Deal with empty list
  - Make a *copies* of the head pointer
  - Shift the head pointer to its next item
  - Call `free` on a copy of the head pointer
  - ***Return the popped copy to caller***

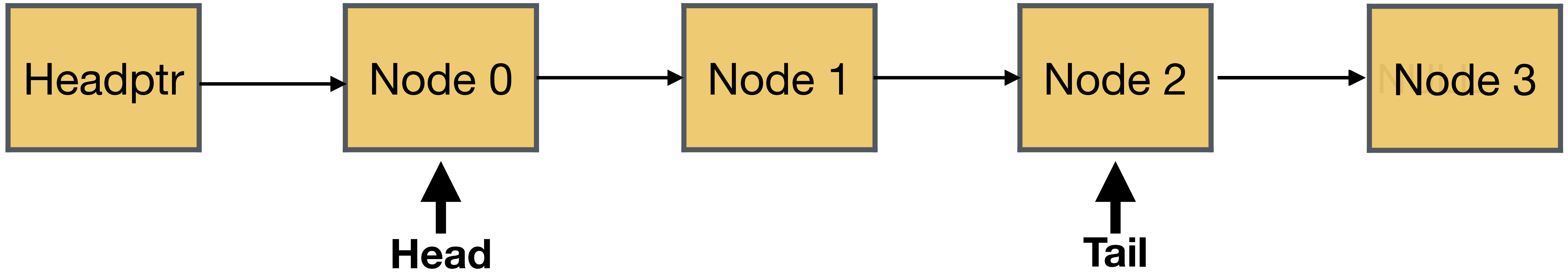
```
node * pop(node **headptr){

}
```

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

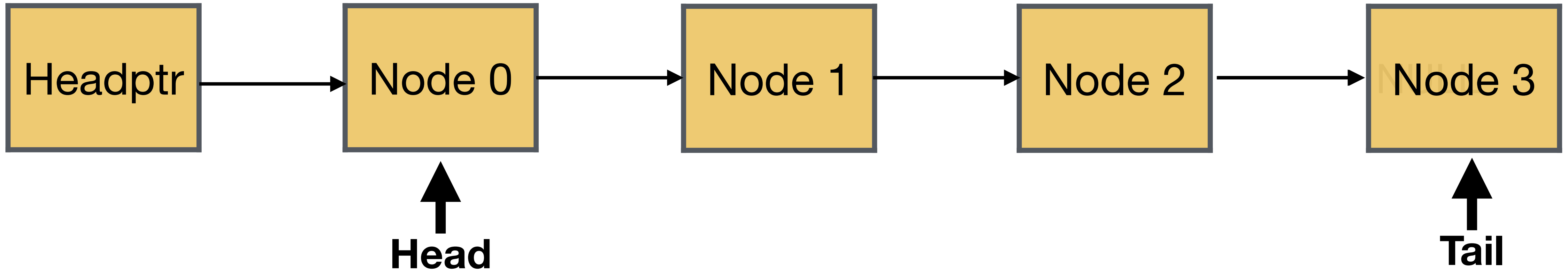
Enqueue



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

Dequeue



Enqueue using linked lists

- To add (enqueue) a **node** to a queue
  - If queue empty, add right away, else
  - Go till the end

```
void enqueue(node **cursor, node *new){



}
```

Same as insert at tail

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){

}
```

Same as delete at head!

Relations

```c
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;
  }
}
```

Linked List	Stack	Queue
Add at head	Push	
Delete at head	Pop	Dequeue
Add at tail		Enqueue

```c
    *headptr = (*headptr)->next;
    *cursor = temp;
} else{
    temp->next = *cursor;
    *cursor = temp;
  }
}
```

```c
node * pop(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;
  }
}
```

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

Relations

Linked List	Stack	Queue
Add at head	Push	
Delete at head	Pop	Dequeue
Add at tail		Enqueue

```c
void add_at_tail(node **cursor, node *new){
  if (*cursor == NULL)
    add_at_head(cursor, new);
  else
    add_at_tail(&(*cursor)->next, new);
}
```

```c
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);
}
```

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.

- <u>Hint:</u> 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

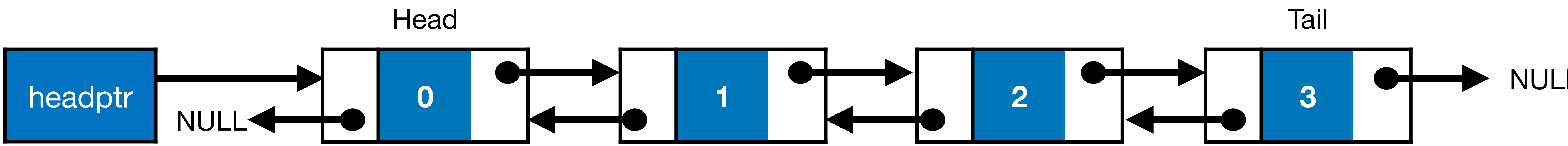
```
#include <stdio.h>

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 for “later”

- 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

Doubly linked list



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

- <u>Advantages:</u>
  - Allows backward and forward traversal
  - Easier to delete a node - why?
- <u>Disadvantages</u>
  - 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;
```

More on this when we talk about Trees!