ECE220

Lecture x0012
Linked lists - stacks & queues

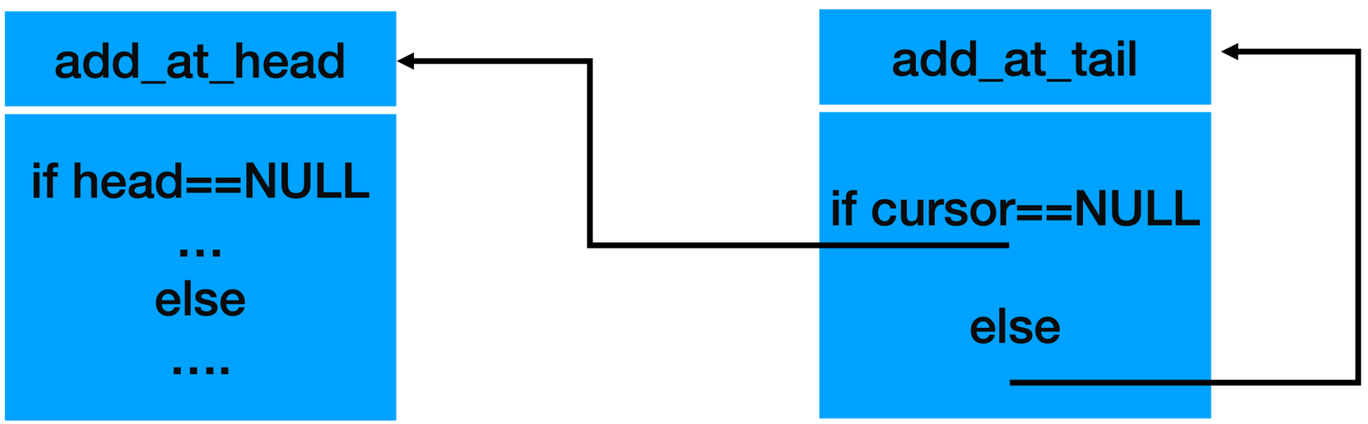
Announcements

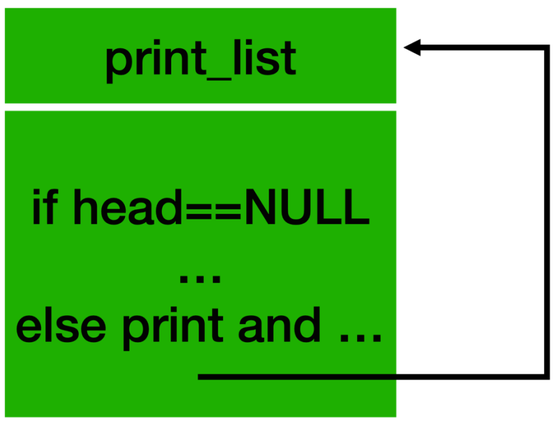
- **Exam on 04/02:**
  - Study material has been posted
    - HKN Material is available on their website.
- Lectures 9 through 16 inclusive
- No lecture on Thursday.
- **Check room allocations.**

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)





del_head

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

del_tail

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

Today - Lesson Objectives

- Understand how recursion works in the case of linked lists.
- Understand stacks and queues as special cases of generic linked lists.
- Be able to implement stacks and queues using linked lists.
- Be able to implement binary search on a sorted linked list.

Memory addresses are for illustration — it may not match LC3 CC or RTS.

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

Review: Adding at tail set-up

```
int main(void){
  int nums[] = {5, 2, 1};
  int total = 3;
  node temp;
  node *headptr = NULL;

  for (int i = 0; i < total; i++) {
    temp.data = nums[i];
    temp.next = NULL;
    add_at_tail(&headptr, &temp);
  }
```

	x3000	NULL	headptr
	x3019	5	temp.data
	x3020	NULL	temp.next
nums	5	2	1

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

Memory addresses are for illustration — it may not match LC3 CC or RTS.

Review: Adding at tail position

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

	x3000	NULL	headptr
	x3019	5	temp.data
	x3020	NULL	temp.next
nums	5	2	1

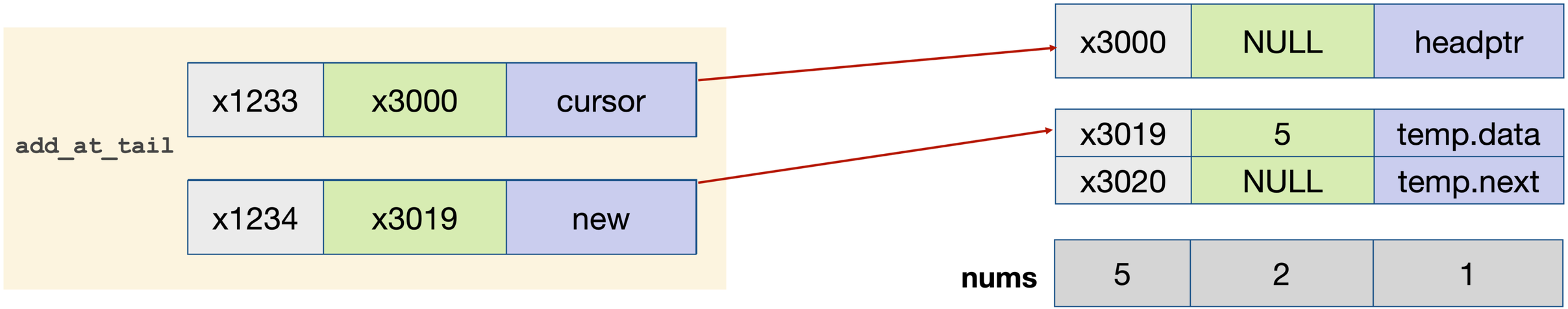
Memory addresses are for illustration — it may not match LC3 CC or RTS.

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

Review: add_at_tail

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

```
Function call : add_at_tail(&headptr, &temp);
```



Memory addresses are for illustration — it may not match LC3 CC or RTS.

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

add_at_tail: Walk-through

x2017	5	temp.data
x2018	NULL	temp.next

add_at_head

x2019	x3000	cursor
x2020	x3019	new

add_at_tail

x1233	x3000	cursor
x1234	x3019	new

```
void add_at_head(node **cursor, node *new) {
  node *temp = malloc(sizeof(node));
  temp->data = new->data;
  temp->next = new->next;

  if (*cursor == NULL) {
    *cursor = temp;
  } else {
    temp->next = *cursor;
    *cursor = temp;
  }
}
```

Function call : **add at head(cursor, new);**

x3000	**x2017**	headptr

x3019	5	~~temp~~.data
x3020	NULL	~~temp~~.next

nums	5	2	1

Memory addresses are for illustration — it may not match LC3 CC or RTS.

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

add_at_tail: Recursive call

x2017	5	~~temp~~.data
x2018	NULL	~~temp~~.next

```
int main(void){
  int nums[] = {5, 2, 1};
  int total = 3;
  node temp;
  node *headptr = NULL;

  for (int i = 0; i < total; i++) {
    temp.data = nums[i];
    temp.next = NULL;
    add_at_tail(&headptr, &temp);
  }
```

x3000	x2017	headptr

x3019	2	temp.data
x3020	NULL	temp.next

nums	5	2	1

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

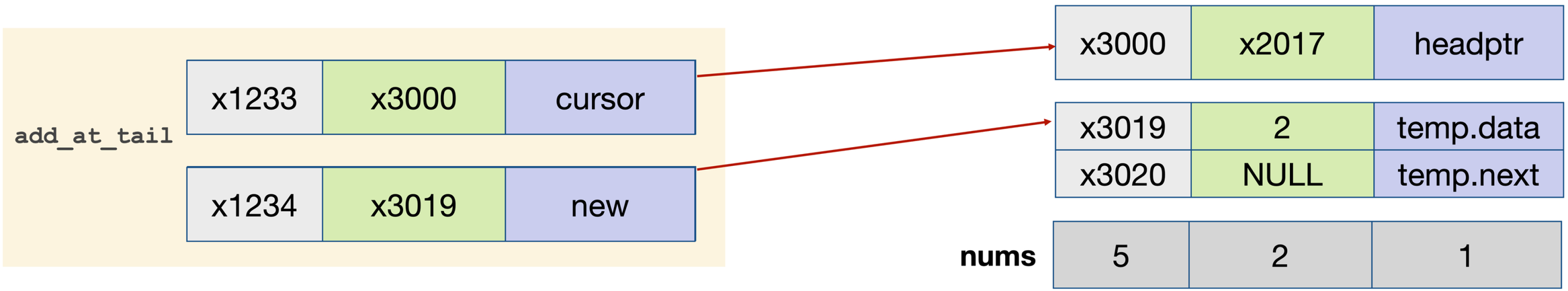
Memory addresses are for illustration — it may not match LC3 CC or RTS.

Recursive call: add_at_tail

x2017	5	~~temp~~.data
x2018	NULL	~~temp~~.next

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

```
Function call : add_at_tail(&headptr, &temp);
```



Memory addresses are for illustration — it may not match LC3 CC or RTS.

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

add_at_tail: Second recursion

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

```
Function call : add_at_tail(&(*cursor)->next,
                            &temp);
```

x2017	5	~~temp~~.data
x2018	NULL	~~temp~~.next

add_at_tail

x1133	x2018	cursor
x1134	x3019	new

x3000	x2017	headptr

x3019	2	temp.data
x3020	NULL	temp.next

nums

5	2	1

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

Memory addresses are for illustration — it may not match LC3 CC or RTS.

Review: add_at_head

x4017	2	temp.data
x4018	NULL	temp.next

x2017	5	~~temp~~.data
x2018	x4017	~~temp~~.next

add_at_head

xFF10	x2018	cursor
xFF11	x3019	new

add_at_tail

x1133	x2018	cursor
x1134	x3019	new

```
void add_at_head(node **cursor, node *new) {
  node *temp = malloc(sizeof(node));
  temp->data = new->data;
  temp->next = new->next;

  if (*cursor == NULL) {
    *cursor = temp;
  } else {
    temp->next = *cursor;
    *cursor = temp;
  }
}
```

Function call : **add_at_head(cursor, new);**

x3000	x2017	headptr

x3019	2	~~temp~~.data
x3020	NULL	~~temp~~.next

nums	5	2	1

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

Memory addresses are for illustration — it may not match LC3 CC or RTS.

add_at_head: Walkthrough

x4017	2	temp.data
x4018	NULL	temp.next

x2017	5	~~temp~~.data
x2018	x4017	~~temp~~.next

add_at_head

xFF10		cursor
xFF11		new

add_at_tail

x1133	x2018	cursor
x1134	x3019	new

```
void add_at_head(node **cursor, node *new) {
  node *temp = malloc(sizeof(node));
  temp->data = new->data;
  temp->next = new->next;

  if (*cursor == NULL) {
    *cursor = temp;
  } else {
    temp->next = *cursor;
    *cursor = temp;
  }
}
```

x3000	x2017	headptr

x3019	2	temp.data
x3020	NULL	temp.next

nums	5	2	1

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

Memory addresses are for illustration — it may not match LC3 CC or RTS.

Conclusion: add_at_tail

x4017	2	temp.data
x4018	**NULL**	temp.next

x2017	5	~~temp~~.data
x2018	**x4017**	~~temp~~.next

End Result

x2017	x2018
Data	**5**
Next	x4017

x4017	x4018
Data	**2**
Next	NULL

We showed:

- ***One way*** to add items at the tail position:
  - In the process, we also examined how an element gets added at the head.
- Practice practice practice!

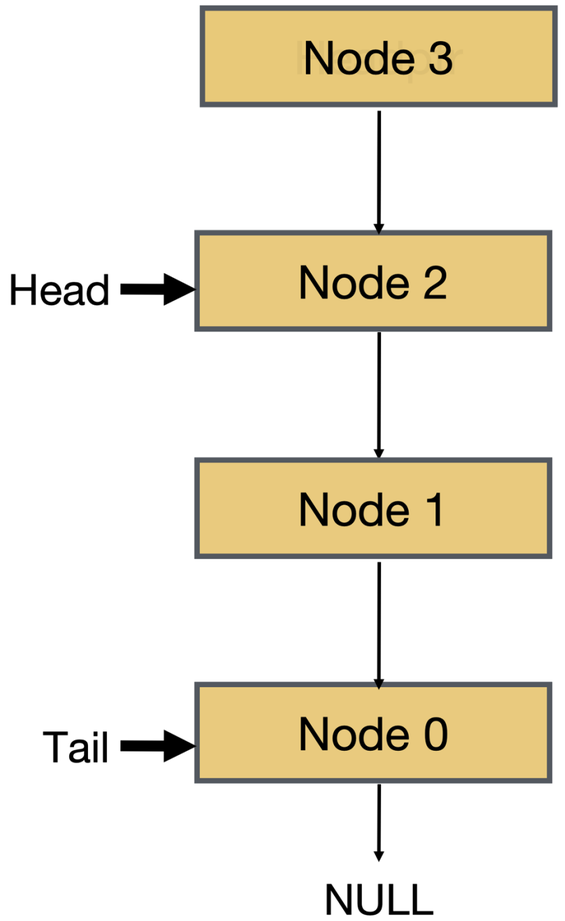
x3000	x2017	headptr

x3019	2	temp.data
x3020	NULL	temp.next

nums	5	2	1

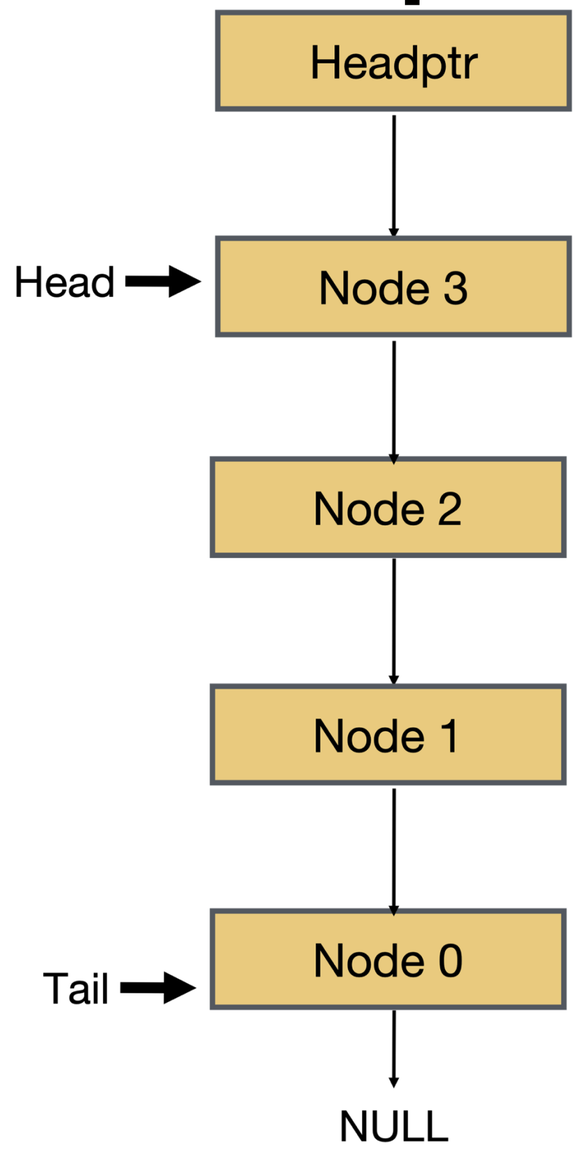
Stack using linked lists: Push

- 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: Pop

- 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;

  if (cursor == NULL)
      *cursor = temp;
  else{
    temp->next = *cursor;

  }
}
```

Stack pop using linked lists

- 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***

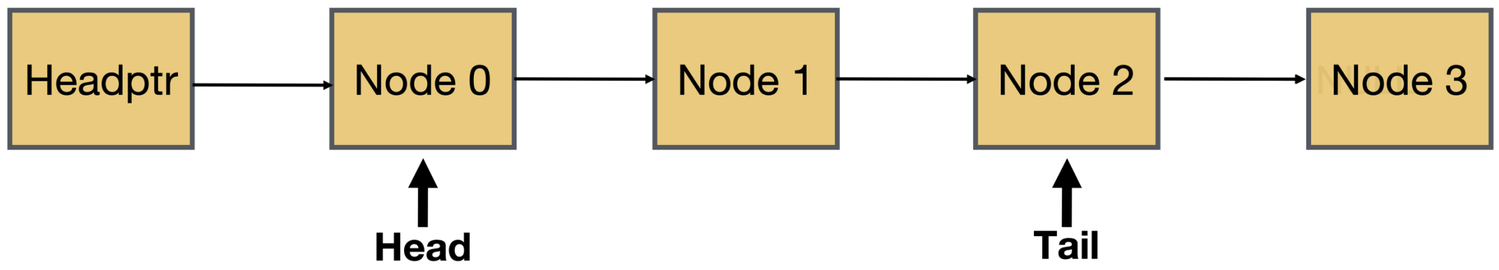
Similar to delete at head

```
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 - Enqueue

- 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

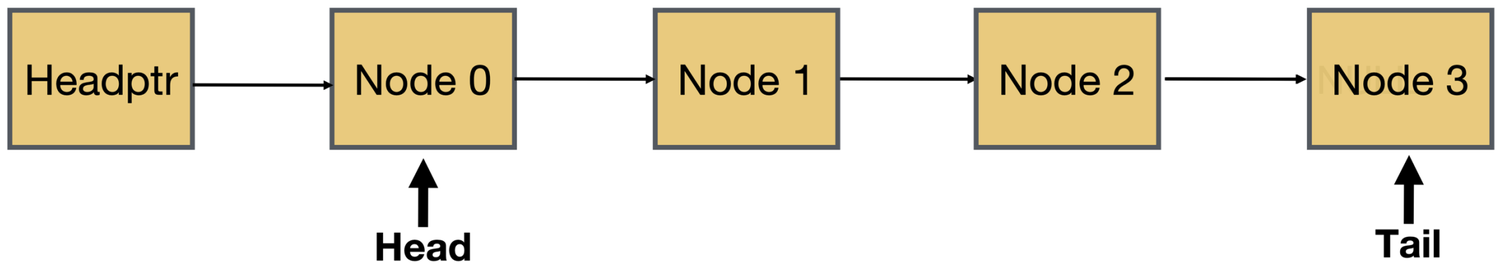
Enqueue



Queue using linked lists - Dequeue

- 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

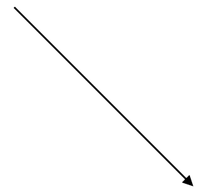
Dequeue



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){
  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

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;
  }
}
```

***Similar to* 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

```c
#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