

A collection of historical artifacts is arranged on a light-colored, textured surface. In the top left, a portion of a wooden chessboard with a checkered pattern and several chess pieces is visible. Below the chessboard, there are two medals: one with a red ribbon and a star-shaped face, and another with a blue ribbon and a star-shaped face. A small, ornate compass is located in the bottom left corner. A quill pen with a red-tipped nib is positioned diagonally across the center. The text 'Linked Lists' is printed in a large, serif font on the right side of the image.

Linked Lists

**Notes from Dr. Charlie Obimbo
and revised by Yan Yan.**



Assignment 1

1. A1 file updated

1. The revised bubble sort had an error, and it is now fixed.

2. What are counted as operations?

1. We view *ith* line of pseudocode (or actual code implementations) as constant time, and count it as 1 in the calculations.



Announcement

1. Slides in Week 1 and 2 updated
2. Volunteer note taker needed
 1. Contact SAS +1-519-824-4120 Ext. 56208
sas@uoguelph.ca
3. Zybook



Zybook

1. Sign in or create an account at learn.zybooks.com
2. Enter zyBook code **UOGUELPHCIS2520DSEYanFall2024**
3. Subscribe
 - ◆ A subscription is **\$64 USD**.
 - ◆ Students may begin subscribing on Aug 19, 2024 and the cutoff to subscribe is Dec 15, 2024. Subscriptions will last until Jan 14, 2025.

Note: this is NOT a mandatory textbook for this course



Contents

1. Lists and implementation
2. Singly-linked list
3. Doubly-linked list (next set of slides)



Learning Objectives

1. Describe the difference between lists and arrays
2. Name one of the applications of list
3. Describe the typical list operations informally
4. Implement typical list operations in C
5. Distinguish singly- and doubly- linked list and describe their advantages and disadvantages



List abstract data type (ADT)

- ◆ List -- common ADT for holding ordered data
- ◆ Examples:
 - a list of student records, a list of webpages, a list of song
- ◆ Common operations
 - Append
 - Insert
 - Modify
 - Remove a data item
 - Search
 - Print, etc



List Application – Tabbed Browsing

- ◆ Each tab stores a URL to be displayed (among other data).
- ◆ Need arbitrary number of tabs.
- ◆ User can delete any tab at any time.
- ◆ User can create a new tab at any time.
- ◆ When the browser window is closed, the list is destroyed (its memory recycled).



List Applications

- ◆ Cache in your browser that allows you to back button (linked list of URLs).
- ◆ An image viewer software uses a linked list to view the previous and the next images using the previous and next buttons.
- ◆ Maintain directory of names
 - For instance, consider the 'like' function on Facebook. When a user 'likes' a post, their user ID is added to a 'likes' list associated with the particular post.



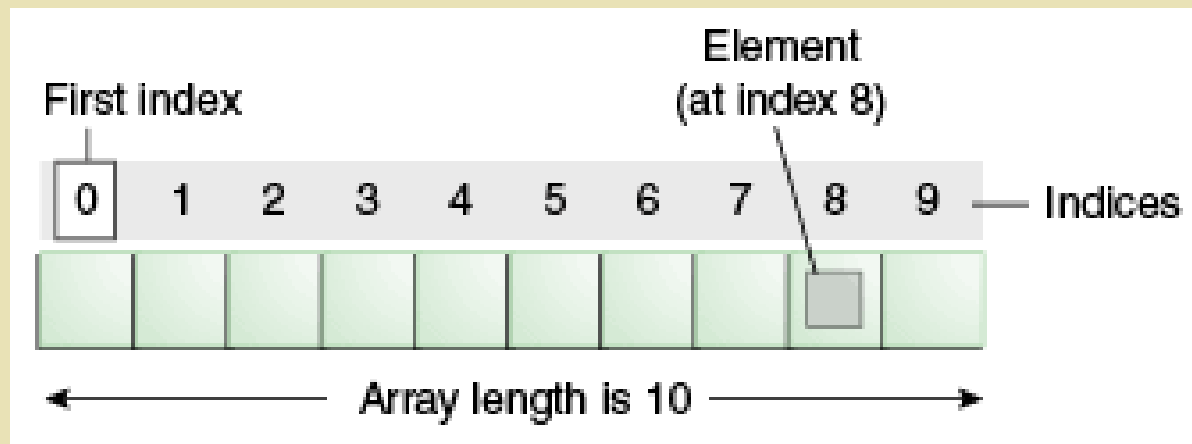
List Implementations

- ◆ We will look at the most common operations that are usually used.
 - The most obvious implementation is to use either an array or linked list

List Implementations

◆ Array:

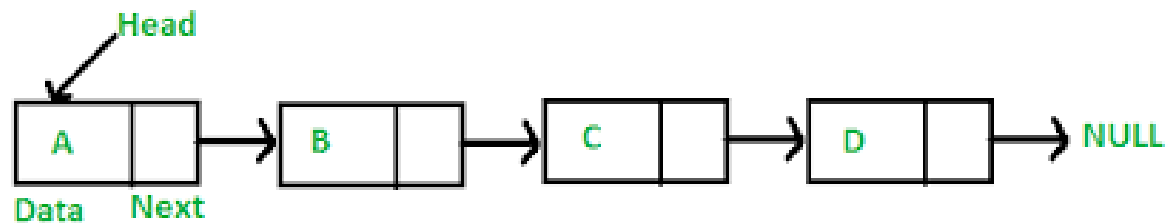
- a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created.



List Implementations

◆ Linked Lists

- data structure which can change during execution. Successive elements are connected by pointers. Last element points to NULL.

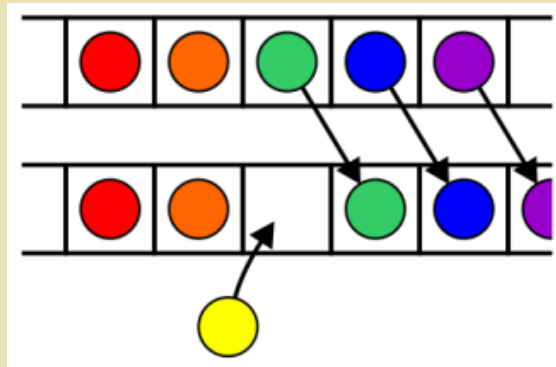
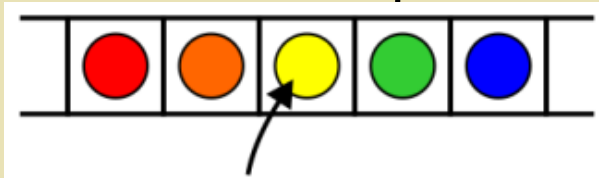


Operations

3.1.1

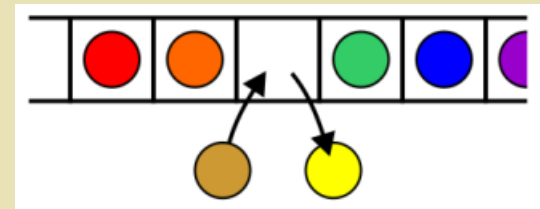
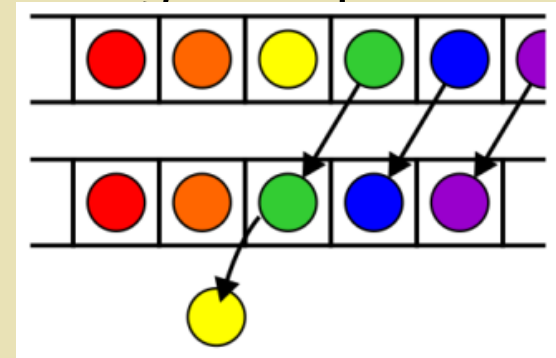
Operations at the k^{th} entry of the list include:

Access to the object



Insertion of a new object

Erasing an object

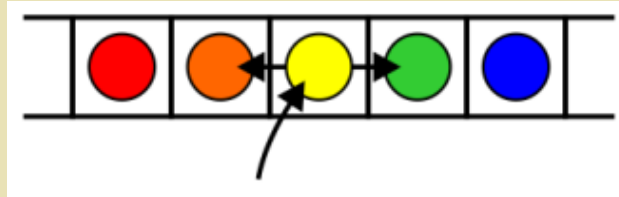


Replacement of the object

Operations

3.1.1

Given access to the k^{th} object, gain access to either the previous or next object



Given two abstract lists, we may want to

- Concatenate the two lists
- Determine if one is a sub-list of the other

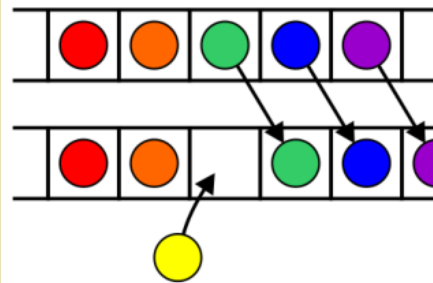


List abstract data type (ADT)

- ◆ Why not just use arrays for sequential data?
 - Arrays have a fixed size; we may have unlimited amount of data.
 - Some operations are computational expensive.

Why Linked List?

- ◆ Arrays can be used to store linear data of similar types, but arrays have the following limitations.
 1. The **size of the arrays is fixed**: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
 2. **Inserting** a new element in an array of elements is expensive because the **room has to be created** for the new elements and to create room existing elements have to be shifted but in Linked list if we have the head node then we can traverse to any node through it and insert new node at the required position.



Why Linked List?

- ◆ For example, in a system, if we maintain a sorted list of IDs in an array `id[]`.

`id[] = [1000, 1010, 1050, 2000, 2040]`.

- ◆ To **insert** a new ID 1005, and maintain the sorted order, we have to move all the elements after 1000.

0	1	2	3	4
<code>id[] = [1000, 1010, 1050, 2000, 2040]</code> .				

0	1	2	3	4	5
<code>id[] = [1000, , 1010, 1050, 2000, 2040]</code> .					

0	1	2	3	4	5
<code>id[] = [1000, 1005, 1010, 1050, 2000, 2040]</code> .					

How many operations
needed for this insertion?

Why Linked List?

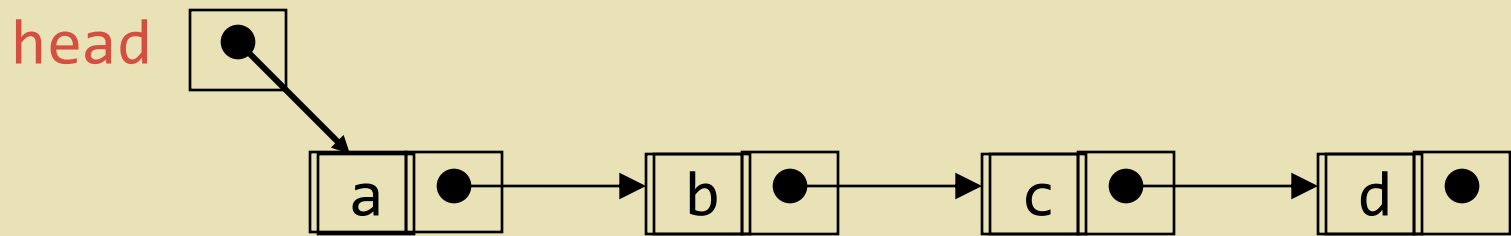
- ◆ **Deletion** can also be expensive with arrays. For example, to delete 1010 in `id[]`, everything after 1010 has to be moved due to this so much work is being done which affects the efficiency of the code.

0	1	2	3	4		
id[]	=	[1000,	1010,	1050,	2000,	2040].

0	1	2	3	4		
id[]	=	[1000,	1050,	2000,	2040,	☹].

Anatomy of a linked list

- ◆ A linked list consists of:
 - A sequence of **nodes**



Each node contains a **value**
and a **link** (pointer or reference) to some other node
The last node contains a **null link**



More terminology

- ◆ A node's **successor** is the next node in the sequence
 - The last node has no successor
- ◆ A node's **predecessor** is the previous node in the sequence
 - The first node has no predecessor
- ◆ A list's **length** is the number of elements in it
 - A list may be **empty** (contain no elements)



Advantages & Disadvantages

◆ Advantages over arrays

- Dynamic size
- Ease of insertion/deletion

◆ Drawbacks:

- **Random access is not allowed.** We have to access elements sequentially starting from the first node (head node). So we cannot do binary search with linked lists efficiently with its default implementation.
- **Extra memory** space **for** a **pointer** is required with each element of the list.
- **Not cache friendly.** Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.



Pointers and references


- ◆ C – “pointers”
 - C allows you to modify pointers in arbitrary ways, and to point to anything
 - Available operations are:
 - Dereferencing
 - copying
 - comparing for equality
 - There are constraints on what kind of thing is referenced: for example, a reference to an **array of int** can *only* refer to an **array of int**

Implementation in C

// A linked list node

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





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

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

```
    struct Node {  
        int data;  
        struct Node *next;
```

```
};
```

```
// Linked list with 3 nodes
```

```
int main(){
```

```
    struct Node *head = NULL;
```

```
    struct Node *second = NULL;
```

```
    struct Node *third = NULL;
```

```
// allocate 3 nodes in the heap
```

```
    head = (struct Node *)malloc(sizeof(struct Node));
```

```
    second = (struct Node *)malloc(sizeof(struct Node));
```

```
    third = (struct Node *)malloc(sizeof(struct Node));
```




```
/* Three blocks have been allocated dynamically.
```

We have pointers to these three blocks as head, second and third

head	second	third
+---+---+	+---+---+	+---+---+
# #	# #	# #
+---+---+	+---+---+	+---+---+

represents any random value.


Data is random because we haven't assigned anything yet */

```
head->data = 1;           // assign data in first node
head->next = second;      // Link first node with
second->data = 2;         // assign data to second node
second->next = third;     // Link second node with the third node
```

```
/* head          second          third
   |             |             |
+---+---+      +---+---+      +---+---+
| 1 | o----->| 2 | o----->| # | # |
+---+---+      +---+---+      +---+---+ */
```

```
third->data = 3; // assign data to third node
third->next = NULL;
return 0;
```

```
}
```



```
// A simple C program for traversal of a linked list
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node *next;
};

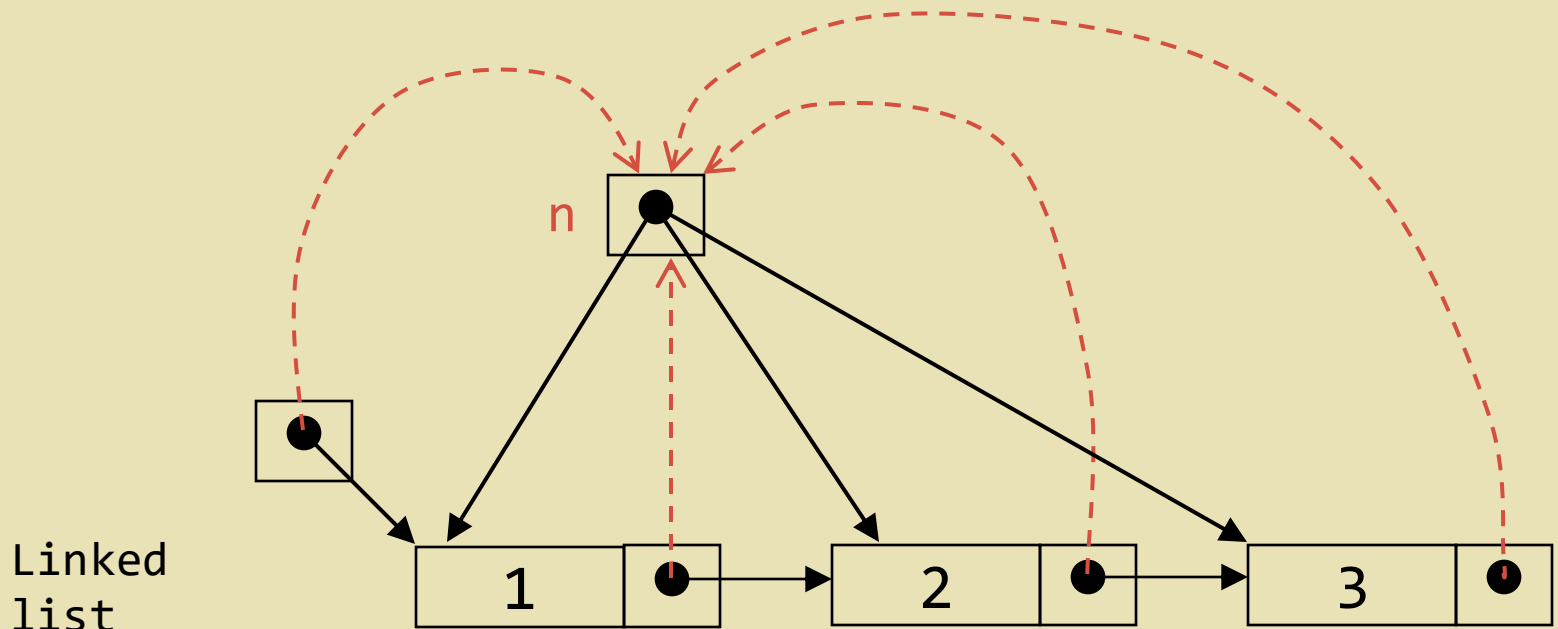
int main() {
    struct Node *head = NULL;
    struct Node *second = NULL;
    struct Node *third = NULL;

    // allocate 3 nodes in the heap
    head = (struct Node *)malloc(sizeof(struct Node));
    second = (struct Node *)malloc(sizeof(struct Node));
    third = (struct Node *)malloc(sizeof(struct Node));
    head->data = 1; // assign data in first node
    head->next = second; // Link first node with second
    second->data = 2; // assign data to second node
    second->next = third;
    third->data = 3; // assign data to third node
    third->next = NULL;

    return 0;
}
```

Traversing (animation)

```
// Prints contents of linked list
// starting from the given node
void printList(struct Node *n){
    while (n != NULL) {
        printf(" %d ", n->data);
        n = n->next;
    }
}
```





Traversal

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

struct Node {
    int data;
    struct Node *next;
};

// Prints contents of linked list starting from the given node
void printList(struct Node *n){
    while (n != NULL) {
        printf(" %d ", n->data);
        n = n->next;
    }
}

int main(){
    struct Node *head = NULL;
    struct Node *second = NULL;
    struct Node *third = NULL;
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));
    head->data = 1; // assign data in first node
    head->next = second; // Link first node with second
    second->data = 2; // assign data to second node
    second->next = third;
    third->data = 3; // assign data to second node
    third->next = NULL;
    printList(head);
    return 0;
}
```




Singly-Linked Lists (SLL)

- ◆ A **singly-linked list** is a data structure for implementing a list ADT
 - Each node has data and a pointer to the next node.
 - The list structure typically has pointers to the list's first node and last node.
 - SLL first node is called the **head**, and the last node the **tail**.
 - A list where elements contain pointers to the next and/or previous elements in the list

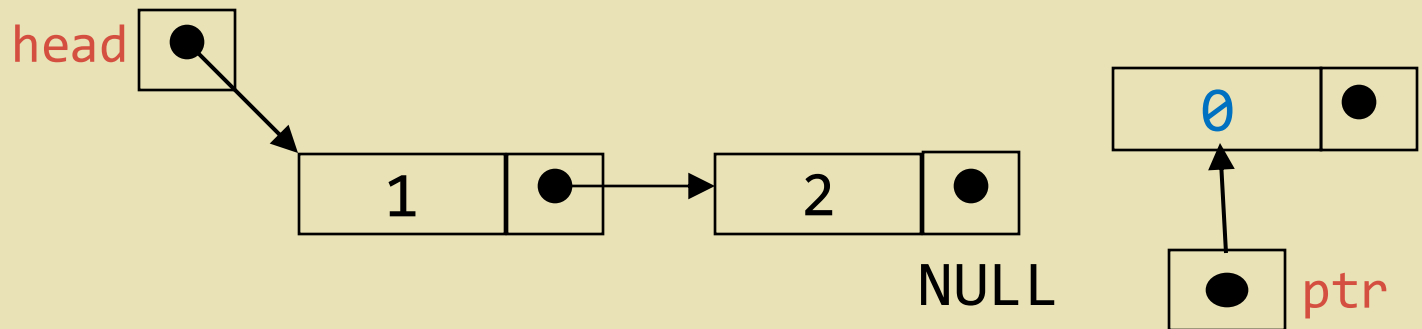


Inserting a node into a SLL

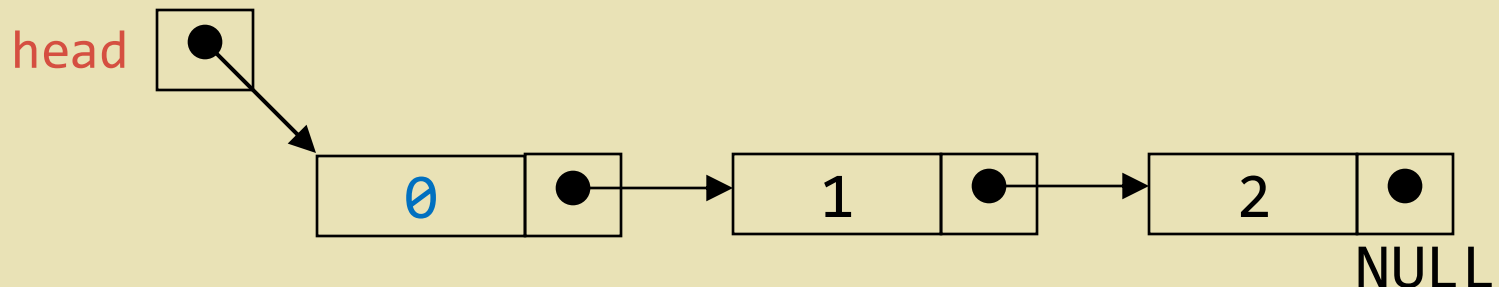
- ◆ There are many ways you might want to insert a new node into a list:
 - As the new first element
 - As the new last element
 - Before a given node (specified by a *reference*)
 - After a given node
 - Before a given value
 - After a given value
- ◆ All are possible, but differ in difficulty

Inserting as a new first element

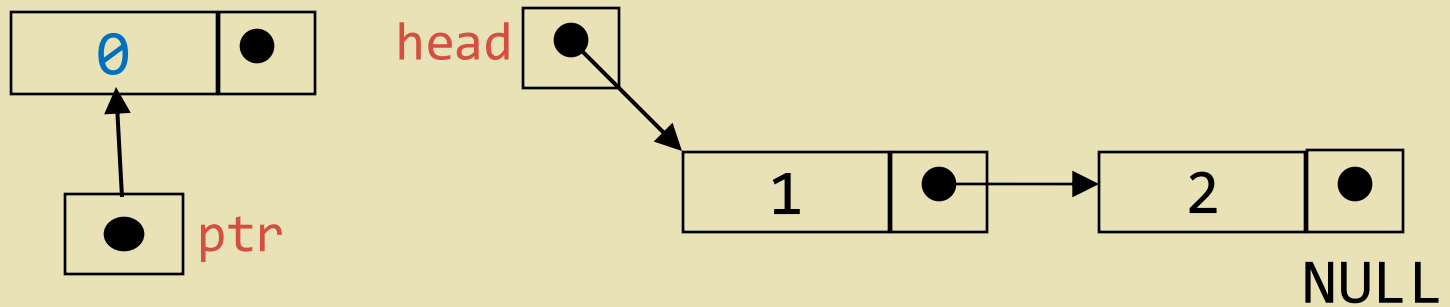
- ◆ This is probably the easiest method to implement
- ◆ Assume we have the following linked list created



- ◆ And we want to insert “zero” before the first element



Inserting as a new first element



```
ptr -> next = head;
```

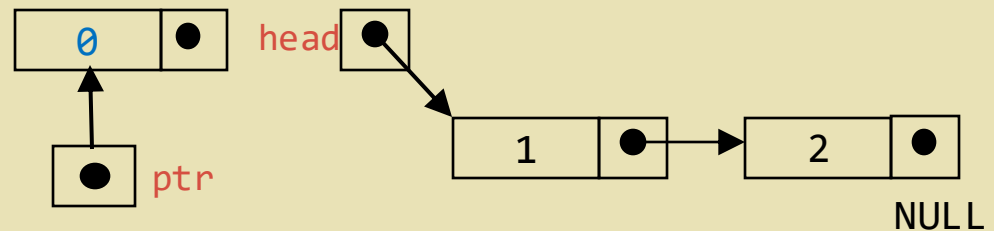
```
head = ptr; // Order matters!
```

- ◆ We could create a function `add_begin` to do it

Inserting as a new first element

```
Struct node* add_begin (struct node* head, int d)
{
    struct node *ptr = malloc(sizeof(struct node));
    ptr -> data = d;
    ptr -> next = NULL;

    ptr -> next = head;
    head = ptr;
    return head;
}
```

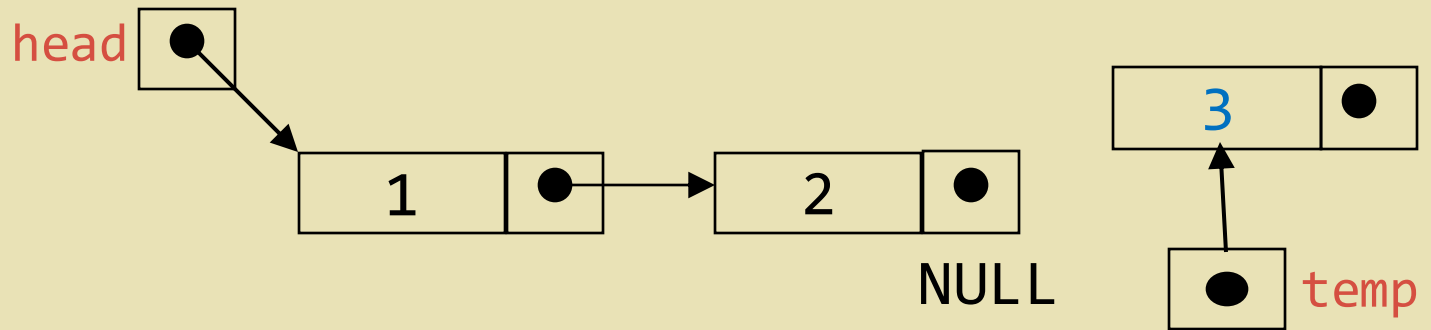


```
// call the function, assuming the linked list already
    created
```

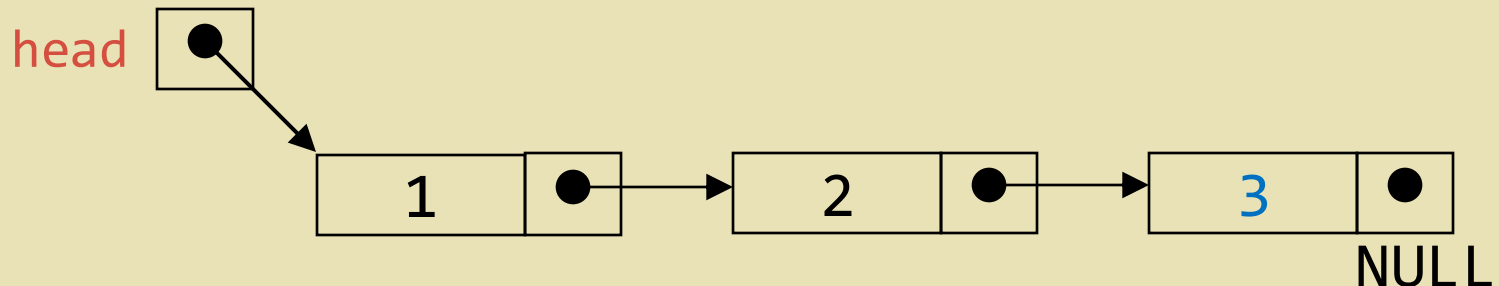
```
int add_data = 0;
head = add_begin(head, add_data);
```

Inserting as a new last element

- ◆ Insert to the end
- ◆ Assume we have the following linked list created

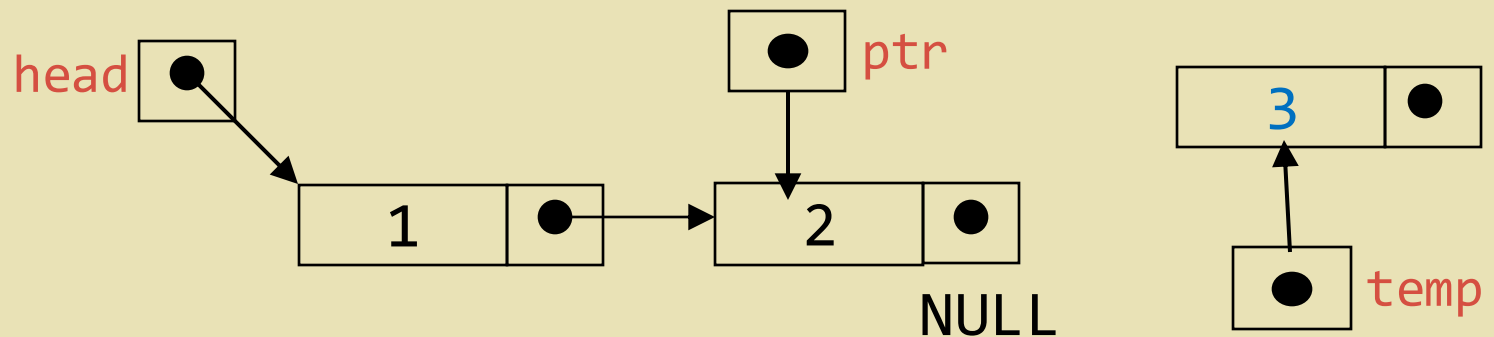


- ◆ And we want to insert “three” at the end



Inserting as a new last element

- ◆ How do we know where is the end? -- Traversal



`ptr -> next = temp;`

- ◆ We could create a function `add_end` to do it



Inserting as a new last element

```
void add_end (struct node *head, int d)
{
    struct node *ptr, *temp;
    ptr = head;
    temp = (struct node *)malloc(sizeof(struct node));

    temp -> data = d;
    temp -> next = NULL;

    while(ptr -> next != NULL){
        ptr = ptr -> next;
    } // Traversal the current list

    ptr -> next = temp;
}
```

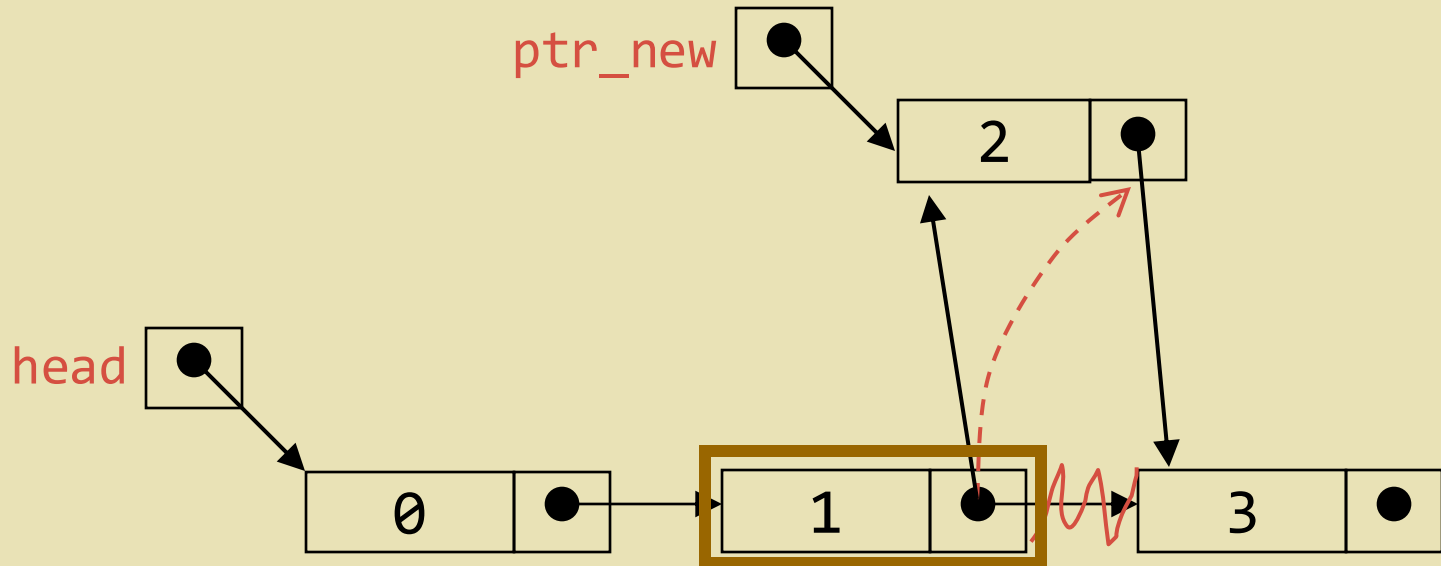



Inserting as a new last element (cont)

```
// call the function, assuming the linked list already  
created
```

```
int add_data = 3;  
add_end(head, add_data);
```

Inserting after an element (animation)



Find the node you want to insert after

First, copy the link from the node that's already in the list

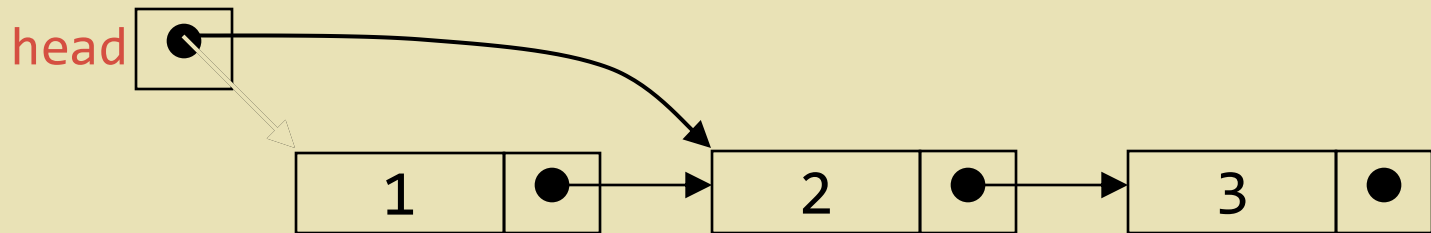
Then, change the link in the node that's already in the list

Deleting a node from a SLL

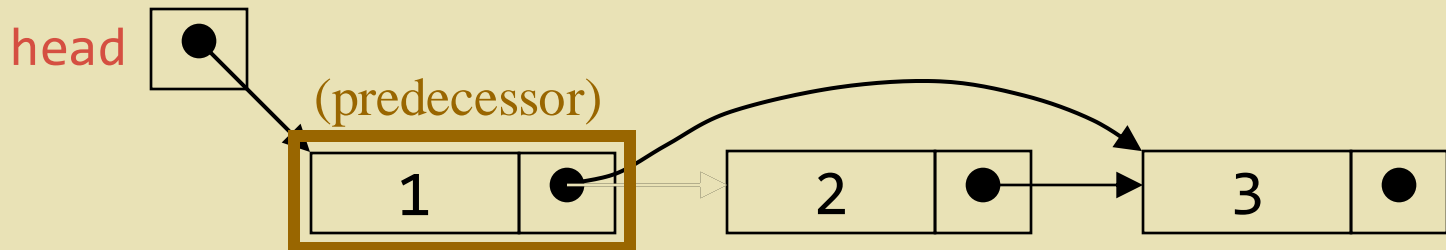
- ◆ In order to delete a node from a SLL, you have to change the link in its *predecessor*
- ◆ This is slightly tricky, because you can't follow a pointer backwards
- ◆ Deleting the first node in a list is a special case

Deleting an element from a SLL

- To delete the first element, change the head



- To delete some other element, change the link in its predecessor



- Deleted nodes will eventually be garbage collected



References and Useful Resources

- ◆ Video “Creating the Node of a Single Linked List” by Neso Academy
 - <https://www.youtube.com/watch?v=DneLxrPmmsw>
- ◆ C struct and malloc
 - <https://www.cs.toronto.edu/~heap/270F02/node31.html>
- ◆ Data structures and applications
 - <https://www.geeksforgeeks.org/what-is-data-structure-types-classifications-and-applications/>

That's
about this
lecture!

