COMPUTER ENGINEERING LABORATORY

Luigi Rizzo

luigi.rizzo@unipd.it

October 2024-January 2025

UNIVERSITÀ DEGLI STUDI DI PADOVA

Lists

Memory management

During previous week lessons

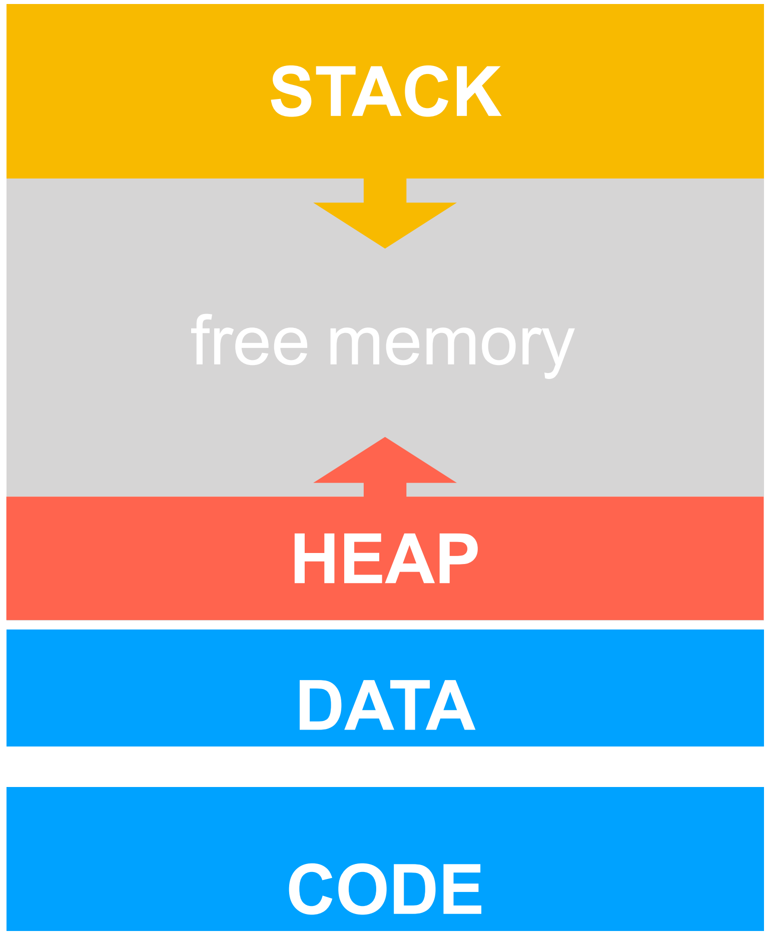
- We recalled the ways in which the C language manages memory (RAM) allocation
- We saw in particular static and automatic allocation
- We introduced dynamic memory allocation and the main C functions associated with it (malloc, calloc, realloc and free)

Recap: memory allocation

The term memory allocation is used to indicate the allocation of a block of RAM memory to be used by a program/process.

RAM memory can be allocated:

- Statically
- Automatically
- Dynamically



Memory management

- The core of the dynamic memory allocation system in C consists of the functions:
  - malloc()/calloc()/realloc(): (re)allocate a block of free memory
  - free(): frees previously allocated memory
  - These functions are defined in the <stdlib.h> library
- Dynamic allocation is fundamental in practice (memory needs are often not known a priori)
  - This applies first and foremost to “simple” variables / data structures
  - But it is essentially unavoidable for more advanced data structures (lists, trees, graphs, ...)

The lists

- In C programming, a list is not a built-in data structure like in some other programming languages.
- However, you can implement a basic form of a list using structures and pointers.
- Lists, also known as linked lists, are a dynamic data structure in C that allow for the storage and manipulation of a collection of elements.
- Unlike arrays, linked lists don't have a fixed size and can grow or shrink during runtime.
- Each element in a linked list is represented by a node, and nodes are linked together using pointers.

The lists

- A list is a collection of elements, where each element contains some data and a reference (pointer) to the next element in the sequence. A list is a finite succession of elements of one type.
- The information encoded by the list concerns:
  - The succession of elements (values)
  - The order relationship between the elements themselves
- The list is qualified not only by the values it represents but also by the operations performed on it
  - Insertion and deletion (head, tail, intermediate)
  - Visit / search
  - Initialization

The lists

- We can obtain different types of lists by defining different methods of inserting/deleting elements
  - stack: it is a LIFO type structure, i.e. "last in first out
  - queue: it is a FIFO type structure, i.e. "first in first out"
- A list can be represented:
  - In **sequential** form: the elements are represented in an array and their order is implicitly encoded by the position
  - In **linked** form: in this case the relationship is made explicit and each element is associated with the information that identifies the successor (this can be an index or a pointer)
- A **pointer-linked list** is a succession of elements (nodes) connected by pointers

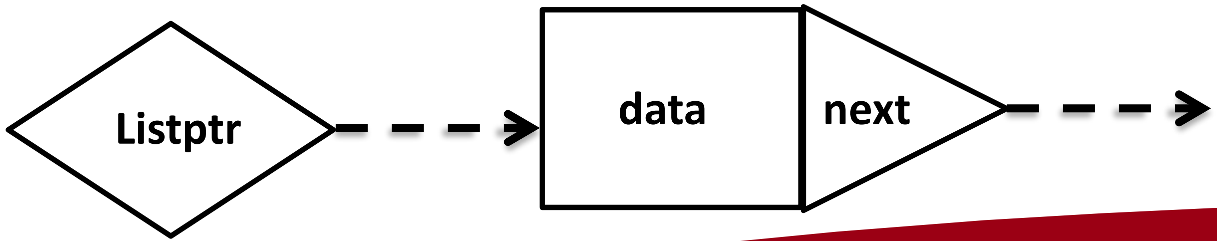
linked lists vs. arrays

- Linked lists are dynamic structures and their size can increase/decrease at run time, unlike arrays, linked lists allow for efficient insertion and deletion of elements without the need for resizing or reallocating memory.
- Arrays, on the other hand, have a pre-fixed size and can fill up
  - The elements of an array are stored contiguously;
    - + this allows immediate access to an item
    - - but item insertions or deletions are slow
    - The elements of a linked list, are "logically" in sequence, but will occupy non-contiguous memory addresses

Lists linked by means of pointers

- Linked lists do not offer immediate access to their items, but inserting/deleting items can follow different strategies
  - LIFO, FIFO, in “generic” position
- The atomic element of a list is the **<u>node</u>** that can be defined by a “self-referential” structure:

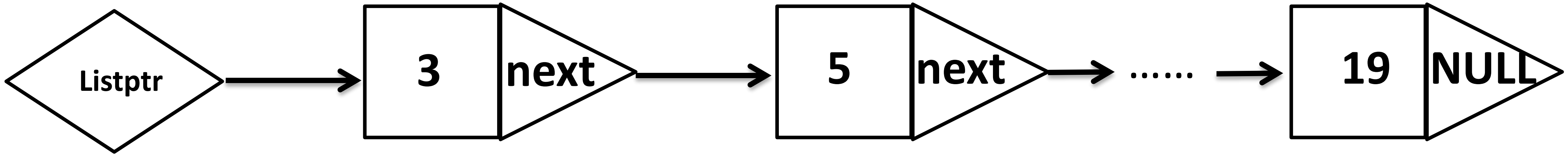
```
struct Node {
   int data;
   struct Node* next;
};
```



Lists linked by means of pointers

Example / graphical representation of a linked list:

List of elements: {3, 5, …, 19}



Functions defined on linked lists

Types of Linked Lists

- **Singly Linked List**: each node points to the next node.
- **Doubly Linked List**: each node points to both the previous and the next node.
- **Circular Linked List**: the last node points back to the first node

Functions defined on linked lists

- Using linked lists requires some basic functions definition:
  - Initialization
  - Visit (usually will print node values)
  - Search for an item
  - Insertion
  - Deletion
  - Now we will start to
    - develop insertion, visit, etc. functions on lists

Functions defined on linked lists

- Let’s consider a basic structure of a node in a singly linked list:

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

- data: Holds the value of the node.
- next: A pointer to the next node in the sequence.

A linked list is formed by connecting these nodes together. The last node in the list points to NULL to indicate the end of the list.

Functions defined on linked lists

- Initialization

```
// Initialize an empty list
struct Node* myList = NULL;
```

Functions defined on linked lists

```
// Function to create a new node
struct Node* createNode(int value) {
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  if (newNode == NULL) {
    printf("Memory allocation failed\n");
  }
  else {
      newNode->data = value;
      newNode->next = NULL;
  }
  return newNode;
}
```

Functions defined on linked lists

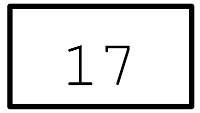
```
int main()
{
	struct node *list = NULL;
	struct newNode = NULL;
	// ...
	if ((newNode = createNode(17)) != NULL)
	{
		//…
	}
	// ...
}
```

Functions defined on linked lists

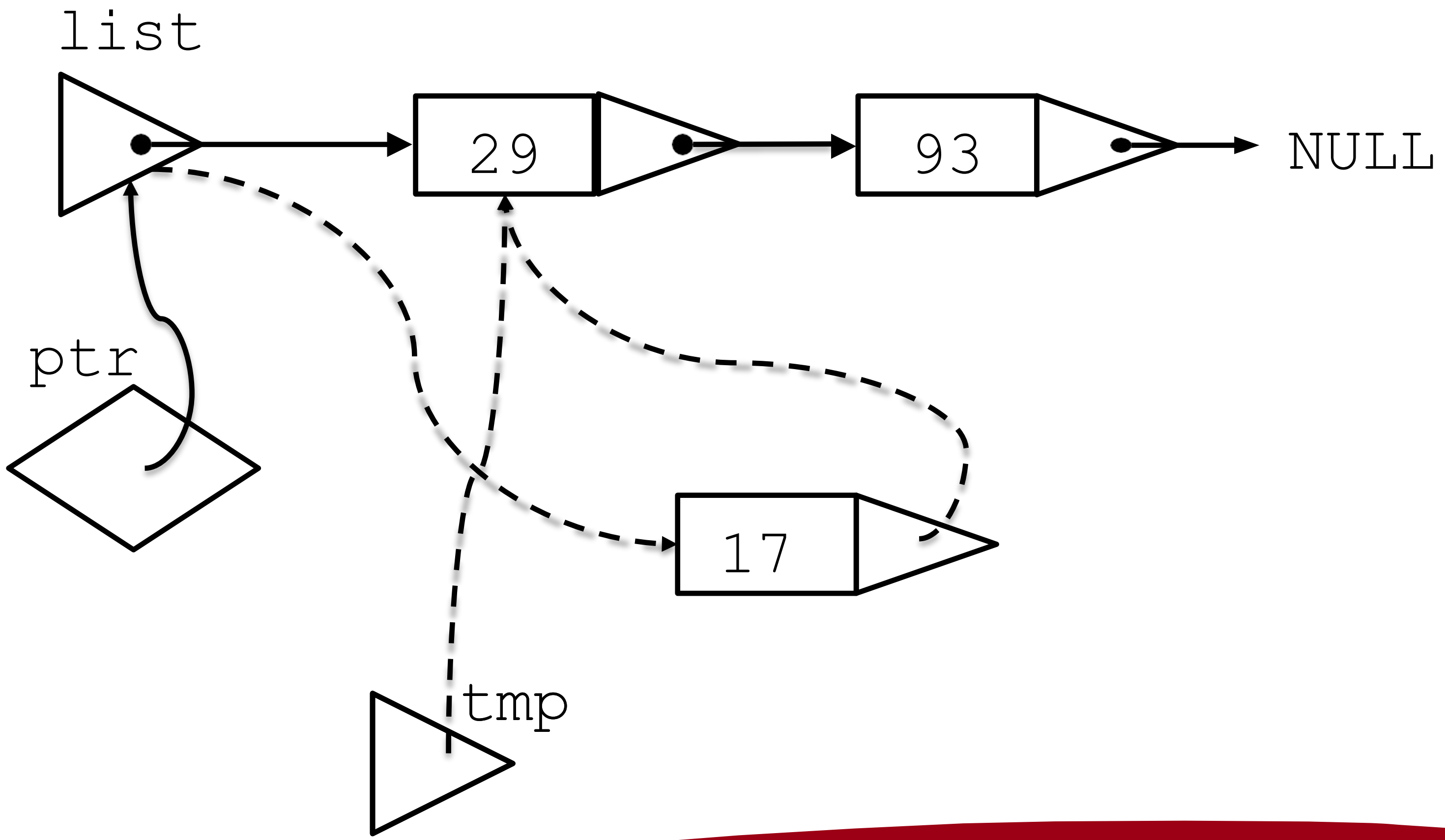
- Insertion
    - Insertion at the beginning of the list
    - Insertion at the end of the list
    - Insertion in a certain position of the list

Functions defined on linked lists

- Insertion at the beginning of the list

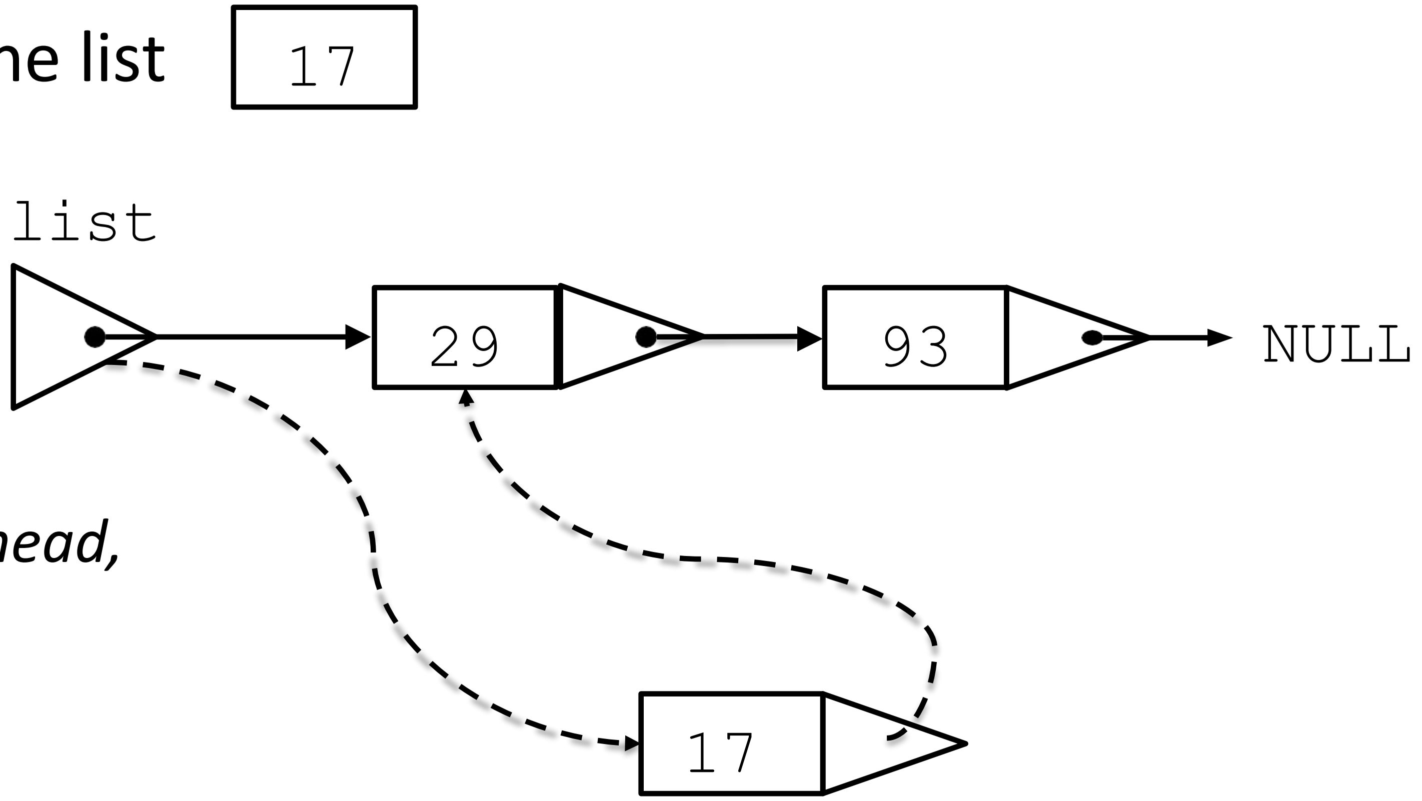


```
void head_insert(struct Node **ptr,
         struct Node *newNode)
{
    struct Node *tmp;

    tmp = *ptr;
    *ptr = newNode;
    (*ptr)->next = tmp;
}
```



Functions defined on linked lists

- Insertion at the beginning of the list

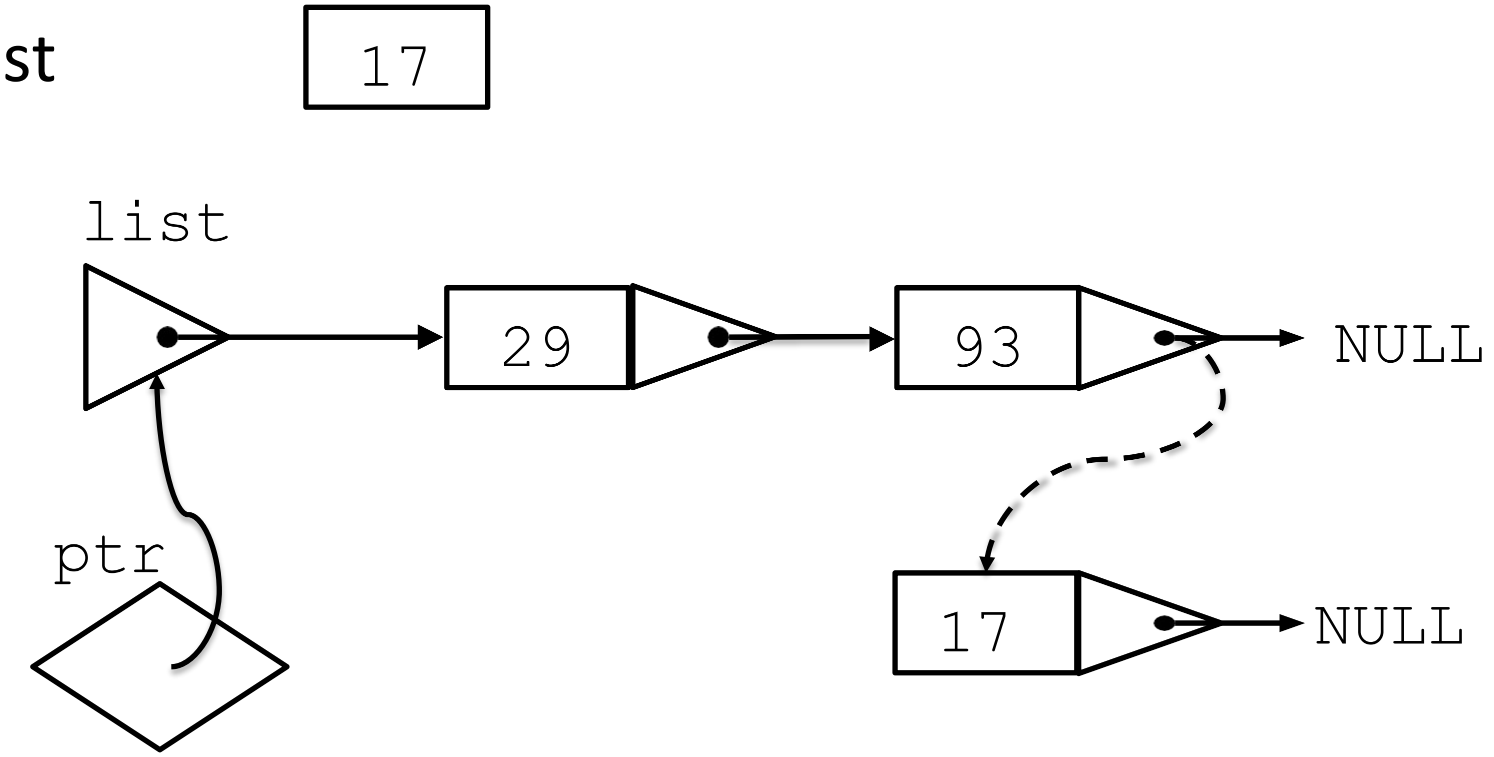


```
struct Node*head_insert(struct Node *head,
            struct Node *newNode)
{
     newNode->next = head;
    return(newNode); //new head


}
```

Functions defined on linked lists

- Insertion at the end of the list

```
void tail_insert(struct Node **ptr,
        struct Node *newNode)
{
    struct Node *tmp;

    tmp = *ptr;
    while (tmp->next != NULL)
        tmp = tmp->next;
    tmp->next = newNode;
}
```



```
if (*ptr==NULL)
    *ptr = newNode;
```

Functions defined on linked lists

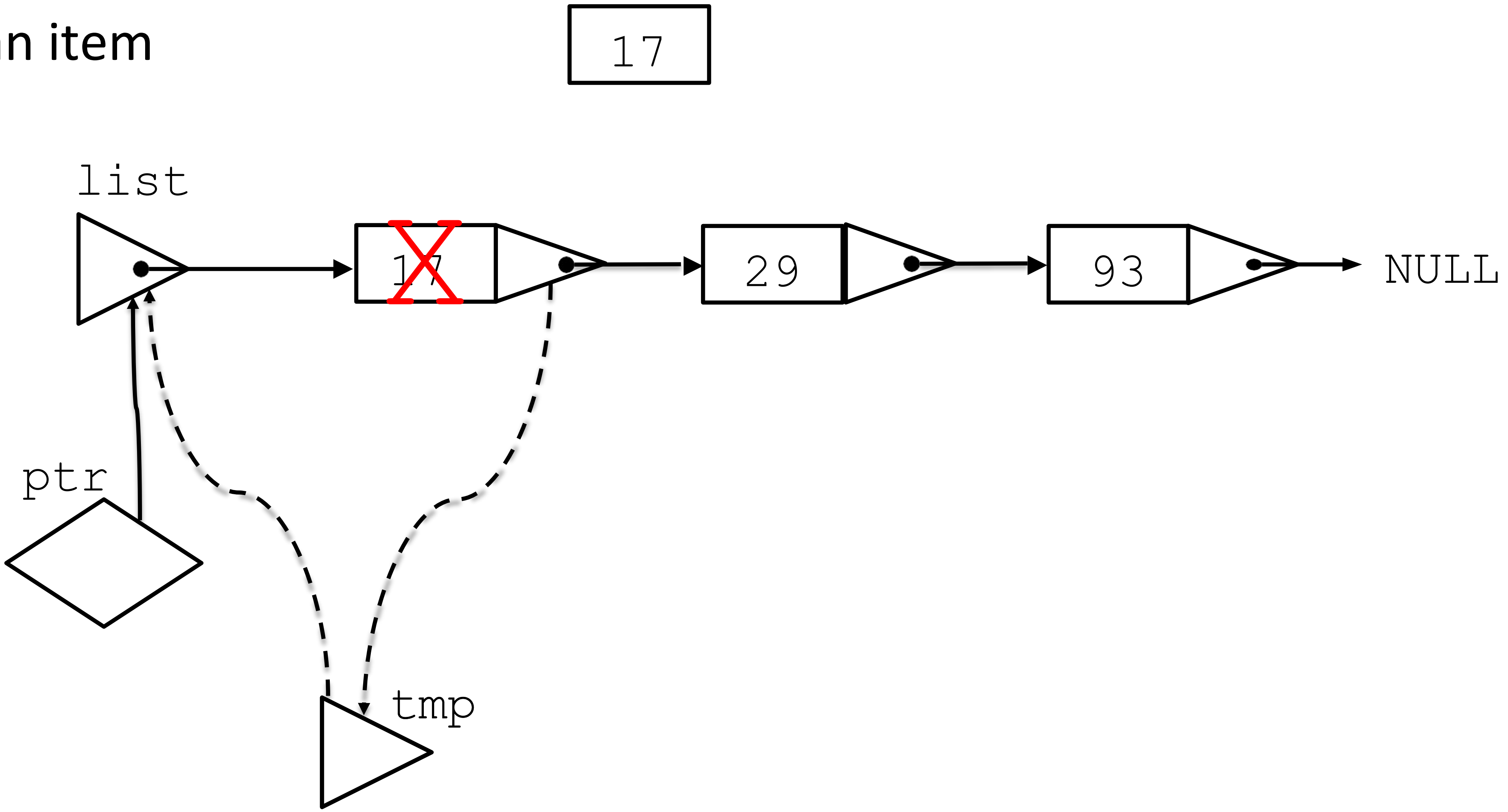
Università degli Studi di Padova

- Search for an item

```
struct Node * searchItem(struct Node *head, int itemToSearch)
{

    while ((head != NULL) && (head->data != itemToSearch))
            head = head->next;

    return(head);
}
```

Functions defined on linked lists

- Delete an item



Functions defined on linked lists

- Delete an item

```
unsigned int deleteItem(struct Node *head, int itemToDelete)
{
        struct Node *prev = head;
        while ((head != NULL) && (head->data != itemToDelete))
        {
                prev = head;
                head = head->next;
        }
        if (head->data==itemToDelete)
        {
                prev->next = head->next;
                free(head);
                return(0); // item deleted
        }
        else
                return(1);
}
```

Functions defined on linked lists

- Delete an item

```
unsigned int deleteItem(struct Node **ptr, int itemToDelete)
{
        struct Node *head = *ptr;
        struct Node *prev = head;
        while ((head != NULL) && (head->data != itemToDelete))
        {
                prev = head;
                head = head->next;
        }
        …
}
```

```
if (head->data==itemToDelete)
{
        if (head != *ptr)
        {
                prev->next = head->next;
        }
        else
        {
                *ptr = head->next;
        }
        free(head);
        return(0); // item deleted
}
else
        return(1);
```

Functions defined on linked lists

- Delete an item

```
struct Node* deleteByValue(struct Node* head, int key) {
    struct Node* temp = head;
    struct Node* prev = NULL;

    // If the head node itself holds the key
    if (temp != NULL && temp->data == key) {
        head = temp->next; // Change head
        free(temp);        // Free old head
        return head;
    }
    …
}
```

```
// Search for the key
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    // If the key was not present
    if (temp == NULL) return head;

    prev->next = temp->next; // Unlink the node
    free(temp);              // Free memory
    return head;
```

Functions defined on linked lists

- Visit / traverse

```
// Function to print the elements of the list
void printList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}
```

Other types of lists

The list from the previous slides is called a one way linked list.

- One way lists are generally better than arrays.
    - Many deletions and additions
    - Fairly simple to manage

In the one way list presented earlier, we had no concern for order.

- Insert at the beginning
- Append at the end

An ordered list adds elements in order based on a key field of the record.

- Other types of lists give other benefits.

Other types of lists

- A one way list has certain deficiencies.
  - Can't visit any element from any other element
- Some list problems require additional pointers to provide additional efficiency.
  - May have need for doubly linked list
  - An editor with a linked list of lines
- May have a list where each element itself contains a head pointer for another list (a network)
  - Linked airline flights with linked passengers
  - Linked words in a file with linked line number, (i.e, a cross reference)

Ordered lists

In the context of C programming, an "ordered list" typically refers to a collection of elements where the elements are arranged in a specific order based on a comparison function. The ordering could be ascending or descending, depending on the desired arrangement.

The term "ordered list" itself does not correspond to a specific built-in data structure in C. Instead, you might implement an ordered list using arrays or linked lists and ensure that the elements are inserted in the correct order.

Ordered lists

Same as previous linked list except insertion is a function of a key field (int data for example). Elements are then in order.

Let’s consider a simple example of how you might implement an ordered list using a singly linked list in C.

Ordered lists

```
// Function to insert a node in an ordered list
struct Node* insertInOrder(struct Node* head, struct Node *newNode) {
    // If the list is empty or the new node should be inserted at the beginning
    if (head == NULL || newNode->data < head->data) {
        newNode->next = head;
        return newNode;
    }
    // Traverse the list to find the correct position for the new node
    struct Node* current = head;
    while (current->next != NULL && current->next->data < newNode->data) {
        current = current->next;
    }

    // Insert the new node in the correct position
    newNode->next = current->next;
    current->next = newNode;

    return head;
}
```

Functions defined on linked lists

Università degli Studi di Padova

- Insert an item after…

```
unsigned int insertItem(struct Node *head, int itemToInsAfter, struct Node *toBeInserted)
{
        struct Node *tmp;
        while ((head != NULL) && (head->data != itemToInsAfter))
                head = head->next;

        if (head->data== itemToInsAfter)
        {
                tmp = head->next;
                 head->next = toBeInserted;
                 toBeInserted->next = tmp;
                return(0); // item inserted
        }
        else
                return(1);
}
```

Circular lists

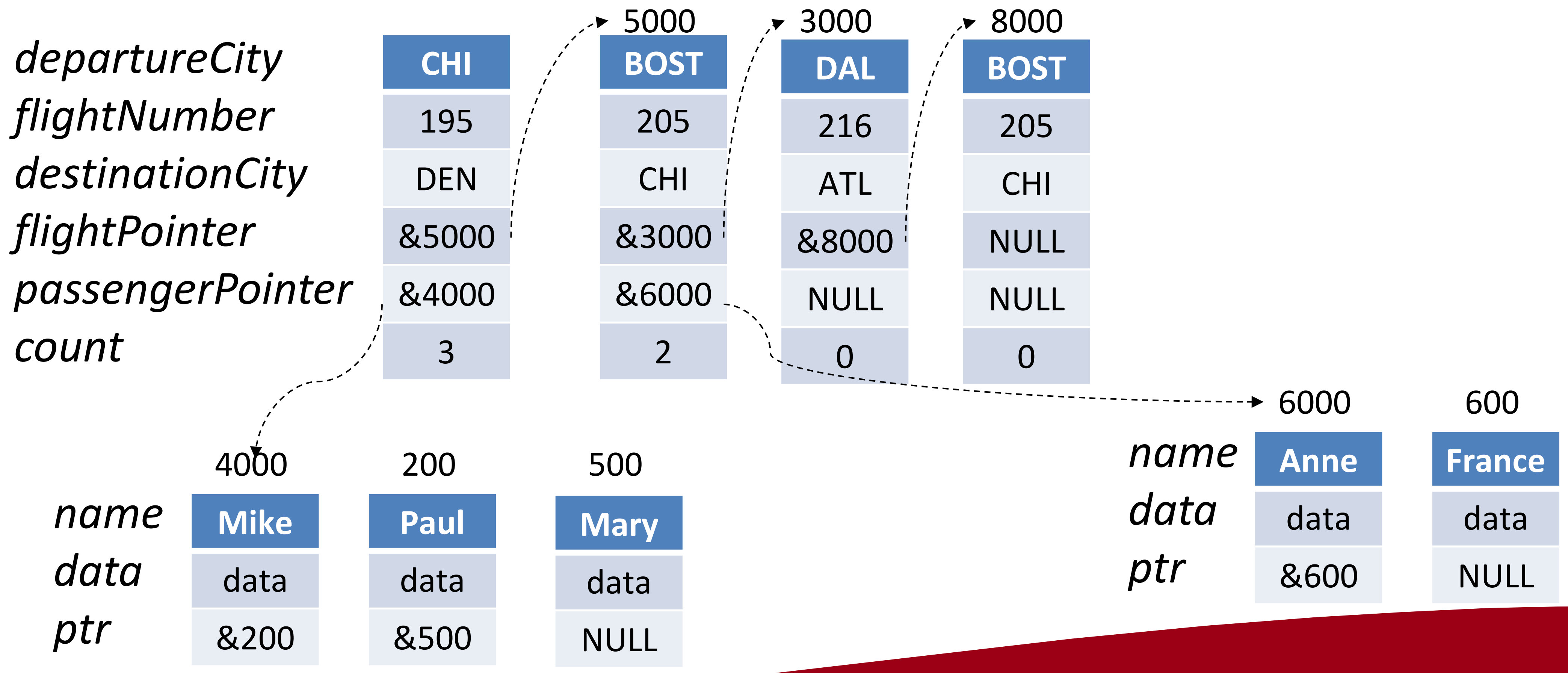
- There are some basic differences between a circular list and a one directional list.
  - There is no first element.
  - There is no end of the list.
- There will be a current element.
  - This is the starting point for the next list operation.
- Each element points to the next one.
- Circular lists are used heavily in memory management schemes.

Two ways lists

- A two_way list structure needs two pointers.
- Two ways lists allow processing in either direction.
  - Can cut down access time
  - Print from 20 to 50
  - Print 16 and preceding 4
  - Very often two ways lists are also circular.
- The list has two pointers.
  - Forward pointer link
  - Backward pointer link
- There are two print/visit routines forward, backward.

Nested lists

- There are many other kinds of linked lists applications.
- Consider a list of flights, each with a log of passengers.



What are Lists?

- An element of a list is usually defined as a structure.
  - A passenger

    ```
    struct passenger {
    char name[20];
    int flight_no;
    char seat[4];
    };
    ```

  - A job in an operating system

    ```
    struct job {
    int owner;
    int priority;
    char *files[20];
    };
    ```

- A window

  ```
  struct window {
  int x_upper_left;
  int y_upper_left;
  int x_lower_right;
  int y_lower_right;
  };
  ```

What are Lists?

- A list is a collection of (usually) like objects.
  - passengers on an airline
  - jobs in an operating system
  - windows on a display
- Lists are usually dynamic.
  - The number of elements in the list varies with time.
  - There is no upper limit on the size of the list.
- Common list operations include:
  - adding an element
  - inserting an element
  - deleting an element
  - printing the list
  - combining two lists

Lists as arrays

- A list is a dynamic data structure. An array is fixed.
- This contradiction leads to inefficiencies in:
  - Adding to the list
    - what about when there is no more room?
  - Deleting from a list
    - moving too much data
  - Combining two lists
    - not room enough in either list

Lists as arrays

- An array can be used to represent a list.
  - An array is a fixed size data type.
  - Size is based on a worstcase scenario.
  - The number of elements in the list would be kept in a separate variable.
- Array representation could lead to inefficiencies.
  - Adding an element
    - Since an array is a fixed data structure, there would be no way of extending it.
  - Inserting an element
    - All elements below the inserted one would need to be pushed down one element.

Lists as arrays

- Deleting an element
  - Each element needs to be moved up a position, or the position of the deleted one could be marked with a special value.
- Combining two lists
  - The sum of the number of elements from the two lists could be larger than the capacity for either array.

- A better representation than an array is a data structure, which is allocated only when it is needed.
  - The need is usually signaled by a user request to the program.

Benefits of Linked Lists

- Linking elements is a dynamic way of building lists.
- The problem of fixed size disappears.
- Deleting an element becomes a matter of pointer manipulation.
- Inserting an element also is a pointer manipulation problem.
- Combining two lists need not worry about size restriction.

A list of linked elements

- Create the storage for an element when it is needed.
  - Describe a template for any list element.
  - Allocate as needed malloc.
- Leads to many allocations at various locations
  - Provide an extra structure member, a pointer.
  - Link each new allocation to previous ones.
    - Insert append in order

A list of linked elements

- Inserting or appending an element
    - The problem of extending the array size disappears.
- Deleting an element
    - There is no inefficiency involved as with an array.
- Combining two lists
    - There is no concern about combined sizes.
    - Modify one pointer.