# EEE243 – Applied Computer Programming

Data Structures

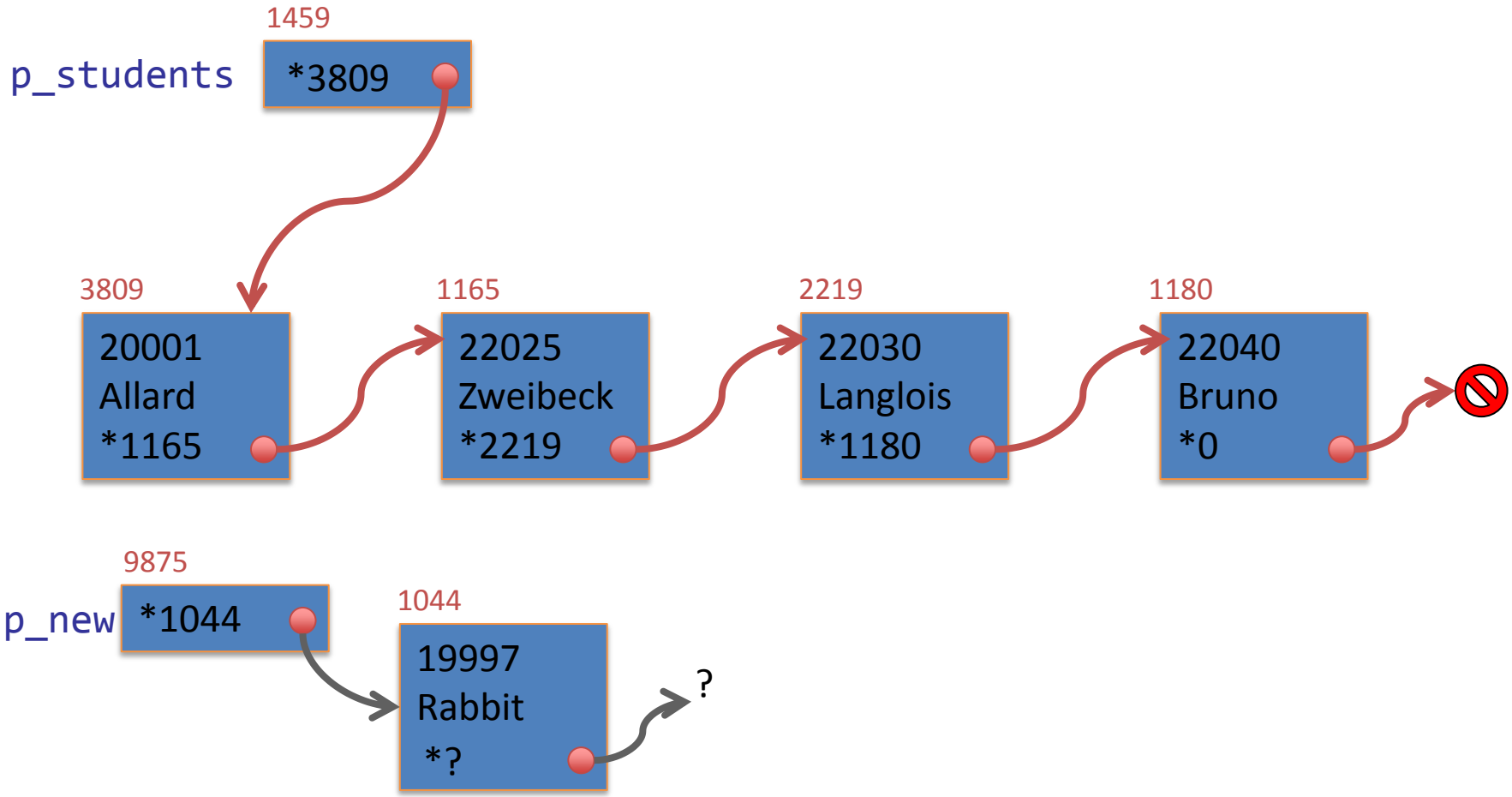

# Outline

1. Review
2. Ordered and unordered LL
3. Ordered and unordered arrays
4. Examples
5. Other data structures

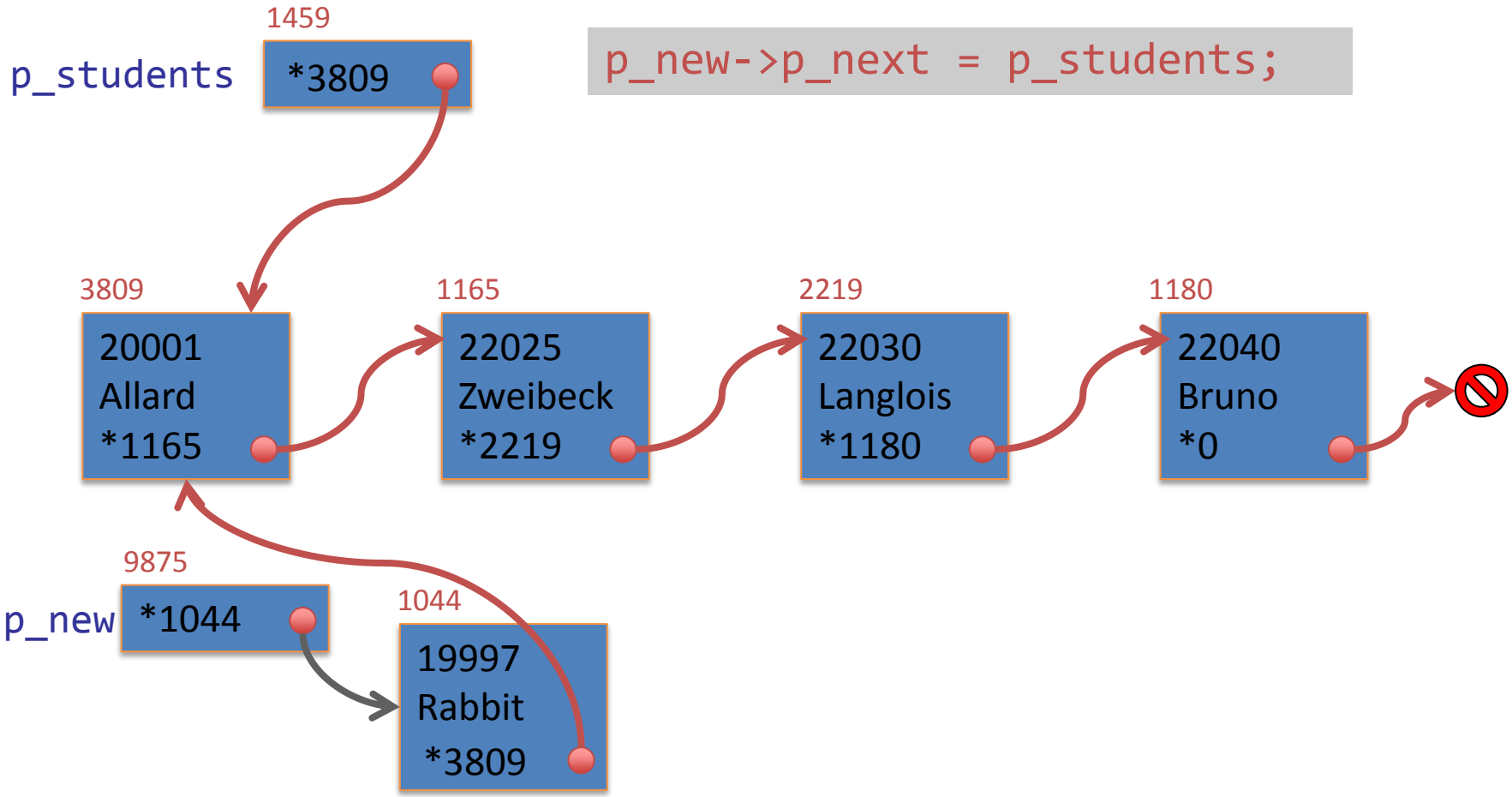
# Review

- Where in memory are linked lists stored?
- Does each Node in a linked list have a symbolic name (variable name) associated with it?
- What is the head pointer used for?
- Describe the steps to traverse a linked list.
- Describe the steps to insert somewhere other than the start in a linked list.

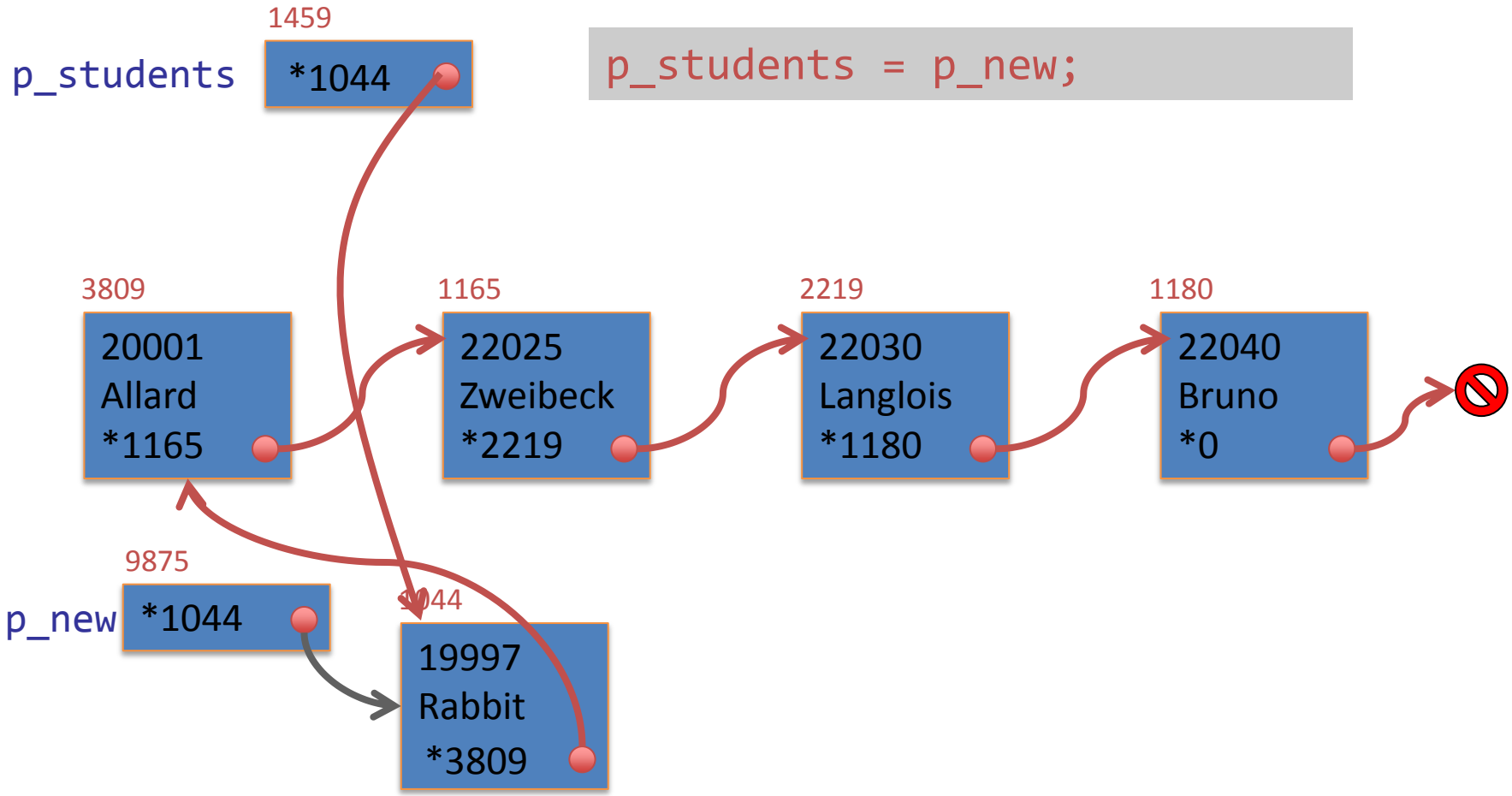
# Recall: insert at beginning of list



# Insert at beginning of list



# Insert at beginning of list



# Ordered and unordered data structures

- When you use data structures such as arrays or LL you must be aware of *execution time tradeoffs*
- When dealing with a sequence of `struct`, you will have to make two main design decisions based on these tradeoffs
  - Selection of a data structure (array or LL) and
  - Selection of an ordered or an unordered sequence

# Ordered and unordered data structures

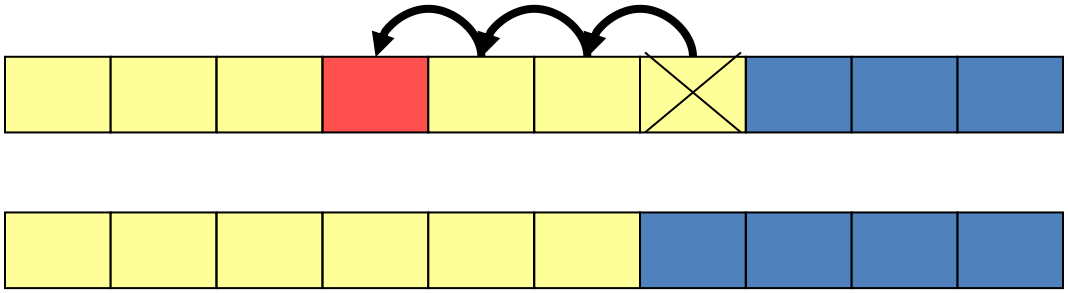
- Elements inside arrays and LL can be ordered using some sorting *key* in the structure
- A key can be any of the fields in your structure:
  - Last name,
  - College number,
  - Height,
  - Weight,
  - SN,…
- The sorting can also be done with a combination of two keys:
  - By `last_name` (*primary key*) and then by `first_name` (*secondary key*).

# Ordered Arrays

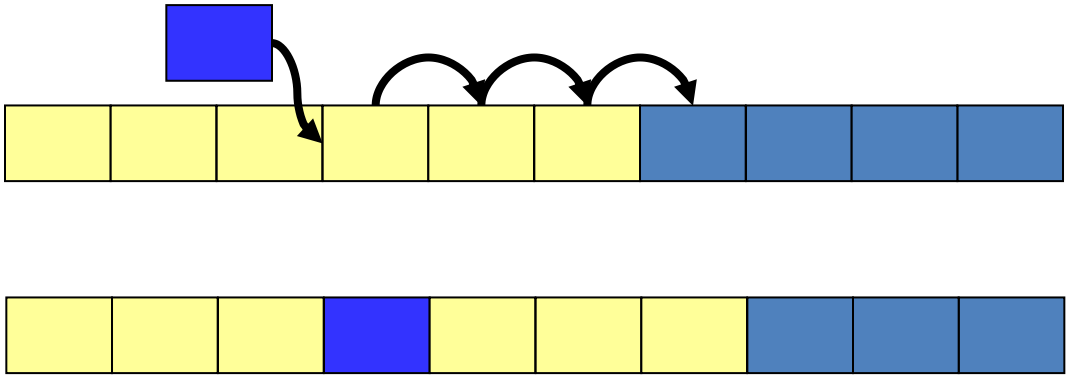
- We previously discussed that there are performance issues if we want to keep array elements in some arbitrary order
- The execution time for *inserting* and *deleting* nodes is proportional to the number of elements in the array
  - In the *worst case*, you may have to move all the elements in the array
- However, *searching* in an ordered array can be very fast using a binary search

# Ordered Arrays – Delete Insert

- Deleting an element in an array:



- Inserting an element in an array:



# Unordered Arrays

- By contrast, *inserting* elements in an unordered array is very fast;
  - you put the new element at the end of the array in the first free slot
  - for this you must keep a variable that contains the first empty element in the array
- However, *searching* and *deleting* in an unordered array is proportional to the number of elements in the array

# Unordered Linked Lists

- Unlike arrays, linked lists do not have *indexes*;
  - even if your LL is ordered, you cannot use a binary search
  - You cannot jump “inside” a LL because there are no symbolic names

- The design decision that you made for the ordering of your LL will not help speed up the execution time
  - You will need to start searching from the head of the list and go from node to node for all operations (searching, inserting and deleting)

# Unordered Linked Lists

- It is obvious that an unordered list will have a search time proportional to the number of elements in the list.
- Deleting a node in a LL is also proportional to the number of elements, because we must find the right node to delete
- Inserting in an unordered LL is however quite fast – you either insert at the head or tail
  - the size of the list does not affect this operation

# Ordered Linked Lists

- If the list is *ordered*, you can stop searching:
  - As soon as you find your structure(s)
  - When you pass the point where the structure would be
    - you then know it is not in the list
- If you intend to search relatively often (in comparison inserting/deleting elements), the *ordered LL* is a slight improvement over the unordered LL ,
- Still in the worst case, you may have to go through the entire LL
  - The execution time is proportional to the number of elements

# Choosing the right Data Structure

- If you intend to *insert and remove a lot* of information on a regular basis and perform relatively *fewer searches*; then use an *unordered LL*
- If you intend *few changes* in your sequence and perform relatively *many searches*; it is better to use an *ordered array*.
  - As long as the array size can be specified so that it doesn't have to change (a max size of elements is known)
  - Better yet, use an implementation based on hash tables.

# Ordered LL with multiple ordering criteria

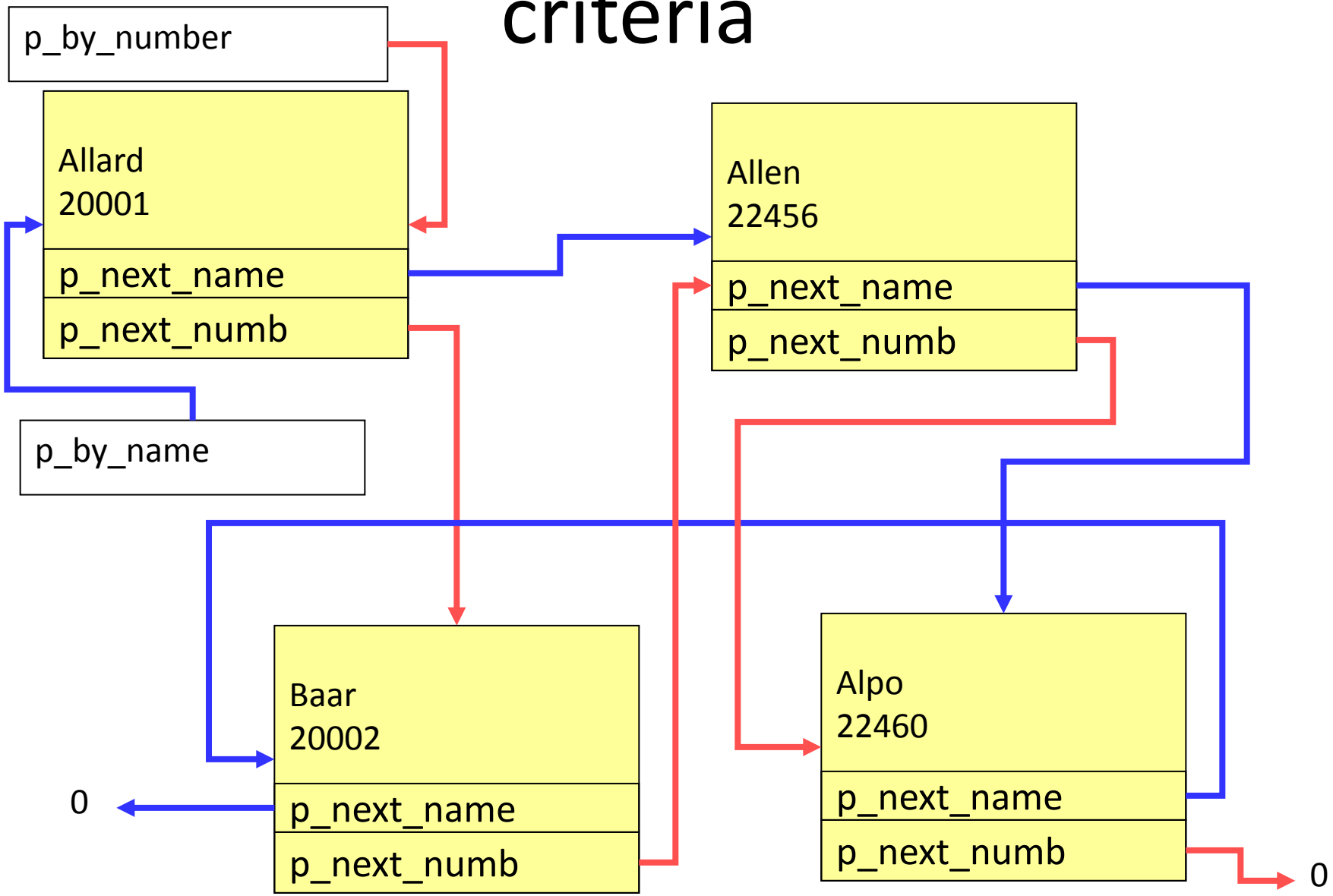
- One of the powers of LL, is that *the same LL* can be ordered in *two different ways* by using two different pointers inside each node
  - In fact you could use *N* pointers inside each node to order your nodes in *N* different ways;
  - You would need *N* arrays or *N* sorting operations to accomplish the same without a linked list
- e.g., if I want to be able to print an ordered RMC student list by last names or college numbers, I could use an ordered LL with two pointers inside each `Student` and keep these relations

# Ordered LL with multiple ordering criteria

```
typedef struct STUDENT_TAG {
     char first_name[15];
     char last_name[25];
     unsigned long college_number;
     float average;
     struct STUDENT_TAG *p_next_name;
     struct STUDENT_TAG *p_next_numb;
} Student; //name of the type

Student *p_by_name = NULL;
Student *p_by_number = NULL;
```

# Ordered LL with multiple ordering criteria



# Doubly Linked List

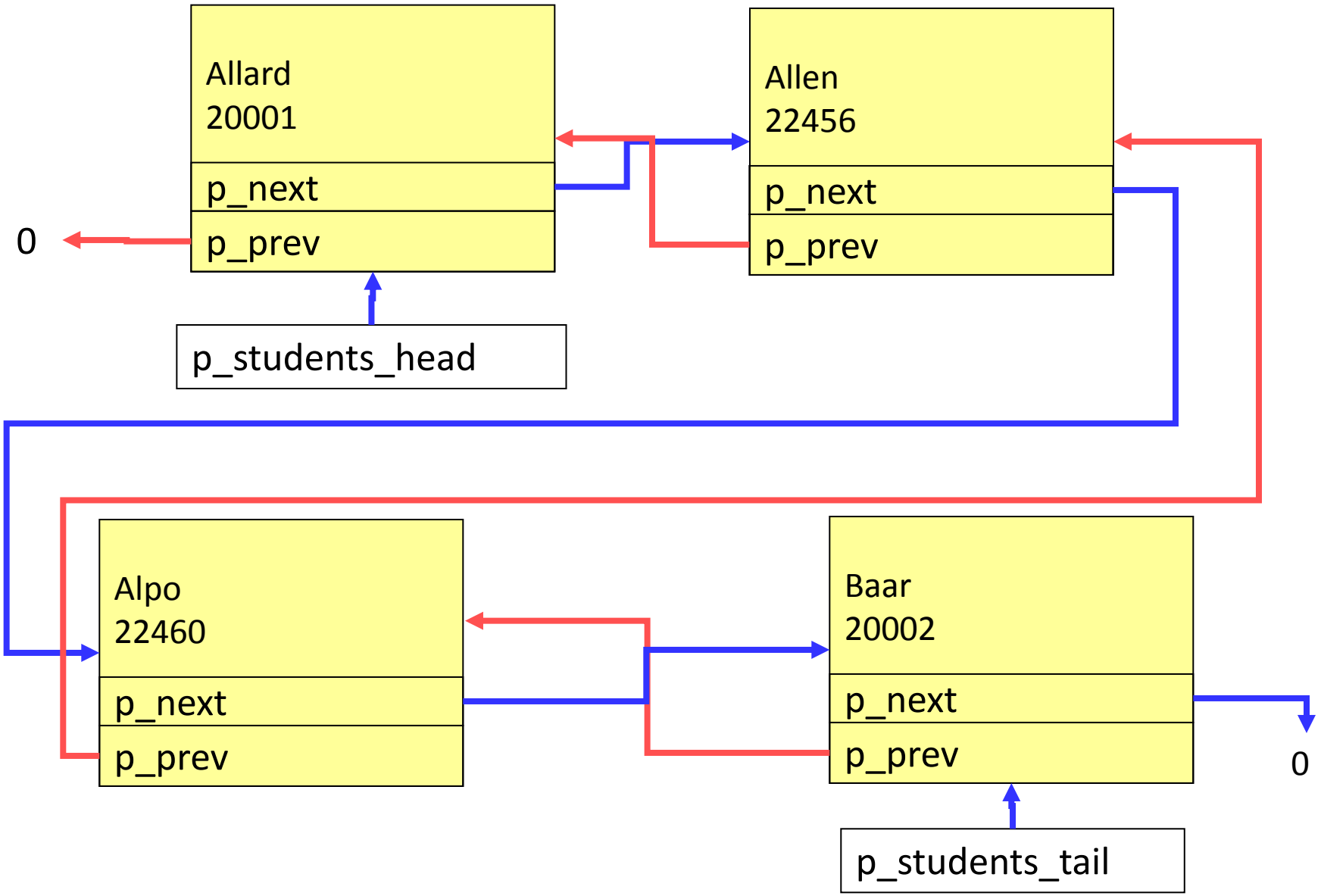
- As we just saw in the last slide, it is possible to have two pointers to create “two linked lists for the price of one”
  - more accurately for only a small price (the extra pointer requires memory)
- This can be used for more than ordering.
- A doubly linked list is a linked list that can be traversed in both directions
  - From head to tail and tail to head
- In order to build such a list, we include a pointer to next and a pointer to previous as part of the node
- We also declare a new pointer that points to the tail
  - similar to the head pointer

# Doubly Linked List

```
typedef struct STUDENT_TAG {
     char first_name[15];
     char last_name[25];
     unsigned long college_number;
     float average;
     struct STUDENT_TAG *p_next;
     struct STUDENT_TAG *p_prev;
} Student; //name of the type

Student *p_students_head = NULL;
Student *p_students_tail = NULL;
```

# Doubly Linked List

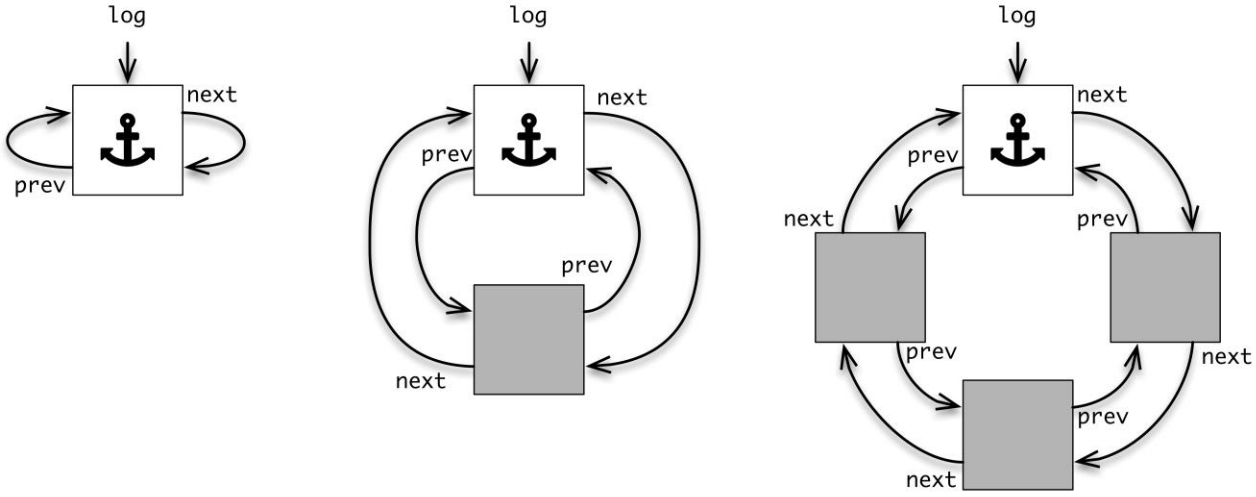


# Doubly Linked List

- The traversing operation is very similar to singly LL but more powerful
  - You can “back up” in the list a lot faster with a doubly LL
- The insert and delete operations must assign values to both the next and previous pointers inside a node
  - A slight increase in time

# Circular Linked Lists

- We need a way to mark the entry point where we start.
- One option: a sentinel node



# Real World Examples of LL

- Browser history
- Queues
- Undo-Redo functionality
- Stacks
- Representations for the corner of a polygon

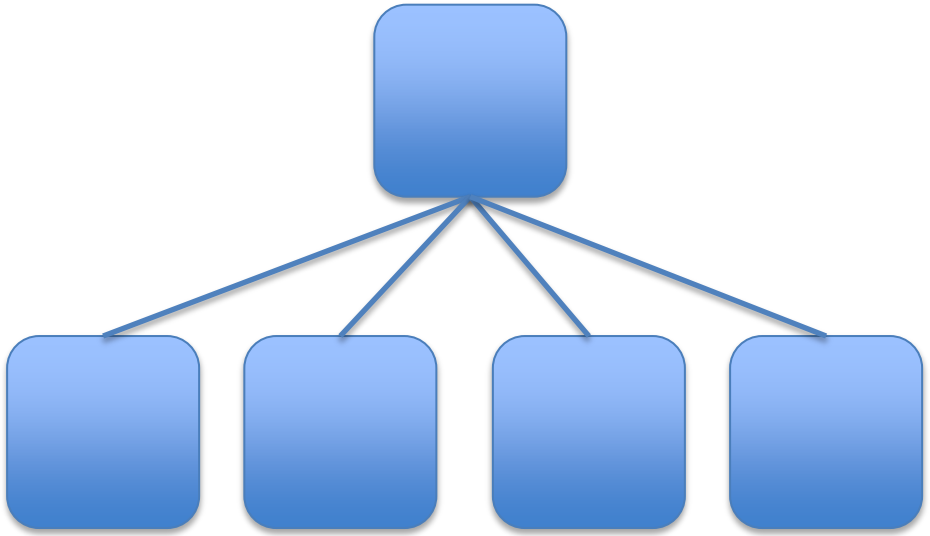
# Real World Examples of LL

- Linux processes
  - Each `task_struct` data structure describes a process or task in the system.

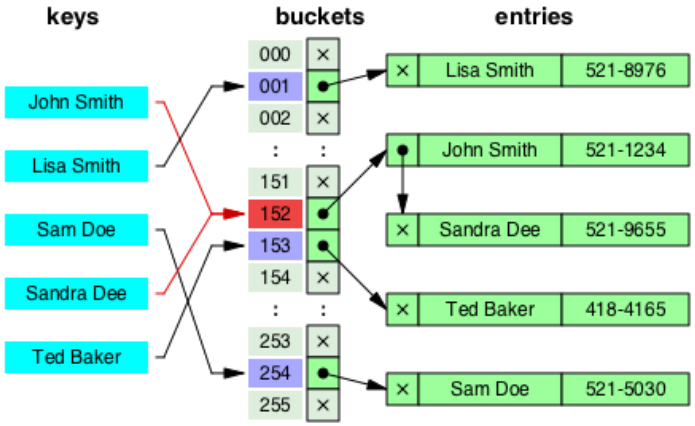
```
struct task_struct {
    volatile long       state;          /* -1 unrunnable, 0 runnable, >0 stopped */
    long                counter;
    long                priority;
    unsigned            long signal;
    unsigned            long blocked;   /* bitmap of masked signals */
    unsigned            long flags;     /* per process flags, defined below */
    int errno;
    long                debugreg[8];    /* Hardware debugging registers */
    struct exec_domain  *exec_domain;
    struct linux_binfmt *binfmt;
    struct task_struct  *next_task, *prev_task;
    struct task_struct  *next_run,  *prev_run;
    unsigned long       saved_kernel_stack;
    unsigned long       kernel_stack_page;
    int                 exit_code, exit_signal;
     ...
    };
```

# Other Data Structures

- Trees



- Hash table



# Questions

- What kind of data structure should I use if I want to enter the member of parliament after an election and be able to search for information relatively often? Ordered?
- What kind of data structure should I use if I want to keep track of letters read by my robot if I do not know how many letters to read? Ordered?

# Exercise

- Give the code for deleting a NODE from a doubly linked list.
- assume `p_walker` points to node to delete

Questions?