ITERATORS CONTD, QUEUE

Problem Solving with Computers-I

https://ucsb-cs24-sp17.github.io/

pa04 grades so far: Number of students 127 Number of submissions: 109 : 85%

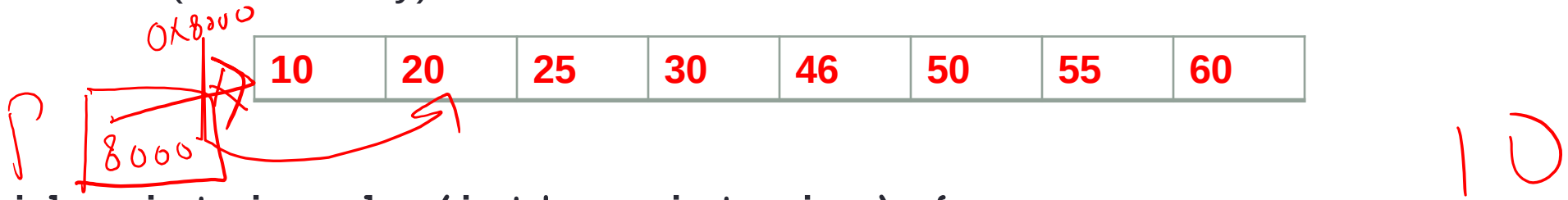
Score	Percentage of students
100	21%
90 - 100	1.5%
80 - 90	0.5%
70 - 80	2%
Below 60% (not zero)	49%
0	26%

Announcements

- Midterm next week (Wed)
- Will heavily focus on PA3 and PA4 material.
- Topics include chapters 1-7

C++ Iterators

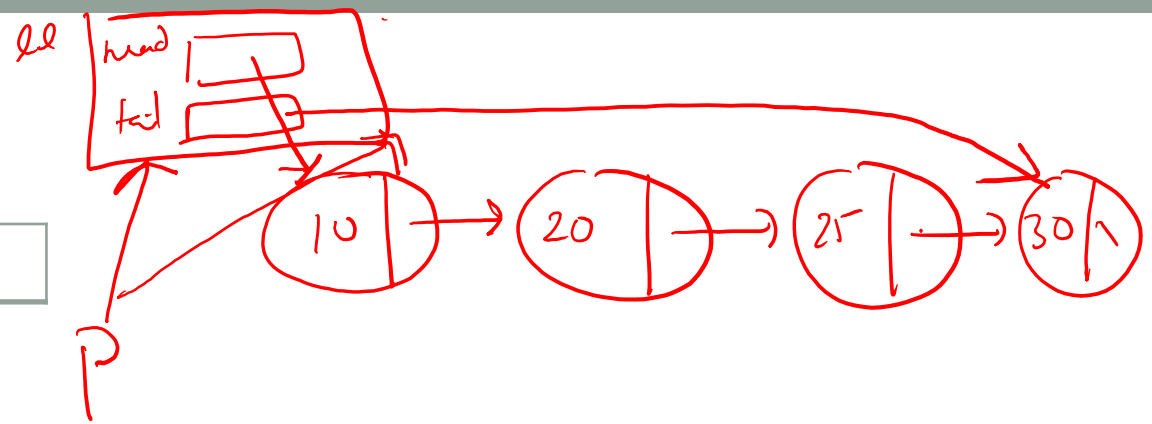
- Iterators are generalized pointers.
- Let's consider a very simple algorithm (printing in order) applied to a very simple data structure (sorted array)




```
void print_inorder(int* p, int size) {
    for(int i=0; i<size; i++) {
        std::cout << *p << std::endl;
        ++p;
    }
}
```

- We would like our print "algorithm" to also work with other data structures
- How should we modify it to print the elements of a LinkedList?

C++ Iterators



10	20	25	30	46	50	55	60

p

Consider our implementation of LinkedList

```
void print_inorder(LinkedList<int> *p, int size)  {
    for(int i=0; i<size; i++)
    {
        std::cout << *p <<std::endl;
        ++p;
    }
}
```

10
20
25
30

When will the above code work?

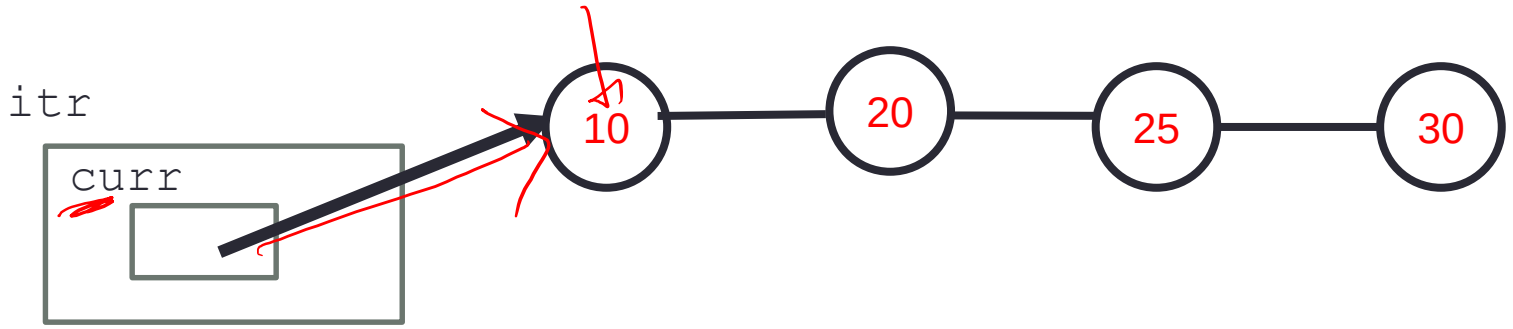
A. The operator "<<" is overloaded to print the data key of a LinkedList Node
B. The LinkedList class overloads the ++ operator
C. Both A and B
D. None of the above

(*p).head
p → head

C++ Iterators

- To solve this problem the LinkedList class has to supply to the client (print_inorder) with a generic pointer (an iterator object) which can be used by the client to access data in the container sequentially, without exposing the underlying details of the class

```
void print_inorder(LinkedList<int>& ll)   {
  LinkedList<int>::iterator itr = ll.begin();
  LinkedList<int>::iterator en = ll.end();

  while(itr!=en)
    {
        std::cout << *itr <<std::endl;
        ++itr;
    }
}
```

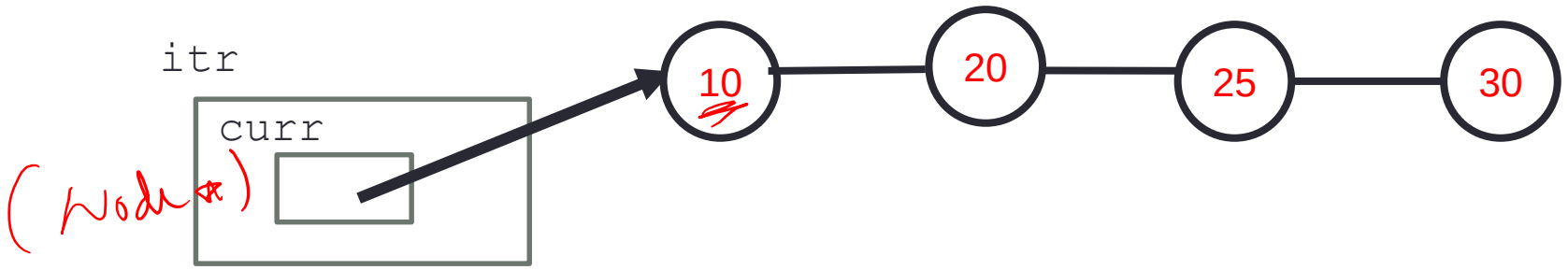


C++ Iterators

```
void print_inorder(LinkedList<int>& ll)  {
  LinkedList<int>::iterator itr = ll.begin();
  LinkedList<int>::iterator en = ll.end();

  while(itr!=en)
    {
        std::cout << *itr <<std::endl;
        ++itr;
    }
}
```

What should **begin()** return?

A. The address of the first node in the linked list container class
B. An iterator type object that contains the address of the first node
C. None of the above

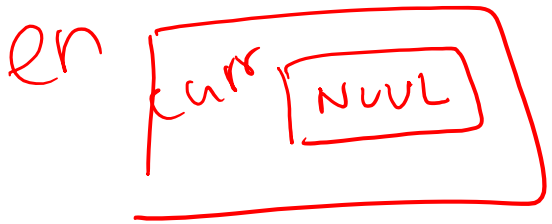


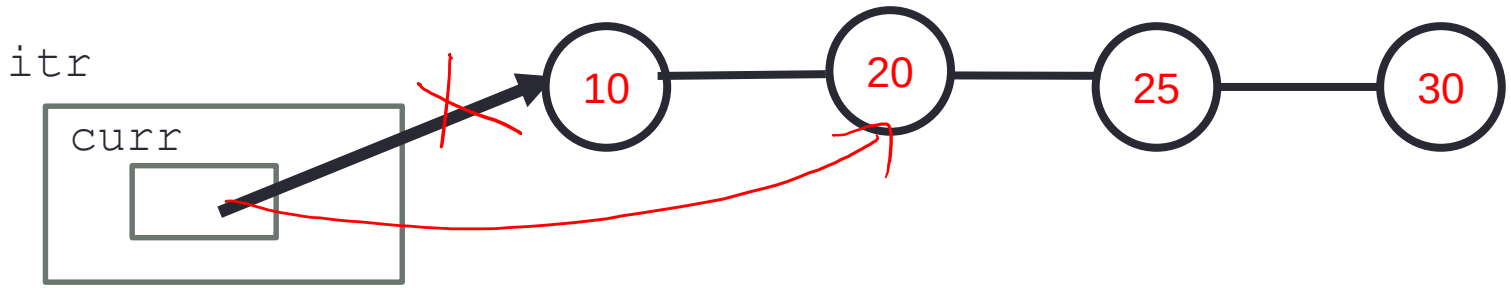
C++ Iterators

```
void print_inorder(LinkedList<int>& ll)  {
  LinkedList<int>::iterator itr = ll.begin();
  LinkedList<int>::iterator en = ll.end();

  while(itr!=en)
    {
        std::cout << *itr <<std::endl;
        ++itr;
    }
}
```

List the operators that the iterator has to implement?

A. *
B. ++
C. !=
D. All of the above
E. None of the above

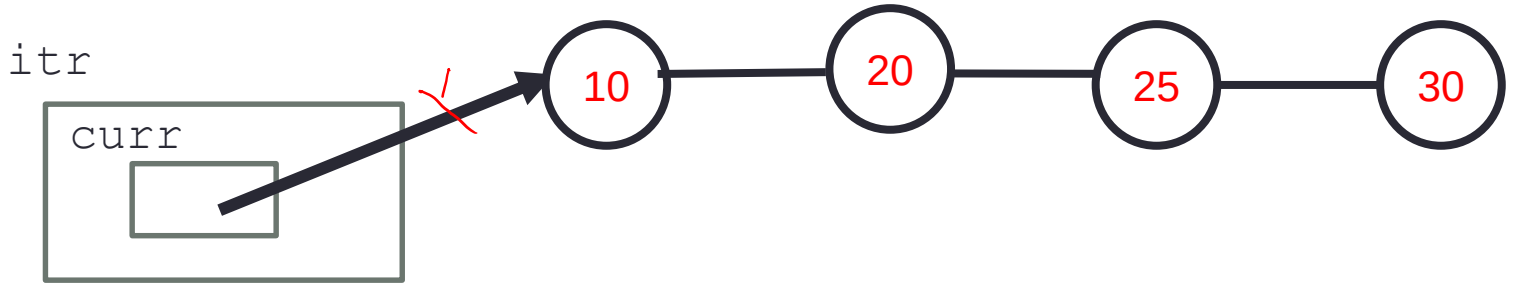




C++ Iterators

```
void print_inorder(LinkedList<int>& ll)  {
  LinkedList<int>::iterator itr = ll.begin();
  LinkedList<int>::iterator en = ll.end();

  while(itr!=en)
    {
        std::cout << *itr <<std::endl;
        ++itr;
    }
}
```

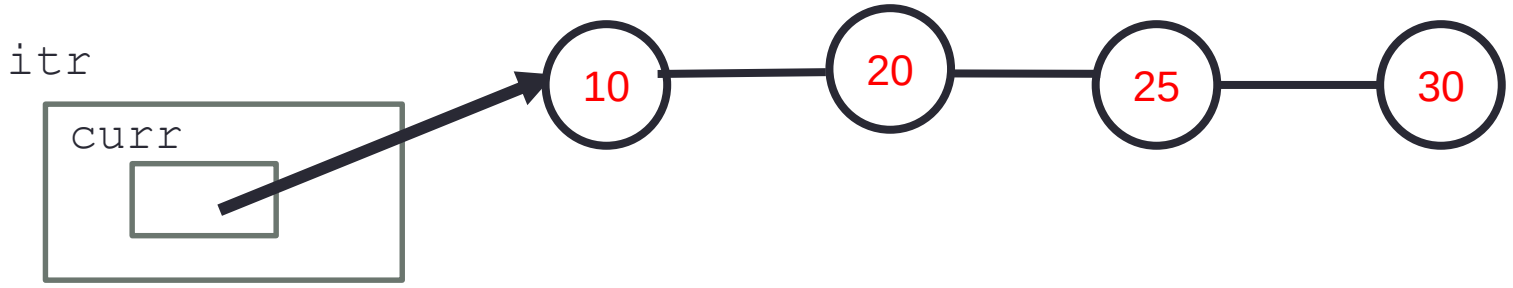
How should the diagram change as a result of the statement ++itr; ?



C++ Iterators

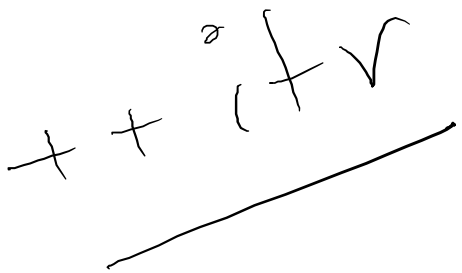
```
void print_inorder(LinkedList<int>& ll)   {
  auto itr = ll.begin();
  auto en = ll.end();

  while(itr!=en)
    {
        std::cout << *itr <<std::endl;
        ++itr;
    }
}
```

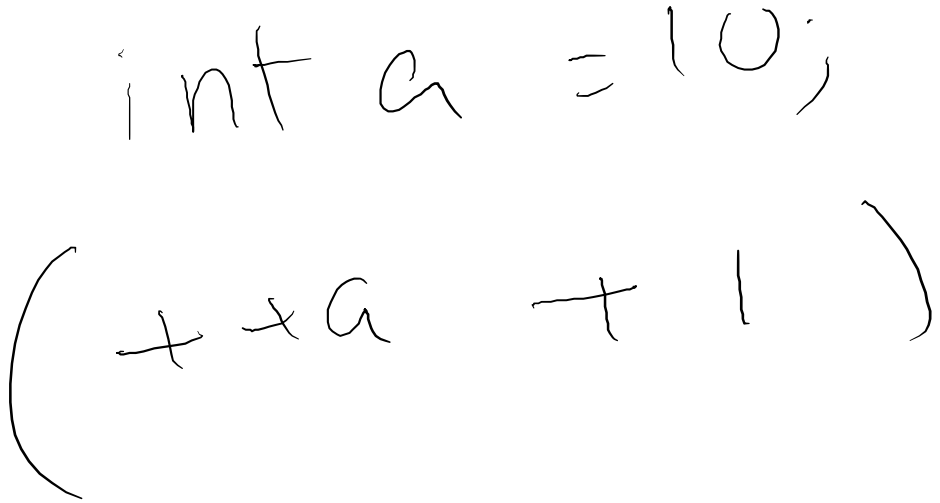
How should the diagram change as a result of the statement ++itr; ?

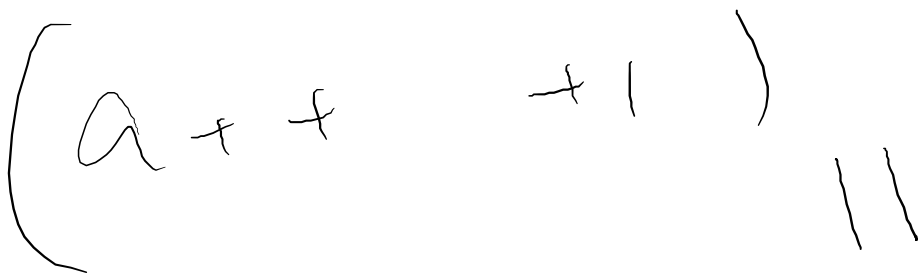


Demo

- Provide an iterator to the linkedList template class written in last lecture

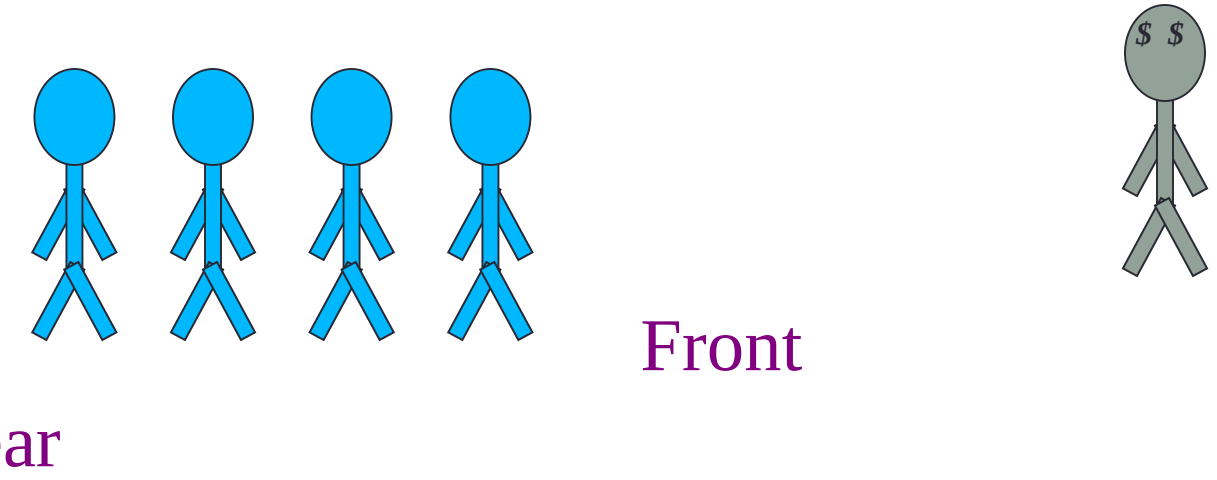






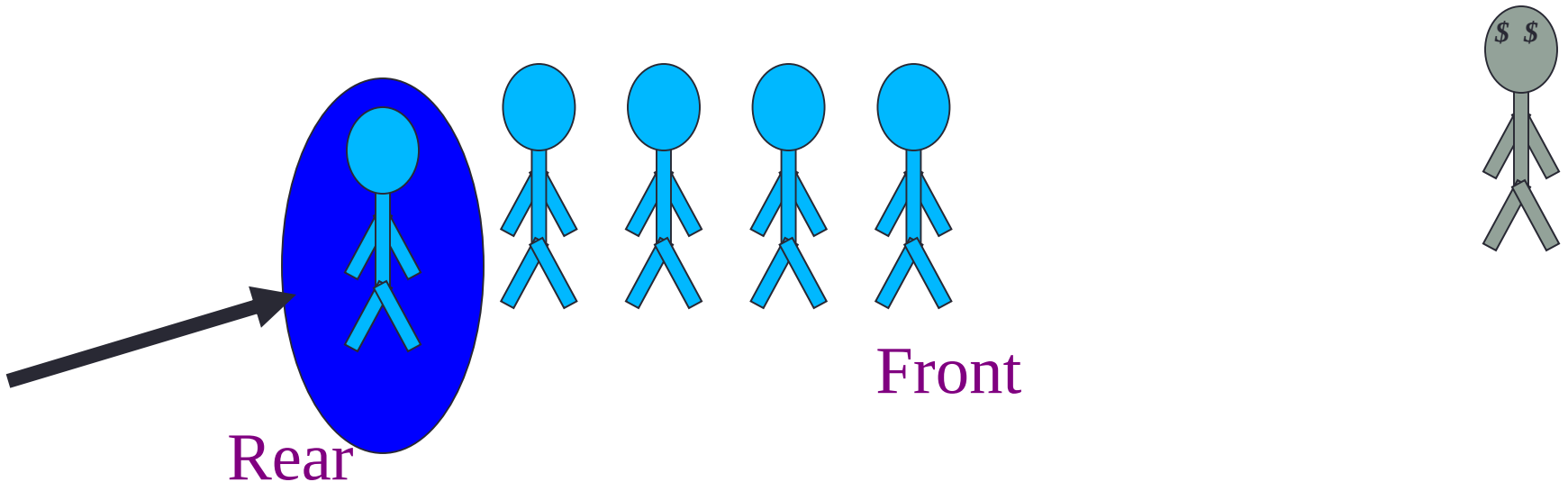
The Queue Operations

- A queue is like a line of people waiting for a bank teller. The queue has a **front** and a **rear**.



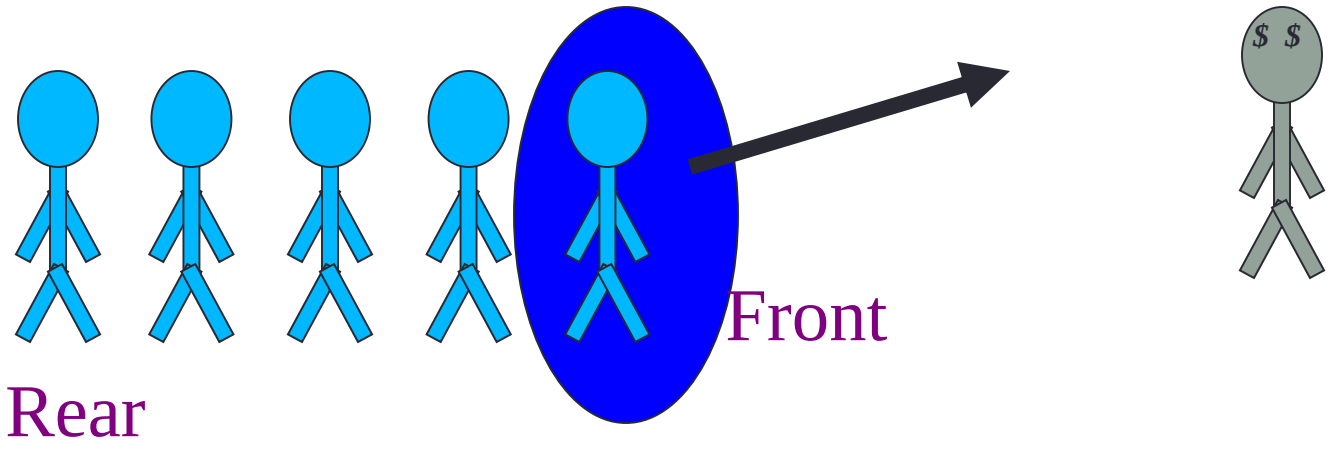
The Queue Operations

- New people must enter the queue at the rear. The C++ queue class calls this a **push**, although it is usually called an **enqueue** operation.



The Queue Operations

- When an item is taken from the queue, it always comes from the front. The C++ queue calls this a **pop**, although it is usually called a **dequeue** operation.



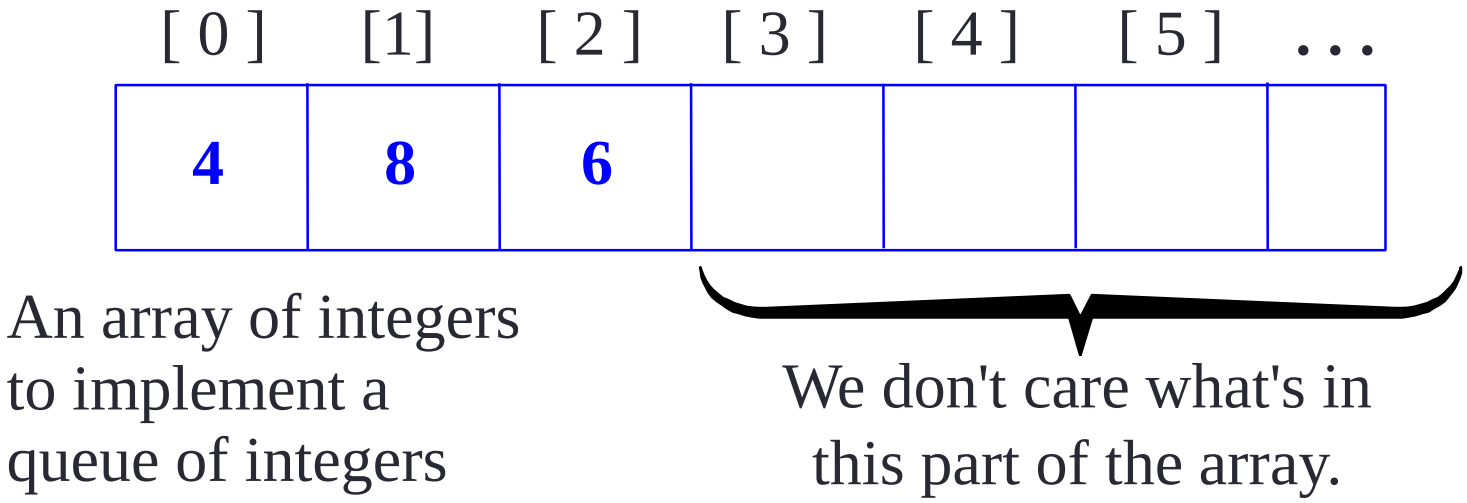
The Queue Class

- The C++ standard template library has a queue template class.
- The template parameter is the type of the items that can be put in the queue.

```
template <class Item>
class queue<Item>
{
public:
        queue( );
        void push(const Item& entry);
        void pop( );
        bool empty( ) const;
        Item front( ) const;
        …
```

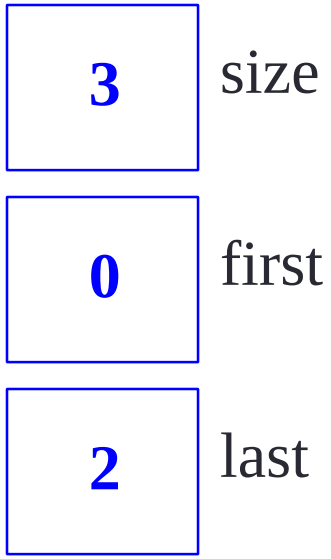
Array Implementation

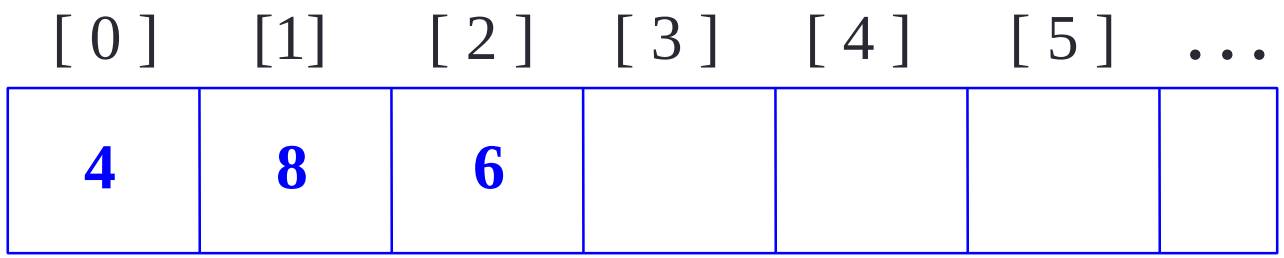
- A queue can be implemented with an array, as shown here. For example, this queue contains the integers 4 (at the front), 8 and 6 (at the rear).



Array Implementation

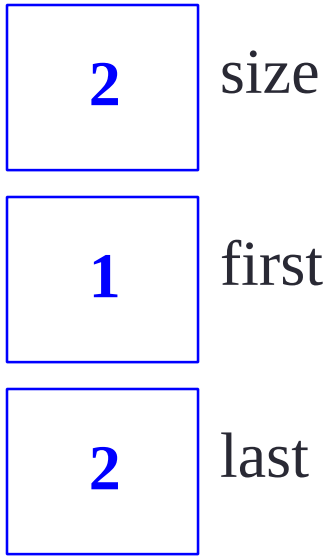
- The easiest implementation also keeps track of the number of items in the queue and the index of the first element (at the front of the queue), the last element (at the rear).

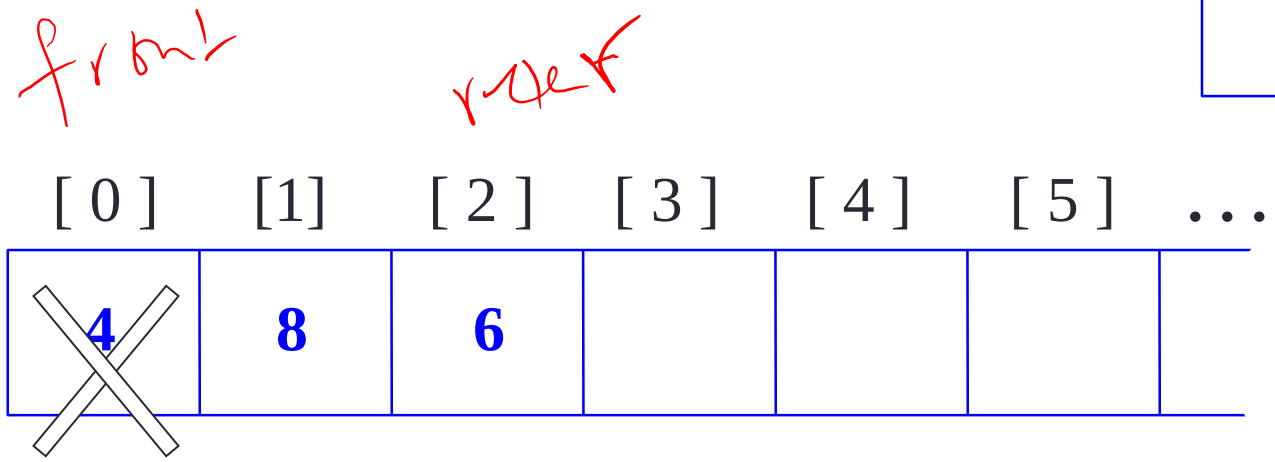




A Dequeue Operation

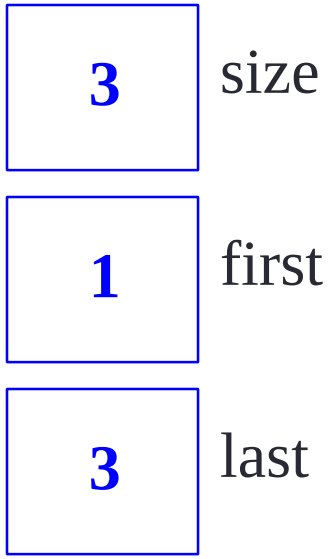
- When an element leaves the queue, size is decremented, and first changes, too.

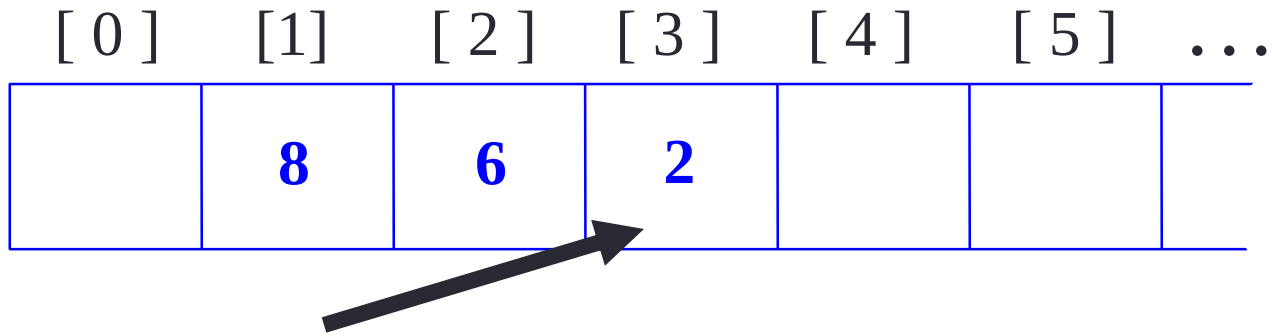




An Enqueue Operation

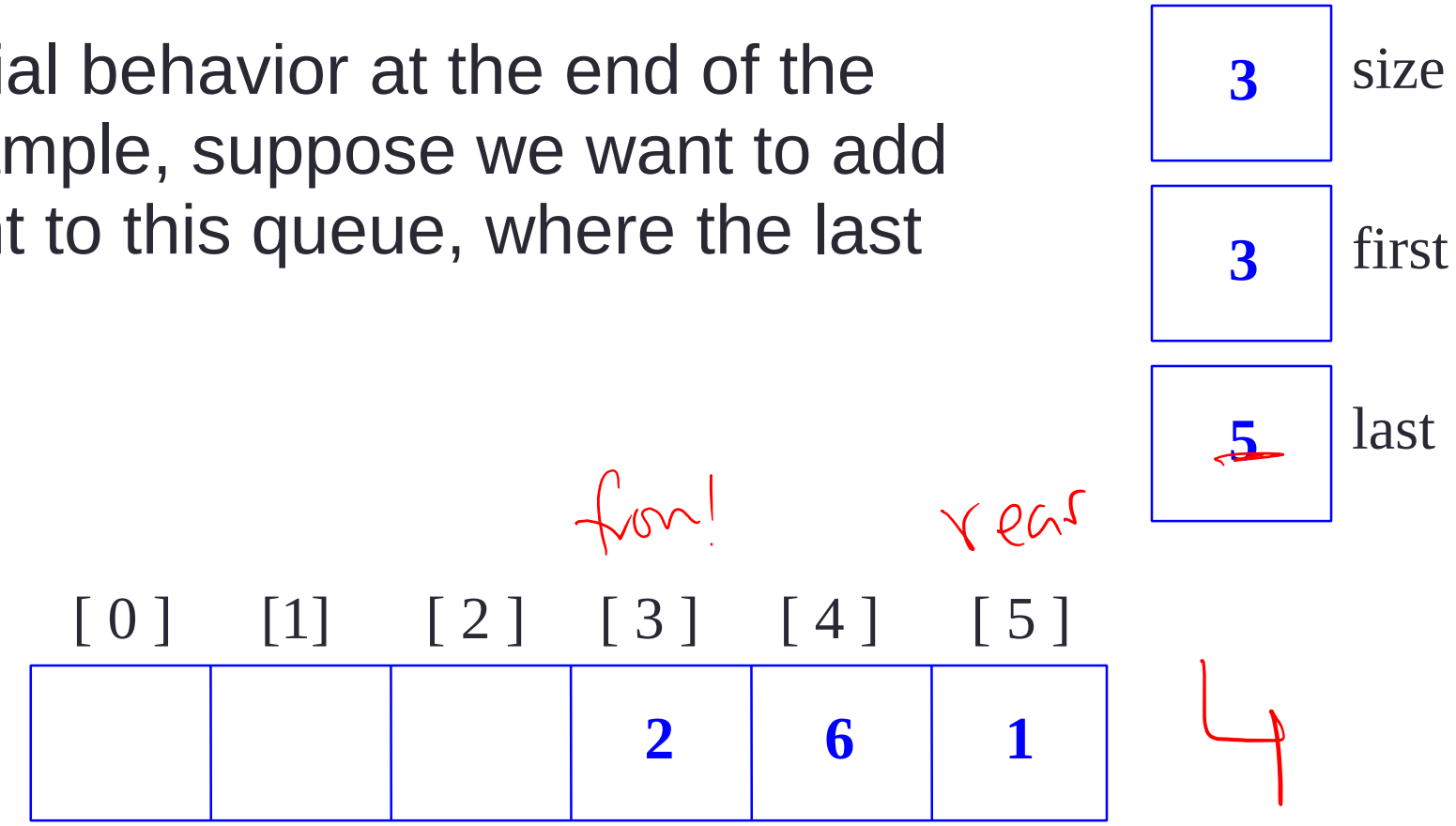
- When an element enters the queue, size is incremented, and last changes, too.





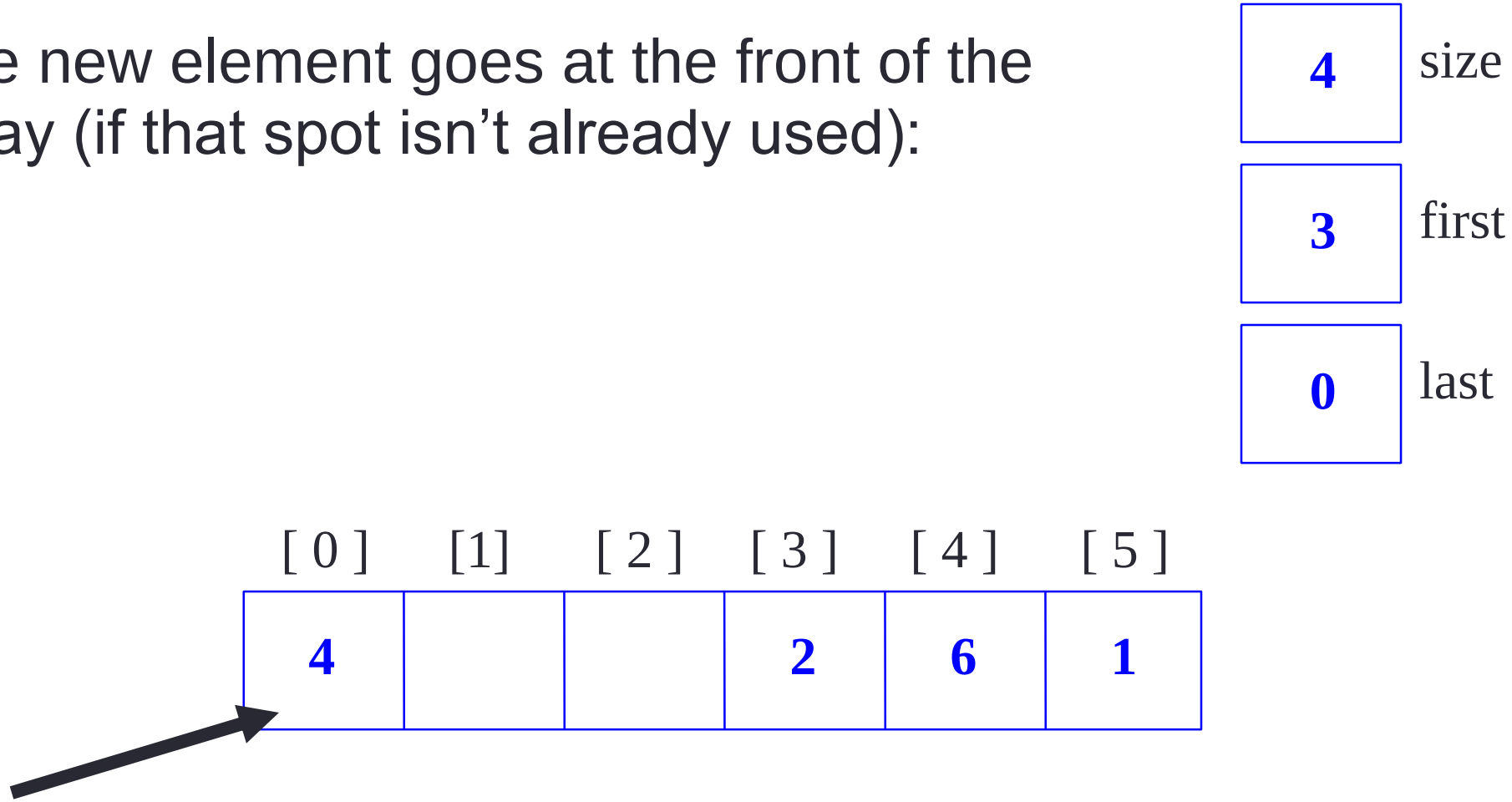
At the End of the Array

- There is special behavior at the end of the array. For example, suppose we want to add a new element to this queue, where the last index is [5]:



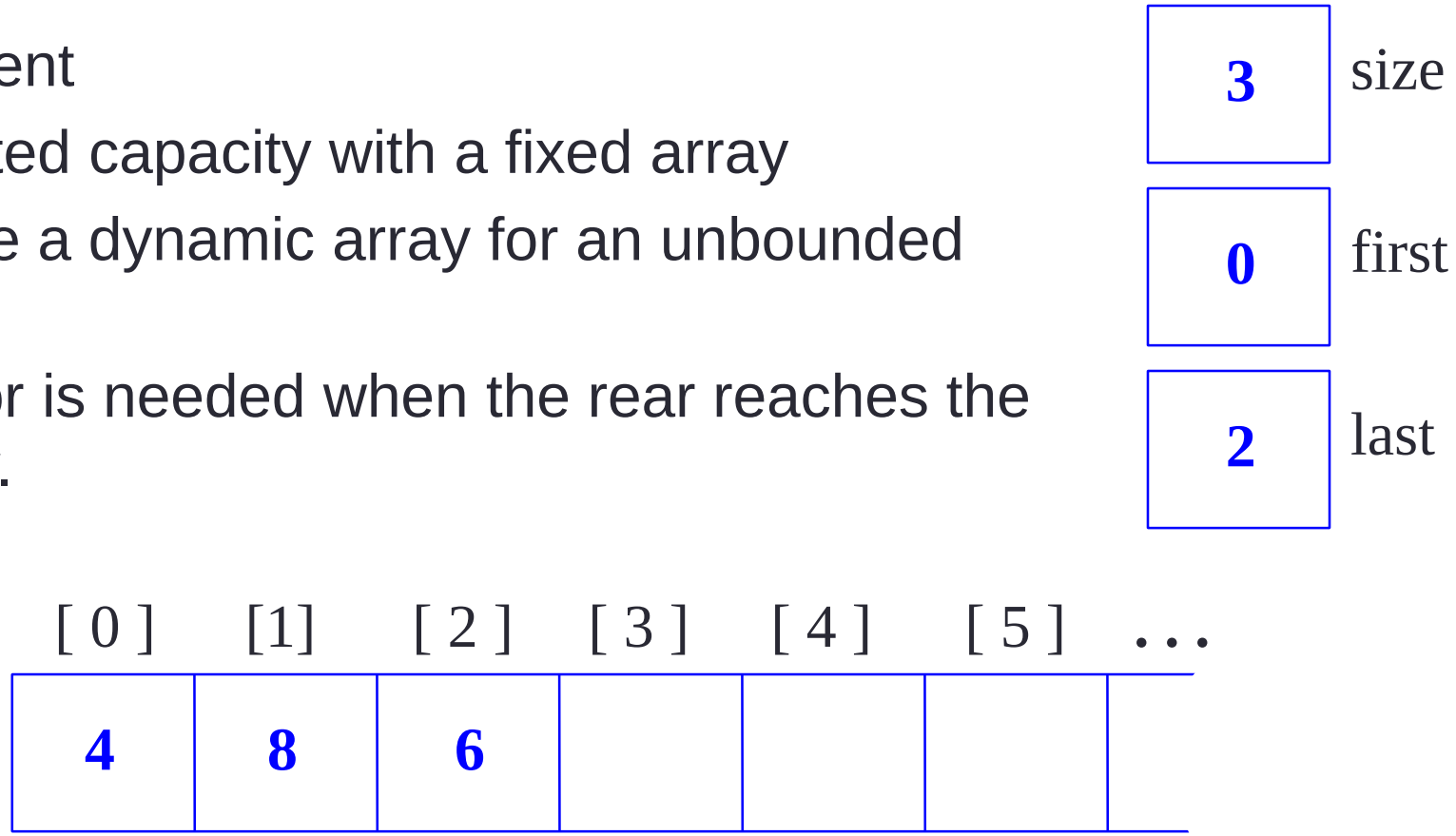
At the End of the Array

- The new element goes at the front of the array (if that spot isn't already used):



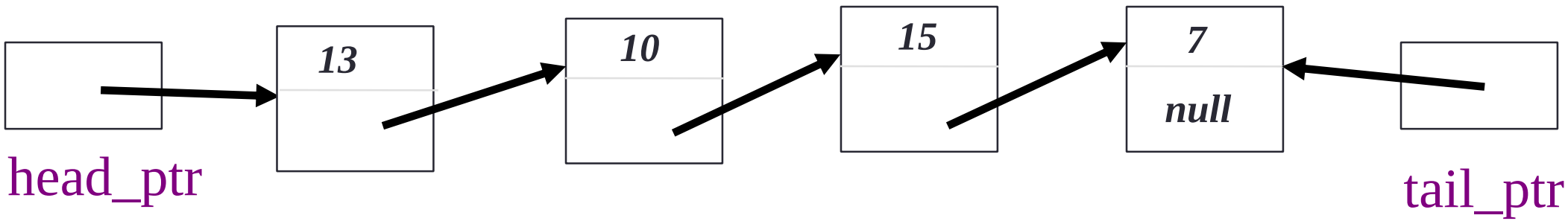
Array Implementation

- Easy to implement
- But it has a limited capacity with a fixed array
- Or you must use a dynamic array for an unbounded capacity
- Special behavior is needed when the rear reaches the end of the array.



Linked List Implementation

- A queue can also be implemented with a linked list with both a head and a tail pointer.

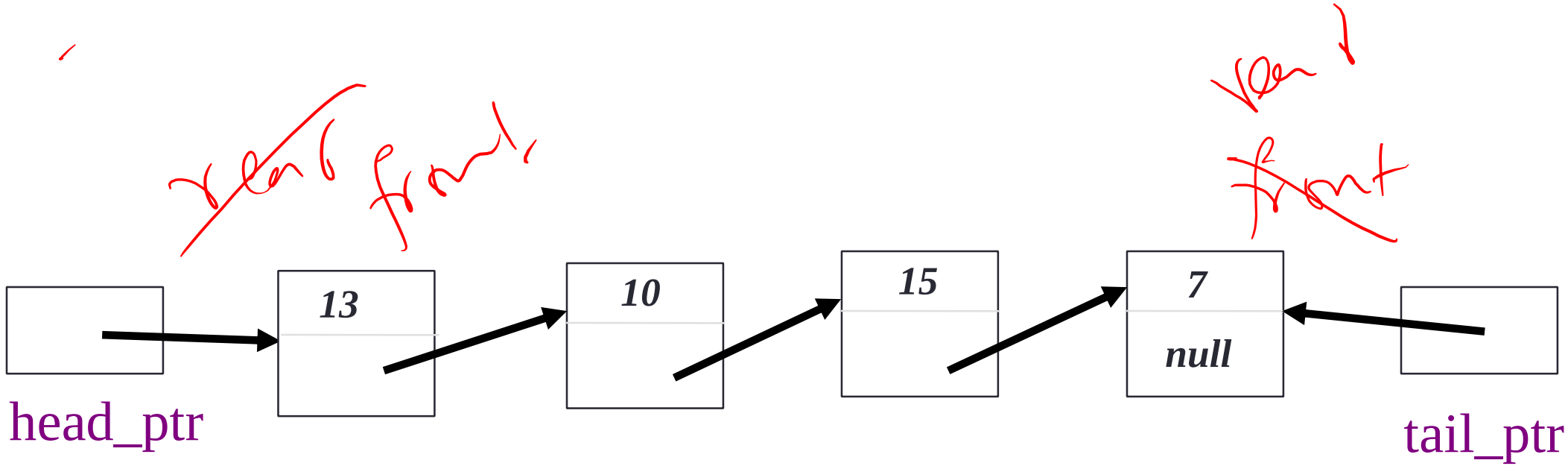


Linked List Implementation

- Which end do you think is the front of the queue? Why?

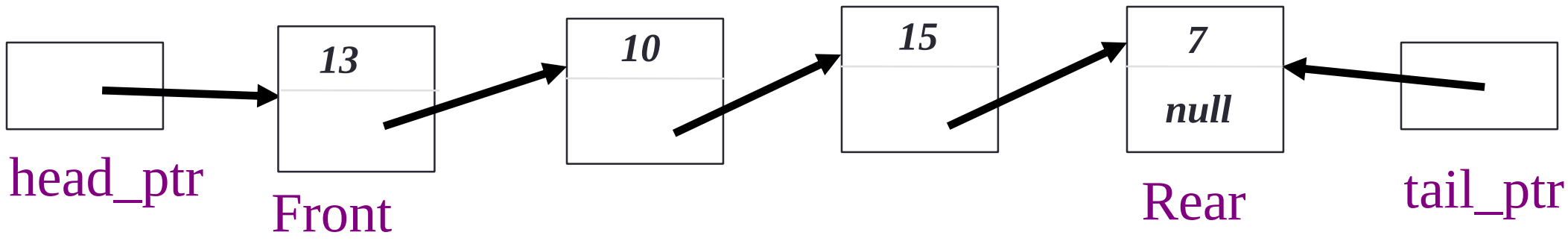
A. Node with value 13

B. Node with value 7



Linked List Implementation

- The head_ptr points to the front of the list.
- Because it is harder to remove items from the tail of the list.



Summary

- Like stacks, queues have many applications.
- Items enter a queue at the rear and leave a queue at the front.
- Queues can be implemented using an array or using a linked list.