Programming Abstractions

CS106B

Cynthia Bailey Lee
Julie Zelenski

Topics:

- `LinkedList` **Applications, Algorithms, and Variants**
  - Using a linked list for a queue
  - Tail pointers
  - The undo-enqueue operation
  - Doubly-linked lists
- Preview of our next topic: Binary Search Trees
  - Starting with a dream: binary search in a linked list?
  - How our dream provided the inspiration for the BST

> Fun fact: linked list algorithms are a classic technical job interview question category!

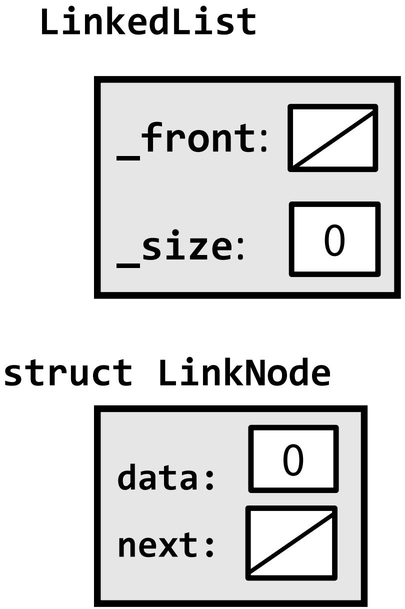
Queue implementation with a linked list

REAL-WORLD APPLICATION OF LINKED LISTS

linkedlist.h (for comparison—we will copy this design)

```
class LinkedList {
public:
    LinkedList();
    ~LinkedList();
    void add(int value);
    void clear();
    int get(int index) const;
    void insert(int index, int value);
    bool isEmpty() const;
    void remove(int index);
    void set(int index, int value);
    int size() const;

private:
    ListNode* _front;
    int       _size;
};
```



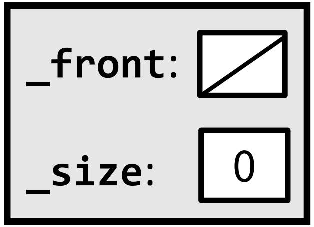
queueLL.h [Version 1]

```
class QueueLL {
public:
    QueueLL();
    ~QueueLL();
    void enqueue(int value);
    void clear();
    int dequeue(int index);
    int peek(int index) const;
    bool isEmpty() const;
    int size() const;

private:
    ListNode* _front;
    int       _size;
};
```

Internal structure is exactly the same as `LinkedList` class.

`QueueLL`

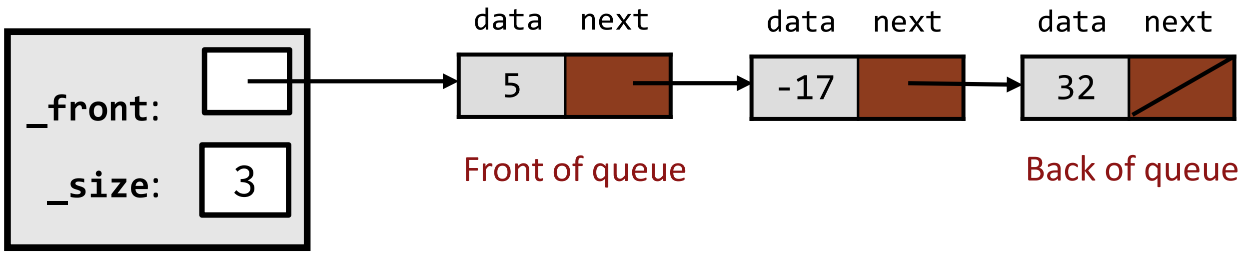


Public-facing methods are renamed and curated to provide the usual queue interface.

`struct LinkNode`

Queue implemented with a linked list

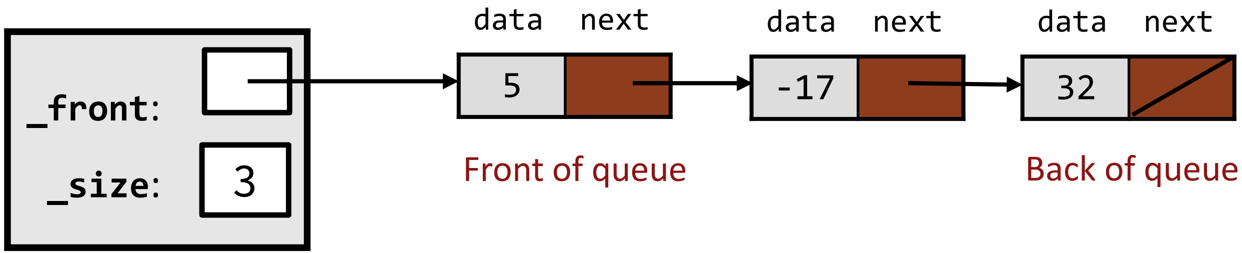
- Front of the list is the front of the queue
  - Need to dequeue from here
  - No problem! Unlike array O(N), removing from the front of a linked list is just O(1)
- Back of the list is the back of the queue
  - Need to enqueue to here
  - Hmmm…not good. O(N) because we have to traverse in a loop to the end of the list



Queue implemented with a linked list

- Front of the list is the front of the queue
  - Need to dequeue from here
  - No problem! Unlike array O(N), removing from list is just O(1)
- Back of the list is the back of the queue
  - Need to enqueue to here
  - Hmmm…not good. O(N) because we have to traverse in a loop to the end of the list

Key insight: actual add is O(1), it's just getting there that takes a long time.

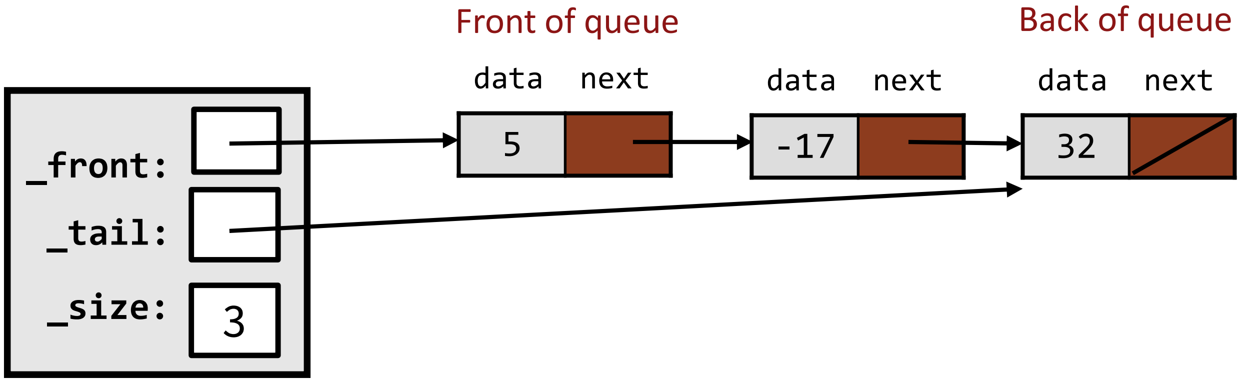


Tail Pointers

BONUS FEATURE TO IMPROVE LINKED LIST PERFORMANCE FOR APPLICATIONS LIKE QUEUE

Queue implemented with a linked list <u>with Tail Pointer</u>

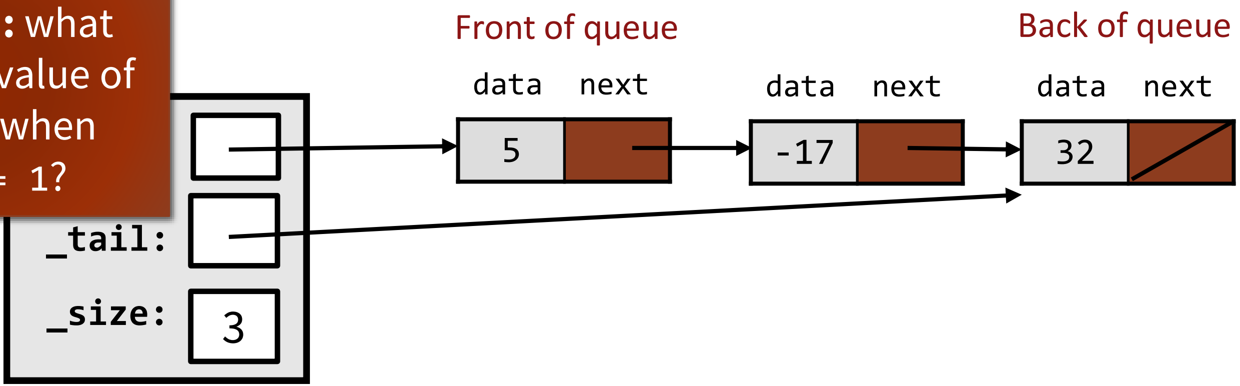
- We add a third private member variable to our `LinkedList` class
  - `_front` enables dequeue in O(1)
  - `_tail` enables enqueue in O(1)
  - (`_size` stays the same)
  - When `_size = 0`, `_front` and `_tail` will be both be `nullptr`



Queue implemented with a linked list <u>with Tail Pointer</u>

- We add a third private member variable to our `LinkedList` class
  - `_front` enables dequeue in O(1)
  - `_tail` enables enqueue in O(1)
  - (`_size` stays the same)
  - When `_size = 0`, `_front` and `_tail` will be both be `nullptr`

Your Turn: what should the value of `_tail` be when `_size = 1`?

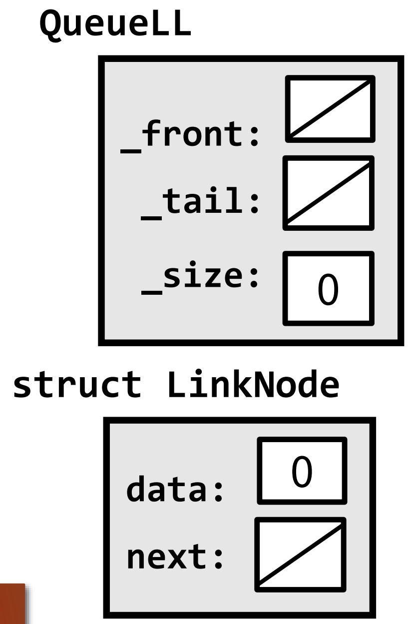


queueLL.h [Version 2]

```
class QueueLL {
public:
    QueueLL();
    ~QueueLL();
    void enqueue(int value);
    void clear();
    int dequeue(int index);
    int peek(int index) const;
    bool isEmpty() const;
    int size() const;

private:
    ListNode* _front;
    ListNode* _tail;
    int       _size;
};
```

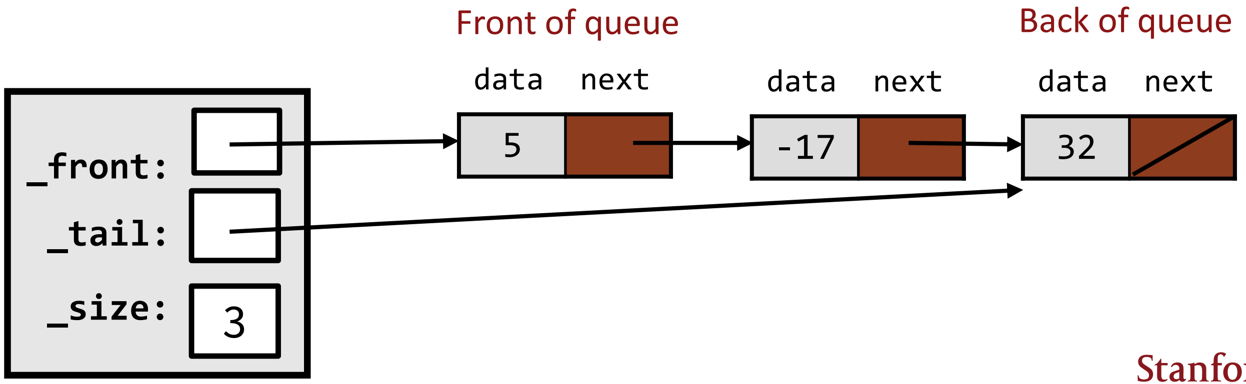
New tail pointer member variable.



Implementing enqueue

```
// Appends the given value to the end of the list.
void QueueLL::enqueue(int value) {
    ...
}
```

- What pointer(s) must be changed to add a node to the **end** of a list?
- What different cases must we consider?



Code for list add() compared to code for enqueue()

```
// (in linkedlist.cpp)
void LinkedList::add(int value)
{
    if (_front == nullptr) {
        // adding to an empty list
        _front = new ListNode(value);
    } else {
        // adding to the end of an existing list
        ListNode* current = _front;
        while (current->next != nullptr) {
            current = current->next;
        }
        current->next = new ListNode(value);
    }
    _size++;
}
```

```
// (in queueLL.cpp)
void QueueLL::enqueue(int value)
{
    if (_front == nullptr) {
        // adding to an empty list
        _front = new ListNode(value);
        _tail = _front;
    } else {
        // adding to the end of an existing list
        _tail->next = new ListNode(value);
        _tail = _tail->next;
    }
    _size++;
}
```

Code for list add() compared to code for enqueue()

```
// (in linkedlist.cpp)
void LinkedList::add(int value)
{
    if (_front == nullptr) {
        // adding to an empty list
        _front = new ListNode(value);
    } else {
        // adding to the end of an existing list
        ListNode* current = _front;
        while (current->next != nullptr) {
            current = current->next;
        }
        current->next = new ListNode(value);
    }
    _size++;
}
```

```
// (in queueLL.cpp)
void QueueLL::enqueue(int value)
{
    if (_front == nullptr) {
        // adding to an empty list
        _front = new ListNode(value);
        _tail = _front;
    } else {
        // adding to the end of an existing list
        _tail->next = new ListNode(value);
        _tail = _tail->next;
    }
    _size++;
}
```

Don't need the loop anymore—just go straight to using the tail pointer.

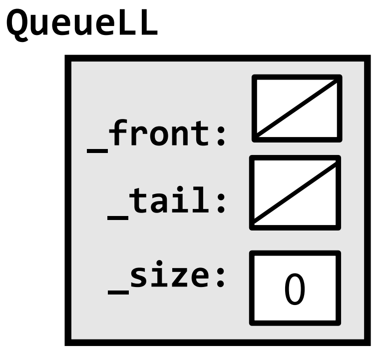
Implementing an undo-enqueue operation

FOR THOSE “NEVERMIND, THIS RAMEN NAGI LINE IS TO LONG, I’LL GO TO A DIFFERENT RESTAURANT!” MOMENTS

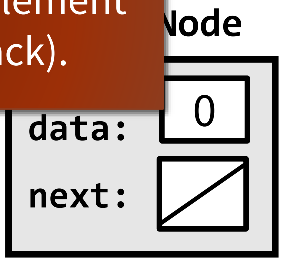
queueLL.h [Version 3]

```
class QueueLL {
public:
    QueueLL();
    ~QueueLL();
    void enqueue(int value);
    void clear();
    int dequeue(int index);
    int peek(int index) const;
    bool isEmpty() const;
    int size() const;
    void undoEnqueue();

private:
    ListNode* _front;
    ListNode* _tail;
    int       _size;
};
```



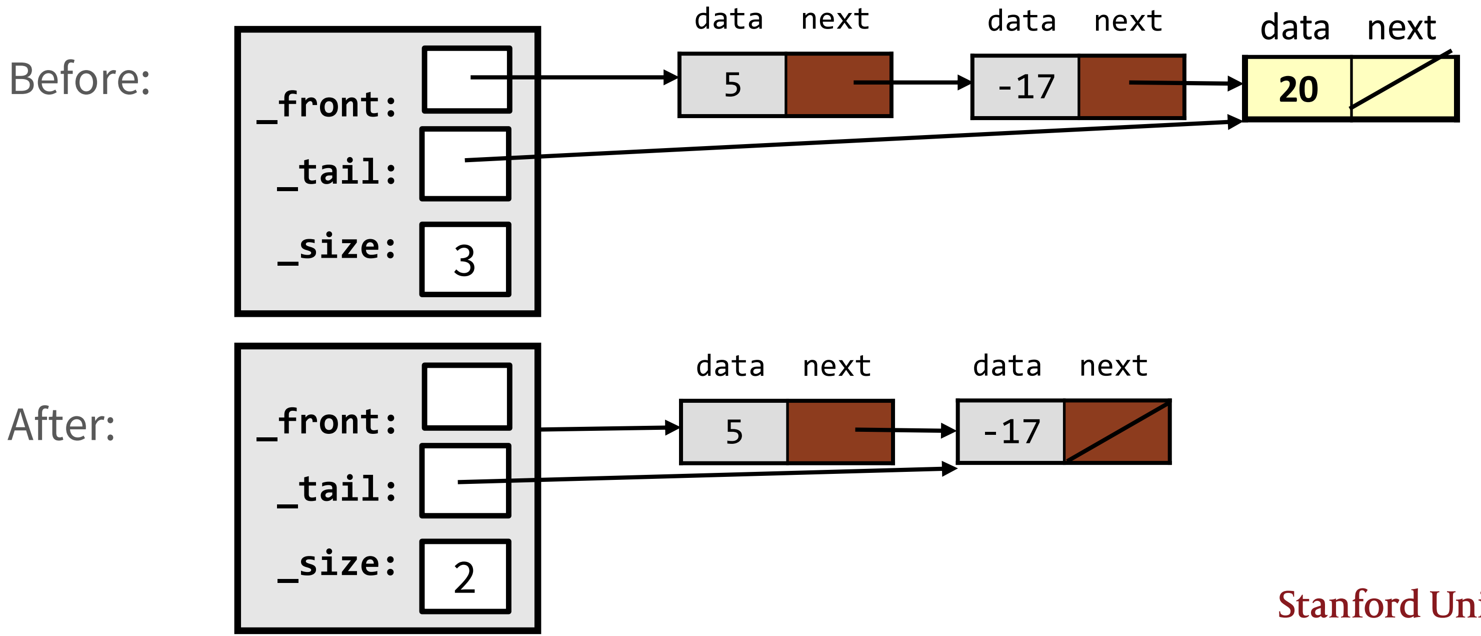
This function would remove the <u>most-recently-enqeued</u> element (similar to pop in a stack).



Implementing a prepend operation

```
void QueueLL::undoEnqueue() {
    ...
}
```

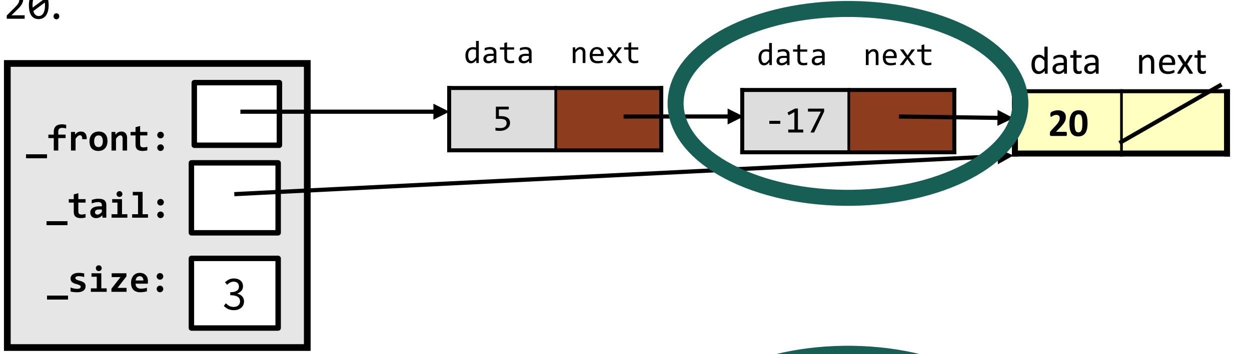
- Removes the most-recently-enqueued item.



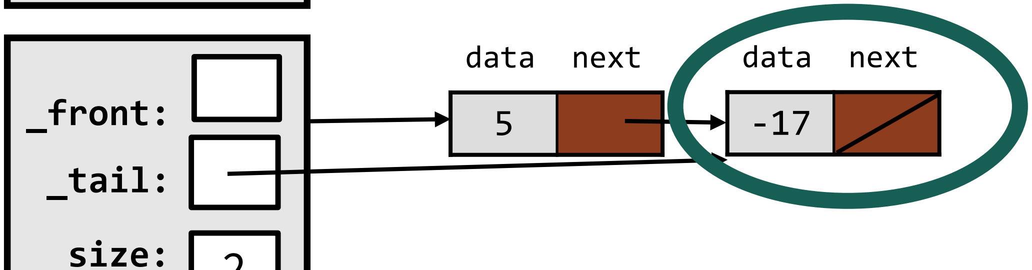
Options for implementing a prepend operation

- Could just copy our code from `LinkedList remove(index)`, with `index` set to `size() - 1`, but this is O(N).
  - It's disheartening to see that our new `_tail` pointer doesn't help us. ☹
- That's because the node whose `next` pointer needs to change is the one with `-17`, not `20`.

Before:



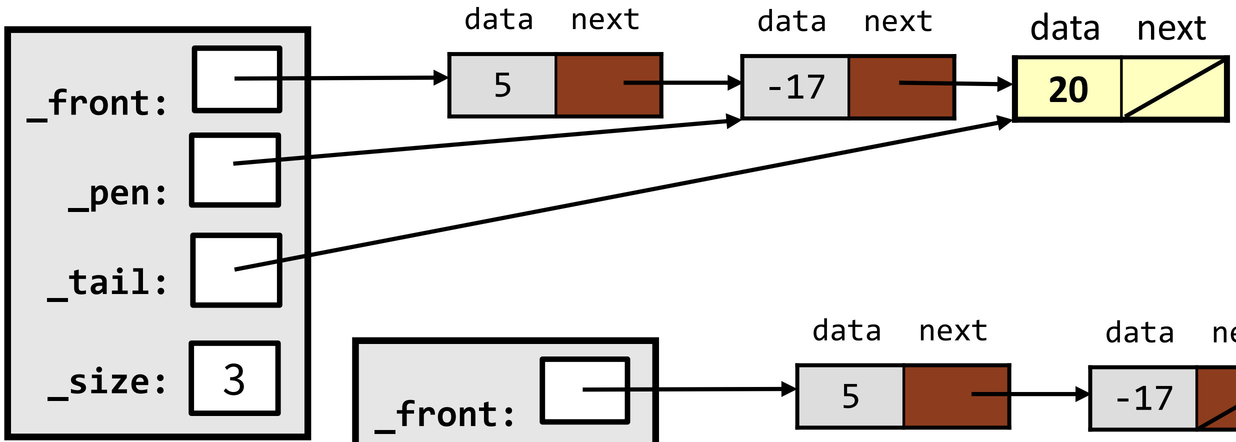
After:



More options for implementing a prepend operation?

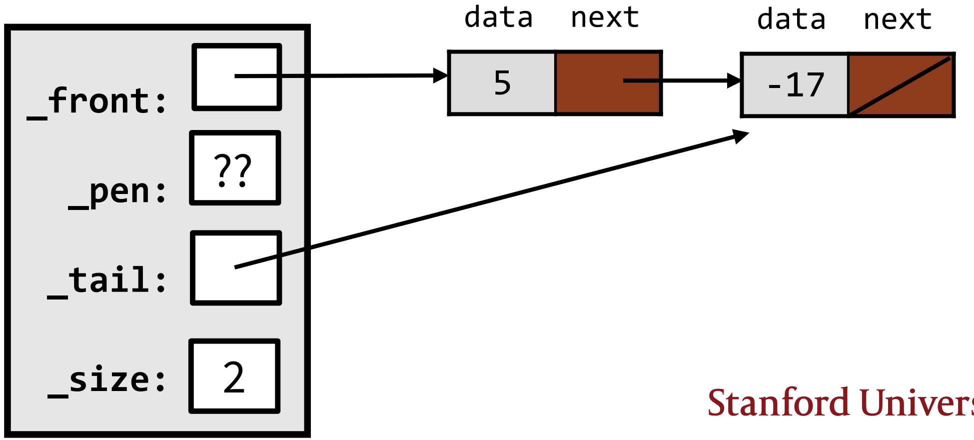
- What if we add a penultimate-node pointer to our member variables?
  - It will point to the second-to-last element in the list.

Before:



After: our _pen pointer helps us get this far…

…but what about the update to _pen?



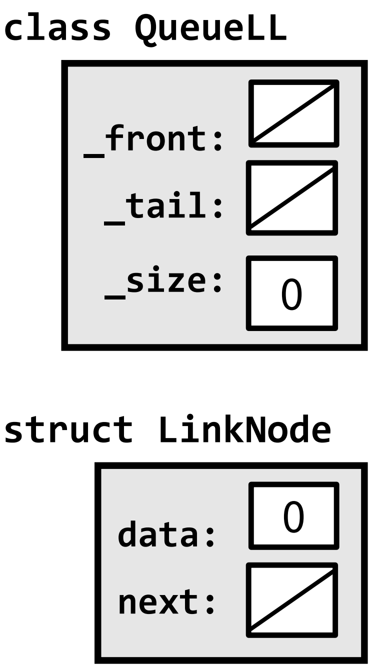
The Doubly-Linked List structure

ANOTHER VERY COMMON BONUS FEATURE TO IMPROVE LINKED-LIST PERFORMANCE

queueLL.h [Version 3, again]

```
class QueueLL {
public:
    QueueLL();
    ~QueueLL();
    void enqueue(int value);
    void clear();
    int dequeue(int index);
    int peek(int index) const;
    bool isEmpty() const;
    int size() const;
    void undoEnqueue();

private:
    ListNode* _front;
    ListNode* _tail;
    int       _size;
};
```

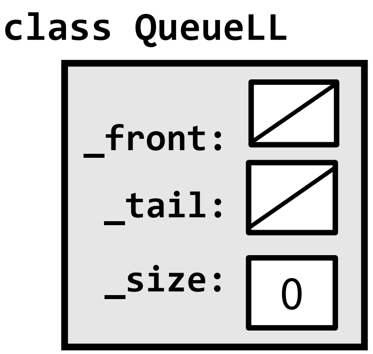


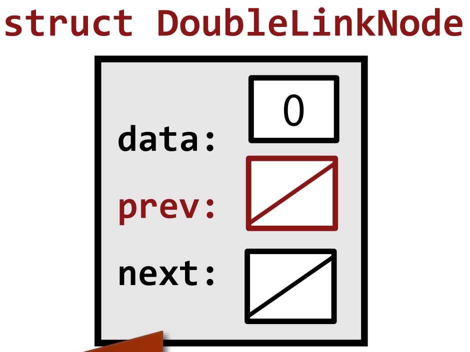
This time, instead of changing our list class, let's reconsider the node struct that we've been using all this time.

queueLL.h [Version 4]

```
class QueueLL {
public:
    QueueLL();
    ~QueueLL();
    void enqueue(int value);
    void clear();
    int dequeue(int index);
    int peek(int index) const;
    bool isEmpty() const;
    int size() const;
    void undoEnqueue();

private:
    ListNode* _front;
    ListNode* _tail;
    int       _size;
};
```

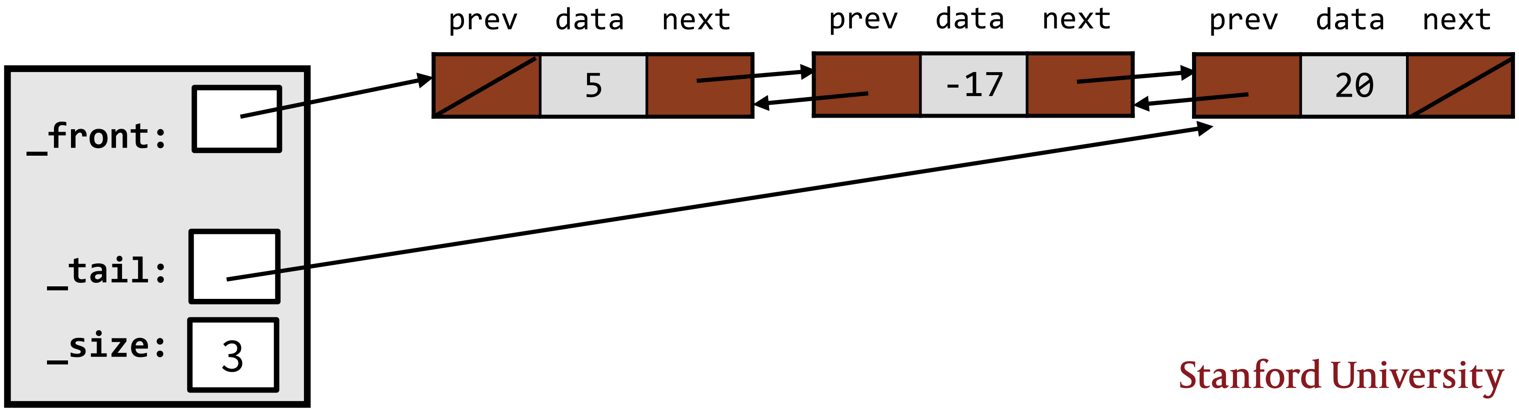




Now each node will have two pointers: a previous and a next.

Doubly-Linked List

- Benefits:
  - Easy access to nodes before your node, when needed for edits
- Drawbacks:
  - Linked list already roughly doubles amount of storage needed to hold our data (compared to array), now doubly-linked list triples it
  - More work in every add, remove, insert, etc operation to maintain correct pointer placements



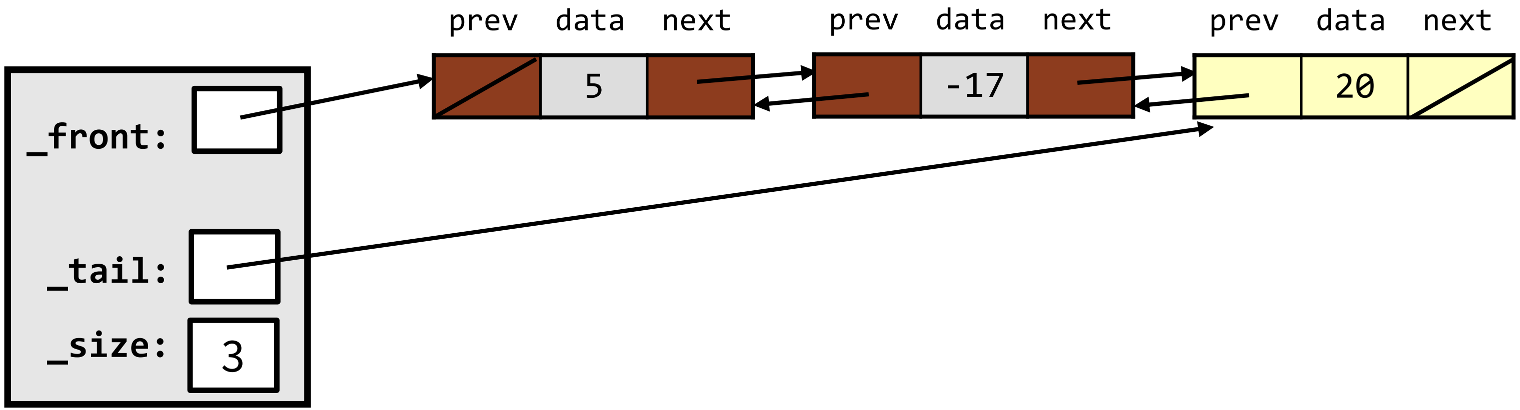
Implementing an undo-enqueue operation (now lets do it)

FOR THOSE "NEVERMIND, THIS RAMEN NAGI LINE IS TO LONG, I'LL GO TO A DIFFERENT RESTAURANT!" MOMENTS

Implementing a prepend operation

```
void QueueLL::undoEnqueue() {
   ...
}
```

- What pointer(s) must be changed to remove the node at the the **end** of a list?
- What different cases must we consider?



Implementing a prepend operation

```
void QueueLL::undoEnqueue() {
    if (size() == 0) {
        error("Cannot remove from empty queue!");
    }

    DoubleLinkNode* trash = _tail;
    if (size() == 1) {
        _tail = _front = nullptr;
    } else {
        _tail->prev->next = nullptr;
        _tail = _tail->prev;
    }
    delete trash;
    _size--;
}
```

SWITCHING GEARS!

Preview of our next topic: Binary Search Tree

Binary Search in a Linked List?

EXPLORING A GOOD IDEA, FINDING WAY TO MAKE IT WORK

Recall our beautiful algorithm: binary search!

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

- How long does it take us to find data in a sorted **array**?
  - **Use binary search!**
  - **O(logn):** awesome!!

Q. Can we do binary search on a linked list?

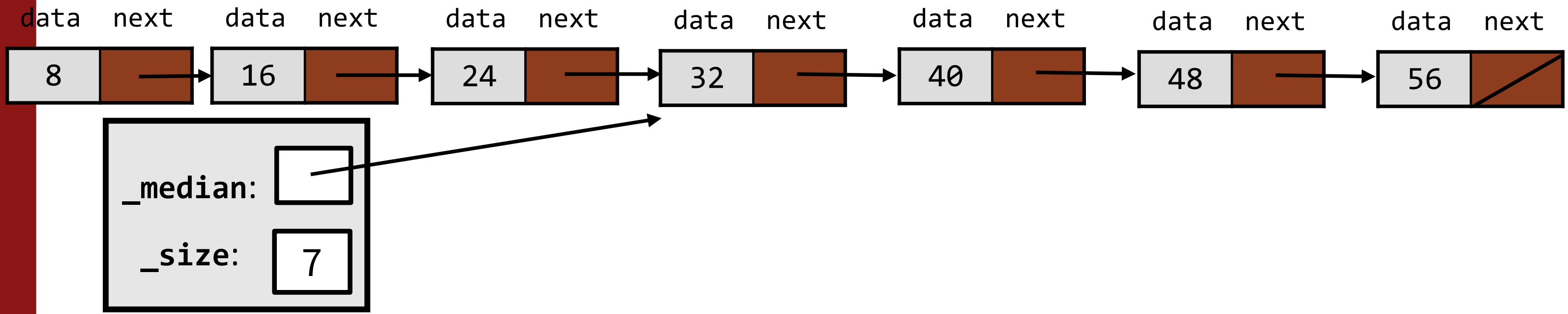
A. No.

- The nodes are spread all over memory, and we must follow “next” pointers one at a time to navigate (the treasure hunt).
- **Therefore cannot jump right to the middle.**
- **Therefore cannot do binary search.**
- **Find is O(N):** not terrible, but pretty bad compared to O(logn) or O(1)

Let’s brainstorm a wild idea and then see if we can make it work

"What if…?"
The inspiration for Binary Search Trees

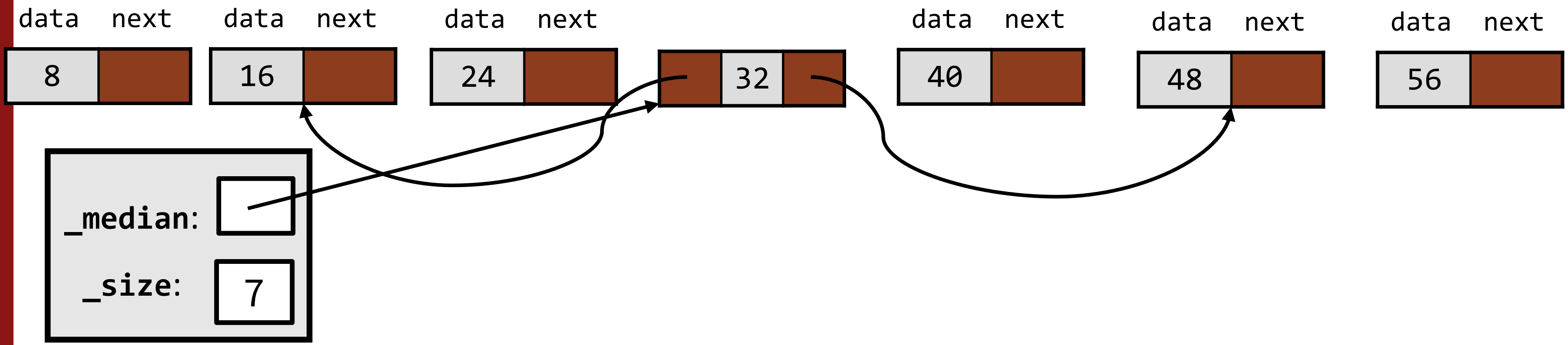
- What if…
- …instead of having a _front pointer in our linked list, we had a pointer to the element we want to look at first in binary search: the exact median/middle element?



- That would make the first step of our binary search **really** fast/easy!
- What about the next step? (and the front half of our list, lol)

"What if…?"
The inspiration for Binary Search Trees

- What about the next step? (and the front half of our list, lol)
- Well, we could have the middle element point to the middle element of both the left half and the right half, so the 2nd step of our binary search is easy/fast too!



- Keep doing this until all elements have pointers to the middle of what remains to their left/right sides…voila!

An Idealized Binary Search Tree

- Our class will have a pointer to the median element*, and each element has pointers to the medians of everything to their left and right
  - ** actually it's hard to guarantee it will be the <u>exact</u> middle element, more on this, and lots more about Binary Search Trees, next time!*