Programming Abstractions

CS106B

Cynthia Bailey Lee
Julie Zelenski
Topics:

- **LinkedList: Further 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;
};
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;
};
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
Tail Pointers

**BONUS FEATURE TO IMPROVE LINKED LIST PERFORMANCE FOR APPLICATIONS LIKE QUEUE**
Queue implemented with a linked list with Tail Pointer

- 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 implementation diagram](image)
Queue implemented with a linked list with Tail Pointer

- 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: describe what should the value of `_tail` should be when `_size = 1`. 

Stanford University
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++;
}

Your Turn: What are the Big-O costs of `add()` and `enqueue()`, respectively?
Implementing an undo-enqueue operation

FOR THOSE “NEVERMIND, THIS RAMEN NAGI LINE IS TO LONG, I’LL GO TO A DIFFERENT RESTAURANT!” MOMENTS
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 most-recently-enqueue element (similar to pop in a stack).
Implementing an undoEnqueue operation

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

- Removes the most-recently-enqueued item.

Before:

![Diagram showing the queue structure before the undoEnqueue operation](image)
Implementing an undoEnqueue operation

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

- Removes the most-recently-enqueued item.

**Before:**

- `_front`: 3
- `_tail`: 3
- `_size`: 3

```
<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>-17</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>
```

**After:**

- `_front`: 2
- `_tail`: 2
- `_size`: 2

```
<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>-17</td>
<td></td>
</tr>
</tbody>
</table>
```

**Your Turn:** What is the best big-O cost we could achieve for undoEnqueue, with our current class design?
Options for implementing an undoEnqueue operation

- The node whose `next` pointer needs to change is the one with -17, not 20.
  - Our new `_tail` pointer doesn’t help us. 😞
- Still possible! But we have to loop from `_front` to reach the penultimate node.
  - But this is O(N) 😞

Before:

<table>
<thead>
<tr>
<th>_front:</th>
<th>_tail:</th>
<th>_size:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>-17</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

After:

<table>
<thead>
<tr>
<th>_front:</th>
<th>_tail:</th>
<th>_size:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>-17</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>
More options for implementing an undoEnqueue 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;
};

class QueueLL
    _front:
    _tail:
    _size: 0

struct LinkNode
data: 0
next:

This time, instead of changing our list class, let’s reconsider the node struct that we’ve been using all this time.
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:
    DoubleLinkNode* _front;
    DoubleLinkNode* _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 TOO LONG, I’LL GO TO A DIFFERENT RESTAURANT!”
MOMENTS
Implementing an undoEnqueue operation

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

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

- How long does it take us to find data in a sorted array?
  - Use binary search!
  - $O(\log n)$: 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!
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 exact middle element, more on this, and lots more about Binary Search Trees, next time!