Programming Abstractions

CS106B

Cynthia Bailey Lee
Julie Zelenski
Topics:

- **Wednesday: Link Nodes**
  - The LinkNode struct
  - Chains of link nodes
  - LinkNode operations

- **Today: Link Lists**
  - Providing a cohesive interface to chains of link nodes with a LinkedList class
  - LinkedList class implementation
  - LinkedList methods
Linked Nodes

A GREAT WAY TO EXERCISE YOUR POINTER UNDERSTANDING
FIRST RULE OF LINKED NODE/LISTS CLUB:

DRAW A PICTURE OF LINKED LISTS

Do no attempt to code linked nodes/lists without pictures!
LinkNode Summary!

```c
struct LinkNode {
    int data;
    LinkNode* next;
};
```

LinkNode* heapNode = new LinkNode;
heapNode->data = 6;
heapNode->next = nullptr;

Stack

Heap

heapNode

6
Linked List Data Structure

PUTTING THE LISTNODE TO USE
Let's write a collection class named LinkedList.

- Has the same public members as Vector
  - add, clear, get, insert, isEmpty, remove, size, toString

- The list is internally implemented as a **chain of linked nodes**
  - The LinkedList keeps a pointer to its _front node as a field
  - nullptr is the end of the list; a nullptr in _front signifies an empty list

```cpp
class LinkedList

_private:
  _size: int
  _front: ListNode*

public:
  ListNode* root;

  int size() {
    return _size;
  }

  void add(int value) {
    // add a new node
  }

  void insert(int index, int value) {
    // insert a new node
  }

  void remove(int index) {
    // remove node at index
  }

  string toString() {
    // return a string representation
  }
```

![Diagram showing a linked list structure]
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;
    string toString() const;

private:
    ListNode* _front;
    int _size;
};
Our first LinkedList Class
Method

TOSTRING()
Traversing the list for `toString()` // BUG VERSION

```cpp
// goal: construct string like {5, -17, 32} for list
string LinkedList::toString() const {
    string contents = "{";
    while (_front != nullptr) {
        contents += (integerToString(_front->data)); // add data
        if (_front->next != nullptr) contents += ", "; // comma unless at end
        _front = _front->next;                         // move to next node
    }
    return contents + "}";
}
```

- What's **right** and what’s **wrong** with this approach to traverse the list?
  - *Does correctly generate the string.*

- But it permanently loses the linked list as it is traversing it!
Traversing a list (12.2) (bug fixed version)

- The correct way to traverse the list:
  ```
  // goal: construct string like {5, -17, 32} for list
  string LinkedList::toString() const {
    string contents = "{";
    ListNode* current = _front;
    while (current != nullptr) {
      contents += (integerToString(current->data));
      if (current->next != nullptr) contents += ", ";
      current = current->next;
    }
    return contents + "}";
  }
  ```
- Changing the temp current does not damage the list.
LinkedList Class add() Method

METHOD NUMBER TWO
Implementing add

// Appends the given value to the end of the list.
void LinkedList::add(int value) {
    ...
}

- What pointer(s) must be changed to add a node to the end of a list?
- What different cases must we consider?
Case 1: Add to empty list

Before adding 20:

- _front: 0
- _size: 0

After:

- _front: 20
- _size: 1

- We must create a new node and attach it to the list.
- For an empty list to become non-empty, we must change _front.
Case 2: **Non-empty list**

Before adding value 20 to end of list:

Before:

| _front: | 0 |
| _size:  | 2 |

After:

| _front: | 0 |
| _size:  | 3 |

Remember to use a temporary pointer for traversal to end.
Managing our temporary pointer, `current`

Must modify the next pointer of the last node.

- Think about where `current` should be pointing, to add 20 at the end

Q: Which loop test will stop us at this place in the list?
   
   A. `while (current != nullptr) { ...`
   B. `while (_front != nullptr) { ...`
   C. `while (current->next != nullptr) { ...`
   D. `while (_front->next != nullptr) { ...`

Stanford University
Code for add

// (in linkedlist.cpp)
// Adds the given value to the end of the list.
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++;
}
More LinkedList Class Methods!

GET(), INSERT(), REMOVE()
Implementing get

// Returns value in list at given index.
int LinkedList::get(int index) {
    ...
}

- Fun tip: we’ve been using a while loop to traverse our linked list (to go to the end for add). But for insert at a specified index, a for loop is handy to get us there in a defined number of steps
// Returns value in list at given index.
int LinkedList::get(int index)
{
    if (index >= size()) {
        error("Index out of bounds!");
    }
    ListNode* current = _front;
    for (int i = 0; i < index; i++) {
        current = current->next;
    }
    return current->data;
}
Implementing insert

// Inserts the given value at the given index.
void LinkedList::insert(int index, int value) {
    ...
}

Before insert() where index = 2, value = 20:

```
_front:  
_size:  3
```

After:

```
_front:  
_size:  4
```

Before:

```
  data  next
element 0  5 -> -17 -> 32
```

After:

```
  data  next
element 0  5 -> -17 -> 20 -> 32
```
Inserting into a list

Before `insert()` where `index = 2, value = 20`:

Before:

- **_front:**
  - **_size:** 3

- **data**
  - 5

- **next**
  - -17

- **data**
  - 32

  element 0     element 1     element 2

After:

- **_front:**
  - **_size:** 4

- **data**
  - 5

- **next**
  - -17

- **data**
  - 20

- **next**
  - 32

  element 0     element 1     element 2     element 3

Your Turn: If current starts out equal to `_front`, how many times do we advance current (in the for loop) to prepare for insert?

A. `index - 1` times  
B. `index` times  
C. `index + 1` times  
D. Other
Implementing remove

// Removes value at given index from list.
void LinkedList::remove(int index) {
    ...
}

- What pointer(s) must be changed to remove a node from a list?
- What different cases must we consider?

Before remove with index = 2

- _front: 0
- _size: 4
- element 0
  data: 5
  next: 17
- element 1
  data: -17
  next: 20
- element 2
  data: 20
  next: 32
- element 3
  data: 32
  next: -1
Case 1: Removing from **front** (index 0)

Before removing element at index 0:

Before:

<table>
<thead>
<tr>
<th>front:</th>
<th>size:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

After:

<table>
<thead>
<tr>
<th>front:</th>
<th>size:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

To remove the first node, we must change _front.

Be sure to delete this!
Code for remove

// Removes value at given index from list.
void LinkedList::remove(int index) {
    if (index >= size()) {
        error("Index out of bounds!");
    }
    ListNode* trash = nullptr;
    // removing first element
    if (index == 0) {
        trash = _front;
        _front = _front->next;
    } else {
        // removing elsewhere in the list
        // left for the reader 😊
    }
    delete trash;
    size--;
Case 2: Removing from “middle” of list (ex: index 2)

Before removing element at index = 2:

- Front: 4
- Size: 4

<table>
<thead>
<tr>
<th>Index</th>
<th>Data</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-17</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>

After:

- Front: 3
- Size: 3

<table>
<thead>
<tr>
<th>Index</th>
<th>Data</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-17</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>

- Where should current be pointing?
- How many times should it advance from _front?

Be sure to delete this!

Stanford University
Case 3 (?) : Removing the only element

Before:

- \texttt{\_front}: \\
- \texttt{\_size}: 1

\begin{center}
\text{data} \quad \text{next} \quad \text{element 0}
\end{center}

After:

- \texttt{\_front}: \\
- \texttt{\_size}: 0

\begin{center}
\text{data} \quad \text{next}
\end{center}

- We must change the \texttt{\_front} field to store \texttt{nullptr} instead of pointing to a node.
- Do we \textit{really} need a special case to handle this?

\begin{center}
\text{Be sure to delete this!}
\end{center}
Other list features

A nice LinkedList class will also want to have the following public member functions:

- `size()`
- `isEmpty()`
- `set(index, value)`
- `clear()`
- `toString()`