CS 310: Linked List

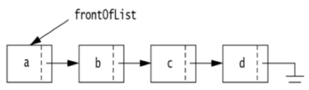
Connor Baker

January 31, 2019

List

- List notation from JavaDoc
  - package `java.util`;
  - `public interface List<E> extends Collection<E>`
  - An **unordered** collection (also known as a **sequence**). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.
  - Basic operations: `.get()`, `.set()`, `.add()`, `.remove()`
- Implementation
  - Dynamic array: based on contiguous memory
  - Linked lists: based on a series of linked nodes residing in (typically non-contiguous) memory

Singly-Linked List



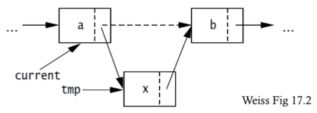
- Weiss Figure 17.1
- Node structure
  - How do we define the link between nodes?
- We can make a node that is a reference to an object, or we can make a truly generic node
  - The `Object` variety:

    ```
    class ListNode {
        Object element;
        ListNode next;
    }
    ```

  - The generic variety:

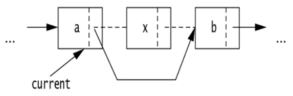
    ```
    class ListNode<T> {
        T value;
        ListNode<T> next;
    }
    ```

Linked List: Insertion



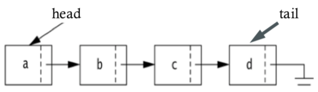
- Create a new node
- Set the fields of the new node: `data(value)`, `next`
- Add the node into the linked list
  - Find the point to insert; set/update links
- *No need to shift elements!*

Linked List: Deletion



- Set the `next` of the old predecessor
- What do we do with `Node x`?
  - Just un-link it from the other nodes. Without a reference to it, the node is garbage collected.
- What do we do if we want to delete the first node?
  - Just shift the head to the next item.
- *No need to shift elements!*

Demo

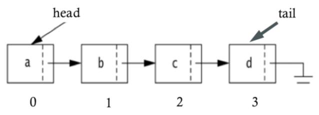


- Singly-linked list to implement a list type
  - Internally maintain a linked list
    * Node type
    * Special reference to `head`/`tail`
  - Externally support the same operations: `.get()`, `.set()` (or `.replace()`), `.append()`, `.insert()`, `.remove()`, `.indexOf()`...

Create `MyList`

```java
public class MyLList<T> {
    ...
    public int size(); // Number of items stored
    public void append(T x); // Add an element to the end of the list
    public T get(int i); // Access the i-th element
    public void set(int i, T x); // Can only replace existing elements
    public void insert(int i, T x); // Insert at position i
    public T remove(int i); // Remove element at position i
    public int indexOf(T x);
}
```

Operations on Lists



- Define an internal index based on the location with respect to the head of the linked list
  - `.set(i, x)` and `.get(i)`
- Changing the membership
  - `.append(x)`: insert at the tail (which updates where the tail is pointing)
  - `.insert(i, x)`, `.remove(i, x)`: link in a new node or de-link a node
- Iteration
  - `.indexOf(x)`, `.contains(x)`
  - `Iterator`: discuss later

Implementation

- Demo

Key Observations

- No pre-defined index like static/dynamic array
  - More expensive operations involving an index
- Most of the time we'll need to traverse the linked list
  - To traverse: how do we know where to start and where to stop?
  - For every node: what do we do with the current node? How do we know there is another node? How do we get to the next node, if it exists?

Complexity

- Linked list of size N

Method	Big-O
`.size()`	$O(1)$
`.get(i)`	$O(n)$
`.set(i, x)`	$O(n)$
`.append(x)`	$O(1)$
`.insert(i, x)`	$O(n)$
`.remove(x)`	$O(n)$
`.contains(x)`	$O(n)$

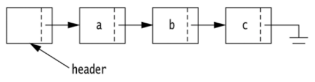
- How's the space complexity?
  - Not great – for each node, we must maintain a reference to the next node. It's $O(2n)$ (assuming that a reference to a node is the same size as a reference to the datum), which is $O(n)$, but still.

Linked List Variants

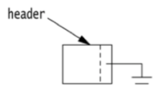
- List fields
  - Keep a reference to `head` node
  - Keep a reference to `tail` node
  - Use a dummy header node
- Node fields
  - Reference to the next node (**singly**-linked list)
  - Reference to both previous and next nodes (**doubly**-linked list)

Header Node

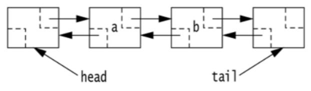
- Now we can remove the node from any location consistently
- Non-empty linked list:



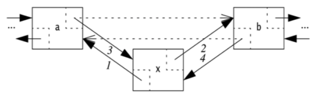
- Or an empty list with only a header:



Doubly Linked List



Doubly Linked List: Insertion



- Be careful about the order

Linked List Comparison

Implementation	get/set	add/del at end	add/del at start	add/del at mid	search	can grow?
Singly-Linked	N	1, N	1	N	N	yes
Doubly-Linked	N	1	1	N	N	yes

- Singly linked list `.add()`is constant time but `.remove()` requires searching down to the node before the end to set it to null
- Remember: to insert and remove from the middle you first have to search for the correct position in the list, which is $O(n)$
- Doubly linked uses more memory, but is still $O(n)$

Array vs. Linked List

- Array/Static array – a “row” of memory
  - We can run out of space with these because they are static
- Dynamic arrays – arrays that can grow in size
  - Cost to copy repeatedly (not so bad)
  - Insert/remove is expensive (one of the worst performance aspects)
- Linked Lists – tiny blocks of memory “linked” together
  - No “quick” memory access since we must traverse an array
  - Bad space complexity – there’s a lot of memory wasted on just keeping track of addresses

Arrays vs. Others

- Arrays are simple
  - `.get()` and `.set()` are trivial
  - `.add()` and `.remove()` are obvious (though we need to keep track of the size of the structure)
  - Very clear how data is laid out (it is contiguous, after all)
- Just about every other data structure is more complicated
  - `.get()` and `.set()` are nontrivial
  - We have to preserve some internal structure so that we can control access
  - Element-by-element access takes time

List Implementation O(n) Summary

Implementation	get/set	add/del at end	add/del start	add/del mid	search	can grow?
Static Array	1	1	N	N	N	no
Dynamic Array	1	**1***	N	N	N	no
Singly-Linked	N	1, N	1	N	N	yes
Doubly-Linked	N	1	1	N	N	yes

*Amortized analysis

Extra: Sorting with Linked List

- How do we do the following with a linked list?
  - Insertion
    - Insert using a `Comparator` to insert after the greatest lower bound.
  - Selection
  - Big-O?
    - It’ll be $O(n)$ since we’re effectively searching through the entire list before inserting our object.
- How do we code with a linked list?
  - Could you implement `.orderedInsert(x)`?
  - Could you implement `.selectionSort(x)`?

Summary

- Linked list
  - Basic structure; insertion/deletion/search
  - There are a number of variations
  - Practice by completing the code, adding more methods, and try variations
    - Doubly-linked lists, list with header nodes, etc...
    - `.orderedInsert(x)`, `.clear()`, etc...
  - Time/space analysis
- Next lecture: `Iterator`
  - Reading: Chapter 6, Chapter 15