Lecture 7

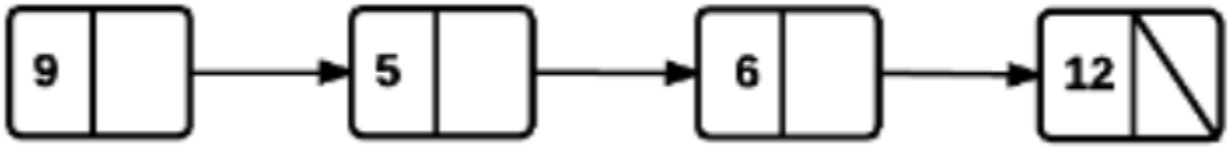
Christopher Godley

CSCI 2270 Data Structures

June 12th, 2018

Linked Lists

- Two types:
  - Singly Linked List
    - Each node stores the pointer to the next



Linked Lists

- Doubly Linked
  - Each node stores the pointer to the previous as well as the next node

Linked Lists

- Each element in the list is called a **node**
- The first node in a linked list is called the **head**,
- The last node is called the **tail**
- Typically
  - Only the head of a linked list is known
  - Sometimes, the tail is also stored

Linked List: ADT - Abstract Data Type

LinkedList:

private:

head

tail

public:

Init()

insertNode(previousValue, value)

search(value)

traverse()

deleteNode(value)

deleteList()

Linked List: Node

- A node may be represented with either structs or classes

```
struct singleNode{
        int key;
        singleNode* next;
}

struct doubleNode{
        int key;
        doubleNode* next
        doubleNode* previous;
}
```

Linked List: Example

- Build a singly linked list with three nodes with key values of 5, 6, 7
- Let’s repeat the process for a doubly linked list

Linked Lists: Traversal

- In an array, all elements are stored sequentially and are directly addressable by their index
- Let's write an algorithm:
  - Pre-Conditions
    - The *head* node is defined in the linked list ADT or included as an argument
  - Post-Conditions
    - Values of the nodes in the list are displayed

Linked Lists: Traversal

- Algorithm:

```
traverse() {
        tmp = head
        while (tmp != NULL)
                print tmp.key
                tmp = tmp.next
}
```

Linked Lists: Search

- To search a linked list, you simply traverse from the head until the desired value is found

```
search() {
        tmp = head
        returnNode = NULL
        found = false
        while (!found and tmp != NULL)
                if (tmp.key == value)
                        found = true
                        returnNode = tmp
                else
                        tmp = tmp.next
        return returnNode
}
```

Linked Lists: Insert

- We saw a brief example of how inserting into a linked list is more complicated than in an array
- Inserting into a linked list presents the opportunity to lose pointers to nodes we still need
- There are 3 cases to consider:
  - Inserting a node at the head
  - Inserting a node at the tail
  - Inserting a node in the middle

Algorithm in section 5.7

Linked Lists: Insert

- Let’s generalize the insert algorithm:
  - Pre-conditions
    - *leftValue* is a valid key value for a node in the list, or NULL
    - *value* is a valid key value
  - Post-conditions
    - The new node has been added to the list after the *leftValue* node

Linked Lists: Insert

```
insertNode(leftValue, value) {
        left = search(leftValue)
        node.key = value
        if left == NULL // this is the head node
                node.next = head
                head = node
        else if left.next == NULL // this is the tail node
                left.next = node
                tail = node
        else // this is a middle node
                node.next = left.next
                left.next = node
}
```

Linked Lists: Delete

- Deleting a node from a linked list is a matter of bypassing the pointer to the node you wish to delete and then freeing that memory
- Just as with insert, be sure to perform the requisite steps in order to prevent losing reference to your pointers
- Same 3 Cases:
  - Deleting the node at the head
  - Deleting the node at the tail
  - Deleting the node in the middle

Linked Lists: Delete

- Let’s generalize the delete algorithm:
  - Pre-Conditions
    - *Head* pointer is set in the linked list
    - *value* is a valid search parameter
  - Post-Conditions
    - Node where the key equals the value has been deleted from the list

Linked Lists: Delete

- Algorithm

```
delete(value)
        if (head.key == value) // delete the head
                tmp = head
                head = head.next
                delete tmp
        else // middle or tail
                …
```

Linked Lists: Delete

```
delete(value)
        …
        else // middle or tail
                left = head
                tmp = head.next
                found = false
                while tmp != NULL && !found
                        if tmp.key == value
                                left.next = tmp.next
                                if tmp == tail
                                        tail = left
                                delete tmp
                                found = true
                        else
                                left = tmp
                                tmp = tmp.next
```

Linked Lists: Doubly Linked

- The pseudocode in this lecture has been for singly linked lists
- Let’s try and implement these functions for doubly linked lists!