14: Analysis of Algorithms II
Lecture 14: Analysis of Algorithms II

- Theory of Algorithms
- Running Time of Linked List operations
- Running Time of Linked Stack operations
- Running Time of Linked Queue operations
- Running Time of ArrayList operations
- Memory Consumption of Stacks

Some slides adopted from Algorithms 4th Edition or COS226
Type of analyses

- **Best case**: lower bound on cost.
  - What the goal of all inputs should be.
  - Often not realistic, only applies to "easiest" input.

- **Worst case**: upper bound on cost.
  - Guarantee on all inputs.
  - Calculated based on the "hardest" input.

- **Average case**: expected cost for random input.
  - A way to predict performance.
  - Not straightforward how we model random input.
Asymptotic Notations

- **Θ notation**: bounds function from above and below.
- **O notation**: bounds function from above.
- **Ω notation**: bounds function from below.
Big O - asymptotic upper bound

- For a given function $g(n)$, $O(g(n))$ is the set of functions
  \[ \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n), \text{ for all } n > n_0 \} \]
Asymptotic analysis simplifies analyzing worst-case performance

- We will be dropping constants. For example:
  - $3n^3 + 2n + 7 = O(n^3)$
  - $2^n + n^2 = O(2^n)$
  - $1000 = O(1)$

- Yes, $3n^3 + 2n + 7 = O(n^6)$, but that’s a rather useless bound.

- Sorting them by increasing rate of growth:
  - $O(1), O(\log n), O(n), O(n \log n), O(n^2), O(n^3), O(2^n), O(n!)$
How to interpret Big O

- \( O(1) \) or "order one": running time does not change as size of the problem changes, that is running time stays constant and independent of problem size.

- \( O(\log n) \) or "order log n": running time increases as problem size grows. Whenever problem size doubles, running time increases by a constant.

- \( O(n) \) or "order n": time increases proportionally to the rate of growth of the size of the problem, that is in a linear rate. Double the problem size, you get double running time.

- \( O(n^2) \) or "order n squared": Double the problem size you get quadruple running time.
Lecture 14: Analysis of Algorithms II

- Theory of Algorithms
- Running Time of Linked List operations
- Running Time of Linked Stack operations
- Running Time of Linked Queue operations
- Running Time of ArrayList operations
- Memory Consumption of Stacks
add() in singly linked lists is $O(1)$ for worst case

```java
public void add(Item item) {
    // Save the old node
    Node oldfirst = first;

    // Make a new node and assign it to head. Fix pointers.
    first = new Node();
    first.item = item;
    first.next = oldfirst;

    n++; // increase number of nodes in singly linked list.
}
```
get() in singly linked lists is $O(n)$ for worst case

```java
public Item get(int index) {
    rangeCheck(index);

    Node finger = first;
    // search for index-th element or end of list
    while (index > 0) {
        finger = finger.next;
        index--;
    }
    return finger.item;
}
```
**RUNNING TIME OF LINKED LIST OPERATIONS**

The running time of the `add(int index, Item item)` operation in singly linked lists is $O(n)$ for worst case.

```java
public void add(int index, Item item) {
    rangeCheck(index);

    if (index == 0) {
        add(item);
    } else {

        Node previous = null;
        Node finger = first;
        // search for index-th position
        while (index > 0) {
            previous = finger;
            finger = finger.next;
            index--;
        }
        // create new value to insert in correct position.
        Node current = new Node();
        current.next = finger;
        current.item = item;
        // make previous value point to new value.
        previous.next = current;

        n++;
    }
}
```
remove() in singly linked lists is $O(1)$ for worst case

```java
public Item remove() {
    Node temp = first;
    // Fix pointers.
    first = first.next;

    n--;

    return temp.item;
}
```
remove(int index) in singly linked lists is $O(n)$ for worst case

```java
public Item remove(int index) {
    rangeCheck(index);

    if (index == 0) {
        return remove();
    } else {
        Node previous = null;
        Node finger = first;
        // search for value indexed, keep track of previous
        while (index > 0) {
            previous = finger;
            finger = finger.next;
            index--;
        }
        previous.next = finger.next;
        
        n--;
        // finger's value is old value, return it
        return finger.item;
    }
}
```
addFirst() in doubly linked lists is $O(1)$ for worst case

```java
public void addFirst(Item item) {
    // Save the old node
    Node oldfirst = first;

    // Make a new node and assign it to head. Fix pointers.
    first = new Node();
    first.item = item;
    first.next = oldfirst;
    first.prev = null;

    // if first node to be added, adjust tail to it.
    if (last == null)
        last = first;
    else
        oldfirst.prev = first;

    n++; // increase number of nodes in doubly linked list.
}
```
addLast() in doubly linked lists is $O(1)$ for worst case

```java
public void addLast(Item item) {
    // Save the old node
    Node oldlast = last;

    // Make a new node and assign it to tail. Fix pointers.
    last = new Node();
    last.item = item;
    last.next = null;
    last.prev = oldlast;

    // if first node to be added, adjust head to it.
    if (first == null)
        first = last;
    else
        oldlast.next = last;

    n++;
}
```
add(int index, Item item) in doubly linked lists is $O(n)$ for worst case

```java
public void add(int index, Item item) {
    rangeCheck(index);

    if (index == 0) {
        addFirst(item);
    } else if (index == size()) {
        addLast(item);
    } else {

        Node previous = null;
        Node finger = first;
        // search for index-th position
        while (index > 0) {
            previous = finger;
            finger = finger.next;
            index--;
        }
        // create new value to insert in correct position
        Node current = new Node();
        current.item = item;
        current.next = finger;
        current.prev = previous;
        previous.next = current;
        finger.prev = current;

        n++;
    }
}
```
removeFirst() in doubly linked lists is $O(1)$ for worst case

```java
public Item removeFirst() {
    Node oldFirst = first;
    // Fix pointers.
    first = first.next;
    // at least 1 nodes left.
    if (first != null) {
        first.prev = null;
    } else {
        last = null; // remove final node.
    }
    oldFirst.next = null;

    n--;

    return oldFirst.item;
}
```
removeLast() in doubly linked lists is \( O(1) \) for worst case

```java
public Item removeLast() {
    Node temp = last;
    last = last.prev;

    // if there was only one node in the doubly linked list.
    if (last == null) {
        first = null;
    } else {
        last.next = null;
    }
    n--;
    return temp.item;
}
```
remove(int index) in doubly linked lists is $O(n)$ for worst case

```java
public Item remove(int index) {
    rangeCheck(index);

    if (index == 0) {
        return removeFirst();
    } else if (index == size() - 1) {
        return removeLast();
    } else {
        Node previous = null;
        Node finger = first;
        // search for value indexed, keep track of previous
        while (index > 0) {
            previous = finger;
            finger = finger.next;
            index--;
        }
        previous.next = finger.next;
        finger.next.prev = previous;

        n--;
        // finger's value is old value, return it
        return finger.item;
    }
}
```
Today’s Lecture in a Nutshell

Lecture 14: Analysis of Algorithms II

- Theory of Algorithms
- Running Time of Linked List operations
- Running Time of Linked Stack operations
- Running Time of Linked Queue operations
- Running Time of ArrayList operations
- Memory Consumption of Stacks
push(Item item) in linked stack is $O(1)$ for worst case

```java
public void push(Item item) {
    Node oldfirst = first;
    first = new Node();
    first.item = item;
    first.next = oldfirst;
    n++;
}
```

- Same time complexity both for singly and doubly linked list
pop() in linked stack is $O(1)$ for worst case

```java
public Item pop() {
    if (isEmpty()) throw new NoSuchElementException("Stack underflow");
    Item item = first.item;
    first = first.next;
    n--;
    return item;
}
```

- Same time complexity both for singly and doubly linked list
Lecture 14: Analysis of Algorithms II

- Theory of Algorithms
- Running Time of Linked List operations
- Running Time of Linked Stack operations
- Running Time of Linked Queue operations
- Running Time of ArrayList operations
- Memory Consumption of Stacks
enqueue(Item item) in (doubly) linked queue is $O(1)$ for worst case

```java
public void enqueue(Item item) {
    Node oldlast = last;
    last = new Node();
    last.item = item;
    last.next = null;
    if (isEmpty())
        first = last;
    else
        oldlast.next = last;
    n++;
}
```
dequeue(Item item) in (doubly) linked queue is $O(1)$ for worst case

```java
public Item dequeue() {
    if (isEmpty())
        throw new NoSuchElementException("Queue underflow");
    Item item = first.item;
    first = first.next;
    n--;
    if (isEmpty())
        last = null;
    return item;
}
```
Queues as singly linked lists

- $O(n)$ if only head pointer and have to enqueue at the tail.
- $O(1)$ if we have a tail pointer.
  - Simple modification in code, big gains!
  - Version that textbook follows.
Lecture 14: Analysis of Algorithms II

- Theory of Algorithms
- Running Time of Linked List operations
- Running Time of Linked Stack operations
- Running Time of Linked Queue operations
- Running Time of ArrayList operations
- Memory Consumption of Stacks
Worst-case performance of `add()` is $O(n)$

- **Cost model**: 1 for insertion, $n$ for copying $n$ items to a new array.
- **Worst-case**: If arraylist is full, `add()` will need to call `resize` to create a new array of double the size, copy all items, insert new one.
- **Total cost**: $n + 1 = O(n)$.

- Realistically, this won’t be happening often and worst-case analysis can be too strict. We will use *amortized time analysis* instead.
Amortized analysis

- **Amortized cost per operation**: for a sequence of $n$ operations, it is the total cost of operations divided by $n$.

  - Simplest form of amortized analysis called aggregate method. More complicated methods exist, such as accounting (banking) and potential (physicist’s).
Amortized analysis for $n$ add() operations

As the arraylist increases, doubling happens *half as often* but costs *twice as much*.  

\[ O(\text{total cost}) = \sum (\text{"cost of insertions"}) + \sum (\text{"cost of copying"}) \]

\[ \sum (\text{"cost of insertions"}) = n. \]

\[ \sum (\text{"cost of copying"}) = 1 + 2 + 2^2 + \ldots 2^{[\log_2 n]} \leq 2n. \]

\[ O(\text{total cost}) \leq 3n, \text{ therefore amortized cost is } \leq \frac{3n}{n} = 3 = O(1), \text{ but "lumpy".} \]
Amortized analysis for $n$ add() operations when increasing arraylist by 1.

\[ \sum ("\text{cost of insertions"}) = n. \]

\[ \sum ("\text{cost of copying"}) = 1 + 2 + 3 + \ldots + n - 1 = n(n - 1)/2. \]

\[ O(\text{total cost}) = n + n(n - 1)/2 = n(n + 1)/2, \text{ therefore amortized cost is } (n + 1)/2 \text{ or } O(n). \]

Same idea when increasing arraylist size by a constant.
Lecture 14: Analysis of Algorithms II

- Theory of Algorithms
- Running Time of Linked List operations
- Running Time of Linked Stack operations
- Running Time of Linked Queue operations
- Running Time of ArrayList operations
- Memory Consumption of Stacks
A (linked) stack with $n$ items uses $\approx 40n$ bytes

- 16 bytes (object overhead)
- 8 bytes (inner class overhead)
- 8 bytes (reference to an Item)
- 8 bytes (reference to next node)
- Total: 40 bytes per stack Node

This analysis does not take into consideration the size of the Item objects.
Readings:

- Textbook:
  - Chapter 1.4 (pages 197-199)

- Website:

Practice Problems:

- 1.4.1, 1.4.5 - 1.4.7, 1.4.32, 1.4.35-1.4.36.