Linked Lists

- Series of nodes (not necessarily adjacent in memory)

```
public class Node<C> {
    C data;
    Node<C> next;
    public Node(C d, Node<C> n) {
        data = d;
        next = n;
    }
}
```

Figure 3.1 A linked list

- Each node stores at least two variables
  - Element
  - next: Link to a node containing its successor
    - NULL, for last node
Linked Lists: Insertion

- Insertion into a linked list:
  - Create a new node
  - Change two link references

```java
public void insertAfter(Node<C> n, C d){
    if (n.next != null)
        Node<C> tmp = new Node<C>(d, n.next);
    else
        Node<C> tmp = new Node<C>(d, null);
    n.next = tmp;
}
```

```
public class Node<C> {
    C data;
    Node<C> next;
    public Node(C d, Node<C> n){
        data = d; next = n;}
}
```

Linked Lists: Deletion

- Deletion from a linked list:
  - Change one link reference

```java
public void deleteAfter(Node<C> n){
    Node<C> tmp = n.next;
    if (tmp != null)
        n.next = tmp.next;
}
```

```
public class Node<C> {
    C data;
    Node<C> next;
    . . . }
```
Swapping Adjacent List Elements

- PIAZZA Problem:
  - Swap two adjacent elements by adjusting only the links (and not the data)

```java
public class Node<C> {
    C data;
    Node<C> next;
    . . .
}

public void swapNextTwo(Node<C> n) {
    // Code
}
```

Doubly-linked Lists

- Each node stores:
  - Element
  - `next`: Link to a node containing its successor
    - NULL, for last node
  - `prev`: Link to a node containing its predecessor
    - NULL, for first node

- List could be circular as well
  - First node is successor of last node
  - Last node is predecessor of first node
Implementing Queue Using a Linked List

- Let’s write this together:
  - What do we do for creating the queue?
  - How do we support:
    - enqueue – add an element at the rear
    - dequeue – remove an element from the front
    - peek – return (not remove) the element at the front
    - ...

```
6 3 1 4
```

front          rear

Array or Linked List: Which one to use?

<table>
<thead>
<tr>
<th>Array</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>May waste unneeded space or run out of space</td>
<td>Always just enough space</td>
</tr>
<tr>
<td>Uses just-enough space per element</td>
<td>But needs more space per element</td>
</tr>
<tr>
<td>Operations (so far) very simple/fast</td>
<td>Operations (so far) very simple/fast</td>
</tr>
<tr>
<td>Access to k\textsuperscript{th} element</td>
<td>Access to k\textsuperscript{th} element</td>
</tr>
<tr>
<td>Simple access</td>
<td>Must traverse all earlier elements</td>
</tr>
<tr>
<td>insertAtPosition:</td>
<td>insertAtPosition:</td>
</tr>
<tr>
<td>Must shift all later elements</td>
<td>After traversal, simple insert</td>
</tr>
</tbody>
</table>

Which operations would run faster?
Why Not Time the Implementations?

- Our consideration: which one will run faster?
- Why not just run the two implementations and time them?
  - Several factors cause variability:
    - Hardware: CPU, memory, …
    - Software: OS, Java version, libraries, drivers, …
    - Other programs running
    - Implementation efficiency
    - Choice of input (and operations)
      - Timing difference doesn’t even show up for small n
  - Timing evaluates a specific implementation in a given environment
    - Doesn’t evaluate the algorithm
    - Demo Tester.java

Comparing Algorithms: Asymptotically

- Can an algorithm be (roughly) analyzed for performance before coding it up?
  - Answer independent of CPU, language, hacks, …
  - Answer is general and rigorous

- Asymptotic Analysis (large inputs):
  - Most algorithms are fast for small n
    - If n is 5, anything should be fast enough
  - In practice, n is often large
    - Databases, Management of highly-shared resources, …
RUNNING TIME ESTIMATIONS
Big-O Notation, Code Analysis

Math Review

- Make sure you review Math sections in Chapter 1
  - Mathematics Review (Section 1.2, 3rd edition)
    - Including exponents, logarithms, binary numbers, series, ...
  - A Brief Introduction to Recursion (Section 1.3, 3rd edition)
Big-O Definition

- **Definition:**
  - \( T(N) = O(f(N)), \) if there are positive constants \( c \) and \( n_0 \)
    such that \( T(N) \leq cf(N) \) when \( N \geq n_0 \)

- **Example:**
  - Suppose for an input of size \( N \), my algorithm runs in time
    proportional to \( T(N) = 2N^2 + 10N + 34 \)
  - Then, for any \( N \geq 1 \),
    \[
    T(N) = 2N^2 + 10N + 34 
    \leq 2N^2 + 10N^2 + 34N^2 
    = 46N^2 
    \]
  - Thus, \( T(N) = O(N^2) \)
    - Using \( c = 46, n_0 = 1 \)
  - Intuitively, \( T(N) \) is *dominated* by the \( N^2 \) term
    - Eliminates *low-order terms and coefficients*

Big-O Examples

- **Definition of Big-O:**
  - \( T(N) = O(f(N)), \) if there are positive constants \( c \) and \( n_0 \)
    such that \( T(N) \leq cf(N) \) when \( N \geq n_0 \)
    - Eliminates low-order terms (by using \( \leq \))
    - Eliminates coefficients (by allowing any positive \( c \))
    - Helps specify how functions grows with large \( n \) (by using \( \geq n_0 \))

- **Examples:**
  - \( 12n + 50 \)
  - \( 5n \log(n) + 6n + 32 \)
  - \( n^3 + 2^n + 3n \)
  - \( n \log(10n^2) \)
**Big-O vs Θ**

- **O**: upper bound
  - $T(N) = O(f(N))$, if there are positive constants $c$ and $n_0$ such that $T(N) \leq cf(N)$ when $N \geq n_0$

- **Ω**: lower bound
  - $T(N) = \Omega(f(N))$, if there are positive constants $c$ and $n_0$ such that $T(N) \geq cf(N)$ when $N \geq n_0$

- **Θ**: tight bound
  - $T(N) = \Theta(f(N))$, if and only if $T(N) = O(f(N))$ and $T(N) = \Omega(f(N))$

- **Common practice**:
  - Use a tight (smallest) bound in specifying $O(.)$
  - We’ll also try to do this in this course

---

**Which Grows Faster?**

- $n^3 + 2n^2$ vs $100n^2 + 1000$
Which Grows Faster?

- $10 \log n$ vs $n$

![Graphs showing growth comparison]

Big-O: Common Bounds

- With decreasing performance (for asymptotically large $n$):
  - $O(1)$
    - Constant – $O(3)$ is same as $O(1)$
  - $O(\log n)$
    - Logarithmic – $\log n^2$, $\log_k n$ are both $O(\log n)$
  - $O(n)$
    - Linear
  - $O(n \log n)$
    - $\log n$ is $O(n \log n)$, but NOT $\Theta(n \log n)$
  - $O(n^2)$
    - Quadratic
  - $O(n^k)$
    - Polynomial (where $k > 1$)
  - $O(k^n)$
    - Exponential (where $k > 1$)
In-class Problem

- POSTED ON PIAZZA
- True or False? For each, also give a $\Theta$ bound.
  1. $63 + 24n$ is $O(n)$
  2. $13 \log n + 24n$ is $O(\log n)$
  3. $23 \log n + 45$ is $O(\log n^3)$
  4. $N^{75}$ is $O(1.1n)$