Getting in front

- **Suppose we want to add a new element**
  - At the back of a string or an ArrayList or a …
  - At the front of a string or an ArrayList or a …
  - Is there a difference? Why? What’s complexity?

- **Suppose this is an important problem: we want to grow at the front (and perhaps at the back)**
  - Think editing film clips and film splicing
  - Think DNA and gene splicing

- **Self-referential data structures to the rescue**
  - References, reference problems, recursion, binky
ArrayLists and linked lists as ADTs

- As an ADT (abstract data type) ArrayLists support
  - Constant-time or $O(1)$ access to the $k$-th element
  - Amortized linear or $O(n)$ storage/time with add
    - Total storage used in $n$-element vector is approx. $2n$, spread over all accesses/additions (why?)
  - Adding a new value in the middle of an ArrayList is expensive, linear or $O(n)$ because shifting required

- Linked lists as ADT
  - Constant-time or $O(1)$ insertion/deletion anywhere, but...
  - Linear or $O(n)$ time to find where, sequential search

- Good for sparse structures: when data are scarce, allocate exactly as many list elements as needed, no wasted space/copying (e.g., what happens when vector grows?)
Linked list applications

- Remove element from middle of a collection, maintain order, no shifting. Add an element in the middle, no shifting
  - What’s the problem with an ArrayList?
  - Editor visits several files, internally keeps a linked-list of open files
  - Naively keep characters in a linked list, but in practice too much storage, need more esoteric data structures

- What’s \((3x^5 + 2x^3 + x + 5) + (2x^4 + 5x^3 + x^2 + 4x)\) ?
  - As a vector \((3, 0, 2, 0, 1, 5)\) and \((0, 2, 5, 1, 4, 0)\)
  - As a list \(((3, 5), (2, 3), (1, 1), (5, 0))\) and ________?
  - Most polynomial operations sequentially visit terms, don’t need random access, do need “splicing”

- What about \((3x^{100} + 5)\) ?
Linked list applications continued

- If programming in C, there are no “growable-arrays”, so typically linked lists used when # elements in a collection varies, isn’t known, can’t be fixed at compile time
  - Could grow array, potentially expensive/wasteful especially if # elements is small.
  - Also need # elements in array, requires extra parameter
  - With linked list, one pointer used to access all the elements in a collection

- Simulation/modelling of DNA gene-splicing
  - Given list of millions of CGTA... for DNA strand, find locations where new DNA/gene can be spliced in
    - Remove target sequence, insert new sequence
Linked lists, CDT and ADT

- **As an ADT**
  - A list is empty, or contains an element and a list
  - \((\quad)\) or \((x, (y, (\quad)\quad))\)

- **As a picture**

- **As a CDT (concrete data type)**
  ```java
  public class Node {
      String info;
      Node next;
  }
  ```

  ```java
  Node p = new Node();
  p.info = "hello";
  p.next = null;
  ```
Building linked lists

- **Add words to the front of a list (draw a picture)**
  - Create new node with next pointing to list, reset start of list

```java
class Node {
    String info;
    Node next;
    Node(String s, Node link) {
        info = s;
        next = link;
    }
};
public class Node {
    String info;
    Node next;
    Node(String s, Node link) {
        info = s;
        next = link;
    }
};
// ... declarations here
Node list = null;
while (scanner.hasNext()) {
    list = new Node(scanner.nextString(), list);
}
```
- **What about adding to the end of the list?**
Dissection of add-to-front

- **List initially empty**
- **First node has first word**

- **Each new word causes new node to be created**
  - New node added to front

- **Rhs of operator = completely evaluated before assignment**

```java
list = new Node(word, list);
```
Standard list processing (iterative)

- Visit all nodes once, e.g., count them or process them

```java
public int size(Node list) {
    int count = 0;
    while (list != null) {
        count++;
        list = list.next;
    }
    return count;
}
```

- What changes in code above if we change what “process” means?
  - Print nodes?
  - Append “s” to all strings in list?
Nancy Leveson: Software Safety

Founded the field

- Mathematical and engineering aspects
  - Air traffic control
  - Microsoft word

"C++ is not state-of-the-art, it’s only state-of-the-practice, which in recent years has been going backwards"

- Software and steam engines: once extremely dangerous?

- THERAC 25: Radiation machine that killed many people
See Splice.java

Consider prepending (add to front) and two methods:

```java
public void prepend(String s) {
    myString = s + myString;
}

public void prepend(String s) {
    myFront = new Node(s, myFront);
    myCount += s.length();
}
```

What is hidden complexity of these operations? Why?
Timings in Splice.java

<table>
<thead>
<tr>
<th>length</th>
<th>method</th>
<th>StringStrand</th>
<th>LinkStrand</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>108,000</td>
<td>4.253</td>
<td>0.001</td>
</tr>
<tr>
<td>5000</td>
<td>135,000</td>
<td>7.028</td>
<td>0.001</td>
</tr>
<tr>
<td>6000</td>
<td>162,000</td>
<td>11.133</td>
<td>0.001</td>
</tr>
<tr>
<td>7000</td>
<td>189,000</td>
<td>16.418</td>
<td>0.001</td>
</tr>
<tr>
<td>8000</td>
<td>216,000</td>
<td>??</td>
<td>??</td>
</tr>
</tbody>
</table>
New task in Strand.java

- Rather than simply prepending, what about splicing anywhere?
  - We have `s.insert(k, str)` to add string at k\(^{th}\) position, so prepending is `s.insert(0, str)`

- We want to mirror this behavior in all classes
  - What do we do in base class?
  - How do we implement in LinkStrand class?
    - What are issues?
    - How fast will it be?
What about adding a node to the end of the list?
  - Can we search and find the end?
  - If we do this every time, what’s complexity of building an N-node list? Why?

Alternatively, keep pointers to first and last nodes of list
  - If we add node to end, which pointer changes?
  - What about initially empty list: values of pointers?
    - Will lead to consideration of header node to avoid special cases in writing code

What about keeping list in order, adding nodes by splicing into list? Issues in writing code? When do we stop searching?
Standard list processing (recursive)

- Visit all nodes once, e.g., count them

```java
public int recsize(Node list) {
    if (list == null) return 0;
    return 1 + recsize(list.next);
}
```

- Base case is almost always empty list: null pointer
  - Must return correct value, perform correct action
  - Recursive calls use this value/state to anchor recursion
  - Sometimes one node list also used, two “base” cases

- Recursive calls make progress towards base case
  - Almost always using list.next as argument
Recursion with pictures

- Counting recursively

```java
int recsize(Node list){
    if (list == null)       return 0;
    return 1 +
    recsize(list.next);
}
```

```java
System.out.println(recsize(ptr));
```
Recursion and linked lists

- Print nodes in reverse order
  - Print all but first node and...
    - Print first node before or after other printing?

```java
public void print(Node list) {
    if (list != null) {
        System.out.println(list.info);
        System.out.println(list.info);
        print(list.next);
    }
}
```
Complexity Practice

- **What is complexity of Build? (what does it do?)**

  ```java
  public Node build(int n) {
      if (null == n) return null;
      Node first = new Node(n, build(n-1));
      for(int k = 0; k < n-1; k++) {
          first = new Node(n,first);
      }
      return first;
  }
  ```

- **Write an expression for T(n) and for T(0), solve.**
  - Let T(n) be time for build to execute with n-node list
  - T(n) = T(n-1) + O(n)
Changing a linked list recursively

- Pass list to method, return altered list, assign to list
  - Idiom for changing value parameters

```java
list = change(list, "apple");
public Node change(Node list, String key) {
  if (list != null) {
    list.next = change(list.next, key);
    if (list.info.equals(key)) return list.next;
    else
      return list;
  }
  return null;
}
```

- What does this code do? How can we reason about it?
  - Empty list, one-node list, two-node list, n-node list
  - Similar to proof by induction