Getting in front

- **Suppose we want to add a new element**
  - At the back of a string or a vector or a ...
  - At the front of a string or a vector 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**
  - Pointers, pointer problems, binky
Vectors and linked lists as ADTs

- As an ADT (abstract data type) vectors support
  - *Constant-time* or $O(1)$ access to the $k$-th element
  - *Amortized* linear or $O(n)$ storage/time with `push_back`
    - Total storage used in $n$-element vector is approx. $2n$, spread over all accesses/additions (why?)
  - Adding a new value in the middle of a vector 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 a vector (array)?
  - Emacs visits several files, internally keeps a linked-list of buffers
  - 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
  - \((\ )\) or \((x, (y, (\ )\ )))\)

- **As a picture**

```
 p
```

- **As a CDT (concrete data type)**
  ```
  struct Node
  {
    string info;
    Node * next;
  };

  Node * p = new Node();
  p->info = "hello";
  p->next = 0;  // 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

```cpp
struct Node {
    string info;
    Node * next;
    Node(const string & s, Node * link)
        : info(s), next(link) {
    }
};
// ... declarations here
Node * list = 0;
while (input >> word) {
    list = new Node(word, 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

```cpp
list = new Node(word, list);
Node(const string& s, Node * link)
    : info(s), next(link)
    { }
```
Standard list processing (iterative)

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

```c
int size(Node * list)
{
    int count = 0;
    while (list != 0) {
        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?
See splice.cpp

- Consider prepending (add to front) and two methods:

  ```cpp
  virtual void prepend(const string& s) {
    myString = s + myString;
  }
  
  virtual void prepend(const string& s) {
    myFront = new Node(s, myFront);
    myCount += s.length();
  }
  ```

- What is hidden complexity of these operations? Why?
# Timings in `strand.cpp`

<table>
<thead>
<tr>
<th>length</th>
<th>method</th>
<th>StrStrand</th>
<th>LinkStrand</th>
</tr>
</thead>
<tbody>
<tr>
<td>135,000</td>
<td></td>
<td>1.79</td>
<td>0.01</td>
</tr>
<tr>
<td>270,000</td>
<td></td>
<td>7.99</td>
<td>0.01</td>
</tr>
<tr>
<td>540,000</td>
<td></td>
<td>32.49</td>
<td>0.02</td>
</tr>
<tr>
<td>810,000</td>
<td></td>
<td>??</td>
<td>??</td>
</tr>
</tbody>
</table>
New task in strand.cpp

• **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?
Building linked lists continued

- 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

  ```
  int recsize(Node * list) {
    if (list == 0) return 0;
    return 1 + recsize(list->next);
  }
  ```

- Base case is almost always empty list – NULL/0 node
  - 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

```cpp
int recsize(Node * list) {
    if (list == 0) return 0;
    return 1 + recsize(list->next);
}
```

cout << recsize(ptr) << endl;

```cpp
recsize(Node * list)
return 1+
recsize(list->next)
```

```cpp
recsize(Node * list)
return 1+
recsize(list->next)
```

```cpp
recsize(Node * list)
return 1+
recsize(list->next)
```
Recursion and linked lists

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

```cpp
void Print(Node * list)
{
    if (list != 0)
    {
        cout << list->info << endl;
        cout << list->info << endl;
        Print(list->next);
    }
}
```
"For much of my life, I have been a software voyeur, peeking furtively at other people's dirty code. Occasionally, I find a real jewel, a well-structured program written in a consistent style, free of kludges, developed so that each component is simple and organized, and designed so that the product is easy to change."

"We must not forget that the wheel is reinvented so often because it is a very good idea; I've learned to worry more about the soundness of ideas that were invented only once."