Linked lists

- Low-level (concrete) data structure, used to implement higher-level structures
  - Used to implement sequences/lists (see `CList` in Tapestry)
  - Basis of common hash-table implementations (later)
  - Similar to how trees are implemented, but simpler
- Linked lists as ADT
  - Constant-time or O(1) insertion/deletion from anywhere in list, but first must get to the list location
  - Linear or O(n) time to find an element, sequential search
  - Like a film or video tape: splicing possible, access slow
- 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

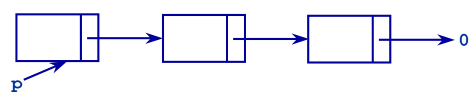
- Remove element from middle of a collection, maintain order, no shifting. Add an element in the middle, no shifting
  - What's the problem with a vector (array)?
  - Emacs visits several files, internally keeps a linked-list of *buffers*
  - Naively keep characters in a linked list, but in practice 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



- 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

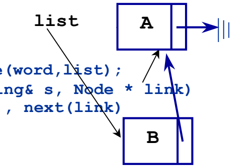
- Add words to the front of a list (draw a picture)
  - Create new node with next pointing to list, reset start of list

```
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




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

- Each new word causes new node to be created
  - New node added to front
- Rhs of operator = completely evaluated before assignment

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 (iterative)

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

```
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?

Standard list processing (recursive)

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

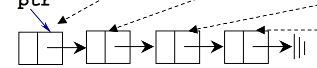
```
int recsize(Node * list)
{
    if (list == 0) return 0;
    return 1 + recsize(list->next);
}
```

- 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

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




```
cout << recsize(ptr) << endl;
```

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

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

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

```
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?

```
void Print(Node * list)
{
    if (list != 0)
    {
       Print(list->next);
       cout << list->info << endl;
       cout << list->info << endl;
       Print(list->next);
    }
}
```


Changing a linked list recursively

- Pass list to function, return altered list, assign to passed param

```
list = Change(list, “apple”);
Node * Change(Node * list, const string& key)
{
    if (list != 0) {
       list->next = Change(list->next, key);
       if (list->info == key) return list->next;
       else                   return list;
    }
    return 0;
}
```

- 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

Header (aka dummy) nodes

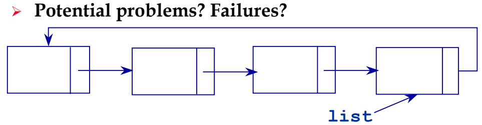
- Special cases in code lead to problems
  - Permeate the code, hard to reason about correctness
  - Avoid special cases when trade-offs permit
    - Space, time trade-offs
- In linked lists it is useful to have a header node, the empty list is not NULL/0, but a single “blank” node
  - Every node has a node before it, avoid special code for empty lists
  - Header node is skipped by some functions, e.g., count the values in a list
  - What about a special “trailing” node?
  - What value is stored in the header node?

Header Nodes example/motivation

```
Node * addInOrder(Node * list, const string& s)
// pre: list in order (or empty)
// post: node with s added to list, list in order
{
    if (list == 0) {
        return new Node(s,0);
    }
    if (s <= list->info) {
        return new Node(s, list);
    }
    // what does loop look like here?
}
```


Circularly linked list

- If the last node points to NULL/0, the pointer is “wasted”
- Can make list circular, so it is easy to add to front or back
  - Want only *one* pointer to list, should it point at first or last node?
  - How to create first node?
  - Potential problems? Failures?




```
// circularly linked, list points at last node
Node * first = list->next;
Node * current = first;
do
{
   Process(current);
   current = current->next;
} while (current != first);
```


Eric Raymond

- Open source evangelist
  - The Cathedral and the Bazaar

http://ot.op.org/cathedral-bazaar.html

  - How to construct software

“Good programmers know what to write. Great ones know what to rewrite (and reuse).”

- How to convince someone that guns are a good idea? Put this sign up:
- THIS HOME IS A GUN-FREE ZONE