ADTs and vectors, towards linked lists

- `tvector` is a class-based implementation of a lower-level data type called an array (compatible with STL/standard vector)
  - `tvector` grows dynamically (doubles in size as needed) when elements inserted with `push_back`
  - `tvector` protects against bad indexing, vector/arrays don’t
  - `tvector` supports assignment: `a = b`, arrays don’t
- 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

What is big-Oh about? (preview)

- Intuition: avoid details when they don’t matter, and they don’t matter when input size (N) is big enough
  - For polynomials, use only leading term, ignore coefficients
    - $y = 3x$  \quad $y = 6x - 2$  \quad $y = 15x + 44$
    - $y = x^2$  \quad $y = x^2 - 6x + 9$  \quad $y = 3x^2 + 4x$
- The first family is $O(n)$, the second is $O(n^2)$
  - Intuition: family of curves, generally the same shape
  - More formally: $O(\times(n))$ is an upper-bound, when n is large enough the expression $cf(n)$ is larger
  - Intuition: linear function: double input, double time, quadratic function: double input, quadruple the time

More on O-notation, big-Oh

- Big-Oh hides/obscures some empirical analysis, but is good for general description of algorithm
  - Allows us to compare algorithms in the limit
    - $20N$ hours vs $N^2$ microseconds: which is better?
- O-notation is an upper-bound, this means that N is $O(N)$, but it is also $O(N^2)$; we try to provide tight bounds. Formally:
  - A function $g(N)$ is, by definition, $O(\times(N))$ if there exist constants $c$ and $n$ such that $g(N) < cf(N)$ for all $N > n$

Big-Oh calculations from code

- Add a new element at front of vector by shifting, complexity?
  - Only count vector assignments in code below, don’t account for vector growing
    - `a.push_back(newElement); // make room for it`
    - `for(int k=a.size()-1; k>=0; k--){ a[k+1] = a[k]; // shift right }`
    - `a[0] = newElement;`
  - If we call the code above N times on an initially empty vector, what’s the complexity using big-Oh?
  - Now, what about complexity of growing a vector after N insertions
    - If vector doubles in size? If vector increases by one?
Some helpful mathematics

- \(1 + 2 + 3 + 4 + \ldots + N\)
  - \(\frac{N(N+1)}{2}\), exactly = \(\frac{N^2}{2} + \frac{N}{2}\) which is \(O(N^2)\) why?
- \(N + N + N + \ldots + N\) (total of \(N\) times)
  - \(N\cdot N = N^2\) which is \(O(N^2)\)
- \(N + N + N + \ldots + N + \ldots + N + \ldots + N\) (total of \(3N\) times)
  - \(3N\cdot N = 3N^2\) which is \(O(N^2)\)
- \(1 + 2 + 4 + \ldots + 2^N\)
  - \(2^{N+1} - 1 = 2 \times 2^N - 1\) which is \(O(2^N)\)

Impact of last statement on adding \(2^N+1\) elements to a vector
- \(1 + 2 + \ldots + 2^N + 2^N+1 = 2^{N+2} - 1 = 4\times2^N-1\) which is \(O(2^N)\)

Running times @ 10^6 instructions/sec

<table>
<thead>
<tr>
<th>(N)</th>
<th>(O(\log N))</th>
<th>(O(N))</th>
<th>(O(N \log N))</th>
<th>(O(N^2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.000003</td>
<td>0.0001</td>
<td>0.000033</td>
<td>0.0001</td>
</tr>
<tr>
<td>100</td>
<td>0.000007</td>
<td>0.0010</td>
<td>0.000064</td>
<td>0.1000</td>
</tr>
<tr>
<td>1,000</td>
<td>0.000010</td>
<td>0.0100</td>
<td>0.010000</td>
<td>1.0</td>
</tr>
<tr>
<td>10,000</td>
<td>0.000013</td>
<td>0.1000</td>
<td>0.132900</td>
<td>1.7 min</td>
</tr>
<tr>
<td>100,000</td>
<td>0.000017</td>
<td>1.0000</td>
<td>1.661000</td>
<td>2.78 hr</td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.000020</td>
<td>19.9</td>
<td>11.6 day</td>
<td>318 centuries</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>0.000030</td>
<td>16.7 min</td>
<td>18.3 hr</td>
<td></td>
</tr>
</tbody>
</table>

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 \((3x^3 + 2x^2 + x + 5) + (2x^4 + 5x^3 + x^2 + 4x)\)?
  - 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 are 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;
  }; // ... declarations here

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

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
  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

  ```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?

Standard list processing (recursive)

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

  ```c
  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

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

  ```c
  cout << recsize(ptr) << endl;
  ```
Recursion and linked lists

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

```c
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

```c
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;
}
```

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?

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?

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