Wordcounting, sets, and the web

- How does Google store all web pages that include a reference to “peanut butter with mustard”?
  - What efficiency issues exist?
  - Why is Google different (better)?

- How do `readset.cpp` and `readset2.cpp` differ?
  - Mechanisms for insertion and search
  - How do we discuss? How do we compare performance?

- If we stick with linked lists, how can we improve search?
  - Where do we want to find things?
  - What do thumb indexes do in a dictionary?
    - What about 256 different linked lists? `readset3.cpp`

Analysis: Algorithms and Data Structures

- We need a vocabulary to discuss performance and to reason about alternative algorithms and implementations
  - What’s the best way to sort, why?
  - What’s the best way to search, why?
  - Which is better, `readset`, `readset2`, `readset3`, `readset4`?

- We need both empirical tests and analytical/mathematical reasoning
  - Given two methods, which is better? Run them to check.
    - 30 seconds vs. 3 seconds, easy. 5 hours vs. 2 minutes, harder
  - Which is better? Analyze them.
    - Use mathematics to analyze the algorithm, the implementation is another matter

Multiplying and adding big-Oh

- Suppose we do a linear search then we do another one
  - What is the complexity?
  - If we do 100 linear searches?
  - If we do n searches on a vector of size n?

- What if we do binary search followed by linear search?
  - What are big-Oh complexities? Sum?
  - What about 50 binary searches? What about n searches?

- What is the number of elements in the list (1,2,2,3,3,3)?
  - What about (1,2, ..., n,n,...,n)?
  - How can we reason about this?

Helpful formulae

- We always mean base 2 unless otherwise stated
  - What is \( \log(1024) \)?
  - \( \log(xy) = \log(x) + \log(y) \)
  - \( \log(2^n) = 2 \log n \)
  - \( n \log(2) = n \)
  - \( 2 \log n = n \)

- Sums (also, use sigma notation when possible)
  - \( 1 + 2 + 4 + 8 + \ldots + 2^k = 2^{k+1} - 1 = \sum_{i=0}^{k} 2^i \)
  - \( 1 + 2 + 3 + \ldots + n = n(n+1)/2 = \sum_{i=1}^{n} i \)
  - \( a + ar + ar^2 + \ldots + ar^{n-1} = a(r^n - 1)/(r-1) = \sum_{i=0}^{n-1} ar^i \)
Different measures of complexity

- **Worst case**
  - Gives a good upper-bound on behavior
  - Never get worse than this
  - Drawbacks?

- **Average case**
  - What does average mean?
  - Averaged over all inputs? Assuming uniformly distributed random data?
  - Drawbacks?

- **Best case**
  - Linear search, useful?

Recurrences

- **Counting nodes**
  
  ```c
  int length(Node * list)
  {
    if (0 == list) return 0;
    else return 1 + length(list->next);
  }
  ```
  
  - What is complexity? Justification?
  - \( T(n) = \text{time to compute length for an n-node list} \)
    \[
    T(n) = T(n-1) + 1
    \]
    \[
    T(0) = 1
    \]
  - Instead of 1, use \( O(1) \) for constant time
    - independent of \( n \), the measure of problem size

Solving recurrence relations

- **plug, simplify, reduce, guess, verify?**

  \[
  T(n) = T(n-1) + 1 \\
  T(0) = 1 \\
  T(n-1) = T(n-1-1) + 1 \\
  T(n) = [T(n-2) + 1] + 1 = T(n-2)+2 \\
  T(n-2) = T(n-2-1) + 1 \\
  T(n) = [ (T(n-3) + 1) + 1 ] + 1 = T(n-3)+3 \\
  T(n) = T(n-k) + k 
  \]

  find the pattern!

  Now, let \( k=1 \), then \( T(n) = T(0)+n = 1+n \)

  - get to base case, solve the recurrence: \( O(n) \)

Consider merge sort for linked lists

- **Given a linked list, we want to sort it**
  - Divide the list into two equal halves
  - Sort the halves
  - Merge the sorted halves together

- **What’s complexity of dividing an n-node in half?**
  - How do we do this?

- **What’s complexity of merging (zipping) two sorted lists?**
  - How do we do this?

  \[
  T(n) = \text{time to sort n-node list} = 2 \ T(n/2) + O(n) \quad \text{why?}
  \]
sidebar: solving recurrence

\[ T(n) = 2T(n/2) + n \]
\[ T(1) = 1 \]

\[ T(\text{XXX}) = 2T(\text{XXX}/2) + \text{XXX} \]

(note: \( n/2/2 = n/4 \))

\[ T(n) = 2 \left[ 2T(n/4) + n/2 \right] + n \]
\[ = 4T(n/4) + n + n \]
\[ = 4 \left[ 2T(n/8) + n/4 \right] + 2n \]
\[ = 8T(n/8) + 3n \]
\[ = \ldots \text{eureka!} \]
\[ = 2^k T(n/2^k) + kn \]

let \( 2^k = n \)

\[ k = \log n, \text{this yields } 2^{\log n} \cdot T(n/2^{\log n}) + n(\log n) \]

\[ n T(1) + n(\log n) \]

\[ O(n \log n) \]

Complexity Practice

- What is complexity of \texttt{Build}? (what does it do?)

```c
Node * Build(int n)
{
    if (0 == n) return 0;
    else
    {
        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.

Recognizing Recurrences

- Solve once, re-use in new contexts
  - \( T \) must be explicitly identified
  - \( n \) must be some measure of size of input/parameter
    - \( T(n) \) is the time for quicksort to run on an \( n \)-element vector

\[ T(n) = T(n/2) + O(1) \quad \text{binary search} \quad O(\ log \ n \ ) \]
\[ T(n) = T(n-1) + O(1) \quad \text{sequential search} \quad O(\ n \ ) \]
\[ T(n) = 2T(n/2) + O(1) \quad \text{tree traversal} \quad O(\ n \ ) \]
\[ T(n) = 2T(n/2) + O(n) \quad \text{quicksort} \quad O(\ n \log n \ ) \]
\[ T(n) = T(n-1) + O(n) \quad \text{selection sort} \quad O(\ n^2 \ ) \]

- Remember the algorithm, re-derive complexity

Binary Trees

- Linked lists have efficient insertion and deletion, but inefficient search
  - Vector/array: search is efficient, insertion/deletion are not
- Binary trees are structures that yield efficient insertion, deletion, and search
  - trees used in many contexts, not just for searching, e.g., expression trees
  - search is as efficient as binary search in array, insertion/deletion as efficient as linked list (once node found)
  - binary trees are inherently recursive, difficult to process trees non-recursively, but possible (recursion never required, but often makes coding/algorithms simpler)
From doubly-linked lists to binary trees

- Instead of using prev and next to point to a linear arrangement, use them to divide the universe in half
  - Similar to binary search, everything less goes left, everything greater goes right
  - How do we search?
  - How do we insert?

Basic tree definitions

- Binary tree is a structure:
  - empty
  - root node with left and right subtrees
- terminology: parent, children, leaf node, internal node, depth, height, path
  - link from node N to M then N is parent of M
    - M is child of N
  - leaf node has no children
    - internal node has 1 or 2 children
  - path is sequence of nodes, N₁, N₂, ... Nₖ
    - Nᵢ is parent of Nᵢ₊₁
    - sometimes edge instead of node
  - depth (level) of node: length of root-to-node path
    - level of root is 1
  - height of node: length of longest node-to-leaf path
    - height of tree is height of root

Printing a search tree in order

- When is root printed?
  - After left subtree, before right subtree.

```cpp
void Visit(Node * t)
{
  if (t != 0)
  {
    Visit(t->left); cout << t->info << endl; Visit(t->right);
  }
}
```

- Inorder traversal

Insertion and Find? Complexity?

- How do we search for a value in a tree, starting at root?
  - Can do this both iteratively and recursively, contrast to printing which is very difficult to do iteratively
  - How is insertion similar to search?

- What is complexity of print? Of insertion?
  - Is there a worst case for trees?
  - Do we use best case? Worst case? Average case?

- How do we define worst and average cases
  - For trees? For vectors? For linked lists? For vectors of linked-lists?
From concept to code with binary trees

- Trees can have many shapes: short/bushy, long/stringy
  - if height is h, number of nodes is between h and 2^h - 1
  - single node tree: height = 1, if height = 3

- C++ implementation, similar to doubly-linked list

```c++
struct Tree {
  string info;
  Tree * left;
  Tree * right;
  Tree(const string& s, Tree * lptr, Tree * rptr) :
    info(s), left(lptr), right(rptr) {
  }
};
```

Tree functions

- Compute height of a tree, what is complexity?
  ```c++
  int height(Tree * root) {
    if (root == 0) return 0;
    else {
      return 1 + max(height(root->left),
                      height(root->right));
    }
  }
  ```

- Modify function to compute number of nodes in a tree, does complexity change?
- What about computing number of leaf nodes?

Tree traversals

- Different traversals useful in different contexts
  - Inorder prints search tree in order
    - Visit left-subtree, process root, visit right-subtree
  - Preorder useful for reading/writing trees
    - Process root, visit left-subtree, visit right-subtree
  - Postorder useful for destroying trees
    - Visit left-subtree, visit right-subtree, process root

```c++
void insert(Tree *& t, const string& s) // pre: t is a search tree // post: s inserted into t, t is a search tree {
  if (t == 0) t = new Tree(s,0,0);
  else if (s <= t->left) insert(t->left,s);
  else                   insert(t->right,s);
}
```

- Note: in each recursive call, the parameter t in the called clone is either the left or right pointer of some node in the original tree
  - Why is this important?
  - Why must t be passed by reference?
  - For alternatives see readset4.cpp
Balanced Trees and Complexity

- A tree is height-balanced if
  - Left and right subtrees are height-balanced
  - Left and right heights differ by at most one

```cpp
bool isBalanced(Tree * root)
{
    if (root == 0) return true;
    else
    {
        return
            isBalanced(root->left) &&
            isBalanced(root->right) &&
            abs(height(root->left) - height(root->right)) <= 1;
    }
}
```

What is complexity?

- Assume trees are “balanced” in analyzing complexity
  - Roughly half the nodes in each subtree
  - Leads to easier analysis

- How to develop recurrence relation?
  - What is T(n)?
  - What other work is done?

- How to solve recurrence relation
  - Plug, expand, plug, expand, find pattern
  - A real proof requires induction to verify that pattern is correct

Simple header file (see readset4.cpp)

```cpp
class TreeSet
{
    public:
        TreeSet();
        bool contains(const string& word) const;
        void insert(const string& word);
    private:
        struct Node
        {
            string info;
            Node * left * right; // need constructor
        };
        Node * insertHelper(Node * root, const string& s);
        Node * myRoot;
};
```

Helper functions in readset4.cpp

```cpp
void TreeSet::insert(const string& s)
{
    myRoot = insertHelper(myRoot);
}
TreeSet::Node * TreeSet::insertHelper(Node * root, const string& s)
{
    // recursive insertion
}
```

- Why is helper function necessary? Is it really necessary?
  - Alternatives for other functions: print/contains, for example
  - What about const-ness for public/private functions?
  - What about TreeSet::Node syntax? Why?