Scoreboard

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Insertion</th>
<th>Deletion</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unssorted Vector/array</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted vector/array</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linked list</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted linked lists</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Doubly-linked/circular</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>linked lists</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash Maps</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- What else might we want to do with a data structure?
- What are maps’ limitations?

Graphs: Structures and Algorithms

- How do packets of bits/information get routed on the internet
  - Message divided into packets on client (your) machine
  - Packets sent out using routing tables toward destination
    - Packets may take different routes to destination
    - What happens if packets lost or arrive out-of-order?
  - Routing tables store local information, not global (why?)

- What about The Oracle of Bacon, Six Degrees of Separation, Erdos Numbers, and Word Ladders?
  - All can be modeled using graphs
  - What kind of connectivity does each concept model?

- Graphs are everywhere in the world of algorithms (world?)

Vocabulary

- Graphs are collections of vertices and edges
  - Vertex is sometimes called a node
  - An edge connects two vertices
    - Direction is sometimes important, other times not so
    - Sometimes edge has a weight/cost associated with it

- A sequence of vertices $v_0, v_1, v_2, ..., v_{n-1}$ is a path where $v_k$ and $v_{k+1}$ are connected by an edge.
  - If some vertex is repeated, the path is a cycle
  - Trees are cycle-free graphs with a root
  - Nodes have in-degree and out-degree
    - Number of edges entering or leaving a node
    - Binary trees are acyclic rooted graphs where the nodes have out-degree $\leq 2$ and in-degree $= 1$ for all nodes besides root

Binary Trees

- Binary trees are structures that yield efficient insertion, deletion, and search
  - Trees used in many contexts, not just for searching, e.g., expression trees
  - Insertion 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?
- How are lists and trees related?

Printing a search tree in order

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

```c++
void Visit(Node * t)
{
    if (t != 0)
    {
        Visit(t->prev);
        cout << t->info << endl;
        Visit(t->next);
    }
}
```

- Inorder traversal

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, N_1, N_2, ..., N_k
    - N_i is parent of N_{i+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

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 different implementations of multiset (append, prepend, move-to-front)
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

```cpp
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?

```cpp
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

```
```

Insertion into search tree

- Simple recursive insertion into tree

```cpp
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 basictree.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

```c
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

Sidebar: solving recurrence

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

\[ T(1) = 1 \]

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

Let \(2^k = n\), this yields \(2^{\log n} T(n/2^{\log n}) + n(\log n)\)

Class implementation issues: trees

- Consider a class Set implemented using search trees
  - Could develop a search tree class, could use “raw” nodes in implementing the set
  - For tree class, exposing Nodes to client programs makes coding more simple than encapsulating tree
    - We view trees/nodes as a concrete implementation
    - Client classes/code manipulate trees via pointers
    - Other alternatives exist, (i.e., complete encapsulation)

- Many tree functions are recursive, parameter is a Node/Tree
  - Don’t want clients of ADT to know trees used, therefore cannot have public functions with Node params
  - Alternative? Private helper function called by public
**Simple header file with helper functions**

```cpp
class TreeSet
{
  public:
    TreeSet();
    bool add(const string& word);
    void print() const;
  
  private:
    struct Node
    {
      string info;
      Node * left * right;
    }
    void addHelp (Node *& root, const string& word);
    void printHelp(Node * root) const;
    Node * myRoot;
};
```

**Private helper functions to the rescue**

```cpp
void TreeSet::print() const
{
  printHelp(myRoot);
}
void TreeSet::printHelp(Node * root) const
{
  // standard inorder traversal
}
```

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

**Deleting a node from a tree**

- Deleting nodes is tricky
  - What about gopher, leopard?
  - What about monkey?
  - What about jaguar?

- Delete giraffe?
  - Minimize moved nodes
  - Possibilities?

- Use inorder successor as replacement, guaranteed at most one child
  - Where is it, why only one child?