Binary Trees

- Linked lists have efficient insertion and deletion, but inefficient search
  - Vector/array: search can be efficient, insertion/deletion 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 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, …, Nₙ
    - 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 (measured in nodes)
  - 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);      // Visit left subtree
        cout << t->info << endl; // Visit root
        visit(t->right);      // Visit right subtree
    }
}
```

- 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
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 `readsettree.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
tool 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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(n) = T(n/2) + O(1)</td>
<td>binary search</td>
</tr>
<tr>
<td>T(n) = T(n-1) + O(1)</td>
<td>sequential search</td>
</tr>
<tr>
<td>T(n) = 2T(n/2) + O(1)</td>
<td>tree traversal</td>
</tr>
<tr>
<td>T(n) = 2T(n/2) + O(n)</td>
<td>quicksort</td>
</tr>
<tr>
<td>T(n) = T(n-1) + O(n)</td>
<td>selection sort</td>
</tr>
</tbody>
</table>

- Remember the algorithm, re-derive complexity
Simple header file (see readsettree.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 * myRoot;
}

Helper functions in readsettree.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?

Balanced Search Trees

- BST: efficient lookup, insertion, deletion
  - Average case: $O(\log n)$ for all operations since find is $O(\log n)$ [complexity of insert after find is $O(1)$, why?]
  - Worst case is bad, what's big-Oh? What's the tree shape?
  - If we can guarantee $\log n$ in worst case, why might this be preferable to hashing? Properties of search tree?

- Balanced Search trees
  - Use rotations to maintain balance, different implementations rotate/rebalance at different times
  - AVL tree is conceptually simple, bookkeeping means coefficient for big-Oh is higher than other ideas
  - Red-black tree harder to code but good performance: basis for Java map classes and most C++ map classes

Balanced trees we won't study

- B-trees are used when data is both in memory and on disk
  - File systems, really large data sets
  - Rebalancing guarantees good performance both asymptotically and in practice. Differences between cache, memory, disk are important

- Splay trees rebalance during insertion and during search, nodes accessed often move closer to root
  - Other nodes can move further from root, consequences?
    - Performance for some nodes gets better, for others ...
  - No guarantee running time for a single operation, but guaranteed good performance for a sequence of operations, this is good amortized cost (vector push_back)
Balanced trees we will study

- Both kinds have worst-case $O(\log n)$ time for tree operations
- AVL (Adel'son-Velskii and Landis), 1962
  - Nodes are "height-balanced", subtree heights differ by 1
  - Rebalancing requires per-node bookkeeping of height
- Red-black tree uses same rotations, but can rebalance in one pass, contrast to AVL tree
  - In AVL case, insert, calculate balance factors, rebalance
  - In Red-black tree can rebalance on the way down, code is more complex, but doable
  - STL in C++ uses red-black tree for map and set classes
  - Standard Java.util.TreeMap/TreeSet use red-black

Rotations and balanced trees

- Height-balanced trees
  - For every node, left and right subtree heights differ by at most 1
  - After insertion/deletion need to rebalance
  - Every operation leaves tree in a balanced state: invariant property of tree
- Find deepest node that's unbalanced then make sure:
  - On path from root to inserted/deleted node
  - Rebalance at this unbalanced point only

Rotation to rebalance

```
Tree * doLeft(Tree * root)
{
    Tree * newRoot = root->left;
    root->left = newRoot->right;
    newRoot->right = root;
    return newRoot;
}
```

Rotation up close (doLeft)

```
Tree * doLeft(Tree * root)
{
    Tree * newRoot = root->left;
    root->left = newRoot->right;
    newRoot->right = root;
    return newRoot;
}
```

Why is this called doLeft?
- N will no longer be root, new value in left->left subtree
- Left child becomes new root
- Rotation isn’t “to the left”, but rather “brings left child up”
- doLeftChildRotate?
Rotation to rebalance

- Suppose we add a new node in right subtree of left child of root
  - Single rotation can't fix
  - Need to rotate twice
- First stage is shown at bottom
  - Rotate blue node right
    - (its right child takes its place)
  - This is left child of unbalanced

```cpp
Tree * doRight(Tree * root) {
  Tree * newRoot = root->right;
  root->right = newRoot->left;
  newRoot->left = root;
  return newRoot;
}
```

Double rotation complete

- Calculate where to rotate and what case, do the rotations

```cpp
Tree * doRight(Tree * root) {
  Tree * newRoot = root->right;
  root->right = newRoot->left;
  newRoot->left = root;
  return newRoot;
}
```

AVL tree practice

- Insert into AVL tree:
  - 18 10 16 12 6 3 8 13 14
  - After adding 16:
    - doLeftRight
    - doLeft
    - After 3, doLeft on 16

```cpp
Tree * doLeftRight(Tree * root) {
  doLeft(root);
  doRight(root);
  return root;
}
```

AVL practice: continued, and finished

- After adding 13, ok
- After adding 14, not ok
  - doRight at 12

```cpp
Tree * doLeft(Tree * root) {
  Tree * newRoot = root->left;
  root->left = newRoot->right;
  newRoot->right = root;
  return newRoot;
}
```
Trie: efficient search of words/suffixes

- A trie (from retrieval, but pronounced “try”) supports
  - Insertion: a word into the trie (delete and look up)
  - These operations are $O$ (size of string) regardless of how many strings are stored in the trie! Guaranteed!

- In some ways a trie is like a 128 (or 26 or alphabet-size) tree, one branch/edge for each character/letter
  - Node stores branches to other nodes
  - Node stores whether it ends the string from root to it

- Extremely useful in DNA/string processing
  - Monkeys and typewriter simulation which is similar to some methods used in Natural Language understanding (n-gram methods)

Trie picture and code (see trie.cpp)

- To add string
  - Start at root, for each char create node as needed, go down tree, mark last node

- To find string
  - Start at root, follow links
    - If NULL, not found
  - Check word flag at end

- To print all nodes
  - Visit every node, build string as nodes traversed

- What about union and intersection?
  - Indicates word ends here

Scoreboard

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Insertion</th>
<th>Deletion</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted 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>Hash Maps</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary search tree</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVL tree</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- What else might we want to do with a data structure?

Boggle: Tries, backtracking, structure

Find words on 4x4 grid

- Adjacent letters:
  - $\phi \gamma \eta \iota \kappa \lambda \mu$
- No re-use in same word

Two approaches to find all words

- Try to form every word on board
  - Look up prefix as you go
    - Trie is useful for prefixes

- Look up every word in dictionary
  - For each word: on board?

- ZEAL and SMILES
Search board for word: trieboggle.cpp

```cpp
void search(int row, int col, TrieNode * t, string soFar) {
    // pre: row, col are on board, soFar is valid prefix,
    //      string constructed on board during current search
    //      t represents the path in the trie of soFar
    if (!legal(row, col) || isVisited(row, col)) return;
    char ch = myBoard[row][col]; // check if still a prefix
    Node * child = t->links[ch]; // NOT a prefix, stop
    if (child == 0) return;      // still prefix, continue
    markVisited(row, col);       // don’t revisit
    string newPrefix = word + ch;    if (child->isWord) cout << newPrefix << endl;
    doFind(row-1,col-1,child,newPrefix); // up-left
    doFind(row-1,col, child,newPrefix); // straight up
    doFind(row-1,col+1,child,newPrefix); // still 5 more calls
    unVisit(row,col);            // now ok to revisit
}
```

Search for all words: boggle.cpp

```cpp
bool wordFound(const string& s, const Point& p) {
    // pre: s is suffix of word searched for, prefix so far
    //      is found and last letter of found prefix at p(row,col)
    if (s.length() == 0) return true;  // no more suffix, done
    tvector<Point> points = myPointsFor(s[0]);
    for(int k=0; k < points.size(); k++) {
        Point nextP = points[k];
        if (IsAdjacent(p,nextP) && ! isVisited(nextP)) {
            markVisited(nextP);     // don’t visit again
            if (wordFound(s.substr(1,s.length()-1),nextP)) {
                return true;
            }
            unVisit(nextP);         // ok to visit again
        }
    }
    return false; // tried to find s, failed in all attempts
}
```