Admin

- Today's topics
  - Binary search trees, implementing Map as a tree
- Reading
  - Ch 13

Map as Vector

<table>
<thead>
<tr>
<th></th>
<th>Unsorted</th>
<th>Sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map()</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>~Map()</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>add()</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>getValue()</td>
<td>O(N)</td>
<td>O(logN)</td>
</tr>
<tr>
<td>Overhead per entry</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

A different strategy

- Sorting the Vector
  - Provides fast lookup, but still slow to insert (because of shuffling)
- Does a linked list help?
  - Easy to insert, once at a position
  - But hard to find position to insert...
  - Will rearranging pointers help?

Voila... a binary search tree!

Tree terminology

- Node, tree, subtree, parent, child, root, leaf
**Trees in general**

- Rules for all trees
  - Recursive branching structure
  - Single root node
  - Every node reachable from root by unique path
- Examples
  - Game tree
  - Family tree
  - Filesystem hierarchy
  - Decomposition tree
  - Binary, ternary, n-ary
  - Binary search tree

![Tree Diagram]

**Binary search tree in specific**

- Binary tree
  - Each node has at most 2 children
- Binary search tree
  - Arranged for efficient search/insert
  - All nodes in left subtree are less than root, all nodes in right subtree are greater

```c
struct node {
    int val;
    node *left, *right;
};
```

**Operating on trees**

- Many tree algorithms are recursive
  - Not surprisingly!
  - Handle current node, recur on subtrees
  - Base case is empty tree (NULL)
- Tree traversals to visit all nodes
  - Handle current node, visit left/right subtrees
- Whether current node before/after its subtrees determines order of traversal
  - Pre: cur, left, right
  - In: left, cur, right
  - Post: left, right, cur
  - Others: level-by-level, reverse orders, etc.

**Tree traversals at work**

- **INORDER**
  ```c
  void PrintTree(node *t) {
    if (t != NULL) {
      PrintTree(t->left);
      cout << t->key << endl;
      PrintTree(t->right);
    }
  }
  ```
- **POSTORDER**
  ```c
  void FreeTree(node *t) {
    if (t != NULL) {
      FreeTree(t->left);
      FreeTree(t->right);
      delete t;
    }
  }
  ```
Implementing Map as tree

- Each Map entry adds node to tree
  - Node contains:
    - string key, client-type value, pointers to left/right subtrees
- Tree organized for binary search
  - Key is used as search field
  - Quickly find matching key or place to insert new key
- getValue
  - Searches tree, comparing keys, find existing match or error
- add
  - Searches tree, comparing keys, overwrites existing or adds new node

Private members for Map

```
template <typename ValType>
class Map
{
    public:
        // as before
    private:
        struct node {
            string key;
            ValType value;
            node *left, *right;
        };
        node *root;
    
        node *treeSearch(node *t, string key) {
            if (t == NULL) return NULL;   // doesn't exist
            if (key == t->key)                     // found match
                return t;
            else if (key < t->key)
                return treeSearch(t->left, key);   // search left
            else
                return treeSearch(t->right, key);  // search right
        }

        ValType getValue(string key)     // getValue is wrapper
        {
            node *found = treeSearch(root, key);       // for treeSearch rec fn
            if (found == NULL)
                Error("getValue of non-existent key!");
            else
                return found->value;
        }

        typename Map<ValType>::node *
        template <typename ValType>
        typename Map<ValType>::::treeSearch(node *t, string key)
        {
            if (t == NULL) return NULL;   // doesn't exist
            if (key == t->key)                     // found match
                return t;
            else if (key < t->key)
                return treeSearch(t->left, key);   // search left
            else
                return treeSearch(t->right, key);  // search right
        }
```

Map implementation

```
template <typename ValType>
ValType Map<ValType>::::getValue(string key)
{
    node *found = treeSearch(root, key);       // for treeSearch rec fn
    if (found == NULL) Error("getValue of non-existent key!");
    else
        return found->value;
}
```

Adding to a binary search tree

- Starts like getValue
  - Trace out path where node should be
- Add node as new leaf
  - Don’t change any other nodes/pointers
  - Correct place to maintain binary search property
Map implementation

```cpp
template <typename ValType>
void Map<ValType>::add(string key, ValType val) // add is wrapper
{
    treeEnter(root, key, val); // call rec helper to do enter
}
```

Recursive treeEnter

```cpp
template <typename ValType>
void Map<ValType>::treeEnter(node * & t, string key, ValType val)
{
    if (t == NULL) {
        t = new node;
        t->key = key;
        t->value = val;
        t->left = t->right = NULL;
    } else if (key == t->key) {
        t->value = val;
    } else if (key < t->key) {
        treeEnter(t->left, key, val);
    } else {
        treeEnter(t->right, key, val);
    }
}
```

Trace treeEnter

```
Insert new node
```

Evaluate Map as tree

- **Space used**
  - Overhead of two pointers per entry (typically 8 bytes total)
  - Tree adds nodes as needed, no excess capacity maintained
- **Runtime performance**
  - Add/getValue take time proportional to tree height
  - Height expected to be $O(\log N)$
A balanced tree
- Values: 2 8 14 15 18 20 21
- Different trees possible, depends on order inserted
- 7 nodes, expected height $\lg 7 \approx 3$
- Perfectly balanced

![Balanced Tree Diagram]

Entered: 15 8 2 20 21 14 18
(one possibility)

Mostly balanced trees
- Same values: 2 8 14 15 18 20 21
- Mostly balanced, height 4 or 5

![Mostly Balanced Tree Diagrams]

Entered: 20 8 21 18 14 15 2
(one possibility)
Entered: 18 14 15 8 2 20 21
(one possibility)

Degenerate trees
- Same values: 2 8 14 15 18 20 21
- Totally unbalanced, height = 7

![Degenerate Trees Diagrams]

Entered: 2 8 14 15 18 20 21
(only possibility)
Entered: 21 20 18 15 14 8 2
(only possibility)

Even more degenerate trees
- What is relationship between worst-case inputs for tree insertion and Quicksort?

![Even More Degenerate Trees Diagrams]

Entered: 21 2 20 8 14 15 18
Entered: 2 8 21 20 18 14 15
What to do about it?

- Might ignore degenerate outcomes if rare
  - But does that apply here?
- Wait til problem then re-balance entire tree
  - Monitor height to note when out of whack
  - Copy values to array (travel inorder to get sorted)
  - Take middle element and create new root node
  - Recursively convert left/right subarrays to subtrees
- Never let it get lopsided to begin with
  - Constantly monitor balance for each subtree
  - Rebalance subtree before going too far astray

AVL trees

- Self-balancing binary search tree
- Track balance factor for each node
  - Height of right subtree - height of left subtree
- Balance factor of 0 or 1 is ok
  - Tree is within one level of balanced
- When balance factor hits 2, restructure
  - "Rotation" moves nodes from heavy to light side
    - Local rearrangement around specific node
    - When finished, node has 0 balance factor

Compare Map implementations

<table>
<thead>
<tr>
<th></th>
<th>Vector</th>
<th>Sorted Vector</th>
<th>BST</th>
</tr>
</thead>
<tbody>
<tr>
<td>getValue</td>
<td>O(N)</td>
<td>O(lgN)</td>
<td>O(lgN)</td>
</tr>
<tr>
<td>add</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(lgN)</td>
</tr>
</tbody>
</table>

- Space used
  - Vector is just key+value, no overhead
  - BST adds 8 bytes of pointers (+ balance factor?) to each entry