Binary Trees

- Linked lists: efficient insertion/deletion, inefficient search
  - ArrayList: search can be efficient, insertion/deletion not

- Binary trees: efficient insertion, deletion, and search
  - trees used in many contexts, not just for searching, e.g., expression trees
  - search in $O(\log n)$ like sorted array
  - insertion/deletion $O(1)$ like list, once location found!
  - binary trees are inherently recursive, difficult to process trees non-recursively, but possible
    - recursion never required, often makes coding 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, $N_1, N_2, \ldots, N_k$
    - $N_i$ is parent of $N_{i+1}$
  - 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

A TreeNode by any other name...

- What does this look like?
  - What does the picture look like?

public class TreeNode
{
    TreeNode left;
    TreeNode right;
    String info;
    TreeNode(String s,
        TreeNode llink, TreeNode rlink)
    {
        info = s;
        left = llink;
        right = rlink;
    }
}
Printing a search tree in order

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

```java
void visit(TreeNode t) {
    if (t != null) {
        visit(t.left);
        System.out.println(t.info);
        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 arrays of linked-lists?

### See SetTiming code

- What about ISimpleSet interface
  - How does this compare to java.util?
  - Why are we looking at this, what about Java source?

- How would we implement most simply?
  - What are complexity repercussions: add, contains
  - What about iterating?

- What would linked list get us? Scenarios where better?
  - Consider N adds and M contains operations
  - Move to front heuristic?

### What does contains look like?

```java
public boolean contains(E element) {
    return myList.indexOf(element) >= 0;
}
```

```java
public boolean contains(E element){
    returns contains(myHead, element);
}
```

```java
private boolean contains(Node list, E element) {
    if (list == null) return false;
    if (list.info.equals(element)) return true;
    return contains(list.next,element);
}
```

- Why is there a private, helper method?
  - What will be different about Tree?
What does contains look like?

```java
public boolean contains(E element) {
    return contains(myRoot, element);
}
private boolean contains(TreeNode root, E element) {
    if (root == null) return false;
    if (root.info.equals(element)) return true;
    if (element.compareTo(root.info) <= 0) {
        return contains(root.left, element);
    } else {
        return contains(root.right, element);
    }
}
```

- What is recurrence? Complexity?
- When good trees go bad, how can this happen?

What does insertion look like?

- Simple recursive insertion into tree (accessed by root)
  ```java
  root = insert("foo", root);
  ```

- Java implementation, similar to doubly-linked list
  ```java
  public class Tree {
      String info;
      TreeNode left;
      TreeNode right;
      TreeNode(String s, TreeNode llink, TreeNode rlink) {
          info = s; left = llink; right = rlink;
      }
  }
  ```

Removal from tree?

- For insertion we can use iteration (see BSTSet)
  ```java
  if (root == null) return false;
  if (root.info.equals(element)) return true;
  if (element.compareTo(root.info) <= 0) {
      return contains(root.left, element);
  } else {
      return contains(root.right, element);
  }
  ```

- What is recurrence? Complexity?
- When good trees go bad, how can this happen?

- Removal is tricky, depends on number of children
  ```java
  if (right == null) {
      root = root.left;
  } else {
      TreeNode successor = root.right;
      while (successor.left != null) {
          successor = successor.left;
      }
      root.info = successor.info;
      successor.info = root.info;
      root = root.right;
  }
  ```

- Removal is tricky, depends on number of children
- Straightforward when zero or one child
- Complicated when two children, find successor
  ```java
  if (right == null) {
      root = root.left;
  } else {
      TreeNode successor = root.right;
      while (successor.left != null) {
          successor = successor.left;
      }
      root.info = successor.info;
      successor.info = root.info;
      root = root.right;
  }
  ```

Implementing binary trees

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

- Java implementation, similar to doubly-linked list
  ```java
  public class Tree {
      String info;
      TreeNode left;
      TreeNode right;
      TreeNode(String s, TreeNode llink, TreeNode rlink) {
          info = s; left = llink; right = rlink;
      }
  }
  ```
Tree functions

- Compute height of a tree, what is complexity?
  ```java
  int height(Tree root)
  {
    if (root == null) return 0;
    else {
      return 1 + Math.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

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

  ```java
  boolean isBalanced(Tree root)
  {
    if (root == null) return true;
    return
    isBalanced(root.left) && isBalanced(root.right) &&
    Math.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 correctness
Danny Hillis

- The third culture consists of those scientists and other thinkers in the empirical world who, through their work and expository writing, are taking the place of the traditional intellectual in rendering visible the deeper meanings of our lives, redefining who and what we are.

(Wired 1998) And now we are beginning to depend on computers to help us evolve new computers that let us produce things of much greater complexity. Yet we don’t quite understand the process - it’s getting ahead of us. We’re now using programs to make much faster computers so the process can run much faster.

That’s what’s so confusing - technologies are feeding back on themselves; we’re taking off. We’re at that point analogous to when single-celled organisms were turning into multicelled organisms. We are amoebas and we can’t figure out what the hell this thing is that we’re creating.

Searching, Maps, Tries (hashing)

- Searching is a fundamentally important operation
  - We want to search quickly, very very quickly
  - Consider searching using Google, ACES, issues?
  - In general we want to search in a collection for a key

- We’ve searched using trees and arrays
  - Tree implementation was quick: O(log n) worst/average?
  - Arrays: access is O(1), search is slower

- If we compare keys, log n is best for searching n elements
  - Lower bound is \(\Omega(\log n)\), provable
  - Hashing is O(1) on average, not a contradiction, why?
  - Tries are O(1) worst-case!! (ignoring length of key)

From Google to Maps

- If we wanted to write a search engine we’d need to access lots of pages and keep lots of data
  - Given a word, on what pages does it appear?
  - This is a map of words->web pages

- In general a map associates a key with a value
  - Look up the key in the map, get the value
  - Google: key is word/words, value is list of web pages
  - Anagram: key is string, value is words that are anagrams

- Interface issues
  - Lookup a key, return boolean: in map or value: associated with the key (what if key not in map?)
  - Insert a key/value pair into the map

Interface at work: MapDemo.java

- Key is a string, Value is # occurrences
  - Code below shows how Map interface/classes work

```java
while (it.hasNext()) {
    String s = it.next();
    Counter c = map.get(s);
    if (c != null) c.increment();
    else map.put(s, new Counter());
}
```

- What clues are there for prototype of map.get and map.put?
  - What if a key is not in map, what value returned?
  - What kind of objects can be put in a map?
Replacing Counter with Integer

- With autoboxing (and unboxing) do we need class Counter?
  - What if we access a key that's not there?

```java
while (it.hasNext()) {
    String s = it.next();
    if (map.containsKey(s)){
        map.put(s, map.get(s)+1);
    } else map.put(s,1);
}
```

- What is key? What is value?
  - What if a key is not in map, what value returned?
  - Is use of get() to determine if key is present a good idea?

Getting keys and values from a map

- Access every key in the map, then get the corresponding value
  - Get an iterator of the set of keys: `keySet().iterator()`
  - For each key returned by this iterator call `map.get(key)` ...

- Get an iterator over (key,value) pairs, there's a nested class called `Map.Entry` that the iterator returns, accessing the key and the value separately is then possible
  - To see all the pairs use `entrySet().iterator()`

External Iterator without generics

- The Iterator interface access elements
  - Source of iterator makes a difference: cast required?

```java
Iterator it = map.keySet().iterator();
while (it.hasNext()){
    Object value = map.get(it.next());
}
```

```java
Iterator it2 = map.entrySet().iterator();
while (it2.hasNext()){
    Map.Entry me = (Map.Entry) it2.next();
    Object value = me.getValue();
}
```
External Iterator with generics

- Avoid Object, we know what we have a map of
  - Is the syntax worth it?

```java
Iterator<String> it = map.keySet().iterator();
while (it.hasNext()){
    Counter value = map.get(it.next());
}
```

- Iterator
```java
Iterator<Map.Entry<String,Counter>> it2 = map.entrySet().iterator();
while (it2.hasNext()){
    Map.Entry<String,Counter> me = it.next();
    Counter value = me.getValue();
}
```

Hashing: Log \(10^{100}\) is a big number

- Comparison based searches are too slow for lots of data
  - How many comparisons needed for a billion elements?
  - What if one billion web-pages indexed?

- Hashing is a search method: average case O(1) search
  - Worst case is very bad, but in practice hashing is good
  - Associate a number with every key, use the number to store the key
    - Like catalog in library, given book title, find the book

- A hash function generates the number from the key
  - Goal: Efficient to calculate
  - Goal: Distributes keys evenly in hash table

Hashing details

- There will be collisions, two keys will hash to the same value
  - We must handle collisions, still have efficient search
  - What about birthday "paradox": using birthday as hash function, will there be collisions in a room of 25 people?

- Several ways to handle collisions, in general array/vector used
  - Linear probing, look in next spot if not found
    - Hash to index h, try h+1, h+2, ..., wrap at end
    - Clustering problems, deletion problems, growing problems
  - Quadratic probing
    - Hash to index h, try h, h^2, h^2 + 1, h^2 + 2, ..., wrap at end
    - Fewer clustering problems
  - Double hashing
    - Hash to index h, with another hash function to j
      - Try h, h+j, h+2j, ...

Chaining with hashing

- With n buckets each bucket stores linked list
  - Compute hash value h, look up key in linked list table[h]
  - Hopefully linked lists are short, searching is fast
  - Unsuccessful searches often faster than successful
    - Empty linked lists searched more quickly than non-empty
    - Potential problems?

- Hash table details
  - Size of hash table should be a prime number
  - Keep load factor small: number of keys/size of table
  - On average, with reasonable load factor, search is O(1)
  - What if load factor gets too high? Rehash or other method
Hashing problems

- Linear probing, $\text{hash}(x) = x \mod \text{tablesize}$
  > Insert 24, 12, 45, 14, delete 24, insert 23 (where?)
  
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>12</td>
<td>45</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Same numbers, use quadratic probing (clustering better?)
  
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>12</td>
<td>14</td>
<td>45</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- What about chaining, what happens?

What about hash functions

- Hashing often done on strings, consider two alternatives
  
  ```java
  public static int hash(String s)
  {
    int k, total = 0;
    for(k=0; k < s.length(); k++)
    {
      total += s.charAt(k);
    }
    return total;
  }
  ```

  > Consider $\text{total} \text{ } += (k+1) \cdot \text{s.charAt(k)}$, why might this be better?
  > Other functions used, always mod result by table size

- What about hashing other objects?
  > Need conversion of key to index, not always simple
  > Ever object has method hashCode()!

Trie: efficient search words/suffixes

- A trie (from retrieval, but pronounced “try”) supports
  > Insertion: put string into trie (delete and look up)
  > These operations are $O(\text{size of string})$ 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
  > Very useful matching suffixes: suffix tree/suffix array

Trie picture/code (see TrieSet.java)

- 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, iteration?
  > Indicates word ends here
Guy L. Steele, Jr.

Co-invented/developed Scheme, continues to develop Java

If, several years ago, with C++ at its most popular, ... you had come to me, O worthy opponents, and proclaimed that objects had failed, I might well have agreed. But now that Java has become mainstream, popularizing not only object-oriented programming but related technologies such as garbage collection and remote method invocation, ... we may now confidently assert that objects most certainly have not failed.