Data Structures revisited

- Linked lists and arrays and ArrayLists and...
  - Linear structures, operations include insert, delete, traverse, ...
  - Advantages and trade-offs include ...

- We want to move toward structures that support very efficient insertion and lookup, lists can't do better than $O(n)$ for one of these: consider binary search and insert for arrays, or insert and lookup for linked lists

- Interlude: linear structures that facilitate certain algorithms: Stack and Queue (and Dequeue)
  - Restricted access linear structures

Wordladder Story

- Ladder from ‘white’ to ‘house’
  - White, while, whale, shale, ...
- I can do that... optimally
  - My brother was an English major
  - My ladder is 16, his is 15, how?

- There’s a ladder that’s 14 words!
  - The key is ‘sough’
- Guarantee optimality!

Stack: What problems does it solve?

- Stacks are used to avoid recursion, a stack can replace the implicit/actual stack of functions called recursively

- Stacks are used to evaluate arithmetic expressions, to implement compilers, to implement interpreters
  - The Java Virtual Machine (JVM) is a stack-based machine
  - Postscript is a stack-based language
  - Stacks are used to evaluate arithmetic expressions in many languages

- Small set of operations: LIFO or last in is first out access
  - Operations: push, pop, top, create, clear, size
  - More in postscript, e.g., swap, dup, rotate, ...

Simple stack example

- Stack is part of java.util.Collections hierarchy
  - It’s an OO abomination, extends Vector (like ArrayList)
    - Should be implemented using Vector
    - Doesn’t model “is-a” inheritance
  - what does pop do? What does push do?

```java
Stack<String> s = new Stack<String>();
s.push("panda");
s.push("grizzly");
s.push("brown");
System.out.println("size = "+s.size());
System.out.println(s.peek());
String str = s.pop();
System.out.println(s.peek());
System.out.println(s.pop());
```
Implementation is very simple

- Extends Vector, so simply wraps Vector/ArrayList methods in better names
  - push==add, pop==remove (also peek and empty)
  - Note: code below for ArrayList, Vector is used
    - Stack is generic, so Object replaced by generic reference (see next slide)

```java
public Object push(Object o){
    add(o);
    return o;
}
public Object pop(){
    return remove(size()-1);
}
```

Uses rather than "is-a"

- Suppose there's a private ArrayList myStorage
  - Doesn't extend Vector, simply uses Vector/ArrayList
  - Disadvantages of this approach?
    - Synchronization issues

```java
public class Stack<E> {
    private ArrayList<E> myStorage;
    public E push(E o) {
        myStorage.add(o);
        return o;
    }
    public E pop() {
        return myStorage.remove(size()-1);
    }
}
```

Postfix, prefix, and infix notation

- Postfix notation used in some HP calculators
  - No parentheses needed, precedence rules still respected
    - 3 5 + 4 2 * 7 + 3 - 9 7 + *
  - Read expression
    - For number/operand: push
    - For operator: pop, pop, operate, push

- See Postfix.java for example code, key ideas:
  - Use StringTokenizer, handy tool for parsing
  - Note: Exceptions thrown, what are these?

- What about prefix and infix notations, advantages?
Exceptions

- Exceptions are raised or thrown in exceptional cases
  - Bad indexes, null pointers, illegal arguments, ...
  - File not found, URL malformed, ...

- Runtime exceptions aren’t meant to be handled or caught
  - Bad index in array, don’t try to handle this in code
  - Null pointer stops your program, don’t code that way!

- Other exceptions must be caught or rethrown
  - See FileNotFoundException and IOException in Scanner class implementation

- RuntimeException extends Exception, catch not required

Prefix notation in action

- Scheme/LISP and other functional languages tend to use a prefix notation

  (define (square x) (* x x))

  (define (expt b n)
    (if (= n 0)
      1
      (* b (expt b (- n 1)))))

Postfix notation in action

- Practical example of use of stack abstraction
- Put operator after operands in expression
  - Use stack to evaluate
    - operand: push onto stack
    - operator: pop operands push result
- PostScript is a stack language mostly used for printing
  - drawing an X with two equivalent sets of code

  %!
  200 200 moveto
  100 100 rlineto
  200 300 moveto
  100 100 rlineto
  stroke showpage

  %!
  100 –100 200 300 100 100 200 200 moveto rlineto moveto rlineto stroke showpage

Queue: another linear ADT

- FIFO: first in, first out, used in many applications
  - Scheduling jobs/processes on a computer
  - Tenting policy?
  - Computer simulations

- Common operations: add (back), remove (front), peek ?
  - java.util.Queue is an interface (jdk5)
    - offer(E), remove(), peek(), size()
  - java.util.LinkedList implements the interface
    - add(), addLast(), getFirst(), removeFirst()

- Downside of using LinkedList as queue
  - Can access middle elements, remove last, etc. why?
Stack and Queue implementations

- Different implementations of queue (and stack) aren’t really interesting from an algorithmic standpoint
  - Complexity is the same, performance may change (why?)
  - Use ArrayList, growable array, Vector, linked list, ...
    - Any sequential structure

- As we’ll see java.util.LinkedList is good basis for all
  - In Java 5+, LinkedList implements the Queue interface, low-level linked lists/nodes facilitate (circular list!)

- ArrayList for queue is tricky, ring buffer implementation, add but wrap-around if possible before growing
  - Tricky to get right (exercise left to reader)

Using linear data structures

- We’ve studied arrays, stacks, queues, which to use?
  - It depends on the application
  - ArrayList is multipurpose, why not always use it?
    - Make it clear to programmer what’s being done
    - Other reasons?

- Other linear ADTs exist
  - List: add-to-front, add-to-back, insert anywhere, iterate
    - Alternative: create, head, tail, Lisp or
    - Linked-list nodes are concrete implementation
  - Deque: add-to-front, add-to-back, random access
    - Why is this “better” than an ArrayList?
    - How to implement?

Queue applications

- Simulation, discrete-event simulation
  - How many toll-booths do we need? How many express lanes or self-checkout at grocery store? Runway access at airport?
  - Queues facilitate simulation with mathematical distributions governing events, e.g., Poisson distribution for arrival times

- Shortest path, e.g., in flood-fill to find path to some neighbor or in word-ladder
  - How do we get from "white" to "house" one-letter at a time?
    - white, while, whale, shale, shake, ...?
Queue for shortest path (see APT)

```java
public boolean ladderExists(String[] words, String from, String to) {
    Queue<String> q = new LinkedList<String>();
    Set<String> used = new TreeSet<String>();
    for (String s : words) {
        if (oneAway(from, s)) {
            q.add(s);
            used.add(s);
        }
    }
    while (q.size() != 0) {
        String current = q.remove();
        if (oneAway(current, to)) return true;
        // add code here, what?
    }
    return false;
}
```

Shortest Path reprised

- How does use of Queue ensure we find shortest path?
  - Where are words one away from start?
  - Where are words two away from start?

- Why do we need to avoid revisiting a word, when?
  - Why do we use a set for this? Why a TreeSet?
  - Alternatives?

- What if we want the ladder, not just whether it exists
  - What's path from white to house? We know there is one.
  - Ideas? Options?

Shortest path proof

- Every word one away from start is on queue before loop
  - Obvious from code
- All one-away words dequeued before two-away...n-away
  - See previous assertion, property of queues
- Every two-away word is one away from a one-away word
  - So all enqueued after one-away, before three-away
    - How do we find three-away word?
- Word w put on queue is one-away from an n-away word
  - w is n+1(away), can't be earlier than that, why?

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, \( N_0, N_1, \ldots, N_k \)
  - \( N_i \) 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 (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?

```java
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);
    }
}
```

```java
void inorder(TreeNode t)
{
    if (t != null) {
        return;
    }
}
```
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;
}

public boolean contains(E element){
    return contains(myHead, element);
}

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 (list.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
public TreeNode insert(TreeNode t, String s) {
  if (t == null) t = new TreeNode(s, null, null);
  else if (s.compareTo(t.info) <= 0)
    t.left = insert(t.left, s);
  else t.right = insert(t.right, s);
  return t;
}
```

- 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 the idiom `t = treeMethod(t,..)` be used?

Removal from tree?

- For insertion we can use iteration (see BSTSet)
  
  ```java
  int height(TreeNode root) {
    if (root == null) return 0;
    else {
      return 1 + Math.max(height(root.left),
          height(root.right));
    }
  }
  ```

- Removal is tricky, depends on number of children
  
  ```java
  ```

Implementing binary trees

- Trees can have many shapes: short/bushy, long/stringy
  
  ```java
  public class Tree {
    String info;
    TreeNode left;
    TreeNode right;
    TreeNode(String s, TreeNode llink, TreeNode rlink) {
      info = s; left = llink; right = rlink;
    }
  }
  ```

- A simple implementation, similar to doubly-linked list

- If height is h, number of nodes is between h and $2^h - 1$
- Single node tree: height = 1, if height = 3

Tree functions

- Compute height of a tree, what is complexity?
  
  ```java
  ```

- Modify function to compute number of nodes in a tree, does complexity change?
  
  ```java
  ```

- 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.