31251 – Data Structures and Algorithms

Week 6

Luke Mathieson

This time, in 31251 DSA:

- Binary Trees
- Tree Traversals
- Expression Trees
- Binary Search Trees
- Just for “fun”: AVL Trees

Binary Trees

- A *Tree* is a graph with no cycles.
  - You can't walk through the graph and get back to your starting point without backtracking.
- A *Binary* Tree is a tree where every vertex has at most three neighbours.

3? What?

- If it has 3 at most 3 neighbours, why is it “binary”?
- We normally think of them as having an order:
  - One vertex is the root.
  - Each vertex has at most two children, and at most one parent.
  - Vertices with no children are called *leaves*.

Uses of Binary Trees

- Binary Trees find uses in many areas of computer science:
  - 3D rendering (binary space partition).
  - Networking (Binary Tries, Treaps).
  - Cryptology (GGM Trees).
  - Coding and Compression (Huffman Trees).
  - Hashing (Hash Trees).
  - Sorting (Heaps and Heapsort).
  - Searching (Binary Search Trees).
  - Parsing (Expression Trees)

- There are two basic methods for building a binary tree:
  1. Kind of like a LinkedList with with two next pointers (left and right children).
  2. Embedded in an array, where the children of the vertex at index i are at indices $2i$ and $2i + 1$.

How do we get around them?

- No matter which representation you choose, we can use versions of the same two traversals we saw with normal graphs:
  1. Breadth first - start with the root, then visit its children, then their children, then their children...
  2. Depth first - start at the root, go all the way to the bottom first, then backtrack.

Depth First Traversals

- Very amenable to recursive implementation.
- At each node we have 3 things to do:
  1. Deal with the current node,
  2. visit the left child,
  3. visit the right child.
- Gives 3 different traversals.
  1. Pre-order (deal with the current node first)
  2. In-order (deal with the current node between visiting the descendents)
  3. Post-order (deal with the current node last).

Depth First Traversal – Implementation

Pre-order traversal, recursive:

```
preorderTraversal(Node n){
  if (n == null) return;

  visit(n);
  preorderTraversal(n.leftChild());
  preorderTraversal(n.rightChild());
}
```

Depth First Traversal – Implementation

We can switch from recursive to iterative by swapping the implicit use of the call stack with an explicit stack.

Pre-order traversal, iterative:

```
preorderTraversal(Node n){
  Stack<Node> s = new Stack<Node>();
  Node current = n;

  while (current != null){
    visit(current);
    if (current.rightChild() != null)
      s.push(current.rightChild());
    if (current.leftChild() != null)
      s.push(current.leftChild());

    current = stack.pop();
  }
}
```

Depth First Traversal – Implementation

In-order traversal, recursive:

```
inorderTraversal(Node n){
  if (n == null) return;

  preorderTraversal(n.leftChild());
  visit(n);
  preorderTraversal(n.rightChild());
}
```

Depth First Traversal – Implementation

In-order traversal, iterative:

```
inorderTraversal(Node n){
  Stack<Node> s = new Stack<Node>();
  Node current = n;

  while (current != null || !s.isEmpty()){
    if (current != null){
      s.push(current);
      current = current.leftChild();
    }
    else {
      current = s.pop();
      visit(current);
      current = current.rightChild();
    }
  }
}
```

Post-order traversal, recursive:

```
postorderTraversal(Node n){
  if (n == null) return;

  preorderTraversal(n.leftChild());
  preorderTraversal(n.rightChild());
  visit(n);
}
```

Post-order traversal, iterative:

```
postorderTraversal(Node n){
  Stack<Node> s = new Stack<Node>();
  Node current = n;
  Node last = null;

  while (current != null || !s.isEmpty()){
    if (current != null){
      s.push(current);
      current = current.leftChild();
    }
    else{
      if (s.peek().rightChild() != null &&
            s.peek().rightChild() != last){
        current = s.peek().rightChild();
      }
      else{
        current = s.pop();
        visit(current);
        last = current;
      }
    }
  }
}
```

- Visits vertices according to their level in the tree ("left to right, top to bottom").
- Simple to implement iteratively using a queue.

Breadth First Traversal - Implementation

```
breadthFirst(Node n){
  Queue<Node> q = new Queue<Node>();

  q.add(n);

  while (!q.isEmpty()){

    Node current = q.remove();
    visit(current);

    if (current.leftChild != null)
      q.add(current.leftChild());
    if (current.rightChild != null)
      q.add(current.rightChild());

  }
}
```

Binary Search Trees

Binary Search Trees

- Binary Search Trees (BSTs) are a simple data structure that allows fast insertion, removal and lookup of the elements they store.
- They are an ordered data structure, but because they use a binary tree, you don't have to reshuffle everything to put something new in. (Unlike array-like data structures, for example.)
- They follow a simple rule to find or insert:
  - If what you're looking for (or what you have) has a smaller key than the current vertex, go to the left. Otherwise, go to the right. (Special case: duplicate keys)
- Insertion then is just traversing the tree to the bottom and adding the new element wherever you stop.
- Finding something mimics binary search, if you get to the bottom without finding it, it's not there.

Operation	Average Case	Worst Case
Space	$O(n)$	$O(n)$[1]
Insert	$O(\log n)$	$O(n)$
Remove	$O(\log n)$	$O(n)$
Find	$O(\log n)$	$O(n)$

At the cost of more complicated code, there are several self-balancing BST data structures which reduce the worst case insert, remove and find to $O(\log n)$: 2-3 Trees, Red-Black Trees, AVL Trees, Splay Trees and others. (Some material at the end if we have time, or to peruse at your leisure.)

[1] Using the array embedding, this can actually end up as $O(2^n)$.

Some Uses of Binary Trees

Representing Grammars with Trees

- Many things in computer science can be expressed in terms of *Formal Grammars*.
  - (We don't need to know what they are though).
- In particular, expressions that have syntax can be modelled with grammars.
  - e.g. every programming language (and much more).
- One way of showing how a concrete expression derives from a grammar is by using a tree.
  - ... and for certain types of expression grammars, binary trees!

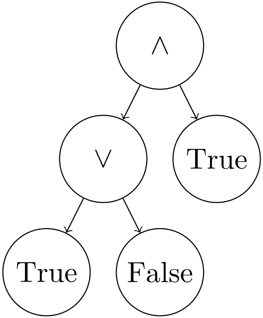
- We can take Boolean expressions as an example. They have simple rules:
  1. True is a Boolean expression.
  2. False is a Boolean expression.
  3. If A a Boolean expression, $\neg A$ is a Boolean expression.
  4. If A and B are Boolean expressions, $A \wedge B$ is a Boolean expression.
  5. If A and B are Boolean expressions, $A \vee B$ is a Boolean expression.

- Just for those that are interested, a grammar for that looks like:

$$S \to \neg S \mid S \wedge S \mid S \vee S \mid \mathrm{True} \mid \mathrm{False}$$

- You can build more complicated ones that build in operator precedence and associativity too.

- We can convert these rules to a binary tree structure.
- For example, the expression $(\mathrm{True} \vee \mathrm{False}) \wedge \mathrm{True}$ can be represented by:



Why is this useful?

- Given an expression as text, it is not immediately obvious how to begin computing its value.
- You need to read the whole thing, then decide which bits you are going to do first.
    - Operator precedence is important! ($3 \times 2 - 1$ vs $3 \times (2 - 1)$)
- If we can quickly build a tree like this, then we can *recursively* evaluate it!
    - It becomes a simple divide and conquer algorithm!
- We can also traverse it looking for other properties (very useful when compiling code).

Prefix, Infix and Postfix

- Another thing we can do is turn the expression back into a string in different ways.
  1. Prefix notation results from a pre-order traversal of the tree,
  2. Infix notation results from an in-order traversal, and...
  3. (you guessed it) Postfix notation results from a post-order traversal of the tree.
- Prefix and Postfix are unambiguous, and don't require operator precedence or associativity rules to parse.
  - So converting an infix expression (what we meat-bags normally write in) to postfix or prefix can make things easier for the computer (or better still easier for the computer programmer).

A More Balanced Binary Search Tree - AVL Trees

Better Balance = Less Falling Over

- Binary Search Trees work well if the tree is balanced.
- ... but this doesn't always happen.
- We can correct this by doing some extra work to maintain balance.

AVL Trees do extra work at each insertion and deletion, to maintain good running times no matter what:

Operation	Average Case	Worst Case
Space	$O(n)$	$O(n)$[2]
Insert	$O(\log n)$	$O(\log n)$
Remove	$O(\log n)$	$O(\log n)$
Find	$O(\log n)$	$O(\log n)$

[2] Same caveat as the BST.

How do they balance things?

- AVL trees keep things balanced by maintaining an invariant at each vertex: the balance factor.
- The balance factor of a vertex v is the height of the left subtree minus the height of the right subtree.
- The balance factor of a vertex can be -1, 0 or 1. Anything else, and the tree needs rebalancing.
- Because inserting a new element only adds one vertex, the balance factor will only get as bad as -2 or 2, before something is done.

Insertion into AVL Trees

- First we insert as normal with a BST.
- Then we need to check if the height of the subtrees of the ancestors of the newly inserted vertex are okay.
- The balance factor of any vertex is the height of its left subtree minus the height of its right subtree.
- If everything is okay, the balance factor is -1, 0 or 1.
- If we always do our check from the bottom, we also know that if the tree is unbalanced, it will be 2 or -2.
- We can store the height at each vertex, then we only need to update the relevent ones when we add a new child.

- A tree rotation is an order invariant operation on binary trees.
- It changes the structure, but the traversal order remains the same.
- A tree rotation can be left or right.
- Given a vertex x, with a right subtree γ, and a left child y which has subtrees α and β, we can rotate right to get y with left subtree α, and right child x which has subtrees β and γ.
- A left rotation is just the reverse operation.

Back to Insertion

If we get a vertex x with balance factor 2:

1. We look at the left child y (this must be the larger one).
2. If it has balance factor -1, it "leans to the right".
    1. We rotate left the child y and its right child z. This makes z the left child of x.
    2. Then we rotate right z and x to get a balanced tree.
3. Otherwise
    1. Rotate right with y and x to get a balanced tree.

Back to Insertion

The -2 case is analogous:

1. If y has balance factor 1
    1. Rotate right with z and y.
    2. Rotate left with z and x.
2. Else
    1. Rotate left with y and x.

Back to Insertion

- This leaves the subtree with balance factor -1, 0 or 1, depending on the exact balance factors of its subtrees (but entirely predictable).
- Then we continue up the tree checking.
- So we only need to follow a path to the root, doing at most two rotations at each step – turns out to not be that expensive.

- We need to keep track of a couple of things:
  1. Let x be the vertex we want to delete.
  2. y is a vertex whose value we will move.
  3. and z is the actual vertex we remove.

Deletion

1. If x is a leaf, or has one child $z := x$.
2. Otherwise
   1. Find the largest value in the left subtree, or the smallest in the right subtree of x, this will be y.
   2. Copy the value at y over the value at x.
   3. Now $z := y$.
3. If z has a subtree, attach it to z's parent in its place, and delete z (or set the child to the root if z is the root).
4. Starting with z's parent, rebalance the tree.