Thinking Points

Where have you met trees before in your courses?

Storing a sequence of N integers in a linked list takes $64N$ bits (assuming 32-bit words) e.g. each node stores the integer and a pointer. Given this, how much memory would we need to store a tree containing N integers.

I want to store prerequisites of CS courses to answer the query *what prerequisites does X have*. What data structure should I use?

I want to play a game. At the start of each lecture, each student thinks of a number and all numbers get saved. At the end I try and guess one of those numbers, if I'm right I get all the chocolate, if I'm wrong you do.

- Should I use a linked list or a binary search tree?
- What if I also had to guess the number of times it had been added?

Lecture 9
Advanced Data Structures Part 2
Trees

COMP26120

Giles Reger

November 2019

Last Time

Abstract Data Types

- Describe desired behaviour
- Exist in the problem domain
- Get implemented by data structures

Hash Tables

- Implement the Dictionary ADT
- Need a good hash function and collision strategy
- Amortized average constant time insertion/lookup
- Large-ish space overhead, worst case insertion/lookup $O(n)$

Trees

May be the natural data structure for a particular domain.

Standard examples included family trees, taxonomies, syntax trees, file systems etc

Can also be a useful structure for storing and retrieving data, particularly when ordered and balanced (see later)

Tree ADT

A node is a simple ADT that has a single `element()` operation

A tree T supports the following fundamental operations:

`root()`:	returns the root of the tree
`insert(e)`:	inserts element e into the tree
`remove(v)`:	removes node v from the tree
`find(e)`:	returns the node storing element e or NULL if e is not stored in the tree
`parent(v)`:	returns the parent of node v (error if v is the root)
`children(v)`:	returns a set containing the children of node v
`elements()`:	returns a set of all of the elements in the tree
`isExternal(v)`:	returns true if the node v has no children in the tree. An external node is also often referred to as a *leaf*

Binary Tree ADT

The most common kind of tree.

Assumes that there are at most two children.

Replaces `children` with `left` and `right` operations

May add a `sibling` operation

Definitions (lots of reading)

The depth of a node v in a tree is the number of ancestors of v excluding v itself. The height of a tree is the maximal depth of an external node. Note that one applies to nodes and the other to trees.

A proper or full binary tree is one where all nodes have either 0 or 2 children. We generally assume proper trees (without less of generality).

A complete tree is one where all levels are full, possibly apart from the last where all nodes are as far left as possible. A perfect tree is a proper tree where all external nodes have the same depth.

An ordered binary tree is one where all elements in the left sub-tree are ‘smaller’ than the elements in the right sub-tree and the stored element, and the stored element is ‘smaller’ than the elements in the right subtree

Later we will define a notion of a balanced tree but this is specific to a certain kind of self-balancing tree

Vector/Array Representation of Binary Trees

We can use a vector/array of size N to store a tree of height $\log_2 N$

- Root node is at $A[0]$
- Parent of non-root node $A[k]$ is $A[(k-1)/2]$
- Left child of node $A[k]$ is $A[2k+1]$
- Right child of node $A[k]$ is $A[2k+2]$

If we run out of space we need to *resize and reinsert* as in other array-based approaches

What if the tree is *sparse*? (what does that mean?)

What is the space complexity?

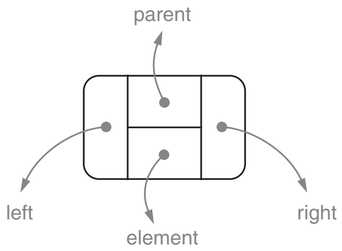
Linked-Structure Representation of Binary Trees

A node stores a value and two pointers (left and right) to child nodes

The empty tree is the NULL pointer

Like a linked list where each node has two ‘next’ nodes instead of one

May want to store backward ‘parent’ link (e.g. like a doubly-linked list)



Traversal

A traversal of a tree is a systematic way of visiting each node. There are three main kinds of traversal that have different applications:

Preorder - Visit a node and then its children (in order)

Postorder - visit a nodes children (in order) and then the node

Inorder - (Applies to ordered trees only), visit ‘smaller’ children (on the left) then the node and then ‘larger’ children (on the right)

When might we want to use each kind of traversal?

(see Goodrich and Tamassia for pseudo-code)

Search in Unordered Trees

In an unordered tree there are two ways we can perform a search.

Depth-first search: Search a node's children before its sibling.

Breadth-first search: Search all nodes at level k before nodes at level $k+1$.

Usually search happens from 'left to right' but does not need to.

What is the time complexity? What about the space complexity?

When might we want to use each kind of search?

We also have the notion of depth-limited DFS used iteratively, why?

Binary Search Tree

A binary search tree is an ordered binary tree

It has a special name as we can now efficiently perform binary search

Given an element e to search for we compare it to the element stored at the current node with the possible cases:

1. It is the same, we are finished, or
2. It is *larger*, we recurse on the left subtree, or
3. It is *smaller*, we recurse on the right subtree

This is similar to binary search in an ordered list

What is its complexity? What is the worst case?

Self-Balancing Trees

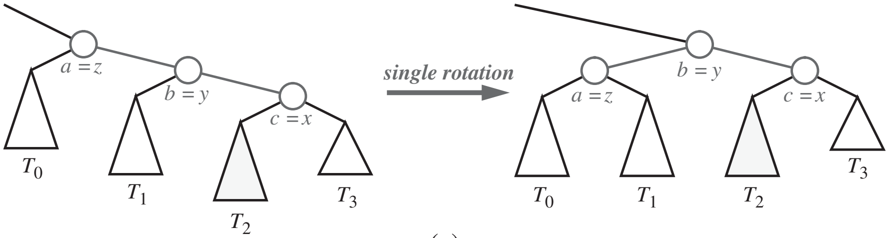
Many tree operations are $O(h)$ where h is the height of the tree. The worst case is where h is close to n and we want to avoid this.

The general idea of a self-balancing tree is to maintain a kind of balance on the tree that gives us some assurances about the maximum height of the tree.

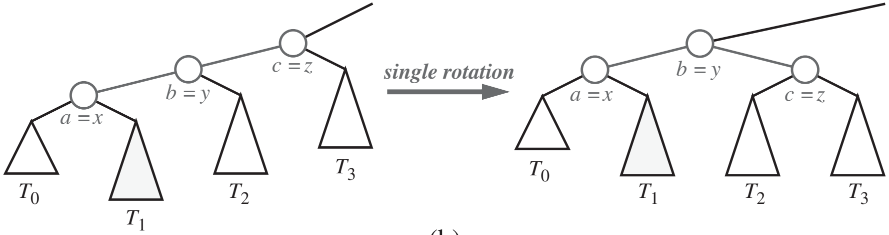
The notion of balance is different for different kinds of self-balancing tree.

Balance is achieved by performing rotations on the tree that have the effect of reducing the height whilst preserving the ordering property.

Single Rotations



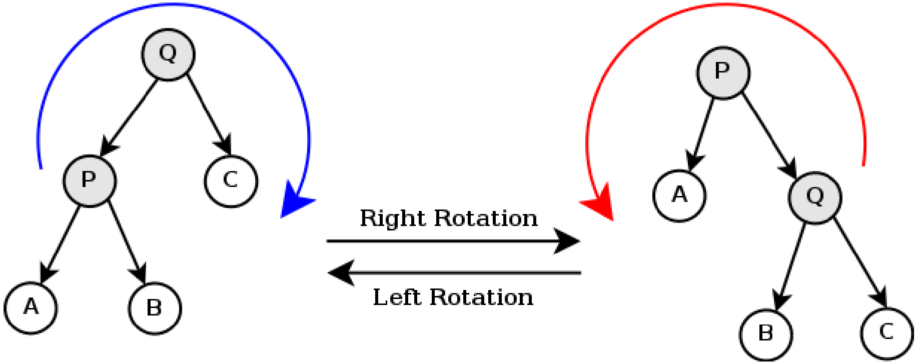
(a)



(b)

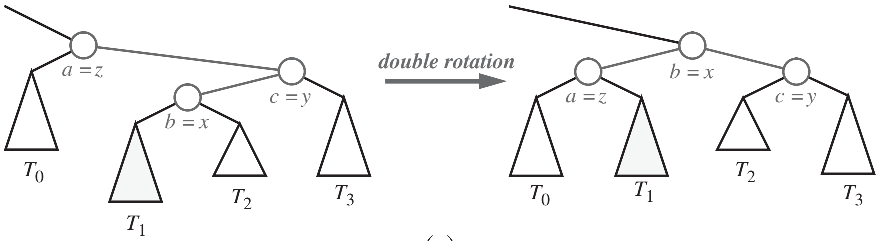
(Image from Goodrich and Tamassia)

Rotation Direction

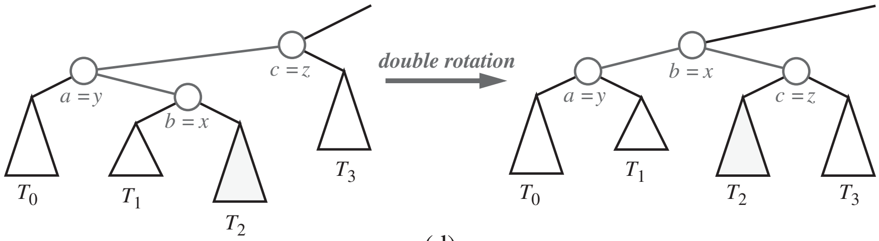


(Image from Wikipedia)

Double Rotations



(c)



(d)

(Image from Goodrich and Tamassia)

Height-Balance Property

Definition (Height-Balance Property)

For every internal node v in T, the heights of the children of v may differ by at most 1. That is, if a node v in T has children x and y then

$$|\text{height}(x) - \text{height}(y)| \leq 1$$

We can also define this by a useful property of nodes.

Definition (Balance Factor)

The balance factor of a node n is

$$\text{bf}(n) = \text{height}(\text{right}(n)) - \text{height}(\text{left}(n))$$

The height-balance property holds if for each node n, $\text{bf}(n) \in \{-1, 0, 1\}$.

AVL Trees

An AVL tree is a binary search tree with the height-balance property.

Operations are the same as for BST apart from the need to maintain the height-balance property invariant.

We do this by an extra retracing step after altering the tree.

Maximum height of AVL tree

Theorem

The height of an AVL tree storing n items is $O(\log n)$

Proof idea:

- We compute a lower bound on the number n_h of internal nodes an AVL tree with height h must have
- It is easy to see that $n_1 = 1$ and $n_2 = 2$
- In the general case, $n_h = 1 + n_{h-1} + n_{h-2}$ (look familiar?)
- Simplifies to $n_h > 2n_{h-2}$, which implies that $n_h > 2^{\frac{h}{2}-1}$
- Finally, $\log n_h > \frac{h}{2} - 1$ and $h < 2 \log n_h + 2$

So as long as we maintain the height-balance property we get $O(\log n)$ operations (as long as maintaining the property is not too expensive).

Retracing

Idea: travel up from an updated node and perform rotations if balance factor becomes left heavy (-2) or right heavy (+2)

There are four cases for a node A and its parent B:

Left heavy

1. A is the left child of B, $\text{bf}(B) = -2$ and $\text{bf}(A) = 0$ - do LeftRight
2. A is the left child of B, $\text{bf}(B) = -2$ and $\text{bf}(A) = -1$ - do Right

Right heavy

3. A is the right child of B, $\text{bf}(B) = 2$ and $\text{bf}(A) = 0$ - do RightLeft
4. A is the right child of B, $\text{bf}(B) = 2$ and $\text{bf}(A) = 1$ - do Left

Worst case complexity? Average?

Summary

Tree ADT

Tree representations (vector or linked)

Tree properties, traversal, and search

Self-balancing trees

AVL tree (height-balance property, rotations)

Red-Black tree is similar to AVL with a different notion of balance

Next time - Heaps and Skip Lists