Trees:

1. A tree can be empty
2. A single node is a tree
3. If n is a node and T1, …, Tk are trees with roots n1, …, nk, then construct a new tree with n the parent of n1, …, nk, n is the root of a tree.

Vocab:

Parent/child/sibling/ancestor/descendant

Root/branch/leaves/path/length of path (number of edges)

Height of a tree (longest path from a root to a leaf)

Height of a node: ?

Depth of a node: (number of node levels from the root)

Leftmost child, right sibling, a list of lists

Binary tree: 0-2 children. Efficiency of sorted array with benefits of dynamic memory allocation. Smallest on the left.

Expression tree: internal nodes are operators, leaves are operands

Binary Space Partitioning tree: used in game engines to determine which spaces to render

Huffman encoding: build a tree of character frequency to enable an optimal encoding for high-occurrence characters.

BSTrees:

Remove():

if (isLeaf(p))

prune p;

else if (leftChild(p) == NULL)

replace p with right child of p

else if (rightChild(p) == NULL);

replace p with left child of p

else // two children

x = leftchild of p

while (rightChild(x) != NULL)

x = rightChild(x);

replace p with x

delete x

Huffman encoding

E 100

A 50

R 25

S 20

P 10

Pick the two smallest values. P on the left, S on the right. P + S = PS (30), which becomes the root node.

PS 30

PSR 55

ARPS 105

EARPS

E (100) A (50) R (25) P (10) S (20)

Priority Queue ADT:

Collection of items where insertions and removals are based on a priority value; higher priority items are removed first.

Operations: insert(), remove()/removeMin(), peek()

Efficiently implemented with an array of queues or as a binary heap.

Binary heap ADT:

Binary tree that is filled left to right and is complete (no holes in the middle; full at all but the lowest level)

A heap node is always smaller than its children (higher pri has lower values)

Tree indices in an array

0-based:

Parent of n: (n – 1) / 2

Left child of n: (2(n+1) – 1)

Right child of n: (2(n+1) )

Insert: while x is less than parent(x), find(parent of x) and swap(x, parent)

Delete/reheapification (need to reheapify)

Building an entire heap, one item at a time, is O(n log(n))

Building an entire heap when all the items are available is O(n)

Graph is a set of edges and nodes AKA vertices.

In some cases, each edge has a direction, which is known as a directed graph or digraph.

Edges can have labels that represent some value.

Typical ways to implement graphs:

Adjacency matrix (2d array) stores graph relationships (1s show a path, 0s don’t)

Multiplying the matrix by itself shows the number of paths to a node

Adjacency list: linked list of nodes, each containing a linked list of edges

If u is connected to v (u -> v), v is adjacent to u

Cycle – connected nodes that wrap around to the starting node

Dijkstra’s algorithm – shortest path

It’s a greedy algorithm – what works in the short run should work in the long run.

Matrix:

C 1 2 3 4 5

1 - 50 20 - 30

2 - - - 10 5

3 – 20 – 40 -

4 - - - - -

5 - - - 25 -

Tables stores the shortest distance between the node or nodes. C is the ‘cost array’, AKA the graph.

// Update route table T with the minimum cost to traverse edges

// In the beginning, ‘v’ is the source vertex

// On subsequent iterations, ‘v’ is reassigned to the smallest of the unvisited vertices

// From the cost table, the connected nodes of V are found and calculated.

min(T[w].dist, T[v].dist + C[v][w] (50)

T[v].dist 1 2 3 4 5

V = 1, W = 2, 3, and 5) 0 50 20 - 30

V = 3, W = 2 and 4) 0 40 20 60 30

V = 5, W = 4) 0 40 20 55 30

V = 4, W = //) 0 40 20 55 30

V = 2, W= 4) 0 40 20 50 30

Save and aggregate the Vs

T[v].path 1 2 3 4 5

0 0 0 0 0

V = 1, W = 2, 3, and 5) 0 1 1 3 1

V = 3, W = 2 and 4) 0 3 1 5 1

V = 5, W = 4) 0 3 1 2 1

V = 3: W = 2, T[w].dist = 50, T[v].dist + C[v][w] = 20 + 20 = 40

V = 5: W = 4 (the only adjacent edge of 5; see original table)

Start with a node. Traverse the row looking for the non-infinity value.

For 1 to nodes - 1

Choose V (the smallest distance of all the not-visited nodes)

Mark V as visited.

For each w adjacent to v

If (w is not visited)

T[w].dist = min(T[w].dist, T[v].dist + C[v][w])

Midterm question: Run dijkstra’s algorithm on a provided graph, sketch out the high-level pseudo code, and come up with a T table. You may write out the pseudocode on a page of notes.

Depth-first: re-draws graph to make it more tree-like. Goes as deep into the graph as possible, then back up again.

calls depth-first-search-recursive

depth-first-search:

mark all nodes not visited

for v = 1 to n

if (! Visited(v))

depthFirstRecursive(v);

depth-first-search-recursive

mark v visited

print v or whatever

for each vertex w adjacent to v

if (! Visited(w))

depth-first-search-recursive(w);

depth-first spanning tree/forest – follow each node, smallest to largest, until it ends.

Different types of edges when spanning graphs as trees:

Tree edges, cross edges, back-edges,

Bi-connectivity

Disconnected: no vertices ; removal of a node will disconnect the graph

Articulation point: a node whose removal disconnects a graph

Euler circuit: has a path that visits every edge exactly once

Topological sort

Beadth-first-search

Mark all nodes as not visited

For (v = 1 to n)

If (v is not visited)

Bread-first(v);

Breadth-first(v)

Mark v as visited

Q.enqueue(v)

While (! Q.isEmpty())

X = Q.dequeue();

For each vertex w adjacent to x

If (! W.visited())

Q.enqueue(w);

w.Visited = true;

AVL tree – balanced BSTree

BSTree with the balanced condition that for every node in the tree, the height of the left and right subtree of every node can differ by at most one (note – height of node kept in node)

Height of an empty subtree is -1

For each insertion into an AVL tree, if needed to balance, exactly one rotation (either single or double (two singles)) is performed.

4 kinds of rotations:

Single: left-left (insert at left subtree, left child of the root of the unbalanced subtree) and right-right (right subtree, right child of the root of the unbalanced subtree)

Double: left-right and right-left (see yellow paper for details)

B-tree of order M:

The root is either a leaf or has between 2 and M children

All non-leaf nodes have between M/2 and M children

All leaves are at the same depth

2-3 trees: B-tree of order 3. Each interior node has 2 or 3 children

Log 3 (n) <= T.height <= log 2 (n)

Node in 2-3 tree loos like

Small Large

Left middle right

Items < S found in left subtree

Items > S and < L found in middle subtree

Items > L found in right subtree

Tree grows from the bottom up

Might need to create a new root

20, 50, 100

For 3 potential values in a node, do split operation:

Item in middle moves up, other 2 items become left and right children.

Trie – comes from retrieval

One way to store a dictionary (can store any set of n distinct bitstrings)

Empty tree:

Arrays of letters from the alphabet, each letter pointing to the next. Figure out why this is cool.

A b c d e f g … z $

Midterm:

Extend a lab by adding a function

ADTs and ADSes (heaps, priority queue, graphs, 2-3 trees, BTrees, AVL trees) Huffman encoding, dijkstra’s

Cheat sheet – handwritten notes, 1 piece of paper

Inheritance (single) – class is derived from one base class

Objects are defined as extensions of previously-defined classes. The extension, the derived class, inherits all members (functions and data).

UML

Aggregation/composition

General aggregation:

One (white-diamond + ‘1’) tree holds many (\*) nodes

Composition:

One (black-diamond + ‘1’) tree holds many (\*) nodes

Design Principles:

1. SRP – Single Responsibility Principle: a class should have one reason to change.
2. OCP – Open-closed principle: software entities (classes, fns, etc) should be open for extension but closed to modification
3. LSP – Liskov Substitution principle: subtypes must be substitutable for their base types
4. DIP – Dependency inversion principle: abstractions should not depend on details; details should depend on abstractions.

Nov 7, 2007

Hashing – an array with keys.

Goal is a 1:1 mapping; impossible, so collision resolution is required.

Often, items are not removed from hash tables

Open hashing: store items separately from the hash table. Each hash value points to a collection of values, stored somewhere (LL, BST, FS, etc). h(x) = x % whatever

Closed hashing: store items in the array. Probing is used to resolve collisions. Linear probing: distance from h(x) on ith collision. Primary clustering: when items cluster; causes performance problems.

Note: the array size should be a prime number (allows modulus remainders to be unique)