unit 2

## Binary Search Trees

Each node has 2 children and is ordered from left to right (smallest to largest)

**How to build a BST**

if(newKey<currentKey)

go left

else(>=)

go right

**3 ways to ensure a balanced BST**

1. Dynamic self-balancing: every time a node is added, the tree will check whether it is still complete. If not, it will trigger a rebalancing mechanism.
   1. examples: AVL trees, red-black trees
2. Amortization: can have a balancing method that gets called at other times, to amortize the costs of inserts.
3. Randomization: maybe the data already has a fairly random distribution? This will result in a fairly balanced tree.

**Node and ADT:**

struct Node

{

type key;

Node \*parent; //implementation dependent

Node \*leftChild;

Node \*rightChild;

};

**BST ADT:**

private:

Node root

Node insertHelper(Node, value)

Node searchHelper(Node, value)

Node deleteHelper(Node, value)

Node getMinValue(Node)

Node getMaxValue(Node)

void destroySubtree(Node)

public:

init() //constructor

insert(value)

search(value)

delete(value)

disp()

deleteTree()//destructor

**Recursion**

We are used to functions calling other functions. For example, main calls foo0, and foo0 calls foo1, and so on.

main()

{

foo0(…);

}

foo0()

{

foo1(…);i

}

There is another approach to designing programs: a function can be called, and then if certain ondition is met, the function calls another *instance of itself.* This is called recursion.

For a proper recursive function, the algorithm needs to eventually reach a *base case*.

*Example: write a recursive function to solve n!*

n! = n \* n-1 \*… \*1

int fact(int n)

{

if (n>1)

return n \* fact(n-1);

else

return 1;

}

execution:

fact(3)

if(n>1)

return n\*fact(n-1)

fact(2)

if(n>1)

return n\*fact(n-1)

fact(1) //we have made 3 calls, but haven’t finished one yet.

//functions are stacked on top of the function stack and

//are now going to be popped off

if (n>1)

else

return 1;

fact(2)

return n\*fact(n-1)

//2\*1

fact(3)

return n\*fact(n-1)

//3\*1

thus main would receive 6.

**Recursion w/ BST**

Because of the self-similarity nature of trees, a recursive approach is an obvious choice

void f(Node n)

{

if(n) // check if node exists

{

f(n->next)

cout << n->key << endl;

f(n->right);

}

}

**Traversals**

With a linear data structure like an array or a Linked List, it is assumed we display from beginning to end

with a tree, we have choices. 3 conventions:

pre-order: root, left, right

4, 2, 7, 5, 9

in-order: left, root, right



2, 4, 5, 7, 9



post-order: left, right, root

2, 5, 9, 7, 4



**Continuing ADT for BST**

**BST ADT:**

private:

Node root

Node insertHelper(Node, value)

Node searchHelper(Node, value)

Node deleteHelper(Node, value)

Node dispHelper(Node, value)

Node getMinValue(Node)

Node getMaxValue(Node)

void destroySubtree(Node)

public:

init() //constructor

insert(value)

search(value)

delete(value)

disp()

deleteTree()//destructor

void :: BST::printTree()

{

printTreeHelper(root); //remember that root is private and cannot

//be accessed by the user

cout << endl;

}

voidBST::printTreeHelper(Node \*currNode)

{

if(currNode)

{

printTreeHelper( currNode->left);

cout << “ “ << currNode->key;

printTreeHelper(currNode->right);

{

}

**Insert Node**

Given a key value, insert a node to the next available location, ensuring that BST properties are maintained

void insert(int data)

insertHelper(root,data)

Node\* insert(Node \*crawler, int data)

1. nothing in tree OR if a leaf has been reached
   1. base case
   2. create node

Node \*newNode = new Node;

newNode->key = data;

newNode->left = NULL;

newNode->right = NULL;

1. else if(data>crawler->key)

crawler->right = insertHelper(crawler->right,data);

1. else if(data<crawler->key)

crawler->left = insertHelper(crawler->left,data);

1. return crawler;

return

e.g start with empty tree

insert(8)

insertHelper(root, 8)

//take branch 1: add new Node and make it root

insert(0)

insertHelper(root,0)

//take branch 3:

crawler->left = insertHelper(crawler->left, 0)

take branch 1:

create new node

return new node

crawler->left = new node

return root;

**Insert**

bool search(value)

Node = searchHelper(root, value);

Node searchHelper(crawler, value)

//check for empty tree

if crawler == NULL

return NULL

//check if search value is in the current node

if crawler->key == value

return crawler;

if value<crawler->key

return searchHelper(crawler->left, value)

return searchHelper(crawler->right, value);



**Delete**

Given a tree and a key value, delete a node.



delete(value)

deleteHelper(root,value)

deleteHelper(Node, value)

//Algorithm

//1. node == nullptr

// either empty tree, or finished deletion

//return nullptr;

//2. if value<node

//recursively traverse to the left child

//3. if value>node

//recursively traverse to right child

//4. Node found

//4 Possible cases

//1. Node has no children

// delete node

//Node = nullptr

//2. Node has only right child

//replace node with the right child

//3. Node has only left child

//replace node with the left child

//4. Node has two children

//find the min value of the right subtree

//replace the node with min

//move min’s right child to replace min

delete(8)

root = deleteHelper(root, 8);

minOfRight = min(node->right);

node->key = minOfRight->key;

node->right = deleteHelper(node->right, minofRight->key);

**Destructor**

BinaryST::~BinaryST()

{

if root(!=nullptr)

destructorHelper(root);

root = nullptr;

else{}

}

//recall the different traversal algorithms.

//post-order: left, right, root

void BinaryST::destructorHelper(Node \*n)

{

if(n->leftChild != nullptr)

destructorHelper(n->leftChild)

if(n->rightChild != nullptr)

destructorHelper(n->rightChild)

delete n;

}

## Balanced Search Trees

A balanced search tree allows for an O(logn) performance

**Red-Black Trees**

Rules:

* A node is either red or black (color can change as tree re balances)
* Root node I sblack
* every leaf node is black, empty and has nullptr children
* if a node is red, both of its children must be black
* for every node in the tree, all paths to a descendant leaf node must pass through same number of black nodes

For red-black trees, we define a set of special operations:

* Recolor a node
  + red or black
* rotate – changes height of tree
  + rotate right
  + rotate left

**Operations of RB trees**

*insertNode()*

1. set color of new node to red
2. set children to point to null nodes
3. resolve RB violations by recoloring or rotating

Always initialize new node to red

Insert Approach:

* insert a node just like you would into a bst
* check if parent node is red, if so, a repair is needed. one of 6 possible scenarios:

1. Parent of new node is left child
   1. uncle node is red
   2. uncle node is balck and new node is a right child
   3. uncle node is black and new node is a left child
2. Parent of new node is RC
   1. same 3 scenarios

case 1: uncle node is red

* color parent node black
* (z->parent->color = black)
* color uncle node black (z->parent->parent->right->color = black)
* color grand parent red (z->parent->parent ->color = red)
* move node up to grandparent (z = z->parent->parent)

case 2: uncle node is black and z node is RC

* check if uncle node is black
  + z->parent->parent->right->color == black
* check if z node is right child?
  + z==z->parent->right
* if so,
  + set z to point to its parent
    - z=z->parent
  + left rotate on z
    - leftRotate(z)

case 3: uncle node is black *and* z is a left child

* color z parent node black
* color the z grandparent node red
* righr rotate on grandparent

## Graphs

* Directional graphs (A->B)
* Examples:
  + facebook friends (non-directional)
  + twitter follows (directional)
* Definition: graph G (V, E)
  + V: set of vertices (like num of nodes in the graph)
  + E: (a,b) connection of two vertices (number of edge)
    - order does not matter if it is non-directional
    - directional -> <a,b> nondirectional -> (a,b) or (b, a)
* Complete Graph
  + When all pairs of vertices are connected
  + #n vertices
    - Undirectional [n(n-1)]/2
    - directional n(n-1)
* Degree = number of edges for each vertices
  + non-directional: #of degrees = 2\*E
* Regular graph:
  + Every vertices has the same degree
* Directional Degrees:
  + indegree: receiving edges
  + outdegree: sending edges
* Walk = sequenece of edges
* Trail = A->D -> B -> A -> D WHAT
* Path =
  + Trail
  + Each vertices should be different
* Directional Chart thing
  + Row sum = out degree
  + col sum = in degree

**ADT**

Objects: V & E

Functions:

createGraph

InsertVertex(V)

insertEdge(V1, V2, w)

where w is the weight of the edge

deleteVertex(V)

 deleteEdge(v1, v2)

printGraph()

search(V)

struct Edge

{

vertex \*V;

int distance;

};

struct Vertex

{

string name;

};

class Graph

{

private:

List <Vertex> vertices;

}

**Graph Recap:**

* a collection of vertices connected by edges G= {V,E}
* each vertex contains a “key” and a list of edges (connections to other vertices
* List of edges is stored in an adjacency matrix or adjacency list
* Applications:
  + Navigation
  + Social Networking Applications
    - Facebook friends
    - Twitter follows
* Unlike a BST< there are no defined relationships between vertices
  + cannot say v0 > v1 bc of parent child relationship
  + all relationships have to be explicitly set

**STL Vectors**

Standard Template Library – very widely used set of template classes. Includes most of the common data structures (list, queue, stack, vector, etc.)

Template class – a class that works generically on any time (primitive or user defined

Primitive type example:

vector<int> v0;

User defined type example:

struct myStruct

{

int numbers;

string words;

};

vector<myStruct> vectorOfStructs;

**ADT**

-undirected – Vx->Vy implies Vy->Vx (no directionality)

- weighted – every edge has a weight

private:

vertices

(edges) //contained in vertices

public:

insertVertex(value);

addEdge(startValue, endValue, weight);

deleteVertex(value);

deleteEdge(startValue, endValue);

displayGraph();

//no search method

**C++ Definitions**

private:

vector<vertex\*> vertices;

struct vertex

{

string key;

vector<edge> adj; //adjacency list

};

struct edge

{

vertex \*v;

int weight; //optinonal: weighted vs unweighted

};

**Insert**

addVertex(key) – inserts a vertex into a graph with no edges (empty adjacency list)

Algorithm:

* 1. Search to ensure no duplicate key already exists
  2. Create a vertex with key value

void Graph :: addVertex(string n)

{

//1.:

bool found = false;

for (int x = 0; x < vertices.size(); x++)

if(vertices[i]->name == n)

found = true;

//2.:

if(!found)

vertex \* v = new vertex;

v->name = n;

vertices.push\_back(v);

)

**Add Edge**

addEdge(key0, key1, weight) – add a connection between 2 keys with a specified weight

Approach:

1. Locate key0 in graph – call it v0
2. Locate key1 in graph – call this v1
3. create a new edge (e0)
   1. set e0 to point to v1
   2. set eight of e0
   3. append e0 to v0 adjacency list
4. create new edge (e1)
   1. set e1 to point to v0
   2. set weight of e1
   3. append e1 to v1 adjacency list

**Breadth First Search (BFT**)

* Breadth first traversal does not depend on weight and works well for unweighted and undirected graphs
* Given a starting vertex, we visit all neighboring vertices, before moving onto the next depth level

Let’s update our vertex struct so that we can keep track of which vertices have been visited

struct vertex

{

string key;

bool visited = false’

vector<edge> adj;

};

struct edge

{

vertex \*v;

}

Now the BFT algorithm:

void breadthFirstTraverse(keyStart);

Find the starting vertex based on keyStart

set vStart as “visited”

create a queue (q)

Enqueue vStart onto q

Loop until q is empty

n = dequeue (remember queues are FIFO)

loop across n’s adjacency list

if !n->adj[x].visited

mark visited = true

display(n->adj[x].v->key)

enqueue onto q

**Breadth First Search**

The breadth first order of traversing a graph has a very useful property: we can figure out the shortest path between two vertices

Update the BFT algorithm so it takes in two values:

1. start key
2. end key

Now we are searching for the shortest path from one vertex to another

Some teaks to the BFT are required:

* Update the vertex struct to include a distance member (int; initialize to 0)
* In the traversal, update the distance when visiting each vertex from adjacency list
* add a check for if the end key is found (return)

struct vertex

{

string key;

bool visited;

int distance = 0;

vector <adjVertex> adj;

};

breadthFirstSearch(keyStart, searchKey)

Find key of starting vertex (can think of it as root)

set vStart as “visited”

set vStart distance = 0;

Create a queue (q);

enqueue vStart onto queue

Loop until q is empty

n = dequeue

loop across n’s adjacency list

check each element to see if visited != true

n->adj[x].v->distance = n->distance + 1;

if n->adj[x].v->key == searchKey

return n->adj[x].v;

else

mark visited = true;

engueue onto q;

**Depth First Traverse (DFT)**

Sometimes it is more useful to explore each “branch” of the graph until the end is reached, before retreating and searching the next “branch”.

Approach:

Keep visiting the first non-visited vertex in each adj list. Once adj list with 0 non-visited vertices is encountered, go back to last intersection and visit next vertex in adjancency list.

depthFirstTraverse(value)

root = find(value);

display(root.key);

depthFirstRecurse(root)

depthFirstRecurse(vertex)

vertex.visited = true;

loop across adj list of vertex

if(vertex.visited = false)

display(adj vertex.key)

depthFirstRecurse(adj vertex)

**Disconnected Sub-Graphs**

A graph does not necessarily have to have a path between any two vertices.

Neither BFT or DFT has the ability to “hop” between disconnected components

Both BFT and DFT guarantee that if depoloyed on any vertex in a given sub-graph, entire sub-graph will be traversed (every vertex will be visited)

Approach:

1. pick a starting vertex(does not matter which one – start with first in list)
2. initialize a counter (numSubGraphs) to 0
3. invoke a traversal (BFT or DFT) – mark each visited vertex
4. increment numSubGraphs
5. check if any unvisited vertices remain in entire graph
   1. T: back to step 3. call traversal algorithm first unvisited vertex
   2. if false; finished, no more vertices

**Graph two-colorable?**

A special kind of graph property we can check for is whether a graph is *bipartite* or not:

* Bipartite graph is one who’s bertices can be split into two groups, such that all vertices of one group only have edges with vertices of second group
* It is possible to color the graph’s vertices in two alternating colors, such that no two same-color vertices are ever adjacent
* a graph is either bipartite or not
* various applications in parallel computing, machine vision (object recognition)

*Problem****:***

Write a program that checks whether a given graph is bipartite or not

* the function takes no parameters
* Returns bool (bipartite or not)
* all vertex colors are initialized as empty string (neither red or blue)

Approach: modify the BFT

1. Initialize bipartite = true
2. Pick a starting vertex (first element In vector is fine)
3. set starting vertex as red and visited
4. EnQ starting vertex
5. Loop until the Q is empty



* 1. n = q.deQ
  2. loop over n’s adjacency list
     1. Check the visited adjacent vertices:
        1. if n->color == “red” && n->adj[x].v->color== “red”
        2. if n->color == “blue” && n->adj[x].v->color== “blue”
           1. return false
     2. if not visited
        1. Color neighbor of n as opposite color
        2. mark as visited
        3. enQ non-visited nodes

1. Return true

**Dijsktra’s Algorithm**

breadth first finds the shortest path in an *unweighted graph*.

if given a graph with edge weights, BFS will not work to give shortest path (will give fewest number of vertices.)

use Dijkstra’s instead

consider flight routing example: number of lay-overs vs total distance

need to update the vertex struct:

struct vertex

{

string key;

vector<edge> edges;

bool solved; //similar to visited

int distance; //total distance of weights

vertex \*parent;

};

struct edge

{

vertex \*v;

int weight;

};

Find the shortest path from A to E

1. Start at A and mark as solved. Add to solved list
2. Traverse the entire solved list
   1. Scan A’s adj list and find unsolved vertex that is nearest A
   2. Mark closest vertex as solved, add to solved list
3. Traverse the entire sovlved list (again)
   1. Scan A’s adj list and find unsolved vertex nearest A. (C )
   2. Scan B’s adj list and find unsolbed vertex nearest A (none)
   3. Mark the closest vertex (min(a.b)) as solved, add to solved list
4. Traverse entire solved list
   1. Scan A’s adj list – closest = E-5
   2. Scan B’s adj list = closest = nothing
   3. Scan C’s adj list = closest = D – 3
   4. min(a,b,c) = D-3 *note: oly D is marked as solved*
5. Traverse entire solved list
   1. Scan A’s adj list – closest = E-5
   2. Scan B’s adj list = closest = nothing
   3. Scan C’s adj list – closest – nothing
   4. Scan D’s adj list – closest – E-4
   5. min(a,b,c,d) = E – 4

**Dijkstra’s pt 2**

vertex(start,end)

//given the starting and ending vertices,

// find the shortest path and return the ending vertex

vStart = search(start)

vEnd = search(end)

vStart.solved = true

//create a linked list to store solved vertices

//can use a vector

solvedList.add(vStart);

while (!vEnd.solved)

{

//set min distance to some “huge” value

minDist = INT\_MAX

//need a temporary pointer

solvedV = null

//iterate across the list of solved vertices

for x = 0 to solvedList.size()

s = solvedList(x)

//now iterate across s’s adj list

for y = 0 to s.adjList.size()

//only visit unsolved vertices

if(!s.adjList[y].v.solved)

dist = s.distance + s.adj[y].weight

//now, check if this distance is less than smallest

//distance found so far

if(dist<minDist)

//set solved vertex

solvedV = s.adjList[y].v

minDist = dist;

//set pointer to keep track of the path

s.adjList[y].v.parent = s

solvedV.distance = minDist;

solved.solved = true

solvedList.add(solved) `

}

return vEnd;

//Solved = if it is the min of all of the vertices currently in the adj list? ish

## Hash Tables

a data structure that stores records ina linear structure, such as array. the index of the array is calculated for each element (record) by a hashing function.

example:

have four records to store: Anna, Jamie, Bryan, Luara

hashF(Anna)-> 2 //index of an array

hashF(Jamie)-> 0

hashF(Bryan)-> 1

hashF(Laura)-> 3

in summary, a hash table data structure can be broken down to two components:

1. hash function
   1. generate a *unique* code that corresponds to a valid array index
   2. is repeatable (hashF(Laura) => 3, I get a 3 everytime)
2. hash table

**hash functions**

*Mod-of-sum Hash Function*

* sum all the values in a string, then mod by array length

int hashSumMod(key, tableSize)

sum = 0;

for(int i = 0; i<key.size(); i--)

sum = sum + key[i]

return sum%tableSize

//in C++. string class can be indexed into

//extract individual characters

deploy hashSum on “Anna”. Assume table size = 5

A = 41

n = 110

n = 110

a = 91

sum = 358

sum%5 = 3;

*Store Anna record at index 3 of array*

**Multiplicative Hash Function:**

**int hashMult(key, tableSize)**

1. sum all the ascii values of key (kSum)

int hash = 0;

//calculate sum of key (like in mod of sum)

1. Multiply by a constant decimal value A where 0 < A < 1

A = 13/32;

hash = A\*hash;

1. Keep the fractional part

hash = fract(hash); //3.14 -> .14

1. Multiply by table size

hash = hash \* tableSize;

return hash;

Deply hashSum on “ Anna “. Assume tableSize = 1024

1. sum = 358
2. A\*hash = 145.4375
3. fract(hash) = .4375
4. tableSize\*hash = 448

Store Anna record at index 448

other hash functions exist: division

**Operations on a Hash Table:**

* store record
  + compute the idex value (hash(key))
  + write data to the table (array) at the index
* retrieve record (search)
  + use hash function to calculate the index
  + read data from the array at the index

what about operation costs?

* with a hash table, there is no need to traverse the other elements to find the location
* does not depend on the key size

**O(1)!**

what happens if a function produces a non-unique result?

* collision

**cryptographic hash functions**

* maps a “message” of any size to a hash value of a fixed size
* one-way function not invertible (cannot be cracked or reverse engineered)

**Collision Resolution**

1. Open addressing
   1. If location in table is occupied, find some other available location
2. Chaining
   1. Each element of the table is linked list head pointer

**Open Addressing**

if a collision occurs, add record at some other available location

*Linear Probing*: finds the next available location and stores the record there

index = hash(record.key)

while(table[index] is occupied)

index++

table[index] = record;

downside: clustering

* elements are likely to get bunched up in one area of the table
* performance goes from O(1) -> O(n)

*Quadratic Probing:* instead of looking at next adjacent slot, skip over by i^2 indices 0

i = hash(record.key)

index = i

a = 1

while(table[index] is occupied){

index = i + a^2

a++

}

table[index] = record

open addressing - cont

1. have a secondary hash function. this is called **double hashing**

h = hash(record.key)

h2 = hash2(record.key)

index = h

i = 1

while(table[index] is occupied){

index = (h + i\*h2)%tableSize

}

table[index] = record

**Universal Hashing**

* hash table uses a family of hash functions
* at the beginning of program execution, one of the functions is selected.
* this same hash function is used throughout the lifetime of the given hash table
* next time the program executes, a different hash function is randomly chosen

# heap

* an ordered tree: parent/child ordering
* unlike BST, where left key < right key
* 2 main types of heaps exist:
  + min-heap:
    - root of the tree stores the smallest value
  + max-heap
    - root of the tree stores the largest value
* complete tree: height between two bracnes is <= 1

implementation: we store the tree in an array (not individually dynamically allocated nodes like a BST)

the indexes get calculated as follows:

* root = 0
* for any element at index x
  + LeftChild(x) = 2x+1
  + RightChild(x) = 2x+2
  + parent(x) = floor((x-1)/2)

**min heap ADT**

public:

init – (constructor)

instert(key)

extractMin()

peek()

printHeap()

private:

int \*heap //ptr to array

int capacity //array size

int currentSize //counter

minHeapify(index) //recursive method to fix heap

int parent(index) {return (i-1)/2}

int leftChild(index)

int rightChild(index)

void swap (int &x, int &y)

# MIDTERM 2 REVIEW

**Binary Search Tree**

* parent-child relationship
* non-linear DS
* big-O of O(logn) when balanced
* 3 traversals:
  + pre-order (root,left,right)
  + in-order (left,root,right)
  + post-order(left, right, root)
* ADT:
  + print
  + search
  + insert
  + delete
    - node has 0 children
    - node has 1 child
    - node has 2 children
    - find min of right subtree
  + destructor
    - post-order
* Red Black Tree (conceptual
  + Rules of a RB Tree
    - A node is either red or black (color can change as a tree re-balances)
    - root node is black
    - every leaf node is black, empty and null
    - if a node is red, both its children must be black
    - for every node in the tree all paths to a descendant leaf node must pass through same number of black nodes
  + recoloring
  + rotation

**Graphs**

* Nodes and edges get added explicityly
* No implied parent/child relationships
* weighted v unweighted
* directred v undirected
* edges can be stored in either adjacency lists or adjacency matrices
* STL vectors basics
  + what is the underlying mechanism
  + declaration syntax
  + access, add, and remove elements
* ADT:
  + insert vertex – *simply allocate new vertex dynamically, then add pointer to vertex vector*
  + add edge – *undirected graph, add one in each direction to each vertex’s adj list*
  + traversals
    - Breadth First
      * traverse
        + use queue, run till Q empty
        + scan adjacency list of each item in Q, add every unvisited vertex to Q
        + mark every checked node as visited
      * search - add distance member
        + n->adj[x].v->distance = n->distance + 1
      * complexity – O(V+E)
    - Depth First
      * recursive algorithm:
        + given a vertex, mark as visited, and call revursively with first non visited item from adj list
        + base case = no unvisited vertices left in adj list
      * complexity – O(V+E)
    - Dijkstra’s Traverse
      * finds shortest path between two vertices based on weighted edges
      * maintains a list of “solved” vertices (not queue)
      * *vertex gets added gto solved list if it is the min of all the unsolved vertices in the adj lists of all the vertices in the solved list*
      * vertex does not get removed from solved list once added
      * algorithm terminates as end vertex gets marked as solved
      * complexity O(E logV) for adjacency list implementation (not on exam)

**Hash Table**

* hash functions
  + sum-mod
  + multiplicative-mod
* collision resolution
  + open addressing
    - linear probing
    - quadratic probing
    - double hashing
  + chaining
    - linked list
* complexity
  + perfect hash O(1)
  + lots of collisions O(N)

Uses:

* priority queue
  + unlike a standard queue, where the order is first in first out, PQ always returns the itehm with highest priority
* heapsort: c++ performance comparison

limited data structure bc you can only remove from the root