Data structures: final exam review essay sample

•Depth: length of the unique path from root to node

•Height: length of the longest path from the node to a leaf •Keep children in a linked list

•Preorder traversal: work at the node is done before its children are processed •Postorder traversal: work at a node is performed after its children are evaluated •Binary tree: no node can have more than two children

oAverage depth is O(rootN), O(logN) for binary search tree

oCan maintain references to children cuz there’s only 2

•Example of a binary tree: expression tree

oLeaves are operands, other nodes contain operators

oInorder traversal: recursively print left child, then parent, then right •O(N)

oPostorder traversal: recursively print left subtree, right subtree, then operator → O(N)

oPreorder traversal: print operator, then recursively print the left and right subtrees

oConstructing an expression tree from a postfix expression: read one symbol at a time; if operand, create a one-node tree and push it onto a stack. If operator, pop two trees T1, T2 from stack, and form a new tree whose root is the operator, and whose left and right children are T2 and T1; push new tree onto stack •Binary search tree: binary tree with the property that for every node X, the value of all items in its left subtree are < X and the value of all items in the right subtree are > X oContains: Uses O(logN) stack space

ofindMin, findMax: traverse all the way left or right from the root oinsert: traverse down tree as would with contains, stick it at the end oremove: easy

if leaf or has one child; if two children; replace data in node with smallest data of right subtree, and recursively delete that node oLazy deletion: if expected number of deletions is small, just mark the node as deleted but don’t actually do anything; small time penalty as depth doesn’t really increase oRunning time of all operations on a node is O(depth), and the average depth is O(logN) oIf input is presorted, inserts takes O(N^2) since there are no left children •AVL Trees

oBinary search tree with a balance condition: ensure depth is O(logN) by requiring that for every node in the tree, the height of the left and right subtrees can differ by at most 1 (height of empty tree is -1) oMinimum number of nodes S(h) of an AVL tree of height h is S(h) = S(h-1) + S(h-2) + 1 where S(1) = 2

oAll operations O(logN) except possibly insertion

oRebalancing:

•Violation after inserting into left subtree of left child, or right subtree of right child → single rotation •Violation after inserting into right subtree of left child or left subtree of right child → double rotation •Splay Trees

oAmortized O(logN) cost per operation

oMove accessed nodes to root

oZig-zag: node is a left child and its parent is a right child or vice versa oZig-zig: node and its parent are both left or right children •Level-order traversal: all nodes at depth d processed before any node at d+1; not done recursively, it uses a queue instead of stack recursion •Set interface: unique operations are insert, remove, and search oTreeset maintains order, basic operations

take O(logN) worst case •Map interface: collection of entries consisting of keys and their values oKeys are unique, but several keys can map to the same values oSortedMap: keys maintained in sorted order

oOperations include isEmpty, clear, size, containsKey, get, put oNo iterator, but:

• Set keySet()

•Collection values()

•Set entrySet()

oFor an object of type Map. Entry, available methods include •KeyType getKey()

•ValueType getValue()

•ValueType setValue(ValueType newValue)

•TreeSet and TreeMap implemented with a balanced binary search tree

Ch. 5 Hashing

•Hashing is a technique for inserting, deleting and searching in O(N) average, so findMin, findMax and printing the table in order aren’t supported •Hash function maps a key into some number from 0 to TableSize – 1 and places it in the appropriate cell •If the input keys are integers, then usually Key (mod TableSize) works •Want to have TableSize be prime

•Separate chaining: maintain a list of all elements that hash to the same value •Load factor = average length of a list = number of elements in table/size oIn an unsuccessful search, number of nodes to examine is O(load) on average; successful search requires ~ 1 + (load/2) links to be traversed •Instead of having linked lists, use h(x) = (hash(x) + f(i)) (mod Tablesize)

where f is the collision resolution strategy oGenerally, keep load below . 5 for these “ probing hash tables” oLinear probing: f(i) = i; try cells sequentially with wraparound •Primary clustering: even if table is relatively empty, blocks of occupied cells form which makes hashes near them bad oQuadratic probing; f(i) = i^2

•No guarantee of finding an empty cell if table is > ½ full (or before if size isn’t prime) •Secondary clustering: elements hashed to same position probe same alternative cells oDouble Hashing: f(i) = ihash_2(x) so probe hash_2(x), 2hash_2(x), … •Hash_2(x) = R – x (mod R) with R prime < size is good

oRehash: build new table, twice as big, hash everything with new function •O(N): N elements to rehash, table size about 2N, but actually not that bad because it’s infrequent (must have been N/2) insertions prior to last rehash, so it essentially adds a constant cost to insert •Can rehash when half full, after failed insertion, or at certain load •Standard Library has HashSet and HashMap (they use separate chaining) •HashTable useful for:

o1. Graph theory problem where nodes have names instead of numbers o2. Symbol table: keeping track of declared variables in source code o3. Programs that play games

o4. Online spell checkers

•But they require an estimate of the number of elements to be used

Ch. 7 Sorting

•Bubble sort: O(N^2) but O(N) if presorted

•Insertion sort: p passes, at each pass move element p left until in right

place oO(N^2) average, O(N) on presorted

•Shellsort: increment sequence h1, h2, …, h_t

oAfter a phase, all elements spaced h_k apart are sorted

oWorst-case O(N^2)

oHibbard’s sequence 1, 3, 7,…, 2^k – 1 gives worst-case O(N^3/2)

•Heapsort: build a minHeap in O(N), deleteMin N times so O(NlogN) for all cases oUses an extra array so O(N) space

•Mergesort: O(NlogN) worst case, but uses O(N) extra space/memory

•Quicksort: use median of left/right/center elements, sort elements smaller and larger than pivot, then merge oPartitioning strategy: move i right, skip over elements smaller than pivot, move j left, skip over elements larger than pivot oWorst-case pivot is smallest element = O(N^2), happens on near-sorted data oBest-case pivot is middle = O(NlogN)

oAverage-case O(NlogN)

Ch. 8 The Disjoint Set Class

•The equivalence problem is to check for any a, b if a~b

•Find: returns the name of the equivalence class containing a given element

•Add a relation a~b: perform find on a, b then union the classes •Impossible to do both operations in constant worst-case, but can do either •Quick-find: array entry of node is name of its class; makes union O(N) •We start with a forest of singleton trees; the array representation contains the name of the parent, with -1 for no parent oUnion: merge two trees by making the parent link of one tree’s root link to the root of the other tree. O(1) oFind is proportional to depth of the node so worst-case is O(N), or O(mn) for m

consecutive operations oAverage case depends on the model but is generally O(mn)

•Union-by-size: make the smaller tree a subtree of the larger, break ties any way oDepth of a node is never more than logN → find is O(logN) oHave the array entry of each root contain the negative of the size of its tree, so initially all -1. After a union, the new size is the sum of the old •Requires no extra space

oMost models show M operations is O(M) average time

•Union-by-height: maintain height instead of size of tree, and during unions make the shallow tree a subtree of the deeper one oAlso guarantees depth = O(logN)

oEasy: height only goes up (by 1) when equally deep trees are unioned

oStore the negative of the height, minus an additional 1 (again start at -1)

•Problem: worst case O(MlogN) occurs frequently; if there are many more finds than unions the running time is worse than the quick-find algorithm

•Path Compression

oAfter find(x), every node on the path from x to root has its parent changed to the root oM operations requires at most O(MlogN) time; unknown average oNot compatible with union by height since it changes heights •When we use both union-by-size and path compression, almost linear worst case oTheta(M*Ackerman’s function), where Ackerman’s is only slightly faster than constant, so it’s not quite linear oBook proves any M union/find operations is O(Mlog*N) where log*N = number of times needed to apply log to N until N