CS 260 Course Notes

Andy Gnias

January 27, 2019

Examples

Look at diagrams in course notes and textbook examples throughout the chapters which have solutions.

Definitions

Algorithm - finite sequence of instructions that can be performed with a finite amount of effort in a finite amount of time

Abstract Data Type - mathematical model with a collection of operations defined on that model, ex. list

Data Type - set of values that a variable may assume, ex. boolean

Data Structure - collection of variables connected in different ways. Can have different types

Cell - building block of a data structure

Array - sequence of cells

Field - collection of cells

Record - cell made from a field

File - sequence of values of some type

Pointer - Cell whose value indicates another cell

Cursor - integer value cell that points to an array

Greedy Algorithm - No regards for the optimal solution

List - sequence of zero or more elements of a given type

Rule of Sums - Runtime of a fixed sequence is the run time of the step with the largest run time

Set - Collection of unique elements, of which each element is either itself a set or an atom

Atom - Primitive element of a set

Disjoint sets - Sets that have no members in common

Dictionary - Set ADT with operations $INSERT$, $DELETE$, and $MEMBER$

Trees - collection of nodes

Nodes - elements of a tree

Binary Tree - Tree where no node has a vertex of more than 2

Balanced Tree - A tree with an average seek time of $\log n$

Big-O and Program Efficiency

Program Runtime

Program runtime is measured in terms of worst case scenarios, as it can be difficult to calculate an "average" input.

Reasons to disregard program efficiency:

- Time consuming to create
- Small inputs
- Difficult for others to understand
- RAM usage
- Accuracy and stability concerns

O-Notation

Upper-bound of the growth rate of $T(n)$.

Gives limited information, only tells us how a function grows.

Ω-Notation

Lower-bound of the growth rate of $T(n)$.

Abstract Data Types

Arrays

Arrays inherently track

- Capacity

- Size

Function Table

Function	Big-O Time	Notes
First(A) = p	O(1)	p=position
Next(p,A) = p	O(1)	
Retrieve(p,A) = x	O(1)	Arrays utilize RAM
Append(x,A)	O(1)	Add value + increment size
Find(x,A) = p	O(n)	O($n \log n$) if sorted
Delete(p,A)	O(1)	When order doesn't matter
DeleteSorted(p,A)	O(n)	Worst case, everything moves
Insert(x,A)	O(n)	
Size(A)	O(1)	
MakeNull(A)	O(1)	Just set size to 0
Resize(A,c)	O(n)	

Additional Notes about functions:

- *Delete* - When order doesn't matter, replace the item with the last element and decrement the size. There is no need to "erase" the memory, as it will always have some sort of data because its capacity remains the same.
- *Resize* - Cost is measured by number of copies needed. The cost of resizing does not increase with the size of the copy.
- Arrays utilize random access memory, which makes the better for implementation.

Lists

Lists inherently track

- Position

Function Table

Function	Big-O Time	Notes
First(L) = p	O(1)	p=position
End(L) = bool	O(1)	"Are we at the end?"
Retrieve(p,L) = x	O(1)	
Append(x,L)	O(1)	
Find(x,L) = p	O(n)	Need to walk the list
Delete(p,L)	O(1)	Likely need 2 pointers in implementation
Insert(x,L)	O(1)	
Size(L)	O(1)	

Additional Notes about functions:

- *Append* - If we traversed the list to append an item, this would be O(n). However, if we implement a tail pointer, it can be O(1).
- *Size* - O(n) if we walk the list, or we can store size in the implementation for O(1).
- Memory grows linearly
- Same Big-O time for doubly linked lists

"Dummy" First Node

- "Get" is really "Get.Next" in the implementation
- "Remove" is constant time because it is not necessary to start at the beginning of the list.

Records

- If structures don't exist, like in bash, use parallel arrays.
- Can make a linked list with indices

Dictionary

- Set of key, value pairs
- A vector is an instance of a map. V: N-¿ R

Stacks

Last in, first out behavior. Stacks inherently store:

- Capacity
- Size

Function Table

Function	Big-O Time	Notes
Init() = S	O(1)	
Push(S,x)	O(1∗)	May change if array is resized
Pop(S) = x	O(1∗)	May change if array is resized
IsNull(S) = bool	O(1)	
Peak(S) = v	O(1)	
MakeNull(S)	O(1)	

Queues

First in, first out behavior. Queues inherently store:

- Position

Examples of queues include:

- Bash: pipe
- C: mkfifo()

All operations in a queue are constant if elements are added and removed with head and tail pointers.

Function Table

Function	Big-O Time	Notes
Init() = Q	O(1)	
Enqueue(Q,x)	O(1)	Linear in an array implementation
Dequeue(Q) = x	O(1)	
Front(Q) = x	O(1)	
Empty(Q) = v	O(1)	

For more on the implementation of queues, see notes on doubly-ended queues.

Recursion

Needs:

- A base case
- Way to approach a base case

Can be replaced with a while loop and a stack.

Trees

- A single node by itself is a tree. This is also the root of the tree. A tree with multiple nodes also has subtrees.
- The length of path is one less than the number of nodes in the path, so there is a path of length zero from every node to itself.
- A node with no proper descendants is called a leaf.
- A node's height is the length of the longest path from the node to a leaf. The height of a tree is the height of the root.
- The depth of a node is the length of the unique path from the root to that node.

- Siblings are children of the same node.

Trees store

- Data
- Left Pointer
- Right Pointer
- Parent (optional)

In a tree implementation, any N tree can be stored as a binary tree; left child, rightmost sibling.

Functions

- Search (Balanced Tree) - O($\log n$)
- Size - Needs to search the entire tree for a count; O(n)
- Height - Similar to size function, but take max instead of size; O($\log n$)

Node Order

Children of a node are ordered from left to right, but can be in an unordered tree as well. Data from any 2 of the below traversals can give the order of a tree

Preorder

Root, Left, Right

Postorder

Left, Right, Root

Inorder

Left, Root, Right

Huffman Tree

Used to compress data. Created by generating a probability chart for each data point, then picking the 2 smallest trees and merging them. This process is used to generate Huffman Codes. No code forms a prefix to any other code, so a code can be decoded if given as 1 large string.

Sets

Linear order of a set satisfies the following properties:

- For any a and b in S, exactly one of $a < b, a = b$, or $b < a$ is true
- For all a, b, c in S, if $a < b$ and $b < c$, then $a < c$ (transitivity)

Other rules about sets:

- A is included in B if every member of A is a member of B

Set operations:

- Union - Combining atoms of A and B
- Intersection - Set of elements in A and B
- Difference - Set of elements in A not in B

Hash Tables

Open Hashing