CPSC 2120 Section 2 Notes

8/21 -

Textbook: Data Structures and Algorithm analysis in C++
Link To Book PDF:

http://iips.icci.edu.iq/images/exam/DataStructuresAndAlgorithmAnalysisInCpp_2014.pdf

Office Hours: Tuesday/Thursday 2:00-3:00

11 Lab Meetings

8/26 Notes

Stack - First in/Last out style. Elements are added from “bottom to top” and the removed “top to bottom” - Think of this like a stack of paper on a desk

Queue - First in/First Out style. Uses Enqueue/Dequeue (add/remove respectively)
IE works like a line at a cash register (first to get in line is the first to leave line)

Priority Queue - The highest priority item leaves first. There is no standard order for unprioritized Items. IE, all elements must contain useful data AND a priority level.

IE consider the following

	recent				old
Added	first	sec	third	fourth	fifth
Value	1	2	3	4	5
Priority	1	3	2	5	4

Stack- last pushed value = 5
Value to pop = 1

Queue - last pushed - 1
Value to pop = 5

Priority - last added = 1
Value to pop = 1

Tree - Each “branch” in the tree can link multiple nodes. Nodes do not point back up to a previous nodes. Called “Acyclic” or a “Divergent Data Structure”

Useful for making classifications that get more and more specific

Nodes have “children” - IE the nodes that are branched to from a “parent” node”

Each child node can only have one parent node

This is a subset of the “Graph” organization structure

Leaf Node - When a node does not point to any other node (IE has null pointer)

Root Node - The “ultimate parent” node. Ie nothing points TO this node

Binary Tree - Each parent node has 0-2 children (no more). Each parent node must have 2 pointers. This is a special case of the tree organization

Heap - use of a tree (IE to find the largest value in a data structure). Any parent node must have a larger value than its parent node

End leaf nodes dont necessarily correspond to the least values

The Least value will be ONE of the leaf nodes

Usefulness of heap - As long as the tree is maintained in a heap form

The root node will always have the greatest value

When using a priority queue - it is beneficial to store priority values as a heap.

If heap condition isnt preserved, use a bubblesort “swap” method to bring it back to heap standards

Bubble Sort - When two side by side elements are out of order, they will be swapped to rectify the order. This is iterated until the list is sorted.

Selection Sort - Scans through an unsorted list, grabs the max/min value and is placed in the corresponding location (will always find the global min/,max and place it first)

Insertion Sort - It takes an element, it is inserted in the place it fits best

IE this order of changes

3 7 2 5 1 4

2 3 7 5 1 4 < Here it took the two and put it in the beginning since it was lower than prev

2 3 5 7 1 4

1 2 3 4 5 7

Done

Insertion sort can be harder to implement if there isnt an “insert” method into the array/vector. The “bump forward” element is hard to implement in some containers, but easier in a linked list (since inserting a node is comparable to making a new node in terms of resources needed.

Bubblesort, InsertionSort, and SelectionSort, have a time complexity of O(n^2)

Mergesort - The array will be divided into a trivial base case (each has a list of one element). Then each “sublist” is re-added together in a sorted order until the full list is created.

IE compare a first element against the first element of every other sub-list., add it to a now larger sub-list until only one list exists

Time complexity log2(n)

Quicksort - Randomly selects a particular value in the middle of the list, then picks one on either side of the randomly selected value. If they should be swapped, it swaps the values.

Heapsort - Uses a heap to find the largest value and then moves this to the appropriate place

Similar to selection sort, with the benefit of using a heap. Keep in mind every element added the the heap must be checked “upstream” to see if the heap structure is maintained

---

TASKS

Linear Search - time complexity of O(n) - checks a structure for a specific value by sequentially going through all data in order.

Binary search - Select middle of the array. Is the desired value on the left or right. Pick appropriate side and use the shrunken list (now half size) for the next iteration

Time complexity O(log2(N))

THE ARRAY MUST BE SORTED TO USE BINARY SEARCH

-

8/28-2019

How to compart

Asymptotic

Best/worst/average case

Deterministic vs Stochastic

Deterministic - apply no randomization. Follow a prescribed course of actions

(Divide and conquer technique like MergeSort)

Stochastic - have a component of randomness in decisionmaking

(Elements of random selection such as Quicksort)

Basic vs Amortized

Basic - single operation (suggests absolute worth case sometimes, even if rare)

Amortized - average over many uses of the alg

O, o, (omega) Ω, (lowercase omega), (theta)

General order of growth an algorithms time complexity will scale at

Uses “Big Oh”, “Little Oh”, Big omega, little omega, theta

How to compare (time complexity)

$O(n^2) \leq k*n^2$

^ upper bound of the order of complexity

$o(n) < k*n$

^ drops the equal to case, is the exceeding complexity (hence greater always) always higher than the alg comparing

Beyond the bounds of the algorithm

$\Omega(n^3) \geq k*n^3$

^ Concerned with lower bound order of complexity, IE best case of the alg has a complexity of n^3

$LittleOmega(n^{1.5}) > k*n^{1.5}$

^ best case complexity, but lower than the algorithm (hence less than) strictly better than the best case

Beyond the bounds of the algorithm

$Theta(\log(n)) = k*\log(n)$

^ exactly equal to

Algorithms vs design principles

Algorithms

- Exhaustive search (search whole data structure until find)

- Divide and conquer algorithms (such as merge sort, breaks down to trivially simple base case)
- Iterative refinement (start with an estimation and get closer and closer to the answer
- Greedy algorithms (a heuristic that tries to quickly find an approximation to the optimum by strictly choosing the best option of the options available in each iteration
- Dynamic programming ("Recursion with memory", in stead of invoking a recursive function call, this will maintain a data structure of the parameters used from a recursive function call and when the function is called with the same parms, it gives the pre-saved result to save time
- Random algorithms (incorporate randomness in control structures or in operations)
- Heuristics (used with algorithms in exchange for settling with an approximate, IE many cases where the optimal result is impractical to find, but an approximation works)
- Graph algorithms (represents a network with many nodes, nodes are connected to form a network. Algorithms specifically handle these data structures

Templates

- Exhaustive searches
- Depth First (for trees)
  - Begin at root node
  - Go from current node to child node on leftmost
    - If leaf node, back up and pick next child

- Breadth first (for trees)
  - Begin at root
  - Check all nodes on same level
  - Step to next level and check all nodes
  - Repeat until all checked and values are found

Templates

- Greedy Algorithms (for trees)
  - No-name
    - Goes the path in a tree with the larger value until it finds the local max (not true max)
- Dynamic Programming
  - Call tree memorized
    - IE save things like what f(1) (fibbonacci) is in order to find it easier, so calculations neednt be done anymore

For a memorized call tree - can you use “waypoint” values that a program can use in order to make the next calculations easier as well?

Dynamic Programming vs Divide and conquer
- Saves against function calls down a trace

Template
- Random Algorithms
  - IE a linked list where items point to the next node
    - Randomly placed and randomly spaced nodes to tell you if youre
      - searching to far or not far enough
- Heuristics
  - Annealing - start far off and gets closer
  - DNA - start with random “answer”, mutate until it converges towards and answer
- Graph Algorithm
  - Graph - network of interconnected nodes
    - Choose a starting node, and then find the shortest path to an ending node
      - IE Google’s pagerank algorithm

C++ vs C
- A struct and a class are almost the same in C
  - Neither need a typedef
    - (having class name in class functions is non standard)
      - Thats for constructors
  - Struct defaults to public access
  - Class defaults to private access
- Void in parameter list is deprecated (no need to use void)
- C++ has a bool data type - use it
- C++ has new and delete
  - Almmost never uses void-pointers/malloc/calloc/free
- C++ has nullptr (so NULL is deprecated)

CPSC 2120 - 8/30/19 notes

Recursion

A function is recursive when it is called in terms of itself

```
int fact(int n)
{
    if (n < = 1) // base case
        return 1;
    else
        return n*fact(n-1);
     // this line uses recursion. Suppose n = 4
     // the next call uses 4*fact(3),
     // then 3*fact(2)
     // then 2*fact(1) - here it is done since
                                fact(1) is the base case
}
```

IE this is a recursive function, it is the factorial problem shown in class (shown in code)

Base Case: This is the case when recursion occurs. its like asking you to count down from zero to zero (in that it is trivially simple and requires no induction to get to final step)

Recursion goes from the given case down to base case

A recursive function cannot continue indefinitely - if it goes too long, you'll eat up all the RAM (IE a memory leak)

Programming Styles

Header Files -

Each class gets a header file (class declaration) (.h file)

Function definitions go in implementation file (.cpp files)

Do not include these implementation files in the headers

C++ standard headers use #include<>

C++ user made header uses #include””

#pragma once - this includes each library ONLY once

THIS IS NOT STANDARD - but most compilers use this too

Alternatively use this format -

```
#ifndef
#define NAME
// libraries go here
#endif
```

This is an if structure that will only run the inside (between ifndef and endif) if the NAME hasnt already been defined

Pass By Alias

- Also called pass by reference
    - This is passing a variable to a function by pointer or reference variable
        - A ref parameter has & with the ref var in the parameter list

Const’s

- These are variables that cannot or should not be changed
    - This can help with information hiding
- This can also be for member fxns that dont change the object

- A function does not change its parameter(s)
    - IE when you pass by reference but dont want chage
        - Do this for speed of copy/lookup

Initializer lists

- Instead of having initialization be in the body of a funciton
        - Use an initializer list to make it clear and easy (possibly empty function)

Classes and object oriented

Types of constructors

Defualt constructor

Called with no parms, everyhing is defualt

Call w/

MyString A; //default constructor

Mystring &ptr = new MyString(); //default constructor

Copy constructor

Takes data from one object, and copies it to a new object

Pointers the same = shallow copy

New pointers/same data = deep copy

Using a reference variable allows the memory location to be passed and copied more efficiently

Any and all changes within the fxns will be reflected in the instance being called upon

Constructor -

A general fxn that will take parameters used to populate the class Attributes

Destructor

A fxn that takes an object and de-allocates any memory in use by the instance of the class

Rule Of Three

The compiler makes the copy operator, copy constructor, and defualt constructor

automatically but does not handle any dynamically allocated memory

C++11 added the move constructor and move assignment operator

Operator overloading

Comparison - use

```
Bool operator==(lhs, rhs)
{
        // cmpr statements
}
```

Stream insertion/extraction

The first parameter is always going to be istream or ostream or Fstream

Keep in mind you will need to be able to “chain” the insertion operator, so make sure that the istream/ostream operator is repeated in order to be used multiple times

In general , all operators can be overloaded to work with a class

If you need parm lists or documentation for an operator to overload use:
http://www.cplusplus.com/reference

Public/Private Operators

Class defualts to private
Structs defualt to public

Recall - (call means a fxn call or data call)

- Public - anything can call this as long as it is instantiated
- Private - must be accessed through public member/called through the class
- Friend - any friend class can call these

IN GENERAL

- Fxns are public
- Data is private
- (there are many exceptions to this though)

Class functions
- Big 5
    - Default constructor
    - Copy constructor
        - Make a clone of an object

        - The compiler generated copy constructor is a shallow copy
            - Pointer members retain the same values
        - Use deep copy when allocated data exists in the list
    - Copy operator (=)
        - If you implement copy constructor, make the copy operator
    - Move constructor
    - Move operator

Shallow Copy

```
Ptr1 --------------> 0x567A45----------> (int) 50
                         ^
ptr2----------------------- |
```

Deep copy

```
Ptr1 ---------------->0x567A45---------->(int) 50
Ptr2 ---------------->0x567a46----------->(int) 50
```

You can see the shallow copy ptr1 and ptr2 share the same memory address. It copys the superficial data in each variable (be it an int, a char, a string, or a pointer to a mem loc)

The deep copy example has 2 seperate memory locations containing the same information. This means that a NEW memory location is initialized with the same data. Now when ptr1 gets de-allocated, ptr2 will retain the information in the different mem location

Notes on move assignment and move operator

C++20 can use rvalue for reference - an rvalue is the value that will ONLY show up on the rhs

of an assignment operator

Lvalue = a value that can be on the left of an assignment operator

Notes on stream insertion operator

To overload the operator

```
Ostream & operator<< ( ostream & out, const Pear &ob)
^                                  ^             ^
Returns ostream to chain           lhs member    value to print
```

Memory Management

Any variable created by a declaration is auto released

Any allocated data must be manually deleted using the new/delete operators

IE

If you call

Pear P;

If no member in P is dynamically allocated, then at the End-Of-Life for the variable, it will be automatically de-allocated because the destructor does this

Pear* P - the memory for the pointer is de-allocated, but the object being pointed to will remain in memory and not be un-allocated

Smart pointers - keep track of how many pointers to the object exist, and delete the mem when all pointers are removed from pointing to the object

Memory leak - dynamically allocated memory having the pointing variable changed prior to de-allocation

How to use delete operator

Delete ptr; // its that easy

Set pters to nullptr to prevent use after de-allocation
(otherwise you do meaningless operations on meaningless data)

Destructor functions

```
~Pear()
{
    delete ptr;
}
```

Inheritance
Advantages
Code reuse
Abstraction

Allows one class to be a specialization of another class
IE a Person class could retain an is-a relationship with the Student class

Syntax for derived class

```
Class <name>: public <containing class>
{
    //inherited member variables from base additional member variabels

    //new constructors
    //
}
```

Access control
A derived class has direct access to the public members of the base class
A derived class can only access the private members through the base class

One can use a protected in the base class to give access to a base class but not to a general user

Base class pointer can be used to work with dervied objects
Cant use derived class pointer to work on base class

The BASE CLASS CONSTRUCTOR OPERATES FIRST
Tghen the derived class constructor

Destructors go from derived class out to base class

Lab Template Classes

Node, ListStack, ListQueue are all TEMPLATE classes

This means they are of ambiguous data type

This is the concept that lets

vector<int> and vector<char> both work

To denote a templated object

Prefix the fxns with

template<class <#Name>>

^^^^^^^^^^- #Name is user defined

Virtual Functions, pure functions, and Abstract Base Classes

Recall how variable casting works

Base *R = new Derived(); //okay

To use the member fxns in the derived class

“Remind” compiler that R points to a derived object

IE:

Derived *Q = static_cast<Derived*> (R) ;

What this does is takes the Base* object (R) and then casts it to a Derived Object. This is then pointed to by a Derived object *Q, and The mem location will then tell compiler to read the information in the mem location as type Derived and not type Base.

The derived class can provide its own version of base class function

Recall fxns are determined by name and parms, not return type

-Which function is executed? Base or derived?

Pay attention to the calling object, a base object will call base functions and vice versa

-How do I dictate which one to use

If a fxn in the base class is labeled “virtual”, the derived class fxn will always be used

In java- all fxns are implicitly virtual

If a function is virtual in a base class, and a derived class object doesnt define the fxn, the derived class will also be virtual

IE suppose we have derived1 and derived2. The fxn is defined in derived 2, which is derived from derived1, which is derived from base

Shown

Base(virtual)<-Derived1(virtual)<-Derived2(non-virtual)

Destructors
- Destructor should always be virtual
  - The derived classes will be different, so each needs its own destructor
    - Again only if the derived class has dynamically allocated data

Pure Virtual Functions
- A virtual fxn may have no implementation in the base class
  - This is a pure virtual fxn (IE it doesnt exist in the base class. It purely exists for virtual use)

IE
```
Virtual void base::scream() = 0;
```
^^^- the =0 postfix shows that there will be no actual definition of the fxn

//IN BASE NOBODY CAN HEAR YOU SCREAM

When you implement a pure virtual function you MUST implement in the derived class
- Otherwise you are making a call to a fxn that does not actually exist

Abstract Base class
- When a class has at least one pure function, it becomes an abstract base class
  - Cannot create objects of these classes because they don't have all functions defined

  - Instantiate these by creating the derived objects

- Java: the interface is a class without member variables whose member fxns are pure virtual

Linked Lists

A linked list is a collection of nodes

There is a pointer called *head/front/start/root/etc* that points to the first node
- Each node points to the next
  - The last points to nullptr

These use a strictly linear fashion (IE no node goes out of sequence or points to multiple nodes)

Traditionally nodes are structs
- You can use a class and
  - Specify access
  - Provide accessor/mutator methods
  - Make linked list class a friend of the node class

Insert/Delete
- The run time for a linked list is linear to trace through the list

- A pointer to move down a linked list is called an iterator and gets used to delete a Node
- The key to insertion and deletion methods are to move the iterator to the node before the change occurs

Doubly linked list
- Also have back pointers

- insert/delete has many cases
  - 4 specifically
    - 1 - update ptr to 1st node (place in first positon)
    - 2 - update ptr to last node (place in last position)
    - 3 - update ptr to adjacent nodes (place in intermediate position)
    - 4 - set head and tail to null (the only node in the list)

  - It is common to have a head and tail dummy node
    - These hold no appreciable data and just eliminate case 1 and 2 with Insert or delete operations

Circularly linked list

Get rid of tail node
Last node points to the first node again

Can store 2 d shapes as well
Ie each vertex is a node

Stack
Last in first oout data structure
LIFO

Insertion - push onto the stack (put on top)
Removal - pop from stack (remove top value)

Queue
First in First Out -> removal and insertion order are the same
FIFO

Insert - enqueue -> insert value to back of queue (put last)
Remove - dequeue-> remove value on front of queue (pop front)

Implementation
Deleting a stack or a queue
Deletion takes time order O(1)
Accessing takes O(n)

Implementation of queue using circular array
Array wraps around to beginning

(index++)%size -> this walks through a circular array with an ever increasing value for index

9/11/2019

Infix vs. Postfix expression

Math is typically in infix notation

EX

A+B

Alternatively, can write using postfix expression (AKA reverse polish notation)

EX

AB+

Advantages of Postfix Expression

Very easy to calculate (no operator precedence required)

Basically, postfix notation makes the calculation direct instead of parsing and ordering the operations

Postfix is left->right associative

In postfix, a+b*c becomes

abc*+

And AB+C

AB+C*

HOW TO USE POSTFIX - evaluate a postfix expr using a stack

Push operands onto stack

Then using the operands (in the postfix order)

Using the top items on stack

Perform operation on the top stack data

IE

ABC *+ gets pushed onto stack to become

C, B, A

AND the operator stack becomes

* +

Thus the first op will be using

C, B doing * = B*C

Then the final op will be
(B*C), A doing + = A+(B*C)

Converting Infix To postfix

Why use a stack for operands

The operands are in the SAME order - ie push the operands onto the stack in order they appear

Then - operators

First, the top of the stack is checked for any operator of higher precedence or of equal precedence. IE if this is true, pop the top of the stack and output to postfix expression. When no higher or equal precedence operators are found, then place the held operator onto the stack. Finally, put the popped operators back onto the stack(dont know abot this last one)

Precedence when reading from the stack

(-> highest
): special case - pop from stack until you find “(“
*and / and %
+, - -> lowest precedence

IE

Open paren
Closed paren - find open
Mul.div,mod
Add,sub

Precedence when popping from the stack
Mul, div, mod
add , sub
(- lowest precedence to keep) on stack until reaching)

Big O notation

In a pure sense, the run time is numebr of operations

The runtime is proportional to some function in terms of n
Its called the “order” of the code

It will follow the format

O(n^2)

For example, and it would mean that “at most, the code will run n^2 operations,
Where n is the ______________

There is an understood constant, IE
N^2 + k is O(n^2) where k is any constant number
You are only concerned with non constant factors in the code
A const means it runs EVERY time the prog runs

Adding with BigO notation
The larger power term prevails, as it will raise more rapidly

IE
O(n) + O(n^2) = O(n^2)

Multiplying BigO
Multiply like you would with fxns

O(n) x O(n^2) = O(n^3)

Specail cases
Linear - O(n)
Like accessing a member in a linked list

Constant - run time is not dependant on input size
Written O(1)

If two code segments are run in sequence, then the order is the “sum”

If one code segment is nested in another loop
The order of the out of loop progam * the order of the in loop program

GENERAL RULE

Every loop will increase runtime by order n

(IE every level of loop is +1 to the order of the program

IE count the number of loops and that will be the min order

Asymptotic analysis

IE what order are you approaching in the program

Why bother?

Resource analysis for the program

With modern hardware, small values are held in negligibly short time

Order Notations

Asymptotic Analysis

Type	Pronounced	Meaning (order of)	Examples
O(n)	Big-O	≤ c*n	5n, log(n), 1
o(n)	Little-o	< c*n	log(n), 1
Ω(n)	Big-Omega	≥ c*n	5n, $100n^2$, n!
ω(n)	Little-Omega	> c*n	$100n^2$, n!
Θ(n)	Theta	= c*n	5n, 100n, 0.01n

If a pr

More asymptotic analysis

$O(n^2)$	$\leq k*n^2$
$o(n)$	$< k*n$
$\Omega(n^3)$	$\geq k*n^3$
$\omega(n^{1.5})$	$> k*n^{1.5}$
$\Theta(\log(n))$	$= k*\log(n)$

These are examples of the same piece of code with each order being represented

EX

n^2 + 2n - 1

Well the order is n^2, thus theta is order n^2

Big O, must be of the order of the fxn or higher

O(n^2), O(n^3)...

Lil o, must be explicitly bigget

o(n^3) …

Big Omega is the same order or LOWER

Lil omega is an EXPLICITLY LOWER ORDER

How do you get Log in big O

If you divide by a factor each time, its gonna be log()

<u>Complexity</u>	<u>Name</u>	<u>Example</u>
$\Theta(1)$	Constant	Insert element at end of array
$\Theta(\log(n))$	Logarithmic	Binary search (sorted)
$\Theta(n)$	Linear	Naive search for maximum value (unsorted)
$\Theta(n \log(n))$	n log n	Mergesort
$\Theta(n^2)$	Quadratic	Two nested loops
$\Theta(n^k)$ - really, $O(n^k)$	Polynomial	Finding GCD with Euclid's algorithm
$\Theta(2^n)$ - really, $O(2^n)$	Exponential	Check for subset
$\Theta(n!)$ - really, $O(n!)$	Factorial	Generate all permutations

CPSC 2120 - 9/16/2019

Algorithm Complexity and order analysis

Examples

Shortest time - most desireable

theta(1) - inserting element at the end of an array(const time order)
theta(log(n)) - binary search on a sorted collection (note binary search requires a sorted collection, size halves each iteration)

^best cases

theta(n) - naive search for max val (IE unsorted) (linear)
theta(nlog(n)) - mergesort

^somewhat linear - controllable growth

theta(n^2) - two nested loops that cannot be **unrolled***
theta(n^k) - really O(n^k) - exponential - find GCD with euclids alg

^these are problematic with scaling - complexity grows and speeding up processors wont help
(below) DANGER ZONE - time orders that are slower than here are NOT ADVISABLE

theta(2^n) really O(2^n) - check for subset

IE Make, from set of all real integer numbers, all subsets of real ints
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
…
0 1 2 3 4 5 6 7 8 9 10 … ∞

thetan(n!) - really O(n!) - generate all permutations

Longest time - LEAST DESIREABLE

Linear Search example

Unrolling - this is the act of taking a fixed iteration loop (ie a for loop) and the statements within are placed sequentially. This works as long as the loop is explicitly stated how many times it will run, and thus DIRECTLY translated to DISCRETE statements. CanNOT be done on something like the following
while(!stack.empty())

Unrolling will work on this

```
This code                         unrolls into          this
for(int i = 0; i <= 3; i++) ------------------------> arr[1] = 0;
{                                                     arr[2] = 0;
        Arr[i] = 0;                                   arr[3] = 0
}
This code                         unrolls into          this
```

This means that the case in which an outer loop runs a variable number of times, and the inner loop runs a fixed number of times, the order of this structure will be O(n), since the inner loop gets unrolled and there becomes a linear relationship between iterations and time complexity.

Linear search example

Case	Exact	Order	How
Best	1	O(1)	Picked input
Worst	n	O(n)	Picked input
Average	(n+1)/2	O(n)	Uniform distribution on input

Amortized analysis -

- Amortized - to pay a cost gradually by little bits at a time

- Aggregate method
  - Find an upper bound $T(n)$ that holds for every sequence of n operations
    - IE this will be the LARGEST number of instructions to execute
  - The Cost of a single operation is $T(n)/n$
- Any n operations take $O(n)$ time
  - $O(1)$ amortized
- Any n operations take n^3 time
  - $O(n^2)$ amortized

- $T(n)$ = cost of n operations
  - $n*(1+.5+.25+.125+.0675\ldots)$
  - $< 2n$ (asymptotic case)
  - $O(1)$ amortized)

Summary

- Asymptotic analysis is HARDWARE INDEPENDANT
- Amortized analysis lets you find avg cost over n ops
- Only leading order term matters
- Good depends on the area, but logn is good “max”

9/18/2019
CPSC 2120
Quiz Friday - 9/20

Examples of big O notation/order analysis
With math and code

Example 1).

O(n) : up to order n: contains n, log(n) and any smaller growth functions

THETA(N) <-> order n: 2n, .5n, n+100, etc (highest order of n is 1)

Big OMEGA(n) -> at least order n: n, n^2, 2^n

Example 2).
Quadratic expression - f(x) = ax^2+bx+c (general quadratic form)
O(x^2) if there are no constraints on a, b, or c
Special Cases: a = 0
The function would now be order O(x)

Special case a != 0
The function is order O(x^2)

Special case: f(x) * g(x) & a!=0
The function has Big OMEGA(x^2)
This is because the MINIMUM order it can be is
Order 2.

Code Example 1).

Start
Cout << “x”; has order O(1) -> it does not have variable run time

Consider the cout statement is put in a for loop, and that loop runs n times
The order is now variable and dependent on N, therefore

```
for(n iterartions)
    Cout
```

Will have a a time complexity of

O(n)

The actual run time as well will be

THETA(n) -> this is because n is directly related to a function call,

IE for n iterations of the for loop, you can expect n function calls

Are log base(2) (X) and log base(3) (X), these will not be the same order

A higher base logarithm will grow SLOWER than a lower base log

Also,

Log(Log(n)) is a lower order than Log(n)

9/20/2019

Space complexity of a sorting algorithm does not include the space for the array
- The space complexity refers to the memory used by the algorithm such as either a
  - recursive function call or a making a new array

Stable sorting algorithm - if a data collection contains identical values, the order of identical values is not modified.

Any sorting algorithm that can be done in place has a space complexity O(1)
- Some will hold one element

- Sort in place = do not make a copy, modify the original array

Value you are sorting with respect to - key value
- IE the key is there to maintain a stable sort

Insertion sort
Bubble Sort
Selection sort

Faster sorting algorithms
- Based on insertion sort
  - Rather than going back into the array,
    - Shellsort uses a gap sequence in which multiple insertion sort
      - processesses with decreasing gap size
      - Gap size = how far to decrement index

      - Each greater iteration of shellsort will create a number of
      - Sub arrays that are the equal to gap size

      - Since shell sort works faster than insertion sort
        - Somewhere between factor of 2 and 4 in comparison

        - BEST CASE O(nlog(n))
        - Average - open problem (depend on gap sequence)

Worst - open problem (depend on gap sequence)

Monday starts with mergesort

9/25

Heap sort
- Take the root node, its the largest value
- Do maintenance to move everything and preserve heap structure

General formula for a heap
- To go from parent to child
  - Multiply the index for parent node by 2, add 1 for left child
  - Add 2 for right child

Shell sort - still finding average case as it is heavily dependent on gap size

Quicksort has average runtime of nlogn
- Assuming you pick a pivot value that shows up an average amount of times

- IE youll pick a low pivot as much as a high pivot
  - Becuase of this, you can assume half falls less than pivot
  - Half falls above pivot

Unordered data structures
- Set - only concerns whether an element appears in set
- Map- data structure accesses items in constant time using a key to lookup value
- Priority Queue - FIFO but higher priority leaves first
- Tree - has root/leaves
- Heap - tree where children are less value than parents
- Hash table - takes value as input, does alg, uses as key to find output

What structure should I use?

Data needs to be displayed in order	Sorted Array/List, Binary Search Tree
Fastest way to find data (asymptotically)	Hash Table
Slowest way to find data (array)	Linear Search
Mixes up the data order (unordered)	Hash Table
Can quickly find the largest/smallest element (one of these)	Heap
Can quickly find the largest/smallest element (both)	Tree
Find value (sorted): Runs in log(n) time	Binary Search
Find value (unsorted): Runs in O(n) time	Linear Search
Find value (unordered): Runs in O(1) time (expected/amortized)	Hash Table
Dealing with a complex situation where normal searches don't work (unorganized and non-linear structure (i.e. tree/graph))	Exhaustive Search Algorithm

9/27/2019

- Exam 1 - Covers up through sorting
  - Questions about implementation of algorithms
  - So yes there will be code questions - know how to implement each alg

  - Open ended coding wont be on the test
  - Provided code - match with use will be present
  - Fill in blanks in code
  - Order lines of code

  - Emphasis on
    - Mergesort
    - Quicksort
    - Recursion
    - Implement Linked list based stack and queue

REVIEW SESSION

- Next week
  - Exam review will probably be wednesday

Iterative refinement

- Alg gives result each iter that is closer to the final result

Incremental Construction

- Specific final result built one item at a time

Iteration vs Recursion

- Recursion outlook - deal with first el, then recursively solve rest of problem
  - IE Insertion sort

  - Selection Sort - also recursed

- This approach often maps naturally to linked lists, giving simple implementations

Any sorting algorithm is STABLE if you sacrifice working IN PLACE

- Ie make a copy, and now youll be a stable sort

Memory Issues with Sort algs

- Rather than sort large record
- Sort pointers to records
- Some advanced algs only move elements O(n) times

- How will caching affect performance of sorting algs?
  - If all accesses in sequence, cache mem will speed up alg time

Ideal Sorting Algorithm
- Stable AND in Place
- Only needs O(n) moves (memory writes)
- Deterministic (no random steps)
- Runs in O(nlogn) time - There is a bigomega(nlogn) worst case lower bound on Comparison based sorting algs
- Would run in closer to O(n) time if data is mostly sorted (adaptive)
- SIMPLE to implement and analyze

Non comparison sorting algorithms exist
- Gets time complexity down to Theta(n)

- Bucket Sort
  - Array of empty “buckets” that hold values
  - Read through list, place in buckets
  - Sort non-empty buckets
  - Visit bucket and re-assemble list

9/30

- Exam Friday
- Review Session Wednesday Night

Hash Tables

- Stuff like that
- Use hash tables for spell-check

Bucket Sort

- Make buckets that refer to intervals
    - Place items in their bucket
        - Sort the non-empty buckets
            - (recurse to making more sub-buckets for each bucket)
        - Take items out of the buckets in order
- Order(N) if the distribution is even AND #buckets scales with length

Comparison Vs Non Comparison sorting

- Comparison only works when data is comparable
- Non comparison sorting is faster

Counting Sort

- NOT STABLE SORT
- NOT IN PLACE
- Order; O(N + K)
    - N = # elements
    - K = initializing the count array
- A bucket can hold only ONE exact element (that el is called a key value)
- Can be done in Linear time

- Instead of making buckets, use a list of possible values
    - Every time a val appears, incr counter for it
    - Thus, you read-back the count array and rebuild the return for that

Radix sort
- Stable sort is CRITICAL to this working (when going from LSD to MSD)
  - Otherwise it is unstable when going from MSD->LSD
- Order
  - O(N*K)
    - N = number of elements
    - K = #digits per element (using largest element)
- Break into parts with a particular sequence
  - Key is a multi difit number with positional digit values
    - # of keys = base(10 in base 10, 0 - 9)

- IE you sort based on digit significance
  - For the first pass, compare ones place
  - Second pass, 10’s place
  - …
  - Finally, sort based on Most Significant Digit

  - Each pass, digits are only 0-9
    - Thus use counting sort to do the sorting mid-pass

Sets/Dictionsaries
- Abstract Data Type
  - Set/dictionary

- MUST support operations
  - Insert key into structure
  - Remove Key from structure
  - Find whether a key is in the structure
  - Enumerate - all keys in structure in any order

Choices for implementation

- Sorted or unsorted
- Linked list / Array

Sets & dictionaries

Each item has a key,
and keys are unique

Sets: Key has no associated value, simply a collection of UNIQUE vals

Dictionary Key maps to an associated value
Note key not always same as
Associated value
IE
1 = red
2 = blue
3 = green
4 = … <other colors and so on>

ALSO:
In dictionary, the KEY accesses the ITEM
Not backwards

Look at the ppt for the runtime of each operation on the maps/dicts/sets

Set Abstraction
Given N strings
Find all duplicates
Using set abstraction makes soln and runtime analysis much easier

Direct access table
A lare array of bools, called A
Presence of key k in structure implies that A[k] = true;

Hash Tables
Sort els into arry
Key k stored in h(k) (hash table at K)
h() is known as hash function

Exam on Friday
Know the best/worst case of the algs we discussed

Hash table
Uses Dictionary/set data structure
Purpose is to FIND values quickly

Using a parallel array of bools, to show wether a key is in a set, can make the time order for a lookup to be O(1) (array index) -> ARRAY IS IMPORTANT (it goes to shit with a LL)

How do you get around a direct access table (mentioned immediately above)
How to use a smaller array for the same stuff

MAKE a hash function
A hash function will take a parameter, do some shit to it, generate a NUMERIC value (Key, call it k) within a range based on input. This then gets put into h(k)

Basically, the hash function works like a funnel. It takes

Ideal Hash Functions
Give uniform distribution of values
IE each val gets returned an equivalent number of times

Avoid Patterns as well (then people can predict output of your hash table)
IE avoid common factors between mod and the values in your has functions

Collision - when a newly inserted key maps to an existing “slot” in the hash Table

Hash Tables: Positives
Any Arbitrary value will be a valid key
Any data is a sequence of ints to feed to hashing functions

Space for a hash table is small
Ideally uses O(N) space for storing N elements
Size can be variable, to expand when data gets too big
Collisions are INEVITABLE
There is a (#full/#size)% chance of collision Happening

Sometimes called the “pigeonhole principle”

Handling Collisions

When you have N elements in the hash table, but n+k elements mapped

You have some outputs get mapped from multiple inputs

Probing - if collision found, then keep scanning forward until we reach an empty slot

Find(K): start at pos h(K) and scan forward until you reach key k or an empty slot

Performance will be decent, O(1) as long as table is sufficiently larger than N and gives

Note you get O(1) because ideally, hash tables have an empty space

EVERY other space (thus size = 2*#elements)

uniform (unpatterned) output

Stopping conditions for probing

When a match is found, check if it is empty. Then you have an unsuccessful find. Otherwise, keep going until you find the value

How to handle NO empty slots?

Make the key table DOUBLE length

REHASH EVERYTHING WOOO

Linear probing works kinda like a circular array

Past last index = fisrt index

Removing Elements with Probing

Open addressinhg

You move from an element if the first one is full

You have the same stop conditions with probing

If you reach an empty cell first, nothing to remove with that key

(its not in table

If you have a match, remove the element

NOW, with probing, if you remove the element outright, youll cutoff other

keys placed through probing

Get around by having an attribute to indicate if the element has

Been removed or not

Make a tombtone - a way for computer to tell a NON EMPTY data loc is infact

not to be considered

Tombstone - marks a removed item. Stops probe removing/adding/finding from exiting prematurely when it reaches the tombstone

Chaining - Each element is the start of a linked list

AKA closed addressing hashing

Bc hash function determines the FINAL address of the el

In this case, you push the matching hash key to a linked list starting with the pre-existing hash key

IE if 21 hashes to element 0
BUT 56 also hashes to 0
    Build a linked list such that
        21 -> 56 -> (NULL)