Binary System Notes

Introduction

Most of us have worked in the deca system our entire life. Computers however use an entirely different number system to operate.

Why is it Important?

Computers at their most basic level use electricity to operate. Electricity only has two reliable states, on(1) and off(0). For this reason, computers use the binary number system, which only operate off two numbers, 0 and 1. Computers don't have the ability to use the deca system. Since computer science is the study of computers, it is helpful that we understand the math that computers use, so we can better understand their logic.

Lesson Notes:

The deca system is the number system we typically use. It runs off something called "Base 10". This means after 10 of each number's place, we increase the number of the next greatest magnitude. So we have 1, 10, 100, 1000, 10000, and so on.

If we add 1 to 9, the one's place gets reset to 0, and then the next magnitude is increased. So we are left with 10. If we have 99 and increase by 1, the 1's spot is reset to 0, the 10's spot is increased by one. There is already a 9 here, so it gets reset to 0 as well and then increments the next spot by 1. So we are left with 100. This makes it easy to do math, as each spot just adds an additional zero.

It may help to think of the numbers with leading 0's. So 00000998. If we add 1, it's now 00000999. Add one more, and we have to increment each 9. They get reset, and then the zero to the left is set to 1, 00001000.

The binary system however is different than this. Instead of having 10 numbers it can use, it only has 2. It has a 0 and a 1. This is due to how computers work at a basic level. They are only capable of reading whether a switch is on, or if it is off. They can't reliably read anything in between. This means all the math a computer does must be based off of this "on" (1) and this "off" (0).

A CPU processor has millions of these little switches all working in unison, combining and reorganizing the 1's and 0's to perform operations.

This binary system means that the number system works off of "Base 2" instead of "Base 10". This means it goes up in order of 2's.

Let's compare the deca and binary system with some tables. Below we have the number 1,578,483 broken out into it's magnitudes (number's places).

1	5	7	8	4	8	3
10^6	10^5	10^4	10^3	10^2	10^1	10^0
1,000,000	100,000	10,000	1,000	100	10	1

Each new position is an order of 10. So the 100's position is 10^2, the 1,000's position is 10^3 and so on. Since it is the deca system, we just move up by an order of 10 each time we run out of space in the current number. To get our final number, we just add up the top number multiplied by it's spot.

So in this situation we have 1,000,000 + 500,000 + 70,000 + 8,000 + 400 + 80 + 3 = 1,578,483

The Binary system however works a little bit differently.

1	0	1	1	1	1	1
2^6	2^5	2^4	2^3	2^2	2^1	2^0
64	32	16	8	4	2	1

The number above is 1011111, which is the binary representation of the deca number 95. Notice the differences between the first table and this table. Instead of 10^x, it is 2^x. This makes for a much smaller transition between each number. Where the 6th place represents 1,000,000 in the deca system, it only represents 64 in the binary system.

We can get the number above from just adding like with the previous table. So we have 64+0+16+8+4+2+1 = 95.

Converting Binary to Decimal

To convert from a binary number back in to a deca number, all you have to do is what we just did above. Just add up the magnitudes with a 1 in them. In this case it would 64 +0 +16+8+4+2+1 = 95. The number 1011 would be 8 + 0 + 2 + 1 = 11.

If you don't want to draw a table. You can just add the magnitudes into the equation. So 1011 would be:

(1*(2^3)) + (0*(2^2)) + (1*(2^1)) + (1*(2^0)) => (1 * 8) + (0 *4) + (1 *2) + (1 *1) => 8 + 0 + 2 + 1 = 11

Converting Decimal to Binary

Converting from a deca number to a binary number is a little tricky, but not too hard. To do this, you just subtract the largest number you can from the deca number, and add that to your binary number. It works a little like this.

Deca Number = **55**

First we look for the column that will take out the most from our number without going over. In this case it is the 32. We put a one in this column and subtract this number from our original.

55 – 32 = **23**

	1					
2^6	2^5	2^4	2^3	2^2	2^1	2^0
64	32	16	8	4	2	1

We then repeat this step over and over until our number hits zero. Any numbers that don't have a 1, get filled in as a 0.

23 – 16 = **7**

	1	1				
2^6	2^5	2^4	2^3	2^2	2^1	2^0
64	32	16	8	4	2	1

Note our largest number here isn't the 8's spot. That would go over our 7, so we go to the next spot of 4.

7 – 4 = **3**

	1	1	0	1		
2^6	2^5	2^4	2^3	2^2	2^1	2^0
64	32	16	8	4	2	1

3 – 2 = **1**

	1	1	0	1	1	
2^6	2^5	2^4	2^3	2^2	2^1	2^0
64	32	16	8	4	2	1

1 – 1 = **0**

	1	1	0	1	1	1
2^6	2^5	2^4	2^3	2^2	2^1	2^0
64	32	16	8	4	2	1

So our final number comes out to be 55 = 110111.

Additional Resources:

https://www.sqa.org.uk/e-learning/NetTechDC01ACD/page_11.htm

http://www.math.grin.edu/~rebelsky/Courses/152/97F/Readings/student-binary

Math Refresher Notes

Introduction

A quick refresher on some of the key math concepts we will see throughout the course. Don't worry if this goes over your head right now. We will retouch on these elements over and over.

Why is it Important?

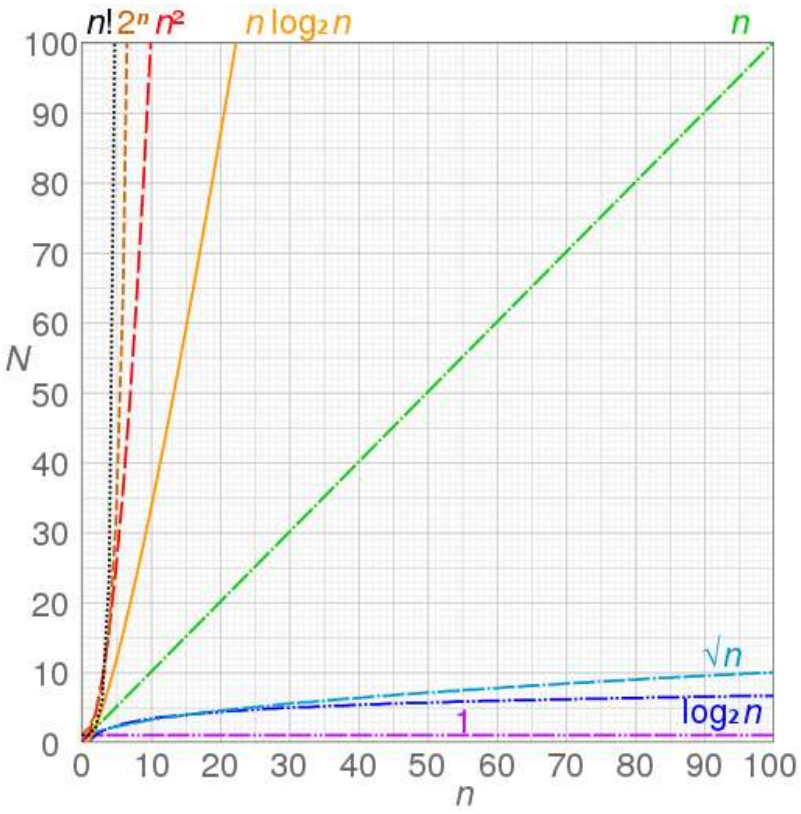
Computer science in its essence is applied mathematics. This means it has a strong foundation in many different types of math. Understanding the basics of some math functions will help to get a better picture of computer science as a whole.

Lesson Notes:

Logarithmic Functions:

Log (short for logarithmic) functions are commonly referred to in computer science. This is because they are the inverse of exponential functions. Where an exponential function grows more and more over time, a log function grows **less** and **less** over time.

Below you can see the difference between these functions. On the bottom you will see a log(n), and then it's inverse, near the top, the 2^n. Note how the log has a little two with it, this means it is using base two. Remember from the binary lesson, base 2 uses only 1 and 0. This is important when using with an online calculator. Make sure the log is set to base 2. Notice how the log function becomes almost horizontal, while the exponential function becomes almost vertical.



Let's say for example this is a graph of run times. So the bottom is how many pieces of data are inserted, and the left is how long it will take. You will notice with a log function, we can insert 100 pieces of data and have a run-time below 5. With an exponential function however, we only have to insert around 5 pieces of data before our run-time exceeds 100.

http://tutorial.math.lamar.edu/Classes/Alg/LogFunctions.aspx#ExpLog_Log_Ex1_a

Here is a good link that describes them in a slightly different way and has some practice examples. Understanding log functions in their entirety **IS NOT** required for this class. All you need to know is that they are efficient, and that they are the inverse of exponential functions. (We will build on them later when we talk about sorting algorithms and trees.)

Factorial Expressions:

Factorials are interesting expressions. The notation for them are n! . What this means is that we are going to multiply a series of numbers up until the variable n.

4! = 1 * 2 * 3 * 4 = 24

5! = 1 * 2 * 3 * 4 * 5 = 120

8! = 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 = 40,320

10! = 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10 = 3,628,800

As you can see, factorials grow at an astounding rate. If any of our functions ever reach this, they will most likely never finish. Almost, if not every algorithm in computer science can be accomplished in faster than n! time.

Algebraic Expressions:

We will encounter a few algebraic expressions throughout the course. They may look something like nlogn or 2nlog(n^2) etc. All this means is that we have a variable n that will be plugged in to the equation. Because we don't actually know how many pieces of data will go in to the algorithm, we use this n placeholder. (More about this in the next lecture).

Once we know the n value however, we can plug it in, and calculate the answer. For example, if n=32, we will then have nlogn or 32 * log(32). This would come out to 32 * 5 = 160 units of time.

Additional Resources:

http://tutorial.math.lamar.edu/Classes/Alg/LogFunctions.aspx#ExpLog_Log_Ex1_a

https://www.mathsisfun.com/numbers/factorial.html

https://study.com/academy/lesson/algebraic-function-definition-examples.html

N-Notation Notes

Introduction

In computer science we need a way to be able to compare different algorithms with each other. We typically use the n-Notation for this.

Why is it Important?

Run times of different programs are usually written in n-Notation. So if we want to look up how fast a certain algorithm is, we will need to know this notation to be able to understand it.

Lesson Notes:

N-Notation is represented as a function of n. This simply means that the program is going to be related to the variable n in some way.

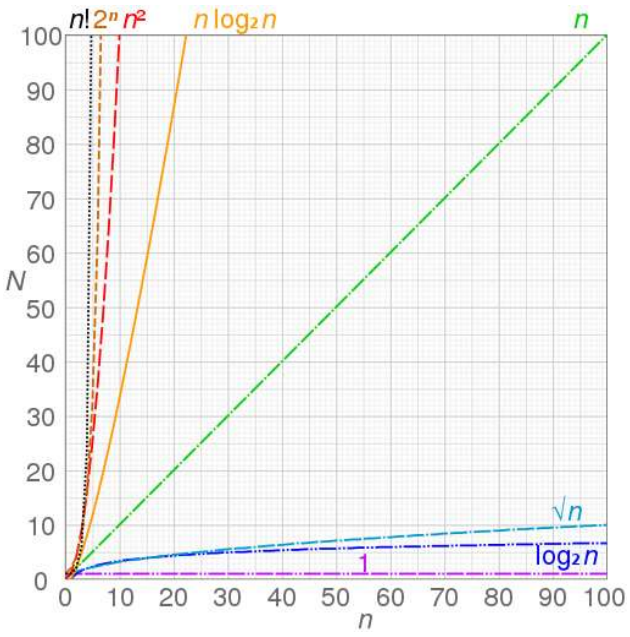
N-notation is NOT an analysis of how long an algorithm will take to run, but an analysis of how the algorithm will scale with more and more input data.

This method is based off looking at the input of the algorithm and analyzing how the algorithm works with this input. However, we don't particularly know how many pieces of data will come in at any given moment. What we do is replace this input number with a variable, n.

For example, let's say we have an algorithm which runs through the amount of Facebook friends you have. When we are writing the algorithm, we have no clue how many friends are going to be added as input. You could have 250, or 2,000, or 2. So instead of giving an actual number, we just say n pieces of data will be put into this algorithm.

(The n here is just a variable, it could be x, or w, or z, but we typically use n). It is usually a count of how many operations are being run by a program. This can be the amount of inputs, the calculations, or the outputs.

The reason we use n is because it will show us how our program might react differently to different amounts of data. We as computer scientists never know how many pieces of data our programs are going to handle. So instead, we look at how our program is going to grow with different amounts of inputs.



As you can see with this chart, the different values yield vastly different results. The difference between n and n^2 is substantial. If you brought this out to infinity, the difference would almost be a 90-degree angle. (As the difference between them would grow to nearly infinity)

We can use these different values to compare how our different algorithms might react to different amounts of data.

N =	N	N^2	Nlog(n)	1
1	1	1	1	1
10	10	100	33	1
100	100	10,000	664	1
1,000	1,000	1,000,000	9965	1

The above table represents the change. It inputs some different numbers as n and then displays what different n values would give. These numbers vary a lot with the given n-formulas.

For example, with only 1,000 pieces of data, the difference between n^2 and nlog(n) is from 1,000,000 to 9965. Even though they look decently close on the line-chart above, nlog(n) will only take 1% of the time as n^2. This difference will grow the larger our n becomes.

This is the key concept behind n-notation. Even though we don't know the specific amount of data that is coming through our program, we can atleast compare it to another that accomplishes the same task. If one runs at n^2 while the other runs at n, we know the n will be faster in ***every*** case.

Big O:

We usually don't use n by itself however. Typically we tie it together with a Greek letter to give it some context. Rarely does our program operate at the same timing for every single step. So instead of having exact values for things, we want to look at them in boundaries, or cases in which they are greater, less than, or equal.

Notation	Meaning
o(n)	Faster
O(n)	Faster or Equal
Θ(n)	Equal To
Ω(n)	Slower or Equal To
ω(n)	Slower Than

For this notation, we just use one of the Greek symbols above, with our n notation inside of the parenthesis. So for example, O(nlogn) would mean that the program is faster or equal to nlogn.

The most important notation above is the Omicron, or "Big O". The reason for this is because it gives us a worse case scenario. That means we know the absolute worse case scenario of our program, meaning if we plan for this, we won't have any surprises or weird corner cases that come up.

We can then compare the worst case scenarios of programs to see which are faster. Since this is the absolute slowest the program can run, we know that one program will always run slower than another program!

For example let's say that we have two programs, program A which runs at Ω(n) and program B which runs at Ω(n^2). Which of these two programs is better? Well all we know is that program A is slower or equal to n and program B is slower or equal to n^2. However that doesn't guarantee anything. Program A could end up running around n^3 or n! and meet the requirements of Ω(n). B could do the exact same thing, so their speed is still arbitrary. So with these symbols, we don't learn too much about the two programs.

However, if we used O(n) for A and O(n^2) for B, we can now compare the two. We know without a doubt that A will not run slower than n while B can run up to n^2. Therefore the faster is Program A, because it can never get slower than B.

Let’s use some numbers to drive this point home a little more.

Let’s say we have a time scale to determine how fast our algorithm runs. In this scale the closer to 0 we go, the faster our algorithm. We can use to show why Big O is typically the most useful of these notations.

ω(7) - The algorithm will run slower, or > 7. How much greater? 1,000. 1,000,000? There is no way to know. This algorithm could take years to complete the simplest of tasks.

Ω(7) - The algorithm will run "slower or equal" to 7, or >=7. Still could run in to infinity or really large run times.

Θ(7) - The algorithm is equal to 7, or = 7. This would be great. If we can guarantee an algorithm will always run equal to something we would use this. However, this is highly unlikely in the real world, so it's not used too often.

O(7) - The algorithm will run "faster or equal" to 7, or <=7. This is good. We have a limit on the algorithm. We know in all cases, 7 is the worst this algorithm can do. This can be used to plan with. No surprises will come up here.

o(7) - The algorithm will run faster than 7, or < 7. There is nothing inherently wrong with this, except that it's just less accurate than O(7). How much faster will it run? 6? 2?. We don't know the limit. Although we can still plan a worst case scenario with this, it's less accurate than O(7) which is why it's rarely used.

There you have it, BIg O notation. This is used all across computer science, and now you know how to read it!

Additional Resources:

https://stackoverflow.com/questions/487258/what-is-a-plain-english-explanation-of-big-o-notation

https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation

Fixed Array Notes

Introduction

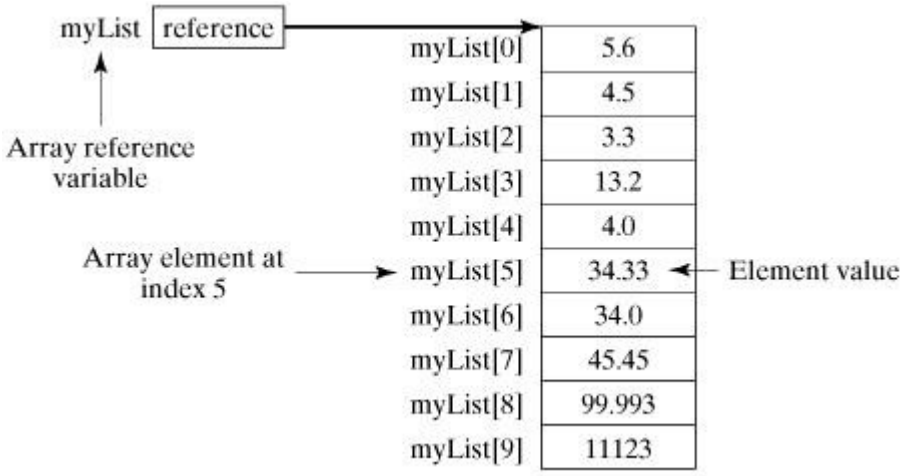
Arrays are a way of storing data in a linear sequence. This creates a fast-organized data structure.

Why is it Important?

Arrays are used everywhere. Because of their easy design and quick recall, they are widely used to store information during program execution. They can also be used to build more advanced data structures like stacks and queues.

Lesson Notes:

Fixed arrays are the most basic version of an array. They are created with a fixed size and stick to this size throughout their execution.



An array works by allocating a section of memory for storage. Then whenever we call the array, it returns the location of start of the array. We can then apply the appropriate index number to get to any point in that array instantly.

For example, in the figure above, the array myList is created and references the very beginning of the array. You can see this through the “reference” box that points to before the first element of the array.

We can then put a number in the [] after myList to get a specific element of the array. So if we put myList[5] we would get in return 34.33.

***Run Times*:**

Insertion Random: O(n) – It's easy to insert randomly anywhere in the array. Getting to the location of insertion take O(1) time. However, if you want to preserve order, there is a chance you will have to move O(n) elements to put the number there. Therefore, it's O(n).

Insertion Front: O(n) – Inserting in the front of an array will usually take O(n) time, as you have to shift up to n elements backwards to insert properly.

Insertion Back: O(1) – Nothing has to be shifted, so you can accomplish this in O(1) time.

Deletion Random: O(n) – Deleting randomly from the array is just like inserting randomly. You can get to and remove the element in O(1) time. However, if you need to delete an element and not create a hole, then this becomes O(n) as you will need to shift everything to cover the hole.

Deletion Front: O(n) – Same as Insertion, a shift of up to n elements will be required.

Search Unsorted: O(n) – If the array is unsorted, there is no way of knowing where the element is going to be. So at worst case, it's going to be a search of the entire array to find the element.

Search Sorted: O(logn) – If the array is sorted, we can keep cutting the array in half to find the element we are searching for. This means it will take at most logn operations to find our element. (Reverse exponential).

Additional Resources:

https://www.tutorialspoint.com/java/java_arrays.htm

https://www.w3schools.com/js/js_arrays.asp

Circular and Dynamic Array Notes

Introduction

Arrays can be improved from basic fixed sized arrays. These improvements can help make the arrays more flexible and efficient.

Why is it Important?

Through these improvements we are able to dynamically store data, as well as decrease the insertion time and deletion times.

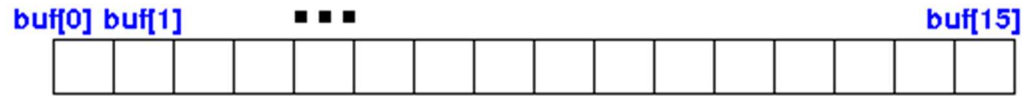
Lesson Notes:

Circular Array:

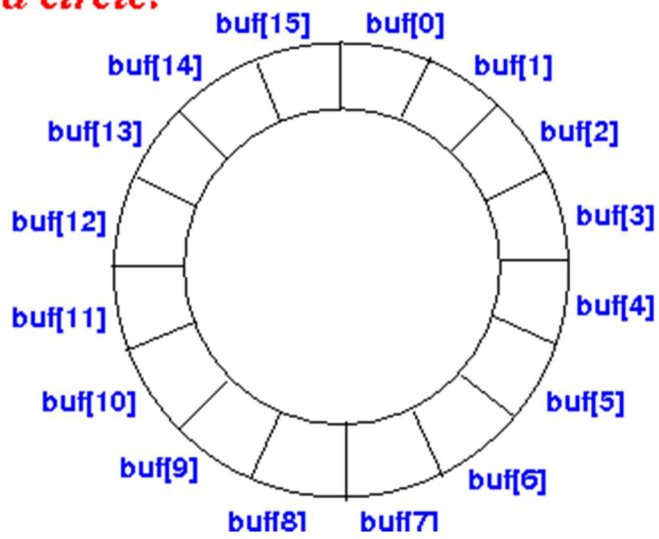
One of the major problems with a typical array is that we could reach O(n) whenever we tried inserting or deleting into the front. The entire array had to be shifted, causing a lot of data to be touched that didn't really need to be touched.

To improve this, we pretend that the array is circular. This makes it so we never need to shift data.

Array:



Pretend array is a circle:



So how do we pretend it's circular? We create two cursors. A "start" cursor, which points to the beginning of the array, and an "end" cursor which points to the end of the array. These cursors are then adjusted when an element is deleted or added.

For example, let's say we had some data in buff[0] that we wanted to delete. If we look at the typical horizontal array above, you will notice that when we delete this data, we will end up having to shift everything backwards to fill in the hole.

However, with the circular array, what we do is to move the "front" cursor to buff[1]. This makes buff[1] our new beginning of the array. We can now add or delete anything, and instead of moving the data, we just move the cursor around.

At some point the beginning of the array might be at buff[13] and the end might be at buff[6]. It just depends on how data is added and removed. Overall though, since we are only moving the location of the cursor, our run time improves to O(1) from O(n).

Dynamic Arrays:

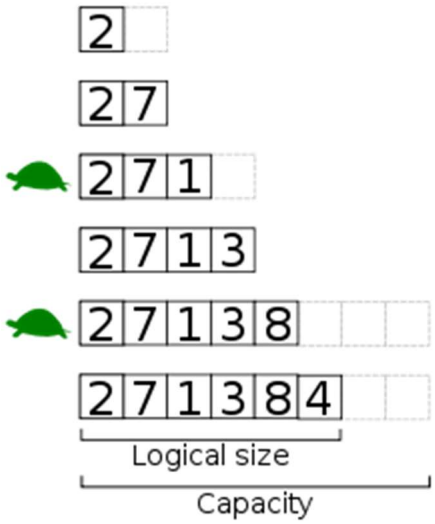
A dynamic array gives us the ability to expand our array whenever we run out of room. This means we don't need to know how much data is being input beforehand, making it a lot more "real-world" friendly.

Expanding our array each time we run out of data however runs in O(n) time. This is because once an array is allocated, it can't be expanded. So what we do is create a new array with a new size, and then transfer over all of the contents from the previous array.

This though requires we touch every single piece of data in the array, making the O(n) time. We can however improve this if we think about the problem a little bit further.

Instead of increasing the size of the array by 1 each time when we run out of data, we can instead double the size of the array.

For this, we use the terms **logical size**, and **physical size**. The logical size is how many pieces of data are actually allocated in the array. The physical size is the size of the array itself. So what we want to do, is every time our logical size reaches our physical size, we want to double the physical size.



This doubling provides a less and less frequent use of the O(n) copying operation, which overall leads to something called an **amortized** O(1) relationship. Amortize here just means averaged. This means is that it is technically O(n) because of the copy operation, but when averaged out to infinity, it more closely resembles that of O(1). Because of this, we can say it's basically O(1).

There is also a space-time complexity that needs to be thought of with this problem. The more you allocate after each "full" insert, the more chance there is of wasted space. It may speed up the run time, but may cost you a lot of storage space.

For example, let's say you have an array with 65,536 slots that gets filled up. If we want to add another element to this, we will then have an array which is 131,072 elements big. That is a gigantic leap. So this is usually optimized depending on the program.

This can all be expanded a bit farther. With some clever ingenuity, you can actually create a dynamic circular array, combining the best of everything mentioned.

This array would just be a dynamic array with front and end cursors. This would allow you to delete and add in constant time, and also to make sure that when the array runs out of room it wouldn't take up much time either.

Additional Resources:

http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/8-List/array-queue2.html

https://en.wikipedia.org/wiki/Dynamic_array

Singly-Linked List Notes

Introduction

Linked Lists are a core data structure that can link data together from multiple locations.

Why is it Important?

Linked Lists help us link data together from separate locations. In this way they are infinitely scalable and can contain multiple different types of data. This gives us a flexibility that arrays don't give us.

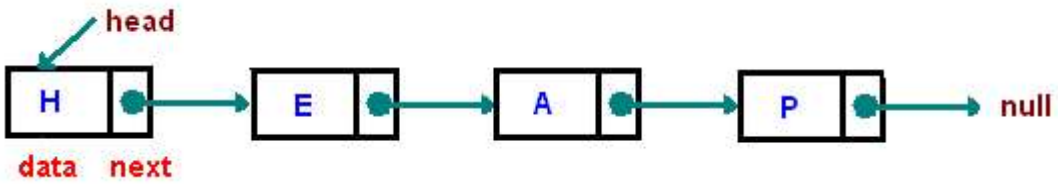
Lesson Notes:

Nodes:

Nodes are the building blocks of Singly Linked Lists. They are a data structure which holds a few key pieces of information.

The node usually contains a piece of data (a number, letter, location, etc) and then pointers to other nodes. In a singly-linked list it contains a pointer to the next node in the array. If it is at the end of the array, then it just points to NULL, which means nothing.

Linked List:



The very first node is designated as the "head" of the linked list. It is the node that is always pointed to when we call our linked list. This head is important, without it, we would lose our entire linked list.

An important difference between linked lists and arrays is that the data in the linked list aren't directly accessible.

For example, if we had an array of tempArray = [H,E,A,P], we could access the P by calling tempArray[3]. This would instantly return us the P.

With a linked list however, we can't do that. We only have the first node, and then where that node points to. This means that we have to traverse the entire list before we get to the P. So, if we want the 4###sup/sup### element, we will have to touch the H, E, and A before we get to it.

This means Linked Lists search times are typically slower than that of an array, even if the list is sorted, as there is no way to jump to a certain point.

Linked List Run Times:

Insert(Rand): O(n) - To insert at a particular location, one has to traverse the list up to that point to insert there.

Insert(Front): O(1) – We just move the head pointer to the new node and point the new node at the old head.

Insert(Back): O(n) – We will have to traverse the entire array to get to the back. (We will find a way to improve this a little bit later)

Delete(Random): O(n) – We must traverse the length of the element we want to delete

Delete(Front): O(1) – Just as easy as inserting, just have to remove the first element and repoint the head pointer

Search(Sorted): O(n) – Doesn't matter if it's sorted or not. We at worst have to traverse the entirety of the list to find the element. (And if the element isn't in the list, we have to traverse the entire list to figure that out.)

Search(Unsorted): O(n) – Exactly the same as the Sorted.

Additional Resources:

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked%20Lists/linked%20lists.html

http://www.geeksforgeeks.org/data-structures/linked-list/

Doubly List Notes

Introduction

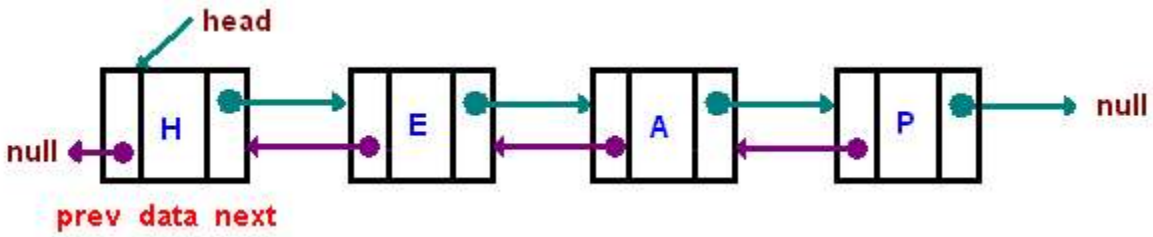
We can improve linked lists by making them doubly-linked as well as creating a tail pointer.

Why is it Important?

Giving us the ability to go forward AND backwards can be very helpful in a linked list. Also having the ability to start from the back can not only speed up access time, but it can make insertions to the back constant time.

Lesson Notes:

Doubly-Linked List:



A doubly-linked list is the same as a singly-linked list with one major difference, instead of just a "next" pointer within each node, there is also a "previous" pointer. Note, this will typically increase the memory needed for the list, as each node now has another pointer to keep track of.

This however allows one to not only go forwards in the list, but also to go backwards in the list. This can be helpful for a large amount of applications.

For example, let's say we want to print out the list back to front. With a singly-linked list, we would need to go to the last element, and print it. We would then have to start back at the beginning, go to n-1, then to n-2, and so on. We would have to keep restarting from the beginning.

So if we had the H->E->A->P example. If we wanted to print PEAH, we would have to first access H->E->A to get to P. We would then have to access H->E to get to the A and so on. This would make it so we would have to touch 10 pieces of data just to print out 5 pieces. This would scale up as well. If the array was 5 large, it would require 15 touches, 6 would require 21, and 7 would require 28.

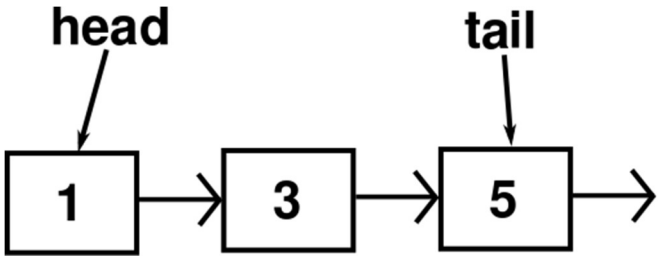
The formula for this is actually known as a linear summation, or n(n+1) / 2, where n is the size of the list. So with our H->E->A->P example, it would be n=4, or 4(4+1) / 2 => 4(5)/2 => 20/2 => 10.

If we multiply the n through, we get (n^2 + n) / 2, which you guessed it, shows us it's an n^2 formula.

If the list was doubly-linked however, we could go to the end of the list, print and move backwards until we get back to the front.

This means this particular operation will be reduced from O(n^2) to O(n), quite a change.

Tail Pointer:



A tail pointer is a small addition that can improve the deletion and insertion to the back of the list. By default, we have a “head” pointer. This, as seen in the picture above, points to the beginning of a list. It is used to give us a node to start our traversals from. Without it, we would have no way of getting to the beginning of the list, and therefore the list would be lost in memory.

We however can also create another pointer, a “tail” pointer. This will point directly to the very last node added to the list.

How does this help us? Well think of the two cases in which we want to delete or add information to the back of the list. Without a tail pointer we have to traverse the entire list until we get to the end, which by default is an O(n) operation.

So in the previous example H<->E<->A<->P (<-> signifies doubly-linked). Even with it being doubly-linked, it would still take O(n) operation to get to the back, and then O(n) operation to print from back to front. This would create O(2n) as the final run time. (This is simplified down to O(n) sure, but efficiency does matter at some point).

With a tail pointer however, we can just start directly on the end, and then add or delete from there. This takes the O(n) operation and brings it back to O(1). So now our print back to front will require a O(1) + O(n) operation, which is just a natural O(n) timing.

Insert (Back): O(n) -> O(1)

Delete (Back): O(n) -> O(1)

So with two small additions, we have taken an operation that would have been O(n^2), and reduced it down to an actual O(n) timing. This is what computer science is all about. Coming up with solutions to help make programs more efficient.

Additional Resources:

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked%20Lists/linked%20lists.html

Stack Notes

Introduction

Now that we know the basic data structures, we can begin using them to build different, more helpful data structures.

Why is it Important?

Stacks are used for a variety of operations in computer science. They can do anything from helping to navigate through a maze, to helping traverse a graph.

Lesson Notes:

Stacks are a data structure that are built off the backs of other data structures. They work off a set of important rules.

These rules are usually applied to either a linked list or array. It's like a specialization. Through the rules we are able to create a specialized structure with new pros and cons.

A stack works like a stack of trays in the cafeteria. You always grab from the top, not the bottom. This means you will have a relationship where the most recent element that has come, will be the first serviced.

Let's say 100 items come into a stack in order. So 1 is inserted, then 2, then 3, etc. In this case, 100 would be "served" first, then 99, then 98, then 97 and so on.This is helpful for something like an undo system on an application. Every step you make is inserted onto the stack. Then whenever you want to undo, you just "pop" off the top of the stack to go back in time.

With this pattern, we now have what is known as a "Last in First Out (LIFO)" relationship. This is also commonly referred to as a "Last Come, First Served(LCFS)" relationship.

Going along with the terminology. These are some key terms to know when working with stacks and queues.

Pop - To remove an element from a stack. (Can sometimes be used for queues).

Push - To insert an element onto a stack. (Can sometimes be used for queues).

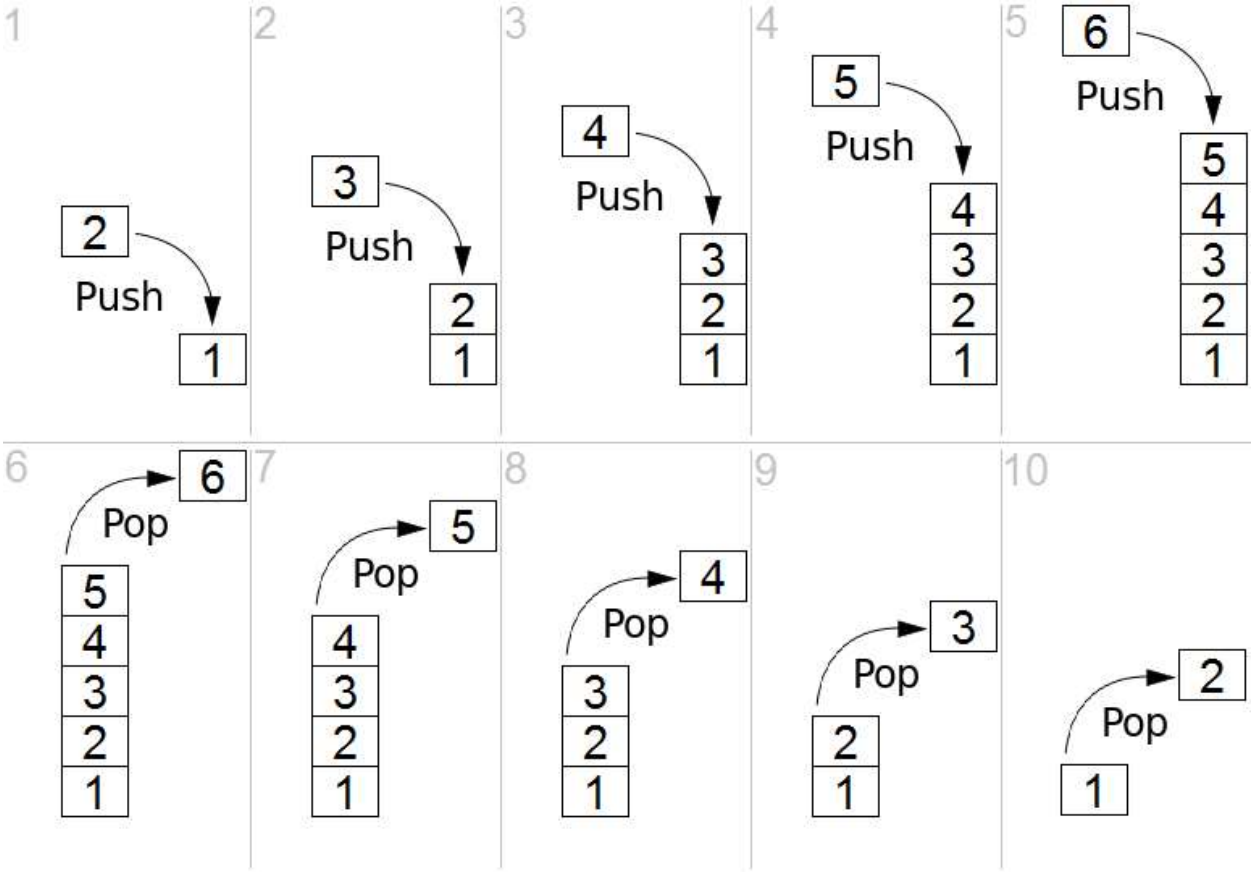
Enqueue - To insert an element onto a queue

Dequeue - To remove an element from a Queue.

FIFO - Means "First in First Out" It's the technical term for a queue-like data structure.

LIFO - Means “Last In First Out”. It’s the technical term for a stack-like data structure.

Let's take a look at how a stack operates through an example:



Notice how it looks like we are "stacking" books or trays up. We keep stacking. Then when we want to remove one, we take it off the top. **We never remove from anywhere else in a stack!**

Now, if we kept popping and pushing in a loop, we would get a situation where number 1 could potentially never be touched again. For this reason, these aren't typically used for scheduling purposes. Imagine if you used a stack to run your computer. Old windows would be frozen indefinitely as new requests keep getting serviced first.

In the next set of lessons, we will cover a data structure that is more apt to deal with situations like this.

Additional Resources:

https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

http://www.geeksforgeeks.org/stack-data-structure/

Queue Notes

Introduction

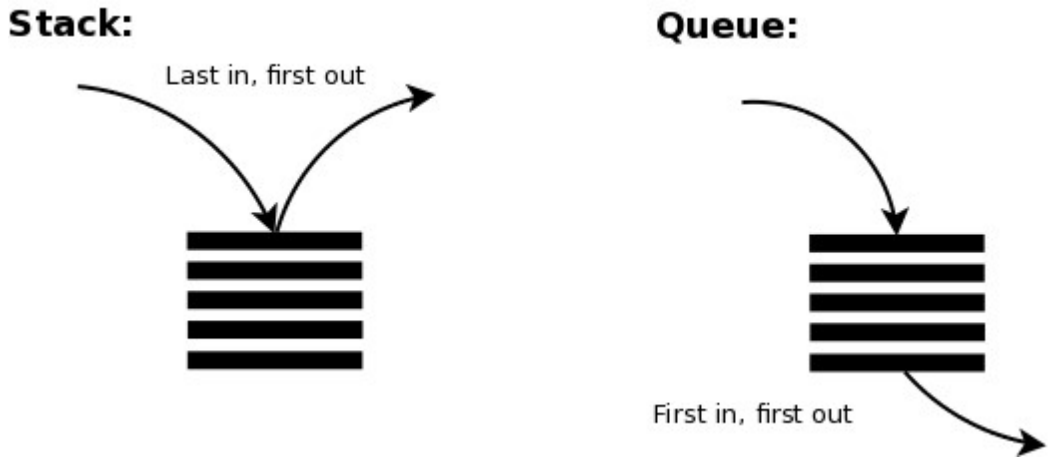
A queue is similar to a stack, but with one large difference, insertions and deletions take part from separate ends.

Why is it Important?

Queues, just like stacks, are used for a variety of different tasks throughout computer science. They are great for processing data where order is important, like CPU operations, or tree traversals.

Lesson Notes:

A queue and a stack are very similar, except that a queue inserts and deletes from separate sides. This creates a data structure that is a lot like a line/queue to buy tickets for a concert. Everyone lines up near the box office. The first one that arrives is the first one that is served. The last one to arrive is the last one that gets served.



This creates a “First In First Out” relationship with the data. Where in a stack there is a chance the first element could never get touched, in a queue it is a lot more "fair". The most recent to come in will not be removed until every piece of data that was there before it is removed.

The naming convention for removing and adding to a queue are typically dequeue and enqueue, respectively. However, it isn’t uncommon to see pop and push used for a queue as well.

Stack and Queue Run Benefits?

Why use a stack or queue, what makes them special? In computer science, modeling is very important. Through modeling and abstraction we can take a machine that only does 1's and 0's and turn it in to anything we want.

Stacks and queues are just another example of this abstraction. It's just another layer of rules to add onto a pre-existing structure, so that we can control the data in a slightly different manner.

Stacks are widely used in almost every aspect of computer design. Just through reading your email, hundreds of stacks are probably being implemented. Same with queues. Processors love queues as they give it a linear set of instructions to execute.

Stack and Queue Run Times

Now, let's take a look at some run times associated with these data structures.

Because these data structures are so specialized, we can actually get O(1) run time on all operations of both stacks and queues. This is because we are only accessing the "ends" of the data structures. So we don't have to worry about trying to get to the middle.

Linked lists are usually best for these structures, as they can expand indefinitely.

With a stack, we just keep inserting and removing from the front of the data structure. We learned in the previous lessons that both of these operations are O(1).

With a queue, we insert to the front and remove from the back. We can do this in O(1) time as well. (We would need to implement a tail pointer for this to work. It's simply a piece of data that always points to the end of the linked list).

Additional Resources:

http://cs.joensuu.fi/~appro/ics/05-03.php

http://www.geeksforgeeks.org/queue-data-structure/

Bubble Sort Notes

Introduction

Bubble Sort is arguably the simplest sorting algorithm. It works by repeatedly swapping adjacent elements that are in the wrong order.

Why is it Important?

A study of sorting algorithms always starts here. This is because the intuition behind Bubble Sort is really easy to understand, and it's good to analyze why this algorithm is not efficient. We can then use that to move in to the more advanced sorting algorithms.

Lesson Notes:

The algorithm works by "bubbling up" the largest value each time. It does this through simple swaps of data in the array.

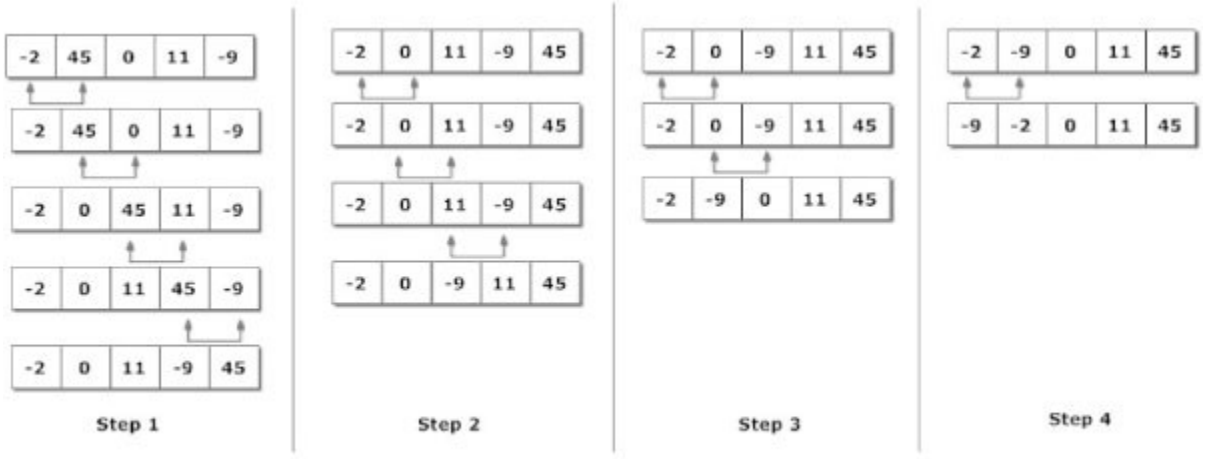


Figure: Working of Bubble sort algorithm

As you can see in the figure above, in each pass through the algorithm it starts at the left side of the array. It grabs the first number and sets it as the maximum. If the next number in the array is smaller than this number, it swaps their position. If the next number is larger, it sets the max to this number, and continues down the array.

Items are only swapped with adjacent numbers. So each pass will successfully "bubble" the next largest number up to the top of the array.

For example, above in step 1, it starts at the -2 and set's that to max. It then looks at the 45. Well the 45 is larger than -2, so it makes 45 the new max. It then looks to the right of 45 to the 0. It

sees that the 0 is less than 45, so it swaps the two. It does the same with the 11 and the -9. It is now the farthest to the right that it can go, so it stops and starts over again.

This program however is not very efficient. On average it will run at O(n^2) time. This is because if the array is completely unsorted, or sorted backwards, there is a good chance it will have to run through the entire array, which has n amount of elements, n amount of times. (Since only one element is sorted at a time. It has to keep running through the array over and over again).

Best Time: Ω(n) : This happens if the array is already sorted. It only has to run through the array one time. However, this is a sorting algorithm, so the chance the data is coming in sorted is very slim.

Average Time: Θ(n^2) : This happens because of the fact that it will most likely have to run through the array n times. Since the array is length n, this means that it will be nXn or n^2

Worst Time: O(n^2) : This happens because of the fact that it in it's worst case, it will have to run through the n array n amount of times. Since the array is length n, this means that it will be nXn or n^2. The worst case scenario for bubble sort is if the array comes in in reverse sorted order.

Additional Resources:

https://www.programiz.com/dsa/bubble-sort

https://visualgo.net/en/sorting

Selection Sort Notes

Introduction

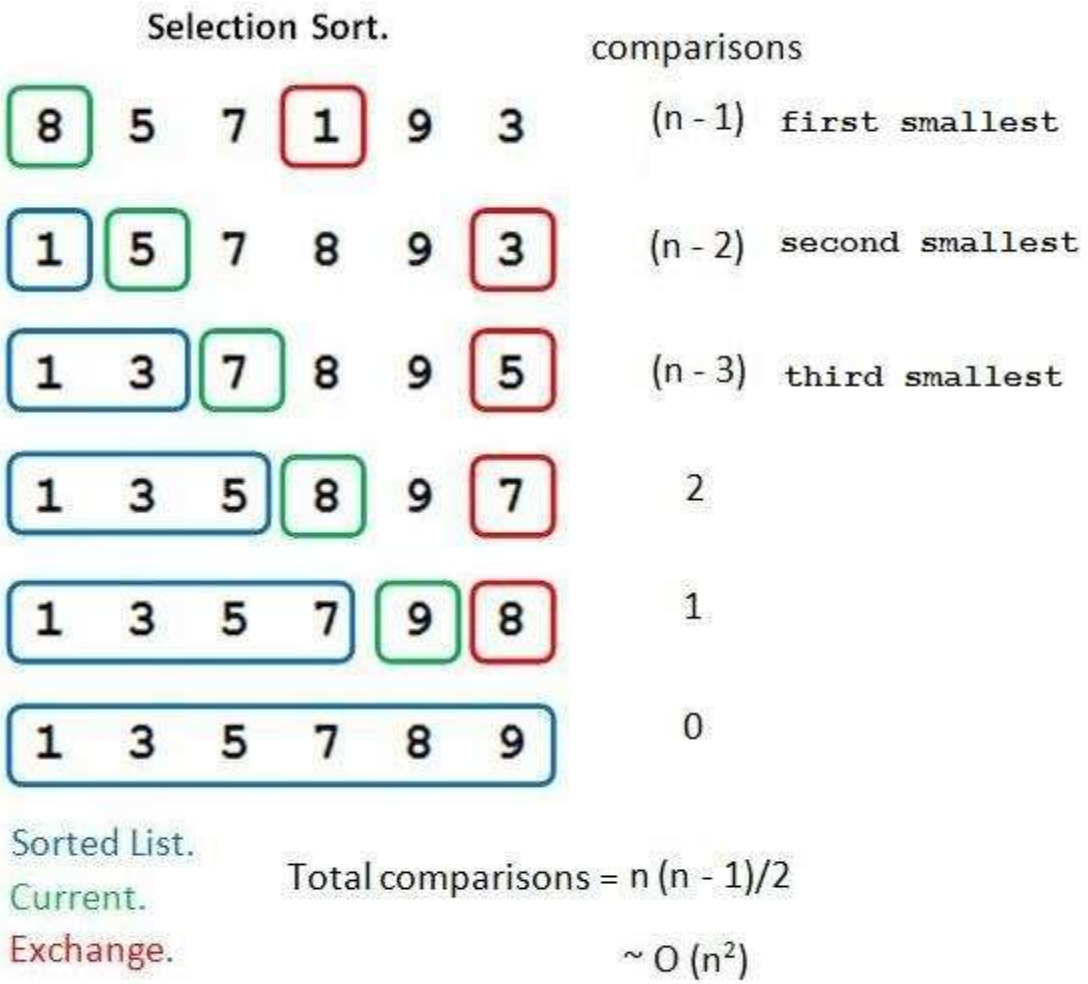
The selection sort is very similar to that of bubble sort. Instead of finding a max however, It repeatedly finds the minimum element from the unsorted portion, and puts it into a "sorted portion".

Why is it Important?

Selection sort is the next logical step from bubble sort. It does what bubble sort does, but has one large difference. Instead of searching through the entire array each time, it creates and only searches an "unsorted portion".

Lecture Notes:

The algorithm works by creating two subarrays within an array. The first subarray is the portion that is already sorted. The second is everything that is unsorted. Typically this is kept track of with an index number. So if our index number was 3, then everything to the right of and equal to 3, is the current sorted portion.



As you can see in the figure above, with each operation, the sorted portion grows and the unsorted portion shrinks. The algorithm works by finding the minimum number, then swapping it with the array index closest to the sorted portion.

For example, in the first example, 8 is the number that will be swapped. This is because the sorted portion doesn't exist yet, so the 8 is the closest to the left. We then go through the array and find the minimum value, which in this case is 1. We swap 8 and 1 and do it again.

The 1 is now in the sorted portion. We move to the closest value to the sorted portion (So one to the right of the boundry of the sorted portion) which is the 5. We then look for the smallest number in the unsorted portion. We find the 3, and swap the 5 and 3. Now the sorted portion is two big. We continue this process until the entire array is sorted.

This although slightly faster than bubble sort, doesn't improve the run time by any large magnitude. It still is on average a runtime of n^2. You can see the math for this under the

diagram. It is still making n(n-1)/2 comparisons. This reduces to (n^2-n)/2 operations. Remember when we go to infinity we grab the largest magnitude. So in this case, the n^2 is the only thing that stays.

Best Time: Ω(n^2) : Even if the array is already sorted, selection sort must run through its entire algorithm to figure this out. This means it will generate an already sorted portion, and only increase it by 1 on each operation. So the total run time will still come out to be n^2

Average Time: Θ(n^2) : Same with the average time. It doesn't matter how unsorted or sorted the array is, it will run through all operations each time. Although the magnitudes of this and bubble sort are the same, this one is technically faster because it reduces the size of the search area each time, unlike bubble sort which searches the entire array each time.

Worst Time: O(n^2) : Same with the worst-case time. It will use the same operations no matter how sorted or unsorted the array is. No one case is better or worse for selection sort.

Additional Resources:

https://stackoverflow.com/questions/15799034/insertion-sort-vs-selection-sort/15799532

https://visualgo.net/en/sorting

Insertion Sort Notes

Introduction

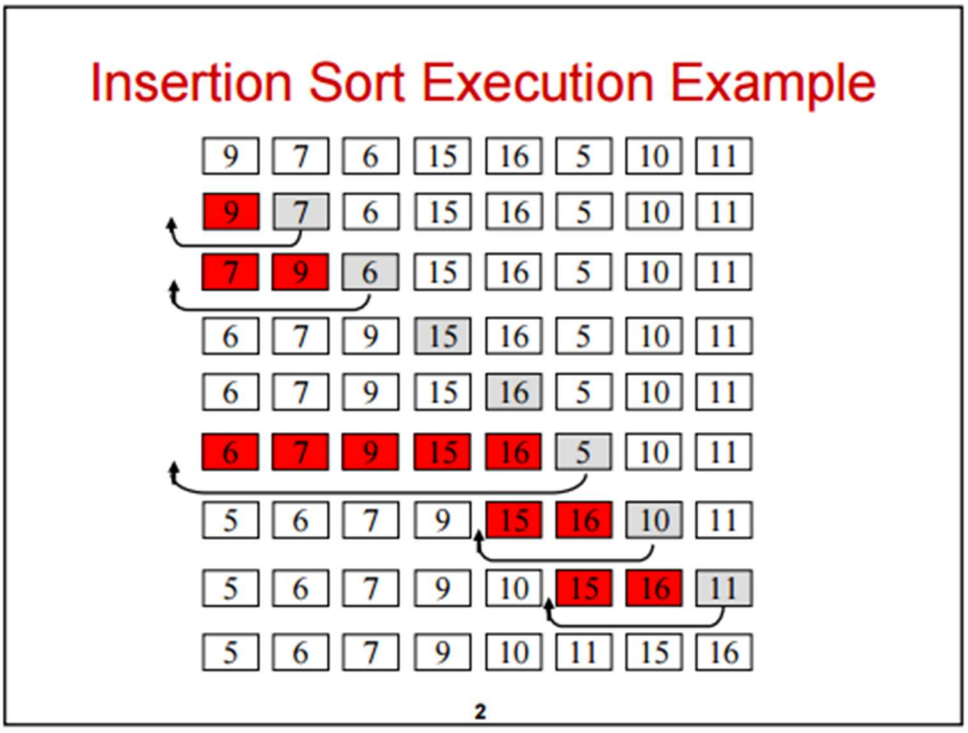
Insertion sort is another take on sorting elements one at a time. This algorithm sorts much like a human being would sort cards.

Why is it Important?

Insertion sort is another important sorting algorithm that is easy to follow along with. It is the next sort that is typically covered in a Computer Science curriculum.

Lecture Notes:

Insertion sort works similar to that of selection sort, in that it has a sorted and unsorted portion. Instead of finding the minimum in the unsorted portion, it takes the closest number to the unsorted portion and inserts it in to the sorted portion. (Hence the name, insertion



sort)

So, in the example above, it begins by making the 9 the first member of the sorted subarray. It then looks to the element that is to the right of the sorted subarray. In this case it is the number 7.

It grabs this number and inserts it in to the sorted portion. Since it is less than 9, it inserts it behind the 9.

Our sorted portion is now 2 big. We then take the next number that is to the right of the sorted portion. This is the 6. We insert that to the left of the 7, as 6 is less than 7. Our sorted portion is now 3 large. We look at the next number which is a 15. This number however is the largest number in the array, so we just keep it where it's at and move to the next number. We continue this until the array is sorted.

This algorithm, although smarter than the other two, is still about the same magnitude of run time. The best case scenario is better than that of selection sort, as if the number to the right of the sorted subarray is larger than the entire sorted array, then it just moves the sorted array over to fill this number. It then just moves on. If the entire array comes in sorted, then it just does 1 pass through the entire array and returns that it is sorted.

Best Time: Ω(n) : If the array comes in sorted, it can do just one pass to figure this out. This makes it's best case scenario, an already sorted array, run at n time.

Average Time: Θ(n^2) : The average time of this is similar to that of selection sort. On average we have to search the array 1 less time each iteration. So the run time is generally around n(n-1)/2. This means that It will simplify to (n^2-n)/2 which means it will be n^2.

Worst Time: O(n^2) : This is when the array is reverse sorted. The array will have to insert and swap backwards for every single number in the array, making it have to make n operations, for the n sized array. In other words n*n which is n^2.

Additional Resources:

https://stackoverflow.com/questions/15799034/insertion-sort-vs-selection-sort/15799532

https://visualgo.net/en/sorting

Quick Sort Notes

Introduction

Quick Sort is our first "fast" sorting algorithm. It divides the data cleverly to reduce the run time of the algorithm.

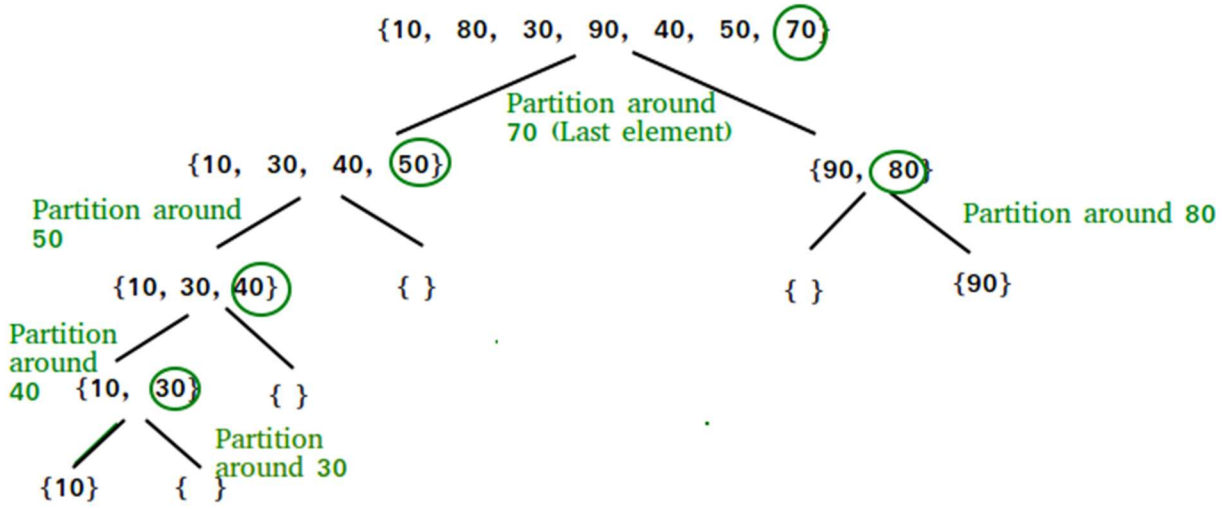
Why is it Important?

Quick sort is the next step in sorting algorithms. This sorting algorithm uses a smart way to divide the data so that it can reduce the run time. This comes at the cost of added complexity.

Lecture Notes:

Quick Sort works off picking a "pivot" point. All numbers less than the pivot point go to the left, and all numbers greater than the pivot point go to the right. It then reapplies this algorithm to each side of the pivot.

This creates an algorithm which implements a divide and conquer approach to sorting. Because of it's ability to split up the sorting, it is able to sort in a log(n) like manner.



As you can see, as the algorithm splits itself with every step, it is slowly creating a sorted algorithm. No longer does it need to run through the array for each number to be sorted. Now it can just keep breaking up the numbers until they are all sorted.

If we look at the example above, we can read it from left to right and have a sorted array. We have {10,30,40,50,70,80,90} Each level of this algorithm takes n amount of time. Because of the way we split up the information however, there will only be log(n) levels. This means the general

run time of the algorithm is going to be the number of levels multiplied by n, or nlogn, a strong improvement on the n^2 we have been seeing.

With this algorithm however, the pivot point is important. If we keep picking pivot points that don't split up the data, then we will not divide the data correctly. Instead of log(n) levels, we have the possibility of getting n levels. This means it will come out to n X n again, getting us the n^2 time.

Best Time: Ω(nlogn) : The best case in this scenario is if we pick a perfect pivot point to split the data. It doesn't matter if the array comes in sorted or not. The pivot point is what matters the most. If we pick a pivot point that perfectly splits the data each time, then we will have log(n) levels, and therefore the nlogn run time.

Average Time: Θ(nlogn) : The average time happens when we choose a decent pivot point. As long as we can split up the data and let the program divide and conquer, we get an nlogn time.

Worst Time: O(n^2) : The worst-case scenario happens when we choose a bad pivot point. Instead of dividing the work up to log(n), we keep only splitting out 1 number each time. (The number is larger or smaller than the rest of the array). This makes it so we have roughly n levels. Therefore we end up having the n X n relationship, instead of the n X logn relationship.

Additional Resources:

http://www.geeksforgeeks.org/quick-sort/

https://visualgo.net/en/sorting

Merge Sort Notes

Introduction

Merge Sort is the fastest algorithm we are going to be analyzing. (Fastest on average, quicksort is technically faster, but has the problem of the n^2 worst case). It has a way of dividing the data up like in quick sort, except that it doesn't require selecting a pivot point to get the nlogn run times.

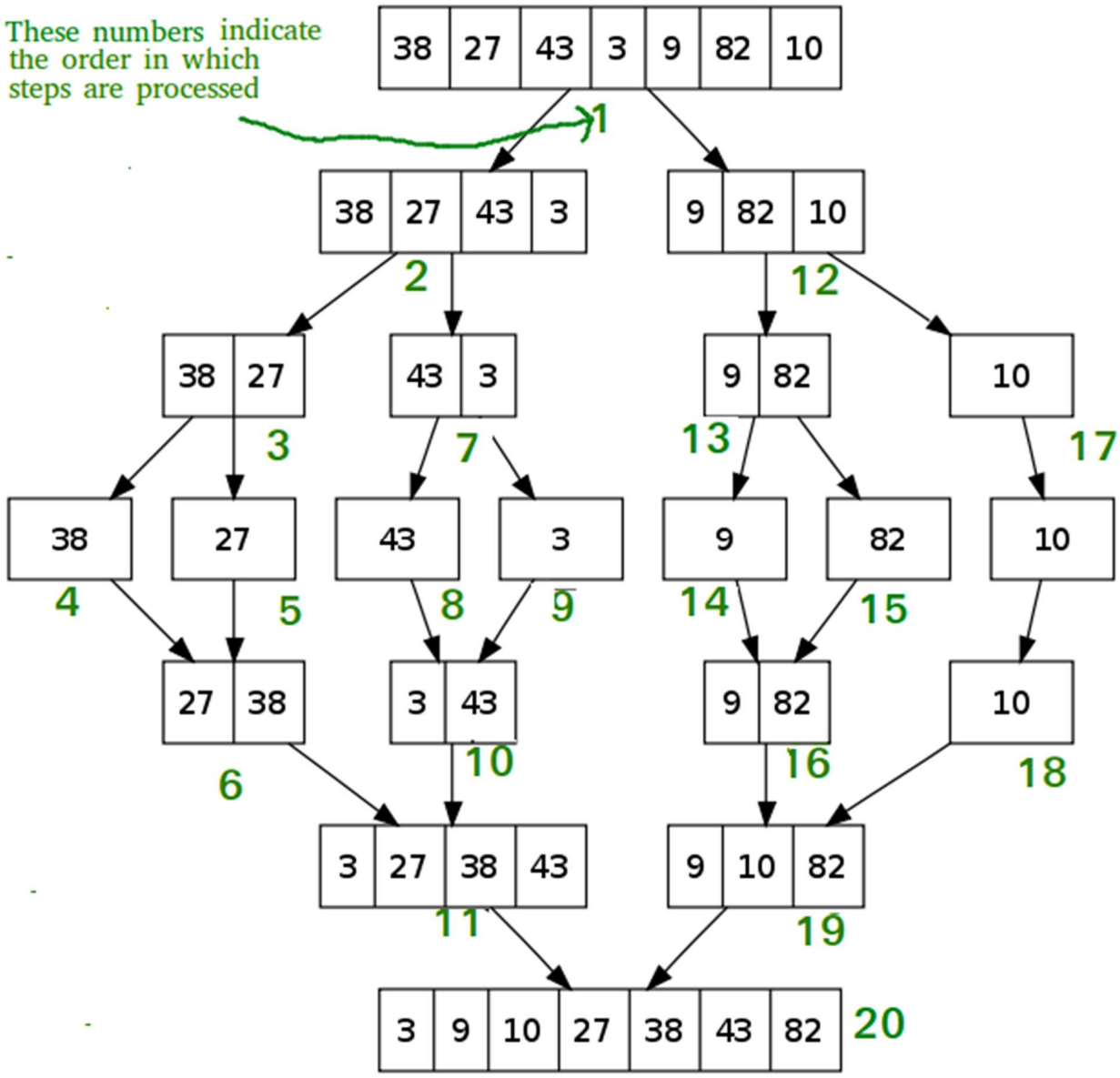
Why is it Important?

Merge sort is one of the fastest comparison sorts. It runs at nlogn for all run times, making it reliably fast at splitting up and sorting data.

Lecture Notes:

Merge sort is one of the fastest algorithms for sorting using comparisons. However, this also makes it quite complex as well. It uses the divide and conquer methodology used in the quick sort algorithm.

It works by dividing up the data in to smaller and smaller chunks. It then recombines them in to a sorted algorithm.



The array breaks down the data until it is in single unit subarrays. It then recombines them using cursors that only have touch each element once for each combination.

So each level is going to be at most n amount of “touches”. Because of the way it breaks up the algorithm, it has logn amount of levels.

This means that is will always run at nlogn timing, as there is no way to break it up worse than logn levels. So it will be n amount of touches per level multiple by logn amount of levels, giving us an even nlogn no matter what array comes in.

Best Time: Ω(nlogn) : The same algorithm is applied no matter what type of array comes in. This means it will always be nlogn

Average Time: Θ(nlogn) : The same algorithm is applied no matter what type of array comes in. This means it will always be nlogn

Worst Time: O(nlogn) The same algorithm is applied no matter what type of array comes in. This means it will always be nlogn

Additional Resources:

http://www.geeksforgeeks.org/merge-sort/

https://visualgo.net/en/sorting

Stable Vs Non-Stable Notes

Introduction

Sometimes the order of data within data structures is important. Certain sorting algorithms adhere to this importance, while others don't. If a data structure takes order in to account, we call it stable, if it doesn't, we call it not stable or non-stable.

Why is it Important?

It is important to understand what stability is. There are a lot of cases where stability matters, and understanding which sorting algorithms will get you a stable or non-stable result is important.

Lecture Notes:

Say we have a data set that contains groups of information. It looks like this:

{

(Bob, A)

(Chris, B)

(Kathy, A)

(Phillis, B)

}

This algorithm is already sorted in to alphabetic order. Now, let's say we want to sort this algorithm based on the letters on the back end, so A's come before B's.

If we use a non-stable sorting algorithm, we could very well get a result that looks like this:

{

(Chris, A)

(Phillis, A)

(Kathy, B)

(Bob, B)

}

If you notice, the A's and B's have been sorted, however the order of the names wasn't respected (Kathy now comes before Bob). Because of the non-stable nature of the sorting algorithm, we lost the alphabetical grouping of the names.

This is situation where a stable sorting algorithm would be important. If we applied a stable sorting algorithm to this group, we would instead get:

{

(Chris, A)

(Phillis, A)

(Bob, B)

(Kathy, B)

}

Now, not only are the names sorted by A's and B's, but they are also sorted with respect to the order they came in, which was alphabetical.

Sorting Algorithms:

Bubble Sort: Stable – This algorithm is stable because it just swaps the largest value up the structure to the top. If two objects are the same, no swap takes place. This means equal values that were to the left will stay to the left.

(2a 1 2b) -> (1 2a 2b) (The 2a will never swap with the 2b because swaps don't take place when two values are equal)

Selection Sort: Nonstable – By default selection sort isn't stable. This is because it takes a number and swaps it to the left "sorted" side. This gives the possibility that a number can be swapped behind another equal number.

(2a 2b 1) -> (1 2b 2a) (The 2a was swapped with the lowest number 1 which was on the right side of 2b. The swap takes place, and the order is not preserved)

Insertion Sort: Stable – Insertion sort uses a similar swapping mechanism as Bubble sort. It starts from the right while swapping, but never swaps equal values, meaning there is never a chance for elements positions to be flipped.

Quick Sort: Nonstable – Quick sort uses a splitting mechanism to order to sort. If this "pivot" is a number which has a duplicate, there is a good chance the order will be broken.

(2a 2b 1) -> (1 2b 2a) (In this scenario, 2b was chosen as the pivot, anything less than it went to the left, and anything greater than or equal went to the right. 2a is equal so it went to the right, breaking the stability of the algorithm)

***Merge Sort*:** Stable – Merge Sort splits up the data and recombines it in a way that grabs the smallest element and sticks it back in to the array. It however adheres to the order of values, giving preference to values that are in the left subarray to maintain order.

Additional Resources:

http://www.geeksforgeeks.org/stability-in-sorting-algorithms

https://visualgo.net/en/sorting

Tree Notes

Introduction

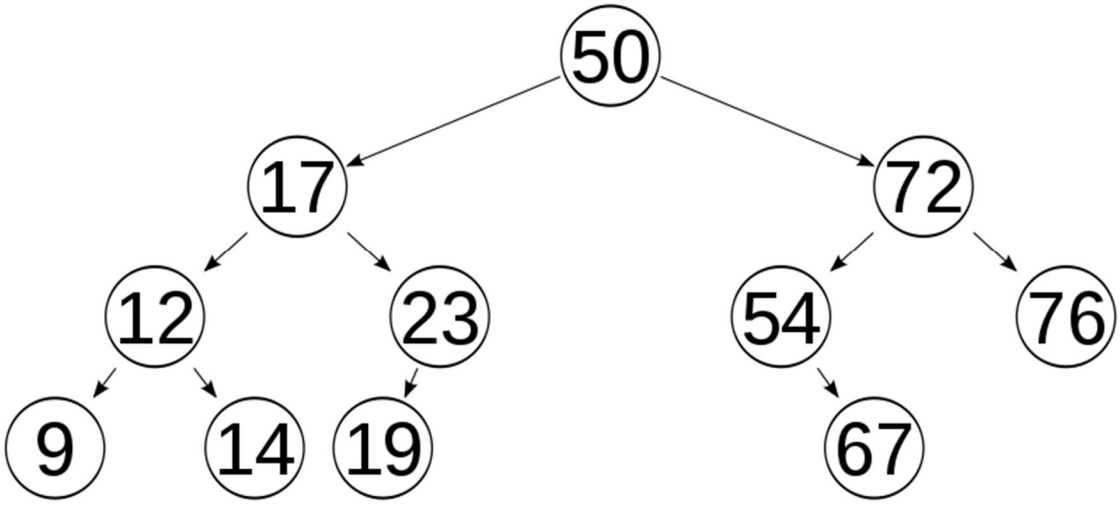
We can improve linked lists by making them doubly linked as well as created a tail pointer.

Why is it Important?

Trees are some of the most used data structures in computer science. They link information together in a relational way. You can use them to sort, to store information, to make inferences, and so much more. They are even used in databases to help index (find) the data in the database.

Lesson Notes:

Trees



Trees are known as a hierarchical data structure. Instead of pointing from one piece of data to the next, a tree has many different paths that you can take from each node. This creates a system of parents, and children.

Parents are the nodes that are directly above another node, while children are ones that are directly below. For example, node 23 is a child of node 17, and node 17 is therefore the parent of node 23. However, the relationship is only a direct relationship. So, node 67 is not a child of node 23, even though it's one layer lower.

The layers of the tree are also important. They are known as the height of tree. Trees are also typically one directional, so you go from parent, to child, to child, to child, etc. You don't typically go back up the tree for any purpose, except maybe a printing of the tree/

You also have nodes called “leaves”. These are the nodes without any children, meaning they are at the end of the tree structure. Once you reach them, you cannot go any further in the tree.

Tree Structures:

Trees can have a variety of structures. They can have a set number of children or have an infinite amount. It all comes down to how you want to use them. A special type of tree however is known as the Binary Search Tree (BST).

Binary Search Tree:

A binary search tree is a tree with these rules:

- Each node can only have at most two children
- All right children must be greater than
- All left children must be less than or equal to. (You can put the equal to on the right or left)

Through these rules we create a tree which can actually be used to search for things. Because we know the above rules, we can actually traverse the tree to find a number. The above picture is a binary search tree.

Let’s see if 19 is in the tree. We take the root node (the one at the very top) 50, and ask, is 19 greater or less than this node of 50? Well, it’s less. So we look at the left child. 50->17. We then ask the same question, is 19 greater or less than 17? It’s greater, so we take the right child. 50->17->23. Now is 19 greater or less than 23? It’s less than. We take the left child, and there we have it, the number is in the tree.

If the number were for example, 22, we would do the same procedure, and find out that when we tried to look at the right child of 19, there is nothing there. We would then return that no, 22 is not in the tree.

So with this ability to search the tree with only touching a few elements, we get some great time efficiencies with this.

(This assumes random order of inserting. If you insert a series of consecutive numbers, you will grow the tree in a straight line, making all operations the same as a list, so O(n). There are more advanced trees known as red-black trees and AVL trees that make sure this never happens. But they are quite complex)

Search: O(log n) – We only have to touch log elements (the height of the tree) to figure out if the element is in the list or not.

Insert: O(log n) – Same with inserting in to the tree. We ask the same questions as above, and find the empty place in the tree, and insert there.

Delete O(log n) – Same as the rest, we can delete by just asking the same questions.

Additional Resources:

https://www.geeksforgeeks.org/binary-tree-set-1-introduction/

https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/

https://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm

https://www.cs.auckland.ac.nz/software/AlgAnim/AVL.html (AVL Tree if you are feeling adventurous)

Heap Notes

Introduction

Heaps are a subdivision of Trees with a few rules. These rules, just like with Stacks and Queues, give these data structures many essential roles.

Why is it Important?

Heaps are used for a variety of applications. They can act as priority queues, help with graph traversals, and provide a data structure that makes finding the extreme value O(1) time.

Lesson Notes:

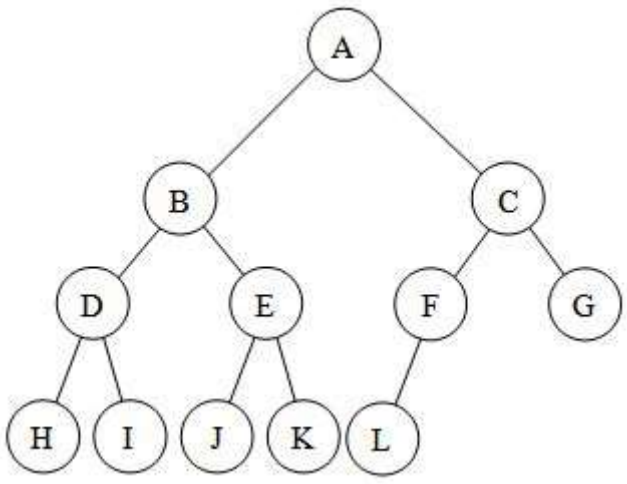
Heaps in essence are just stricter trees. They have all the same properties of trees, with the additional “Heap Property” rule added.

The heap property is a rule which gives a relationship between the parent and child nodes within a tree. It states that the parent node must always be either greater, or less than its children. If it has to be greater, then it is a max heap, if less, a min heap.

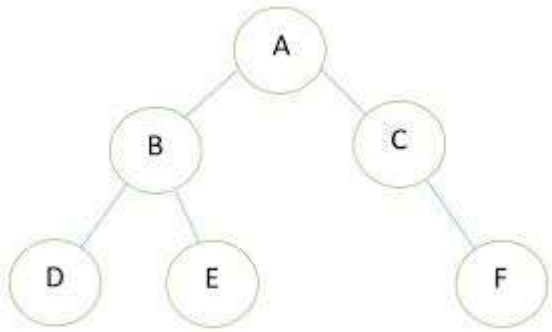
This property propagates throughout the tree. Since every parent has to be greater or less than, the root, or the top of the tree, must be the greatest/least.

The simplest heap is the binary heap. This is a heap in which each node of the heap has only two children and must maintain a complete tree pattern.

A complete tree is a tree which every level, expect the last, is filled, and all nodes are as far left as possible.



This is an example of a complete tree. Notice how it's completely filled, and the last level fills from left to right.



This is an example of a tree which isn't complete. The F should be shifted as a left child of C to make this complete.

Due to the structure of a heap, most of the run times are logn, as it only takes the height of the tree to get to any given node. We have previously learned that trees are built in a structure that lends this height to be equal to logn time.

Run Times:

Insert: O(logn)

Delete: O(logn)

GetMin: O(1)

Applications

Heaps are great for priority queues, as the most "important" task will always be at the top, and really fast to retrieve. They are also used in more complex algorithms like Dijkstra's Shortest Path Algorithm.

Additional Resources:

https://www.geeksforgeeks.org/binary-heap/

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Binary%20Heaps/heaps.html

Graph Terminology Notes

Here are some of the terms we have covered.

Vertex (vertices) – A point on the graph. We have called these “nodes” in the past. Usually represented as letters or numbers in theory.

Edge – A connection between to vertices. Represented by naming the two vertices. So (A,B) would be the edge between Vertex A and Vertex B.

Set - A group of items that have no importance or order to them. So {A,B,C} is equivalent to {B,C,A} and {C,B,A} etc. The order doesn't' matter. They are just a group of items that share something in common.

Cyclic – A directed graph which has at least one cycle. A cycle is where you can start at one vertex, and arrive back at that vertex through the graph. So A->B->C->A. This path starts at A, and ends at A.

Ayclic – A directed graph which doesn’t have any cycles. Once you touch a vertex, there is no way to get back to that vertex.

Connected – A graph in which each vertex is connected to together by some path. So from any node, one could follow the edges and make it to every other node.

Disconnected – A graph in which every single vertex is not connected. This could be a single vertex by itself or a bunch of smaller connected graphs. The point is atleast one of these vertices are not connected with the rest.

Adjacency – When two vertices are connected to one another. So if we have the edge (A,B), A and B are adjacent to one another.