Midwestern State University

CMPS 3013: Algorithms

Course Notes

Professor: Griffin

Spring 2021

Last Updated: April 20, 2021

Contents

1 Introduction

This course is called "Advanced Structures and Algorithms" and its essence is learning how to create complex structures along with the algorithms that provide efficient ways to store and retrieve data from within those structures. The Efficiency of an advanced structure is something you will learn to measure by evaluating the **run times** (aka costs) of each algorithm we explore, and how the data structure used for storing the data may impact overall efficiency.

In my opinion, this course is your first "real" programming course. Some students feel that data structures (1063) was a tough course, and I don't disagree, however this course will require you to use all the skills you gained ... and then some. If you struggle with arrays, pointers, or linked lists you may want to come see me, or review the following materials:

1. Arrays
2. Pointers and Memory
3. Linked List Basics

1.1 Array vs List

I just eluded to the fact that your previous course had a goal of getting you comfortable with certain concepts. One of those concepts was: "Do I use an array to store my data? Or do I used a linked list?" In computer science we use the terms: **array based structure** and **list based structure** for our two choices on how to store data in memory. However, to create list based structures you need to "link" memory locations, and so you were introduced to **pointers** which allowed you to create **linked lists**. Both arrays and linked lists have pros and cons associated with them, and understanding the pros and cons of each structure will help you understand why certain algorithms work better when implemented using one or the other. There is not a single best choice, an array can do anything a linked list can, and vice versa. But in certain situations, one will be better than the other. This course will help you know when to choose one over the other. Below is a summary of both:

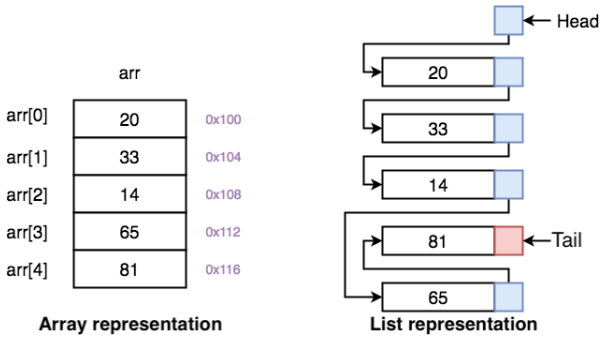
Array:

1. Direct access (random access) using subscript.
2. Growing and shrinking is costly.
3. Bad for lots of insertions and deletions (because of empty slots and need for shifting).
4. Good for searching (if items are ordered).

Linked Lists:

1. Access requires traversal using a pointer.
2. Grows and shrinks easily.
3. Good for lots of insertions and deletions (because of previous point).

4. Bad for searching (linear only).



Comparison	Array	Linked List
Generic	Contiguous memory locations of fixed size.	Dynamically allocated memory locations linked by pointers.
Size	Fixed at allocation.	Grows and shrinks easily.
Order of the elements	Stored consecutively	Stored randomly
Accessing	Direct access using subscript.	Sequentially accessed from head or other pointer.
Insertion and deletion	Slow if shifting is required.	Easier, fast and efficient.
Searching	Binary search (if ordered values) and linear search.	Linear search only.
Memory required	Size of data stored.	Size of data stored + pointer.

Table 1: Array VS Linked List[12]

1.2 Stacks and Queues

Something else you should have taken away from your previous CS course is the concepts of two simple data structures: 1) **Stacks** and 2) **Queues** . Not only the concept of how they work (LIFO v FIFO), but also how to implement either of them using an array or a list.

1. Stacks and Queues Helper Material

There are many other things you should have taken away from your data structures course, but to continue in this class with some comfort, you really should comfortably grasp the following. Each of the items is a link to a gist with example code.

1. Array Based Stack
2. List Based Stack
3. Array Based Queue
4. List Based Queue

If you can comfortably write all of those combinations basically from scratch, you are in a good position to continue. If not, practice!!

1.3 Algorithm vs Data Structure

We have laid the groundwork and are ready to discuss what *advanced structures & algorithms* really are. They are (to put it another way) *advanced data structures* ***and the*** *algorithms used to manipulate them*. They are two separate concepts, but in this context they are highly related. Lets look at each term separately.

Data Structure: In computer science, a *data structure* is a data organization, management, and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data[17].

Algorithm: In mathematics and computer science, an *algorithm* is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation. Algorithms are always unambiguous and are used as specifications for performing calculations, data processing, automated reasoning, and other tasks[16].

Both of these fundamental blocks are needed to solve problems in computer science, and are also why our field is so tightly coupled with mathematics. A good data structure allows for the efficient storage and retrieval of data. But to make this happen we (typically) need to follow a sequence of well-defined instructions to guarantee *unambiguous* behavior by our data structure [1]. What exactly does that mean? Let's use an example to clarify.

1.4 Stack vs Queue Example

Let me explain how the two relate by providing an example using concepts you have already been introduced to. When implementing a data structure, you have two choices for your **storage format**: 1) Array Based or 2) a List Based structure[2]. In other words I can use an **array** or I can use a **linked list** to implement my stack or queue. But what exactly makes an array or a linked list a **stack** or a **queue**? They both have push operations. They both have pop operations[3]. The difference is in the algorithm that you employ to add or retrieve that data. A stack uses **LIFO** (Last In First Out) while a queue uses **FIFO** (First In First Out). A small change in the algorithm you use to implement the data structure changes the entire behavior of that data structure regardless of the *storage format* that you chose. The algorithm that we use to add or remove items from each structure will guarantee, unambiguously, that the data structure will behave in a correct manner.

Let's look at the difference in algorithms between a stack and a queue. This example will use an array as its storage container. See the implementation of **Push** and **Pop** below:

[1] Describe what "ambiguous" behavior of an algorithm or data structure might be
[2] Or some hybrid of the two. But in general, those are the choices
[3] Some books use 'enqueue' and 'dequeue' as operations for a queue, but they still add and remove items.

Op	Stack	Queue

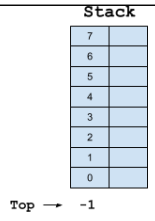
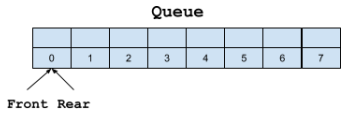
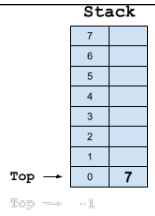
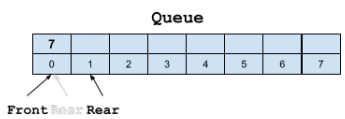
Push	`//`
	`bool Push(int x){`
	`  if(!full()){`
	`  ++top;`
	`  array[top] = x;`
	`  return true;`
	`  }`
	`  return false;`
	`}`
		`bool Push(int x){`
		`  if(!full()){`
		`  array[rear] = x;`
		`  rear = (rear + 1) % array_size;`
		`  return true;`
		`  }`
		`  return false;`
		`}`

Pop	`//`
	`int Pop(){`
	`  if(!empty()){`
	`  int retval = array[top];`
	`  --top;`
	`  return retval;`
	`  }`
	`  return INT_MIN;`
	`}`
		`int Pop(){`
		`  if(!empty()){`
		`  int retval = array[front];`
		`  front = (front + 1) % array_size;`
		`  return retval;`
		`  }`
		`  return INT_MIN;`
		`}`

Table 2: Algorithms for Stack and Queue

If you notice, the methods used to implement *Push* and *Pop* for both a *Stack* and a *Queue* are nearly identical, however, one small change to the algorithm used to control where items get inserted or removed, changes the behavior in a pretty big way. It changes from *FIFO* to *LIFO* and vice versa depending on whether you insert and remove from "Top" or if your insert and remove from "Rear" and "Front" respectively.

Look at the example below of inserting values into each data structure using their respective algorithms. We tend to draw stacks with a vertical orientation, and queues with a horizontal orientation because those orientations fit visually with how most individuals picture each respective structure. A "stack" is pictured like items are being "stacked" one on top of another, and a "queue" is pictures like items "getting in line" one behind the other.

OP	Stack	Queue
	 Initial State	 Initial State
Push 7		

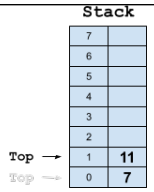
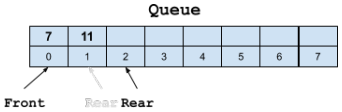
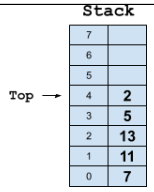
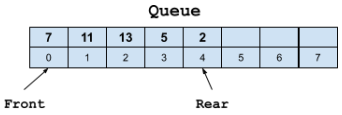
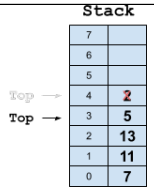
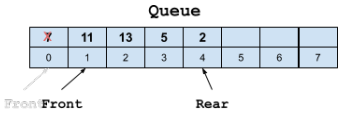
	1. Increment Top 2. Insert item at Top	1. Insert item at Rear 2. Increment Rear
Push 11	 1. Increment Top 2. Insert item at Top 1. Insert item at Rear 2. Increment Rear	
Push 13,5,2	 Resulting Stack After 13,5,2 are pushed Resulting Queue After 13,5,2 are pushed	

Table 3: Stack Queue Example Inserts

The resulting data structures are identical after inserting 5 values into each. The values are in the same order. They are using the same type of container (an array), and they have the same types of operations (Push, Pop). The main difference between the two is in how each algorithm implements the "removal" method. Stacks remove from the "top" and queues remove from the "front", entirely changing the order in which they operate. But is that true? Below we remove an item from each, and yes, we get a 2 from the stack, and a 7 from the queue. But does the subtle differences in each algorithm make these structures really that different?

Pop	 1. Copy value at Top (2) 2. Decrement Top 3. Return value 2	

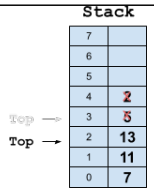
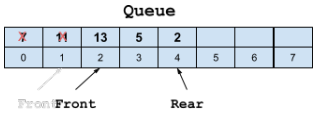
	1. Copy value at Front (7) 2. Increment Front 3. Return value 7	
Pop	 1. Copy value at Top (5) 2. Decrement Top 3. Return value 5 1. Copy value at Front (11) 2. Increment Front 3. Return value 11	

Table 4: Stack Queue Example Removals

Try mixing up the operations performed by doing pushes and pops randomly[4]. Our *containers would have been extremely different* in that scenario. In fact you should do this as an exercise to make sure you understand how both data structures work.

I hope that I have shown you that a "data structure" coupled with an "algorithm" become partners in creating "advanced structures". It is a little stretch to call a stack or a queue an "advanced structure", but as introductory examples, they work fine. They showed us that a "data structure" stores data and an "algorithm" unambiguously controlled how that data was stored. Now all we need to do is use more complex algorithms to control how data the data is stored and retrieved, and we will be solving cool complex problems[5].

[4] I didn't do this because I didn't want to have a 4 page example

[5] Also known as "Advanced Structures & Algorithms"

2 Pointers

In C++, pointers are variables that store the memory addresses of other variables.

2.1 Addresses in C++

If we have a variable **var** in our program, **&var** will give us its address in the memory.

For example,

```
#include <iostream>
using namespace std;

int main(){
    // declare variables
    int var1 = 3;
    int var2 = 24;
    int var3 = 17;

    // print address of var1
    cout << "Address of var1: "<< &var1 << endl;
    // print address of var2
    cout << "Address of var2: " << &var2 << endl;
    // print address of var3
    cout << "Address of var3: " << &var3 << endl;
}
```

Output

```
Address of var1: 0x7fff5fbff8ac
Address of var2: 0x7fff5fbff8a8
Address of var3: 0x7fff5fbff8a4
```

Here, **0x** at the beginning represents the address is in the hexadecimal form. Notice that the first address differs from the second by 4 bytes and the second address differs from the third by 4 bytes. This is because the size of an **int** variable is 4 bytes in a 64-bit system[6].

[6]You may not get the same results when you run the program.

2.2 C++ Pointers

As mentioned above, pointers are used to store addresses rather than values.
Here is how we can declare pointers.

```
int *pointVar;  // preferred syntax
```

Here, we have declared a pointer **pointVar** of the **int** type. We can also declare pointers in the following way.

```
int* pointVar;
```

Let's take another example of declaring pointers.

```
int* pointVar, p;
```

Here, we have declared a pointer **pointVar** and a normal variable **p** .

Note: The * operator is used after the data type to declare pointers.

2.2.1 Assigning Addresses to Pointers

Here is how we can assign addresses to pointers:

```
int* pointVar, var;
var = 5;

// assign address of var to pointVar pointer
pointVar = &var;
```

Here, **5** is assigned to the variable **var** . And, the address of **var** is assigned to the **pointVar** pointer with the code **pointVar = &var** .

2.2.2 Get the Value from the Address Using Pointers

To get the value pointed by a pointer, we use the * operator. For example:

```
int* pointVar, var;
var = 5;

// assign address of var to pointVar
pointVar = &var;

// access value pointed by pointVar
cout << *pointVar << endl;   // Output: 5
```

In the above code, the address of var is assigned to **pointVar** . We have used the ***pointVar** to get the value stored in that address.

When * is used with pointers, it's called the **dereference operator**. It operates on a pointer and gives the value pointed by the address stored in the pointer. That is, ***pointVar = var** .

Note: In C++, **pointVar** and ***pointVar** are completely different. We cannot do something like ***pointVar**

2.2.3 Example 2: Working of C++ Pointers

```cpp
#include <iostream>
using namespace std;
int main() {
    int var = 5;

    // declare pointer variable
    int* pointVar;

    // store address of var
    pointVar = &var;

    // print value of var
    cout << "var = " << var << endl;

    // print address of var
    cout << "Address of var (&var) = " << &var << endl
         << endl;

    // print pointer pointVar
    cout << "pointVar = " << pointVar << endl;

    // print the content of the address pointVar points to
    cout << "Content of the address pointed to by pointVar (*pointVar) = " << *pointVar << endl;

    return 0;
}
```

Output:

```
var = 5
Address of var (&var) = 0x61ff08

pointVar = 0x61ff08
Content of the address pointed to by pointVar (*pointVar) = 5
```

Figure 1: Working of C++ pointers

2.2.4 Changing Value Pointed by Pointers

If **pointVar** points to the address of **var**, we can change the value of **var** by using ***pointVar**.

For example,

```
int var = 5;
int* pointVar;

// assign address of var
pointVar = &var;

// change value at address pointVar
*pointVar = 1;

cout << var << endl; // Output: 1
```

Here, **pointVar** and **&var** have the same address, the value of **var** will also be changed when ***pointVar** is changed.

2.2.5 Example 3: Changing Value Pointed by Pointers

```
#include <iostream>
using namespace std;
int main() {
    int var = 5;
    int* pointVar;

    // store address of var
    pointVar = &var;
```

```
9
10      // print var
11      cout << "var = " << var << endl;
12
13      // print *pointVar
14      cout << "*pointVar = " << *pointVar << endl
15           << endl;
16
17      cout << "Changing value of var to 7:" << endl;
18
19      // change value of var to 7
20      var = 7;
21
22      // print var
23      cout << "var = " << var << endl;
24
25      // print *pointVar
26      cout << "*pointVar = " << *pointVar << endl
27           << endl;
28
29      cout << "Changing value of *pointVar to 16:" << endl;
30
31      // change value of var to 16
32      *pointVar = 16;
33
34      // print var
35      cout << "var = " << var << endl;
36
37      // print *pointVar
38      cout << "*pointVar = " << *pointVar << endl;
39      return 0;
40  }
```

Output

```
var = 5
*pointVar = 5

Changing value of var to 7:
var = 7
*pointVar = 7

Changing value of *pointVar to 16:
var = 16
*pointVar = 16
```

2.3 Common mistakes when working with pointers

Suppose, we want a pointer **varPoint** to point to the address of **var**. Then,

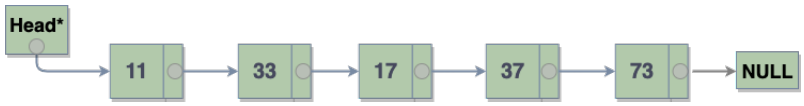
```
int var, *varPoint;

// Wrong!
// varPoint is an address but var is not
varPoint = var;

// Wrong!
// &var is an address
// *varPoint is the value stored in &var
*varPoint = &var;

// Correct!
// varPoint is an address and so is &var
varPoint = &var;

 // Correct!
// both *varPoint and var are values
*varPoint = var;
```

3 Linked Lists

Name: Linked List
Year Invented: 1955–1956
Invented by: Allen Newell, Cliff Shaw and Herbert A. Simon
Structure / Container: Linked Structure

Complexity

Algorithm	Average	Worst case	Best case
Space	O(n)	O(n)	O(n)
Un-Ordered Search	O(n)	O(n)	O(1)
Ordered Search	O(n/2)	O(n)	O(1)
Insert	O(n)	O(n)	O(1)



3.1 Linked Lists in General

A linked list as a linear collection of data elements whose order is not a contiguous set of memory locations like an array, and who's order is not determined by their actual (physical) location in memory. The order is determined by one element containing the address to the next element in the list. This means nodes in a linked list could be stored in an order that is nothing like we visualize a list as being. The above image is the typical representation of a linked list that abstracts out all of the "address" information. The following image is a similar representation, but showing actual addresses for each node (as well as the address for head).

At a minimum, each node contains some data element and a reference (pointer) to the next element in the list. A list can be ordered or unordered depending on needs of implementation. But, one inherent attribute of a linked list is the efficient insertion and deletion of nodes at any an all positions. The structure grows and shrinks with each addition or deletion. One drawback that lists have is the need to access them linearly. When compared to arrays, accessing elements directly (aka random access) is not possible. Another advantage of arrays is they have better cache locality[6].

A linked list whose nodes contain two fields: an integer value and a link to the next node. The last node is linked to a terminator used to signify the end of the list. Linked lists are among the simplest and most common data structures. They can be used to implement several other common abstract data types, including lists, stacks, queues, associative arrays, and S-expressions, though it is not uncommon to implement those data structures directly without using a linked list as the basis.

The principal benefit of a linked list over a conventional array is that the list elements can be easily inserted or removed without reallocation or reorganization of the entire structure because

the data items need not be stored contiguously in memory or on disk, while restructuring an array at run-time is a much more expensive operation. Linked lists allow insertion and removal of nodes at any point in the list, and allow doing so with a constant number of operations by keeping the link previous to the link being added or removed in memory during list traversal.

On the other hand, since simple linked lists by themselves do not allow random access to the data or any form of efficient indexing, many basic operations—such as obtaining the last node of the list, finding a node that contains a given datum, or locating the place where a new node should be inserted—may require iterating through most or all of the list elements. The advantages and disadvantages of using linked lists are given below. Linked list are dynamic, so the length of list can increase or decrease as necessary. Each node does not necessarily follow the previous one physically in the memory.

```
void PrintList(){
    if(!Head){
        cout<<"Empty"<<endl;
        return;
    }else{
        cout<<"The sizeof `Node` in bytes is: "<<sizeof(Head->Data)<<endl;
        cout<<"Address of `Head` is: "<<&Head<<endl;
        cout<<"`Head` points to ->"<<Head<<endl<<endl;

        Node *Temp = Head;
        while(Temp != NULL){
            cout<<"|Address: "<<Temp<<" |Data: "
                <<Temp->Data<<" |Next: "<<Temp->Next<<" | "
                <<" -> "<<endl;
            Temp = Temp->Next;
        }
        cout<<"NULL"<<endl<<endl;
    }
}
```

Output:

```
The sizeof `Node` in bytes is: 4
Address of `Head` is: 0x7ffeeeafd550
`Head` points to ->0x7faaf3c05920

|Address: 0x7faaf3c05920 |Data: 1 |Next: 0x7faaf3c05910 |  ->
|Address: 0x7faaf3c05910 |Data: 56 |Next: 0x7faaf3c05900 |  ->
|Address: 0x7faaf3c05900 |Data: 20 |Next: 0x7faaf3c058f0 |  ->
|Address: 0x7faaf3c058f0 |Data: 39 |Next: 0x7faaf3c058e0 |  ->
|Address: 0x7faaf3c058e0 |Data: 20 |Next: 0x7faaf3c058d0 |  ->
|Address: 0x7faaf3c058d0 |Data: 56 |Next: 0x0 |  -> NULL
```

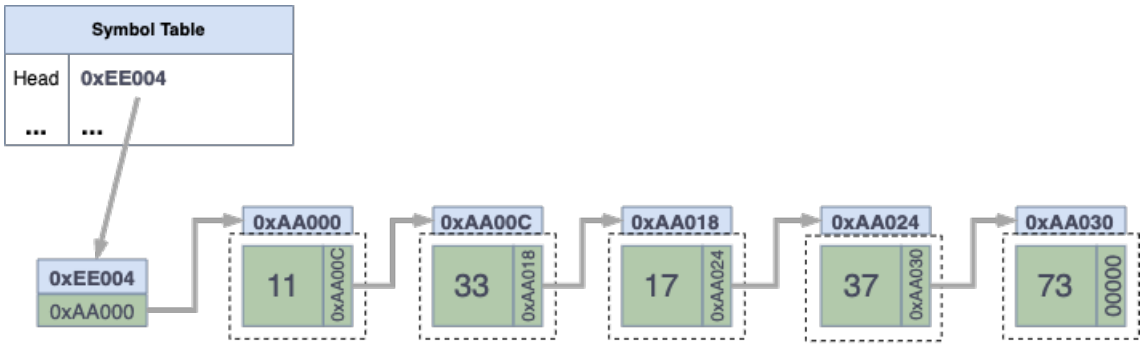


Figure 2: Detailed View of a Linked List

```
struct Node {
    Person Data;
    Node *Next;
    Node(Person p) {
        Data = p;
        Next = NULL;
    }
};

struct Person {
    string Name;
    int Age;
    Person() {
        Name = "";
        Age = 0;
    }
    Person(string n, int a) {
        Name = n;
        Age = a;
    }
};

void Print() {
    Node *Current = Head;
    cout << sizeof(Current->Data) << endl;
    while (Current) {
        cout << "|" << Current->Data.Name << "|" << Current << " ->\n";
        cout ;
        Current = Current->Next;
    }
    cout << "NULL" << endl;
}
```

Output:

```
|Abby|0x8d2200| ->
|Chuck|0x8d2240| ->
|Daphne|0x8d23c0| ->
|Donnie|0x8d2280| ->
|Echo|0x8d22c0| ->
|Gerald|0x8d2380| ->
|MarkyMark|0x8d2300| -> NULL
```

3.2 Singly Linked Lists

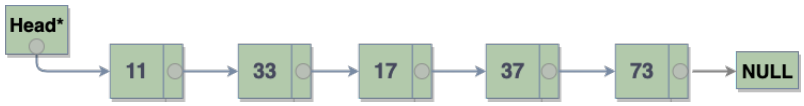


Figure 3: Singly Linked List

3.3 Circular Linked Lists

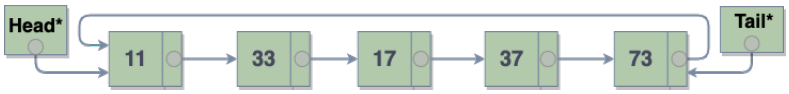


Figure 4: Circular Linked List

3.4 Doubly Linked Lists

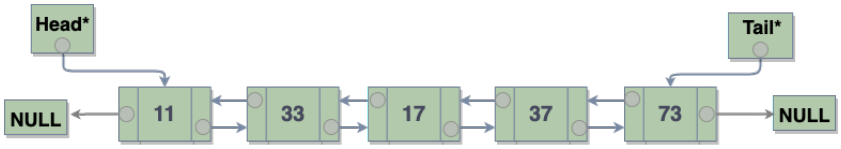


Figure 5: Doubly Linked List

3.5 Circular Doubly Linked Lists

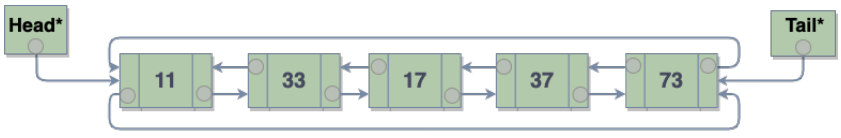
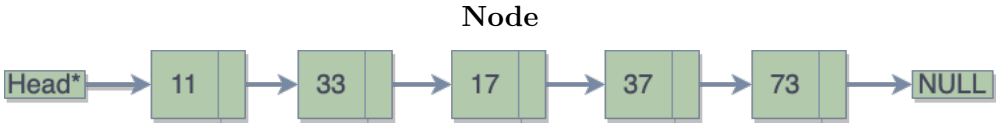


Figure 6: Circular Doubly Linked List

3.6 Linked List Refresher

A linked list is a sequence of data structures, which are connected together via links (pointers).

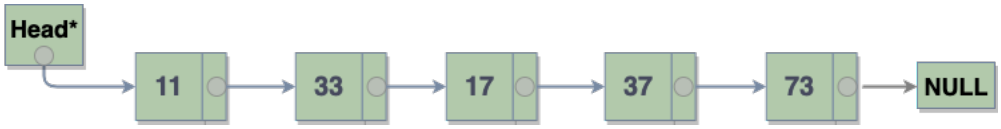


Linked List is a sequence of nodes which contains items. Each node contains a connection to another node. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Node**: Each node of a linked list stores a data element. Something simple like an **int** or complex like a class or **struct**.
- **Next**: Each node of a linked list contains a **pointer** to the next node. Typically called **Next**.
- **Head**: Every linked list needs a way to start accessing it. We typically call this pointer **Head** or **Front**.

3.6.1 Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



Looking at the image we can see:

- A *Head* pointer that gives us access to the list. Lose *Head*, and lose the list!
- Each Node has data, and a pointer to the next node in the list..
- The last Node's "Next" pointer, points to *NULL*. This tells us its the end of the list.

3.6.2 Types of Linked List

Following are the various types of linked list.

- **Singly Linked List** Only traverse this list one way.
- **Doubly Linked List** Can traverse this list forwards and backwards.
- **Circular Linked List** Last item contains link of the first element as next and the first element has a link to the last element as previous.

3.6.3 Basic Operations

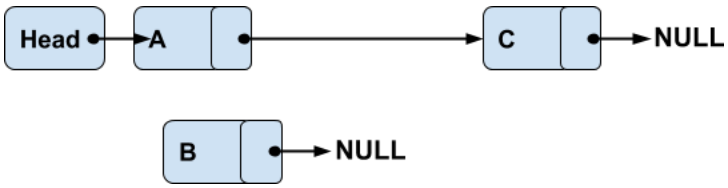
Following are the basic operations supported by a list.

- **Insert** Adds an element at the beginning or end of the list, depending on your implementation.
- **Search** Searches an element using the given key.
- **Delete** Deletes an element from the list. If its not the first or last Node, then we accompany this with a search to find the proper node to delete using the given key.
- **Print** Prints the list. Mostly for us programmers to debug our linked list code and make sure its working as expected.

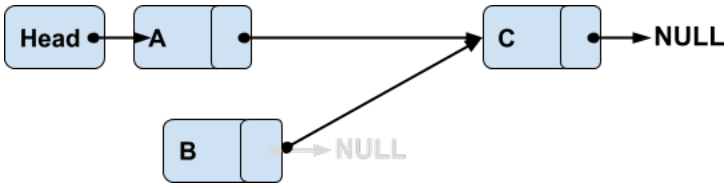
3.6.4 Insertion Operation

Adding a new node to a list requires the creation of a new node, and the manipulation of a couple of pointers. Lets add a node to the following list:

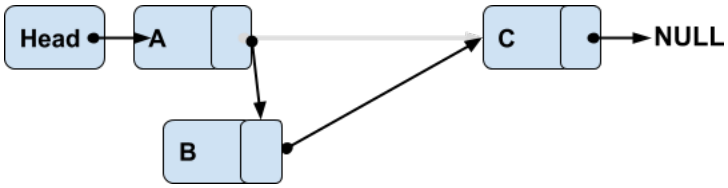
We start by creating a new node "**B**" which we want to add between **A** and **C** (for visual effect I moved **C** further away):



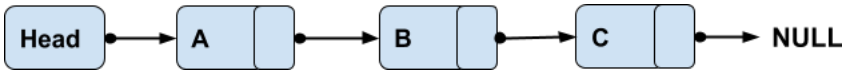
Our first step is to "link" the new node into the list, by pointing its **next** pointer to **C**.



Then we complete the insertion by pointing **A** to **B**



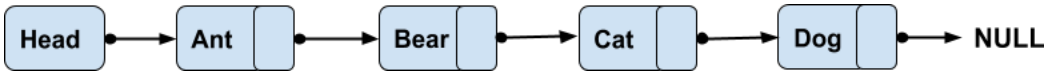
Your resulting linked list:



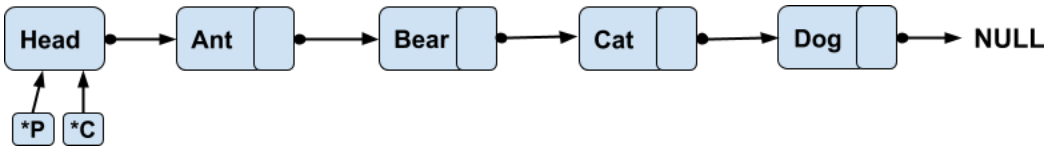
Its similar if you insert a node at the beginning or end of a list. This is just an overview, and I won't be posting any code specific examples. https://repl.it/@rugbyprof/intlinkedlistmain.cpp is a linked list implementation on Repl.it.

3.6.5 Deletion Operation

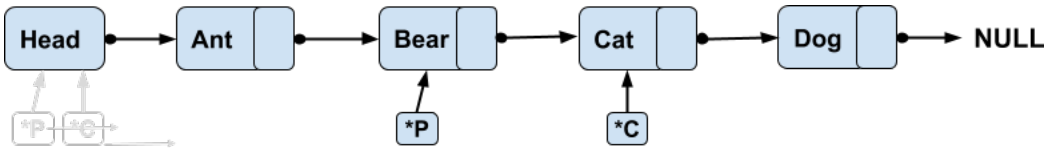
Given the list below, lets say we want to delete the node with the value "Cat" in it.



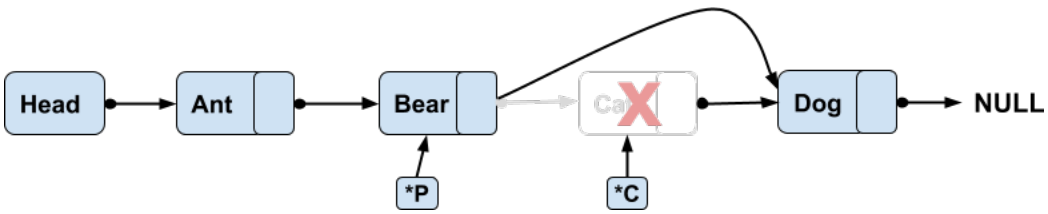
Deletion is also a more than one step process. You first need to traverse the list to find the item you want to delete, and this starts by placing two pointers at the head of our list. We will be using "P" for previous, and "C" for current.



We then traverse the list searching to find the "key" we want to delete. We use the "C" pointer to look at the data in each node, and we use the "P" pointer to trail behind staying one node behind the current pointer.



Why do we need two pointers? Look at the image above. Does the "C" pointer have the ability to make the previous node *bypass* the node it is pointing to? NO! That's why we use a "P" pointer. It gives us the ability to jump over "current" and thereby removing the node from our list.



Resulting list after "Cat" is deleted.

4 Recursion

In computer science, recursion is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem. Such problems can generally be solved by iteration, but this needs to identify and index the smaller instances at programming time. Recursion solves such recursive problems by using functions that call themselves from within their own code. The approach can be applied to many types of problems, and recursion is one of the central ideas of computer science.[8]

Recursion is a very frustrating concept that only comes naturally to a lucky few. So if you are one of those ... good for you. If your not, remember that most people do not "figure out" or "see" the solutions to many of the examples we will look at. Those come with time and practice. We also need to remember that as cool as recursion can be, it is not efficient at all. In fact, if a compiler can turn your recursive function into an iterative one ... it will!

4.1 Components of a Recursive Function

There are three basic components that define a recursive function. You must have all three to make a recursive function actually work. Those components are:

1. A function that calls itself.
2. A Base Case.
3. Subsequent calls that approach the base case.

Recursive functions can be thought of as a weird type of loop, because they continuously call themselves until some boolean statement informs them to stop. When we place code in typical loops (for, while, do), we call that iteration. And if we solve a problem using loops and not recursion, we call that an **iterative** solution. The sad truth is, iteration is more efficient than recursion, and in fact many recursive functions get turned into iteration by the compiler! Why do we use recursion then? It just fits certain problems so well. I won't discuss that yet, just trust that there is a place for recursion. Look at the recursive example below. It has all the necessary components listed above.

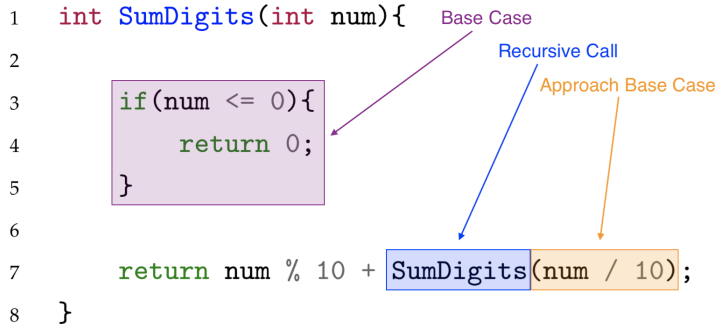


Figure 7: Recursive Function Components

What is this recursive function doing? It *sums* the digits. A call to SumDigits(123) would return 6 as an answer. Figure 7 shows each of the components necessary for recursion labeled.

Recursive Solution:

```
int SumDigits(int num){
    if(num<=0){
        return 0;
    }
    return num % 10 +
           SumDigits(num / 10);
}
```

```
int TreeHeight(Node* Root){

    // Base Case
    if(!Root){
        return NULL;
    }

    int lh = TreeHeight(Root->left);    // recursive call 1
    int rh = TreeHeight(Root->right);   // recursive call 2


    if(lh > rh)
        return 1 + lh;
    else
        return 1 + rh;
}
```

Iterative Solution:

```
int SumDigits(int num){
    int sum = 0;
    while(num>0){
        sum += num % 10;
        num /= 10;
    }
    return sum;
}
```

This function gets the least significant digit of "num" using modulo 10. It then adds that value to another call to SumDigits with the number divided by 10 (approaching the base case). Since "num" is an integer, it loses the decimal, basically stripping off the last digit with every call. Eventually this will reach zero, and return zero, terminating the series of recursive calls and propagating each intermediate answer back and summing it with the previous call. Eventually, we have all the digits summed up as an answer.

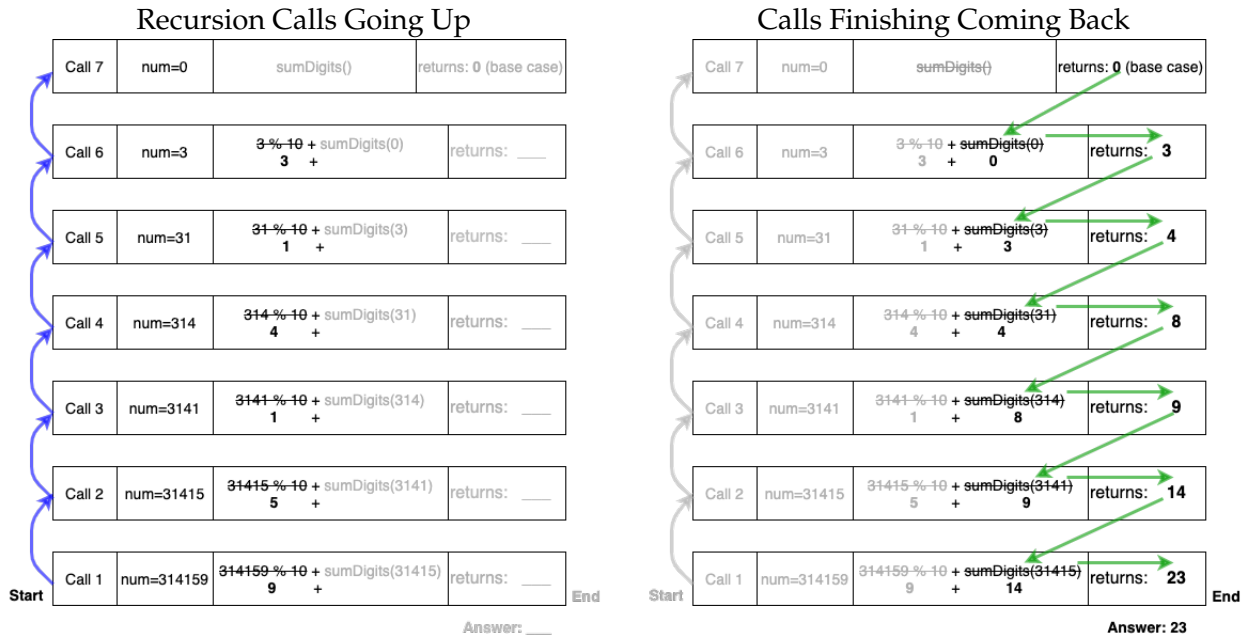


Table 5: Recursive Trace of SumDigits Function

4.2 Steps to Understanding Recursion

4.2.1 Step 1: Define the Prototype

Write and define the prototype of the function

Since functions are the basic unit of recursion, it's important to know what the function does. The prototype you use will dictate how the recursion behaves. Here's a function which will sum the first n elements of an array. It is obviously not implemented, but we know what parameters we are passing in.

```
// arr : int array
// n   : first n values to sum
int sum( int arr[], int n );
```

4.2.2 Step 2: Sample Call

Write out a sample function call

Once you've determined what the function does, then we imagine a function call.

```
int result = sum( arr, n );
```

So, the call is `sum( arr, n )`. This will sum the first **n** elements of **arr**. You want to pick the most generic function call. For example, you don't want to have a call like:

```
int result = sum( arr, 10 );
```

Why? Never use constants!! You want to use variables when possible, because that's the most general and robust way to use a function.

4.2.3 Step 3: Smallest Version

Think of the smallest version of the problem

The smallest version is called the *base case*. Most people mistakenly pick a base case that's too large. In this case, you will pick a specific value for n.

So, what's the smallest version of the problem? Here are three choices:

```
sum( arr, 2 );  // Choice 1
sum( arr, 1 );  // Choice 2
sum( arr, 0 );  // Choice 3
```

Some people pick choice 1, reasoning that if you are to sum elements of an array, then you must have at least two elements to sum. However, that's really not necessary. In math, there is something called a "summation". It's perfectly valid to have a summation of only one element. You just return that one element.

Some people pick choice 2, because it doesn't make sense to sum an array of size 0, whereas an array of size 1 seems to make sense.

However, it turns out choice 3 is the smallest choice possible. You can sum zero elements of an array. What value should it return? It should return 0. As it turns out, 0 is the additive identity. Anything added to 0 is that number. If we wanted to multiply all elements of an array, we would have picked the multiplicative identity, which is 1.

Admittedly, picking such a small value requires a more extensive knowledge of math, but that shouldn't scare you from picking the smallest value possible.

There are several mistakes people make with a base case. The first one we've already mentioned: picking too large a base case. Second, not realizing there may be more than one base case. Finally, thinking that the base case only gets called when the input size is the smallest. In fact, the recursion ALWAYS makes it to some base case. Thus, the base case is where the recursion eventually stops. Don't think of it as merely called when the input is, say, 0. It gets called for all cases (eventually).

4.2.4 Step 4: Smaller Version (again)

Think of smaller versions of the function call

Here's the function call

```
sum( arr, n )  // sums first n elements of arr
```

It tries to solves a problem of size "n". We want to think of a smaller problem which we will assume can be solved correctly. The next smallest problem is to sum "n - 1" elements.

```
sum( arr, n - 1 )  // sums first n - 1 elements of arr
```

Assume this problem has been solved for you. How would you solve the original, larger problem? If the first n - 1 elements have already been summed then only the n^{th} element is left to be summed. The n^{th} element is actually at index $n-1$ (because arrays start at index 0).

So, the solution to solving `sum( arr, n )` is to add `sum( arr, n - 1 )` to `arr[ n - 1 ]`.

```
int sum( int arr[], int size ) {
    if ( size == 0 )  // base case
        return 0;
    else{
        // recursive call
        int smallResult = sum( arr, size - 1 );

        // use solution of recursive call to solve this problem
        return smallResult + arr[ size - 1 ];
    }
}
```

Some people don't like multiple return statements. That can be easily handled:

```
int sum( int arr[], int size ) {
    int result;
    if ( size == 0 )  // base case
        result = 0;
    else{
        // recursive call
        int smallResult = sum( arr, size - 1 );

        // use solution of recursive call to solve this problem
        result = smallResult + arr[ size - 1 ];
    }
    return result;
}
```

You may even think there's no reason to declare smaller result and prefer to write:

```
int sum( int arr[], int size ) {
    if ( size == 0 ){  // base case
```

```
        return 0;
    }else{
        return sum( arr, size - 1 ) + arr[ size - 1 ];
    }
}
```

4.3 Recursion Examples

1. Write the reverse() function recursively. This function takes a string and the length of the string as arguments and returns the same string with its characters in the reverse order.

```
void reverse(char *s, int len)
{
        char temp;
        if (len > 1) {
                temp = s[0];
                s[0] = s[len-1];
                s[len-1] = temp;
                reverse(s+1,len-2);
        }
}
```

2. A palindrome is a sequence of characters or numbers that looks the same forwards and backwards. For example, "Madam, I'm Adam" is a palindrome because it is spelled the same reading it from front to back as from back to front. The number 12321 is a numerical palindrome. Write a function that takes a string and its length as arguments and recursively determines whether the string is a palindrome:

```
int ispalindrome(char *s, int len)
{
        if (len <=1 )
                return 1;
        else
                return((s[0] == s[len-1]) && ispalindrome(s+1, len-2));
}
```

3. Write a recursive function void replace(char *s, char from, char to); that changes all occurrences of the **from** character into the **to** character. For example, if **s** were "steve", and $from = e$ and $to = a$, **s** would become "stava".

```
void replace(char *s, char from, char to)
{
        if (*s != '\0') {
                if (*s == from) *s = to;
                replace(s+1, from, to);
        }
}
```

4. Write a function to recursively print out an integer in any base from **base 2** to **base 9**.

```
void print_base(int num, int base){
        if (num / base)
                print_base(num / base, base);
        putchar(num % base + '0');
}
```

5. Write a recursive function int count_digit(int n, int digit); to count the number of digits in a number n (n > 0) that are equal to a specified digit. For example, if the digit we're searching for is **2** and the number we're searching is **220**, the answer would be **2**.

```
int count_digit(int n, int digit)
{
        if (n == 0) return 0;

        if (n % 10 == digit)
            return 1 + count_digit(n / 10, digit);
        else
            return count_digit(n / 10, digit);
}
```

6. For some reason, the computer you're working on doesn't allow you to use the modulo operator

```
int remainder(int num, int den){
    if (num < den)
        return num;
    else
        return(remainder(num - den, den));
}
```

Is there a better way?

There is a much more efficient method to compute the remainder by taking advantage of integer division:

```
int remainder(int num, int den) {
    return num - (den * (num / den));
}
```

7. The following function iteratively computes exponentiation x^n :

```
int exponentiate_i(int x, int n){
    int sum=1;
    while(n>0){
        sum *= x;
        n--;
    }
    return sum;
}
```

Write a function to do this recursively in $O(n)$ time.

```
int exponentiate_r(int x, int n){
    if (n==0)
        return 1;
    else
        return x * exponentiate_r(x, n-1);
}
```

8. Use the knowledge that $x^n = (x^2)^{(n/2)}$ when n is even to write a more efficient solution to the above problem. Not totally necessary, but pretty cool.

- If n is even, then $x^n = (x^2)^{(n/2)}$
- If n is odd, then $x^n = x * (x^2)^{(n-1)/2}$

So:

```
int exponentiate2_r(int x, int n){
    if (n==0)
        return 1;
    else if (n % 2 == 0)
        return exponentiate2_r(x*x, n/2);
    else
        return x * exponentiate2_r(x*x, (n-1) / 2);
}
```

9. The classic fibonacci problem is where the next term in the sequence is the sum of the previous two terms. For example we want the series: **1 1 2 3 5 8 13 ...**

```
int fib(int x){
    if (x == 0){
        return 0;
    }

    if (x == 1){
        return 1;
    }

    return fib(x-1)+fib(x-2);
}
```

10. Can anyone fix the function below to make it work like the following:

- When $p == 0, return\ n + m$
- When $p == 1, return\ n * m$
- When $p == 2, return\ n^m$

```
// Needs fixing...
int mystery(n, m, p){
        int i, result = 0;
```

```
        if (p==0){
            return n+m;
        }

        for (i=0; i< m; i++){
            result += mystery(result,n,p-1);
        }

        return result;
}
```

5 Introduction to Data Structures and Algorithms

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have some data which has, player's **name** "Virat" and **age** 26. Here "Virat" is of **String** data type and 26 is of **integer** data type.
We can organize this data as a record like **Player** record, which will have both player's name and age in it. Now we can collect and store player's records in a file or database as a data structure. **For example**: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33.
If you are aware of Object Oriented programming concepts, then a `class` also does the same thing, it collects different type of data under one single entity. The only difference being, data structures provides for techniques to access and manipulate data efficiently.
In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily. It represents the knowledge of data to be organized in memory. It should be designed and implemented in such a way that it reduces the complexity and increases the efficiency.

5.1 Basic types of Data Structures

As we have discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of **Abstract Data Structure** are :

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.

The data structures can also be classified on the basis of the following characteristics:

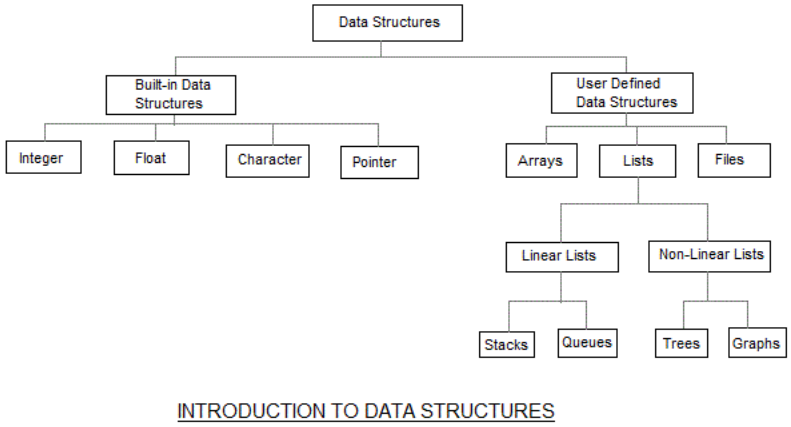


Figure 8: Caption

Characteristic	**Description**
Linear	In Linear data structures,the data items are arranged in a linear sequence. Example: **Array**
Non-Linear	In Non-Linear data structures,the data items are not in sequence. Example: **Tree**, **Graph**
Homogeneous	In homogeneous data structures,all the elements are of same type. Example: **Array**
Non-Homogeneous	In Non-Homogeneous data structure, the elements may or may not be of the same type. Example: **Structures**
Static	Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example:**Array**
Dynamic	Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations changes. Example: **Linked List created using pointers**

5.2 What is an Algorithm ?

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as **pseudocode** or using a **flowchart**.

Every Algorithm must satisfy the following properties:

1. **Input**- There should be 0 or more inputs supplied externally to the algorithm.
2. **Output**- There should be atleast 1 output obtained.
3. **Definiteness**- Every step of the algorithm should be clear and well defined.
4. **Finiteness**- The algorithm should have finite number of steps.
5. **Correctness**- Every step of the algorithm must generate a correct output.

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties :

1. Time Complexity
2. Space Complexity

5.3 Space Complexity

Its the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

An algorithm generally requires space for following components :

- **Instruction Space:** Its the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
- **Data Space:** Its the space required to store all the constants and variables(including temporary variables) value.
- **Environment Space:** Its the space required to store the environment information needed to resume the suspended function.

5.4 Time Complexity

Time Complexity is a way to represent the amount of time required by the program to run till its completion. It's generally a good practice to try to keep the time required minimum, so that our algorithm completes it's execution in the minimum time possible.

6 Space Complexity of Algorithms

Whenever a solution to a problem is written some memory is required to complete. For any algorithm memory may be used for the following:

1. Variables (This include the constant values, temporary values)
2. Program Instruction
3. Execution

> ***Space complexity*** *is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.*

Sometime **Auxiliary Space** is confused with Space Complexity. But Auxiliary Space is the extra space or the temporary space used by the algorithm during it's execution.
Space Complexity = **Auxiliary Space + Input space**

6.1 Memory Usage while Execution

While executing, algorithm uses memory space for three reasons:

1. **Instruction Space** It's the amount of memory used to save the compiled version of instructions.
2. **Environmental Stack** Sometimes an algorithm(function) may be called inside another algorithm(function). In such a situation, the current variables are pushed onto the system stack, where they wait for further execution and then the call to the inside algorithm(function) is made.

   For example, If a function `A()` calls function `B()` inside it, then all th variables of the function `A()` will get stored on the system stack temporarily, while the function `B()` is called and executed inside the funciton `A()`.
3. **Data Space**

   Amount of space used by the variables and constants.

But while calculating the **Space Complexity** of any algorithm, we usually consider only **Data Space** and we neglect the **Instruction Space** and **Environmental Stack**.

6.2 Calculating the Space Complexity

For calculating the space complexity, we need to know the value of memory used by different type of datatype variables, which generally varies for different operating systems, but the method for calculating the space complexity remains the same.

Type	Size
`bool, char, unsigned char, signed char,__int8`	`1 byte`
`__int16, short, unsigned short, wchar_t, __wchar_t`	`2 byte`
`float, __int32, int, unsigned int, long, unsigned long`	`4 byte`
`double, __int64, long double, long long`	`8 byte`

Now let's learn how to compute space complexity by taking a few examples:
In the above expression, variables `a`, `b`, `c` and `z` are all integer types, hence they will take up 4 bytes each, so total memory requirement will be `(4(4) + 4) = 20 bytes`, this additional 4 bytes is for **return value**. And because this space requirement is fixed for the above example, hence it is called **Constant Space Complexity**.
Let's have another example, this time a bit complex one,

- In the above code, `4*n` bytes of space is required for the array `a[]` elements.
- 4 bytes each for `x`, `n`, `i` and the return value.

Hence the total memory requirement will be `(4n + 12)`, which is increasing linearly with the increase in the input value `n`, hence it is called as **Linear Space Complexity.**
Similarly, we can have quadratic and other complex space complexity as well, as the complexity of an algorithm increases.
But we should always focus on writing algorithm code in such a way that we keep the space complexity **minimum**.

7 Big-O Notation

Asymptotic notation is a way to express the performance of an algorithm in relation to the size of the input to that algorithm. Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big-O specifically describes the worst-case scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.

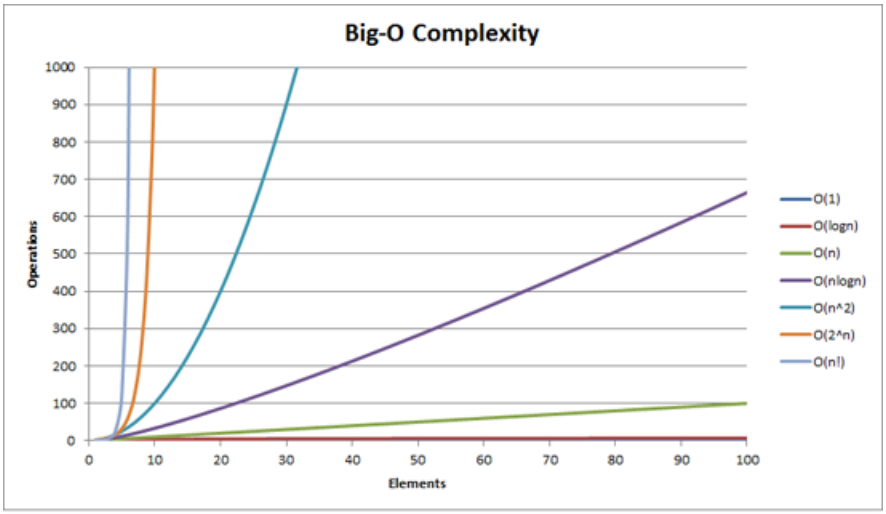


Figure 9: Big-O complexity can be visualized with this graph

As Vineet Choudhary said in his original article: "*the best way to understand Big O thoroughly is by using examples with code*". So, that's what we will do.

7.1 Example Complexity Times

7.1.1 Constant Time

Constant time code is code that doesn't depend in any input:

```
void printArrayElement(int arr[],int i){
    cout<<arr[i]<<endl;
}
```

This function runs in $O(1)$ time (or "constant time") relative to its input. The input array could be 1 item or 1,000 items, but this function would still just require one step.

7.1.2 Linear Time

Most programs (well when dealing with data) depend on the size of the data being processed. Linear time is the most straight forward. For every item in the data set (n), we will run at least 1 instruction. This is called $O(n)$ (linear time).

```
void printArrayElements(int arr[], int size){
    for (int i = 0; i < size; i++){
        cout<<arr[i]<<endl;
    }
}
```

In other words, if the array has 10 items, we have to print 10 times (10 instructions). If it has 1000 items, we have to print 1000 times (1000 instructions).

7.1.3 Logarithmic Time

Logarithmic time is faster than linear time. Where in linear time we ran at last one instruction per items in n, in logarithmic time we run way less! Without getting into how a binary search works, I can tell you that searching data where $n = 1024$ we would only need to run 10 instructions! If $n = 1048576$ we only need to run 20 instructions! We call logarithmic time $O(logn)$.

```
int BinarySearch(int *Data, int size, int key){
  int left = 0;
  int right = size-1;
  int middle = (left + right) / 2;
  bool found = false;

  while(1){
    if(Data[middle] == key){
      return middle;
    }else if(middle == left || middle == right){
      return -1;
    }else{
      if(key < Data[middle]){
        right = middle;
      }else{
        left = middle;
      }
      middle = (left + right) / 2;
    }
  }
  return -1; // not found
}
```

Binary search runs in logarithmic time because it eliminates half the search space with every pass. It is very efficient.

7.1.4 Exponential Time

$O(n^2)$

For every 1 item in n, we're processing another n items. $O(n^2)$ typically implies nested loops like in **bubble_sort**. In *bubble_sort* we are comparing every item in the data set, to every other item. This means we are looping $n * n$ times or $O(n^2)$.

```
void bubbleSort(int A[],int size){
  int temp=0;
  bool swapped = false;
  for(int i=0;i<size;i++){
    swapped = false;
    for(int j=0;j<size-(i+1);j++){
      if(A[j] > A[j+1]){
        swapped = true;
        temp = A[j];
        A[j] = A[j+1];
        A[j+1] = temp;
      }
    }
    if(!swapped){
      return;
    }
  }
}
```

$O(2^n)$

An example of an $O(2^n)$ function is the recursive calculation of Fibonacci numbers. $O(2^n)$ denotes an algorithm whose growth doubles with each addition to the input data set, and they are often recursive algorithms that solve a problem of size N by recursively solving two smaller problems of size N-1, just like below.

```
int fibonacci(int num){
    if (num <= 1){
        return num;
    }
    return fibonacci(num - 2) + fibonacci(num - 1);
}
```

If its not obvious, we try to avoid something that runs in $O(2^n)$. Another example of a $O(2^n)$ algorithm is Towers of Hanoi.

7.1.5 Factorial Time

One algorithm that runs in $O(n!)$ is one to generate all permutations of an array of letters (or a string). A permutation is a mathematical technique that determines the number of possible arrangements in a set when the order of the arrangements matters.

```
void permute(string a, int l, int r) {
    // Base case
```

```
    if (l == r){
        cout<<a<<endl;
    }else{
        // Permutations made
        for (int i = l; i <= r; i++){
            swap(a[l], a[i]);
            permute(a, l+1, r);
            swap(a[l], a[i]);
        }
    }
}
```

Another example of something that runs in factorial time is the naive approach to solving the Traveling Salesman Problem.

7.2 Constants Get Booted

When you're calculating the big O complexity of something, you just boot the constants.

```
void printAllItemsTwice(int arr[], int size){
    for (int i = 0; i < size; i++){
        cout<<arr[i]<<endl;
    }

    for (int i = 0; i < size; i++){
        cout<<arr[i]<<endl;
    }
}
```

This is $O(2 * n)$ or $O(2n)$ which we just call $O(n)$ cause the 2 is constant and doesn't effect the run time significantly.

```
void printFirstItemThenFirstHalfThenSayHi100Times(int arr[], int size){
    cout<<"First element of array = "<<arr[0]<<endl;

    for (int i = 0; i < size/2; i++){
        cout<<arr[i]<<endl;
    }

    for (int i = 0; i < 100; i++){
        cout<<"Hi"<<endl;
    }
}
```

This is $O(1 + \frac{n}{2} + 100)$, which we just call $O(n)$.

Why can we get away with this? Remember, for big O notation we're looking at what happens as $O(n)$ gets arbitrarily large. As n gets really big, adding 100 or dividing by 2 has a decreasingly significant effect (or increasingly insignificant effect).

7.3 Drop the less significant terms

```
void printAllNumbersThenAllPairSums(int arr[], int size){
    for (int i = 0; i < size; i++){
        cout<<arr[i]<<endl;
    }

    for (int i = 0; i < size; i++){
        for (int j = 0; j < size; j++){
            cout<<arr[i]<<" + "<<arr[j]<<endl;
        }
    }
}
```

Here our runtime is $O(n + n^2)$, which we just call $O(n^2)$.

Similarly:

- $O(n^3 + 50n^2 + 10000) is O(n^3)$
- $O((n + 30) * (n + 5)) is O(n^2)$

Again, we can get away with this because the less significant terms quickly become, well, less significant as n gets big.

7.4 Big-O: Whats the Worst?"

```
bool arrayContainsElement(int arr[], int size, int element){
    for (int i = 0; i < size; i++){
        if (arr[i] == element){
            return true;
        }
    }
    return false;
}
```

Here we might have 100 items in our array, but the first item might be the that element, in this case we would return in just 1 iteration of our loop.

In general we'd say this is $O(n)$ runtime and the "worst case" part would be implied. But to be more specific we could say this is worst case $O(n)$ and best case $O(1)$ runtime. For some algorithms we can also make rigorous statements about the "average case" runtime.

7.5 Another View

Let's look at analyzing some code and not depend on asymptotic notation this time. Lets run a function that loops from $0 \rightarrow 10000$ and prints each value of i:

```
#include <stdio.h>
void print_values(int end) {
    for (int i = 0; i < end; i++){
        cout<<i<<endl;
    }
}
int main(){
    print_values(10000);
    return 0;
}
```

Now we add a timer (clock time) so we can know how many milliseconds (thousandth of a second) this function takes to run.

```
#include <stdio.h>
#include <time.h>
void print_values(int end) {
    for (int i = 0; i < end; i++){
        cout<<i<<"\n";
    }
}
int main(){
    clock_t t;
    t = clock();

    print_values(10000);

    float diff = ((float)(clock() - t)) / CLOCKS_PER_SEC ;
    cout<<"\n\n diff="<<diff<<"\n\n";

    return 0;
}
```

Then we know how much it costs ... right?? It costs us **X** milliseconds!

Run it 3 or 4 more times. Odds are that your run times will be different. Now change machines, and run it again. It will definitely be different if you use a faster (or slower) CPU. Actually there are tons of reason your program will run differently: cache size, number of processes running on the computer at one time, CPU speed, etc. This is why asymptotic notation is important. It provide us with a mathematical foundation for representing the running time of our algorithms consistently, without worrying about specs of a system.

We create this consistency by assigning costs to each operation that our code has to perform. We give the most weight (highest cost) to those operations that depend on the size of our input (that thing we always call n).

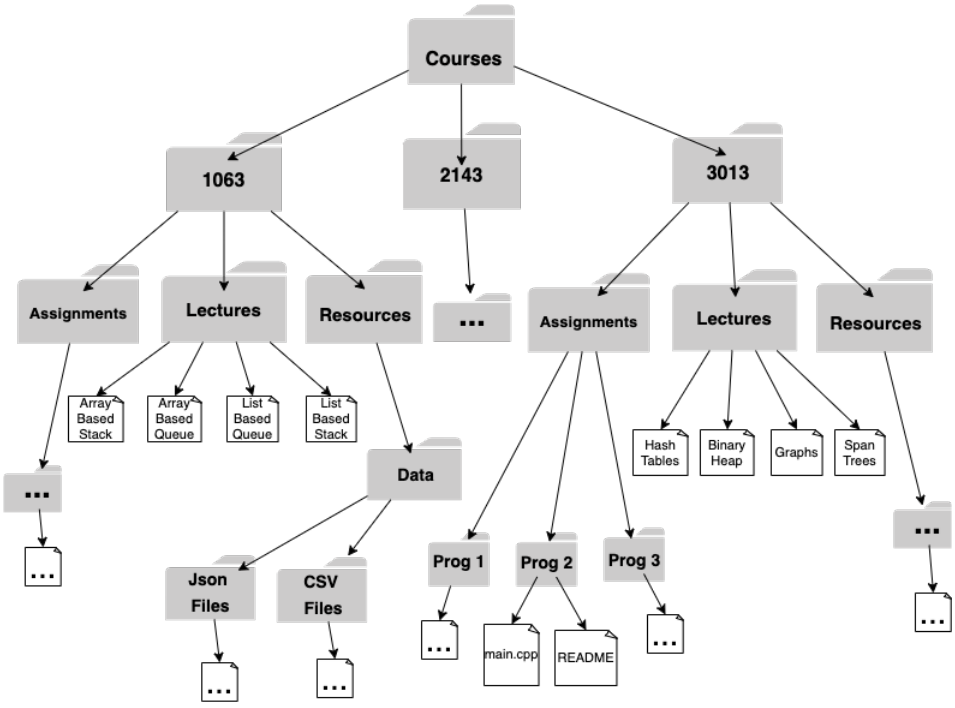
8 Trees

A graph is an ordered pair ***G = (V, E)*** comprising a set ***V*** of vertices, (aka: nodes or points) together with a set ***E*** of edges, (aka: arcs or lines).

A tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, any acyclic connected graph is a tree.

Below is a directory tree. This is a tree that represents a portion of a file system on pretty much any operating system. It's a generic representation of a folder hierarchy, displaying subfolders, and files in folders. Notice that there is no restriction on the number of children. Nodes have from 1-4 children, and could have many more depending on how many items a folder were to contain. All trees represent relationships between items, this tree happens to represent folders and files. In academia we tend to lean toward very defined tree's (Binary Search Tree, b-tree[3], r-tree[9], m-ary tree[13], trie[10], and many more.) but these tree are nothing without the algorithms that define their properties and behaviors. This section is just to label components of generic trees. We will get into specifically defined trees later.

8.1 Tree Vocab



Node A node is a fundamental part of a tree. It should contain information. Occasionally all or part of the nodes data is called a “key.” The key is a unique value we use to perform comparisons when finding the proper location to place the node in the tree. The node is not limited to storing only a key, in fact in the real world, it will store much more information in addition to the "key". Pretty much only in the classroom and books do we find trees that store only "keys" (like an integer).

Edge An edge is another fundamental part of a tree. An edge connects two nodes to show that there is a relationship between them. Every node (except the root) is connected by exactly one incoming edge from another node. Each node may have several outgoing edges. For example in a binary tree, outgoing edges are limited to two edges, whereas in m-ary tree trees, the edges are limited to M where M is an integer $0 < m <= M$.

Root The root of the tree is the only node in the tree that has no incoming edges. It is the first node we look at in a binary tree. `Courses` is the root in the file system tree.

Path A path is an ordered list of nodes that are connected by edges. For example, in the tree above: `Courses` $\rightarrow$ `3013` $\rightarrow$ `Assignments` $\rightarrow$ `Prog 2` $\rightarrow$ `main.cpp` is a path.

Parent A node is the parent `P`, of all the nodes it connects to with outgoing edges. In the tree above the nodes `1063,2143,3013` are each parents to sub-folders `Assignments, Lectures, and Resources` (2143’s children were collapsed for viewing).

Children The set of nodes `C` that have incoming edges from the same node `P`, are said to be the children of that node. In the directory tree, `1063, 2143, 3013` are children of `Courses` (which also happens to be the root).

Sibling Nodes in the tree that are children of the same parent are said to be siblings. `1063, 2143, 3013` are siblings, as well as `Prog 1, Prog 2, Prog 3` are siblings (under `3013` $\rightarrow$ `Assignments`).

Subtree A subtree is a set of nodes and edges comprised of a parent and all the descendants of that parent.

Leaf Node A leaf node is a node that has no children. Look at the directory tree and notice all the "documents" are leaves. This not always the case (a folder could be empty and have no children).

Level The level of a node `N` is the number of edges on the path from the root node to `N`. The level of Assignments (all of them) is level 2. The level of the README in Prog 2 is level 4.

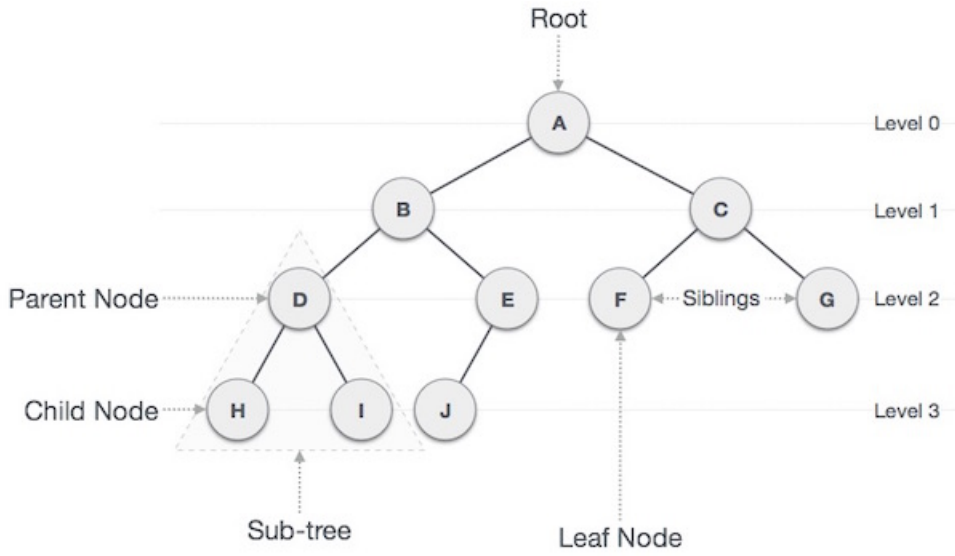
Height The height of a tree is equal to the maximum level of any node in the tree. What is the height of this tree? It’s not 4.

9 Binary Search Trees

Name: Binary Tree
Invented: 1960
Invented by: P.F. Windley, A.D. Booth, A.J.T. Colin, and T.N. Hibbard
Structure / Container: Linked Structure

Complexity

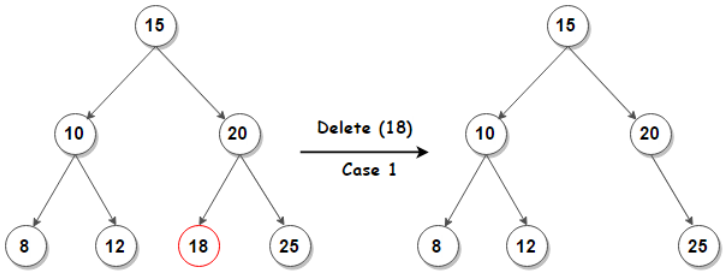
Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)



9.1 Delete

9.1.1 Case 1: Zero Children

Deleting a node with no children: remove the node from the tree.



9.1.2 Case 2: Two Children

Find the node to be deleted **N**. Do not delete **N**. Instead, choose either its in order successor node or its in order predecessor node, ‘R‘. Copy the value of ‘R‘ to ‘N‘, then recursively call delete on ‘R‘ until reaching one of the first two cases. If we choose the inorder successor of a node, as the right subtree is not NULL (our present case is a node with 2 children), then its inorder successor is a node with the least value in its right subtree, which will have at a maximum of 1 subtree, so deleting it would fall in one of the first 2 cases.
[inorder](https://www.techiedelight.com/inorder-tree-traversal-iterative-recursive/)

10 Traversals

10.1 Tree Traversals

10.1.1 Pre-Order

10.1.2 In-Order

10.1.3 Post-Order

10.1.4 Level-Order

10.2 Graph Traversals

10.2.1 Depth First Search

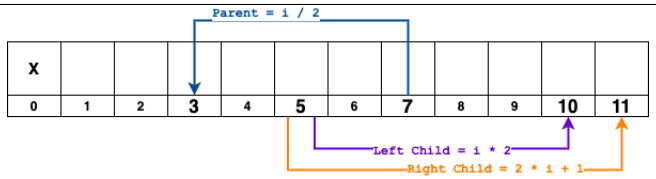
10.2.2 Breadth First Search

11 Array Based Binary Tree

When storing values in an array as if it were a binary tree, you use the current location: **i** and multiply it by 2 for the location of the left child, or multiply by 2 and add 1 for the location of the right child. For this reason, we ignore the 0^{th} location in the array, since zero multiplied by anything is ... well ... zero.

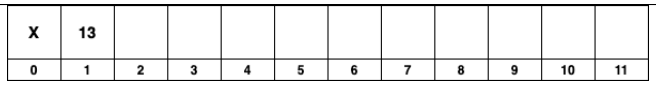
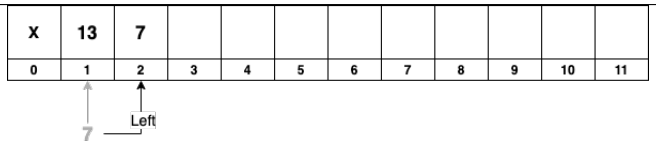
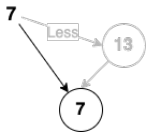
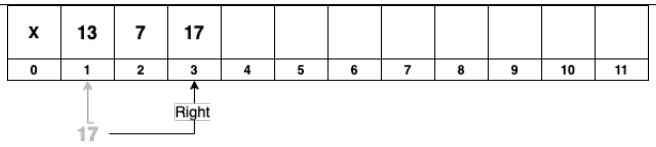
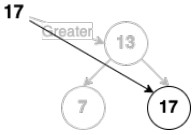
The table below shows us the calculations for *Left Child*, *Right Child*, and *Parent* in the left column. The right column gives a visual example in which we can see that index **7** would have **3** as its parent ($i/2$ or $7/3$) and the left and right children of index **5** are **10** and **11** respectively ($2*5$ and $2*5+1$).

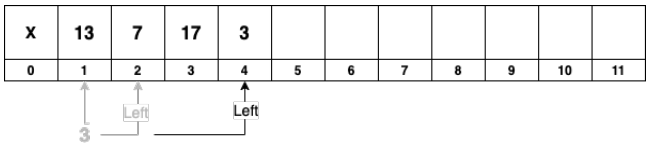
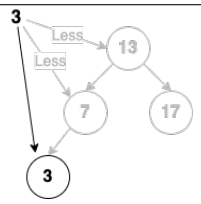
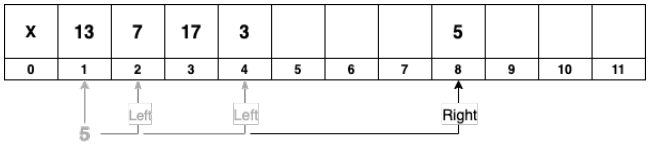
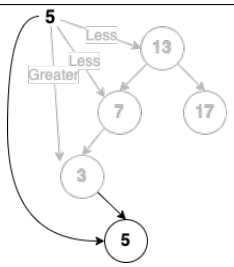
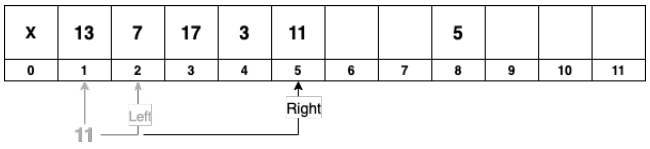
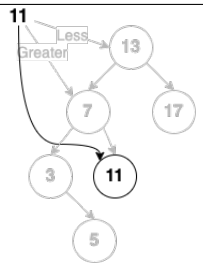
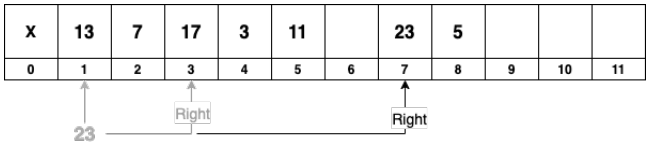
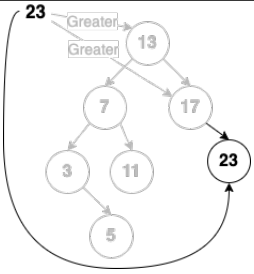
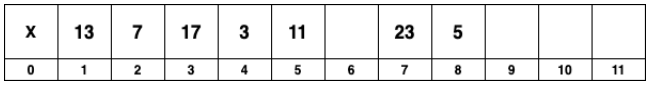
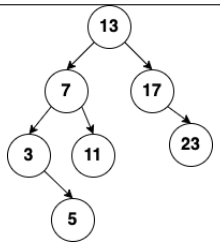
11.0.1 Formulas

Formulas	Visualization
• Left Child $= 2*i$ • Right Child $= 2*i+1$ • Parent $= i/2$	

11.0.2 Example

Below is an example inserting the values 13, 7 , 3, 17, 5, 11, 23 into an array based tree, along with it's binary tree equivalent.

Event	Array Based Tree	Binary Tree
Insert 13	 The root of the tree is empty. So location **1** gets the value.	
Insert 7	 7 is less than 13, go left. Its empty, store it there.	
Insert 17	 17 is greater than 13, go right. Its empty, store it there.	

Insert 3	 3 is less than 13, go left. 3 is less than 7, go left. Its empty, store it there.	
Insert 5	 5 is less than 13, go left. 5 is less than 7, go left. 5 is greater than 3, go right. Its empty, store it there.	
Insert 11	 11 is less than 13, go left. 11 is greater than 7, go right. Its empty, store it there.	
Insert 23	 23 is greater than 13, go right. 23 is greater than 17, go right. Its empty, store it there.	
Done		

You can see the resulting array and tree above. Notice the empty slot at index 6. This is the type of thing you want to avoid with array based structures. One empty slot is not a big deal you say? Probably not. However, this is a small forced example. Why don't you continue this example on paper and insert the values 2 then 1. How big did you need to resize your array?

12 Heap Data Structure

Name: Binary Heap
Invented: 1964
Invented by: J. W. J. Williams
Structure / Container: Array

Complexity

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(n)	O(n)
Insert	O(1)	O(log n)
Find-min	O(1)	O(1)
Delete-min	O(log n)	O(log n)

When you hear the term *heap* your brain can go one of two ways: heap memory or heap data structure. Early on in your CS career, you lean toward the memory version of a heap, since that is fresh on your brain. You just got through allocating dynamic memory with the "new" operator. Where does dynamic memory get stored? Well, the heap! However, this section is about a heap memory that allows for easy implementation of an efficient priority queue[5, 7]. So, are you wondering why we have a type of memory called "the heap" and a data structure call "a heap"? You aren't the only one. Donald Knuth said that back in 1975, several authors began to call the pool of available memory (not allocated to your process) a "heap"[18], but he never really clarified who or why. There are other "guesses" but it's not really clear. My opinion is that since "heap" has multiple meanings, it might just fit both scenarios.

Heap Defintion:

- n. A group of things placed or thrown, one on top of the other.
- n. A great deal; a lot.

A heap data structure is a group of items, placed one on top of another (with a special order to them), and heap memory could also refer to how the memory is allocated, OR to the excess memory not assigned to any process. Who knows?!? It's a conundrum.

This section will discuss the data structure variety that implements a priority queue. More specifically, we will describe an array based implementation of a binary heap. Because of how heap operations work, using an array to store values works. Even though it is possible to store a binary tree in an array, it doesn't mean that we should do it unless certain properties are maintained. If these properties are not maintained, its a sketchy situation at best. Meaning, lots of wasted space and empty slots. See section Array Based Binary Search Trees for information on how to store a binary search tree in an array.

12.1 Array Based Binary Heap

Knowing how to store values in an array as a binary search tree requires the programmer to implement certain algorithms. Likewise, to turn that array based binary tree into a binary heap, we

must alter the algorithm (or operations) on how we interact with the data structure. It's the specific algorithms that we apply to generic containers (like an array) that makes them what they are. In the next section, we will take our simple binary search tree algorithm to another level when we add the functionality to implement a binary heap.

12.1.1 Background

A binary heap is a heap data structure that takes the form of a binary tree. The binary heap was introduced by J. W. J. Williams in 1964, as a data structure for heapsort[2]. It turns out, it could be used for more than sorting values. In fact heaps are a very good way of implementing **priority queues**'s. All we have to do with our binary tree to make it a heap is make sure that it fulfills the following properties:

1. It stays a complete binary tree.
2. Each value stored in the tree is either less than or equal (Min Heap) or greater than or equal (Max Heap) to its parent.
3. If a child does not meet the previous property, it will be swapped with its parent.

A heap data structure uses multiple methods to maintain its heap structure:

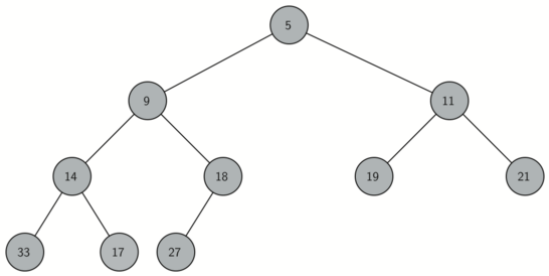
- Insert
- Remove
- Heapify
- BubbleUp
- BubbleDown

12.1.2 Overview

In order to make our heap work efficiently, we will take advantage of the logarithmic nature of the binary tree to represent our heap. In order to guarantee logarithmic performance, we must keep our tree balanced[7]. A tree is balanced if the difference of the left and right subtrees differ by no more than 1. We determine if the entire tree is balanced by looking at the height difference from the root's left and right subtrees.

We already mentioned we want our heap implementation to be a complete binary tree. This means each level has all of its nodes. The exception to this is the bottom level of the tree, which we fill in from left to right. The Figure below shows an example of a complete binary tree with its array implementation.
How do we ensure that our heap implementation remains a complete tree? By using the array implementation we discussed above, and inserting new items into the tree by placing them into the next open element in the array. Look at the figure above. By inserting another value at array location 11, we give node 18 a right child, or node 27 a right sibling. Our tree stays "complete". If we were to expand our array and keep inserting into the next available slot, we would continue to fill the tree in from left to right at the bottom, until we started a new row. We never get "holes" in our array, and our tree stays complete and balanced.

[7] Go see the binary tree section for info about logarithmic performance.



(a) A Complete Binary Tree

X	5	9	11	14	18	19	21	33	17	27	
0	1	2	3	4	5	6	7	8	9	10	11

(b) Array Representation

Figure 10: A Complete Tree and its Array Representation

12.1.3 Heap Operations

Even though we take advantage of an array based binary tree implementation, it doesn't mean we will insert items the same. In the array based binary tree (not a heap yet!) we would compare our new value to the root of the tree, then find the location to store that value by comparing to the root, and then propagate **down** the tree moving left and right as needed (see figure 11).

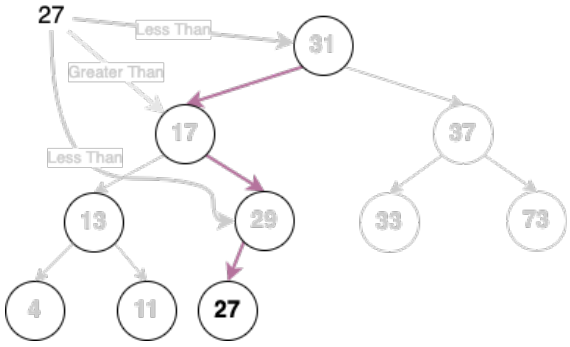


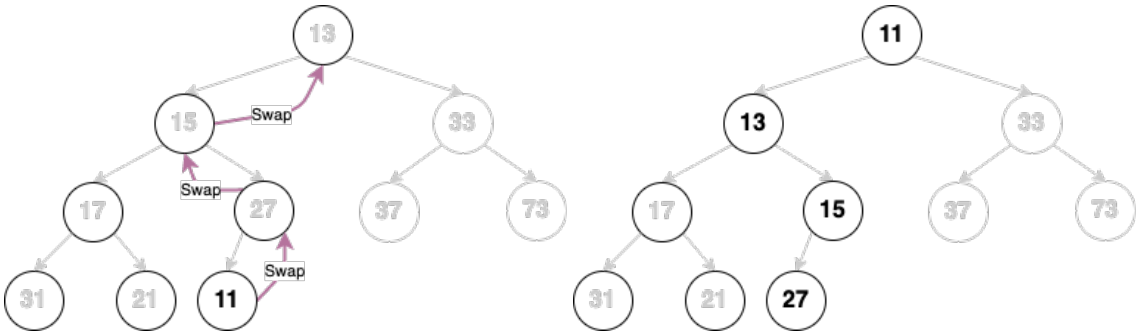
Figure 11: Inserting into Binary Search Tree

Because the heap property doesn't care about a total ordering of the tree, it only cares about the relationship between parent and child, we can insert values a little differently.

Insert

The easiest, and most efficient, way to add an item to a heap is to simply append the item to the next available slot in the array. The good news about this is that it will ALWAYS result in a complete tree. The bad news about appending is that we will very likely violate the heap structure property and have to re-arrange values based on whether it is a min-heap or a max-heap.

Fixing any violations of the heap structure property can be done by comparing the newly added



(a) Bubbling Up (b) After Swaps

Figure 12: Inserting value into a heap

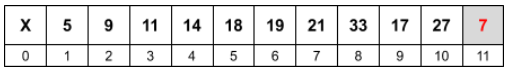
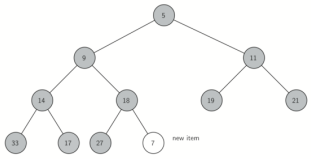
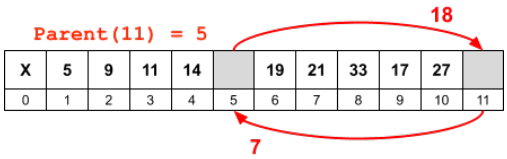
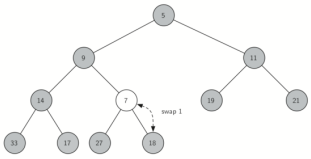
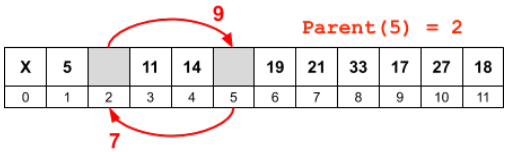
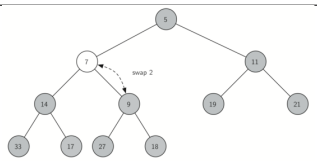
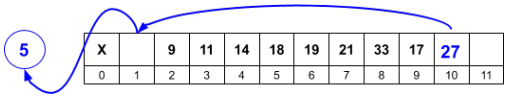
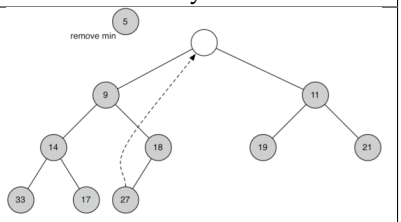
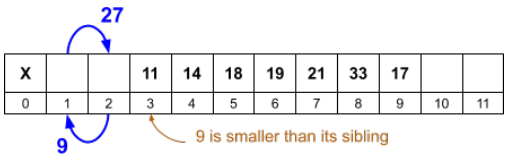
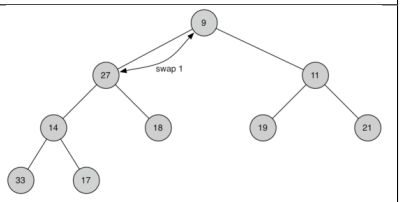
	Array	Binary Tree
Append to last slot.		
Swap with smaller parent.		
Swap with smaller parent (again).		

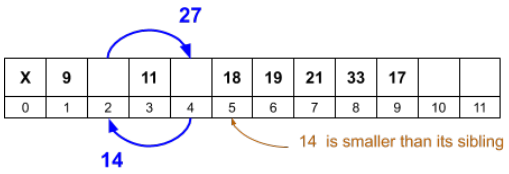
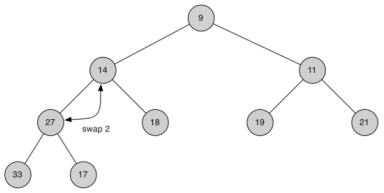
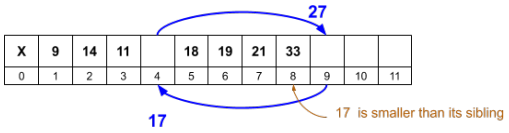
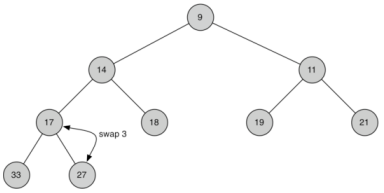
Table 6: Example insert showing comparisons while bubbling up.

item with its parent. If the newly added item is less than its parent (assuming a min heap), then we can swap the item with its parent. Since the tree is always a complete tree and therefore balanced, we can assume worse case that an insert item will have to swap `O(lg n)` times.

In **figure 12** we see an example showing the series of swaps needed to percolate the newly added item up to its proper position in the tree. In table 6 we can see the steps (**using different values**) broken down better.

Notice that when we percolate an item up, we are restoring the heap property between the newly added item and the parent. We are also preserving the heap property for any siblings. Of course, if the newly added item is very small, we may still need to swap it up another level. In fact, we may need to keep swapping until we get to the top of the tree. **Figure 12** shows the `Bubble UP` or (percUp, siftUp, etc.) method, which swaps a new item as far up in the tree as it needs to go to maintain the heap property. Again, we use ***i / 2*** to find the parent and swap the values if the child

	Array	Binary Tree
Remove smallest value (5), swap with last item in array (27).		
Then start to place the swapped value into it's proper location by comparing it to its children and restoring the heap property.		

	Array	Binary Tree
Continue comparing and swapping as long as the heap property is violated (min-heap: parents must be smaller than their children).		
Stop swapping when no child is smaller, or there are no more children to compare to.		

is smaller than the parent[8].

`RemoveMin`

Since the heap property requires that the root of the tree be the smallest item in the tree, finding the minimum item is easy. The involved part of `RemoveMin` is restoring full compliance with the heap structure and heap order properties after the root has been removed. Restoring compliance can be solved in two simple steps: 1) A swap, and 2) a Bubble Down. More specifically:

1. Swap the last item in the heap (from the back) to the top of the heap.
2. Bubble this item down to its proper position since it will definitely ruin the heap order.

[8]Remember, this is a **minheap**, for **maxheap** we swap if child is larger.

Heapify

To finish our discussion of binary heaps, we will look at a method to build an entire heap from an array of keys. If we insert items into the heap, one at a time we would get a performance of ***O(n lg n)***. Remember the height of a complete tree is ***lg n*** (so each new item cannot bubble up further than *lg n*). By inserting *n* items, we get ***O(n lg n)*** performance.

The heap property specifies that each node in a binary heap must be at least as large as both of its children. In particular, this implies that the largest item in the heap is at the root. Sifting down and sifting up are essentially the same operation in opposite directions: move an offending node until it satisfies the heap property:

siftDown swaps a node that is too small with its largest child (thereby moving it down) until it is at least as large as both nodes below it. siftUp swaps a node that is too large with its parent (thereby moving it up) until it is no larger than the node above it. The number of operations required for siftDown and siftUp is proportional to the distance the node may have to move. For siftDown, it is the distance to the bottom of the tree, so siftDown is expensive for nodes at the top of the tree. With siftUp, the work is proportional to the distance to the top of the tree, so siftUp is expensive for nodes at the bottom of the tree. Although both operations are O(log n) in the worst case, in a heap, only one node is at the top whereas half the nodes lie in the bottom layer. So it shouldn't be too surprising that if we have to apply an operation to every node, we would prefer siftDown over siftUp.

The buildHeap function takes an array of unsorted items and moves them until they all satisfy the heap property, thereby producing a valid heap. There are two approaches one might take for buildHeap using the siftUp and siftDown operations we've described.

Start at the top of the heap (the beginning of the array) and call siftUp on each item. At each step, the previously sifted items (the items before the current item in the array) form a valid heap, and sifting the next item up places it into a valid position in the heap. After sifting up each node, all items satisfy the heap property.

Or, go in the opposite direction: start at the end of the array and move backwards towards the front. At each iteration, you sift an item down until it is in the correct location.

Which implementation for buildHeap is more efficient? Both of these solutions will produce a valid heap. Unsurprisingly, the more efficient one is the second operation that uses siftDown.

Let h = log n represent the height of the heap. The work required for the siftDown approach is given by the sum

(0 * n/2) + (1 * n/4) + (2 * n/8) + ... + (h * 1). Each term in the sum has the maximum distance a node at the given height will have to move (zero for the bottom layer, h for the root) multiplied by the number of nodes at that height. In contrast, the sum for calling siftUp on each node is

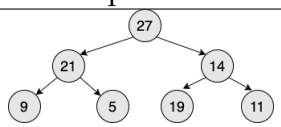
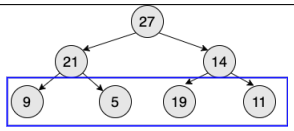
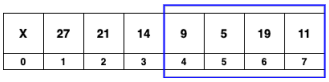
(h * n/2) + ((h-1) * n/4) + ((h-2)*n/8) + ... + (0 * 1). It should be clear that the second sum is larger. The first term alone is hn/2 = 1/2 n log n, so this approach has complexity at best O(n log n).

==========

However, if start with an unordered array of keys we can improve that time to

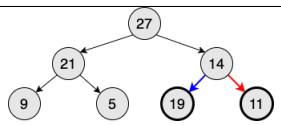
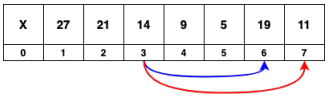
Inserting n items gives us the first "n". And since we maintain a complete tree, it will be no taller than lg n, and so each can bubble up no further than lg n.

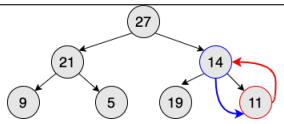
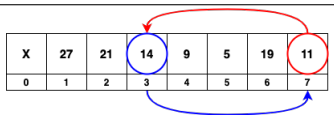
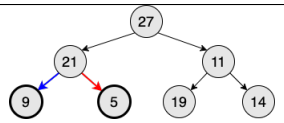
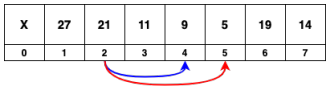
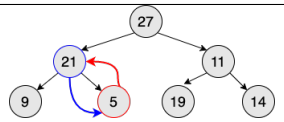
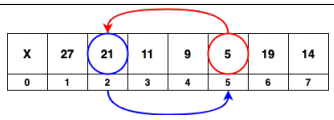
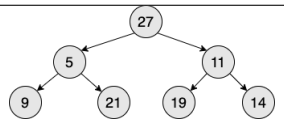
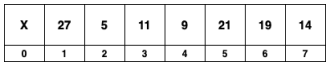
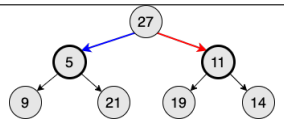
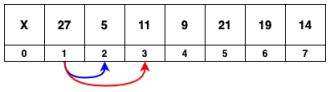
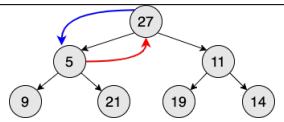
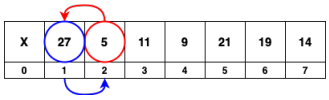
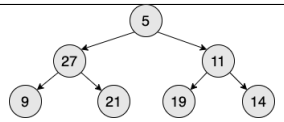
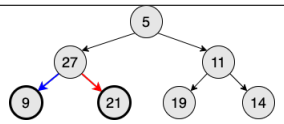
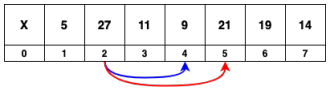
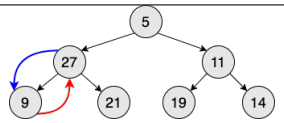
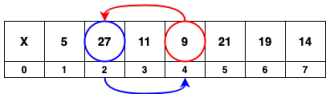
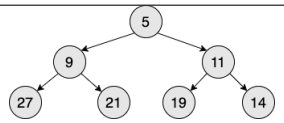
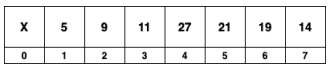
The first method you might think of may be like the following. Given a list of keys, you could easily build a heap by inserting each key one at a time. Since you are starting with a list of one item, the list is sorted and you could use binary search to find the right position to insert the next key at a cost of approximately ***O(lg n)*** operations. However, remember that inserting an item in the middle of the list may require ***O(n)*** operations to shift the rest of the list over to make room for the new key.

	Tree Representation	Array Representation
Heapify: Starts with an array of unordered numbers.		
We can ignore the bottom half of the array since all their children will be off the end of the array.		

Therefore, to insert ***n*** keys into the heap would require a total of ***O(n lg n)*** operations. However, if we start with an entire list then we can build the whole heap in ***O(n)*** operations.

The assertion that we can build the heap in ***O(n)*** may seem a bit mysterious at first, and were not proving anything. However, the key to understanding that you can build the heap in ***O(n)*** is to remember that the ***O(lg n)*** factor is derived from the height of the tree. For most of the work in `Heapify`, the tree is shorter than ***lg n***.
Using the fact that you can build a heap from a list in ***O(n)*** time, you could easily construct a sorting algorithm that uses a heap and sorts a list in ***O(n lg n)*** cost.

Find first location that has 1 or two children, and see if we need to swap (still assume min heap). Index 3 in this table our first candidate.		

Make the swap with the smaller of the two children (11 < 19). 14 cannot bubble down any further so we stop.		
We move up one index, and look at our children. Index 2 has children at locations 4 and 5.		
We swap index 2 with the smallest of its children (5 < 9). 21 can bubble down no further so we stop.		
We processed indexes 3 then 2 with minimal swapping. We still have index one to process.		
We compare index 1 with its two children at indexes 2 and 3.		
We swap index 1 with the smaller of its two children (5 < 11).		
The value now at index 2 still has room to bubble down.		
We again compare it to its children.		
The value once again gets swapped with the smaller of its two children. So, this value bubbled down from the 1st index to the bottom of the tree (distance of lg n).		
We now have a valid min-heap.		

13 AVL Trees

Name: Avl Tree
Invented: 1962
Invented by: Georgy Adelson-Velsky and Evgenii Landis
Structure / Container: Linked Structure

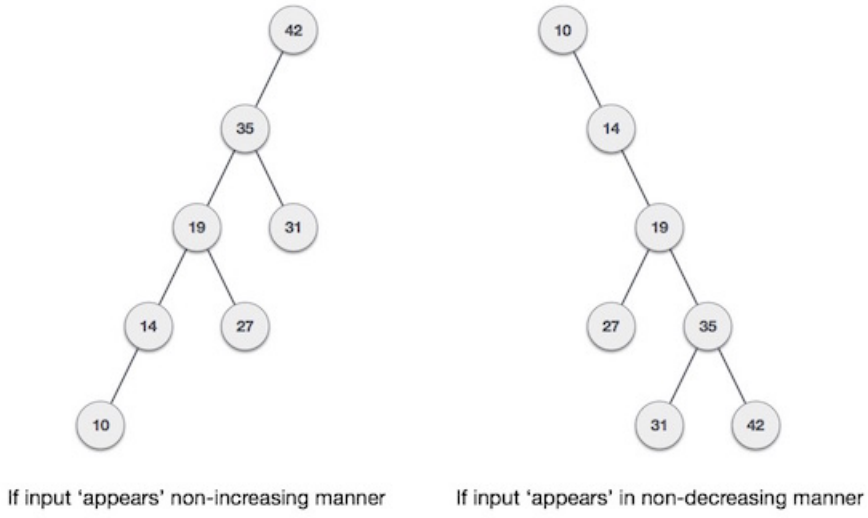
Complexity

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Table 7: Avl Tree Overview[1]

Copied From TutorialsPoint AvlTrees [15]

What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this:



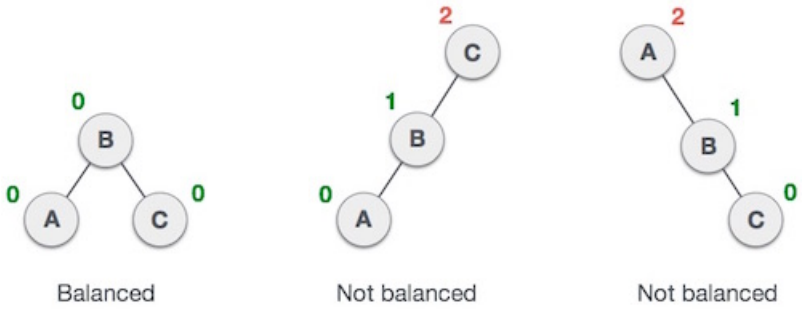
If input 'appears' non-increasing manner

If input 'appears' in non-decreasing manner

It is observed that BST's worst-case performance is closest to linear search algorithms, that is (n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson**, **Velski** & **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor** .

Here we see that the first tree is balanced and the next two trees are not balanced



In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

```
BalanceFactor = height(left-subtree) - height(right-subtree)
```

If the difference in the height of left and right sub-trees is more than **1**, the tree is balanced using some rotation. You might think that since we don't know which side of that equation might be bigger, than we might a value that is negative. That is ok! We will use the sign of the result to determine which direction to apply our rotation which we will describe next.

13.1 AVL Rotations

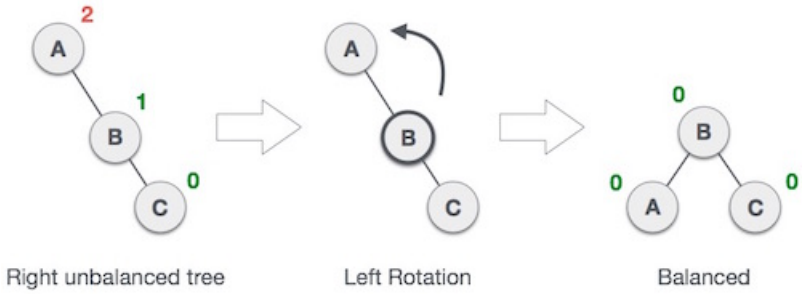
To balance itself, an AVL tree may perform the following four kinds of rotations

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are what we classify as a **single rotation** and the next two are classified as a **double rotation**. The minimum height of our tree in which a rotation can be applied is **2**. The best way to understand a tree rotation is to follow the steps visually. So, let's do that.

13.1.1 Left Rotation

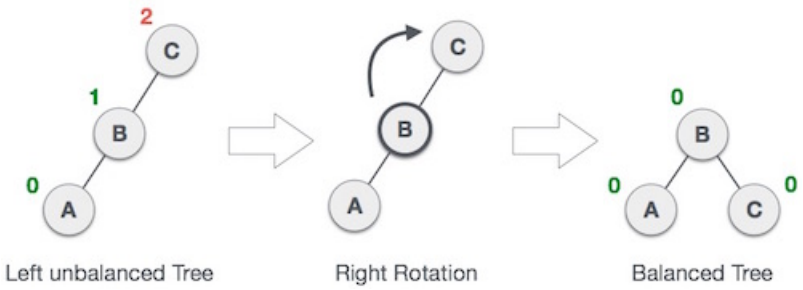
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation. Notice the balance factor of "A" (the leaf node) is **2**, "B" is **1**, and "C" is **0** (recall BalanceFactor definition from above).



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

13.1.2 Right Rotation

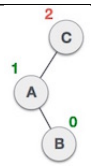
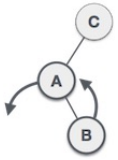
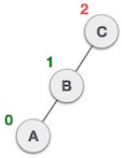
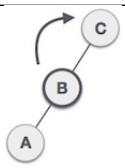
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

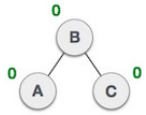


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

13.1.3 Left-Right Rotation

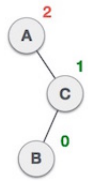
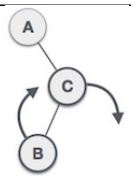
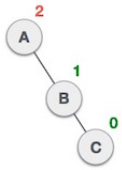
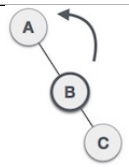
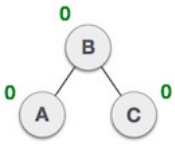
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.

	The tree is now balanced.

13.1.4 Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.
	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.
	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.
	The tree is now balanced.

14 Hashing

Name: Hash Table
Invented: 1953
Invented by: Multiple People [9]
Structure / Container: Linked or Array Structure

Complexity

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

Table 10: Hash Table Overview[4]

14.1 Overview

When we say: “hash table” you should think of a an array data structure that strives to be a “constant time lookup table”. It works similar to an array, but we get to use strings as the indexes. Up to now, most of you are pretty used to arrays having integers as indexes. Look at figure 13 below. Let’s call this container "produce" and it stores the number of cases for various produce product. If I access the produce container at index 4 like so: *produce[4]*, I will see that there are 13 cases of "number 4 produce". But, what is "number 4" produce? I would need to use parallel arrays that store additional information about the produce so I could look the name of that particular item up.

33	44	23	17	13	11	73	5	27	10
0	1	2	3	4	5	6	7	8	9

Figure 13: Typical container indexed by integers.

What if I could access that same structure using a string (see figure 14) like so: *produce["eggplant"]*? It would also return the number of cases, but I would actually know which item I was searching for. I think you may agree that it is a much simpler method (conceptually) of storing and accessing data. This is not the biggest motivation behind hash tables, it’s a convenient by-product of their implementation. The biggest motivation is still having the ability to do constant time lookups.

[9] The idea of hashing arose independently in different places. In January 1953, Hans Peter Luhn wrote an internal IBM memorandum that used hashing with chaining. Gene Amdahl, Elaine M. McGraw, Nathaniel Rochester, and Arthur Samuel implemented a program using hashing at about the same time. Open addressing with linear probing (relatively prime stepping) is credited to Amdahl, but Ershov (in Russia) had the same idea.

33	44	23	17	13	11	73	5	27	10
apple	fig	banana	kiwi	eggplant	squash	cherry	orange	pear	ganja

Figure 14: Dictionary container indexed by strings.

Many languages provide hash tables as a “built in” data structure, a few are listed below:

- Python gives us **dictionaries**
- Php gives us **associative arrays**
- C++ gives us **maps**

Note: I need to mention that Python dictionaries and Php associative arrays are implemented as a hash table, but maps in C++ are implemented as red black trees. Huh? There is an ordered map in C++ which is implemented as a hash table under the hood, but the unordered version is not. If this interests you, go do some research on the matter, otherwise I won’t bring it up again.

14.2 Terms

- **Hash Table**: an array in which keys are mapped to array positions by a hash function.
- **Key**: The value received by the hash function which gets turned into an integer value used as an index.
- **Hash Function**: function H, that receives a key k, and turns it into an integer value i (array index). The returned integer value can be adjusted to fit an array by using modulus like so: $(H(k) = i \,\%\, array\_size)$
- **Collision**: When a hash function returns the same key (or array index) for multiple distinct values. The goal is to avoid this, but it is mostly unavoidable.
- **Collision Handling Technique**: If and when collisions do happen, this is the method in which we deal with them. Collision handling is all about where to place the hashed value when it gets hashed to an occupied position.
- **Load Factor**: The load factor λ, is a measure of how full the hash table is as a ratio between slots in the table, and number of items currently in the table. The "load factor" is also used to determine when the capacity of the table is increased.

14.3 Hash Functions

NEEDS SOME CLEANING UP

The main thing we think of in an algorithms class when we hear "hash function" is "hash tables". Well, that’s what I think of anyway. However, hash functions can be used for more than just hash tables, and more than what I’ve listed below:

- Cryptographic or 1 Way Hash
- Error Correction
- Checking for changes (file(s) or directory(s))
- Hash table (constant time lookup)

The discussion of hash functions below deals with the use of hash functions in regards to hash tables and constant time lookup. So hash functions take a parameter as input (the "key") and return an integer in return.

- Hash functions are typically designed to return a value in the full unsigned range of an integer.
- For a 32-bit integer, this means that the hash functions will return a value in the range $[\mathbf{0..4294967296}]$ or $[\mathbf{0..2^{32}}]$

What we do with this (possibly) extremely large number is up to us. Typical use is to convert the possibly large number to a number that is between 0 and your table size. Below are a few simple methods for doing this.

14.3.1 Simple Hash Functions

The following functions map a single integer key k to a small integer bucket value $h(k)$. We also use m to indicate the size of the hash table (number of buckets).

Division method (Cormen)

- Choose a prime that isn't close to a power of 2.
- $h(k) = k \; mod \; m$. Works badly for many types of patterns in the input data.

Knuth Variant on Division

- $h(k) = k(k+3) \; mod \; m$
- Supposedly works much better than the raw division method.

Multiplication Method (Cormen)

- Choose m to be a power of 2. Let A be some random-looking real number.
- Knuth suggests $M = \mathbf{0.5} * (sqrt(\mathbf{5}) - \mathbf{1})$. Then do the following:

```
1       s = k*A
2       x = fractional part of s
3       h(k) = floor(m*x)
```

This seems to be the method that the theoreticians like.

To do this quickly with integer arithmetic, let w be the number of bits in a word (e.g. 32) and suppose m is 2^p. Then compute:

```
s = floor(A * 2^w)
x = k*s
h(k) = x >> (w-p)       // i.e. right shift x by (w-p) bits
                        // i.e. extract the p most significant
                        // bits from x
```

14.3.2 Hashing sequences of characters

The hash functions in this section take a sequence of integers $k = k_1, ..., k_n$ and produce a small integer bucket value $h(k)$. Again, m is the size of the hash table (number of buckets). The sequence of integers might be a list of integers or it might be an array of characters (like a string).

The specific tuning of the following algorithms assumes that the integers are all, in fact, character codes.

- In C++, a character is a char variable which is an 8-bit integer.
- ASCII uses only 7 of these 8 bits. Of those 7, the common characters (alphabetic and number) use only the low-order 6 bits. And the first of those 6 bits primarily indicates the case of characters, which is relatively insignificant.
- So the following algorithms concentrate on preserving as much information as possible from the last 5 bits of each number, and make less use of the first 3 bits.

When using the following algorithms, the inputs k_i must be unsigned integers. Feeding them signed integers may result in odd behavior.

For each of these algorithms, let h be the output value. Set h to 0. Walk down the sequence of integers, adding the integers one by one to h. The algorithms differ in exactly how to combine an integer k_i with h. The final return value is $h \bmod m$.

CRC variant:

Do a 5-bit left circular shift of h. Then XOR in k_i. Specifically:

```
highorder = h & 0xf8000000    // extract high-order 5 bits from h
                              // 0xf8000000 is the hexadecimal representation
                              //   for the 32-bit number with the first five
                              //   bits = 1 and the other bits = 0
h = h << 5                    // shift h left by 5 bits
h = h ^ (highorder >> 27)     // move the highorder 5 bits to the low-order
                              //   end and XOR into h
h = h ^ ki                    // XOR h and ki
```

PJW hash:

Left shift h by 4 bits. Add in k_i. Move the top 4 bits of h to the bottom. Specifically:

```
    // The top 4 bits of h are all zero
    h = (h << 4) + ki               // shift h 4 bits left, add in ki
    g = h & 0xf0000000              // get the top 4 bits of h
    if (g != 0)                     // if the top 4 bits aren't zero,
       h = h ^ (g >> 24)            //   move them to the low end of h
       h = h ^ g
    // The top 4 bits of h are again all zero
```

PJW and the CRC variant both work well and there's not much difference between them. We believe that the CRC variant is probably slightly better because:

- It uses all 32 bits. PJW uses only 24 bits. This is probably not a major issue since the final value m will be much smaller than either.
- 5 bits is probably a better shift value than 4. Shifts of 3, 4, and 5 bits are all supposed to work OK.
- Combining values with XOR is probably slightly better than adding them. However, again, the difference is slight.

BUZ hash:
Set up a function R that takes 8-bit character values and returns random numbers. This function can be precomputed and stored in an array. Then, to add each character k_i to h, do a 1-bit left circular shift of h and then XOR in the random value for k_i. That is:

```
    highorder = h & 0x80000000    // extract high-order bit from h
    h = h << 1                    // shift h left by 1 bit
    h = h ^ (highorder >> 31)     // move them to the low-order end and
                                  // XOR into h
    h = h ^ R[ki]                 // XOR h and the random value for ki
```

Rumor has it that you may have to run a second hash function on the output to make it random enough. Experimentally, this function produces good results, but is a bit slower than the **CRC** variant and **PJW**.

Additive Hash

```
//---------------------------------------------------------------
// Additive Hash
// Adds all of the characters together using ascii values.
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//---------------------------------------------------------------
unsigned int hash_functions::add_hash (string val)
{
  unsigned int h = 0;
```

```
10
11    for (unsigned int  i = 0; i < val.length(); i++ )
12      h += val[i];
13
14    return h;
15  }
```

XOR hash

```
1   //---------------------------------------------------------------------
2   // XOR hash
3   // Repeatedly folds the bytes together using the XOR operation to
4   // produce a seemingly random hash value.
5   // @Param: string val - word to be hashed
6   // @Returns: unsigned int - hash key value
7   //---------------------------------------------------------------------
8   unsigned int hash_functions::xor_hash (string val)
9   {
10    unsigned h = 0;
11
12    for (unsigned i = 0; i < val.length(); i++ )
13      h ^= val[i];
14
15    return h;
16  }
```

Rotating hash

```
1   //---------------------------------------------------------------------
2   // Rotating hash
3   // The rotating hash is identical to the XOR hash except instead of simply
4   // folding each byte of the input into the internal state, it also performs
5   // a fold of the internal state before combining it with the each byte of
6   // the input
7   // @Param: string val - word to be hashed
8   // @Returns: unsigned int - hash key value
9   //---------------------------------------------------------------------
10  unsigned int hash_functions::rot_hash (string val)
11  {
12    unsigned h = 0;
13
14     for (unsigned i = 0; i < val.length(); i++ )
15        h = ( h << 4 ) ^ ( h >> 28 ) ^ val[i];
16
17    return h;
18  }
```

Bernstein hash

```
//-----------------------------------------------------------------
// Bernstein hash
// Dan Bernstein created this algorithm and posted it in a newsgroup.
// It is known by many as the Chris Torek hash because Chris went a
// long way toward popularizing it. Since then it has been used
// successfully by many, but despite that the algorithm itself is not
// very sound when it comes to avalanche and permutation of the internal
// state. It has proven very good for small character keys, where it
// can outperform algorithms that result in a more random distribution.
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//-----------------------------------------------------------------
unsigned int hash_functions::bernstein_hash (string val)
{
  unsigned h = 0;

        for (unsigned i = 0; i < val.length(); i++ )
      h = 33 * h + val[i];

  return h;
}
```

Modified Bernstein hash

```
//-----------------------------------------------------------------
// Modified Bernstein hash
// A minor update to Bernstein's hash replaces addition with XOR for
// the combining step. This change does not appear to be well known
// or often used, the original algorithm is still recommended by
// nearly everyone, but the new algorithm typically results in a
// better distribution.
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//-----------------------------------------------------------------
unsigned int hash_functions::mod_bernstein_hash (string val)
{
  unsigned h = 0;

        for (unsigned i = 0; i < val.length(); i++ )
      h = 33 * h ^ val[i];

  return h;
}
```

Shift-Add-XOR hash

```
//-----------------------------------------------------------------
// Shift-Add-XOR hash
```

```
// The shift-add-XOR hash was designed as a string hashing function,
// but because it is so effective, it works for any data as well with
// similar efficiency. The algorithm is surprisingly similar to the
// rotating hash except a different choice of constants for the rotation
// is used, and addition is a preferred operation for mixing. All in
// all, this is a surprisingly powerful and flexible hash. Like many
// effective hashes, it will fail tests for avalanche, but that does
// not seem to affect its performance in practice.
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//-----------------------------------------------------------------
unsigned int hash_functions::shift_add_xor_hash (string val)
{
  unsigned h = 0;

   for (unsigned i = 0; i < val.length(); i++ )
      h ^= ( h << 5 ) + ( h >> 2 ) + val[i];

  return h;
}
```

FNV hash

```
//-----------------------------------------------------------------
// FNV hash
// The FNV hash, short for Fowler/Noll/Vo in honor of the creators,
// is a very powerful algorithm that, not surprisingly, follows the
// same lines as Bernstein's modified hash with carefully chosen
// constants. This algorithm has been used in many applications with
// wonderful results, and for its simplicity, the FNV hash should be
// one of the first hashes tried in an application. It is also
// recommended that the FNV website be visited for useful descriptions
// of how to modify the algorithm for various uses.
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//-----------------------------------------------------------------
unsigned int hash_functions::fnv_hash (string val)
{
  unsigned int h = 2166136261;

   for (unsigned i = 0; i < val.length(); i++ )
      h = ( h * 16777619 ) ^ val[i];

  return h;
}
```

One-at-a-Time hash

```
//---------------------------------------------------------------------
// One-at-a-Time hash
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//---------------------------------------------------------------------
unsigned int hash_functions::one_at_a_time_hash (string val)
{
  unsigned int h = 0;

  for (unsigned i = 0; i < val.length(); i++ ) {
    h += val[i];
    h += ( h << 10 );
    h ^= ( h >> 6 );
  }

  h += ( h << 3 );
  h ^= ( h >> 11 );
  h += ( h << 15 );

  return h;
}
```

Jsw hash

```
//---------------------------------------------------------------------
// Jsw hash
// This is a hash of my own devising that combines a rotating hash
// with a table of randomly generated numbers. The algorithm walks
// through each byte of the input, and uses it as an index into a
// table of random integers generated by a good random number
// generator. The internal state is rotated to mix it up a bit, then
// XORed with the random number from the table. The result is a
// uniform distribution if the random numbers are uniform. The size
// of the table should match the values in a byte. For example, if a
// byte is eight bits then the table would hold 256 random numbers.
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//---------------------------------------------------------------------
unsigned int hash_functions::jsw_hash (string val)
{
  unsigned h = 16777551;

  for (unsigned i = 0; i < val.length(); i++ )
    h = ( h << 1 | h >> 31 ) ^ rand_vals[val[i]];

  return h;
}
```

Elf Hash

```
//-------------------------------------------------------------------
// Elf hash
// The ELF hash function has been around for a while, and it is believed
// to be one of the better algorithms out there. Though ELF hash does not
// perform sufficiently better than most of the other algorithms
// in this collection to justify its slightly more complicated implementation.
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//-------------------------------------------------------------------
unsigned int hash_functions::elf_hash (string val)
{
  unsigned h = 0 , g;

        for (unsigned i = 0; i < val.length(); i++ ) {
          h = ( h << 4 ) + val[i];
          g = h & 0xf0000000L;

          if ( g != 0 )
                h ^= g >> 24;

          h &= ~g;
        }

  return h;
}
```

15 Intro to Sorting

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Sorting algorithms provide an introduction to a variety of core algorithm concepts, such as categories of algorithms (e.g. divide and conquer) , best-, worst- and average-case analysis, time-space trade-offs, as well as upper and lower bounds.

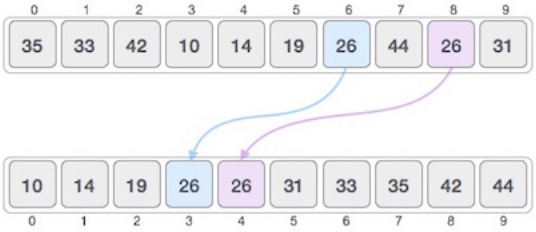
15.1 Classification

How do we classify sorting algorithms? Below is a list of general criteria that lets us group like algorithms together.

- **Computational complexity** (worst, average and best behavior) of element comparisons in terms of the size of the list (n). For typical sorting algorithms, a good behavior is $O(n\ log\ n)$ and a bad behavior is $O(n^2)$.

- **Computational complexity of swaps** How complex and how much additional memory (if any) is used for algorithms that sort in place (like quick sort).
- **Memory usage** (and use of other computer resources). In particular, some sorting algorithms are "in place". This means that they need only $O(1)$ or $O(log\ n)$ memory beyond the items being sorted and they don't need to create auxiliary locations for data to be temporarily stored, as in other sorting algorithms.
- **Recursion**: Some algorithms are either recursive or non-recursive.
- **Stability**: Stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).
- Whether or not they are a **comparison sort**. A comparison sort examines the data only by comparing two elements with a comparison operator.
- **General methods**: insertion, exchange, selection, merging, etc.
- **Adaptability**: Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

15.1.1 Stability



Stable sorting algorithms maintain the relative order of records with equal keys. If all keys are different then this distinction is not necessary. But if there are equal keys, then a sorting algorithm is stable if whenever there are two records (let's say R and S) with the same key, and R appears before S in the original list, then R will always appear before S in the sorted list [14]. When equal elements are indistinguishable, such as with integers or social security numbers, or more generally, any data where the entire element is the key, stability is not an issue. But what if we do have keys that are similar, but can be differentiated in other ways. It's not always easy to find a legit example, but this one should do the trick. What if we were sorting playing cards on the face value of the card? Usually we could have a C++ struct (or class) or a Python dictionary to store a card and all of its info. The face value is just one field that we could sort on, but if we use the face value, we also know there are four identical values, one in every suit. In figure 15, a **stable sort** will keep the fives in the same order in which they were encountered. If the fives get shifted around, its considered an **unstable sort**. Unstable in this context does not mean "bad". It's just a term to describe the behavior of a sorting algorithm so you have additional information when choosing a sort method.

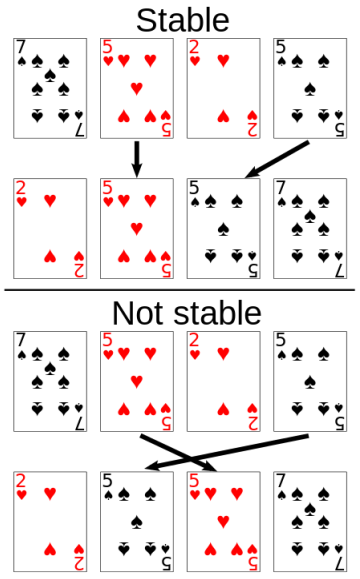


Figure 15: Stable Sort Example

To summarize, let me ask you this "Why does it matter if a sort is stable or not?" Because the term itself helps us understand the behavior of the algorithm we are using, and can help us make

informed decisions. Stable sorts preserve the previous order of items if they were not explicitly moved by the sorting routine. If that previous order really matters, then I find and choose a stable sort. If it doesn't matter, I'm free from worrying about whether a sort is stable or not.

15.1.2 Multiple Keys

But what if a stable sort isn't enough to do what you want? What if we need to ensure order on more than one field (using multiple keys)? This happens more than you might think. This would require us to explicitly track multiple key values like a primary key ($\boldsymbol{key_1}$) and a sub key ($\boldsymbol{key_2}$). To maintain order we first sort on key_1 and then sort on key_2 within each "like" value in key_1. We need to be specific about what values key_1 and key_2 represent. As an example, let's sort the following list of playing cards two different ways.

Figure 16: Unsorted Cards

First we will make key_1 be the cards value and key_2 be the cards suit. If we sorted with the keys assigned in that manner we would get:

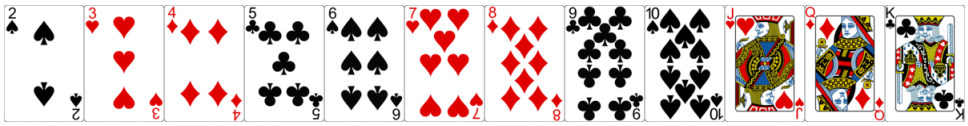


Figure 17: Sorted By Face Value

However if we swapped key_1 and key_2 we would get:

Figure 18: Sorted By Suit and Then Face Value

Let's stick with the cards example. In this case $\boldsymbol{key_1 = face_value}$ and $\boldsymbol{key_2 = suit}$

15.2 Direct Comparison

Algorithm	Best	Average	Worst	Space	Stable	Method
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes	Insertion
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	No	Selection
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes	Exchanging
Quicksort	O(n log n)	O(n log n)	O(n^2)	O(log n)	No	Partitioning
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes	Merging
Heapsort	O(n log n)	O(n log n)	O(n log n)	O(1)	No	Selection

15.3 Bubble Sort

Bubble sort is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. Because it only uses comparisons to operate on elements, it is a comparison sort.

15.3.1 Step-by-Step Example

Assume we have an array **5, 1, 4, 2, 8** and we want to sort the array from the lowest number to the greatest number using bubble sort.

Basic Implementation

```
void bubbleSort(int *arr, int n)
{
    for(int i=0; i<n; i++)
    {
       for(int j=0; j<n-i-1; j++)
       {
          if(array[j]>array[j+1])
          {
             int temp = array[j+1];
             array[j+1] = array[j];
             array[j] = temp;
          }
       }
   }
}
```

An Improved Alternative Implementation

```
void bubbleSort(int *arr, int n)
{
    for(int i=0; i<n; i++)
    {
      bool flag = false;
       for(int j=0; j<n-i-1; j++)
       {
          if(array[j]>array[j+1])
          {
            flag = true;
             int temp = array[j+1];
             array[j+1] = array[j];
             array[j] = temp;
          }
       }
      // No Swapping happened, array is sorted
      if(!flag){
         return;
      }
   }
}
```

Here, instead of doing $n(n-1)$ comparisons, we reduce it to $(n-1)+(n-2)+...+1 = n(n-1)/2$ comparisons.

15.3.2 Performance

- Worst case performance: $O(n^2)$
- Best case performance: $O(n)$
- Average case performance: $O(n^2)$
- Worst case space complexity: $O(n)$ total, $O(1)$ auxiliary

Bubble sort is not a practical sorting algorithm when **n** is large.

15.4 Selection Sort

Selection sort is an in-place comparison sort. It has $O(n^2)$ complexity, making it inefficient on large lists, and generally performs worse than the similar *insertion sort*. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.

15.4.1 Algorithm

1. Find the minimum value in the list
2. Swap it with the value in the first position
3. Repeat the steps above for the remainder of the list (starting at the second position and advancing each time)

Effectively, we divide the list into two parts: the sublist of items already sorted and the sublist of items remaining to be sorted.

```c
// function to swap the the position of two elements
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

void selectionSort(int array[], int size) {
  for (int step = 0; step < size - 1; step++) {
    int min_idx = step;
    for (int i = step + 1; i < size; i++) {

      // To sort in descending order, change > to < in this line.
      // Select the minimum element in each loop.
      if (array[i] < array[min_idx])
        min_idx = i;
    }

    // put min at the correct position
    swap(&array[min_idx], &array[step]);
  }
}
```

15.4.2 Performance

- Worst case performance: $O(n^2)$
- Best case performance: $O(n^2)$
- Average case performance: $O(n^2)$
- Worst case space complexity: $O(n)\ total, O(1)$ auxiliary

How many comparisons does the algorithm need to perform? How many swaps does the algorithm perform in the worst case?

15.4.3 Analysis

Selecting the lowest element requires scanning all n elements (this takes $n-1$ comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning all $n-1$ elements and so on, for $(n-1)+(n-2)+...+2+1(O(n^2))$ comparisons. Each of these scans requires one swap for $n-1$ elements.

15.5 Insertion Sort

Insertion sort is a comparison sort in which the sorted array (or list) is built one entry at a time. It is much less efficient on large lists than more advanced algorithms such as *quicksort*, *heapsort*, or *merge sort*. However, insertion sort provides several advantages:

- Simple implementation
- Efficient for small data sets
- Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is $O(n+d)$, where d is the number of inversions
- More efficient in practice than most other simple quadratic algorithms such as selection sort or bubble sort: the average running time is $n^2/4$, and the running time is linear in the best case
- Stable, i.e., does not change the relative order of elements with equal keys
- In-place, i.e., only requires a constant amount $O(1)$ of additional memory space
- Online, i.e., can sort a list as it receives it

15.5.1 Algorithm

Every iteration of insertion sort removes an element from the input data, inserting it into the correct position in the already-sorted list, until no input elements remain. The choice of which element to remove from the input is arbitrary, and can be made using almost any choice algorithm.

Sorting is typically done in-place. The resulting array after k iterations has the property where the first k+1 entries are sorted. In each iteration the first remaining entry of the input is removed, inserted into the result at the correct position, thus extending the result.

```
1 void insertionSort(int array[], int size) {
2   for (int step = 1; step < size; step++) {
3     int key = array[step];
4     int j = step - 1;
5
6     // Compare key with each element on the left of it until an element smaller than
7     // it is found.
```

```
    // For descending order, change key<array[j] to key>array[j].
    while (key < array[j] && j >= 0) {
      array[j + 1] = array[j];
      --j;
    }
    array[j + 1] = key;
  }
}
```

15.5.2 Performance

- Worst case performance: $O(n^2)$
- Best case performance: $O(n)$
- Average case performance: $O(n^2)$
- Worst case space complexity: $O(n)$ total, $O(1)$ auxiliary

15.6 A Comparison of N Squared Algorithms

Among simple average-case $O(n^2)$ algorithms, selection sort almost always outperforms bubble sort, but is generally outperformed by insertion sort.

Insertion sort's advantage is that it only scans as many elements as it needs in order to place the $k + 1^{st}$ element, while selection sort must scan all remaining elements to find the $k + 1^{st}$ element. Experiments show that insertion sort usually performs about half as many comparisons as selection sort. Selection sort will perform identically regardless of the order the array, while insertion sort's running time can vary considerably. Insertion sort runs much more efficiently if the array is already sorted or "close to sorted."

Selection sort always performs $O(n)$ swaps, while insertion sort performs $O(n^2)$ swaps in the average and worst case. Selection sort is preferable if writing to memory is significantly more expensive than reading.

Insertion sort or selection sort are both typically faster for small arrays (i.e., fewer than 10-20 elements). A useful optimization in practice for the recursive algorithms is to switch to insertion sort or selection sort for "small enough" sub-arrays.

15.7 Merge Sort

Name: MergeSort
Invented: 1945
Invented by: John von Neumann
Class: Sorting Algorithm
Stable: Yes

Complexity

	Best	Average	Worst
Space	$O(n)$	$O(n)$	$O(n)$
Cost	$O(n \log n)$	$O(n \log n)$	$O(n \lg n)$

Merge sort is an $O(n \log n)$ comparison-based sorting algorithm. It is an example of the divide and conquer algorithmic paradigm.

15.7.1 Algorithm

Conceptually, a merge sort works as follows:

- If the list is of length 0 or 1, then it is already sorted.
- Otherwise:
  - Divide the unsorted list into two sublists of about half the size.
  - Sort each sublist recursively by re-applying merge sort.
  - Merge the two sublists back into one sorted list.

Merge sort incorporates two main ideas to improve its runtime:

1. A small list will take fewer steps to sort than a large list.
2. Fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists. For example, you only have to traverse each list once if they're already sorted.

```
1  // Merges two subarrays of arr[].
2  // First subarray is arr[l..m]
3  // Second subarray is arr[m+1..r]
4  void merge(int arr[], int l, int m, int r)
5  {
6      int n1 = m - l + 1;
7      int n2 = r - m;
8
9      // Create temp arrays
10     int L[n1], R[n2];
```

```
11
12      // Copy data to temp arrays L[] and R[]
13      for (int i = 0; i < n1; i++)
14          L[i] = arr[l + i];
15      for (int j = 0; j < n2; j++)
16          R[j] = arr[m + 1 + j];
17
18      // Merge the temp arrays back into arr[l..r]
19
20      // Initial index of first subarray
21      int i = 0;
22
23      // Initial index of second subarray
24      int j = 0;
25
26      // Initial index of merged subarray
27      int k = l;
28
29      while (i < n1 && j < n2) {
30          if (L[i] <= R[j]) {
31              arr[k] = L[i];
32              i++;
33          }
34          else {
35              arr[k] = R[j];
36              j++;
37          }
38          k++;
39      }
40
41      // Copy the remaining elements of
42      // L[], if there are any
43      while (i < n1) {
44          arr[k] = L[i];
45          i++;
46          k++;
47      }
48
49      // Copy the remaining elements of
50      // R[], if there are any
51      while (j < n2) {
52          arr[k] = R[j];
53          j++;
54          k++;
55      }
56  }
57
```

```
58 void mergeSort(int arr[],int l,int r){
59     if(l>=r){
60         return;//returns recursively
61     }
62     int m =l+ (r-l)/2;
63     mergeSort(arr,l,m);
64     mergeSort(arr,m+1,r);
65     merge(arr,l,m,r);
66 }
```

Source: https://www.geeksforgeeks.org/merge-sort/

15.7.2 Performance

Worst case performance: $O(n\ log\ n)$ Best case performance: $O(n\ log\ n)$ typical Average case performance: $O(n\ log\ n)$ Worst case space complexity: $O(n)$ total,$O(n)$ auxiliary

15.8 Quicksort

Name: Quicksort
Invented: 1959
Invented by: Tony Hoare
Class: Sorting Algorithm
Stable: No

Complexity

	Best	Average	Worst
Space	$O(log\ n)$	$O(log\ n)$	$O(n)$
Cost	$O(n\ log\ n)$	$O(n\ log\ n)$	$O(n^2)$

Quicksort is a well-known sorting algorithm that, on average, makes$O(n\ log\ n)$ comparisons to sort n items. However, in the worst case, it makes $O(n^2)$ comparisons. Typically, quicksort is significantly faster than other $O(n\ log\ n)$ algorithms, because its inner loop can be efficiently implemented on most architectures, and in most real-world data, it is possible to make design choices which minimize the probability of requiring quadratic time

Quicksort is a comparison sort and, in efficient implementations, is not a stable sort.

15.8.1 Algorithm

Quicksort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.

The steps are:

1. Pick an element, called the pivot, from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements. The base case of the recursion are lists of size zero or one, which are always sorted

```
/* The main function that implements QuickSort
 arr[] --> Array to be sorted,
  low  --> Starting index,
  high  --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
```

```
            at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Source: https://www.geeksforgeeks.org/cpp-program-for-quicksort/

Quicksort is similar to merge sort in many ways. It divides the elements to be sorted into two groups, sorts the two groups by recursive calls, and combines the two sorted groups into a single array of sorted values. However, the method for dividing the array in half is much more sophisticated than the simple method we used for merge sort. On the other hand, the method for combining these two groups of sorted elements is trivial compared to the method used in mergesort.

The correctness of the partition algorithm is based on the following two arguments:

1. At each iteration, all the elements processed so far are in the desired position: before the pivot if less than or equal to the pivot's value, after the pivot otherwise.
2. Each iteration leaves one fewer element to be processed.

The disadvantage of the simple version above is that it requires O(n) extra storage space, which is as bad as merge sort. The additional memory allocations required can also drastically impact speed and cache performance in practical implementations. There is a more complex version which uses an in-place partition algorithm and use much less space.

The partition code:

```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high];    // pivot
    int i = (low - 1);  // Index of smaller element

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
```

```
15          if (arr[j] <= pivot)
16          {
17              i++;    // increment index of smaller element
18              swap(&arr[i], &arr[j]);
19          }
20      }
21      swap(&arr[i + 1], &arr[high]);
22      return (i + 1);
23  }
```

15.8.2 Choosing a Good Pivot Element

The choice of a good pivot element is critical to the efficiency of the quicksort algorithm. If we can ensure that the pivot element is near the median of the array values, then quicksort is very efficient. There are a few techniques to choose a good pivot element, but remember that the only guaranteed choice is to calculate the median value and this is expensive.

1. Choose the median value in the array.
   - Will give the best pivot point.
   - Lots of overhead calculating the median value defeating the speed of 'quicksort' .
2. Choosing the first element or the last element in the array.
   - Easy to implement.
   - Bad on partially sorted data.
3. Choosing the middle (not the median) element of the array.
   - Also easy to implement.
   - Better on partially sorted data.
4. Choose three values from the array and then use the middle of these three values as the pivot element. Those three can be randomly chosen or use the first, middle, and last elements in the array. Choosing three random values is preferred and is called the **median-of-3 method** . It's preferred because it's a more robust solution when you do not know what kind of data you are sorting (totally random, partially ordered, already ordered).
   - Pretty easy to implement.
   - Should gives a decent approximation (on average) of what the median value may be.

16 Heapsort

Name: Heapsort
Invented: 1964
Invented by: J. W. J. Williams
Class: Sorting Algorithm
Stable: No

Complexity

	Best	Average	Worst
Space	$O(1)$	$O(1)$	$O(1)$
Cost	$O(n \log n)$	$O(n \log n)$	$O(n \lg n)$

Heapsort is a comparison-based sorting algorithm, and is part of the selection sort family. Although somewhat slower in practice on most machines than a good implementation of quicksort, it has the advantage of a worst-case O(n log n) runtime. Heapsort combines the time efficiency of merge sort and the storage efficiency of quicksort.

Heapsort is similar to selection sort in that it locates the largest value and places it in the final array position. Then it locates the next largest value and places it in the next-to-last array position and so forth. However, heapsort uses a much more efficient algorithm to locate the array values to be moved.

16.1 How it works?

The heapsort begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap and removes the largest remaining item and places it in the next open position from the end of the sorted array.

This is repeated until there are no items left in the heap and sorted array is full. Elementary implementations require two arrays – one to hold the heap and the other to hold the sorted elements. In advanced implementation however, we have an efficient method for representing a heap (complete binary tree) in an array and thus do not need an extra data structure to hold the heap.

17 Source Code

17.1 Graph Algorithms

- Dijkstra's Algorithm
- Breadth First Search
- Connected Components

17.2 Hashing

- Chaining
- Linear Probing
- Quadratic Probing
- Double Hashing

18 Trie's

Name: Trie (prefix tree)
Invented: 1959-1960
Invented by: René de la Briandais and Edward Fredkin (independantly)
Structure / Container: Linked Structure

Complexity

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Tries were first described in a computer context by René de la Briandais in 1959. The idea was independently described in 1960 by Edward Fredkin,who coined the term *trie*, pronouncing it /tri/ (as in "tree"), after the middle syllable of retrieval. But... NO! say /tri/ (as in "try"). It's to confusing to say "tree". The complexity of creating a trie is $O(W * L)$, where **W** is the number of words, and **L** is an average length of the word: you need to perform **L** lookups on the average for each of the **W** words in the set. Same goes for looking up words later: you perform **L** steps for each of the **W** words. Hash insertions and lookups have the same complexity: for each word you need to check equality, which takes $O(L)$, for the overall complexity of $O(W * L)$.

In computer science, a **Trie**, is also called *digital tree* or *prefix tree*, and is a type of search tree. It is a tree data structure used for locating specific keys from within a set [11]. You can see an example (an extremely small one) in figure 19. Following any path from the root down will start to build either a *word* or the *prefix* to a word. You know that it is a word if there is a green check mark on the node.

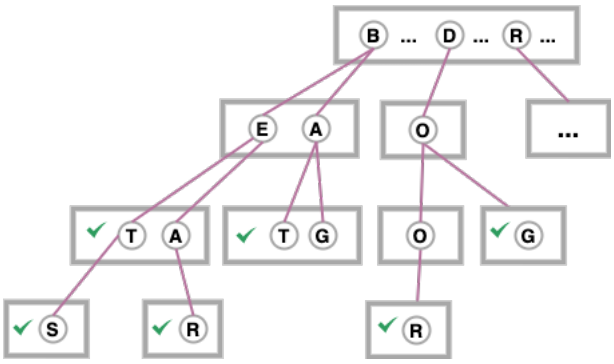


Figure 19: Example Trie

Storing data in a tree structure in this manner lets us keep similar words organized and in the same path. That's why this data structure is also called a "Prefix Tree". There are more space efficient methods of storing words in a prefix like manner as well. One of those is a Radix Tree.

18.1 TrieNode

The base of every data structure, is the datatype in which you store your data. Figure 21 shows us what the internal structure is for a TrieNode looks like.

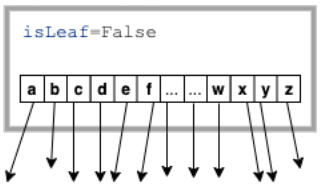


Figure 20: Trie Node

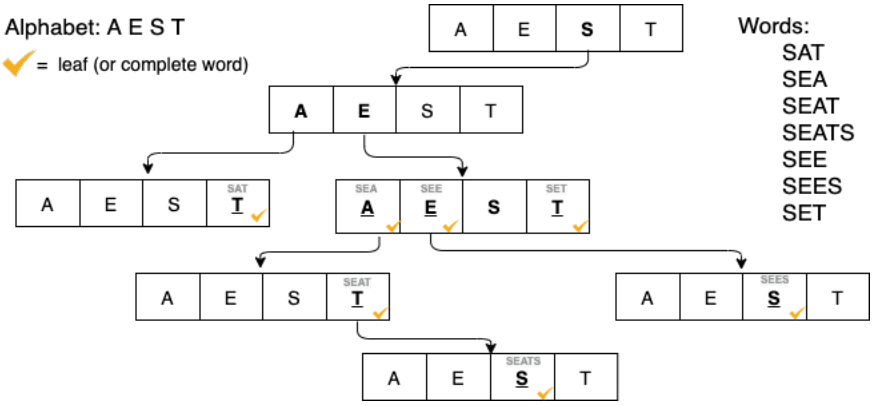


Figure 21: Trie Tree using a minimal alphabet and subset of words

It simply has a boolean variable called **isWord** and an array of TrieNode pointers. The isWord variable tells a search method if "this" node constitutes the end of a word. There could be more letters beyond this node constituting longer words. For example the three letters "cat" is the first three letters of "**cat**alogue" or "**cat**tle". I will talk about this more in the section discussing the search method. The code for a TrieNode is below:

```
1  struct TrieNode{
2      bool isWord;            // is this node end of a word
3      TrieNode **children;    // letters further down prefix
4      TrieNode(){
5          children = new TrieNode*[26];
6          isWord = false;
7          for (int i = 0; i < 26; i++) {
8              children[i] = nullptr;
9          }
10     }
11 };
```

18.2 Insertion

18.3 complexity

Creation - O(N*L) Insertion = O(L) Search = O(L) Deletion = O(L)

19 Uploading to UVA Online Judge

Adapted from Zachary Kingcades tutorial.

19.1 Overview

There is a little different angle we must take when writing code to be uploaded to https://onlinejudge.org/. Just like I have stressed in class (oh remember the days when we had class . . .) having a base knowledge of the use of the command line can be beneficial. This tutorial will discuss a couple of those skills and prepare you for uploading code to UVA Online judge.
Keep in mind that solving these types of problems isn't just for competition, its relevant for school and especially as a prep for interviewing.

19.1.1 Registration and Overview

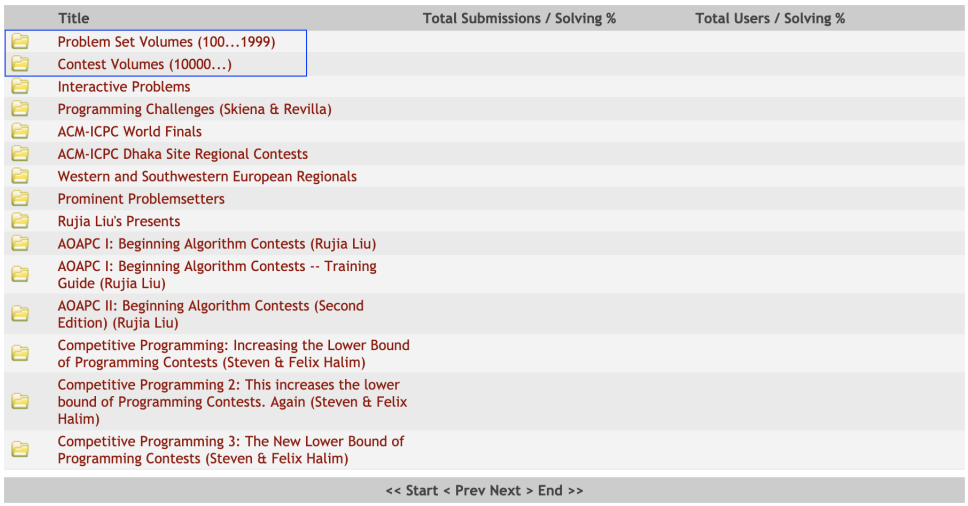
Onlinejudge has thousands of problems to browse and solve. You can look at them without registering, but you will need to register to submit solutions so go ahead and do that HERE.
The major portion of Onlinejudge is its uHunt section. This is where you browse problems, submit your solutions, see statistics about each problem, see latest submissions, and most importantly where you see the status of your own submissions. You can click on the link I provided, **or** look for this icon on the main page:

19.1.2 Selecting A Problem

You're registered, and ready to solve problems! How do you pick a problem? There are many different ways and reasons to select a problem.
We are solving problems that require the understanding of specific data structures as well as problem solving paradigms. Data structures include: lists, arrays, stacks, queues, trees, graphs along with variants or combos of each. Problem Solving Paradigms include: brute force, divide and conquer, greedy, and dynamic programming.
We will use Competitive Programming 3 to help us choose problems that will best be solved by choosing the proper data structure accompanied with the correct problem solving paradigm.

> **Note 1:** I hate stealing from any author, especially one who created such a great resource. I'm sure he would understand our dilemma this semester. I bought my own copy, and would encourage all of you to purchase a copy. Its a great programming resource since it really concentrates on problem solving in general, not just competitive programming.
>
> **Note 2:** The only reason I haven't used this from the beginning is . . . cheating :(You can find most of the solutions online. Even though I have a defense for this, its just exhausting. More later . . .

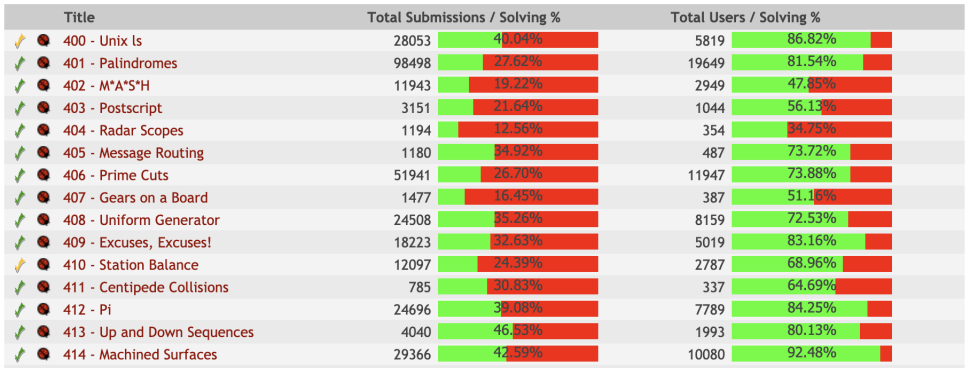
You don't only have to choose problems based on data structures and problem solving paradigms, believe it or not, people actually solve these for fun! I would recommend you solve as many of these as you have time for in addition to your course work. Lots of solutions will pad your resume' AND prepare you for interviews. So how do you choose problems to solve if its not for specific data structures? You can choose based on difficulty and how cool the name is. To do this, you have to browse the problem sets. Below is the first menu of problems to choose from. We will concentrate on the two sets I placed a box around:



Title	Total Submissions / Solving %	Total Users / Solving %
Problem Set Volumes (100...1999)		
Contest Volumes (10000...)		
Interactive Problems		
Programming Challenges (Skiena & Revilla)		
ACM-ICPC World Finals		
ACM-ICPC Dhaka Site Regional Contests		
Western and Southwestern European Regionals		
Prominent Problemsetters		
Rujia Liu's Presents		
AOAPC I: Beginning Algorithm Contests (Rujia Liu)		
AOAPC I: Beginning Algorithm Contests -- Training Guide (Rujia Liu)		
AOAPC II: Beginning Algorithm Contests (Second Edition) (Rujia Liu)		
Competitive Programming: Increasing the Lower Bound of Programming Contests (Steven & Felix Halim)		
Competitive Programming 2: This increases the lower bound of Programming Contests. Again (Steven & Felix Halim)		
Competitive Programming 3: The New Lower Bound of Programming Contests (Steven & Felix Halim)		

<< Start < Prev Next > End >>

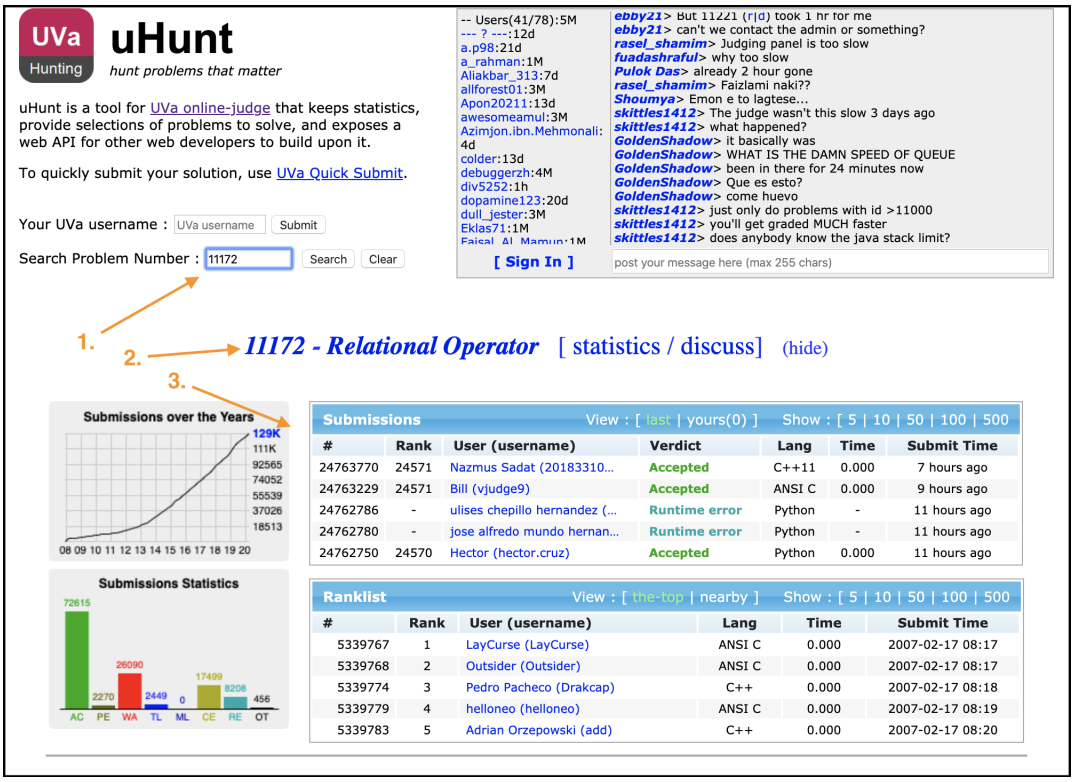
As you click folders, they drill down into other folders and finally lists of problems. If you drill down into any of the folders you will end up seeing something similar to below:



Title	Total Submissions / Solving %	Total Users / Solving %
400 - Unix ls	28053 40.04%	5819 86.82%
401 - Palindromes	98498 27.62%	19649 81.54%
402 - M*A*S*H	11943 19.22%	2949 47.85%
403 - Postscript	3151 21.64%	1044 56.13%
404 - Radar Scopes	1194 12.56%	354 34.75%
405 - Message Routing	1180 34.92%	487 73.72%
406 - Prime Cuts	51941 26.70%	11947 73.88%
407 - Gears on a Board	1477 16.45%	387 51.16%
408 - Uniform Generator	24508 35.26%	8159 72.53%
409 - Excuses, Excuses!	18223 32.63%	5019 83.16%
410 - Station Balance	12097 24.39%	2787 68.96%
411 - Centipede Collisions	785 30.83%	337 64.69%
412 - Pi	24696 39.08%	7789 84.25%
413 - Up and Down Sequences	4040 46.53%	1993 80.13%
414 - Machined Surfaces	29366 42.59%	10080 92.48%

We can see in a brief glance that lots of "red" = hard, and lots of "green" = easy (generally). The left column shows how many total submissions vs how many solved. In other words, lots of red means it takes many submissions before someone gets it correct. The right column is percentage of people that attempted the problem actually solved it. So, if you want an easy problem, find one that has a left column with 90+ solution percentage.

If you need to find a specific problem, they all have numbers (the ones we will use). So you can go browse and find problems using the number to navigate folders, or you can go to uHunt and put the number in a search form (see #1 on image).



This will bring up a link to the problem description (see #2 on image) which you can download and keep as a reference as your solving. It also give you basic stats and current “whats going on” with the problem (see #3 on image).
To stay organized I typically create a folder with the problem number as the folder name, and save everything from that associated problem in there. There will always be the pdf for the problem, multiple data sets to test your problem, and the actual source code. So, organizing with folders and problem numbers is recommended.
Aside from being organized and trying to decide what problems to solve, you really just need to break the ice and get a solution uploaded to uHunt. So lets start with a super easy one called `11172 - Relational Operator`

19.1.3 Solving A Problem

You can refer to the previous section and go get the pdf of the problem we will solve. Its number is "11172". But for this tutorial, here are the contents:

19.1.4 11172 - Relational Operator

Time limit: 3.000 seconds

Some operators checks about the relationship between two values and these operators are called relational operators. Given two numerical values your job is just to find out the relationship between them that is (i) First one is greater than the second (ii) First one is less than the second or (iii) First and second one is equal.

Input First line of the input file is an integer `t (t < 15)` which denotes how many sets of inputs are there. Each of the next `t` lines contain two integers `a` and `b (|a|, |b| < 1000000001)`.

Output For each line of input produce one line of output. This line contains any one of the relational operators >, < or =, which indicates the relation that is appropriate for the given two numbers.

Sample Input

```
3
10 20
20 10
10 10
```

Sample Output

```
<
>
=
```

19.1.5 Coding The Problem

Start You should read over the problem and start to formulate a solution by breaking it down into manageable components (more later). This problem is easy. I will say, however, that initially none of the problems **read** easy. They use mathematical notation to remove ambiguity from the problem definition, and it is a bit confounding at first. However it simply requires a little practice or a nice **slow** approach to absorb its meaning.

Think For each problem we recommend that you THINK about it, re-read it, then start your solution on paper. Also, when solving these problems, always look at the limits as described in the problem (e.g. `t < 15` and `(|a|, |b| < 1000000001)`). These could have an impact on the solution. For example, does `1000000001` fit in an integer data type? Or should it be long? Int works when uploaded for this solution, but on a tougher problem this could bite you in the butt.

Code After you come up with a proposed solution, start to code it. Normal coding conventions that we stress in class are a little out the window. **Not all**, but some. Rules like self documenting variable names, comments for each variable, organized procedural solution (lots of functions) are all relaxed for these solutions.

Nuances Also, we want all the little speedups we can manage. **Example 1)** in larger solutions where a function seems appropriate, you may want to keep your code in main (inline), a function call is ever so slightly slower than inline code. **Example 2)** Another miniscule speedup is something as simple as re-defining `endl` to `\n`. The first one is interpreted as a newline **AND** a flush of the output buffer. If one of the problems you are solving has you writing to standard out thousands

and thousands of times, the extra flush may slow you down. Both of these are tiny little speedups, but put a lot of those tiny speedups together, and it could make a difference. Difference in what you ask? The first line after the title in our problem statement states: `Time limit: 3.000 seconds`. That's the problem. Each solution has a time limit to complete.

Stdin Lastly **cin** is NOT being used to prompt a user for input. It is being used to read from an input file! Jump below the code and input file . . .

```
#include <iostream>

#define endl "\n"

using namespace std;

int  main(){
    int c,l,r,i=0;
    cin>>c;
     while(i<c){
         cin>>l>>r;
         if(l<r){
             cout<<'<'<<endl;
         }else if(l>r){
             cout<<'>'<<endl;
         }else{
             cout<<'='<<endl;
         }
         i++;
     }
     return 0;
}
```

infile

```
3
10 20
20 10
10 10
```

cin reads from stdin which means a default connected stream for whatever platform you are on. Typically when you run a program, and you need input, you pause execution of the program with a `cin` and then it waits for input from the "keyboard". And when you write using `cout` it writes to `stdout` which is usually the terminal (or dos window). As a side note there is also `stderr` which is where errors are directed to (we don't deal with stderr, as its usually used with log files and such). We change `stdin` to be input from a file. Why? If you upload your code to be auto run, how do you process a file? You could be given a path and a file name, but its actually easier to redirect a file into your program.
Given:

- code is in `main.cpp`

- we compile and its executable is now `main.exe`
- we have an input file called `infile`

We can read `infile` using `cin` by doing:

```
./main.exe < infile
```

The command above says to take `infile` and send it into `main.exe` as if we opened it, except we can use `cin` to read from it. Basically, we changed `stdin` to be a file, instead of the keyboard.
If you refer back to the code snippet, and infile, - line 9 reads in the 3, telling us how many "pairs" of values we need to read. - line 11 then reads in two values for every iteration of the loop

19.1.6 Run in Visual Studio :(

You have to edit the projects properties. So whichever side your solution explorer is:

- Right Click on the Project and select Properties.
- Project -> Properties -> Debugging -> Command Arguments
- Add the following to your Command Arguments: `< input.txt`
- Or something like that

19.1.7 Run in Terminal

A better way is to use command line and a bash console like `gitbash`.

- Change into your folder with your code and input file.
- Compile:
  - `g++ <filename> -o <executablename>`
  - `g++ main.cpp -o main`
- Run:
  - `./main < inputfile`

If you use any of the c++11 or later code constructs you will need to set the compiler standard when compiling:

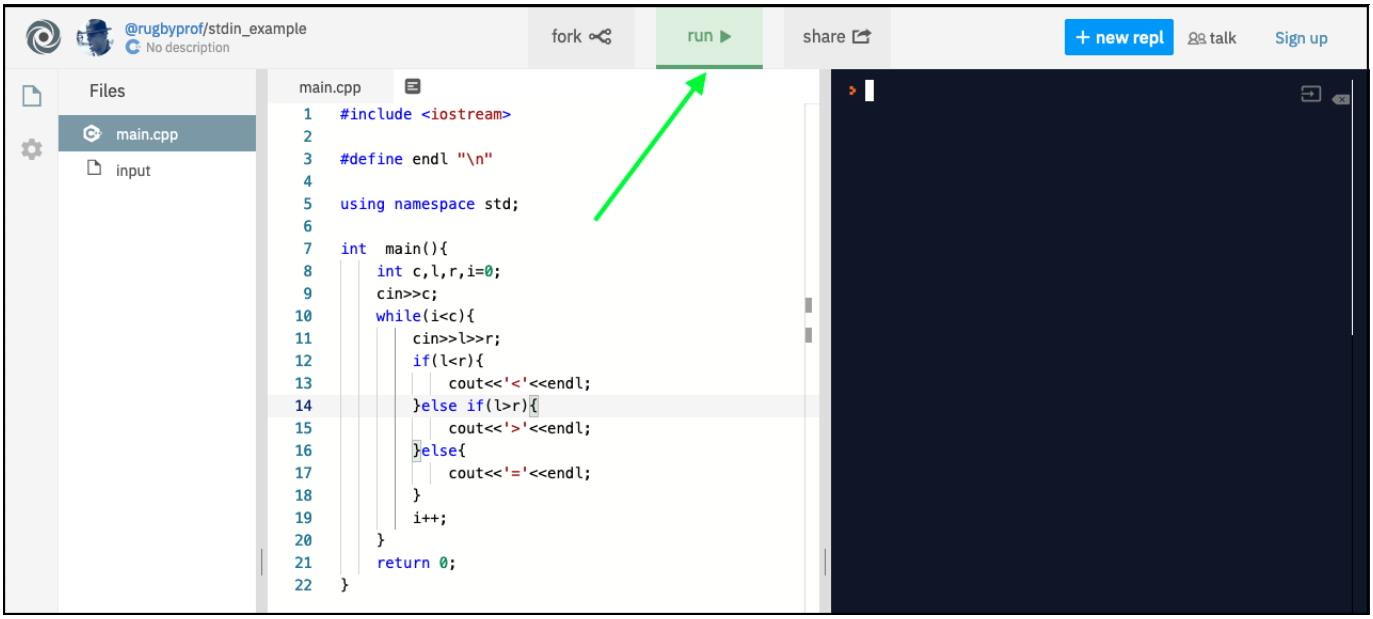
- Compile:
  - `g++ -std=c++17 main.cpp -o main`
  - or replace c++17 with c++11 if thats what you need

19.1.8 Repl.it

Repl.it is kinda awesome. Yes I said it. They give us a pretty nice virtual linux environment to run our programs in. Basically, they give us a virtual machine / command line solution . . . in the browser. I have the example from above here: https://repl.it/@rugbyprof/stdinexample. Even if you don't have an account it will let you run and edit that code. I would recommend creating an account. Here is a tutorial on creating and using repl.it: https://cs.msutexas.edu/replit_tutorial/

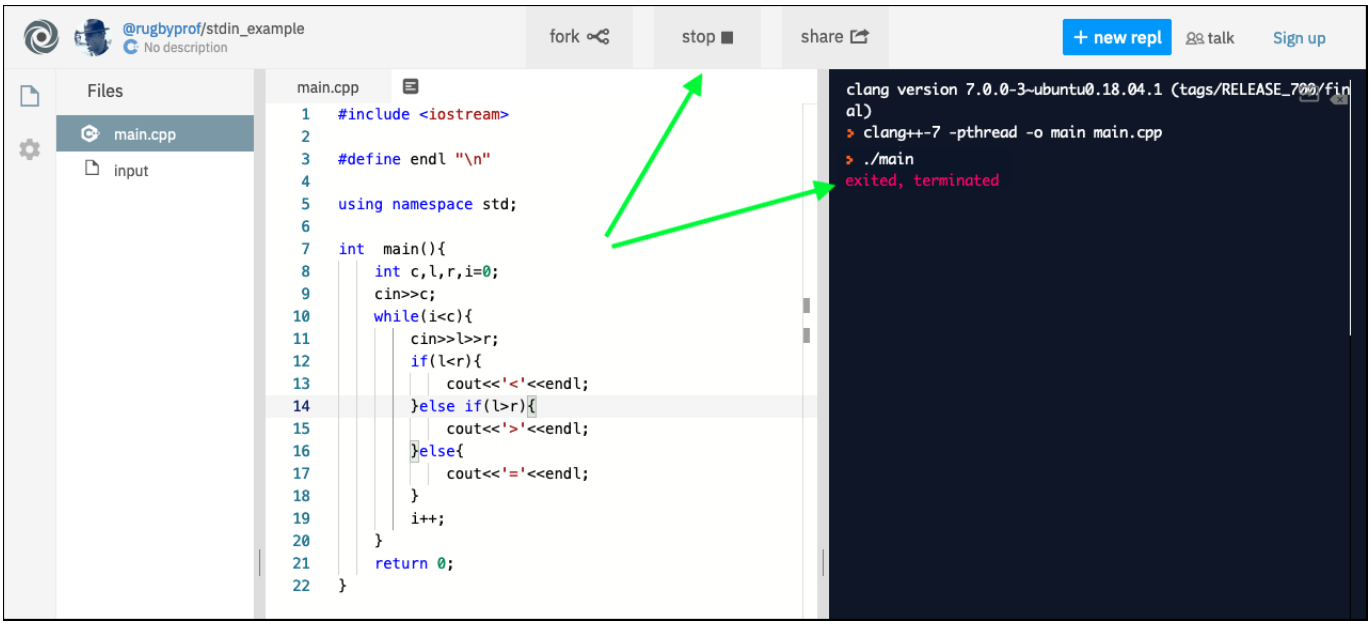
Start

- Click run so that it “compiles” first time.
- You will have to stop it (see next pic)



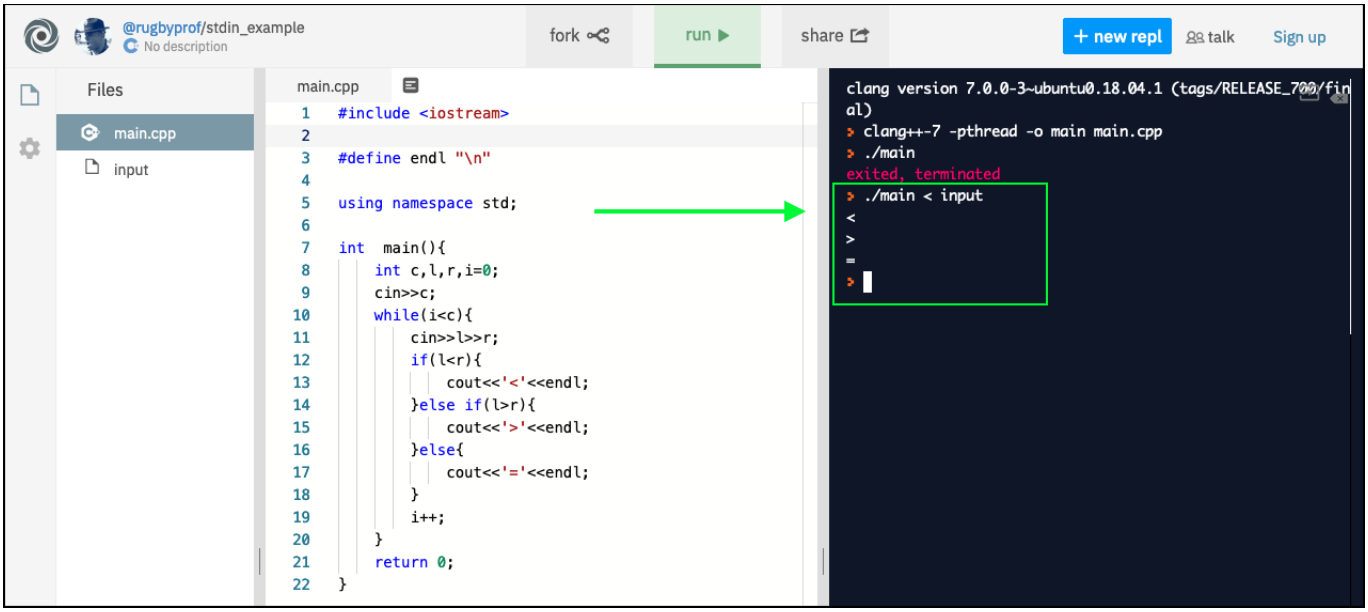
Stop

- It will not finish, since its waiting for input:)
- `cin` is still defined to read keyboard input, so hit the stop button.
- You should see `exited, terminated` in the terminal.



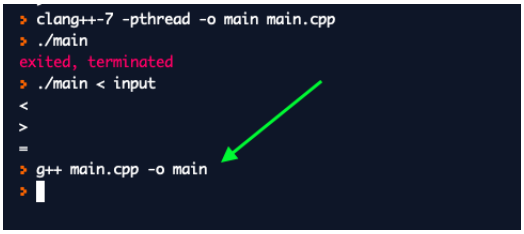
Test Run

- When we hit “run” the first time , Replit compiled our code and made an executable with the same name as our file (minus the extension).
- Since Replit is a linux virtual machine, we can run our program from the provided command line!
- So, you can type `./main < input` in the terminal and get a correct run (notice we have output in the green box).



Compile

- We don’t want to have to hit “run” to get a new executable every time we make a change to our source code. And we don’t have to.
- Again, Replit gives us a linux terminal, so let us compile our own code using the terminal they so nicely gave us.
- So as you are testing your code, you can use the `up arrow` to cycle through the necessary commands (after you have typed them at least once).
  - Make a change, then using the terminal:
  - Compile: `g++ main.cpp -o main`
  - Run: `./main < input`



19.1.9 Testing Your Solution

You can now run your code using an input file as stdin. But how do we know if the solution is correct? You know its correct if:

- You thought of every edge case possible.
- You incorporated all nuances of the problem description into your solution.
- Your output is formatted perfectly.
- You are smarter than anyone you know.

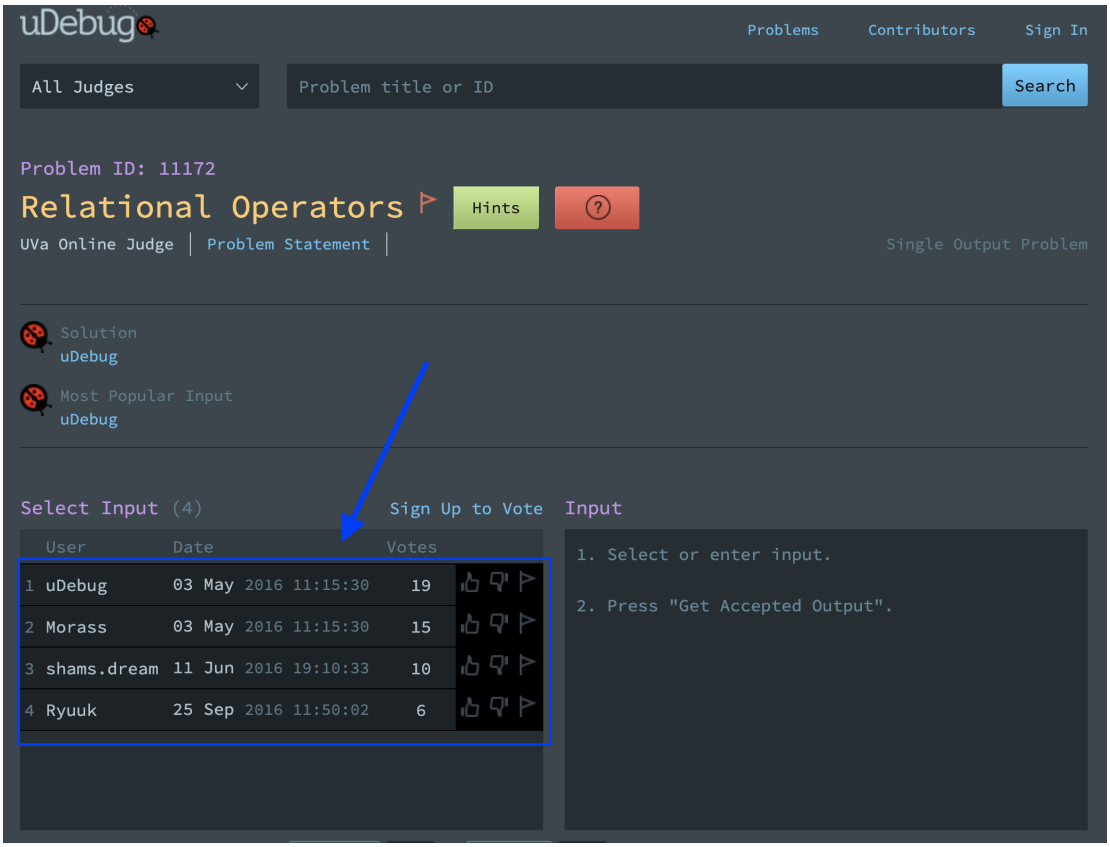
The problem is, you really do not know if your solution is correct. You can upload it to uHunt and roll the dice to see if your solution will be accepted. Or, you can try to ensure your solution is correct by running multiple test data sets.

If you go to the problems page where you got the pdf from, you will see a lady bug at the top right above the problem statement. Click on it. It will take you from uHunt to uDebug where you have some options for testing your solution more thoroughly using data sets uploaded by other users. They go way beyond the example data given in the problem, and will help you determine if you really did thing about all the "edge cases" that might appear in the data.

Go Debug

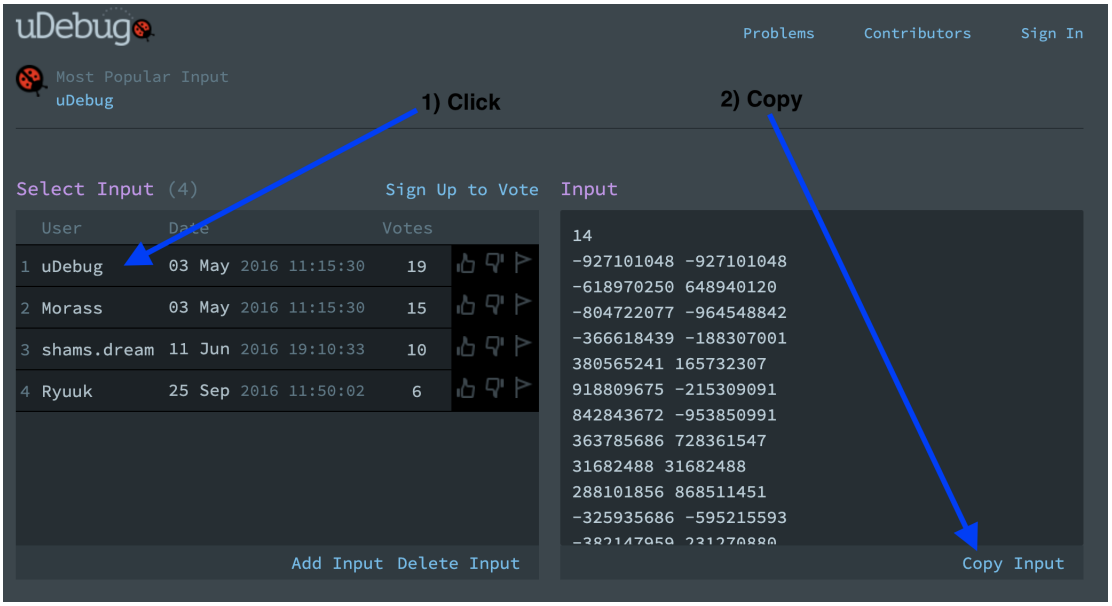
Getting Data

- When you go to the debug site for your problem, you will see multiple data sets to choose from as I enclosed in a box.
- Above the blue box, you will see a link to the most popular input data. But all of them should help you debug.
- If your code can process all of the data sets, you should be good (but not always :) I have one solution that passed ALL data sets but fails online.)



Pick a Set

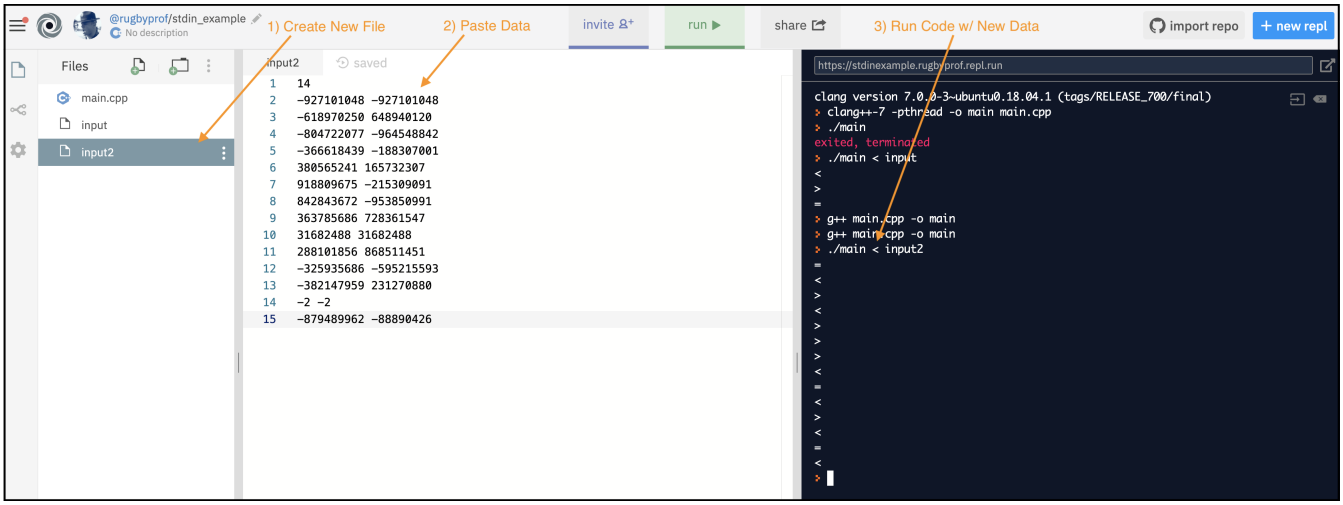
1. Pick a data set
2. Copy it to your clipboard



Use New Data

1. Create a new file
2. Paste new data in file
3. Run your code with new data file

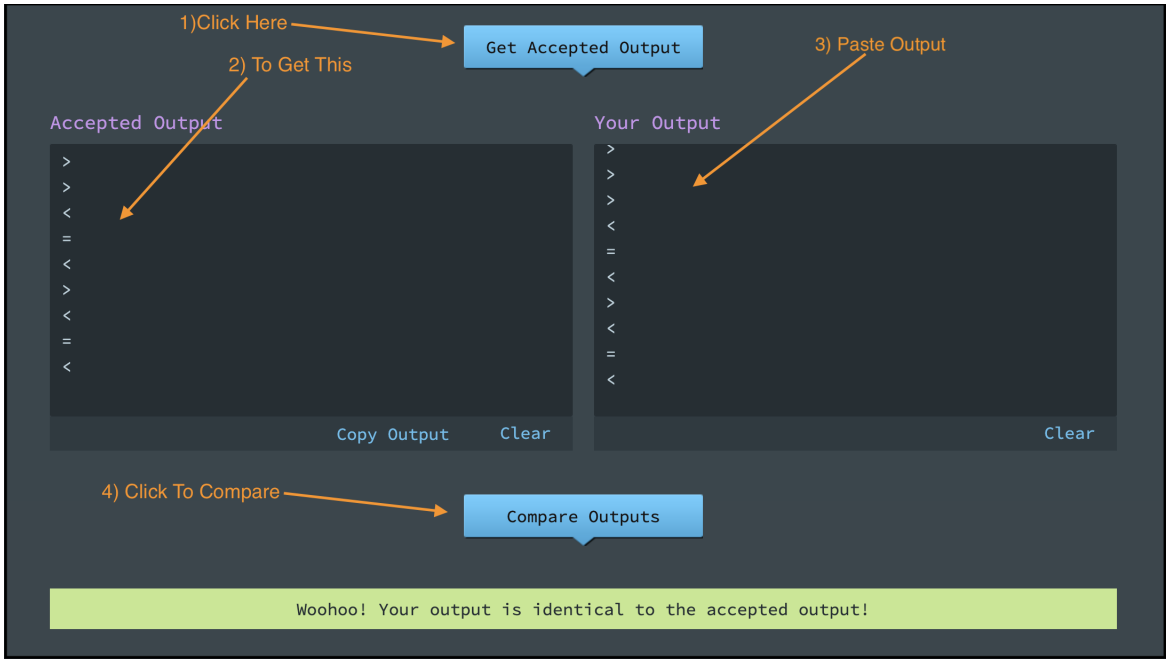
   Your new output will be in the terminal, you need to copy it to bring back to uDebug.



Check New Output

1. Click to get accepted output.
2. This window shows whats expected.
3. This is where you paste your output to be compared with accepted output.
4. Click to see if it matches!

   Note: uDebug gives better feedback than simply uploading your solution. In fact lots of times solutions are rejected for an extra “newline”. This can be fixed here at uDebug.



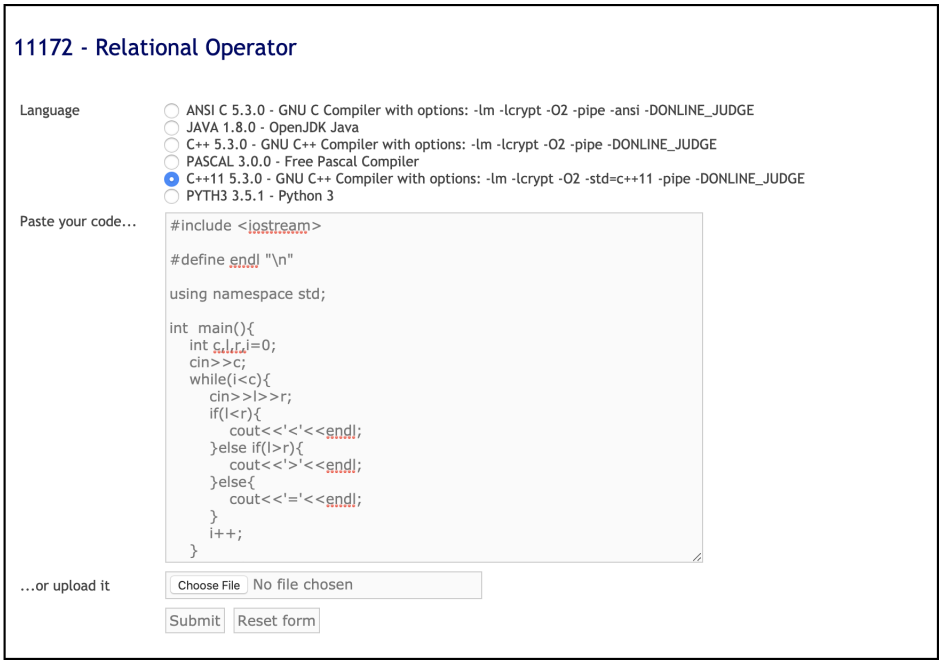
19.1.10 Uploading Solution

Start Submission

- Go back to the problem page and click on submit.

Do Submission

- On the next page:
  - Choose your version of c++
  - Paste your code into the submission text field or Upload a file.
  - Press “submit”

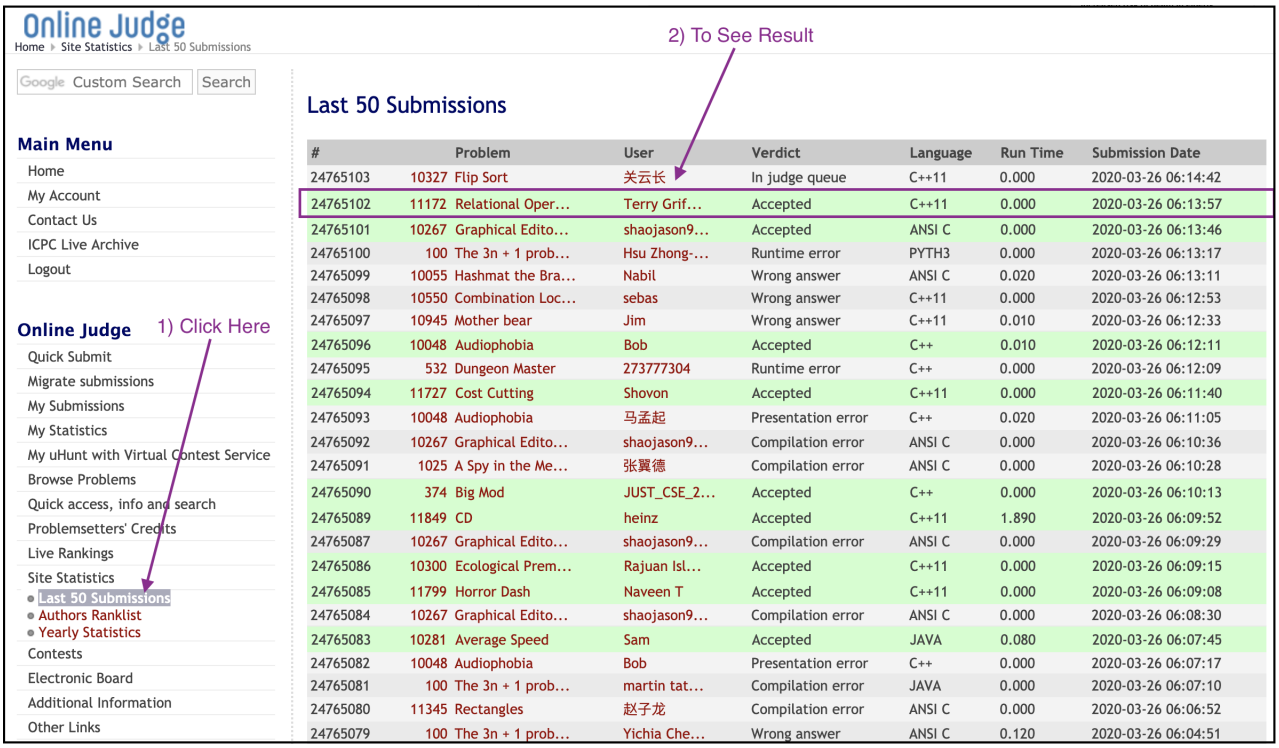


Confirm Submission

- You Get redirected back to the problem page with a confirmation.

Submission Results

- You will get an email telling you what happened, but I like to view the stats on the website.
- On the main onlinejudge site.
- Click on last 50 submissions (under site statistics).
- You should see your results (if you don't wait too long).



#		Problem	User	Verdict	Language	Run Time	Submission Date
24765103	10327	Flip Sort	关云长	In judge queue	C++11	0.000	2020-03-26 06:14:42
24765102	11172	Relational Oper...	Terry Grif...	Accepted	C++11	0.000	2020-03-26 06:13:57
24765101	10267	Graphical Edito...	shaojason9...	Accepted	ANSI C	0.000	2020-03-26 06:13:46
24765100	100	The 3n + 1 prob...	Hsu Zhong-...	Runtime error	PYTH3	0.000	2020-03-26 06:13:17
24765099	10055	Hashmat the Bra...	Nabil	Wrong answer	ANSI C	0.020	2020-03-26 06:13:11
24765098	10550	Combination Loc...	sebas	Wrong answer	C++11	0.000	2020-03-26 06:12:53
24765097	10945	Mother bear	Jim	Wrong answer	C++11	0.010	2020-03-26 06:12:33
24765096	10048	Audiophobia	Bob	Accepted	C++	0.010	2020-03-26 06:12:11
24765095	532	Dungeon Master	273777304	Runtime error	C++	0.000	2020-03-26 06:12:09
24765094	11727	Cost Cutting	Shovon	Accepted	C++11	0.000	2020-03-26 06:11:40
24765093	10048	Audiophobia	马孟起	Presentation error	C++	0.020	2020-03-26 06:11:05
24765092	10267	Graphical Edito...	shaojason9...	Compilation error	ANSI C	0.000	2020-03-26 06:10:36
24765091	1025	A Spy in the Me...	张翼德	Compilation error	ANSI C	0.000	2020-03-26 06:10:28
24765090	374	Big Mod	JUST_CSE_2...	Accepted	C++	0.000	2020-03-26 06:10:13
24765089	11849	CD	heinz	Accepted	C++11	1.890	2020-03-26 06:09:52
24765087	10267	Graphical Edito...	shaojason9...	Compilation error	ANSI C	0.000	2020-03-26 06:09:29
24765086	10300	Ecological Prem...	Rajuan Isl...	Accepted	C++11	0.000	2020-03-26 06:09:15
24765085	11799	Horror Dash	Naveen T	Accepted	C++11	0.000	2020-03-26 06:09:08
24765084	10267	Graphical Edito...	shaojason9...	Compilation error	ANSI C	0.000	2020-03-26 06:08:30
24765083	10281	Average Speed	Sam	Accepted	JAVA	0.080	2020-03-26 06:07:45
24765082	10048	Audiophobia	Bob	Presentation error	C++	0.000	2020-03-26 06:07:17
24765081	100	The 3n + 1 prob...	martin tat...	Compilation error	JAVA	0.000	2020-03-26 06:07:10
24765080	11345	Rectangles	赵子龙	Compilation error	ANSI C	0.000	2020-03-26 06:06:52
24765079	100	The 3n + 1 prob...	Yichia Che...	Wrong answer	ANSI C	0.120	2020-03-26 06:04:51

[heading=bibintoc]

References

[1] Avl tree - wikipedia. https://en.wikipedia.org/wiki/AVL_tree. (Accessed on 03/25/2021).

[2] Binary heap - wikipedia. https://en.wikipedia.org/wiki/Binary_heap. (Accessed on 01/22/2021).

[3] B-tree - wikipedia. https://en.wikipedia.org/wiki/B-tree. (Accessed on 01/24/2021).

[4] Hash table - wikipedia. https://en.wikipedia.org/wiki/Hash_table. (Accessed on 03/25/2021).

[5] Heap (data structure) - wikipedia. https://en.wikipedia.org/wiki/Heap_%28data_structure%29. (Accessed on 01/21/2021).

[6] Locality of reference - wikipedia. https://en.wikipedia.org/wiki/Locality_of_reference. (Accessed on 01/19/2021).

[7] Priority queue - wikipedia. https://en.wikipedia.org/wiki/Priority_queue. (Accessed on 01/22/2021).

[8] Recursion (computer science) - wikipedia. https://en.wikipedia.org/wiki/Recursion_(computer_science). (Accessed on 02/20/2021).

[9] R-tree - wikipedia. https://en.wikipedia.org/wiki/R-tree. (Accessed on 01/24/2021).

[10] Trie - wikipedia. https://en.wikipedia.org/wiki/Trie, . (Accessed on 01/24/2021).

[11] Trie - wikipedia. https://en.wikipedia.org/wiki/Trie, . (Accessed on 02/18/2021).

[12] Difference between array and linked list (with comparison chart) - tech differences. https://techdifferences.com/difference-between-array-and-linked-list.html. (Accessed on 01/04/2021).

[13] m-ary tree - wikipedia. https://en.wikipedia.org/wiki/M-ary_tree. (Accessed on 01/24/2021).

[14] Category:stable sorts - wikipedia. https://en.wikipedia.org/wiki/Category:Stable_sorts. (Accessed on 04/04/2021).

[15] Data structure and algorithms - avl trees - tutorialspoint. https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm. (Accessed on 01/06/2021).

[16] Algorithm - wikipedia. https://en.wikipedia.org/wiki/Algorithm, . (Accessed on 01/04/2021).

[17] Data structure - wikipedia. https://en.wikipedia.org/wiki/Data_structure, . (Accessed on 01/04/2021).

[18] D. E. Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

Glossary

A | B | C | H | I | L | M | P | R | T | U

A

acyclic In terms of a graph, acyclic means that there are no cycles in the graph. 41

B

b-tree In computer science, a B-tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree generalizes the binary search tree, allowing for nodes with more than two children. Unlike other self-balancing binary search trees, the B-tree is well suited for storage systems that read and write relatively large blocks of data, such as disks. It is commonly used in databases and file systems. 41

Big O Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation. The letter O is used because the growth rate of a function is also referred to as the order of the function. A description of a function in terms of big O notation usually only provides an upper bound on the growth rate of the function. 35

binary tree A binary tree is a tree data structure in which each node has at most two children. 46

C

complete In terms of a binary tree, All levels except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right. 50

H

heap data structure A type of data structure that efficiently implements a priority queue using a binary structure in an array. It can also be used to sort numbers. 49

heap memory A type of memory that is typically associated with dynamically allocated data. 49

heapsort Sorting values using a binary heap by inserting the values into a heap and then removing the items 1 by 1. 50

I

in order predecessor In a binary search tree, the in order predecessor of some node *N*, can also be defined as the node with the largest key smaller than the key of *N*. Or it is largest value in the left sub-tree of node *N*. 44

in order successor In a binary search tree, the in order successor of some node *N*, can also be defined as the node with the smallest key greater than the key of *N*. Or it is smallest value in the right sub-tree of node *N*. 44

iteration Iteration in computing is the technique marking out of a block of statements within a computer program for a defined number of repetitions. That block of statements is said to be iterated; a computer scientist might also refer to that block of statements as an "iteration". 21

L

lexicographical order Think of lexicographical order as alphabetical order. There are some variations, but in computer science we are almost always referring to the sorting of strings. 71

M

m-ary tree In graph theory, an m-ary tree (also known as k-ary or k-way tree) is a rooted tree in which each node has no more than m children. A binary tree is the special case where m = 2, and a ternary tree is another case with m = 3 that limits its children to three. 41, 42

P

parallel arrays Parallel arrays are a set of 2 or more arrays that store homogeneous data (data that is similar or related). For a simple example lets say we have three arrays: first_name, last_name, age. To know the first name, last name, and age of a single individual, I would simply access each array with the same index. 61

priority queue A queue like data structure in which an element with high priority gets preference (removed or served) before an element with lower priority. 49

R

r-tree R-trees are tree data structures used for spatial access methods, i.e., for indexing multi-dimensional information such as geographical coordinates, rectangles or polygons. The R-tree was proposed by Antonin Guttman in 1984 and has found significant use in both theoretical and applied contexts. A common real-world usage for an R-tree might be to store spatial objects such as restaurant locations or the polygons that typical maps are made of: streets, buildings, outlines of lakes, coastlines, etc. and then find answers quickly to queries such as "Find all museums within 2 km of my current location", "retrieve all road segments within 2 km of my location" (to display them in a navigation system) or "find the nearest gas station" (although not taking roads into account). The R-tree can also accelerate nearest neighbor search for various distance metrics, including great-circle distance. 41

Radix Tree In computer science, a radix tree (also radix trie or compact prefix tree) is a data structure that represents a space-optimized trie (prefix tree) in which each node that is the only child is merged with its parent. The result is that the number of children of every internal node is at most the radix r of the radix tree, where r is a positive integer and a power x of 2, having x 1. Unlike regular trees, edges can be labeled with sequences of elements as well as single elements. This makes radix trees much more efficient for small sets (especially if the strings are long) and for sets of strings that share long prefixes. 87

T

trie In computer science, a trie, also called digital tree or prefix tree, is a kind of search tree—an ordered tree data structure used to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated; i.e., the value of the key is distributed across the structure. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Keys tend to be associated with leaves, though some inner nodes may correspond to keys of interest. Hence, keys are not necessarily associated with every node. 41

U

undirected graph In a graph, G = (V, E), none of the edges have an "orientation" (no direction associated with the edge). This means that we can traverse an edge $A - B$ from $A \to B$ or from $B \to A$. Given a directed edge (one with orientation) and an edge $A \to B$, we can only traverse this edge in that direction, and **not** from $B \to A$. 41