(1) Algorithm Complexity

Why have all these different data structures?

- Ability to represent data in a meaningful way
- Performance

How do we actually quantify the performance?

Let's consider an example:

Say, it is 1995, Bill Gates just announced Windows 95. I have a Pentium II desktop and I am writing some C++ code to parse an array:

```
int N = 10; // 1 ms
int a[N];   // 1 ms

int n = 0; // 1 ms
for(int i; i<N; i++)
   a[i] = rand()%100;// 10 ms per iteration
```

Total time? 103 ms

What if N = 100 ? 1004 ms

What if N = 1e6 ? 10,000,004 ms

(2) Complexity 2

continuing from previous page:

The one thing we know about this algo is that no matter what system it is run on, it will increase in time as N goes up in time.

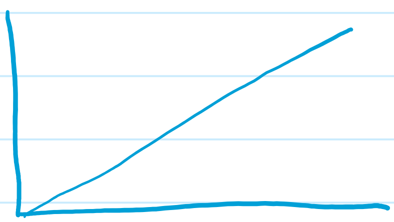
This algorithm will need to perform N operations in order to complete.

We use something called the big-O notation to describe the theoretical upper bound of an algorithm as N reaches infinity.

With big-O notation we drop all units and the constants

example:

O(N) not $O(N\cancel{*3})$ O(N)

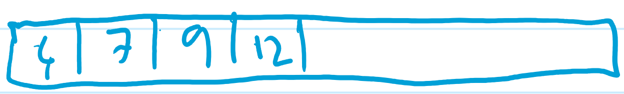


n

Best possible complexity?

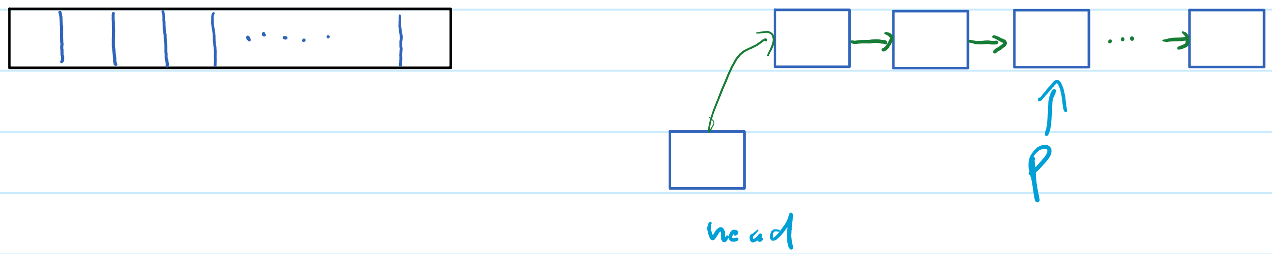
O(1)

insert (1)



(3) Complexity 3

Let's compare the two kinds of lists we are familiar with:



What worst case big-O complexity does inserting into an array have? $O(N)$

Array search? $O(N)$

Array access (assuming we know the index)?

$O(1)$

What about a linked list?

Insert - assuming we know exactly where the new node is to be inserted? $O(1)$

Search? $O(N)$

Access? $O(N)$

Complexity is often used to compare sorting algorithms: bubble sort, heap sort, etc..
With regards to data structures, we talk about complexity of common operations: access, search, insert, delete