Data Structures and Big O

Led by Cade and Evan

What is an array?

- **Contiguously-allocated** data structure
- Has a **fixed-size**, whether dynamically or statically allocated
- O(1) **random access** of an element through its **address / index**

Let's analyze the Big O of arrays!

Unsorted Array

- **Insert** - O(1)*
  - add new items to end of array
- **Delete -** O(1)
  - overwrite the target with last element
- **Search -** O(N)
  - must perform a *linear search*

Sorted Array

- **Insert** - O(N)
  - Find position, then make room by *shifting* elements to the right
- **Delete** - O(N)
  - Find target, overwrite it by *shifting* elements to the left
- **Search** - O(logN)
  - can perform a *binary search*

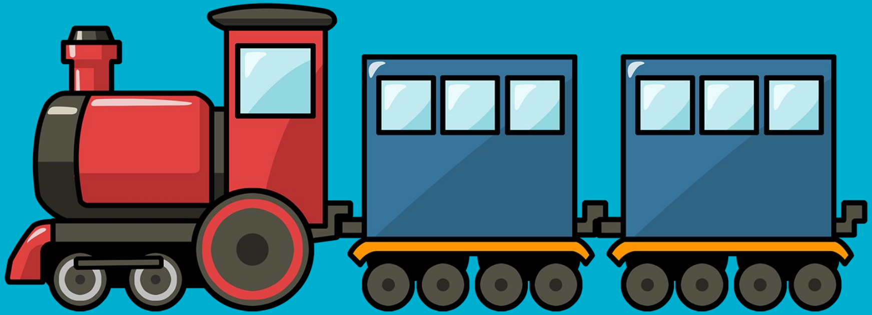
When should I use an array?

Use an array if your program...

- Requires **random access** to array elements
    - Binary search
- Knows **exactly how much memory** to allocate
    - No pointer entries for each item, but watch out for unused memory!
    - Ex: Nth letter of the alphabet: alphabet[N-1]
- Requires **fast iteration** speeds
    - Contiguous traversal vs. random pointer jumping
    - When you have to ‘do something’ to every item frequently
        - Ex: Array of Zombies to call move() function on each

What is a linked list?

- **Non-contiguous** data structure
- **Sequential access** of data items ONLY
  - Each node holds the pointer to the next node
- No restrictions on size



Let’s analyze the Big O of linked lists!

Unsorted & Sorted List

- **Insert** - O(1)
    - No shifting required, only have to find the correct position in the list
- **Delete** - O(1)
    - No shifting required, only have to find the correct target in the list
- **Search** - O(N)
    - CANNOT perform *binary search*, only the slower *linear search*

When should I use a linked list?

Use a linked list if your program...

- Performs a lot of **insertions** and **deletions**, especially **in the middle of a list**
  - No shifting of elements required for insert & delete
  - Ex: Linked List representing a playlist of songs in a music app
- Implements a **back-and-forth** functionality
  - Ex: Doubly-linked List representing a user's browser history

What are stacks and queues?

- Used primarily as **programmer's tools** to model particular problems
- ADT that **enforces restricted access** to items
  - Stacks = LIFO data structure
  - Queue = FIFO data structure
- Typically implemented by arrays or linked lists

Rear of Queue
Front of Queue
next...
PAY HERE

Let’s analyze the Big O of stacks and queues!

Stack

- **Insert/Push -** O(1)
  - insert at the end of the internal array
- **Delete/Pop -** O(1)
  - remove the last element of the internal array
- **Search -** O(N)
  - Can only check every element by removing from the top

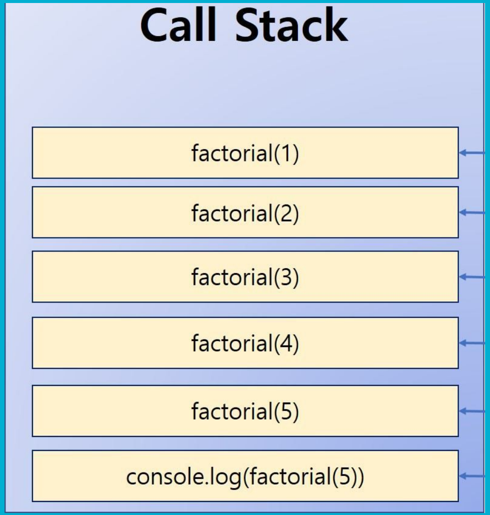
Queue

- **Insert/Push -** O(1)
  - insert at the ‘back’ of the circular array
- **Delete/Pop -** O(1)
  - delete at the ‘front’ of the circular array
- **Search -** O(N)
  - Can only check every element by removing from the front

When should I use a stack?

Use a stack if your program uses...

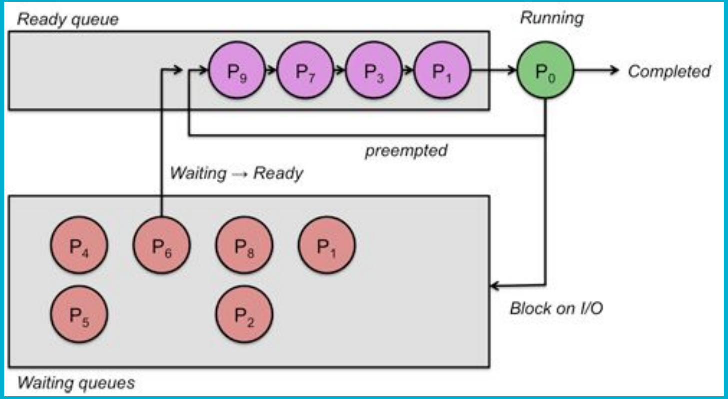
- Recursion (implicit use of a stack)
- Depth-first graph traversal
  - Ex: Preorder, inorder, postorder tree traversal
- Postfix to Infix notation
  - Expression evaluation
- Simple sorting
  - ascending order <-> descending order
- Undo/Redo functionality



When should I use a queue?

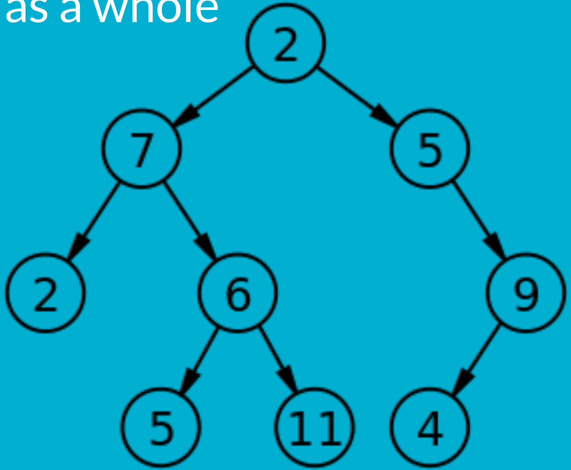
Use a queue if your program uses...

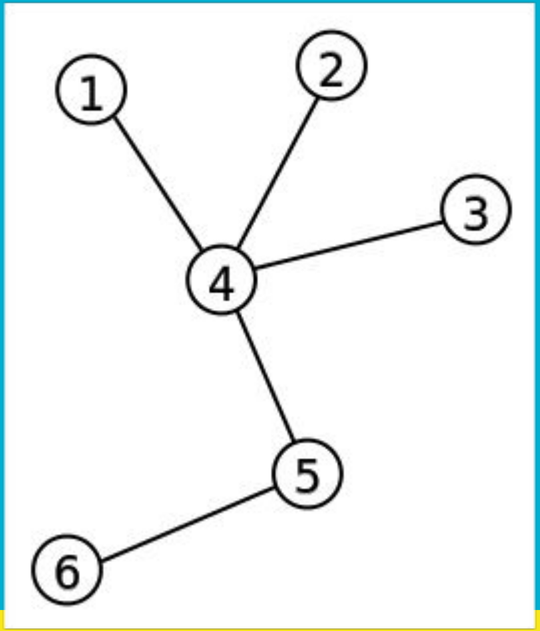
- A queue/waiting list
  - Ex: A print queue, modeling a grocery store line
- A resource that is shared
  - Ex: CPU process scheduling to ensure fairness
- Breadth-first graph traversal
  - Ex: Level-order traversal of a tree



What are trees?

- Trees can be defined recursively, as a node with a value and a collection of children, with the constraints that no reference is duplicated, and none points to the root
- We can also think of trees as a whole
-
- Trees are more useful if
  - We use them as BSTs
  - We use them as heaps





Let’s analyze the Big O of trees!

For a BST

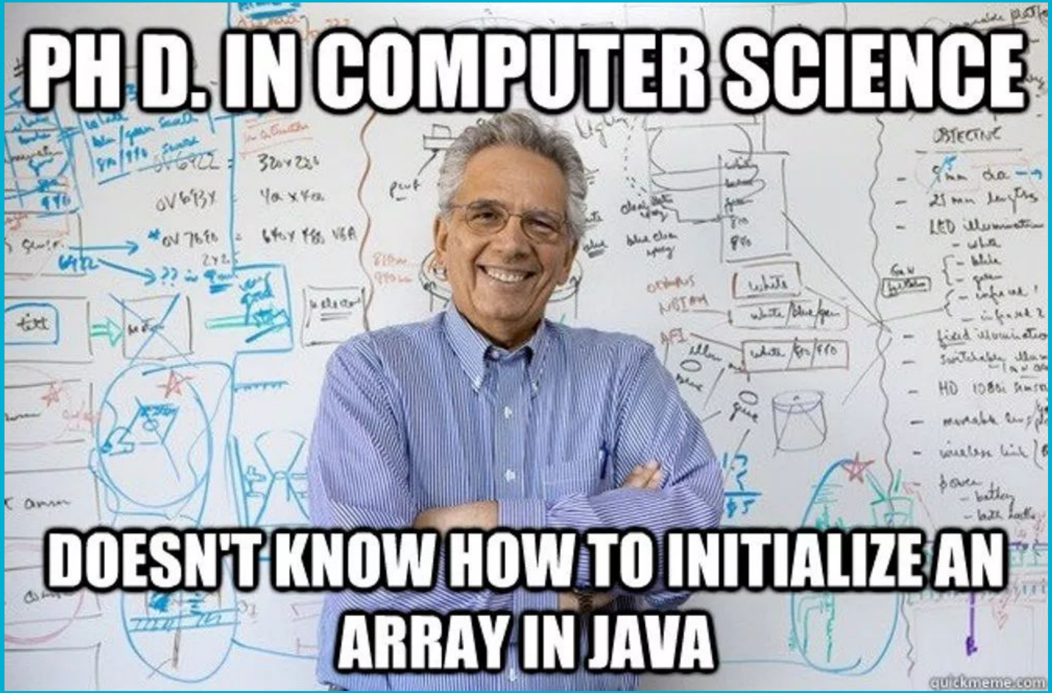
- **Insert** - O(log n)
    - No shifting required, only have to find the correct position in the tree
- **Search -** O(log n)
    - Binary Search
- **Delete** - O(log n)
        - Start with a binary search. No shifting required after

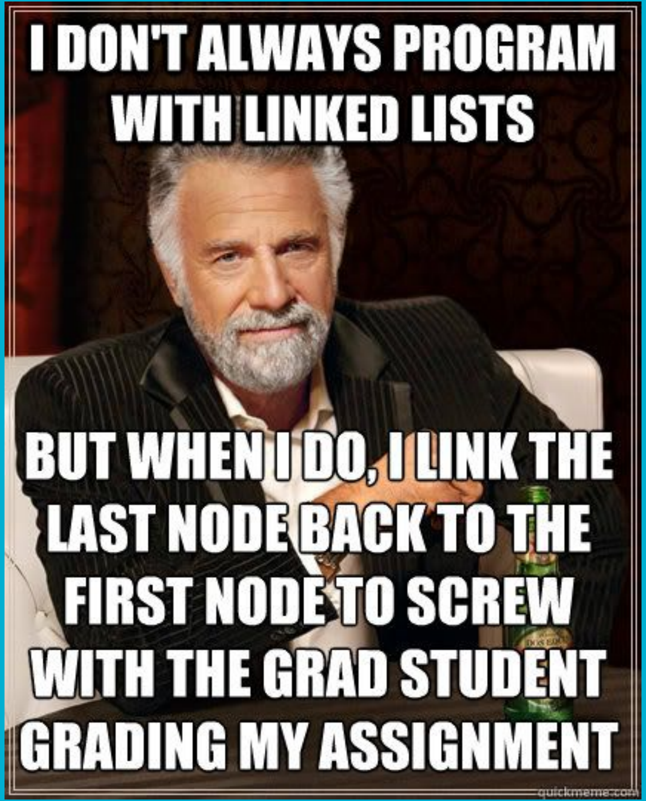
Why does it not make sense to analyze the complexity of a normal (non-BST) tree?

When should I use a tree?

Use a tree if your program...

- Will often need to store data that is well balanced
  - Perfectly unbalanced binary trees perform in O(n) for all operations
- Accesses keys in sorted order frequently
  - Then you can just do in-order traversal of the tree


PH D. IN COMPUTER SCIENCE
DOESN'T KNOW HOW TO INITIALIZE AN ARRAY IN JAVA
quickmeme.com


I DON'T ALWAYS PROGRAM WITH LINKED LISTS
BUT WHEN I DO, I LINK THE LAST NODE BACK TO THE FIRST NODE TO SCREW WITH THE GRAD STUDENT GRADING MY ASSIGNMENT
quickmeme.com

Let’s try some examples with arrays and LLs

https://github.com/cayd/LLvsArr

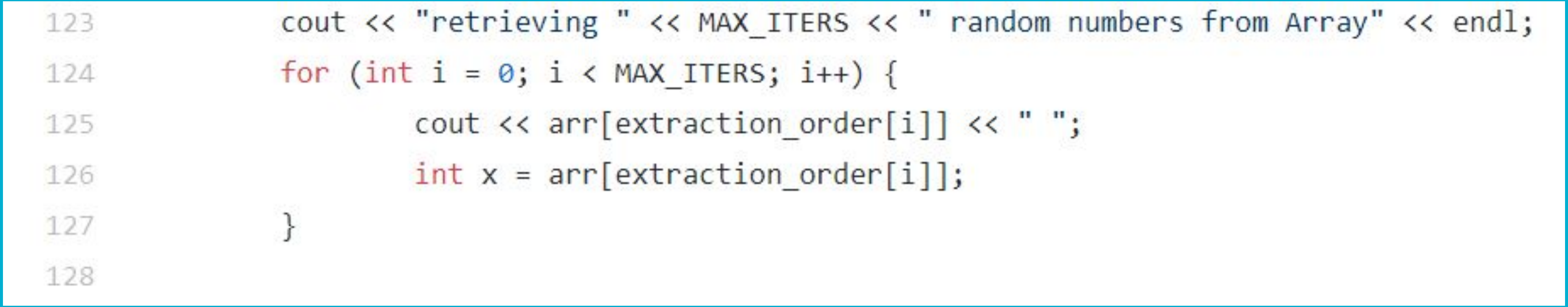
Which would you prefer here?

I want to insert 1000 random integers into my data structure.

Then I want to extract the elements in a random order.

Let’s run it

Where are we spending most of our time?

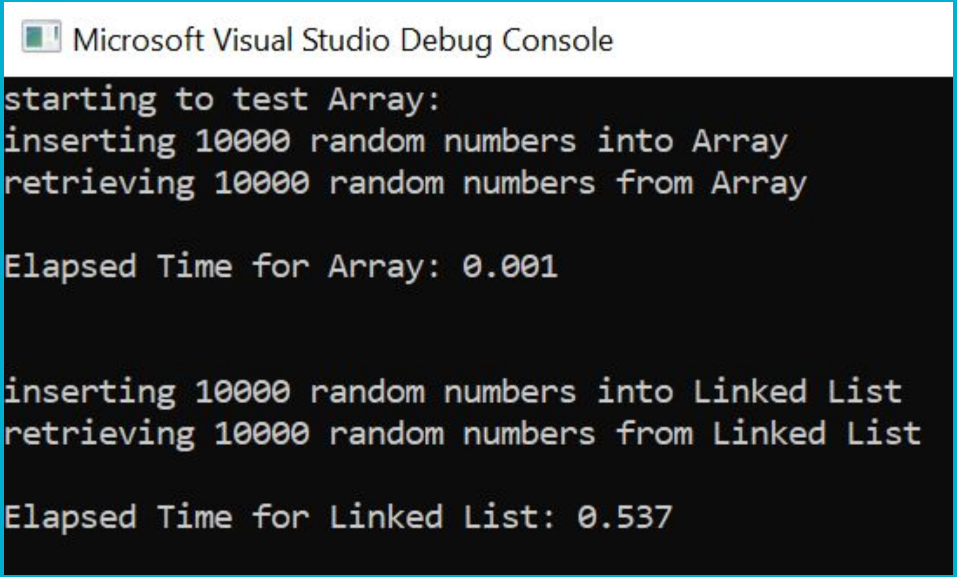


```
        cout << "retrieving " << MAX_ITERS << " random numbers from Array" << endl;
        for (int i = 0; i < MAX_ITERS; i++) {
                cout << arr[extraction_order[i]] << " ";
                int x = arr[extraction_order[i]];
        }

```

It turns out we left our print statements inside our timing code. Darn.

It seems the results are...



```
starting to test Array:
inserting 10000 random numbers into Array
retrieving 10000 random numbers from Array

Elapsed Time for Array: 0.001


inserting 10000 random numbers into Linked List
retrieving 10000 random numbers from Linked List

Elapsed Time for Linked List: 0.537
```

Linked lists are slow for random insertion and deletion.

We also know the size of the struct up front, so there is no wasted space from unused elements in the array.

How about here?

I want to insert a random number 100-1000 of random elements 100-1000 into my data structure.

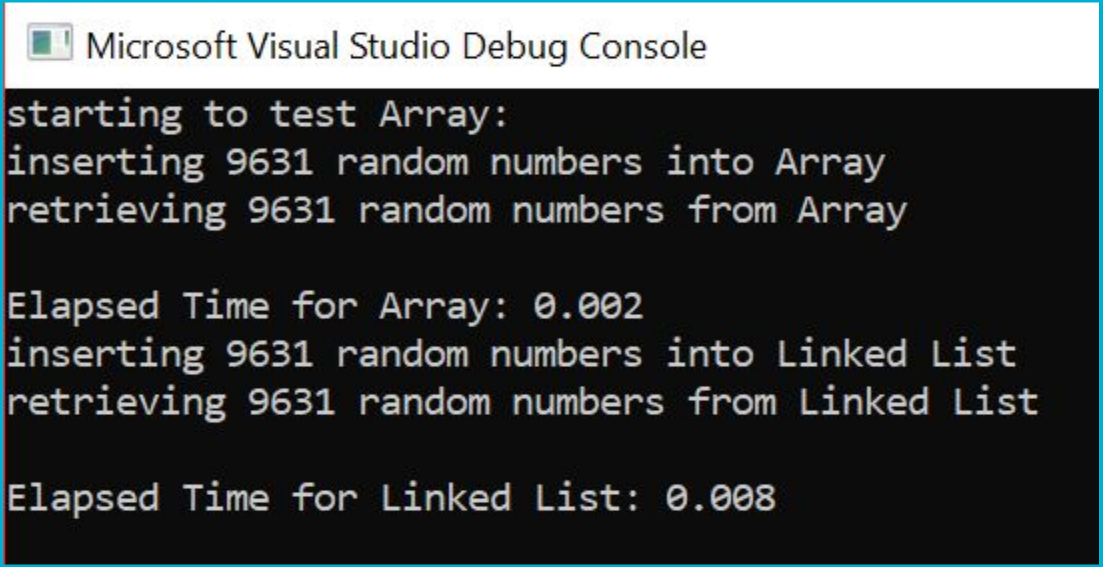
Then I want to retrieve them in the reverse order of insertion.

Ex. I insert 1, 987, 62.

Then I want to withdraw 62, 987, 1

Let’s run it

Results are...



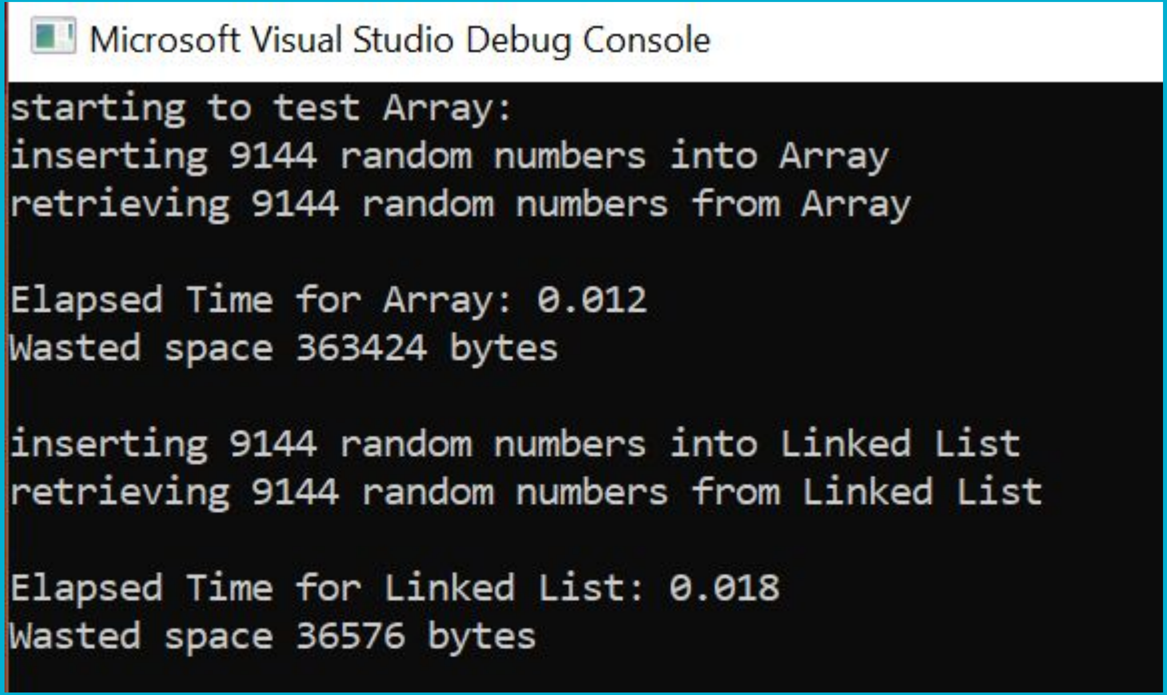
```
starting to test Array:
inserting 9631 random numbers into Array
retrieving 9631 random numbers from Array

Elapsed Time for Array: 0.002
inserting 9631 random numbers into Linked List
retrieving 9631 random numbers from Linked List

Elapsed Time for Linked List: 0.008
```

Not quite the performance gains we were expecting

Remember, Linked Lists save space



```
starting to test Array:
inserting 9144 random numbers into Array
retrieving 9144 random numbers from Array

Elapsed Time for Array: 0.012
Wasted space 363424 bytes

inserting 9144 random numbers into Linked List
retrieving 9144 random numbers from Linked List

Elapsed Time for Linked List: 0.018
Wasted space 36576 bytes
```

So let’s think

Come up with a scenario where a linked list would be quicker than an array.

Some other situations to think through

- What if I need to nest more data structures?
- What if I want to sort the data structure?
- What if I frequently want to remove elements and keep the space contiguous?

Recap

Arrays are contiguous space in memory with a fixed size.

Linked lists are easy to dynamically allocate and grow as you need them to without overallocating space. They also make deletions a very fast operation.

The main reason to prefer a stack or queue is if its behavior is what you want from it.

BSTs are consistently quick for all operations as long as the data is well balanced.

If you aren't sure which to use, just create an example that's representative of your use case and try it out!

Questions?