echo "# Datastructures\_and\_algorithms" >> README.md

git init

git add README.md

git commit -m "first commit"

git branch -M main

git remote add origin https://github.com/seppotk/Datastructures\_and\_algorithms.git

git push -u origin main

**…or push an existing repository from the command line**

git remote add origin https://github.com/seppotk/Datastructures\_and\_algorithms.git

git branch -M main

git push -u origin main

<https://github.com/TT00FE39-3001/lecture1>

<https://cpp.sh/>

C:\Users\Seppo>g++ --version

g++ (MinGW-W64 x86\_64-ucrt-posix-seh, built by Brecht Sanders) 12.2.0

Copyright (C) 2022 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO

warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

C:\Users\Seppo>clang --version

(built by Brecht Sanders) clang version 14.0.6

Target: x86\_64-w64-windows-gnu

Thread model: posix

InstalledDir: C:/mingw64/bin

**Outline**

* Data structures & Abstract Data Types (ADT)
* Algorithms
* Course Mechanics?

**Part 1**

* [Abstract Data Types (ADT)](https://en.wikipedia.org/wiki/Abstract_data_type)
* [Data Structures](https://en.wikipedia.org/wiki/Data_type)
* Data structures are language dependant

[**Data Structures In C++**](https://www.softwaretestinghelp.com/data-structures-in-cpp/)

* Introduction
* Classification
* [Advantages](https://www.softwaretestinghelp.com/data-structures-in-cpp/)

**Stacks vs Queues (1/2)**

* [Stack Data Structure In C++ With ARRAY/Linked list implementation](https://www.softwaretestinghelp.com/stack-in-cpp/)
* [Queue Data Structure In C++ With Illustration](https://www.softwaretestinghelp.com/queue-in-cpp/)

**Stacks vs Queues (2/2)**

* [Data structures 101: Stacks vs Queues](https://www.educative.io/blog/data-structures-stack-queue-java-tutorial)
* [Visualization](https://www.cs.usfca.edu/~galles/visualization/Algorithms.html)
* [FIFO](https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics))
* [LIFO](https://en.wikipedia.org/wiki/Stack_(abstract_data_type))

**Applications**

* Recursion
* Browser history
* Undo (Word processor)

**Part 2**

* Classification
* Analysis of Algorithm Efficiency
* Space vs Time
* Algorithms language agnostic
  + Pseudo code / C++ etc
* [Introduction To Searching Algorithms In C++](https://www.softwaretestinghelp.com/searching-algorithms-in-cpp/)
* [Introduction To Sorting Techniques In C++](https://www.softwaretestinghelp.com/sorting-techniques-in-cpp/)
* [Algorithms In STL](https://www.softwaretestinghelp.com/algorithms-in-stl/)

**Classification**

* Brute Force
* Divide-and-Conquer
* Transform-and-Conquer
* Greedy Technique
* Dynamic Programming

**Analysis of Algorithm Efficiency**

[Big-O, Little-O, Theta, Omega](https://cathyatseneca.gitbooks.io/data-structures-and-algorithms/content/analysis/notations.html)

* Big �, Little �
* Big Ω, Little �
* Θ

**Space vs Time**

* Same tools
* Space and Time Trade-Offs

**Tools**

* [Algorithm Visualizations 1](https://www.youtube.com/playlist?list=PLlzjt-kuLwlo6QuXHSZCkITkfn9M2yHg1)
* [Algorithm Visualizations 2](https://gbhat.com/algorithms/selection_sort.html)
* [Animation engine for explanatory math videos](https://github.com/3b1b/manim)

first try:

#include <iostream>

using namespace std;

int main()

{

cout<<"Hello,World!! This is C++ Tutorial!!\n";

cin.get();

return 0;

}

HOMEWORK

**# Homework**

**## Reading**

- [Stack Data Structure In C++](<https://www.softwaretestinghelp.com/stack-in-cpp/>)

- [Queue Data Structure In C++](<https://www.softwaretestinghelp.com/queue-in-cpp/>)

**## Pre-Lecture**

- [Data Structures & Algorithms (~16min)](<https://youtu.be/bum_19loj9A>)

- [An Overview of Arrays and Memory (~20min)](<https://youtu.be/pmN9ExDf3yQ>)

- [Introduction to Big O Notation and Time Complexity (~36min)](<https://youtu.be/D6xkbGLQesk>)

**## Recommended: Review**

- Videos:

  - [C++ Tutorial for Beginners - Full Course (4 HOURS!)](<https://youtu.be/vLnPwxZdW4Y>)

  - C[++ Programming All-in-One Tutorial Series (10 HOURS!)](<https://youtu.be/_bYFu9mBnr4>)

  - [ C++ Programming Course - Beginner to Advanced (30 HOURS!)](<https://youtu.be/8jLOx1hD3_o>)

- <https://www.learncpp.com/>

-----------------------------------------

ANSWERS:

**## Reading**

- [Stack Data Structure In C++](<https://www.softwaretestinghelp.com/stack-in-cpp/>)

- [Queue Data Structure In C++](<https://www.softwaretestinghelp.com/queue-in-cpp/>)

**All That You Need To Know About Stack In C++.**

Stack is a fundamental data structure which is used to store elements in a linear fashion.

Stack follows **LIFO (last in, first out)** order or approach in which the operations are performed. This means that the element which was added last to the stack will be the first element to be removed from the stack.

**=>**[**Visit Here To See The Entire C++ Training Series For All.**](https://www.softwaretestinghelp.com/cpp-tutorials/)

**Basic Operations**

Following are the basic operations that are supported by the stack.

* **push –** Adds or pushes an element into the stack.
* **pop –**Removes or pops an element out of the stack.
* **peek –** Gets the top element of the stack but doesn’t remove it.
* **isFull –**Tests if the stack is full.
* **isEmpty –** Tests if the stack is empty.

#include<iostream>

**using** **namespace** std;

#define MAX 1000 //max size for stack

**class** Stack

{

**int** top;

**public**:

**int** myStack[MAX]; //stack array

   Stack() { top = -1; }

**bool** push(**int** x);

**int** pop();

**bool** isEmpty();

};

   //pushes element on to the stack

**bool** Stack::push(**int** item)

   {

**if** (top >= (MAX-1)) {

      cout << "Stack Overflow!!!";

**return** **false**;

   }

**else** {

   myStack[++top] = item;

   cout<<item<<endl;

**return** **true**;

   }

}

//removes or pops elements out of the stack

**int** Stack::pop()

{

**if** (top < 0) {

      cout << "Stack Underflow!!";

**return** 0;

   }

**else** {

**int** item = myStack[top--];

**return** item;

   }

}

//check if stack is empty

**bool** Stack::isEmpty()

{

**return** (top < 0);

}

// main program to demonstrate stack functions

**int** main()

{

**class** Stack stack;

   cout<<"The Stack Push "<<endl;

   stack.push(2);

   stack.push(4);

   stack.push(6);

   cout<<"The Stack Pop : "<<endl;

**while**(!stack.isEmpty())

      {

      cout<<stack.pop()<<endl;

      }

**return** 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\Programs> .\stack\_using\_arrays

The Stack Push

2

4

6

The Stack Pop :

6

4

2

#include <iostream>

using namespace std;

// class to represent a stack node

class StackNode {

   public:

   int data;

   StackNode\* next;

   };

StackNode\* newNode(int data) {

   StackNode\* stackNode = new StackNode();

   stackNode->data = data;

   stackNode->next = NULL;

   return stackNode;

   }

int isEmpty(StackNode \*root) {

   return !root;

   }

void push(StackNode\*\* root, int new\_data){

   StackNode\* stackNode = newNode(new\_data);

   stackNode->next = \*root;

   \*root = stackNode;

   cout<<new\_data<<endl;

   }

int pop(StackNode\*\* root){

   if (isEmpty(\*root))

   return -1;

   StackNode\* temp = \*root;

   \*root = (\*root)->next;

   int popped = temp->data;

   free(temp);

   return popped;

}

int peek(StackNode\* root)

{

   if (isEmpty(root))

   return -1;

   return root->data;

}

int main()

{

   StackNode\* root = NULL;

   cout<<"Stack Push:"<<endl;

   push(&root, 100);

   push(&root, 200);

   push(&root, 300);

   cout<<"\nTop element is "<<peek(root)<<endl;

   cout<<"\nStack Pop:"<<endl;

   while(!isEmpty(root)){

   cout<<pop(&root)<<endl;

}

cout<<"Top element is "<<peek(root)<<endl;

return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\Programs> .\stack\_using\_linked\_list

Stack Push:

100

200

300

Top element is 300

Stack Pop:

300

200

100

Top element is -1

**A Brief Introduction To Queue In C++ With Illustration.**

The queue is a basic data structure just like a stack. In contrast to stack that uses the LIFO approach, queue uses the FIFO (first in, first out) approach. With this approach, the first item that is added to the queue is the first item to be removed from the queue. Just like Stack, the queue is also a linear data structure.

In a real-world analogy, we can imagine a bus queue where the passengers wait for the bus in a queue or a line. The first passenger in the line enters the bus first as that passenger happens to be the one who had come first.

**The queue data structure includes the following operations:**

* **EnQueue:** Adds an item to the queue. Addition of an item to the queue is always done at the rear of the queue.
* **DeQueue:** Removes an item from the queue. An item is removed or de-queued always from the front of the queue.
* **isEmpty:** Checks if the queue is empty.
* **isFull:** Checks if the queue is full.
* **peek:** Gets an element at the front of the queue without removing it.

#include <iostream>

#define MAX\_SIZE 5

using namespace std;

class Queue {

private:

int myqueue[MAX\_SIZE], front, rear;

public:

Queue(){

front = -1;

rear = -1;

    }

bool isFull(){

if(front == 0 && rear == MAX\_SIZE - 1){

return true;

        }

return false;

    }

bool isEmpty(){

if(front == -1) return true;

else return false;

    }

void enQueue(int value){

if(isFull()){

cout << endl<< "Queue is full!!";

        } else {

if(front == -1) front = 0;

rear++;

myqueue[rear] = value;

cout << value << " ";

        }

    }

int deQueue(){

int value;

if(isEmpty()){

cout << "Queue is empty!!" << endl; return(-1); } else { value = myqueue[front]; if(front >= rear){      //only one element in queue

front = -1;

rear = -1;

            }

else {

front++;

            }

cout << endl << "Deleted => " << value << " from myqueue";

return(value);

        }

    }

    /\* Function to display elements of Queue \*/

void displayQueue()

    {

int i;

if(isEmpty()) {

cout << endl << "Queue is Empty!!" << endl;

        }

else {

cout << endl << "Front = " << front;

cout << endl << "Queue elements : ";

for(i=front; i<=rear; i++)

cout << myqueue[i] << "\t";

cout << endl << "Rear = " << rear << endl;

        }

    }

};

int main()

{

    Queue myq;

myq.deQueue();      //deQueue

cout<<"Queue created:"<<endl; myq.enQueue(10); myq.enQueue(20); myq.enQueue(30); myq.enQueue(40); myq.enQueue(50); //enqueue 60 => queue is full

myq.enQueue(60);

myq.displayQueue();

    //deQueue =>removes 10

myq.deQueue();

    //queue after dequeue

myq.displayQueue();

return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\Programs> .\queue\_using\_arrays

Queue is empty!!

Queue created:

10 20 30 40 50

Queue is full!!

Front = 0

Queue elements : 10 20 30 40 50

Rear = 4

Deleted => 10 from myqueue

Front = 1

Queue elements : 20 30 40 50

Rear = 4

#include <iostream>

using namespace std;

struct node {

int data;

struct node \*next;

};

struct node\* front = NULL;

struct node\* rear = NULL;

struct node\* temp;

void Insert(int val) {

if (rear == NULL) {

rear = new node;

rear->next = NULL;

rear->data = val;

front = rear;

     } else {

temp=new node;

rear->next = temp;

temp->data = val;

temp->next = NULL;

rear = temp;

   }

}

void Delete() {

temp = front;

if (front == NULL) {

cout<<"Queue is empty!!"<<endl; } else if (temp->next != NULL) {

temp = temp->next;

cout<<"Element deleted from queue is : "<<front->data<<endl;

free(front);

front = temp;

   } else {

cout<<"Element deleted from queue is : "<<front->data<<endl;

free(front);

front = NULL;

rear = NULL;

   }

}

void Display() {

temp = front;

if ((front == NULL) && (rear == NULL)) {

cout<<"Queue is empty"<<endl;

return;

   }

while (temp != NULL) {

cout<<temp->data<<" "; temp = temp->next;

   }

cout<<endl;

}

int main() {

cout<<"Queue Created:"<<endl;

Insert(10);

Insert(20);

Insert(30);

Insert(40);

Insert(50);

Display();

Delete();

cout<<"Queue after one deletion: "<<endl;

Display();

return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\Programs> .\queue\_using\_linked\_list

Queue Created:

10 20 30 40 50

Element deleted from queue is : 10

Queue after one deletion:

20 30 40 50

**Stack Vs. Queue**

Stacks and queues are secondary data structures which can be used to store data. They can be programmed using the primary data structures like arrays and linked lists. Having discussed both the data structures in detail, it’s time to discuss the main differences between these two data structures.

| **Stacks** | **Queues** |
| --- | --- |
| Uses LIFO (Last in, First out) approach. | Uses FIFO (First in, First out) approach. |
| Items are added or deleted from only one end called “Top” of the stack. | Items are added from “Rear” end of the queue and are removed from the “front” of the queue. |
| The basic operations for the stack are “push” and “Pop”. | The basic operations for a queue are “enqueue” and “dequeue”. |
| We can do all operations on the stack by maintaining only one pointer to access the top of the stack. | In queues, we need to maintain two pointers, one to access the front of the queue and the second one to access the rear of the queue. |
| The stack is mostly used to solve recursive problems. | Queues are used to solve problems related to ordered processing. |

0:13 / 16:34

•- [Data Structures & Algorithms (~16min)](<https://youtu.be/bum_19loj9A>)

Intro

**Data Structures & Algorithms #1 - What Are Data Structures?**

[CS Dojo](https://www.youtube.com/@CSDojo)

CS Dojo

Kiinnitti CS Dojo

[CS Dojo](https://www.youtube.com/channel/UCxX9wt5FWQUAAz4UrysqK9A)

[4 vuotta sitten](https://www.youtube.com/watch?v=bum_19loj9A&lc=UgzvD1h-PRr2WtG55LZ4AaABAg)

Here's the playlist for this entire series: [https://goo.gl/wy3CWF](https://www.youtube.com/redirect?event=comments&redir_token=QUFFLUhqbmhtRzVFNGxZbnNhVFJSWV9qclhCMm9mN3JrQXxBQ3Jtc0trM05hTlotREx3cFVDNmkzc2QzZ0QydUlnU0dOMlZTZDRkQUFrQ3ByVUlNSnoxQ0tQQXVlTE5qajFKLTF3N1JIRHdZTzd4cFZLdE5JdmZIZ2tEM1VwRkxYZWlZRXhycVV5UTJWTmRuRE5JZDE5andfVQ&q=https%3A%2F%2Fgoo.gl%2Fwy3CWF&stzid=UgzvD1h-PRr2WtG55LZ4AaABAg)

442

<https://www.youtube.com/playlist?list=PLBZBJbE_rGRV8D7XZ08LK6z-4zPoWzu5H>

- [An Overview of Arrays and Memory (~20min)](<https://youtu.be/pmN9ExDf3yQ>)

# An Overview of Arrays and Memory (Data Structures & Algorithms #2

Here’s an outline of this video:

[0:37](https://www.youtube.com/watch?v=pmN9ExDf3yQ&t=37s): A quick overview / refresher of arrays

[2:49](https://www.youtube.com/watch?v=pmN9ExDf3yQ&t=169s): What is memory?

[6:46](https://www.youtube.com/watch?v=pmN9ExDf3yQ&t=406s): Examining memory and storage on my laptop

- [Introduction to Big O Notation and Time Complexity (~36min)](<https://youtu.be/D6xkbGLQesk>)

# Introduction to Big O Notation and Time Complexity (Data Structures & Algorithms

36:21

This was #7 of my data structures & algorithms series. You can find the entire series in a playlist here: [https://goo.gl/wy3CWF](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbjk4b3d3Y2kxb1RnUmlKbXJpbGhOUWlmTnRlQXxBQ3Jtc0ttUXhZdDJYMS1ldzc2bDh5VVZJQl9KdS1td3podE5teDQ1SkpKanJqc0VUbWhDZzZxNVMtYldGSDViYzU4RTVOb0FXNHRtcDFxSUE3bVNZbmhRN19vSmptMHZZQkpGbDZmNlpiaGxybDl2Xy1MaXNGOA&q=https%3A%2F%2Fgoo.gl%2Fwy3CWF&v=D6xkbGLQesk)

Graphical user interface, text, application, chat or text message

Description automatically generated

ACTIVITY 1

**# Activities**

**## Task 1**

Watch this video (~2min):

<https://youtu.be/ohSzM7WtwOk>

Discuss the difference between:

- Queue vs Stack

- LIFO vs FIFO

**## Task 2**

- Use the following link to push/pop data from the stack:

<https://www.cs.usfca.edu/~galles/visualization/StackArray.html>

- Use the following tool to enqueue / dequeue data from the queue:

<https://www.cs.usfca.edu/~galles/visualization/QueueArray.html>

**## Task 3**

- What is the difference between Array Implementation and the Linked List Implementation of Stacks. Refer to the following link:

<https://www.cs.usfca.edu/~galles/visualization/StackLL.html>

- What is the difference between Array Implementation and the Linked List Implementation of Queues. Refer to the following link:

<https://www.cs.usfca.edu/~galles/visualization/QueueLL.html>

**## Links**

- <https://www.educative.io/blog/data-structures-stack-queue-java-tutorial>

ANSWERS:

**## Task 1**

Watch this video (~2min):

<https://youtu.be/ohSzM7WtwOk>

Discuss the difference between:

- Queue vs Stack

- LIFO vs FIFO

QUEUE FIFO

First in first out

add and remove from a opposite ends of the list

Double pointer

operations enqueue, queue, peek

STACK LIFO

Last in last out

add and remove from a single end of the list

single pointer

operations push, pop, peek

**## Task 2**

- Use the following link to push/pop data from the stack:

<https://www.cs.usfca.edu/~galles/visualization/StackArray.html>

- Use the following tool to enqueue / dequeue data from the queue:

<https://www.cs.usfca.edu/~galles/visualization/QueueArray.html>

Table

Description automatically generated

Table, Excel

Description automatically generated

**## Task 3**

- What is the difference between Array Implementation and the Linked List Implementation of Stacks. Refer to the following link:

<https://www.cs.usfca.edu/~galles/visualization/StackLL.html>

- What is the difference between Array Implementation and the Linked List Implementation of Queues. Refer to the following link:

<https://www.cs.usfca.edu/~galles/visualization/QueueLL.html>

Graphical user interface, text, application, table, Excel

Description automatically generated

A picture containing Excel

Description automatically generated

117

There are multiple different ways to implement queues and stacks with linked lists and arrays, and I'm not sure which ones you're looking for. Before analyzing any of these structures, though, let's review some important runtime considerations for the above data structures.

In a singly-linked list with just a head pointer, the cost to prepend a value is O(1) - we simply create the new element, wire its pointer to point to the old head of the list, then update the head pointer. The cost to delete the first element is also O(1), which is done by updating the head pointer to point to the element after the current head, then freeing the memory for the old head (if explicit memory management is performed). However, the constant factors in these O(1) terms may be high due to the expense of dynamic allocations. The memory overhead of the linked list is usually O(n) total extra memory due to the storage of an extra pointer in each element.

In a dynamic array, we can access any element in O(1) time. We can also append an element in [amortized O(1)](http://en.wikipedia.org/wiki/Amortized_analysis), meaning that the total time for n insertions is O(n), though the actual time bounds on any insertion may be much worse. Typically, dynamic arrays are implemented by having most insertions take O(1) by appending into preallocated space, but having a small number of insertions run in Θ(n) time by doubling the array capacity and copying elements over. There are techniques to try to reduce this by allocating extra space and lazily copying the elements over (see [this data structure](https://stackoverflow.com/questions/4834490/what-would-be-an-ideal-data-structure-for-an-add-only-no-removals-collection), for example). Typically, the memory usage of a dynamic array is quite good - when the array is completely full, for example, there is only O(1) extra overhead - though right after the array has doubled in size there may be O(n) unused elements allocated in the array. Because allocations are infrequent and accesses are fast, dynamic arrays are usually faster than linked lists.

Now, let's think about how to implement a stack and a queue using a linked list or dynamic array. There are many ways to do this, so I will assume that you are using the following implementations:

* Stack:
  + Linked list: As a [singly-linked list](http://en.wikipedia.org/wiki/Linked_list#Singly.2C_doubly.2C_and_multiply_linked_lists) with a head pointer.
  + Array: As a [dynamic array](http://en.wikipedia.org/wiki/Dynamic_array)
* Queue:
  + Linked list: As a singly-linked list with a head and tail pointer.
  + Array: As a [circular buffer](http://en.wikipedia.org/wiki/Circular_buffer) backed by an array.

Let's consider each in turn.

**Stack backed by a singly-linked list.** Because a singly-linked list supports O(1) time prepend and delete-first, the cost to push or pop into a linked-list-backed stack is also O(1) worst-case. However, each new element added requires a new allocation, and allocations can be expensive compared to other operations.

**Stack backed by a dynamic array.** Pushing onto the stack can be implemented by appending a new element to the dynamic array, which takes amortized O(1) time and worst-case O(n) time. Popping from the stack can be implemented by just removing the last element, which runs in worst-case O(1) (or amortized O(1) if you want to try to reclaim unused space). In other words, the most common implementation has best-case O(1) push and pop, worst-case O(n) push and O(1) pop, and amortized O(1) push and O(1) pop.

**Queue backed by a singly-linked list.** Enqueuing into the linked list can be implemented by appending to the back of the singly-linked list, which takes worst-case time O(1). Dequeuing can be implemented by removing the first element, which also takes worst-case time O(1). This also requires a new allocation per enqueue, which may be slow.

**Queue backed by a growing circular buffer.** Enqueuing into the circular buffer works by inserting something at the next free position in the circular buffer. This works by growing the array if necessary, then inserting the new element. Using a similar analysis for the dynamic array, this takes best-case time O(1), worst-case time O(n), and amortized time O(1). Dequeuing from the buffer works by removing the first element of the circular buffer, which takes time O(1) in the worst case.

To summarize, all of the structures support pushing and popping n elements in O(n) time. The linked list versions have better worst-case behavior, but may have a worse overall runtime because of the number of allocations performed. The array versions are slower in the worst-case, but have better overall performance if the time per operation isn't too important.

These aren't the only ways you can implement lists. You could have an [unrolled linked list](https://en.wikipedia.org/wiki/Unrolled_linked_list), where each linked list cell holds multiple values. This slightly increases the locality of reference of the lookups and decreases the number of allocations used. Other options (using a balanced tree keyed by index, for example) represent a different set of tradeoffs.

ACTIVITY 2

**# Activities**

Many data structures are used in four basic ways, which we refer to as operations. These operations are:

- Read: Reading refers to looking something up at a particular spot within the data structure. With an array, this means looking up a value at a particular index.

- Search: Searching refers to looking for a particular value within a data structure. With an array, this means looking to see if a particular value exists within the array, and if so, at which index.

- Insert: Insertion refers to adding a new value to our data structure. With an array, this means adding a new value to an additional slot within the array.

- Delete: Deletion refers to removing a value from our data structure. With an array, this means removing one of the values from the array.

Measuring the speed of an operation is also known as measuring its time complexity, efficiency, performance interchangeably. They all refer to the number of steps a given operation takes.

> In the following tasks we will analyze the time complexity of arrays and an array-based set. This is a primitive first order analysis. We will use special tools later

> A set is a data structure that does not allow duplicate values to be contained within it e.g. an array-based set is an array with one additional constraint of barring duplicates.

**## Task 1: Arrays**

Discuss in group whether the following statements are True or false.

- Reading from an array takes one step.

- Searching an array of N elements takes up to N steps e.g. for an array of 5 elements, the maximum number of steps is 5. For an array of 500 elements, the maximum number would take is 500.

- Insertion of an element in an array of length N, takes (N + 1) steps in worst-case scenario.

- Deletion of an element from an array of length N, takes N steps in worst-case scenario.

**## Task 2: Sets**

Discuss in group whether the following statements are True or false.

- Reading from an an array-based set takes one step.

- Searching an array-based set of N elements takes up to N steps.

- Insertion of an element in an array-based set of length N, takes (2N + 1) steps steps in worst-case scenario.

- Deletion of an element from an array-based set of length N, takes N steps in worst-case scenario.

**## Taks 3**

- Array-based sets are arrays with one additional constraint of barring duplicates. How does this single Rule affect efficiency?

- Should we avoid sets because insertion is slower for sets than regular arrays?

**## Reference**

- A Common-Sense Guide to Data Structures and Algorithms,Jay Wengrow

ANSWERS

Many data structures are used in four basic ways, which we refer to as operations. These operations are:

- Read: Reading refers to looking something up at a particular spot within the data structure. With an array, this means looking up a value at a particular index.

- Search: Searching refers to looking for a particular value within a data structure. With an array, this means looking to see if a particular value exists within the array, and if so, at which index.

- Insert: Insertion refers to adding a new value to our data structure. With an array, this means adding a new value to an additional slot within the array.

- Delete: Deletion refers to removing a value from our data structure. With an array, this means removing one of the values from the array.

Measuring the speed of an operation is also known as measuring its time complexity, efficiency, performance interchangeably. They all refer to the number of steps a given operation takes.

> In the following tasks we will analyze the time complexity of arrays and an array-based set. This is a primitive first order analysis. We will use special tools later

> A set is a data structure that does not allow duplicate values to be contained within it e.g. an array-based set is an array with one additional constraint of barring duplicates.

**## Task 1: Arrays**

Discuss in group whether the following statements are True or false.

- Reading from an array takes one step.

- Searching an array of N elements takes up to N steps e.g. for an array of 5 elements, the maximum number of steps is 5. For an array of 500 elements, the maximum number would take is 500.

- Insertion of an element in an array of length N, takes (N + 1) steps in worst-case scenario.

- Deletion of an element from an array of length N, takes N steps in worst-case scenario.

- Reading from an array takes one step.

Reading from an array is, therefore, an efficient operation, since the computer

can read any index by jumping to any memory address in one step.

- Searching an array of N elements takes up to N steps e.g. for an array of 5 elements, the maximum number of steps is 5. For an array of 500 elements, the maximum number would take is 500.

So, it turns out that for an array of five cells, the maximum number of steps

linear search would take is five. For an array of 500 cells, the maximum

number of steps linear search would take is 500.

- Insertion of an element in an array of length N, takes (N + 1) steps in worst-case scenario.

We can say that insertion in a worst-case scenario can take *N + 1 steps* for

an array containing N elements. This is because we need to shift all N elements

over, and then finally execute the actual insertion step.

- Deletion of an element from an array of length N, takes N steps in worst-case scenario.

We can say then, that for an array containing N elements,

the maximum number of steps that deletion would take is N steps.

**## Task 2: Sets**

Discuss in group whether the following statements are True or false.

- Reading from an an array-based set takes one step.

- Searching an array-based set of N elements takes up to N steps.

- Insertion of an element in an array-based set of length N, takes (2N + 1) steps steps in worst-case scenario.

- Deletion of an element from an array-based set of length N, takes N steps in worst-case scenario.

- Reading from an an array-based set takes one step.

Reading from a set is exactly the same as reading from an array—it takes just

one step for the computer to look up what’s contained within a particular index.

As I described earlier, this is because the computer can jump to any index

within the set since it can easily calculate and jump to its memory address.

- Searching an array-based set of N elements takes up to N steps.

Searching a set also turns out to be no different than searching an array—it

takes up to N steps to search for a value within a set. And deletion is also

identical between a set and an array—it takes up to N steps to delete a value

and move data to the left to close the gap.

- Insertion of an element in an array-based set of length N, takes (2N + 1) steps steps in worst-case scenario.

In the worst-case scenario, where we’re inserting a value at the *beginning* of

a set, the computer needs to search N cells to ensure that the set doesn’t

already contain that value, another N steps to shift all the data to the right,

and another final step to insert the new value. That’s a total of 2N + 1 steps.

Contrast this to insertion into the beginning of a regular array, which only

takes N + 1 steps.

- Deletion of an element from an array-based set of length N, takes N steps in worst-case scenario.

This is true. Same as array deletion at the beginning of the set. If the deletion is at the end of set it takes one step.

**## Task 3**

- Array-based sets are arrays with one additional constraint of barring duplicates. How does this single Rule affect efficiency?

- Should we avoid sets because insertion is slower for sets than regular arrays?

- Array-based sets are arrays with one additional constraint of barring duplicates. How does this single Rule affect efficiency?

Reading from a set is exactly the same as reading from an array—it takes just

one step for the computer to look up what’s contained within a particular index.

As I described earlier, this is because the computer can jump to any index

within the set since it can easily calculate and jump to its memory address.

Searching a set also turns out to be no different than searching an array—it

takes up to N steps to search for a value within a set. And deletion is also

identical between a set and an array—it takes up to N steps to delete a value

and move data to the left to close the gap.

Insertion, however, is where arrays and sets diverge. Let’s first explore

inserting a value at the *end* of a set, which was a best-case scenario for an

array. We saw that with an array, the computer can insert a value at its end

in a single step.

With a set, however, the computer first needs to determine that this value

doesn’t already exist in this set—because that’s what sets do: they prevent

duplicate data from being inserted into them.

Now, how will the computer ensure that the new data isn’t already contained

in the set? Remember, a computer doesn’t know offhand what values are

contained within the cells of an array or set. Because of this, the computer

will first need to *search* the set to see whether the value we want to insert is

already there. Only if the set does not yet contain our new value will the

computer allow the insertion to take place.

So, every insertion into a set *first requires a search.*

- Should we avoid sets because insertion is slower for sets than regular arrays?

Now, does this mean you should avoid sets just because insertion is slower

for sets than regular arrays? Absolutely not. Sets are important when you

need to ensure that there is no duplicate data. (Hopefully, one day my phone

book will be fixed.) But when you don’t have such a need, an array may be

preferable, since insertions for arrays are more efficient than insertions for

sets. You must analyze the needs of your own application and decide which

data structure is a better fit.

**## Reference**

- A Common-Sense Guide to Data Structures and Algorithms,Jay Wengrow

<https://pragprog.com/titles/jwdsal2/a-common-sense-guide-to-data-structures-and-algorithms-second-edition/>

<https://www.linkedin.com/in/jaywengrow/>

ACTIVITY 3

**# Git / Github**

- Clone today's repo

- Create a new branch e.g answers

- Create a repository in GitHub

- Change remote to point to your repo

**## Links**

- [How To Change Git Remote Origin](<https://devconnected.com/how-to-change-git-remote-origin/>)

DONE

<https://github.com/seppotk/Datastructures_and_algorithms>