# Table of Contents

[Introduction 3](#_Toc517781978)

[Data Structures and Algorithms 3](#_Toc517781979)

[Data Abstractions 3](#_Toc517781980)

[Algorithm Design and Analysis 5](#_Toc517781981)

[Memory and Number Systems 6](#_Toc517781982)

[Computer Memory 6](#_Toc517781983)

[Binary Representation 6](#_Toc517781984)

[Hexadecimal Representation 6](#_Toc517781985)

[Number System Conversion 6](#_Toc517781986)

[Prefixes 7](#_Toc517781987)

[Max/Min Values of Binary Number 8](#_Toc517781988)

[2’s Compliment 8](#_Toc517781989)

[Variable Types 8](#_Toc517781990)

[Memory 9](#_Toc517781991)

[Pointers 10](#_Toc517781992)

[Algorithms and Arrays 11](#_Toc517781993)

[Arrays 11](#_Toc517781994)

[Algorithms 11](#_Toc517781995)

[Components of an Algorithm 12](#_Toc517781996)

[Evaluation of an Algorithm 12](#_Toc517781997)

[Runtime Analysis 12](#_Toc517781998)

[Array Search Function Example 14](#_Toc517781999)

[Adding Elements to an Array 15](#_Toc517782000)

[Deleting Elements from an Array 15](#_Toc517782001)

[Doubling an Array 16](#_Toc517782002)

[Memory 16](#_Toc517782003)

[Heap 16](#_Toc517782004)

[Classes 17](#_Toc517782005)

[Classes 17](#_Toc517782006)

[Class vs Struct 17](#_Toc517782007)

[Class Methods 17](#_Toc517782008)

[Classes Example 18](#_Toc517782009)

[Updating Private Variables 18](#_Toc517782010)

[Linked Lists 19](#_Toc517782011)

[Linked Lists 19](#_Toc517782012)

[Linked List ADT 19](#_Toc517782013)

[Node 20](#_Toc517782014)

[Traversal 20](#_Toc517782015)

[Search 20](#_Toc517782016)

[Insert 21](#_Toc517782017)

[Delete 21](#_Toc517782018)

[Complexity 22](#_Toc517782019)

[Stacks & Queues 23](#_Toc517782020)

[Stacks 23](#_Toc517782021)

[Stack Implementation 23](#_Toc517782022)

[Stack ATD 24](#_Toc517782023)

[Queues 24](#_Toc517782024)

[Queues ADT 25](#_Toc517782025)

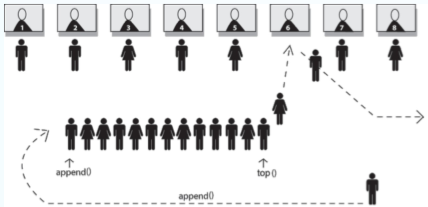
# Introduction

## Data Structures and Algorithms

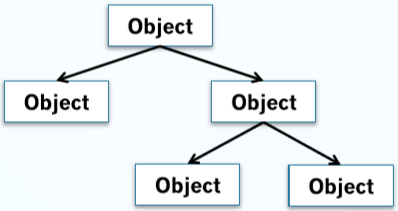
* Catalog Definition: Studied data abstractions[[1]](#footnote-1) and their representation techniques[[2]](#footnote-2).
  + Introduces concepts used in algorithm design and analysis including criteria for selecting data structures to fit their applications.

## Data Abstractions

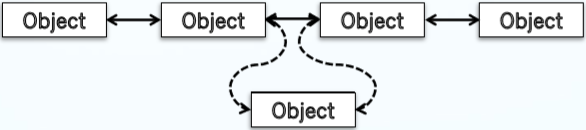
* *Abstract Data Type* (ADT) – The implementation mechanism commonly used in an object-oriented language
* *Programming* – A computational representation of the world
  + Abstract meaningful details for representation
  + Consists of variables and processes (Objects)
    - Bool
    - String
    - Int
    - Processes
* Collections of Objects
  + i.e. Collection of persons in line (queue) at a store
    - Features
      * Individual person objects
      * Order
      * [*Queue*](#_Queues)[[3]](#footnote-3) - Adding at one end, remove at the other



* + *Data Tree* – Collection of objects in a hierarchy
    - Features:
      * Individual objects
      * Parent-child object ordering
      * Adding, removing, and maintaining tree structure
      * Searching
    - i.e. [*Binary Search Tree*](#_Binary_Search_Trees)[[4]](#footnote-4) for sorting



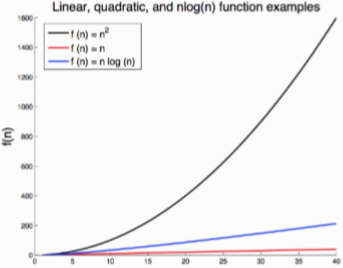
* + Arrays and Stacks
    - [*Array*](#_Algorithms_and_Arrays)[[5]](#footnote-5) Features
      * Contiguous[[6]](#footnote-6) in memory
      * Fixed locations
      * Egg[0] next to Egg[1]
      * Add, remove, search
    - [*Stack*](#_Stacks)[[7]](#footnote-7) Features
      * Similar to a *queue*
        + However, first in first out
      * Add and remove from the top
      * Order matters
      * Example: Commands executed in computer program
  + [*Linked List*](#_Linked_Lists)[[8]](#footnote-8) – Collection of objects linked together



* + - Features
      * Individual objects
      * Pointers between objects establish order
      * Add, remove by changing pointers
      * Locations not fixed
      * Number not fixed
    - i.e. Dynamic data storage[[9]](#footnote-9)

## Algorithm Design and Analysis

* How many operations to access a linked list or an array?
* How does an algorithm scale as data becomes really large?
  + n is the size of the data structure
    - i.e. number of elements in an array
  + f(n) is an operation of the data structure of size n
    - quadratic, linear, logarithmic



# Memory and Number Systems

## Computer Memory

* Memory on a computer is stored using binary *bits*
* *Bit* – Either a 1 or a 0
* *Byte* – A group of 8 bits
* Everything is a number
  + Your English paper, your favorite song, etc…

## Binary Representation

* Endian
  + Binary can be interpreted either reading left🡪right or right🡪left depending on the system
  + We will default to reading all binary in this course in the following format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 |

## Hexadecimal Representation

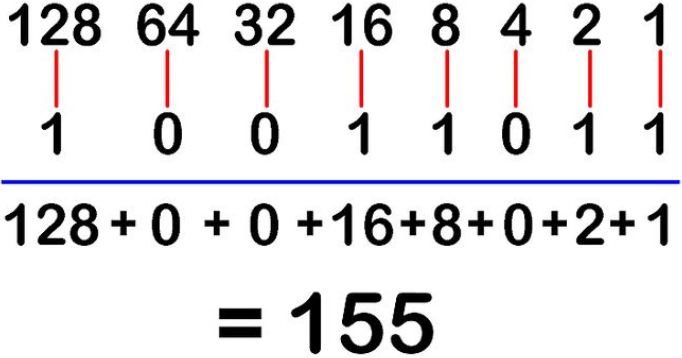
* Hexadecimal – Base 16
  + Numerical digits 0-9 are the same as decimal
  + A-F are 10-15

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Hexadecimal** | A | B | C | D | E | F |
| **Decimal** | 10 | 11 | 12 | 13 | 14 | 15 |

## Number System Conversion

* Converting from Binary to Decimal
  + Use powers of two to calculate decimal value

**OR**



* Converting from Decimal to Binary
  + Start binary value right🡪left
  + Divide decimal value by 2
  + If there is a remainder, enter a 1, else enter a 0

If(num/2 is odd) ? 1 : 0

* Converting to Hexadecimal

|  |  |  |
| --- | --- | --- |
| **Hex** | **Dec** | **Bin** |
| A | 10 | 1010 |
| B | 11 | 1011 |
| C | 12 | 1100 |
| D | 13 | 1101 |
| E | 14 | 1110 |
| F | 15 | 1111 |

* + Binary to Hexadecimal
    - Split the binary bits into groups of four
    - Map each group of four bits to a hexadecimal value
  + Decimal to Hexadecimal
    - Best way is to convert to Binary and then convert to Decimal/Hexadecimal
    - Very complicated to convert straight to/from Decimal

## Prefixes

* 0a 🡪 Decimal
* 0b 🡪 Binary
* 0x 🡪 Hexadecimal

## Max/Min Values of Binary Number

* The largest binary number[[10]](#footnote-10):
  + For any given number of bits, all bits are set to 1
* The smallest binary number:
  + For any given number of bits, all bits are set to 0

## 2’s Compliment

* Take 729 and convert to Binary
* Invert all numbers
* Add 1

## Variable Types

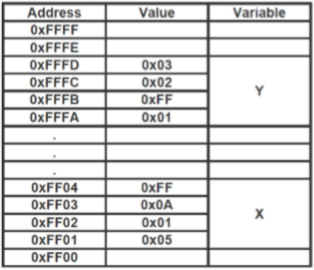
* Char – 1 byte
* Int – 4 bytes
* Long – 8 bytes
* Float – 4 bytes
* Double – 8 bytes

## Memory

* Variables are stored in discrete locations in memory
* The amount of memory required by a variable depends on the type of variable
* In the figure below…
  + Each address is 1 byte
  + Variable 0xACAA is stored at 0xFF02[[11]](#footnote-11)

|  |  |
| --- | --- |
| **Address** | **Value** |
| 0xFF06 |  |
| 0xFF05 |  |
| 0xFF04 |  |
| 0xFF03 |  |
| 0xFF02 | AC |
| 0xFF01 | AA |
| 0xFF00 |  |

* Example
  + X and Y would be integer values or a float
  + If we change the value of that variable, the value at that address would change aswell.



* + More in-depth example in “*2 – Memory & Number Systems\Memory.cpp*”

## Pointers

|  |  |
| --- | --- |
| 0xAAAA | 0000 |
| 0xAAAB | 0000 |
| 0xAAAC | 0000 |
| 0xAAAD | 0100 |
| **.**  **.**  **.**  **.**  **.** |  |

* + Integer X takes up 4 bytes due to being an integer
* Regular int variable may be declared as such
  + int X = 4;
* The pointer to this variable would be declared as
  + int \*ptrX = &X;
    - Define integer pointer X as address X
      * & is the address of the variable
        + i.e. print(&X) = 0xAAAA

We are printing the address of integer X

* + - * \* is the dereference pointer value
        + i.e. print(\*ptrX) = 4

We are printing the value of X at which the pointer is pointing to.

* + - Example in “*2 – Memory & Number Systems\Pointer.cpp*” and *“2 -Memory & Number Systems\passByReference.cpp*”
  + Should I put the \* before the ptr or after the int?
    - int\* is an integer pointer pointing to something
    - \*ptrX means ptrX is a pointer to an int value

# Algorithms and Arrays

## Arrays

* Arrays are used to store collections of data
* Each element of an array…
  1. Must be the same type and size
  2. Is stored contiguously in memory
  3. May be collected and stored over time
  4. Is addressable by its index in the array
* Fixed size
  + Once and array is declared, it are fixed to that size
  + Remember – The character array declared below remains the same size at every step, regardless of string length

char myString[11] = “HelloWorld”;

myString = “Hi!”

myString[1] = “\0”

* What if we need more space?
  + Expensive array doubling
* Types of Operations:
  + Search
  + Add/Insert
  + Delete

## Algorithms

* In any computer program, there is a specific set of instructions that tells the computer what to do
  + This is similar to a recipe in that there is an objective to accomplish, and a set of steps in a specified order
  + These instructions also known as an *Algorithm*[[12]](#footnote-12)
* As an example, an algorithm that puts the following sequence of number is ascending order
  + Would produce an output

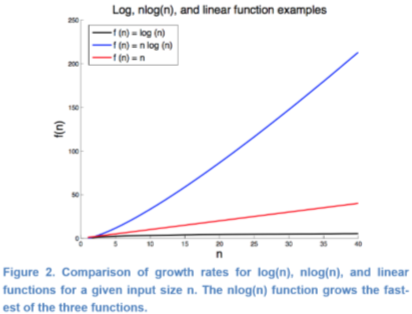
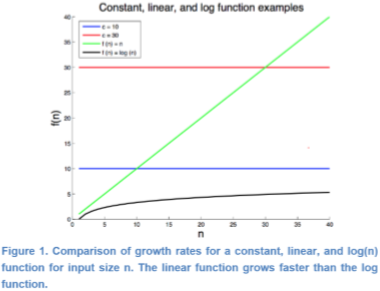
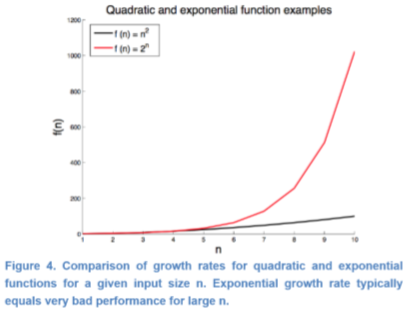
## Components of an Algorithm

* *Pre-Condition* – Conditions that must be true prior to the algorithms execution in order or it to work as defined
  + Can include the inputs to the algorithm and the restrictions on the typed and range of values on the inputs
  + Can also include other dependencies, such as other algorithms that need to execute first
* *Post-Condition* – The expected changes, or the return value, after the algorithm executes

## Evaluation of an Algorithm

* + Correctness
    - The algorithm returns the desired result or performs the desired action appropriately
  + Cost
    - Various ways to measure, application specific
    - Two standard metrics
      * Runtime (follow n)
      * Memory usage

## Runtime Analysis

* Constant function –
  + This function has a constant runtime, such that the output is not dependent of the value or size of the input n
  + Examples:
    - Variable assignment
    - Inserting to the front of a linked list
    - Overwriting element in an array
    - Accessing element in hash table
* Logarithmic Function –
  + The logarithmic runtime will frequently be base 2
  + vs vs
  + Examples:
    - Minimum height of a Binomial Search Tree (BST)
    - Searching in a BST
* Linear Function –
  + The value and size of *n* directly correlated to the runtime
  + What about or ?
    - Constant do not detract from linear runtime
  + Examples:
    - Traversing elements in linked list
    - Traversing elements of a 1D array
    - Shifting elements in a 1D array
* N-Log-N Function –
  + Log(n) repeats n times
  + Examples:
    - N searches on BST
    - Merge sort
* Quadratic Function –
  + NOT GOOD!!
  + Examples:
    - Traversing a 2D matrix with n rows and n columns
    - Algorithms with nested *for* *loops*
    - Bubble Sort
* Polynomial Function –
* Exponential Function –
  + b is some constant we call the base, most commonly 2
  + ****WORST CASE SCENARIO!!

## Array Search Function Example

* Pre-condition
  + *A* is an array
  + *v* is the search value, must be same type as elements in *A*
* Post-condition:
  + Return the index x where A[x] = v
* Pseudocode and Runtimes:

searchArray(A,v)

found = false *constant (c)*

index = -1 *constant (c)*

x=0 *constant* *(c)*

while(!found and x <= A.end) *linear* *(n)*

if A[x] == v *constant* *(c)*

found = true *constant (c)*

index = x *constant* *(c)*

else *constant* *(c)*

x++ *constant* *(c)*

return index

* + Example in “*3 – Algorithms and Arrays\arrays.cpp*”
  + Total Runtime:
    - Worst case O(n) – Linear
  + If this was a 2D array, need a nested loop inside while() loop, making it O(n2) – Quadratic

## Adding Elements to an Array

* Remember: elements of an array are stored contiguously
* Adding at the end is easy if there is room
* Pre-Conditions:
  + An array – A
  + A new value to insert – v
  + The index you want to store the new value in – index
  + The number of elements already in the array – numElements
* Post-Conditions:
  + Array A is updated such that A[index] = v
  + numElements has been incremented
* Algorithm Complexity: What is the runtime of this algorithm?

insertArrayElements(A, v, index, numElements)

for x = numElements-1 to index

A[x+1] = A[x]

A[index] = v

numElements++

* + Example in “*3 – Algorithms and Arrays\arrays.cpp*”
  + Runtime is linear – O(n)

## Deleting Elements from an Array

* Pre-Conditions:
  + An array – A
  + The index containing the value to delete – index
  + The number of elements already in the array - numElements
* Post-Conditions:
  + The value at A[x] is overwritten
  + numElements is decreased by 1
* Algorithm Complexity: What is the runtime of this algorithm?

deleteArrayElements(A, v, numElements)

for x = index to numElements-2

A[x] = A[x+1]

numElements = numElements-1

* + Example in “*3 – Algorithms and Arrays\arrays.cpp*”
  + Runtime is linear – O(n)

## Doubling an Array

* Pre-conditions
  + A and B are arrays of the same type
  + B >= A
* Post-Conditions
  + Every index of B is the same as the corresponding index in A
    - B[0] = A[0], B[1] = A[1], etc…
* Algorithm Complexity: What is the runtime of this algorithm?

copyArray(A,B)

for x = 0 to A.end

B[x] = A[x]

* + Runtime is linear – O(n)

doubleArray(A)

B.length = A.length\*2

For x = 0 to A.end

B[x] = A[x]

return B

* + Example in “*3 – Algorithms and Arrays\arrays.cpp*” and *Assignment2.cpp*
  + Runtime is linear – O(n)

## Memory

* The Stack
  + Local variables
  + Limited space
    - Carefully managed to preserve space
* The Heap
  + Much larger than stack
  + Used for storing variables created dynamically during runtime
  + Variables using pointers
    - Why? (intuition?)

## Heap

* Features:
  + Allocated memory stays allocated until specifically de-allocated
    - Every call to malloc() must have a corresponding free()
  + Dynamically allocated memory must be accessed through a pointer
    - We don’t necessarily need pointers to work in the stack
  + Allocate large arrays, structures, and objects on the heap

# Classes

## Classes

* A *class* is a template for a complex data type
* Instances of a class are called an *object*
* Features of Classes:
  + Complex type
  + Functions included in class definition, called *methods*, control access to member variables
  + Variables and methods can be public or private
  + Constructor called to create instance of class
  + Destructor to destroy class (free memory!)
  + Many more exist, but these are the foundational features

## Class vs Struct

* Stuct:
  + How do me maintain data integrity?
    - If we’re setting the value for month, what prevents us from entering a 13?
* A Struct is like a Class, but:
  + Structs only contain data
  + Classes maintain member variables along with functions

## Class Methods

* Public
  + Can be accessed outside of the class
* Private
  + Can only be accessed by class methods
    - i.e. A private variable cannot be viewed in the main function directly

## Classes Example

* Create a class called date with three private members: month, day, year
  + The constructor takes 3 integer arguments for month, day, year
  + Public method to print month, day, year

class Date

private:

int month

int day

int year

public:

Date(int m, int d, int y)

month = m

day = d

year = y

void printDate()

cout << month << "/" << day << "/" << year << endl

int main()

Date myDateObject1 = Date(6,12,2018)

Date myDateObject2(6,12,2018)

myDateObject1.printDate()

myDateObject2.printDate()

return 0

* + Output: 6/12/2018
  + Example in “*4 – Classes\classes.cpp*”

## Updating Private Variables

* How can we update a private variable?
  + Create a public method that controls access
* i.e. Add the following method to the Date class

void setMonth(int m)

if m < 12 and m < 13

month = m;

else

cout << “out of range” << endl;

* + Example in “*4 – Classes\classes.cpp*”

# Linked Lists

## Linked Lists

* Two types:
  + Singly Linked List
    - Each node stores the pointer to the next node

Screen Clipping

* + Doubly Linked List
    - Each node stores the pointer to the next as well as the previous node

Screen Clipping

* Each element in the list is called a *node*
  + *Head* – the first node in a linked list
  + *Tail* – The last node in a linked list
* Typically
  + Only the head of a linked list is known
  + Sometimes, the tail is also stored

## Linked List ADT[[13]](#footnote-13)

Linked List

private

head

tail

public

Init()

insertNode(previousValue, value)

search(value)

traverse()

deleteNode(value)

deleteList()

## Node

* A node may be represented with either structs or classes

struct singleNode

int key

singleNode\* next

struct doubleNode

int key;

doubleNode\* next

doubleNode\* previous

* + Examples in “*5 – Linked Lists\singleyLinkedList.cpp*” and “*5 – Linked Lists\doublyLinkedList.cpp*”

## Traversal

* In an array, all elements are stored sequentially and are directly addressable by their index
* Algorithm:
  + Pre-Conditions
    - The head node is defined in the linked list ADT or included as an argument
  + Post-Conditions:
    - Values of the nodes in the list are displayed
  + Pseudocode:

tmp = head

while (tmp != NULL)

print tmp.key

tmp = tmp.next

## Search

* To search a linked list, you simply traverse from the head until the desired value is found

tmp = head

returnNode = NULL

found = FALSE

while(!found and tmp != NULL)

if(tmp.key == value)

found = true

returnNode = tmp

else

tmp = tmp.next

return returnNode

## Insert

* Inserting into a linked list presents the opportunity to lose pointers to nodes we still need
* There are 3 cases to consider:
  1. Inserting a node at the head
     + Make a new node, set a pointer from the new head to the original head – O(c)
  2. Inserting a node at the tail
     + Search to the end of the node, make a new node, set a pointer from the original tail to the new tail – O(n)
  3. Inserting a node in the middle
* Algorithm
  + Pre-conditions:
    - leftValue is a valid key value for a node in the list, or NULL
    - value is a valid key value
  + Post-conditions
    - The new node has been added to the list after the leftValue node
  + Algorithm

left = search(leftValue)

node.key = value

if(left == NULL) // head node

node.next = head

head = node

else if(left.next == NULL) // tail node

left.next = node

tail = node

else // middle node

node.next = left.next

left.next = node

## Delete

* Deleting a node from a linked list is a matter of bypassing the pointer to the node you with to delete and then freeing that memory
* Just as with insert, be sure to perform the requisite steps in order to prevent losing reference to your pointers
* Same 3 Cases:
  1. Deleting the node at the head
  2. Deleting the node at the tail
  3. Deleting the node in the middle
* Generalized Delete Algorithm
  + Pre-Conditions
    - head pointer is set in the liked list
    - value is a valid search parameter
  + Post-Conditions
    - Node where the key equals the value has been deleted from the list
  + Algorithm

if(head.key == value) // delete the head

tmp = head

head = head.next

delete tmp

else // delete middle or tail

left = head

tmp = head.next

found = false

while (tmp != NULL and !found)

if (tmp.key == value)

left.next = tmp.next

if(tmp == tail)

tail = left

delete tmp

found = true

else

left = tmp

tmp = tmp.next

## Complexity

* Insert
  + Head/Tail: O(1)
  + End of singly liked: O(n)
  + Middle/Overall: O(n)
* Search
  + O(n)
* Delete
  + Head/Tail: O(1)
  + Middle/Overall: O(n)

# Stacks & Queues

## Stacks

* *Stacks* store a collection of elements and restricts which element may be assessed at any time
  + Operate on a last in, first out principle (LIFO)
  + Think about a stack of plates…
    - You have 3 plates stacked on top of each other
    - To place a new plate onto the stack, it must go on top
    - To remove a place from the stack, it must come from the top
* Placing a new element onto the stack is called a *push*
* Removing an element from the stack is called a *pop*
  + Think about reading a sentence:
    - Push each word you read onto the stack
    - After reading the whole sentence, pop each word from the stack
    - Compare this to the original sentence
* Example

|  |  |  |  |
| --- | --- | --- | --- |
| The | quick | brown | fox |

* + As you read the sentence, push onto a stack

|  |
| --- |
| fox |
| brown |
| quick |
| The |

* + - Each word you push is stacked onto the previous word
  + You can pop the words back into the array, starting from the top

|  |  |  |  |
| --- | --- | --- | --- |
| fox | brown | quick | The |

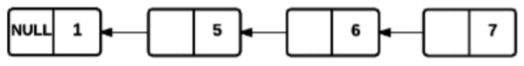
* + - This is great for reversing arrays
  + Stacks are useful to check if you have a palindrome: racecar

## Stack Implementation

* Arrays
  + An array can be turned into a stack by implementing restrictions on where you can add/remove elements
  + The “top of a stack implemented from an array is set to be the numElements-1 index



* Linked Lists
  + Linked lists may also be used to implement stacks
  + Each node represents a data element of the stack
  + Each node stores a pointer to the *prev* node in the list
    - The bottom of the list has a *prev* pointer to NULL
  + The *top* of the stack is the pointer to a node



## Stack ATD

Stack:

private:

top

data

maxSize

public:

Init()

isFull()

isEmpty()

push(value)

pop()

* Example in *“6 – Stacks & Queues\stacks.cpp”*

## Queues

* A queue is similar to the other data structures we’ve covered
  + Stores collection of elements
  + Restricts which element may be accessed
* Unlike stacks, queues are first in, first out
  + Think of the waiting queue at the DMV
    - Get a ticket
    - First ticket gets served first
* Words are added at the *tail*
* Words are removed from the *head*
* The position of the *tail* and *head* move as elements are added

## Queues ADT

Queue:

private:

head

tail

data

queueSize

maxQueue

isEmpty()

isFull()

public:

Init()

enqueue(value)

dequeue()

* Example in “6 – Stacks & Queues\queue.cpp”

1. i.e. stacks, queues, lists, trees, etc. [↑](#footnote-ref-1)
2. i.e. linking, arrays, etc. [↑](#footnote-ref-2)
3. More on Queues in “Queues & Stacks” [↑](#footnote-ref-3)
4. More on Binary Search Trees in “Binary Search Trees” [↑](#footnote-ref-4)
5. More on Arrays in “Algorithms & Arrays” [↑](#footnote-ref-5)
6. *Contiguous* – sharing a common border; together in sequence [↑](#footnote-ref-6)
7. More on Stacks in “Stacks & Queues” [↑](#footnote-ref-7)
8. More in “Linked Lists” [↑](#footnote-ref-8)
9. A dynamic data structure refers to an organization or collection of data in memory that has the flexibility to grow or shrink in size, enabling the programmer to control exactly how much memory is utilized. [↑](#footnote-ref-9)
10. The largest binary number for 32 bits is 2,147,483,647 and for 64 bits is 9,223,372,036,854,775,807 [↑](#footnote-ref-10)
11. The address of the variable is always the first byte that the variable is stored in, despite it taking up multiple bytes. [↑](#footnote-ref-11)
12. *Algorithm* – A defined set of steps that are followed to solve a problem [↑](#footnote-ref-12)
13. A Linked List is defined in a class, but the nodes must be defined in a struct. (next page) [↑](#footnote-ref-13)