Data Structures - Automation - Problem Solving

Algorithms - in - C++

Programming & Design

CodeWell Academy0

with RMZ Trigo

CodeWell Academy()
and
R.M.Z. Trigo
present:

Algorithms

C++ Edition

Data Structures, Automation & Problem Solving

w/ Programming & Design

Algorithms Series

Table of Contents

Introduction

========== ======

Welcome to the Path of Mastery

We thank you for purchasing & downloading our work, “Algorithms”.

We understand you want to expand your Programming skills to a deeper, more advanced level. You want your apps to be more sophisticated; you want them to auto-generate results or processing tons of information for you - all in the fastest, most efficient way possible.

But before all that can happen, you need a great start.

Hence, you’ve come to the right place…

Algorithms

Algorithms can be as simple as a few lines of good code, or as complex as an Artificial Intelligence bot.

How does a Search Engine give you thousands of web pages based on the phrases you enter? How does a search engine finish your search keywords before you even finish typing? How are puzzles randomly created - or automatically solved? How is a list of names instantly sorted in alphabetical order?

To understand computing is to understand algorithms. To craft a good algorithm is to give a computer system a really good set of procedures. And these procedures create answers, manage data, or generally do some work for you - all in the most efficient instructions possible.

And, more importantly, to craft a good algorithm is to understand how all the data will be arranged and laid out. Then, how quickly and easily the algorithm or computer can pick a data item and process it.

The Master Structure

At this point, you pick up where you’ve left off in your coding journey. You know the basic data types, functions, and other programming terms. You may have started to code using a different programming language; or you may have chosen this book’s main language. Either way, you’re here to enhance your skills.

We start with computing a list of data items. We take a good look at the data and how it behaves, then identify the proper data structure that comes along with it. Then we add the functions and methods that are essential to the data structure. And as the book progresses over time, we start to take a look at more complex data arrangements.

Pseudocode Reference

This book comes in its own main language and an easily identifiable pseudocode. The pseudocode is included to teach you one major thing: to recognize the common procedures

between the two codes. You can find them at the Reference Section, as well as throughout the book. Afterwards, once you master the material in the book’s main language, you can use the pseudocode to help you transition into your next language.

Overall, good luck and code well. This is the start of something good.

========== ======

C++ Introduction

========== ======

Software System Special

C++ is one of the most widely used programming languages not just to develop games and apps, but even operating systems and hardware programming. C++ programs can use algorithms for managing and manipulating not only lists and collections, but system aspects (like memory) as well. This is because of the procedural speed and memory management in C++.

C++ Workshops:

Just like our other works, these workshops are yours to complete in whichever way you like.

They’re designed to put recent concepts into real-life practice. Yet they also give you flexibility and critical thinking along the way.

And of course, flexibility and deep critical thinking are key programmer traits!

Find them throughout the book!

========== ======

WAIT!!

It’s best to know the coding fundamentals before you move forward.

If you feel like you need to, you can start with our Programming: Master’s Edition Series

Click here to view.

We also have Sample Chapters throughout the book to help you out.

Part 1: Identifying the Data Types

Chapter 1a: Abstract Data Types: The Stack

The Stack

A stack is literally what it is - a stack of objects.

And as an abstract Data Type, the stack would hold a list of data objects for later processing.

The data objects in a stack would be inserted and retrieved in a Last-In-First-Out process.

A Stack will usually have two main functions defined along with it: push (aka insert) and pop (aka remove/retrieve).

How the Stack Works

Think of a stack as a pile of plates “stacked” on top of one another. And each plate had a letter in it. The top plate would be the easiest to retrieve.

So retrieving a data object from a Stack would be the last object that was entered.

For example, let’s have a ‘stack’ of dishes as an array:

// stackOfPlates are a collection of strings

// Each string resembles a Plate

string stackOfPlates[5] = {“A”, “B”, “C”, “D”}

If you wanted to insert a new plate in this stack, it would be placed at the end of the list.

Now, for example, let’s take a look at this stack:

string stackOfPlates[5] = {“A”, “B”, “C”, “D”}

Then, let’s enter a plate called “E”. To enter items into a stack, you need a function to insert it into the stack. This function is usually called push().

stackOfPlates.push(“E”)

Afterwards, here’s what the stack would look like:

string stackOfPlates[5] = {“A”, “B”, “C”, “D”, “E”}

If we wanted to retrieve a plate from this stack, the LAST item in the collection would be removed from the stack. Then that item would be given. You need a function to remove and retrieve an item from the stack. This function is called pop().

// This would print E

print(stackOfPlates.pop())

// This would print D

print(stackOfPlates.pop())

// This would print C

print(stackOfPlates.pop())

Then, here's what the stack would look like afterwards:

string stackOfPlates[5] = {"A", "B"}

When to Use the Stack

Carefully look at how a list of data objects should be processed. Once you notice it uses a Last-In-First-Out process for any data objects that enter or leave, use a stack.

Also, complex algorithms and functions use stacks. If a computer processes a procedure as a stack, it will "stack" all the functions that are called. Then it will proceed to complete each function called, starting with the most recent (you'll learn more about all this in-depth later).

For future reference, picture yourself in a tech interview. If you ever are asked how a certain function's procedure works, you'll be very fortunate to know how a stack works and how that function will proceed from start to finish.

However, recall that stacks normally use lists - either arrays or linked lists. So which one do you choose? Recall these lines from the Master's Handbook Series:

Linked Lists = any size, faster add & remove

Arrays = fixed size, faster access/process

So depending on which data structure you use, the next chapters show you how you implement a stack.

Chapter 1b: The Stack, as an Array

The Data Structure

You'll need the actual array, as well as two integers: the max size and the current item count.

// Abstract Data Type: Stack, as an Array

// The Stack Has:

// - the Actual Array (with Size and Data Type)

// - the Max Size of the Stack (Integer)

// - the Current Item Count (Integer)

CompositeStructure ArrayStack {

Integer MAXSIZE

Integer COUNT

(choose a data type) ARRAY[MAXSIZE]

}

Now, let's look at how the Push() and Pop() methods work, in detail:

Array Stack: Push() Function

The Push() function from an Array Stack adds items into that stack. It does this in the following procedure, in this particular order:

1) Check if there's room in the array for the extra item

2) If there's no room in the array, either report a stack overflow error (if fixed-size array) or increase the array size (if variable array)

3) If there's room in the array, add the item to the stack. Set the new item's index as the current Item Count (so the item is the very last item in the Stack)

4) And Finally, Increment the Item Count

After the push() procedure successfully inserts an item, you should have that item at the end of the array and the item count should be increased by 1.

Array Stack: Pop() Function

In contrast to the Push() method, the Pop() method from an Array Stack removes an item from that stack. The function returns that item as the Function Output. It does this in the following procedure, in this particular order:

1) Check the Array Count if there are any items to pop.

2) If not, either return NULL or report a stack underflow error

3) If there’s an item to pop, Decrement the Item Count first. We’re going to remove an item!

4) Then,create a Local Variable. Set it to be your return item (it should be the item at the end of the stack!) Remove the item at the End of the Stack afterwards

5) And Finally, the function returns your item and ends.

After the pop() procedure successfully removes and retrieves an item, the item should be the item that was at the end of the array. Also, the item count should be decreased by 1.

Chapter 1c: The Stack, as a Linked List

The Data Structure

You'll need one Linked List node - the Head - as well as an integer to keep track of the Item Count. Luckily, there are no Stack Overflows as a Linked List. Chances are, you may have to design the Linked List Nodes as well.

// A Singly Linked List Node has:

// - the Next Node (Point to the Next Node)

// - the Data Item (choose what data type you want)

CompositeStructure Node {

Node NEXTID

(choose a data type) ITEM

}

// Abstract Data Type: Stack, as a Linked List

// The Stack Has:

// - a Head (Linked List Node)

CompositeStructure LinkedListStack {

Node HEAD

Integer COUNT

}

Now, let's look at how the Push() and Pop() methods work, in detail:

Linked List Stack: Push() Function

The Push() function from a Linked List Stack adds items into that stack. It does this in the following procedure, in this particular order:

1) Make a new Local Variable. Set it as a Node. This will be the New Linked List head.

2) Set your input item to be the New Head Node's data

3) Set the previous head's next node to the New Head.

4) Set the New Head as the Linked List's Stack Head

5) And Finally, Increment the Item Count

After the push() procedure successfully inserts an item, you should have that item at the head of the linked list and the item count should be increased by 1.

Linked List Stack: Pop() Function

In contrast to the Push() method, the Pop() method from a Linked List Stack removes an item from that stack. The function returns that item as the Function Output. It does this in the following procedure, in this particular order:

1) Check if the Head Node isn't Null (if the Head Node Exists)

2) If so, either return NULL or report a stack underflow error

3) If there's an item to pop, create a Local Variable. Set it to be your soon-to-be return item (it should be the Head Node!)

4) Get the Node NEXT to the Head. This will NOW be the new LinkedList Head.

5) Decrement the Item Count

6) And Finally, the function returns your item and ends.

After the pop() procedure successfully removes and retrieves an item, the item should be the item that was the Linked List Head. Also, the item count should be decreased by 1.

C++ 01a: The Stack, as an Array

Here, we start playing around with some basic data structures in C++ code. You may also use online IDE's such as rextester.com, ideone.com, or codepad.org.

Future algorithms start to become more complex, so understand these well!

The Data Structure, in C++

In C++, we're going to define our ArrayStack as a data class. First, we define our Class and its fields. Later, we'll include the methods Push() and Pop()

By default, we'll set the Array Size to 10. There is also the constructor included. In your main code, if you need to create an instance of an array stack, simply call the constructor method.

```
// Abstract Data Type: Stack, as an Array
// The Stack Has:
// - the Actual Array (with Size and Data Type)
// - the Max Size of the Stack (Integer)
// - the Current Item Count (Integer)
class ArrayStack {
int MAXSIZE = 10;
int COUNT;
<choose a data type> ARRAY[10];

public:
// INPUT: -nothing
// OUTPUT: - nothing
// EFFECT: Constructor: Array Stack, with its max size as Initial Value
ArrayStack() {
this->COUNT = 0;
};

// INSERT PUSH() FUNCTION HERE
//_____

// INSERT POP() FUNCTION HERE
```

//_____

};

Note: the data type can be whatever you wish. It depends on the data you wish to process.

Array Stack: Push() Function, in C++

Remember the procedure behind the push() function. First note the steps it takes to enter the data into the stack, then check that the steps actually process the way you intend it to be.

In this version, we simply get our system to print “Stack Overflow” and end the function if we try to enter too many items in the array

We’ve also included a few print lines that show you that the function actually pushed the value into the array

// INPUT: - an item (some Data Type)

// OUTPUT: - nothing

// EFFECT: inserts an item (some Data Type) onto our Array Stack

void push(<chosen data type> item) {

// 1) Check if there’s room in the array for the extra item

if (this->MAXSIZE <= this->COUNT) {

// 2) If there’s no room in the array,

// print a Stack Overflow and end the function

std::cout << “Stack Overflow!\n”;

return;

}

// 3) If there's room in the array, add the item to the stack.

// Set the new item's index as the current Item Count

this->ARRAY[COUNT] = item;

// PROOF that it inserted:

std::cout << "New Item is " << this->ARRAY[COUNT] << " At Index "<< COUNT << "\n";

// 4) And Finally, Increment the Item Count

this->COUNT += 1;

// PROOF of incremented Item Count:

std::cout << "Count is " << this->COUNT << "\n";

};

OPTIONAL: Array Size Increaser (phase 1.2)

If you choose to go with a variable-size array, include this method along with the others. This version will double the array size where needed. Take note of the procedure steps. Array sizes, by definition, are fixed. However, this will simply replace the old array with a larger one - entering each item one by one in its original order.

Array Stack: Pop() Function, in C++

Just like the Push() function, take note of the procedural steps in how a Pop function is performed. Here, we implement it in C++.

If pop() is called when there is no items in the stack, we'll just print out a message and end the function.

// INPUT: - nothing

// OUTPUT:- an item (some Data Type)

// EFFECT: removes and retrieves an item (some Data Type) from our Array Stack

<chosen data type> pop() {

// 1) Check the Array Count if there are any items to pop.

if (this->COUNT <= 0) {

```
// 2) If not, print a message and return null
std::cout << “Stack Underflow!\n”;
return NULL;
};

// 3) If there’s an item to pop, Decrement the Item Count first.
// We’re going to remove an item!
this->COUNT -= 1;



// 4) Then,create a Local Variable.
// Set it to be your return item
// (it should be the item at the end of the stack!)
<chosen data type> r = this->ARRAY[COUNT];

// PROOF that it loaded the end item:
std::cout << “Loaded is ” << this->ARRAY[COUNT] << ” at index “<< this->COUNT << “\n”;

// Remove the item at the End of the Stack afterwards
this->ARRAY[COUNT] = NULL;

// 5) And Finally, the function returns your item and ends.
return r;
}



};
```

C++ 01b: The Stack, as a Linked List

The Data Structure, in C++

In C++, we use pointers to link the nodes together. This way, each node has the memory address of the next node. Instead of duplicating nodes and taking up data space in the computer memory, we just “point” out to what the next node is.

We have class definitions for both a node and the linked list. We also have constructors for both a Node and the Linked List Stack. You need both of them together, as the Stack requires nodes to be made

```
// A Singly Linked List Node has:
// - the Next Node (Point to the Next Node)
// - the Data Item (choose what data type you want)
class Node {
public:
Node* NEXT;
<chosen data type> ITEM;
// INPUT: - a data item
// OUTPUT: - nothing
// EFFECT: Constructor: Linked List Node, with the Data Item as its input
Node(<chosen data type> item) {
this->ITEM = item;
};
};
```

```
// Abstract Data Type: Stack, as a Linked List
// The Stack Has:
// - a Default Head (Linked List Node)
// - an item count (integer)
class LinkedListStack {
Node* HEAD;
```

int COUNT;

public:
// INPUT: - nothing
// OUTPUT: - nothing
// EFFECT: Constructor: Linked List Stack,
LinkedListStack() {
this->HEAD = new Node(<data with chosen data type>);
this->COUNT = 0;
};

// INSERT PUSH() FUNCTION HERE
//_____

// INSERT POP() FUNCTION HERE
//_____
};

For advanced users, the item count is optional. However, it does come in handy in optimizing your data structures and operations, as well as helping it become more robust.

Linked List Stack: Push() Function, in C++
// INPUT: - an item (some Data Type)
// OUTPUT: - nothing
// EFFECT: inserts an item (some Data Type) onto our Linked List Stack
void push(<chosen data type> INPUT) {

// 1) Make a new Local Variable. Set it as a Node.
// This will be the New Linked List head.
Node NEWHEAD = new Node(INPUT);*

// 2) Set your input item to be the New Head Node's data
// (The Constructor does this)

// 3) Set the previous head's next node to the New Head
NEWHEAD->NEXT = this->HEAD;

// 4) Set the New Head as the Linked List's Stack Head
this->HEAD = NEWHEAD;

// PROOF that it inserted:
std::cout << "Head item is:" << HEAD->ITEM << "\n";
std::cout << "Next Head item is:" << HEAD->NEXT->ITEM << "\n";

// 5) Increment the Item Count
this->COUNT += 1;
};

Linked List Stack: Pop() Function
// INPUT: - nothing
// OUTPUT:- an item (some Data Type)
// EFFECT: removes and retrieves an item (some Data Type) from our Linked List Stack
<chosen data type> pop() {

// 1) Check if the Head Node isn't Null (if the Head Node Exists)
if (this->HEAD->NEXT == NULL) {

// 2) If so, print a message and return null
std::cout << "Stack Underflow!" << "\n";
return NULL;
}

```
// 3) If there's an item to pop, create a Local Variable.
// Set it to be your soon-to-be return item
// (it should be the Head Node!)
<chosen data type> r = HEAD->ITEM;

// PROOF that it loaded:
std::cout << "loaded item is:" << r << "\n";

// 4) Get the Node NEXT to the Head.
// This will NOW be the new LinkedList Head.
this->HEAD = this->HEAD->NEXT;

// PROOF that head has been moved:
std::cout << "Head item is now: " << HEAD->ITEM << "\n";

// 5) Decrement the Item Count
this->COUNT -= 1;

// 7) And Finally, the function returns your item and ends.
return r;
}
```

C++ Workshop #1

Stacks At Work

Now we're going to see the Stack in action.

Part 1:

1) Set up an IDE of your choice (online IDE's such as rextester.com, ideone.com, or codepad.org will do as well)

2) In the above portion of your code, make sure you have these lines included:

#include <iostream>

using namespace std;

These allow your C++ code to use strings and other basic data types.

3) Copy-paste either the ArrayStack or the LinkedListStack class from the previous chapters, along with their respective push() and pop() methods. You may remove the Print PROOF statements if you wish.

4) In all places of your code that require you to choose your data type, set them as a string:

<chosen data type> ====> string

5a) If you chose ArrayStack, to avoid errors, change the following lines from the pop() method below:

this->ARRAY[COUNT] = NULL;

to

this->ARRAY[COUNT].clear();

5b) If you chose LinkedListStack, to avoid errors, change the following lines from the constructor method below:

this->HEAD = new Node(<data with chosen data type>);

to

this->HEAD = new Node("BOTTOM NODE");

6) Afterwards, copy these lines as your main() method:

// ————————————-

int main()

{

// Fill Blank as ArrayStack or LinkedListStack**

*// DO NOT forget the star! **

```
___ m;

// Fill blank as ArrayStack() or LinkedListStack()
// depending on whatever you chose previously
// It's a Constructor: DO NOT forget the parentheses! ()
m = new ___;

string s;
m->push("A");
m->push("B");
m->push("C");
m->push("D");
string d = m->pop();
d = m->pop();
std::cout << "local var d is C? " << d << "\n";

string c = m->pop();
std::cout << "local var c is B? " << c << "\n";

m->push("Y");
m->push("D");
m->push("D");
m->push("U");
m->push("B");
m->push("T");
m->push("S");
m->push("E");
m->push("B");
m->push("X");

for(int i = 0; i < 9; i++) {
```

std::cout << “Popped: ” << m->pop() << “\n”;

};

d = m->pop();

d = m->pop();

std::cout << “Trigger Stack Underflow: ” << m->pop() << “\n”;

};

// ————————————-

What happened?

There should be two lines printed that say

“ local var d is C? C

local var c is B? B ”

Then, we attempted to spell BESTBUDDY.

If you used an ArrayStack, we managed to get all 9 letters in, but not the “X”. Why? Because a stack overflow has just happened. We tried to get the extra item “X” in, but we couldn’t. The stack was full.

If you used a LinkedListStack, we managed to input the “X”. However, the “Y” in BESTBUDDY wasn’t popped. Because a Linked List has no limits, the “X” was popped as part of the 9 pops - spelling XBESTBUDD.

Afterwards, we tried to pop a few more items.

But eventually, the stack was empty. So what happened? A stack underflow occurred.

Part 2: Switching List Types

If you switch to the other list type (array or linked list), you should have the same results. Try it out, and note the differences between the two data structures.

Chapter 2a: Abstract Data Types: The Queue

The Queue

A Queue is also a list or sequence of items. Each item awaits for its turn to be processed - as if it were a person lining up for something.

The data objects in a queue would be inserted and retrieved in a First-In-First-Out process.

A Queue will usually have two main functions defined along with it: enqueue (insert to the lineup, at the back) and dequeue (remove from the lineup).

How the Queue Works

Think of a queue as a lineup. For anything. It could be the lineup at your bank, at your favourite fast food restaurant, the nightclub, anything that resembles a lineup.

And in a lineup, someone enters the line at the back. As the lineup grows, more people enter the line at the back. Whoever is at the front of the line is serviced - they deal with customer service, have their order taken, get in the nightclub, anything the lineup does. And whenever someone is serviced, the lineup becomes smaller.

A queue works essentially the same way. You input data items into a lineup, with the most recent data items at the back of the line. When data items are retrieved, the oldest item is retrieved first.

When to Use the Queue

Whenever you have a list of data items and they must be processed in a ‘lineup matter’, use the Queue. Once you notice it uses a First-In-First-Out process for any objects it takes in, use a queue.

Online Video Games are a great example. If you had to log into any game server to play, your login request would be sent to a “lineup” of login requests from everyone else who wants to play too. The most recent request is always at the end of that lineup. The game server processes each login request to have its player log into the server - one by one, starting with the front of the lineup.

Also, recall that queues normally use lists - either arrays or linked lists. Again, which one do you choose?

By default, use Linked Lists for queues - as queueing and dequeueing are computed in constant time (O(1)) - the quickest tier for algorithm speed.

On the other hand, use Arrays for queues if any data items within that queue need to be accessed often. For Array Queues, you’ll use Circular Arrays - an array with the starting index (0) right after the ending index (the array size - 1). This is so that data items are added and removed without having to shift EVERY data item left or right all the time(which would really strain your program and computer effort).

And Circular Array Queues can be just as fast - queueing and dequeueing can also be in constant time. You just have the extra flexibility to edit the lineup as you please.

So depending on which data structure you use, the next chapters show you how you implement a stack.

Chapter 2b: The Queue, as an Array

The Data Structure

Just like the Array Stack, you'll need your actual array. But this time, we'll need four integers: the front index, the back index, the max size and the current item count.

// Abstract Data Type: Queue, as an Array

// The Queue Has:

// - the Actual Array (with Size and Data Type)

// - the Index of the Front Item (Integer)

// - the Index of the Back Item(Integer)

// - the Max Size of the Queue (Integer)

// - the Current Item Count (Integer)

CompositeStructure ArrayQueue {

Integer FRONT

Integer BACK

Integer MAXSIZE

Integer COUNT

(choose a data type) ARRAY[MAXSIZE]

}

Now, let's look at how the Enqueue() and Dequeue() methods work, in detail:

Array Queue: Enqueue() Function

The Enqueue() function from an Array Queue adds items into that queue. It does this in the following procedure, in this particular order:

1) Check if there's room in the array for the extra item

2) If there's no room in the array, either report a queue overflow error (if fixed-size array) or increase the array size (if variable array)

3) If there's room in the array, we add the item to the queue. Set the new item's index as the current Back Index (so the item is the very last item in the Queue)

4) Increment the Back Index. If the Back Index becomes bigger than or equal to the Array Size, set the Back Index to 0 (it's now at the start of the Circular Array).

4) And Finally, Increment the Item Count

After the enqueue() procedure successfully inserts an item, you should have that item as

the last item in the circular array, regardless of what index it has. Also, the item count should be increased by 1.

Array Queue: Dequeue() Function

The Dequeue() method from an Array Queue removes an item from that queue. The function returns that item as the Function Output. It does this in the following procedure, in this particular order:

1) Check the Array Count if there are any items to dequeue

2) If not, either return NULL or report a queue underflow error

3) If there's an item to dequeue, create a Local Variable. Set it to be your return item (it should be the item at the front of the queue!). Remove the item at the Front of the Queue afterwards

4) Increment the Front Index. If the Front Index becomes bigger than or equal to the Array Size, set the Front Index to 0 (it's now at the start of the Circular Array).

5) Decrement the Item Count

6) And Finally, the function returns your item and ends.

After the dequeue() procedure successfully removes and retrieves an item, the item should be the first item from the array. Also, the item count should be decreased by 1.

Remember: function returns should be the very last thing in a function procedure - as it ends the function.

Chapter 2c: The Queue, as a Linked List

The Data Structure

This is similar to the Linked List Stack. You'll need two Linked List nodes - the Front and the Back. You also need an integer to keep track of the Item Count. You may also have to design the Linked List Nodes as well.

// A Singly Linked List Node has:

// - the Next Node (Point to the Next Node)

// - the Data Item (choose what data type you want)

CompositeStructure Node {

String ID

String NEXTID

(choose a data type) ITEM

}

// Abstract Data Type: Queue, as a Linked List

// The Queue Has:

// - the Front (Linked List Node)

// - the Back (Linked List Node)

CompositeStructure LinkedListQueue {

Node FRONT

Node BACK

Integer COUNT

}

Now, let's look at how the Enqueue() and Dequeue() methods work, in detail:

Linked List Stack: Enqueue() Function

The Enqueue() function from a Linked List Queue adds items into the back of the Queue. It does this in the following procedure, in this particular order:

1) Make a new Local Variable. Set it as a Node. This will be the New Linked List Back.

2) Set your input item to be the New Back Node's data

3) Make sure to set the new back's next node to null

4) Set the New Back as the Linked List's Back of Queue. If this queue was previously empty, the New Node will also be the new Front.

5) And Finally, Increment the Item Count

After the Enqueue() procedure successfully inserts an item, you should have that item at the back of the linked list and the item count should be increased by 1.

Linked List Queue: Dequeue() Function

The Dequeue() method from a Linked List Queue removes the item at the very front of the Queue. The function returns that item as the Function Output. It does this in the following procedure, in this particular order:

1) Check if the Front Node isn't Null (if the Front Node Exists)

2) If so, either return NULL or report a queue underflow error

3) If there's an item to dequeue, create a Local Variable. Set it to be your soon-to-be return item (it should be the Front Node!)

4) Get the Node NEXT to the Front. This will NOW be the new LinkedList Front.

5) Decrement the Item Count

6) And Finally, the function returns your item and ends.

After the dequeue() procedure successfully removes and retrieves an item, the item should be the item that was at the very front of the Linked List. Also, the item count should be decreased by 1.

C++ 02a: The Queue, as a Circular Array

The Data Structure, in C++

Again, by default, we'll set the Array Size to 10.

```
// Abstract Data Type: Queue, as an Array
// The Queue Has:
// - the Actual Array (with Size and Data Type)
// - the Index of the Front Item (Integer)
// - the Index of the Back Item(Integer)
// - the Max Size of the Queue (Integer)
// - the Current Item Count (Integer)
class ArrayQueue {
int FRONT;
int BACK;
int MAXSIZE = 10;
int COUNT;
<chosen data type> ARRAY[10];

public:
// INPUT: - nothing
// OUTPUT: - nothing
// EFFECT: Constructor: Array Queue
ArrayQueue() {
this->FRONT = 0;
this->BACK = 0;
this->COUNT = 0;
};

// INSERT ENQUEUE() FUNCTION HERE
//_____
```

// INSERT DEQUEUE() FUNCTION HERE

//_____

};

Array Queue: Enqueue() Function, in C++

// INPUT: - an item (chosen Data Type)

// OUTPUT: - nothing

// EFFECT: inserts an item (some Data Type) onto our Array Queue

void enqueue(<chosen data type> item) {

// 1) Check if there's room in the array for the extra item

if (this->MAXSIZE <= this->COUNT) {

// 2) If not, print a message and return null

std::cout << "Queue Overflow!\n" ;

return;

}

// 3) If there's room in the array, we add the item to the queue.

// Set the new item's index as the current Back Index

// (so the item is the very last item in the Queue)

this->ARRAY[BACK] = item;

// 4) Increment the Back Index.

// If the Back Index becomes bigger than or equal to the Array Size,

// set the Back Index to 0 (it's now at the start of the Circular Array).

this->BACK +=1;

if (this->BACK >= MAXSIZE) this->BACK = 0;

```
// 5) And Finally, Increment the Item Count
this->COUNT += 1;
};
```

Array Queue: Dequeue() Function, in C++

```
// INPUT: - nothing
// OUTPUT:- an item (some Data Type)
// EFFECT: removes and retrieves an item (some Data Type)
// from our Array Queue
<chosen data type> dequeue() {

// 1) Check the Array Count if there are any items to dequeue.
if (this->COUNT <= 0) {

// 2) If not, print a message and return null
std::cout << "Queue Underflow!\n";
return NULL;
}

// 3) If there's an item to dequeue, create a Local Variable.
// Set it to be your return item
// (it should be the item at the front of the queue!)
<chosen data type> r = this->ARRAY[FRONT];

// Remove the item at the Front of the Queue afterwards
this->ARRAY[FRONT] = NULL;

// 4) Increment the Front Index.
// If the Front Index becomes bigger than or equal to the Array Size,
// set the Front Index to 0 (it's now at the start of the Circular Array).
this->FRONT+=1;
```

```
if (this->FRONT >= MAXSIZE) this->FRONT = 0;

// 5) Decrement the Item Count
this->COUNT -= 1;

// 6) And Finally, the function returns your item and ends.
return r;
}
```

C++ 02b: The Queue, as a Linked List

The Data Structure, in C++

First, we have class definitions for our Linked List Queue - for both a Node and the actual Linked List Queue. We also have constructors for both classes. We use the same Node Class from the LinkedListStack from above. And just like the Stack version, you need both of them together; the Queue requires nodes to be made.

```
// A Singly Linked List Node has:
// - the Next Node (Point to the Next Node)
// - the Data Item (choose what data type you want)
class Node {
public:
Node* NEXT;
<chosen data type> ITEM;
// INPUT: - a data item
// OUTPUT: - nothing
// EFFECT: Constructor: Linked List Node, with the Data Item as its input
Node(<chosen data type> item) {
this->ITEM = item;
};
};
```

```
// Abstract Data Type: Queue, as a Linked List
// The Queue Has:
// - the Front (Linked List Node)
// - the Back (Linked List Node)
class LinkedListQueue {
Node* x = new Node(NULL);
Node* FRONT = x;
Node* BACK = x;
```

int COUNT;

public:
// INPUT: - nothing
// OUTPUT: - nothing
// EFFECT: Constructor: Linked List Queue,
LinkedListQueue() {

this->COUNT = 0;
};

// INSERT ENQUEUE() FUNCTION HERE
//_____

// INSERT DEQUEUE() FUNCTION HERE
//_____
};

Linked List Queue: Enqueue() Function, in C++
// INPUT: - an item (some Data Type)
// OUTPUT: - nothing
// EFFECT: inserts an item (some Data Type) onto our Linked List Queue
void enqueue(<chosen data type> INPUT) {

// 1) Make a new Local Variable. Set it as a Node.
// This will be the New Linked List back.
Node NEWBACK = new Node(INPUT);*

// 2) Set your input item to be the New Back Node's data
// (already done in Constructor)

// 3) make sure to set the new back's next node to null
NEWBACK->NEXT = NULL;

// 4) Set the New Back as the Linked List's Back of Queue
this->BACK->NEXT = NEWBACK;
this->BACK = NEWBACK;

// If this queue was previously empty,
// the New Node will also be the new Front
if (this->COUNT <= 0) {
this->FRONT = NEWBACK;
}

// 5) And Finally, Increment the Item Count
this->COUNT += 1;

Linked List Queue: Dequeue() Function, in C++
// INPUT: - nothing
// OUTPUT:- an item (some Data Type)
// EFFECT: removes and retrieves an item (some Data Type) from our Linked List Queue
<chosen data type> dequeue() {

// 1) Check if the Front Node isn't Null (if the Front Node Exists)
if (this->COUNT <= 0) {

// 2) If so, either return NULL
// or report a queue underflow error
// (we do both here)
std::cout << "Stack Underflow!" << "\n";
return NULL;

```
}

// (assume error() reports an error for you)


// 3) If there's an item to dequeue, create a Local Variable.
// Set it to be your soon-to-be return item
// (it should be the Front Node!)
<chosen data type> r = FRONT->ITEM;

// PROOF: return item loaded
std::cout << "Return Item Loaded: " << r << "\n";


// 4) Get the Node NEXT to the Front.
// This will NOW be the new LinkedList Front.
this->FRONT = this->FRONT->NEXT;

// 5) Decrement the Item Count
this->COUNT -= 1;

// 7) And Finally, the function returns your item and ends.
return r;
}
```

C++ Workshop #2

Message Queue

Now we're going to see the Queue in action.

Part 1:

1) Set up an IDE of your choice (online IDE's such as rextester.com, ideone.com, or codepad.org will do as well)

2) In the above portion of your code, make sure you have these lines included:

#include <iostream>

using namespace std;

These allow your C++ code to use strings and other basic data types.

3) Copy-paste either the ArrayQueue or the LinkedListQueue class from the previous chapters, along with their respective enqueue() and dequeue() methods. You may remove the Print PROOF statements if you wish.

4) In all places of your code that require you to choose your data type, set them as a string:

<chosen data type> ====> string

5a) If you chose ArrayQueue, to avoid errors, change the following lines from the dequeue() method below:

this->ARRAY[FRONT] = NULL;

to

this->ARRAY[FRONT] = "";

5b) If you chose LinkedListQueue, to avoid errors, change the following lines from the constructor method below:

Node* x = new Node(NULL);

to

Node* x = new Node("DEFAULT");

6) Afterwards, copy these lines as your main() method:

int main()

{

// Fill Blank as ArrayStack or LinkedListStack**

*// DO NOT forget the star! **

____ *q;*

```
// Fill blank as ArrayStack() or LinkedListStack()
// depending on whatever you chose previously
// It's a Constructor: DO NOT forget the parentheses! ()
q = new ____;


q->enqueue("HI ");
q->enqueue("THERE! ");
q->enqueue("How's ");
q->enqueue("it ");
q->enqueue("going? ");
q->enqueue("Did ");
q->enqueue("you ");
q->enqueue("know ");
q->enqueue("you're ");
q->enqueue("my ");
q->enqueue("Best Friend? ");

string a[12];

for(int i = 0; i < 12; i++) {
a[i] = q->dequeue();
std::cout << a[i] ;
}
}
```

What happened?

Notice how, after running the code, the message clearly displays in perfect order.

If you used an Array Queue, we tried to enqueue 10 strings - but the 11th string was not enqueued. What happened? A queue overflow. Next, the iterating code tried to do dequeue 12 times. But once there was nothing to dequeue, a Queue Underflow occurred.

If you used a Linked List Queue, we manage to enqueue all 11 strings. How? Linked Lists have no list sizes, remember? But then again, once there was nothing to dequeue, a Queue Underflow also occurred.

Part 2: Switching List Types

If you switch to the other list type (array or linked list), you should have the same results. Try it out, and note the differences between the two data structures.

REFERENCE SECTION:

REF-01: The Stack

Array Stack, Data Structure in Pseudocode

// Abstract Data Type: Stack, as an Array

// The Stack Has:

// - the Actual Array (with Size and Data Type)

// - the Max Size of the Stack (Integer)

// - the Current Item Count (Integer)

CompositeStructure ArrayStack {

Integer MAXSIZE

Integer COUNT

(choose a data type) ARRAY[MAXSIZE]

}

Array Stack: Push() Function in Pseudocode

// INPUT: - an item (some Data Type)

// OUTPUT: - nothing

// EFFECT: inserts an item (some Data Type) onto our Array Stack

void push(<chosen data type> ITEM) {

// 1) Check if there's room in the array for the extra item

if (this.MAXSIZE < this.COUNT) {

// 2) If there's no room in the array,

// …either report a stack overflow error (if fixed-size array)

// (assume error() reports an error for you)

error("Stack Overflow")

// …or increase the array size (if variable array)

// (assume increaseArray() increases array size by 1)

this.increaseArray()

}

// 3) If there's room in the array, add the item to the stack.
// Set the new item's index as the current Item Count
this.ARRAY[COUNT] = ITEM

// 4) And Finally, Increment the Item Count
this.COUNT += 1
}

Array Stack: Pop() Function in Pseudocode

// INPUT: - nothing
// OUTPUT:- an item (some Data Type)
// EFFECT: removes and retrieves an item (some Data Type) from our Array Stack
<chosen data type> pop() {

// 1) Check the Array Count if there are any items to pop.
if (this.COUNT <= 0) {

// 2) If not, either return NULL
return NULL
// or report a stack underflow error
// (assume error() reports an error for you)
error("Stack Underflow")

// 3) If there's an item to pop, Decrement the Item Count
this.COUNT -= 1

// 4) Create a Local Variable.
// Set it to be your return item
// (it should be the item at the end of the stack!)
<chosen data type> r = this.ARRAY[COUNT]

// Remove the item at the End of the Stack afterwards

this.ARRAY[COUNT] = null

// 5) And Finally, the function returns your item and ends.

return r

}

Note how we decremented the Item Count BEFORE we outputted our item and ended the function. If you recall, function returns should be the very last thing in a function procedure - as it ends the function.

For experienced programmers, you can also opt to forego the local variable if you wish, depending on what language you wish to use and how much performance you want.

Linked List Stack, Data Structure in Pseudocode

// A Singly Linked List Node has:

// - the Next Node (Point to the Next Node)

// - the Data Item (choose what data type you want)

CompositeStructure Node {

Node NEXTID

(choose a data type) ITEM

}

// Abstract Data Type: Stack, as a Linked List

// The Stack Has:

// - a Head (Linked List Node)

CompositeStructure LinkedListStack {

Node HEAD

Integer COUNT

}

Now, let’s look at how the Push() and Pop() methods work, in detail:

Linked List Stack: Push() Function in Pseudocode

// INPUT: - an item (some Data Type)

// OUTPUT: - nothing

// EFFECT: inserts an item (some Data Type) onto our Linked List Stack

void push(<chosen data type> INPUT) {

// 1) Make a new Local Variable. Set it as a Node.

// This will be the New Linked List head.

Node NEWHEAD = new Node

// 2) Set your input item to be the New Head Node's data

NEWHEAD.ITEM = INPUT

// 3) Set the previous head's next node to the New Head

NEWHEAD.NEXT = this.HEAD

// 4) Set the New Head as the Linked List's Stack Head

this.HEAD = NEWHEAD

// 5) And Finally, Increment the Item Count

this.COUNT += 1

}

Linked List Stack: Pop() Function

// INPUT: - nothing

// OUTPUT:- an item (some Data Type)

// EFFECT: removes and retrieves an item (some Data Type) from our Array Stack

<chosen data type> pop() {

// 1) Check if the Head Node isn't Null (if the Head Node Exists)

if (this.HEAD = NULL) {

// 2) If so, either return NULL

return NULL

}

// or report a stack underflow error

// (assume error() reports an error for you)

error(“Stack Underflow”)

}

// 3) If there’s an item to pop, create a Local Variable.

// Set it to be your soon-to-be return item

// (it should be the Head Node!)

<chosen data type> r = HEAD.ITEM

// 4) Get the Node NEXT to the Head.

// This will NOW be the new LinkedList Head.

this.HEAD = this.HEAD.NEXT

// 5) Decrement the Item Count

this.COUNT -= 1

// 7) And Finally, the function returns your item and ends.

return r

}

REF-02: The Queue

Array Queue: Data Structure in Pseudocode

// Abstract Data Type: Queue, as an Array

// The Queue Has:

// - the Actual Array (with Size and Data Type)

// - the Index of the Front Item (Integer)

// - the Index of the Back Item(Integer)

// - the Max Size of the Queue (Integer)

// - the Current Item Count (Integer)

CompositeStructure ArrayQueue {

Integer FRONT

Integer BACK

Integer MAXSIZE

Integer COUNT

(choose a data type) ARRAY[MAXSIZE]

}

Array Queue: Enqueue() Function in Pseudocode

// INPUT: - an item (some Data Type)

// OUTPUT: - nothing

// EFFECT: inserts an item (some Data Type) onto our Array Queue

void enqueue(<chosen data type> ITEM) {

// 1) Check if there's room in the array for the extra item

if (this.MAXSIZE < this.COUNT) {

// 2) If there's no room in the array,

// …either report a Queue overflow error (if fixed-size array)

// (assume error() reports an error for you)

error("Queue Overflow")

// …or increase the array size (if variable array)

// (assume increaseArray() increases array size by 1)

this.increaseArray()

}

// 3) If there's room in the array, we add the item to the queue.

// Set the new item's index as the current Back Index

// (so the item is the very last item in the Queue)

this.ARRAY[BACK] = ITEM

// 4) Increment the Back Index.

// If the Back Index becomes bigger than or equal to the Array Size,

// set the Back Index to 0 (it's now at the start of the Circular Array).

this.BACK +=1

if (this.BACK >= MAXSIZE) this.BACK = 0

// 5) And Finally, Increment the Item Count

this.COUNT += 1

}

Array Queue: Dequeue() Function in Pseudocode

// INPUT: - nothing

// OUTPUT:- an item (some Data Type)

// EFFECT: removes and retrieves an item (some Data Type) from our Array Queue

<chosen data type> dequeue() {

// 1) Check the Array Count if there are any items to dequeue.

if (this.COUNT <= 0) {

// 2) If not, either return NULL

return NULL

// or report a queue underflow error

// (assume error() reports an error for you)

error(“Queue Underflow”)

// 3) If there’s an item to dequeue, create a Local Variable.

// Set it to be your return item

// (it should be the item at the front of the queue!)

<chosen data type> r = this.ARRAY[FRONT]

// Remove the item at the Front of the Queue afterwards

this.ARRAY[FRONT] = null

// 4) Increment the Front Index.

// If the Front Index becomes bigger than or equal to the Array Size,

// set the Front Index to 0 (it’s now at the start of the Circular Array).

this.FRONT+=1

if (this.FRONT >= MAXSIZE) this.FRONT = 0

// 5) Decrement the Item Count

this.COUNT -= 1

// 6) And Finally, the function returns your item and ends.

return r

}

Linked List Queue: Data Structure in Pseudocode

// A Singly Linked List Node has:

// - the Next Node (Point to the Next Node)

// - the Data Item (choose what data type you want)

CompositeStructure Node {

String ID

String NEXTID

(choose a data type) ITEM

}

// Abstract Data Type: Queue, as a Linked List

// The Queue Has:

// - the Front (Linked List Node)

// - the Back (Linked List Node)

CompositeStructure LinkedListQueue {

Node FRONT

Node BACK

Integer COUNT

}

Now, let's look at how the Enqueue() and Dequeue() methods work, in detail:

Linked List Queue: Enqueue() Function in Pseudocode

// INPUT: - an item (some Data Type)

// OUTPUT: - nothing

// EFFECT: inserts an item (some Data Type) onto our Linked List Queue

void enqueue(<chosen data type> INPUT) {

// 1) Make a new Local Variable. Set it as a Node.

// This will be the New Linked List back.

Node NEWBACK = new Node

// 2) Set your input item to be the New Back Node's data

NEWBACK.ITEM = INPUT

// 3) Set the previous back's next node to the New Back

NEWBACK.NEXT = this.back

// 4) Set the New Back as the Linked List's Back of Queue

this.BACK = NEWBACK

// If this queue was previously empty,

// the New Node will also be the new Front

if (this.COUNT <= 0) {

this.FRONT = NEWBACK

}

// 5) And Finally, Increment the Item Count

this.COUNT += 1

}

Linked List Queue: Dequeue() Function in Pseudocode

// INPUT: - nothing

// OUTPUT:- an item (some Data Type)

// EFFECT: removes and retrieves an item (some Data Type) from our Linked List Queue

<chosen data type> dequeue() {

// 1) Check if the Front Node isn't Null (if the Front Node Exists)

if (this.FRONT = NULL) {

// 2) If so, either return NULL

return NULL

}

// or report a queue underflow error

// (assume error() reports an error for you)

error("QueueUnderflow")

}

// 3) If there's an item to dequeue, create a Local Variable.

```
// Set it to be your soon-to-be return item
// (it should be the Front Node!)
<chosen data type> r = FRONT.ITEM

// 4) Get the Node NEXT to the Front.
// This will NOW be the new LinkedList Front.
this.FRONT = this.FRONT.NEXT

// 5) Decrement the Item Count
this.COUNT -= 1

// 7) And Finally, the function returns your item and ends.
return r
}
```

BONUS CHAPTERS:

BONUS CHAPTER 01a: Lists

At one point, computers will eventually have a collection of data to deal with - regardless whether they are objects or basic data.

Thus, we have lists.

Lists in General

As an Abstract Data Type, lists are essentially a series of data items connected together.

Computers will generally go through data item on the lists one by one.

There are two general types of lists: linked lists and arrays. Most programming languages will usually include either type of list.

Linked Lists v.s. Arrays. Which list to use?

IMPORTANT NOTE: The ironic thing about most programming languages is the fact that they ALL have arrays and linked lists available. But, what if a colleague or boss asks you which one to use? See, sometimes making the slightest decisions like these will be such big deals that make or break an app - and sometimes, even job applications, careers and startups.

In general, use Linked Lists for quickly adding and deleting data items for that list, no matter what order they are. Whether the item is first, last, or in the middle of the list, it takes the same time and effort for the computer to add/remove data items. Also, if you also don’t know how big your list will be, linked lists are preferable.

In general, use Arrays for any computing where you need to access or process each element in the list. The indices in the Array will help your computer process the list faster, as well as give your computer multi-process potential for that list. Also, use arrays if each item in the list needs an index. For example, if you wanted to randomly get list items, you can only do it in an array (in a linked list, how would the computer know what item to get? Think about it…)

Bottom line:

Linked Lists = any size, faster add/remove

Arrays = fixed size, faster access/process

Linked Lists v.s. Arrays: Data Management

In general, if given the same data type to store in lists, arrays take less memory/storage space than linked lists. You’ll find out about this later, but linked lists generally need slightly more data in its structure than arrays. However, in lists with very high number of items, this slight difference is actually significant.

View Programming: Master Handbook Series Here

BONUS CHAPTER 01b: Linked Lists

The concept of linked lists is actually very simple.

Think of linked lists as a chain - literally. In a linked list, each “link” in the chain will have its item, followed by whatever “link” is next. If a certain link has no links next, then that link is at the end of the chain - or rather, the end of the linked list.

Here’s an example of what a doubly linked list would look like visually:

[ID01][item][Prev: NULL][Next: ID03]

[ID04][item][Prev: ID03][Next: NULL]

[ID03][item][Prev: ID01][Next: ID04]

There are three “links” in the chain, with link ID04 being the last one. Why is ID04 last? Easy - because there is no link after it.

And note how the IDs go from 01, then 04, and 03, and they’re not in order. However, in linked lists, the numbers DO NOT tell you the order. I REPEAT. The NUMBERS DO NOT tell you the order

It’s because of one crucial fact about how the ordering in linked lists are set up.

“The Ordering of a Linked List is this: The very first node (no nodes before it), then its next one, then the next one, and so on - until there’s a node with nothing after it.”

Building Linked Lists From Scratch:

If you ever had to design a linked list from scratch, the concept is simple. A single link in a linked list consists of the actual data, then the ID of whatever the next or previous nodes are. Singly linked list nodes only point to the next node, while Doubly Linked List nodes point to both next and previous nodes.

In our pseudocode below, there are four key fields in a doubly linked list node: the ID, the previous node’s ID, the next node’s ID, and the actual data item.

// Abstract Data Type: Linked List

// Each element in the Linked List is represented by a Node.

// A Node has:

// - an ID (String)

// - the ID of the Previous Node (String)

// - the ID of the Next Node (String)

// - the Data Item (choose what data type you want)

CompositeStructure Node {

String ID

String PREVID

String NEXTID

(choose a data type) ITEM

}

Usually, you can only use one data type per each linked list. The reason is simply how the computer system itself processes and organizes data (you'll learn more about this if you know how Computer Systems themselves work).

View Programming: Master Handbook Series Here

BONUS CHAPTER 01c: Arrays

If you look up the word ‘array’ in the dictionary, you’ll find out that it’s an elaborate, and sometimes beautiful, arrangement of items in a particular order.

Emphasize the phrase “arrangement of items in a particular order”.

And in programming, you now know the key point of Arrays - they’re a list of data items sorted in a particular order.

Here’s an example of what an array with a size of 5 would look like visually:

[0][item A]

[1][item B]

[2][NULL]

[3][item D]

[4][NULL]

Here, you can see how quickly and easily you can add or get data items from an array. If you know the index where you stored your data, just access the array, the index, then the data is yours.

Also, arrays sizes are usually fixed. In some programming languages, there are also Dynamic or Variable-Size Arrays. But depending on how the programming language works, the data items are usually re-added into arrays that have as much room as the number of items.

Building Arrays From Scratch:

If you ever had to design a data array from scratch, the concept is simple too. But unlike linked lists, there aren’t usually any individual nodes. You simply build the array-type list itself. It requires two main things: the length or size of the array and the data type you want to store in it.

// Abstract Data Type: Array

// An Array has:

// - an Array Size

// - the Type of Data Item to Store (choose what data type you want)

CompositeStructure Array {

Integer SIZE

(choose a data type) ITEM

}

And just like linked lists, you can only use one data type per each array. Unlike linked lists, however, arrays are strict about having only one data type per array. The reason for this is how the computer assigns data memory/storage for that array. Having one data type per each array makes the array much more structured and easy to access.

View Programming: Master Handbook Series Here