Data Structures

Data management systems are concerned with various aspects of data handling, such as:

Measurement	Storage
Collection	Aggregation
Transcription	Update
Validation	Retrieval
Organization	Protection

Some useful goals for a data management system:

- Data must be represented and stored so that they can accessed later
- Data must be organized so that they can be selectively and efficiently accessed
- Data must be processed and presented so that they support the user environment effectively
- Data must be protected and management so that they retain their value

Data structures help to achieve the first two goals.

A Data Structure is a way of organizing data so that it may be used efficiently. It is a class of data that can be characterized by its organization and the operations that are defined on it. With this definition, a data structure is no different from an object class or even a data type.

Operations on Data Structures:

Create (Constructors)
Destroy (Destructor)
Access (get and set functions)
Predicates (Boolean functions)

Records

The record is a simple yet fundamental type of data structure. A record generally contains an identifying field (key), e.g., EMPLOYEE, a data structure of type record:

													Telephone Numbers			
			Name			Office							Phone		Phone	
Job Title	Empl ID	Rate	Last	Firs	MI	Bldg	Rm	Project Codes					Area	Local	Area	Local
ANALYST	123456789	15.93	Mudd	Joe	H	CSC	403	18	40	41	50	53	202	1234567	301	1234567

What is a … ?
Array within a record:
Record within record:
Record within record within array:

The value of the ***key*** uniquely identifies a record from within a collection of records.

A ***record*** is a finite, ordered collection of possibly heterogeneous elements, which may themselves be composite structures. The primary purpose of the record data type is to group together logically related fields. A collection of logically related record occurrences that are treated as a unit is called a ***file***.

Lists

A list is a one-dimensional structure consisting of a collection of elements. If the elements are in contiguous locations, then the list is a linear list. [All vectors are linear lists.]

Arrays

An ***array*** is a finite set of ordered, homogeneous elements.

Finite – a specific number of elements in the array
Ordered – elements are arranged so there is a first, second, etc.
Homogeneous – all elements must be of the same data type

Arrays are used when it is necessary to keep a large number of items in memory at the same time and to perform some operation(s) on them in a uniform manner.

Basic array operations:
- Declaration of the array
- Extraction of a given element value from the array (uses subscript)
- Storing a value in a given position in the array
- Searching for a target element
- Sorting the array

An array (or, linear list) is an example of a static structure. ***Static structures*** allow for relatively sophisticated and often complex data organization. Their simplicity and easy access are offset by their "lackluster" existence, i.e., they lack any potential for change during their existence.

Static structures have the following characteristics:
- None of the descriptive information in the definition of the structure changes in any way during the lifetime of the structure.
- The data elements of an allocated structure are physically contiguous; a single block of storage is allocated to accommodate all the data of that structure.
- Interrelationships among data elements do not change during the lifetime of a structure.

Variable-length Arrays (linear lists)

The objective of a variable-length list is to allow natural representation and manipulation of data used in certain applications, i.e., the data set must be allowed to grow and shrink in size ("dynamically") during run time. By definition, then, an array cannot be a variable-length array; it can, however, be a ***house*** for a variable-length data structure.

Accessing variable-length arrays by subscript is far less common and less important than with fixed-size arrays since each array element may change its position as the array changes in size. Instead, access is relative, e.g., "get the next element."

Stack – a sequence for which elements may be inserted and deleted from one end only. We insert to and delete from the "Top."

Queue – a sequence for which elements may be inserted only at the "Rear" and deleted only from the "Front."

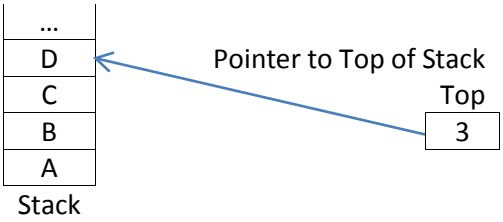
Deque (double-ended queue) – a sequence for which insertions and deletions may only take place at the ends.

Stacks

A ***stack*** is an ordered set of data items into shich new items may be insrerted and from which items may be deleted at one end only, called the **Top** of the stack. A stack is the most commonly used linear data structure.

Rules:
- New items must be placed on top of the stack
- Only items at the top of the stack may be removed
- Stacks expand and shrink with the passage of time
- The last element inserted must be the first element deleted (LIFO)
- You only have access to knowledge about the items at the Top. You know nothing about any of the items below it unless you keep an auxiliary record.

When do we use a stack? When ordering doesn't matter (e.g., memory locations).



Operations on a Stack:
- Push – adds an item onto the Top of the Stack, e.g., Push(s,x) adds item x onto stack s.
- Pop – removes the top element off the stack, e.g., x=Pop(s) removes the top element off stack s and moves its value to x.
- Test for Empty stack – since it is illegal to pop an item off of an empty stack (this is called 'stack underflow") we must first check for that condition, e.g., Empty (s) determines whether stack s is empty and returns the value true or false. Alt - IsEmpty(s).
- Peek – determines what the top tiem of the stack is without removing it. (Equivalent to a Pop and Push together.) e.g., Peed(s) returns the value of the top element of s without removing it from the stack.
- Top (s) is used to refer to or change the top value of the s.
- Test for full stack (overflow) – this is only important with static allocation, when you are implementing the stack inside an array.

Although an array cannot be a stack it can be a home for a stack. The stack will shrink and grow within the confines of the array.

Stacks are used when we want to process data in reverse order in which they were encountered. Compilers make a lot of use of this type of data structure . e.g.,

(1) matching parenthesis in arithmetic expressions. We can treat an arithmetic expression as a string, e.g.,
"((A-B)/C-D*(E+F))"

When stack is empty parens are matched.

[Exercise: Given any string, write an algorithm which will check whether the parentheses match. Input string. Output Yes or No.]

(2) testing for proper nesting sequence

The stack below could be used to test for proper nesting in the following piece of code:

```
while (data != end){
if (score > 90) grade= 'A';
else if (score >80) grade = 'B';
else if ...
```

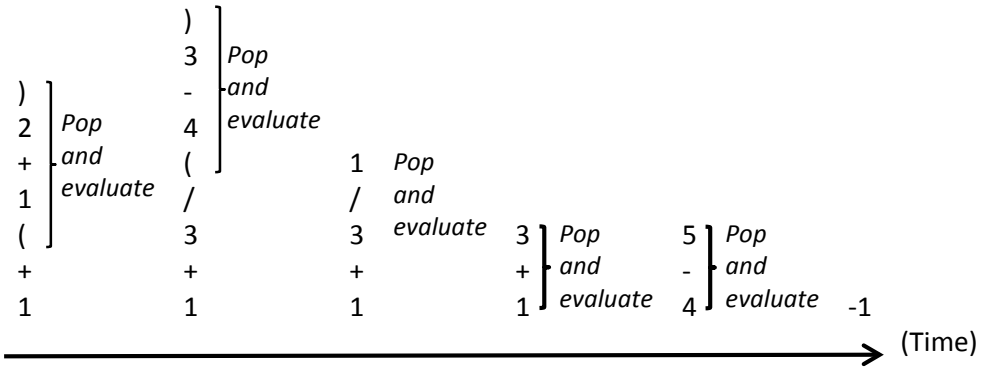
…
if
else
if
{
while

(3) expression evaluation

For this expression (maybe in a string): A + (A+B)/(D-C) – E

With these values:

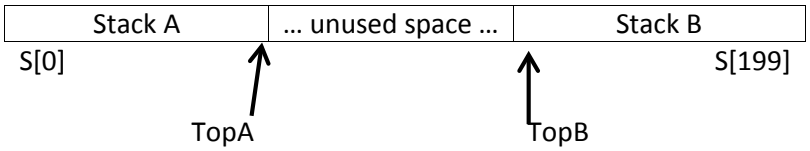
A	B	C	D	E
1	2	3	4	5



(4) Implementing recursive calls

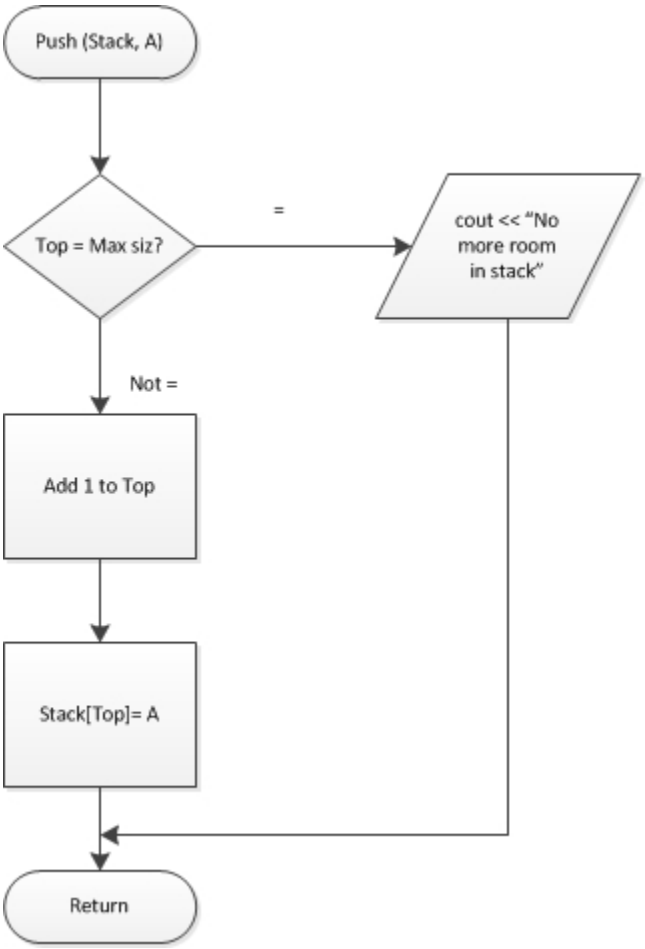
Special case:

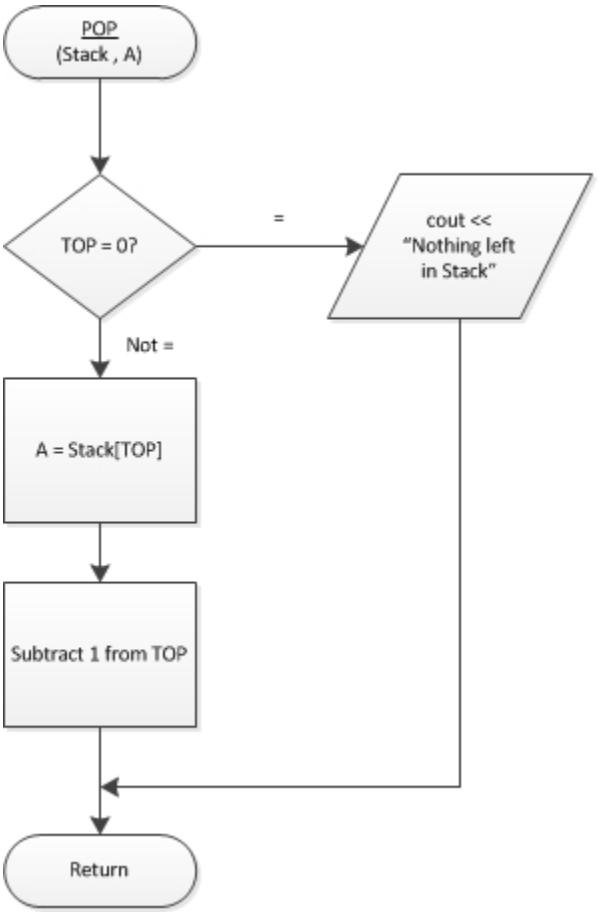
Suppose we want to implement two stacks, A and B. We may use two arrays of (say) size 100 each or a single array of size 200 (oer, perhaps, less). Then the two stacks can grow and shrink dynamically within the confines of the array. Stack A grows 'up' from s[0] while Stack B grows 'down' from s[99].

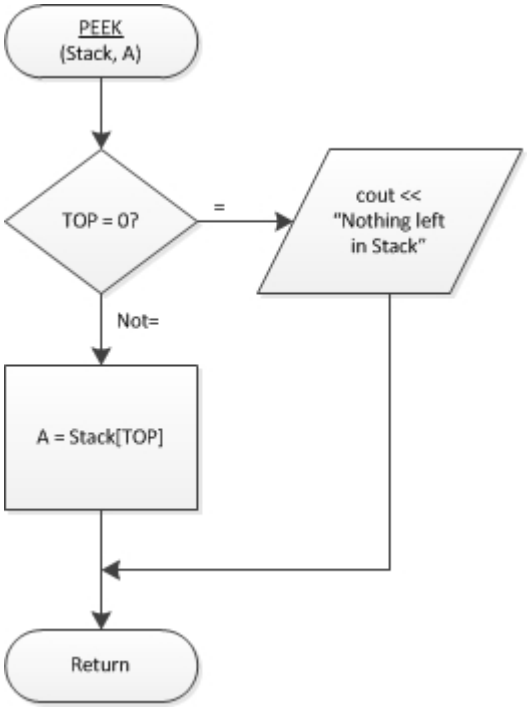


The advantage of this technique is that one stack may have more than 100 elements as long as the other stack has correspondingly few elements. Then, overflow occurs only when the sum of the number of elements in both stacks reaches 200.

Algorithms for Stack operations:




POP
(Stack , A)
TOP = 0?
=
cout <<
"Nothing left
in Stack"
Not =
A = Stack[TOP]
Subtract 1 from TOP
Return


PEEK
(Stack, A)
TOP = 0?
=
cout <<
"Nothing left
in Stack"
Not=
A = Stack[TOP]
Return

Question: How to search a stack? Suppose we wish to determine whether a particular element is in the stack.

Solution #1: We could Pop each element in turn, put it on another stack, and then restack.

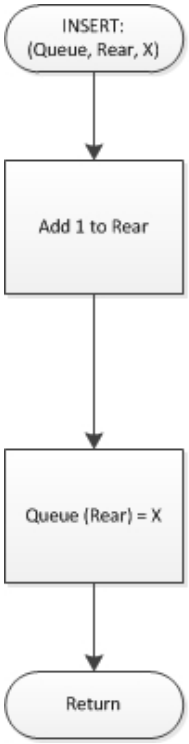
Solution #2: Since we have implemented the stack as an array, it is simpler and more efficient to (temporarily) stop looking at it as a stack and look at it as an array just for the search.

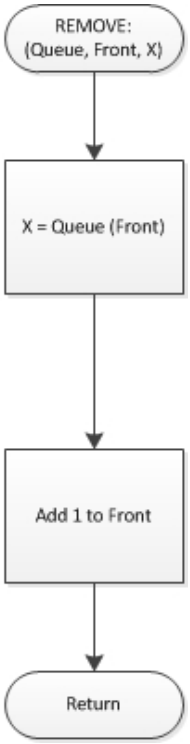
Queues

A queue is an ordered set of items from which items may be deleted only at one end (Front) and into which items may be inserted only at the other end (Rear). This is a FIFO system. Examples: bank, barber shop, supermarket, bus stop.

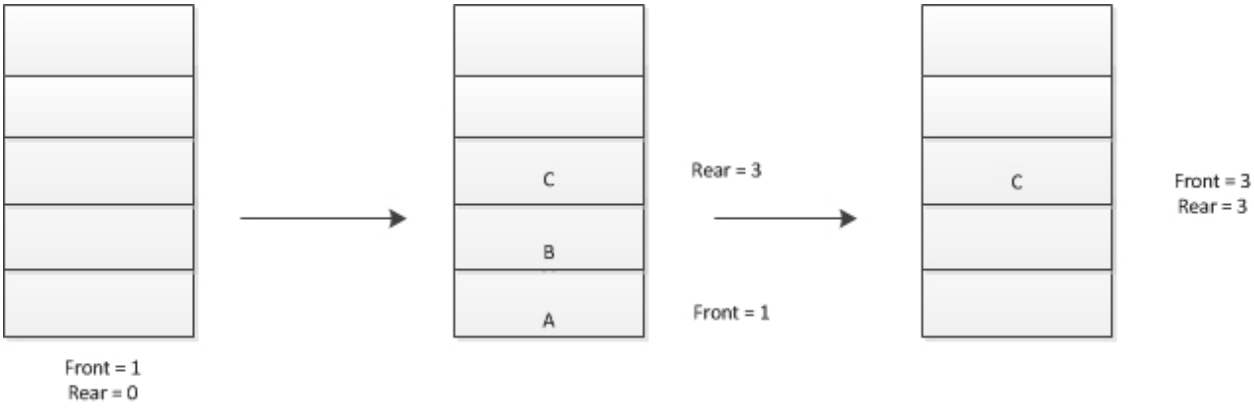
Operations on a queue:

- Create – declare and initialize
- Insert – inserts an item at the Rear of the queue. Insert (Queue, X)
- Remove – deletes an item from the Front of the queue. X=Remove(Queue)
- Empty – Boolean function to determine whether the queue contains any values. IsEmpty(Queue)
- Full – in array implementation, is the Queue full? We can't insert into a full Queue.

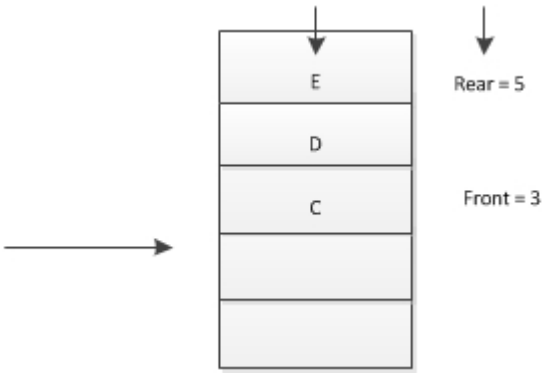




Initially (in the creation operation), REAR is equal to 0 and FRONT is equal to 1. Then, the queue is empty whenever REAR < FRONT.



Here's a problem – we reach a wall at the end of the "queue" (really, the end of the array housing the queue), even though the beginning of the array is empty.



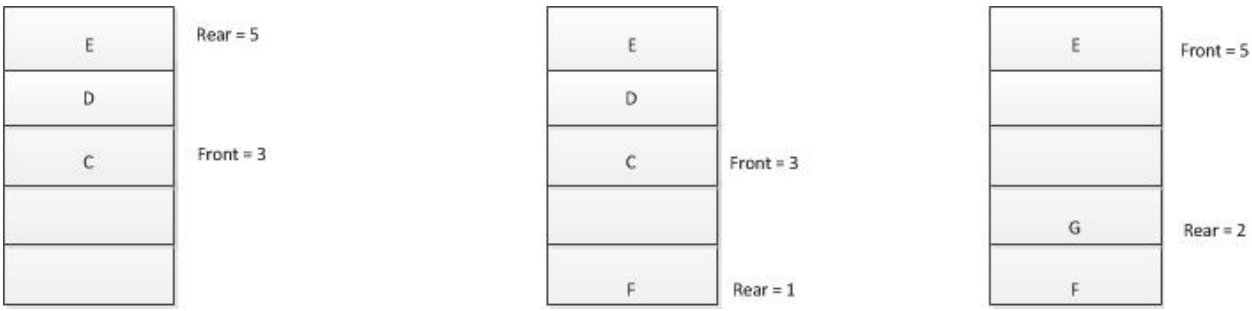
So, we cannot insert F even though the queue has room.

Solution #1:
Modify the REMOVE operation so that when an item is deleted, the whole queue is shifted to the beginning of the array. With this solution there is no need for FRONT since the first element of the array is always the beginning of the queue. An empty queue is represented by REAR=0.

Disadvantages of this solution:

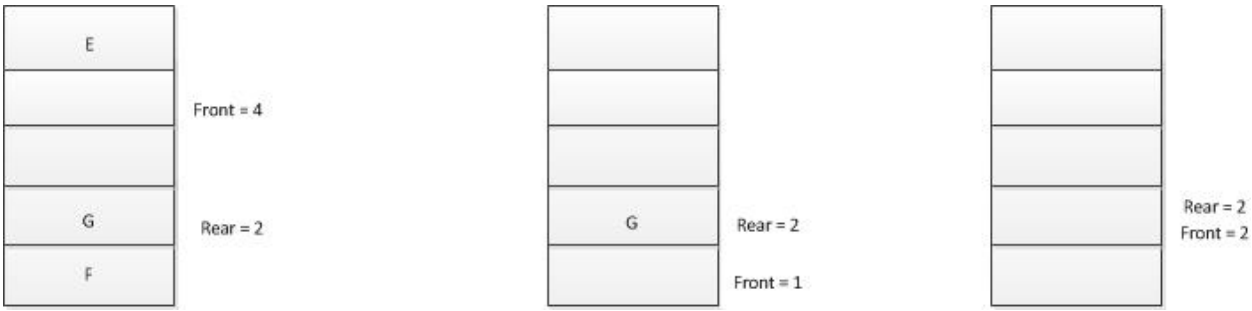
- inefficient – each deletion involves moving all the elements in the queue.
- compromises the abstract meaning of the queue

Solution #2:
This one is better (we will use it). View the queue as a (virtual) circle rather than as a straight line. Let the first element follow the last element.

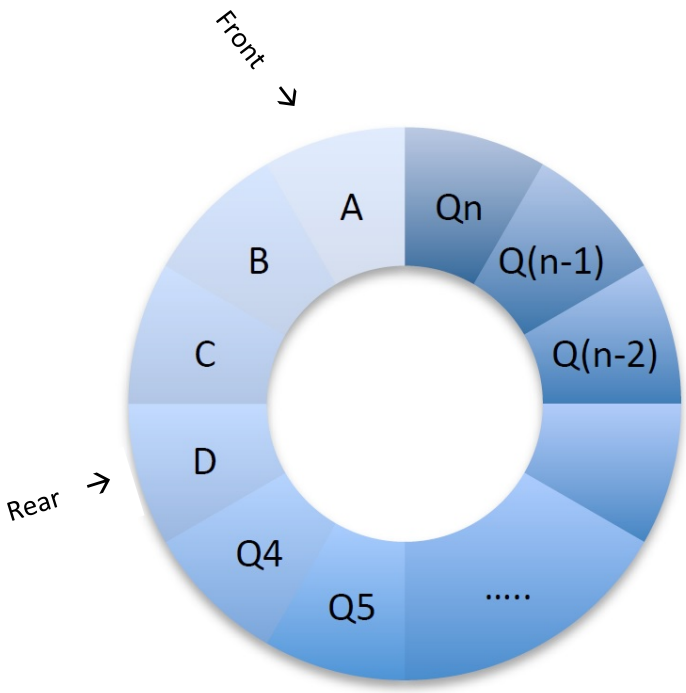


Note that REAR < FRONT no longer determines an empty queue.

Now, let FRONT point to the element *preceding* the first element of the queue. Then FRONT=REAR means the queue is empty.

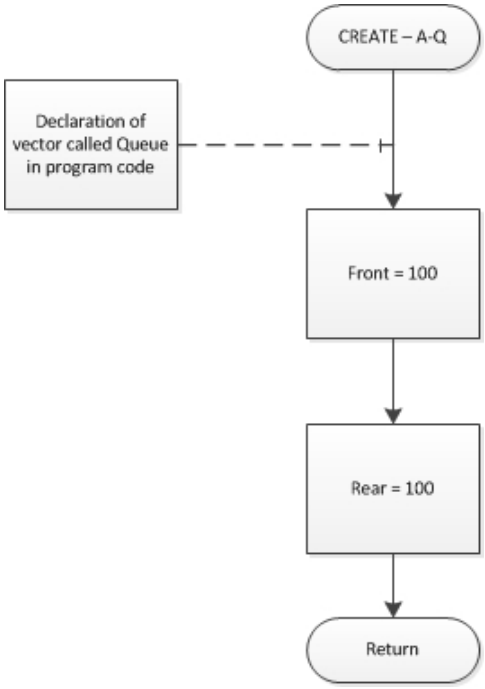


Or...

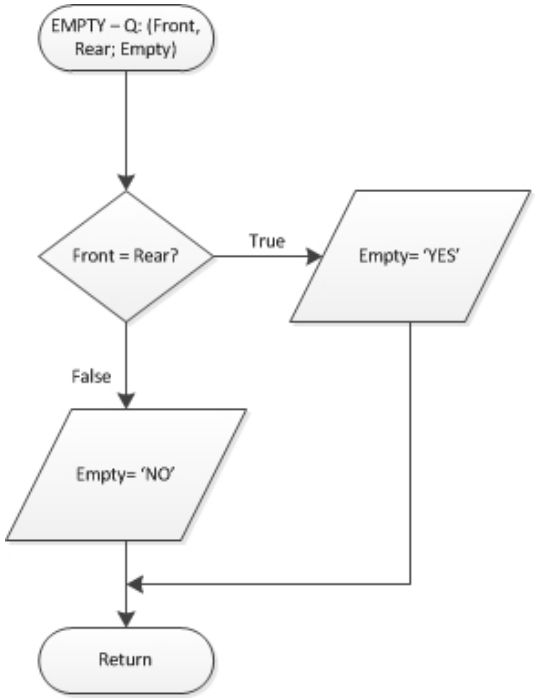

Front
Rear
A
B
C
D
Q4
Q5
.....
Qn
Q(n-1)
Q(n-2)

Implementing Queue Operations

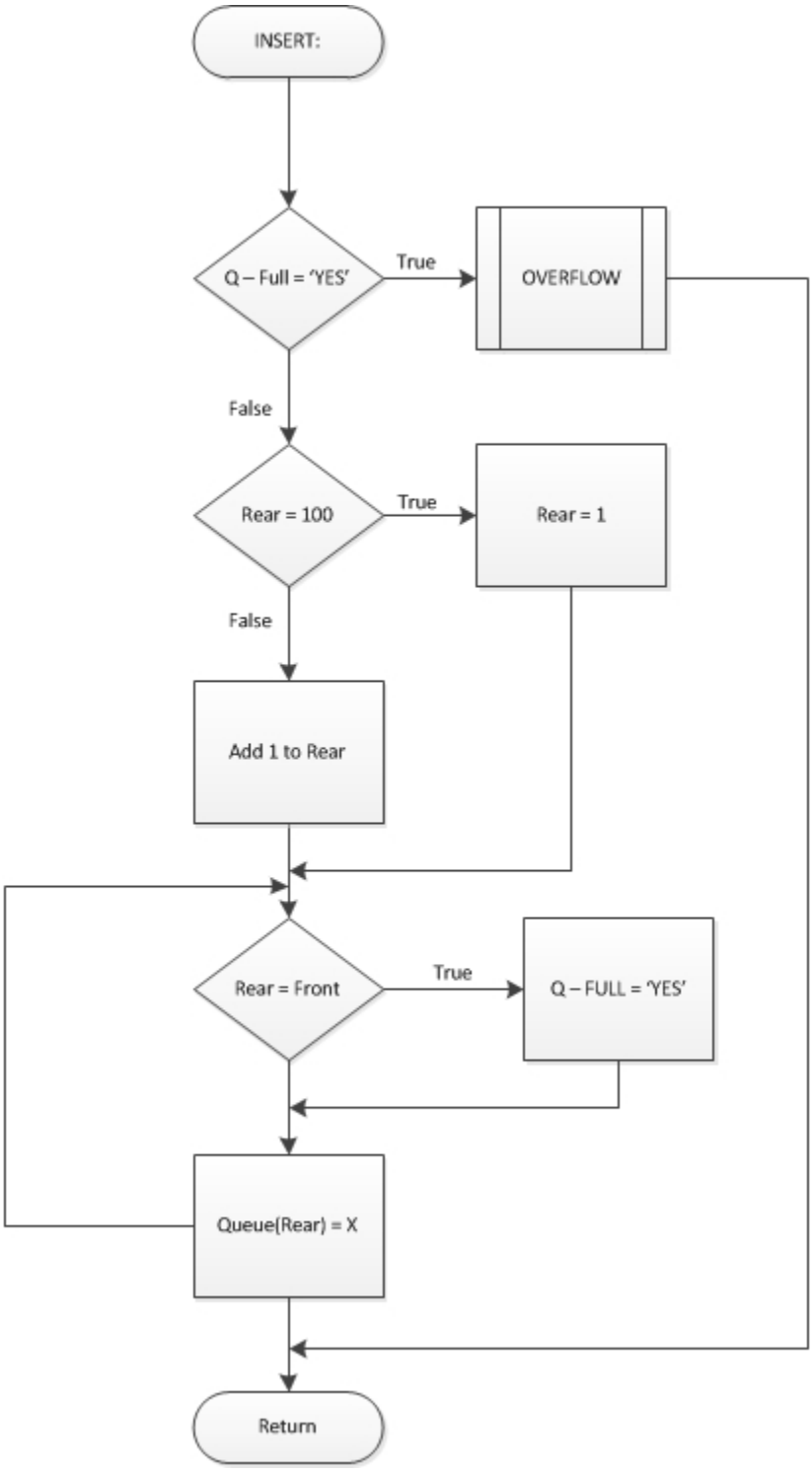
Create:



Test for Empty Queue:

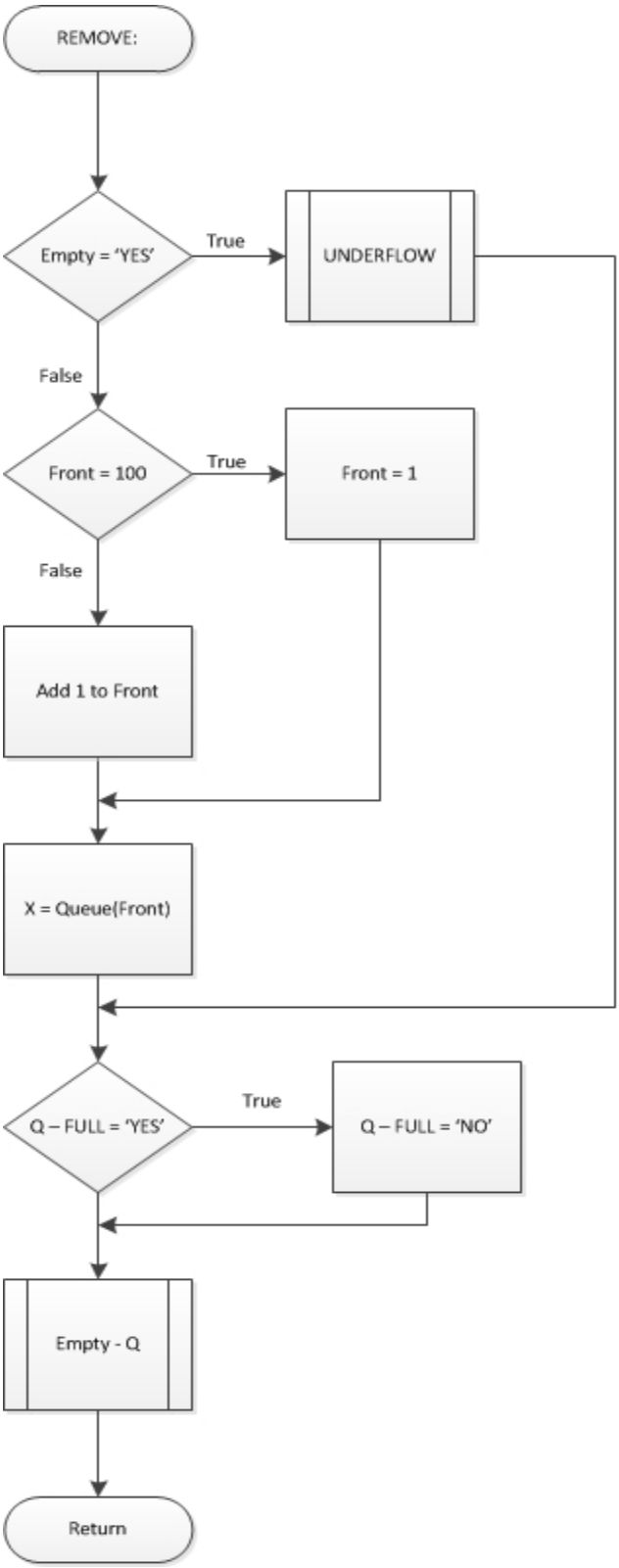


Insert:



When inserting a new element at the REAR of a Queue, we must first check for "overflow" condition (i.e., full array).

Remove:


REMOVE:
Empty = 'YES'
True
UNDERFLOW
False
Front = 100
True
Front = 1
False
Add 1 to Front
X = Queue(Front)
Q – FULL = 'YES'
True
Q – FULL = 'NO'
Empty - Q
Return