Lesson-3-Stack, Queue and List

https://yongdanielliang.github.io/animation/animation.html

Wholeness Statement

Knowledge of data structures allows us to pick the most appropriate data structure for any computer task, thereby maximizing efficiency.

Science of Consciousness: Pure knowledge has infinite organizing power and administers the whole universe with minimum effort.

Kinds of Abstractions

1. Procedural abstraction introduces new functions/operations
2. Data abstraction introduces new types of data objects (ADTs-Abstract Data Type)
3. Iteration abstraction allows traversal of the elements in a collection without revealing the details of how the elements are obtained
4. Type hierarchy allows us to create families of related types
   - All members have data and operations in common that were defined in (inherited from) the supertype

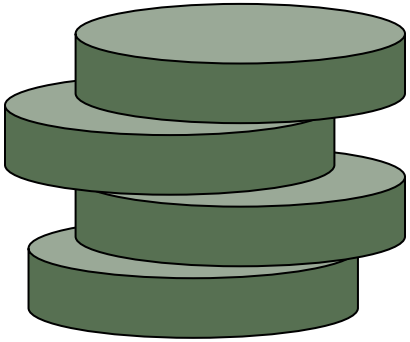
Algorithms and Data Structures

- Closely linked
  - Algorithm (operation)
    - a step-by-step procedure for performing and completing some task in a finite amount of time
  - Data structure
    - an efficient way of organizing data for storage and access by an algorithm
- An ADT provides services to other algorithms
  - E.g., operations (algorithms) are embedded in the data structure (ADT)

Abstract Data Types (ADTs)

- An ADT is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations
- Today we are going to look at several examples:
  - Stack
  - Queue
  - List

Stacks



Outline and Reading

- The Stack ADT
- Applications of Stacks

The Stack ADT operations

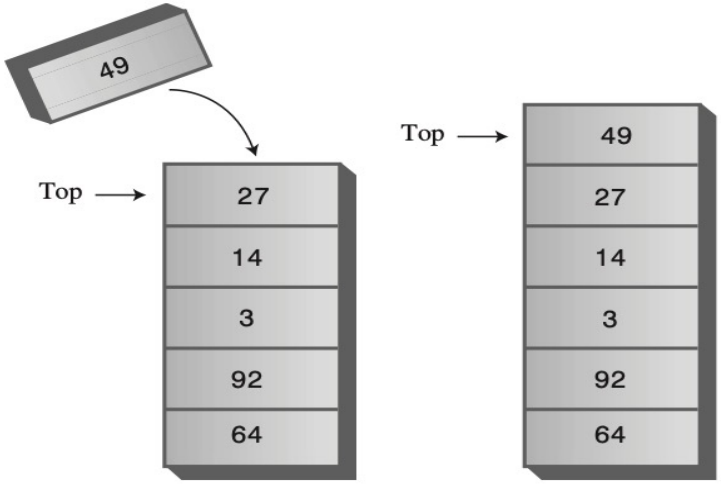
- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out (LIFO) scheme
  - Like a spring-loaded plate dispenser
- Main stack operations:
  - void push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored

Exceptions

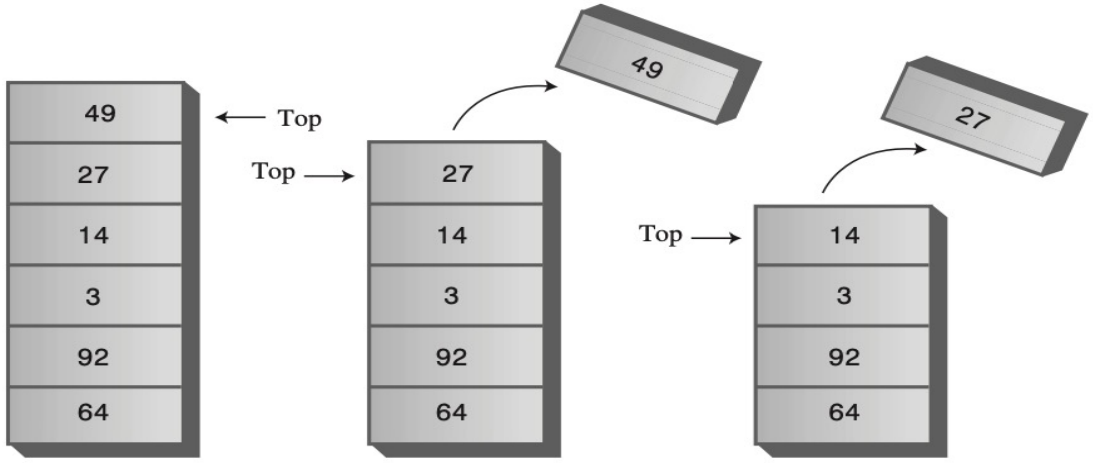
- Operations on the ADT may cause an error condition, called an exception
- Exceptions are said to be “thrown” when an operation cannot be executed
- Operations pop and top cannot be performed if the stack is empty
  - Attempting a pop or top on an empty stack causes an EmptyStackException to be thrown

Stack push and pop

https://yongdanielliang.github.io/animation/web/Stack.html



New item pushed on stack

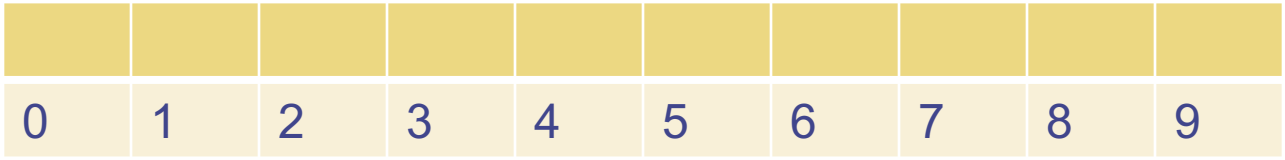


Two items popped from stack

Array-based Stack Implementation

- A simple way of implementing the Stack ADT uses an array. It's a straightforward implementation.
- Elements are added from left to right
- A variable top keeps track of the index of the top element

Stack S



top = -1

Stack Push operation

- The array storing the stack elements may become full
- A push operation will then throw a StackFullException, if stack is full and no room to store data.

push(10)

Algorithm *push(o)*
  if $t = S.length - 1$ **then**
    throw *StackFullException*
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$

Stack S

10									
0	1	2	3	4	5	6	7	8	9

top = 0

Stack Push operation

push(20)
push(30)
push(40)
push(50)

Algorithm *push*(*o*)
  if *t* = *S.length* – 1 **then**
    throw *StackFullException*
  else
    t ← *t* + 1
    S[*t*] ← *o*

Stack S

10	20	30	40	50					
0	1	2	3	4	5	6	7	8	9

top = 4

pop() – deletion operation

To pop an item off the stack, we return the value in location top and decrease top by 1. pop() – return 50 and top becomes 3.

```
Algorithm pop()
  if isEmpty() then
    throw EmptyStackException
  else
    del ← s[t]
    t ← t – 1
    return del
```

Stack S

10	20	30	40	50					
0	1	2	3	4	5	6	7	8	9

top = 4

Stack S after pop()

10	20	30	40						
0	1	2	3	4	5	6	7	8	9

top = 3

size(), isEmpty(), peek()

```
Algorithm size()
  return t + 1

Algorithm isEmpty()
      return top==-1

Algorithm peek()
  if isEmpty() then
    throw EmptyStackException
  else
    element ← s[t]
  return element
```

size(): returns the number of elements stored

isEmpty(): return the booelan value, whether elements stored or not

peek() operation returns the top item on the stack, without removing it from the stack.

Performance and Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined at creation and cannot be changed
  - Trying to push a new element onto a full stack causes an implementation-specific exception

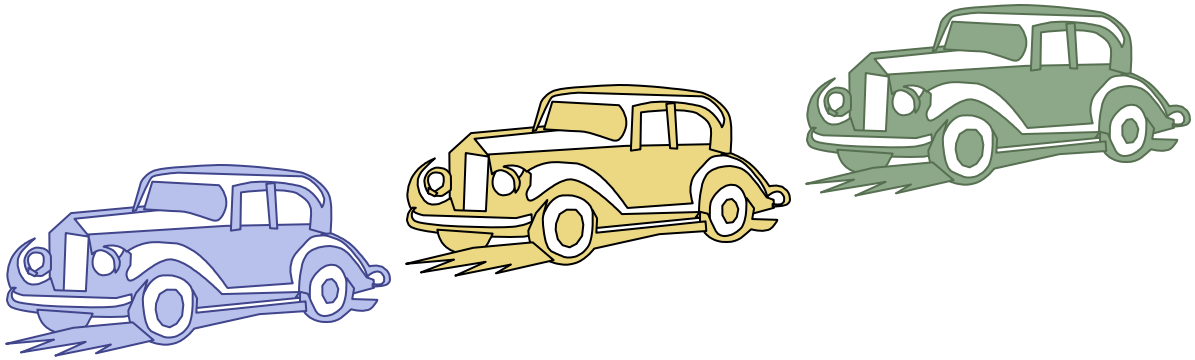
Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - In JavaScript, method calls, and function executions are indeed maintained using a call stack.
  - Evaluate an expression
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Main Point

1. Stacks are data structures that allow very specific and orderly insertion, access, and removal of their individual elements, i.e., only the top element can be inserted, accessed, or removed. *Science of Consciousness*: The infinite dynamism of the unified field is responsible for the orderly changes that occur continuously throughout creation.

Queues



Outline and Reading

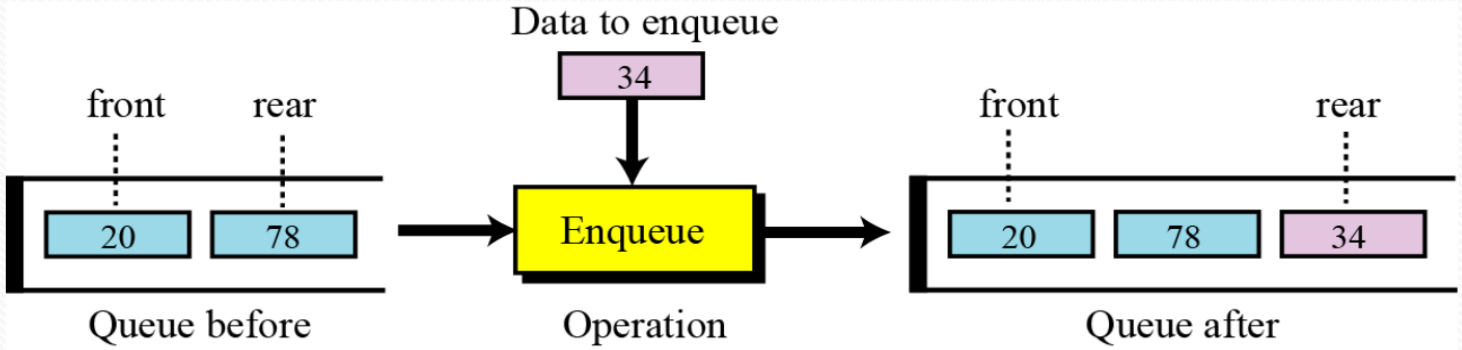
- The Queue ADT
- Implementation with a circular array

The Queue ADT

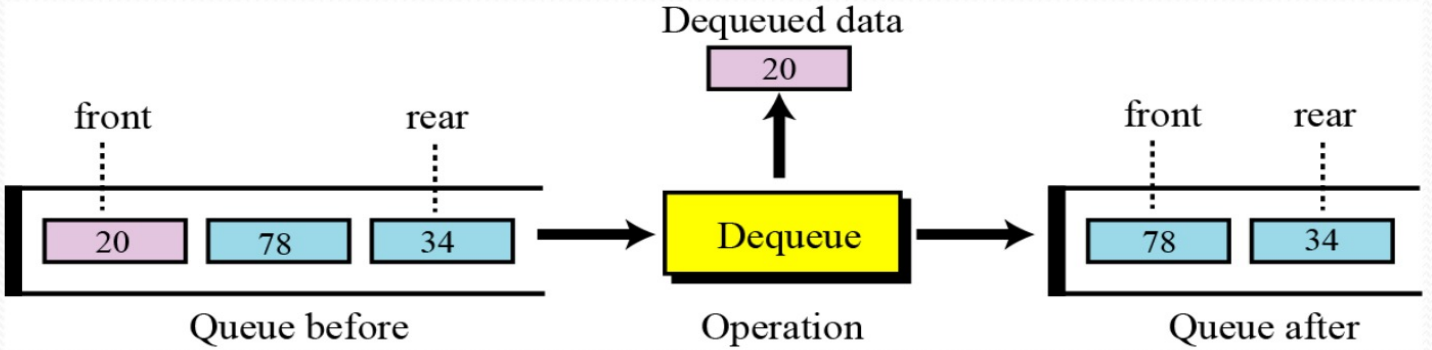
- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out (FIFO) scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - void enqueue(object): inserts an element at the end of the queue
  - object dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - object front(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored
- Exceptions
  - Attempting the execution of remove or front on an empty queue throws an EmptyQueueException

Operations on Queue

- **Enqueue/insert/add operation:**



Dequeue/delete/remove operation:



Applications of Queues

- Direct applications
  - Waiting lists, bureaucracy(First come first serve)
  - Access to shared resources (e.g., printer)
  - Multiprogramming (OS) – Task scheduling
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

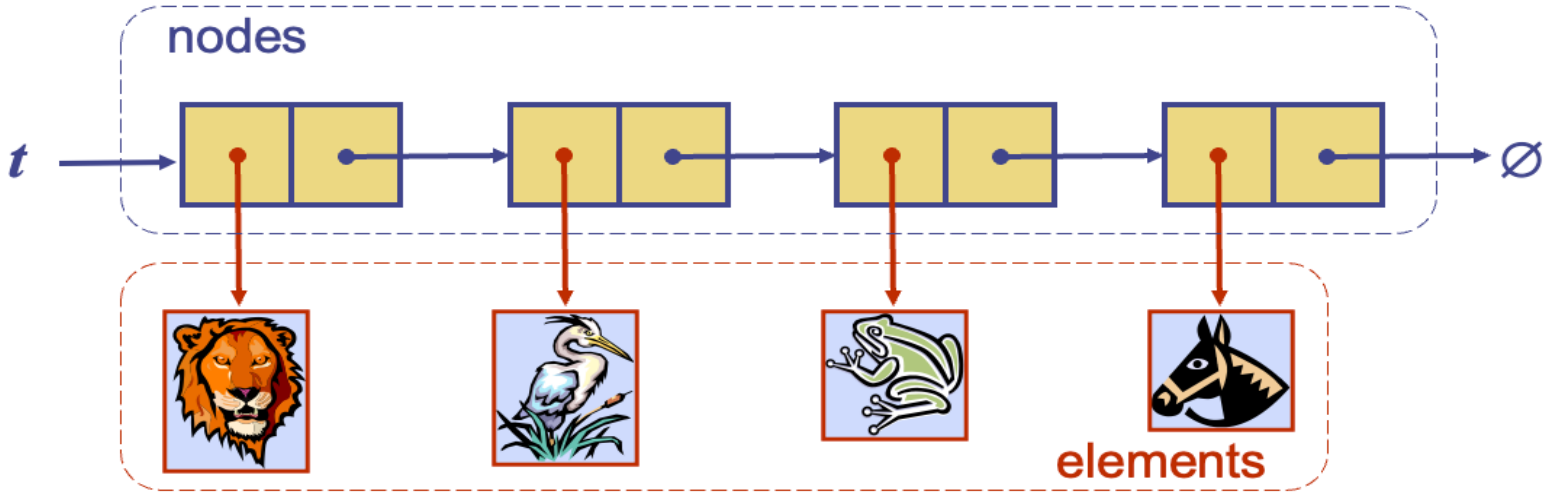
Main Point

2. The Queue ADT is a special ADT that supports orderly insertion, access, and removal. Queues achieve their efficiency and effectiveness by concentrating on a single point of insertion (end) and a single point of removal and access (front).
   Science of Consciousness: Similarly, nature is orderly, e.g., an apple seed when planted properly will yield only an apple tree.

Stack with a Singly Linked List

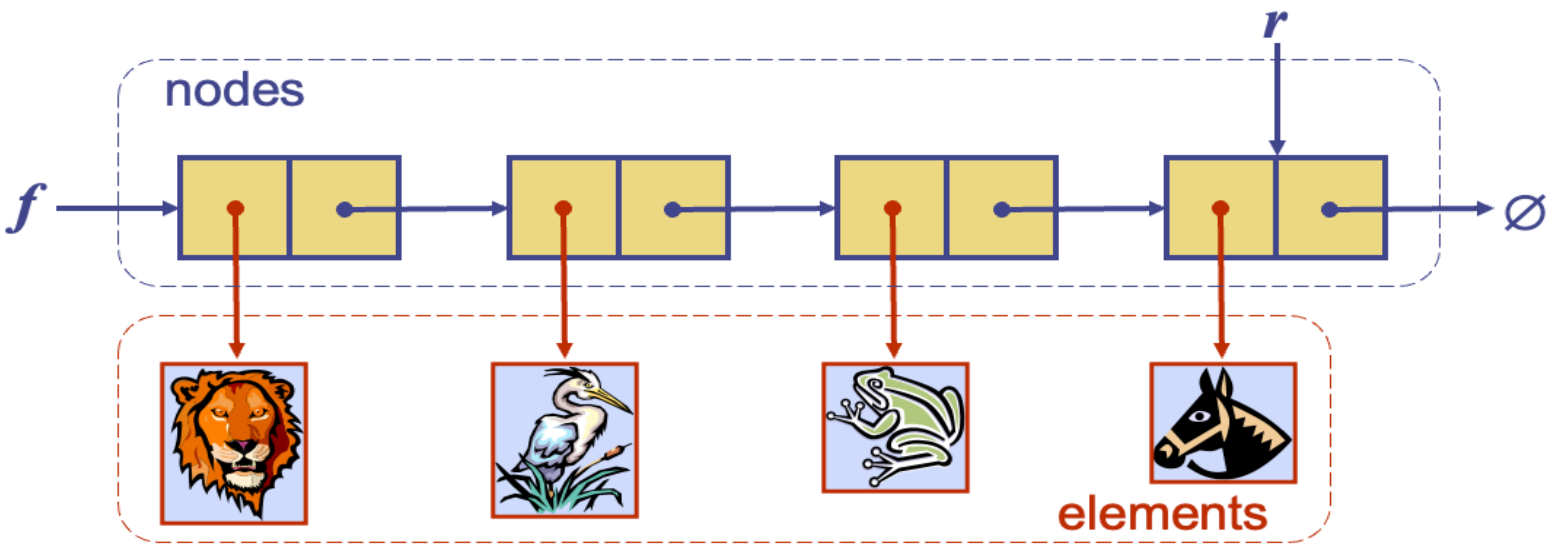
https://www.cs.usfca.edu/~galles/visualization/StackLL.html

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time. Refer the implementation in the class notes



Queue with a Singly Linked List

- We can implement a queue with a singly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time



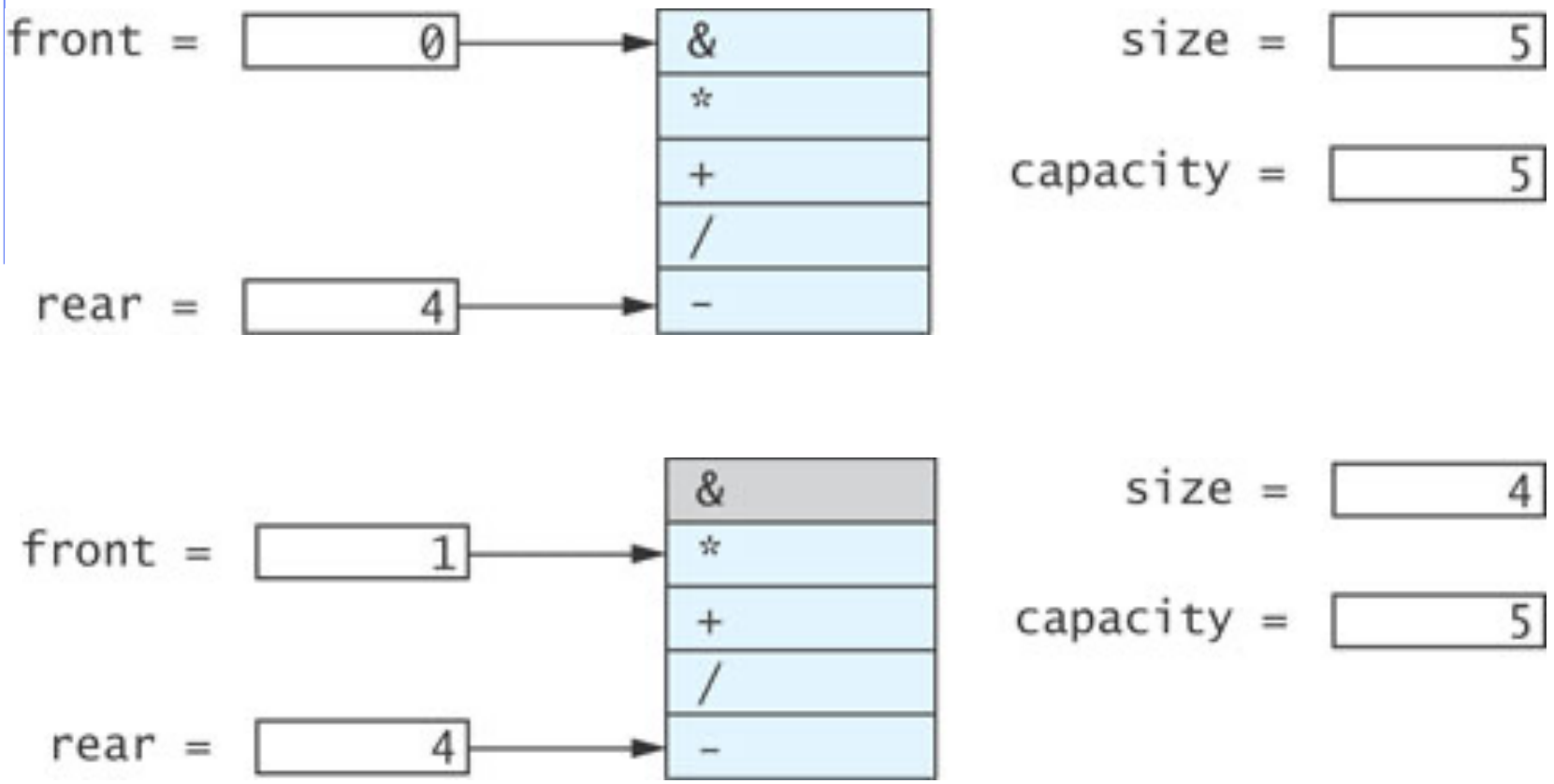
Queue ADT Implementation

- Can be based on either an array or a linked list
- Linked List
  - Implementation is straightforward
- Array
  - Need to maintain pointers to index of front and rear elements
  - Need to wrap around to the front after repeated insert and remove operations using circular array
  - May have to enlarge the array

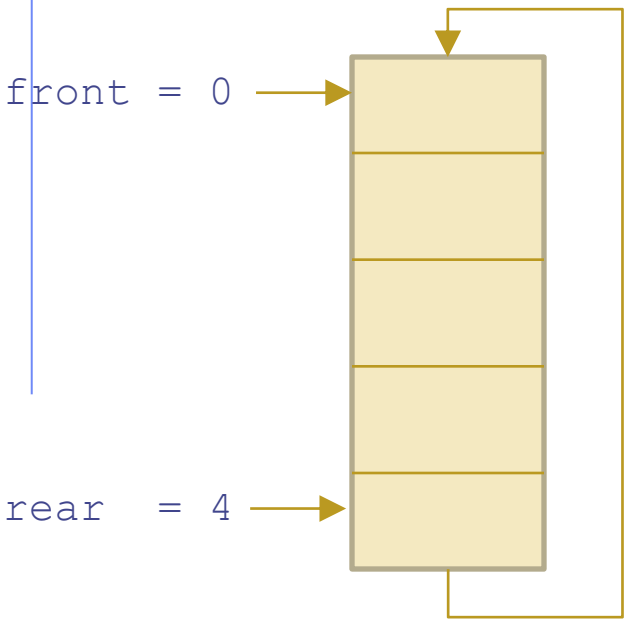
Implementing a Queue Using a Circular Array

- The time efficiency of using a single- or double-linked list to implement a queue is acceptable
- However, there are some space inefficiencies
- Storage space is increased when using a linked list due to references stored in the nodes
- Array Implementation
  - Insertion at rear of array is constant time O(1)
  - Removal from the front is linear time O(*n*), due to shifting down the elements.
  - Removal from rear of array is constant time O(1)
  - Insertion at the front is linear time O(*n*)
- We now discuss how to avoid these inefficiencies in an array using circular way of array implementation.

Implementing a Queue Using a Circular Array (cont.)

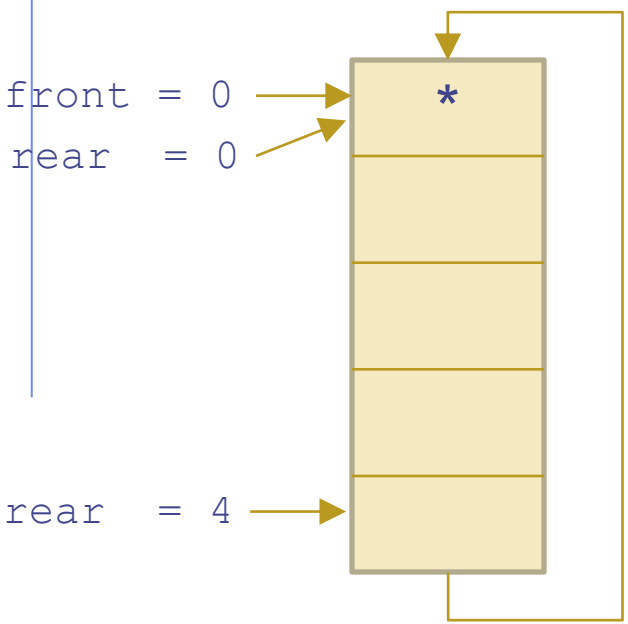


Implementing a Queue Using a Circular Array (cont.)




```
size = 0 // How many elements hold
Capacity = 5 // Length of the array

  theData = new array
  front = 0;
  rear = capacity - 1;
```

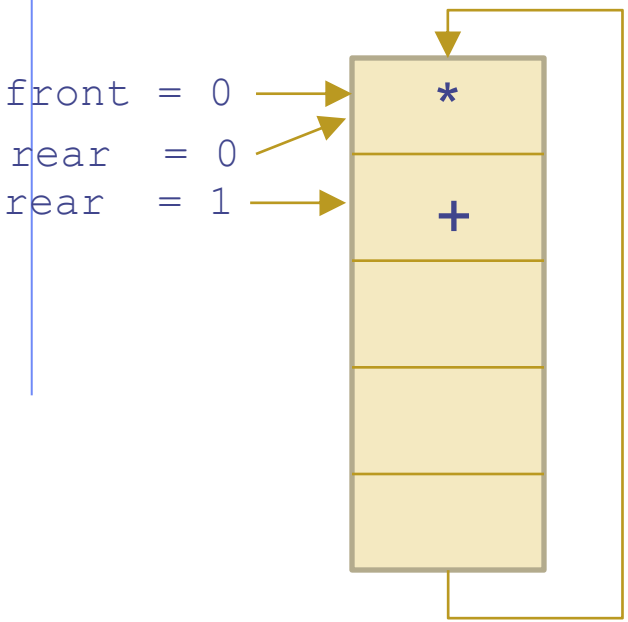
Implementing a Queue Using a Circular Array (cont.)




```
q.enqueue('*');

size     = 1
capacity = 5
```

```
Algorithm enqueue(item)
  if (size == capacity)
    throw EmptyQueueException()

  size++;
  rear = (rear + 1) % capacity;//(0+1)%5 = 1
  theData[rear] = item;
  return true;
```

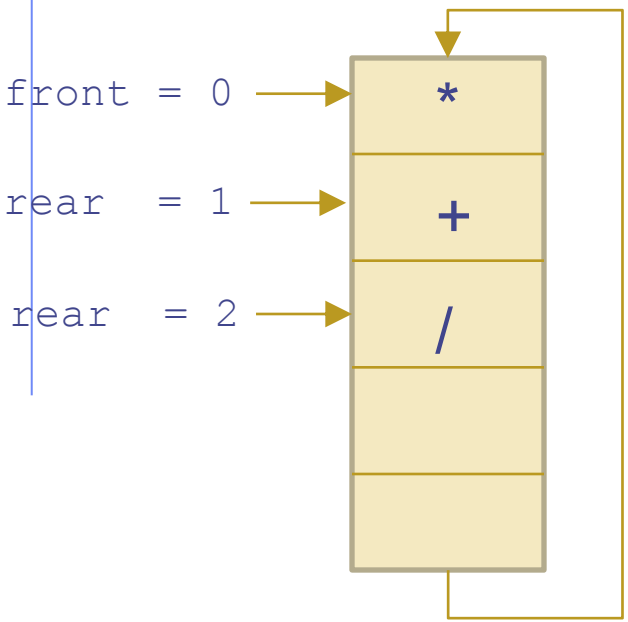
Implementing a Queue Using a Circular Array (cont.)




```
q.enqueue('+');

size     = 2

capacity = 5
```

```
Algorithm enqueue(item)
  if (size == capacity)
    throw EmptyQueueException()

  size++;
  rear = (rear + 1) % capacity;//(0+1)%5 = 1
  theData[rear] = item;
  return true;
```

Implementing a Queue Using a Circular Array (cont.)

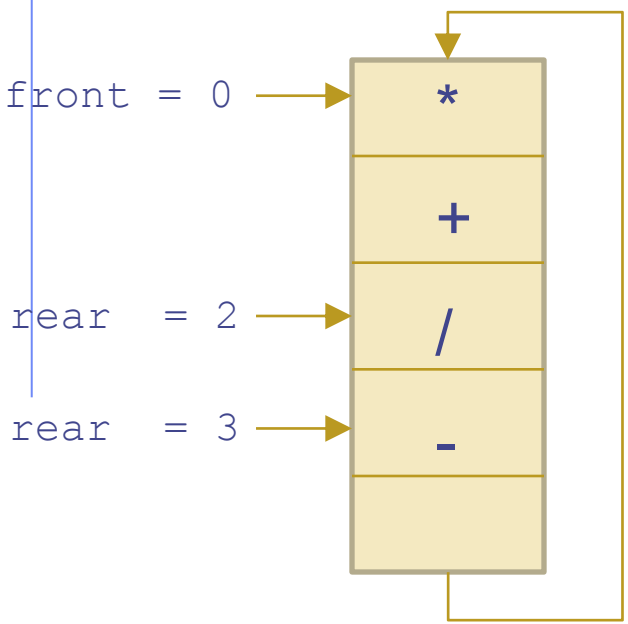



```
q.enqueue('/');
```

```
size     = 3
capacity = 5
```

```
Algorithm enqueue(item)
  if (size == capacity)
    throw EmptyQueueException()

  size++;
  rear = (rear + 1) % capacity;//(1+1)%5 = 2
  theData[rear] = item;
  return true;
```

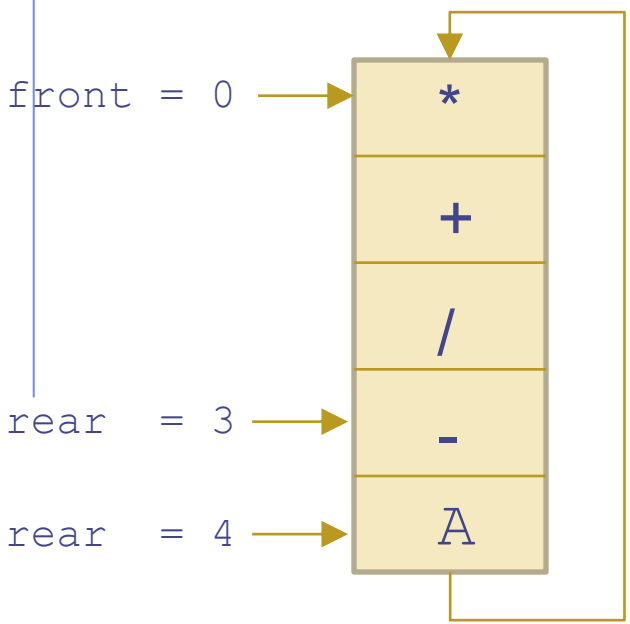
Implementing a Queue Using a Circular Array (cont.)




```
q.enqueue('-');

size     = 4
capacity = 5
```

```
Algorithm enqueue(item)
  if (size == capacity)
    throw EmptyQueueException()

  size++;
  rear = (rear + 1) % capacity;//(2+1)%5 = 3
  theData[rear] = item;
  return true;
```

Implementing a Queue Using a Circular Array (cont.)

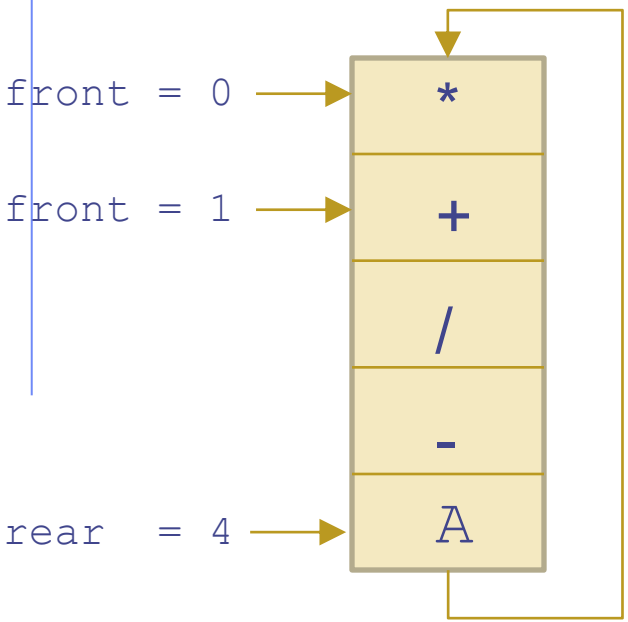



```
q.enqueue('A');
```

```
size     = 5
capacity = 5
```

```
Algorithm enqueue(item)
  if (size == capacity)
    throw EmptyQueueException()

  size++;
  rear = (rear + 1) % capacity;//(3+1)%5 = 4
  theData[rear] = item;
  return true;
```

Implementing a Queue Using a Circular Array (cont.)




```
next = q.dequeue();
```

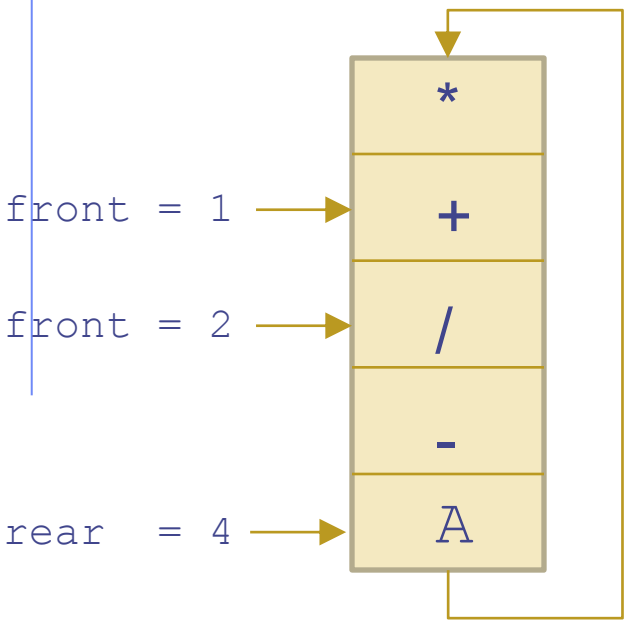
size = 4

capacity = 5

```
Algorithm dequeue()
  if (size == 0)
    return null

  result = theData[front];
  front = (front + 1) % capacity; //(0+1)%5 = 1
  size--;
  return result;
```

result = '*'

Implementing a Queue Using a Circular Array (cont.)




```
next = q.dequeue();
```

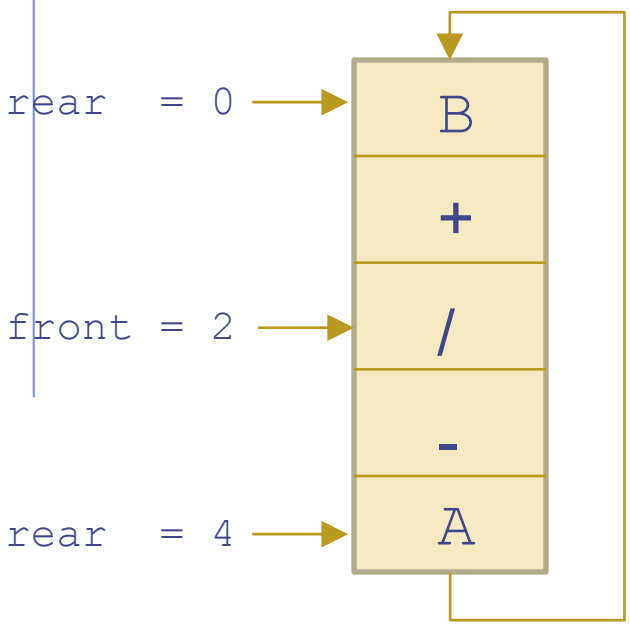
size = 3

capacity = 5

```
Algorithm dequeue()
  if (size == 0)
    return null

  result = theData[front];
  front = (front + 1) % capacity; //(0+1)%5 = 1
  size--;
  return result;
```

result = '+'

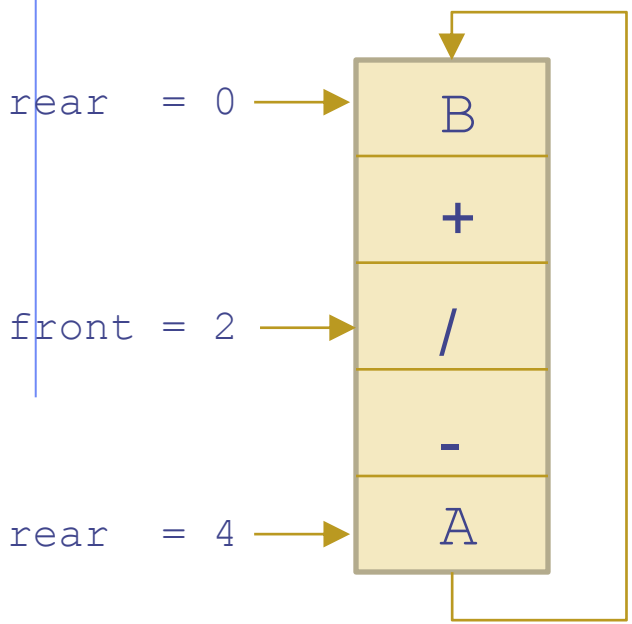
Implementing a Queue Using a Circular Array (cont.)




```
q.enqueue('B');

size     = 4
capacity = 5
```

```
Algorithm enqueue(item)
  if (size == capacity)
    reallocate();

  size++;
  rear = (rear + 1) % capacity;//(4+1)%5 = 0
  theData[rear] = item;
  return true;
}
```

Implementing a Queue Using a Circular Array (cont.)




```
q.dequeue();

size     = 4
capacity = 5
```

```
Algorithm dequeue()
  if (size == 0)
    return null

  result = theData[front];
  front = (front + 1)% capacity;//(0+1)%5= 1
  size--;
  return result;
```

result = '/'

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. The List ADT may be used as an all-purpose class for storing collections of objects with only *sequential access* to its elements.

2. The underlying implementation of an ADT determines its efficiency depending on how that data structure is going to be used in practice.

3. **Transcendental Consciousness** is the unbounded, silent field of pure order and efficiency.
4. **Impulses within Transcendental Consciousness**: Within this field, the laws of nature continuously organize and govern all activities and processes in creation.
5. **Wholeness moving within itself**: In Unity Consciousness, when the home of all knowledge has become fully integrated in all phases of life, life is spontaneously lived in accord with natural law for maximum achievement with minimum effort.