# Review and Relatives of Big-O

## Review of Big Oh notation

**Definition**:

f(n) is O(g(n)) if there is a constant c > 0 and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

- f(n) is O(g(n)) means that
  - f(n) is asymptotically less than g(n)
  - g(n) is an asymptotic upper bound on f(n)

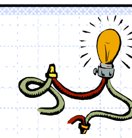


## Relatives of Big-Oh

- **big-Omega**
  - f(n) is $\Omega$(g(n)) if there is a constant c > 0 and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$
- **big-Theta**
  - f(n) is $\Theta$(g(n)) if there are constants c' > 0 and c'' > 0 and an integer constant $n_0 \geq 1$ such that $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$ for $n \geq n_0$
- **little-oh**
  - f(n) is o(g(n)) if, for any constant c > 0, there is an integer constant $n_0 > 0$ such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$
- **little-omega**
  - f(n) is $\omega$(g(n)) if, for any constant c > 0, there is an integer constant $n_0 > 0$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$



## Intuition for Asymptotic Notation



**Big-Oh**
- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

**big-Omega**
- f(n) is $\Omega$(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

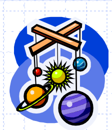
**big-Theta**
- f(n) is $\Theta$(g(n)) if f(n) is asymptotically **equal** to g(n)

**little-oh**
- f(n) is o(g(n)) if f(n) is asymptotically **strictly less** than g(n)

**little-omega**
- f(n) is $\omega$(g(n)) if f(n) is asymptotically **strictly greater** than g(n)



## Example Uses of the Relatives of Big-Oh



- **$5n^2$ is $\Omega(n^2)$**

  f(n) is $\Omega$(g(n)) if there is a constant c > 0 and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

  let c = 5 and $n_0 = 1$
- **$5n^2$ is $\Omega(n)$**

  f(n) is $\Omega$(g(n)) if there is a constant c > 0 and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

  let c = 1 and $n_0 = 1$
- **$5n^2$ is $\omega(n)$**

  f(n) is $\omega$(g(n)) if, for any constant c > 0, there is an integer constant $n_0 > 0$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

  need $5n_0^2 \geq c \cdot n_0 \rightarrow$ given c, the $n_0$ that satifies this is $n_0 \geq c/5 > 0$



## Asymptotic Notation in Practice

- The fastest algorithm in practice or for practical size input data sets is not always revealed!!!
- Because
  - Constants are dropped
  - Low-order terms are dropped
  - Algorithm efficiencies on small input sizes are not considered
- However, asymptotic notation is very effective
  - for comparing the scalability of different algorithms as input sizes become large

## Lecture 2: Stacks, Queues, Vectors, Lists, Sequences

Pure Knowledge Has
Infinite Organizing Power

## Wholeness Statement

Knowledge of data structures allows us to pick the most appropriate data structure for any computer task, thereby maximizing efficiency. Pure knowledge has infinite organizing power, and administers the whole universe with minimum effort.



## What is a type?



## Algorithms and Data Structures

- Closely linked
- Algorithm
  - a step by step procedure for performing 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)

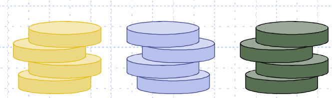


## Abstract Data Types (ADTs)

- 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
  - Vector
  - List
  - Sequence



## Stacks



## Outline and Reading

- The Stack ADT (§2.1.1)
- Applications of Stacks (§2.1.1)
- Array-based implementation (§2.1.1)

- Growable array-based Stack (tomorrow)



## The Stack ADT

- 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



## Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures



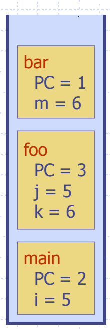
## Runtime Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the JVM pushes onto the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack

```
main() {
  int i = 5;
  foo(i);
}

foo(int j) {
  int k;
  k = j+1;
  bar(k);
}

bar(int m) {
  …
}
```



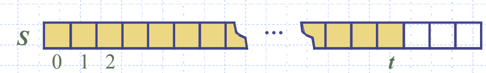



## Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- Elements are added from left to right
- A variable keeps track of the index of the top element

**Algorithm** *size*()
  **return** $t + 1$

**Algorithm** *pop*()
  **if** *isEmpty*() **then**
    **throw** *EmptyStackException*
  **else**
    $t \leftarrow t - 1$
    **return** $S[t + 1]$



## Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a StackFullException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

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




## 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

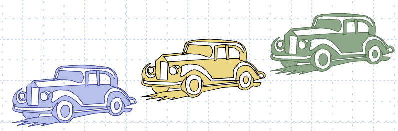


## Main Point

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. 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 (§2.1.2)
- Implementation with a circular array (§2.1.2)
- Queue interface in Java
- Growable array-based queue (tomorrow)



## 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 dequeue or front on an empty queue throws an EmptyQueueException

## Applications of Queues

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



## Queue ADT Implementation

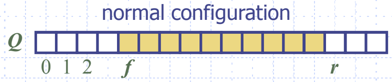
- Can be based on either an array or a linked list
- Linked List
  - Implementation is straightforward
- Array
  - Need to maintain pointers to index of front and rear elements
  - Need to wrap around to the front after repeated enqueue and dequeue operations
  - May have to enlarge the array



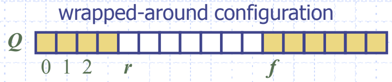
## Array-based Queue

- Use an array of size $N$ in a circular fashion
- Two variables keep track of the front and rear
  - $f$ index of the front element
  - $r$ index immediately past the rear element
- Array location $r$ is kept empty

normal configuration




wrapped-around configuration



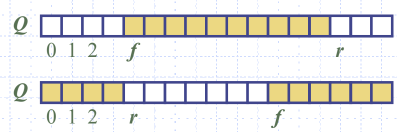



## Queue Operations

- We use the modulo operator (remainder of division)

**Algorithm** *size*()
  **return** $(N - f + r) \bmod N$

**Algorithm** *isEmpty*()
  **return** $(f = r)$

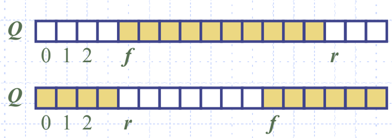





## Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

**Algorithm** *enqueue*($o$)
  **if** *size*() $= N - 1$ **then**
    **throw** *FullQueueException*
  **else**
    $Q[r] \leftarrow o$
    $r \leftarrow (r + 1) \bmod N$

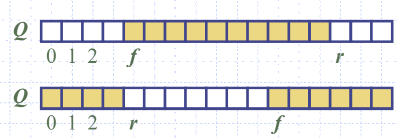





## Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

**Algorithm** *dequeue*()
  **if** *isEmpty*() **then**
    **throw** *EmptyQueueException*
  **else**
    $o \leftarrow Q[f]$
    $f \leftarrow (f + 1) \bmod N$
    **return** $o$



## A Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Requires the definition of exception classes
- No corresponding built-in Java class

```
public interface Queue {
    public int size();
    public boolean isEmpty();
    public Object front()
        throws EmptyQueueException;
    public void enqueue(Object o)
        throws FullQueueException;
    public Object dequeue()
        throws EmptyQueueException;
}
```

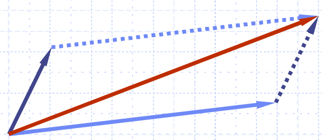


## Main Point

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). Similarly, nature is orderly, e.g., an apple seed when planted properly will yield only an apple tree.



## Vectors



## Outline and Reading

- The Vector ADT (§2.2.1)
- Array-based implementation (§2.2.1)



## The Vector ADT

- A Vector stores a sequence of elements
- Element access is based on the concept of Rank
  - Rank is the number of elements that precede an element in the sequence
- An element can be accessed, inserted, or removed by specifying its rank
- An exception is thrown if an incorrect rank is specified (e.g., a negative rank)



## Main Vector operations:

object elemAtRank(integer r):
- returns the element at rank r without removing it

object replaceAtRank(integer r, object o):
- replace the element at rank r with o and return the old element

void insertAtRank(integer r, object o):
- insert a new element o to have rank r

object removeAtRank(integer r):
- removes and returns the element at rank r

- Additional operations size() and isEmpty()

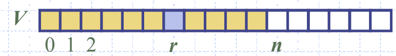
## Applications of Vectors

- Direct applications
  - Sorted collection of objects (elementary database)
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures



## Array-based Vector

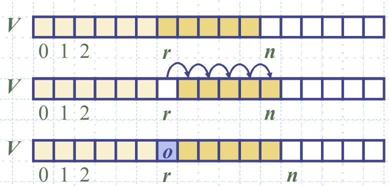
- Use an array $V$ of size $N$
- A variable $n$ keeps track of the size of the vector (number of elements stored)
- Operation ***elemAtRank***$(r)$ is implemented in $O(1)$ time by returning $V[r]$






## Insertion

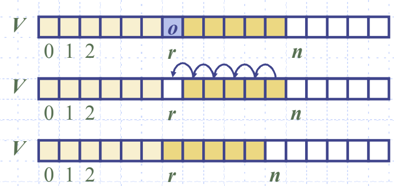
- In operation ***insertAtRank***$(r, o)$, we need to make room for the new element by shifting forward the $n - r$ elements $V[r], \ldots, V[n - 1]$
- In the worst case $(r = 0)$, this takes $O(n)$ time






## Deletion

- In operation ***removeAtRank***$(r)$, we need to fill the hole left by the removed element by shifting backward the $n - r - 1$ elements $V[r + 1], \ldots, V[n - 1]$
- In the worst case $(r = 0)$, this takes $O(n)$ time






## Performance

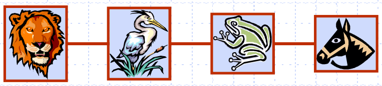
- In the array based implementation of a Vector
  - The space used by the data structure is $O(n)$
  - ***size***, ***isEmpty***, ***elemAtRank*** and ***replaceAtRank*** run in $O(1)$ time
  - ***insertAtRank*** and ***removeAtRank*** run in $O(n)$ time
- If we use the array in a circular fashion, ***insertAtRank***$(0)$ and ***removeAtRank***$(0)$ run in $O(1)$ time
- In an ***insertAtRank*** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one



## Main Point

3. *Rank* is the number of elements that precede an element in a linear sequence; this is a very simple idea, yet is the powerful basis of the random access operations of the Vector ADT. Pure consciousness is the simplest state of awareness, yet is the source of all activity in the universe.

# List ADT





# Outline and Reading

- Singly linked list
- Position ADT and List ADT (§2.2.2)
- Doubly linked list (§ 2.2.2)



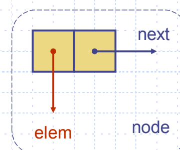
# Linked List

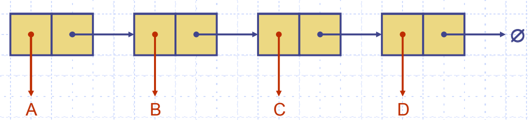
- Motivation:
  - need to handle varying amounts of data
  - eliminate the need to resize the array
  - grows and shrinks exactly when necessary
  - efficient handling of insertion or removal from the middle of the data structure
  - random access is often unnecessary
- Built-in list data structures
  - Lisp, Scheme, ML, Haskell



# Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node



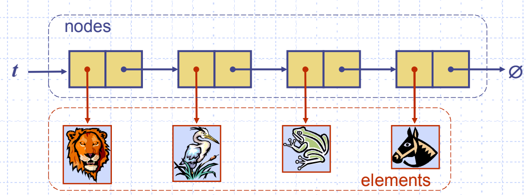



- Are there any ADT's that could be implemented efficiently with a linked list?



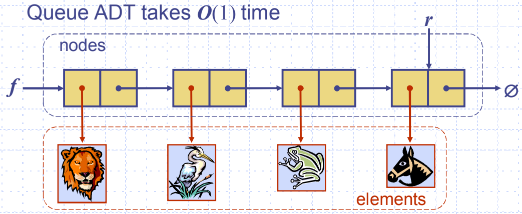
# Stack with a Singly Linked List

- 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



## 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






- What about the Vector ADT?



## Key Idea

- Elements are accessed by Position
- Position is an ADT that models a particular place or location in a data structure
- We will use this abstraction in several data structures (today in the List ADT)
- We can think of List ADT as being like a Java Interface that is implemented in different ways



## Position ADT

- The Position ADT models the notion of place within a data structure where a single object is stored
- It gives a unified view of diverse ways of storing data, such as
  - a cell of an array
  - a node of a linked list or tree
- Just one method:
  - object element(): returns the element stored at the position



## List ADT

- The List ADT models a sequence of positions storing arbitrary objects
- It establishes a before/after relation between positions
- Generic methods:
  - size(), isEmpty()
- Query methods:
  - isFirst(p), isLast(p)

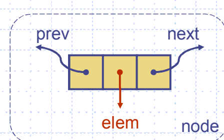
Accessor methods:
- first(), last()
- before(p), after(p)

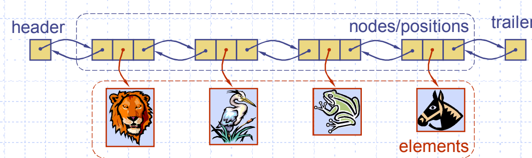
- Update methods:
  - replaceElement(p, e), swapElements(p, q)
  - insertBefore(p, e), insertAfter(p, e),
  - insertFirst(e), insertLast(e)
  - remove(p)



## Doubly Linked List

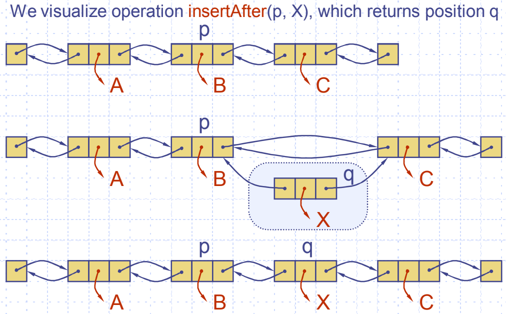
- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special header and trailer nodes



# Insertion

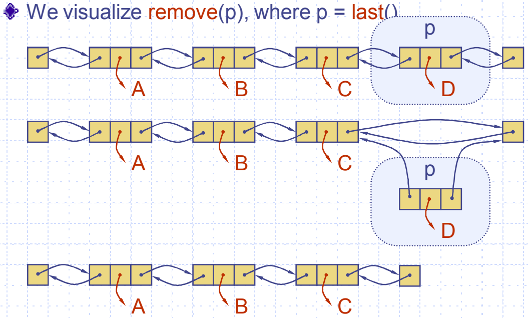
- We visualize operation insertAfter(p, X), which returns position q






# Deletion

- We visualize remove(p), where p = last()






# Performance of Linked List implementation of List ADT

- Generic methods:
  - size(), isEmpty()
- Query methods:
  - isFirst(p), isLast(p)
- Accessor methods:
  - first(), last()
  - before(p), after(p)
- Update methods:
  - replaceElement(p, e), swapElements(p, q)
  - insertBefore(p, e), insertAfter(p, e),
  - insertFirst(e), insertLast(e)
  - remove(p)



# Performance

- In the implementation of the List ADT by means of a doubly linked list
  - The space used by a list with $n$ elements is $O(n)$
  - The space used by each position of the list is $O(1)$
  - All the operations of the List ADT run in $O(1)$ time
  - Operation element() of the Position ADT runs in $O(1)$ time



# Main Point

4. The algorithm designer needs to consider how a sequence of objects is going to be used because linked lists are much more efficient than arrays (vectors) when many insertions or deletions need to be made to random parts of a sequence (or list).
   Nature always functions with maximum efficiency and minimum effort.



# Sequence ADT

# Outline and Reading

- Sequence ADT (§ 2.2.3)
- Implementations of the sequence ADT (§ 2.2.3)
- Iterators (2.2.3)



# Sequence ADT

- The Sequence ADT is the union of the Vector and List ADTs
- Elements accessed by
  - Rank, or
  - Position
- Generic methods:
  - size(), isEmpty()
- Vector-based methods:
  - elemAtRank(r), replaceAtRank(r, o), insertAtRank(r, o), removeAtRank(r)
- List-based methods:
  - first(), last(), before(p), after(p), replaceElement(p, o), swapElements(p, q), insertBefore(p, o), insertAfter(p, o), insertFirst(o), insertLast(o), remove(p)
- Bridge methods:
  - atRank(r), rankOf(p)



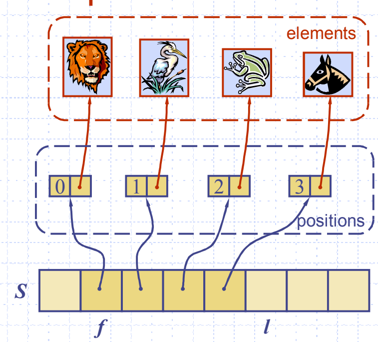
# Applications of Sequences

- The Sequence ADT is a basic, general-purpose, data structure for storing an ordered collection of elements
- Direct applications:
  - Generic replacement for stack, queue, vector, or list
  - small database (e.g., address book)
- Indirect applications:
  - Building block of more complex data structures



# Array-based Implementation

- We use a circular array storing positions
- A position object stores:
  - Element
  - Rank
- Indices $f$ and $l$ keep track of first and last positions






# Sequence Implementations

| Operation | Array | List |
|---|---|---|
| size, isEmpty | | |
| atRank(r), elemAtRank(r) | | |
| replaceAtRank(r, o) | | |
| insertAtRank(r, o), removeAtRank(r, o) | | |



# Sequence Implementations

| Operation | Array | List |
|---|---|---|
| size, isEmpty | 1 | 1 |
| atRank(r), elemAtRank(r) | 1 | $n$ |
| replaceAtRank(r, o) | 1 | $n$ |
| insertAtRank(r, o), removeAtRank(r, o) | $n$ | $n$ |

# Sequence Implementations

| Operation | Array | List |
|---|---|---|
| rankOf(p) | | |
| first(), last() | | |
| before(p), after(p) | | |
| replaceElement(p, o), swapElements(p, q) | | |
| insertFirst(o), insertLast(o) | | |
| insertAfter(p, o), insertBefore(p, o) | | |
| remove(p) | | |



# Sequence Implementations

| Operation | Array | List |
|---|---|---|
| rankOf(p) | 1 | $n$ |
| first(), last() | 1 | 1 |
| before(p), after(p) | 1 | 1 |
| replaceElement(p, o), swapElements(p, q) | 1 | 1 |
| insertFirst(o), insertLast(o) | 1 | 1 |
| insertAfter(p, o), insertBefore(p, o) | $n$ | 1 |
| remove(p) | $n$ | 1 |



# Iterators

- An iterator abstracts the process of scanning through a collection of elements
- Methods of the *ObjectIterator* ADT:
  - boolean hasNext()
  - object nextObject()
  - reset()
- Extends the concept of Position by adding a traversal capability
- Implementation with an array or singly linked list
- An iterator is typically associated with another data structure
- We can augment the Stack, Queue, Vector, List and Sequence ADTs with method:
  - ObjectIterator elements()
- Two notions of iterator:
  - snapshot: freezes the contents of the data structure at a given time
  - dynamic: follows changes to the data structure



# Main Point

5. The Sequence ADT captures the abstract notion of a mathematical sequence; it specifies the operations that any list or vector should support. The specifications of the Sequence ADT can be satisfied based on different implementation strategies with different concrete implementations. Likewise, pure awareness is an abstraction of individual awareness; each individual provides a specific, concrete realization of unbounded, unmoving pure awareness.



# Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. The Sequence 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.