What makes an algorithm “best”?

Trade-offs in the algorithms?

- Lots of “if” statements
  - Lots of “if” statements, but centralized in functions
- Cascaded “if … else” statements
- Switch statement
- Boundary and rates in arrays for us to search
  - Linear search
  - Binary search
  - Use a formula to calculate the array entry that we want
- 2d array of all the data

Trade-offs

- One independent “if” statement for each case
  - Pro: quick to code; every statement is self-contained to check
  - Con: lots of repetition; an error in one case is easy to miss; more elements in the “if” expressions
- Set of “if - else” statements
  - Pro: can be made more efficient in terms of number of tests done; “if” expressions can be simpler
  - Con: need to carry implicit information as you get into nested “if”s
- Switch statement
  - Pro: boundaries are easily maintained; no complicated “if” structure; the compiler makes the check efficient
  - Con: new cases still need expanded code; encoding of multiple criteria may not seem natural

Trade-offs

- **Two-dimensional array for each country/rate combination**
  - Pro: expands easily to any number of boundaries and rates; checking on correct values has us look at the table values rather than search through the code
  - Con: code looks more complex; more space needed for variables to store the rates; common rates aren't combined; implicit connection between the different arrays
- **Two-dimensional array for all possible country/weight combination**
  - Pro: very fast lookup of a rate; same lookup time for any weight (good for real-time systems)
  - Con: uses lots of space; takes time to load up the initial table

What is being traded-off?

- Efficiency
- Memory / space
- Complexity / difficulty to code
- Maintainability
- Readability
- Understandability
- Expandability

Does our program work? Test cases

- Any program that solves the problem should pass tests based on the requirements
  - Don't need an implementation to define the test
  - The test is meaningful no matter the implementation
  - Called opaque tests / blackbox tests
- Some tests may probe specific aspects of your implementation approach
  - The test is meaningful for one implementation but may not seem meaningful for another implementation
  - Called transparent tests / whitebox tests

Does our program work? Test cases

- Independent of implementation (opaque tests / blackbox tests)
  - Case boundaries
    - Try values on either side of each weight boundary
    - Try with each country
  - Input boundaries
    - Country: each country, invalid country numbers
    - Weight: negative, zero, 1-500, 501 or more
    - Non-integer values when integers are expected
  - Output cases
    - Value < $1, value with one integer digit, value with two integer digits

Does our program work? Test cases

- **Dependent on implementation (transparent / whitebox)**
  - **Multiple "if" statements**
    - **Try each case and ensure appropriate output**
  - **Linear search in array**
    - **Search for first, middle, and last entry**
    - **Search for entry not in the array**
  - **Binary search in array**
    - **Search for element requiring**
      - Left – left search
      - Left – right search
      - Right – left search
      - Right – right search

Opaque / blackbox test cases

Weight	Canada	US
1	0.85	1.20
30	0.85	1.20
31	1.20	1.80
50	1.20	1.80
51	1.80	2.95
100	1.80	2.95
101	2.95	5.15
200	2.95	5.15
201	4.10	10.30
300	4.10	10.30
301	4.70	10.30
400	4.70	10.30
401	5.05	10.30
500	5.05	10.30

Weight	Canada	US
-1	?	?
0	?	?
501	?	?
String	?	?

Country:
0 -- ?
3 -- ?
String -- ?

The problem doesn't specify what to do in the bad cases, so document your assumption and ensure that your code does it.

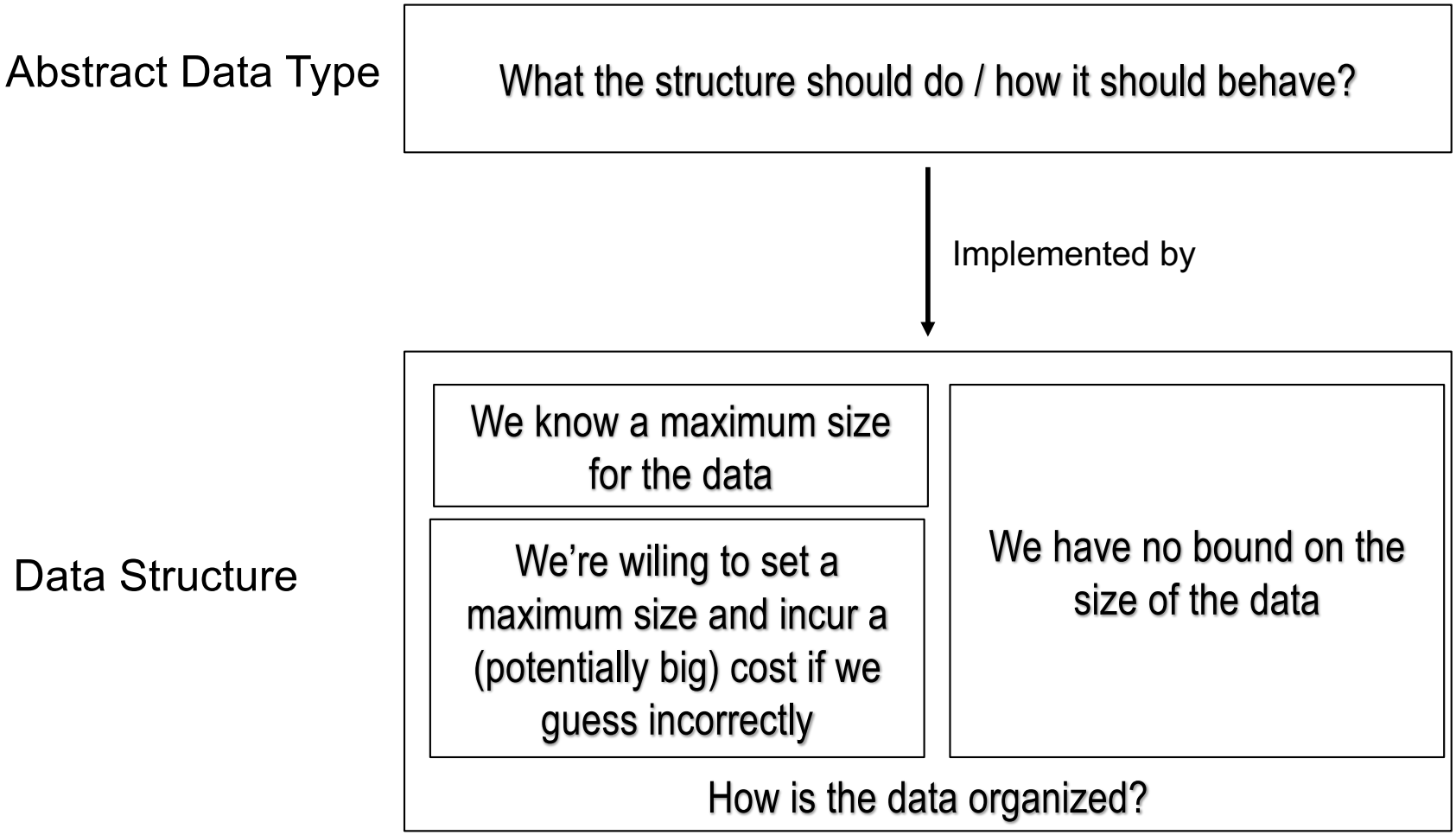
In general, report an error condition.

Abstract Data Types and Data Structures

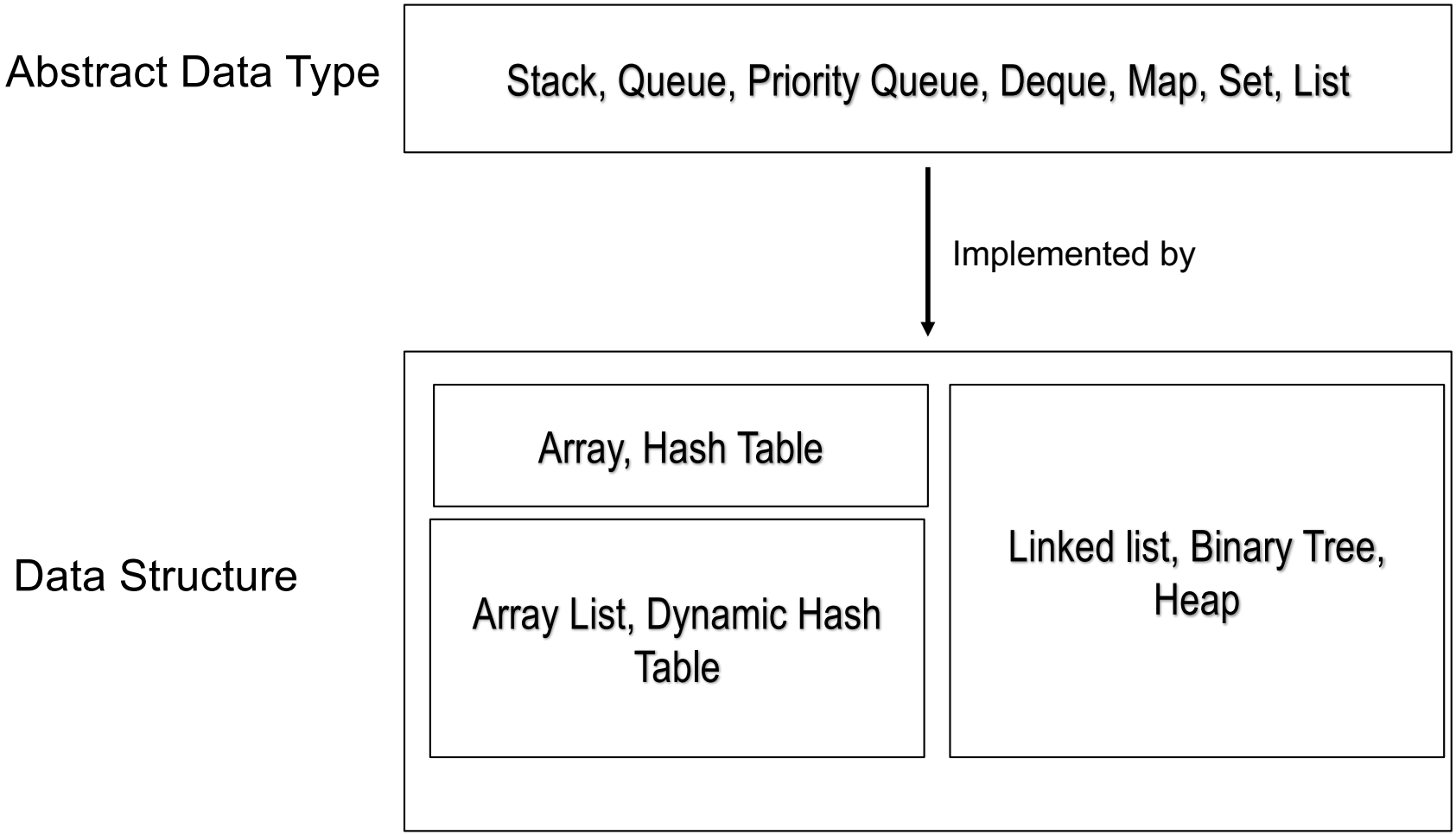
Abstract Data Type vs Data Structure

- **Abstract Data Type (ADT)**
  - "a mathematical model for data types, where a data type is defined by its behavior (semantics) from the point of view of a *user* of the data"
    (https://en.wikipedia.org/wiki/Abstract_data_type, September 6, 2018)
- **Data Structure**
  - "a data organization, management and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data values, the *relationships among them*, and the functions or operations that can be applied to the data."
    (https://en.wikipedia.org/wiki/Data_structure, September 6, 2018. Emphasis added.)
- **An ADT can typically be implemented using different data structures.**

Abstract Data Type and Data Structures



Abstract Data Type and Data Structures



Abstract Data Type vs. Data Structure

- ADT
  - Stack
    - Push, Pop, IsEmpty
  - Queue
    - Enqueue, Dequeue, IsEmpty
  - PriorityQueue
    - AddWithPriority, RemoveHighestPriority, IsEmpty
  - Deque
    - AddHead, AddTail, RemoveHead, RemoveTail, IsEmpty
  - Map
    - Store, Retrieve, Delete
  - Set
    - Add, ElementOf, Union, Intersection, Complement, Cardinality

- Data Structure
  - Array
  - Hash table
    - Relationship between data and storage location
  - Linked list
  - Binary tree
    - Relationship between children and parents
  - Heap
    - Relationship between children and parents

Stack

Queue

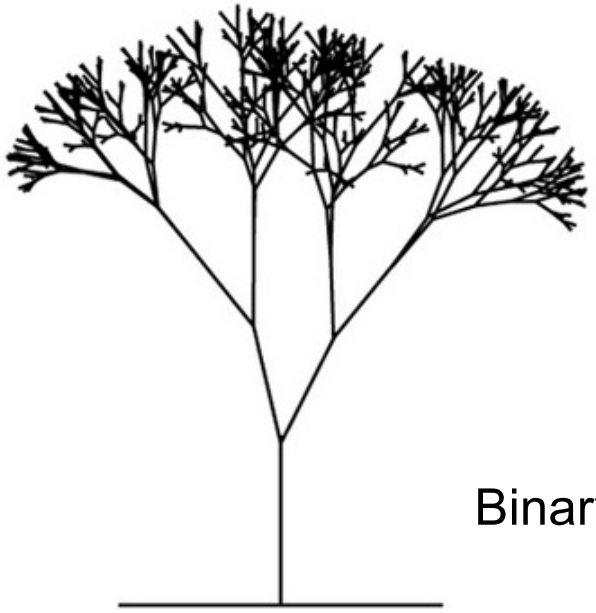
Map

Images from google images

Array

Linked list

Hash table



Binary tree

Heap

Images from google images

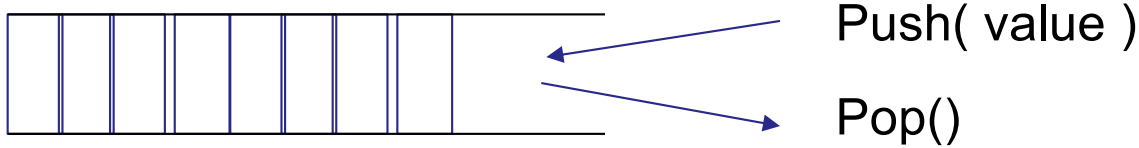
Name mixing of ADT and data structures in Java

- Java contains specific implementations of some ADTs. Don't confuse the implementation (which is a data structure) with the data type.

- For example, the Java classes of HashSet, TreeSet, and LinkedHashSet are all implementations of the set ADT:
  - HashSet: set implemented using a hash table
  - TreeSet: set implemented using a binary search tree
  - LinkedHashSet: set implemented using a combination of hash table and linked list.

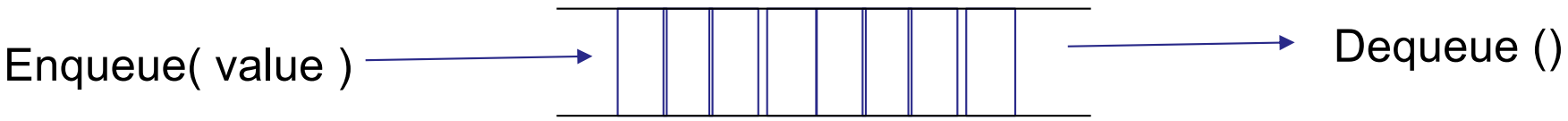
Stack

- Behaviour
  - Stores a collection of items
  - Items retrieved from the stack in last in first out order (LIFO)
  - Cannot access random elements of the collection
- Operations
  - Push(value)
  - Pop( ) -> value
  - IsEmpty( ) -> boolean



Queue

- **Behaviour**
  - Stores a collection of items
  - Items retrieved from the queue in first in first out order (FIFO)
  - Cannot access random elements of the collection
- **Operations**
  - Enqueue(value)
  - Dequeue( ) -> value
  - IsEmpty( ) -> boolean



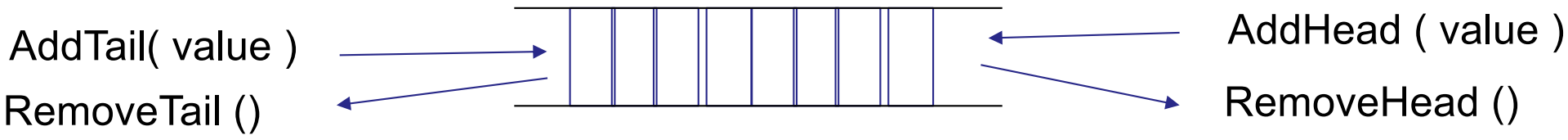
DALHOUSIE UNIVERSITY
Inspiring Minds

Deque (doubly-ended queue)

- **Behaviour**
  - Stores a collection of items
  - Concept of a queue where you can add or remove from either end (but not from the middle)
  - Cannot access random elements of the collection
- **Operations**
  - AddHead(value)
  - RemoveHead( ) -> value
  - IsEmpty( ) -> boolean

AddTail(value)

RemoveTail( ) -> value



Priority Queue

- **Behaviour**
  - Stores a collection of items where each item has a “priority” to designate importance
  - Items retrieved in order of priority
  - Cannot access random elements of the collection
- **Operations**
  - AddWithPriority(value, priority)
  - RemoveWithHighestPriority( ) -> value
  - IsEmpty( ) -> boolean

Priority Queue

- Other ATDs might be seen as an instance of a priority queue
  - Stack is a priority queue where
    - The priority number is the time of insertion
    - High priority means a high priority number
  - Queue is a priority queue where
    - The priority number is the time of insertion
    - High priority means a low priority number

Set

- Behaviour
  - Stores a collection of items
  - Concept of a mathematical set
    - Only one copy of each item allowed
  - Can access any item
  - No presumed order to the items
- Operations
  - Add(value)
  - Remove(value)
  - ElementOf(value) -> Boolean
  - Union(set1, set2) -> set
  - Intersection(set1, set2) -> set
  - Complement(universe, set) -> set (optional)
  - Cardinality(set) -> integer

List

- Behaviour
  - Stores a collection of items
  - Concept of a mathematical sequence
  - Can access any item
  - Items are ordered in the order of insertion
- Operations
  - Add(value)
  - Remove(value)
  - Contains(value) -> boolean
  - Sort()