1.1 Data structures

Data structures

A ***data structure*** is a way of organizing, storing, and performing operations on data. Operations performed on a data structure include accessing or updating stored data, searching for specific data, inserting new data, and removing data. The following provides a list of basic data structures.

Table 1.1.1: Basic data structures.

Data structure	Description
Record	A ***record*** is the data structure that stores subitems, often called fields, with a name associated with each subitem.
Array	An ***array*** is a data structure that stores an ordered list of items, where each item is directly accessible by a positional index.
Linked list	A ***linked list*** is a data structure that stores an ordered list of items in nodes, where each node stores data and has a pointer to the next node.
Binary tree	A ***binary tree*** is a data structure in which each node stores data and has up to two children, known as a left child and a right child.
Hash table	A ***hash table*** is a data structure that stores unordered items by mapping (or hashing) each item to a location in an array.
Heap	A ***max-heap*** is a tree that maintains the simple property that a node's key is greater than or equal to the node's childrens' keys. A ***min-heap*** is a tree that maintains the simple property that a node's key is less than or equal to the node's childrens' keys.
Graph	A ***graph*** is a data structure for representing connections among items, and consists of vertices connected by edges. A ***vertex*** represents an item in a graph. An ***edge*** represents a connection between two vertices in a graph.

PARTICIPATION ACTIVITY | 1.1.1: Basic data structures.

1) A linked list stores items in an unspecified order.

- True
- False

2) A node in binary tree can have zero, one, or two children.

- True
- False

3) A list node's data can store a record with multiple subitems.

- True
- False

4) Items stored in an array can be accessed using a positional index.

- True
- False

Choosing data structures

The selection of data structures used in a program depends on both the type of data being stored and the operations the program may need to perform on that data. Choosing the best data structure often requires determining which data structure provides a good balance given expected uses. Ex: If a program requires fast insertion of new data, a linked list may be a better choice than an array.

PARTICIPATION ACTIVITY 1.1.2: A linked list avoids the shifting problem.

Animation content:

undefined

Animation captions:

1. Inserting an item at a specific location in an array requires making room for the item by shifting higher-indexed items.
2. Once the higher index items have been shifted, the new item can be inserted at the desired index.
3. To insert new item in a linked list, a list node for the new item is first created.

4. Item B's next pointer is assigned to point to item C. Item A's next pointer is updated to point to item B. No shifting of other items was required.

PARTICIPATION ACTIVITY | 1.1.3: Basic data structures.

1) Inserting an item at the end of a 999-item array requires how many items to be shifted?

Check Show answer

2) Inserting an item at the end of a 999-item linked list requires how many items to be shifted?

Check Show answer

3) Inserting an item at the beginning of a 999-item array requires how many items to be shifted?

Check Show answer

4) Inserting an item at the beginning of a 999-item linked list requires how many items to be shifted?

Check Show answer

1.2 Introduction to algorithms

Algorithms

An ***algorithm*** describes a sequence of steps to solve a computational problem or perform a calculation. An algorithm can be described in English, pseudocode, a programming language, hardware, etc. A ***computational problem*** specifies an input, a question about the input that can be answered using a computer, and the desired output.

PARTICIPATION ACTIVITY | 1.2.1: Computational problems and algorithms.

Animation captions:

1. A computational problem is a problem that can be solved using a computer. A computational problem specifies the problem input, a question to be answered, and the desired output.
2. For the problem of finding the maximum value in an array, the input is an array of numbers.
3. The problem's question is: What is the maximum value in the input array? The problem's output is a single value that is the maximum value in the array.
4. The FindMax algorithm defines a sequence of steps that determines the maximum value in the array.

PARTICIPATION ACTIVITY | 1.2.2: Algorithms and computational problems.

Consider the problem of determining the number of times (or frequency) a specific word appears in a list of words.

1) Which can be used as the problem input?

- ○ String for user-specified word
- ○ Array of unique words and string for user-specified word
- ○ Array of all words in the list and string for user-specified word

2) What is the problem output?

- ○ Integer value for the frequency of most frequent word
- ○ String value for the most frequent word in input array
- ○ Integer value for the frequency of specified word

3) An algorithm to solve this computation problem must be written using a programming language.

- True
- False

Practical applications of algorithms

Computational problems can be found in numerous domains, including e-commerce, internet technologies, biology, manufacturing, transportation, etc. Algorithms have been developed for numerous computational problems within these domains.

A computational problem can be solved in many ways, but finding the best algorithm to solve a problem can be challenging. However, many computational problems have common subproblems, for which efficient algorithms have been developed. The examples below describe a computational problem within a specific domain and list a common algorithm (each discussed elsewhere) that can be used to solve the problem.

Table 1.2.1: Example computational problems and common algorithms.

Application domain	Computational problem	Common algorithm
DNA analysis	Given two DNA sequences from different individuals, what is the longest shared sequence of nucleotides?	*Longest common substring problem:* A longest common substring algorithm determines the longest common substring that exists in two input strings. DNA sequences can be represented using strings consisting of the letters A, C, G, and T to represent the four different nucleotides.
Search engines	Given a product ID and a sorted array of all in-stock products, is the product in stock and what is the product's price?	*Binary search:* The binary search algorithm is an efficient algorithm for searching a list. The list's elements must be sorted and directly accessible (such as an array).
Navigation	Given a user's current location and desired location, what is	*Dijkstra's shortest path:* Dijkstra's shortest path algorithm determines the shortest path from a start vertex to each vertex in a

	the fastest route to walk to the destination?	graph. The possible routes between two locations can be represented using a graph, where vertices represent specific locations and connecting edges specify the time required to walk between those two locations.

PARTICIPATION ACTIVITY | 1.2.3: Computational problems and common algorithms.

Match the common algorithm to another computational problem that can be solved using that algorithm.

If unable to drag and drop, refresh the page.

Binary search **Shortest path algorithm** **Longest common substring**

	Do two student essays share a common phrase consisting of a sequence of more than 100 letters?
	Given the airports at which an airline operates and distances between those airports, what is the shortest total flight distance between two airports?
	Given a sorted list of a company's employee records and an employee's first and last name, what is a specific employee's phone number?

Reset

Efficient algorithms and hard problems

Computer scientists and programmers typically focus on using and designing efficient algorithms to solve problems. Algorithm efficiency is most commonly measured by the algorithm runtime, and an efficient algorithm is one whose runtime increases no more than polynomially with respect to the input size. However, some problems exist for which an efficient algorithm is unknown.

NP-complete problems are a set of problems for which no known efficient algorithm exists. NP-complete problems have the following characteristics:

- No efficient algorithm has been found to solve an NP-complete problem.
- No one has proven that an efficient algorithm to solve an NP-complete problem is impossible.
- If an efficient algorithm exists for one NP-complete problem, then all NP-complete problems can be solved efficiently.

By knowing a problem is NP-complete, instead of trying to find an efficient algorithm to solve the problem, a programmer can focus on finding an algorithm to efficiently find a good, but non-optimal, solution.

PARTICIPATION ACTIVITY | 1.2.4: Example NP-complete problem: Cliques.

Animation captions:

1. A programmer may be asked to write an algorithm to solve the problem of determining if a set of K people who all know each other exists within a graph of a social network?
2. For the example social network graph and K = 3, the algorithm should return yes. Xiao, Sean, and Tanya all know each other. Sean, Tanya, and Eve also all know each other.
3. For K = 4, no set of 4 individual who all know each other exists, and the algorithm, should return no.
4. This problem is equivalent to the clique decision problem, which is NP-complete, and no known polynomial time algorithm exists.

PARTICIPATION ACTIVITY | 1.2.5: Efficient algorithm and hard problems.

1) An algorithm with a polynomial runtime is considered efficient.

- True
- False

2) An efficient algorithm exists for all computational problems.

- True
- False

3) An efficient algorithm to solve an NP-complete problem may exist.

- True
- False

1.3 Relation between data structures and algorithms

Algorithms for data structures

Data structures not only define how data is organized and stored, but also the operations performed on the data structure. While common operations include inserting, removing, and searching for data, the algorithms to implement those operations are typically specific to each data structure. Ex: Appending an item to a linked list requires a different algorithm than appending an item to an array.

PARTICIPATION ACTIVITY | 1.3.1: A list avoids the shifting problem.

Animation content:

undefined

Animation captions:

1. The algorithm to append an item to an array determines the current size, increases the array size by 1, and assigns the new item as the last array element.
2. The algorithm to append an item to a linked list points the tail node's next pointer and the list's tail pointer to the new node.

PARTICIPATION ACTIVITY | 1.3.2: Algorithms for data structures.

Consider the array and linked list in the animation above. Can the following algorithms be implemented with the same code for both an array and linked list?

1) Append an item

- Yes
- No

2) Return the first item

- Yes
- No

3) Return the current size

- Yes
- No

Algorithms using data structures

Some algorithms utilize data structures to store and organize data during the algorithm execution. Ex: An algorithm that determines a list of the top five salespersons, may use an array to store salespersons sorted by their total sales.

Figure 1.3.1: Algorithm to determine the top five salespersons using an array.

```
DisplayTopFiveSalespersons(allSalespersons) {
   // topSales array has 5 elements
   // Array elements have subitems for name and total sales
   // Array will be sorted from highest total sales to lowest total sales
   topSales = Create array with 5 elements

   // Initialize all array elements with a negative sales total
   for (i = 0; i < topSales⇢length; ++i) {
      topSales[i]⇢name = ""
      topSales[i]⇢salesTotal = -1
   }

   for each salesPerson in allSalespersons {
      // If salesPerson's total sales is greater than the last
      // topSales element, salesPerson is one of the top five so far
      if (salesPerson⇢salesTotal > topSales[topSales⇢length -
1]⇢salesTotal) {

         // Assign the last element in topSales with the current
salesperson
         topSales[topSales⇢length - 1]⇢name =  salesPerson⇢name
         topSales[topSales⇢length - 1]⇢salesTotal =
salesPerson⇢salesTotal

         // Sort topSales in descending order
         SortDescending(topSales)
      }
   }

   // Display the top five salespersons
   for (i = 0; i < topSales⇢length; ++i) {
      Display topSales[i]
   }
}
```


PARTICIPATION ACTIVITY 1.3.3: Top five salespersons.

1) Which of the following is *not* equal to the number of items in the topSales array?

- topSales⇢length
- 5
- allSalesperson⇢length

2) To adapt the algorithm to display the top 10 salespersons, what modifications are required?

- Only the array creation
- All loops in the algorithm
- Both the creation and all loops

3) If allSalespersons only contains three elements, the DisplayTopFiveSalespersons algorithm will display two elements with no name and -1 sales.

- True
- False

1.4 Abstract data types

Abstract data types (ADTs)

An ***abstract data type*** (***ADT***) is a data type described by predefined user operations, such as "insert data at rear," without indicating how each operation is implemented. An ADT can be implemented using different underlying data structures. However, a programmer need not have knowledge of the underlying implementation to use an ADT.

Ex: A list is a common ADT for holding ordered data, having operations like append a data item, remove a data item, search whether a data item exists, and print the list. A list ADT is commonly implemented using arrays or linked list data structures.

PARTICIPATION ACTIVITY | 1.4.1: List ADT using array and linked lists data structures.

Animation captions:

1. A new list named agesList is created. Items can be appended to the list. The items are ordered.
2. Printing the list prints the items in order.
3. A list ADT is commonly implemented using array and linked list data structures. But, a programmer need not have knowledge of which data structure is used to use the list ADT.

PARTICIPATION ACTIVITY | 1.4.2: Abstract data types.

1) Starting with an empty list, what is the list contents after the following operations?

Append(list, 11)
Append(list, 4)
Append(list, 7)

- 4, 7, 11
- 7, 4, 11
- 11, 4, 7

2) A remove operation for a list ADT removes the specified item. Given a list with contents: 2, 20, 30, what is the list contents after the following operation?

Remove(list, item 2)

- 2, 30
- 2, 20, 30
- 20, 30

3) A programmer must know the underlying implementation of the list ADT in order to use a list.

- True
- False

4) A list ADT's underlying data structure has no impact on the program's execution.

- True
- False

Common ADTs

Table 1.4.1: Common ADTs.

Abstract data type	Description	Common underlying data structures
List	A ***list*** is an ADT for holding ordered data.	Array, linked list
Dynamic array	A ***dynamic array*** is an ADT for holding ordered data and allowing indexed access.	Array
Stack	A ***stack*** is an ADT in which items are only inserted on or removed from the top of a stack.	Linked list
Queue	A ***queue*** is an ADT in which items are inserted at the end of the queue and removed from the front of the queue.	Linked list
Deque	A ***deque*** (pronounced "deck" and short for double-ended queue) is an ADT in which items can be inserted and removed at both the front and back.	Linked list
Bag	A ***bag*** is an ADT for storing items in which the order does not matter and duplicate items are allowed.	Array, linked list
Set	A ***set*** is an ADT for a collection of distinct items.	Binary search tree, hash table
Priority queue	A ***priority queue*** is a queue where each item has a priority, and items with higher priority are closer to the front of the queue than items with lower priority.	Heap
Dictionary (Map)	A ***dictionary*** is an ADT that associates (or maps) keys with values.	Hash table, binary search tree

PARTICIPATION ACTIVITY 1.4.3: Common ADTs.

Consider the ADTs listed in the table above. Match the ADT with the description of the order and uniqueness of items in the ADT.

If unable to drag and drop, refresh the page.

Bag List Set Priority queue

	Items are ordered based on how items are added. Duplicate items are allowed.
	Items are not ordered. Duplicate items are not allowed.
	Items are ordered based on items' priority. Duplicate items are allowed.
	Items are not ordered. Duplicate items are allowed.

Reset

1.5 Applications of ADTs

Abstraction and optimization

Abstraction means to have a user interact with an item at a high-level, with lower-level internal details hidden from the user. ADTs support abstraction by hiding the underlying implementation details and providing a well-defined set of operations for using the ADT.

Using abstract data types enables programmers or algorithm designers to focus on higher-level operations and algorithms, thus improving programmer efficiency. However, knowledge of the underlying implementation is needed to analyze or improve the runtime efficiency.

PARTICIPATION ACTIVITY 1.5.1: Programming using ADTs.

Animation content:

Static figure:
Program requirements are listed on the left. Two are highlighted:
- "Maintain list of 5 most recently visited websites"
- "Display websites in reverse chronological order", which is highlighted red

- "For each website visited, remove oldest entry and add new website to list", which is highlighted yellow

There are two available ADTs, List and Queue, on the right. Available operations for each ADT are below that ADT's name. Some operations are colored in corresponding ways to the highlights on the requirements.
List has these operations: Append, which is yellow; Prepend; Iterate in reverse, which is red; an ellipsis here designates that there are more operations now shown; Remove, which is yellow; GetLength. Below the last operation is the text "Implementation hidden."

Queue has these operations: Enqueue item, which is yellow; Dequeue item, which is yellow; Peek; IsEmpty; GetLength. Below the last operation is the text, "Implementation hidden." There is also text that reads, "No matching abstraction for reverse iteration through Queue."

Step 1: Abstraction simplifies programming. ADTs allow programmers to focus on choosing which ADTs best match a program's needs. The requirements and the two available ADTs are visible.

Step 2: Both the list and queue ADTs support efficient interfaces for removing items from one end (removing oldest entry) and adding items to the other end (adding new entries). The requirement, "For each website visited, remove oldest entry and add new website to list," is highlighted yellow. In the List ADT, the "Append" and "Remove" operations are colored yellow. In the Queue ADT, the "Enqueue item" and "Dequeue item" operations are colored yellow.

Step 3: The list ADT supports iterating through list contents in reverse order, but the queue ADT does not. The requirement, "Display websites in reverse chronological order," is highlighted red. In the List ADT, the "Iterate in reverse" operation is colored red. Next to the Queue ADT, the text, "No matching abstraction for reverse iteration through Queue," appears.

Step 4: To use the list (or queue) ADT, the programmer does not need to know the underlying implementation. Below each set of operations for both of the ADTs is the text, "Implementation hidden."

Animation captions:

1. Abstraction simplifies programming. ADTs allow programmers to focus on choosing which ADTs best match a program's needs.
2. Both the list and queue ADTs support efficient interfaces for removing items from one end (removing oldest entry) and adding items to the other end (adding new entries).
3. The list ADT supports iterating through list contents in reverse order, but the queue ADT does not.
4. To use the list (or queue) ADT, the programmer does not need to know the underlying implementation.

PARTICIPATION ACTIVITY | 1.5.2: Programming with ADTs.

Consider the example in the animation above.

1) The _____ ADT is the better match for the program's requirements.

- queue
- list

2) The list ADT _____.

- can only be implemented using an array
- can only be implemented using a linked list
- can be implemented in numerous ways

3) Knowledge of an ADT's underlying implementation is needed to analyze the runtime efficiency.

- True
- False

ADTs in standard libraries

Most programming languages provide standard libraries that implement common abstract data types. Some languages allow programmers to choose the underlying data structure used for the ADTs. Other programming languages may use a specific data structure to implement each ADT, or may automatically choose the underlying data-structure.

Table 1.5.1: Standard libraries in various programming languages.

Programming language	Library	Common supported ADTs
Python	Python standard library	list, set, dict, deque
C++	Standard template library (STL)	vector, list, deque, queue, stack, set, map

Java	Java collections framework (JCF)	Collection, Set, List, Map, Queue, Deque

PARTICIPATION ACTIVITY 1.5.3: ADTs in standard libraries.

1) Python, C++, and Java all provide built-in support for a deque ADT.

- True
- False

2) The underlying data structure for a list data structure is the same for all programming languages.

- True
- False

3) ADTs are only supported in standard libraries.

- True
- False

1.6 Algorithm efficiency

Algorithm efficiency

An algorithm describes the method to solve a computational problem. Programmers and computer scientists should use or write efficient algorithms. ***Algorithm efficiency*** is typically measured by the algorithm's computational complexity. ***Computational complexity*** is the amount of resources used by the algorithm. The most common resources considered are the runtime and memory usage.

PARTICIPATION ACTIVITY 1.6.1: Computational complexity.

Animation captions:

1. An algorithm's computational complexity includes runtime and memory usage.

2. Measuring runtime and memory usage allows different algorithms to be compared.
3. Complexity analysis is used to identify and avoid using algorithms with long runtimes or high memory usage.

PARTICIPATION ACTIVITY 1.6.2: Algorithm efficiency and computational complexity.

1) Computational complexity analysis allows the efficiency of algorithms to be compared.
   - True
   - False

2) Two different algorithms that produce the same result have the same computational complexity.
   - True
   - False

3) Runtime and memory usage are the only two resources making up computational complexity.
   - True
   - False

Runtime complexity, best case, and worst case

An algorithm's ***runtime complexity*** is a function, T(N), that represents the number of constant time operations performed by the algorithm on an input of size N. Runtime complexity is discussed in more detail elsewhere.

Because an algorithm's runtime may vary significantly based on the input data, a common approach is to identify best and worst case scenarios. An algorithm's ***best case*** is the scenario where the algorithm does the minimum possible number of operations. An algorithm's ***worst case*** is the scenario where the algorithm does the maximum possible number of operations.

Input data size must remain a variable

A best case or worst case scenario describes contents of the algorithm's input data only. The input data size must remain a variable, N. Otherwise, the overwhelming majority of algorithms would have a best case of N=0, since no input data would be processed. In both theory and practice, saying "the best case is when the algorithm doesn't process any data" is not useful. Complexity analysis always treats the input data size as a variable.

PARTICIPATION ACTIVITY | 1.6.3: Linear search best and worst cases.

Animation captions:

1. LinearSearch searches through array elements until finding the key. Searching for 26 requires iterating through the first 3 elements.
2. The search for 26 is neither the best nor the worst case.
3. Searching for 54 only requires one comparison and is the best case: The key is found at the start of the array. No other search could perform fewer operations.
4. Searching for 82 compares against all array items and is the worst case: The number is not found in the array. No other search could perform more operations.

PARTICIPATION ACTIVITY | 1.6.4: FindFirstLessThan algorithm best and worst case.

Consider the following function that returns the first value in a list that is less than the specified value. If no list items are less than the specified value, the specified value is returned.

```
FindFirstLessThan(list, listSize, value) {
   for (i = 0; i < listSize; i++) {
      if (list[i] < value)
         return list[i]
   }
   return value // no lesser value found
}
```

If unable to drag and drop, refresh the page.

Best case **Worst case** **Neither best nor worst case**

	No items in the list are less than `value`.
	The first half of the list has elements greater than `value` and the second

half has elements less than `value`.

The first item in the list is less than `value`.

Reset

PARTICIPATION ACTIVITY 1.6.5: Best and worst case concepts.

1) The linear search algorithm's best case scenario is when N = 0.

- True
- False

2) An algorithm's best and worst case scenarios are always different.

- True
- False

Space complexity

An algorithm's ***space complexity*** is a function, S(N), that represents the number of fixed-size memory units used by the algorithm for an input of size N. Ex: The space complexity of an algorithm that duplicates a list of numbers is S(N) = 2N + k, where k is a constant representing memory used for things like the loop counter and list pointers.

Space complexity includes the input data and additional memory allocated by the algorithm. An algorithm's ***auxiliary space complexity*** is the space complexity not including the input data. Ex: An algorithm to find the maximum number in a list will have a space complexity of S(N) = N + k, but an auxiliary space complexity of S(N) = k, where k is a constant.

PARTICIPATION ACTIVITY 1.6.6: FindMax space complexity and auxiliary space complexity.

Animation content:

undefined

Animation captions:

1. FindMax's arguments represent input data. Non-input data includes variables allocated in the function body: maximum and i.
2. The list's size is a variable, N. Three integers are also used, listSize, maximum, and i, making the space complexity S(N) = N + 3.
3. The auxiliary space complexity includes only the non-input data, which does not increase for larger input lists.
4. The function's auxiliary space complexity is S(N) = 2.

PARTICIPATION ACTIVITY 1.6.7: Space complexity of GetEvens function.

Consider the following function, which builds and returns a list of even numbers from the input list.

```
GetEvens(list, listSize) {
   i = 0
   evensList = Create new, empty list
   while (i < listSize) {
      if (list[i] % 2 == 0)
         Add list[i] to evensList
      i = i + 1
   }
   return evensList
}
```

1) What is the maximum possible size of the returned list?

- listSize
- listSize / 2

2) What is the minimum possible size of the returned list?

- listSize / 2
- 1
- 0

3) What is the worst case auxiliary space complexity of GetEvens if N is the list's size and k is a constant?

- S(N) = N + k
- S(N) = k

4) What is the best case auxiliary space complexity of GetEvens if N is the list's size and k is a constant?

- S(N) = N + k
- S(N) = k

1.7 LAB: Introduction to data structures labs

This section's content is not available for print.