1. Introduction to Data Structures and Algorithms

1.1 Data structures

data structure	A data structure is a way of organizing, storing, and performing operations on data.
record	A record is the data structure that stores subitems, with a name associated with each subitem.
array	An array is a data structure that stores an ordered list of items, with each item is directly accessible by a positional index.
linked list	A linked list is a data structure that stores 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.
max-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.
min-heap	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.

vertex	A vertex represents an item in a graph.
edge	An edge represents a connection between two vertices in a graph.
Table	1.1.1 Basic data structures.
Question set	1.1.1 Basic data structures.
Animation	1.1.2 A list avoids the shifting problem.
Question set	1.1.3 Basic data structures.

1.2 Introduction to algorithms

algorithm	An algorithm describes a sequence of steps to solve a computational problem or perform a calculation.
computational problem	A computational problem specifies an input, a question about the input that can be answered using a computer, and the desired output.
NP-complete	NP-complete problems are a set of problems for which no known efficient algorithm exists.
Animation	1.2.1 Computational problems and algorithms.
Table	1.2.1 Example computational problems and common algorithms.
Animation	1.2.2 Example NP-complete problem: Cliques.
Question set	1.2.3 Efficient algorithm and hard problems.
Question set	1.2.4 Computational problems and common algorithms.
Question set	1.2.5 Algorithms and computational problems.

1.3 Relation between data structures and algorithms

Figure	1.3.1 Algorithm to determine the top five salesperson using an array.
Question set	1.3.1 Top five salespersons.
Animation	1.3.2 A list avoids the shifting problem.
Question set	1.3.3 Algorithms for data structures.

1.4 Abstract data types

abstract data type	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.
ADT	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.
list	A list is an ADT for holding ordered data.
stack	A stack is an ADT in which items are only inserted on or removed from the top of a stack.
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.
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.
bag	A bag is an ADT for storing items in which the order does not matter and duplicate items are allowed.
set	A set is an ADT for a collection of distinct items.

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.
dictionary	A dictionary is an ADT that associates (or maps) keys with values.
Table	1.4.1 Common ADTs.
Animation	1.4.1 List ADT using array and linked lists data structures.
Question set	1.4.2 Abstract data types.
Question set	1.4.3 Common ADTs.

1.5 Applications of ADTs

Table	1.5.1 Standard libraries in various programming languages.
Question set	1.5.1 ADTs in standard libraries.
Animation	1.5.2 Programming using ADTs.
Question set	1.5.3 Programming with ADTs.

1.6 Algorithm efficiency

Algorithm efficiency	Algorithm efficiency is typically measured by the algorithm's computational complexity.
Computational complexity	Computational complexity is the amount of resources used by the algorithm.
runtime complexity	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.
best case	An algorithm's best case is the scenario where the algorithm does the minimum possible number of operations.

worst case	An algorithm's worst case is the scenario where the algorithm does the maximum possible number of operations.
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.
auxiliary space complexity	An algorithm's auxiliary space complexity is the space complexity not including the input data.

- **Animation** 1.6.1 Computational complexity.
- **Question set** 1.6.2 Algorithm efficiency and computational complexity.
- **Aside** Input data size must remain a variable
- **Animation** 1.6.3 Linear search best and worst cases.
- **Question set** 1.6.4 FindFirstLessThan algorithm best and worst case.
- **Question set** 1.6.5 Best and worst case concepts.
- **Animation** 1.6.6 FindMax space complexity and auxiliary space complexity.
- **Question set** 1.6.7 Space complexity of GetEvens function.

2. Lists, Stacks, and Queues

2.1 List abstract data type (ADT)

list	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.

- **Animation** 2.1.1 List ADT.

Question set	2.1.2 List ADT.
Table	2.1.1 Some common operations for a list ADT.
Question set	2.1.3 List ADT common operations.

2.2 Singly-linked lists

singly-linked list	A singly-linked list is a data structure for implementing a list ADT, where each node has data and a pointer to the next node.
head	A singly-linked list's first node is called the head, and the last node the tail.
tail	A singly-linked list's first node is called the head, and the last node the tail.
positional list	A singly-linked list is a type of positional list: A list where elements contain pointers to the next and/or previous elements in the list.
Append	Given a new node, the Append operation for a singly-linked list inserts the new node after the list's tail node.
Prepend	Given a new node, the Prepend operation for a singly-linked list inserts the new node before the list's head node.
null	Null is a special value indicating a pointer points to nothing.

Animation	2.2.1 Singly-linked list: Each node points to the next node.
Question set	2.2.2 Singly-linked list data structure.
Animation	2.2.3 Singly-linked list: Appending a node.
Question set	2.2.4 Appending a node to a singly-linked list.

Animation	2.2.5 Singly-linked list: Prepending a node.
Question set	2.2.6 Prepending a node in a singly-linked list.
Aside	null

2.3 Singly-linked lists: Insert

InsertAfter	Given a new node, the InsertAfter operation for a singly-linked list inserts the new node after a provided existing list node.
Animation	2.3.1 Singly-linked list: Insert nodes.
Question set	2.3.2 Inserting nodes in a singly-linked list.
Question set	2.3.3 Singly-linked list insert-after algorithm.

2.4 Singly-linked lists: Remove

RemoveAfter	Given a specified existing node in a singly-linked list, the RemoveAfter operation removes the node after the specified list node.
Animation	2.4.1 Singly-linked list: Node removal.
Question set	2.4.2 Removing nodes from a singly-linked list.
Question set	2.4.3 ListRemoveAfter algorithm execution: Intermediate node.
Question set	2.4.4 ListRemoveAfter algorithm execution: List head node.

2.5 Linked list search

search	Given a key, a search algorithm returns the first node whose data matches that key, or returns null if a matching node was not found.
Animation	2.5.1 Singly-linked list: Searching.

Question set	2.5.2 ListSearch algorithm execution.
Question set	2.5.3 Searching a linked-list.

2.6 Doubly-linked lists

doubly-linked list	A doubly-linked list is a data structure for implementing a list ADT, where each node has data, a pointer to the next node, and a pointer to the previous node.
positional list	A doubly-linked list is a type of positional list: A list where elements contain pointers to the next and/or previous elements in the list.
Append	Given a new node, the Append operation for a doubly-linked list inserts the new node after the list's tail node.
Prepend	Given a new node, the Prepend operation of a doubly-linked list inserts the new node before the list's head node and points the head pointer to the new node.
Question set	2.6.1 Doubly-linked list data structure.
Animation	2.6.2 Doubly-linked list: Appending a node.
Question set	2.6.3 Doubly-linked list data structure.
Animation	2.6.4 Doubly-linked list: Prepending a node.
Question set	2.6.5 Prepending a node in a doubly-linked list.

2.7 Doubly-linked lists: Insert

InsertAfter	Given a new node, the InsertAfter operation for a doubly-linked list inserts the new node after a provided existing list node.
Animation	2.7.1 Doubly-linked list: Inserting nodes.

Question set 2.7.2 Inserting nodes in a doubly-linked list.

2.8 Doubly-linked lists: Remove

Remove The Remove operation for a doubly-linked list removes a provided existing list node.

Animation 2.8.1 Doubly-linked list: Node removal.

Question set 2.8.2 Deleting nodes from a doubly-linked list.

Question set 2.8.3 ListRemove algorithm execution: Intermediate node.

Question set 2.8.4 ListRemove algorithm execution: List head node.

2.9 Linked list traversal

list traversal A list traversal algorithm visits all nodes in the list once and performs an operation on each node.

reverse traversal A reverse traversal visits all nodes starting with the list's tail node and ending after visiting the list's head node.

Figure 2.9.1 Linked list traversal algorithm.

Animation 2.9.1 Singly-linked list: List traversal.

Question set 2.9.2 List traversal.

Figure 2.9.2 Reverse traversal algorithm.

Question set 2.9.3 Reverse traversal algorithm execution.

2.10 Sorting linked lists

Animation 2.10.1 Sorting a doubly-linked list with insertion sort.

Question set 2.10.2 Insertion sort for doubly-linked lists.

Aside	Algorithm efficiency
Animation	2.10.3 Sorting a singly-linked list with insertion sort.
Figure	2.10.1 ListFindInsertionPosition algorithm.
Question set	2.10.4 Sorting singly-linked lists with insertion sort.
Aside	Algorithm efficiency
Table	2.10.1 Sorting algorithms easily adapted to efficiently sort linked lists.
Table	2.10.2 Sorting algorithms that cannot as efficiently sort linked lists.
Question set	2.10.5 Sorting linked-lists vs. sorting arrays.

2.11 Linked list dummy nodes

dummy node	A linked list implementation may use a dummy node (or header node): A node with an unused data member that always resides at the head of the list and cannot be removed.
header node	A linked list implementation may use a dummy node (or header node): A node with an unused data member that always resides at the head of the list and cannot be removed.
Figure	2.11.1 Singly-linked list with dummy node: append, prepend, insert after, and remove after operations.
Question set	2.11.1 Singly-linked list with dummy node.
Question set	2.11.2 Singly-linked list with dummy node.
Figure	2.11.2 Doubly-linked list with dummy node: append, prepend, insert after, and remove operations.

Question set	2.11.3 Doubly-linked list with dummy node.
Animation	2.11.4 Doubly-linked list append and prepend with 2 dummy nodes.
Figure	2.11.3 Doubly-linked list with 2 dummy nodes: insert after and remove operations.
Aside	Removing if statements from ListInsertAfter and ListRemove
Question set	2.11.5 Comparing a doubly-linked list with 1 dummy node vs. 2 dummy nodes.
Animation	2.11.6 Singly-linked lists with and without a dummy node.
Question set	2.11.7 Singly linked lists with a dummy node.
Question set	2.11.8 Condition for an empty list.

2.12 Stack abstract data type (ADT)

stack	A stack is an ADT in which items are only inserted on or removed from the top of a stack.
push	The stack push operation inserts an item on the top of the stack.
pop	The stack pop operation removes and returns the item at the top of the stack.
last-in first-out	A stack is referred to as a last-in first-out ADT.
Animation	2.12.1 Stack ADT.
Question set	2.12.2 Stack ADT: Push and pop operations.
Table	2.12.1 Common stack ADT operations.

Question set 2.12.3 Common stack ADT operations.

2.13 Stacks using linked lists

Animation 2.13.1 Stack implementation using a linked list.

Question set 2.13.2 Stack push and pop operations with a linked list.

2.14 Queue abstract data type (ADT)

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.

push The queue push operation inserts an item at the end of the queue.

pop The queue pop operation removes and returns the item at the front of the queue.

first-in first-out A queue is referred to as a first-in first-out ADT.

Animation 2.14.1 Queue ADT.

Question set 2.14.2 Queue ADT.

Table 2.14.1 Some common operations for a queue ADT.

Question set 2.14.3 Common queue ADT operations.

2.15 Queues using linked lists

Animation 2.15.1 Queue implemented using a linked list.

Question set 2.15.2 Queue push and pop operations with a linked list.

2.16 Deque abstract data type (ADT)

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.

peek	A peek operation returns an item in the deque without removing the item.
Animation	2.16.1 Deque ADT.
Question set	2.16.2 Deque ADT.
Table	2.16.1 Common deque ADT operations.
Question set	2.16.3 Common queue ADT operations.

2.17 Array-based lists

array-based list	An array-based list is a list ADT implemented using an array.
append	Given a new element, the append operation for an array-based list of length X inserts the new element at the end of the list, or at index X.
Prepend	The Prepend operation for an array-based list inserts the a new item at the start of the list.
InsertAfter	The InsertAfter operation for an array-based list inserts a new item after a specified index.
search	Given a key, the search operation returns the index for the first element whose data matches that key, or -1 if not found.
remove-at	Given the index of an item in an array-based list, the remove-at operation removes the item at that index.
Animation	2.17.1 Appending to array-based lists.
Animation	2.17.2 Array-based list resize operation.
Question set	2.17.3 Array-based lists.
Animation	2.17.4 Array-based list prepend and insert after operations.

3. Searching and Algorithm Analysis

3.1 Searching and algorithms

algorithm An algorithm is a sequence of steps for accomplishing a task.

Linear search Linear search is a search algorithm that starts from the beginning of a list, and checks each element until the search key is found or the end of the list is reached.

runtime An algorithm's runtime is the time the algorithm takes to execute.

3.2 Binary search

Binary search	Binary search is a faster algorithm for searching a list if the list's elements are sorted and directly accessible (such as an array).
Animation	3.2.1 Using binary search to search contacts on your phone.
Question set	3.2.2 Using binary search to search a contact list.
Animation	3.2.3 Binary search efficiently searches sorted list by reducing the search space by half each iteration.
Figure	3.2.1 Binary search algorithm.
Question set	3.2.4 Binary search algorithm execution.
Animation	3.2.5 Speed of linear search versus binary search to find a number within a sorted list.
Question set	3.2.6 Linear and binary search runtime.

3.3 Constant time operations

constant time operation	A constant time operation is an operation that, for a given processor, always operates in the same amount of time, regardless of input values.
Animation	3.3.1 Constant time vs. non-constant time operations.
Question set	3.3.2 Constant time operations.
Table	3.3.1 Common constant time operations.
Question set	3.3.3 Identifying constant time operations.

3.4 Growth of functions and complexity

Asymptotic notation	Asymptotic notation is the classification of runtime complexity that uses functions that indicate only the growth rate of a bounding function.

O notation	O notation provides a growth rate for an algorithm's upper bound.
Ω notation	Ω notation provides a growth rate for an algorithm's lower bound.
Θ notation	Θ notation provides a growth rate that is both an upper and lower bound.
Lower bound	Lower bound: A function f(N) that is ≤ the best case T(N), for all values of N ≥ 1.
Upper bound	Upper bound: A function f(N) that is ≥ the worst case T(N), for all values of N ≥ 1.
Table	3.4.1 Notations for algorithm complexity analysis.
Aside	Upper and lower bounds in the context of runtime complexity
Animation	3.4.1 Upper and lower bounds.
Question set	3.4.2 Upper and lower bounds.
Question set	3.4.3 Asymptotic notations.

3.5 O notation

Big O notation	Big O notation is a mathematical way of describing how a function (running time of an algorithm) generally behaves in relation to the input size.
Table	3.5.1 Growth rates for different input sizes.
Question set	3.5.1 Big O notation.
Figure	3.5.1 Rules for determining Big O notation of composite functions.

Question set	3.5.2 Big O notation for composite functions.
Learning tool	3.5.3 Computational complexity graphing tool.
Figure	3.5.2 Runtime complexities for various code examples.
Question set	3.5.4 Big O notation and growth rates.
Animation	3.5.5 Determining Big O notation of a function.

3.6 Algorithm analysis

worst-case runtime	The worst-case runtime of an algorithm is the runtime complexity for an input that results in the longest execution.
Question set	3.6.1 Worst-case runtime analysis.
Question set	3.6.2 Constant time operations.
Animation	3.6.3 Runtime analysis of nested loop: Selection sort algorithm.
Figure	3.6.1 Common summation: Summation of consecutive numbers.
Question set	3.6.4 Nested loops.
Animation	3.6.5 Runtime analysis: Finding the max value.
Animation	3.6.6 Simplified runtime analysis: A constant number of constant time operations is O(1).

3.7 Recursive definitions

recursive function	A recursive function is a function that calls itself.
algorithm	An algorithm is a sequence of steps, including at least 1 terminating step, for solving a problem.

recursive algorithm	A recursive algorithm is an algorithm that breaks the problem into smaller subproblems and applies the algorithm itself to solve the smaller subproblems.
base case	Base case: A case where a recursive algorithm completes without applying itself to a smaller subproblem.
Table	3.7.1 Sample recursive functions: Factorial, CumulativeSum, and ReverseList.
Question set	3.7.1 CumulativeSum recursive function.
Animation	3.7.2 Recursive factorial algorithm.
Question set	3.7.3 Recursive algorithms.

3.8 Recursive algorithms

Fibonacci sequence	The Fibonacci sequence is a numerical sequence where each term is the sum of the previous 2 terms in the sequence, except the first 2 terms, which are 0 and 1.
Fibonacci number	Fibonacci number: A term in the Fibonacci sequence.
Binary search	Binary search is an algorithm that searches a sorted list for a key by first comparing the key to the middle element in the list and recursively searching half of the remaining list so long as the key is not found.
Figure	3.8.1 FibonacciNumber recursive function.
Question set	3.8.1 FibonacciNumber recursive function.
Question set	3.8.2 Recursive binary search.
Question set	3.8.3 Recursive binary search base case.
Figure	3.8.2 BinarySearch recursive algorithm.

3.9 Analyzing the time complexity of recursive algorithms

recurrence relation Recurrence relation: A function f(N) that is defined in terms of the same function operating on a value < N.

recursion tree Recursion tree: A visual diagram of a operations done by a recursive function, that separates operations done directly by the function and operations done by recursive calls.

- Animation 3.9.1 Worst case binary search runtime complexity.
- Animation 3.9.2 Recursion trees.
- Question set 3.9.3 Recursion trees.
- Question set 3.9.4 Matching recursion trees with runtime complexities.
- Question set 3.9.5 Recursion trees.
- Question set 3.9.6 Binary search and recurrence relations.

4. Sorting Algorithms

4.1 Sorting: Introduction

Sorting Sorting is the process of converting a list of elements into ascending (or descending) order.

- Learning tool 4.1.1 Sort by swapping tool.
- Question set 4.1.2 Sorted elements.

4.2 Selection sort

Selection sort Selection sort is a sorting algorithm that treats the input as two parts, a sorted part and an unsorted part, and repeatedly selects the proper next value to move from the unsorted part to the end of the sorted part.

- Animation 4.2.1 Selection sort.

4.3 Insertion sort

Insertion sort Insertion sort is a sorting algorithm that treats the input as two parts, a sorted part and an unsorted part, and repeatedly inserts the next value from the unsorted part into the correct location in the sorted part.

nearly sorted A nearly sorted list only contains a few elements not in sorted order.

4.4 Shell sort

Shell sort Shell sort is a sorting algorithm that treats the input as a collection of interleaved lists, and sorts each list individually with a variant of the insertion sort algorithm.

gap value A gap value is a positive integer representing the distance between elements in an interleaved list.

Animation	4.4.1 Sorting interleaved lists with shell sort speeds up insertion sort.
Question set	4.4.2 Shell sort's interleaved lists.
Animation	4.4.3 Interleaved insertion sort.
Question set	4.4.4 Insertion sort variant.
Animation	4.4.5 Shell sort algorithm.
Question set	4.4.6 ShellSort.

4.5 Quicksort

Quicksort

Quicksort is a sorting algorithm that repeatedly partitions the input into low and high parts (each part unsorted), and then recursively sorts each of those parts.

pivot

The pivot can be any value within the array being sorted, commonly the value of the middle array element.

Animation	4.5.1 Quicksort partitions data into a low part with data less than/equal to a pivot value and a high part with data greater than/equal to a pivot value.
Question set	4.5.2 Quicksort pivot location and value.
Question set	4.5.3 Low and high partitions.
Animation	4.5.4 Quicksort.
Figure	4.5.1 Quicksort algorithm.
Learning tool	4.5.5 Quicksort tool.
Question set	4.5.6 Quicksort runtime.

Question set — 4.5.7 Worst case quicksort runtime.

4.6 Merge sort

Merge sort — Merge sort is a sorting algorithm that divides a list into two halves, recursively sorts each half, and then merges the sorted halves to produce a sorted list.

Animation — 4.6.1 Merge sort recursively divides the input into two halves, sorts each half, and merges the lists together.

Question set — 4.6.2 Merge sort partitioning.

Animation — 4.6.3 Merging partitions: Smallest element from left or right partition is added one at a time to a temporary merged list. Once merged, temporary list is copied back to the original list.

Question set — 4.6.4 Tracing merge operation.

Figure — 4.6.1 Merge sort algorithm.

Question set — 4.6.5 Merge sort runtime and memory complexity.

4.7 Radix sort

bucket — A bucket is a collection of integer values that all share a particular digit value.

Radix sort — Radix sort is a sorting algorithm specifically for an array of *integers*: The algorithm processes one digit at a time starting with the least significant digit and ending with the most significant.

Animation — 4.7.1 A particular single digit in an integer can determine the integer's bucket.

Question set — 4.7.2 Using the 1's digit, place each integer in the correct bucket.

Question set	4.7.3 Using the 10's digit, place each integer in the correct bucket.
Question set	4.7.4 Bucket concepts.
Question set	4.7.5 Assigning integers to buckets.
Animation	4.7.6 Radix sort algorithm (for non-negative integers).
Figure	4.7.1 RadixGetMaxLength and RadixGetLength functions.
Question set	4.7.7 Radix sort algorithm.
Question set	4.7.8 Radix sort algorithm analysis.
Figure	4.7.2 RadixSort algorithm (for negative and non-negative integers).
Question set	4.7.9 Sorting signed integers.
Aside	Radix sort with different bases

4.8 Overview of fast sorting algorithms

fast sorting algorithm	A fast sorting algorithm is a sorting algorithm that has an average runtime complexity of O($NlogN$) or better.
element comparison sorting algorithm	A element comparison sorting algorithm is a sorting algorithm that operates on an array of elements that can be compared to each other.
Table	4.8.1 Sorting algorithms' average runtime complexity.
Question set	4.8.1 Fast sorting algorithms.
Table	4.8.2 Identifying comparison sorting algorithms.
Question set	4.8.2 Comparison sorting algorithms.

Table	4.8.3 Fast sorting algorithm's best, average, and worst case runtime complexity.
Question set	4.8.3 Runtime complexity.

5. Hash Tables

5.1 Hash tables

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 (or vector).
key	In a hash table, an item's key is the value used to map to an index.
bucket	Each hash table array element is called a bucket.
hash function	A hash function computes a bucket index from the item's key.
modulo operator %	A common hash function uses the modulo operator %, which computes the integer remainder when dividing two numbers.
collision	A collision occurs when an item being inserted into a hash table maps to the same bucket as an existing item in the hash table.
Chaining	Chaining is a collision resolution technique where each bucket has a list of items (so bucket 5's list would become 55, 75).
Open addressing	Open addressing is a collision resolution technique where collisions are resolved by looking for an empty bucket elsewhere in the table (so 75 might be stored in bucket 6).
Animation	5.1.1 Hash table data structure.

Question set	5.1.2 Hash tables.
Aside	Item representation
Question set	5.1.3 Hash tables.
Question set	5.1.4 Hash table search efficiency.
Question set	5.1.5 Hash table collisions.
Aside	Empty cells

5.2 Chaining

Chaining

Chaining handles hash table collisions by using a list for each bucket, where each list may store multiple items that map to the same bucket.

Animation	5.2.1 Hash table with chaining.
Figure	5.2.1 Hash table with chaining: Each bucket contains a list of items.
Question set	5.2.2 Hash table with chaining: Inserting items.
Question set	5.2.3 Hash table with chaining: Search.

5.3 Linear probing

linear probing

A hash table with linear probing handles a collision by starting at the key's mapped bucket, and then linearly searches subsequent buckets until an empty bucket is found.

empty-since-start

An empty-since-start bucket has been empty since the hash table was created.

empty-after-removal

An empty-after-removal bucket had an item removed that caused the bucket to now be empty.

Animation	5.3.1 Hash table with linear probing.
Question set	5.3.2 Hash table with linear probing: Insert.
Question set	5.3.3 Hash with linear probing: Bucket status.
Animation	5.3.4 Insert with linear probing.
Question set	5.3.5 Hash table with linear probing: Insert with empty-after-removal buckets.
Animation	5.3.6 Remove with linear probing.
Question set	5.3.7 Hash table with linear probing: Remove.
Question set	5.3.8 Hash table with linear probing: Search.
Animation	5.3.9 Search with linear probing.

5.4 Quadratic probing

quadratic probing	A hash table with quadratic probing handles a collision by starting at the key's mapped bucket, and then quadratically searches subsequent buckets until an empty bucket is found.
probing sequence	Iterating through sequential i values to obtain the desired table index is called the probing sequence.
Animation	5.4.1 Hash table insertion using quadratic probing: c1 = 1 and c2 = 1.
Question set	5.4.2 Insertion using quadratic probing.
Animation	5.4.3 Search and removal with quadratic probing: c1 = 1 and c2 = 1.

Question set	5.4.4 Using empty buckets during search, insertion, and removal.
Figure	5.4.1 HashInsert with quadratic probing.
Figure	5.4.2 HashRemove and HashSearch with quadratic probing.
Question set	5.4.5 Hash table with quadratic probing: search and remove.

5.5 Double hashing

Double hashing

Double hashing is an open-addressing collision resolution technique that uses 2 different hash functions to compute bucket indices.

probing sequence

Iterating through sequential i values to obtain the desired table index is called the probing sequence.

Animation	5.5.1 Hash table insertion using double hashing.
Animation	5.5.2 Hash table insertion, search, and removal using double hashing.
Question set	5.5.3 Double hashing.
Question set	5.5.4 Hash table insertion, search, and removal with double hashing.
Question set	5.5.5 Hash table with double hashing: search, insert, and remove.

5.6 Common hash functions

perfect hash function

A perfect hash function maps items to buckets with no collisions.

modulo hash

A modulo hash uses the remainder from division of the key by hash table size N.

multiplicative string hash	A multiplicative string hash repeatedly multiplies the hash value and adds the ASCII (or Unicode) value of each character in the string.
mid-square hash	A mid-square hash squares the key, extracts R digits from the result's middle, and returns the remainder of the middle digits divided by hash table size N.
Figure	5.6.1 Modulo hash function.
Question set	5.6.1 Good hash functions and keys.
Question set	5.6.2 Decimal mid-square hash function.
Figure	5.6.2 Mid-square hash function (base 2).
Question set	5.6.3 Binary mid-square hash function.
Figure	5.6.3 Multiplicative string hash function.
Question set	5.6.4 Multiplicative string hash function.

5.7 Direct hashing

direct hash function	A direct hash function uses the item's key as the bucket index.
direct access table	A hash table with a direct hash function is called a direct access table.
search	Given a key, a direct access table search algorithm returns the item at index key if the bucket is not empty, and returns null (indicating item not found) if empty.
Animation	5.7.1 Direct hash function.
Figure	5.7.1 Direct hashing: Insert, remove, and search operations use item's key as bucket index.

Question set	5.7.2 Direct access table search, insert, and remove.
Question set	5.7.3 Direct hashing limitations.

5.8 Hashing Algorithms: Cryptography, Password Hashing

Cryptography	Cryptography is a field of study focused on transmitting data securely.
encryption	Encryption: alteration of data to hide the original meaning.
decryption	Decryption: reconstruction of original data from encrypted data.
cryptographic hash function	A cryptographic hash function is a hash function designed specifically for cryptography.
password hashing function	A password hashing function is a cryptographic hashing function that produces a hash value for a password.
Animation	5.8.1 Basic encryption: Caeser cipher.
Question set	5.8.2 Caesar cipher.
Question set	5.8.3 Cryptography.
Animation	5.8.4 A hash value can help identify corrupted data downloaded from the internet.
Animation	5.8.5 Password hashing function.
Question set	5.8.6 Hashing functions for data.
Question set	5.8.7 Password hashing function.

6. Trees

6.1 Binary trees

binary tree	In a binary tree, each node has up to two children, known as a *left child* and a *right child*.
Leaf	Leaf: A tree node with no children.
Internal node	Internal node: A node with at least one child.
Parent	Parent: A node with a child is said to be that child's parent.
ancestors	A node's ancestors include the node's parent, the parent's parent, etc., up to the tree's root.
Root	Root: The one tree node with no parent (the "top" node).
edge	The link from a node to a child is called an edge.
depth	A node's depth is the number of edges on the path from the root to the node.
level	All nodes with the same depth form a tree level.
height	A tree's height is the largest depth of any node.
full	A binary tree is full if every node contains 0 or 2 children.
complete	A binary tree is complete if all levels, except possibly the last level, are completely full and all nodes in the last level are as far left as possible.
perfect	A binary tree is perfect, if all internal nodes have 2 children and all leaf nodes are at the same level.
Animation	6.1.1 Binary tree basics.
Question set	6.1.2 Binary tree basics.
Animation	6.1.3 Binary tree terminology: height, depth, and level.

6.2 Applications of trees

Binary space partitioning

Binary space partitioning (BSP) is a technique of repeatedly separating a region of space into 2 parts and cataloging objects contained within the regions.

BSP

Binary space partitioning (BSP) is a technique of repeatedly separating a region of space into 2 parts and cataloging objects contained within the regions.

BSP tree

A BSP tree is a binary tree used to store information for binary space partitioning.

6.3 Binary search trees

binary search tree

A binary search tree (BST), which has an ordering property that any node's left subtree keys ≤ the node's key, and the right subtree's keys ≥ the node's key. That property enables fast searching for an item.

search	To search nodes means to find a node with a desired key, if such a node exists.
successor	A BST node's successor is the node that comes after in the BST ordering, so in A B C, A's successor is B, and B's successor is C.
predecessor	A BST node's predecessor is the node that comes before in the BST ordering.
Figure	6.3.1 BST ordering property: For three nodes, left child is less-than-or-equal-to parent, parent is less-than-or-equal-to right child. For more nodes, all keys in subtrees must satisfy the property, for every node.
Animation	6.3.1 BST ordering properties.
Question set	6.3.2 Binary search tree: Basic ordering property.
Figure	6.3.2 Searching a BST.
Animation	6.3.3 A BST may yield faster searches than a list.
Question set	6.3.4 Searching a BST.
Table	6.3.1 Minimum binary tree heights for N nodes are equivalent to $\lfloor log_2 N \rfloor$.
Animation	6.3.5 Searching a perfect BST with N nodes requires only O($logN$) comparisons.
Question set	6.3.6 Searching BSTs with N nodes.
Figure	6.3.3 A BST defines an ordering among nodes.
Question set	6.3.7 Binary search tree: Defined ordering.

6.4 BST search algorithm

search Given a key, a search algorithm returns the first node found matching that key, or returns null if a matching node is not found.

- Animation — 6.4.1 BST search algorithm.
- Question set — 6.4.2 BST search algorithm.
- Question set — 6.4.3 BST search algorithm decisions.
- Question set — 6.4.4 Tracing a BST search.

6.5 BST insert algorithm

insert Given a new node, a BST insert operation inserts the new node in a proper location obeying the BST ordering property.

- Animation — 6.5.1 Binary search tree insertions.
- Question set — 6.5.2 BST insert algorithm.
- Question set — 6.5.3 BST insert algorithm decisions.
- Question set — 6.5.4 Tracing BST insertions.
- Aside — BST insert algorithm complexity

6.6 BST remove algorithm

remove Given a key, a BST remove operation removes the first-found matching node, restructuring the tree to preserve the BST ordering property.

- Animation — 6.6.1 BST remove: Removing a leaf, or an internal node with a single child.
- Animation — 6.6.2 BST remove: Removing internal node with two children.

Figure 6.6.1 BST remove algorithm.

Question set 6.6.3 BST remove algorithm.

Aside BST remove algorithm complexity

6.7 BST inorder traversal

tree traversal A tree traversal algorithm visits all nodes in the tree once and performs an operation on each node.

inorder traversal An inorder traversal visits all nodes in a BST from smallest to largest.

Figure 6.7.1 BST inorder traversal algorithm.

Animation 6.7.1 BST inorder print algorithm.

Question set 6.7.2 Inorder traversal of a BST.

6.8 BST height and insertion order

height A tree's height is the maximum edges from the root to any leaf.

Animation 6.8.1 Inserting in random order keeps tree height near the minimum. Inserting in sorted order yields the maximum.

Question set 6.8.2 BST height.

Animation 6.8.3 BSTGetHeight algorithm.

Question set 6.8.4 BSTGetHeight algorithm.

6.9 BST parent node pointers

Figure 6.9.1 BSTInsert algorithm for BSTs with nodes containing parent pointers.

Figure 6.9.2 BSTReplaceChild algorithm.

Figure 6.9.3 BSTRemoveKey and BSTRemoveNode algorithms for BSTs with nodes containing parent pointers.

Question set 6.9.1 BST parent node pointers.

6.10 BST: Recursion

Animation 6.10.1 BST recursive search algorithm.

Question set 6.10.2 BST recursive search algorithm.

Figure 6.10.1 BST get parent algorithm.

Question set 6.10.3 BST get parent algorithm.

Figure 6.10.2 Recursive BST insertion and removal.

Question set 6.10.4 Recursive BST insertion and removal.

7. Balanced Trees

7.1 AVL: A balanced tree

AVL tree An AVL tree is a BST with a height balance property and specific operations to rebalance the tree when a node is inserted or removed.

height balanced A BST is height balanced if for any node, the heights of the node's left and right subtrees differ by only 0 or 1.

balance factor A node's balance factor is the left subtree height minus the right subtree height.

Animation 7.1.1 An AVL tree is height balanced: For any node, left and right subtree heights differ by only 0 or 1.

Question set 7.1.2 AVL trees.

Animation	7.1.3 Storing height at each AVL node.
Question set	7.1.4 Storing height at each AVL node.
Question set	7.1.5 AVL tree height.
Figure	7.1.1 An AVL tree doesn't have to be a perfect BST.

7.2 AVL rotations

rotation — A rotation is a local rearrangement of a BST that maintains the BST ordering property while rebalancing the tree.

Animation	7.2.1 A simple right rotation in an AVL tree.
Question set	7.2.2 AVL rotate right: 3 nodes.
Animation	7.2.3 In a right rotate, B's former right child C becomes D's left child, to maintain the BST ordering property.
Question set	7.2.4 AVL rotate right: 4 nodes.
Question set	7.2.5 AVL rotate left.
Question set	7.2.6 Right rotation algorithm.
Figure	7.2.1 AVLTreeUpdateHeight, AVLTreeSetChild, AVLTreeReplaceChild, and AVLTreeGetBalance algorithms.
Question set	7.2.7 AVL tree utility algorithms.
Animation	7.2.8 Right rotation algorithm.
Figure	7.2.2 AVLTreeRebalance algorithm.
Question set	7.2.9 AVLTreeRebalance algorithm.

7.3 AVL insertions

Animation	7.3.1 After an insert, a rotation may rebalance the tree.
Animation	7.3.2 Sometimes a double rotation is necessary to rebalance.
Question set	7.3.3 Double rotate: Left-then-right.
Figure	7.3.1 Four imbalance cases and rotations (indicated by blue arrow) to rebalance.
Question set	7.3.4 Determine the imbalance case and appropriate rotations.
Example	7.3.1 AVL example: Phone book.
Question set	7.3.5 AVL balance.
Figure	7.3.2 AVLTreeInsert algorithm.
Question set	7.3.6 AVLTreeInsert algorithm.
Animation	7.3.7 Four AVL imbalance cases are possible after inserting a new node.
Aside	AVL insertion algorithm complexity

7.4 AVL removals

remove	Given a key, an AVL tree remove operation removes the first-found matching node, restructuring the tree to preserve all AVL tree requirements.
Figure	7.4.1 AVLTreeRebalance algorithm.
Figure	7.4.2 AVLTreeRemoveKey algorithm.
Figure	7.4.3 AVLTreeRemoveNode algorithm.

7.5 Red-black tree: A balanced tree

red-black tree A red-black tree is a BST with two node types, namely red and black, and supporting operations that ensure the tree is balanced when a node is inserted or removed.

7.6 Red-black tree: Rotations

Question set 7.6.6 Red-black tree rotations.

7.7 Red-black tree: Insertion

insert Given a new node, a red-black tree insert operation inserts the new node in the proper location such that all red-black tree requirements still hold after the insertion completes.

Figure 7.7.1 RBTreeInsert algorithm.

Animation 7.7.1 RBTreeBalance algorithm.

Figure 7.7.2 RBTreeGetGrandparent and RBTreeGetUncle utility functions.

Question set 7.7.2 Red-black tree: insertion.

Question set 7.7.3 RBTreeInsert algorithm.

7.8 Red-black tree: Removal

remove Given a key, a red-black tree remove operation removes the first-found matching node, restructuring the tree to preserve all red-black tree requirements.

Figure 7.8.1 RBTreeGetPredecessor utility function.

Figure 7.8.2 RBTreeRemove algorithm.

Figure 7.8.3 RBTreeRemoveNode algorithm.

Question set 7.8.1 Removal concepts.

Figure 7.8.4 RBTreeAreBothChildrenBlack algorithm.

Figure 7.8.5 RBTreeGetSibling algorithm.

Figure 7.8.6 RBTreePrepareForRemoval pseudocode.

8. Heaps and Treaps

8.1 Heaps

max-heap	A max-heap is a binary tree that maintains the simple property that a node's key is greater than or equal to the node's childrens' keys.
insert	An insert into a max-heap starts by inserting the node in the tree's last level, and then swapping the node with its parent until no max-heap property violation occurs.
percolating	The upward movement of a node in a max-heap is sometime called percolating.
remove	A remove from a max-heap is always a removal of the root, and is done by replacing the root with the last level's last node, and swapping that node with its greatest child until no max-heap property violation occurs.

min-heap A min-heap is similar to a max-heap, but a node's key is less than or equal to its children's keys.

Figure 8.1.1 Max-heap property: A node's key is greater than or equal to the node's childrens' keys.

Question set 8.1.1 Max-heap property.

Animation 8.1.2 Max-heap insert and remove operations.

Question set 8.1.3 Max-heap inserts and deletes.

Example 8.1.1 Web server cache.

Question set 8.1.4 Web server cache example.

8.2 Heaps using arrays

Animation 8.2.1 Max-heap stored using an array.

Question set 8.2.2 Heap storage.

Table 8.2.1 Parent and child indices for a heap.

Question set 8.2.3 Heap parent and child indices.

Figure 8.2.1 Max-heap percolate up algorithm.

Figure 8.2.2 Max-heap percolate down algorithm.

Question set 8.2.4 Percolate algorithm.

8.3 Heap sort

Heapsort Heapsort is a sorting algorithm that takes advantage of a max-heap's properties by repeatedly removing the max and building a sorted array in reverse order.

heapify	The heapify operation is used to turn an array into a heap.
Animation	8.3.1 Heapify operation.
Table	8.3.1 Max-heap largest internal node index.
Question set	8.3.2 Heapify operation.
Question set	8.3.3 Heapify operation - critical thinking.
Animation	8.3.4 Heapsort.
Question set	8.3.5 Heapsort.
Figure	8.3.1 Heap sort.
Question set	8.3.6 Heapsort algorithm.

8.4 Priority queue abstract data type (ADT)

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.
push	The priority queue push operation inserts an item such that the item is closer to the front than all items of lower priority, and closer to the end than all items of equal or higher priority.
pop	The priority queue pop operation removes and returns the item at the front of the queue, which has the highest priority.
peek	A peek operation returns the highest priority item, without removing the item from the front of the queue.
PushWithPriority	PushWithPriority: A push operation that includes an argument for the pushed item's priority.
Animation	8.4.1 Priority queue push and pop.

Question set	8.4.2 Priority queue push and pop.
Table	8.4.1 Common priority queue ADT operations.
Question set	8.4.3 Common priority queue ADT operations.
Animation	8.4.4 Priority queue PushWithPriority operation.
Question set	8.4.5 PushWithPriority operation.
Table	8.4.2 Implementing priority queues with heaps.
Question set	8.4.6 Implementing priority queues with heaps.

8.5 Treaps

treap	A treap uses a main key that maintains a binary search tree ordering property, and a secondary key generated randomly (often called "priority") during insertions that maintains a heap property.
search	A treap search is the same as a BST search using the main key, since the treap is a BST.
insert	A treap insert initially inserts a node as in a BST using the main key, then assigns a random priority to the node, and percolates the node up until the heap property is not violated.
delete	A treap delete can be done by setting the node's priority such that the node should be a leaf (-∞ for a max-heap), percolating the node down using rotations until the node is a leaf, and then removing the node.
Animation	8.5.1 Treap insert: First insert as a BST, then randomly assign a priority and use rotations to percolate node up to maintain heap.
Question set	8.5.2 Recognizing treaps.

Question set	8.5.3 Treap insert.
Question set	8.5.4 Treap insert.
Animation	8.5.5 Treap delete: Set priority such that node must become a leaf, then percolate down using rotations.
Question set	8.5.6 Treap delete algorithm.
Question set	8.5.7 Treaps.

9. Graphs

9.1 Graphs: Introduction

graph	A graph is a data structure for representing connections among items, and consists of vertices connected by edges.
vertex	A vertex (or node) represents an item in a graph.
edge	An edge represents a connection between two vertices in a graph.
adjacent	Two vertices are adjacent if connected by an edge.
path	A path is a sequence of edges leading from a source (starting) vertex to a destination (ending) vertex.
path length	The path length is the number of edges in the path.
distance	The distance between two vertices is the number of edges on the shortest path between those vertices.

Figure	9.1.1 Examples of connected items: Subway map, electrical power transmission, electrical circuit.
Animation	9.1.1 A graph represents connections among items, like among computers, or people.

Question set	9.1.2 Graph basics.
Animation	9.1.3 Graphs: adjacency, paths, and distance.
Question set	9.1.4 Graph properties.

9.2 Applications of graphs

Animation	9.2.1 Driving directions use graphs and shortest path algorithms to determine the best route from start to destination.
Animation	9.2.2 A graph of product relationships can be used to produce recommendations based on purchase history.
Question set	9.2.3 Using graphs for road navigation.
Question set	9.2.4 Using graphs for flight navigation.
Question set	9.2.5 Product recommendations.
Animation	9.2.6 Professional networks represented by graphs help people establish business connections.
Question set	9.2.7 Professional network.

9.3 Graph representations: Adjacency lists

adjacent	Two vertices are adjacent if connected by an edge.
adjacency list	In an adjacency list graph representation, each vertex has a list of adjacent vertices, each list item representing an edge.
sparse graph	A sparse graph has far fewer edges than the maximum possible.
Animation	9.3.1 Adjacency list graph representation.
Question set	9.3.2 Adjacency lists.

Question set	9.3.3 Adjacency lists: Bus routes.

9.4 Graph representations: Adjacency matrices

adjacent	Two vertices are adjacent if connected by an edge.
adjacency matrix	In an adjacency matrix graph representation, each vertex is assigned to a matrix row and column, and a matrix element is 1 if the corresponding two vertices have an edge or is 0 otherwise.
Animation	9.4.1 Adjacency matrix representation.
Question set	9.4.2 Adjacency matrix.
Question set	9.4.3 Adjacency matrix: Power grid map.

9.5 Graphs: Breadth-first search

graph traversal	An algorithm commonly must visit every vertex in a graph in some order, known as a graph traversal.
breadth-first search	A breadth-first search (BFS) is a traversal that visits a starting vertex, then all vertices of distance 1 from that vertex, then of distance 2, and so on, without revisiting a vertex.
discovered	When the BFS algorithm first encounters a vertex, that vertex is said to have been discovered.
frontier	In the BFS algorithm, the vertices in the queue are called the frontier, being vertices thus far discovered but not yet visited.
Animation	9.5.1 Breadth-first search.
Example	9.5.1 Social networking friend recommender using breadth-first search.
Question set	9.5.2 BFS: Friend recommender.

Example	9.5.2 Application of BFS: Find closest item in a peer-to-peer network.
Question set	9.5.3 BFS application: Peer-to-peer search.
Animation	9.5.4 BFS algorithm.
Question set	9.5.5 BFS algorithm.
Question set	9.5.6 Breadth-first search traversal.

9.6 Graphs: Depth-first search

graph traversal	An algorithm commonly must visit every vertex in a graph in some order, known as a graph traversal.
depth-first search	A depth-first search (DFS) is a traversal that visits a starting vertex, then visits every vertex along each path starting from that vertex to the path's end before backtracking.
Animation	9.6.1 Depth-first search traversal with starting vertex A.
Question set	9.6.2 Recursive DFS algorithm.
Animation	9.6.3 Depth-first search.
Question set	9.6.4 Depth-first search traversal.
Question set	9.6.5 DFS algorithm.
Figure	9.6.1 Recursive depth-first search.

9.7 Directed graphs

directed graph	A directed graph, or digraph, consists of vertices connected by directed edges.
digraph	A directed graph, or digraph, consists of vertices connected by directed edges.

directed edge	A directed edge is a connection between a starting vertex and a terminating vertex.
adjacent	In a directed graph, a vertex Y is adjacent to a vertex X, if there is an edge from X to Y.
path	A path is a sequence of directed edges leading from a source (starting) vertex to a destination (ending) vertex.
cycle	A cycle is path that starts and ends at the same vertex.
cyclic	A directed graph is cyclic if the graph contains a cycle, and acyclic if the graph does not contain a cycle.
acyclic	A directed graph is cyclic if the graph contains a cycle, and acyclic if the graph does not contain a cycle.
Question set	9.7.1 Directed graph basics.
Animation	9.7.2 Directed graph: Paths and cycles.
Question set	9.7.3 Directed graphs: Cyclic and acyclic.
Figure	9.7.1 Directed graph examples: Process flow diagram, airline routes, and college course prerequisites.
Animation	9.7.4 A directed graph represents connections among items, like links between web pages, or airline routes.
Example	9.7.1 Cycles in directed graphs: Kidney transplants.

9.8 Weighted graphs

path length	In a weighted graph, the path length is the sum of the edge weights in the path.
weighted graph	A weighted graph associates a weight with each edge.

weight	A graph edge's weight, or cost, represents some numerical value between vertex items, such as flight cost between airports, connection speed between computers, or travel time between cities.
cost	A graph edge's weight, or cost, represents some numerical value between vertex items, such as flight cost between airports, connection speed between computers, or travel time between cities.
cycle length	The cycle length is the sum of the edge weights in a cycle.
negative edge weight cycle	A negative edge weight cycle has a cycle length less than 0.
Animation	9.8.1 Path length is the sum of edge weights.
Animation	9.8.2 Weighted graphs associate weight with each edge.
Question set	9.8.3 Path length and shortest path.
Figure	9.8.1 A shortest path from A to D does not exist, because the cycle B, C can be repeatedly taken to further reduce the cycle length.
Question set	9.8.4 Weighted graphs.
Question set	9.8.5 Negative edge weight cycles.

9.9 Algorithm: Dijkstra's shortest path

Dijkstra's shortest path algorithm	Dijkstra's shortest path algorithm, created by Edsger Dijkstra, determines the shortest path from a start vertex to each vertex in a graph.
distance	A vertex's distance is the shortest path distance from the start vertex.

predecessor pointer	A vertex's predecessor pointer points to the previous vertex along the shortest path from the start vertex.
Question set	9.9.1 Dijkstra's shortest path traversal.
Animation	9.9.2 Determining the shortest path from Dijkstra's algorithm.
Question set	9.9.3 Shortest path based on vertex predecessor.
Animation	9.9.4 Dijkstra's algorithm may not find the shortest path for a graph with negative edge weights.
Question set	9.9.5 Dijkstra's shortest path algorithm: Supported graph types.
Aside	Algorithm efficiency
Animation	9.9.6 Dijkstra's algorithm finds the shortest path from a start vertex to each vertex in a graph.

9.10 Algorithm: Bellman-Ford's shortest path

Bellman-Ford shortest path algorithm	The Bellman-Ford shortest path algorithm, created by Richard Bellman and Lester Ford, Jr., determines the shortest path from a start vertex to each vertex in a graph.
distance	A vertex's distance is the shortest path distance from the start vertex.
predecessor pointer	A vertex's predecessor pointer points to the previous vertex along the shortest path from the start vertex.
Animation	9.10.1 The Bellman-Ford algorithm finds the shortest path from a source vertex to all other vertices in a graph.
Question set	9.10.2 Bellman-Ford shortest path traversal.
Question set	9.10.3 Bellman-Ford: Distance and predecessor values.

Animation	9.10.4 Bellman-Ford: Checking for negative edge weight cycles.
Figure	9.10.1 Bellman-Ford shortest path algorithm.
Question set	9.10.5 Bellman-Ford algorithm: Checking for negative edge weight cycles.
Aside	Algorithm Efficiency

9.11 Topological sort

topological sort	A topological sort of a directed, acyclic graph produces a list of the graph's vertices such that for every edge from a vertex X to a vertex Y, X comes before Y in the list.
Figure	9.11.1 GraphGetIncomingEdgeCount function.
Question set	9.11.1 Topological sort algorithm.
Animation	9.11.2 Topological sort algorithm.
Question set	9.11.3 Topological sort.
Question set	9.11.4 Topological sort matching.
Aside	Algorithm efficiency
Question set	9.11.5 Identifying valid topological sorts.
Animation	9.11.6 Topological sorting can be used to order course prerequisites.
Question set	9.11.7 Course prerequisites.
Question set	9.11.8 Topological sort algorithm.
Animation	9.11.9 Topological sort.

9.12 Minimum spanning tree

Kruskal's minimum spanning tree algorithm

Kruskal's minimum spanning tree algorithm determines subset of the graph's edges that connect all vertices in an undirected graph with the minimum sum of edge weights.

minimum spanning tree

A graph's minimum spanning tree is a subset of the graph's edges that connect all vertices in the graph together with the minimum sum of edge weights.

connected

A connected graph contains a path between every pair of vertices.

Animation — 9.12.1 Using the minimum spanning to minimize total length of power lines connecting cities.

Question set — 9.12.2 Minimum spanning tree.

Aside — Algorithm efficiency

Question set — 9.12.3 Minimum spanning tree - critical thinking.

Question set — 9.12.4 Minimum spanning tree algorithm.

Animation — 9.12.5 Minimum spanning tree algorithm.

9.13 All pairs shortest path

all pairs shortest path

An all pairs shortest path algorithm determines the shortest path between all possible pairs of vertices in a graph.

Floyd-Warshall all-pairs shortest path algorithm

The Floyd-Warshall all-pairs shortest path algorithm generates a |V| x |V| matrix of values representing the shortest path lengths between all vertex pairs in a graph.

negative cycle

A negative cycle is a cycle with edge weights that sum to a negative value.

Animation — 9.13.1 Shortest path lengths matrix.

Question set	9.13.2 Shortest path lengths matrix.
Animation	9.13.3 Floyd-Warshall algorithm.
Question set	9.13.4 Floyd-Warshall algorithm.
Animation	9.13.5 Path reconstruction.
Question set	9.13.6 Path reconstruction.
Aside	Algorithm efficiency
Figure	9.13.1 FloydWarshallReconstructPath algorithm.
Question set	9.13.7 Shortest path lengths matrix.

10. Algorithms

10.1 Heuristics

heuristic	Heuristic: A technique that willingly accepts a non-optimal or less accurate solution in order to improve execution speed.
heuristic algorithm	A heuristic algorithm is an algorithm that quickly determines a near optimal or approximate solution.
0-1 knapsack problem	0-1 knapsack problem: The knapsack problem with the quantity of each item limited to 1.
self-adjusting heuristic	A self-adjusting heuristic is an algorithm that modifies a data structure based on how that data structure is used.

Animation	10.1.1 Introduction to the knapsack problem.
Animation	10.1.2 Non-optimal, heuristic algorithm to solve the 0-1 knapsack.
Question set	10.1.3 Heuristics.

Question set	10.1.4 The knapsack problem.
Question set	10.1.5 Heuristic algorithm and the 0-1 knapsack problem.
Question set	10.1.6 Move-to-front self-adjusting heuristic.
Animation	10.1.7 Move-to-front self-adjusting heuristic.

10.2 Greedy algorithms

greedy algorithm	A greedy algorithm is an algorithm that, when presented with a list of options, chooses the option that is optimal at that point in time.
fractional knapsack problem	The fractional knapsack problem is the knapsack problem with the potential to take each item a fractional number of times, provided the fraction is in the range [0.0, 1.0].
activity selection problem	The activity selection problem is a problem where 1 or more activities are available, each with a start and finish time, and the goal is to build the largest possible set of activities without time conflicts.

Animation	10.2.1 MakeChange greedy algorithm.
Figure	10.2.1 FractionalKnapsack algorithm.
Animation	10.2.2 Activity selection problem algorithm.
Question set	10.2.3 ActivitySelection algorithm.
Question set	10.2.4 Fractional knapsack problem.
Question set	10.2.5 Greedy algorithms.

10.3 Dynamic programming

Dynamic programming

Dynamic programming is a problem solving technique that splits a problem into smaller subproblems, computes and stores solutions to subproblems in memory, and then uses the stored solutions to solve the larger problem.

longest common substring

The longest common substring algorithm takes 2 strings as input and determines the longest substring that exists in both strings.

- Animation — 10.3.1 Finding longest common substrings in DNA.
- Animation — 10.3.2 FibonacciNumber algorithm: Recursion vs. dynamic programming.
- Animation — 10.3.3 Longest common substring algorithm.
- Question set — 10.3.4 FibonacciNumber implementation.
- Question set — 10.3.5 Dynamic programming.
- Question set — 10.3.6 Longest common substring matrix.
- Question set — 10.3.7 Common substrings in DNA.
- Question set — 10.3.8 Longest common substrings - critical thinking.
- Aside — Longest common substring algorithm complexity
- Aside — Optimized longest common substring algorithm complexity

11. B-trees

11.1 B-trees

B-tree

A B-tree with order K is a tree where nodes can have up to K-1 keys and up to K children.

order	The order is the maximum number of children a node can have.
full	A 2-3-4 tree node containing exactly 3 keys is said to be full, and uses all keys and children.
2-node	A node with 1 key is called a 2-node.
3-node	A node with 2 keys is called a 3-node.
4-node	A node with 3 keys is called a 4-node.
Question set	11.1.1 B-tree properties.
Table	11.1.1 2-3-4 tree internal nodes.
Question set	11.1.2 2-3-4 tree properties.
Question set	11.1.3 2-3-4 tree nodes.
Animation	11.1.4 Order 3 B-trees.
Question set	11.1.5 Validity of order 3 B-trees.
Example	11.1.1 A valid order 5 B-tree.
Figure	11.1.1 2-3-4 child subtree labels.

11.2 2-3-4 tree search algorithm

search	Given a key, a search algorithm returns the first node found matching that key, or returns null if a matching node is not found.
Question set	11.2.1 2-3-4 tree search.
Table	11.2.1 2-3-4 tree child node to choose based on search key.

Animation	11.2.2 2-3-4 tree search algorithm.
Question set	11.2.3 2-3-4 tree search algorithm.

11.3 2-3-4 tree insert algorithm

insert	Given a new key, a 2-3-4 tree insert operation inserts the new key in the proper location such that all 2-3-4 tree properties are preserved.
preemptive split	The preemptive split insertion scheme always splits any full node encountered during insertion traversal.
split	An important operation during insertion is the split operation, which is done on every full node encountered during insertion traversal.

Animation	11.3.1 Split operation.
Question set	11.3.2 Split operation.
Table	11.3.1 2-3-4 tree non-full-leaf insertion cases.
Animation	11.3.3 B-tree insertion with preemptive split algorithm.
Question set	11.3.4 Preemptive split insertion.
Question set	11.3.5 Insertion of key into leaf node.
Animation	11.3.6 B-tree split operation.
Question set	11.3.7 B-tree split operation.
Figure	11.3.1 Inserting a key with children within a parent node.

11.4 2-3-4 tree rotations and fusion

rotation	A rotation is a rearrangement of keys between 3 nodes that maintains all 2-3-4 tree properties in the process.

right rotation	A right rotation on a node causes the node to lose one key and the node's right sibling to gain one key.
left rotation	A left rotation on a node causes the node to lose one key and the node's left sibling to gain one key.
fusion	A fusion is a combination of 3 keys: 2 from adjacent sibling nodes that have 1 key each, and a third from the parent of the siblings.

- Animation — 11.4.1 Left and right rotations.
- Question set — 11.4.2 2-3-4 tree rotations.
- Question set — 11.4.3 Rotation Algorithm.
- Figure — 11.4.1 BTreeRemoveKey pseudocode.
- Animation — 11.4.4 Left rotation pseudocode.
- Animation — 11.4.5 Root fusion.
- Question set — 11.4.6 Root fusion.
- Animation — 11.4.7 Non-root fusion.
- Question set — 11.4.8 Non-root fusion.
- Question set — 11.4.9 Utility functions for rotations.

11.5 2-3-4 tree removal

remove	Given a key, a 2-3-4 tree remove operation removes the first-found matching key, restructuring the tree to preserve all 2-3-4 tree rules.

preemptive merge	The preemptive merge removal scheme involves increasing the number of keys in all single-key, non-root nodes encountered during traversal.
merge	A B-Tree merge operates on a node with 1 key and increases the node's keys to 2 or 3 using either a rotation or fusion.

12. Sets

12.1 Set abstract data type

set	A set is a collection of distinct elements.
add	A set add operation adds an element to the set, provided an equal element doesn't already exist in the set.

key value	Key value: A primitive data value that serves as a unique identifier for the element.
remove	Given a key, a set remove operation removes the element with the specified key from the set.
search	Given a key, a set search operation returns the set element with the specified key, or null if no such element exists.
subset	A set X is a subset of set Y only if every element of X is also an element of Y.

- **Animation** 12.1.1 Set abstract data type.
- **Question set** 12.1.2 Set abstract data type.
- **Animation** 12.1.3 Element keys and removal.
- **Question set** 12.1.4 Element keys and removal.
- **Animation** 12.1.5 SetIsSubset algorithm.
- **Question set** 12.1.6 Searching and subsets.

12.2 Set operations

union	The union of sets X and Y, denoted as X ∪ Y, is a set that contains every element from X, every element from Y, and no additional elements.
intersection	The intersection of sets X and Y, denoted as X ∩ Y, is a set that contains every element that is in both X and Y, and no additional elements.
difference	The difference of sets X and Y, denoted as X \ Y, is a set that contains every element that is in X but not in Y, and no additional elements.

filter	A filter operation on set X produces a subset containing only elements from X that satisfy a particular condition.
filter predicate	The condition for filtering is commonly represented by a filter predicate: A function that takes an element as an argument and returns a Boolean value indicating whether or not that element will be in the filtered subset.
map	A map operation on set X produces a new set by applying some function F to each element.

Animation	12.2.1 Set union, intersection, and difference.
Question set	12.2.2 Union, intersection, and difference.
Animation	12.2.3 SetFilter and SetMap algorithms.
Question set	12.2.4 Using SetFilter with a set of strings.
Question set	12.2.5 Using SetMap with a set of numbers.
Question set	12.2.6 SetFilter and SetMap algorithm concepts.

12.3 Static and dynamic set operations

dynamic set	A dynamic set is a set that can change after being constructed.
static set	A static set is a set that doesn't change after being constructed.

Table	12.3.1 Static and dynamic set operations.
Question set	12.3.1 Static and dynamic set operations.
Question set	12.3.2 Choosing static or dynamic sets for real-world datasets.

13. Additional Material

13.1 Bubble sort

Bubble sort Bubble sort is a sorting algorithm that iterates through a list, comparing and swapping adjacent elements if the second element is less than the first element.

Figure 13.1.1 Bubble sort algorithm.

Question set 13.1.1 Bubble sort.

13.2 Quickselect

Quickselect Quickselect is an algorithm that selects the k^{th} smallest element in a list.

Figure 13.2.1 Quickselect algorithm.

Question set 13.2.1 Quickselect.

13.3 Bucket sort

Bucket sort Bucket sort is a numerical sorting algorithm that distributes numbers into buckets, sorts each bucket with an additional sorting algorithm, and then concatenates buckets together to build the sorted result.

bucket A bucket is a container for numerical values in a specific range.

Figure 13.3.1 Bucket sort algorithm.

Question set 13.3.1 Bucket sort.

Aside Bucket sort terminology

13.4 List data structure

list data structure

A list data structure is a data structure containing the list's head and tail, and may also include additional information, such as the list's size.

Animation — 13.4.1 Linked lists can be stored with or without a list data structure.

Question set — 13.4.2 Linked list data structure.

13.5 Circular lists

circular linked list

A circular linked list is a linked list where the tail node's next pointer points to the head of the list, instead of null.

Animation — 13.5.1 Circular list structure and traversal.

Question set — 13.5.2 Circular list concepts.