COMP3506 Algos and Datas Summary

Gabriel Field

17/08/2023 - END OF COURSE

Contents

1 Boilerplate

This document contains a summary of *data structures* (section 2) and their associated *algorithms* (section 3).

Each data structure gives its ADT and references to algorithms that can be used on it.

Each algorithm gives a pseudocode representation.

I denote data types `LikeThis`. Each data type symbol `D` represents both the type itself, and also the set of all objects of that type (this is abusive, I know). This lets me write $x \in$ `D` to mean "x is of type `D`" and `method(args)` $\to$ `D` to mean "`method()` returns type `D`". For example, `foo(`$x \in$ `X)` $\to$ `Y` is a method which takes a single argument x of data type `X` and returns objects of type `Y`.

The object `null` is a member of every data type.

I denote parameterised data types like Java does; i.e. `Like<This>` where `Like` is a type parameterised by the type `This`. For example, `Set<Node>` is the type of `Set`s of `Node`s.

Some common abbreviations:

- “amo.”: amortised
- “obv.”: obvious

The LaTeX source code for this file, along with the Java code I wrote to generate the macro `\dataprintalgos`, can be found at this GitHub repo. The Java code is terrible – I know – but it gets the job done.

If you want an example of what a `Tree` is good for, check out that repo ;)

Changelog:

(2023-09-16 20:00) Content up to the end of week 8 lectures is now summarised here. User is no longer a goat.

(2023-09-09 14:04) Content up to the end of week 7 lectures is now summarised here. Comments are now green.

(2023-09-02 10:48) Content up to the end of week 6 lectures is now summarised here. Type annotated some methods. Highlighted method names in data structures.

(2023-08-27 19:13) Content up to the end of week 5 lectures is now summarised here.

(2023-08-25 16:08) Content up to the end of week 4 lectures is now summarised here.

(2023-08-17 12:00) Started this project.

2 Data Structures

2.1 General Linear Structures

Definition 2.1.1 (General Linear Structure)
A data structure is a **general linear** structure iff it **extends** either of:

- `StaticSequence` (ADT 2.1.2)
- `DynamicSequence` (ADT 2.1.3)

Algorithms this data structure(s) may utilise:

- Sort:
  - Comparison sort:
    - Selection sort (algo 1)
    - Insertion sort (algo 2)
    - Merge sort (algo 3)
    - Quick sort (algo 4)
    - Heap Sort (algo 5)
  - Non-comparison sort:
    - Bucket sort (algo 6)
    - Lexicographic sort (algo 7)
      - Radix sort (algo 8)
      - Binary radix sort (algo 9)

ADT 2.1.2 (`StaticSequence`)
Associated classes: `StaticSequence` = `StaticSequence<Data>`, `Data`.

Stores an *ordered* sequence X of elements $x_0, \dots, x_{n-1}$, potentially with duplicates.

Method	Function
`build`(X)	Create new data structure to store X
`len`()	Return n
`get`($i \in \{0, \dots, n-1\}$)	Return x_i
`set`($i \in \{0, \dots, n-1\}$, x)	Set x_i to x

Algorithms this data structure(s) may utilise:

- Sort:
  - Comparison sort:
    - Selection sort (algo 1)
    - Insertion sort (algo 2)
    - Merge sort (algo 3)
    - Quick sort (algo 4)
    - Heap Sort (algo 5)
  - Non-comparison sort:
    - Bucket sort (algo 6)
    - Lexicographic sort (algo 7)
      - Radix sort (algo 8)
      - Binary radix sort (algo 9)

ADT 2.1.3 (`DynamicSequence`)
Associated classes: `DynamicSequence` = `DynamicSequence<Data>`, `Data`.

Stores an *ordered* sequence X of elements $x_0, \dots, x_{n-1}$, potentially with duplicates, where the number n of elements is allowed to change.

Method	Function
`build`(X)	Create new data structure to store X
`len`()	Return n

Method	Function
`get`($i \in \{0, \dots, n-1\}$)	Return x_i
`set`($i \in \{0, \dots, n-1\}$, x)	Set x_i to x
`add`(x)	Add x as a new element

Algorithms this data structure(s) may utilise:

- Sort:
  - Comparison sort:
    - Selection sort (algo 1)
    - Insertion sort (algo 2)
    - Merge sort (algo 3)
    - Quick sort (algo 4)
    - Heap Sort (algo 5)
  - Non-comparison sort:
    - Bucket sort (algo 6)
    - Lexicographic sort (algo 7)
      - Radix sort (algo 8)
      - Binary radix sort (algo 9)

2.1.1 Children of `StaticSequence`

ADT 2.1.4 (`Array` implements `StaticSequence` (ADT 2.1.2))
Associated classes: `Array` = `Array<Data>`, `Data`.

A static sequence stored in a *contiguous* chunk of memory. We store:

- `size`: n
- the array (duh)

Method	Function	Runtime complexity (worst)
Memory usage	store	$\Theta(n)$
`build`(X)	Create new data structure to store X	idk lol
`len`()	Return n	$\Theta(1)$
`get`($i \in \{0, \dots, n-1\}$)	Return x_i	$\Theta(1)$
`set`($i \in \{0, \dots, n-1\}$, x)	Set x_i to x	$\Theta(1)$
`iterate`(f `a function`)	Iterate through collection	$\mathcal{O}(n \cdot T_f(s))$ for T_f the runtime of f and s the size of elements

Algorithms this data structure(s) may utilise:

- Sort:
  - Comparison sort:
    - Selection sort (algo 1)
    - Insertion sort (algo 2)
    - Merge sort (algo 3)
    - Quick sort (algo 4)
    - Heap Sort (algo 5)
  - Non-comparison sort:
    - Bucket sort (algo 6)
    - Lexicographic sort (algo 7)
      - Radix sort (algo 8)
      - Binary radix sort (algo 9)

2.1.2 Children of `DynamicSequence`

ADT 2.1.5 (`LinkedList` implements `DynamicSequence` (ADT 2.1.3))
Associated classes: `LinkedList` = `LinkedList<Data>`, `Data`

A linear arrangement of (at least singly) linked nodes. We store:

- `size`: n
- `head`: reference to first node in the list
- `tail`: reference to last node in the list *Note: only exists sometimes*

Method	Function	Runtime complexity (worst)
Memory usage	store	$\Theta(n)$
`build`(X)	Create new data structure to store X	$\Theta(n)$
`len()`	Return n	$\Theta(1)$
`get`($i \in \{0, \dots, n-1\}$)	Return x_i	$\Theta(n)$ (cf. `ExtensibleList`)
`set`($i \in \{0, \dots, n-1\}$, x)	Set x_i to x	$\Theta(n)$ (cf. `ExtensibleList`)
`iterate`(f `a function`)	Iterate through collection	$\mathcal{O}(n \cdot T_f(s))$ for T_f the runtime of f and s the size of elements
`add`(x)	Add x to list (at either the `head` or the `tail`)	$\Theta(1)$
`insert`($i \in \{0, \dots, n-1\}$, x)	Insert x immediately before element i	$\Theta(n)$

This data structure has the following variants:

- `DoublyLinkedList` extends `LinkedList`
- `CircularlyLinkedList` extends `LinkedList`

Algorithms this data structure(s) may utilise:

- Sort:
  - Comparison sort:
    - Selection sort (algo 1)
    - Insertion sort (algo 2)
    - Merge sort (algo 3)
    - Quick sort (algo 4)
    - Heap Sort (algo 5)
  - Non-comparison sort:
    - Bucket sort (algo 6)
    - Lexicographic sort (algo 7)
      - Radix sort (algo 8)
      - Binary radix sort (algo 9)

ADT 2.1.6 (`ExtensibleList` implements `DynamicSequence` (ADT 2.1.3))
Associated classes: `ExtensibleList` = `ExtensibleList<Data>`, `Data`.

An array-based implementation of `DynamicSequence` where the array is resized if need be. We store:

- `size`: logical size n
- `capacity`: (current) length of the internal array
- the array (duh)

Method	Function	Runtime complexity (worst)
Memory usage	store	$\Theta(n)$
`build`(X)	Create new data structure to store X	$\Theta(n)$
`len()`	Return n	$\Theta(1)$
`get`($i \in \{0, \dots, n-1\}$)	Return x_i	$\Theta(1)$ (cf. `LinkedList`)
`set`($i \in \{0, \dots, n-1\}$, x)	Set x_i to x	$\Theta(1)$ (cf. `LinkedList`)

Method	Function	Runtime complexity (worst)
`iterate(`f `a function)`	Iterate through collection	$\mathcal{O}(n \cdot T_f(s))$ for T_f the runtime of f and s the size of elements
`append(`x`)`	Add x to the tail of the list	*amortised* $\Theta(1)$; raw $\mathcal{O}(n)$
`insert(`$i \in \{0, \dots, n-1\}$`,` x`)`	Insert x immediately before element i	$\Theta(n)$

We assume that the internal array is resized according to a *constant multiple* scheme; i.e. we have a fixed number $r \in \mathbb{Z}_{>1}$ such that each resize has `capacity` $\leftarrow r \cdot n$.

Algorithms this data structure(s) may utilise:

- Sort:
  - Comparison sort:
    * Selection sort (algo 1)
    * Insertion sort (algo 2)
    * Merge sort (algo 3)
    * Quick sort (algo 4)
    * Heap Sort (algo 5)
  - Non-comparison sort:
    * Bucket sort (algo 6)
    * Lexicographic sort (algo 7)
      · Radix sort (algo 8)
      · Binary radix sort (algo 9)

Apparently `PositionalList` exists, too. Is it important? idk.

2.2 Stacks and Queues

ADT 2.2.1 (`Stack`)
Associated classes: `Stack` = `Stack<Data>`, `Data`.

A dynamic-size FILO data structure storing n elements. `Stack` stores

- `size`: n
- `top`: pointer to the top of the stack (maybe the index of the top element, in an array-based implementation)

The runtime complexity in the following table depends on the implementation. I've listed the complexity for an 'ideal' implementation below.

Method	Function	Runtime complexity (worst)
Memory usage	store	$\Theta(n)$
`build(`X`)`	Create new data structure to store X	depends on implementation
`push(`$x \in$ `Data)` $\to$ `void`	Push x onto the stack	$\Theta(1)$ (perhaps amortised from raw $\mathcal{O}(n)$, depending on implementation)
`pop()` $\to$ `Data`	Return and remove the element at the `top`	$\Theta(1)$
`peek()` aka `top()` $\to$ `Data`	Return the element at the `top`	$\Theta(1)$
`isEmpty()` $\to$ `boolean`	Return `true` iff $n \neq 0$	$\Theta(1)$

Implementation strategies:

- `LinkedList`
- `ExtensibleList` (amortised $\Theta(1)$-time `push()` operation, with raw $\mathcal{O}(n)$ complexity)
- (static-size stack) `Array`

Algorithms this data structure(s) may utilise: (none yet)

ADT 2.2.2 (`Queue`)
Associated classes: `Queue` = `Queue<Data>`, `Data`.

A dynamic-size FIFO data structure storing n elements. `Queue` stores

- `size`: n
- `front`: pointer to the front of the queue (maybe an index in an array)
- `back`: pointer to the back of the queue (maybe an index in an array)

The runtime complexity in the following table depends on the implementation. I've listed the complexity for an 'ideal' implementation below.

Method	Function	Runtime complexity (worst)
Memory usage	store	$\Theta(n)$
`build(`X`)`	Create new data structure to store X	depends on implementation
`enqueue(`x`)` $\to$ `void`	Enqueue x onto the `back` of the queue	$\Theta(1)$ (perhaps amortised from raw $\mathcal{O}(n)$, depending on implementation)
`dequeue()` $\to$ `Data`	Return and remove the element at the `front`	$\Theta(1)$
`first()` $\to$ `Data`	Return the element at the `front`	$\Theta(1)$
`isEmpty()` $\to$ `boolean`	Return `true` iff $n \neq 0$	$\Theta(1)$

Implementation strategies:

- `LinkedList`
- `ExtensibleList` (amortised $\Theta(1)$-time `enqueue()` operation, with raw $\mathcal{O}(n)$ complexity)
- (static-size queue) `Array` (circular arrangement)

Algorithms this data structure(s) may utilise: (none yet)

2.2.1 Priority Queues

ADT 2.2.3 (`PriorityQueue` extends `Queue` (ADT 2.2.2))
Associated classes: `PriorityQueue` = `PQ<Key, Value>` (shorthand: `PQ`), `Entry` = `Entry<Key, Value>`, `Key`, `Value`.

A dynamic-size structure storing n $(\text{key}, \text{value})$ pairs. Entries with lower keys are retrieved before entries with higher keys.

`PriorityQueue` stores

- `size`: n

`Entry` stores

- `key` $\in$ `Key`
- `value` $\in$ `Value`

The runtime complexity in the following table depends on the implementation. I've listed the complexity for an 'ideal' implementation below.

Method	Function	Runtime complexity (worst)
Memory usage	store	$\Theta(n)$
`PQ.build(`X`)`	Create new data structure to store X	$\Theta(n)$, but depends on implementation
`PQ.insert(`$k \in$ `Key`, $v \in$ `Value)`	insert a new entry storing (k, v)	depends on implementation*
`PQ.removeMin()` $\to$ `Entry`	Return and remove the `Entry` with smallest `key`, and at the front amongst entries with the same key	depends on implementation*

Method	Function	Runtime complexity (worst)
`PQ.min()` $\rightarrow$ `Entry`	Return the `Entry` with smallest `key`, and at the front amongst entries with the same key	depends on implementation*
`PQ.size()`	Return `size`	$\Theta(1)$
`PQ.isEmpty()`	Return `true` iff $n \neq 0$	$\Theta(1)$
`Entry.getKey()`	Return this `Entry`'s key	$\Theta(1)$
`Entry.getValue()`	Return this `Entry`'s value	$\Theta(1)$

*see table 8 for a comparison of runtime depending on implementation.

Implementation strategies:

- unsorted `LinkedList`
- sorted `ExtensibleList` (or `Array` for static-sized PQ)
- `Heap`

Runtime comparison (depending on implementation):

Method	unsorted `LinkedList`	sorted `ExtensibleList`	`Heap`
`PQ.insert()`	$\Theta(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log(n))$ (amo.)
`PQ.removeMin()`	$\mathcal{O}(n)$	$\Theta(1)$	$\mathcal{O}(\log(n))$
`PQ.min()`	$\mathcal{O}(n)$	$\Theta(1)$	$\Theta(1)$

Table 8: Comparison of runtime based on implementation

Algorithms this data structure(s) may utilise: (none yet)

ADT 2.2.4 (`AdaptablePriorityQueue` extends `PriorityQueue` (ADT 2.2.3))

Associated classes:

- `AdaptablePriorityQueue` = `APQ<Key, Value>` (shorthand: `APQ`),
- `Position` = `Position<Key, Value>`,
- `Entry` = `Entry<Key, Value>`,
- `Key`, `Value`.

A dynamic-size structure storing n (key, value) pairs, which may be removed or edited at will. Entries with lower keys are retrieved before entries with higher keys.

`AdaptablePriorityQueue` stores

- `size`: n

`Position` stores

- `entry` $\in$ `Entry`: entry at this position

`Entry` stores

- `key` $\in$ `Key`: key for this entry
- `value` $\in$ `Value`: value for this entry
- `position` $\in$ `Position`: position of this entry

The runtime complexity in the following table depends on the implementation. I've listed the complexity for an 'ideal' implementation below.

Method	Function	Runtime complexity (worst)
Memory usage	store	$\Theta(n)$
(inherit)	(from `PriorityQueue`)	(ADT 2.2.3)
`APQ.remove(`$e \in$ `Entry)`	Remove and return e, if it is present	depends on implementation*
`APQ.replaceKey(`$e \in$ `Entry,` $k \in$ `Key)`	Replace key of e, and return the old key of e	depends on implementation*

Method	Function	Runtime complexity (worst)
APQ.replaceValue($e \in$ Entry, $v \in$ Value)	Replace value of e, and return the old value of e	depends on implementation*

*see table 10 for a comparison of runtime depending on implementation.
Implementation strategies:
- unsorted LinkedList
- sorted ExtensibleList (or Array for static-sized PQ)
- Heap

Runtime comparison (depending on implementation):

Method	unsorted LinkedList	sorted ExtensibleList	Heap
APQ.remove()	$\Theta(1)$	$\Theta(1)$	$\mathcal{O}(\log(n))$
APQ.replaceKey()	$\Theta(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log(n))$
APQ.replaceValue()	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Table 10: Comparison of runtime based on implementation. See also table 8.

Algorithms this data structure(s) may utilise: (none yet)

2.3 Trees

ADT 2.3.1 (Tree)
Associated classes: Tree = Tree<Data>, Node = Node<Data>, Data.

A dynamic-size hierarchical structure of n nodes (with arbitrarily many children). The tree stores:
- size $\in \mathbb{Z}_{\geq 0}$: n
- height $\in \mathbb{Z}_{\geq 0}$: height of the tree
- root $\in$ Node: pointer to the root of the tree (maybe an index in an array)

The nodes (of type Node) store:
- parent $\in$ Node: pointer to the parent of this Node
- data: data stored at this node
- children $\in$ Set<Node>: set of children

The runtime complexity in the following table depends on the implementation. I've listed the complexity for an 'ideal' implementation below. Here, n is the number of nodes, and h is the height.

Method	Function	Runtime complexity (worst)
Memory usage (all)	store	$\Theta(n)$
Tree.build(X)	Create new data structure to store X	depends on implementation
Tree.size()	Return n	$\Theta(1)$
Tree.isEmpty()	Return true iff $n = 0$	$\Theta(1)$
Tree.root()	Return root	$\Theta(1)$
Tree.iterator()	Return an iterator for this tree	depends on implementation
Tree.positions()	Not in Joel's headcanon	depends on implementation
Node.parent()	Return this.parent	$\Theta(1)$
Node.children()	Return this.children	$\Theta(1)$
Node.numChildren()	Return this.children.size()	$\Theta(1)$
Node.isInternal()	Return true iff this node is internal; i.e. it has children	$\Theta(1)$

Method	Function	Runtime complexity (worst)
`Node.isExternal()`	Return `true` iff this node is external; i.e. it is a leaf	$\Theta(1)$
`Node.isRoot()`	Return `true` iff this node is the root of a tree; i.e. `this.parent` = `null`	$\Theta(1)$

In addition, a **concrete data type** implementing `Tree` may support the following methods.

Method	Function	Runtime complexity (worst)
`Tree.replace(`$x \in$ `Node`, $y \in$ `Node)`	Replace x with y	$\Theta(1)$
`Tree.addRoot(`$x \in$ `Node)`	Set the root of this `Tree` to x, and the old root to one of x's children	$\Theta(1)$
`Tree.remove(`$x \in$ `Node)`	Remove x from this tree	$\Theta(1)$

A `Tree` is k-*ary* iff each node has at most $k \in \mathbb{Z}_{>0}$ children.
Algorithms this data structure(s) may utilise: (none yet)

ADT 2.3.2 (`BinTree` extends `Tree` (ADT 2.3.1))
Associated classes: `BinTree` = `BinTree<Data>`, `Node` = `Node<Data>`, `Data`.
A 2-ary tree. The tree stores the same as in `Tree`. The nodes store:

- `parent` $\in$ `Node`: pointer to the parent of this `Node`
- `data`: data stored at this node
- `left` $\in$ `Node`: left child
- `right` $\in$ `Node`: right child

The runtime complexity in the following table depends on the implementation. I've listed the complexity for an 'ideal' implementation below. These methods are *in addition* to those supported by `Tree`.

Method	Function	Runtime complexity (worst)
(inherit)	(from `Tree`)	(ADT 2.3.1)
`Node.left()`	Return `this.left`	$\Theta(1)$
`Node.right()`	Return `this.right`	$\Theta(1)$

Definition (full level): Level l of a binary tree is *full* iff it contains 2^l non-`null` nodes.
Definition (complete tree): $T \in$ `BinTree` is *complete* iff every level except the bottom level is full, and all leaves are as leftmost as possible.
Definition (proper tree, full tree): $T \in$ `BinTree` is *proper* (aka *full*) iff every non-leaf node of T has exactly two children.
Algorithms this data structure(s) may utilise: (none yet)

ADT 2.3.3 (`Heap` extends `BinTree` (ADT 2.3.2))
Associated classes: `Heap` = `Heap<Data>`, `Node` = `Node<Data>`, `Data`.
We describe a min-heap here. A max-heap is similar, but the internal sorting is the opposite.
A dynamic-size structure which stores *totally ordered* elements.
Class invariants: $H \in$ `BinTree` is a (min-)*heap* iff

- *(Heap-order)* For all nodes n in H such that $n \neq H$`.root`, n`.data` $\geq n$`.parent.data` (in a heap implementing a priority queue, `.data` means `.key`), and
- *(Shape)* H is a complete binary tree.

`Heap` stores

- (inherit from `BinTree`)
- `last` $\in$ `Node`: rightmost node of maximum depth

The runtime complexity in the following table depends on the implementation. I've listed the complexity for an 'ideal' implementation below. These methods are *in addition* to those supported by `Tree`.

Method	Function	Runtime complexity (worst)
(inherit)	(from `BinTree`)	(ADT 2.3.2)
`Heap.getLast()`	Return `this.last`	$\Theta(1)$
`Heap.insert(`$x \in$ `Data)`	Store new `Node` with data x in this `Heap`	$\mathcal{O}(\log(n))$

A heap is essentially an *auto-sorting* data structure.

Implementation via arrays: Store a heap H of size n in an array A (actually, an extensible list) of size n according to the following rules:

- H`.root` is stored at index 0
- For any node `node` stored at index i,
  - `node.left` is stored at index $2i + 1$
  - `node.right` is stored at index $2i + 2$
- (It may be helpful to store references to location-aware `Node` objects in the array, rather than just the data itself)

Note that:

- H`.last` is stored at index $n - 1$
- The next node to insert into will go at index n

This encoding is an injection ${}^{\texttt{Heap}}/_{\simeq} \hookrightarrow {}^{\texttt{Array}}/_{\simeq}$ of heaps (up to isomorphism) into arrays (up to isomorphism).

Lemma. *The height of a heap of size n is $\mathcal{O}(\log(n))$.*

Algorithms this data structure(s) may utilise:

- Heap Methods:
  - Upheap (algo 10)
  - Downheap (algo 11)
  - Heap.build() (algo 12)
  - Heap.insert() (algo 13)
  - Heap.removeMin() (algo 14)

2.3.1 Search Trees

Data Structure 2.3.4 (`BinSearchTree` extends `BinTree` (ADT 2.3.2))

Associated classes: `BinSearchTree` = `BinSearchTree<Key, Value>`, `Node` = `Node<Key, Value>`, `Key`, `Value`.

Binary search trees are a structure for storing sorted data in an optimised manner.

Definition. $T \in$ `BinTree` is a *binary search tree* iff

- `Key` is totally ordered (by some comparator $\leq$), and,
- T is a proper binary tree, and,
- All leaves in T store no data (could be regarded as null pointers), and,
- All non-leaves p in T (with left child l and right child r) have the property that $\text{key}(l) \leq \text{key}(p) \leq \text{key}(r)$.

`BinSearchTree` stores:

- `parent` $\in$ `Node`: pointer to the parent of this `Node`
- `key` $\in$ `Key`: key stored at this node
- `value` $\in$ `Value`: value stored at this node
- `left` $\in$ `Node`: left child
- `right` $\in$ `Node`: right child

The runtime complexity in the following table depends on the implementation. I've listed the complexity for an 'ideal' implementation below. These methods are *in addition* to those supported by `BinTree`.

Method	Function	Runtime complexity (worst)
(inherit)	(from `BinTree`)	(ADT 2.3.2)
`Node.key()`	Return `this.key`	$\Theta(1)$
`Node.value()`	Return `this.value`	$\Theta(1)$

Binary search trees are good for storing sparse data, while maintaining a 'binary search'-style algorithm.

Note. The standard binary search tree methods are all have $\mathcal{O}(\text{height of tree})$ runtime complexity.

Algorithms this data structure(s) may utilise:

- Search:
  - Binary search (algo 15)
- Binary Search Tree Methods:
  - BinSearchTree.remove() (algo 16)

Data Structure 2.3.5 (`AVLTree` extends `BinSearchTree` (data structure 2.3.4))
Associated classes: `AVLTree` = `AVLTree<Key, Value>`, `Node` = `Node<Key, Value>`, `Key`, `Value`.

AVL trees are a kind of self-balancing binary tree. They guarantee logarithmic time operations due to their self-balancing nature.

Definition. $T \in$ `BinSearchTree` is an *AVL tree* iff

- All nodes p with left-child l and right-child r in T have $|\text{height}(l) - \text{height}(r)| \leq 1$.

`AVLTree` stores the same as `BinSearchTree`. `Node` stores, in addition to what it stored in `BinSearchTree`,

- `height` $\in \mathbb{Z}_{\geq 0}$: height of this node

The runtime complexity in the following table depends on the implementation. I've listed the complexity for an 'ideal' implementation below. These methods are *in addition* to those supported by `BinSearchTree`.

Method	Function	Runtime complexity (worst)
(inherit)	(from `BinSearchTree`)	(ADT 2.3.4)
`AVLTree.rebalance(newNode` $\in$ `Node)`	Rebalance this tree, seeking up from newNode.	$\mathcal{O}(\text{depth of newNode})$
`AVLTree.triNodeRestructuring(unbalanced` $\in$ `Node)`	Restructure an unbalanced node	$\Theta(1)$

AVL trees are important because of the following lemma, which ensures that binary search tree operations in an AVL tree are logarithmic time.

Lemma. For any AVL tree T, $\text{height}(T) \in \mathcal{O}(\log(\text{number of nodes in } T))$.

Note. I've defined AVL trees as an extension of binary serach trees. However, they can be generalised to binary trees in general. Our course has seemed to only focus on their application to binary search trees.

Algorithms this data structure(s) may utilise:

- Search:
  - Binary search (algo 15)
- Binary Search Tree Methods:
  - BinSearchTree.remove() (algo 16)
- AVL Tree Methods:
  - AVLTree.insert(), AVLTree.remove() (algo 17)
  - AVLTree.rebalance(), AVLTree.triNodeRestructuring() (algo 18)

Data Structure 2.3.6 (`SplayTree` extends `BinSearchTree` (data structure 2.3.4))
Associated classes: `SplayTree` = `SplayTree<Key, Value>`, `Node` = `Node<Key, Value>`, `Key`, `Value`.

Splay trees are a kind of self-restructuring binary search tree. Whereas AVL trees prioritise fast operations on *all* nodes, splay trees prioritise fast oeprations on *recently accessed* nodes. This makes splay trees empirically faster for data that is searched frequently, at the cost of potentially linear time operations on data that is requested infrequently.

`SplayTree` stores the same as `BinSearchTree`.

The runtime complexity in the following table depends on the implementation. I've listed the complexity for an 'ideal' implementation below. These methods are *in addition* to those supported by `BinSearchTree`.

Method	Function	Runtime complexity (worst)
(inherit)	(from `BinSearchTree`)	(ADT 2.3.4)
`SplayTree.splay(`$x \in$ `Node)`	Splay x	$\mathcal{O}$ (depth of x)

Although I haven't listed it in the algorithms section of this document (section 3), the `BinSearchTree` operations are modified by requiring that a node is splayed after the operation finishes. The node to splay is as in the following table.

Method	Splay this
Search for key k	The node at which k was found, if k was found; else, the parent of the leaf node hit
Insert	Node inserted
Remove key k	Parent of the *internal* node that was deleted (not necessarily the node which originally stored key k; cf. 2.3.4)

Algorithms this data structure(s) may utilise:

- Search:
  - Binary search (algo 15)
- Binary Search Tree Methods:
  - BinSearchTree.remove() (algo 16)
- Splay Tree Methods:
  - SplayTree.splay() (algo 19)

Data Structure 2.3.7 (`MultiWaySearchTree` extends `Tree` (ADT 2.3.1))
Associated classes: `MWSearchTree` = `MultiWaySearchTree<Key, Value>`, `Node` = `Node<Key, Value>`, `Key`, `Value`.

Multi-way search trees are a structure for storing sorted data in an optimised manner. They generalise the notion of a binary search tree, and are the basis for (a, b) trees.

Definition. $T \in$ `Tree` is a *multi-way search tree* iff

- `Key` is totally ordered (by some comparator $\leq$), and,
- All leaves in T store no data (could be regarded as null pointers), and,
- All non-leaves p in T storing keys $k_1, \ldots, k_p$ and children $c_0, \ldots, c_p$ have the property that for each $i \in \{0, \ldots, p-1\}$, key$(c_i) < k_{i+1} <$ key(c_{i+1}).

`MultiWaySearchTree` stores:

- `parent` $\in$ `Node`: pointer to the parent of this `Node`
- `keys` $\in$ `List<Key>`: keys stored at this node
- `values` $\in$ `List<Value>`: values stored at this node
- `children` $\in$ `List<Node>`: Children of this node

The runtime complexity in the following table depends on the implementation. I've listed the complexity for an 'ideal' implementation below. These methods are *in addition* to those supported by `BinTree`.

Method	Function	Runtime complexity (worst)
(inherit)	(from `BinTree`)	(ADT 2.3.2)
`MWSearchTree.mWIOTraveral()`	Multi-way in-order traversal	$\mathcal{O}$ (number of nodes)
`MWSearchTree.mWSearch()`	Multi-way search	$\mathcal{O}(h \cdot k)$ where h is the height of the tree and k is the maximum number of keys stored at each node
`Node.keys()`	Return `this.keys`	$\Theta(1)$
`Node.values()`	Return `this.values`	$\Theta(1)$

Multi-way search trees are good for storing sparse data, while maintaining a 'binary search'-style algorithm. They are preferable to binary search trees if your objective is to *minimise tree height* (e.g. database systems).

Algorithms this data structure(s) may utilise: (none yet)

We only did a brief overview of red-black trees, so I'm just leaving brief notes.

Notes 2.3.8 (`RedBlackTree` extends `MultiWaySearchTree` (data structure 2.3.7))
The idea is loosely:

- Colour nodes either red or black,
- Simplify the ideas of $(2, 4)$ trees,
- Require fewer restructurings than $(2, 4)$ trees.

Comparison vs AVL trees (data structure 2.3.5):

Thing	AVL	Red-black
balance	stricter	not
search	faster	not
insert	slower	not
delete	slower	not

We only did a brief overview of B-trees, so I'm just leaving brief notes.

Notes 2.3.9 (`BTree` extends `MultiWaySearchTree` (data structure 2.3.7))
The idea is loosely to optimise for fewest node lookups.

- Hence also very shallow trees are developed,
- Great for storing in external memory (cf. databases).

2.4 Sets, Maps and Hashing

ADT 2.4.1 (`Set`)
Associated classes: `Set` = `Set<Data>`, `Data`.

A dynamic-size unordered collection of n items which does not maintain duplicate items. Useful for querying whether an item has been seen before. `Set` stores

- `size`: n

The runtime complexity in the following table depends on the implementation. I've listed the complexity for an 'ideal' implementation below.

Method	Function	Runtime complexity (worst)
Memory usage	store	$\Theta(n)$
`build(`X`)`	Create new data structure to store X	depends on implementation
`add(`x`)`	Store x in this set.	$\Theta(1)$ (perhaps amortised from raw $\mathcal{O}(n)$, depending on implementation)
`remove(`x`)`	Remove x from this set.	$\Theta(1)$

Method	Function	Runtime complexity (worst)
`contains`(x)	Return `true` iff x is in this set.	idk
`iterator`()	Return an iterator for this set.	idk
`union`(`other` $\in$ `Set`)	Return a new `Set` representing `this` $\cup$ `other`.	idk
`intersection`(`other` $\in$ `Set`)	Return a new `Set` representing `this` $\cap$ `other`.	idk
`difference`(`other` $\in$ `Set`)	Return a new `Set` representing `this` $\setminus$ `other`.	idk
`addAll`(`other` $\in$ `Set`)	`this` $\leftarrow$ `this` $\cup$ `other`.	idk
`retainAll`(`other` $\in$ `Set`)	`this` $\leftarrow$ `this` $\cap$ `other`.	idk
`removeAll`(`other` $\in$ `Set`)	`this` $\leftarrow$ `this` $\setminus$ `other`.	idk

Variants:
- `Multiset` (aka `Bag`): unordered collection of objects which may maintain duplicate entries

Algorithms this data structure(s) may utilise: (none yet)

ADT 2.4.2 (`Map`)
Associated classes: `Map` = `Map<Key, Value>`, `Entry` = `Entry<Key, Value>`, `Key`, `Value`.

A mapping of n distinct keys to (perhaps not distinct) values. Useful for maintaining a partial function `Key` $\rightharpoonup$ `Value`. `Map` stores

- `size`: n

The runtime complexity in the following table depends on the implementation. I've listed the complexity for an 'ideal' implementation below.

Method	Function	Runtime complexity (worst)
Memory usage	store	$\Theta(n)$
`build`(X)	Create new data structure to store X	depends on implementation
`get`(k $\in$ `Key`) $\to$ `Value`	Return associated value, or `null` if not present.	$\Theta(1)$
`put`(k $\in$ `Key`, v $\in$ `Value`) $\to$ `Value`	Store $k \mapsto v$ in this map, and return old `Value` (or `null` if not present).	$\Theta(1)$
`remove`(k $\in$ `Key`)	Delete $k \mapsto$ (whatever) in this map, and return old `Value` (or `null` if not present).	idk
`size`()	Return n.	$\Theta(1)$
`isEmpty`()	Return `true` iff $n = 0$.	$\Theta(1)$
`entrySet`() $\to$ `Set<Entry>`	Return set of (key, value) pairs maintained.	idk
`keySet`() $\to$ `Set<Key>`	Return set of keys maintained.	idk
`values`() $\to$ `Collection<Value>`	Return collection of values maintained.	idk

Note: Keys must be unique.
Implementation strategies:
- Hash tables
- Unsorted list
- Sorted list

Algorithms this data structure(s) may utilise: (none yet)

Hash tables don't really constitute an ADT, so I'm leaving only brief notes regarding them. For this part of the document, let `Key`, `Value`, `Container` = `Container<Key, Value>` be given data structures.

Definition 2.4.3 (Hash function)
Let $N \in \mathbb{Z}_{>0}$ be a positive integer, and let hc, cmp and hash be functions.

hc is a **hash code** function iff it is a function `Key` $\to \mathbb{Z}$.

cmp is a **compression function** iff it is a function $\mathbb{Z} \to \{0, \ldots, N-1\}$.

hash is a **hash function** iff it is the composition $\text{hash} = \text{cmp} \circ \text{hc} :$ `Key` $\to \{0, \ldots, N-1\}$ of a compression function with a hash code function.

Definition 2.4.4 (Hash table)
Let $N \in \mathbb{Z}_{>0}$ be a positive integer, let A be an object and h be a function.

(A, h) is a **hash table** iff $A \in$ `Array<Container<Key, Value>>` and h is a hash function.

Data Structure 2.4.5 (`HashTable`)
Associated classes: `HashTable` = `HashTable<Key, Value>`, `Container` = `Container<Key, Value>`, `Key`, `Value`.

A hash table is motivated by implementing a `Map<Key, Value>`. The ideas are that:

- Implementing a `Map<`$\{0, \ldots, N-1\}$`, Value>` is *really easy* by indexing into an array of size N;
- We can use a hash function $h :$ `Key` $\to \{0, \ldots, N-1\}$ to convert keys into indices for an array;
- Using our hash function, we can implement a `Map<Key, Value>` by hashing our `Key`s into integers in the range $\{0, \ldots, N-1\}$, and then using a `Map<`$\{0, \ldots, N-1\}$`, Value>`.
  - Effectively, h translates our `Key`s into integers, so that our array can pretend that keys are indices.
  - A commutative diagram is

$$\begin{tikzcd}
 & \mathbb{Z} \arrow[d, "\text{compress}"] \\
\texttt{Key} \arrow[ur, "\text{hashCode}"] \arrow[r, "h"] \arrow[dr, dashed, "\text{the } \texttt{Map} \text{ we want}"'] & \{0, \ldots, N-1\} \arrow[d, "\text{index into array}"] \\
 & \texttt{Value}
\end{tikzcd}$$

Goal for hash codes: try to inject `Key` $\hookrightarrow \mathbb{Z}$; i.e. try to reduce occurances of distinct keys mapping to common integers.

Some example hash codes are, where hc is a function that accepts components of a key and returns integers, $z \in \mathbb{Z}$ is fixed, $s \in \mathbb{Z}_{>0}$ is fixed, and $<<$ denotes cyclic bit-shift:

$$\text{ComponentSum} : \texttt{Keys} \longrightarrow \mathbb{Z}$$

$$\text{components:} \quad (b_\alpha, \ldots, b_0) \longmapsto \sum_{i=0}^{\alpha} (\text{hc}\,(b_i))$$

$$\text{PolynomialAccumulation} : \texttt{Keys} \longrightarrow \mathbb{Z}$$

$$\text{bitstring:} \quad b_\alpha \cdots b_0 \longmapsto \sum_{i=0}^{\alpha} \left(b_i \cdot z^i\right)$$

$$\text{CyclicShift} : \texttt{Keys} \longrightarrow \mathbb{Z}$$

$$\text{bitstring:} \quad b_\alpha \cdots b_0 \longmapsto (b_\alpha \cdots b_0 << s) \quad \text{regarded as an integer}$$

$z \in \{33, 37, 39, 41\}$ seem to work well in practice.

Goal for compression functions: try to surject $\mathbb{Z} \twoheadrightarrow \{0, \dots, N-1\}$ and try to get an injection `Keys` $\overset{h}{\hookrightarrow} \{0, \dots, N-1\}$; i.e. try to reduce hash collisions.

Some example compression functions are, where N is the table size:

$$\begin{aligned} \text{division} : \mathbb{Z} &\longrightarrow \{0, \dots, N-1\} & N \text{ is prime} \\ x &\longmapsto x \bmod N \\ \text{MAD} : \mathbb{Z} &\longrightarrow \{0, \dots, N-1\} & p > N \text{ is prime, and} \\ x &\longmapsto ((a \cdot x + b) \bmod p) \bmod N & a, b \in \{0, \dots, p-1\} \end{aligned}$$

Goal for collision handling: still store the key, but do it in a time- and space-efficient way. Some strategies are, where k is the key stored:

Separate chaining: Each entry in the table stores a *list* of `Entry<Key, Value>` items.

- Lookups are now far more expensive as more hash collisions occur.
- The list may be sorted, which makes insertion slower but loopups faster.

Probing: Iterate for $i = 0, 1, \dots, N-1$, and "probe" the indices $f(k, i)$ (f will be defined below). Store into the first empty entry.

- Lookups are now more expensive.
- So is insertion.

Linear: $f(k, i) = (h(k) + i) \bmod N$
Quadratic: $f(k, i) = \left(h(k) + i^2\right) \bmod N$
Double hashing: $f(k, i) = (h(k) + id(k)) \bmod N$, where d is another hash function

Hashing performance: for any of `get()`, `put()`, `remove()`:

- *Expected* runtime: $\Theta(1)$
- *Worst* runtime: $\mathcal{O}(n)$ *(if all keys collide to the same value)*

The **load factor** is $\alpha = \frac{n}{N} = \frac{\text{size}}{\text{capacity}}$. The expected number of probes we need will be $\frac{1}{1-\alpha}$.
Acceptable load factors:

- Separate chaining: $\alpha \in [0.8, 1.0]$
- Probing: $\alpha \in [0, 2/3)$

See also:

- Bloom filters,
- Perfect hashing,
- Cuckoo hashing.

Final words of wisdom: "Don't make your own hash function." *~Joel, killer of dreams.*

2.5 Graphs

Definition 2.5.1 (Graph)
A pair $G = (V, E)$ is a **graph** iff V is a finite set (whose elements are called **vertices**) and E is a finite set (whose elements are called **edges**), and each $e \in E$ has two associated vertices $u, v \in V$.
Self-loops and parallel edges are permissible in this course.

Vertices and edges often store data, and may be augmented in a variety of ways (e.g. weighted edges).

Edges from u to v may be either

- *Directed*, and associate u to v, or,
- *Undirected*, and associate u to v and v to u.

A graph may store edges which are directed or undirected.

Note: I will not recount all the language used surrounding graphs, because it is much the same as in MATH2302. I'm only listing the terminology that is used differently to this course in this document.

Definition 2.5.2 (Directed graph, Undirected graph)
A graph $G = (V, E)$ is said to be *undirected* iff every edge $e \in E$ is undirected.

A graph $G = (V, E)$ is said to be *directed* iff every edge $e \in E$ is directed.

Definition 2.5.3 (Path, Simple path, Cycle, Simple cycle, Tree)
Let $G = (V, E)$ be a graph.

A **path** (MATH2302: *walk*) P in G is an alternating sequence $P = (v_0, e_1, v_1, \ldots, e_n, v_n)$ with each $v_i \in V$ and each $e_i \in E$ an edge $v_{i-1} \to v_i$. We say that P *starts* at v_0 and *ends* at v_n.

A **simple path** (MATH2302: *path*) P in G is a path in which each vertex and each edge is distinct.

A **cycle** (MATH2302: *walk which returns to where it started*) C in G is a path whose start vertex and end vertex are the same.

A **simple cycle** (MATH2302: *cycle*) C in G is a cycle $C = (v_0, e_1, v_1, \ldots, e_n, v_n)$ in which each v_i for $i \in \{1, \ldots, n\}$ is distinct, and each e_i is distinct. In essence, this is a cycle in which all but the first vertex is required to be distinct.

G is a **tree** iff G is an undirected graph which is connected and has no simple cycles. This corresponds with the MATH2302 definition, under the assumption that G is undirected.

Definition 2.5.4 (Density, Sparse graph, Dense graph)
Let $G = (V, E)$ be a directed or undirected graph with $v = |V|$ vertices and $e = |E|$ edges.

The **density** D of G is defined by

$$D = \begin{cases} \dfrac{2e}{v(v-1)} & \text{if } G \text{ is undirected} \\ \dfrac{e}{v(v-1)} & \text{if } G \text{ is directed} \end{cases}$$

and in either case, D is equal to the quotient

$$\frac{\text{number of edges in } G}{\text{maximum number of edges } G \text{ could have, if we keep } V \text{ fixed}}$$

We say that G is **sparse** (resp. **dense**) iff $e \approx \mathcal{O}(v)$ (resp. $e \approx \mathcal{O}\left(v^2\right)$).

Definition 2.5.5 (Strongly connected)
A directed graph $G = (V, E)$ is said to be **strongly connected** iff for all vertices $u, v \in V$, we have that v is reachable from u in G.

Definition 2.5.6 (DAG, Topological ordering)
A directed graph $G = (V, E)$ is a **Directed Acyclic Graph** (DAG) iff it is directed and contains no cycles.

Let $G = (V, E)$ be a directed graph. A **topological ordering** τ of G is an injective function $V \to \{1, \ldots, |V|\}$ that assigns to each vertex a unique label $1, \ldots, |V|$, and which is subject to the constraint that

$$\text{for all } u, v \in V(G), \quad u \to v \quad \implies \quad \tau(u) \le \tau(v)$$

i.e. if there is an edge $u \to v$ in G, then $\tau(u) \le \tau(v)$.

For my own reference, here is another definition of a topological ordering: τ is a topological ordering of a digraph $G = (V, E)$ iff τ is an injective functor from the category generated by G into any totally ordered set. The archetypal totally ordered set, when G is finite, is $\{1, \ldots, \operatorname{card}(V)\}$.

Lemma 2.5.7 (Characterisation of DAGs by topological orderings)
Let $G = (V, E)$ be a directed graph. Then, G is a DAG iff there is a topological ordering of G.

Proof.

($\Rightarrow$) Algo 23 is correct.

($\Leftarrow$) Any topological ordering τ (in the categorical sense) of G demonstrates that the category C generated by G is isomorphic to a subcategory C' of some toset T. Supposing (for contradiction) that G had a cycle, then

$$G \text{ has a cycle} \xRightarrow{\text{cat. gen. by}} C \text{ has a cycle} \xRightarrow{\text{iso. cat.}} C' \text{ has a cycle} \xRightarrow{\text{subcat.}} T \text{ has a cycle}$$

so the toset T has a cycle. This is a contradiction.

ADT 2.5.8 (`Graph`)

Associated classes: `Graph` = `Graph<Vertex, Edge>`, `Vertex` = `Vertex<VData>`, `Edge` = `Edge<EData>`, `VData`, `EData`.

A model of a graph in which vertices and edges store associated data.

The runtime complexity in the following table depends on the implementation. I've listed the complexity for select operations separately. Methods prefixed by "*Graph*." could be made into methods of the appropriate `Vertex` or `Edge` class, so long as objects of type `Vertex` and of type `Edge` are not stored in two different `Graph`s simultaneously.

Method	Function
Memory usage (all)	store
`Graph.build`(V)	Create new data structure to store collection V of vertices
`Graph.numVertices()`	Return number of vertices in this graph
`Graph.vertices()`	Return a collection (or iteration) of vertices of this graph
`Graph.numEdges()`	Return number of edges in this graph
`Graph.edges()`	Return a collection (or iteration) of edges of this graph
`Graph.getEdge`($u \in$ `Vertex`, $v \in$ `Vertex`) $\to$ `Edge`	Return an edge $u \to v$, or `null` if no such edge exists
Graph.`endVertices`($e \in$ `Edge`) $\to$ `Array<Vertex>`	Return an array (or pair) $[u, v]$ of length 2 such that $u \xrightarrow{e} v$
Graph.`opposite`($u \in$ `Vertex`, $e \in$ `Edge`)	Return the vertex v such that $u \xrightarrow{e} v$ in this graph, or `null` if no such v exists.
Graph.`outDegree`($v \in$ `Vertex`)	Returns number of outgoing edges from v
Graph.`outgoingEdges`($v \in$ `Vertex`) $\to$ `Iteration<Vertex>`	Returns an iteration of the outgoing edges from v
Graph.`inDegree`($v \in$ `Vertex`)	Returns number of incoming edges to v
Graph.`incomingEdges`($v \in$ `Vertex`) $\to$ `Iteration<Vertex>`	Returns an iteration of the incoming edges to v
`Graph.insertVertex`($x \in$ `VData`) $\to$ `Vertex`	Maintain a new `Vertex` in this `Graph` to store data x, and return this vertex
`Graph.insertEdge`($u \in$ `Vertex`, $v \in$ `Vertex`, $x \in$ `EData`) $\to$ `Edge`	Maintain a new `Edge` $u \to v$ in this `Graph` which stores data x, and return this edge
`Graph.removeVertex`($v \in$ `Vertex`) $\to$ `void`	Remove v and all edges incident with v from this graph
`Graph.removeEdge`($e \in$ `Edge`) $\to$ `void`	Remove e from this graph

Method	Function
`Vertex.element()`	Return data stored here
`Edge.element()`	Return data stored here

A comparison of worst-case big-$\mathcal{O}$ runtime complexity between implementations is provided here. In this table, V denotes the number of vertices and E denotes the number of edges.

Method	`ELGraph` (DS 2.5.13)	`ALGraph` (DS 2.5.14)	`AMGraph` (DS 2.5.15)
Memory usage (all)	$V + E$	$V + E$	V^2
outgoingEdges($v \in$ `Vertex`),	E	$\deg(v)$	V
incomingEdges($v \in$ `Vertex`)	E	$\deg(v)$	V
getEdge($u, v \in$ `Vertex`)	E	$\min\{\deg(u), \deg(v)\}$	1
insertVertex(x)	1	1	V^2
insertEdge(u, v, x)	1	1	1
removeVertex(v)	E	$\deg(v)$	V^2
removeEdge(e)	1	1	1

Table 23: Comparison of graph implementations

Some rudimentary observations:

- `EdgeListGraph` is pretty mediocre, but is never bad;
- `AdjacencyListGraph` only ever improves upon `EdgeListGraph` (although it does require more moving parts to get going);
- `AdjacencyMatrixGraph` has a high cost for most operations regarding vertices, but is by far the fastest when it comes to accessing edge data – one may wish to use this implementation if they know ahead of time that they will never have to modify the vertices.

Algorithms this data structure(s) may utilise:

- Graph Algorithms:
  - Graph Traversals:
    - Breadth-first search (algo 21)
    - Depth-first search (algo 20)
  - Graph Problem Solvers:
    - Strong connectivity solver (algo 22)
    - Topological sort (algo 23)

Here are some graph problems we have encountered in the course.

Problem 2.5.9 (Pathfinding)
Let $G = (V, E)$ be a graph and $u, v \in V$ be vertices. **Determine** if there is a path $u \rightsquigarrow v$.

A related problem: If G is weighted and there is a path $u \rightsquigarrow v$, **find** a shortest path.

Solution.
Use your favourite graph traversal, e.g. depth-first search (algo 20) or breadth-first search (algo 21), starting from u.

Finding a path of minimum weight can be done by Dijkstra's algorithm.

Problem 2.5.10 (Connectivity)
Let G be a graph. **Determine** if G is connected.

Solution.
Use your favourite graph traversal, e.g. depth-first search (algo 20) or breadth-first search (algo 21), starting from any vertex of G.

Problem 2.5.11 (Strong Connectivity)
Let $G = (V, E)$ be a directed graph. **Determine** whether for all $u, v \in V$, there is a path $u \rightsquigarrow v$.

Solution.
See algo 22.

Problem 2.5.12 (Topological Sorting)
Let $G = (V, E)$ be a DAG. **Find** a topological ordering of G.

Solution.
See algo 23.

We now give some implementations of the `Graph` ADT.

Data Structure 2.5.13 (`EdgeListGraph` implements `Graph` (ADT 2.5.8))
Associated classes: `EdgeListGraph` = `ELGraph<Vertex, Edge>`, `Vertex` = `Vertex<VData>`, `Edge` = `Edge<EData>`, `VData`, `EData`.

An implementation of a `Graph` (ADT 2.5.8) by edge lists. `ELGraph` stores

- `numVertices` $\in \mathbb{Z}_{\geq 0}$: number of vertices
- `numEdges` $\in \mathbb{Z}_{\geq 0}$: number of edges
- `vertices` $\in$ `Sequence<Vertex>`: vertex list
- `edges` $\in$ `Sequence<Edge>`: edge list

`Vertex` stores

- `element` $\in$ `VData`: data stored here
- `position` $\in$ `Positions<ELGraph.vertices>`: position in vertex list

`Edge` stores

- `element` $\in$ `EData`: data stored here
- `position` $\in$ `Positions<ELGraph.edges>`: position in edge list
- `origin` $\in$ `Vertex`: as in $\texttt{origin} \xrightarrow{\text{this } \texttt{Edge}} \texttt{destination}$
- `destination` $\in$ `Vertex`: as in $\texttt{origin} \xrightarrow{\text{this } \texttt{Edge}} \texttt{destination}$

Algorithms this data structure(s) may utilise:

- Graph Algorithms:
  - Graph Traversals:
    - Breadth-first search (algo 21)
    - Depth-first search (algo 20)
  - Graph Problem Solvers:
    - Strong connectivity solver (algo 22)
    - Topological sort (algo 23)

Data Structure 2.5.14 (`AdjacencyListGraph` extends `EdgeListGraph` (DS 2.5.13))
Associated classes: `AdjacencyListGraph` = `ALGraph<Vertex, Edge>`, `Vertex` = `Vertex<VData>`, `Edge` = `Edge<EData>`, `VData`, `EData`.

An implementation of a `Graph` (ADT 2.5.8) by adjacency lists. `ALGraph` augments to

- (inherit)

`Vertex` augments to

- (inherit from `ELGraph.Vertex`)
- `incidentEdges` $\in$ `Sequence<Edge>`: sequence of incident edges

`Edge` augments to

- (inherit from `ELGraph.Edge`)
- `positions` $\in$ `Collection<Position<Vertex.incidentEdges>>`: collection storing, for each `Vertex` of the relevant `ALGraph` with which this `Edge` incident, the position in that `Vertex.incidentEdges` sequence which this `Edge` occupies. Probably* good to implement this via a `Map` $v \mapsto$ (position in `.incidentEdges`).

*This change defines the `AdjacencyMapGraph` class.

One could alternatively get rid of the `Edge` class entirely, should the edges not need to store any data.

Algorithms this data structure(s) may utilise:

- Graph Algorithms:
  - Graph Traversals:
    - Breadth-first search (algo 21)
    - Depth-first search (algo 20)

- Graph Problem Solvers:
  - Strong connectivity solver (algo 22)
  - Topological sort (algo 23)

Data Structure 2.5.15 (`AdjacencyMatrixGraph` extends `EdgeListGraph` (DS 2.5.13))
Associated classes: `AdjacencyMatrixGraph` = `ALMGraph<Vertex, Edge>`, `Vertex` = `Vertex<VData>`, `Edge` = `Edge<EData>`, `VData`, `EData`.

An implementation of a `Graph` (ADT 2.5.8) by an adjacency matrix. `AMGraph` augments to

- (inherit)
- `adjacencyMatrix` $\in$ `Matrix`$_{\text{square}}$`<Edge>`: matrix storing at entry (i, j) a `Edge` $u \to v$, where $u, v \in$ `Vertex` have `keys` i and j respectively. Could be a collection of edges instead.

`Vertex` augments to

- (inherit from `ELGraph.Vertex`)
- `key` $\in \mathbb{Z}_{\geq 0}$: corresponding index in `AMGraph.adjacencyMatrix`

One could alternatively get rid of the `Edge` class entirely, should the edges not need to store any data.

Algorithms this data structure(s) may utilise:

- Graph Algorithms:
  - Graph Traversals:
    - Breadth-first search (algo 21)
    - Depth-first search (algo 20)
  - Graph Problem Solvers:
    - Strong connectivity solver (algo 22)
    - Topological sort (algo 23)

3 Algorithms

3.1 Sort

Definition 3.1.1 (Stable sort)
Let $\mathcal{A}$ be an algorithm which sorts objects (k, v) by their keys k. We say that $\mathcal{A}$ is *stable* iff for each fixed key k, the order in which the values v appear in the sorted output of $\mathcal{A}$ is the same as the order they appeared in the unsorted input to $\mathcal{A}$.

3.1.1 Comparison sort

All comparison sorts (except perhaps heap sort) are *stable* sorts.

For a comparison of comparison sorts, see table 24.

Algo	In-place?	Worst runtime	Avg. runtime	Best runtime
Selection	yep	$\Theta\left(n^2\right)$	same	same
Insertion	yep	$\Theta\left(n^2\right)$	same	same
Merge	nope	$\Theta\left(n\log(n)\right)$	same	same
Quick	depends on implementation	$\mathcal{O}\left(n^2\right)$	$\Theta\left(n\log(n)\right)$	same

Table 24: Comparison of comparison sorts

Theorem 3.1.2 (Runtime of comparison sorts)
Let $\mathcal{A}$ be a comparison sort algorithm with input size n. Then, $\mathcal{A}$ runs in $\Omega\left(n\log(n)\right)$ time.

Algorithm 1: Selection Sort

```
/* This method is a stable sort.                                    */
/* Runtime complexity: Θ(n^2)                                       */
method selectionSort(A ∈ GeneralLinearStructure, n ∈ ℤ≥0) → void
    Input    : A of length ≤ n
    Requires: A is totally ordered by ≤
    Does     : In-place sorts A
    if n > 1 then
        maxIndex ← 0;
        for i ← 1 to n − 1 do
            if A[i] > A[maxIndex] then
                maxIndex ← 0;
        // Swap max with last
        swap(A[maxIndex], A[n − 1]);
        // Sort the rest
        selectionSort(A, n − 1);
```

Algorithm 2: Insertion Sort

```
/* This method is a stable sort.                                    */
/* Runtime complexity: Θ(n²)                                        */
method insertionSort(A ∈ GeneralLinearStructure, n ∈ ℤ≥0) → void
    Input    : A of length ≤ n
    Requires: A is totally ordered by ≤
    Does     : In-place sorts A

    for i ← 1 to n − 1 do
        valueToInsert ← A[i];
        j ← i − 1;
        // Find where to insert valueToInsert
        while j ≥ 0 and A[j] > valueToInsert do
            // Shift inputs upwards
            A[j + 1] ← A[j];
            j ← j − 1;
        // j is the index of the first value ≤ valueToInsert
        A[j + 1] ← valueToInsert;
```

Algorithm 3: Merge Sort

```
/* This method is a stable sort.                                    */
/* Runtime complexity: Θ(n²)                                        */
method mergeSort(A ∈ GeneralLinearStructure, l, r ∈ ℤ≥0) → void
    Input    : A of length > r
    Does     : Destructively sort A[{l, ..., r}]

    if l < r then
        m ← ⌊(l+r)/2⌋;
        mergeSort(A, l, m) ;                              // Divide
        mergeSort(A, m + 1, r) ;                          // Divide
        merge(A, l, m, r) ;                               // Conquer

method merge(A ∈ GeneralLinearStructure, l, m, r ∈ ℤ≥0) → void
    Input    : Structure A, left index l, middle index m, right index r
    Requires: A has length > r + 1 and 0 ≤ l ≤ m ≤ r
    Does     : Replace A by the sorted union of A[{l, ..., m − 1}] and A[{m, ..., r − 1}]

    Llength ← m − l + 1;
    Rlength ← r − m;
    L ← A[{l, ..., m − 1}];
    R ← A[{m, ..., r − 1}];
    Aind ← l;
    Lind ← 0;
    Rind ← 0;
    // Merge
    while Lind < Llength and Rind < Rlength do
        if L[Lind] ≤ R[Rind] then
            A[Aind++] ← L[Lind++];
        else
            A[Aind++] ← R[Rind++];
    // Copy leftovers. At most one of L, R is non-empty
    while Lind < Llength do
        A[Aind++] ← L[Lind++];
    while Rind < Rlength do
        A[Aind++] ← R[Rind++];
```

Algorithm 4: Quick Sort

```
/* Worst-case runtime complexity: O(n^2) */
/* Average-case, best-case runtime complexity: Θ(n log(n)) */
method quickSort(A ∈ GeneralLinearStructure, n ∈ ℤ_{>0})
  → GeneralLinearStructure
    Input   : Structure A of length ≤ n
    Returns : Destructively sorted copy of A
    if n = 1 then
        return A;
    // Else...
    // Divide
    p ← pivot index chosen from {0, ..., n − 1};        // often randomly chosen
    (L, E, G, l, e, g) ← partition(A, p);
    // Recurse
    L ← quickSort(L, l);
    G ← quickSort(G, g);
    // Conquer
    return L.appendAll(E).appendAll(G);        // appendAll() does the obvious
method partition(A ∈ GeneralLinearStructure, p ∈ ℤ_{≥0})
  → GeneralLinearStructure^3 × ℤ^3_{≥0}
    Input   : Structure A of length > p,
              p index in A of pivot A[p]
    Returns : (L, E, G, l, e, g) where:
        • L contains all things a ∈ A with a < A[p]
        • E contains all things a ∈ A with a = A[p]
        • G contains all things a ∈ A with a > A[p]
        • l, e, g are the lengths of L, E, G respectively
        • the order in A is maintained in L, E, G
    L, E, G, ← empty sequences of capacity length(A);
    l, e, g ← 0;
    pivot ← A.remove(p);
    while A is not empty do
        element ← A.remove(A.first());
        if element < pivot then
            L.add(element);
            l ← l + 1;
        else if element = pivot then
            E.add(element);
            e ← e + 1;
        else
            G.add(element);
            g ← g + 1;
    return (L, E, G, l, e, g);
```

Algorithm 5: Heap Sort

/* This method is not a stable sort. */
/* This method can be made in-place if A is an array and the heap you construct is stored using A. */
/* Runtime complexity: $\Theta(n \log(n))$ */
method heapSort($A \in$ *GeneralLinearStructure*, $n \in \mathbb{Z}_{\geq 0}$) $\rightarrow$ *void*
  Input : A of length n
  Does : Destructively sort A
  // Put stuff to sort in the auto-sorting Heap structure
  sorter $\leftarrow$ Heap.build(A, n);
  // Read sorted data
  while not sorter.isEmpty() **do**
    A.append(sorter.removeMin());
  // This algorithm is so cool omg

3.1.2 Non-comparison sort

Algorithm 6: Bucket Sort

/* This method is a stable sort. */
/* Runtime complexity $\mathcal{O}(n + N)$ */
method bucketSort($A \in$ *GeneralLinearStructure*, $n \in \mathbb{Z}_{\geq 0}$, $N \in \mathbb{Z}_{>0}$) $\rightarrow$ *void*
  Input : Structure A of n key-value pairs $(k, v) \in A$. The keys k are elements $k \in \{0, \ldots, N-1\}$.
  Does : Destructively sort A by keys
  Buckets $\leftarrow$ new Array<List<$\mathbb{Z}_{\geq 0}$>> of length N; // initially $[\varnothing, \ldots, \varnothing]$
  // Sort into buckets
  for pair **in** A **do**
    A.remove(pair);
    Buckets[pair.getKey()].append(pair);
  // Pour buckets into A
  for $i \leftarrow 0$ **to** $N-1$ **do**
    for pair **in** Bucket[i] **do**
      Bucket[i].remove(pair);
      A.append(pair);

Algorithm 7: Lexicographic Sort

/* Runtime complexity $\mathcal{O}(d \cdot T(n))$ for d the number of components in each tuple, T the runtime function of stableSort() and n the length of A. */

method lexicographicSort($A \in$ *GeneralLinearStructure*, $d \in \mathbb{Z}_{>0}$) $\rightarrow$ *void*

Input : Structure A of d-tuples

Requires: The data type $\mathtt{D}_i$ of the i-th component is totally ordered by $\leq$, for each $i \in \{1, \ldots, d\}$

Does : Destructively sort A according to lexicographic order

for $i \leftarrow d$ **downto** 1 **do**

  stableSort(A, *i-th component*); // keys are the i-th component

Algorithm 8: Radix Sort

/* Specialisation of lexicographicSort() which uses bucketSort() and applies only to non-negative integers. */

/* Runtime complexity $\mathcal{O}(d \cdot (n + N))$ */

method radixSort($A \in$ *GeneralLinearStructure*, $d \in \mathbb{Z}_{>0}$, $n \in \mathbb{Z}_{\geq 0}$, $N \in \mathbb{Z}_{>0}$) $\rightarrow$ *void*

Input : Structure A of d-tuples in $\{0, \ldots, N-1\}^d$ of length n

Does : Destructively sort A

for $i \leftarrow d$ **downto** 1 **do**

  bucketSort(A, n, N, *i-th component*); // keys are the i-th component

Algorithm 9: Binary Radix Sort

/* Specialisation of radixSort() which works in binary. */

/* Runtime complexity $\mathcal{O}(b \cdot n)$ */

method binaryRadixSort($A \in$ *GeneralLinearStructure*, $n \in \mathbb{Z}_{\geq 0}$, $b \in \mathbb{Z}_{>0}$) $\rightarrow$ *void*

Input : Structure A of length n storing b-bit non-negative integers

Does : Destructively sort A

for $i \leftarrow 0$ **to** $b - 1$ **do**

  bucketSort(A, n, 2, *i-th bit*); // keys are the i-th bit

3.2 Heap Methods

Algorithm 10: Upheap

```
method upheap(H ∈ Heap, z ∈ Node) → void
    Input : "Heap" H and node z to upheap (H may not technically be a heap at this
            point, but the point of this method is to fix that)
    Does  : Assuming H was not a heap because only the node z violates the heap
            property, fix H so that it is a heap again
    while z ≠ H.root and z.parent.data > z.data do
        swapData(z, z.parent);
        z ← z.parent;
```

Algorithm 11: Downheap

```
method downheap(H ∈ Heap) → void
    Input : "Heap" H and (H may not technically be a heap at this point, but the
            point of this method is to fix that)
    Does  : Assuming H was not a heap because only the (children of) the root
            violates the heap property, fix H so that it is a heap again
    while z.hasChildren() and (z.data > m.data for some child m of z) do
        m ← z.left or z.right, whichever has smallest data;
        swapData(z, m);
        z ← m;
```

Algorithm 12: Bottom-up heap construction; `Heap.build()`

```
method build(X ∈ Collection, n ∈ ℤ_{≥0}) → void
    Input    : Collection X of size n containing items to store in a new heap
    Requires: The elements of X are totally ordered by ≤
    Returns  : New heap storing X

    if n = 1 then
        return new Heap with only X.remove() at the root;
    // These can be found using a simple brute force algorithm
    (f, e) ← f, i ∈ ℤ_{≥0} such that n = 2^0 + ··· + 2^e + f and 0 < f ≤ 2^{e+1};
    heaps, done ← new empty queues of capacity 2^{e+1};
    // Build lowest level ("floor") of heap
    for dc ← 1 to f do
        heaps.enqueue(new Heap.build(X.remove()));
    // Each non-floor level
    for i ← e downto 0 do
        // Construct level i
        for dc ← 1 to 2^i do
            heap ← new Heap.build(X.remove());
            // Merge two lower heaps with this heap
            if not heaps.isEmpty() then
                heap.root.setLeft(heaps.dequeue().root);
            if not heaps.isEmpty() then
                heap.root.setRight(heaps.dequeue().root);
            heap.downheap();                                  // Make this a heap
            done.enqueue(heap);
        // Register level i
        while not done.isEmpty() do
            heaps.enqueue(done.dequeue());
    // Return only heap
    return heaps.dequeue();
```

Algorithm 13: `Heap.insert()`

```
method insert(H ∈ Heap, x ∈ Data) → void
    Input    : This heap H, and data x to insert
    Does     : Store x in this heap

    insertHere ← node in H to insert into;     // see array-based implementation
    insertHere.setData(x);
    H.upheap();                                              // Fix the heap
```

Algorithm 14: `Heap.removeMin()`

```
method removeMin(H ∈ Heap) → void
    Input    : This heap H
    Does     : (Assuming H implements a PriorityQueue) remove the min data in H
    Returns  : Return the min element from H

    w ← H.last;
    swapData(w, H.root);
    returnMe ← w.getData();
    H.remove(w);  // w.getParent().remove(w), unless w is the root
    H.downheap();                                            // Fix the heap
```

3.3 Search

Algorithm 15: Binary Search

```
method binsearch(T ∈ BinSearchTree<Key, Value>, k ∈ Key) → Node<Key, Value>
    Input   : Binary search tree T to search,
              Key k to search for
    Returns : Node<Key, Value> with key k stored in T, if it exists, or null otherwise.
    return binSearcher (T.getRoot(), k);
method binSearcher(cursor ∈ Node<Key, Value>, k ∈ Key) → Node<Key, Value>
    Input   : Node cursor currently being searched from,
              Key k to search for
    Returns : Node<Key, Value> with key k stored in the subtree rooted at the cursor, if
              it exists, or null otherwise.
    // If we hit a leaf, we're done searching
    if cursor = null then
        return null;
    // Obvious
    if cursor.getKey() = k then
        return cursor;
    // Recurse on children
    if k < cursor.getKey() then
        return binSearcher (cursor.getLeft(), k);
    else
        return binSearcher (cursor.getRight(), k);
```

3.4 Binary Search Tree Methods

Algorithm 16: `BinSearchTree.remove()`

method `remove`($T \in$ *BinSearchTree<Key, Value>*, $k \in$ *Key*) $\rightarrow$ *void*
  Input : This binary search tree T, and key k to remove
  Does : Removes a node with key k from T, if such a node exists
  Returns : The node that got removed, or null if no node was removed

```
killMe ← find(k, T.getRoot()); // Return a node in T with key k if exists, else null
if killMe = null then
    return null;
// k was in the tree somewhere
if killMe has 2 leaf children then
    Replace killMe by a leaf in killMe.getParent();
else if killMe has exactly 1 leaf child then
    Replace killMe by the non-leaf child of killMe in killMe.getParent();
else // killMe has no leaf children
    replacement ← getReplacement(killMe);
    killMe.setData(replacement.getData()); /* Replace the key-value pair in killMe by that in replacement */
    replace replacement by a leaf in replacement.getParent();
```

method `getReplacement`(*killMe* $\in$ *Node<Key, Value>*) $\rightarrow$ *void*
  Input : Node killMe to get replacement for
  Returns : The node whose key-value pair should be placed in killMe

```
returnMe ← killMe.getRight();
while returnMe.getLeft() is not a leaf do
    returnMe ← returnMe.getLeft();
return returnMe;
```

3.5 AVL Tree Methods

Algorithm 17: `AVLTree.insert()`, `AVLTree.remove()`

```
method insert(T ∈ AVLTree<Key, Value>, k ∈ key, v ∈ Value) → void
    Input    : This AVL tree T, and new data (k, v) to insert
    Does     : Inserts (k, v) into this tree

    newNode ← super.insert(k, v); /* Insertion from BinSearchTree. Return new
     node created/node overridden                                          */
    rebalance(T, newNode);

method remove(T ∈ AVLTree, k ∈ Key) → void
    Input    : Key k to remove in tree T
    Does     : Exactly what you expect it to do

    parent ← super.remove(k); /* Removal from BinSearchTree. Return parent of
     removed node, or null if nothing was removed.                          */
    if parent = null then
        return;
    rebalance(T, parent);
```

Algorithm 18: `AVLTree.rebalance()`, `AVLTree.triNodeRestructuring()`

```
method rebalance(T ∈ AVLTree<Key, Value>, newNode ∈ Node<Key, Value>) → void
    Input    : This AVL tree T, and node newNode to rebalance from
    Requires: An insertion or a removal was just performed on T, and newNode is the
              relevant node of interest
    Does     : Rebalances T, seeking upwards from newNode

    z ← newNode;
    while |height(z.getLeft()) − height(z.getRight())| ≤ 1 do
        if not z.hasParent() then
            return ; // Hit root without needing to rebalance
        z ← z.getParent();
    triNodeRestructuring(T, z);

method triNodeRestructuring(T ∈ AVLTree<Key, Value>, z ∈ Node<Key, Value>)
  → void
    Input    : Tree T to restructure in, and node z to restructure from
    Does     : Restructure at z

    y ← child of z with largest height;
    x ← child of y with largest height;
    (a, b, c) ← in-order listing of x, y, z; // i.e. a ≤ b ≤ c, with {a, b, c} = {x, y, z}
     /* This pseudocode isn't very good because I can't put in pictures to
        diagram what's going on. Refer to the lecture slides or your own
        personal notes for this :)                                          */
    if (a, b, c) = (z, y, x) then
        // Single rotation
        Make b the root of the subtree from z, preserving in-order structure of T;
    else if a, b, c) = (x, y, z) then // symmetric to previous case
        // Single rotation
        Make b the root of the subtree from z, preserving in-order structure of T;
    else if (a, b, c) = (z, x, y) then
        // Double rotation
        Make b the root of the subtree from c, preserving in-order structure of T;
        Make b the root of the subtree from z, preserving in-order structure of T;
    else // symmetric to previous case
        // Double rotation
        Make b the root of the subtree from a, preserving in-order structure of T;
        Make b the root of the subtree from z, preserving in-order structure of T;
```

3.6 Splay Tree Methods

Algorithm 19: `SplayTree.splay()`

```
method splay(T ∈ SplayTree<Key, Value>, x ∈ Node<Key, Value>) → void
    Input : Node x to splay in tree T
    Does  : Splays x to the root of T
    // This pseudocode isn't very good, because I can't insert visual
       diagrams! Refer to lecture slides or your own personal notes :)
    // Note: I call things zigs and zags as they were first presented in
       the lecture slides, which is different to what they are called in
       the original paper, and also what is in the lecture slides'
       flowchart.
    if x = T.getRoot() then
        return;
    y ← x.getParent();
    if y = T.getRoot() then
        if x is the left child of y then
            zag x, y;
            splay(T, x);
        else // x is the right child of y
            zig x, y;
            splay(T, x);
    z ← y.getParent();
    if x left y left z then // x left child of y and y left child of z
        zag-zag x, y, z;
        splay(T, x);
    else if x left y right z then
        zag-zig x, y, z;
        splay(T, x);
    else if x right y left z then
        zig-zag x, y, z;
        splay(T, x);
    else // x right y right z
        zig-zig x, y, z;
        splay(T, x);
```

3.7 Graph Algorithms

3.7.1 Graph Traversals

Traversals of directed graphs can be made by adapting the breadth-first search (algo 21) or depth-first search (algo 20) algorithms. The edge markings may change, though; for instance, directed dfs has

- unexplored edges,
- discovery edges,
- back edges,
- forward edges,
- cross edges

Depth-first search is good for:

- Computing the connected component of a vertex
  - Determining connectivity of a graph
- Computing a spanning forest of a graph (This is given by the vertices, and the discovery edges)

- Pathfinding (see lecture slides; I'm not copying that out)
- Cycle detection (see lecture slides; I'm not copying that out)

Algorithm 20: Depth-first search

```
method dfs(G ∈ Graph) → void
    Input    : A graph G
    Does     : Visits all vertices of G, and marks edges as either discovery edges or back
               edges
    Runtime: Θ(|V(G)| + |E(G)|) worst
    for v ∈ G.vertices() do
        Mark v as unvisited; // Could be implemented using a Set<Vertex>
    for e ∈ G.edges() do
        Mark e as unexplored; /* Could be implemented using a Map<Edge, ENUM>
         where ENUM consists of "unexplored", "back" and "discovery"        */
    for v ∈ G.vertices() do
        if v is not marked as visited then
            dfs(G, v);
method dfs(G ∈ Graph, v ∈ Vertex) → void
    Input    : Graph G to dfs in, and vertex v to dfs from
    Does     : Visits all vertices of G which are reachable from v, and marks edges as
               either discovery edges or back edges
    Mark v as visited;
    for e ∈ G.outgoingEdges(v) do
        w ← G.opposite(v, e);
        if e is marked as unexplored then
            Mark e as a discovery edge (for w);
            dfs(G, w); /* Could be done using a Stack<Vertex> instead,
             mirroring the implementation of bfs                        */
```

Breadth-first search is good for:
- Computing the connected component of a vertex
  - Determining connectivity of a graph
- Computing a spanning forest of a graph (This is given by the vertices, and the discovery edges)
- Compute paths to destination vertices which use a *minimal number of edges* from a given starting vertex

Algorithm 21: Breadth-first search

```
method bfs(G ∈ Graph) → void
    Input    : A graph G
    Does     : Visits all vertices of G, and marks edges as either discovery edges or cross
               edges
    Runtime: Θ (|V(G)| + |E(G)|) worst

    for v ∈ G.vertices() do
        Mark v as unvisited; // Could be implemented using a Set<Vertex>
    for e ∈ G.edges() do
        Mark e as unexplored; /* Could be implemented using a Map<Edge, ENUM>
         where ENUM consists of "unexplored", "cross" and "discovery"      */
    for v ∈ G.vertices() do
        if v is not marked as visited then
            bfs(G, v);

method bfs(G ∈ Graph, seed ∈ Vertex) → void
    Input    : Graph G to bfs in, and vertex seed to bfs from
    Does     : Visits all vertices of G which are reachable from seed, and marks edges as
               either discovery edges or cross edges

    traversal ← new empty Queue<Vertex>;
    traversal.enqueue(seed);
    while not traversal.isEmpty() do      /* short-circuit this if searching for
     minimum-edge paths */
        v ← traversal.dequeue();
        Mark v as visited;
        for e ∈ G.incidentEdges(v) do
            if e is marked as unexplored then    /* technically need to pre-mark as
             unexplored... idc. */
                w ← G.opposite(v, e);
                if w is not marked as visited then    /* technically need to pre-mark
                 as unvisited... idc. */
                    Mark e as a discovert edge (for w);
                    traversal.enqueue(w);
                else
                    Mark e as a cross edge;
```

3.7.2 Graph Problem Solvers

Algorithm 22: Strong connectivity (problem 2.5.11) solver

method `strongConnectivity`($G \in$ *`Graph`*) $\to$ *`boolean`*

  Input : A graph G

  Returns : `true` iff G is strongly connected

  Runtime: $\Theta(|V(G)\| + |E(G)\|)$ worst, assuming computation of G^{op} takes $\mathcal{O}(|V(G)| + |E(G)|)$ worst-case time

  if G.vertices().isEmpty() **then**

    return `true`;

  seed $\leftarrow$ any element of G.vertices();

  seedReachesEverything $\leftarrow$ `dfsVisitsAll`(G, *seed*); `/* returns true iff a depth-first search (algo 20) visits every vertex of` G`; could be any traversal */`

  if not seedReachesEverything **then**

    return `false`;

  $G^{\text{op}} \leftarrow G$ with all edges reversed;

  seedReachableByEverything $\leftarrow$ `dfsVisitsAll`(G^{op}, *seed*);

  return seedReachableByEverything;

Algorithm 23: Topological sort (problem 2.5.11 solver)

```
method topologicalSort(G ∈ Graph) → Map<Vertex, {1,...,n}>
    Input    : A directed graph G of size n = ‖V(G)‖
    Requires: G is a DAG
              G is connected
    Returns : A topological ordering of G

    Sorter ← a (deep) copy of G;
    n ← a reference to an integer storing G.numVertices();    /* I will refer to the
        reference as "n" and the value as "N" */
    τ ← new empty Map<Vertex, {1,...,N}>;
    while not Sorter.isEmpty() do
        v ← any vertex of Sorter with no outgoing edges;  // can be found using dfs
        τ.put(v, N);
        N ← N − 1;
        Sorter.remove(v);
    return τ;

/* Implementation of topologicalSort() by depth-first search (algo 20)   */
method topSortDFS(G ∈ Graph) → Map<Vertex, {1,...,n}>
    Input    : A directed graph G
    Requires: G is a DAG
    Returns : A topological ordering of G
    Runtime: Θ(|V(G)| + |E(G)|) worst; same as dfs()

    n ← a reference to an integer storing G.numVertices();    /* I will refer to the
        reference as "n" and the value as "N" */
    for v ∈ G.vertices() do
        Mark v as unvisited;
    τ ← new empty Map<Vertex, {1,...,N}>;
    for v ∈ G.vertices() do
        if v is not marked as visited then
            topSortDFS(G, v, τ, n);
    return τ;

/* Helper for topSortDFS(Graph)                                          */
method topSortDFS(G ∈ Graph, seed ∈ Vertex, τ ∈ Map<Vertex, ℤ>0>,
  n ∈ Reference<ℤ>0>) → void
    Input    : A directed graph G
    Requires: G is a DAG
    Does     : Updates τ and n as per the component of vertices of G reachable from seed

    Mark v as visited;
    for e ∈ G.outgoingEdges(v) do
        w ← G.opposite(v, e);
        if w is not marked as visited then
            // e is a discovery edge for w
            topSortDFS(G, w, τ, n);
        // else, e is a forward or cross edge
    τ.put(v, N);
    N ← N − 1;
```