

COMP3506 Algos and Datas Summary

Gabriel Field

17/08/2023 - END OF COURSE

Contents

1	Boilerplate	1
2	Data Structures	3
2.1	General Linear Structures	3
2.1.1	Children of <code>StaticSequence</code>	4
2.1.2	Children of <code>DynamicSequence</code>	5
2.2	Stacks and Queues	6
2.2.1	Priority Queues	7
2.3	Trees	9
2.3.1	Search Trees	11
2.4	Sets, Maps and Hashing	14
2.5	Graphs	17
2.5.1	Definitions	17
2.5.2	Graph ADT	19
2.5.3	Graph problems	21
2.5.4	Graph ADT implementations	23
3	Strings, Pattern Matching and Compression	25
3.2	Pattern Matching	25
3.3	Compression	28
4	Algorithms	30
4.1	Sort	30
4.1.1	Comparison sort	30
4.1.2	Non-comparison sort	33
4.2	Heap Methods	35
4.3	Search	37
4.4	Binary Search Tree Methods	37
4.5	AVL Tree Methods	38
4.6	Splay Tree Methods	40
4.7	Graph Algorithms	40
4.7.1	Graph Traversals	40
4.7.2	Graph Problem Solvers	43
4.8	Strings, Pattern Matching and Compression	50

1 Boilerplate

This document contains a summary of *data structures* (section 2) and their associated *algorithms* (section 4).

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**) $\rightarrow D$ to mean “**method**() returns type **D**”. For example, **foo**($x \in X$) $\rightarrow Y$ 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 **Sets** of **Nodes**.

Some common abbreviations:

- “amo.”: amortised
- “obv.”: obvious

The L^AT_EX source code for this file, along with the Java code I wrote to generate the macro `\dataprintalgorithms`, 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:

- (2023-10-14 13:00) Content up to the **end of week 11 lectures** is now summarised here. This includes **all weeks of assessable content**.
- (2023-10-05 11:22) Content up to the **end of week 9 lectures** is now summarised here.
- (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
<code>build(X)</code>	Create new data structure to store X
<code>len()</code>	Return n
<code>get($i \in \{0, \dots, n-1\}$)</code>	Return x_i
<code>set($i \in \{0, \dots, n-1\}, x$)</code>	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
<code>build(X)</code>	Create new data structure to store X
<code>len()</code>	Return n

Method	Function
<code>get($i \in \{0, \dots, n-1\}$)</code>	Return x_i
<code>set($i \in \{0, \dots, n-1\}, x$)</code>	Set x_i to x
<code>add(x)</code>	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	$\Theta(n)$
<code>build(X)</code>	Create new data structure to store X	idk lol
<code>len()</code>	Return n	$\Theta(1)$
<code>get($i \in \{0, \dots, n-1\}$)</code>	Return x_i	$\Theta(1)$
<code>set($i \in \{0, \dots, n-1\}, x$)</code>	Set x_i to x	$\Theta(1)$
<code>iterate(f a function)</code>	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	$\Theta(n)$
<code>build(X)</code>	Create new data structure to store X	$\Theta(n)$
<code>len()</code>	Return n	$\Theta(1)$
<code>get(i ∈ {0, ..., n - 1})</code>	Return x_i	$\Theta(n)$ (cf. <code>ExtensibleList</code>)
<code>set(i ∈ {0, ..., n - 1}, x)</code>	Set x_i to x	$\Theta(n)$ (cf. <code>ExtensibleList</code>)
<code>iterate(f a function)</code>	Iterate through collection	$\mathcal{O}(n \cdot T_f(s))$ for T_f the runtime of f and s the size of elements
<code>add(x)</code>	Add x to list (at either the head or the tail)	$\Theta(1)$
<code>insert(i ∈ {0, ..., n - 1}, x)</code>	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	$\Theta(n)$
<code>build(X)</code>	Create new data structure to store X	$\Theta(n)$
<code>len()</code>	Return n	$\Theta(1)$
<code>get(i ∈ {0, ..., n - 1})</code>	Return x_i	$\Theta(1)$ (cf. <code>LinkedList</code>)
<code>set(i ∈ {0, ..., n - 1}, x)</code>	Set x_i to x	$\Theta(1)$ (cf. <code>LinkedList</code>)

Method	Function	Runtime complexity (worst)
<code>iterate(f a function)</code>	Iterate through collection	$\mathcal{O}(n \cdot T_f(s))$ for T_f the runtime of f and s the size of elements
<code>append(x)</code>	Add x to the tail of the list	<i>amortised</i> $\Theta(1)$; raw $\mathcal{O}(n)$
<code>insert($i \in \{0, \dots, n-1\}, x$)</code>	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	$\Theta(n)$
<code>build(X)</code>	Create new data structure to store X	depends on implementation
<code>push($x \in \text{Data}$) \rightarrow \text{void}</code>	Push x onto the stack	$\Theta(1)$ (perhaps amortised from raw $\mathcal{O}(n)$, depending on implementation)
<code>pop() \rightarrow \text{Data}</code>	Return and remove the element at the top	$\Theta(1)$
<code>peek()</code> aka <code>top() \rightarrow \text{Data}</code>	Return the element at the top	$\Theta(1)$
<code>isEmpty() \rightarrow \text{boolean}</code>	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	$\Theta(n)$
<code>build(X)</code>	Create new data structure to store X	depends on implementation
<code>enqueue(x) → void</code>	Enqueue x onto the back of the queue	$\Theta(1)$ (perhaps amortised from raw $\mathcal{O}(n)$, depending on implementation)
<code>dequeue() → Data</code>	Return and remove the element at the front	$\Theta(1)$
<code>first() → Data</code>	Return the element at the front	$\Theta(1)$
<code>isEmpty() → boolean</code>	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 (key, 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	$\Theta(n)$
<code>PQ.build(X)</code>	Create new data structure to store X	$\Theta(n)$, but depends on implementation
<code>PQ.insert(k ∈ Key, v ∈ Value)</code>	insert a new entry storing (k, v)	depends on implementation*
<code>PQ.removeMin() → Entry</code>	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)
<code>PQ.min()</code> \rightarrow <code>Entry</code>	Return the <code>Entry</code> with smallest <code>key</code> , and at the front amongst entries with the same <code>key</code>	depends on implementation*
<code>PQ.size()</code>	Return <code>size</code>	$\Theta(1)$
<code>PQ.isEmpty()</code>	Return <code>true</code> iff $n \neq 0$	$\Theta(1)$
<code>Entry.getKey()</code>	Return this <code>Entry</code> 's <code>key</code>	$\Theta(1)$
<code>Entry.getValue()</code>	Return this <code>Entry</code> 's <code>value</code>	$\Theta(1)$

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

Implementation strategies:

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

– *Make sure you tie-break priorities (e.g. with a counter)!

Runtime comparison (depending on implementation):

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

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 (inherit)	store (from <code>PriorityQueue</code>)	$\Theta(n)$ (ADT 2.2.3)
<code>APQ.remove($e \in$ <code>Entry</code>)</code>	Remove and return e , if it is present	depends on implementation*
<code>APQ.replaceKey($e \in$ <code>Entry</code>, $k \in$ <code>Key</code>)</code>	Replace key of e , and return the old key of e	depends on implementation*

Method	Function	Runtime complexity (worst)
APQ. replaceValue ($e \in \text{Entry}$, $v \in \text{Value}$)	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 <code>LinkedList</code>	sorted <code>ExtensibleList</code>	<code>Heap</code>
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` $\in \text{Node}$: pointer to the root of the tree (maybe an index in an array)

The nodes (of type `Node`) store:

- `parent` $\in \text{Node}$: pointer to the parent of this `Node`
- `data`: data stored at this node
- `children` $\in \text{Set<Node>}$: 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	$\Theta(n)$
<code>Tree.build(X)</code>	Create new data structure to store X	depends on implementation
<code>Tree.size()</code>	Return n	$\Theta(1)$
<code>Tree.isEmpty()</code>	Return <code>true</code> iff $n = 0$	$\Theta(1)$
<code>Tree.root()</code>	Return <code>root</code>	$\Theta(1)$
<code>Tree.iterator()</code>	Return an iterator for this tree	depends on implementation
<code>Tree.positions()</code>	Not in Joel's headcanon	depends on implementation
<code>Node.parent()</code>	Return <code>this.parent</code>	$\Theta(1)$
<code>Node.children()</code>	Return <code>this.children</code>	$\Theta(1)$
<code>Node.numChildren()</code>	Return <code>this.children.size()</code>	$\Theta(1)$
<code>Node.isInternal()</code>	Return <code>true</code> iff this node is internal; i.e. it has children	$\Theta(1)$

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

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

Method	Function	Runtime complexity (worst)
<code>Tree.replace($x \in \text{Node}$, $y \in \text{Node}$)</code>	Replace x with y	$\Theta(1)$
<code>Tree.addRoot($x \in \text{Node}$)</code>	Set the root of this Tree to x , and the old root to one of x 's children	$\Theta(1)$
<code>Tree.remove($x \in \text{Node}$)</code>	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** $\in \text{Node}$: pointer to the parent of this **Node**
- **data**: data stored at this node
- **left** $\in \text{Node}$: left child
- **right** $\in \text{Node}$: 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)
<code>Node.left()</code>	Return <code>this.left</code>	$\Theta(1)$
<code>Node.right()</code>	Return <code>this.right</code>	$\Theta(1)$

Definition (full level): Level l of a binary tree is *full* iff it contains 2^l non-null nodes.

Definition (complete tree): $T \in \text{BinTree}$ is *complete* iff every level except the bottom level is full, and all leaves are as leftmost as possible.

Definition (proper tree, full tree): $T \in \text{BinTree}$ 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: $H \in \text{BinTree}$ is a (min-)heap iff

- (*Heap-order*) For all nodes n in H such that $n \neq H.\text{root}$, $n.\text{data} \geq n.\text{parent}.\text{data}$ (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 <code>BinTree</code>)	(ADT 2.3.2)
<code>Heap.getLast()</code>	Return <code>this.last</code>	$\Theta(1)$
<code>Heap.insert($x \in \text{Data}$)</code>	Store new <code>Node</code> with data x in this <code>Heap</code>	$\mathcal{O}(\log(n))$

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:

- $H.\text{root}$ 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:

- $H.\text{last}$ is stored at index $n - 1$
- The next node to insert into will go at index n

This encoding is an injection $\text{Heap}/\simeq \hookrightarrow \text{Array}/\simeq$ 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. $T \in \text{BinTree}$ 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 <code>BinTree</code>)	(ADT 2.3.2)
<code>Node.key()</code>	Return <code>this.key</code>	$\Theta(1)$
<code>Node.value()</code>	Return <code>this.value</code>	$\Theta(1)$

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

Note. The standard binary search tree methods 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. $T \in \text{BinSearchTree}$ 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 <code>BinSearchTree</code>)	(ADT 2.3.4)
<code>AVLTree.rebalance(newNode \in Node)</code>	Rebalance this tree, seeking up from <code>newNode</code> .	$\mathcal{O}(\text{depth of newNode})$
<code>AVLTree.triNodeRestructuring(unbalanced \in Node)</code>	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 search 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 operations 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 <code>BinSearchTree</code>)	(ADT 2.3.4)
<code>SplayTree.splay($x \in \text{Node}$)</code>	Splay x	$\mathcal{O}(\text{depth of } x)$

Although I haven't listed it in the algorithms section of this document (section 4), 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 <i>internal</i> 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. $T \in \text{Tree}$ 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, \dots, k_p and children c_0, \dots, c_p have the property that for each $i \in \{0, \dots, p-1\}$, $\text{key}(c_i) < k_{i+1} < \text{key}(c_{i+1})$.

`MultiWaySearchTree` stores:

- `parent` $\in \text{Node}$: pointer to the parent of this `Node`
- `keys` $\in \text{List<Key>}$: keys stored at this node
- `values` $\in \text{List<Value>}$: values stored at this node
- `children` $\in \text{List<Node>}$: 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 <code>BinTree</code>)	(ADT 2.3.2)
<code>MWSearchTree.mWIOTraverse()</code>	Multi-way in-order traversal	\mathcal{O} (number of nodes)
<code>MWSearchTree.mWSearch()</code>	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
<code>Node.keys()</code>	Return <code>this.keys</code>	$\Theta(1)$
<code>Node.values()</code>	Return <code>this.values</code>	$\Theta(1)$

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	$\Theta(n)$
<code>build(X)</code>	Create new data structure to store X	depends on implementation
<code>add(x)</code>	Store x in this set.	$\Theta(1)$ (perhaps amortised from raw $\mathcal{O}(n)$, depending on implementation)
<code>remove(x)</code>	Remove x from this set.	$\Theta(1)$

Method	Function	Runtime complexity (worst)
<code>contains(x)</code>	Return true iff x is in this set.	idk
<code>iterator()</code>	Return an iterator for this set.	idk
<code>union(other ∈ Set)</code>	Return a new Set representing $\text{this} \cup \text{other}$.	idk
<code>intersection(other ∈ Set)</code>	Return a new Set representing $\text{this} \cap \text{other}$.	idk
<code>difference(other ∈ Set)</code>	Return a new Set representing $\text{this} \setminus \text{other}$.	idk
<code>addAll(other ∈ Set)</code>	$\text{this} \leftarrow \text{this} \cup \text{other}$.	idk
<code>retainAll(other ∈ Set)</code>	$\text{this} \leftarrow \text{this} \cap \text{other}$.	idk
<code>removeAll(other ∈ Set)</code>	$\text{this} \leftarrow \text{this} \setminus \text{other}$.	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 $\text{Key} \rightarrow \text{Value}$. `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	$\Theta(n)$
<code>build(X)</code>	Create new data structure to store X	depends on implementation
<code>get(k ∈ Key) → Value</code>	Return associated value, or null if not present.	$\Theta(1)$
<code>put(k ∈ Key, v ∈ Value) → Value</code>	Store $k \mapsto v$ in this map, and return old Value (or null if not present).	$\Theta(1)$
<code>remove(k ∈ Key)</code>	Delete $k \mapsto$ (whatever) in this map, and return old Value (or null if not present).	idk
<code>size()</code>	Return n .	$\Theta(1)$
<code>isEmpty()</code>	Return true iff $n = 0$.	$\Theta(1)$
<code>entrySet() → Set<Entry></code>	Return set of (key, value) pairs maintained.	idk
<code>keySet() → Set<Key></code>	Return set of keys maintained.	idk
<code>values() → Collection<Value></code>	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 $\text{Key} \rightarrow \mathbb{Z}$.

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

hash is a **hash function** iff it is the composition $\text{hash} = \text{cmp} \circ \text{hc} : \text{Key} \rightarrow \{0, \dots, N-1\}$ 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.

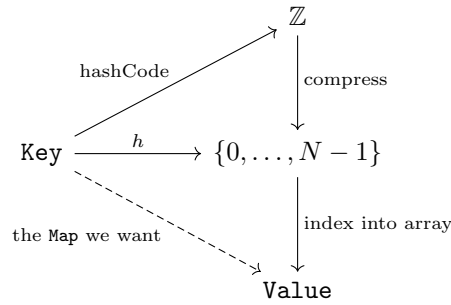
(**A**, **h**) is a **hash table** iff $A \in \text{Array}<\text{Container}<\text{Key}, \text{Value}>>$ and **h** is a hash function.

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



Goal for hash codes: try to inject $\text{Key} \hookrightarrow \mathbb{Z}$; i.e. try to reduce occurrences 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 \ll denotes cyclic bit-shift:

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

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

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

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

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

bitstring: $b_\alpha \cdots b_0 \mapsto (b_\alpha \cdots b_0 << s)$ 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 $\text{Keys} \xrightarrow{h} \{0, \dots, N-1\}$; i.e. try to reduce hash collisions.

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

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

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 lookups 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: $f(k, i) = (h(k) + i^2) \bmod N$

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

2.5.1 Definitions

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, Subgraph, Spanning subgraph)

Let $G = (V, E)$ be a graph.

A **path** (MATH2302: *walk*) P in G is an alternating sequence $P = (v_0, e_1, v_1, \dots, e_n, v_n)$ with each $v_i \in V$ and each $e_i \in E$ an edge $v_{i-1} \rightarrow v_i$. 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 $C = (v_0, e_1, v_1, \dots, e_n, v_n)$ in which each v_i for $i \in \{1, \dots, n\}$ 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.

A graph H is a **subgraph** of G iff $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$; that is, H consists only of vertices and edges found in G .

A subgraph H of G is a **spanning subgraph** iff $V(H) = V(G)$; that is, H consists of *every* vertex found in G .

A **spanning tree** is a spanning subgraph which is also a tree.

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

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

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. $e \approx \mathcal{O}(v^2)$).

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 $V \rightarrow \{1, \dots, |V|\}$ that assigns to each vertex a unique label $1, \dots, |V|$, and which is subject to the constraint that

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

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

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 (free, thin) category generated by G into any totally ordered set. The archetypal totally ordered set, when G is finite, is $\{1, \dots, \text{card}(V)\}$.

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 25 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. 

Definition 2.5.8 (Weight, Distance)

Let $G = (V, E)$ be a graph and let $w : E \rightarrow \mathbb{R}$ be a function. We say that w is a *weight* function, and we call (G, w) a *weighted graph*.

A subgraph H of G has weight $w(H) := \sum_{e \text{ an edge of } H} (w(e))$, and a path $P = (v_0, e_1, \dots, e_n, v_n)$ has weight $w(P) = \sum_{i=1}^n (w(e_i))$.

In a weighted graph (G, w) , the *distance* $d(u, v)$ between any two vertices is

$$d(u, v) := \begin{cases} +\infty & \text{if } u \text{ and } v \text{ are disconnected in } G \\ -\infty & \text{if } u \text{ and } v \text{ are connected in } G \text{ via a negative cycle} \\ \min_{P \text{ path from } u \text{ to } v \text{ in } G} \{w(P)\} & \text{else} \end{cases}$$

Note that the distance may fail to be a real number if either u and v are disconnected, or if there is a negative cycle in the connected component of u and v .

2.5.2 Graph ADT

ADT 2.5.9 (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 **Graphs** 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

Method	Function
<code>Graph.addEdge($u \in \text{Vertex}, v \in \text{Vertex}$)</code> $\rightarrow \text{Edge}$	Return an edge $u \rightarrow v$, or null if no such edge exists
<code>Graph.endVertices($e \in \text{Edge}$)</code> $\rightarrow \text{Array}<\text{Vertex}>$	Return an array (or pair) $[u, v]$ of length 2 such that $u \xrightarrow{e} v$
<code>Graph.opposite($u \in \text{Vertex}, e \in \text{Edge}$)</code>	Return the vertex v such that $u \xrightarrow{e} v$ in this graph, or null if no such v exists.
<code>Graph.outDegree($v \in \text{Vertex}$)</code>	Returns number of outgoing edges from v
<code>Graph.outgoingEdges($v \in \text{Vertex}$)</code> $\rightarrow \text{Iteration}<\text{Vertex}>$	Returns an iteration of the outgoing edges from v
<code>Graph.inDegree($v \in \text{Vertex}$)</code>	Returns number of incoming edges to v
<code>Graph.incomingEdges($v \in \text{Vertex}$)</code> $\rightarrow \text{Iteration}<\text{Vertex}>$	Returns an iteration of the incoming edges to v
<code>Graph.insertVertex($x \in \text{VData}$)</code> $\rightarrow \text{Vertex}$	Maintain a new Vertex in this Graph to store data x , and return this vertex
<code>Graph.insertEdge($u \in \text{Vertex}, v \in \text{Vertex}, x \in \text{EData}$)</code> $\rightarrow \text{Edge}$	Maintain a new Edge $u \rightarrow v$ in this Graph which stores data x , and return this edge
<code>Graph.removeVertex($v \in \text{Vertex}$)</code> $\rightarrow \text{void}$	Remove v and all edges incident with v from this graph
<code>Graph.removeEdge($e \in \text{Edge}$)</code> $\rightarrow \text{void}$	Remove e from this graph
<code>Vertex.element()</code>	Return data stored here
<code>Edge.element()</code>	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 2.5.24)	ALGraph (DS 2.5.25)	AMGraph (DS 2.5.26)
Memory usage (all)	$V + E$	$V + E$	V^2
<code>outgoingEdges($v \in \text{Vertex}$)</code>	E	$\deg(v)$	V
<code>incomingEdges($v \in \text{Vertex}$)</code>	E	$\deg(v)$	V
<code>getEdge($u, v \in \text{Vertex}$)</code>	E	$\min\{\deg(u), \deg(v)\}$	1
<code>insertVertex(x)</code>	1	1	V^2
<code>insertEdge(u, v, x)</code>	1	1	1
<code>removeVertex(v)</code>	E	$\deg(v)$	V^2
<code>removeEdge(e)</code>	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:

- * Dijkstra (algo 22)
- * Single source shortest path via topological sort (algo 23)
- * Strong connectivity solver (algo 24)
- * Topological sort (algo 25)
- * Cycle detection via DFS (algo 26)
- * Prim-Jarnik (algo 27)
- * Kruskal (algo 28)

2.5.3 Graph problems

See figure 1 for a graph of graph problems seen in this course.

The [Wikipedia page for shortest path problems](#) is an invaluable reference here. Donate to Wikipedia if it is financially responsible for you to do so.

Problem 2.5.10 (Single pair reachability)

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


Solution.

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

Problem 2.5.11 (Single pair shortest path)

Let $G = (V, E, w)$ be a weighted graph and $u, v \in V$ be vertices. **Determine** whether a shortest path $u \rightsquigarrow v$ exists, and **find** such a shortest path if it exists.


Solution.

Specialise any appropriate solution to *single source shortest path* (problem 2.5.13). 

Problem 2.5.12 (Single source reachability)

Let $G = (V, E)$ be a graph and $u \in V$ be a vertex. **Determine** the set of vertices v such that there is a path $u \rightsquigarrow v$.

Solution.

Your favourite graph traversal. E.g. BFS (algo 21) or DFS (algo 20). 

Problem 2.5.13 (Single source shortest path)

Let $G = (V, E, w)$ be a weighted graph and $u \in V$ be a vertex. **Determine** the set of vertices v such that there is a shortest path $u \rightsquigarrow v$, and **find** a shortest path $u \rightsquigarrow v$ for each such v .

Solution.


- G undirected, $w(v) = 1$ for all v :
- G undirected, w non-negative: Dijkstra's algorithm (algo 22)
- G a DAG, $w(v) = 1$ for all v : A modification of a topological sorting algorithm (algo 23)
- G directed, w non-negative: A modification of Dijkstra's algorithm
- G directed, w arbitrary, no negative cycles: [Bellman-Ford](#) algorithm

all solve their respective problems. 

Problem 2.5.14 (All pairs reachability)

Let G be a graph. **Determine** the set of pairs of vertices (u, v) such that there is a path $u \rightsquigarrow v$.

Solution.

Have a source reachability solver (problem 2.5.12). Layer dynamic programming on top of an iteration of the solver for each vertex, so as to eliminate repeat computations. 

Problem 2.5.15 (All pairs shortest path)

Let G be a weighted graph. **Determine**, the set of pairs of vertices (u, v) such that there is a shortest path $u \rightsquigarrow v$, and **find** such a shortest path for each such pair of vertices.

Graph problems

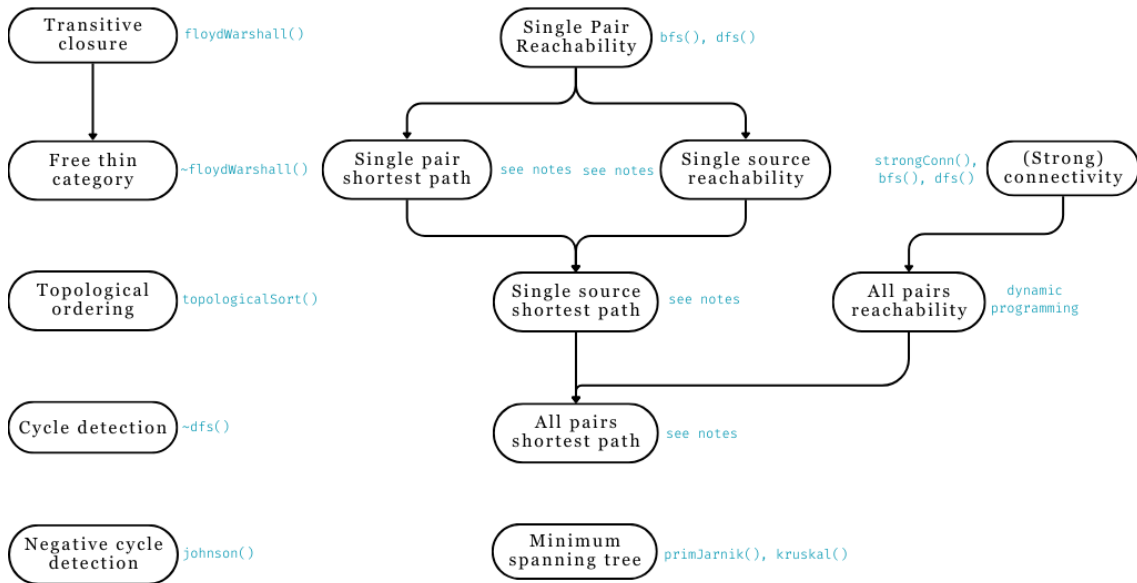


Figure 1: Graph of graph problems. There is an edge $p \rightarrow q$ iff solving q yields, with minimal modifications, a solution to p (i.e. p is an ‘easy subproblem’ of q). Taking the transitive closure of this graph yields a graph in which there is an edge $p \rightarrow q$ iff p is a proper subproblem of q . (For my own reference: [this is the URL to edit.](#))

Solution.


- G undirected, w non-negative: **Floyd-Warshall** algorithm
- G directed, w arbitrary, no negative cycles: **Floyd-Warshall** algorithm, or **Johnson's** algorithm

all solve their respective problems.

Problem 2.5.16 (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 2.5.17 (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 24.

Problem 2.5.18 (Transitive closure)

Let G be a graph with no parallel edges. **Find** the transitive closure G^* of G .

G^* consists of the same vertices of G , but an edge $u \xrightarrow{G^*} v$ iff there is a path $u \rightsquigarrow^G v$.

Solution.

The **Floyd-Warshall** algorithm does this.

Problem 2.5.19 (Free thin category)

Let G be a graph with no parallel edges. **Find** the free thin category generated by G .

The free thin category generated by G is the same as G^* , but with exactly one self-loop on every vertex.


Solution.

A trivial modification of the [Floyd-Warshall](#) algorithm does this. 

Problem 2.5.20 (Topological Sorting)

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

Solution.

Topological sort; algo [25](#). 

Problem 2.5.21 (Cycle detection)

Let $G = (V, E)$ be a graph and $u \in V$. **Determine** whether there is a cycle in the connected component of u .

Solution.

See algo [26](#). 

Problem 2.5.22 (Negative cycle detection)

Let $G = (V, E, w)$ be a weighted, directed graph and $u, v \in V$. **Determine** whether there is a negative cycle C and a path $u \rightsquigarrow C \rightsquigarrow v$ from u to v which passes through some vertex on the negative cycle.

A negative cycle C in G is a cycle in G with negative weight $w(C) < 0$.

Solution.


The [Bellman-Ford](#) algorithm does this. 

Problem 2.5.23 (Minimum spanning tree)

Let $G = (V, E, w)$ be a weighted, undirected graph. Suppose that G is connected. **Find** a minimum spanning tree of G .

A *minimum spanning tree* T of G is a spanning tree T whose weight $w(T)$ (as a subgraph of G) is minimised; i.e. for any other spanning tree T' of G , we have that $w(T) \leq w(T')$.

Solution.

The Prim-Jarnik algorithm (algo [27](#)) and Kruskal's algorithm (algo [28](#)) solve this. 

2.5.4 Graph ADT implementations

We now give some implementations of the Graph ADT.

Data Structure 2.5.24 (EdgeListGraph implements Graph (ADT [2.5.9](#)))

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

An implementation of a Graph (ADT [2.5.9](#)) by edge lists. `ELGraph` stores

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

`Vertex` stores

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

`Edge` stores

- `element` $\in \text{EData}$: data stored here
- `position` $\in \text{Positions}<\text{ELGraph.edges}>$: position in edge list
- `origin` $\in \text{Vertex}$: as in `origin` $\xrightarrow{\text{this Edge}}$ `destination`
- `destination` $\in \text{Vertex}$: as in `origin` $\xrightarrow{\text{this Edge}}$ `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:
 - * Dijkstra (algo 22)
 - * Single source shortest path via topological sort (algo 23)
 - * Strong connectivity solver (algo 24)
 - * Topological sort (algo 25)
 - * Cycle detection via DFS (algo 26)
 - * Prim-Jarnik (algo 27)
 - * Kruskal (algo 28)

Data Structure 2.5.25 (`AdjacencyListGraph` extends `EdgeListGraph` (DS 2.5.24))

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

An implementation of a `Graph` (ADT 2.5.9) 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:
 - * Dijkstra (algo 22)
 - * Single source shortest path via topological sort (algo 23)
 - * Strong connectivity solver (algo 24)
 - * Topological sort (algo 25)
 - * Cycle detection via DFS (algo 26)
 - * Prim-Jarnik (algo 27)
 - * Kruskal (algo 28)

Data Structure 2.5.26 (`AdjacencyMatrixGraph` extends `EdgeListGraph` (DS 2.5.24))

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

An implementation of a `Graph` (ADT 2.5.9) by an adjacency matrix. `AMGraph` augments to

- (inherit)

- `adjacencyMatrix` \in `Matrixsquare<Edge>`: matrix storing at entry (i, j) a `Edge` $u \rightarrow v$, 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:
 - * Dijkstra (algo 22)
 - * Single source shortest path via topological sort (algo 23)
 - * Strong connectivity solver (algo 24)
 - * Topological sort (algo 25)
 - * Cycle detection via DFS (algo 26)
 - * Prim-Jarnik (algo 27)
 - * Kruskal (algo 28)

3 Strings, Pattern Matching and Compression

Not much data structures stuff here, so I'm having trouble figuring out where to put this in the document. I think here's alright.

3.2 Pattern Matching

Definition 3.2.1 (String, Lexicographic order)

A **string** s over an alphabet \mathcal{A} is a finite sequence $s = s_0 \cdots s_{n-1}$ of characters. Its **length** $\text{len}(s)$ is n .

I might denote the set of all strings over \mathcal{A} by \mathcal{A}^* .

The **lexicographic order** on \mathcal{A}^* is given by:

- For individual characters $\mathbf{a}, \mathbf{b} \in \mathcal{A}$ with $\mathbf{a} \leq \mathbf{b}$, " \mathbf{a} " \leq " \mathbf{b} "
- For strings $s = s_0 \cdots s_{n-1}$ and $\tilde{s} = \tilde{s}_0 \cdots \tilde{s}_{n-1}$, the lexicographic comparison $s \leq \tilde{s}$ is done the same as for tuples.
- For strings $s = s_0 \cdots s_{n-1}$ and $\tilde{s} = s_0 \cdots s_{n-1} \cdots s_m$ where s is a (*proper*) *prefix* of \tilde{s} , we have $s < \tilde{s}$.

A **prefix**, **suffix** and **substring** are exactly what you think they are.

The big problem from lecture 10 was the following.

Problem 3.2.2 (Pattern matching)

Let $T, P \in \mathcal{A}^*$ be strings. **Determine** whether P is a substring of T .

Variations include **finding** all indices i in T where $P = T[i, \dots, i + \text{len}(P) - 1]$.

We call T the *text* and P the *pattern* which we wish to find.

Another important variation on this pattern has a *set* $S \subseteq \mathcal{A}^*$ of texts, and asks to **determine** whether a given pattern is one of the strings in S .

Solution.

Any of these algorithms may be appropriate:

- Brute force pattern matching
- Boyer-Moore (algo 29)
- Knuth-Morris-Pratt (algo 30)
- Suffix array binsearch

There are trade-offs to each of them which may be found in table 24.



Algo	Worst runtime without preprocessing	Worst preprocessing runtime	Pre-processing reusable on other patterns?
Brute force	$\Theta(\text{len}(T) \cdot \text{len}(P))$	$\mathcal{O}(0)$	no preprocessing lol
Boyer-Moore (algo 29)	$\Theta(\text{len}(T) \cdot \text{len}(P))$	$\mathcal{O}(\text{len}(P) + \mathcal{A})$	No
Knuth-Morris-Pratt (algo 30)	$\mathcal{O}(\text{len}(T) + \text{len}(P))$	$\mathcal{O}(\text{len}(T))$	No
Suffix array bin-search	$\mathcal{O}(\log(\text{len}(T)) \cdot \text{len}(P))$	$\mathcal{O}(\text{len}(T))$	Yes

Definition 3.2.3 (Last occurrence function)

Let $P \in \mathcal{A}^*$ be a string. The **last occurrence function** $L : \mathcal{A} \rightarrow \{-1, 0, \dots, \text{len}(P) - 1\}$ associated to P is the function with, for all characters $c \in \mathcal{A}$,

$$L(c) = \begin{cases} \max i \in \{0, \dots, \text{len}(P) - 1\} \text{ such that } P[i] = c & \text{if such } i \text{ exists} \\ -1 & \text{else} \end{cases}$$

This is central to the Boyer-Moore algorithm (algo 29).

This can be computed given P in linear time $\mathcal{O}(\text{len}(P))$.

Definition 3.2.4 (Failure function)

Let $P \in \mathcal{A}^*$ be a string. The **failure function** associated to P is the function $F : \{0, \dots, \text{len}(P) - 1\} \rightarrow \mathbb{Z}_{\geq 0}$ given by, for all $j \in \{0, \dots, \text{len}(P) - 1\}$,

$$F(j) = \max_{\substack{S \text{ prefix of } P[0, \dots, j] \\ S \text{ suffix of } P[1, \dots, j]}} \{\text{len}(S)\}$$

with $F(0) = 0$ (via $S = ""$ the empty string).

This is central to the Knuth-Morris-Pratt algorithm (algo 30).

This can be computed given P in linear time $\mathcal{O}(\text{len}(P))$.

Since the domain of F is the non-negative integers $0, \dots, \text{len}(P) - 1$, we may as well encode F as an array (indices are keys, data stored are values) of length $\text{len}(P)$.

Data Structure 3.2.5 (Trie)

Let $S \subseteq \mathcal{A}^*$ be a collection of strings over an alphabet \mathcal{A} . A **trie** T for S is a tree storing nothing at the root, and storing individual characters at each node (except for potentially some cringe leaves) with the property that

- For each string $s = s_0 \dots s_{n-1} \in S$, there is a unique path from the root of T to a leaf of T which passes through nodes storing the characters s_0, \dots, s_{n-1} in this order; and
- For any common prefix $p = p_0 \dots p_{k-1}$ of any two strings $s, s' \in S$, the paths $\text{root} \rightsquigarrow s$ and $\text{root} \rightsquigarrow s'$ travel through the same nodes for each of p_0, \dots, p_{k-1} (but may diverge after the prefix is exhausted).

A minor technicality is that if the path of a string $s = s_0 \dots s_{n-1} \in S$ ends at an internal node of T , then a new empty child of the node reached storing s_{n-1} is created to mark this.

For example, if $\mathcal{A} = \{\mathbf{a}, \dots, \mathbf{z}\}$ and

$$S = \{\text{"according"}, \text{"aviation"}, \text{"a"}, \text{"bee"}, \text{"fly"}\}$$

then figure 2 is a trie for S (where leaves should be drawn in square boxes, but I couldn't be bothered to do that. Likewise, figure 3 is a *compressed* trie for S , constructed by bringing up the leaves of any internal nodes with one child.

Regarding S as a 2D array of strings ($S[\text{whichString}][\text{whichCharacter}]$), i.e. (moving down increments the whichString index)

$$S = \begin{bmatrix} \mathbf{a} & \mathbf{c} & \mathbf{c} & \mathbf{o} & \mathbf{r} & \mathbf{d} & \mathbf{i} & \mathbf{n} & \mathbf{g} \\ \mathbf{a} & \mathbf{v} & \mathbf{i} & \mathbf{a} & \mathbf{t} & \mathbf{i} & \mathbf{o} & \mathbf{n} & \\ \mathbf{a} & & & & & & & & \\ \mathbf{b} & \mathbf{e} & \mathbf{e} & & & & & & \\ \mathbf{f} & \mathbf{l} & \mathbf{y} & & & & & & \end{bmatrix}$$

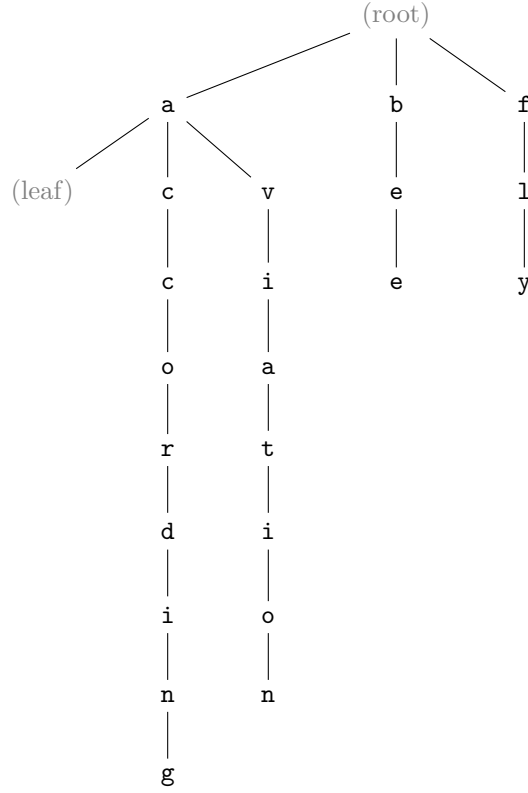


Figure 2: Trie for bee movie

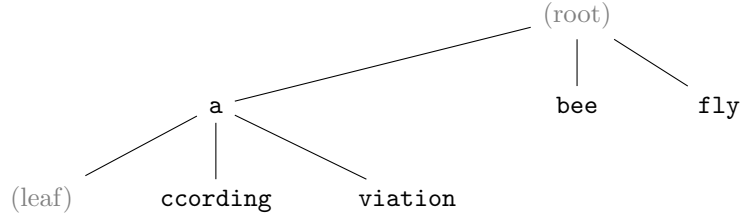


Figure 3: Compressed trie for bee movie

we can compress the compressed trie into its *compact representation*, given by replacing a string by a triple (whichString, startCharIndex, endCharIndex) (inclusive of endCharIndex). Figure 4 shows an example.

Tries support:

Function	Worst runtime
(Storage space)	$\Theta(S)$ for compact representation
Search for P	$\mathcal{O}(\mathcal{A} \cdot \text{len}(P))$
Insert, delete P	$\mathcal{O}(\mathcal{A} \cdot \text{len}(P))$ for a standard trie

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

Data Structure 3.2.6 (Suffix trie)

Let $T = t_0 \cdots t_{n-1} \$ \in \mathcal{A}^*$ be a string over an alphabet \mathcal{A} . A **suffix trie** for T is a trie for the set $\{t_0 \cdots t_{n-1} \$, t_1 \cdots t_{n-1} \$, \dots, t_{n-1} \$, \$\}$ of suffixes of T . Here, $\$$ is a special “end of string” character not found in \mathcal{A} , which has $\$ < c$ for any character $c \in \mathcal{A}$.

Pattern matches of patterns P in T can be computed by finding any suffix s of T which has P as a prefix of s ; i.e. it suffices to search the suffix trie to pattern match for patterns in T .

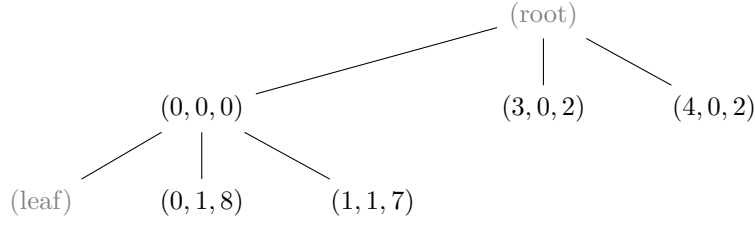


Figure 4: Compact representation of a compressed trie for bee movie

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

Data Structure 3.2.7 (Suffix array)

Let $T = t_0 \dots t_{n-1} \$ \in \mathcal{A}^*$ be a string over an alphabet \mathcal{A} . A **suffix array** for T is given by constructing the following array (the array is $[0, 1, \dots, n-1, n]$, with the suffixes at indices i in the array shown for clarity)

$$\begin{pmatrix} 0 : & t_0 t_1 \dots t_{n-1} \$ \\ 1 : & t_1 \dots t_{n-1} \$ \\ \vdots & \vdots \\ n-1 : & t_{n-1} \$ \\ n : & \$ \end{pmatrix}$$

and *sorting* the suffixes lexicographically, and keeping track of the indices

$$\begin{pmatrix} n : & \$ \\ \vdots & \vdots \\ 42 : & \text{some suffix lol} \$ \end{pmatrix}$$

produces the suffix array by looking at how the indices get shifted around (in this case, the suffix array is $[n, \dots, 42]$). For example, the suffix array of **aviation** is constructed by

$$\begin{pmatrix} 0 : & \text{aviation} \$ \\ 1 : & \text{viation} \$ \\ 2 : & \text{iation} \$ \\ 3 : & \text{ation} \$ \\ 4 : & \text{tion} \$ \\ 5 : & \text{ion} \$ \\ 6 : & \text{on} \$ \\ 7 : & \text{n} \$ \\ 8 : & \$ \end{pmatrix} \xrightarrow{\text{lexico. sort}} \begin{pmatrix} 8 : & \$ \\ 3 : & \text{ation} \$ \\ 0 : & \text{aviation} \$ \\ 2 : & \text{iation} \$ \\ 5 : & \text{ion} \$ \\ 7 : & \text{n} \$ \\ 6 : & \text{on} \$ \\ 4 : & \text{tion} \$ \\ 1 : & \text{viation} \$ \end{pmatrix}$$

so the suffix array is $[8, 3, 0, 2, 5, 7, 6, 4, 1]$.

Binary search on the suffix array, with comparator done by comparing the corresponding suffixes lexicographically, is the same as binary searching the suffixes of T for a given pattern. Thus, to search for a pattern P in T , we can simply “binary search the suffix array”.

Algorithms this data structure(s) may utilise:

3.3 Compression

Alright, I’m not making full notes on this because it’s a bit messy to write down technically (it’s far easier in pictures, and making pictures in L^AT_EX is a pain in the ass).

Definition 3.3.1 (Compression and Coding)

Given a frequency/probability model for how often characters occur in a text, a **coding** algorithm is a construction of a compression codec.

A **compression** algorithm is the application of a coding algorithm to a given frequency/probability model.

An *optimal prefix code* is found by Huffman's algorithm (algo [31](#)).
The Lempel-Ziv algorithm exists.

4 Algorithms

4.1 Sort

Definition 4.1.1 (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} .

4.1.1 Comparison sort

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

For a comparison of comparison sorts, see table 26.

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

Table 26: Comparison of comparison sorts

Theorem 4.1.2 (Runtime of comparison sorts)

Let \mathcal{A} be a comparison sort algorithm with input size n . Then, \mathcal{A} runs in $\Omega(n \log(n))$ time.

Algorithm 1: Selection Sort

```

1  /* This method is a stable sort.                                     */
2  /* Runtime complexity:  $\Theta(n^2)$                                      */
3  method selectionSort( $A \in \text{GeneralLinearStructure}, n \in \mathbb{Z}_{\geq 0}$ )  $\rightarrow \text{void}$ 
   |   Input   :  $A$  of length  $\leq n$ 
   |   Requires:  $A$  is totally ordered by  $\leq$ 
   |   Does    : In-place sorts  $A$ 
4  if  $n > 1$  then
5  |   maxIndex  $\leftarrow 0$ ;
6  |   for  $i \leftarrow 1$  to  $n - 1$  do
7  |       |   if  $A[i] > A[\text{maxIndex}]$  then
8  |       |       |   maxIndex  $\leftarrow i$ ;
9  |       |   // Swap max with last
10 |       swap( $A[\text{maxIndex}], A[n - 1]$ );
11 |       // Sort the rest
12 |       selectionSort( $A, n - 1$ );

```

Algorithm 2: Insertion Sort

```
1  /* This method is a stable sort. */
2  /* Runtime complexity:  $\Theta(n^2)$  */
3  method insertionSort( $A \in \text{GeneralLinearStructure}$ ,  $n \in \mathbb{Z}_{\geq 0}$ )  $\rightarrow void$ 
    Input    :  $A$  of length  $\leq n$ 
    Requires:  $A$  is totally ordered by  $\leq$ 
    Does     : In-place sorts  $A$ 
4  for  $i \leftarrow 1$  to  $n - 1$  do
5      valueToInsert  $\leftarrow A[i]$ ;
6       $j \leftarrow i - 1$ ;
7      // Find where to insert valueToInsert
8      while  $j \geq 0$  and  $A[j] > \text{valueToInsert}$  do
9          // Shift inputs upwards
10          $A[j + 1] \leftarrow A[j]$ ;
11          $j \leftarrow j - 1$ ;
12     //  $j$  is the index of the first value  $\leq \text{valueToInsert}$ 
13      $A[j + 1] \leftarrow \text{valueToInsert}$ ;
```

Algorithm 3: Merge Sort

```
1  /* This method is a stable sort. */
2  /* Runtime complexity:  $\Theta(n \log(n))$  */
3  method mergeSort( $A \in \text{GeneralLinearStructure}$ ,  $l, r \in \mathbb{Z}_{\geq 0}$ )  $\rightarrow void$ 
    Input    :  $A$  of length  $> r$ 
    Does     : Destructively sort  $A[\{l, \dots, r\}]$ 
4  if  $l < r$  then
5       $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$ ;
6      mergeSort( $A, l, m$ ) ; // Divide
7      mergeSort( $A, m + 1, r$ ) ; // Divide
8      merge( $A, l, m, r$ ) ; // Conquer
9  method merge( $A \in \text{GeneralLinearStructure}$ ,  $l, m, r \in \mathbb{Z}_{\geq 0}$ )  $\rightarrow void$ 
    Input    : Structure  $A$ , left index  $l$ , middle index  $m$ , right index  $r$ 
    Requires:  $A$  has length  $> r + 1$  and  $0 \leq l \leq m \leq r$ 
    Does     : Replace  $A$  by the sorted union of  $A[\{l, \dots, m - 1\}]$  and  $A[\{m, \dots, r - 1\}]$ 
10  Llength  $\leftarrow m - l + 1$ ;
11  Rlength  $\leftarrow r - m$ ;
12   $L \leftarrow A[\{l, \dots, m - 1\}]$ ;
13   $R \leftarrow A[\{m, \dots, r - 1\}]$ ;
14  Aind  $\leftarrow l$ ;
15  Lind  $\leftarrow 0$ ;
16  Rind  $\leftarrow 0$ ;
17  // Merge
18  while Lind < Llength and Rind < Rlength do
19      if  $L[\text{Lind}] \leq R[\text{Rind}]$  then
20           $A[\text{Aind}++] \leftarrow L[\text{Lind}++]$ ;
21      else
22           $A[\text{Aind}++] \leftarrow R[\text{Rind}++]$ ;
23  // Copy leftovers. At most one of  $L, R$  is non-empty
24  while Lind < Llength do
25       $A[\text{Aind}++] \leftarrow L[\text{Lind}++]$ ;
26  while Rind < Rlength do
27       $A[\text{Aind}++] \leftarrow R[\text{Rind}++]$ ;
```

Algorithm 4: Quick Sort

```
1  /* Worst-case runtime complexity:  $\mathcal{O}(n^2)$  */
2  /* Average-case, best-case runtime complexity:  $\Theta(n \log(n))$  */
3  method quickSort( $A \in \text{GeneralLinearStructure}$ ,  $n \in \mathbb{Z}_{>0}$ )
     $\rightarrow \text{GeneralLinearStructure}$ 
    Input    : Structure  $A$  of length  $\leq n$ 
    Returns : Destructively sorted copy of  $A$ 

4  if  $n = 1$  then
5      | return  $A$ ;
6      // Else...
7      // Divide
8       $p \leftarrow$  pivot index chosen from  $\{0, \dots, n-1\}$ ;           // often randomly chosen
9       $(L, E, G, l, e, g) \leftarrow \text{partition}(A, p)$ ;
10     // Recurse
11      $L \leftarrow \text{quickSort}(L, l)$ ;
12      $G \leftarrow \text{quickSort}(G, g)$ ;
13     // Conquer
14     return  $L.\text{appendAll}(E).\text{appendAll}(G)$ ;           // appendAll() does the obvious

15 method partition( $A \in \text{GeneralLinearStructure}$ ,  $p \in \mathbb{Z}_{\geq 0}$ )
     $\rightarrow \text{GeneralLinearStructure}^3 \times \mathbb{Z}_{\geq 0}^3$ 
    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  $a \in A$  with  $a < A[p]$ 
        •  $E$  contains all things  $a \in A$  with  $a = A[p]$ 
        •  $G$  contains all things  $a \in A$  with  $a > A[p]$ 
        •  $l, e, g$  are the lengths of  $L, E, G$  respectively
        • the order in  $A$  is maintained in  $L, E, G$ 

16  $L, E, G \leftarrow$  empty sequences of capacity  $\text{length}(A)$ ;
17  $l, e, g \leftarrow 0$ ;
18  $\text{pivot} \leftarrow A.\text{remove}(p)$ ;
19 while  $A$  is not empty do
20     |  $\text{element} \leftarrow A.\text{remove}(A.\text{first}())$ ;
21     | if  $\text{element} < \text{pivot}$  then
22     |     |  $L.\text{add}(\text{element})$ ;
23     |     |  $l \leftarrow l + 1$ ;
24     | else if  $\text{element} = \text{pivot}$  then
25     |     |  $E.\text{add}(\text{element})$ ;
26     |     |  $e \leftarrow e + 1$ ;
27     | else
28     |     |  $G.\text{add}(\text{element})$ ;
29     |     |  $g \leftarrow g + 1$ ;
30 return  $(L, E, G, l, e, g)$ ;
```

Algorithm 5: Heap Sort

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

4.1.2 Non-comparison sort

Algorithm 6: Bucket Sort

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

Algorithm 7: Lexicographic Sort

```
1  /* 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  $A$ .
   */
2  method lexicographicSort( $A \in \text{GeneralLinearStructure}$ ,  $d \in \mathbb{Z}_{>0}$ )  $\rightarrow \text{void}$ 
   |   Input   : Structure  $A$  of  $d$ -tuples
   |   Requires: The data type  $D_i$  of the  $i$ -th component is totally ordered by  $\leq$ , for each
   |                $i \in \{1, \dots, d\}$ 
   |   Does    : Destructively sort  $A$  according to lexicographic order
3  for  $i \leftarrow d$  downto 1 do
4  |   stableSort( $A$ ,  $i$ -th component);           // keys are the  $i$ -th component
```

Algorithm 8: Radix Sort

```
1  /* Specialisation of lexicographicSort() which uses bucketSort() and
   /* applies only to non-negative integers.                                     */
2  /* Runtime complexity  $\mathcal{O}(d \cdot (n + N))$                                */
3  method radixSort( $A \in \text{GeneralLinearStructure}$ ,  $d \in \mathbb{Z}_{>0}$ ,  $n \in \mathbb{Z}_{\geq 0}$ ,  $N \in \mathbb{Z}_{>0}$ )
    $\rightarrow \text{void}$ 
   |   Input   : Structure  $A$  of  $d$ -tuples in  $\{0, \dots, N - 1\}^d$  of length  $n$ 
   |   Does    : Destructively sort  $A$ 
4  for  $i \leftarrow d$  downto 1 do
5  |   bucketSort( $A$ ,  $n$ ,  $N$ ,  $i$ -th component);           // keys are the  $i$ -th component
```

Algorithm 9: Binary Radix Sort

```
1  /* Specialisation of radixSort() which works in binary.                   */
2  /* Runtime complexity  $\mathcal{O}(b \cdot n)$                                      */
3  method binaryRadixSort( $A \in \text{GeneralLinearStructure}$ ,  $n \in \mathbb{Z}_{\geq 0}$ ,  $b \in \mathbb{Z}_{>0}$ )  $\rightarrow \text{void}$ 
   |   Input   : Structure  $A$  of length  $n$  storing  $b$ -bit non-negative integers
   |   Does    : Destructively sort  $A$ 
4  for  $i \leftarrow 0$  to  $b - 1$  do
5  |   bucketSort( $A$ ,  $n$ , 2,  $i$ -th bit);           // keys are the  $i$ -th bit
```

4.2 Heap Methods

Algorithm 10: Upheap

```
1 method upheap( $H \in \text{Heap}$ ,  $z \in \text{Node}$ )  $\rightarrow \text{void}$ 
   |   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
2   while  $z \neq H.\text{root}$  and  $z.\text{parent}.\text{data} > z.\text{data}$  do
3   |   swapData( $z$ ,  $z.\text{parent}$ );
4   |    $z \leftarrow z.\text{parent}$ ;
```

Algorithm 11: Downheap

```
1 method downheap( $H \in \text{Heap}$ )  $\rightarrow \text{void}$ 
   |   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
2   while  $z.\text{hasChildren}()$  and ( $z.\text{data} > m.\text{data}$  for some child  $m$  of  $z$ ) do
3   |    $m \leftarrow z.\text{left}$  or  $z.\text{right}$ , whichever has smallest data;
4   |   swapData( $z$ ,  $m$ );
5   |    $z \leftarrow m$ ;
```

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

```
1 method build( $X \in \text{Collection}$ ,  $n \in \mathbb{Z}_{\geq 0}$ )  $\rightarrow \text{void}$ 
   Input    : Collection  $X$  of size  $n$  containing items to store in a new heap
   Requires: The elements of  $X$  are totally ordered by  $\leq$ 
   Returns  : New heap storing  $X$ 

2 if  $n = 1$  then
3   return new Heap with only  $X.\text{remove}()$  at the root;

4 // These can be found using a simple brute force algorithm
5  $(f, e) \leftarrow f, i \in \mathbb{Z}_{\geq 0}$  such that  $n = 2^0 + \dots + 2^e + f$  and  $0 < f \leq 2^{e+1}$ ;
6 heaps, done  $\leftarrow$  new empty queues of capacity  $2^{e+1}$ ;
7 // Build lowest level ("floor") of heap
8 for  $dc \leftarrow 1$  to  $f$  do
9   heaps.enqueue(new Heap.build( $X.\text{remove}()$ ));
10 // Each non-floor level
11 for  $i \leftarrow e$  downto 0 do
12   // Construct level  $i$ 
13   for  $dc \leftarrow 1$  to  $2^i$  do
14     heap  $\leftarrow$  new Heap.build( $X.\text{remove}()$ );
15     // Merge two lower heaps with this heap
16     if not heaps.isEmpty() then
17       heap.root.setLeft(heaps.dequeue().root);
18     if not heaps.isEmpty() then
19       heap.root.setRight(heaps.dequeue().root);
20     heap.downheap(); // Make this a heap
21     done.enqueue(heap);
22   // Register level  $i$ 
23   while not done.isEmpty() do
24     heaps.enqueue(done.dequeue());
25 // Return only heap
26 return heaps.dequeue();
```

Algorithm 13: `Heap.insert()`

```
1 method insert( $H \in \text{Heap}$ ,  $x \in \text{Data}$ )  $\rightarrow \text{void}$ 
   Input    : This heap  $H$ , and data  $x$  to insert
   Does     : Store  $x$  in this heap

2 insertHere  $\leftarrow$  node in  $H$  to insert into; // see array-based implementation
3 insertHere.setData( $x$ );
4  $H.\text{upheap}()$ ; // Fix the heap
```

Algorithm 14: `Heap.removeMin()`

```
1 method removeMin( $H \in \text{Heap}$ )  $\rightarrow \text{void}$ 
   Input    : This heap  $H$ 
   Does     : (Assuming  $H$  implements a PriorityQueue) remove the min data in  $H$ 
   Returns  : Return the min element from  $H$ 

2  $w \leftarrow H.\text{last}$ ;
3 swapData( $w$ ,  $H.\text{root}$ );
4 returnMe  $\leftarrow w.\text{getData}()$ ;
5  $H.\text{remove}(w)$ ; //  $w.\text{getParent}().\text{remove}(w)$ , unless  $w$  is the root
6  $H.\text{downheap}()$ ; // Fix the heap
```

4.3 Search

Algorithm 15: Binary Search

```
1 method binsearch( $T \in \text{BinSearchTree}\langle \text{Key}, \text{Value} \rangle, k \in \text{Key} \rangle \rightarrow \text{Node}\langle \text{Key}, \text{Value} \rangle$ 
   Input    : Binary search tree  $T$  to search,
               Key  $k$  to search for
   Returns  :  $\text{Node}\langle \text{Key}, \text{Value} \rangle$  with key  $k$  stored in  $T$ , if it exists, or null otherwise.
2   return binSearcher ( $T.\text{getRoot}(), k$ );
3 method binSearcher( $\text{cursor} \in \text{Node}\langle \text{Key}, \text{Value} \rangle, k \in \text{Key} \rangle \rightarrow \text{Node}\langle \text{Key}, \text{Value} \rangle$ 
   Input    : Node cursor currently being searched from,
               Key  $k$  to search for
   Returns  :  $\text{Node}\langle \text{Key}, \text{Value} \rangle$  with key  $k$  stored in the subtree rooted at the cursor, if
               it exists, or null otherwise.
4   // If we hit a leaf, we're done searching
5   if cursor = null then
6   |   return null;
7   // Obvious
8   if cursor.getKey() =  $k$  then
9   |   return cursor;
10  // Recurse on children
11  if  $k < \text{cursor.getKey}()$  then
12  |   return binSearcher (cursor.getLeft(),  $k$ );
13  else
14  |   return binSearcher (cursor.getRight(),  $k$ );
```

4.4 Binary Search Tree Methods

Algorithm 16: BinSearchTree.remove()

```
1 method remove( $T \in \text{BinSearchTree}\langle \text{Key}, \text{Value} \rangle, k \in \text{Key}$ )  $\rightarrow \text{void}$ 
   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
2   killMe  $\leftarrow$  find( $k, T.\text{getRoot}()$ ); // Return a node in  $T$  with key  $k$  if exists,
   else null
3   if killMe = null then
4   |   return null;
5   //  $k$  was in the tree somewhere
6   if killMe has 2 leaf children then
7   |   Replace killMe by a leaf in killMe.getParent();
8   else if killMe has exactly 1 leaf child then
9   |   Replace killMe by the non-leaf child of killMe in killMe.getParent();
10  else // killMe has no leaf children
11  |   replacement  $\leftarrow$  getReplacement(killMe);
12  |   killMe.setData(replacement.getData()); /* Replace the key-value pair in
   |   killMe by that in replacement */
13  |   replace replacement by a leaf in replacement.getParent();
14 method getReplacement( $\text{killMe} \in \text{Node}\langle \text{Key}, \text{Value} \rangle$ )  $\rightarrow \text{void}$ 
   Input    : Node killMe to get replacement for
   Returns  : The node whose key-value pair should be placed in killMe
15  returnMe  $\leftarrow$  killMe.getRight();
16  while returnMe.getLeft() is not a leaf do
17  |   returnMe  $\leftarrow$  returnMe.getLeft();
18  return returnMe;
```

4.5 AVL Tree Methods

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

```
1 method insert( $T \in \text{AVLTree}\langle \text{Key}, \text{Value} \rangle$ ,  $k \in \text{key}$ ,  $v \in \text{Value}$ )  $\rightarrow \text{void}$ 
   Input   : This AVL tree  $T$ , and new data  $(k, v)$  to insert
   Does    : Inserts  $(k, v)$  into this tree
2   newNode  $\leftarrow$  super.insert( $k, v$ ); /* Insertion from BinSearchTree. Return new
   node created/node overridden */
3   rebalance( $T$ , newNode);
4 method remove( $T \in \text{AVLTree}$ ,  $k \in \text{Key}$ )  $\rightarrow \text{void}$ 
   Input   : Key  $k$  to remove in tree  $T$ 
   Does    : Exactly what you expect it to do
5   parent  $\leftarrow$  super.remove( $k$ ); /* Removal from BinSearchTree. Return parent of
   removed node, or null if nothing was removed. */
6   if parent = null then
7     return;
8   rebalance( $T$ , parent);
```

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

```
1 method rebalance( $T \in \text{AVLTree}\langle \text{Key}, \text{Value} \rangle$ ,  $\text{newNode} \in \text{Node}\langle \text{Key}, \text{Value} \rangle$ )  $\rightarrow \text{void}$ 
   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
2    $z \leftarrow \text{newNode}$ ;
3   while  $|\text{height}(z.\text{getLeft}()) - \text{height}(z.\text{getRight}())| \leq 1$  do
4     if not  $z.\text{hasParent}()$  then
5       return; // Hit root without needing to rebalance
6      $z \leftarrow z.\text{getParent}()$ ;
7   triNodeRestructuring( $T$ ,  $z$ );
8 method triNodeRestructuring( $T \in \text{AVLTree}\langle \text{Key}, \text{Value} \rangle$ ,  $z \in \text{Node}\langle \text{Key}, \text{Value} \rangle$ )
    $\rightarrow \text{void}$ 
   Input   : Tree  $T$  to restructure in, and node  $z$  to restructure from
   Does    : Restructure at  $z$ 
9    $y \leftarrow$  child of  $z$  with largest height;
10   $x \leftarrow$  child of  $y$  with largest height;
11   $(a, b, c) \leftarrow$  in-order listing of  $x, y, z$ ; // i.e.  $a \leq b \leq c$ , with  $\{a, b, c\} = \{x, y, z\}$ 
12  /* 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 :) */
13  if  $(a, b, c) = (z, y, x)$  then
14    // Single rotation
15    Make  $b$  the root of the subtree from  $z$ , preserving in-order structure of  $T$ ;
16  else if  $a, b, c = (x, y, z)$  then // symmetric to previous case
17    // Single rotation
18    Make  $b$  the root of the subtree from  $z$ , preserving in-order structure of  $T$ ;
19  else if  $(a, b, c) = (z, x, y)$  then
20    // Double rotation
21    Make  $b$  the root of the subtree from  $c$ , preserving in-order structure of  $T$ ;
22    Make  $b$  the root of the subtree from  $z$ , preserving in-order structure of  $T$ ;
23  else // symmetric to previous case
24    // Double rotation
25    Make  $b$  the root of the subtree from  $a$ , preserving in-order structure of  $T$ ;
26    Make  $b$  the root of the subtree from  $z$ , preserving in-order structure of  $T$ ;
```

4.6 Splay Tree Methods

Algorithm 19: `SplayTree.splay()`

```
1 method splay( $T \in \text{SplayTree}\langle \text{Key}, \text{Value} \rangle, x \in \text{Node}\langle \text{Key}, \text{Value} \rangle$ )  $\rightarrow \text{void}$ 
   Input    : Node  $x$  to splay in tree  $T$ 
   Does     : Splays  $x$  to the root of  $T$ 

2   // This pseudocode isn't very good, because I can't insert visual
   // diagrams! Refer to lecture slides or your own personal notes :)
3   // 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.

4   if  $x = T.\text{getRoot}()$  then
5   |   return;
6    $y \leftarrow x.\text{getParent}();$ 
7   if  $y = T.\text{getRoot}()$  then
8   |   if  $x$  is the left child of  $y$  then
9   |   |    $\text{zag } x, y;$ 
10  |   |    $\text{splay}(T, x);$ 
11  |   else //  $x$  is the right child of  $y$ 
12  |   |    $\text{zig } x, y;$ 
13  |   |    $\text{splay}(T, x);$ 
14   $z \leftarrow y.\text{getParent}();$ 
15  if  $x$  left  $y$  left  $z$  then //  $x$  left child of  $y$  and  $y$  left child of  $z$ 
16  |    $\text{zag-zag } x, y, z;$ 
17  |    $\text{splay}(T, x);$ 
18  else if  $x$  left  $y$  right  $z$  then
19  |    $\text{zag-zig } x, y, z;$ 
20  |    $\text{splay}(T, x);$ 
21  else if  $x$  right  $y$  left  $z$  then
22  |    $\text{zig-zag } x, y, z;$ 
23  |    $\text{splay}(T, x);$ 
24  else //  $x$  right  $y$  right  $z$ 
25  |    $\text{zig-zig } x, y, z;$ 
26  |    $\text{splay}(T, x);$ 
```

4.7 Graph Algorithms

4.7.1 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 (algo 26)

Algorithm 20: Depth-first search

```

1 method dfs( $G \in \text{Graph}$ )  $\rightarrow void$ 
   | Input    : A graph  $G$ 
   | Does    : Visits all vertices of  $G$ , and marks edges as either discovery edges or back
   |           edges
   | Runtime:  $\Theta(|V(G)| + |E(G)|)$  worst
2   for  $v \in G.\text{vertices}()$  do
3   |   Mark  $v$  as unvisited; // Could be implemented using a Set<Vertex>
4   |   for  $e \in G.\text{edges}()$  do
5   |   |   Mark  $e$  as unexplored; /* Could be implemented using a Map<Edge, ENUM>
5   |   |   |   where ENUM consists of "unexplored", "back" and "discovery"      */
6   |   for  $v \in G.\text{vertices}()$  do
7   |   |   if  $v$  is not marked as visited then
8   |   |   |   dfs( $G, v$ );
9 method dfs( $G \in \text{Graph}, v \in \text{Vertex}$ )  $\rightarrow void$ 
   | 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
10  |   Mark  $v$  as visited;
11  |   for  $e \in G.\text{outgoingEdges}(v)$  do
12  |   |    $w \leftarrow G.\text{opposite}(v, e)$ ;
13  |   |   if  $e$  is marked as unexplored then
14  |   |   |   Mark  $e$  as a discovery edge (for  $w$ );
15  |   |   |   dfs( $G, w$ ); /* Could be done using a Stack<Vertex> instead,
15  |   |   |   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

```
1 method bfs( $G \in \text{Graph}$ )  $\rightarrow \text{void}$ 
   Input    : A graph  $G$ 
   Does     : Visits all vertices of  $G$ , and marks edges as either discovery edges or cross
              edges
   Runtime:  $\Theta(|V(G)| + |E(G)|)$  worst
2   for  $v \in G.\text{vertices}()$  do
3     Mark  $v$  as unvisited; // Could be implemented using a Set<Vertex>
4   for  $e \in G.\text{edges}()$  do
5     Mark  $e$  as unexplored; /* Could be implemented using a Map<Edge, ENUM>
7     where ENUM consists of "unexplored", "cross" and "discovery" */
6   for  $v \in G.\text{vertices}()$  do
7     if  $v$  is not marked as visited then
8       bfs( $G, v$ );
9 method bfs( $G \in \text{Graph}, \text{seed} \in \text{Vertex}$ )  $\rightarrow \text{void}$ 
   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
10  traversal  $\leftarrow$  new empty Queue<Vertex>;
11  traversal.enqueue(seed);
12  while not traversal.isEmpty() do /* short-circuit this if searching for
   minimum-edge paths */
13     $v \leftarrow$  traversal.dequeue();
14    Mark  $v$  as visited;
15    for  $e \in G.\text{incidentEdges}(v)$  do
16      if  $e$  is marked as unexplored then /* technically need to pre-mark as
   unexplored... idc. */
17         $w \leftarrow G.\text{opposite}(v, e)$ ;
18        if  $w$  is not marked as visited then /* technically need to pre-mark
   as unvisited... idc. */
19          Mark  $e$  as a discovery edge (for  $w$ );
20          traversal.enqueue( $w$ );
21        else
22          Mark  $e$  as a cross edge;
```

4.7.2 Graph Problem Solvers

Algorithm 22: Dijkstra's algorithm (problem 2.5.13 solver)

```
1 method dijkstra( $G \in \text{Graph}, s \in \text{Vertex}$ )  $\rightarrow \text{void}$ 
   Input : A vertex  $s$  in a weighted graph  $G$ 
   Requires:  $G$  has non-negative weights
   Returns :  $\text{distances} \in \text{Map}\langle \text{Vertex}, \mathbb{R}_{\geq 0} \rangle$  which assigns to each vertex  $v$  in the
               connected component of  $s$ , the distance  $d(v, s)$ , and to each vertex  $v$  not in
               the connected component of  $s$ , a sentinel  $\infty$  representing that  $s \not\leftrightarrow v$ .

2   // At a high level...
3    $\text{distances} \leftarrow$  new empty  $\text{Map}\langle \text{Vertex}, \mathbb{R}_{\geq 0} \rangle$ ;
4   for  $v \in G.\text{vertices}()$  do
5       if  $v = s$  then
6            $\text{distances.put}(v, 0)$ ;
7       else
8            $\text{distances.put}(v, \infty)$ ;
9        $\text{travMain.insert}(\text{distances.get}(v), v)$ ; // (key, value)
10  while not every vertex has been marked as seen do
11       $v \leftarrow$  any vertex  $v$  not yet marked as seen, which has a minimum  $\text{distances.get}(v)$ 
          value among such vertices;
12      Mark  $v$  as seen;
13      for  $e \in G.\text{outgoingEdges}(v)$  do
14          // Relax edge  $e$ 
15           $z \leftarrow G.\text{opposite}(v, e)$ ;
16           $\text{distances.put}(z, \min\{\text{distances.get}(z), \text{distances.get}(v) + e.\text{getWeight}()\})$ ;
17  return  $\text{distances}$ ;

18 method dijkstra( $G \in \text{Graph}, s \in \text{Vertex}$ )  $\rightarrow \text{void}$ 
   Input : A vertex  $s$  in a weighted graph  $G$ 
   Requires:  $G$  has non-negative weights
   Returns :  $\text{distances} \in \text{Map}\langle \text{Vertex}, \mathbb{R}_{\geq 0} \rangle$  which assigns to each vertex  $v$  in the
               connected component of  $s$ , the distance  $d(v, s)$ , and to each vertex  $v$  not in
               the connected component of  $s$ , a sentinel  $\infty$  representing that  $s \not\leftrightarrow v$ .
   Runtime:  $\Theta((|V(G)| + |E(G)|) \cdot \log(|V(G)|))$  worst
19  // An implementation of this algorithm...
20   $\text{travMain} \leftarrow$  new empty  $\text{AdaptablePriorityQueue}\langle \text{Vertex} \rangle$ ;
21   $\text{distances} \leftarrow$  new empty  $\text{Map}\langle \text{Vertex}, \mathbb{R}_{\geq 0} \rangle$ ;
22  for  $v \in G.\text{vertices}()$  do
23      if  $v = s$  then
24           $\text{distances.put}(v, 0)$ ;
25      else
26           $\text{distances.put}(v, \infty)$ ;
27       $\text{travMain.insert}(\text{distances.get}(v), v)$ ; // (key, value)
28  while not  $\text{travMain.isEmpty}()$  do
29       $u \leftarrow \text{travMain.removeMin}()$ ;
30      for  $e \in G.\text{outgoingEdges}(u)$  do
31          // Relax edge  $e$ 
32           $z \leftarrow G.\text{opposite}$  maybeShorter  $\leftarrow \text{distances.get}(u) + e.\text{getWeight}()$ ;
33          if maybeShorter  $< \text{distances.get}(z)$  then
34               $\text{distances.put}(z, \text{maybeShorter})$ ;
35               $\text{travMain.replaceKey}(z, \text{maybeShorter})$ ;
36  return  $\text{distances}$ ;
```

Algorithm 23: Topological sort, adapted to single-source shortest path (problem 2.5.13 solver)

```

1 method dagDistances( $G \in \text{Graph}$ ,  $s \in \text{Vertex}$ )  $\rightarrow \text{void}$ 
    Input : A vertex  $s$  in a weighted graph  $G$ 
    Requires:  $G$  is a DAG (but may have negative weights)
    Returns:  $\text{distances} \in \text{Map}\langle \text{Vertex}, \mathbb{R} \rangle$  which assigns to each vertex  $v$  in the
                connected component of  $s$ , the distance  $d(v, s)$ , and to each vertex  $v$  not in
                the connected component of  $s$ , a sentinel  $\infty$  representing that  $s \not\rightarrow v$ .
2    // At a high level...
3     $\text{distances} \leftarrow \text{new empty Map}\langle \text{Vertex}, (-\infty, \infty] \rangle$ ;
4    for  $v \in G.\text{vertices}()$  do
5        if  $v = s$  then
6             $\text{distances.put}(v, 0)$ ;
7        else
8             $\text{distances.put}(v, \infty)$ ;
9     $\tau \leftarrow \text{topologicalSort}(G, s)$ ; // start from  $s$ , topological sort on the
        connected component from  $s$ 
10    $\tau^{-1} \leftarrow \tau^{-1}$ ; // inverse map
11   for  $i \leftarrow 1$  to  $n$  do
12        $u \leftarrow \tau^{-1}.\text{get}(i)$ ; // This amounts to visiting the nodes in their
           topological order
13       for  $e \in G.\text{outgoingEdges}(u)$  do
14           // Relax  $e$ 
15            $z \leftarrow G.\text{opposite}(u, e)$ ;
16            $\text{maybeShorter} \leftarrow \text{distances.get}(u) + e.\text{getWeight}()$ ;
17           if  $\text{maybeShorter} < \text{distances.get}(z)$  then
18                $\text{distances.put}(z, \text{maybeShorter})$ ;
19   return  $\text{distances}$ ;

```

Algorithm 24: Strong connectivity (problem 2.5.17) solver

```

1 method strongConnectivity( $G \in \text{Graph}$ )  $\rightarrow \text{boolean}$ 
    Input : A graph  $G$ 
    Returns:  $\text{true}$  iff  $G$  is strongly connected
    Runtime:  $\Theta(|V(G)| + |E(G)|)$  worst, assuming computation of  $G^{\text{op}}$  takes
                 $\mathcal{O}(|V(G)| + |E(G)|)$  worst-case time
2    if  $G.\text{vertices}().\text{isEmpty}()$  then
3        return true;
4     $\text{seed} \leftarrow \text{any element of } G.\text{vertices}()$ ;
5     $\text{seedReachesEverything} \leftarrow \text{dfsVisitsAll}(G, \text{seed})$ ; /* returns true iff a
        depth-first search (algo 20) visits every vertex of  $G$ ; could be any
        traversal */
6    if not  $\text{seedReachesEverything}$  then
7        return false;
8     $G^{\text{op}} \leftarrow G$  with all edges reversed;
9     $\text{seedReachableByEverything} \leftarrow \text{dfsVisitsAll}(G^{\text{op}}, \text{seed})$ ;
10   return  $\text{seedReachesEverything} \wedge \text{seedReachableByEverything}$ ;

```

Algorithm 25: Topological sort (problem 2.5.17 solver)

```
1  /* The high-level procedure */
2  method topologicalSort( $G \in \text{Graph}$ )  $\rightarrow \text{Map}\langle \text{Vertex}, \{1, \dots, n\} \rangle$ 
    Input   : A directed graph  $G$  of size  $n = \|V(G)\|$ 
    Requires:  $G$  is a DAG
              $G$  is connected
    Returns : A topological ordering of  $G$ 
3  Sorter  $\leftarrow$  a (deep) copy of  $G$ ;
4   $n \leftarrow$  a reference to an integer storing  $G.\text{numVertices}()$ ; /* I will refer to the
    reference as " $n$ " and the value as " $N$ " */
5   $\tau \leftarrow$  new empty  $\text{Map}\langle \text{Vertex}, \{1, \dots, N\} \rangle$ ;
6  while not Sorter.isEmpty() do
7       $v \leftarrow$  any vertex of Sorter with no outgoing edges; // can be found using dfs
8       $\tau.\text{put}(v, N)$ ;
9       $N \leftarrow N - 1$ ;
10     Sorter.remove( $v$ );
11 return  $\tau$ ;

12 /* Implementation of topologicalSort() by depth-first search (algo 20) */
13 method topSortDFS( $G \in \text{Graph}$ )  $\rightarrow \text{Map}\langle \text{Vertex}, \{1, \dots, n\} \rangle$ 
    Input   : A directed graph  $G$ 
    Requires:  $G$  is a DAG
    Returns : A topological ordering of  $G$ 
    Runtime:  $\Theta(|V(G)| + |E(G)|)$  worst; same as dfs()
14  $n \leftarrow$  a reference to an integer storing  $G.\text{numVertices}()$ ; /* I will refer to the
    reference as " $n$ " and the value as " $N$ " */
15 for  $v \in G.\text{vertices}()$  do
16     Mark  $v$  as unvisited;
17  $\tau \leftarrow$  new empty  $\text{Map}\langle \text{Vertex}, \{1, \dots, N\} \rangle$ ;
18 for  $v \in G.\text{vertices}()$  do
19     if  $v$  is not marked as visited then
20         topSortDFS( $G, v, \tau, n$ );
21 return  $\tau$ ;

22 /* Helper for topSortDFS(Graph) */
23 method topSortDFS( $G \in \text{Graph}, \text{seed} \in \text{Vertex}, \tau \in \text{Map}\langle \text{Vertex}, \mathbb{Z}_{>0} \rangle,$ 
     $n \in \text{Reference}\langle \mathbb{Z}_{>0} \rangle \rightarrow \text{void}$ 
    Input   : A directed graph  $G$ 
    Requires:  $G$  is a DAG
    Does    : Updates  $\tau$  and  $n$  as per the component of vertices of  $G$  reachable from seed
24 Mark  $v$  as visited;
25 for  $e \in G.\text{outgoingEdges}(v)$  do
26      $w \leftarrow G.\text{opposite}(v, e)$ ;
27     if  $w$  is not marked as visited then
28         //  $e$  is a discovery edge for  $w$ 
29         topSortDFS( $G, w, \tau, n$ );
30     // else,  $e$  is a forward or cross edge
31  $\tau.\text{put}(v, N)$ ;
32  $N \leftarrow N - 1$ ;
```

Algorithm 26: Cycle detection via depth-first search (I'm pretty sure the code they gave us in the lectures, which I've more or less just copied here, is **wrong**. This should fail any triangle graph.)

```

1 method cycleDFS( $G \in \text{Graph}, v \in \text{Vertex}$ )  $\rightarrow \text{void}$ 
    Input : A vertex  $v$  in a graph  $G$ 
    Returns : true iff there is a cycle in  $G$  containing  $v$ 
    Runtime:  $\Theta(|V(G)| + |E(G)|)$  worst
2   cycleDFS( $G, v, \text{new empty Stack}$ );

3 method cycleDFS( $G \in \text{Graph}, v \in \text{Vertex}, \text{cycle} \in \text{Stack} \langle \text{Vertex} \cup \text{Edge} \rangle$ )  $\rightarrow \text{void}$ 
    Input : A vertex  $v$  in a graph  $G$ , and a stack which tracks a potential cycle from  $v$ 
    Does : If there is a cycle in  $G$  containing  $v$ , then upon termination of the
           algorithm, this cycle will be stored in the cycle stack
    Returns : true iff there is a cycle in  $G$  containing  $v$ 

4   Mark  $v$  as visited;
5   cycle.push( $v$ );
6   for  $e \in G.\text{outgoingEdges}(v)$  do
7       if  $e$  is marked as unexplored then
8            $w \leftarrow G.\text{opposite}(v, e)$ ;
9           if  $w$  is not marked as visited then
10              Mark  $e$  as a discovery edge (for  $w$ );
11              cycle.push( $e$ );
12              if cycleDFS( $G, w, \text{cycle}$ ) then
13                  return true;
14              cycle.pop();
15           else
16              Remove prefix of cycle up to the first occurrence of  $w$ ;      // Not 100% sure
17              what this means tbh
18              return true;
19   cycle.pop();
20   return false;

```

Algorithm 27: Prim-Jarnik algorithm (problem 2.5.23 solver)

```
1 method primJarnik( $G \in \text{Graph}$ )  $\rightarrow \text{void}$ 
   Input   : An undirected weighted graph  $G$ 
   Requires:  $G$  is connected
   Returns : A minimum spanning tree of  $G$ 
2   // At a high level...
3    $T \leftarrow$  any single-vertex subgraph of  $G$ ;
4   while  $T.\text{numVertices}() \neq G.\text{numEdges}() - 1$  do
5        $e \leftarrow$  any edge that connects  $T$  to a vertex outside  $T$  which minimises the weight
           among such vertices;
6        $T \leftarrow T$ , with  $e$  and its endpoints added;
7   return  $T$ ;

8 method primJarnik( $G \in \text{Graph}$ )  $\rightarrow \text{void}$ 
   Input   : An undirected weighted graph  $G$ 
   Requires:  $G$  is connected
   Returns : A minimum spanning tree of  $G$ 
   Runtime:  $\mathcal{O}(|V(G)| + |E(G)| \cdot \log(|V(G)|))$  worst
9   // An implementation of this algorithm...
10   $s \leftarrow$  any vertex of  $G$ ;
11  distances  $\leftarrow$  new empty Map<Vertex,  $(-\infty, \infty]$ >;
12  for  $v \in G.\text{vertices}()$  do
13      if  $v = s$  then
14          distances.put( $v$ , 0);
15      else
16          distances.put( $v$ ,  $\infty$ );
17   $T \leftarrow$  new empty Graph;
18  travMain  $\leftarrow$  new AdaptablePriorityQueue< $(-\infty, \infty]$ , Vertex  $\times$  Edge>;
19  // Set up traversal maintainer
20  for  $v \in G.\text{vertices}()$  do
21      travMain.put(distances.get( $v$ ), ( $v$ , null));
22  // Traverse
23  while not travMain.isEmpty() do
24       $(u, e) \leftarrow$  travMain.removeMin();
25       $T.\text{addVertex}(u)$ ;
26       $T.\text{addEdge}(e)$ ;
27      for  $e' \in G.\text{outgoingEdges}(u)$  such that  $e'$  is in travMain do      /* Check with a
           set or smth idk */
28          // "Relax" edge  $e'$ 
29          if  $e'.\text{getWeight}() < \text{distances.get}(v)$  then
30              distances.put( $v$ ,  $e'.\text{getWeight}()$ );
31              /* Looking up whatever was here in these lines can be done
                 with a Map */
32              travMain.updateKey(( $v$ , whatever was here), distances.get( $v$ ));
33              travMain.updateValue(( $v$ , whatever was here), ( $v$ ,  $e'$ ));
34  return  $T$ ;
```

Algorithm 28: Kruskal's algorithm (problem 2.5.23 solver)

```
1 method kruskal( $G \in \text{Graph}$ )  $\rightarrow \text{void}$ 
   Input    : An undirected weighted graph  $G$ 
   Requires:  $G$  is connected
   Returns : A minimum spanning tree of  $G$ 
2   // At a high level...
3    $E \leftarrow$  new empty Set<Edge>;
4   while  $E.\text{cardinality}() \neq G.\text{numVertices}() - 1$  do
5        $e \leftarrow$  any edge  $e$  of  $G$  which isn't already in  $E$  such that adding  $e$  to  $E$  doesn't
           produce any cycles in  $E$ , and  $e$  is minimum weight among such choices;
6        $E \leftarrow E \cup \{e\}$ ;
7   return The subgraph of  $G$  with edges  $E$  and vertices all vertices of  $G$ ;

8 method kruskal( $G \in \text{Graph}$ )  $\rightarrow \text{void}$ 
   Input    : An undirected weighted graph  $G$ 
   Requires:  $G$  is connected
   Returns : A minimum spanning tree of  $G$ 
   Runtime: idk lol
9   // An implementation of this algorithm...
10  clusters  $\leftarrow$  new empty UnionFind<Vertex>; // UnionFind extends Set
11  // Populate clusters with singletons
12  for  $v \in G.\text{vertices}()$  do
13      clusters.get( $v$ )  $\leftarrow$  new Set with only  $v$  as an element;
14  travMain  $\leftarrow$  new empty PriorityQueue;
15  for  $e \in G.\text{edges}()$  do
16      travMain.put( $e.\text{getWeight}()$ ,  $e$ );
17   $T \leftarrow$  new empty Graph;
18  while  $T.\text{numEdges}() < G.\text{numVertices}() - 1$  do
19       $e \leftarrow$  travMain.removeMin();
20       $(u, v) \leftarrow e.\text{getEndpoints}()$ ;
21      if clusters.get( $u$ )  $\neq$  clusters.get( $v$ ) then
22           $T.\text{addVertex}(u)$ ;
23           $T.\text{addVertex}(v)$ ;
24           $T.\text{addEdge}(e)$ ;
25          Merge clusters.get( $u$ ) and clusters.get( $v$ ) into their union;
26  return  $T$ ;
```

4.8 Strings, Pattern Matching and Compression

Algorithm 29: Boyer-Moore (problem 3.2.2 solver)

```

1 method boyerMoore( $T \in \text{String}\langle \mathcal{A} \rangle, P \in \text{String}\langle \mathcal{A} \rangle, \mathcal{A} \in \text{Alphabet}$ )
   $\rightarrow \mathbb{Z}_{\geq 0} \cup \{\text{NO MATCH}\}$ 
  Input : A pattern  $P$  to find in a text  $T$ , both of which are strings over an alphabet  $\mathcal{A}$ 
  Returns : The smallest  $i \in \mathbb{Z}_{\geq 0}$  such that  $P = T[i, \dots, i + \text{len}(P) - 1]$ , or NO MATCH if there is no such  $i$ 
  Runtime:  $\mathcal{O}(\text{len}(P) \cdot \text{len}(T) + |\mathcal{A}|)$  worst
2  $L \leftarrow \text{lastOccuranceFunction}(P, \mathcal{A});$ 
3  $t \leftarrow \text{len}(P) - 1;$ 
4  $p \leftarrow \text{len}(P) - 1;$ 
5 do
6   if  $T[t] = P[p]$  then // this character matches
7     if  $p = 0$  then // whole pattern matches
8       return  $i$ ; // match at index  $i$ 
9     else
10       $t \leftarrow t - 1;$ 
11       $p \leftarrow p - 1;$ 
12   else
13     // Character jump
14      $t \leftarrow t + \text{len}(P) - \min\{p, 1 + L.\text{get}(T[i])\};$ 
15      $p \leftarrow \text{len}(P) - 1;$ 
16 while  $t < \text{len}(T);$ 
17 return NO MATCH;

```

Algorithm 30: Knuth-Mogus-Pratt (problem 3.2.2 solver)

```
1 method knuthMorrisPratt( $T \in \text{String}\langle\mathcal{A}\rangle, P \in \text{String}\langle\mathcal{A}\rangle$ )  $\rightarrow \mathbb{Z}_{\geq 0} \cup \{\text{NO MATCH}\}$ 
   Input   : A pattern  $P$  to find in a text  $T$ , both of which are strings over an alphabet  $\mathcal{A}$ 
   Returns : The smallest  $i \in \mathbb{Z}_{\geq 0}$  such that  $P = T[i, \dots, i + \text{len}(P) - 1]$ , or NO MATCH if there is no such  $i$ 
   Runtime:  $\Theta(\text{len}(T) + \text{len}(P))$  worst

2  $F \leftarrow \text{failureFunction}(P);$ 
3  $t \leftarrow 0;$ 
4  $p \leftarrow 0;$ 
5 while  $t < \text{len}(T)$  do
6   if  $T[t] = P[p]$  then // this character matches
7     if  $p = \text{len}(P) - 1$  then // matched whole string
8       return  $t - p;$ 
9     else
10      // Keep looking...
11       $t \leftarrow t + 1;$ 
12       $p \leftarrow p + 1;$ 
13   else
14     // Mismatch...
15     if  $p > 0$  then
16       // Shift over pattern as much as possible
17        $p \leftarrow F[p - 1];$ 
18     else
19       // Try next character of target
20        $t \leftarrow t + 1;$ 
21 return NO MATCH
```

Algorithm 31: Huffman

```
1 method huffman( $X \in \text{String}\langle\mathcal{A}\rangle, \mathcal{A} \in \text{Alphabet}$ )  $\rightarrow \mathbb{Z}_{\geq 0} \cup \{\text{NO MATCH}\}$ 
   Input   : A string  $X$  over an alphabet  $\mathcal{A}$ 
   Returns :  $T \in \text{BinTree}\langle\mathcal{A} \cup \{\text{NULL}\}\rangle$  such that regarding  $T$  as an encoding trie produces an optimal prefix code
   Runtime:  $\mathcal{O}(\text{len}(X) + d \log(d))$  worst, where  $d$  is the number of distinct characters in  $X$ 

2  $\text{freq} \leftarrow \text{Map}\langle\mathcal{A}, \mathbb{Z}_{\geq 0}\rangle$  which associates to each character  $c$  the number of times  $c$  occurs in  $X$ ;
3  $\text{huffer} \leftarrow \text{new empty PriorityQueue}\langle\text{BinTree}\langle\mathcal{A} \cup \{\text{NULL}\}\rangle\rangle;$ 
4 for  $a \in \mathcal{A}$  do
5    $\text{huffer.insert}(\text{freq.get}(a), \text{new BinTree}\langle\mathcal{A} \cup \{\text{NULL}\}\rangle \text{ with a single node storing } a);$ 
6 while  $\text{huffer.getSize()} > 1$  do
7    $(f_1, T_1) \leftarrow \text{huffer.removeMin}();$ 
8    $(f_2, T_2) \leftarrow \text{huffer.removeMin}();$ 
9    $\text{huffer.insert}(f_1 + f_2, \text{new BinTree}\langle\mathcal{A} \cup \{\text{NULL}\}\rangle \text{ with an empty root node, left subtree } T_1 \text{ and right subtree } T_2);$ 
10  $(dc, T) \leftarrow \text{huffer.removeMin}();$ 
11 return  $T;$ 
```
