COMP3506 Algos and Datas Summary

Gabriel Field

17/08/2023 - END OF COURSE

Contents

1 Boilerplate

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

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

Each algorithm gives a pseudocode representation.

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

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

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

Some common abbreviations:

- "amo.": amortised

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

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

Changelog:

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

(2023-08-27 19:13) Content up to the end of week 5 lectures is now summarised here.
(2023-08-25 16:08) Content up to the end of week 4 lectures is now summarised here.
(2023-08-17 12:00) Started this project.

2 Data Structures

2.1 General Linear Structures

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

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

Algorithms this data structure(s) may utilise:

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

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

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

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

Algorithms this data structure(s) may utilise:

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

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

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

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

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

Algorithms this data structure(s) may utilise:

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

2.1.1 Children of `StaticSequence`

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

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

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

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

Algorithms this data structure(s) may utilise:

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

2.1.2 Children of `DynamicSequence`

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

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

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

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

This data structure has the following variants:

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

Algorithms this data structure(s) may utilise:

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

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

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

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

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

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

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

Algorithms this data structure(s) may utilise:

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

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

2.2 Stacks and Queues

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

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

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

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

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

Implementation strategies:

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

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

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

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

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

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

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

Implementation strategies:

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

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

2.2.1 Priority Queues

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

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

`PriorityQueue` stores

- `size`: n

`Entry` stores

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

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

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

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

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

Implementation strategies:

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

Runtime comparison (depending on implementation):

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

Table 8: Comparison of runtime based on implementation

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

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

Associated classes:

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

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

`AdaptablePriorityQueue` stores

- `size`: n

`Position` stores

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

`Entry` stores

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

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

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

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

*see table 10 for a comparison of runtime depending on implementation.
Implementation strategies:

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

Runtime comparison (depending on implementation):

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

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

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

2.3 Trees

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

A dynamic-size hierarchical structure of n nodes (with arbitrarily many children). The tree stores:

- `size` $\in \mathbb{Z}_{\geq 0}$: n
- `height` $\in \mathbb{Z}_{\geq 0}$: height of the tree
- `root` $\in$ `Node`: pointer to the root of the tree (maybe an index in an array)

The nodes (of type `Node`) store:

- `parent` $\in$ `Node`: pointer to the parent of this `Node`
- `data`: data stored at this node
- `children` $\in$ `Set<Node>`: set of children

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

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

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

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

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

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

ADT 2.3.2 (`BinTree` extends `Tree` (ADT 2.3.1))
Associated classes: `BinTree` $=$ `BinTree<Data>`, `Node` $=$ `Node<Data>`, `Data`.

A 2-ary tree. The tree stores the same as in `Tree`. The nodes store:

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

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

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

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

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

Definition (proper tree, full tree): $T \in$ `BinTree` is *proper* (aka *full*) iff every level of T is full.
Algorithms this data structure(s) may utilise: (none yet)

ADT 2.3.3 (`Heap` extends `BinTree` (ADT 2.3.2))
Associated classes: `Heap` $=$ `Heap<Data>`, `Node` $=$ `Node<Data>`, `Data`.

We describe a min-heap here. A max-heap is similar, but the internal sorting is the opposite.
A dynamic-size structure which stores *totally ordered* elements.
Class invariants: $H \in$ `BinTree` is a (min-)*heap* iff

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

`Heap` stores

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

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

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

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

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

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

Note that:

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

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

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

Algorithms this data structure(s) may utilise:

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

2.4 Sets, Maps and Hashing

ADT 2.4.1 (`Set`)

Associated classes: `Set` = `Set<Data>`, `Data`.

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

- `size`: n

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

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

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

Variants:

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

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

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

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

- `size`: n

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

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

Note: Keys must be unique.
Implementation strategies:

- Hash tables
- Unsorted list
- Sorted list

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

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

Definition 2.4.3 (Hash function)
Let $N \in \mathbb{Z}_{>0}$ be a positive integer, and let hc, cmp and hash be functions.
hc is a **hash code** function iff it is a function Key $\to \mathbb{Z}$.
cmp is a **compression function** iff it is a function $\mathbb{Z} \to \{0, \dots, N-1\}$.
hash is a **hash function** iff it is the composition $\text{hash} = \text{cmp} \circ \text{hc} : \texttt{Key} \to \{0, \dots, N-1\}$ 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 $A \in$ Array<Container<Key, Value>> and h is a hash function.

Notes 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 : Key $\to \{0, \dots, N-1\}$ 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

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

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

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

$$
\begin{aligned}
\text{ComponentSum} : \texttt{Keys} &\longrightarrow \mathbb{Z} \\
\text{components:} \quad (b_\alpha, \dots, b_0) &\longmapsto \sum_{i=0}^{\alpha} (\text{hc}\,(b_i)) \\
\text{PolynomialAccumulation} : \texttt{Keys} &\longrightarrow \mathbb{Z} \\
\text{bitstring:} \quad b_\alpha \cdots b_0 &\longmapsto \sum_{i=0}^{\alpha} \left(b_i \cdot z^i\right) \\
\text{CyclicShift} : \texttt{Keys} &\longrightarrow \mathbb{Z}
\end{aligned}
$$

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

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

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

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

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

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

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

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

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

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

Linear: $f(k, i) = (h(k) + i) \bmod N$

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

Double hashing: $f(k, i) = (h(k) + id(k)) \bmod N$, where d is another hash function

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

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

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

Acceptable load factors:

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

See also:

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

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

3 Algorithms

3.1 Sort

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

3.1.1 Comparison sort

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

For a comparison of comparison sorts, see table 16.

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

Table 16: Comparison of comparison sorts

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

Algorithm 1: Selection Sort

/* This method is a stable sort. */

/* Runtime complexity: $\Theta\left(n^2\right)$ */

method selectionSort($A \in$ *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

if $n > 1$ **then**

  maxIndex $\leftarrow 0$;

  for $i \leftarrow 1$ **to** $n - 1$ **do**

    if $A[i] > A[\text{maxIndex}]$ **then**

      maxIndex $\leftarrow 0$;

  // Swap max with last

  swap(A[maxIndex], $A[n-1]$);

  // Sort the rest

  selectionSort(A, $n-1$);

Algorithm 2: Insertion Sort

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

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

Algorithm 3: Merge Sort

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

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

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

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

Algorithm 4: Quick Sort

```
/* Worst-case runtime complexity:  O(n^2)                              */
/* Average-case, best-case runtime complexity:  Θ(n log(n))            */
method quickSort(A ∈ GeneralLinearStructure, n ∈ ℤ>0)
  → GeneralLinearStructure
    Input    : Structure A of length ≤ n
    Returns  : Destructively sorted copy of A

    if n = 1 then
        return A;
    // Else...
    // Divide
    p ← pivot index chosen from {0, ..., n − 1};        // often randomly chosen
    (L, E, G, l, e, g) ← partition(A, p);
    // Recurse
    L ← quickSort(L, l);
    G ← quickSort(G, g);
    // Conquer
    return L.appendAll(E).appendAll(G);        // appendAll() does the obvious

method partition(A ∈ GeneralLinearStructure, p ∈ ℤ≥0)
  → GeneralLinearStructure^3 × ℤ^3≥0
    Input    : Structure A of length > p,
               p index in A of pivot A[p]
    Returns  : (L, E, G, l, e, g) where:
        • L contains all things a ∈ A with a < A[p]
        • E contains all things a ∈ A with a = A[p]
        • G contains all things a ∈ A with a > A[p]
        • l, e, g are the lengths of L, E, G respectively
        • the order in A is maintained in L, E, G

    L, E, G, ← empty sequences of capacity length(A);
    l, e, g ← 0;
    pivot ← A.remove(p);
    while A is not empty do
        element ← A.remove(A.first());
        if element < pivot then
            L.add(element);
            l ← l + 1;
        else if element = pivot then
            E.add(element);
            e ← e + 1;
        else
            G.add(element);
            g ← g + 1;
    return (L, E, G, l, e, g);
```

Algorithm 5: Heap Sort

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

3.1.2 Non-comparison sort

Algorithm 6: Bucket Sort

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

Algorithm 7: Lexicographic Sort

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

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

Input : Structure A of d-tuples

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

Does : Destructively sort A according to lexicographic order

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

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

Algorithm 8: Radix Sort

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

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

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

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

Does : Destructively sort A

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

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

Algorithm 9: Binary Radix Sort

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

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

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

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

Does : Destructively sort A

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

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

3.2 Heap Methods

Algorithm 10: Upheap

method `upheap`($H \in$ *Heap*, $z \in$ *Node*) $\rightarrow$ *void*

Input : "Heap" H and node z to upheap (H may not technically be a heap at this point, but the point of this method is to fix that)

Does : Assuming H was not a heap because only the node z violates the heap property, fix H so that it is a heap again

```
while z ≠ H.root and z.parent.data > z.data do
    swapData(z, z.parent);
    z ← z.parent;
```

Algorithm 11: Downheap

method `downheap`($H \in$ *Heap*) $\rightarrow$ *void*

Input : "Heap" H and (H may not technically be a heap at this point, but the point of this method is to fix that)

Does : Assuming H was not a heap because only the (children of) the root violates the heap property, fix H so that it is a heap again

```
while z.hasChildren() and (z.data > m.data for some child m of z) do
    m ← z.left or z.right, whichever has smallest data;
    swapData(z, m);
    z ← m;
```

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

```
method build(X ∈ Collection, n ∈ ℤ≥0) → void
    Input    : Collection X of size n containing items to store in a new heap
    Requires: The elements of X are totally ordered by ≤
    Returns : New heap storing X
    if n = 1 then
        return new Heap with only X.remove() at the root;
    // These can be found using a simple brute force algorithm
    (f, e) ← f, i ∈ ℤ≥0 such that n = 2^0 + ··· + 2^e + f and 0 < f ≤ 2^(e+1);
    heaps, done ← new empty queues of capacity 2^(e+1);
    // Build lowest level (‘‘floor") of heap
    for dc ← 1 to f do
        heaps.enqueue(new Heap.build(X.remove()));
    // Each non-floor level
    for i ← e downto 0 do
        // Construct level i
        for dc ← 1 to 2^i do
            heap ← new Heap.build(X.remove());
            // Merge two lower heaps with this heap
            if not heaps.isEmpty() then
                heap.root.setLeft(heaps.dequeue().root);
            if not heaps.isEmpty() then
                heap.root.setRight(heaps.dequeue().root);
            heap.downheap();                              // Make this a heap
            done.enqueue(heap);
        // Register level i
        while not done.isEmpty() do
            heaps.enqueue(done.dequeue());
    // Return only heap
    return heaps.dequeue();
```

Algorithm 13: `Heap.insert()`

```
method insert(H ∈ Heap, x ∈ Data) → void
    Input    : This heap H, and data x to insert
    Does     : Store x in this heap
    insertHere ← node in H to insert into;      // see array-based implementation
    insertHere.setData(x);
    H.upheap();                                              // Fix the heap
```

Algorithm 14: `Heap.removeMin()`

```
method removeMin(H ∈ Heap) → void
    Input    : This heap H
    Does     : (Assuming H implements a PriorityQueue) remove the min data in H
    Returns : Return the min element from H
    w ← H.last;
    swapData(w, H.root);
    returnMe ← w.getData();
    H.remove(w); // w.getParent().remove(w), unless w is the root
    H.downheap();                                            // Fix the heap
```