

# COMP3506 Algos and Datas Summary

Gabriel Field

17/08/2023 - END OF COURSE

## Contents

<b>1</b>	<b>Boilerplate</b>	<b>1</b>
<b>2</b>	<b>Data Structures</b>	<b>3</b>
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
<b>3</b>	<b>Algorithms</b>	<b>18</b>
3.1	Sort	18
3.1.1	Comparison sort	18
3.1.2	Non-comparison sort	21
3.2	Heap Methods	23
3.3	Search	25
3.4	Binary Search Tree Methods	25
3.5	AVL Tree Methods	26
3.6	Splay Tree Methods	28

## 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) → D` to mean “`method()` returns type `D`”. For example, `foo(x ∈ X) → 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

The  $\text{\LaTeX}$  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:**

**(2023-09-09 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 ∈ {0, ..., n - 1})</code>	Return $x_i$
<code>set(i ∈ {0, ..., 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(<math>i \in \{0, \dots, n-1\}</math>)</code>	Return $x_i$
<code>set(<math>i \in \{0, \dots, n-1\}, x</math>)</code>	Set $x_i$ to $x$
<code>add(<math>x</math>)</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(<math>X</math>)</code>	Create new data structure to store $X$	idk lol
<code>len()</code>	Return $n$	$\Theta(1)$
<code>get(<math>i \in \{0, \dots, n-1\}</math>)</code>	Return $x_i$	$\Theta(1)$
<code>set(<math>i \in \{0, \dots, n-1\}, x</math>)</code>	Set $x_i$ to $x$	$\Theta(1)$
<code>iterate(<math>f</math> 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 <b>head</b> or the <b>tail</b> )	$\Theta(1)$
<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(<math>i \in \{0, \dots, n-1\}, x</math>)</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(<math>x \in \text{Data}</math>) \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 <b>top</b>	$\Theta(1)$
<code>peek()</code> aka <code>top() \rightarrow \text{Data}</code>	Return the element at the <b>top</b>	$\Theta(1)$
<code>isEmpty() \rightarrow \text{boolean}</code>	Return <b>true</b> 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 <b>back</b> 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 <b>front</b>	$\Theta(1)$
<code>first() → Data</code>	Return the element at the <b>front</b>	$\Theta(1)$
<code>isEmpty() → boolean</code>	Return <b>true</b> 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 <b>Entry</b> with smallest <b>key</b> , 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`

Runtime comparison (depending on implementation):

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



Method	Function	Runtime complexity (worst)
APQ. <a href="#">replaceValue</a> ( $e \in \text{Entry}$ , $v \in \text{Value}$ )	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. <a href="#">remove</a> ()	$\Theta(1)$	$\Theta(1)$	$\mathcal{O}(\log(n))$
APQ. <a href="#">replaceKey</a> ()	$\Theta(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log(n))$
APQ. <a href="#">replaceValue</a> ()	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

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.<a href="#">build</a>(<math>X</math>)</code>	Create new data structure to store $X$	depends on implementation
<code>Tree.<a href="#">size</a>()</code>	Return $n$	$\Theta(1)$
<code>Tree.<a href="#">isEmpty</a>()</code>	Return <code>true</code> iff $n = 0$	$\Theta(1)$
<code>Tree.<a href="#">root</a>()</code>	Return <code>root</code>	$\Theta(1)$
<code>Tree.<a href="#">iterator</a>()</code>	Return an iterator for this tree	depends on implementation
<code>Tree.<a href="#">positions</a>()</code>	Not in Joel's headcanon	depends on implementation
<code>Node.<a href="#">parent</a>()</code>	Return <code>this.parent</code>	$\Theta(1)$
<code>Node.<a href="#">children</a>()</code>	Return <code>this.children</code>	$\Theta(1)$
<code>Node.<a href="#">numChildren</a>()</code>	Return <code>this.children.size()</code>	$\Theta(1)$
<code>Node.<a href="#">isInternal</a>()</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 <b>true</b> iff this node is external; i.e. it is a leaf	$\Theta(1)$
<code>Node.isRoot()</code>	Return <b>true</b> 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(<math>x \in \text{Node}</math>, <math>y \in \text{Node}</math>)</code>	Replace $x$ with $y$	$\Theta(1)$
<code>Tree.addRoot(<math>x \in \text{Node}</math>)</code>	Set the root of this <b>Tree</b> to $x$ , and the old root to one of $x$ 's children	$\Theta(1)$
<code>Tree.remove(<math>x \in \text{Node}</math>)</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`  $\in$  `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 <b>Tree</b> )	(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(<math>x \in \text{Data}</math>)</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 are all have  $\mathcal{O}(\text{height of tree})$  runtime complexity.

**Algorithms** this data structure(s) may utilise:

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

**Data Structure 2.3.5** (`AVLTree` extends `BinSearchTree` (data structure 2.3.4))

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

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

**Definition.**  $T \in \text{BinSearchTree}$  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(<math>x \in \text{Node}</math>)</code>	Splay $x$	$\mathcal{O}(\text{depth of } x)$

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

Method	Splay this
Search for key $k$	The node at which $k$ was found, if $k$ was found; else, the parent of the leaf node hit
Insert	Node inserted
Remove key $k$	Parent of the <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 <b>true</b> 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 <b>Set</b> representing $\text{this} \cup \text{other}$ .	idk
<code>intersection(other ∈ Set)</code>	Return a new <b>Set</b> representing $\text{this} \cap \text{other}$ .	idk
<code>difference(other ∈ Set)</code>	Return a new <b>Set</b> 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 <b>null</b> if not present.	$\Theta(1)$
<code>put(k ∈ Key, v ∈ Value) → Value</code>	Store $k \mapsto v$ in this map, and return old <b>Value</b> (or <b>null</b> if not present).	$\Theta(1)$
<code>remove(k ∈ Key)</code>	Delete $k \mapsto$ (whatever) in this map, and return old <b>Value</b> (or <b>null</b> if not present).	idk
<code>size()</code>	Return $n$ .	$\Theta(1)$
<code>isEmpty()</code>	Return <b>true</b> iff $n = 0$ .	$\Theta(1)$
<code>entrySet() → Set&lt;Entry&gt;</code>	Return set of (key, value) pairs maintained.	idk
<code>keySet() → Set&lt;Key&gt;</code>	Return set of keys maintained.	idk
<code>values() → Collection&lt;Value&gt;</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.

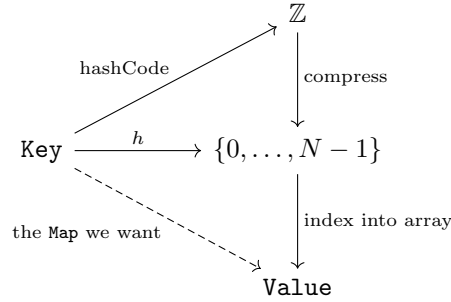
$(A, h)$  is a **hash table** iff  $A \in \text{Array}<\text{Container}<\text{Key}, \text{Value}>>$  and  $h$  is a hash function.

**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*.

## 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 22.

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 22: 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  $\Omega(n \log(n))$  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:  $\Theta(n^2)$  */
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
```

---

### 3.1.2 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
```

---

## 3.2 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
```

---



### 3.3 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(),  $k$ );
```

---

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

---

### 3.5 AVL Tree Methods

---

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

---

```
1 method insert( $T \in \text{AVLTree}\langle \text{Key}, \text{Value} \rangle, k \in \text{key}, v \in \text{Value} \rangle \rightarrow \text{void}$ 
   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( $T, \text{newNode}$ );
4 method remove( $T \in \text{AVLTree}, k \in \text{Key} \rangle \rightarrow \text{void}$ 
   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( $T, \text{parent}$ );
```

---

---

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

---

```
1 method rebalance( $T \in \text{AVLTree}\langle \text{Key}, \text{Value} \rangle, \text{newNode} \in \text{Node}\langle \text{Key}, \text{Value} \rangle \rangle \rightarrow \text{void}$ 
   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( $T \in \text{AVLTree}\langle \text{Key}, \text{Value} \rangle, z \in \text{Node}\langle \text{Key}, \text{Value} \rangle \rangle$ 
    $\rightarrow \text{void}$ 
   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  $T$ ;
```

---

### 3.6 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           zag  $x, y;$ 
10          splay( $T, x$ );
11       else //  $x$  is the right child of  $y$ 
12           zig  $x, y;$ 
13           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       zag-zag  $x, y, z;$ 
17       splay( $T, x$ );
18   else if  $x$  left  $y$  right  $z$  then
19       zag-zig  $x, y, z;$ 
20       splay( $T, x$ );
21   else if  $x$  right  $y$  left  $z$  then
22       zig-zag  $x, y, z;$ 
23       splay( $T, x$ );
24   else //  $x$  right  $y$  right  $z$ 
25       zig-zig  $x, y, z;$ 
26       splay( $T, x$ );
```

---