

COMP3506 Algos and Datas Summary

Gabriel Field

17/08/2023 - END OF COURSE

Contents

1	Boilerplate	1
2	Data Structures	2
2.1	General Linear Structures	2
2.1.1	Children of <code>StaticSequence</code>	3
2.1.2	Children of <code>DynamicSequence</code>	3
2.2	Stacks and Queues	5
2.3	Trees	6
3	Algorithms	8
3.1	Sort	8
3.1.1	Comparison sort	8
3.1.2	Non-comparison sort	11

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`.

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

Changelog:

2023-08-25 16:04 Content up to the **end of week 4 lectures** is now summarised here.

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`
 - `DynamicSequence`
- 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)
 - Non-comparison sort:
 - * Bucket sort (algo 5)
 - * Lexicographic sort (algo 6)
 - Radix sort (algo 7)
 - Binary radix sort (algo 8)

ADT 2.1.2 (`StaticSequence`)

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)
 - Non-comparison sort:
 - * Bucket sort (algo 5)
 - * Lexicographic sort (algo 6)
 - Radix sort (algo 7)
 - Binary radix sort (algo 8)

ADT 2.1.3 (`DynamicSequence`)

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
<code>get(i ∈ {0, ..., n - 1})</code>	Return x_i
<code>set(i ∈ {0, ..., n - 1}, x)</code>	Set x_i to x
<code>add(x)</code>	Add x as a new element

Algorithms this data structure(s) may utilise:

- Sort:
 - Comparison sort:

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

2.1.1 Children of StaticSequence

ADT 2.1.4 (Array implements StaticSequence)

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)
 - Non-comparison sort:
 - * Bucket sort (algo 5)
 - * Lexicographic sort (algo 6)
 - Radix sort (algo 7)
 - Binary radix sort (algo 8)

2.1.2 Children of DynamicSequence

ADT 2.1.5 (LinkedList implements DynamicSequence)

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

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

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

Method	Function	Runtime complexity (worst)
<code>add(x)</code>	Add x to list (at either the head or the tail)	$\Theta(1)$
<code>insert($i \in \{0, \dots, n-1\}, x$)</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)
 - Non-comparison sort:
 - * Bucket sort (algo 5)
 - * Lexicographic sort (algo 6)
 - Radix sort (algo 7)
 - Binary radix sort (algo 8)

ADT 2.1.6 (`ExtensibleList` implements `DynamicSequence`)

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($i \in \{0, \dots, n-1\}$)</code>	Return x_i	$\Theta(1)$ (cf. <code>LinkedList</code>)
<code>set($i \in \{0, \dots, n-1\}, x$)</code>	Set x_i to x	$\Theta(1)$ (cf. <code>LinkedList</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>append(x)</code>	Add x to the tail of the list	<i>amortised</i> $\Theta(1)$; raw $\mathcal{O}(n)$
<code>insert($i \in \{0, \dots, n-1\}, x$)</code>	Insert x immediately before element i	<i>amortised</i> $\Theta(1)$; raw $\mathcal{O}(n)$

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)
 - Non-comparison sort:
 - * Bucket sort (algo 5)
 - * Lexicographic sort (algo 6)
 - Radix sort (algo 7)
 - Binary radix sort (algo 8)

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

2.2 Stacks and Queues

ADT 2.2.1 (Stack)

A dynamic-size FILO data structure storing n elements.

- **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)	Push x onto the stack	$\Theta(1)$ (perhaps amortised from raw $\mathcal{O}(n)$, depending on implementation)
pop()	Return and remove the element at the top	$\Theta(1)$
peek() aka top()	Return the element at the top	$\Theta(1)$
isEmpty()	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)

A dynamic-size FIFO data structure storing n elements.

- **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)	Enqueue x onto the back of the queue	$\Theta(1)$ (perhaps amortised from raw $\mathcal{O}(n)$, depending on implementation)
dequeue()	Return and remove the element at the front	$\Theta(1)$
first()	Return the element at the front	$\Theta(1)$
isEmpty()	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)

2.3 Trees

ADT 2.3.1 (Tree)

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** $\in \text{Set}<\text{Node}>$: 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	$\Theta(n)$
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	$\Theta(1)$
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 ($x \in \text{Node}, y \in \text{Node}$)	Replace x with y	$\Theta(1)$
Tree.addRoot ($x \in \text{Node}$)	Set the root of this Tree to x , and the old root to one of x 's children	$\Theta(1)$
Tree.remove ($x \in \text{Node}$)	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)

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

- **parent** $\in \text{Node}$: pointer to the parent of this **Node**
- **data**: data stored at this node
- **left** $\in \text{Node}$: left child
- **right** $\in \text{Node}$: right child

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

Method	Function	Runtime complexity (worst)
Inherit from Tree	Methods inherited	inherited
<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 level of T is full.

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

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 are *stable* sorts.

For a comparison of comparison sorts, see table 9.

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 9: 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( $A \in \text{GeneralLinearStructure}, n \in \mathbb{Z}_{\geq 0}$ )  $\rightarrow void$ 
    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  $(L, E, G, l, e, g)$ ;
```

3.1.2 Non-comparison sort

Algorithm 5: 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 6: 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( $A \in \text{GeneralLinearStructure}$ ,  $d \in \mathbb{Z}_{>0}$ )  $\rightarrow void$ 
    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      $\text{stableSort}(A, i\text{-th component})$ ;           // keys are the  $i$ -th component

```

Algorithm 7: 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 8: 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
```
