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 a list of references to algorithms that can be used on it.

Each algorithm gives a pseudocode representation, and a list of references to ADTs which may call it.

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`". 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.

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 $x_0, \dots, x_{n-1}$.

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)
  - 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 $x_0, \dots, x_{n-1}$, with the number n of elements allowed to increase.

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
`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)
    - 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*

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)

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	$\Theta(n)$

- 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	$\Theta(n)$
`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`)
`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	*amortised* $\Theta(1)$; 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 stack) `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` $\in$ `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	$\Theta(n)$
`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(`$x \in$ `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`)
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`	Methods inherited	inherited
`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)

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 1.

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 1: Comparison of comparison sorts

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

Algorithm 1: Selection Sort

```
/* This method is a stable sort.                                   */
/* Runtime complexity: Θ(n^2)                                      */
method selectionSort(A ∈ GeneralLinearStructure, n ∈ Z≥0) → void
    Input    : A of length ≤ n
    Requires: A is totally ordered by ≤
    Does     : In-place sorts A

    if n > 1 then
        maxIndex ← 0;
        for i ← 1 to n − 1 do
            if A[i] > A[maxIndex] then
                maxIndex ← 0;
        // Swap max with last
        swap(A[maxIndex], A[n − 1]);
        // Sort the rest
        selectionSort(A, n − 1);
```

Algorithm 2: Insertion Sort

```
/* This method is a stable sort.                                         */
/* Runtime complexity:  Θ(n^2)                                           */
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(n^2)                                           */
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:` $\mathcal{O}\left(n^2\right)$ `*/`

`/* Average-case, best-case runtime complexity:` $\Theta\left(n \log(n)\right)$ `*/`

method `quickSort(`$A \in$ *GeneralLinearStructure*, $n \in \mathbb{Z}_{>0}$`)` $\rightarrow$ *GeneralLinearStructure*

> **Input :** Structure A of length $\leq n$
>
> **Returns :** Destructively sorted copy of A
>
> **if** $n = 1$ **then**
> > **return** A;
>
> `// Else...`
>
> `// Divide`
>
> $p \leftarrow$ pivot index chosen from $\{0, \ldots, n-1\}$; `// often randomly chosen`
>
> $(L, E, G, l, e, g) \leftarrow$ `partition(`A, p`)`;
>
> `// Recurse`
>
> $L \leftarrow$ `quickSort(`L, l`)`;
>
> $G \leftarrow$ `quickSort(`G, g`)`;
>
> `// Conquer`
>
> **return** L.appendAll(E).appendAll(G); `// appendAll() does the obvious`

method `partition(`$A \in$ *GeneralLinearStructure*, $p \in \mathbb{Z}_{\geq 0}$`)` $\rightarrow$ *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
>
> $L, E, G, \leftarrow$ empty sequences of capacity length(A);
>
> $l, e, g \leftarrow 0$;
>
> pivot $\leftarrow A$.remove(p);
>
> **while** A is not empty **do**
> > element $\leftarrow A$.remove(A.first());
> >
> > **if** element $<$ pivot **then**
> > > L.add(element);
> > >
> > > $l \leftarrow l + 1$;
> >
> > **else if** element $=$ pivot **then**
> > > E.add(element);
> > >
> > > $e \leftarrow e + 1$;
> >
> > **else**
> > > G.add(element);
> > >
> > > $g \leftarrow g + 1$;
>
> **return** (L, E, G, l, e, g);

3.1.2 Non-comparison sort

Algorithm 5: 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 6: Lexicographic Sort

```
/* Runtime complexity O(d · 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 ∈ GeneralLinearStructure, d ∈ ℤ>0) → void
    Input    : Structure A of d-tuples
    Requires: The data type D_i of the i-th component is totally ordered by ≤, for each
               i ∈ {1, ..., d}
    Does     : Destructively sort A according to lexicographic order
    for i ← d downto 1 do
        stableSort(A, i-th component);          // keys are the i-th component
```

Algorithm 7: Radix Sort

```
/* Specialisation of lexicographicSort() which uses bucketSort() and
   applies only to non-negative integers.                              */
/* Runtime complexity O(d · (n + N))                                   */
method radixSort(A ∈ GeneralLinearStructure, d ∈ Z_{>0}, n ∈ Z_{≥0}, N ∈ Z_{>0})
 → void
   Input  : Structure A of d-tuples in {0, ..., N − 1}^d of length n
   Does   : Destructively sort A
   for i ← d downto 1 do
      bucketSort(A, n, N, i-th component);      // keys are the i-th component
```

Algorithm 8: Binary Radix Sort

```
/* Specialisation of radixSort() which works in binary.                */
/* Runtime complexity O(b · n)                                         */
method binaryRadixSort(A ∈ GeneralLinearStructure, n ∈ Z_{≥0}, b ∈ Z_{>0}) → void
   Input  : Structure A of length n storing b-bit non-negative integers
   Does   : Destructively sort A
   for i ← 0 to b − 1 do
      bucketSort(A, n, 2, i-th bit);                   // keys are the i-th bit
```