CT2109 - OBJECT ORIENTED PROGRAMMING: DATA STRUCTURES & ALGORITHMS

OBJECT ORIENTED PROGRAMMING: DATA STRUCTURES & ALGORITHMS

Andrew Hayes

2BCT

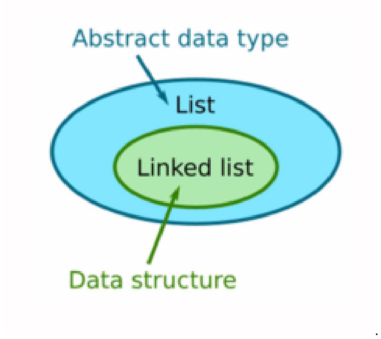
University of Galway

April 20, 2023

Contents

1 Abstract Data Types

An **Abstract Data Type (ADT)** is an abstract model of a data structure that specifies the data stored & oeprations that may be performed on the data. An ADT specifies *what* each operation does, but not *how*. In object-oriented languages such as Java, this naturally corresponds to an **interface definition**. An ADT is *realised* as a concrete data structure. In Java, this is a class that **implements** the interface.



Composite ADTs are used manage *collections* of data, e.g., Arrays, Lists, Stacks, Queues, Hash Tables, etc.

1.1 Stacks & Queues

Stacks & **Queues** are linearly ordered ADTs for list-structured data.

1.1.1 Stacks

A **Stack** is a last in, first out (LIFO) data structure. No sort order is assumed. Items can only enter or leave via the *top* of the stack. Items can be **pushed** & **popped** to add & remove. Example applications of a stack include processing nested structures or the "undo" operation in an editor. Objects stored in a stack are a *finite sequence* of elements of the **same type**.

Stacks have few operations. For a stack `s`, node `n`, & boolean value `b`:

- `s.push(n)` - Place item `n` on top of the stack.
- `s.pop()` → `n` - Remove top item from the stack & return it.
- `s.top` → `n` - Examine the top item on the stack without removing it.
- `s.isEmpty()` → `b` - Returns `b = true` if the stack is empty.
- `s.isFull()` → `b = true` if the stack is full (relevant if storage is limited).

Java has a built-in stack interface from `java.util.Stack`. However, we will look at making our own for the sake of learning. Our stack implementation may look something like this:

```
public interface Stack {
    public void push(Object n);
    public Oject pop();
    public Object top();
    public boolean isEmpty();
    public boolean isFull();
```

Other stack operations include `size()` & `makeEmpty()`. We could implement this stack using an array, linked list, or other storage type.

1.1.2 Queues

A **Queue** is a first in, first out (FIFO) data structure. No sort order is assumed. Items enter at the rear of the queue, and leave at the front of the queue. Items can be **enqueued** & **dequeued** to add & remove them from the queue. Example applications of a queue include ensuring "fair treatment" to each of a list of pending tasks (first come, first served) or simulation: modelling & analysis of real-world problems. Objects stored in a queue are a finite sequence of elements of the same type. The item at the front of the queue has been in the queue the longest, while the item at the rear has entered the queue most recently.

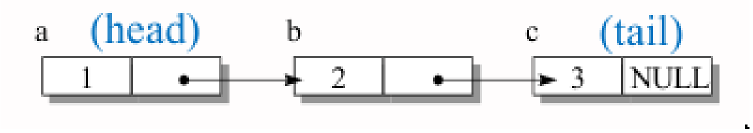
Queues have few operations. For a queue `q`, element `e`, & boolean value `b`:

- `q.enqueue(e)` - Place `e` at the rear of `q`, assuming there is space.
- `q.dequeue()` → `e` - Remove fron item `e` from `q` and return it.
- `q.front()` → `e` - Returns front element `e` without removing it.
- `q.isEmpty()` → `b` - Returns `b = true` if the queue is empty.
- `q.isFull()` → `b` - Returns `b = true` if the queue is full.

With an array implementation of a queue, items must be "shuffled" towards the front after a *dequeue*. Note that with an array implementation, once `rear` becomes equal to $N - 1$, no further items can be enqueued (array space limitation).

1.2 Linked Lists

A **Linked List** is an abstract data type which stores an arbitrary-length list of data objects as a sequence of *nodes*. Each node consists of data and has a **link** to the next node. Each node, excepting the last, is links to a **successor node**.



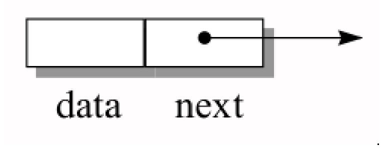
Characteristics of Linked Lists:

- **Self-referential** structure type - Every node has a pointer to a node of the same type.
- Very useful for **dynamically** growing/shrinking lists of data.
- Compared to arryas, drastically reduces the effort required to add/remove items from the middle of the list.
- Solves the potential problem of **overflow** that arrays have.
- **Sequential access** - It is **inefficient** to retrieve an element at an arbitrary position, relative to an array.

1.2.1 Implementation of Linked Lists

We define a **Node** class, with members **data** (whichever variables are required) & **next** (reference to another Node object).

Each node occcurrence is linked to a succeeding occurence by way of the member **next**. If `next` is `null`, then there is no item after this node in the list (termed the **tail** node). The starting point for the list is the **head** node. We can trace from the head node to any other node.



```java
public class Node {
    // instance variables
    private Object element;
    private Node next;

    // creates node with null refs to its element \& next node
    public Node() {
        this(null, null);
    }

    // creates node with the given element & next node
    public Node(Object e, Node n) {
        element = e;
        next = n;
    }

    // accessor methods
    public Object getElement() {
        return element;
    }
    public Node getNext() {
        return next;
    }

    // mutator methods
    public void setElement(Object newElem) {
        element = newElem;
    }
    public void setNext(Node newNext) {
        next = newNext;
    }
}
```

Generally, we don't create nodes manually, rather we just supply element data to a method which keeps track of the current position in the list.

Typical methods in a Linked List ADT include:

- `long size()` - Returns the size of the list.
- `boolean isEmpty()` - Returns `true` if the list is empty, `false` otherwise.
- `Object getCurr()` - Returns the element at the current position.
- `boolean gotoHead()` - Sets the current position to `head`, returning `true` if successful.
- `boolean gotoNext()` - Moves to the next position, returning `true` if successful.
- `void insertNext(Object el` - Creates a new node after the current node.
- `void deleteNext()` - Removes the node after the current node.
- `void insertHead(Object el)` - Creates a new node at the head.
- `void deleteHead()` - Removes the head node.

1.2.2 Singly Linked List Class

A **singly linked list** is one in which each node links to a *single* other node.

The Singly Linked List Class should store the head of the list & the current position. For efficiency, it also keeps track of the current size of the list (alternatively, we could just count its nodes when needed).

```java
public class SLinkedList {
    protected Node head;    // head node of the list
    protected Node curr;    // current position in list
    protected long size;    // number of nodes in the list

    // default constructor which creates an empty list
    public SLinkedList() {
        curr = head = null;
        size = 0;
    }

    // insert, remove, & search methods go here
}
```

2 Algorithm Analysis

All algorithms take CPU **time** & **memory** space. Often, we can make tradeoffs, choosing algorithm variants that either user more memory, or more CPU. If the memory space requirements of an algorithm are large, the program may use disk sawp space rather than RAM, which is much slower. If the memory requirements are too large, then the program cannot run. Often, we identify the algorithms that don't require "too much" spaace, and then choose the one with the lowest CPU requirements. The purpose of algorithm analysis is comparing the time & space requirements of various algorithms.

"Why not just run the algorithm and measure the time & space used?" - While this is sometimes done when theoretical analysis is difficult, it is better to be able to evaluate algorithms "on paper" without first having to implement, debug, and test them all. It's important to have a measure that's *independent* of particular computer configurations and to be able to compare algorithms reliably, without being influenced by variations in implementation. We want to understand how an algorithm will perform on large problems and identify "hot spots" to give our attention to when developing & optimising programs.

2.1 Algorithm Analysis Basics

Theoretical Analysis uses a high-level *pseudocode* description of the algorithm instead of a real implementation, and characterises run-time as a function of input size n. This function specifies the **order of growth** of rate of runtime as n increases. Theoretical analysis takes into account all possible inputs and evaluates speed independent of hardware or software.

2.1.1 Counting Primitive Operations

The basic approach is deriving the function for the **count of the primitive operations**. The primitive operations are the individual steps performed by a program. We assume that each step takes the same amount of time and examine any terms that control repetition.

Example: Algorithm to find the largest element of an array. We count the maximum number of operations as function of array size n.

```
Algorithm arrayMax(A, n)            // Number of Operations
    currentMax = A[0]               // 2
    for i = 1 to n -1 do            // 2n
        if A[i] > currentMax then   // 2(n-1)
            currentMax = A[i]       // 2(n-1)
            {increment coutner i}   // 2(n-1)
    return currentMax               // 1
                                    // Total: 8n-3
```

We could consider the **average**, **best**, or **worst** case. Usually, we analyse the worst case, as we want our algorithms to work well even in bad cases. The average case is quite important too, if different from the worst case. These counts are the basis of (big) O notation.

2.2 O Notation

The basic approach to **O Notation** involves deriving an expression for the count of basic operations (as discussed). We focus on the *dominant term*, and ignore constants. E.g., $O(5n^2 + 1000n - 3) \rightarrow O(n^2)$. Since contants & low-order terms are eventually dropped, we can disregard them when counting primitive operations.

O Notation is used for **asymptotic analysis of complexity** - the trend in the algorithms runtime as n gets large. We look at the **order of magnitude** of the number of actions, independent of computer/compiler/etc.

Note: We specifically care about the **tightest** upper bound. Technically speaking, an algorithm that is $O(n^2)$ is also $O(n^3)$, but the former is more informative. The function specified in O notation is the **upper bound** on the behaviour of the algorithm being analysed. This can be the best/average/worst case behaviour.

Example: Let $f(n) = 6n^4 - 2n^3 + 5$. Apply the following rules:

- If $f(n)$ is a sum of several terms, then only the one with largest rate of growth is kept.
- If $f(n)$ is a product of several factors, any constants that do not depend on n are ommitted.

Thus, we say that $f(n)$ has a "big-oh" of (n^4). We can write $f(n)$ is $O(n^4)$.

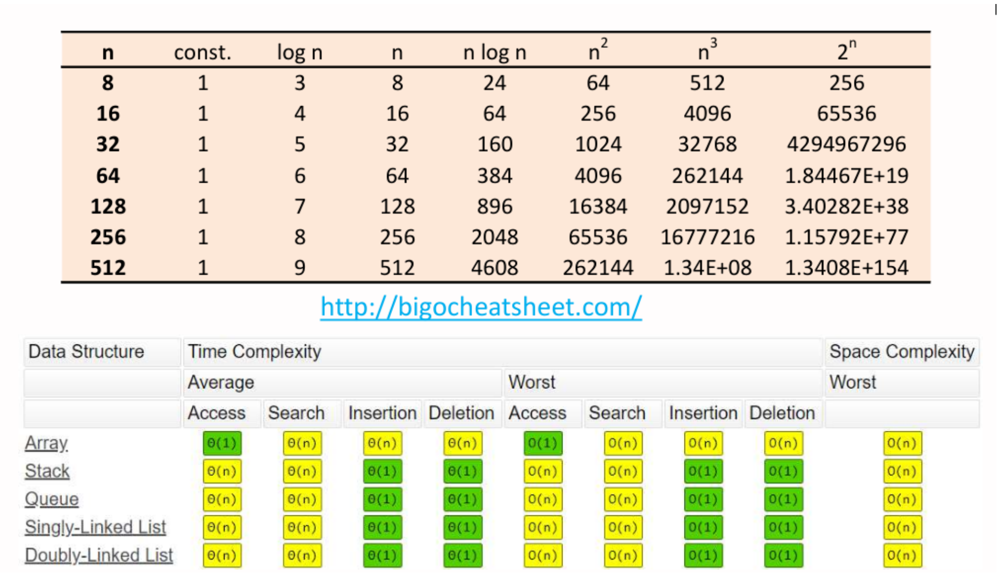
2.2.1 Important Functions Used in O Notation

Functions commonly used include:

- **Constant:** $O(1)$.
- **Logarithmic:** $O(\text{log}n)$.
- **Linear:** $O(n)$.
- **n-Log-n:** $O(n\text{log}n)$.
- **Quadratic:** $O(n^2)$.
- **Cubic:** $O(n^3)$.
- **Exponential:** (1^n).

Notes:

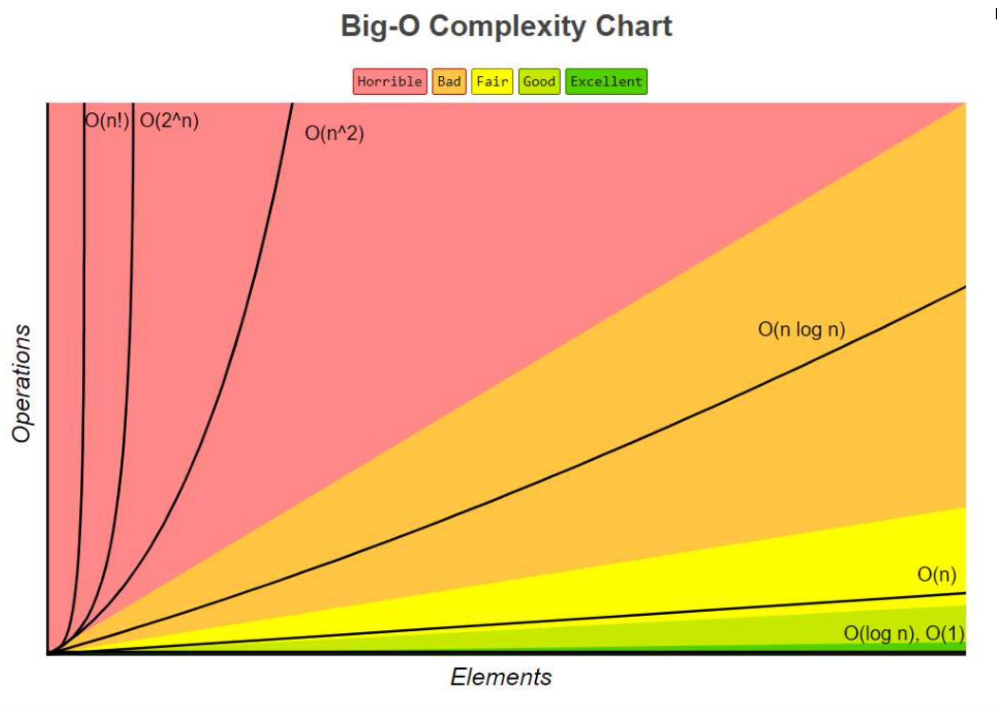
- By convention, all logs are base 2 unless otherwise stated.
- Two algorithms having the same complexity doesn't been that they are exactly the same, it means that their running times will be *proportional*.



n	const.	log n	n	n log n	n^2	n^3	2^n
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4096	65536
32	1	5	32	160	1024	32768	4294967296
64	1	6	64	384	4096	262144	1.84467E+19
128	1	7	128	896	16384	2097152	3.40282E+38
256	1	8	256	2048	65536	16777216	1.15792E+77
512	1	9	512	4608	262144	1.34E+08	1.3408E+154

http://bigocheatsheet.com/

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Stack	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$



2.2.2 Efficiency & O Notation

- **Constant:** Most efficient possible, but only applicable to simple jobs.
- **Logarithmic, Linear, & n-Log-n:** If an algorithm is described as "efficient", this usually means $O(\log n)$ or better.
- **Quadratic & Cubic:** Not very efficient, but polynomial algorithms are usually considered "tractable" (acceptable for problems of reasonable size).
- **Exponential:** Very inefficient. Problems that (provably) require an algorithm of O greater than polynomial complexity are called "**hard**".

2.3 Recursion Review

Methods can call other methods, but they can also call themselves, either directly, or indirectly, via another method. This creates a type of loop called **recursion**.

Iteration can be used anywhere that you can use recursion. Sometimes, recursion can be a more elegant solution, if it reflects the way that the problem is usually thought about, as we aim to use the most intuitive representation of the problem. Recursion can make complexity analysis easier in some cases.

The drawbacks of recursion include:

- Inefficient use of the function: Large amount of concurrent, deeply nested method calls.
- If done naively, the number of calls can explode.
- Depending on the algorithm, we need to take care not to recompute values unnecessarily.

3 Dynamic Programming

The basic idea of **Dynamic Programming** is to solve complex problems by breaking them into simpler sub-problems. When a solution to a sub-problem is found, store it ("memo-ize") so that it can be re-used without recomputing it. Combine the solutions to the sub-problems to get the overall solution.

This is particularly useful when the number of repeating sub-problems grows exponentially with the problem size.

In general, dynamic programming takes problems that appear exponential and produces polynomial-time algorithms for them. The trade-off in dynamic programming is between *storage* & *speed*. Dynamic programming is widely used in heuristic optimisation problems.

For Dynamic Programming, the problem structure requires three components:

1. **Simple sub-problems:** Must be able to break the overall problem into indexed sub-problems & sub-sub-problems.
2. **Sub-problem decomposition:** Optimal/correct solution to the overall problem must be composed from sub-problems.
3. **Sub-problem overlap:** So that elements can be re-used.

The basic steps in the approach to DynProg:

1. Set up the overall problem as one that is decomposable into overlapping sub-problems that can be indexed.
2. Solve the sub-problems as they arise and **store solutions** in a table.
3. Derive the overall solution from the solutions in the table.

3.1 More Big Greek Letters

$O(n\log n)$ ("Big Oh"):

- Upper bound on asymptotic complexity.
- In this case, there is a constant c_2 such that $c_2 n\log n$ is an upper bound on asymptotic complexity.

$\Omega(n\log n)$ ("Big Omega"):

- Specifies a lower bound on asymptotic complexity.
- In this case, the algorithm has a lower bound of $c_1 n\log n$.

$\Theta(n\log n)$ ("Big Theta"):

- Specifies the upper & lower bounds.
- In this case, there exist two constants, c_1 & c_2, such that $c_1 n\log n < f(n) < c_2 n\log n$.

Of these, $\Theta()$ makes the strongest claims: It specifies that the rate of growth is no better and no worse than some level. Requires additional analysis relative to O.

There are also some others that are common in Mathematics but not in Computer Science:

- **Little o:** $o(g(n))$ specifies a function $g(n)$ that grows much faster than the one that we are analysing.
- **Little omega:** $\omega(g(n))$ specifies a function $g(n)$ that grows much slower than the one that we are analysing.

Don't confuse upper/lower bounds with best/worst case: all cases have bounds.

3.2 P, NP, & NP-Complete Problems

P Problems are those for which there is a **deterministic** algorithm that solves it in **Polynomial Time**. In other words, the algorithm's complexity is $O(p(n))$ where $p(n)$ is a polynomial function.

A (trivial) example of a P problem is searching an array of integers for a certain value.

Problems that can be solved in polynomial time are termed **tractable**, while worse problems are termed **intractable**.

NP Problems (Non-deterministic Polynomial) are those algorithms which have two repeating steps:

- Generate a *potential solution*, either randomly or systematically.

- Verify whether the potential solution is right, and if not, repeat.

If the verification step is **polynomial**, the algorithm & associated problem are **NP**.

An example of an NP problem is factoring large integers as used in RSA encryption. Another example of an NP problem is the **subset problem**: Given a set of integers, does some non-empty subset of them sum to 0? There is no polynomial algorithm to solve this problem. However, verification of a potential solution is polynomial ($O(n)$ (just add up the numbers in the potential solution)).

Note that P is a subset of NP.

NP-Complete problems are those that are "**as hard as** all others" in NP, i.e. algorithms that are comparable to ("**polynomially reducible** to") others in NP but not reducible to P. If an algorithm is **polynomially reducible**, there is some polynomial-time transformation that converts the inputs for Problem X to inputs for Problem Y.

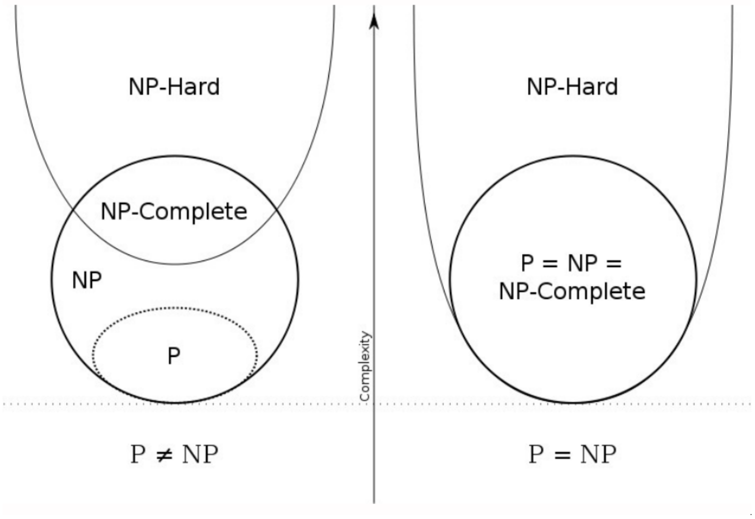
NP-Complete is a complexity class which represents the set of all problems X in NP for which it is possible to reduce any other NP problem Y to X in polynomial time. Intuitively, this means that we can solve Y quickly if we know how to solve X quickly. What makes NP-complete problems inportant is that if a deterministic polynomial time algorithm can be found to solve one of them, every NP problem is solvable in polynomial time.

NP-Hard problems are those that are "**as hard or harder**" than all others in NP. A problem X is Np-Hard if NP-Complete problems are polynomially reducible to it.

Intuitively, NP-hard problems are problems that are at least as hard as the NP-complete problems. Note that NP-hard problems do not have to be in NP, and they do not have to be decision problems. The precise definition here is that "a problem X is NP-hard, if there is an NP-complete problem Y, such that Y is reducible to X in polynomial time". An example of an NP-hard problem is *the halting problem*: Given a program P and input I, will it halt?

The "P versus NP" problem is a major unsolved problem in computer science: "If the solution to a problem is easy to verify, is the problem also easy to solve?" or "whether every problem whose solution can be **quickly verified** by a computer can also be **quickly solved** by a computer". "Quickly" here means that there exists an algorithm to solve the task that runs in polynomial time. An answer to the $P = NP$ question would determine whether all problems that can be verified in polynomial time can also be solved in polynomial time. If it turned out that $P \neq NP$, then it would mean that there are problems in NP (such as NP-complete problems) that are harder to compute than to verify. We already know that $P \subseteq NP$.

In theoretical computer science, the problems considered for P & NP are **decision problems**, i.e. problems that don't produce numeric results but yes or no answers.



- **P** (Polynomial): Solvable in polynomial time.
- **NP** (Non-Deterministic Polynomial): Only *verifiable* in polynomial time.

4 Searching & Sorting

4.1 Keys & Values

Each object to be sorted can be considered to have a **key** & a **value**, e.g. A Student has properties Name, ID, & grade.

4.2 Java Interface: Comparator

The `Comparator` interface compares two objects to say which should come first. In Java, any class that implements `java.util.Comparator` interface is only required to implement one method:

```java
int compare(Object ob1, Object ob2);
```

This returns a negative number if `ob1` is less than `ob2`, a positive number if `ob1` is greater than `ob2`, and `0` if `ob1` is equal to `ob2`.

4.3 Java Interface: Comparable

The `Comparable` interface compares a given object to another to see which object should come first. The two objects that are being compared must be of a class that implements `java.lang.Comparable`, which has just one method to implement:

```java
int compareTo(Object other);
```

Standard classes such as `String` implement this.

4.4 Insertion Sort

Consider sorting a bookshelf using **Insertion Sort**:

1. Remove the next unsorted book.
2. Slide the sorted books to the right one by one until you find the right sport for for the removed book.
3. Insert the book into its new position once it is found.

Algorithm 1 Insertion Sort Pseudocode

```
Require: $A[0 \ldots N-1]$                                        ▷ Unsorted Array
Ensure: $A[0 \ldots N-1]$                                         ▷ Sorted Array
  procedure INSERTION SORT($A[0 \ldots N-1]$)
      for $ToSort \leftarrow 1$ to $N-1$ Step 1 do
          $Index = ToSort - 1$
          $ToSortEl = A[ToSort]$

          while $Index \geq First$ AND $A[Index] > ToSortEl$ do
              $A[Index+1] \leftarrow A[Index]$                    ▷ Shuffle elements to the right
              $Index \leftarrow Index - 1$
          end while

          $A[Index+1] = ToSortEl$          ▷ Insert the element to sort in its appropriate place
      end for
      return $A[]$                                                ▷ Return the sorted array
  end procedure
```

The worst case efficiency of Insertion Sort is $O(n^2)$. The best case efficiency of Insertion Sort is $O(n)$. If the array is closer to sorted order, the algorithm does less work, and the operation is more efficient. This leads to a related algorithm: Shell Sort.

4.5 Shell Sort

Shell Sort is more efficient than Selection Sort or Insertion Sort. It works by comparing distant items first, and works its way down to nearby items. The interval is called the **gap**. The gap begins at one half of the length of the list and is successively halved until each item has been compared with its neighbour.

Insertion Sort only tests elements in *adjacent* locations - it might take several steps to get to the final location. Insertion Sort is more efficient if an array is partially sorted. By making larger jumps, Shell Sort makes the array become “more sorted” more quickly.

Algorithm 2 Shell Sort Pseudocode

```
Require: $A[0 \dots N-1]$                                        ▷ Unsorted Array
Ensure: $A[0 \dots N-1]$                                          ▷ Sorted Array
  procedure SHELL SORT($A[0 \dots N-1]$)
      $Gap \leftarrow floor(\frac{Gap}{2})$     ▷ Round to the nearest odd number, as it's best with an odd-sized gap

      while $Gap \geq 1$ do
          for $ToSort \leftarrow 1$ to $N-1$ Step 1 do
              $Index = ToSort - 1$
              $ToSortEl = A[ToSort]$

              while $Index \geq First$ AND $A[Index] > ToSortEl$ do
                  $A[Index+1] \leftarrow A[Index]$                ▷ Shuffle elements to the right
                  $Index \leftarrow Index - 1$
              end while

              $A[Index+1] = ToSortEl$          ▷ Insert the element to sort in its appropriate place
          end for

          $Gap \leftarrow floor(\frac{Gap}{2})$
      end while
  end procedure
```

The worst-case complexity of Shell Sort is $O(n^2)$. However, this is because the gap is sometimes even which results in sub-arrays that include all the elements of an array that was already sorted. To avoid this, we round the gap up to the nearest odd number which gives us a worst-case complexity of $O(n^{1.5})$. Other gap sequences can improve performance a little more, but this is beyond the scope of this topic.

4.6 Quick Sort

Quick Sort is a divide-and-conquer algorithm.

1. Firstly, it partitions the array into two sub-arrays that are **partially sorted**.
2. Then, it picks a **pivot value**, and re-arranges the elements such that all elements less than or equal to the pivot value are on the left of the pivot, and all elements that are greater than it are on the right.
3. The array is now divided into sub-arrays and a pivot value.
4. This procedure is then repeated **recursively** for each sub-array, to further sort each of them.
5. When the algorithm has reached the level of a sub-array with just one element, that sub-array is sorted. All sub-arrays are sorted relative to each other, so the whole array is sorted when all the sub-arrays are.

5 Trees

The data structures that we looked at previously placed data in linear order, but we sometimes need to organise data into groups & sub-groups. This is called **hierarchical classification**, where data items appear at various levels within the organisation.

In Computer Science, a tree is an abstract model of a hierarchical structure which consists of nodes with a parent-child relation. Trees have applications in organisation charts, file systems, programming environments, & more. A **tree** is a set of nodes, connected by edges. The edges indicate relationships between nodes.

Nodes are arranged in **levels**, which indicate the node's position in the hierarchy. Nodes with the same parent node are called **siblings**. The only node with no parent is the **root** node. All other nodes have one parent each. A node with no child nodes is called a **leaf** node or an **external** node. All other nodes are referred to as **internal** nodes.

A node is reached from the root by a **path**. The length of a path is the number of edges that compose it. This is also referred to as the depth of the node. The **height** of a tree is the number of levels in the tree. The number of nodes along the longest path is equal to the maximum depth plus one. Note that we talk about the depth of a node, but the height of a tree.

The ancestors of a node include its parent, grandparent, great-grandparent, etc. The descendants of a node include its children, grandchildren, great-grandchildren, etc. A **subtree** of a node is a tree that has that node as its root, including the node, all its descendants, and the arcs connecting them.

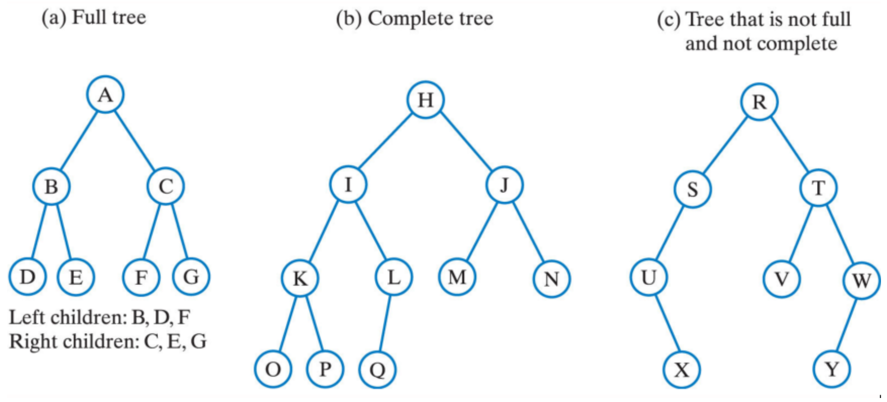
5.1 Binary Trees

A **binary tree** is one in which each internal node has at most two children (& exactly two if it is a "**proper**" binary tree). The children of a node are an **ordered pair**. We refer to the children of an internal node as the left child & the right child. Non-binary trees are termed general trees.

An alternative (recursive) definition for binary trees is a tree that is either just single node or a tree whose root has an ordered pair of children, each of which is a binary tree.

A **full binary tree** is one in which is a proper binary tree, and in which all the leaves are on the same level. This is only achievable for certain numbers of nodes. A **complete binary tree** is a binary tree which is full to the penultimate level and the leaves on the last level are filled frmo left to right. This is achievable for any number of nodes.

The height of either a complete or full binary tree with n nodes is $\log_2(n+1)$.



(a) Full tree

(b) Complete tree

(c) Tree that is not full and not complete

5.2 Generics in Java

The < > operators relate to the concept of **generics**. Generics are used to specify a specific type parameter for a generic collection class. This saves us from having to cast objects in methods such as `add()`, `set`, &

remove. This is a big advantage, as type checking is now done at compile time. Without generics, compile-time type-checking is impossible, since we don't have a type specification for the list. Other object-oriented programming languages have similar concepts, such as templates in C++.

ArrayLists are part of the Java Collections framework, a standard library of pre-built data structures. The underlying storage of an ArrayList is an array. The ArrayList class looks after resizing it as required. ArrayLists can be used with or without generics notation.

```java
// ArrayList code without generics:
ArrayList words = new ArrayList();  // holds objects
words.add("hello");
String a = (String) words.get(0);   // return type of get() is object, so must cast to String

// ArrayList parameterised to specifically hold Strings
ArrayList<String> words = new ArrayList<String>();
words.add("hello");
String a = words.get(0);            // no cast needed
```

Creating a Generics collection:

```java

public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}
public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

6 Search Trees

A **Search Tree** organises its data so that searching it is efficient.

6.1 Binary Search Trees

A **Binary Search Tree (BST)** is a binary tree with nodes that contain Comparable objects. A node's data is greater than the data in the left subtree and less than the data in the right subtree. Usually, no duplicates are allowed.

An **in-order** traversal of a BST will visit all nodes in ascending order.

BSTs are not uniquely structured - The structure of a BST depends on what node is chosen as the root of the tree and the order in which all the other nodes are added.