# Data Structures

Dhruva Karkada

Fall 2017

## Contents

# 1 Programming Principles

Programming is hard because it is nonlinear. A small change to the initial conditions–a misplaced parenthesis, or a mistyped number–can lead to dramatic and often unpredictable changes in the program's behavior. Specifically, it is difficult and discouraged to add redundancy–to 'overengineer' software. This means that a single broken component can result in the failure of the entire software.

In addition, software is constantly evolving. Maintenance of software takes up 55% of the software's lifecycle, and includes fixing bugs, adding new features, and keeping up with a changing environment. Software never exists in isolation, and tends to depend heavily on other software; when its dependencies evolve, you are forced to evolve as well.

Testing is a large part of the software development process, and is a large part of what makes programming difficult. Testing aims to ensure program correctness, to a reasonable degree of confidence.

**Blackbox Testing**
Eyeballing; Controlling input; Analyzing final output only

**Unit Testing**
Testing individual components, not just the whole software together

**Statement Coverage**
The testing covers every statement in the code. Not sufficient; doesn't cover every path and therefore may not reveal all bugs.

**Path Coverage**
Much stronger condition, but harder; use Direct Automated Random Testing

**Test Harness**
A controlled testing environment. Aims to minimize the number of free variables.

In the waterfall model of programming, there is a strict, sequentially linear timeline for development. The program is carefully considered and a solution is carefully designed before construction; testing is performed after construction. This model tends to be inflexible for design changes during the development.

On the other hand, the extreme programming model (a type of agile development) involves frequent, small releases; incremental programming and testing (simultaneously during the construction process); and extensive code review or peer programming. The idea behind this is to minimize the time spent finding bugs in massive amounts of code. Debugging is the most time-consuming and expensive part of the software development cycle, and this method aims to reduce that cost.

# 2 Object Oriented Programming

A programming language is said to be Turing-complete if it supports integers, arithmetic, if/else, and looping. These very basic functions are the basis of most computing; higher-level abstractions have been created to make software development easier.

Object-oriented programming is one such abstraction. Object-oriented languges exist that explicitly support object-oriented syntax for convenience, but object-oriented programming is a paradigm; it is not dependent on language. The main goals of object oriented programming are:

1. Reuse code
2. Integrate modules
3. Facilitate change
4. Understand the function of the whole

To help achieve these goals, three main principles were developed: encapsulation, inheritance, and subtype polymorphism.

## 2.1 Encapsulation

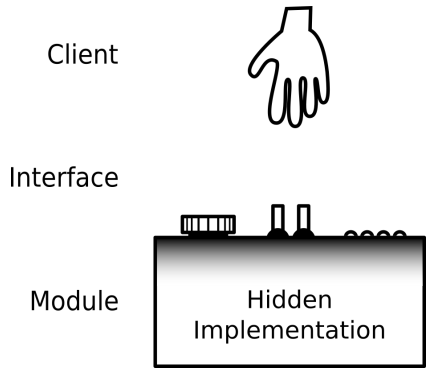
**Information Hiding Principle**

Dave Parnas, 1972.

Use of a module shouldn't know more than how to use it.

Implementor of a module shouldn't know more than how to implement it.

In OOP, the encapsulation of data and the hiding of implementation details helps accomplish OOP goals 2, 3, and 4. In this view, modules are blackboxes for the client; only their interface is visible, not their internal mechanics. This way, the internal mechanics can be changed by the developer with no effect on client code. It also abstracts and clarifies the interactions between subparts of the software.




## 2.2 Inheritance

**Inheritance**

A way by which code can be reused between modules, by defining a supertype-subtype relationship.

Inheritance directly helps accomplish goal 1. In Java, a subclass relationship implies a subtype relationship. Java allows for two main ways of using inheritance: extending classes and implementing interfaces.

**Class extension**
A subclass inherits methods and instance fields from its parent; however, it can't directly access private fields in its parent. A class may have at most one parent.

**Abstract class**
A non-instantiable class which may contain method declarations without implementations; concrete subclasses must override and implement these abstract methods.

**Interface implementation**
A class can implement multiple interfaces, which defines a set of methods but no implementations and no fields. Interfaces are not directly instantiable.

## 2.3 Typing and Polymorphism

Typing is the process of associating variables in the code with types. This ensures that correct methods are called and that operations are legal.

**Type**
A set of possible values and legal operations. Give semantics of language. In Java, classes act as modules and as types. Subclass $\rightarrow$ Subtype.

**Typing (Type checking)**
Determining a type signature given an expression (i.e. determining the types of variables).

**Static Typing**
Occurs during compilation; uses declared types. Faster, safer.

**Dynamic Typing**
Occurs during runtime. More precise.

**Binding**
Choosing a method to run given a type signature.
Dynamic: Use dynamic type checking
Static: "gravest possible crime" in OOP

Java uses static typing (to check legality of operations) and dynamic binding (for method-calling).

Polymorphism is the abstraction of types. There are different kinds of polymorphism; subtype polymorphism is most often associated with OOP.

**Subtype Polymorphism**
Using a subtype where a supertype is expected.
Example: passing a Triangle as a method param where the method asks for a Shape.

**Ad-hoc Polymorphism**
Method overloading

**Parametric Polymorphism**
Generics (type-independence).

**Downcasting**
Changing a supertype to a subtype; requires introspection to be safe.

**Covariance**
Interaction between subtype and parametric polymorphism.
Example: Is List<Triangle> a subtype of List<Shape>? In Java, no; they aren't covariant. But Triangle[ ] is subtype of Shape[ ], so arrays are covariant.

| **Algorithm** Sample code | |
|---|---|
| 1: Triangle $t$ = new Triangle(); | |
| 2: Shape $s$ = new Shape(); | |
| 3: $t = s$; | ▷ **X** Compile error |
| 4: reset($t, s$) | |
| 5: $s = t$; | ▷ ✓ Subtype Poly |
| 6: reset($t, s$) | |
| 7: $t$ = (Triangle)$s$; | ▷ **X** Runtime error |
| 8: reset($t, s$) | |
| 9: $s$ = new Triangle(); | |
| 10: $t$ = (Triangle)$s$; | ▷ ✓ Downcast |

The example shows which cases are valid and which are invalid. The reset method is assumed to be equivalent to lines 1 and 2.

Subtype polymorphism helps accomplish goals 1 and 4.

# 3 Algorithms and Sorts

## 3.1 Algorithms

**Big O (upper bound)**
Function $f(n)$ is $O(g(n))$ if there are positive constants $C,\ k$ such that

$$\forall n > k.\ f(n) \leq C \cdot g(n)$$

**Big $\Omega$ (lower bound)**
Function $f(n)$ is $\Omega(g(n))$ if there are positive constants $C,\ k$ such that

$$\forall n > k.\ f(n) \geq C \cdot g(n)$$

**Big $\Theta$ (tight bound)**
Function $f(n)$ is $\Theta(g(n))$ if it is both $O(g(n))$ and $\Omega(g(n))$.

$$\forall n > k.\ C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n)$$

Sorting is one major application of studying algorithms. A sort is considered stable if it preserves the relative order between equal elements. Particularly inefficient comparison-based sorts include bubble sort and selection sort. A more efficient divide-and-conquer sort is mergesort (stable).

**Theorem:** Any deterministic comparison-based sort must perform $\Omega(n \log(n))$ comparisons in the worst case.

## 3.2 Radix Sort

Radix sort is an example of a sort that is not comparison based. It runs in $O(n)$, but makes assumptions about the type of data that it is sorting (namely, the data must have positional notation, e.g. integers).

Radix sort uses bins to sort integers by their place-values. Since there are 10 bins for each fo the 10 digits. On the first pass, the data is binned by the ones-place; the next pass by the tens-place; until all the places on all the data have been reached. The result is a sorted list. In total, there are $w$ passes, where $w$ is the length of the longest element.

## 3.3 Quicksort

Quicksort is another fast comparison-based sorting algorithm; unlike merge-sort, it is not stable. However, it can be performed in-place, meaning that it requires no extra space.

**Algorithm** Quicksort

```
function QSORT(int[ ] arr, low, hi)                    ▷ recursive sort
    if low < hi :
        pivInd ← PART(arr, hi, low)                    ▷ partition arr around pivot
        QSORT(arr, low, pivInd-1)
        QSORT(arr, pivInd+1, hi)
function PART(int[ ] arr, low, hi)
    pivi ← MEDIAN3(arr, low, hi)                       ▷ choose pivot index
    pivot ← arr[pivi]
    SWAP(arr, pivi, hi)                                ▷ move pivot to end
    pivi ← hi                                          ▷ update pivot index
    decr hi                                            ▷ update hi index
    while low ≤ hi :
        while arr[low] < pivot :
            incr low
        while arr[hi] > pivot :
            decr hi
                                                       ▷ found elements in wrong partition
        if low ≤ hi :
            SWAP(arr, low, hi)                         ▷ swap badly-partitioned elements
            incr low
            decr hi
    SWAP(arr, low, pivi)                               ▷ move pivot back to middle
    return low                                         ▷ return pivot index
```

# 4 Trees

Data structures are an implementation of abstract data types. An abstract data type just specifies a set of potential values and operations on them; they are not tied to implementation. For example, the linked list ADT can be implemented as an array or as separate nodes. A data structure, on the other hand, specifies the implementation, so we can calculate the time complexity of certain operations.

**Tree: graph definition**
A collection of $n$ nodes connected by $(n-1)$ edges.

**Tree: unique path definition**
A set of nodes, one of which designated as root, such that a unique path exists between the root and any other node.
Theorem: A tree cannot contain cycles.

**Tree: recursive definition**
A root node connected to 0 or more disjoint subtrees which don't include the root.

**Binary Tree**
A tree where each node has at most 2 children.

**Balanced Binary Tree**
A BT such that the height is approximately $O(\log n)$

**Complete Binary Tree**
A balanced BT tree where all levels except possibly the last are filled; last layer is filled left to right.

**Perfect Binary Tree**
A complete BT tree with the last level filled.

**Pathological Tree**
A tree such that each node has only one child; identical to a linked list.

**Binary Search Tree**

For all nodes $x$, all nodes in $x$'s left subtree have key values $< x$ key value; all nodes in $x$'s right subtree have key values $> x$ key value. This is the BST property. In this definition, no duplicate keys.

**Tree traversal**

Typically uses recursion (stack-based) to visit every node.
**Pre-Order**: Process current node; Visit left subtree; Visit right subtree
**In-Order**: Visit left subtree; Process current node; Visit right subtree
**Post-Order**: Visit left subtree; Visit right subtree; Process current node
**Level-Order**: Non-recursive (i.e. queue-based) traversal

The main operations on BST (generalized as *dictionary operations*) are:

- find(key)
  - use BST property to search
  - runs in $O(h)$, $h$ = height
- insert(key, val)
  - find(key);
  - if found, replace current value with new val
  - if not found, insert new leaf at closest node
- delete(key)
  - find(key);
  - if node is leaf, snip
  - if node has one child, splice child with parent
  - if two children, swap with smallest val in right subtree; snip/splice

# 5 AVL Trees

Since tree operations are $O(h)$, it is useful to minimize the height of the tree. This is known as balancing the tree, and advanced trees include self-balancing mechanisms.

**AVL Tree**

Adelsen-Velskii & Landis, 1962

A self-balancing BST such that the height difference between every node's left & right subtrees $\leq 1$.

Uses rotations after insert/delete to keep the tree balanced.

**Tree Rotation**

A constant-time operation which changes the structure of a BST while maintaining the BST property.

**Theorem:** At worst, $h = 1.44 \log n \quad \rightarrow \quad O(\log n)$ find/insert/delete

**Top-Down Operation**

A top-down tree operation only involves one pass down the tree. This is usually for finding the target node; performing the operation and and tree repairs are constant time.

**Bottom-Up Operation**

A bottom-up tree operation involves a downwards pass to find the node, then an upwards pass to repair the tree after the operation.

AVL trees can easily use bottom-up insertion. First do regular BST insert. Then go back up the tree and look for an imbalanced node; perform a rotation (or double rotation if necessary). These are all that's needed to balance the tree.

A cleverer insertion is top-down. On the pass downward (during the find), keep track of the lowest node with balance $\neq 0$. If the insertion causes the balance property to be violated, this lowest node is the node which requires rotation. By keeping track of it, we don't need a pass up the tree.

Deletion is only possible with bottom-up. First, find and BST delete. Then, go up until an illegal imbalance is found; rotate or double-rotate to fix. Repeat until root.

# 6 Queues and Heaps

**Queue (FIFO)**
An ADT which removes elements in first in, first out order.

**Stack (LIFO)**
An ADT which removes elements in last in, first out order.

**Priority Queue**
An ADT which can only remove and return the minimum element.
The efficiencies of the priority queue operations are shown below for different implementations.

| | **insert(key, val)** | **findMin()** | **removeMin()** |
|---|---|---|---|
| **Unsorted LinkedList** | $O(1)$ | $O(n)$ | $O(n)$ |
| **Sorted LinkedList** | $O(n)$ | $O(1)$ | $O(1)$ |
| **BST** | $O(h)$ | $O(h)$ | $O(h)$ |
| **Binary Heap** | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |

**Binary Heap**
A *complete* BT such that the priority of each node is $\leq$ that of its children (heap property). Duplicates allowed.

- findMin()
  - $O(1)$ lookup of root
- insert(key, val)
  - First satisfy structure property (completeness); insert into leftmost location in lowest level.
  - Then satisfy less restrictive sorting property by percolating up
- removeMin()
  - Replace root with last leaf (retain structure property)
  - Percolate down (retain sorting property)

Since insertion is $O(\log n)$, naïve heapbuilding is $O(n \log n)$. However, if we do all insertions without worrying about the sorting proprety, and then percolate at the end, then we can achieve $O(n + n) = O(n)$ complexity.

Proof that the final percolation is possible in $O(n)$ time: At level $i$, the number of subheaps is $2^i$. But, in the worst case, half of these are already percolating from higher nodes, leaving $2^{i-1}$ subheaps to begin percolating. If a subheap at level $i$ begins percolating down, the number of swaps necessary is $(h - i)$. We will start at level $i + 1$ to account for our assumption that the levels above have already started percolating. We end at level $h - 1$ since there is no percolation at the last level (mathematically, it's irrelevant because the $(h - i)$ term would disappear). Then

$$\sum_{i=1}^{h-1} 2^{i-1}(h - i) = n - h - 1 = O(n).$$

Heapsort is a sort (not stable) in which we build a heap ($O(n)$) and then pop off the minimum element ($O(\log n)$) for each element ($O(n)$) until the heap is empty. The total complexity is $O(n + n \log n) = O(n \log n)$ which agrees with the theorem about comparison-based sorting.

# 7 Red-Black Trees

**Red-Black Tree**

Guibas & Sedgewick, 1978

A self-balancing BST such that

1. each node is designated either 'red' or 'black'
2. root is black
3. any red node must have black children
4. every path from a node down to a null reference has the same number of black nodes

Here, red represents imbalance, imperfection; rule 3 limits this. Without red, rules are too constraining; insert and delete are $O(n)$ to maintain completeness (rule 4).

**Top-Down Insertion**

Use color swaps on the way down to minimize red on the path; insert red leaf; check for red-red conflicts and rotate if necessary. These rotations aim to throw red over to the other side of the subtree, in order to distribute the imbalance.

**Color Swap**

A black node with two red children is changed to a red node with two black children.

Preserves rule 4; if parent is red, violates rule 3, will need fixup.

Diagrams here

**Top-Down Deletion**

Possible and fast, but messy.

**Bottom-Up Deletion**

Slower but cleaner. First BST deletion swap, such that target node $t$ is either a leaf or a node with one leaf child $a$ (if $t$ is leaf, then $a$ is black since null is black). Then:

- Either $t$ or $a$ is red: snip/splice, keeping black and removing red
- Both are black: Remove value, leaving 'double-black' shell
  - While shell isn't root, perform rotations to push shell up to root
  - If shell encounters a red, in some cases it can 'swallow' the red, removing the shell and turning the red into black.
  - If shell at root, perform rotation to allow snip at root; reduce black height everywhere

# 8 Amortized Analysis

AVL trees and red-black trees insist on always being balanced. However, this is a strong condition; if we can show that *on average* the cost of an operation is low, this is a weaker condition to satisfy.

**Amortized Analysis**

Show that expensive operations occur rarely. In particular, show that the an upper bound of actual cost is actually, on average, inexpensive.

Consider an empty stack with multipop(k) which is $O(k)$ if $k < n$. We will use three methods to independently derive that the amortized cost of any operation is $O(1)$.

## 8.1 Aggregate Method

Here, the goal is to aggregate the total cost for a sequence of $m$ operations, then divide the cost by $m$ to find the average cost per operation. Observe that:

1. the number of pops + the total number of items removed by multipop is always $\leq$ than the number of pushes.
2. The number of pushes must be $\leq$ the total number of operations $m$
3. And by (1), the number of pops and multipops $\leq m$
4. So a sequence of $m$ operations must have $\leq m + m$ cost $= 2m$ cost
5. Amortized cost per operation $= 2m/m = 2$, which is $O(1)$

## 8.2 Accounting Method

Here, we will balance the actual cost of operations with an upper bound 'price'. In other words, we show that our price for each operation is an upper bound of actual cost by showing that we never go into debt. Then, we show that our price is inexpensive.

| | **Actual Cost** | **My price** | **Worst case loss** |
|---|---|---|---|
| **push** | \$1 | \$2 | 0 |
| **pop** | \$1 | \$0 | 0 |
| **multipop(k)** | \$$\min(k, n)$ | \$0 | 0 |

Since every push gives us a revenue of \$2, we can use \$1 towards the actual cost of pushing, and save \$1. Then, when we pop or multipop, each item being popped has an associated dollar saved, which goes towards thhe actual cost of popping them. Thus, we can never go into debt if we start with an empty stack. Finally, notice that our (amortized) price is $O(1)$.

## 8.3 Potential Energy Method

Let $C_i$ be the cost of the $i^{th}$ operation, $D_i$ be the state of the data structure after the $i^{th}$ operation, and $\Phi(D_i)$ be the 'potential energy' stored in $D$ after $i^{th}$ operation. Then the amortized cost of the $i^{th}$ operation is

$$AC_i = C_i + \Delta\Phi = C_i + (\Phi(D_i) - \Phi(D_{i-1})).$$

Note that when we sum this over all operations, we have $AC = C + \Phi$. We can guarantee that total amortized cost is an upper bound for total cost, i.e. $AC \geq C$, if we let $\Phi \geq 0$ always.

For our stack, we will let $\Phi(D_i)$ be the height of the stack, $n$. Then

| | Amortized Cost | Complexity |
|---|---|---|
| **push** | $AC_i = 1 + ((n+1) - n) = 2$ | $O(1)$ |
| **pop** | $AC_i = 1 + ((n-1) - n) = 0$ | $O(1)$ |
| **multipop(k)** | $AC_i = k + ((n-k) - n) = 0$ | $O(1)$ |

# 9 Splay Trees

**Splay Tree**

Sleater & Tarjan, 1985

A BST with no strong structure property; instead, has splay operation which flattens pathological trees over time.

**Move-to-root**

An early attempt to increase efficiency by using rotations to move nodes to root.

However, pathological examples exist where consecutive operations are very expensive.

**Splay operation**

A variation of move-to-root which has inexpensive amortized cost; on average, flattens a pathological tree.

- Case 0: $x$ is root. Done!
- Case 1: $x$ is child of root. Perform zig. Done!
- Case 2: $x$ is inner grandchild of some node $g$. Perform zig zag. Recurse until Case 1 or 0.
- Case 3: $x$ is outer grandchild of some node $g$. Perform zig zig. Recurse until Case 1 or 0.

Then the dictionary operations are:

- find(key)
  - BST find
  - Splay $x$ (or the closest leaf if key not found).
- insert(key, val)
  - Find and Splay
  - Splice new node into root
- delete(key)
  - Find and Splay
  - if key found (at root), delete
  - Find and Splay replacement root (BST delete)
  - $O(1)$ fixup

Notice that the Find and Splay operation is a bottom-up operation. It is possible to make it a top-down operation by cleverly rotating on the way down, keeping track of which nodes will end up in the right subtree and which will be in the left subtree.

Observe that right rotations higher in the tree will throw over larger values than will right rotations lower in the tree; this is analogously true for left rotations. Using this, we can construct the left and right subtrees while we go down the tree, and then do some stitching at the end to create the tree with $x$ at the root.

**Theorem:** For any sequence of $m$ dictionary operations on a splay tree, the amortized cost of each operations is $O(\log n)$.

# 10 B-Trees

Disk operations are very slow. Accessing and manipulating data in memory is orders of magnitude faster than accessing data in the hard disk, due to the high latency between the CPU and disk.

To minimize the number of disk reads, we access chunks of data at a time (rather than bit-by-bit); these chunks are then processed in memory. Similarly, we store data in B-Trees in the disk to minimize the number of disk reads necessary.

**B-Tree**

Bayer & McCreight, 1972

An $m$-ary tree which is optimized for reading and writing large chunks of data. Has the properties:

1. Data items are stored at the leaves
2. Interior nodes contain $< m$ keys to guide searching
3. Root is either a leaf or has between $[2, m]$ children
4. Interior nodes other than root have $[\lceil m/2 \rceil, m]$ children
5. All leaves have the same depth
6. All leaves have $[\lceil l/2 \rceil, l]$ data items

This creates a very bushy, balanced tree. Because the number of children can be a range, it allows branches to merge and split upon insertion and deletion. A deletion that underfills a leaf can be solved by borrowing data from a neighbor, or merging. An insertion that overfills a leaf can be solved by splitting. Both are recursively true for internal nodes as well. When the root overfills, we split it and add a new root.

# 11 Graphs

## 11.1 Graph Traversal

**Graph**
A set $V$ of vertices and a set $E$ of edges connecting them.

**Dense Graph**
$|E|$ is $\Theta(|V|^2)$

**Size of a graph**
$|G| = |E| + |V|$

**Adjacency Matrix**
Graph implementation good for dense graphs. Low overhead, fast lookup.

**Adjacency List**
Graph implementation which is better for sparse graphs, with fast lookup only if list is hashed.

**Breadth-First Search**
A traversal method where neighbors are added to a Queue, and visited in FIFO order. A node is never revisisted.

**Depth-First Search**
A traversal method where recursion is used (LIFO order). A node is never revisited.

**DFS number**
Numbering vertices by the order in which they're visited in DFS.

**Detecting Cycles**
Number vertices by DFS order. A *back edge* is an edge from a higher DFS number to a lower number; they create cycles.

**Bi-Connected Graph**

For every triplet of vertices $\{u, v, w\}$ there exists a path between $u$ and $v$ that doesn't pass through $w$. There are no articulation points.

**Low number**

low($v$) = min(dfs($v$), dfs($x$), low($w$))
where $x$ is ancestor of $v$ with back edge $(v, x)$, $w$ is any descendant of $v$
Post-order traversal for computing low numbers.

**Articulation Point**

A node which, when removed, disconnects the graph.
If the root has $\geq 2$ dfs children, it is an articulation point.
Non-root vertex $v$ is AP iff it has a direct child $w$ with low($w$) $\geq$ dfs($v$)

The low number calculates the highest reachable level (given any number of downward steps but only one backedge). If a vertex has a child which cannot reach further back than the vertex itself (i.e low num of child $\geq$ dfs of vertex) then removing that vertex must disconnect the graph.

## 11.2 Dijkstra's Algorithm

Dijkstra's algorithm in a greedy algorithm for finding the shortest path between two nodes in a weighted graph.

**Algorithm** Dijkstra's Algorithm

```
n ← |V|
Dists[n]                          ▷ shortest distance to each vertex
Weights[n, n]                     ▷ edge weights; unconnected = ∞

S ← {1}                           ▷ visited nodes
for i from 2 to n :
    Dists[i] ← Weights[1, i]      ▷ set initial distances
for i from 2 to n :
    Find w in V − S such that Dist[w] is minimum
    S ← S + w                     ▷ mark w visited
    for v in w.neighbors :
        Dists[v] ← min(D[v], D[w] + W[w, v])   ▷ update dists
```