Algorithms & Data Structures Cheat Sheet

https://medium.com/@alexander.s.augenstein/on-the-road-to-professional-web-development-6817d4cee9

Part I: Algorithms & Data Structures

This topic may span semesters worth of study for undergraduate software engineering or computer science majors. Without prior exposure, I'd recommend watching the lecture videos for MIT 6.0001 and MIT.6.006. It's not unreasonable to absorb the content of each class in only a few weeks. Algorithms research will always remain a topic of current interest in research, but we don't need to dive into the darkest depths of the field to have a solid grip on what we'll need to know to have successful careers as developers.

Deep knowledge of data structures and algorithms may be used rarely, if ever, in a career as a developer. In practice, using a proven solution is often preferable to authoring our own. But for the purposes of an interview, new programmers typically aren't expected to know the nuances of specific languages, tools, or technologies, so these topics offer a good source of puzzles and brain teasers that candidates can prepare for. Beyond the interview, the concepts covered in this post form the foundation of a vocabulary shared by many developers.

We'll explore some of the topics listed below (derived from typical CS curricula), but not all are required for all dev careers. Having prior background in the first half of this list will be of assistance for the remainder of this post:

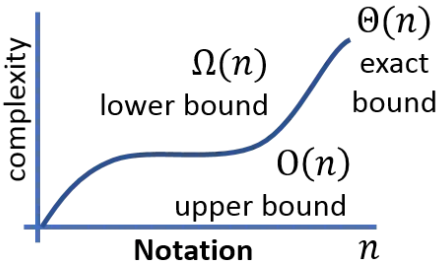
- Set theory (including N0, N1, Z, Q, R, C, cardinality)
- Functions, functionals
- Combinatorics, counting probabilities, probability distributions
- Recursion (and how it relates to proof by logical induction)
- Logic, K-maps
- Graph theory
- Complexity (Theta, O, Omega)
- Linear algebra, ODEs
- Matrix / tensor calculus
- Optimization
- Design of experiments (full / fractional factorial, Latin hypercube)
- Select topics from information theory
- Stats and stochastic processes, Markov chains
- Orthogonal functions, encoding / decoding, PDEs

Review content follows.

ASYMPTOTIC COMPLEXITY

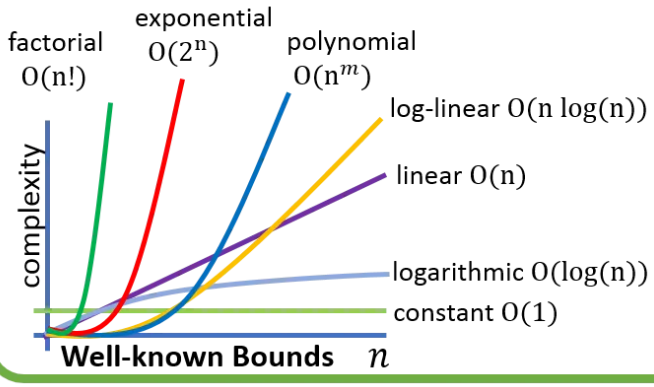
Asymptotic complexity is a measure of performance to compare how well algorithms and data structures scale with increasing input size (denoted **n**). All computations require time and space. Space is commonly measured in terms of memory required. Time isn't measured with a clock due to hardware dependence. Instead, it is measured by the number of operations performed.

Notation for common complexity Measures are summarized to the right (**Big O**, **Big Theta**, **Big Omega**).
It's common to say "Big O" when the performance is actually "Big Theta" (in these cases, both are accurate).



We may be interested in **worst-case**, **best-case**, **average**, or **amortized** performance.
'Average' is single-run performance for a typical input, 'amortized' is averaged over many runs.

Well known complexity bounds for common classes of algorithms are shown below. Others exist.
Faster growth implies higher cost-per-unit-input.



Complexity classes include **P**, **NP**, **ZPP**, **RP**, **BPP**, **BQP**. A problem belongs to a complexity class if it satisfies the requirements for that class. **P** problems can be solved deterministically in polynomial time. **NP** problems can be solved nondeterministically in polynomial time, with solutions deterministically verifiable in polynomial time. A problem is **NP-hard** if it is at least as difficult to solve as the most difficult in NP. NP-hard problems aren't necessarily NP. If a problem is both NP and NP-hard, it is **NP-complete**. P is contained in NP, but a famous open question is whether or not P = NP.

RECURSIVE ALGORITHMS

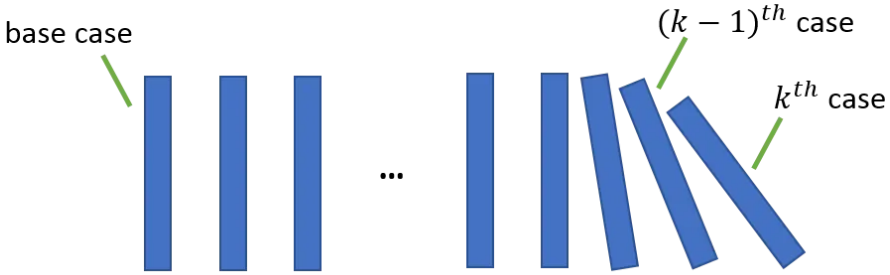
Mathematical proof by induction:

To prove a claim using induction, first demonstrate something is true for the base case.
Next, demonstrate that *if* it were true for some arbitrary case k, then it must also be true for the next case.
Lastly, set the arbitrary case to the base case, which proves it holds for the next case, and the next, and so on.

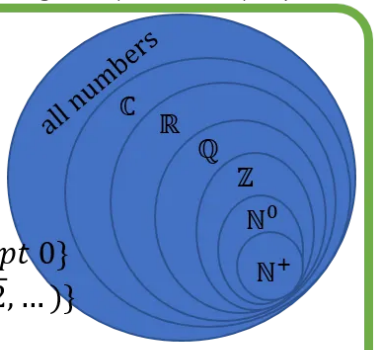
Because induction ties the proof of a known case to the proof of successive cases, it's often visualized as a "domino effect".

Recursive algorithms:

Recursion is similar to (but not the same as) mathematical induction. In induction, we start from the base case and let the blocks fall to the right forever. In recursion, we start at some arbitrary point in the middle and let the blocks fall left until they hit the base case (which solves our problem).



PRIMITIVE DATA TYPES

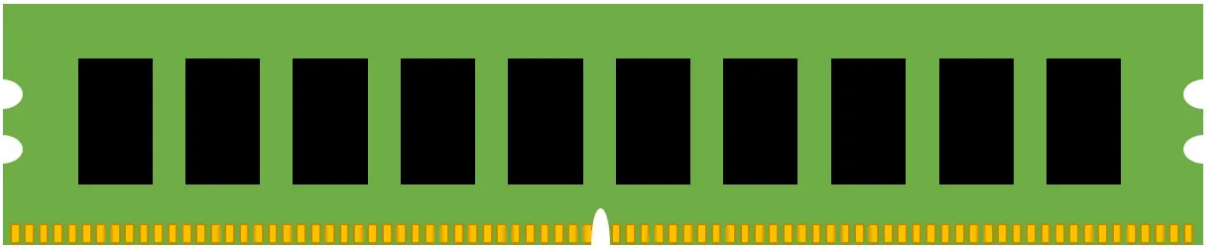


Number sets:
The natural numbers, starting at 1: $\mathbb{N}^+ = \{1,2,3, \dots\}$
The natural numbers, starting at 0: $\mathbb{N}^0 = \{0\} \cup \mathbb{N}^+ = \{0,1,2, \dots\}$
The set of integers (German word Zahlen): $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$
The set of rational numbers (quotients): $\mathbb{Q} = \{integer \div integer\ except\ 0\}$
The set of real numbers: $\mathbb{R} = \mathbb{Q} \cup \{all\ irrational\ numbers\ (e.g. \pi, \sqrt{2}, \dots)\}$
The set of complex numbers: $\mathbb{C} = \mathbb{R} \cup \sqrt{-1}$
Each set listed contains the previous as: $\mathbb{N}^+ \subset \mathbb{N}^0 \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$. Sets containing $\mathbb{C}$ exist.

Data types in computing:
All the sets above are infinite, but computer memory is finite. For programmers, this means:
1. primitive type represented in a computer can only ever be a subset of the above listed sets (e.g. bool= $\{0,1\} \subset \mathbb{N}^0$, uint16 = $\{0,1, \dots, 65\,535\} \subset \mathbb{N}^0$, int16= $\{-32768, \dots, 32767\} \subset \mathbb{Z}$)
2. all types in a computer are finite, so they are **countable**, and can map to a set of integers. (e.g. chars map to ASCII integers, calculations are **finite-precision**, memory pointers are integers)

Representations:
Since all primitive data types can map to a set of integers, and all integers can be represented in any base (**binary** base 2, **octal** base 8, **decimal** base 10, **hex** base 16 being the most common), all computer data can be represented in binary without loss of generality (**WLG**). Memory registers contain one **word** of data (typically 32 or 64 bits). Multiple registers may be used to represent larger numbers (**double** occupies two registers), and more to represent even larger numbers. In all cases, the amount of available memory (registers) eventually runs out.

ARRAY DATA STRUCTURES

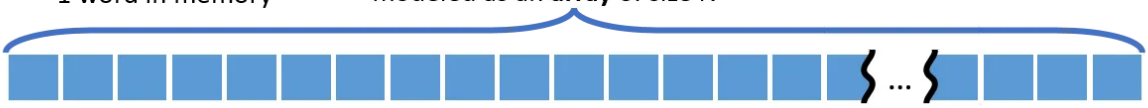


RAM as in **Random Access Memory** (physical hardware)

Implements RAM as in **Random Access Machine** (a model of computation)

= a register that can hold 1 word in memory

RAM that can store N words can be modeled as an **array** of size N



Reading / writing a constant number of words costs: O(1) time

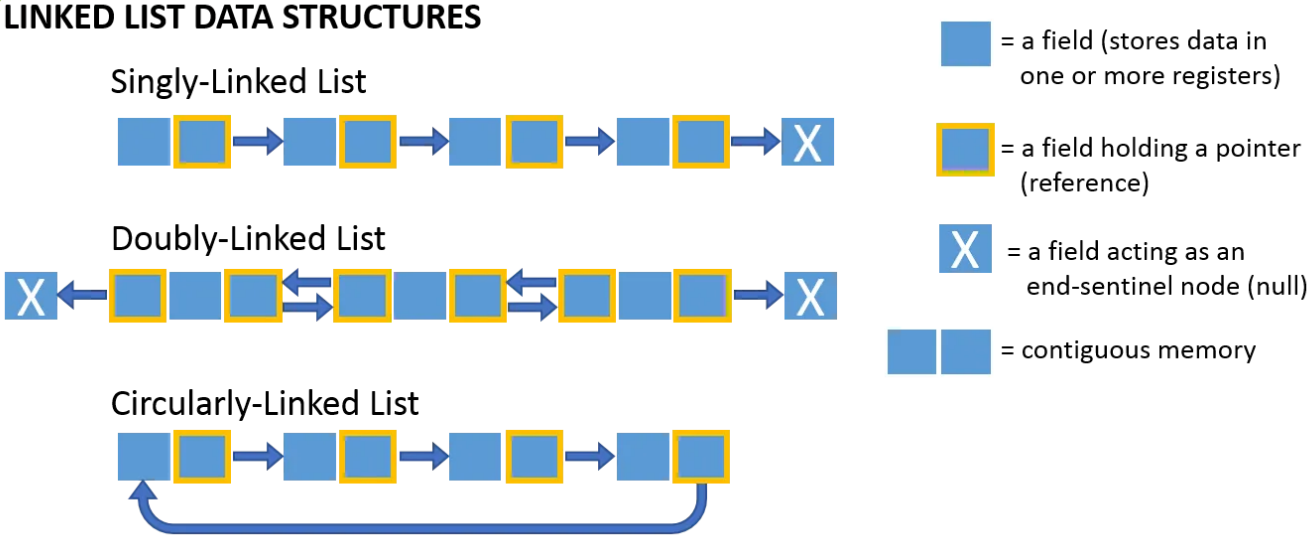
Calculating a constant number of operations costs: O(1) time

Static arrays (shown above) aren't resizable, are pre-allocated, and waste 0 space

Dynamically allocated arrays are static arrays that are allocated at runtime

Dynamic arrays are resizable at runtime in O(1) time, middle-insertion costs O(N) time, space wasted is O(N)

LINKED LIST DATA STRUCTURES



Lists (data structures) implement **pointer machines** (models of computation), often via RAM

Lists don't necessarily store data in contiguous memory, so they don't support random access.
List handles are pointers to the first (and sometimes also the last) element of the list.
End-sentinel nodes are optional. They indicate when the end of the list has been reached.

Indexing costs:	O(N) time
Insertion at beginning (or when location known) costs:	O(1) time
Middle insertion costs:	O(N) time
List concatenation costs:	O(1 + sum of list sizes) time
List sorting costs:	O(N log(N)) time

Space wasted is O(N)

STACK DATA STRUCTURES

Stacks behave as last-in-first-out (**LIFO**),
or equivalently first-in-last-out (FILO)

They are implemented as arrays or singly-linked lists.
Stack overflow is the condition of no more room to **push**

= a field (stores data in one or more registers)

= a field holding a pointer (reference)

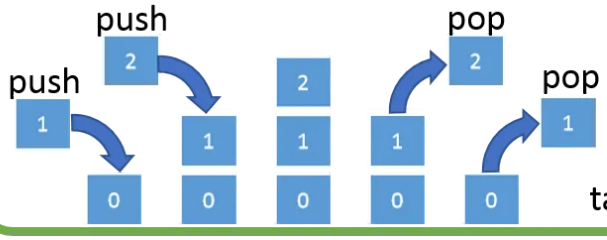
X = a field acting as an end-sentinel node (null)

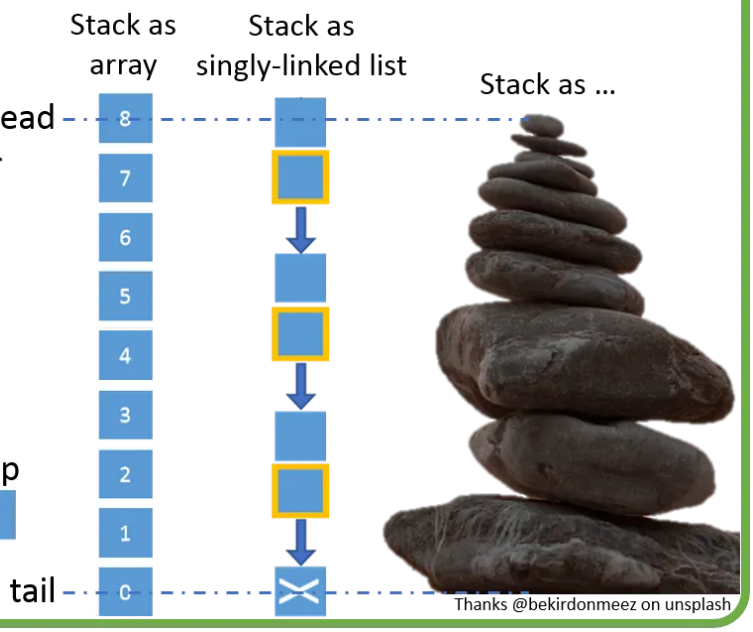
= contiguous memory

The two basic stack operations
are **push** and **pop**, each cost O(1) time

Stacks can be fully described with:
1. contents, 2. maxsize, 3. head pointer

Stacks are used in depth-first-search

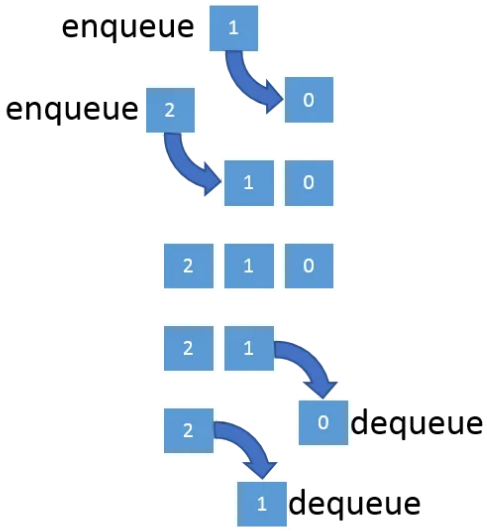




Thanks @bekirdonmeez on unsplash

QUEUE DATA STRUCTURES

Queues behave as first-come-first-served (FCFS),
or equivalently first-in-first-out (**FIFO**),
or equivalently, last-in-last-out (LILO)

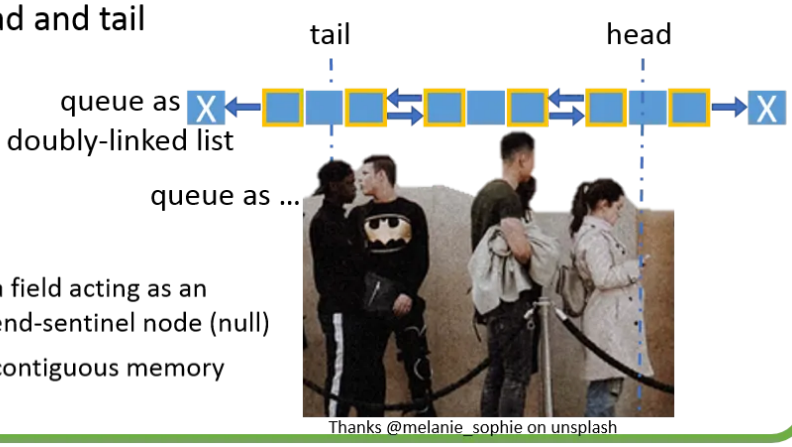


The two basic queue operations
are **enqueue** and **dequeue**,
each costs O(1) time

Linked lists implement queues

Double-ended queues (**deques**)
allow enqueue/dequeue at the head and tail

Queues are used in data buffers
and breadth-first-search



Thanks @melanie_sophie on unsplash

= a field (stores data in one or more registers)

= a field holding a pointer (reference)

X = a field acting as an end-sentinel node (null)

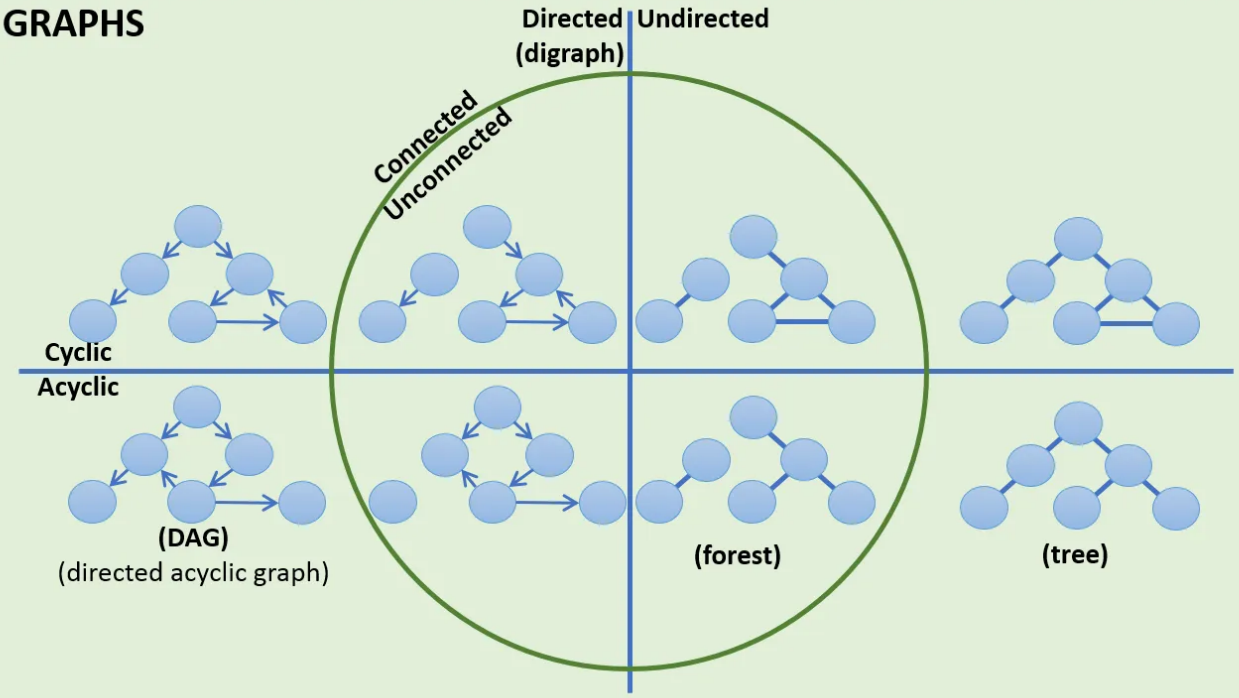
= contiguous memory

GRAPH DATA STRUCTURES

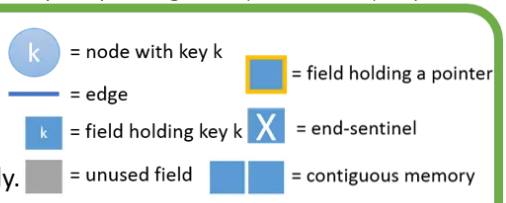
= node
= undirected edge
= directed edge

The **graphs** shown below have a finite number of nodes.
Nodes may contain data such as **keys** or **key-value pairs**.
Edges below are unweighted. **Networks** (**weighted graphs**) have weighted edges.

GRAPHS



BINARY TREE DATA STRUCTURES

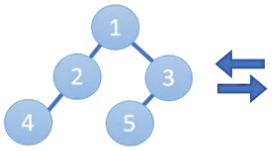


Linked lists can implement binary trees.

Pros: 1. data can be any bytesize, 2. applies to non-binary trees.

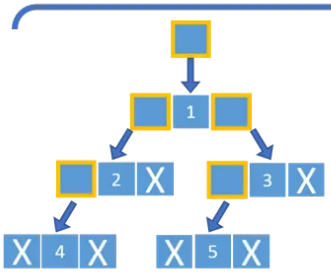
Cons: 1. no random access 2. traversed nodes must be tracked manually.

conceptual representation



Linked-lists implement trees with O(N) memory efficiency

memory representation



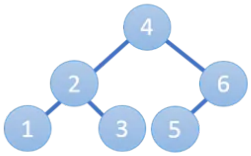
1: 2, 3
2: 4, X
3: 5, X
4: X, X
5: X, X

adjacency list

	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	1	0
3	0	0	0	0	1
4	0	0	0	0	0
5	0	0	0	0	0

adjacency matrix

Arrays memory-efficiently implement **complete binary trees**, since adjacency is implicitly defined.



Implementing binary trees via arrays:

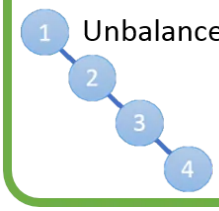
Root key stored in array index $i = 1$,

$parent(i) = floor(i/2)$

$left\_child(i) = 2 * i$

$right\_child(i) = 2 * i + 1$

Unbalanced binary trees as arrays waste polynomial space



HEAP DATA STRUCTURES

Heaps are a specialized **tree** satisfying the max (min) heap property: a nearly-complete binary tree where parent nodes have values greater (less) than their child node(s)

heapify corrects a single violation of max(min) heap property in O(log(N)) time

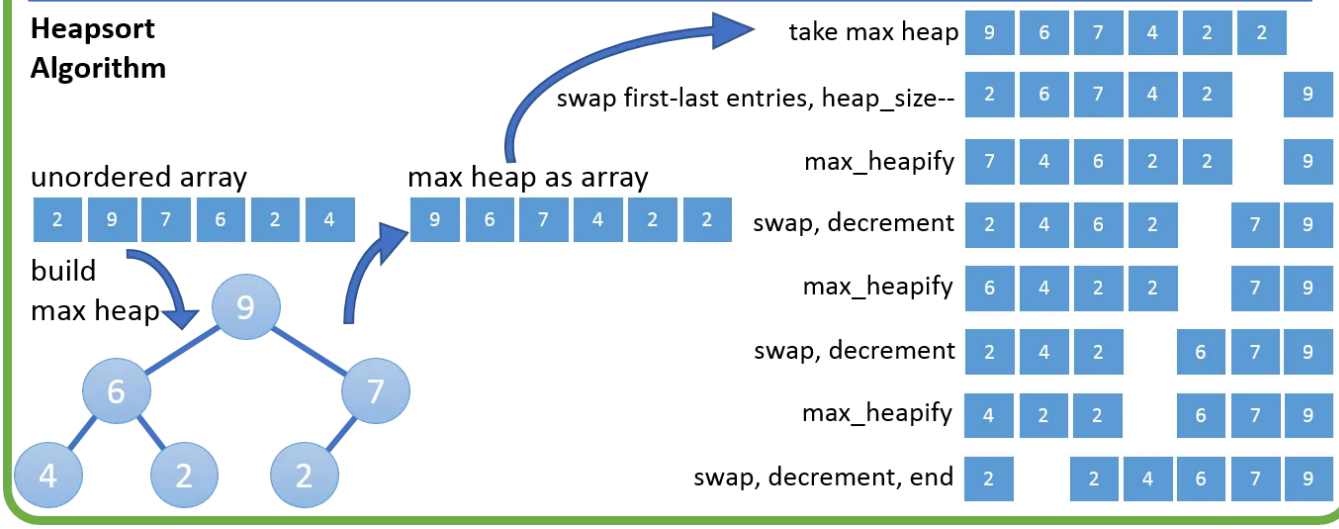
With insertions (filling the tree from left to right) and heapifys, **build_heap** builds a max(min) heap from an unordered array

k = a node with key k

= an edge (connects nodes)

k = an array element holding k

Heaps efficiently implement **priority queues**, queues where nodes with max keys pop first

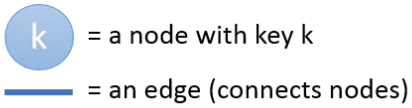


BINARY SEARCH TREE DATA STRUCTURES

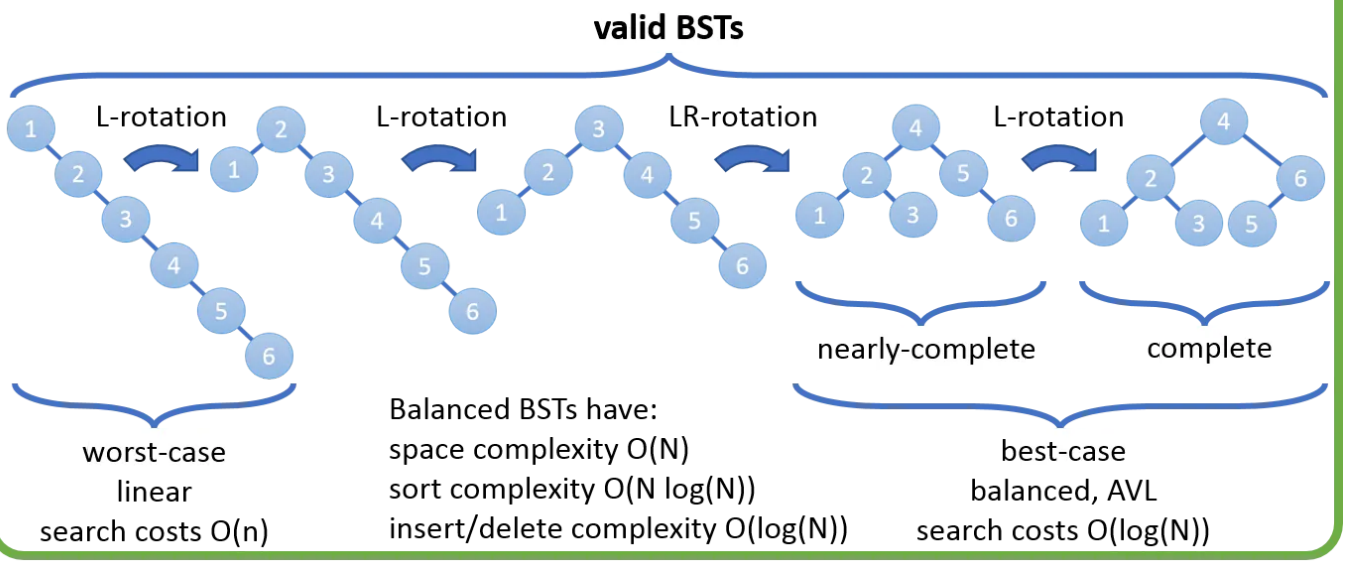
Binary search trees (**BST**) contain nodes such that keys of all left-descendants are ≤, right are ≥

The BST **order relation** does not strictly forbid duplicate keys,
but BST implementations often disallow duplicate keys.

Balance factor = height of left subtree – height of right subtree



AVL trees are a type of BST such that |balance factor| ≤ 1
they are **self-balancing** using only **right and left rotations**



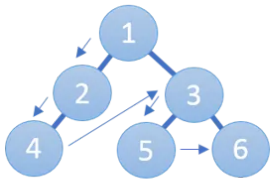
TREE SEARCH ALGORITHMS

BFS & DFS cost: O(|V|+|E|) time, O(|V|) space

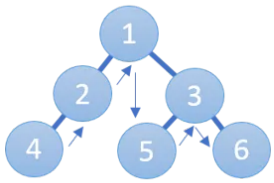
Depth First Search (**DFS**)

DFS supports:

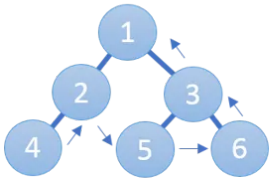
- Pre-order traversal: parent → L-child → R-child



- In-order traversal: L-child → parent → R-child



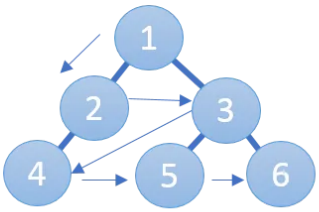
- Post-order traversal: L-child → R-child → parent



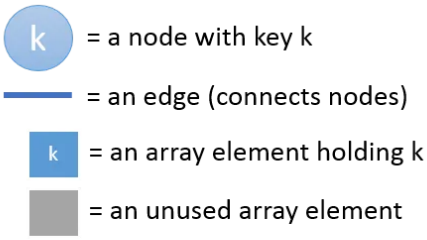
Breadth First Search (**BFS**)

BFS supports:

- Level-order traversal



BFS on an array-implemented BST is trivial (just increment index i)



DFS and BFS require no prior knowledge of tree structure.

Other tree search algorithms exist, notably Monte-Carlo Tree Search (MCTS).

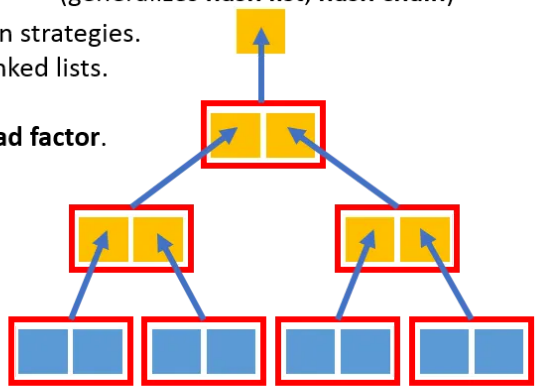
HASH TABLE, ASSOCIATIVE ARRAY, & SET DATA STRUCTURES

Sets are data structures that contain unique elements (e.g. inserting 2 into the set {1,2,3} yields the set {1,2,3}).

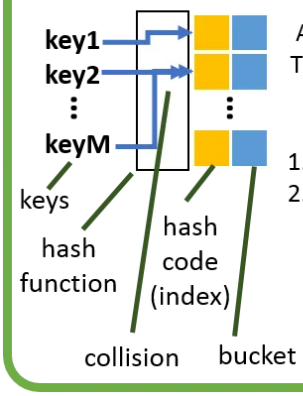
Associative arrays (**maps**) extend sets by mapping keys to values as **key-value pairs**.
Associative arrays that allow one key to map to more than one value are called **multimaps**.

Hash tables efficiently implement associative arrays and sets.
Collisions (multiple keys mapped to same index) are inevitable.
Chaining and **open addressing** (**OA**) are common collision resolution strategies.
When chaining, collisions result in buckets containing pointers to linked lists.
Collisions in OA result in a search for empty buckets via **probing**.
OA has better memory performance but is more sensitive to the **load factor**.
Load factor = keys/buckets. **Dynamic resizing** (e.g. **table doubling**) requires **rehashing**, but can be used to improve the load factor.

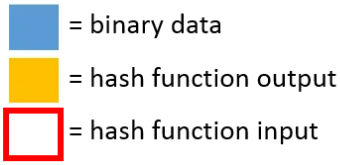
Hash Tree (Merkel Tree)
(generalizes **hash list**, **hash chain**)



Hash Table



Hash functions map a scalar input (binary data) to a scalar output (less binary data, called a **hash**).
Anything stored on a computer is hashable.
The mapping is many-to-one, not unique. By comparing the hash of some data to some other data, we can infer that:
1. the data could be identical (hashes match)
2. the data is not identical (otherwise)

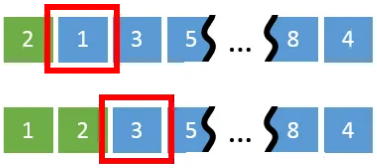


Karp-Rabin Algorithm
Karp-Rabin is to check if substring is in another string. Substring has size 2 in below illustration.

Compute the hash of windowed elements. If it matches the hash of the substring, verify it equals the substring. Else, shift the window to the right and try again.

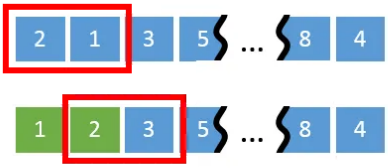
COMPARISON MODEL SORTING ALGORITHMS

Insertion Sort



Read the leftmost unsorted element.
Compare to sorted elements to left, insert.
Repeat. Sorts in-place with time complexity $O(n^2)$.

Bubble Sort

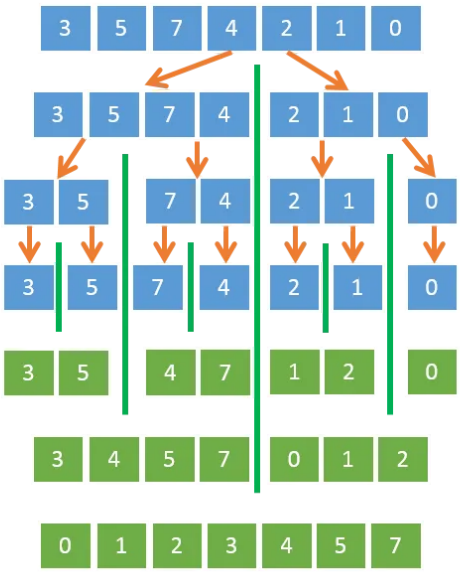


Read the two leftmost unsorted elements.
Swap if needed, shift window, repeat to end of list.
Repeat until sorted. Sorts in-place with time complexity $O(n^2)$.

Quicksort

Easier said than seen: quicksort picks a **pivot** (there are a few methods to do this), then compares it to the rest of the list until all larger elements shift to the right and all smaller shift to the left. After this, the pivot by definition is in its final location in the list. Repeat until the list is sorted.
Quicksort sorts in-place in $O(n \log(n))$ time.

Merge Sort



A “two-finger” sorting algorithm.
Split the list in half repeatedly (un-merging, in a sense).
Sort pairs-of-pairs and repeat until list is merged again.
Time complexity is $O(n \log(n))$.
Does not sort in-place.

= list entry = sorted list entry = sorting window

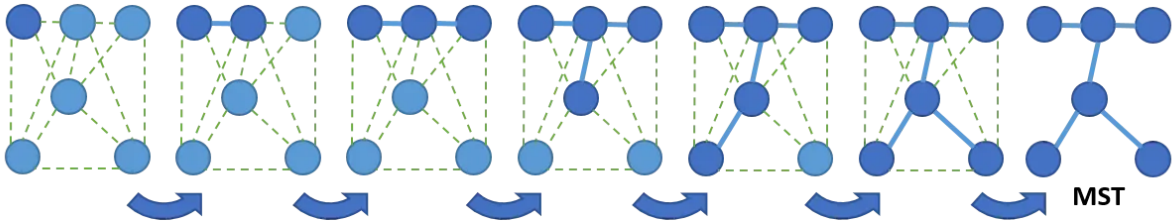
Non-comparison model sorting algorithms exist, notably counting sort and radix sort.

MINIMUM SPANNING TREE ALGORITHMS

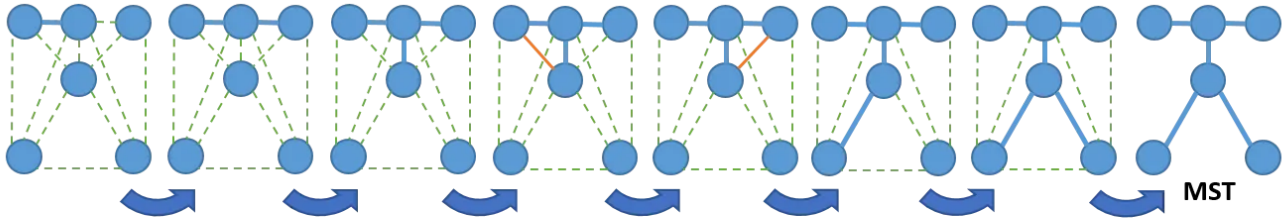
A **spanning tree** is a **subgraph** connecting all nodes (**vertices**) using the minimum possible number of edges.
A Minimum Spanning Tree (**MST**) is a spanning tree that connects all nodes with the minimum weight.

The examples below don't show a numeric weight on each edge, but the cost of traversal is Euclidean distance.
Both are **greedy algorithms**, and cost $O(|E| \log(|V|))$ time.

Prim's Algorithm (initialize by picking any node at random, grow the tree edge by edge)



Kruskal's Algorithm (each node represents a tree. pick the minimum weight edge. if it connects two trees keep it, else delete it. repeat until all edges are gone.)

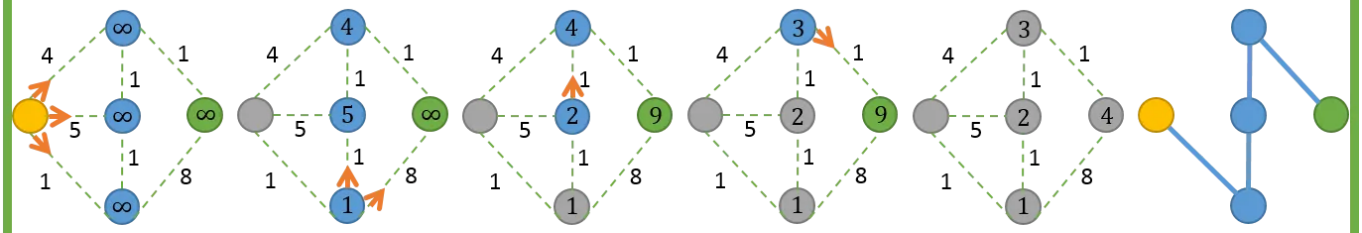


= node (a.k.a. vertex)
= node added to spanning tree (Prim's)
= edge that could become part the spanning tree (Prim's) or spanning forest (Kruskal's)
= edge in spanning tree / forest
= edge selected for deletion (Kruskal's)

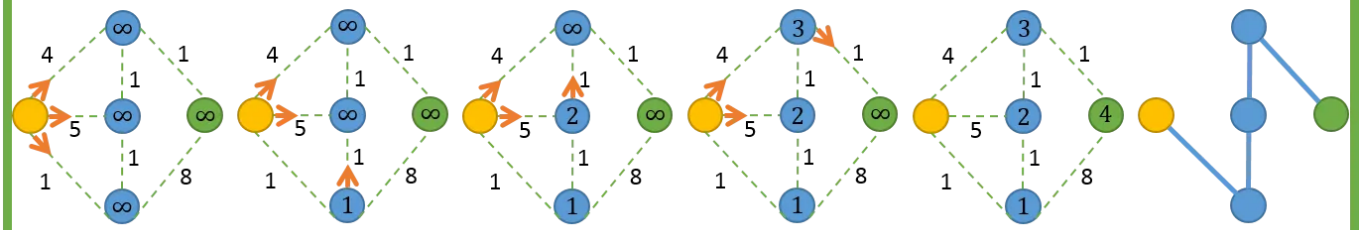
SINGLE-SOURCE SHORTEST PATH ALGORITHMS

The **shortest path problem** is to identify a minimum cost path in a graph from a source node to a destination.

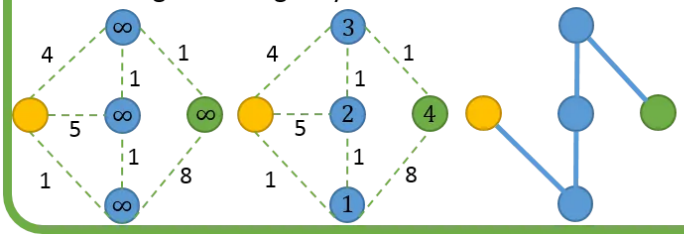
Dijkstra's algorithm uses a priority queue to greedily select the closest vertex that has not yet been processed, and **relaxes** all outgoing edges. Performance is $O(|E| + |V| \log(|V|))$. Does not support negative-weight cycles.

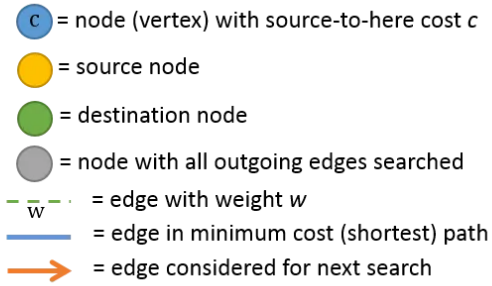


A* algorithm uses edge weights plus a heuristic (often Euclidean distance) to relax edges in **best-first** order. Dijkstra's is a special case. Performance is $O(|E|)$. Does not support negative-weight cycles.



Bellman-Ford algorithm relaxes all $|E|$ edges at most $|V| - 1$ times. Edge selection order matters. Performance is $O(|V||E|)$. Detects negative-weight cycles.





Graphic by asa55.github.io, MIT Licensed, May 2020

DYNAMIC PROGRAMMING

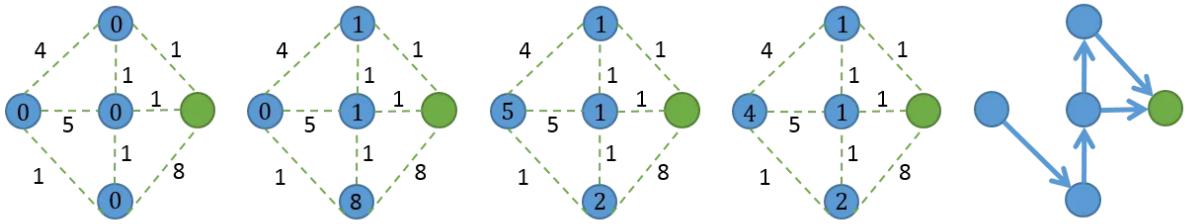
Dynamic programming applies to any problem with **optimal substructure** (optimal solutions to subproblems yield the optimal solution) and **overlapping subproblems** (sub-problems solvable with the same function).
A large class of seemingly exponential problems have polynomial solutions when using dynamic programming.

Dynamic programming solves shortest paths, parenthesization, edit distance, knapsack problems and more.

Steps include:
1. define subproblems
2. guess part of the solution
3. relate subproblem solutions
4. recurse + memoize
5. solve original problem
   (a subproblem or combining
   subproblem solutions)

c = node (vertex) with estimated cost to destination *c*
= destination node
w = edge with weight *w*
= directed edge in optimal policy

Dynamic programming underlies **value iteration**, used to determine an optimal policy from any node to the destination.



If properties of the search space aren't known in advance, **Approximate Dynamic Programming** (**ADP**) may apply.

Part II: Algorithms & Data Structures For Technical Interviews

While Part I covered a broad-brush of the vocabulary relevant to conversational fluency in these topics, the application of these ideas to the interview process and beyond has not yet been addressed.

Interview questions on these topics most frequently arise in the context of a programming language. As such, the implementation will depend heavily on our language of choice. In all cases, the builtin tools we use will have time and memory complexity aspects we should be aware of. Most interview questions will be posed in terms of a highly specific problem (e.g. invert a string, rotate a matrix in-place, etc.), so we'll need to decide which abstract concepts apply. It's worth grinding through practice problems in advance, but memorizing all the answers is both impractical and insufficient for the needs of the interviewer. Remember that these questions can be made arbitrarily easy or difficult on the fly — most interviewers are looking to examine our thought processes, so we need to keep an open dialogue and say what's on our mind. Solving the problem is important, but working with the interviewer to analyze it from multiple angles is

equally important. It's inappropriate to flounder on an implementation of a specific algorithm or data structure of our choosing (once we name an intended solution), but it will always be appropriate to work closely with the interviewer to fully understand the problem.

There are a several interview-appropriate problems that are so famous they have a name. Occasionally you'll hear of interviewers who present these famous problem statements without naming them. While I'd speculate this is uncommon (I haven't personally experienced it at least), if it happens to us we should be mindful that it's our job to call them out by name (and ideally know some properties of optimal solutions). Some of the more popular problems include:

- the Traveling Salesman Problem (TSP)
- the Towers of Hanoi
- the N-Queens Problem
- the M-Coloring Problem
- the Knapsack Problem
- the Fractional Knapsack Problem

The infamous problems listed above are far less common than some of the standard fare used to demonstrate comfort and fluency with a programming language itself. While only tangentially related to the above discussion on algorithms & data structures, it is always a good idea to practice language-agnostic interview questions like the ones listed here.

If you're interested in truly putting the topics we've covered into action, an excellent categorized pool of more difficult algorithm & data structures questions are listed here.