Data Structures

- Contiguously-allocated structures are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.
- Linked data structures are composed of distinct chunks of memory bound together by *pointers*, and include lists, trees, and graph adjacency lists.

Linked Lists

- Singly-linked – each elements links to the next one.
- Doubly-linked – each elements links to the next and previous elements (Java's `LinkedList` is doubly-linked).
- Operations supported: search, insert, delete.
- Cannot overflow (as opposed to static arrays which have a finite length).
- Simpler insertion/deletion than arrays.
- Requires extra memory for pointers than arrays.
- Not efficient for random access to items.
- Worse memory locality than arrays.

Example

- https://github.com/jiangyang5157/golang-start/blob/master/data/list/linked_list.go

Stacks and Queues

- Stacks – LIFO (push, pop)
  - Very efficient, good to use when retrieval order doesn't matter at all (like for batch jobs).
  - LIFO usually happens in recursive algorithms.
- Queues – FIFO (enqueue, dequeue)
  - Average "waiting time" for jobs is identical for FIFO and LIFO. Maximum time varies (FIFO minimizes max waiting time).
  - Harder to implement, appropriate when order is important.
  - Used for searches in graphs.
- Stacks and Queues can be effectively implemented by dynamic arrays or linked lists. If upper bound of size is known, static arrays can also be used.

Example

- https://github.com/jiangyang5157/golang-start/blob/master/data/stack/stack.go
- https://github.com/jiangyang5157/golang-start/blob/master/data/queue/queue.go

Dictionary Structures

- Enable access to data items by content (key).
- Operations: search, insert, delete, max, min, predecessor/successor (next or previous element in sorted order).

- Implementations include: **hash tables**, **(binary) search trees**.

Binary Search Trees

- Each node has a key, and *all* keys in the left subtree are smaller than the node's key, *all* those in the right are bigger.
- Operations: search, traversal, insert, delete.
- Searching is `O(log(n)) = O(h)`, where `h = height` of the tree (`log(n)` for 2 child nodes in each tree root, if tree is perfectly balanced).
- The min element is the leftmost descendant of the root, the max is the rightmost.
- Traversal in `O(n)`, by traversing the left node, processing the item and traversing the right node, recursively ("in-order traversal").
- Insertion is `O(log(n))` and requires recursion.
- Deletion is `O(log(n))` and has three cases (no children, 1 child, 2 children).
- Dictionary implemented with binary search tree has all 3 operations `O(h)`, `h = ceil(log(n))`, smallest when the tree is perfectly balanced.
- In general, a tree's height can go from `log(n)` (best) to `n` (worst), this depends on the order of creation (insertion).
- For random insertion order, on average, the tree will be `h = log(n)`.
- There are balanced binary search trees, which guarantee at each insertion/deletion that the tree is balanced, thus guaranteeing dictionary performance (insert, delete, query) of `O(log(n))`. Such implementations: **Red-Black tree**, **Splay tree** (see below).
- Tree Rotation is an operation on a binary tree that changes the structure without interfering with the order of the elements.
- A tree rotation moves one node up in the tree and one node down. It is used to change the shape of the tree, and in particular to decrease its height by moving smaller subtrees down and larger subtrees up, resulting in improved performance of many tree operations.

Use BST over hash table for:

- Range searches (closest element to some key)
- Traverse elements in sorted order
- Find predecessor/successor to element

Tree Traversal

- DFS (depth first)
  - Pre-order: root, left, right
  - In-order: left, root, right
  - Post-order: left, right, root
- BFS (breadth first)
  - Implemented using a queue, visit all nodes at current level, advance to next level (doesn't use recursion).

Balanced BST

- A self-balancing binary search tree is any node-based binary search tree that automatically keeps its height small in the face of arbitrary item insertions and deletions.

Example

- https://github.com/jiangyang5157/golang-start/blob/master/data/binarysearchtree/bst.go
- **Red-Black Tree**
    - Java's `TreeMap` is a Read-Black tree.
    - Self-balancing is provided by painting each node with one of two colors in such a way that the resulting painted tree satisfies certain properties that don't allow it to become significantly unbalanced.
    - When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties.
    - The properties are designed in such a way that this rearranging and recoloring can be performed efficiently.
    - The balancing of the tree is not perfect but it is good enough to allow it to guarantee searching, insertion, and deletion operations, along with the tree rearrangement and recoloring in `O(log(n))` time.
    - Tracking the color of each node requires only 1 bit of information per node because there are only two colors.
    - Properties: root is black, all null leaves are black, both children of every red node are black, every simple path from a given node to any of its descendant leaves contains the same number of black nodes.
    - From this we get => the path from the root to the furthest leaf is no more than twice as long as the path from the root to the nearest leaf. The result is that the tree is roughly height-balanced.
    - Insertion/deletion may violate the properties of a red–black tree. Restoring the red-black properties requires a small number (`O(log(n))` or amortized `O(1)`) of color changes. Although insert and delete operations are complicated, their times remain `O(log(n))`.

Example

- https://github.com/jiangyang5157/golang-start/blob/master/data/redblacktree/rbt.go
- **Splay Tree**
    - A self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again (they will move nearer to the root).
    - Performs basic operations such as insertion, look-up and removal in `O(log(n))` amortized time.
    - For many sequences of non-random operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown.
    - All normal operations on a binary search tree are combined with one basic operation, called *splaying*.
    - Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree.
    - One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top.
    - Frequently accessed nodes will move nearer to the root where they can be accessed more quickly, an advantage for nearly all practical applications and is particularly useful for implementing caches and garbage collection algorithms.

    - Simpler to implement then red-black or AVL trees.
    - The most significant disadvantage of splay trees is that the height of a splay tree can be linear (in worst case), on average `log(n)`.
    - By performing a splay operation on the node of interest after every access, the recently accessed nodes are kept near the root and the tree remains roughly balanced, so that we achieve the desired amortized time bounds.
    - The splay operation consists of rotating the tree around the accessed node and its parent, and around the parent and the grandparent (depends on the specific structure).

- **AVL Tree**

    - The heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, re-balancing is done to restore this property.
    - Lookup, insertion, and deletion all take `O(log(n))` time in both the average and worst cases.
    - Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.
    - AVL trees are more rigidly balanced than red-black trees, leading to slower insertion and removal but faster retrieval.

Hash Tables

- An implementation of a dictionary.
- A hash function is a mathematical function that maps keys to integers.
- The value of the hash function is used as an index into an array, and the item is stored at that position in the array.
- The hash function `H` maps each string (key) to a unique (but large) integer by treating the characters of the string as "digits" in a `base-α` number system (where `α` = number of available characters, 26 for English).
- Java's `String.hashCode()` function is computed as: `s[0] * 31^(n-1) + s[1] * 31^(n-2) + ... + s[n-1]`.
- This produces a very large number, which is reduced by taking the remainder of `H(S) mod m` (where `m` is the size of the hash table and `H` is the hashing function).
- Two distinct keys will occasionally hash to the same value, causing a *collision*.
- One solution to collision is *chaining* – representing the hash table as a array of `m` linked lists, where each list's items are hashed to the same value.
- A second solution to collision is *open addressing* – inserting an item in its place, or if it's taken, in the next available place. Upon query, after finding the hash, you check if the key is identical. If not, you move to the next cell in the array to check there (the array is initiated with all null values, and moving on to the next cell is done until a null cell is encountered). Upon deletion of an item, all items must be reinserted to the array.
- A third solution is *double hashing* – repeatedly hashing on the first hash until the item is found, an empty cell is found, or the entire table has been scanned.
- All three solutions are `O(m)` for initializing an `m`-element hash table to set null elements prior to first insertion.
- Traversing all elements in the table is `O(n + m)` for chaining and `O(m)` for open addressing (`n` = inserted keys, `m` = max hash table size).
- Pragmatically, a hash table is often the best data structure to maintain a dictionary.

- In Java the size of a `HashMap` is always a power of 2, for the bit-mask (modulo equivalent) to be effective.
- In Java 8 the `HashMap` can store bins as trees in addition to linked lists (more than 8 objects get converted to a red-black tree).

String Hashing

- Rabin-Karp algorithms – a way to find a substr in a string. Compute the hash of the pattern p, and of every substring the length of p in the original string, then compare the hashes. This usually takes `O(n + m)`, unless we have a hash collision in which case we continue to check all collisions by the data itself.
- Hashing is a fundamental idea in randomized algorithms, yielding linear expected-time algorithms for problems otherwise `θ(n * log(n))`, or `θ(n^2)` in the worst case.

Priority Queues

- The heap is a priority queue.
- Provide more flexibility than simple sorting, because they allow new elements to enter a system at arbitrary intervals.
- It is much more cost-effective to insert a new job into a priority queue than to re-sort everything on each such arrival.
- Operations: insert, find-minimum/maximum, delete-minimum/maximum.
- The "priority" criteria can be "length", "size", "id", etc.

Heap

- Heaps are a simple and elegant data structure for efficiently supporting the priority queue operations insert and extract-min.
- They maintain a partial order (which is weaker than the sorted order), but stronger than random – the min item can be found quickly.
- The heap is a slick data structure that enables us to represent binary trees without using any pointers.
- We will store data as an array of keys, and use the position of the keys to implicitly satisfy the role of the pointers.
- In general, we will store the `2^l` keys of the `l`-th level of a complete binary tree from left-to-right in positions `2^(l-1)` to `2^l - 1`.
- The positions of the parent and children of the key at position `k` are readily determined. The left child of `k` sits in position `2k` and the right child in `2k + 1`, while the parent of `k` holds court in position `floor(k/2)`.
- This structure demands that there are no holes in the tree – that each level is packed as much as it can be – only the last level may be incomplete.
- This structure is less flexible than a binary search tree: we cannot move subtrees around easily by changing a single pointer. We also can't store arbitrary tree structures (only full trees) without wasting a lot of space.
- Performing a search in a heap requires linear scanning – we can't do a `log(n)` search like in a binary search tree.
- All we know in a min-heap is that the child is larger than the parent, we don't know about the relationship between the children.

- To insert an item we add it at the end of the array, and then bubble it up if it's smaller that its parent (switch between the parent and the child), until we reach a smaller parent or the root of the tree. This insertion in `O(log(n))`.
- The minimum of the tree is the root. To extract it, we remove the root, and replace it with the last element in the array (bottom rightmost leaf). We then check if it's smaller than both its children. If not, we perform a swap with the smallest child, and bubble the swap down recursively all the way until the criteria is met. This is called *"heapify"*. This operation requires `O(log(n))`.
- Heapify gets the array, index of the node, and the length of the heap.
- To build a heap from an array iterate from the last element to the first and call heapify on each. This costs `O(n)` for a tight analysis (a less tight analysis yields `O(nlog(n))`, which is correct but not tight).

Example

- https://github.com/jiangyang5157/golang-start/blob/master/data/heap/heap.go

Fibonacci Heap

- A more efficient priority queue than binary heaps in terms of time complexity.
- Harder to implement.

Graphs

- A Graph is comprised of vertices `V` (the points) and edges `E` (the lines connecting the points).
- Assume `n` is the number of vertices and `m` is the number of edges.
- Directed graphs: inner city streets (one-way), program execution flows.
- Undirected graphs: family hierarchy, large roads between cities.
- Data structures:
    - **Adjacency Matrix**: we can represent a graph using an `n * n` matrix, where element `[i, j] = 1` if `(i, j)` is an edge (an edge from point `i` to point `j`), and `0` otherwise.
    - The matrix representation allows rapid updates for edges (insertion, deletion) or to tell if an edge is connecting two vertices, but uses a lot of space for graphs with many vertices but relatively few edges.
    - **Adjacency Lists**: we can represent a sparse graph by using linked lists to store the neighbors adjacent to each vertex.
    - Lists make it harder to tell if an edge is in the graph, since it requires searching the appropriate list, but this can be avoided.
    - The parent list represents each of the vertices, and a vertex's inner list represents all the vertices the vertex is connected to (the edges for that vertex).
    - Each vertex appears at least twice in the structure – once as a vertex with a list of connected vertices, and at least once in the list of vertex for the vertices it's connected to (in undirected graphs).
- We'll mainly use adjacency lists as a graph's data structure.
- Traversing a graph (breadth or depth) uses 3 states for vertices: undiscovered, discovered, processed (all edges have been explored).
- Most fundamental graph operation is traversal.

Example

- https://github.com/jiangyang5157/go-graph/blob/master/graph/graph.go

Additional Trees

Trie (Prefix Tree)

- A trie is a tree structure, where each edge represents one character, and the root represents the null string.
- Each path from the root represents a string, described by the characters labeling the edges traversed.
- Any finite set of words defines a trie.
- Two words with common prefixes branch off from each other at the first distinguishing character.
- Each leaf denotes the end of a string.
- Tries are useful for testing whether a given query string `q` is in the set.
- Can be used to implement auto-completion and auto-correction efficiently.
- Supports ordered traversal and deletion is straightforward.
- We traverse the trie from the root along branches defined by successive characters of `q`. If a branch does not exist in the trie, then `q` cannot be in the set of strings. Otherwise we find `q` in `|q|` character comparisons regardless of how many strings are in the trie.
- Tries are space and time efficient structures for text storage and search.
- Very simple to build (repeatedly insert new strings) and very fast to search.
- More memory efficient if there are many common prefixes, great for language dictionaries (many words have same prefix).
- Very quick lookups if key isn't found.

Example

- https://github.com/jiangyang5157/golang-start/blob/master/data/tries/keyword_tree.go

Ternary Search Tree

- A type of Trie where nodes are arranged in a manner similar to a BST but with up to three children.
- More space efficient compared to Trie at the cost of speed.
- Common applications include spell-checking and auto-complete.
- Each node has a single character and pointers to three children: `equal`, `lo` and `hi`.
- Can also contain pointer to parent and flag if it's the end of a word.
- The `lo` pointer must point to a node whose character value is less than the current node (`hi` for higher).
- The `equal` points to the next character in the word.
- Average running time for operations: `O(logn)`; Worst-case: `O(n)`.
- Also relevant: Patricia Tree – A compact representation of a Trie in which any node that is an only child is merged with its parent.

Radix Tree

- A more memory-efficient (compressed) representation of a trie (prefix tree).
- Space-optimized by merging parent that have single-child nodes.

Suffix Tree

- A data structure to quickly find all places where an arbitrary query string `q` occurs in reference string `S` (check if `S` contains `q`).
- Proper use of suffix trees often speeds up string processing algorithms from `O(n^2)` to linear time.
- Suffix tree is simply a trie of the `n - 1` suffixes of an `n`-character string `S` (to construct just iterate over all suffixes and insert them into the trie).
- Suffix tree is simply a trie of all the proper suffixes of `S`. The suffix tree enables you to test whether `q` is a substring of `S`, because any substring of `S` is the prefix of some suffix. The search time is again linear in the length of `q`.

B-Tree

- *Balanced* search trees designed to work well on disks; used by many databases (or variants of B-trees).
- Typical applications of B-trees have amounts of data that can't fit into main memory at once.
- Similar to red-black trees but they are better at minimizing disk IO.
- All leaves are at the same depth in the tree.
- Nodes may have many children (1s-1000s).
- A node has `k` keys, which separate (partition) the range of values handled by `k + 1` children.
- When searching for a key in the tree we make a `k + 1`-way decision based on comparisons with the `k` keys stored at any node (`k` can differ between nodes).
- Nodes are usually as large as a whole disk page to minimize the number of reads/writes.
- The root of a B-tree is usually kept in main memory.

Interval Tree

- An *interval tree* is a red-black tree that maintains a dynamic set of elements, each containing an interval.
- Two intervals `a` and `b` overlap if: `a.low <= b.high` *and* `a.high >= b.low`.
- The key of each tree node is the *low* endpoint of the node's interval (start time) => an in-order traversal lists the intervals in sorted *start time*.
- Additionally, each node stores a value `max` which is the maximum value of *any* interval's end time stored in the *subtree* rooted at that node.
- To find an overlapping interval in the tree with some external interval `i`, we begin to traverse the tree. If `i.low <= root.left.max` we recurse into the left tree; otherwise to the right one. Keep going until one of the intervals overlap or we reach `null`.

DAWG

- Directed acyclic *word* graph.
- Represents the set of all substrings of a string.
- Similar in structure to a suffix tree.

Bloom Filter

- A probabilistic data structure.
- Allows to test if an element is a member of a set.
- False positive matches **are possible**, but false negatives are not.
- A query returns either "possibly in set" or "definitely not in set".