Algorithm Construction

Notation	$f(x) = O(g(x)) \Leftrightarrow \lim_{x \to \infty} \frac{g(x)}{f(x)} > 0$ $f(x) = \Omega(g(x)) \Leftrightarrow \lim_{x \to \infty} \frac{g(x)}{f(x)} < \infty$
Memory Models	Caches and virtual memory can make memory an issue of speed
Strategies	Divide, conquer, combine Greediness: make the choice which seems best right now: problem must have **optimal substructure** (solution for one problem is contained within the solution for larger problems) Dynamic programming: used where subproblems overlap (remembers partial solutions by solving increasingly large problems or by memoisation)
ADTs	May model application with successively decomposed ADTs

Sorting

Selection	Iterates once for every list item: on the ith iteration the next smallest item in the unsorted list is moved to its final position Unstable due to exchanges
Bubble	Iterate over adjacent pairs of items, exchanging them if they are out of order: upon every iteration at least 1 element is moved to its final position Stable
Insertion	Iterates over all list items, and the ith iteration places the ith item in its correct position amongst the first i items This sorting method is stable if you don't sink on equal key values
Shell	Performs a series of "stride s" insertion sorts on list subsets (subset k ($0 \leq k < s$) contains items k, s + k, 2s + k, ..) Try stride sequence $s_{i-1} = 3s_i + 1$ Unstable due to subsetting
Quick	Divide items into two partitions by pivot value, recursively quick sort the two halves and finally join everything together Partition by keeping two pointers: one for an item in the left half greater than the pivot, one for an item in the right half less than the pivot. When you have a pair of valid pointers, swap and iterate until they cross, move in partition Can use logarithmic space if you first recurse on the smaller partition then iterate on the larger In-place partition usually unstable
Heap	Create a max-heap by bottom up heapification, then read items off it to the same array **in place** Unstable
Merge	Mergesort the two halves of the array and then combine the two sorted halves Merge in place by either copying the merged arrays back into the original array or only copy the left sub array to workspace (we will never overwrite the right array before they have been consumed) Stable if we favour first half
Counting	Assume all keys are in range 0 to k – 1, then use items as indexes into an array of that size, incrementing it as a count. When all items are accounted for, translate the counts into indexes into the array that indicate where the last value of that key is to be stored. You can now make a pass backwards through the original array placing each item at the position indicated by the count array, and update the index that array stores appropriately Stable, due to reversed last pass
Radix	A sequence of stable sub-sorts: sort by the digits of the key from least to most significant using e.g. counting sort Stable due to LS digit first
Bucket	If we know that key values are distributed over some range, divide that range into N intervals (buckets), which store linked lists. You can now add items to those linked lists (maintaining sorted order), and do a final pass to read them back out of the buckets

Order Statistics

Quick	Pick an item in the list as a pivot, partition on it then recurse on the appropriate partition

Worst Case Linear

Find the median element of an array storing the medians of each group of 5 elements in the array: this can be used as the pivot
Since this guarantees at least 3N/10 values greater than or less than the pivot we get a linear worst case after the partitioning

Data Structures

SLL

Use a sentinel head node to avoid having to special case insertion

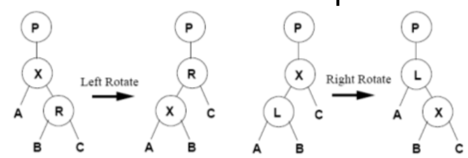
Deque

Superset of queue and stack

Heap

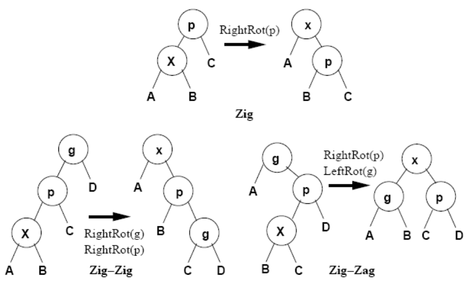
Represented as a complete binary tree (every level filled): every node obeys the heap property
When stored in an array the node at index i has children at 2i + 1 and 2i + 2
Add new items at the end of the array then bubble the new item up by comparing with parent
Remove items by moving the last element in the array into the hole then bubbling it down
Bottom up heapification works by heapifying on successive subtrees from the lowest levels upwards

Trees



Deleting from a BST when there are two non-empty subtrees can be done by moving the smallest node in the right subtree to root and attach the deleted nodes left subtree to it (it can't have one of its own since it's the smallest)

Splay Tree

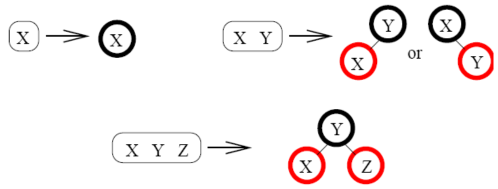


Splay nodes (move them to root) as above upon insertion/search
Number of comparisons required for M insert/searches in a N node tree is O((N+M)log(N+M)), so for M = O(N) have amortised O(log(N)) amortized cost
Cost of any sequence of splay operations is within a constant factor of access to any static tree
Adapt to NU access patterns

2-3-4 Tree

All pointers to empty subtrees are the same distance from the root
To insert, search until we reach a leaf node. Insertion is now trivial unless it's a 4-node: in this case split it into 2 2-nodes (inserting the median value from the 4-node into the parent). This may cause cascading inserts: in the case that it reaches the root, split it and increase the length of both sides
Avoid cascading inserts by splitting any 4-nodes on the way down the tree
To delete, find the node with value to delete. If it is a leaf node, remove it, otherwise find the largest key in its subtree and use it as a replacement. Empty leaf nodes are solved by transferring a key via the parent from a sibling 4-node or by merging a sibling 2 or 3-node with it and taking a key from the parent. This may cause cascading deletions: in the case that it reaches the root, merge the two children of the root and decrease path length

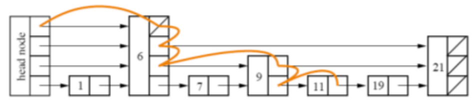
Red-Black Tree



Every red node has a black parent, and there must be an equal number of black nodes on every path from the root to a node with fewer than two children (no greater than a factor of 2 out)
All operations inferable from the 2-3-4 equivalent to the tree

Skip Lists

A linked lists with various heights of nodes that allow you to skip most items when searching



Head node acts as a sentinel with

the maximum possible height
When inserting or deleting, use a helper function which finds the largest node at each level with key less than an argument: this makes the actual operation trivial
Given a number of levels $O(\log(N))$ and random distributions at each level have search time $O(\log(N))$
Grow the maximum height dynamically: add a level whenever $N = 2^{h+2}$
Choose a nodes height by taking advantage of the fact that it requires $\Pr(h=k)=2^{-k}$ and noting that that is the probability of the first k bits or a r.v. being all 0

Hash Table
Use a key value with hash function to index into a hash table
Possible hash functions, M table:
Divide: $h(k) = k \bmod M$
Multiply: $h(k) = \text{flr}[M(kA \bmod 1)]$
Open addressing uses secondary probes to find a new bucket given collision, using a hash function h(k, i) where i is probe number
Linear: $h(k, i) = h'(k) + ci \bmod M$
Leads to primary clustering
Quadratic: $h(k, i) = h'(k) + c1i + c2i2 \bmod M$
Pick c_1 and c_2 carefully to visit every slot: at least coprime to M
Get secondary clustering (keys with same initial hash value have same sequence of probes)
Double Hashing: $h(k, i) = h_1(k) + ih_2(k) \bmod M$
Avoid secondary clustering by picking:
$h_1(k_1) = h_1(k_2) \Rightarrow h_2(k_1) \neq h_2(k_2)$
Deletion hard: use sentinel values, but they accumulate
Keep an eye on the load factor, and allocate a new table when it gets too high (require rehash all)
Better than open addressing is chaining, where you have buckets of items, as in bucket sort earlier

Radix Search
Organises data in a tree, normal comparison is replaced with branching on successive bits of key
No sorted order implied

Trie
Similar to radix tree, but data is stored only in leaf nodes: the path to a leaf node is the shortest key prefix which distinguishes the key from all others in the trie
Sort is just in-order traversal, and a representation is unique
Effective with long keys (strings?), imbalance bounded by key length
To insert, insert node directly (if you find an empty subtree) otherwise insert just enough new internal nodes to disambiguate the new tree item from others
To delete, remove the leaf node and then delete ancestor internal nodes which don't have 2 subtrees
Multi-way tries branch on multiple bits of the key instead of one: better data locality (smaller trees) and search work is not affected unlike 2-3-4 trees (just use key to index into the subtree array)