Binary Search

Key Idea. Decrease-and-conquer algorithm that moves towards the target at each iteration.

Invariant. $A[begin] \leq key \leq A[end]$

BubbleSort

Key Idea. The array is sorted by repeatedly swapping the adjacent elements if they are in a wrong order.

Loop Invariant. At the end of iteration *j*, the biggest *j* items are correctly sorted in the final *j* positions of the array.

InsertionSort

Key Idea. The array is sorted by repeatedly placing the current element *j* into the sorted array *A[1 .. j-1]*

Loop Invariant. At the end of iteration *j*, the first *j* items in the array are in sorted order.

Additional Notes.
1. InsertionSort is very fast on almost sorted array.

Selection Sort

Key Idea. Array is sorted by repeatedly finding the minimum element from the unsorted part and putting it at the beginning.

Loop Invariant. At the end of iteration *j*, the smallest *j* items are correctly sorted in the first *j* positions of the array.

MergeSort

Key Idea. The array is repeatedly divided into half. Each side is recursively sorted. The sorted arrays are then merged.

Space complexity. Merge usually takes O($n \log n$) total space allocated. However, it is possible to do it in O(n) space by reusing a temporary array, switching between the main array and the temporary array.

Optimisations.
1. Since *MergeSort* is slow on small arrays, the base case of the recursion could be another faster sort such as *InsertionSort*.
2. *MergeSort* is stable if the merge operation copies from the left subarray first.

QuickSort.

Key Idea. A pivot is repeatedly chosen. Elements are then partitioned around the pivot. Each side is then recursively sorted.

In-place 3-way One Pass Partitioning: Algorithm.
```
partition(A[1..n], left, right , pIndex)
  pivot = A[pIndex]
  swap(A[1], A[pIndex])
  lessThan = left
  curr = left
  greaterThan = right
  while (curr<=greaterThan)
    if A[curr] < pivot
      lessThan++
      swap(A[curr], A[low])
      curr++
    if A[curr] == pivot
      curr++
    if A[curr] > pivot
      swap(A[curr], A[high])
      high–
```

In-place 3-way One Pass Partitioning: Invariants
1. Each region has its proper elements.
2. At the end of each iteration, the in-progress region is decreased by one.

Choice of Pivot. Random pivot is the best. There are no deterministic bad inputs. Running time is a *random variable*. The *worst case expected* runtime of randomised QuickSort is O($n \log n$)

Randomized Algorithm vs Average-case Analysis. In a randomized algorithm, the algorithm makes random choices. Hence, for every input, there is a good probability of success. In average-case analysis, the environment chooses the random input.

QuickSort Optimisations.
1. Recurse into the smaller half first
   a. Less space on call stack
2. Halt recursion early, leaving small arrays unsorted. Then perform InsertionSort on entire array.

Order Statistic (QuickSelect)

Key Idea. Partition the array but only recurse into the correct half.

Algorithm.
```
Select(A[1..n], n, k)
  if (n==1) return A[1]
  else
    pIndex = randomint(1, n)
    p = partition(A[1..n], n pIndex)
    if (k==p)
      return A[p]
    else if (k<p)
      return Select(A[1..p-1], k)
    else if k>p
      then Select(A[p+1..n], k-p)
```

Time complexity. O(n)

Shuffling

Objective. Every possible permutation has equal chance of being produced.

Fisher-Yates Shuffle: Algorithm. Used in recitation to shuffle any array

```
for i from 0 to n-2 do
  j ← random integer such that i ≤ j ≤ n
  exchange a[j] and a[i]
```

Binary Trees

Definition. A binary tree is either empty or is a node pointing to two binary trees.

BST Property. All in left sub-tree < key < all in right sub-tree

Height.

$$h(v) = \begin{cases} 0 & if\ leaf \\ \max\big(h(v.left), h(v.right)\big) + 1 & otherwise \end{cases}$$

Insertion: Algorithm
```
insert(int key, value)
  if (key<value)
    if (leftTree == null)
      leftTree.insert(key, value)
    else leftTree = new TreeNode(key, value)
  if (key>value)
    ⋮ (mirror)
  else return
```

Insertion: Time Complexity. O(n)

Traversals. Here are few common traversals of a binary tree.
1. In-order-traversal
   a. Left, self, right.
   b. Dot on the bottom
2. Pre-order-traversal
   a. Self, left, right.
   b. Dot on the left
3. Post-order-traversal
   a. Left, right, self.
   b. Dot on the right

Successor: Algorithm
```
successor()
  if (rightTree != null)
    return right.Tree.searchMin()
  TreeNode parent = parentTree
  TreeNode child = this
  while ((parent != null) && (child == parent.rightTree))
    child = parent
    parent = child.parentTree
  return parent
```

Delete: Algorithm
```
delete(v)
  if (v has no children)
    remove v
  else if (v has one child)
    remove v
    connect child(v) to parent(v)
  else if (v has two children)
    x = successor(v)
    delete(x)
    remove v
    connect x to left(v), right(v), parent(v)
```

AVL Trees

Height Invariant. A node v is height-balanced if:

$$|v.left.height - v.right.height| \leq 1$$

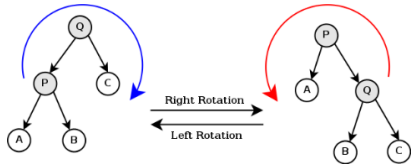
A binary search tree is height-balanced if every node in the tree is height-balanced.

Height of Height-balanced BST. A height-balanced tree with n nodes has at most h < 2log n ↔ A height-balanced tree with height h has at least n > $2^{\frac{h}{2}}$ node.
If n_h is the minimum number of nodes in a height balanced tree of height h, then

$$n_h \geq 1 + n_{h-1} + n_{h-2} \geq 2n_{h-2}$$

Hence, $n_h \geq 2^{\frac{h}{2}}$

Tree Rotations.



Using Tree Rotations during Insertion. Let v be the lowest out-of-balance node after an insertion.
If v is left-heavy:
1. v.left is balanced: right-rotate(v)
2. v.left is left-heavy: right-rotate(v)
3. v.left is right-heavy: left-rotate(v.left)
   right-rotate(v)

If v is right-heavy:
⋮ (mirror left and right)

The *maximum* number of rotations needed during an insertion is 2.

Delete: Algorithm. If v has two children, swap with successor. Delete node v from the binary tree (and reconnect children). For *every* ancestor of the deleted node, check if it is height-balanced and rebalance if needed.
The maximum number of rotations needed during a deletion is O(log n)

Height Optimisation. It is sufficient to store difference in height from parent.

Trie

Trie Trade-offs.

Time	Space
• Tends to be faster when searching • Does not depend on the total text • Does not depend on number of strings	• Tends to use more space • BST and Trie both uses O(text size) space • But Trie has more nodes and more overhead.

Tries are also good from partial string operations such as prefix queries, longest prefix, and wildcards.

Cost for Searching. Depends on the length of the search string. O(L)

Space Complexity. O((size of text) * overhead)

Augmented Search Trees

Features of a Dynamic Data Structure.
1. Maintain a set of items.
2. Modify a set of items.
3. Answer queries.

Basic Methodology for Augmenting Data Structures.
1. Choose underlying data structure.
2. Determine additional info needed.
3. Modify data structure to *maintain* additional info when the structure changes.
4. Develop new operations.

Order Statistics Trees

Augmentation: Size of Subtree in Every Node. By knowing the size of the left subtree, we know the rank of the current node.

Select: Algorithm
```
select(k) // find the node with rank k
  rank = left.weight + 1
  if (k==rank)
    return v
  else if (k<rank)
    return left.select(k)
  else if (k>rank)
    return right.select(k-rank) // note
```

Rank: Algorithm
```
rank(node) // find the rank of a node
  rank = node.left.weight + 1
  while (node != null)
    if node is left child
      do nothing
    else if node is right child
      rank += node.parent.left.weight + 1
    node = node.parent
  return rank
```

Maintaining Weight During Rotations. Since weight can be calculated using a local calculation, it only takes $O(1)$ time

Interval Trees

Interval-Search: Algorithm
```
interval-search(x)
  c = root
  while (c != null and x is not in c.interval)
    if (c.left == null)
      c = c.right
    else if (x > c.left.max)
      c = c.right
    else
      c = c.left
return c interval

Proof:
Claim 1. If search goes right, then there is no overlap in the
left subtree.
Claim 2. If the search goes left, it was safe to go left.
```

Node. Each node stores an interval sorted by the left endpoint.

Augmentation: Maximum endpoint in Subtree. The maximum will ensure that we search in the correct subtree.

Orthogonal Range Searching

Augmentation. Points are only in the leaves. Internal nodes store max of any lead in the left subtree

Query: Analysis. Time complexity of k + $\log n$

(a, b)-trees

(a, b)-Tree Rules. The three rules are as follows.

1. **(a, b)-child Policy.**

Node Type	#Keys		#Children	
	Min	Max	Min	Max
Root	1	$b-1$	2	b
Internal	$a-1$	$b-1$	a	b
Leaf	$a-1$	$b-1$	0	0

2. **Key-ordering.** A non-leaf node must have one more child than its number of keys.
3. **Leaf Depth.** All leaf nodes must all be at the same depth from the root.

Hashing

Load. $\alpha = \frac{number\ of\ items}{number\ of\ buckets}$

Collisions. Collisions are unavoidable. Two distinct keys k_1 and k_2 collide if:

$$h(k_1) = h(k_2)$$

Method 1: Chaining

Key Idea	Each bucket contains a linked list of items
Hash Function	A mapping from the universe of U to a small set of size m. $h: U \to \{1..m\}$
Simple Uniform Hashing Assumption	Every key is equally likely to map to every bucket. Keys are mapped independently
Space Complexity	$O(m+n)$
Insert: Algorithm	Calculate h(key). Lookup h(key) and add (key, value) to the linked list. Worst-case: $O(1)$ Expected: $O(1)$
Search: Algorithm	Calculate h(key). Search for (key, value) in the corresponding linked list. Worst-case: $O(1+n)$ Expected: $O\left(1+\frac{n}{m}\right) \approx O(1)$
Delete: Algorithm	Calculate h(key). Search for (key, value) in corresponding linked list. Delete node. Worst-case: $O(1+n)$ Expected: $O\left(1+\frac{n}{m}\right) \approx O(1)$
(m == n)	We can still add new items to the hash table. Still able to search efficiently ($\approx O(1)$).

Method 2: Open Addressing

Key Idea	If the bucket is full, try again in another bucket.
Hash Function	Hash function will take in the key and the number of collisions that have occurred. $h(key, i): U \to \{1..m\}$ A good hash function must be able to: 1. Enumerate all possible buckets. a. For every bucket j, there is some i such that $h(key, i) = j$. b. In other words, the hash function is a permutation of $\{1..m\}$ 2. Satisfy Uniform Hashing Assumption
Double Hashing	Let $f(k)$, $g(k)$ be two ordinary hash functions. $h(k,i) = f(k) + i \cdot g(k)\ mod\ m$ If g(k) is relatively prime to m, then $h(k, i)$ hits all buckets.
Uniform Hashing Assumption	Every key is equally likely to be mapped to every permutation, independent of every other key.
Space Complexity	$O(m)$
Insert: Algorithm	`insert(key, value)` `for i from 1 to m` `bucket = h(key, i)` `if (table[bucket] is empty or DELETED)` `insert (key, value) into bucket` `throw TableFullException` Worst-case: $O(n)$ Expected: $O(\frac{1}{1-\alpha})$
Search: Algorithm	`search(key)` `for i from 1 to m` `bucket = h(key, i)` `if (table[bucket] == null)` `return key-not-found` `if (table[bucket].key == key)` `return T[bucket]` `throw TableFullException` Worst-case: $O(n)$ Expected: $O(\frac{1}{1-\alpha})$
Delete: Algorithm	`delete(key):` `bucket = search(key)` `bucket.setDeleted()` Worst-case: $O(n)$ Expected: $O(\frac{1}{1-\alpha})$
(m == n)	Table is full and we cannot insert any more items. Can no longer search efficiently. ($O(n)$)

Open Addressing Trade-offs.

Advantages	Disadvantages
Saves space	Sensitive to hash function
Rarely allocate memory	Sensitive to load
Better cache performance	

Hashing in Java

`hashCode()`. Every object supports method `hashCode()` that is inherited from `Object`. `hashCode` has 3 rules.
1. Always return the same value, if the object hasn't changed.
2. If two objects are equal, they return the same `hashCode`.
3. Must redefine `.equals` to be consistent with `hashCode`.

Default Java Implementation of `hashCode`. Returns the memory location of the object. Every object hash to a different location. Hence, you must implement `hashCode()` for your class.

`equals()`. Equals must be reflexive, symmetric, transitive, consistent. Also `x.equals(null) = false` for all object `x`. This function is used to check if 2 keys are equal in the hashMap.

Sorting Summary

Name	Best	Average	Worst	Space	Stable
Bubble	n	n^2	n^2	1	Yes
Selection	n^2	n^2	n^2	1	No
Insertion	n	n^2	n^2	1	Yes
Merge	$n \log n$	$n \log n$	$n \log n$	n	Yes
Quick	$n \log n$	$n \log n$	n^2	1	No

Recurrence Relation Helper.

Given a recurrence relation that looks like

$$T(n) = a\,T\left(\frac{n}{b}\right) + \Theta(n^k(\log n)^i)$$

a	b	k	i	Solution
1	2	0	0	$\log n$
x	x	0	0	n
x	x	1	0	$n \log n$
2	4	0	0	$\sqrt{2}$

Notes.

1. When analysing algorithms with actual numbers, the base case matters.

Rejected Content.

Pancake Sorting

Flips to order. $2 \times (n - 2) + 1 = 2n - 3$. The base case is 2 pancakes, which will take a maximum of 1. All other cases will take a maximum of 2.

In-place Partition: Algorithm(QuickSort)

```
partition(A[1..n], n , pIndex)
  pivot = A[pIndex]
  swap(A[1], A[pIndex])
  low = 2
  high = n + 1
  while (low<high)
    while (A[low]<pivot) and (low < high)
      low++
    while (A[low]>pivot) and (low < high)
      high–
    if (low<high)
      swap(A[low], A[high])
  swap(A[1], A[low-1])
  return low-1
```

In-place Partition: Invariants

1. For all i >= high, A[i] > pivot
2. For all 1 < j < low, A[j] < pivot

Dictionary Interface.

Return	Method	Description
`void`	`insert(Key k, Value v)`	`Insert (k, v) into table`
`Value`	`search(Key k)`	`Get value paired with k`
`Key`	`successor(Key k)`	`Find next key > k`
`Key`	`predeccessor(Key k)`	`Find next key < k`
`void`	`delete(Key k)`	`Remove key k`
`boolean`	`contains(Key k)`	
`int`	`size()`	`Number of (k,v) pairs`

Symbol Table

Return	Method	Description
`void`	`insert(Key k, Value v)`	`Insert (k, v) into table`
`Value`	`search(Key k)`	`Get value paired with k`
`void`	`delete(Key k)`	`Remove key k`
`boolean`	`contains(Key k)`	
`int`	`size()`	`Number of (k,v) pairs`

1. No duplicate keys
2. No *mutable* keys

Expected Maximum Cost of Inserting n Items in a chaining hash table. $O(\log n)$

Performance of Linear Probing. In theory, linear probing is considered slow. If the table is $\frac{1}{4}$ full, there will be clusters of size $O(\log n)$. Hence, operations will also take $O(\log n)$ time.
However real-time performance of linear probing is very fast due to caching. Since it is cheap to access nearby cells, it is common for the whole cluster to be able to fit in the cache. Its real life performance is no better than the wacky probing sequence.

(a, b)-Tree Rules. The three rules are as follows.

4. **(a, b)-child Policy.**

Node Type	#Keys		#Children	
	Min	Max	Min	Max
Root	1	$b-1$	2	b
Internal	$a-1$	$b-1$	a	b
Leaf	$a-1$	$b-1$	0	0

5. **Key-ordering.** A non-leaf node must have one more child than its number of keys.
6. **Leaf Depth.** All leaf nodes must all be at the same depth from the root.

(a, b)-Tree Max Height. $\log_a n + 1$

(a, b)-Tree Min Height. $\log_b n$

(a, b)-Tree Search. $O(\log b \times \log n) = O(\log n)$

(a, b)-Tree Insertion. $O(\log n + b \log n) = O(b \log n) = O(\log n)$

(a, b)-Tree Deletion.

Useful Summations.

$$\sum_{i=1}^{n} i^c = O(n^{c+1})$$

$$\sum_{i=1}^{n} \frac{1}{i} = O(\log n)$$

$$\sum_{i=1}^{n} c^i = O(c^n)$$

$$\sum_{i=1}^{n} \log i^c = O(n \log n^c)$$