Algorithm Bank

USComputing.makeCheatSheet();

contributors

> ['Joshua Chew Jian Xiang', 'Ryan Ch\'ng Wei Han']

# category

<algorithm template>

```
<pseudocode> // time complexity
```

<Description>

Time / Space Complexity

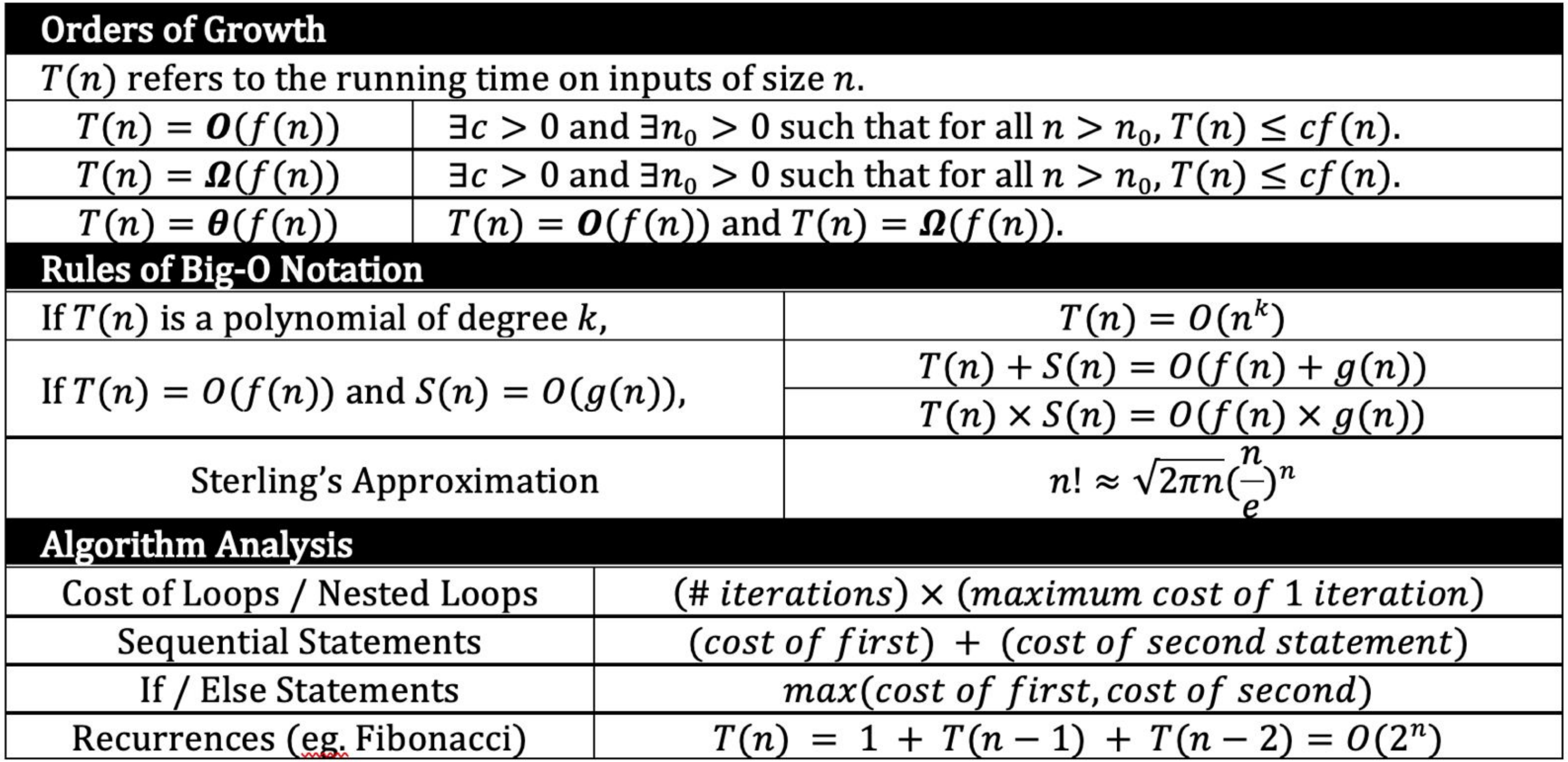
Invariants

[data structure template]

<available implementations>

<Description>

{orders of growth}



Orders of Growth	
$T(n)$ refers to the running time on inputs of size n.	
$T(n) = \boldsymbol{O}(f(n))$	$\exists c > 0$ and $\exists n_0 > 0$ such that for all $n > n_0, T(n) \le cf(n)$.
$T(n) = \boldsymbol{\Omega}(f(n))$	$\exists c > 0$ and $\exists n_0 > 0$ such that for all $n > n_0, T(n) \le cf(n)$.
$T(n) = \boldsymbol{\theta}(f(n))$	$T(n) = \boldsymbol{O}(f(n))$ and $T(n) = \boldsymbol{\Omega}(f(n))$.
Rules of Big-O Notation	
If $T(n)$ is a polynomial of degree k,	$T(n) = O(n^k)$
If $T(n) = O(f(n))$ and $S(n) = O(g(n))$,	$T(n) + S(n) = O(f(n) + g(n))$
	$T(n) \times S(n) = O(f(n) \times g(n))$
Sterling's Approximation	$n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$
Algorithm Analysis	
Cost of Loops / Nested Loops	$(\#\ iterations) \times (maximum\ cost\ of\ 1\ iteration)$
Sequential Statements	$(cost\ of\ first)\ +\ (cost\ of\ second\ statement)$
If / Else Statements	$max(cost\ of\ first, cost\ of\ second)$
Recurrences (eg. Fibonacci)	$T(n)\ =\ 1 + T(n-1)\ +\ T(n-2) = O(2^n)$

{solving recurrences}

Useful Summations:

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

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

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

$$\sum_{i=1}^{n} \log(i)^c \in \Theta(n \cdot \log(n)^c)$$

$$\sum_{i=1}^{n} \log(i)^c \cdot i^d \in \Theta(n^{d+1} \cdot \log(n)^c)$$

$$\sum_{i=1}^{n} \log(i)^c \cdot i^d \cdot b^i \in \Theta(n^d \cdot \log(n)^c \cdot b^n)$$

T(n) = T(n/2) + O(1)	O(log n)	1 + 1/2 + … + 1/n (Harmonic Progression)
T(n) = T(n-1) + O(1)	O(n)	
T(n) = 2T(n/2) + O(1)	O(n)	
T(n) = T(n-1) + O(n)	O(n^2)	1 + 2 + … + n (Arithmetic Progression)
T(n) = 2T(n/2) + O(n)	O(n log n)	

Master Theorem:

If $T(n) \le aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

## <binary search>

```
binarySearch(A, key, n) // O(Log n)
     begin = 0
     end = n-1
     while (begin < end):
          mid = begin + (end-begin)/2;
          if (key <= A[mid]) then
               end = mid
          else begin = mid+1
     return A[begin]
```

From a sorted array `A[1..n]`, returns the element `key` if it is found in the array.

Loop Invariant: `A[begin]` ≤ `key` ≤ `A[end]`

<peak finding for 1D array>

```
findPeak(A, n) // T(n) = O(Log n)
    if (A[n/2] is a peak)
        return n/2
    else if (A[n/2 + 1] > A[n/2])
        findPeak(A[n/2+1..n], n/2) // T(n/2)
    else if (A[n/2 - 1] > A[n/2]) then
        findPeak(A[1..n/2-1], n/2)
```

Returns a local maximum element / peak from some array `A[1..n]`. `A[i]` is a local maximum iff `A[i-1] ≤ A[i] and A[i+1] ≤ A[i].`

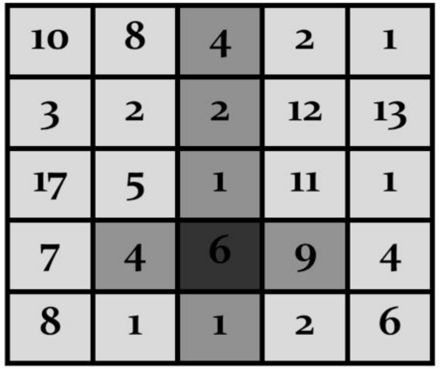
Loop Invariant: There exists a peak in the range `[begin, end]`, and every peak in that range is a valid peak in `[1,n]`.

Key Property: If we recurse on the right half, then there exists a peak in that right half. This can be proven by induction cooker.

<slower peak finding for 2D array>

Given a 2D array `A[1..n,1..m]`, returns a peak that is not smaller than the (at most) 4 neighbours.

```
findPeak() // T(n,m)
     find maximum element of middle column; // O(n)
     if (that element is a peak)
          return that element;
     else if (left neighbour is larger)
          recurse on the left half; // T(n,m/2)
     else if (right neighbour is larger)
          recurse on the right half;
```



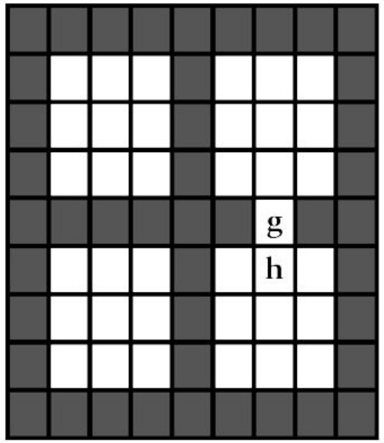
10	8	4	2	1
3	2	2	12	13
17	5	1	11	1
7	4	6	9	4
8	1	1	2	6

Time Complexity: `T(n,m) = T(n,m/2) + O(n) = O(n log m)`

<faster peak finding for 2D array>

Given a 2D array `A[1..n,1..m]`, returns a peak that is not smaller than the (at most) 4 neighbours.

```
findPeak() // T(n,m)
     find max element in the border and cross; // O(n + m)
     if (that element is a peak)
          return that element;
     else // T(n/2, m/2)
          recurse on the quadrant containing the element bigger than max
```

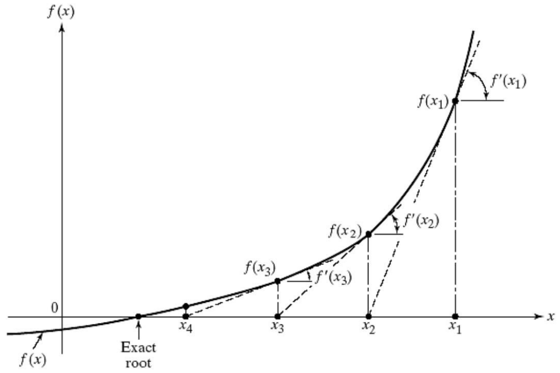


Time Complexity: `T(n,m) = T(n/2,m/2) + cn + cm = O(n + m)`

Proof of Correctness: If we recurse on a quadrant, then the quadrant definitely contains a peak. Every peak in the quadrant is not a peak in the matrix. Recursing in the quadrant would give us a peak at least as large as every element on the boundary.

<newton's method>

```
findRoot(float x, float error)
      diff = error + 1;
      while (diff > error)
            float f = function(x);
            float df = derivative(x);
            float newX = x - (f/df);
            diff = abs(x - newX);
            x = newX;
      return x;
```



This algorithm finds a root of `f(x) = 0` via successive approximation.

After every iteration `i`, we update our value of root x to be

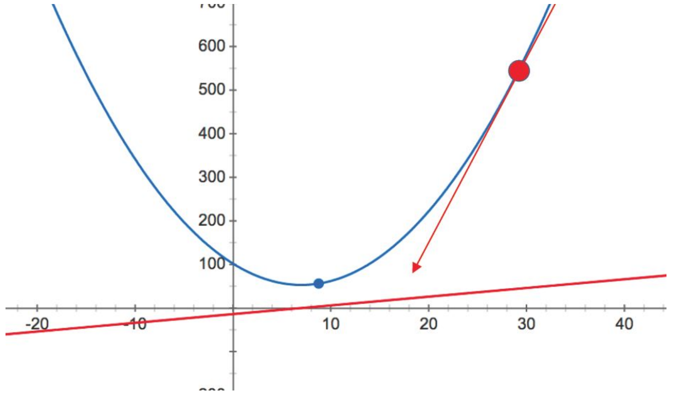
$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Time Complexity: If we have quadratic convergence, which means that at every iteration, the number of correct digits doubles. So, for `d` digits of accuracy, we require `O(log d)` iterations. However, it has more expensive computation.

Conditions for Quadratic Convergence: The function `f` must be continuously differentiable, have a non-zero derivative at root, and have a second derivative at root.

<gradient descent>

```
findMin(float x, float step, float error)
     diff = error + 1;
     while (diff > error)
          float df = derivative(x);
          float newX = x – (step * df);
          diff = |x – newX|;
          x = newX;
     return x
```



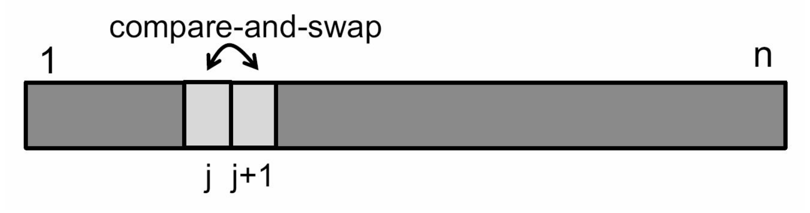
Finds the minimum value of `f(x)` via successive approximation.

Time Complexity: It has more iterations than Newton's method because it has linear convergence, but it has faster computation.

Conditions for Convergence: This algorithm will converge if f is convex and differentiable, and its gradient is L-Lipschitz, and if step size $\boldsymbol{\gamma} \leq 1/L$.

<bubble sort>

```
bubbleSort(A, n)
    repeat (until no swaps): // Worst: n iterations
        for j ← 1 to n - 1 // O(n)
            if (A[j] > A[j + 1])
                swap(A[j], A[j + 1])
```



Advantages: If you are stuck in a crowded ISB and you need to alight ASAP, all you have to shout is “Eh, eh, BubbleSort me in front”.

Time Complexity

- **Worst and Average:** `O(n^2)`
- **Best:** `O(n)` when `A` is already sorted

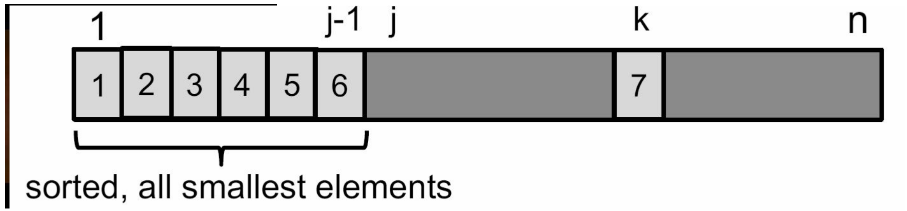
Loop Invariant
At the end of iteration `j` of the outer loop, the biggest `j` items are correctly stored in the final `j` positions of the array.

Stable? Yes
In-place? Yes *// O(1) space complexity*

Note: Coefficient is much worse than other `O(n^2)` algorithms.

<selection sort>

```
selectionSort(A, n) // O(n^2) and Ω(n^2)
     for j ← 1 to n - 1:
          find minimum element A[j] in A[j..n]; // Time: n - j
          swap(A[j], A[k]);
```



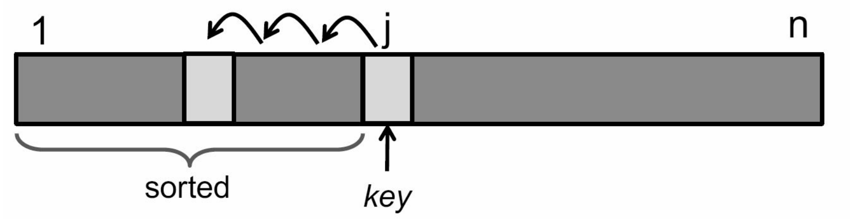
Loop Invariant
At the end of iteration j, the smallest j items are correctly stored in the first j positions of the array.

Stable? No
In-place? Yes *// O(1) space complexity*

Advantages: SelectionSort minimises the number of times data is written (only 2n writes needed). Use SelectionSort if you have very large data to sort - comparing is relatively cheap.

<insertion sort>

```
insertionSort(A, n)
    for j ← 2 to n
        key ← A[j]
        // insert key into sorted array A[1..j-1]
        while (i > 0) and (A[i] > key) // at most j times
            A[i + 1] ← A[i]
            i ← i - 1
        A[i + 1] ← key
```



Time Complexity

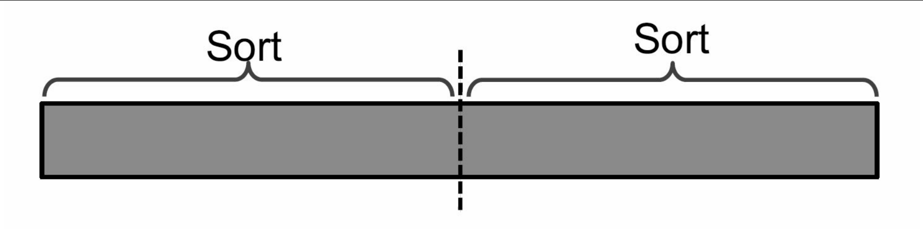
- **Worst and Average:** `O(n^2)`
- **Best:** `O(n)` when `arr` is already sorted

Invariant: `A[1..j-1]` is sorted, ie. the first few elements are *relatively* sorted.

Stable? Yes
In-place? Yes *// O(1) space complexity*

<merge sort>

```
mergeSort(A, n) // T(n)
    if (n == 1) return; // θ(1)
    else
        x ← mergeSort(A[1..n/2], n/2); // T(n/2)
        y ← mergeSort(A[n/2 + 1, n], n/2); // T(n/2)
    return merge(x, y, n/2); // θ(n)
```



Time Complexity:
`T(n) = 2T(n/2) + cn = O(n log n)`

Stable? Yes
In-Place? No, as it requires additional temporary arrays *// additional space complexity of O(n)*

For implementation of linked lists, however, additional space complexity is `O(log n)`.

Disadvantages: MergeSort is slow when there are a small number of items to sort. Hence, at a base case of recursion, use a different sort (eg. InsertionSort for `n < 1024`).

<quick sort>

```
paranoidQuickSort(A[1..n], n) // T(n)
     if (n == 1) then return;
     else
          repeat // expected: executed max 2 times only
               pIndex = random(1, n);
          until p > (1/10)n and p < (9/10)n

          p = partition(A[1..n], n, partitionIndex); // O(n)
          x = quickSort(A[1..p-1], p-1); // T(k)
          y = quickSort(A[p+1..n], n-p); // T(n-k)
```

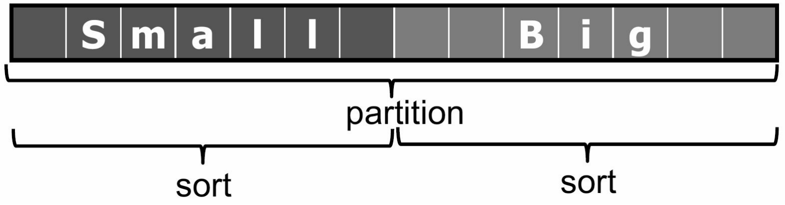
Partition the array into two sub-arrays around a pivot p such that elements in the lower subarray ≤ p ≤ elements in upper subarray. Then, recursively sort these subarrays.

Time Complexity: O(n log n)

Stable? No
In-place? Yes

Choice of Pivot: A pivot is good if it divides the array into two pieces, each of which is size at least n/10.

Improvement: Halt recursion early, leaving small arrays unsorted. Perform InsertionSort on entire array.



<partition>

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

Assuming no duplicates in the array `A`, and `pIndex` is the index of the `pivot`.

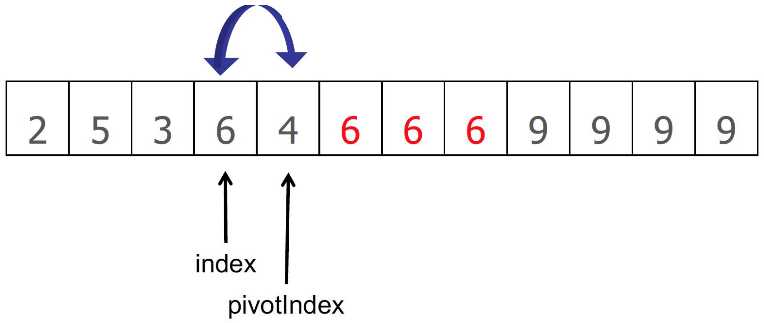
Time Complexity
`O(n)`

Loop Invariant: At the end of every loop iteration, for all `i > high, A[i] > pivot`; for all `1 < j < low, A[j] < pivot`.

<3-way partition>

```
twoPassPartitioning()
        do a regular partition;
        pack duplicates;
```

Three-way partitioning is used when there are duplicate pivots.



<multi pivot quicksort>

```
multiPivotQuickSort(A) // T(n)
    choose k pivots at random;
    sort the pivots; // O(k log k)
    for each item in the array: // n items
        use binary search to identify the right bucket O(log k)
        place item in the right bucket;
    for all the buckets: // k buckets
        multiPivotQuicksort(bucket) // T(n/k)
```

For QuickSort algorithm with k pivots and n items to sort, where $n > k$.

Time Complexity

$T(n) = kT(n/k) + O(n \log k) = O(n \log n)$

Although multi-pivot quicksort runtime gives no improvement at all, in reality, 2 and 3 pivot QuickSort is actually faster than regular QuickSort.

<quick select>

```
quickSelect(A[1..n], n, k)
      if (n == 1) return A[1];
      else
            choose random pivot index pIndex;
            p = partition(A[1..n], n, pIndex);
            if (k == p) return A[p];
            else if (k < p) return quickSelect(A[1..p-1], k);
            else return quickSelect(A[p+1..n], k-p);
```

From an array A of n elements, returns the kth smallest element.

Time Complexity: `O(n)`

Key Point: Only recurses once. Take note that recursing left and right are not exactly the same. But dominated by the O(n) operation that is partition.

<finding minimum using queues>

We will use a Queue ADT which contains pointers which indicate "`valleys`" in the queue. A `valley` is an item that is smaller than everything in the queue after it.

Maintain a secondary linked list between these valleys: each element has a pointer to the `previous` valley and `next` valley. The pointer to the first valley is `minimum`.

```
dequeue()
     if (dequeuing the current minimum)
          minimum = next valley;
          minimum.previous = null;
          remove node from queue;
     else dequeue normally;
```

```
enqueue(x)
     add x to head of the queue; // it instantly becomes a valley
     traverse all previous valleys from the tail of list:
          remove all pointers from the nodes we just traversed;
          if (found valley smaller than x):
               stop and update the next pointer of the node we stopped at to x;

findMin() { return first valley }
```

<knuth shuffle>

```
KnuthShuffle(A[1..n]) // O(n log n)
    for i = 2 to n do
        r = random(1, i)
        swap(A, i, r)
```

Generates a random permutation of an array. It does so by iterating through an array, creating a random prefix.

<height>

```
height()
      leftHeight = -1;
      rightHeight = -1;
      if (m_leftTree != null)
            leftHeight = m_leftTree.height();
      if (m_rightTree != null)
            rightHeight = m_rightTree.height();
      return max(leftHeight, rightHeight)+1;
```

The height of a tree is the number of edges on the longest path from the root to the leaf.

```
h(null) = -1;
h(v) = 0 if v is a leaf;
h(v) = max(h(v.left), h(v.right)) + 1;
```

The largest possible height `h` is `θ(n)`.

Balance:
A BST is balanced if `h = O(log n)`. All operations on such a tree will run in `O(log n)` time.

<search>

```
search(queryKey) // O(h)
     if (queryKey.compareTo(key) < 0)
          if (leftTree != null)
               return leftTree.search(queryKey);
          else return null;
     else if (queryKey.compareTo(key) > 0)
          if (rightTree != null)
               return rightTree.search(queryKey);
          else return null;
     else return this;
```

```
searchMax()
     if (rightTree != null)
          return rightTree.searchMax();
     else return this;

searchMin()
     if (leftTree != null) {
          return leftTree.searchMin();
     else return this;
```

<successor>

```
successor()
     if (this.right != null)
          return this.right.searchMin();
     else
          parent = this.parent;
          child = this;
          while ((parent != null) && (child == parent.right))
               child = parent;
               parent = child.parent;
          return parent;
```

<traversal>

```
inOrderTraversal() // O(n)
     if (leftTree != null) leftTree.inOrderTraversal();
     visit(this);
     if (rightTree != null) rightTree.inOrderTraversal();

preOrderTraversal()
     visit(this);
     if (leftTree != null) leftTree.preOrderTraversal();
     if (rightTree != null) rightTree.preOrderTraversal();

postOrderTraversal()
     if (leftTree != null) leftTree.postOrderTraversal();
     if (rightTree != null) rightTree.postOrderTraversal();
     visit(this);
```

<insertion>

```
insert(x)
     if (x < key) left.insert(x);
     else right.insert(x);
     height = max(left.height, right.height) + 1;
     walk up tree and for every step
          check for balance;
          if out of balance, use rotations to rebalance;
```

For an AVL tree, after every insertion, you will have to update the height and rebalance the tree.

In the worst case, only **two** rotations are required.

<deletion>

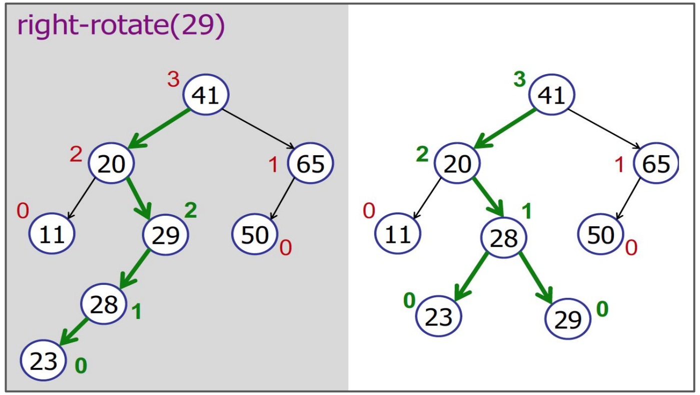
```
delete(v)
     if (v has no children)
          remove(v);
     else if (v has only one child)
          remove (v);
          connect(child(v), parent(v))
     else if (v has two children)
          x = successor(v);
          swap(v, x);
          delete v which is in the original position of the successor;
     update heights up to the root;
     for every ancestor of the deleted node:
          if it is not height-balanced, perform a rotation;
```

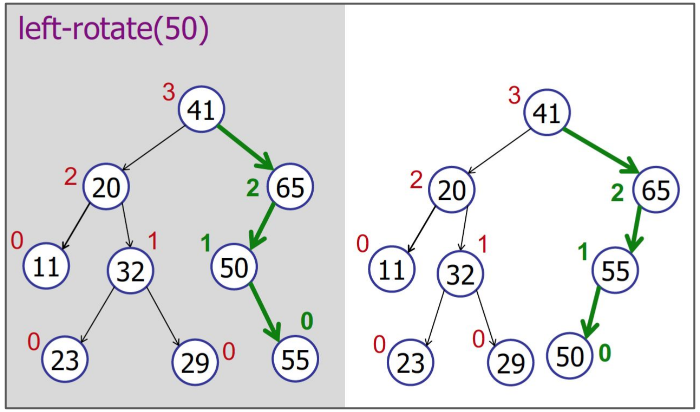
For an AVL tree, deletion can take up to `log n` rotations.

<rotate>

```
rightRotate(node) // left != null
      oldParent = node.parent;
      newParent = node.left;
      grandChild = newParent.right;

      connect(newParent, node);
      connect(team, grandChild);
      connect(oldParent, newParent);
```

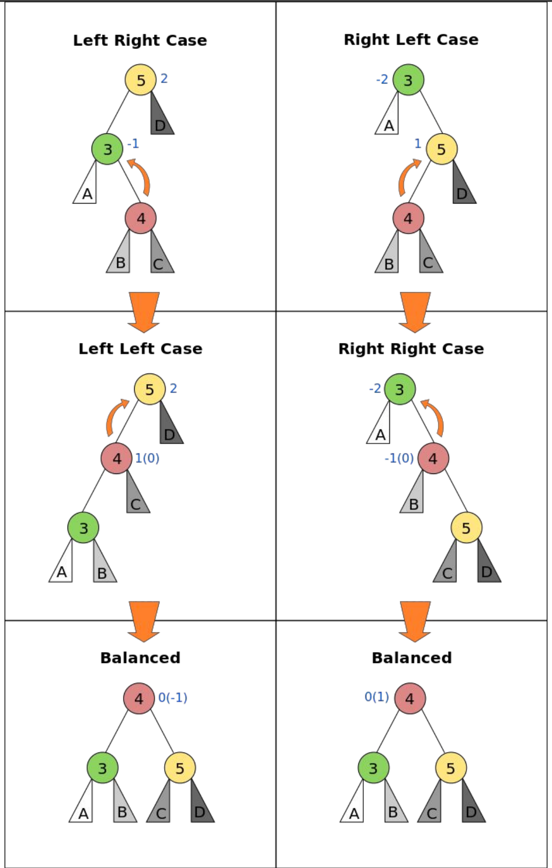
```
leftRotate(node) // right != null
      oldParent = node.parent;
      newParent = node.right;
      grandchild = newParent.left;

      connect(newParent, team);
      connect(team, grandchild);
      connect(oldParent, newParent);
```





<rebalance node>

```
rebalance(node)
      if (node is right-heavy):
            if (node.right is balanced
                  || node.right is right-heavy)
                  leftRotate(node);
            else if (node.right is left-heavy):
                  rightRotate(node.right); leftRotate(node);
      else if (node is left-heavy):
            if (node.left is balanced
                  || node.left is left-heavy):
                  rightRotate(node);
            else if (node.left is right-heavy):
                  leftRotate(node.left); rightRotate(node);
```



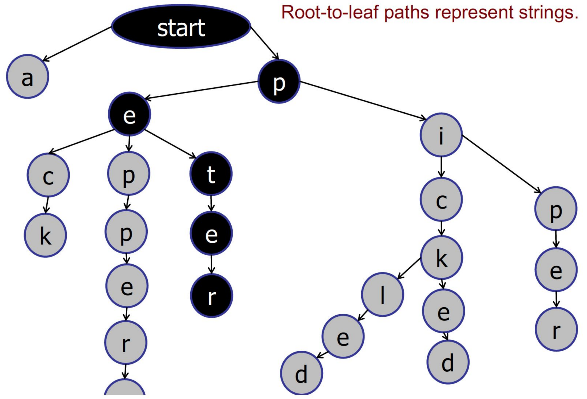
[tries]

In a trie, one letter is stored in each node.
Root-to-leaf paths represent strings.

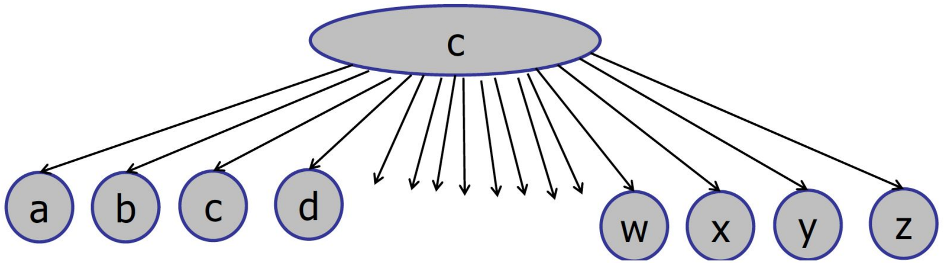
`search(string); // O(L)`

`insert(string);`

Space Complexity:
Storing a trie: `O(size of text * overhead)`



TrieNode children[] = new TrieNode[256];



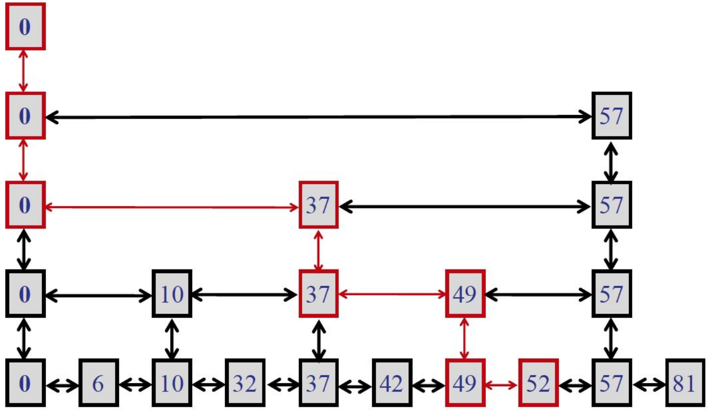
[skip lists]

```
search(key) // O(log n)
          node = root
          while (node.key < key) and (node.level > 1):
                    while (node.next.key < key):
                              node = node.next
                    node = node.down
         while (node.next.key < key)
                    node = node.next
         return node.key == key ? node : null

insert(key) // O(log n)
          k = 0
          while (!done):
                    flip a fair coin:
                              with probability ½ : done = true
                              with probability ½ : k++
```

Every search and insert operation completes in `O(log n)` time with high probability of at least `1 - 1/n`.

In a **skip list** with two lists, the express list skips √n elements per stop, and search takes `O(√n)`. A proper skip list has `log n` levels.

Invariant: The bottom list contains every element, and about half of the elements in level `j` gets promoted to level `j + 1`.

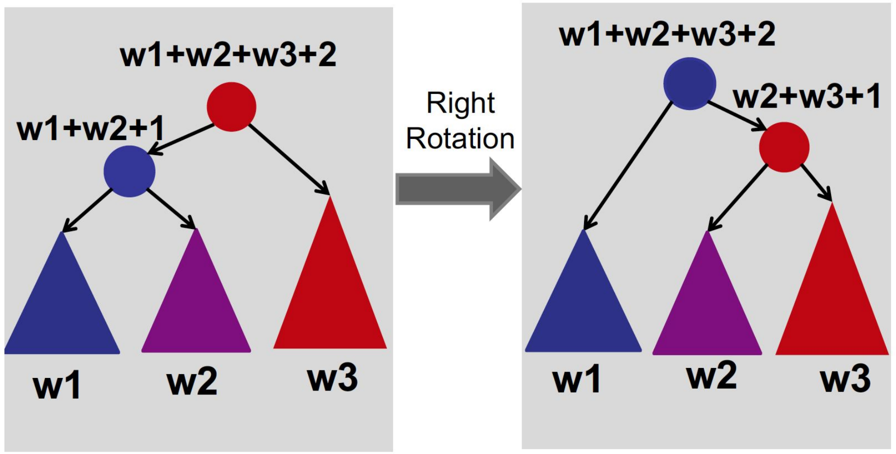


[order statistics trees]

In an **Order Statistics Tree**, we also store the **size** of the subtree in every node. The **weight** of a node is the size of the tree rooted at that node.

```
w(leaf)                  =                  1;
w(v) = w(v.left) + w(v.right) + 1;
```

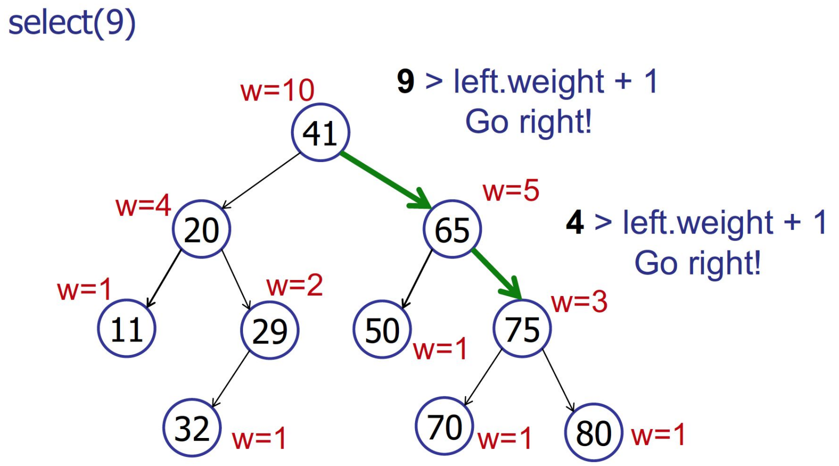
This weight attribute must be maintained whenever a rotation occurs.



<select>

```
select(k)
        rank = m_left.weight + 1;
        if (k == rank) then
                return v;
        else if (k < rank) then
                return m_left.select(k);
        else if (k > rank) then
                return m_right.select(k – rank);
```

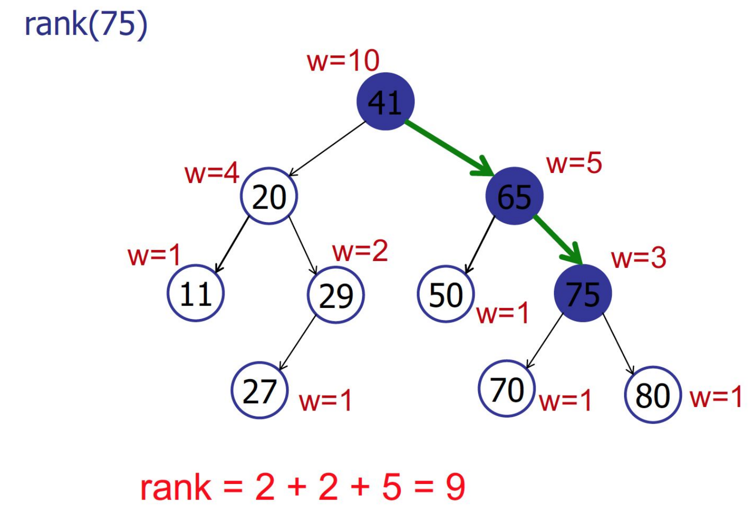
Finds the node of rank k within the tree.



<rank>

```
rank(node)
      rank = node.left.weight + 1;
      while (node != root)
            if (node is left child)
                  do nothing;
            else if (node is right child)
                  rank += node.parent.left.weight + 1;
                  node = node.parent;
      return rank;
```

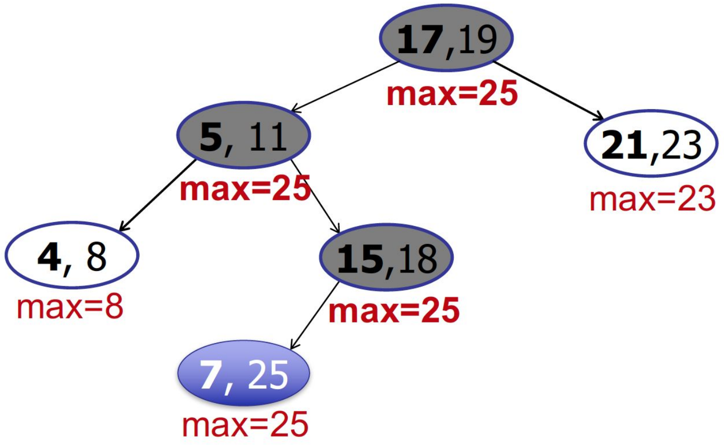
Computes the rank of a node v.



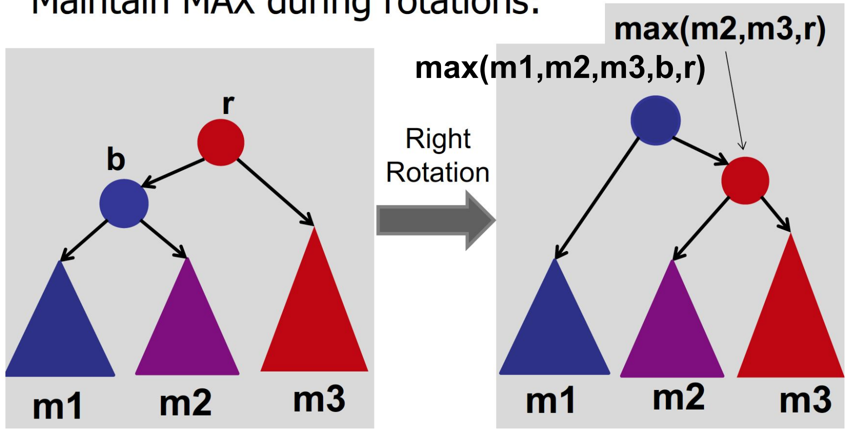
[interval trees]

An **interval tree** is an AVL tree where each node is an interval `(low, high)` and contains the maximum endpoint `max` in its subtree. It is sorted by left endpoint.

For every insertion, the `max` has to be updated every node along the path from the inserted node to the root.



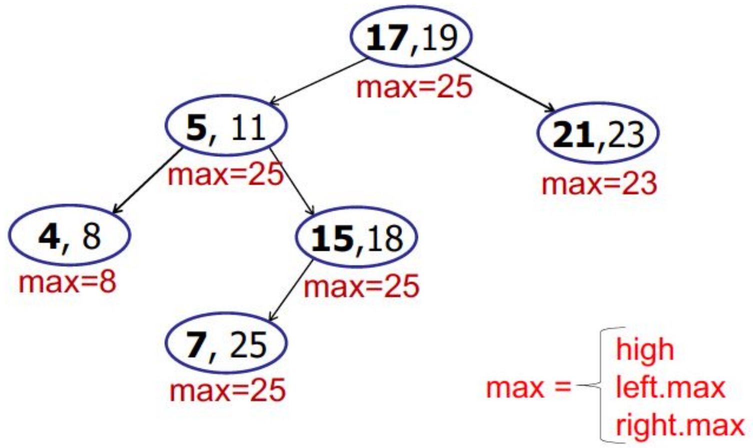
Maintain MAX during rotations:



<interval search>

```
intervalSearch(x) // O(log n)
     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;
```

Finds an interval that contains x.



Correctness: If the search goes right, then there is no overlap in the left subtree → either search finds key in the right subtree or it is not in the tree.

If the search goes left, and there is no overlap in the left subtree → there is no overlap in the right subtree either.

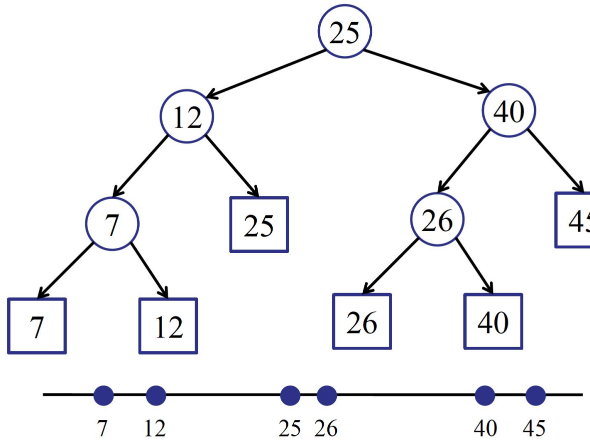
<all overlaps>

```
allOverlaps(x) // O(k log n)
    repeat until no more intervals:
        search for an interval containing x;
        add that interval to list;
        delete that interval;
    repeat for all intervals in list:
        add the interval back in tree;
```

Lists all the intervals that overlap with point `x`. The time complexity if there are `k` overlapping intervals is `O(k log n)`, although the best known solution has a time complexity of `O(k + log n)`.

[range trees]

In a **Range Tree**, we store all points in the leaves, whereas internal nodes only store copies. Each internal node `v` stores the maximum of any leaf in its left subtree.



Preprocessing Time Complexity: `O(n log n)`

Space Complexity: `O(n)`

A one-dimensional range query, **`query`**`(low, high)`, will return all the points in the tree that fall between the range of `low` and `high`.

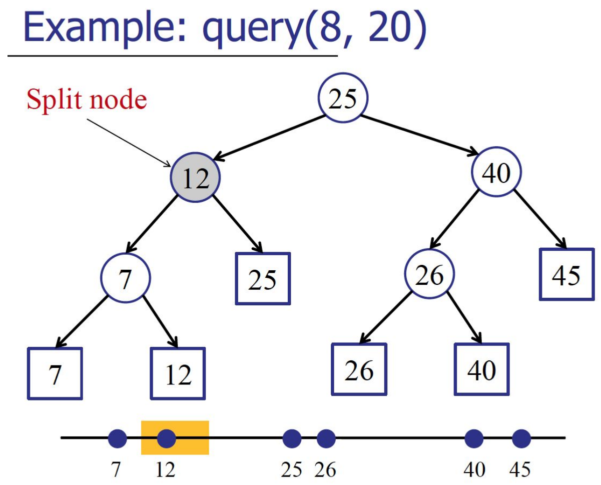
```
rangeQuery(low, high)
      v = findSplit(low, high); // O(log n)
      LeftTraversal(v.left, low, high);
      RightTraversal(v.right, low, high);
```

Time Complexity: `O(k + log n)`, where `k` is the number of points found.

<find split>

```
findSplit(low, high) // O(log n)
      v = root;
      done = false;
      while (!done)
            if (high <= v.key) then v=v.left;
            else if (low > v.key) then v=v.right;
            else done = true;
      return v;
```

A **split node** is the first node from the root in which its point value is between the range query.



<range traversal>

```
LeftTraversal(v, low, high)
     if (low <= v.key)
          all-leaf-traversal(v.right);
          LeftTraversal(v.left, low, high);
     else
          LeftTraversal(v.right, low, high);
```

```
RightTraversal(v, low, high)
     if (v.key <= high)
          all-leaf-traversal(v.left);
          RightTraversal(v.right, low, high);
     else
          RightTraversal(v.left, low, high);
```

For the left traversal, we either
→ output all in the right subtree and recurse left
→ *or* recurse right.

For the right traversal, we either
→ output all in the left subtree and recurse right
→ *or* recurse left.

Time Complexity:
Traversal algorithm recurses at most `O(log n)` times. Outputting all in a subtree has a cost of `O(k)` where `k` is the number of items found.

<number of points in range>

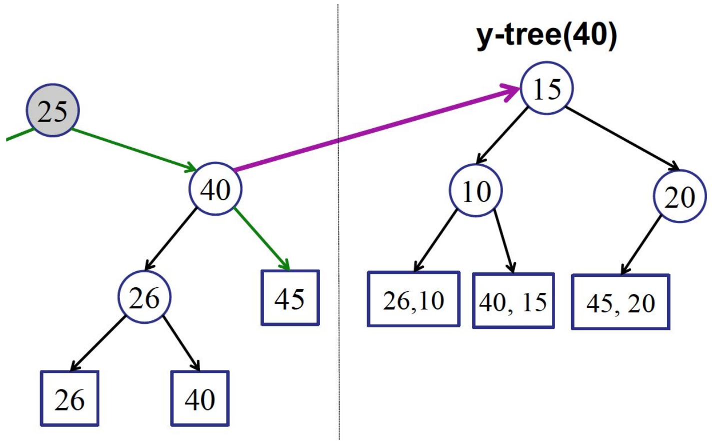
```
LeftTraversal(v, low, high)
      if (low <= v.key)
            total += v.right.count;
            LeftTraversal(v.left, low, high);
      else
            LeftTraversal(v.right, low, high);
```

To just find out the number of points in the range, just keep a count of the number of nodes in each subtree.

[two-dimensional range trees]

For a **two-dimensional range tree,**

1. Create a 1D-range tree (using only x-coordinates) of the points (ie. the `x-nodes`.
2. Store a `y`-tree (using only `y`-coordinates) at each `x-node` containing all the points in the `x-node`'s subtree.



Space Complexity: `O(n log n)`
Each point in the `x-tree` appears at most `O(log n)` `y-trees` (ie. at most one `y-tree` per level). The rest of the `x-tree` takes `O(n)` space.

BuildTree Time Complexity: `O(n log n)`

Inserting / Deleting Nodes: `O(n)`
On every insertion / deletion, you will have to do a rotation, and every rotation requires us to entirely rebuild the `y-trees` for the rotated nodes.

<two-dimensional range queries>

```
2DRangeQuery(low, high)
     v = findSplit(low, high); // O(log n)
     2DLeftTraversal(v, low, high);
     2DRightTraversal(v, low, high);
```

```
2DLeftTraversal(v, low, high)
     if (v.key.x >= low.x)
          ytree.search(low.y, high.y); // O(log n)
          2DLeftTraversal(v.left, low, high);
     else
          2DLeftTraversal(v.right, low, high);
```

Given a 2D array of points, searches for all the points that fall within a rectangular box.

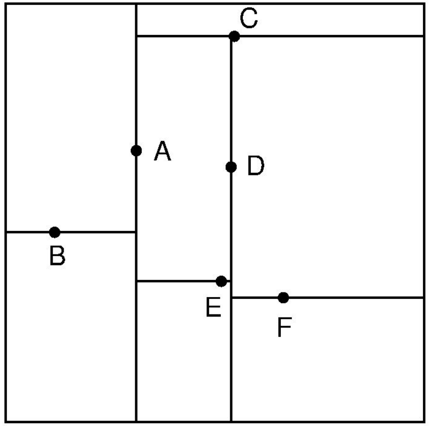
Time Complexity:
O(log n) to find split node
O(log n) recursing steps
O(log n) y-tree searches of O(log n)
O(k) enumerating output

Curse of Dimensionality:
For d-dimensional range trees, recursively construct a (d-1)-dimensional range tree in each node of a 1D-range tree.

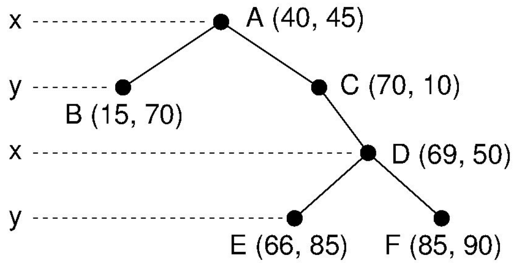
- **Query cost:** O(log^d n+ k)
- **buildTree cost:** O(n log^(d-1) n)
- **Space:** O(n log^(d-1) n)

[kd-trees]

In a **kd-tree**, each alternate level in the tree divides the points on a plane into half either vertically or horizontally.



(a)



(b)

Query Cost: Worst case O(√n), but it sometimes works better in practice for most queries.

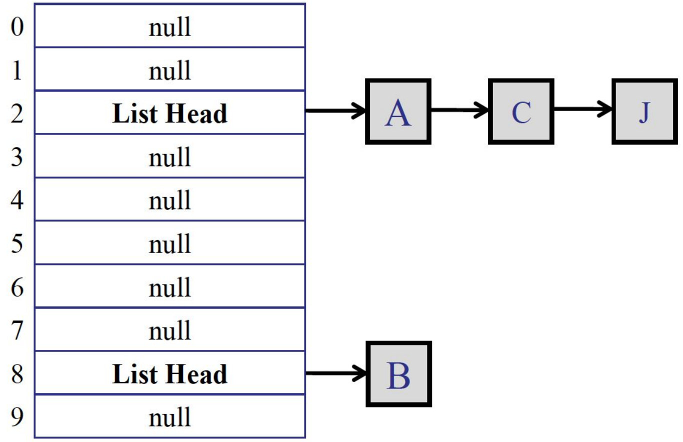
Advantages over 2D range trees: Easier to update dynamically and good for nearest-neighbour queries.

[hash tables with separate chaining]

```
insert(key, value)     // Worst: O(1)
      calculate h(key);
      lookup h(key) in hash table;
      add h(key) to linked list;

search(key) // Worst: O(n), Expected: O(1)
      calculate h(key);
      search for (key,value) in linked list

delete(key) // O(1 + n/m)
      calculate h(key);
      search for (key,value) in linked list
      delete item from linked list
```

Given `n` items, a hash table of these entries would have `m` buckets, each bucket containing a linked list of items to store multiple keys.

Space Complexity: `O(m + n)`



<open addressing>

```
insert(key, value) // Average ≤ 1 / (1 - α)
    int num_collisions = 1;
    while (num_collisions <= m)
        int bucket = h(key, num_collisions);
        if (T[bucket] == null){
            T[bucket] = {key, value};
            return success;
    num_collisions++;
    throw new TableFullException();

delete(key)
    find key to delete;
    remove it from the table;
    set bucket to DELETED;
```

When you insert items, on collision, probe a sequence of buckets in the hash table until you find an empty one.

The **hash function** is redefined as `h(key,num_collisions)`.

Load α of a hash table is the average #items per bucket, i.e. $\alpha = n/m$.

<hash functions>

Linear Probing:

```
h(key, i) = (h(key) + i) % m
```

Modified Linear Probing:

```
h(key, i) = (h(key) + i*d) % m
```

where d > 1 and is coprime to m.

Quadratic Probing:

```
h(key, i) = (h(key) + i^2) % m
```

Double Hashing:

```
h(key, i) = (f(key) + i*g(key)) % m
```

A **hash function** stores a key k into a bucket `h(k)`. A good **hash function** `h(key, i)`, where `i` is the number of collisions, should enumerate all possible buckets and follow the Simple Uniform Hashing Assumption.

Simple Uniform Hashing Assumption: Every key is equally likely to map to every bucket, and keys are mapped independently.

If table is ¼ full, there will be clusters of size `θ(log n)`, but may still be fast due to caching - it is cheap to access nearby cells.

If g(k) is relatively prime to m, then h(k,i) hits all the buckets.

<resizing>

```
growTable
     if  (n  ==  m)  then  m  =  2m;

shrinkTable
     if (n < m/4) then  m = m/2;
```

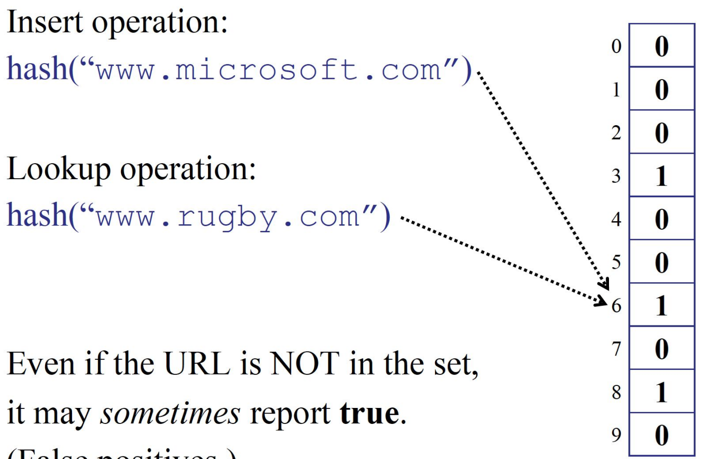
Amortized Analysis:
Operation has amortized cost `T(n)` if **for every integer k**, the cost of k operations is ≤ `k T(n)`.

Time Complexity:
Inserting k elements into a hash table takes time `O(k)`. Hence, the insert operation has amortized cost `O(1)`.

[fingerprint hash table]

A **fingerprint hash table** (of size `m` with `n` elements) implements a Set ADT that maintains a vector of 0/1 bits.

```
insert(key)
     m_table[hash(key)] = 1;

lookup(key)
     return (m_table[hash(key)] == 1);
```



Even if the URL is NOT in the set, it may *sometimes* report **true**. (False positives.)

For a **lookup**(key),

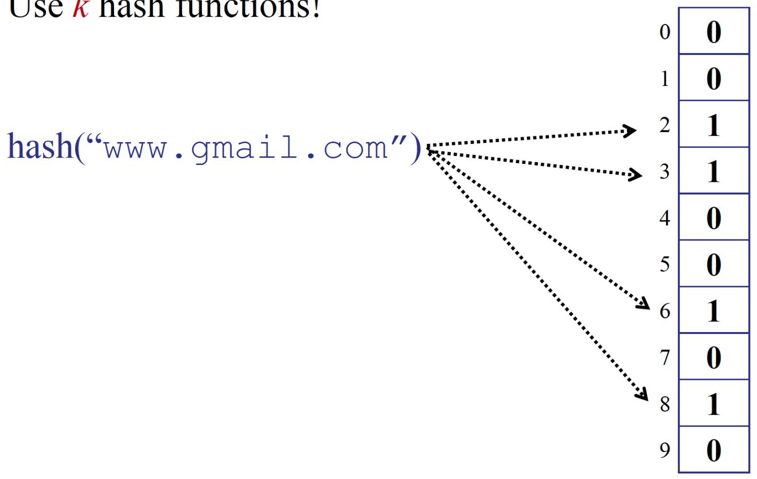
```
P(no false positive) = P(no collision) = (1 - 1/m)^n ≈ (1/e)^(n/m)
P(false positive) = 1 - (1/e)^(n/m)
```

[bloom filter]

A **bloom filter** is a fingerprint hash table that makes use of more than one hash functions. For the case of just 2 hash functions below:

```
insert(URL)
     k1 = hash1(URL); k2 = hash2(URL);
     T[k1] = 1; T[k2] = 1;

query(URL) // requires 2 collisions for false +ve
     k1 = hash1(URL); k2 = hash2(URL);
     if (T[k1] && T[k2]) return true;
     else return false
```

Use *k* hash functions!



For a bloom filter with k hash functions, assuming (bogus) fact that each table slot is independent,

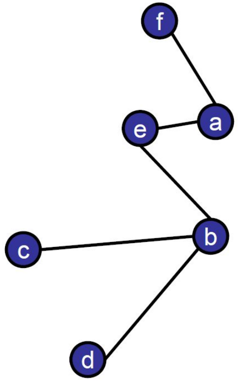
`P(a given bit is 0) = (1 - 1/m)^kn ≈ e^(-kn/m)`
`P(false positive) = P(collision at all k spots) = (1 - (1/e)^(kn/m))^k`

Optimal number for k is `(m/n) ln 2`.

[adjacency matrix]

```
Int[][] adjMatrix = new int[n][n];
```

	a	b	c	d	e	f
a	0	0	0	0	1	1
b	0	0	1	1	1	0
c	0	1	0	0	0	0
d	0	1	0	0	0	0
e	1	1	0	0	0	0
f	1	0	0	0	0	0



Format:

Undirected, unweighted: `A[v][w]` = 1 iff (v,w) ∈ E

Undirected, weighted: `A[v][w]` and `A[w][v]` = `weight(v,w)` else 0

Directed, weighted: `A[v][w]` = `weight(v,w)` else 0

Good for: Dense graph, $|E| = \theta(V^2)$

Space Complexity: $O(V^2)$

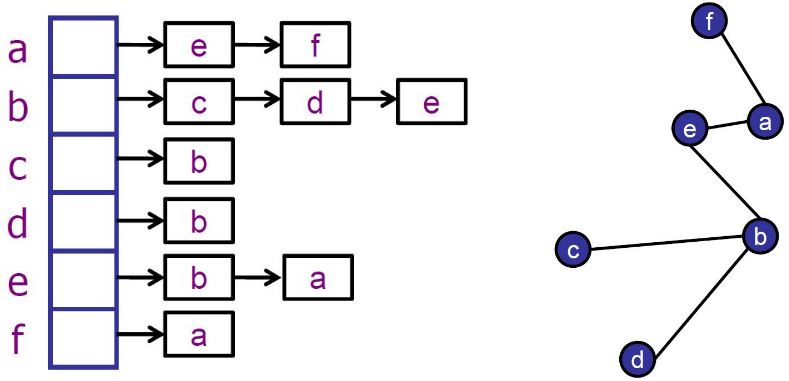
Find neighbours of v: Slow

Enumerate neighbours: Slow

Are v and w neighbours: Fast

[adjacency list]

```
<implementations>
```



Format: 1D array of **V** lists, one for each vertex. For each vertex **`i`**, `AdjList[i]` stores list of `i` neighbours.

Good for: Sparse graph, $|E| < \theta(V^2)$

Space Complexity: $O(V+E)$

Find neighbours of v: Fast

Enumerate neighbours: Fast

Are v and w neighbours: Slow

[edge list]

```
<implementations>
```

Format: ArrayList EdgeList of E edges. For each edge i, EdgeList[i] stores an integer triple {u,v,w(u,v)}

Space Complexity: O(E)

<breadth-first search>

```
BFS(G, s, f)
  Queue.add(s)
  while not Queue.empty()
    curr = Queue.dequeue()
    if curr == f
      return curr
    for each neighbor u of curr
      if u is not visited
        visit(u)
        Queue.enqueue(u)
  return null
```

Explores the graph level by level, advancing a frontier.

Works best with adjacency list. Visits every node and every vertex, but not every path.

Capable of finding shortest path in tree and unweighted graph.

Time / Space Complexity

Time: O(V+E)

<depth-first search>

```
DFS(G, s, f)
  Stack.push(s)
  while not Stack.empty()
    curr = Stack.pop()
    if curr == f
      return curr
    for each neighbor u of curr
      if u is not visited
        visit(u)
        Stack.push(u)
  return null
```

Follow path until you get stuck. Backtrack until you find a new edge. Recursively explore it, don't repeat a vertex. This implementation uses a stack to recurse.

Works best with adjacency list. Visits every node and every vertex, but not every path.

Capable of finding shortest path in tree.

Time / Space Complexity

Time: O(V+E)

<recursive depth-first search>

```
DFS(G, v, f)
  visit(v)
  if v == f
    return v
  for each neighbor u of v
    if u is not visited
      w = DFS(G, u, f)
        if w is not null
          return w
  return null
```

Follow path until you get stuck. Backtrack until you find a new edge. Recursively explore it, don't repeat a vertex. This implementation uses recursion.

Works best with adjacency list. Visits every node and every vertex, but not every path.

Capable of finding shortest path in tree.

Time / Space Complexity

Time: O(V+E)

<floyd-warshall’s>

```
D[i][j] = 0 if no edge

for k in V
  for i in V
    for j in V
      D[i][j] = foo(D[i][j],
          bar(D[i][k], D[k][j]))
```

Slow/costly preprocessing algorithm but enables all subsequent queries to be handled in O(1).

Time / Space Complexity

O(V^3) due to 3 nested loops

Use Case	foo	bar
APSP	min	sum
Minimax	min	max
Maximin	max	min

<relaxation>

```
relax(int u, int v)
  if dist[v] > dist[u]+weight(u,v) // if SP can be shortened
    dist[v] ← dist[u]+weight(u,v) // relax this edge
    predecessor[v] ← u // remember/update the predecessor
    // if necessary, update some data structure
```

Key operation in all SSSP algorithms. “Updates” the estimated shortest-path distance of a vertex from the source.

<bellman-ford’s>

```
BellmanFord(G)
  for each v ∈ V
    D[v] ← INF
    p[v] ← -1 // to represent NULL
    D[s] ← 0
  for i = 1 to |V|-1 // O(V)
    for each edge(u, v) ∈ E // O(E)
      relax(u, v) // O(1)

Seth format
BellmanFord(G)
  n = V.length;
  for (i=0; i<n; i++)
    for (Edge e : graph)
      relax(e)
```

Inefficient but guaranteed-to-work method of finding shortest path. Works if graph has negative weight edges.

Uses an Edge List.

Time / Space Complexity

Time: O(VE)

Invariants

<one-pass bellman-ford’s>

```
DAGBellmanFord(V, E)
  toposort(V) // O(V+E)
  for each v ∈ V
    D[v] ← INF
    p[v] ← -1 // to represent NULL
    D[s] ← 0
  for each vertex u in toposort(V) // O(E)
    for each neighbour v
      relax(u, v, w(u,v)) // O(1)
```

Perform topological sort on vertices first in O(V+E). Then, in that order, relax outgoing edges of vertices.

Time / Space Complexity

Time: O(V+E)

Use Case

Directed Acyclic Graph

<dijkstra’s algorithm>

```
searchPath(int start) {
  pq.insert(start, 0.0);
  distTo = new double[G.size()];
  Arrays.fill(distTo, INFTY);
  distTo[start] = 0;
  while (!pq.isEmpty()) {
    int w = pq.deleteMin(); // |V| times
    for (Edge e : G[w].nbrList) relax(e); // |E| times

relax(Edge e) {
  int v = e.from(); int w = e.to();
  double weight = e.weight();
  if (distTo[w] > distTo[v] + weight) {
    distTo[w] = distTo[v] + weight;
    parent[w] = v;
    if (pq.contains(w))
      pq.decreaseKey(w, distTo[w]); // |E| times
    else pq.insert(w, distTo[w]); // |V| times
```

Can be thought of as a specialised case of BFS where the queue (FIFO to advance frontier) is replaced by a PQ (to greedily find shortest edge)

Time / Space Complexity

Insert/DeleteMin |V| times
Relax/decreaseKey |E| times
PQ is O(log V)
Total O((V+E)logV)

Use Case

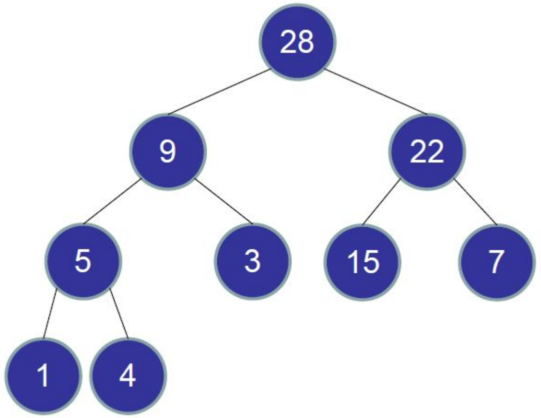
No negative weight edge

[binary heap]

```
insert(Priority p, Key k) {
  Node v = m_completeTree.insert(p,k);
  bubbleUp(v);
}

delete(Help why the slide don’t have)
```

A **binary heap** is an implementation of a Max/Min priority queue ADT that is a complete binary tree.

Every level is full, except possibly the last. All nodes are as far left as possible.



<bubble-up>

```
bubbleUp(Node v) {
      while (v != null) {
            if (priority(v) > priority(parent(v)))
                  swap(v, parent(v));
            else return;
            v = parent(v);
      }
}
```

<Description>

Time / Space Complexity

Invariants

<bubble-down>

```
bubbleDown(Node v)
     while (!leaf(v)) {
          leftP = priority(left(v));
          rightP = priority(right(v));
          maxP = max(leftP, rightP, priority(v));
          if (leftP == max) {
               swap(v, left(v));
               v = left(v); }
          else if (rightP == max) {
               swap(v, right(v));
               v = right(v); }
     else return;
}
```

<Description>

Time / Space Complexity

Invariants

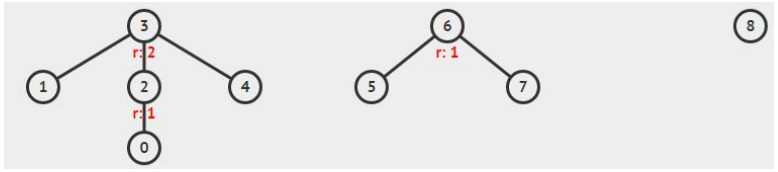
[union-find disjoint set]

```
union(int p, int q) {
  while (parent[p] !=p) p = parent[p];
  while (parent[q] !=q) q = parent[q];
  if (size[p] > size[q]):
    parent[q] = p; // Link q to p
    size[p] = size[p] + size[q];
  else:
    parent[p] = q; // Link p to q
    size[q] = size[p] + size[q];

findRoot(int p) {
  root = p
  while (parent[root] != root) root = parent[root];
  while (parent[p] != p) {
    temp = parent[p];
    parent[p] = root;
    p = temp;
  return root;
```

A **UFDS** is used to determine which set an item belongs to, test if two items belong to the same set, and union two disjoint sets into one.

With both **path compression** and **union-by-rank** heuristic applied, UFDS operations run in **near O(1)** time.

p	2	3	3	3	3	6	6	6	8
idx	0	1	2	3	4	5	6	7	8



<kruskal’s algorithm>

```
sort the set of E edges by increasing weight
// O(E log E)
T={}
while there are unprocessed edges left // O(E)
  pick unprocessed edge e with min cost // O(1)
  if adding e to T does not form a cycle // O(1)
    add e to T // O(1)
T is an MST
```

Consider the a set of 1-vertex trees. Remove smallest edge from set of edges and add it to graph if it does not form a cycle.

Uses a UFDS to check for cycle formation by observing if two edges are already in the same tree.

<prim’s algorithm>

```
T = {s}
enqueue edges connected to s into PQ
while unprocessed edges in PQ
  poll topmost edge e
  if vertex v linked with edge e is not taken yet
  T = T ∪ v
  enqueue edges adjacent to v not in T into PQ
T is an MST
```

Initially, tree consists of source vertex. Repeatedly find minimum weight edge on the “cut” between tree and rest of graph by extracting from PQ.

Uses a PQ

Time / Space Complexity

We only process each edge once (enqueue and dequeue it), O(E)
Each time, we enqueue/dequeue from a PQ in O(log E)
As E= $O(V^2)$, we have $O(\log E) = O(\log V^2) = O(2 \log V) = O(\log V)$
Total time O(E)*O(logV) = O(ElogV)

DS	Operations	Big-O	Main point

Array List	1.Get
	2.Add
	3.AddBack
	4.Remove
	5.RemoveBack
		1.O(1)
		2.O(N)
		3.O(1)
		4.O(N)
		5.O(1)
			1.Good for accessing, not good for adding/removing
			2.Can be made circular to reduce the need to double the array

Linked List	1.Get
	2.Add
	3.AddFront
	4.Remove
	5.RemoveFront
		1.O(N)
		2.O(N)
		3.O(1)
		4.O(N)
		5.O(1N)
			1.Good for adding and removing
			2.Tailed Linked List has RemoveBack in O(1)
			3.DoublyLinkedList has slightly faster search, average O(n/2)
			4.Implement ListNode class

HashMap	1.Find
	2.Insert
	3.Delete
		1.O(1)
		2.O(1)
		3.O(1)
			1.Linear Probing (hash(key) + k) % m
			2.Modified (hash(key) + k*d) % m;d,m coprime
			3.Quadratic (hash(key) + k^2) % m
			4.Double: (hash(key) + 3*hash2(key)) % m

DS	Operations	Big-O	Main point

BST/AVL	1.Insert
	2.Delete
	3.Search
	4.findMin/Max
	5.Pred/Succ
	6.Inorder
	7.Select
	8.Rank
		1.O(h)
		2.O(h)
		3.O(h)
		4.O(h)
		5.O(h)
		6.O(N)
		7.O(h)
		8.O(h)
			1.y.key< x.key if y is in left subtree of x
			2.y.key ≥ x.key if y is in right subtree of x
			3.For AVL tree, h=logN, h < $2\log_2 n$
			4.AVL Tree A node v is height-balanced if: |v.left.height – v.right.height| <= 1
			5.Made of nodes/vertices
			6.Traversal is O(N) – can use to sort/transfer to another data structure

UFDS	1.findSet(i)
	2.isSameSet(i,j)
	3.unionSet(i,j)
	4.Construct(N)
		All O(α(N)), basically O(1)	1.Union by rank: tree with higher height becomes root when union
			2.Path compression: findset compresses path by making root into parent
			3.Used in MST Kruskal's to check for cycle

Binary Max Heap	1.ShiftUp/Swim
	2.ShiftDown/Sink
	3.Insert
	4.ExtractMax/Min
	5.Fast Create
		1.O(log N)
		2.O(log N)
		3.O(log N)
		4.O(log N)
		5.O(log N)
			1.Implementation for PriorityQueue ADT
			2.Binary tree height log N
			3.Use compact array
			4.ExtractRoot replaces with successor, then sink if need
			5.Fast create by pretending array is already heap, then fixing

Algorithm	Complexity	When to Use	Characteristics
Bellman-Ford's	O(VE)	General	Relax all edges in V-1 passes. Optimise by stopping if any pass finishes without relaxation. Without optimisation, bound is O(VE) cannot do better (input insensitive).
BFS	O(V+E)	Unweighted	Replace visited array with distance array, initialise all INF and then replace with counter D[v]=D[u]+1
BFS/DFS	O(V+E)=O(V)	Tree	Replace visited array with distance array, initialise all INF and then replace with counter D[v]=D[u]+1
Dijkstra's	O((V+E)logV)	No neg weight	Greedily extract vertex with lowest D[u] estimate and relax edges out of it – when given undirected/unweighted graph, is a BFS
Modified Dijkstra's	O((V+E)logV)	No neg cycle	Do not delete inferior copy of D[u], at the cost of possibly longer runtime with –ve edges present (but correct answer)
1-Pass Bellman-Ford	O(V+E)	DAG	Literally 1-pass Bellman Ford relaxing edges in **topological** order of source vertex
Prim's	O(ElogV)	MST	Start from source vertex and greedily add vertex with smallest weight across the cut. Need PQ to choose edges.
Kruskal's	O(ElogV) from sorting	MST	Sort edges. Add edges as long as they don't form a cycle. Need UFDS to check for cycle forming.

BST Property:
All in left subtree < node < all in right subtree
Successor is minimum of right subtree if right subtree exists

Binary Max Heap Property:
A[parent(i)] > A[i], binary tree is complete (all levels filled except last)
A larger int can be at a lower level than a smaller int

Tree Property:
Tree is smallest connected graph and has V-1 edges
Only one path from any vertex to another
MST and SSSPT are same

MST: If all edge weights are distinct, MST is unique
“Can we run algo faster” yes if we change the data structure first. No constraint saying cannot change edge list to adj list for example <- watch out chongkf likes to sike ppl