THEOREM

Master Theorem

Given a recurrence of the form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

for constants $a\,(\geq 1)$ and $b\,(> 1)$ with f asymptotically positive, the following statements are true:

- **Case 1.** If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.
- **Case 2.** If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \log n\right)$.
- **Case 3.** If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some $\epsilon > 0$ $\left(\text{and } af\left(\frac{n}{b}\right) \leq cf(n) \text{ for some } c < 1 \text{ for all } n \text{ sufficiently large}\right)$, then $T(n) = \Theta(f(n))$.

1. $T(n) = 2T(n/2) + O(n) \log n = O(n \log^2 n)$.
2. $T(n) = T(n/2) + O(1) = O(\log n)$ binary search.
3. $T(n) = T(n/2) + O(n) = O(n)$ quicksort
4. $T(n) = 2T(n/2) + O(n) = O(n \log n)$ sorts.

$T(n) = T(\sqrt{n}) + O(\log n) =$

$n^{\frac{1}{2^k}} = 1$

$k = \log n$.

$k = O(\log \log n)$.

$\log n$

$\frac{1}{2}\log n = \log \sqrt{n}$.

$\frac{1}{4}\log n$

$\vdots$

1

n

$n^{1/2}$

$n^{1/2^2}$

$\vdots$

$n^{1/2^k} \rightarrow 2^{-k}\log n = 1 \mid 1+2+4+\cdots+\log n = O(\log n)$.

$2\log n = 2^k = \log n \log n$

$T(n) = T(n-2) + O(n^2) = O(n^3)$

Algorithms:

Searching:

1. Binary Search $O(\log n)$

Functionality:
- if element is in the array, return index of element
- if element is not in array, return -1

Precondition: (fact that is true when the function begins) ⇒ important for it to work correctly.
- Array is of size n
- Array is sorted

```
int search (A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        mid = begin + (end - begin)/2;
        if key <= A[mid]
            then end = mid
        else
            begin = mid+1

    return   A[begin] == key
           ? begin
           : -1
```

Postcondition: (fact that is true when the function ends) ⇒ show that the computation was done correctly.
- if element is in the array:

  A[begin] = key

Invariant:
- relationship between variables that is always true.

Loop Invariant
- relationship between variables that is true at the beginning (or end) of each iteration of a loop.
- A[begin] ≤ key ≤ A[end]. (correctness)
  (the key is in the range of the array).
- (end - begin) <= $n/2^k$ in iteration k (performance)

Not just for searching arrays
- eg. A complicated function.
  the function that is always increasing.
  - min value that satisfies the complicated function.

	$O(\log [(\log n)^2]) = O(\log \log n) <$		
Logarithmic Time	$O(\log_3 n) = O(\log n) <$		
Sublinear Time	$O(n^{0.00000001}) <$	$O(100\, n^{2/3}) <$	
Linear Time	$O(20n) <$		
Linearithmic Time	$O(n \log n) = O(\log n!) <$		
Quadratic Time	$O(4n2) <$		
Polynomial Time	$O(n^{2.5}) <$		
Exponential Time	$O(2^n) = O(2^{n+1}) <$	$O(3^n) <$	$O(2^{2n}) = O(4^n) <$
Factorial Time	$O((n-1)!) <$	$O(n!) <$	
!!!	$O(n^n)$		

2. Peak (Local Max) Finding $O(\log n)$

Input: Some array A[0...n-1]

Output: A local maximum in A A[i-1] ≤ A[i] and A[i+1] ≤ A[i] (change to < for strict peaks)

Assume: A[-1] = A[n] = -MAX_INT.

```
FindPeak(A,n)
    if A[n/2+1] > A[n/2]
        then FindPeak(A[n/2+1....n], n/2)   right half
    else if A[n/2+1] > A[n/2]
        then FindPeak(A[1.... n/2-1], n/2)   left half.
    else
        A[n/2] is a peak
        return n/2.
```

Invariant (key-property)
- If we recurse in the right half, then there exist a peak in the right half ⇒ The array must have a peak.
- Prove by Induction (Invariant)

Correctness
- There exists a peak in the range [begin, end]
- Every peak in [begin, end] is a peak in [1, n].

Sorting:

Total space = O(n)

Input: array A[1...n] of words/numbers (unsorted)

Output: array B[1...n] that is a permutation of A such that:

$B[1] \le B[2] \le \dots \le B[n]$ (sorted)

1. BogoSort (A[1...n]) O(n·n!) in-place

Repeat: (not stable!)

1. Choose a random permutation of the array A
2. If A is sorted, return A.

2. BubbleSort (A, n) O(n) → O(n²) in-place

Repeat ~~n times~~ until no swaps:

for j = 1 to n-1

if A[j] > A[j+1] (only swap elem if they are different ∴ stable!)

then swap (A[j], A[j+1])

Best case:
(already sorted) O(n)

Avg case:
(random inputs) O(n²)

Worst case:
(worse input, n iterations) O(n²).

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

Correctness:
After n iterations → sorted.

3. SelectionSort (A, n) O(n²) only. in-place

for j = 1 to n-1:

find min element A[j] in A[j...n] (Time: (n-j))

swap (A[j], A[k]). (not stable! swap elem while ignoring in-between.)

least swaps vs insertion & bubble.

Loop invariant:
At the end of iteration j:
the smallest j items are correctly sorted
in the first j position of the array.

4. Insertion Sort (A, n) O(n) → O(n²) in-place

for j = 2 to n

key = A[j]

[Insert key into sorted array A[1...j-1]]

i = j-1

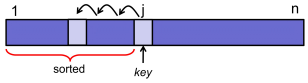
While i > 0 and A[i] > key (Stable, do not swap identical elements.)

A[i+1] = A[i]

i = i-1

A[i+1] = key

(Repeat at most j times)



Loop invariant:
At the end of iteration j:
the first j items in the array
are in sorted order.

Best case:
(already sorted) O(n)

Avg case:
(random permutation) $\sum_{j=2}^{n} O(j/2) = O(n^2)$
(on avg, a key in pos j needs to move back j/2 slots (in expectation).)

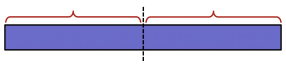
Worst case:
(inverse sorted) O(n²)

Insertion vs Bubble ⇒ almost sorted array.

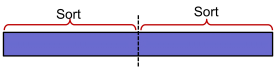
Note: Stability: preserves order of equal elements.

5. Mergesort (Divide-and-Conquer) O(c n log n) Space: O(n log n)

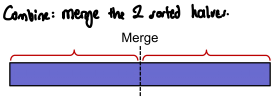
1. Divide: split array into 2 halves



2. Recurse: sort the 2 halves



3. Combine: merge the 2 sorted halves.



iterative ver.: from bottom up.

O(n) space ver.: use only 1 temp. array with merge.

S(n) = 2S(n/2) + O(1) = O(n)

```
MergeSort(A, begin, end, tempArray)
    if (begin=end) then return;
    else:
        mid = begin + (end-begin)/2
        MergeSort(A, begin, mid, tempArray);
        MergeSort(A, mid+1, end, tempArray);
    Merge(A[begin..mid], A[mid+1, end], tempArray);
    Copy(tempArray, A, begin, end);
```

MergeSort (A, n)

if n=1 then return; Base case. O(1)

else:

x = MergeSort (A[1 ... n/2], n/2); Recursive! T(n/2)

Y = MergeSort (A[n/2+1 ... n], n/2); Conquer T(n/2)

return Merge (X, Y, n/2); Combine solutions O(n)

In each iter, move 1 elem to final list
After n iters, all items are in final list.
Each iter takes O(1) time to compare 2 elems and copy).

Not in-place!

(Mostly sorted) ⇒ slower than insertion, O(n log n)

(small number of items) ⇒ slow, use insertion sort for n < 1024 ⇒ due to caching / branch prediction.

⇒ base case = use a slower sort

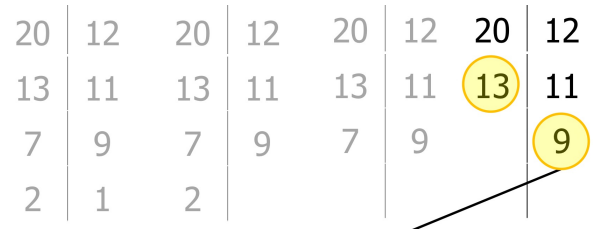
Space usage:

- need extra space for merge
- merge copies data to new array.
  - temporary array of size n.

Stability: Stable.

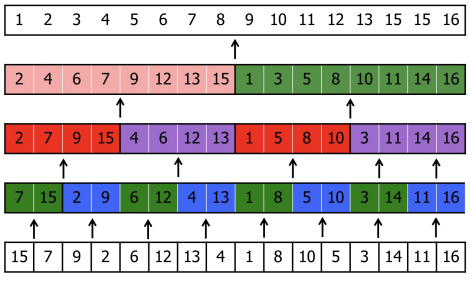
- Mergesort is stable if merge is stable.
- merge is stable if carefully implemented.

Merge:



Need temp array of size n.

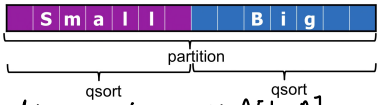
Iterative version (doesn't require extra space)



Name	Best Case	Average Case	Worst Case	Extra Memory	Stable?
Bubble Sort	O(n)	O(n²)	O(n²)	O(1)	Yes
Selection Sort	O(n²)	O(n²)	O(n²)	O(1)	No
Insertion Sort	O(n)	O(n²)	O(n²)	O(1)	Yes
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n log n)	Yes
Quick Sort	O(n log n)	O(n log n)	O(n²)	~~O(log n)~~ in-place ∴ O(1)	No

6. QuickSort (A[1...n], n) O(nlogn)

```
if n==1 then return;
else:
    choose pivot index pIdx          ← depends if partition is in-place.
    p = partition(A[1...n], p, pIdx)
    x = QuickSort(A[1...p-1], p-1)
    y = QuickSort(A[p+1...n], n-p)
```



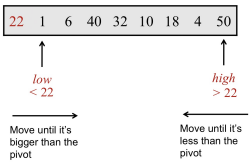
Given: n element array A[1...n]

1. Divide: Partition the array into 2 sub-arrays around a pivot x ← efficiency key.
   such that elements in lower subarray ≤ x ≤ elements in upper sub-array.

2. Conquer: Recursively sort the 2 sub-arrays
3. Combine: Trivial, do nothing.

Partition: O(n)

1. Choose a pivot
2. Find all elements smaller than the pivot
3. Find all elements larger than the pivot



Cnot in-place)

```
partition (A[2...n], n, pivot)
//assume no duplicates
    B = new n element array
    low = 1;
    high = n;
    for (i = 2; i ≤ n; i++)
        if (A[i] < pivot)
            then B[low] = A[i];
                 low++;
        else if (A[i] > pivot)
            then B[high] = A[i]
                 high--;
    B[low] = pivot
    return <B, low>
```

Array B is partitioned around the pivot

proof:

Invariants:

1. For every i < low: B[i] < pivot
2. For every j > high: B[j] > pivot.

In the end, every element from A is copied to B.

Thus: B[i] = pivot

By invariants, B is partitioned around the pivot.

(in-place using 2 pointers)

```
partition (A[2...n], n, pIdx)  → assume no duplicates, pIdx is index of pivot
    pivot = A[pIdx];
    swap(A[1], A[pIdx]);  → store pivot in A[1]
    low = 2;  → start after pivot in A[1]
    high = n+1;  → define A[n+1] = ∞.
    while (low < high)
        while (A[low] < pivot) and (low < high)
            do low++;
        while (A[high] > pivot) and (low < high)
            do high--;
        if (low < high)
            then swap(A[low], A[high]);
    swap(A[1], A[low-1]);
    return low-1;
```

Invariant:

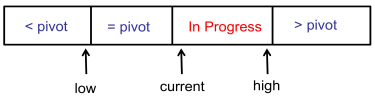
- A[high] > pivot at the end of each iter
- At the end of every loop iteration:
  for all i > high, A[i] > pivot.
  for all 1 < j < low, A[j] < pivot.

∴ Array A is partitioned around the pivot.

Duplicates?

3 way partitioning

- 2 pass partitioning
  1. Regular partition
  2. Move duplicates.
- 1 pass partitioning
  - more complicated
  - maintain 4 regions of the array.



Choice of pivot? Worst case doesn't matter ⇒ Sorting the array takes n execution of partition.
⇒ equally bad
- each call to partition sorts one element
- each call to partition of size k takes ≥ k.

- First element A[1]? eg. reverse sorted ⇒ O(n²)
- last element A[n]? eg. sorted ⇒ O(n²).
- middle element A[n/2]?
- median of (A[1], A[n/2], A[n])?

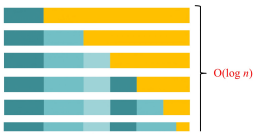
If array is split evenly

$T(n) = T(n/2) + T(n/2) + cn$
$= 2T(n/2) + cn$
$= O(n \log n)$.

If array is split (1/10) : (9/10)

$T(n) = T(n-k) + T(k) + n$
$\approx O(n \log n)$.

Assume larger part shrinks by at least 9/10 every iteration:



∴ Choose pivot at random.
⇒ running time is a random variable.
⇒ for every input, there's a good probability of success.

Set of outcomes for X = $(e_1, e_2, e_3, \ldots, e_k)$:
- $Pr(e_1) = p_1$
- $Pr(e_2) = p_2$
- ...
- $Pr(e_k) = p_k$

Expected outcome: $E[X] = e_1p_1 + e_2p_2 + \ldots + e_kp_k$

Linearity of expectation: E[A+B] = E[A] + E[B].

eg. Flipping an (unfair) coin:
- Pr(heads) = p
- Pr(tails) = (1 − p)

How many flips to get at least one head?

$E[X] = (p)(1) + (1-p)(1 + E[X])$

How many more flips to get a head?

Idea: If I flip "tails," the expected number of additional flips to get a "heads" is still E[X]!

paranoid quicksort: Then we know:

$E[T(n)] = E[T(k)] + E[T(n-k)] + E[\#\text{ pivot choices}](n)$
$<= E[T(k)] + E[T(n-k)] + 2n$
$= O(n \log n)$

How to partition efficiently? (key!)

- Duplicates
  If you ignore duplicates, partitioning can be very slow!
- In-place partitioning
  One pass, use invariants to ensure correctness.
- Stability
  In-place partitioning isn't stable.
- Choosing a pivot (randomization)
  Deterministic pivots are generally bad.
  Randomization is an easy way to find a good pivot!

How to analyze?

- Divide-and-conquer recurrence
  It is sufficient to do a 90:10 split to get O(n log n) performance! (no need exact half)
- What is the probability that a random pivot yields a 90:10 split? Quite good! (high p)
- Simplification: Paranoid QuickSort
- How many repetitions to find a good pivot, in expectation? O(1)
- Solve using recurrence: Linearity of expectation FTW.

Optimisation:

Base case?

1. Recurse all the way to single-element arrays.
2. Switch to InsertionSort for small arrays.
3. Halt recursion early, leaving small arrays unsorted. Then perform InsertionSort on entire array.

Relies on fact that InsertionSort is very fast on *almost sorted* arrays!

Double pivot.

Order Statistics O(n)

worst case $O(n^2)$ for largest/smallest element as pivot.

- Find the k^{th} smallest element in an unsorted array.
  - 1. Sort the array
    2. return element number k
    ⇒ O(nlogn)
  - Only do min number of sorting
    1. Partition the array.
    2. Continue searching in the correct half.

```
QuickSelect (A[1...n], n, k)
    if (n==1) then return A[1];
    else Choose random pivot index pIdx.
        p = partition (A[1...n], n, pIdx)
        if (k==p) then return A[p];
        else if (k<p) then
            return Select(A[1...p-1], k)   → Only recurse once! ⇒ not sorting
        else if (k>p) then                   recursing right and left
            return Select(A[p+1], k-p)       are not exactly the same.
```

Recurrence: $T(n) = T(\frac{n}{2}) + O(n) \approx O(n)$

Trees

Binary Tree : either (a) empty
(b) a node pointing to 2 binary trees

```
public class TreeNode {
    private TreeNode leftTree;
    private TreeNode rightTree;
    private keyType key;
    private valueType value;
    ...
}
```

property:
all in left sub-tree < key < all in right sub-tree.

height
- Number of edges on longest path from root to leaf.
- h(v) = 0
  h(v) = max(h(v.left), h(v.right)) + 1.
  h(null) = -1.
-
```
public int height(){
    int leftHeight = -1;
    int rightHeight = -1;
    if(leftTree != null)
        leftHeight = leftTree.height();
    if(rightTree != null)
        rightTree = rightTree.height();
    return max(leftHeight, rightHeight) + 1;
```
  - max of subtrees.

search (max/min) O(h)

```
public TreeNode searchMax(){
    if (rightTree != null){
        return rightTree.searchMax();
    }
    else return this; ⇒ key is here!
}
```

Worst Case: O(n) ⇒ eg. linear tree
Avg Case: O(h) ⇒ O(logn).

search (any key)

```
public TreeNode search(int queryKey){
    if (queryKey < this.key){
        if (leftTree != null)
            return leftTree.search(queryKey);
        else return null;
    }
    else if (queryKey > this.key){
        if (rightTree != null)
            return rightTree.search(queryKey);
        else return null;
    }
    else return this;
}
```

insert O(h)

```
public void insert(int inskey, int insValue){
    if(inskey < this.key){
        if(leftTree != null)
            leftTree.insert(inskey);
        else leftTree = new TreeNode(inskey, insValue);
    }
    else if(inskey > key){
        if(rightTree != null)
            rightTree.insert(inskey);
        else rightTree = new TreeNode(inskey, insValue);
    }
    else return; ⇒ key als in Tree.
}
```

performance of tree algorithms depends on shape.
- ↳ order of insertion.
- ↳ each order does not yield a unique shape.
- ↳ insert keys in a random order ⇒ balanced.

↗ can reconstruct a BST using any-order-traversal.

in-order-traversal (v) O(n) ⇒ visit each node at most once
- left-subtree
  self
  right-subtree

```
- public void in-order-traversal(){
pre-order →
    // Traverse left sub-tree
    if (leftTree != null)
        leftTree.in-order-traversal();
in-order → visit(this);
    // Traverse right sub-tree
    if(rightTree != null)
        rightTree.in-order-traversal();
post-order →
}
```

successor O(h)
- If you search for a key not in tree
  ↳ either find predecessor or successor.
- successor(key)
  1. Search for key in the tree
  2. if (result > key), then return result.
  3. if (result ≤ key), then search for successor of result.

```
public TreeNode successor(){
    if (rightTree != null)
        return rightTree.searchMin();

    TreeNode parent = parentTree;
    TreeNode child = this;
    while ((parent != null) && (child == parent.rightTree))
        child = parent;
        parent = child.parentTree;
    }
    return parent;
}
```

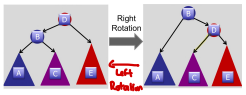
delete O(height)
- 3 cases
  - No Children
    - remove v
  - 1 child
    - remove v
    - connect child(v) to parent(v)
  - 2 children.
    - x = successor(v)
    - delete(x)
    - remove v
    - connect x to left(v), right(v), parent(v).

    ‖ swap x w. v.
- Successor of deleted node has at most 1 child.

Balanced Trees

- Operations take O(h) time
  $\log(n)-1 \le h \le n$
- * A BST is balanced if $h = O(\log n)$
- On a balanced BST: all operations run in $O(\log n)$ time
- How to get a balanced tree?
  - define a good property of a tree
  - Show that if the good property holds, then the tree is balanced.
  - After every insert/delete, make sure the good property still holds. If not, fix it. → invariant.

Tree Rotations ⇒ Can create every possible tree shape!



A < B < C < D < E

Rotations maintain ordering of keys.

⇒ Maintains BST property.

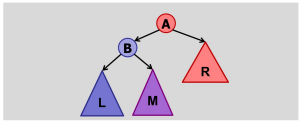
Right Rotation: The root of the subtree moves right. (req. left child)

Left Rotation: The root of the subtree moves left. (req. right child)

```
right-rotate(v)
  // assume v has left != NIL
  w = v.left
  w.parent = v.parent
  v.parent = w
  v.left = w.right
  w.right = v
```

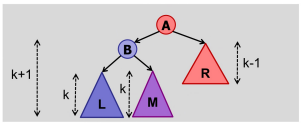
A is Left-heavy if left sub-tree has longer height than right sub-tree.
A is Right-heavy if right sub-tree has longer height than left sub-tree.

Note: Rotations reduces root height by 1, and everything higher in tree remains unchanged.



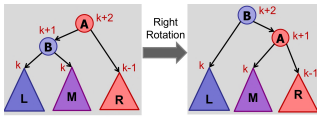
Assume **A** is the lowest node in the tree violating balance property.

Assume A is **LEFT-heavy**.



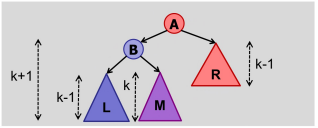
Assume **A** is the lowest node in the tree violating balance property.

Case 1: **B** is equi-height : h(**L**) = h(**M**)
h(**R**) = h(**M**) − 1



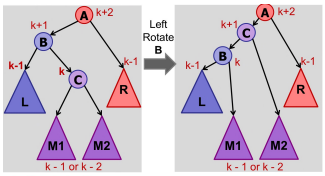
right-rotate:

Case 1: **B** is equi-height : h(**L**) = h(**M**)
h(**R**) = h(**M**) − 1



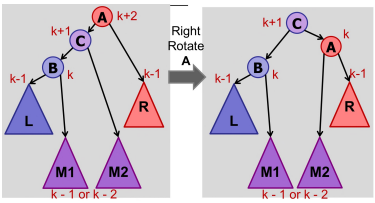
Assume **A** is the lowest node in the tree violating balance property.

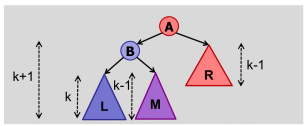
Case 3: **B** is right-heavy : h(**L**) = h(**M**) - 1
h(**R**) = h(**L**)



Left-rotate B

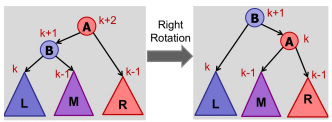
After left-rotate B: **A** and **C** still out of balance.





Assume **A** is the lowest node in the tree violating balance property.

Case 2: **B** is left-heavy : h(**L**) = h(**M**) + 1
h(**R**) = h(**M**)



right-rotate:

Case 2: **B** is left-heavy: h(**L**) = h(**M**) +1
h(**R**) = h(**M**)

Summary:

If v is out of balance and 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 out of balance and right heavy:
Symmetric three cases....

AVL Trees

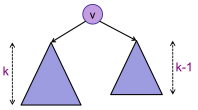
Step 0: Augment

- In every node v, store height: (need not be height, can be difference in height from parent etc.)
  v.height = h(v)
- On insert & delete update height:

```
insert(x)
  if (x < key)
    left.insert(x)
  else right.insert(x)
  height = max(left.height, right.height) + 1
```

Step 1: Define Balance Condition / Invariant*

- A node is height-balanced if:
  $|v.left.height - v.right.height| \le 1$



- A binary search tree is height-balanced if every node in the tree is height-balanced.
- A height-balanced tree with n nodes has at most height $h < 2\log(n)$.
  $\Leftrightarrow \frac{h}{2} < \log(n)$
  $\Leftrightarrow 2^{h/2} < 2^{\log n}$
  $\Leftrightarrow 2^{h/2} < n$
- A height-balanced tree with height h has at least $n \ge 2^{h/2}$ nodes.

Proof:

Let n_h be the min num of nodes in a height-balanced tree of height h.

Show: $n_h > 2^{h/2}$

$n_h > n_{h-1}$

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

note: no. of nodes rooted at v is at most 2^h

$\therefore n_h \ge 2n_{h-2} \ge 2^1 n_{h-2}$
$\ge 4n_{h-4} \ge 2^2 n_{h-4}$
$\ge 8n_{h-6} \ge 2^3 n_{h-6}$
$\ge \dots \ge \dots$
$\ge 2^h n_0$
$= 1$ (base case $n_0 = 1$)

$k = h/2$

$\therefore n_h \ge 2^{h/2} n_0$
$\ge 2^{h/2}$

$\therefore n_h > 2^{h/2}$

$\Leftrightarrow h < 2\log(n)$

$h = 1.44 \log(n)$ using Fibonacci no.

Step 2: Maintain Balance

- During insertion (≤ max 2 rotations) ⇒ can be 0 rotations
  ↳ Tree Rotations. (height is out-of-balance by 1)
  ↳ Insert key in BST
  - Walk up tree:
    - At every step, check for balance
    - If out-of-balance, use rotation to rebalance and return
  - only need to fix the lowest out-of-balanced node
- During deletion (≤ O(log n) rotations)
  ↳ 3 cases
  1. No children
  2. 1 child
  3. 2 children

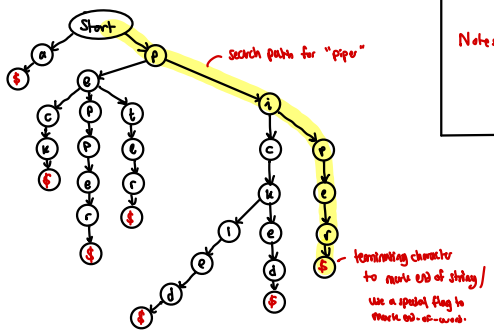
  ↳ Can also mark a node as 'delete' and leave it on the tree (Lazy deletion)
  ↳ performance degrades over time.

```
↳ delete(v)
  1. If v has 2 children, swap it with its successor
  2. Delete node v from binary tree (can guarantee child ≤ 1)
  3. For every ancestor of the lowest node:
     - check if it is height-balanced
     - if not, perform a rotation
     - Continue to the root.
```

↳ note: 2 rotations not enough!
- delete reduces height
- rotation reduces height
- rebalancing to one node may change its height (caused by insertion)

Trie

- Cost of comparing 2 strings:
  - Cost [A ? B] = min(A.length, B.length)
  - Compares strings letter by letter.
- Cost of tree operation:
  - Assume string has length L
  - O(hL) - worst case
- One letter in each node. O(L) space: O(size of text)* overhead)



- Trie tends to be faster
  ↳ Does not depend on size of total text.
  ↳ Does not depend on number of strings.
  ↳ even faster if string not in Trie.
- Trie tends to use more space
  ↳ BST and Trie use O(text size) space
  ↳ Trie has more nodes and more overhead.
- Trie node:
  - has many children
  - For strings: fixed degree
  - Ascii characters: `TrieNode children[] = new TrieNode[256]` (wasted space!)

Applications:
- Searching.
- Sorting / enumerating strings
- partial string operations:
  - prefix queries
  - long prefix
  - wildcard

Trees are a good way to store, summarize and search dynamic data

- Operations that create a data structure
  - build (preprocess)
- Operations that modify the structure.
  - insert
  - delete
- Query Operations
  - search, select etc.

Notes:
- every height-balanced tree is balanced.
- not every balanced tree is height-balanced.

(a, b) - trees

└ $2 \le a \le \frac{b+1}{2}$.

└ a: min no. of children in an internal node of a tree.

└ b: max no. of children in an internal node of a tree.

Binary trees	(a, b)-trees
Each node *has at most* 2 children	Each node *can have more than* 2 children
Each node *stores exactly one* key	Each node *can store multiple* keys

Note: (a,b) - trees in general are good for searching big data.

↳ more cache memory utilized.

Rules:

1. (a,b) - child policy

root: $1 \le$ keys $\le b-1$
$2 \le$ children $\le b$
— no. of children always one more than keys.

internal: $a-1 \le$ keys $\le b-1$
$a \le$ children $\le b$

leaf: $a-1 \le$ keys $\le b-1$
no children!

2. Key ranges

A non-leaf node must have one more child than its number of keys.

└ to ensure all value ranges due to its keys are covered in its subtrees.

3. Leaf depth

All leaf nodes must all be at the same depth (from root).

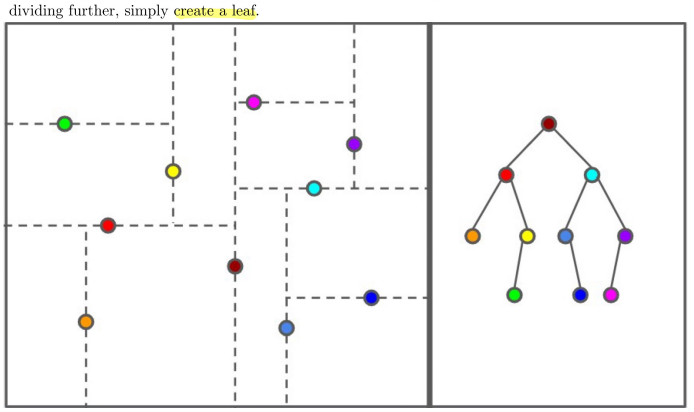
Kd-trees

A kd-tree is another simple way to store geometric data in a tree. Let's think about 2-dimensional data points, i.e., points (x, y) in the plane. The basic idea behind a kd-tree is that each node represents a rectangle of the plane. A node has two children which divide the rectangle into two pieces, either vertically or horizontally.

For example, some node v in the tree may split the space vertically around the line $x = 10$: all the points with x-coordinates ≤ 10 go to the left child, and all the points with x-coordinates > 10 go to the right child.

Typically, a kd-tree will alternate splitting the space horizontally and vertically. For example, nodes at even levels split the space vertically and nodes at odd levels split the space horizontally. This helps to ensure that the data is well divided, no matter which dimension is more important.

All the points are stored at the leaves. When you have a region with only one node, instead of dividing further, simply create a leaf.



B - trees

└ Simply (B, 2B)-trees.

└ Operations: • Split

Conditions:

1. node contains $\ge 2a$ keys because of its splitting,
   LHS has $\ge a-1$ keys
   RHS has $\ge a$ keys
2. node's parent contains $\le b-2$ keys.

1. pick median key v_m as split key.
2. split node x into LHS and RHS using v_m
3. create a new node y.
4. move LHS split from x to y.

dealing with root splitting:

5. create a new empty node e
6. insert v_m into e
7. promote e to new root node
8. Assign y and x to be left and right child of e
9. Assign previous subtree t_m associated with v_m to be final child of y.

• Insertion O(blogn)

1. Start at node x = root
2. Repeat until x = leaf: (preemptive split nodes at max capacity ⇒ proactive approach.)
   2.1 If x contains $b-1$ keys,
      2.1.1 split x into y (LHS) and z (RHS) using split key v_m.
      2.1.2 Examine v_m returned by the split operation
         • If $x \le v_m$, set $x = y$
         • else, set $x = z$.
   2.2 Search for a in the keylist of x
      • If a is larger than all the keys, set a to be the final child
      • else, look for first key $v_i \ge a$ and set x to be t_i (subtree w/ key range $\le v_i$)
3. Add x to w's keylist.

Scenario	Operation	Algorithm
• y and z are siblings • Have $< b-1$ keys together	`merge(y, z)`	1. In parent, delete key v (the key separating the siblings) 2. Add v to the keylist of y 3. In y, merge in z's keylist and treelist 4. Remove z
• y and z are siblings • Have $\ge b-1$ keys together	`share(y, z)`	1. `merge(y, z)` 2. Split newly combined node using the regular `split` operation

• Deletion

1. Start at node $w = root$
2. Repeat until w contains x or w is a leaf:
   2.1. If w contains $a-1$ keys and z is a sibling of w
      • If w and z together contain $< b-1$ keys,
         i. `merge(w,z)`
         ii. Set w to be the newly merged node
      • Else if w and z together contain $\ge b-1$ keys,
         i. `share(w,z)`
         ii. Set w to be the newly split node whose key range contains x
   2.2. Search for x in the keylist of w
      • If x is larger than all the keys, set w to be the final child
      • Else, look for the first key $v_i \ge x$ and set w to be t_i (subtree with key range $\le v_i$)
3. Check for x in w's keylist,
   • If it exists, remove it and return success
   • Else, return failure

Augmented Trees.

Basic Methodology:

1. Choose underlying data structure
   (tree, hash table, linked list, stack etc.)
2. Determine additional info needed
3. Modify data structure to maintain additional info when the structure changes
   (subject to insert/delete etc.)
4. Develop new operations.

Order Statistics.

Input: A set of integers.
Output: select(k)
↳ the k^{th} item in the set.

	Sort.	QuickSelect.
preprocess:	O(nlogn)	O(1)
select:	O(1)	O(n)

↳ trade-off: how many items to select?

Implement a data structure that supports:
- insert(int key)
- delete(int key)

and also:
- select(int k)

Solution 1:
Basic structure: sorted array A.
insert(int item): add item to sorted array A.
select(int k): return A[k]

7	9	13	18	22	25	43	52	65	67	87	92

Solution 2:
Basic structure: unsorted array A.
insert(int item): add item to end of array A.
select(int k): run QuickSelect(k)

7	9	13	18	22	25	43	52	65	67	87	92

Solution 1 > Solution 2
iff there are more selects than insert.

	Insert	Select
Solution 1: Sorted Array	O(n)	O(1)
Solution 2: Unsorted Array	O(1)	O(n)

Solution 3: Use a (balanced) tree *

- Use rank (index of the element in the sorted array)
  ↳ do not store ranks in every node ⇒ need update all nodes per insert.
  ↳ Store size (weight) of sub-tree in every node
  - w(leaf) = 1
  - w(v) = w(v.left) + w(v.right) + 1.
  - "rank in sub-tree" = left.weight + 1.

↳ select(k)

```
rank = n.left.weight + 1;
if (k == rank) then
    return v;
else if (k < rank) then
    return n.left.select(k);
else if (k > rank) then
    return n.right.select(k − rank);
```

- examples:
  - find the k^{th} tallest student in the class. → select(k) : find the node with rank k.
  - determine the ancestor of Johnny's height.
    Is Johnny in the left ancestor or right ancestor? → rank(v)

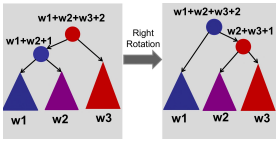
- rank(node)

```
rank = node.left.weight + 1;
while (node != null) do
    if (node is left child) then
        do nothing
    else if (node is right child) then
        rank += node.parent.left.weight + 1   // * Take into account that parent is smaller than node.
    node = node.parent;
return rank;
```

Basic Methodology:

1. Choose underlying data structure
   AVL tree
2. Determine additional info needed. Weight of each node
3. Modify data structure to maintain additional info when the structure changes
   Update weight as needed.
4. Develop new operations.
   Select and Rank.

Maintain weight during insertion:
- maintain weight during rotations

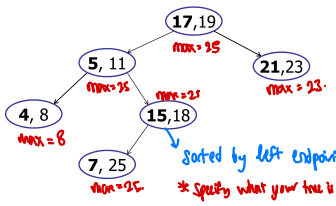


Interval Queries

- Solutions where space is (near) linear in the number of things stored.
- operation are logarithmic in number of things stored.

- given an interval, (a, b), augment: maximum endpoint in subtree

max = { node.right, left.max, right.max }



sorted by left endpoint.
* Specify what your tree is sorted by.

- note: maintain max during rotation → nodes that are involved in rotations.

interval-search(x): find interval containing x. O(logn).

```
c = root;
while (c != null and x is not in c.interval) do
    if (c.left == null) then
        c = c.right;
    else if (x > c.left.max) then
        c = c.right;
    else c = c.left;
return c.interval;
```

- if search goes right, then there is no overlap in the left subtree.
- if a search goes left, then safe to go left.
- if a search goes left and fails, then key < every interval in right subtree.

List all intervals that overlap at their points: O((k+1)logn) ⇓ no. of overlapping intervals.

All-Overlap algorithm:

Repeat until no more intervals:
- search for an interval
- add to list
- delete interval.

add & delete is cheap.

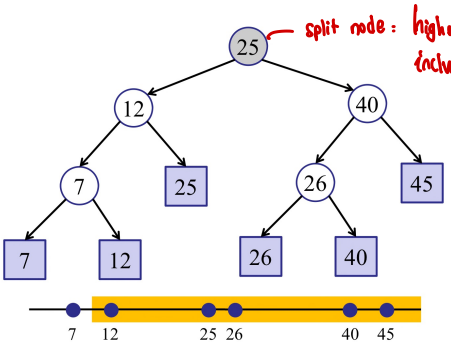
Repeat for all intervals on list:
- add intervals back to tree.

Faster algorithm: O(k+logn).
aka orthogonal range searching.

Orthogonal Range Searching

One dimension

- use a binary tree
- Store all pts in the leaves of the tree.
  - internal nodes store only copies.
- Each internal node v stores the MAX of any leaf in the left sub-tree.



split node: highest node that includes both left and right subtree.

Algorithm:

- find the split node. v. findSplit (low, high)
- Do left traversal. LeftTraversal (v.left, low, high)
- Do right traversal. RightTraversal (v.right, low, high).

```
findSplit(low, high)   O(logn)
    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);
    }                    ↓ found split
    return v;
```

```
LeftTraversal (v, low, high)   O(logn)
    if (low <= v.key) {
        all-leaf-traversal(v.right)   ← output all right subtree and recurse left
        LeftTraversal (v.left, low, high);
    }
    else {
        LeftTraversal (v.right, low, high);
    }                          ↓ recurse right.
}
```

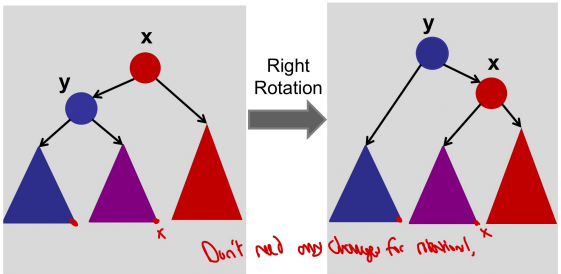
```
RightTraversal (v, low, high)   O(logn)
    if ( v.key <= high) {
        all-leaf-traversal(v.left)   ← output all left subtree and recurse right
        RightTraversal (v.right, low, high);
    }
    else {
        RightTraversal (v.left, low, high);
    }                          ↓ recurse left.
}
```

Invariant:

The search interval for a left-traversal at node v including the max item in the subtree rooted at v

all-leaf-traversal → O(k), k = num of items found

Rotations:



Count Points!

- augment the tree
- keep a count of the number of nodes in each sub-tree.
- Instead of walking the entire sub-tree, just remember the count.

```
LeftTraversal(v, low, high)
    if (low <= v.key) {
        all-leaf-traversal(v.right);
        total += (v.right.count);   ← Keep track of the count
        LeftTraversal(v.left, low, high);
    }
    else {
        LeftTraversal(v.right, low, high);
    }
}
```

∴ Query Time Complexity = O(k+logn)

↓ k is the no. of pts. found.

Preprocessing (build tree) Time Complexity = O(nlogn)

Total space Complexity = O(n).

- each pt stored once + O(n) guide nodes.

Symbol Table

- no order. – no predecessor/successor
- search by name
- no comparison
- supports insert, search, delete, contains, size.
  O(1) O(1)
- Sorting
  - Dictionary? ⇒ AVL Tree O(n log n)
  - Symbol Table? O(n²)
- Direct Access Tables

0	null
1	null
2	item1
3	null
4	Seth
5	item3
6	null
7	null
8	item2
9	null

O(1) insert
O(1) search

How to get index?

↳ Hashing.

Hash Functions

↳ There are a lot of possible keys, but very small table.

↳ Define $h: U \to \{1 \dots m\}$
- Store key k in bucket $h(k)$
- Time: O(time to compute + time to access bucket)
  ≈ O(1 + search).
  unless long strings.

↳ Collision
- 2 distinct keys k_1 and k_2 collide if: $h(k_1) = h(k_2)$
- Unavoidable by pigeon hole principle.
- Solved by chaining & open addressing.

Chaining.

- put both items in the same bucket!

0	null
1	null
2	List Head → A → C → J
3	null
4	null
5	null
6	null
7	null
8	List Head → B
9	null

↳ Each bucket contains a linked list of items

↳ insert: calc. h(key) O(1)
(using h(key)), add (key, value) to list.
cost: O(1 + cost(h))
cost of computing hash function
higher for duplicates, need to search for them.

↳ search: calc. h(key)
search for (key, value) in list.
cost: O(1 + cost(h))
time depends on length of list.

Note: if all keys in same bucket – O(n²).
for sorting.

Total space = O(m+n)
no. of buckets
size of linked list.

Simple Uniform Hashing Assumption

Note: randomly choose bucket would improve searching later steps

- each key is equally likely to map to every bucket (random bucket)
- keys are mapped independently.
- as long as there is enough bucket, there won't be too many keys in one bucket.
- Define: load (hash table) = n/m
  = average number of items / buckets.
- Expected search time = 1 + expected no. of items per bucket.
  - 1: hash function + array alloc
  - n/m: linked list traversal.
- note: P(chose 1 bucket) = 1/m.
- Expected search time = 1 + n/m (assume m=2n).
  = O(1)
- Worst case search time = O(n)
- Worst case insertion time = O(1)
- Max cost to insert n items in hash table: O(log n) ≈ Θ(log n / log log n).
  ↳ max no. of items in 1 bucket.

Table size ≈ m = Θ(n)

Since Expected search time : O(1 + n/m)

Don't know how many keys we need to store.

Grow table?

↳ choose new size m.

↳ choose new hash function (related to table size).

↳ for every item, compute & copy to new table.

expensive!

assume $m_1 < n$ – grow the table once to full $m_2 > 2n$.

cost: $O(m_1 + m_2 + n)$
≈ O(n).
initializing a table of size m_2
times × time.