Relationship between order notations

$f(n) \in \Theta(g(n))$	$\Leftrightarrow g(n) \in \Theta(f(n))$
$f(n) \in O(g(n))$	$\Leftrightarrow \ g(n) \in \Omega(f(n))$
$f(n) \in o(g(n))$	$\Leftrightarrow \ g(n) \in \omega(f(n))$
$f(n) \in o(g(n))$	$\Rightarrow f(n) \in O(g(n))$
$f(n) \in o(g(n))$	$\Rightarrow f(n) \notin \Omega(g(n))$
$f(n) \in \omega(g(n))$	$\Rightarrow f(n) \in \Omega(g(n))$
$f(n) \in \omega(g(n))$	$\Rightarrow f(n) \notin O(g(n))$

$\log(n) \in o(n)$

$(\log n)^a \in o(n^d)$ (big) (small) for $a, d > 0$

Limit Theorem

For all $n \geq n_0$, $f(n) > 0$, $g(n) > 0$ and $L = \lim_{n\to\infty} \frac{f(n)}{g(n)}$

Then $f(n) \in \begin{cases} o(g(n)) & \text{if } L = 0 \\ \Theta(g(n)) & \text{if } 0 < L < \infty \\ \omega(g(n)) & \text{if } L = \infty \end{cases}$

Recurrence Relation

Recursion	resolves to	example
$T(n) \leq T(n/2) + O(1)$	$T(n) \in O(\log n)$	binary-search
$T(n) \leq 2T(n/2) + O(n)$	$T(n) \in O(n \log n)$	merge-sort
$T(n) \leq 2T(n/2) + O(\log n)$	$T(n) \in O(n)$	heapify (*)
$T(n) \leq cT(n-1) + O(1)$ for some $c < 1$	$T(n) \in O(1)$	avg-case analysis (*)
$T(n) \leq 2T(n/4) + O(1)$	$T(n) \in O(\sqrt{n})$	range-search (*)
$T(n) \leq T(\sqrt{n}) + O(\sqrt{n})$	$T(n) \in O(\sqrt{n})$	interpol. search (*)
$T(n) \leq T(\sqrt{n}) + O(1)$	$T(n) \in O(\log \log n)$	interpol. search (*)

O-notation $f(n) \in O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. $|f(n)| \leq c\,|g(n)|$ for all $n \geq n_0$

Ω-notation $f(n) \in \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. $c\,|g(n)| \leq |f(n)|$ for all $n \geq n_0$

Θ-notation $f(n) \in \Theta(g(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ s.t. $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ for all $n \geq n_0$

o-notation

$f(n) \in o(g(n))$ if **for all constants** $c > 0$, there exists a constant $n_0 \geq 0$ s.t. $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$

ω-notation

$f(n) \in \omega(g(n))$ if for all constants $c > 0$, there exists a constant $n_0 \geq 0$ s.t. $0 \leq c\,|g(n)| \leq |f(n)|$ for all $n \geq n_0$

$$\sum_{i=1}^{n}\sum_{j=i}^{n} c \leq \sum_{i=1}^{n}\sum_{j=1}^{n} c \qquad \sum_{i=1}^{n}\sum_{j=i}^{n} c \geq \sum_{i=1}^{\frac{n}{2}}\sum_{j=\frac{n}{2}}^{n} c$$

Use Θ to compare algorithms

MergeSort: $O(n \log n)$ Merge: $O(n)$

MergeSort is stable

Algebra for Order Notation

1. $f(n) \in \Theta(f(n))$
2. $f(n) \in O(g(n)) \cdot g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$
   $f(n) \in \Omega(g(n)) \cdot g(n) \in \Omega(h(n)) \Rightarrow f(n) \in \Omega(h(n))$
   $f(n) \in O(g(n)) \cdot g(n) \in o(h(n)) \Rightarrow f(n) \in o(h(n))$
3. $f(n) + g(n) \in \Omega(\max\{f(n), g(n)\})$
   $f(n) + g(n) \in O(\max\{f(n), g(n)\})$

Useful Sums

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \qquad \sum_{i=0}^{n-1} 2^i = 2^n - 1$$

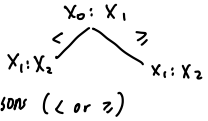
$$\sum_{i=0}^{n-1} (a + di) = na + \frac{dn(n-1)}{2} \in \Theta(n^2) \text{ if } d \neq 0$$

$$\sum_{i=0}^{n-1} ar^i = \begin{cases} a\frac{r^n - 1}{r-1} \in \Theta(r^{n-1}) & \text{if } r > 1 \\ na \in \Theta(n) & \text{if } r = 1 \\ a\frac{1-r^n}{1-r} \in \Theta(1) & \text{if } r < 1 \end{cases}$$

$$\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6} \in \Theta(1) \qquad \sum_{i=1}^{n} i^k \in \Theta(n^{k+1}) \text{ for } k \geq 0$$

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = \frac{1}{2} \in \Theta(1) \qquad \sum_{i=1}^{\infty} ip(1-p)^{i-1} = \frac{1}{p} \text{ for } 0 < p < 1$$

Decision Tree

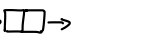


- Each nodes are comparisons ($<$ or $\geq$)
- Tree height h is the worst case num of comparisons

Comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons

Stack ADT (LIFO)

Queue ADT (FIFO)



Priority Queue ADT (stores items with priority)

- insert: insert an item tagged with a priority
- deleteMax: remove and return the item of highest priority

PQ-Sort $O(\text{init} + n \cdot \text{insert} + n \cdot \text{deleteMax})$ Space: $O(n)$

```
init PQ to an empty priority queue
for i←0 to n-1 do
    PQ.insert(A[i])
for i←n-1 downto 0 do
    A[i]←PQ.deleteMax()
```

Binary Tree

nodes $n \leq 2^{h+1} - 1 \Rightarrow h \geq \log(n+1) - 1 \in \Omega(\log n)$

height $\log(n+1) - 1 \leq h \leq n - 1$

Max-oriented binary heap
- all levels are filled, left adjusted
- key[parent(i)] ≥ key[i]

Height of a heap with n nodes is $\Theta(\log n)$

Left: $2i+1$ Right: $2i+2$ Parent: $\lfloor \frac{i-1}{2} \rfloor$

Fix-up(A, i) $O(h) = O(\log(n))$

```
while parent(i) and A[parent(i)].key < A[i].key do
    swap A[i] and A[parent(i)]
    i←parent(i)
```

Insert(x) $O(\log n)$

```
increase size
l←last()
A[l]←x
fixup(A, l)
```

fix-down(A, i, n) $O(h) = O(\log n)$

```
while i is not a leaf do
    j←left(i)
    if i has right and A[right].key > A[left].key then
        j←right(i)
    if A[i].key ≥ A[j].key break
    swap A[i] and A[j]
    i←j
```

delMax() $O(\log n)$

```
l←last()
toReturn = A[root()]
A[root()] = A[l]
size--
fix-down(A, root(), size)
return toReturn
```

heapify(A) $\Theta(n)$

```
for i←parent(last()) downto 0 do
    fix-down(A, i)
```

A heap can be built in linear time if we know all items in advance

HeapSort(A) $\Theta(n \log n)$

```
n←A.size()
heapify(A)
while n > 1 do
    swap A[root()] and A[last()]
    n--
    fix-down(A, root(), n)
```

QuickSort Improves(A, n)

```
initialize a stack S of index-pairs with {(0, n-1)}
while S is not empty
    (l, r)←S.pop()
    while r-l+1 > 10
        p←choose-pivot(A, l, r)
        i←partition(A, l, r, p)
        if i-1 > r-i do
            S.push((l, i-1))   Store larger problem in S
            l←i+1              work on smaller
        else
            S.push((i+1, r))
            r←i-1
InsertionSort(A)
```

Proof: decision tree must have at least $n!$ leaves to store all permutations

h must be at least $2^h \geq n!$

$$h \geq \log(n!) = \log(n) + \ldots + \log\left(\frac{n}{2}+1\right) + \log\left(\frac{n}{2}\right) + \ldots + \log(1)$$
$$\geq \frac{n}{2}\log\frac{n}{2} = \frac{n}{2}\log n - \frac{n}{2} \in \Omega(n \log n)$$

Average Runtime

$$T^{avg}(n) = \frac{\sum_{I \in I_n} T(I)}{|I_n|}$$

(# permutations with at least k comp) − (# permutations with at least $k+1$ comp) / (# permutations with exactly k comp)

Expected Runtime (for randomized)

$$T^{exp}(I) = \sum_{\text{all possible sequences } R} T(I,R) \cdot Pr(R)$$
$$= \sum_{\langle x, R' \rangle} Pr(x) Pr(R')\, T(I, \langle x, R' \rangle)$$

$$T^{exp}(n) = \max_{A \in I_n} \sum_{R} T(I,R) \cdot Pr(R) \text{ (Worst case)}$$

$$\sum_{\langle r_1, r_2 \rangle} c \cdot r_1 \cdot r_2 = \sum_{r_1} c \cdot r_1 \sum_{r_2 \in [0 \ldots n-1]} r_2$$

$$\sum_{R'} T(A[0, \ldots, \tfrac{n}{2}-1], R') \cdot Pr(R') \leq \max_{A' \in I_{\frac{n}{2}}} \sum_{R'} T(A', R') \cdot Pr(R')$$
$$= T^{exp}\left(\frac{n}{2}\right)$$

partition(A, p) $\Theta(n)$

```
swap(A[n-1], A[p])
i←-1  j←n-1  v←A[n-1]
loop
  do i←i+1 while A[i] < v
  do j←j-1 while j>i and A[j]>v
  if i≥j then break
  else swap(A[i], A[j])
end loop
swap(A[n-1], A[i])
return i
```

QuickSelect(A, k)

```
p←choose-pivot(A)
i←partition(A, p)
if i=k then return A[i]
else if i>k then
  return QS(A[0..i-1], k)
else if i<k then
  return QS(A[i+1..n-1], k-(i+1))
```

Best case: $\Theta(n)$
Worst case: $\Theta(n^2)$
$T^{avg}(n) = T^{exp}(n) = \Theta(n)$

Quick Sort(A) $T^{exp}(n) = T^{avg}(n) = O(n \log n)$

```
if n≤1 then return
p←choose-pivot(A)
i←partition(A, p)
QuickSort(A[0,...,i-1])
QuickSort(A[i+1,...,n-1])
```

Non-Comparison-Based Sorting (need to make assumptions)

Represent numbers in base R
- digits go from 0 to $R-1$ (larger R → smaller m)
- R buckets $R = 2, 10, 128, 256, \ldots$
- number are in the range $[0, 1, \ldots, R^m - 1]$

Single-Digit-Bucket-Sort(A, d) space/time: $\Theta(n+R)$ Sorting is stable

```
init B[0,...,R-1] of empty lists (buckets)
for i←0 to n-1 do
    next←A[i]
    append next at end of B[dth digit of next]
i←0
for j←0 to R-1 do
    while B[j] is non-empty do
        move first element of B[j] to A[i++]
```

MSD-Radix-Sort(A, l←0, r←n-1, d←leading digit index)

```
if l<r
  single-digit-bucket-sort(A[l..r], d)
  if there are digits left then
    l'←l
    while l'<r do
      Let r'≥l' be the maximal s.t. A[l'..r'] have same dth digit
      MSD-Radix-Sort(A, l', r', d+1)
      l'←r'+1
```

time: $O(mnR)$ space: $\Theta(mn+R)$

Advantage: many digits may remain unchecked

Drawback: many recursions

LSD-radix-sort(A) time: $\Theta(m(n+R))$ space: $\Theta(n+R)$

```
for d←least significant downto most do
    bucket-sort(A, d)
```

BST: delete search / insert $\Theta(h)$

Case 1: if x has an empty subtree
- delete x
- if x has a parent, reconnect the other subtree of x to parent of x

Case 2: if x has both nonempty
- swap KVP of x with KVP at successor node (smallest key node in the right subtree)
- delete successor node (case 1 applies)

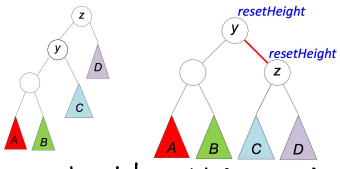
BST height: best case $\Theta(\log n)$ worst case $\Theta(n)$
insert at random $\Theta(\log n)$

AVL Tree

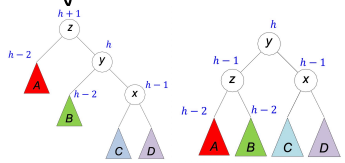
Balance factor = height(v.right) − height(v.left)

AVL tree on n nodes has $\Theta(\log n)$ height

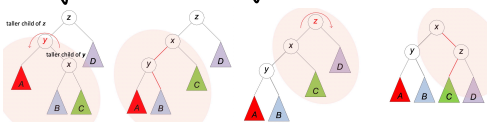
Left left Imbalance (Right Rotation)



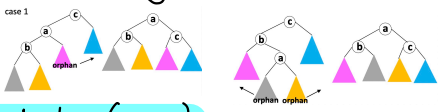
Right Right Imbalance (Left Rotation)



Left Right Imbalance (double right rotation)



Tri-node Restructuring



restructure(x,y,z)

```
Case: return rotate_right(z) //left left imbalance
Case: z.left ← rotate_left(y) //left right imbalance
      return rotate_right(z)
Case: z.right ← rotate_right(y)
      return rotate_left(z)
Case: return rotate_left(z)
```

AVL: insert(k,v) $\Theta(h)$

```
z ← BST.insert(k,v)
while (z is not NIL)
    if (balance > 1) then
        y ← tallest child of z
        x ← tallest child of y
        z ← restructure(x,y,z)
        break
    setHeightFromSubtrees(z)
    z ← parent of z
```

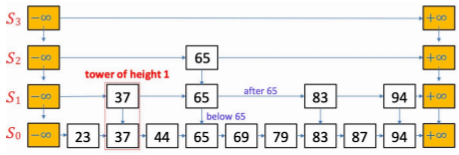
setHeightFromSubtree(u)

```
if u is not an empty subtree
    u.height ← 1 + max(left, right)
```

AVL: delete(k) $\Theta(h)$

```
z ← BST.delete(k)
while (z is not NIL)
    if (balance > 1) then
        y ← tallest child of z
        x ← tallest child of y
        z ← restructure(x,y,z) //same like ← may call Θ(h) times
    setHeightFromSubtrees(z)
    z ← parent of z
```

Skip List



At each level, predecessor of key k is
- if k is present: node before node with k
- if not: node before node where k would have been

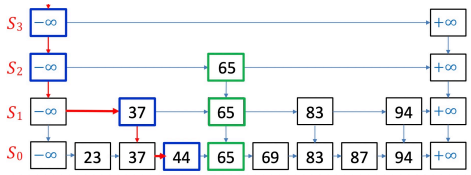
getPredecessors(k) $O(\log n)$

```
p ← root
P ← stack of nodes, initially containing p
while p.below ≠ NIL do
    p ← p.below
    while p.after.key < k do
        p ← p.after
    P.push(p)
return P
```

SL::search(k)

```
P ← getPredecessors(k)
q ← P.top()
if q.after.key = k
return q.after
else
return "not found"
```

Insert



Expected length of $S_i = \frac{n}{2^i}$

$E[h] \le 2 + \log(n)$

Search in Unordered Array

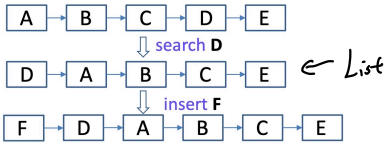
Optimal Static Ordering (know distribution)

key	A	B	C	D	E
frequency of access	2	8	1	10	5
access probability	2/26	8/26	1/26	10/26	5/26

Minimize expected cost.

Dynamic Ordering

MTF heuristic (insert/search from back if in array)



← List

Transpose heuristic

Avoid dramatic change but still adapting to access patterns

Frequency-count heuristic

keep counters how often items were accessed, and sort in non-decreasing order

compressed-trie::insert(w) and delete(w)

1. search for path P to word w
2. uncompressed this path
3. insert/delete w as in uncompressed trie
4. compress path from root to where changes happened

$\Omega(\log n)$ comparisons required for search in Comparison based model

Interpolation Search (A,n,k)

```
l ← 0, r ← n-1
while (l ≤ r)
    if (k < A[l] or k > A[r]) return "not found"
    if (k = A[r]) return "found at A[r]"
    m ← l + ⌊(k-A[l])/(A[r]-A[l]) (r-l)⌋
    if A[m] = k return "found at A[m]"
    else if (A[m] < k)
        l ← m+1
    elsif r ← m-1
```

use interpolation search for $\log n$ steps, then switch to binary search

$O(\log n)$ worst case, but could be $O(\log\log n)$

Trie

insert/search/delete $\in O(|w|)$ time

worst case space $\Theta(n \cdot \max \text{word length})$

Leaf references (longest word in subtree)

get_path_to(w)

```
P ← empty stack; z ← root; d ← 0; P.push(z)
while d ≤ |w|
    if z has a child-link labelled with w[d]
        z ← child at this link; d++; P.push(z)
    else break
return P
```

Trie::search(w)

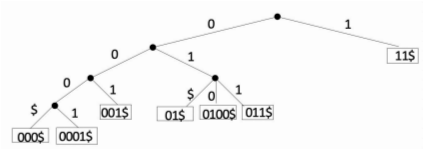
```
P ← get_path_to(w); z ← P.top()
if z is not a leaf: "not found"
```

Prefix-search(w)

If match, stack size = prefix size + 1

Pruned Trie

Sub-trie with one key has only one node



Compressed-Trie Space: $O(n)$

If each internal node has 2 or more children, then there are at most n−1 internal nodes. (n leaves)

So at most 2n−1 total nodes.

Patricia-Trie