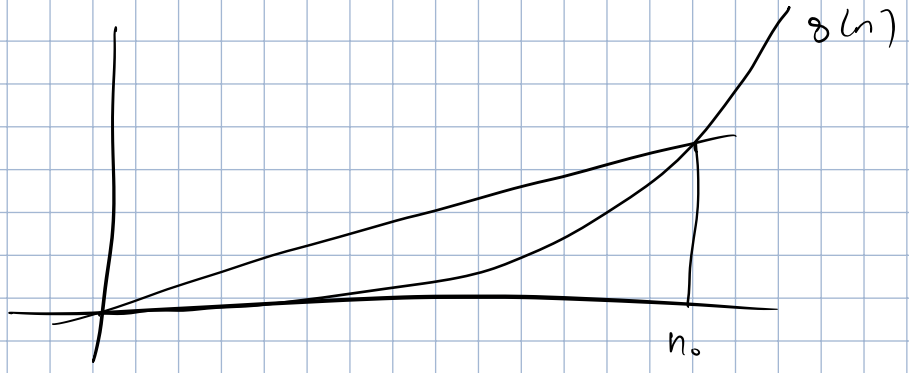
WEEK 1

Analysis of algorithm
- Runtime → Experiment
- Runtime → Size of the problem relation
- Memory

Asymptotic: growth rate of algorithm.

① Big O: upper bound on g.c.

$f(n) \in O(g(n)): \exists c > 0,\ n_0 \geq 0.\ \forall n \geq n_0 \Rightarrow |f(n)| \leq c|g(n)|$



Prove:

1. Use inequality to derive c and n_0.

$$2n^2 + 3n + 11 \in O(n^2)$$

$$2n^2 + 3n + 11 \leq 2n^2 + 3n^2 + 11n^2$$

$$\leq 16n^2 \quad \forall n \geq 1$$

Properties:

1. $a f(n) \in O(f(n))$
2. $f(n) \in O(h(n)) \wedge h(n) \in O(g(n)) \Rightarrow f(n) \in O(g(n))$ — Also applies to Ω
3. $\max(f(n), g(n)) \in O(f(n) + g(n))$
4. $a_0 + a_1 n + \ldots + a_k n^k \in O(n^k)$
5. $n^x \in O(a^n) \quad x > 0,\ a > 1$

n^x: Polynomial → a^n: Exponential

6. $(\log_3 n)^2 \in O(n^2)$ $a, b \geq 0$

($\log_3 n$: log; n^2: polynomial)

7. $O(f(n) + g(n)) \in O(\max\{f(n), g(n)\})$ → Also applies to Ω

② Big omega: lower bound or s.c.

$\exists c > 0, n_0 > 0 . \forall n > n_0 \Rightarrow |f(n)| \geq c|g(n)|$, then $f(n) \in \Omega(g(n))$

③ Big theta: tight bound on s.c.

1. $f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n)) \Rightarrow f(n) \in \Theta(g(n))$
2. $\exists c_1, c_2 > 0, n_0 > 0 . \forall n > n_0 \Rightarrow c_1|g(n)| \leq f(n) \leq c_2|g(n)|$

Tricks:

Key property: $\log n \leq n \quad \forall n \geq 2$

Fractional:

$\frac{1}{2}n^2 - 5n \in \Omega(n^2)$

1. Smaller constant than leading term.

$\frac{1}{2}n^2 - 5n \geq \frac{1}{4}n^2$

$\frac{1}{4}n^2 + \frac{1}{4}n^2 - 5n \geq \frac{1}{4}n^2$

$\frac{1}{4}n^2 - 5n \geq 0$

$n^2 - 20n \geq 0$

$n(n-20) \geq 0$

$\therefore n \geq 20$ ✓

$c = \frac{1}{4}$, $n_0 = 20$

④ Little o: $g(n)$ dominate $f(n)$

Stricter than Big O

$\forall c > 0, \exists n_0 > 0 . \forall n > n_0 \Rightarrow |f(n)| < c(g(n))$

Trick: set arbitrary $c \Rightarrow$ find n_0 to satisfy inequality

Use intermediary

$2020n^2 + 1388n \in o(n^3)$

Let $c > 0$. $h(n) \Rightarrow$ int. func.

Similar to more complicated func. + include c

$$2020n^2 + 1388n \leq h(n) \leq cn^3$$

$$2020n^2 + 1388n \leq 5000n^2 \leq cn^3$$

obv. — Non-obv.

$$5000\ n^2 \leq n^3$$

$$5000\ n^2 < cn^3$$

$$5000 \leq cn$$

$$n \geq \frac{5000}{c}$$

n_0 !

⑤ Little omega: $f(n)$ dominates $g(n)$

Exact same defn., switch inequality as o

Limit technique:

Not sufficient for proof

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)} \Rightarrow f(n) \in \begin{cases} o(n) \Rightarrow L = 0 \\ \Theta(n) \Rightarrow 0 < L < \infty \\ \omega(n) \Rightarrow L = \infty \end{cases}$$

↳ Use L'Hôpital's

Doubling:

a) Constant:

$$T(n) = c \Rightarrow T(2n) = c$$

b) Logarithmic:

$$T(n) = c\log n \Rightarrow T(2n) = c\log(2n)$$
$$= c\log 2 + c\log n$$
$$= T(n) + c$$

c) Linear:

$$T(n) = cn \Rightarrow T(2n) = 2T(n)$$

d) Linearithmic:

$T(n) = cn\log n \Rightarrow T(2n) = c(2n)\log(2n)$

$= 2cn\log 2 + 2cn\log n$ ← $2T(n)$

$= 2T(n) + cn$

e) Quadratic:

$T(n) = cn^2 \Rightarrow T(2n) = 4T(n)$

WEEK 2

How to crete $T(n)$?

Strategy: count # of atomic ops $\Rightarrow \Theta(1)$

↳ # of iterations = runtime.

Calculating # of itrs

Tight

Construct exact summation + evaluate.

Often much more difficult.

Sloppy

Prove $O(\ldots) \wedge \Omega(\ldots)$

How? Mess around w/ sums + limits of sum

Evaluate to some func. ←

O: sum over maximum value

$$\sum_{i=0}^{n} i \leq \sum_{i=0}^{n} n = n^2$$

↳ Remove var from summ.

Sum over as many iterations as possible.

Ω: Sum over minimum value

$$\sum_{i=0}^{n} i \geq \sum_{i=0}^{n} \frac{n}{2} = \frac{n^2}{2}$$

Sum over as little itr as possible.

$$\sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=i}^{j} 1 \geq \sum_{i=1}^{n/3} \sum_{j=1}^{n/3} \sum_{k=1}^{n/3} = \frac{n^3}{3}$$

While loops?

while cond:

- ~
- ~

① What variable det. cond.

② start value + op. on value.

③ Let k = # of iterations until cond. breaks $\Rightarrow$ Solve for $k \Rightarrow$ use in sums.

```
sum ← 0
for i ← 1 to n do
    j ← i
    while j ≥ 1 do
        sum ← sum + i/j
        j ← ⌊j/2⌋
```

$$T(n) = \sum_{i=1}^{n} \#\text{ of while loop iter.}$$

① $j : i \to 1$

② Let k = # of while iter.

$$i \cdot \left(\frac{1}{2}\right)^k = 1$$

$$\log i - k = 1$$

$$k = \log i - 1$$

→ # of iter $\leq \lfloor k \rfloor + 1$

$$T(n) = \sum_{i=1}^{n} k = \sum_{i=1}^{n} (\lfloor \log i \rfloor + 1)$$

O:

$$T(n) \leq \sum_{i=1}^{n} (\log n + 1) = n \log n + n$$

Ω:

$$\sum_{i=1}^{n/2} (\log n + 1) = \frac{n}{2} \log n + \frac{n}{2}$$

$$\in \Omega(n \log n + n)$$

* 2 instances of same size not guaranteed to have same $T(n)$*

↳ Sorting.

↳ Consider: worst instance, average instance

Compare 2 diff algos? Use θ, not O, use worst-case

Recurrences:

① Expand + find pattern

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

$$= 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn$$

$$= 4T\left(\frac{n}{4}\right) + cn + cn$$

$$= 4\left(T\left(\frac{n}{8}\right) + c\frac{n}{4}\right) + cn + cn$$

$$= \vdots$$

$$= nT(1) + \left[2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + 8 \cdot \frac{n}{8} + \dots + \frac{n}{2} \cdot \frac{n}{n/2}\right]$$

$$= nT(1) + n \log n \in \theta(n \log n)$$

$n \times \left(\frac{1}{2}\right)^k = 1$

$\log n - k = 1$

$k = \log n - 1$

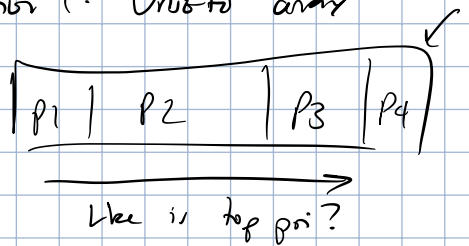
Priority Queue ADT:

Insert item w/ priorities ⇒ access + delete highest pri item.

Operations:
- insert
- deleteMax

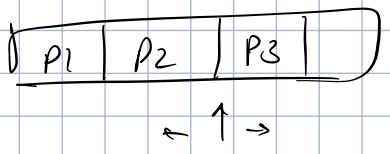
Note: minimum-oriented PQ

Realization 1: Unsorted array



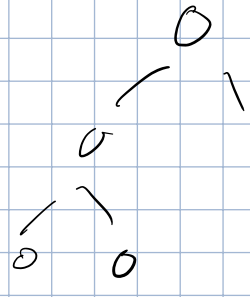
Insertion: $O(1)$
Deletion: $O(n)$

Realization 2: sorted array



Insert: $O(n)$
Deletion: $O(1)$

Realization 3: binary heap



① Every node > children ⇒ Heap order

② Level is complete except last one (left justified) ⇒ Structural prop.

Height: $\Theta(\log n)$

Store priorities in binary heap!

Code: Array:

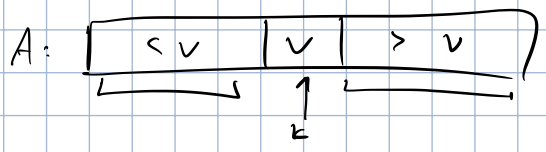
Left child of node $[i]$: node $[2i+1]$
Right " : node $[2i+2]$

Root: $A[1]$
Last node: $A[n-1]$
Parent: $\lfloor \frac{i-1}{2} \rfloor$

Module 3: Sorting + Randomized Algos

Quick Select

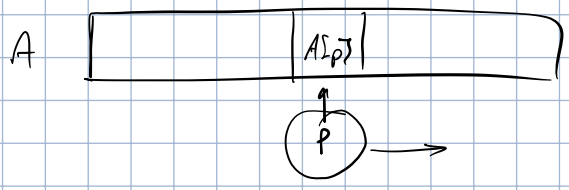
Given value v, find index of v if A sorted



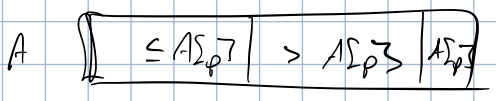
Given index k, what value must k in k if sorted?

Quick select:

1. Choosing pivot



2. Partitioning around pivot



How?

1. Choosing pivot: return the last index

2. Partition:

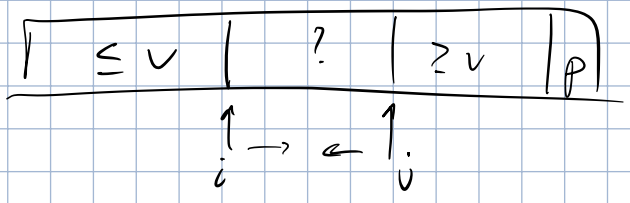
i) Simple

```
partition(A, p)
    small ← []
    equal ← []
    large ← []
    v ← A[p]
    Loop through → add elem in list
    append [small, equal, large]
```

Time: $\Theta(n)$

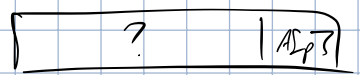
Space: $\Theta(n)$
↳ Array with everything same!

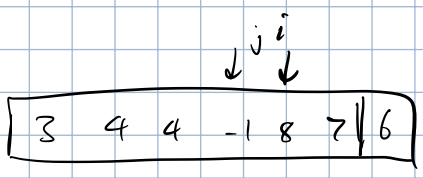
ii) Hoare: in-place



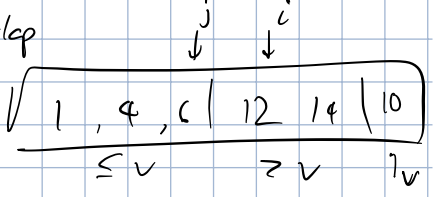
Swap values at $A[i]$ + $A[j]$ according to pivot $\Rightarrow i \geq j$

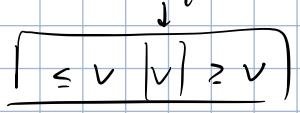
```
partition (A, p)
    // Ensure pivot is last
    swap(A[n-1], A[p])
    i ← -1, j ← n-1
    v ← A[n-1]

    loop
        do i ← i+1 while A[i] < v
        do j ← j-1 while A[j] > v and j ≥ i
        // Make sure we didn't blo of overlap
        if i ≥ j: break
        swap(A[i], A[j])
    end loop
    // Everything below i is less than pivot ⇒
    swap(A[n-1], A[i])
        ↑ pivot
    return i
```





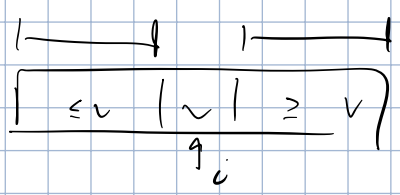
Efficient for duplicates.





Put all this together:

```
quick-select I (A, k)
    p ← choose pivot(A)
    i ← partition(A, p)
    if i == k:
        return A[i]
    else if k < i:
        return quick-select(A[0...i-1], k)
    else:
        return quick-select(A[i+1:n-1], k-i+1)
```



→ 0.w. ⇒ overwrite i.

→ length of ≤ v subarr.

Analysis: * Tricks *

1. Best-case analysis:

Think: what situation leads to fast outcome.

Don't recurse: $i = k \Rightarrow \Theta(n)$

2. Worst case analysis

Think: situation goes poorly ⇒ maximize looping, recursions ...

$$T(n) = \begin{cases} cn + T(n-1) & \Rightarrow i = 1, \text{ or } 0 \quad (n \geq 1) \\ c & n = 1 \end{cases}$$

① Find out generalized expression

$$T(n) = cn + T(n-1)$$
$$= cn + c(n-1) + T(n-2)$$
$$= cn + c(n-1) + c(n-2) + T(n-3)$$
$$= \vdots$$
$$= cn + c(n-1) + \dots + c(n-i) + T(n-i-1)$$

② Figure out when generalized expression = base case

$$n - i - 1 = 1$$
$$i = n - 2$$

③ Substitute + write summation

$$T(n) = cn + \dots + 2c + T(1)$$
$$= c \sum_{i=2}^{n} i + c \quad \in \Theta(n^2)$$

3. Average case analysis

$$T^{avg}(n) = \frac{1}{\#\text{ of instances of size } n} \sum_{I: \text{size}(n)} T(I)$$

① See if you can make assumptions

1. All elements are distinct
2. Sorted from $0 - n-1$

② Find out # of numbers of instances.

$\boxed{n \mid n-1 \mid n-2 \mid \cdots \mid 1}$ $\Rightarrow$ $n!$ # of instances of size n

↳ NO DUPLICATES!

③ Break out sum according to decisions we make

3 decisions:

1. Pivot
2. Partition $\Rightarrow$ Choose pivot to sum on, b/c easier to do
3. Recurse

$$\sum_{i=0}^{n-1} \sum_{\substack{I : size(I)=n \\ pivot = i}} T(I)$$

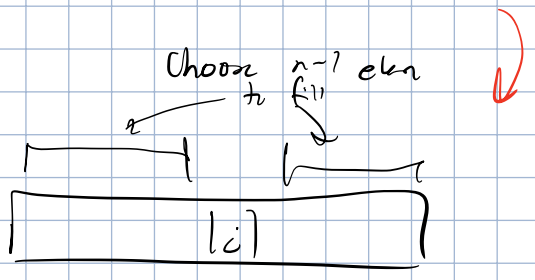
↑ All possible partitions

④ Create upper bound:

$$\sum_{\substack{I : size(I)=n \\ pivot = i}} T(I) \leq \underbrace{(n-1)! \, cn}_{\text{Partition}} + \underbrace{(n-1)! \max\{T(i), T(n-i+1)\}}_{\text{Recursion}}$$

Use recurrence!!

$(n-1)!$: # of possible arrays w/ pivot i



⑤ Simplify sum

$$T^{Avg}(n) \leq \frac{1}{n!} \sum_{i=0}^{n-1} (n-1)! \, cn + (n-1)! \max\{\ldots\}$$

$$\leq \frac{(n-1)! \, cn \; n}{n!} + \frac{1}{n} \sum_{i=0}^{n-1} (n-1)! \max\{\ldots\}$$

$$\leq cn + \frac{1}{n} \sum_{i=0}^{n-1} \max\{T(i), T(n-i+1)\}$$

↑ Bounded by IH (Induction)

$\therefore T^{Avg}(n) \in O(n)$

Randomized Algorithms.

Involves RNG to make decision

Randomic thing ⇒ input doesn't matter ⇒ Worst-case = avg. case

How to do randomized algo analysis (recap)

$$T^{exp}(I) = E[T(I,R)] = \sum_{\text{all possible } R} T(I,R) \cdot P(R)$$

1. Figure out recurrence:

$$T(n) = \begin{cases} ① & n \geq \dots \\ ② & n \leq \dots \end{cases}$$

2. Figure out the probability of either of recurs.

$P(①) = \dots$ $P(②) = \dots$ ⇒ With RNG: this is uniform probability.

3. Expectation:

$P(1 \text{ elem}) = \frac{1}{n}$

$$T^{exp}(n) = T(①)\,P(①) + T(②)\,P(②) + \dots$$

↳ Solve recurrence like ob.

Absolute worst case: unlucky w/ RNG ⇒ exact same thought as worst-case analysis

Quick select ⇒ randomize pivot selection

```
choose_pivot():
    return random(n-1)   → some random index in [0, n-1]
```

$$T^{exp}(n) = \sum_{i=0}^{n-1} \underbrace{T^{exp}(n,i)}_{\text{recurs}} \cdot P(i \text{ is pivot})$$

$1/n$ = uniform

$$\leq \frac{1}{n} \sum_{i=0}^{n-1} cn + \max\{T^{exp}(i), T^{exp}(n-i-1)\}$$

$$\leq cn + \frac{1}{n} \sum_{i=0}^{n-1} \cdots$$

Quick Sort

```
quick_sort(A):
    if n ≤ 1: return
    p ← choose_pivot(A)
    i ← partition(A, p)
```

quick-sort (A[0:i-1])

quick-sort (A[i+1:])

Time Analysis:

① Best case: little work in recursion ⇒ recursions must be eq.

$$T(n) = 2T\left(\frac{n}{2}\right) + cn \quad \in \Theta(n \log_2 n)$$

② Worst-case: maximize recursion

$$T(n) = T(n-1) + cn \quad \in \Theta(n^2)$$

③ Avg. case:

$$T(n) = \frac{1}{n} \sum_{i=0}^{n-1} (cn + T(i) + T(n-i+1))$$

Doesn't matter, only counting comp., not work

① Simplify

Symmetric

$$T(n) = \frac{1}{n} \sum_{i=0}^{n-1} n + \frac{1}{n} \sum_{i=0}^{n-1} T(i) + \frac{1}{n} \sum_{i=0}^{n-1} T(n-i+1)$$

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

② Remove all fractions

$$nT(n) = n^2 + 2 \sum_{i=0}^{n-1} T(i) \longrightarrow T(0) + T(1) \ldots + T(n-2) + T(n-1)$$

③ Remove the sum by considering next smallest/largest term.

Smaller: $(n-1)T(n-1) = (n-1)^2 + 2 \sum_{i=0}^{n-2} T(i)$

$$nT(n-1) - T(n-1) = n^2 - 2n + 1 + 2\left(T(0) + T(1) + \ldots + T(n-2)\right)$$

Subtract this from $T(n)$

$$nT(n) - nT(n-1) + T(n-1) = 2n - 1 + 2T(n-1)$$

$$nT(n) = 2n - 1 + (n+1)T(n-1)$$

Divide + simplify into pseudo-recur

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2n-1}{(n+1)n}$$

④ Change of variables:

$$A(n) = \frac{T(n)}{n+1}$$

$$\therefore A(n) = A(n-1) + \frac{2n-1}{n(n+1)}$$

⑤ Solve recurrence:

$$A(n) = A(n-1) + \frac{2n-1}{n(n+1)}$$

$$= A(n-2) + \frac{2(n-1)-1}{(n-1)n} + \frac{2n-1}{n(n+1)}$$

$$= \vdots$$

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

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

$\Theta(\log n)$ $\quad$ $\Theta(1)$

$$\therefore A(n) \in \Theta(\log n)$$

⑥ Replace CoV:

$$\frac{T(n)}{n+1} \in \Theta(\log n)$$

$$T(n) \in \Theta(n \log n)$$

Improvements:

1. Randomized partition
2. Tail call elimination to reduce recursion
3. Almost-sort: limit on depth of recursion
   ↳ Insertion sort for even better perf.
4. Handle duplicates via 3-way partition.

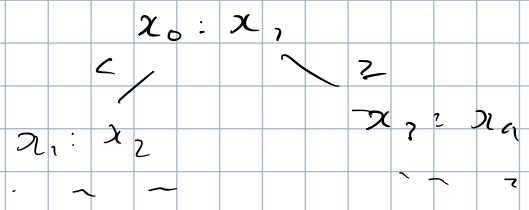
Lower Bound for Comparison-Based Sorting

Lower bound for comp.-based: $\Omega(n \log n)$

↳ Sorting that use $A[i] \leq A[j]$

Proof: decision tree

Decision tree: shows all comparisons



Worst case of a sorting algo ⇒ max number of comp.

↳ Equiv to height of decision tree

Need to figure out height! Use # of leaves

↳ Leaf: represents unique output (e.g. unique sorting, unique selection)

1. Figure out # of unique outputs
2. $\text{decisions}^{\text{height}} \geq \text{leafs}$ ⇒ find lower bound on h.

Ex:// Assume n-distinct elements. Lower bound of sorting using comp.-based algo

① Decisions / node

$\leq, >$ ⇒ 2 decisions

② Leafs:

$[1, 1, 0] \Leftrightarrow [2, 2, 1]$ (same # of comp. ⇒ same leaf)

What matters ⇒ ordering of elem!

∴ n elements ⇒ n! orderings.

③ Lower bound:

$$\text{decision}^{\text{height}} \geq \text{leafs}$$

$$2^h \geq n!$$

$$h \geq \log(n!)$$

$$h \in \Omega(n \log n)$$

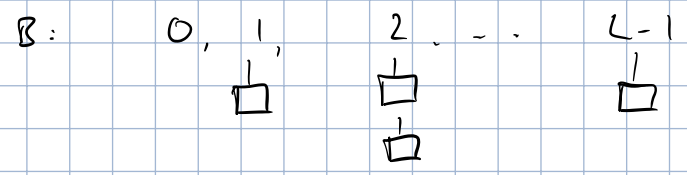
Non-comp. based algorithms

Almost always ⇒ assumptions made (eg. range of elements, structural property, max size)

① Bucket sort

Assumption: array in range $[0, L-1]$

1. Create bucket array from $[0, \dots, L-1]$
2. Iterate through array + add to respective bucket (LL)



3. Iterate through bucket array + pop elements into array

Time: $\Theta(n)$, Space: $\Theta(L+n)$

L: buckets, n: nodes

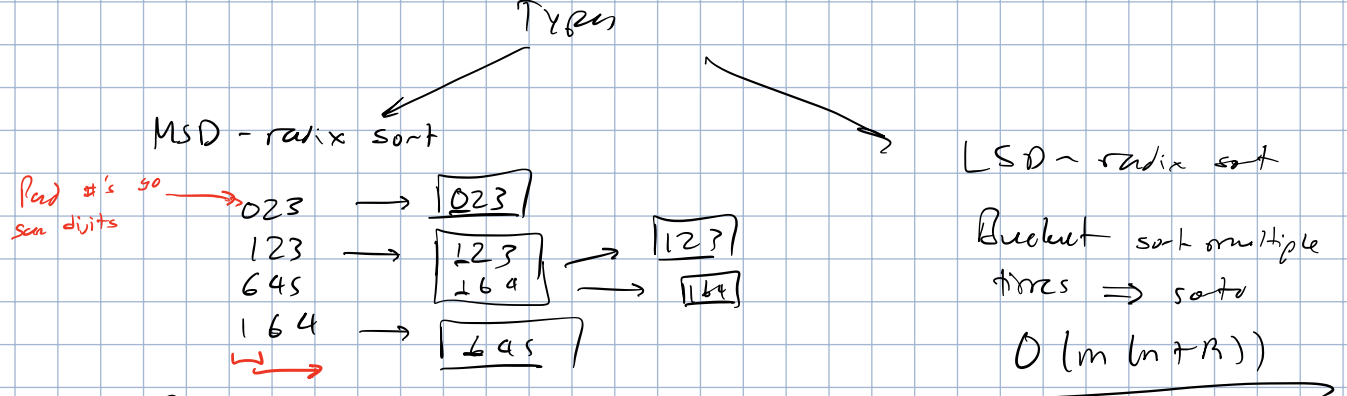
Great if $L \approx n$. If $L \gg n$, inefficient

② Digit-based sorting

Looks at digits one-by-one ⇒ bucket sort on digit ⇒ repeat

↳ Stable sorting

↳ Dependent on base representation. If base R ⇒ R buckets



Recursion-based.

Runtime: $O(n \cdot \overset{\#\text{ of digits}}{m} \cdot R)$

Space: $\Theta(n + m + R)$

Bucket sort multiple times ⇒ sorts

$O(m(n+R))$

Why is this $O(n)$?

If R is known + range ⇒ m can be found.

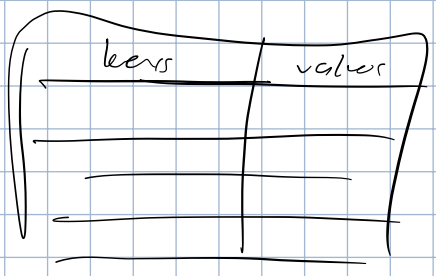
∴ $[0, \dots, 10^4 - 1]$

∴ $m = 4$, $R = 10$

∴ $O(n)$

Module 5 - AVL Trees for Dicts

Dictionary ADT



search (key) → value
insert (k, v)
delete (k)

Implementations:

Comparison + insert (K, V) is in constant time

1. Unordered array list:
   - search: $\Theta(n)$, insert: $\Theta(1)$, Deletion: $\Theta(n)$
2. Ordered array:
   - Search: $\Theta(\lg n)$, insert: $\Theta(n)$, Deletion: $\Theta(n)$
3. AVL tree:

Binary search tree

Search: trivial

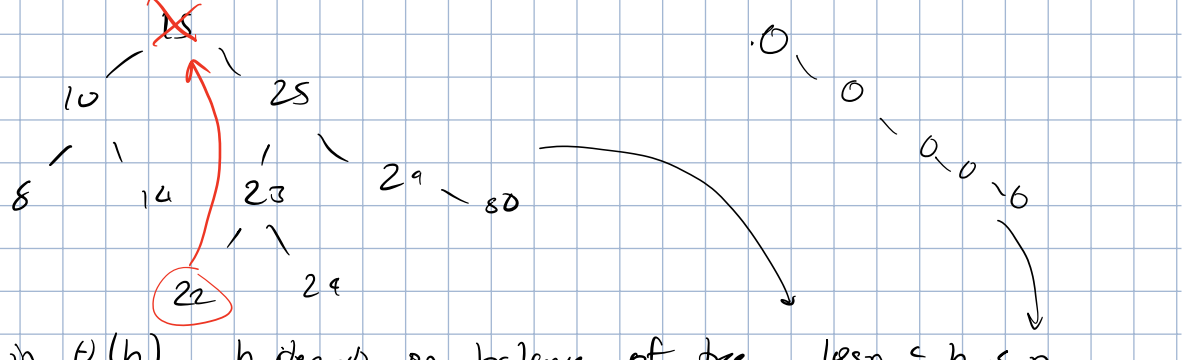
Insertion: search → add into node

Deletion:

Case #1: deleting leaf ⇒ nothing but delete node

Case #2: deleting node w/ 1 child ⇒ move child to current node

Case #3: deleting w/ 2 children ⇒ swap w/ next node in in order traversal (will be leaf) + delete



All operations in $\Theta(h)$. h depends on balance of tree. $\log n \leq h \leq n$

AVL Trees:

Self-balancing BST w/ height-balance prop:

Height-balance prop: $\forall$ nodes in tree, $|\text{height(left)} - \text{height(right)}| \leq 1$

↳ balance(v) := height(R) − height(L) ($\forall v \in T$, balance(v) $\in \{-1, 0, 1\}$)

Store height at each node

Proof that height of AVL tree $\in \Theta(\log n)$.

1. Height of AVL tree $\in \Omega(\log n)$

Let h be height of tree. # of nodes < # of nodes in complete binary tree $(2^{h+1} - 1)$

$$\therefore \quad n \leq 2^{h+1} - 1$$

$$\vdots$$

$$h \geq \log(n+1) - 1 \Rightarrow h \in \Omega(\log n)$$

2. Height of AVL tree $\in O(\log n)$

N(h): nodes for AVL tree of height h (minimum)

$$N(h) = N(h-1) + N(h-2) + 1$$

= Minimum cases.

Proof method!!
Minimum ⇒ balance = −1

Claim $N(h) \geq \sqrt{2}^h - 1$ (prove via induction)

$$n \geq N(h) \geq \sqrt{2}^h - 1$$

$$n + 1 \geq \sqrt{2}^h$$

$$\log(n+1) \geq h/2$$

$$h \leq 2\log(n+1) \quad \in O(\log n)$$

Insertion:

insert(k, v) → z (node of tree)

1. Use regular BST insert
2. Fix (unbalanced)

```
insert (k, v):
    z ← bst_insert (k, v)
    while (z is not null):
        if |z.left − z.right| > 1
            y ← taller child of z
            x ← taller child of y
            z ← restructure (x, y, z)
            break
    setHeightSubtrees (z)
    z ← z.parent
```

→ No balancing for rest of tree!!

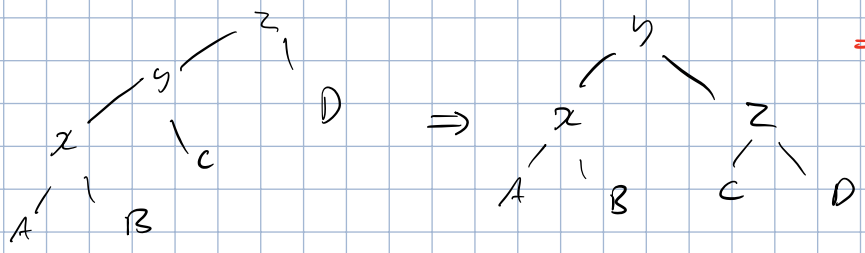
→ Recurse up + fix heights

```
setHeightSubtrees (v):
    v.height ← 1 + max (v.left.height, v.right.height)
```

Maximum subtree height

Restructuring: Rotation.

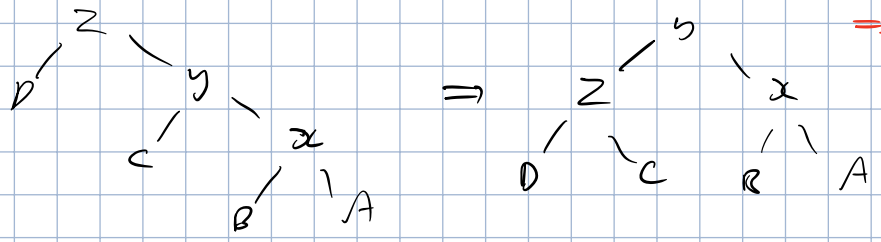
① Right rotation:



⇒ x causes imbalance.

Do this if node inserted in left st. of left s.t.

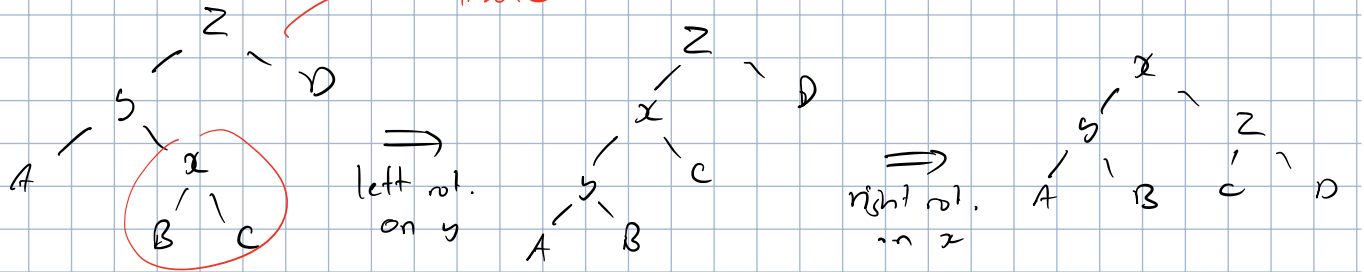
② Left rotation:



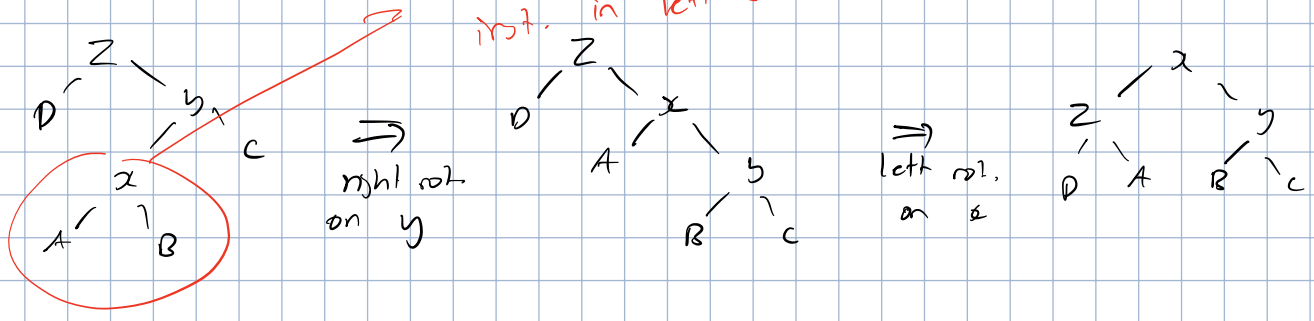
⇒ x causes imbalance

D. this if node ins. in right s.t. of right sub.

③ Double right:



③ Double left:



Deletion:

```
delete (k):
    z ← BST-delete(k)
    while (z not null)
        if z inbalanced:
            y ← taller child of z
            x ← taller child of y
            z ← restruct (z, y, x)
        set height (z)
        z ← z.parent
```

Might have to do multiple restructurings.

Analysis

Search: $\Theta(h) \Rightarrow \Theta(\log n)$

Insertn: $\Theta(h) \Rightarrow \Theta(\log n)$ ⇒ Restructure once

Deletion: $\Theta(h)$, restructuring can be called $\Theta(h)$ time ⇒ $\Theta(\log n)$ (expensive!)

Restructure: $\Theta(1)$ (swapping, but exp.)

PROOF

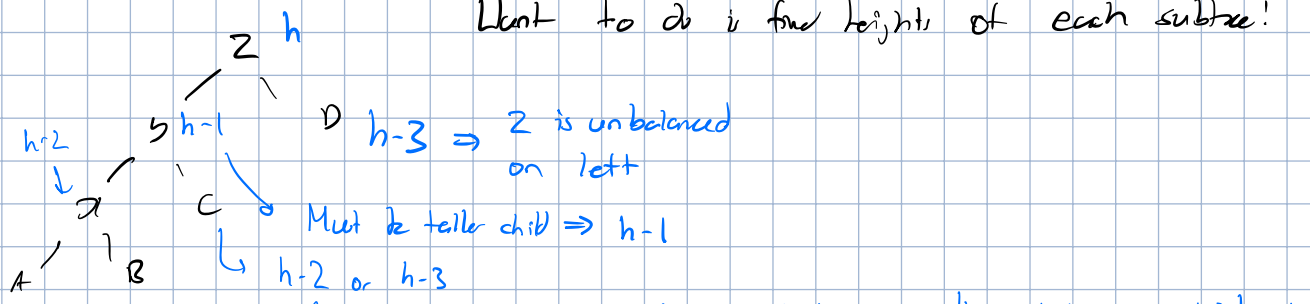
Claim: run restructure 1x on insertn:

1. Tree after restructure (T') is balanced
2. height (T') = height of tree before insertn!

Proof method: case Analysis on all restructure methods
↳ Analyze: before insertion, after insertion, after restructure

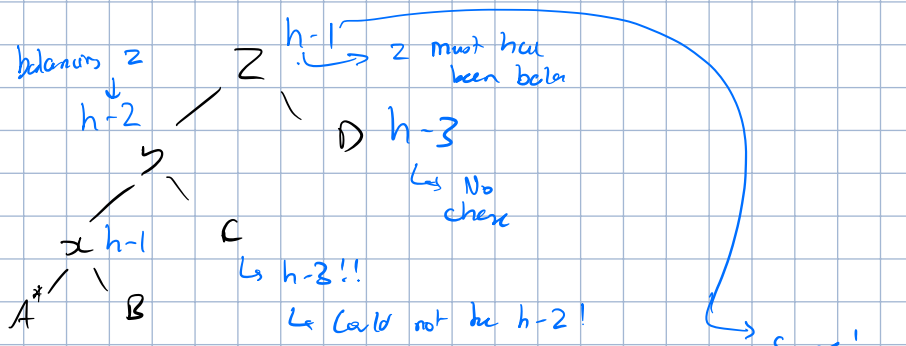
Right rotation case:

① After insertion:



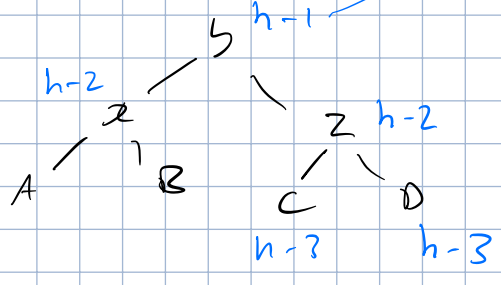
Want to do is find heights of each subtree!

② Before insertion:



Same!

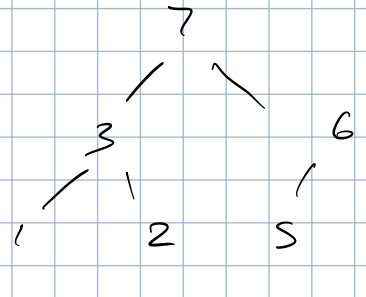
③ After restructure:



Continuation of Week 2:

Operations in Heap Tree

1. Insert
   1. Insert at end
   2. Fix up:
      Is child < parent?
      ↳ Swap
      Else:
      ↳ Done



just checking parents

⇓

$O(\log n)$

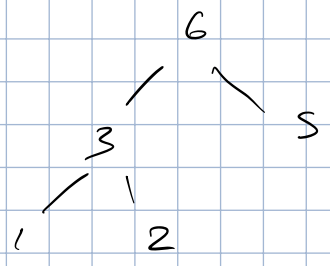
2. Deletion:

1. Replace top node w/ last node

2. Fix down

↳ Swaps parent w/ biggest child

↳ Do this until parent > children



PQ-Sort + Heap-Sort + Heapify

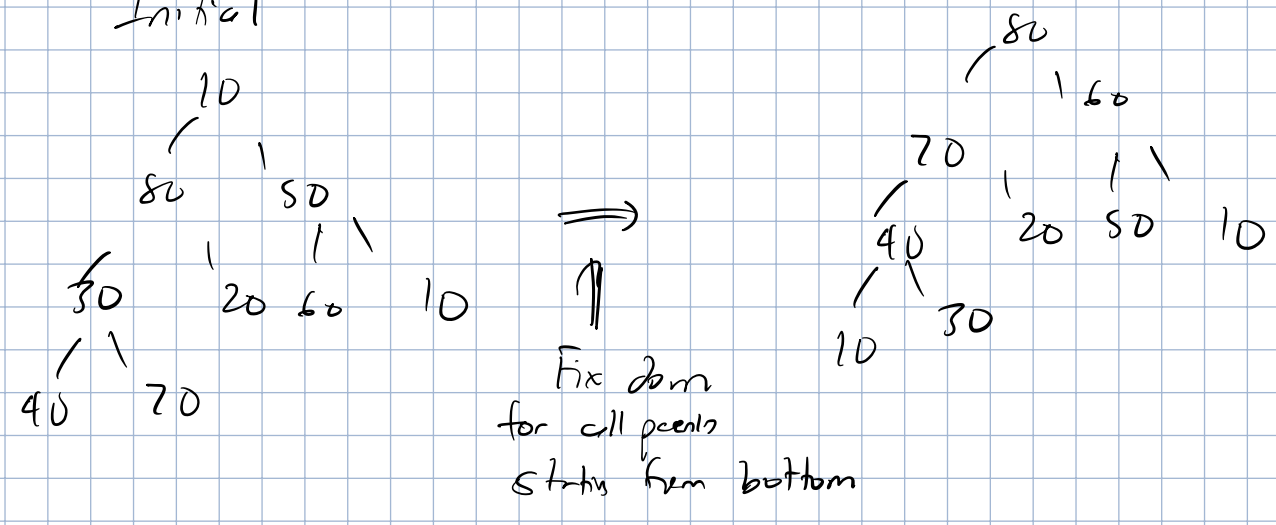
PQ-sort:

1. Put everything into heap (key = value = A[i])

2. Delete max n times ⇒ back into Array starting at back

Time: $\Theta(n \log n)$. Space: $\Theta(n)$

Heapify: takes array → heap array $\Theta(n)$!



Why $\Theta(n)$?

$$T(n) = \Theta(\text{worst case of steps})$$

$$= \Theta(0 \cdot 2^h + 1 \cdot 2^{h-1} + 2 \cdot 2^{h-2} + \dots + h 2^{h-h})$$

$0 \cdot 2^h$: # of leaves at bottom; $1 \cdot 2^{h-1}$: 2nd last

$$= \Theta\left(2^h \left(0 + \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \dots + \frac{h}{2^h}\right)\right) \qquad \leq 2$$

$$= \Theta(2^h) \qquad n = 2^{h+1} - 1$$

$$= \Theta(n)$$

Heap sort:

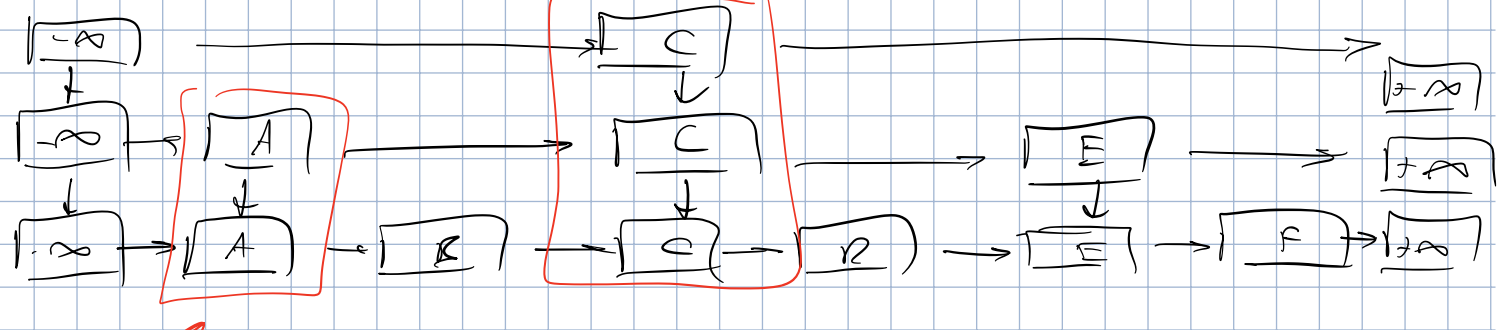
1. Heapify
2. "Delete" but now keeps getting small so larger elements at back of array

MODULE 5: SKIP LISTS

Realization

- AVL
- BST
- Ordered array / list.
  - ↳ Issue: bottleneck ⇒ search $\Theta(n)$

Skip: extension of LL ⇒ binary search.



Height 1 — Towers. — Height 2

Basic facts:

- Height: harmonic # of nodes at each height = $\sum_{i=1}^{h} \frac{n}{2i} \in \Theta(h)$
- Search: visiting 2 nodes for each level ⇒ # of comp = $2 \log n$
  ↳ $\Theta(\log n)$

} Not correct

Insertion, deletion is diff ⇒ randomization

Randomization: randomly choose which node go to next level

↳ S_0: 0^{th} level has all nodes

S_h: top level has only sentinel $(-\infty, +\infty)$

Height of tower = # of links in tower.

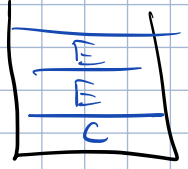
All nodes have following atomic ops:

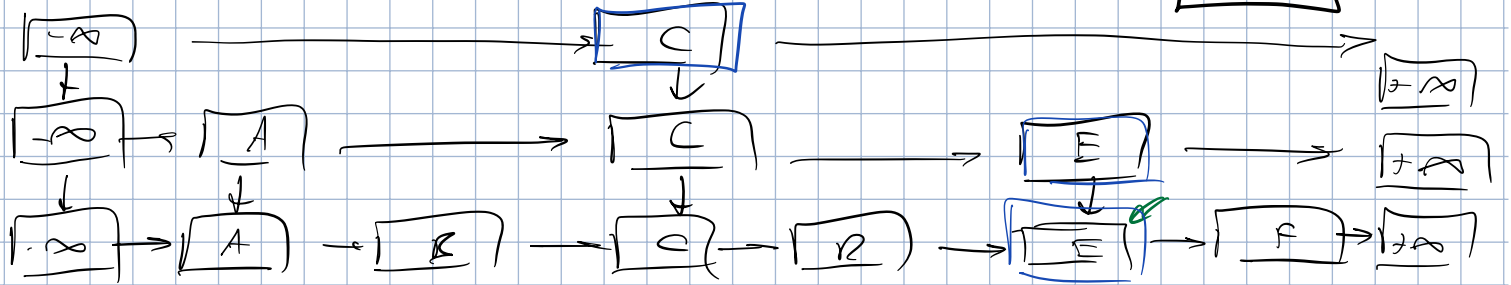
1. After (next node)
2. Below (node below)

SEARCH

1. Init at top most $-\infty$
2. If after(node) < target
   node ← node.after
3. Push node to stack P
4. node ← below(node)
5. Do this until node = NULL

} getPredecessors(target)





Predecessor Stack: if text node into the tower stack
P hold which node would have gone below

search (text):

P ← get Predecessors (text)

top ← P.top()

if top.after.key == text: return top.after.

return "not found"

Insertion:

Randomize w/ list of H, T

↳ Tower creation: # of times we get H before T = # of times node "promoted"

- H, H, H, T ⇒ insert in $S_0 \to S_1 \to S_2 \to S_3$ □

(no math with)

insert (k, v):

i ← # of heads in tails in randomization

h ← height of skip list so far. (increment for each time we do below() on sentinel)

If h ≤ i (tower height > existing)

Add additional sentinel layers.

P ← set Predecessors (k)

// Insert into S_0

p ← P.pop()

zBelow ← new Node with k, v

insertion (zBelow, p)

// Create tower:

while i > 0

p ← P.pop()

z ← new node with k, v

z.below ← zBelow

zBelow ← z

i ← i - 1

Deletion:

1. Get predecessor for key to delete
2. While P.top.after == key => delete p.after
   ↳ Checking entire tower
3. Remove any unnecessary sentinels.

delete (k):

P ← getPredecessor (k)

while P is not empty

P ← P.pop()

if p.after = k:

p.after = p.after.after

else:

break

p ← left sentinel of root

while p.below.after = ∞:

p.below = p.below.below

p.after = p.after.below.below

} Removing useless sentinel nodes.

Time analysis:

We find $E[h]$ = height of skip list det. all ops.

1. Define X_k: height of tower for key k
   - ↳ R.V.! Randomization determine X_k
   - ↳ Find $P(X_k \geq i)$
     - $P(X_k \geq 1) = P(1^{st} \text{ flip } \rightarrow \text{ head}) = 1/2$ (above S_0)
     - $P(X_k \geq 2) = P(1^{st} \text{ flip} \cap 2^{nd} \text{ flip}) = 1/2 \times 1/2 = 1/4$
     - $P(X_k \geq i) = (1/2)^i$ (i flips needed)
2. Height is unbounded by n ⇒ possible to have v. large height
   - ↳ Expected height is a better metric

3. $|S_i|$ = # of keys / level i sentinels
   - ↳ $|S_0| = n$
   - ↳ $|S_1| = ??$ ⇒ Use indicator variable!

4. Define indicator var to find expected level of key, in level.

$$I_{i,k} = \begin{cases} 0 & \text{if } k \text{ not in } S_i \\ 1 & \text{if } k \text{ in } S_i \end{cases} \Rightarrow P(k \text{ in } S_i) = \text{if height of } X_k \geq i = P(X_k \geq i)$$

∴ Using indicator variable trick:

$$E[S_i] = \sum_{k=1}^{n} E[I_k] \quad \Rightarrow \text{Go through all keys, + calculate expected!}$$

$$= \sum_{k=1}^{n} P(X_k \geq i)$$

$$= \sum_{k=1}^{n} (1/2)^i$$

$$= \boxed{\frac{n}{2^i}}$$

5. Use $|S_i|$ to find height:

Define another indicator variable:

$$I_i = \begin{cases} 0 & \text{if no keys in level } i \Rightarrow |S_i| = 0 \\ 1 & \text{if keys in level } i \Rightarrow |S_i| \geq 1 \end{cases}$$

$$\therefore \quad h = 1 + \sum_{i \geq 1} I_i$$

$$E[h] = 1 + \sum_{i \geq 1} E[I_i]$$

Bounds on $E[I_i]$ ⇒ Derive O from this:

1. $I_i \leq |S_i| \Rightarrow E[I_i] \leq E[|S_i|] \leq \frac{n}{2^i}$

2. $I_i \leq 1 \Rightarrow E[I_i] \leq 1$

Break the sum apart

$$E[h] = 1 + \sum_{i \geq 1}^{\log n} E[I_i] + \sum_{i = \log n + 1}^{\infty} E[I_i]$$

$$\leq 1 + \sum_{i \geq 1}^{\log n} 1 + \sum_{i = \log n + 1}^{\infty} \frac{n}{2^i}$$

$$\vdots$$

$$\leq 2 + \log n$$

$$\therefore \underline{E[h] \in O(\log n)}$$

Space:

Sentinels: $2h + 2$

$$\therefore E[\#\text{ of sentinels}] = 4 + 2\log n + 2$$
$$\in O(\log n)$$

Space for nodes:

$|S_i|$ ⇒ # of keys in level i

↳ $E[|S_i|] = n/2^i$

$$E\left[\sum_{i\geq 0} |S_i|\right] = \sum_{i\geq 0} E[|S_i|] = \sum_{i\geq 0} \frac{n}{2^i}$$

$$= n \sum_{i\geq 0} \frac{1}{2^i} \rightarrow \text{Converges to 2}$$

$$= 2n \in O(n)$$

$\therefore$ Total space: $O(n)$

Search, insert, delete:

Dominated by set Predecessors.

set Predecessor,

# of times drop down — # of times scan forward.

Drop-down analysis:

Can't drop more than height

# of drop downs $= O(h) = O(\log n)$

Scan-forward analysis:

Let u be the leftmost node in S_i (what we're at)

Let w be the node after it

↳ Height $(w) = i \Rightarrow$ No way height $(w) > i$ b/c would have compared it already!

$P(\text{tow } w = i \mid \text{tow of } w \geq i) = P(\text{getting T})$

$= \underline{1/2}$

$\therefore P(\text{scanning forward}) \leq 1/2$

$\therefore P(\text{scanning forward } \ell \text{ times}) \leq \left(\frac{1}{2}\right)^{\ell}$

$$E[\#\text{ of scan towers}] = \sum_{k \geq i} k\left(\tfrac{1}{2}\right)^k = 1$$

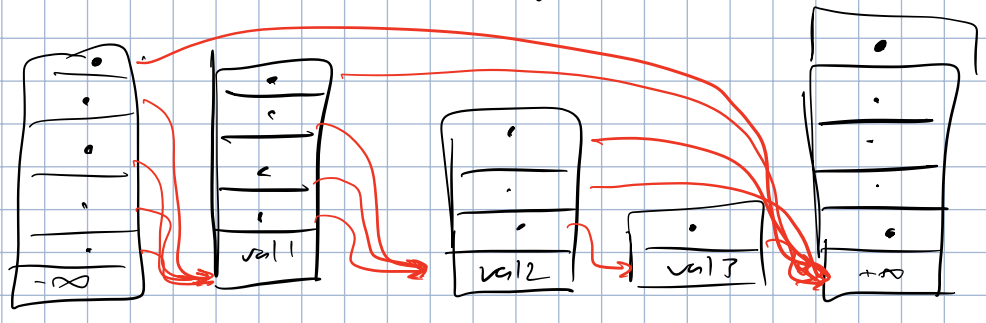
Scan fwd $\in O(1)$

$\therefore$ Get predecessor $= O(\log n)$

↳ Insertion, search, deletion $\Rightarrow \log n$

Optimization:

1. Make each tower an array:



Level = array index $\Rightarrow$ find necessary links to next tower

or re decrement in arr

2. Biased coin flips

$P(X_k \leq i) = \left(\tfrac{1}{2}\right)^i \Rightarrow$ Fair

↳ $P(X_k \leq i) = p^i \Rightarrow p < \tfrac{1}{2}$ (unfair)

Doing analysis again $\Rightarrow \log_{1/p} n < \log_2 n$

Reordering Items:

Unordered array/LL:

↳ Search: $\Theta(n)$

↳ Insertion: $\Theta(1)$

↳ Deletion: $\Theta(n)$

Bottlenecked.

One major optimization: put the most freq. accessed items at front.

Optimization

Access distr. known

Order keys from highest access prob. to lowest access prob.

Ex: //

key	A	B	C	D	E
freq.	2	8	1	10	5
prob.	2/26	8/26	1/26	10/26	5/26

1 possible order:

C A B D E

Expected # of comp:

$$E[C] = \frac{2}{26}\cdot 1 + 2\cdot\frac{2}{26} + 3\cdot\frac{8}{26} + 4\cdot\frac{10}{26} + 5\cdot\frac{5}{26}$$

$$\approx 3.261$$

Another: heuristic.

D B E A C

$$E[C] = 2.54$$

Access dist. not known.

Idea: temporal locality.

1. Move to front heuristic:

Access → move item to front

↳ Pf it reversed if correct (move + insert to back)

$T^{exp}(MTF) \leq 2T^{exp}(\text{known dist.})$

2. Transpose heuristic:

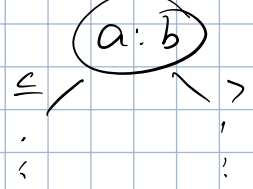
Access → move item up by 1

MODULE 6: SPECIAL KEY DICT

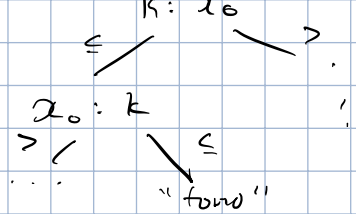
Lower Bound For search

Comparison-based dict $\Rightarrow \Omega(\log n)$

Proof:



$\Rightarrow$ How does this account for equality



Double check.

# of leafs = # of distinct inputs/outputs,
= $n+1$ ⇒ Any of n options to search in array or did not find any ($+1$).

$\therefore \quad 2^h \geq n+1$

$h \geq \log(n+1)$

$\therefore h \in \Omega(\log_2(n+1))$

Interpolation Search

Can we possibly speed up binary search?

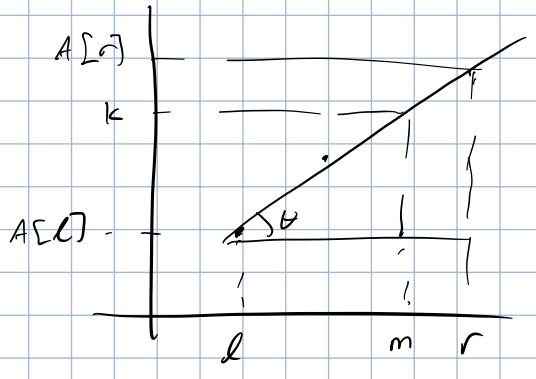
Recall binary search:

Middle point: $\left\lfloor \frac{l+r}{2} \right\rfloor = l + \left\lfloor \frac{1}{2}(r-l) \right\rfloor$ (Halved range)

Interpolation search:

$$m = l + \left\lfloor \frac{k - A[l]}{A[r] - A[l]} (r-l) \right\rfloor$$ (Weighted range)

Idea: we don't need to check middle. If we know that our array grows linearly ⇒ weight our side our cradle.



$\tan\theta = \tan\theta$

$\frac{A[r]-A[l]}{r-l} = \frac{k-A[l]}{m-l}$

$\therefore \quad m = l + (r-l)\cdot\left(\frac{k-A[l]}{A[r]-A[l]}\right)$

Analysis:

If keys are linear + uniform ⇒ $T^{avg}(n) = T^{avg}(\sqrt{n}) + \Theta(1) \in O(\log\log n)$

↳ Worst-case: $\Theta(n)$.

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

Assume n is pow(2) ⇒ $2 \to 2^2 \to 2^4 \to 2^8 \to 2^k \to 2^{2^i}$ ($\sqrt{n}$)

Go backwards: $T(2^{2^i}) = T(2^{2^{i-1}}) + c$

$$= T(2^{2^{i-2}}) + 2c$$

$$= T(2^{2^{i-3}}) + 3c$$

$$\vdots$$

$$= \underbrace{T(2^{2^{i-i}})}_{O(1)} + ic$$

$2^{2^i} = n \Rightarrow 2^i = \log n$

$i = \log \log n$

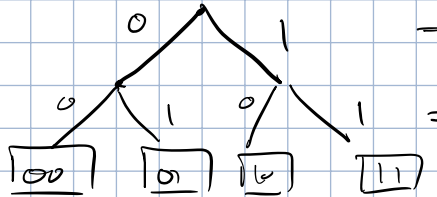
Pseudocode:

```
interpolation-search (A, k, n):
    l ← 0,  r ← n-1
    while (l ≤ r):
        if (A[k] < A[l] or A[k] > A[r]): "not found"        ← Bound check
        if (A[l] == A[r]) return "found at l"              ⇒ Avoid division by zero
        m ← l + (k - A[l]) / (A[r] - A[l]) (r - l)
        if A[m] < A[k]:
            l ← m + 1
        else if A[m] > A[k]:
            r ← m - 1
        else:
            found at m
```

Tries

Dictionary for strings

Bitstring tries: bitstrings (0, 1, 001, 110, ...)

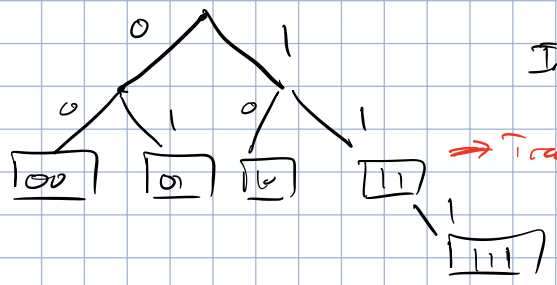


→ 1st che

⇒ 2nd che

→ All leaves are strings

Problem: prefixes



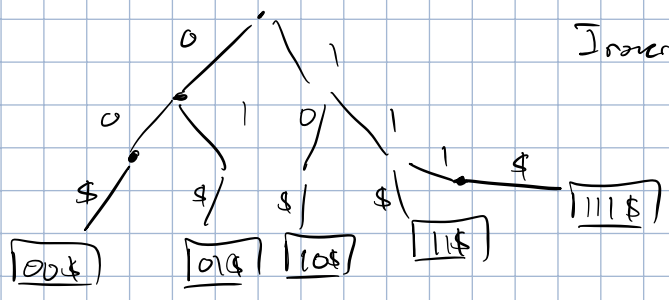
Insert 111

⇒ Tree has no way of knowing that this is a string!

Soln:

1. Enforce all strings of same length

2. More flexible: ensure none of strings can be a prefix ⇒ append end of word char ($)



Insert 111

Operations:

① Search:

trie node, digit, string searching

```
search (v, d, x)
    if v is a leaf: return v
    else:
        v' ← child of v labelled w x[d]
        if no v': return None ⇒ No child w/ digit
        else:
            return search (v', d+1, x)
```

next digit

② Insert:

Do search → last successful node → add more nodes until trie like

③ Deletion:

Do search → remove node & ancestors until ancestor has other child

Analysis:

- Time: $\Theta(w) \Rightarrow w = |string|$ (character comp.)
- Space: Worst case => each word has distinct path

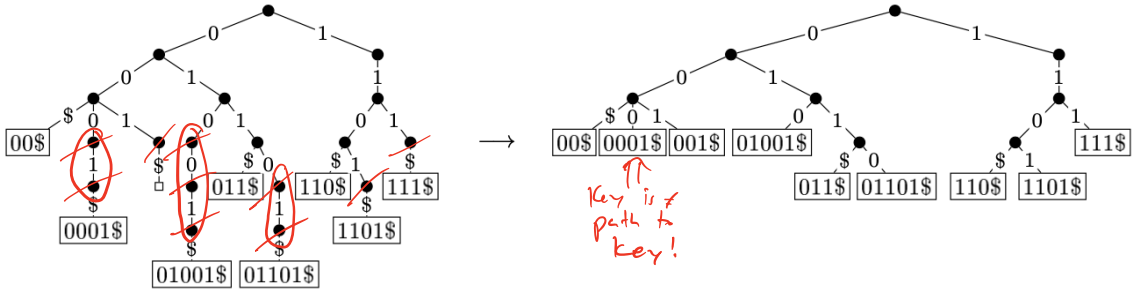
$$O\left(\sum_{\text{word } x} |x|\right) \Rightarrow$$ Usually words share paths.

Variations:

① Don't store keys ⇒ path uniquely identifies string.

② Allow prefixes: have an internal flag if node is word or not
↳ No need for $

③ Pruned trie: a node has a child ⇔ at least 2 descendants



How to store key

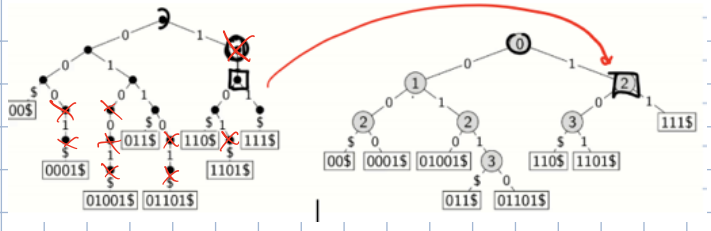
How to convert a trie ⇒ pruned trie

① For each node, check if it has ≥ 2 leaf descendants

② If so, move on

③ Else, delete.

④ Compressed trie: only store node w/ 2 children



Additional thing: each node has index ⇒ depth of node

Why? Skipping letters.

Compressed.

Trie w/ n keys $\Rightarrow$ # of internal node $\leq n-1$ $\Rightarrow$ Prove via induction.

# of leafs $= n$

# of nodes $\leq 2n-1$

Operations in compressed trie:

```
search (v ← root, x)
    if v is a leaf:
        return x == v.string
    d = v.index
    if d > len(x):
        return "not found"
    v' ← child of v w/ path going to it = x[d]
    if no child:
        return not found.
    search(v', x).
```

$O(|x|)$

Paths don't uniquely determine string $\Rightarrow$ have to compare key

Skipping nodes & length! We need to check if depth is too much.

delete: search $\rightarrow$ remove $\rightarrow$ compress path

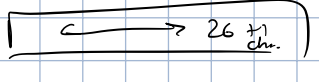
insert: search to maximal $\rightarrow$ uncompress $\rightarrow$ insert $\rightarrow$ recompress.

⑤ Multiway trie: over any alphabet Σ

Runtime: $O(|x| \cdot$ time to find child$)$

↳ Soln 1: Array of size $|\Sigma|+1$ $\Rightarrow$ search in char spot.

node: a



$O(1)$ for child search, but $O(|\Sigma|+1)$ space / node.

↳ Soln 2: children rarely have letter.

$O(|\Sigma|)$ to search child, $O(1)$ space.

→ Soln 3: Dictionary of children

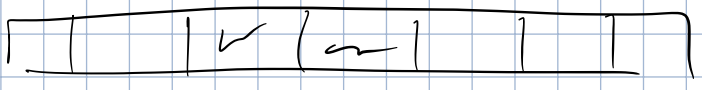
Hashmap, AVL, Skip ⇒ $O(1) \to O(\log n)$ time, $O(|\Sigma|)$

Module 7: Dictionaries via Hashing

Intro

$O(1)$ search:

- Direct addressing: know the range of keys



$O(1)$ time, $\Theta(M)$ space ⇒ M being the range.

Disadvantages:

1. Only works for ints
2. Space wasted ⇒ $M \gg n$

- Hashing: keys ⟶ $\{0, \ldots, M-1\}$ for direct addressing

Need hash function

Disadvantage: collision (map to same key)

Dictionary that uses hashing ⇒ hash table.

Probability of collision:

n independent values uniformly

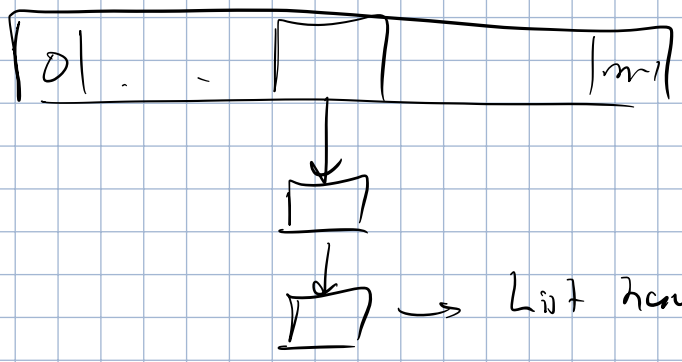
Prob. of no collision: $\dfrac{M(M-1)(M-2)\cdots(M-(n-1))}{M^n}$

Prob. of collision = $1 - \dfrac{M(M-1)(M-2)\cdots(M-(n-1))}{M^n}$

Can't compute.

What we can do ⇒ Fix probability α of collision & choose M accordingly

Separate Chaining:



→ List have some keys.

Operations.

Search (k): Find k in $T[h(k)]$

Insert (k, v): insert k, v in $T[h(k)]$

Deletion (k, v): search → delete

Analysis:

Insertion: $O(1)$

- Hashing: $O(1)$
- Insertion in LL: front → $O(1)$

Search:

- Worst: $O(1 +$ size of bucket $T[h(k)])$
- Load factor:

$$\alpha = \frac{n}{M} \Rightarrow \text{Avg. bucket size.}$$

- Avg. case $\Sigma O(1+\alpha)$
- Uniform Hashing Assumption:

$\forall$ key k, $\forall j \in \{0 \ldots M-1\} \Rightarrow h(k)=j$ has prob. $1/M$

- Analysis can be tight:

Claim: Each key will collide w $\frac{n-1}{M}$ other keys $\Rightarrow$ avg. case cost $\in O(1+\alpha)$.

$\frac{n-1}{M}$ bucket size!

Proof:

$\forall i, i' \in \{1, \ldots, n\}, i \neq i'$

$$P(h(k_i) = h(k_{i'})) = \sum_{j=0}^{M-1} P(h(k_i) = h(k_{i'}) = j)$$

$$= \sum_{j=0}^{M-1} \frac{1}{M} \cdot \frac{1}{M}$$

$$= \frac{1}{M}$$

$\therefore$ Prob. of collision $= \frac{1}{M}$

Let $X_{i,i'} = \begin{cases} 0 & \text{if } h(k_i) \neq h(k_{i'}) \\ 1 & \text{if } h(k_i) = h(k_{i'}) \end{cases}$

Expected # of collisions: $\sum_{i \neq i'} E[X_{i,i'}]$

$$= \sum_{i \neq i'} P(\text{collision})$$

$$= \sum_{i \neq i'} \frac{1}{M}$$

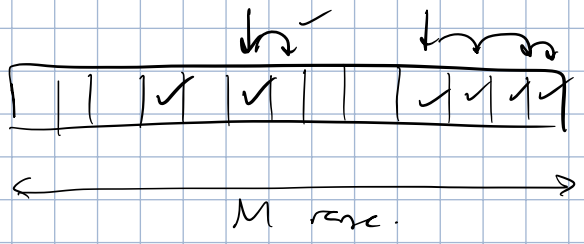
$$= \frac{n-1}{M}$$

- Key: keep α small
  - Say α too big $\Rightarrow$ M is bigger $\Rightarrow$ rehash into bigger table
    - Really, ignore
  - Say α too small $\Rightarrow$ M is smaller $\Rightarrow$ rehash into smaller table
  - If $1/4 < \alpha < 2 \Rightarrow M \in \Theta(n)$

$\therefore \alpha \in O(1) \Rightarrow$ Cost of chaining is $O(1)$

+ Space: $O(n)$

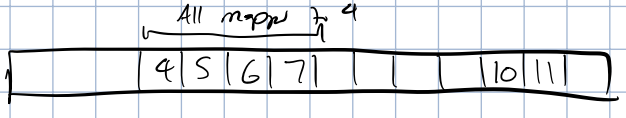
Probe Sequences



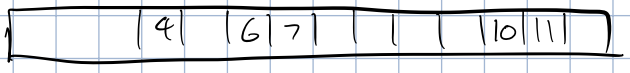
Operations:

Search / insert: Map to $T[h(k)]$ → move forward until key found.

Deletion: do search → mark spot deleted w/ flag.



1. Delete 5 (empty)



2. Search for 6
   - ↳ Map to 4
   - ↳ Stop @ empty spot!

How to continually move to next spot

$$h(k, i) = (h(k) + i) \bmod M$$

Problem: clustering issues (all keys map to similar hash → clump of keys to search through.

1. Quadratic hashing:

$$h(k, i) = (h(k) + c_1 i + c_2 i^2) \bmod M$$

Still have clustering issue, but diff keys have diff paths, so not as bad.

Make sure c_1, c_2 are chosen well

2. Double hashing:

$$h(k, i) = (h_0(k) + i\, h_1(k)) \bmod M$$

Key: $h_1(k)$ is non-zero (o.w. collisions all the time)
& $h_1(k) \perp h_0(k)$

How to make $h_0(k) \perp h_1(k)$?

↳ Multiplicative method.

$0 \to M$

$$h_1(k) = 1 + \lfloor (M-1)(kA - \lfloor kA \rfloor) \rfloor$$

Non-zero. Fractional: $[0, 1)$

Typically $A = \varphi$

Make M prime: reach all spots in hash table & h_1 is rel. prime w/ h_0

Ex// Insert 194. $M = 11$. Double hashing.

① Make indep hash functions via multiplicative meth.

$$h_0(k) = k \bmod 11$$

$$h_1(k) = 1 + \lfloor 10(k\varphi - \lfloor k\varphi \rfloor) \rfloor$$

$$\therefore h(k,i) = \left(h_0(k) + i\, h_1(k)\right) \bmod 11$$

② Probe sequence:

$$h(k,0) = 7$$
$$h(k,1) = 5$$
$$h(k,2) = 3$$
$$h(k,3) = 1$$

$7 + h_1(194)$

$1 + \lfloor 10^*(194 \cdot \varphi - \lfloor k\varphi \rfloor) \rfloor$

$= 4$

$7 + 18 = 25$

$7 + 27 = 34$

③ Go down probe sequence & find first spot available.

Cuckoo Hashing

2 tables: $T_0\ (h_0)$, $T_1\ (h_1)$

A key is in either T_0 or T_1, cannot be in both.

Operations:

Search: $T_0[h_0(k)]$ or $T_1[h_1(k)]$

Delete: search → delete.

Insertion:

① Map $h_0(k)$. If not occupied → done

② If occupied, replace k' w/ k and hash $T_1[h_1(k')]$

③ If not occupied, done

④ If occupied, replace k'' w/ k' and hash $T_0[h_0(k'')]$

- Operation: $O(1)$ ~~time~~ if α is small.
- Pseudo code:

```
insert (k, v):
    i ← 0
    do at most 2n times:        ← If inserted 2n+1 times, it will go forever
        if T_i[h_i(k)] is empty:
            T_i[h_i(k)] = (k, v)
            return "success"
        swap ((k, v), T_i[h_i(k)])
        i ← 1 − i
    return "failure"
```

Ex:// $M = 11$, $h_0(k) = k \bmod 11$, $h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

	T_0	T_1
0	44	
1		
2		
3		
4	26	
5		59
6		
7	81	95
8		
9	92	
10		

① insert (95)

1. T_0 hashs

   $h_0(95) = 7$

2. Kick out + hash to T_1

   $h_1(81) = 5$

② insert (26)

1. T_0

   $h_0(26) = 4$

2. T_1
$h_1(59) = 5$

3. T_0
$h_0(51) = 7$

4. T_1
$h_1(95) = 7$

(5) search (59)
$h_0(59) = 4 \rightarrow ? \rightarrow \times$
$h_1(59) = 5 \rightarrow ? \rightarrow \checkmark$

Notes:

1. $|T_1| \neq |T_0| \Rightarrow$ can be diff., can be same

2. Load factor:

$$\alpha = \frac{n}{|T_1| + |T_0|}$$

If $\alpha < 1/2 \Rightarrow O(1)$ insertion.

3. Order of T_0/T_1 doesn't matter.

Programmer does best choice.

All operations w/ hashing take avg. time of $O(1)$. Worst case: $O(n)$

Hash Function Strategies

Prime objective: we want uniform hashing assumption

↳ We want 2 things:

① Hash function ⊥ data

② Depend on all key pts

Want to hash pair (x, y)
Combo (x, y) is unique, not just x or just y
$\therefore h_0((x, y)) = x + y \bmod M$

Int. keys

Modular:
$h(k) = k \bmod M$
↑
M is prime

Multiplication
$h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$ for some $0 \leq A < 1$

No hash func. is perfect ⇒ exists some seq. of keys that do poorly

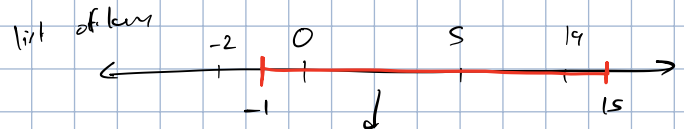
Multidimensional keys: flatten to int.

- Ex: // APPLE ⇒ $65R^4 + 80R^3 + 80R^2 + 76R + 68$ (ASCII)
  ↳ mod M

MODULE 8: RANGE SEARCHING IN DICTS

Range Searching:

- 1D case:



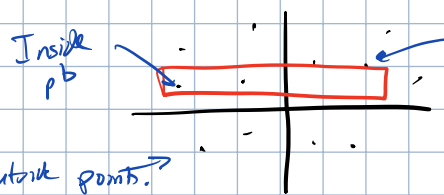
Range of $-1 \to 15$ ⇒ All values w/ keys in range.

Note: We have multiple items to search, insert, delete
↳ Analysis based on output size: s

1D implementations:

- Unsorted array/hash table/list: check each item in D.S.
  - $\Omega(n)$
- Sorted array/list: binary search for endpoints ⇒ return output
  - $O(\log n + s)$
- BST: almost same ⇒ $O(h+s)$

- Multidimensional searches:
  - Aspects: dimensions
    - ≥ 2 dimension keys. Ex: // geographical coordinates.
  - Multiple aspects ⇒ search across all aspects
    ↳ Define query polygon (2D case, do a rectangle).

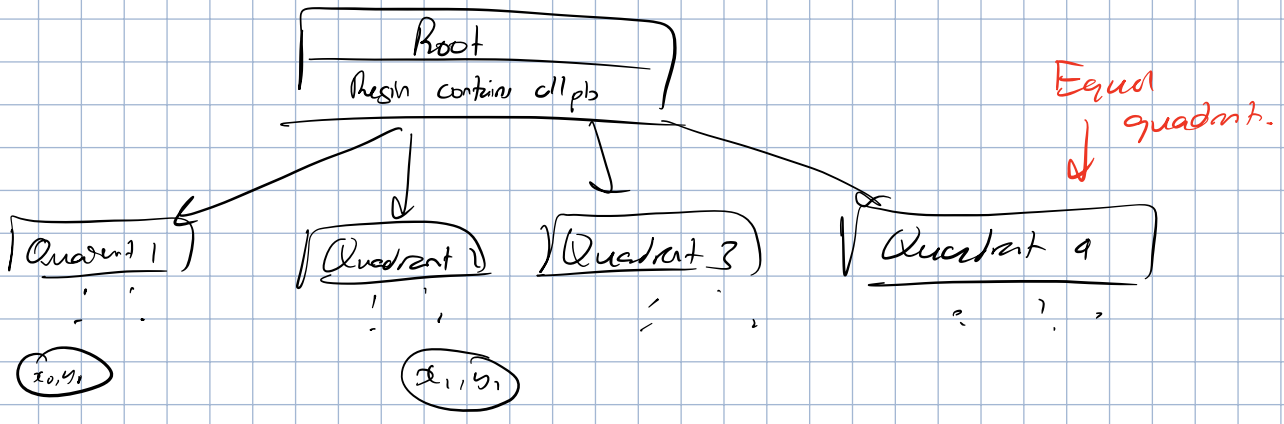


Implementation:

1. Try to map aspect → 1 dimension: very complicated
2. Use dictionary per aspect → get points → union: inefficient
3. Other data structures.

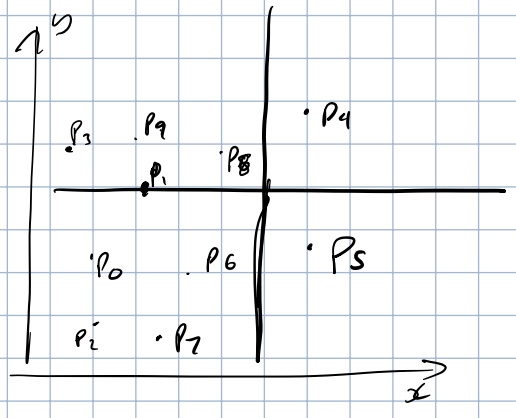
Quadtree

Defn: quadtree of n points $\{(x_0, y_0), (x_1, y_1) \dots\}$



- Recursively define struct:
  1. If quadrant has 1/0 pts ⇒ point or leaf
  2. Split into equal size quadrants if not
     - ↳ Split point set to match split in quadrants.
     - ↳ Convention: points on line → go to top/right quadrant

Ex: //



Draw quadrant tree:

① Root ⇒ all points

Find out max and min of x, y

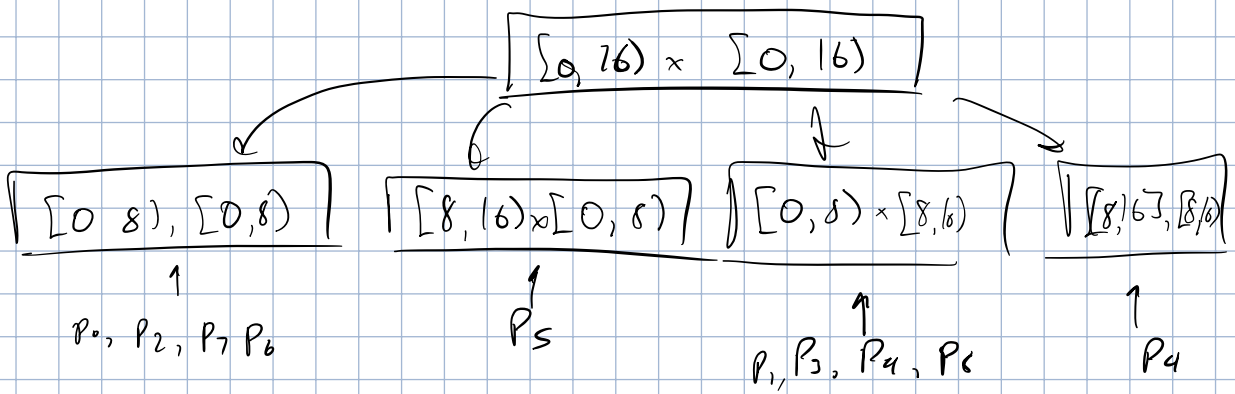
$[0, 16) \times [0, 16)$

② Subdivide

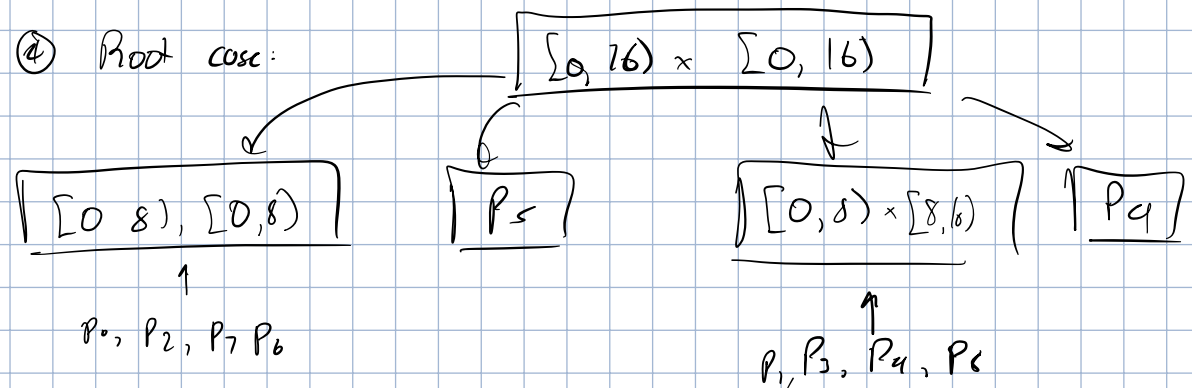
$[0, 16) \times [0, 16)$

$[0, 8), [0, 8)$ | $[8, 16) \times [0, 8)$ | $[0, 8) \times [8, 16)$ | $[8, 16), [8, 16)$

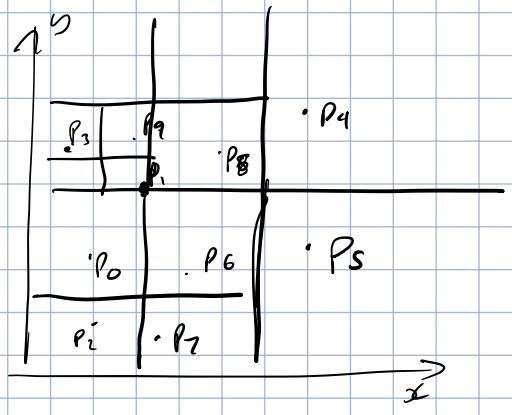
③ Subdivide points.



④ Root case:



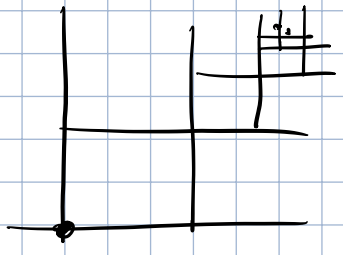
⑤ Recurse until pts are leaves.



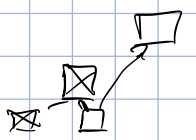
Operations on Quadtrees:

1. Single point search: BST search w/ x & y comparisons.

2. Insert 1 point: search → leaf → split leaf in 4 quadrant until leaves have 1 point.



3 Delete: search → delete → recursively delete parent if only 1 child.



4. Range Search:

```
rangeSearch (r, A):
    R ← region of root
    if R ⊆ A:                    R completely within A: inside node
        return everything below r
    if R ∩ A = ∅:                R is not overlapping w/ A: outside node
        return
    if r is a leaf:
        p ← point of r
        if p ∈ A:
            return p
    else:
        return rangeSearch (all children, A)
```

} Bounding nodes.

Analysis:

Note: Height of quadtree ∝ # of points!

↳ Distribution of points determines height

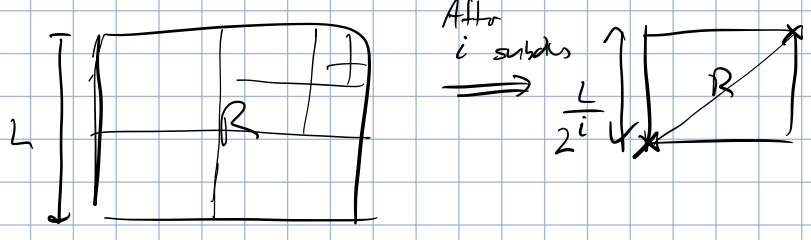
Distribution measure: spread factor

$$\beta(s) = \frac{\text{sidelength of } R}{\text{min. distance of pts in } s}$$

$$\underline{h \in O(\log(\beta(s)))}$$

↳ Proof that $h \in O(\log(\beta(s)))$:

Region R:



Thus:

Max distance between pts: $\sqrt{2} \cdot \frac{L}{2^i}$

$$d_{min} \leq \sqrt{2} \cdot \frac{L}{2^i}$$

$$2^i \le \sqrt{2}\,\frac{L}{d_{min}}$$

$$2^i \le \sqrt{2}\,\beta(S)$$

# of subdivs = height of tree $\Rightarrow$ $i \le \log_2(\sqrt{2}\cdot\beta(S))$

$\therefore\ h \in O(\log(\beta(S)))$

Proof technique:
↳ Try to subdivide R into side length $\Rightarrow$ spread factor.

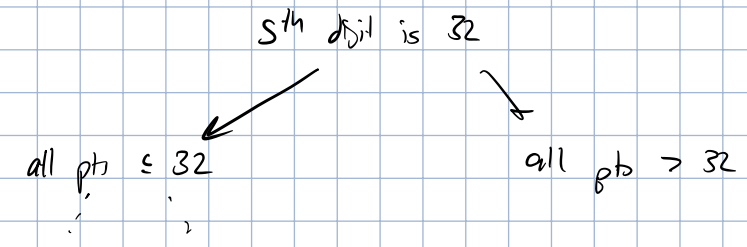
Operation analysis:

Time to build quadtree: $\Theta(nh)$

Range search: $\Theta(nh)$ $\Rightarrow$ search through all nodes

Multidimensional Quadtrees:

1D: put pts in base 2 $\Rightarrow$ use trie logic to subdivide



3D: octree

Variations:

1. Multiple pts in 1 leaf
2. Stop splitting prematurely

kd Trees

Idea: split points into 2 $\Rightarrow$ equal # of points / not equal size

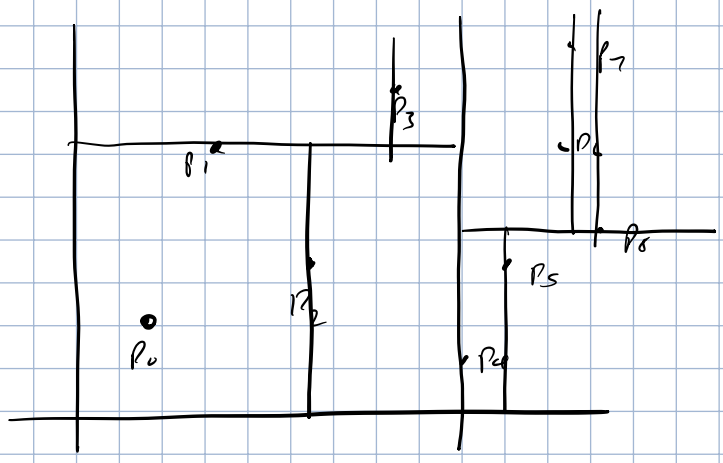
Construction:

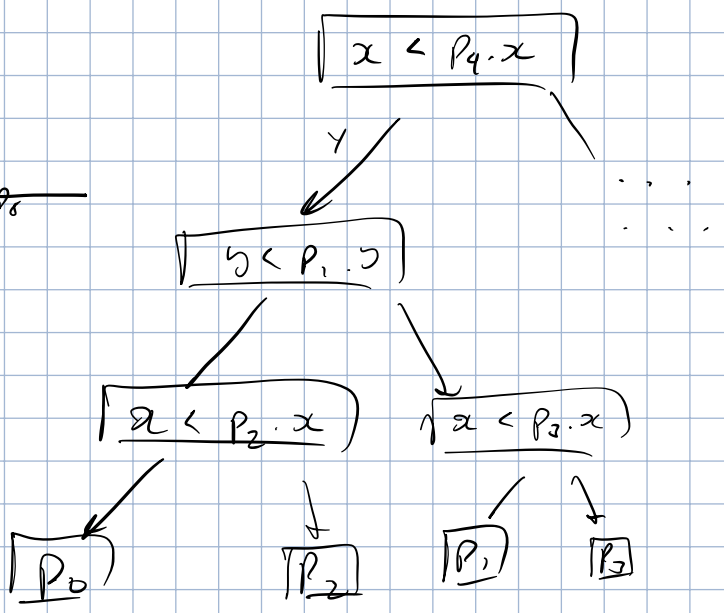
① If if $|S| = 1$ $\Rightarrow$ leaf

② Else: Do quick select to find median x-coordinate
↳ Partition into $S_{x<x}$, $S_{x\ge x}$

③ Recursively create kd tree on left/right subtrees but alternate y coordinates.

Ex://





Running Time Analysis:

Finding median $\Rightarrow \Theta(n)$ (quick select)

Recursion: $T^{exp} = 2T(n/2) + O(n) \approx^{exp} T(n) \in O(n \log n)$

(Median: $O(n)$; Recursive construction: $2T(n/2)$)

Height: $O(\log n)$

Operations:

1. Search for point: BST $\Rightarrow O(\log n)$
2. Insertion: BST insertion
3. Deletion: BST deletion

} will impact recon $\Rightarrow$ unbalanced, so $h \notin O(\log n)$

$\Downarrow$

Solution:

① Rebuild tree

② Don't care $\Rightarrow$ range search take longer.

Bottom line: not good for insertion/deletion.

4. Range search: exact same also as quad tree, use 2 children.

↳ Time complexity = # of points retrieved * # of recursion.

$\Downarrow$ $O(s)$

$\Downarrow$ $Q(n) = 2Q(\frac{n}{4}) + O(1)$

↳ $O(\sqrt{n})$

$\therefore O(s + \sqrt{n})$

Multidimensional:

Alternate all d dimensions when doing splits

Space complexity: $O(n)$, Construction time: $O(n \log n)$

Range search: $O(s + n^{1-\frac{1}{d}})$

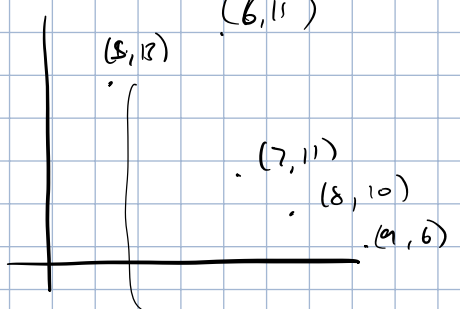
Range Trees

Range Tree

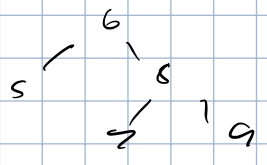
Primary Tree
⇓
BST w/ all pts w/ x coordinate as key.

Associate Trees.
⇓ per node in primary tree
BST of all descendant nodes of primary node including itself.
key = y coordinate.

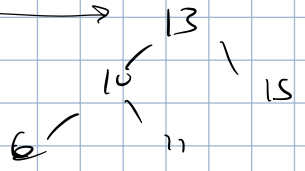
Ex:



Primary Tree:



Associate tree at 6:



Space complexity:

Primary tree: $O(n)$

Associate tree: $T(v)$ will have $O(|P(v)|)$

Size of all associate trees: $\sum_v |P(v)| = \sum_{v,w} \delta_{v,w} \Rightarrow \delta_{v,w} = \begin{cases} 1 & \text{if } w \in P(v) \\ 0 & \text{if } w \notin P(v) \end{cases}$

(v is ancestor of w; v is not ancestor of w)

$= \sum_w \left[\sum_v \delta_{v,w} \right] \Rightarrow$ # of ancestors to node w (fixing w ⇒ going through all nodes!!)

Think: # of ancestors of node? Path to root! $O(\log n)$

# of points = n

Size of all associate trees: $O(n \log n)$

Operations:

1. Search for single coordinate: BST using x
   - if point: done
   - if not: do a search on associate tree via B.S.

2. Insert point: do search for x-coordinate ⇒ insert ⇒ walk to root + insert in each associate tree.

3. Delete point: "

Problem: BST gets unbalanced after insertion / deletion.
- Soln #1: AVL, but really confusing (rotations)
- Soln #2: leave it alone, rebuild occasionally

4. Range search:

1D Range Search via BST:

a) Code-first way:

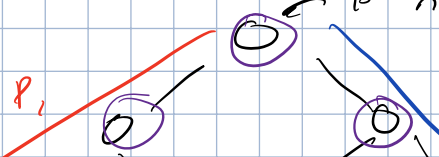
left bound ↓ , ↙ right bound

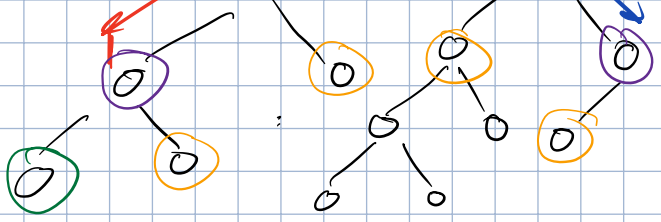
```
RangeSearch(r ← root, x1, x2):
    if r not a node:
        return {}                    ⇒ No descendants

    if x1 ≤ r.val ≤ x2:
        L ← RangeSearch(r.left, x1, x2)
        R ← RangeSearch(r.right, x1, x2)
        return L ∪ {r.val} ∪ R       ⇒ Inorder traversal ⇒ in a BST, ordered set of points!

    elif r.val < x1:                 (Current node is small ⇒ go right. Opposite.)
        return RangeSearch(r.right, x1, x2)

    elif r.val > x2:
        return RangeSearch(r.left, x1, x2)
```

b) Analysis-friendly way:

Try to create paths P_1 (goes to smallest value in range) and P_2 (largest value in range).

1st node that is in between $[x_1, x_2]$





Goal:

- Boundary nodes: nodes on boundary
- Outside nodes: nodes that are left of P_1 / right of P_2
- Inside nodes: nodes that are right of P_1 & left of P_2
  - ↳ Just return all descendants!

Analysis:

Time to find paths: assuming balanced $= O(\log n)$

↳ # of boundary nodes $= O(\log n)$

- At very least have to check all boundary nodes $\Rightarrow \log n$ factor.
- # of inside nodes: children of $\log n$ boundary nodes, so there are $\log n$ inside nodes that we check in $O(1)$

↳ Returning descendants $= O(s)$

$T(n)$ = time to get boundary nodes + time to get inside nodes + time to get descendants

$= c_1 \log n + c_2 \log n + s$

$= O(\log n + s)$

2D Range Search:

Also:

1. Do a 1D range search on primary tree $([x_1, x_2])$
   - Check all boundary nodes
   - Get inside nodes $\Rightarrow$ range search $([y_1, y_2])$
     - ↳ Optimize: use top most node b/c already has descendants.

Analysis: $O(\log^2 n + s)$

Summary of analysis: let d be dimensions.

1. Space: $O(n(\log n)^{d-1})$
2. Construct tree: $O(n(\log n)^d)$
3. Rege search: $O(s + (\log n)^d)$

} Not great for higher dimensions.

Rege Searching

Quadtree
+ Simple
− Don't work if skewed
− Space waste

KD trees
+ Linear space
+ Great as dimensions ↑
+ Search time: $O(\sqrt{n} + s)$
− Insertion / deletion

Rege Trees
+ $O((\log^2 n + s)$ rege search
− Waste space.
− Insertion & deletion

MODULE 9: STRING MATCHING

Introduction:

Goal: search for some pattern $P[0 \ldots m-1]$ in text $T[0 \ldots n-1]$
↳ Find 1st occurence of pattern in T (index where pattern starts).

Terminology:
- Substring: $T[i \ldots j]$
  - Length: $j - i + 1$
- Prefix: substring with $T[0 \ldots j]$
- Suffix: substring with $T[i \ldots n-1]$

Also Terminology:
- Guess: posn i where P might start at $T[i]$
  ↳ Valid guesses are in range $0 \le i \le n-m$ ↳ Must fit m characters!

$T[n-m+1 : n-1]$
↳ $n-1-((n-m+1)+1$
↳ $n-1-n+m-1+1$
↳ $m-1$ (cannot fit P)

- Check: see if $T[i+j] = P[j]$ for guess i, $0 \le j \le m$

↳ Need to perform m checks.

Diagrams:

T	a	b	b	a	. .
Itr.					
0	P[0]	P[1]	P[2]		
1		P[0]	P[1]		
2			P[0]	.	
3				P[1]	. .
⋮					

Brute force: check every substring of n in text T

- Time: $\Theta(nm)$
- Worst input: $T = a \ldots aa$, $P = a \ldots ab$.
  ↳ How to guess all $n-m+1$ occurrences, up to m checks

Karp Rabin Algorithm

Can we use hashes to make this quicker?

Naive version of hashing KR:

Ex: // $P = 5\,9\,2\,6\,5$ $T = 3\,1\,4\,1\,5\,9\,2\,6\,5\,3\,5$

Diagram:

3 1 4 1 5 9 2 6 5 3 5

h(31415)=

Can hash values equal the hash value of pattern?

Break when $h(P) = h(\text{substring})$

↳ Collisions! ⇒ still check if $P =$ substring

Analysis:

1. Construct pattern hash $\Rightarrow O(1)$
2. Construct hash value for each possible substring: $O(mn)$
   - Build string & hash
   - Repeat for all possible substrings.

Worse than brute force:

We have to check all characters when doing hash, no early exit if mismatch

Improvement: compute hash value from prev. in $O(1)$ time. (Rolling hash)

Ex: $S_1 = 41592$ $\quad S_2 = 15926$

$41592 \xrightarrow{-4\times10^5} 1592 \xrightarrow{\times 10} 15920 \xrightarrow{+6} 15926$

$h(15926) = h((h(41592) - h(4 \cdot 10^5)) \cdot 10 + 6)$

Formula:

$$h_{S_2} = h((h_{S_1} - T[i] \cdot s) \cdot 10 + T[i+m])$$

$s = h(10^{m-1})$

$T[i]$: First character of S_1

$T[i+m]$: Next character

Pseudocode:

```
KRA (T, P):
    M ← prime #
    hp ← h(P[0...m-1])
    hT ← h(T[0...m-1])  ⇒ O(m) time
    s ← 10^(m-1) mod M
    for i: 0 ⟶ n-1:
        if hT = hp:                    O(m)
            if T[i...i+m-1] = P:
                return i
        if i < n-m
            hT = formula to precompute.
```

Analysis:

Decent: practically fast

Worst case: $\Theta(mn)$

↳ Lots of collisions.

Knuth-Morris-Pratt Algo

Terminology:

- i : posn. in T that we are comparing
- j : " P "

Basic idea:

if $T[i] \neq P[j]$ and $j == 0$:
    $i += 1$

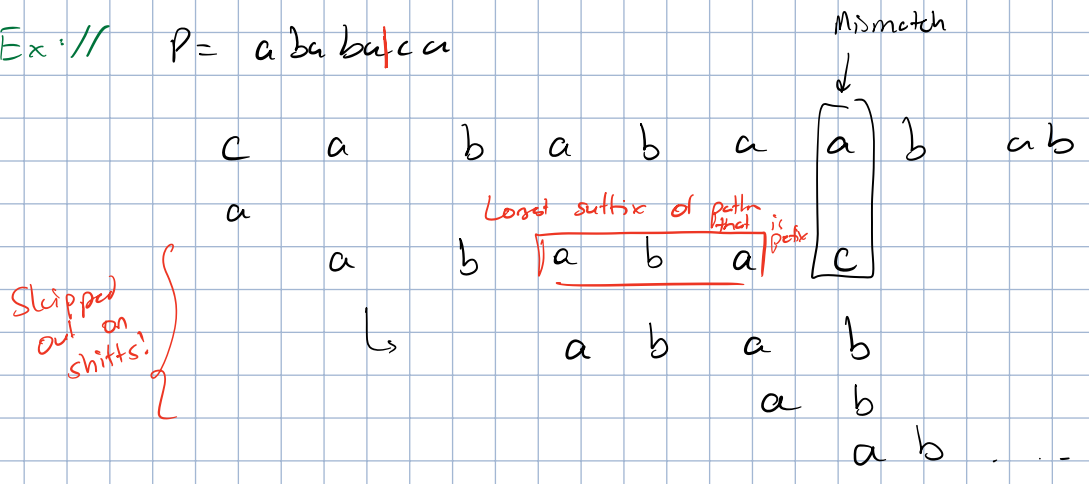
} Special case of j at beginning. Continue checking P, no need to increment i

elif $T[i] == P[j]$:
    $i += 1$
    $j += 1$

} Move to next char.

else ($T[i] \neq P[j]$ or $j > 0$):
    Use P to find shift to move to

Ex: // P = ababacca



Idea: use the longest suffix of P[1..j-1] s.t. P[1..j-1] is a prefix to shift ⇒ update j accordingly (j = len(P[1..j-1]))

Failure array: longest valid suffix of P[1..j] at F[j]

↳ If failure at j, check F[j-1]

Ex: // P = abababca

	0	1	2	3	4	5	6
F =	0	0	1	2	3	0	1

P[1..1] = b — Not a prefix

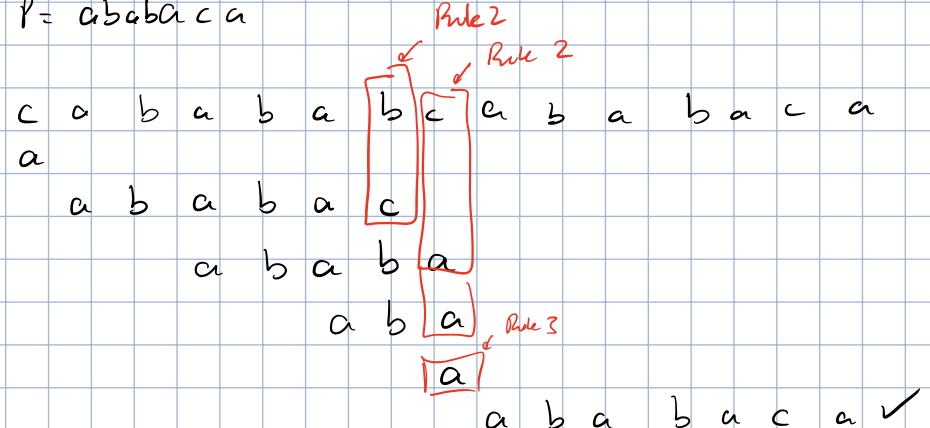
P[1..2] = ba — a is also a prefix

P[1..3] = bab — a is prefix

Naive: $O(m^3)$ to construct F

Final KMP also:

Ex:// P = ababaca



F = [0, 0, 1, 2, 3, 0, 1, 3]

Rule 1: match $(\substack{i+=1 \\ j+=1})$

Rule 2: mismatch at $j > 0$
↳ $j = F[j-1]$

Rule 3: mismatch at $j = 0$
$i += 1$

Analysis:

Time = # of horizontal shifts (i ++) + vertical shifts (j ↓)

$O(n)$: n characters limit

$O(n)$: n iterations max.

$\in O(n)$

Failure array computation:

Idea: Run KMP on pattern against itself

Builds up failure array ⇒ use failure array during building (DP).

Notation: ℓ = position in pattern that we are checking.

Ex:// Compute failure array of ababaca

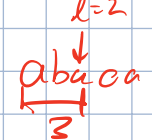
① Initialize F, $F[0] = 0$

② Initialize $j = 1$ ⇒ no need to parse P[0].
$\ell = 0$ ⇒ matching w/ 1st char in pattn.

③ Cases:

a) $P[j] = P[\ell]$: ⇒ From 0 → ℓ, we have a suffix that matches.

1. Increment ℓ
ℓ += 1 ⇒ Length of the suffix



2. Update failure array

$F[j] = \ell$

3. Continue to next text:

$j += 1$

b) If not match & $\ell > 0$ (not at beginning)

$\ell = F[\ell - 1]$ ⇒ Precomputed already

c) If not match & $\ell == 0$

$F[j] = 0$ ⇒ No suffix to match

$j += 1$

This runs in $\Theta(m)$!

Overall: KMP runs in $\Theta(m+n)$ worst case.

Boyer-Moore Algorithm

Fastest algo for string matching.

Idea: reverse search of pattern

Algo:

1. Look @ m^{th} character in pattern & $T[i+m-1]$
2. If match ⇒ look @ $i-1$ char... until 0
3. If no match ⇒ bad character heuristic.

   If text char not in pattern → move $m-1$ steps ahead (check new substr's).

   If text char in pattern ⇒ shift to last occurrence of shift character in pattern. Check at end of pattern.

Ex:// P = aaron T = acranapple

	a	c	r	a	n	a	p	p	l	e
0				o	n					
1.			a	a	r	o	n			
						a	a	r		

Derivation:

1. Construct last character occurrence array:

$LC(c)$ = last index of 'c' in P ($c \notin P \Rightarrow LC(c) = -1$)

Iterate in string $\Rightarrow$ put char index in array when iterating.

$\Theta(m)$

2. If mismatch at i in text and j in pattn.

Case #1: $LC(c) > 0$ (not -1) & $L[c] < j^{old}$

$j = m - 1 \Rightarrow$ starting @ back again

$i = i + m - 1 - LC(c) \Rightarrow$ text pointer to start comp.

Case #2: $LC(c) > 0$ & $L[c] > j^{old}$

↳ Already passed last char.

$j = m - 1$

$i = i + m - 1 - (j^{old} - 1)$

Case #3: $L[c] = -1$

Use case #1

Unified: $i = i + m - 1 - \min(LC(c), j^{old} - 1)$

Worst case: $O(nm) \Rightarrow$ Have string where pattn almost there, but first char incorrect.

↳ Good suffix heuristic: $O(n + m + |\Sigma|)$

Suffix Trees

Create trie of suffixes of text

Search pattn $\Rightarrow$ it will occur as prefix in trie!

space : $O(n \cdot |\Sigma|)$

Constructing trie: $O(n \cdot |\Sigma|)$ ↳ Searching for character insert

Searching: $O(m)$

Conclusion:

- Good
- Slow for constr. & takes up space.

Suffix Arrays

Also:

1. List all suffixes & put in order of appearance.

T = aaronaa$

0 aaronaa$
1 aronaa$
2 ronaa$
3 onaa$
4 naa$
5 aa$
6 a$

2. Sort alphabetically

0 $
1 a$
2 aaronaa$
3 aronaa$
4 naa$
5 onaa$
6 ronaa$

3. Suffix array: map sorted posn → appearance order in T

A^S :

0	1	2	3	4	5	6
6	5	0	1	4	3	2

4. Matching of pattn:

Binary search on suffix array!

search (A^S, P): (A^S: 0 → n-1)

$l = 0$, $r = n-1$

while $l < r$

$v \leftarrow \lfloor \frac{l+r}{2} \rfloor$

$i \leftarrow A^S[v]$

$s \leftarrow$ strcmp (T(i ... i + m - 1), P) (Same pattn size)

if $s < 0$:

$l \leftarrow v + 1$

else if $s > 0$:

$r \leftarrow v - 1$

else:
return 'found at ' + i

Analysis:

Construction:

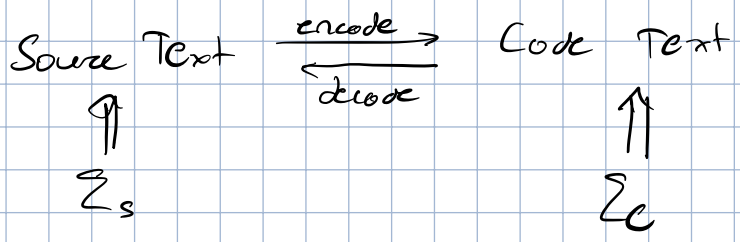
MSD radix sort $\Rightarrow \Theta(n^2)$ worst case, but can improve to $O(n \log n)$

Search:

$\Theta(\log n)$ comp, $\Theta(m)$ string comp $\Rightarrow \Theta(m \log n)$

MODULE 10: COMPRESSION

Encoding & Decoding



Texts are stored as streams $\Rightarrow$ encode one char at a time.

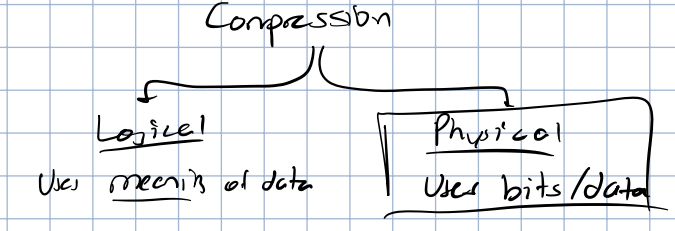
Analysis: compression ratio:

$$\frac{|C| \cdot \log_2 |\Sigma_c|}{|S| \cdot \log_2 |\Sigma_s|} \Rightarrow$$

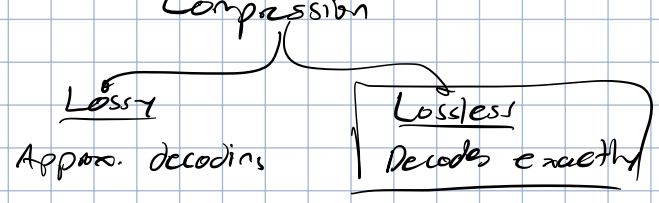
Good compression $\Rightarrow$ ratio < 1

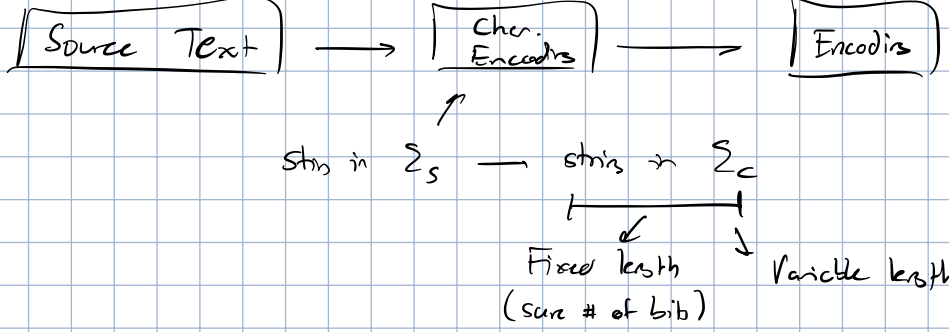
Types:

① Logical vs. physical



② Lossy vs. lossless





Encodings → Char. Encodings → Source

Could be diff. rep (diff D.S.) of encodings
Prefix free
↑ trie

Analysis:

- Encode: $O(|T|)$
- Decode: $O(|C| + |\Sigma_S|)$ ⇒ if impl. as trie

Huffman Encoding

Guarantee to get shortest string if using binary

Also:

① Calculate character frequencies
↳ Need access to entire stream

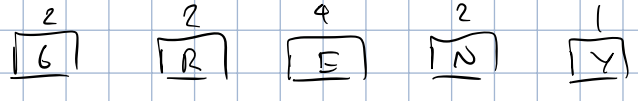
② Put character & frequency as trie nodes

③ Join 2 least frequent nodes into tries. Tie ⇒ alphabetical ordering.
↳ 0 to left child, 1 for right child
↳ New node has freq. = sum of left + right freq.

④ Repeat until all joined.

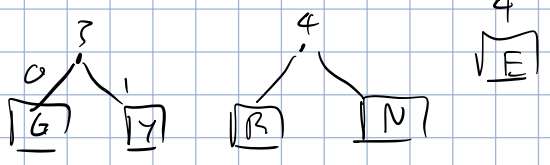
Ex:// Text: G: 2 E: 4 Y: 1
B: 2 N: 2

①



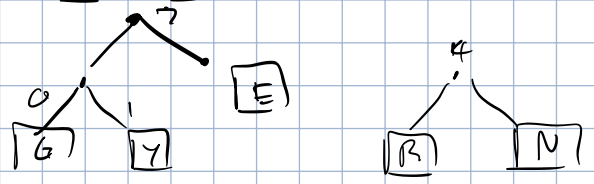
②

③

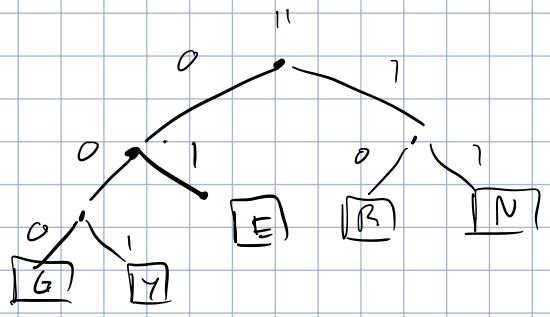


⇒ Smaller tree is given pref.

④



(5)



Use min-heap to store min. freq nodes. $\Rightarrow O(|\Sigma_s| \log(|\Sigma_s|))$

The actual tree encoding $\Rightarrow O(|\Sigma_s| + c)$

Total encoding time: $O(|\Sigma_s| \log(|\Sigma_s|) + c)$

" decoding " : $O(|c|)$

Notes:

① Tree is not unique

② Tree must be encoded to decoding

③ Doesn't use storm $\Rightarrow$ needs entire text (reading char $\times 2$)

Run Length Encoding

Uses multi-character encoding: multiple source chars $\rightarrow$ 1 codeword

Assumption: $\Sigma_s = \{0, 1\}$

Strength: multiple same chars in string (0...0, 1...1...)

Naive:

1. First bit indicates bit that we start off with
2. Following #s tell us sequence of bit

Ex: // 0 0 0 0 0 1 1 1 1 0 0 0 1 1 1 1

$\Downarrow$

5 zeros

0 5 4 3 5

↑ start off w/ 0

4 1s

Elias Gamma

To encode run length $k \Rightarrow \lfloor \log k \rfloor$ copies of 0s + binary rep. of k

To decode: take # of 0s + 1 ⇒ # of bits to read next for repeat of bit.

Ex:// 1111111 001 000000 ... 0 1 ... 1 (20 zeros, 15 ones)

① Start by putting 1st bit in encoding.

$C = 1$

② Find $k = 7$

③ Encode:

$\lfloor \log 7 \rfloor = 2$

$C = 1 + 00111$

④ $k = 2$

↳ $C = 1 + 00111 + 010$

⑤ $k = 1$

$C = 1 + 00111 + 010 + 1$.

⋮

Ex:// Decode RLE 0 0001101 001001010

① Read initial bit

0

② Read as many 0s

3 0s ⇒ next 3+1 bits for decoding

③ Read 3+1 bits.

$1101 = 13$

Run length of 13 for 0 ⇒ switch to 1 for next decoding

$S = 0 \ldots 0$ (13)

④ Read as many 0s

2 0s ⇒ next 2+1 bits for run length 2

⑤ Read $2+1$ bits

$100 = 4 \Rightarrow$ switch to 0 for next decode.

$S = \underbrace{0\cdots0}_{12}\ \underbrace{1\cdots1}_{4}$

Analysis:

- Encode: $(|S| + |C|)$
- Decode: $(|S| + |C|)$

Note:

- It can fail $\Rightarrow$ we don't have enough bits when looking for $k+1$ bits in decoding.
- Depends on input: if no repeats, doesn't work well
  - ↳ If $k < 6 \Rightarrow$ no compression
    - ↳ $k=2$, $k=4 \Rightarrow$ expansion.

LZW

Major idea: encode substrings $\Rightarrow$ if repeated, we already have encoding!

Really good if you have repeating substrings

This allows dictionary to be created during reading / decoding

↳ No need to send dictionary!

Encoding Algo

1. Initialize D_0 for Σ_S

D_0: { char 1: 0, char 2: 1, ... char n: $|\Sigma_S| - 1$ }

2. At each iteration i

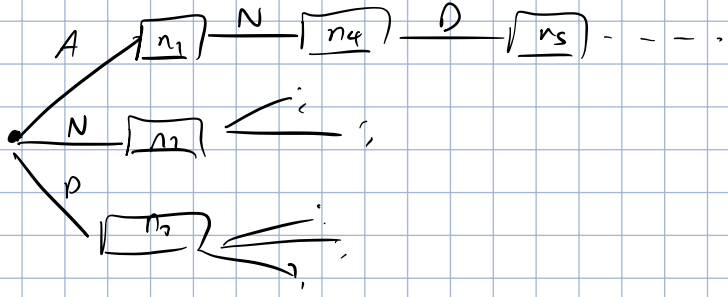
a) Pop off char i

b) Find longest substring starting at i already in dict

c) $D_{i+1} = D_i$ - append ({ $\underbrace{T_i \ldots T_{i+k}}_{\text{Substring that is just beyond } D_i}$, next avail code })

Dict implementation for encoding trie

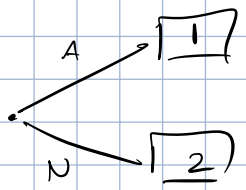
- Ex:// $\Sigma_S = \{A, N, D\}$



Ex:// $\Sigma_S = \{A, N\}$. Encode: ANANANANNNA

① Initialize:

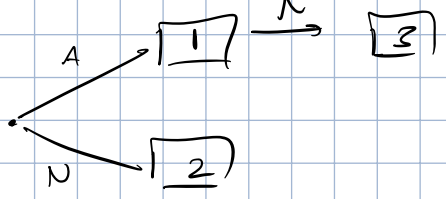
D_0:



② Consume as much chars as we can.

A → code: 1

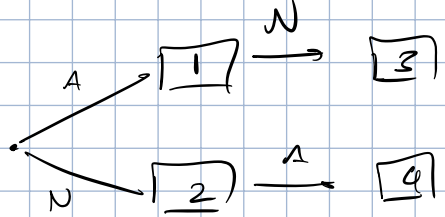
Add AN → code: 2



③ N ✓ → A ✗

↳ code: 1. append (2)

↳ Dict: D_1. append ('A', 4)



④ A ✓ → N ✓ → A ✗

↳ code. append (12). append (5)

↳ Dict: D_2. append (ANA, 5)

String to encode ↓ $O(|s|)$

Decoding Algo

Create dict while encoding, but will be behind by 1 char.

Ex:// C = 65, 76, 128, 130, 78, 129

① Initialize D_0:

We can use a reg. Dict.

$D_0 = \{$ codeword $\rightarrow$ string $\}$

codeword: 0 — 65

string: ␣ — ... — Z

② Consume:

a) 65

$D_0(65) = A$

$S = A$

$S\_prev = A$

b) 78

$D_0(76) = N$

$S = AN$

$D_0.add(next\_avail\_code, S\_prev + S[0])$

↳ $D_0.add(128, AN)$

$S\_prev = N$

c) 128

$D_0(128) = AN$

$S = ANAN$

$D_0.add(129, NA)$

$S\_prev = AN$

d) 130

130 not in dict

$S.append(S\_prev + S\_prev[0])$

↳ (ANA)

⋮

Time: $O(|s|)$

bzip 2

Uses transforms ⇒ makes text easier to decode

$T_0 \rightarrow$ [Burrow Wheeler] $\rightarrow T_1 \rightarrow$ [MTF] $\rightarrow T_2 \rightarrow$ [RLE] $\rightarrow T_3 \rightarrow$ [Huffman]

MTF Transform

Store dict in array: [a, b, c, d, _ _ _] ← code = index of char

Each time it's used → move alphabet to front

↳ Repeated characters ⇒ whole bunch of 0s

Encoding

1. Get text in array
2. Put index in string
3. Move to front

Decoding

1. start w same dict
2. Do exact same ops.

} We set skewed distribution ⇓ Really nice for Huffman.

Ex: MISSISSIPPI

D_0: [a: 0, b: 1, _ _ _ _ _ _ z: 25]

① Consume M:

$M = 13 \Rightarrow D = 13, \ldots$

Move M to front: [M, A, B, _ _]

② I:

$I = 9, \Rightarrow D = 13, 9$

[I, M, A, _ _]

③ S:

$S = 18 \Rightarrow D = 13, 9, 18$

[S, M, A _ _]

④ S:

$S = 0 \Rightarrow D = 13, 9, 18, 0$

[S, M, A, B, ...]

Burrows Wheeler Transform.

Obj: source text ⟶ coded text (contain exact same letters, in diff. order)

Assumption: EOS char $

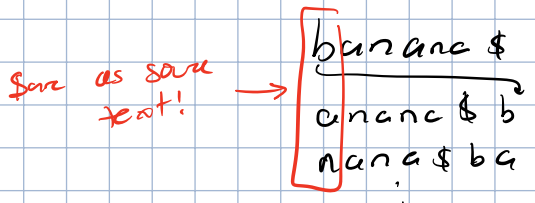
Use cyclic shifts: banana $

↓

a $ banan

Encoding

① Create an array of all cyclic shifts.



② Sort array alphabetically

↳ This is what make this awesome for MTF transform ⇒ same chars. will line up.

↳ Efficient: suffix sorting (MSD sort in $O(n \log n)$)

③ Use last column for encoding.

$O(n \log n + n) \in O(n \log n)$

Sort — Cyclic, last char

Decoding

① Create array w/ last character = coded char.

C = $ aaaaaa e ff f ll l st

[_ _ _ _ _ _ (a, 0) ← run number.

(f, 1)

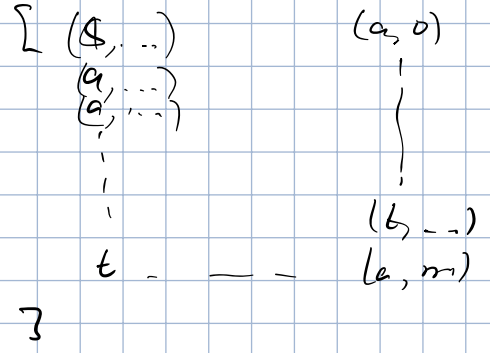
(f, 2)

(t, 1)

(a, n-1)

]

② Generate first char for each string by taking last col → sorting.



③ Find row w/ last character as $ (row ↓ strings).

④ Take 1st char → decoding.

⑤ Take 1st char row number → find row # of sorted string & get first char

⑥ Repeat until we get $.

Ex:// C = ard$rcaaaabb

① Array at end:

end =

	0	1	2	3	4	5	6	7	8	9	10	11
A:	a, 0	r, 1	d, 2	$, 3	r, 4	c, 5	a, 6	a, 7	a, 8	a, 9	b, 10	b, 11

② Sort:

A =

	0	1	2	3	4	5	6	7	8	9	10	11
A:	$, 3	a, 0	a, 6	a, 7	a, 8	a, 9	b, 10	b, 11	c, 5	d, 2	r, 1	r, 4

③ Look @ end to find $ ⇒ index 3 in end.

④ A[3] = (a, 6) ⇒ S = a, index = 6

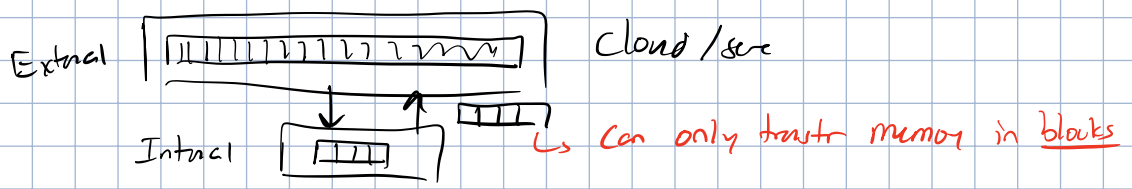
⑤ A[index] = (b, 10) ⇒ S = ab, index = 10

⑥ A[index] = (r, 1) ⇒ S = abr, index = 1

Time complexity: $O(n + |\Sigma_S|)$

MODULE 11: EXTERNAL MEMORY

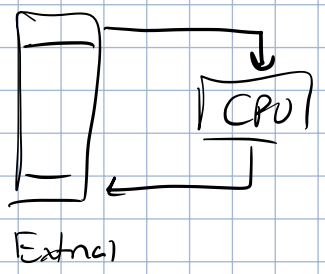
Motivation



Block transfers are $$ ⇒ minimize block transfers.

Block transfers dominate running time ⇐ all analysis on block transfers.

Stream Based Algorithms



External memory: stream blocks

Efficient

↳ Input: $O\left(\frac{n_{input}}{B}\right)$

↳ block size

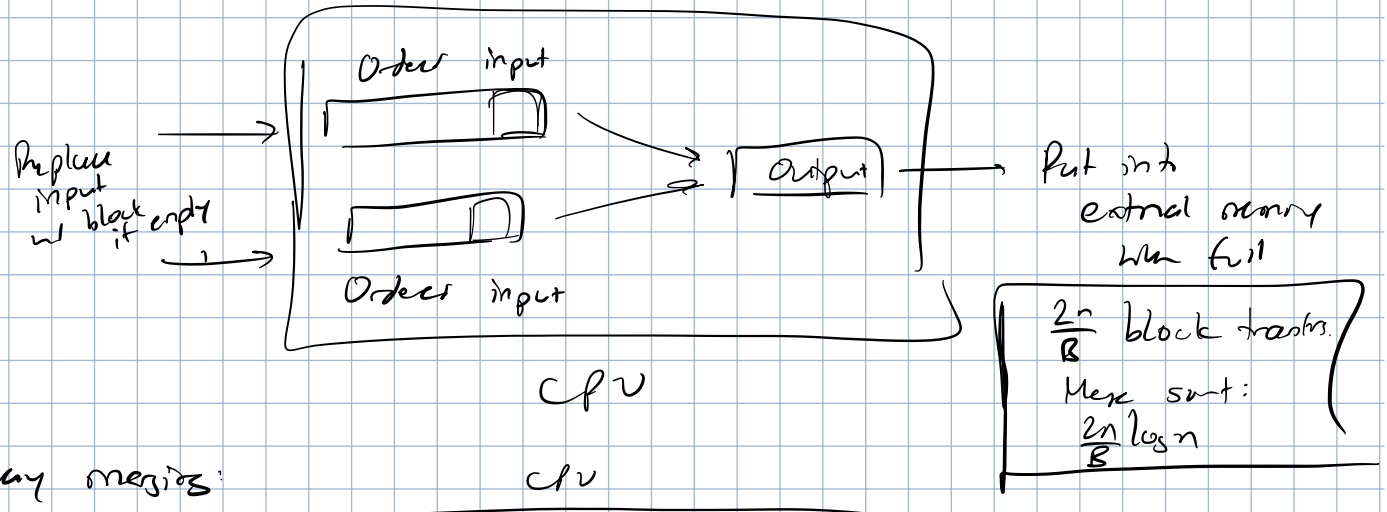
Output: $O\left(\frac{s_{output}}{B}\right)$

Any stream-based algo cannot do better

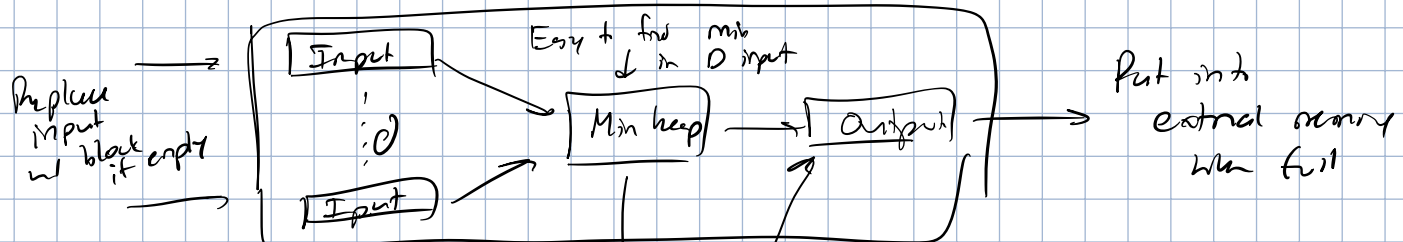
↳ Text matching & compression.

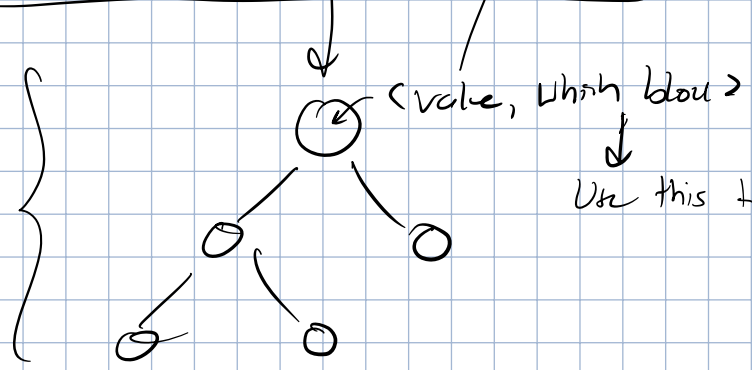
External Sorting

Merging is a pretty good candidate:



D-way merging:





Runtime:

$\Theta(n \log d)$ ← Taking n items; $\log d$ deletion / insertion } just looking at RAM

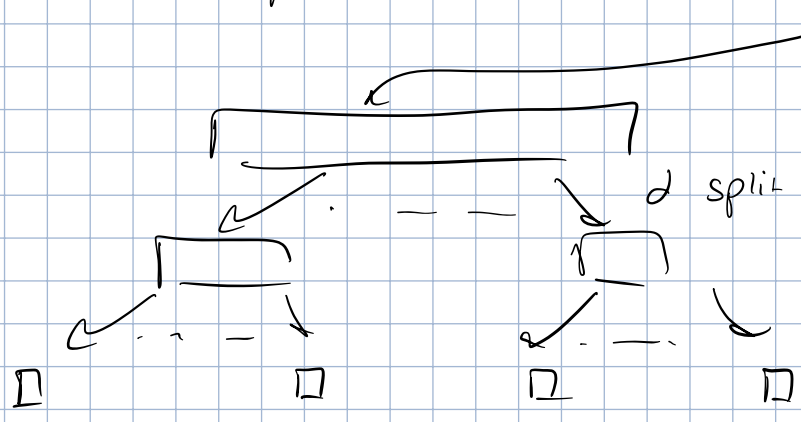
# of block transfers:

We can only put $d+1$ blocks into internal memory
(input, output)

1 block transfer $\Rightarrow \Theta(n/B)$
↳ Stayed same!

d-way merge sort

Also:
1. Split input into d arrays
2. Merge each d parts together = d-way merge } Recursion depth: $O(\log_d n)$



$\log_d n$ height! b/c d splits / level instead of 2.

Runtime:

$$\underbrace{\log_d n}_{\text{depth}} \cdot \underbrace{n \log d}_{\text{merge}} = n \frac{\log n}{\log d} \in \boxed{O(n \log n)}$$

Has same runtime as merge sort, but far better for external

# of block transfers = recursion depth × # of block transfers / level

$$\leq \log_d n \times \frac{2n}{B}$$

We can improve:

Stop at block level $M \Rightarrow M$ bits can fit into array. No need to go to 1

$\therefore$ # of block transfers = recursion depth × # of block trans. / level

$$= \left[\log_d n - \log_d m\right] \times \frac{2n}{B}$$

$$= \log_d n/m \times \frac{2n}{B} \qquad \downarrow \; d \approx \frac{M}{B} - 1$$

($\frac{M}{B}$: # input blocks; -1: output block)

$$\in \Theta\left(\frac{\log(n/M)}{\log(M/B)} \times \frac{n}{B}\right) \Rightarrow \text{Really fast}$$

Minimizing transfers.

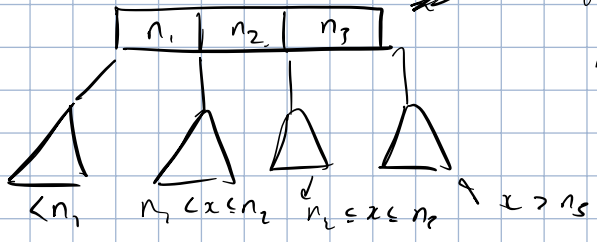
This is optimal. Cannot be better.

2-4 Trees

Motivation: AVL tree has poor memory locality $\Rightarrow$ # of block transfers ↑

2-4 tree is good for internal memory

Idea:



Multiple keys / node.

BST!

Multiway tree:

It stores $b-1$ nodes $\Leftrightarrow b$ subtrees

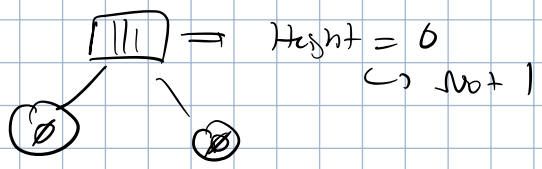
Properties:

- Each node is either 1-node, 2-node, 3-node
  - 1-node: 1 key, 2 subtrees
  - 2-node: 2 keys, 3 subtrees
  - 3-node: 3 keys, 4 subtrees.
- All empty trees are in same level
- Order property: BST property w/ more trees:

$$\langle T_0, n_1, T_1, n_2, \ldots T_3 \rangle$$

(in order of keys)

Note: not counting empty trees for height



Operations:

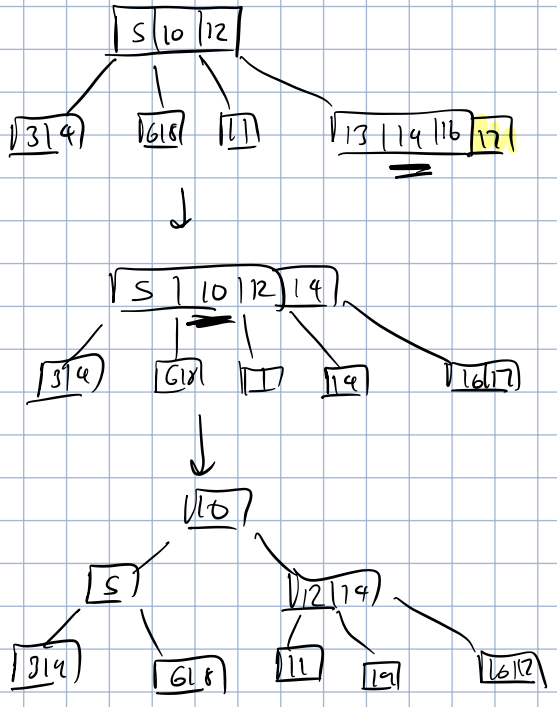
① Search (k): compare k among node keys ⇒ BST

- If k not in tree ⇒ return parent of stopping empty node.

② Insert (k):

1. Search → get leaf node (non-empty)
2. Insert KVP into tree & create another subtree.
3. Split up to restore 1-3 node property.
   a) Find middle node → push to parent
   b) Split up node into 2 subtrees.
4. Repeat until root is balanced. (new root if necessary)

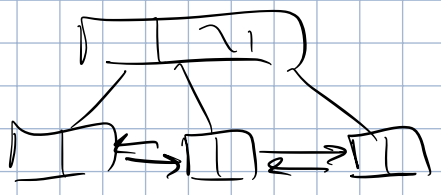
Ex:



③ Delete (k):

Terminologies.

1. Immediate sibling: closest left/right node



} Every node has immediate siblings.

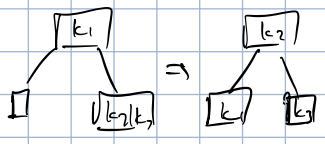
2. Inorder successor: smallest key in right child. (always a leaf).

Also:

1. Search for key
2. If node has non-empty children → swap w/ inorder successor
   ⇓ Node is at leaf
3. Delete key & empty subtree.
4. If underflow (< 1-node in tree):

Case #1: If immediate sibling is "rich" (2/3-node):

a) Transfer key to parent (closest key)



b) Parent will transfer appropriate key to leaf ⇒ no longer empty

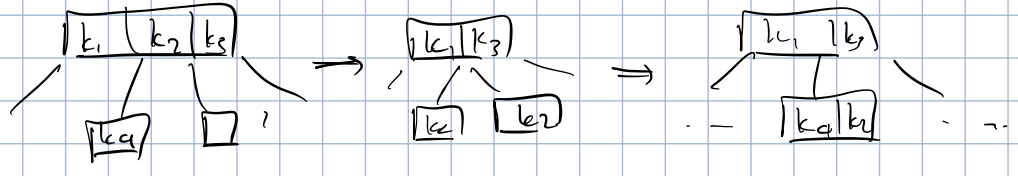
c) Done ⇒ no more repair needed.

Case #2: If both siblings poor

a) Take in-between key in parent → merge in leaf w/ immediate sibling.

b) Repeat until parent not empty

↳ If root becomes empty ⇒ delete it



Analysis:

height $\in O(\log n)$

↳ Ops: $O(\log n)$

Not better than AVL tree, can generalize.

a,b - Tree

Properties:

* ∗ ◦ a: minimum # of subtrees / node
* ∗ ◦ b: maximum "

} Unless root ⇒ can have min. of 2 min. (a doesn't apply to root)

⇓

Used if storing very few keys, (i.e. storing 1 key, but a=3)

◦ Empty leaves @ same level

◦ Order property

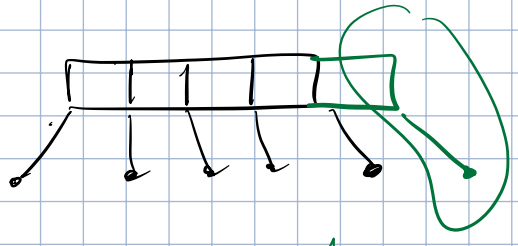
* ∗ ◦ $a \leq \lceil b/2 \rceil = \lfloor \frac{b+1}{2} \rfloor$

Operations: Exact Same!

↳ Clarify insertion:

Too many keys in a node? Whenever # of subtrees > b

Ex: // (3, 5) tree



⇒ For this node: 6 subtrees! ⇒ > b

Added key & subtree

Split ⇒ Splitting into $\lfloor \frac{b+1}{2} \rfloor$ subtrees / split (restores property).

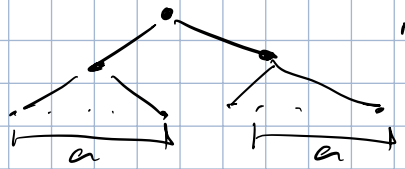
Why do we need $a \leq \lceil \frac{b}{2} \rceil$?

If we had (4,5) - tree ⇒ overflow ⇒ split will create (2,5) subtree which violates (4-5) - tree!

Analysis:

Q: What is height of (a,b) tree?

Find smallest # of KVPs ⇒ height (will give us $O(\dots)$)



min # of KVPs = $1 + 2 + 2a + 2a^2 + \dots$

$= 1 + \sum_{i=1}^{h-1} 2a^i(a-1)$

Skipping root

Root

$= \vdots$

$= 2a^h - 1$

∴ If n KVPs stored:

$$n \geq 2a^h - 1$$

$$\vdots$$

$$h \in O(\log_a n) \in O(\log n)$$

INTERESTING NOTE:

# of KVPs in $[a,b)$ tree = # of empty subtrees -1

Super useful in proofs.

Q: Runtime of ops?

Height $\in O(\log_a n)$

↳ Maximize a to get min. height. $a \leq \lceil \frac{b}{2} \rceil$, $\underbrace{a = b/2}_{\text{Max.}}$

To actually find keys in node ⇒ AVL tree

↳ # of KVPs in node $\in O(b)$

∴ Search in AVL tree = $O(\log b)$

In total:

$$O(\text{find the node} \cdot \text{find a key}) = O(\log_a n \cdot \log b)$$

$$\in O(\log n)$$

Not faster than AVL, but useful for memory.

B Tree

Idea: specialize (a,b) tree s.t. it can do well in external mem.

Want to minimize height

↳ $a = b/2 \Rightarrow \underline{b}$ dictates the entire tree (node)

What should b be? $b \in \Theta$ (block size)

Choose b s.t. node w/ $b-1$ keys can fit into 1 block

$\therefore$ 1 block $\longrightarrow$ 1 node to search in CPU

Why is this nice:

# of block transfers = height of search

$= \theta(\log_b n) \Rightarrow$ Super efficient