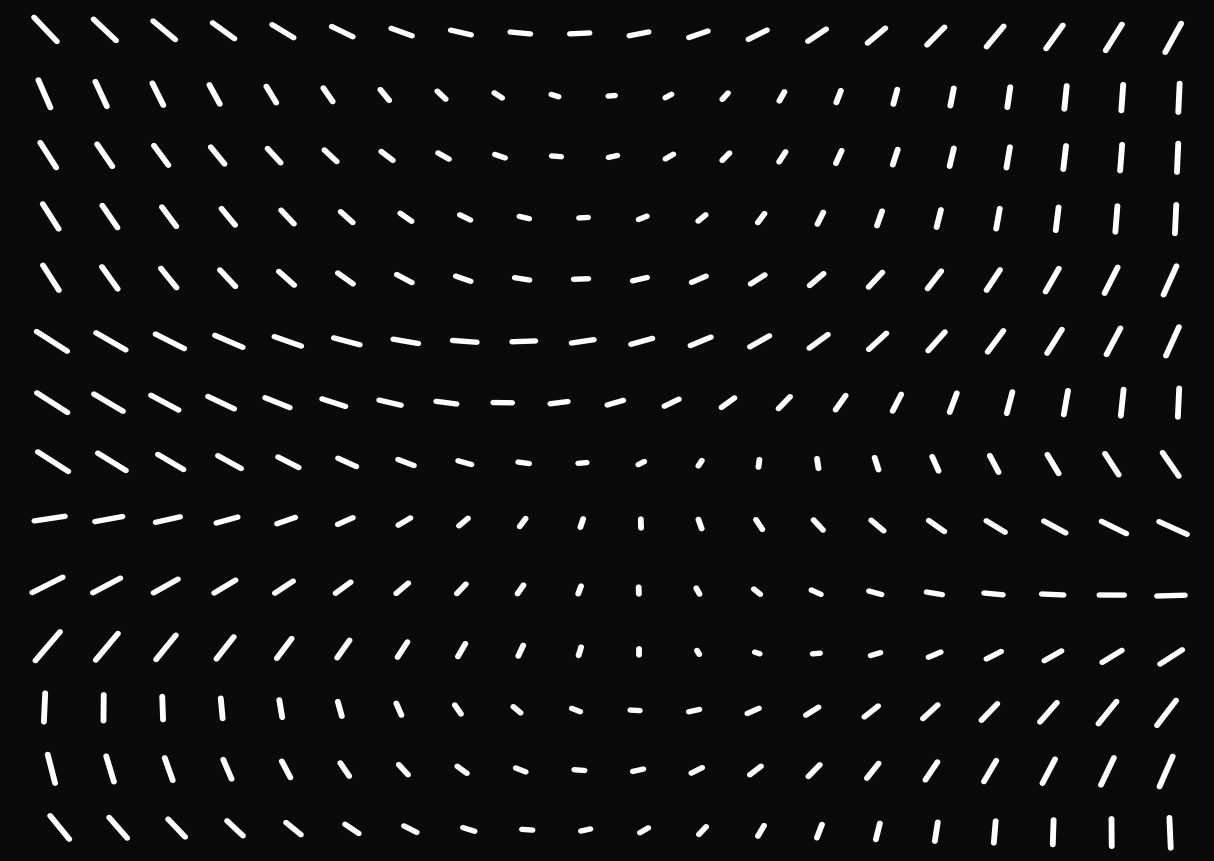


INTRODUCTION AND ASYMPTOTIC ANALYSIS

ALGORITHM DESIGN

- problem: given a problem instance, carry out particular computational task
- problem instance: input for specified problem
- problem sltn: output (must be correct ans) for specified problem instance
- size of problem instance: Size(I) is +ve int that measures size of instance I
- algorithm: step-by-step process for carrying out series of computations, given arbitrary I
  - ↳ e.g. pseudocode
- Algorithm A solves problem Π if for every instance I of Π, A computes valid sltn for instance I in finite time
- program: implementation of algorithm using specified computer language
  - ↳ method of communicating an algorithm to a computer
- pseudocode: method of communicating an algorithm to another person
  - ↳ omits obvious details (e.g. var declarations)
  - ↳ has limited or no error detection
  - ↳ sometimes uses English descriptions
  - ↳ sometimes uses mathematical notation
- can have several algorithms for problem Π
- algorithms in practice:
  1) design algorithm A that solves Π
     - algorithm design
  2) assess correctness & efficiency of A
     - algorithm analysis
  3) if acceptable, implement A

ANALYSIS OF ALGORITHMS PT I

- primarily concerned w/ runtime of program
- sometimes interested in auxiliary space (i.e. amount of additional mem) program requires
- amount of time & mem required by program will depend on Size(I)
- one way to analyze runtime of algorithms / programs is by experimental studies
  1) write program implementing algorithm
  2) run program w/ inputs of varying size & composition
  3) use method (e.g. clock() from time.h) to get accurate measure of actual running time
  4) plot / compare results
  - ↳ disadvantages include:
    - complicated / costly implementation
    - timings are affected by many factors:
      - → hardware (e.g. processor, mem)
      - → software environment (e.g. OS, compiler, programming language)
      - → human (e.g. programmer)
    - can't test all inputs so must choose good sample inputs
    - can't easily compare 2 algorithms / programs
- framework to analyze runtime preferably doesn't require algorithm implementation, is indep of hardware/software env, & takes into account all input instances
- to overcome dependency on hardware/software:
  - ↳ algorithms are in structured HL pseudocode that's language-indep
  - ↳ analysis of algorithms is based on idealized computer model
  - ↳ count # primitive operations instead of time
  - ↳ efficiency of algorithm wrt time is measured in terms of its growth rate (i.e. complexity of algorithm)
- Random Access Machine (RAM) model is set of mem cells that stores 1 item / word of data each

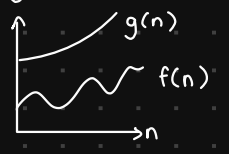
↳ implicitly assume cells are big enough
↳ access to mem location takes constant time
↳ primitive operation takes constant time
  • implicitly assume all ops have similar runtimes on diff systems
↳ running time of program is proportional to # mem accesses + # primitive ops

· to simplify comparisons, use order notation (i.e. constants in front & lower orders usually don't matter)
↳ only consider largest asymptotic
  • e.g. what's larger: $100n$ or $10n^2$?
    ANS: $10n^2$
↳ special case: e.g. what's larger: $4n^3$, $300n^{2.807}$, or $10^{67} n^{2.373}$?
  ANS: unclear

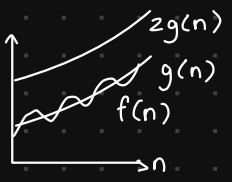
ASYMPTOTIC NOTATION

· O-notation: $f(n) \in O(g(n))$ (i.e. f is asymptotically bounded above by g) if there exist constant $c > 0$ & $n_0 \geq 0$ st $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$ (where $n \in \mathbb{Z}$)

↳ e.g.



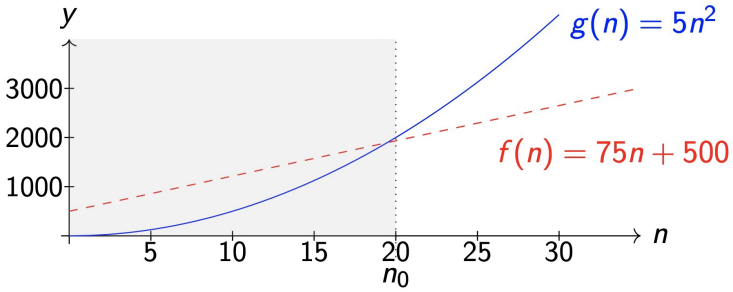
$f(n) \leq g(n)$ for any $n \geq 0$
so $f(n) \in O(g(n))$ w/ $c = 1, n_0 = 0$



$f(n) \leq 2g(n)$ for $n \geq 0$
so $f(n) \in O(g(n))$ w/ $c = 2, n_0 = 0$

↳ e.g.

Example: $f(n) = 75n + 500$ and $g(n) = 5n^2$ (e.g. $c = 1, n_0 = 20$)



↳ abs value signs in defn are irrelevant for analysis of runtime/space, but useful in other applications of asymptotic notation
↳ no optimization of constants is needed (i.e. just need c & n_0 that works)

· e.g.

In order to prove that $2n^2 + 3n + 11 \in O(n^2)$ from first principles, we need to find c and n_0 such that the following condition is satisfied:

$$0 \leq 2n^2 + 3n + 11 \leq c\,n^2 \text{ for all } n \geq n_0.$$

note that not all choices of c and n_0 will work.

Take $f(n) = 2n^2 + 3n + 11$, $g = n^2$. We prove $f(n) \in O(g(n))$

$2n^2 = 2n^2$, $n \geq 0$
$3n \leq 3n^2$, $n \geq 0$
$11 \leq 11n^2$, $n \geq 1$
$f(n) \leq 16n^2$ for $n \geq 1$

ANS: We take $c = 16$, $n_0 = 1$. This proves $f(n) \in O(n^2)$.
↳ $f(n) \in O(n^2)$, but also $f(n) \in O(n^{10})$
· we want tight asymptotic bound
· O-notation is simply upper bound
↳ e.g. $f(n) \in O(n^2)$
$g(n) \in O(n^3)$ ⟶ $f(n)$ is not necessarily better than $g(n)$

· Ω-notation: $f(n) \in \Omega(g(n))$ (i.e. f is asymptotically bounded below by g) if there exist constants $c > 0$ & $n_0 \geq 0$ st $c|g(n)| \leq |f(n)|$ for all $n \geq n_0$

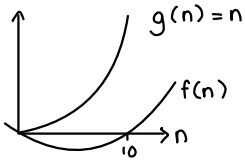
- **Θ-notation**: $f(n) \in \Theta(g(n))$ (i.e. f is **asymptotically tightly bounded** by g) if there exist constants $c_1, c_2 > 0$ & $n_0 \geq 0$ st $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ for all $n \geq n_0$
- $f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$
- e.g.

Prove that $f(n) = 2n^2 + 3n + 11 \in \Omega(n^2)$ from first principles.

$2n^2 = 2n^2 \quad n \geq 0$
$3n \geq 0 \quad n \geq 0$
$11 \geq 0 \quad n \geq 0$

$f(n) \geq 2n^2$ for $n \geq 0$. We take $c = 2$ & $n_0 = 0$. This proves that $f(n) \in \Omega(n^2)$.

Prove that $\frac{1}{2}n^2 - 5n \in \Omega(n^2)$ from first principles.



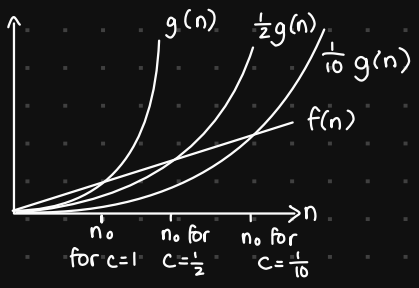
We want to find c, n_0 st $\frac{1}{2}n^2 - 5n \geq cn^2$ for $n \geq n_0$.
↳ take $c = \frac{1}{4}$, then $\frac{1}{2}n^2 - 5n \geq \frac{1}{4}n^2$
$\frac{1}{4}n^2 \geq 5n$
$\frac{1}{4}n \geq 5$
$n \geq 20$

⟶ We take $c = \frac{1}{4}$ & $n_0 = 20$. This proves $f(n) \in \Omega(n^2)$.

Prove that $\log_b(n) \in \Theta(\log n)$ for all $b > 1$ from first principles.

$\log_b(n) = \frac{\log(n)}{\log(b)}$
We prove $\log_b(n) \in O(\log n)$ for all $b > 1$. $\log_b(n) \leq \log n$ for all $n \leq 1$. With $c = 1$ & $n_0 = 1$, this proves $\log_b(n) \in O(\log n)$. Also, $\log_b(n) > \log n$ for all $n > 1$. So, with $c = 1$ & $n_0 = 1$, $\log_b(n) \in \Omega(\log n)$. As such, it follows that $\log_b(n) \in \Theta(\log n)$ for all $b > 1$ for $c_1 = 1$, $c_2 = 1$, $n_0 = 1$.

- **o-notation**: $f(n) \in o(g(n))$ (i.e. f is **asymptotically strictly smaller** than g) if for all constants $c > 0$, there exists a constant $n_0 \geq 0$ st $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$
  ↳ n_0 should depend on c
  ↳



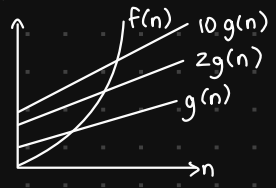
  ↳ $n^d \in o(n^{d+\varepsilon})$ for $\varepsilon > 0$
  ↳ e.g. Prove $10n^2 \in o(n^n)$, given that $c > 0$.
  Find n_0 st $\underbrace{10n^2 \leq cn^n}_{(1)}$ for $n \geq n_0$

  $(1) \iff \frac{10}{c} \leq n^{n-2}$

  In particular, if $\frac{10}{c} \leq n$ & $n \leq n^{n-2}$, then (1) is true
  - $\frac{10}{c} \leq n$ for $n \geq \frac{10}{c}$
  - $n^1 \leq n^{n-2}$ if $1 \leq n-2$, which is $n \geq 3$

  ANS: Taking $n_0 = \max(\frac{10}{c}, 3)$, this proves that $10n^2 \in o(n^n)$

- **ω-notation**: $f(n) \in \omega(g(n))$ (i.e. f is **asymptotically strictly larger** than g) if for all constants $c > 0$, there exists a constant $n_0 \geq 0$ st $|f(n)| \geq c|g(n)|$ for all $n \geq n_0$
  ↳



ALGEBRA OF ORDER NOTATIONS

- **identity rule**: $f(n) \in \Theta(f(n))$
- **transitivity**:
  - ↳ if $f(n) \in O(g(n))$ & $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$
  - ↳ if $f(n) \in \Omega(g(n))$ & $g(n) \in \Omega(h(n))$, then $f(n) \in \Omega(h(n))$
- **max rules**: suppose $f(n) \geq 0$ & $g(n) \geq 0$ for all $n \geq n_0$, then:
  - ↳ $f(n) + g(n) \in O(\max\{f(n), g(n)\})$
  - ↳ $f(n) + g(n) \in \Omega(\max\{f(n), g(n)\})$
  - ↳ e.g. prove $\max\{f(n), g(n)\} \leq f(n) + g(n) \leq 2\max\{f(n), g(n)\}$
    1) $f(n) \leq \max\{f(n), g(n)\}$ & $g(n) \leq \max\{f(n), g(n)\}$
       so $f(n) + g(n) \leq 2\max\{f(n), g(n)\}$
    2) If $\max\{f(n), g(n)\} = f(n)$, then $\max\{f(n), g(n)\} \leq f(n) + g(n)$ since $g(n) \geq 0$
       If $\max\{f(n), g(n)\} = g(n)$, then $\max\{f(n), g(n)\} \leq f(n) + g(n)$ since $f(n) \geq 0$
    ∴ $\max\{f(n), g(n)\} \leq f(n) + g(n) \leq 2\max\{f(n), g(n)\}$ ∎ the identity is proven
- e.g suppose $f(n) \in \Theta(n^2 + \log(n))$
  1) $n^2 + \log(n) \in \Theta(n^2)$ — max rule
  2) $f(n) \in \Theta(n^2)$ — transivity

- suppose that $f(n) > 0$ & $g(n) > 0$ for all $n \geq n_0$ & suppose $L = \lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists, then $f(n) \in \begin{cases} o(g(n)) & \text{if } L = 0 \\ \Theta(g(n)) & \text{if } 0 < L < \infty \\ \omega(g(n)) & \text{if } L = \infty \end{cases}$
  - ↳ result gives sufficient (but not necessary) conditions for stated conclusion to hold
  - ↳ if there's limit, then there's a Θ
    - converse isn't always true
- e.g. let $f(n)$ be polynomial of deg $d \geq 0$, $f(n) = c_d n^d + c_{d-1} n^{d-1} + \ldots + c_1 n + c_0$, for some $c_d > 0$; then, $f(n) \in \Theta(n^d)$
  - ↳ proof:

$$f(n) = c_d n^d + \ldots + c_1 n + c_0, \quad c_d > 0$$
$$\frac{f(n)}{n^d} = c_d + \frac{c_{d-1}}{n} + \ldots + \frac{c_1}{n^{d-1}} + \frac{c_0}{n^d}$$
$$\lim_{n\to\infty} \frac{f(n)}{n^d} = c_d$$

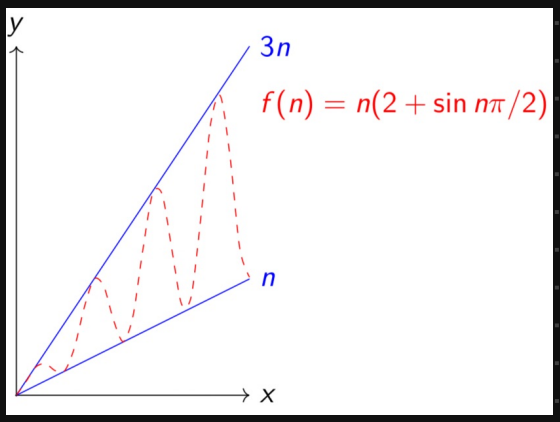
By limit test, $f(n) \in \Theta(n^d)$

- e.g prove that $n(2 + \sin\frac{n\pi}{2})$ is $\Theta(n)$
  - ↳ note: $\lim_{n\to\infty} (2 + \sin\frac{n\pi}{2})$ DNE
  - ↳ proof:

$$-1 \leq \sin\frac{n\pi}{2} \leq 1$$
$$1 \leq 2 + \sin\frac{n\pi}{2} \leq 3$$
$$n \leq n(2 + \sin\frac{n\pi}{2}) \leq 3n$$

Thus, this proves $n(2 + \sin\frac{n\pi}{2}) \in \Theta(n)$

  - ↳



- e.g. compare growth rates of $f(n) = \log n$ & $g(n) = n$ using l'Hôpital's rule
  - ↳ $f(n) = \frac{\ln(n)}{\ln(2)}$ $\quad g(n) = n$
    $f'(n) = \frac{1}{\ln(2)} \cdot \frac{1}{n}$ $\quad g'(n) = 1$
    $\lim_{n\to\infty} \frac{f'(n)}{g'(n)} = \lim_{n\to\infty} (\frac{1}{\ln(2)} \cdot \frac{1}{n})$
    $= 0$

Since $\lim_{n\to\infty} f(n) = \infty$, $\lim_{n\to\infty} g(n) = \infty$, & $\lim_{n\to\infty} \frac{f'(n)}{g'(n)} = 0$, then by l'Hôpital, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

$\therefore f(n) \in o(g(n))$ by limit test

- e.g. compare growth rates of $f(n) = (\log n)^c$ & $g(n) = n^d$, where $c > 0$ & $d > 0$ are arbitrary #s
  - ↳ $f(n) = \log(n)$ $\quad f'(n) = \frac{1}{\ln 2} \cdot \frac{1}{n}$
    $g(n) = n^k$, $k > 0$ $\quad g'(n) = kn^{k-1}$
    $\lim_{n\to\infty} \frac{f'(n)}{g'(n)} = \frac{1}{\ln 2} \cdot \frac{1}{n} \cdot \frac{1}{kn^{k-1}}$
    $= \frac{1}{\ln 2} \cdot \frac{1}{n} \cdot \frac{n}{kn^k}$
    $= \frac{1}{\ln 2} \cdot \frac{1}{kn^k}$
    $= 0$

  $\therefore \lim_{n\to\infty} \frac{\log(n)}{n^k} = 0$ so $f(n) \in o(g(n))$

  Now:
  $f(n) = \log(n)^c$
  $g(n) = n^d$

  $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log(n)^c}{n^d}$
  $= \lim_{n\to\infty} \left(\frac{\log(n)}{n^{d/c}}\right)^c$
  $= 0$ ← let $k = \frac{d}{c}$

  $\therefore f(n) \in o(g(n))$

- depending on rltnship btwn $f(n)$ & $g(n)$, can draw conclusions abt their **growth rates**
  - ↳ if $f(n) \in \Theta(g(n))$, then growth rates of $f(n)$ & $g(n)$ are **same**
  - ↳ if $f(n) \in o(g(n))$, then growth rate of $f(n)$ is **less than** $g(n)$'s
  - ↳ if $f(n) \in \omega(g(n))$, then growth rate of $f(n)$ is **greater than** $g(n)$'s
- typically, $f(n)$ is complicated & $g(n)$ is chosen to be simple fcn
- commonly encountered growth rates in analysis of algorithms (inc order of growth rate):
  - ↳ $\Theta(1)$ is constant
  - ↳ $\Theta(\log n)$ is logarithmic
  - ↳ $\Theta(n)$ is linear
  - ↳ $\Theta(n \log n)$ is linearithmic
  - ↳ $\Theta(n \log^k n)$ for some constant k is quasi-linear
  - ↳ $\Theta(n^2)$ is quadratic
  - ↳ $\Theta(n^3)$ is cubic
  - ↳ $\Theta(2^n)$ is exponential
- how running time is effected when size of problem instance **doubles**:

• constant complexity: $T(n) = c$	$\rightsquigarrow T(2n) = c.$
• logarithmic complexity: $T(n) = c \log n$	$\rightsquigarrow T(2n) = T(n) + c.$
• linear complexity: $T(n) = cn$	$\rightsquigarrow T(2n) = 2T(n).$
• linearithmic $\Theta(n \log n)$: $T(n) = cn \log n$	$\rightsquigarrow T(2n) = 2T(n) + 2cn.$
• quadratic complexity: $T(n) = cn^2$	$\rightsquigarrow T(2n) = 4T(n).$
• cubic complexity: $T(n) = cn^3$	$\rightsquigarrow T(2n) = 8T(n).$
• exponential complexity: $T(n) = c\,2^n$	$\rightsquigarrow T(2n) = (T(n))^2/c.$

- some **rltnships** btwn order notations:
  - ↳ $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$
  - ↳ $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
  - ↳ $f(n) \in o(g(n)) \Leftrightarrow g(n) \in \omega(f(n))$
  - ↳ $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$
  - ↳ $f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$
  - ↳ $f(n) \in o(g(n)) \Rightarrow f(n) \notin \Omega(g(n))$
  - ↳ $f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$
  - ↳ $f(n) \in \omega(g(n)) \Rightarrow f(n) \notin O(g(n))$

ANALYSIS OF ALGORITHMS PT2

goal is to use asymptotic notation to simplify run-time analysis

- running time of algorithm depends on **input size n**
- identify **primitive ops** that require $\Theta(1)$ time
- complexity of loop is **sum** of complexities of each iteration of loop
  - ↳ for **nested loops**, start w/ innermost loop & proceed outwards
    - gives nested summations
- 2 strategies for run-time analysis:
  - ↳ **brute force**: use Θ-bounds throughout analysis & obtain Θ-bound for complexity of algorithm
  - ↳ **separate into O & Ω**: prove O-bound & matching Ω-bound separately

- e.g.

```
Test1(n)
1.  sum ← 0
2.  for i ← 1 to n do
3.      for j ← i to n do
4.          sum ← sum + (i − j)^2
5.  return sum
```

Let $T_1(n)$ be runtime. Let $S_1(n)$ be #times inner loop is executed.

Brute force:

$T_1(n) = \Theta(S_1(n))$

$$\begin{aligned} S_1(n) &= \sum_{i=1}^{n} \sum_{j=i}^{n} 1 \\ &= \sum_{i=1}^{n} (n-i+1) \\ &= n^2 + n - \sum_{i=1}^{n} i \\ &= n(n+1) - \frac{n(n+1)}{2} \\ &= \frac{n(n+1)}{2} \\ &= \frac{n^2+n}{2} \end{aligned}$$

So, $T_1(n) \in \Theta(n^2)$.

Separate into O & Ω:

1) $$\begin{aligned} S_1(n) &\le \sum_{i=1}^{n} \sum_{j=1}^{n} 1 \\ &= \sum_{i=1}^{n} n \\ &= n^2 \end{aligned}$$

$\therefore S_1(n) \in O(n^2)$

2) $$\begin{aligned} S_1(n) &= \sum_{i=1}^{n} \sum_{j=i}^{n} 1 \\ &\ge \sum_{i=1}^{n/2} \sum_{j=i}^{n} 1 \qquad \text{(assume } n \text{ is divisible by 2)} \\ &\ge \sum_{i=1}^{n/2} \sum_{j=n/2}^{n} 1 \\ &= \sum_{i=1}^{n/2} \frac{n}{2} \\ &= \left(\frac{n}{2}\right)^2 \\ &= \frac{n^2}{4} \end{aligned}$$

$\therefore S_1(n) \in \Omega(n^2)$

We see $S_1(n) \in \Theta(n^2)$ & $T(n) \in \Theta(n^2)$.

- e.g.

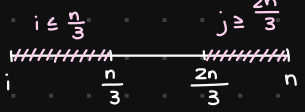
```
Test2(A, n)
1.  max ← 0
2.  for i ← 1 to n do
3.      for j ← i to n do
4.          sum ← 0
5.          for k ← i to j do
6.              sum ← A[k]
7.  return max
```

Let $T_2(n)$ be run-time. Let $S_2(n)$ be #times line 6 is executed.

1) $$\begin{aligned} S_2(n) &\le \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} 1 \\ &= n^3 \end{aligned}$$

$\therefore S_2(n) \in O(n^3)$

2) We'll try to restrict ranges so that inner sum is at least $\frac{n}{3}$.



$$S_2(n) \geq \sum_{i=1}^{n/3} \sum_{j=2n/3}^{n} \sum_{k=n/3}^{2n/3} 1$$

$$\geq \left(\frac{n}{3}\right)^3$$

$$= \frac{n^3}{27}$$

$$\therefore S_2(n) \in \Omega(n^3)$$

We see $S_2(n) \in \Theta(n^3)$ so $T(n) \in \Theta(n^3)$.

- algorithms can have diff running times on 2 instances of same size
  - ↳ let $T_A(I)$ denote run-time of algorithm A on instance I
    - **worst-case**: $T_A(n)^{worst} = \max\{T_A(I) : Size(I) = n\}$
      - ↳ maps input size to longest run-time for any input the size of n
    - **best-case**: $T_A(n)^{best} = \min\{T_A(I) : Size(I) = n\}$
      - ↳ maps input size to shortest run-time for any input the size of n
    - **avg-case**: $T_A(n)^{avg} = \frac{1}{|\{I : Size(I)=n\}|} \sum_{\{I : Size(I)=n\}} T_A(I)$
      - ↳ maps input size to avg run-time over all instances of size n
  - ↳ e.g insertion sort

```
Test3(A, n)
A: array of size n
1.   for i ← 1 to n − 1 do
2.       j ← i
3.       while j > 0 and A[j] < A[j − 1] do
4.           swap A[j] and A[j − 1]
5.           j ← j − 1
```

Best case:
A is in inc order so while loop on line 3 is nvr run. $\Theta(1)$ is complexity for each execution of while loop. Total run-time is $\Theta(n)$.

Worst case:
A is in dec order so $\Theta(i)$ is complexity for each execution of while loop. Total run-time is $\Theta(n^2)$.

- try not to compare algorithms using O-notation b/c:
  - ↳ worst-case run-time may only be achieved on some instances (we don't know freq)
  - ↳ O-notation is upper bound only so we should always use Θ-notation

ANALYSIS FOR MERGESORT

design idea for **MergeSort** on array A of n ints:

1) split A into 2 subarrays
   - A_L consists of first $\lceil \frac{n}{2} \rceil$ elmts in A
   - A_R consists of last $\lfloor \frac{n}{2} \rfloor$ elmts in A
2) recursively run MergeSort on A_L & A_R
3) after A_L & A_R are sorted, use fcn Merge to merge them into single sorted array

↳

```
MergeSort(A, n, ℓ ← 0, r ← n − 1, S ← NIL)
A: array of size n, 0 ≤ ℓ ≤ r ≤ n − 1
1.   if S is NIL   initialize it as array S[0..n − 1]
2.   if (r ≤ ℓ) then
3.       return
4.   else
5.       m = ⌊(r + ℓ)/2⌋
6.       MergeSort(A, n, ℓ, m, S)
7.       MergeSort(A, n, m + 1, r, S)
8.       Merge(A, ℓ, m, r, S)
```

- 2 tricks to reduce run-time & auxiliary space:
  - → recursion uses params that indicate range of array that needs to be sorted

→ array used for copying is passed along as param

↳

Merge(A, ℓ, m, r, S)
$A[0..n-1]$ is an array, $A[\ell..m]$ is sorted, $A[m+1..r]$ is sorted
$S[0..n-1]$ is an array

1. copy $A[\ell..r]$ into $S[\ell..r]$
2. $(i_L, i_R) \leftarrow (\ell, m+1)$;
3. **for** $(k \leftarrow \ell; k \leq r; k{+}{+})$ **do**
4. **if** $(i_L > m)$ $A[k] \leftarrow S[i_R{+}{+}]$ (when pointer is at end of 1st half)
5. **else if** $(i_R > r)$ $A[k] \leftarrow S[i_L{+}{+}]$ (when pointer is at end of 2nd half)
6. **else if** $(S[i_L] \leq S[i_R])$ $A[k] \leftarrow S[i_L{+}{+}]$
7. **else** $A[k] \leftarrow S[i_R{+}{+}]$

- e.g. $n=5$, $\ell=0$, $r=n-1=4$, $m=2$ w/ Merge
  $A = [2\ 4\ 7\ 5\ 6] \rightarrow S = [2\ 4\ 7\ 5\ 6]$
  (i_L on 4, i_R on 5)

  $k=0$, $A=[2\ \_\ \_\ \_\ \_]$ $S=[2\ 4\ 7\ 5\ 6]$
  $k=1$, $A=[2\ 4\ \_\ \_\ \_]$ $S=[2\ 4\ 7\ 5\ 6]$
  $k=2$, $A=[2\ 4\ 5\ \_\ \_]$ $S=[2\ 4\ 7\ 5\ 6]$
  $k=3$, $A=[2\ 4\ 5\ 6\ \_]$ $S=[2\ 4\ 7\ 5\ 6]$
  $k=4$, $A=[2\ 4\ 5\ 6\ 7]$ $S=[2\ 4\ 7\ 5\ 6]$
- e.g. let $T(n)$ denote time to run Mergesort on array of length n
  ↳ initialize array takes time $\Theta(n)$
  ↳ recursively call MergeSort takes time $T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor)$
  ↳ call Merge takes time $\Theta(n)$
  ↳ express $T(n)$ as recurrence:

$$T(n) = \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + \Theta(n) & , n > 1 \\ \Theta(n) & , n = 1 \end{cases}$$

↳ simplification 1: replace $\Theta(n)$ w/ constants

$$T(n) = \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + cn & , n > 1 \\ c & , n = 1 \end{cases}$$

↳ simplification 2: assume $n = 2^k$ for some $k \geq 0$

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + cn & , n > 1 \\ c & , n = 1 \end{cases}$$

For $n = 2^k$:

$$\begin{aligned} T(n) &= T(2^k) \\ &= 2T(2^{k-1}) + c2^k \\ &= 2(2T(2^{k-2}) + c2^{k-1}) + c2^k \\ &= 2^2 T(2^{k-2}) + c2^k + c2^k \\ &= 2^2(2T(2^{k-3}) + c2^{k-3}) + 2c2^k \\ &= 2^3 T(2^{k-3}) + 3c2^k \\ &\vdots \\ &= 2^k T(1) + kc2^k \\ &= 2^k c + kc2^k \\ &= c2^k(1+k) \\ &= cn(\log n + 1) \\ &= cn\log n + cn \end{aligned}$$

Thus, $T(n) \in n\log n$.

- some recurrence relations:

Recursion	resolves to	example
$T(n) = T(n/2) + \Theta(1)$	$T(n) \in \Theta(\log n)$	Binary search
$T(n) = 2T(n/2) + \Theta(n)$	$T(n) \in \Theta(n \log n)$	Mergesort
$T(n) = 2T(n/2) + \Theta(\log n)$	$T(n) \in \Theta(n)$	Heapify (*)
$T(n) = T(cn) + \Theta(n)$ for some $0 < c < 1$	$T(n) \in \Theta(n)$	Selection (*)
$T(n) = 2T(n/4) + \Theta(1)$	$T(n) \in \Theta(\sqrt{n})$	Range Search (*)
$T(n) = T(\sqrt{n}) + \Theta(\sqrt{n})$	$T(n) \in \Theta(\sqrt{n})$	Interpol. Search (*)
$T(n) = T(\sqrt{n}) + \Theta(1)$	$T(n) \in \Theta(\log \log n)$	Interpol. Search (*)

PRIORITY QUEUES

ABSTRACT DATA TYPES

- abstract data type (ADT): descrip of info ? collection of operations on that info
  - ↳ info is accessed only thru ops
- there's various realizations of ADT, which specify:
  - ↳ data structure is how info is stored
  - ↳ algorithm is how ops are performed

- stack is ADT consisting of collection of items w/ops:
  - ↳ push: insert item
  - ↳ pop: remove (? typically return) most recently inserted item
- items are removed in last-in first-out (LIFO) order
  - ↳ enter at top ? removed from top
- applications include recently visited websites ? procedure calls
- realizations of stack ADT:
  - ↳ arrs
  - ↳ linked lists

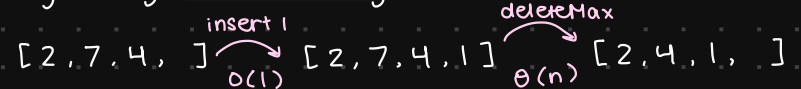
- queue is ADT consisting of collection of items w/ops:
  - ↳ enqueue: inserting item
  - ↳ dequeue: removing (? typically returning) least recently inserted item
- items are removed in first-in first-out (FIFO) order
  - ↳ enter at rear ↳ removed from front
- can have extra ops like size, isEmpty, ? front
- applications are waiting lines ? printer queues
- realizations of queue ADT:
  - ↳ circular arrs
  - ↳ linked lists

PRIORITY QUEUE ADT

- priority queue: ADT consisting of a collection of items (each having priority) w/ops.
  - ↳ insert: inserting item tagged w/ priority
  - ↳ deleteMax: removing ? returning item of highest priority
    - other names are extractMax ? getmax
- priority is aka key
- applications are to-do list, simulation systems, ? sorting
- e.g. use priority queue to sort:

```
PQ-Sort(A[0..n − 1])
1.  initialize PQ to an empty priority queue
2.  for i ← 0 to n − 1 do
3.      PQ.insert(A[i])
4.  for i ← n − 1 down to 0 do
5.      A[i] ← PQ.deleteMax()
```

  - ↳ run-time depends on PQ implementation
    - sometimes written as O(init + n insert + n · deleteMax)
- e.g. using unsorted array as realization.



  - ↳ insert: O(1)

- assume dynamic arrays (i.e. expand by doubling) so amortized over all insertions, this takes O(1) extra time:
  Suppose we want to insert 1, 2, 3, ... in arr
  → init A = [ ] (size 1)
  → if A is full, copy all its entries into arr of 2x size
  A[ ]
  ins 1: A[1] → A[1,]
  ins 2: A[1, 2] → A[1, 2, ,]
  ins 3: A[1, 2, 3,]
  Assume that copying into new arr takes time cn, n = size of A (c is constant).
  For $n = 2^k$, total time spent in copies is $c(1 + 2 + 4 + \dots + 2^k) = c(2^{k+1} - 1) = c(2n - 1)$.
  The avg (i.e. amortized) time per insert = $\frac{c(2n-1)}{n} \leq c2$ → O(1) extra time

↳ deleteMax: O(n)
↳ using unsorted linked list is identical
↳ PQ-sort w/ this realization is selection sort
  - worst-case is $O(n^2)$

- e.g. using sorted arrs as realization

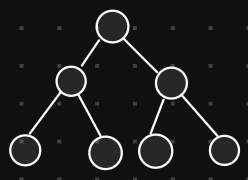
[2, 4, 7,] —insert 1, Θ(n)→ [1, 2, 4, 7] —deleteMax→ [1, 2, 4,]

↳ insert: O(n)
↳ deleteMax: O(1)
↳ using sorted linked list is identical
↳ PQ-sort is insertion sort
  - worst-case is $O(n^2)$

BINARY HEAPS

- binary heap is certain type of binary tree
  ↳ lvl of node = #edges on path from root → node
  ↳ height: max lvl of any node
  ↳ e.g.

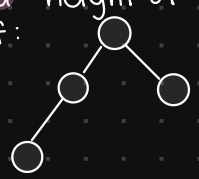
- any binary tree w/ n nodes has height at least $\log(n+1) - 1 \in \Omega(\log n)$
  ↳ proof:
  We want $n \leq$ some fcn of h → some fcn of $n \leq h$



$h = 2 \rightarrow n \leq 7$
$n \leq 1 + 2 + 4 + \dots + 2^h$
$n \leq 2^{h+1} - 1$
$n + 1 \leq 2^{h+1}$
$\log(n+1) \leq h + 1$
$\log(n+1) - 1 \leq h$

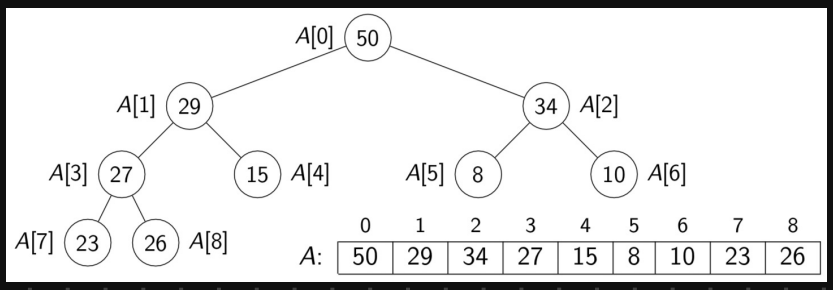
  ↳ i.e. in any binary tree w/ n nodes & height h, $\log(n+1) - 1 \leq h$

- heap is binary tree w/ 2 properties:
  ↳ structural: all lvls of heap are completely filled except for possibly last lvl
    - filled items in last lvl are left-justified
  ↳ heap-order: for any node i, key of parent of i is ≥ key of i (left & right children aren't compared)
  ↳ i.e. max-oriented binary heap
  ↳ lemma: height of heap w/ n nodes is $\Theta(\log n)$
    - proof:



$h = 2 \rightarrow n \geq 4$
In general, $n \geq 1 + 2 + 4 + \dots + 2^{h-1} + 1$ (min requirement is that h^{th} lvl has at least 1 node)
$n \geq 2^h$
$\log n \geq h$

- heaps should not be stored as binary trees
- to store heap in arr, let H be heap of n items ? let A be arr of size n ; store root in A[0] ? cont w/ elmts lvl-by-lvl from top to bottom, in each lvl left-to-right
  ↳ e.g.



- easy to navigate heap using this arr rep:
  ↳ root node is index 0
  ↳ last node is n-1
    - n is size
  ↳ left child of node i (if it exists) is $2i+1$
  ↳ right child of node i (if it exists) is $2i+2$
  ↳ parent of node i (if it exists) is $\lfloor \frac{i-1}{2} \rfloor$
- hide implementation details using helper fcns
  ↳ e.g. root(), last(), parent(i)
  ↳ some fcns may need to known but omit from code for simplicity

OPERATIONS IN BINARY HEAPS

- to insert in heap:
  1) place new key at 1st free leaf
  2) heap-order property might be violated so perform fix-up
  ↳

```
fix-up(A, i)
i: an index corresponding to a node of the heap
1.  while parent(i) exists and A[parent(i)].key < A[i].key do
2.      swap A[i] and A[parent(i)]
3.      i ← parent(i)
```

  - new item "bubbles up" until reaches correct place
  - run-time: O(height of heap) = O(logn)
- to deleteMax in heaps:
  1) max item of heap is just root node
  2) replace root by last leaf
  3) heap-order property might be violated so perform fix-down
  ↳

```
fix-down(A, i, n)
A: an array that stores a heap of size n
i: an index corresponding to a node of the heap
1.  while i is not a leaf do
2.      j ← left child of i        // Find the child with the larger key
3.      if (i has right child and A[right child of i].key > A[j].key)
4.          j ← right child of i
5.      if A[i].key ≥ A[j].key break
6.      swap A[j] and A[i]
7.      i ← j
```

  ↳ run-time: O(height of heap) = O(logn)
- priority queue realization using heaps:
  ↳ store items in arr A ? globally keep track of size
  ↳ insert ? deleteMax: O(logn) time

↳

```
insert(x)
1.  increase size
2.  ℓ ← last()   // last = A.size() - 1
3.  A[ℓ] ← x
4.  fix-up(A, ℓ)
```

```
deleteMax()
1.  ℓ ← last()
2.  swap A[root()] and A[ℓ]
3.  decrease size
4.  fix-down(A, root(), size)
5.  return A[ℓ]   // max is at A[ℓ]
```

- e.g. deleteMax()
  A[50, 48, 34, ..., 26, 15] size = 11
  A[15, 48, 34, ..., 26 | 50] size = 10
  ↑ ℓ
  ↳ 50 is not deleted from heap, but next time new elmt is added, it gets overridden

PQ-SORT AND HEAPSORT

- any PQ can be used to sort in time $O(\text{init} + n \cdot \text{insert} + n \cdot \text{deleteMax})$
- w/ bin-heaps implementation of PQs, we get sorting algorithm:

```
PQsortWithHeaps(A)
1.  initialize H to an empty heap
2.  for i ← 0 to n − 1 do
3.      H.insert(A[i])
4.  for i ← n − 1 down to 0 do
5.      A[i] ← H.deleteMax()
```

↳ both ops run in O(logn) time so PQ-Sort takes O(nlogn) time
↳ can improve this w/ Heapsort:
- heaps are built faster if input is known in advance
- use same arr for input + heap

- e.g. **Problem:** Given n items all at once (in $A[0 \cdots n-1]$) build a heap containing all of them.

↳ **Solution 1:** Start with an empty heap and insert items one at a time:

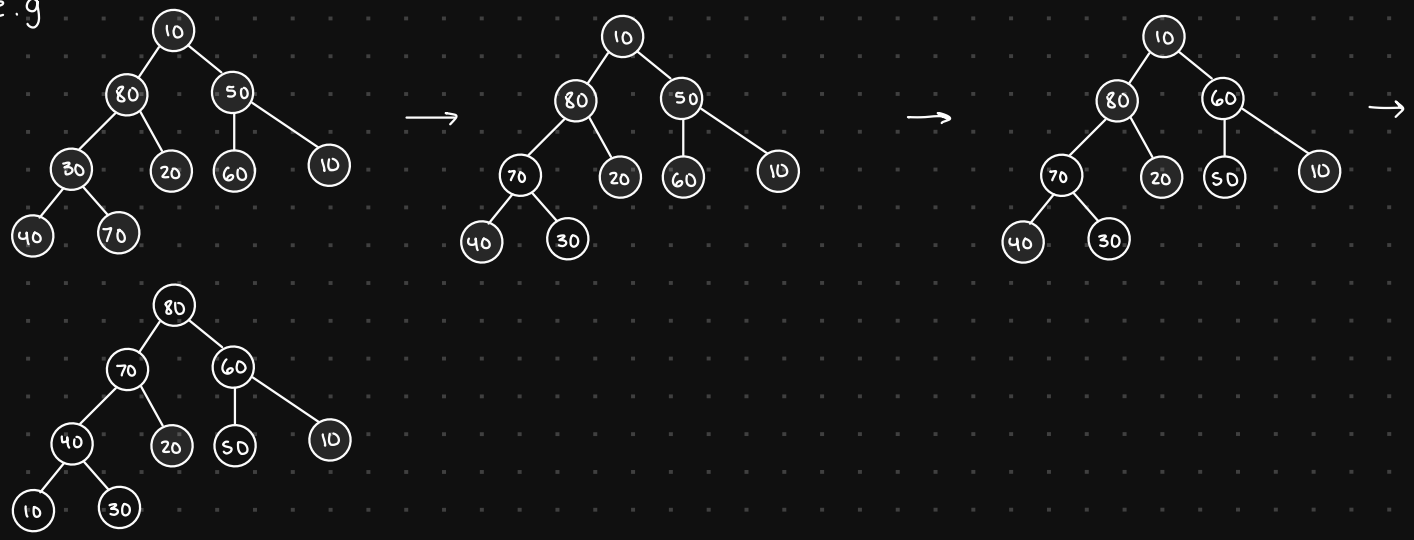
```
simpleHeapBuilding(A)
A: an array
1.  initialize H as an empty heap
2.  for i ← 0 to A.size() − 1 do
3.      H.insert(A[i])
```

- soln 1 corresponds to doing fix-ups
- worst-case run-time: $\Theta(n \log n)$ (proof in tut 3)

↳

```
heapify(A)
A: an array
1.  n ← A.size()
2.  for i ← parent(last()) downto root() do
3.      fix-down(A, i, n)
```

- soln 2 (i.e. heapify) uses fix-downs
- e.g



- worst-case run-time is $\Theta(n)$
  - → proof.

For lower bound, heapify visits all non-leaf nodes

Last leaf is $n-1$? parent of last leaf is $\lfloor \frac{n-2}{2} \rfloor = \lfloor \frac{n}{2} \rfloor - 1$

$\in \Omega(n)$

For upper bound, assume last row is complete

$n = 2^{h+1} - 1$

↑ #nodes ↖ h = height

Worst-case # key swaps $= \sum_{i=0}^{h} i\,2^{h-i}$

$= 2^h \underbrace{\sum_{i=0}^{h} \frac{i}{2^i}}_{\leq 2}$

$\in O(2^h)$

$\in O(n)$

PQ-sort w/ heaps:

```
HeapSort(A, n)
1.    // heapify
2.    n ← A.size()
3.    for i ← parent(last()) downto 0 do
4.        fix-down(A, i, n)

5.    // repeatedly find maximum
6.    while n > 1
7.        // 'delete' maximum by moving to end and decreasing n
8.        swap items at A[root()] and A[last()]
9.        decrease n
10.       fix-down(A, root(), n)
```

↳ for loop takes $\Theta(n)$ time

↳ while loop takes $O(n \log n)$ time

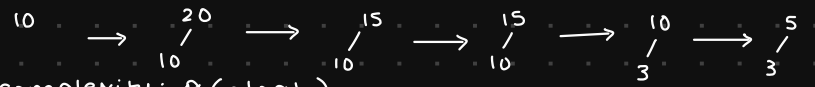
TOWARDS THE SELECTION PROBLEM

- problem: find k^{th} smallest item in arr A of n distinct #s ($k = 0, \ldots, n-1$)
  - ↳ if A was sorted:
    - $k=0$: $A[0]$ is min
    - $k=n-1$: $A[n-1]$ is max of A

- $k = \frac{n}{2}$: $A[\frac{n}{2}]$ is median of A
- soln 1: sort A, then return A[k]
  - ↳ complexity: $\Theta(n \log n)$
- soln 2: make k+1 passes thru arr, deleting min # each time
  - ↳ complexity: $\Theta(kn)$
- soln 3: scan arr & maintain k+1 smallest #s seen so far in max-heap
  - ↳ e.g.

    A = [10, 20, 15, 35, 3, 5], k = 1

    Heap contains 2 smallest elmts so far & root is 2nd smallest elmt so far

  - ↳ complexity: $\Theta(n \log k)$
    - # iterations
    - cost per iteration

soln 4: create min-heap w/ heapify(A); call deleteMin(A) k+1 times

↳ complexity: $\Theta(n + k \log n)$

- heapify
- (k+1) deleteMin

- proof that soln 4 is better than soln 3.

$n \leq n \log k$ $\quad k \log n \leq n \log k$

$$\frac{k}{\log k} \leq \frac{n}{\log n}$$

$f(k) \leq f(n)$ $\quad$ w/ $f(x) = \frac{x}{\log x}$ $\quad$ b/c f is inc fcn & $k \leq n$

SORTING AND RANDOMIZED ALGORITHMS

ANALYZING AVG-CASE RUN-TIME

- defn of avg-case run-time: $T^{avg}(n) = \frac{\sum_{I:\,size(I)=n} T(I)}{\#\text{instances of size } n} = \frac{\sum_{I \in \mathcal{I}_n} T(I)}{|\mathcal{I}_n|}$
  - ↳ $\mathcal{I}_n$ is finite
- e.g.

```
avgCaseDemo(A, n)
A: array of size n with distinct items
1.    if n ≤ 2 return
2.    if A[n−2] < A[n−1]
3.        avgCaseDemo(A[0..n/2−1], n/2)        // Good case
4.    else avgCaseDemo(A[0..n−3], n−2)         // Bad case
```

  - ↳ let T(A) be #recursions, which is asymptotically same as run-time
  - ↳ worst-case analysis: recursive call always has size n-2
    - $T^{worst}(2) = 0$

$$T^{worst}(n) = 1 + T^{worst}(n-2)$$
$$= 1 + 1 + T^{worst}(n-4)$$
$$= 1 + 1 + \dots + 1 + T^{worst}\left(n - 2\left(\tfrac{n-2}{2}\right) = 2\right) \longrightarrow \text{solve } n - 2x = 2$$
$$= \tfrac{n-2}{2} + 0 \qquad n - 2 = 2x$$
$$= \tfrac{n}{2} - 1 \qquad x = \tfrac{n-2}{2}$$

$T^{worst}(n) \in \Theta(n)$

  - ↳ best-case analysis: recursive call always has size n/2
    - $T^{best}(n) = 1 + T^{best}(\frac{n}{2})$
    - $= 1 + 1 + T^{best}(\frac{n}{4})$
    - ...
    - $= \log(n-1)$

$T^{best}(n) \in \Theta(\log n)$

- need to take avg run-time for algorithms that use arrs
  - ↳ difficult to characterize input of size n b/c there's infinitely many sets of n #s
  - ↳ assume all input #s are **distinct**
    - can be forced by using tie-breakers
  - ↳ **comparison-based** algorithms have same run-time on inputs like:
    - A = [14, 3, 2, 6, 1, 11, 7]
    - A' = [14, 4, 2, 6, 1, 12, 8]
    - acc #s don't matter b/c **relative order** is same
    - e.g.
      - A = [14, 3, 2]
      - A' = [14, 4, 2]
      - A'' = [2, 1, 0]

      } treated as same input for comparison-based algorithm

$$T^{avg}(3) = \tfrac{1}{6}(\overset{G}{T[0,1,2]} + \overset{B}{T[0,2,1]} + \overset{G}{T[1,0,2]} + \overset{B}{T[1,2,0]} + \overset{G}{T[2,0,1]} + \overset{B}{T[2,1,0]})$$

- characterize relative order via **sorting permutation** $\pi \in \Pi_n$, for which $A[\pi(0)] \le A[\pi(1)] \le \dots \le A[\pi(n-1)]$
  - ↳ e.g.
    - A = [14, 3, 2, 6, 1, 11, 7] —sorted→ [1, 2, 3, 6, 7, 11, 14]
    - $\pi = \langle 4, 2, 1, 3, 6, 5, 0 \rangle$ → A[4] A[2] A[1] A[3] A[6] A[5] A[0]
    - $\pi^{-1} = \langle 6, 2, 1, 3, 0, 5, 4 \rangle$
    - π^{-1} has same sorting permutation as A
  - ↳ assume all n! sorting permutations are equally likely

↳ $T^{avg}(n) = \frac{1}{n!} \sum_{\text{all permutations } A \text{ of } [0,\dots,n-1]} T(A)$

$= \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi)$

$T^{avg}(n) = \frac{1}{|\Pi_n|} \left(\sum_{\pi \text{ good}} T(\pi) + \sum_{\pi \text{ bad}} T(\pi) \right)$

($\frac{n!}{2}$ terms; $\frac{n!}{2}$ terms)

- recursive formula for 1 instance π:

$$T(\pi) = \begin{cases} 1 + T(\text{first } \frac{n}{2} \text{ terms}) & \text{if } \pi \text{ is good} \\ 1 + T(\text{first } n-2 \text{ terms}) & \text{if } \pi \text{ is bad} \end{cases}$$

- recursive formula for all instances π tgt:

$$\sum_{\pi \in \Pi_n} T(\pi) = \sum_{\pi \in \Pi_n : \pi \text{ good}} \left(1 + T^{avg}\left(\tfrac{n}{2}\right)\right) + \sum_{\pi \in \Pi_n : \pi \text{ bad}} \left(1 + T^{avg}(n-2)\right)$$

↳ e.g. good permutations for n = 4 (last 2 #s are in order)

[2 3 0 1]	[3 2 0 1]
[1 3 0 2]	[3 1 0 2]
[1 2 0 3]	[2 1 0 3]
[0 3 1 2]	[3 0 1 2]
[0 2 1 3]	[2 0 1 3]
[0 1 2 3]	[1 0 2 3]
run-time: $1 + T([0\ 1])$	$1 + T([1\ 0])$
↑ 1st half is in order	↑ 1st half is in reverse order

$\sum_{\pi \text{ good}} T(\pi) = 12 + 6T([0,1]) + 6T([1,0])$

$= 12\left(1 + \frac{T([0,1]) + T([1,0])}{2}\right)$

$= 12(1 + T^{avg}(2))$

$= (\#\text{good})(1 + T^{avg}(\frac{n}{2}))$ (12 good; 2)

- $\sum_{\pi \text{good}} T(\pi) = \frac{n!}{2}\left(1 + T^{avg}\left(\frac{n}{2}\right)\right)$
- $\sum_{\pi \text{bad}} T(\pi) = \frac{n!}{2}\left(1 + T^{avg}(n-2)\right)$

(half of cases are good; half of cases are bad)

$\frac{1}{n!} \sum_{\pi} T(\pi) = \frac{1}{n!}\left(\frac{n!}{2}\left(1 + T^{avg}\left(\frac{n}{2}\right)\right) + \frac{n!}{2}\left(1 + T^{avg}(n-2)\right)\right)$

$T^{avg}(n) = 1 + \frac{1}{2}T^{avg}\left(\frac{n}{2}\right) + \frac{1}{2}T^{avg}(n-2)$

- claim: $T^{avg}(n) \leq 2\log n$

↳ proof:

$T^{avg}(n) = 1 + \frac{1}{2}T^{avg}\left(\frac{n}{2}\right) + \frac{1}{2}T^{avg}(n-2)$

$T^{avg}(1) = T^{avg}(2) = 0$

By induction, OK for $n \leq 2$. Assume OK for all ints less than n. Prove for n:

$T^{avg}(n) = 1 + \frac{1}{2}T^{avg}\left(\frac{n}{2}\right) + \frac{1}{2}T^{avg}(n-2)$

$\leq 1 + \frac{1}{2}\left(2\log\left(\frac{n}{2}\right)\right) + \frac{1}{2}(2\log(n-2))$ (induction assumption; $n-2 < n$)

$\leq 1 + \log\left(\frac{n}{2}\right) + \log n$

$\leq \log 2 + \log n - \log 2 + \log n$

$\leq 2\log n$

↳ avgCaseDemo has avg-case run-time $O(\log n)$, compared to $\Theta(n)$ worst-case time

RANDOMIZED ALGORITHMS

- if algorithm has better avg-case time than worst-case time, randomization is often good idea
- randomized algorithm relies on some random #s in addition to input
  - ↳ run-time depends on input & random #s
  - ↳ generation of random #s always has uniform distribution
- goal is to shift dependency of run-time from what we can't control (input) to what we can (random #s)
  - ↳ no more bad instances, just unlucky #s
- T^{avg} is for avg-case analysis & T^{exp} uses randomized algos

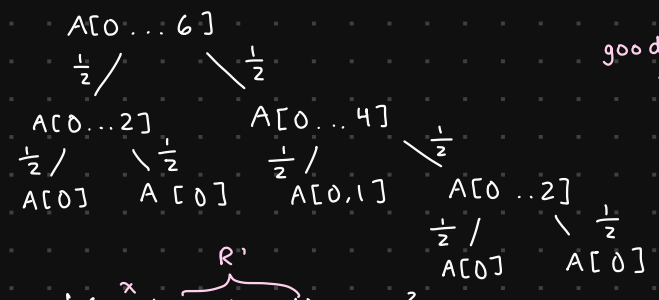
· e.g.

```
expectedDemo(A, n)
A: array of size n with distinct items
1.   if n ≤ 2 return
2.   if random(2) swap A[n−1] and A[n−2]
3.   if A[n−2] ≤ A[n−1]
4.       expectedDemo(A[0..n/2−1], n/2)      // Good case
5.   else expectedDemo(A[0..n−3], n−2)       // Bad case
```

random(2) = 1 → swap A[n-1] ? A[n-2]
= 0 → don't swap

input A \ random	0	1
A[n-2] < A[n-1]	good	bad
A[n-2] > A[n-1]	bad	good

· define T(I, R) to be run-time of randomized algo $\mathcal{A}$ for instance I & seq of random choices R
· expected run-time $T^{exp}(I)$ for instance I is expected val: $T^{exp}(I) = E[T(I,R)] = \sum_R T(I,R) \cdot Pr(R)$ (probability ↓)
↳ take max over all instances of size n to define expected run-time of $\mathcal{A}$:
$T^{exp}(n) := \max_{I \in \mathcal{I}_n} T^{exp}(I)$
∘ $\mathcal{I}_n$ is set of all possible inputs of size n
↳ can still have good/bad luck so occasionally discuss worst that can happen (i.e. $\max_I \max_R T(I,R)$)
· e.g. calc $T^{exp}(n)$ for expectedDemo
random choices $R = \langle x, x', x'', \ldots \rangle = \langle x, R' \rangle$ (← rest of sequence x', x'', ...)
↳



∘ Re { (bad, bad, bad), .. } (x, R')
↳ $T(A,R) = \begin{cases} 1 + T^{exp}(A[0 \ldots \frac{n}{2}-1], R') & \text{if } x = \text{good} \\ 1 + T^{exp}(A[0 \ldots n-3], R') & \text{if } x = \text{bad} \end{cases}$
↳ take A st $T^{exp}(A) = T^{exp}(n)$
$T^{exp}(A) = \sum_R P(R) \cdot T(A,R)$

$$= \underbrace{\frac{1}{2}}_{\text{x good}} \sum_{R'} P(R')\left(T(A[0..\tfrac{n}{2}-1], R') + 1\right) + \underbrace{\frac{1}{2}}_{\text{x bad}} \sum_{R'} P(R')\left(T(A[0..n-3], R') + 1\right)$$

$$\leq \frac{1}{2}\left(\underbrace{T^{exp}(\tfrac{n}{2})}_{\text{best case}} + 1\right) + \frac{1}{2}\left(\underbrace{T^{exp}(n-2)}_{\text{worst case}} + 1\right)$$

↳ in summary, $T^{exp}(n) \leq \frac{1}{2}\left(T^{exp}(\frac{n}{2}) + 1\right) + \frac{1}{2}\left(T^{exp}(n-2) + 1\right)$
$T^{exp}(n) \leq 1 + \frac{1}{2}T^{exp}(\frac{n}{2}) + \frac{1}{2}T^{exp}(n-2)$
∘ recurrence rlth

QUICKSELECT

· selection problem: given arr A of n #s ? $0 \leq k < n$, find elmt that would be at pos k of sorted arr
↳

0	1	2	3	4	5	6	7	8	9
30	60	10	0	50	80	90	10	40	70

select(3) should return 30.

↳ special case is **median finding** = selection w/ $k = \lfloor \frac{n}{2} \rfloor$
↳ selection done w/ heaps in time $\Theta(n + k\log n)$; median finding is $\Theta(n\log n)$

- **QuickSelect** algo can do selection in linear time
- QuickSelect ? QuickSort algos rely on 2 subroutines
  ↳ **choose-pivot(A)**: return index p in A
    - use **pivot-value** $v \leftarrow A[p]$ to rearrange A
      → select rightmost elmt for pivot-val
  ↳ **partition(A, p)**: rearrange A ? return pivot-index i st:
    - v is in A[i]
    - all items in A[0, ..., i-1] are $\leq v$
    - all items in A[i+1, ..., n-1] are $\geq v$
  ↳ e.g. p = n-i, v = A[n-i] = 70

$i = 7$

A = [30, 60, 10, 0, 50, 10, 40, 70, 80, 90]

- linear-time implementation of partition:

partition(A, p)
A: array of size n, p: integer s.t. $0 \leq p < n$
1. Create empty lists *smaller*, *equal* and *larger*.
2. $v \leftarrow A[p]$
3. **for** each element x in A
4. **if** $x < v$ **then** *smaller.append*(x)
5. **else if** $x > v$ **then** *larger.append*(x)
6. **else** *equal.append*(x).
7. $i \leftarrow$ *smaller.size*
8. $j \leftarrow$ *equal.size*
9. Overwrite $A[0 \ldots i-1]$ by elements in *smaller*
10. Overwrite $A[i \ldots i+j-1]$ by elements in *equal*
11. Overwrite $A[i+j \ldots n-1]$ by elements in *larger*
12. return i

- QuickSelect algo:

QuickSelect(A, k)
A: array of size n, k: integer s.t. $0 \leq k < n$
1. $p \leftarrow$ *choose-pivot*(A)
2. $i \leftarrow$ *partition*(A, p)
3. **if** $i = k$ **then**
4. **return** $A[i]$
5. **else if** $i > k$ **then**
6. **return** *QuickSelect*$(A[0, 1, \ldots, i-1], k)$
7. **else if** $i < k$ **then**
8. **return** *QuickSelect*$(A[i+1, i+2, \ldots, n-1], k-(i+1))$

Idea: After partition have

$\leq v$	v	$\geq v$
	i	

- **analysis** of QuickSelect: let $T(A, k)$ be #key-comparisons in size-n arr A w/ param k (which is asymptotically same as run-time)
  ↳ written $T(\pi, k)$ for any arr w/ sorting permutation π
  ↳ partition uses n key-comparisons

↳ **worst-case** analysis: pivot-index is last ⇒ $k = 0$
- $T^{worst}(n, 0) \geq n + (n-1) + (n-2) + \ldots + 1 \in \Theta(n^2)$

↳ **best-case** analysis: 1st chosen pivot is k^{th} elmt so there's no recursive calls
- $T^{best}(n, k) = n \in \Theta(n)$

↳ **avg-case** analysis: use sorting permutations (i.e. $T^{avg}(n) = \frac{1}{n!} \sum_{\pi \in \Pi_n} \max_k T(\pi, k)$

Assume sorting permutation π gives pivot-index i. If new arr after partition is A', then

$T(\pi, k) \leq n + \max\{T(A'[0..i-1], k), T(A'[i+1..n-1], k-i-1)\}$

↑ partition; ↑ 1st half of arr befor pivot-index i (size i); ↑ 2nd half of arr after pivot-index i (size $n-i-1$)

Claim: $\sum_{\pi \in \Pi_n} T(\pi) \leq \sum_{\pi \in \Pi_n} (n + \max\{T^{avg}(i), T^{avg}(n-i-1)\})$

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

Assume we know $\sum_{i=0}^{n-1} \max(i, n-i-1) \leq \frac{3}{4}n^2$ ← hint will be given

$T^{avg}(0) = T^{avg}(1) = 0$

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

Suppose $T^{avg}(j) \leq 4j \ \forall j = 0, \ldots, n-1$. We prove $T^{avg}(n) \leq 4n$.

By induction:

$T^{avg}(n) \leq n + \frac{1}{n} \underbrace{\sum_{i=0}^{n-1} 4\max(i, n-i-1)}_{\text{induction assumption since } i=0,\ldots,n-1}$

$\leq n + \frac{1}{n} \cdot 4\left(\frac{3}{4}n^2\right)$ using hint

$\leq n + 3n$

$\leq 4n$

RANDOMIZING QUICKSELECT

- 1st idea: randomly permute (i.e. order) input using shuffle

↳

```
shuffle(A)
A: array of size n
1.    for i ← 1 to n−1 do
2.        swap( A[i], A[random(i+1)] )
```

↳ works well but can do it directly within routine

- 2nd idea: change pivot selection (preferred)

↳

```
RandomizedQuickSelect(A, k)
1.    ...
2.    p ← random(A.size)
3.    i ← partition(A, p)
4.    ...
```

- Pr(pivot has index i) $= \frac{1}{n}$
- assume that first random gives pivot-index i
  - ↳ recurse in arr of size i or $n-i-1$ (or not at all)
  - ↳ if new arr after partition is A' & $R = \langle i, R' \rangle$, then:

$$T(\pi, k, \langle i, R' \rangle) \leq n + \begin{cases} T(A'[0..i-1], k, R') & \text{if } i > k \\ T(A'[i+1..n-1], k-i-1, R') & \text{if } i < k \\ 0 & \text{otherwise} \end{cases}$$

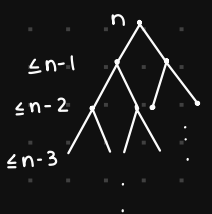
- recurrence: $T^{exp}(n) \leq n + \frac{1}{n} \sum_{i=0}^{n-1} \max\{T^{exp}(i), T^{exp}(n-i-1)\}$
  - ↳ $T^{exp}(n) \in O(n)$ & $\Theta(n)$

QUICKSORT

- Hoare developed partition & QuickSelect in 1960
  - ↳ also developed QuickSort:

```
QuickSort(A)
A: array of size n
1.  if n ≤ 1 then return
2.  p ← choose-pivot(A)
3.  i ← partition(A, p)
4.  QuickSort(A[0, 1, ..., i−1])
5.  QuickSort(A[i+1, ..., n−1])
```

- **analysis** of QuickSort: set $T(A)$ = # key-comparisons for QuickSort in size-n arr A
  - ↳ worst-case: recursive call could always have size $n-1$



≤ 2

≤ 1

- $T^{worst}(n) \geq n + T^{worst}(n-1) \in \Omega(n^2)$
  - → acc $\Theta(n^2)$ b/c depth of recursion is at most n
  - → take A in inc order, size n

$$T(A) = n + T(A[0, \ldots, n-2])$$
$$= n + n-1 + T(A[0, \ldots, n-3])$$
$$\in \Theta(n^2)$$

$T^{worst}(n) \geq T(A)$ so $T^{worst}(n) \in \Omega(n^2)$

- ↳ best-case analysis: if pivot-index is always in middle, then we recurse in 2 sub-arrs of size $\leq \frac{n}{2}$
  - $T^{best}(n) \leq n + 2T^{best}(\frac{n}{2}) \in O(n \log n)$
    - → same as MergeSort
    - → can be shown as $\Theta(n \log n)$

- **avg-case analysis** of QuickSort:

Let $T^{avg}(n)$ be avg-case #comparisons for QuickSort in size-n arr

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

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

} difficult to prove

- ↳ $T^{avg}(n) \leq 2n \log(n) \in O(n \log n)$
  - proven by induction on n

- to **randomize** QuickSort:

```
RandomizedQuickSort(A)
1.  ...
2.  p ← random(A.size)
3.  i ← partition(A, p)
4.  ...
```

- ↳ Pr(pivot has index i) $= \frac{1}{n}$
- ↳ assuming we know pivot index is i, recurse in 2 arrs of size i & $n-i-1$
  - $T^{exp}(n) \leq n + \frac{2}{n} \sum_{i=0}^{n-1} T^{exp}(i) \in O(n \log n)$

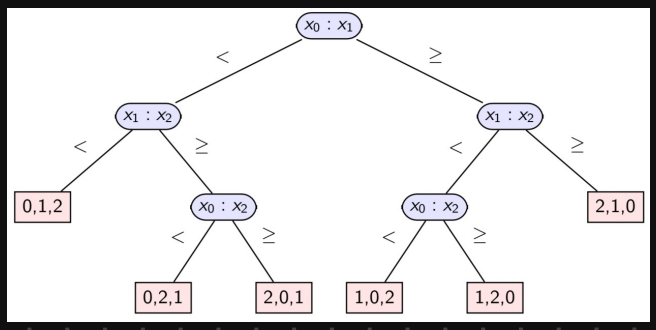
- for avg-case & expected analyses, don't need to know recurrence rlts & prove run-times
  - ↳ simply know resulting run-time bounds

LOWER BOUND FOR COMPARISON-BASED SORTING

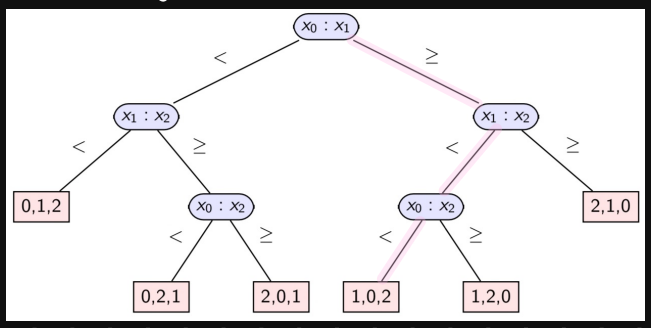
- summary of sorting algos so far:

Sort	Running time	Analysis
Selection Sort	$\Theta(n^2)$	worst-case
Insertion Sort	$\Theta(n^2)$	worst-case
Merge Sort	$\Theta(n \log n)$	worst-case
Heap Sort	$\Theta(n \log n)$	worst-case
QuickSort	$\Theta(n \log n)$	average-case
RandomizedQuickSort	$\Theta(n \log n)$	expected

- can we do better than $\Theta(n \log n)$ run-time?
  - ↳ no for comparison-based sorting
    - lower bound is $\Omega(n \log n)$
  - ↳ yes for non-comparison-based sorting
    - can achieve $O(n)$
- in **comparison model**, data accessed in 2 ways:
  - ↳ comparing 2 elmts
  - ↳ moving elmts around (e.g. copying, swapping)
- comparison-based algos can be expressed as **decision tree**
  - ↳ to sort $\{x_0, x_1, x_2\}$:



  - ↳ e.g. sorting $\{4, 2, 7\}$:



- **theorem**: any correct comparison-based sorting algo requires at least $\Omega(n \log n)$ comparison ops to sort n distinct items
  - ↳ Claim: for $h \geq 0$ & any bin tree T, there's at most 2^h leaves of lvl $\leq h$ in T (can use as fact)

Sketch of proof: induction on h

← leaves are leaves of either left or right subtree (if T is just root)

There are $n!$ inputs so there's $n!$ reachable leaves.

h = lvl of deepest leaf

h = worst case #comparisons in size n

$n! \leq 2^h$ claim

$\log(n!) \leq h$

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

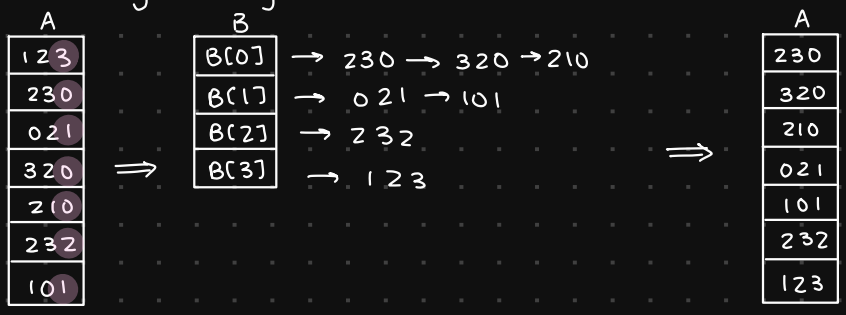
NON-COMPARISON-BASED SORTING

- assume keys are #s in base R (R: radix)
  - ↳ R = 2, 10, 128, 256 are most common
- assume all keys have same # m of digits
  - ↳ can achieve after padding w/ leading 0s
- e.g. single-digit bucket sort

  R = 4

  A = [123, 230, 021, 320, 210, 232, 101]

  Sort by last digit:



- pseudocode for bucket sort:

Bucket-sort(A, d)
A: array of size n, contains numbers with digits in $\{0, \ldots, R-1\}$
d: index of digit by which we wish to sort

1. Initialize an array $B[0 \ldots R-1]$ of empty lists (**buckets**)
2. **for** $i \leftarrow 0$ to $n-1$ **do**
3. Append $A[i]$ at end of $B[d^{\text{th}}$ digit of $A[i]]$
4. $i \leftarrow 0$
5. **for** $j \leftarrow 0$ to $R-1$ **do**
6. **while** $B[j]$ is non-empty **do**
7. move first element of $B[j]$ to $A[i{+}{+}]$

  - ↳ run-time is $\Theta(R+n)$
    - 1) $\Theta(R)$
    - 2-3) $\Theta(n)$
    - 5-7) $O(Rn)$

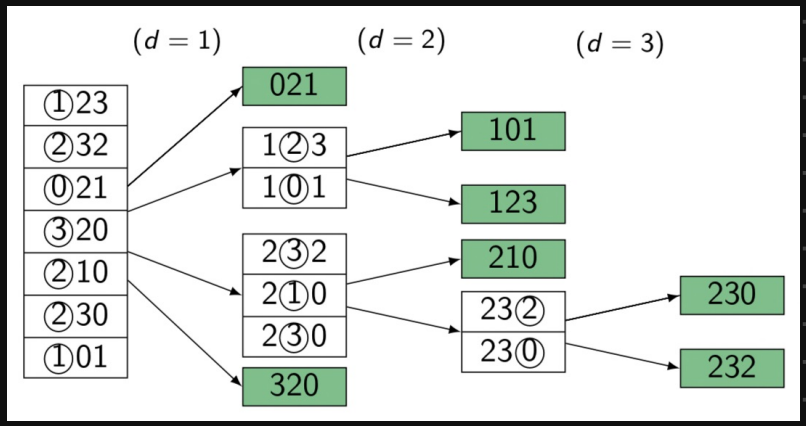
↑ loop ↖ at most n entries in B[j]

$$\Theta\left(\sum_{j}^{R}(1+n_j)\right) = \sum_{j}^{R} 1 + \sum_{j} n_j$$

j: j=1, j<R; n_j: #elmts in B[j]; $\sum_j^R 1$: R; $\sum_j n_j$: all elmts in subarrs add up to n

$$= \Theta(R+n)$$

  - ↳ auxiliary space is $\Theta(R+n)$
  - ↳ this is stable, meaning equal items stay in og order
  - ↳ possible to replace lists by 2 auxiliary arrs of size R & n (i.e. count-sort)
- MSD-Radix-Sort sorts arr of m-digit radix-R #s recursively: sort by leading digit, then each group by next digit, & so on
  - ↳ e.g.



↳ pseudocode:

```
MSD-Radix-sort(A,  ℓ ← 0,  r ← n−1,  d ← index of leading digit)
ℓ, r: range of what we sort, 0 ≤ ℓ, r ≤ n−1
1.   if ℓ < r
2.       bucket-sort(A[ℓ..r], d)
3.       if there are digits left    // recurse in sub-arrays
4.           ℓ' ← ℓ
5.           while (ℓ' < r) do
6.               Let r' ≥ ℓ' be maximal s.t. A[ℓ'..r'] all have same dth digit
7.               MSD-Radix-sort(A, ℓ', r', d+1)
8.               ℓ' ← r' + 1
```

- drawback of MSD-Radix-Sort is that there's many recursions
- auxiliary space is $\Theta(n+R+m)$
  ↳ for bucket-sort & recursion stack
- run-time is $\Theta(mnR)$ b/c may have $\Theta(mn)$ subproblems
  ↳ more detail in tut

LSD-Radix-Sort is same as MSD-Radix-Sort, but sorts by least significant digit

↳ e.g.

12③		2③0		①01		021
23⓪		3②0		②10		101
02①	$(d=3)$	2①0	$(d=2)$	③20	$(d=1)$	123
32⓪	$\Longrightarrow$	0②1	$\Longrightarrow$	⓪21	$\Longrightarrow$	210
21⓪		1⓪1		①23		230
23②		2③2		②30		232
10①		1②3		②32		320

↳ pseudocode:

```
LSD-radix-sort(A)
A: array of size n, contains m-digit radix-R numbers
1.   for d ← least significant to most significant digit do
2.       Bucket-sort(A, d)
```

- run-time is $\Theta(m(n+R))$
- auxiliary space is $\Theta(n+R)$
- loop invariant: A is sorted wrt digits d, .., m of each entry
  → proof:
  Suppose that after doing d=2, A is sorted wrt digits 2, ..., m
  We do bucket-sort at index 1.
  1) bucket-sort sorts:
  All 0___ come before 1___
  All 1___ come before 2___
  etc.
  2) bucket-sort is stable:
  All 0___ are sorted b/c they were sorted before bucket-sort by assumption
  etc.

DICTIONARIES

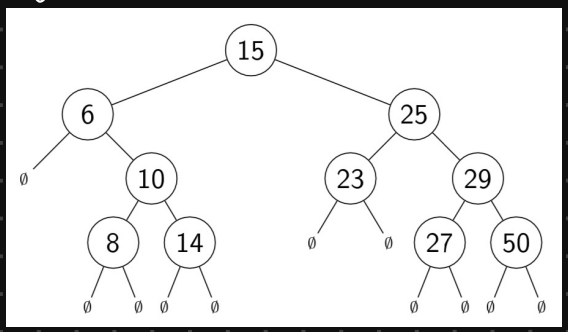
ADT DICTIONARY

- **dictionary**: ADT consisting of collection of items, each of which contains a key & some data (val), which is called **key-value pair (KVP)**
  - ↳ keys can be compared & are typically unique
- operations:
  - ↳ **search(k)**
    - aka findElement(k)
  - ↳ **insert(k, v)**
    - aka insertItem(k, v)
  - ↳ **delete(k)**
    - aka removeElement(k)
  - ↳ optional: closestKeyBefore, join, isEmpty, size
- **assumptions**:
  - ↳ dict has n KVPs
  - ↳ each KVP uses constant space
    - if not, val could be pointer
  - ↳ keys can be compared in constant time
- using **unordered arr / linked list**:
  - ↳ search is $\Theta(n)$
  - ↳ insert $\Theta(1)$
    - except arr occasionally needs to resize
  - ↳ delete is $\Theta(n)$
    - need to search
- using ordered arr:
  - ↳ search is $\Theta(\log n)$
    - via bin search
  - ↳ insert is $\Theta(n)$
  - ↳ delete is $\Theta(n)$

BINARY SEARCH TREES

structure is all nodes have 2 (possibly empty) subtrees
- ↳ every node stores KVP
- ↳ empty subtrees usually not shown

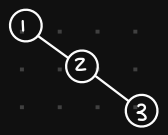
- **ordering**:
  - ↳ every key k in T.left is < root key
  - ↳ every key k in T.right > root key
- e.g.



- **BST::search(k)**: start at root, compare k to current node's key, stop if found or subtree is empty, else recurse at subtree
- **BST::insert(k, v)**: search for k, then insert (k, v) as new node
- **BST::delete(k)**:
  - ↳ search for node x that contains key

↳ if x is leaf (i.e. both subtrees empty), delete it
↳ if x has 1 non-empty subtree, move child up
↳ else (x has 2 children), swap x w/ key at successor / predecessor node & delete that node
- for **succesor**, go right once, then left until not possible
- for **predecessor**, go left once, then right until not possible

- search, insert, & delete all have cost $\Theta(h)$
↳ h = height of tree = max path length from root to leaf
↳ if n items are inserted one at a time, size of h is:
- worst-case: $n-1 = \Theta(n)$
→ e.g.



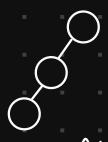
$n = 3$
$h = 2 = n-1$

- best-case: $\Theta(\log n)$
→ any bin tree w/n nodes that $h \geq \log(n+1) - 1$
- avg-case: $\Theta(\log n)$
→ difficult to prove

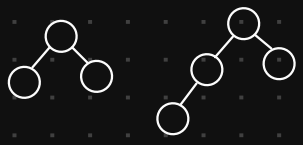
AVL TREES

- **AVL tree** is BST w/ additional **height-balance property** at every node
↳ heights of left & right subtree differ by at most 1
- i.e. if node v has left subtree L & right subtree R, then **balance(v) := height(R) - height(L)** must be in {-1, 0, 1}
→ balance(v) = -1 means v is **left-heavy**
→ balance(v) = 1 means v is **right-heavy**

↳ height of empty tree is -1
↳ e.g. not AVL tree

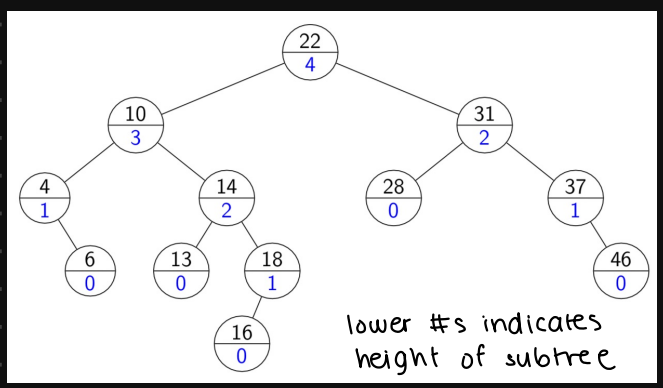


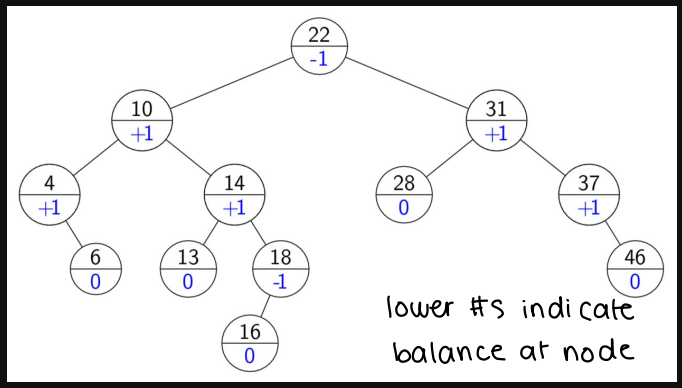
↳ e.g. AVL trees



↳ need to store at each node v height of subtree rooted at it
- suffices to store balance(v) instead

↳ e.g.



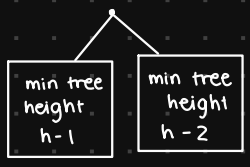


- **theorem**: AVL tree on n nodes has $\Theta(\log n)$ height
↳ proof.
$h \leq \log_c(n)$ ∃ constant c

$c^h \leq n$

Define $N(h)$ to be least #nodes in AVL tree of height h. We want $N(h) \geq c^h$

h	-1	0	1	2	3
$N(h)$	0	1	2	4	7
tree	$\emptyset$	•			



$N(h) = 1 + N(h-1) + N(h-2)$
(root, left, right)

$N(h) \geq 0 + N(h-2) + N(h-2)$

$N(h) \geq 2N(h-2)$

$\geq 2^2 N(h-2 \cdot 2)$

$\geq 2^3 N(h-3 \cdot 2)$

$\geq 2^4 N(h-4 \cdot 2)$

$\vdots$ repeat $\frac{h}{2}$ times to get $N(h=0)$

$\geq 2^{\frac{h}{2}} N(h - \frac{h}{2} \cdot 2)$

$N(h) \geq 2^{\frac{h}{2}} N(0)$

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

Thus, we set $c = \sqrt{2}$ and prove $c^h \leq n$.

↳ search, insert, & delete all cost $\Theta(\log n)$ in worst case

INSERTION IN AVL TREES

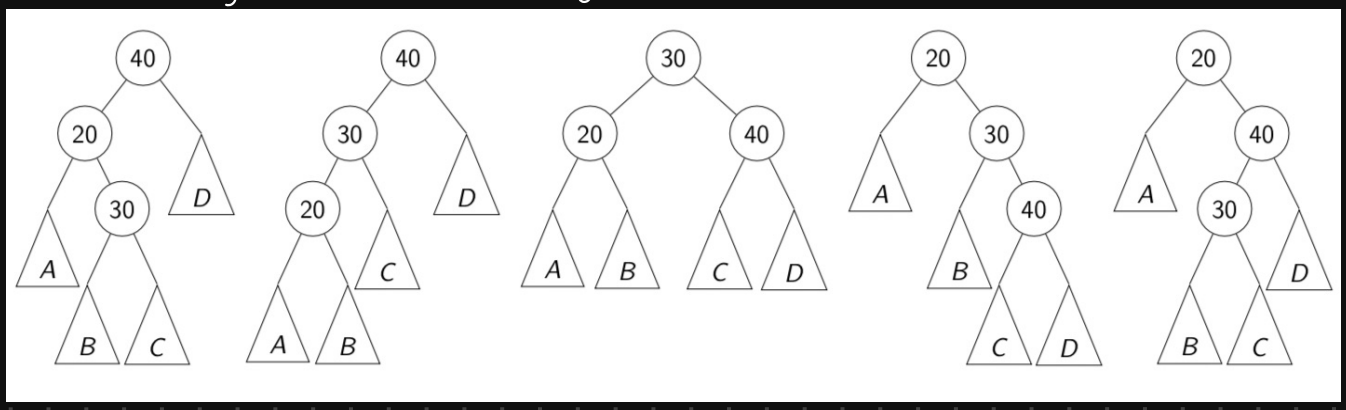
- to perform `AVL::insert(k,v)`:
  - ↳ insert (k,v) w/usual BST insertion
  - ↳ assume this returns new leaf z where key is stored
  - ↳ move up tree from z, updating heights
    - assume we only have parent-links & not full path to z
  - ↳ if height diff becomes ±2 at node z, then z is unbalanced
    - restructure tree
- pseudocode:

```
AVL::insert(k, v)
1.    z ← BST::insert(k, v)   // leaf where k is now stored
2.    while (z is not NIL)
3.        if (|z.left.height − z.right.height| > 1) then
4.            Let y be taller child of z
5.            Let x be taller child of y (break ties)
6.            z ← restructure(x, y, z) // see later
7.            break          // can argue that we are done
8.        setHeightFromSubtrees(z)
9.        z ← z.parent
```

```
setHeightFromSubtrees(u)
1.    u.height ← 1 + max{u.left.height, u.right.height}
```

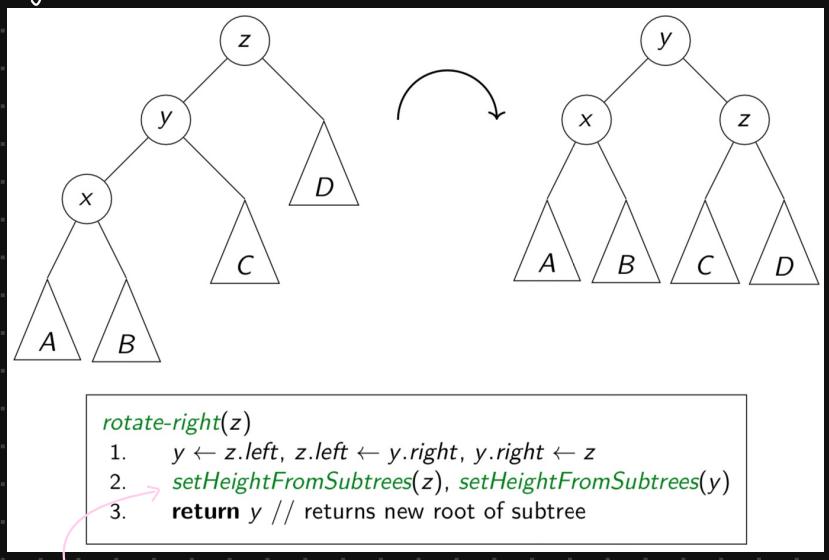
RESTORING AVL PROPERTY: ROTATIONS

- there are many diff. BSTs w/ same keys:



↳ goal is to change structure among 3 nodes w/o changing order ∴ st. subtree becomes balanced

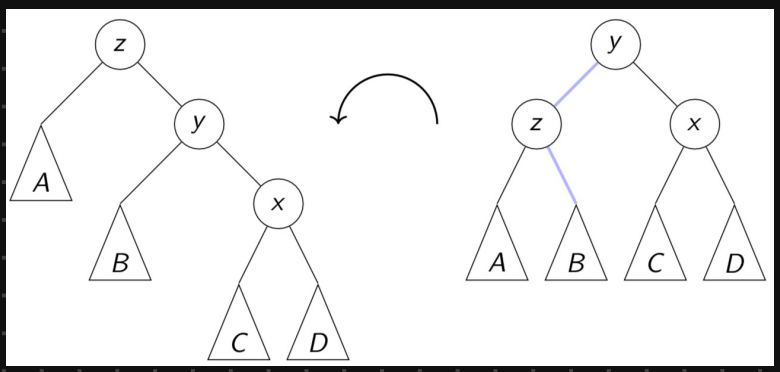
- right rotation on node z:




```
rotate-right(z)
1.  y ← z.left, z.left ← y.right, y.right ← z
2.  setHeightFromSubtrees(z), setHeightFromSubtrees(y)
3.  return y // returns new root of subtree
```

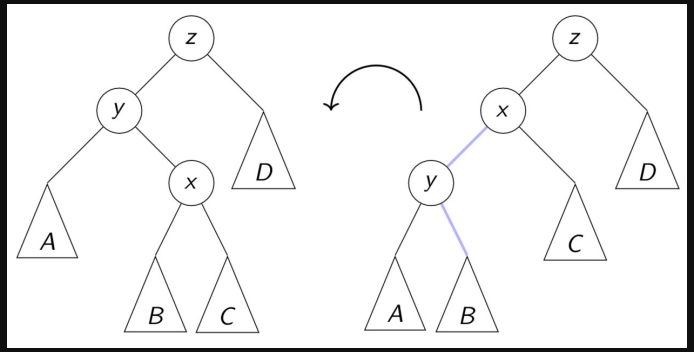
swap parent(y) ∧ parent(z)

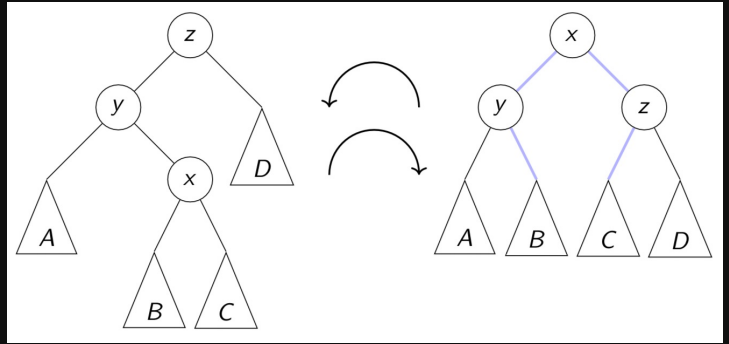
- left rotation on node z:



↳ useful to fix right-heavy imbalance

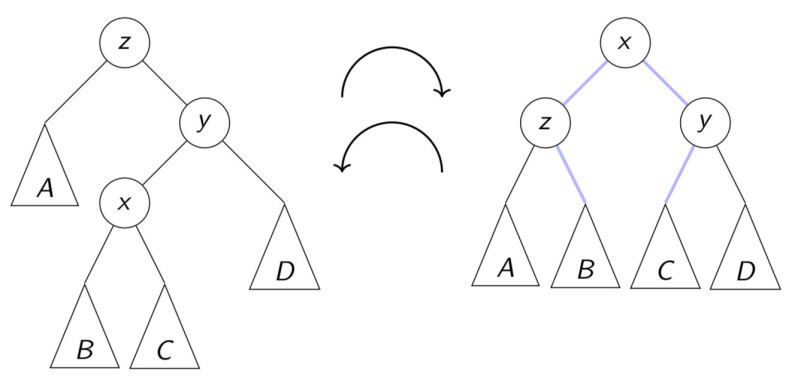
- double right rotation on node z:





↳ first, left rotation at y
↳ next, right rotation at z

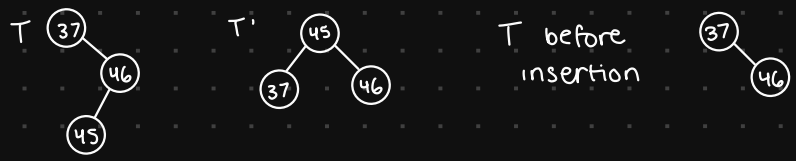
- **double left rotation** on node z:



- **restructure pseudocode**:

```
restructure(x, y, z)
node x has parent y and grandparent z
1.   case
       :  // Right rotation
          return rotate-right(z)
       :  // Double-right rotation
          z.left ← rotate-left(y)
          return rotate-right(z)
       :  // Double-left rotation
          z.right ← rotate-right(y)
          return rotate-left(z)
       :  // Left rotation
          return rotate-left(z)
```

↳ ties (i.e. one of nodes have children w/ same heights) must be broken to prefer single rotation
↳ rule is that **middle key** (i.e. node w/ middle val) becomes new root
↳ proof that restructuring at most 1 node is enough to guarantee entire AVL tree is fixed
Let T be subtree of of z. Let T' be subtree of restructure.



T' is AVL tree.
- height of T' = height of T before insertion

- **remove key k** w/ BST::delete
↳

```
AVL::delete(k)
1.    z ← BST::delete(k)
2.    // Assume z is the parent of the BST node that was removed
3.    while (z is not NIL)
4.        if (|z.left.height − z.right.height| > 1) then
5.            Let y be taller child of z
6.            Let x be taller child of y (break ties to prefer single rotation)
7.            z ← restructure(x, y, z)
8.        // Always continue up the path and fix if needed.
9.        setHeightFromSubtrees(z)
10.       z ← z.parent
```

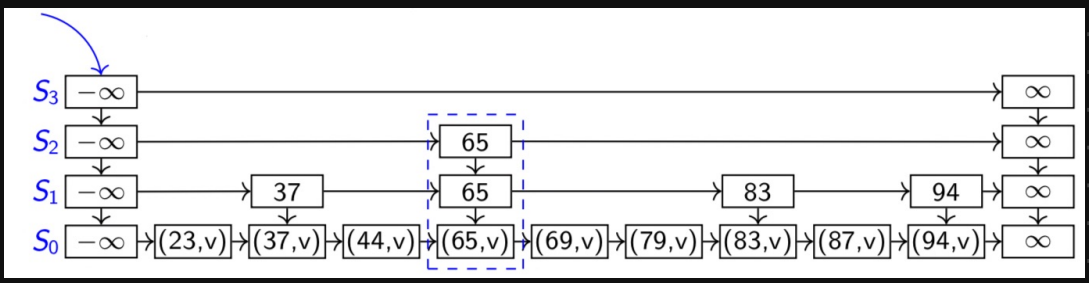
↳ find node where structural change happened
  - not necessarily near node that had k
↳ go back up to root, update heights, ? rotate if needed
↳ ties must be broken to prefer single rotation or resulting tree might not be AVL tree

OTHER DICTIONARY IMPLEMENTATIONS

- realizations of dictionaries so far:
  - ↳ unordered arr / linked list
    - $\Theta(1)$ insert, $\Theta(n)$ search & delete
  - ↳ ordered arr
    - $\Theta(\log n)$ search, $\Theta(n)$ insert & delete
  - ↳ BST
    - $\Theta(\text{height})$ search, insert, & delete
  - ↳ balanced BST (e.g. AVL trees)
    - $\Theta(\log n)$ search, insert, & delete

SKIP LISTS

- skip lists have a hierarchy S of ordered linked lists (lvls) $S_0, S_1, \ldots, S_n$
  - ↳ each list S_i contains special keys $-\infty$ & $+\infty$ (i.e. sentinels)
  - ↳ list S_0 contains KVPs of S in inc order
    - other lists store only keys or links to nodes in S_0
  - ↳ each list is subseq of previous one
    - i.e. $S_0 \supseteq S_1 \supseteq \ldots \supseteq S_n$
  - ↳ list S_n contains only sentinels
    - left sentinel is root
- structure:

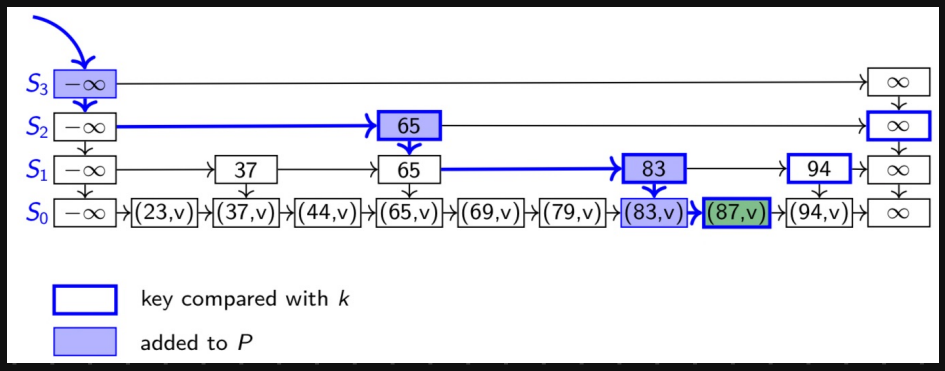


  - ↳ each KVP belongs to tower of nodes
  - ↳ usually more nodes than keys
  - ↳ skip list consists of ref to topmost left node
  - ↳ each node p has ref p.after & p.below
- to search in skip lists, find predecessor for each lvl (i.e. node before where k would be)
  - ↳ pseudocode:

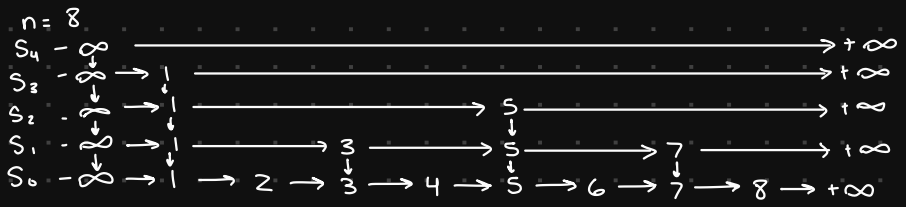
```
getPredecessors (k)
1.   p ← root
2.   P ← stack of nodes, initially containing p
3.   while p.below ≠ NIL do
4.        p ← p.below
5.        while p.after.key < k do p ← p.after
6.        P.push(p)
7.   return P
```

```
skipList::search (k)
1.   P ← getPredecessors(k)
2.   p0 ← P.top() // predecessor of k in S0
3.   if p0.after.key = k    return p0.after
4.   else return "not found, but would be after p0"
```

↳ e.g. search(87)



- ideal skip list: #nodes is halved every lvl



↳ # nodes = 8 + 4 + 2 + 1
= 15
= 2 8 - 1
= 2n - 1

↳ height = 4
= $\log_2 n$ + 1

↳ cost of search : O(log n)
- do at most 1 step to the right

- skipList : insert(k, v)

↳ randomly toss coin until we get tails
- -ve binomial distr

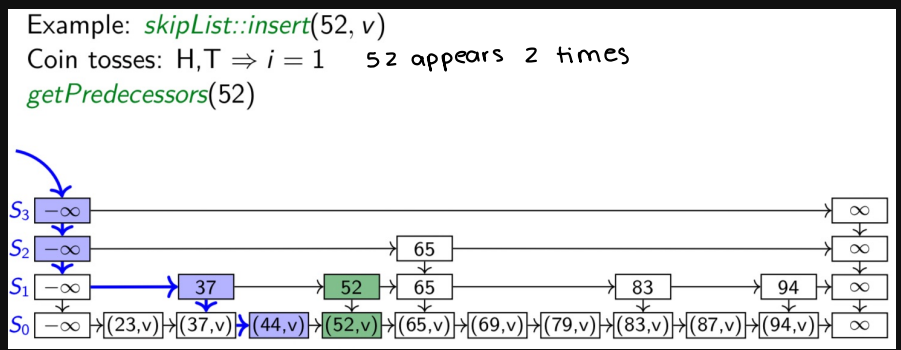
↳ let i be #times coin came up heads
- want k to be in lists $S_0, \ldots, S_i$ so it gets add i + 1 times
- i is height of tower of k
- P(tower of key has height ≥ i) = $(\frac{1}{2})^i$

↳ inc h of skip list if needed to have h > i lvls

↳ use getPredecessors(k) to get stack P
- top i+1 items of P are predecessors of $p_0, p_1, \ldots, p_i$ of where k should be in list $S_0, S_1, \ldots, S_i$

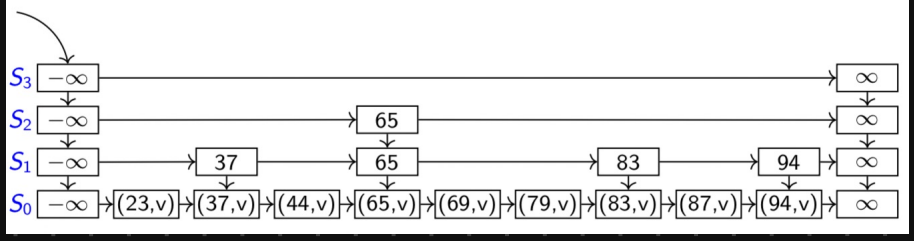
↳ insert (k, v) after p_0 in S_0 & k after p_j in S_j for $1 \le j \le i$

↳ e.g.

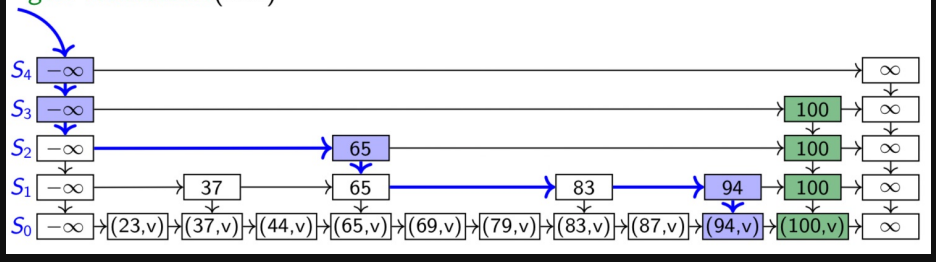


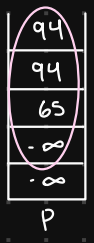
↳ e.g.

Example: *skipList::insert*(100, v)
Coin tosses: H,H,H,T $\Rightarrow i = 3$ 100 appears 4 times



Example: *skipList::insert*(100, v)
Coin tosses: H,H,H,T $\Rightarrow i = 3$
Height increase
getPredecessors(100)

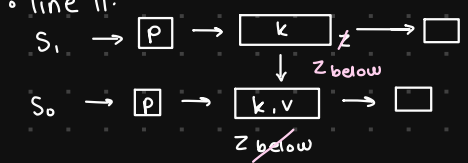




↳ pseudocode (slightly diff from examples):

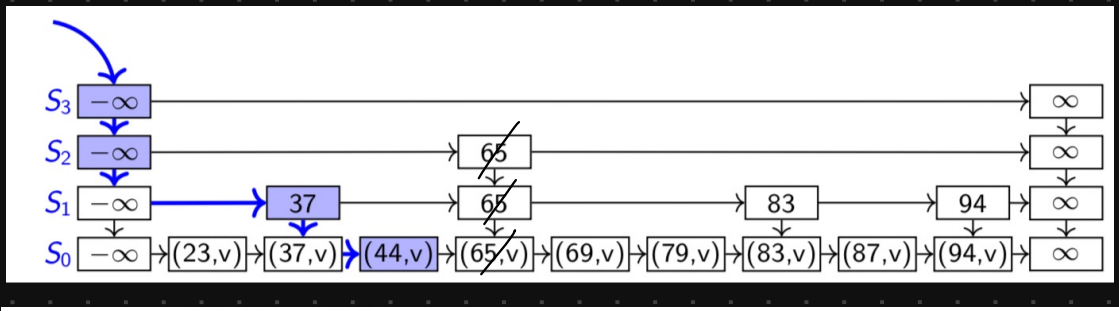
```
skipList::insert(k, v)
1.    P ← getPredecessors(k)
2.    for (i ← 0; random(2) = 1; i ← i+1) {}      // random tower height
3.    while i ≥ P.size()   i+1 ≥ P.size()          // increase skip-list height?
4.        root ← new sentinel-only list, linked in appropriately
5.        add left sentinel of root at bottom of stack P
6.    p ← P.pop()                                  // insert (k, v) in S_0
7.    z_below ← new node with (k, v), inserted after p
8.    while i > 0                                  // insert k in S_1, ..., S_i
9.        p ← P.pop()
10.       z ← new node with k added after p
11.       z.below ← z_below; z_below ← z
12.       i ← i − 1
```

- i+1 is # times k is inserted
- line 11:

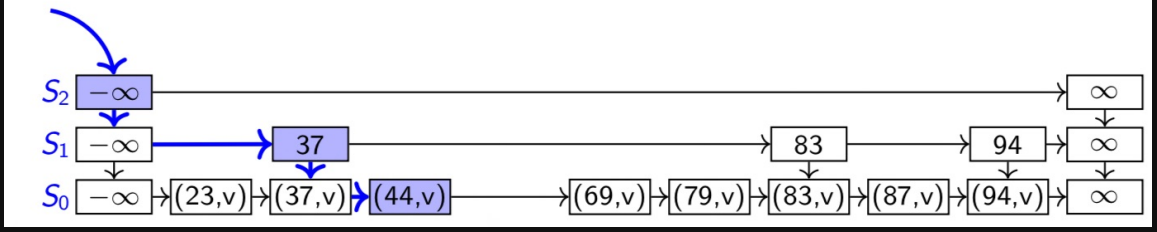


remove key after finding all predecessors, then eliminate layers if there's multiple w/ only sentinels
↳ e.g.

Example: *skipList::delete*(65)
getPredecessors(65)



Example: *skipList::delete*(65)
getPredecessors(65)
Height decrease



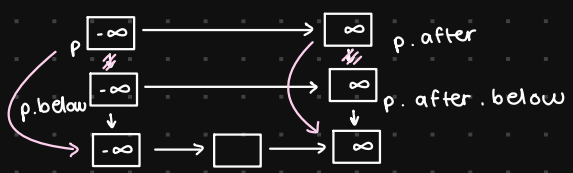
↳ pseudocode:

```
skipList::delete(k)
1.    P ← getPredecessors(k)
2.    while P is non-empty
3.        p ← P.pop()          // predecessor of k in some layer
4.        if p.after.key = k
5.            p.after ← p.after.after
6.        else break           // no more copies of k

7.    p ← left sentinel of the root-list
8.    while p.below.after is the ∞-sentinel
          // the two top lists are both only sentinels, remove one
9.        p.below ← p.below.below
10.       p.after.below ← p.after.below.below
```

- line 4:

- line 8:



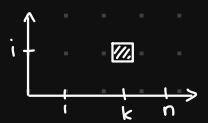
- analysis of skipLists:
  - ↳ expected space usage is $O(n)$
  - ↳ expected height is $O(\log n)$

- e.g. ideal skiplist

3 • 1
2 • . 2
1 • . . . 4
0 • • • • • • • • n = 8

↳ # nodes = $n + \frac{n}{2} + \frac{n}{4} + \ldots \simeq 2n$
↳ height = $\log_2 n$
↳ cost for search = $\Theta(\log_2 n)$ b/c we only visit 1/2 nodes per lvl
↳ in acc skiplist, same run-times in expectation
  - proof:
    For key k, $1 \leq k \leq n$, probability that tower k has height at least i is $\frac{1}{2^i}$, which is probability of $\underbrace{HH\ldots H}_{i \text{ times}}\ldots = \frac{1}{2^i}$.

    $E(\#\text{keys at lvl } i \text{ in tower } k) = 1 \cdot \frac{1}{2^i} + 0 \cdot (1 - \frac{1}{2^i})$



$= \frac{1}{2^i}$

$E(\#\text{keys at lvl } i) = \underbrace{\frac{1}{2^i} + \ldots + \frac{1}{2^i}}_{n \text{ towers}}$

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

$E(\#\text{keys}) = \frac{n}{2^0} + \frac{n}{2^1} + \frac{n}{2^2} + \ldots$
$= 2n$

Thus, space complexity = $\Theta(n)$.

summary:
↳ $O(n)$ expected space
↳ all ops take $O(\log n)$ expected time
↳ biased coin flip to determine tower height gives smaller expected run-times
↳ can save links (↑ space) by implementing towers as arr

RE-ORDERING ITEMS

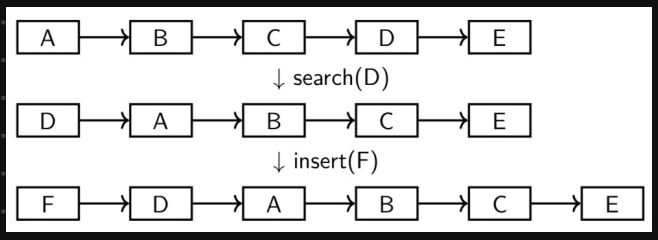
- unordered list / arr of ADT dict
  ↳ search $\Theta(n)$
  ↳ insert $\Theta(1)$
  ↳ delete $\Theta(1)$ after search
- can we do smth to make search more effective?
  ↳ no if items are accessed equally likely
  ↳ yes if we have prob distribution of items
    - frequently accessed items should be in front
    - for short lists / extremely unbalanced distributions, may be faster? easier implementation than AVL trees / skip lists
- e.g. optimal static ordering

key	A	B	C	D	E
frequency of access	2	8	1	10	5
access-probability	$\frac{2}{26}$	$\frac{8}{26}$	$\frac{1}{26}$	$\frac{10}{26}$	$\frac{5}{26}$

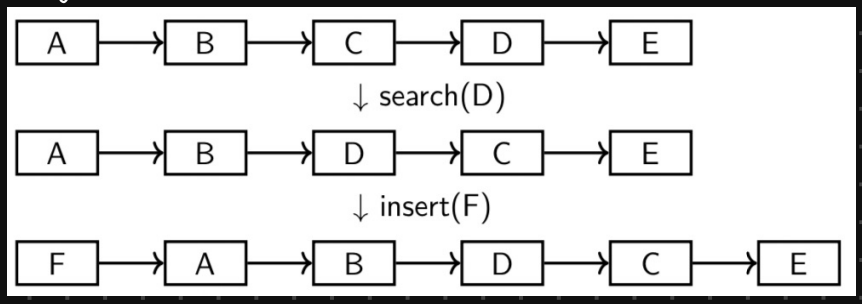
↳ count cost i for accessing i^{th} pos
↳ order A, B, C, D, E has expected access cost = $\frac{2}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{1}{26} \cdot 3 + \frac{10}{26} \cdot 4 + \frac{5}{26} \cdot 5 = \frac{86}{26} \simeq 3.31$
↳ order D, B, E, A, C has expected access cost = $\frac{10}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{5}{26} \cdot 3 + \frac{2}{26} \cdot 4 + \frac{1}{26} \cdot 5 = \frac{54}{26} \simeq 2.07$
  - probs are in dec order
  - optimal order

- **claim**: over all possible static orderings, the one that sorts items by non-inc access prob minimizes expected access cost
  - ↳ proof: optimal ⇒ dec
    We prove contrapositive: not dec ⇒ not optimal
    Take any ordering X, which is not dec
    $X = (x_1, \ldots, x_n)$
    There's an index i st $P(x_i) < P(x_{i+1})$
    $E(\text{cost of search for } X) = 1 \cdot P(x_1) + \ldots + i \cdot P(x_i) + (i+1) \cdot P(x_{i+1}) + \ldots + n \cdot P(x_n)$
    We define ordering X' by swapping x_i & x_{i+1}:
    $X' = (x_1, \ldots, x_{i+1}, x_i, \ldots, x_n)$
    $E(\text{cost of search for } X') = 1 \cdot P(x_1) + \ldots + i \cdot P(x_{i+1}) + (i+1) \cdot P(x_i) + \ldots + n \cdot P(x_n)$
    $E(X') - E(X)$
    $= P(x_i) - P(x_{i+1})$
    < 0
    Thus, $E(\text{cost of search for } X') < E(\text{cost of search for } X)$ ⇒ X is not optimal.

- what if we don't know access probabilities ahead of time?
  - ↳ rule of thumb is **temporal locality** (i.e. recently accessed item is likely to be used again soon)
  - ↳ in list, always insert at front
  - ↳ **Move-To-Front (MTF)** heuristic: upon successful search, move accessed item to front of list
    - e.g.



    - can also do MTF on arr, but should insert & search from back so we have room to grow
  - ↳ **transpose** heuristic: upon successful search, swap accessed item w/ item preceeding it
    - e.g.

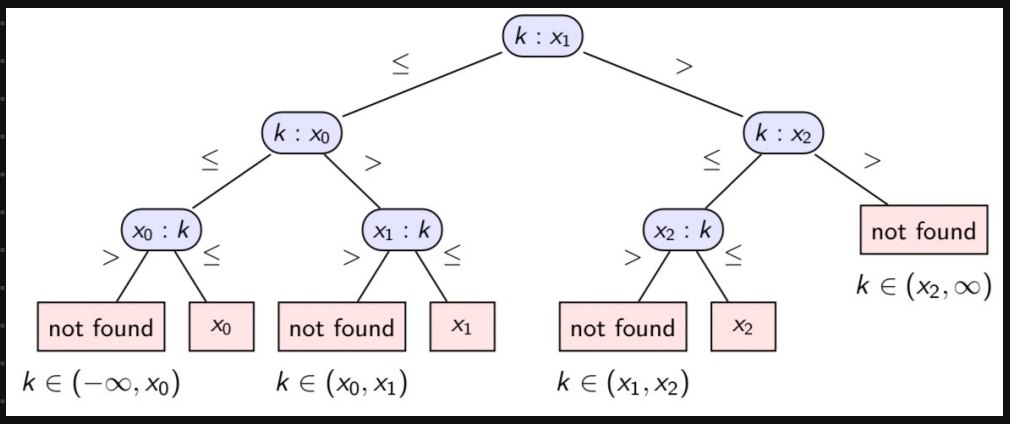


    - doesn't adapt quickly to changing access patterns
  - ↳ MTF works well in practice
  - ↳ MTF is **2-competitive**
    - i.e. no more than 2x bad as optimal static ordering
    - proof:
      Start from list in given order $X^{(0)}$. After N searches, can write $E(\text{cost of search}) = f(N, X^{(0)})$.
      Can prove $\lim_{N \to \infty} f(N, X^{(0)}) = F(\text{probabilities})$.
      optimal static ordering $\leq F(\text{probs}) \leq 2 \cdot$ optimal static ordering

DICTIONARIES FOR SPECIAL KEYS

LOWER BOUND

- fastest realizations of ADT dictionary require $\Theta(\log n)$ time to search among n items
- thm: in comparison model on keys, $\Omega(\log n)$ comparisons are required to search size-n dict
  - ↳ proof: via decision trees for items $x_0, \ldots, x_{n-1}$



INTERPOLATION SEARCH

- for binary search, run-times in sorted arr are:
  - ↳ insert: $\Theta(n)$
  - ↳ delete: $\Theta(n)$
  - ↳ search: $\Theta(\log n)$
  - ↳ pseudocode:

binary-search(A, n, k)
A: Sorted array of size n, k: key

1. $\ell \leftarrow 0$, $r \leftarrow n-1$
2. **while** $(\ell \leq r)$
3. $\quad m \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$
4. $\quad$ **if** $(A[m] == k)$ **then return** "found at $A[m]$"
5. $\quad$ **else if** $(A[m] < k)$ **then** $\ell \leftarrow m+1$
6. $\quad$ **else** $r \leftarrow m-1$
7. **return** "not found, but would be between $A[\ell-1]$ and $A[\ell]$"

- pseudocode for interpolation search is similar to binary search, but compare at interpolated index
  - ↳ need a few extra tests to avoid crash during computation of m
  - ↳

interpolation-search(A, n, k)
A: Sorted array of size n, k: key

1. $\ell \leftarrow 0$, $r \leftarrow n-1$
2. **while** $(\ell \leq r)$
3. $\quad$ **if** $(k < A[\ell]$ or $k > A[r])$ **return** "not found"
4. $\quad$ **if** $(k == A[r])$ **then return** "found at $A[r]$"
5. $\quad m \leftarrow \ell + \lfloor \frac{k-A[\ell]}{A[r]-A[\ell]} \cdot (r-\ell) \rfloor$
6. $\quad$ **if** $(k == A[m])$ **then return** "found at $A[m]$"
7. $\quad$ **else if** $(A[m] < k)$ **then** $\ell \leftarrow m+1$
8. $\quad$ **else** $r \leftarrow m-1$
9. // We always return from somewhere within while-loop

↳ e.g.

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	449	450	600	800	1000	1200	1500

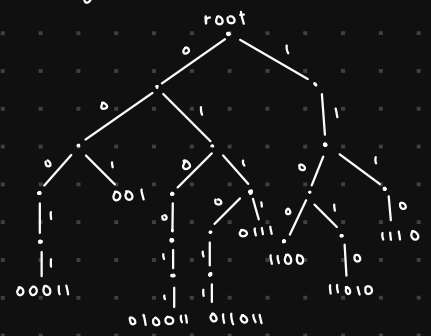
interpolation-search(A[0..10],449):

1) $l = 0$, $r = n - 1 = 10$, $m = l + \lfloor \frac{449 - 0}{1500 - 0} (10 - 0) \rfloor = l + 2 = 2$
2) $l = m + 1 = 3$, $r = 10$, $m = l + \lfloor \frac{449 - 3}{1500 - 3} (10 - 3) \rfloor = l + 2 = 5$
3) $l = 5$, $r = m - 1 = 4$, found at A[4]

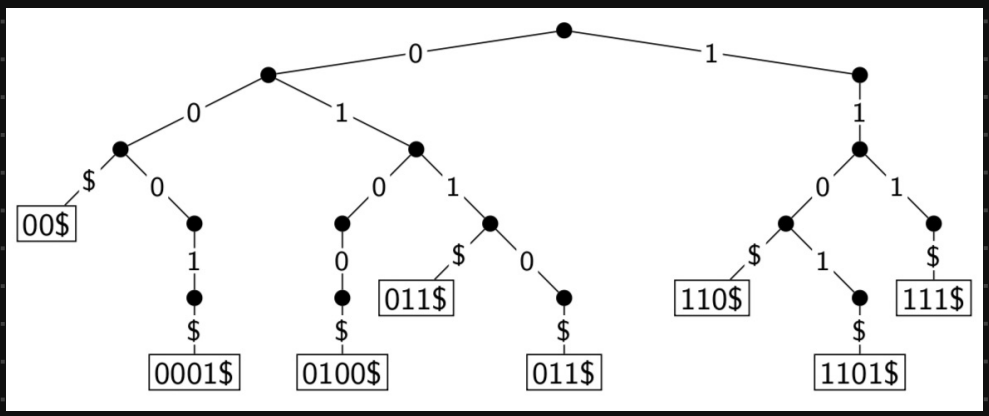
- works well if keys are **uniformly** distributed
  ↳ recurrence rltn is $T^{avg}(n) = T^{avg}(\sqrt{n}) + \Theta(1)$
  $T^{avg}(n) \in \Theta(\log(\log n))$
- worst case performance is $\Theta(n)$

TRIES

- **trie** (aka radix tree): dict for bitstrs
  ↳ comes from word "retrieval", but pronounced "try"
  ↳ tree based on **bitwise comparisons**
    - edge is labelled w/ corresponding bit
  ↳ similar to radix sort b/c it uses individual bits ≠ not whole key
  ↳ e.g.



- assume dict is **prefix-free** (i.e. no str is prefix of another)
  ↳ satisfied if all strs have same length
  ↳ satisfied if all strs end w/ **"end-of-word" char $**
  ↳ e.g. trie for {00$, 0001$, 0100$, 011$, 0110$, 110$, 1101$, 111$}

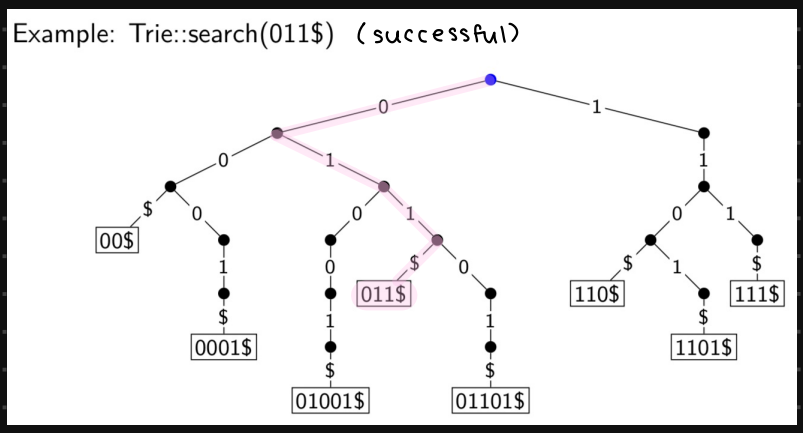


  ↳ items (i.e. keys) are only stored in leaf nodes
- for **search**:
  1) start from root ≠ most significant bit of x
  2) follow link that corresponds to current bit of x ≠ return failure if link is missing
  3) return success if we reach leaf
     - must store x
  4) else, recurse on new node ≠ next bit of x

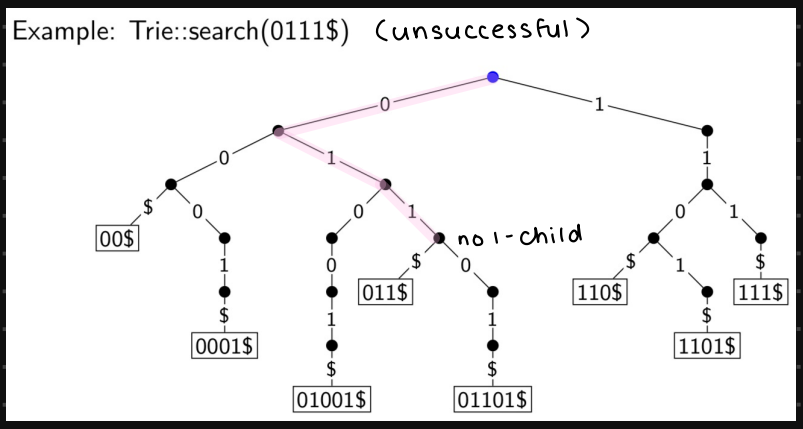
↳ pseudocode:

```
Trie::search(v ← root, d ← 0, x)
v: node of trie; d: level of v, x: word stored as array of chars
1.  if v is a leaf
2.      return v
3.  else
4.      let v' be child of v labelled with x[d]
5.      if there is no such child
6.          return "not found"
7.      else Trie::search(v', d + 1, x)
```

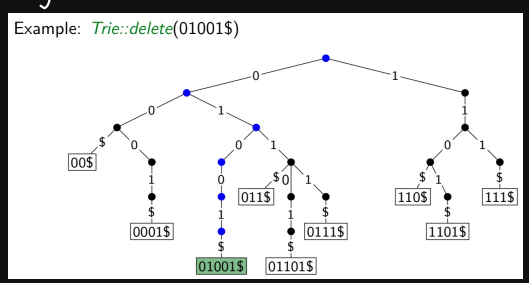
↳ e.g.

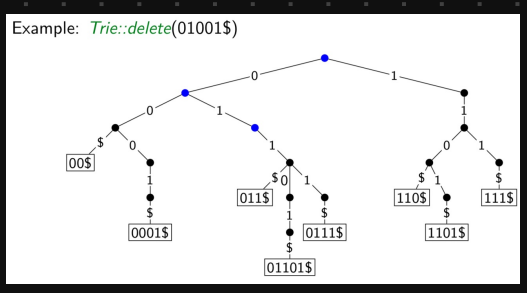


↳ e.g.

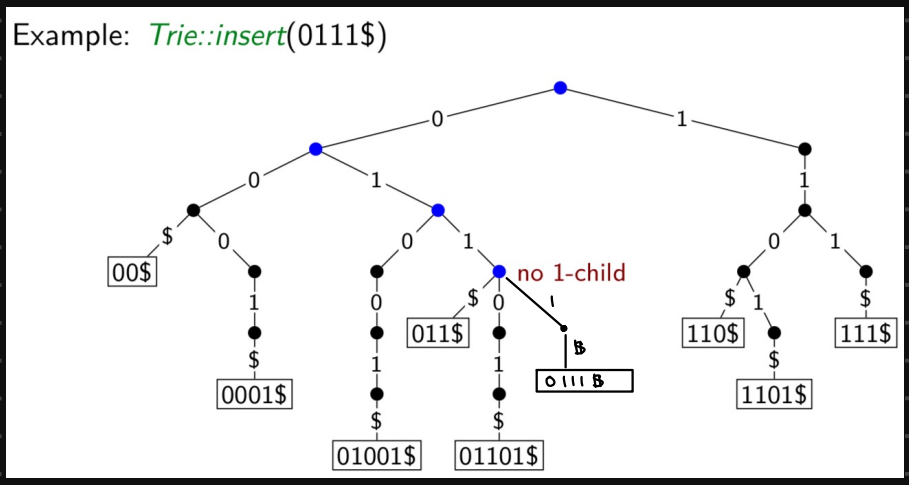


- Trie:. insert(x):
  1) search for x, which should be unsuccessful
  2) suppose we finish at node v that's missing suitable child
  3) expand trie from node v by adding necessary nodes that correspond to extra bits of x
- Trie:: delete(x):
  1) search for x
  2) let v be leaf where x is found
  3) delete v & all ancestors of v until we reach ancestor that has 2 children
- time complexity of all ops is $\Theta(|x|)$
  ↳ $|x|$ is length of bin str x (i.e. # bits)
- e.g.

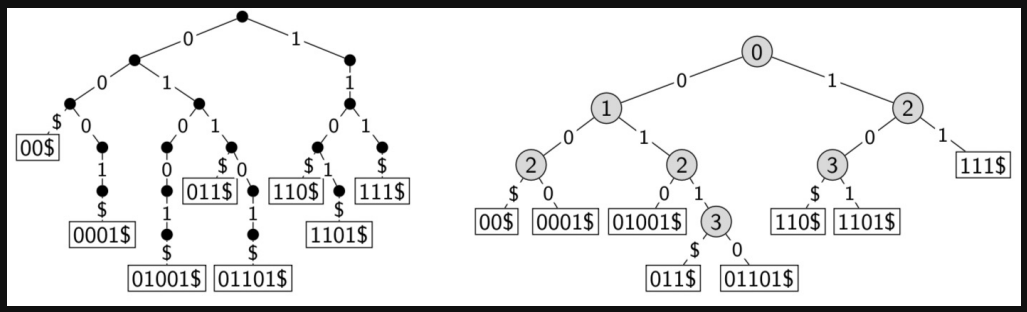




- e.g.



- compressed trie: compress paths of nodes w/only 1 child
  - ↳ each node stores index, corresponding to depth of in uncompressed trie
    - gives next bit to be tested during search
  - ↳ compressed trie w/ n keys has at most n-1 internal nodes
  - ↳ e.g.

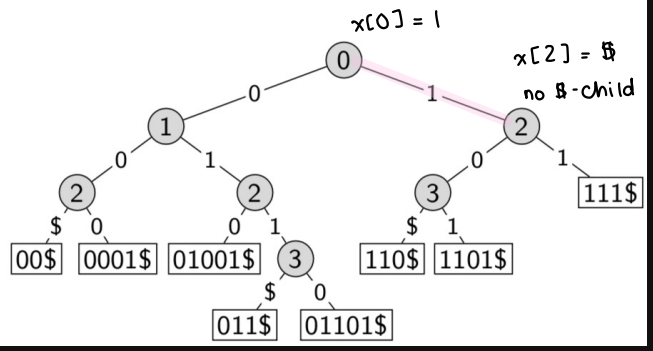


  - ↳ aka Practical Algo to Retrieve Info Coded in Alphanumeric (Patricia) trees
- CompressedTrie::search:
  1) start from root & bit indicated at that node
  2) follow link that corresponds to current bit in x
     - return failure if link is missing
  3) if we reach leaf, explicitly check whether word is stored at leaf is x
  4) else, recurse on new node & next bit of x
  - ↳ pseudocode:

```
CompressedTrie::search(v ← root, x)
v: node of trie; x: word
1.    if v is a leaf
2.        return strcmp(x, v.key)
3.    d ← index stored at v
4.    if x has at most d bits
5.        return "not found"
6.    v' ← child of v labelled with x[d]
7.    if there is no such child
8.        return "not found"
9.    CompressedTrie::search(v', x)
```

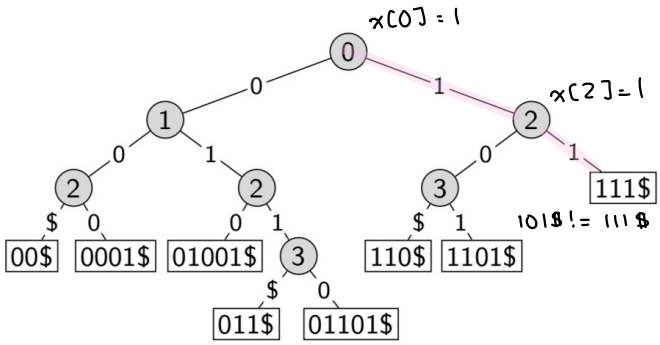
  - ↳ e.g.

Example: CompressedTrie::search(10$) (unsuccessful)



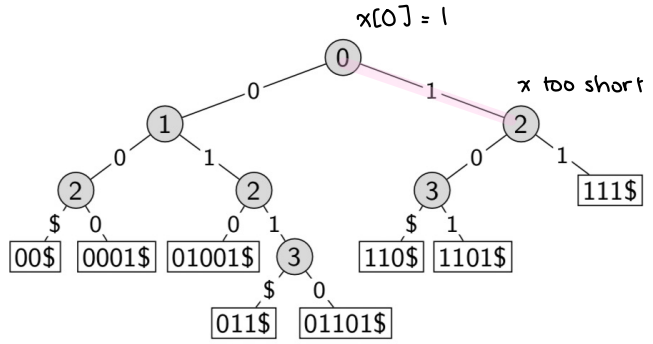
↳ e.g.

Example: CompressedTrie::search(101$) (unsuccessful)



↳ e.g.

Example: CompressedTrie::search(1$) (unsuccessful)

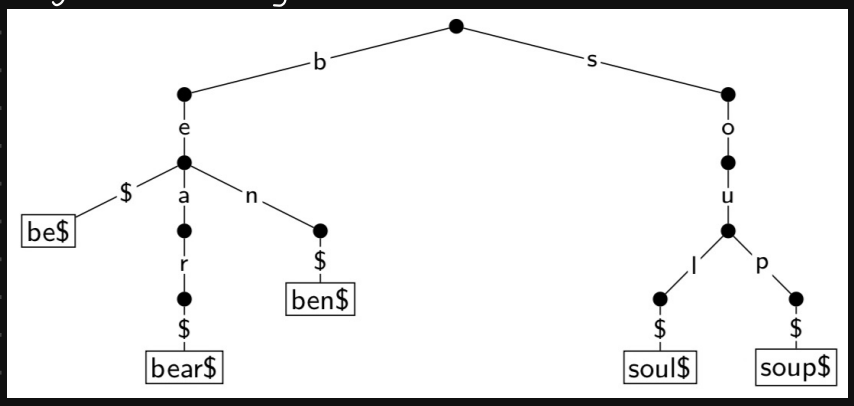


- `CompressedTrie::delete(x)`:
  1) perform search(x)
  2) remove node v that stored x
  3) compress along path to v wherever possible
- `CompressedTrie::insert(x)`:
  1) perform search(x)
  2) let v be node where search ended
  3) simplest approach:
     - uncompress path from root to v
     - insert x as uncompressed trie
     - compress paths from root to v & from root to x

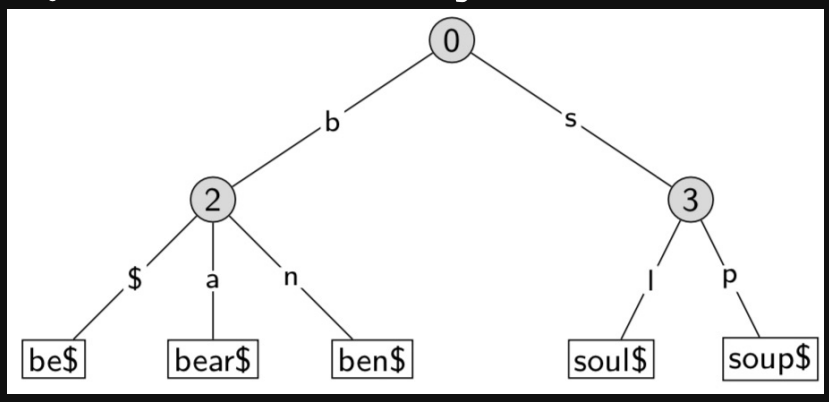
  ↳ for step 3, can also be done by adding only nodes that are needed
     - requires `leaf-links`: every node stores a link to a leaf that's a descendant
- all ops take `O(|x|)` time

- more complicated, but space savings are worth it if words are unevenly distributed

- **multiway tries** can rep strs over any fixed alphabet Σ
  - ↳ any node will have at most $|\Sigma| + 1$ children
    - 1 child for end-of-word char $
  - ↳ e.g. trie holding strs {bear$, ben$, be$, soul$, soup$}



- **compressed multiway tries** · compress paths as before
  - ↳ e.g. compressed trie holding strs {bear$, ben$, be$, soul$, soup$}



- ops search(x), insert(x), ? delete(x) are exactly as for tries for bitstrs
- run-time is **O(|x|·(time to find appropriate child))**
- each node has up to $|\Sigma| + 1$ children; to store them:
  - ↳ arr of size $|\Sigma| + 1$ for each node
    - O(1) time to find child; $O(|\Sigma|)$ space per node
  - ↳ list of children for each node
    - $O(|\Sigma|)$ time to find child; O(#children) space per node
  - ↳ dict of children for each node
    - O(log(#children)) time; O(#children) space per node
    - best in theory, but not worth in practice unless $|\Sigma|$ is huge

DICTIONARIES VIA HASHING

HASHING INTRO

- for a known $M \in \mathbb{N}$, every key k is int w/ $0 \leq k < M$
  - ↳ implement dict by using arr A of size M that stores (k,v) via $A[k] \leftarrow v$
  - ↳ e.g.

0	
1	
2	dog
3	
4	
5	
6	cat
7	
8	pig

  - ↳ search(k): check whether $A[k]$ is NULL
  - ↳ insert(k, v): $A[k] \leftarrow v$
  - ↳ delete(k): $A[k] \leftarrow$ NULL
- each **op is $\Theta(1)$** & total **space is $\Theta(M)$**
- 2 disadvantages of direct addressing:
  - ↳ can't be used if keys aren't ints
  - ↳ wastes space if M is unknown or $n \ll M$
- **hashing idea**: map keys to ints in range $\{0, \ldots, M-1\}$, then use direct addressing
  - ↳ assume all keys come from some **universe U**
    - typically, U = non-ve ints
    - $|U|$ is sometimes finite
  - ↳ design **hash fcn** $h: U \to \{0, 1, \ldots, M-1\}$
    - commonly used is **$h(k) = k \bmod M$**
  - ↳ store dict in **hash table**
    - i.e. arr T of size M
  - ↳ item w/ key k should ideally be stored in **slot $h(k)$** (i.e. $T[h(k)]$)
- e.g.

$U = \mathbb{N}$, $M = 11$, $h(k) = k \bmod 11$.
The hash table stores keys 7, 13, 43, 45, 49, 92. (Values are not shown).

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	
9	
10	43

- generally, hash fcn h is not injective so many keys can map to same int
  - ↳ e.g. $h(46) = 2 = h(13)$ if $h(k) = k \bmod 11$
  - ↳ get **collisions**, where we want to insert (k,v) into table but $T[h(k)]$ is alr occupied

HASHING WITH CHAINING

- simplest collision-resolution strat: each slot stores bucket containing 0+ KVPs
  - ↳ bucket could be implemented by any dict realization
  - ↳ simplest approach is to use **unsorted linked lists** for buckets
    - called collision resolution by **chaining**
- **search(k)**: look for key k in list at T[h(k)]
  - ↳ apply MTF heuristic
- **insert(k,v)**: add (k,v) to front of list a T[h(k)]
- **delete(k)**: perform search, then delete from linked list
- insert takes **$\Theta(1)$ time**
- search & delete have run-times of **$\Theta(1 + \text{size of bucket } T[h(k)])$**
  - ↳ avg bucket size is **$\alpha = \frac{n}{M}$**
    - α is **load factor**
    - doesn't imply that avg-case cost search & delete is $\Theta(1+\alpha)$
      - → if all keys hash to same slot, then avg bucket size is still α, but ops take time $\Theta(n)$ on avg
- to analyze what happens on avg, switch to **randomized hashing**
  - ↳ assume:
    - universe U is finite
    - hash fcn is chosen at random (equally likely)
  - ↳ e.g.

    $U = \{1, 2, 3\}$

    $M = 2$ (size of hash table)

h(1)	h(2)	h(3)
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

    - there are $2^3 = M^{|U|}$ fcns
    - there are $4 = 2^2$ fcns st $h(1) = 0$ ($k = 1$, $i = 0$)
    - there are $2 = 2^1$ fcns st $h(1) = 0$ & $h(2) = 1$
  - ↳ we can show **$P(h(k) = i) = \frac{1}{M}$** for any key k & slot i
    - fix k in U
    - fix i in $\{0, \ldots, M-1\}$
    - prob that $h(k) = i$ is $P = \frac{M^{|U|-1}}{M^{|U|}}$ ← #hash fcns that send k to i (can't choose 1 elmt in hash fcn arr) / ↖ total #hash fcns

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

  - ↳ we can show hash vals for any **2 keys are indep** of each other:
    - fix k, k' in U & $k \neq k'$
    - fix i, i' in $\{0, \ldots, M-1\}$
    - $P(h(k) = i \text{ & } h(k') = i) = \frac{M^{|U|-2}}{M^{|U|}}$ ← #hash fcns st $k \to i$ & $k' \to i'$ / ↖ total # hash fcns

$$= \frac{1}{M^2}$$

$$= \left(\frac{1}{M}\right)\left(\frac{1}{M}\right)$$ ← shows 2 keys are indep of each other

- each key in dict is expected to **collide w/ $\frac{n-1}{M}$ other keys** & expected cost of search & delete is **$\Theta(1+\alpha)$**
  - ↳ $\alpha = \frac{n}{M}$

↳ proof:

$P(h(k) = h(k'))$

$= \underbrace{P(h(k) = h(k') = 0)}_{\frac{1}{M^2}} + \underbrace{P(h(k) = h(k') = 1)}_{\frac{1}{M^2}} + \ldots + \underbrace{P(h(k) = h(k') = M-1)}_{\frac{1}{M^2}}$

$= M\left(\frac{1}{M^2}\right)$

$= \frac{1}{M}$

$E(\#\text{collisions w/ } k_i) = \sum_{j=2}^{n} P(\text{collision btwn } k_i \text{ \& } k_j)$

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

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

$= \alpha - \frac{1}{M}$ ← $\frac{1}{M}$ is small so we can disregard in search & delete run-times

- for all collision resolution strats, run-time eval is done in terms of load factor $\alpha = \frac{n}{M}$
- keep α small by rehashing when needed
  - ↳ keep track of n & M throughout ops
  - ↳ if α gets too large (e.g. $> \frac{3}{4}$), create new 2x bigger hash-table, new fcns, & re-insert all items in new table
  - ↳ rehashing costs $\Theta(M+n)$ but happens rarely enough that we can ignore this term when amortizing over all ops
  - ↳ rehash when α gets too small (e.g. $< \frac{1}{2}$) so that $M \in \Theta(n)$ throughout & space is always $\Theta(n)$
- if we maintain $\alpha \in \Theta(1)$ under uniform hashing assumption, expected cost for hashing w/ chaining is $O(1)$ & space is $\Theta(n)$

PROBE SEQUENCES

- main idea is to avoid links needed for chaining by permitting only 1 item per slot, but allowing key k to be in multiple slots
- search & insert follow probe sequence of possible locations for key k: $\langle h(k,0), h(k,1), h(k,2), \ldots \rangle$ until empty spot is found
- for delete, can't leave empty spot behind or next search might stop too soon
  - ↳ use lazy deletion: mark spot as deleted rather than NULL & continue searching past deleted spots

simplest method for open addressing is linear probing

↳ $h(k,i) = (h(k) + i) \bmod M$ for some hash fcn h

- e.g. insert

$M = 11$, $h(k) = k \bmod 11$, $h(k,i) = (h(k) + i) \bmod 11$.

insert(20)

$h(20,2) = 0$

$h(20,1) = 10$

$h(20,0) = 9$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	43

e.g. delete

$M = 11$, $h(k) = k \bmod 11$, $h(k,i) = (h(k) + i) \bmod 11$.

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	*deleted*

delete(43)

$h(43, 0) = 10$

- e.g. search

$M = 11$, $h(k) = k \bmod 11$, $h(k,i) = (h(k) + i) \bmod 11$.

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	*deleted*

search(63)

$h(63, 6) = 3$
not found

- probe sequence ops pseudocode:

```
probe-sequence::insert(T, (k, v))
1.    for (j = 0; j < M; j++)
2.        if T[h(k, j)] is NIL or "deleted"
3.            T[h(k, j)] = (k, v)
4.            return "success"
5.    return "failure to insert"      // need to re-hash
```

```
probe-sequence-search(T, k)
1.    for (j = 0; j < M; j++)
2.        if T[h(k, j)] is NIL
3.            return "item not found"
4.        else if T[h(k, j)] has key k
5.            return T[h(k, j)]
6.        // ignore "deleted" and keep searching
7.    return "item not found"
```

- some hashing methods require 2 hash fcns h_0 & h_1
  - ↳ fcns should be indep (i.e. random vars $P(h_0(k) = i)$ & $P(h_1(k) = j)$ are indep)
- using 2 modular fcns may lead to dependencies, so use multiplication method for 2nd hash fcn
  - ↳ $h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$

- A is some float w/ $0 < A < 1$
- $kA - \lfloor kA \rfloor$ computes fractional part of kA, which is in $[0, 1)$
- multiply w/ M to get float in $[0, M)$
- round down to get int in $\{0, \ldots, M-1\}$

↳ multiply w/ A to scramble keys
- $A = \varphi = \frac{\sqrt{5}-1}{2} \approx 0.618034$ to achieve good scramble

- assume we have 2 indep hash fcns h_0 & h_1
  ↳ $h_1(k) \neq 0$ & $h_1(k)$ is relative prime w/ table size M for all keys k
  - choose M prime
  - modify std hash fcns to ensure $h_1(k) \neq 0$
    → e.g. modified multiplication method: $h(k) = 1 + \lfloor (M-1)(kA - \lfloor kA \rfloor) \rfloor$
- **double hashing**: open addressing w/ probe seq $h(k, i) = (h_0(k) + i \cdot h_1(k)) \bmod M$
  ↳ search, insert, & delete work just like for linear probing, but w/ diff probe seq
  ↳ e.g.

$M = 11$, $h_0(k) = k \bmod 11$, $h_1(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$

insert(41)

$h_0(41) = 8$

$h(41, 0) = 8$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

↳ e.g.

$M = 11$, $h_0(k) = k \bmod 11$, $h_1(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$

insert(194)

$h_0(194) = 7$

$h(194, 0) = 7$

$h_1(194) = 9$

$h(194, 1) = 5$

$h(194, 2) = 3$

0	
1	45
2	13
3	194
4	92
5	49
6	
7	7
8	41
9	
10	43

CUCKOO HASHING

- use 2 indep hash fcns h_0 & h_1 & 2 tables T_0 & T_1
- main idea is an item w/ key k can only be at $T_0[h_0(k)]$ or $T_1[h_1(k)]$
  ↳ **search & delete** take constant time
  ↳ **insert** always initially puts new item into $T_0[h_0(k)]$
  - if $T_0[h_0(k)]$ is occupied, kick out other item k' & attempt to re-insert into alt pos $T_1[h_1(k')]$
  - may lead to loop of kicking out
    → detect this by aborting after too many attempts
  - if failure, rehash w/ larger M & new hash fcns
  - may be slow, but expected constant time if load factor is small enough
- **pseudocode**:

```
cuckoo::insert(k, v)
1.    i ← 0
2.    do at most 2n times:
3.        if T_i[h_i(k)] is NIL
4.            T_i[h_i(k)] ← (k, v)
5.            return "success"
6.        swap((k, v), T_i[h_i(k)])
7.        i ← 1 − i
8.    return "failure to insert"      // need to re-hash
```

↳ after $2n$ iterations, there's loop in kicking out seq

· e.g.

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

insert(51)

$i = 0$
$k = 51$

$h_0(k) = 7$
$h_1(k) = 5$

	T_0	T_1
0	44	
1		
2		
3		
4	59	
5		
6		
7	51	
8		
9		92
10		

· e.g.

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

insert(95)

$i = 1$
$k = 51$

$h_0(k) = 7$
$h_1(k) = 5$

	T_0	T_1
0	44	
1		
2		
3		
4	59	
5		51
6		
7	95	
8		
9		92
10		

· 2 hash tables don't need to be same size
· load factor $\alpha = \frac{n}{\text{size of } T_0 + \text{size of } T_1}$
↳ if $\alpha < \frac{1}{2}$, insert has $O(1)$ expected run-time
· many variations:
↳ 2 hash tables can be combined into 1
↳ be flexible when inserting (i.e. always consider both possible pos)

↳ use $k > 2$ allowed locations (i.e. k hash fcns)

- complexity of open addressing strategies:

Expected # probes $\leq$	*search (unsuccessful)*	*insert*	*search (successful)*
Linear Probing	$\frac{1}{(1-\alpha)^2}$	$\frac{1}{(1-\alpha)^2}$ if $\alpha \geq 1$, $\to \infty$	$\frac{1}{1-\alpha}$ (on avg. over keys)
Double Hashing	$\frac{1}{1-\alpha} + o(1)$	$\frac{1}{1-\alpha} + o(1)$ if $\alpha \geq 1$, $\to \infty$	$\frac{1}{1-\alpha} + o(1)$
Cuckoo Hashing	1 (worst-case)	$\frac{\alpha}{(1-2\alpha)^2}$ if $\alpha \geq \frac{1}{2}$, $\to \infty$	1 (worst-case)

↳ for open addressing scheme, must have $\alpha \leq 1$

↳ for analysis, require $\alpha < 1$ ∴ cuckoo hashing requires $\alpha < \frac{1}{2}$

↳ all ops have O(1) expected run-time if hash fcn is chosen uniformly ∴ α is kept sufficiently small

- for fixed hash fcn, worst-case run-time is usually $\Theta(n)$

HASH FUNCTION STRATEGIES

- satisfying uniform hashing assumption is impossible b/c there's too many hash fcns
  ↳ wouldn't know how to look up $h(k)$
- compromise by choosing hash fcn that's easy to compute, but aim for $P(h(k) = i) = \frac{1}{M}$ wrt key distr
  ↳ doable if all keys used equally often, but not the case in practice
- get good performance by choosing hash fcn that's:
  ↳ unrelated to any possible patterns in data
  ↳ depends on all parts of key

2 basic methods for int keys:

↳ modular: $h(k) = k \bmod M$
- choose M to be prime

↳ multiplicative: $h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$
- A is constant float ∴ $0 < A < 1$

- Carter-Wegman's universal hashing is randomization that uses easy-to-compute hash fcns
  ↳ requires all keys are in $\{0, \ldots, p\}$ for some big prime p
  ↳ choose $M < p$ arbitrarily
  - power of 2 is ok
  ↳ use hash fcn $h(k) = ((ak+b) \bmod p) \bmod M$
  - $h(k)$ computed in $O(1)$ time
  - choosing h doesn't satisfy uniform hashing assumption, but can prove 2 keys collide w/prob at most $\frac{1}{M}$
    → enough to prove run-time bounds for chaining
- if keys are multi-dimensional (e.g. strs in Σ^*), flatten str w/ int $f(w) \in \mathbb{N}$
  ↳ e.g.
  APPLE → (65, 80, 80, 76, 69) ASCII
  → $65R^4 + 80R^3 + 80R^2 + 76R^1 + 69R^0$ for some radix R
  ↳ combine w/ modular hash fcn $h(w) = f(w) \bmod M$
  ↳ to compute $O(|w|)$ time w/o overflow, use Horner's rule ∴ apply mod early
  - e.g.
    $h(\text{APPLE}) = ((((((65R+80) \bmod M)R + 80) \bmod M)R + 76) \bmod M)R + 69) \bmod M$

RANGE-SEARCHING IN DICTIONARIES OF POINTS

RANGE SEARCHES

search(k) looks for 1 specific item

- RangeSearch looks for all items that fall within given range
  - ↳ input is range (i.e. interval $I = (x, x')$)
    - may be open / closed at ends
  - ↳ output is all KVPs in dict whose key k satisfies $k \in I$
  - ↳ e.g.

5	10	11	17	19	33	45	51	55	59

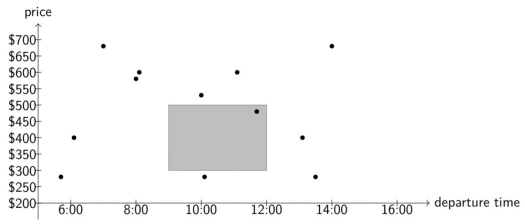
RangeSearch((18,45]) should return {19, 33, 45}

- let s be output size (i.e. # items in range)
  - ↳ need $\Omega(s)$ time to report items
  - ↳ sometimes $s=0$ & sometimes $s=n$ so we keep it as separate param when analyzing run-time
- in unsorted list / arr / hash table, range search requires $\Omega(n)$ time b/c must check for each item explicitly
- in sorted arr, range search done in $O(\log n + s)$ time
  - ↳ use bin search & find i st x is at A[i]
  - ↳ use bin search & find i' st x' is at A[i']
  - ↳ report all items $A[i+1, \ldots, i'-1]$
  - ↳ report A[i] & A[i'] if in range

MULTI-DIMENSIONAL DATA

- range searches are of special interest for multi-dimensional data
  - ↳ e.g.

Example: flights that leave between 9am and noon, and cost $300-$500



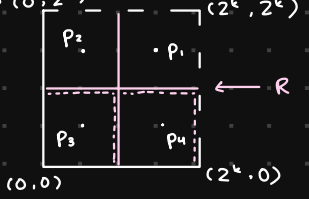
  - ↳ each item has d aspects / coords : $(x_0, x_1, \ldots, x_{d-1})$
  - ↳ aspect vals are #s
  - ↳ each item corresponds to point in d-dimensional space
  - ↳ concentrate on d=2 (i.e. Euclidean plane)
- d-dimensional range search : given query rectangle A, find all points that lie within A
  - ↳ design new data structs specifically for points:
    - quadtrees
    - kd-trees
    - range-trees
  - ↳ assume points are in general position, meaning no 2 x-coords or y-coords are same

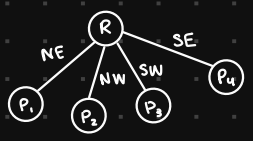
QUADTREES

- have n points $S = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ in plane
- need bounding box $R = [0, 2^k) \times [0, 2^k)$, which is square containing all pts
  - ↳ find smallest k st max x & y values in S are $< 2^k$

↳ variation is to pick left coord based on min val st size is power of 2

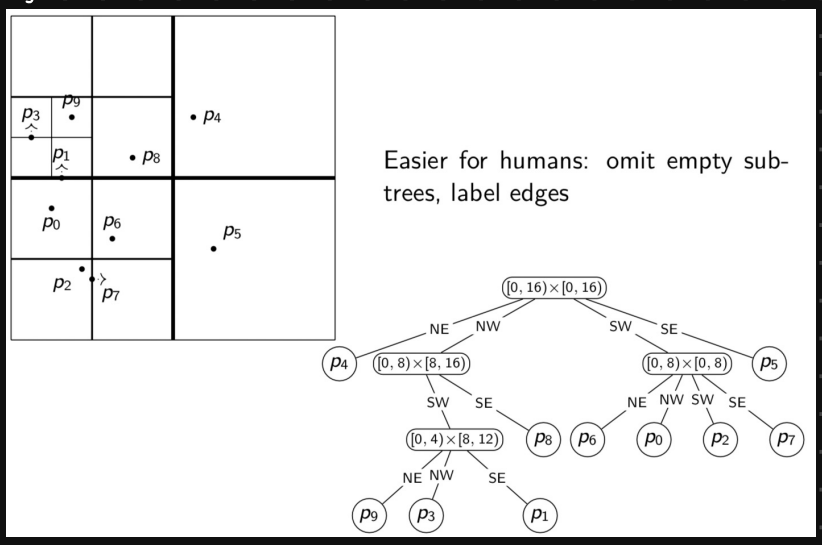
↳ $(0, 2^k)$ $(2^k, 2^k)$



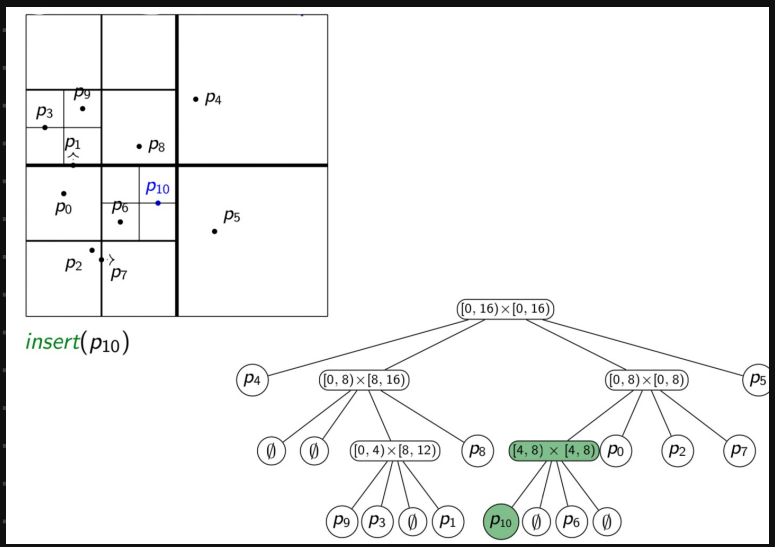


° open on top & right

- to build quadtree that stores S, structure is:
  - ↳ root r of quadtree is associated w/region R
  - ↳ if R contains 0 or 1 pts, then root r is leaf that stores pt
  - ↳ else split by partitioning R into 4 equal subsquares (i.e. quadrants)
    - ° R_{NE}, R_{NW}, R_{SW}, R_{SE}
  - ↳ partition S into sets S_{NE}, S_{NW}, S_{SW}, S_{SE} of pts in these regions
    - ° pts on split lines belong to right / top side
  - ↳ recursively build tree T_i for pts S_i in region R_i & make them children of root
  - ↳ e.g.



- search is analogous to BSTs & tries
- insert:
  - ↳ search for point
  - ↳ split leaf while there's 2 points in 1 region
  - ↳ e.g.



- delete:
  - ↳ search for pt

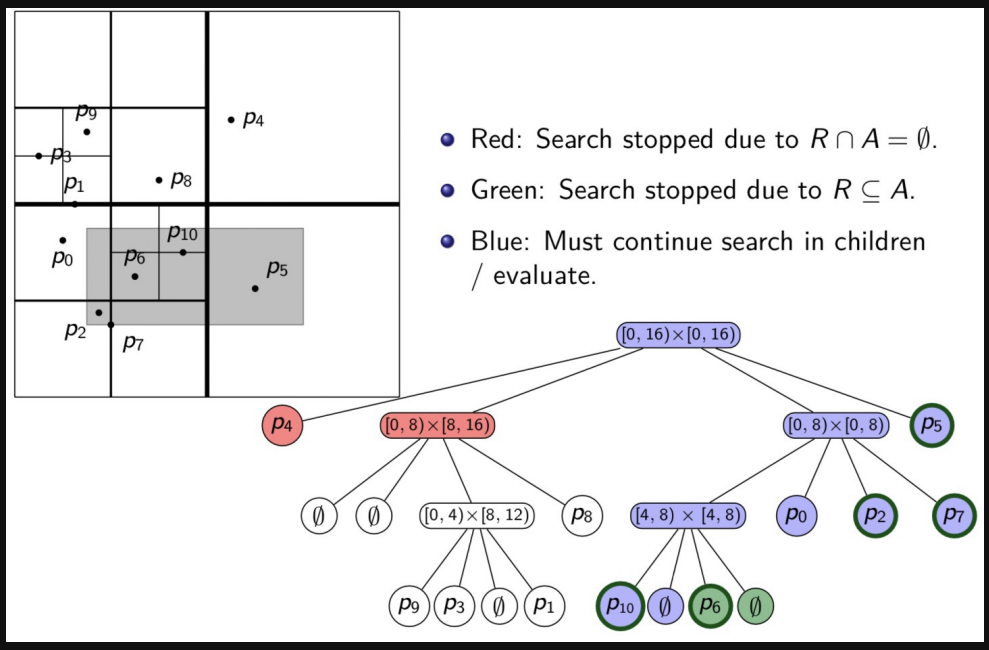
↳ remove pt
↳ if its parent s has only 1 pt left, delete parent & recursively all ancestors that have only 1 pt left

- RangeSearch for quadtree pseudocode:

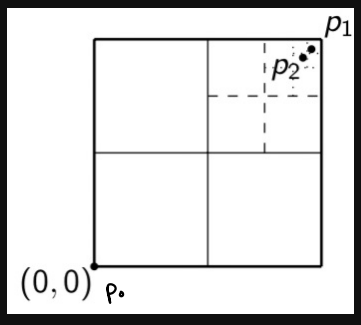
```
QTree::RangeSearch(r ← root, A)
r: The root of a quadtree, A: Query-rectangle
1.   R ← region associated with node r
2.   if (R ⊆ A) then                    // inside node
3.         report all points below r; return
4.   if (R ∩ A is empty) then            // outside node
5.         return
                                        // The node is a boundary node, recurse
6.   if (r is a leaf) then
7.       p ← point stored at r
8.       if p is in A return p
9.       else return
10.  for each child v of r do
11.      QTree::RangeSearch(v, A)
```

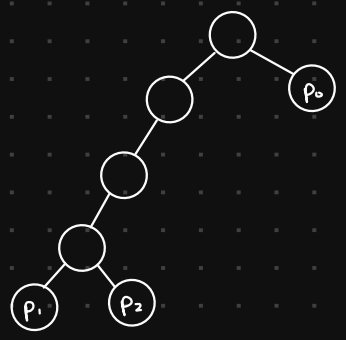
← A.left ≤ R.left & R.right ≤ A.right
A.down ≤ R.down & R.up ≤ A.up

↳ assume that each node of quadtree stores associated square
↳ e.g.



- quadtree analysis:

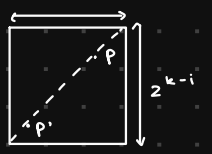




← bad case b/c we have very large height since pts are badly distributed

↳ can prove that if $p_0 = (0,0)$, $p_1 = (1 - \frac{1}{2^s}, 1 - \frac{1}{2^s})$, $p_2 = (1 - \frac{2}{2^s}, 1 - \frac{2}{2^s})$, quadtree has height $\Theta(s)$
↳ define $\beta = \frac{\text{sidelength of } R}{d_{min}} = \frac{2^k}{d_{min}}$ to be spread factor of points S
- d_{min} is min distance btwn pts in S
- prove $h \in \Theta(\log \beta)$:

Take deepest internal node in T. Its lvl is $i = h-1$? its region has sidelength 2^{k-1}. There's at least 2 pts in it.



diagonal

$d_{min} \leq d(P, P') \leq \sqrt{2}\, 2^{k-i}$

↑ defn of d_{min}

$\log(d_{min}) \leq \frac{1}{2} + k - i$

$\log(d_{min}) \leq \frac{1}{2} + k - h + 1$

$h \leq k - \log(d_{min}) + \frac{3}{2}$

$h \leq \log\beta + \frac{3}{2}$ $\quad \log\beta = k - \log(d_{min})$

↳ complexity to build initial tree is $\Theta(nh)$ worst-case

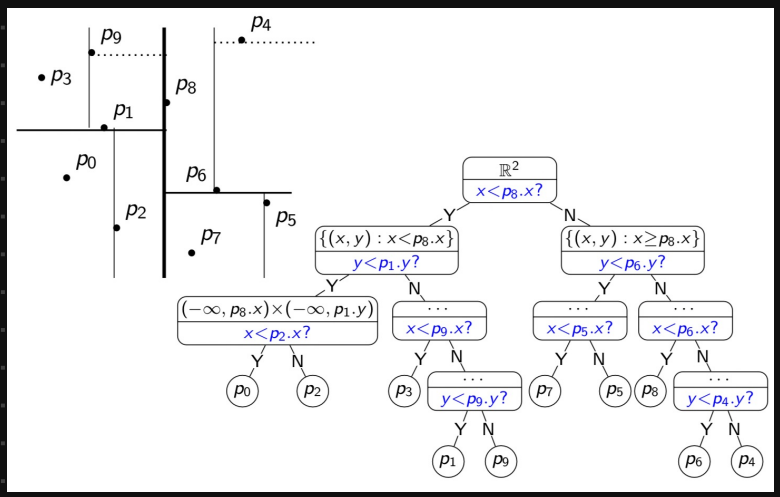
↳ complexity of range search is $\Theta(nh)$ even if ans is $\emptyset$

↳ much faster in practice

- easy to compute ? handle b/c there's no complicated arithmetic
  ↳ only div by 2 (bit-shift) if width / height of R
- space is potentially wasteful, but good if pts are well-distributed
- easy to generalize to higher dimensions
  ↳ e.g. 3D is octree

KD-TREES

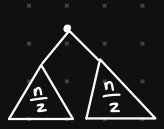
- have n pts $S = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$
- quadtrees split square into quadrants regardless of where pts are
- pt-based kd-tree: split region st roughly half of pts are in each subtree
  ↳ each node of kd-tree keeps track of splitting line in 1D
    - in 2D, either vertical / horizontal
  ↳ convention is pts on split lines belong to right / top side
  ↳ continue splitting, switching btwn vertical ? horizontal lines, until every pt is in separate region
- e.g.



- to build kd-tree w/ init split by x on pts S (w/ initial y-split symmetric):
  ↳ if $|S| \leq 1$, create leaf ? return
  ↳ else, x = quick-select$(S, \lfloor \frac{n}{2} \rfloor)$
    - select by x-coord
  ↳ partition S by x-coord int $S_{x<x}$? $S_{x\geq x}$
    - $\lfloor \frac{n}{2} \rfloor$ pts on 1 side ? $\lceil \frac{n}{2} \rceil$ on other
  ↳ create left subtree recursively (splitting by y) for pts $S_{x<x}$
  ↳ create right subtree recursively (splitting by y) for pts $S_{x\geq x}$
- run-time is $\Theta(n \log n)$ expected time
  ↳ find X ? partition S in $\Theta(n)$ expected time using randomized-quick-select

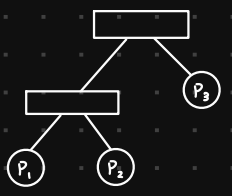
↳ both subtrees have $\simeq \frac{n}{2}$ pts so $T^{exp}(n) = 2T^{exp}(\frac{n}{2}) + O(n)$
- resolves to $\Theta(n \log n)$ expected time
- $\Theta(n \log n)$ worst-case by pre-sorting

- **height** is $h(1) = 0$, $h(n) \leq h(\lceil \frac{n}{2} \rceil) + 1$
↳ proof:
$h(n)$ = worst-case height of kd-tree w/ n pts
$h(n) \in \Theta(\log n)$ ⟵ same type of algo as Binary Search
↳ specifically, $h(n) \in \Theta(\lceil \log n \rceil)$



$h(n) = h(\frac{n}{2}) + 1$
sloppy recurrence b/c we don't care abt ceiling

- **space complexity**:



$n = 3$ leaves
2 internal nodes
$3 + 2 = 5$ nodes

In kd-tree, every internal node has 2 children.
internal nodes $\leq \underbrace{\#\text{leaves} - 1}_{n-1}$ (refer to compressed tries)

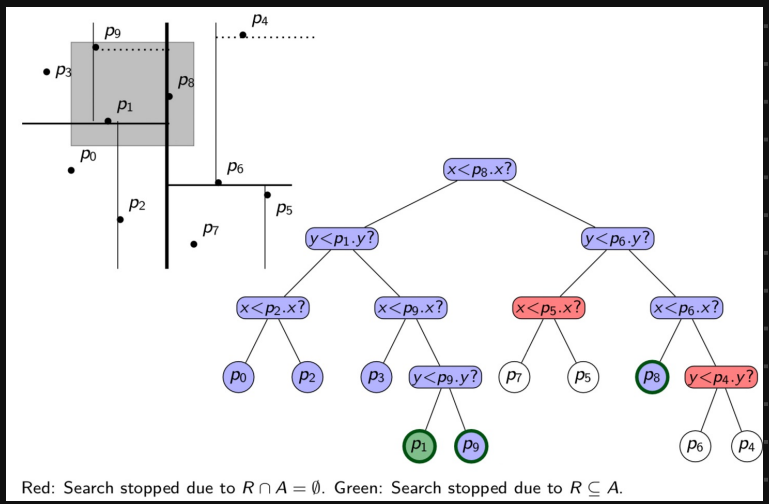
total #nodes $\leq n + (n-1)$
$\leq 2n - 1$

- **search** is like BST using indicated coord
- **insert** is search, then insert as new leaf
- **delete** is search, then remove leaf
- after insert / delete, split may no longer be exact median ? height is no longer guaranteed $\lceil \log n \rceil$
↳ can maintain $O(\log n)$ by occasionally rebuilding entire subtrees
↳ kd-trees don't handle insertion / deletion well

range search for kd-trees is same as quad-trees, except w/ only 2 children:

kdTree::RangeSearch($r \leftarrow root, A$)
r: The root of a kd-tree, A: Query-rectangle
1. $R \leftarrow$ region associated with node r
2. **if** ($R \subseteq A$) **then** report all points below r; **return**
3. **if** ($R \cap A$ is empty) **then return**
4. **if** (r is a leaf) **then**
5. $p \leftarrow$ point stored at r
6. **if** p is in A **return** p
7. **else return**
8. **for** each child v of r **do**
9. *kdTree::RangeSearch*(v, A)

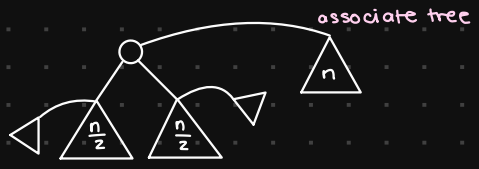
↳ assume each node stores its associated region
↳ e.g.



Red: Search stopped due to $R \cap A = \emptyset$. Green: Search stopped due to $R \subseteq A$.

- **range search complexity**:
  Cost at green node r
  $= O(\#\text{nodes in its subtree})$
  $= O(\#\text{leaves in its subtree})$
  $= \#\text{keys reported by r}$
  $= O(s)$ ← output size

  Total cost $= O(s + Q(n))$ ← worst-case #blue nodes
  $Q(n)$ satisfies recurrence rltn:
  $Q(n) \leq 2Q(\frac{n}{4}) + c, \quad Q(1) = c$
  Let $n = 4^k$? $\sqrt{n} = 2^k$
  $Q(4^k) \leq 2Q(4^{k-1}) + c$
  $\leq 2(2Q(4^{k-2}) + c) + c$
  $\leq 2^2 Q(4^{k-2}) + 2c + c$
  $\leq 2^3 Q(4^{k-3}) + 2^2 c + 2c + c$
  $\vdots$
  $\leq 2^k Q(4^{k-k} = 1) + 2^{k-1} c + \ldots + 2c + c$
  $\leq c(\frac{2^k - 1}{2-1})$
  $\leq c(2^k - 1)$
  $\leq c(\sqrt{n} - 1)$
  Thus, $Q(n) \in O(\sqrt{n})$. So, complexity of range search in kd-trees is $O(s + \sqrt{n})$
- kd-tree for **d-dimensional space**:
  - ↳ at root, pt set is partitioned based on 1st coord
  - ↳ at subtrees of root, partition is based on 2nd coord
  - ↳ at depth d-1, partition is based on last coord
  - ↳ at depth d, start all over ? partition based on 1st coord
  - ↳ **storage**:
    Assume that each node in main BST is x-median.

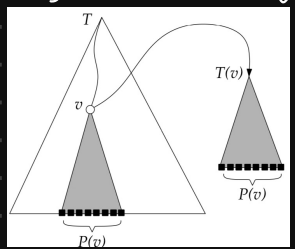


The two subtrees handle 2 dimensions while the associate tree handles remaining dimensions.
$S(n) = 2S(\frac{n}{2}) + \Theta(n)$
$S(n) \in \Theta(n \log n)$

- ↳ height is $O(\log n)$
- ↳ construction time is $O(n \log n)$
- ↳ range search time is $O(s + n^{1 - \frac{1}{d}})$
  - assume d is constant

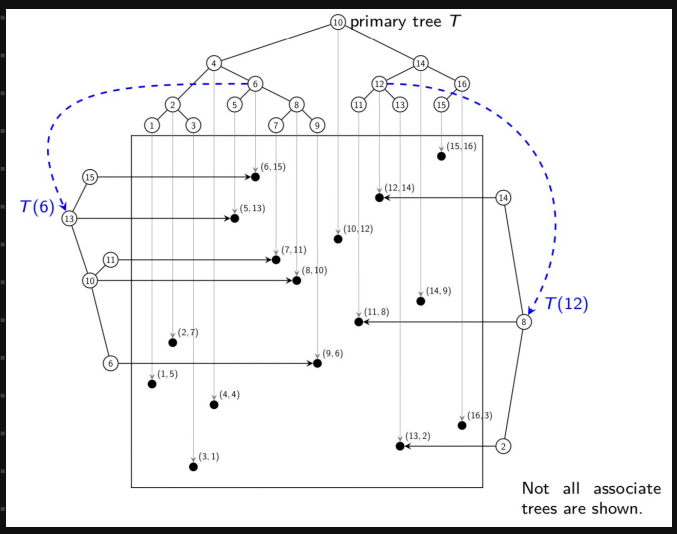
RANGE TREES

- range trees are somewhat wasteful in space, but much faster range search
  - ↳ **tree of trees**
- primary struct of **2D range tree** is balanced BST T that stores P ? uses x-coords as keys
  - ↳

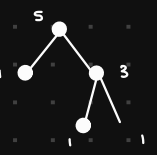


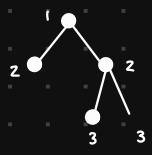
↳ every node v of T stores **associate structure** $T(v)$
- $P(v)$ is all pts in subtree of v in T, including pt at v
- $T(v)$ stores $P(v)$ in balanced BST, using y-coords as key
- v is not necessarily root of $T(v)$

· e.g.



· primary tree uses $O(n)$ space
· associate tree $T(v)$ uses $O(|P(v)|)$ space
↳ $P(v)$ are pts at descendants of v in T
↳ $\sum_v |P(v)|$

$= \sum_v$ 

$= \sum_v$ 

$= \sum_v$ #ancestors of v (same as #times it appears in associate trees)

$= \sum_v O(\log n)$
$= O(n \log n)$
- $w \in P(v)$ means v is ancestor of w in T
- every node w has $O(\log n)$ ancestors in T so it belongs to $O(\log n)$ sets $P(v)$

· range tree w/ n pts uses **$O(n \log n)$ space**

· **search**: search by x-coord in T
· **insert**:
↳ insert pt by x-coord into T
↳ walk back up to root & insert pt by y-coord in all associate trees $T(v)$ of node v on path to root
· **delete**: analogous to insertion
· want to make BSTs balanced so must completely rebuild highly unbalanced subtrees
· **range-search**: search by x-range in T & among found pts, search by y-range in some associated trees
· 1D range search pseudocode:
↳ keys are reported in-order

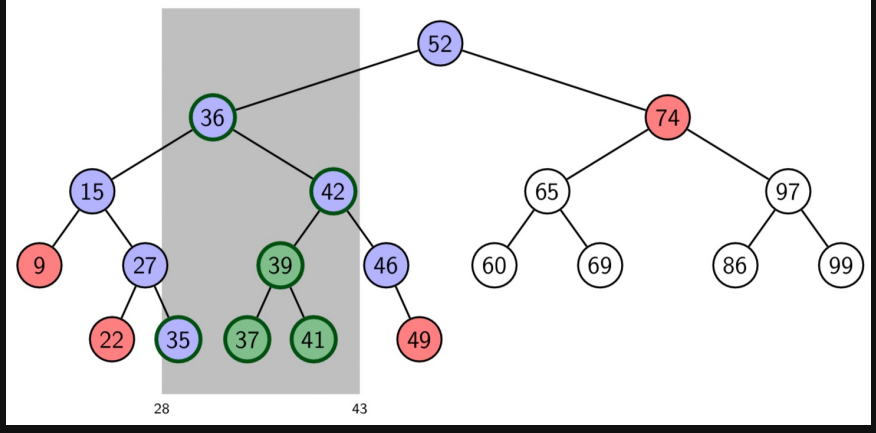
↳

BST::RangeSearch-recursive$(r \leftarrow root, x_1, x_2)$
r: root of a binary search tree, x_1, x_2: search keys
Returns keys in subtree at r that are in range $[x_1, x_2]$

```
1.  if r = NIL then return
2.  if x1 ≤ r.key ≤ x2 then
3.      L ← BST::RangeSearch-recursive(r.left, x1, x2)
4.      R ← BST::RangeSearch-recursive(r.right, x1, x2)
5.      return L ∪ r.{key} ∪ R
6.  if r.key < x1 then
7.      return BST::RangeSearch-recursive(r.right, x1, x2)
8.  if r.key > x2 then
9.      return BST::RangeSearch-recursive(r.left, x1, x2)
```

- e.g.

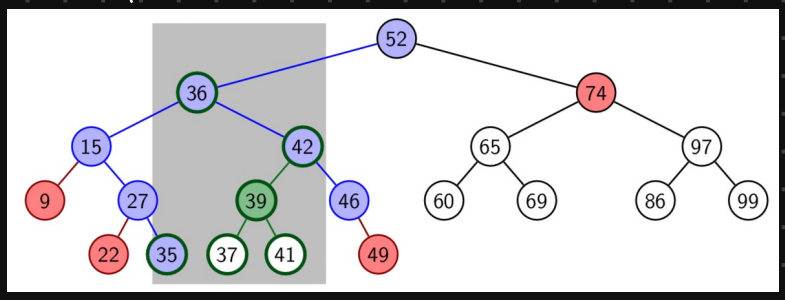
BST::RangeSearch-recursive$(T, 28, 43)$



↳ search from 39 was unnecessary b/c all of its descendants are in range

- rephrasing range search:
  - ↳ search for left boundary x_1 ⇒ get path P_1
  - ↳ search for right boundary x_2 ⇒ get path P_2
  - ↳ partitions T into 3 groups
    - outside paths
    - on paths
    - btwn paths

↳

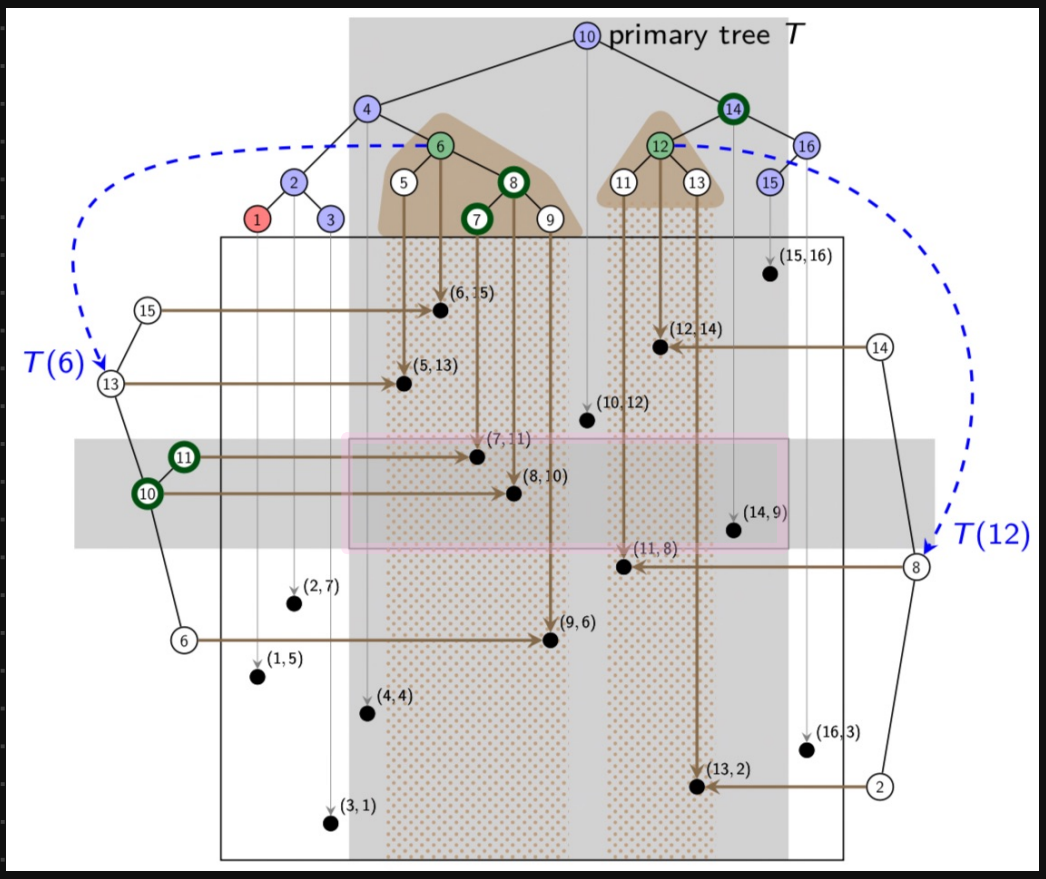


- boundary nodes are in P_1 / P_2
  - → for each node, test whether it's in range
- outside nodes are left of P_1 / right of P_2
  - → not in range ⇒ don't visit
- inside nodes are right of P_1 & left of P_2
  - → keep list of topmost inside nodes b/c all its descendants are in range

- assuming BST is balanced, range search analysis:
  - ↳ search for path P_1 is $O(\log n)$
  - ↳ search for path P_2 is $O(\log n)$
  - ↳ $O(\log n)$ boundary nodes

- spend $O(1)$ time on each
  - ↳ spend $O(1)$ time per topmost inside node v
    - takes $O(\log n)$ time b/c they're children of boundary nodes so this takes $O(\log n)$ time
  - ↳ for 1D range search, report descendants of v
    - have $\sum_{v \text{ topmost inside}} \#(\text{descendants of } v) \leq s$ since subtrees of topmost inside nodes are disjoint
      - → s is #nodes inside range
  - ↳ run-time for 1D range search is $O(\log n + s)$

- range search for $A = [x_1, x_2] \times [y_1, y_2]$ is 2-stage process
  - ↳ perform range search on x-coords for $[x_1, x_2]$ in primary tree T
  - ↳ get boundary & topmost inside nodes
  - ↳ for every boundary node, test if corresponding pt is within region A
  - ↳ for every topmost inside node v:
    - let $P(v)$ be pts in subtree v of T
      - → $P(v)$ is stored in $T(v)$
    - perform range search on $T(v)$ w/ $[y_1, y_2]$
  - ↳ e.g



- run-time for 2D range search:
  - ↳ takes $O(\log n)$ time to find boundary & topmost inside nodes in primary tree
  - ↳ there's $O(\log n)$ nodes
  - ↳ $O(\log n + s_v)$ time for each topmost inside node v
    - s_v is #pts in $T(v)$ reported
  - ↳ 2 topmost inside nodes have no common pt in their trees b/c every pt is reported in at most 1 associate tree
    - $\sum_v s_v \leq s$
  - ↳ overall run-time is $\sum_v (\log n + s_v) \in O(\log^2 n + s)$
- range trees generalized to d-dimensional spaces:
  - ↳ space is $O(n (\log n)^{d-1})$
  - ↳ construction time is $O(n (\log n)^d)$
  - ↳ range search time is $O(s + (\log n)^d)$

STRING MATCHING

INTRODUCTION

- search for **pattern** $P[0 \ldots m-1]$ in body of **text** $T[0 \ldots n-1]$
- strs are over **alphabet** Σ
- return 1st occurrence of P in T
  - ↳ i.e. smallest i st $P[j] = T[i+j]$ for $0 \leq j \leq m-1$
  - ↳ else return FAIL
- pattern matching algos consist of guesses & checks
  - ↳ **guess** / shift is pos i st P might start at $T[i]$
    - initial valid guesses are $0 \leq i \leq n-m$
  - ↳ **check** of guess is pos j w/ $0 \leq j < m$ where we compare $T[i+j]$ to $P[j]$
    - must perform m checks of single correct guess, but may make fewer ones of incorrect guess

brute-force algo:

Bruteforce::patternMatching$(T[0..n-1], P[0..m-1])$
T: String of length n (text), P: String of length m (pattern)

```
1.  for i ← 0 to n − m do
2.      if strcmp(T[i..i+m−1], P) = 0
3.          return "found at guess i"
4.  return FAIL
```

Note: *strcmp* takes $\Theta(m)$ time.

strcmp$(T[i..i+m-1], P[0..m-1])$

```
1.  for j ← 0 to m − 1 do
2.      if T[i + j] is before P[j] in Σ then return -1
3.      if T[i + j] is after P[j] in Σ then return 1
4.  return 0
```

↳ e.g.

Example: T = abbbababbab, P = abba

	a	b	b	b	a	b	a	b	b	a	b
a	a	b	b	a							
		a									
			a								
				a							
					a	b	b				
						a					
							a	b	b	a	

↳ worst case is when $P = a^{m-1}b$ & $T = a^n$
- run-time is $\Theta((n-m) \cdot m) = \Theta(mn)$

KARP-RABIN ALGORITHM

- compute **fingerprint** (hash fcn) for each guess
- if diff then P's fingerprint, then guess can't be occurrence so no need to do str compare
- e.g.

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

- Use standard hash-function: flattening + modular (radix $R = 10$):

$$h(x_0 \ldots x_4) = (x_0 x_1 x_2 x_3 x_4)_{10} \bmod 97$$

- $h(P) = 59265 \bmod 97 = 95$.

3	1	4	1	5	9	2	6	5	3	5
hash-value 84										
	hash-value 94									
		hash-value 76								
			hash-value 18							
				hash-value 95						

simple algo:

```
Karp-Rabin-Simple::patternMatching(T, P)
1.    h_P ← h(P[0..m−1])
2.    for i ← 0 to n − m
3.        h_T ← h(T[i..i+m−1])
4.        if h_T = h_P
5.            if strcmp(T[i..i+m−1], P) = 0
6.                return "found at guess i"
7.    return FAIL
```

↳ h(T[i.. i+m-1]) depends on m chars so naive computation takes θ(m) per guess
↳ run-time is θ(mn) if P not in T

- to improve algo, update fingerprints in constant time
  ↳ use prev hash to compute next one
  ↳ O(1) time per hash except 1st one
  ↳ e.g.
  Pre-compute: $10000 \bmod 97 = 9$
  Prev hash: $41592 \bmod 97 = 76$
  Next hash: $15926 = (41592 - 4 \cdot 10000) \cdot 10 + 6$
  $= ((41592 \bmod 97 - 4 \cdot 10000 \bmod 97) \cdot 10 + 6) \bmod 97$
  $= ((76 - 4 \cdot 9) \cdot 10 + 6) \bmod 97$
  $= 18$

- Karp-Rabin algo:

```
Karp-Rabin-RollingHash::patternMatching(T, P)
1.    M ← suitable prime number
2.    h_P ← h(P[0..m−1])
3.    h_T ← h(T[0..m−1])
4.    s ← 10^(m−1) mod M
5.    for i ← 0 to n − m
6.        if h_T = h_P
7.            if strcmp(T[i..i+m−1], P) = 0
8.                return "found at guess i"
9.        if i < n − m // compute hash-value for next guess
10.           h_T ← ((h_T − T[i] · s) · 10 + T[i+m]) mod M
11.   return "FAIL"
```

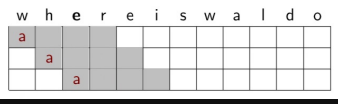
↳ M is random prime in $\{2, \ldots, mn^2\}$
↳ expected time is O(m+n)
↳ worst case is O(mn), which is extremely unlikely

BOYER-MOORE ALGORITHM

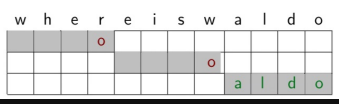
- fastest pattern matching on English text
- reverse-order searching: compare P w/ guess moving bwd
- when mismatch occurs:
  ↳ bad char jumps elim guesses based on mismatched chars of T
  ↳ good suffix jumps elim guesses based on matched suffix of P
- fwd vs reverse searching:

P: aldo
T: whereiswaldo

Forward-searching:

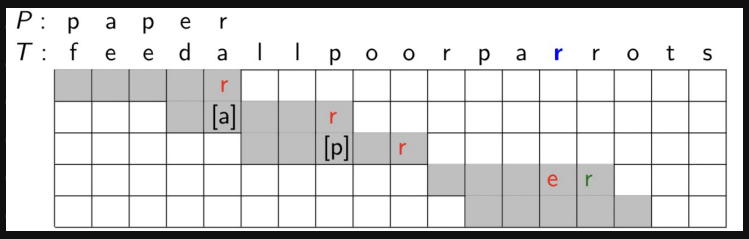


Reverse-searching:



↳ w/ fwd-searching, no guesses ruled out

- **bad char heuristic** works w/ rev-searching
  ↳ e.g.



1) mismatched char is a so shift guess until a in P aligns w/ a in T
2) mismatched char is p so shift guess until last p in P aligns w/ p in T
3) mismatched char is o, which doesn't appear in P, so shift completely past it
4) mismatched char is r, which is last char in P so it doesn't shift guess fwd at all
   - shift by 1 unit

- **last-occurrence arr** L maps Σ to ints
  ↳ L[c] is largest index i st P[i] = c
  - if c isn't in P, set L[c] = -1

  ↳ e.g.

Pattern:

0	1	2	3	4
p	a	p	e	r

Last-Occurrence Array:

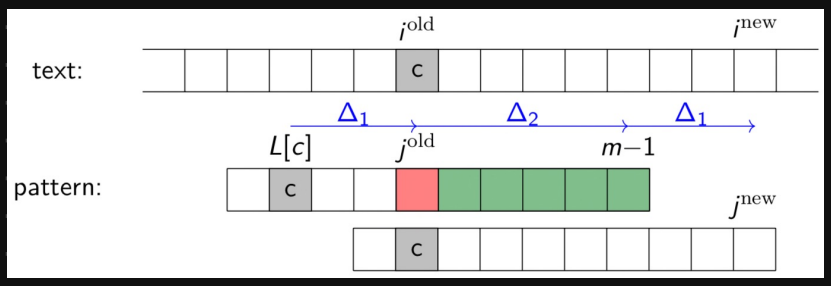
char	p	a	e	r	all others
$L[\cdot]$	2	1	3	4	−1

↳ pseudocode:

```
BoyerMoore::lastOccurrenceArray(P[0..m−1])
1.  initialize array L indexed by Σ with all −1
2.  for j ← 0 to m−1 do L[P[j]] ← j
3.  return L
```

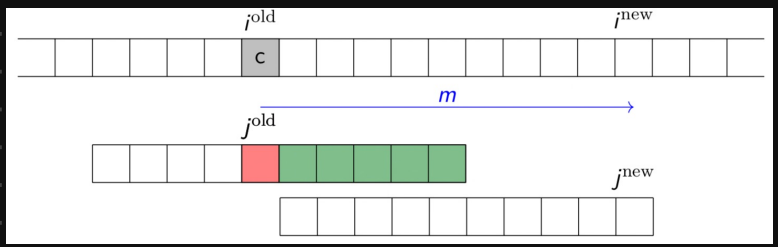
  - run-time is $O(m + |\Sigma|)$

- diff cases of bad char heuristic formula when updating L at mismatch:
  ↳ **good case** is when L[c] < j so c is left of P[j]



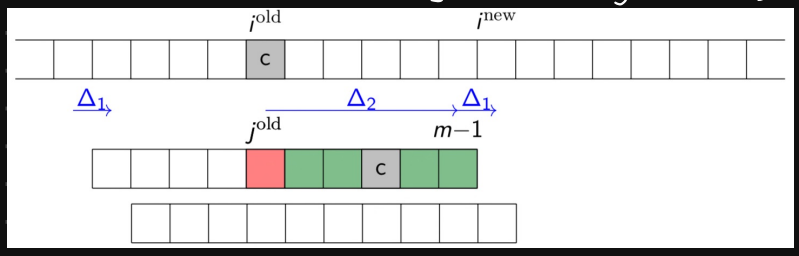
  - i^{old} & j^{old} is where mismatch occurred
  - $j^{new} = m - 1$
    → restart search from right end
  - $i^{new} = i^{old} + \Delta_2 + \Delta_1$
    $i^{new} = i^{old} + (m-1) - L[c]$
    → Δ_1 = amount we should shift = $j^{old} - L[c]$
    → Δ_2 = how much we compared = $(m-1) - j^{old}$

  ↳ **1st bad case** is when c doesn't occur in P



- shift past $T[i^{old}]$ so $i^{new} = i^{old} + m$
- $i^{new} = i^{old} + (m-1) - L[c]$
  - → $L[c] = -1$

↳ 2nd bad case is when $L[c] > j$ so c is right of $P[j]$



- bad case heuristic isn't helpful so shift by $\Delta_1 = 1$ units
- $i^{new} = i^{old} + \Delta_2 + \Delta_1$
  $= i^{old} + 1 + (m-1) - j^{old}$

↳ unified formula for all cases is $i^{new} = i^{old} + (m-1) - \min\{L[c], j^{old} - 1\}$

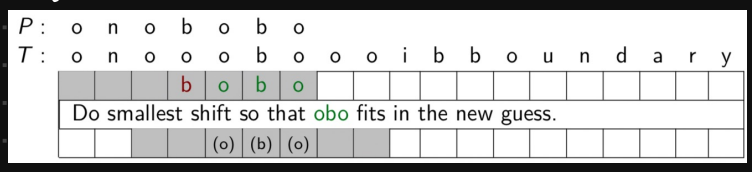
- Boyer-Moore algo:

```
Boyer-Moore::patternMatching(T,P)
1.   L ← lastOccurrenceArray(P)
2.   S ← good suffix array computed from P
3.   i ← m − 1,      j ← m − 1
4.   while i < n and j ≥ 0 do
         // current guess begins at index i − j
5.       if T[i] = P[j]
6.           i ← i − 1
7.           j ← j − 1
8.       else
9.           i ← i + m−1 − min{L[T[i]], j−1}
10.          j ← m − 1
11.  if j = −1 return "found at T[i+1..i+m]"
12.  else return FAIL
```

↳ line 9 is $i \leftarrow i + m - 1 - \min\{L[T[i]], S[j]\}$ if good suffix heuristic is used

- when using good suffix heuristic, $S[j]$ expresses since $P[j+1 \ldots m-1]$ was matched, how much to shift?

↳ e.g.

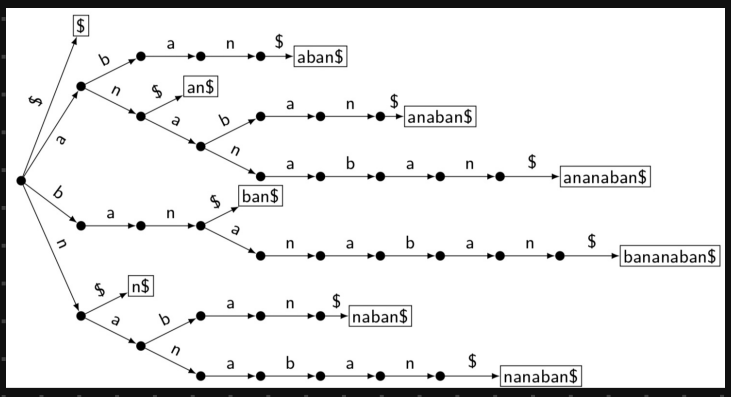


- on typical English text, only looks at apx 25% of T
- worst case run-time is $O(mn)$, but in practice much faster

SUFFIX TREES

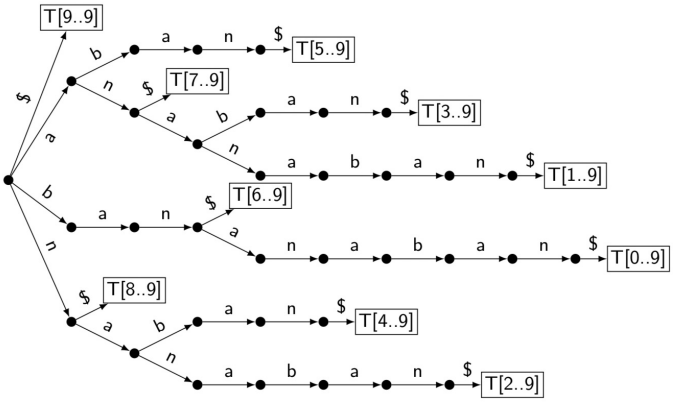
- to search for many patterns P within same fixed text T, use suffix tree
  - ↳ preprocess T instead of P
  - ↳ P is a substr of T iff P is prefix of some subfix of T
  - ↳ store all of T's suffixes in trie
    - to save space:
      - → compressed trie
      - → store suffixes implicitly via indices into T
- e.g.

T = bananaban has suffixes
{bananaban, ananaban, nanaban, anaban, naban, aban, ban, an, n, Λ}



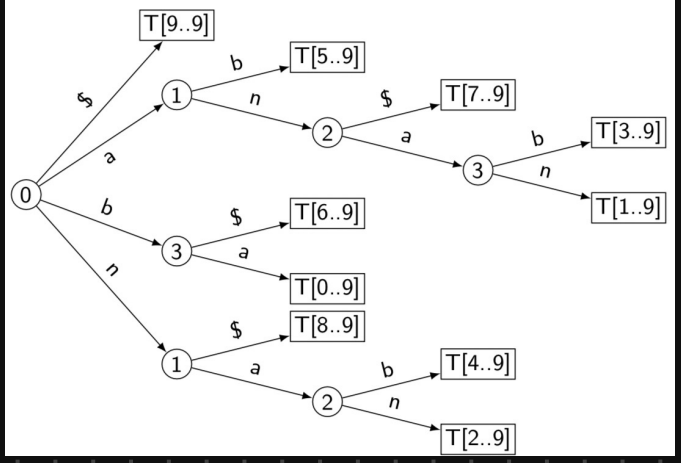
Store suffixes via indices:

	0	1	2	3	4	5	6	7	8	9
$T =$	b	a	n	a	n	a	b	a	n	$



Suffix tree: Compressed trie of suffixes

	0	1	2	3	4	5	6	7	8	9
$T =$	b	a	n	a	n	a	b	a	n	$



- T has n chars ⇒ n+1 suffixes
  - ↳ build suffix tree by inserting each suffix into compressed trie
    - run-time of $\Theta(n^2|\Sigma|)$
- pattern matching is searching for P in compressed trie
  - ↳ may need changes since P may be only prefix of stored word
  - ↳ run-time of $O(|\Sigma|m)$

SUFFIX ARRAYS

- store sorting permutation of suffixes of T
  - ↳ e.g.

Text T:

	0	1	2	3	4	5	6	7	8	9
	b	a	n	a	n	a	b	a	n	$

i	suffix $T[i..n-1]$
0	bananaban$
1	ananaban$
2	nanaban$
3	anaban$
4	naban$
5	aban$
6	ban$
7	an$
8	n$
9	$

→ sort lexicographically

j	$A^s[j]$	
0	9	$
1	5	aban$
2	7	an$
3	3	anaban$
4	1	ananaban$
5	6	ban$
6	0	bananaban$
7	8	n$
8	4	naban$
9	2	nanaban$

Suffix array:

	0	1	2	3	4	5	6	7	8	9
	9	5	7	3	1	6	0	8	4	2

- construct using MSD-Radix-Sort
  - ↳ fast in practice b/c suffixes are unlikely to share many leading chars
  - ↳ worst case run-time is $\Theta(n^2)$
    - n recursions b/c there's n chars
    - each round takes $\Theta(n)$ time

- to improve run-time:
  - ↳ subarrs after 1 round have same leading char & ties are broken by rest of words
  - ↳ these rest of words are also suffixes so they've sorted elsewhere
  - ↳ double length of sorted part every round
  - ↳ O(logn) rounds suffice so run-time becomes **O(nlogn)**
- to do **pattern matching**, apply bin search
  - ↳ e.g.

P = ban:

	j	$A^s[j]$	$T[A^s[j]..n-1]$
$\ell_1 \longrightarrow$	0	9	$
	1	5	aban$
	2	7	an$
	3	3	anaban$
$v_1 \longrightarrow$	4	1	ananaban$
$\ell_2 \rightarrow \nu{=}\ell \rightarrow$	5	6	ban$ found
$r \rightarrow$	6	0	bananaban$
$v_2 \longrightarrow$	7	8	n$
	8	4	naban$
$r_2 \longrightarrow r_1 \longrightarrow$	9	2	nanaban$

  - ↳ O(logn) comparisons
    - O(m) time per comparison
  - ↳ run-time is **O(mlogn)**
  - ↳ pseudocode:

```
SuffixArray::patternMatching(T, P, A^s[0...n−1])
A^s: suffix array of T
1.   ℓ ← 0, r ← n − 1
2.   while (ℓ < r)
3.       ν ← ⌊(ℓ+r)/2⌋
4.       i ← A^s[ν]                          // Suffix is T[i..n−1]
5.       s ← strcmp(P, T[i..i+m−1])
6.           // Assuming strcmp handles "out of bounds" suitably
7.       if (s > 0) do ℓ ← ν + 1
8.       else if (s < 0) do r ← ν − 1
9.       else return "found at guess T[i..i+m−1]"
10.  if strcmp(P, T[A^s[ℓ]..A^s[ℓ]+m−1]) = 0
11.      return "found at guess T[A^s[ℓ]..A^s[ℓ]+m−1]"
12.  return FAIL
```

- **summary** of str matching algo complexities:

	Brute-Force	Karp-Rabin	DFA	Knuth-Morris-Pratt	Boyer-Moore	Suffix Tree	Suffix Array
Preproc.	—	$O(m)$	$O(m\|\Sigma\|)$	$O(m)$	$O(m+\|\Sigma\|)$	$O(n^2\|\Sigma\|)$ [$O(n\|\Sigma\|)$]	$O(n \log n)$ [$O(n)$]
Search time	$O(nm)$	$O(n+m)$ expected	$O(n)$	$O(n)$	$O(n)$ or better	$O(m)$	$O(m \log n)$ [$O(m + \log n)$]
Extra space	—	$O(1)$	$O(m\|\Sigma\|)$	$O(m)$	$O(m+\|\Sigma\|)$	$O(n)$	$O(n)$