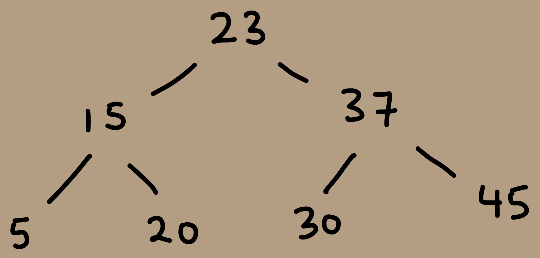
CS 240E
Personal Notes



Marcus Chan

Taught by Therese Biedl

UW CS '25

Chapter 1:
Algorithm Analysis

HOW TO "SOLVE" A PROBLEM

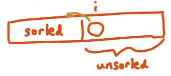
When solving a problem, we should

① Write down exactly what the problem is;

eg Sorting Problem
→ given n numbers in an array, put them in sorted order

② Describe the idea;

eg Insertion Sort



Idea:
repeatedly move one item into the correct space of the sorted part.

③ Give a detailed description; usually pseudocode.

eg Insertion Sort:

```
for i=1, ..., n-1
    j=i
    while j>0 and A[j-1]>A[j]
        swap A[j] and A[j-1]
        j--
```

④ Argue the correctness of the algorithm.

→ In particular, try to point out loop invariants & variants.

⑤ Argue the run-time of the program.

→ We want a theoretical bound. (using asymptotic notation).

To do this, we count the # of primitive operations.

PRIMITIVE OPERATIONS

In our computer model,

① our computer has memory cells
② all cells are equal
③ all cells are big enough to store our numbers

Then, "primitive operations" are $+, -, *, \div$, load & following references.

We also assume each primitive operation takes the same amount of time to run.

ASYMPTOTIC NOTATION

BIG-O NOTATION : $O(f(n))$

We say that "$f(n) \in O(g(n))$" if there exist $c>0$, $n_0>0$ s.t.

$$|f(n)| \le c|g(n)| \quad \forall n \geq n_0.$$

eg $f(n) = 75n+500$ & $g(n) = 5n^2$,
$c=1$ & $n_0=20$

Usually, "n" represents input size.

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

To show the above, we need to find c, n_0 such that $0 \le 2n^2+3n+11 \le cn^2 \quad \forall n \geq n_0$.

Sol$^{\underline{n}}$. Consider $n_0=1$. Then

$$1 \le n \Rightarrow 1 \le n^2 \Rightarrow 11 \le 11n^2$$
$$1 \le n \Rightarrow n \le n^2 \Rightarrow 3n \le 3n^2$$
$$(+) \qquad 2n^2 \le 2n^2$$
$$\Rightarrow 2n^2+3n+11 \le 11n^2+2n^2+3n^2 = \underset{?}{16n^2}$$

Hence $c=16$ & $n_0=1$, so $2n^2+3n+11 \in O(n^2)$. ⊠

Ω-NOTATION (BIG OMEGA): $f(n) \in \Omega(g(n))$

We say "$f(n) \in \Omega(g(n))$" if there exist $c>0$, $n_0>0$ such that

$$c|g(n)| \le |f(n)| \quad \forall n \geq n_0.$$

Θ-NOTATION (BIG THETA) : $f(n) \in \Theta(g(n))$

We say "$f(n) \in \Theta(g(n))$" if there exist $c_1, c_2>0$, $n_0>0$ such that

$$c_1|g(n)| \le |f(n)| \le c_2|g(n)|.$$

Note that

$$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \ \& \ f(n) \in \Omega(g(n)).$$

o-NOTATION (SMALL O) : $f(n) \in o(g(n))$

We say "$f(n) \in o(g(n))$" if for any $c>0$, there exists some $n_0>0$ such that

$$|f(n)| < c|g(n)| \quad \forall n \geq n_0.$$

If $f(n) \in o(g(n))$, we say $f(n)$ is "asymptotically strictly smaller" than $g(n)$.

ω-NOTATION (SMALL OMEGA) : $f(n) \in \omega(g(n))$

We say $f(n) \in \omega(g(n))$ if for all $c>0$, there exists some $n_0>0$ such that

$$0 \le c|g(n)| < |f(n)| \quad \forall n \geq n_0.$$

FINDING RUNTIME OF A PROGRAM

To evaluate the run-time of a program, given its pseudocode, we do the following:

① Annotate any primitive operations with just "$\Theta(1)$";

② For any loops, find the worst-case bound for how many times it will execute;

③ Calculate the big-O run time of the program;

④ Argue this bound is tight (ie show program is also in $\Omega(g(n))$, so runtime $\in \Theta(g(n))$.)

eg insertion sort

```
for i=1, ..., n-1
    j=i   Θ(1)
    while j>0 and A[j-1]>A[j]
        swap A[j] and A[j-1]   Θ(1) ] Θ(n) ] Θ(n)
        j--   Θ(1)
```

Then, let c be a const s.t. the upper bounds all the times needed to execute one line.

So run-time $\leq n \cdot n \cdot c = c \cdot n^2 \in O(n^2)$.

Next, consider the worst pos. case of insertion sort.

For each $A[i]$, we need $i-1$ swaps.

So

$$\text{runtime} \geq \sum_{i=1}^{n-1} (i-1) = \sum_{i=1}^{n-2} i = \frac{(n-2)(n-1)}{2} \in \Omega(n^2),$$

and so

$$\text{runtime} \in \Theta(n^2).$$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = L < \infty \Rightarrow f(n) \in O(g(n))$

<<LIMIT RULE I>> (L1.1(2))

Let $f(n), g(n)$ be such that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = L < \infty$.

Then necessarily $f(n) \in O(g(n))$.

Proof. We know $\lim_{n\to\infty} \frac{f(n)}{g(n)} = L$.

$\Rightarrow \forall \varepsilon > 0 : \exists n_\varepsilon$ s.t. $\left|\frac{f(n)}{g(n)} - L\right| < \varepsilon \ \forall n \geq n_\varepsilon$.

We want to show

$\exists c > 0, \exists n_0 \ni \forall n \geq n_0, f(n) \leq c \cdot g(n)$.

Choose $\varepsilon = 1$. Then there exists a n_1 s.t.

$$\forall n \geq n_1, \quad \left|\frac{f(n)}{g(n)} - L\right| \leq 1.$$

$$\Leftrightarrow \frac{f(n)}{g(n)} - L \leq \left|\frac{f(n)}{g(n)} - L\right| \leq 1.$$

$$\Leftrightarrow \frac{f(n)}{g(n)} \leq L + 1$$

$$\Leftrightarrow f(n) \leq Lg(n) + g(n) \quad (\text{since } g(n) > 0)$$

Choose $c = L+1$. Note $f(n), g(n) > 0$, so $L+1 > 0$, and since $L < \infty$, thus $c < \infty$.

Now, for all $n \geq n_1$, $f(n) \leq c \cdot g(n)$, and so $f(n) \in O(g(n))$. ☒

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \overset{\Leftrightarrow}{=} f(n) \in o(g(n))$

<<LIMIT RULE II>> (L1.1(1))

Let $f(n), g(n)$. Then $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ iff $f(n) \in o(g(n))$.

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = L > 0 \Rightarrow f(n) \in \Omega(g(n))$

<<LIMIT RULE III>> (L1.1(3))

Let $f(n), g(n)$ such that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = L > 0$.

Then necessarily $f(n) \in \Omega(g(n))$.

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) \in \omega(g(n))$

<<LIMIT RULE IV>> (L1.1(4))

Let $f(n), g(n)$ such that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Then necessarily $f(n) \in \omega(g(n))$.

OTHER LIMIT RULES

The following are corollaries of the limit rules:

① $f(n) \in \Theta(f(n))$	(Identity)
② $K \cdot f(n) \in \Theta(f(n)) \ \forall K \in \mathbb{R}$	(Constant multiplication)
③ $f(n) \in O(g(n)),\ g(n) \in O(h(n))$ $\Rightarrow f(n) \in O(h(n))$ ④ $f(n) \in \Omega(g(n)),\ g(n) \in \Omega(h(n))$ $\Rightarrow f(n) \in \Omega(h(n))$	(Transitivity)
⑤ $f(n) \in O(g(n)),\ g(n) \leq h(n) \ \forall n \geq N$ $\Rightarrow f(n) \in O(h(n))$ ⑥ $f(n) \in \Omega(g(n)),\ g(n) \geq h(n) \ \forall n \geq N$ $\Rightarrow f(n) \in \Omega(h(n))$	(Dominance)
⑦ $f_1(n) \in O(g_1(n)),\ f_2(n) \in O(g_2(n))$ $\Rightarrow f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$ ⑧ $f_1(n) \in \Omega(g_1(n)),\ f_2 \in \Omega(g_2(n))$ $\Rightarrow f_1(n) + f_2(n) \in \Omega(g_1(n) + g_2(n))$	(Addition)
⑨ $h(n) \in O(f(n) + g(n))$ $\Rightarrow h(n) \in O(\max(f(n), g(n)))$	(Maximum-rule for O)
⑩ $h(n) \in \Omega(f(n) + g(n))$ $\Rightarrow h(n) \in \Omega(\max(f(n), g(n)))$	(Maximum-rule for Ω)

$f(n) \in P_d(\mathbb{R}) \Rightarrow f(n) \in \Theta(n^d)$ <<POLYNOMIAL RULE>>

Let $f(n) \in P_d(\mathbb{R})$, ie of the form $f(n) = c_0 + c_1 n + \cdots + c_d n^d$.

Then necessarily $f(n) \in \Theta(n^d)$.

$b > 1$; $\log_b(n) \in \Theta(\log n)$ <<LOG RULE I>>

*convention:

"log" = "$\log_2$".

Let $b > 1$. Then necessarily $\log_b(n) \in \Theta(\log n)$.

Proof. Note

$$\lim_{n\to\infty} \frac{\log_b(n)}{\log n} = \lim_{n\to\infty} \frac{\left(\frac{\log(n)}{\log(b)}\right)}{\log(n)} = \lim_{n\to\infty} \frac{1}{\log(b)} > 0,$$

So by Limit Rules 1 & 3, $\log_b(n) \in \Theta(\log n)$. ☒

$c, d > 0$; $\log^c n \in o(n^d)$ <<LOG RULE II>>

Let $c, d > 0$. Then necessarily $\log^c n \in o(n^d)$,

where $\log^c n \equiv (\log n)^c$.

Proof. See that

$$\lim_{n\to\infty} \frac{\ln^k n}{n} \overset{L'H}{=} \lim_{n\to\infty} \frac{k \ln^{k-1}(n) \cdot \frac{1}{n}}{1} \overset{L'H}{=} \cdots \overset{L'H}{=} \lim_{n\to\infty} \frac{k!}{n} = 0,$$

so $\ln^k n \in o(n)$.

Fix $c, d > 0$. Then

$$\lim_{n\to\infty} \frac{\ln^c n}{n^d} = \left(\lim_{n\to\infty} \frac{\ln^{\frac{c}{d}} n}{n}\right)^d \leq \left(\lim_{n\to\infty} \frac{\ln^{\lceil \frac{c}{d} \rceil} n}{n}\right)^d = 0^d = 0.$$

As $\log^c n = \left(\frac{1}{\ln 2}\right)^c \ln^c n$, thus $\lim_{n\to\infty} \frac{\log^c n}{n^d} = \left(\frac{1}{\ln 2}\right)^c \cdot 0 = 0$.

Proof follows from the limit rule. ☒

$f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$

Suppose $f(n) \in o(g(n))$. Then $f(n) \in O(g(n))$.

$f(n) \in o(g(n)) \Rightarrow f(n) \notin \Omega(g(n))$

Suppose $f(n) \in o(g(n))$. Then $f(n) \in \Omega(g(n))$.

Proof. Prove by contra tve: '

$f(n) \in \Omega(g(n)) \Rightarrow f(n) \notin o(g(n))$.

Consider cases for $\lim_{n\to\infty} \frac{f(n)}{g(n)}$.

Case 1 It DNE.

$\Rightarrow f(n) \notin o(g(n))$.

Case 2 Limit exists.

Then by $f(n) \in \Omega(g(n))$, thus

$f(n) \geq c \cdot g(n)$ for some $c > 0$ & $n \geq n_0$.

$\Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c > 0$

$\Rightarrow$ limit $\neq 0$

$\Rightarrow f(n) \notin o(g(n))$. ☒

$f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$

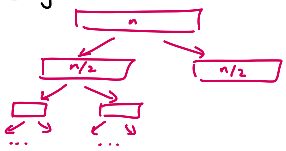
Suppose $f(n) \in \omega(g(n))$. Then $f(n) \in \Omega(g(n))$.

WORST-CASE RUNTIME: $T_A^{worst}(n)$

The "worst-case runtime" for an algorithm, denoted $T_A^{worst}(n)$, is the max run-time among all instances of size n.

ANALYZING RECURSIVE ALGORITHMS

Consider merge sort:



Analysis of MergeSort:

```
MergeSort(A, n, ℓ←0, r←n-1, S←NIL)
A: array of size n,   0≤ℓ≤r≤n-1
    if S is NIL    init it as array S[0...n-1]
    if (r≤ℓ) then
        return
    else
        m = (r+ℓ)/2
        MergeSort(A, n, ℓ, m, S)
        MergeSort(A, n, m+1, r, S)
        Merge(A, ℓ, m, r, S)

Merge(A, ℓ, m, r, S)
A[0,...,n-1] is an array. A[ℓ,...,m] is sorted,
A[m+1,...,r] is sorted, S[0,...,n-1] is an array
    copy A[ℓ...r] into S[ℓ...r]
    int i_L←ℓ ; int i_R←m+1
    for (k←ℓ; k≤r; k++) do
        if (i_L>m) A[k]←S[i_R++]
        else if (i_R>r) A[k]←S[i_L++]
        else if (S[i_L]≤S[i_R]) A[k]←S[i_L++]
        else A[k]←S[i_R++]
```

Arguing run-time:

Let $T(n)$ = run-time of merge sort with n items.

$$\therefore T(n) \leq \begin{cases} c, & n \leq 1 \\ 2 \cdot T(\frac{n}{2}) + c \cdot n, & \text{otherwise} \end{cases}$$

$\in \Theta(n \log n)$ (see below)

SOME RECURRENCE RELATIONS

Note:

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
$T(n) = T(cn) + \Theta(n),\ 0<c<1$	$T(n) \in \Theta(n)$	Selection
$T(n) = 2T(n/4) + \Theta(1)$	$T(n) \in \Theta(\sqrt{n})$	Range Search
$T(n) = T(\sqrt{n}) + \Theta(1)$	$T(n) \in \Theta(\log\log n)$	Interpolation Search

SORTING PERMUTATION [OF AN ARRAY]

A "sorting permutation" of an array A is the permutation $\pi : \{0, \dots, n-1\} \to \{0, \dots, n-1\}$ such that

$$A[\pi(0)] \leq A[\pi(1)] \leq \cdots \leq A[\pi(n-1)].$$

* $A \circ \pi$ would be "sorted".

eg if $A = [14, 3, 2, 6, 1, 11, 7]$, then

$\pi = \{4, 2, 1, 3, 6, 5, 0\}$.

For a sorting permutation π, we define its inverse π^{-1} to be the array which entries have exactly the same "relative order" as in A.

eg $\pi^{-1} = [6, 2, 1, 3, 0, 5, 4]$ (if A is as above)

We denote "Π_n" to be the set of all sorting permutations of $\{0, \dots, n-1\}$.

AVERAGE-CASE RUNTIME: $T_A^{avg}(n)$

The "average-case run-time" of an algorithm is defined to be

$$\boxed{T_A^{avg}(n) := \underset{I \in \mathcal{I}_n}{\text{avg}}\ T_A(I) = \frac{1}{|\mathcal{I}_n|} \sum_{I \in \mathcal{I}_n} T_A(I),}$$

where $\mathcal{I}_n$ is the set of instances of size n.

In particular, if we can map each instance I to a permutation $\pi \in \Pi_n$, where Π_n is the set of all permutations of $\{0, \dots, n-1\}$, then we may alternatively state that

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

where $T(\pi)$ is the number of comparisons on instance π^{-1}.

EXAMPLE: AVGCASEDEMO

Consider the algorithm

```
avgCaseDemo(A, n)
// array A stores n distinct numbers
if $n \le 2$ return
if $A[n-2] \le A[n-1]$ then
    avgCaseDemo($A[0 \dots \frac{n}{2}-1], \frac{n}{2}$)    // good case
else
    avgCaseDemo($A[0 \dots n-3], n-2$)    // bad case
```

We claim $T^{avg}(n) \in O(\log n)$.

Proof. To avoid constants, let $T(\cdot) :=$ # of recursions; the run-time is proportional to this.

As all numbers are distinct, we may associate each array with a sorting permutation.

So for $\pi \in \Pi_n$, let $T(\pi) =$ # of recursions done if the input array has sorting permutation π.

Note we have two kinds of permutations:

① "Good" permutations — if $A[n-2] < A[n-1]$; or

② "Bad" permutations — if $A[n-2] > A[n-1]$.

Denote

Π_n^{good} = # of good permutations of size n

& Π_n^{bad} = # of bad permutations of size n.

Then, we claim that

$$\sum_{\pi \in \Pi_n^{good}} T(\pi) \le |\Pi_n^{good}|(1 + T^{avg}(\lfloor \tfrac{n}{2} \rfloor))|$$

$$\& \sum_{\pi \in \Pi_n^{bad}} T(\pi) \le |\Pi_n^{bad}|(1 + T^{avg}(n-2))|.$$

Proof. We only prove this for good permutations; the other claim is similar.

Fix $\pi \in \Pi_n^{good}$, and let π_{half} be the permutation of the recursion;

ie π_{half} = the sorting perm of $\pi^{-1}[0, \dots, \frac{n}{2}-1]$ &

$T(\pi) = 1 + T(\pi_{half})$.

Note that $\pi_{half} \in \Pi_{\lfloor \frac{n}{2} \rfloor}$. Then see that

$$\sum_{\pi \in \Pi_n^{good}} T(\pi) = \sum_{\pi \in \Pi_n^{good}} (1 + T(\pi_{half}))$$

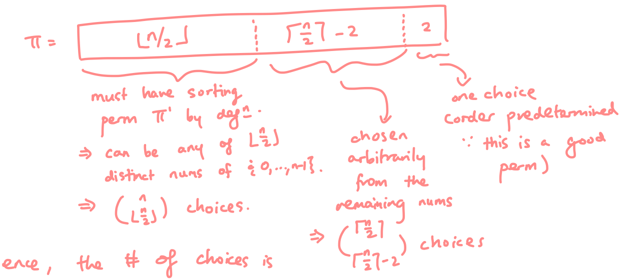
$$= |\Pi_n^{good}| + \sum_{\pi \in \Pi_n^{good}} T(\pi_{half})$$

$$= |\Pi_n^{good}| + \sum_{\pi' \in \Pi_{\lfloor \frac{n}{2} \rfloor}} |\{\pi \in \Pi_n^{good} \text{ for which } \pi_{half} = \pi'\}| \cdot T(\pi').$$

We next prove the following claim:

Claim If $n \ge 3$, then $|\Pi_n^{good}(\pi')| = \dfrac{n!}{2!(\lfloor \frac{n}{2} \rfloor)!}$ $\forall \pi' \in \Pi_{\lfloor \frac{n}{2} \rfloor}$.

Proof. Fix $\pi \in \Pi_n^{good}(\pi')$. See that



must have sorting perm π' by defn.

⇒ can be any of $\lfloor \frac{n}{2} \rfloor$ distinct nums of $\{0, \dots, n-1\}$.

⇒ $\binom{n}{\lfloor \frac{n}{2} \rfloor}$ choices.

chosen arbitrarily from the remaining nums

⇒ $\binom{\lceil \frac{n}{2} \rceil}{\lceil \frac{n}{2} \rceil - 2}$ choices

one choice (order predetermined ∵ this is a good perm)

Hence, the # of choices is

$$\binom{n}{\lfloor \frac{n}{2} \rfloor} \cdot \binom{\lceil \frac{n}{2} \rceil}{\lceil \frac{n}{2} \rceil - 2} (\lceil \tfrac{n}{2} \rceil - 2)! = \dots = \frac{n!}{\lfloor \frac{n}{2} \rfloor! \cdot 2},$$

as needed. ✗

Then, since $|\Pi_n^{good}| = \frac{1}{2}|\Pi_n| = \frac{n!}{2}$, it follows that $|\Pi_n^{good}| / |\Pi_{\lfloor \frac{n}{2} \rfloor}|$, and so

$$\sum_{\pi \in \Pi_n^{good}} T(\pi) = |\Pi_n^{good}| + \sum_{\pi' \in \Pi_{\lfloor \frac{n}{2} \rfloor}} |\Pi_n^{good}(\pi')| \cdot T(\pi')$$

$$= |\Pi_n^{good}| + \sum_{\pi' \in \Pi_{\lfloor \frac{n}{2} \rfloor}} \frac{|\Pi_n^{good}|}{|\Pi_{\lfloor \frac{n}{2} \rfloor}|} \cdot T(\pi')$$

$$= |\Pi_n^{good}| \left(1 + \frac{1}{|\Pi_{\lfloor \frac{n}{2} \rfloor}|} \sum_{\pi' \in \Pi_{\lfloor \frac{n}{2} \rfloor}} T(\pi')\right)$$

$$= |\Pi_n^{good}| (1 + T^{avg}(\lfloor \tfrac{n}{2} \rfloor))$$

as needed. #

Next, see that $|\Pi_n^{good}| = |\Pi_n^{bad}|$, since we can map any bad perm to a good one (and v.v.) by swapping $A[n-2]$ & $A[n-1]$.

Thus $|\Pi_n^{good}| = |\Pi_n^{bad}| = \frac{n!}{2}$, and so

$$T^{avg}(n) \le \frac{1}{|\Pi_n|} \sum_{I \in \Pi_n} T(I) \le \frac{1}{|\Pi_n|} \left(\sum_{I \in \Pi_n^{good}} T(I) + \sum_{I \in \Pi_n^{bad}} T(I) \right)$$

$$\le \frac{1}{|\Pi_n|} \left(|\Pi_n^{good}| (1 + T^{avg}(\lfloor \tfrac{n}{2} \rfloor)) + |\Pi_n^{bad}| (1 + T^{avg}(n-2)) \right)$$

$$= 1 + \tfrac{1}{2} T^{avg}(\lfloor \tfrac{n}{2} \rfloor) + \tfrac{1}{2} T^{avg}(n-2).$$

Finally, we show $T^{avg}(n) \le 2 \log n$ by induction.

Clearly, this holds for $n \le 2$.

So, assume $n \ge 3$. Assume the inductive hypothesis. Then see that

$$T^{avg}(n) \le 1 + \tfrac{1}{2} T^{avg}(\lfloor \tfrac{n}{2} \rfloor) + \tfrac{1}{2} T^{avg}(n-2)$$

$$\le 1 + \tfrac{1}{2}(2 \log(\lfloor \tfrac{n}{2} \rfloor)) + \tfrac{1}{2}(2 \log(n-2))$$

$$\le 1 + \log(n-1) + \log(n)$$

$$= 2 \log(n),$$

which suffices to prove the claim. ▣

EXPECTED-CASE RUNTIME: $T_{\mathcal{A}}^{exp}(n)$

The "expected-case runtime" of an algorithm is defined to be

$$\boxed{T_{\mathcal{A}}^{exp}(I) = E[T_{\mathcal{A}}(I,R)] = \sum_{\text{all } R} T_{\mathcal{A}}(I,R)\,P(R)}$$

where R is a sequence of random outcomes, and $P(R)$ denotes the probability the random variables in the algorithm $\mathcal{A}$ have outcomes R.

AMORTIZED RUN-TIME : $T_{\mathcal{O}}^{amort}$

Let $\mathcal{O}$ be an operation, and let $T^{actual}(\mathcal{O})$ be the actual run-time of $\mathcal{O}$.

Then, we say $\mathcal{O}$ has "amortized run-time $T^{amort}(\mathcal{O})$" if for any sequence of operations $\mathcal{O}_1, \dots, \mathcal{O}_k$ that could occur, we have

$$\boxed{\sum_{i=1}^{k} T^{actual}(\mathcal{O}_i) \leq \sum_{i=1}^{k} T^{amort}(\mathcal{O}_i).}$$

POTENTIAL FUNCTION [OF A DATA STRUCTURE]: Φ

A "potential function" is a function $\Phi(\cdot)$ that depends on the status of the data structure.

In particular, for any sequence $\mathcal{O}_1, \dots, \mathcal{O}_k$ of operations:

① $\Phi(i) \geq 0 \quad \forall i \geq 0$, where Φ_i = the value of Φ after $\mathcal{O}_1, \dots, \mathcal{O}_i$ have been executed; &

② $\Phi(0) = 0$.

$$T^{amort}(\mathcal{O}) := T^{actual}(\mathcal{O}) + \Phi_{after} - \Phi_{before}$$

UPPER BOUNDS ACTUAL RUN-TIME

For any potential function Φ, the function

$$\boxed{T^{amort}(\mathcal{O}) := T^{actual}(\mathcal{O}) + \Phi_{after} - \Phi_{before}}$$

upper-bounds the actual run-time, where Φ_{before} & Φ_{after} denote the state of the potential function before & after $\mathcal{O}$.

Proof. Fix a sequence of operations $\mathcal{O}_1, \dots, \mathcal{O}_k$. Summing up the amortized times and using a telescoping sum, we get

$$\begin{aligned}
\sum_{i=1}^{k} T^{amort}(\mathcal{O}_i) &= \sum_{i=1}^{k}\left(T^{actual}(\mathcal{O}_i) + \Phi(i) - \Phi(i-1)\right) \\
&= \sum_{i=1}^{k} T^{actual}(\mathcal{O}_i) + \sum_{i=1}^{k}\left(\Phi(i) - \Phi(i-1)\right) \\
&= \sum_{i=1}^{k} T^{actual}(\mathcal{O}_i) + \underbrace{\Phi(k)}_{\geq 0} - \underbrace{\Phi(0)}_{=0} \\
&\geq \sum_{i=1}^{n} T^{actual}(\mathcal{O}_i). \quad \blacksquare
\end{aligned}$$

STEPS TO PERFORM AMORTIZED ANALYSIS USING THE POTENTIAL FUNCTION METHOD

To do amortized analysis using potential functions, we do the following:

① Define a "time unit", so that an operation with run-time $\Theta(k)$ takes at most k time units.

② Define a potential function Φ and verify $\Phi(0) = 0$ & $\Phi(i) \geq 0 \quad \forall i \geq 0$.

③ For each operation $\mathcal{O}$, compute

$$T^{amort}(\mathcal{O}) := T^{actual}(\mathcal{O}) + \Phi_{after} - \Phi_{before},$$

and find an asymptotic upper bound for it.

EXAMPLE: DYNAMIC ARRAYS

Consider dynamic arrays  with two operations:

① insert; &

② rebuild, where we "lengthen" the array by a factor of 2.

Hence, insert takes $\Theta(1)$ time, & rebuild takes $\Theta(n)$ time (where n is the size of the array).

Define a "time unit" such that 'insert' takes "one unit of time" & 'rebuild' takes "n units of time".

We claim $T^{amort}(\text{insert}) = 3$ & $T^{amort}(\text{rebuild}) = 0$.

Proof. Let the potential function Φ be defined by

$$\Phi(i) = \max\{0,\ 2 \cdot \text{size} - \text{capacity}\}.$$

Clearly $\Phi(i) \geq 0$. Also initially size $= 0$ & cap ≥ 0, so $\Phi(0) = 0$ as desired.

Now, the amortized run-time for insert is

$$\begin{aligned}
T^{amort}(\text{insert}) &= T^{actual}(\text{insert}) + \Phi_{after} - \Phi_{before} \\
&\leq 3,
\end{aligned}$$

as the actual time is ≤ 1 unit, the size increases by 1 & the capacity does not change.

Similarly,

$$\begin{aligned}
T^{amort}(\text{rebuild}) &= T^{actual}(\text{rebuild}) + \Phi_{after} - \Phi_{before} \\
&\leq n + (0 - n) \\
&= 0. \quad \blacksquare
\end{aligned}$$

Chapter 2:
Priority Queues and Heaps

ADT PRIORITY QUEUES

1. A "priority queue" stores items that have a priority, or key.

2. Operations:
   - ① insert (a given item & priority as a k-v pair)
   - ② deleteMax (return item with largest priority)
   - ③ size, isEmpty

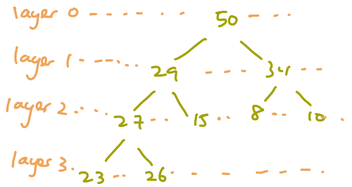
3. We can use a PQ to sort:

```
PQ-Sort (A[0, ..., n-1])
    init PQ to an empty PQ
    for i←0 to n-1 do
        PQ.insert (A[i])
    for i←n-1 down to 0 do
        A[i] ← PQ.deleteMax()
```

4. The run-time of the above algorithm is O(initialization + n·insert + n·deleteMax).

BINARY HEAPS

1. "Binary heaps" are binary trees with the following properties:
   - ① Each level is filled except the last, which is filled from the left; } "structural" property
   - ② $key(i) \leq key(parent(i))$ } "heap-order property"

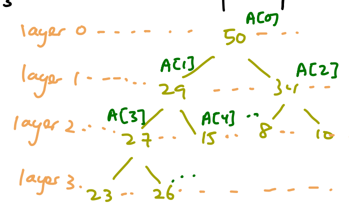


Each node is "shorthand" for a reference to a <key-value> pair:

"50" ⟷ priority = 50, data = <..>

2. Note that binary heaps have height O(log n).

3. We store binary heaps using an array:



⇒ A

0	1	2	3	4	5	..
50	29	34	27	15	8	...

Then

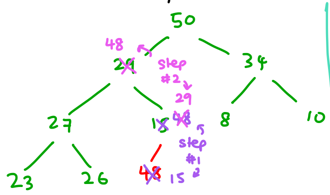
$$parent(i) = \left\lfloor \frac{i-1}{2} \right\rfloor,$$

similar implementation for left/right child.

INSERT IN HEAPS

1. To insert into a heap, we just place the new key at the first free leaf.

2. We also employ "fix-up":

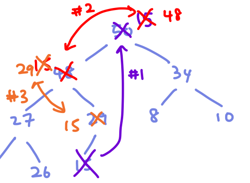


Pseudocode

```
fix-up(A,k)
k: an index corresponding to a node of the heap
    while parent(k) exists and A[parent(k)] < A[k] do
        swap A[k] & A[parent(k)]
        k ← parent(k)
```

DELETEMAX IN HEAPS

1. The maximum item of a heap is just the root node.
2. We replace the root by the last leaf, which is taken out.
3. Then, we use "fix-down":



Pseudocode

```
fix-down (A, n, k)
// A: an array that stores a heap of
      size n
// k: an index corresponding to a node
      of the heap
  while k is not a leaf do
      // find child with larger key
      j ← left child of k
      if (j is not last(n) & A[j+1] > A[j])
          j ← j+1
      if A[k] ≥ A[j] break
      swap A[j] and A[k]
      k ← j
```

* we "swap down" from the root-node until the heap-ordering is satisfied.

4. Run-time: O(height) = O(log n)

HEAPSORT

1. If we use a heap as a PQ and sort using it, the run-time of said algorithm is

$$\boxed{\begin{aligned} T(n) &\in O(\text{initialization} + n \cdot \text{inset} + n \cdot \text{deleteMax}) \\ &= O(\Theta(1) + n \cdot \Theta(\log n) + n \cdot \Theta(\log n)) \\ &= O(n \log n). \end{aligned}}$$

2. Pseudocode:

```
HeapSort (A, n)
    // heapify
    n ← A.size()
    for i ← parent(last()) down to 0 do
        fix-down (A, i, n)
    // repeatedly find maximum
    while n > 1
        // 'delete' maximum by moving to end and
           decreasing n
        swap item at A[root()] and A[last()]
        n --
        fix-down (A, root(), n)
```

* Heapify:

Given: all items that should be on the heap

It builds the heap all at once.

How? → by fixing-down incrementally.

Heapify has run-time $\underline{\underline{\Theta(n)}}$.

Why? → analyze a recursive version:

```
heapify (node i)
    if i has left child
        heapify (left child i)
    if i has right child
        heapify (right child i)
    fix-down(i)
```

Now, let

T(n) := runtime of heapify on n items.

(Assume n divisible as needed.)

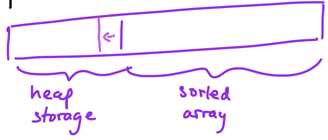
Then:

size (left child) = size (right child) = $\frac{n-1}{2}$.

$$\therefore \; T(n) = \begin{cases} \Theta(1), & n \leq 1 \\ T(\frac{n-1}{2}) + T(\frac{n-1}{2}) + \Theta(\log n), & \text{otherwise} \end{cases}$$

$$\Rightarrow T(n) \leq 2T(\tfrac{n}{2}) + \Theta(\log n) \in O(n).$$ ∎

3. Note this uses O(1) auxiliary space, since we use the same input-array A for storing the heap.



OTHER PQ OPERATIONS

We can also support the following operations:

① "findMax" — finds the max element without removing it;
- in a bin heap this takes Θ(1)
- since it is just the root node

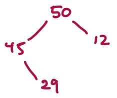
② "decreaseKey" — takes in a ref i to the location of one item of the heap and a key k_{new}, and decreases the key of i to k_{new} if $\underline{k_{new} < key(i)}$
- does nothing if $k_{new} \geq key(i)$
- easy to do in a bin heap; just need to change the key & then call fix-down on i to its children

③ "increaseKey" — "opposite" of decreaseKey.
- easy in bin heap, but just call fix-up instead of fix-down

④ "delete" — delete the item i (which we have a ref to).
- for bin heap: increase value of key at i to ∞ (or findMax().key()+1);
- then call deleteMax
- takes O(log n)

MELDABLE HEAP

A "meldable heap" is the same as a "binary heap", except it drops the structural property.



* so just that each root is the maximum of all its children

Operations for meldable heaps:

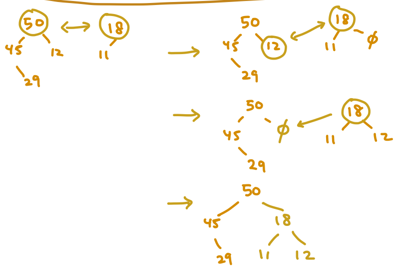
① insert
- create 1-node meldable heap P' w/ new k-v pair we want to add
- call merge(P, P') w/ existing & newly-created meldable heap

② deleteMax
- max is root, so just remove that
- then return merge(P_L, P_R), where P_L & P_R are the "subheaps" of the original heap

③ merge

```
meldableHeap::merge(r1, r2)
input: r1, r2 (roots of two meldable heaps)
  - r1 ≠ NIL
output: returns root of merged heap
    if r2 is NIL then return r1
    if r1.key < r2.key then swap r1, r2
    randomly pick one child c of r1
    replace c by result of merge(r2, c)
    return r1
```

eg



(since 18<50)

AVG RUN-TIME (MERGE) = $O(\log n_1 + \log n_2)$ (L2.3)

Note that avg run-time (merge) = $O(\log n_1 + \log n_2)$ where n_1, n_2 are the sizes of the heaps to be merged.

BINOMIAL HEAPS

FLAGGED TREES

$_1$ A "flagged tree" is one where every level is full, but the root node only has a left child.

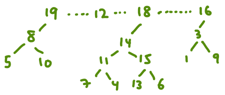
$_2$ Note that a flagged tree of height h has 2^h nodes.

BINOMIAL HEAP (D2.3)

A "binomial heap" is a list L of binary trees such that

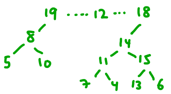
① any tree in L is a flagged tree (structural property); &

② for any node v, all keys in the left subtree of v are no bigger than v.key (order property).



PROPER BINOMIAL HEAP

We say a BH is "proper" if no two flagged trees in L have the same height.



A PBH OF SIZE n CONTAINS $\leq \log(n)+1$ FLAGGED TREES (O2.2)

Let a PBH have size n.

Then it contains at most $\log(n)+1$ flagged trees.

Proof. Let T be the tree w/ max height, say h, in the list L of flagged trees.
Then T has 2^h nodes, so $2^h \leq n$,
ie $h \leq \log(n)$.
Since the trees in L have distinct heights, we have at most one tree for each height $0, \ldots, h$, and so at most $h+1 \leq \log n+1$ trees. ☒

MAKING A BH PROPER

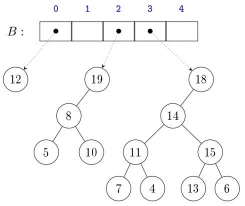
$_1$ To make a BH proper, we do the following: if the BH has two flagged trees T, T' of the same height h, then they both have 2^h nodes.

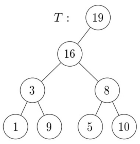
$_2$ We want to combine them into one flagged tree of height $h+1$.

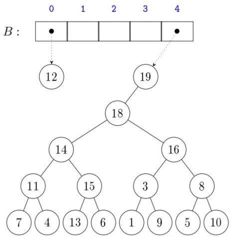
$_3$ To do so, we use the following algorithm:

(A2.11)

```
binomialHeap :: makeProper()
    n ← size of the binomial heap
    for (ℓ=0; n>1; n ← ⌊n/2⌋) do ℓ++          // compute ⌊log n⌋
    B ← array of size ℓ+1, initialized at NIL
    L ← list of flagged trees
    while L is non-empty do
        T ← L.pop(), h ← T.height
        while T' ← B[h] is not NIL do
            if T.root.key < T'.root.key then swap
            T & T'
            T'.right ← T.left, T.left ← T', T.height ← h+1     // merge T with
            B[h] ← NIL, h++
        B[h] ← T
    for (h=0; h≤ℓ; h++) do                      // copy B back to the list
        if B[h] ≠ NIL then L.append(B[h])
```







PQ OPERATIONS FOR PBHs

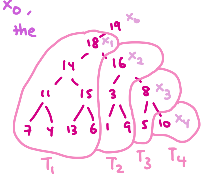
Each of the PQ operations can be performed in $O(\log n)$ time with makeProper.

① merge(P_1, P_2)
- concat lists of P_1, P_2 into one
- then call makeProper
- takes time $O(\log n_1 + \log n_2) \leq O(\log n)$

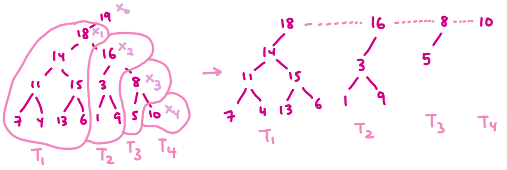
② insert(k, v)
- like w/ meldable heaps; create a single-node binomial heap, then call merge
- takes time $O(\log n)$

③ findMax()
- scan through L and compare keys of the roots
- largest is just maximum
- takes time $O(|L|) \leq O(\log n)$

④ deleteMax()
- assume we found the max at root x_0 of flagged tree T, say w/ height h (this takes $O(\log n)$ time if heights not already known)
- remove T from L and split it as follows:
- let x_1 be the left child of x_0, and for $1 \leq i \leq h$, let x_{i+1} be the right child of x_i.



- let T_i be the tree consisting of x_i & its left subtree; this is a flagged tree.
- create a second BH P' consisting of $T_1, \ldots, T_h$, and note this covers all keys in T except x_0.
- then, merge P' into what remains of the original binomial heap P.
- as P was proper, and the list of P' has length $h \leq \log n$, merging (and thus deleteMax) has run-time $O(\log n)$.



Chapter 3: Sorting

THE SELECTION PROBLEM

The "selection problem" is :

> "Given an array A[0,..., n-1] and an index $0 \leq k < n$, select(A,k) should return the element in A that would be at index k if we sorted A".

eg if A[0,...,9] = 30 60 10 0 50 80 90 10 40 70

then the sorted array would be

0 10 10 30 40 50 60 70 80 90

so

select(3) = 30.

PARTITION [THE FUNCTION]

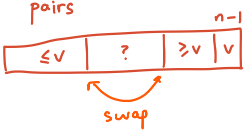
1 The "partition" function does the following: given the pivot-value $v = A[p]$ and an array A[0,.., n-1], rearrange A s.t.

and return the pivot-index i.

2 Partition algorithm ("efficient in-place partition — Hoare") :

```
partition(A,p)
  // A : array of size n
  // p : integer s.t. 0≤p≤n
1.   swap(A[n-1], A[n])
2.   i←-1,  j←n-1,  v←A[n-1]
3.   loop
4.       do i←i+1  while A[i]<v
5.       do j←j-1  while j≥1 & A[j]>v
6.       if i≥j then break (goto 9)
7.       else swap(A[i], A[j])
8.   end loop
9.   swap(A[n-1], A[i])
10.  return i
```

* we keep swapping the outer-most wrongly-positioned pairs

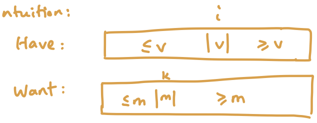


QUICK-SELECT

1 The "quick-select" algorithm:

```
quick-select1(A,k)
  // A : array of size n
  // k : integer s.t. 0<k<n
  p← choose-pivot1(A)        // for now, p=n-1
  i← partition(A,p)
  if i=k then
      return A[i]
  else if i>k then
      return quick-select1(A[0,...,i-1], k]
  else if i<k then
      return quick-select1(A[i+1, i+2,..., n-1], k-i-1)
```

* intuition:



2 Run-time analysis of quick-select:

We analyze the # of key-comparisons.

↳ so we don't mess with constants.

In particular, partition uses n key comparisons.

Then, the run-time on an instance I is

$$T(I) = \underbrace{n}_{\text{partition}} + \underbrace{T(I')}_{\text{sub array}}$$

How big is I'?

best case: don't recurse at all → $\theta(n)$

worst case: $|I'| = n-1$

$$\therefore T^{worst}(n) = \max_{I} T(I)$$
$$= n + T^{worst}(n-1)$$
$$= n + (n-1) + T^{worst}(n-2)$$
$$= \cdots$$
$$= \frac{n(n+1)}{2} \in \Theta(n^2).$$

Average-case:

$$T^{avg}(n) = \frac{1}{|\mathcal{I}_n|} \sum_{I \in \mathcal{I}_n} T(I)$$
$$= \frac{1}{n} \sum_{k=0}^{n-1} \frac{1}{n!} \sum_{\pi} T(\pi, k).$$

We will do

① Creating a random quick select;

② Analyze exp. runtime; then

③ Argue that this implies bound on avg-case of quick-select.

RANDOMIZED QUICK-SELECT

1 Consider the "randomized quick-select" algorithm:

```
quick-select1(A, k)
 // A : array of size n
 // k: integer s.t. 0<k<n
    p ← random(n)    // key step
    i ← partition(A, p)
    if i=k then
        return A[i]
    else if i>k then
        return quick-select1(A[0,...,i-1], k]
    else if i<k then
        return quick-select1(A[i+1, i+2,...,n-1], k-i-1)
```

2 Then, what is $P(\text{pivot-index} = i)$?

$\Rightarrow$ pivot-value is equally likely to be any of $A[0], \dots, A[n-1]$

$\Rightarrow$ $\therefore$ pivot-index is equally likely to be any of $0, \dots, n-1$

$\Rightarrow$ $P(\text{pivot index} = i) = \frac{1}{n}$

3 We claim $T^{exp}(n) \in O(n)$.

Proof. Recall that

$$T^{exp}(n) = \max_{I} \sum_{\substack{\text{random} \\ \text{outcomes } R}} P(R) \cdot T(I,R)$$

In particular, note that

$$\underset{\substack{\uparrow \\ \langle A,k\rangle}}{T(I}, \underset{\substack{\nwarrow \\ \langle i,R'\rangle}}{R)} = \begin{cases} n + T(A[0,\dots,i-1], k, R') & i>k \\ n + T(\text{right subarray}, k-1-1, R'), & i<k \\ n & i=k \end{cases}$$

(we count # of comparisons)

Then,

$$\sum_R P(R)\,T(I,R) = \sum_R P(R)\,T(A,k,R)$$

$$= \sum_{(i,R')} \underbrace{P(i)}_{\downarrow\ \frac{1}{n}} P(R') \underbrace{T(A,V,R)}_{\downarrow\ n + \{\vdots}$$

$$= n + \frac{1}{n}\sum_{i=0}^{n-1} \begin{cases} \sum_{R'} P(R')\,T(A_\ell, k, R') & i>k \\ \sum_{R'} P(R')\,T(A_r, k-i-1, R') & i<k \\ 0 & i=k \end{cases}$$

$$\leq n + \frac{1}{n}\sum_{i=0}^{n-1} \begin{cases} \max_{k'} \max_{A_\ell} \sum_{R'} P(R')\,T(A_\ell, k', R') & i>k \\ \max_{k'} \max_{A_r} \sum_{R'} P(R')\,T(A_r, k', R') & i<k \\ 0 & i=k \end{cases}$$

$$= n + \frac{1}{n}\sum_{i=0}^{n-1} \begin{cases} T^{exp}(i) & i>k \\ T^{exp}(n-i-1) & i<k \\ 0 & i=k \end{cases}$$

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

Then, we show $T^{exp}(n) \leq 8n$.

Proof. By induction.

$n=1$: $\text{comp} = 0 < 8(1) = 8$.

step:

$$T(n) \leq n + \frac{1}{n}\sum_{i=0}^{n-1} \max\{8i, 8(n-i-1)\}$$

$$\leq n + \frac{1}{n}\sum_{i \text{ good}} 8\left(\tfrac{3}{4}n\right) + \frac{1}{n}\sum_{i \text{ bad}} 8n$$

$$= n + \frac{1}{n}\cdot\frac{n}{2}(6n) + \frac{1}{n}\cdot\frac{n}{2}(8n)$$

$$= n + 3n + 4n = 8n. \quad \#$$

$\therefore T^{exp}(n) \in O(n)$. ▨

QUICK-SORT

1 Pseudocode:

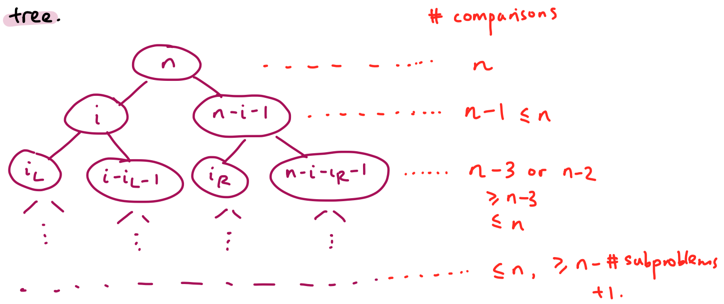
```
quick-sort1(A)
 // A: array of size n
    if $n \le 1$ then return
    $p \leftarrow$ choose-pivot1(A)
    $i \leftarrow$ partition(A, p)
    quick-sort1(A[0, ..., i-1])
    quick-sort1(A[i+1, ..., n-1])
```

2 There are better implementations of this idea.

3 Runtime:

$$\boxed{T(n) = n + T(i) + T(n-i-1).}$$

But there's a simpler method: the recursion tree.



Thus

$$\boxed{\#\text{ comparisons} \le n \cdot \#\text{ layers} = n \cdot \text{height of the recursion tree.}}$$

4 So what can we say about the recursion tree's height?

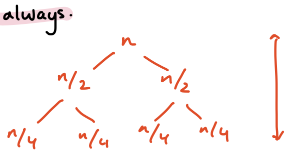
① Worst case: height $= \Theta(n)$

* tight if array is sorted.

– we need n recursions.

Thus run-time $\in \Theta(n^2)$.

② Best case: if the pivot-index is $\sim \frac{n}{2}$ always.



$\boxed{\therefore \text{height} = \log n.}$

Thus run-time $\in \Theta(n \log n)$.

5 We can improve QuickSort's run-time by

① Not passing sub-arrays, but instead passing "boundaries";

② Stopping recursion early;

- stop recursing when array of subproblem is ≤ 10
- then use insertion-sort to sort the array
- which has $\Theta(n)$ best-case run time if the array is (almost) sorted

③ Avoid recursions;

④ Reduce auxiliary space;

- in the worst case (currently), the auxiliary space is $|S| \in O(n)$.
- but if we put the bigger subproblem on the stack, the space can be reduced to $|S| \in O(\log n)$.

⑤ Choose the pivot-index efficiently;

- don't let $p = n-1$. (run-time $= O(n^2)$)
- use "median-of-3": use the median of $\{A[0], A[\lfloor \frac{n}{2} \rfloor], A[n-1]\}$ as the pivot-value

AVERAGE-CASE QUICK-SORT

1 We accomplish this via randomization of the algorithm:

```
RandQSort(A)
 // A: array of size n
    if $n \le 1$ then return
    $p \leftarrow$ random(n)
    $i \leftarrow$ partition(A, p)
    quick-sort1(A[0, ..., i-1])
    quick-sort1(A[i+1, ..., n-1])
```

2 Then,

$$T^{exp}(n) = \text{exp \# of comparisons of the algo.}$$

3 Note that

$$\boxed{\underset{\text{instance}}{T(A}, \underset{\text{outcomes}}{R)} = n + T(\underbrace{\text{left sub array}}_{\text{size } i}, R') + T(\underbrace{\text{right sub array}}_{\text{size } n-i-1}, R').}$$

4 Hence

$$\sum_R P(R)T(A,R) = n + \frac{1}{n}\sum_{i=0}^{n-1} T^{exp}(i) + \frac{1}{n}\sum_{i=0}^{n-1} T^{exp}(n-i-1)$$

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

and so

$$\boxed{T(n) = \begin{cases} 0, & n \le 1 \\ n + \frac{2}{n}\sum_{i=2}^{n-1} T(i), & \text{otherwise.} \end{cases}}$$

5 We claim $T(n) \in O(\log(n))$, so the expected run-time of RandQSort is in $O(n \log n)$, and so the average-case run time of QSort is in $O(n \log n)$.

Proof. Specifically, we prove $T(n) \le 2 \cdot n \cdot \ln(n)$.

By induction. Base $(n=1)$ is trivial.

Step:

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

$$= n + \frac{4}{n}\sum_{i=2}^{n-1} i\ln(i)$$

$$\le n + \frac{4}{n}\int_2^n x\ln x \, dx \quad \text{(since } x\ln x \text{ is increasing)}$$

$$\le n + \frac{4}{n}\left(\frac{1}{2}n^2\ln(n) - \frac{1}{4}n^2\right)$$

$$= n + 2n\ln(n) - n$$

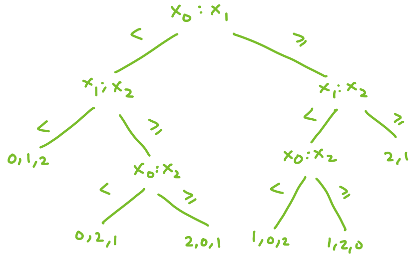
$$= 2n\ln(n). \quad \square$$

ANY COMPARISON BASED SORTING ALGORITHM USES $\Omega(n\log n)$ KEY-COMPARISONS IN THE WORST-CASE

As above.

Proof. Fix an arbitrary comparison-based sorting algorithm $\mathcal{A}$, and consider how it sorts an instance of size n.

Since $\mathcal{A}$ uses only key-comparisons, we can express it as a <u>decision tree</u> $\mathcal{T}$.



A decision tree for an algo to sort 3 elements x_0, x_1, x_2 with $\leqslant 3$ key-comparisons.

For each sorting perm π, let I_π be an instance that has distinct items and sorting perm π (ie $I_\pi = \pi^{-1}$). Executing $\mathcal{A}$ on I_π leads to a leaf that stores π.

Note that no two sorting perms can lead to the same leaf, as otherwise the output would be incorrect for one (as the items are distinct).

We then have $n!$ sorting perms, and hence at least $n!$ leaves that are reached for some I_π.

Let h be the largest layer-number of a leaf reached by some I_π.

Since $\mathcal{T}$ is binary, for any $0 \leq \ell \leq h$ there are at most 2^ℓ leaves in layers $0, \ldots, \ell$.

Therefore $2^h \geqslant n!$, or

$$
\begin{aligned}
h \geqslant \log(n!) &= \log(n(n-1)\cdots 1) \\
&= \log(n) + \log(n-1) + \cdots + \log(1) \\
&\geqslant \log(n) + \log(n-1) + \cdots + \log(\lceil \tfrac{n}{2} \rceil) \\
&\geqslant \log(\tfrac{n}{2}) + \log(\tfrac{n}{2}) + \cdots + \log(\tfrac{n}{2}) \\
&\geqslant \tfrac{n}{2}\log(\tfrac{n}{2}) \\
&= \tfrac{n}{2}\log(n) - \tfrac{n}{2} \in \Omega(n\log n).
\end{aligned}
$$

Finally, consider the sorting perm π that has its leaf on layer h.

Executing algorithm $\mathcal{A}$ on I_π hence takes $h \in \Omega(n\log n)$ key-comparisons, so the worst-case bound holds. ☒

SORTING INTEGERS

BUCKET-SORT

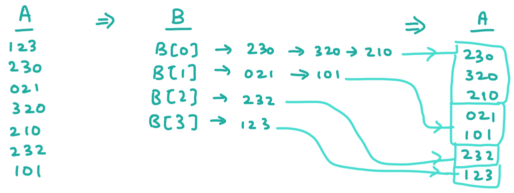
Bucket-sort can be used to sort a collection of integers by a specific "position" of digit.

eg the last digit   1234$\underline{5}$

Pseudocode:

```
bucket-sort (A, n ← A.size, ℓ ← 0, r ← n-1, d)
// A: array of size ≥ n with numbers with
      m digits in {0,…, R-1}
// d: 1 ≤ d ≤ m
// output: A[ℓ…r] is sorted by "d" digit.
initialize an array B[0…R-1] of empty
lists
for i ← ℓ to r do                                // move array-items
    append A[i] at the end of                    //   to buckets
    B [dth digit of A[i]]
i ← ℓ
for j ← 0 to R-1 do                              // move bucket-items
    while B[j] is not empty do                   //   to array
        move first element of B[j]
        to A[i]
        i++
```

eg



See that

① Run-time = $\Theta(n+R)$; and

② Auxiliary space = $\Theta(n+R)$.

MSD-RADIX-SORT

"MSD (Most Significant Digit) radix sort" can be used to sort multi-digit numbers.

Pseudocode:

```
MSD-radix-sort (A, n ← A.size, L ← 0, r ← n-1, d ← 1)
// A: array of size ≥ n, contains numbers with
      m digits in {0,…, R-1}, m, R are global
      variables
// ℓ, r: range (ie A[ℓ…r]) we wish to sort
// d: digit we wish to sort by
if ℓ < r then
    bucket-sort (A, n, ℓ, r, d)
    if d < m then
        // find sub-arrays that have the same
           dth digit and recurse
        int ℓ' ← ℓ
        while ℓ' < r do
            int r' ← ℓ'
            while r' < r & dth digit of A[r'+1] = dth digit of
            A[ℓ'] do r'++
            MSD-radix-sort (A, n, ℓ', r', d+1)
            ℓ' ← r' +1
```

Note that

① run-time = $\Theta(mRn)$; &

② auxiliary space = $\Theta(n+R+m)$.

LSD-RADIX-SORT

"LSD-Radix-Sort" is a better way of sorting multi-digit numbers (than MSD) because

① it has a faster runtime;

② it uses less auxiliary space; and

③ it uses no recursion.

Pseudocode:

```
LSD-radix-sort (A, n ← A.size)
// A: array of size n, contains m-digit
      radix-R numbers, m, R are global
for d ← m down to 1 do
    bucket-sort (A, d)
```

Clearly

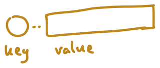
① run-time = $\Theta(m(n+R))$; &

② auxiliary space = $\Theta(n+R)$.

Chapter 4:
ADT Dictionaries

DICTIONARIES

Dictionaries store key-value pairs, or "KVP".



In particular,

① search(key): return the KVP for this key.

② insert(key, value): add the KVP to the dictionary.
  - key is distinct from existing keys

③ delete(key): remove KVP with this key.

Assumptions:

① we assume all keys are distinct.

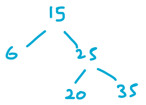
② we also assume keys can be compared.

Implementations:

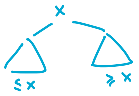
① Unsorted list
  - fast insert, slow search, slow delete

② Sorted array
  - fast search, slow insert, slow delete

③ Binary search tree / BST



- we only show the keys (implied each node is a KVP)
- has the property



In particular,

insert, delete, search ∈ O(height of tree).

* Height is Θ(n) is worst case, but typically much better (Θ(log n))

LAZY DELETION

Consider a sorted array:

delete(20):

- usually takes Θ(n) time to "backtrack" all other elements
- but we can avoid this if we instead just mark the box as "isDeleted".
- thus run-time (search) = O(log n).
- but we don't get any space back!
- however, we can occasionally "clean' up':
  - create a new initialization; &
  - move all items into the new array if they were "real".
- clean-up takes O(n * insert).

  But we can frequently do better; in this example, we can perform clean-up in O(n) time.

Then, the amortized time is O(insert + delete) for doing a deletion, and sometimes better.

Why do we do lazy deletion?

① It is simpler;

② It might be faster.
  - for sorted arrays: Θ(n) worst-case for delete but Θ(log n) amortized time for lazy deletion)

③ It sometimes is required.

Why not?

① It wastes space.
  - we have to allocate space to store the "isDeleted" flag.

② Occasionally deletion is very slow.

AVL-TREES

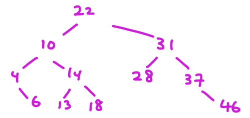
Goal: $\Theta(\log n)$ worst case time.

Structural condition:

For any node z, we have

$$\boxed{\text{balance}(z) = |\text{height}(z.\text{left}) - \text{height}(z.\text{right})| \leq 1.}$$

eg



* we define the height of an empty subtree is -1.

- notice the structural property holds for every node.

We also store the height of the subtree at every node, or we can also store the balance.

(usually we will store the height).

$h \in O(\log n)$

Let h be the height of an AVL tree with n nodes.

Then necessarily $h \in O(\log n)$.

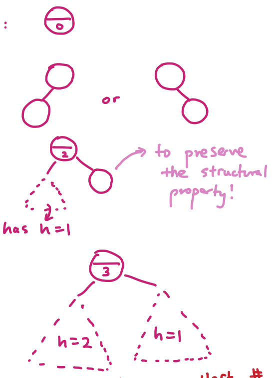
nodes

height 0 : 1

height 1 : 2

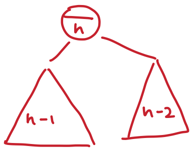
height 2 : 4

height 3 : 4+2+1 = 7



In general, let $N(h)$ = smallest # of nodes of height h.

h :



$\therefore \; N(h) = N(h-1) + N(h-2) + 1.$

Thus

h	0	1	2	3	4	5	…
$N(h)$	1	2	4	7	12	20	…
$N(h)+1$	2	3	5	8	13	21	

We see $N(h)+1$ is the Fibonacci numbers!

(ie $F(0)=0$, $F(1)=1$, $F(i) = F(i-1) + F(i-2)$ $\forall i \geq 2$.)

By a easy induction proof, we can show

$$N(h) + 1 = F(h+3).$$

In particular, we know

$$F(i) = \frac{1}{\sqrt{5}} \phi^{i} + \Theta(1), \quad \phi = \text{the golden ratio}.$$

Therefore

$$N(h) = \frac{1}{\sqrt{5}} \phi^{h+3} + \Theta(1)$$

Hence, for any AVL tree of height h,

$$\#\text{ nodes} = n \geq N(h) \approx \frac{1}{\sqrt{5}} \phi^{h+3},$$

and so

$$h \approx \log_{\phi}(\sqrt{5}\, n) - 3 \in O(\log n).$$

AVL OPERATIONS

INSERT, PART 1

First, we call BST::insert, and then rebalance the ancestors of z.

```
AVL::insert(k,v)
    $z \leftarrow$ BST::insert(k,v)
    while z is not NIL do
        if (|z.left.height - z.right.height| > 1) then
            Let y $\leftarrow$ taller child of z
            Let x $\leftarrow$ taller child of y
            $z \xleftarrow{p}$ restructure(x,y,z)
            break
        z.height $\leftarrow$ 1 + max{z.left.height,          // we alias this as
                                z.right.height}          "setHeightFromSubtrees"
        $z \leftarrow$ z.parent
```

ROTATIONS [OF BSTS]

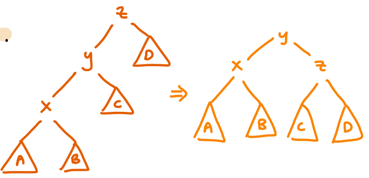
Let T be a BST with a node z that has a child y and a grandchild x. Then, a "rotation at z with respect to y & x" is a restructuring of T such that the result is again a BST, and sub-tree references have been changed only at x, y, z.

For restoring balances at AVL-trees, we want the four rotations that make the median of x, y, z the new root of the subtree:

① Right rotation; $x < y < z$.

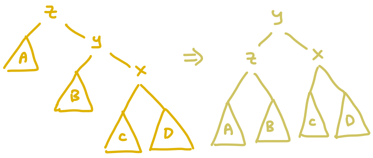
We want y to become the root.

pseudocode similar to left rotation; see below



② Left rotation; $z < y < x$.

We want y to be the root.

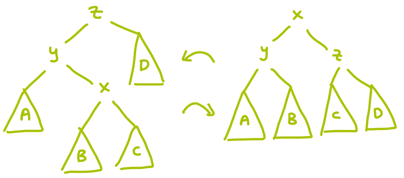


```
rotate-left(z)
    $y \leftarrow$ z.right
    z.right $\xleftarrow{p}$ y.left
    y.left $\xleftarrow{p}$ z
    setHeightFromSubtrees(z)
    setHeightFromSubtrees(y)
    return y
```

③ Double-right rotation: $y < x < z$.

We want x to be the root.

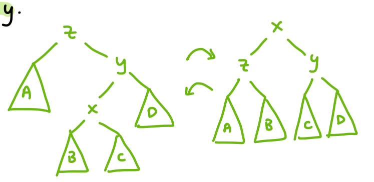
* can also be written as single-left rotation at y and then a single-right rotation at z (double-rotation)



④ Double-left rotation: $z < x < y$.

We want x to be the root.

* can also be written as single-right rotation at y and then a single-left rotation at z (double-rotation)



INSERT, PART 2

We now give the implementation of restructure in the insert function from the first part;

```
restructure(x,y,z)
    // node x has parent y & grandparent z
    if  y = z.left & x = y.left                //   z
                                               //  y
                                               // x
        return rotate-right(z)                 // right rotation
    else if  y = z.left & x = y.right          //   z
                                               //  y
                                               //   x
        z.left $\xleftarrow{p}$ rotate-left(y)
        return rotate-right(z)                 // double-right rotation
    else if  y = z.right & x = y.left          // z
                                               //  y
                                               // x
        z.right $\xleftarrow{p}$ rotate-right(y)     // double-left rotation
        return rotate-left(z)
    else                                       // z
                                               //  y
                                               //   x
        return rotate-left(z)                  // left rotation
```

DELETE

Pseudocode:

```
AVL::delete(k)
    $z \leftarrow$ BST::delete(k)        // z is the parent of the BST node that was removed
    while z is not NIL do
        if (|z.left.height - z.right.height| > 1) then
            let y = taller child of z
            let x = taller child of y
            // (break ties to prefer single rotation)
            $z \xleftarrow{p}$ restructure(x,y,z)
            // do not break — continue up the path & rotate if needed.
        setHeightFromSubtrees(z)
        $z \leftarrow$ z.parent
```

RUNTIME

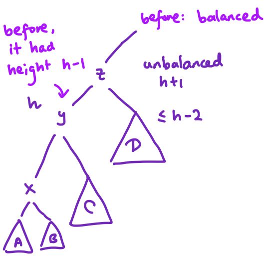
Both insert & delete have run-time $\Theta(\log n)$.

Why?
- height of tree = $O(\log n)$
- rotations take $\Theta(1)$ time
- so delete takes $O(\log n)$.
- tight if we insert/delete at lowest level & never rotate.

CORRECTNESS [FOR INSERTION]

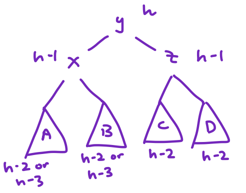
If we restructure at z during an insertion, then
① the subtree is balanced; and
② the subtree has now the height that it had before the insertion.

Proof.

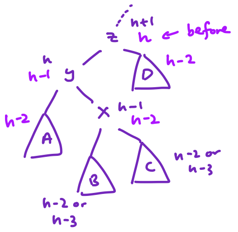


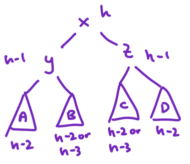
Case 1: "x" is the left node
- x has height h-2 before
- C has height h-2

After rotation:



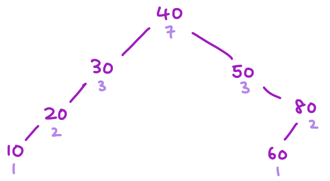
Case 2: x is the right child.





SCAPEGOAT TREES

"Scapegoat trees" are BSTs such that
① Each node v stores the size of its subtree;
and
② v.size ≤ $\alpha \cdot$ v.parent.size for all nodes that are not the root.



* example, with $\alpha = \frac{2}{3}$.
Verify that
$p.size \geq \frac{3}{2} v.size$
for any parent p of any node v.

HEIGHT = O(log n)

Any scapegoat tree has height $O(\log n)$.

Proof. At any leaf ℓ (which has size 1), the parent has size $\geq \frac{1}{\alpha}$ by def$^{\underline{n}}$ of a scapegoat tree.
Repeating this argument, the grand-parent has size $\geq (\frac{1}{\alpha})^2$, and so on.
As the root has size n, it follows that

$$\left(\frac{1}{\alpha}\right)^d \leq n,$$

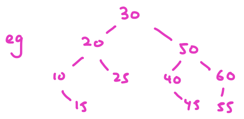
where d = max depth of a leaf = height.
Thus

$$d \leq \log_{(\frac{1}{\alpha})}(n) \in O(\log n). \quad \boxtimes$$

PERFECTLY BALANCED BST

1. A "perfectly balanced BST" is one where for any node v, we have

$$|v.left.size - v.right.size| \leq 1.$$



2. Given any n-node BST T, we can build a perfectly balanced BST with the same KVPs in $\Theta(n)$ time.

INSERT

1. Pseudocode:

```
ScapegoatTree::insert(k,v)
    $z \leftarrow$ BST::insert(k,v)
    $S \leftarrow$ stack initialized w/ z
    while $p \leftarrow$ z.parent ≠ NIL do
        p.size++
        S.push(p)
        $z \leftarrow p$
    while S.size ≥ 2 do
        $p \leftarrow$ S.pop()
        if p.size < $\frac{1}{\alpha}$ max{p.left.size, p.right.size} then
            completely rebuild the subtree rooted
            at p as a perfectly balanced BST
            break
```

2. insert has worst case runtime $\Theta(n)$, but amortized runtime $O(\log n)$.

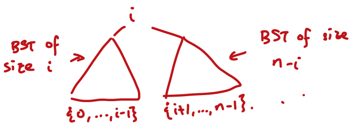
RANDOMIZED BUILT BST HAS EXPECTED HEIGHT $\Theta(\log n)$

1 A randomized built BST has expected height $\Theta(\log n)$.

2 Here, "randomly built" means we take a permutation (randomly) and insert items in the order of said permutation.

Proof. First, the first item of the perm π becomes the root.

Then, once we know the root i:



Moreover,

$$P(\text{first item of } \pi \text{ is } i) = \frac{1}{n}.$$

Thus exp. height of the tree w/ perm π is

$$H(\pi) = 1 + \max\{H(\pi_L), H(\pi_R)\}.$$

Now, define $Y(\pi) = 2^{H(\pi)}$, & $Y(n) = E[Y(\pi)]$.

Then

$$E[H(\pi)] = E[\log(Y(\pi))]$$
$$\leq \log(E(Y(\pi))). \quad (\because \log \text{ is concave})$$

We will show $E[Y(\pi)] \leq (n+1)^3$, which implies

$$H(n) \leq \log((n+1)^3) = 3\log(n+1).$$

So, let's do so. We note

$$Y(\pi) = 2^{1+\max\{H(\pi_L), H(\pi_R)\}}$$
$$= 2 \cdot \max\{2^{H(\pi_L)}, 2^{H(\pi_R)}\}$$
$$\leq 2(2^{H(\pi_L)} + 2^{H(\pi_R)})$$
$$= 2(Y(\pi_L) + Y(\pi_R)),$$

and so

$$Y(n) = \frac{1}{n}\sum_{i=0}^{n-1}(2Y(i) + 2Y(n-i-1))$$
$$= \frac{4}{n}\sum_{i=0}^{n-1} Y(i).$$

Now, we show $Y_n \leq (n+1)^3$.

Base: $n=1 \Rightarrow \text{height}=0 \Rightarrow Y(1)=2^0=1 \leq 8.$

$n=0 \Rightarrow \text{height}=-1 \Rightarrow Y(0)=2^{-1}=1/2 \leq 1.$

Step: $Y(n) = \frac{4}{n}\sum_{i=0}^{n-1}(i+1)^3$

$$= \frac{4}{n}\sum_{i=1}^{n} i^3 = \frac{4}{n} \cdot \frac{n^2(n+1)^2}{4}$$
$$= n(n+1)^2 \leq (n+1)^3,$$

as needed. ☒

EXPECTED VS AVE

1 We know

$E(\text{height of BST}) \in \Theta(\log n)$.

But

avg of height of BST is not $\Theta(\log n)$!

2 We can show the average height of a BST is in $\Theta(\sqrt{n})$.

3 Intuition on why:

P(randomly built BST has shape ) = $\frac{1}{n}$

But

\# BSTs w/ shape  / total # of BSTS = # BST on n-1 items / # BST on n items

$$= \frac{C(n-1)}{C(n)} \approx \frac{1}{4}$$

where $C(k)$ are the Catalan numbers.

TREAPS

1 A "treap" is a randomized version of a BST, also called a "priority search tree".

2 A treap has the following properties:

① Each node has a KVP and a priority p;

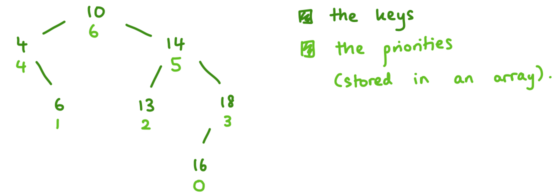
② The treap acts like a BST wrt to the KVPs; &

* everyone in left ≤ root ≤ everyone in right

③ The treap acts like a heap wrt to the priority.

* each node stores the item with max priority among all nodes in its subtree

eg



TREAP INSERTION

When inserting into the treap, we just

① BST insert; and

② do "fix-up" to restore the heap-order property for the priorities.

```
treap :: fix-up-with-rotations(z)
// z : node whose priority may have increased
    while (y ← z.parent is not NIL & z.priority > y.priority) do
        if z is the left child of y then rotate-right(y)
        else rotate-left(y)

treap :: insert(k, v)
    n ← P.size
    z ← BST :: insert(k,v) ;  n++      // z is the leaf where k is now stored

    p ← random(n)
    if p < n-1 then        // change priority of other node
        z' ← P[p],  z'.priority ← n-1,  P[n-1] ← z'
        fix-up-with-rotations(z')
    z.priority ← p,  P[p] ← z
    fix-up-with-rotations(z)
```

RUN-TIME / SPACE

Note that treaps are

① BSTs with expected height $O(\log n)$, and so the expected runtime of all operations is $O(\log n)$;

② have a large space over-head, since we store the BST, and parent-references & priorities for each node.

SKIP LISTS

A "skip list" is a hierarchy S of ordered linked lists (levels) $S_0, \ldots, S_h$, such that

① Each list S_i contains the special keys $-\infty$ & $+\infty$, called "sentinels";

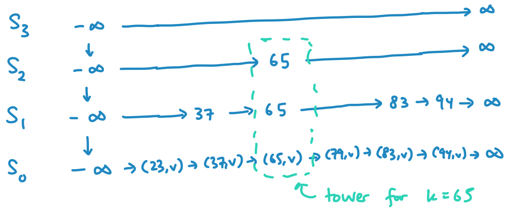
② S_0 contains the KVP of S in non-decreasing order, and the other lists store only keys;

③ Each list is a subsequence of the previous one, ie

$S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h$; &

④ S_h contains only the sentinels.

eg



Notes:

① A "node" in a skip list is any node in the linked lists of the hierarchy;

② Each node has two references:
- 'after' — points to the next node in the LL
- 'below' — points to the copy of the node in S_{i-1}

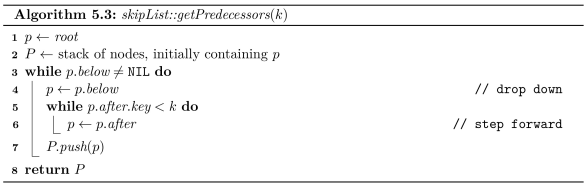
③ The "tower" of a key k is the set of all nodes that contain k;

④ The "height" of a tower is the maximum index i such that the tower includes a node in S_i;

⑤ The "height" of the skip list is the maximum height of a tower, which is equal to h.

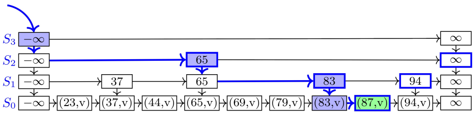
getPredecessors(k)

'getPredecessors(k)' is a helper routine for insert & delete.



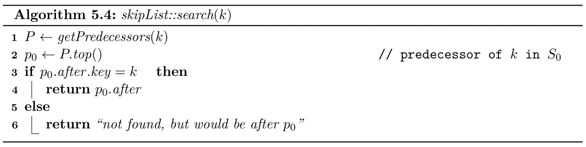
```
Algorithm 5.3: skipList::getPredecessors(k)
p ← root
P ← stack of nodes, initially containing p
while p.below ≠ NIL do
    p ← p.below                                   // drop down
    while p.after.key < k do
        p ← p.after                               // step forward
    P.push(p)
return P
```

Example:



SEARCH

'search' is very simple; it just uses getPredecessors.



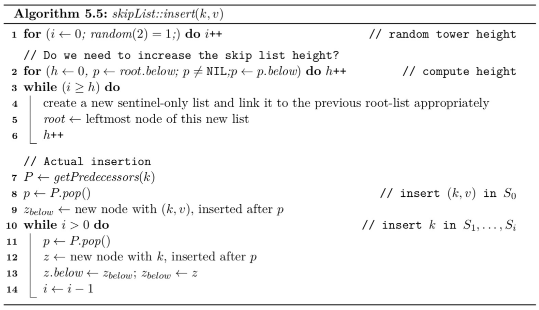
```
Algorithm 5.4: skipList::search(k)
P ← getPredecessors(k)
p_0 ← P.top()                                     // predecessor of k in S_0
if p_0.after.key = k    then
    return p_0.after
else
    return "not found, but would be after p_0"
```

INSERT

For 'insert', we first call 'getPredecessors', which tells us which node would precede the inserted KVP at each S_i.

But, we determine whether k should be in S_i randomly; in particular,

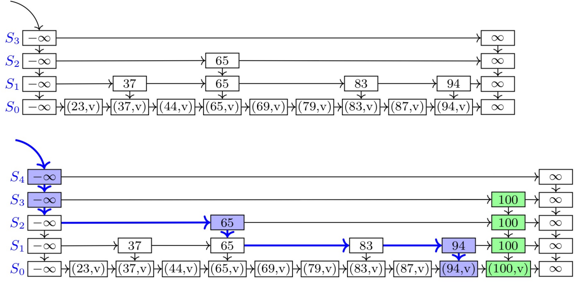
$$P(\text{tower of key } k \text{ has height} \geq i) = \frac{1}{2^i}.$$



```
Algorithm 5.5: skipList::insert(k, v)
for (i ← 0; random(2) = 1;) do i++                // random tower height
// Do we need to increase the skip list height?
for (h ← 0, p ← root.below; p ≠ NIL;p ← p.below) do h++      // compute height
while (i ≥ h) do
    create a new sentinel-only list and link it to the previous root-list appropriately
    root ← leftmost node of this new list
    h++
// Actual insertion
P ← getPredecessors(k)
p ← P.pop()                                       // insert (k, v) in S_0
z_below ← new node with (k, v), inserted after p
while i > 0 do                                    // insert k in S_1, ..., S_i
    p ← P.pop()
    z ← new node with k, inserted after p
    z.below ← z_below; z_below ← z
    i ← i − 1
```

Example:

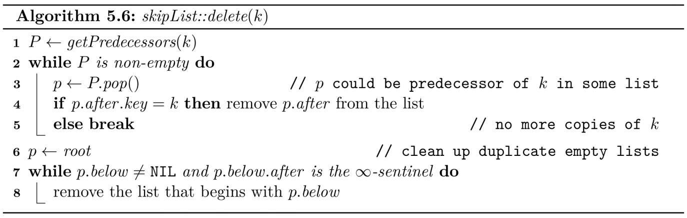
Insertion of key = 100, with the determined tower height = 3.



DELETE

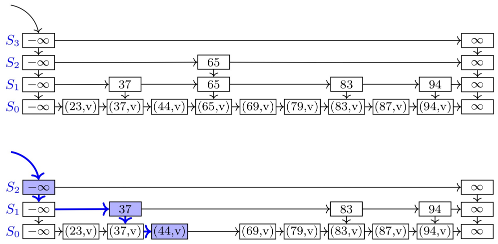
To delete in a skip list, we find the key (which gives the stack of predecessors) and then remove it from all lists that it was in.

We also "clean-up" the stack: if deleting a key results in multiple lists that store only sentinels, then delete all but one of them.



```
Algorithm 5.6: skipList::delete(k)
P ← getPredecessors(k)
while P is non-empty do
    p ← P.pop()                    // p could be predecessor of k in some list
    if p.after.key = k then remove p.after from the list
    else break                                    // no more copies of k
p ← root                                          // clean up duplicate empty lists
while p.below ≠ NIL and p.below.after is the ∞-sentinel do
    remove the list that begins with p.below
```

Example:



* deleting the KVP w/ key=65.

$E(\text{len}(S_i)) = \frac{n}{2^i}$

In a skip list, the length of a list S_i is $\frac{n}{2^i}$.

Proof. Let X_k be the rv that denotes the height of the tower w/ key k.

Let $I_{i,k}$ be an indicator variable that is 1 if $X_k \geq i$ (ie list S_i contains key k) & 0 otherwise.

Then
$$|S_i| = \sum_{\text{key } k} I_{i,k}$$
and so
$$\begin{aligned} E[|S_i|] &= \sum_{\text{key } k} E[I_{i,k}] \\ &= \sum_{\text{key } k} (P(X_k \geq i)) \\ &= \sum_{\text{key } k} \frac{1}{2^i} \\ &= \frac{n}{2^i}. \end{aligned}$$ ∎

$E(\text{height of SL}) \leq \log n + O(1)$

The expected height of a skip list is at most $\log(n) + O(1)$.

Proof. Let I_i be an ind var s.t. $I_i = 1$ if $|S_i| \geq 1$ & 0 otherwise.

Recall height (SL) = h, where the lists are $S_0, \ldots, S_h$.

Since $S_0, \ldots, S_{h-1}$ all contain keys, thus
$$h = \sum_{i \geq 0} I_i .$$

Then, note that by defn $I_i \leq 1$ & $I_i \leq |S_i|$, so
$$E[I_i] \leq \min\{1, \frac{n}{2^i}\}.$$

If $i \approx \log n$ then $1 \approx \frac{n}{2^i}$, so we use this to "break up" the sum.

In particular, notice
$$\begin{aligned} E[h] = \sum_{i \geq 0} E[I_i] &\leq \sum_{i=0}^{\lceil \log n \rceil - 1} E[I_i] + \sum_{i \geq \lceil \log n \rceil} E[|S_i|] \\ &\leq \sum_{i=0}^{\lceil \log n \rceil - 1} (1) + \sum_{i \geq \lceil \log n \rceil} \frac{n}{2^i} \\ &\leq \lceil \log n \rceil + \sum_{j \geq 0} \frac{2^{\lceil \log n \rceil}}{2^{j + \lceil \log n \rceil}} \\ &\leq \log n + 1 + \sum_{j \geq 0} \frac{1}{2j} \\ &= 3 + \log n, \end{aligned}$$

as needed. ∎

EXPECTED SPACE = $\Theta(n)$

The expected space of a skip list is in $\Theta(n)$.

In particular, the expected number of nodes is $2n + o(n)$.

Proof. Each list S_i has $|S_i|$ nodes that store keys.

Hence expected # of nodes with keys is
$$E[\sum_{i \geq 0} |S_i|] = \sum_{i \geq 0} \frac{n}{2^i} = n \sum_{i \geq 0} \frac{1}{2^i} = 2n .$$

There are $2h+2$ nodes that do not store keys (the sentinels on each list $S_0, \ldots, S_h$), but we have $E[h] \leq \log(n) + O(1) \in o(n)$, & so the bound holds. ∎

getPredecessors: $E(\text{forward steps of } S_i) \leq 1$

During 'getPredecessors', the expected number of forward-steps within list S_i is at most 1.

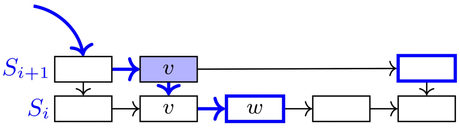
Proof. Let v be the leftmost node in list S_i that we visited during the search.

If $i = h$ (the topmost list) then we do not step forward at all and are done, so assume $i < h$ & we reached v by dropping down from list S_{i+1}.

Let w be the item after v in S_i.

Consider the prob. we step forward from v to w: if w also exists in list S_{i+1}, then we compared (before dropping down to v) the search key k with w.key.

So, we must have had $k \leq w.\text{key}$, else we would not have dropped down from v. Thus in list S_i we will immediately drop down.



Taking the contrapositive, if we step forward from v in list S_i, then the next node w in S_i did not exist in S_{i+1}.

In other words, the tower of w had height exactly i.

The probability of this is $\frac{1}{2}$, because the decision to expand the tower of w into the list above was based on a "coin flip".

So we step forward from v w/ prob $< \frac{1}{2}$.

Repeating this argument, we step forward from w with prob $< \frac{1}{2}$, presuming we arrived at w in the first place, so the probability of this happening is at most 1/4. Repeating, we see the prob of stepping forward i times is $< \frac{1}{2^i}$.

Thus
$$\begin{aligned} E[\text{\# of forward steps}] &= \sum_{\ell \geq 1} P(\text{\# of forward-steps is} \geq \ell) \\ &= \sum_{\ell \geq 1} \frac{1}{2^\ell} \leq 1. \end{aligned}$$ ∎

EXPECTED RUN-TIME OF SEARCH / INSERT / DELETE IS $O(\log n)$

As above.

Proof. First, exp. run-time for getPredecessors is
$$O(E[h + \sum_{i \geq 0} F_i]),$$
where F_i = # of forward steps on level S_i $\in O(\log n)$.

By the previous results, the exp. time for getPredecessors is in $O(\log n)$.

Once the predecessors are found, all other operations take $O(h)$ time.

This has exp $O(\log n)$. ∎

BIASED SEARCH REQUESTS

STATIC SCENARIO

1. In the "static scenario", we know beforehand how frequently a key is going to be accessed.

key	A	B	C	D	E
access - freq	2	8	1	10	5
access - prob	$2/26$	$8/26$	$1/26$	$10/26$	$5/26$

2. Terms:
   ① "Access frequency" — amount of times key is accessed
   ② "Access probability" — proportion of accesses for a key

3. In particular, we want to find the optimum assignment of keys to locations; ie the assignment that minimizes

$$\boxed{\text{exp. access cost} = \sum_{\text{key } k} P(\text{want to access } k)\cdot(\text{cost of accessing } k)}$$

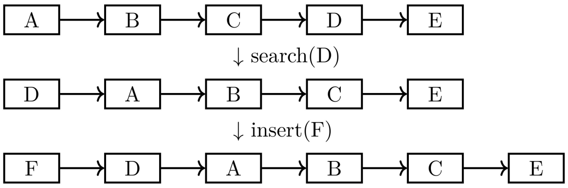
DYNAMIC SCENARIO

1. Here, we do not know how frequently a key is going to be accessed, so we cannot hope to build the best-possible data structure.

2. However, we can still change the data structure when we have accesses, to bring those items that were recently accessed to a place where the next access will be cheap.
   - if we access an item, it is fairly likely we will access it again soon
   - "temporal locality"

MOVE-TO-FRONT / MTF HEURISTIC IN A LIST

1. The MTF heuristic involves moving the most recently accessed item in an unsorted list to the front.

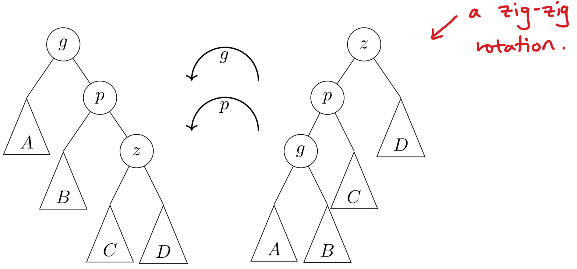


2. The heuristic is "2-competitive": it takes at most twice as many comparisons that would have been taken using the optimal static ordering.

SPLAY TREES

1. Splay trees are BSTs where after every operation, we apply zig-zag or zig-zig rotations (and perhaps one single rotation at the root) so the accessed item is at the root.

2. Here,
   ① "zig-zag rotations" are just double rotations; we do these if the "z-p-g" path contains a left & right child; &
   ② "zig-zig rotations" are applied if the "z-p-g" path contains two left or two right children.



a zig-zig rotation.

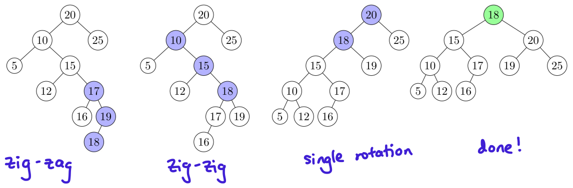
INSERT

1. Pseudocode:

Algorithm 5.8: *splayTree::insert(k, v)*

```
z ← BST::insert(k, v)
while (z has a parent p and a grand-parent g) do
    if  then                                         // Zig-zig rotation rightward
        p ← rotate-right(g), z ← rotate-right(p)
    else if  then                                    // Zig-zag rotation rightward
        g.left ← rotate-left(p), z ← rotate-right(g)
    else if  then                                    // Zig-zag rotation leftward
        g.right ← rotate-right(p), z ← rotate-left(g)
    else //                                          // Zig-zig-rotation leftward
        p ← rotate-left(g), z ← rotate-left(p)
if (z has a parent p) then                           // single rotation
    if (z = p.left) then  z ← rotate-right(p)
    else  z ← rotate-left(p)
```

2. Example:



RUN-TIME ANALYSIS

The amortized run-time of insert is $O(\log n)$, where n is the size of the tree.

Proof. For a splay tree S, let the pot. func.

$$\phi(i) = \sum_{v \in S} \log(n_v^{(i)})$$

where n_v = size of the subtree rooted at v.

Clearly this is a potential func ($\phi(0) = 0$ & $\phi(i) \geq 0 \because n_v^{(i)} \geq 1 \ \forall i, v$).

Insert has three phases:

① BST::insert;

② Bringing up the node with zig-zig & zig-zag rotation; &

③ Doing the (last) single rotation.

Lemma #1: ① increases ϕ by at most $\log n$.

Proof. Let the nodes in the path from z to the root be $z, z_1, \ldots, z_d$, where z_d = root.
After adding z, the size of the subtrees at all $z_1, \ldots, z_d$ increase by 1, whilst it is unchanged at all other nodes.
So, the contribution to ϕ only changes at $z_1, \ldots, z_d$, and in particular

$$n_{z_k}^{after} = n_{z_k}^{before} + 1 \leq n_{z_{k+1}}^{before} \quad \forall 1 \leq k \leq d.$$

Thus

$$\begin{aligned}
\Delta\phi &= \sum_v \log(n_v^{after}) - \log(n_v^{before}) \\
&= \underbrace{\log(n_z^{after})}_{1} + \sum_{k=1}^{d} (\log(n_{z_k}^{after}) - \log(n_{z_k}^{before})) \\
&\leq 0 + \sum_{k=1}^{d-1} (\log n_{z_{k+1}}^{before} - \log n_{z_k}^{before}) \\
&\quad + \log n_{z_d}^{after} - \log n_{z_d}^{before} \\
&= \log n_{z_d}^{before} - \underbrace{\log n_{z_1}^{before}}_{\geq 1} + \log \underbrace{n_{z_d}^{after}}_{n} - \log n_{z_d}^{before} \\
&\leq \log n
\end{aligned}$$

as needed. ☒

Lemma #2: Let O_i be a zig-zag / zig-zig rotation that moves z two levels up.
Then $\Phi^{after}(O_j) - \Phi^{before}(O_j) \leq 3\log(n_z^{after}) - 3\log(n_z^{before}) - 2$.

Proof. See TB.

Main proof: We finally show the amortized run-time of insert is $O(\log n)$.
Note $T^{actual}(insert) = 1 + d$, where d = depth from root to z.
Let the seq of operations in insert be

$$\underbrace{O_i}_{\text{BST::inset}}, \underbrace{O_{i+1}, \ldots, O_{i+R}}_{\text{zz / single}}.$$

Then

$$\begin{aligned}
\Delta\phi(insert) &= \phi(i+R) - \phi(i-1) \\
&= \sum_{j=i}^{i+R} (\phi(j) - \phi(j-1)) \\
&= \sum_{j=i}^{i+R} \Delta\phi(O_j) \\
&\leq \underbrace{\log n}_{\Delta\phi(O_i)} + \sum_{j=i+1}^{i+R-1} (\underbrace{3\log(n_z^{(j)}) - 3\log(n_z^{(j-1)}) - 2}_{\Delta\phi \text{ for zig-zig or zig-zag}}) + (\underbrace{3\log(n_z^{(i+R)}) - 3\log(n_z^{(i+R-1)})}_{\Delta\phi(O_{i+R})}) \\
&\leq \log n + 3\log(n_z^{(i+R)}) - 3\log(n_z^{(i)}) - 2(R-1) \\
&\leq \log n + 3\log n - 2R + 2,
\end{aligned}$$

and so

$$\begin{aligned}
T^{actual}(insert) + \Delta\phi(insert) &\leq (1+d) + 4\log n - 2R + 2 \qquad (\text{as } R \approx \lceil \tfrac{d}{2} \rceil) \\
&\leq 4\log n + 4 \\
&\in O(\log n)
\end{aligned}$$

as needed. ☒

BINARY SEARCH REVISITED

1. We cannot do better than binary search.
   - asymptotically, comparison-based

2. But, we can do a bit better;
   - shave constant

3. But, we can do a lot better; &

4. We can do even better!

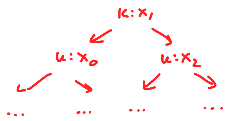
] drop comparison based

ANY COMPARISON-BASED SEARCH TAKES $\Omega(\log n)$ TIME

1. We want to prove a lower bound for searching.

2. Any search that is comparison-based among n elements takes $\Omega(\log n)$ worst-case time, even if the elements are sorted.

Proof. Fix an algorithm, and look at its decision tree.



We are given items x_0, x_1, x_2; & we search for k.

We want to show we have a lot of (accessible) leaves.
In particular, we have at least n leaves (one for each x_i).
Thus, the height is $\geq \log(n) \in \Omega(\log n)$. ☒

We can also note

$$\text{height} \geq \lceil \log(n) \rceil.$$

However, we also have $n+1$ "not found" leaves, corresponding to

$$k \in (-\infty, x_0), \quad k \in (x_0, x_1), \ \ldots, \quad k \in (x_n, +\infty).$$

3. In fact, we can show the height $\geq \lceil \log(2n+1) \rceil$.

Proof. In particular, we now show

$$\# \text{ of leaves} \geq 2n+1$$

(and so height $\geq \lceil \log(2n+1) \rceil$).
To do so, we create $2n+1$ instances

$$x_0 < x_1 < x_2 < \cdots \leq x_{n-1}.$$

Searching for x_i could result in the possibilities:

$$x_0, x_1, x_2, \ldots, x_{n-1}$$

Searching between x_i:

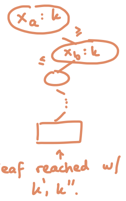
$$(x_0, x_1), (x_1, x_2), \ldots, (x_{n-2}, x_{n-1})$$

Searching outside:

$$(-\infty, x_0), (x_{n-1}, +\infty).$$

Claim: no two reach the same leaf.

Proof by contradiction.



Assume $\exists k', k''$, $k' \neq k''$, s.t. we reach the same leaf.
Then $k \neq x_i$ (as otherwise $k' \neq k''$), so it follows the leaf must be a 'not found'.

Consider the above example.
In particular, we have $x_b \leq k', k'' < x_a$, and so on.
Since k', k'' are 'not founds', there must exist an x_i between a k' & k''.
Thus

$$k' < x_i < k''$$

for some x_i.
Then, a search for x_i would reach the exact same leaf, which is a contradiction since the leaf is a 'not found'.

Chapter 6: Special Key Dictionaries

OPTIMIZED BIN SEARCH

1 Normal binary search takes

$$\boxed{T(n) = 2 + T(n/2) \approx 2\log(n).}$$

But can we do better?

2 Pseudocode:

```
binary-search-optimized(A, n, k)
    $\ell \leftarrow 0$, $r \leftarrow n-1$, $x \leftarrow 0$
        // $x$ is a bool var that tells us whether
           we're in the left subarray
    while ($\ell < r$)
        $m \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$
        if ($A[m] < k$) then $\ell \leftarrow m+1$
        else $r \leftarrow m$, $x \leftarrow 1$
    if ($k < A[\ell]$) then
        return "not found, bw $A[\ell-1]$ and $A[\ell]$"
    else if ($x = 1$) or ($k \leq A[\ell]$) then
        return "found at $A[\ell]$"
    else return "not found, bw $A[\ell]$ & $A[\ell+1]$"
```

3 We claim

① this algorithm terminates;

② this gives the correct answer; &

③ this uses at most $\lceil \log n \rceil + 2$ comparisons (without x), which is about $\lceil \log(2n+1) \rceil + 1$ comparisons.

④ With the x, this uses $\leq \lceil \log(2n+1) \rceil$ comparisons, so this is optimal.

Proof. See TB.

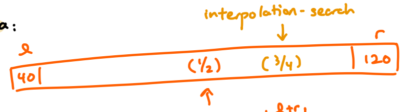
INTERPOLATION SEARCH

1 "Interpolation search" assumes that

① we are given an array $A[0 \ldots n-1]$ of numbers in sorted order; &

② we want to search for the number k.

2 Idea:



bin-search: $m = \lfloor \frac{\ell+r}{2} \rfloor$
$= \ell + \lfloor \frac{1}{2}(r-\ell) \rfloor$

Consider search(100).

⇒ bin-search searches naively in the middle.

⇒ interpolation search searches based on the "endpoints" of the keys, and interpolates where the key would be.

In particular, we notice that

$A[r] - A[\ell] = 80$ (the "distance" between nums).

Then

$k - A[\ell] = 60$.

So, we should search for 100 roughly $\frac{60}{80} = \frac{3}{4}$ in the range.

More generally, we search at the index,

$$\underbrace{\ell}_{\text{start index}} + \underbrace{\frac{A[r]}{A[\ell] - A[0]}}_{\text{ratio}} \underbrace{(r-\ell)}_{\text{indices in range}}$$

and otherwise, the search works like bin-search.

3 Pseudocode:

```
interpolation-search(A, n, k)
    $\ell \leftarrow 0$, $r \leftarrow n-1$
    while ($\ell \leq r$)
        if ($k < A[\ell]$ or $k > A[r]$) return "not found"
        if ($k = A[r]$) then return "found at $A[r]$"
        $m \leftarrow \ell + \lfloor \frac{k - A[\ell]}{A[r] - A[\ell]} \cdot (r-\ell) \rfloor$
        if ($A[m] == k$) then return "found at $A[m]$"
        else if ($A[m] < k$) then $\ell \leftarrow m+1$
        else $r \leftarrow m-1$
    // we always return from somewhere within the
       while loop
```

T^{avg} OF INTERPOLATION SEARCH IS $O(\log \log n)$

We can show under some assumptions,

$$\boxed{T^{avg}(n) \in O(\log \log n).}$$

→ see next page for proof under a realization.

T^{worst} OF INTERPOLATION SEARCH IS $\Theta(n)$

1 We can show that T^{worst} of interpolation-search is bad!

eg Consider

"0 1 2 ... 8 9 11".

and let's search for 10.

Then $A[r] - A[\ell]$ is huge! ⇒ so $m \approx \ell + 0 = 0$.

Then our next lower bound is 1, and so on, and we look at every element!

2 In particular,

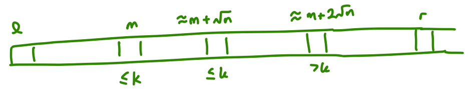
$$\boxed{T^{worst}(n) \in \Theta(n).}$$

OPTIMIZED INTERPOLATION-SEARCH

1 **Pseudocode:**

```
interpolation-search-modified(A, n, k)
if ($k < A[0]$ or $k > A[n-1]$) return "not found"
if ($k = A[n-1]$) return "found at index n-1"
$\ell \leftarrow 0$, $r \leftarrow n-1$                // $A[\ell] \leq k < A[r]$
while ($N \leftarrow (r-\ell-1) > 1$)
    $m \leftarrow \ell + \left\lceil \frac{k - A[\ell]}{A[r]-A[\ell]} \cdot (r-\ell-1) \right\rceil$
    if ($A[m] \leq k$)
        for $h = 1, 2, \ldots$
            $\ell \leftarrow m + (h-1)\lceil\sqrt{N}\rceil$, $r' \leftarrow \min\{r, m + h\lceil\sqrt{N}\rceil\}$
            if ($r' = r$ or $A[r'] > k$) then $r \leftarrow r'$ and break
    else ...
if ($k = A[\ell]$) return "found at index $\ell$"
else return "not found"
```

eg



① we compare here... (a 'probe')

② but we also probe here, if $A[m] < k$.

③ and we keep going, hopping by $\approx\sqrt{n}$ each time until
- $A[m + x\sqrt{n}] > k$; or
- $m + x\sqrt{n}$ is out of bounds.

So: the idea is we use more probes to guarantee the subarray has size $O(\sqrt{n})$.

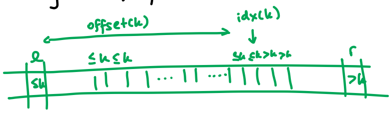
2 If $T(n) = \#$ of comparisons on n numbers, then

$$\boxed{T(n) = T(n') + \# \text{ of probes}}$$

where $n' \leq \sqrt{n}$.

3 Moreover, avg # of probes ≤ 2.5.

Proof.



Assume nums in $A[\ell+1 \ldots r-1]$ have been chosen uniformly at random b/w $A[\ell]$ & $A[r]$.

We want $E(\#\text{probes}) \leq 2.5$.

Then see that

$P(\#\text{probes} \geq 1) = 1$, $P(\#\text{probes} \geq 2) \leq 1$, &

$$P(\#\text{probes} \geq 3) = P(A[\text{id of } 2^{nd} \text{ probe}] \leq k)$$
$$= P(idx(k) \geq m + \lceil\sqrt{N}\rceil)$$

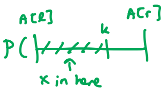
(where $idx(k)$ is a rv that is the largest i w/ $A[i] \leq k$.)

$$\leq P(|idx(k) - m| \geq \sqrt{N})$$
$$= P(|idx(k) - E(idx(k))| \geq t)$$
$$\underset{\uparrow \text{ Chebyshev}}{\leq} \frac{V[idx(x)]}{t^2} \qquad (V(X) = Var(X)).$$

Then, see that $offset(k) = id(k) - \ell$, &

$P(offset(k) = i) = P(\text{exactly } i \text{ of the } r-\ell-1 \text{ randomly chosen numbers are} \leq k)$

Then

$P(\text{one number } x \text{ is } \leq k) = P($  $)$

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

and so $= p$,

$P(offset(k) = i) = \binom{N}{i} p^i (1-p)^{N-i}$. (ie $offset(k) \in Bin(N, p)$)

Thus

- $E[offset(k)] = Np$
- $E[idx(k)] = \ell + Np$
- $V(idx(k)) = V(offset(k)) = Np(1-p) \leq \frac{N}{4}$.

So

$$P(\#\text{probes} \geq 3) \leq \frac{V[idx(k)]}{N} \leq \frac{1}{4},$$

and in general

$$P(\#\text{probes} \geq h) \leq \frac{1}{4(h-2)^2}.$$

So

$$E(\#\text{probes}) = \sum_{h=0}^{\infty} h \cdot P(\#\text{probes} = h)$$
$$= \sum_{h=1}^{\infty} P(\#\text{probes} \geq h)$$
$$\leq \underbrace{1}_{h=1} + \underbrace{1}_{h=2} + \sum_{k=3}^{\infty} \frac{1}{4(h-2)^2}$$
$$= 2 + \frac{1}{4}\sum_{i=1}^{\infty}\frac{1}{i^2} = 2 + \frac{1}{4}\frac{\pi^2}{6} \leq 2.5. \quad ☒$$

4 Therefore,

$$\boxed{T^{avg}(n) \in O(\log\log n).}$$

Proof. Specifically, we prove

$$T^{avg}(n) \leq 2.5\lceil\log\log n\rceil$$

for $n \geq 4$.

Let's consider $L \in \mathbb{Z}^+$ s.t.

$$2^{2^{L-1}} < n \leq 2^{2^L}.$$

$\Rightarrow 2^{L-1} < \log(n) \leq 2^L$

$\Rightarrow L-1 < \log\log(n) \leq L$

$\Rightarrow \lceil\log\log(n)\rceil = L$

Observe that $\sqrt{n} \leq \sqrt{2^{2^L}} = 2^{\frac{1}{2}(2^L)} = 2^{2^{L-1}}$

$\Rightarrow \log\log(\sqrt{n}) \leq L-1 = \lceil\log\log(n)\rceil - 1$.

Proof by induction, and we only consider the step:

$$T^{avg}(n) \leq T^{avg}(n') + 2.5, \qquad n' \leq \sqrt{n}$$
$$\leq 2.5\lceil\log\log(n')\rceil + 2.5$$
$$\leq 2.5\lceil\log\log(\sqrt{n})\rceil + 2.5$$
$$\leq 2.5(\lceil\log\log n\rceil - 1) + 2.5 = 2.5\lceil\log\log n\rceil,$$

as needed. ☒

DICTIONARIES OF WORDS

WORDS

1. "Words" are strings; ie an array of chars in some alphabet Σ.

   eg $\Sigma = \{0,1\}$, $\Sigma =$ ASCII, etc

2. Note words have arbitrary length.

3. We sort words lexicographically:

   c a t f i s h
   c o r n
   c o r n e r

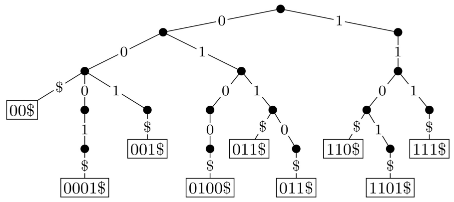
4. Comparing strings (we denote this by 'strcmp') takes $T^{worst} = \Theta(\min\{|w_1|, |w_2|\})$ run-time.

TRIES

1. A "trie" is a dictionary of words.

2. It is "prefix-free" — no string is a prefix of another.

3. This is satisfied if
   ① all strings have the same length; or
   ② all strings end with an "end-of-word" character '$'.



* Note: items (keys) are only stored at the leaves.

OPERATIONS ON TRIE TAKE O(|x|) TIME

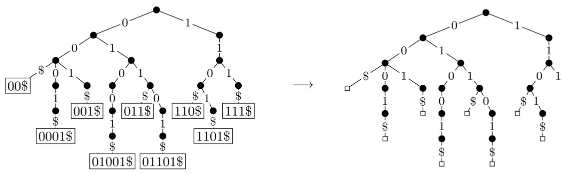
Note that in a trie, search, insert & delete all take O(|x|) time.

NO-LEAF-LABEL TRIES

1. Idea: we don't store the actual keys at the leaves.

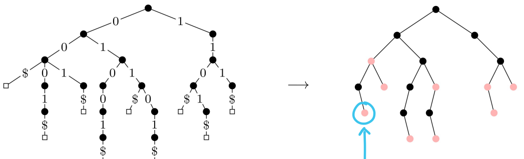
2. The key is stored implicitly through the characters along the path to the leaf.

3. Hence, this halves the amount of space needed.



ALLOW-PROPER-PREFIXES TRIES

1. Idea: we permit storing words at interior vertices, so we can accomodate words that are prefixes of others.

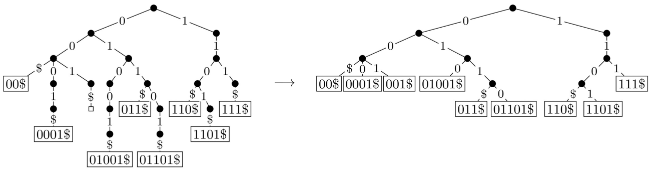


indicates the value of the 'flag' which says whether the node stores a value.

PRUNED TRIES

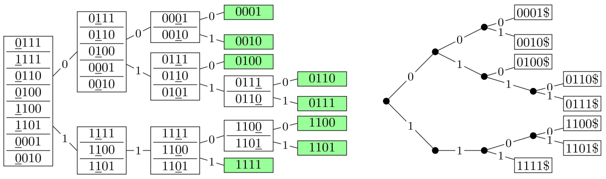
1. Idea: a pruned trie is such that we stop adding nodes as soon as the key is unique.

2. In particular,
   ① a node has a child only if it has two descendants;
   ② we save space if there are only a few long bitstrings; &
   ③ we can store infinite bitstrings (real numbers!)



3. Note that we must store the full keys.

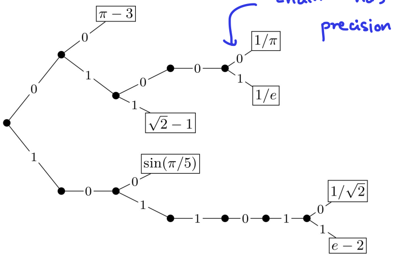
4. Also note that pruned tries are the recursion trees of MSD-radix-sort.



PRUNED TRIES FOR REAL NUMBERS

We can use pruned tries to store infinite length numbers, eg real numbers:

"chain" has "just enough precision"



T IS A PRUNED TRIE THAT STORES n RANDOMLY CHOSEN INFINITE BITSTRINGS $\Rightarrow T^{exp}(\text{search}(B)) = O(\log n)$

As above. (B is an infinite bitstring).

Proof. Say we store $B_1, \dots, B_n$, and search for B.
Then, each bit of each B_i was chosen in $\{0,1\}$ uniformly.

Let the indicator variable

$$I_i = \begin{cases} 1, & \text{if we compared } B[i] \text{ to someone} \\ 0, & \text{otherwise.} \end{cases}$$

In particular,

$$\#\text{ comps for search}(B) = \sum_{i=0}^{\infty} I_i .$$

Then, let

$$I_{i,k} = \begin{cases} 1, & \text{we compared } B[i] \text{ w/ } B_k[i] \\ 0, & \text{otherwise.} \end{cases}$$

Then

$$P(I_{i,k}=1) = P(B \text{ \& } B_k \text{ agree on first } i \text{ bits})$$
$$= \left(\tfrac{1}{2}\right)^i \quad \text{(since bits of } B_i \text{ chosen randomly).}$$

Hence

$$E[I_{i,k}] = P(I_{i,k}=1) = \left(\tfrac{1}{2}\right)^i,$$

and so

$$E[I_i] \leq E\left[\sum_{k=1}^{n} I_{i,k}\right] \leq \sum_{k=1}^{n} \underbrace{E[I_{i,k}]}_{1/2^i} = \frac{n}{2^i}.$$

But since $E[I_i] \leq 1$, hence $E[I_i] \leq \min\{1, \frac{n}{2^i}\}$,
and so

$$E\left[\sum_{i=0}^{\infty} I_i\right] \leq O(1) + \log(n).$$

The rest of the proof follows like the skip-list proof.

COMPRESSED TRIE

1. Here, we compress paths of nodes with only one child.
2. Each node stores an index corresponding to the depth in the uncompressed trie.
3. This gives the next bit to be tested during a search.

* index: which bit of x to compare when searching for x.

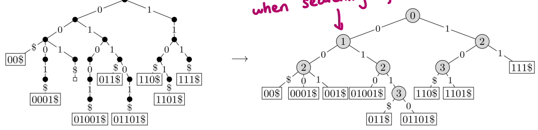


Figure 6.13: An example of a compressed trie.

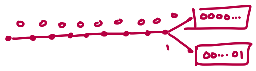
4. In particular, we know every node has ≥2 children.
5. Thus, if the trie has n leaves, then it has ≤ n-1 internal nodes, and so

$$\boxed{\text{total \# of nodes} \leq 2n+1.}$$

6. Hence the trie takes O(n) space, whereas other tries uses space that depends on the length of the words stored.

* if we only consider the space taken by the nodes.

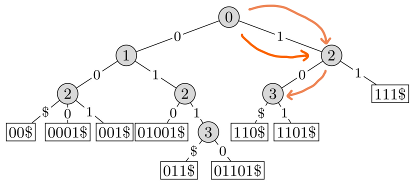
eg Consider



So space is much bigger in a pruned trie.

SEARCH IN COMPRESSED TRIES

1. Consider:



- Search (001$)
  - no such key
- search (0000$)
  - needs to strcmp the leaf
  - then fail
- Search (110$)
  - see A[0] & A[2] (we skip A[1]!)
  - need to do strcmp at the leaf
- search (1$)
  - needs to fail with "too short"
  - since A[2] DNE.

2. Pseudocode:

```
Algorithm 6.8: compressedTrie::search($v \leftarrow root, x$)
  Input : Node $v$ of the trie; word $x$ to search for
if $v$ is a leaf then
    return strcmp($x$, $v$.key)
else
    $d \leftarrow$ index stored at $v$
    if $x$ has at most $d$ bits then
        return "not found"
    else
        let $v'$ be the child of $v$ for which the link from $v$ is labeled with $x[d]$
        if there is no such child then
            return "not found"
        else
            compressedTrie::search($v'$, $x$)
```

3. Run-time: O(|x|), where |x| is the length of the word to search.

INSERT IN COMPRESSED TRIES

1. Idea:
   ① Find where it should be;
   ② Modify trie to put it there.
2. Details are messy & omitted.
3. Run-time: O(|x|).

DELETE IN COMPRESSED TRIES

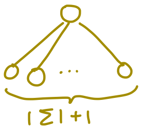
1. Run-time: O(|x|)

PREFIX-SEARCH

1. Given x & a compressed trie, is x a prefix of a stored word?
2. Can also be done in O(|x|) time.

MULTIWAY TRIES

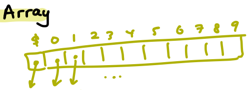
1. These are used to represent larger alphabets.
2. Main question: how do we store the children?



- need to find (during search(x)) the child that stores x[d].

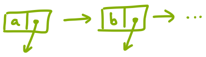
3. Options:

① Array



references to children

② List

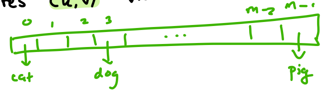


③ Dictionary

Chapter 7: Hashing

DIRECT ADDRESSING

1. Assume each key k is an integer with $0 \le k < M$.
2. Then, we can store the k's by using an array A of size M that stores (k,v) via $A[k] \leftarrow v$.



3. search, insert, delete have $\Theta(1)$ run-time.
4. But the total space is $\Theta(M)$.

HASHING

1. We can do direct addressing after modifying (ie "hashing") the key.
2. Assumptions:
   ① keys come from a universe U
   - typically, U integers, |U| finite

   ② Size of hash table, m, is pre-determined

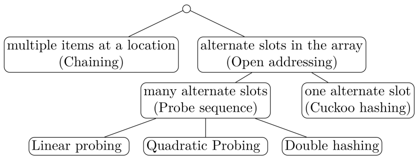
   ③ We use a hash function $h: U \Rightarrow \{0, \ldots, m-1\}$ that maps indexes in U to an index in the array

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

eg $U = \mathbb{N}$, $m = 11$, $h(k) = k \bmod 11$ ↑ slot T[i]

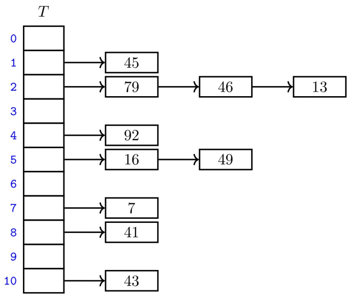
COLLISIONS

1. Collisions occur if we want to insert (k,v) into the table, but $T[h(k)]$ is already occupied.
2. Solutions:



HASHING WITH CHAINING

1. Idea: use lists to resolve collisions.
2. Each slot stores a list.



* we use the MTF heuristic when inserting.

3. Run-time:

   ① Insert: $O(1)$ + time to compute hash-value
   - should choose h(k) so time to compute h(k) is $O(1)$

   ② Search/delete: worst-case $\Theta(\text{length of } T[h(W)])$

   ↳ this is $\Theta(n)$ in worst-case
   - we search in $T[h(k)]$, which is a list

RANDOMIZED VERSION OF HASHING

1. We randomly pick the hash function among all possible hash functions uniformly.
2. This is called the "uniform hashing assumption".
3. Under the uniform hashing assumption,

$$\boxed{E[\text{length of } \underbrace{\text{bucket } i}_{\text{list at } T[i]}] = \frac{n}{m} = \alpha.}$$

Here, "α" is called the "load factor".

Proof. Note $P(h(k) = i)$, for some key k & slot i, is

$$P(h(k) = i) = \frac{1}{m}.$$

We have n keys $\dashrightarrow \frac{n}{m}$.

Thus, the exp. run-time for an unsuccessful search is $O(\alpha)$. ▨

However, if k is in the dictionary, then

$$E[\text{length of bucket } h(k)] = \underbrace{1}_{\substack{\uparrow \\ k \text{ is in} \\ \text{the bucket}}} + \underbrace{\frac{n-1}{m}}_{\le \alpha} \le 1 + \alpha.$$

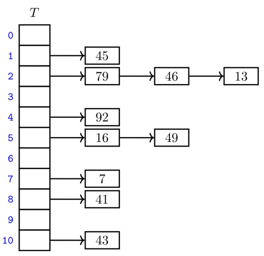
Hence, under uniform hashing,

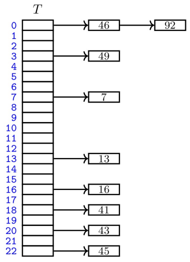
$$\boxed{\text{search, delete take time } O(1+\alpha).}$$

REHASHING

1. We choose α by choosing M. So, as n increases, we increase M so that α stays small.

2. This is called "rehashing".





3. If with rehashing we keep $\alpha \in \Theta(1)$, then all operations take $O(1)$ time.

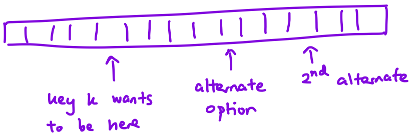
4. Moreover,

$$\boxed{\text{space} \approx M + n = \frac{n}{\alpha} + n = O(n).}$$

HASHING BY PROBING

1. We want to avoid lists, since they have massive overhead.

2. <u>Idea</u>: we allow keys to use multiple slots.



3. For each key, we have a "probe sequence"

$$\boxed{\langle h(k,0),\ h(k,1),\ \dots,\ h(k,M-1) \rangle}$$

key

index of option

LINEAR PROBING

1. Here, we define

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

0	1	2	3	4	5	6	7	8	9	10
	45	13		92	49	37	7	41		43

Consider insert(37).

$\Rightarrow h(k) = 4.$

Probe seq is

$\langle \underset{\times}{\overset{\uparrow}{4}}, \underset{\times}{\overset{\uparrow}{5}}, \underset{\checkmark}{\overset{\uparrow}{6}}, 7, 8, 9, 10, 0, 1, 2, 3 \rangle.$

2. Linear probing builds big "clusters" of elements.

OPERATIONS OF PROBE HASHING

1. In the array,

① To delete an key k, "mark" it as deleted.

② To search, we
- follow the probe sequence:
- ignore any "deleted" entries; &
- continue until we find k or NIL.

③ To insert, we
- follow the probe sequence; and
- continue until we find a vacant spot (ie empty / deleted) or we reach the end of the probe sequence.

* this is called the "lazy deletion" technique.

2. We also track how many elements are "deleted"; if this gets too large, we re-hash.

3. Pseudocode:

```
Algorithm 7.1: probeSequenceHash::insert(k, v)
re-hash the table the load factor is too big
for (j = 0; j < M; j++) do
    if T[h(k, j)] is NIL or 'deleted' then
        T[h(k, j)] ← (k, v)
        return "item inserted at index h(k, j)"
    else
        return "failure to insert"                    // need to re-hash
```

```
Algorithm 7.2: probeSequenceHash::search(k)
for (j = 0; j < M; j++) do
    if T[h(k, j)] is NIL then
        return "item not found"
    else if T[h(k, j)] has key k then
        return "item found at index h(k, j)"
    else        // the current slot was 'deleted' or contains a different key
        // ignore this and keep searching
return "item not found"
```

```
Algorithm 7.3: probeSequenceHash::delete(k)
i ← index returned by probeSequenceHash::search(k)
T[i] ← 'deleted'
re-hash the table if there are too many 'deleted' items
```

QUADRATIC PROBING

1. Here, we define

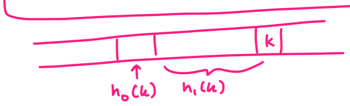
$$\boxed{h(k,i) = (h(k) + c_1 i + c_2 i^2) \bmod M}$$

for constants c_1, c_2, which are picked so the probe sequence visits all slots in the hash table.

DOUBLE HASHING

1. "Double hashing" uses two hash functions h_0, h_1, and we define

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



2. Requirements:
① $h_1(k) \neq 0$;
② $h_1(k)$ is relatively prime with M;
③ h_1 & h_0 should be independent.

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

$\varphi = \frac{\sqrt{5}-1}{2}$

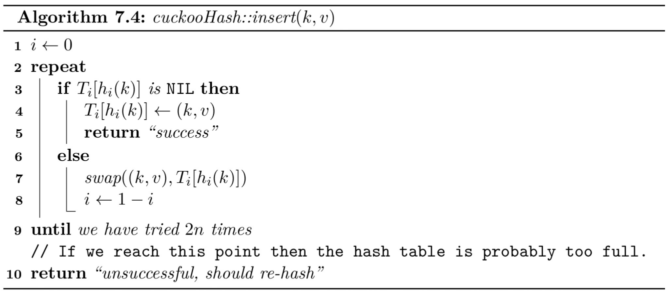
CUCKOO HASHING

1. In "cuckoo hashing", we use two hash tables & two hash functions, and promise key k is <u>always</u> at $T_0[h_0(k)]$ <u>or</u> $T_1[h_1(k)]$.

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

2. In particular, <u>search</u> & <u>delete</u> have $O(1)$ worst-case run-time.

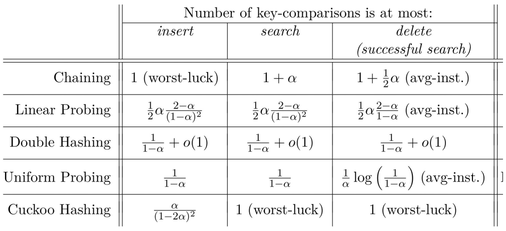
3. Pseudocode for insert:



```
Algorithm 7.4: cuckooHash::insert(k, v)
i ← 0
repeat
    if $T_i[h_i(k)]$ is NIL then
        $T_i[h_i(k)] ← (k, v)$
        return "success"
    else
        $swap((k, v), T_i[h_i(k)])$
        $i ← 1 − i$
until we have tried 2n times
// If we reach this point then the hash table is probably too full.
return "unsuccessful, should re-hash"
```

COMPLEXITY OF OPEN ADDRESSING STRATEGIES

1. Note:



	Number of key-comparisons is at most:		
	insert	*search*	*delete* (*successful search*)
Chaining	1 (worst-luck)	$1+\alpha$	$1+\frac{1}{2}\alpha$ (avg-inst.)
Linear Probing	$\frac{1}{2}\alpha\frac{2-\alpha}{(1-\alpha)^2}$	$\frac{1}{2}\alpha\frac{2-\alpha}{(1-\alpha)^2}$	$\frac{1}{2}\alpha\frac{2-\alpha}{1-\alpha}$ (avg-inst.)
Double Hashing	$\frac{1}{1-\alpha}+o(1)$	$\frac{1}{1-\alpha}+o(1)$	$\frac{1}{1-\alpha}+o(1)$
Uniform Probing	$\frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\log\left(\frac{1}{1-\alpha}\right)$ (avg-inst.)
Cuckoo Hashing	$\frac{\alpha}{(1-2\alpha)^2}$	1 (worst-luck)	1 (worst-luck)

2. Thus, all operations have $O(1)$ expected run-time if hash function is chosen randomly & α is kept sufficiently small.

3. But for a fixed hash function, the worst-case run-time is $\Theta(n)$.

CHOOSING HASH FUNCTIONS

MODULAR METHOD

1. Here, we let

$$\boxed{h(k) = k \bmod M,}$$

where we usually pick M to be prime.

*don't pick $M=2^m$ or $M=10^m$!

MULTIPLICATION METHOD

1. Here, we let

$$\boxed{h(k) = \lfloor \underbrace{M(\underbrace{A\cdot k - \lfloor A\cdot k \rfloor}_{\in[0,1)})}_{\in[0,M)} \rfloor}$$

where
① $M = 2^k$ for some $k \geq 0$; &
② $A \in (0,1)$ (preferably an irrational).

HASHING MULTI-DIMENSIONAL DATA

1. Suppose we wanted to hash keys that are words (ie in Σ^*).

2. We can "flatten" the string w into an integer:

eg A·P·P·L·E $\rightarrow$ (65, 80, 80, 76, 69) (ASCII)

$\rightarrow 65R^4 + 80R^3 + 80R^2 + 76R + 69$, $R=255$

3. We can then combine this with a modular hash function (ie $h(w) = f(w) \bmod M$).

4. To compute this in $O(|w|)$ time without overflow, we use "<u>Horner's rule</u>" and apply mod early:

ie $((((((((65R+80) \bmod M)R+80) \bmod M)R+76) \bmod M)R+69) \bmod M$

RANDOMLY-CHOSEN HASH FUNCTIONS

1. There are $|U|^M$ many possible hash functions, & ideally we would choose randomly amongst them.

2. But then we cannot compute the hash value quickly!

3. Idea: Fix a family $\mathcal{H}$ of hash-functions that are easy to compute, & choose uniformly among them.

UNIVERSAL HASH-FUNCTIONS

For analysis, we want uniform hash-values; ie

$$\boxed{P(h(k) = i) = \frac{1}{m}.}$$

But we also need small probability of collisions (ie "universal hashing"); in other words.

$$\boxed{P(h(k) = h(k')) \leq \frac{1}{m} \quad \forall k \neq k'}$$

CARTER-WEGMAN HASH FUNCTION

The "Carter-Wegman hash-function" is defined to be

$$\boxed{\begin{aligned} h_{a,b}(k) &= f_{a,b}(k) \bmod m \\ &= (a \cdot k + b \bmod p) \bmod m \end{aligned}}$$

where $k \in \mathbb{Z}_p$, p is prime, & $a \neq 0, b$ are chosen randomly.

We claim $f_{a,b}$ defines a permutation of $\mathbb{Z}_p$; ie $f_{a,b}(k) \neq f_{a,b}(k')$ for $k \neq k'$.

Proof. Assume $f_{ab}(k) = f_{ab}(k')$.

$$\Rightarrow (ak+b) \% p - (ak'+b) \% p = 0 \qquad (\% p \Leftrightarrow \bmod p)$$

$$\Rightarrow ak + b - ak' - b \equiv_p 0 \qquad (\equiv_p \Leftrightarrow (\bmod p))$$

$$\Rightarrow a(k-k') \equiv_p 0$$

Since $a \in \{1, \dots, p-1\}$ & $k-k' \in \{-(p-1), \dots, (p-1)\}$, thus

$$k-k' = 0,$$

ie $k=k'$, and we're done. ☒

CARTER-WEGMAN FUNCTIONS ARE UNIVERSAL

See that

$$\boxed{P(h_{a,b}(k) = h_{a,b}(k')) \leq \frac{1}{m}.}$$

Proof. Assume $h_{a,b}(k) = h_{a,b}(k')$ for $k \neq k' \in \mathbb{Z}_p$.

We know that

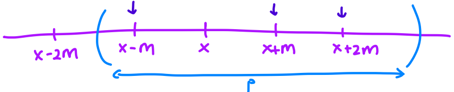
$$\underbrace{f_{a,b}(k)}_{x} \neq \underbrace{f_{a,b}(k')}_{x'},$$

but $x \% m = x' \% m$ (by def^{n} of $h_{a,b}$).

Thus

$$x - x' \equiv_m 0.$$

How many such "bad" pairs (x, x') could there be in $\mathbb{Z}_p \times \mathbb{Z}_p$?



x' is one of these, but $x' \neq x$.

Hence, x' is among $\lceil \frac{p}{m} \rceil - 1$ numbers, and so

$$\# \text{ choices for } x' \text{ is } \leq \frac{p-1}{m}.$$

Fixing x, it follows that

$$\# \text{ bad pairs} \leq \underbrace{p}_{\text{choices for } x} \cdot \underbrace{\frac{p-1}{m}}_{\text{choices for } x'}.$$

Therefore

$$\begin{aligned} P(h_{a,b}(k) = h_{a,b}(k')) &= P((x,x') \text{ formed a "bad" pair}) \\ &= \sum_{\text{bad pairs } x,x'} P(f_{ab}(k) = x, \; f_{ab}(k') = x') \\ &\quad \dots \\ &= \sum_{\text{bad pairs } x,x'} P(a = (k-k')^{-1}(x-x') \% p, \; b = (x-ak) \% p) \\ &= \sum_{\text{bad pairs } x,x'} \frac{1}{p-1} \cdot \frac{1}{p} \\ &= \frac{1}{p(p-1)} \cdot \# \text{ bad pairs} \leq \frac{1}{m}. \end{aligned}$$

☒

Chapter 8: Range Search

Idea$_1$:

① we are given two keys $x_1 \leq x_2$; &

② We want to get all items between x_1 & x_2.

Sorted array$_2$:

5 10 12 17 23 32 45 62 71 ...

> rangeSearch(13, 36)

① Search for x_1
→ will tell us where x_1 would be "between"

② Search for x_2
→ likewise.

③ Repeat everyone in between.

* this approach works for tries, SL's, etc.

Run-time$_3$: $O(\log n + s)$, where "s" is the "output size".

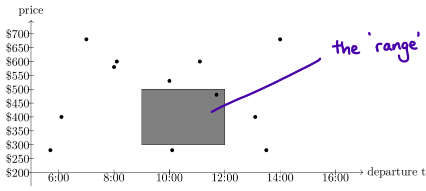
MULTI-DIMENSIONAL DATA

Idea$_1$: "points" instead of single values.

ie $(x_0, x_1, \dots, x_n)$

* we will assume all data are integers.

Range search then looks like$_2$:



the 'range'

This is called an "orthogonal range-search"$_3$:

① We are given a query rectangle $\boxed{A = [x, x'] \times [y, y'];}$ &

② We want the points in A.

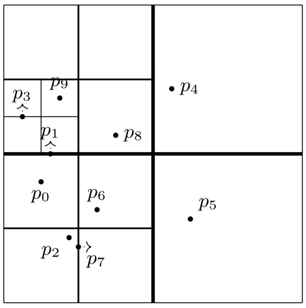
QUAD-TREES

For "quad-trees"$_1$:

① stores points in 2D;

② we assume the points are in a bounding box $[0, 2^{\ell}) \times [0, 2^{\ell}]$

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

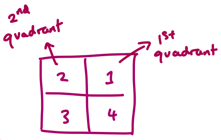


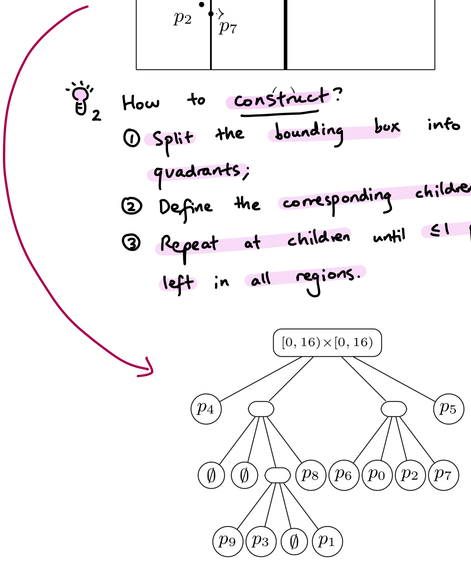
How to construct?$_2$

① Split the bounding box info quadrants;

② Define the corresponding children;

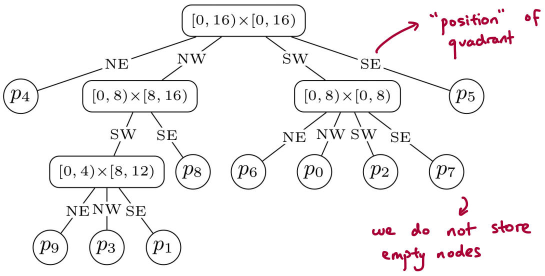
③ Repeat at children until ≤1 point left in all regions.





Note$_3$: points on the split line go above/right.

Alternative depiction$_4$:



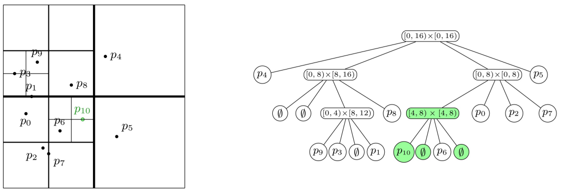
"position" of quadrant

we do not store empty nodes

① We do not need to store the "sub-areas" (we always cut in half)

OPERATIONS OF QUAD-TREES

1. **Insert:**



2. **Run-time:**

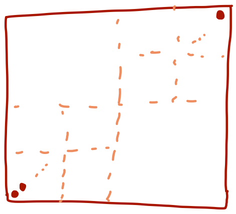
① No good bound depending on n;

② We can only say it is O(tree height)

HEIGHT OF A QUAD-TREE

1. But we have no bound on the height either!

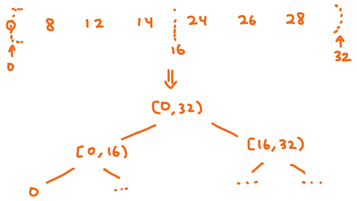
eg



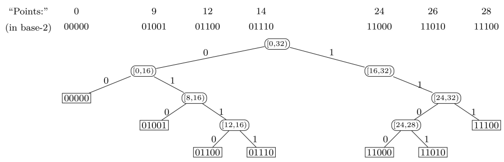
→ height arbitrary.

2. However, we can bound the height if the coordinates are integers; specifically in the range $\{0, \ldots, 2^\ell - 1\}$. Then the height of the quad-tree is $\leq \ell$.

Proof. Detour: let's look at a quad-tree in 1d.



But we can also represent the points in base-2:



This is a pruned trie!

The height of a 1d quad-tree of $\underbrace{\text{integers in } \{0, \ldots, 2^\ell - 1\}}_{\text{bitstrings of length } \ell}$

$\leq$ the length of longest bitstring $= \ell$

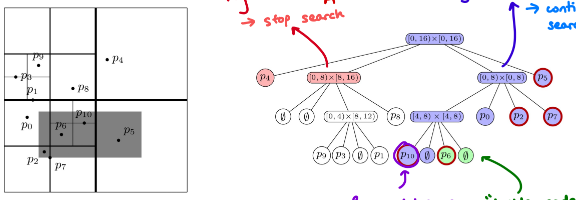
This argument generalizes to higher dimensions (eg 2D, ie quad trees)

3. In particular, a quad-tree is a pruned trie where we split by two keys in parallel.

4. The expected height of a quad-tree of randomly chosen points is O(log n).

RANGE-SEARCH IN QUAD-TREES

1. **Idea:**



"outside" node: region disjoint from A → stop search

"boundary" node: region overlaps with A → continue search

if boundary node & contains a single point, we just check explicitly

"inside node": point fully inside A → all points in range

2. **Pseudocode:**

```
Algorithm 8.1: quadTree::rangeSearch(r ← root, A)
Input   : Node r of a quad-tree. Query-rectangle A
R ← square associated with node r
if R ⊆ A then                                          // inside node
    report all points in subtree at r and return
else if R ∩ A is empty then                            // outside node
    return
else                                                   // boundary node
    if r is a leaf then
        if r stores a point p and p ∈ A then
            return p
        else
            return
    else
        foreach child v of r do
            quadTree::rangeSearch(v, A)
```

① Checking "$R \cap A = \emptyset$" is O(1).

- since we use rectangles.

3. **Run-time:**

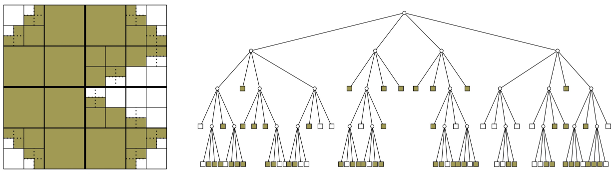
① We have no bounds in terms of n or s.

② Only thing we can say:

run-time ∈ O(size of quad tree).

OTHER USES OF QUAD-TREE

1. We can use quad-trees to store pixelated images:



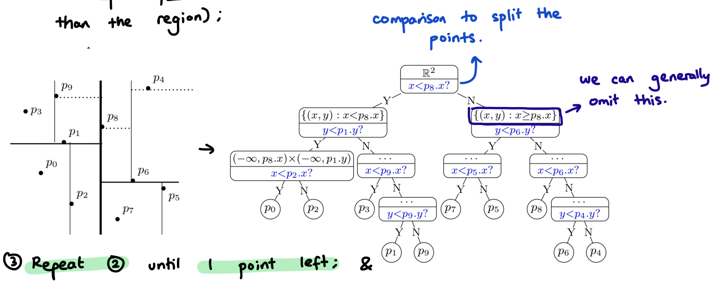
kd-TREE

* d = "dimensional"

Idea:

① Similar to quad-trees; but

② We split points in half (rather than the region);



comparison to split the points.

we can generally omit this.

③ Repeat ② until 1 point left; &

④ Alternate split by x & y.

Note: we always split at QuickSelect$(\lfloor \frac{n}{2} \rfloor)$.

ie the median if n is odd, & the upper median if n is even.

We assume the points are in "general position": no two x coordinates are the same, nor no two y coordinates are the same.

Under this, each split leads to

① $\lfloor \frac{n}{2} \rfloor$ points in the left child; &

② $\lceil \frac{n}{2} \rceil$ points in the right child.

Therefore, the height is $O(\log n)$.

OPERATIONS OF kd-TREE

Search: $O(\log n)$

Insert / delete: basically impossible!

However, we can build the whole thing, given all n points, in $O(n \log n)$ expected time:

① Find the median; &

→ $O(n)$ expected

② Recurse in children

→ size ≈ $\frac{n}{2}$ each.

RANGE-SEARCH FOR kd-TREES

Pseudocode:

```
Algorithm 8.2: kdTree::rangeSearch(r ← root, A)
Input  : Node r of a kd-tree. Query-rectangle A
R ← region associated with node r
if R ⊆ A then                                              // inside node
    report all points in subtree at r and return
else if R ∩ A is empty then                                // outside node
    return
else                                                       // boundary node
    if r is a leaf then
        if the point p stored at r satisfies p ∈ A then
            return p
        else
            return
    else
        foreach child v of r do
            kdTree::rangeSearch(v, A)
```

* nearly identical to that in quad-trees.

Run-time: $O(\# \text{ of boundary nodes} + \underbrace{s}_{\text{output size}})$.

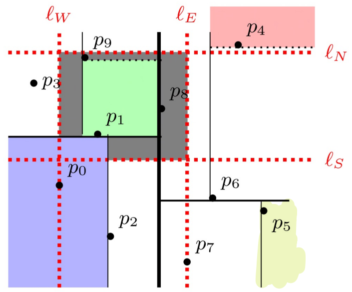
Then, we see that

$$\boxed{\# \text{ of boundary nodes} \in O(\sqrt{n}),}$$

so

$$\boxed{\text{run-time} = O(\sqrt{n} + s).}$$

Proof.



How many nodes have an associated region that intersects one of these lines?

If the point does not intersect, then either

- it is an outside region (eg p_5); or
- it is an inside region (eg p_1):

ie

boundary nodes ≤ nodes whose associated region intersects one of ℓ_W, ℓ_E, ℓ_N, ℓ_S.

Let $Q(n)$ = # of boundary nodes.

Let $Q(n, \ell) = \max_{\text{kd-trees w/ n pts}}$ that intersect a given line ℓ.

This is independent of ℓ (shift points), so only consider whether ℓ is horizontal / vertical ⇝ $Q_v(n)$, $Q_h(n)$.

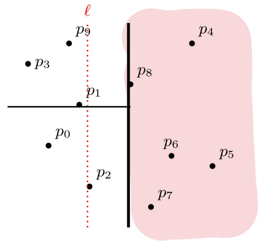
Then

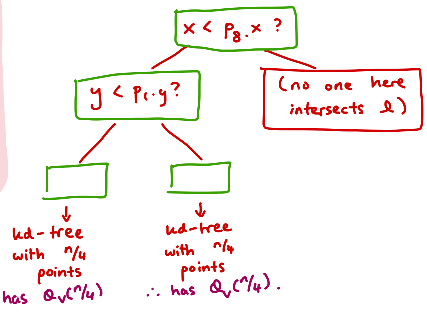
$$Q(n) \leq Q(n, \ell_W) + Q(n, \ell_N) + Q(n, \ell_E) + Q(n, \ell_S)$$
$$\leq 2Q_v(n) + 2Q_h(n).$$

So, if we prove $Q_v(n) \in O(\sqrt{n})$ (and symmetrically $Q_h(n) \in O(\sqrt{n})$), we are done.

See that $Q_v(n)$ = # of nodes whose associated region intersects a vertical line ℓ.

Consider making the kd-tree:





kd-tree with $n/4$ points ∴ has $Q_v(n/4)$

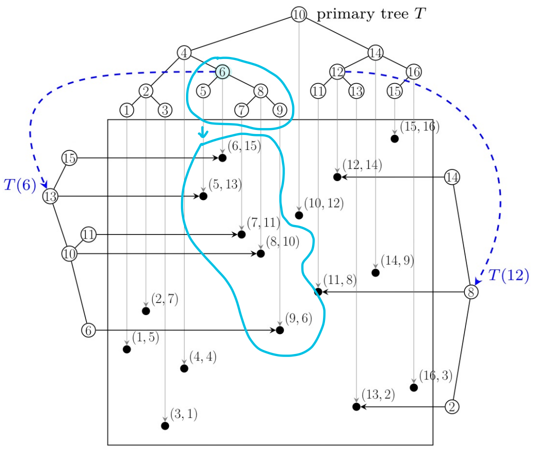
kd-tree with $n/4$ points ∴ has $Q_v(n/4)$.

Therefore

$$Q_v(n) \leq 2Q_v(n/4) + 2.$$

This resolves to $O(\sqrt{n})$, as needed. ▣

RANGE TREE



Idea:

① We store points in a "primary tree" T: a BST by the x-coordinate. (This is a scapegoat tree).

② For every node in T, we have a BST associated with it.

- let P(v) be the subtree at v.
- T(v) stores P(v) in a balanced BST, by the y-coordinate.

Space: $O(n \log n)$ (assuming primary tree has height $O(\log n)$.)

Proof. How often is each point stored?

- once in the primary tree
- once in each associated tree of each ancestor of x. (there are $O(\log n)$ such ancestors).

∴ Each point is stored $O(\log n)$ times

∴ space = $O(n \log n)$.

* this is tight.

DICTIONARY OPERATIONS

Search: just search the primary tree.

- balanced (so time = $O(\log n)$).

Insert:

① insert into the primary tree, say at node x

② at all ancestors of x, insert the point into the associated tree.

- we have $O(\log n)$ ancestors, and each insert takes $O(\log n)$ time
- thus time = $O(\log^2 n)$.

③ if the primary tree is inbalanced, rebuild that subtree & all associated subtrees.

- we can show the amortized time for insert is still $O(\log^2 n)$.

Delete: similar to insert.

- takes time = $O(\log^2 n)$.

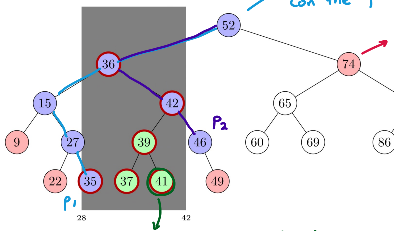
RANGE-SEARCH

Steps:

① A range-search in the primary tree (with a small twist) by the x-coord.

- search for left boundary (gives a path P_1)
- search for right boundary (gives a path P_2)

boundary node (on the paths)



outside node (left of P_1 or P_2) → nobody in subtree in the range

inside node (between P_1 & P_2)
- all descendants in range

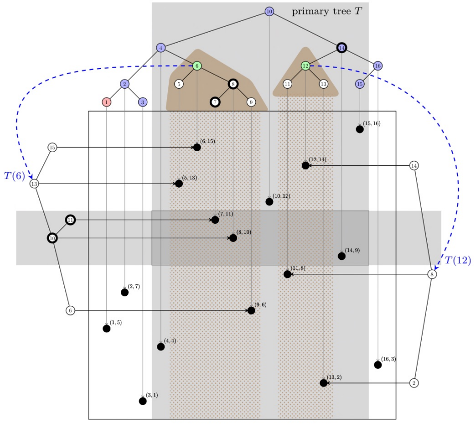
- this has run-time $O(\log n + s)$ (to get the classification & to report if wanted)

② Return all the boundary nodes & the top-most inside nodes.

③ For the boundary nodes, we explicitly check.

④ For the inside nodes,

- all points are in range with x-coord.
- so, we run a range-search on the associated tree by the y-coord.



Run-time: $O(\log^2 n + s)$.

- we do one range-search in the primary tree ($O(\log n)$)
- we do range searches in $O(\log n)$ associated trees (each of these takes $O(\log n)$ time → $O(\log^2)$ time
- report all points in range:

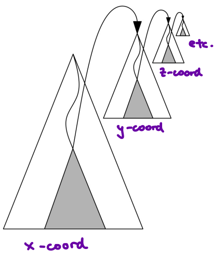
$O(\log^2 n)$ [
→ boundary nodes in primary tree
→ boundary nodes in same associated tree we searched in

$O(s)$ [→ inside nodes in some associated tree
- only need to report

HIGHER DIMENSIONS

Run-time of range search: $O(\log^d n + s)$

Space: $O(\log^d n \cdot n)$



3-SIDED RANGE SEARCH

Idea: return (x,y) with $x_1 \le x \le x_2$ and $y \ge y'$.

Naive approach: Range-tree.

- runtime of range-search = $\Theta(\log^2 n + s)$
- space = $\Theta(n \log n)$.

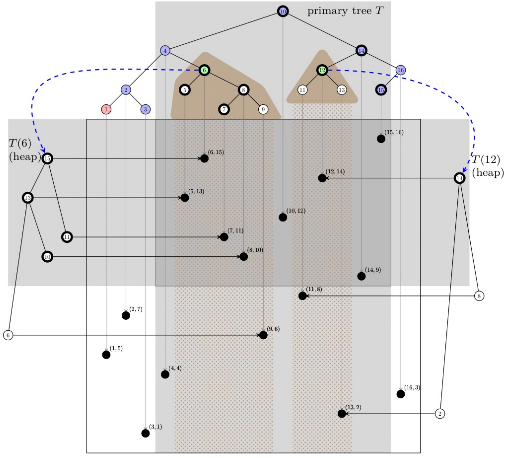
ASSOCIATED HEAPS

Idea:

① Primary tree: balanced BST
② Associated tree: binary heap
③ Space: $\Theta(n \log n)$

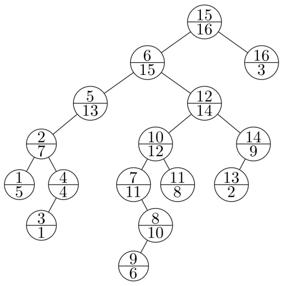
Range search:

① Search in primary tree as before; &
② In associated heaps: search by y-coordinate in $O(1+s)$ time.
③ Run-time: $O(\log n + s)$.



CARTESIAN TREES

Idea: Use a treap, but use x-coordinate as the key & the y-coordinate as the priority.



Space: $\Theta(n)$

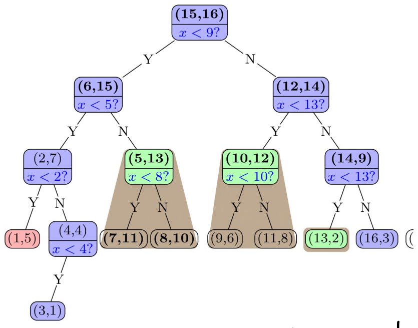
Range-search:

① Do BST::range-search(x_1, x_2) to get boundary and topmost inside nodes.
② For each inside node, do 1-sided search in heap at that node by y-coordinate.
③ Run-time: $\Theta(\text{height} + s)$

PRIORITY SEARCH TREES

Idea:

① Store y-coordinates in "heap-order"
② Split in half by x-coordinate
- split-line coordinates: use median x-coordinate among the points in subtree
- so height = $\lceil \log n \rceil$



Note: x-coordinate stored for splitting can be different from the x-coordinate of the stored point.

Range-search time: $O(\log n + s)$.

Space: $\Theta(n)$.

SUMMARY OF 3-SIDED RANGE SEARCH ADTS

Summary:

	associated heaps	Cartesian tree	priority search tree
time for range-search	$O(\log n + s)$ (optimal)	$O(\text{height} + s)$ (height is bad)	$O(\log n + s)$ (good!)
space	$\Theta(n \log n)$ (bad!)	$\Theta(n)$ (good!)	$\Theta(n)$ (good!)
insert/delete	$\Theta(\log^2 n)$ (amort.)	$\Theta(\text{height})$	$O(\log n)$ (feasible – worst case)
build from scratch	$\Theta(n \log n)$	sort by x-coord + $O(n)$ time	$O(n \log n)$ or sort + $O(n)$

Chapter 9:
Pattern Matching

Idea:

① Given text $T[0 \dots n-1]$ of length n & a pattern $P[0 \dots m-1]$ of length m

② Want to know: is P a substring of T?

eg T : abbaabaab P : ab

substring prefix

substring

substring suffix

In this course: report one occurrence (usually the leftmost).

(IRL: report all of them)

BRUTE-FORCE

Idea: check every possible guess.

```
Algorithm 9.1: bruteForce::patternMatching(T, P)
Input  : Text T of length n, Pattern P of length m
for $i \leftarrow 0$ to $n - m$ do
    if $strcmp(T[i..i+m-1], P) = 0$ then return found at guess i
return FAIL
```

Run-time:

① Strcmp takes $\Theta(m)$ time.

② Worst possible input: $P = a^{m-1}b$, $T = a^n$

③ So worst case runtime $= \Theta((n-m+1)m)$

④ This is $\Theta(mn)$ if $m = n/2$.

PRE-PROCESSING IDEA

Idea: break a problem into 2 parts:

① Build a data structure / info that will make later queries easy.
(this part can be slow)

} preprocessing

② Do the actual query.
(this part can be fast)

How to improve?

① Do extra preprocessing on the pattern P
- eliminate guesses based on completed matches & mismatches

Karp-Rabin, Boyer-Moore, DFA, KMP (length m)
→ used for web searches

② Do extra preprocessing on the text T
- we create a data structure to find matches easily

Suffix tree, Suffix array (length n)
→ bioinformatics

KARP-RABIN FINGERPRINT ALGORITHM

Idea: Use hashing to eliminate guesses

① Compute hash function for each guess, compare with pattern hash

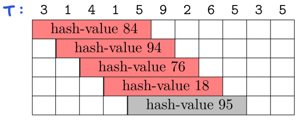
② If values are unequal, guess cannot be an occurrence

eg $P = 5\ 9\ 2\ 6\ 5$ $T = 3\ 1\ 4\ 1\ 5\ 9\ 2\ 6\ 5\ 3\ 5$

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

$h(x_0 \cdots x_4) = (x_0 \cdots x_4)_{10} \bmod 97$

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



If P occurs at guess $i \Rightarrow h(T[i \dots i+m-1]) = h(P)$.

First attempt:

```
Algorithm 9.2: KarpRabin::patternMatching-naive(T, P)
$h_P \leftarrow$ word-hash-value(P, 10, M)
for $i \leftarrow 0$ to $n - m$ do
    $h_T \leftarrow$ word-hash-value(T[i..i+m-1], 10, M)
    if $h_T = h_P$ then
        if $strcmp(T, P, i, i+m-1) = 0$ then
            return "found at guess i"
return FAIL
```

- never misses a match
- $h(T[i \dots i+m-1])$ depends on m characters, so naive computation takes $\Theta(m)$ time per guess
- running time $= \Theta(mn)$ if P not in T.

Idea to be faster: our hash function is a "rolling" hash function.

- ie given $h(T[i \dots i+m-1])$, compute $h(T[i+1 \dots i+m])$ quickly.

eg Know $h(41592) = 76$, what is $h(15926)$?

$15926 \bmod 97 = \left[(41592 \bmod 97 - 4(10000) \bmod 97) * 10 + 6\right] \bmod 97$

- in general,

$$\boxed{T[i+1 \dots i+m] \bmod M = \left[(T[i \dots i+m-1] \bmod M - T[i] \cdot \underbrace{10^m}_{} \bmod M) * 10 + T[i+m]\right] \bmod M.}$$

next hash value

h(previous guess)

precompute

Second attempt:

```
Algorithm 9.3: KarpRabin::patternMatching(T, P)
Input  : T and P are texts of length n and m over alphabet $\Sigma = \{0, \dots, R-1\}$
bool needToReset ← TRUE
for $i \leftarrow 0$ to $n - m$ do
    if needToReset then          // get random prime and initialize fingerprints
        $M \leftarrow$ prime number randomly chosen in $\{1, \dots, mn^2 \log R\}$
        $h_P \leftarrow$ word-hash-value(P, R, M)
        $h_T \leftarrow$ word-hash-value(T[i..i+m-1], R, M)
        $s \leftarrow$ word-hash-value('10...0', R, M) // passed word has m chars
        needToReset ← FALSE;
    else                                     // get next fingerprint from previous
        $h_T \leftarrow ((h_T - T[i-1] \cdot s) \cdot R + T[i+m-1]) \bmod M$
    if $h_T = h_P$ then
        if $strcmp(T, P, i, i+m-1) = 0$ then
            return "found at guess i"
        else
            needToReset ← TRUE;
return FAIL
```

- choose "table size" M to be a random prime in $\{2, \dots, mn^2\}$
- $T^{exp} = O(m+n)$
- improvement: reset M if no match at $h_T = h_P$.

KARP-RABIN ANALYSIS

1. If we never match hash-value:
   - ① $\Theta(m)$ preprocessing
   - ② $\Theta(1)$ per guess

   $\therefore$ total runtime $= \Theta(m+n)$.

2. If $h_P = h_T$ and there was a match:
   - ① $\Theta(m)$ preprocessing
   - ② $\Theta(1)$ per guess
   - ③ $\Theta(m)$ strcmp & break

   $\therefore$ total run-time $= \Theta(m+n)$

3. **Worst-case:** If $h_P = h_T$ but no match (false positive)
   - if this happens $\Theta(n)$ times, then run-time is $\Theta(m(n-m))$.

4. We can show

   $$\boxed{P(\geq 1 \text{ false positive}) \leq \frac{2c}{n}}$$

   for a constant c.

5. Thus

   $$\boxed{\begin{aligned} T^{exp} &\leq P(\text{false positive}) \cdot (n+m) + P(\text{false positive}) \cdot n \cdot m \\ &\leq n+m + 2c(m) \\ &\in O(n+m). \end{aligned}}$$

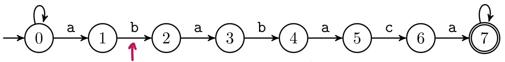
6. Auxiliary space: $O(1)$.

DFA / NFA

1. Idea: Use a DFA to pattern match.
   - we parse T, and reach state Ⓜ <=> P occurs in T.

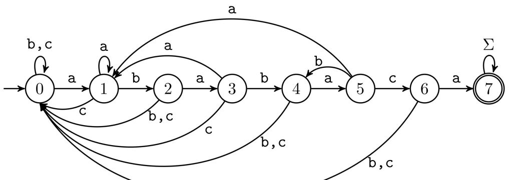
2. We set our DFA up so state j means we have just now seen $P[0...j-1]$ in what was parsed.

$\Sigma = \{a,b,c\}$



each "forward arc" is labelled with $P[j]$

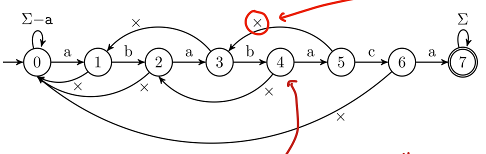
3. We can show thee is an equivalent small DFA.



- then, it is easy to test whether P is in T.

KNUTH-MORNS-PRATT / KMP

1. DFA: would work, but complicated.
2. Idea: use a new kind of finite automata.



"failure":
- use this transition if no other fits
- does not consume a character
- every state has ≤ 1 failure arc

"have seen $P[0...3]$"

3. With these rules, computations of the automaton are deterministic.
4. We want to set up the failure arcs so that P is in T <=> we are at Ⓜ at some point.

KMP ALGORITHM

1. Pseudocode:

preprocessing

```
Algorithm 9.4: KMP::patternMatching(T, P)
   F ← failureArray(P)
   i ← 0                              // index of current character of T to parse
   j ← 0                              // index of current state
   while i < n do
       if P[j] = T[i] then  → "can I use the forward arc"
           if j = m − 1 then                  // all characters were matched
①              return "found at guess i − m + 1"
           else                                  // go along forward-arc
②              i ← i + 1  ]  use forward arc
②              j ← j + 1  ]
       else
           if j > 0 then             // go along failure-arc from state j
③              j ← F[j − 1]
           else                      // at leftmost state; go along loop
④              i ← i + 1
   return FAIL
```

follow failure arc
- $F[j-1]$: stores where the failure arc from state j goes
  - * note the j-1!!

we are at state 0, the initial state
- just consume (character is "useless")

2. We can show

   $$\boxed{\# \text{ of executions of while loop} \leq 2i - j + \chi(\text{found match})}$$

   where $\chi(\text{found match})$ is 1 if we found a match, & 0 otherwise.

   Proof. Initially, $i=j=0$ & no executions of while-loop.
   Assume we had $\leq 2i-j$ executions at some later time (the "inductive hypothesis").
   In the next execution, we had 4 cases: (see above)

   ① We found pattern P.
   Then
   # of executions $\leq 2i-j+1 = 2i-j+\chi(\text{found match})$.

   ② We use the forward arc.
   $\Rightarrow i' = i+1, \; j' = j+1 \Rightarrow 2i'-j' = 2i-j+1$
   $\geq$ # of executions.

   ③ We use the failure arc.
   $\Rightarrow$ j decreases, i is the same
   So $2i'-j' \geq 2i-j+1$

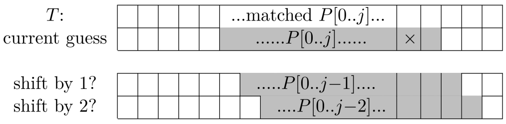
   ④ We consume $T[i]$.
   $\Rightarrow$ i increases, j stays the same
   So $2i'-j' \geq 2i-j+1$.

3. Thus,

   $$\boxed{\text{run-time without computation of failureArray is } O(\max i) = O(n).}$$

KMP FAILURE ARRAY

1 Assume we reach state $j+1$ and now have a mismatch.



- we can eliminate "shift by 1" if $P[1...j] \neq P[0...j-1]$.
- in general, we can eliminate "shift by k" if $P[1...j]$ does not end with $P[0...j-k]$

2 So, we want the longest prefix $P[0... \ell-1]$ that is a suffix of $P[1...j]$;

3 Thus our failure-function is defined by

> $F[j]$ = head of failure arc from state $j+1$
> = length of longest prefix of P that is a suffix of $P[1...j]$.

COMPUTING THE FAILURE-ARRAY QUICKLY

1 Recall: we reach state ℓ
⟺ we have seen $P[0... \ell-1]$
⟺ $P[0... \ell-1]$ is a suffix of what was parsed

2 Let's instead consider parsing $P[1...]$.

So

> $F[j]$ = state reached when parsing $P[1...j]$ on the KMP-automaton for P.

3 Pseudocode:

Algorithm 9.5: *KMP::failureArray(P)*

```
F ← array of length $m$; initialize $F[0] ← 0$
$j ← 1$  → start at index 1      // index of current character of $P[1..m]$ to parse
$\ell ← 0$                                        // index of current state
while $i < m$ do  → does the next character match?
    if $P[\ell] = P[j]$ then                     // go along forward-arc
        $\ell ← \ell + 1$
        $F[j] ← \ell$  → set failure function     // Have parsed $P[1..j]$ and reached $\ell$
        $j ← j + 1$
    else
        if $\ell > 0$ then                        // go along failure-arc
            $\ell ← F[\ell - 1]$  → we have already computed F[ℓ-1] if we get here
        else                                      // at leftmost state
            $F[j] ← 0$
            $j ← j + 1$
```

- in particular, we can use the KMP-automaton to build itself.

4 Run-time = time to parse $P[1... m-1] = O(m)$ time.

5 Therefore,

> total worst-case time for KMP = $O(m+n)$
> aux. space = $O(m)$ (failure array)

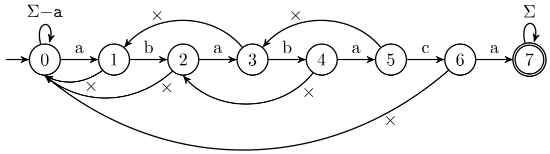
USING KMP-AUTOMATON TO COMPUTE THE DFA IN O(n) TIME

Note: from the KMP-automaton, we could compute the DFA in $O(|\Sigma| \cdot m)$ time.

IMPROVING THE FAILURE FUNCTION

1 We can also improve the failure-function.

2 Consider the "bad case":



Suppose we reach state 4, and the next char is not a.
→ failure arc goes to state 2
→ forward arc at state 2 also has 'a'.
We will fail again! ⇒ go to $F[1]$ right away.

3 Improved failure-function:

> $F^+[j] = \begin{cases} \text{length } \ell \text{ of the longest prefix of } P \text{ that is a suffix of } P[1...j] \text{ and } \underline{P[\ell] \neq P[j+1]}. \\ \text{or } 0 \text{ if no such } \ell \text{ exists} \end{cases}$

4 F^+ takes into account whether forward arcs mismatch.

5 To compute F^+:

$$F^+[j] = \begin{cases} F[j], & P[j+1] \neq P[F[j]] \text{ or } F[j]=0 \\ F^+[F[j]-1], & \text{otherwise} \end{cases}$$

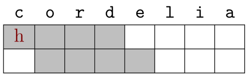
BOYER-MOORE

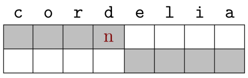
REVERSE SEARCHING

Idea of reverse searching:

① compare from right to left

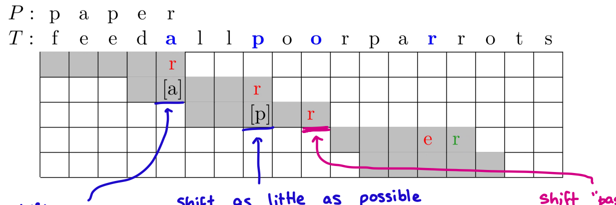
② if we have a mismatch, shift so the new guess fits the character of T.





BAD CHARACTER HEURISTIC

We use the "bad character heuristic":



shift so [a] in paper is here.

shift as little as possible (so we shift to the "rightmost" p)

shift "past" the character if char not in P.

- we shift the guess so that T[i] matches the last occurrence in P (ie $P[\underbrace{L[T[i]]}_{\ell}]$.)

L = last occurrence array.

LAST OCCURRENCE ARRAY

We need to precompute the "last occurrence array":

char c	p	a	e	r	all others
L[c]	2	1	3	4	−1

character

how much to shift by?

if no match:
- shift "past" the character.

If we were comparing T[i] with P[j], and ℓ = L[T[i]], then we want to shift j−ℓ units.

SIMPLIFIED BAYER-MOORE

Pseudocode:

```
Algorithm 9.6: BoyerMoore::patternMatching(T, P)
L ← last occurrence array computed from P          - preprocessing
~~S ← good suffix array computed from P~~
$i \leftarrow m-1$                                  // currently inspected character of T
$j \leftarrow m-1$                                  // currently inspected character of P
while $i < n$ and $j \geq 0$ do
    // invariant: The current guess begins at $T[i-j]$
    if $T[i] = P[j]$ then                           // match, go one character leftward
        $i \leftarrow i-1$
        $j \leftarrow j-1$
    else                                            // mismatch, find next guess to try
        $i \leftarrow m-1+i-\min\{L[T[i]], j-1, \cancel{S[j]}\}$   // shifting forward based on L
        $j \leftarrow m-1$
if $j = -1$ then
    return "found at guess $i+1$"
else
    return FAIL
```

```
Algorithm 9.7: BoyerMoore::lastOccurrenceArray(P)
L ← array indexed by alphabet Σ
Initialize L to be −1 everywhere
for $i \leftarrow 0 \dots m-1$ do $L[P[i]] \leftarrow i$
return L
```

If we only do

① reverse-order searching; &

② bad-character jumps,

this is called the "simplified Bayer-Moore algorithm."

This works well in practice

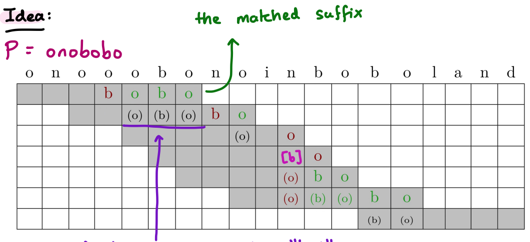
(skips ~ 25% of T in expeiments)

Worst case run-time = O(mn).

GOOD SUFFIX ARRAY

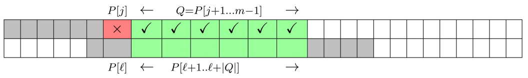
Idea:

P = onobobo



we shift here to get the "last" occurrence of the suffix

① we use the same strategy as KMP: use what we matched to eliminate some guesses.



② if there is no matched string, move forward by 1 (the last-occurrence array might help).

③ if the matched string is not again in the pattern, try matching a suffix of the matched string to a prefix of P.

④ If only the empty suffix fits, shift past the word.

⑤ if Q = "", use the last occurrence.

Pseudocode:

```
Algorithm 9.6: BoyerMoore::patternMatching(T, P)
L ← last occurrence array computed from P
S ← good suffix array computed from P
$i \leftarrow m-1$                                  // currently inspected character of T
$j \leftarrow m-1$                                  // currently inspected character of P
while $i < n$ and $j \geq 0$ do
    // invariant: The current guess begins at $T[i-j]$
    if $T[i] = P[j]$ then                           // match, go one character leftward
        $i \leftarrow i-1$
        $j \leftarrow j-1$
    else                                            // mismatch, find next guess to try
        $i \leftarrow m-1+i-\min\{L[T[i]], j-1, S[j]\}$   // shift according to L & S
        $j \leftarrow m-1$
if $j = -1$ then
    return "found at guess $i+1$"
else
    return FAIL
```

WILDCARDS: P*

We have 3 main cases, depending on how much of Q exists again in P.

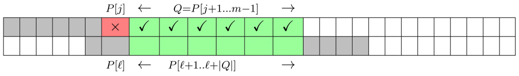
To unify them, we add "wildcards" to P.

$$P^* = \underbrace{** \cdots **}_{m} P[0 \cdots m-1]$$

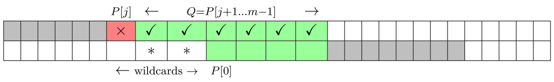
where a wildcard matches any character.

Then, the 3 cases become

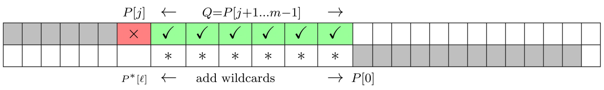
① Q is a substring of P.



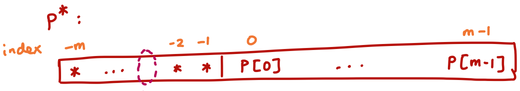
② A suffix of P is a prefix of Q.



③ No character of Q is matched.



Then, Q is a substring of P* and take the rightmost occurrence except not at the very end.



let the "beginning" position to match be ℓ+1.

Then Q is a prefix of $P^*[\ell+1 \dots m-1]$;

ie $P[j+1 \dots m-1]$ is a prefix of $P^*[\ell+1 \dots m-1]$.

COMPUTING THE GOOD SUFFIX ARRAY AS A HUMAN

	Q	$P^*[-m..m-2]$													match starts at index	
		-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5		
j	$P[j+1..m-1]$	*	*	*	*	*	*	*	b	o	o	b	o	b		$S[j]$
5	o												o		4	3
4	bo											b	o		3	2
3	obo										o	b	o		2	1
2	bobo						b	o	b	o					-2	-3
1	obobo					o	b	o	b	o					-3	-4
0	oobobo				o	o	b	o	b	o					-4	-5

- we use

$$\boxed{S[j] = \max_{\ell} P[j+1 \ldots m-1] \text{ is a prefix of } P^*[\ell+1 \ldots m-2].}$$

FINDING S[j] IN LINEAR TIME

1. We note

$$S[j] = \max_{\ell} P[j+1 \ldots m-1] \text{ is a prefix of } P^*[\ell+1 \ldots m-2]$$

$$= \max_{\ell} \underbrace{P[m-1 \ldots j+1]}_{\substack{\text{"}P[j+1 \ldots m-1]^{rev}\text{";} \\ \text{the "reverse" string}}} \text{ is a suffix of } \underbrace{P^*[m-2 \ldots \ell+1]}_{\text{"}P^*[\ell+1 \ldots m-2]^{rev}\text{"}}$$

$$= \max_{\ell} \left\{ \begin{array}{l} \text{when parsing } P^*[m-2 \ldots \ell+1] \text{ at the KMP-automaton} \\ \text{for } P[m-1 \ldots j+1]\text{, then we reach state } j+1. \end{array} \right\}$$

2. This evaluates to

$$\boxed{S[m-q-1] = m-2-\min_{k} \left\{ \begin{array}{l} \text{parsing } R[1 \ldots k] \text{ on the KMP-automaton for} \\ R = P^{rev}{**\ldots**} \text{ brings us to state } q \end{array} \right\}.}$$

3. We can do this by parsing in O(m) time, to compute S.

SUMMARY

1. With the good suffix-heuristic, we can skip even more.
2. But, we can skip even more using $S^+[\,]$ vs. $S[\,]$.
3. This algorithm can be made to run in $O(m+n)$ time.

SUFFIX TREE

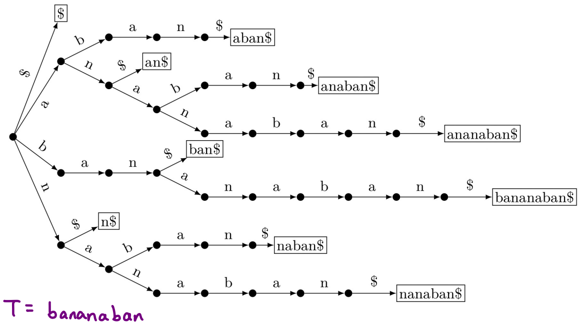
1. **Idea:** Is P a substring of text T

<=> is P a prefix of a suffix of T?

2. So, we can store all suffixes of T, and then do a prefix-search for P.

SUFFIXES IN A TRIE

1. We can store the suffixes in a trie:



2. To see if P is in T:

① Traverse the trie using P.

② If we are still "in" the trie after ①, then P is a substring of T.

3. **Run-time:** $O(|P|) = O(m)$.

4. But the price to pay:

- T has length n
- so we have n+1 suffixes of length 0, 1, …, n
- thus, the trie can have $\Omega(n^2)$ nodes.

SUFFIX TREE

1. We can instead store the suffixes in a compressed trie:

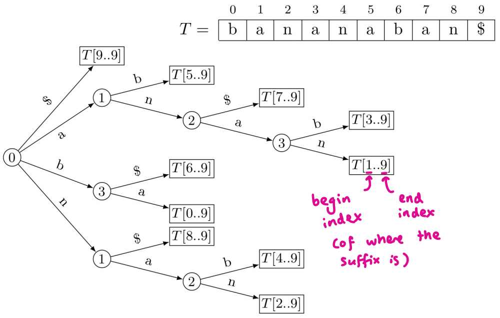


Figure 9.16: A suffix tree

2. Prefix-search for P takes $O(|P|) = O(m)$ time

(details omitted)

3. We can build the trie in $O(n)$ time.

(no details — too complex!)

4. **Space:** $O(n)$.

SUFFIX ARRAY

1. We can also store the suffixes in a "suffix array": the sorting permutations of the suffixes.

i	
0	bananaban$
1	ananaban$
2	nanaban$
3	anaban$
4	naban$
5	aban$
6	ban$
7	an$
8	n$
9	$

Sort lexicographically

start index of suffix

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$

the sorting permutation (ie the "suffix array").

2. **Space:** $O(n)$ (and a very small constant).

3. **Building:**

① Sort (using MSD-radix-sort)

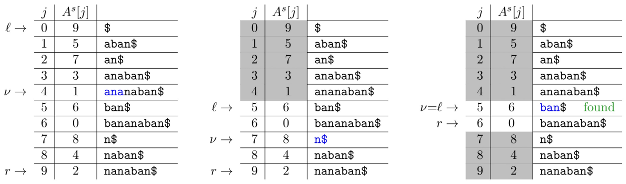
- run-time = $O(n^2)$ worst case, but usually faster.

② A special algorithm

- run-time = $O(n\log n)$
- covered in CS 482

4. **Pattern-matching:**

- use binary search.



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

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

	j	$A^s[j]$	
	0	9	$
	1	5	aban$
	2	7	an$
	3	3	anaban$
	4	1	ananaban$
$\nu=\ell \rightarrow$	5	6	ban$ found
$r \rightarrow$	6	0	bananaban$
	7	8	n$
	8	4	naban$
	9	2	nanaban$

Figure 9.18: Search for ban using a suffix array

5. **Run-time:** $O(\log n)$ comparisons.

- comparisons = compare P to a suffix: $\Theta(m)$

So $\boxed{\text{run-time} = \Theta(m\log n)}$.

(slightly slower than suffix-trees.)

Chapter 10:
Text Compression

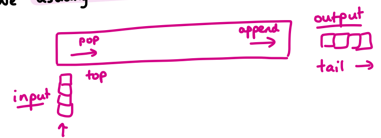
1. Idea:
   ① Input: source texts S (huge!)
   ② Output: map text S into a new text C (smaller).
2. Note S lives on alphabet Σ_S, and C lives on alphabet Σ_C.
3. Objectives:
   ① Minimize the "compression ratio"; where

$$\boxed{\text{comp. ratio} = \frac{|C|}{|S|}\,\frac{\log|\Sigma_C|}{\lceil \log|\Sigma_S| \rceil}}$$

   ② Fast encoding & decoding.
4. All compressions we look at are lossless; we can get S back from C uniquely.

STREAMS

1. We usually store S and C as "streams."



2. Input stream: Read one char at a time via top/pop.
   - also supports 'isEmpty' and 'reset'.
3. Output stream: Write one character at a time (via append)
4. This is convenient for handling large texts (we start processing while loading)

CHAR-BY-CHAR ENCODING

1. Idea: assign to each $c \in \Sigma_S$ a codeword E(c).

Char	A	B	C	D	...
E(c)	B	C	D	E	...

(Caesar shift)

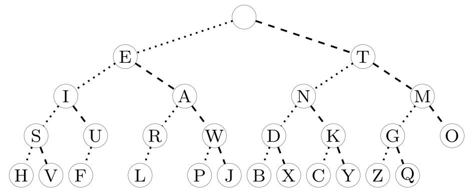
E(c)	10000010	1000001		...

(ASCII)

2. The above are "fixed-length encodings": all codewords have the same length.
3. Better: "variable-length encoding" — more frequent characters get shorter codewords.

VARIABLE-LENGTH CODES

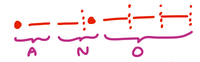
1. Overall goal: find a short encoding.
2. Idea: Some letters in Σ occur more often than others, so use shorter codes for more frequent characters.
3. Example: encoding the alphabet in Morse Code.
   we build an "encoding trie":



→ more frequent letters (eg E & T) go "higher" on the trie than less frequent ones (eg F)

MORSE CODE

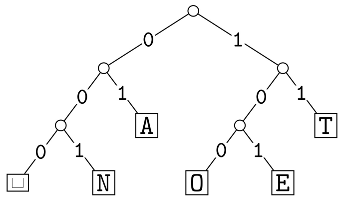
1. Using the encoding trie in the previous section, we can convert the alphabet into Morse Code
   (eg if "left" = • & "right" = —)
2. Problem: Morse Code is not lossless!
   eg WATT



(unless we use an "end of char" pause)

PREFIX-FREE ENCODING/DECODING

1. To achieve lossless encoding, we use an encoding trie where the codewords are prefix-free.
   (ie no codeword is a prefix of another codeword)
2. In particular, we make it so the encoded trie has codewords only at the leaves.
3. Run-time to decode C: $O(|C|)$
   Run-time to decode S: $O(\sum_{c \in S} |E(c)|) = |C|$



ENCODING ALGORITHM

1. Imagine we have some character encoding $E : \Sigma_S \to \Sigma_C^*$.
2. Note: E is a dictionary with keys in Σ_S.

```
Algorithm 10.1: charByChar::encoding(S, C)
Input   : input-stream S, output-stream C
// We assume the class stores an encoding dictionary E
while S is not empty do
    c ← S.pop()
    x ← E.search(c)
    C.append(x)
```

DECODING OF PREFIX-FREE CODES

1. Pseudocode:

```
Algorithm 10.2: prefixFree::decoding(C, S)
Input   : input-stream C, output-stream S
// We assume the class stores a prefix-free encoding trie T
while C is not empty do
    r ← T.root
    while r is not a leaf do
        if C is empty or r has no child labeled with C.top() then
            return "invalid encoding"
        else
            r ← child of r that is labeled with C.pop()
    S.append(character stored at r)
```

2. Run-time: $O(|C|)$.

ENCODING FROM THE TRIE

1. We can also encode directly from the trie:

```
Algorithm 10.3: prefixFree::encoding(S, C)
// We assume the class stores a prefix-free encoding trie T
Initialize encoding-array E indexed by characters of Σ_S
forall leaves ℓ in T do  E[character at ℓ] ← ℓ        // setup
while S is non-empty do
    v ← E[S.pop()]                  // Find code-word w by going up from leaf v.
    w ← empty string
    while v is not the root do
        w.prepend(character on link from v to its parent)  ]
        v ← parent of v                                     ] run-time: O(|w|)
    C.append(w)
```

2. Run-time: $O(|T| + |C|)$.
   (This is in $O(|\Sigma_S| + |C|)$ if T has no node with 1 child.)

HUFFMAN-ENCODING

Idea: frequent chars should have short codewords.

In particular,

$$|C| = \sum_{c \in S} |E(c)|$$

$$= \sum_{c \in \Sigma_S} (\text{freq of } c \text{ in } S) \cdot |E(c)|$$

$$= \sum_{c \in \Sigma_S} f(c) \cdot \underbrace{(\text{depth of } c \text{ in the } T)}_{d_T(c)}$$

$$= \sum_{c \in \Sigma_S} f(c) \cdot d_T(c)$$

We define

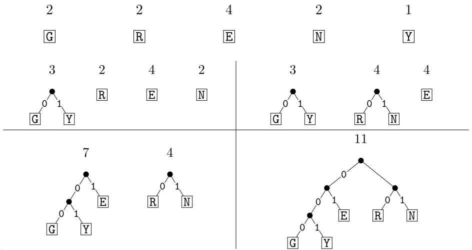
$$\boxed{\text{cost}(T) = f(c)\, d_T(c).}$$

Assumption: $|\Sigma_S| \geqslant 2$.

Idea:

eg GREENENERGY, $\Sigma_S = \{G, R, E, N, Y\}$

char freq: G:2, R:2, E:4, N:2, Y:1



① Set up lots of 1-char tries;

② Find the 2 least frequent chars a, a' and make them siblings in the trie

eg

③ Repeat ②.

PSEUDOCODE

```
Algorithm 10.5: Huffman::encoding(S, C)
Input  : Input-stream S where S contains ≥ 2 distinct characters, output-stream C
f ← array indexed by $\Sigma_S$, initially all-0                  // compute frequencies
while S is non-empty do increase f[S.pop()] by 1

T ← Huffman::buildTrie($\Sigma_S$, f)                               // find encoding trie
Store T in the class
C.append(trie T encoded suitably)

reset input-stream S
prefixFree::encoding(S, C)                                      // do actual encoding
```

RUN-TIME OF HUFFMAN-ENCODING

Priority queue has $|\Sigma_S|$ items initially;

We do delete in $|\Sigma_S|$ time.

- so, building T takes time $O(|\Sigma_S| \log |\Sigma_S|)$.

Rest of the encoding takes time $O(|S| + |C|)$.

But,

① We need to send the trie along. &

② We need to go through S twice.

→ but, cost(Huffman trie) is small.

T_H HAS MIN COST AMONG ALL PREFIX-FREE BINARY ENCODING TRIES

The Huffman-trie T_H has minimum cost among all prefix-free binary encoding tries.

Proof. We will show for any prefix-free binary encoding trie T_0, we have

$$\text{cost}(T_H) \leq \text{cost}(T_0).$$

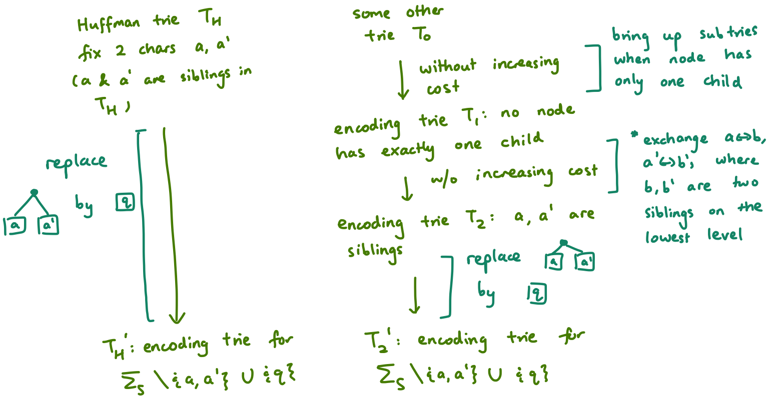
We do this by induction on $|\Sigma_S|$.

Base case: $|\Sigma_S| = 2$.

T_H:

T_0 can't be better.

Step:



By induction,

$$\text{cost}(T_H') \leq \text{cost}(T_2')$$

$$\Rightarrow \text{cost}(T_H') + f(a) + f(a') \leq \text{cost}(T_2') + f(a) + f(a')$$

$$\Rightarrow \text{cost}(T_H) \leq \text{cost}(T_2) \quad (\leq \text{cost}(T_0)).$$

Proof follows. ☑

This is not necessarily true for other bases.

This also does not always make shorter.

NO LOSSLESS COMPRESSION ALGORITHM HAS CMP RATIO <1 FOR ALL INPUT STRINGS

No lossless compression algorithm can have compression ratio <1 for all input strings.

Proof. Assume $\Sigma_S = \Sigma_C = \{0,1\}$. Assume we had such an algo.

Consider all the bitstrings of length n. There are 2^n such strings.

The algorithm maps these to the bitstrings of length $\leq n-1$, of which there are

$$1 + 2^1 + \cdots + 2^{n-1} = 2^n - 1 < 2^n.$$

This set is smaller; so, by Pigeonhole Principle, ∃ bitstrings x_1, x_2 of length n that both gets encoded as bitstring w.

But this contradicts the losslessness of the algo!

Proof follows. ☑

MULTI-CHARACTER ENCODING

Idea: Encode multiple characters (a substring of S) using one code word.

RUN-LENGTH ENCODING

1. We assume $\Sigma_S = \{0,1\}$ (the inputs are bitstrings).

2. A "run" is a maximal substring that uses only one character.

   eg $S = \underbrace{11111}_{5}\underbrace{000}_{3}\underbrace{1111}_{4}$

3. **Idea:**
   ① We write down the lengths of the runs in S; & then
   ② we encode the string by these lengths.
   ③ We also need to specify the first bit we started with.

   eg $S = \underbrace{1}_{\text{first bit}}\ \underbrace{5\ \ 3\ \ 4}_{\text{the run-lengths}}$

4. However, we want Σ_C to be finite, and preferably $\{0,1\}$.

5. So, we need to map integers to bitstrings, so we do not need separations.

ELIAS GAMMA CODING

1. **Idea:** to encode k:
   ① Write $\lfloor \log k \rfloor$ copies of 0; then
   ② The binary representation of k (always starts with 1). (has length $\lfloor \log k \rfloor + 1$.)

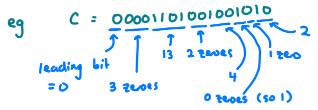
k	$\lfloor \log k \rfloor$	k in binary	encoding $E(k)$
1	0	1	1
2	1	10	010
3	1	11	011
4	2	100	00100
5	2	101	00101
6	2	110	00110
⋮	⋮	⋮	⋮

2. Pseudocode:

```
Algorithm 10.6: RLE::encoding(S, C)
Input : Non-empty input-stream S of bits, output-stream C
b ← S.top()                                   // bit-value for the current run
C.append(b)
while S is not empty do
    k ← 1                                     // length of run
    while S is not empty and S.pop() = b do k++
    K ← empty string                          // binary encoding of k
    while k > 1 do
        C.append(0)
        K.prepend(k mod 2)
        k ← ⌊k/2⌋
    K.prepend(1)
    C.append(K)
    b ← 1 − b                                 // flip bit for next run
```

DECODING

1. **Idea:**
   ① Extract the leading bit;
   ② Compute the length of the 0-run, ℓ, and extract ℓ+1 next bits.
   ③ Convert this into a number, and write this number of bits.
   ④ Repeat steps ②-③ until the string is empty.



⇒ $S = \underbrace{0000000000000}_{13}\underbrace{1111}_{4}\underbrace{0}_{1}\underbrace{11}_{2}$

2. Pseudocode:

```
Algorithm 10.7: RLE::decoding(C, S)
Input : Non-empty input-stream C of bits, output-stream S
// pre:  C is a valid run-length encoding
b ← C.pop()                                   // bit-value for the current run
while C is not empty do
    ℓ ← 0                                     // length of binary encoding, minus 1
    while C.pop() = 0 do ℓ++
    // The last pop() removed the leading bit of the binary encoding
    k ← 1
    for j ← 1 to ℓ do                         // get binary encoding and convert
        k ← k ∗ 2 + C.pop()
    for j ← 1 to k do S.append(b)             // append the run
    b ← 1 − b                                 // flip bit for next run
```

3. **Run-time:** $O(|S| + |C|)$.

4. We also note
   ① This works well for long runs.

   eg $0^n \longrightarrow \underbrace{\lfloor \log n \rfloor + 1}_{\text{binary representation}} + \underbrace{\lfloor \log n \rfloor}_{\text{the leading 0's}} + \underbrace{1}_{\text{leading bit}} = 2\lfloor \log n \rfloor + 2.$

   ② But this works really badly for short runs.

   eg A run of length 2.

   $\ldots\ldots 11 \ldots\ldots \longrightarrow \ldots\ldots 010 \ldots\ldots$

   ③ It is useful for transmitting black & white pictures (especially text).

   Why? → long runs of white/black pixels.

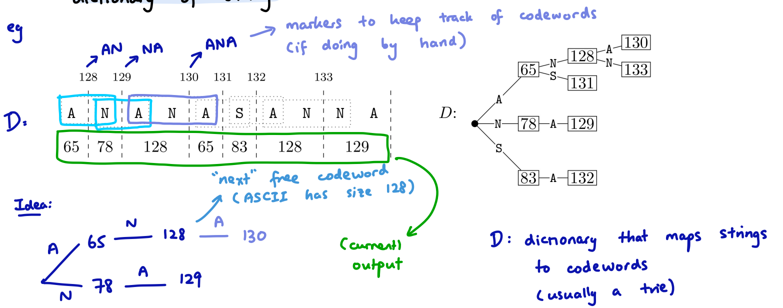
LEMPEL-ZIV-WELCH ENCODING

1. This is used in compress & GIF.

2. **Idea:** We instead automatically detect longer substrings that got codewords.

3. To do this, we repeatedly
   ① Find the longest substring, in the part we want to encode, for which we have a codeword.
   ② Add this string + next char to our dictionary of strings with codewords.

eg



D: dictionary that maps strings to codewords (usually a trie)

We keep adding to the trie in this way until we run out of input.

4. We assume
   ① the text is in ASCII; &
   ② D initially stores ASCII.

5. For now, the output is a list of integers.
   We need to convert this into bitstrings.
   ① We could use Elias-Gamma codes
      - but this gets too long.
   ② Instead, we use "12-bit encoding" of the numbers.
      - but we stop adding codewords after code 4095.

$$\underbrace{000001000001}_{65}\underbrace{000001001110}_{78}\underbrace{000010000000}_{128}\underbrace{000001000001}_{65}\underbrace{000001010011}_{83}\underbrace{000010000000}_{128}\underbrace{000010000001}_{129}$$

6. Pseudocode:

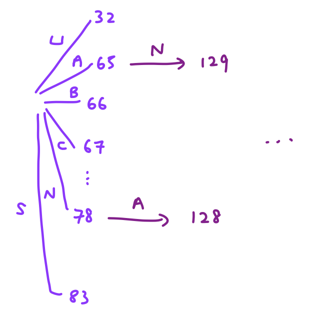
```
Algorithm 10.8: LZW::encoding(S,C)
Input   : Input-stream S of ASCII-characters, output-stream C
Initialize dictionary D as a trie that maps ASCII to {0, ..., 127}
idx ← 128                          // global counter for first free code-number
while S is not empty do
    v ← root of trie D
    while S is non-empty and v has a child c labeled with S.top() do
        v ← c
        S.pop()
    C.append(code-number stored at v)
    if S is non-empty then
        create child of v labeled S.top() with code-number idx
        idx++
```

7. Encoding-time: O(|S|)

DECODING

1. Idea:
   ① Initialize D with ASCII;
   ② Read the next code number;
   ③ Look up corresponding string in D;
   ④ Expand D as the encoder would have done;
   (but we are one step behind)
   ⑤ Repeat ②-④.

eg 65 78 128 65 83 128 129



input	decodes to	Code #	String
67	C		
65	A	128	CA
78	N	129	AN
32	␣	130	N␣
66	B	131	␣B
129	AN	132	BA
133	???	133	

2. Note that at the end, we could get a code number that was about to be added.

3. If we want to use the decoding of the codeword we are about to add:
   We can show the string is then
   the previous string + 1st char of the prev. string.

4. Pseudocode:

```
Algorithm 10.9: LZW::decoding(C,S)
Input   : Input-stream C of integers, output-stream S
Initialize D as a dictionary that maps {0, ..., 127} to ASCII
idx ← 128                          // global counter for first free code-number
code ← C.pop()                     // first number creates no entry in D
s ← LZW::dictionaryLookup(D, code)
S.append(s)
while C is not empty do
    s_prev ← s                      previous string
    code ← C.pop()
    if code < idx then s ← LZW::dictionaryLookup(D, code)
    else if code = idx then s ← s_prev ++ s_prev[0]          // special situation
    else return "invalid encoding"     ↑ concatenation
    S.append(s)
    insert s_prev + s[0] into D with code-word idx
    idx++
```

5. Decoding time:
   - each round of the while takes time proportional to the # of chars written to the output

   Thus the run-time is O(|S|).

6. Notes:
   ① This compresses well in practice; but
   ② This is very bad if no repeated substrings.

bzip2

1 **Idea:** We transform the text into something that is not necessarily shorter, but has other desirable qualities.

Transformation name	Properties	Alphabet
Text T_0: `barbarabarbarbaren$`		ASCII
↓ Burrows-Wheeler transform	If T_0 has repeated longer substrings, then T_1 has long runs of characters.	
Text T_1: `nrbbbbbr$arrreaaaaa`		ASCII
↓ Move-to-front transform	If T_1 has long runs of characters, then T_2 has long runs of zeros.	
Text T_2: 110,114,100,0,0,0,0,1,6,100,2,0,0,103,2,0,0,0,0		{0,...,127}
↓ Modified RLE	If T_2 has long runs of zeroes, then T_3 has chars A' and B' very frequently	
Text T_3: 110,114,100,A',B',1,6,100,2,B',103,2,A',B'		{1,...,127}∪{A', B'}
↓ Huffman encoding	Compresses well since input-chars are unevenly distributed	
Text T_4: 00011101111100010000000111110101110010110001		{0,1}

MODIFIED RLE

1 **Idea:** Encode only runs of 0, using binary bijection numeration.

($\Sigma = \{0, \ldots, 127\}$)

eg $S = 110, 114, 100, \underbrace{0,0,0}_{3}, 1, 6, 100, 2, \underbrace{0,0}_{2}, 103$

$\Rightarrow C = 110, 114, 100, A', B', 1, 6, 100, 2, B', 103$

k	0	1	2	3	4	...
E(k)	Λ	A'	B'	A',A'	A'.B'	...

← maps runs of 0's of this length

← to this substring.

2 Output alphabet: $\Sigma = \{1, \ldots, 127\} \cup \{A', B'\}$.

MOVE-TO-FRONT TRANSFORM

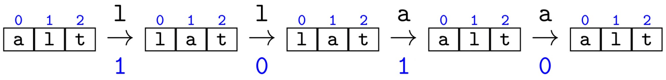
1 **Idea:**

① Initialize D: array of size $|\Sigma_S|$ that stores Σ_S (typically ASCII)

② Get char c from the input.

③ Write $D^{-1}(c)$ to the output.
(using brute force).

④ Update D by bringing c to the "front" of D
(the MTF-heuristic).



2 If we have k-consecutive same characters, we get a run of k-1 0's in the output.

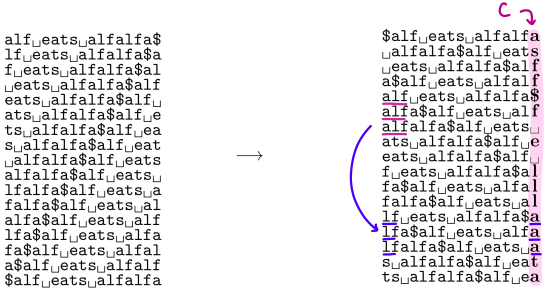
3 We should have lots of 0, 1, 2 ... , & very few of 125, 126, ...

4 Decoding: Same except D^{-1} becomes D.

BURROWS-WHEELER TRANSFORM

1 **Idea:**

eg S = alf␣eats␣alfalfa$



① Write all cyclic shifts; ie $S[i+1 \ldots n-1] +\!\!+ S[0 \ldots i]$ $\forall i$.

② Sort them lexicographically.
($ < ␣ < a < e < ...)

③ C = rightmost column of the result "matrix".

2 Observe that if S has repeated substrings, then C likely has long runs of chars.

eg in our example, "alf" shows up 3 times.
⇒ there exists 3 cyclic shifts that start with "alf".
⇒ there exist 3 cyclic shifts that start with "lf" and end at "a".
Likely, these shifts will end up consecutive.
If they are consecutive, this implies the 3 'a's are consecutive in C.
For the same reasoning, we also get 3 'l's that are likely to be consecutive in C.

FAST BURROW-WHEELER TRANSFORM

1 **Idea:**

start-index of cyclic shift	
0	alf␣eats␣alfalfa$
1	lf␣eats␣alfalfa$a
2	f␣eats␣alfalfa$al
3	␣eats␣alfalfa$alf
4	eats␣alfalfa$alf␣
5	ats␣alfalfa$alf␣e
6	ts␣alfalfa$alf␣ea
7	s␣alfalfa$alf␣eat
8	␣alfalfa$alf␣eats
9	alfalfa$alf␣eats␣
10	lfalfa$alf␣eats␣a
11	falfa$alf␣eats␣al
12	alfa$alf␣eats␣alf
13	lfa$alf␣eats␣alfa
14	fa$alf␣eats␣alfal
15	a$alf␣eats␣alfalf
16	$alf␣eats␣alfalfa

→

A_s 〃 $\pi(k)$ — the corresponding suffixes in the suffix array.

k	$\pi(k)$	
0	16	$alf␣eats␣alfalfa
1	8	␣alfalfa$alf␣eats
2	3	␣eats␣alfalfa$alf
3	15	a$alf␣eats␣alfalf
4	0	alf␣eats␣alfalfa$
5	12	alfa$alf␣eats␣alf
6	9	alfalfa$alf␣eats␣
7	5	ats␣alfalfa$alf␣e
8	4	eats␣alfalfa$alf␣
9	2	f␣eats␣alfalfa$al
10	14	fa$alf␣eats␣alfal
11	11	falfa$alf␣eats␣al
12	1	lf␣eats␣alfalfa$a
13	13	lfa$alf␣eats␣alfa
14	10	lfalfa$alf␣eats␣a
15	7	s␣alfalfa$alf␣eat
16	6	ts␣alfalfa$alf␣ea

① We need the sorting permutation of the cyclic shifts.

② This is the same as the sorting permutation of the suffixes.

③ So, to compute the encoding, we
- compute the suffix array ($O(n \log n)$)
- output $C[i] = S[(A_s[i] - 1) \bmod n]$.

DRAWBACKS OF BWT

1 Run-time: $O(n \log n)$ (maybe $O(n)$)

2 Space: $O(n)$ (with suffix array S).

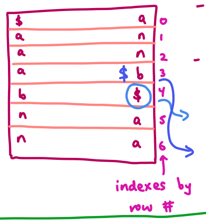
3 Here, we need the entire text at once.

DECODING IN BWT

Example 1:

eg $C =$ annb\$aa

Reconstruct the matrix of cyclic shifts:



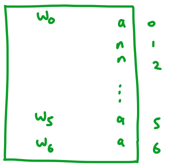
- rightmost column is just C.
- leftmost column is the same as C, but sorted.

First character of S is b.

Second character of S is a.

The char after b is the a in row 6.

Proof. Define w_i = the word in front of char x in row i.



We note $w_0 \underset{lex}{<} w_5 \underset{lex}{<} w_6$

(no equality ∵ end of char symbol is in different positions).

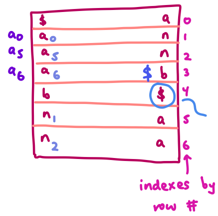
We have 3 cyclic shifts

The a's do not change the lexicographic order, so

$a_0 w_0 \underset{lex}{<} a_5 w_5 \underset{lex}{<} a_6 w_6$.

Restating:

The characters in the first column are in the same relative order as they were in the last column.



S: $b\,a_6\,n_2\,a_5\,n_1\,a_0\,\$$

Example 2:

eg $C =$ ard\$rcaaaabb

$\$_3$	a_0
a_0	r_1
a_6	d_2
a_7	$\$_3$
a_8	r_4
a_9	c_5
b_{10}	a_6
b_{11}	a_7
c_5	a_8
d_2	a_9
r_1	b_{10}
r_4	b_{11}

Final string:

abacadabra\$

(see course notes for more details.)

$\$_3 \to a_0 \to r_1 \to b_{10} \to a_6 \to \cdots$

(append backwards.)

Pseudocode:

Algorithm 10.14: *BWT::decoding*(C, S)

```
Input   : Coded text $C$ as array (not stream), output-stream $S$
$A \leftarrow$ array of size $n \leftarrow C.size$
for $i = 0$ to $n-1$ do  $A[i] \leftarrow (C[i], i)$
Stably sort $A$ by first aspect    // eg radix sort, merge sort
for $j = 0$ to $n-1$ do if $C[j] = \$$ then break                 // find $-char
repeat                                                            // actual decoding
    $S$.append(first entry of $A[j]$)
    $j \leftarrow$ second entry of $A[j]$
until last appended character was $
```

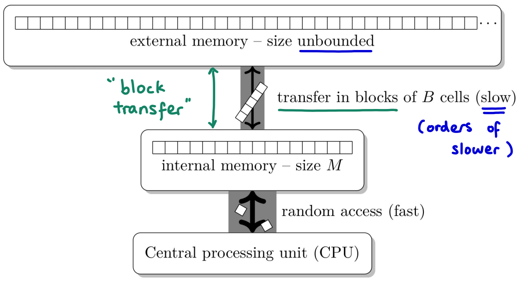
This is simple, & runs in O(n) time.

Chapter 11: External Memory

1. So far, our computer model assumed "all memory cells are equal" and "we have infinitely many of them".
2. Now, we still assume we have infinite memory, but distinguish how easy it is access.

THE EXTERNAL-MEMORY MODEL / EMM

1. Idea:



"block transfer"

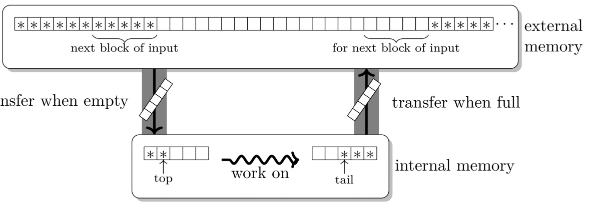
(orders of magnitude slower)

*we only model one gap.

2. In particular,
   ① B is the "block size"; the amount transferred at once (~ few MB)
   ② M is the size of the internal memory. (can load $\leq \frac{M}{B}$ blocks at once) (~ few GB)
   ③ n is the size of the input. (~ few TB/PB)

STREAMS

1. Idea:



2. Total # of block transfers: $\Theta(\frac{n}{B})$ (this is optimal).

GOOD/BAD ALGOS FOR EM

1. Good algorithms:
   ① Huffman
   ② LZW
   ③ KMP
   ④ Boyer-Moore

   } use streams

2. Bad algorithms:
   ① BWT
   ② Suffix array & tree

   } require whole input

MERGE

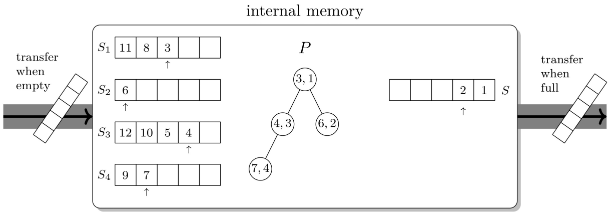
Merge takes $\Theta(\frac{n}{B})$ block transfers.

MERGESORT

1. Mergesort uses $\lceil \log_2(n) \rceil$ rounds of "splitting" & "merging".
2. So, we need $\Theta(\frac{n}{B} \cdot \log n)$ block transfers.

d-WAY MERGE

1. Idea:



- Choose s.t. the blocks fit into internal memory $\boxed{(d \approx \frac{M}{B})}$.

2. Pseudocode:

```
Algorithm 11.2: d-way merge(S_1, ..., S_d, S)
Input : Input-streams $S_1, \ldots, S_d$ that are in sorted order. Output-stream $S$.
$P \leftarrow$ empty priority queue that stores pairs and is min-oriented w.r.t. first aspect
for $i \leftarrow 1$ to $d$ do $P.insert(\langle S_i.top(), i \rangle)$     // Populate priority queue
while $P$ is not empty do                                                 // repeatedly extract minimum
    $\langle x, i \rangle \leftarrow P.deleteMin()$
    // Item $x$ is top-item of stream $S_i$
    $S.append(S_i.pop())$
    if $S_i$ is not empty then $P.insert(\langle S_i.top(), i \rangle)$
```

3. If we split & merge d ways, we only need $\lceil \log_d n \rceil$ rounds.
4. So, # of block transfers = $\Theta(\log_d n \cdot \frac{n}{B})$.
5. But we can improve further (details omitted)

   ($\Theta(\log_{\frac{n}{B}}(\frac{n}{M}) \cdot \frac{n}{B})$ → "optimal" bound for EMM.

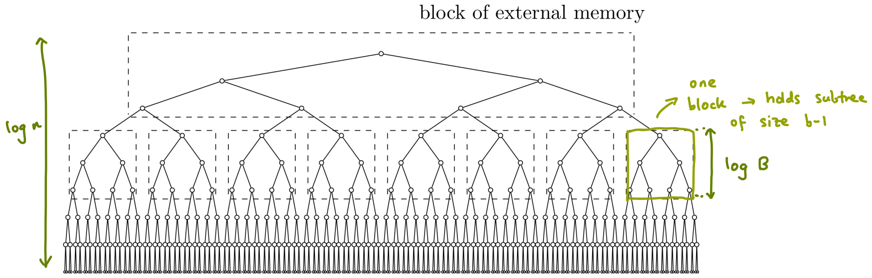
COMPARISON-BASED DICTIONARIES

BSTs

1. We know search/insert/delete can be done with $O(\log n)$ block transfers (naively).
2. We can show we need at least $\Omega(\log_B n)$ block transfers for search.
3. Can we achieve $O(\log_B n)$ block transfers?

IDEAL STRUCTURE FOR SEARCH

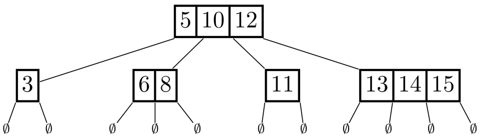
1. Idea:



2. Search path hits $\frac{\Theta(\log n)}{\log b} \Rightarrow \Theta(\log_b n)$ block transfers, where b be $\Theta(B)$.
3. We can also view each block as one node in a multiway tree
   (b-1 KVPs, b subtrees)

a-b-TREE

1. Idea:



eg 2-4 tree

① $a, b \in \mathbb{Z}^+$, $2 \leq a \leq \lfloor \frac{b+1}{2} \rfloor$.
   (usually we set $a = \lfloor \frac{b+1}{2} \rfloor$.)

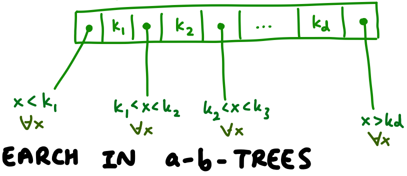
② Every node has $a \leq$ # subtrees $\leq b$, except at the root, which has $2 \leq$ # subtrees $\leq b$.

③ All empty subtrees are on the same layer.

④ A node with d KVPs has exactly $d+1$ subtrees.

2. Order-property:

   Close up on a node.



SEARCH IN a-b-TREES

1. Similar to BSTs:

   ① We load the root;

   ② Find the best place for the search key in the node;

   ③ If k is not in the node, repeat in the appropriate subtree.

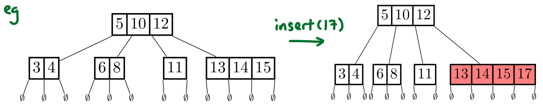
   ④ Otherwise, return k & the corresponding value.

2. Run-time (in RAM model): $O(\text{height} \cdot \log b)$.

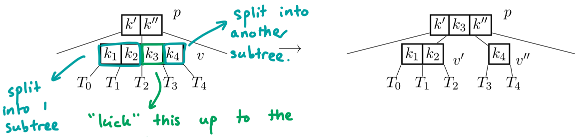
INSERT IN a-b-TREES

1. Idea:
   - Call search, and add key and an empty subtree at the leaf.
2. However, if we do this, we might get that a node is too big.

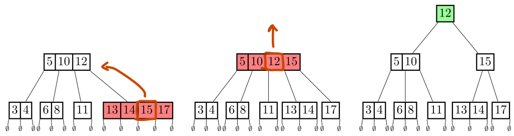
eg



3. To solve this, we use "node splitting":



4. We have to repeat node-splitting until all nodes are "happy".



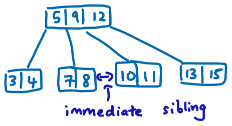
5. Pseudocode:

```
Algorithm 11.4: 24Tree::insert(k)
v ← 24Tree::search(k)                                        // leaf where k should be
Add k and an empty subtree at the appropriate place in the key-subtree-list of v
while v has 4 keys do                                        // overflow leads to node split
    Let ⟨T_0, k_1, ..., k_4, T_4⟩ be key-subtree list at v
    if v has no parent then                                  // tree grows upwards
        create a parent of v without keys
    p ← parent of v
    v' ← new node with keys k_1, k_2 and subtrees T_0, T_1, T_2
    v'' ← new node with key k_4 and subtrees T_3, T_4
    Replace ⟨v⟩ by ⟨v', k_3, v''⟩ in key-subtree-list of p
    v ← p
```

6. Note:
   - overflow means b keys & $b+1$ subtrees
   - after the node split, new nodes have $\geq \lfloor \frac{b-1}{2} \rfloor$ keys
   - since we required $a \leq \lfloor \frac{b+1}{2} \rfloor$, this is $\geq a-1$ keys as required.
7. Run-time: search ($O(\text{height} \cdot \log b)$)
   + update ($O(\text{height} \cdot \log b)$)

   ∴ run-time = $O(\text{height} \cdot \log b)$.

IMMEDIATE SIBLING



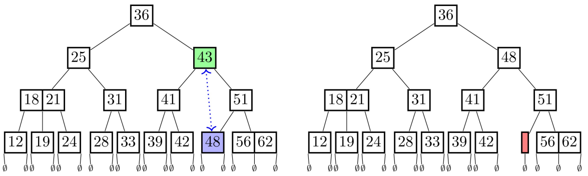
DELETE IN a-b-TREES

1 **Idea:**

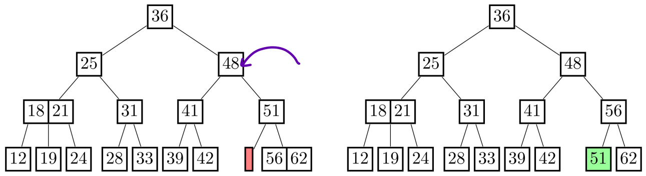
① Search for k, then trade with successor if KVP is not at a leaf.

② Remove the key & one empty subtree from leaf

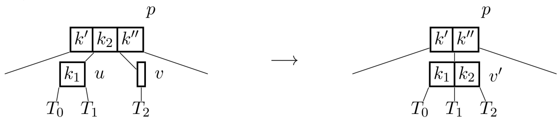
③ But this could lead to underflow: ie < a subtrees.



2 To resolve underflow, we can use "rotations", and "steal" from the immediate sibling if it has "extras".



3 Otherwise, we can perform a "node merge", by stealing from the parent.



- we repeat this until all nodes are "happy".

4 If this process continues until the root has only one subtree, it self-destructs.

5 Pseudocode:

Algorithm 11.5: *24Tree::delete*(k)

```
v ← 24Tree::search(k)                                    // node containing k
if v is not a leaf then
    v ← leaf that contains successor k' of k, and exchange k and k'
delete k and one empty subtree in v
while v has 0 keys do                                    // underflow
    if v is the root then delete v, make its child the new root and break
    if v has an immediate sibling u with 2 or more keys then
        rotate a key and subtree from u to v as in Figure 11.13 and break
    else
        merge v with an immediate sibling as in Figure 11.15
        v ← parent of v // and repeat the while-loop
```

6 Run-time: O(height · log b).

HEIGHT OF AN a-b-TREE

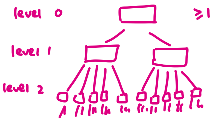
1 Claim:

$$\boxed{\text{height} \in O(\log_a n)}.$$

where n = # of KVPs.

* note: # of KVPs >> # of nodes.

Proof. Consider # of nodes on level $i \geq 1$.



≥1 node

≥ 2 nodes

≥ 2a nodes

≥ $2a^2$ nodes

⋮

This is at least $2a^{i-1}$.

Thus

$$\# \text{ of nodes} \geq 1 + \sum_{i=1}^{h} 2a^{i-1} \quad (\text{if } h = \text{height})$$

$$= 1 + 2\frac{a^h - 1}{a - 1}.$$

Then

$$n = \# \text{ of KVPS} \geq \underbrace{1}_{\text{root}} + \underbrace{(a-1)\, 2\frac{a^h - 1}{a - 1}}_{\text{everything else}}$$

$$= 1 + 2(a^h - 1)$$

$$= 2a^h - 1.$$

$$\Rightarrow h \leq \log_a\left(\frac{n+1}{2}\right) \in O(\log_a n)$$

as needed. ☒

2 With a similar proof, we can show

$$\boxed{\text{height} \in \Omega(\log_b n).}$$

3 We also choose $a \in \Theta(b)$, so the a-b-tree has height

$$\boxed{\text{height} \in \Theta(\log_b n).}$$

RUN-TIMES OF a-b-TREE OPERATIONS

1 All the run-times (in the RAM model) were asymptotically height · log b.

2 Then

$$\boxed{\text{height} \leq \log_a\left(\frac{n+1}{2}\right) \in O(\log_b n) = O\left(\frac{\log n}{\log b}\right).}$$

3 Thus

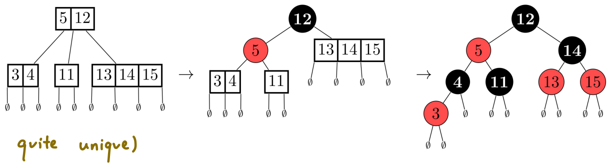
$$\boxed{\text{run-time} \in O(\log n).}$$

4 This is no faster than AVL trees.

CONVERT 2-4-TREES TO BSTs : RED-BLACK TREES

1 This has O(log n) run-time, but faster than AVL-trees in practice.

2 A d-node becomes a black node with d-1 red children.



(not quite unique)

3 The resultant trees have height O(log n).

4 A "red-black tree" has the following properties:

① It is a BST;
② Every node is either red or black;
③ Every red node has a black parent;
④ Every empty subtree T has the same "black depth";
ie the # of black nodes from the root to T.

5 Insert / delete:

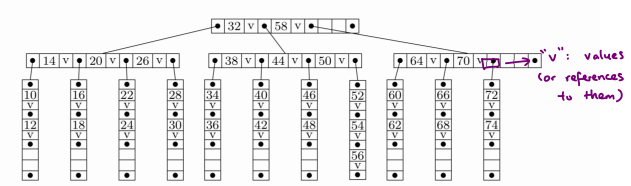
① Convert the red-black tree into a 2-4 tree;
② Perform the operation on the 2-4-tree; then
③ Convert the 2-4-tree back into a red-black tree.

B-TREES

1 A "B-tree" is an a-b-tree tailored to Emm.

2 Idea:

① Every node is one block of memory (of size B)
② b is chosen maximally s.t a node with b-1 KVPs fit into a block; & (b is called the "order" of the tree.)
* $b \in \Theta(B)$ typically.
③ $a = \lceil \frac{b}{2} \rceil$.



3 Note that

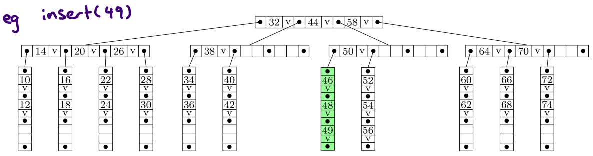
$$\boxed{\# \text{ block transfers in B-tree} = O(\text{height}) = \Theta(\log_b n) = \Theta(\log_B n).}$$

4 This is asymptotically optimal.

B-TREE IMPROVEMENT 1 — PRE-EMPTIVE SPLITTING / MERGING

1 Motivation:

eg insert(49)



① Standard insert does the following:
- load the blocks "on the way down"
- load them again "on the way up".

② We want to avoid this.

2 Idea:

① If node is not full, keep searching
② If node is full, immediately split.
③ Then keep searching in appropriate new node.

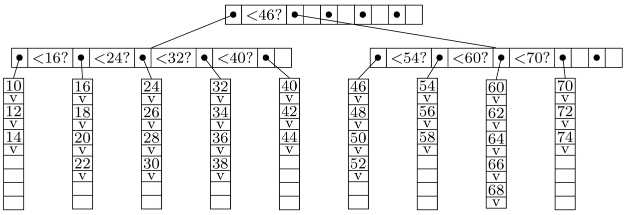
3 ie we split "just in case".

4 This can save half of the block transfers.

B-TREE IMPROVEMENT 2 — B⁺ TREES

1 Idea:

① Each node is one block of memory.
* ② All KVPs are stored at leaves, each of which are at least half full.
③ Interior nodes store only keys for comparison during search.
④ Interior non-root nodes have at least half of the possible subtrees.
⑤ Insert / delete use pre-emptive merging / splitting.



→ This can store $5^3 = 125$ KVPs.

2 Advantages:

① Bigger order;
② Can store more KVPs for the same height.

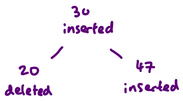
B-TREE IMPROVEMENT 3 — LSM-TREES

1 Idea:

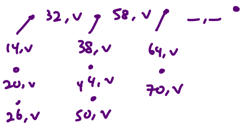
① Store dictionary in internal memory that logs all changes;
② To search: first search in C_0, then (if needed) in C_1
③ If internal memory full; do lots of updates in C_1 at once.

$\boxed{(32,v) \bullet (58,v) \bullet (\_,\_)}$

C_0 (log of changes)



C_1 (B-tree)



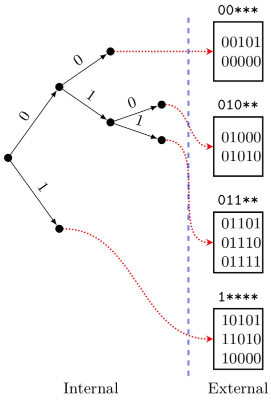
2 Note: many further improvements.

HASHING IN EMM

1. Each operation takes expected amortized $\Theta(1)$ runtime.
2. This means each operation also takes $O(1)$ block-transfers.
3. However, amortized bounds are a problem!
   → rehashing is the issue.
4. So, how do we do hashing without re-hashing?

TRIE OF BLOCKS

1. **Assumption:** we store non-negative integers (bitstrings)
2. **Idea:**
   ① Build pruned trie D (the "directory") of integers in internal memory.
   ② Stop splitting in trie when remaining items fit in one block.
   ③ Each leaf of D refers to block of external memory that stores the items.



3. **Search(k):**
   ① get bitstring for k, say 00101
   ② search in the trie
   ③ load block at leaf
   ④ search in it

→ This is 1 block transfer (assuming trie fits into internal memory)

4. **Insert:**
   ① Find the block that needs it, & insert.
   ② If the block is full, split the block by the next bit.
   ③ Repeat ② until the blocks have space.

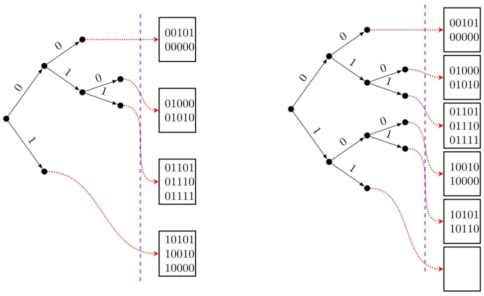


Figure 11.29: Inserting 10110 into a trie of blocks.

→ It is very likely we are done after 2-3 block transfers.

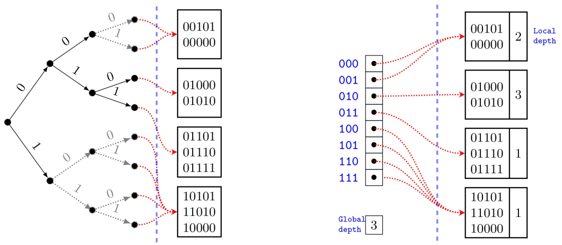
5. **Delete:** similar.

6. If no split works (ie duplicate bitstrings), we extend the hash function.
   ① Suppose h(k) yields the bitstring corresponding to k.
   ② Let $h_1 : k \rightarrow$ bitstring be another hash-function.
   ③ Then, use

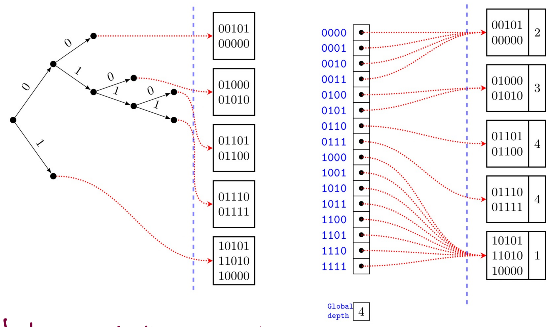
$$\boxed{h \underset{\uparrow \text{concatenation}}{++} h_1 : k \rightarrow \text{bitstring} = h(k) ++ h_1(k)}$$

EXTENDIBLE HASHING IN TRIE OF BLOCKS

1. We can also save links by "expanding" the trie and only storing the leaves.
2. We store the D as an array, rather than a trie:



3. We need to store
   ① the global depth d_D; &
   ② the local depth.
4. **Inset:** "convet" trie into table & v.v.
5. If the trie's height increases, we need to update the array.



– but we don't need to load blocks to update the array.

6. This uses no more block transfers than with tries.