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 $+, -, \times, \div$, 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.

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

eg $f(n) = 7sn + 500$ & $g(n) = 5n^2$, $c=1$ & $n_0 = 20$

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 $0 \le 2n^2+3n+11 \le cn^2 \quad \forall n \ge n_0$.

Sol^n. Consider $n_0 = 1$. Then

$1 \le n \Rightarrow 1 \le n^2 \Rightarrow 11 \le 11n^2$

$1 \le n \Rightarrow n \le n^2 \Rightarrow 3n \le 3n^2$

$2n^2 \le 2n^2$ (+)

$\Rightarrow 2n^2+3n+11 \le 11n^2+2n^2+3n^2 = 16n^2$

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

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

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

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

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

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

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

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

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

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

So runtime $\leq n \cdot n \cdot c = c \cdot n^2 \in O(n^2)$.

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. $\left|\frac{f(n)}{g(n)} - L\right| < \varepsilon \quad \forall n \geq n_\varepsilon$.

We want to show

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

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

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

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

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

$\Leftrightarrow f(n) \leq L g(n) + g(n)$ (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 $f(n) \in O(g(n))$. ∎

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

<< 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 $K \cdot f(n) \in \Theta(f(n)) \quad \forall K \in \mathbb{R}$ } (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 >>

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

*convention: "log" = "$\log_2$".

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 contrative:

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

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 $\neq 0$

$\Rightarrow f(n) \notin o(g(n))$. ∎

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

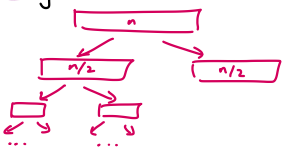
Suppose $f(n) \in \omega(g(n))$. Then $f(n) \in \Omega(g(n))$.

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

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

* $A \circ \pi$ 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

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

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

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

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
1. if n≤2 return
2. if A[n-2] ≤ A[n-1] then
3.     avgCaseDemo(A[0...⌈n/2⌉-1], ⌈n/2⌉)   // good case
4. else
5.     avgCaseDemo(A[0...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 $\pi^{-1}[0,\dots,\lceil \tfrac{n}{2} \rceil - 1]$ &

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

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

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

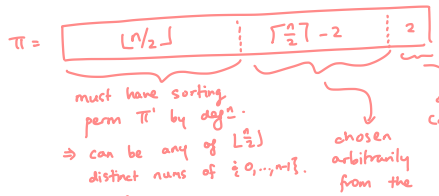
$$= |\Pi_n^{good}| + \sum_{\pi \in \Pi_n^{good}} T(\pi_{half})$$

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

We next prove the following claim:

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

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



must have sorting perm π' by def^n.
⇒ can be any of $\lfloor \frac{n}{2} \rfloor$ distinct nums of $\{0,\dots,n-1\}$.
⇒ $\binom{n}{\lfloor n/2 \rfloor}$ choices.

chosen arbitrarily from the remaining nums
⇒ $\binom{\lceil n/2 \rceil}{\lceil n/2 \rceil - 2}$ choices

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

Hence, the # of choices is

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

as needed. ※

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

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

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

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

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

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: $T_A^{exp}(n)$

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

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

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

AMORTIZED RUN-TIME: T_O^{amort}

Let O be an operation, and let $T^{actual}(O)$ be the actual run-time of O. Then, we say O has "amortized run-time $T^{amort}(O)$" if for any sequence of operations $O_1, \dots, O_k$ that could occur, we have

$$\sum_{i=1}^{k} T^{actual}(O_i) \le \sum_{i=1}^{k} T^{amort}(O_i).$$

POTENTIAL FUNCTION [OF A DATA STRUCTURE]: Φ

1. A "potential function" is a function $\Phi(\cdot)$ that depends on the status of the data structure.
2. In particular, for any sequence $O_1, \dots, O_k$ of operations:
   - ① $\Phi(i) \ge 0 \quad \forall i \ge 0$, where Φ_i = the value of Φ after $O_1, \dots, O_i$ have been executed; &
   - ② $\Phi(0) = 0$.

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

UPPER BOUNDS ACTUAL RUN-TIME

For any potential function Φ, the function

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

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

Proof. Fix a sequence of operations $O_1, \dots, O_k$. Summing up the amortized times and using a telescoping sum, we get

$$\begin{aligned}
\sum_{i=1}^{k} T^{amort}(O_i) &= \sum_{i=1}^{k} \left(T^{actual}(O_i) + \Phi(i) - \Phi(i-1)\right) \\
&= \sum_{i=1}^{k} T^{actual}(O_i) + \sum_{i=1}^{k} \left(\Phi(i) - \Phi(i-1)\right) \\
&= \sum_{i=1}^{k} T^{actual}(O_i) + \underbrace{\Phi(k)}_{\ge 0} - \underbrace{\Phi(0)}_{=0} \\
&\ge \sum_{i=1}^{k} T^{actual}(O_i).
\end{aligned}$$

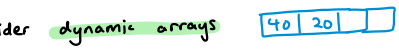
□

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 $\Phi(0) = 0$ & $\Phi(i) \ge 0 \ \forall i \ge 0$.
3. For each operation O, compute
   $$T^{amort}(O) := T^{actual}(O) + \Phi_{after} - \Phi_{before},$$
   and find an asymptotic upper bound for it.

EXAMPLE: DYNAMIC ARRAYS

1. Consider dynamic arrays  with two operations:
   - ① insert; &
   - ② rebuild, where we "lengthen" the array by a factor of 2.
2. Hence, insert takes $\Theta(1)$ time, & rebuild takes $\Theta(n)$ time (where n is the size of the array).
3. Define a "time unit" such that 'insert' takes "one unit of time" & 'rebuild' takes "n units of time".
4. 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 $\Phi(i) \ge 0$. Also initially size $= 0$ & cap ≥ 0, so $\Phi(0) = 0$ as desired.

Now, the amortized run-time for insert is

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

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

Similarly,

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

□

Chapter 2: Priority Queues and Heaps

ADT PRIORITY QUEUES

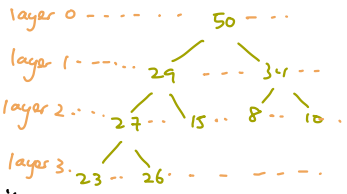
1. A "priority queue" stores items that have a priority, or key.
2. Operations:
   1. insert (a given item & priority as a k-v pair)
   2. deleteMax (return item with largest priority)
   3. size, isEmpty
3. We can use a PQ to sort:

```
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

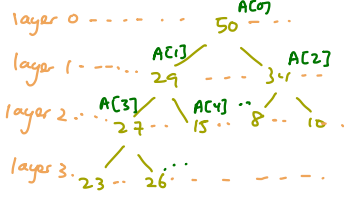
1. "Binary heaps" are binary trees with the following properties:
   1. Each level is filled except the last, which is filled from the left; } "structural" property
   2. key(i) ≤ key(parent(i)) } "heap-order property"



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:



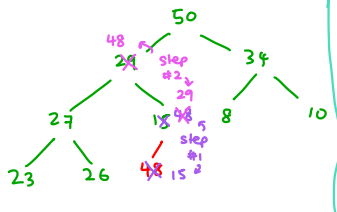
⇒

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

Then $parent(i) = \lfloor \frac{i-1}{2} \rfloor$, 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":

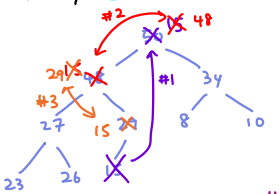


Pseudocode

```
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":



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

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
```

4. Run-time: $O(\text{height}) = O(\log n)$

HEAPSORT

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

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

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 $\Theta(n)$.

Why? → analyze a recursive version:

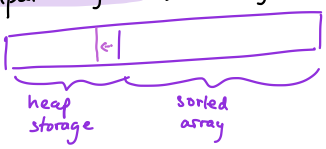
```
heapify(node i)
    if i has left child
        heapify(left child i)
    if i has right child
        heapify(right child i)
    fix-down(i)
```

Now, let
$T(n) :=$ runtime of heapify on n items.
(Assume n divisible as needed.)
Then:
size(left child) = size(right child) $= \frac{n-1}{2}$.

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

$$\Rightarrow T(n) \le 2T(\tfrac{n}{2}) + \Theta(\log n) \in O(n). \quad \square$$

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



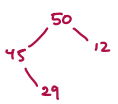
OTHER PQ OPERATIONS

We can also support the following operations:

1. "findMax" — finds the max element without removing it;
   - in a bin heap this takes $\Theta(1)$
   - 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 $k_{new} < \text{key}(i)$
   - does nothing if $k_{new} \ge \text{key}(i)$
   - 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

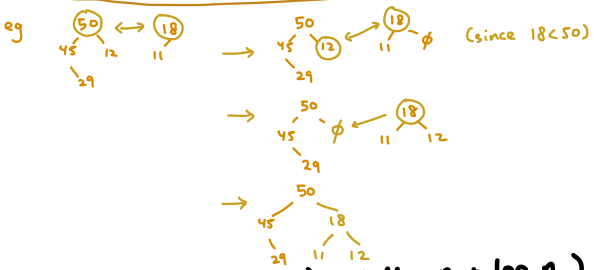
Operations for meldable heaps:

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



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

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

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

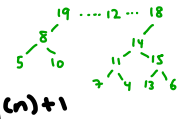
BINOMIAL HEAP (CO2.3)

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 $0,\dots,h$, and so at most $h+1 \leq \log n+1$ trees. ∎

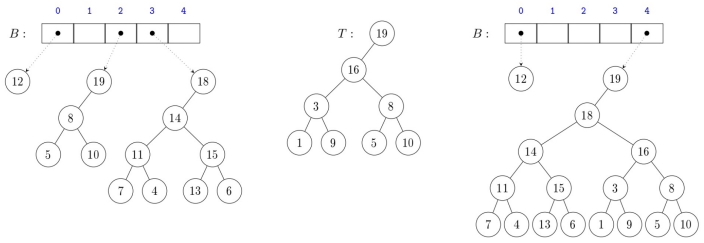
MAKING A BH PROPER

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.

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

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])
```

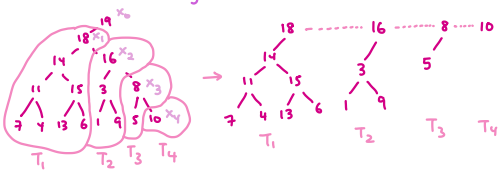


PQ OPERATIONS FOR PBHS

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 $T_1, \dots, T_h$, 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 $0 \le k < n$, 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

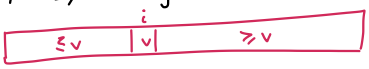
so

select(3) = 30.

PARTITION [THE FUNCTION]

The "partition" function does the following:

given the pivot-value v = A[p] and an array A[0,..,n-1], rearrange A s.t.

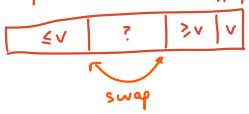


and return the pivot-index i.

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[p])
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

The "quick-select" algorithm:

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

* intuition:

Have:

Want:

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: $|I'| = n-1$

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

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
1.   p ← random(n)   // key step
2.   i ← partition (A, p)
3.   if i=k then
4.       return A[i]
5.   else if i>k then
6.       return quick-select1 (A[0,...,i-1], k]
7.   else if i<k then
8.       return quick-select1 (A[i+1, i+2, ..., n-1], k-i-1)
```

2. Then, what is P(pivot-index = i)?

   ⇒ pivot-value is equally likely to be any of A[0], ..., A[n-1]

   ⇒ ∴ pivot-index is equally likely to be any of 0, ..., n-1

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

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

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

$\uparrow$ $\nwarrow$
$\langle A,k\rangle$ $\langle i,R'\rangle$

(we count # of comparisons)

Then,

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

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

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

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

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

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

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

Proof. By induction.

$n=1$: Comp = $0 < 8(1) = 8$.

step:

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

bad	good	bad
$n/4$	$3n/4$	

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

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

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

∴ $T^{exp}(n) \in O(n)$. ∎

QUICK-SORT

Pseudocode:

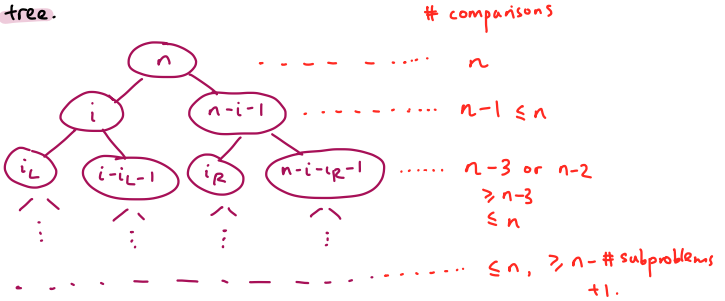
```
quick-sort1(A)
// A: array of size n
1. if n≤1 then return
2. p ← choose-pivot1(A)
3. i ← partition(A,p)
4. quick-sort1(A[0,...,i-1])
5. quick-sort1(A[i+1,...,n-1])
```

There are better implementations of this idea.

Runtime:

$T(n) = n + T(i) + T(n-i-1)$.

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



Thus

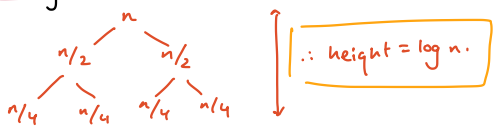
comparisons ≤ n · # layers = n · height of the recursion tree.

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.



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

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

We accomplish this via randomization of the algorithm:

```
RandQSort(A)
// A: array of size n
1. if n≤1 then return
2. p ← random(n)
3. i ← partition(A,p)
4. quick-sort1(A[0,...,i-1])
5. quick-sort1(A[i+1,...,n-1])
```

Then,

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

Note that

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

($T(A,R)$: instance; R: outcomes)

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

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

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 $T(n) \leq 2 \cdot n \cdot \ln(n)$.

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

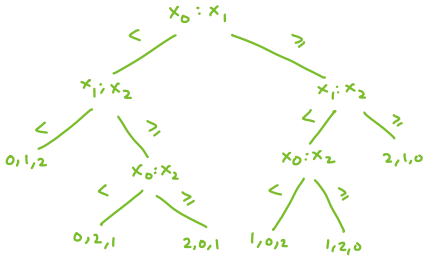
Step:
$$T(n) \leq n + \frac{2}{n}\sum_{i=2}^{n-1} 2i\ln(i)$$
$$= n + \frac{4}{n}\sum_{i=2}^{n-1} i\ln(i)$$
$$\leq n + \frac{4}{n}\int_2^n x\ln x\, dx \quad \text{(since } x\ln x \text{ is increasing)}$$
$$\leq n + \frac{4}{n}\left(\frac{1}{2}n^2\ln(n) - \frac{1}{4}n^2\right)$$
$$= n + 2n\ln(n) - n$$
$$= 2n\ln(n). \quad \blacksquare$$

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 decision tree Υ.



A decision tree for an algo to sort 3 elements x_0, x_1, x_2 with ≤ 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 Υ is binary, for any $0 \leq \ell \leq h$ there are at most 2^ℓ leaves in layers $0, \dots, \ell$.

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

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

Finally, consider the sorting perm π that has its leaf on layer h.
Executing algorithm A on I_π hence takes $h \in \Omega(n \log n)$ key-comparisons, so the worst-case bound holds. ☒

SORTING INTEGERS

BUCKET-SORT

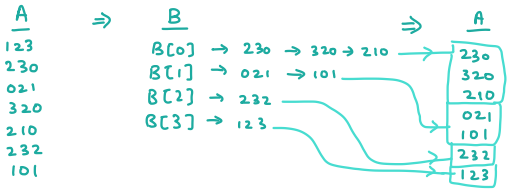
Bucket-sort can be used to sort a collection of integers by a specific "position" of digit.

eg the last digit 12345

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.
1.  initialize an array B[0...R-1] of empty
    lists
2.  for i←ℓ to r do
3.      append A[i] at the end of          // move array-items
        B [dth digit of A[i]]                 to buckets
4.  i←ℓ
5.  for j←0 to R-1 do                      // move bucket-items
6.      while B[j] is not empty do            to array
7.          move first element of B[j]
            to A[i]
8.          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
1.  if ℓ<r then
2.      bucket-sort(A, n, ℓ, r, d)
3.      if d<m then
            // find sub-arrays that have the same
               dth digit and recurse
4.          int ℓ'←ℓ
5.          while ℓ'<r do
6.              int r'←ℓ'
7.              while r'<r & dth digit of A[r'+1] = dth digit of
                A[ℓ'] do r'++
8.              MSD-radix-sort(A,n,ℓ', r', d+1)
9.              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
1.  for d←m down to 1 do
2.      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,

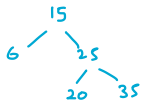
1. search(key): return the KVP for this key.
2. insert(key, value): add the KVP to the dictionary.
   - key is distinct from existing keys
3. delete(key): remove KVP with this key.

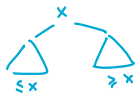
Assumptions:

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 $\in O(\text{height of tree})$.

* Height is $\Theta(n)$ is worst case, but typically much better ($\Theta(\log n)$)

LAZY DELETION

Consider a sorted array:

delete(20):

- usually takes $\Theta(n)$ 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 $O(n * \text{insert})$.

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

Then, the amortized time is $O(\text{insert} + \text{delete})$ 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: $\Theta(n)$ worst-case for delete but $\Theta(\log n)$ 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

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

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).

$h \in O(\log n)$

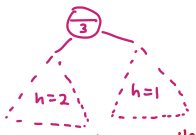
Let h be the height of an AVL tree with n nodes.

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

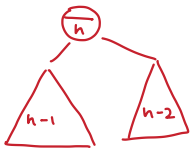
height 0:  # nodes: 1

height 1:  # nodes: 2

height 2:  # nodes: 4

height 3:  # nodes: $4+2+1 = 7$

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)
1.  z ← BST::insert(k,v)
2.  while z is not NIL do
3.      if (|z.left.height - z.right.height| > 1) then
4.          Let y ← taller child of z
5.          Let x ← taller child of y
6.          z ←P restructure(x,y,z)
7.          break
8.      z.height ← 1 + max{z.left.height, z.right.height}     // we alias this as "setHeightFromSubtrees"
9.      z ← z.parent
```

ROTATIONS [OF BSTS]

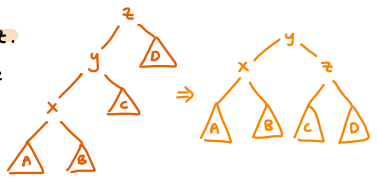
Let T be a BST with a node z that has a child y and a grandchild x. Then, a "rotation at z with respect to y & x" is a restructuring of T such that the result is again a BST, and sub-tree references have been changed only at x, y, z.

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; $z < y < x$.

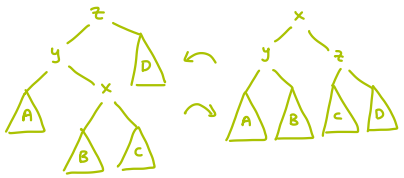
We want y to be the root.


```
rotate-left(z)
1.  y ← z.right
2.  z.right ←P y.left
3.  y.left ←P z
4.  setHeightFromSubtrees(z)
5.  setHeightFromSubtrees(y)
6.  return y
```

③ Double-right rotation: $y < x < z$.

We want x to be the root.

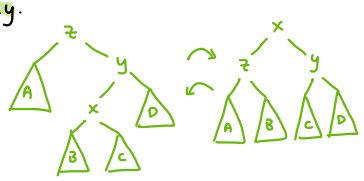
* can also be written as single-left rotation at y and then a single-right rotation at z (double-rotation)



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
1.  if y = z.left & x = y.left                    // x-y-z
2.      return rotate-right(z)                    // right rotation
3.  else if y = z.left & x = y.right              // y-z
4.      z.left ←P rotate-left(y)                  //   x
5.      return rotate-right(z)                    // double-right rotation
6.  else if y = z.right & x = y.left              // z-y
7.      z.right ←P rotate-right(y)                //   x
8.      return rotate-left(z)                     // double-left rotation
9.  else                                          // z-y-x
10.     return rotate-left(z)                     // left rotation
```

DELETE

Pseudocode:

```
AVL::delete(k)
1.  z ← BST::delete(k)        // z is the parent of the BST node that was removed
2.  while z is not NIL do
3.      if (|z.left.height − z.right.height| > 1) then
4.          let y = taller child of z
5.          let x = taller child of y
            // (break ties to prefer single rotation)
6.          z ← (p) restructure(x,y,z)
            // do not break — continue up the path & rotate if needed.
7.      setHeightFromSubtrees(z)
8.      z ← z.parent
```

RUNTIME

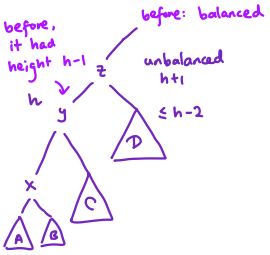
Both insert & delete have run-time $\Theta(\log n)$.

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)

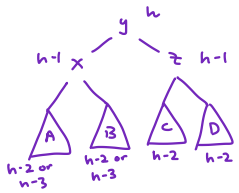
If we restructure at z during an insertion, then
① the subtree is balanced; and
② the subtree has now the height that it had before the insertion.

Proof.

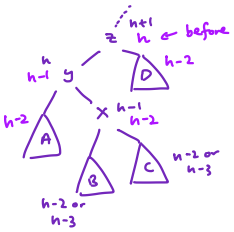


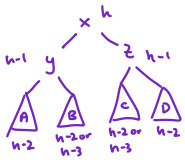
Case 1: "x" is the left node
- x has height h-2 before
- C has height h-2

After rotation:



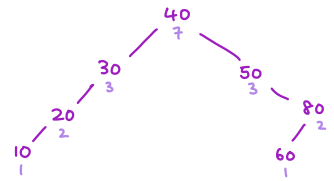
Case 2: x is the right child.





SCAPEGOAT TREES

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



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

HEIGHT = O(log n)

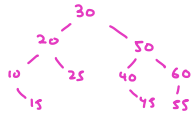
Any scapegoat tree has height $O(\log n)$.

Proof. At any leaf ℓ (which has size 1), the parent has size $\ge \frac{1}{q}$ by def^n of a scapegoat tree.
Repeating this argument, the grand-parent has size $\ge (\frac{1}{q})^2$, and so on.
As the root has size n, it follows that
$(\frac{1}{q})^d \le n$,
where d = max depth of a leaf = height.
Thus
$d \le \log_{(\frac{1}{q})}(n) \in O(\log n)$. ∎

PERFECTLY BALANCED BST

A "perfectly balanced BST" is one where for any node v, we have
$|v.\text{left.size} - v.\text{right.size}| \le 1$.

eg



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

INSERT

Pseudocode:

```
ScapegoatTree::insert(k,v)
1.  z ← BST::insert(k,v)
2.  S ← stack initialized w/ z
3.  while p ← z.parent ≠ NIL do
4.      p.size++
5.      S.push(p)
6.      z ← p
7.  while S.size ≥ 2 do
8.      p ← S.pop()
9.      if p.size < (1/q) max{p.left.size, p.right.size} then
10.         completely rebuild the subtree rooted at p as a perfectly balanced BST
11.         break
```

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

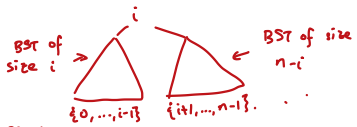
RANDOMIZED BUILT BST HAS EXPECTED HEIGHT $\Theta(\log n)$

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

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

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

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

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

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

We know

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

But

avg of height of BST is <u>not</u> $\Theta(\log n)$!

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

Intuition on why:

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

But

$$\frac{\#\text{ BSTs w/ shape } \triangle}{\text{total \# of BSTs}} = \frac{\#\text{ BST on } n-1 \text{ items}}{\#\text{ BST on } n \text{ items}}$$
$$= \frac{C(n-1)}{C(n)} \approx \frac{1}{4}$$

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

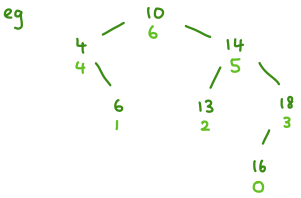
TREAPS

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

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 $\leq$ root $\leq$ 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



☑ the keys
☑ the priorities
(stored in an array).

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
1.  while (y ← z.parent is not NIL & z.priority > y.priority) do
2.      if z is the left child of y then rotate-right(y)
3.      else rotate-left(y)
```

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

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

RUN-TIME / SPACE

Note that treaps are

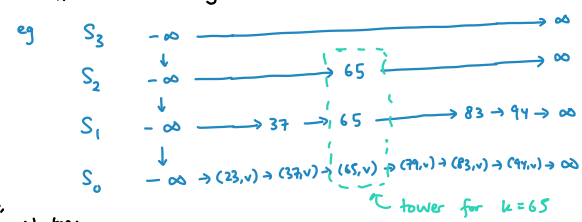
1. BSTs with expected height $O(\log n)$, and so the expected runtime of all operations is $O(\log n)$;
2. have a large space over-head, since we store the BST, and parent-references & priorities for each node.

SKIP LISTS

A "skip list" is a hierarchy S of ordered linked lists (levels) $S_0, \dots, S_h$, 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 $S_0 \supseteq S_1 \supseteq \dots \supseteq S_h$; &
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.

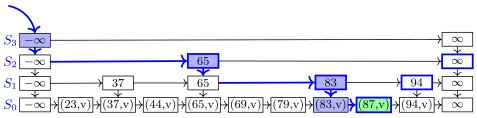
getPredecesors(k)

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

Algorithm 5.3: *skipList::getPredecessors(k)*

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

Example:



SEARCH

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

Algorithm 5.4: *skipList::search(k)*

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

INSERT

For 'insert', we first call 'getPredecesors', 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,

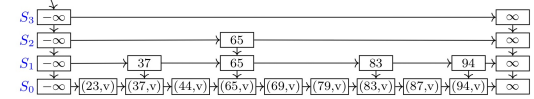
$$P(\text{tower of key } k \text{ has height} \geq i) = \frac{1}{2^i}.$$

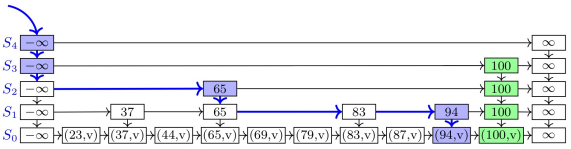
Algorithm 5.5: *skipList::insert(k, v)*

```
1 for (i ← 0; random(2) = 1;) do i++                  // random tower height
  // Do we need to increase the skip list height?
2 for (h ← 0, p ← root.below; p ≠ NIL;p ← p.below) do h++   // compute height
3 while (i ≥ h) do
4 |   create a new sentinel-only list and link it to the previous root-list appropriately
5 |   root ← leftmost node of this new list
6 |   h++
  // Actual insertion
7 P ← getPredecessors(k)
8 p ← P.pop()                                  // insert (k, v) in S0
9 z_below ← new node with (k, v), inserted after p
10 while i > 0 do                              // insert k in S1, ..., Si
11 |   p ← P.pop()
12 |   z ← new node with k, inserted after p
13 |   z.below ← z_below; z_below ← z
14 |   i ← i − 1
```

Example:

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





DELETE

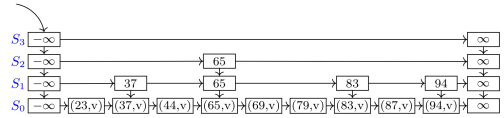
To delete in a skip list, we find the key (which gives the stack of predecessors) and then remove it from all lists that it was in.

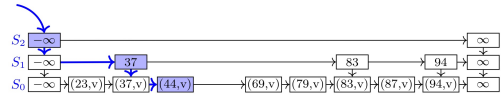
We also "clean-up" the stack: if deleting a key results in multiple lists that store only sentinels, then delete all but one of them.

Algorithm 5.6: *skipList::delete(k)*

```
1 P ← getPredecessors(k)
2 while P is non-empty do
3 |   p ← P.pop()          // p could be predecessor of k in some list
4 |   if p.after.key = k then remove p.after from the list
5 |   else break                                  // no more copies of k
6 p ← root                                       // clean up duplicate empty lists
7 while p.below ≠ NIL and p.below.after is the ∞-sentinel do
8 |   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(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 def $I_i \leq 1$ & $I_i \leq |S_i|$, so

$$E[I_i] \leq \min\{1, \tfrac{n}{2^i}\}.$$

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

In particular, notice

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

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

$$E\left[\sum_{i \geq 0} |S_i|\right] = \sum_{i \geq 0} \frac{n}{2^i} = n \sum_{i \geq 0} \frac{1}{2^i} = 2n.$$

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(forward steps of S_i) ≤ 1

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

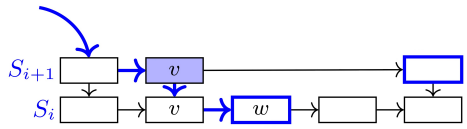
Proof. Let v be the leftmost node in list S_i that we visited during the search.

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

$$O(E[h + \textstyle\sum_{i \geq 0} F_i]),$$

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

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

Terms:

① "Access frequency" — amount of times key is accessed

② "Access probability" — proportion of accesses for a key

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

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

DYNAMIC SCENARIO

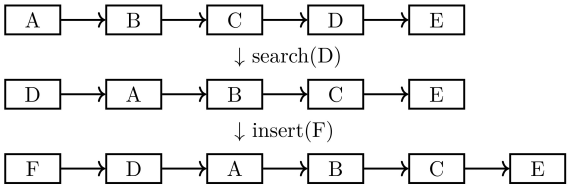
Here, we do not know how frequently a key is going to be accessed, so we cannot hope to build the best-possible data structure.

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

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



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

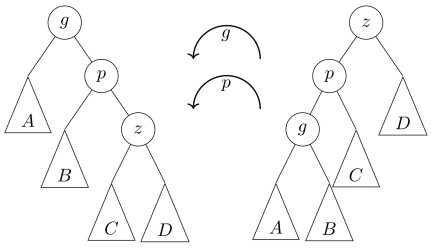
SPLAY TREES

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.

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.



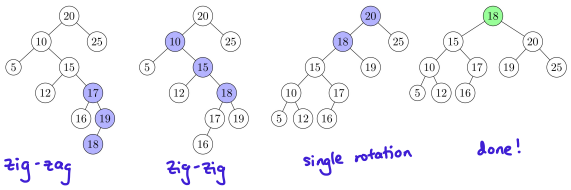
INSERT

Pseudocode:

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

```
1  z ← BST::insert(k, v)
2  while (z has a parent p and a grand-parent g) do
3      if (g, p, z) then                                  // Zig-zig rotation rightward
4          p ← rotate-right(g), z ← rotate-right(p)
5      else if (g, p, z) then                             // Zig-zag rotation rightward
6          g.left ← rotate-left(p), z ← rotate-right(g)
7      else if (g, p, z) then                             // Zig-zag rotation leftward
8          g.right ← rotate-right(p), z ← rotate-left(g)
9      else //  (g, p, z)                                 // Zig-zig-rotation leftward
10         p ← rotate-left(g), z ← rotate-left(p)
11 if (z has a parent p) then                             // single rotation
12     if (z = p.left) then z ← rotate-right(p)
13     else z ← rotate-left(p)
```

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 $z, z_1, \dots, z_d$, where z_d = root.
After adding z, the size of the subtrees at all $z_1, \dots, z_d$ increase by 1, whilst it is unchanged at all other nodes.
So, the contribution to Φ only changes at $z_1, \dots, z_d$, 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

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

($n_{z_1}^{before} \geq 1$, $n_{z_d}^{after} = n$)

as needed. ☑

Lemma #2: Let O_j 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 $T^{actual}(\text{insert}) = 1 + d$, where d = depth from root to z.
Let the seq of operations in insert be

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

Then

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

and so

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

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; &] drop comparison based
4. We can do even better!

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
height $\geq \lceil \log(n) \rceil$.
However, we also have $n+1$ "not found" leaves, corresponding to
$k \in (-\infty, x_0)$, $k \in (x_0, x_1)$, …, $k \in (x_n, +\infty)$.

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

Proof. In particular, we now show
# of leaves $\geq 2n+1$
(and so height $\geq \lceil \log(2n+1) \rceil$).
To do so, we create $2n+1$ instances

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

Searching for x_i could result in the possibilities:
$x_0, x_1, x_2, \dots, x_{n-1}$
Searching between x_i:
$(x_0, x_1), (x_1, x_2), \dots, (x_{n-2}, x_{n-1})$
Searching outside:
$(-\infty, x_0)$, $(x_{n-1}, +\infty)$.

Claim: no two reach the same leaf.

Proof by contradiction.

leaf reached w/ k', k''.

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 $x_0 \leq k', k'' < x_a$, 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

Normal binary search takes

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

But can we do better?

Pseudocode:

```
binary-search-optimized(A,n,k)
1.  ℓ←0, r←n-1, χ←0
      // χ is a bool var that tells us whether
         we're in the left subarray
2.  while (ℓ<r)
3.      m← ⌊(ℓ+r)/2⌋
4.      if (A[m]<k) then ℓ←m+1
5.      else r←m, χ←1
6.  if (k<A[ℓ]) then
7.      return "not found, bw A[ℓ-1] and A[ℓ]"
8.  else if (χ=1) or (k=A[ℓ]) then
9.      return "found at A[ℓ]"
10. else return "not found, bw A[ℓ] & A[ℓ+1]
```

We claim

1. this algorithm terminates;
2. this gives the correct answer; &
3. this uses at most $\lceil \log n \rceil + 2$ comparisons (without χ), which is about $\lceil \log(2n+1) \rceil + 1$ comparisons.
4. with the χ, this uses $\leq \lceil \log(2n+1) \rceil$ comparisons, so this is optimal.

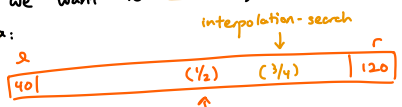
Proof. see TB.

INTERPOLATION SEARCH

"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.

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,

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

(start index)

and otherwise, the search works like bin-search.

Pseudocode:

```
interpolation-search(A,n,k)
1.  ℓ←0, r←n-1
2.  while (ℓ≤r)
3.      if (k<A[ℓ] or k>A[r]) return "not found"
4.      if (k=A[r]) then return "found at A[r]"
5.      m← ℓ + ⌊ (k-A[ℓ])/(A[r]-A[ℓ]) · (r-ℓ) ⌋
6.      if (A[m] == k) then return "found at A[m]"
7.      else if (A[m]<k) then ℓ←m+1
8.      else r←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,

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

→ see next page for proof under a realization.

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

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!

In particular,

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

OPTIMIZED INTERPOLATION-SEARCH

1 Pseudocode:

```
interpolation-search-modified(A, n, k)
1.  if (k < A[0] or k > A[n-1])  return "not found"
2.  if (k = A[n-1])  return "found at index n-1"
3.  ℓ ← 0,  r ← n-1                    // A[ℓ] ≤ k < A[r]
4.  while (N ← (r-ℓ-1) > 1)
5.      m ← ℓ + ⌈ (k-A[ℓ]) / (A[r]-A[ℓ]) · (r-ℓ-1) ⌉
6.      if (A[m] ≤ k)
7.          for h = 1, 2, ...
8.              ℓ ← m + (h-1)⌈√N⌉,  r' ← min{r, m+h⌈√N⌉}
9.              if (r'=r or A[r'] > k) then r ← r' and break
10.     else ...
11.         if (k = A[ℓ]) return "found at index ℓ"
12.         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 ≈√n each time until
- A[m+x√n] > k; or
- m+x√n is out of bounds.

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

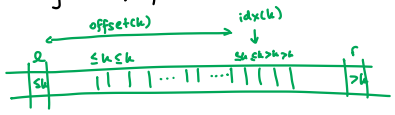
2 If $T(n)$ = # of comparisons on n numbers, then

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

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

3 Moreover, avg # of probes ≤ 2.5.

Proof.



Assume nums in $A[\ell+1 \dots r-1]$ have been chosen uniformly at random bw $A[\ell]$ & $A[r]$.

we want $E(\#\text{probes}) \le 2.5$.

Then see that

$P(\#\text{probes} \ge 1) = 1$, $P(\#\text{probes} \ge 2) \le 1$, &

$P(\#\text{probes} \ge 3) = P(A[\text{id of } 2^{\text{nd}} \text{ probe}] \le k)$

$= P(idx(k) \ge m + \lceil\sqrt{n}\rceil)$

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

$\le P(|idx(k) - m| \ge \sqrt{n})$

$= P(|idx(k) - E(idx(k))| \ge t)$

$\le \frac{V[idx(x)]}{t^2}$ $(V(cx) = Var(x))$.

↑ Chebyshev

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

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

Then

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

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

and so $= p$,

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

Thus

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

So

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

and in general

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

So

$E(\#\text{probes}) = \sum_{h=0}^{\infty} h \cdot P(\#\text{probes} = h)$

$= \sum_{h=1}^{\infty} P(\#\text{probes} \ge h)$

$\le \underbrace{1}_{h=1} + \underbrace{1}_{h=2} + \sum_{h=3}^{\infty} \frac{1}{4(h-2)^2}$

$= 2 + \frac{1}{4} \sum_{i=1}^{\infty} \frac{1}{i^2} = 2 + \frac{1}{4}\frac{\pi^2}{6} \le 2.5$. ∎

4 Therefore,

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

Proof. Specifically, we prove

$T^{avg}(n) \le 2.5 \lceil \log\log n \rceil$ for $n \ge 4$.

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

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

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

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

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

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

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

Proof by induction, and we only consider the step:

$T^{avg}(n) \le T^{avg}(n') + 2.5$, $n' \le \sqrt{n}$

$\le 2.5 \lceil \log\log(n') \rceil + 2.5$

$\le 2.5 \lceil \log\log(\sqrt{n}) \rceil + 2.5$

$\le 2.5 (\lceil \log\log n \rceil - 1) + 2.5 = 2.5 \lceil \log\log n \rceil$,

as needed. ∎

DICTIONARIES OF WORDS

WORDS

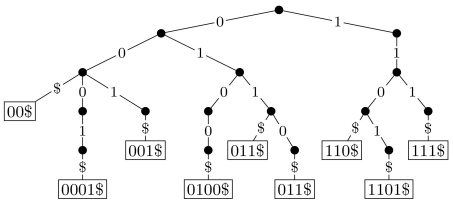
- "Words" are strings; ie an array of chars in some alphabet Σ.
  - eg $\Sigma = \{0,1\}$, Σ = ASCII, etc
- Note words have arbitrary length.
- We sort words lexicographically:

c i a ' t ' f i ' s ' h
c o r n
c o r n e r

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

TRIES

- A "trie" is a dictionary of words.
- It is "prefix-free" — no string is a prefix of another.
- This is satisfied if
  1. all strings have the same length; or
  2. all strings end with an "end-of-word" character '$'.



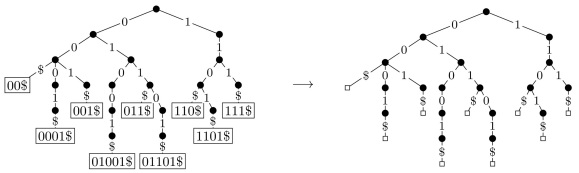
* Note: items (keys) are only stored at the leaves.

OPERATIONS ON TRIE TAKE O(|x|) TIME

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

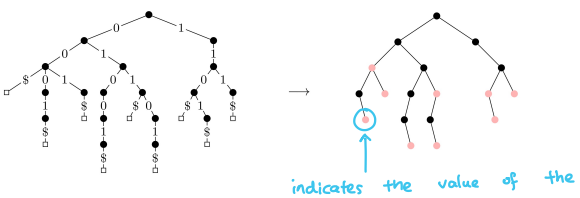
NO-LEAF-LABEL TRIES

- Idea: we don't store the actual keys at the leaves.
- The key is stored implicitly through the characters along the path to the leaf.
- Hence, this halves the amount of space needed.



ALLOW-PROPER-PREFIXES TRIES

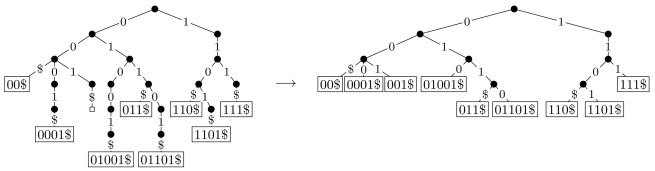
- Idea: we permit storing words at interior vertices, so we can accomodate words that are prefixes of others.



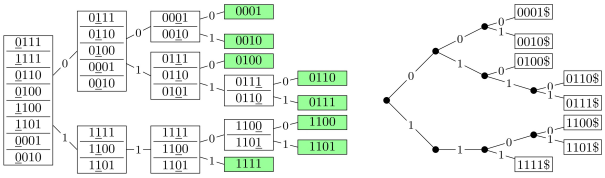
indicates the value of the 'flag' which says whether the node stores a value.

PRUNED TRIES

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

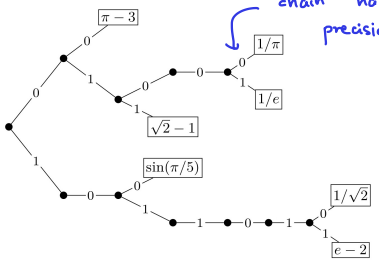


- Note that we must store the full keys.
- Also note that pruned tries are the recursion trees of MSD-radix-sort:



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 $B_1, \ldots, B_n$, 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

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

Hence

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

and so

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

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

and so

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

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.

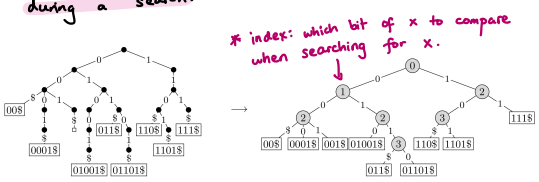


Figure 6.13: An example of a compressed trie.

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

   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.

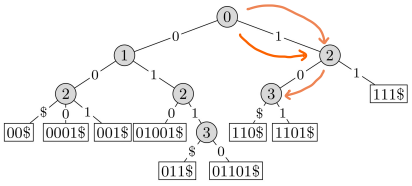
   * if we only consider the space taken by the nodes.

   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 ← 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

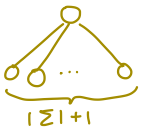
Run-time: $O(|x|)$

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.

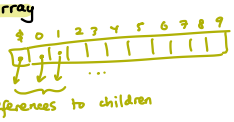
MULTIWAY TRIES

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



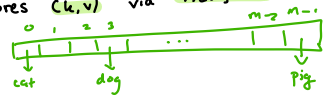
   ② List

   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

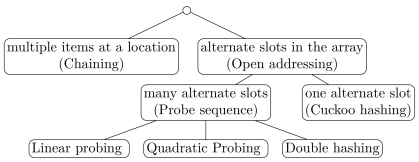
1. We can do direct addressing after modifying (ie "hashing") the key.
2. Assumptions:
   1. keys come from a universe U
      - typically, U integers, $|U|$ finite
   2. Size of hash table, M, is pre-determined
   3. We use a hash function $h: U \to \{0, \dots, M-1\}$ that maps indexes in U to an index 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 $T[i]$

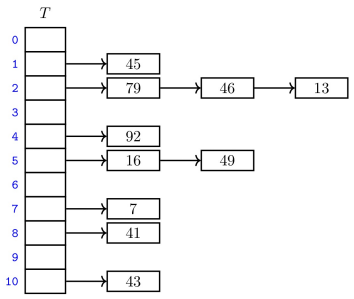
COLLISIONS

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 Θ(length of $T[h(k)]$)
      ↳ 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,

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

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

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

Hence, under uniform hashing,

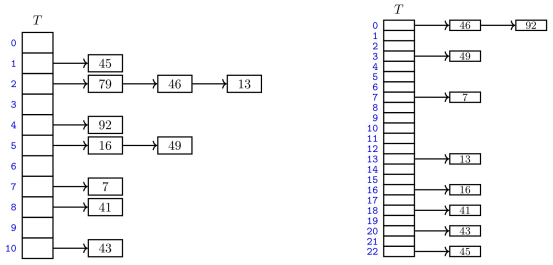
search, delete take time $O(1+\alpha)$.

REHASHING

We choose α by choosing M.

So, as n increases, we increase M so that α stays small.

This is called "rehashing".



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

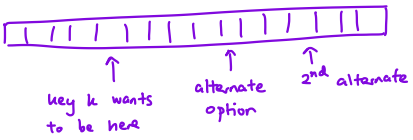
Moreover,

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

HASHING BY PROBING

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

Idea: we allow keys to use multiple slots.



For each key, we have a "probe sequence"

$$\langle h(k,0),\ h(k,1),\ \ldots,\ h(k,M-1) \rangle$$

key; index of option

LINEAR PROBING

Here, we define

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

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

$\langle 4, 5, 6, 7, 8, 9, 10, 0, 1, 2, 3 \rangle$.

Linear probing builds big "clusters" of elements.

OPERATIONS OF PROBE HASHING

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.

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

Pseudocode:

Algorithm 7.1: *probeSequenceHash::insert*(k, v)

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

Algorithm 7.2: *probeSequenceHash::search*(k)

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

Algorithm 7.3: *probeSequenceHash::delete*(k)

1. $i \leftarrow$ index returned by *probeSequenceHash::search*(k)
2. $T[i] \leftarrow$ 'deleted'
3. re-hash the table if there are too many 'deleted' items

QUADRATIC PROBING

Here, we define

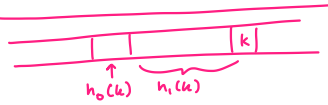
$$h(k,i) = (h(k) + c_1 i + c_2 i^2) \bmod M$$

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

DOUBLE HASHING

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

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



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

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

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

In particular, search & delete have $O(1)$ worst-case run-time.

Pseudocode for insert:

Algorithm 7.4: *cuckooHash::insert*(k, v)

```
1  i ← 0
2  repeat
3      if T_i[h_i(k)] is NIL then
4          T_i[h_i(k)] ← (k, v)
5          return "success"
6      else
7          swap((k, v), T_i[h_i(k)])
8          i ← 1 − i
9  until we have tried 2n times
   // If we reach this point then the hash table is probably too full.
10 return "unsuccessful, should re-hash"
```

COMPLEXITY OF OPEN ADDRESSING STRATEGIES

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)

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

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

CHOOSING HASH FUNCTIONS

MODULAR METHOD

Here, we let

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

where we usually pick M to be prime.

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

MULTIPLICATION METHOD

Here, we let

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

where

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

HASHING MULTI-DIMENSIONAL DATA

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

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

eg A·P·P·L·E → (65, 80, 80, 76, 69) (ASCII)

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

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

To compute this in $O(|w|)$ time without overflow, we use "Horner's rule" 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

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

But then we cannot compute the hash value quickly!

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

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

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

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

CARTER-WEGMAN HASH FUNCTION

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

$$h_{a,b}(k) = f_{a,b}(k) \bmod m$$
$$= (a \cdot k + b \bmod p) \bmod m$$

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

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

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

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

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

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

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

We know that

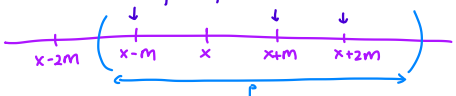
$$\underbrace{f_{a,b}(k)}_{x} \neq \underbrace{f_{a,b}(k')}_{x'},$$

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

choices for x' 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

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

∎

Chapter 8: Range Search

Idea:
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:

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.

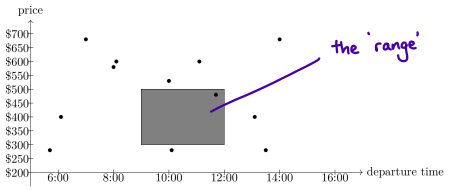
Run-time: $O(\log n + s)$,
where "s" is the "output size".

MULTI-DIMENSIONAL DATA

Idea: "points" instead of single values.
ie $(x_0, x_1, \ldots, x_n)$

* we will assume all data are integers.

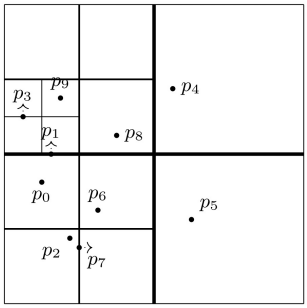
Range search then looks like:



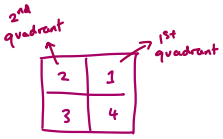
This is called an "orthogonal range-search":
① We are given a query rectangle
$A = [x, x'] \times [y, y']$; &
② we want the points in A.

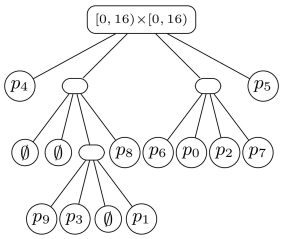
QUAD-TREES

For "quad-trees":
① stores points in 2D;
② we assume the points are in a bounding box $[0, 2^k) \times [0, 2^k]$
eg $[0, 16) \times [0, 16)$



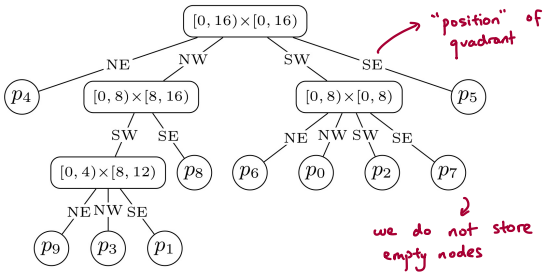
How to construct?
① Split the bounding box into quadrants;
② Define the corresponding children;
③ Repeat at children until ≤1 point left in all regions.





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

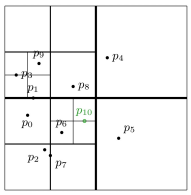
Alternative depiction:

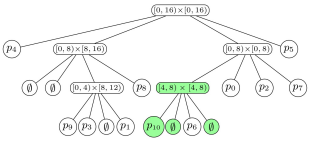


① We do not need to store the "sub-areas" (we always cut in half)

OPERATIONS OF QUAD-TREES

Insert:





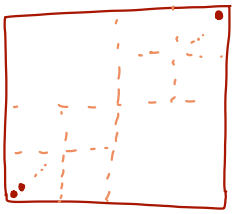
Run-time:

① No good bound depending on n;

② We can only say it is $O(\text{tree height})$

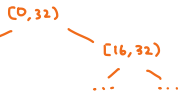
HEIGHT OF A QUAD-TREE

But we have no bound on the height either!

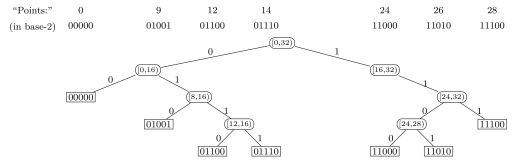
eg  → height arbitrary.

However, we can bound the height if the coordinates are integers:
specifically in the range $\{0, \dots, 2^\ell - 1\}$.
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 integers in $\{0, \dots, 2^\ell - 1\}$ $\leq$ the length of longest bitstring = ℓ (bitstrings of length ℓ)

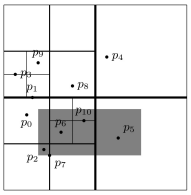
This argument generalizes to higher dimensions (eg 2D, ie quad trees)

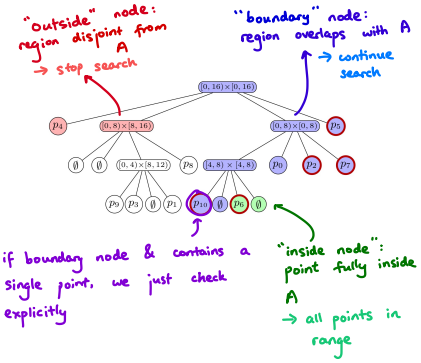
In particular, a quad-tree is a pruned trie where we split by two keys in parallel.

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

RANGE-SEARCH IN QUAD-TREES

Idea:





Pseudocode:

Algorithm 8.1: *quadTree::rangeSearch*($r \leftarrow root, A$)

Input : Node r of a quad-tree. Query-rectangle A

```
1  R ← square associated with node r
2  if R ⊆ A then                                   // inside node
3  |  report all points in subtree at r and return
4  else if R ∩ A is empty then                     // outside node
5  |  return
6  else                                            // boundary node
7  |  if r is a leaf then
8  |  |  if r stores a point p and p ∈ A then
9  |  |  |  return p
10 |  |  else
11 |  |  |  return
12 |  else
13 |  |  foreach child v of r do
14 |  |  |  quadTree::rangeSearch(v, A)
```

① Checking "$R \cap A = \emptyset$" is $O(1)$.
- since we use rectangles.

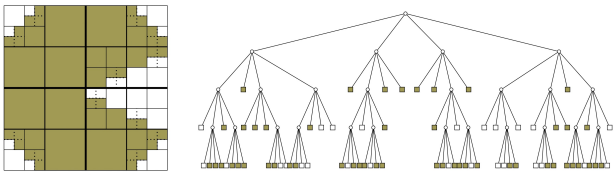
Run-time:

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

② Only thing we can say:
run-time $\in O(\text{size of quad tree})$.

OTHER USES OF QUAD-TREE

We can use quad-trees to store pixelated images:



kd-TREE

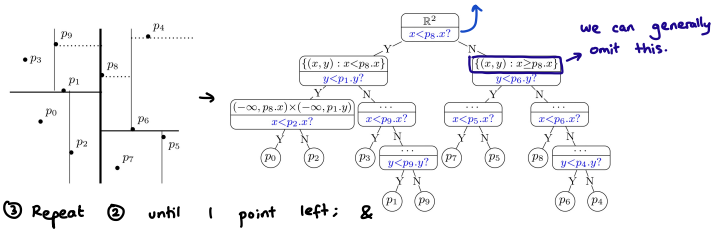
* d = "dimensional"

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 $\approx \frac{n}{2}$ each.

RANGE-SEARCH FOR kd-TREES

Pseudocode:

Algorithm 8.2: *kdTree::rangeSearch*($r \leftarrow root, A$)

```
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(# of boundary nodes + s).

output size

Then,

# of boun