Chapter 1

Foundations

1 Calculus

Theorem 1.1 (Taylor's Theorem)**.** *If f has derivatives of all orders in an open interval I containing a, then for each positive integer n and for each x in I,*

$$f(x) = \sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!}(x-a)^k + R_n(x)$$

where $R_n(x) = \frac{f^{(n+1)}(c)}{(n+1)!}(x-a)^{n+1}$ for some c between a and x.

Theorem 1.2. *If there is a positive constant M such that $|f^{n+1}(t)| \leq M$ for all t between x and a, inclusive, then $R_n(x) \leq M\frac{|x-a|^{n+1}}{(n+1)!}$*

Taylor series:

$e^x = 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \cdots = \sum_{n=0}^{\infty} \frac{x^n}{n!},\ |x| < \infty$

$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!},\ |x| < \infty$

$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots + (-1)^n \frac{x^2 n}{(2n)!} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!},\ |x| < \infty$

$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots + (-1)^{n-1} \frac{x^n}{n} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^{n-1} x^n}{n},\ -1 < x \leq 1$

$\ln \frac{1+x}{1-x} = 2\tanh^{-1} x = 2(x + \frac{x^3}{3} + \frac{x^5}{5} + \cdots + \frac{x^{2n+1}}{2n+1} + \cdots) = 2\sum_{n=0}^{\infty} \frac{x^{2n+1}}{2n+1},\ |x| < 1$

$\tan^{-1} x = x - \frac{x^3}{3} + \frac{x^5}{5} - \cdots + (-1)^n \frac{x^{2n+1}}{(2n+1)} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)},\ |x| \leq 1$

For $f(x) = \ln(1+x)$, we have $R_n(x) = \frac{(-1)^n x^{n+1}}{(n+1)(1+c)^{n+1}}$. It follows that $x - \frac{x^2}{2} \leq \ln(1+x) \leq x$ when $x \geq 0$, and $\ln(1+x) \leq x - \frac{x^2}{2}$ when $x < 0$.

1.1 Common Functions and Inequalities

$1 + x \leq e^x$ for any x
$e^x \leq 1 + x + x^2$ when $x \leq 1$

$(1+\frac{x}{n})^n < e^x$, for any $\frac{x}{n} > -1$
$\lim_{n\to\infty}(1+\frac{x}{n})^n = e^x$ for all x
$\frac{x}{1+x} \le \ln(1+x) \le x$ when $x > -1$
Stirling's approximation: $n! = \sqrt{2\pi n}(\frac{n}{e})^n(1+\Theta(\frac{1}{n}))$
$n! = \sqrt{2\pi n}(\frac{n}{e})^n e^{\alpha(n)}$ where $\frac{1}{1+12n} < \alpha(n) < \frac{1}{12n}$
$\lg^* n = \min\{i \ge 0 : \lg^{(i)} \le 1\}$
Harmonic function $\ln(1+n) \le H_n = 1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n} \le \ln n+1$, $H_n = \ln n + O(1)$ (Hint: $\frac{1}{n+1} < \ln(1+\frac{1}{n}) < \frac{1}{n}$ for all $n > 0$ or consider $\int_1^n \frac{1}{x}dx$)
$(1+x)^n \ge 1+nx$ for all $x > -1$

2 Probability inequalities

Theorem 2.1 (Union Bound)**.** *For any events $A_1, \ldots, A_n$, we have:*

$$P(\bigcup_{i=1}^{n} A_i) \le \sum_{i=1}^{n} P_i$$

Theorem 2.2 (Cauchy-Schwarz)**.** *For any r.v.s X and Y with finite variances,*

$$E(XY) \le \sqrt{E(X^2)E(Y^2)}$$

Theorem 2.3 (Jensen)**.** *Let X be a random variable. If g is a convex function,then $E(g(X)) \ge g(E(X))$. If g is a concave function, then $E(g(X)) \le g(E(X))$. In both cases, the only way that equality can hold is if there are constants a and b such that $g(X) = a + bX$ with probability 1.*

Markov, Chebyshev, Chernoff: bounds on tail probabilities. They provide bounds on the probability of an r.v. taking on an "extreme" value in the right or left tail of a distribution.
See:

- lecture notes from MIT.
- Stéphane Boucheron, Gábor Lugosi, and Olivier Bousquet. Concentration inequalities. In *Summer School on Machine Learning*, pages 208–240. Springer, 2003

By $E(X) = \int_0^\infty \mathbb{P}[X > t]\, \mathrm{d}t$, we have:

Theorem 2.4 (Markov)**.** *For any nonnegative r.v. X and constant $t > 0$,*

$$\mathbb{P}[X \ge t] \le \frac{E(X)}{t}$$

It follows that for any r.v. X and real number t, $\mathbb{P}[X \ge t] = \mathbb{P}[\phi(X) \ge \phi(t)] \le \frac{E(\phi(X))}{\phi(t)}$, where ϕ is any strictly increasing, nonnegative function.
Let $\phi(x) = x^2$ and $\phi(x) = e^x$, we have:

Theorem 2.5 (Chebyshev)**.** *Let X have mean μ and variance σ^2. Then for any t,*

$$P(|X-\mu| \ge t) \le \frac{\sigma^2}{t^2}$$

Theorem 2.6 (Chernoff). *For any r.v. X and constants $s > 0$ and t,*

$$\mathbb{P}(X \geq t) = \mathbb{P}[e^s X \geq e^s t] \leq \frac{E(e^{sX})}{e^{st}}$$

There are many different forms of Chernoff bounds, each tuned to slightly different assumptions. Below is the case of a sum of independent Bernoulli trials.

Theorem 2.7 (Chernoff Bounds). *Let $X = \Sigma_{i=1}^n X_i$, where $X_i = 1$ with probability p_i and $X_i = 0$ with probability $1 - p_i$, and all X_i are independent. Let $\mu = E(X) = \Sigma_{i=1}^n p_i$. Then:*

- *Upper Tail: $Pr[X \geq (1+\delta)\mu] \leq e^{-\frac{\delta^2}{2+\delta}\mu} \quad \delta \geq 0$*
- *Lower Tail: $Pr[X \leq (1-\delta)\mu] \leq e^{-\frac{\delta^2}{2}\mu} \quad 0 \leq \delta \leq 1$*

hint. $E(e^{sX_i}) = pe^s + (1-p) \leq e^{p(e^s-1)} \Rightarrow E(e^{sX} \leq e^{\mu(s-1)})$ □

3 Asymptotic analysis

The notations we used to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are set of natural numbers $\mathbb{N} = \{0, 1, 2, \dots\}$. Sometimes we *abuse* asymptotic notation by extending the domain to real numbers or retricting it to a subset of natural numbers.

3.1 Definitions of aysmptotic notation

1. $O(g(n)) = \{f(n) : \exists c, n_0 > 0 \text{ such that } \forall n \geq n_0,\ 0 \leq f(n) \leq c \cdot g(n)\}$
2. $o(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \text{ such that } \forall n \geq n_0, f(n) < c \cdot g(n)\}$
3. $\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0 \text{ such that } \forall n \geq n_0,\ f(n) \geq c \cdot g(n)\}$
4. $\omega(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \text{ such that } \forall n \geq n_0, f(n) > c \cdot g(n)\}$
5. $\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 \text{ such that } \forall n \geq n_0, 0 < c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$

Definition	Comparison notation	Limit definition
$f(n) \in \mathcal{O}(g(n))$	$f(n) \leq c \cdot g(n)$	$0 \leq \lim_{n\to\infty} \sup \left\lvert \frac{f(n)}{g(n)} \right\rvert < \infty$
$f(n) \in \Omega(g(n))$	$f(n) \geq c \cdot g(n)$	$0 < \lim_{n\to\infty} \inf \left\lvert \frac{f(n)}{g(n)} \right\rvert \leq \infty$
$f(n) \in \Theta(g(n))$	$f(n) \approx c \cdot g(n)$	$0 < \lim_{n\to\infty} \inf \left\lvert \frac{f(n)}{g(n)} \right\rvert \leq \lim_{n\to\infty} \sup \left\lvert \frac{f(n)}{g(n)} \right\rvert < \infty$
$f(n) \in o(g(n))$	$f(n) \ll c \cdot g(n)$	$\lim_{n\to\infty} \left\lvert \frac{f(n)}{g(n)} \right\rvert = 0$
$f(n) \in \omega(g(n))$	$f(n) \gg c \cdot g(n)$	$\lim_{n\to\infty} \left\lvert \frac{f(n)}{g(n)} \right\rvert = \infty$
$f(n) \sim g(n)$	$f(n) = c \cdot g(n)$	$\lim_{n\to\infty} \left\lvert \frac{f(n)}{g(n)} \right\rvert = c$

Table 1.1: Some interpretations.

Therefore:

- $f(n) = O(g(n)) \leftrightarrow g(n) = \Omega(f(n))$
- $f(n) = o(g(n)) \leftrightarrow g(n) = \omega(f(n))$

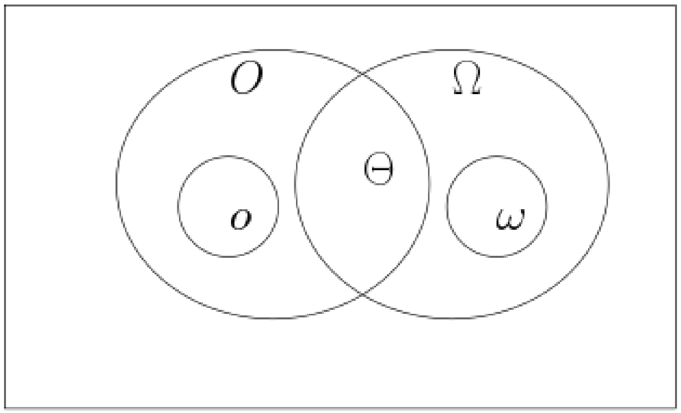

Figure 1.1: Set relations between symbols.

For $f(n), g(n)$, it may be the case that neither $f(n) \in O(g(n))$ nor $f(n) \in w(g(n))$ holds.

Q. How to interpret equations with asymptotic notations ?

1. when the asymptotic notation stands alone (that is, not within a larger formula) on the right-hand

side of an equation (or inequality), as in $n = O(n^2)$, we have already defined defined the equal sign to mean set membership: $n \in O(n^2)$

2. when asymptotic notation appears in a formula, we interpret it as standing for some anonymous function that we do not care to name. For example, $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that $2n^2 + 3n + 1 = 2n^2 + f(n)$, where $f(n)$ is some function in the set $\Theta(n)$. The number of anonymous functions in an expression is understood to be equal to the number of times the asymptotic notation appears. For example, $\Sigma_{i=1}^{n} O(i)$ is *not* the same as $O(1) + O(2) + \cdots + O(n)$, which doesn't really have a clean interpretation.
3. when asymptotic notation appears on the left-hand side of an equation, as in $2n^2 + \Theta(n) = \Theta(n^2)$. We interpret such equations using the following rule: *No matter how the anonymous functions are chosen on the left of the equal sign, there is a way to choose the anoymous functions on the right of the equal sign to make the equation valid.*

In short, consider asymptotic notations as sets and interpret the equality as $\in$ if no asymptotic notation on the left-hand side or $\subseteq$ otherwise.

3.2 Meaning of running time

Let T denote the running time of an algorithm, and n denote the input size, with $n \in \mathbb{N}$.
When performing algorithm analysis, we are interested in the following cases;

Worst case Time taken if the worst possible thing happens. $T(n)$ is the maximum time taken over all inputs of size n.

Average case The expected running time, given some probability distribution on the inputs (usually uniform). $T(n)$ is the average time taken over all inputs of size n.

Best case Time taken if the best possible thing happens. $T(n)$ is the best time taken over all inputs of size n.

Probabilistic The expected running time for a random input. Expresses the running time and the probability of getting it.

Amortized The running time for a series of executions, divided by the number of executions.

When we say that the *running time* (no modifier) of an algorithm is $\Omega(g(n))$, we mean the best-case running time. In contrast, *running time* (no modifier) of $O(g(n))$ means worst-case running time. *e.g.* The running time of insertion sort is $\Omega(n)$ and $O(n^2)$.

4 Sorting

Quick-Sort:

- `https://www.cs.auckland.ac.nz/compsci220s1c/lectures/2016S1C/CS220-Lecture10.pdf`
- randomized algorithm including quick-sort: `https://www.cs.cmu.edu/afs/cs/academic/class/15210-s15/www/lectures/random-notes.pdf`

5 Solving recurrence

There are three ways:

1. subsitution method

2. recursion-tree method
3. master method

5.1 Technicalities in recurrences

Technically, the worst-case runnint time of MERGE-SORT is:

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

When we state and solve recurrences, we often omit floors, ceillings and boundary conditions. We forge ahead without these details and later determine whether or not they matter. For example, we normally state the above recurrences as:

$$T(n) = 2T(n/2) + \Theta(n)$$

5.2 The master theorem

Theorem 5.1 (Master theorem). *Let $a \geq q$ and $b > 1$ be contants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonegative integers by the recurrence*

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. *If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$. then $T(n) = O(n^{\log_b a})$*
2. *If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = O(n^{\log_b a} \log n)$*
3. *If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$*

Chapter 2

Data Structures

1 Disjoint sets: unifon-find

We want to support two operations:

- $Find(x)$
- $Union(x, y)$

Simple implementation using linked-lists $Find$ takes $O(1)$ time, but $Union$ takes $O(n)$ time. Using weighted-union heuristic, $Union$ takes $O(\log n)$ in average. That is, for a sequence of m operations, in which there are at most $n-1$ $Union$ operations, the total time is $O(m+n^2)$ and $O(m+n\log n)$, respectively. Note that we can obtain $O(1)Union$ and $O(n)Find$ easily. But generally there are more $Find$ operations than $Union$ operations.

Using rooted trees Simple rooted trees gives $O(1)Union$ and $O(n)Find$. There are two heuristics:

- Union by rank. Here "rank" can be height, size
- Path compression.

The time costs become:

- $O(\log n)$ $Find$, if "Union by rank" used.
- $O(n + f\log_{1+f/n} n)$ for a sequence of operations with f $Find$s, if "Path compression" used.
- $O(m\alpha(n))$ for a sequence of m operations, if both used.

2 Hash Table

Hash table is used to maintain a dynamic set (key-value pairs) supporting dictionary operations.
Direct-address table:

- makes effective use of our ability to examine an arbitrary position in an array in $O(1)$ time
- works well when the universe U is reasonably small (and keys are integers)
- $O(1)$ worst-case time

Hash table $h : U \to \{0, \ldots, m-1\}$: impossible to avoid collision since $m < |U|$

2.1 Chaining

Hashing with chaining (hashing time is $O(1)$):

- $O(1)$ insertion time if it assumes the item is not already present (insert at the head of list)
- $O(1)$ deletion time if the input is a pointer in the list, rather than a key of an item
- searching time is $O(L)$ where L is the maximum length of lists. (worst-case time $O(n)$)

Define *load factor* $\alpha = n/m$, which is the average length of lists. The average-case performance depends on how well the hash function distributes the n keys into m slots.
Simple uniform hashing: any given key is equally likely to hash into any of the m slots, independently of where any other key has hashed to.
Under simple uniform hashing assumption, the average-case time is $\Theta(1 + \alpha)$, which is $O(1)$ if $m = \Omega(n)$.

2.2 Hashing function

Unfortunately, we typically have no way to check this condition, since we rarely know the probability distribution from which the keys are drawn. Moreover, the keys might not be drawn independently.

- In practice, we can often employ heuristic techniques to create a hash function that performs well.
- A good approach derives the hash value in a way that we expect to be independent of any patterns that might exist in the data.
- Finally, we note that some applications of hash functions might require stronger properties than are provided by simple uniform hashing.

The division method: $h(k) = k \bmod m$. A prime not too close to an exact power of 2 is often a good choice for m.
The multiplication method: $h(k) = \lfloor (kA) \bmod 1 \rfloor$ where "kA mod 1" means the fractional part of kA. An advantage of the multiplication method is that the value of m is not critical. We typically choose it to be a power of 2 ($m = 2^p$ for some integer p), since we can then easily implement the function on most computers.

2.3 Universal Hashing

Any fixed hash function is vulnerable to a malicious adversary. The only effective way to improve the situation is to choose the hash function randomly in a way that is independent of the keys that are actually going to be stored.
Universal hashing:

- first select the hash function at random from a carefully designed class of functions.
- fix this hash function and use it to do hashing.

Let $\mathcal{H}$ be a finite collection of hash functions that map a given universe U of keys into the range $\{ 0, \ldots, m-1 \}$. Such a collection is called *universal* if for each pair of keys $k, l \in U$, the number of hash functions $h \in \mathcal{H}$ for wich $h(k) = h(l)$ is at most $|\mathcal{H}|/m$. It means the chance of collision between l and k is no more than $1/m$, which is the chance of collision if $h(k)$ and $h(l)$ are chosen randomly and independently from $\{ 0, \ldots, m-1 \}$

Theorem 2.1. *Using universal hashing,* $E[L] \leq 1 + \alpha$.

Let p be a prime large enough so that every possible key $k \in \{ 0, \ldots, p-1 \}$. Let $h_{ab} = ((ak + b) \bmod p) \bmod m$, then $\mathcal{H}_{pm} = \{ h_{ab} : a \in \mathrm{Z}_p^*, b \in \mathrm{Z}_p \}$ is universal.

2.4 Open addressing

In open addressing, each table entry contains either an element of the dynamic set or NIL. (α can never exceed 1)
$h : U \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$, the *probe sequence* is required to be a permutation of $\{0, \ldots, m-1\}$.
We assume uniform hashing: the probe sequence of each key is equally likely to be any of the $m!$ permutations of $< 0, \ldots, m >$. Uniform hashing generalizes the notion of simple uniform hashing defined earlier to a hash function that produces not just a single number, but a whole probe sequence. True uniform hashing is difficult to implement, however, and in practice suitable approximations (such as double hashing, defined below) are used:

- Linear probing: $h(k, i) = (h'(k) + i) \bmod m$
- quadratic probing: $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
- double probing: $h(k, i) = (h_1(k) + i h_2(k)) \bmod m$

Assuming universal hashing, the time complexity (number of probs) is:

- unsuccessful search $1/(1-\alpha)$ expected time, successful search $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ expected time.
- insertion $1/(1-\alpha)$ expected time.

2.5 Perfect hashing

When the set of keys is static, we can achieve $O(1)$ worst-case time using hashing (it is called perfect hashing).
Two levels of universal hashing:

- the first level is the same as hashing with chaining
- the second level use a carefully chosen hash function h_j for slot j instead of using a list.

We need to let the size m_j of the hash table S_j to be the square of the number n_j of keys hashing into slot j.

Theorem 2.2. *Suppose that we store n keys in a hash table of size $m = n^2$ using a hash function h randomly chosen from a universal class of hash functions. Then, the probability is less than $1/2$ that there are any collisions.*

Given the set K of n keys to be hashed (remember that K is static), it is thus easyto find a collision-free hash function h with a few random trials.

Theorem 2.3. *Suppose that we store n keys in a hash table of size $m = n$ using a hash function h randomly chosen from a universal class of hash functions. Then, we have*

$$E[\sum_{j=0}^{m-1} n_j^2] < 2n$$

Again, if we test a few randomly chosen hash functions from the universal family, we will quickly find one that uses a reasonable amount of storage (e.g. $4n$).

Chapter 3

Trees

1 Preliminaries

Some conventions:

- the height of the root is 0

In a tree with n nodes V, suppose some nodes $V' \subset V$ are marked. Let u, v be the two most distant nodes that are both marked. Then for any node z, $d(u,z) \leq K, d(v,z) \leq K$ if and only if $d(w,z) \leq K$ for any $w \in V'$. Further more, u, v can be found by two DFS. https://www.geeksforgeeks.org/count-nodes-within-k-distance-from-all-nodes-in-a-set/

Traversal Preorder, inorder, level-order. Given two traversal sequence, a unique binary tree can be constructed if and only if one sequence is inorder traversal. However, for binary search tree, arbitrary traversal sequence is sufficient.
Binary trees
Full binary trees: all nodes except leaves have two children.

- the number of leaves is one more than the number of internal nodes.

Exercises

Problem 1. Given a tree T with n nodes, compute $sum[i]$ for every node i, where $sum[i]$ is the sum of distances from i to all other nodes. ($O(n)$ time, LeetCode 834)

2 Search Trees

Search trees are data structures used to efficiently searches and updates on sets of items that satisfy a total order. Possible operations includes:

- empty, singleton: create an empty tree or a tree with a single item.
- search, max, min, successor, predecessor
- insert, delete
- split(T, k): split a tree T, return (T_1, F, T_2) with all keys in T_1 less than k, all keys in T_2 greater than k, and F be true if and only if k exists.
- join(T_1, T_2): join two trees, where all keys in T_1 are less than that in T_2, takes $O(\log(|T_1| + |T_2|))$
- union, intersect, difference

Some details:

1. The set may be a *multiset*, i.e. allow the same key appear multiple times.
2. For some operations, such as successor, the input may be either a key or a node.
3. Sometimes an operation may ask for only a value, other times it ask for a pointer to the corresponding node.
4. Updating a node can be implemented by deleting it followed by inserting a node.

Full binary tree. A rooted tree where each non-leaf node has two children. The number of leaf nodes is $n_l = \frac{n+1}{2}$.
Balanced. A binary tree is defined to be *perfectly balanced* if it has the minimum possible height, i.e. $\lceil \log(n+1) \rceil$. It turns out to be impossible to maintain a perfectly balanced tree while allowing $O(\log n)$ inserting operations. Hence, we aim to keep a tree *balanced*, which means it has $O(\log n)$ height.

2.1 Binary Search Trees

The binary-search-tree property: If y is in the left subtree of x, then $y.key \leq x.key$. If y is in the right sub-tree of x, then $y.key \geq x.key$.
Each subtree comprises an interval of the list. It follows that if a node x has nonempty right subtree, then x's successor must be in the right subtree (the minimum of it).
The expected height of a randomly built binary search tree (created by insertion alone) on n distinct keys is $O(\log n)$. Let X_n be the height for n keys, $Y_n = 2^{X_n}$, $Z_{n,i} = I\{R_n = i\}$ where R_n is the rank of the root. Then $Y_n = \sum_{i=1}^{n} Z_{n,i} 2 \max \{ Y_{i-1}, Y_{n-i} \} \Rightarrow E[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} E[Y_i] \Rightarrow E[Y_n] \leq (n^3 + 6n^2 + 11n + 6)/24$.
A list of balanced (binary) search trees (i.e. root-leaf paths are of approximate length):

- AVL trees
- Red-black trees
- Weight balanced ($BB[\aleph]$) trees
- Treaps: Combine trees and heaps. Assign randomly a priority to each key and grow the tree by inserting keys one by one in the order of priorities, hence resulting in heap property with respect to those priorities.
- Splay trees
- Other binary trees including scapegoat trees, skip lists
- Non-binary trees, such as 2-3 trees, brother trees, B-trees

For balanced binary search trees, such as red-black trees, operations that follow take $O(\log n)$ if not specified. $(n = |T|, n = max(|T_1|, |T_2|), m = min(|T_1|, |T_2|))$

1. search, min, max, predecessor, successor
2. insert, delete
3. union, intersect, diff: all take $O(m \log \frac{n}{m})$
4. split, join

The *split* and *join* operations can be implemented easily using $Treaps$. A treap has $O(\log n)$ height with hight probability. To see this, just relate it to the $Quick - sort$ algorithm.
For how to split/join with treaps and other details, refer to https://www.cs.cmu.edu/afs/cs/academic/class/15210-s15/www/lectures/bst-notes.pdf

2.2 Red-black trees

Assume all nodes except NILs have two children. Then all leaves are NILs and the tree becomes a full binary tree.
red-black properties:

1. Every node is either red or black
2. The root is black
3. Every leaf (NIL) is black
4. If a node is red, both its children are black
5. For each node x, all simple paths from the node to descendant leaves contain the same number of black nodes.

For a node x, define the *black height* $bh(x)$ to be the number of black nodes on any such path from, but not including x, down to a leaf. We define the black of a tree to be the black height of its root.

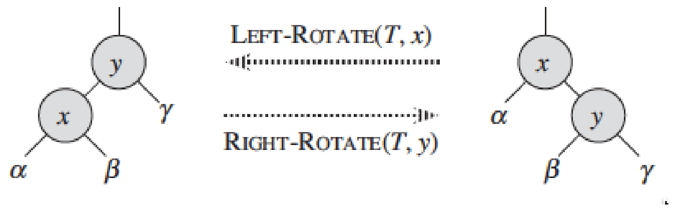

Figure 3.1: The rotations on a binary search tree.

The insertion and deletion takes at most 2 and 3 rotations, respectively.

2.3 B-Trees

In a B-tree:

1. each node v has $t-1 \le n_v \le 2t-1$ keys $k_1, k_2, \dots, k_{n_v}$, except that the root has $1 \le n_v \le 2t-1$ keys, where $t >= 2$
2. each non-leaf node v has $n_v + 1$ children $c_1, c_2, \dots, c_{n_v+1}$. All keys in c_i are less than k_i, which is less than all keys in c_{i+1}, for all $1 \le i \le n_v$
3. each leaf-node has the same height

If the number of nodes and keys are N, n respectively. Then $t^h \le N \le \frac{(2t)^{h+1}-1}{2}$, $(t^h - 1)(t-1) + 1 \le n \le (2t-1)\frac{(2t)^{h+1}-1}{2}$, hence the height is $O(\log_t n)$

Operations including search, insertion, deletion take $O(h)$ disk access time and $O(th)$ cpu time.
Red-black trees can be regarded as $2-3-4$ trees by contracting red children into corresponding black nodes, and vice versa. Hence we can mimic the operations on a black-red tree as that on a $2-3-4$ tree. For example, the insert, delete, split operations are hard to describe for red-black trees, but are easy for B-trees (in particular, $2-3-4$ trees).

Q. How to split a B-tree T by key k ?

1. Find the node v_0 containing k. If it doesn't exist, let v_0 be the leaf node that is visited lastly. From v_0, clime up along the edges pointing to parents, until reaching a node whose parent edge is the leftmost or the rightmost one in its level, or reaching the root r. Let the stopping node be v_s.
2. Cut through the path, splitting each of $v_i, (0 \leq i \leq s)$ into two (possible empty) nodes. Each edge in the path is split into two copies connecting the corresponding "split" nodes. The result is two trees T_1, T_2 which satisfy all the properties of b-trees except that $v_0, \ldots, v_s$ has less than $t-1$ keys. It's easy to see v_s and r are the roots of T_1, T_2.
3. Here we only talk about how to remedy T_1 all keys of which are less than k. Suppose the depth of v_0 in T_1 is $p, (p \geq s)$, then its root is v_p. Note that each node v_i with n_i keys has $n_i + 1$ children. Iterate through $i = 0, \ldots, p$:

   (a) if $n_i \geq t-1$, continue.
   (b) It's not hard to see that $v_i, (i < p)$ must has at lest one left sibling by the definition of v_p. Let u_i be the nearest one and m be the common parent key between u_i and v_i. If u_i has enough keys to lend to v_i, we move the rightmost $t - 1 - n_i$ keys to v_i, along with corresponding children. (Actually, those keys are shifted from left through m to v_i's left side.)
   (c) Otherwise, we merge u_i, m, v_i into a new v_i, resulting in removing a key from v_{i-1}.
   (d) Note that when reaching $i = p$, we only need to check whether v_p is empty, since it can has as least as one key.

Q. How to join two trees T_1, T_2, where all keys in T_1 are less than that of T_2 ?

Hint: merge the right-most branch of T_1 and the left-most branch of T_2 such that the resulting tree has all leaves in the same level. Suppose the two branches are $u_0, \ldots, u_p$ and $v_0, \ldots, v_q$ respectively. W.l.o.g, assume $p \geq q$. Then we merge u_{p-q+i} with v_i for $i = 0, \ldots, q$. What remains is to remedy those merged nodes.

3 Euler Tour Trees

Using any balanced BST, the following operations can be done in $O(\log n)$ time:

1. reroot
2. cut
3. link

4 Segment trees

Segment Tree is used in cases where there are multiple range queries on array and modifications of elements of the same array. E.g. the Range Minimum Queries and Range Sum Queries problems involving modifications (DRMQ, DRSQ).
Segment tree is basically a binary tree, in which each node represents an interval. Consider an array $A[n]$, the segment tree can be constructed recursively:

- the root represents the whole array $A[0 : n-1]$
- each non-leaf node (say $A[l, r], (l < r)$) has two children representing $A[l, \frac{l+r}{2}], A[\frac{l+r}{2}, r]$
- each leaf node represents a single element.

The height and size of the segment tree are $O(\log n)$ and $O(n)$, respectively. For a given problem, we use

$< Q, U >$ to denote its query/update time complexity. Then both the DRMQ and DRSQ problems have $< O(\log n), O(\log n) >$ complexity.

Bibliography

[1] Stéphane Boucheron, Gábor Lugosi, and Olivier Bousquet. Concentration inequalities. In *Summer School on Machine Learning*, pages 208–240. Springer, 2003.