Amortized Analysis

Def: DS supporting operations $o_1, \ldots, o_k$. Amortized time of o_i is $t_i(n)$ if any valid sequence of n operations, in which o_i occurs n_i times, takes total time $\mathcal{O}(\sum_j n_j t_j(n))$.

Accounting method: Charge $t_j(n)$ for operation o_j. If $t_j(n) >$ cost: distribute the remainder to accounts. If $t_j(n) <$ cost: charge some accounts. Prove non-negative balance at all times.

Potential method: D_i is the DS after i-th operation, $\Phi(D_i)$ is the DS potential (eg. sum of all accounts). Real cost c_i, amortized $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$. Total cost $\sum_i \hat{c}_i = \sum_i c_i + \Phi(D_n) - \Phi(D_0)$. If $\Phi(D_n) \geq \Phi(D_0)$ then $\hat{c}$ is upper bound on true cost. Usually $\Phi(D_0) = 0$, then need $\Phi(D_i) \geq 0$.

RMQ and LCA

Trivial: LCA: no preprocessing, $\mathcal{O}(n)$ query; $\mathcal{O}(n^3)$ preprocessing, $\mathcal{O}(n^2)$ memory, $\mathcal{O}(1)$ query; RMQ: ① $\mathcal{O}(n^2)/\mathcal{O}(1)$.

RMQ→LCA: Cartesian tree: root is minimum $A[i]$, left subtree is cartesian tree of $A[< i]$, right is CT of $A[> i]$ (min-heap, in-order traversal is A). Linear time construction: process A left to right, walk up the right spine of the tree.

LCA→ ±1RMQ: Euler tour (DFS) to get depth array, store index of first visit for each node. $\mathcal{O}(n)$ preprocessing.

RMQ in $\mathcal{O}(n \log n)/\mathcal{O}(1)$ ② (BIG from HW) Store answer for all intervals with length 2^k. Combine two to answer query.

±1RMQ: Split to ${}^{2n}/_{\log n}$ groups of size $n' = {}^1/_2 \log n$. Store minimum from each group, use ②. Since ±1, only $2^{n'} = \sqrt{n}$ different block types (min pos), use ①: $\mathcal{O}(\log^2 n)$ different queries for each type, $\mathcal{O}(\log \log n)$ bits for answer, total $\mathcal{O}(n)$ preprocess. For final answer compare three candidate positions.

Segment trees

Root: interval $[0, n)$, **Leafs:** intervals $[i, i+1)$

Node $[i, j)$ has children $[i, k)$ and $[k, j)$ where $k = \lfloor \frac{i+j}{2} \rfloor$, stores $A[i] \circ A[i+1] \circ \cdots \circ A[j-1]$. Total $2n - 1$ nodes, height $\lceil \log n \rceil$.

Query: $A[x] \circ \cdots \circ A[y-1]$ ie $[x, y)$, canonical decomposition, $\mathcal{O}(\log n)$ time.

Canon. decomp: node $[i, j)$, invariant: overlaps $[x, y)$. If $[i, j) \subseteq [x, y)$ return $\{[i, j)\}$. Else $R := \emptyset$, if $[i, k)$ overlaps $[x, y)$ recurse left, add result to R. Same for right, then return R.

Inverted Index (Full-Text Keyword Search)

For N words (n distinct) of max length m with alphabet size σ and result count p:

Index storage	*Query*	*Preprocessing*
Sorted array	$\mathcal{O}(m \log n + p)$	$\mathcal{O}(mN \log N)$
BST (balanced)	$\mathcal{O}(m \log n + p)$	$\mathcal{O}(mN \log n)$
Hash (avg/worst)	$\mathcal{O}(m + p)/\mathcal{O}(mn + p)$	$\mathcal{O}(mN)/\mathcal{O}(mNn)$
Trie	$\mathcal{O}(m \log \sigma + p)$	$\mathcal{O}(mn \log \sigma)$

External Memory

Model: infinite slow disk, fast memory with M words; transfer between them in blocks of B words; count number of transfers.

B-tree: parameter T; node v has $v.n$ keys and 0 or $v.n - 1$ children; keys are sorted, children contain only values between two keys; all leaves same depth; every node except root $T - 1 \leq v.n \leq 2T - 1$, root has $1 \leq v.n \leq 2T - 1$. Set T so each node fits within $\mathcal{O}(1)$ blocks. Height/time for all operations is $\mathcal{O}(\log_{B+1} n)$.

Cache oblivious: don't know B, M; block transfers implicit.

vEB static tree: *van Emde Boas* memory layout – cut tree in middle height; get top of $\approx \sqrt{n}$ nodes and $\approx \sqrt{n}$ bottom subtrees each with $\approx \sqrt{n}$; store all recursively.

For analysis: look at *level of detail* in above splitting where each subtree fits within B – will have height $\geq {}^1/_2 \log B$. If $M \geq 2B$ then need $\mathcal{O}(\log_{B+1} n)$ transfers.

Full-Text Keyword Search

Problem: preprocess static set of documents (sequences of words) to answer queries: given word w list all documents containing it.

Inverted index: store all words along with list of document IDs; see $\boxed{\text{inverted index}}$ for storage options.

Trie: tree, edges labeled with letters, path from root is a string; store document IDs in nodes where word ends. For $|w| = m$, alphabet size σ, total word length D:

Node edge storage	*Search*	*Insert*	*Space*
Array of size σ	$\mathcal{O}(m)$	$\mathcal{O}(m\sigma)$	$\mathcal{O}(D\sigma)$
Sorted array	$\mathcal{O}(m \log \sigma)$	$\mathcal{O}(m \log \sigma + \sigma)$	$\mathcal{O}(D)$
BST	$\mathcal{O}(m \log \sigma)$	$\mathcal{O}(m \log \sigma)$	$\mathcal{O}(D)$

Fibonacci heap

Structure: list of rooted trees, node degree $\leq D(n) = \mathcal{O}(\log n)$, sibling pointers, non-root node marked iff lost child since getting father, $\Phi = \#\text{roots} + 2\#\text{marked nodes}$

Lazy: $\mathcal{O}(1)$, *Insert:* new root, *Min:* look, *Union:* join root list

ExtrMin: $\mathcal{O}(\log n)$ am., remove min root, child to root list, consolidate - join trees until each different degree

DecrKey(x): $\mathcal{O}(1)$ am., decrease, if violates heap cond. cut x with subtree to root list, cascading cut - try to mark $p(x)$, if alredy marked cut him to root, cascade again, rec. call cost 0 - pays with Φ change, everything else $\mathcal{O}(1)$ am.

Delete: $\mathcal{O}(\log n)$ am., decrease key to -inf, extract min

Analysis: Prove $D(n) = \mathcal{O}(\log n)$: **1)** $\forall k \geq 2 : F_{k+2} = 1 + \sum_{i=0}^{k} F_i, F_{k+2} \geq \phi^k$, **2)** x: children $y_1, \ldots, y_k$ in link order, $j \geq 2 : y_j.deg \geq j - 2$, **3)** x deg. k, size of subtree $\geq F_{k+2}$ (induction on smallest deg k tree).

Sorted List Intersection (2 Keyword Search)

Output: intersection of two sorted lists, lengths $m \leq n$.

Merging: iterate over both at once: $\mathcal{O}(m + n)$

Binary search: iterate over smaller, search through the larger one: $\mathcal{O}(m \log n)$

Doubling search: like binary search but $\mathcal{O}(\log i)$ (where i is the result) instead of $\mathcal{O}(\log n)$ – double search interval until it overshoots, then binary search; $\mathcal{O}(m \log {}^n/_m)$

Perfect Hashing

Top level: universal hash function to table of size $\Theta(N)$

Second level: bucket i with c_i elements hashed to a table of size αc_i^2. Expected # of collisions: $\sum_{\substack{u,v \in X_i \\ u \neq v}} Pr[h(u) = h(v)] \leq c_i^2 \frac{c}{m_i} = \frac{c}{\alpha} < {}^1/_2$ for some c, α

Search: $\mathcal{O}(1)$ **Space:** $\mathcal{O}(n)$ (deterministic)

Expected preprocessing time: $\mathcal{O}(n)$

Bloom Filters

Representation: bit string $B[0, \ldots, m-1]$ and k hash functions $h_i : U \to \{0, \ldots, m-1\}$

Insert(x): set bits $B[h_1(x)], \ldots, B[h_k(x)]$ to 1

Contains(x): check if $B[h_1(x)], \ldots, B[h_k(x)]$ are all 1

- if yes, claim x is in the set, *possibility of error*
- otherwise answer no, *surely true*

False positive: if all h_i are totally random and independent, the probability of error is at most $(1 - e^{-nk/m})^k$

Meldable heaps

Union(H_1, H_2): $\mathcal{O}(\log n)$, WLOG $H_1.key \leq H_2.key$

Insert: union with node; **ExtrMin:** union of root subtrees; **DecrKey:** delete, insert; **Delete:** cut, union subtrees, rebalance

Random: H_2 union with $H_1.left$ or $H_1.right$ at random, expected random root-nullptr walk length $\mathcal{O}(\log n)$

Leftist: $HN = \text{Union}(H_1.right, H_2)$, if $HN.s \leq H_1.s$ put as right child of H_1 otherwise left; $s(x)$ - dist. from x to nearest nullptr, $s(x) \leq \log(n+1)$, $x \to$ nullptr path only to right has lenght $s(x)$; $\mathcal{O}(\log n)$ worst case

Skew: $\text{Union}(H_1.right, H_2)$, put as left child of H_1; $D(x) = |\text{subtree rooted at } x|$; edge $(v, p(v))$ heavy - $D(v) > D(p(v))/2$; root path $\leq \log n$ light edges, $\Phi = \#$ heavy right edges; cut children before recursion, add after, cost only when $H_1.r$ was light before cut; $\mathcal{O}(\log n)$ amortized

2-INF-237 VPDS - Unofficial Cheatsheet

Splay trees

BST w/o balance info, am. $\mathcal{O}(\log n)$ ops., move el. to root after access, decr. overall height
Splay(x): zig/zag rotations until x root, real cost: # rotations, $D(x)$ size of x subtree, rank $r(x) = \log D(x)$, $\Phi(T) = \sum_{x \in T} r(x)$, cost $\mathcal{O}(\log n)$ am.
Search, Delete, Insert(x): as in BST then splay, insert increases rank on root-x path, at most $\mathcal{O}(\log n)$, **fMin(x):** splay(x), splay leftmost, ret. it, **join(v_1, v_2):** keys $v_1 \leq v_2$, m=fMin(v_2), splay(v_1), v_1 as left child of m, **SplitBefore/After(x):** splay(x), cut one child

Link-cut trees

Path: splay tree, key - pos. on path (not stored), findHead, split, join same
L-C: forest, each node ≤ 1 solid edge, other dashed; tree as coll. of solid paths, connected by dashed parent ptrs
Expose(x): make x lower end of solid path to root, $\mathcal{O}(\log^2 n)$ am., splitBelow(x), jump to solid path root, splice root, repeat; **Splice(x):** splitBelow($p(v)$), make that edge dashed, join x and $p(x)$
FindRoot: expose, findHead, **Link(v, w):** v root, make v child of w; expose both, join paths, **Cut(v):** v not root, cut edge v to parent; expose, splitAbove

Search for pattern $|P| = m$ in string $|T| = n$

Triv. alg: $\mathcal{O}(nm)$; **NFA:** simulate in $\mathcal{O}(m\sigma)$ steps
DFA: NFA $\rightarrow$ DFA ($m+1$ states). Build time $\mathcal{O}(m\sigma)$, Simulation $\mathcal{O}(n)$
MP: NFA, add eps transitions i to $j = sp[i]$ ($j < i$) when P[0, ..., j-1] is the longest suffix of P[0, ..., i-1]
Build, memory $\mathcal{O}(m)$, simulation $\mathcal{O}(n)$
KMP: $sp2[i]$ = first transition on eps chain which has diff symbol.

Succint DS – Rank

Lower bound: $OPT = \lg |\mathcal{U}|$ bits to store any $x \in \mathcal{U}$
Implicit: $OPT + \mathcal{O}(1)$ **Succint:** $OPT + o(OPT)$ **Compact:** $\mathcal{O}(OPT)$ bits.
Rank (and select): n bit vector; rank and select in $\mathcal{O}(1)$; size $n + o(n)$ bits; $rank(i)$ = # of 1s in $A[0, \ldots, i]$
Superblocks size $t_1 = \log^2 n$; keep global rank at boundary, $\mathcal{O}(\frac{n}{t_1} \log n) = o(n)$; *Blocks* size $t_2 = \frac{1}{2} \log n$, keep rank in superblock at boundary, $\mathcal{O}(\frac{n}{t_2} \log t_1) = o(n)$.
Store blocks as t_2-bit integers $= n$ bits.
Block size t_2, all possible blocks 2^{t_2}, total memory $\mathcal{O}(2^{t_2} t_2 \log t_2) = \mathcal{O}(\sqrt{n} \log n \log \log n) = o(n)$ bits

Approximate string matching $\mathcal{O}(nm)$

Edit ops ($u, v \in \Sigma^*$, $a, b \in \Sigma$): insrt ($uv \rightarrow uav$), del ($uav \rightarrow uv$), subs ($uav \rightarrow ubv$)
Edit dist $d_E(S, T)$ = shortest sequence of edit operations that changes S to T. DP algorithm for $d_E(S, T)$: $A[i, j] = d_E(S[1 \ldots i], T[1 \ldots j])$; set $A[0, i] = A[i, 0] = i$; then $A[i, j] = min\{A[i-1, j-1] + [S[i] \neq T[j]], A[i-1, j] + 1, A[i, j-1] + 1\}$

Geometric Data Structures (d-D)

Planar point location: given planar map (straight edges, no crossings); *query:* which face contains given point
Vertical ray shooting: given point find the edge a vertical ray hits first; solves PPL (pointer from edge to face below)
Line sweep: move through interesting x coordinates, store segments in BST in y order; use *partial persistent* BST for (static) point (x, y) query – find successor of y in BST at time x; $\mathcal{O}(\log n)$. Use *fully retroactive* BST for dynamic (can add/delete edges, *horizontal segments only*) $\mathcal{O}(\log n)$.
Orthogonal range searching: given points in d-D space and a box, report existence/count/list of points inside the box.
Range tree: for *1D* store points in balanced BST. Answer is all leaves between the interval (1D box) successor/predecessor.
For *2D* store by x coordinate same way, for each internal node store pointer to BST storing same subtree by y; query: $\mathcal{O}(\log^2 n)$
Same for d-D: space $\mathcal{O}(n \log^{d-1} n)$; query $\mathcal{O}(\log^d n)$
Layered RT: 2D: store sorted by y in each node as sorted array; also store for each element its rank in left/right child's array. Search by y once in root. Then search by x in the BST, update y array range in $\mathcal{O}(1)$. Query: $\mathcal{O}(\log n)$
For d-D: use 2D as basecase for RT, query: $\mathcal{O}(\log^{d-1} n)$.

Succint DS - Wavelet and RRR

Wavelet tree: recursively split Σ to $\Sigma_{0,1}$, then store binary vector $B[i] = j$ for $S[i] \in \Sigma_j$. Recursively until both alphabet partitions are only single character. "Tree" depth and time for *rank* or *select* is $\mathcal{O}(\log |\Sigma|)$.
RRR: Similar to rank, instead of storing block as t_2-bit integer store *class* (# of 1s in block) and *signature* (lexi order of this block among blocks with that # of 1s). For many 1s or many 0s this will require fewer bits total.
Entropy: $H(S) = \sum_{a \in \Sigma} \frac{n_a}{n} \lg \frac{n}{n_a} \leq \lg |\Sigma|$ where a occurs n_a times. Total space $nH(S) + o(n)$.

Succint binary tree

Create tree. Instead of NULL pointers add fake nodes. Number verticies in BFS order. Create array A[i] = 1 $\Leftrightarrow$ vertex i is real. Left(i) = $2 \cdot rank(i)$. Right(i) = $1 + Left(i)$. Parent(i) = $select(i//2)$. *Proof:* induction on i, children of node i-1 preceed childr. of i

Burrows-Wheeler Transform

Construct: sort all rotations of string lexicographically; out: last column L.
Reverse: sort lex. to obtain first column F. $F[i]$ follows $L[i]$. j-th occurrence of x in F is the same as j-th occurrence of x in L. $LF[i]$ = row j in which $F[j]$ corresponds to $L[i]$
alg: $T[n] = \$$, $s = 0$. $i \in (n, 0] : T[i] = L[s]$; $s = LF[s]$
C[x]: the index of first occurance of x in F.
rank[x,i]: the number of occurances of x in $L[0 \ldots i]$
LF[i]: $C[L[i]] + rank[L[i], i-1]$
Counting occurrences of P (FM index) search for string backwards using LF transformation
$l = 0, r = n$ $i \in (m, 0]:$ $a = P[i]$, $l = C[a] + rank[a, l-1]$, $r = C[a] + rank[a, r] - 1$, if $l > r$ return 0. output: $r - l + 1$.

Suffix Trees and Suffix Arrays

Repr.: Trie of suffixes of *T\$*, deg. 2 nodes merged.
Generalized: More strings, in leaves store source index.
Max. repeat: substr. (node) $T[i \ldots i+k]$ is max. repeat $\Leftrightarrow$ $\exists j : T[i \ldots i+k] = T[j \ldots j+k]$, but $T[i-1] \neq T[j-1] \wedge T[i+k+1] \neq T[j+k+1]$. Max. repeat $\Leftrightarrow$ is diverse
Apx. match: Insert T and P in gen. tree, preproc. LCA in $O(|T| + |P|)$, LCA is LCP $\Rightarrow$ for each suffix in T jump $\#err = k$ times using LCA $\Rightarrow O(nk)$
Printing doc.: Append distinct $\$_i$ for each doc., create suffix tree. DFS: Nodes corresp. to intervals. In leaves $\$_i$ store pos. of prev. $\$_i$. #docs matching node $w = interval(i, j)$ in tree - #leaves in $(i, j) < i$ (RMQ).
Tree→array: DFS, children in asc. order.
Array→tree: Build $L[i] = LCP[i, i+1]$, this is depth of node. Create Cartesian tree on L: find all mins, recurse on intervals before, between, after the mins. DFS traverse, add leaves to each node with value from array in order.
Search in array: $O(m + logn)$, $LCP(i, j) = lcp(T[SA[i] \ldots n-1], T[SA[j] \ldots n-1)$. As binary search, let L, R boundaries, $k = \frac{L+R}{2}$, $XL = LCP(X, L), XR = LCP(X, R)$, if $XL \geq XR$:

- if $LCP(L, k) > XL \Rightarrow L := k$
- if $LCP(L, k) < XL \Rightarrow R := k, XR := LCP(L, k)$
- if $LCP(L, k) = XL \Rightarrow$ compare at XL, move accordingly

Compute LCP lemma: $L[i] = LCP(i, i+1)$. if $SA[x+1]+1 = SA[y+1] \Rightarrow L[y] \geq L[x] - 1$. $\boxed{\text{LCP on SA}}$
Create array: Add enough zeroes to the end, create groups of 3 letters for every pos. $\Rightarrow$ create 3 sets: S_0, S_1, S_2, Recursively create suffix array for $concat(S_1, S_2) = SA_{1,2}$. Compute *rank* for $SA_{1,2}$. For S_0, represent $S[3i \ldots n]$ as $(S[3i], rank[3i+1])$, get suffix array S_0 by radix sort. Merge SA_0 and $SA_{1,2}$, comparing either $(S[i], rank[i+1])$ for $i\%3 == 1$ or $(S[i], S[i+1], rank[i+2])$ for $i\%3 == 2$.

Persistent Data Structures

Partial Persistence: Update latest version ⇒ versions linearly ordered
Full Persistence: Update any version (branch) ⇒ versions form tree
Confluent Persistence: Can combine two versions → new versions ⇒ versions form DAG
Functional: Never modify nodes. Only make new nodes.
Backtracking: Query and update only current version, revert to an older version
Retroactive: insert updates to the past, delete past updates, query at any past time relative to current set of updates
General transformation with fat nodes: Add $\mathcal{O}(logn)$ factor overhead
Fat Node - binary search tree with time as keys
Each BST node holds a node of the original d.s.
Query of original node at time t: predecessor search in BST
Update of original node: insert a new maximum to BST
Arbitrary pointer machine d.s. with $f(n)$ update, $g(n)$ query
Partially persistent version with $\mathcal{O}(f(n))$ update, $\mathcal{O}(g(n))$ query, $\mathcal{O}(g(n)\log n)$ past query
General transformation with node copying: Removes $\mathcal{O}(logn)$ factor overhead, assumes original structure has in-degree $\mathcal{O}(1)$
Arbitrary pointer machine d.s. with at most $p = \mathcal{O}(1)$ incoming pointers per node and $f(n)$ update, $g(n)$ query
Partially persistent version with $\mathcal{O}(f(n))$ amortized update, $\mathcal{O}(g(n))$ query
New node: original node, p reverse pointers for current version only, $2p$ mods (version, field, value), multiple such nodes for an original node
Version: time t and original root node at time t
Read node at time t: apply all mods with version $< t$. $\mathcal{O}(1)$ overhead.
Update node: change $n.x$ from z to y
If node not full, add a new mod
Otherwise add a new node n' with latest version of n
Other nodes may have back pointers to n, change to n'
Recursively change pointes to n to point to n' in the newest version - keep pointer to n in the old version
Add back pointer from y to n'
Remove back pointer from z to n
Φ : total number of mods in the latest versions of nodes

Full Retroactivity on Decomposable Search Problems

Search problem: set S, insert, delete, query(x,S)
Decomposable search prob: query(x,$A \cup B$) = query(x,A) $*$ query(x,B). $*$ comp in $\mathcal{O}(1)$, possibly req. A, B disjoint; $x = (op, elem)$
Examples: exact set membership, nearest neighbor, predecessor
Full retro for dec. s. prob.: each operations corresponds to some time interval (b_a, e_a). (for example time of some element in Set). Build segment tree on this timeline. Each operations is in at most $\mathcal{O}(\log n)$ nodes (canon decomp).
Retro update: find interval (b_a, e_a) delete it from segment tree, add interval (b'_a, e'_a)
query(x,t): Find a leaf for predecessor of t. Search in every node on path to root. Combine result using $*$. $\mathcal{O}(\log max_{time})$ factor overhead on everything.
Example: Set membership problem. Each interval corresponds to time of some element in S. When performing query whether some element is in S in some time t, query leaf at time t. Every node on path from this leaf to root contains this interval which means they have information about elements that lived at time t. Operation $*$ is now logical or.

Landau-Vishkin Approximate String Matching $\mathcal{O}(kn)$

Approximate string matching: substrings of T with $d_E \le k$.
$A[i,j] = \min_k d_E(P[0,\ldots,i], T[k,\ldots,j])$
Diagonal number d: all values $A[i,j]$ where $j - i = d$
$L[d,e]$ maximum row i on diagonal d with $A[i, i+d] \le e$

```
for e ∈ [0,k] do
    for d ∈ [−e,n] do
        i ← min{m, max{L[d,e−1]+1, L[d−1,e−1], L[d+1,e−1]+1}}
        while i < m ∧ i+d < n ∧ P[i] = T[i+d] do
            i ← i+1
        L[d,e] ← i
        if L[d,e] = m then
            print occurrence ending at d+m
```

INT – van Emde Boas tree

Universe: Size M values $\{0,\ldots,M-1\}$
Problem: store set of items, support successor query
Split into $\sqrt{M}$ clusters of size $\sqrt{M}$ plus summary structure for the clusters.
Hierarchical coordinates: $x = \langle c, i\rangle$ where $c = \lfloor \frac{x}{\sqrt{M}} \rfloor$, $i = x \mod \sqrt{M}$ (recursive, M gets smaller)
Cluster[c] handles $\langle c, i\rangle$ for all i
Keep cluster min, *not* stored recursively. Also keep max, store recursively. **Space:** $\mathcal{O}(M)$
Time: $T(M) = T(\sqrt{M}) + \mathcal{O}(1) = \mathcal{O}(\log\log M)$
Successor: only one recursive call always.
Insert: in case of two recursive calls the first one is $\mathcal{O}(1)$ (inserting to empty only stores as *min*).

INT – x-fast trie

Structure: Make binary vector for universe, 1 iff that element is present in the set. Build binary tree over this, each node is logical OR of children.
Successor: Given pointer to some leaf binary search on the path to root for first "1" node (monotone sequence). Other child of that node (not on this leaf-to-root path) contains the predecessor or successor (left or right sibling), get min/max present element from that subtree. Store all present items in linked list to convert between predecessor and successor. $\mathcal{O}(\log\log M)$ time, $\mathcal{O}(M\log\log M)$ space
X-fast: to save space don't store tree but only paths to non-zero leaves as "0"/"1" strings (left/right branch). For each string store all prefixes in hash table. Then can binary search in the hash table. *Space:* $\mathcal{O}(n\log M)$

MP sp table

```
sp[0] = sp[1] = 0
j = 0                    ▷ invariant j = sp[i − 1]
for i ∈ [2, m] do
    while j > 0 ∧ P[i − 1] ≠ P[j] do
        follow epsilon transitions
        j ← sp[j]
    if P[i − 1] == P[j] then
        j + +
    sp[i] ← j
```

MP search

```
state ← 0
for i ∈ [0, n) do
    while state > 0 ∧ T[i] ≠ P[state] do
        state ← sp[state]
    if T[i] == P[state] then
        state + +
    if T[i] == m then
        print occurance
```

MP → KMP

```
sp2[0] = 0
for i ∈ [1, m] do
    if i == m ∨ P[sp[i]] ≠ P[i] then
        sp2[i] = sp[i]
    else
        sp2[i] ← sp2[sp[i]]
```

SPLAY – Expose(v)

```
y ← cutPathBelow(v)
if y ≠ null then
    findPathHead(y).dashed ← v
while true do
    x ← findPathHead(v)
    w ← x.dashed
    if w = null then              ▷ x is root
        break
    x.dashed ← null
    q ← splitPathBelow(w)
    if q ≠ null then
        findPathHead(q).dashed ← w
    linkPaths(w, x)
    v ← w
```

vEB – Insert(V, x = ⟨c, i⟩)

```
if V is empty then
    V.min, V.max ← x
    return
if x < V.min then swap x ↔ V.min
if x > V.max then
    V.max ← x
if V.cluser[c].min = null then
    Insert(V.summary, c)              ▷ O(1)
Insert(V.cluser[c], i)
```

vEB – Successor(V, ⟨c, i⟩)

```
if x < V.min then return V.min
if i < V.cluster[c].max then
    return
⟨c, Successor(V.cluster[c], i)⟩
c′ ← Successor(V.summary, c)
return ⟨c′, V.cluster[c′].min⟩
```

LCP on SA

```
h = 0
for i ∈ [0, n] do
    if rank[i] > 0 then
        while T[i + h] == T[k + h] do
            h + +
        L[rank[i] − 1] = h
        if h > 0 then
            h − −
```

FIB – ExtractMin

```
z ← H.min
add each child of z to root list
remove z from root list
Consolidate()
return z
```

FIB – Consolidate

```
A[0..maxdeg] ← null
for x ∈ root list do
    while A[x.deg] ≠ null do
        y ← A[x.deg]
        A[x.deg] ← null
        x ← HeapLink(H, x, y)
    A[x.deg] ← x
create root list from A
```

FIB – HeapLink(x, y)

```
WLOG x.key ≤ y.key
remove y from root list
make y child of x
x.deg ← x.deg + 1
y.mark ← false
return x
```

FIB – DecreaseKey

```
x.key ← k
y ← x.parent
if y ≠ null ∧ x.key < y.key
then
    Cut(x, y)
    CascadingCut(y)
update H.min
```

FIB – CascadingCut(y)

```
z ← y.parent
if z ≠ null then
    if y.mark = false then
        y.mark ← true
    else
        Cut(H, y, z)
        CascadingCut(z)
```