# Data Structures

**Note**: Strange structure due to finals topics. Will improve later.

## Binary trees and their balances

### AVL trees

Vertices contain -1,0,+1 – difference of *balances* between the left and the right subtree. *Balance* is just the maximum depth of the subtree.

Analysis: prove by induction that for depth $k$, there are between $F_k$ and $2^k$ vertices in every tree. Since balance only shifts by 1, it's pretty simple.

### RB trees

Vertices contain color (red and black). Chief condition: all paths to the leaves are equally black. All leaves are considered to be black, and a parent of a red vertex must be black.

## Heaps

We usually do: INSERT, DELETEMIN, MERGE, DECREASEKEY, INCREASEKEY, and DELETE.

### Binary/regular heap

Standard static heap, insert and delete by balancing it upwards. Construction in $O(n)$, lookup in $O(\log n)$.

### Binomial heap

Recursive definition of binomial trees, binomial heap is just a collection of binomial trees. Every tree must be present only once. INSERT is just adding an element and merging heaps. Merging is like adding binary numbers, so it happens fast. Adding $n$ elements again in $O(n)$.

### Leftist heap

**D:***dtl*$(x) \equiv$ minimum distance from $x$ to leaf (downwards).

Leftist rule: left son has dtl $\geq$ to the right son.

Only one tree, compared to binomial, but worse amortized complexity.

### Fibonacci heap

Vertices of any degree, main assertion: $v$ has $k$ sons means that there are at least $F_{k+2}$ vertices below $v$. Fibonacci heap $H$.

MERGE is just concatenation. FINDMIN is either lazy or not, DELETEMIN removes children, merges them together, and concatenates the result into $H$.

When doing DECREASEKEY($v$) and INCREASEKEY($v$), we also balance the supertree after removing the tree at $v$ – go up the hierarchy, and cut any marked parent out of the tree and concatenate it with $H$. Increase also rips apart the vertex $v$, concatenates children into $H$, and finally concatenates $\{v\}$ as a trivial tree.

## Tries

Compressing dictionary structure in a tree. General tries can grow based on the dictionary, and quite fast. Long paths are inefficient, so we define *compressed tries*, which are tries with no non-splitting vertices. Initialization $O(\Sigma l)$ for uncompressed tries, $O(l + \Sigma)$ for compressed (we are only making one initialization of a vertex – the last one).

**T:**Assuming all $l$-length sequences come with uniform probability and we have a sampling of size $n$, $E[\text{c} - \text{trie size}] = log_k d$.

**P:**$q_d \equiv$ probability trie has depth $d$. $E[\text{c} - \text{triesize}] = \sum_d d(q_d - q_{d+1}) = \sum_d q_d$.

Calculate opposite event – trie is within depth $d-1$. Then prefixes from $n$ must decode all words uniquely, so: $P[\text{unique}] = \frac{\binom{k^d}{n} k^{n(d-l)}}{\binom{k^l}{n}}$.

$$q_d \leq 1 - P[\text{unique}] = 1 - \prod_{i=0}^{n-1}(1 - i/k^{d-1}) \leq 1 - e - n^2/k^{d-1} \leq n^2/k^{d-1}.$$

The substitution of the product to $e$ is done through integration.

### Suffix trees

## B-trees

**General (a,b)-trees**: Every vertex has at least $a$ and at most $b$ sons. Also $a \geq 2$ and $b \geq 2a - 1$, so we can split vertices efficiently. Does not hold for root.

**O:**(2,4)-trees are RB-trees.

**B-trees** are simply $(a, b)$ trees where $(a, b)$ is $m/2$ and $m$, respectively.

Extensions: We can delay splitting, if it is inconvenient, by simply moving elements to our siblings. We can then split 2 into 3, or 3 into 4 – such trees are called $B^*$-trees.

Also we can have redundant data, prefix trees, variable length data, or finger trees.

## General hashing

Traditional operations: MEMBER, INSERT, DELETE.

**D:**Notation: Universe $U$, of size $u$. Sampling $S$ of size $s$. Hash function $h$. List size usually $l$. Domain of hash function either $U$ or $S$, depends on universality. Codomain of hash function (buckets) $M$ of size $m$. Load factor $\alpha \equiv s/m$.

## Separated chains

Assume the following:

- $h$ splits items into buckets independently and equally. Think $h(x) = x \bmod m$.
- $S$ is randomly independently chosen from $U$.

Separated chains $\equiv$ just linked lists for each bucket.

$P[\text{one list of size exactly } l] = \binom{s}{l}(1/m)^l(1 - 1/m)^{n-l}$. Since this is a binomial distribution, easily $E[\text{chain length}] = \sum(lP[\text{len} = l]) = \alpha$.

**Expected maximum list length**

$$P[\max_i \text{len}(j) \geq j) \leq \sum_i P[\text{len}(i) \geq j] \leq m\binom{s}{j}(1/m)^j$$

$$= \frac{\prod_{k=0}^{j-1}(s-k)}{j!}(1/m)^{j-1} \leq s(s/m)^{j-1}/j!.$$

Bound $s$ by nearest larger factorial $k_0!$. Then $k_0 = O(\frac{\log s}{\log \log s})$. Clearly, the smallest $j_0$ such that $s(s/m)^{j-1}/j! \leq 1$ is smaller than $k_0$.

Split the $EMS$ sum until the probability drops below one:

$$EMS = \sum_j P[\max_i \text{len}(j) \geq j] \leq j_0 + 1/j_0 = O(\frac{\log s}{\log \log s}).$$

### Number of tests

**D:**Tests $\equiv$ number of comparisons after insertion.

Expected number of tests with an unsuccessful insertion: we test all elements if the list is nonempty, otherwise just 1 test. (Since $E$ has no memory, assume $j$ is one fixed value.) In total: $1P[\text{ list empty}]s + \sum_l lP[\text{len}(j) = l] = e^{-\alpha} + \alpha$.

Expected number of succesful tests: 1+ expected list length after each insertion: $1 + n - 1/2m = 1 + \alpha/2$.

### Replacement lists

If we want to avoid allocating the memory for the separated chains, we can create chains inside the hash table itself. The simplest solution is the replacement lists, where we simply link a collision to the next free cell in the hash table, and create a linked list. This solution is rather slow for DELETE, because we have to rearrange the list.

Number of tests is however still the same.

### Two-pointer hashing

Two pointers mean that in the hash entry $j$, we store the link to the beginning of the linked list for $j$, which can begin elsewhere. INSERT and DELETE are simpler.

The number of tests is greater.

### Coalesced hashing: EISCH, LISCH

Have no shared memory, still index within the hash tables. Lists coalesce – they grow together if they collide. Early v. Late insertion.

### Coalesced hashing with external memory: VISCH, EISCH, LISCH

### Linear adding

### Double hashing

# Universal hashing

Using universal hashing, we want to simulate random choice of $S$ from $U$, which is not always true. We assume $S$ is given, we construct a set of hashing functions that will ensure uniform choice.

**D:**A system of functions is *c-universal* $\equiv$ only a few elements collide:

$$\forall x, y \in U, x \neq y : |\{i \in I; h_i(x) = h_i(y)\}| \leq \frac{c|I|}{m}.$$

**O:**There exists a $c$-universal system for any $S$.

**P:**Assume $U$ is just $1 \dots N$ for some prime. Choose functions of type $h_{a,b}(x) = (ax + b \bmod N) \bmod m$. If $h_{a,b}(x) = h_{a,b}(y)$, then the two functions have a common remainder $i$ and both $r, s$ such that $i + rm = i + sm \bmod N$. Therefore, the number of solutions is at most $|\{(i, r, s)\}| = m\lceil N/m \rceil^2$. The system is therefore universal for $c \equiv \lceil N/m \rceil^2/(N/m)^2$.

# Perfect hashing

# Sorting in external/internal memory

# Lower bounds for ordering (decision trees)

**TODO:**That old lower bound, plus some probabilistic results from PALG.

# Dynamization

One semidynamization approach needs the following condition: $f(x, A) \,\&\, f(x, B) = f(x, A \cup B)$. Some problems are not decomposable like this (convex hull).

Our approach is based on a binomial heap.

### Semidynamization

Represent dynamic structure as a list of sizes exactly $2^i$. All sizes $2^i$ are basically buckets. INSERT means inserting into the first empty bucket and concatenating with all previous buckets. MEMBER goes through all buckets. Both have time $O(XlogX)$, which may be $O(X)$ if $X = n^{\varepsilon > 0}$. INSERT has this complexity amortized.

To make INSERT with worst case $O(X \log X)$, keep more structures of the same size and "portion" the steps with batches of $P_S(i)/2^i$. The last structure of one bucket $i$ will be semi-constructed, the remaining are complete. If you finish one structure $A_i$, take two from the previous bucket and start merging a new one. Once you finish INSERT and there are $\geq 2$ finalized structures $A_i$ on the highest level, move them to the bucket $i + 1$ and finish.

### Full Dynamization

In order to implement DELETE, take a number (say 1/8) and keep buckets full by $2^i$ with lazy deletion ensuring that you start reworking it only after there is less than 1/8 undeleted elements. Once you have too few elements, try to merge the bucket with the previous bucket $A_{i-1}$, reworking both in the process.

# Self-correcting structures

**Lists.**

- MFR: move first element to front.
- TR: slowly increase element's position when accessed.
- TIMESTAMP: have timestamps, move back based on them.

**T:**Expected time of MFR $P_{MFR} \leq 2P_{OPT}$.

**P:**Proving only for MEMBER(x). Call $\beta_x$ probability that $x$ on input. Build Markov chain based on $\beta_x$. Markov chain is irreducible, aperiodic, has a stationary distribution $\gamma_\pi$. $\pi$ is a list of elements, $B$ is the ground set.

**D:**$\delta(x, y) \equiv \sum\{\gamma_\pi | \pi(x) < \pi(y)\}$.

**O:**$P_{OPT} = \sum_{i=1}^{n} i\beta_i$.

**O:**$P_{MFR} = \sum_{x \in B} \beta_x (1 + \sum_y \delta(y, x))$.

**O:**$\delta(x, y) = \frac{\beta_x}{\beta_x + \beta_y}$.

Calculate the result using the previous observations.

**Splay trees.**

Trees with no data for balancing, only ZIG, ZIG-ZIG, and ZIG-ZAG rules. Splay any element to the root.

*Static optimality:*

*Dynamic optimality:*

# Relaxed search trees

Relaxation $\equiv$ attempt at accomodating parallelism through laziness.