Advanced Data Structures

Martin Hafskjold Thoresen

Updated: July 23, 2017

Introduction

This book is a collection of course notes for the course *Advanced Data Structures* at ETH Zürich, in the spring of 2017. The chapters are arranged in the same way as the lectures, and some chapters covers material from two lectures. Most of the lectures were either loosely or firmly based off of Eric Demaines MIT course 6.851 of the same title[1].

[1] `http://courses.csail.mit.edu/6.851/spring21/`

Contents

Chapter 1

Hashing

Topics: *Universal hashing, tabulation hashing, hash tables, chaining, linear probing, cuckoo hashing.*

1.1 Motivation

Operations we need: QUERY(x), INSERT(x), DELETE(x), where $x \in [\mathcal{U}]$. $\mathcal{U}$ is called the *Universe*. We already know how to do this using Balanced Binary Search Trees (BBSTs), with $O(n)$ space and $O(\log n)$ time on all operations. We would like to have $O(1)$ time, while still having $O(n)$ space.

1.2 The Hash Function

A *Hash Function* is a function $h : [\mathcal{U}] \to [m]$ where $\mathcal{U} \gg m$. m is the table size. Ideally, h is a totally random family of hash functions, meaning there is no correlation between its input $u \in \mathcal{U}$ and $h(u)$. However, encoding a totally random function takes $O(u \log m)$ space.

This is a problem since $n = U$, which is large. Therefore we settle on a family $\mathcal{H}$ of hash functions of small size which is *Universal*. Universality means that the probability of two non-equal keys having the same hash value is $O(1/m)$:

$$\forall x \neq y \; P_{h \in \mathcal{H}}[h(x) = h(y)] = O(1/m)$$

Example: $h(x) = ((ax) \mod p) \mod m$ where $0 < a < p$, p is prime and $p > m$ is a universal hash function.

Definition — *k-independent*: A family of hash functions $\mathcal{H}$ is k-independent if

$$\forall x_1, \ldots x_k \ \Pr[\bigwedge_i h(x_i) = t_i] = O(1/m^k)$$

Example: $((ax + b) \mod p) \mod m$ is 2-independent.

Example: A polynomial of degree k, $((\sum_{i=0}^{k-1} a_i x^i) \mod p) \mod m$, is k-independent.

1.3 Simple Tabulation Hashing

Tabulation hashing is a hashing scheme. We view x as a vector of bit blocks $x_1, \ldots, x_c$, and use a totally random hash table on *each* block. Then we `xor` the blocks together to get the final value:

$$h(x) = T_1(x_1) \oplus T_2(x_2) \oplus \cdots \oplus T_c(x_c)$$

Since a block is u/c bits, the universe for one block is of size $u^{1/c}$. One table is of size $u^{1/c}$ (we assume the hash value are machine words, as in [1]), so the total space used is $O(cu^{1/c})$. The time spent is $O(c)$, since each lookup is $O(1)$. Simple tabulation hashing is 3-independent, but not 4-independent.

1.4 Chaining

Because of the birthday paradox, hashing collisions in the table are very probable[1]. Therefore we need a scheme to handle collisions, the simplest of which is *chaining*. Each buchet in the hash table is a linked list of the values mapping to that bucket.

What can we say about the length of the chain? Let C_t be the length of chain t. $\mathbb{E}[C_t] = \sum_i \Pr[h(x_i) = t]$ If we have universal hashing we know that $\Pr[h(x_i) = t] = O(1/m)$, so $\mathbb{E}[C_t] = O(1)$, for $m = \Omega(n)$

Since we need to traverse the list for all operations, the cost is quadratic in C_t, so we are interested in $\mathbb{E}[C_t^2]$:

$$\mathbb{E}[C_t^2] = 1/m \sum_{s \in [m]} E[C_s^2] = 1/m \sum_{i \neq j} \Pr[h(x_i) = h(x_j)]$$

[1] unless we know all keys up front, and $n \leq m$

If we have universal hashing this is $\frac{1}{m}n^2O(\frac{1}{m}) = O(n^2/m^2) = O(1)$ for $m = \Omega(n)$. With a totally random hash function $C_t = O(\frac{\log n}{\log\log n})$ with probability $1 - 1/n^c$ for any c. This also holds for simple tabulation hashing.

1.5 Perfect Hashing (FKS hashing)

Idea: resolve collisions with another layer of hash tables.

On collision in bucket tables, rebuild the table using a different hash function. When n gets too large, double m and start over, in order to keep the number of elements in the bucket tables small. If a bucket table is too large, double its size and rehash.

Since $\mathbb{E}[C_t]$ is the number of elements that should not collide in the table, we can adjust the table size in such a way that $\mathbb{E}[C_t] \leq 1/2$, so $\Pr[\text{no collisions in } C_t] \geq 1/2$. Use size $\Theta(C_t^2)$ for the bucket tables. This makes the expected number of rebuilds of the bucket tables $O(1)$ before getting zero collisions.

$\mathbb{E}[\text{space}] = \Theta(m + \sum_t C_t^2) = \Theta(m + n^2/m) = \Theta(n)$ for $m = \Theta(n)$.

Results: $O(1)$ deterministic query, $O(n)$ expected update (w.h.p.), $O(n)$ space.

1.6 Linear Probing

Idea: store values directly in the table. On collision, look for next available spot. On deletion, replace element with a "tombstone", so that searches does not stop too early. Great for cache performance. The main problem of linear probing is the increasing lengths of the "runs", that is intervals of non-empty cells.

We require $m \geq (1+\epsilon)n$ (not just $m = \Omega(n)$), in order to have available cells in the table. Space is naturally $O(m)$. With a totally random hashing function, or by using tabulation hashing, the expected time for operations is $O(1/\epsilon^2)$ With a $O(\log n)$-wise or 5-wise independent hashing function, constant time is expected.

1.7 Cuckoo Hashing

Idea: have two hash tables with different hashing functions. On collision, swap the colliding values, and try to insert the swapped value in the other table. Deletes are simple. If we get a cycle (swap a with b, swap b with c, and c hashes to a again), we rebuild the tables. The table sizes are $m \geq (1+\epsilon)n$, so the space used is $O((2+\epsilon)n)$.

Any value x is either in $A[h_A(x)]$ or $B[h_B(x)]$ $\implies$ $O(1)$ deterministic query. If the hashing functions are fully random or $\log n$-wise independent we get $O(1)$ expected update and $O(1/n)$ probability of failure.

Chapter 2

Static Tree Queries

Topics: *Range Minimum Query (*$\mathrm{RMQ}(i, j)$*), Lowest Common Ancestor (*$\mathrm{LCA}(x, y)$*), Level Ancestor (*$\mathrm{LA}(x, n)$*), Range Queries, Range Trees.*

We would like $O(1)$ query time on selected operations, and still only $O(n)$ space; if we allow $O(n^2)$ space this is trivial, as we can simply precompute all queries and store them.

2.1 Range Minimum Query

We would like to retrieve the index of the minimum element in an array, between index i and j. Our goal is to get $O(n)$ time and space preprocessing, and constant time query. Note that this is easy to do in $O(\log n)$ time using Range Trees — store minimum in internal nodes, and traverse from i to j in $O(\log n)$ time.

It turns out that LCA and RMQ are equivalent.

2.1.1 Reduction from RMQ to LCA

Build a *Cartesian Tree*: Walk through the array, while keeping track of the right spine of the tree. When inserting a new element, if the element is the largest element in the spine, insert it at the end. Else, we have an edge from v to w, where the new element a should be in between. Make a the right child of v, and make w the *left* child of a. This step looks like it will be linear, but the more we have to search through the right spine, the more swaps we do, and the smaller the spine gets, so it amortizes to constant time, making the algorithm linear.

Note that simpler divide and conquer algorithm runs in $O(n \log n)$.

We see that $\mathrm{LCA}(i, j)$ in the cartesian tree is the same as $\mathrm{RMQ}(i, j)$ in A.

2.1.2 Reduction from LCA to RMQ

Traverse the tree in-order, and write out the *depth* of each node to A. Now $\text{RMQ}(i, j) = \text{LCA}(i, j)$. Naturally, since this scheme should work for any tree, we cannot write out the node values, since the tree may not be a cartesian tree.

Note that if we go from RMQ to LCA and back again, we end up with different numbers in the array. However, since we are not interested in the actual minimum, but only the index of the minimum, the two arrays act exactly the same. A consequence of this is that we get *Universe reduction*, where we have mapped the universe $\mathcal{U}$ to $[n-1]$.

2.2 Constant time LCA (RMQ)

Step 1: Reduction

We start by reducting the problem to ± 1RMQ, in which adjacent elements of the array differs by at most 1. Walk an *Euler Tour* of the tree: visit every edge twice, and for each edge write down the node we left. Each node store a pointer to the first visit of that node in the array, and the array elements store a pointer to its element in the tree. RMQ and LCA are still equivalent.

We need this later on (step 4).

Step 2: Precomputation

Get $O(1)$ time and $O(n \log n)$ space RMQ. Precompute and store all queries from any starting point where the interval length is a power of 2. Since there are n starting points, and $\log n$ powers of 2 to choose from, there are $n \log n$ such queries. The key observation is that any arbitrary interval is the union of two such intervals. For instance $A[4..11] = A[4..8] \cup A[7..11]$. The double counting does not matter, since min is an idempotent operation. The intervals are trivially computed.

Step 3: Indirection

Indirection. Make a two layer structure: divide the numbers in A into groups of size $1/2 \log n$, which makes the bottom layer. The top layer consists of the minimum element in each block. Since there are $n/(2 \log n) = 2n/\log n$ such blocks, there are equally many items in the top layer.

A query in this structure consists of (potentially) three parts: A query in the bottom block in which i is, a query in the top block for all blocks which are completely covered by the interval $[i, j]$, and a query in the bottom block in which j is. We need all three queries to be $O(1)$.

The gain from this is that the top layer only stores $O(n/\log n)$, so we can afford Step 2, since the log factors cancel. We get $O(1)$ query and $O(n)$ space for the top structure.

Step 4: Lookup Tables

We use lookup tables for the bottom groups. The groups are of size $n' = 1/2 \log n$. RMQ queries in these groups are invariant over value "shifts" in the gruop: if we add a to all elements in the group, the queries are still the same. Shift all groups by its first element, such that all groups start with 0. Now every group is completely defined by the difference of adjacent elements, so the blocks can be encoded as a bitstring of the same length as a block, where 0 is decreasing and 1 is increasing: $[3, 4, 5, 4] \to [0, 1, 2, 1] \to [-, 1, 1, 0]$. There are $2^{n'} = \sqrt{n}$ possible such blocks, $(1/2 \log n)^2$ possible queries, and each answer requires $\log \log n$ bits, so storing a lookup tables for all possible blocks, over all possible queries with all possible answers take $\sqrt{n}\ (1/2 \log n)^2\ \log \log n = o(n)$ bits. Now each bottom block can simply store a pointer into the table, and we get $O(1)$ query for the bottom groups.

2.3 Constant Time LA

Level Anscestor queries take a node x and a level n, and the goal is to find the nth parent of x in the tree. The simplest way to do this is for each node to store its parent, making the query $O(n)$. We want $O(1)$.

Step 1: Jump Pointers

Instead of having each node storing only its parent, each node can store its 2^kth parent. Each node has $O(\log n)$ such parents, making the space requirement $O(n \log n)$. Query time is $O(\log n)$, since we at least halve n each jump.

Step 2: Long-path Decomposition

Decompose the tree into a set of paths. Find the longest path in the tree from the root, and store the nodes in an array. The nodes themselves store the array and its index in the array. Recurse on the subtrees that are hanging from the path.

With this scheme, a query is done as follows: if n is less than the nodes index in its path, we jump directly to the node. Else, we jump to the first node in our path, subtract n by xs index, and repeat. We risk at most to visit $O(\sqrt{n})$ such paths, since we know that the paths are the *longest* paths. We end up using $O(n)$ space, and $O(\sqrt{n})$ query time.

Step 3: Ladder Decomposition

Extend each path upwards by twice its length. Now the arrays overlap, but the nodes still only store their original array and index. This doubles the space of Step 2, but we are still linear. The improvement of this step is that we at least double the length from the node to the end of its path, since the path in which we can jump freely goes at least twice that length up.

Step 4: Step 1 + Step 3

We combine the jump pointers and the ladder decomposition. Jump pointers are great for long jumps, and ladders are great for short jumps. We follow the jump pointer $k/2 \leq 2^{\lfloor \log k \rfloor} \leq k$ steps up, for some k. Since we have gone up a path from x to a node by the jump pointer, we know that the node we hit is of large height, and hence is part of a long ladder. Since its height from the end of the path can be doubled by Step 3, we can get from $k/2$ to k, in which we know our target is. Hence, we get $O(1)$ query, but still $O(n \log n)$ space (and preprocessing).

Step 5: Only Leaves Store Jump Pointers

Since all nodes have constant access to a leaf node (by its ladder, of which the last node is a leaf, by the maximal property), only leaves need to store the jump pointers. In other words, we make all queries start at leaves.

Step 6: Leaf Trimming

Definition — *Maximally Deep Node*: A node with $\geq \frac{1}{4} \log n$ descendants.

Split the tree in two layers by the maximally deep nodes. The number of leaves in the top part is now $O(n/\log n)$, since for each $1/4 \log n$ nodes in the original tree we have "replaced" it with a subtree (the bottom structure). If we now use Step 5 on the top, we get $O(n)$ space.

Step 7: Lookup Table

For the bottom trees, we use lookup tables. The trees are of size $n' \leq 1/4 \log n$. The number of rooted trees on n' nodes is limited by $2^{2n'} \leq \sqrt{n}$ by encoding an euler tour as a string of ± 1, like in Section 2.2. There are $(n')^2 = O(\log^2 n)$ possible queries (if n is large, we just go to the top structure), and an answer takes $O(\log \log n)$ bits, so a lookup table for all possible trees, with all possible queries takes only $\sqrt{n}\ O(\log^2 n)\ O(\log \log n) = o(n)$ bits.

We end up with $O(1)$ time queries, using $O(n)$ space!

Chapter 3

Strings

Topics: *String search, suffix trees, suffix arrays.*

We want to to *String Matching* efficient: given a *text* T and a *pattern* P we want to see if $P \in T$, or to find all occurences of P in T. Both P and T are strings over some alphabet Σ. There exists linear time algorithms to do this, like Knuth-Morris-Pratt, Boyer-Moore, or Karp-Rabin.

We look at a static data structure on T, with the goal of getting string matching in $O(|P|)$ time and $O(|T|)$ space.

3.1 Warmup: Library Search

Given a set of strings $T_1, \ldots, T_k$, we query with a pattern P and want to get Ps predecessor among the Ts.

Definition — *Trie*: A Trie sis a rooted tree with child branches labeled with letters in Σ. Let T be the number of nodes in the trie. This is bounded by $\sum_{i=1}^{k} T_i$ (equality if no pair of strings share a prefix).

A trie can encode multiple strings, by having the edges in a path from the root to a leaf spell out the string. However, we need a terminal symbol \$ to denote the end of a string, so we can have prefixes of a string in the same trie.If each node traverses its edges in sorted order an in-order traversal of the trie yields the strings of the trie in sorted order.

3.1.1 Trie Representation

How do we represent a trie? More specifically, how does each node represent its children?

Array

Nodes can store an array of length Σ, with pointers to the next nodes, or $\perp$ pointers signaling absence. Query for a node in $O(1)$ time, predecessor can also be $O(1)$ by having each node point to its predecessor, since the tree is static. The real downside of this approach is the space, since it is proportional to the alphabet size.

Balanced Binary Search Tree

We can put nodes in a BBST for each child, with a pointer to the child node. Each query is $O(\log \Sigma)$, and there are $O(T)$ queries to be done. Predecessor is simple, and space is $O(T)$.

Hash Table

We can use a hash table to map characters from Σ to a pointer to the child node. This is fast, so query is $O(P)$ in total. However, we do not support predecessor queries, which we need. Space is $O(T)$.

van Emde Boas/y-fast tree

Use a van Emde Boas tree for the children, which allows lookup in $O(\log \log \Sigma)$ time. Also handles predecessor. See Section 8.1 for details on the van Emde Boas tree.

Hashing + vEB

Have both a hash table *and* a van Emde Boas tree. Use the hash table until we get a miss, and use the tree to find its predecessor. After this first miss, we only care about the largest string in the remaining subtree, which can be explicitly stored for the hash tables, since it is only a constant overhead per node. This makes the query time $O(P + \log \log \Sigma)$, since we only use the tree once.

Weight Balanced BST

Instead of having a balanced BST over each nodes children, we can weight each child with the number of leaves in its subtree. This ensures that every second jump in the BST either reduces the number of candidate strings *in the trie* to 2/3 of its size, or it finds a new trie node in the WBBST (hence we advance P one letter). An intuition for this claim is this:we might be so lucky as to cut out 1/2 of the leaves when leaving a node, unless there is some really heavy child in the middle (remember we have to retain ordering). But then in the next step this large child will surely be either to the far left or to the far right, which means we either follow it (which gets us to a new node in the trie), or we discard it, discarding a large number of leaves.

We end up with a running time of $O(P + \log k)$ where k is the number of leaves, and remain at $O(T)$ space.

Leaf Trimming

Definition — *Maximally Deep Node*: A node with $\geq \Sigma$ leaf descendants.

As in Section 2.3 we cut the tree in two parts, the top part and the bottom parts. The tree is cut at the minimal maximally deep nodes. Now the number of leaves in the top part is $\leq |T|/|\Sigma|$, and the number of branching nodes in the top part is *also* $\leq |T|/|\Sigma|$. Now we can use arrays in the top part of the tree as well as for the cut nodes, since there are only $|T|/|\Sigma|$ nodes, so the Σs cancel for the space. For the non-branching nodes, nodes that only have one descendant, we can simply store its decendant and its label. For the bottom part of the tree we can use leaf trimming, since $k = \Sigma$, by the definition of maximally deep nodes.

We end up using $O(T)$ space for the top structure, and get $O(P + \log \Sigma)$ query time.

Table 3.1 sums up all approaches, with query time and space requirement.

	Node representation	Query	Space
1	Array	$O(P)$	$O(T\Sigma)$
2	Balances BST	$O(P \log \Sigma)$	$O(T)$
3	Hash Table	$O(P)$	$O(T)$
3.5	van Emde Boas/y-fast tree	$O(P \log \log \Sigma)$	$O(T)$
3.75	(3) + (3.5)	$O(P + \log \log \Sigma)$	$O(T)$
4	Weight-balanced BST	$O(P + \log k)$	$O(T)$
5	Leaf Trimming + (1) + (4)	$O(P + \log \Sigma)$	$O(T)$

Table 3.1: Table over node representation, query time, and space requirement.

3.2 Suffix Tree

Definition — *Compressed Trie*: A trie where all internal nodes are branch nodes, and the edge labels are strings over Σ rather than characters.

A Suffix Tree is a compressed trie of all suffixes of a string. The tree has $|T| + 1$ leaves: one leaf for each suffix, including the empty string. The edge labels are typically stored as indices in the string, instead of the string itself. Instead of appending \$ to each of the suffixes, which are the strings we are inserting into the tree, we can simply append \$ to the string T, since this will make it the last character in all of the suffixes. The structure takes $O(T)$ space.

3.2.1 Applications of Suffix Trees

Suffix trees are surprisingly useful, and with some of the results from Chapter 2 we can get some impressive results.

String Matching

A search for P in the tree yields a node which leaves corresponds to all matches of P in the text. The search is done in $O(P)$ time (trivial). We can then list the first k occurences of P in $O(k)$ time, by simply traversing the subtree. If we precompute the number of leaves below all nodes, we can retrieve the number of occurences of a string in $O(1)$ time after the inital search, making the total time $O(P)$.

Longest Repeating Substring

We are looking for the longest substring $S \subseteq T$ that is present at least twice. This can be done in $O(T)$ time using suffix trees, since it is the branching node of maximum "letter depth".

Longest Substring Match for i, j

How long is the longest common substring for $T[i..]$ and $T[j..]$? Find the two nodes LCA in $O(1)$ time to get the common prefix.

3.3 Suffix Arrays

While suffix trees are constructable in $O(T)$ time, it is difficult. We look at a simpler structure, the Suffix Array, which is equivalent to a suffix tree, but easier, although slower, to construct.
Definition — *Suffix Array*: An array A of indices into a text T such that a_i gives the suffixes $T[a_i :]$ in sorted order.

We let $\forall_{\sigma \in \Sigma}\ \$ < \sigma$. Suffix arrays are searchable in $O(P \log T)$ time using binary search. In addition to the suffixes, the suffix array also holds information about the difference in adjacent suffixes: This allows for faster searching, as we know how much of the adjacent suffix matches what we already have matched.
Definition — LCP$[i]$: Longest common prefix of ith and $i+1$th suffix in the order they are in the suffix array.

Example: Consider $T =$ `banana$`. The suffixes are shown in Table 3.1a, and the suffix array is shown in Table 3.1b. Note that the suffixes are not stored explicitly in the array, but only the indices i and the LCP.

3.3.1 Suffix Tree → Suffix Array

This way is simple: traverse the tree in-order.

i	Suffix
0	`banana$`
1	`anana$`
2	`nana$`
3	`ana$`
4	`na$`
5	`a$`
6	`$`

a The suffixes of T

i	Suffix	LCP
6	`$`	0
5	`a$`	1
3	`ana$`	3
1	`anana$`	0
0	`banana$`	0
4	`na$`	2
2	`nana$`	–

b The Suffix Array of T.

3.3.2 Suffix Array → Suffix Tree

We make a Cartesian Tree (see Section 2.1.1) of the LCP array. This time we put *all* minimum values at the root[1]. The suffixes of T are the leaves of the tree. Note that the LCP value of the internal nodes is the letter depth of that node, so the edge length between two internal nodes is the difference in LCP. We know from Section 2.1.1 that this is doable in linear time.

3.3.3 Construction

If we have the suffix array it is possible to construct the LCP array in linear time.We look at a method of constructing the suffix array from scratch in $O(T + \text{sort}(\Sigma))$ time.

Definition — $\langle a, b \rangle$: Let $\langle a, b \rangle$ denote the concatenation of the strings a and b.

Step 1

Sort Σ. This step can be skipped if we do not need the children of the suffix tree nodes in order. If we skip this step, the construction is done in $O(T)$ time.

Step 2

Replace each letter in the text by its rank in the sorted array. By doing this we guarantee that $|\Sigma'| \leq |T|$, in case $|\Sigma|$ is really large.

[1] note that the number of 0s in the array is equal to the number of different characters in T

Step 3

Let

$$T_0 = \langle (T[3i+0], T[3i+1], T[3i+2]) \text{ for } i = 0, 1, 2, \dots \rangle$$
$$T_1 = \langle (T[3i+1], T[3i+2], T[3i+3]) \text{ for } i = 0, 1, 2, \dots \rangle$$
$$T_2 = \langle (T[3i+2], T[3i+3], T[3i+4]) \text{ for } i = 0, 1, 2, \dots \rangle$$

If $T =$ `banana`, $T_0 = \langle$`ban ana`$\rangle$, $T_1 = \langle$`ana na`$\rangle$, and $T_2 = \langle$`nan a`$\rangle$. We think of the elements of T_i as "letters". Our goal is to use the Ts to construct the suffix array for T. Note that $\textsc{Suffix}(T) \cong \textsc{Suffix}(T_0) \cup \textsc{Suffix}(T_1) \cup \textsc{Suffix}(T_2)$.

If $T =$ `banana`, $\textsc{Suffix}(T_0) = \{$`ban ana`, `ana`, $\epsilon\}$, since we operate on the triplets as letters.

Step 4

Recurse on $\langle T_0, T_1 \rangle$. End up with a suffix array of the suffixes in $\langle T_0, T_1 \rangle$. Note that the number of suffixes when recursing is $O(2/3)$ of what we started with.

Step 5

Now we want to use the suffix array just obtained to radix sort the suffixes in T_2. Note that

$$T_2[i..] = T[3i+2..] = \langle T[3i+2],\ T[3i+3..] \rangle = \langle T[3i+2],\ T_0[i+1..] \rangle.$$

We can take off the first letter of the suffix, and get a suffix which we know the sorted order or, since it is in T_0, which we sorted in Step 4. This is like Radix sort, but with only two values, both of which can be compared in $O(1)$. This allows us to sort T_2.

Step 6

Merge T_2 into T_0, T_1. While this seems straight forward (merging is $O(n)$) we still need to make sure that the suffix comparisons are done in constant time. This is done in a similar fashion to Step 5: take off the first letter in each suffix and rewrite the suffix in terms of a single letter + suffixes we already know the order of.

Since all operations are linear, with the exception of the recursive call, we get the following recurrence: $T(n) = T(\frac{2}{3}n) + O(n) = O(n)$. Linear time!

Chapter 4

Temporal Structures

Topics: *Partial persistency, full persistency, funcitonal persistency, pareial retroactivity, full retroactivity.*

When working with temporal data structures, our machine model is the *Pointer Machine*
Definition — *Pointer Machine*: A computational model in which we have *Nodes* with $O(1)$ fields. The fields can be values or *Pointers*, which points to other nodes. Field access, node creation, data operations are all $O(1)$ time. The pointer machine does not have arrays.

4.1 Persistence

A persistent data structure is a data strucure that "never changes". On a mutable operation to the structure, a new structure is made. This means that all operations done to the data strucure is relative to a specific version of the structure.

There are four levels of persistence:

1. Partial persistence: Only the latest version is editable. The versions are linearly ordered through time.

2. Full persistence: Any version can be updated, to produce a new version. The versions makes a tree.

3. Confluent persistence: Versions can be merged to produce a new version. The graph is now a DAG.

4. Functional: Nodes in the data structur are never modified, only new ones are created. The version of the structure is maintained by a pointer.

4.1.1 Partial Persistence

Any data structure using the pointer machine with $\leq p = O(1)$ pointers to any node in any version can be made partially persistent, with $O(1)$ amortized multiplicative overhead and $O(1)$ space per change. We show how this can be done.

Each node store *Back Pointers*, which points to all nodes that have a pointer to this node. Since there are at most p such pointers, this is $O(1)$ extra space. In addition, the nodes store a history of modifications $(version, field, value)$ tuples. The history is of size $\leq 2p$ (using the fact that $p = O(1)$). On field reads we read the value explicitly stored in the node, and then apply all the modifications that were done up to the verison we are reading from. When the history is full, we create a new node with the entire modification history applied to its fields, and an empty history. Now we need to update the back pointers of the nodes we point to, since these nodes also store back pointers. This is easy, since the back pointers are not persistent. But we also need to update the pointers (not back pointers) of other nodes (which is the nodes we have in our back pointer list), and these pointers are persistent, so we must recurse.

It is not trivial to see that this recursion terminates. What if the nodes we update get a full history from updating their pointer to us, which makes us update our pointers making our history full again?

We can use the *Potential Method* for amortized analysis. Remember that *Amortized cost* is actual cost + change in potential. Let $\Phi = c \cdot \Sigma$, where Σ is the number of modifications in all nodes latest versions. Observe that when we make a change, the amortized cost is

$$\leq c + c\ [-2cp + p \cdot \text{recursions}]?$$

The first c is for computation, the second c is the added potential, since we just added one modification to a node, and $[-2cp + p \cdot \text{recursions}]?$ are for the recursions, if there is one. If there is no recursion, we have constant time. If there is a recursion, we replace it by the same expression, which causes the $-2cp$ to cancel. We do end up with another recursion, but if it does happen, we still get rid of the $-2cp$ term.

In conclusion, we use $O(3p) = O(1)$ extra space for each node, and allow $O(1)$ extra time on updates, to get partial persistence for any data structure.

4.1.2 Full Persistence

We take a similar approach to full persistency as we did with partial persistency. The first difference is that we need *all* pointers to be bi-directional, and not only the field pointers, as previously. The second and most important difference is that now versions of the structure is no longer a line, but a tree. In order to go around this problem we linearize the version tree: traverse the tree, and write out first and last visit of each node.

However, we need this structure to be dynamic, so we use an *Order Maintenance data structure*, which supports insertion before or after a given item in $O(1)$ (like a linked list), and supports relative order query of two items in $O(1)$ time.

Let p still be the in-degree of nodes. Let d be the maximum number of fields in a node. We now allow $2(d + p + 1)$ modifications in a nodes history.

A single nodes history now consists of the same triples, but the *version* can no longer be compared by $\leq$, since the versions are in a tree. This is what we need the $O(1)$ relative ordering for.

On field read we can simply go through all modifications in the node (since there is a constant of them), and use relative ordering to find the latest update of the field we are reading, from an ancestor of the current version.

Updates when the history is full is different from the approach taken in partial persistence. We would like to split the history tree into two parts of the same size, and apply all of the modifications from the root to the newly cut out subtree to the new node. This was the original node loses the modifications in the new subtree, and the new node loses the modifications that are left in the original node. Now we need to update pointers. We have d fields, p back pointers, and $d + p + 1$ items in the history, all of which could be pointers, which makes $\leq 2d + 2p + 1$ pointers. Amortization scheme still works out (although the potential function is slightly different), and we get $O(1)$ updates.

Confluent persistency and functional were not covered in class.

4.2 Temporal Data Structures

Temporal data structures allow all operations to specify a time in which the operation is to be performed. The key difference between temporal and persistent structures is that in a temporal structure previous alterations to the structure should "propagate" through time, and update the current structure. A persistent data structure would not update the current strucure, but simply make a new data structure for each of the paths taken through time.

4.2.1 Retroactivity

We need three operations:

1. INSERT(t, op(...)): retroactively perform the specified operation at time t.
2. DELETE(t): retroactively undo the operation at time t.
3. QUERY(t, op(...)): execute the query at time t.

We differentiate between *Partial retroactivity*, in which we are only allowed to QUERY in the present, and *Full retroactivity*, in which we may query whenever we want.

A Simple Case

If the updates we perform are commutative ($x, y = y, x$) and invertible ($x, x^{-1} = \varnothing$) then partial retroactivity is easy: Insert operations can be done in the present, since the ordering does not matter, and delete operations can be done as inverse insert operations.

An example is if we were to sum a sequence of numbers. If we sum $1 + 3$ and would like to insert a $+2$ in between, we can put it at the end. If we want to delete the $+3$, we can add a -3 at the end.

4.2.2 Full Retroactivity

What do we need for full retroactivity?
Definition — *Decomposable Search Problem*: $\text{query}(x, A \cup B) = f(\text{query}(x, A), \text{query}(x, B))$

Example: Nearest-Neighbour, Successor, and Point-Location are all decomposable search problems.

If what we query is a DSP, we can achieve full retroactivity in $O(\log m)$ factor overhead, where m is the number of retroactive operations. We can do this by using a *Segment Tree* over the operations. We think of the operations as begin in a time interval, from they are made until they are deleted. Then we can store the operations in the nodes of the maximal subtrees that covers the interval. Each node in the segment tree has its own copy of the underlying datastructure which we are making retroactive. Since there are $O(\log m)$ nodes an element can be in, and the individual results can be joined *somehow* (via f, since the problem is a DSP), we get $O(\log m)$ multiplicative overhead.

4.2.3 General Transformations

What can we do if we do in general? The most obvious method is the *Rollback Method*: when we want to do a retroactive change, we roll back the data structure to the given time, make the change, and replay all changes afterwards. This makes for a $O(r)$ multiplicative overhead, where r is the number of time units in the past. Unfortunately, depending on the exact model used, this is the best we can do in general.

There is a lower bound: $\Omega(r)$ overhead can be necessary for retroactivity. This means that the most efficient way to make retroactive changes is to go back, make the change, and redo whatever comes after — the rollback method!

To see why this is the case, consider a very simply computer with two registers X and Y, and with the following operations: $X = x$, $Y \mathrel{+}= \Delta$, and $Y = XY$. On query, the machine returns Y, and all operations are $O(1)$. The operation sequence

$$\langle Y \mathrel{+}= a_n,\ Y = XY,\ Y \mathrel{+}= a_{n-1},\ \ldots,\ Y \mathrel{+}= a_0 \rangle$$

computes the polynomial $\sum_i^n a_i x^i$. We can use this to compute the polynomial for any X by retroactively inserting $X = x$ at $t = 0$. However, it is not possible to reeveluate such a polynomial using field operations any faster than to just evaluate it again.

4.2.4 Priority Queue

We now look at an example of a retroactive priority queue that supports INSERT, DELETE-MIN, and is partially retroactive. We assume keys are only inserted once.

We can plot the lifetime of the queue in 2D, where the x dimension is time and the y dimensions in key value. Keys in the queue are plotted as points when they are inserted, and are extended as horizonal rays. On DELETE-MIN, we shoot a ray from the x-axis at the time of the delete upward untill it hits a horizontal ray. This makes ⌉ patterns.

Let Q_t be the set of keys in the queue at time t. What happens if we INSERT$(t, \text{insert(k)})$? Its ray will extend to the right and hit a deletion ray, which will delete it. This will make the element that ray previously deleted to not be deleted after all, so its ray will extend further to the right and hit another deletion ray. In effect, the element that ends ut not getting deleted is $\max\{k, k' \mid k' \text{ deleted at time} \geq t\}$, but this relation is hard to maintain.

In order to make things easier we define a *Brige*.

Definition — *Bridge*: A Bridge is a time t such that $Q_t \in Q_{\text{now}}$

Now if t' is the bridge preceding t we see that

$$\max\{k' \mid k' \text{ deleted at time} \geq t\} = \max\{k' \notin Q_{\text{now}} \mid k' \text{ inserted at time} \geq t'\}$$

In other terms, the largest key deleted after a time t is the largest key inserted after the previous bridge that is not in the final set.

Now we can store Q_{now} as a BBST on values, and all insertions in a BBST on time. The latter trees nodes is also augmented with the value of the largest insert in its subtree that is not in Q_{now}. At last, we store a third BBST with *all* the updates, ordered on time, and also augmented as follows:

$$\text{Augmentation} = \begin{cases} 0 \text{ if INSERT}(k), \text{ and } k \in Q_{\text{now}} \\ +1 \text{ if INSERT}(k), \text{ and } k \notin Q_{\text{now}} \\ -1 \text{ if DELETE-MIN} \end{cases}$$

In addition the internal nodes store subtree sums and min/max prefix sums. We do this in order to detect brdiges, since a bridge is a prefix summing to 0. When we have to find out which element to insert into Q_{now} we can walk up from the node in the insertion BST, and find the max to the right, since this BST stores this for all subtrees. $O(\log n)$ time.

4.2.5 Other Structures

We list other strucutes that can be made retroactive. A Queue can be made partial retroactive with $O(1)$ overhead and full retroactive with $O(\log m)$ overhead. A Deque and UNION-FIND can also be make fully retroactive with $O(\log m)$ overhead. The priority queue, which we just made partially retroactive with $O(\log m)$ overhead, can be made fully retroactive with $O(\sqrt{m}\log m)$ overhead. Successor queries can be done in $O(\log m)$ partial retroactive, and since it is a decomposable search problem (see 4.2.2) we can pay a log factor to make it fully retroactive, with $O(\log^2 m)$ overhead. However, it is also possible to get full retroactivity with only $O(\log m)$ overhead.

4.3 List Order Maintenance

Before tackeling the real problem, we look at an easier problem.

4.3.1 List Labeling

We want to store integer labels in a list, such that insert/delete queries around a node in the list are constant, and that the list is in a strictly monotone ordering. Let *Label Space* be the size of the labels as a function of the number of elements in the list we want to store. Table 4.1 shows the best known updates for different sizes of the label space.

Label Space	Best known update
$(1+\epsilon)n \dots n \log n$	$O(\log^2 n)$
$n^{1+\epsilon} \dots n^{O(1)}$	$O(\log n)$
2^n	$O(1)$

Table 4.1: Table showing label space vs best known update for LIST-LABELING

Chapter 5

Geometry

Topics: *Orthogonal Range Search, Range Trees, Layered Range Trees, Fractional Cascading*

We look at problems involving geometry, for instance queries in 2D space: given a set of points, which points are in an axis aligned rectangle?

In general, geometric data strucutres is all about data in higher dimensions. We differentiate between static structures and dynamic structures.

5.1 Line Sweep

We look at the problem of maintaining line-segments in 2D; we would like to store the order of the lines and the intersections. In Chapter 4 we looked at time traveling data structures. We can use these to shave off a single dimension on geometry problems by pretending that one axis is time.

By walking along the x-axis we can maintain a BBST with the points. On each time t where something happends, that is either a segment is started, ended, or a crossing occur, we can translate this to an operation in the BBST. For the static version we need a persistent BBST. This allows queries to be done in a specific time, which is our x coordinate. Now if we would like to know which line is above a point (x, y), we can translate the x coordinate to a time t, and query for the successor of y in the BBST at time t. We get $O(\log n)$ query after $O(n \log n)$ preprocessing (building the persistent BBST).

The dynamic version is similar, but we need a retroacitve BBST, since we need to insert segment begin and segmen end events.

5.2 Orthogonal Range Search

In this problem we want to maintain n points in d dimentional space, and answer queries where we ask for the points in a d dimentional hypercube $[a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$. We want existence, count, or the actual points. The time bound we are aiming for initially is $O(\log^d n + k)$ where k is the number of points we return (trivial bound). Again we differentiate between the dynamic and the static version.

5.2.1 $d = 1$

We store the points in a BBST where all the points are leaves (this only doubles the space). For a range search $[a, b]$ we find the nodes $a' = pred(a)$, $b' = succ(b)$, and $\text{LCA}(a', b')$, and report the points in the subtrees in between a' and b'. Since both *pred* and *succ* are $O(\log n)$ and the size of the subtree between a' and b' is $O(k)$ we get queries in $O(\log n + k)$ time.

5.2.2 $d = 2$

We store a 1-dimentional tree on the x coordinate of all points, similar to in the $d = 1$ case. This time however, we augment the nodes of the tree with a new tree, containing the same points, but sorted on y. That is, any node is itself the root of a tree sorted on x which containts the points P as leaves. The node also points to a new tree which stores P on y. The y-trees are independent, and have no cross links.

We note that each point is in $O(\log n)$ trees: the main tree, and one for each ancestor node, of which there are $O(\log n)$. On a query, we find the subtrees of the main tree containing all points with x coordinates in the range. Then we go through all y-trees and do a range search on y. This gives us $O(\log^2 n + k)$ query time.

The space requirement is only $O(n \log n)$, and construction time is also $O(n \log n)$. Observe that the space recurrence is $S(n) = 2S(n/2) + O(n)$, the same as the time recurrence for Merge-Sort.

5.2.3 $d = D$

The approach taken in the $d = 2$ case generalizes to any dimension. We end up with $O(\log^D n + k)$ query, $\Theta(n \log^{D+1} n)$ space and construction, and $\Theta(\log^D n)$ update (in the dynamic setting).

5.3 Layered Range Tree

We observe that the approach taken in the previous section is wasteful: when $d = 2$ we search for the same y-intervals in $O(\log n)$ trees. We want to take advantage of this by reusing the searches.

Instead of having the nodes in the x-tree store another tree, this time they only point to a sorted array on y. The idea is that we only want to do a single search on y, which will be in the

root node (the array containing all points). Now, when we walk down from the root to $\mathrm{LCA}(a', b')$ (the pred. and succ.) we can follow pointers into the child array, so that we know at once where we are. Now when we get to the subtrees we want to output we have the y-interval of points that we are interested in, and since the subtree is completely covered on x, these are exactly the points we are looking for, so we get this search "for free".

Note that we depend on the fact that a child node has a subset of the point that the parent has.

We start out with querying the points in the y array which takes $O(\log n)$ time, and then we walk down to the leaves, which is a walk of length $O(\log n)$. On each step we output points, of which there are k in total. Hence we end up with $O(\log n + \log n + k) = O(\log n + k)$ queries ($O(\log^{d-1} n)$ in general). The space is the same as previously: $O(n \log n)$ ($O(n \log^{d-1} n)$ in general). The construction time is also the same, since the pointer setup can be done in linear time, so we get the same reccurence as Merge-Sort, yet again.

Unfortinately, this does not generalize to higher dimensions: we can only shave off one log factor using this approach.

5.4 Weight-Balance trees

We would like to use range trees in a dynamic setting. The tree we look at is the BB[α] tree. A weight-balanced tree is similar to a *height* balance tree, which we know: AVL trees and Red-Black trees are examples of height-balanced trees. With weight-balanced trees we would naturally like to balance the weight — the number of nodes — of the subtrees instead of the height.

More formally, we have the following invariant:

$$\begin{aligned} &\forall x \; size(left(x)) \geq \alpha \; size(x) \\ &\quad size(right(x)) \geq \alpha \; size(x) \quad \text{where } \alpha \in [0, 1/2] \end{aligned}$$

A curious property of this invariant is that it implies height balance: $h \leq \log_{\frac{1}{1-\alpha}} n$

On update we simply insert at the leaves, and update the weights upward in the tree, assuming all internal nodes store the weights of its children explicitly. When a node becomes unbalanced, we simply rebuild the entire subtree from scratch. While this might seem slow, we can use an amortization scheme where we charge the $\Theta(k)$ updates in a subtree for that subtrees rebuild time, since we need a lot of changes in a subtree before the root of that subtree becomes unbalanced. The details are a little messy, but the bottom line is we get $O(\log n)$ amortized update.

We can apply this to the range tree from Section 5.2 to get $O(\log^d n)$ amortized updates.

We would also like to use the layered approach from Section 5.3 to shave off a log factor, but it turns out that array rebuilding is problematic. However, we only need something array like in the root node of the tree, since we only need a binary search there, and we never use random access for the arrays in the internal nodes as we only follow pointers. We can replace the root

array with a BBST, and the internal array with linked lists. We end up with the same query time $O(\log^{d-1} n + k)$, since the procedure is exactly the same, but also get $O(\log^d n)$ updates.

Chapter 6

Connectivity in Dynamic Graphs

Topics: *Dynamic connectivity on trees, Euler tour trees.*

Before starting, we point out that this chapter is subject to fewer proofs, and more stated results.

We would like to solve the problem of connectivity queries. We maintain a graph which are subject to updates (edge insertion and deletion), and we answer queries of the form "is u and v connected"? As in previous sections we split the problem into two variants: fully dynamic and partially dynamic.

Definition — *Fully Dynamic*: Connectivity queries in which the graph is fully dynamic

Definition — *Partially Dynamic*: Connectivity queries in which the graph update can be either edge insertions *or* edge deletions, but not both. Only insertions is called *incremental*, and only deletion is called *decremental*.

Unless specified, we consider fully dynamic connectivity.

6.1 General Results

Trees

We can handle connectivity queries for trees in $O(\log n)$ time, by using Link-Cut trees or Euler-Tour Trees (Section 6.2). If we limit ourselves to decremental connectivity, constant time is possible.

Plane Graphs

A plane graph is a planar graph with a fixed embedding; that is, edges know which faces they divide, and updates specify the face of the inserted element. Similar to with trees, $O(\log n)$ is also

possible.

General Graphs

Is $O(\log n)$ per operation possible? This is an open problem, but we know how to get $O(\log^2)$ (amortized) update, and $O(\frac{\log n}{\log \log n})$ query. If we are willing to get slower updates for faster queries, $O(\sqrt{n})$ update and $O(1)$ query is possible.

For the incremental case, we can get $\Theta(\alpha(a, b))$, where α is the inverse Ackermann function, by using UNION-FIND.

Decremental is possible in $O(m \log n + n \text{ polylog } n + \#$ queries), where m is the number of edges and n is the number of vertices.

There is also a known fundamental lower bound: either update or query have to be $\Omega(\log n)$.

6.2 Euler-Tour Trees

We now look at the specifics regarding the result on tree connectivity, namely that $O(\log n)$ per operation is possible. We have already seen Euler Tour trees, in Section 2.2. The general idea is to traverse the tree and write down a node every time we get to it. Then we build a BBST of the written down nodes, where the ordering is the order in the list. Each node in the tree store the first and last visit in the BBST. The Euler Tree supports the following operations:

1. MAKE-TREE: Make a new isolated tree
2. FIND-ROOT(v): find the root of vs tree
3. LINK(u, v): Attach u as a child of v
4. CUT(v): remove v from its parent

We look at how each operation is implemented. Before we proceed, we remind ourselved of some of the operations that a BBST supports in $O(\log n)$ time: SPLIT(x): turn the tree into two trees, one in which have the keys $< x$ and the other have the keys $> x$; CONCAT(x, y): turn the two trees x and y where $\forall x_i, y_i \; x_i < y_i$ into one tree with the keys $x \cup y$. Both operations can be done in $O(\log n)$ time.

6.2.1 Make-Tree

This is trivial: the tree for a singleton is the singleton itself.

6.2.2 Find-Root

Note that the root of the tree is not the root of the BBST. We start in the first tour visit of v, walk up to the root, and down to the rightmost node in the tree. The rightmost node is the first visited node, which is the root of the *actual* tree in which we want to find the root. This takes $O(\log n)$ time.

6.2.3 Link

We find the last occurence of v in the BBST, and insert the tree of u in there. We also need to make sure that u and v themselves are occuring as they should after concatinating in us tree. A single split and two concats.

Note that u have to be the root of its tree. What do we do if it is not? We can *reroot* the tree: pick up the node we want to be the new root, such that the remaining of the tree "falls down". This is a cyclic shift in the euler tour, and can be done in one split and one concat, by splitting at the first occurence of v in the tour, and concating it to the end.

6.2.4 Cut

We find the first and last occurence of v in the tree, and cut at those two places, since vs subtree is a contiguous interval in the euler tour.

Then we concat the first and last part together, and remove one of the $parent(v)$ nodes, so there are not two in a row. Two splits and one concat.

Since all operations consists of walking up or down, splitting or concating, which all takes $O(\log n)$ time, we get $O(\log n)$ for all operations. Connectivity queries can be done by comparing the roots of the nodes we are querying.

6.3 Fully Dynamic Graphs

We look at how to obtain $O(\log^2 n)$ amortized queries for fully dynamic graphs. We maintain a spanning forest of the graph, using Euler-Tour trees. Now edge insertion corresponds to LINK. Edge deletion have two cases: if the edge deleted is not in the spanning forest we maintain, nothing has changed; If it *is* we run into trouble, since simply deleting the edge does not imply that the graph becomes disconnected: there might be another edge that we did not use in the spanning tree, because the two components were already connected by the edge that we are now deleting. If we know that no such edge exist, we can simply CUT out the tree, and we are done.

The way we do this is to assign *levels* to edges, and store $O(\log n)$ levels spanning forests, where some edges may get lost when going a level down. All edges start at level $\log n$, and the level is monotoically decresing, and at least 1.

Let G_i be the subgraph of edges with level $\leq i$. Note that $G_{\log n} = G$.

Let F_i be the spanning forest of G_i, stored using Euler-Tour trees. Note that $F_{\log n}$ answers the connectivity queries in G, since the forest spans the entire graph, and support connectivity queries in $O(\log n)$ time.

We maintain the following invariants:

Invariant 1

Every connected component of G_i has $\leq 2^i$ vertices.

Invariant 2

The forests nest: $F_0 \subseteq F_1 \subseteq \cdots \subseteq F_{\log n}$, and are gived by $F_i = F_{\log n} \cap G_i$. There is only one forest, and F_i is just the part of the forest with the lower levels. This also means that $F_{\log n}$ is a minimal spanning forest with respect to edge levels.

Insertion

On insertion of $e = (u, v)$ we set $e.level = \log n$, and add e to u and vs indicence lists. If (u, v) are not connected we add e to $F_{\log n}$. This makes for $O(\log n)$ insertion.

Removal

This is the hard part. We start by removing e from the indicence lists of the veritces it is connected to. This can be done in constant time if the edge itself stores a pointer into where it is in those lists. Then we check if $e \in F_{\log n}$ we are done (if e is in any forest it is in $F_{\log n}$, since they nest). Else, we have to delete e from $F_{e.level} \dots F_{\log n}$, which is exactly the trees that e lives in. All of these are Euler-Tour trees, and there are at most $O(\log n)$ of them, which makes a total cost of $O(\log^2 n)$.

Now we have to look for a replacement edge. We know by invariant 2 that there are no edges with a lower level, since then that edge would be in the tree istead of e. So if there is a replacement edge, it has level $\geq e.level$. We search upwards from $e.level$ to $\log n$. For each level i we let T_u and T_v be the trees of F_i containing u and v. Without loss of generality, let $|T_u| \leq |T_v|$. By invariant 1, we know that the sizes of these components are limited: $|T_u| + |T_v| \leq 2^i$, since they were connected before deleting e. This means that $T_u \leq 2^{i-1}$, so we *can* push down all edges in T_u to level $i - 1$ without destroying invariant 1. We will use this as the charging scheme to get the amortized running time we want.

We look at all edges $e' = (x, y), x \in T_u$ at level i. The edge is either internal to T_u, or it goes to T_v, like e does. Why can it not go to another component T_w? Assume there is an edge $f = (x, w),\ w \in T_w$ of level i. Since $f.level = i$ we know that $f \in G_i$, and since $F_{\log n}$ is a minimal spanning forest, we know that if T_u and T_w are connected in G they are connected in G_i, since f can be used. But this contradicts the assumption, namely that f is not internal to T_u. Therefore T_u and T_w cannot be connected in G, so f cannot exist.

If e' is internal to T_u it does not help us, so we set $e'.level = i - 1$, which we can afford. If e' goes to T_v we are done, since it is a replacement edge; insert it into $F_i, \ldots, F_{\log n}$.

Overall we pay $O(\log^2 n + \log n \cdot \#$ level decreses), but the number of level decreses is bounded by the number of inserts times $\log n$, since edge levels are strictly decresing and between $\log n$ and 1. We can charge inserts with $\log n$, making the amortized cost of delete $O(\log^2 n)$, which is what we wanted.

The last complication is that we need to augment the tree with subtree sizes at every node, in order to make the comparison $T_u \leq T_v$ in constant time, and that we somehow must find all edges on a certain level. To handle this we store in all internal nodes in the Euler-Tour trees an array, signaling whether the nodes in this subtree has any level i edges adjacent to them. In addition, the adjacency lists of the nodes store one list for each level, instead of having one list for all edges. This makes the search to find the next level i edge $O(\log n)$.

6.4 Other Results

We list some other related results in the field of conenctivity.

6.4.1 k-connectivity

2-edge connectivity is maintainable in $O(\log^4 n)$ time, and 2-vertex connectivity in $O(\log^5 n)$ time.

6.4.2 Minimum Spanning Forest

The general problem of maintaining a minimum spanning forest can be solved dynamically in $O(\log^4 n)$ time.

Chapter 7

Lower Bounds

Topics: *Dynamic Partial Sums, Dynamic Connectivity.*

In this chapter we will look at lower bounds for basic computational problems. We will use the *Cell Probe* model, in which we get computation "for free" while we pay for memoroy accesses.

7.1 Dynamic Partial Sum

The problem is to maintain an array A of length n containing numbers, while supporting two operations: UPDATE(i, x), which sets $A[i] = x$, and QUERY(i) where we query the prefix sum $A[1] + \cdots + A[i]$. We will show that there is a lower bound of $\Omega(\log n)$ on either UPDATE or QUERY. In the proof we will use the universe $[n]$ and $+$ as operator, but any group of polynomial size with an arbitrary operator works.

The idea of the proof is to construct an access sequence where we update with random numbers, which will break any data structure. Imagine we list out the queries done through time, and build a mental binary tree over them. We want the sequence to be *maximally interleaved*, which means that any access in a left subtree affects a query in a right subtree for the same node.

The way we choose this sequence is by iterating i from 0 through n, and reverse its bits to get the index. We assume n is a power of two. For instance, if $n = 8$, we get the sequence

$$\texttt{0b000},\ \texttt{0b001},\ \texttt{0b010},\ \texttt{0b011},\ \texttt{0b100}, \cdots \implies 0, 4, 2, 6, 1, \ldots$$

The operations we do on each cell is as follows: we first QUERY the number there, and then we set it to a totally random number. The argument we will make is that the information transfer guarantees our lower bound.

We claim that for a given node in our mental tree, $\Omega(T)$ memory values are written in the left subtree and then read in the right subtree, where T is the size of the subtrees. This implies

that $\Omega(n \log n)$ work is begin done, simply by moving $\log n$ bits of information (one number) for each of the n operations. We prove the claim: each x_i has $\Theta(\log n)$ bits of entropy, since they are totally random numbers. Let R be the memory cells read in the right subtree, and W the memory cells written in the left subtree. We see that if we are given the state of the data structure before performing the operations in the left subtree and the operationg we are going to perform in the right subtree, we can recover the operations done in the left subtree, without knowing what they are. That is, by having the state before and the queries on the right we can recover the x_i we are writing in the left.

To see this, we consider again the cast where $n = 8$, so the left tree is the sequence $0, 4, 2, 6$ and the right tree is $1, 5, 3, 7$. $Q(1) = x_0$ (since $x_1 = 0$), so we know x_0 which was just written. $Q(5) = x_0 + x_2 + x_4 + x_1$; we have to wait for this one. $Q(3) = x_0 + x_2 + x_1$: we know x_1 and x_0, so now we know x_2, which gets us x_4 from $Q(5)$. $Q(7)$ gives is x_6.

Now we can explicitly encode $|R \cap W| \log n$ bits, and use this to extract $\Omega(T \log n)$ bits of entropy: the x_is. Since we have extracted $\Omega(T \log n)$ bits of entropy from $|R \cap W| \log n$ bits, we have $|R \cap W| = \Theta(T)$, which proves the claim.

7.2 Dynamic Connectivity

We have already looked at this problem in Chapter 6, and we claimed that there is a lower bound of $\Omega(\log n)$ on dynamic connectivity. We will now prove it.

The graph we will consider is laid out on a grid, such that adjacent columns have a perfect matching; the graph can be seen as a composition of permutations. We have n vertices, and the grid is a perfect square, so there are $\sqrt{n}$ permutations permuting $\sqrt{n}$ elements. Each permutation requires $\sqrt{n} \log n$ bits.

The operations we consider is updaing the ith perimutation, and querying that a prefix of the permutations is equivalent to a given permutation. These are *block operations*, in the sense that they do not act on single edges or vertices, but groups of them; more specifically they correspond to $O(\sqrt{n})$ regular operations. This can only make our problem easier, since we now can apply amortization schemes, or somehow expolit that we have information about nearby operations. The reason we do not simply query for the total permutation is that this is hard to do using connectivity queries: we would have to search for it, using a lot of queries.

We claim that $\sqrt{n}$ updates and $\sqrt{n}$ verify sums require $\Omega(\sqrt{n}\sqrt{n} \log n)$ cell probes, which implies a lower bound of $\Omega(\log n)$ per operation, since we do $\sqrt{n}$ block operations, which all corresponds to $\sqrt{n}$ graph operations.

7.2.1 The Proof

Similar to in Section 7.1 we will consider the interleaving access pattern. We will look at how much information has to be carried over from the left to the right subtree for a given node. We claim that that every node in a right subtree have to do $\Omega(l\sqrt{n})$ expected cell probes reading cells

that were written in the left subtree, where l is the number of leaves in the subtree. In addition, we sum this lower bound over every node in the tree. Note that there is no double counting of reads, since we only count a read in the LCA for the read and the latest write. Since every leaf is in $\log n$ subtrees, and the size of the tree is $O(\sqrt{n})$ we have $\sum l = O(\log n\sqrt{n})$, so we end up with $\Omega(\sqrt{n}\sqrt{n}\log n)$, which is the claim.

Now we have to prove the claim from the previous paragraph. We can do this in a similar manner as in Section 7.1. The left subtree for a given node contains $l/2$ updates with $l/2$ independent, totally random permutations. An encoding of these updates must use $\Omega(l\sqrt{n}\log n)$ bits (l updates, which are $\sqrt{n}$ numbers of size n). We show that if the claim is false, there is a smaller encoding, which contradicts information theory. We will encode verified sums in the right subtree with which we can recover the permutations in the left subtree. This is done by encoding the *query* permutations on the right.

However, what we assume to know is a little different this time. We still assume we know everything before the left subtree, but we can not assume to know what we are passing into the queries on the right, because this is exactly what we are trying to figure out! In fact, these two things convey the same information, as both is reconstrucable from the other. However, this time we know that the queries, which asks if a composition prefix of the permutations is the same as the given permutation, always answer "Yes".

Short version: We end up also encoding a separator of the sets $R \setminus W$ and $W \setminus R$, where R and W are the cells read and writte to in the right and left subtree respectively. Then we can simulate all possible input permutations and use the separator to quit early if we see that the cell accessed is not right. The encoding size ends up begin $\Omega(l\sqrt{n}\log n)$, which implies that $|R| + |W| = \Omega(l\sqrt{n}\log n)$, which was the claim we needed to prove.

Chapter 8

Integer Structures

Topics: *van Emde Boas, x-fast trees, y-fast trees, fusion trees.*

In this chapter we look at data structures operating on integers in the word machine. By limiting ourselves to only act on integers we can achieve time bounds as a function of the word size.

The general problem we want to solve is PREDECESSOR: we maintain a structure containing n words, and would like to support insertion, deletion, and predecessor and/or successor. The universe U we get the integers from is asusmed to be a power of two 2^w; we call w the word size. We also assume that $n \leq \log w$, that is, that the number of elements in the structure is representable in a word.

In a comparison based model, the precesessor problem has a lower bound of $\Theta(\log n)$. Table 8.1 summarizes the structures we will look at, with time and space complexities. Note that vEB with hashing or y-fast trees are faster than fusion trees if w is small.

Name	Operation	Space
van Emde Boas Trees	$O(\log w)$	$\Theta(U)$
vEB + Hashing	$O(\log w)$ whp.	$\Theta(n)$
y-fast trees	$O(\log w)$ whp.	$\Theta(n)$
Fusion trees	$O(\log_w n)$ whp.	$\Theta(n)$

Table 8.1: The integer structures we look at in this chapter.

8.1 van Emde Boas Tree

The general idea of the van Emde Boas Tree is to try to obtain the time recurrence $T(n) = T(\sqrt{n}) + O(1)$. We can do this by splitting the universe into $\sqrt{u}$ clusters, each of size $\sqrt{u}$, and recurse on the cluster. For universes that are a power of 2, this means splitting the bits of the number if half. Consider a word $a = \langle c, i \rangle$; we call the most significant bits c and the least significant bits i. This is simply to do in the word machine: we simply mask out the lower bits to get i, and we shift down and mask to get c.

The van Emde Boas tree is a recursive structure. At each level we have four things: $\sqrt{u}$ *Clusters*, a vEB trees of size $\sqrt{u}$ which signals which of the clusters are empty; one *Summary*, which is also a vEB tree of size $\sqrt{u}$; the minimum element in the tree; and the maximum element in the tree. The minimum element is not stored elsewhere in the tree, but the maximum element is.

8.1.1 Operations

With this layout, we can see how we will do the recursion: we index the cluster using c, the most significant bits, and then continue by using i in the next query. We will call the vEB tree v, such that $v.cluster[0]$ is the first cluster in the current tree. We assume no key duplication.

Successor

If the queried element is smaller than $v.cluster[c].max$, we find the successor for i in $v.cluster[c]$. When returning we also have to "rebuild" our number, by appending c in front. If the element is larger than $v.max$, we have to find the next non-empty cluster, and select the minimum element in it, since this will be our successor. Note that we know one such element will exist, since $x < v.max$. To find this we do a sucessor query of c in $v.summary$. When rebuilding we now have to use c', the successor of c in the summary, instead of c.

In total, we get one recursive call in either case.

Insert

At first this might seem simple. We find the correct cluster, and recurse into it. However, we also need to update the summary, if the cluter was empty. This makes for *two* recursive calls, which we cannot afford, since $T(u) = 2T(\sqrt{u}) + O(1)$ solves to $O(\log u)$, rather than $O(\log \log u)$ which is what we want ($w = \log u$). However, we observe that if the cluster is empty, the insert call to it will be trivial since we only set $v.min$ which is not stored elsewhere in the tree.

Delete

Deleting an element is slightly more complex. First we check if $x = v.min$. If the tree only contains one element we remove min and max, and return; else we take the min from the first cluster, which

we find by taking the min in the summary, and set *our* min to be this min. Since min keys are only stored in one place, we continue with $x = v.min$; that is, by overwriting $v.min$ we deleted the original queried key, but we have to clean up by deleing the key we just copied.

Now we are either still looking for the original x, or we have changed x. In any case, we recurse on cluster c, and delete i. After deleting, we have to check if the cluster we recursed on got its min removed, because we then have to mark it in the summary. Like with insert, we risk having two recursive calls here, but again one of these calls will be trivial; this time it is the first call that is trivial, since if we removed the last element in the tree we just set $v.max = v.min = None$, and did no further recursing.

8.2 Saving Space

The strucutre presented in Section 8.1 takes $O(U)$ space, which is a lot. We look at ways to reduce this.

The obvious way to shave off some space is to not store the clusters in an array, but in a hash table. This way we only pay for the tables we actually use. Somehow, this gives us a space bound of $O(n \log w)$, which can be improved to $O(n)$ using indirection.

8.2.1 Tree View

We can look at the van Emde Boas tree in a different view. Consider a balanced binary tree of height w, where the leaf nodes are element markers (that is, the path from the root to a leaf is the number, and the leaf is either 0 or 1, if the element is not or is in the tree). The internal nodes are the `OR` of their children. The upper half (in terms of height) of the tree can be considered at the Summary, and each subtree as one cluster.

In this structure, updates are $O(w)$. Queries can be done faster by noticing that a path from a leaf to the root is monotone: it is a string of 0s followed by 1s (unless the tree is empty), so we can binary search it to find the transition point. Then, we can look at the other child of the node that has the first 1 in the path, and get the min or max, depending on if it is to the left or right. If all subtrees store this, this is constant. What if we are looking for successor, but get predecessor from this method? If all 1 leaves store pointers to the next and previous 1 leaves, this can also be done in constant time.

8.2.2 X-fast Trees

We take inspiration from the Tree view in the previous section to build a X-fast tree, where we store all of the 1s in the tree, as binary strings, in a hash table. This lets us perform the binary search. Updates are still $O(\log w)$, since we have to search through the path, which is w long, queries can use the same binary search trick, to get a time of $\Theta(\log w)$, and space is $O(nw)$.

8.2.3 Y-fast Trees

We use indirection on the X-fast tree to get faster updates and smaller space. Split the tree into two structures, an upper and lower structure. The upper structure consists of $O(n/w)$ of the tree nodes, and the bottom ones are BBSTs of size $O(\log w)$. Queries are $O(\log w)$ in both structures, but we make updates $O(\log w)$ amortized, since for each w update in the bottom stucture we only need one update in the top structure. Now the space is only $O(n/ww + n) = O(n)$. Linear space!

8.3 Fusion Trees

Fusion trees are more a hypotethical structure: is is fast when the word size is very large. We want a strcuture with $O(\log_w n)$ operations. Thus, using fusion trees or vEB trees we can get a time bound of $\min\{\log_w n, \log w\} \leq \sqrt{\log n}$. We look at the static version of fusion trees; a dynamic version is possible with $O(\log_w n + \log\log n)$ time operations.

The general idea of fusion trees is to have a B-tree with a branching factor of $w^{1/5}$. This will make the height of the tree $h = \Theta(\log_w n)$. However, simply searching through the keys will be to slow — we need $O(1)$ time in each node (binary search in each node would be $\Theta(\log w \log_w n)$). Let $k = w^{1/5}$ be the number of keys stored in each node, and let $x_0 < x_1 < \cdots < x_k$ be the keys in the node. We would like to pack the bits in $O(w)$ space, and have $O(1)$ predecessor/successor queries on x_i. Note that the queried key x' does not have to be one of the stored keys x_i. We will allow polynomial preprocessing.

8.3.1 Overview

We look at how to approach this problem. The general idea follows three steps: key storage, node search, and key retrieval. We find a way to store the keys in the given space, then a way to find the predecessor of the queired key, and then a way to get back the actual key stored in the node — since we cannot simply store all keys, we have stored a different representation.

Distinguishing the Keys

We make a mental trie of the bits of all keys in the node, and note that certain levels are more "interresting", namely the ones with branching nodes, of which there are $k-1$. Let b_i be important bit i, and let $sketch(x)$ be all interesting bits from x. We claim that $sketch(x_i) < sketch(x_{i+1})$. To see why, we consider the first different bit in the numbers. Since $x_i < x_{i+1}$ this bit have to be 0 for x_i and 1 for x_{i+1}. Since this is a branch, this bit is a part of the sketch, so $sketch(x_i)$ is the smaller.

Since each sketch is k bits long, and there are k of them, we can fit all sketches into a single word ($k^2 = (w^{1/5})^2 = w^{2/5}$). This packing is computable in $O(1)$ time. We would like to somehow search for our queried element in this packing.

Node Search

For a query q we would like to compare it "in parallel" to all x_is, in $O(1)$ time. While this might seem obviously impossible, we remind ourselves that the size of all x_is in our representation is $O(w)$, so all standard operaions are done in constant time for the entire set.

Desketchifying

So we find out that $sketch(x_i) \leq sketch(q) < sketch(x_{i+1})$; now what? Simply retrieving x_i somehow might not be the correct answer: if q branches on some level that is not considered important by x_i, we have effectively ignored a bit of q in its *sketch*. Consider $\mathbf{x} = [\underline{0}0\underline{0}0, \underline{0}0\underline{1}0, \underline{1}1\underline{0}0, \underline{1}1\underline{1}1]$, where the important bits are underlined. If we query for $q = 0101$, we get $sketch(q) = 00$, so $sketch(x_0) = sketch(q)$ even though $x_1 < q$! Even worse, this is not a off-by-one error; they can be very far from each other.

However, all hope is not lost. We can look at the LCA of x_i or q and the LCA of x_{i+1} and q (choose the longer): this is the nodes where we first fell off the tree. In our example above, we went right when both x_0 and x_1 went left. Then we know that the subtree we are in after the LCA node contains no keys. If there had been a key there, we would have gotten this key as either x_i or x_{i+1}, depending on the other important bits of the key. So now, if we are looking for the predecessor, we can simply find the max key in the left subtree of the LCA node. We can do this by searching for $e = y0111\ldots1$ in the left tree. In contrary to the q case, this will work, since any sketch we search for will be less than the sketch of e, since it will only contain 1s. Therefore, we will get the max sketch in the subtree, which is the max value[1]. If we are looking for the successor and went left when the x_is went right, $e = y100\ldots0$.

So to sum up the big picture: we start out by computing $sketch(q)$. Then we find i such that $sketch(x_i) \leq sketch(q) < sketch(x_{i+1})$, and set $y = \mathcal{LCA}(q, x_i) or \mathcal{LCA}(q, x_{i+1})$, depending on which is the longer. We then compute e, find $sketch(e)$, and at last find j such that $sketch(x_j) \leq sketch(e) < sketch(x_{j+1})$, We claim that this will give us the right answer, for both successor and predecessor queries.

However, there are a lot of details that needs explanation. How do we make the sketches? How do we search in the x_is? What about computing the LCA? We look at these questions, in order.

8.3.2 Approximate Sketch

We remember that the perfect sketch takes the exact bits we care about, and packs them right after another, using $O(w^{2/5})$ space. How can we do this? We could start off by masking out the

[1]Note that the reason the sketch search failed for q was that it branched on some bit that was not in the sketch. With e, we only care about getting the maximum/minimum sketch.

important bits; this is easy. However, packing them together to be consecutive is harder. Maybe we do not need to perfectly pack them together? We can allow to have some zeroes in between the important bits, if they are in a predictable pattern, since this does not change the ordering of the sketches. We will make an *approximate sketch*, of size $O(w^{4/5})$ bits, making the total bits in a node $O(w)$, which is our limit.

We start out by masking out the important bits:

$$x' = x \ \texttt{AND} \ \sum_{i=0}^{r-1} 2^{b_i} = \sum_{i=0}^{r-1} x_{b_i} 2^{b_i}$$

In order to pack the bits somewhat together, we will use multiplication.

$$x' \cdot m = (\sum_{i=0}^{r-1} x_{b_i} 2^{b_i})(\sum_{j=0}^{r-1} 2^{m_j}) = \sum_{i=0}^{r-1} \sum_{j=0}^{r-1} x_{b_i} 2^{b_i + m_j}$$

That is, $b_i + m_j$ is something. We claim that there exist an m such that

1. $b_i + m_j$ are all distinct, so we avoid collisions,
2. $b_0 + m_0 < \ldots b_{r-1} + m_{r-1}$, so we preserve ordering, and
3. $b_{r-1} + m_{r-1} - b_0 + m_0 = O(r^4) = O(w^{4/5})$, so the number is small.

Then

$$\mathit{approx\_sketch} = \left[(x \cdot m) \ \texttt{AND} \ \sum_{i=0}^{r-1} 2^{b_i + m_i}\right] \gg (b_0 + m_0)$$

Note that we have discarded all terms where $i \neq j$.

We first consider 1. Choose $m'_0, \ldots m'_{r-1} < r^3$, such that all $b_i + m'_j$ are distinct mod r^3. Do the choosing by induction. When we want to pick m'_t, we must avoid all numbers equal to $m'_i + b_j - b_k \ \forall_{i,j,k}$ (we moved b_i from the left to b_k on the right). i ranges from 0 to t, and j and k ranges from 0 to r, so the total numbers we must avoid is tr^2. But $t < r$, and we are working mod r^3, so there is a number that is available.

Next, we let $m_i = m'_i + (w - b_i + ir^3)$ rounded down to a multiple of r^3. We use the ir^3 part to spread the bits out, since each $m_i < r^3$; this achieves 2. In order not to mess up the collision freeness we have achieved, we need the term we add to be a multiple of r_3, so that $m_i \equiv m'_i$ mod r^3. Since $m_{r-1} = O(r^4)$ and $m_0 \approx m'_0 < r^3$ and $m'_0 \geq 0$, we get $m_{r-1} - m_0 = O(r^4)$.

8.3.3 Parallel Comparison

Now all sketches for all keys in a node takes $O(w)$ space. We want to search in all of them, in constant time. We use subtraction to compare the keys. Let $\mathit{sketch}(\mathit{node}) = 1 \ \mathit{sketch}(x_0) \ 1 \ldots 1 \ \mathit{sketch}(x_{k-1})$,

and let $sketch(q)^k = 0\ sketch(q)\ 0 \ldots 0\ sketch(q)$; this can be computed by multiplying the sketch of q with $00^n10^n1 \ldots 0^n1$, where n is the length of a sketch minus 1. Now we can look at $sketch(node) - sketch(q)^k$: the 1 bits in front of the sketch will be zero if and only if $sketch(q) > sketch(x_0)$. We can mask out these bits. Note that since the x_is are in order, this sequence is monotone, and of the form 0^a1^b.

Now we are interested in knowing where the transition is. The way we will do this is to multiply the masked number with $0^n1 \ldots 0^n1$: this multiplication will make all of the 1s in the number to hit the same bit in the product, which will be summed. That is, instead of finding the position of the transition, we simply count the number of high bits. This is the number of x_is that are larger than q. Note that we also have space in the sketch room for the entire sum, since the length of a sketch is at most $k-1$, and we only have at most k 1s to sum.

8.3.4 LCA

Lastly, we need to compute the LCA for our mental tree. This is equivalent to the first set bit in the `XOR` of the two numbers, also called the most significant bit. We split the word into $\sqrt{w}$ clusters of $\sqrt{w}$ bits each. Then we find the non-empty clusters: get the `msb`s of the clusters, and clear the `msb`s from x. Subtract the cleared x from the `msb` mask, and mask out all bits except the `msb`. `XOR` with the mask again, to flip the `msb`s. Now, each block is either `00...00` or `10...00`, depending on wether the bits except `msb` was 0 or not. The special case we still need to handle is when a cluster is `10...00`. `OR` in the `msb`s, to fix this. Now all non-empty clusters are `10...00`, and empty clusters are `00...00`.

Next we need to find the first non-empty cluster, and then find the first set bit in that cluster. To do this we would first like to compress the bits into a string of just the bits we care about, of length $\sqrt{w}$. Now we use *perfect* sketch (details omitted). Then, to find the most significant set bit, we use parallel comparison, by comparing the sketch repeated to one-hot strings of length $\sqrt{w}$. By finding the largest power of two the string is greater than we find the `msb`. Note that since the size of the sketch is $\sqrt{w}$, and there are $\sqrt{w}$ one-hot strings of length $\sqrt{w}$, the total space for all the one-hot strings is w, so it fits.

Now we have found the first cluster that has a set bit. Next, we go to that cluster, and use the exact same method again to find the `msb` there. This works since the sketch and the cluter are the exact same size. At last, we combine the two indices (cluster index and internal index to the cluster) to get the global index.

Chapter 9

Succinct Structures

Topics: *Rank, Select*

In this chapter we will look at ways to store data using as little extra space as possible. These structures are often static, and the one we will look at, the bit vector, will indeed be static.

We divide the space hierarchy for data structures into three levels: Implicit data structures, which uses the information theorerical optimum space, plus a constant amount of bits; Succinct data structures, which uses the optimum space + $o(OPT)$ extra space; and Compact data structures, which uses $O(OPT)$ space.

9.1 Bit Vector

A Bit Vector is an array of bits. We want to support the following operations: $Rank(i)$ = the number of 1s in the interval $[0, i]$, and $Select(i)$ = the position of the ith 1. In a way, select is the inverse operation of rank. We also want the queries to be fast, and within the memory requirements we have set.

9.1.1 Rank

We would like to precompute queries in chunk of size $\frac{1}{2}\log n$. Since this is a bit vector, the total number of different such chunk is $2^{1/2\log n} = \sqrt{n}$. The number of different possible queries to be done on this chunk is $1/2\log n$, and the size of each answer requires $\log\log n$ bits to be represented. This means if we are to make a lookup table of all possibilities, it would take $\sqrt{n}\cdot\frac{1}{2}\log n\cdot\log\log n = o(n)$.

Next we divide the vector into chunks of size $\log^2 n$, and store the cumulative rank of the chunk boundaries. Since there is $n/\log^2 n$ chunks and we need $\log n$ bits to store the rank, we use

$n/\log n = o(n)$ extra bits. We now need to handle the chunks of size $\log^2 n$, which is slightly too big for the lookup tables. We use indirection a 2nd time!

Split each chunk into subchunks of size $1/2 \log n$. This time, in between the subchunks we store the *local* cumulative rank, instead of the global. This is what differs this approach from simply dividing the entire vector into $1/2 \log n$ chunks. Now, since the size of the chunk is $\log^2 n$, the local rank needs only $\log \log n$ bits, and there are $\frac{\log^2 n}{1/2 \log n} = 2 \log n$ chunk, so we use $2 \log n \cdot \log \log n$ bits for each chunk. Again, there are $\frac{n}{\log^2 n}$ chunks, the total extra space we use is:

$$\frac{n}{\log^2 n} \cdot 2 \log n \cdot \log \log n = \frac{2n \cdot \log \log n}{\log n} = o(n)$$

On queries, we take the rank of the chunk plus the relative rank of the subchunk in the chunk, and then the relative rank of the element within in subchunk, which is precomputed using the lookup table.

It is possible to get Rank in $O(n/\log^k n)$ bits for any $k = O(1)$.

9.1.2 Select

This time we store an array of indices of every $\log n \cdot \log \log n$th 1-bit. That is, instead of dividing the vector into chunks of equal size, we divide it into chunks with the same number of 1s in it. With this we can find out which chunk a query is in in constant time, but we still need to search inside the chunk. In addition, the chunk sizes are now different. Let r be the size of a given chunk. We consider different sizes of r.

If $r \geq (\log n \log \log n)^2$ we can afford to store a lookup table of the answers: there are $\log n \log \log n$ queries for the chunk (one for each 1), and each answer, the index, is $\log n$ bits, so the cost for *one* such chunk is $\log^2 n \log \log n$. Since the size of the chunk is at least $(\log n \log \log n)^2$ there cannot be more than $\frac{n}{(\log n \log \log n)^2}$ of them, so the total size for this scheme in the entire bit vector is $\log^2 n \log \log n \cdot \frac{n}{(\log n \log \log n)^2} = \frac{n}{\log \log n}$ bits.

If $r \leq (\log n \log \log n)^2$ we store the relative indices for every subchunk of $(\log \log n)^2$ 1th bit. We get at most $\frac{n}{(\log \log n)^2}$ subchunks, but they only need to store $\log \log n$ bits each, since the index is relative, which makes the total cost $\frac{n}{(\log \log n)}$ bits. Again we consider different sizes of the now subchunks: if $r \geq (\log \log n)^4$ we store all the answer, as relative indices:

$$O(\frac{n}{(\log \log n)^4} \cdot (\log \log n)^2 \log \log n) = O(\frac{n}{\log \log n}) \text{ bits.}$$

If not, we have subchunks that are so small that we can use the lookup table.

We end up with using $O(\frac{n}{\log \log n})$ bits, while still having constant time. As with Rank, it is possible to get Select in $O(n/\log^k n)$ bits for any $k = O(1)$.

9.2 Navigating Binary Trees

We look at a succinct representation of a binary tree. Consider the tree encoding obtained by depth first search where each node outputs one bit for each children that is a 1 if the child is present and 0 if it is not.This representation clearly uses $2n$ bits. Now if we insert a (1) in front of the encoding, we can nagivate the tree using Rank and Select from the previous section, using the encoding as a bit string.

For a node i in the bit vector, its left and right children are in positions $2Rank(i)$ and $2Rank(i)+1$, and its parent is in position $Select(\lfloor i/2 \rfloor)$, similar to in a binary heap.

Chapter 10

Concurrency

Topics: *Locks, Lock-free structures, lists, priority queues.*

In this chapter we will look at primitives and data strucutres that handles multiple threads acting on them at the same time. Multithreaded programming is extremely error prone, and implementing the primitives and data strutcures even more so. For this reason, we will in this chapter write out pseudocode, instead of an english description.

10.1 Mutual Exclusion

Imagine we would like to find the sum of an array. The sequential way to do this is simple. Let $s := 0$, and for each i to up n, read a_i, add it to s, and write to s. We note that in this simple example we *load* and *write* variables; this is rarely though of in single threaded code, but it is paramount in multithreaded code. We can try the same approach for the parallel version. Assume we have n threads, so that one threads job is to add a_i to s for its i. Now, let $R_1(x)$ and $W_2(x)$ denote that thread 1 reads the variable x and that thread 2 writes the variable x to a shared storage respectively. Note that we differentiate between *shared* and *local* storage for the threads. Processor 1s instructions will be $R_1(a_i),\ R_1(r),\ W_1(r)$; note that we have omitted the addition, since it is local. Processor 2s instruction is similar. Now, we assume that the actual outcome of this computation is the same as some interleaving of the two threads instruction streams. Since the interleaving is not defined, we must make sure that we obtain the correct answer no matter how we interleave the streams. However, consider the following interleaving:

$$R_1(a_i),\ R_1(r),\ R_2(a_j),\ R_2(r),\ W_1(r),\ W_2(r)$$

Both threads read its a and r, then to the adding, and then write out to r again. This is a problem, since a_i will be lost in the sum, as the local r thread 2, which is the value that gets written out

last and which contains a_j does not contain a_i, since r was read before thread 1 had written it out. Note that this problem is just as real if we only have two threads, and each thread gets an interval of the array.

We would like the consistency of sequential execution without giving up the speed increases of parallel execution. The simplest way to solve our problem above is by using a *Mutex*. A mutex is a synchronization primitive that has the following properties: let a *Critical Section* be the interval of the instruction stream which is surrounded by a mutex *lock* and *unlock*; a mutex ensures non-overlapping execution of critical sections, and in any non-trivial suffix of the linearized instruction stream (that is, the interleaved stream), some *lock* or *unlock* call to the mutex succedes.

10.1.1 Petersons Mutex Algorithm

We show an algorithm for making a mutex supporting two threads using only *Read* and $Write$. We have three shared variables: x, *he_wants*, and *she_wants*. x is the value we would like to use exclusively. In addition, we have a C-style `enum`, *turn*, which is either *his* or *hers*.

Alices code

```
1  she_wants = T
2  turn = his
3  while he_wants and turn=his
4      spin
5  {
6      access(x)
7  }
8  she_wants = F
```

Bobs code

```
1  he_wants = T
2  turn = hers
3  while she_wants and turn=hers
4      spin
5  {
6      access(x)
7  }
8  he_wants = F
```

In terms of *lock* and *unlock*, the code before the curly brackets is *lock*, and the code after is *unlock*. We show that this implements a mutex for two threads, by contradiction. Assume that the two threads are both inside the critical section. Without loss of generality, let Bob be the last writer of *turn*: $W_A(turn) \leq W_B(turn)$. We can clearly see from the code that $W_A(she\_wants) \leq W_A(turn)$, which implies $W_A(she\_wants) \leq W_B(turn)$. Since $W_A(she\_wants) \leq W_A(turn) \leq W_B(turn) \leq R_B(she\_wants)$, B will read $she\_wants = T$. In addition, since we let B be the last writer of *turn*, we know that $turn = hers$, which means that both conditions for the **while** loop is fulfilled. B will spin!. Now B will spin until A has gone out of her critical section, and writes $she\_wants = F$. Then B may enter his critical section, but now A is already done! No overlap of the critical sections, which is a contradiction.

10.2 Lock-Free Algorithms

What does it mean for an algorithm or data structure to be "Lock-Free"? Lock-free code contains special instructions, the most notable of which is the *Compare-and-Swap*, or CAS for short. CAS takes three arguments, an address, an old value and a new value. It checks whether the value stored at the address given is equal to the old value, and if it is, it writes the new value and returns T; if the value is not equal it returns F. This might not seem special at all, but the key is that it happens *atomically*; that is, we do not have the problem from Section 10.1 where one thread may overwrite anothers data.

Example: It is possible to make a lock using the CAS instruction: Note that we do not need to CAS the write at line 7, since the thread from the critical section is the only thread that will change the value of *lock* when it is set to F.

CAS-Lock

```
1  lock = F
2  while !CAS(&lock, F, T)
3      spin
4  {
5      ...
6  }
7  lock = F
```

Example: We make a map-fold style reducer using the CAS instruction. *result* is the shared variable for the accumulated result. We imagine that the threads all get a different slice of the array, which they loop over.

CAS-Lock

```
1  for f in array
2      tmp = compute(f)
3      do {
4          old = Read(result)
5          new = result + tmp
6      } while !CAS(&result, old, new)
```

We prefer lock-free to locking, because lock-free algorithms does not block threads: we cannot rely that all threads get a somewhat balanced running time in an interval; some threads may run for a while, and then stop for a while. If such a thread has aquired a lock before pausing for a long time, all threads which need that lock simply have to wait. One could imagine a scheme which

allows the threads that actually do work to somehow be prioritized over threads that are more idle. In a sense this is what *CAS* allows us to do: all threads may read at any time, and if they are to write, they have to check that nothing has changed during the time from they first read the value to now when they want to write back a new value.

We also note a property of the previous example: it will not block; that is, no execution will cause *result* to stay the same for a long time (given that *some* thread gets execution time). To see why, let us first assume that the domain of *compute* is non-negative. We note that the only reason that the *CAS* should fail is if *result* has been increased by some other thread. So while it is possible that a given thread will loop indefinitely if *result* always keep chaning, this also means that some *other* thread is doing great in computing its values.

10.3 Treiber Stack

We show a lock-free stack implementation know as a *Treiber Stack*. The idea, which is a popular one in lock free data structures, is to build the stack out of a linked. The stack is just a pointer to the head node, which is the top of the stack. A node has a value and a pointer to the next node. The code for PUSH and POP is shown below.

```
Push(v)
1  n = MAKE-NODE(v)
2  do {
3      n.next = Read(head)
4  } while !CAS(head, n.next, n)
```

```
Pop()
1  curr = Read(head)
2  while curr {
3      if CAS(head, curr, curr.next)
4          break
5      curr = Read(head)
6  }
7  return curr
```

The idea of PUSH is to first make the node that is the new head. Then we set the *next* pointer to be the head of the stack, such that we only need to update the stacks head pointer. If this update fails, we update the *next* pointer of our local node, and retry. Intuetivaly we understand that this operation is safe, since the stack itself is not changed until the *CAS* is successful, and we are done.

In POP we read the stacks head pointer, and try to *CAS* the head pointer to the next node in the stack. If we fail, someone else have either inserted or removed an element from the stack, so we have to read the head again. If we succeed, our new node is not reachable for anyone else, so we do not need to do anything special; we can simply return the node.

10.4 Concurrent Lists

Next, we want to make a concurrent list, without locks. We consider an implementation similar to the Treiber Stack, where we make a node on insert, and CAS it into place, and similarily with deletions.

However, with a little bit of imagination, we run into trouble. Consider a list with nodes A, B, C, D, where the first and last node are sentinel nodes, such that nodes with a value never has a *null* pointer, and that the lists head pointer is never a *null* pointer. Assume we want to insert the node B' between B and C. We start out by setting $B'.next = C$. But then another thread comes in and deletes the node B, by CASing $A.next = C$. Then the first thread resumes its execution, and CASes $B.next = B'$, which succedes. The insert will look like it succedes, but the node that precedes the new node in the list is in fact not reachable from the head node!

Another case: we have the same list. Thread 1 wants to delete B, thread 2 wants to delete C. Thread 2 starts out and finds the B node, which it wants to CAS the next pointer on. But then, Thread 1 comes in and CASes $A.next = C$, which succedes. Thread 2 CASes $B.next = D$, and both threads returnes sucecsfully. When we are done, the list is A, C, D, so we have just removed a single element namely B, even though both delete calls was "succesfull".

This illustrates why lock free programming is hard; a seemingly simple task has a lot of pitfalls, and recognizing the pitfalls is not trivial.

What can we do to handle these issues? If we allow *some* locking, we can use the "hand-over-hand" approach with locks. We note that inserting a note just requires locking the nodes around the place where the new node should be. If we want to insert between C and D we can lock A and B, then release A and lock C, and then release B and lock D. Now we know that the *next* pointer of B will not change, and we can safely insert the new node. Note that this also requires locking of delete, but in a slightly different manner: on delete we need to lock the node we are deleting as well as its predecessor, so that we do not risk deleting a node we are using for insertion.

$O(n)$ locks are problematic in terms of real world performance. Is there some way to avoid this? We can search through the list and find the nodes we need, lock them, and then search *again*, to ensure that they are still reachable from the head. Now we are doing a double traversal of the list, but only require two locks. Whether this is faster or slower depends on use case.

Another approach is to use the least significant bits of the pointers to store information; these are usually 0 for alignment reasons. We can then "tag" the *next* pointer of the node we want to use, or delete, in order for other CASes to fail.

Bibliography

[1] Mihai Patrascu and Mikkel Thorup. "The Power of Simple Tabulation Hashing". In: *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*. STOC '11. San Jose, California, USA: ACM, 2011, pp. 1–10. ISBN: 978-1-4503-0691-1. DOI: 10.1145/1993636.1993638. URL: http://doi.acm.org/10.1145/1993636.1993638.