Data structures I – NTIN066

November 21, 2014

Prerequisites

None of these should be any issue for anyone who finished *our* CS bachelor.

Asymptotic notation. We will use it extensively and perform computations in it. We use the master theorem in a proof or two, but I think it might be even enough to read about that on Wikipedia.

Basic data structures. It is essential to understand basics such as linked lists, arrays, stacks (LIFO) and FIFO queues.

Combinatorial basics, mainly to understand properties of binomial numbers (computations in proofs).

More for hashing. In Part IV we will use basic *probability theory*, mainly expected values, variances, (in)dependence, properties of binomial distribution, and likely Markov and Chebychev inequalities (I can state those anyway, perhaps without proof). In a hashing proof we will also likely use some basic properties from (linear) *algebra*, such as modular arithmetic and invertibility of regular matrices over (finite) fields (but again, understanding *why* they are invertible is not needed to understand the proof).

Tentative plan for winter 2014

1. Trees (representing ordered sets): (a, b)-trees, splay trees, overview of other solutions.
2. Heaps (priority queues): regular, binomial and Fibonacci heaps.
3. Memory hierarchies: modelling I/O, cache-oblivious and cache-aware techniques.
4. Hashing (representing unordered sets): collisions, choice of hash function, universal hashing, cuckoo hashing.

Contents

Part I
Trees

1 Common

Comparison-based dictionary

In this part we suppose an arbitrary *universe set* U and focus on representing a subset $S \subseteq U$. We call elements of S *keys*, and denote $n := |S|$.

We require that U is *totally ordered*, i. e. for any pair of keys we can decide whether they are equal or either is smaller. Elements of U are *opaque* to us; we only read, write or compare pairs, e. g. we use no arithmetics on U like in hashing.

The keys can also have some data associated with them. We will not discuss this extension because it is simple to extend the opaque keys with data or a pointer to them, and the change does not affect how the structures behave.

All the common representations use rooted trees with ordered children.

Supported operations

The basic operations supported by all tree types:

- INSERT(T, x) and DELETE(T, x) to modify the set (and also creating a representation of the empty set);
- MEMBER(T, x) which is often called FIND in the usual case when we are interested in the attached data, and e. g. also predecessor/successor queries;
- MIN(T) and MAX(T) and various interval queries;
- building in $\mathcal{O}(n)$ time from a sorted sequence of keys.

Trees can be augmented to support these queries (for example):

- ORD(T, k) finds the k-th smallest key in T (notation: counting from zero);
- RANK(T, x) returns the number of keys less than x (an inverse query to ORD).

Some trees also support fast splitting and joining, which is mostly useful for representing key-less string-like sequences.

- SPLIT(T, x) splits the represented set by x into two trees T_1 and T_2. The resulting trees contain the keys from T that are less than x or greater than x, respectively.

- JOIN2(T_1, T_2) and JOIN3(T_1, x, T_2) are the inverse operations to SPLIT. We require that MAX$(T_1) <$ MIN(T_2) and the result is a tree that represents all the keys from T_1, T_2 and also x in the case of JOIN3. In various structures we only show the simpler of the operations and the other one can be simulated by adding an extra insertion or deletion.

2 (a,b)-trees

The (a, b)-trees are rooted trees where:

- all leaves are in the same depth, one for each represented key;
- inner nodes have between a and b child nodes (inclusively), and
- the root node has between 2 and b child nodes (unless the tree is a single leaf).

Remark (practice, caches). In practice we might rather represent between c and d keys in one leaf node, but that is an easy modification of the structure. Good choice of the constants depends on the particular use case, and they will typically grow with relative cost of random access (which is high e. g. for rotating disks).

Literature

Here we study general (a, b)-trees. In literature one can rather find variants of B-trees instead, but the principles are always the same. Some basic treatment of B-trees is present in practically any data-structural textbook, including *Handbook of Data Structures and Applications* and *Introduction to Algorithms.*

Standard B-trees also use our splitter positions for keys, which slightly complicates the structure and typically worsens performance on disks, because the inner nodes need more space due to additional data attached to the keys. The standard B^+-trees are leaf-oriented as our (a, b)-trees, and moreover they add level-links on the leaf level (we get more from level-links in the finger-tree section on page 10).

Lemma. *The depth of any (a, b)-tree with n leaves is between $\log_b n$ and $1 + \log_a \frac{n}{2}$. That is, the depth is $\Theta(\log n)$.*

Proof. Obviously, the number of leaves of an (a, b)-tree of height h is between $2a^{h-1}$ and b^h. By taking logarithms we get the claimed inequalities. □

Lemma. *The tree uses less than n nonleaf nodes, so it needs space is $\Theta(n)$.*

Proof. On every level above leaves the number of nodes gets halved at least, so it is at most $\sum_{i=1}^{\text{depth}} n/2^i < n \sum_{i=1}^{\infty} 1/2^i = n$. □

Every nonleaf node v of the tree consists of the following fields:

- $p(v)$ is the number of child nodes of v.
- $S_v\left[1 \ldots p(v)\right]$ is an array of pointers to child nodes.
- $H_v\left[1 \ldots p(v)-1\right]$ is an array of splitters. They are keys such that $\forall i$ $\textsc{max}(S_v[i]) \leq H_v[i] < \textsc{min}(S_v[i+1])$. Note that the H_v values in the tree recursively and unambiguously split the whole universe set into intervals, which are 1-to-1 mapped to leaves.

In the usual variant of (a,b)-trees we need that $a \geq 2$ and $b \geq 2a-1$. However, it is recommended to have $b \geq 2a$, which significantly decreases the total number of needed modifications in long operation sequences.

2.1 Standard algorithms

The algorithms for MEMBER, INSERT and DELETE always do at most constant amount of work on every level of the tree, bounding the asymptotic complexities by $\mathcal{O}(\log n)$.

Member(T,x) We start in the root and repeatedly choose the correct child to go down (by the values of H_v and x). In the leaf we compare the key with x.

Insert(T,x) We find the correct leaf to insert after and check that x is not represented by T yet. Adding a new leaf can overfill the parent; in that case we split it in halves, which results into adding a new child of the grandparent, etc. Note that during the node split we move one splitter to the parent. The splitting can cascade up to the root, whose splitting would increase the depth of the tree. Note that $b \geq 2a-1$ ensures exactly that the split nodes always have enough children.

Delete(T,x) We find the leaf to be deleted and remove reference to it from the parent node. Whenever a nonroot node is only left with $a-1$ children, we merge it with its brother. If the resulting node has more than b children, we re-split it in halves. Then we correct the common parent of the nodes, possibly resulting into a cascade of merges. If the root node is only left with one child, we make the child the new root (decreasing the depth of the tree).

Exercise. Start with an empty 2, 3-tree and succesively perform the following operations. Insert 7, 2, 3, 0, 6. Then delete 2, insert 4 and 8, and delete 0, 6, 7.

2.2 Splitting and joining

Join2(T_1,T_2) We take the lower of T_1 and T_2, and merge its root with the adjacent border node in the correct depth of the other tree, so the leaves are in the same depth. Then we fix the node the same way as after insertion. If we know a splitter to put between the two trees and we know their depth difference, it is enough to work on as many levels, resulting into $\mathcal{O}(|h(T_1) - h(T_2)| + 1)$ complexity.

Split(T,x) We start with two empty stacks for left and right subtrees, S_1 and S_2. Then we proceed like in MEMBER(T,x), but we split every node according to the chosen child into the left and right fragment (one of those can be empty), pushing them on the corresponding stacks. Finally we repeatedly JOIN2 the top trees on the left stack which creates T_1; T_2 is created from the right stack the same way.

It remains to prove joining each stack only takes $\mathcal{O}(\log n)$ time. Note that the heights of the fragments pushed onto each stack strictly decrease, and they are correct (a,b)-trees, except that some fragments can have a root with a single child. That violation does not affect the JOIN2 operations, and we can easily delete the degenerate root after joining the whole stack.

Note that JOIN2(T_L, T_R) can only create a tree of depth $\max\left\{h(T_L), h(T_R)\right\}$ or one level higher. Let $h_1 < h_2 < \cdots < h_k$ denote the heights of trees on the stack and let h'_i denote the resulting height after joining together the first i trees. Notice that $h'_i \leq h_i + 1$, by induction:

1. $h'_1 = h_1$ by definition;
2. for $i > 1$, $h'_i \leq \max\left\{h'_{i-1}, h_i\right\} + 1 \leq \max\left\{h_{i-1}+1, h_i\right\} + 1 \leq h_i + 1$

The total work of joining a stack is bounded by $W \leq \sum_{i=1}^{k-1} \mathcal{O}(|h_{i+1} - h'_i| + 1)$. Since $h_{i+1} \geq h_i + 1 \geq h'_i$, we can remove the absolute value and get

$$W \lesssim \sum_{i=1}^{k-1} h_{i+1} - \sum_{i=1}^{k-1} h'_i + k - 1 \leq \sum_{i=2}^{k-1} h_i + h_k - \sum_{i=2}^{k-1} h'_i + k.$$

It trivially holds that $h'_i \geq h_i$, so the sums cancel out and we get $W \leq \mathcal{O}(\log n)$.

2.3 Order statistics

To suport the ORD and RANK operations, we need to add a field $N_v\,[1 \ldots p(v)-1]$ into every node, where $N_v[i]$ is the number of leaves in the $S_v[i]$ subtree. It is quite clear how to change the mentioned operations to also maintain the N_v values, especially note that we can reconstruct the values in a node from the values in its children in constant time (if the children are already correct).

RANK(T, x) We proceed like in MEMBER(T, x), but we also accumulate all the N_v values to the left of the chosen path.

ORD(T, k) We find $\max\left\{i \in \{1 \ldots p(root)\} \mid \sum_{j=1}^{i-1} N_{root}[j] < k\right\}$ and we continue as in ORD$\left(S[i], k - \sum_{j=1}^{i-1} N_{root}[j]\right)$.

Remark. This augmentation can be generalized in a stragihtforward way. We showed the case of accumulating 1-values assigned to every leaf, combined by addition. Every leaf can have any value selected on insertion, and we may allow to modify it. For meaningful RANK we only need that the chosen combining operator is associative. For unambiguous selection with ORD according to a predicate, we need that the predicate can not be turned from true to false by accumulating more values by the operator.

We can also easily keep more independent order statistics at once. The generalizations get more useful for finger-tree variant without keys (see pg. 10).

2.4 Top-down balancing

The stated insertion and deletion algorithms are not suitable for parallel execution. They might have to lock the root during the whole operation, because we do not know for sure if it will be modified. Note that this uncertainty only happens when the node is *critical*, i. e. when it has b children on insertion or a children on deletion.

We can shorten the locks by *top-down balancing*: whenever we pass through a critical node, we split/merge it to be safe (before continuing down). Since the update can not propagate above a non-critical node, we can release a lock on its parent before continuing down. That way each updating thread at any time only locks one node and optionally its parent.

We need $b \geq 2a$ to maintain the invariant, and it is recommended to have $b \geq 2a + 2$.

2.5 Finger trees

It is common for ordered-set data structures to support some efficient way of accessing elements near to a known location. Such structure variants are usually called finger trees. Now we show how to modify (a, b)-trees to support finger operations.

For a tree T and finger f we aim for operation complexities proportional to $\log(|\mathrm{ORD}(T, f) - \mathrm{ORD}(T, x)| + 2)$. That is asymptotically at most the previous $\log n$, but it can be much less if f points "near x".

Finger will be just a pointer to an arbitrary leaf. For easy access from there we need to add some pointers to every node of the tree: parent-node pointer, and pointers to the previous and following node on the same level. All modifying operations can be easily adapted to maintain these additional pointers without worsening the asymptotic time or space.

Operations MEMFROM(f, x), DELFROM(f, x) and INSFROM(f, x) work almost the same as their non-finger counterparts. The main difference is in finding the correct leaf – we start from the finger f and always alternate one step on the same level towards x with one step up (into the parent), until we skip over the value of x. Then we go down just like when searching for x.

Observe that the i-th level-step skips at least a^{i-1} leaves. As a consequence the time to find the leaf is proportional to $\log_a(|\mathrm{ORD}(T, f) - \mathrm{ORD}(T, x)| + 2)$. In the modifying operations we also have to handle node spliting or merging, which does not change the situation much (without detailed proof here).

Exercise. Explain why we needed to add the level pointers.

Fingers help navigate the tree fast, but modifying operations can generally cascade up to the root. The following shows the work will amortize on series of operations.

Theorem. *Let $b \geq 2a$, $a \geq 2$ and T be a (finger) (a, b)-tree representing a set of size at most n during a series of lookups, insertions and deletions. Then the total time needed to apply the series on T is $\mathcal{O}(\log n +$ time needed for searching$)$.*

Remark (dictionaries without keys). Surprisingly, in many use cases it makes sense to use trees without keys, only representing sequences of values (stored in leaves for the case of (a, b)-trees). A typical programming example is *string* of values. Then the ordering is given implicitly by the operations – searching by key makes no sense, and most operations need a finger pointing to the exact place where the insertion/deletion/splitting/rank should happen. Note that splitters in inner nodes would also make no sense anymore. It is useful to combine this with order (or other) statistics. For example, such structures are important building blocks for representations of dynamic graphs, such as ET-trees and ST-trees.

2.6 A-sort

A-sort is a sorting algorithm that is efficient for almost sorted sequences. It uses an (a, b)-tree with parent pointers. We start with an empty tree and successively insert all elements. We always search for the position from the rightmost leaf (we have a finger there). Finally we walk through the leaves and output them in order (in linear time).

Note that the number leaves skipped on i-th insertion is equal to the number of inversions of the i-th element. That is defined as the number of preceding elements that are smaller, i. e. $f_i := |\{j : j > i \wedge x_j < x_i\}|$. For the total number of inversions $F := \sum_i f_i$ we know $0 \leq F \leq \binom{n}{2}$.

The total work spent on node splitting is $\mathcal{O}(n)$, because we only do insertions, and splitting a node increases the number of nodes by one, and the final number of nodes is $\mathcal{O}(n)$. The time spent on searching is proportional to $\sum_i \log(f_i + 2) = \log \prod_i (f_i + 2) = n \log \left[\prod_i (f_i + 2)\right]^{1/n}$. Since $\left[\prod_{i=1}^n \alpha_i\right]^{1/n} \leq \frac{1}{n} \sum_{i=1}^n \alpha_i$ (geometric vs. arithmetic mean of nonnegative values), we get total time bound $\mathcal{O}(n \log(2 + F/n))$. Note that this is $\mathcal{O}(n \log n)$ in the worst case, but it can be much better if the sequence is partially sorted. The authors also show that the time bound is asymptotically optimal w.r.t. F: they compute the number of permutations for given F and use the decision-tree technique described on page 52.

3 Binary search trees

We define BSTs as rooted trees with the following properties:

- Every nonleaf v has (exactly) two child nodes $\mathsf{l}(v)$, $\mathsf{r}(v)$ and it is uniquely assigned to an element $\mathsf{key}(v)$ of the represented set.
- Leaves carry no information. In practice they would not be present, i. e. pointers to leaves are typically *null* pointers.
- The keys are in infix ordering, i. e. all keys in subtree of $\mathsf{l}(v)$ are less than $\mathsf{key}(v)$ and all keys in $\mathsf{r}(v)$ are greater.[1]

Note again that every vertex covers an open interval from the ordered universe set – the root starts with the whole universe, then the subtrees always split the parent interval according to the key, and the leaves exactly correspond to all intervals between the represented elements.

Corollary 3.1. *BST representing n keys contains one nonleaf vertex for every key and $n + 1$ leaves, so it needs $\Theta(n)$ space.*

Literature

Handbook of Data Structures and Applications covers in detail all we do and more, including AVL trees.

Introduction to Algorithms, second or third edition, explains BSTs without balancing and red-black trees. Splitting, joining, and splay trees are omitted.

3.1 BSTs without balancing

Operations:

MEMBER/FIND(T, x) We start from the root, always compare the key and either return success or continue down the correct child node.

MIN(T) and **MAX**(T) We just find the leftmost or rightmost non-leaf node.

INSERT(T, x) We start the same as MEMBER(T, x). If we find x, we exit. Otherwise we replace the leaf that we ended in with a new node (having key x and two leaf children).

[1] We may want to allow equality in subtrees to support multiple values with the same key. That often makes sense in practice, but we do not complicate the descriptions here.

DELETE(T, x) We first find the node u with $\mathsf{key}(u) = x$ or we fail. If both children of u are leaves, we just replace u with a leaf node. Similarly if only one child is non-leaf, we replace u with it. Otherwise we find v that contains the left neigbour key in the ordering. That is the rightmost non-leaf node in the subtree of $\mathsf{l}(u)$. We move $\mathsf{key}(v)$ to u and remove v instead, which is simple because $\mathsf{r}(v)$ has to be a leaf.

JOIN3(T_1, x, T_2) is trivial exercise.

SPLIT(T, x) We proceed like in MEMBER(T, x) building T_L and T_R along the way, similarly to (a, b)-trees. We maintain pointers to the rigtmost leaf of T_L and leftmost leaf of T_R. We always replace these leaves by the left or right subtrees that we cut along the way down in T. When we reach leaf, we have distributed all vertices in T_L and T_R.

RANK(T, x) and **ORD**(T, k) We do the same as in (a, b)-trees. We also need to add a counter of keys present in every subtree.

Exercise (easy)**.** Find linear algorithms for building a perfectly balanced tree from a sorted array or list, and for printing all represented keys in sorted order.

Keeping BSTs fast

Note that again the complexity of any operation is proportional to the depth of the tree. The depth is expected to be logarithmic, assuming uniformly random distribution of arguments to modifying operations, but that is a very unrealistic assumption. There are many modifications that strive to bound the complexities in various ways:

- *Balancing* the tree ensures $\mathcal{O}(\log n)$ depth. We cover Red-black trees on the following page. AVL trees are skipped now, but still described on page 47.
- *Randomizing* the modifying operations to make the expected complexities *independent* of the sequence of modifications performed on the tree (covered in DSII course). The approach is alike to universal hashing, described on page 40.
- *Self-modify* the tree to optimize for some operation patterns. We cover Splay trees.

Balancing BSTs can be balanced by edge *rotations*. A single rotation reverses the direction of one edge. There is only one way of doing it, because we must not disturb the infix order of the nodes. Let WLOG rotate the edge $\mathsf{l}(v) = u$. After the rotation, u takes the place of v (from the perspective of the parent), and we have $\mathsf{r}(u) = v$. Also, the right subtree of u becomes the left subtree of v.

3.2 Red-black trees

RB-trees are BSTs where every vertex is either red or black, and the following conditions hold:

1. leaves and the root are black,
2. red vertex always has a black parent, and
3. all paths from root to a leaf contain the same amount of black vertices.

Now we show how these conditions bound the height of the tree. We consider the shortest and longest root-to-leaf paths possible in one RB-tree. They have the same number of black vertices, all start and end in a black vertex, and at most every other vertex can be red. Thus the ratio of the longest to the shorterst path is at most 2. If we combine it with the following lemma, we get that the *depth is bounded by* $2\log(n + 2)$.[2]

Lemma. *The shortest root-to-leaf path in a tree with n nonleaf vertices has less than $\log(n + 2)$ edges.*

Proof. For the sake of contradiction suppose that all paths have length at least $\log(n + 2)$. Then there is no leaf up to the depth $\lceil\log(n + 2)\rceil - 1$ (inclusive), so the number of nonleaf vertices is at least $\sum_{i=0}^{\lceil\log(n+2)\rceil-1} 2^i = 2^{\lceil\log(n+2)\rceil} - 1 \geq n + 1 > n$. □

Non-modifying operations are the same as in BSTs without balancing. In INSERT and DELETE we just perform some rebalancing at the end of the operation to restore the RB conditions.

Insert

The BST algorithm always replaces a leaf with a new vertex which we make red. This can create a **2-partial** RB-tree, where all conditions hold except for one red-red edge. Let us denote the edge $(u \to v)$; WLOG $v = \mathsf{r}(u)$, as the other case is just mirrored.

If u is the root, we just color it black and we are done. Otherwise let t be the parent of u, and w the sibling of u. We know that u and v are red; t is black because 2-partial RB-tree only allows one red-red edge. Three cases follow; in each we do some rotations and/or recolorings that either fix the tree or make it 2-partial where the broken edge is higher than before.

[2]All logarithms are binary, unless specified otherwise.

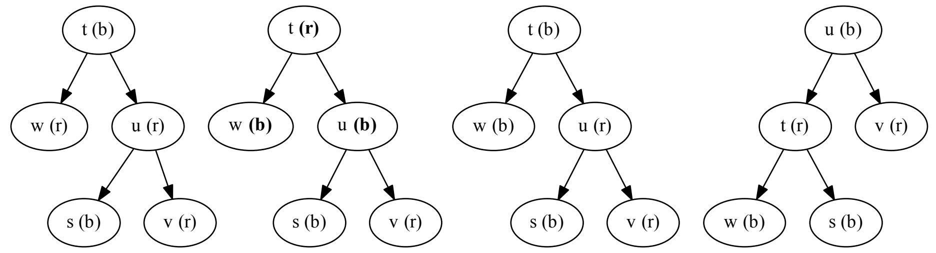

Figure 3.1: RB insertion: Cases 1 and 2. Note that for Case 1 the subtrees of w and u may look the other way around. As for all RB figures, the nodes can have larger subtrees which are not indicated here.

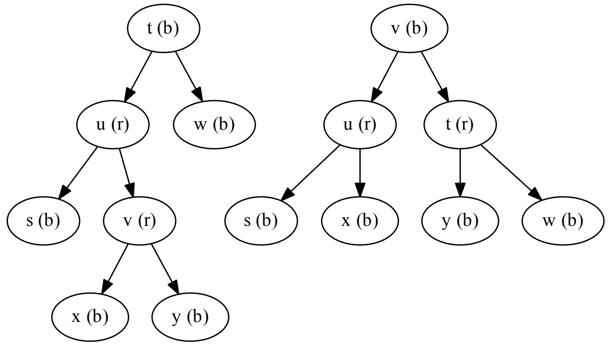

Figure 3.2: RB insertion: Case 3.

Case 1. w is red. We recolor the siblings u and w to black, and change their parent t to red. This may break the edge between t and its parent, so we may have to continue there (two levels higher).

Case 2. w is black and it is the left sibling of u. We rotate the $(t \to u)$ edge; then recolor t to red and u to black. We can stop here, because the root of the rotation remains black.

Case 3. w is black and it is the right sibling of u. We double-rotate v up, and recolor t to red and v to black. We can stop there, because the root of the rotation remains black, as in the previous case.

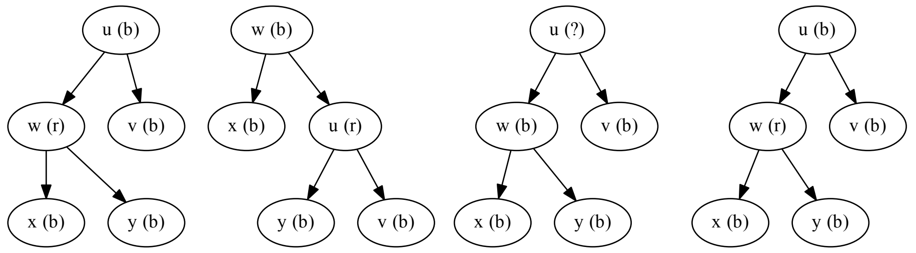

Figure 3.3: RB deletion: Cases 2 and 3.

Delete (no details during the lecture)

The BST algorithm always ends with removal of a node that has (at least) one leaf child. If the removed node was black then we are left with a **3-partial** RB-tree, where all conditions hold except that root-to-leaf paths through one vertex v have one less black node than all the other root-to-leaf paths.

If v is the root, we are done. Otherwise let u be the parent of v, w be the other child of u, and WLOG let $v = \mathsf{r}(u)$ (the other case is symmetrical). Five cases follow; in each we do some rotations and/or recolorings that either fix the tree or make it 3-partial where the broken vertex is higher than before.

Case 1. v is red. We just change v to black, which fixes everything.

Case 2. the sibling w is red (and thus all around are black). We rotate the $(u \to w)$ edge, recolor u to red and w to black. The result is still 3-partial in v, but now its sibling is black and we continue with some of the following cases.

Case 3. the sibling w and both its children are black. We recolor w to red and set the parent u to black. If u was red, the problem is fixed (e. g. always after Case 2); otherwise we now have a 3-partial tree in u and continue one level higher.

Case 4. the sibling w and its right child are black, and w's left child x is red. We rotate the $(u \to w)$ edge, recolor u and x to black, and w retains the former color of u. The tree is OK now.

Case 5. the sibling w is black and its right child y is red. We let y double-rotate up and get the former color of u; u gets black. The tree is OK now.

Exercise. Verify that no operation can do more than three rotations (a double rotation equals two regular ones).

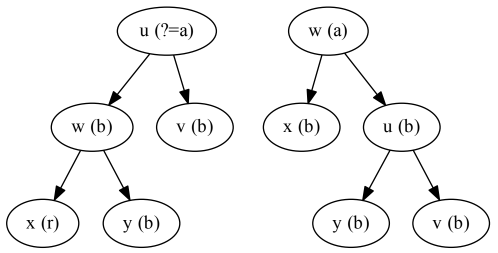

Figure 3.4: RB deletion: Case 4.

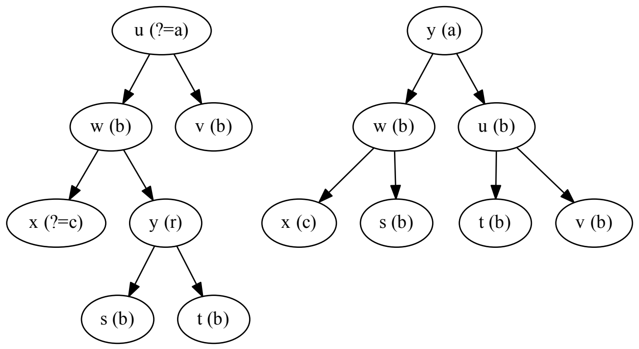

Figure 3.5: RB deletion: Case 5.

Notes

- The RANK and ORD operations can be added the same way as in BSTs without balancing.
- JOIN3 and SPLIT work like in (a, b)-trees, only the depth counts in black vertices. JOIN3(T_1, x, T_2) finds an edge of the deeper tree in correct black-depth, and makes x subdivide the edge with the less deep tree as the other child.

Corollary. *The complexities of all operations on RB-trees are* $\Theta(\log n)$.

Comparison to AVL

- The worst-case depth (up to small additive constants) is $\doteq 1.4 \log n$ in AVL and $2 \log n$ in RB, but on random data they are almost the same.
- RB typically do less work on modifications. In particular, they always do at most three rotations for any operation, whereas AVL deletion can do $\Omega(\log n)$ rotations. That gives RB-trees an advantage especially in applications with expensive rotations, e. g. when much data is associated with nodes and need to be recomputed on restructuring.
- RB seem to be the most popular solution for implementation of a dictionary in general-purpose programming libraries. That holds even in cases where only the basic operations are used, so e. g. hash tables would be more efficient. Arguably, it is because there are more pitfalls in hashing, and $\log n$ is bounded by a not-too-large constant in practice (e. g. we can hardly hope to ever store 2^{100} elements).

3.3 Amortized complexity

Amortization is not a tree-specific topic; it is useful in all kinds of data structures, including the following subsection. You can find more detailed explanation e. g. in *Handbook of Data Structures and Applications* or *Introduction to Algorithms*.

Motivation: Often we are not much interested in complexities of individual operations but rather complexities of their *sequences*. Thus we may tolerate doing a lot of work once upon a time if we know that this has saved us at least that much work in the past.

During an amortized analysis we define an amortized complexity for each operation. The only real requirement is that for any sequence of operations starting from a well-defined state (e. g. an empty structure), the sum of actual work done is no larger than the sum of their declared amortized complexities.

Amortization resembles some kind of expected complexity for one operation, but note that there is *no probability* involved here. Also note that the operations in the sequence do not need to have the same amortized complexities.

Analysis via potentials

Amortized complexity bounds are typically proven by a variation of the following method. We assign a *nonnegative potential* to the whole structure. The potential represents how much of the allowed work has been saved for future, and it is often defined as the sum over potentials of all parts of the structure. (Sometimes it is imagined as savings in coins and called *banker's method*.) In most cases the potential does not need to be explicitly stored in the structure, and it is only used to analyze the complexity.

Let us analyze some operation that transforms the structure from potential Φ_{i-1} to potential Φ_i and needs at most W_i real work at the moment. Then we want to have amortized complexity W_i^A of the operation such that $W_i + \Phi_i - \Phi_{i-1} \leq W_i^A$. (The potential is always scaled in some units of work.)

If we prove the above property, the desired bound is ensured. Take any sequence of k operations where the potentials went from Φ_0 to Φ_k, and i-th operation needed W_i real work and had W_i^A amortized complexity. Then the total work is:

$$\sum_{i=1}^{k} W_i \leq \sum_{i=1}^{k} \left[W_i^A - (\Phi_i - \Phi_{i-1})\right] = \sum_{i=1}^{k} W_i^A + \Phi_0 - \Phi_k$$

Since $\Phi_k \geq 0$ and Φ_0 is a constant (usually zero), we get the desired bound.

Trivial example: extending an array by doubling when full. We can choose potential as the number of occupied slots minus the number of free ones.

3.4 Splay trees

- Splay operation: rotates a given key to the root (in a specific way).
- All other operations are trivially converted to splaying.
- No explicit balancing; restructuring only by splaying the accessed nodes.
- Operations can take $\Omega(n)$, but they are all $\mathcal{O}(\log n)$ amortized (starting from an empty tree) and even better for specific access patterns.
- Advantages: simple and automagically taking advantage of various kinds of access patterns.
- Disadvantages: very many rotations, even when seaching only, and thus slightly higher multiplicative constants.

Splay algorithm

When splaying vertex u, we repeat the following until u is in root:

Case 1. u has no grandparent. We just rotate the edge to the parent.

Case 2. (zig-zig) the key of u is *not between* its parent's and grandparent's. We rotate the edge to the parent of u and then the edge to its new parent. See Figure 3.6.

Case 3. (zig-zag) Otherwise, we first rotate the edge between the parent and grandparent of u, and then rotate the edge to the new parent of u. See Figure 3.7 on the facing page.

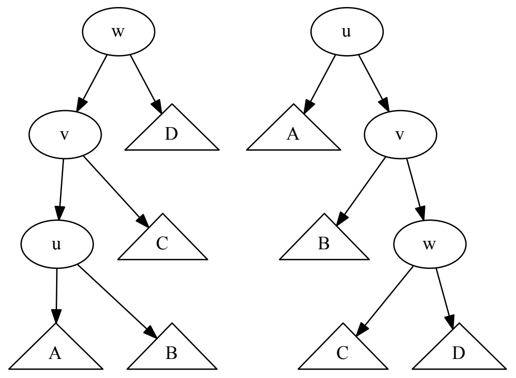

Figure 3.6: Splay step, Case 2 (called zig-zig). The case can also be mirrored.

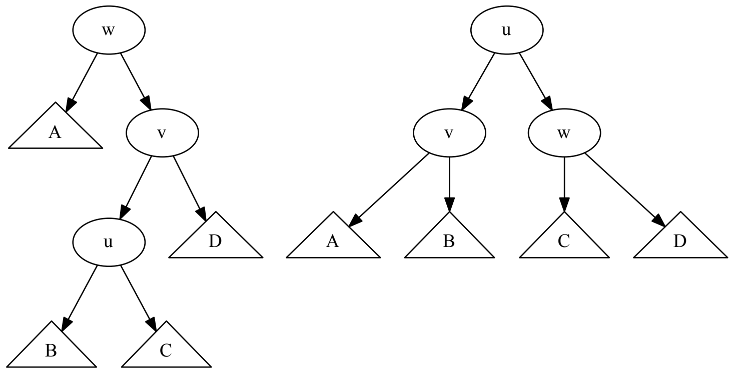

Figure 3.7: Splay step, Case 3 (called zig-zag). The case can also be mirrored.

Analysis of splaying

- For each key we choose a weight, in our case always equal to one.
- For each vertex u we define rank $r(u)$ as the logarithm of the total weight in the subtree of u (scaled to suitable units of work time).[3]
- Use the potential $\Phi := \sum_u r(u)$. Note: it is always nonnegative.
- Aim show that the amortized cost of splaying u is $\lesssim 1{+}3\,(r(\mathrm{root}) - r(u))$.
- Analyze each step of the algorithm separately; they telescope.
- Notation: splaying vertex u with parent v and (possibly) grandparent w. Let r' be rank after the step and t the amortized time of the step.
- Notes: weights and thus ranks can change only in those three vertices. Similarly, root of the (double) rotation preserves its rank. Any vertex has larger rank than any of its descendants.

[3]The original definition from Tarjan used rounded integer ranks instead, but that unnecessarily complicates the analysis.

Case 1. $t \approx 1{+}r'(u){-}r(u){+}r'(v){-}r(v) \;=\; 1{-}r(u){+}r'(v) < 1{+}r(\mathrm{root}){-}r(u)$.

Case 2. We count the rotations as two units of work and get:

$$\begin{aligned} t \;&\approx 2 - r(u) + r'(v) - r(v) + r'(w) \;< \\ &< 2 - r(u) + r'(u) - r(u) + r'(w) \;\leq^{(*)}\; 3r'(u) - 3r(u). \end{aligned}$$

Claim (*). $2{+}r(u){+}r'(w) \leq 2r'(u)$. First note that due to logarithm being concave, $2\log(p+q) \;=\; 2\log\frac{p+q}{2} + 2 \;\geq\; \log p + \log q$.

(Draw pictures.) Let us denote the subtree weights:

$$\begin{aligned} r(u) &= \log\left(|A| + |B| + 1\right), \\ r'(w) &= \log\left(|C| + |D| + 1\right). \end{aligned}$$

Now we have:

$$\begin{aligned} 2r'(u) \;&=\; 2\log\left(|A| + |B| + 1 \;+\; |C| + |D| + 1 \;+ 1\right) \;\geq \\ &\geq\; 2 + \log\left(|A| + |B| + 1\right) + \log\left(|C| + |D| + 1\right) \;= \\ &=\; 2 + r(u) + r'(w). \quad \square \end{aligned}$$

Case 3. It is similar to Case 2:

$$\begin{aligned} t \;&\approx 2 - r(u) + r'(v) - r(v) + r'(w) \;< \\ &< 2 - r(u) + r'(v) - r(u) + r'(w) \;\leq^{(*)}\; 2r'(u) - 2r(u). \end{aligned}$$

Claim (*). $2 + r'(v) + r'(w) \leq 2r'(u)$.
Proof: analogous to the corresponding claim in Case 2.

The cases together finish the splay analysis. With unit key weights, the ranks are between zero and $\log n$, so splay suffices with $\mathcal{O}(\log n)$ amortized time.

Exercise. Formulate algorithms for all operations, and prove their $\mathcal{O}(\log n)$ amortized complexities thoroughly. Note that other actions than splaying can change the potential, which has to be accounted into the complexity.

Properties and conjectures

Splay trees exhibit asymptotic optimality in many situations. The following properties are typically proven just by clever setting of the weights during the analysis (the structure itself is unchanged).

Sequences of operations Consider a sequence of operations on a splay tree. If it starts on an empty tree, the starting potential is zero, so the real work is $\lesssim \sum_i \log n_i$ where n_i is the size after i-th operation. If the sequence starts with *any* state of the tree with n_0 elements, its potential is $0 \le \Phi \le n_0 \log n_0$; therefore, the real work on the sequence is $\lesssim n_0 \log n_0 + \sum_i \log n_i$.

Static optimality Splay trees are asymptotically at least as fast as the best static BST constructed for the query sequence given in advance, even though the splay trees do not know the access pattern in advance.

Static finger optimality Imagine a finger f on any static value from the key universe. Then accessing x only needs the same amortized time as a finger tree would, i. e. $\lesssim \log(1 + |\textsc{ord}(x) - \textsc{ord}(f)|)$. Note that again the splay tree is oblivious to the position of the imagined finger.

Working set theorem If we accessed t_x different elements since the last access to x, then the amortized time to access it again is $\lesssim \log(1 + t_x)$.

Dynamic optimality conjecture Informally: splay trees might be asymptotically optimal among any kind of binary search trees. This *conjecture* has been open for over thirty years, and it is not known to hold for any BST. There are specially crafted Tango trees, which are at most $\mathcal{O}(\log \log n)$-times worse than the optimum.

Part II
Priority queues (heaps)

4 Common

Priority queue interface

Priority queues, often called heaps, maintain a collection of some data where every element is assigned a *priority* (from a totally ordered universe U).

Basic operations we want to support:

- $\textsc{insert}(H, x)$ adds an element with priority x to H,
- $\textsc{min}(H)$ returns an element with the smallest priority in H, and
- $\textsc{delMin}(H)$ removes $\textsc{min}(H)$ from H.

PQs often also support some of the following operations:

- $\textsc{delete}(H, p)$ removes from H an element to which we have a pointer p,
- $\textsc{increase}(H, p, \delta)$ and $\textsc{decrease}(H, p, \delta)$ change priority of an element,
- $\textsc{makeHeap}(A)$ creates a heap from an (unordered) list of elements (sometimes it is even *asymptotically* faster than building by insertions), and
- $\textsc{meld}(H_1, H_2)$ merges two heaps into one that contains the (disjoint) union of the elements (no restriction on priorities like in $\textsc{join}$ for BSTs).

Note (min/max). We will only consider min-heaps; you can surely imagine the reversed variants, and there are simple ways of converting heaps to support both sets of min- and max-oriented operations.

Overview

The following sections describe increasingly complicated PQ data structures. As shown in Table 4.1 on the next page, the later heaps improve asympotic complexities of some operations; on the other hand, the constant factors increase, and also the algorithms and analyses get more complicated. In practice the multiplicative constants are very important, so it is usually best to choose the simplest data structure from those that satisfy the best asymptotic behaviour on the frequently used operations.

Notation. The current number of elements in the heap is denoted by n.

	INSERT	MELD	DEL(MIN)	INCREASE	DECREASE
$\mathcal{O}(1)$-regular	$\log n$	(n)	$\log n$	$\log n$	$\log n$
leftist*	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
binomial	$1 \mid \log n$	$\log n$	$\log n$	$\log^2 n$	$\log n$
lazy binomial	1	1	$\log n \mid n$	$\log^2 n$	$\log n$
Fibonacci	1	1	$\log n \mid n$	$\log n \mid n$	$1 \mid n$

Table 4.1: Comparison of Θ-bounds on operation speeds of various heap types. Cells with two values show the amortized and real-time bound (if they differ). (*) Leftist heaps are currently omitted from the lecture, but you can find them in Section 17 on page 57.

Literature Both *Handbook of Data Structures and Applications* and *Introduction to Algorithms* contain descriptions for all topics done in this part but for a few exceptions. Both only consider 2-regular heaps instead of *d*-regular. The latter textbook removed binomial heaps in its third edition. The first one also contains leftist heaps, which are currently omitted from the lecture.

5 Regular heaps

Definition

For a parameter $d \geq 2$, a *d-regular heap* is a d-ary rooted tree with 1-1 mapping between vertices and the stored elements. As a consequence the space is $\Theta(n)$. The classic variant is $d = 2$.

The heap condition: priority of any vertex is at least the priority of its parent. As a consequence the root contains an element with the smallest priority.

The shape of the tree: it is perfectly balanced, and the leaf level is aligned to the left. Note that the shape is *exactly determined* just by n.

Array implementation (storing vertices in one big contiguous array):

As the tree shape is clear, we store *no pointers*: let the root have the index 0; for a vertex of index i, we give its d children the indices $di + 1, \ldots, di + d$. Consequently, index of the parent can be always computed by $\lfloor (i-1)/d \rfloor$.

We can verify that there are no "holes" between the levels, so the rule gives a 1-1 mapping to indices $0, \ldots, n-1$. Via a simple induction on the indexing definition, the position of the first vertex in depth l is $\left(d^l - 1\right)/(d-1)$. That corresponds to the fact that there are d^l elements on a full level l.

As a consequence, we see that the index of the last element $= n - 1 \geq \left(d^{\text{height}} - 1\right)/(d-1) \geq d^{\text{height}-1}$, so height $\leq \lfloor \log_d n \rfloor + 1$ (assuming $n \geq 1$).

Note. The array implementation is very simple and significantly decreases multiplicative constants in comparison to other heaps (which by default allocate one node per element or more). It is perhaps the strongest reason why the binary heaps are so popular in practice.

5.1 Algorithms and running time

First we define auxiliary UP and DOWN operations.

UP(H, p): It applies if the priority on position p got decreased (but otherwise the heap was OK). We start at p and compare to its parent. We end if it is OK, otherwise we swap them and continue one level higher. Note that these swaps can not break the heap property with the other children of the parent. The needed time is obviously proportional to the depth of p (in the worst case).

DOWN(H, p): It applies if the priority on position p got increased (but otherwise the heap was OK). We start at p and find a child c with the *smallest* priority (among p's children). We end if $p \leq c$, otherwise we swap p with c and continue from the former position of c. Note that these swaps restore the heap property in p. The needed time is obviously proportional to $d \times$ depth of p.

INSERT(H, x): The shape determines the place for any new element (end of array), so we put it there. Then we run UP on it to restore the heap property.

DELETE(H, p): The shape determines the element to be de-allocated, so we first switch it with p. Then we run UP or DOWN on the element to restore the heap property. (The priority on the place can increase or decrease.)

DELMIN(H) and **MIN:** The first element of the array is an element with the smallest priority. We can delete it by the general algorithm.

INCREASE(H, p, δ) and **DECREASE(H, p, δ):** We just change the priority and then fix the heap property by DOWN(H, p) or UP(H, p), respectively.

makeHeap(A)**:** We take the array as the heap. Then we run down on every nonleaf vertex, from the end of the array to the beginning (bottom-up).

Analysis: let h denote the height of the heap. The down operations need less time in the lower part of the heap. We split away the time for running down on elements of level $h-1$ and bound the total time of makeHeap by

$$\begin{aligned}\mathcal{O}(n) + \sum_{l=0}^{h-2}\left[d^l \cdot \mathcal{O}(d\,(h-l))\right] &\overset{l':=h-2-l}{\lesssim} n + \sum_{l'=0}^{h-2} d^{h-2-l'} d\,(l'+2) \leq \\ &\leq n + d^{h-1}\sum_{l'=0}^{h-2}(l'+2)/d^{l'} \lesssim \mathcal{O}(n)\end{aligned}$$

where the last inequality follows from $n > d^{h-1}$ and from the fact that the last sum is bounded by a constant independent of d ($d \geq 2$, even if summed up to infinity).

Note (alternative asymptotic analysis). We can magine the process as a top-down recursion. First the two subtrees of the root are made correct heaps, and then the root is propagated down a path in either subtree. To simplify, assume the whole tree is complete, i. e. all leaves are in the same depth; that can only increase the time, and the resulting number of elements at most doubles to $n' < 2n$. For the total time we have $T(n') \leq 2T(n'/2) + \Theta(d \log_d n')$, so by the *master theorem* (assuming $d \in \mathcal{O}(n^{1-\epsilon})$ with $\epsilon > 0$) we have $T(n) \leq T(n') \leq \Theta(n') = \Theta(n)$.

Corollary. *d-regular heaps need $\Theta(n)$ space;* makeHeap *needs $\Theta(n)$ time;* insert *and* decrease *need $\Theta(\log_d n)$ time. Operations* delMin*,* delete *and* increase *need $\Theta(d \log_d n)$ time.*

6 Towards Fibonacci heaps

The final target of the section are Fibonacci heaps, which will:

- speed up decrease to $\Theta(1)$! (amortized),
- improve insert and meld to $\Theta(1)$ (both worst-case and amortized), and
- otherwise keep the same amortized complexities, although the worst-case performance of some operations will degrade to $\Theta(n)$.[4]

The basic ideas come from binomial heaps, constructed from binomial trees.

[4]Deamortized Fibonacci-like heaps do exist, but they are significantly more complicated.

6.1 Binomial heaps

Definition. *Binomial tree* of rank i, denoted H_i, is a tree where every node corresponds to one key, and it satisfies the heap ordering on the keys (priorities). H_0 consists of a single node; H_{i+1} can be created by adding the root of one H_i as a new child under another H_i.

Note some properties of binomial trees (easily proven by induction):

- H_i contains 2^i vertices, has height i, and the root of H_i has i children.
- The i child subtrees of an H_i are isomorphic to $H_{i-1}, H_{i-2}, \ldots, H_1, H_0$.
- We have a simple way of joining two trees of the same rank i, which creates a tree of rank $i+1$ (in constant time). We choose the order of joining by the priorities of the two roots to preserve the heap ordering.

Definition. *Binomial heap* is a collection of binomial trees of *different* ranks.

Shape: The heap is usually implemented as an array of binomial trees that is indexed by ranks. Note the correspondence to binary numbers: any binomial tree of rank i can only represent a heap of size 2^i, and no two trees of the same rank can occur, so the shape of a binomial heap containing n elements is exactly given by the binary representation of the number n.

Potential: For amortized analysis we choose the potential as *the number of binomial trees* in the heap, which is between 0 and $\lceil \log n \rceil$.

Meld: It works as an analogy to binary addition. We start from the lowest ranks, and whenever we encounter two trees of the same rank, we join them. Together it obviously takes $\Theta(\log n)$ time where $n := n_1 + n_2$ is the size of the resulting heap. As the total potential can only change by $\mathcal{O}(\log n)$, we also have the amortized complexity $\mathcal{O}(\log n)$.

Insert: It can be done by meld with a tree of rank zero. Thus the worst-case complexity is $\Theta(\log n)$. We can easily prove that the amortized complexity of insert is $\Theta(1)$: we "flip some number of bits from one to zero and one bit from zero to one", so the decrease in potential pays for longer insertions. As a consequence we can asymptotically afford to build binomial heaps by insertions, as it only takes $\Theta(n)$ time in the worst case.

DelMin: We start by finding a tree that contains the smallest priority in the root, and we remove that tree. Children of its root form a collection of binomial trees, so we MELD them with the rest of the heap, as if they were another binomial heap. Together DELMIN takes $\Theta(\log n)$ both worst-case and amortized time.

Decrease and **increase:** They can be done like in regular heaps by UP and DOWN operations. The length of any path is bounded by $\Theta(\log n)$. When going up, we only need constant work on every level, which gives us $\Theta(\log n)$ worst-case time bound. When going down, we need to choose a child with the smallest priority on every level, so the bound is $\Theta(\log^2 n)$. The potentials are not touched, so the amortized complexities are the same.

Delete: As in many heaps, deleting arbitrary element can be done by decreasing the priority to "minus infinity", and then removing this minimum. Here this approach needs $\Theta(\log n)$ time, both worst-case and amortized.

Note. As is usual in heaps, we can maintain a pointer to an element with the smallest priority without increasing asymptotic complexities of the operations.

Corollary. *Binomial heaps need $\Theta(n)$ space. In the worst case we have all operations in $\Theta(\log n)$ except for:* INCREASE *in $\Theta(\log^2 n)$,* MIN *in $\Theta(1)$ and* MAKEHEAP *in $\Theta(n)$. Amortized bounds are asymptotically the same but for $\Theta(1)$* INSERT.

6.2 Lazy binomial heaps

Now we modify the operations to be lazy – we postpone as much work as possible and hope we can avoid most of it :-) As a result, work can accumulate and the worst-case complexities grow, but amortized complexities decrease for some operations.

We no longer require that the binomial trees have different ranks, so we have no bound on their number, and we store them in a list (often a doubly-linked circular list held by a minimal root).

Definition. Lazy binomial heap is a doubly linked list of binomial trees.

We leave the amortization potential as the number trees in the heap. **Meld** just concatenates two lists and **insert** adds one leaf to it, so both operations need $\Theta(1)$ both amortized and worst-case time. **decrease** and **increase** operations are the same as in strict heaps.

Finding a minimum

The problem is to find another minimum after we delete the old one – we have to walk through the whole list that can have length up to n. When we do the work, we also **reorganize** the list as follows.

Algorithm: We create a temporary array indexed by tree ranks (we have a $\log n$ bound on ranks). Then we push the whole list into it, merging as many trees as possible, and finally we convert the array back into a linked list.

Analysis: When we merge a tree under another vertex, we decrease the potential by one, which is enough to pay for all the work done with that tree, including that step in the linked list. At the end we have at most $\Theta(\log n)$ trees which were unpaid for, so the amortized cost of reorganization is $\Theta(\log n)$.

Algorithm details: If the implementation maintains a pointer to a minimum root, **delMin** first cuts it, then MELDs its children into the list, reorganizes it, and finally finds a new minimum. Otherwise DELMIN first reorganizes, then finds the minimum, cuts it, and MELDs the children. In either case we do $\Theta(\log n)$ work and a reorganization, which is $\Theta(n)$ in the worst case but $\Theta(\log n)$ amortized. General **delete** again works like DECREASE and DELMIN combined, so it needs the same asymptotic time.

Corollary. *The complexities of lazy binomial heaps only differ (from strict ones) in worst-case time complexity of some operations:* DELMIN *and* DELETE *need $\Theta(n)$ time,* INSERT *and* MELD *need $\Theta(1)$. (Amortized complexities remain the same.)*

Lazy binomial heaps do not seem to be an improvement, and they are rather an intermediate step towards the final structure. Our main long-term aim was to speed up DECREASE, which is needed e. g. for asymptotically fast Dijkstra's algorithm. We improve it to $\mathcal{O}(1)$ in the following structure.

6.3 Fibonacci heaps

Now we even drop the condition that the trees are binomial, so the shape is not explicitly defined. We guarantee that any vertex can only lose one child since the last moment it was in a root. We add an indicator into every vertex telling whether it has already lost a child or not, so we can maintain the property.

Bounding vertex degrees

Lemma 6.1. *Let v be a vertex in a Fibonacci heap and let u be its i-th oldest child. (Age counts from the moment a child is adopted by its current parent.) Then u has at least $i-2$ children.*

Proof. Let us look at the moment when u was joined under v. The $i-1$ older children of v were there, and both u and v had the same number of children. Since then u could have lost only one child, because it is not a root, so it still has at least $i-2$ children. □

Definition. Denote the i-th *Fibonacci number* by F_i: $F_0 := 0$, $F_1 := 1$, $F_{i+2} := F_{i+1} + F_i$.

Lemma. *Any subtree of a vertex with i children contains at least F_{i+2} vertices.*

Proof. By induction on i.

Case 1. $i = 0$. It is just one leaf, and $F_{0+2} = 1$.

Case 2. $i \geq 1$. Let us look at the children from the oldest one, and apply Lemma 6.1 on them. We get lower bounds on sizes of their subtrees: the total number of vertices including the root is at least $1 + \sum_{j=1}^{i} F_j$.

Now we show that $1 + \sum_{j=1}^{i} F_j = F_{i+2}$, which is easy by induction:

Case 1. $i = 0$. $1 = F_2$.

Case 2. $i \geq 1$. $1 + \sum_{j=1}^{i} F_j = 1 + \sum_{j=1}^{i-1} F_j + F_i = F_{i+1} + F_i = F_{i+2}$.

□

Fact 6.2. $F_i = \frac{1}{\sqrt{5}}\left[\varphi^i - (1-\varphi)^i\right] \geq \frac{1}{\sqrt{5}}\varphi^{i-1}$ *where* $\varphi := \frac{1+\sqrt{5}}{2} \doteq 1.62$ *is the golden ratio.*

Proof. A straightforward exercise. First, the equality follows by induction on i, using that $\varphi + 1 = \varphi^2$. For the inequality we use $0 < (1-\varphi) < 1 < \varphi$:

$$F_i = \frac{1}{\sqrt{5}}\left[\varphi^i - (1-\varphi)^i\right] = \frac{1}{\sqrt{5}}\left[\varphi^{i-1} + \underbrace{\varphi^{i-2}}_{\geq 1} - \underbrace{(1-\varphi)^i}_{\leq 1}\right] \geq \frac{1}{\sqrt{5}}\varphi^{i-1}.$$

□

Corollary 6.3. *The degree of any vertex in a Fibonacci heap is $\mathcal{O}(\log n)$.* The bound is approximately $1.44 \cdot \log n$, alike to depth bound for AVL trees.

Operation algorithms and analyses

The **potential** will be the the number of trees plus twice the number of vertices with their indicator set to true.

Again, **MELD** just concatenates two lists and **INSERT** adds one leaf to it, so both operations are still $\Theta(1)$ amortized and worst-case.

DELMIN also just removes a minimum root and reorganizes the list. The number of marked vertices can only decrease, so we use Corollary 6.3 with the reorganization accounting from lazy binomial heaps to get $\Theta(\log n)$ amortized bound (it is $\Theta(n)$ in the worst case).

All other problems are solved by **cutting** vertices and reinserting them into the list of trees. We mark the parent and increase the number of trees, so a single cut costs us $\Theta(1)$ real work plus potential increase by 3. However, if the parent was already marked before, we would be in an inconsistent state, so we also cut this parent the same way, which can cascade. We argue that resolving each such "doubly-marked" vertex can be paid from potential decrease: we remove a double-mark, add one mark, and add one tree, so the potential decreases by one. As a result the cutting is $\Theta(1)$ amortized (but $\Theta(n)$ in the worst case).

Note. We could not afford UP or DOWN operations, because the trees may degenerate to paths of length up to n.

If **DECREASE** breaks the heap condition with the parent, we cut the vertex and reinsert it with decreased priority, which is $\Theta(1)$ amortized. In **INCREASE** we need to cut not only the vertex but also all its children, so by using Corollary 6.3 we prove $\Theta(\log n)$ amortized time.

As in lazy binomial heaps, general **DELETE** is implemented via DECREASE and DELMIN, and we can build the heap by insertions in total $\Theta(n)$ time.

Corollary. *Fibonacci heaps need $\Theta(n)$ space. In the worst case we have all operations in $\Theta(n)$ except for $\Theta(1)$* MIN, INSERT *and* MELD. *Amortized bounds are asymptotically the same except for $\Theta(1)$* DECREASE *and $\Theta(\log n)$* INCREASE, DELMIN *and* DELETE.

That is, in comparison with lazy binomial heaps, we increased the worst-case complexities of several operations to $\Theta(n)$, but we improved the amortized bounds of DECREASE to $\Theta(1)$ and INCREASE to $\Theta(\log n)$. You can review the comparison (again) in Table 4.1 on page 25.

7 Application to Dijkstra's algorithm

Dijkstra's algorithm is a classical application for heaps. The input is an undirected combinatorial graph with n nodes and m edges. The edges are weighted by nonnegative numbers. The algorithm computes the lengths of all shortest paths from a given starting node. It uses a heap of the nodes (so it has size at most n), and its asymptotical complexity can be bounded by $n\times$using operations INSERT and DELETE, and $m\times$using DECREASE.

When we use the Fibonacci heap (which is the most efficient in asymptotic amortized sense), we get a time bound of $\mathcal{O}\left(m + n\log n\right)$.

Using d-regular heaps

The problem with Fibonacci heap is that it is quite complicated, and thus the multiplicative constants in time and space bounds are much higher than for many simpler heaps. We now show that for "most" graphs we can achieve the same asymptotic performance with d-regular heaps if we set $d := \min\{2, \Theta(m/n)\}$.

Remember that INSERT and DECREASE need time $\Theta(\log_d n)$ and DELETE needs $\Theta(d\log_d n)$ (all worst-case).

Case 1. Sparse graphs, i. e. $m = \mathcal{O}(n)$. In that case we have $d = \Theta(1)$, so every operation only needs $\Theta(\log n)$ time, and the total complexity is $\mathcal{O}(n\log n)$, which is the same as Fibonacci heaps for that case.

Case 2. Dense graphs where $\exists\epsilon > 0 :\ m \approx n^{1+\epsilon}$. In that case we have $d \approx n^{\epsilon}$, so each of the the cheaper operations need $\approx \log n \,/\, \log\left(n^{\epsilon}\right) = 1/\epsilon$, and each DELETE needs $\approx d/\epsilon \approx n^{\epsilon}/\epsilon$. That gives us compexity for Dijkstra bounded by $\lesssim m\cdot 1/\epsilon + n\cdot n^{\epsilon}/\epsilon \approx m/\epsilon$, which is the same as Fibonacci heaps for that case (for a bounded ϵ).

Part III
Memory hierarchies (caches)

Literature

The *Handbook of Data Structures and Applications* textbook devotes Chapter 34 to this topic. *Introduction to Algorithms* still does not contain anything on this topic (in its third edition).

Part IV
Hashing

Warning!

The content in this hashing part follows older version on the lecture.

---

8 The dictionary problem

- We are solving the *(unordered) dictionary problem*: we have some universe set $U = \{0, \dots, N-1\}$ and we want to store a set $S \subseteq U$, $|S| = n$, only using $\mathcal{O}(n)$ space (typically $N \gg n$). To do so, we use an array of length $m = \Theta(n)$ and hash functions of type $h : U \to \{0, \dots, m-1\}$.
- In these dictionaries we need to support operations INSERT, DELETE and MEMBER for manipulating the represented set S. Moreover, we aim to implement them in $\mathcal{O}(1)$ expected (amortized) time.
- Usual assumptions:
  - The hash function can be evaluated in $\mathcal{O}(1)$ time and it uniformly distributes the universe set (i. e. $\forall y_1, y_2 \; \left|h^{-1}(y_1)\right| - \left|h^{-1}(y_2)\right| \leq 1$).
  - Any element of the universe set is equally likely to be an argument of any of the operations (used in most expected-time analyses). Note that this assumption is very unrealistic; it can be avoided with universal and perfect hashing.
- Sometimes we will need to *rehash*: iterate over the contents of the whole structure, insert all the keys into a new hash table (possibly with different setting of m, h, etc.) and delete the old structure.
- The event that two different elements in S have the same hash is called a *collision*.

9 Separate chaining

- The simplest method: our array contains singly linked lists of elements to resolve hash collisions. On index i the array has a linked list that contains the set $\{x \in S \mid h(x) = i\}$.
- When performing an operation on the argument x, we first compute $i = h(x)$ and only work with the i-th linked list. The operations INSERT, DELETE and MEMBER on a linked list need time proportional to the length of the list.

Theorem 9.1. *The expected length of a chosen chain equals $\frac{n}{m} \equiv \alpha$ (*"load factor*").*

Proof. Following from the uniformity assumptions, the length of a chain is distributed according to binomial distribution $\mathrm{Bi}\left(n, \frac{1}{m}\right)$.[5] It only remains to prove that $\mathrm{E}\left[\mathrm{Bi}(n,p)\right] = np$ (easy exercise). ☐

Similarly, $\mathrm{Var}\left[\mathrm{Bi}\left(n,p\right)\right] = np(1-p) < np$, so we get variance even lower than α.

Theorem. *The expected length of the longest chain is $\Theta\left(\frac{\log n}{\log\log n}\right)$ (for $n \in \langle m/2, m\rangle$).*

Proof in Czech lecture, but very technical, about 1 A4 page.

- However, it is possible to have all the elements in one chain, so the wost-case complexity is $\mathcal{O}(n)$. Note that such inputs could be chosen by malicious attacker if (s)he just knows h (DOS attacks based on this exist in practice).
- We usually insert elements to the front of the list. It is also possible to insert at the end of the list or to maintain the list in sorted order. Maintaining sorted order allows us to reduce the time needed for unsuccessful searches in the list – on average we save half of the time (the asymptotic times are unchanged in any of these cases).

[5]The distribution would only be exactly binomial if the elements in S were chosen "with replacement". However, it's still an upper bound and it's very close in the typical case of $n \ll N$.

9.1 Periodic rebuilding

Theorem 9.1 suggests we should use a table of $m = \Theta(n)$. That way we get expected times of operations $\Theta\left(\frac{n}{m}\right) = \Theta(1)$ and the needed space is $\Theta(n+m) = \Theta(n)$, which is optimal (we need to store the n elements). To maintain the property, we can for example choose two constants and maintain $\frac{n}{m} \in \langle \alpha_1, \alpha_2 \rangle$. Whenever we break the constraint, we rehash the whole table with $m := \frac{2}{\alpha_1+\alpha_2} n$ (that changes h). The rehash takes $\Theta(n+m)$ time (in this sentence, m is the value *before* rehash), but we had to do at least $\frac{\alpha_2-\alpha_1}{2} m$ insertions or deletions since the last rehash, so the time can be *amortized* into additional $\Theta(1)$ time per modifying operation (amortization is discussed in Section 3.3).

9.2 Storing chains in the array

The array normally only contains pointers to the beginnings of the lists, but it is also possible to implement the linked lists inside of the array (we need $m \geq n$). We show two methods. In both we need to be able to find a free index in the array in constant time, which can be accomplished by maintaining another chain with all free positions. The operation complexities only differ by constant amount needed for an extra test or relocation.

Hashing with relocations: every index of the array contains fields key, next and prev, where next and prev act as pointers in a doubly linked list containing the chain. When inserting into a nonempty list, we find an empty index of the array, store the key in it and connect it with the list. When inserting into an empty list, it can happen that the index where the list should start is occupied by another item. In this situation we first have to *move* the item to another free position (that's why the list is doubly linked). To find the free space quickly, we may e. g. maintain also doubly linked list of the empty positions in the same way.

Hashing with two pointers: as the need to move elements is annoying, we show a modified scheme where every index of the array contains fields begin, key and next. Begin contains the index where the corresponding chain starts or a special value if that chain is empty. The other two fields form the actual singly linked lists: key contains the stored key and next contains the index of the next element in the chain or a special value if this is the last item. To find the free space quickly, we may e. g. maintain also *singly* linked list of the empty positions as a LIFO stack.

10 Coalesced hashing

- We want to store the chains in the array and want to save more space, although it can make the chains longer (again, we need $m \geq n$). The array items contains fields key and next, the chains are formed by the next links. There are two groups of methods:
  - *Standard coalesced hashing* (SCH) uses a standard array of length m.
  - *Coalesced hashing* (CH) uses a little longer array, where the auxiliary part is used for better handling of collisions (so the chains coalesce less). When finding a free position for a new *colliding* element, we first check the auxiliary part. Consequently, until the auxiliary part fills up, the chains don't coalesce (they are separated).
- Since the chains coalesce, we can't in general case perform efficient deletion. Instead, we mark the elements as deleted, so we can skip them while searching without breaking the chain (*fake deletion*). These ghosts can be replaced by other insertions into that chain, but in general we may have to rehash the table after performing many deletions (omitting the marked elements).
- There are various methods of CH and SCH that differ in the point of insertion into chains.
  - EICH and EISCH (*early insertion*) insert the elements after the first element in the chain.
  - LICH and LISCH (*late insertion*) insert the elements after the last element in the chain.
  - VICH (*varied insertion*) works like LICH in the auxiliary part and like EICH in the main part of the table. That is, the new element is put after the last element in the chain that is still in the auxiliary part of the table (or after the first element if the whole chain is in the main part).
- When maintaining a constant load factor α, all the coalesced methods have expected $\mathcal{O}(1)$ performance. Generally, the auxiliary memory in CH methods helps to speed up the operations in comparison to SCH methods. Early insertion is considered better than late insertion because usually it holds that the elements inserted later are more likely to be accessed. The VICH method has the best performance.

- All coalesced methods perform well even when filling the whole table, having at most around 2 expected tests per operation (averaging over all possible inputs). It is recommended to divide the memory between main and auxiliary part of the array in ratio around 85% : 15%. In the Czech classes, there is an analysis of *ISCH number of tests. It's technical and about 1-2 A4 pages long.

11 Open addressing

- We want to get rid of the additional fields in the array and so further improve the constant involved in memory consumption. These techniques are the most popular in practice, although they are more sensitive to a bad choice of the hash function for the input (in practice it usually doesn't hold that all inputs are equally probable, which can break our average-case analyses).

- The array only contains the keys and chains always start on their given indices, so the chains for different hashes can coalesce. It follows that in general we can only perform fake deletions like in coalesced hashing.

- All the algorithms iterate over the indices belonging into the chain until they find an empty space (the end of the chain). It is clear that when the table fills up, all the elements coalesce into one huge chain. That's why we have to keep lower load factor than in other hashing methods (we can afford that, because we use the memory more efficiently).

- We show two methods that differ in the way of finding the next index of the element in the chain.

11.1 Hashing with linear probing

- Hashing with linear probing chooses the next index in the array as the position of the following element (circularly, going to the first index after the last). That makes the algorithm very simple and it behaves well with memory caches.

- The load factor has to be kept low to prevent forming too long chains ($\alpha < 0.7$ is recommended). The method is also very sensitive to breaking the uniformity assumptions.

- When $\alpha \leq \alpha'$ for any fixed $\alpha' < 1$, the algorithms need $\mathcal{O}(1)$ expected time per operation.

11.2 Double hashing

- Double hashing uses two hash functions h_1, h_2 where $h_2(x) \neq 0$. For an element x, the position of i-th element of its chain is $(h_1(x) + ih_2(x)) \bmod m$. In order to cover the whole array with any of those sequences, we need $\gcd(h_2(x), m) = 1$ which is easily achieved by choosing prime m.
- The hashing with linear probing is a special case with $h_2(x) = 1$.
- Since for every element the chain jumps over different locations in the table, the coalescing isn't so severe. The recommended load factor is $\alpha < 0.9$.
- Restricting m to primes can be a problem when resizing the table. It holds that $\forall k > 0\ \exists\,\text{prime} \in \langle k, 2k\rangle$, no fast algorithm is known for finding any. (In practice we could precompute a suitable sequence of primes in advance.)

Czech classes contain analysis of performance conditioned by full randmoness of the sequence of probes (which can't be fulfilled; length ∼1/3 A4).

12 Universal hashing

- Universal hashing methods drop the unrealistic assumption that all elements of the universe are equally likely to be present in the input. It bounds the running time for *any* input averaging over outputs of a (pseudo)random generator.
- The basic method works just like hashing with separate chaining with the difference that the hash function is previously randomly chosen from a universal family of functions.
- A family of functions $H = \{h_i \mid i \in I\}$, $h_i : U \to \{0, \dots, m-1\}$ is called *c-universal* for $c \in \mathbb{R}$ iff $\forall x, y \in U,\ x \neq y :\ |\{i \in I \mid h_i(x) = h_i(y)\}| \leq c\frac{|I|}{m}$. Alternative formulation: $\forall x \neq y :\ \Pr_i [h(x) = h(y)] \leq \frac{c}{m}$.
- A popular c-universal family is $H = \{h_{a,b} \mid a, b \in \{0, \dots, p-1\}\}$ where $p \geq |U|$ is a prime, m is the size of the table and $h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$.

  Proof. Let $x, y \in U$, $x \neq y$ be fixed. We want to show that the number of pairs (a, b) such that $h_{a,b}(x) = h_{a,b}(y)$ is at most $\frac{cp^2}{m}$ for some c. For any such pair we denote $i := h_{a,b}(x) = h_{a,b}(y)$ and it must hold

$$\begin{aligned} ax + b &\equiv i + rm \pmod{p} \\ ay + b &\equiv i + sm \pmod{p} \end{aligned}$$

for some $r, s \in \{0, \dots, \lceil \frac{p}{m} \rceil - 1\}$. The same congruence in a matrix form:

$$\begin{pmatrix} x & 1 \\ y & 1 \end{pmatrix} \cdot \begin{pmatrix} a \\ b \end{pmatrix} \equiv \begin{pmatrix} i + rm \\ i + sm \end{pmatrix} \pmod{p}.$$

Since $x \neq y$, the first matrix is regular and we are solving a set of linear equations over a (Galois) field. From a theorem from linear algebra we get that for any combination of the i, r, s parameters there is exatly one satisfying pair $(a, b) \in \{0, \dots, p-1\}^2 \supseteq U^2$. Therefore, the number of such pairs is bounded by the number of i, r, s combinations, which is $m \lceil \frac{p}{m} \rceil^2 = (1+\epsilon)\frac{p^2}{m}$ for some $\epsilon > 0$ (created by rounding; normally $p \gg m$, so $\epsilon \ll 1$). □

12.1 Properties of c-universal systems

- For $i \in I$ and $x, y \in U$ we define $\delta_i(x, y) := \begin{cases} 1 & \text{if } x \neq y \text{ and } h_i(x) = h_i(y) \\ 0 & \text{otherwise} \end{cases}$.

  Moreover, for $S \subseteq U$ we define $\delta_i(x, S) := \sum_{z \in S} \delta_i(x, z)$. This is the number of elements in S colliding with x.

- The expected number of elements in S colliding with x equals

$$\begin{aligned} \frac{1}{|I|} \sum_{i \in I} \delta_i(x, S) &= \frac{1}{|I|} \sum_{i \in I} \sum_{z \in S} \delta_i(x, z) = \frac{1}{|I|} \sum_{z \in S} \sum_{i \in I} \delta_i(x, z) \leq \\ & \text{now we use the definition of } c\text{-universality} \\ &\leq \frac{1}{|I|} \sum_{z \in S \setminus \{x\}} \frac{c|I|}{m} = \frac{c\,|S \setminus \{x\}|}{m} \leq \alpha c. \end{aligned}$$

  Note that x and S were arbitrary. Consequently, we have the expected length of a chain bounded by αc, so when maintaining a constant load factor, the expected complexity of operations is $\mathcal{O}(1)$ for *any* input.

- There is a lower bound on the size of a c-universal family of functions: $|I| \geq \frac{m}{c}(\lceil \log_m N \rceil - 1)$ where $N := |U|$. As a consequence, the number of random bits for choosing a function is at least $\log |I| \gtrapprox \log m - \log c + \log\log N - \log\log m$.

- There exists a c-universal system that asymptotically achieves the lower bound on the number of needed random bits.

- For an arbitrary c-universal system it holds $c \geq 1 - \frac{m}{N} \doteq 1$ (usually $m \ll N$).

13 Perfect hashing

In perfect hashing we try to find for a set $S \subseteq U$ a function h that satisfies the following requirements:

1. h has no collisions: $\nexists x, y \in S : x \neq y,\ h(x) = h(y)$.
2. m is sufficiently small. We will need $m \in \mathcal{O}(n)$ to fit into linear space.
3. h is easily computable, ideally we can find $h(x)$ in small-constant time.
4. The description of h doesn't take much space (if h was defined by a table of all values, it would not help us, reducing hashing to tables again).

13.1 Perfect families

- A family H of hash functions is called (N, m, n)*-perfect* iff for every $S \subseteq U$ of size n there *exists* $h \in H$ such that it uses a table of size m and it is perfect for S (no collisions on S).
- There are lower bounds on the size of a (N, m, n)-perfect system.

$$|H| \geq \max\left\{ \frac{\binom{N}{n}}{\binom{m}{k}\left(\frac{N}{m}\right)^n}, \frac{\log N}{\log m} \right\}$$

- It is possible to prove the existence of a (N, m, n)-perfect family of functions by examining all possible functions, but the proof doesn't guarantee the requirements (3) and (4) and it only shows the existence, not a way of finding the function reasonably quickly.
- Typical usage: show that the number of suitable functions is always at least a fixed fraction of H. By trying randomly we then find one such in expected constant number of trials (moreover, the probability of reaching attempt k decreases exponentially in k, e. g. by applying Chernoff bounds).

13.2 Construction of a perfect function

We show an algorithm that constructs a two-level static hash table such that it satisfies all the mentioned requirements via any c-universal family H of hash functions.

- The first level will be a hash table of size $m := 2cn$ with hash function h, dividing the n elements into chains of length n_i. We require that the number of ordered collision pairs is at most n, i. e. $R := \sum_i n_i (n_i - 1) \leq n$.
- Every chain will be stored in a *perfect* table of size $m_i := 2cn_i^2$ by a hash function h_i. We also need to store the parameters of these functions h and h_i, so we can evaluate them when working with the table.
- The idea of a proof:
  - We knew that for *any* table with c-universal family with parameters n, m, the expected number of collisions with a fixed $x \in U$ is at most $\alpha c = \frac{n}{m}c$, so the expected number of ordered collision pairs at most $\sum_x \frac{n}{m}c = \frac{n^2}{m}c$.
  - For h we chose $m = 2cn$, so $\mathrm{E}_h [R] \leq \frac{n^2}{m}c = \frac{n^2}{2cn}c = \frac{n}{2}$, so by Markov inequality we get that the probability of finding a suitable function in H (with $R \leq n$) is at least $\frac{1}{2}$.
  - For h_i we chose $m_i = 2cn_i^2$, so $\mathrm{E}_h [R_i] \leq \frac{n_i^2}{m_i}c = \frac{n_i^2}{2cn_i^2}c = \frac{1}{2}$ and by Markov inequality we get that the probability of finding a perfect function $h_i \in H$ is at least $\frac{1}{2}$ (since $R_i \in \mathbb{N}$, $R_i < 1$ implies $R_i = 0$).
  - The tables obviously need space

    $$\mathcal{O}\left(m + \sum_i m_i\right) = \mathcal{O}\left(n + \sum_i n_i^2\right) = \mathcal{O}\left(n + R + n\right) = \mathcal{O}(n)$$

    where the middle equality is due to:

    $$R = \sum_i n_i(n_i - 1) = \sum_i n_i^2 - \sum_i n_i = \sum_i n_i^2 - n.$$

- Note: this static construction can be modified to a dynamically maintained perfect hash table (with bigger constant factors). That gives us $\mathcal{O}(1)$ search in the wost case and $\mathcal{O}(1)$ expected amortized updates in $\mathcal{O}(n)$ space (worst-case update time is unbounded again).

14 Exendible hashing

Hashing in external memories has some specifics. We illustrate them on a straightforward method by *Fagin*.

14.1 The external memory model

- We have external memory which we call *disk* and (internal) *memory*.
- Disk is divided into blocks and it can't be accessed directly. We can only read a block from disc to memory or write a block to disk. When storing keys, we denote b the number of keys that fit into one block and suppose $b > 1$.
- Usually, we can only keep $\mathcal{O}(1)$ blocks in memory and $\mathcal{O}(1)$ additional data.
- The access to disk is considered so slow compared to memory that we only count the number of disk operations for the time complexity.

14.2 The hashing method

- We use a hash function h that hashes every key into a word of zeros and ones.
- The structure consists of a *directory* that holds pointers to disk blocks containing the actual keys. The directory is an array of length 2^l and on index i it contains the identifier of the block that stores all the keys whose first l bits of hash are equal to i (and there may be some other keys).
- Every block P has a number $l_P \leq l$ which is the number of starting bits that are guaranteed to be common for all the keys stored within P. This common prefix of length l_P is called *critical word*. The indices in the directory that point to the block P are exactly those that start with the critical word of P. As a consequence we have 2^{l-l_P} pointers to P.

14.3 The operation algorithms

- When analyzing the operations we assume that the directory is held in the main memory. Otherwise we would need some more block reading and writing, especially when resizing the directory.
- If we wanted to store some additional data associated with the keys, the operations would also need some additional block reading or writing proportional to the size of stored data.

Member(x) We first compute the first l bits of the hash value $h(x)$ and on this index of the directory we find the indentifier of a block (pointer to a block). If the pointer is empty, we exit immediately, otherwise we read the block into memory and search its contents for x. Thus we at most need to read one block.

Insert(x) We first find the block P in which x belongs (reading P in the process). We insert x into P (unless it was there already), but if P was full, then:

- *We split P in two.* The hashes of the keys in P share the first l_P bits. To free one more position in P we have to find l'_P: a higher value for l_P such that it splits away at least one element. The value $l_P + 1$ usually suffices, but it may have to be higher. Every increment of l_P splits in half the part of the universe set covered by the block. If both of the halves contain at least one key, we fit in. Otherwise we have to split again. This process divides the keys from P into two blocks (and possibly some "empty blocks" which we don't allocate). We allocate that new block on disc and write the new contents into the two blocks (we reuse the old block of P).
- *We make sure that the directory is large enough* ($l \geq l'_P$). The resize just creates the new directory array of length $2^{l'_P}$ and every block pointer from the old array is replaced by its copies on consecutive $2^{l'_P - l}$ indices. The usual case is increasing l by 1 so every pointer is doubled. Since usually we suppose that the directory fits into the main memory, the resize needs no disc operations.
- *We update the directory* (regardless of its resizing). We redirect indices that pointed to P to point to the correct of the two blocks (or empty pointers). There was at most 2^{l-l_P} pointers to P and we find them on the indices corresponding to P's *critical word*.

To sum up, the worst case for needed disc block operations is when P has to be split and it consists of one reading, one writing and one allocation with writing.

Delete(x) We find the block P in which x belongs (reading P in the process). If x isn't in P, we terminate the operation. Otherwise we remove and then take care of merging possibilites for P:

- We take the *critical word* of P (the first common l_P bits), flip the least significant bit and look at the resulting index in our directory. If it points to such a block Q that $l_P = l_Q$ and their keys can fit into a single block (we read Q in the process), we merge P and Q. That consists of deallocating Q, writing the new contents into P and replacing pointers to Q by pointers to P (l_P is decreased by one). It also may be possible to merge the resulting block with adjacent "empty blocks" represented in the directory by nil pointers (the inverse case to multiple splitting during INSERT).
- As we decreased l_P, we can reduce the size of the dictionary if now $l > \max\{l_R \mid \text{block } R\}$. To be able to find this out, we can store the l_R values within the directory and always scan them (if it can be kept in the main memory). A more efficient way is to maintain the total counts of used l_R values, which is just an array of length l containing small integers.
- Resizing the dictionary is again straightforward. Let $l' = \max\{l_R \mid \text{block } R\}$, then we are guaranteed that the consecutive segments of length $2^{l-l'}$ in the dictionary contain the same block pointers, so we can replace the segments by single items.

To sum up, the worst case for needed disc block operations is when P is merged and it consists of two readings, one writing and one deallocation.

14.4 Expected properties

- For analysis we suppose the usual uniformity assumptions for simple hashing.
- The expected number of used blocks equals $\frac{n}{b \ln 2}$. That is, the blocks are expected to be around 69% full.
- The expected size of the directory equals $\frac{b}{\ln 2} n^{1+1/b}$. This can be a problem for low b values where the size of the directory grows almost quadratically.

Part V
Omitted from the lecture

Some topics are no longer taught in the course, but they may be of interest.

15 AVL trees

AVL trees are binary search trees (see pg. 12) that maintain a simple balancing condition: in every vertex the heights of its child subtrees differ by at most one. We need to add a field $\mathsf{bal}(v) := \mathsf{height}\,(\mathsf{r}(v)) - \mathsf{height}\,(\mathsf{l}(v))$.

Now we show how this condition bounds the height of the tree.

Lemma 15.1. *Let $f(h)$ be the minimal number of elements representable by an AVL of height h. Then $f(h) = F_{h+2} - 1$ where F_i is the i-th Fibonacci number ($F_0 = 0$, $F_1 = 1$, $F_{i+1} = F_i + F_{i-1}$).*

Proof. By induction on h.

Case 1. $h = 0$. A single leaf represents the empty set, $f(0) = 0 = F_2 - 1$.

Case 2. $h = 1$. Root with two leaves as children represents a set of size one, $f(1) = 1 = F_3 - 1$.

Case 3. $h \geq 2$. Root with two subtrees of lower height. The worst case is when we have children of height $h-1$ and $h-2$, and both represent the minimum number of elements possible, thus

$$f(h) = 1 + f(h-1) + f(h-2) = 1 + F_{h+1} - 1 + F_h - 1 = F_{h+2} - 1.$$

□

Fact (see 6.2 on page 31). $F_i = \frac{1}{\sqrt{5}}\left[\varphi^i - (1-\varphi)^i\right] \geq \frac{1}{\sqrt{5}}\varphi^{i-1}$ *where* $\varphi = \frac{1+\sqrt{5}}{2}$.

Corollary 15.2. *The height of AVL tree containing n elements is at most* $\log_\varphi n + \mathcal{O}(1)$.

Proof. We know $n \geq F_{h+2} - 1 \geq \frac{1}{\sqrt{5}}\varphi^{h+1} - 1$ so $\log_\varphi\left(\sqrt{5}(n+1)\right) - 1 \geq h$, thus

$$h \leq \log_\varphi(n+1) + \log_\varphi \frac{\sqrt{5}}{\varphi} \doteq 1.44\log(n+1) + 0.67.$$

□

Operations

Simple operations like MEMBER, MIN and MAX are the same as in general BSTs. In INSERT and DELETE we just update the values of bal and then perform some rebalancing at the end of the operation to restore that $|\mathsf{bal}\,(u)| \leq 1$, as follows.

The invariant can only break in the way that $|\mathsf{bal}\,(u)| = 2$ in some vertices on the affected path. We shall start with the lowest broken vertex v. Here we only discuss the case of $\mathsf{bal}\,(u) = 2$, as the other one is symmetrical. Let $v := \mathsf{r}(u)$, there are several cases (there's a figure for each):

Case 1. $\mathsf{bal}\,(v) = 1$. We rotate the $(u \to v)$ edge. That restores $\mathsf{bal}(u) = \mathsf{bal}(v) = 0$ and decreases the height in the root of rotation by one.

Case 2. $\mathsf{bal}\,(v) = 0$. We also rotate the $(u \to v)$ edge, but this time we get $\mathsf{bal}(u) = 1$ and $\mathsf{bal}(v) = -1$. The height in the root of rotation remains the same.

Case 3. $\mathsf{bal}\,(v) = -1$. We double-rotate the $w := \mathsf{l}(u)$ vertex, that is, we first rotate $(v \to w)$ and then $(u \to w)$. That gives $\mathsf{bal}(w) = 0$; for v and u the new balance depends on the old balance of w, but it is OK in any case. The height in the root of double rotation is decreased by one.

After fixing the invariant locally we continue up the path fixing the bal values. It can be shown that:

- After INSERT there can only be cases 1 or 3, so after hitting one of these cases, the height in the root of (double) rotation gets down back to the value before insertion and the operation can end at that point.
- After DELETE it may be needed to rotate the whole path.

The RANK and ORD operations can be added the same way as in general BSTs. It is obvious now that in any case we do at most some constant work on every level.

Corollary 15.3. *The complexities of all operations on AVL trees are $\Theta(\log n)$.*

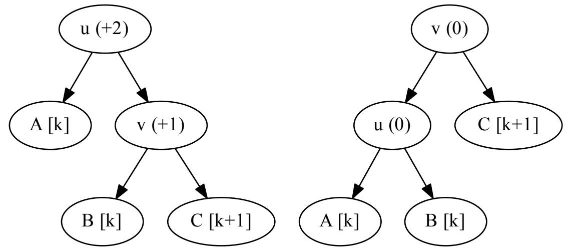

Figure 15.1: AVL: rotation for Case 1. As for all AVL figures, balance information is in parentheses, and heights are in brackets (parametrized by k). Note that nodes can have larger subtrees that are not shown.

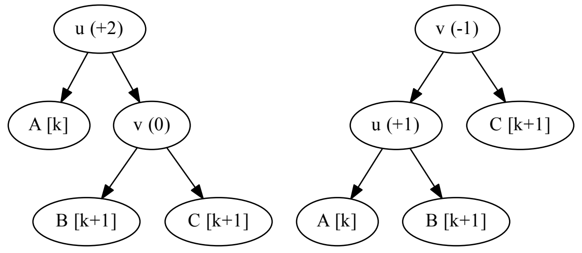

Figure 15.2: AVL: rotation for Case 2.

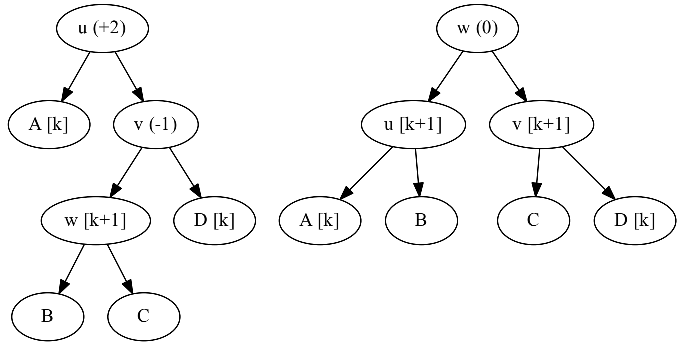

Figure 15.3: AVL: double rotation for Case 3.

16 Sorting and searching arrays

Conventions: input and output in an array of length n, we want the ascending order.

16.1 Comparison-based sorting

In comparison-based sorting we only allow pairwise comparisons of elements. We will see that the time for worst-case input is asymptotically logarithmic.

HeapSort works via constructing a d-regular max-heap in the input array with $d \in \Theta(1)$. Then it repeatedly extracts the maximum and puts it at the end of the array into the place which is no longer used by the heap, until we sort it all.

Properties: $\Theta(n \log n)$ time and comparisons, but high multiplicative constants for both. It's robust (almost always the same time) and in-place (only constant additional memory is needed).

MergeSort starts with a queue for sorted sequences that is initialized by the singleton elements (for example). Then repeatedly the first two sequences are merged and the result is put on the end of the queue, until we only have one sorted sequence.

Merging two sorted sequences: iteratively compare the first elements and extract a smaller one into the output. To achieve stability (never change order of elements that compare equal), we select from the first sequence whenever there is a tie. Obviously this takes $n_1 + n_2 - 1$ comparisons and time.

Running time: every time the queue is rotated, perhaps leaving one sequence into the next round, the number of sequences is halved (rounded up), so every element takes part in at most $\lceil \log n \rceil$ merges. Thus the total number of comparisons is at most $n \lceil \log n \rceil$, which is exactly the best possible in the worst case (up to rounding).

Properties: faster than heapsort in practice, but not in-place[6]; easy to paralelize, stable, low number of comparisons, good memory locality. It is very useful on memories with substantially faster sequential access, such as disks or tapes, usually modified to do multi-way merges. The start can be modified to work faster almost-sorted sequences, but then we need a more complicated way of choosing the pairs to merge (Huffman tree).

A-sort: described on page 11.

[6]However, there exists a very complicated version that is in-place.

QuickSort

Quicksort starts by choosing any element as the pivot (somehow) and exchanging it with the first element. Then we partition the rest of the array according to comparison with the pivot as follows. We have two pointers – one starts on the leftmost position and finds the first element larger than the pivot, the latter starts on the right and finds an element smaller than the pivot. Then those elements are exchanged and all continues until the pointers cross. To finish the partition, we exchange the pivot with the last element of the created left segment. Finally we use recursion on both segments.

Properties: fast and popular in practice (uses relatively sequential access), in-place, good memory locality, but sensitive to pivot choice or malicious inputs.

Pivot choice: if we are unlucky and e .g. always choose the smallest element, then the running time is obviously $\Theta\left(n^2\right)$. Several common ways to deal with it:

- Use some fast heuristic choice, e. g. the median of the first, middle and last element. This is still $\Theta\left(n^2\right)$ in the worst case, but such inputs are less natural (the original formulation degenerates even on almost-sorted sequences), and even the average case might get improved a little.
- Choose randomly. As proven below, this gives $\Theta(n \log n)$ expected time, regardless of the input. However, using so much randomness is usualy too expensive in practice.
- Choose a median or other good-enough element (at least a constant fraction from both ends in the order). This can be done in $\Theta(n)$ time in the worst case, so we can guarantee $\Theta(n \log n)$ time. However, the constants get higher than for mergeSort or heapSort, so it isn't practical.
- IntroSort or similar solution: do some simple choice (e. g. a middle element), but switch to a worst-case method (typically heapSort) whenever the depth of recursion reaches some logarithmic threshold. Very popular in practice.

Expected case: denote the elements $x_1 < x_2 < \cdots < x_n$, then observe that x_i and x_j, $i < j$, are compared iff either is chosen as the pivot before they are split by choosing some pivot in between. If the pivot choice is uniform, then clearly $\Pr[x_i \text{and } x_j \text{get compared}] = 2/(j-i+1)$. The expected number of comparisons:

$$\sum_{1\leq i<j\leq n} \frac{2}{j-i+1} \overset{k:=j-i+1}{=} \sum_{i=1}^{n-1}\sum_{k=2}^{n-i+1} \frac{2}{k} < 2n\sum_{k=2}^{n}\frac{1}{k} < 2n\int_1^n \frac{1}{k} = 2n\ln n \doteq 2.88{\cdot}n\log n$$

Lower bounds via decision trees

Decision trees: for simplicity we suppose the algorithm is deterministic, and we imagine a tree describing comparisons on all possible inputs. Thus each vertex corresponds to a comparison, and the tree is binary. Each leaf corresponds to one result of execution (assuming we pruned unreachable vertices), so the depth of the tree will correspond to the worst case number of comparisons.

Sorting needs to differentiate among all permutations of the input sequence, which is $n!$ possibilities (for a sequence without repeats). That means the depth is bounded by $\log n! \geq \log n^{n/2} = \frac{n}{2}\log n$ (first inequality by pairing as $k\cdot(n-k) \geq n$; almost exact analysis: $\log n! \doteq n \log \frac{n}{\mathrm{e}}$).

Similarly for a simple search in a **dictionary** of size n, we need to differentiate among $n+1$ possibilities, which gives us a lower bound of $\log n$ comparisons.

16.2 Non-comparison sorting

CountingSort / BucketSort works on integers in some range of length m. We allocate an array of m counters initialized to zeros, and on the first pass we count the numbers of keys. If we don't care e. g. about data associated with the keys, we could produce the output by one pass over the counters. Otherwise we accumulate the numbers from the left, so the new counter values will contain the sums of the old counter values to the left (excluding itself). Note that now the counter for each key marks the first position in the output array that should be occupied by some equal key. Now we pass over the input and put every element at the position denoted by its counter, while incrementing that counter. Obviously, the algorithm is correct and stable; it needs additional space is $\Theta(m)$ and total time is $\Theta(n+m)$.

Note: the bucketSort implementation is often explained in a way that instead of counters we have buckets, i. e. linear linked lists containing the elements. However, the implementation above behaves the same asymptotically and it has much better multiplicative constants in practice.

RadixSort expands the linear behaviour from linear to polynomial ranges. We imagine each key is split into digits of $\log n + \Theta(1)$ bits, which means each digit has range $\Theta(n)$. We start with the input sequence, and in phase i we bucket-sort the output of the previous phase by the i-th least significant digit.

Note that by induction, after i phases we get a sequence that is sorted lexicographically according to the last i digits (using stability of bucketSort). Each phase takes $\Theta(n)$ time and memory and the number of phases is $\Theta(\log m/\log n)$. If the total range is polynomial, i. e. $m = n^{\Theta(1)}$, we get total time $\Theta(n)$. RadixSort is very fast even in practice, if the keys are relatively short.

WordSort expands the linear behaviour to the case when lengths of keys differ much. We denote the total number of digits in the input as N and the alphabet for digits as Σ. First we construct a list of digits occuring on every position in keys: we radix-sort all pairs (position,digit) from the input, and we remove repeated occurences. That will need time and space $\Theta(N + |\Sigma|)$.

Then we bucket-sort all keys by their length (number of digits), which needs $\Theta(N)$ time and space. We continue similarly to radixSort, sorting by digits of keys. In early phases we don't include keys that aren't long enough (we can imagine the keys aligned on the *most* significant bit). Also in each phase we use the sorted list of occuring digits, so we skip the other digits, and the time needed in a phase is only linear in the number of keys included in that phase (without dependency on $|\Sigma|$). As each key takes part in as many phases as its number of digits, the total time and space is $\Theta(N + |\Sigma|)$, including preprocessing.

HybridSort shows a good example of using distribution of input data. We suppose the keys are distributed uniformly over some interval. We split that interval uniformly into $m := \alpha n$ buckets for an arbitrary $\alpha \in \Theta(1)$, and we distribute the input into an array with one linked list per bucket. Then we sort each bucket and send its elements in order to the output.

Worst-case analysis of $\Theta(n \log n)$: if bucket i gets B_i elements, then the time of the second phase is asymptotically bounded by $\sum_i B_i \log B_i \le \sum_i B_i \log n = n \log n$.

Expected-time analysis of $\Theta(n)$. B_i are binomially distributed random variables: $\Pr[B_i = k] = \binom{n}{k}\left(\frac{1}{m}\right)^k\left(1-\frac{1}{m}\right)^{n-k}$. If we leave out buckets of size less than two, then the expected time for bucket i is asymptotically

$$\sum_{k=2}^{n} \Pr[B_i = k]\, k \log k \le \sum_{k=2}^{n} \Pr[B_i = k] \cdot k(k-1) =$$

$$\sum_{k=2}^{n} \frac{k(k-1)\cdot n!}{k!(n-k)!}\left(\frac{1}{m}\right)^k\left(1-\frac{1}{m}\right)^{n-k} = \sum_{k=2}^{n} \frac{n(n-1)(n-2)!}{(k-2)!(n-k)!}\left(\frac{1}{m}\right)^k\left(1-\frac{1}{m}\right)^{n-k}$$

$$\overset{k':=k-2}{=} \frac{n(n-1)}{m^2}\sum_{k'=0}^{n-2}\binom{n-2}{k'}\left(\frac{1}{m}\right)^{k'}\left(1-\frac{1}{m}\right)^{n-2-k'} < \frac{n^2}{m^2}\left(\frac{1}{m}+1-\frac{1}{m}\right)^{n-2}.$$

Thus the total expected time is $\mathcal{O}\left(n + m + m \cdot n^2/m^2\right) = \mathcal{O}\left(n + \alpha n + \frac{n^2}{\alpha n}\right) = \mathcal{O}(n)$.

Note that this scheme can be directly generalized to a dynamic data structure that works with expected constant time per operation (for uniformly distributed data). For the bucket structures we would better choose e. g. binary search trees, so most operations get logarithmic in the worst case as well.

16.3 Selecting the k-th element

Sometimes we don't need sorting but just finding the k-th smallest element (e. g. median). That can be done in linear time even in comparison-based setting. For simplicity we assume that no elements are equal, so the choice is unambiguous.

QuickSelect is a trivial modification of quickSort – instead of recursing into both subsequences, it is sufficient to explore just one of them. More precisely: after splitting according to a pivot with l elemnts to the left, we know that the pivot will have the current position $l+1$ in the sorted order. Thus if $k = l+1$, we have the answer; if $k < l+1$, we search for k-th on the left subsequence. Finally, if $k > l+1$, we search for $(k-l-1)$-th in the right subsequence.

Analyses: in the worst case we split away just one element per recursion step, so it can take $\Theta(n^2)$ time. However, with random input/pivot, we are expected to cut away at least $1/4$ of current elements.[7] Now we use induction by n: if we have $T(n') \le cn' + d$ for some constants c, d and all $n' < n$; then if we recurse, for some constants s and t, we have expected time $T(n) \le T\left(\frac{3}{4}n\right) + sn + t \le n\left(\frac{3}{4}c + s\right) + \frac{3}{4}d + t$, so it suffices to choose $c \ge 4s$ and $d \ge 4t$ for the base case of some small constant n, and we get $T(n) = \Theta(n)$.

Median of medians can do the selection in linear worst-case time by choosing a good-enough pivot in linear time (splitting takes linear time anyway). First it conceptually divides the input into 5-tuples and finds a median in each, e. g. by sorting.[8] Then it finds a median p of these medians by recursively calling the selection algorithm. By transitivity over the medians smaller than p (or equal), we have at least $\frac{3}{5}\frac{n}{2} = 0.3\,n$ elements with value at most p (up to small additive constant due to rounding). The situation is symmetric, so by choosing p as the pivot we reduce the size of the task to $0.7\,n$ in the worst case.

Analysis by induction on n, denoting the time by $T(n)$: if we have $T(n') \le cn' + d$ for some constants c, d and all $n' < n$, then by the recursion step, for some constants s and t, we have $T(n) \le sn + T(n/5) + T(0.7\,n) + t = sn + 0.9\,cn + 0.9\,d + t$. That is, if we choose $c \ge 10s$ and $d \ge 10t$ to get $T(n) = \Theta(n)$.

Practice: the constants get high by using this guaranteed algorithm, so people tend to use a simpler rule for pivot choice (for both quickSelect and quickSort).

[7]The following analysis can also be solved by directly applying Master Theorem instead. An extended version also simply solves the median-of-medians analysis.

[8]The last group can be smaller. We can also use 7-tuples instead.

16.4 Searching sorted arrays

It's a static version of the ordered dictionary problem. We assume the elements are ordered and stored in an array (consecutively; we can do direct addressing). The only thing a comparison-based search can do is to successively *probe* some positions in the array (i. e. compare them with the key that we search for).

The well-known binary search needs $\log n + \Theta(1)$ probes. Exercise: in the worst case any comparison-based data structure can't answer MEMBER(x) faster (hint: decision tree lower bound). However, if we assume some properties about the data, we can expect to be even exponentially faster.

Notation: $A[i]$ for the value stored on the index i, and $A[l \dots h]$ for the currently searched segment. We assume the keys are numbers that are uniformly distributed over some interval, and we're searching for key x.

Interpolation search

- The next index to probe is always $l + \left\lceil \frac{x-A[l]}{A[h]-A[l]}(h-l) \right\rceil$ (naturally!).
- The expected number of probes needed is $\log\log n + \Theta(1)$, which is optimal (for searching sorted arrays via probes; cf. dynamizing bucket sort).
- However, in the worst case the number of probes is $\Theta(n)$.

The expected complexity is quite difficult to prove, so we show a modified algorithm where it's much easier and we only have to pay a multiplicative constant.

Generalized quadratic search

The probes are generated in blocks, where each looks like this:

- First we do an interpolation step. This determines the length of the following *unary* steps as $\left\lfloor \sqrt{h-l} \right\rfloor$.
- We alternate unary steps with binary steps until we jump over the value of x with a unary step. Unary step just additively moves l or h.

Such blocks get repeated until we find x. First note that together we do at most $3\log n$ probes (in the worst case!), because at least every third probe is binary.

The length of the segment always decreases to at most square root, so there can be at most $\log\log n$ blocks.

Theorem (Chebyshev inequality). $\forall k > 0 \quad \Pr\left[|X - \mathrm{E}[X]| \geq k\right] \leq \dfrac{\mathrm{Var}[X]}{k^2}$.

Expected-case analysis Denote X as the index of x within the segment $A[l \dots h]$, and let $p := \frac{x-A[l]}{A[h]-A[l]}$. Notice that p is the probability that an element in $A[l \dots h]$ is before x, so X is binomially distributed within the segment, i. e. $X \sim \mathrm{Bi}(h-l, p)$, and $\mu := \mathrm{E}[X] = p(h-l)$ (note: that's our probe position), $\sigma^2 := \mathrm{Var}[X] = p(1-p)(h-l) \leq \frac{1}{4}(h-l)$. Now by using Cheybshev inequality with $k := i\sqrt{h-l}$ we have that

$$\Pr\left[|X - p(h-l)| \geq i\sqrt{h-l}\right] \leq \frac{\frac{1}{4}(h-l)}{i^2\sqrt{h-l}^2} = \frac{1}{4i^2}.$$

Consequently, the expected number of unary steps needed to skip x is at most $1 + \sum_{i=1}^{\infty} \Pr[\text{more than } i \text{unary steps}] = 1 + \sum_{i=1}^{\infty} \frac{1}{4i^2} = 1 + \frac{\pi^2}{6}$. The number of binary steps is lower, so the expected total number of all steps within a block is bounded by $\underbrace{1}_{\text{i.}} + \underbrace{\frac{\pi^2}{6}}_{\text{binary}} + \underbrace{1 + \frac{\pi^2}{6}}_{\text{unary}} = 2 + \frac{\pi^2}{3} < 5.3$.

Corollary. The total expected number of probes of generalized quadratic search is bounded by $5.3 \log\log n$.

17 Leftist heaps

The main advantage of leftist heaps against regular heaps is that they allow MELD in $\Theta(\log n)$ time. They are much simpler than binomial or Fibonacci heaps.

Definition

- Binary tree with heap property, stored elements are 1-1 mapped to vertices.
- Let $\mathsf{sp}(v)$ be the length of the shortest path from v within its subtree to a vertex with at least one child missing.
- Leftist property: if a vertex has only one child, then it's left; if v has two children, then $\mathsf{sp}(\mathsf{l}(v)) \geq \mathsf{sp}(\mathsf{r}(v))$.
- Each vertex also maintains a pointer to its parent.

Note that due to the leftist property the rightmost path from any v is shortest among the paths down the subtree and so it has lenght $\mathsf{sp}(v)$. Therefore the subtree of v is full up to the level of $\mathsf{sp}(v)$: $n \geq \sum_{d=0}^{\mathsf{sp}(v)} 2^d = 2^{\mathsf{sp}(v)+1} - 1$, so $\mathsf{sp}(v) \leq \log(n+1) - 1 \leq \log n$ (for $n > 0$).

Operations

Note that paths in leftist heaps could be very long, so we can't afford using UP and DOWN as in regular heaps.

MELD(H_1, H_2): The basic operation. Easy if either heap is empty, otherwise WLOG $\mathsf{root}(H_1) \leq \mathsf{root}(H_2)$. We replace $\mathsf{r}(\mathsf{root}(H_1))$ by its MELD with H_2 (recursively). Then we recompute the value of sp from values in children, and we switch them if the leftist property got broken. Note that in every recursion step we shorten sp of one of the heaps being melded, so the total running time is $\Theta(\log n)$.

INSERT(H, x): We just MELD with a one-element heap created from x.

DELMIN(H): We remove the root and MELD its children.

MAKEHEAP(A): We create a FIFO queue of heaps, initialized by singletons from the input. Then we repeatedly pop the first two heaps, MELD them and push the result on the end of the queue. When only one heap remains, we have the result.

We show that the complexity is linear. We look at the queue's rounds of rotation: in round i we MELD at most $\lceil n/2^i \rceil$ pairs of heaps with sizes at most 2^{i-1}. That means $\mathsf{sp} \leq \log\left(2^{i-1}+1\right) - 1 \leq i-1$ (as $i \geq 1$), so each MELD in round i needs at most $2(i-1)+1 < 2i$ steps. Total work: $\sum_{i=1}^{\lceil \log n \rceil} 2i \lceil n/2^i \rceil \leq \log n + 2n \sum_{i=1}^{\lceil \log n \rceil} i/2^i \leq \mathcal{O}(n)$, as this is a finite sum alike to MAKEHEAP of regular heaps.

FIX(H, p): An auxiliary operation to fix the values of sp just after it got decreased at some position p (assuming all other properties are OK). We repeatedly go to the parent, recompute its sp and switch its children if necessary. We can stop when sp didn't change in the current vertex.

Note that whenever we decreased sp of a vertex, we set it to $1 + \mathsf{sp}$ of the child that we came from. Thus the values of sp (after FIX) on the path that we walked are strictly increasing, and since $\forall v\ \mathsf{sp}(v) \leq \log n$, the time is $\Theta(\log n)$.

DECREASE(H, p, δ): We cut the subtree of p, then FIX H from p's former parent. Now p points to a root, so we can just decrease its priority and MELD the two heaps.

DELETE(H, p): We remove p, leaving its child subtrees with the rest of h, then FIX H from p's former parent. Now MELD the three heaps together (in two operations).

INCREASE(H, p, δ): We cut both children of p and then FIX(H, p). Now p points to a leaf, so we can just increase its priority and MELD the three heaps.

Corollary. *Leftist heaps need $\Theta(n)$ space and* MAKEHEAP *needs $\Theta(n)$ time.* INSERT, DELMIN, DELETE, DECREASE, INCREASE *and* MELD *need $\Theta(\log n)$ time.*

Remark. It is possible to balance the subtrees by number of elements instead of sp ("weight-biased leftist trees"). Experiments suggest that it is faster in practice.