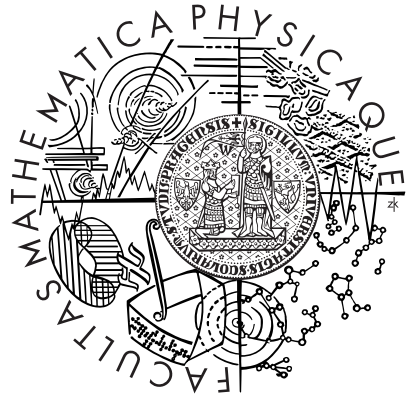
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Jan Bulánek

Datové struktury pro setříděné ukládání dat

Matematický ústav AV ČR, v.v.i.
Vedoucí diplomové práce: Mgr. Michal Koucký, Ph.D.

Studijní program: Softwarové systémy (ISS)

2010

Rád bych poděkoval vedoucímu své práce Michalu Kouckému za trpělivost, podnětné konzultace, připomínky a opravy, bez nichž by tato práce nemohla vzniknout.

Dále bych rád poděkoval své přítelkyni Petře Krouparové za lásku a závěrečné jazykové korekce a Zbyňku Faltovi za nejrůznější připomínky všeho druhu.

V neposlední řadě patří můj dík mým rodičům, kteří mě podporovali celý můj život a umožnili mi tak i napsání této práce.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne Jan Bulánek

Content

Název práce: Datové struktury pro setříděné ukládání dat
Autor: Jan Bulánek
Vedoucí: Mgr. Michal Koucký Ph.D.,
Matematický ústav AV ČR, v.v.i.
e-mail vedoucího: koucky@math.cas.cz

Abstrakt: V předložené práci studujeme dvě varianty přihrádkovací hry. Tato hra je použita v důkazu spodního odhadu časové složitosti vkládání prvků do setříděného pole. Ukážeme, že tyto varianty přihrádkovací hry mají až na konstantní faktor ekvivalentní časovou složitost. Dále ukážeme výhody použití setříděných polí z hlediska vyrovnávacích pamětí. Na závěr ukážeme jednu možnou implementaci vyhledávací datové struktury s použitím pole velikosti $n^{1+\epsilon}$.
Klíčová slova: přihrádkovací hra, udržování setříděného pole

Title: Data structures for file maintenance problem
Author: Jan Bulánek
Supervisor: Mgr. Michal Koucký Ph.D.,
Institute of Mathematics of the AS CR
Supervisor's e-mail address: koucky@math.cas.cz

Abstract: In this work we study two variants of a bucketing game. This game is used for the lower bound proof of time complexity of item insertion into a sorted array, a data structure for the order maintenance problem. We show that these two variants of the bucketing game have the same time complexity up to a constant factor. Then we show that sorted arrays can use cache efficiently for certain operations. Finally, we present one implementation of the order maintenance data structure using the array of size $n^{1+\epsilon}$.

Keywords: bucketing game, ordered array maintenance

Chapter 1

Introduction

In this thesis we study several problems connected to the *order maintenance data structure*. An ordered maintenance data structure is a data structure for storing a linearly ordered set [7]. This data structure maintains a current ordered list of elements. In its basic variant it supports the following operations:

- *insert(x, y)*: insert item y after x into the list
- *delete(x)*: delete item x from the list
- *order(x, y)*: return true if x is before y in the list, otherwise return false

Such a data structure can be realized for example by a linked list or by a binary search tree. However, we focus on a data structure introduced by Itai et al. in [12]. Although the insert operation has $O(\log^2 n)$ amortized time complexity in this data structure, where n is the number of items in the list, it allows an efficient implementation of the following operation:

- *scan(x, y)*: traverse through the items which are after x and before y in the list.

This operation is very important in many practical applications and surprisingly it is performed much faster by the Itai et al. data structure than for example by binary search trees. This is because in the Itai et al. data structure the elements are stored in the array of size $O(n)$ in the same order as in the list. Thus the sequential operations over the list are much faster since the items are stored in the memory consecutively which is good for the behaviour of cache. By

contrast the items stored in the leaves of a binary tree are stored in the memory in random locations and although two items are neighbours in the list they could be "very far" from each other in the memory. This is very bad for cache since there is a high probability of many cache misses during the scan operation. And each cache miss is time consuming. More details will be given in the next chapter, where we introduce some memory models, which give us a toolkit for the precise analysis of algorithm behaviour regarding the cache.

The ideas from the Itai et al data structure [12] are used for example in [4] for developing the dictionary data structure which has optimized performance for all levels of the memory hierarchy even though it has no memory-hierarchy-specific parameterization (using the cache oblivious model). There is another result [3] using this idea and many other results based on the Itai et al. result, which emphasizes the importance of this data structure.

Although this topic is investigated quite extensively there still remain many questions to be solved. The one we find to be the most important is whether there exists a structure similar to the Itai et al. one but with a faster insert operation. The known result is, that an improvement cannot be achieved by algorithms implementing smooth strategies [8, 13] – which is the case of the Itai et al. structure. And even though nonsmooth strategies intuitively seem to be worse than smooth ones, there is no proof of that.

To contribute to the solution of the above problem we study the very same data structure which differs just by the size of the underlaying array which is $O(n^{1+\epsilon})$. It was proved that such data structure cannot perform insertion faster than $O(\log n)$ [9] in worst case. Two possible implementations are given in [7], for $\epsilon \geq 1$ and E. Demaine's lecture notes [6] mention that this should be possible for any $\epsilon > 0$. We are not aware of any literature where such a structure would be described. Thus we develop one in the last chapter of the thesis.

Known limited lower bound are based on analysis of a *bucketing game* [9]. Let us start with the brief description of this game. Imagine that you have k infinitely large buckets and you want to place N items into these buckets. Between any two insert operations, you can perform a merge operation, which means to move some items among different buckets. The cost of every operation is equal to the number of items in involved buckets. You want to minimize the total cost of all insert and merge operations. There are two versions of this game. In the first version (so called *unordered bucketing*) you can insert each item into an arbitrary buckets and also perform merge on an arbitrary subset of buckets. In the second version (which is known as the *prefix bucketing*), the buckets are numbered and you can insert items only to a bucket number one and merge can be performed

only on buckets $1, 2, \ldots, m$ where each merge can choose different m.

We start with the simple upper bound estimate for the prefix (and thus also for the unordered) bucketing.

Claim 1. *The upper bound on the cost of a prefix bucketing of N items into $\log N$ buckets is $O(N \log N)$.*

Proof. The proof is done simply by constructing a strategy producing such bucketing. This strategy is inspired by the algorithm presented in [11] and is performed in the binary counting manner. Recall that buckets are numbered from 1 to $\log N$. Let us define the capacities of the buckets. The capacity of the first bucket is 2 and the capacity of the i-th bucket ($i \neq 1$) is 2^{i-1}. Insert is always performed into the bucket with id 1. If the first bucket is full (i.e. it contains 2 items) we perform a merge operation. Let M be the greatest integer such that all buckets with id $1, 2, \ldots M$ are full. We perform a merge over all the first M+1 buckets and put all the items into bucket $M + 1$.

The cost of all insert operations is $O(N)$ and since every item takes place in at most $O(\log N)$ merge operations (since by every merge in which it occurs, it is moved to the bucket with higher id), the overall cost of such bucketing is $O(N \log N)$. □

In [9] the following estimates are shown.

Claim 2. *The cost of the unordered bucketing of N items into k buckets is $\Omega(N \log N / \log k)$.*

It can be trivially seen that this claim is tight for $k = N$ – we just place one item into every bucket. We can also show, that this estimate is tight for $k = \left\lceil \sqrt{N} \right\rceil$. One possible strategy is to insert two items into each bucket and then merge all items into one arbitrary bucket. Then we insert two items into each of $k - 1$ empty buckets and then we merge items from these $k - 1$ buckets into an arbitrary bucket of these $k - 1$ buckets. Then we repeat the very same for the $k - 2$ remaining empty buckets etc. until we fill all the buckets. After we are finished, the k-th bucket contains $2k$ items, the $(k - 1)$-th bucket contains $2k - 2$ items, the $(k - 2)$-th bucket contains $2k - 4$ items etc. Thus the overall number of items in all the buckets is $2^{\frac{k(k+1)}{2}}$ and since $k = \left\lceil \sqrt{N} \right\rceil$, the number of items in the buckets is at least N. The cost of the unordered bucketing of N items into $\left\lceil \sqrt{N} \right\rceil$ buckets is $O(N)$ since the cost of all insertions is smaller than

$2N$ (the maximal cost of one insertion is 2) and every item is merged exactly once.

However the most important results are for $k = O(\log N)$ buckets. The conjecture stated in [9] says the following:

Conjecture 1. *The cost of the unordered bucketing of N items into $O(\log N)$ buckets is $\Omega(N \log N)$.*

This is much worse than the lower bound on the cost given in Claim 2. The conjecture is surprising in the light of the following result:

Claim 3. *The cost of the prefix bucketing of N items into $k = O(\log N)$ buckets is $\theta(NlogN)$.*

This means that for $O(\log N)$ buckets, the cost of the prefix bucketing would be (if the conjecture is correct) asymptotically equal to the cost of the unordered bucketing. Despite the fact that the unordered bucketing seems to be much stronger than the prefix bucketing. However this is our result presented in the next chapters.

Theorem 1. *Let $L > 1$ be an integer and $C = 0^L$ be an empty initial bucket configuration. Let s be an optimal unordered bucketing strategy. Then there exists an online prefix bucketing strategy s' such that for every number of items N it holds that $3c(s, C, N) + N > c(s', C, N)$.*

Simply, this means, that the cost of the unordered bucketing is equal to the cost of the prefix bucketing up to a constant factor. Thus we can directly infer that Conjecture 1 is correct.

This thesis is divided into several sections. In the first section we introduce some memory models and we show the importance of those models for developing efficient algorithms. In the next section we introduce an order maintenance problem and we show its relationship with the bucketing game. In the fourth section we present our results about bucketing. We start with the detailed description of bucketing strategies, especially of our concept of online and offline strategies. Then we describe a uniform bucketing and the relationship between the unordered bucketing and the uniform bucketing. Finally, we show that any inseparable bucketing can be transformed into prefix bucketing and since our uniform bucketing is inseparable, we obtain a relationship between the unordered bucketing and the prefix bucketing. In the fifth section we introduce an already mentioned order maintenance data structure with an array of size $O(n^{1+\epsilon})$.

Chapter 2

Memory Models

In this chapter we introduce several memory models, in particular a cache oblivious model. Then we show why such models are important for accurate calculations of time complexity according to the computer architecture.

In computer science, memory models are often very simplified. On the other hand if we are interested only in an asymptotic time complexity, this simplification does not matter. However it turns out, that in many practical situations even a constant factor speed up can be very important and then it is necessary to have a more refined model of memory to capture the differences between any two algorithms. Let us introduce a couple of such models, from the easiest one to the best known nowadays.

RAM

Random access machine is the oldest and the most widely used model. It assumes that every piece of memory can be accessed in the same constant time. This model is very simple and well studied. For the most cases, this is a sufficient abstraction, which up to a constant factor gives us an idea about the time an algorithm spends accessing the memory. However, when we have a memory hierarchy, this model cannot express that some memory location can be accessed (much) faster than another. This is solved by the next model.

Two level I/O model

Imagine that you have to sort a huge amount of data, which cannot be placed in the memory and thus it resides on a disk and you have to access the disk from time to time. Then you will be interested in the number of read/write (I/O)

operations between the memory and the disk while processing the data. In [1] they try to minimize the number of these I/O operations. They realized, that new model is necessary to calculate this number since the RAM model does not differentiate between memory and disk access. Thus they introduced a two level memory model. These two levels are the following (we are using the notation introduced by [5] which is clearer than the original one)

- *cache* – (relatively) small (size M) but very fast memory consisting of blocks of size B
- *disk* – unlimited size but very slow

These two levels communicate by sending blocks of size B. A block is sent whenever *a cache miss* occurs, i.e. required data are not stored in the cache. The complexity of the algorithm is then calculated as the number of block transfers (which is equivalent to the number of cache misses) between these two memory levels.

This model can obviously be used not only to cover relationship memory-disk but also cache-memory. However there is a couple of problems. The main problem is, that to obtain an optimal algorithm, the knowledge of M and B is necessary. Unfortunately these differ system to system. But this problem can be solved quite well by simple tests which can give at least some estimates of these values. The second problem is, that this model explicitly assumes, that you have the control over the placement of blocks in the memory and on the disk. Such assumption is very far from reality, where typically you have no control over such things. And finally, this model was developed just for two memory levels which is insufficient if you realize that usually you have two or three levels of cache, a main memory, a disk and often some external storages even slower than the disk.

The cache oblivious model

The main idea behind this model is surprisingly simple – let us forget about M and B and try to write algorithms which use optimal number (up to rounding) of memory transfers between two levels of the memory for each M and B. Notice the interesting consequences, this has. The most important is, that if the algorithm behaves well between two adjacent levels of the memory with unknown M and B it has to behave well between every two adjacent levels of the memory hierarchy. That is, instead of an algorithm with two hardly obtainable parameters, which

is optimal only for specific two levels of the memory, we obtain one algorithm, which is optimal for every two adjacent levels of the memory.

However, several assumptions are necessary. The first one is, that for every two adjacent levels of memory hierarchy cache-disk relationship has to hold. In other words, the closer to CPU, the smaller and faster memory is and vice versa. This assumption is pretty reasonable and no one would probably dispute it, which is not the case of the second assumption.

When the cache (recall that by our definition, a cache is the smaller one of the two adjacent levels in the memory hierarchy) on any level of the memory hierarchy is full and you ask for a block which is not stored in this cache, it is necessary to discard one block stored in the cache and replace it by the demanded block from the disk. The problem is, that you do not have full control about the memory, so you do not know which block will be discarded. Obviously you can develop such discarding strategy, which will perform very badly. Nevertheless, we will assume that our replacing strategy is optimal and the discarded block would be needed farthest in the future. Although this assumption seems to be too strong the following result makes it reasonable.

Lemma 1 (in [10])**.** *If an algorithm makes T memory transfers on a cache of size $M/2$ with optimal replacement, then it makes at most $2T$ memory transfers on a cache of size M with LRU or FIFO replacement strategies (and the same block size B).*

Here LRU denotes the strategy which replaces the least recently used block and FIFO denotes the strategy which replaces the oldest one. In other words, LRU and FIFO replacement performs as well as the optimal replacement up to a constant factor. And since these strategies (or their approximations) are really used in real systems, this assumption seems to be reasonable.

Obviously, if an algorithm is optimal (according to the number of cache misses) in cache oblivious model, it will be also optimal (up to constant) in the two level I/O model.

Cache optimality justification

No matter which model we use, the goal is always the same – minimize the number of memory transfers. But even if we design an optimal algorithm (optimal with respect to the number of memory transfers), we never obtain more than just a constant factor speed up. On the other hand, designing optimal algorithms in cache oblivious model can be quite difficult, and the proofs of their complexity

can be nontrivial. The natural question is, whether such effort is worthwhile since we can (almost) always buy better CPUs. However the problem is, that for last couple of years, there is no significant speed up of a single CPU core and higher performance is obtained by adding more CPU cores. But not all tasks can be split into parallel subtasks and thus even constant factor speedup might be useful. Moreover there is a huge cost for transferring data from RAM to CPU caches. This cost will be always big due to physical limits of used technologies and thus it makes sense to decrease the number of these transfers as much as possible (according to Intel programmers manual, the cost of a transfer between memory and CPU is about 100 CPU cycles). This gives us an idea, that the difference between two algorithms with the same time complexity can be quite large. Thus we consider such techniques to be very useful.

Chapter 3

Order maintenance problem

First we introduce *the online list labeling problem* using a definition from [12]. Let us have N distinct items from a linearly ordered set and a set of "labels" from some discrete linearly ordered set of some limited cardinality. We are getting items one by one in non predictable order. The goal is to maintain an assignment of labels to the received items, so that the labels are ordered in the same way as the items they label. Sometimes this might be impossible since there is no free adequate label. Then we have to change the label assignment (relabel items). Our task is to minimize the total number of item relabellings.

This problem is very interesting, since it can be directly used for creation of the *order maintenance data structure*. The data structure maintains a list of items and supports the following operations [7]:

- *insert(x, y)*: insert item y after x into the list
- *delete(x)*: delete item x from the list
- *order(x, y)*: return true if x is before y in the list, otherwise return false

Such a data structure can be realized for example by a linked list or by a binary search tree. However, we focus for a while on a data structure introduced by Itai et al. in [12]. Although the insert operation has $O(\log^2 n)$ amortized time complexity in their data structure, where n is the number of items in the list, the data structure is interesting as it allows for an efficient implementation of the following operation:

- *scan(x, y)*: traverse through the items which are after x and before y in the list.

And this very important operation can be performed very efficiently by Itai et al. data structure because the number of cache misses caused by this operation is very close to the optimal number of cache misses needed for that operation. This is the reason why the ideas presented in this structure were used in many other implementations of data structures which behave very well to cache [4]. Its good behaviour is caused by the following fact: items are stored in a sparse array with some (predefined and thus constant) minimal and maximal density, in such a way, that maximal gap between two items in the array is constant.

This implies that the scan can be performed with only cn_s/B block transfers where n_s is the number of scanned items, B is the size of each cache block and c is a constant depending on the array density (we are using the cache oblivious model). Detailed calculations can be found in [5]. This is much better than the usual number of block transfers performed for example by red-black trees when performing the scan (since every leaf is stored independently we may have to perform up to n_s block transfers). The trade-off is the slightly higher cost of the *insert* operation which is $O(\log^2 n)$ amortized where n is the number of items in the array.

Now let us briefly describe the basic idea of the data structure. Let us have an array of size S. We divide this array into chunks, each containing $\log_2 S$ slots of the array. These chunks will be the leaves of a binary tree structure built "over" that array. Then one maintains some minimal and maximal density in every node of the tree, i.e. every subtree of the tree cannot contain more (and less respectively) than predefined number of items.

The fundamental question is whether the insertion time can be improved in such a data structure. The following results are known. Let n be the number of items we want to store in the array. If the size of the array, where we store n items is $O(n^{1+\epsilon})$ than the tight upper bound for storing all items is $\Theta(n \log n)$ [9] (In chapter 5, we present an implementation of this data structure). If the size of the array is $O(n)$ the upper bound is $O(n \log^2 n)$ (using the structure we just described) which is tight for smooth strategies [8, 13]. Finally if the size of the array is exactly n the upper bound is $O(n \log^3 n)$ [2]. The most interesting result for us, is the first one [9], which introduces so called *bucketing game*. We hope that better understanding of this game will lead to subsequent results in this area.

Bucketing

Let us introduce this game.

Imagine that you have L infinitely large buckets $A_1, A_2 \ldots, A_L$ and N items. Your task is to insert these items into those buckets arbitrarily. The cost of an

insertion of one item into the i-th bucket A_i is equal to $a_i + 1$ where a_i denotes the number of items already in the bucket A_i. Such task is easy – obviously the optimal strategy of distributing items evenly leads to $\Theta(N^2/L)$ time complexity. However the game starts to be interesting when a *merge* operation is introduced. Now between every two insertions one can perform a merge operation, which means that one chooses an arbitrary subset of buckets and rearranges their items among themselves. The cost is again equal to the number of items in the buckets involved in the operation (not to the number of items which are actually moved). Such game is called *an unordered bucketing game*.

This game has a variant called *a prefix bucketing problem*. Let us add an ordering of the buckets. Then insert can be performed only into the "smallest" bucket (according to the ordering) and merge can be performed only on the prefix of the bucket list.

The following theorem holds [9].

Theorem 2. *When the number of labels is $O(n^{1+\epsilon})$ for some $\epsilon > 0$, the worst case cost for online labeling of n items is at least proportional to the cost for the prefix bucketing of n items into $O(\log n)$ buckets.*

In the rest of the chapter, we provide a brief overview of the proof. The proof shows, that the time complexity of the labeling has to be at least as high as the cost of prefix bucketing of n items into $O(\log n)$ buckets. Then it is shown that the cost of this prefix bucketing cannot be smaller than $O(n \log n)$.

First notice, that we can easily transform the labeling task into an item insertion into an array. Let us have a set of labels $1, 2, \ldots M$ (which we can assume, since we can rename labels to obtain this). Let us consider an array of size M. When we label an item with a label i, it will be simulated by inserting the item into the i-th slot of the array. Relabeling is equivalent to moving an item from one slot of the array to another one. In the following text we will use the equivalence of these two tasks and we will use both terminologies.

Imagine that you want to cause as many relabelings as possible. Intuitively, the best way to do this is to insert new items into the part of the array which already contains many items. If we define *a density* of the continuous interval of the array as the fraction of the number of items in this interval and the length of this interval, you want to insert items into the interval with a high density. However such definition depends on the choice of the interval and it is not clear which interval to choose. This problem is solved by the next lemma, which shows that there always exists a position in the array, such that every interval containing this position has a large density.

Lemma 2. *Consider any nonnegative, integrable function f on the interval [0,1]. For each (nontrivial) subinterval $I \subseteq [0,1]$, define*

$$\rho(I) = \frac{1}{|I|} \int_I f(x)dx.$$

Then there is some point $x_0 \in [0,1]$ such that $\rho(I) \geq \frac{1}{2}\rho([0,1])$ holds whenever I includes x_0. The point x_0 is called dense point.

The importance of this lemma is obvious. We can always find a position in the interval $[0,1]$, such that every interval containing this point has the density comparable to the density of whole interval. This lemma can be extended for the labelings. Let us have a set of m labels. We define a function f that is either 0 or 1 on the interval $[0,1]$ and 0 everywhere else. We divide interval $[0,1]$ into m equally long subintervals and the value of f in the i-th such subinterval is one if the i-th label is used and zero otherwise.

The above lemma implies the following claim.

Claim 4. *In each labeling, there is a used label in the middle third of the used labels (where rounding is in favor of that middle third) such that every label-space subinterval containing that label is at least one-sixth as dense as the entire label space.*

Let us have an arbitrary strategy and a set of labels where some labels are already used. If we want to make the strategy to do as many relabelings as possible, it seems to be reasonable to choose every new item in such a way that it has to obtain a label that is in the most dense interval. Lemma 2 gives us an idea of how to choose such an item. However it is not enough to choose an item which will be inserted to an arbitrary dense point, since if we choose such dense points for example in round-robin fashion, the strategy may perform all inserts without any relabeling. Thus the following construction is provided.

We construct a sequence of a label-space intervals $I_1 \supset I_2 \supset \ldots \supset I_k$ using the following notation – the number of used labels in each interval I is denoted by $\text{used}(I)$ and the total number of labels in each interval I is denoted by $\text{total}(I)$. Furthermore if $I' \subset I$, then the difference $I - I'$ consists of at most two intervals (left and right). We denote the number of used labels as $\text{leftused}(I - I')$ and $\text{rightused}(I - I')$. Intervals $I_1, I_2, \ldots I_k$ are chosen to satisfy the following rules:

1. I_1 is the whole label space.
2. $\text{used}(I_k) = O(1)$.

3. $\forall i$, total$(I_{i+1}) \leq$ total$(I_i)/2$.

4. $\forall i$, used$(I_{i+1}) = \Omega($used$(I_i)/2)$.

5. $\forall i$, leftused$(I_i - I_{i+1}) = \Theta($rightused$(I_i - I_{i+1}))$.

There are such intervals $I_1, I_2, \ldots I_k$ for $k = O(\log n)$. Additionally we can assume, that every strategy always relabels some continuous interval containing a label of the last inserted item (other relabelings can be done later). Let us choose an item for insertion, such that it obtains a label belonging to I_k. Because of the conditions 4 and 5, we know, that the number of relabelings performed by the strategy is at most constant times smaller than the number of the used labels in the smallest interval containing the interval affected by the strategy. Therefore, if we consider the differences $I_i - I_{i+1}$, $i = 1, 2, \ldots, k-1$ and the interval I_k to be the buckets, then the strategy solves the prefix bucketing problem. It remains to show, that after the insertion or merge is performed, we are able to restore buckets in the affected interval. This is ensured by the following lemma.

Lemma 3. *Each sufficiently long interval I containing sufficient number of used labels contains a subinterval I' such that*

- $\forall i$, *total*$(I_{i+1}) \leq$ *total*$(I_i)/2$.
- $\forall i$, *used*$(I_{i+1}) = \Omega($*used*$(I_i)/2)$.
- $\forall i$, *leftused*$(I_i - I_{i+1}) = \Theta($*rightused*$(I_i - I_{i+1}))$.

Thus after every insertion or merge, the buckets can be restored and thus we can determine the next item for insertion.

This sketch of the proof from [9] gives us a rough idea of relationship between the labeling and the bucketing. We do not present a proof of the lower bound for the prefix bucketing of n items into $O(\log n)$ buckets as it is unrelated to our proofs and can be found in [9]. We use a different approach to obtain a lower bound on the cost of the unordered bucketing (we construct a reduction of the unordered bucketing to the prefix bucketing).

Chapter 4

Bucketing

In this chapter we would like to show the relationship between an unordered bucketing and a prefix bucketing. We also provide a precise definition of the bucketing strategies, which we find to be very interesting and important.

We start with the precise definition of the unordered bucketing problem.

Definition 1. *Let us have an infinitely large set I of mutually undistinguishable items and L ($L > 1$) buckets $A_1, A_2, \ldots, A_L$ such that each bucket can store infinitely many items from I. The number of items in the bucket A_i is denoted by a_i. Let $S = \{j_1, j_2, \ldots, j_n\}$ be a subset of $\{1, 2, \ldots, L\}$. Then A_S denotes the set of the buckets $A_{j_1}, A_{j_2}, \ldots, A_{j_n}$ and $|A_S|$ denotes $\sum_{i \in S} a_i$.*

Now let us define two operations: an insert *and a* merge.

- *An insert operation with a parameter $i \leq L$ moves one arbitrary item from I into the bucket A_i. The cost of such operation is equal to the number of items in A_i after the insert was performed.*
- *A merge operation for a given set $S \subset \{1, 2, \ldots, L\}$ moves arbitrary items among buckets $A_{j_1}, A_{j_2}, \ldots A_{j_n}$ where $S = \{j_1, j_2, \ldots j_n\}$. So the parameters of the merge are the set of the buckets and the set of the moves of the items among those buckets (indeed, as the items are indistinguishable, only the final number of items in every bucket after the operation matters). The cost of such operation is equal to $|A_S|$.*

Finally a bucketing *denotes a sequence of insert and merge operations with their parameters over the given buckets $A_1, A_2, \ldots, A_L$ and the set of items I. The cost of the bucketing B is equal to the sum of the costs of all the performed operations and we denote it as $c(B)$.*

We allow bucketing to start with non-empty buckets.

Notice that the cost of merge operations does not depend on the number of items which were actually moved.

The simple observation about the lower bound of the cost of the bucketing can be done immediately.

Observation 1. *Let B be an arbitrary bucketing and N denote the number of insert operations in it. Then $c(B) \geq N$.*

Definition 2. *Let us have L buckets containing some items of I. Then by the configuration of the buckets we mean the distribution of items in the buckets, i.e. the configuration is given by an L-tuple $(a_1, a_2, \ldots, a_L) \in \mathbb{N}^L$ where a_i is the number of items in each bucket A_i.*

Definition 3. *Let us have L buckets and the bucketing B. The distribution of items, after i operations of B was performed, is called the configuration of the bucketing B after i steps. The distribution before any operation of B was performed is called the initial configuration of B.*

4.1 Bucketing strategies

From the given definitions, it is obvious, that the order of the operations and their parameters influence the cost of the bucketing significantly. Thus in order to obtain the best results, we should choose a strategy very carefully. Unfortunately, the intuitive definition of a *strategy* can be misleading. Therefore we provide two definitions which differ in one important aspect – whether the overall number of items we want to store in the buckets (which consists of the items contained in the initial configuration and the items we want to insert) is known in advance.

Definition 4. *An* online strategy *is a function, which for any $L > 1$ and any bucket configuration $C \in \mathbb{N}^L$ returns an operation (either an insert or a merge) and its parameters.*

If we know the number of inserted items in advance, the following definition takes it into account.

Definition 5. *An* offline strategy *is a function, which for any $L > 1$, any bucket configuration $C \in \mathbb{N}^L$ and any number M of item we want to store eventually returns a bucketing with the initial configuration C and the final configuration containing M items.*

The natural motivation for these two definitions is the fact, that you will probably perform different operation if you know the number of items you have to insert than if you do not know it. For example, if you know that there remains only one item to insert, you will not perform merge among all the buckets. In other words, offline strategies have more information and this information can improve their behaviour compared to online strategies. Surprisingly, under certain circumstances the difference is not as large as one could expect.

Let us show, how an online strategy induces bucketing for any initial configuration. Let $L > 1$ be an integer, $C_1 \in \mathbb{N}^L$ be an initial bucket configuration and s an online strategy. Then s gives us for the configuration C_1 an operation and its parameters (let us denote them as O_1). If we apply this operation O_1 to the configuration C_1 we obtain a new configuration C_2. Generally by applying s to the configuration C_i we obtain an operation with parameters O_i. By applying the operation O_i to the configuration C_i we obtain a new configuration C_{i+1}. We continue until we insert the intended number of items.

Obviously the sequence of operations $O_1, O_2, \ldots$ and the initial configuration C_1 define some bucketing B. The number of operations in the bucketing B depends on the strategy s, we only know, that the number of insert operations will be exactly N. Notice, that if the strategy s is deterministic, the induced bucketing B for the given C_1 will be always the same.

The result of this paragraph is summarized in the following definition:

Definition 6. *Let $L > 1$ be an integer, $C \in \mathbb{N}^L$ be a bucket configuration and N the number of items we are about to insert. Let s be an arbitrary strategy. Then the strategy s, the initial configuration C and N uniquely determine one specific bucketing. We denote such bucketing by $B_{s,C,N}$.*

Notice that the offline strategies really obtain all necessary information, since the number of items already inserted in the buckets can be obtained from the configuration C. On the other hand, if s is online, it does not know about N and this number is provided only for the purpose of analysis. If the number of inserted items is undefined, we cannot compare two strategies which will be shown later.

Now we can define a cost of a strategy, which becomes our tool for comparing the quality of two strategies. We will use the cost of bucketings they induce when inserting N items.

Definition 7. *Let $L > 1$ be an integer, $C \in \mathbb{N}^L$ be a bucket configuration and N the number of items we are about to insert. Let s be an arbitrary strategy. Let*

$B_{s,C,N}$ be the bucketing determined by s, C and N. Then the cost of the strategy s to insert N items to C is equal to the cost of the bucketing $B_{s,C,N}$. We denote that cost by $c(s, C, N)$.

It turns out that we can easily define optimality (according to the cost) of offline strategies but as shown below the case of online strategies is more complicated.

Definition 8. *Let S denote the set of all existing strategies. Then the strategy s is optimal if for any $L > 1$, any configuration $C \in \mathbb{N}^L$, any strategy $s' \in S$ and any number of inserted items N, $c(s, C, N) \leq c(s', C, N)$.*

Let $L > 1$ be an integer. The strategy s is optimal for a configuration $C \in \mathbb{N}^L$ if for any strategy $s' \in S$ and any number of inserted items N, $c(s, C, N) \leq c(s', C, N)$.

Analogously to asymptotical time complexity we define relation between two strategies which has a similar meaning.

Definition 9. *Let us have two arbitrary strategies s_1, s_2 and $L > 1$. Then we say that s_1 is at most k times worse than s_2 if for any configuration $C = \mathbb{N}^L$ and any number of inserted items N, $c(s_1, C, N) \leq kc(s_2, C, N)$.*

Let $L > 1$ and $C' \in \mathbb{N}^L$ be a bucket configuration. s_1 is at most k times worse than s_2 for the configuration C' if for any number of inserted items N, $c(s_1, C', N) \leq kc(s_2, C', N)$.

The following claim shows, that we can create optimal strategy.

Claim 5. *There exists an optimal offline strategy.*

Proof. It is easy to prove that there exists an optimal offline strategy as for every C and for every N you can find the bucketing with the minimal cost among all the bucketings and thus determine your strategy for every C and N. □

However the situation is not so easy for online strategies. Let us show a construction of reasonably good online strategy (which means good enough for our further purposes). We start with the claim, which helps us to understand the behaviour of offline strategies.

Definition 10. *Let $L > 1$ be an integer and $C \in \mathbb{N}^L$ be a bucket configuration. Let s be an arbitrary strategy and N denote the number of inserted items. Then $B_{s,C,N}(k)$ denotes the smallest prefix of the bucketing $B_{s,C,N}$, that contains k insert operations (recall that bucketing is a sequence of the insert and merge operations).*

Notice that $B_{s,C,N}(k)$ is also bucketing and thus we can directly apply the definition of the cost of the bucketing to it.

Claim 6. *Let s be an optimal offline strategy. Then for any $L > 1$, any configuration $C \in \mathbb{N}^L$, any number of inserted items N and an arbitrary integer N' such that $N' > N$ it holds that $c(B_{s,C,N}(N)) + M \geq c(B_{s,C,N'}(N))$ where M is N plus the number of items in C.*

Proof. Let us denote the configuration obtained by the bucketing $B_{s,C,N}(N)$ as C_1 and the configuration obtained by $B_{s,C,N'}(N)$ as C_2. Let us assume that

$$c(B_{s,C,N}(N)) + M < c(B_{s,C,N'}(N)).$$

Then let us denote as B' the following bucketing: first we perform $B_{s,C,N}(N)$ and thus we obtain C_1. The we perform merge over all buckets in such a way, that we obtain C_2. The cost of such merge is clearly M. Finally we continue using the bucketing $B_{s,C,N'}$ since the configuration of the buckets is very same as if we have performed $B_{s,C,N'}(N)$. Obviously, the cost of B' is smaller than the cost of $B_{s,C,N'}$ which is a contradiction since s is optimal for C. □

The meaning of this claim is very important: any prefix of the optimal bucketing cannot be much more expensive than the optimal bucketing for a smaller number of items. It will be very important in the proof of the next lemma, where we use this claim to construct an online strategy.

The next lemma shows us the relationship between online and offline strategies. Notice that not only we show this relationship, but the proof itself shows us the way how to construct the "good" online strategy.

Lemma 4. *Let s be an optimal offline strategy. Then there exists an online strategy s' such that for $L > 1$, a bucket configuration $C \in \mathbb{N}^L$ and an arbitrary number of inserted items N, $c(B_{s,C,N}) + O(M) \geq c(B_{s',C,N})$ where M is N plus the number of items in C.*

Proof. First notice, that an arbitrary sequence of consecutive merge operations can be replaced by one merge operation, which performs all moves in one step. For simplicity we assume that every bucketing is a sequence of couples consisting of an insert operation and a merge operation (where the merge operation can move zero items in which case we count its cost to be zero). We will assume that every bucketing in this proof follows this assumption.

Recall that s is the optimal strategy. Let us define the following sequence of configurations $C_0, C_1, C'_1, C_2, C'_2, C_3, C'_3 \ldots$. Let C_0 be an empty initial configuration. Then C_1 is obtained as the final configuration of $B_{s,C_0,1}(1)$. Now consider

C'_1 which is the last but one configuration obtained by $B_{s,C_0,2}(2)$. C'_1 can be obviously obtained from C_1 by a merge performed on all buckets. Now we consider C_2 which is obtained as the final configuration of $B_{s,C_0,2}(2)$ and C'_2 which is the last but one configuration obtained by $B_{s,C_0,4}(3)$. Again we can obtain C'_2 from C_2 by a merge operation on all buckets. C_3 will be the final configuration of $B_{s,C_0,4}(3)$ and C'_3 is obtained by a consecutive merge operation performed by $B_{s,C_0,4}$. C_4 is the final configuration of $B_{s,C_0,4}(4)$ etc.

Generally for $i \neq 2^k$, any integer k and ℓ such that $2^{\ell-1} \leq i < 2^\ell$, C_i is obtained as the final configuration of $B_{s,C_0,2^\ell}(i)$ and C'_i is equal to the last but one configuration obtained by $B_{s,C_0,2^\ell}(i+1)$. For $i = 2^k$ for an integer k, C_i is obtained as the final configuration of $B_{s,C_0,i}(i)$ and C'_i is the last but one configuration obtained by $B_{s,C_0,2i}(i+1)$. Obviously the sequence of configurations $C_0, C_1, C'_1, C_2, C'_2, C_3, C'_3 \ldots$ defines a bucketing and we denote it by B^*. It is important to realize, that B^* is in fact obtained from the operations of another bucketings. Also notice, that for $i = 2^k$ we actually "switch" from one bucketing to another.

Now we show several facts about the cost of B^*. First we calculate the cost of the additional merges (i.e. merge operations performed to obtain C'_i from C_i for $i = 2^k$). These operations are added to "switch" from one bucketing to another one and are not part of any bucketing used to obtain B^*. First notice, that the cost of every such merge on the configuration C_i is equal to i since the number of items in C_i is i. Furthermore, such merge is performed only for each $i = 2^k$. Let N_f be the number of inserted items and ℓ be the greatest integer such that $2^\ell \leq N_\mathrm{f}$. Then the cost of all additional merges is equal to $\sum_{k=1}^{\ell} 2^k$ which is at most $2N_\mathrm{f}$. In other words, the cost of the additional merge operations is at most two times larger than the number of items placed in the buckets.

Now we will focus on the difference between $c(B^*(N_\mathrm{f}))$ and $c(B_{s,C_0,N_\mathrm{f}})$ where N_f denotes the number of inserted items. Notice, that since s is optimal, for integers j, j' such that $j < j'$, it holds that

$$c(B_{s,C_0,j}(j)) \leq B_{s,C_0,j'}(j)).$$

Let ℓ be the smallest integer such that $2^\ell \geq N_\mathrm{f}$. From the construction of B^*, we can infer

$$2N_\mathrm{f} + c(B_{s,C_0,2^\ell}(N_\mathrm{f})) - c(B_{s,C_0,2^\ell}(2^{\ell-1})) + c(B_{s,C_0,2^{\ell-1}}(2^{\ell-1})) - c(B_{s,C_0,2^{\ell-1}}(2^{\ell-2})) +$$

$$c(B_{s,C_0,2^{\ell-2}}(2^{\ell-2})) - c(B_{s,C_0,2^{\ell-2}}(2^{\ell-3})) \ldots \geq c(B^*(N_\mathrm{f})),$$

where $2N_{\mathrm{f}}$ stands for the additional merges. Together with the previous observation, we directly obtain that

$$c(B_{s,C_0,2^\ell}(N_{\mathrm{f}})) + 2N_{\mathrm{f}} \geq c(B^*(N_{\mathrm{f}})).$$

From Claim 6 we know that

$$c(B_{s,C_0,N_{\mathrm{f}}}(N_{\mathrm{f}})) + N_{\mathrm{f}} \geq c(B_{s,C_0,2^\ell}(N_{\mathrm{f}}))$$

which implies that

$$c(B_{s,C_0,N_{\mathrm{f}}}(N_{\mathrm{f}})) + 3N_{\mathrm{f}} \geq c(B^*(N_{\mathrm{f}})).$$

Now we use B^* for creating an online strategy s' as follows – let C_{a} be an arbitrary configuration and N_{a} be the number of items in C_{a}. If $C_{\mathrm{a}} = C'_{N_{\mathrm{a}}}$ then s' performs an insert so that it obtains $C_{N_{\mathrm{a}}+1}$. Otherwise it performs the cheapest possible merge on C_{a} so that it obtains $C'_{N_{\mathrm{a}}}$. If $C_{\mathrm{a}} = C_{N_{\mathrm{a}}}$, the operation used by B^* to obtain $C'_{N_{\mathrm{a}}}$ may be different, however the cost of the operation returned by s' is not greater and the result is the same. Notice that s' converts in the first step the initial configuration into one of the configurations of B^* and than it just follows the operations of B^*.

Now we calculate the cost of the bucketings obtained by s' and we compare it to those obtained by the optimal strategy s and thus we show that s' satisfies the lemma. Recall that C is the initial configuration, N is the number of items we are about to insert and M is N plus the number of items in C.

Let N_C denote the number of items in C. First we assume that $C = C_{N_C}$. First notice that

$$c(B_{s,C,N}) + N_C \geq c(B_{s,C_0,M}) - c(B_{s,C_0,N_C}).$$

Otherwise the bucketing obtained by connecting $B_{s,C,N}$ and B_{s,C_0,N_C} would have smaller cost than $B_{s,C_0,M}$ which is the contradiction to the optimality of s. From optimality of s also follows that

$$c(B^*(M)) - c(B^*(N_C)) \leq c(B^*(M)) - c(B_{s,C_0,N_C})$$

and from the above paragraph we can infer

$$c(B^*(M)) - c(B^*(N_C)) \leq c(B_{s,C_0,M}) + 3M - c(B_{s,C_0,N_C})$$

and thus

$$c(B^*(M)) - c(B^*(N_C)) \leq c(B_{s,C,N}) + N_C + 3M.$$

But the left side of this inequality corresponds to the cost of $B_{s',C_{N_C},N}$ since it is exactly the cost of the part of B^* performed by s'. It only remains to add the cost for the initial merge in case that $C \neq C_{N_C}$. However this cost is at most N_C and since $N_C \leq M$ the proof is finished.

□

This claim does not seem to be very useful and in fact, for nonempty C it is not unless we insert at least N_C (the number of items in C) items. But for the empty configuration C, the number of inserted items is equal to the number of items when the bucketing is finished, thus the additional cost for using non-optimal strategy is constant per inserted item. Therefore we directly obtain that s' is just constant times worse than s, which might be surprising. In other words, we just proved, that if we are inserting to initially empty buckets, we can create online strategies, which are "asymptotically" optimal (i.e. at most a constant factor worse) for empty initial configuration.

It remains to be seen whether such a strategy is good enough, since it is not clear, whether online strategies cannot be optimal. This question is answered in the next lemma.

Claim 7. *Let s be the online strategy. Then s cannot be optimal.*

Proof. Let us assume that there exists an optimal online strategy s. Let $L > 1$ and $C \in \mathbb{N}^L$ be an arbitrary initial bucket configuration.

First assume that s never performs nontrivial merge operation (i.e. a merge operation in which at least 2 nonempty buckets are involved). Notice, that such strategy s has to always insert new item into the bucket with the smallest number of items (otherwise this strategy cannot be optimal). Thus we obtain exactly one specific strategy. Now it is easy to find some strategy s_{f} such that for some number of inserted items N is $c(B_{s,C,N}) > c(B_{s_{\mathrm{f}},C,N})$.

We do it as follows. Let us have L buckets each containing N_L items where $N_L > 3$. Notice, that after sufficient number of insertions performed by s, we obtain a configuration C_{N_L} of the buckets, such that every bucket contains the same number of items and $N_L > 3$ (recall, s always inserts new item into the smallest bucket). Now we define a better strategy s_{f}. Let s_{f} be a strategy such that obtains the configuration C_{N_L} for the same cost as s (for example by doing the same operations as s). Let us now define next decisions of s_{f}. Let s_{f} merge two arbitrary buckets in such a way that it puts all the items from those buckets into one of those bucket. The cost of such merge is $2N_L$. Then s_{f} can make next 3 insertions so that the cost of them is 6 (it inserts all three items into the empty

bucket). So in total the cost of s_{f} for the last 3 insertions is $2N_L + 6$. However s does not perform any merge and thus its cost for the 3 insertions is $3N_L + 3$. And since $N_L > 3$, we obtain that $3N_L + 3 > 2N_L + 6$. Therefore the strategy s_{f} is cheaper for the last three insertions than the strategy s and since their cost before these insertions was the same we obtain a contradiction to the fact, that s is optimal.

Now consider a strategy s that *will* perform nontrivial merge. Let N denotes the number of items inserted into C before the first nontrivial merge was performed by the strategy s. Let C_M be the configuration just before the first merge was performed. Let A denote a set of buckets of C and $B_{s,C,N}$ be the bucketing determined by s, C, and N and let s' denote an online strategy such that the bucketing $B_{s',C,N}$ is equal to $B_{s,C,N}$. Then it obviously holds that $c(s, C, N) = c(s', C, N)$. Let A_{S_M} denote the subset of buckets of the configuration C_M for which this first merge was performed. Then the cost c_M of this first merge is by definition equal to $|A_{S_M}|$. Thus we can infer that

$$c(s, C, N + 1) \geq c(s, C, N) + c_M + 1$$

where the 1 stands for the minimal cost of the insertion (when we are inserting into empty bucket).

On the other hand, let $j \in S_M$. Then it obviously holds that $a_j + 1 \leq |A_{S_M}|$. Therefore if s' performs just insertion into a_j in the configuration C_M we can infer that

$$c(s', C, N + 1) = c(s', C, N) + a_j + 1$$

which implies that

$$c(s', C, N + 1) < c(s, C, N + 1)$$

and we obtain a contradiction since s is not the cheapest strategy for $N + 1$.

□

Actually, this claim is not very surprising. Intuitively, we expect, that sometimes every strategy has to do expensive operations, to save more steps in the future. However if you do not know the number of inserted items in advance, you can hardly calculate, whether such an operation is profitable.

Using this definitions and claims, we are ready to state our results in the rest of the chapter.

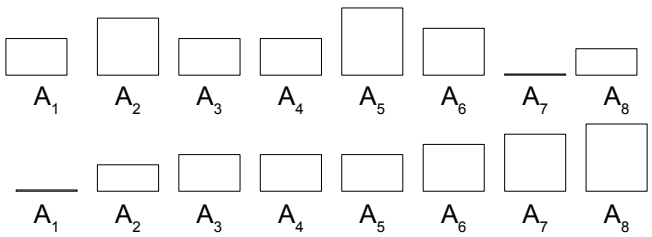


Figure 4.1: Reordering of the buckets according to their size.

4.2 Uniform bucketing

In order to transform the unordered bucketing to the prefix bucketing we introduce *a uniform bucketing* (or *a uniform strategy* respectively). This bucketing puts some limitations on insert and merge operations. On the other hand it still use the ability of the unordered bucketing, which is the insertion of an item into an arbitrary bucket and a merge of an arbitrary subset of buckets.

Our goal of this section is to define a uniform bucketing and show that for every strategy for the unordered bucketing there exists a uniform strategy that is at most 3 times worse than the original strategy.

Let us introduce a *uniform* bucketing strategies. Notice that buckets of a uniform strategies are denoted by D.

The strategy over the set of buckets D is called uniform if it has the following properties.

(i) For any set of buckets D_S used for a merge operation, it holds that if $D_i \in D_S$ and $D_j \notin D_S$ then $d_i \leq d_j$. In other words D_S is always the set of the "smallest" buckets.

(ii) Insert is always performed to a bucket with the smallest number of items.

The next theorem shows the relationship between a uniform bucketing strategy and an arbitrary bucketing strategy.

Theorem 3. *For every offline bucketing strategy s, there exists a uniform strategy s' that is offline and at most* 3 *times worse than s.*

To prove this theorem, we will show how to construct s' based on s. Then we show that the cost of s' is at most constant time greater than the cost of s. This is very useful, since if we find the optimal strategy for an arbitrary number of inserted items, buckets and an arbitrary configuration, we automatically obtain "very good" uniform strategy (which we use in the next section). In the following text, the strategy s will be called *original strategy* and will be used to obtain the uniform strategy s'.

The main rules for the construction mentioned above are the following.

a) For each insert operation of the original strategy the new strategy s' performs an insert and possibly a merge. The cost of these two operations is smaller than the cost of the original operation up to a factor 3.

b) For each merge operation of the original strategy s' performs a merge and the cost of this operation is smaller than the cost of the original operation up to a factor 3.

If we keep these rules, then s' will be at most 3 times worse than s.

Let us now introduce some notation which helps us to describe new strategy s'.

Notation 1. *Let us have L_1 ($L_1 > 1$) infinitely large buckets $A_1, A_2, \ldots, A_{L_1}$ for the original strategy and L_2 ($L_2 > 1$) infinitely large buckets $D_1, D_2, \ldots, D_{L_2}$ for our new strategy. In the following section the i-th bucket of the original strategy will be denoted by A_i before an operation is performed and A'_i after the operation is performed. The number of items in the bucket A_i is denoted by a_i. The configuration of all the buckets of the original strategy is denoted by A or A' respectively. Let $S = \{A_{j_1}, A_{j_2}, \ldots, A_{j_n}\}$ be a subset of $\{1, 2, \ldots, L_1\}$. Then A_S denotes the set of buckets $A_{j_1}, A_{j_2}, \ldots, A_{j_n}$. $|A_S|$ denotes $\sum_{i \in S} a_i$. Analogously we define notation for a new strategy, but we are using d, D and L_2 instead of a, A and L_1.*

Remark 1. *For simplicity, whenever we talk about buckets in this section we assume that they are sorted unless otherwise stated. We sort buckets according to the number of items in them, so the smallest buckets are in the "left" as shown in Fig. 4.1.*

Let us now describe how to obtain a uniform strategy for the given original strategy so that we fulfill the requirements of Theorem 3. Our uniform strategy s' will make its decisions according to the operations given by s (as described

above). Since there are only two possible operations s can use (either a merge or an insert operation), we can define s' as follows.

Definition 11. *Let $L > 1$ and $D \in \mathbb{N}^L$ be an arbitrary configuration of buckets and $n > 0$. Then we define* merge* *operation with parameters n and D in the following way. Let m be a number such that $|D_{\{1\ldots m\}}| < 2n$ and m is the greatest possible. Then the merge* will move all the items from $D_{\{1\ldots m\}}$ to an arbitrary bucket of $D_{\{1\ldots m\}}$. Obviously the cost of merge* is smaller than $2n$ (it just performs merge on $D_{\{1\ldots m\}}$).*

Definition 12. *Let B be an arbitrary bucketing over buckets A with an initial configuration C. Then uniform*(B) over buckets D is the bucketing with the initial configuration C obtained from B as follows. Let us take all operations of B one by one and for each such operation O we add some operations to uniform*(B) using these rules:*

i) Let O be an merge operation on the set A_S. Let $n = |A_S|$. Then we add the merge(n, D) operation to uniform*(B).*

ii) Let O be an insert operation to a bucket A_i. Then we add an insert to the smallest bucket of D and the merge$(a_i + 1, D)$ to uniform*(B).*

For an offline strategy s, a uniform(s) strategy is the strategy s' obtained from s by replacing each bucketing B returned by s by uniform*(B).*

Now we show that for each bucketing B it holds that $3c(B) \geq c(\text{uniform}^*(B))$. We define an invariant that will hold during all operations of the original bucketing B and the uniform*(B) bucketing. In other words, we perform one operation of the original bucketing B, according to this operation the uniform*(B) strategy performs some operations and then we compare the configuration obtained by B with the configuration obtained by the uniform*(B) bucketing whether the invariant is satisfied. Then we continue with the next operation of the original bucketing B.

Recall that in the following text we also assume that the buckets of configurations A, A', D and D' are sorted and numbered in the ascending order (similarly to Fig. 4.1).

Now we can define the invariant:

Definition 13. *Let $L > 1$ and $A \in \mathbb{N}^L$ and $D \in \mathbb{N}^L$ be the configurations of buckets. Then we say that A and D satisfy the invariant if*

$$\forall t \leq L, \sum_{i=1}^{t} a_i \geq \sum_{i=1}^{t} d_i$$

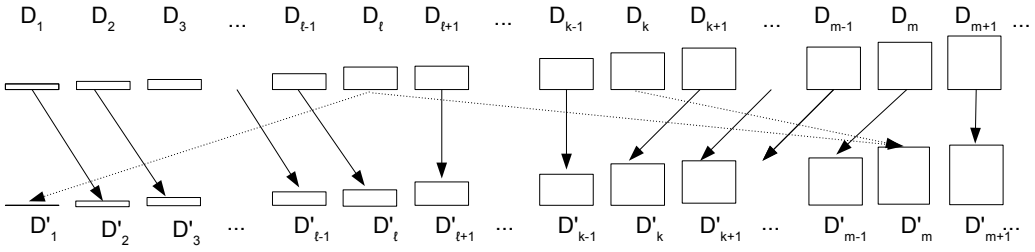


Figure 4.2: Relationship between D and D' in the proof of Claim 8.

The following claims show us several operations over bucket configurations that preserve the invariant.

Claim 8. *Let A and D be configurations of buckets which satisfy the invariant. Let $A' = A$ and let D' be obtained from D in such a way that we move all items from a bucket D_ℓ to a bucket D_k where $\ell < k$. Then A' and D' satisfy the invariant.*

Proof. After we move all items from the bucket D_ℓ to D_k we obtain new buckets D'_1 and some D'_m where $d'_1 = 0$ and $d'_m = d_\ell + d_k$ while losing D_ℓ and D_k. Let us show the relationship between D and D' because $A = A'$. As we can see in Fig. 4.2, where the first line stands for the original configuration and the second line stands for the newly obtained configuration, buckets can be divided into four groups.

For $1 \leq j < \ell$, it holds that $D_j = D'_{j+1}$ since the empty bucket D'_1 was put in front of all buckets. For $\ell < j < k$, $D_j = D'_j$. For $k < j \leq m$, $D_j = D'_{j-1}$ and finally for $j > m$, $D_j = D'_j$.

It can be easily seen that for $t \geq m$, $\sum_{i=1}^{t} d_i = \sum_{i=1}^{t} d'_i$ since all items were moved only between the buckets with indexes smaller or equal to m. Since $d'_1 = 0$ it also holds that for $t < \ell$, $\sum_{i=1}^{t} d_i = \sum_{i=1}^{t+1} d'_i$. In particular it holds that $\sum_{i=1}^{\ell} d_i = d_\ell + \sum_{i=1}^{\ell} d'_i$. This implies $\sum_{i=1}^{t} d_i \geq \sum_{i=1}^{t} d'_i$ for $t \leq \ell$. Furthermore this can be directly extended to the interval $\ell < t < k$ because of the relation between D and D' in this interval. Thus we obtain for $1 < t < k$, $\sum_{i=1}^{t} d_i \geq \sum_{i=1}^{t} d'_i$.

The remaining case of $k \leq t < m$ is the most interesting. Recall that for $k \leq j < m$ it holds that $d_j \leq d'_j$. Assume the existence of t' such that $k \leq t' < m$, $\sum_{i=1}^{t'} d_i < \sum_{i=1}^{t'} d'_i$. From the relation between D and D' we know that

$\sum_{t'+1}^{m} d_i \leq \sum_{t'+1}^{m} d'_i$. However then we cannot obtain $\sum_{i=1}^{m} d_i = \sum_{i=1}^{m} d'_i$ which we know is true. Thus we know that $\forall t$, $\sum_{i=1}^{t} d'_i \leq \sum_{i=1}^{t} d_i \leq \sum_{i=1}^{t} a_i = \sum_{i=1}^{t} a'_i$. So the invariant is preserved in all four intervals. □

Claim 9. *Let A and D be configurations of buckets which satisfy the invariant. Let S be the subset of $\{1, 2, \ldots, L\}$ and k be a number such that $\forall i \in S, i < k$. Let $A' = A$ and let D' be obtained from D in such a way that we move all items from the buckets D_S to the bucket D_k. Then A' and D' satisfy the invariant.*

Proof. We just repeatedly apply Claim 8. □

Claim 10. *Let A and D be configurations of buckets which satisfy the invariant. Let A_1 be empty. Let $D' = D$ and let A' be obtained from A in such a way that we move some items from an arbitrary bucket A_n to the bucket A_1. Then A' and D' satisfy the invariant.*

Proof. It is enough to realize, that this operation is inverse to the one in Claim 8. Thus if in Claim 8 it holds that for $\forall t, \sum_{i=1}^{t} d_i \geq \sum_{i=1}^{t} d'_i$, it implies that $\forall t, \sum_{i=1}^{t} a_i \leq \sum_{i=1}^{t} a'_i$ which is enough for the preservation of the invariant. □

Claim 11. *Let A and D be configurations of buckets which satisfy the invariant. Let m be the integer such that buckets $A_1, A_2, \ldots A_m$ are empty. Let $D' = D$ and let A' be obtained from A in such a way that we move some items from an arbitrary bucket A_k to the buckets $A_1, A_2, \ldots A_m$. Then A' and D' satisfy the invariant.*

Proof. We just repeatedly apply Claim 10. □

Claim 12. *Let A and D be configurations of buckets which satisfy the invariant and $|A| = |D|$.. Let S be a subset of $\{1, 2, \ldots, L\}$ such that $|A_S| \neq |A|$ and m be the smallest number such that $|A_S| < |D_{\{1\ldots m\}}|$. Let D' be obtained from D in such a way that we remove all items from $D_{\{1\ldots m-1\}}$ and $|A_S| - |D_{\{1\ldots m-1\}}|$ items from D_m. A' is obtained from A by removing all items from the set A_S. Then A' and D' satisfy the invariant.*

Proof. First, let us consider the changes of A and D. For the sake of an argument we add all empty buckets of A to S; this does not influence anything but it will be consistent with the set $D_{\{1,2,\ldots,m\}}$ where we take these buckets by definition of the uniform bucketing.

The case of D is easier. Let us recall that we remove all items from the set of buckets $\{D_1, D_2, \ldots, D_{m-1}\}$ and then $|A_S| - |D_{\{1\ldots m-1\}}|$ items from D_m

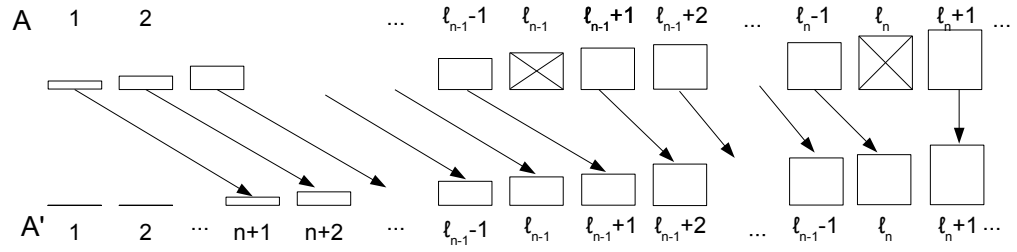


Figure 4.3: Relationship between A and A' in the proof of Claim 12. Instead of A_i we write just i in the first row and similarly for A'_i in the second row.

according to the definition of the uniform bucketing . So it directly follows, that for $t < m$, $\sum_{i=1}^{t} d'_i = 0$ and for $t \geq m$, $\sum_{i=1}^{t} d'_i = \sum_{i=1}^{t} d_i - |A_S|$.

The situation for A is more complicated. Let us denote buckets of A_S as $A_{\ell_1}, A_{\ell_2}, \ldots, A_{\ell_n}$ where we assume $\ell_1 < \ell_2 < \ldots < \ell_n$. Notice that in fact this operation "creates" A' from A by removing A_S and inserting n new empty buckets.

Now we can find the relation between n and m. We claim that n (the number of empty buckets in A') is smaller than m ($m-1$ is the number of empty buckets in D'). This obviously holds, since otherwise the size of S is greater or equal to m while $|A_S| < |D_{\{1\ldots m\}}|$ and thus the invariant could not hold for A and D and we obtain contradiction. This ensures us $m > n$.

For A and A' we can further see (Fig. 4.3) that for $j > \ell_n$, $A_j = A'_j$, for $\ell_{n-1} + 1 < j \leq \ell_n$, $A_{j-1} = A'_j$, for $\ell_{n-2} + 2 < j \leq \ell_{n-1} + 1$ is $A_{j-2} = A'_j$, etc. Generally for some $p < n$ we obtain: $\forall j, \ell_{n-p} + p < j \leq \ell_{n-p+1} + p - 1, A_{j-p} = A'_j$ Thus we get the following equations for sums. For $t > \ell_n$, $\sum_{i=1}^{t} a'_i = \sum_{i=1}^{t} a_i - |A_S|$, for $\ell_{n-1}+1 < t \leq \ell_n$ we get $\sum_{i=1}^{t} a'_i = \sum_{i=1}^{t-1} a_i - |A_S| + a_{\ell_n}$, for $\ell_{n-2}+2 < t \leq \ell_{n-1}+1$, $\sum_{i=1}^{t} a'_i = \sum_{i=1}^{t-2} a_i - |A_S| + a_{\ell_n} + a_{\ell_{n-1}}$ etc. So for some $p < n$ we obtain: $\forall j, \ell_{n-p} + p < t \leq \ell_{n-p+1} + p - 1, \sum_{i=1}^{t} a'_i = \sum_{i=1}^{t-p} a_i - |A_S| + \sum_{j=0}^{p-1} a_{\ell_{n-j}}$

Now we would like to show that for $t \geq n$, $\sum_{i=1}^{t} a'_i \geq \sum_{i=1}^{t} a_i - |A_S|$. It obviously holds that for $\ell_{n-1} + 1 < t \leq \ell_n$, $a_t \leq a_{\ell_n}$, for $\ell_{n-2} + 2 < t \leq \ell_{n-1} + 1$, $a_t \leq a_{\ell_{n-1}}$ etc. Therefore we can show that for $\ell_{n-1} + 1 < t \leq \ell_n$, $\sum_{i=1}^{t} a'_i = \sum_{i=1}^{t-1} a_i - |A_S| + a_{\ell_n} \geq \sum_{i=1}^{t-1} a_i - |A_S| + a_t = \sum_{i=1}^{t} a_i - |A_S|$, for $\ell_{n-2} + 2 < t \leq \ell_{n-1} + 1$, $\sum_{i=1}^{t} a'_i = \sum_{i=1}^{t-2} a_i - |A_S| + a_{\ell_n} + a_{\ell_{n-1}} \geq \sum_{i=1}^{t-2} a_i - |A_S| + a_t + a_{t-1} = \sum_{i=1}^{t} a_i - |A_S|$ etc. Finally we get for $t \geq n$, $\sum_{i=1}^{t} a'_i \geq \sum_{i=1}^{t} a_i - |A_S|$.

Putting it all together, we know that for $t > n$, $\sum_{i=1}^{t} a'_i = \sum_{i=1}^{t} a_i - |A_S|$

and for $t \leq n$, $\sum_{i=1}^{t} a'_i = 0$ and for $t \geq m$, $\sum_{i=1}^{t} d'_i = \sum_{i=1}^{t} d_i - |A_S|$ and for $t < m$, $\sum_{i=1}^{t} d'_i = 0$. Obviously for $t \geq \max(m, n+1)$, $\sum_{i=1}^{t} a'_i \geq \sum_{i=1}^{t} a_i - |A_S| \geq \sum_{i=1}^{t} d_i - |A_S| = \sum_{i=1}^{t} d'_i$ since the invariant holds for A and D. And since $m > n$ there will be at least same number of empty buckets in D' as in A' and thus the invariant is preserved. □

Claim 13. *Let A and D be configurations of buckets which satisfy the invariant. Let $S = \{j_1, j_2, \ldots, j_r\}$ be the subset of $\{1, 2, \ldots, L\}$ such that there does not exist m such that $|A_S| < |D_{\{1\ldots m\}}| < 2|A_S|$. Let n be the greatest number such that $|D_{\{1\ldots n\}}| \leq |A_S|$. Let A' be obtained from A by moving all items from the buckets $A_{j_1}, A_{j_2}, \ldots, A_{j_r}$ to A_{j_1}. Let D' be obtained from D by moving all items from the buckets $D_1, D_2, \ldots, D_n$ to D_n. Then A' and D' satisfy the invariant.*

Proof. Assume that $|A_S| \neq |A|$ (otherwise the proof is trivial). Then the following holds. Since there does not exist appropriate m, we know that $\sum_{i=1}^{n} d_i \leq |A_S|$ and $\sum_{i=1}^{n+1} d_i \geq 2|A_S|$ which implies that $d_{n+1} \geq |A_S|$. If we remove all items from $D_{\{1\ldots n\}}$ and also from A_S and insert them to empty buckets D'_ℓ ($d'_\ell = |D_{\{1\ldots n\}}|$) and A'_k ($a'_k = |A_S|$) respectively, then two possibilities might occur. Notice that $\ell = n$, since $|D_{\{1\ldots n\}}| \leq |A_S|$ and $|A_S| \leq d_{n+1}$.

Let us start with the case of $k < n$. First, notice that A'_k consists only from items from such bucket A_j that $j \leq k$ which implies that for $t \geq k$, $\sum_{i=1}^{t} a'_i = \sum_{i=1}^{t} a_i$. Analogously, we obtain for $t \geq n$, $\sum_{i=1}^{t} d'_i = \sum_{i=1}^{t} d_i$. In addition, from the uniform bucketing definition it holds that for $t < n$, $\sum_{i=1}^{t} d_i = 0$. Putting it all together it directly implies, that for $t \leq L$, $\sum_{i=1}^{t} d'_i \leq \sum_{i=1}^{t} a'_i$ since the invariant holds for A and D.

For the other case of $k \geq n$ the situation is a little more complicated. As previously, only such items from buckets A_j that $j < k$ can be involved in the merge and thus for $t \geq k$, $\sum_{i=1}^{t} a'_i = \sum_{i=1}^{t} a_i$ and for the similar reason for $t \geq n$, $\sum_{i=1}^{t} d'_i = \sum_{i=1}^{t} d_i$. This implies for $t \geq k$, $\sum_{i=1}^{t} a'_i \geq \sum_{i=1}^{t} d'_i$ because the invariant holds for A and D.

In addition we know that for $t < n$, $\sum_{i=1}^{t} d'_i = 0$ thus it only remains to solve such t that $n \leq t < k$.

We observe that since $|A_S| \leq d'_{n+1}$ it holds for $n < j \leq k$, $a'_j \leq d'_j$ which directly implies that for $n < t \leq k$, $\sum_{i=t}^{k} d'_i \geq \sum_{i=t}^{k} a'_i$. Let us assume that there exists such t',$n \leq t' < k$, for which the invariant does not hold for A' and D'. It means that for this t',$\sum_{i=1}^{t'} a'_i < \sum_{i=1}^{t'} d'_i$. However since we know that $\sum_{i=t'+1}^{k} d'_i \geq \sum_{i=t'+1}^{k} a'_i$ it never can do $\sum_{i=1}^{k} a'_i \geq \sum_{i=1}^{k} d'_i$, which is true. Thus we obtain a contradiction and the invariant is preserved. □

Claim 14. *Let A and D be configurations of buckets which satisfy the invariant. Let A_1 and D_1 be empty buckets. Let A' be obtained from A by inserting k items to A_1 and D' be obtained from D by inserting k items to D_1. Then A' and D' satisfy the invariant.*

Proof. Let us assume that buckets of A and D are sorted, then if A contains empty buckets one of them is surely A_1 (the very same holds for D). Let us denote changed buckets as A'_n and D'_m. Then it holds that for $j < n$, $A_{j+1} = A'_j$ and for $j > n$, $A_j = A'_j$. Analogously for $j < m$, $D_{j+1} = D'_j$ and $D_j = D'_j$ for $j > m$.

Now we can compare the sums of A' and D'. For $t < \min(m, n)$, $\sum_{i=1}^{t} a'_i = \sum_{i=1}^{t+1} a_i$ and for $t > \max(m, n)$, $\sum_{i=1}^{t} a'_i = \sum_{i=1}^{t} a_i + k$. The very same holds for D and D' and thus for $t < \min(m, n)$, $\sum_{i=1}^{t} a'_i = \sum_{i=1}^{t+1} a_i \geq \sum_{i=1}^{t+1} d_i = \sum_{i=1}^{t} d'_i$ and for $t > \max(m, n)$ it holds that $\sum_{i=1}^{t} a'_i = \sum_{i=1}^{t} a_i + k \geq \sum_{i=1}^{t} d_i + k = \sum_{i=1}^{t} d'_i$.

Now we have to look at the remaining interval from $\min(m, n)$ to $\max(m, n)$. First we assume that $n \leq m$. It means that for any j in this interval it holds that $A'_j \geq k$ and $k \geq D'_j$ (recall that $a'_n = k$ and $d'_m = k$) and it follows that for $n \leq t \leq m$, $\sum_{i=n}^{t} a'_i \geq \sum_{i=n}^{t} d'_i$. And since we know that $\sum_{i=1}^{n-1} a'_i \geq \sum_{i=1}^{n-1} d'_i$ we directly obtain for $n \leq t \leq m$, $\sum_{i=1}^{t} a'_i \geq \sum_{i=1}^{t} d'_i$.

For $n > m$ the situation is more interesting since $A'_j \leq k \leq D'_j$. Again it holds that $\sum_{i=1}^{m} a'_i \geq \sum_{i=1}^{m} d'_i$ but now for $m \leq t \leq n$, $\sum_{i=t}^{n} a'_i \geq \sum_{i=t}^{n} d'_i$. However let us assume that there is t' such that $m \leq t' \leq n$, $\sum_{i=1}^{t'} a'_i < \sum_{i=1}^{t'} d'_i$. But we also know that for $m \leq t' \leq n$, $\sum_{i=t'+1}^{n} a'_i < \sum_{i=t'+1}^{n} d'_i$ and $\sum_{i=1}^{n+1} a'_i \geq \sum_{i=1}^{n+1} d'_i$. It follows that it cannot do the last equation for such t' and thus we obtain a contradiction and the proof is finished.

□

Now we can use these claims to show the efficiency of the uniform* strategy of the original strategy. We start with the merge operations.

Lemma 5. *Let A and D be configurations of buckets which satisfy the invariant. Let S be the subset of $\{1, 2, \ldots, L\}$ and m be the greatest integer such that $|D_{\{1\ldots m\}}| < 2|A_S|$. Let the original bucketing B perform a merge on the set of buckets A_S obtaining a configuration of buckets A'. Then the uniform*(B) bucketing performs a merge on the set of buckets $D_{\{1\ldots m\}}$ (so that all items from $D_{\{1\ldots m\}}$ are moved to D_m according to definition) obtaining a configuration D'. Then A' and D' satisfy the invariant and the cost of the merge performed by the uniform*(B) bucketing is at most two times greater than $|A_S|$.*

Proof. First, let us assume that $|D_{\{1\dots m\}}| > |A_S|$. Notice, that a merge operation can be seen as three independent steps. First, we remove all items from the involved buckets, then we insert all these items to one bucket of A_S and finally, we redistribute some items from this bucket to get the same state of A' as B. Similarly we remove $|A_S|$ items from $D_{\{1\dots m\}}$ in such a way that we find the greatest n such that $|D_{\{1\dots n\}}| \leq |A_S|$. Then we remove all items from $D_{\{1\dots n\}}$ and $|A_S| - |D_{\{1\dots n\}}|$ items from D_{n+1}. Then we insert these removed items to one of the empty buckets of D (notice, that $n \geq |S|$ because of the invariant). We use Claim 12 which ensures us satisfying of the invariant for the "remove" part. Then we insert $|A_S|$ items to the empty bucket of A' and also to an empty bucket of D' using Claim 14 (notice that there will be an empty bucket, since we remove items from at least two buckets). Finally, we adjust newly obtained sequences using Claim 9 and Claim 11. For A' this means redistributing some items from the bucket to which we inserted all removed items because we cannot assume that the original bucketing always performs merge into one bucket. For D' it means taking the remaining items from $D'_{\{n\dots m-1\}}$ and move them to the bucket D_m (recall that the uniform*(B) bucketing is defined to take *all* items in the "smallest" buckets and we assume, that D' is sorted). As you can see all these operations we performed preserve the invariant.

For the other case of $|D_{\{1\dots m\}}| \leq |A_S|$ we use Claim 13 because we move all items from A_S to one bucket of A_S (let us denote this bucket A'_i) and also we move all items from $D_{\{1\dots m\}}$ to a bucket of D_m. Due to Claim 13 the invariant is preserved. Finally, we take some items from A'_i and distribute them to get the same configuration as the original bucketing. And again Claim 11 ensures us the preserving of the invariant.

So for both cases the invariant is preserved and because we can perform all these steps in one, the cost will not exceed $2|A_S|$. □

This result can be directly used for the following lemma, which shows that insert is done in such a way that also preserves invariant.

Lemma 6. *Let A and D be configurations of buckets which satisfy the invariant. Let the original bucketing B perform an insert to an arbitrary bucket A_i obtaining configuration A'. Let m be the greatest integer such that $|D_{\{1\dots m\}}| < 2|A_i|+1$. Then the uniform*($B$) bucketing performs an insert to D_1 and a merge on the set of the buckets $D_{\{1\dots m\}}$ so that all items from $D_{\{1\dots m\}}$ are moved to D_m obtaining buckets D' (again according to the definition). Then A' and D' satisfy the invariant and the cost of the insert and the merge performed by the uniform*(B) bucketing is at most $3|A_i| + 3$.*

Proof. Imagine that we have one virtual empty bucket. We insert a new item to that bucket and then we perform a merge. Since the insertion of one item to an empty bucket in A and either in D preserves the invariant (Claim 14), we can now perform a merge using Lemma 5 (we merge the virtual bucket with A_i). Thus the invariant is preserved and also the cost of the operation performed by the uniform*(B) bucketing is at most two times greater than $|A_i| + 1$. However, we cannot assume this for the uniform*(B) bucketing, since we can merge more than two buckets. But what we can do is, after inserting the new item to D_1, we can consider D_1 to be two buckets – one virtual containing the new item and one containing remaining items of D_1. Now if we assume that the new item is also inserted into the virtual bucket in A we can perform a merge. Finally, because A and D satisfy the invariant, we can infer that $d_1 \leq a_1$. Therefore the proposed abstraction is correct, because we know that D_1 will be merged and thus the overall cost of the insert and the subsequent merge is at most three times greater then $a_i + 1$. □

Now we can prove the main theorem of the section.

Proof of Theorem 3. First, we show that for an arbitrary bucketing B over the buckets A with the initial configuration C and a bucketing $B' =$uniform*(B) over the buckets B with the initial configuration C, it holds $3c(B) \geq c(B')$.

In the very beginning both bucketings start with the same initial configuration C of the buckets, therefore the invariant is satisfied. Let $C_1^A = C_1^D = C$. Let $i = 1$. Now let us make an induction step. Let O_i be the i-th operation of $B_{s,C,N}$ and C_i^A be the configuration of A just before this operation was performed. According to the definition of the uniform*(B) bucketing, there are some operation(s) O'_i in B', which corresponds to O_i. Let C_i^D be the configuration of D just before the operation(s) O'_i were performed. If we apply O_i to C_i^A we obtain C_{i+1}^A and similarly from O'_i and C_i^D we obtain C_{i+1}^D. From Lemma 5 and Lemma 6 we know, that if C_i^A and C_i^D satisfy the invariant (which is ensured by induction), then also C_{i+1}^A and C_{i+1}^D satisfy the invariant. Moreover we know, that the cost of O_i is at most three times smaller than the cost of O'_i. Thus $3c(B) \geq c(B')$.

Now let us consider an offline strategy s and s' which is uniform*(s) strategy. The strategy s' is obtained from s by replacing every bucketing B obtained as a result of s by the $B' =$uniform*(B)). However since for each B and B', it holds that $3c(B) \geq c(B')$, we can infer, that s' is at most 3 times worse than s.

□

As you can see, this proof is constructive and for every original strategy s we obtain one specific uniform strategy s'. We say that s' was obtained from s.

4.3 Prefix vs. unordered bucketing

In this section, we will show a relationship between the unordered bucketing and the prefix bucketing. This answers one of the open questions in [9]. First we precisely define the prefix bucketing.

Definition 14. *Let us have L ($L > 1$) infinitely large buckets $A_1, A_2, \ldots, A_L$ and an infinitely large set I of mutually undistinguishable items. Let us have the smallest f such that A_f is nonempty ($f = L$ if A does not contain any item). Then let us define the following operations.*

- *An* insert *operation with the parameter i ($i \leq f$) moves one arbitrary item from I into the bucket A_i. The cost of such operation is equal to the number of items in A_i after the insert was performed.*
- *A* merge *operation for the given number m moves arbitrary items among buckets $A_1, A_2, \ldots A_m$. So the parameters of the merge is the number m and the set of moves of items among those buckets. The cost of such operation is equal to $|A_{\{1,2,\ldots,m\}}|$.*

A sequence of such operations is called a prefix bucketing. *As previously, the cost of the prefix bucketing B is equal to the sum of the costs of all performed operations and we denote it as $c(B)$.*

In this section, the strategy which produces an unordered bucketing will be called *an unordered bucketing strategy* and the strategy which produces a prefix bucketing will be called *a prefix bucketing strategy*. We will also use all definitions stated in the previous section, we just replace the strategy by the prefix bucketing strategy.

Again we can make a simple observation about the lower bound of the cost of the prefix bucketing.

Observation 2. *Let B be an arbitrary prefix bucketing and N denote the number of insert operations in it. Then $c(B) \geq N$.*

Now we can state the main theorem of this section.

Theorem 4. *Let $L > 1$ be an integer and $C = 0^L$ be an empty initial bucket configuration. Let s be an optimal unordered bucketing strategy. Then there exists an online prefix bucketing strategy s' such that for every number of items N it holds that $3c(s, C, N) + N > c(s', C, N)$.*

Let us spend a while with this theorem before we start to prove it since we want to emphasize its importance. The meaning of this theorem is, that for empty initial configuration C there exists online prefix bucketing strategy that is at most constant times worse than the optimal offline(!) strategy. This is very surprising since the unordered bucketing seems to be much more powerful than the prefix bucketing. Hopefully this result will help us to obtain more interesting results in future and to use the bucketing game more widely, since it might be easier, to prove the properties of prefix bucketing and then just apply this to the unordered bucketing than prove them directly. In addition Conjecture 1 can be obtained as the corollary of this theorem and Theorem 1.

To prove Theorem 4 we will need a couple of new concepts and definitions. We define a special kind of bucketing, which never "splits" items from any bucket. This bucketing will be used in many of following claims since it turns out that this property can make our life much easier. First we limit a merge operation.

Definition 15. *Let us consider a merge operation over the set of buckets S. Then we say, that such merge is* inseparable*, if all the items from buckets A_S are moved into just one bucket from S.*

Then the definition of a new bucketing is following.

Definition 16. *Let B be a bucketing, which consists only from insert and inseparable merge operations. We say that B is* inseparable *bucketing. The strategy used for obtaining B is also called inseparable.*

Additionally, we introduce numbering of all items according to their order of insertion. Notice that due to the numbering of items it is insufficient to describe items inserted to buckets just by an n-tuple of integers (configuration). The following definition solves this problem.

Definition 17. *Let $L > 1$ be an integer and I the set of items. Let J be a subset of I. Then a partition of J into L subsets (possibly empty) is called an* assignment of L buckets*.*

It is obvious, that every distribution of items into buckets during an unordered bucketing can be described by one particular assignment. However this does not hold for a prefix bucketing, since an assignment does not store any order of partitioning.

Notation 2. *Since every subset of an assignment corresponds to a specific bucket of some bucketing, we will sometimes say "buckets" of the assignment instead of "subsets" of assignment.*

Additional notation about parts of an assignment will be introduced in the next definition.

Definition 18. *Let $L > 1$ be an integer and I be a set of items. Let C be an arbitrary assignment of L buckets of some items from I. Then each non-empty subset in C and each item in any subset in C forms so called* group.

Finally we define the terminology about the relationship between bucketings and assignments.

Definition 19. *Let B be an arbitrary bucketing. Let $\mathbb{C}$ be a set of all assignments induced by B. Let G be a set of all groups induced by the set of assignments $\mathbb{C}$. Then we say that G is obtained from B.*

With these definitions we describe a bucketing using the "groups" terminology. Let B be an arbitrary bucketing. Let us have two assignments C and C', such that C describes buckets of B after the i-th step of the bucketing B and C' after the $i+1$-th step of the bucketing B. Let $g \in C$ and $g' \in C'$ be an arbitrary groups. Then if g and g' have at least one item in common and $g \neq g'$, we say that g' is a *parent* of g and g is a *child* of g'. Notice, that g' could be obtained in two ways. The first one is that g' and g correspond to buckets which took a part in a merge operation. The second case is an insertion of a new item into g. In this section, the second case will be considered as a merge of the bucket containing items of the group g with the new item (which forms a group itself). Finally, let $g_1, g_2, \ldots g_k$ be a sequence of groups such that g_i is a parent of g_{i+1}. Then for every m, n, $m < n$ we say, that g_n is *descendant* of g_m.

Let us define the following graph based on the groups of some bucketing. This graph will describe somehow a process of bucketing and will be used in the definition of prefix bucketing based on the original one.

Definition 20. *Let B be an arbitrary bucketing and G be a set of groups obtained from B. Let us have a graph whose nodes are groups from G and its edges are defined by* parent-child *relationship between the groups of G. We denote such graph by T and we say that T is based on B or G.*

For such graph T we can state the following observation.

Observation 3. *Let B be a inseparable bucketing and T be a graph based on B. Then graph T is a forest and the number of trees in it is equal to the number of non-empty buckets after finishing the bucketing.*

The proof of this observation is simple, since the bucketing B is inseparable, every group has at most one parent and we can directly infer that T is a forest. The second part of the observation is obvious since the number of the trees is equivalent to the number of groups without a parent. And if a group does not have a parent, it was not merged and thus it corresponds to one bucket after B was finished.

Let us now introduce more terminology about groups, which will be used when describing process of bucketing. Notice that every group either was not created yet, exists in some bucket, or was merged or split to obtain a new group.

Definition 21. *Let B be an arbitrary bucketing and G a set of groups obtained from B. Let $g_1 \in G$ and $g_2 \in G$ be arbitrary groups. Then g_1 and g_2 are* independent *if they do not have any item in common.*

Definition 22. *Let $L > 1$ be an integer and C be a bucket assignment of L buckets. Then a group g is* actual in *C if $g \in C$.*

Definition 23. *Let $L > 1$ be an integer and C be a bucket assignment of L buckets. The group is* started in *C if C contains at least one item from this group. The group is* finished in *C if all its items are placed in one bucket of C.*

Observation 4. *Let G be the set of groups obtained from an inseparable bucketing B. Let $C_1, C_2, \ldots C_N$ be the sequence of assignments induced by B (sorted in order of appearance). Let g be a group finished in the assignment C_i. Then for any $j > i$ the group g is finished in C_j.*

Now we can state a simple claim about independent and dependent groups which helps us understand the graph structure.

Claim 15. *Let B be an inseparable bucketing and G be a set of groups obtained from B. Let $g_1, g_2 \in G$ be two different groups. Then g_1 and g_2 are either independent or g_1 is ancestor of g_2 or g_1 is descendant of g_2.*

Proof. Let us assume that g_1 and g_2 are not independent. Thus they have at least one item in common. If $g_1 \subset g_2$ or $g_2 \subset g_1$ we are finished so let us assume that $g_1 \not\subset g_2$ and $g_2 \not\subset g_1$. Notice that finally, after the B is finished, all actual groups are independent and since we cannot split groups, g_1 and g_2 has common ancestor.

On the other hand, let S denote the set of items which have both groups in common. But those items are descendants not only for g_1 but also for g_2. And since we know, that T based on an inseparable B is a tree, we obtain contradiction since we just find circle. □

Let us now establish the relationship between tree structures and bucket assignments. Let T be a graph based on a inseparable bucketing B, i.e. T is a forest. Let G_S be a set of nodes of T such that if a node $u \in G_S$ then all ancestors and descendants of u are not in G_S. We say that G_S is a *state of* T. First notice, that G_S consists of independent groups (recall that nodes of T are groups) – since T is based on an inseparable B, we can apply Claim 15. Thus all groups which are not independent have to be in ancestor-descendant relationship which is forbidden for G_S.

Now we can infer the following observation.

Observation 5. *Let T be a graph based on an inseparable bucketing B. Let G_S be a state of T. Then groups in G_S define an assignment of $|G_S|$ buckets.*

The proof is simple, since the groups of G_S are independent and thus every group can be assigned to one of $|G_S|$ buckets. This observation will be very important, since it enables us to describe bucketing as a sequence of the states of T.

Let us now define the ordering of the leaves of an arbitrary graph T using relations over items.

Definition 24. *Let T be a graph based on a inseparable bucketing B. Let R be an arbitrary antisymmetric, transitive and total binary relation over all items that will be inserted by B. Then we define the ordering of leaves of T according to R as follows - let a and b be arbitrary items. If aRb then we say that a is right of b and b is left of a.*

Notice that if we define relation over items according to their id's we can obtain ordering of leaves based on it. we call such ordering *id ordering*. The example of an id ordering is in Fig. 4.4. Notice that the item with the smallest id is "on the right" (Fig. 4.4).

Now we will show how to use obtained tree structure.

Definition 25. *Let T be a graph based on a inseparable bucketing B. Let G_S be the state of T and C the bucket assignment defined by G_S. Then A is the algorithm with the following properties.*

1. *Let us set $G_C = G_S$.*
2. *Until G_C contains some group with a parent we repeat these steps:*
    i) If there exists a group such that all its children are finished in C defined by G_C we perform merge on those child groups.

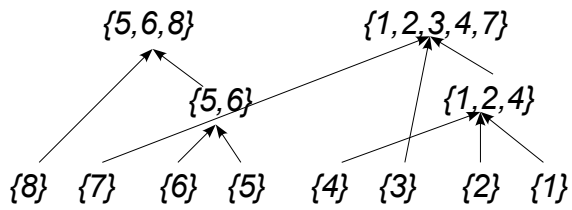


Figure 4.4: A graph T with id ordering of leaves.

ii) If there did not exist a group such that all its children are finished in C defined by G_{C}, we choose the rightmost not yet inserted item (according to the ordering of leaves of T) and we insert it. If this item should be merged with exactly one group and this group is finished in C, this item is inserted into this group directly instead of using new empty bucket.

iii) Finally we store new state into G_{C}.

Such algorithm A with some graph T forms the unordered bucketing, let us denote it B'. The next claim shows us more about its property. Notice that since we do not know the number of buckets used by B', we will assume that this bucketing needs N buckets where N is the number of inserted items – obviously this number has to be sufficient.

Claim 16. *Let $L > 1$ be an integer and $C = 0^L$ be an empty initial bucket configuration. Let B be an inseparable bucketing whose initial configuration was C and which inserts N items. Let T be a graph based on the inseparable bucketing B such that a state of T is empty (notice that the initial configuration C is also empty) and an ordering of leaves of T is arbitrary. Let B' be the bucketing formed from the algorithm A and the graph T and let $C' = 0^N$ be an empty configuration. Then $c(B') = c(B)$ and the set of non-empty bucket produced by B will be the same as the set produced by B'.*

Proof. Bucketing B' obviously generates the very same set of groups as B during its run and therefore the cost of both bucketings is equal (which follows from the construction of T, which contains all groups obtained from B and the algorithm A creates them all).

Additionally we know that a set of non-empty buckets after finishing B is equal to a set of groups without parents. But exactly those groups remain after the finishing B' because of the termination condition in A. Therefore the set of non-empty bucket produced by B will be same as the set produced by B'. □

Notice that the order of finishing the groups can be different since the sequence of bucket assignments produced by B' may be different than the sequence of bucket assignments produced by B. Thus the number of buckets used by B' can be different than by B. The solution of this problem will be given later.

The following claim shows us, that if we just change the order of creating groups according to algorithm A but the order of inserting items remains unchanged, the cost will be same and we never use more buckets.

Claim 17. *Let $L > 1$ be an integer and $C = 0^L$ be an empty initial bucket configuration. Let B be an inseparable bucketing whose initial configuration was C and which inserts N items. Let T be a graph based on the inseparable bucketing B such that a state of T is empty and T has an id ordering of the leaves. Let B' be the bucketing formed from the algorithm A and the graph T. Then $c(B') = c(B)$. In addition B' will not use more buckets than B.*

Proof. The first part of the claim follows from Claim 16, thus we can focus only on the number of used buckets.

First notice that the maximal number of buckets is used just after some insertion, since B is inseparable and thus merge operations only reduce the number of nonempty buckets. Let us look at the number of non-empty buckets when the i-th item was inserted. Both bucketings have inserted the same set of items so they can differ only by performed merges. Obviously the original bucketing $B(i)$ cannot perform more merges than $B'(i)$, because $B'(i)$ performs as many merges as possible (according to the definition of the algorithm A). In addition, from the way of creating T it follows that the set of merges (where every merge is determined by the group it produces) performed by the original strategy $B(i)$ is a subset of the merges performed by the strategy $B'(i)$. Let C_i be an assignment of buckets of the $B(i)$ and C'_i be an assignment of buckets of the $B'(i)$ after the i-th insertion (we intentionally ignore the number of buckets since it is not important now). Then the number of actual groups in C'_i cannot be greater than in C_i. And it directly implies that the number of non-empty buckets after the i-th insertion in B' is not higher than in B. □

Now we make an observation about the insertion order of items.

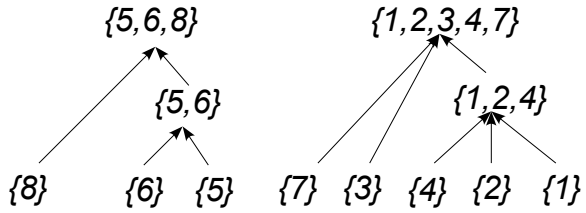


Figure 4.5: A graph T with different ordering.

Observation 6. *The order of an insertion of items can be arbitrary.*

In other words a new strategy can define a bucketing which inserts an item with a higher id first and an item with a smaller id later. This is possible since these id are defined just formally.

Now we show how to change the creation order of groups using the previous observation. If we change the ordering of leaves of some graph T while we keep the algorithm A, not only we change the insertion order, but we also change the order of the groups creation. From Claim 16, we know that if we ensure, that our new strategy does not use more buckets in any bucketing it produces than the original one, we can use it instead of the original one, because the result and the cost will be always the same.

Let us define the operator $P(group)$ as the smallest id from all items id in this *group*. Now we use this operator to define new ordering of the items. Let us just informally show new "ordering" of the leaves of T using P.

1. For each group, we sort its children according to the P so that the group for which is $P(group)$ smallest is on the right.

2. All groups without a parent are sorted as if they all have one virtual parent.

Example of such sorting is in Fig. 4.5.

Now we will introduce formally the leaves ordering which produce the very same order of leaves as we have just proposed.

Definition 26. *Let B be an inseparable bucketing and G a set of groups obtained by B. Let a and b be two items which will be inserted. Let $g_a \in G$ be the largest*

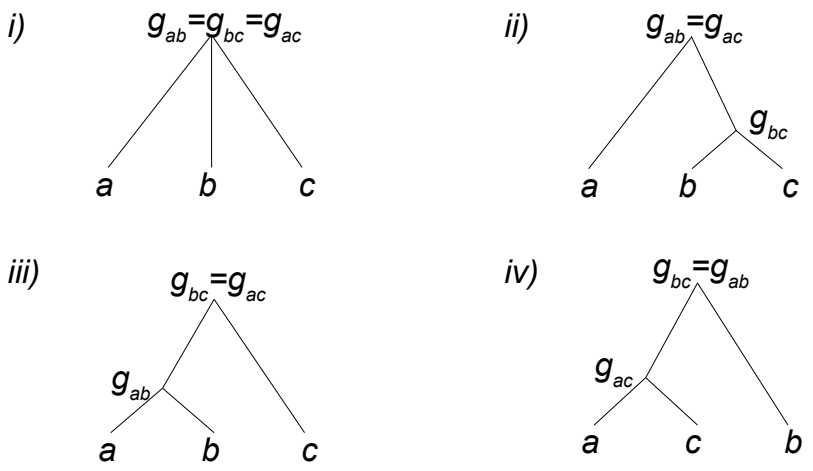


Figure 4.6: Illustration of four cases in the proof of transitivity of R_{P}.

group such that $a \in g_a$ and $b \notin g_a$. Analogously we define $g_b \in G$ as the largest group such that $b \in g_b$ and $a \notin g_b$. Then R_{P} over B is such relation over the items inserted by B that $aR_{\mathrm{P}}b$ if and only if $P(g_a) < P(g_b)$ or $a = b$. We call R_{P} a prefix relation.

Claim 18. *Let B be an inseparable bucketing and G the set of groups obtained by B. Then a prefix relation R_{P} over B is antisymmetric, transitive and total binary relation over all items that will be inserted.*

Proof. The antisymmetry of R_{P} is obvious. It is also total, because for every pair of items a and b there exist groups $g_a \in G$ and $g_b \in G$ such that $a \in g_a$ and $b \notin g_a$ and $b \in g_b$ and $a \notin g_b$ (for example groups consisted from items itself) – thus we have to choose greatest of them. However it is not obvious, whether such definition of groups is not ambiguous. Let us assume that we have two groups g_a^1 and g_a^2 such that $a \in g_a^1, g_a^2$ and $b \notin g_a^1, g_a^2$. Then from Claim 15 it follows that either g_a^1 is an ancestor of g_a^2 or g_a^2 is an ancestor of g_a^1 and thus $|g_a^1| > |g_a^2|$ or $|g_a^1| < |g_a^2|$. The very same holds for groups containing b, thus we can always unambiguously find the pair of the largest groups.

It remains to show that R_{P} is transitive. Let us have three items a, b and c such that $aR_{\mathrm{P}}b$ and $bR_{\mathrm{P}}c$. Then we want to show that $aR_{\mathrm{P}}c$. Let us assume for

simplicity, that there exists a group g_p which is a parent of those groups which do not have a real parent. Let g_{ab} be such a group that contains a and b but non of its children contains both a and b (it is the "smallest" group containing a and b). Similarly we define g_{bc} as the "smallest" group containing b and c and g_{ac} as the "smallest" group containing a and c. Then a few cases can occur (see Fig. 4.6).

i) $g_{ab} = g_{bc} = g_{ac}$. Let $g_{abc} = g_{ab}$. Let g_a be the child of g_{abc} that contains a, g_b be the child of g_{abc} that contains b and finally, g_c be the child of g_{abc} that contains c. Notice that from the definition of g_{ab} (and also g_{bc} and g_{ac}) we know that g_a is the largest group containing a but not containing b and c and similarly for g_b and g_c. Thus we can simply infer that, transitivity holds for this case, since if $P(g_a) < P(g_b)$ and $P(g_b) < P(g_c)$ then $P(g_a) < P(g_c)$.

ii) g_{ab} (or g_{ac} as well) is an ancestor of g_{bc}. Let g_a be the child of g_{ab} that contains a, g_b be the child of g_{ab} that contains b and finally g_c be the child of g_{ab} that contains c. Notice that since g_{ab} is an ancestor of g_{bc} we simply obtain, that $g_b = g_c$. Thus the result of $P(g_a) < P(g_b)$ has to be same as the result of $P(g_a) < P(g_c)$.

iii) g_{bc} (or g_{ac} as well) is an ancestor of g_{ab}. Let g_a be the child of g_{bc} that contains a, g_b be the child of g_{bc} that contains b and finally g_c be the child of g_{bc} that contains c. Notice that since g_{bc} is an ancestor of g_{ab} we simply obtain, that $g_a = g_b$. Thus the result of $P(g_a) < P(g_c)$ has to be same as the result of $P(g_b) < P(g_c)$.

iv) g_{bc} (or g_{ab} as well) is an ancestor of g_{ac}. Let g_a be the child of g_{bc} that contains a, g_b be the child of g_{bc} that contains b and finally g_c be the child of g_{bc} that contains c. Notice that since g_{bc} is an ancestor of g_{ac} we simply obtain, that $g_a = g_c$. But then we simply obtain a contradiction with an assumption since the result of $P(g_a) < P(g_b)$ has to be same as the result of $P(g_c) < P(g_b)$.

Notice that because of the fact that B is inseparable and Claim 15 we do not have to check more cases. For example for g_{ab} and g_{ac} we obtain that $g_{ab} = g_{ac}$ or we can use Claim 15 and obtain that they are in ancestor-descendant relation since they have item a in common. The very same holds for remaining pairs.

Therefore we have proved that R_{P} is antisymmetric, transitive and total binary relation. □

Claim 19. *Let B be an inseparable bucketing and G a set of groups obtained by B. Let R_{P} be a prefix relation over B. Let $g_1, g_2 \in G$ be independent groups. Then if for an arbitrary item $a \in g_1$ and an arbitrary item $b \in g_2$ holds that $aR_{\mathrm{P}}b$ then for all pairs of items $a_i \in g_1$ and $b_j \in g_2$ hold that $a_i R_{\mathrm{P}} b_j$.*

Proof. Let us assume, that there is g_p that is a parent of all groups without a parent. Then let g_{ab} be a group that contains a and b and non of its children contain both a and b. Let g_a be the child of g_{ab} containing a and g_b be the child of g_{ab} that contains b. Notice that g_1 is descendant of g_{ab} and is either equal to g_a or it is descendant of g_a. The very same holds for g_2 and g_b. Thus we can infer, that if we ask whether $aR_{\mathrm{P}}b$ we actually compare $P(g_a)$ and $P(g_b)$. But the very same groups are compared for all items from g_1 and g_2 since $g_1 \subset g_a$ and $g_2 \subset g_b$. □

Now we can use just defined prefix relation for the definition of new leaves ordering.

Definition 27. *Let T be a graph based on an inseparable bucketing B. Let R_{P} be the prefix relation over B. Then* a prefix *leaves ordering of leaves is the ordering of leaves of T according to R_{P}.*

Now we make an interesting observation, which tells us more about the ordering of items when using prefix ordering of leaves.

Observation 7. *Let T be a graph based on an inseparable bucketing B such that it has a prefix ordering of leaves. Then due to Claim 19 all leaves which belong to one group form "continuous" interval of leaves (see Fig. 4.5).*

The following definition helps us to "navigate" in the tree structure.

Definition 28. *The group g_1 is left from the group g_2 if these groups are independent and all items of g_1 are left from the items of g_2. The group g_1 is right from the group g_2 if these groups are independent and all items of g_1 are right from the items of g_2.*

Let T be a graph based on an inseparable bucketing B such that it has a prefix ordering of leaves. Now consider the bucketing B' that consists from T and the algorithm A. It is enough to prove that B' does not use too many buckets since the remaining properties follow from Claim 16.

Lemma 7. *Let $L > 1$ be an integer and $C = 0^L$ be an empty initial bucket configuration of an inseparable bucketing B. Let N be the number of items inserted by B. Let T be a graph based on the bucketing B such that it has a prefix ordering of leaves and the state of T is empty. Let B' be the bucketing obtained from T and the algorithm A. Then $c(B) = c(B')$, the maximal number of non-empty buckets among all configurations produced by B' is at most the maximal number of non-empty buckets among all configurations produced by B and a final configuration of B is equal to a final configuration of B'.*

Proof. Let C_i be the assignment of B and C'_i be the assignment of B' just after an item with id i was inserted. Notice, that the number of items in C_i can be different than in C'_i. Let m denote item with id i. From the definition of A, we know that all independent groups right of item m (recall that every item itself is also a group) are finished in C'_i. Then there can be two kinds of the actual groups in C'_i.

1. Those containing at least one item with id at most i.
2. Those containing only items with id greater than i.

First we prove, that there cannot exist an actual group of the second type. Let us assume that there exists such a group g containing only items with id greater than i. We know, that this group is independent to the group containing item m (since it cannot contain m). Let us find the largest group g_ℓ such that contain g but does not contain m. Notice that due to Observation 7 such group has to form a continuous interval in the items ordering. In other words, g_ℓ has to be obtained from g in such a way that we "extend" the borders of g over the ordered array of items. In addition it implies that g_ℓ cannot contain any item left of m. But then we can infer that $g = g_\ell$ because the only way how to extend g to g_ℓ is to use items right of m which we already did and algorithm A ensures us that g is greatest possible. On the other hand, let us denote g_m the largest group not containing g but containing m. Then we know, that $P(g_m) \leq i < P(g)$ which is contradiction since all items from g have to be left of m.

Now we can deal with the remaining groups. Let S_i denote the set of actual groups in C_i and S'_i denote the set of actual groups in C'_i. If $|S_i| \geq |S'_i|$ we are finished because the number of buckets used by B' is smaller than by B after this insertion. However if it is not the case we find such a group g_c ($g_c \in S'_i$) so that there does not exist $g \in S_i$ such that $P(g) = P(g_c)$ (notice that such a group has to exist because of pigeonhole principle). Consider such group $g_o \in S_i$ that contains an item with the id equal to $P(g_c)$. Such g_o has to exist because

$P(g_c)$ has to be at most i as we proved in the previous paragraph and thus this item was inserted in C_i. In addition we know that $P(g_c) < P(g_o)$. On the other hand g_o is started in C'_i because of g_c (the group is started when at least one of its items is inserted) which has to be descendant of g_o (Claim 15 and the fact that g_o contains item with id $P(g_c)$ but $P(g_o) \neq P(g_c)$). In addition g_o does not contain the item with id greater than i because otherwise it could not be finished in C_i.

The question is why g_o is unfinished in C'_i. And the only reason can be, that some of its descendants were not finished. Such descendants have to be placed left of the m (otherwise they will be finished). First let us assume that $m \notin g_o$. Then we simply obtain a contradiction, since the items of g_o would be split into at least two intervals by m and this is forbidden due to Observation 7.

So let us assume that m belongs to g_o. Notice, that it implies that g_o can be divided in two groups – m itself and the remaining items of g_o. This is true, because the only way, how could be m placed to g_o in B was to insert it to the bucket containing this group (recall, that m was just inserted and no merge was performed since that). Let n be one of items, which belongs to g_o and are not inserted in C'_i. Then let g_n be the largest group containing n but not m and g_m be the largest group containing m but not n. Notice that since g_o can be divided into those two groups, than $g_m = \{m\}$. But this directly implies that $P(g_m) > P(g_n)$ since m is the item with greatest id in g_o and thus we obtain a contradiction.

Thus the group g_o cannot exist and it implies that also g_c cannot exist. Therefore the configuration C'_i never contains more non-empty buckets than C_i and thus B' does not use more buckets than B.

The remaining properties follow from Claim 16.

□

This Lemma 7 ensures us that we can use the bucketing B' instead the original bucketing B and the final assignment of the buckets will be the same, the cost will be the same and B' does not need more buckets than the original bucketing. It remains to show one very important lemma.

Lemma 8. *Let $L > 1$ be an integer and $C = 0^L$ be an empty initial bucket configuration of an inseparable bucketing B. Let T be a graph based on the bucketing B such that it has a prefix ordering of leaves and the state of T is empty. Let B' be the bucketing obtained from T and the algorithm A. Then the bucketing B' is the prefix bucketing.*

Proof. First notice that every group forms a continuous interval of items (Observation 7) and also that the items of one group are (at least one step of the bucketings) in one bucket. Therefore we can split inserted items into continuous intervals where every interval corresponds to one bucket. It also defines an ordering of the buckets. Finally if we create a new group, it is obtained from the groups, that form a continuous interval.

Let us now use the left/right convention used for groups. Then every insertion is made to the leftmost non-empty bucket or its empty neighbour – which is in prefix bucketing manner.

So it only remains to solve the merge operations. Assume that we can do merge that does not contain the leftmost non-empty bucket. Then such merge should be done before we made any insertion to this bucket (according to the property of the algorithm A) and thus we obtain a contradiction. So we only have to prove that we always make merge on the continuous interval of the buckets. But it directly follows from the fact that the items of one group always form the continuous interval and we always make merge on the items from exactly one group.

Because both operation can be done in the prefix bucketing way, B' is the prefix bucketing. □

The question is, whether the inseparable property of bucketings is not too limiting. And the answer is following claim.

Claim 20. *Let B be a uniform bucketing. Then B is inseparable.*

Proof. Obviously an insert operation cannot split anything, therefore we focus on a merge operations. But a uniform bucketing (or uniform strategy respectively) performs merge only in one specific way – it takes all items from the given set of buckets and places them into one of them. Thus it never splits buckets. □

Now we can proof quite interesting lemma.

Lemma 9. *Let $L > 1$ be an integer and $C = 0^L$ be an empty initial bucket configuration and N number of items we want to insert. Let s be an unordered bucketing strategy. Then there exists an offline prefix bucketing strategy s' such that for every number of items N it holds that $3c(s, C, N) \geq c(s', C, N)$ and s' will never use more buckets than s.*

Proof. Let s_u be a uniform strategy obtained from s. Then we know that s_u is at most three times worse than s (Theorem 3). Let B_u be a bucketing obtained from s_u, C and N. Then B_u is inseparable and $c(B) \leq 3c(s, C, N)$. Thus we can

use Lemma 7 and Lemma 8 which together ensures us the existence of the prefix bucketing B', which for C and N produce same result for the same cost as B. Since we can do this construction for any N, we obtain a definition of a prefix strategy s' for C and an arbitrary N. And therefore we are finished. □

This lemma is very interesting just by itself. It says, that if we are inserting into empty buckets, then there exists an offline prefix bucketing strategy which is up to a constant factor optimal. However we want to show more. Let us look closer at Claim 6 and its proof. We discovered, that this claim holds also for prefix strategies. And thus we can prove the main theorem of the section.

Proof of Theorem 4. Let $L > 1$ be an integer and $C = 0^L$ be an empty initial bucket configuration. Let s be the optimal unordered bucketing strategy. Then there exists an optimal prefix bucketing strategy s_p (Lemma 9) such that for an arbitrary number of items N it holds that $3c(s, C, N) > c(s_p, C, N)$. And if we apply Lemma 4 then there must exist an online prefix bucketing strategy s' such that $c(s_p, C, N) + N > c(s', C, N)$. Therefore it holds $3c(s, C, N) + N > c(s', C, N)$. □

Finally notice that since we are inserting N items, it also holds that $4c(s, C, N) > c(s', C, N)$. However, the previous result can be better for those who have strategy with greater than linear cost.

Although this lemma is very powerful, we have to emphasize, that it does not give us any directions, how to obtain an optimal prefix bucketing strategy – we know only an upper bound for its cost. Thus we cannot construct s' either.

Chapter 5

Ordered array implementation

In this chapter, we describe a structure which shows, that the lower bound given in the Theorem 2 is tight. Such structure was mentioned in [7], but it needs $O(N^2)$ space. In E. Demaine's lecture notes, there is a remark, that the $O(N^2)$ space is not necessary, however he did not provide any implementation of it. Thus we try to fill this gap and we provide a structure which is inspired by the Itai et al. idea [12].

Recall that the task is to insert N items into an array in such a way that after every single insertion, the items in the array are sorted according to their value. The insert operation of the original Itai et al. structure is costly, its amortized time complexity is $O(\log^2 N)$. We tried a different point of view. Instead of time we sacrifice space. Finally, we obtain the structure, which performs insert in amortized $O(\log N)$ time and search in $O(\log N)$ time but it needs $O(N^{1+\epsilon})$ space. In addition we preserve the time complexity of the interval queries (scan).

The main idea is similar to the one used by Itai et al. Let us have an array of size S. We divide this array into chunks, each containing $\log_2 S$ slots of the array. These chunks will be the leaves of a binary tree structure built "over" that array (Fig. 5.1). For the purpose of the analysis, we will assume, that the initial size of S is 2, the size of every chunk is exactly $\lfloor \log_2 S \rfloor$ and that remaining $S \mod \lfloor \log_2 S \rfloor$ slots of the array do not form a chunk and cannot be used for insertion. We call these slots *omitted*.

Before we describe the operations over that data structure as well as the details about that structure a few definitions are necessary.

Definition 29. *Let I be the continuous interval of slots in the array, then $|I|$ denotes the length of the interval I. The number of items in the interval I is denoted by N_I. The density δ_I of the interval I is defined as $N_I/|I|$.*

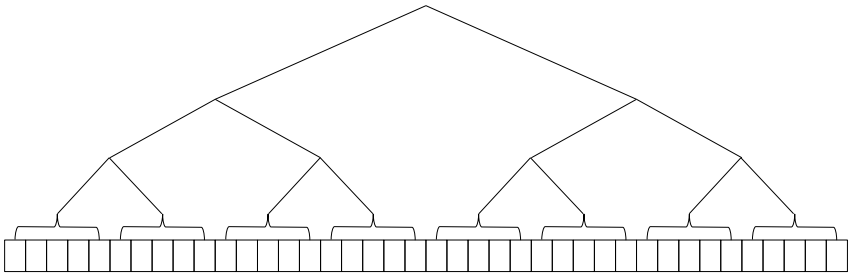


Figure 5.1: The array with the tree.

This definition of the density is similar to that definition [12].

For the next definition we have to look closer at the tree structure. We can notice that leaves of every subtree in the tree form a continuous interval. Now the remaining definitions follow.

Definition 30. *Let u be an arbitrary node in the tree. Let us denote by I_u the interval defined by the subtree with the root u (Fig. 5.2). The density of the node u is defined as the density of I_u and we denote it δ_u. For every node we define the maximal allowed density ($\max \delta$) as follows. Let α be an arbitrary number such that $1/2 < \alpha < 1$. For leaves, the maximal density is 1. Recursively we define the maximal density of every node. Let the node v be a child of the node u. Then $\max \delta_u = \alpha \max \delta_v$. Additionally every leaf has to be either empty or has its density at least $1/2$. Finally, we define the capacity C_u of the node u to be equal to $\lceil |I_u| \max \delta_u \rceil$. The capacity C of the whole array is the capacity of the root.*

Notice that the omitted slots do not belong to the interval of any node.

Now we can describe our structure over the array more precisely. In every node u, we store the minimal and the maximal item in I_u, the density δ_u and pointers to the children and to the parent of u. In every leaf we additionally store a pointer to the previous and to the next nonempty leaf.

Notice that some nodes can have their intervals empty. We omit such nodes in our structure and we do not even create them since we do not need them

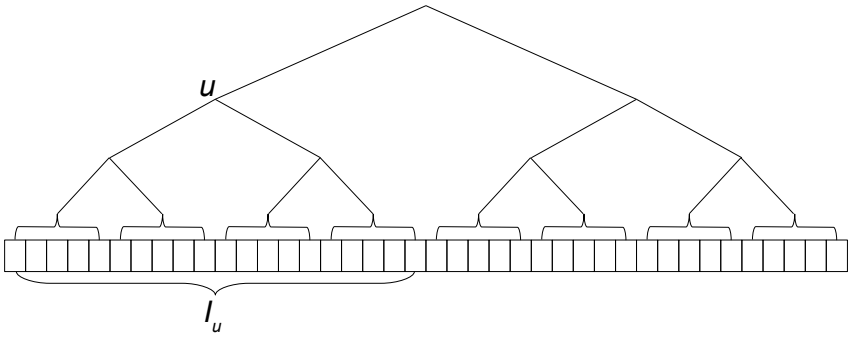


Figure 5.2: The node u and the interval I_u defined by u.

for our operations. In addition, if we were to create the nodes with an empty interval, we would affect negatively the time complexity of some operations.

Now we describe the operations with the structure.

- The *SEARCH* operation is done in straightforward manner. First we find the leaf that contains a searched item. Then we perform a binary search in the interval defined by this leaf.

- The *INTERVAL QUERY* operation is also quite simple. We find the first item of the interval and then we traverse all the items until we run across the leaf border. Then we go to its nonempty neighbour and we continue until we read the first element outside of the interval (or the end of the array).

- The *INSERT* operation is the most interesting. First we find the leaf, into which the item should be stored. If there are two possible leaves, we choose the left one. Now there are three possibilities

  1. The number of items in the whole array is equal to the capacity of the whole array. Then we enlarge the array twice and rearrange all the items (the rearrange procedure will be described later). After that we perform the insert again.

2. The density of the leaf u_ℓ is smaller than the maximal density of the leaf. Then we insert the new item into its position in the chunk (according to its value) and we move other items in the interval of the node u_ℓ to make a space for it if necessary (it is enough to do it in linear time according to the N_{u_ℓ}).
3. The leaf is *full* i.e. its density is equal to its maximal density. Then we find the first ascendant node u_a (using the pointer to the parent node in every node) such that $\delta_{u_a} < \max \delta_{u_a}$. After that we rearrange all items in I_{u_a}. Notice that such node u_a has to exist, because otherwise we would enlarge the array by case 1.

Notice that since the initial size of array is 2, this algorithm ensures us, that the size of the array S will be always a power of 2. This implies that $\log_2 S$ is always integer and thus the size of every chunk is exactly $\log_2 S$.

Rearrange operation

The *REARRANGE* operation is performed every time when we need to distribute items more evenly among the given interval. This procedure rearranges all items in the given interval so they will be distributed evenly (in some sense). It is done in the following way. Assume that we perform the rearrange operation for the interval I_u. First we delete all nodes in the subtree with the root u. Notice that since we maintain only those nodes corresponding to nonempty intervals, only nonempty nodes have to be deleted.

Then we take all items in I_u and split them into $c_g = \lfloor N_{I_u}/(0.5 \log_2 S) \rfloor$ groups of the same size (up to rounding) in such a way, that every group obtains an arbitrary but unique id and for all items a, b it holds that if $a < b$ then the id of the group containing a is smaller or equal to the id of the group containing b. Now notice that all such groups will satisfy the restrictions for the number of items in the leaf. Thus we can consider every group to be a set of items of one leaf of the tree. Having those groups-leaves which are nonempty we distribute them in the order corresponding to the group ids evenly in the leaves of u in such a way that the number of empty leaves between two neighbouring nonempty leaves is $\lfloor (\log_2 |I_u|)/c_g \rfloor$ up to 1 (the overall number of leaves divided by the number of nonempty leaves). For every nonempty leaf we update its pointers to the neighbouring nonempty leaves. Then we place items in every nonempty chunk so that the items are placed continuously from the left border of the chunk.

Finally, we "repair" the tree in the following way – we take all nonempty leaves and we create their ascendant nodes up to the node u. In other words we will be creating ascendants of those leaves until we should create the node

whose interval I would be equal to I_u and then we finish. To do that we put all nonempty leaves in the queue in the same order as they are represented in the array. Then we go through this queue and for every node in it we construct the parent of this node. We only have to ensure, that every node is not created twice (which could happened if two nodes with the same parent are in the queue). Since the nodes are stored in the same order as their intervals, it is enough to check only neighbours of the current node and if they have the same parent, create it only once. After we are finished with all nodes in the queue, we take newly obtained nodes and we continue, until the node with the interval I_u is created. It is obvious that the properties of u as well as the properties of all ascendants of u remain unchanged.

That is all to the data structure description.

The time complexity of the tree operations

Now we analyze data structure operation time complexity. We start with the following claim, which shows us the relation between the overall capacity of the array and its size.

Claim 21. *Let S denote the size of the array and C be the capacity of the array. Then it holds that $S \in O(C^{1+\epsilon})$ where $\epsilon = -(\log_2 \alpha)/(1 + \log_2 \alpha)$.*

Proof. Let us assume, that the number of items in the tree is equal to the capacity of the tree C. Let L denote the overall number of leaves of the tree, L_N denote the number of nonempty leaves and h denote the height of the tree. Since the capacity is full, the following estimate is correct

$$C \leq L_N \log_2 S$$

since every nonempty leaf cannot contain more than $\log_2 S$ items. On the other hand, since the density of the root is equal to the α^h, from the definition of capacity we obtain

$$\alpha^h L \log_2 S \leq \lceil \alpha^h L \log_2 S \rceil = C.$$

Thus we can infer

$$\alpha^h L \log_2 S \leq L_N \log_2 S$$

and

$$\alpha^h L \leq L_N.$$

Since the number of tree leaves can be estimate as $L \geq 2^{h-1}$ it implies

$$(2\alpha)^h \leq 2L_N$$

$$h \cdot \log_2(2\alpha) \leq \log_2(2L_N)$$

The height of the tree h is surely greater than $\log_2 L$, therefore

$$\log_2 L \cdot \log_2(2\alpha) \leq \log_2(2L_N)$$

and since $2\alpha > 1$ the following holds

$$\log_2 L \leq \frac{\log_2(2L_N)}{\log_2(2\alpha)}$$

$$L < (2L_N)^{\frac{1}{\log_2(2\alpha)}}$$

Let us have $\epsilon = -(\log_2 \alpha)/(1 + \log_2 \alpha)$ then we can rewrite the inequality as

$$L < (2L_N)^{1+\epsilon}$$

It remains to show the relation between C and S. Recall that L_N is the number of nonempty leaves and the capacity of the array is full. Also recall that the number of items in every leaf is at least $0.5 \log_2 S$ and at most $\log_2 S$. Thus we can infer

$$0.5 L_N \log_2 S \leq C$$

and since the number of omitted items is smaller than $\log_2 S$

$$S < L \log_2 S + \log_2 S.$$

and since $L > 1$ (which follows from the minimal size of S which is 2)

$$S < 2L \log_2 S.$$

Obviously

$$L \log_2^{1+\epsilon} S < (2L_N \log_2 S)^{1+\epsilon}$$

and thus

$$\frac{1}{2} S \log_2^{\epsilon} S < (4C)^{1+\epsilon}$$

and since $\log_2^{\epsilon} S > 1$

$$\frac{1}{2} S < (4C)^{1+\epsilon}$$

it directly implies that $S \in O(C^{1+\epsilon})$ where $\epsilon = -(\log_2 \alpha)/(1 + \log_2 \alpha)$. □

Now we know the relation between the capacity of the array (C) and the size of the array (S). The following observation shows the relation between the number of items in the array N and C.

Observation 8. *Let N be the actual number of items in the array. Then it holds that $C/2 \leq N \leq C$.*

The second inequality is obvious from the definition of the C and the definition of the insert operation. The first one can be easily inferred from the algorithm description. When we enlarge the array the capacity grows less than twice while the number of items in the array remains unaffected.

This observation implies directly the next observation.

Observation 9. *For the actual number of items in the array N it holds that $S \in O(N^{1+\epsilon})$ where $\epsilon = -(\log_2 \alpha)/(1 + \log_2 \alpha)$.*

Now we can continue in our analysis of the time complexity of the algorithm operations.

Lemma 10. *Let N be the number of items in the array. Then the search operation can be performed in $O(\log N)$ time in our structure. The interval query can be performed in $O(\log N + \ell)$ time, where ℓ is the number of items in the interval of the query.*

Proof. We can see, that finding the correct leaf costs $O(\log_2 S - \log_2 \log_2 S)$ time. Then we spend $O(\log_2 \log_2 S)$ time for one binary search in the interval of the leaf which gives $O(\log_2 S)$ time in total. Together with the time for finding the correct leaf this is $O(\log_2 S)$. From the Observation 9 we know that this is the same as $O(\log_2 N)$.

Very similar time we spent for finding the first item of the interval of the interval query. The time necessary for the traversal part of the interval query will be calculated as follows. Since items form continuous interval in the leaf interval at most constant time is spent for moving to the next item inside that leaf. The move to the next nonempty leaf costs us also constant time, since we have a pointer to the next nonempty leaf. Thus the scan through the whole interval will have $O(\ell)$ time complexity and together with the initial search we obtain $O(\log N + \ell)$ time complexity. □

We will continue with the time complexity of the rearrange procedure which is important for the insert.

Lemma 11. *Let I be the interval for which we perform the rearrange procedure. Let N_I denote the number of items in I. Then the rearrange procedure for the interval I needs $O(N_I)$ time.*

Proof. Removing all items from I will take us $O(N_I)$ time (which we can infer from the time complexity of the traversal part of the interval query). Also placing all items to their new positions can be performed in $O(N_I)$ time. Thus it only remains to solve the time complexity of the tree rebuilding.

Let S denote the current size of the array and h denote current height of the tree. The maximal number of nonempty leaves, whose interval is the subinterval of I, is denoted by L and it holds that $L \leq 2N_I/\log S$. From this number, we can infer that upper bound on the number of the tree nodes affected by this procedure is Lh. Such number of nodes can be deleted and after that created in $O(Lh)$ time. And since $h = \log S$, we obtain the overall time complexity $O(N_I)$ for the rearrange operation on the interval I. □

Now we focus on the insert operation. The first claim will solve such inserts, which do not cause the array enlargement.

Claim 22. *Let N be the number of items in the array and C the capacity of the array. If $N < C$ then the amortized cost of the inserts is $O(\log_2 N)$.*

Proof. The proof of this claim was inspired by the proof from [12]. Let us have a node u and its child v. Then according to the definition $\max \delta_u = \alpha \max \delta_v$ and $|I_v| = |I_u|/2$. When the last rearrange operation was performed in the interval I_u the density of the node v was at most $\max \delta_u$ (and the very same holds for the another child of u). Thus to invoke the next rearrange operation in the whole I_u we need to achieve the maximal density of the node v (or of the sibling of v which is symmetrical). To do that we have to insert at least $\max \delta_v I_v - \max \delta_u I_v = I_v \max \delta_v (1 - \alpha)$ items into I_v.

On the other hand the cost of the rearrange operation in I_u is $O(N_{I_u})$. Let us denote by N'_{I_u} the number of items which were in I_u when the last rearrange operation was performed. Thus each of the at least $|I_v| \max \delta_v (1 - \alpha)$ newly inserted items has to be charged at most $O(N'_{I_u}/(|I_v| \max \delta_v (1 - \alpha)))$ for the cost of the rearrangement in the node u. And since $N'_{I_u} \leq |I_u| \max \delta_v$ we obtain that $O(N'_{I_u}/(|I_v| \max \delta_v (1 - \alpha))) \leq O(1/(2 - 2\alpha))$ which is constant. However, each item belongs to $O(\log_2 S - \log_2 \log_2 S)$ intervals and it has to contribute to rearrangement of each of them. In total it contributes $O(\log_2 S)$ per insertion. Finally, we have to consider the cost for inserting the item into the leaf, which

is performed in $O(\log_2 S)$ time. Thus the amortized time complexity of every insertion is $O(\log_2 S)$, which from the Observation 9 is equal to $O(\log_2 N)$. □

It remains to calculate the time complexity of the array enlargement. We start with the following claim

Claim 23. *Let C_i be the capacity of the array after the i-th array enlargement was performed. Then $C_{i+1} = 2\alpha C_i$.*

Proof. Let S_i denote the size of the array, r_i the root of the tree and finally, h_i the height of the tree just after the i-th array enlargement was performed. First notice that $\max \delta_{r_{j+1}} = \alpha \max \delta_{r_j}$ since $h_{j+1} = h_j + 1$. Therefore we can infer $C_{i+1} = S_{i+1} \max \delta_{r_{i+1}} = 2\alpha S_i \max \delta_{r_i} = 2\alpha C_i$. □

The next claim shows us the time complexity of the array enlargement.

Claim 24. *Let us assume that arbitrary array allocations can be done in constant time. Let L_i be the number of the nonempty leaves and S_i be the size of the array just before the i-th array enlargement was performed. Then the i-th array enlargement (including the tree rebuilding) time complexity is $O(L_{i+1} \log_2 S_{i+1})$.*

Proof. Let L_i be the current number of the nonempty leaves. Then the upper bound on the overall number of the nodes in the tree is $L_i \log_2 S_i$ since $\log_2 S_i$ is the height of the tree. Thus the old tree structure can be deleted in the $O(L_i \log S_i)$ time. In addition it holds that $L_{i+1} \leq 2L_i$ since the worst case is, that all leaves were full and after the array enlargement they all contain only $0.5 \log S_{i+1}$ items. Thus the new tree will consist of $L_{i+1} \log_2 S_{i+1}$ nodes and it implies that it can be built in $O(L_{i+1} \log S_{i+1})$ time.

Except the tree rebuilding, we spent some time on the rearranging items. The number of items in the tree just after the array enlargement is at most $L_{i+1} \log S_{i+1}$ since every leaf can contain at most $\log S_{i+1}$ items. The time complexity of the array rearrangement is linear to the number of items in the array, therefore the time complexity of the array rearrangement is $O(L_{i+1} \log S_{i+1})$.

Since the allocation of the array is considered to be done in the constant time, we obtain that the overall time complexity of the array enlargement including the tree rebuilding is $O(L_{i+1} \log S_{i+1})$. □

Last claim about the array enlargement calculates the cost of the array enlargement per insertion.

Claim 25. *The amortized cost of all array enlargements per one item insertion is* $O(1)$.

Proof. From the Claim 23 we can infer that between the i-th and the $i+1$-th array enlargement was inserted αC_i items. Let us use the notation from the previous claim. Then the i-th array enlargement will cost $O(L_{i+1} \log S_{i+1})$ according to the Claim 24. We also know that $C_{i+1}/\log_2 S_{i+1} \leq L_{i+1} \leq 2C_{i+1}/\log_2 S_{i+1}$ (we can use capacity instead the number of items since the capacity is fulfilled before the array enlargement). This implies that $O(L_{i+1} \log_2 S_{i+1}) = O(C_{i+1})$. And since $C_{i+1} = 2\alpha C_i$, we know, that the number of the inserted items since the last array enlargement is equal to the cost of the current array enlargement up to constant and therefore the proof is finished. □

The next lemma just summarize the few previous claims.

Lemma 12. *Let* N *be the number of items in the array. Then amortized cost of the insertion of the new item is* $O(\log N)$.

Proof. The amortized time complexity of every insertion consists of the time for insertion itself and the time for array enlargement. The insertion itself amortized time complexity is $O(\log N)$ according to the Claim 22. The amortized cost of all array enlargements per one item insertion is $O(1)$ from the Claim 25. Thus the overall amortized cost of the item insertion is $O(\log N)$. □

We have just prove that the time complexity of the operations in our structure are the following

- The *SEARCH* operation – $O(\log N)$.
- The *INTERVAL QUERY* operation – $O(\log N + \ell)$.
- The *INSERT* operation – amortized $O(\log N)$.

Which means that if we use asymptotically more space than the original structure proposed by [12], we can do faster the insert operation while preserving the time complexity of the remaining operations. However the time complexity is not the only important aspect. For those type of structures also the number of cache misses is important. And from this point of view our structure is worse. Obviously the most important operation for this view is traversing through the items. The following lemma tells us more about it.

Lemma 13. *Let ℓ be the number of items through which we should traverse. Let S be the actual size of the array and B the size of the cache. Then the traverse operation causes $O(\ell/B + \ell/\log_2 S)$ cache misses.*

Proof. $O(\ell/\log_2 S)$ cache misses are caused by the fact, that we traverse through such a number of leaves and "jumping" from one leaf to another will usually cause the cache miss. The remaining cache misses happen when traversing through the continuous interval of items (or almost continuous), then after every $O(B)$ items we obtain the cache miss. □

In comparison to $O(\ell/B)$ cache misses obtained by the original algorithm this looks quite bad. However it was quite expectable since our structure is very sparse. It shows us a very important property of such structures – if the items are distributed evenly in such a large array, they are too sparse to gain some profit in the sense of caches. You may argue, that you can define larger leaves, which will make the influence of jumping between the leaves smaller. However, than the cost of the insert operation grows.

To sum up our observations, it does not seem to be reasonable, to solve this task by increasing the size of array, because we lose the benefits for which we propose such structures and in addition the additional memory cost is not insignificant.

Chapter 6

Future work

There still remain some open questions related to our results.

First we conjecture, that the optimal strategies are inseparable. If this is true, we directly obtain the following lemma, since the conversion of the unordered strategy into the uniform strategy could be omitted.

Conjecture 2. *Let $L > 1$ be an integer and $C = 0^L$ be an empty initial bucket configuration and N number of items we want to insert. Let s be an optimal offline unordered bucketing strategy. Then there exists an offline prefix bucketing strategy s' such that for every number of items N it holds that $c(s, C, N) = c(s', C, N)$ and s' will never use more buckets than s.*

This would be nice, since all the lower bounds on the cost of the prefix bucketing could be directly applied to the lower bound on the cost of the unordered bucketing.

However, the more important question is, whether there exists any relation between the labeling problem (or the order maintenance problem) and the unordered bucketing. We hope that nonsmooth labeling strategies are somehow related to the unordered bucketing and that the technique similar to the one used in [9] could be used to analyze them.

The last question is whether there exists a better reduction of offline strategies to online strategies or the additional cost of the online strategy is the smallest possible.

References

[1] A. Aggarwal, J. S. Vitter: The input/output complexity of sorting and related problems. Communications of the ACM, 31(9):1116-1127, September 1988.

[2] A. Andersson, T. W. Lai: Fast updating of well-balanced trees, in Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory (SWAT '90), Lecture Notes in Comput. Sci. 447, Springer-Verlag, Berlin, 1990, pp. 111–121.

[3] M. A. Bender, E. D. Demaine, M. Farach-Colton: Cache-oblivious B-trees. In Proceedings of the 41st Annual Symposium on Foundations of Computer Science, pages 399-409, Redondo Beach, California, November 2000.

[4] M. A. Bender, Z. Duan, J. Iacono, J. Wu: A locality-preserving cache-oblivious dynamic dictionary, Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, p.29-38, January 06-08, 2002, San Francisco, California.

[5] E. Demaine: Cache-Oblivious Algorithms and Data Structures in Lecture Notes from the EEF Summer School on Massive Data Sets, Lecture Notes in Computer Science, BRICS, University of Aarhus, Denmark, June 27–July 1, 2002.

[6] E. Demaine: Advanced Data Structures Lecture Notes [online], 2003, <http://courses.csail.mit.edu/6.897/spring03/scribe_notes/L14/lecture14.pdf>

[7] P. F. Dietz, D. D. Sleator: Two Algorithms for Maintaining Order in a List STOC 1987: 365-372.

[8] P. F. Dietz, J. I. Seiferas, J. Zhang: Lower Bounds for Smooth List Labeling, manuscript.

[9] P. F. Dietz, J. I. Seiferas, Ju Zhang: A Tight Lower Bound for Online Monotonic List Labeling, SIAM Journal on Discrete Mathematics, v.18 n.3, p.626-637, 2005.

[10] M. Frigo, C. E. Leiserson, H. Prokop, S. Ramachandran: Cache-oblivious algorithms. In Proceedings of the 40th Annual Symposium on Foundations of Computer Science, pages 285-297, New York, October 1999.

[11] F. C. Hennie, R. E. Stearns: Two-tape simulation of multitape Turing machines, J. Assoc. Comput. Mach., 13 (1966), pp. 533–546.

[12] A. Itai, A. G. Konheim, M. Rodeh: A Sparse Table Implementation of Priority Queues, Proceedings of the 8th Colloquium on Automata, Languages and Programming, p.417-431, July 13-17, 1981.

[13] J. Zhang: Density Control and On-Line Labeling Problems, Ph.D. Thesis, University of Rochester, New York.