The van Emde Boas Layout

Advanced Data Structures – Final Project

Jan Kohlhase

jan.kohlhase@est.fib.upc.edu

June 21, 2020

1 Introduction

A priority queue is a data structure holding a set of elements in which each item has an associated priority. An item with a high priority gets dequeued before an item with low priority. Items with equal priorities are dequeued according to their queueing order. There are several operations applicable to queues from which the most common are `insert(item, priority)`, `getMax()` and `extractMax()`. In [1] the authors introduced a layout for manipulating priority queues with an average and worst case processing time of $\mathcal{O}(log\ log\ n)$ per instruction. This is for example faster then implementing it by using a binary heap which allows manipulation of priority queues with an average processing time of $\mathcal{O}(log\ n)$ per instruction.

2 Approach

Representing a set of elements as a bitvector, insertions, deletions and lookups are realizable in $\mathcal{O}(1)$ while min, max and predecessor operations can be quite slow (worst case: $\mathcal{O}(n)$). To speed up those operations one can split up the universe of elements into n/b blocks of size b. For each block there is a summary bit describing whether or not the block is empty. These summary bits form a auxaliary bitvector of size n/b as follows.

1	0	1	0
01000100	00000000	00100010	00000000

Making use of the summary bitvector one can speed up ordered dictionary operations to $\mathcal{O}(n/b+b)$ which can be minimized to $\mathcal{O}(\sqrt{n})$ for values of n of the form $n=2^k$. In this structure each operation turns into a recursive operation on a smaller bitvector: firstly perform the operation on the summary vector, secondly perform it on the corresponding block vector. With applying this approach recursively to each of the smaller bitvectors we end up with a recursive data-structure as follows. (i) If $n < 2$ use a simple bitvector. (ii) Otherwise split the input into $\sqrt{n}$ blocks of size $\sqrt{n}$ and add a summary vector on top. For example let us consider a set of elements of size $n = 256$. Our top level structure looks as follows.

0	1	2	...	15	summary
$\uparrow$	$\uparrow$	$\uparrow$	...	$\uparrow$	$\uparrow$

With each element (including the summary) pointing to a structure one level below.

0	1	2	3	summary
$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$

Having a look on the lookup operation we can say that it makes a recursive call to a problem of size $\mathcal{O}(\sqrt{n})$ and does $\mathcal{O}(1)$ work. We have a recurrence relation $T(2) = \mathcal{O}(1)$ and $T(n) \leq T(\sqrt{n}) + \mathcal{O}(1)$ which can be substituted to $T(n) = \mathcal{O}(log\ log\ n)$, see [1]. To speed up min and max operations [1] proposed to store min and max value separately for each bitvector. This results in the following structure called the Van Emde Boas layout or Van Emde Boas tree, short vEB. The top level structure looks as follows.

0	1	2	...	15	summary	min	max
$\uparrow$	$\uparrow$	$\uparrow$	...	$\uparrow$	$\uparrow$		

With each element (including the summary) pointing to a structure one level below.

0	1	2	3	summary	min	max
$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$		

3 Algorithms

The Van Emde Boas layout supports a complete instruction repertoire executable on a set S.

`MIN`	: Compute the smallest element of S.
`MAX`	: Compute the largest element of S.
`INSERT` (j)	: $S := S \cup \{j\}$.
`DELETE` (j)	: $S := S \setminus \{j\}$.
`MEMBER` (j)	: Compute wether $j \in S$ or not.
`EXTRACT MIN`	: Delete the smallest element of S.
`EXTRACT MAX`	: Delete the largest element of S.
`PREDECESSOR` (j)	: Compute the largest element in $S < j$.
`SUCCESSOR` (j)	: Compute the smalles element in $S > j$.
`NEIGHBOUR` (j)	: Compute the neigbour of j in S.
`ALL MIN` (j)	: Delete from S all elements $\leq j$.
`ALL MAX` (j)	: Delete from S all elements $\geq j$.

The data structured was developed as a layout for a priority queue as its main target. Therefore in this work we will mainly focus on the operations: `INSERT`, `DELETE` and `MIN`.

3.1 Insertion

The logic for inserting an element j into the set of elements S works as follows. If S is empty or has just one element, update min and max appropriately and stop. Otherwise displace the min or max value if applicable $(j > max(S), j < min(S))$ and insert $j\ mod\ \sqrt{n}$ into the appropriate substructure. Finally insert $\lfloor j/\sqrt{n} \rfloor$ into the summary iff the summary is empty. This leads to an insertion time $\mathcal{O}(log\ log\ n)$, see [1].

3.2 Deletion

The logic for removing an element j from the set of elements S works as follows. If S has just one element, update min and max appropriately and stop. If j is the min or max value, replace them with the min or max value of the first or last nonempty substructure and proceed as if deleting that value. Move on by deleting $j\ mod\ \sqrt{n}$ from the elements corrosponding substructure. If the substructure then is empty, delete $\lfloor j/\sqrt{n} \rfloor$ from the summary. A deletion has a runtime of $\mathcal{O}(log\ log\ n)$, see [1].

3.3 Minimum element

For implementing a priority queue we need to find the smallest element of S and delete it afterwards. Since the min and max values are stored within the data-structure looking up the min value runs in $\mathcal{O}(1)$. The deletion process for the min value is then realized as described above.

4 Experiments

The experiments are designed regarding the time efficiency towards the application of a Van Emde Boas tree as a priority queue. We will report on steps needed for inserting a new element into and extracting the minimum value consisting of looking up the min value and deleting it afterwards. As a basline we will see a comparison of the Van Emde Boas tree versus a implementation of a priority queue using a binary heap structure. The source code of the implementations is available for authenticated users at the FIBs internal GitLab[1].

4.1 Assumptions

To define the data structure we need to make a restriction on its size. Let h be an arbitrary positive integer, $k = 2^h$ then the size needs to be of the form $n = 2^k$. Therefore we will define experiments for structures of size $h \in \{1, 2, 3, 4\}$.

4.2 Insertion

For testing the insertion efficiency we counted the number of operations required to insert a value. The following table reports on minimum, maximum and average number of operations required to insert n elements into a Van Emde Boas tree of size n. Therefore an array with n values $\{0, 1, ..., n-1\}$ was created, shuffled and inserted into the data structure.

n	Min	Max	Avg
4	1	3	2.250000
16	1	5	3.625000
256	1	7	4.839844
65536	1	9	5.858673

With the same experimental setup the minimum, maximum and average number of operations required to insert n elements into a Heap of size n is shown in the following table.

[1] `https://gitlab.fib.upc.edu/jan.kohlhase/data-structures`

n	Min	Max	Avg
4	1	2	1.500000
16	1	4	2.062500
256	1	8	2.511719
65536	1	15	1.964828

We can see that for small number of elements n the heap structure needs less operations for the insertions. In average case the heap seems to be faster still for bigger values of n but in worst case the Van Emde Boas tree needs less operations for the insertion. The advantages a heap has for small n comes from the overhead the Van Emde Boas layout brings with itself for maintaining the summary vectors as well as the minimum and maximum values.

The figure 1 shows a comparison plot of the number of operations in the worst case for the both structures. The theory says that for a heap the insertion should be in $\mathcal{O}(log$

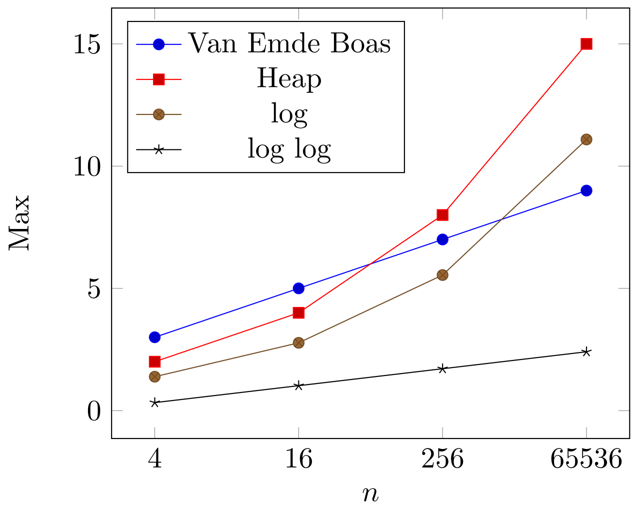

Figure 1: Insertion plot. The plot compares the number of operations needed in worst case for inserting values into a Van Emde Boas tree versus a Heap.

$n)$ while for the Van Emde Boas layout it should be in $\mathcal{O}(log\ log\ n)$. In the plot we can see those values for the different values of n. Even that the values of $log\ n$ and $log\ log\ n$ are lower than the experimental values, the plot shows a similar trend for them.

4.3 Extraction

Another experiment is regarding the deletion time, or rather finding the min/max value and deleting it afterwards as it would be needed in the application of priority queues.

The following table reports on the minimum, maximum and average number of operations needed to delete a value from the Van Emde Boas tree constructed in the former experiment. As finding the minimum in a Van Emde Boas tree is just a lookup, the reported numbers concern the operations needed for deleting one after another the values $\{0, 1, ..., n-1\}$.

n	Min	Max	Avg
4	1	3	2.250000
16	1	5	3.625000
256	1	7	4.839844
65536	1	9	5.858673

This numbers equal the values obtained in the experiments for insertion. For the Heap the reported numbers include the operations needed to find the minimum value, delete it and rearrange the heap structure.

n	Min	Max	Avg
4	2	3	2.500000
16	2	5	3.812500
256	2	9	7.429688
65536	2	17	15.276367

We can see that the number of operations needed for the extraction are slightly higher than for the insertion. For the deletion the Van Emde Boas structure has similar worst case and average numbers of operations for small n, while showing smaller values for large n. For deletion also in average the Van Emde Boas structure is faster. This originates from heapifying the array after extracting the min value in case of the heap structure.

The figure 2 shows a comparison plot of the number of operations needed in worst case for the both structures. One can see that for $n \in \{4, 16\}$ both structures show identical number of operations in worst case, while drifting apart afterwards. Again the values of *log n* and *log log n* are lower than the experimental values, but a similar trend is visible.

5 Conclusion

This work showed some properties of the Van Emde Boas layout, an extremly fast data structure for ordered dictionaries proposed in [1]. The experiments showed that it outperforms a heap structure both for insertion and extraction operations. Also not

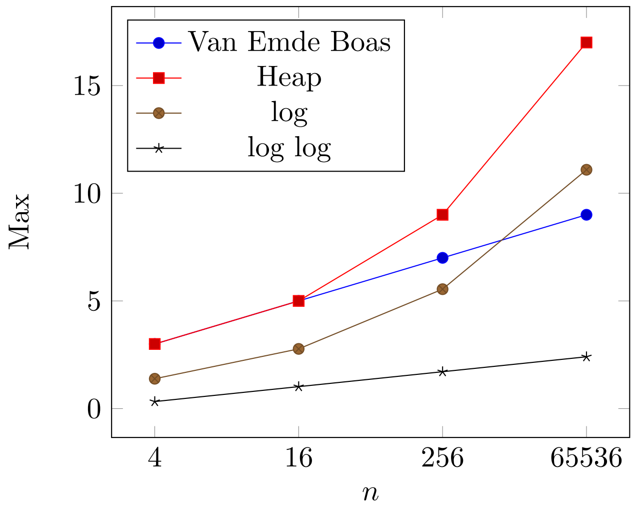

Figure 2: Extraction plot. The plot compares the number of operations needed in worst case for extracting the minimum value from a Van Emde Boas tree versus a Heap.

limited by it the experiments support a *log log* growth for insertion and extraction operations as stated in [1].

On the other hand this layout comes with some limitations. First of all it is only defined for sizes of $n = 2^k$ as seen in section 2. We also saw that for small values of n the overhead associated with maintaining the tree is higher than the performance benefits it brings compared to other data structures for ordered dictionaries. Furthermore on of the main drawbacks of the Van Emde Boas layout is its space usage. It has to be constructed over its whole universe $(1 - n)$ before being able to perform any operations on it. Its space usage is proportional to the size of the universe and not the number of elements stored in it as usually.

References

[1] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, pages 75–84. IEEE, 1975.