Binary Heaps, Fibonacci Heaps and Dijkstras shortest path

Kristoffer Just Andersen, 20051234

Troels Leth Jensen, 20051234

Morten Krogh-Jespersen, 20022362

Project 1, Advanced Data Structures 2013, Computer Science
November 2013
Advisor: Gerth StÃ¸lting Brodal

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Contents

1 Introduction **2**

2 Bit-vector **3**

3 Van Emde Boas Tree **5**
3.1 Finding minimum or maximum key 6
3.2 Finding a member 6
3.3 Finding a successor or a predecessor 6
3.4 Insert 7
3.5 Delete 7

4 Combining vEB tree and bit vectors **8**

5 Conlusion **9**

Bibliography **9**

Chapter 1

Introduction

needs content

Chapter 2

Bit-vector

A bit-vector or a bit array is a datastructure that supports INSERT, DELETE and MEMBER in constant time [**?**, p. 532]. The bit-vector consists of a set of bits, for which the $i'th$ bit indicates the presence of the $i'th$ key. Thus, in order to support queries over an universe of size u we must have a bit-vector with u bits.

To INSERT some key i into the bit-vector the $i'th$ bit is set to true, similarly to DELETE the same key i, the $i'th$ bit is cleared. The MEMBER test for key i is done by checking the $i'th$ bit. All these operations are supported easily through bit shifts [1]. Please note, that these bit operations assume that the universe size does not exceed the word size. If that is not the case, extra computation is needed to first find the correct block; this however is also constant time.

The bit-vector performs badly for MINIMUM, MAXIMUM, PREDECESSOR and SUCCESSOR because the entire array could be traversed in search for the answer.

Operation	Bit-Vector
MEMBER	$\Theta(1)$
MINIMUM	$\Theta(u)$
MAXIMUM	$\Theta(u)$
PREDECESSOR	$\Theta(u)$
SUCCESSOR	$\Theta(u)$
INSERT	$\Theta(1)$
DELETE	$\Theta(1)$

Alternatively, MINIMUM and MAXIMUM can be maintained during all changes to the bit vector, i.e. during insert and delete. With this approach the running time DELETE would in worst case be completly dominated by the linear

[1]

$$\text{TEST\_BIT}(var, pos) = ((var) \quad \& = \quad (1 << (pos)))$$
$$\text{SET\_BIT}(var, pos) = ((var) \quad | = \quad (1 << (pos)))$$
$$\text{CLEAR\_BIT}(var, pos) = ((var) \quad \& = \quad \sim(1 << (pos)))$$

search for a new MINIMUM or MAXIMUM while querying for the MINIMUM or MAXIMUM would become constant time operations.

Chapter 3

Van Emde Boas Tree

A van Emde Boas tree is a recursive data structure that supports finding the predecessor and successor in $\Theta(\log\log m)$ time where m is the size of the universe, in other words, the amount of distinct keys with a total ordering that the tree support [**?**, p. 545] [**?**]. Because van Emde Boas Trees allow universe sizes of any power of 2, we denote $\sqrt[\uparrow]{u} = 2^{\lceil(\log u)/2\rceil}$ and $\sqrt[\downarrow]{u} = 2^{\lfloor(\log u)/2\rfloor}$. Each level of the van Emde Boas Tree has a universe u and it contains $\sqrt[\uparrow]{u}$ clusters/bottom van Emde Boas trees of universe size $\sqrt[\downarrow]{u}$ and one auxilary van Emde Boas Tree of size $\sqrt[\uparrow]{u}$ we denote top. For each recursion the universe shrinks by $\sqrt{u}$.

For each tree we have two attributes that store the minimum and the maximum key. The minimum key cannot be found in any of the bottom trees, neither can the maximum unless it differs from the minimum, which happens if there are more than one element in the tree. The attributes helps reduce the number of recursive calls, because one can in constant time decide if a value lies within the range, without the need to recurse.

Van Emde Boas Trees utilizes that keys are represented as bits and if we view a key x as a $\log u$-bit binary integer, we can divide the bits up in a most significant and a least significant part. The most significant part of the bit vector identifies the cluster $\lfloor x/\sqrt{u}\rfloor$ where x will apear in position $x \mod \sqrt{u}$ in the beforementioned bottom tree. Let us denote $\mathsf{high}(x) = \lfloor x/\sqrt{u}\rfloor$ and $\mathsf{low}(x) = x \mod \sqrt{u}$ then we get the identity $x = \mathsf{high}(x)\sqrt{u} + \mathsf{low}(x)$.

Below is the time-complexities of each operation listed for the corresponding data structure and let u denote the size of the universe and n the number of keys stored in the structure:

Operation	Bit-Vector	Red-Black Tree	van Emde Boas Tree
Member	$\Theta(1)$	$\Theta(\log n)$	$O(\log\log u)$
Minimum	$\Theta(u)$	$\Theta(\log n)$	$O(1)$
Maximum	$\Theta(u)$	$\Theta(\log n)$	$O(1)$
Predecessor	$\Theta(u)$	$\Theta(\log n)$	$O(\log\log u)$
Successor	$\Theta(u)$	$\Theta(\log n)$	$O(\log\log u)$
Insert	$\Theta(1)$	$\Theta(\log n)$	$O(\log\log u)$
Delete	$\Theta(1)$	$\Theta(\log n)$	$O(\log\log u)$

3.1 Finding minimum or maximum key

This is a constant operation, since the van Emde Boas Tree directly stores the minimum and maximum element.

3.2 Finding a member

Finding out if a key is stored in a van Emde Boas Tree is simple. Either the key is the minimum or the maximum element or else recurse until we find the element. It is easy to figure out which tree to recurse on, since this is the $\mathsf{high}(x)$-th tree, and we just have to search in the smaller tree for $\mathsf{low}(x)$.

So how long does the search take in worst-case? The data structure can be described by the following recurrence function:

$$T(u) \leq T(\sqrt[\updownarrow]{u}) + \mathrm{O}\,(1)$$

If we let $m = \log u$ and realize that $\lceil m/2 \rceil \leq 2m/3$ for $m \geq 2$, which is the leaf size of the van Emde Boas Tree, we get:

$$\begin{aligned} T(u) &\leq T(\sqrt[\updownarrow]{u}) + \mathrm{O}\,(1) \\ &\Downarrow \\ T(2^m) &\leq T(2^{\lceil m/2 \rceil}) + \mathrm{O}\,(1) \\ &\Downarrow \\ T(2^m) &\leq T(2^{2m/3}) + \mathrm{O}\,(1) \\ &\Downarrow \\ S(m) &\leq S(2m/3) + \mathrm{O}\,(1) \end{aligned}$$

By the master thorem [**?**, p. 93], has the solution $S(m) = \mathrm{O}\,(\log m)$. Because $T(u) = T(2^m)$ we get $\mathrm{O}\,(\log m) = \mathrm{O}\,(\log \log u)$. Therefore the procedure takes $\mathrm{O}\,(\log \log u)$.

3.3 Finding a successor or a predecessor

As with finding a member, for the base case a successor or predecessor can be found in constant time. If not, we have to determine where to find the next element we are searching for. Finding successor and predecessor is completely analog thus we only describe finding the successor for an element x.

First, we check if the sucessor is in the bottom tree at index $\mathsf{high}(x)$. This check ban be done in constant time since it is a matter of checking maximum for that tree. If maximum exist and it is higher than $\mathsf{low}(x)$, we know we have to search inside that particular tree. This will take $\mathrm{O}\,(\log \log u)$ time.

If maximum does not exist or it is less than $\mathsf{low}(x)$ we use the top tree to search for the successor to $\mathsf{high}(x)$. If we can find such an element it will

gives us an index to a bottom tree b. Hereafter it is a constant lookup to find $\textsc{Minimum}(b)$. This case also runs in $\mathrm{O}(\log\log u)$ time since we have on search in top and a constant lookup. Therefore, the total running time of successor and predecessor is $\mathrm{O}(\log\log u)$.

3.4 Insert

Insert is pretty simple. One of the following can happen:

- The list is empty, which can be discovered in constant time by checking the minimum attribute. If that is the case, insert just set min and max.
- If not in the base case, find out if the bottom of $\mathsf{high}(x)$ is empty. If that is the case, we can just set min and max as above and update the top structure. If not, we just call insert on the bottom tree for $\mathsf{low}(x)$.

Always remember to set max or swap with min (because min was not in the tree, but now has to be since it would no longer be min). Either, the insert function recurses on the top tree of size $\sqrt[\uparrow]{u}$ or on one of the bottom trees of size $\sqrt[\downarrow]{u}$ but not on both. Therefore, the running time is at most $\mathrm{O}(\log\log u)$.

3.5 Delete

If there is only one element or universe size is 2 is easy to perform delete in constant time. Otherwise, some more work has to be done. Consider what should be done if we try to delete the minimum element. Since min is not stored in the tree, we have to find the minimum element in the bottom trees and make sure we delete it so that it can be placed as the new min. Of course, it can happen that the the bottom tree becomes empty so the top structure has to be updated. It might also be that we remove max but updating max is a constant operation.

In the description above we actually could make two recursive calls; one to update the bottom tree and possibly one to update the top. But if we update the top then the bottom tree would only contain one element. But if that is the case the first recursive function takes constant time. Therefore delete runs in $\mathrm{O}(\log\log n)$.

Chapter 4

Combining vEB tree and bit vectors

As mentioned in TODO, when u becomes suffitiantly small it can be benifial to switch to a bit-vector or red black tree. This is possible because of the recursive nature of vEB trees. To decide membership of elements over some universe it is done by deciding it recursively over smaller universes until some minimum size is reached, thus it is no problem to switch to a different datastructure that enables the same queries as vEB trees for universes small enough.

We have implemented as vEB tree that allows switching to bit-vectors when the universe falls below some constant, in our testing we will examine what the value of this constant means for the performance of our implementation.

With some constant in mind, we can conside the total universe $U_{total}(0....2^{24}-1 for this hand in)$ as the product of two different universes U_{vEB} and U_{bit}, respectivly the universes decided exclusively by the vEB tree and bit vectors. Now the vEB tree that uses bit vectors for leafs doesn't strictly speak decide membership of the elements, but they decide which bit-vector would contain the element if it is present.

Operation	Combi-tree
Member	$O(\log\log u_{veb})$
Minimum	$O(1)$
Maximum	$O(1)$
Predecessor	$O(u_{bit} + \log\log u_{veb})$
Successor	$O(u_{bit} + \log\log u_{veb})$
Insert	$O(\log\log u_{veb})$
Delete	$O(\log\log u_{veb})$

Chapter 5

Conlusion

needs contents