Remark 1. *This handout contains some notes from the Advanced Data Structures Course at MIT taught by Erik Demaine and Oren Weimann. I will intersperse questions and problems throughout the text. My comments and clarifications will always appear as* remarks *to help distinguish them from the course notes.*

Remark 2. *In the last handout, you solved several challenging technical problems. This week, we will diverge somewhat from problems that push us technically to readings that push us technically. In this sense, our goal is comprehension and the questions and the problems will expand the ideas given in the text, rather than develop new or different ideas. The readings will be technical, so you should be prepared to discuss the key ideas during our tutorial session.*

Question 1. *Last week we studied* treaps*, a randomized alternative to binary search trees, and a data structure that efficiently solves the predecessor problem under the comparison model. This week we consider again the question of computing models. We know a Turing machine is a good model for studying macro-level complexity since it can emulate most other reasonable models of computation up to a polynomial factor. However, when we talk about polynomial differences in algorithms, the Turing machine model completely breaks down. Why is this?*

6.851: Advanced Data Structures Spring 2007

Lecture 12 — March 21, 2007

Oren Weimann *Scribe: Tural Badirkhanli*

1 Overview

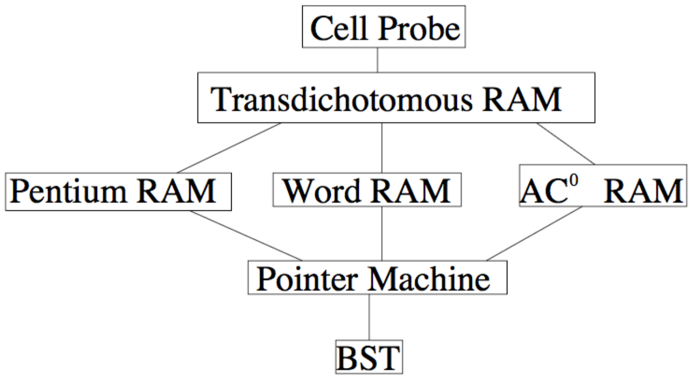
Today we start the topic of integer data structures. We first specify the model of computation we are going to use and then beat the usual $O(\lg n)$ bounds for *insert/delete* and *successor/predecessor* queries using van Emde Boas and y-fast trees. We assume that the elements – inputs, outputs, memory cells – are all w bit integers where w is the word size. We also assume a fixed size universe $\mathcal{U} = \{0, 1, \ldots, u-1\}$ where $u = 2^w$.

Remark 3. *The word size on most computing platforms is either 32 or 64 bits.*

2 Models of Computation

Cell Probe Model: This is the strongest model, where we only pay for accessing memory (reads or writes) and any additional computation is free. Memory cells have some size w, which is a parameter of the model. The model is non-uniform, and allows memory reads or writes to depend arbitrarily on past cell probes. Though not realistic to implement, it is good for proving lower bounds.

Remark 4. *Uniform models of computation essentially specify an algorithm for all input sizes, where as non-uniform models of computation give potentially different algorithms for different input sizes. Think about the differences between Turing machines and circuits. The Turing machine takes*



any input size. A circuit is bounded by its input gates, which is why people often talk about a family of circuits $C = (C_1, \ldots, C_n)$.

Question 2. *Why is the cell probe model good for proving lower bounds?*

Transdichotomous RAM Model: This model tries to model a realistic computer. We assume $w \geq \lg n$; this means that the "computer" changes with the problem size. However, this is actually a very realistic assumption: we always assume words are large enough to store pointers and indices into the data, since otherwise we cannot even address the input. Also, the algorithm can only use a finite set of operations to manipulate data. Each operation can only manipulate $O(1)$ cells at a time. Cells can be addresses arbitrarily; "RAM" stands for Random Access Machine, which differentiates the model from classic but uninteresting computation models such as a tape-based Turing Machine.

Remark 5. *Fredman and Willard coined the term* transdichotomous RAM *"because the dichotomy between the machine model and the problem size is crossed in a reasonable way."*

Depending on which operations we allow, there are several instantiations of the *Transdichotomous RAM Model*:

- *Word RAM:* In this model we have *Transdichotomous RAM Model* $+O(1)$ "C-style" operations like `+ - * / \% & | ^ ~ << >>`. This is the model we are going to use today.
- *AC^0 RAM:* The operations in this model must have an implementation by a constant-depth, unbounded fan-in, polynomial-size (in w) circuit. Practically, it allows all the operations of word RAM except for multiplication in constant time.
- *Pentium RAM* – While interesting practically, this model is of little theoretical interest as it tends to change over time.

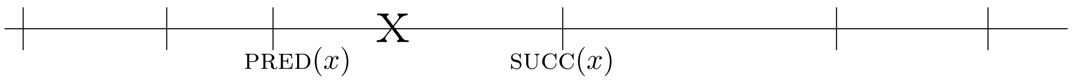
Question 3. *Defend the choice of a Word RAM model as a viable model for algorithm design and analysis.*

Pointer Machine Model: In this model the data structure is described by a directed graph with constant branching factor. For the fixed universe there is a pointer to each element in the universe $\mathcal{U}$. The input to an operation is just one of these pointers.

3 Successor/Predecessor Problem

The goal is to maintain a set S of n items from an ordered universe $\mathcal{U}$ of size u. The elements are integers that fit in a machine word, that is, $u = 2^w$. Our data structure must support the following operations:

- $\textsc{insert}(x, S), \quad x \in \mathcal{U}$
- $\textsc{delete}(x, S), \quad x \in S$ and $\quad x \in S$
- $\textsc{successor}(x), \quad x \in \mathcal{U}$
- $\textsc{predecessor}(x), \quad x \in \mathcal{U}$



As you can see, we have some universe $\mathcal{U}$ (the entire line) and some set S of points. It is important to note that the PREDECESSOR and SUCCESSOR functions can take any element in $\mathcal{U}$, not just elements from S. For our model of computation we use the *Word RAM Model* described above.

3.1 Classic Results

	data structure	time/op	space	model
1962	balanced trees	$O(\lg n)$	$O(n)$	BST
1975	van Emde Boas [1]	$O(\lg w) = O(\lg \lg u)$	$O(u)$	word / AC^0 RAM
1984	y-fast trees [2]	$O(\lg w)$ w.h.p.	$O(n)$	word RAM
1993	fusion trees [3]	$O(\lg_w n) = O\left(\frac{\lg n}{\lg \lg u}\right)$	$O(n)$	word RAM; also AC^0 RAM [4]

These are the classic solutions to the *successor/predecessor* problems up to about 1993. Today we present van Emde Boas and y-fast trees. In the following lectures, we will discuss more recent results not mentioned in this table.

3.2 Combination of classic results

Fusion trees work well when the size of the universe is much bigger than the size of the data or $\lg \lg u \geq \sqrt{\lg n}$, while van Emde Boas and y-fast trees work well when the size of the universe is much closer to the size of the data, or $\lg \lg u \leq \sqrt{\lg n}$. We can therefore choose which data structure to use depending on the application. They are equivalent around when $\Theta(\lg w) = \Theta\left(\frac{\lg n}{\lg w}\right)$, so that is the worst case in which we would use either data structure. In this worst case, we notice that $\lg w = \Theta(\sqrt{\lg n})$, so we can always achieve $O(\sqrt{\lg n})$, which is significantly better than the simpler BST model.

Remark 6. *At this point, you should read the very detailed description and analysis of van Emde Boas trees given by Erik Demaine in the 2003 version of his Advanced Data Structures course. We will call these the* vEB notes. *The vEB notes are available on the course website. The following problems and questions are all with respect to the vEB notes.*

Problem 1. *Given some element $x \in \mathcal{U}$ composed of $w = \lg u$ bits, we define high(x) to be the high-order half of the bits representing x and low(x) to be the low-order half of the bits representing x. Mathematically, these were defined as:*

- $high(x) = \lfloor x/\sqrt{u} \rfloor$
- $low(x) = x \mod \sqrt{u}$

Argue why these mathematical definitions make sense.

Problem 2. *In computer science, we like to think of the logarithm computationally as successive division by 2. Thus, $\lg n$ is the number of times you can successively divide before falling below 1. Under this model, how should we think of $\lg \lg n$?*

4 Theory versus Practice

We will again take a look at theory versus practice. Begin by reading the following paper which is available for download off the course website:

Degermark, M., Brodnik, A., Carlsson, S., and Pink, S. 1997. Small forwarding tables for fast routing lookups. SIGCOMM Comput. Commun. Rev. 27, 4 (Oct. 1997), 3-14.

Question 4. *Write a short review of the* Degermark et al. *paper. Make sure to address the importance of the engineering of the van Emde Boas structure.*

Last week we implemented a treap which has $O(\log n)$ insert, delete, predecessor, successor, and membership test operations. This week, I would like you to implement a van Emde Boas tree and again do some comparisons with Java's `HashSet`. Here are a few notes:

- When working in the Word RAM model, we assume that both arithmetic operations on integers and direct addressing (i.e. array lookups) run in $O(1)$ time. Be careful not to break this assumption by performing operations on the integer which may take more time (for example, performing $w/2$ shifts where w is the word size immediately breaks the stated $O(\lg w)$ goal).
- The van Emde Boas tree has a recursive structure, so your class will contain an array of smaller van Emde Boas trees (the substructures) as well as a single summary van Emde Boas tree. You'll also need to record the minimum and maximum element as well as the universe size. You may prefer to store the *width* of the universe (*i.e.* the number of bits necessary to represent the universe) since it will make computing *high* and *low* easier.
- Make sure that when your universe is an odd power of two that your *high* and *low* functions work appropriately.

Problem 3. *Implement a van Emde Boas tree in Java. Call your class* `VEBTree` *and make sure that it accepts an integer argument in the constructor that determines the size of the universe (i.e. the largest value allowable in your set). Provide methods for* `add`*,* `ceiling`*,* `remove`*,* `floor`*, and* `contains`*. No need to make the class generic since van Emde Boas trees operate exclusively on integers. Your* `ceiling` *and* `floor` *functions should be exclusive so that* `ceiling` *always returns the smallest number strictly larger than the query and* `floor` *always returns the largest number strictly smaller than the query. Now, run an experiment comparing the performance of your* `VEBTree` *with your* `Treap`*. [If you did not manage to get your implementation of Treap to support the operations needed for this exercise, then compare your* `VEBTree` *to Java's* `TreeSet` *structure.] Create a* `VEBTree` *with a universe size of* `Integer.MAX_VALUE / 64`*. Perform 5-10 million integer inserts (randomly pre-compute a sequence of 5-10 million unique integers). Record the time to insert 1000 integers at a time (then divide by 1000) to avoid timing issues. Do the same experiment on your treap. Now plot insertion time as a function of the size of your data structure but make the size dimension a log plot (i.e. the horizontal-axis is* $\log(n)$ *instead of* n*). Note that the van Emde Boas tree has a fixed size universe, so its running time should be a line with no slope. Let me know if you encounter memory problems.*

References

[1] P. van Emde Boas, *Preserving Order in a Forest in less than Logarithmic Time*, FOCS, 75-84, 1975.

[2] Dan E. Willard, *Log-Logarithmic Worst-Case Range Queries are Possible in Space* $\Theta(n)$, Inf. Process. Lett. 17(2): 81-84 (1983)

[3] M. Fredman, D. E. Willard, *Surpassing the Information Theoretic Bound with Fusion Trees*, J. Comput. Syst. Sci, 47(3):424-436, 1993.

[4] A. Andersson, P. B. Miltersen, M. Thorup, *Fusion Trees can be Implemented with* AC^0 *Instructions Only*, Theor. Comput. Sci, 215(1-2): 337-344, 1999.