6.897: Advanced Data Structures Spring 2005

Lecture 10 — March 8, 2005

Prof. Erik Demaine *Scribe: Daniel Kane*

1 Introduction

In this lecture, we discuss the implementation of the fusion tree data structure [2], as part of our discussion of the predecessor problem. Given a static set, fusion trees can answer predecessor or successor queries in $O(\log_w n)$ time. In the dynamic case, both updates and queries can be supported in time $O(\lg_w n + \lg w)$. Using exponential trees [1] (see also the problem set), this can be reduced to $O(\lg_w n + \lg \lg n)$ time per operation.

2 Operations on Words

There are several basic operations on words that we would like to perform in constant time, and below are implementations for the word RAM (usually referred to as "bit tricks").

Masking: given a set of bit positions $p_1, p_2, \ldots, p_k$ we wish to take a word $x = \sum_{i=0}^{w-1} x_i 2^i$ to $\sum_{i=1}^{k} x_{p_i} 2^{p_i}$, thus replacing all bits of x not in a position p_i with a 0. This can be done in a single operation by taking a bitwise AND of x with $\sum_{i=1}^{k} 2^{p_i}$.

Least/most significant set bit: Given a word $x = \sum_{i=1}^{k} 2^{p_i}$ with p_i's distinct, we wish to compute $\max_i p_i$ (alternatively $\min_i p_i$). This gives the index of the most/least significant bit of x that is set. It is easy to give an AC^0 implementation of this operation. One can also given an implementation on the word RAM using multiplication (see [2]). This is quite complicated (it requires around 60 operations), and we will not discuss it.

Word-packed vectors: The true power of the transdichotomous RAM lies in the ability to pack many small values in a single word. For any b, one can pack into a word up to $\lfloor w/b \rfloor$ integers, each of b bits. Each of the integers will occupy a range of bits from the word.

Replicating a value: It is easy to construct a word-packed vector of k values using k shifts and bitwise ORs. However, when we want to set all entries to some value x, this can be done in constant time. We just take x and we multiply it with a pattern of the form 10..0 10..0 10..0 ...

Parallel comparison: Given two word-packed vectors, one can perform many natural operations on them in constant time, using word-level parallelism. For example, we can add the vectors (entry-wise), generating a new word-packed vector with the result. This is done by just one addition of the words. If elements have a zero spacing bit between them, carries will not propagate between entries. These spacing bits can be cleared by masking, as above.

A very useful operation is comparing two word-packed vectors A and B, and generating a vector of bits R (with spacing between them), where $R[i] = 1$ iff $A[i] \geq B[i]$. We pack the entries of A and B with one spacing bit between them. The spacing bits of A are made one, by ORing with a fixed pattern, and the spacing bits of B are made zero, by masking. We now subtract B's word from A's word. Notice that we have not borrows between entries because of the carefully arranged spacing bits. Furthermore, a spacing bit immediately above entry i will be one if $A[i] \geq B[i]$ (no borrow from the spacing bit was needed), and will be zero if $A[i] < B[i]$. We can now mask away everything except the spacing bits, which encode the answers to the comparisons.

Predecessor in a word-packed vector: Given a word-packed vector A, arranged such that $A[1] < A[2] < \dots$, and a b-bit number y, we want to find the index of the predecessor of b in A (i.e. the i so that $A[i] < y \leq A[i+1]$). We first replicate y into every entry of a vector B. Then, we perform a parallel comparison of B and A. Now the result is given by the most significant set bit; we find the index of this bit, and divide by $b+1$ to get the real answer.

3 The Fusion Structure

The building block of fusion trees is a static data structure on $k = O(w^{1/5})$ keys, that can be constructed in $k^{O(1)}$ time, takes $O(k)$ space, and can be queried for successor and predecessor in $O(1)$ time. Observe that it is not clear why word packing should help with this problem, since the integers we are considering have w bits, exactly matching the word. The key insight is that we only need a few bits of each word to determine the predecessor.

Let the keys be $x_1, \dots, x_k$. Interpret these numbers as root-to-leaf paths in a binary trie of height w. Consider the tree induced by these paths. Let $b_1, \dots, b_r$ be the heights of the nodes which have more than one child in the induced tree. Call these bit positions the important bits. Note that because these nodes have more than 1 child, there are at most k such nodes, so $r = O(k)$. Notice also that the b_i's are the set of bit positions at which the bit strings of any two x_i, x_j differ for the first time.

We define the sketch of a number to be just the important bits $(b_1, \dots, b_r)$, extracted from that number. Notice that for any y and y the ordering of $\text{SKETCH}(y)$ and $\text{SKETCH}(z)$ is the same as the ordering on the masks of y and z leaving only the bits in the b_i positions. Since x_i and x_j differ for the first time at some b position, we have that $\text{SKETCH}(x_1), \dots, \text{SKETCH}(x_k)$ and $x_1, \dots, x_k$ have the same relative ordering. A fusion structure stores a word-packed vector with all sketches of $x_1, \dots, x_k$. This fits in a word since it takes $O(kr) = O(k^2) = O(w^{2/5})$ bits. Furthermore, we store the real x_i's consecutively, so that given any x_i we can immediately find its successor and predecessor.

In order to query the data structure to find the predecessor and successor of some q, it suffices to find either one or the other. First, we compute $\text{SKETCH}(q)$ and use our parallel comparison to find the i satisfying $\text{SKETCH}(x_i) < \text{SKETCH}(q) \leq \text{SKETCH}(x_{i+1})$. Note that x_i and x_{i+1} do not necessarily have any relation to the predecessor or successor of q. To see that, consider the case when q diverges from its predecessor and successor (in the trie of height w) at some bit position which was not defined as important. Then, that bit position is ignored by the sketch function. The next important bit positions may be arbitrary in q, causing x_i and x_j to be somewhap haphazard.

The crucial observation is that x_i and x_{i+1} nevertheless give some information about the predecessor or successor of q. Assume by symmetry that q diverges from its predecessor lower that the point of divergence with the successor. Then, one of x_i or x_{i+1} must have a common prefix with q of the same length as the common prefix between q and its predecessor. This is because the common prefix remains identical through sketch, and the sketch predecessor can only deviate from the real predecessor below where q deviates from the real predecessor.

The length of the common prefix can be computed by taking bitwise XOR and finding the most significant set bit. Hence we can find the common prefix between q and its predecessor / successor. Assume that the predecessor deviates below. Now the question is how to find the actual predecessor based on the known common prefix. If C is the common prefix, the predecessor is of the form $C0A$, and q has the form $C1B$ for some A and B. We now construct the word $q' = C011..1$, and we find its predecessor in the sketch world (as above). We claim that the sketch predecessor of q' is actually the predecessor of q. Since C is the longest common prefix between q and any x_i, we know that no x_j begins with the string $C1$. Hence, the predecessor of q must begin with $C0$, and it must also be the predecessor of $q' = C011..1$. Now we claim that the predecessor of q' was computed correctly. Note that all important bits lower than $|C|$ are 1, and thus q' remains the maximum in the subtree beginning with C, even after sketching. The maximum element x_i beginning with C is actually the predecessor we want.

4 Approximate Sketches

In the discussion from above, we assumed we could compute SKETCH(q) in constant time. Unfortunately, we cannot do that. What we can do is compute an approximate sketch. An approximate sketch has all bit positions $b_1, \ldots, b_r$ in order, possibly separated by zero bits. These zero bits do not influence the relative order of two sketches, so an approximate sketch is enough for the algorithm from above. The trick is, of course, to compute *small* approximate sketches. We show how to compute appoximate sketches of size $O(w^{4/5})$, by a clever use of multiplication. This explains why we needed to restrict $k = O(w^{1/5})$ in the fusion structure: k approximate sketches have size $O(w)$, so they can be packed in a word.

To summarize, given bit positions $b_1 < b_2 < \ldots < b_r$ with $r = O(w^{1/5})$ we want to construct, in polynomial time, a set of bit positions $c_1 < c_2 < \ldots < c_r < O(w^{4/5})$ and an operation computable in $O(1)$ time that takes a word $x = \sum_{i=0} x_i 2^i$ to SKETCH(x) $= \sum_{i=1}^{r} x_{b_i} 2^{c_i}$. We accomplish this is several steps:

Step 1: Construct $m_1, m_2, \ldots, m_r$ so that each of $b_i + m_j$ are distinct modulo r^3. This can be done iteratively. If we have already picked $m_1, \ldots, m_t$ so that there are no conflicts, it is enough to pick an m_{t+1} that is not congruent to any $m_i + b_j - b_k$ modulo r^3 for $1 \le i \le t$ and $1 \le j, k \le r$. Since there are fewer than r^3 numbers to avoid, there must be some value of m_{t+1} that works.

Step 2: Let m'_i equal m_i plus the correct multiple of r^3 so that $w + r^3(i-1) \le m'_i + b_i < w + r^3 i$. Hence we have that $m'_i + b_j$ are all distinct because they are distinct modulo r^3, and $w \le m'_1 + b_1 < m'_2 + b_2 < \ldots < m'_r + b_r < w + O(r^4)$.

Step 3: We now construct the SKETCH algorithm. Given $x = \sum_{i=0}^{w-1} x_i 2^i$, we mask to leave only the b_i bits, ending up with $\sum_{i=1}^{r} x_{b_i} 2^{b_i}$. Next we multiply this by $m = \sum_{i=1}^{r} 2^{m'_i}$ to get $\sum_{i,j=1}^{r} x_{b_i} 2^{m'_i + b_i}$. Notice that by the results from step 2, that the powers of 2 in this expression are distinct, hence if we consider the second word in this product and mask to consider only bits of the form $m'_i + b_i - w$, we are left with $\sum_{i=1}^{r} x_{b_i} 2^{m'_i + b_i - w}$. Hence if we let $c_i = m'_i + b_i - w$, this satisfies the properties needed for our function.

5 Fusion Trees

We now store the n elements in a balanced B-tree with branching factor $\Theta(w^{1/5})$. At each node of the tree we store a fusion structure which stores values separating the keys stored in each of its subtrees. We can insist that the branching factors are $\Theta(w^{1/5})$ on all levels except for the last. Hence there are $O(\log_w n)$ levels. Queries are performed by performing a query in the fusion structure at each level to determine which subtree to look in. This takes $O(1)$ time per level for a total of $O(\log_w n)$ time. It is clear that this data structure takes linear space.

Making the structure dynamic is a bit more problematic, because rebuilding a fusion node takes $w^{O(1)}$ time. However, we can reuse the idea from last lecture, which shows how to hide a $w^{O(1)}$ factor in the update by bucketing elements into chunks of size $w^{O(1)}$. Now an operation also need to search in the BST of a chunk, so the running times become $O(\lg_w n + \lg w)$. An alternative dynamization is via exponential trees [1].

References

[1] Arne Andersson, Mikkel Thorup: *Tight(er) worst-case bounds on dynamic searching and priority queues*, STOC 2000: 335-342.

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