6.851: Advanced Data Structures	Spring 2007
Lecture 21 — May 2	
Prof. Erik Demaine	*Scribe: Aaron Bernstein*

1 Overview

Up until now, we have mainly studied how to decrease the query time, and the preprocessing time of our data structures. In this lecture, we will focus on storing the data as compactly as possible. Our goal will be to get as close the information theoretic optimum as possible. We will refer to this optimum as OPT. Note that most "linear space" data structures we have seen are still far from the information theoretic optimum because they typically use O(n) *words* of space, whereas OPT usually uses O(n) bits.

Here are some possible goals we can strive for:

- *Implicit Data Structures* – Space = information-theoretic-OPT + O(1). The O(1) is there so that we can round up if OPT is fractional. Most implicit data structures just store some permation of the data: that is all we can really do.
- *Succinct Data Structures* – Space = OPT + o(OPT). In other words, the leading constant is 1.
- *Compact Data Structures* – Space = O(OPT). Note that most "linear space" data structures are not actually compact because they use O(w·OPT) bits.

1.1 mini-survey

- *Implicit Dynamic Seach Tree* – In 2003, Franceschini and Grossi [1] developed an implicty dynamic search tree which supports insert, delete, and search in O(log(n)) time.
- *Succinct Dictionary* – use $\lg\binom{u}{n} + O(\frac{\lg\binom{u}{n}}{\lg\lg\lg u})$ bits [2] or $\lg\binom{u}{n} + O(\frac{n(\lg\lg n)^2}{\lg n})$ bits [6] , and support $O(1)$ membership queries; u is the size of the universe from which the n elements are drawn.
- *Succinct Binary Tries* – The number of possible binary tries with n nodes is $C_n = \binom{u}{n}/(n+1) \approx 4^n$. Thus, OPT $\approx \log(4^n) = 2n$. In this lecture, we will show how to use 2n + o(n) bits of space. We will be able to find the left child, the right child, and the parent in O(1) time. We will also give some intuition for how to answer subtree-size queries in O(1) time. Subtree size is important because it allows us to keep track of the rank of the node we are at.
- *Almost Succinct k-ary trie* – The number of such tries is $C_n^k = \binom{kn+1}{n} / (kn+1) \approx 2^{(log(k)+log(e))n}$. Thus, OPT = (log(k) + log(e))n. The best known data structures was developed by Benoit *et al.* [3]. It uses $(\lceil log(k)\rceil + \lceil log(e)\rceil)$n + o(n) + O(loglog(k)) bits . This representation still supports the following queries in O(1) time: find child with label i, find parent, and find subtree size.

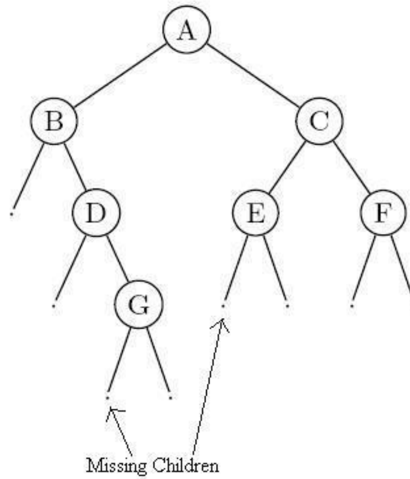

Figure 1: A Binary Trie

- *Succinct Rooted Ordered Trees* – These are different from tries because there can be no absent children. The number of possible trees is C_n, so OPT = 2n. A query can ask us to find the ith child of a node, the parent of a node, or the subtree size of a node. Clark and Munro [4] gave a succinct data structure which uses 2n + o(n) space, and answers queries in constant time.

2 Level Order Representation of Binary Tries

One of the central techniques for Succinctly representing tries is called the *Level Ordered Representation.* We will go through the nodes in level order, and for each one, we will write down 2 bits. The first bit represents whether that node has a left child (1 if it does, 0 if it doesn't), and the second represents whether it has a right child. In the example in Figure 1, we would go through the nodes in the order A,B,C,D,E,F,G, and we would end up with the bit-string B = 11011101000000.

external nodes: Another way of thinking of the level order representation is to add an *external* node wherever we have a missing child. Now, we will go through the nodes (including the external ones) in level order, and write 1 if the node is internal, 0 if it external. It turns out that this gives us the same bit-string as the representation above, except with an extra one in the front. So in figure 1, we would have B = 111011101000000.

2.1 Navigating:

It may seem as though this representation will be very hard to navigate, but the following theorem makes it much easier.

Theorem: The left and right children of the ith internal node are at positions 2i and 2i+1 in the array B.

Proof: Let D be the ith internal node. That is, let D be the ith 1 in our array. Suppose that it is at position i+j. In other words, say that there are j 0's before it. There are i-1 internal nodes before D, so if we include external nodes, these (i-1) nodes have a total of 2(i-1) children. Thus, there are at most 2(i-1) possible nodes that could go between D and left(D). But this includes the (i-1) internal nodes before D, and the j external nodes before D, so there are 2i - 2 - (i-1) - j = i-j-1 nodes between D and left(D). Thus, left(D) is at position 1 + (i+j) + (i-j-1) = 2i. Right(D) is at position 2i+1.

3 Rank and Select

Say that we could support the following operations on an n-bit string in O(1) time, with o(n) extra space:

- rank(i) = number of 1's at or before position i
- select(j) = position of jth one.

This would give us the desired representation of binary tries. The space requirement would be 2n for the level-order representation, and o(n) space for rank/select. Here is how we would support queries:

- left-child(i) = 2rank(i)
- right-child(i) = 2rank(i) + 1
- parent(i) = select($\lfloor i/2 \rfloor$)

3.1 Rank

This algorithm was developed by Jacobsen, in 1989 [5]. It uses many of the same ideas as RMQ. The basic idea is that we use a constant number of recursions until we get down to sub-problems of size k = log(n) / 2. Note that there are only $2^k = \sqrt{n}$ possible strings of size k, so we will just store a lookup table for *all possible bit strings of size k.* For each such string we have k = O(log(n)) possible queries, and it takes log(k) bits to store the solution of each query (the rank of that element). Nonetheless, this is still only O($\sqrt{n}$log(n)loglog(n)) = o(n) bits.

First Attempt: Just as in RMQ, we will split the bit string into 2n/log(n) chunks of size log(n)/2. To find rank(i), we need to find (rank of i in its chunk) + (number of 1's in all preceding chunks). We can find rank(i) within a chunk by just looking in our lookup table. But we also need, for each chunk, the total number of 1's among all of the preceding chunks. We will call this the *cumulative rank* for each chunk. Unfortunately, there are O(n/log(n)) chunks, and each cumulative rank takes O(log(n)) bits to represent, so we will end up with Ω(n) space, which is too big.

Second Attempt: The solution is to use one more level of recursion. We will split into n/$\log^2$(n) chunks of size $\log^2$(n). Now, we can store cumulative ranks in O((n/$\log^2$(n))*log(n)) = o(n) space. To solve rank(i) within a chunk, we will recurse. We will split our chunks into *mini-chunks* of size

log(n)/2. We can solve the mini-chunks with our lookup table. But this time, storing cumulative rank is cheaper since we only have to store the cumulative rank *within the parent chunk*, so it will take O(loglog(n)) bits. Thus, we will only need O(nloglog(n)/log(n)) = o(n) bits.

3.2 Select

This algorithm was developed by Clark and Munro in 1996 [4]. Select is similar to rank, although more complicated. This time, since we are trying to find the position of the ith one, we will break our array up into chunks with equal amounts of ones, as opposed to chunks of equal size.

Step 1: First, we will pick every (log(n)loglog(n))th 1 to be a *special one*. We will store the index of every special one. Storing an index takes log(n) bits, so this will take O(nlog(n)/(log(n)loglog(n))) = O(n/loglog(n)) = o(n). Now, we need to restrict our attention to a single chunk: a sequence of bits between two special ones (note that there are O(n/(log(n)loglog(n))) chunks). Let r be the *total* number of bits in a chunk. If r $>$ $\log^2$(n), we will go to step 2. If r $\leq$ $\log^2$(n), we will go to step 3.

Step 2: There are at most O(n/log^2(n)) chunks of size greater than $\log^2$(n). Thus, we can afford to just brute force the problem by storing the index (in our bit-string) of every 1 in the chunk. There are log(n)loglog(n) ones, and storing each index takes O(log(n)) space, so the total space, over all of these large chunks, is O(nlog(n)loglog(n)log(n)/$\log^2$(n)) = O(n/loglog(n)) = o(n).

Step 3: In this case we recurse again. This time, within a chunk, we pick every $(\text{loglog(n)})^2$th one to be a mini-special one. We then split up into mini-chunks. If a chunk has size greater than $(\text{loglog(n)})^4$, then we brute force as in step 2. There are at most n/$(\text{loglog(n)})^4$ such chunks, each one contains O($(\text{loglog(n)})^2$) ones, and storing each index takes O(log($\log^2$(n))) = O(loglog(n)) bits. Thus, the overall number of bits will be O(n$(\text{loglog(n)})^3$/$(\text{loglog(n)})^4$) = O(n/loglog(n)) = o(n) space. If a chunk is smaller than $(\text{loglog(n)})^4$, then we recurse one last time. But note that $(\text{loglog(n)})^4$ is tiny, so we can afford to store a lookup table for all possible chunks of this size (just as we did in rank).

4 Subtree Sizes

We have shown a Succinct binary trie which allows us to find left children, right children, and parents. But we would still like to find sub-tree size in O(1) time. Level order representation does not work for this, because level order gives no information about depth. Thus, we will instead try to encode our nodes in depth first order.

In order to do this, notice that there are C_n (catalan number) binary tries on n nodes. But there are also C_n rooted ordered trees on n nodes, and there are C_n balanced parentheses strings with n parentheses. Moreover, we will describe a bijection: binary tries $\Leftrightarrow$ rooted ordered trees $\Leftrightarrow$ balanced parentheses. This makes the problem much easier because we can work with balanced parentheses, which have a natural bit encoding: 1 for an open parentheses, 0 for a closed one.

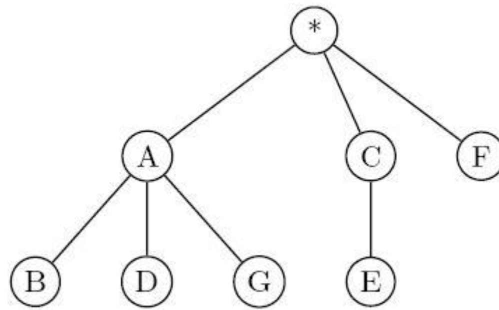

Figure 2: A Rooted Ordered Tree That Represents the Trie in Figure 1

```
( ( ( ) ( ) ( ) ) ( ( ) ) ( ) )
* A B B D D G G A C E E C F F *
```

Figure 3: A Balanced Parentheses String That Represents the Ordered Tree in Fiugre 2

4.1 The Bijections

We will use the binary trie in figure 1. To make this into a rooted ordered tree, we can think of rotating the trie 45 degrees counter-clockwise. Thus, the top three nodes of the tree will be the right spine of the trie (A,C,F). To make the tree rooted, we will add an extra root *. Now, we recurse into the left subtrees of A,C, and F. For A, the right spine is just B,D,G. For C, the right spine is just E: C's only left child. Figure 2 shows the resulting rooted ordered tree.

To go from rooted ordered trees to balanced parentheses strings, we do a DFS of the ordered tree. We will then put an open parentheses when we first touch a node, and then a closed parentheses the second time we touch it. Fig 3 contains a parentheses representation of the ordered tree in figure 2.

Now, we will show how the queries are transformed by this bijection. For example, if we want to find the parent in our binary trie, what does this correspond to in the parentheses string? The bold-face is what we have in the binary trie, and under that, we will describe the corresponding queries from the 2 bijections.

Node:

Rooted Ordered Tree: Also, just a node.

Parentheses String: An open parentheses that corresponds to the first time we visited v.

Left-Child(v)

Rooted Ordered Tree: First-Child(v)

Parentheses String: The next parentheses. If it is closed, then v has no children.

Right-Child(v):

Rooted Ordered Tree: The next sibling of v.

Parentheses String: The parentheses after the matching closed parentheses of v. If this is closed, v has no right child.

Subtree-Size(v):

*Rooted Ordered Tree:*Subtree-size(v) + $\Sigma_{w \in RS}$subtree-size(w). RS stands for the set of right siblings of v.

Parentheses String: 1/2 of the distance to the matching closed parentheses.

Parent(v):

Rooted Ordered Tree: v's left sibling, if it exists. Otherwise, v's parent.

Parentheses String: The nearest enclosing ().

References

[1] G. Franseschini, R.Grossi *Optimal Worst-case Operations for Implicit Cache-Oblivious Search Trees*, Prooceding of the 8th International Workshop on Algorithms and Data Structures (WADS), 114-126, 2003

[2] A.Brodnik, I.Munro *Membership in Constant Time and Almost Minimum Space*, Siam J. Computing, 28(5): 1627-1640, 1999

[3] D.Benoit, E.Demaine, I.Munro, R.Raman, V.Raman, S.Rao *Representing Trees of Higher Degree*, Algorithmica 43(4): 275-292, 2005

[4] D.Clark, I.Munro *Eifficent Suffix Trees on Secondary Storage*, SODA, 383-391, 1996.

[5] G.Jacobson *Succinct Static Data Structures*, PHD.Thesis, Carnegie Mellon University, 1989.

[6] R. Pagh: *Low Redundancy in Static Dictionaries with Constant Query Time*, SIAM Journal of Computing 31(2): 353-363 (2001).