6.897: Advanced Data Structures Spring 2003

Lecture 12 — Monday, April 7, 2003

Prof. Erik Demaine *Scribes: Wayland Ni and Gautam Jayaraman*

1 Overview

In the last lecture we solved the static *least common ancestor* (LCA) problem in constant time per operation after linear preprocessing. We also built suffix trees in linear time after sorting the alphabet.

In this lecture we establish *succinct data structures*, which are [typically static] structures that use the *information-theoretic optimum* space up to lower-order terms and support basic navigational operations quickly.

2 Succinct Data Structures

Our goal is to use space equal to the *information-theoretic optimum* (or *information-theoretic lower bound*) plus lower-order terms and still be able to manipulate and search the structure. Specifically, for an n-bit string, we would like to get the right constant term in front of the n instead of just settling for $O(n)$.

2.1 Binary Trees/Tries

For a given binary trie, we would like to know how many bits it would take to encode the trie. We start by noting that the number of distinct rooted labelled binary tries on n nodes is the **Catalan number** C_n, where $C_n \equiv \binom{2n}{n}/(n+1)$. Then, on average, it would take $\lg C_n$ bits to encode a single trie.

$$\begin{aligned}
\lg C_n &= \lg \frac{(2n)!}{(n!)^2} - \lg(n+1) \\
&= \lg(2n)! - 2\lg(n!) - \lg(n+1) \\
&= 2n\lg(2n) - 2n\lg n + o(n) \qquad \text{(from Stirling's Approximation)} \\
&= (2n)(1+\lg n) - 2n\lg n + o(n) \\
&= 2n + o(n)
\end{aligned}$$

So we can intuitively (log-asymptotically) think of C_n as being roughly 4^n, because $\lg C_n \sim 2n$.

2.1.1 Sidenote: Prefix Codes

Prefix codes are inserted at the beginning of a code to establish the length of the encoding, so the reader knows when to stop reading. A good tool to know is how to make codes into prefix codes.

Given an n-bit string, we must encode its length. Of course, $\lg n$ bits are needed to encode the length of the string. But then, we also need $\lg\lg n$ bits to encode the length of the extra $\lg n$ bits, and $\lg\lg\lg n$ bits to encode the length of the extra $\lg\lg n$ bits, etc. Thus, the length of the optimal prefix code would be $n + \lg n + \lg\lg n + \cdots + 1 + \lg^* n$.

However, since the length of the optimal code is such a complex expression, we can simplify the code by using a little extra space. We use $\lg n$ bits (the "prefix") to encode the length of the n-bit string and another $\lg n$ bits (the "pre-prefix") to encode the length of the $\lg n$-bit prefix in unary. In other words, we repeat 0's followed by a 1 to indicate the length. The total length would then be $n + 2\lg n$.

2.2 Level-Order Representation

The trivial way to use the optimum space, $\lg C_n \approx 2n + o(n)$, would be to use a lookup table so we could refer to a specific binary trie by number. However, our goal is to be able to navigate and operate on the structure efficently as well.

Definition: An *internal node* is a node that contains an element.

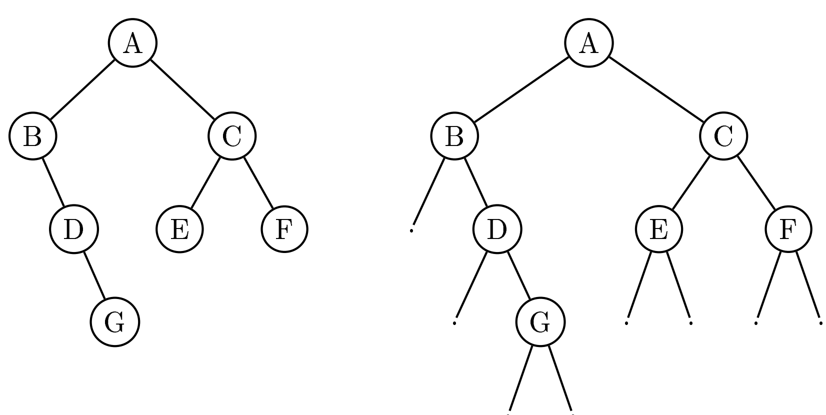
Definition: An *external node* is a node without an element. External nodes are appended to the tree for each missing child.

We now examine an encoding that is more navigable, called *level-order encoding.* To construct the encoding, we first append an *external node* for each missing child. Now, there are $2n+1$ nodes, so we spend 1 bit per node. We visit the nodes in level order, that is, we examine the nodes one row at a time, starting at the root and continuing downwards, going from left to right within each row.

We write down a 1 for an *internal node* and a 0 for an *external node.* Since this encoding takes $2n+1$ total bits, it must be optimal.

Here is an example encoding for the tree below:

```
A B C * D E F * G * * * * * *
1 1 1 0 1 1 1 0 1 0 0 0 0 0 0
```



Visual representation of binary trie (left) and with added missing children (right).

Claim 1. *The positions of the left and right children of the ith internal node are $2i$ and $2i+1$.*

Proof. Suppose there are j external nodes up to the ith internal node. We show that strictly between the ith internal node and its left child, there are $i - j - 1$ nodes:

- offspring of $i - 1$ previous internal nodes $= (2(i - 1))$.
- $i - 1$ of which were internal nodes, (all but root).
- j of which were external nodes.

Thus, the number of nodes strictly between the ith internal node and its left child is $2(i-1) - (i-1) - j = i - j - 1$.

Therefore, the position of left child $= i + j + (i - j - 1) + 1 = 2i$.

2.3 Rank and Select

We define the functions **Rank(**i**)** and **Select(**i**)** in a static bit string:

Rank(i) $\equiv$ Number of 1's at or before index i
Select(i) $\equiv$ Index of ith 1 bit

Note that Rank(i) and Select(i) are inverses. We can now define **left-child(**i**)**, **right-child(**i**)** and **parent(**i**)** as functions of Rank(i) and Select(i):

left-child(i) $\equiv 2 \times$ Rank(i)
right-child(i) $\equiv 2 \times$ Rank(i) $+ 1$
parent(i) $\equiv$ Select($\lfloor i/2 \rfloor$)

Our goal is to support these queries in $O(1)$ time and $o(n)$ extra space.

2.3.1 Rank

Suppose we divide up the string into bit strings of length $\frac{1}{2}\lg n$. Using a lookup table on each piece would require $O(\sqrt{n}\lg n \lg\lg n)$ space. Between each pair of pieces, we must store the number of 1's so far from the left. Since we split the string into $\frac{n}{1/2lgn} = \frac{2n}{\lg n}$ pieces, and $\lg n$ bits are needed to store the cumulative counts of 1's, the total cost is $\Theta(n)$, which is too much.

Instead, using the idea from [J89], we can split the string into $\frac{n}{(\lg n)^2}$ pieces of length $(\lg n)^2$. Then storing the cumulative ranks will cost $O(\frac{n}{(\lg n)^2}\lg n) = O(\frac{n}{\lg n})$ bits, which is $o(n)$.

Now we must split the $(\lg n)^2$-size chunks into $\frac{1}{2}\lg n$-size sub chunks. We then store the cumulative ranks within chunks between each pair of sub chunks. Essentially, we are storing "local" counts of 1 bits between every pair of $\frac{1}{2}\lg n$-bit sub chunks. Moreover, since each count is local to the specific chunk, the maximum number of 1 bits is $(\lg n)^2$, which takes $\lg\lg n$ bits to encode. Since there are $\frac{n}{\lg n}$ total subchunks, and each local count can be encoded by $\lg\lg n$ bits, the amount of space used is $O(\frac{n}{\lg n}\lg\lg n) = o(n)$.

At the bottom level of the structure, a simple lookup table can be used.

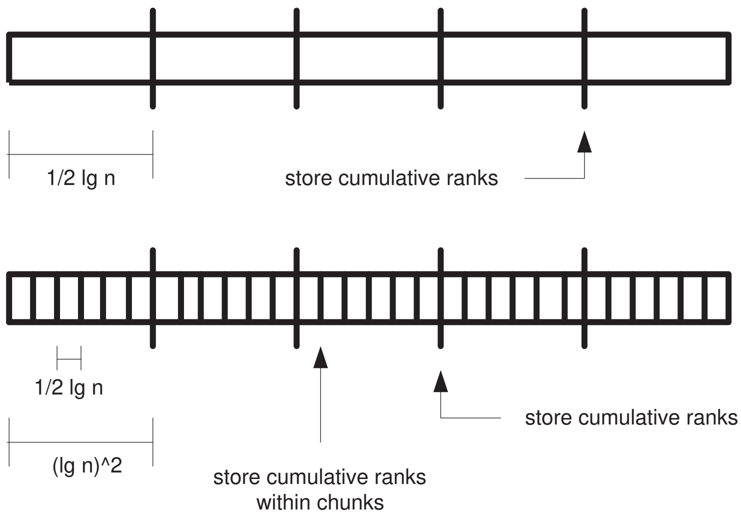


Figure 1: Division of strings into bit strings of length $\frac{1}{2}\lg n$ (top) and length $(\lg n)^2$ (bottom)

2.3.2 Select

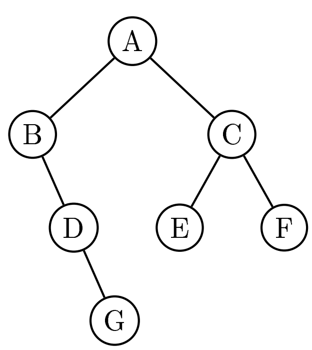
Implementing **Select** is also possible but much harder, because the splits aren't "even" in terms of sparseness of 1 bits. The (complicated) details can be found in [CM96].

2.4 Transformations

We now show three succinct binary trie representations, and how to move between them. These three representations, along with their isomorphism, were discussed in [MR97].

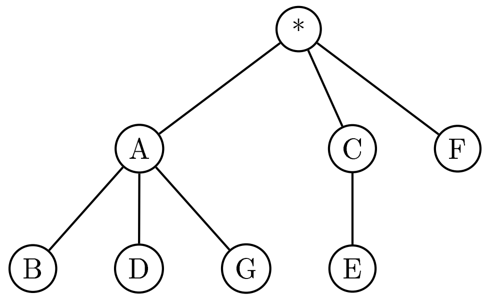
2.4.1 ["Normal"] Binary Trie

This is the type of tree we're used to in e.g. balanced search trees. Each node in a binary tree may or may not have a left child, and may or may not have a right child, but these two children are always distinguished.



2.4.2 Rooted Ordered Tree

Each node in a rooted ordered tree has any number of children (0, 1, 2, ...) and these children are ordered. But you cannot have a second child without having a first child (as you can in binary trees).



A rooted ordered trie can be constructed from a binary trie as follows:

1. Make a **new root node**, called *****.
2. Set the old root node to be the left-child of *****.
3. Examine the nodes one at a time in order. Whenever examining a node that is a right-child, move that node (and its subtree) up so it becomes a right-sibling of its former parent.

2.4.3 Balanced Parentheses

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

These can be generated by taking a DFS walk over the rooted ordered trie, writing a left parenthesis when expanding a node, and a right parenthesis when we are finished with its subtree.

2.4.4 Equivalent Relations

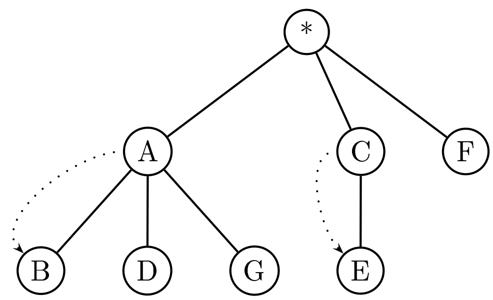
Since these three representations are isomorphic, we can establish equivalencies between common operations on each representation. Each heading below identifies an operation on an ordinary Binary Trie, then the ensuing section explains the equivalent operations for the other two representations. Some of the sections also depict all the examples of that relation in the root-ordered tree and the balanced parentheses.

- **node**
  Rooted Ordered Equivalent: Node. It's the same.
  Balanced Parentheses Equivalent: The position where a left parenthesis corresponds to the node label.

(	(	(	)	(	)	(	)	)	(	(	)	)	(	)	)
*	$\underline{A}$	$\underline{B}$	B	$\underline{D}$	D	$\underline{G}$	G	A	$\underline{C}$	$\underline{E}$	E	C	$\underline{F}$	F	*

Balanced parentheses example with all the node positions underlined.

- **left-child(node)**
  Rooted Ordered Equivalent: The leftmost child of **node**. If **node** has no children, then **null**.



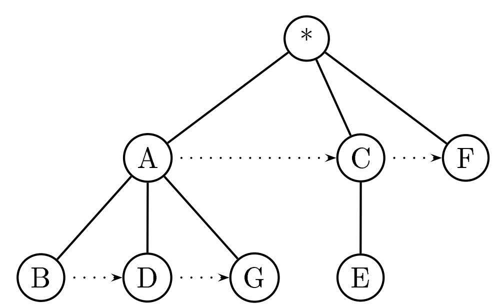
Rooted ordered example with all the left-child relations depicted in dotted lines.

Balanced Parentheses Equivalent: If the next position i after **node** contains a left parenthesis, then$i$. Otherwise **null**.

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

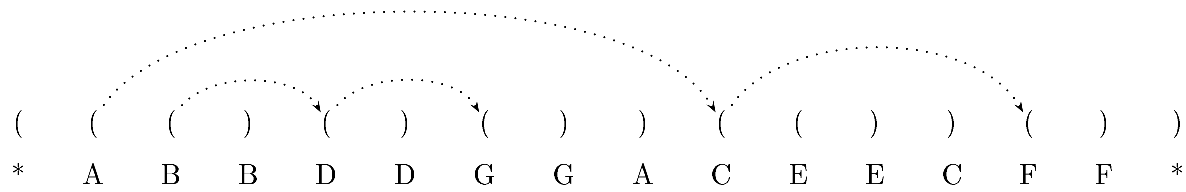
Balanced parentheses example with all the left-child relations depicted in dotted lines.

- **right-child(node)**
  Rooted Ordered Equivalent: The next *sibling* to the right of **node**, if there is one.



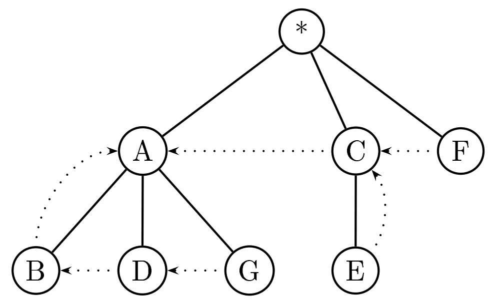
Rooted ordered example with all the right-child relations depicted in dotted lines.

Balanced Parentheses Equivalent: The position i immediately following the right-parenthesis that matches the left-parenthesis at **node**, as long as i contains a left-parenthesis.



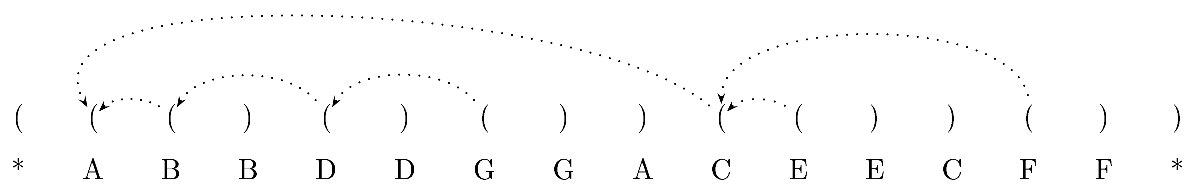
Balanced parentheses example with all the right-child relations depicted in dotted lines.

- **parent(node)**
  Rooted Ordered Equivalent: The left sibling of **node**, if it exists. Else the parent of **node**.



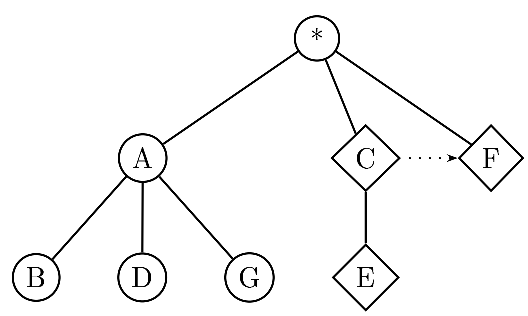
Rooted ordered example with all the parent relations depicted in dotted lines.

Balanced Parentheses Equivalent: If character i immediately preceeding **node** is a right-parenthesis, the node at the matching left-parenthesis of i. Else i.



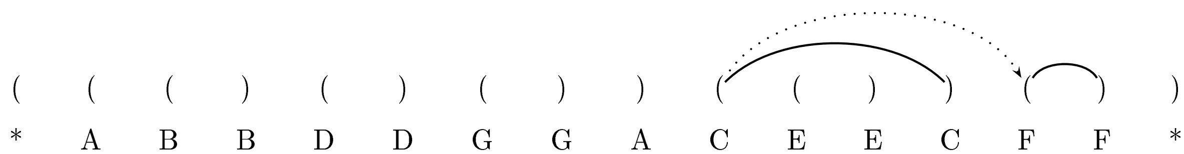
Balanced parentheses example with all the parent relations depicted in dotted lines.

- **subtree-size(node)** – (This is useful because it facilitates the computatation of indices of external nodes, which are indices into suffix arrays.)
  Rooted Ordered Equivalent: size of **node** plus sizes of all rightward siblings (if any).



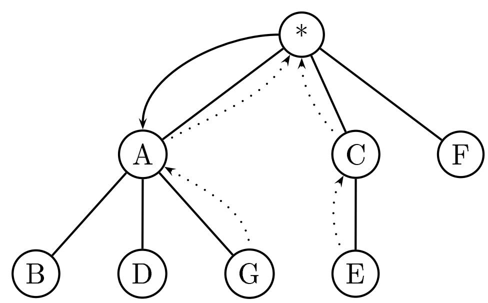
Rooted ordered example depicting subtree at C using
diamond-shaped nodes, showing that subtree-size(C) = 3.

Balanced Parentheses Equivalent: size of **node** plus sizes of all rightward siblings (if any). Size of a node equals (size of inclusive interval between left paren at start of node and matching right paren) / 2.



Balanced parentheses example showing subtree size of C, which is size of C (first solid arc) plus size of F (second solid arc), which is 4/2 + 2/2 = 3.

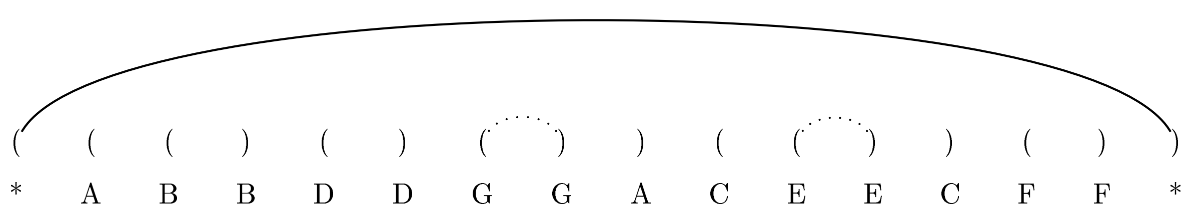
- **lca(node1,node2)**
  Rooted Ordered Equivalent: The more-left sibling among the two children of lca that are on the unique path between **node1** and **node2** .



Rooted ordered example showing how we find (first following dotted arrows, then solid) that lca(E,G) = A.

Balanced Parentheses Equivalent: The more-left sibling among the two children of lca that are on the unique path between **node1** and **node2**. The lca of **node1** and **node2** in this representation is the position of the left-paren of the *minimal common enclosing parenthesis pair* of **node1** and **node2**.

We define **minimal common enclosing parenthesis pair** of a node in the balanced parentheses representation as the pair of matching parentheses that define an interval of minimal length that contains the node.



Balanced parentheses example showing (with solid arc) that * is the minimal common enclosing parenthesis pair of G and E

Summary of Equivalent Relations

Binary Trie	**Rooted Ordered Trie**	**Balanced Parentheses**
node	node	position of left paren with node label
left-child(node)	first-child of node	next position, only if it is **(**
right-child(node)	next-sibling to right of node	next position after **)** that matches **(** of node
parent(node)	prev-sibling of node (if none then parent of node)	prev position (i) to node, if it has "**(**". else the "**(**" that matches paren at i.
subtree-size	size of node + sizes of rightward siblings of node	size of node + sizes of rightward siblings of node, where size of node = dist from "**(**"to matching "**)**" / 2
lca(node1,node2)	among children of lca, the leftmost one along the path between **node1** and **node2**	same as in rooted ordered trie, with lca = positon of left-paren of min common enclosing paren pair of **node1** and **node2**

2.4.5 Performance

In [MR97] all of the above operations are accomplished in $O(1)$ time except lca. In [BDMR99] the results are extended to *cardinal* (or *k-ary*) trees, including the ability to find the ith child of a node in $O(1)$ time.

2.5 Other Succinct Data Structures

2.5.1 "Cardinal" (k-ary) Trees

These are trees where a node may have up to k children, so k can be thought of as the cardinality of the alphabet Σ. For such trees the information-theoretic lower bound is $C_n^k \equiv \binom{kn+1}{n}/(kn+1)$.

The structure described in [BDMR99] takes up space in between $(\lg e + \lg k)n$ and $(2 + \lceil \lg k \rceil)n$ bits, making it succinct for cardinal trees. It finds the child labeled i in time $O(\lg \lg k)$. This bound was improved to $O(\lg \lg \lg k)$ in [RR99].

OPEN question: Can this bound be reduced to constant time?

2.5.2 Suffix Tree Equivalents

Suffix trees, introduced in Lecture 10, have various uses in text searching. This has prompted research to find succinct representations for them. Papers by Grossi, Vitter, Ferragina, Sadakane, as well as some from SODA 2003, provide good samples of this research. There are still unresolved questions about pattern matching with succinct suffix tries. For example:

OPEN question: Using $O(n)$ bits of space can you find one or more matches to a pattern P in a text T in time $O(|P|)$?

References

[BDMR99] D. Benoit, E.D. Demaine, J.I. Munro and V. Raman. *Representing Trees of Higher Degree.* WADS 1999.

[CM96] D.R. Clark and J.I. Munro. *Efficient Suffix Trees on Secondary Storage.* SODA 1996.

[J89] G. Jacobson. *Succinct Static Data Structures.* PhD Thesis, Carnegie-Mellon, 1989.

[MR97] J.I. Munro and V. Raman. *Succinct Representation of Balanced Parentheses, Static Trees and Planar Graphs.* FOCS 1997.

[MRS01] J.I. Munro, V. Raman, and A. Storm. *Representing Dynamic Binary Trees Succinctly.* SODA 2001.

[RRR01] R. Raman, V. Raman, and S.S. Rao. *Succinct Dynamic Data Structures.* WADS 2001.

[RR99] V. Raman and S.S. Rao. *Static Dictionaries Supporting Rank.* ISAAC 1999.