Succinct Trees: Theory and Practice

Kunihiko Sadakane
National Institute of Informatics, Japan
Dec. 1, 2011
Succinct Data Structures

• Succinct data structures
  = succinct representation of data + succinct index

• Examples
  – sets
  – trees, graphs
  – strings
  – permutations, functions
Succinct Representation

- A representation of data whose size (roughly) matches the information-theoretic lower bound.
- If the input is taken from $L$ distinct ones, its information-theoretic lower bound is $\lceil \log L \rceil$ bits.
- Example 1: a lower bound for a set $S \subseteq \{1,2,\ldots,n\}$
  - $\log 2^n = n$ bits
  - $\emptyset$
  - $n = 3$
  - $\{1\}$  $\{2\}$  $\{3\}$
  - $\{1,2\}$  $\{1,3\}$  $\{2,3\}$
  - Base of logarithm is 2
  - $\{1,2,3\}$
• Example 2: $n$ node ordered tree

\[
\log \frac{1}{2n-1} \binom{2n-1}{n-1} = 2n - \Theta(\log n) \text{ bits}
\]
• For any data, there exists its succinct representation
  – enumerate all in some order, and assign codes
  – can be represented in $\lceil \log L \rceil$ bits
  – not clear if it supports efficient queries
• Query time depends on how data are represented
  – necessary to use appropriate representations

\[
n = 4 \left\lceil \log \frac{1}{2n-1} \binom{2n-1}{n-1} \right\rceil = \lceil \log 5 \rceil = 3 \text{ bits}
\]
Succinct Indexes

• Auxiliary data structure to support queries
• Size: $o(\log L)$ bits
• (Almost) the same time complexity as using conventional data structures
• Computation model: word RAM
  – assume word length $w = \log \log L$
    (same pointer size as conventional data structures)
word RAM

• word RAM with word length $w$ bits supports
  – reading/writing $w$ bits of memory at arbitrary address in constant time
  – arithmetic/logical operations on two $w$ bits numbers are done in constant time
  – arithmetic ops.: $+$, $-$, $\times$, $/$, $\log$ (most significant bit)
  – logical ops.: and, or, not, shift

• These operations can be done in constant time using $O(w^\varepsilon)$ bit tables ($\varepsilon > 0$ is an arbitrary constant)
A Variety of Trees

- Ordered/Unordered trees
  - Child nodes are ordered/not ordered
  - Concerning unordered trees, trees can be regarded as the same if they become the same shape by reordering children
- Edges are labeled/not labeled
  - Node labels can be represented by edge labels on the edge toward parent nodes
- In this talk, we consider labeled ordered trees.
Applications of Ordered Trees

• An abstract data type for “dictionary”
• A data structure $D$ is called dictionary if for a set $S$ and a key $k$, it supports
  – $\text{Search}(D, k)$: returns Yes iff $k \in S$
  – $\text{Insert}(D, x)$: adds $x$ to $S$
  – $\text{Delete}(D, x)$: removes $x$ from $S$
• A binary search tree is used as a dictionary.
  – Any balanced binary search tree supports the above operations in $O(\log n)$ time for a set of $n$ elements.
  – We assume that element comparison is done in $O(1)$ time.
Radix Search Trees

• We assume keys are $b$-bit integers
• All elements are stored in leaves
• Left (right) subtree of root stores all the keys whose first bit is 0 (1).
• To search for a key, we traverse the tree from the root. At a node with depth $d$, we go to left (right) if $d$-th bit of the key is 0 (1).
• All operations are done in $O(b)$ time.
Tries

• Trie is a data structure for storing a set of strings.
• A node has at most \( \sigma \) children (\( \sigma \): alphabet size)
• Each edge is labeled a character \( c \)
• The concatenation of characters on edges from the root to a node coincides with the string represented by the node

A Trie for \( S = \{\text{cab, cat, do, doc, dog}\} \)
Compressed Tries

- Compressed Tries are obtained from standard Tries by compressing chains of *redundant* nodes
  - redundant node = a node with only one child
  - \#nodes \leq \#leaves – 1

- An edge represents a string

\[
S = \{\text{cab, cat, do, doc, dog}\}
\]
Suffix Trees

- The compressed trie for all suffixes of a string
- The suffix tree for a string consists of
  - $n$ leaves ($n$: length of string)
  - $\leq n-1$ internal nodes
  - edge labels
  - node depths
  - suffix links

$ababac$

1234567

$\$
Operations on Trees

- **Binary search trees**
  - `left(v)`, `right(v)`: returns the left/right child node of `v`
  - `key(v)`: returns the key stored in node `v`

- **Tries**
  - `child(v, c)`: returns a child `w` of `v` with edge label `c`
  - `key(v)`: returns the key stored in node `v`

- **Compressed tries**
  - `child(v, c)`: returns a child `w` of `v` with edge label `c`...
  - `edge(w, d)`: returns `d`-th character on edge pointing to `w`
  - `key(v)`

```
child(v, a) = w
edge(w, 2) = b
```
Succinct Representations of Ordered Trees

- LOUDS (level order unary degree sequence)
  - [Jacobson 89]
- BP (balanced parentheses)
  - [Munro, V. Raman 97, 01]
- DFUDS (depth first unary degree sequence)
  - [Benoit et al. 99, 05]
LOUDS Representation

• Degrees of nodes are encoded by unary codes in breadth-first order
  – degree $d \rightarrow 1^d0$

• $2n+1$ bits for $n$ nodes (matches the lower bound)

• $i$-th node is represented by $i$-th 1 (ones-based numbering)
Tree Navigational Operations (1)

- $i$-th node: $\text{select}_1(L, i)$ ($i \geq 1$)
- $\text{firstchild}(x)$
  - $y := \text{select}_0(\text{rank}_1(L, x)) + 1$
  - if $L[y] = 0$ then $-1$ else $y$
- $\text{lastchild}(x)$
  - $y := \text{select}_0(\text{rank}_1(L, x) + 1) - 1$
  - if $L[y] = 0$ then $-1$ else $y$
Tree Navigational Operations (2)

- **$\textit{siblings}(x)$**
  - if $L[x+1] = 0$ then $-1$ else $x+1$

- **$\textit{parent}(x) = \text{select}_1(\text{rank}_0(L, x))$**

- **$\textit{degree}(x) = \text{lastchild}(x) - \text{firstchild}(x) + 1$**

- Merits: implemented by only $\text{rank/select}$

- Demerits: cannot compute subtree sizes
Tries using LOUDS

• child($v$, $c$)
  
  $w = \text{firstchild}(v)$, $r = rank_1(L, w)$, $k = 0$

  while ($L[w+k] \neq 0$) {
    
    if ($C[r+k] == c$) return $w+k$

    $k = k+1$

  }

  – Binary search can be also used

• key($v$) is stored in array indexed with $rank_1(L, v)$

  LOUDS  
  
  $L = 101101110110000000$

  $C\quad a\quad c\quad x\quad y\quad z\quad a\quad d$
• A trie with \( n \) nodes and size-\( \sigma \) label set supporting \( \text{child}(v, c) \) in \( O(\log \sigma) \) time is represented in \( n(2+\log \sigma) + o(n) \) bits.
  – \( O(\sigma) \) time sequential search is faster for small \( \sigma \).

• By using an auxiliary array, \( \text{key}(v) \) is performed in \( O(1) \) time.
## Numbering of Nodes

<table>
<thead>
<tr>
<th></th>
<th>Ones-based</th>
<th>Zeros-based</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>i-th node</strong></td>
<td>$s_1(i)$</td>
<td>$s_0(i+1)$</td>
</tr>
<tr>
<td><strong>firstchild</strong>(x)</td>
<td>$s_0(r_1(x))+1$</td>
<td>$s_0(r_1(s_0(r_0(x)−1)+2))$</td>
</tr>
<tr>
<td><strong>lastchild</strong>(x)</td>
<td>$s_0(r_1(x)+1)−1$</td>
<td>$s_0(r_1(x)+1)$</td>
</tr>
<tr>
<td><strong>sibling</strong>(x)</td>
<td>$x+1$</td>
<td>$s_0(r_0(x))$</td>
</tr>
<tr>
<td><strong>parent</strong>(x)</td>
<td>$s_1(r_0(x))$</td>
<td>$s_0(r_0(s_1(r_0(x)−1)+1))$</td>
</tr>
</tbody>
</table>

**Ones-based**

$L = 101101110110000000$

**Zeros-based**

$s_c(i) = \text{select}_c(L, i)$, $r_c(i) = \text{rank}_c(L, i)$

![Tree Diagram]

\[
\begin{align*}
\text{Ones-based:} & \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \\
\text{Zeros-based:} & \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \\
\end{align*}
\]
Double Numbering

• A node is represented by a pair $\langle x, y \rangle$
  – $y$: the level-order of the node
  – $x$: the position in $L$ for the node (ones- or zeros- based)

• $parent(\langle x, y \rangle)$ is done in ones-based numbering by
  – $r = x - y$
  – $x' = s_1(r)$
  – $y' = x' - r$
  – return $\langle x', y' \rangle$

• $(parent(x) = s_1(r_0(x)))$

• The value of $r_0(x)$ can be derived from $s_1(r)$, which was already done when $\langle x, y \rangle$ was obtained.
LOUDS++

• [Delpratt, Rahman, R. Raman 06]
• LOUDS sequence $L$ is represented by runs of ones and runs of zeros
• parent, firstchild, lastchild, sibling are done by select_1 and rank_1 on $R_1$ and $R_0$

$\begin{align*}
R_0 &\quad 1110000001 \\
R_1 &\quad 10100101
\end{align*}$

LOUDS $L$ 101101110110000000
The Merits and Demerits of LOUDS

• Implemented by *rank* and *select*
  – easy to implement
  – fast in practice

• Suitable for labeled trees
  – child() is fast because of locality of reference
  – Tx library by Okanohara

• Many operations are not supported in O(1) time
  – subtree size
  – level ancestor
  – lowest common ancestor, etc.
BP Representation

- Each node is represented by a pair of matching open and close parentheses
- $2n$ bits for $n$ nodes
- The size matches the lower bound
Basic Operations on BP

- A node is represented by the position of ( 
- \textit{findclose}(P,i): returns the position of ) matching with ( at $P[i]$ 
- \textit{enclose}(P,i): returns the position of ( which encloses ( at $P[i]$ 

$$P( ( ( ( ( ( ( ( ( ( ( ( ( ) ) ) ) ) ) ) ) ) ) ) )$$
Tree Navigational Operations

- $parent(v) = enclose(P,v)$
- $firstchild(v) = v + 1$
- $sibling(v) = findclose(P,v) + 1$
- $lastchild(v) = findopen(P, findclose(P,v) - 1)$
Tries using BP

- **child**\((v, c)\)
  
  \[ w = firstchild(v) \]

  while \((w \neq \text{NIL})\) {
  
  if \((C[rank(P, w)] == c)\) return \(w\)
  
  \(w = \text{sibling}(w)\)
  
  }

- **key**\((v)\) is stored in array indexed with \(rank(P, v)\)

\[
\text{BP} \quad P \quad ( ( ( ( ) ) ( ) ) ) \quad ( ( ( ) ) )
\]

\[
C \quad _a \text{x} \text{y} \text{z} \text{c} \text{a} \text{d}
\]
Number of Descendants (Subtree Size)

- The size of the subtree rooted at \( v \) is
  \[
  \text{subtreesize}(v) = \frac{\text{findclose}(P,v) - v + 1}{2}
  \]
- degree (#children) can be computed by repeatedly applying \( \text{findclose} \), but it takes time proportional to the number of children.
History

- [Munro, V. Raman 97, 01]
  - firstchild, sibling, lastchild, parent, subtreesize, depth
- [Munro, V. Raman, Rao 98, 01]
  - leftmostleaf, rightmostleaf, #leaves
- [Sadakane 02, 07]
  - lca (lowest common ancestor)
- [Munro, Rao 04]
  - level ancestor
- [Geary, Rahman, R. Raman, V. Raman 04, 06]
  - simpler findopen and enclose
- [Lu, Yeh 08]
  - degree, i-th child
- [Sadakane, Navarro 10]
  - simpler and smaller
Additional Basic Operations on BP

- $\text{rank}_p(P,i)$: number of pattern $p$ in $P[1..i]$
- $\text{select}_p(P,i)$: position of $i$-th occurrence of $p$ in $P$
- If the length of $p$ is constant, \textit{rank/select} is done in $O(1)$ time

$P = (((()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) (()) ()}
Operations on Leaves

- Each leaf is represented by ( ) in BP
- Position of $i$-th leaf = $select_0(P, i)$
- Number of leaves in a subtree, leftmost/rightmost leaf in a subtree are also found

Subtree rooted at 3
Node Depths

• Define excess array $E[i] = rank(P, i) - rank(P, i)$

$$depth(v) = E[v]$$

• $E$ is not explicitly stored; it can be computed by the $rank$ index on $P$

```
1 2 3 4 5 6 7 8 9 10 11
P ( ( ) ( ( ) ( ) ) ( ) ) ( ( ) ( ) ) ( )
E 1212343432321232321210
```

```
1 2 3 4 5 6 7 8 9 10 11
P ( ( ) ( ( ) ( ) ) ( ) ) ( ( ) ( ) ) ( )
E 1212343432321232321210
```
Lowest Common Ancestor (lca)

- \( lca = \) lowest common ancestor
- \( u = lca(v,w) \): common ancestor of \( v \) and \( w \) which is furthest from root
- Found in \( O(1) \) time
• $u = \text{parent}(RMQ_E(v,w)+1)$
  – $E$ is the excess array, which represents node depths

$m = RMQ_E(v,w)$: the index of a minimum value in $E[v..w]$ ($RMQ =$ Range Minimum Query)
The Merits and Demerits of BP

• More supported operations
  – subtree size
  – lowest common ancestor (lca)
  – level ancestors

• Succinct indexes are complicated
  – the more supported operations, the more index space
  – o(n) size indexes cannot be ignored in practice
• degree, i-th child, etc. were difficult to implement
• No locality of child labels
DFUDS Representation

- It encodes the degrees of nodes in unary codes in depth-first order
  (DFUDS = Depth First Unary Degree Sequence)
- Degree $d \Rightarrow d$ (’s, followed by a )
- Add a dummy ( at the beginning
- $2n$ bits
- DFUDS is balanced

DFUDS

$U$ ( ( ( ( ( ( ( ) ) ) ) ) ( ( ) ) )

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]
Various Operations on DFUDS

• A node with degree $d$ is represented by the first position of its encoding $\left(^d\right)$
  – Position $v$ in the sequence for a node and its preorder $i$ is converted each other by
  $$v = \text{preorder-select}(i) = (\text{select}_v(i - 1)) + 1$$
  $$i = \text{preorder-rank}(v) = (\text{rank}_v(v - 1)) + 1$$

• Degree: $degree(v) = \text{select}_v(\text{rank}_v(v) + 1) - v$
**i-th child**

\[ \text{child}(v, i) = \text{findclose}(\text{select}_{\text{rank}}(v) + 1 - i) + 1 \]
Parent

\[ \text{parent}(v) = \text{select}(\text{rank}(\text{findopen}(v - 1))) + 1 \]
Number of Descendants (Subtree Size)

- Size of a subtree rooted at \( v \) is

\[
\text{subtreesize}(v) = (\text{findclose}(\text{enclose}(v)) - v)/2 + 1
\]
LCA on DFUDS

- Can be computed by almost the same operation for BP

\[ lca(x,y) = parent(RMQ_E(x,y-1)+1) \]

- Leftmost minimum is used
Tries using DFUDS

• child(v, c)
  
  \[ r = \text{rank}_\sigma(U, v), \quad k = 0 \]
  
  while (\(U[v+k] \neq \) ‘) ) {
    
    if (\(C[r+k] == c\)) return child(v, k+1)
    
    \[ k = k+1 \]
  }

• O(\(\sigma\)) time
  
  O(log \(\sigma\)) is possible

DFUDS

\[
U \begin{pmatrix}
( ( ) ) & ( ( ( ) ) ) & ( ( ) ) \\
\end{pmatrix}
\]

\[
C \begin{pmatrix}
_\_ a c x y z a d \\
\end{pmatrix}
\]
Other Operations 1

- $leaf\text{-}rank(v) = rank_{\text{\textbackslash \textbackslash}}(v)$
- $leaf\text{-}select(i) = select_{\text{\textbackslash \textbackslash}}(i)$
- $preorder\text{-}rank(v) = (rank_{\text{\textbackslash}}(v-1))+1$
- $preorder\text{-}select(i) = (select_{\text{\textbackslash}}(i-1))+1$
Other Operations 2

- $inorder\text{-rank}(v) = leaf\text{-rank}(child(v,2) - 1)$
- $inorder\text{-select}(i) = parent(leaf\text{-select}(i) + 1)$
- $leftmost\text{-leaf}(v) = leaf\text{-select}(leaf\text{-rank}(v-1) + 1)$
- $rightmost\text{-leaf}(v) = findclose(enclose(v))$
The Merits and Demerits of DFUDS

• More supported operations
  – subtree size
  – lowest common ancestor (lca)

• Locality of child labels

• Can be compressed into less than $2n$ bits!

• Succinct indexes are complicated
  – depth
  – level-ancestor
Entropy of Sets

• # of sets $S \subseteq \{1,2,...,n\}$ with cardinality $m$ is $\binom{n}{m}$

• We define the entropy of $S$ as

$$nH_0(S) = m \log \frac{n}{m} + (n - m) \log \frac{n}{n - m} \approx \log \binom{n}{m}$$

• Fully Indexable Dictionary [RRR02]
  – $nH_0(S) + O(n \log \log n / \log n)$ bits
  – member/rank/select: constant time
Entropy of Ordered Trees

• Number of ordered trees having $n_i$ nodes with degree $i$ ($i=0,1,...,n$) is

$$L = \frac{1}{n} \binom{n}{n_0 \quad n_1 \quad \cdots \quad n_{n-1}}$$ [Rote 96]

• The ITLB of trees having the degree sequence is

$$\log L \approx \sum_i n_i \log \frac{n}{n_i}$$

• We define tree degree entropy as $H^*(T) = \sum_i \frac{n_i}{n} \log \frac{n}{n_i}$

• $nH^*(T) \leq 2n + o(n)$ for any tree
Trees with Low Entropy

• A binary ordered tree

\[ \sum_{i=0}^{2} n_i \log \frac{n}{n_i} \leq n \log 3 \approx 1.58n < 2n \]

• Full binary tree (A tree in which all internal nodes have exactly two children)

\[ \frac{n-1}{2} \log \frac{2n}{n-1} + \frac{n+1}{2} \log \frac{2n}{n+1} \approx n < 2n \]

internal nodes
(degree 2)

leaves
(degree 0)
How to Achieve the Entropy

- The tree degree entropy is identical to the order-0 entropy of the degree sequence $S$ of the tree

$$H_0(S) = H^*(T) = \sum_i n_i \log \frac{n}{n_i}$$

- Any log $n$ bits of $U$ are recovered from $S$ in $O(1)$ time. → $U$ can be regarded as uncompressed.

DFUDS $U ( ( ( ) ( ( ( ) ) ) ) ( ( ) ) )$

New $S$ 2 3 0 0 0 2 0 0
Simpler Indexes

• [Geary, Rahman, R. Raman, V. Raman 04, 06]
  – recursive representation
• [Sadakane, Navarro 10]
  – range min-max tree
Data Structure for *findclose*

• Divide the parentheses sequence into blocks of length $B = \frac{1}{2} \log n$
  – $b(p)$: block number containing $p$
  – $\mu(p)$: position of parenthesis matching $p$
  – parenthesis $p$ is said to be far $\iff b(p) \neq b(\mu(p))$

• Far open parenthesis $p$ is said to be opening pioneer $\iff$ For the far open parenthesis $q$ which is immediately precedes $p$, $b(\mu(p)) \neq b(\mu(q))$

• Represent positions of parentheses which match with opening pioneers are represented by 0,1 vector
Lemma: Let $\beta$ denote the number of blocks. Then the number of opening pioneers is at most $2\beta - 3$.

Proof: A graph whose nodes correspond to the blocks and whose edges are $(b(p), b(\mu(p)))$ is an outer-planar graph.

Opening/closing pioneers form a BP again.

$$\beta = n/B = 2n/\log n \Rightarrow \text{Length of BP is } O(n/\log n)$$
Representing Recursive Structure

• opening pioneers and their matching parentheses are represented by a 0,1 vector $B$
  \[ \mu(p) = \text{select}(B, \text{findclose}(P_1, \text{rank}(B, p))) \]

• $B$ is a sparse vector of length $2n$ with $O(n/\log n)$ 1’s
  – Can be represented in $O(n \log \log n/\log n)$ bits

\[ \begin{array}{c}
  P \\
  B \\
\end{array} \begin{array}{c}
  ( ( ( ( )) ) ) \\
  0100 0101 0000 0000 0010 1001 \\
\end{array} \begin{array}{c}
  r \\
  q \\
  p \\
  \mu(p) \mu(q) \mu(r) \\
\end{array} \]
• Let $S(n)$ denote the size of BP representation for an $n$ node tree
  
  $- S(n) = 2n + O(n \log \log n / \log n) + S(O(n / \log n))$

• If the number of nodes becomes $O(n / \log^2 n)$, a naïve data structure which stores all the answers uses only $O(n / \log n)$ bits

• Therefore $S(n) = 2n + O(n \log \log n / \log n)$
Algorithm for findclose

- To compute $\mu(p) = \text{findclose}(P, p)$
- If $p$ is not far, $\mu(p)$ is computed by a table
- Find the pioneer $p^*$ that immediately precedes $p$
- Find $\mu(p^*)$ using the BP for pioneers
- If $p$ is not pioneer, $b(\mu(p)) = b(\mu(p^*))$
- The position of $\mu(p)$ is determined from the difference between depths of $p$ and $p^*$
Range Min-Max Trees

• Existing approach uses a distinct data structure for each operation.
• The more functions, the larger indexes.

• New approach uses basically only one basic data structure (range min-max tree) which supports various operations.
How to implement tree operations

- **Def**
  
  \[
  \text{fwd\_excess}(E, i, d) = \min_{j > i} \{ j \mid E[j] = E[i] + d \}
  \]

  \[
  \text{bwd\_excess}(E, i, d) = \max_{j < i} \{ j \mid E[j] = E[i] + d \}
  \]

- **Lemma**
  
  \[
  \text{findclose}(P, i) = \text{fwd\_excess}(E, i, -1)
  \]
  
  \[
  \text{findopen}(P, i) = \text{bwd\_excess}(E, i, 0) + 1
  \]
  
  \[
  \text{enclose}(P, i) = \text{bwd\_excess}(E, i, -2) + 1
  \]

  \[
  \text{level\_ancestor}(P, i, d) = \text{bwd\_excess}(E, i, d - 1) + 1
  \]

  \[
  \text{enclose} \quad \text{findclose}
  \]

  \[
  P \quad ( ( ) ( ( ( ) ( ) ) ( ) ) ( ( ) ( ) ) ( ) )
  \]

  \[
  E \quad 1212343432321232321210
  \]
Range min-max tree

- Partition excess array $E$ into blocks of length $s$
- Each leaf corresponds to a block, and stores min and max in the block
- Each internal node has $l$ children and stores min and max of the children

$s = l = 3$
Properties of range min-max tree

- Each node corresponds to a range of excess array
- Any interval of excess array is decomposed into $O(lh)$ intervals corresponding to internal nodes, and at most two sub-intervals of leaves ($h$: tree height)
Properties of excess array

- For any $i$, either $E[i+1] = E[i] - 1$ or $E[i] + 1$
- Let $a, b$ be min and max of interval $E[u, v]$. Then in the interval all integers $a \leq e \leq b$ exists.
- Difference between min and max in an interval of length $l$ is at most $l - 1$
How to compute $fwd_{\text{excess}}(E,i,d)$

- Decompose $E[i+1,N-1]$ into sub-intervals
- Scan the intervals from left to right, and find the interval containing $E[i]+d$
- $O( lh+s)$ time
The case sequence is short

• Let $w$ be the length of machine word

• Lemma
  If $N < w^c$, $fwd\_excess$ is computed in $O(c^2)$ time
  Size of data structure is $N + O(Nc/w) + \exp(w)$ bits

• Proof: $l = \Theta(w/\log w)$, $s = \Theta(w)$
  tree height $h = O(c)$. Traverse the tree from root.
  To find the child containing the target, examine $l/c$ children at a time.
  min/max value can be stored in $c \log w$ bits
  $\Rightarrow \Theta(w)$ bits for $l/c$ children
  The target can be found in $O(1)$ time.
How to find lowest common ancestor

- $lca(v, w) = parent(RMQ(v, w) + 1)$
  - $RMQ$: position of min in $E[v, w]$
- $O(1)$ time using range min-max tree
How to compute degree

• Let $E[v,w]$ be the interval for node $v$
• $\text{deg}(v) = (\text{number of min in } E[v+1,w-1])$
• For each node of range min-max tree, store the number of min in the corresponding interval
• Sum up stored $\text{#min}$
For long sequences

• Contract tree, and apply existing data structures
  – remove subtrees with $< w^c$ nodes
  – size of auxiliary data structures can be ignored
• For parent operation, pioneers are used
• For lca, the sparse table algorithm is used
Sparse Table Algorithm

• For each interval \([i, i+2^k-1]\) in array \(B[1,m]\), store the minimum value in \(M[i,k]\).
  \((i = 1, \ldots, m, k = 1, 2, \ldots, \lg m)\)

• For a given query interval \([s, b]\)

  1. Let \(k = \lg(b-s)\)

  2. Compare \(M[s, k]\) and \(M[b-2^k+1, k]\), and output the minimum.

• \(O(1)\) time, \(O(m \lg^2 m)\) bit space
• This data structure is used when the length of \( B \) becomes \( O(n/\lg^3 n) \nabla o(n) \) bit space

• Main theorem: There exists a data structure for an \( n \)-node ordinal tree supporting all operations in \( O(1) \) time using \( 2n + O(n/\log^c n) \) bits for any \( c > 0 \).

• In practice, the range min-max tree for large blocks is enough. This leads to an extremely simple implementation.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>findclose/findopen</td>
<td>position of matching parenthesis</td>
</tr>
<tr>
<td>enclose</td>
<td>position of minimum enclosing parentheses</td>
</tr>
<tr>
<td>preorder rank/postorder rank</td>
<td>preorder/postorder of a node</td>
</tr>
<tr>
<td>inspect(i)</td>
<td>returns P [i]</td>
</tr>
<tr>
<td>isleaf (i)</td>
<td>returns yes if P [i] is leaf</td>
</tr>
<tr>
<td>isancestor(x, y)</td>
<td>returns yes if x is an ancestor of y</td>
</tr>
<tr>
<td>depth</td>
<td>depth of node</td>
</tr>
<tr>
<td>parent</td>
<td>parent node</td>
</tr>
<tr>
<td>first child/last child</td>
<td>first/last child of a node</td>
</tr>
<tr>
<td>next sibling/prev sibling</td>
<td>younger/older brother</td>
</tr>
<tr>
<td>subtree size</td>
<td>number of nodes in a subtree</td>
</tr>
<tr>
<td>level ancestor(u, d)</td>
<td>node v s.t. v is an ancestor of u and depth(v) = depth(u) - d</td>
</tr>
<tr>
<td>level next/level prev</td>
<td>next/prev node in BFS order</td>
</tr>
<tr>
<td>level leftmost/level rightmost</td>
<td>leftmost/rightmost node in given depth</td>
</tr>
<tr>
<td>lca(x, y)</td>
<td>lowest common ancestor of nodes x and y</td>
</tr>
<tr>
<td>deepest node</td>
<td>node with maximum depth in a subtree</td>
</tr>
<tr>
<td>preorder select/postorder select</td>
<td>node with given preorder/postorder (slow)</td>
</tr>
<tr>
<td>degree</td>
<td>number of children (slow)</td>
</tr>
<tr>
<td>child(u, i)</td>
<td>i-th child of node u (slow)</td>
</tr>
<tr>
<td>child rank</td>
<td>number of older brothers (slow)</td>
</tr>
<tr>
<td>Operation</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------------------------------------</td>
</tr>
<tr>
<td>degree</td>
<td>number of children (fast)</td>
</tr>
<tr>
<td>child</td>
<td>i-th child (fast)</td>
</tr>
<tr>
<td>child rank</td>
<td>number of older brothers (fast)</td>
</tr>
<tr>
<td>preorder/postorder select</td>
<td>node with given preorder/postorder (fast)</td>
</tr>
<tr>
<td>inorder rank</td>
<td>inorder of node</td>
</tr>
<tr>
<td>inorder select</td>
<td>node with given inorder (slow)</td>
</tr>
<tr>
<td>inorder select</td>
<td>node with given inorder (fast)</td>
</tr>
<tr>
<td>leaf rank</td>
<td>number of leaves with preorder &lt; i</td>
</tr>
<tr>
<td>leaf select</td>
<td>i-th leaf (slow)</td>
</tr>
<tr>
<td>leftmost leaf</td>
<td>leftmost leaf in a subtree (slow)</td>
</tr>
<tr>
<td>rightmost leaf</td>
<td>rightmost leaf in a subtree (slow)</td>
</tr>
<tr>
<td>leaf select</td>
<td>i-th leaf (fast)</td>
</tr>
<tr>
<td>leftmost leaf</td>
<td>leftmost leaf in a subtree (fast)</td>
</tr>
<tr>
<td>rightmost leaf</td>
<td>rightmost leaf in a subtree (fast)</td>
</tr>
</tbody>
</table>
Practical Implementations

- [Gonzalez, Grabowski, Makinen, Navarro 05]
  - rank, select
- [Kim, Na, Kim, Park 05]
  - rank, select
- [Delpratt, Rahman, R. Raman 06]
  - LOUDS++
- [Okanohara, Sadakane 07]
  - rank, select on compressed vectors
- [Valimaki, Makinen, Gerlach, Dixit 09]
  - compressed suffix trees
- [Navarro 09]
  - Hash based BP
- [Arroyuelo, Canovas, Navarro, Sadakane 10]
  - tries
- [Gog, Ohlebusch 11]
  - compressed suffix trees
- [Brisaboa, Canovas, Claude, Martinez-Prieto, Navarro 11]
  - compressed string dictionaries
- [Grossi, Ottaviano 12]
  - compressed tries through path decomposition
Empirical Comparison

- Size for $n$-node ordered trees
  - LOUDS: $2.10n$ bits
  - BP, DFUDS: $2.37n$ bits
- BP is faster than others for all operations except child($\nu$, $c$) and $i$-th child
- For child($\nu$, $c$) and $i$-th child, LOUDS is the fastest, DFUDS is the next, because of locality of access to the label array.
Comparison with Non-Succinct Trees

• Non-succinct trees are 12-20 times faster than LOUDS for $i$-th child.
• Non-succinct trees supporting only $i$-th child are 13 times larger than BP, 15 times larger than LOUDS.
• Trees supporting parent, depth, subtreesize, preorder use 66 times larger space than BP.
• Supporting lca, level_ancestor uses more space…

• Succinct trees support a wide set of functions using small space.
Heavy Path Decomposition

• A rooted tree is decomposed into heavy paths so that any leaf-to-root path consists of at most $O(\log n)$ heavy paths.

• Each heavy path is stored consecutively.
  – Good locality of reference
  – BP based on Range Min-Max Tree is used.

• Fast string dictionaries
Concluding Remarks

• Succinct trees are practical now.
  – BP supports various operations using small space.
  – LOUDS is fast for child operations.
  – Heavy path decomposition is effective for string dictionaries.

• Future work: practical implementations for dynamic labeled trees.
  – Range Min-Max Tree is easy to dynamize.