Succinct Data Structures

Exploring succinct trees in theory and practice

Sam Heilbron
May 12, 2017

Problem Background

Data structures are used to organize and store information in order to efficiently interact with the data. Most data structures are compared by the efficiency of the operations that can be performed. What is the theoretical time complexity to insert an element? Delete an element? Find an element? Merge two structures? In order to complete these operations effectively, additional space may be used. Since the focus of these structures is the speed of the operations that can be performed, this space is considered “free”. But is it really?

There continues to be a massive increase both in size and availability of large data sets that are processed by various applications. Massive data sets present a challenge to balance storage, organization and accessibility. The underlying problem is: *how can I compress the data but still query it quickly?*

Lossless data compression algorithms successfully compress data and allow it to be perfectly reconstructed from the compressed data. However, in order to interact with the data, it must first be decompressed. What if there were a way to store the data in a compressed format, and be able to interact with it, without first having to decompress it? Succinct data structures make this possible.

Succinct data structures require the amount of space that is close to the information-theoretic lower bound [A]. That is, there is very little “extra space”. As long as the data can be used efficiently, less space is desirable because less space implies faster and cheaper algorithms. For example, they allow a data representation which would otherwise need to be written to disk to instead be fit into main memory which improves access time. There exist a range of real-world applications that can utilize this space-efficient storage method, most centered around information retrieval:

Search Engines- Search engines can index billions of web pages and respond to queries about those pages in real-time. Therefore, it is crucial to decrease the space used by their index while still allowing efficient queries.

Mobile applications - Mobile applications have a limited amount of storage available on the device and efficient storage allows for added functionality.

DNA representation - Medical databases are massive and contain sequences that contain patterns and need to be queried quickly. One can successfully represent DNA in a succinct manner.

Streaming environments - In a streaming environment, the next frame(s) need to be accessed quickly and efficiently. The smaller the size of the content, the faster it can be processed, and at a smaller cost.

Problem Statement

We analyze the effectiveness of succinct trees in theory and then compare those major techniques for representing trees succinctly in practice. The pointer form of an n-node tree requires $\Omega(n \log n)$ bits (see Labeled Trees section). However, according to Information Theory [A,B] only $2n - \theta(\log n)$ bits are required to distinguish n nodes. A lot of research [1, 2, 4, 5, 6] has focused on succinct representations which require just 2n + o(n) bits while still being able to implement sophisticated operations in constant time. There are three methods for implementing succinct trees: balanced parentheses (**BP**), depth first unary degree sequence (**DFUDS**) and level-ordered unary degree sequence (**LOUDS**).

For each method, we will **(1)** provide a high level description, **(2)** outline the lower level details related to bit storage and queries, **(3)** detail the operations available on this representation, and **(4)** illustrate the theoretical time complexities of operations.

Only after a strong foundation is achieved will we then compare the effectiveness of these storage methods in practice. These results are based on the results of [6].

What does it mean to be succinct?

A succinct representation of data is one whose size is roughly the information-theoretic lower bound [A,B]. That is, there is little “extra space”. There are **three** categories for succinct data structures:

Let *OPT* refer to the optimum number of bits. See [7] for reference about Big-O notation.

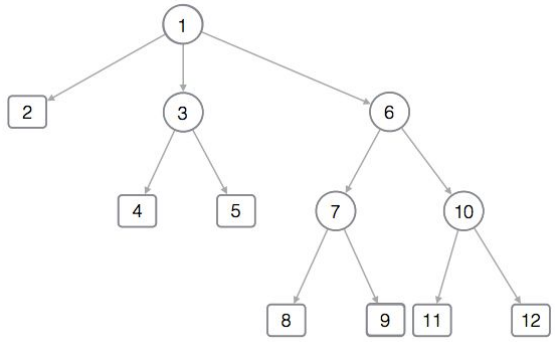
Implicit: *OPT + o(1) bits*. The constant factor is added to round up the number of bits. For an implicit data structure, the structural information is implicit in the way that the data is stored. An example of this is a an array since the length is the only piece of overhead not implicit to the structure. Generally, it is extremely challenging to achieve an implicit data structure that contains a robust suite of operations that run in constant time.

Compact: *O(OPT) bits*. This is easier to achieve, and acts more as a warmup for succinct data structures.

Succinct: *OPT + o(OPT) bits*. **This is what we will focus on.**

Labeled Trees

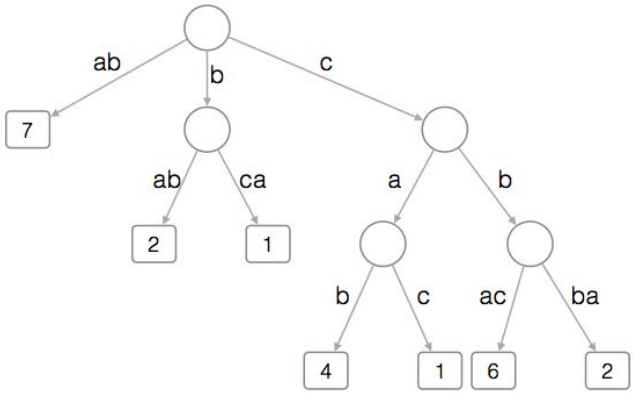
First let's examine the space complexity of trees in pointer form.



Above is an n-node (n = 12) tree in pointer form. A general tree of n nodes requires $O(nw)$ bits of space where $w \geq log_2 n$ is the bit length of a machine pointer. Therefore, the space complexity is $O(n\ log\ n)$. Now let's examine an n-node labeled tree (trie) which represents the following dictionary, D, where D contains strings in the alphabet σ:

$$D = \{ab\ (x7),\ bab\ (x2),\ bca\ (x1),\ cab\ (x4),\ cac\ (x1),\ cbac\ (x6),\ cbba\ (x2)\ \}$$

$$\sigma = \{a...z\}$$

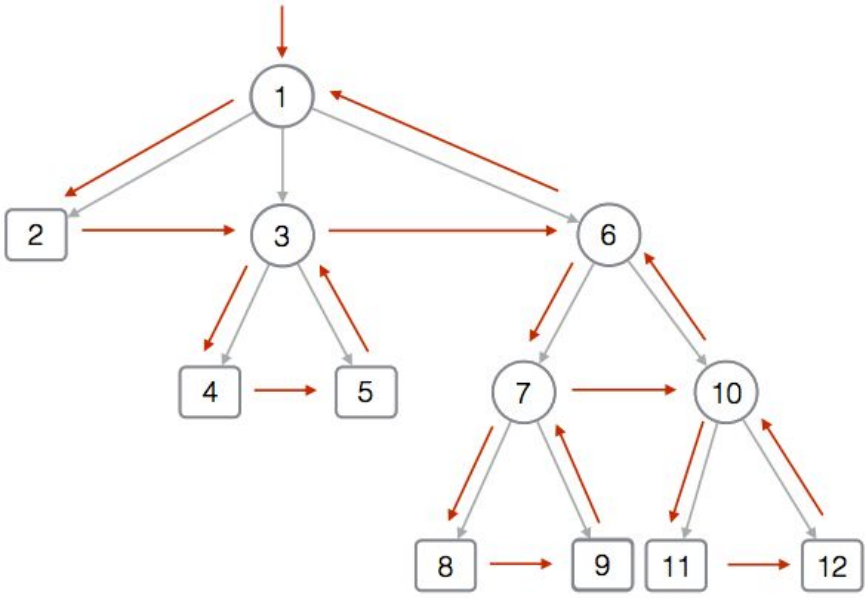


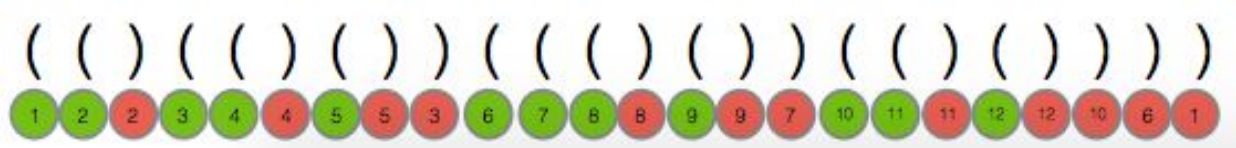
In this case, each child of a node is associated with a distinct label in a range [1, σ]. A general tree of n nodes requires $O(nw)$ bits of space where $w \geq log_2 \sigma$ is the bit length of a machine pointer. In this case the tree is compressed, which means that chains of redundant nodes (nodes with only one child) are joined. For example, normally the leftmost edge, ab, would be two separate edges, a and b, joined in sequence. The values at the leaves represent the number of strings that end in that sequence. For example, in the above word list, the word "cab" appears 4 times.

Succinct Representation of Labeled Trees

Balanced Parentheses (BP)

The balanced parentheses representation was first advocated by Jacobson in 1989 [1] and then expanded upon by Munro and Raman in 2001 [2]. The premise is that while the tree is traversed in depth-first preorder traversal, an open parenthesis is written when a node is reached the first time, and a closing parenthesis is written when that node is reached again. This results in a sequence of 2n balanced parentheses, such that each node is represented by a pair of matching parentheses.





A node is identified by its opening parenthesis and a subtree is the collection of parentheses contained by a node's opening and closing parentheses. For a nice visualization of tree traversals, visit:
https://www.khanacademy.org/computer-programming/depth-first-traversals-of-binary-trees/934024358

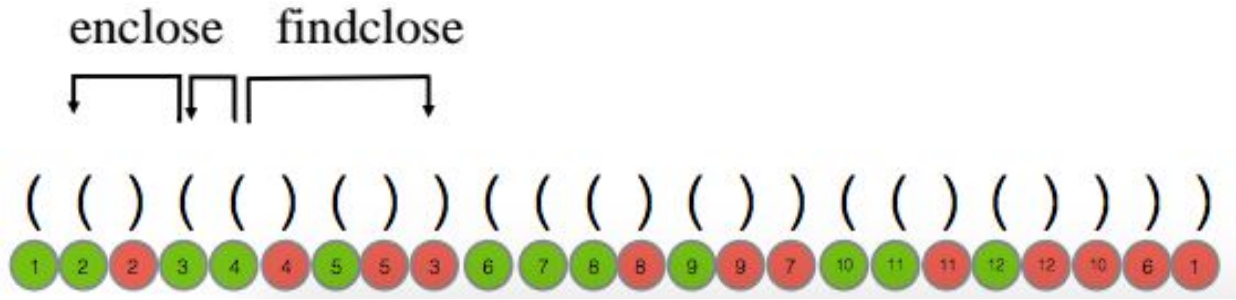
Basic Operations:

These are core operations which are used to perform tree navigation operations. Let S be the string of parentheses.

findclose(S, i): return the position of ")" matching with "(" at S[i]. This finds the index in S, of the closing parenthesis for a node whose opening parenthesis is at position i.

findopen(S, i): return the position of "(" matching with ")" at S[i]. This finds the index in S, of the opening parenthesis for a node whose closing parenthesis is at position i.

enclose(S, i): return the position of "(" which encloses "(" at S[i], This finds the index in S, of an opening parenthesis for the parent of a node whose opening parenthesis is at position i.



rank$_p$(S, i): return the frequency of pattern p in S[1..i]. For example, rank$_($(S, 5) returns the number of open parentheses in the first 5 characters in S, which, in the above picture, is 4.

select$_p$(S, i): return the position of the ith occurrence of pattern p in S. For example, select$_($(S, 5) returns the index where the 5th open parenthesis is found, which, in the above picture, is 7.

Tree Navigation Operations:

parent(v) = enclose(S,v). The parent of a node is just the position of the tightest opening parenthesis that encloses v.

firstchild(v) = v+1. The first child of a node is the opening parenthesis directly following v.

sibling(v) = findclose(S,v) + 1. The right sibling of a node is the node directly after the closing parenthesis of node v.

lastchild(v) = findopen(S, findclose(S,v) - 1). findclose(S,v) - 1 finds the closing parenthesis for the last child of v. Calling findopen on this returns the index (opening parenthesis) for that node.

subtreesize(v) = (findclose(S,v) - v + 1)/2. The subtree is calculated by finding the bounding parenthesis, adjusting it by v (the offset of the opening parenthesis) and then dividing by 2 since 2n parentheses represent n nodes.

degree(v) = iteratively call findclose on all the children. The time complexity of this operation is relative to the number of children that v has.

depth(v) = rank$_($(S, i) - rank$_)$(S, i). The depth of a node is the number of opening parentheses minus the number of closed ones. You can think of it as each open parenthesis is a level down the tree and each closed one is a level up.

lca(v,w) = return the lowest common ancestor of v and w which is furthest from the root. This is calculated using the largest depth that is less than (or equal to) the depth of the two children. Research Range Minimum Query (RMQ) [3] for more information about how to do this.

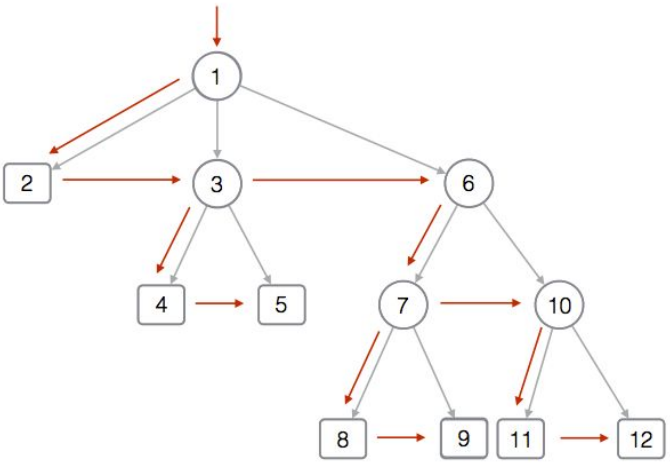
If $rank_p(S, i)$ and $select_p(S, i)$ can be calculated in constant time [O(1)] then each of the operations (except for degree) can also be calculated in constant time since they depend on finding the rank and select of indices. See the appendix [D] for a description of how rank and select are calculated in constant time while maintaining o(n) space.

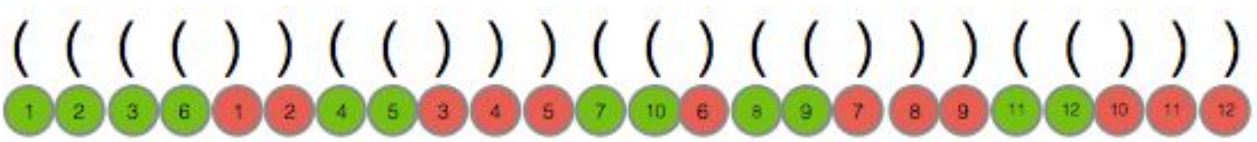
Balanced Parenthesis Overview:

BP contains more supported operations (in constant time) including calculating the subtree size, the lowest common ancestor and level ancestors. However, degree and i^{th} child are difficult to implement. Additionally, the more supported operations that exist, the more index space is required. In theory o(n) sized indices is ok but in practice this cannot be ignored.

Depth-First Unary Degree Sequence (DFUDS)

The depth-first unary degree sequence was first advocated by Benoit, Demain, Munro, Raman, Raman and Rao in 1999 [5]. The premise is that the tree follows the same traversal as BP. However, at each node, i opening parentheses are appended, for i children, and 1 closing parenthesis is appended. This results in a sequence of 2n balanced parentheses, such that each node is represented by a pair of matching parentheses. A node is identified by the position where the parentheses start.





Basic Operations:

These are core operations which are used to perform tree navigation operations. Let S be the string of parentheses.

findclose(S, i): return the position of ")" matching with "(" at S[i]. This finds the index in S, of the closing parenthesis for a node whose opening parenthesis is at position i.

findopen(S, i): return the position of "(" matching with ")" at S[i]. This finds the index in S, of the opening parenthesis for a node whose closing parenthesis is at position i.

enclose(S, i): return the position of "(" which encloses "(" at S[i], This finds the index in S, of an opening parenthesis for the parent of a node whose opening parenthesis is at position i.

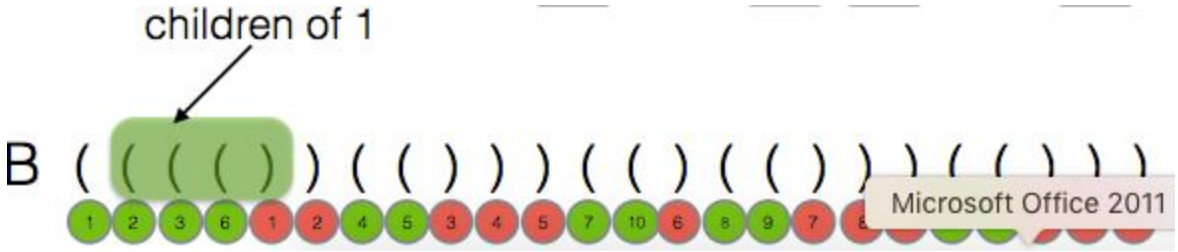
$rank_p$(S, i): return the frequency of pattern p in S[1..i]. For example, $rank_1$(S, 5) returns the number of 1s in the first 5 characters in S, which, in the above picture, is 4.

$select_p$(S, i): return the position of the i^{th} occurrence of pattern p in S. For example, $select_1$(S, 5) returns the index where the 5^{th} 1 is found, which, in the above picture, is 7.

Tree Navigation Operations:

degree(v) = $select_)(rank_)(v) + 1) - v$. Find the index of the node after the number of closing parentheses. Then offset this by v to find the acutal degree.

child(v,i) = $findclose(select_)(rank_)(v) + 1) - i) + 1$. Get 1 more than the number of closing parentheses, then find the index of that number of closing indices offset by 1. Finding the closing parenthesis of that value with give you the node before the i^{th} child so offset by 1.



parent(v) = $select_)(rank_)(findopen(v-1))) + 1$. Similar to child, but just in reverse order.

subtreesize(v) = (findclose(enclose(v)) - v)/2 + 1. First find the closing parenthesis, then divide the difference of the indices by 2 since 2n bits holds n nodes.

lca(v,w) = return the lowest common ancestor of v and w which is furthest from the root. This is calculated using the largest depth that is less than (or equal to) the depth of the two children. Research Range Minimum Query (RMQ) [3] for more information about how to do this.

leaf-rank(v) = $rank_))(v)$. Find the number of leaves to the left of leaf v.

leaf-select(i) =select))(i). Find the i^{th} leaf.

preorder-rank(v) = $rank_)(v-1) + 1$. Find the preorder rank of v.

preorder-select(i) = $select_)(i-1) + 1$. Find the node with preorder rank i.

inorder-rank(v) = leaf-rank(child(v,2) - 1). Find the inorder rank of v.

inorder-select(i) = parent(leaf-select(i) + 1). Find the node with inorder rank i.

leftmost-leaf(v) = leaf-select(leaf-rank(v-1) + 1). Find the leftmost leave of node v.

rightmost-leaf(v) = findclose(enclose(v)). Find the rightmost leave of node v.

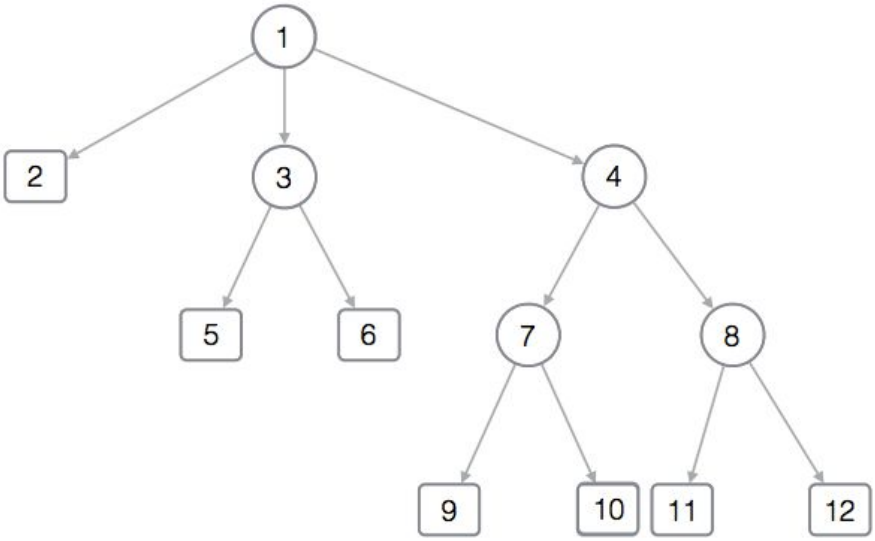
If $rank_p(S, i)$ and $select_p(S, i)$ can be calculated in constant time [O(1)] then each of the operations (except for degree) can also be calculated in constant time since they depend on finding the rank and select of indices. See the appendix [D] for a description of how rank and select are calculated in constant time while maintaining o(n) space.

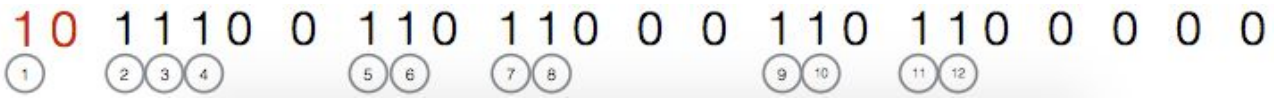
Depth-First Unary Degree Sequence Overview:

DFUDS contains more supported operations (in constant time) including calculating the subtree size and the lowest common ancestor. Additionally, child labels are localized in the bit string. In fact, this can be further compressed into less than 2n bits (I need to do more research needed to defend this). However, depth and level ancestors are challenging to compute.

Level-Ordered Unary Degree Sequence (LOUDS)

The level-ordered unary degree sequence was first advocated by Jacobson in 1989 [1]. The premise is that the tree is traversed in level order. At each node, 1^d0 is appended to the bit string where d is the degree of the node. This results in a sequence of 2n+1 bits (n 1s and n+1 0s). A node, n, is identified by the position where n^{th} 1 is in the string.





Basic Operations:

These are core operations which are used to perform tree navigation operations. Let S be the string of bits.

$rank_p$(S, i): return the frequency of pattern p in S[1..i]. For example, $rank_1$(S, 5) returns the number of 1s in the first 5 characters in S, which, in the above picture, is 4.

$select_p$(S, i): return the position of the i^{th} occurrence of pattern p in S. For example, $select_1$(S, 5) returns the index where the 5^{th} 1 is found, which, in the above picture, is 8.

Tree Navigation Operations:

firstchild(x) = $select_0(rank_1(S,x)) + 1$. $rank_1(S,x)$ finds which node x is (in level order) and then $select_0()$ goes that many 0's deep in the string, which is the 0 right before the child. The offset of 1 ensures that the first child is returned. If this returns 0, then return -1 else return the result.

lastchild(x) = $select_0(rank_1(S,x) + 1))$. $rank_1(S,x) + 1$ goes to the node following x, then $select_0()$ visits the firstchild of that next node. The offset of -1 ensure we go back in the string 1, which will be the last child of x. If this returns 0, then return -1 else return the result.

rightsibling(x) = if S[x+1] == 0 then -1 else x+1. Siblings are represented as 1's next to each other in the bit string so if a 0 follows x, then it is the last child. Otherwise, the rightsibling is in the following index.

parent(x) = $select_1(rank_0(S,x))$. rank0(S,x) finds the number of "clumps" and then rank1() goes to the position of that clump.

degree(x) = lastchild(x) - firstchild(x) + 1.

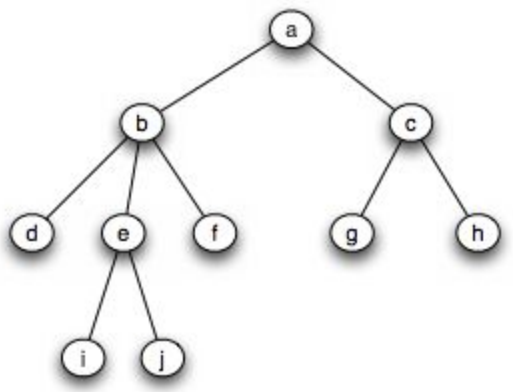
If $rank_p$(S, i) and $select_p$(S, i) can be calculated in constant time [O(1)] then each of the operations can also be calculated in constant time since they depend on finding the rank and select of indices. See the appendix [D] for a description of how rank and select are calculated in constant time while maintaining o(n) space.

Level-Ordered Unary Degree Sequence Overview:

LOUDS is implemented only by rank and select which makes it easier to implement. However, you cannot comput subtree sizes so there is less functionality than can be computed in constant time.

Visual Comparisons of Trees

Below is a visual comparison of the BP, DFUDS and LOUDS sequences for the following tree:



LOUDS sequence: 1 1 0 1 1 1 0 1 1 0 1 0 1 1 0 0 0 0 0 0.

	a	b	c	d	e	f	g	h
LOUDS:	1 1 0	1 1 1 0	1 1 0	1 0	1 1 0	0	0	0

BP sequence: ((( ) (( ) ( )) ( )) (( ) ( ))).

	a	b	d	d	e	i	i	j	j	e	f	f	b	c	g	g	h	h	c	a
BP	(	(	(	)	(	(	)	(	)	)	(	)	)	(	(	)	(	)	)	)

DFUDS sequence: (( ) ((( )) (( )))) (( ))).

	a	b	d	e	i	j	f	c	g	h
DFUDS:	(( )	((( )	)	(( )	)	)	)	(( )	)	)

Succinct Tree Comparison In Practice

When trees are built using their succinct representation and ran on an Intel(R) Core(™)2 Duo processor at 3.16 GHz, with 8GB main memory and 6 MB of cache, running version 2.6.24-24 of Linux kernel, the following results are determined:

A variation of the BP representation, which includes a novel data structure to handle core operations, called a range min-max ree, offers the best combination of space usage, time performance and functionality.

LOUDS can implement a reduced set of navigation operations using just 5% extra space. In fact, it is the fastest choice in several operations.

DFUDS and LOUDS provide the best representations for navigating to a child quickly.

Additional Resources

There are a range of resources available if you're interested in learning further about the material. In addition to the papers that are referenced here, I recommend checking out the following resources:

Erik Demaine lectures at MIT (Lectures 16,17, 18):
https://courses.csail.mit.edu/6.851/spring12/lectures/

Slide Decks with detailed descriptions:
https://cs.uwaterloo.ca/~imunro/cs840/Notes16/SuccinctDS.pdf
https://link.springer.com/chapter/10.1007/3-540-44634-6_39
http://pages.di.unipi.it/rossano/wp-content/uploads/sites/7/2016/07/Slide.pdf

Succinct Data Structure Library (C++):
https://github.com/simongog/sdsl-lite

Appendix

[A] **Information Theoretic Lower Bound**

This examines the lower bounds on a problem or a class of algorithms that can be represented as a binary tree where each decision is a branch and edges are straight line iterations. Therefore, the possible behaviours form a tree, along which each path the algorithms terminated with some output (at the leaf). This tree is not necessarily balanced.

Take for example sorting algorithms. To determine the lower bound, you need to ask a few questions:

1. *What is the minimum number of leaves any such tree must have?*
    a. $n!$, since that is the number of possible combinations of sorting n numbers
2. *If a binary tree has K leaves, what is its height?*
    a. $ceiling\ (log_2 K)$

Therefore, any sorting tree has to have depth d, $d \geq log(n!)$. Since $log(n!) = \theta(n\ log\ n)$, we've shown that for any algorithm that fits into that framework, it must have a depth of $\theta(n\ log\ n)$ which means that the required number of operations is $\theta(n\ log\ n)$.

For more resources about Information Theoretic Lower Bounds check out the following UC Davis lecture video (https://www.youtube.com/watch?v=Ws7EYLT43u4) or wikipedia page (https://en.wikipedia.org/wiki/Information_theory).

[B] **Space Lower Bounds**

Space lower bounds on the required number of bits to represent each class of trees is obtained via information theory[A] by counting the number of trees in the class. For example, the

number of n-node binary trees is known to be the nth Catalan number[C]. Therefore, an encoding of binary trees with n nodes requires at least $log(C_n) = 2n - logn + O(1)$ number of bits.

[C] **Catalan Number**

The Catalan Numbers are a sequence of positive integers that enumerate combinatorial structures of different types. These vary from nonnegative paths in a plane, to triangulations of a convex polygon to full binary trees. Using 0-based numbering, the n^{th} Catalan number is defined by:

$$C_n = \frac{1}{n+1} C_{2n}^n = \frac{(2n)!}{(n+1)!n!} = \prod_{k=2}^{n} \frac{n+k}{k}$$

Asymptotically, Catalan numbers grow:

$$C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

[D] **Calculating Rank and Select in Constant Time**

The attached slideshow (S.D.S - Rank and Select.pdf) gives a brief overview of how Rank and Select can be stored in o(n) space and calculated in O(1) time.

Reference

[1] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th FOCS*, pages 549-554,1989

[2] I. Munro, V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31 (3): 762-776, 2001

[3] https://en.wikipedia.org/wiki/Range_minimum_query

[4] R. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proc. 15th SODA*, pages 1-10, 2004

[5] D.Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, and S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275-292, 2005

[6] Arroyuelo, D., C´anovas, R., Navarro, G., Sadakane, K.: Succinct trees in practice. In: Blelloch, G.E., Halperin, D. (eds.) ALENEX. pp. 84–97. SIAM (2010)

[7]https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation