SUCCINCT TREE

November 9, 2022

Nakul Alawadhi (2021csb1111) ,
Vikalp Dhalwal (2021csb1140) ,
Shobhit Juglan (2021csb1133)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Akanksha

Summary: We have implemented Succinct Tree in 3 different methods. 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". There are three methods for implementing succinct trees: balanced parentheses (BP), depth first unary degree sequence (DFUDS) and level-ordered unary degree sequence (LOUDS).

1. Introduction

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. In order to complete these operations effectively, additional space may be used.

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

In this project, we have implemented succinct data structure, that represents a given tree in succinct encoded form.
Succinct - Data - Structure : A data structure is called succinct if it uses an amount of space that uses nearly the theoretical lower wound of space with lower order additive terms, and still supports the desired operations.

Succinct representation of tree :
We have implemented three methods for succinct representation of tree :
• balanced parentheses (BP)
• depth first unary degree sequence (DFUDS)
• level-ordered unary degree sequence (LOUDS)

The pointer form of an n-node tree requires (n log n) bits. However, according to Information Theory only 2n (log n) bits are required to distinguish n nodes. A lot of research has focused on succinct representations which require just 2n + o(n) bits while still being able to implement sophisticated operations in constant time.

2. Succinct Representation of Trees

2.1. 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.

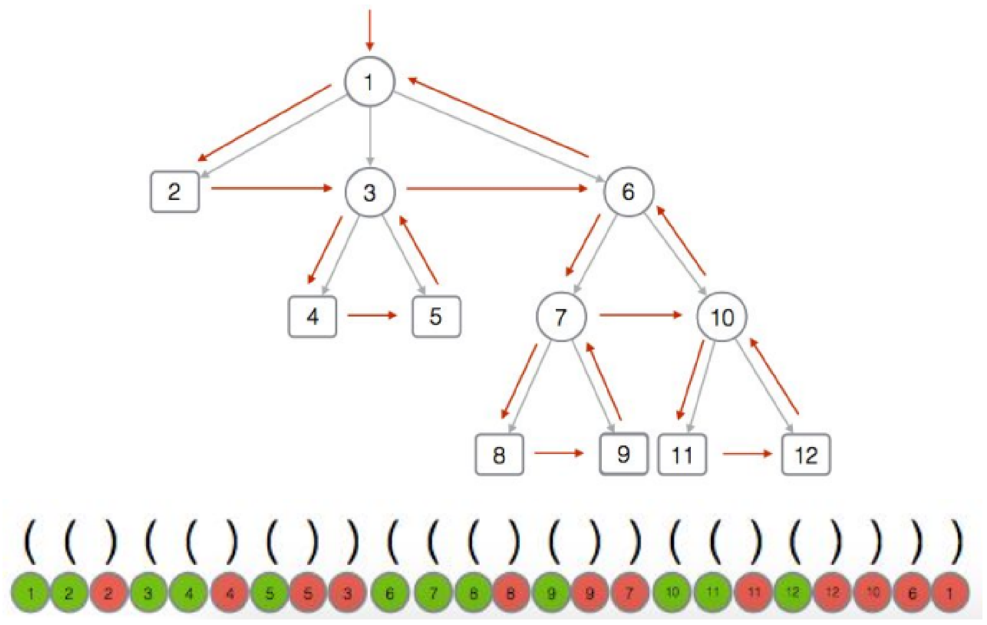


Figure 1: Representing tree as balanced parentheses.

Operations :

• parent bp(v) : This function is used to find the parent of node whose opening parentheses is at position at v. It uses the fact that parent of node is the node corresponding to the opening parentheses that just encloses it.
Time complexity : o(number of nodes)

• f irst child bp(v) : This function returns the index of parentheses corresponding to the first child of node whose parentheses is at v. It exploits the fact that if a vertex has child then all the parentheses corresponding the child lies between the opening and closing parentheses of the give node. Further, first child corresponds to the immediately next opening parentheses after opening parentheses of the node.
Time complexity : o(number of nodes)

• last child bp(v) : This function returns the index of parentheses corresponding to the last child of node whose parentheses is at v. If exist, the last child corresponds to the node whose closing parentheses is just immediately before the closing parentheses of the given node whose opening parentheses is at v.
Time complexity : o(number of nodes)

• sibling(v) : It returns the index of the parentheses corresponding to the sibling of node, if it exists. Sibling of a node is either the node corresponding to the parentheses(say m) just after the closing parentheses of the v, provided m is '('. Else it the node corresponding to the parentheses(say n) just before v, provided n is ')'.
Time complexity : o(number of nodes)

• subtreesize bp(v) : The sub-tree size is calculated by finding the bounding parenthesis of v, adjusting it by v and then dividing by 2.
Time complexity : o(number of nodes)

• degree bp(v) : This function returns the number of children of the node corresponding to v.
Time complexity : O(number of children)

• depth bp(v) : Depth is the length of the path from root to that node. This function returns the depth

of node corresponding to the parentheses at v. It can be calculated as rank of opening parentheses of v - rank of closing parentheses of v
Time complexity : o(number of nodes)

2.2. Depth-First Unary Degree Sequence (DFUDS)

The depth-first unary degree sequence was given in 1999 by Benoit, Demain, Munro, Raman, Raman and Rao. Here we follow depth first traversal for each node. At each node we first append i number of opening parentheses where i is the number of children and then 1 closing parenthesis. This way we use 2n parentheses to depict the entire tree. We use opening paranthesis to identify each node.

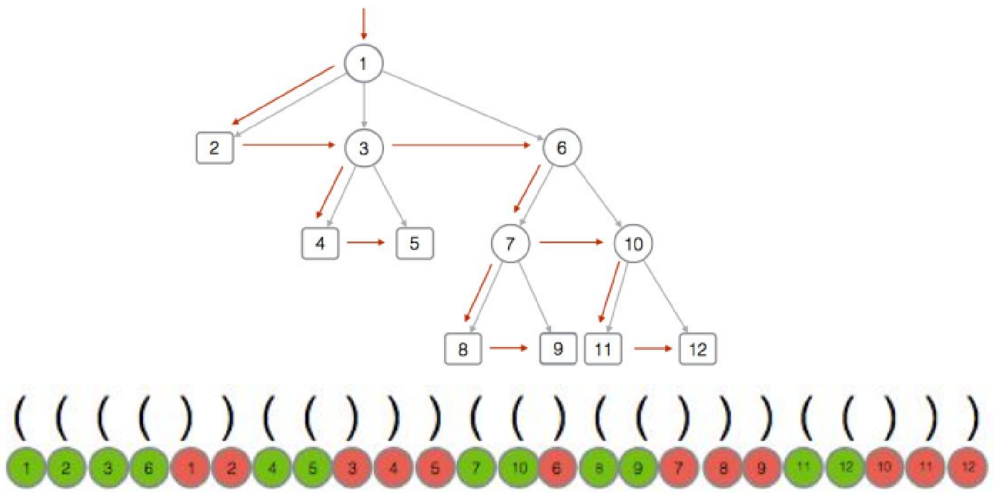


Figure 2: Representing tree as DFUDS.

Operations :

• parentdfuds(v) : This function is used to find the parent of node whose opening parentheses is at position at v. It uses the fact that parent of node is the node corresponding to the first closing parentheses after the node.
Time complexity : O(number of nodes)

• first child dfuds(v) : This function returns the index of parentheses corresponding to the first child of node whose parentheses is at v.
Time complexity : O(number of nodes)

• last child dfuds(v) : This function returns the index of parentheses corresponding to the last child of node whose parentheses is at v. If exist, the last child corresponds to the node whose opening parenthesis is just immediately before the closing parenthesis of the given node whose opening parenthesis is at v.
Time complexity : O(number of nodes)

• degree dfuds(v) : This function returns the number of children of the node corresponding to v.
Time complexity : O(number of nodes)

• leaf rank dfuds(v) : It returns the number of leaves before the given node which is same as the number of occurrences of pattern "))" before the closing parenthesis of the node.
Time complexity : O(number of nodes)

• leaf select dfuds(i) : It returns the index of the ith leaf node which is same as the ith occurrence of the pattern "))".
Time complexity : O(number of nodes

2.3. Level-Ordered Unary Degree Sequence (LOUDS)

In this program we focus on the practical performance the fundamental Level-Order Unary Degree Sequence (LOUDS) representation [Jacobson, Proc. 30th FOCS, 549–554, 1989]. The premise is that the tree is traversed in level order. At each node 111..(d times)0 is appended to the string where d is the number of children of the node. To traverse the tree in the level order we heave used Breadth-First Search. At start, bit 10 is appended to the empty string to accommodate for the 1st node which is traversed from an imaginary super node. Then further traversal is carried using BFS. This results in a sequence of 2n+1 bits (n 1s and n+1 0s). A node, n, is identified by the position where nth 1 is in the string. We have used ones based numbering to represent the string.The numbering procedure is explained below. Ones-based numbering: Numbering the i th node in level-order by the position of the i th 1 bit and at the end of the corresponding level we append 0 to the string. This gives a node a number from 1, ..., 2n + 1. Number of 1 bits =n and number of 0 bits = n+1.

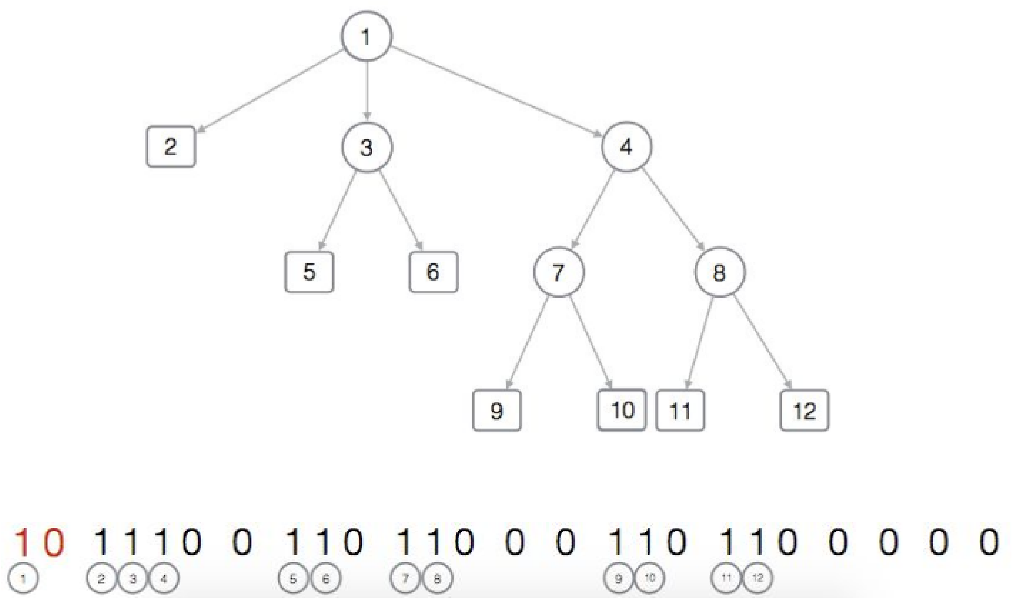


Figure 3: Representing tree as LOUDS.

Operations :

• parent lds(x) : Algorithm = select1(rank0(S,x)). rank0(S,x) finds the number of "clumps" and then rank1() goes to the position of that clump.This gives index of the parent of the corresponding to the current node accessed through bit string stored.
Time complexity : O(n)

• first child lds(x) : rank1(S,x) finds which node x is (in level order) and then select0 () 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.
Time complexity : O(n)

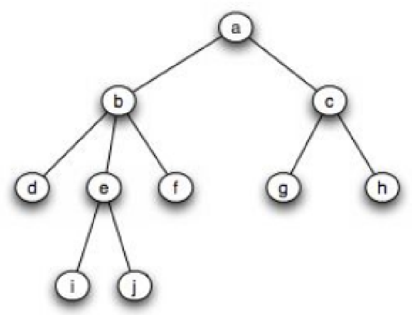
• last child lds(x) :rank1 (S,x) + 1 goes to the node following x, then select0 () visits the first child of that next node. The offset of -1 ensure we go back in the string 1, which will be the last child of x. If this returns 0, then return -1 else return the result.
Time complexity : O(n)

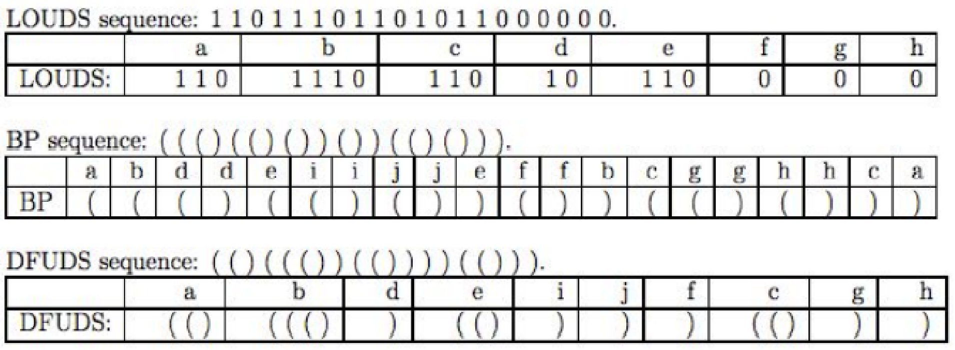
• rightsibling(x) : urn -1 else return the result.Right sibling(x) = if S[x+1] == 0 then -1 else x+1. Siblings are represented as 1's next to each other in the bit string so if a 0 follows x, then it is the last child. Otherwise, the right sibling is in the following index.
Time complexity : O(n)

• degree lds(v) : The degree of the node is equal to the number of children it has. It is calculated using the formula degree(x) = lastchild(x) firstchild(x) + 1
Time complexity : O(n)

2.4. 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:	(( )	((( )	)	(( )	)	)	)	(( )	)	)

Figure 4: Traversal Comparison among 3 Representation.

Succinct Tree Comparison
LOUDS can implement a reduced set of navigation operations using just 5 percent 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.

3. Algorithms

3.1. Rank and Select

Rank and Select algorithms have been implemented using Z-algorithm.

Algorithm 1 Rank (char* pattern, int size, int i)

```
1: integer p ← 0 and c ← 0
2: integer l ← size + 1 + i
3: allocate memory for Z  array ← malloc(l  sizeof(int))
4: allocate memory for new  string ← malloc((l + 1)  sizeof(char))
5: create new  string ← pattern + "A" + parentheses  string
6: a ← 0 and b ← 0
7: for  $j = 1tol$ do
8:   if $j > b$ then
9:     a ← j and b ← j
10:     while $newstring[ba] == newstring[b]b < l$ do
11:       increment b
12:     end while
13:     Z  array[j] ← b  a
14:     increment b
15:   end if
16: else
17:   k getsj  a;
18:   if $Zarray[k] < bi + 1$ then
19:     Z  array[j] ← Z  array[k];
20:   end if
21: else
22:   a ← j;
23:   while $newstring[ba] == newstring[b]b < l$ do
24:     increment b
25:   end while
26:   Z  array[j] ← b  a
27:   increment b
28: end for
29: for $p = 1tol - 1$ do
30:   if $Zarray[p] == size$ then
31:     increment c
32:   end if
33:   free(Z-array)
34:   free(new-string)
35:   return c
36: end for
```

Algorithm 2 Select (char* pattern, int size, int i)

```
integer p ← 0
integer l ← size + 1 + i
allocate memory for Z  array ← malloc(l  sizeof(int))
allocate memory for new  string ← malloc((l + 1)  sizeof(char))
create new  string ← pattern + ”A” + parentheses  string
store value of node and its corresponding ’(’ parenthesis in bp[count]
a ← 0andb ← 0
for $j = 1toldo$ do
    if $j > b$ then
        a ← jandb ← j
        while $newstring[ba] == newstring[b]ANDb < l$ do
            increment b
        end while
        Z  array[j] ← b  a
        increment b
    end if
else
    k ← j  a
    if $Zarray[k] < bi + 1$ then
        Z  array[j] ← Z  array[k]
    end if
else
    a=j;
    while $newstring[ba] == newstring[b]ANDb < l$ do
        increment b
    end while
    Z  array[j] ← b  a
    increment b
end for
for $p = 1tol1$ do
    if $Zarray[p] == size$ then
        increment c
    end if
    if $c == i$ then
        free(Z-array)
        free(new-string)
        return p-size
    end if
    free(Z-array)
    free(new-string)
    return -1
end for
```

3.2. Balanced Parentheses (BP)

Algorithm 3 balanced - parenthesis

```
start from root of the tree(r)
store value of node and its corresponding '(' parenthesis in bp[count]
increment count
for $foreachchildofnode(v)$ do
    balanced  parentheses(v)
end for
store value of node and its corresponding ')' parenthesis in bp[count]
increment count
```

Algorithm 4 find close bp (i)

```
store value of node at bp[i] in some variable temp(say)
for $forj = i + 1to2numberofnodes$ do
    if $valueofnodeatbp[j]isequaltotemp$ then
        return j
    end if
end for
```

Algorithm 5 find open bp (i)

```
store value of node at bp[i] in some variable temp(say)
for $forj¸i1to0$ do
    if $valueofnodeatbp[j]isequaltotemp$ then
        return j
    end if
end for
```

Algorithm 6 enclose bp (i)

```
count $\leftarrow$ 0
for $forj¸i1to0$ do
    if $parenthesesatbp[j]is($ then
        count $\leftarrow$ count + 1
    end if
    if $parenthesesatbp[j]is)$ then
        count $\leftarrow$ count  1
    end if
end for
if $count = 1$ then
    return j
end if
```

3.3. Depth-first Unary Degree Sequence (DFUDS)

Algorithm 7 depth - first

```
 1: start from root of the tree(r)
 2: for $foreachchildofnode(v)$ do
 3:     store value of node and its corresponding '(' parenthesis in dfuds[count]
 4:     increment count
 5: end for
 6: store value of node and its corresponding ')' parenthesis in dfuds[count]
 7: increment count
 8: for $foreachchildofnode(v)$ do
 9:     depth  first(v)
10: end for
```

Algorithm 8 find close dfuds (i)

```
1: store value of node at dfuds[i] in some variable temp(say)
2: for $forj = i + 1to2numberofnodes$ do
3:     if value of node at dfuds[j] is equal to temp then
4:         return j
5:     end if
6: end for
```

Algorithm 9 find open dfuds (i)

```
1: store value of node at dfuds[i] in some variable temp(say)
2: for $forj¸i1to0$ do
3:     if value of node at dfuds[j] is equal to temp then
4:         return j
5:     end if
6: end for
```

Algorithm 10 enclose dfuds (i)

```
1: count $\leftarrow$ 0
2: for $forj¸i1to0$ do
3:     if parentheses at dfuds[j] is ( then
4:         count $\leftarrow$ count + 1
5:     end if
6: end for
```

3.4. Level-Order Unary Degree Sequence (LOUDS)

Algorithm 11 LOUDS (traversal)

```
1: start from root of the tree(r)
2: store value of node and its corresponding '10' bits in lds[count]
3: increment count
4: call ldstraversal(root)
5: increment count
```

Algorithm 12 ldstraversal (root)

```
for $foreachchildofnode(v)$ do
    store the value of the node
    add '1' bit to the string
    enqueue(the current child for BFS traversal)
    increment count by 1
end for
add bit'0' to the string
increment count by 1
if queue is empty then
    return
else
    call ldstraversal(dequeue)
end if
```

Algorithm 13 parent (y)

```
y=select1(rank0(x))
if y<0 then
    return -1
else
    return y
end if
```

Algorithm 14 firstchild (y)

```
y=select0(rank1(x))
if lds[y] = 0 then
    return -1
else
    return y
end if
```

Algorithm 15 lastchild (y)

```
y=select0(rank1(x)+1)-2
if lds[y] = 0 then
    return -1
else
    return y
end if
```

Algorithm 16 rightsibling (y)

```
if lds[y+1] = 0 then
    return -1
else
    return y+1
end if
```

4. Applications

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.

5. Some further useful suggestions

Succinct representation of data structures like trees and graph had been a field of great interest. Many scientists and researchers had published considerable amount of work on same.
Few of them are, Guy Jacobson, J. I. Munro and V. Raman, using their algorithm Rank and select function can be found in nearly constant time, i.e O(1).
Thus using them, other operations which are dependent on them can be found efficiently both in terms of time and space.

6. Conclusions

Through this project, we implemented a space efficient data structure to store data and performed navigational operations on the given static tree.
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(Cn) 2n logn + O(1) number of bits. Further we used three different succinct representation of a given tree to do so. We also analysed how we saved the space in representation of tree in succinct form.

7. Bibliography and citations

7.1. References :

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

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

5) 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)