Succinct Data Structures

May 24, 2022

Arshdeep Singh (2020csb1074) ,
Ashish Alok (2020csb1076) ,
Inayat Kaur (2020csb1088)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Napendra Solanki

Summary: We have implemented Succinct Trees in this project where we have used three different ways to encode a given tree which are Balanced Parentheses, Depth-first Unary Degree Sequence and Level-ordered Unary Degree Sequence and performed various navigation operations on the nodes.

---

1. Introduction

Data structures are used to store and organize some data and perform desired operations on it. Generally, we consider a data structure efficient if it performs desired operations faster, thus we consider very little about the space used by them to do so. Although the processor speeds are increasing day by day, but amount of data to be processed is also increasing at much higher rate. So,space required to store the data is also important factor for efficiency.
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)

For a pointer based representation of tree, if there are n nodes, then pointer size has to be atleast log(n) bits, which means for representation of entire tree structure, we need alteast nlog(n) bits.
Whereas with the help of succinct encoding, we require only 2 bits space per node.Thus overall we require only 2n bits for structure and some additional o(n) for secondary operations. Hence, with succinct representation, we require only $2n + o(n)$ space.

2. Functionality and Algorithms

2.1. Functionality

2.1.1 Balanced Parentheses

Jacobson in 1989, proposed the idea of representation of tree in the form of parentheses.
The idea is that while the tree is traversed in pre-order traversal, an open parenthesis stored when a node is reached the first time, and a closing parenthesis is stored when that node is reached last time. Thus, all the

children of particular vertex are between opening and closing parentheses of a the vertex. As a result of the this procedure we obtain a sequence of 2n characters, representing a pair of parentheses for each vertex.

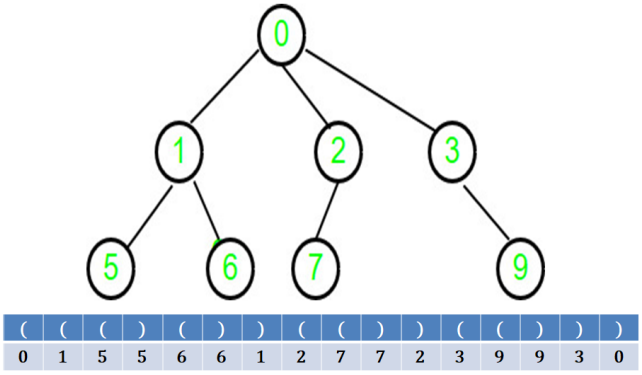

Figure 1: Representing tree as balanced parentheses

Consider the tree represented in the figure 2.
Starting pre-order traversal of tree from the root node (0), thus for root(0) we store a opening parentheses as it is encountered for first time. Now traverse down to its first child (1) and store $'('$ corresponding to it. But as (5) don't have any child so we store $')'$ for (5) and traverse to the next child of (1), i.e (6) and store opening parenthesis for it, but again as (6) don't have any child, we store closing parentheses for (6) and traverse back to (1). Thus in similar manner we traverse the sub-tree rooted at node (3) and node (6) and obtain a sequence of $2n$ parentheses for corresponding tree.
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)
- $first - child - bp(v)$: 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)

2.1.2 Depth-First Unary Degree Sequence

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.

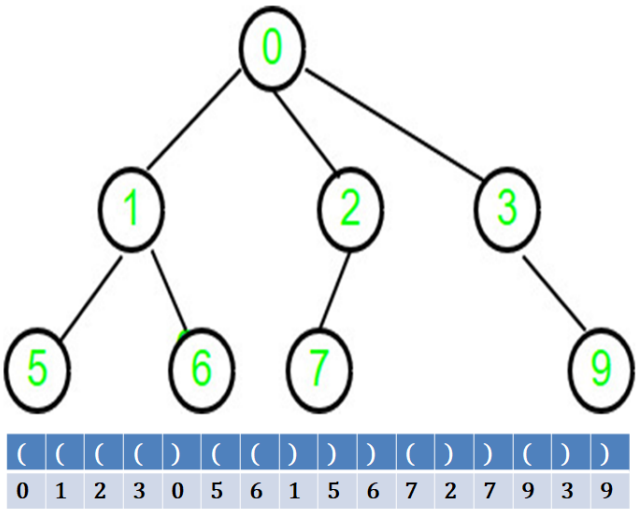

Figure 2: Representing tree as DFUDS

Consider the tree represented in the figure 2.
Starting pre-order traversal of tree from the root node (0), thus for root(0) we store a opening parentheses as it is encountered for first time. Now we store 3 opening parentheses for its children and then one closing parenthesis. Next we move to root's first child and store two opening parentheses for its children and then close the node. Similarly, we traverse the other nodes to obtain a sequence of 2n parentheses for corresponding tree.

Operations

- $parent-dfuds(v)$: 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)

2.1.3 Level-Order 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.

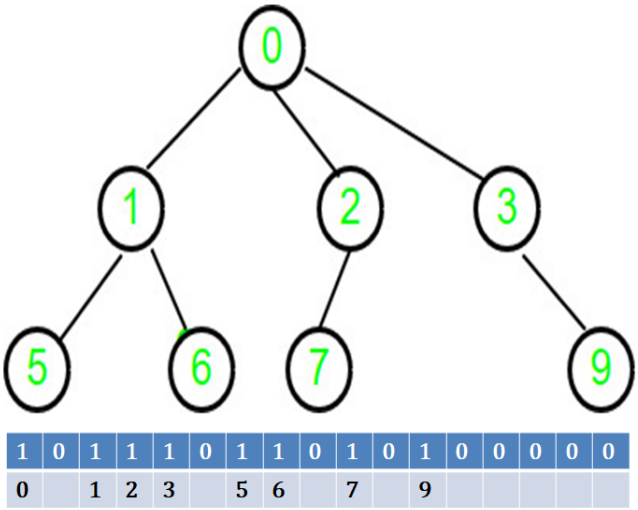

Figure 3: Representing tree as binary string of 1's and 0's

Consider the tree represented in the figure 3. Here we append bits '10' in the string for the node 0.We also store its value. Next it has three children so we append '1110' to the string and also store the values of the nodes.Node 1 has 2 children so we append '110'. Similarly for node 2 and 3 we append '10' and '10'respectively. For the nodes 5,6,7,9 (not having any child) we append '0' for each. Hence we have traversed the tree in the level order and converted the tree in a string of 1's and 0's.

Operations
(x passed in each function is the index of the corresponding node in the bit representation of the tree)

- $parent - lds(x)$: 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 :
- $degree - lds(v)$: 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)

2.2. Algorithms

2.2.1 Rank and select

Rank and Select algorithms have been implemented using Z-algorithm.
$Rank_p$(char* pattern, int size, int i) : return the frequency of pattern p in sequence

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

integer $p \leftarrow 0$ and $c \leftarrow 0$
integer $l \leftarrow size + 1 + i$
allocate memory for $Z - array \leftarrow malloc(l * sizeof(int))$
allocate memory for $new - string \leftarrow malloc((l + 1) * sizeof(char))$
create $new - string \leftarrow pattern + "A" + parentheses - string$
$a \leftarrow 0$ and $b \leftarrow 0$
for $j = 1tol$ **do**
  if $j > b$ **then**
    $a \leftarrow j$ and $b \leftarrow j$
    while $new - string[b - a] == new - string[b]b < l$ **do**
      increment b
    end while
    $Z - array[j] \leftarrow b - a$
    increment b
  end if
else
  $k\ getsj - a$;
  if $Z - array[k] < b - i + 1$ **then**
    $Z - array[j] \leftarrow Z - array[k]$;
  end if
else
  $a \leftarrow j$;
  while $new - string[b - a] == new - string[b]b < l$ **do**
    increment b
  end while
  $Z - array[j] \leftarrow b - a$
  increment b
end for
for p=1 to l-1 **do**
  if $Z - array[p] == size$ **then**
    increment c
  end if
  free(Z-array)
  free(new-string)
  return c
end for

$Select - p$(char* pattern, int size, int i) : return the position of the i th occurrence of pattern p

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

integer $p \leftarrow 0$
integer $l \leftarrow size + 1 + i$
allocate memory for $Z - array \leftarrow malloc(l * sizeof(int))$
allocate memory for $new - string \leftarrow malloc((l + 1) * sizeof(char))$
create $new - string \leftarrow pattern + "A" + parentheses - string$
store value of node and its corresponding '(' parenthesis in bp[count]
$a \leftarrow 0 and b \leftarrow 0$
for $j = 1 to do$ **do**
  if j>b **then**
    $a \leftarrow j and b \leftarrow j$
    while $new_string[b - a] == new - string[b]\ AND\ b < l$ **do**
      increment b
    end while
    $Z - array[j] \leftarrow b - a$
    increment b
  end if
else
  $k \leftarrow j - a$
  if $Z - array[k] < b - i + 1$ **then**
    $Z - array[j] \leftarrow Z - array[k]$
  end if
else
  a = j;
  while $new - string[b - a] == new - string[b]\ AND\ b < l$ **do**
    increment b
  end while
  $Z - array[j] \leftarrow b - a$
  increment b
end for
for $p = 1 tol - 1$ **do**
  if $Z - array[p] == size$ **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

2.2.2 Balanced Parentheses

Algorithm 3 $balanced - parentheses$

```
1: start from root of the tree(r)
2: store value of node and its corresponding '(' parenthesis in bp[count]
3: increment count
4: for for each child of node(v) do
5:     balanced − parentheses(v)
6: end for
7: store value of node and its corresponding ')' parenthesis in bp[count]
8: increment count
```

Algorithm 4 $find - close - bp\ (i)$

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

Algorithm 5 $find - open - bp\ (i)$

```
1: store value of node at bp[i] in some variable temp(say)
2: for for j ← i − 1 to 0 do
3:     if value of node at bp[j] is equal to temp then
4:         return j
5:     end if
6: end for
```

Algorithm 6 $enclose - bp\ (i)$

```
1: count ← 0
2: for for j ← i − 1 to 0 do
3:     if parentheses at bp[j] is '(' then
4:         count ← count + 1
5:     end if
6: else
       count ← count − 1 if count = 1 then
8:         return j
9:     end if
10: end for
```

2.2.3 Depth-first Unary Degree Sequence

Algorithm 7 $depth - first$

```
start from root of the tree(r)
for for each child of node(v) do
    store value of node and its corresponding '(' parenthesis in dfuds[count]
    increment count
end for
store value of node and its corresponding ')' parenthesis in dfuds[count]
increment count
for for each child of node(v) do
    depth − first(v)
end for
```

Algorithm 8 $find - close - dfuds\ (i)$

```
store value of node at dfuds[i] in some variable temp(say)
for for j = i + 1 to 2 ∗ number of nodes do
    if value of node at dfuds[j] is equal to temp then
        return j
    end if
end for
```

Algorithm 9 $find - open - dfuds\ (i)$

```
1: store value of node at dfuds[i] in some variable temp(say)
2: for for j ← i − 1 to 0 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 ← 0
2: for for j ← i − 1 to 0 do
3:     if parentheses at dfuds[j] is '(' then
4:         count ← count + 1
5:     end if
6: end for
```

2.2.4 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 for each child of node(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
```

3. What we have achieved and scope for further improvements

3.1. What we have achieved:

We have encoded given static tree into three different succinct representation, viz. Balanced Parentheses, Depth first unary degree sequence and Level order unary degree sequence.
Space requirement for pointer based representation of tree is 8 bytes(for pointer) per node, whereas we used only two characters, i.e 2 bytes for representation of encoded tree structure. Thus saving 6 bytes per node.
Further,we have implemented *Rank* and *Select* using Z-algorithm (used for pattern matching)
These algorithms take *lineartime* to perform the respective operations. As a result the function which depends on the rank and select also takes linear time.

3.2. Scope for further improvement:

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.

4. Conclusions

Through this project, we implemented a space efficient data structure to store data and performed navigational operations on the given static tree. 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.
For scope of further improvement we saw that more sophisticated like Jacobson's algorithm, rank and select can be calculated in constant time [O(1)] then most of the operations can also be calculated in constant time since they depend on finding the rank and select of indices.

Acknowledgements

Our project Succinct Data structure acknowledges the valuable contributions and guidance of certain people. We express our gratitude to Dr. Anil Shukla for his valuable guidance ,encouragement and support. He introduced us the idea of various data structures. His ideas and insights have greatly contributed to this project.
Our special thanks to our TA , Napendra Solanki. We are very grateful to him for guiding us through out our project and help us achieve the objective of our project. His knowledge has helped us to learn a lot.
We express our sincere gratitude to all our colleagues and staff members who supported us throughout. This project would not have been possible without their support.

5. Bibliography and citations

References

[1] Sam Heilbron. Succinct tree in practice. pages 1–12, 2017.

[2] Guy Jacobson. *Space Efficient Static tree and graphs*. PhD thesis, Carnegie Mellon University, 1989.

[3] MIT opencourseware. Advanced data structures, 2012.

[4] S. Shrinivasa Rao. *Succinct Data Structure*. PhD thesis, IMS, Chennai, 12 2001.
[1–4]