

# Sparse Arrays

The `sparse_array.c` file contains an implementation of a sparse array that attempts to be both space and time efficient.

The sparse array is represented using a tree structure. Each node in the tree contains a block of pointers to either the user supplied leaf values or to another node.

There are a number of parameters used to define the block size:

<code>OPENSSL_SA_BLOCK_BITS</code>	Specifies the number of bits covered by each block
<code>SA_BLOCK_MAX</code>	Specifies the number of pointers in each block
<code>SA_BLOCK_MASK</code>	Specifies a bit mask to perform modulo block size
<code>SA_BLOCK_MAX_LEVELS</code>	Indicates the maximum possible height of the tree

These constants are inter-related:

<code>SA_BLOCK_MAX</code>	<code>= 2 ^ OPENSSL_SA_BLOCK_BITS</code>
<code>SA_BLOCK_MASK</code>	<code>= SA_BLOCK_MAX - 1</code>
<code>SA_BLOCK_MAX_LEVELS</code>	number of bits in <code>size_t</code> divided by <code>OPENSSL_SA_BLOCK_BITS</code> rounded up to the next multiple of <code>OPENSSL_SA_BLOCK_BITS</code>

`OPENSSL_SA_BLOCK_BITS` can be defined at compile time and this overrides the built in setting.

As a space and performance optimisation, the height of the tree is usually less than the maximum possible height. Only sufficient height is allocated to accommodate the largest index added to the data structure.

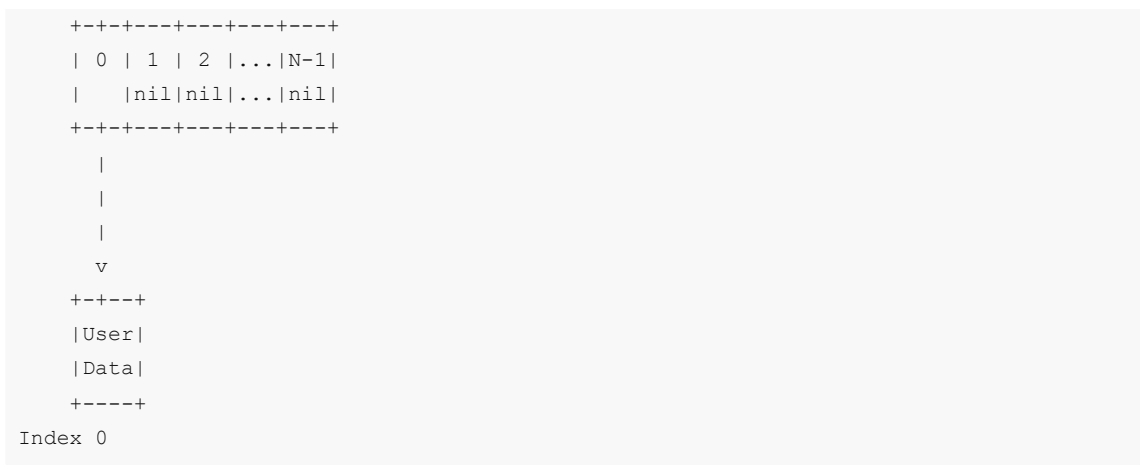
The largest index used to add a value to the array determines the tree height:

+-----+-----+		
Largest Added Index	Height of Tree	
+-----+-----+		
<code>SA_BLOCK_MAX - 1</code>	1	
<code>SA_BLOCK_MAX ^ 2 - 1</code>	2	
<code>SA_BLOCK_MAX ^ 3 - 1</code>	3	
...	...	
<code>size_t max</code>	<code>SA_BLOCK_MAX_LEVELS</code>	
+-----+-----+		

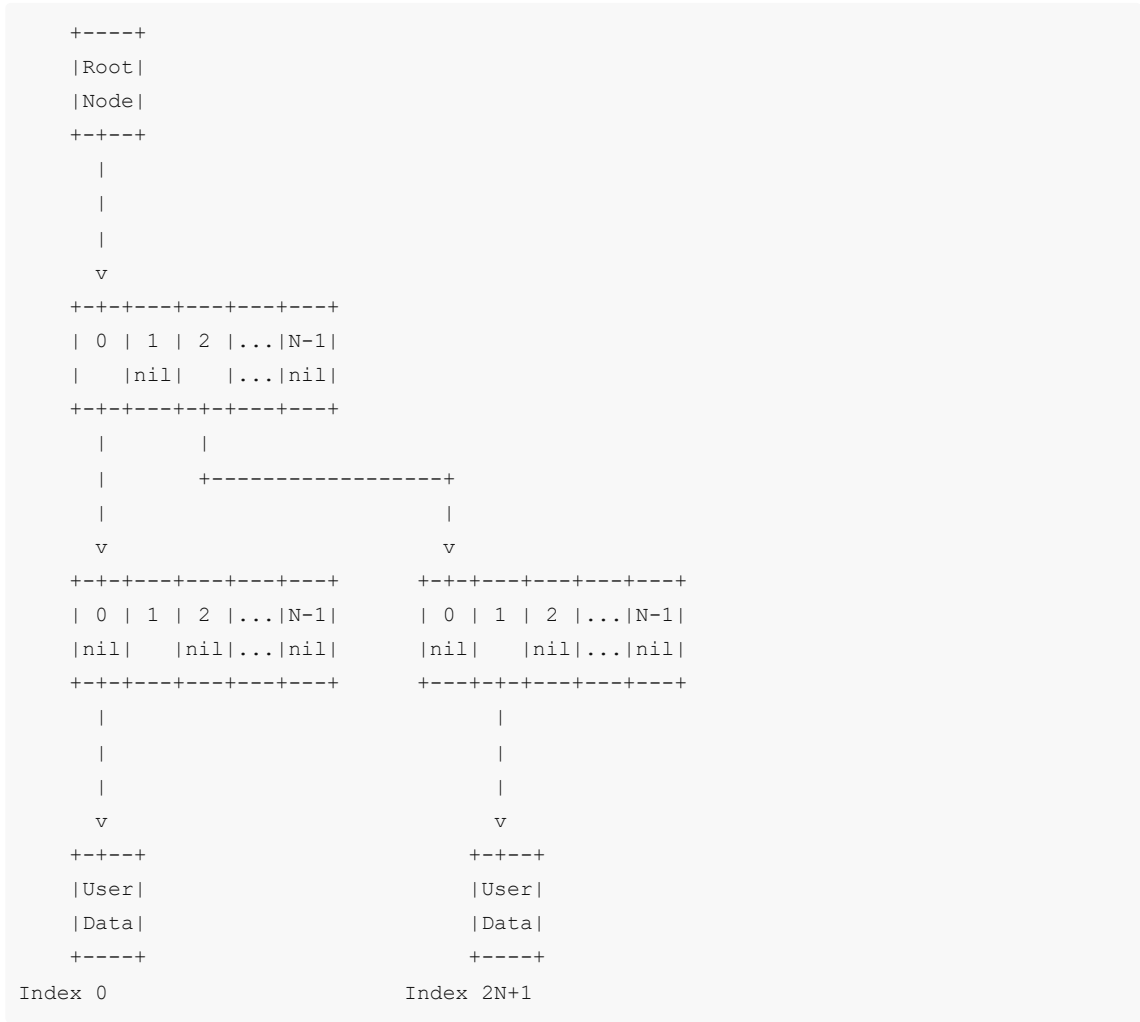
The tree height is dynamically increased as needed based on additions.

An empty tree is represented by a NULL root pointer. Inserting a value at index 0 results in the allocation of a top level node full of null pointers except for the single pointer to the user's data (N = `SA_BLOCK_MAX` for brevity):

+----+
Root
Node
+----+
v



Inserting at element  $2N+1$  creates a new root node and pushes down the old root node. It then creates a second second level node to hold the pointer to the user's new data:



The nodes themselves are allocated in a sparse manner. Only nodes which exist along a path from the root of the tree to an added leaf will be allocated. The complexity is hidden and nodes are allocated on an as needed basis. Because the data is expected to be sparse this doesn't result in a large waste of space.

```

+-----+
|Root|
|Node|
+--+--+
  |
  |
  |
  v
+--+-----+--+-----+
| 0 | 1 | 2 | ...|N-1|
|   |nil|   | ...|nil|
+--+-----+--+-----+
  |           |
  |           +-----+
  |           |
  v           v
+--+-----+--+-----+      +--+-----+--+-----+
| 0 | 1 | 2 | ...|N-1|      | 0 | 1 | 2 | ...|N-1|
|nil|nil|nil|...|nil|      |nil|   |nil|...|nil|
+--+-----+--+-----+      +--+-----+--+-----+
                                |
                                |
                                |
                                v
                                +--+--+
                                |User|
                                |Data|
                                +-----+
                                Index 2N+1

```

Note: sparse arrays only include pointers to types. Thus, `SPARSE_ARRAY_OF(char)` can be used to store a string.

Note: sparse arrays only include pointers to types. Thus, `SPARSE_ARRAY_OF(char)` can be used to store a string.