Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

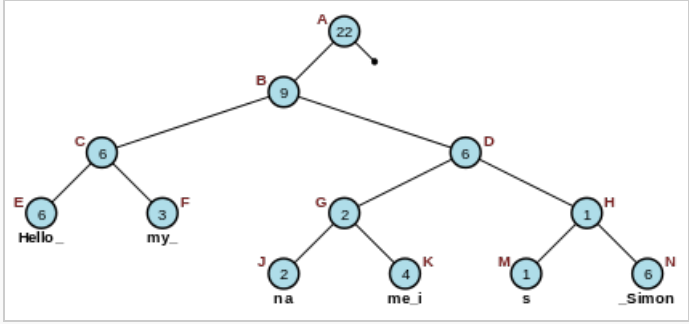
Languages
فارسی
Français
ไทย
Edit links

Rope (data structure)

From Wikipedia, the free encyclopedia
(Redirected from Rope (computer science))

This article **relies largely or entirely upon a single source**. Relevant discussion may be found on the talk page. Please help improve this article by introducing citations to additional sources. *(September 2011)*

In computer programming a **rope**, or **cord**, is a data structure composed of smaller strings that is used for efficiently storing and manipulating a very long string. For example, a text editing program may use a rope to represent the text being edited, so that operations such as insertion, deletion, and random access can be done efficiently.[1]



A simple rope built on the string of "Hello_my_name_is_Simon".

Contents

Description [edit]

A rope is a binary tree having leaf nodes that contain a short string. Each node has a weight value equal to the length of its string plus the sum of all leaf nodes' weight in its left subtree, namely the weight of a node is the total string length in its left subtree for a non-leaf node, or the string length of itself for a leaf node. Thus a node with two children divides the whole string into two parts: the left subtree stores the first part of the string. The right subtree stores the second part and its weight is the sum of the left child's weight and the length of its contained string.

The binary tree can be seen as several levels of nodes. The bottom level contains all the nodes that contain a string. Higher levels have fewer and fewer nodes. The top level consists of a single "root" node. The rope is built by putting the nodes with short strings in the bottom level, then attaching a random half of the nodes to parent nodes in the next level.

Operations [edit]

In the following definitions, *N* is the length of the rope.

Index [edit]

Definition: `Index(i)`: return the character at position *i*

Time complexity: O(log N)

To retrieve the *i*-th character, we begin a recursive search from the root node:

```
 // Note: Assumes 1-based indexing.
 function index(RopeNode
node, integer i)
     if node.weight < i then
         return
index(node.right, i -
node.weight)
     else
         if exists(node.left)
then
             return
index(node.left, i)
         else
             return
node.string[i]
         endif
     endif
 end
```

Figure 2.1: Example of index lookup on a rope.

For example, to find the character at *i=10* in Figure 2.1 shown on the right, start at the root node (A), find that 22 is greater than 10 and there is a left child, so go to the left child (B). 9 is less than 10, so subtract 9 from 10 (leaving *i=1*) and go to the right child (D). Then because 6 is greater than 1 and there's a left child, go to the left child (G). 2 is greater than 1 and there's a left child, so go to the left child again (J). Finally 2 is greater than 1 but there is no left child, so the character at index 1 of the short string "na", is the answer.

Concat [edit]

Definition: `Concat(S1, S2)`: concatenate two ropes, S_1 and S_2, into a single rope.
Time complexity: O(1) (or O(log N) time to compute the root weight)

A concatenation can be performed simply by creating a new root node with *left* = S_1 and *right* = S_2, which is constant time. The weight of the parent node is set to the length of the left child *S1*, which would take O(log N) time, if the tree is balanced.

As most rope operations require balanced trees, the tree may need to be re-balanced after concatenation.

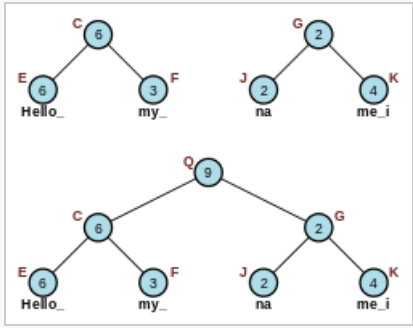


Figure 2.2: Concatenating two child ropes into a single rope.

Split [edit]

Definition: `Split (i, S)`: split the string *S* into two new strings S_1 and S_2, $S_1 = C_1, \ldots, C_i$ and $S_2 = C_{i+1}, \ldots, C_m$.
Time complexity: O(log N)

There are two cases that must be dealt with:

1. The split point is at the end of a string (i.e. after the last character of a leaf node)
2. The split point is in the middle of a string.

The second case reduces to the first by splitting the string at the split point to create two new leaf nodes, then creating a new node that is the parent of the two component strings.

For example, to split the 22-character rope pictured in Figure 2.3 into two equal component ropes of length 11, query the 12th character to locate the node **K** at the bottom level. Remove the link between **K** and the right child of **G**. Go to the parent of **G** and subtract the weight of **K** from the weight of **D**. Travel up the tree and remove any right links to subtrees covering characters past position 11, subtracting the weight of **K** from their parent nodes (only node **D** and **A**, in this case). Finally, build up the newly orphaned nodes **K** and **H** by concatenating them together and creating a new parent **P** with weight equal to the length of the left node **K**.

As most rope operations require balanced trees, the tree may need to be re-balanced after splitting.

Insert [edit]

Definition: `Insert(i, S')`: insert the string *S'* beginning at position *i* in the string *s*, to form a new string $C_1, \ldots, C_i, S', C_{i+1}, \ldots, C_m$.
Time complexity: O(log N).

This operation can be done by a `Split()` and two `Concat()` operations. The cost is the sum of the three.

Delete [edit]

Definition: `Delete(i, j)`: delete the substring $C_i, \ldots, C_{i+j-1}$, from s to form a new string $C_1, \ldots, C_{i-1}, C_{i+j}, \ldots, C_m$.
Time complexity: O(log N).

This operation can be done by two `Split()` and one `Concat()` operation. First, split the rope in three, divided by i-th and $i+j$-th character respectively, which extracts the string to delete in a separate node. Then concatenate the other two nodes.

Report [edit]

Definition: `Report(i, j)`: output the string $C_i, \ldots, C_{i+j-1}$.
Time complexity: O(j + log N)

To report the string $C_i, \ldots, C_{i+j-1}$, find the node u that contains C_i and *weight(u)* >= j, and then traverse T starting at node u. Output $C_i, \ldots, C_{i+j-1}$ by doing an in-order traversal of T starting at node u.

Figure 2.3: Splitting a rope in half.

Comparison with monolithic arrays [edit]

Advantages:

- Ropes enable much faster insertion and deletion of text than monolithic string arrays, on which operations have time complexity O(n).
- Ropes don't require O(n) extra memory when operated upon (arrays need that for copying operations)
- Ropes don't require large contiguous memory spaces.
- If only nondestructive versions of operations are used, rope is a persistent data structure. For the text editing program example, this leads to an easy support for multiple undo levels.

Performance[*citation needed*]

Operation	Rope	String
Index[1]	O(log n)	O(1)
Split[1]	O(log n)	O(1)
Concatenate (destructive)	O(log n)	O(n)
Concatenate (nondestructive)	O(n)	O(n)
Iterate over each character[1]	O(n)	O(n)
Insert	O(log n)	O(n)
Append	O(log n)	O(1) amortized, O(n) worst case
Delete	O(log n)	O(n)
Report	O(j + log n)	O(j)
Build	O(n)	O(n)

Disadvantages:

- Greater overall space usage when not being operated on, mainly to store parent nodes. There is a trade-off between how much of the total memory is such overhead and how long pieces of data are being processed as strings; note that the strings in example figures above are unrealistically short for modern architectures. The overhead is always O(n), but the constant can be made arbitrarily small.
- Increase in time to manage the extra storage
- Increased complexity of source code; greater risk for bugs

This table compares the *algorithmic* characteristics of string and rope implementations, not their "raw speed". Array-based strings have smaller overhead, so (for example) concatenation and split operations are faster on

small datasets. However, when array-based strings are used for longer strings, time complexity and memory usage for insertion and deletion of characters become unacceptably large. A rope data structure, on the other hand, has stable performance regardless of data size. Moreover, the space complexity for ropes and arrays are both O(n). In summary, ropes are better suited when the data is large and frequently modified.

See also [edit]

- The Cedar programming environment, which used ropes "almost since its inception"[1]
- The Model T enfilade, a similar data structure from the early 1970s.
- Gap buffer, a data structure commonly used in text editors that allows efficient insertion and deletion operations clustered near the same location

References [edit]

1. ^ *a b c d e* Boehm, Hans-J; Atkinson, Russ; Plass, Michael (December 1995). "Ropes: an Alternative to Strings" (PDF). *Software—Practice & Experience* (New York, NY, USA: John Wiley & Sons, Inc.) **25** (12): 1315–1330. doi:10.1002/spe.4380251203.

External links [edit]

- SGI's implementation of ropes for C++
- "C cords" implementation of ropes within the Boehm Garbage Collector library
- libstdc++ support for ropes
- Ropes for Java
- Ropes for OCaml
- ropes for Common Lisp
- pyropes for Python

Categories: Binary trees | String data structures