Leftist tree

From Wikipedia, the free encyclopedia

In computer science, a **leftist tree** or **leftist heap** is a priority queue implemented with a variant of a binary heap. Every node has an *s-value* which is the distance to the nearest leaf. In contrast to a *binary heap*, a leftist tree attempts to be very unbalanced. In addition to the heap property, leftist trees are maintained so the right descendant of each node has the lower s-value.

The height-biased leftist tree was invented by Clark Allan Crane.[1] The name comes from the fact that the left subtree is usually taller than the right subtree.

When inserting a new node into a tree, a new one-node tree is created and merged into the existing tree. To delete a minimum item, we remove the root and the left and right sub-trees are then merged. Both these operations take O(log *n*) time. For insertions, this is slower than binomial heaps which support insertion in amortized constant time, O(1) and O(log *n*) worst-case.

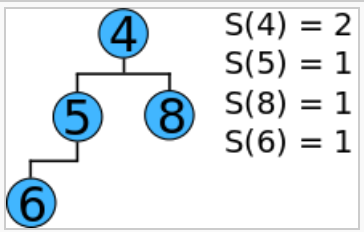
Leftist trees are advantageous because of their ability to merge quickly, compared to binary heaps which take Θ(*n*). In almost all cases, the merging of skew heaps has better performance. However merging leftist heaps has worst-case O(log *n*) complexity while merging skew heaps has only amortized O(log *n*) complexity.

Bias [edit]

The usual leftist tree is a *height-biased* leftist tree.[1] However, other biases can exist, such as in the *weight-biased* leftist tree.[2]

S-value [edit]

The **s-value** (or **rank**) of a node is the distance from that node to the nearest leaf of the extended binary representation of the tree. The extended representation (not shown) fills out the tree so that each node has 2 children (adding a total of 5 leaves here). The minimum distance to these leaves are marked in the diagram. Thus s-value of 4 is 2, since the closest leaf is that of 8 --if 8 were extended. The s-value of 5 is 1 since its extended representation would have one leaf itself.



S-values of a leftist tree

Merging height biased leftist trees [edit]

Merging two nodes together depends on whether the tree is a min or max height biased leftist tree. For a min height biased leftist tree, set the higher valued node as the right child of the lower valued node. If the lower valued node already has a right child, then merge the higher valued node with the sub-tree rooted by the right child of the lower valued node.

After merging, the s-value of the lower valued node must be updated (see above section, s-value). Now check if the lower valued node has a left child. If it does not, then move the right child to the left. If it does have a left child, then the child with the highest s-value should go on the left.

Java code for merging a min height biased leftist tree [edit]

```
public Node merge(Node x, Node y) {
```

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

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

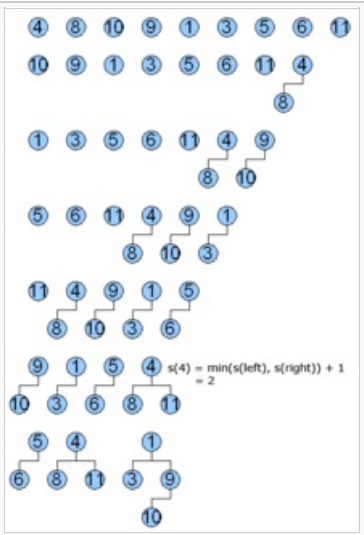
Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Deutsch
فارسی
Српски / srpski
中文
Edit links

```
public Node merge(Node x, Node y) {
  if(x == null)
    return y;
  if(y == null)
    return x;

  // if this was a max height biased leftist tree, then the
  // next line would be: if(x.element < y.element)
  if(x.element.compareTo(y.element) > 0) {
    // x.element > y.element
    Node temp = x;
    x = y;
    y = temp;
  }

  x.rightChild = merge(x.rightChild, y);

  if(x.leftChild == null) {
    // left child doesn't exist, so move right child to the left side
    x.leftChild = x.rightChild;
    x.rightChild = null;

  } else {
    // left child does exist, so compare s-values
    if(x.leftChild.s < x.rightChild.s) {
      Node temp = x.leftChild;
      x.leftChild = x.rightChild;
      x.rightChild = temp;
    }
    // since we know the right child has the lower s-value, we can just
    // add one to its s-value
    x.s = x.rightChild.s + 1;
  }
  return x;
}
```

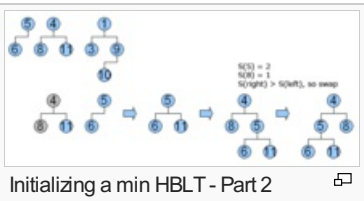
Initializing a height biased leftist tree [edit]



Initializing a min HBLT - Part 1

Initializing a height biased leftist tree is primarily done in one of two ways. The first is to merge each node one at a time into one HBLT. This process is inefficient and takes O(*nlogn*) time. The other approach is to use a queue to store each node and resulting tree. The first two items in the queue are removed, merged, and placed back into the queue. This can initialize a HBLT in O(*n*) time. This approach is detailed in the three diagrams supplied. A min height biased leftist tree is shown.

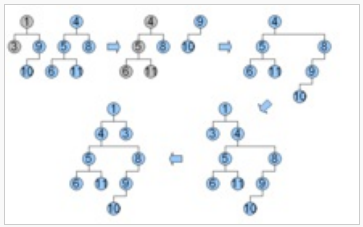
To initialize a min HBLT, place each element to be added to the tree into a queue. In the example (see Part 1 to the left), the set of numbers [4, 8, 10, 9, 1, 3, 5, 6, 11] are initialized. Each line of the diagram represents another cycle of the algorithm, depicting the contents of the queue. The first five steps are easy to follow. Notice that the freshly created HBLT is added to the end of the queue. In the fifth step, the first occurrence of an s-value greater than 1 occurs. The sixth step shows two trees merged with each other, with predictable results.



Initializing a min HBLT - Part 2

In part 2 a slightly more complex merge happens. The tree with the lower value (tree x) has a right child, so merge must be called again on the subtree rooted by tree x's right child and the other tree. After the merge with the subtree, the resulting tree is put back into tree x. The s-value of the right child (s=2) is now greater than the s-value of the left child (s=1), so they must be swapped. The s-value of the root node 4 is also now 2.

Part 3 is the most complex. Here, we recursively call merge twice (each time with the right child 's subtree that is



Initializing a min HBLT - Part 3

not grayed out). This uses the same process described for part 2.

References [edit]

1. ^ [a] [b] Clark A. Crane (1972), *Linear Lists and Priority Queues as Balanced Binary Trees*, Department of Computer Science, Stanford University.
2. ^ Seonghun Cho and Sartaj Sahni (1996), "Weight Biased Leftist Trees and Modified Skip Lists", *Journal of Experimetnal Algorithmics* **3**

External links [edit]

- Leftist Trees, Sartaj Sahni

Further reading [edit]

- Robert E. Tarjan (1983). *Data Structures and Network Algorithms*. SIAM. pp. 38–42. ISBN 978-0-89871-187-5.
- Dinesh P. Mehta; Sartaj Sahni (28 October 2004). *Handbook of Data Structures and Applications. Chapter 5: Leftist trees*. CRC Press. ISBN 978-1-4200-3517-9.

Categories: Trees (data structures) | Heaps (data structures) | Priority queues