CSCI-1200 Data Structures — Fall 2009
Lecture 22 – Priority Queues and Leftist Heaps

Review from Lecture 21

- Queues and Stacks, What's a Priority Queue?
- A Priority Queue as a Heap, `percolate_up` and `percolate_down`
- A Heap as a Vector, Building a Heap, Heap Sort

Today's Class

- Merging heaps are the motivation for *leftist heaps*
- Mathematical background & Basic algorithms

22.1 Leftist Heaps — Overview

- Our goal is to be able to merge two heaps in $O(\log n)$ time, where n is the number of values stored in the larger of the two heaps.
  - Merging two binary heaps requires $O(n)$ time
- Leftist heaps are binary trees where we deliberately attempt to eliminate any balance.
- Leftists heaps are implemented explicitly as trees.

22.2 Leftist Heaps — Mathematical Background

- **Definition:** The *null path length* (NPL) of a tree node is the length of the shortest path to a node with 0 children or 1 child. The NPL of a leaf is 0. The NPL of a NULL pointer is -1.
- **Definition:** A *leftist tree* is a binary tree where at each node the null path length of the left child is greater than or equal to the null path length of the right child.
- **Definition:** The *right path* of a node (e.g. the root) is obtained by following right children until a NULL child is reached.
  - In a leftist tree, the right path of a node is at least as short as any other path to a NULL child.
- **Theorem:** A leftist tree with $r > 0$ nodes on its right path has at least $2^r - 1$ nodes.
  - This can be proven by induction on r.
- **Corollary:** A leftist tree with n nodes has a right path length of at most $\lfloor \log(n+1) \rfloor = O(\log n)$ nodes.
- **Definition:** A *leftist heap* is a leftist tree where the value stored at any node is less than or equal to the value stored at either of its children.

22.3 Leftist Heap Operations

- The `insert` and `delete_min` operations will depend on the `merge` operation.
- Here is the fundamental idea behind the merge operation. Given two leftist heaps, with `h1` and `h2` pointers to their root nodes, and suppose `h1->value <= h2->value`. Recursively merge `h1->right` with `h2`, making the resulting heap `h1->right`.
- When the leftist property is violated at a tree node involved in the merge, the left and right children of this node are swapped. This is enough to guarantee the leftist property of the resulting tree.
- `Merge` requires $O(\log n + \log m)$ time, where m and n are the numbers of nodes stored in the two heaps, because it works on the right path at all times.

22.4 Merge Code

```
template <class T>
class LeftNode {
public:
  LeftNode() : npl(0), left(0), right(0) {}
  LeftNode(const T& init) : value(init), npl(0), left(0), right(0) {}
  T value;
  int npl;         //  the null-path length
  LeftNode* left;
  LeftNode* right;
};
```

Here are the two functions used to implement leftist heap merge operations. Function merge is the driver. Function merge1 does most of the work. These functions call each other recursively.

```
template <class Etype>
LeftNode<Etype>* merge(LeftNode<Etype> *H1,LeftNode<Etype> *H2) {
  if (!h1)
    return h2;
  else if (!h2)
    return h1;
  else if (h2->value > h1->value)
    return merge1(h1, h2);
  else
    return merge1(h2, h1);
}

template <class Etype>
LeftNode<Etype>* merge1(LeftNode<Etype> *h1, LeftNode<Etype> *h2) {
  if (h1->left == NULL)
    h1->left = h2;
  else {
    h1->right = merge(h1->right, h2);
    if(h1->left->npl < h1->right->npl)
      swap(h1->left, h1->right);
    h1->npl = h1->right->npl + 1;
  }
  return h1;
}
```

22.5 Exercises

1. Explain how `merge` can be used to implement `insert` and `delete_min`, and then write code to do so.

2. Show the state of a leftist heap at the end of:

```
insert 1, 2, 3, 4, 5, 6
delete_min
insert 7, 8
delete_min
delete_min
```