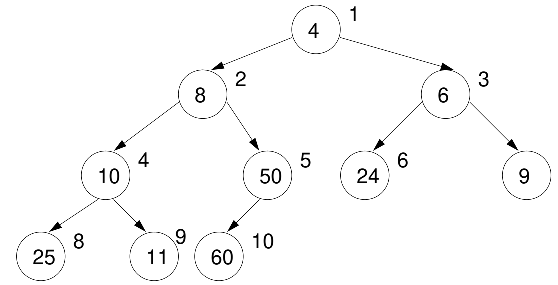
CSE 326
Lecture 24

Spring Quarter 1995

Standard Binary Heap

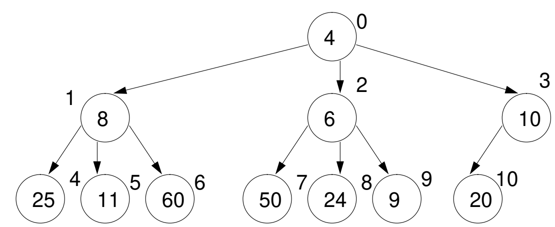


1	2	3	4	5	6	7	8	9	10		
4	8	6	10	50	24	9	25	11	60		

d-Heaps

- Use a complete d-ary tree.
- Each node has d children instead of just 2.
- Store the root at 0.
- The children of i are stored in di+1, di+2, ..., di+d.
- The parent of i is stored in (i-1)/d

3-Heap



0	1	2	3	4	5	6	7	8	9	10	
4	8	6	10	25	11	60	50	24	9	20	

Are d-Heaps Better Than Binary Heaps?

- Weise on d-heaps

  "... the Delete_Min operation is more expensive because, even though the tree is shallower, the minimum of d children must be found, which takes d-1 comparisons using a standard algorithm."

- Weiss is wrong, probably 4-heaps are best.

  1. Theory says that 4 heaps are probably better.
  2. 4 heaps take better advantage of cache memory than 2 heaps
  3. Operations of 4 heaps can be coded as efficiently as 2 heaps. For example, finding the parent for both 2 heaps and 4 heaps can be done by shifting, w/o arithmetic.

The Theory

- d-heaps are better than binary heaps for insertion both in worst case and average case.
- For some d, the d-heap is better than binary heaps for delete_min.

  Assuming that all delete-min's goes to the leaves the time to delete_min is

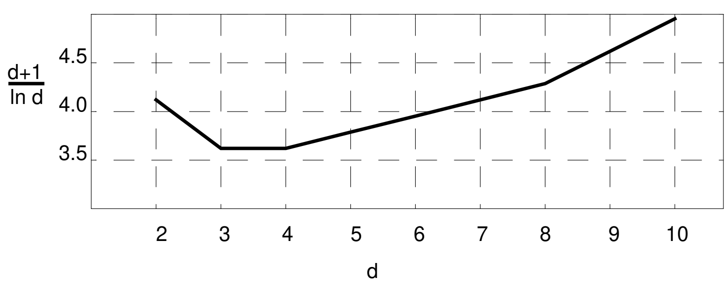
$$(ad + b) \log_d N$$

For simplicity let a = b = 1, then the time per delete_min is

$$(d + 1) \log_d N = (d + 1) \ln N / \ln d$$

d	2	3	4	5	6	7	8	9	10
(d+1) / ln d	4.33	3.64	3.61	3.73	3.91	4.11	4.33	4.55	4.78

Plot of d vs. (d+1) / ln d



Mergeable Priority Queue

- Mergeable priority queues allow for the efficient implementation of the operation Merge(H_1,H_2) which merges two priority queues into a single priority queue.
- Normally, Merge(H_1,H_2) returns a new priority queue destroying H_1 and H_2 in the process.
- Applications

  1. Specific algorithms like one for minimum spanning tree and another for the optimal binary search tree.
  2. Managing work queues in a distributed system.

Leftist Trees

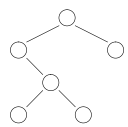
- A leftist tree is a binary tree with two properties.

  1. Heap property - key at a node is less than or equal to the keys of its children.

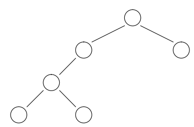
  2. Leftist property - For each node the null path length of the left subtree is greater than the null path length of the right subtree.

- Null path length - The null path length of a node (subtree) is the shortest path to a node without two children. A null has null path length = -1.

not a leftist tree



leftist tree



The Right Path of a Leftist Tree is Short

- In a leftist tree if the right path has length at least r then the leftist tree has at least $2^r - 1$ nodes.
- Proof is by induction on r.

  Basis: r = 1. The tree must have at least 1 node, the root, and $2^1 - 1 = 1$.

  Inductive Step: Assume the statement is true for all r' < r and r > 1. The right subtree has a right path of at least r-1 nodes, so by the induction hypothesis, it has at least $2^{r-1} - 1$ nodes. The left subtree must also have a right path of length at least r - 1. By the inductive hypothesis it has at least $2^{r-1} - 1$ nodes. Thus the total number of nodes is at least

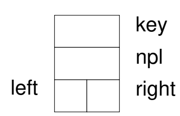
  $$2^{r-1} - 1 + 2^{r-1} - 1 + 1 = 2^r - 1.$$

- Thus a leftist tree with n nodes must have a right path of length $\le \log n$.

Data Structure for Leftist Trees

- Assume that a node of a leftist tree contains the following

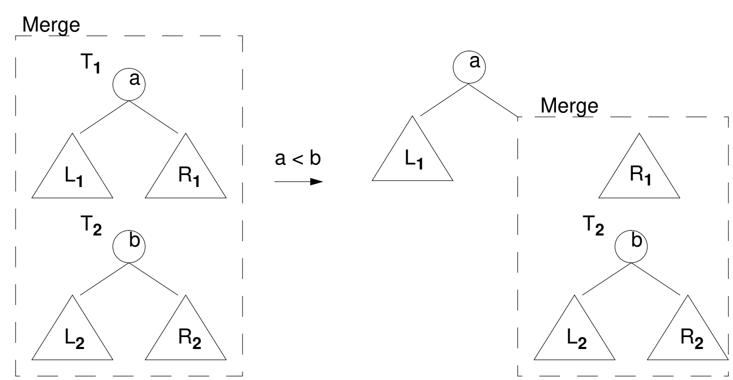
npl = null path length



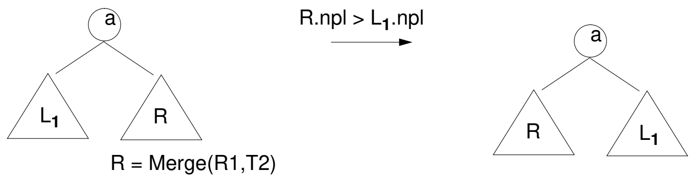
- The basic operation is the merging of two leftist trees. O(log n)
- Insert by merging a single node with a leftist tree. O(log n)
- Delete_min by merging the left and right subtrees of the root. O(log n)

Merging Two Leftist Trees

- Merge(T_1,T_2) returns a new leftist tree assuming T_1 and T_2 are two distinct leftist trees.

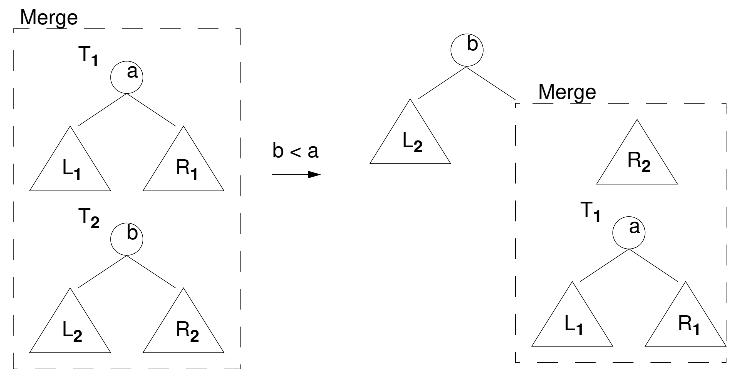


Merge Continued

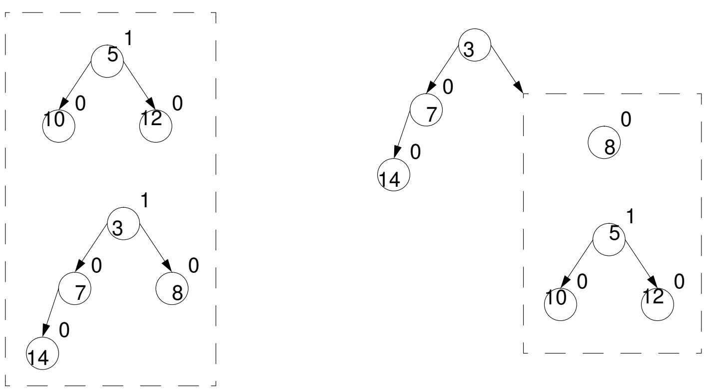


Merge Explanation Continued

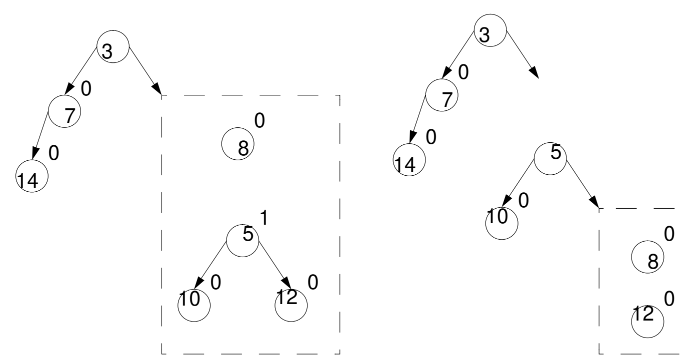
- What if b < a?



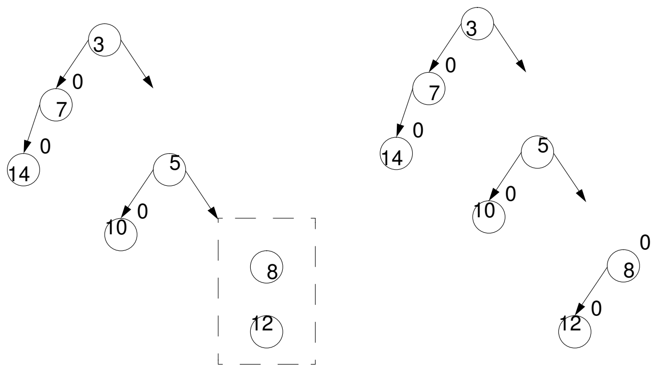
Example



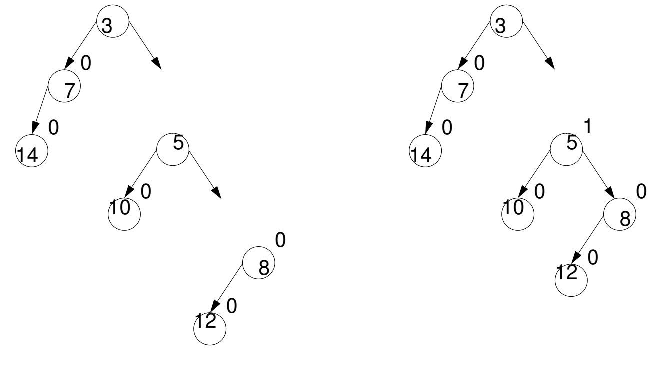
Example Continued



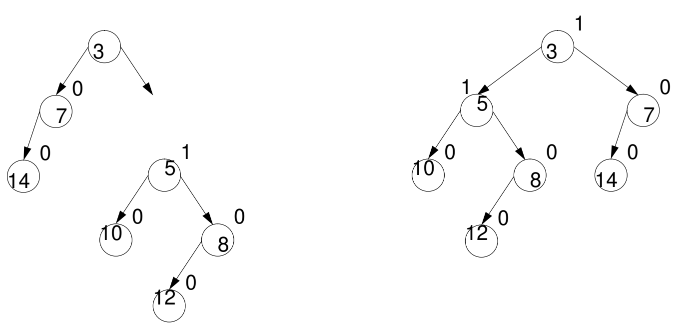
Example Continued



Example Continued



Example Continued



Iterative Leftist Tree Merging

- Two passes to merge T_1 and T_2.

  Downward pass: merge the right paths of T_1 and T_2 forming a new tree T.

  Upward pass: Return along the right path of T computing the null path lengths and interchanging the left and right subtrees to maintain the null path length property.