Binary Heap - Extract Min
- Min key at root $\rightarrow$ save it
- Copy $A[i]$ to root $(A[1])$ $\rightarrow$ decrement $n$
- Sift the root key down
  - Find smaller of two children
  - If larger, swap with this child
- Return saved root key

Def: Leftist Heap is binary tree where:
- Keys are heap ordered
- All nodes $v$, $npl(v.left) \geq npl(v.right)$

Examples:

Leftist Property:
Null path length
$npl(v) = \text{length of shortest path to null}$

- If $v = \text{null}$
  $$npl(v) = \begin{cases} -1 & \text{if } v = \text{null} \\ 1 + \text{min} \left( npl(v.left) \right) & \text{o.w.} \end{cases}$$

Example: Key extract-min()
- if ($n == 0$) Error - Empty heap
- result $\leftarrow A[1]$
- $z \leftarrow A[n-1]$ // get replacement
- $i \leftarrow \text{sift-down}(i, z)$
- $A[i] \leftarrow z$
- return result

Analysis: Both insert + extract-min take time proportional to tree height
Tree is complete $\Rightarrow O(\log n)$ time
Class structure: Leftist Heap  

private class LHNode {
  Key x
  LHNode left, right
  int npl
}

private LHNode root

public LeftistHeap() { root = null }
   " constructor
   " Key extractMin ()
   " void mergeWith (LeftistHeap H)
     ... (other private/protected utilities)

public mergeWith (LeftistHeap H2) {
   root = merge (this.root, H2.root)  // helper function
   H2.root = null  // merger destroys
}

Merge helper: 2 phases

1. Merge right paths by order of keys + update npl's
2. Check leftist property + swap

Lemma: A leftist tree with \( r \geq 1 \) nodes along its rightmost path has \( n \geq 2^{r-1} \) nodes

Proof: (Sketch - see latex notes)

By induction: \( n \geq 2^{r-1} \), \( n = n_c + n_a = 2^{r-1} + 2^{r-1} \) node.

Public LeftistHeap {
  root = null
  " key insert (Key x)
  " Key extractMin ()
  " void mergeWith (LeftistHeap H2)
     ... (other private/protected utilities)
}

Analysis: Time \( n \) Rightmost path

Insert + Extract-min? Exercises

LHNode merge(LHNode u, LHNode v) {
  if (u == null) return v
  if (v == null) return u
  if (u.key > v.key) // swap so u is smaller
    swap u <-> v
  if (u.left == null) u.left = v
  else u.right = merge (u.right, v)
  if (u.left.npl < u.right.npl)
    swap u.let + u.right
  u.npl = 1 + min(u.right.npl)
  return u
}

Final tree!
Dictionary:
- `insert (Key x, Value v)`
  - Insert `(x, v)` in dict. (No duplicates)
- `delete (Key x)`
  - Delete `x` from dict. (Error if `x` not there)
- `find (Key x)`
  - Returns a reference to associated value `v`, or `null` if not there.

Search:
- Given a set of `n` entries each associated with key `x`, value `v`.
  - Time complexity for each operation: `O(1)`.
- Time complexity for inserting a duplicate: `O(1)`.
- Store for quick access and updates.
- Ordered: Assume keys are totally ordered: `<`, `>`, `==`.

Efficiency:
- Depends on tree's height
  - Balanced: `O(log n)`
  - Unbalanced: `O(n)`

Binary Search Trees
- Basic definitions
- Finding keys

Idea: Store entries in binary tree sorted (inorder traversal) by key.

Can we achieve `O(log n)` time for all ops? **Binary Search Trees**

- `Find`: `O(log n)` by binary search
- `Insert/Delete`: `O(n)` time

Find: How to find a key in the tree?
- Start at root `p = root`
- If `(x < p.key)` `find (x, p.left)`
- If `(x > p.key)` `find (x, p.right)`
- If `(x == p.key)` found: return value
- If `(p == null)` return `null`

Example:
- Initial call: `find(x) { return find(x, root) }`
- Value: `find (Key x, BSTNode p)`
  - If `(p == null)` return `null`
  - Else if `(x < p.key)` `return find(x, p.left)`
  - Else if `(x > p.key)` `return find(x, p.right)`
  - Else return `p.value`
Announcements - Thu, 9/15

- Homework 1 - Coming
- Fast-forward recordings
  for Union-Find & Heaps? - Working on these
- My office hours - Soon

- Where is the lecture recording?
  - If you don't see the link
    on the lecture page visit
    ELMS → Panopto
    (and send me a reminder)