Decrease-Key:
- Given an entry \((x, v)\), decrease the key value from \(x\) to \(y\).
- How to identify the entry?
  - Heaps do not support an efficient way to find keys.

Locator: A special (abstract) object that identifies an entry of the heap.

Locator \(r = \text{insert}(x, v)\):
- \(r\):
- \(\text{decrease-key}(r, y)\):

Why not just return a pointer to node \((x, v)\)?
- Private information
  - Locator is a public object (e.g., an inner class of the Heap)
- How about increase-key?
  - Heaps are very asymmetrical w.r.t. keys

Heap: Review
- A data structure storing key-value pairs
- Supports (at a minimum)
  - \(\text{insert}(\text{Key } x, \text{Value } v)\)
  - \(\text{extract-min}()\)
- Example: Binary heap used in Heapsort

Why decrease-key?
- Dijkstra's algorithm
  - Heap tracks distances to vertices from source
  - \(n\) extract-mins
  - upto \(n^2\) decrease-keys
  - want decrease key fast!

History:
- 1984: Fibonacci Heaps (Fredman + Tarjan)
  - many variants
  - Complex to analyze
- 2013: Quake Heap (Timothy Chan)
  - Much simpler

Quake Heap:
- Collection of binary trees
- Nodes organized in levels
- All entries are leaves at level 0
- Internal nodes have 1 or 2 children
- Parent stores smaller of child keys

Quake Heaps I
- Basic definitions
- Operations

Heapsort
- How to identify the entry?
**Basic utilities:**
- `make-root(Node u)`: Make `u` a root
- `trivial-tree(Key x)`: Create 1-node tree with key `x`
- `link(Node u, Node v)`: Link `u` and `v` roots on same level
- `cut(Node w)`: Assuming `w` has right child - cuts it off as new root
- `make-root(Node u)`: Make `u` a root
- `cut(Node w)`

**Quake Heaps II**
- **Utility ops**
- **Insert**
- **Decrease-key**

**void decrease-key(Locator r, Key y)**
- Node `u ← r.get Node()` // get least node
- node `u.child ← null`
- do {
  - `u.key ← y` // update key value
  - `u.child ← u.u.parent` // go up
- } while (`u ≠ null` && `u.child = u.left`)
- if (`u ≠ null`) `cut (u) // cut subtree`

**Decrease Key**
- Use locator to access leaf
- Follow left-child path to highest ancestor
- `Cut (w)` - Now we are free to change key
- In code, we'll change up order of ops

**Insert**: Super lazy! Just make a single node tree

**Locator insert(Key x)**
- Node `u ← new trivial-tree(x)`
- return new Locator(x)

**We'll apply these utilities to implement operations**
Extract-Min:
- Find the root with smallest key (brute force)
- Delete all nodes down to leaf - many trees
- Merge trees together
  - Work bottom-up
  - Merge 2 trees at level k to form tree at level k+1
- Too 'stringy'? Flat @QUAKE!

Quake:
\[
\text{for } (k = 0, 1, 2, \ldots, \text{nLevels} - 2) \{
\text{if } (\text{nodeCt}[k+1] > 0.75 \times \text{nodeCt}[k]) \{
\text{remove all nodes at level } k+1 \\
\text{and higher}
\}
\text{make all nodes at level } k \text{ roots}
\]

Intuition: Tree becomes "stringy" after many extractions.
- This is evidenced by the fact that node counts do not decrease
- When this happens - we flatten so we can build up later

So far:
- insert + decrease-key - lazy!
- Don't worry about tree balance
- number of roots
- insert - \(O(1)\) time
- dec-key - \(O(\log n)\) [later: \(O(1)\)]

Quake Heap III
- Extract Min

Extract Min Example:
Key extract-min()

Node u ← find root (all levels) with smallest key
Key result ← u.key

delete-left-path(u)
remove u from roots [u.level]
merge-trees()
quake()
return result

Extract-min: Recap

- find root with min key
- delete left-chain to leaf
- merge trees
- quake (if needed)
- return result

Faster Decrease-key:

- Each node stores pointer to leaf with key (only one change)
- Each leaf stores highest left chain ancestor (path trace O(1) time)

Quake Heaps IV

Extract min (cont)

- Faster decrease key

Decrease-key:

- Locate leaf node - O(1)
- Trace path up left-child links
- Cut O(1)
- Change key < O(height) = O(log n)

Times:

- Insert - O(1)
- Decrease-key - O(log n)
- Extract-min - ??

Clear-all-above(lev) removes all nodes in levels lev+1..nLevels-1 and makes nodes of lev into roots

void delete-left-path(u)

while (u ≠ null)
  cut(u) → make u subtree into root
  nodeCt[u.level] -= 1
  u ← u.left

void merge-trees()

for (lev = 0..nLevels-2)
  while (roots[lev].size ≥ 2)
    Node u, v ← remove any 2 from roots[lev]
    make-root(link(u,v))

void quake()

for (lev = 0..nLevels-2)
  if (nodeCt[lev+1] < ¾ * nodeCt[lev])
    clear-all-above(lev)

Can we do better? O(1) ? O(n)

 amortized
Amortized Analysis:
- Can show that extract-min runs in $O(\log n)$ amortized time.
- Given any sequence of ops (starting from empty heap) time to do $m$ ops (insert, dec-key, extract-min) is $O(m \cdot \log n) \frac{n}{m} = O(\log n)$.
- $n = \text{max no. of keys}$.

Potential-Based Analysis:
- Each instance of the data structure assigned a potential $\Psi$.
- Low potential $\Rightarrow$ good structure.
- High potential $\Rightarrow$ bad structure.

Why is Quake Heap efficient?
- **Insert**: $O(1)$ worst case.
- **Decrease-key**: $O(1)$ worst case (assuming enhancements).
- **Extract-min**: As bad as $O(n)$ [no. of roots]?

Quake Heaps V
- Analysis (Quick + Dirty)

Intuition:
- Extract min actual cost is high $\triangleright$
- Tree height $> O(\log n)$.
- Quake will flatten.
- Many more roots than $O(\log n)$.
- Merge trees will reduce no. to $O(\log n)$.

Potential decrease compensates for high actual cost.

Lemma: Amortized cost of insert/dec-key $= O(1)$.
extract-min $= O(\log n)$.

Quake Heap Potential:
- Let $N = \text{no. of nodes}$.
- $R = \text{no. of roots}$.
- $B = \text{no. of nodes with 1 child (bad nodes)}$.

$\Psi = N + 2R + 4B$.

Idea: The amortized cost of an operation defined to be $(\text{actual cost}) + (\text{change in } \Psi)$.

Intuition: Expensive ops okay if they improve structure, actual $= \text{high}$, $\Delta \Psi = \text{negative}$.