**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.

**Heap Review**
- A data structure storing key-value pairs.
- Supports (at a minimum)
  - Insert \((\text{Key } x, \text{Value } v)\)
  - 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!

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

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

**Quake Heaps I**
- Basic definitions
- Operations

**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
  - null child always on right

**Locator**
- Heaps are very asymmetrical w.r.t. keys
- 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
cut(Node w): Assuming w has right child—cuts it off as new root

void make-root(Node u)
  u.parent = null
  add u to roots[u.level]

Node trivial-tree(Key x)
  Node u ← new Node(key x, level 0)
  nodeC[t[0]] += 1
  make-root(u)
  return u

Node link(Node u, Node v)
  int lev ← u.level + 1 (≡ v.level + 1)
  if (u.key ≤ v.key)
    w ← new Node(u.key, lev, u, v)
  else
    w ← new Node(v.key, lev, v, u)
  nodeC[t[lev]] += 1
  u.parent ← v.parent ← w
  return w

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 + v
- u + v roots on same level

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

void cut(Node w)
  Node v ← w.right
  if (v ≠ null)
    w.right ← null
    make-root(v)

We’ll apply these utilities to implement operations

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

Decrease Key:
- Use locator to access leaf
- Follow left-child path to highest ancestor
- Cut (u): 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)
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'"? Flatten \text{QUAKE}!

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

Finally, return 4
Key extract-min

- Node $u \leftarrow \text{find root (all levels) with smallest key}$
- $\text{key result} \leftarrow u.\text{key}$
- $\text{delete-left-path}(u)$
- remove $u$ from roots $[u.\text{level}]$
- $\text{merge-trees}()$
- $\text{quake}()$
- $\text{return result}$

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)

Decrease-key:

- Locate leaf node - $O(1)$
- Trace path up left-child links
- Cut $O(1)$
- Change key

Quake Heaps IV

- Extract min (cont)
- Faster decrease key

Quake Heaps:

- Locate leaf node - $O(1)$
- Trace path up left-child links
- Cut $O(1)$
- Change key - $O(\text{height}) = O(\log n)$

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

Clear-all-above $(lev)$ removes all
nodes in levels $lev+1..n\text{levels}-1$ and makes nodes of $lev$ into roots

Times:

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

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

Will show $O(\log n)$ 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 $O(m \cdot \log n)$
  $n = \text{max no. of keys}$

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?
- $\text{insert}$: $O(1)$ worst case
- $\text{decrease-key}$: $O(1)$ worst case (assuming enhancements)
- $\text{extract-min}$: As bad as $O(n)$ [no. of roots]

Intuition:
- Extract min actual cost is high
  - 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

Quake Heaps V
- Analysis (Quick + Dirty)

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

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 = high $\Delta \Psi = \text{negative}$