CMSC 420: Lecture X01
Quake Heaps

Priority Queues and Heaps: A *priority queue* is an abstract data structure storing key-value pairs. The basic operations involve inserting a new key-value pair (where the key represents the *priority*) and extracting the entry with the smallest priority value. These operations are called *insert* and *extract-min*, respectively. Priority queues are often implemented by a tree data structure called a *heap*, where keys decrease monotonically along any path to the root.

In addition to insert and extract-min, there are a number of operations that we may like to perform on priority queues, and heaps are often augmented with different capabilities (such as the capability to merge two heaps). As with many data structures, the objective is to perform these operations in time $O(\log n)$, where n is the number of keys in the data structure.

Decrease-Key Operation: One fundamental operation, called *decrease key*, involves decreasing the key value of a given entry in the heap by a given amount. The decrease-key operation arises when heaps are used in algorithms such as Dijkstra's algorithm for shortest paths in graphs and Prim's algorithm for computing minimum-cost spanning trees. In such algorithms, the decrease-key operation is performed more frequently than the extract-min operation, and so it is critical that it be performed efficiently. (In Dijkstra's algorithm, decrease-key is performed once for every edge of the graph whereas extract-min is performed once for every vertex. A graph with n vertices may have $O(n^2)$ edges, so there may be many more decrease-key operations compared to extract-mins.)

It is not hard to implement the operations insert, extract-min, and decrease-key using a standard binary heap (the same data structure used in heapsort) so that they all run in $O(\log n)$ time each. The question that an implementer of Dijkstra's algorithm would like to know is whether it is possible to implement decrease-key more efficiently, ideally to run in just $O(1)$ time. It is not known how to do this in the worst case, but it is possible to perform decrease-key operations in $O(1)$ *amortized time*, which is sufficient when the data structure is used within an algorithm.

The most famous heap structure supporting decrease-key in $O(1)$ amortized time is the *Fibonacci heap*, which was discovered by Michael Fredman and Robert Tarjan in 1984. Unfortunately, Fibonacci heaps are rather complicated structures to describe and analyze. People have studied simpler alternatives. In this lecture, we will present such a structure called a *quake heap*. It was discovered by Timothy Chan around 2013. Our description of the data structure is a bit different and more detailed that Chan's original description (which is only 5 pages long!), but the ideas are essentially the same.

But not "increase-key"? Take note that heaps are asymmetric with respect to key orders. While decrease-key can be implemented to run in $O(1)$ amortized time (assuming a min-heap), there is no min-heap (of which I am aware) that supports the complementary operation of *increase-key* is $O(1)$ amortized time.

Quake-Heap Specifications: Before listing the Quake Heap operations, we should first discuss how to specify the element on which decrease-key and delete are to be applied. Unlike dictionaries, heaps do not support fast searching. Thus, if we insert a key x, there is no efficient way to later find out where it is in the heap. For this reason, the insert function returns a reference to the node containing x, called a *locator*. When we later want to refer to a key, we do so through its locator.

`Locator r = insert(Key x):` Insert the key x into the heap, and return a reference indicating its location in the heap.

`Key x = extract-min():` Remove the item with the minimum key x from the heap, returning its key value.

`void decrease-key(Locator r, Key y):` Decrease the key of the item referenced by r to y. (If y is larger than the current key an exception is thrown.)

These are the most basic operations. Note that, unlike a binary search tree, duplicate keys are allowed, and the extract-min operation may select among ties arbitrarily. We shall see that the quake heap implements insert and decrease-key in $O(1)$ worst-case time and extract-min is $O(\log n)$ amortized time. Extract-min can take $O(n)$ time in the worst case.

Why is it called "quake heap"? The reason is that the data structure allows itself to slowly go out of balance. When it determines that it is very badly out of balanced, it massively reorganizes itself by "flattening" the entire structure, like a city hit by an earthquake.

Quake Heap Structure: The quake heap is represented as a collection of binary trees, where each node stores a key value. The nodes of these trees are organized into *levels*. All the leaves reside on level 0, and each key in the heap is stored in one leaf of some tree (the shaded blue square nodes in Fig. 1). Each internal node has a left child and an optional right child. Its key value is that of its left child. If the right child exists, its key value is greater than or equal to the left child. Thus, the root of each tree holds the smallest key value in the tree, which is that of the leftmost leaf.

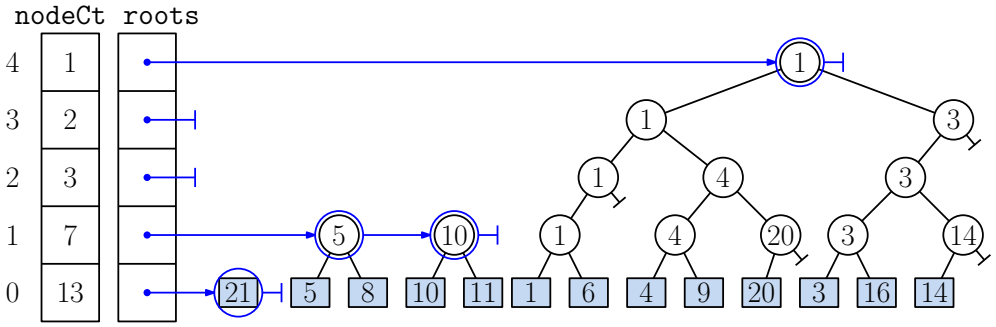


Fig. 1: An example of a quake heap consisting of four trees, storing a total of 13 keys. (As usual, values are omitted.) On the left we show arrays of node counts and roots lists.

Each node stores its key, child pointers, parent pointer, and its level. We maintain two additional arrays organized by level. First, `roots` is an array of linked lists storing the roots of each level. (In Fig. 1 these are shown as singly linked lists in blue, but it would be better to use a doubly linked list for fast insertion and deletion). Second, `nodeCt` is an array of integers storing the total node count (not to be confused with the root count) at each level.

Linking and Cutting: Here are a few useful utility operations, which are applied to manipulate quake heaps. After each operation, the root lists and node counts need to be updated. See the following code fragments for details.

`void make-root(Node u):` Converts node u into a root by setting its parent link to `null` and adding it to the list of roots.

`Node trivial-tree(Key x):` Creates a trivial single-node tree storing the key `x`.

`Node w = link(Node u, Node v):` Link two root nodes `u` and `v` at the same level by joining them under a common root one level higher. The root with the smaller key goes on the left side of the new parent (see Fig. 2).

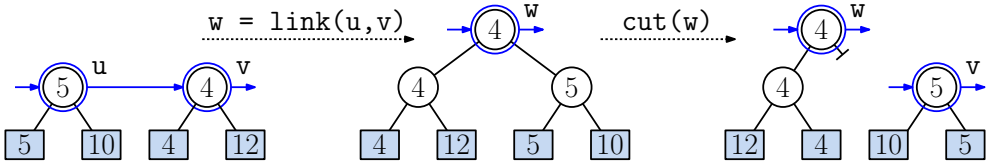


Fig. 2: The operations (a) link and (b) cut.

`void cut(Node w):` If `w`'s right child is `null`, then do nothing. Otherwise, cut the link between `w` and right child, causing this child to become the root of a new tree one level lower. Node `w` now has just a single left child (see Fig. 2).

Trivial Tree and Link Utilities

```
void make-root(Node u) {                        // make u a root node
    u.parent = null                             // null out parent link
    add u to roots[u.level]                     // add it to the list of roots
}

Node trivial-tree(Key x) {                      // create a trivial single-node tree
    Node u = new leaf node with key=x at level=0
    nodeCt[0] += 1                              // increment node count
    make-root(u)                                // make it a root
    return u
}

Node link(Node u, Node v) {                     // link u and v into new tree
    int lev = u.level + 1                       // new node's level
    if (u.key <= v.key)                         // u's key is smaller?
        Node w = new Node(u.key, lev, u, v)     // new root with u's key
    else
        Node w = new Node(v.key, lev, v, u)     // new root with v's key
    nodeCt[lev] += 1                            // increment node count
    u.parent = v.parent = w                     // w is the new parent
    return w
}

void cut(Node w) {                              // cut off w's right child
    Node v = w.right;
    if (v != null) {
        w.right = null;                         // cut off v
        makeRoot(v);                            // ... and make it a root
    }
}
```

Observe that if properly implemented, all three operations can be performed in $O(1)$ time

Quake Heap Updates: Using the above utilities, we will show how to perform the various quake-heap operations. The overall design philosophy is to be as "lazy" as possible with all the operations except for `extract-min`, which is responsible for maintaining proper structure.

`Locator r = insert(Key x):` Create a new single-node tree `x` and return a reference to it. (In general, there should both a key and value, but we omit values here.)

`void decrease-key(Locator r, Key newKey):` Let `oldKey` denote the key value at the node indicated by leaf node `r`. Starting at this leaf node, walk up the path of nodes containing `oldKey`, updating their key values as we go (see Fig. 3). We stop when we first encounter a node having a different key value. Generally, we may be out of heap order with respect to this node, and so we apply cut at this point (see Fig. 3, right)).

As described this operation takes time proportional to the height of the tree, but with some cleverness, it is possible to implement it in $O(1)$ time. We will discuss this below.

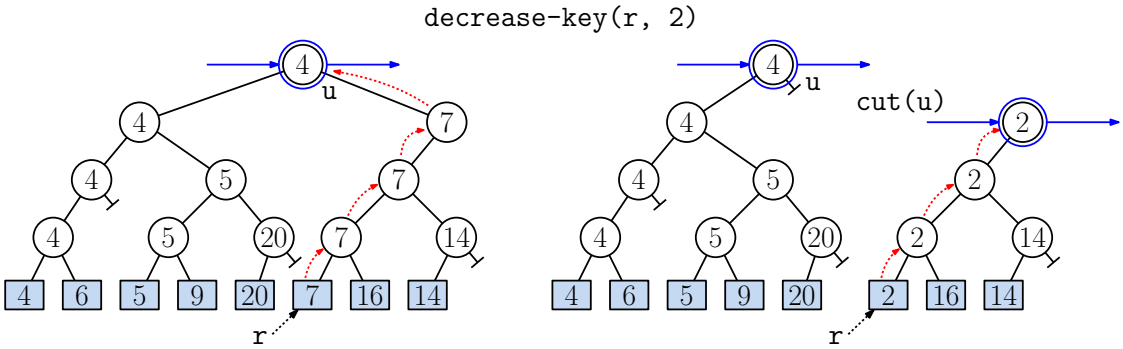


Fig. 3: The operation `decrease-key(r,2)`, which decreases key 7 to 2.

Insert and Decrease-Key

```
Locator insert(Key x) {                          // insert key x in heap
    Node u = trivial-tree(x)                     // create a one-node tree storing x
    return new Locator(u)                        // (the locator stores a link to u)
}

void decrease-key(Locator r, Key newKey) {       // decrease key to newKey
    Node u = r.get()                             // leaf node to be changed
    Node uChild = null                           // u's child on path
    do {
        u.key = newKey                           // update key value
        uChild = u; u = u.parent                 // move up a level
    } while (u != null && uChild == u.left)      // until end of left path
    if (u != null) cut(u)                        // cut us off from parent
}
```

`Key x = extract-min():` This operation is the most complex because it is responsible keeping the data structure in good balance. It involves the following steps:

Find minimum root: First, we visit all the roots of all the trees (traversing the `roots` lists) and find the one with the smallest key. (Among 8, 4, and 7 in Fig. 4(a), this is 4.) Let this root node be `u` and let `x` be the associated (minimum) key.

Delete left path: Next, the function `delete-left-path(u)` traverse the path from `u` down to its leftmost leaf, which contains `x`, and we remove all the nodes along this path (see Fig. 4(b)). As a result, we may obtain a number of new trees. (In Fig. 4(c), we have new trees rooted at 5, 20, and 9.)

Merge trees: We do not want to have too many trees, so we next apply a tree-merging step. The function `merge-trees()` works bottom-up. Whenever we see two trees of the same level, we link them, thus creating a tree at the next higher level. We repeat this until each level has either zero or one trees. (In Fig. 4(d) we have redrawn the trees for clarity. We merge 8 and 20 on level 1, which produces a node 8 on level 2. We merge 5 with 8 on level 2, resulting in node 5 on level 3 (see Fig. 4(e)).

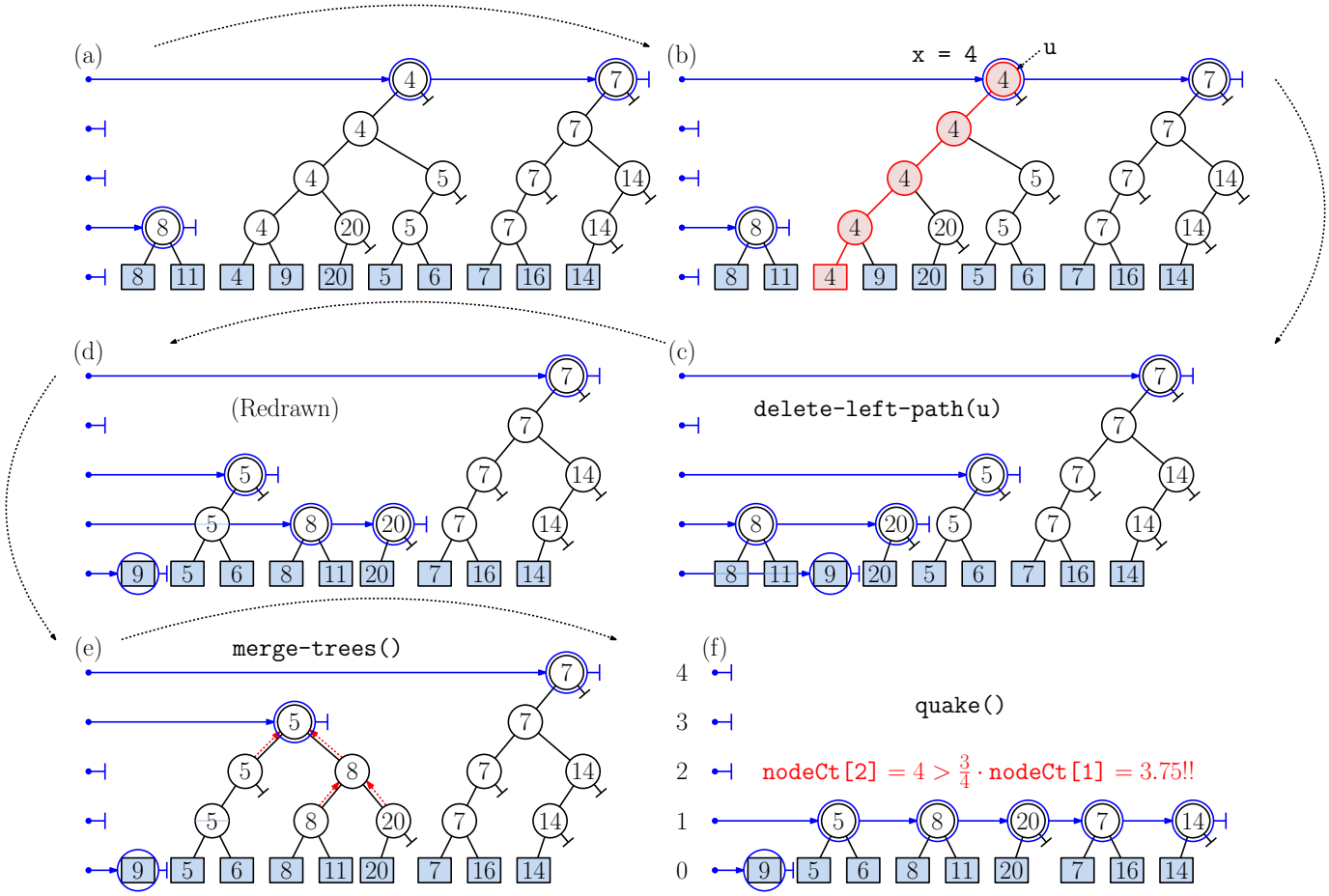


Fig. 4: The operation `extract-min()`.

Quake: When deleting nodes or performing cuts, we tend to produce more and more nodes that have just a single child. To remedy this, we the `quake()` function searches for the smallest level (if any) such that `nodeCt[lev+1]` $> \frac{3}{4} \cdot$ `nodeCt[lev]`. If we find such a level, we remove all nodes at levels `lev` $+ 1$ and higher.[1] (In Fig. 4(e) we see that `nodeCt[2]` $= 4 > \frac{3}{4} \cdot$ `nodeCt[1]` $= \frac{3}{4} \cdot 5 = 3.75$ and in Fig. 4(f) we show the result removing all nodes at level 2 and higher. This generates 5 new trees, one for each node on level 1.)

Result: Finally, return the key from the first step as the final result.

The entire extract-min process is illustrated in the code block below. The member variable `nLevels` stores the number of levels in the data structure. Ideally, the number

[1] By the way, there is nothing magic about the number $\frac{3}{4}$. The analysis works for any constant α, where $\frac{1}{2} < \alpha < 1$.

of levels will expand as more items are added to the heap (e.g., through the use of a Java ArrayList), but here we just assume it is fixed.

Extract-Min and Utilities

```
Key extract-min() {                                 // extract the minimum key
    Node u = find-root-with-smallest-key()          // find the min root (exercise)
    Key result = u.key                              // final return result
    delete-left-path(u)                             // delete entire left path
    remove u from roots[u.level]                    // remove u as a root
    merge-trees()                                   // merge tree pairs
    quake()                                         // perform the quake operation
    return result
}

void delete-left-path(Node u) {                     // delete left path to leaf
    while (u != null) {                             // repeat all the way down
        cut(u)                                      // cut off u's right child
        nodeCt[u.level] -= 1                        // one less node on this level
        u = u.left                                  // go to the left child
    }
}

void merge-trees() {                                // merge trees bottom-up in pairs
    for (int lev = 0; lev < nLevels-1; lev++) {     // process levels bottom-up
        while (roots[lev] size is >= 2) {           // at least two trees?
            Node u = remove any from roots[lev]     // remove any two
            Node v = remove any from roots[lev]
            Node w = link(u, v)                     // ... and merge them
            make-root(w)                            // ... and make this a root
        }
    }
}

void quake() {                                      // flatten if needed
    for (lev = 0; lev < nLevels-1; lev++) {         // process levels bottom-up
        if (nodeCt[lev+1] > 0.75 * nodeCt[lev])     // too many?
            clear-all-above-level(lev)              // clear all nodes above level lev
    }
}
```

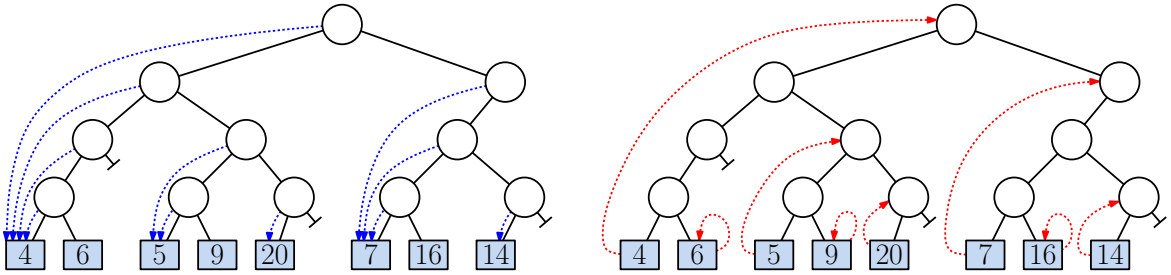
There are two additional utilities. The first is called `find-root-with-smallest-key()`. It searches all the roots for the one with smallest key. The second is called `clear-all-above-level(lev)`. It removes all nodes strictly above level `lev`, and makes all the nodes of this level into roots. This can be done by visiting all the roots at levels `lev+1` and higher, traversing the tree to visit all the nodes on level `lev` and then applying `make-root` to each of them. (We have left the details as an exercise.)

This is everything that you need to know to implement the data structure (but if you want the best performance, see the next section on how to implement decrease-key in $O(1)$ time).

Faster decrease-key: (Optional) As we described it, `decrease-key` requires time proportional to the highest level of the node whose key is being decreased. We will show below that each tree is of height $O(\log n)$, so this is already not bad. But the main reason for presenting

the Quake Heap was to show that `decrease-key` can be performed in $O(1)$ time! So can we speed it up? Yes, this can be achieved with two simple modifications to our implementation:

Left-leaf pointers: Since each key appears in multiple nodes, it takes time to update these values. Instead, we store the key only once, in the associated leaf node. Every internal node along the chain of left-child links that leads to this leaf would normally store this key. Instead, each stores a pointer to the leaf (see Fig. 5(a)). As a result, we need only change the one occurrence of the key in the leaf node. This way, when we wish to change the key, we need only change it in the leaf node in $O(1)$ time.



(a) (b)

Fig. 5: Enhancements for faster `decrease-key()`.

Highest left-side ancestor: To avoid searching level-by-level from a leaf node up to the highest ancestor along the left-child chain, each leaf node stores a link to the highest node containing this key (see Fig. 5(b)).

It is an easy exercise to modify the `link` function to implement these changes in $O(1)$ time. (Note that `cut` does not need to be changed, since it only cuts right child links.) With these two modifications, we can implement `decrease-key` in $O(1)$ time as follows. First, given the leaf node whose key is to be changed, we first change the key in the leaf in $O(1)$ time. Next, we access the leaf's thread, to find its highest ancestor in $O(1)$ time, and we then apply `cut()` on its parent (assuming it has one), which takes $O(1)$ time.

Amortized Analysis: (Optional) For the remainder of the lecture, let's analyze the data structure's running time. On looking at Fig. 4(f) you might protest that this data structure cannot possibly be efficient. The `quake` function could potentially flatten the entire structure all the way down to the leaf level (hence the name "Quake Heap"). We will prove this high degree of flattening is *extremely rare*, and the cost of extract-min, when amortized over a sequence of operations, is only $O(\log n)$.

To make this formal, consider any sequence of m operations (insert, extract-min, decrease-key) starting from an empty heap, and let $n \leq m$ denote the total number of items inserted into the heap over all these operations. For $1 \leq i \leq m$, let T_i, denote *actual time* needed to perform the ith operation. We will ignore constant factors and write 1 for $O(1)$ and $\log n$ for $O(\log n)$.

For any sequence of m operations, the *total time* is $T(m) = \sum_{i=1}^{m} T_i$. The *amortized time* is defined to be the average time over the entire sequence, that is, $A(m) = \frac{1}{m}T(m)$. Here is our main result:

Theorem: Given any sequence of m operations on a Quake Heap involving at most n entries, each insert and decrease-key operation takes $O(1)$ time in the worst-case and extract-min takes $O(\log n)$ amortized time.

How high? Before getting into the proof, we should first derive a bound on how high the trees of the heap might grow.

Lemma: Consider any binary tree, where there are n leaves all at level 0, and for each level $i \geq 0$, the number of nodes at this level, denoted n_i, satisfies the property that $n_{i+1} \leq \alpha n_i$, for some constant $\alpha < 1$. Then the tree has height at most $\log_{1/\alpha} n = O(\log n)$.

We will leave the proof as an easy exercise, but the key idea is that if the number of nodes decreases by a constant factor at each level, we can't keep this up for more than a logarithmic number of levels.

The subtrees that remain after any quake operation satisfy the above lemma for $\alpha = \frac{3}{4}$, and therefore the maximum tree height is $\log_{4/3} n = (\lg n)/\lg(4/3) \approx 2.4 \lg n$. It is easy to verify that the other operations do not increase tree heights, so this bound applies to the entire structure.

Review of Potential-Based Analyses: Our proof employs a *potential-based analysis*. Each instance of the data structure will be associated with a nonnegative function Ψ, called its *potential*. Intuitively, low potential means the structure is well balanced and high values mean it is poorly structured. (We'll present the precise definition below.) For $1 \leq i \leq m$, let Ψ_i denote the value of the potential for the data structure after the ith operation, and let Ψ_0 denote the initial potential. Define the *change in potential* $\Delta_i = \Psi_i - \Psi_{i-1}$. The *amortized time* of the ith operation, denoted A_i, is the sum of the actual cost and the change in potential, that is $A_i = T_i + \Delta_i$.

Why are we doing this? Intuitively, some operations (insert and decrease-key) are cheap in the sense that the actual time T_i is small, but they can make the data structure less balanced, resulting in an increase to the potential. On the other hand, some operations (extract-min in particular) may take much more time, so T_i can be very large. However, these operations improve the data structure's balance, meaning that potential decreases by quite a bit. Thus, there is a trade-off between operations that are cheap (but sloppy) and operations that are expensive (but beneficial).

What do the individual amortized times A_i have to do with the overall amortized time $A(m)$?. To see this, let's write out the sum of the amortized time:

$$\begin{aligned}
\sum_{i=1}^{m} A_i &= \sum_{i=1}^{m}(T_i + \Delta_i) = \sum_{i=1}^{m}(T_i + (\Psi_i - \Psi_{i-1})) = \sum_{i=1}^{m} T_i + \sum_{i=1}^{m}(\Psi_i - \Psi_{i-1}) \\
&= \sum_{i=1}^{m} T_i + (-\Psi_0 + \cancel{\Psi_1} - \cancel{\Psi_1} + \cancel{\Psi_2} - \cancel{\Psi_2} + \cdots - \cancel{\Psi_{m-1}} + \Psi_m) \\
&= \sum_{i=1}^{m} T_i + (\Psi_m - \Psi_0) = T(m) + (\Psi_m - \Psi_0).
\end{aligned}$$

Therefore, the sum of amortized times is equal to the total time plus the overall increase in potential. We will see that the potential of an empty structure is zero, so $\Psi_0 = 0$. Since Ψ is

nonnegative, we conclude that $\Psi_m - \Psi_0 \geq 0$, and therefore

$$A(m) \;=\; \frac{1}{m}T(m) \;\leq\; \frac{1}{m}\big(T(m) + (\Psi_m - \Psi_0)\big) \;=\; \frac{1}{m}\sum_{i=1}^{m} A_i.$$

So in summary, in order to bound the total amortized cost $A(m)$, it suffices to bound the individual amortized costs A_i. The above holds for any data structure. Next, we will show that for the Quake Heap, insert and decrease-key operations, $A_i = O(1)$, and for extract-min $A_i = O(\log n)$.

Quake-Heap Amortized Analysis: In order to bound the amortized cost of an operation, we need to define our potential function. Ideally, our tree would consist of just a single tree (a single root node), and all internal nodes have two children. We say that an internal is *bad* if it has only one child. Consider the tree at any given moment. Let N denote the current number of nodes, R the current number of root nodes, and B the current number of bad nodes. We want to penalize heaps that have multiple roots and lots of bad nodes. Define our potential to be $\Psi = N + 2R + 4B$. (Our definition is a bit different from Chan's, but the analysis is essentially the same.)

Now, let's consider what happens when we perform an operation. Let $\Psi' = N' + 2R' + 4B'$ denote the new values just after completing this operation. We are interested in the actual work T and the change in the potential. Define the change in the number of nodes to be $\Delta N = N' - N$, and define ΔR, ΔB, and $\Delta\Psi$ analogously.

`insert`: Insert takes $T = O(1)$ actual time (just create a node and add it to the leaf-level roots list). We get one new node and one new root, so $\Delta N = \Delta R = 1$, and $\Delta B = 0$. Thus, (ignoring constants) the amortized cost is $A = T + \Delta\Psi = 1 + (1 + 2 \cdot 1 + 4 \cdot 0) = 4$, which is $O(1)$.

`decrease-key`: Assuming fast decrease-key, this operation can be implemented in $T = O(1)$ actual time. The number of roots and number of bad nodes each increase by at most 1. The number of nodes does not change. Therefore, amortized cost is $A = T + \Delta\Psi \leq 1 + (0 + 2 \cdot 1 + 4 \cdot 1) = 8 = O(1)$.

`extract-min`: As might be expected, this is the most complex to analyze. Recall that the maximum level number is $O(\log n)$. Let's ignore the constant factor, and just call this "$\lg n$".

To complete this part of the analysis, we will show that each of its basic elements has an amortized cost of at most $O(\log n)$.

Find-min-root and delete-left-path: We will show that the amortized cost of these operations combined is $R + O(\log n)$. First, observe that finding the minimum key takes actual time proportional to the number of roots R. The delete-left-path operation takes actual time proportional to the maximum tree height, which we have shown to be $\lg n$ (ignoring constant factors). Thus, the total actual time is $T \leq R + O(\log n)$.

How does the potential change? In the process of performing cuts for delete-left-path, we have only decreased the number of nodes and number of bad nodes (both of which are beneficial, but we'll ignore them). We have also converted up to $\lg n$ nodes into new roots. Thus, the increase in potential is at most

$$\Delta\Psi \;=\; \Delta N + 2\Delta R + 4\Delta B \;\leq\; 0 + 2\lg n + 0 \;=\; O(\log n).$$

Therefore, the total amortized cost is $T + \Delta\Psi = R + O(\log n)$.

Merge-trees: The R term in the above amortized cost can be very large, but merge-trees comes to our rescue. After running it, we have at most one root at each of the $\lg n$ levels, that is, $R' \leq \lg n$. Thus, just considering the root portion of the potential, the change is at most $\Delta R = R' - R \leq (\lg n) - R$. (When R is very large, this represents a large decrease.) Let's apply this potential drop to pay for extra R term in find-min and delete-left-path. We have a total amortized cost of

$$A \;=\; T + \Delta\Psi \;\leq\; (R + \lg n) + (\lg n) - R \;=\; 2\lg n \;=\; O(\log n).$$

Of course, we also need to account for the time to perform merge-trees, but we claim that its amortized time is zero. To see why, observe that each time we merge a pair of trees (which takes $O(1)$ time), we create one new node (which is also a root node), but we have also eliminated two root nodes. We never create single-child nodes. So, the contribution to the change in potential is

$$\Delta\Psi \;=\; \Delta N + 2\Delta R + 4\Delta B \;=\; 1 + 2(1-2) + 4\cdot 0 \;=\; -1.$$

The amortized time for merge-trees is $T + \Delta\Psi = +1 - 1 = 0$.

Quake: We claim that the amortized time for this operation is also zero. Suppose that a quake occurs at level j, causing us to remove all the nodes at levels $j+1$ and higher. Let n_j denote the number of nodes at level j (assuming that delete-left-path and merge-trees have already taken place). Let's define $n_{>j}$ to be the total number of nodes strictly above level j. The actual work is proportional to the number of nodes removed, that is, $T \leq n_{>j}$. Since these nodes all are gone, the change in the number of nodes is $\Delta N = -n_{>j}$. Therefore, we have $T + \Delta N = n_{>j} - n_{>j} = 0$.

Each node at level j becomes a new root, so the increase in the number of roots is at most n_j. (We have certainly lost roots at higher levels, but we can ignore these since they only make our potential change better.)

Most importantly, we have eliminated a lot of bad nodes. Let b_j denote the number of bad nodes at level j. Every node at level $j+1$ could have potentially two children at level j, but each bad node has one less child, so we have $n_j \geq 2n_{j+1} - b_{j+1}$. (The number could be higher, because we may have roots at level n_j.) Equivalently, we have $b_{j+1} \geq 2n_{j+1} - n_j$. In order for quake to be triggered, we know that $n_{j+1} > \frac{3}{4}n_j$, and therefore $b_{j+1} > 2\big(\frac{3}{4}n_j\big) - n_j = \frac{1}{2}n_j$. All of these bad nodes are now gone, so we have $\Delta B \leq -b_{j+1} < -\frac{1}{2}n_j$. Recalling that R increased by at most n_j, it follows that the net change in $2\Delta R + 4\Delta B$ is at most $2n_j - 4\big(\frac{1}{2}n_j\big) = 0$.

The amortized cost is $T + \Delta\Psi = (T + \Delta N) + (2\Delta R + 4\Delta B)$, but we have just shown that both of these values are at most zero. Therefore, the amortized cost of Quake is at most zero, completing the amortized analysis. Whew!