# Recitation Note - CS430 Fall 2014

Taeho Jung
Illinois Institute of Technology

Oct 24, 2014

- I do not guarantee I will prepare a note for every recitation.
- I skip the insertion/deletion in 2-3-4 tree, which are interesting but not closely related to this exercise. If you are interested, refer to Chapter 18 in CLRS or `http://en.wikipedia.org/wiki/B-tree`
- As the correctness is obvious, I omit the correctness analysis.

## Exercise 19.4 on page 529-530

Implement the following operations for 2-3-4 heaps. All of the operations should run in $O(\lg n)$ time. In a 2-3-4 heap, any internal node can have 2,3, or 4 children, and all leaves are at the same depth. Besides this property inherited from 2-3-4 trees, 2-3-4 heaps have their own characteristics described below:

1. The root node $r$ stores an attribute $r.height$, which is the height of the heap.
2. Every internal node $x$ (including the root $r$) stores an attribute '$x.small$' which is the value of the minimum key in the sub-tree rooted by $x$.
3. Only leaf nodes contain keys, and each leaf node contains exactly one key.

### MINIMUM

Starting from the root, recursively visit the child whose *small* value is same as $r.small$ until the leaf node.

Finding the leaf node containing the smallest key is $O(height)$, where the height of the 2-3-4 heap *height* is at most $\lg n$. Therefore, the total complexity is $O(\lg n)$.

### DECREASE-KEY

Given the leaf node $x$, change its key to $k$. Then, update the $x.p.small$ to $k$ if $k < x.p.small$. Then, recursively update the grandparents' *small* values if parents' *small* value is less than the ones of grandparents.

The above recursive process may propagate from the leaf node to the root node in the worst case, therefore the complexity is $O(\lg n)$.

### INSERT

Given the node $x$ with key $k$ to insert, append it to any node at the height 1. Then, we have two cases:

1. If $x$'s parent does not have more than 4 children after the insertion, do nothing but recursively update the small values if necessary (same as DECREASE-KEY).

2. If the $x$'s parent has 5 children after the insertion, split the parent into two nodes, and randomly select 3 to append to the first parent and the remaining 2 to the other parent. Then, update the two parents' *small* values correspondingly. After the split, the grandparent may have 5 children, so recursively split the parent nodes in the same way until the grandparent has less than 5 children or the parent to split is the root. If the split node is the root, add a new root $r'$ above the two new parents, and set $r'.height$ as $height + 1$.

Operations at each split is $O(1)$, and the split process propagates from the leaf to the root in the worst case. Therefore the total complexity in the worst case is $O(\lg n)$.

## DELETE

Given the node $x$ to delete, first delete the key. We also have two cases afterwards:

1. If the parent of $x$ has more than one child after the deletion, recursively update the *small* values of the parents if necessary (same as DECREASE-KEY).

2. If the parent of $x$ has only one child after the deletion, remove the parent node and append its child to any node at the same depth as its original parent, and update the *small* value if necessary. If there is an overflow in the new parent, the same recursive process as in INSERT is applied to split the nodes until there is no overflow. If there is only one parent remaining after removing the old parent, the grandparent of $x$ must be the root of the heap. In this case, simply remove the root, and set the new parent $r'$ as the root of the heap, and set $r'.height$ as $height - 1$.

In the first case, we merged two parents into one, and this is recursively propagated from the leaf to the root in the worst case. We also see the operations incurred in each step is $O(1)$. Therefore, the complexity of DELETE is $O(\lg n)$.

## EXTRACT-MIN

First executes MINIMUM ($O(\lg n)$) and then DELETE ($O(\lg n)$) it. The complexity is obviously $O(\lg n)$.

## UNION

Suppose the height of $T_1$ is $h_1$ and $T_2$ is $h_2$.

1. If $h_1 = h_2$, create a new root node $r'$, and link two roots of $T_1, T_2$ to the $r'$. Set $r'.height = h_1 + 1 = h_2 + 1$, and set $r'.small$ as the smaller *small* of the two roots in $T_1, T_2$.

2. If $h_1 < h_2$, append the entire tree $T_1$ to any node in $T_2$ whose height is $h_1 + 1$. If there is an overflow after the merge, recursively apply the same process as in INSERT to split the nodes.

3. If $h_1 > h_2$, it is symmetric to the above case.

The complexity of case 1 is $O(1)$, but recursive split in case 2 & 3 is $O(\lg n)$. Therefore, the total complexity is $O(\lg n)$.