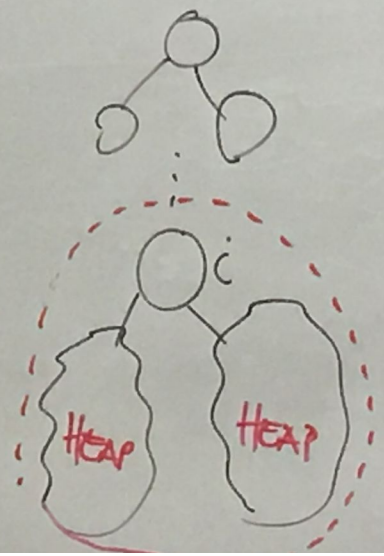
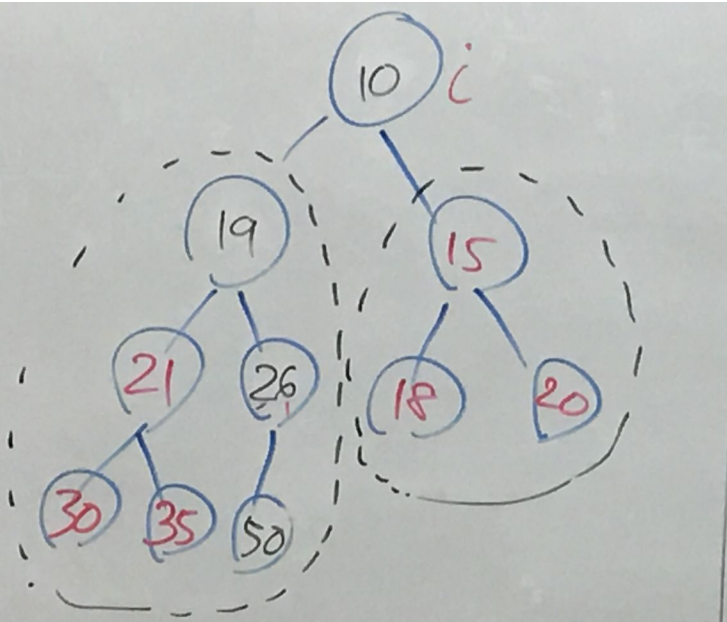
HEAPIFY(A, l, n, i)

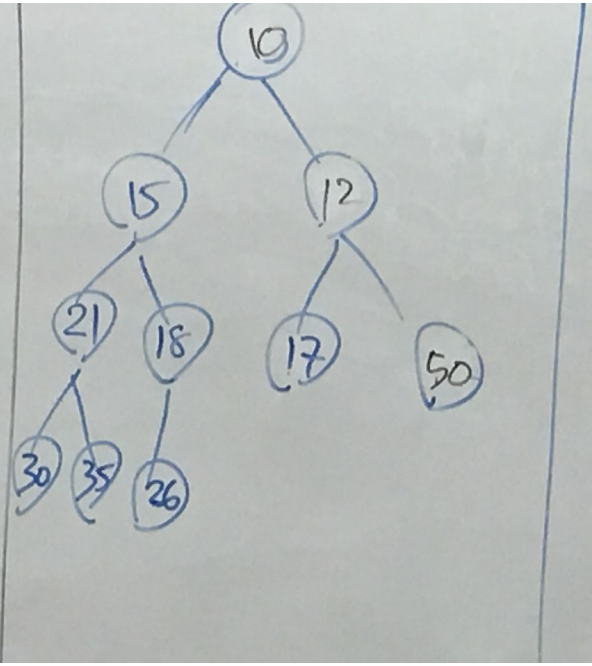



Repeat

if i is a leaf then quit;
if A[i] < A[2i]
and A[i] < A[2i+1]
then quit; else
if A[2i] < A[2i+1] then j = 2i
else {j = 2i+1; SWAP A[i] and A[j];
i = j; }

forever



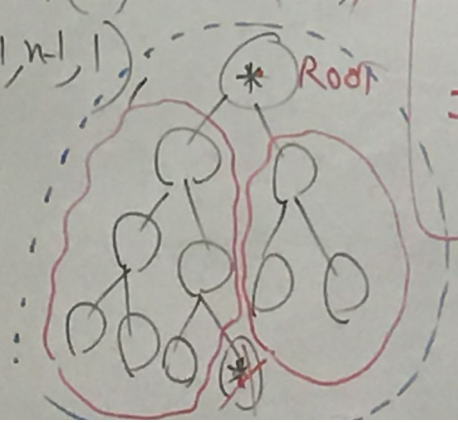




RUN TIME = O( height of the tree rooted at i )

## Delete_Min(A, 1, n):

A(1) = A(n);

HEAPIFY(A, 1, n-1, 1);




RUN TIME = $O(\log n)$.

**THEOREM:** A PRIORITY QUEUE CAN BE IMPLEMENTED S.T EACH OPERATION TAKES $O(\log n)$ TIME. □

COROLLARY:

We can sort $n$ given elements in $O(n \log n)$ TIME. □

HEAP SORT.

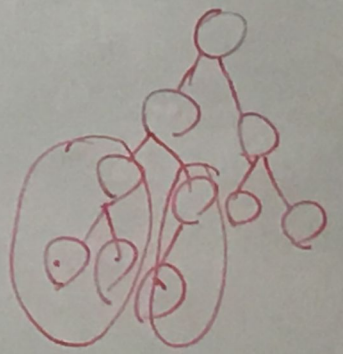
FORMING A HEAP OUT OF $n$ ELEMENTS.

$X = k_1, k_2, \ldots, k_n$.

for $1 \le i \le n$ do $A[i] = k_i$;

for $i = \lfloor \frac{n}{2} \rfloor$ downto 1 do

HEAPIFY$(A, 1, n, i)$;



$2^h = \Theta(n)$.

FACT: The TOTAL RUN TIME is $O(n)$.

PROOF:

At level $i$ there are $\leq 2^{i-1}$ nodes. $1 \leq i \leq h$.

A Call to Heapify at any node in level $i$ will cost us $O(h-i)$ time.

$\Rightarrow$ TOTAL time spent

$$= \sum_{i=1}^{h} 2^{i-1}(h-i)$$

$$= (h-1) + 2\cdot(h-2) + 2^2\cdot(h-3) + \cdots + 2^{h-3}\cdot 2 + 2^{h-2}$$

$$= 2^{h-2}\left[1 + \frac{2}{2} + \frac{3}{2^2} + \frac{4}{2^3} + \cdots + \frac{i}{2^{i-1}} + \frac{(i+1)}{2^i} + \cdots\right] \quad \text{--- (1)}$$

Ratio between successive terms is $\frac{(i+1)}{i}\frac{1}{2}$. This is $\leq \frac{2}{3}$ $\forall i \geq 3$.

Thus the sum within parantheses is

$$\leq 2 + \frac{3}{2^2}\left[1 + \left(\frac{2}{3}\right) + \left(\frac{2}{3}\right)^2 + \left(\frac{2}{3}\right)^3 + \cdots\right]$$

$$= 2 + \frac{3}{4}\left[\frac{1}{1 - \frac{2}{3}}\right] = 2 + \frac{9}{4} = \frac{17}{4}.$$

Substituting this in (1),

the TOTAL TIME FOR HEAP FORMATION is $\leq 2^{h-2} \cdot \frac{17}{4} = \Theta(n)$

Since $2^h = \Theta(n)$.

□

# DICTIONARY

## USING A 2-3 TREE.

IN

DEFN. A TWO THREE TREE:

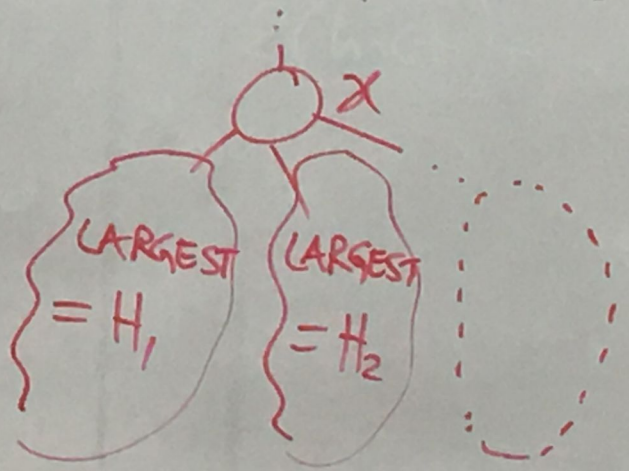
(1) EACH NON LEAF HAS AT LEAST 2 & AT MOST THREE CHILDREN.

(2) All the leaves are at the same level.

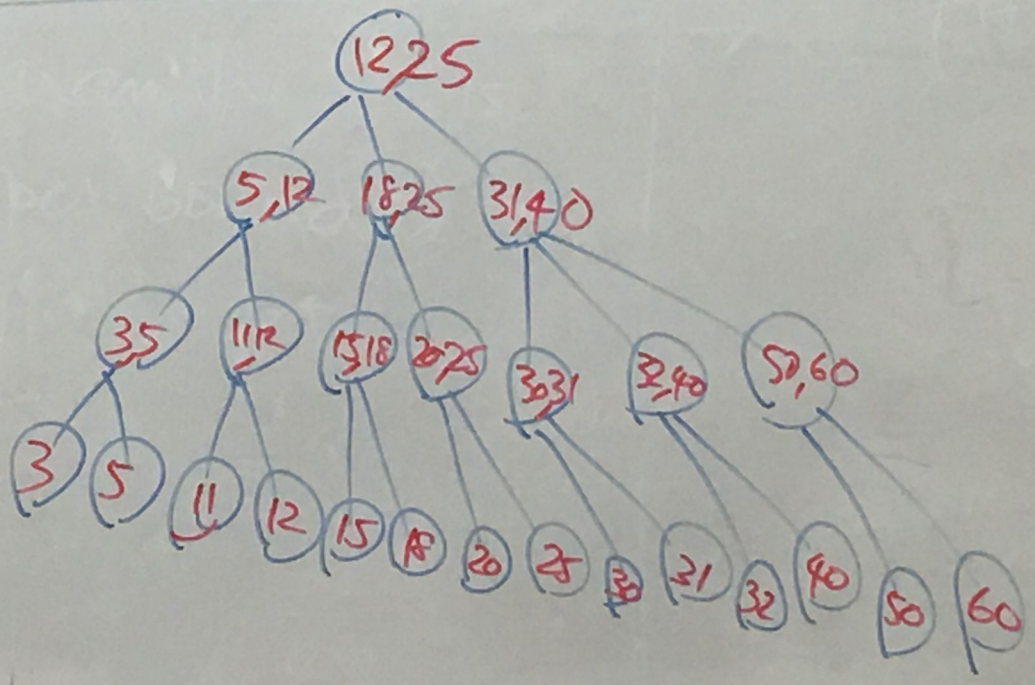
(3) Keys are stored one per leaf.

(4) Keys in the leaves are in sorted order.

5) EACH NON LEAF $x$ stores two keys, $H_1(x)$ and $H_2(x)$.

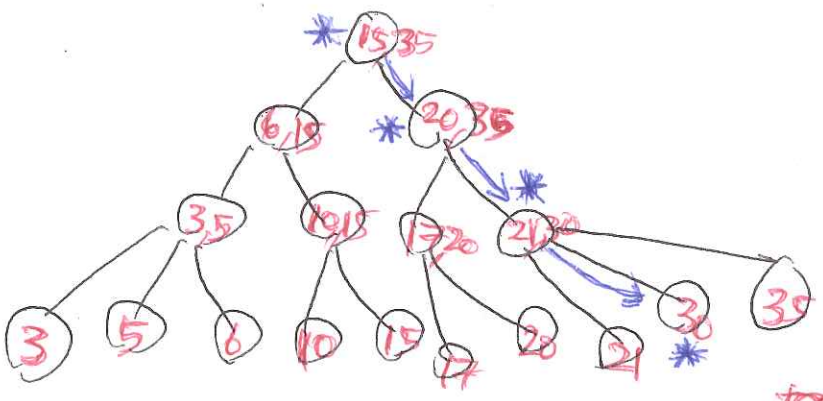



Example:



# SEARCHING IN A 2-3 TREE:

We use $H_1$ and $H_2$ values as guidelines. If we search for $x$, we compare $x$ with $H_1(\text{root})$.
If $x \leq H_1(\text{root})$ we move to the first child of the root & proceed recursively. If $H_1(\text{root}) < x \leq H_2(\text{root})$ we move to the second child & proceed recursively. If $x > H_2(\text{root})$ we move to the third child (if any) & proceed recursively. An Example:




SEARCH (28):

SINCE
$15 < 28 \leq 35$
WE MOVE TO the SECOND CHILD FROM the ROOT;

The SEARCH TERMINATES IN THE LEAF WITH 30.
WE REALIZE that 28 IS NOT IN THE TREE.

# CSE 3500 Algorithms and Complexity – Fall 2016
# Lecture 5: September 13, 2016

In the last lecure we started our discussion on data structures. Trees and related facts and definitions were presented. A heap was introduced. We showed how to perform Fin_Min (in $O(1)$ time) and Insert (in $O(\log n)$ time) operations.

## Heapify and Delete

- To process the Delete_Min operation, we define another operation called Heapify$(A, i, n)$, where $i$ is any node in a complete binary tree stored as $A[1:n]$.
- Heapify is defined for any node $i$ in a complete binary tree where the subtree rooted at the node $2i$ (i.e., the left child of $i$) is a heap and the subtree rooted at the node $2i+1$ is also a heap. When the execution of Heapify$(A, i, n)$ is completed, the subtree rooted at the node $i$ will also become a heap.
- Here is an algorithm that implements Heapify$(A, i, n)$:

```
repeat
    if A[i] < A[2i] and A[i] < A[2i + 1] or i is a leaf then quit;
    else
        Let j = 2i if A[2i] < A[2i + 1] else let j = 2i + 1;
        Swap A[i] with A[j]; i = j;
forever
```

  In the above algorithm the special case when the node $2i+1$ is missing can be handled appropriately. In this case in line 2 we only have to compare $A[i]$ and $A[2i]$ and in line 4, we set $j = 2i$.

- Note that Heapify$(A, i, n)$ takes $O(\log n)$ time since we spend only $O(1)$ time per level of the tree in the worst case.
- The Delete_Min() operation can be handled as:

  $A[1] = A[n]$;
  Heapify$(A, 1, n-1)$

- The run time for Delete_Min() is also $O(\log n)$.

# Sorting

- We can sort $n$ given elements using a priority queue as follows. Let $X = k_1, k_2, \ldots, k_n$ be the input.

  Start with an empty priority queue $Q$;
  **for** $i = 1$ **to** $n$ **do**
  Insert $k_i$ into $Q$;
  **for** $i = 1$ **to** $n$ **do**
  Output Delete_Min();

- Each insert and each delete operation takes $O(\log n)$ time. We perform a total of $n$ inserts and $n$ deletes. Thus the total run time of this sorting algorithm is $O(n \log n)$.

- The above algorithm is called heap sort and it is one of the asymptotically optimal algorithms known for sorting.

- **Fact:** If we are given a sequence $X = k_1, k_2, \ldots, k_n$, we can build a heap consisting of these elements in $O(n)$ time. Here is an algorithm:

  **for** $i = 1$ **to** $n$ **do**
  $A[i] = k_i$;
  **for** $i = \lfloor n/2 \rfloor$ **downto** 1 **do**
  Heapify$(A, i, n)$;

- We can show that the above algorithm runs in $O(n)$ time as follows. We call Heapify on every nonleaf in the complete binary tree. When we call Heapify on a node in level $i$ of the tree, the time spent is $O(h - i)$ where $h$ is the height of the tree. Note that $2^h = \Theta(n)$.

  There are $2^{i-1}$ nodes in level $i$ of the tree (for $1 \leq i < h$) and on each such node we spend $O(h - i)$ time. Thus the total time spent in forming the heap is proportional to

$$\sum_{i=1}^{h-1} 2^{i-1}(h-i) = 2^{h-2} + 2^{h-3}\,2 + 2^{h-4}\,3 + \cdots = 2^{h-2}\left[1 + \frac{2}{2} + \frac{3}{2^2} + \frac{4}{2^3} + \cdots \frac{i}{2^{i-1}} + \frac{i+1}{2^i} + \cdots\right]. \tag{1}$$

  Note that the ratio between two successive terms in the above series is $\frac{i+1}{i}\frac{1}{2}$. This ratio is $\leq \frac{2}{3}$, for all $i \geq 3$. Thus the sum of the series (within parantheses) of Equation 1 is

$$\leq 2 + \frac{3}{2^2}\left[1 + \left(\frac{2}{3}\right) + \left(\frac{2}{3}\right)^2 + \left(\frac{2}{3}\right)^3 + \cdots\right] = 2 + \frac{3}{4}\left[\frac{1}{1 - \frac{2}{3}}\right] = 2 + \frac{9}{4} = \frac{17}{4}.$$

  Substituting this in Equation 1, the total time for heap formation is $\leq 2^{h-2}\left(\frac{17}{4}\right) = \Theta(n)$.□

# An optimal dictionary implementation

- There are numerous optimal implementations of a dictionary using such structures as red-black trees, AVL trees, a* trees, 2-3 trees, etc.
- Each of the anove stcutures have a height of $O(\log n)$ and there exist algorithms that perform the operations of interest in $O(h)$ time, $h$ being the height of the tree.
- Any tree whose height is $\Theta(\log n)$ can be called a balanced tree.
- We use 2-3 trees in our discussion.

# 2-3 trees

- A 2-3 tree is a tree with the following propoerties: 1) Each non leaf has at least 2 and at most 3 children; 2) All the leaves are at the same level; 3) Each leaf stores a key; 4) The keys in the leaves are in sorted order; and 5) Each non leaf $x$ stores two keys $H_1(x)$ and $H_2(x)$, where $H_1(x)$ is the largest key in the first subtree of $x$ and $H_2(x)$ is the largest key in the second subtree of $x$.

# Searching in a 2-3 tree

- Let $x$ be the key searched for.

```
N = root;
repeat
   if N is a leaf then
      if x equals the key of N then output 'yes' else output 'no';
      quit;
   if x ≤ H1(N) then N = N1 else
      if x > H1(N) and x ≤ H2(N) then N = N2 else
         if N has a third child N3 then N = N3 else
            { output 'no' and quit};
forever
```