FINDING AN ELEMENT ≥ THE MEDIAN.

Let p be the (HIGH) prob. of interest.

Prob. that all the k elements picked are < the MEDIAN is $\leq \left(\frac{1}{2}\right)^k$.

we want this prob to be $\leq (1-p)$.

$\Rightarrow \left(\frac{1}{2}\right)^k \leq (1-p)$

$-k \leq \log(1-p)$

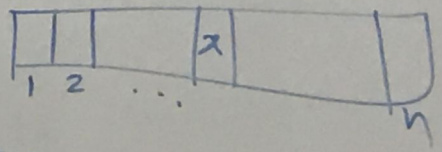
$\Rightarrow k \geq \log\left(\frac{1}{1-p}\right)$.

AN ARRAY AS A DICTIONARY:

INSERT(x): $O(1)$

SEARCH(x): $O(n)$

DELETE(x):

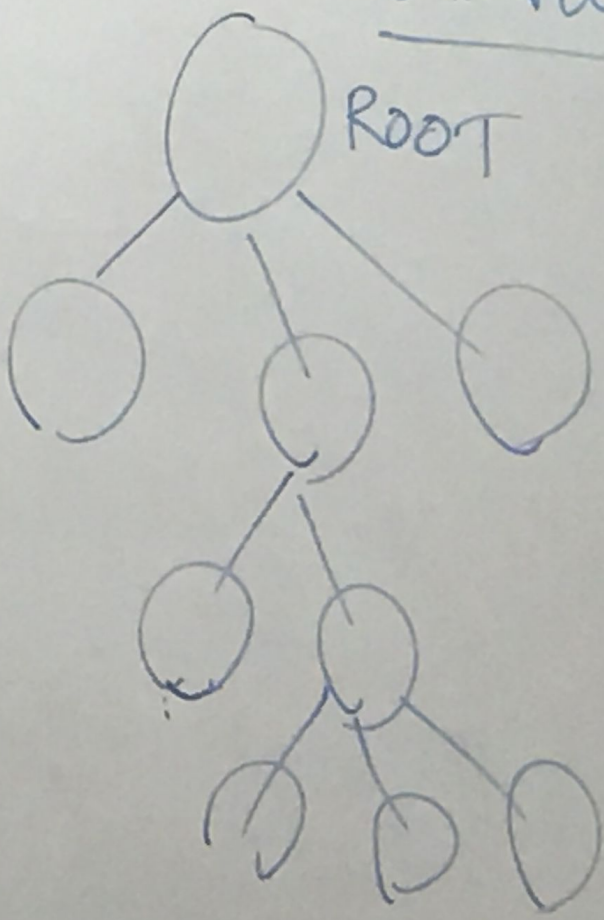


$O(n)$ TIME.

TREES.

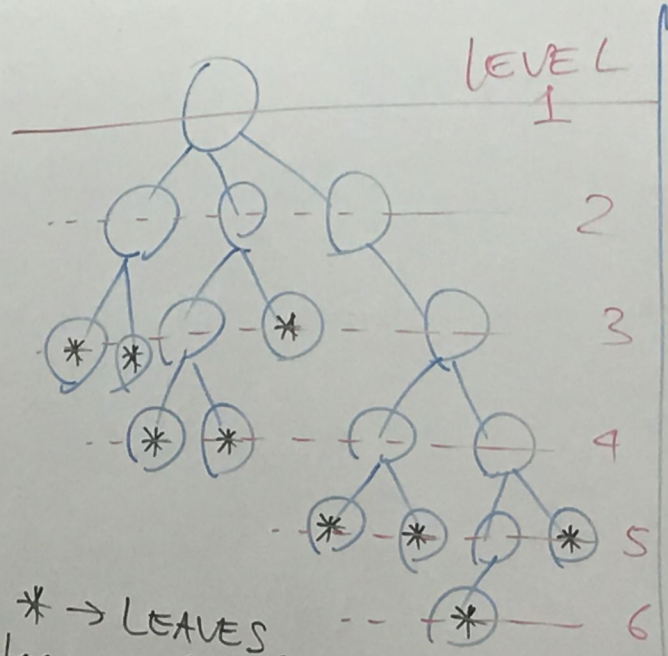
A TREE IS EITHER EMPTY OR IT HAS A SINGLE NODE CALLED THE ROOT, WHERE THE ROOT HAS k CHILDREN $T_1, T_2 \ldots, T_k$, EACH T_i BEING A TREE, $k \geq 2$.

EXAMPLE.



THE LEVEL OF THE ROOT IS ONE.

IF ANY NODE IN A TREE HAS A LEVEL i, THEN ITS CHILDREN WILL HAVE A LEVEL OF (i+1).



* → LEAVES.
HEIGHT = 6.

WE CAN DEFINE THE FOLLOWING FOR ANY NODE: ANCESTORS, DESCENDANTS, PARENT, CHILDREN, SIBLINGS, ...

A NODE WITH NO CHILDREN IS A LEAF OR A TERMINAL NODE.

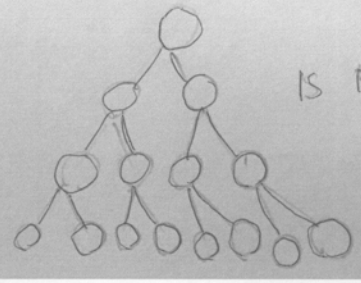
The **HEIGHT** OF A TREE IS the MAXIMUM LEVEL OF ANY NODE IN THE TREE.

DEGREE OF A TREE IS the MAX. # OF CHILDREN That ANY NODE has.

DEGREE OF the example tree = 3.

A TREE WHOSE DEGREE IS 2 IS CALLED A BINARY TREE.

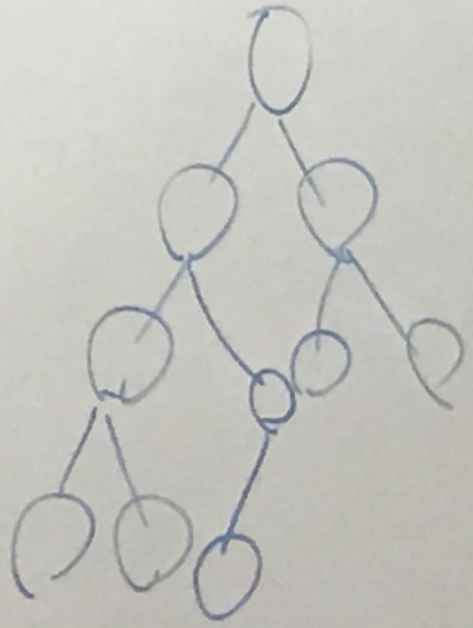
DEFN. A BINARY TREE IS FULL IF EVERY NON LEAF HAS EXACTLY TWO CHILDREN AND ALL THE LAVES ARE IN THE SAME LEVEL.



IS FULL

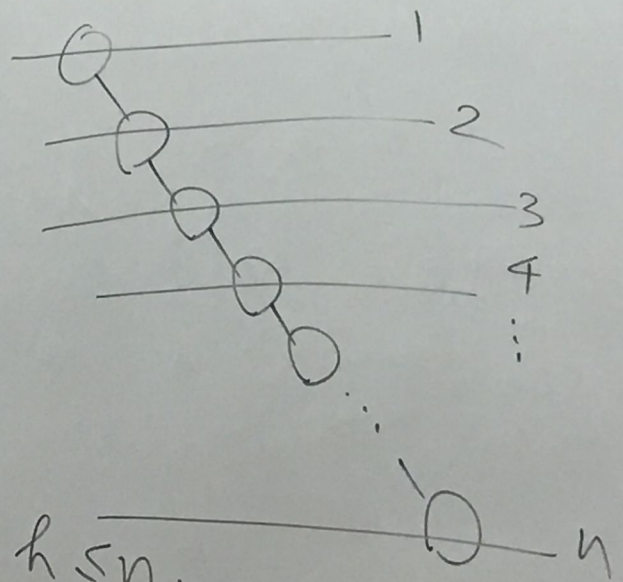
A COMPLETE BINARY TREE IS ALMOST FULL, EXCEPT that there may be some missing nodes in the last level, s.t. the MISSING NODES ARE RIGHT JUSTIFIED.

1.



IS COMPLETE.

Let T be a binary tree whose height is h and which has n nodes.



$h \leq n$.

Max. # of nodes in level 1 = 1.

Max. # of nodes in level 2 = 2.

⋮

Max. # of nodes in level $i = 2^{i-1}$

$\Rightarrow$ Max. # of nodes in a binary tree with a height of h

$$= 1 + 2 + 2^2 + \cdots + 2^{h-1} = 2^h - 1.$$

$\Rightarrow n \leq 2^h - 1.$

$\Rightarrow (n+1) \leq 2^h \Rightarrow \boxed{h \geq \log(n+1)}$

PUT TOGETHER,

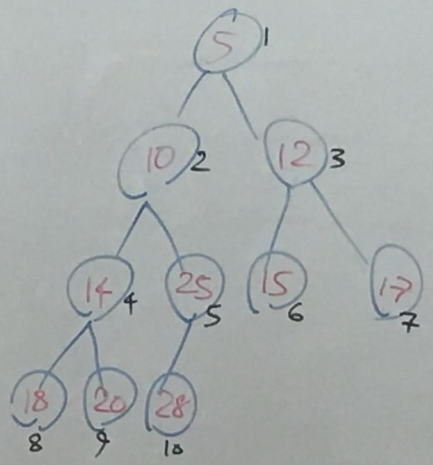
$\log(n+1) \leq h \leq n$

A MIN HEAP IS:

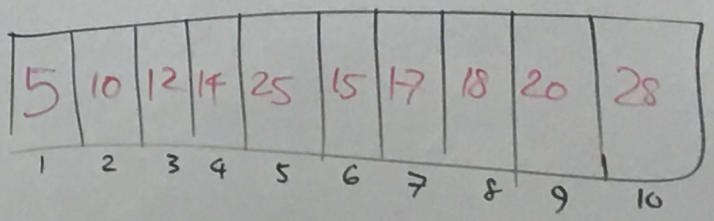
1. A COMPLETE BINARY TREE;
2. EACH NODE HAS A KEY;
3. The KEY AT ANY NODE IS LESS THAN THE KEYS OF ITS CHILDREN.

(HEAP PROPERTY)

Example:



WE CAN REPRESENT A HEAP OF SIZE n USING AN ARRAY A[1:n]



If i IS the LABEL OF A NODE IN A HEAP, THE LABEL OF ITS PARENT IS $\lfloor \frac{i}{2} \rfloor$

The label of its left child is $2i$; the label of the right child is $(2i+1)$.

FINDMIN():

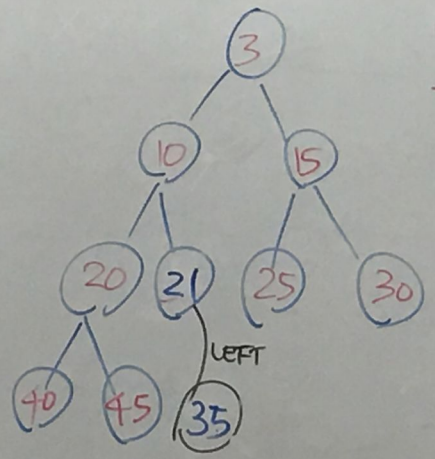
Output A[1];

TAKES O(1) TIME.

INSERT (A, 1, n, x):

/* A[1:n] is the current heap. x has to be inserted */

A[n+1] = x;



INSERT (21)

if $A\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) < x$

then quit;

if not, swap A[n+1] with $A\left[\left\lfloor \frac{n+1}{2} \right\rfloor\right]$;

The Problem now moves to $A\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right)$. PROCEED RECURSIVELY

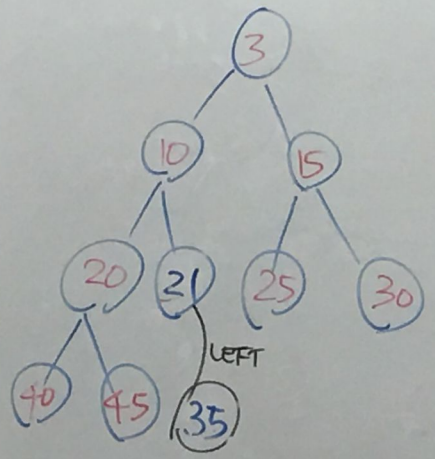
AN EXAMPLE RECURSIVE ALGORITHM:

```
FACTORIAL (n):
    if n=1 then output 1;
    else output n * FACTORIAL (n-1).
```

NOTE: The height of a complete binary tree with n nodes is $\theta(\log n)$.

FOR INSERTING,
WE SPEND O(1)
TIME at any level:

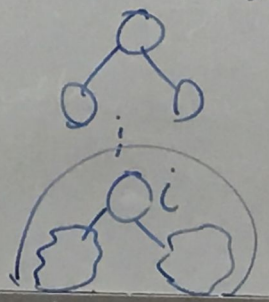
⇒ TOTAL TIME FOR
INSERT IS
O(log n).



INSERT (21)

DECETE_MIN():

TO DO THIS WE
DEFINE AN OPERATOR
HEAPIFY (A, i, n)



CSE 3500 Algorithms and Complexity – Fall 2016
Lecture 4: September 8, 2016

Revision

- In the last lecture we had an introduction to randomized algorithms. We considered the following two problems and devised randomized algorithms: 1) Repeated element identification; and 2) Finding an element $\geq$ the median. For the first problem our randomized algorithm took $\widetilde{O}(\log n)$ time and for the second problem the run time was $O(\log n)$. In both cases we argued that any deterministic algorithm will, in the worst case, need $\Omega(n)$ time.
- We defined a high probability to be a probability that is $\geq (1 - n^{-\alpha})$. However, all the analyses can be done even if we have a different notion of a high probability.
- Let p be the high probability of interest. For the algorithm that we discussed for the problem of finding an element $\geq$ the median, we picked k elements randomly, found and output the maximum of these k elements. Probability that a randomly picked element is $<$ the median is $\leq \frac{1}{2}$. Thus the probability of an incorrect answer from the algorithm is $\leq \left(\frac{1}{2}\right)^k$. We want this probability to be $\leq (1-p)$. This will happen when $k \geq \log\left(\frac{1}{1-p}\right)$.

Data Structures

- A data structure can be thought of as a black box where data can be stored. The black box supports a set of operations. Depending on the type of data and the set of operations supported, we have different data structures.
- Data structures play vital roles in data analysis as well as algorithms design.
- A *dictionary* supports the following three operations:
  1. Search(x): search for the element x in the storage, where x is an arbitrary element;
  2. Insert(x): insert the element x, where x is an arbitrary element;
  3. Delete(x): delete the element x, x being an arbitrary element.
- Example applications for a dictionary: a library system, an employee database, etc.
- A *min priority queue* supports the following operations:

1. Find_Min(): identify the smallest element in the storage;
2. Insert(x): insert the element x, where x is an arbitrary element;
3. Delete_Min(): delete the smallest element.

- A max priority queue can be defined in a similar manner.
- An example application for a priority queue: sorting.
- A dictionary and a priority queue can be imlememnted using arrays and linked lists. In this case each operation takes $O(n)$ time. In each implementation, at least one of the three operations under concern will take $\Omega(n)$ time. We could achieve a better performance using trees.

Trees

- A tree is either empty or it has a node called the root and the root has k children where each child is a tree. Here $k \geq 2$.
- When $k = 2$, we have a binary tree.
- The *level* of the root is 1.
- If the level of a node in a tree is i, then its children have a level of $i + 1$.
- Any node in a tree that has no children is called a *leaf* or a *terminal node*.
- We can define *descendents*, *ancestor*, *siblings*, etc. for any node in a tree.
- The *height* of a tree is defined to be the maximum level of any node in the tree.
- **Fact:** If T is a binary tree with n nodes whose height is h, then, $n \geq h \geq \log(n + 1)$. **Proof:** In a binary tree the maximum number of nodes we can have in level i is 2^{i-1}. This implies that the maximum number of nodes we can have in a binary tree of height h is $1 + 2 + \cdots + 2^{h-1} = 2^h - 1$. In other words, $n \leq 2^h - 1$. I.e., $h \geq \log(n + 1)$. The fact that $h \leq n$ is easy to see. This happens when we have a skewed tree where each node (other than the leaf) has a single child. $\square$
- A binary tree is said to be *full* if every non leaf has exactly two children and all the leaves are at the same level. A binary tree is defined to be *complete* if it is full except that there could be some nodes missing in the last level and the missing nodes in the last level (if any) are right justified. The height of a complete binary tree with n nodes is $\Theta(\log n)$.
- A generic data set can be thought of as consisting of records. Each record has a key that can be used to shape the structure of a tree. In our discussion on data structures we only consider the keys. Keys are stored in the nodes of a tree.

Heap

- An efficient implementation of a priority queue can be achieved using a heap.
- A min heap with n nodes is: 1) a complete binary tree; 2) the root has the smallest key; 3) If a node in a heap has the key k, then its children have keys that are greater than k.
- A heap of size n can be realized as an array $A[1:n]$. $A[1]$ stores the smallest key. For any node i in the heap, we store its left child (if any) in $A[2i]$ and its right child (if any) in $A[2i+1]$. If i is any node in a heap, then its ancestor is $\lfloor i/2 \rfloor$.
- Note that the height of the above heap is $\Theta(\log n)$.
- To process Find_Min(): We output $A[1]$. This takes $O(1)$ time.
- Consider an existing heap $A[1:n]$. To process Insert(x):

  $A[n+1] = x$; $i = (n+1)$;
  repeat
  if $i = 1$ **then** quit;
  $i_1 = \lfloor i/2 \rfloor$;
  if $A[i_1] < x$ **then quit**;
  else
  $A[i] = A[i_1]$; $A[i_1] = x$; $i = i_1$;
  forever
- In the above algorithm for insert, we spend $O(1)$ time per level of the tree. Thus the run time for insert is $O(\log n)$.