Sorting

- Mainly discuss Internal Sorts (without space)
- Briefly discuss external sorting algorithm.

Simple Methods

- Insertion Sort: $\Theta(n^2)$ time.

```
for i:= 2 to n do
    find where a[i] fits among
    a[1...i-1] and move in
```

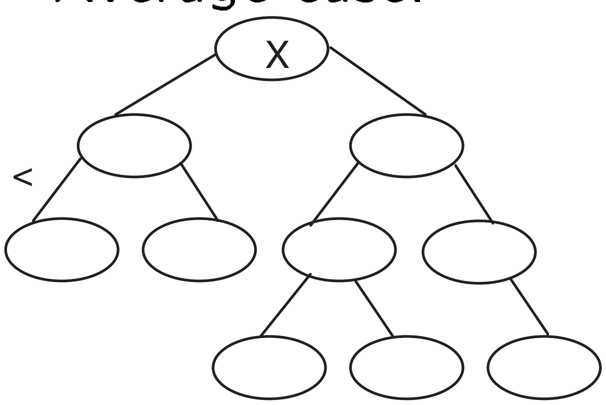
- Bubble sort: $\Theta(n^2)$ time.

```
for i:= 1 to n do
   for j:=1 to n-i do
      Switch A[j] and A[j+1] if
      they are not in order
```

- **Quicksort**

```
Quicksort(A,f,l)
  If l-f is small, then use insertion sort to sort A[f,l].
    else choose an element, x, at random
       Partition set into those
           < x     at left
           = x     middle.
           > x     to right
       Quicksort (those < x);
       Quicksort (those > x).
```

- Worst case is $\Omega(n^2)$.
- Average case:



This is exactly same as (searching for every element in it) a randomly formed binary search tree. $\approx 1.386 n \log n$.

- Improve - median of 3 for pivot element x

- Problem - make sure <u>not</u> to use first or last element: consider almost sorted list!

- Quicksort is fastest internal sorting algorithm (assuming all possible data and using comparison only.)

An application of Quicksort idea.
Problem: Find k-th smallest of n values.

- We can sort then select, but this takes $O(n \log n)$ time.

- Using Quicksort idea, we can achieve $O(n)$ time.

Qselect (k,n,s): select k-th in S and $|S| = n$.

```
Qselect (k,n,s)
   Choose x at random;
   Split S into  < x:  L
                 = x:  E
                 > x:  B
   if |L|<k and |L| + |E| >= k, then x is k-th largest'
      else if |L| >= k then Qselect(k,|L|,L)
      else Qselect(k-|L|-|E|,|B|,B)
```

Cost on Average

$$T(n) = n - 1 + T(i),$$

where $i \in \{0, \ldots, n\}$, taking each value with probability $\frac{1}{n+1}$.

This gives $O(n)$ in on average, but we omit the calculation.

• **Selection Sort**. Sort by repeatedly selecting max of those left. This algorithm also uses $O(n^2)$ time for trivial implementation:

```
For i:=1 to n-1 do
    find max of those in spots i...n
    and swap it with the element in A[i].
```

• This algorithm takes $\Omega(n^2)$ time because to select the maximum at each step takes $\Omega(n)$ time.

• We will find a scheme that allows us to use only $\log n$ comparisons to find the max element. Then we will have a fast $O(n \log n)$ selection algorithm. (Heapsort).

How do we do it? We need a data structure to support us.

- This data structure can't use pointers, since that costs too much space. (Remember, we are doing internal sort.)

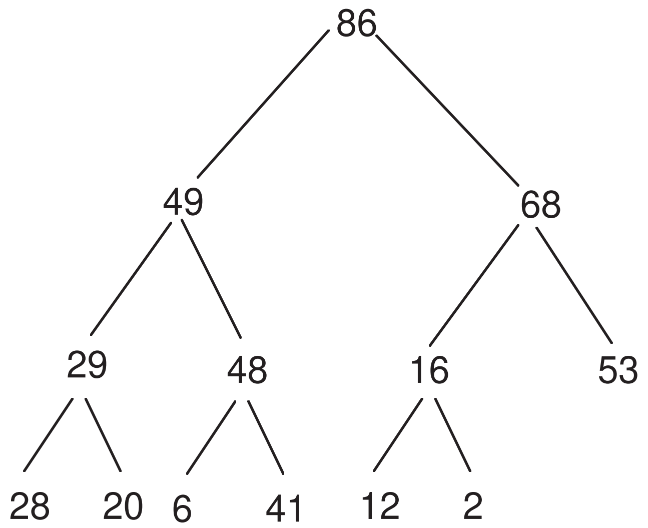
- The structure must support 2 operations efficiently: "delete max" and "insert".

ADT—**Priority Queue**: set of elements with operations "insert" and "deletemax".

Heap (An efficient implementation of Priority Queue):

- balanced binary tree
- all leaves at level h or $h-1$
- all leaves at level h to the left
- each element $\leq$ parent

Example: A Heap of size 13



Map this tree (heap) to an array

`A[1]: root`

`A[2i]: left child of  A[i]`

`A[2i+1] right child of A[i]`

$\Rightarrow$ map to A[1...13].

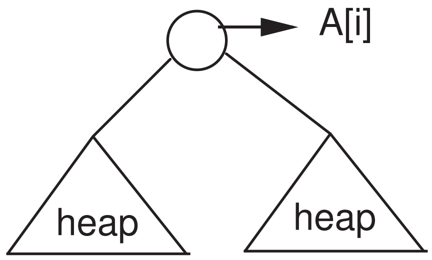
Creating a heap (top down):

Inserting one node into heap

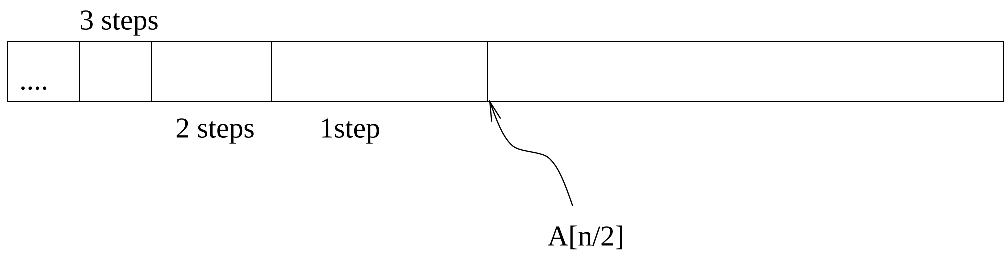
While $i > 1$ and $A[i] > A[i$ div 2]
do swap $A[i], A[i$ div 2]
$i :=i$ div 2 {filter up}
od;

To build heap, just repeatedly insert the n elements, keeping everything in the array. Time complexity for this is: $\sum_{i=1}^{n} \log i \approx n \log n$.

Creating a heap (bottom-up):



for $i = \lfloor \frac{n}{2} \rfloor$ step -1 to 1 do
"Restore" the small heap rooted at $A[i]$;



The time complexity to construct the heap (in place) is only:

$$\sum_{i=1}^{(\log n)-1} \frac{n}{2^{i+1}} i \approx n \sum \frac{1}{2^{i+1}} i = O(n)$$