Tirgul 4

- Order Statistics
  - minimum/maximum
  - Selection
- Heaps
  - Overview
  - Heapify
  - Build-Heap

Order statistics

- The i^{th} order statistics of a set of n elements is the i^{th} smallest element.
- For example the minimum is the first order statistics of the set and the maximum is the n^{th}.
- A *median* is the central element in the set.
- The *median* is a very important characteristic of a set and many times we will prefer using the median then using the average. (why?)

Minimum & Maximum

- How many comparisons are necessary to determine the minimum/maximum of a set of n elements?
- An upper bound of n-1 is easy to obtain, but can we do better?
- It is easy to show that the answer is no.
- How about finding both minimum and maximum, can we do better than 2*(n-1) ?
- yes

Selection in expected linear time

- What happens if we are not looking for the smallest or largest element, but for the i^{th} order statistics?
- One optional solution: sort ($\Theta(n \lg n)$) and index, can we do better?
- We can still get an expected asymptotic running time of $\Theta(n)$ using a modification of a randomized *quicksort.* (average case analysis)

Randomized Select

```
RandomizedSelect(A,p,r,i)
  if p==r
    then return A[p]
  q ← RandomizedPartition(A,p,r)
  if i < q then return RandomizedSelect(A,p,q-1,i)
  else if i > q then
     return RandomizedSelect( A, q+1, r, i −q )
  else return A[ q ]
```

Randomized Select

- We use the same **RandomizedPartition** like in the randomized quicksort.
- This time, instead of recursively sorting both sides of the pivot, we only deal with one.
- Are we guaranteed to do better than sort+select?
- No, like quicksort, we have a worst case of $O(n^2)$ (why?)

- But let's look at the average case:

Randomized Select

- We are using the same technique used to analyze the randomized *quicksort*.

$$T(n) \le \frac{1}{n}\left(\sum_{k=1}^{n-1}\left(\max(T(k), T(n-k))\right)\right) + dn$$

$$\le \frac{2}{n}\sum_{k=n/2}^{n-1}\left(T(k)\right) + dn$$

- Assuming T(k) ≤ *ck we get:*

$$\le \frac{2}{n}\sum_{k=n/2}^{n-1} ck + dn = \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2} k\right) + dn$$

$$= \frac{2c}{n}\left(\frac{n(n-1)}{2} - \frac{n/2(n/2-1)}{2}\right) + dn$$

$$\le c(n-1) - \frac{c}{2}\left(\frac{n}{2} - 1\right) + dn = c\left(\frac{3}{4}n - \frac{1}{2}\right) + dn$$

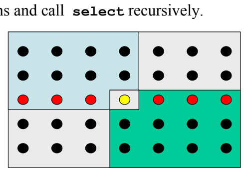
- We can pick c large enough such that: $3/4cn - 1/2c + dn \le cn$

Order Statistics

- So we can find the i^{th} order statistics either in $\Theta(n \lg n)$ time, or in an average $\Theta(n)$ time, but with a worst case of $O(n^2)$.
- Can we do better?
- Yes we can, a modified version of quick-select has a linear worst case time (but with a larger constant).
- We won't get into details (see Cormen, 10.3 – selection in worst-case linear time).

Select in worst case linear time

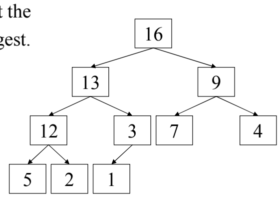
- **`select`** algorithm idea:
  1. Devide the input into n/c groups of c elements (for example, $c = 5$)
  2. Find the median of each group.
  3. Find the median of these medians.
  4. Partition the input around the median of medians and call **`select`** recursively.
- Proof idea:



  - Asymptotically, at least ¼ of the elements are larger than the pivot and at least ¼ are smaller than the pivot.
  - In the worst case, the number of elements in the recursive call is $3n/4$.
  - You've seen in class that quicksort achieves $n \lg n$ time even when the recurrence is called for $9n/10$ of the elements.

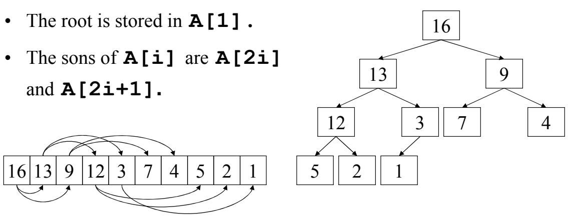
Heaps

- A heap is a complete binary tree, in which each node is larger than both its sons.
- The largest element of **each** sub tree is in the root of the sub tree.
- Note: this does ***not*** mean that the root's 2 sons are the next largest.



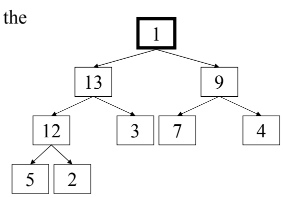
Heaps

- A heap can be represented by an array.
- Levels are stored one after the other.
- The root is stored in `A[1]`.
- The sons of `A[i]` are `A[2i]` and `A[2i+1]`.



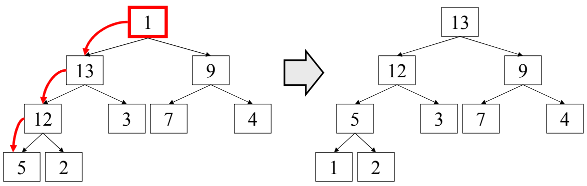
Heapify

- Assumes that both subtrees of the root are heaps, but the root may be smaller than one of its children.
- The idea is to let the value at the root to "float down" to the right position.
- What can we say about complexity?
- Worst case complexity of lg n (the tree is complete).



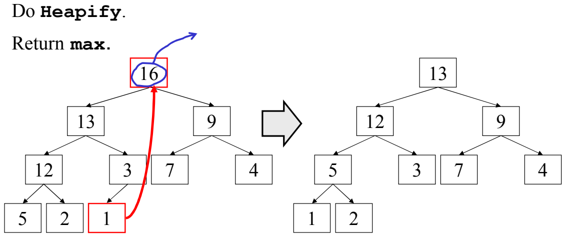
Heapify

```
Heapify(Node x)
    largest = max {left(x), right(x)}
    if ( largest > x )
        exchange (largest, x)
        heapify (x)
```



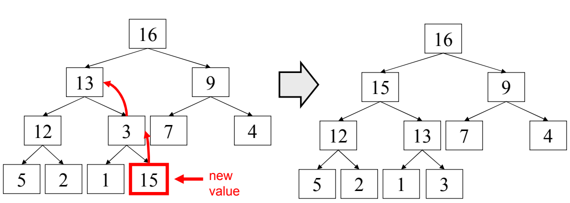
Heap-Extract-Max

- Save the root as **max**.
- Remove the last node and place it in the root.
- Do **Heapify**.
- Return **max**.



Heap-Insert

- Insert new value at the end of the heap.
- Let it "float up" to the right position.
- We still have an $O(\lg n)$ complexity.



Priority Queue

- Each inserted element has a priority.
- Extraction order is according to priority.
- Supported operation are Insert, Maximum, Extract-Max.
- Easily implemented with heaps.

Priority Queue

- Priority Queues using heaps:
  - Maximum operation takes $O(1)$
  - Extract-Max operation takes $O(\log n)$
  - Insert operation takes $O(\log n)$
- Priority Queues using sorted list
  - Maximum operation takes $O(1)$
  - Extract-Max operation takes $O(1)$
  - Insert operation takes $O(n)$

Build-Heap

```
Build-Heap(A)
      for i = ⌊length[A]/2⌋ downto 1
            do Heapify[A,i]
```

3	5	7	2	**1**	9	4	12	16	13

3	5	7	**2**	13	9	4	12	16	1

3	5	**7**	16	13	9	4	12	2	1

3	**5**	9	16	13	7	4	12	2	1

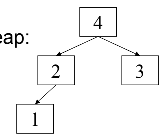
3	16	9	12	13	7	4	5	2	1

16	13	9	12	3	7	4	5	2	1

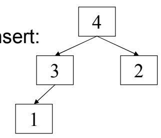
Build-Heap vs. Heap-Insert

- We want to create a new heap, containing n items, what should we do? Build a heap or insert the n items one by one?
- Build-Heap runs in $O(n)$ (why?).
- Inserting n items takes $O(n\log n)$.
- Sometimes Build-Heap and Heap-Insert create different heaps from the same input.
  - For example: the input sequence 1, 2, 3, 4

Build-Heap:

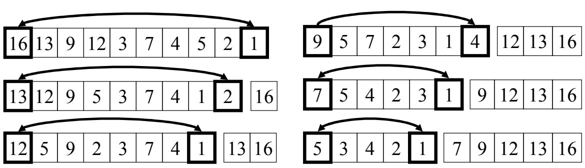


Heap-Insert:



Heapsort

```
Heapsort(A)
    Build-Heap(A)
    for i=length[A] downto 2 do
        exchange A[1] with A[i]
        heap-size[A]=heap-size[A]-1
        Heapify(A, 1)
```



Questions

- How to implement a stack/queue using a priority queue?
- How to implement an Increase-Key operation which increases the value of some node?
- How to delete a given node from the heap in $O(\log n)$?
- How to search for a key in a heap?