Analysis of Median Selection, Binary Heaps, Sorting

(Class 11)

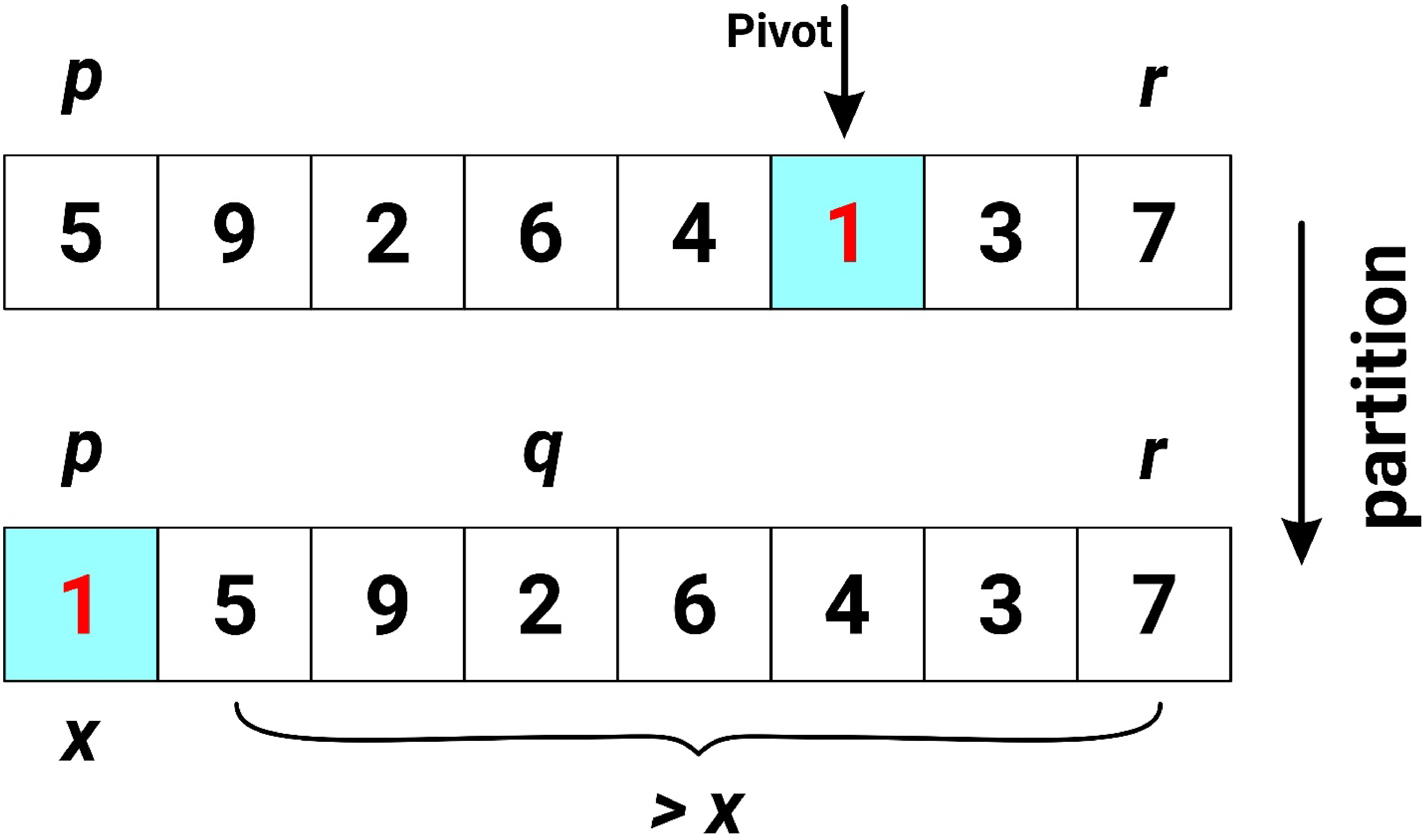
From Book’s Page No 157 (Chapter 6)

Select Algorithm

```
SELECT (array A, int p, int r, int k)
if (p = r)
   then return A[p]
else x ← CHOOSE PIVOT(A, p, r)
   q ← PARTITION(A, p, r, x)
   rank_x ← (q-p+1)
   if k = rank_x
      then return x
   if k < rank_x
      then return SELECT(A, p, q-1, k)
   else return SELECT(A, q+1, r, k-q)
```

Analysis of Selection

- We will discuss how to choose a pivot and the partitioning later.
- For the moment, we will assume that they both take $O(n)$ time.
- How many elements do we eliminate in each time?
- If x is the largest or the smallest, then we may only succeed in eliminating one element.


Pivot
p
r
5 9 2 6 4 1 3 7
partition
p
q
r
1 5 9 2 6 4 3 7
x
> x

- Ideally, x should have a rank that is neither too large nor too small.
- Suppose we are able to choose a pivot that causes exactly half of the array to be eliminated in each phase.
- This means that we recurse on the remaining $\frac{n}{2}$ elements.

- This leads to the following recurrence:

$$T(n) = \begin{cases} 1\,, & if\ n = 1 \\ T\left(\frac{n}{2}\right) + n, & if\ n > 1 \end{cases}$$

- The n term is the time consumed in partitioning and pivot selection.

- If we expand this recurrence, we get:

$$T(n) = n + \frac{n}{2} + \frac{n}{4} + \cdots$$

$$= \sum_{i=0}^{\infty} \frac{n}{2^i}$$

$$= n \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i$$

- Recall the formula for infinite geometric series; for any $|c| < 1$,

$$\sum_{i=0}^{\infty} c^i = \frac{1}{1 - c}$$

- So, solve the equation using this geometric series formula by putting $c = \frac{1}{2}$.

$$T(n) = n \frac{1}{1 - \frac{1}{2}}$$

$$T(n) = 2n$$

$$\boldsymbol{T(n) = O(n)}$$

- This sieve technique produced the $O(n)$ running time.
- Let's think about how we ended up with a $O(n)$ algorithm for selection.
- Normally, a $O(n)$ time algorithm would make a single or perhaps a constant number of passes of the data set.

- In this algorithm. we make a number of passes. In fact, it could be as many as $log\ n$.
- However, because we eliminate a constant fraction of the array with each phase, we get the convergent geometric series in the analysis.
- This shows that the total running time is indeed linear in n.
- This technique is well worth remembering.
- It is often possible to achieve linear running times in ways that you would not expect.

Sorting **(Book's Page No 157 (Chapter 6)**

- For the next few lectures, we will focus on sorting.
- There are a number of reasons for sorting.
- Here are a few important ones:
  - Procedures for sorting are parts of many large software systems.
  - Design of efficient sorting algorithms is necessary to achieve overall efficiency of these systems.
  - Sorting is well studied problem from the analysis point of view. Sorting is one of the few problems where provable lower bounds exist on how fast we can sort.
  - It means that we can compare two algorithms mathematically as well as describe the best-case running times as well.

- Particularly business applications have mainly based on the sorting and searching.
- We will analyze many sorting algorithms.
- Because sorting algorithms are evolved and improved a lot over the time and their analysis will teach us many useful techniques.

- In sorting, we are given an array $A[1 \dots n]$ of n numbers.
- We are to reorder these elements into increasing (or decreasing) order.
- More generally, A is an array of objects, and we sort them based on one of the attributes - the key value.

- The key value need not be a number. It can be any object from a totally ordered domain.
- Totally ordered domain means that for any two elements of the domain, x and y, either $x < y$, $x = y$ or $x > y$.
- Key can be a number, strings, etc.

Slow Sorting Algorithms $\boldsymbol{O(n^2)}$

- There are a number of well-known slow $O(n^2)$ sorting algorithms. These include the following:
  - Bubble sort
  - Insertion sort
  - Selection sort
- These algorithms are easy to implement.
- But they run in $O(n^2)$ time in the worst case.

Bubble Sort

- Scan the array.
- Whenever two consecutive items are found that are out of order, swap them.
- Repeat until all consecutive items are in order.

Insertion Sort

- Assume that $A[1 \ldots i-1]$ have already been sorted.
- Insert $A[i]$ into its proper position in this sub array.
- Create this position by shifting all larger elements to the right.

Selection Sort

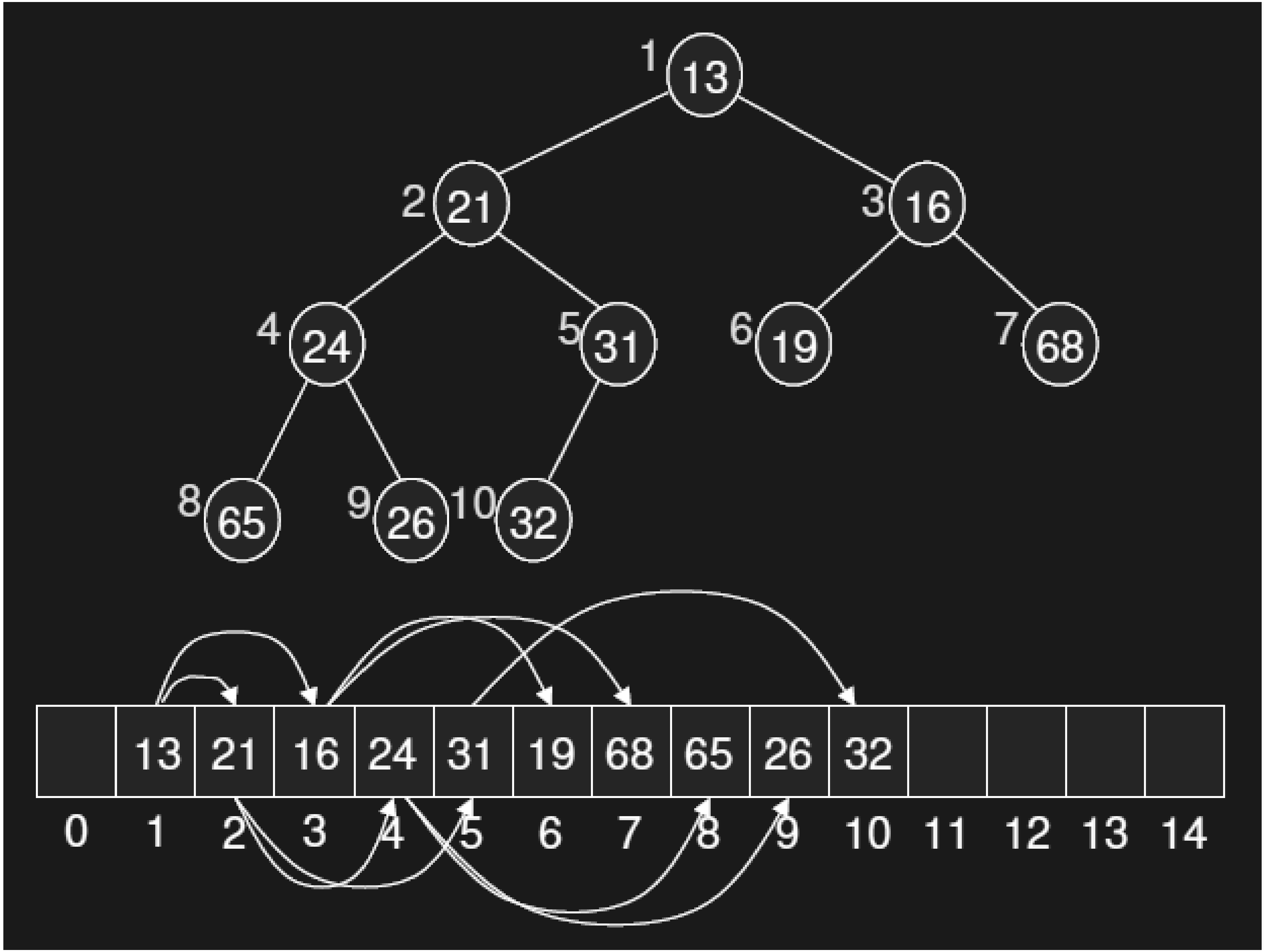
- Assume that $A[1 \dots i-1]$ contain the $i-1$ smallest elements in sorted order.
- Find the smallest element in $A[i \dots n]$.
- Swap it with $A[i]$.

Sorting in $\boldsymbol{O(n\ log\ n)}$ Time

- We have already seen that merge sort sorts an array of numbers in $O(n \log n)$ time.
- We will study two others:
  - Heapsort
  - Quicksort

Heaps (Book's Page No 161 (Chapter 6)

- A *heap* is a left-complete binary tree that conforms to the *heap order*.
- The heap order property is:
  - In a (min) heap, for every node X, the key in the parent is smaller than or equal to the key in X.
  - In other words, the parent node has key smaller than or equal to both of its child nodes.
  - Similarly, in a max heap, the parent has a key larger than or equal both of its child nodes .
  - Thus, in a min heap the smallest key is in the root.
  - While in the max heap, the largest is in the root.



1 13
2 21
3 16
4 24
5 31
6 19
7 68
8 65
9 26
10 32
13 21 16 24 31 19 68 65 26 32
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

- The number of nodes in a complete binary tree of height h is:

$$n = 2^0 + 2^1 + 2^2 + \cdots + 2^h$$

$$= \sum_{i=0}^{h} 2^i$$

$$n = 2^{h+1} - 1$$

- h in terms of n is:

$$h = (\log(n+1)) - 1$$
$$\approx \log n$$

- So, the total number of levels of the binary tree are h.

- One of the clever aspects of heaps is that they can be stored in arrays without using any pointers.
- This is due to the left-complete nature of the binary tree.
- We store the tree nodes in level-order traversal.

- Access to nodes involves simple arithmetic operations.
- For any node i, its parent and child nodes are as follows:
  - $left(i)$: returns $2i$, index of left child of node i.
  - $right(i)$: returns $2i + 1$, the right child.
  - $parent(i)$: returns $\left\lfloor \frac{i}{2} \right\rfloor$, the parent of i.
- The root is at position 1 of the array.