Heap

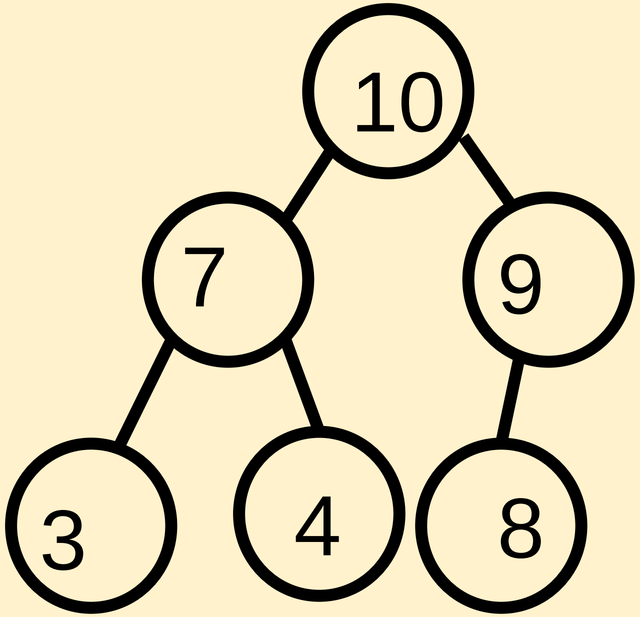
- Heap property
- Inserting an element into a heap
- Creating a heap
- Faster method to build a heap
- Heapsort

Heap property

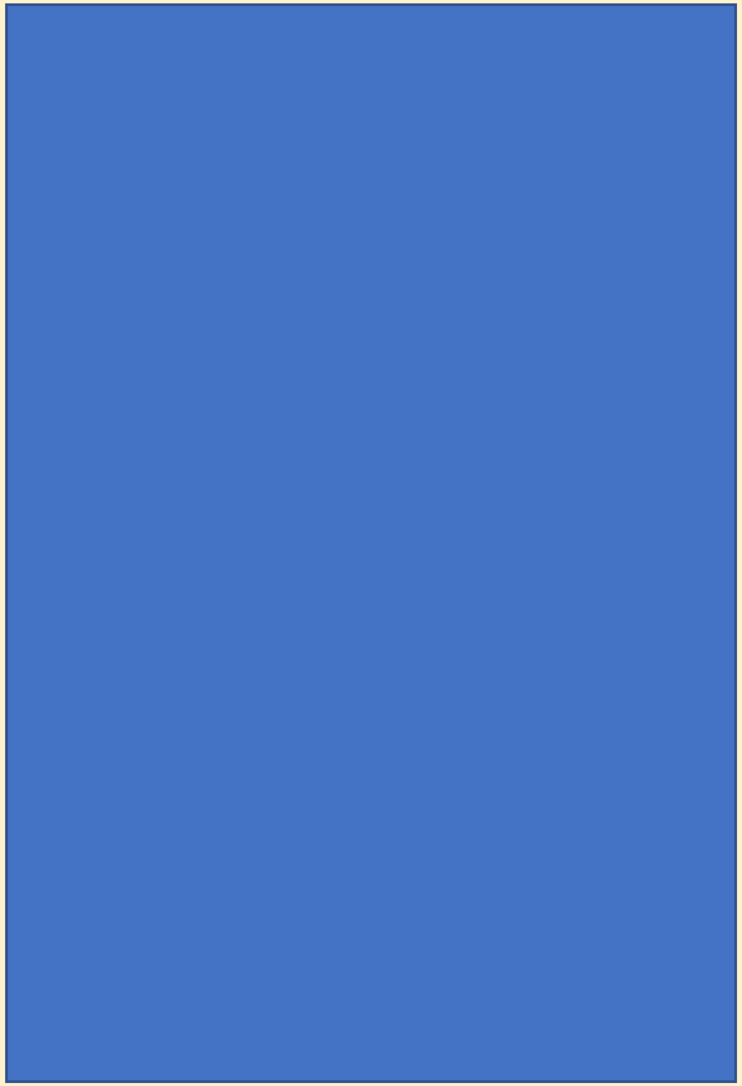
A heap is a binary tree with two properties:

1. Relational or Heap-Order Property: the value (or key) stored in every node is at least as large, a max-Heap (or small, a min-Heap) as the value of its children (if they exist)

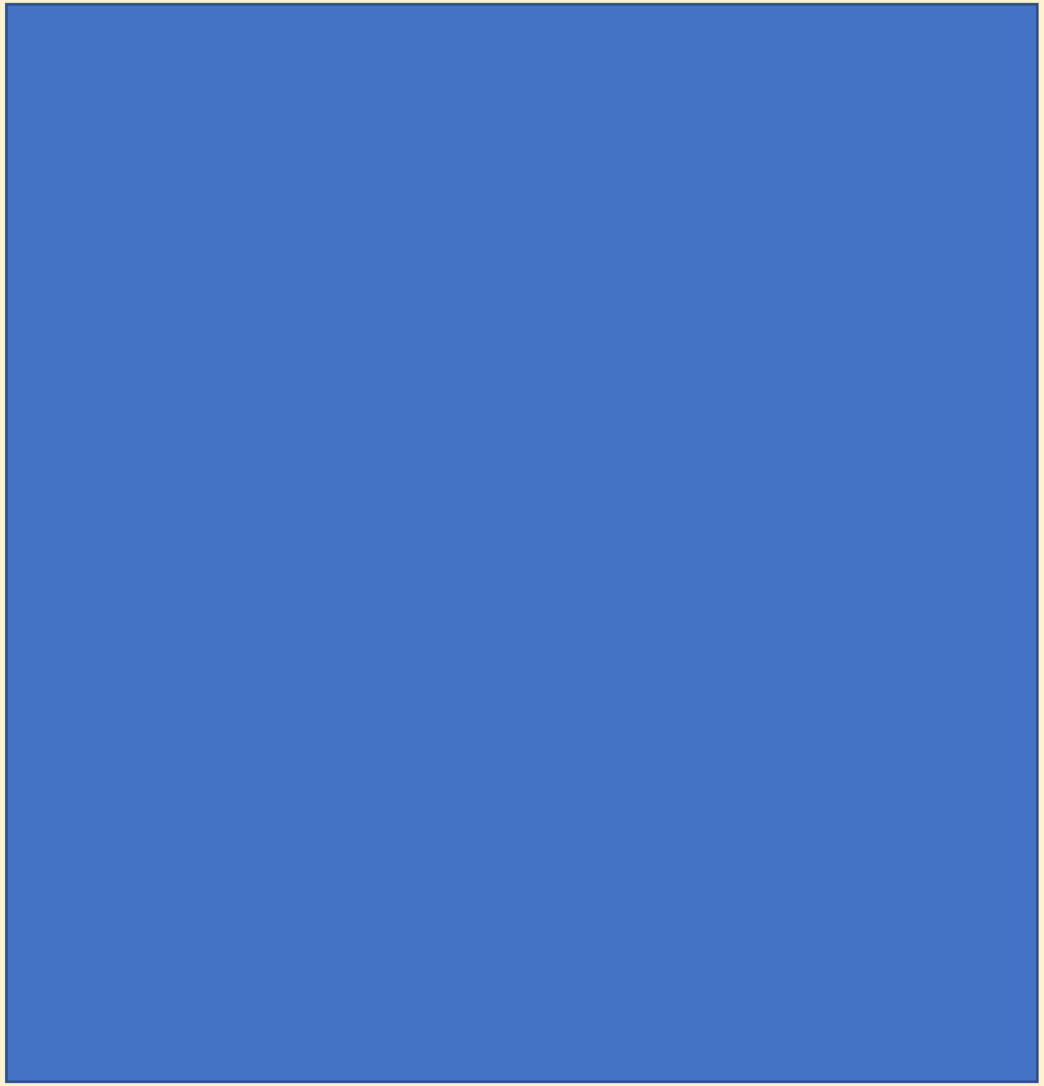
2. Structural Property: the binary tree is Complete i.e. all leaves are on adjacent levels and the nodes can be compactly stored in a 1-D array (can be labelled consecutively)



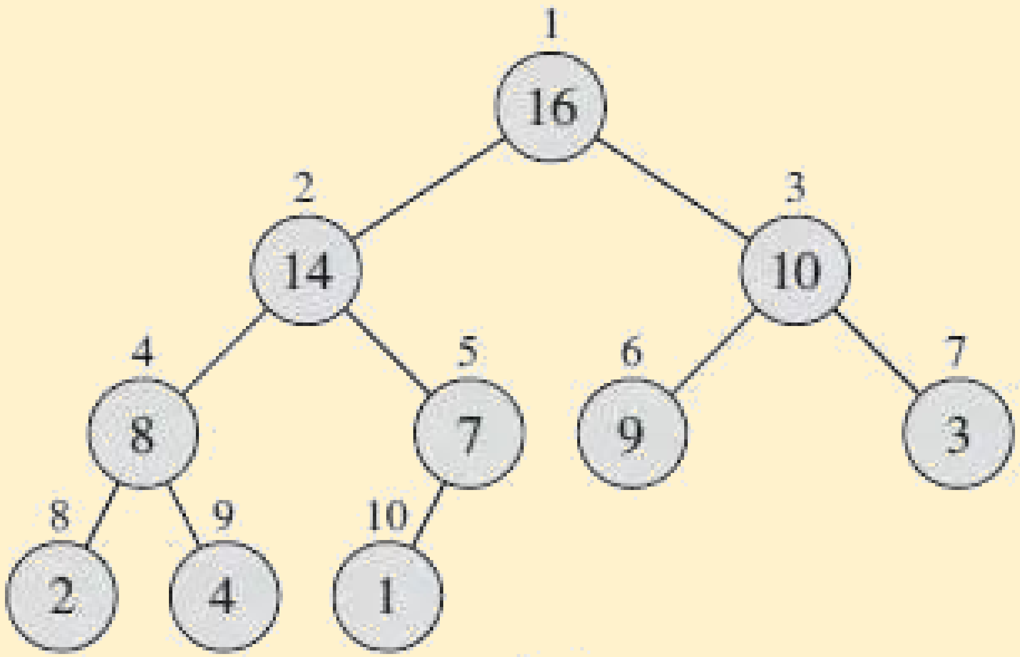
Heap



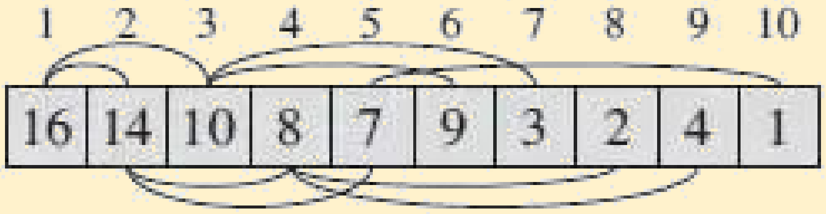
Not Heap



Not complete



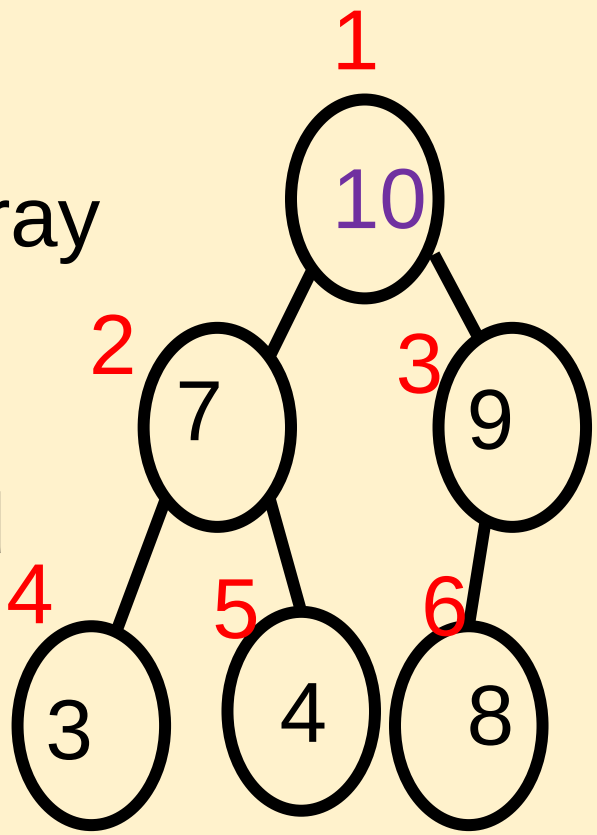
(a)



(b)

- Observe that largest element is always the root in a (max) heap.

- Since a heap is a complete binary tree, its elements can be conveniently stored in an array [10, 7, 9, 3, 4, 8].



- If an element is at position n, then its left child will be in position $2n$, its right child will be in $2n+1$.

- A non-root element at position n will have parent $\frac{n}{2}$

Define Heap (exam type question)

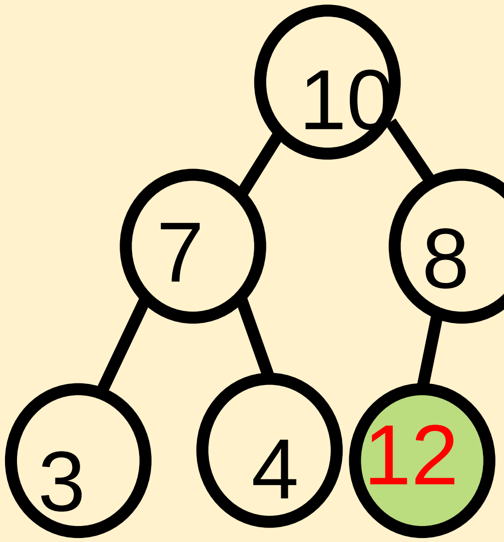
Heap is binary tree with two properties:

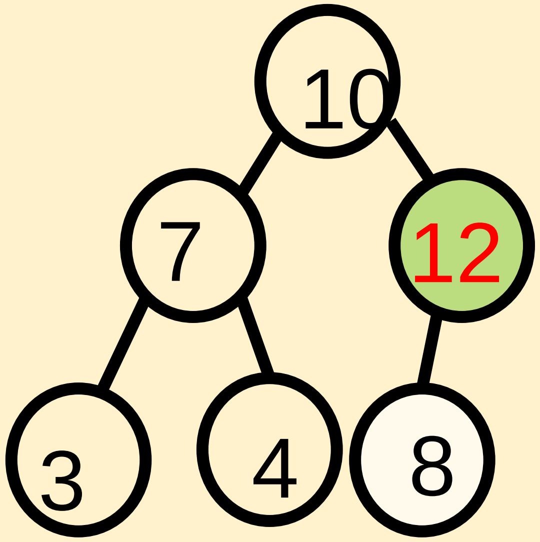
1. Relational or Heap-Order Property:

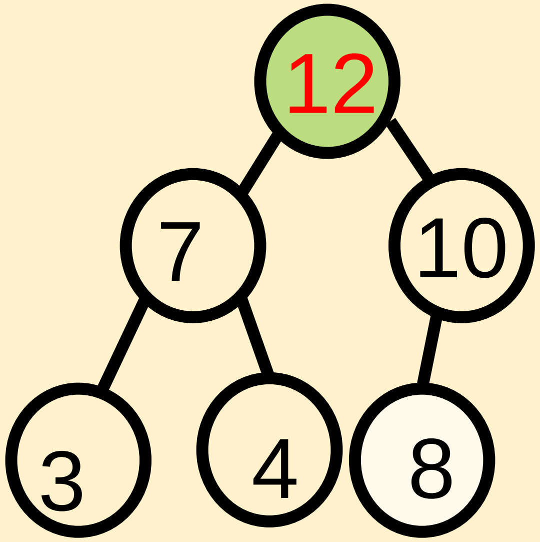
2. Structural Property

Inserting an element into a heap

- To insert an element into the heap, add it at “the bottom”
- Then compares it with its parent, grandparent, great grandparent, etc
- until it is less than or equal to one of them.



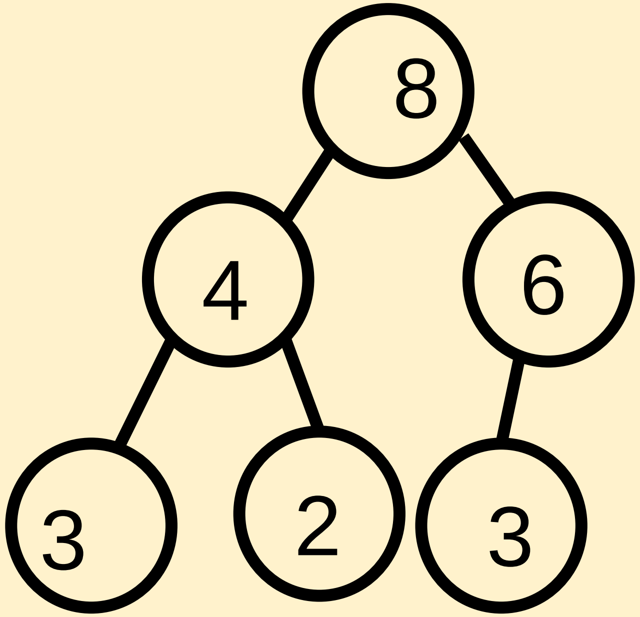




```
Insert(A: Heap; n:integer)
{
    j=n; i=n/2; data=A[n];
    while((i >0)&A[i]< data) {
        A[j] =A[i];
        j=i;
        i=i/2;}
    A[j] =data;
}
```

- Insert an element takes $O(\log n)$ in worst case
- proportional to number of levels

In class exercise : Insert an element 9 into the following heap



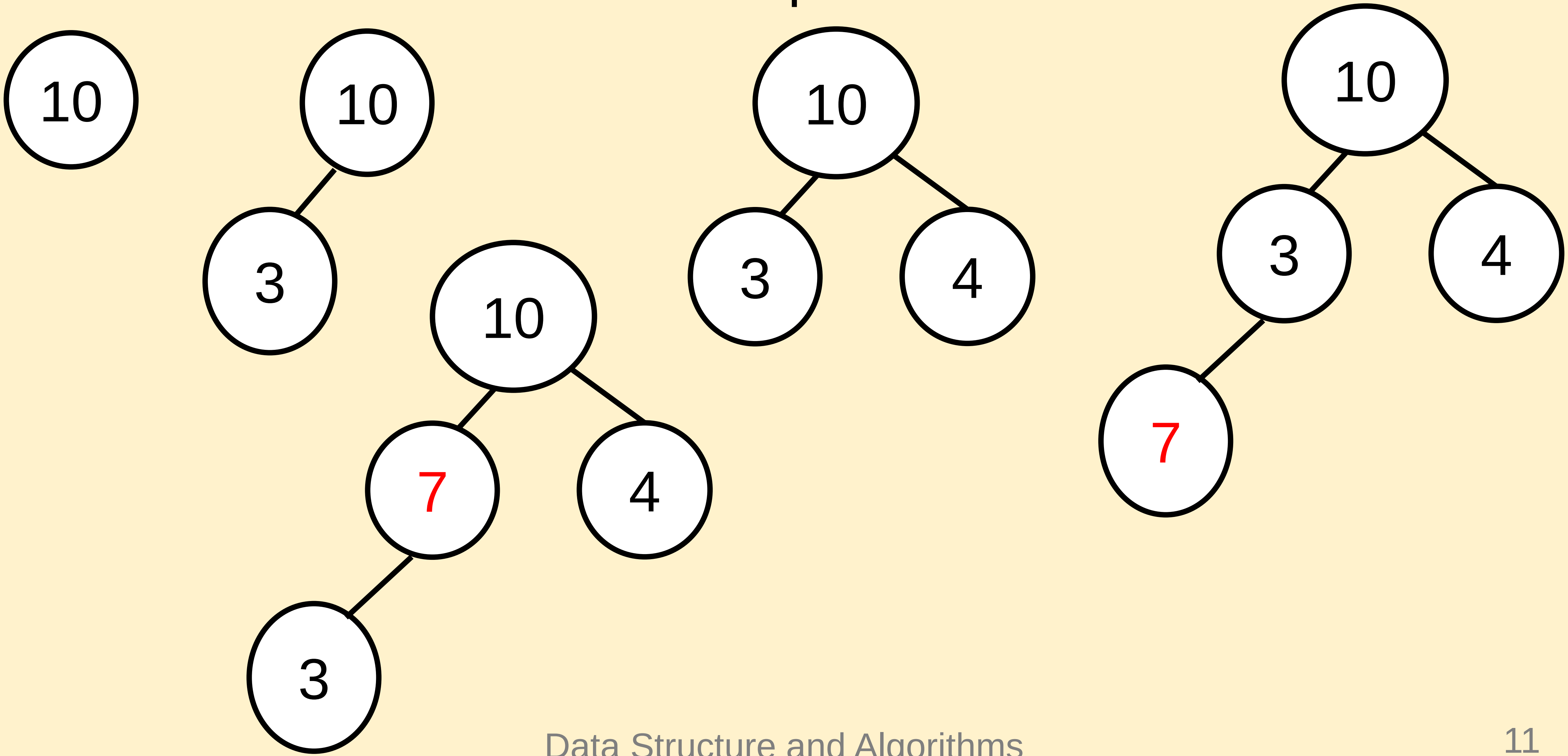
Creating a heap from *n* arbitrary elements

- We start with one element *A[1]*
- Insert each element into the heap by applying the Insert method *(n-1)* times.
- Therefore it has *O(nlogn)* complexity in the worst case
- On average only *O(n)* because elements only move a limited distance.

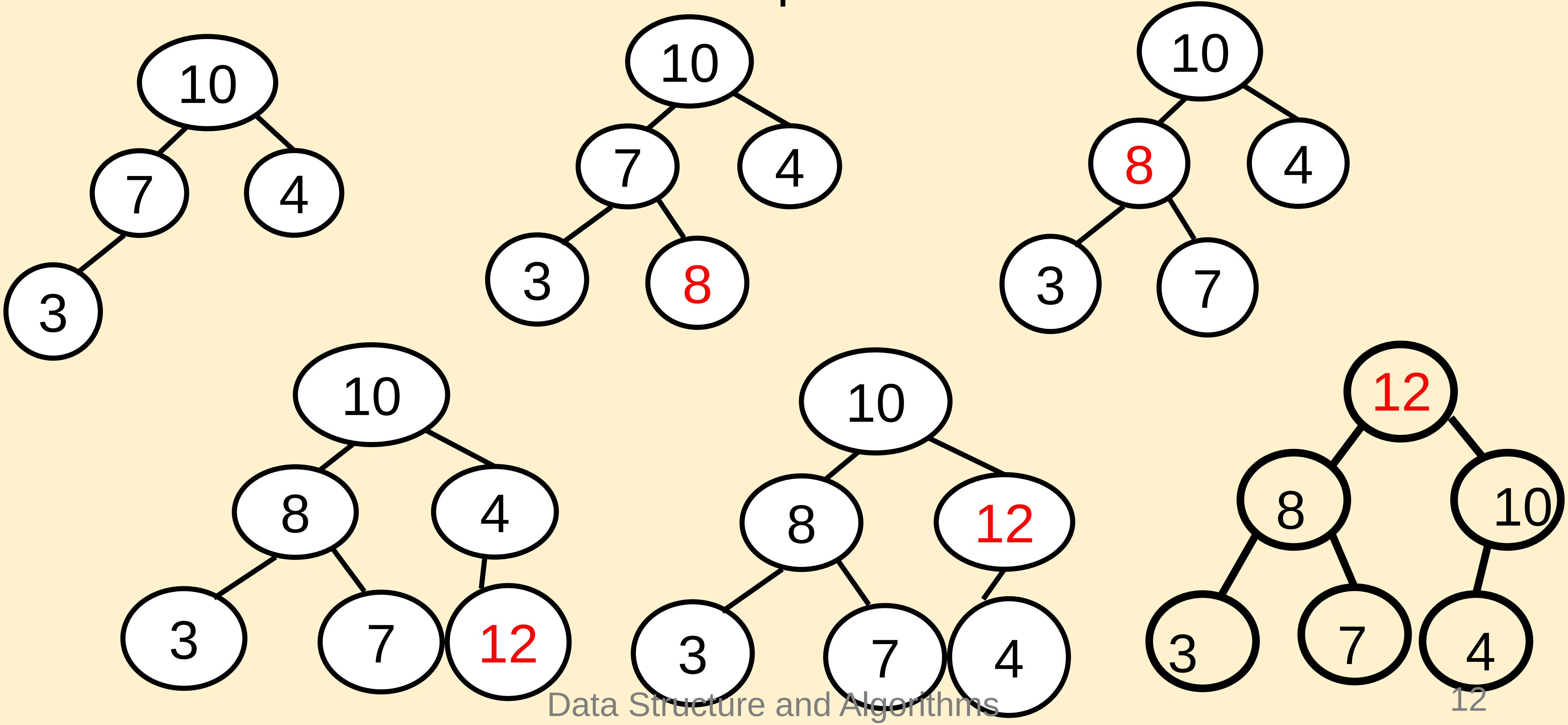
```
Heapify(A: Heap)
    for (i= 2; i < n; i+ +) {
      Insert(A,  i);
    }


Insert(A: Heap; n:integer)
{
      j=n; i=n/2; data=A[n];
      while((i >0)&A[i]< data) {
            A[j] =A[i];
             j=i;
             i=i/2;}
     A[j] =data;
  }
```

In class exercise : Making a heap from {10,3,4,7,8,12}.
Draw out the trees for each step



In class exercise : Making a heap from {10,3,4,7,8,12}.
Draw out the trees for each step

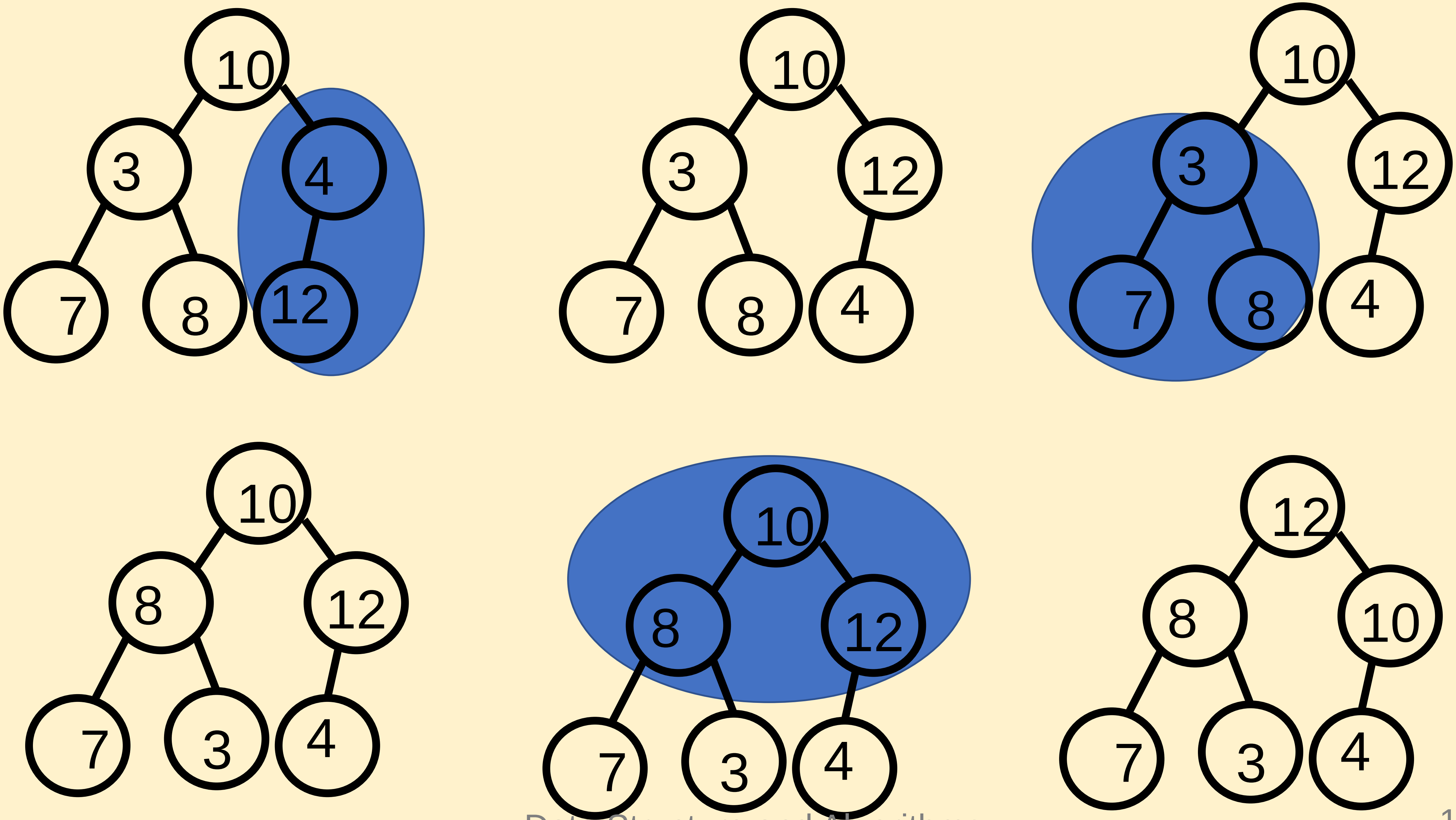


Faster method to build a heap

- At each level, the left and right subtrees of any node are heaps.
- Only the value in the root node may violate the heap property.
- Hence it is possible to work bottom-up recursively to fix the heap.

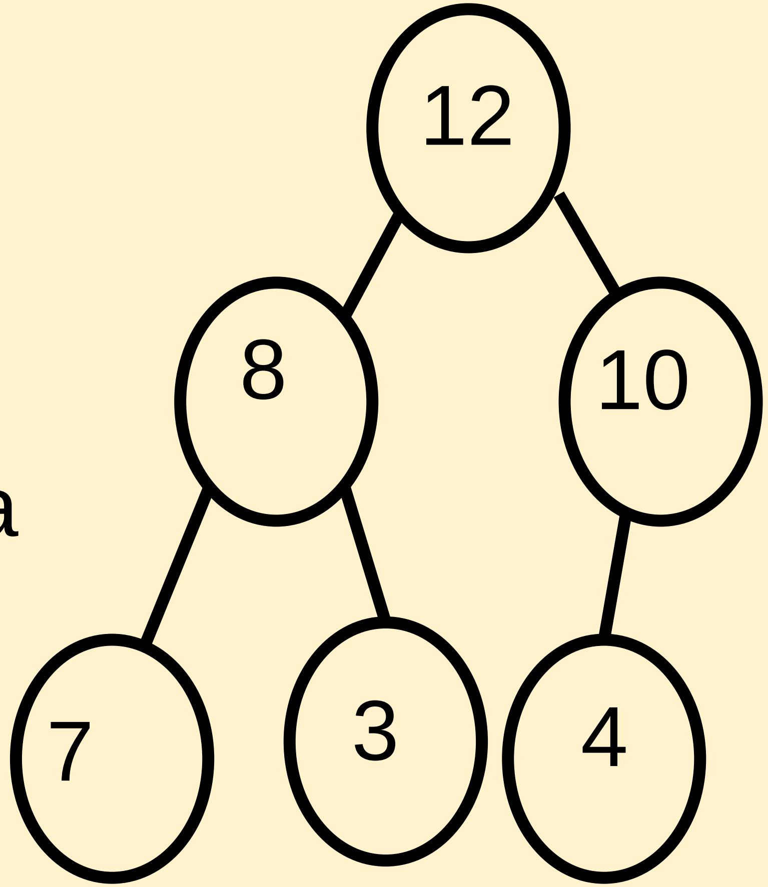
```
MakeHeap(A:Heap; n:integer)
    for( i = n/2; i >= 1; i -- ){
    FixHeap(A,n,i);
  };


FixHeap(A:Heap; n; i:integer){
    j= 2*i; data=A[i];
    while(j <=n){
        if((j < n)& A[j]< A[j+ 1]) j=j+ 1;
        if data >=A[j] break;
        else A[j/2] =A[j]; j=j*2;
  }
    A[j/2] =data;
}
```


10
3
4
7
8
12
10
3
12
7
8
4
10
3
12
7
8
4
10
8
12
7
3
4
10
8
12
7
3
4
12
8
10
7
3
4

Heapsort

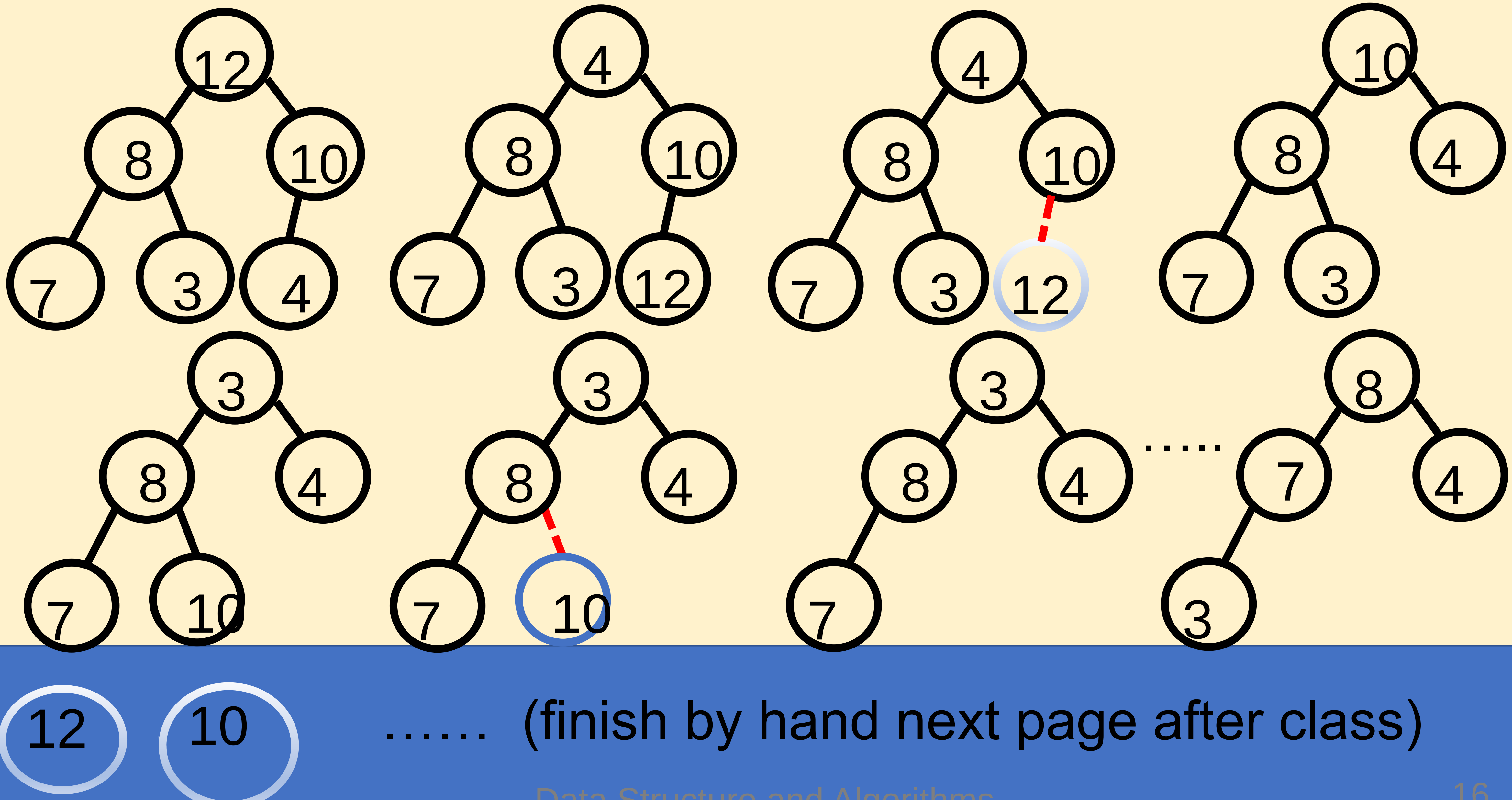
Example: Sorting 12,8,10,7,3,4



- Since in a heap maximum is always at root, we can use this property to devise a sorting method:

1. Making a heap from array $A[1,\ldots,n]$.
2. Swapping the first and last element.
3. Remaking the heap $A[1\ldots n]$.
4. Repeating the last two steps until the heap has only one element.

Make heap, swap, reduce list, fix heap, ….



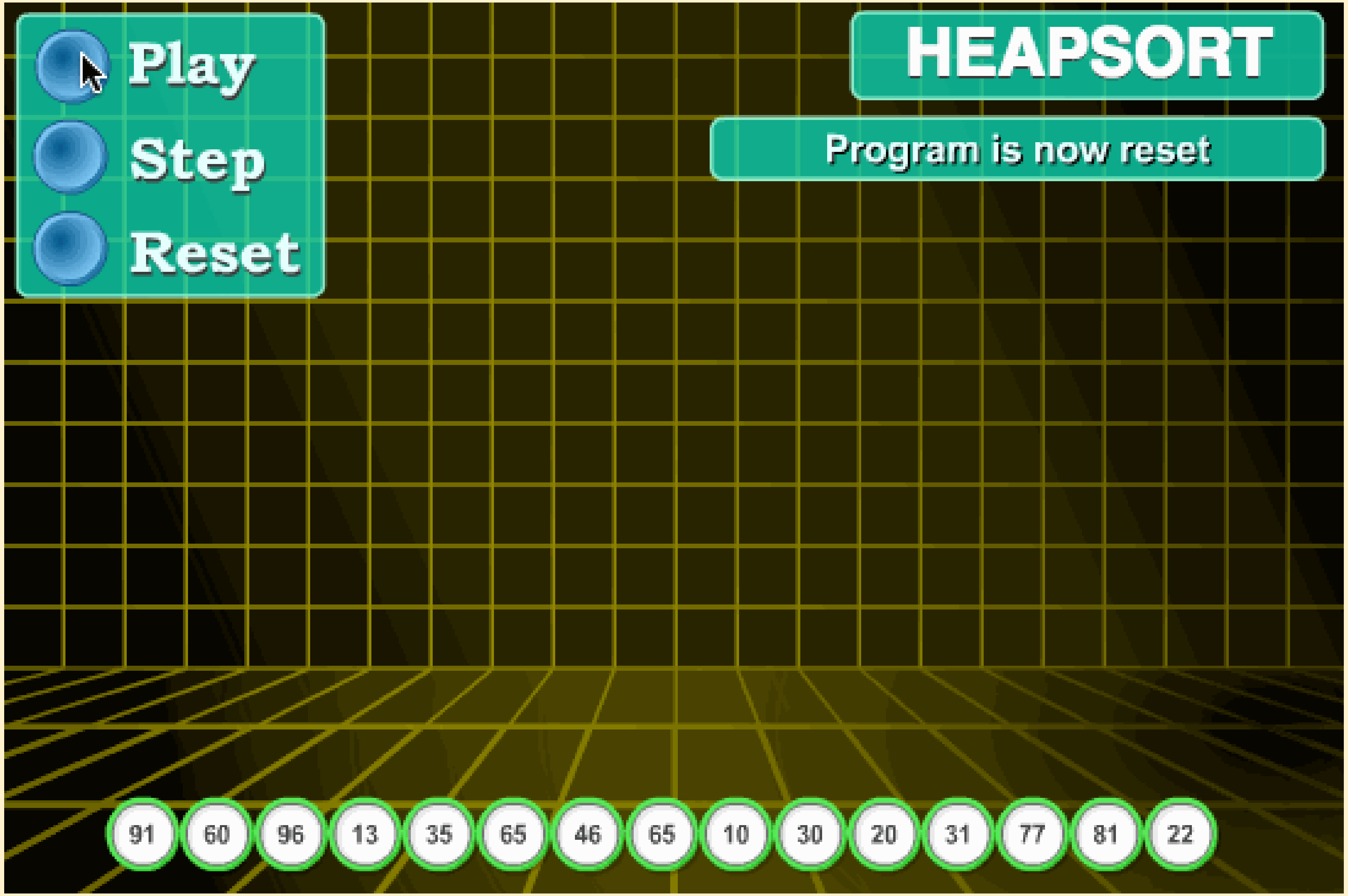
……. (finish by hand next page after class)

```
HeapSort(A:Arraydata; n:integer)
    MakeHeap(A, n);
    for(i=n;i >1; i−−) {
        temp=A[i];
        A[i] =A[1];
        A[1] =temp;
        FixHeap(A,i-1,1);
    }

MakeHeap(A:Heap; n:integer)
     for( i = n/2; i >= 1; i −− ){
        FixHeap(A,n,i);
    };
```

```
FixHeap(A:Heap; n; i:integer){
    j= 2*i; data=A[i];
    while(j <=n){
        if((j < n)& A[j]< A[j+ 1])
        j=j+ 1;
        if data >=A[j] break;
        else A[j/2] =A[j]; j=j*2;
        }
    A[j/2] =data;
    }
```

- A FixHeap costs $O(\log n)$ and for-loop is $O(n)$, Heapsort is $O(n\log n)$.


Play
Step
Reset
HEAPSORT
Program is now reset
91 60 96 13 35 65 46 65 10 30 20 31 77 81 22

Priority queue

- It is a queue, but not first in first out, each element has a key associated with priority.
- Used for scheduling processes in computer, etc.
- In a priority queue, you can add successive pieces of data
- retrieve the one that has the “highest priority” in constant time.
- comparisons can be made between its elements to determine which one has the “highest priority”.
- Heap is a usual way of implementing “Priority queue”.

Summary of heap

- Insertion into heap $O(\log n)$

- Construct a heap
    a) Using insertion $O(n\log n)$
    b) Bottom up $O(n)$

- Heapsort

- Applications of heap
    ---- Implementation of Priority Queues.

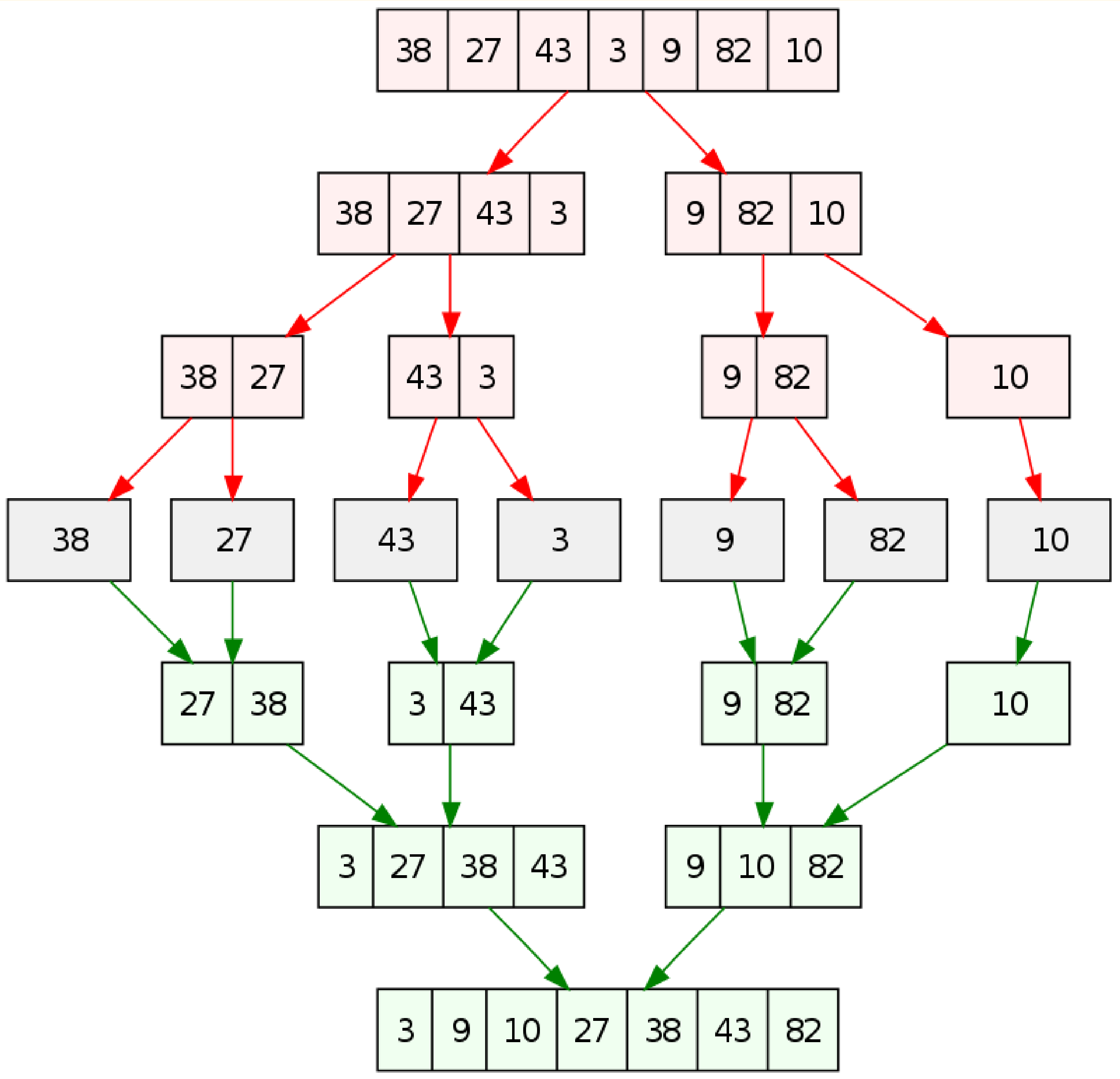
Extension: reverse all examples for minheap

Divide and conquer (D&C) algorithms

- General method
- max-min algorithm
- Selection algorithm

General method

- Given a function to compute on n inputs, the divide and conquer strategy suggests splitting the input in to k distinct subsets, yielding k subproblems.
- These sub-problems must be solved, then a method must be found to combine sub-solutions into a solutions of the whole.
- Often the subproblems are of the same type as the original problem. If the sub-problems are still large, apply D&C to the sub-problem (use recursion).
- Smaller and smaller subproblems are produced until the problem is small enough, which can be solved without splitting.


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Control abstraction for D& C

- By control abstraction, we mean a procedure whose flow of control is clear, but
- whose primary operations are specified by other procedures, of which the precise meaning are left undefined.

Algorithm D&C(P)

if Small(P) return Solve(P)

else{

Divide P into smaller instances $P_1, P_2, \ldots P_k$

Apply D&C to each of these subproblems

return Combine(D&C(P_1*), D&C(*P_2*), …,D&C(*P_k*))*

}

Algorithm D&C(P)

if Small(P) return Solve(P)

else{

Divide P into smaller instances P_1, P_2, … P_k

Apply D&C to each of these subproblems

return Combine(D&C(P_1), D&C(P_2), …,D&C(P_k))

}

- *Small(P)* is a Boolean valued function that determines whether the problem is small enough.
- If yes, the function *Solve(P)* is invoked.
- otherwise, each of subproblems is solved by *D&C* algorithm.
- *Combine* is a function that determines the solution of *P* using the solutions to *k* sub-problems.

Matrix multiplication

Strassen Matrix multiplication

Convex hull

Master theorem

Divide and Conquer

max-min

Selection

Multiplication of two integers

max-min Algorithm

- Find the minimum and maximum element from a given list of *n* elements.
- Without D&C:

```
Algorithm MaxMin(A: list, n, max,  min: integer)
{
    max=A[1];min=A[1];
     for(i= 2;i <=n;i++){
            if(A[i]> max) max=A[i];
            if(A[i]< min)  min=A[i];
     }
}
```

- *Time complexity:* $T(n) = 2(n-1)$

D&C max-min:

1. Divide the list into small groups.
2. Then find max and min of each group.
3. The max/min of result must be one of maxs and mins of the groups.

e.g. $A = [5,7,1,4,10,6]$

$A_1 = [5,7,1]$, $max(A_1) = 7$, $min(A_1) = 1$

$A_2 = [4,10,6]$, $max(A_2) = 10$, $min(A_2) = 4$

So the min and max of A is $min(1,4)$ *and* $max(7,10)$,

i.e. 1 *and* 10

- i, j are parameters set as $1 \leq i \leq j \leq n$; the algorithm finds the maximum and minimum of $A[i{:}j]$
- Small(P) is true for $n \leq 2$:direct solve
  a) List has one element
  b) List has two element
- List has more than two elements: divide into two groups
- Solve the subproblems and combine the solutions

```
Algorithm D&CMaxMin(A:list; i; j; fmax;
fmin:integer)
 {
   if(i==j) {fmax=A[i];fmin=A[i]}
    else
        if(i==j −1)
           if(A[i]> A[j]) {fmin=A[j]; fmax=A[i] }
           else  {fmin=A[i]; fmax=A[j] }
       else
      { mid= (i+j)/2;
      D&CMaxMin(A:i;  mid, gmax, gmin: integer)
      D&CMaxMin(A:mid+ 1,  j; hmax, hmin:
    integer)
       fmax=max(gmax; hmax);
       fmin=min(gmin; hmin);
       }
}
```

Trace of recursive calls

[5, 7,1, 4, 10, 6]

[5, 7,1] [4, 10, 6]

Divide

[5] [7,1] [4] [10, 6]

5,5 1,7 4,4 6,10

1,7 4,10

1,10

Combine

In class exercise: Trace of recursive calls

[1, 2,3, 4, 5, 6, 7, 8]

- Time complexity

$$T(n) = \begin{cases} 2T(n/2) + 2 & \text{for } \; n > 2 \\ 1 & \text{for } \; n = 2 \\ 0 & \text{for } \; n = 1 \end{cases}$$

- Suppose $n=2^k$

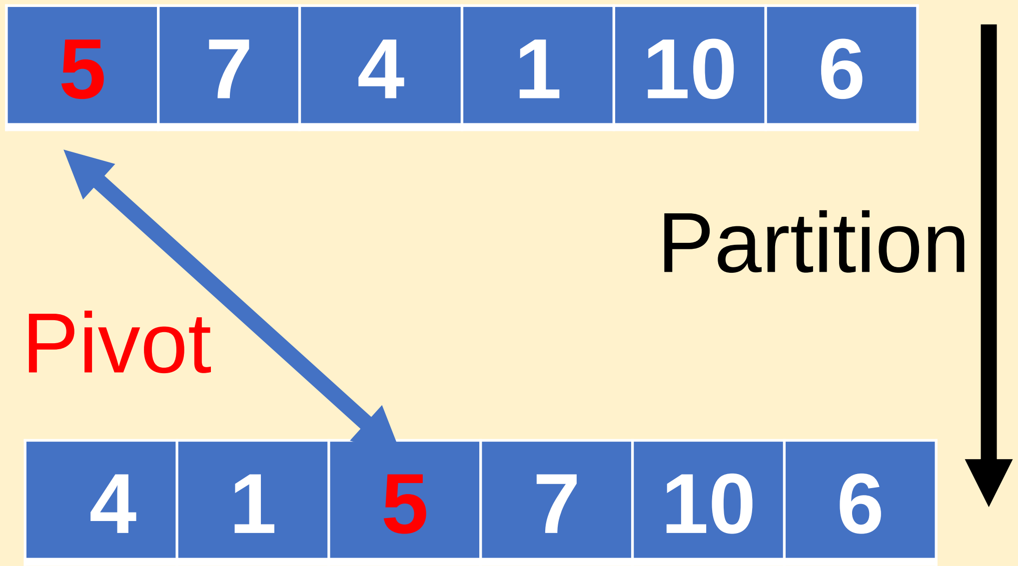
$$T(n) = 2T\left(\frac{n}{2}\right) + 2$$
$$= 2\left(2T\left(\frac{n}{4}\right) + 2\right) + 2 = 2^2T\left(\frac{n}{4}\right) + 4 + 2$$
$$= \dots\dots$$
$$= 2^{k-1}T(2) + (2^{k-1} + \dots . +4 + 2)$$
$$= 2^{k-1} + 2^k - 2$$
$$= 3\,(n/2) - 2$$

- D&C MaxMin: $3(n/2)-2$
- Non D&C MaxMin: $2n-2$
- Saving of 25%
- But needs extra storage (log(n)) for stack variables

Selection algorithm

- Given list of n elements, determine the k^{th} smallest (largest) element (This is an extension of max-min problem, since when $k= 1$, they are equivalent)
  1. Choose a value in the list.
  2. Partition the list so that the chosen value is in its final position if we are sorting. Call this position j. (i.e. all values to the left of j are smaller and all the elements to the right are larger).
     - Since j is in correct place if $j=k$, we have found the kth smallest (if $j=n-k+1$, the k^{th} largest).
     - If $j > k$ then the k^{th} smallest is in the left sub-list otherwise in the right sub-list.

Partition Algorithm




```
Algorithm Partition(A:list, first, last: integer)
{
    v=A[first]; i= first; j=last;
    do{
        do i=i+1  while(A[i]< v);
        do j=j-1  while(A[j]> v);
        if(i <= j) Swap(A[i]; A[j]);
     } while(i < j)
     A[first] =A[j]; A[j] =v;
     return j;
}
```

- Partition the list so that the chosen value is in its final position if we are sorting.
- All values to the left of *j* are smaller and All the elements to the right are larger
- Return *j*

Example

7 >5, $j = j - 1$

10 >5, $j = j - 1$

1 <5, $j = j - 1$

Swap 1 and 7

4 <5, $i = i + 1$

7 >5, $j = j - 1$

4 <5, $j = j - 1$

$A[first] = A[j]; A[j] = v$

$i=2$	**5**	**7**	**4**	**1**	**10**	**6**	$j=6$
$i=2$	**5**	**7**	**4**	**1**	**10**	**6**	$j=5$
$i=2$	**5**	**7**	**4**	**1**	**10**	**6**	$j=4$
$i=3$	**5**	**1**	**4**	**7**	**10**	**6**	$j=4$
$i=4$	**5**	**1**	**4**	**7**	**10**	**6**	$j=3$
$i=3$	**4**	**1**	**5**	**7**	**10**	**6**	$j=3$

In class exercise: Partition   [15,7,4,2,10,6]

Selection algorithm

D&C implicit in the loop

```
Algorithm Selection(A:list, n, k:integer)
{
    first= 1; last=n+1;
    while true {
        j=partition(A, first; last);
            if (k==j)  break;
            if (k < j)  last=j−1;
            else  first=j+ 1;
    }
}
```

```
Algorithm Partition(A:list, first, last: integer)
{
    v=A[first]; i= first; j=last;
    do{
        do i=i+1  while(A[i]< v);
        do j=j-1  while(A[j]> v);
        if(i < j) Swap(A[i]; A[j]);
    } while(i < j)
    A[first] =A[j]; A[j] =v;
    return j;
}
```

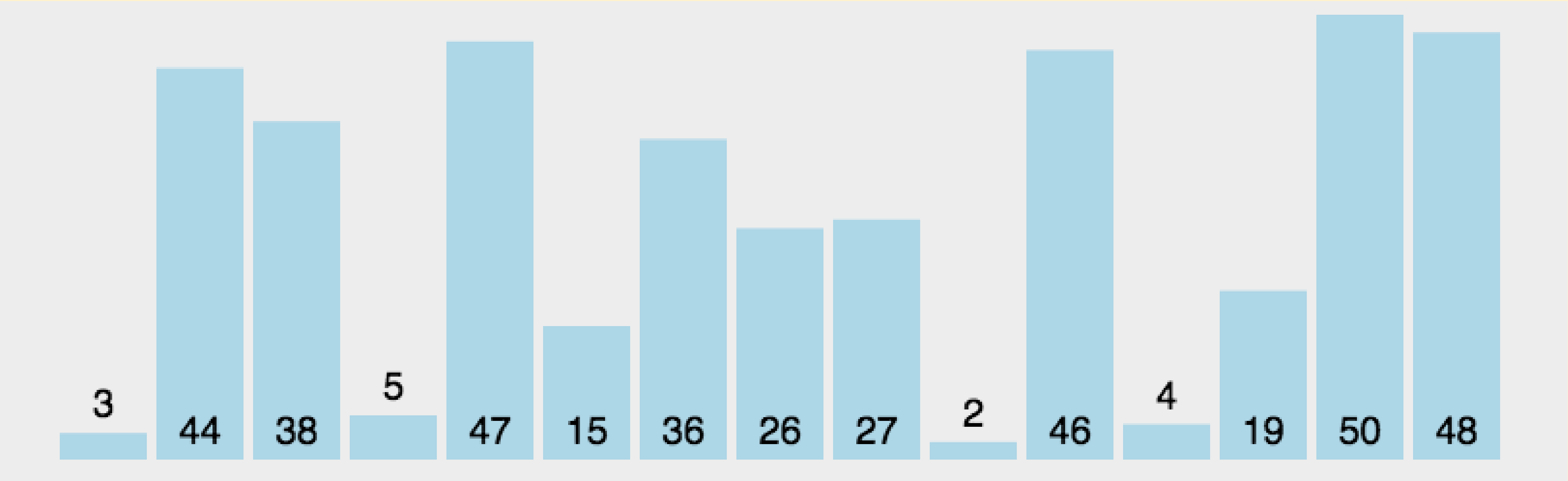
- **Selection Algorithm finds the kth smallest in A, *first and last* are index of the first and last index of a sub-list of A in Partition algorithm**
- **Return kth smallest**
- **Search left sublist**
- **Search right sublist**

In class exercise: Find the second smallest ($k=2$) of [5,7,4,1,10,6]

- [4,1,5,7,10,6] after partition
- 5 is the 3rd smallest and $j=3$.
- Since $j > k$, we consider the left sub-list [4 1 5]
- [1,4,5] after partition.
- 4 is the second smallest.
- $j = k = 2$ ***return***

In class exercise: Find the third smallest ($k=3$) of [15,7,4,2,10,6]

Selection algorithm



https://www.cnblogs.com/onepixel/p/7674659.html