CSE5311 Design and Analysis of Algorithms

- This Class
  - Heaps and Heapsort?
  - QuickSort
  - Mergesort
  - Other Sorting Algorithms

Further Reading Reference books on Algorithms

- At the end of the class
- Binary trees
- Priority queues and heaps
- Quicksort
  - Worstcase
  - Bestcase
- Mergesort
- Recurrences for Quicksort and Mergesort

Course Syllabus

- **Review of Asymptotic Analysis and Growth of Functions, Recurrences**
- **Sorting Algorithms**
- Graphs and Graph Algorithms.
- Greedy Algorithms:
  - **Minimum spanning tree,Union-Find algorithms, Kruskal's Algorithm,**
  - **Clustering,**
  - **Huffman Codes, and**
  - **Multiphase greedy algorithms.**
- Dynamic Programming:
  - **Shortest paths, negative cycles, matrix chain multiplications, sequence alignment, RNA secondary structure, application examples.**
- Network Flow:
  - **Maximum flow problem, Ford-Fulkerson algorithm, augmenting paths, Bipartite matching problem, disjoint paths and application problems**.
- NP and Computational tractability:
  - **Polynomial time reductions; The Satisfiability problem; NP-Complete problems; and Extending limits of tractability.**
- Approximation Algorithms, Local Search and Randomized Algorithms

SORTING ALGORITHMS

Further Reading
Reference books on Algorithms

Heaps and Heapsort

- ❑ **Priority Trees**
- ❑ **Building Heaps**
- ❑ **Maintaining heaps**
- ❑ **Heapsort Algorithm**
- ❑ **Analysis of Heapsort Algorithm**

Priority Queues

What is a priority queue?

A priority queue is an abstract data type which consists of a set of elements. Each element of the set has an associated priority or key Priority is the value of the element or value of some component of an element

Example :

S : {(Brown, 20), (Gray, 22), (Green, 21)} priority based on name
{(Brown, 20), (Green,21), (Gray, 22)} priority based on age

Each element could be a record and the priority could be based on one of the fields of the record

Example

A Student's record:

Attributes :	**Name**	**Age**	**Sex**	**Student No.**	**Marks**
Values :	**John Brown**	**21**	**M**	**94XYZ23**	**75**

Priority can be based on name, age, student number, or marks

Operations performed on priority queues,

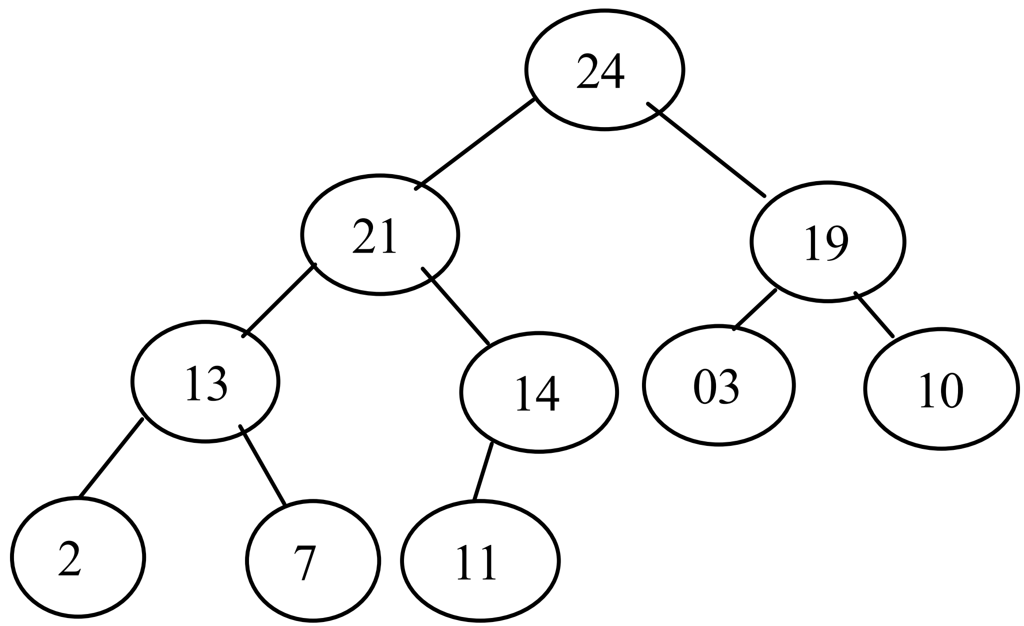
- **-inserting an element into the set**
- **-finding and deleting from the set an element of highest priority**

Priority Queues

Priority queues are implemented on partially ordered trees (POTs)

- **POTs are labeled binary trees**
- **the labels of the nodes are elements with a priority**
- **the element stored at a node has at least as large a priority as the elements stored at the children of that node**
- **the element with the highest priority is at the root of the tree**

Example



HEAPS

The **heap** is a data structure for implementing POT's

Each node of the heap tree corresponds to an element of the array that stores the value in the node

The tree is filled on all levels except possibly the lowest, which are filled from left to right up to a point.

An array A that represents a heap is an object with two attributes

***length*[A], the number of elements in the array and**

***heap-size*[A], the number of elements in the heap stored within the array A**

***heap_size*[A] $\leq$ *length*[A]**

HEAPS (Contd)

The heap comprises elements in locations up to *heap-size*[A] .
A[1] is the root of the tree.

Position	1	2	3	4	5	6	7	8	9	10
Value	24	21	19	13	14	3	10	2	7	11

Given node with index i,

PARENT(i) is the index of parent of i;PARENT(i) = $\lfloor i/2 \rfloor$

LEFT_CHILD(i) is the index of left child of i ;
LEFT_CHILD(i) = $2\times i$;

RIGHT_CHILD(i) is the index of right child of i; and
RIGHT_CHILD(i) = $2\times i$ +1

Heap Property

THE HEAP PROPERTY

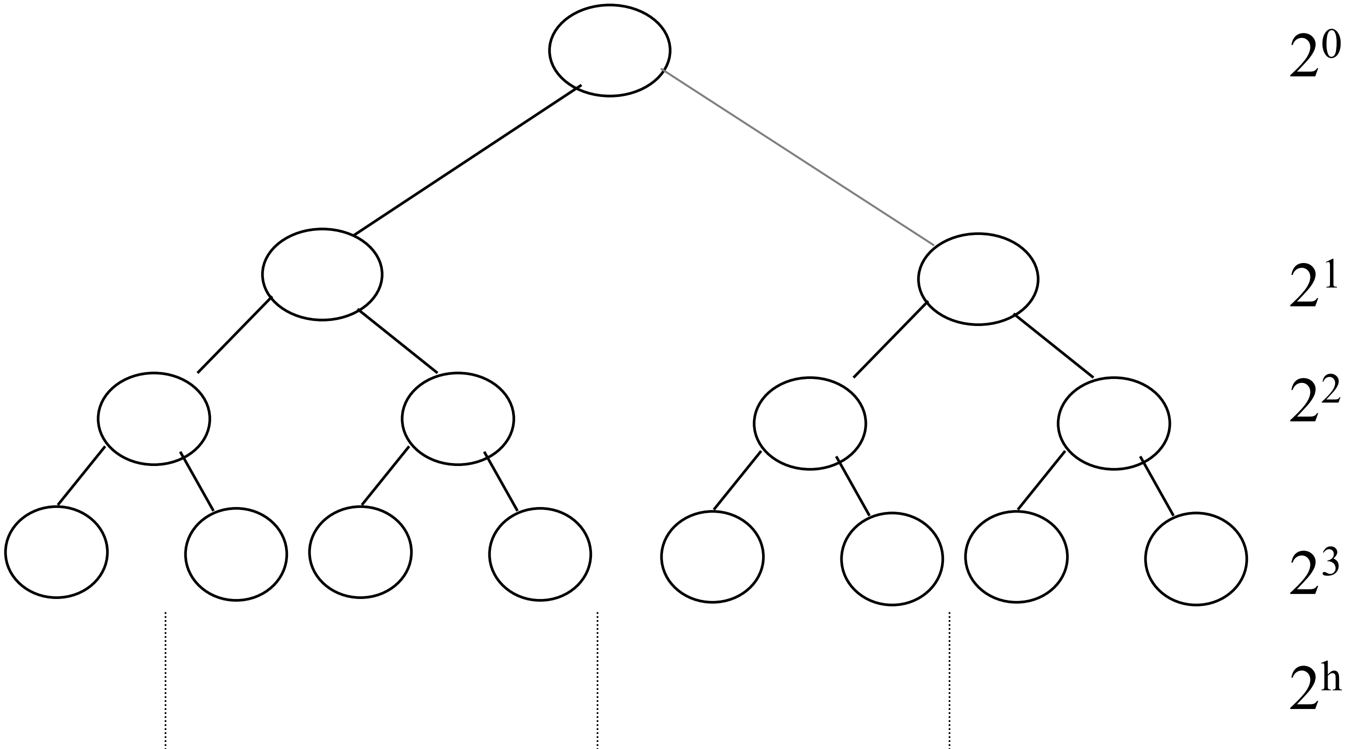
A[PARENT(*i*)] ≥ A[*i*]

The heap is based on a binary tree

The height of the heap (as a binary tree) is the number of edges on the longest simple downward path from the root to a leaf.

The height of a heap with n nodes is O (log n).

All basic operations on heaps run in O (log n) time.



$$\mathbf{n=2^0+2^1+2^2+2^3+\ldots+2^h=2^{h+1}-1}$$

Number of nodes at different levels in a Binary Tree

Heap Algorithms

HEAPIFY
BUILD_HEAP
HEAPSORT
HEAP_EXTRACT_MAX
HEAP_INSERT

HEAPIFY

The HEAPIFY algorithm checks the heap elements for violation of the heap property and restores heap property.

Procedure HEAPIFY (*A*,*i*)

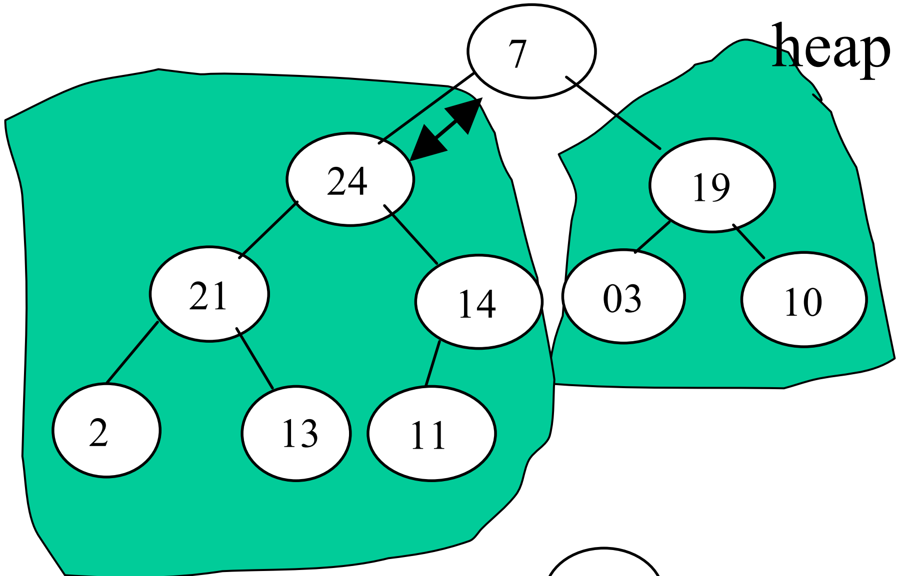
Input: An array A and index *i* to the array. *i* =1 if we want to heapify the whole tree. Subtrees rooted at LEFT_CHILD(*i*) and RIGHT_CHILD(*i*) are heaps

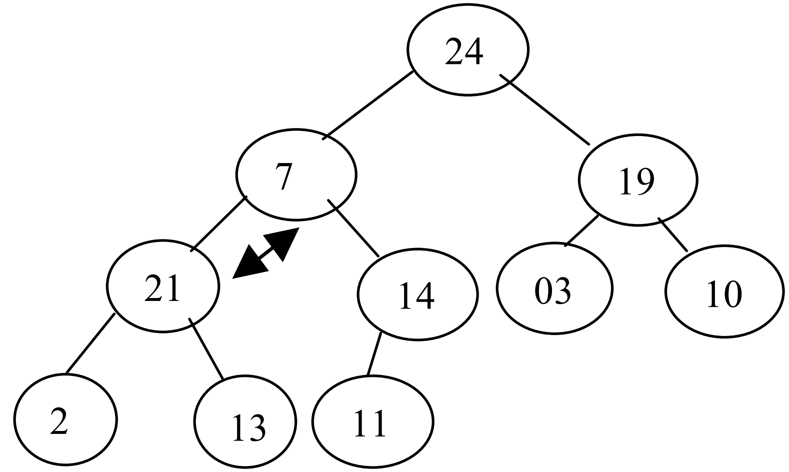
Output: The elements of array A forming subtree rooted at *i* satisfy the heap property.

```
l ← LEFT_CHILD (i);
r ← RIGHT_CHILD (i);
if l ≤ heap_size[A] and A[l] > A[i]
        then largest ← l;
        else largest ← i;
if r ≤ heap_size[A] and A[r] > A[largest]
        then largest ← r;
if largest ≠ i
        then exchange A[i] ↔ A[largest]
                HEAPIFY (A,largest)
```

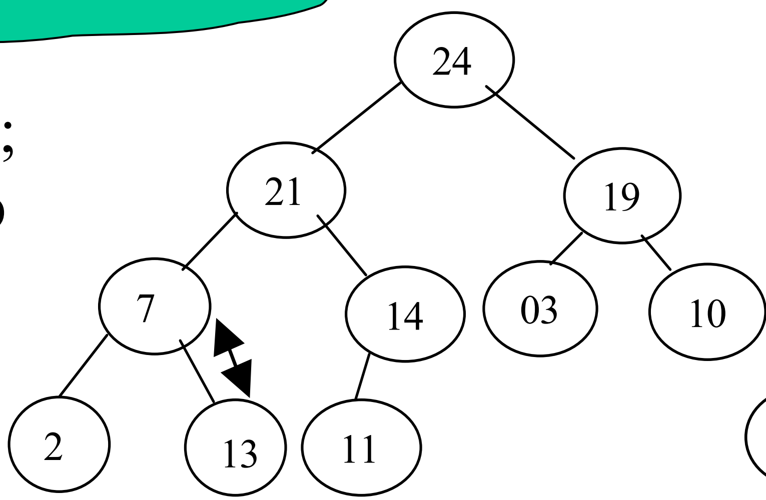
RST,

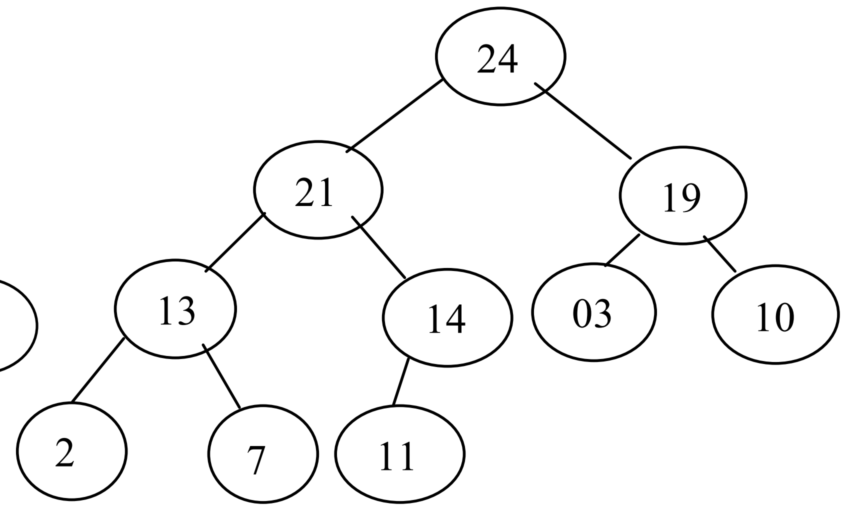
heap





LST;
heap





7	24	19	21	14	03	10	02	13	11
24	7	19	21	14	03	10	02	13	11
24	21	19	07	14	03	10	02	13	11
24	21	19	13	14	03	10	02	07	11

Running time of HEAPIFY

Total running time = steps 1 … 9 + recursive call

$T(n) = \Theta(1) + T(n/2)$

Solving the recurrence, we get $T(n) = O(\log n)$

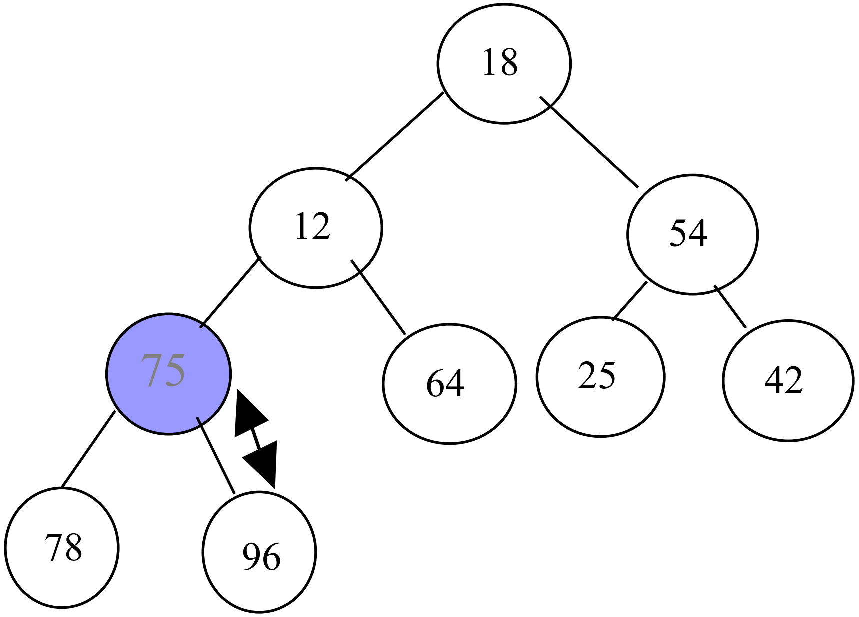
BUILD_HEAP

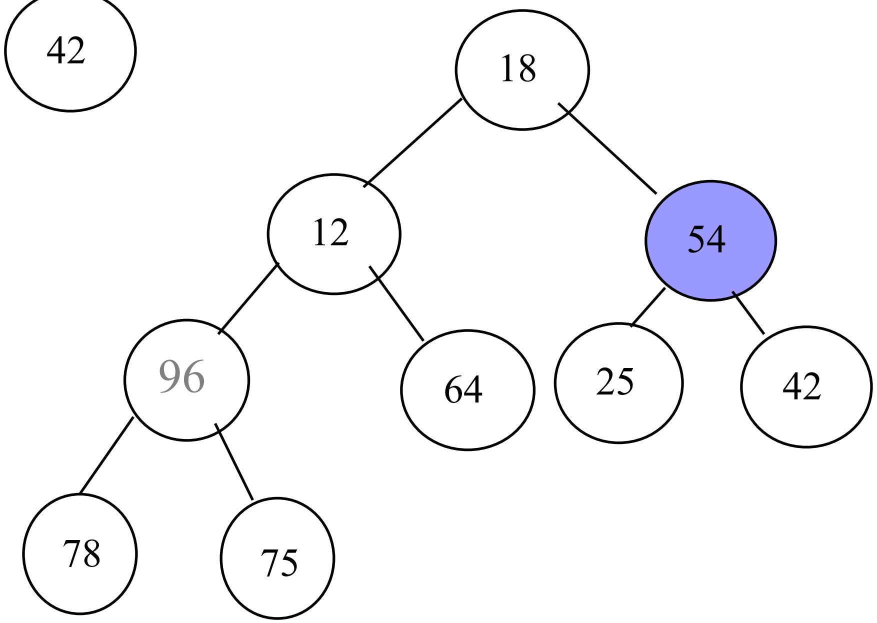
Procedure BUILD_HEAP (A)
Input : An array A of size n = length [A], heap_size[A]
Output : A heap of size n

1. *heap_size*[*A*] $\leftarrow$ *length*[*A*]
2. **for** *i* $\leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
3. HEAPIFY(*A*,*i*)

18	12	54	75	64	25	42	78	96
18	12	54	96	64	25	42	78	75
18	12	54	96	64	25	42	78	75
18	96	54	12	64	25	42	78	75
18	96	54	78	64	25	42	12	75
96	18	54	78	64	25	42	12	75
96	78	54	18	64	25	42	12	75
96	78	54	75	64	25	42	12	18

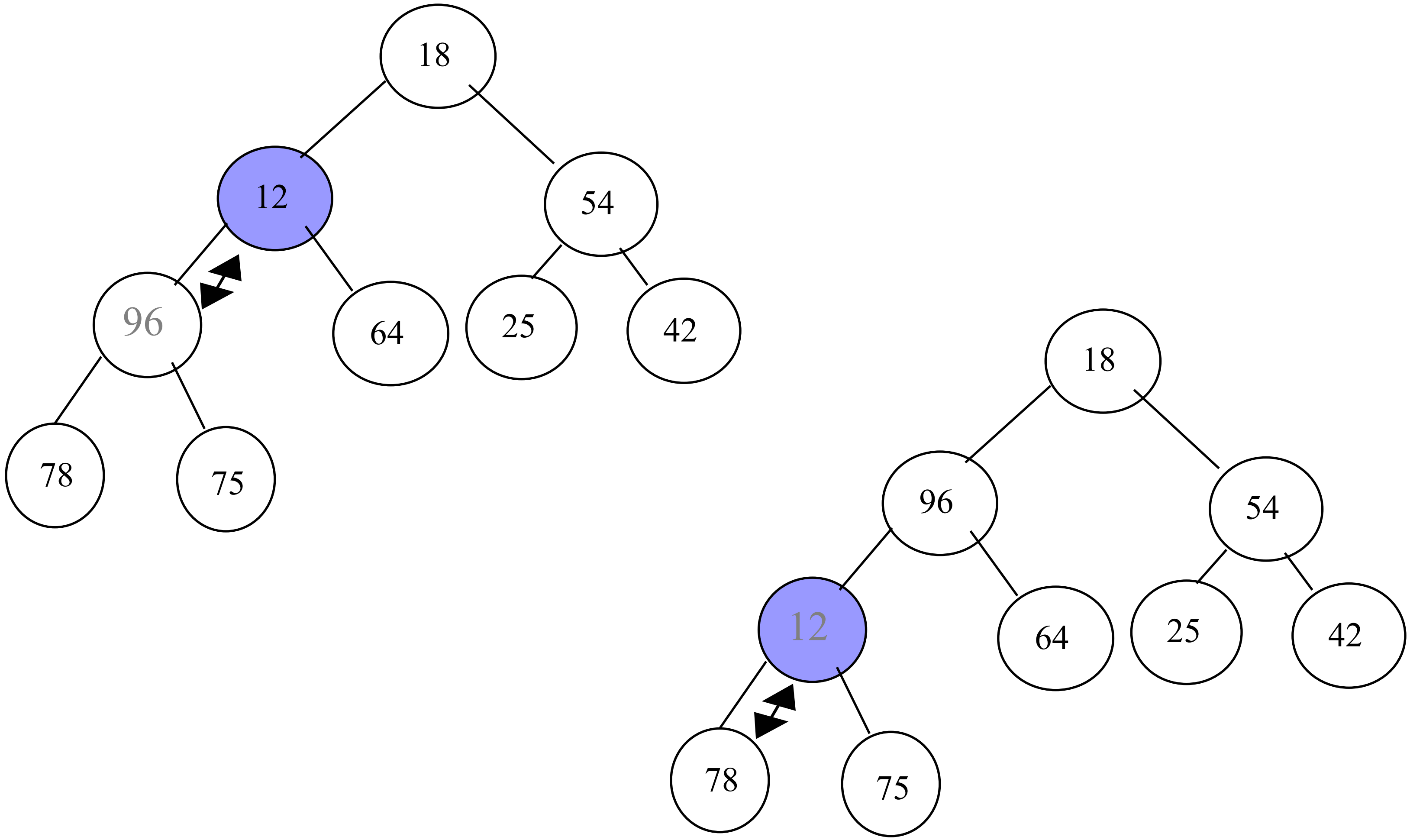
18 12 54 75 64 25 42 78 96





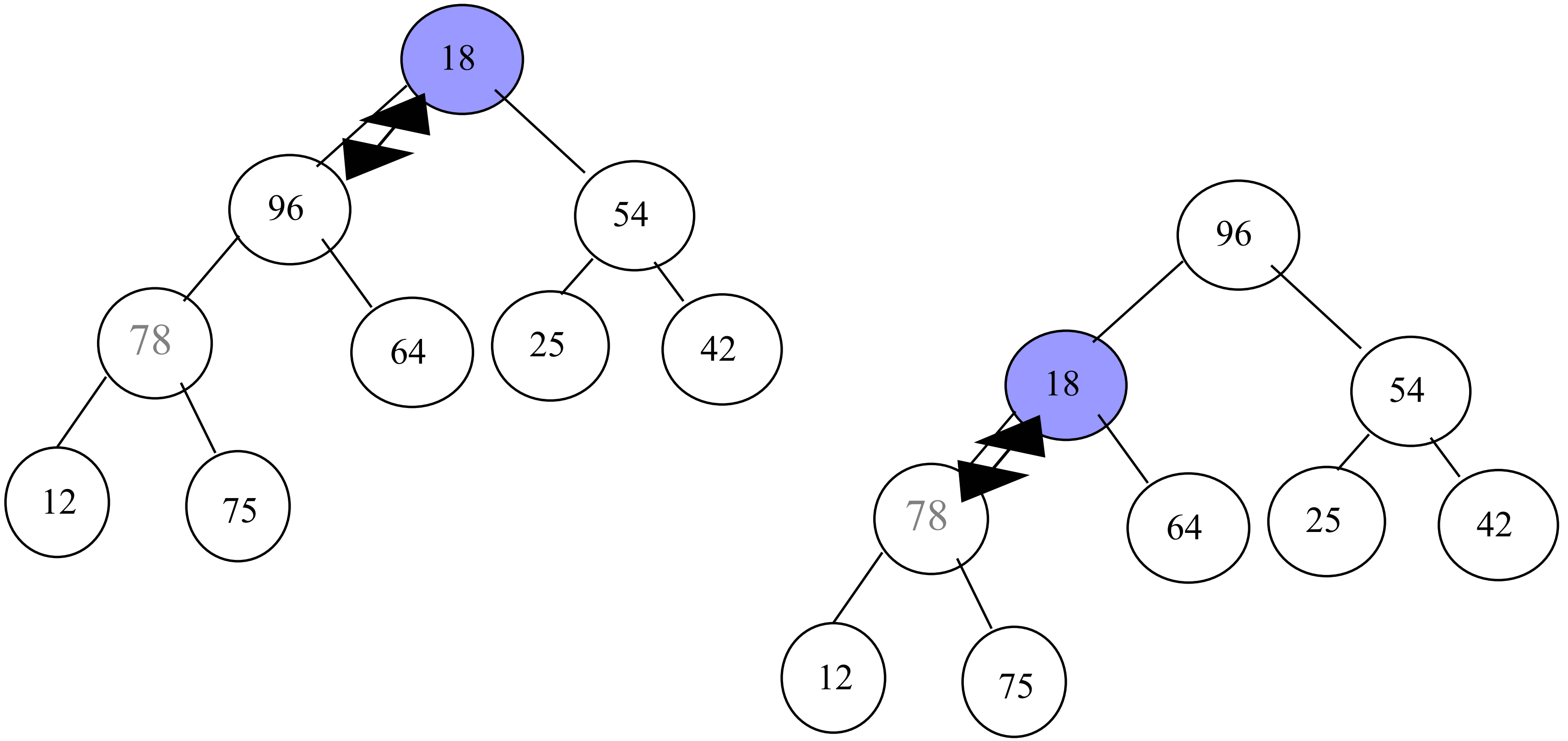
18 12 54 96 64 25 42 78 75

18 12 54 96 64 25 42 78 75



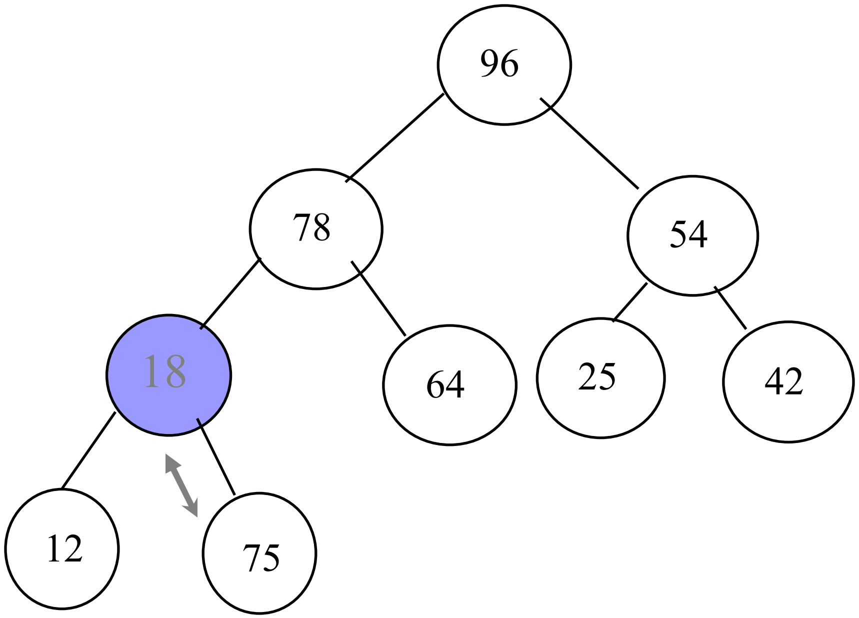
18 96 54 12 64 25 42 78 75

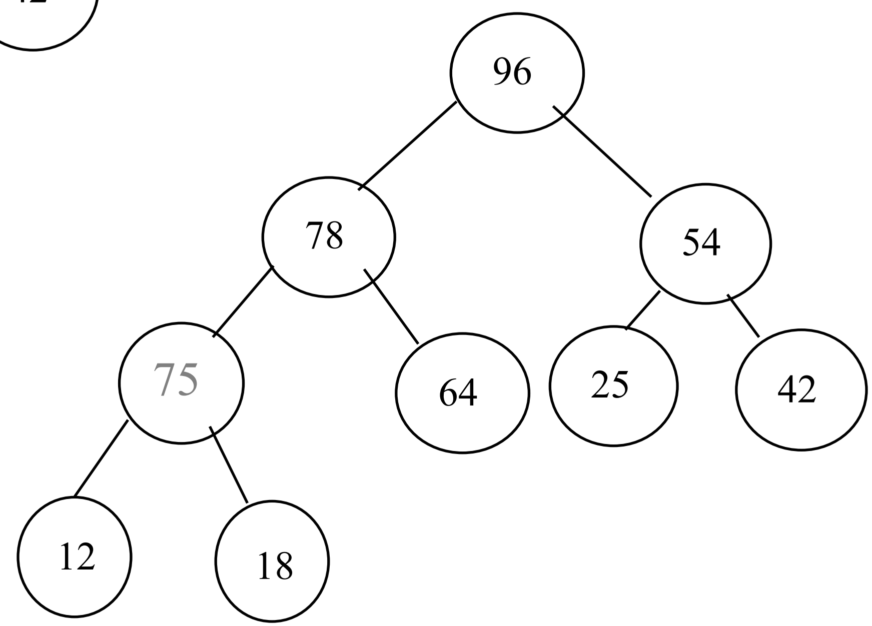
18 96 54 78 64 25 42 12 75

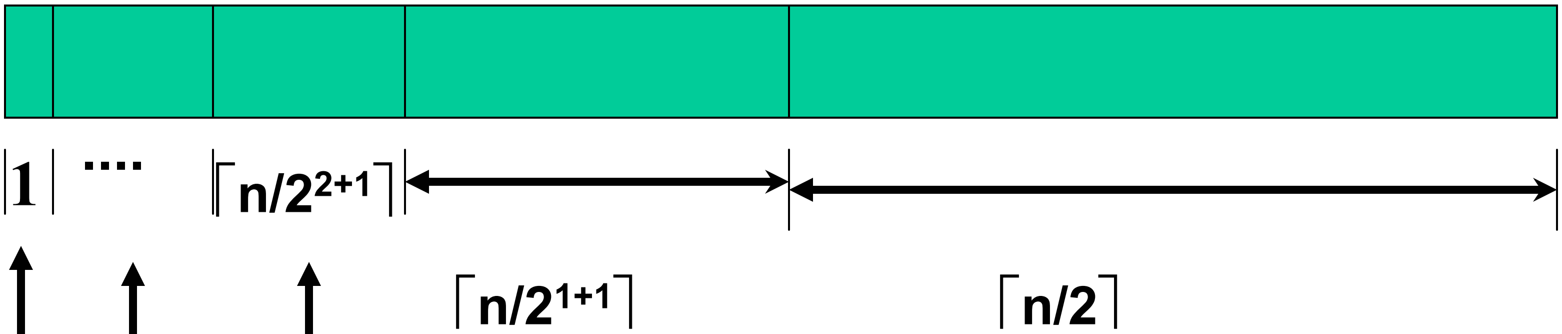


96 18 54 78 64 25 42 12 75

96 78 54 18 64 25 42 12 75







Height of each node = 1, at most 1 comparison

Height of each node = 2, at most 2 comparisons

Height of each node = i, at most i comparisons, $1 \le i \le h$

Height of the root node = h, at most h comparisons

Running time of Build_heap

1. **Each call to HEAPIFY takes O (log n) time**
2. **There are O (n) such calls**
3. **Therefore the running time is at most O(n logn)**

However the complexity of BUILD_HEAP is O(n)

Proof :

In an n element heap there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h

The time required to heapify a subtree whose root is at a height h is O(h)

(this was proved in the analysis for HEAPIFY)

So the total time taken for BUILD_HEAP is given by,

$$\le \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot h$$

$$\le \frac{n}{2} \cdot \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}$$

We know that $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$

$$= O(n)$$

Thus the running time of BUILD_HEAP is given by, O(n)

The HEAPSORT Algorithm

Procedure HEAPSORT(A)
Input : Array A[1…n], n = length[A]
Output : Sorted array A[1…n]

```
1.      BUILD_HEAP[A]
2.      for  i ← length[A] down to 2
3.              Exchange A[1] ↔ A[i]
4.              heap_size[A] ← heap_size[A]-1;
5.              HEAPIFY(A,1)
```

Example : To be given in the lecture

HEAPSORT

Running Time:
Step 1 BUILD_HEAP takes O(n) time,
Steps 2 to 5 : there are (n-1) calls to HEAPIFY which takes O(log n) time
Therefore running time takes O (n log n)

HEAP_EXTRACT_MAX

Procedure HEAP_EXTRACT_MAX(A[1…n])
Input : heap(A)
Output : The maximum element or root, heap (A[1…n-1])

1. **if** heap_size[A] $\geq$ 1
2. max $\leftarrow$ A[1];
3. A[1] $\leftarrow$ A[heap_size[A]];
4. heap_size[A] $\leftarrow$ heap_size[A]-1;
5. HEAPIFY(A,1)
6. **return** max

Running Time : O (log n) time

HEAP_INSERT

Procedure HEAP_INSERT(A, key)

Input : heap(A[1…n]), key - the new element

Output : heap(A[1…n+1]) with k in the heap

1. heap_size[A] $\leftarrow$ heap_size[A]+1;
2. i $\leftarrow$ heap_size[A];
3. **while** i > 1 and A[PARENT(i)] < key
4.   A[i] $\leftarrow$ A[PARENT(i)];
5.   i $\leftarrow$ PARENT(i);
6. A[i] $\leftarrow$ key

Running Time : O (log n) time

Questions:

- What is a heap?
- What are the running times for heap insertion and deletion operations ?
- Did you understand HEAPIFY AND and HEAPSORT algorithms
- Can you write a heapsort algorithm for arranging an array of numbers in descending order?

Quicksort

- Quicksort algorithm
- Quicksort performance
- Quicksort analysis

Quicksort

- **The worst case running time of Quicksort algorithm is $O(n^2)$**
- **However, its expected running time is O (n log n)**
- **Three-step divide-and-conquer process for sorting a subarray A[*l*..r]**

Divide : partition the array A[*l*..r] into two nonempty subarrays A[*l*..q] and A[q+1,r] such that each element of A[*l*..q] is less than or equal to each element of A[q+1,..,r]

Conquer : sort the two subarrays A[*l*..q] and A[q+1..r] by recursive calls to Quicksort

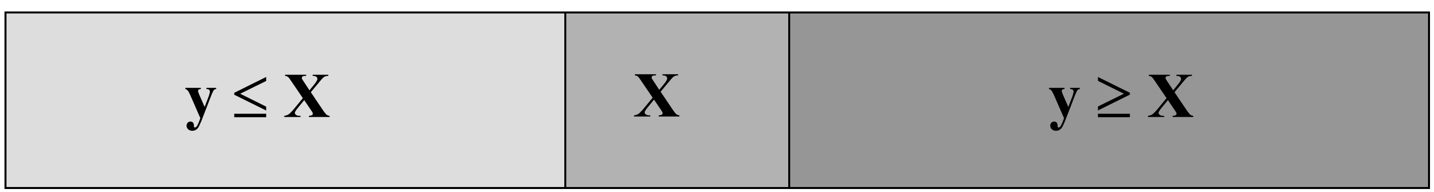
Combine : the subarrays are already sorted in place. No work is needed to combine them

Example

13	02	18	26	76	87	98	11	93	77	65	43	38	09	65	06
13	02	06	26	76	87	98	11	93	77	65	43	38	09	65	18
13	02	06	09	76	87	98	11	93	77	65	43	38	26	65	18
13	02	06	09	11	87	98	76	93	77	65	43	38	26	65	18
11	02	06	09	13	87	98	76	93	77	65	43	38	26	65	18
11	02	06	09	13	87	98	76	93	77	65	43	38	26	65	18
09	02	06	11	13	87	98	76	93	77	65	43	38	26	65	18
09	02	06	11	13	87	98	76	93	77	65	43	38	26	65	18
06	02	09	11	13	87	98	76	93	77	65	43	38	26	65	18
06	02	09	11	13	87	98	76	93	77	65	43	38	26	65	18
02	06	09	11	13	87	98	76	93	77	65	43	38	26	65	18
02	06	09	11	13	87	18	76	93	77	65	43	38	26	65	98
02	06	09	11	13	87	18	76	65	77	65	43	38	26	93	98
02	06	09	11	13	26	18	76	65	77	65	43	38	87	93	98
02	06	09	11	13	18	26	76	65	77	65	43	38	87	93	98
02	06	09	11	13	18	26	76	65	38	65	43	77	87	93	98
02	06	09	11	13	18	26	43	65	38	65	76	77	87	93	98
02	06	09	11	13	18	26	43	38	65	65	76	77	87	93	98
02	06	09	11	13	18	26	38	43	65	65	76	77	87	93	98
02	06	09	11	13	18	26	38	43	65	65	76	77	87	93	98

Procedure **Quicksort(A,*l*,r)**
Input : Unsorted Array (A,*l*,r)
Output : Sorted subarray A(0..r)
To sort the entire array A, ***l* = 1** and **r = length[A]**

```
  if l < r
     then q ← PARTITION(A,l,r)
          QUICKSORT(A,l,q-1)
          QUICKSORT(A,q+1,r)
```



Procedure **PARTITION(A,*l*,r)**

Input : Array $A(l \,..\, r)$

Output : A and q such that $A[i] \le A[q]$ for all $i \le q$ and
$A[j] > A[q]$ for all $j > q$.

$x \leftarrow A[l]$; $i \leftarrow l$; $j \leftarrow r$;
while $i < j$ **do**
  while $A[i] \le x$ and $i \le r$ **do** $i \leftarrow i + 1$;
  while $A[j] > x$ and $j \ge l$ **do** $j \leftarrow j - 1$;
  if $i < j$ **then**
    exchange $A[i] \leftrightarrow A[j]$;
$q \leftarrow j$;
exchange $A[l] \leftrightarrow A[q]$;

y ≤ X	**X**	**y ≥ X**

Running Time of Quicksort

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

It takes n-1 comparisons for the partition
Then we sort smaller sequences of size i-1 and n-i
Each element has the same probability of being selected as the pivot,
The average running time is given by,

$$T(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n} T(i-1) + \frac{1}{n}\sum_{i=1}^{n} T(n-i)$$

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

Detailed Analysis – ? Discussion

Mergesort

Mergesort

Like Quicksort, Mergesort algorithm also is based on the divide-and–conquer principle.

Divide: **This step computes the middle of the array, takes constant time, $\Theta(1)$**

Conquer : **Two subproblems, each of size n/2 are recursively solved.**
Each subproblem contributes 2T(n/2) to the running time.

Combine: **Two sorted sequences are merged, this takes $\Theta(n)$ time**

Example

13	02	18	26	76	87	98	11	93	77	65	43	38	09	65	06
13	02	18	26	76	87	98	11	93	77	65	43	38	09	65	06
13	02	18	26	76	87	98	11	93	77	65	43	38	09	65	06
02	13	18	26	76	87	98	11	93	77	65	43	38	09	65	06
02	13	18	26	76	87	98	11	93	77	65	43	38	09	65	06
02	*13*	*18*	*26*	76	87	98	11	93	77	65	43	38	09	65	06
02	*13*	*18*	*26*	76	87	98	11	93	77	65	43	38	09	65	06
02	*13*	*18*	*26*	76	87	98	11	93	77	65	43	38	09	65	06
02	*13*	*18*	*26*	*76*	*87*	*11*	*98*	93	77	65	43	38	09	65	06
02	*13*	*18*	*26*	*11*	*76*	*87*	*98*	93	77	65	43	38	09	65	06
02	*11*	*13*	*18*	*26*	*76*	*87*	*98*	93	77	65	43	38	09	65	06
02	*11*	*13*	*18*	*26*	*76*	*87*	*98*	06	09	38	43	65	65	77	99
02	*06*	*09*	*11*	*13*	*18*	*26*	*38*	43	65	65	76	77	87	93	98

Procedure MERGESORT(A,*l*,r)
Input : A an array in the range 1 to n.
Output: Sorted array A.

```
        if l < r
              then q ← ⌊(l+r )/2⌋ ;
                    MERGESORT(A,l,q)
                    MERGESORT(A,q+1, r)
                    MERGE (A,l,q,r)
```

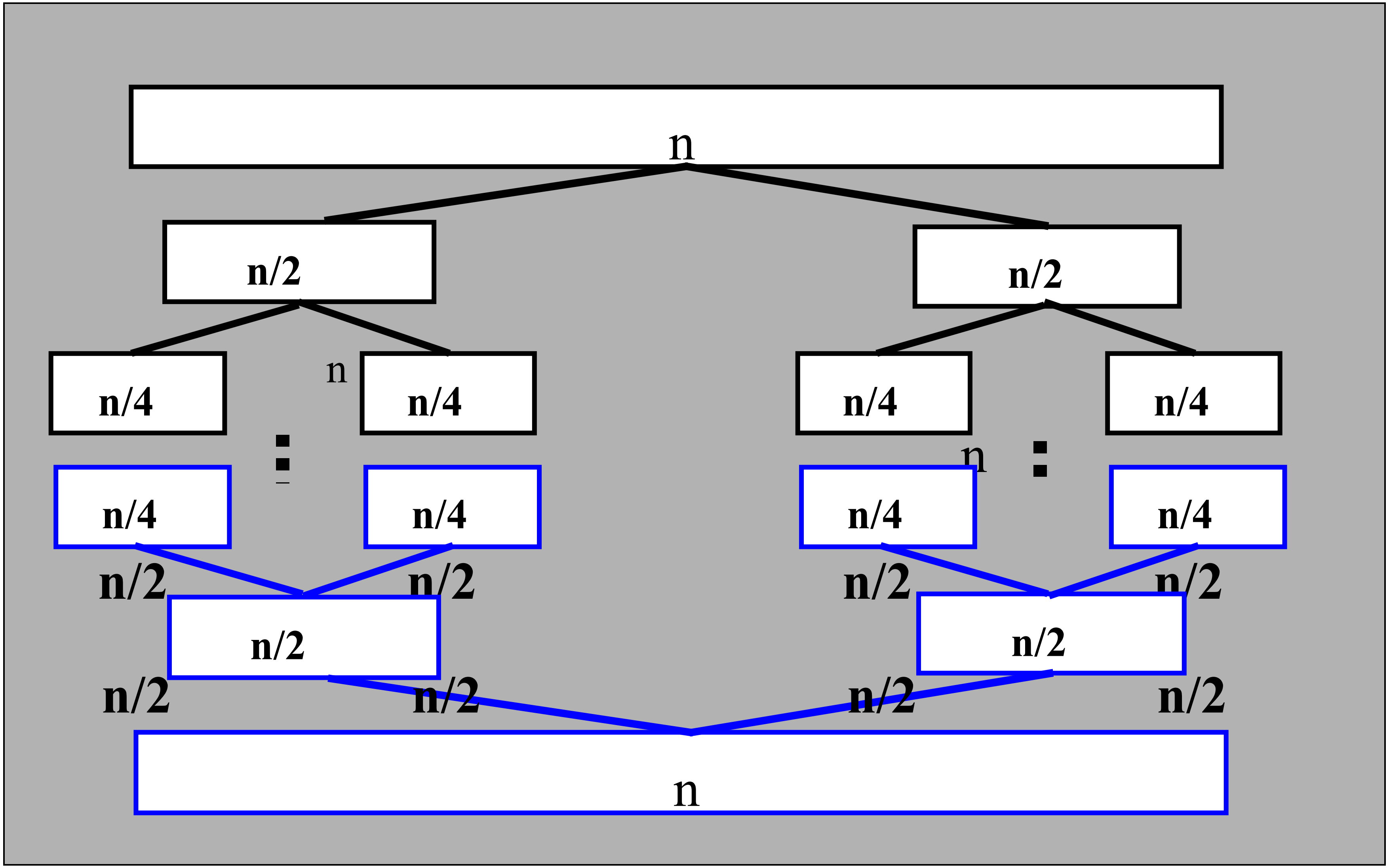
Procedure **MERGE(A,*l*,q,r)**

Inputs: **two sorted subarrays A(*l*, q) and A(q+1, r)**

Output : **Merged and sorted array A(*l*, r)**

```
i ← l;
j ← q+1;
k ← 0;
while (i ≤ q) and ( j ≤ r) do
        k  ← k+1;
        if A[i] ≤ A[j] then
                TEMP[k] ← A[i];
                i ← i +1;
        else
                TEMP[k] ← A[j];
                j ← j +1;
if j > r  then
        for t ←0 to q – i do
                A[r-t] ← A[q-t];
for t ← 0 to k-1 do
        A[l +t] ← TEMP[t];
```

Runtime Complexity of Mergesort



Runtime Complexity of Mergesort

$T(n) = 2\ T(n/2) + \Theta(n)$
$T(n/2) = 2\ T(n/4) + n/2$
$T(n/4) = 2\ T(n/8) + n/4$
$T(n) = 2\{2\ T(n/4) + n/2\} + n = 4\ T(n/4) + 2\ n$
$T(n) = 2^3\ T(n/2^3) + 3n$
. . .
$T(n) = 2^k\ T(n/2^k) + k\ n$ If $n = 2^k$ then , $k = \log n$

Therefore, $T(n) = 2^k\ T(1) + n \log n$
$= n\ \Theta(1) + n \log n$
$T(n) = O(n \log n)$

Questions

- What is a pivot element?
- Do you understand divide-and-conquer?
- What is running time if pivot element is at the center of the array?
- How is Mergesort different from Quicksort?
- Trace the algorithm with the help of an example?
- What is the use of TEMP array?
- Do you know why we execute steps 13 and 14 in the MERGE algorithm?
- What happens if i >q after the while loop (4-11)?

Home Work

- Find applications for selection sort, heapsort, quicksort, and mergesort sorting algorithms?
- Which of these problems are suitable for different types of sorting operations?
- Identify applications for which each sorting algorithm works best

Homework - Reference Books

- Insertion Sort
- Counting Sort
- Find the Maximum
- Find the Minimum