CSE5311 Design and Analysis of Algorithms

Administrivia
Introduction
Review of Basics

IMPORTANT

- **Americans With Disabilities Act**

  The University of Texas at Arlington is on record as being committed to both the spirit and letter of federal equal opportunity legislation; reference Public Law 93112 -- The Rehabilitation Act of 1973 as amended. With the passage of new federal legislation entitled Americans With Disabilities Act - (ADA), pursuant to section 504 of The Rehabilitation Act, there is renewed focus on providing this population with the same opportunities enjoyed by all citizens.

  As a faculty member, I am required by law to provide **"reasonable accommodation"** to students with disabilities, so as not to discriminate on the basis of that disability. Student responsibility primarily rests with **informing faculty at the beginning of the semester and in providing authorized documentation through designated administrative channels.**

Academic Dishonesty

It is the philosophy of The University of Texas at Arlington that academic dishonesty is a completely unacceptable mode of conduct and will not be tolerated in any form. All persons involved in academic dishonesty will be disciplined in accordance with University regulations and procedures. Discipline may include suspension or expulsion from the University.

"Scholastic dishonesty includes but is not limited to cheating, plagiarism, collusion, the submission for credit of any work or materials that are attributable in whole or in part to another person, taking an examination for another person, any act designed to give unfair advantage to a student or the attempt to commit such acts." (Regents' Rules and Regulations, Part One, Chapter VI, Section 3, Subsection 3.2, Subdivision 3.22)

- **Student Support Services Available**

  The University of Texas at Arlington supports a variety of student success programs to help you connect with the University and achieve academic success. These programs include learning assistance, developmental education, advising and mentoring, admission and transition, and federally funded programs. Students requiring assistance academically, personally, or socially should contact the Office of Student Success Programs at 817-272-6107 for more information and appropriate referrals.

IMPORTANT DATES

- Quiz 1 – September 24
- Quiz 2 – 1st week of November
- Exam1 – October 7
- Exam 2 – December 7
- Lab Assignment 1 – Due on October 14 (September 2)
- Lab Assignment 2 – Due on November 18(October 5)
- Research Problem – November 23

IMPORTANT

- Solve Problems ASAP
- Discuss with classmates, TA and Instructor
- Participate in the class
- Complete exercise problems
- Complete homework assignments
- Be creative

What are Algorithms ?

- **An algorithm is a precise and unambiguous specification of a sequence of steps that can be carried out to solve a given problem or to achieve a given condition.**
- **An algorithm is a computational procedure to solve a well defined computational problem.**
- **An algorithm accepts some value or set of values as input and produces a value or set of values as output.**
- **An algorithm transforms the input to the output.**
- **Algorithms are closely intertwined with the nature of the data structure of the input and output values.**

Data structures are methods for representing the data models on a computer whereas data models are abstractions used to formulate problems.

What are these algorithms? Input? Output? Complexity?

ALGO_DO_SOMETHING (A [1,…,n],1,n))

```
for i ← 1 to n-1
    small ← i;
        for j ← i+1 to n
            if A[j] < A[small] then
                small ← j;
        temp ← A[small];
        A[small] ← A[i];
        A[i] ← temp;
end
```

ALGO_IMPROVED (A[1,…,n],i,n)

```
while  i < n
    do   small ← i;
        for  j ← i+1 to n
            if A[j] < A[small] then
                small ← j;
        temp ← A[small];
        A[small] ← A[i];
        A[i] ← temp;
        ALGO_IMPROVED(A,i+1,n)
End
```


Examples

- **Algorithms:**

An algorithm to sort a sequence of numbers into nondecreasing order.

Application : lexicographical ordering

An algorithm to find the shortest path from a source node to a destination node in a graph

Application : To find the shortest path from one city to another.

- **Data Models:**

  Lists, Trees, Sets, Relations, Graphs

- **Data Structures :**

  Linked List is a data structure used to represent a List

  Graph is a data structure used to represent various cities in a map.

SELECTION SORT Algorithm (*Iterative method)*

Procedure SELECTION_SORT (A [1,…,n])
Input : unsorted array A
Output : Sorted array A

```
1.    for i ← 1 to n-1
2.       small ← i;
3.          for j ← i+1 to n
4.             if A[j] < A[small] then
5.                small ← j;
6.          temp ← A[small];
7.          A[small] ← A[i];
8.          A[i] ← temp;
9.     end
```

Example: Given sequence

	5	2	4	6	1	3
i=1	1	2	4	6	5	3
i=2	1	2	4	6	5	3
i=3	1	2	3	6	5	4
i=4	1	2	3	4	5	6


```
1.    for i ← 1 to n-1
2.       small ← i;
3.          for j ← i+1 to n
4.             if A[j] < A[small] then
5.                small ← j;
6.          temp ← A[small];
7.          A[small] ← A[i];
8.          A[i] ← temp;
9.    end
```

Complexity:

The statements 2,6,7,8, and 5 take O(1) or constant time.
The outerloop 1-9 is executed n-1 times and the inner loop 3-5 is executed (n-i) times.
The upper bound on the time taken by all iterations as i ranges from 1 to n-1 is given by, **$O(n^2)$**

- Study of algorithms involves,
  - **designing algorithms**
  - **expressing algorithms**
  - **algorithm validation**
  - **algorithm analysis**
  - **Study of algorithmic techniques**

Algorithms and Design of Programs

- ***An algorithm is composed of a finite set of steps,***
  - **each step may require one or more operations,**
  - **each operation must be definite and effective**
- ***An algorithm,***
  - **is an abstraction of an actual program**
  - **is a computational procedure that terminates**

***A program is an expression of an algorithm in a programming language.**

***Choice of proper data models and hence data structures is important for expressing algorithms and implementation.**

- We evaluate the performance of algorithms based on time (CPU-time) and space (semiconductor memory) required to implement these algorithms. However, both these are expensive and a computer scientist should endeavor to minimize time taken and space required.

- The time taken to execute an algorithm is dependent on one or more of the following,
  - number of data elements
  - the degree of a polynomial
  - the size of a file to be sorted
  - the number of nodes in a graph

Asymptotic Notations

– **O-notation**

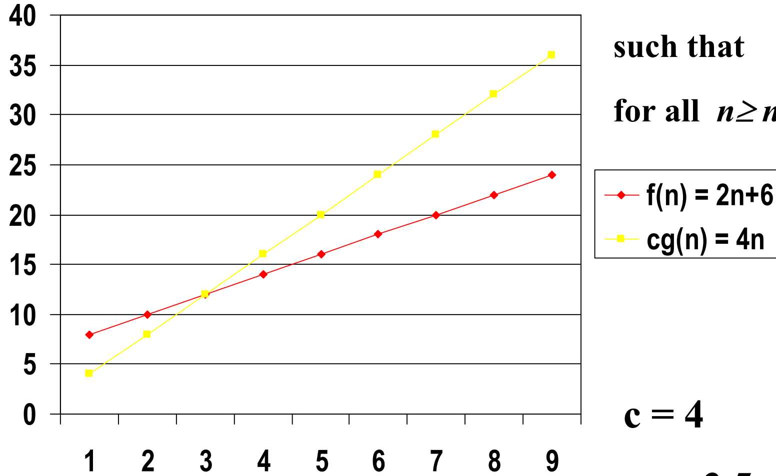
» ***Asymptotic upper bound***

- **A given function $f(n)$, is $O(g(n))$ if there exist positive constants c and n_0 such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0$.**
- **$O(g(n))$ represents a set of functions, and**

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0\}$.

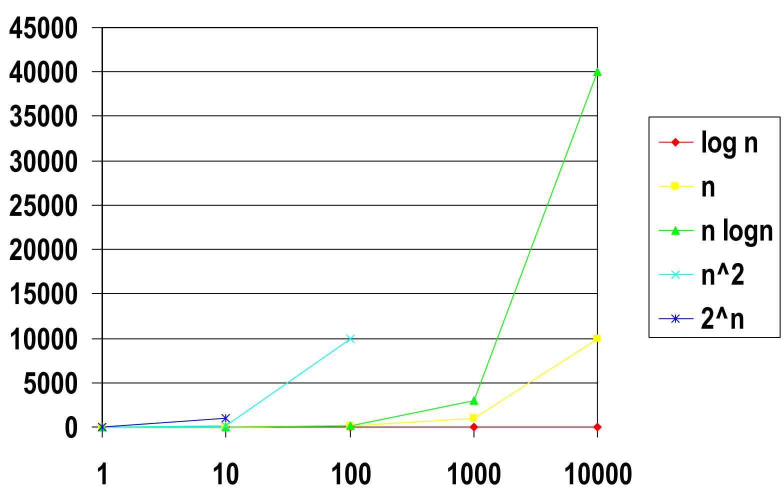
O Notation

f(n), is *O (g(n))* if there exist positive constants *c* and n_0 such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0$.



c = 4

$\mathbf{n_0 = 3.5}$



Ω-notation

Asymptotic lower bound

- A given function $f(n)$, is $\Omega(g(n))$ if there exist positive constants c and n_0 such that $0 \leq c\, g(n) \leq f(n)$ for all $n \geq n_0$.
- $\Omega(g(n))$ represents a set of functions, and

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq c\, g(n) \leq f(n)$ for all $n \geq n_0\}$

Θ-notation

Asymptotic tight bound

- A given function $f(n)$, is $\Theta(g(n))$ if there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq n_0$.
- $\Theta(g(n))$ represents a set of functions, and

$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq n_0$.

O, Ω, and Θ correspond (loosely) to "≤", "≥", and "=".

Presenting algorithms

- **Description : The algorithm will be described in English, with the help of one or more examples**
- **Specification : The algorithm will be presented as pseudocode**

  (We don't use any programming language)
- **Validation : The algorithm will be proved to be correct for all problem cases**
- **Analysis: The running time or time complexity of the algorithm will be evaluated**

SELECTION SORT Algorithm (*Iterative method)*

Procedure SELECTION_SORT (A [1,…,n])
Input : unsorted array A
Output : Sorted array A

```
for i ← 1 to n-1
   small ← i;
      for j ← i+1 to n
            if A[j] < A[small] then
               small ← j;
      temp ← A[small];
      A[small] ← A[i];
      A[i] ← temp;
end
```

Recursive Selection Sort Algorithm

Given an array A[i, …,n], selection sort picks the smallest element in the array and swaps it with A[i], then sorts the remainder A[i+1, …, n] recursively.

Example :
Given A [26, 93, 36, 76, 85, **09**, 42, 64]

Swap 09 with 23, A[1] = 09, A[2,…, 8] = [93,36,76,85,26,42,64]
Swap 26 with 93, A[1,2]= [09,26]; A[3,…,8] = [36,76,85,93,42,64]
No swapping A[1,2,3] = [09,26,36]; A[4,…,8] =[76,85,93,42,64]
Swap 42 with 76, A[1,…,4] =[09,26,36,42]; A[5,…,8] = [85,93,76,64]
Swap 64 with85, A[1,…,5] =[09,26,36,42,64]; A[6,7,8] = [93,76,85]
Swap 76 with 93, A[1,…,6]=[09,26,36,42,64,76]; A[7,8] = [93,85]
Swap 85 with 93, A[1,…,7]=[09,26,36,42,64,76,85]; A[8] = 93
Sorted list : A[1,…,8] = [09,26,36,42,64,76,85,93]

Procedure **RECURSIVE_SELECTION_SORT (A[1,…,n],i,n)**
Input : Unsorted array A
Output : Sorted array A

```
while i < n
   do small ← i;
      for j ← i+1 to n
          if A[j] < A[small] then
             small ← j;
      temp ← A[small];
      A[small] ← A[i];
      A[i] ← temp;
      RECURSIVE_SELECTION_SORT(A,i+1,n)
End
```

Analysis of Recursive selection sort algorithm

Basis: If i = n, then only the last element of the array needs to be sorted, takes Θ (1) time.

Therefore, T(1) = *a*, a constant

Induction : if i < n, then,

1. we find the smallest element in A[i,…,n], takes at most (n-1) steps

swap the smallest element with A[i], one step

recursively sort A[i+1, …, n], takes T(n-1) time

Therefore, T(n) is given by,

T(n) = T(n-1) + b. n (1)

It is required to solve the recursive equation,

T(1) = a; for n =1

T(n) = T(n-1) + b n; for n >1, where b is a constant

T(n-1) = T(n-2) + (n-1)b (2)

T(n-2) = T(n-3) + (n-2) b (3)

. . .

T(n-i) = T(n-(i+1)) + (n-i)b (4)

Using (2) in (1)

T(n) = T(n-2) + b [n+(n-1)]

= T(n-3) + b [n+(n-1)+(n-2)

= T(n-(n-1)) + b[n+(n-1)+(n-2) + . . . +(n-(n-2))]

T(n) = O(n^2)

Questions:

- **What is an algorithm?**
- **Why should we study algorithms?**
- **Why should we evaluate running time of algorithms?**
- **What is a recursive function?**
- **What are the basic differences among O, Ω, and Θ notations?**
- **Did you understand selection sort algorithm and its running time evaluation?**
- **Can you write pseudocode for selecting the largest element in a given array?**

Please write the algorithm now.

Heaps and Heapsort

Further Reading Chapters 6 from Textbook

This week

- **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

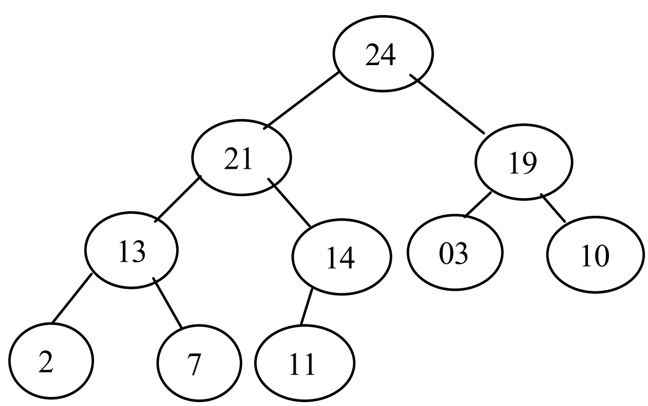
Operations performed on priority queues,
- **-inserting an element into the set**
- **-finding and deleting from the set an element of highest priority**

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] ≤ *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×*i*;

RIGHT_CHILD(*i*) is the index of right child of *i*; and
RIGHT_CHILD(*i*) = 2×*i* +1

Heap Property

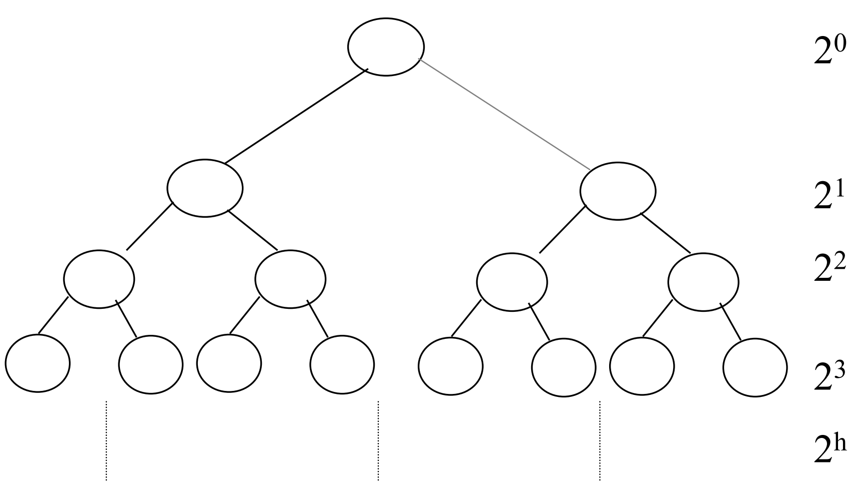
THE HEAP PROPERTY
A[PARENT(i)] $\geq$ A[i]

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.



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

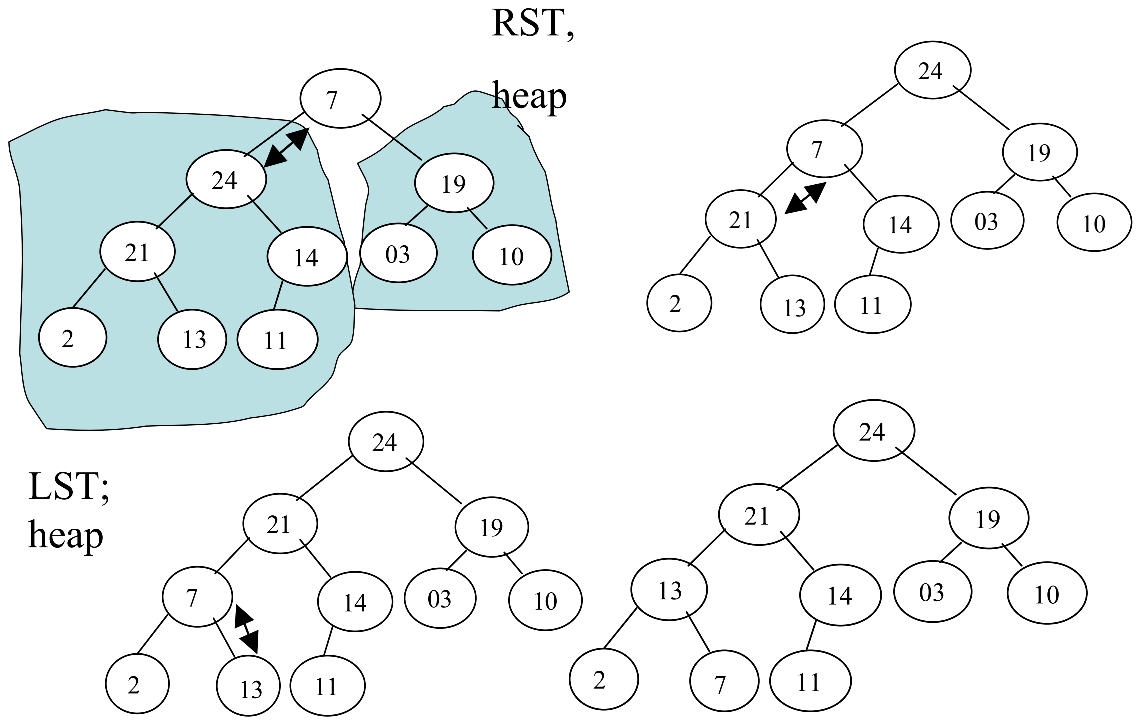
Heap Algorithms

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)
```



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) = Θ (1) + T (n/2)
Solving the recurrence, we get T (n) = O (log n)

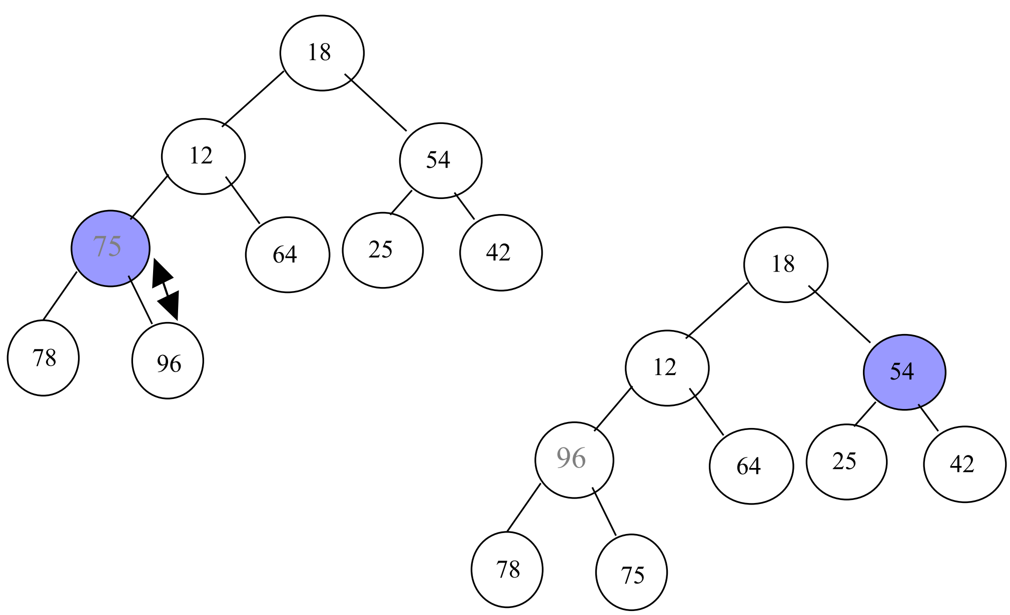
BUILD_HEAP

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

heap_size[*A*] ← *length*[*A*]
for *i* ← ⌊*length*[*A*]/2⌋ **downto** 1
HEAPIFY(*A*,*i*)

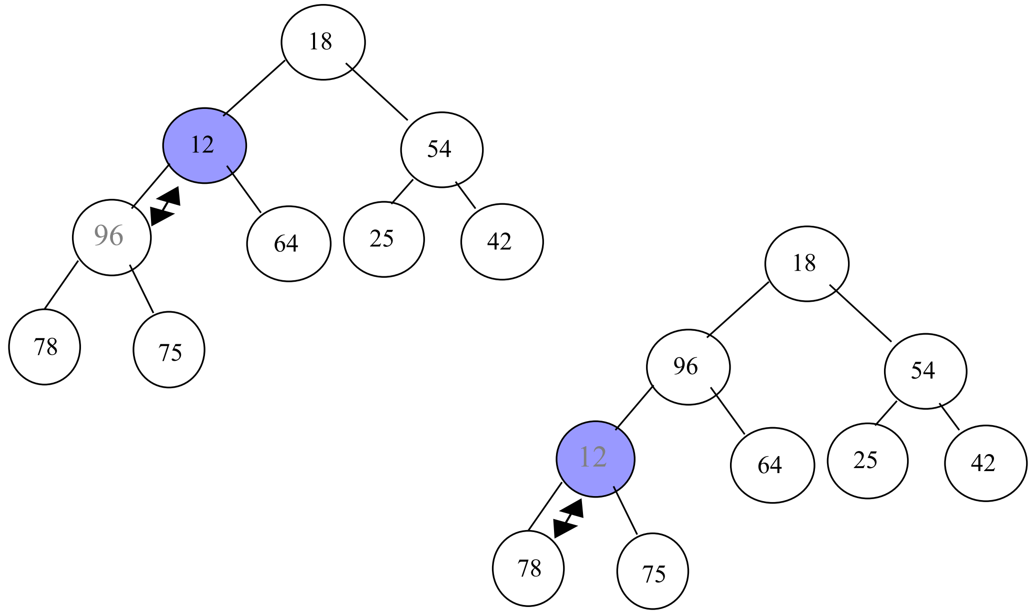
18	12	54	75	64	25	42	78	96
18	12	54	96	64	25	42	78	75
18	12	54	96	64	25	42	78	75
18	96	54	12	64	25	42	78	75
18	96	54	78	64	25	42	12	75
96	18	54	78	64	25	42	12	75
96	78	54	18	64	25	42	12	75
96	78	54	75	64	25	42	12	18

18 12 54 75 64 25 42 78 96



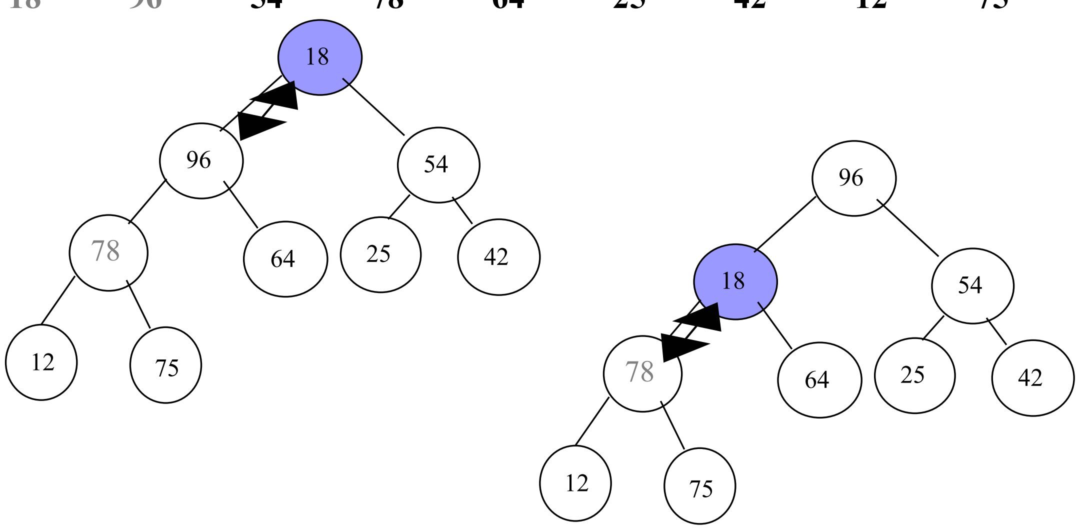
18 12 54 96 64 25 42 78 75

18 12 54 96 64 25 42 78 75



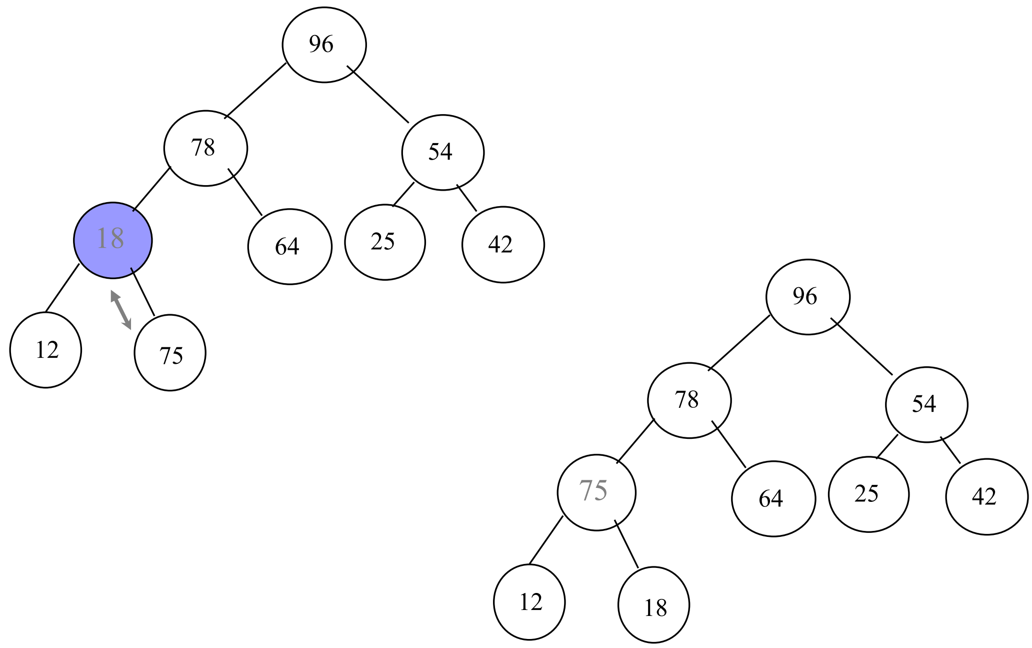
18 96 54 12 64 25 42 78 75

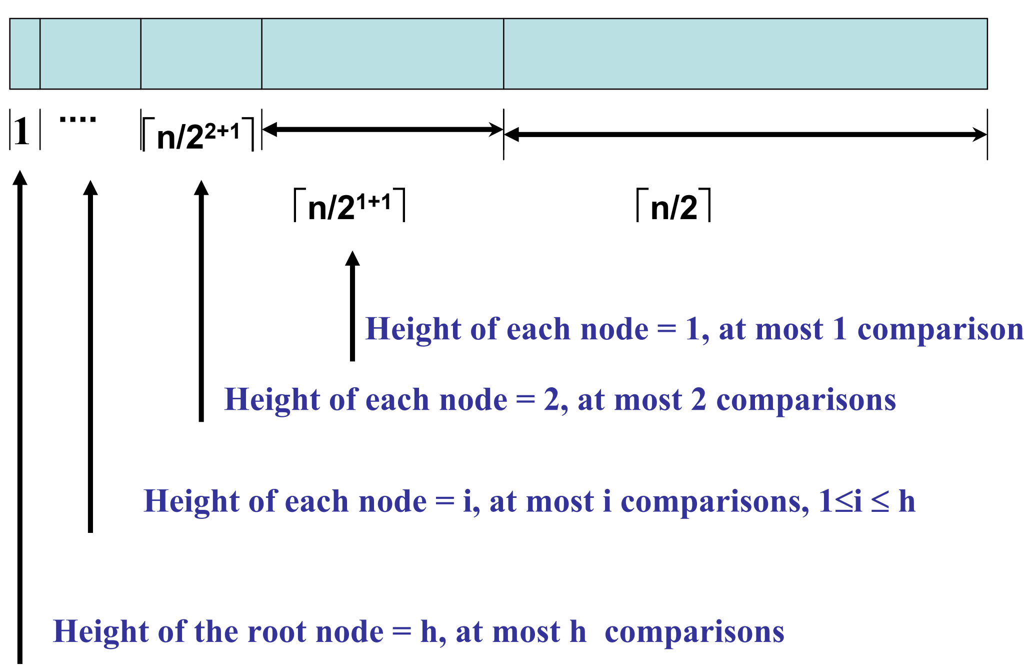
18 96 54 78 64 25 42 12 75



96 18 54 78 64 25 42 12 75

96 78 54 18 64 25 42 12 75





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,

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

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

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 $\leftarrow$ length[A] down to 2
3. Exchange A[1] $\leftrightarrow$ A[i]
4. heap_size[A] $\leftarrow$ 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])

```
if heap_size[A] ≥ 1
        max ← A[1];
        A[1] ← A[heap_size[A]];
        heap_size[A] ← heap_size[A]-1;
        HEAPIFY(A,1)
        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

```
heap_size[A] ← heap_size[A]+1;
i ← heap_size[A];
while i > 1 and A[PARENT(i)] < key
        A[i] ← A[PARENT(i)];
        i ← PARENT(i);
A[i] ← 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?**