CPSC 319

Data Structures, Algorithms and Their Application

Department of Computer Science

Created by:

Fahim Anzum

UNIVERSITY OF CALGARY

Winter 2020

Course Information

- **Course website:**

All course materials, such as lecture slides, important dates, assignments and exercises can be found on the course website, which is located at http://pages.cpsc.ucalgary.ca/~hudsonj/CPSC319W20/

- **For Assignment and Grading:**

D2L will be used for submitting assignments and reporting grades.

https://d2l.ucalgary.ca/d2l/home

- **Tutorial Contents:**

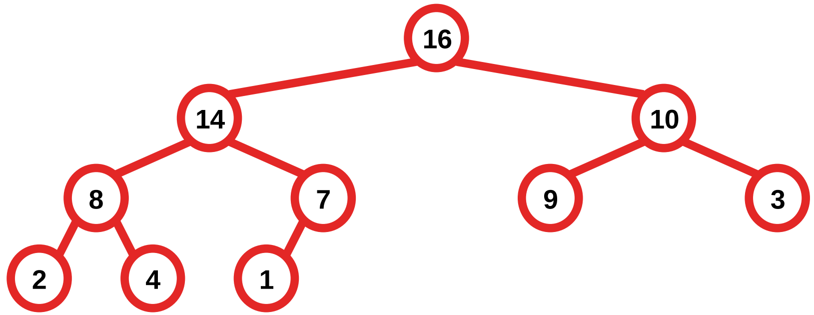
Will be uploaded in the following site:

https://sites.google.com/view/fahimanzum/courses

Binary Heaps

- The (binary) heap data structure is an array object that can be viewed as a complete binary tree
  - Each node of the tree corresponds to an element of the array that stores the value in the node.
  - The tree is completely filled on all levels except possibly the lowest, where it is filled from the left up to a point.

A =	*16*	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*1*



Binary Heaps

- To represent a complete binary tree as an array:
  - The root node is A[1]
  - Node *i* is A[*i*]
  - The parent of node *i* is A[*i*/2]
  - The right child of node *i* is A[2*i*+1]
  - The left child of node *i* is A[2*i*]

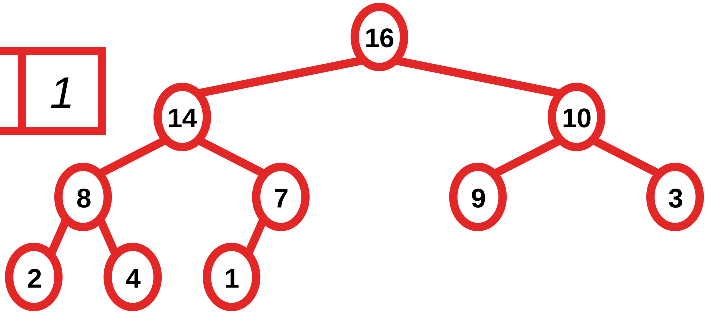
```
Parent(i)
    return floor(i/2)

Right(i)
    return 2i+1

Left(i)
    return 2i
```

A =

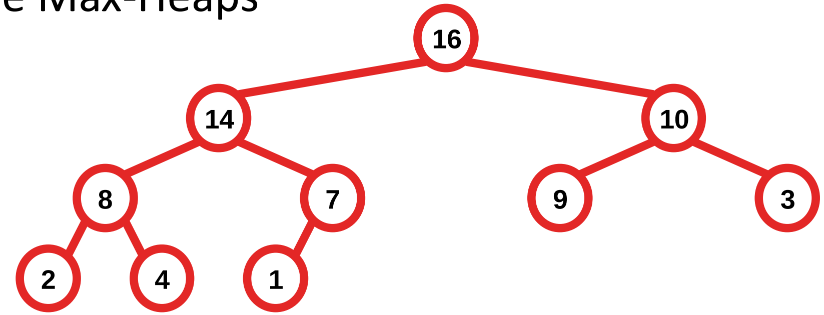
16	14	10	8	7	9	3	2	4	1



Types of Binary Heaps

- **Min-Heaps:**
  - The element in the root is less than or equal to all elements in both of its sub-trees
  - Both of its sub-trees are Min-Heaps

- **Max-Heaps:**
  - The element in the root is greater than or equal to all elements in both its sub-trees
  - Both of its sub-trees are Max-Heaps



The Heap Property

- Max-Heaps satisfy the *heap property*:

  $A[\mathit{Parent}(i)] \geq A[i]$ for all nodes $i > 1$

  - In other words, the value of a node is at most the value of its parent
  - The largest element in a max-heap is stored at the root

- Min-Heaps satisfy the *heap property*:

  $A[\mathit{Parent}(i)] \leq A[i]$ for all nodes $i > 1$

  - In other words, the value of a node is at least the value of its parent
  - The smallest element in a min-heap is stored at the root

Max-Heap Operations: Max-Heapify()

- `Max-Heapify()`: maintain the max-heap property
  - Given: a node *i* in the heap with children *l* and *r*
  - Given: two subtrees rooted at *l* and *r*, assumed to be heaps
  - Problem: The subtree rooted at *i* may violate the heap property
  - Action: let the value of the parent node "float down" so subtree at *i* satisfies the heap property

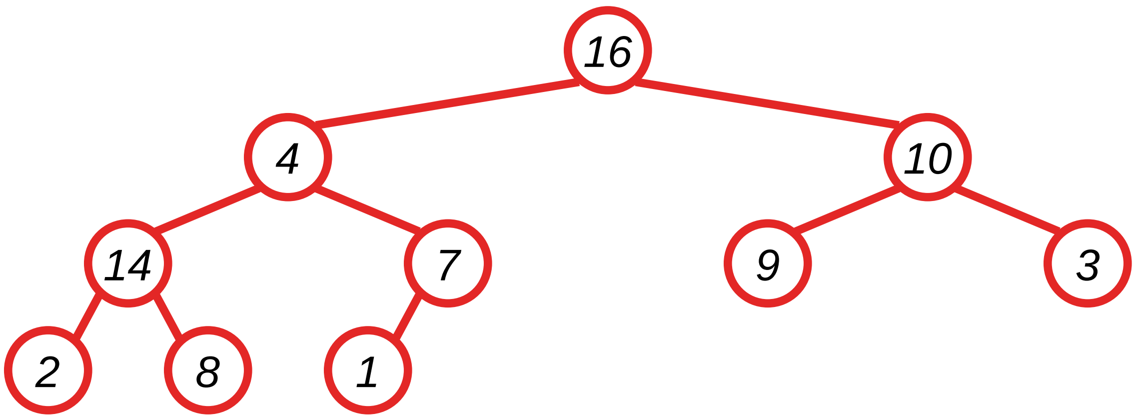
UNIVERSITY OF CALGARY

Max-Heap Operations: Max-Heapify()

Assumed that the binary trees rooted at LEFT(i) and RIGHT(i) are already max-heaps.

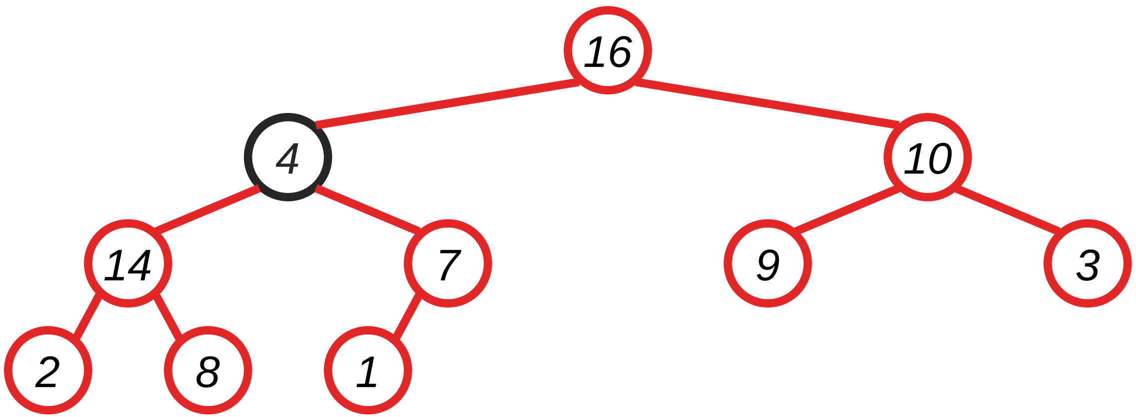
```
MAX-HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ heap-size[A] and A[l] > A[i]
    then largest ← l
    else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
    then largest ← r
if largest ≠ i
    then exchange A[i] ↔ A[largest]
         MAX-HEAPIFY(A, largest)
```

Heapify() Example



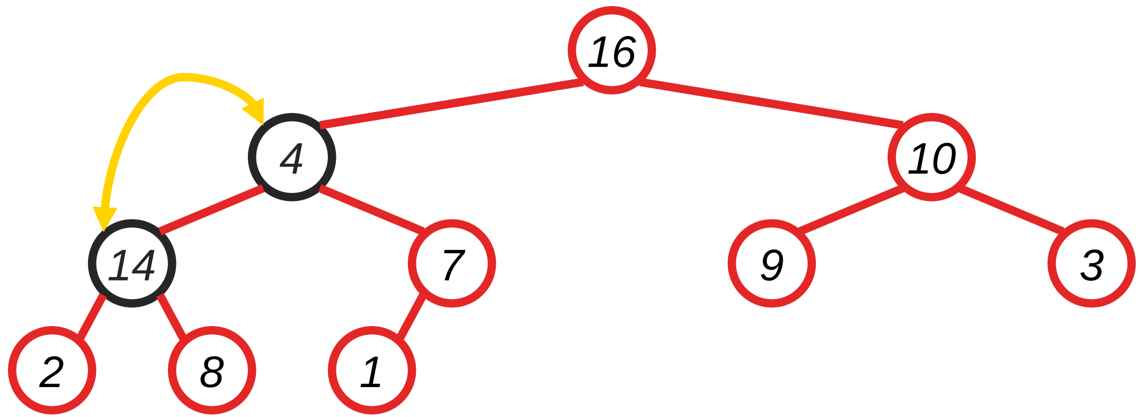
A =	16	4	10	14	7	9	3	2	8	1

Heapify() Example



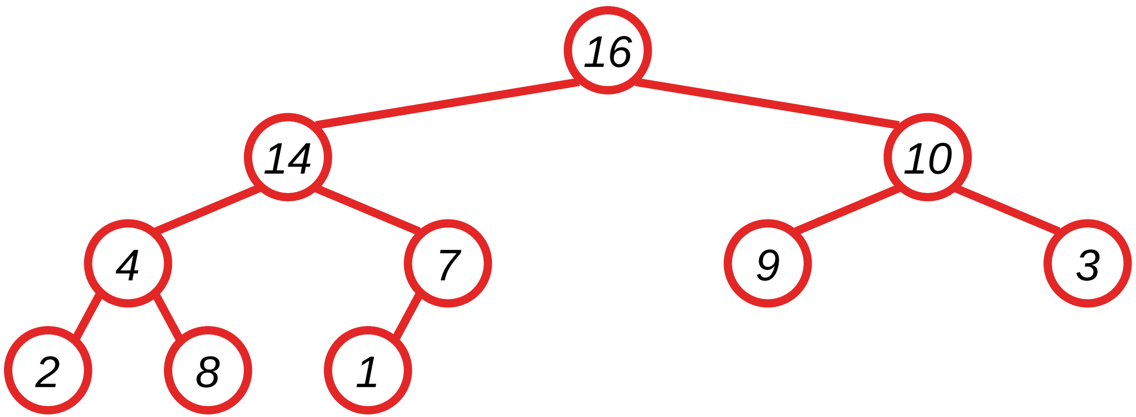
A =	16	4	10	14	7	9	3	2	8	1

Heapify() Example



A =	16	4	10	14	7	9	3	2	8	1

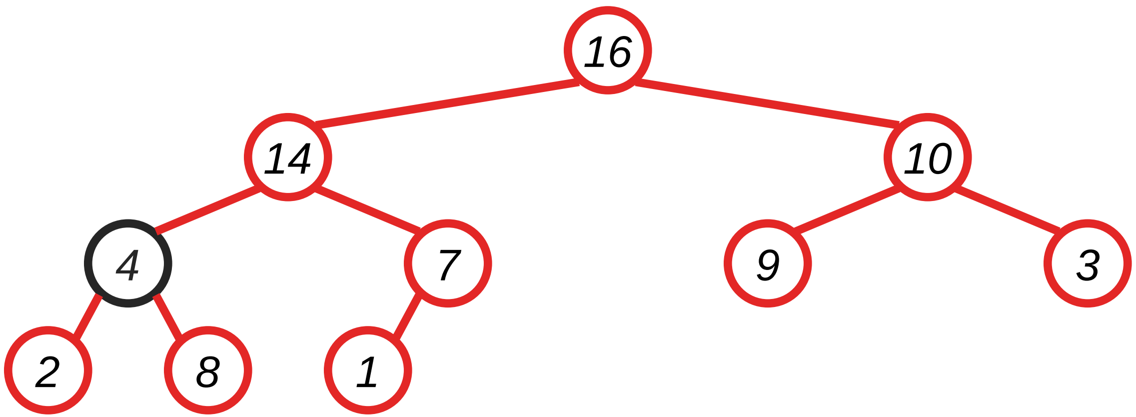
Heapify() Example



$A =$

16	14	10	4	7	9	3	2	8	1

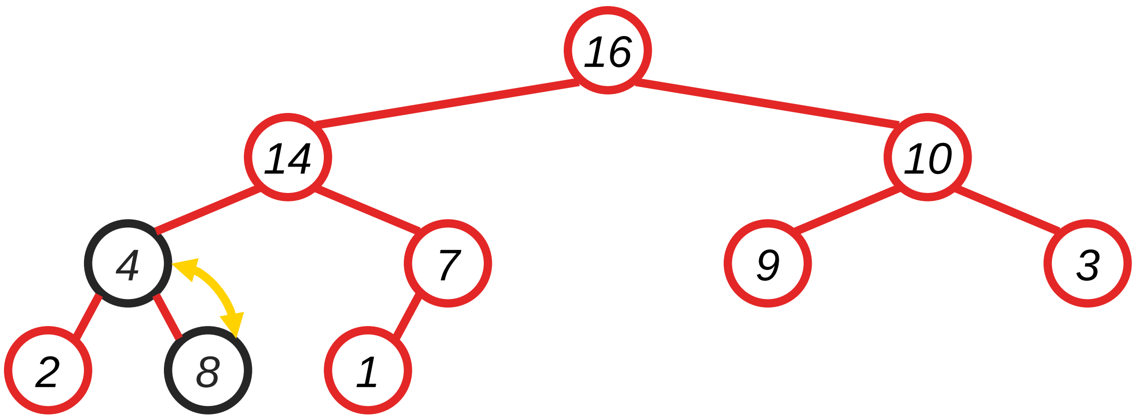
Heapify() Example



A =

16	14	10	4	7	9	3	2	8	1

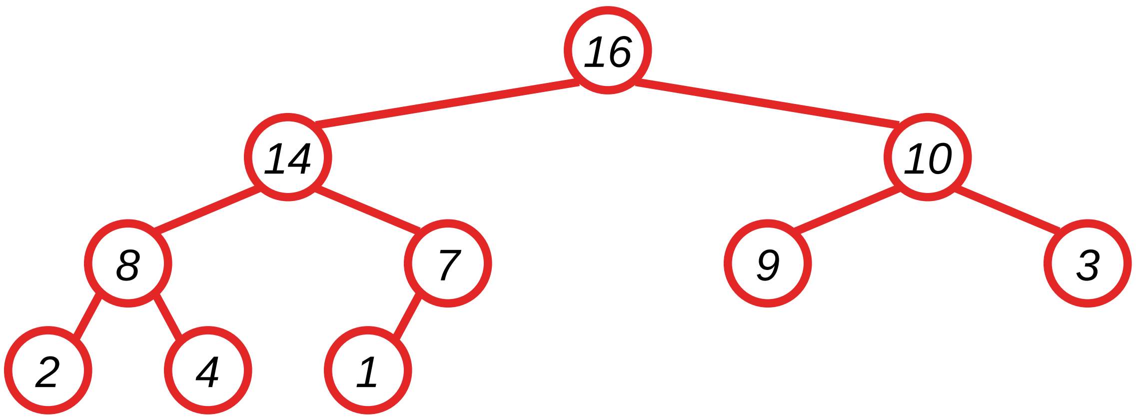
Heapify() Example



A =

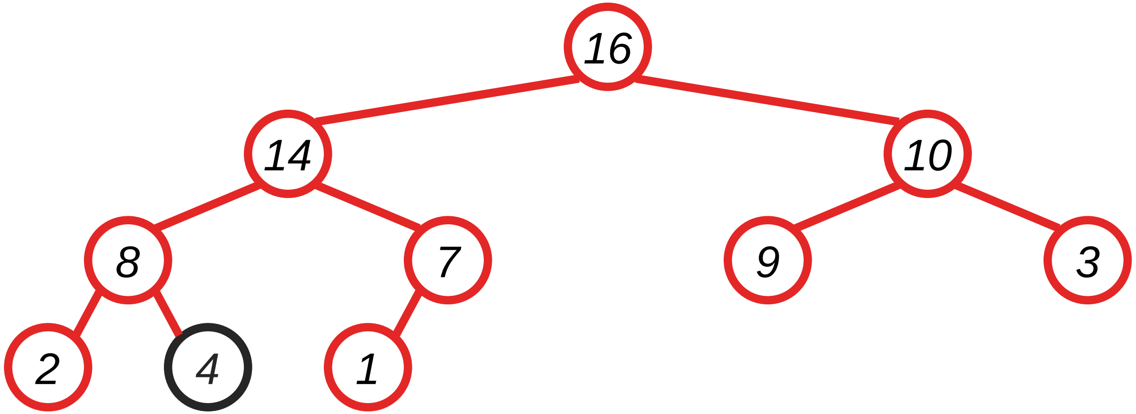
16	14	10	4	7	9	3	2	8	1

Heapify() Example



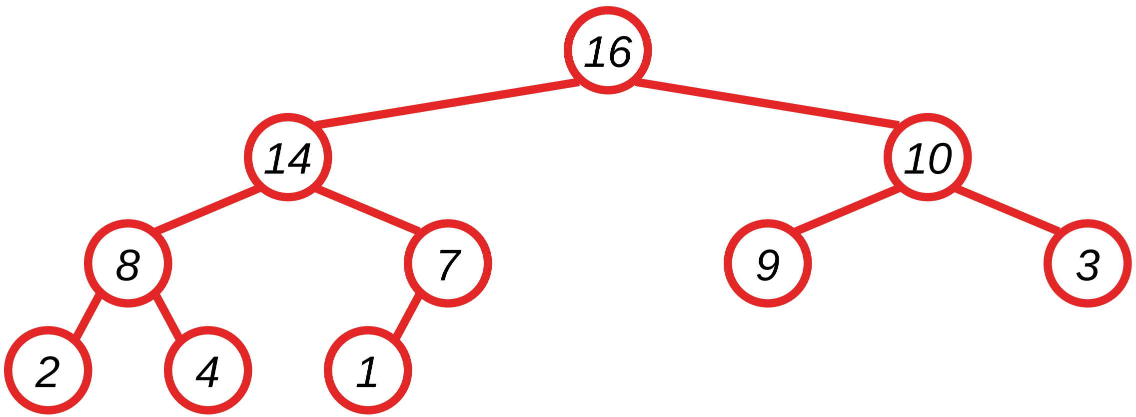
A =	16	14	10	8	7	9	3	2	4	1

Heapify() Example



A =	16	14	10	8	7	9	3	2	4	1

Heapify() Example



A =

16	14	10	8	7	9	3	2	4	1

Heap Operations: BuildHeap()

- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - Fact: for array of length n, all elements in the range $A[\lfloor n/2 \rfloor + 1 \ldots n]$ are heaps
  - So
    - Walk backwards through the array from $n/2$ to 1, calling **`Heapify()`** on each node.
    - Order of processing guarantees that the children of node i are heaps when i is processed

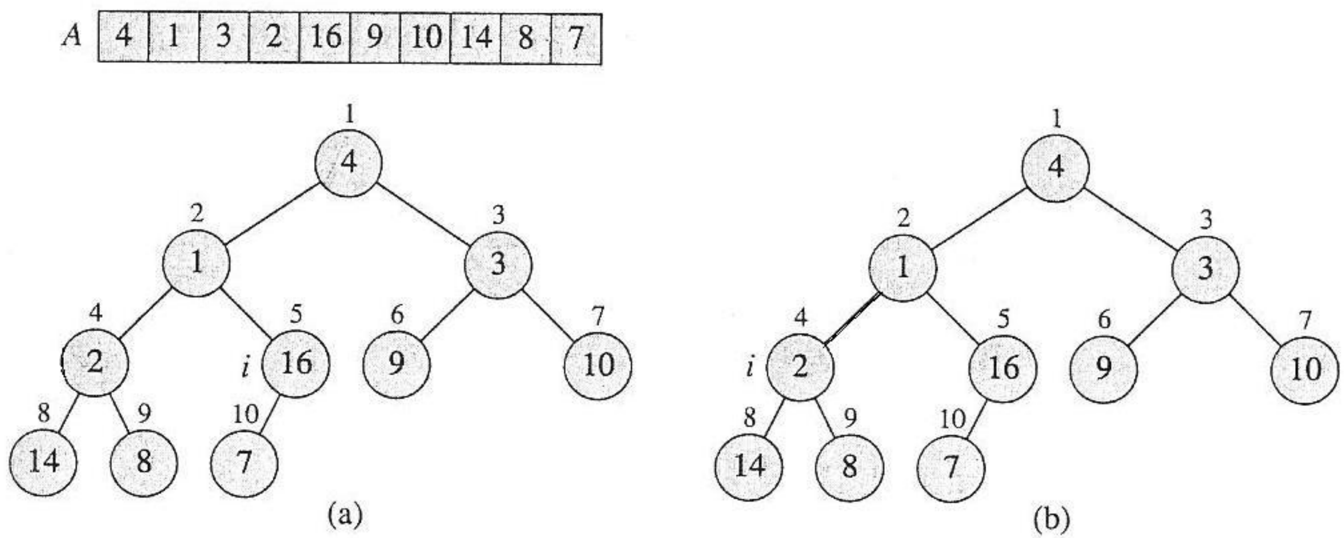
Heap Operations: BuildHeap()

Converts an unorganized array A into a max-heap.

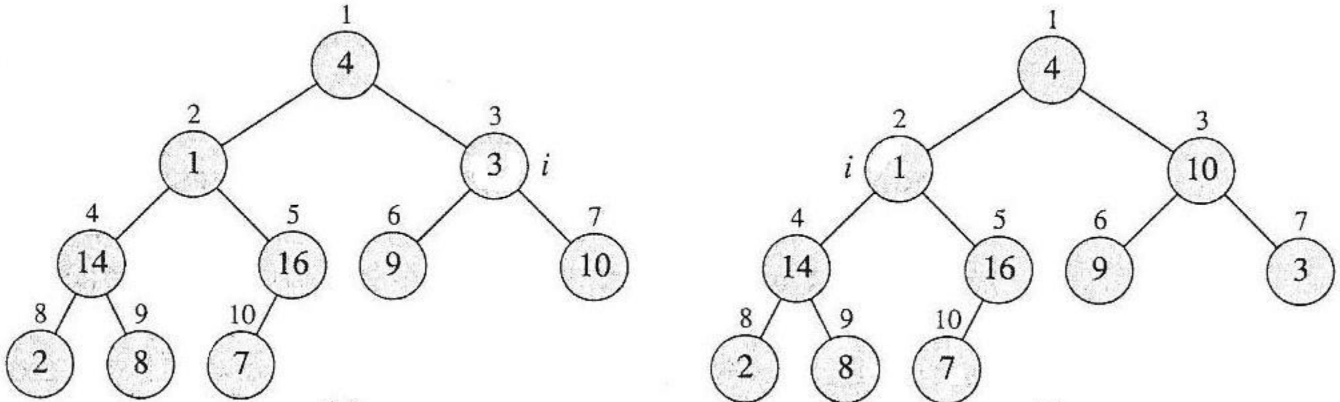
```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3       do MAX-HEAPIFY(A, i)
```

BuildHeap() Example

- Work through example
  A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}

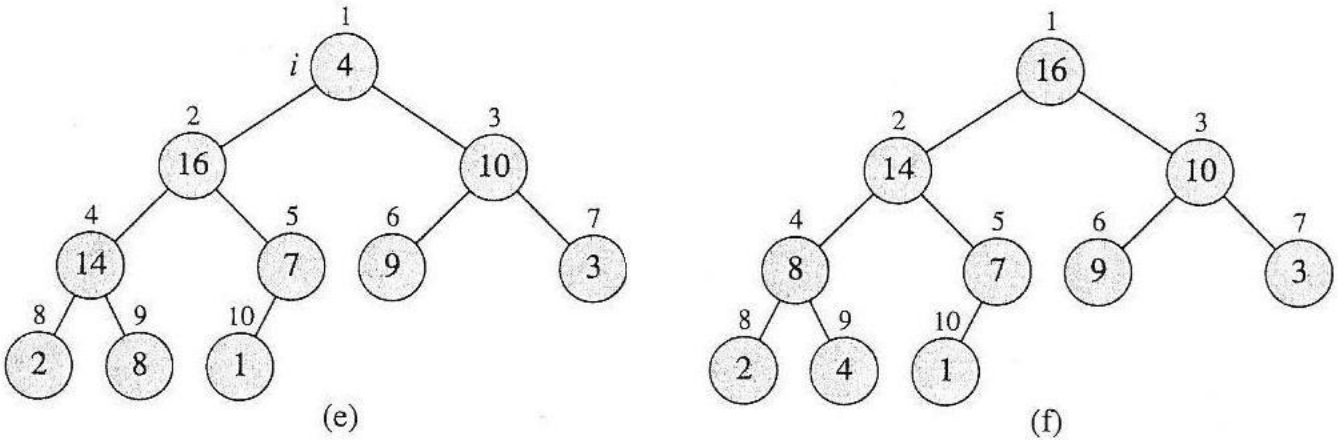


BuildHeap() Example



(c) (d)

BuildHeap() Example



Heapsort

- Given **`BuildHeap()`**, a sorting algorithm can easily be constructed:
  - Maximum element is at A[1]
  - Discard by swapping with element at A[*n*]
    - Decrement heap_size[A]
    - A[*n*] now contains correct value
  - Restore heap property at A[1] by calling **`Heapify()`**
  - Repeat, always swapping A[1] for A[heap_size(A)]

Heapsort

```
Heapsort(A)
{
    BuildHeap(A);
    for (i = length(A) downto 2)
    {
        Swap(A[1], A[i]);
        heap_size(A) = heap_size(A) – 1;
        Heapify(A, 1);
    }
}
```

Heap sort implementation in Java

```java
//Java program for implementation of Heap Sort
public class HeapSort
{
    public void sort(int arr[])
    {
        int n = arr.length;

        // Build heap (rearrange array)
        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);

        // One by one extract an element from heap
        for (int i=n-1; i>=0; i--)
        {
            // Move current root to end
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // call max heapify on the reduced heap
            heapify(arr, i, 0);
        }
    }
```

Heap sort implementation in Java

```java
// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2*i + 1; // left = 2*i + 1
    int r = 2*i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i)
    {
        int swap = arr[i];
        arr[i] = arr[largest];
        arr[largest] = swap;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

/* A utility function to print array of size n */
static void printArray(int arr[])
{
    int n = arr.length;
    for (int i=0; i<n; ++i)
        System.out.print(arr[i]+" ");
    System.out.println();
}
```

Heap sort implementation in Java

```java
    // Driver program
    public static void main(String args[])
    {
        int arr[] = {12, 11, 13, 5, 6, 7};
        int n = arr.length;

        HeapSort ob = new HeapSort();
        ob.sort(arr);

        System.out.println("Sorted array is");
        printArray(arr);
    }
}
```

Output:

```
Sorted array is
5 6 7 11 12 13
```

Thank You

Fahim Anzum

Email: fahim.anzum@ucalgary.ca