# Laboratory 11: Sorting Algorithms part -1

CSC205A Data structures and Algorithms Laboratory B. Tech. 2015

**Vaishali R Kulkarni**

Department of Computer Science and Engineering
Faculty of Engineering and Technology
M. S. Ramaiah University of Applied Sciences
Email: vaishali.cs.et@msruas.ac.in
Tel: +91-80-4906-5555 (2212) WWW: www.msruas.ac.in

# Introduction and Purpose of Experiment

- Sorting provide us with means of organising information to facilitate the retrieval of specific data.
- Searching methods are designed to take advantage of the organisation of information.
- By solving these problems students will be able to use sorting algorithms to sort a randomly ordered set of numbers, and search for key element.

# Aim and objectives

Aim:

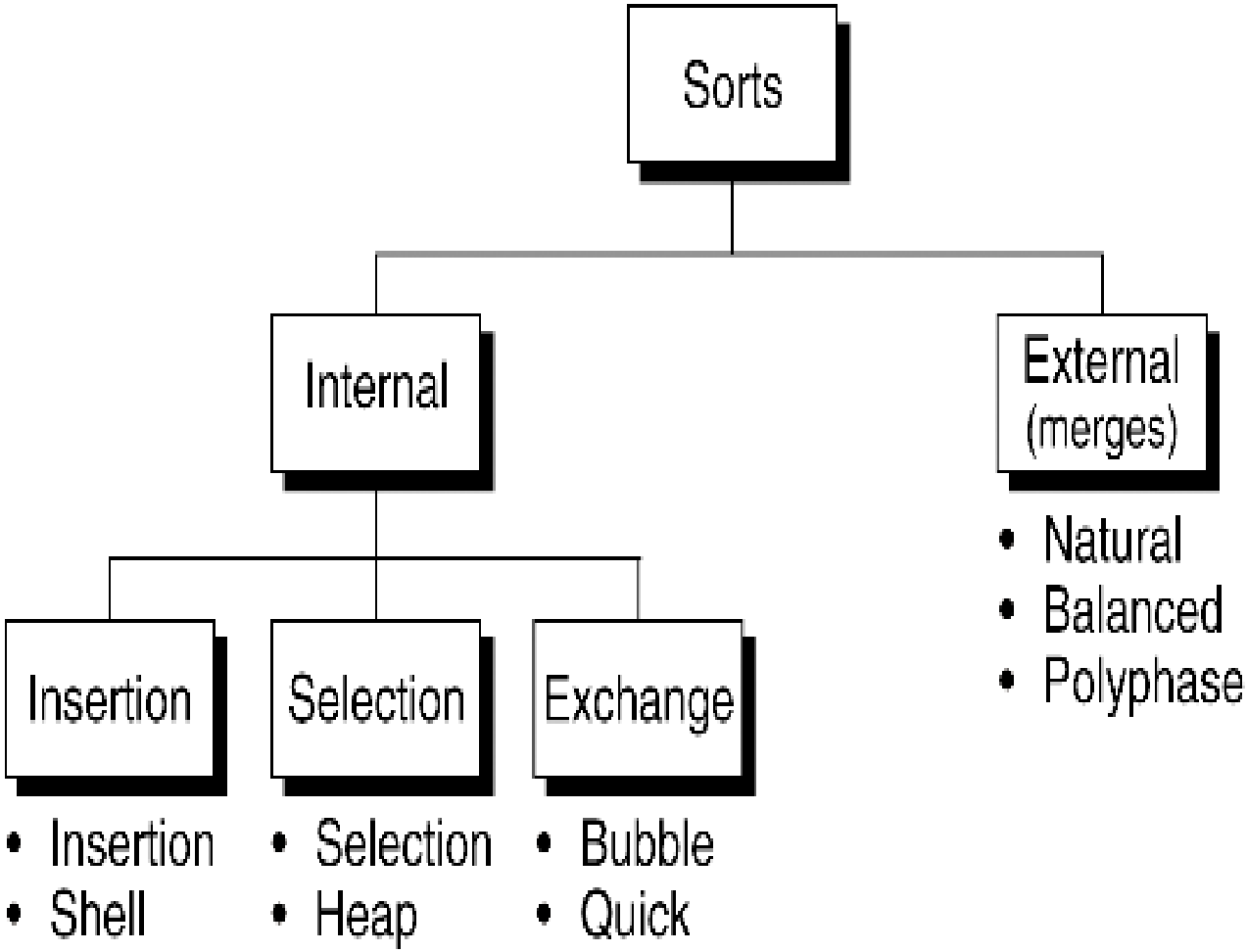
- To design and develop C programs to sort the given data using Insertion, bubble Heap sort, different sorting techniques

Objectives:

At the end of this lab, the student will be able to

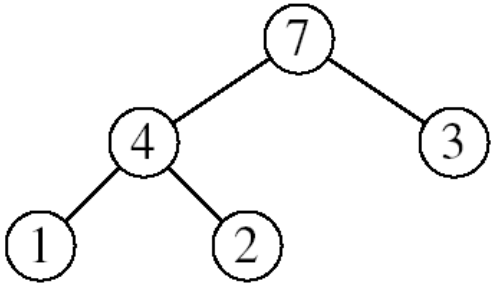
- Create C programs using sorting algorithms such as **heap sort**
- Create C programs using sorting algorithms such as **shell sort**
- Analyse the efficiency of implemented sort algorithms

# Sort Classifications

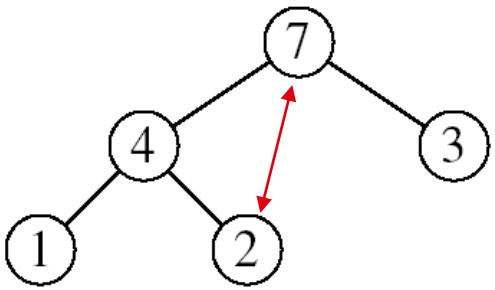


# Heapsort

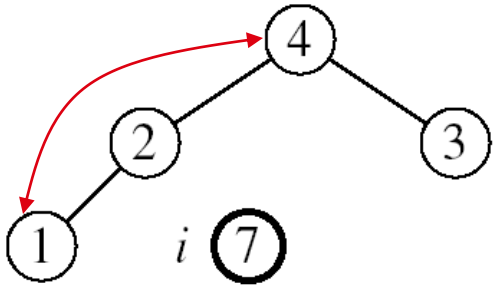
- Goal:
  - Sort an array using heap representations
- Idea:
  - Build a **max-heap** from the array
  - Swap the root (the maximum element) with the last element in the array
  - “Discard” this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains



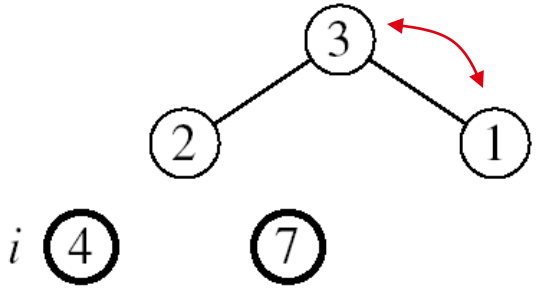
# Example: A=[7, 4, 3, 1, 2]



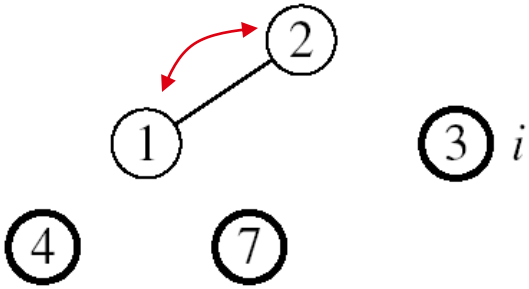

MAX-HEAPIFY(A, 1, 4)

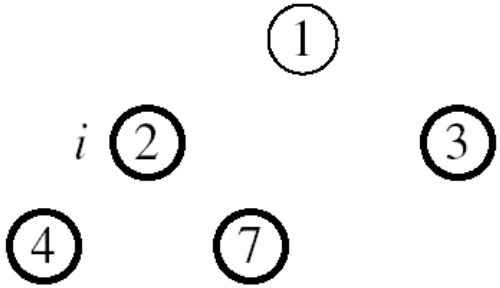



MAX-HEAPIFY(A, 1, 3)




MAX-HEAPIFY(A, 1, 2)




| A | 1 | 2 | 3 | 4 | 7 |
|---|---|---|---|---|---|

MAX-HEAPIFY(A, 1, 1)

# *Alg:* HEAPSORT(*A*)

1. BUILD-MAX-HEAP(*A*) — $O(n)$
2. **for** i ← length[*A*] **downto** 2
3. **do** exchange *A*[1] ↔ *A*[i]
4. MAX-HEAPIFY(*A*, 1, i - 1) — $O(\lg n)$

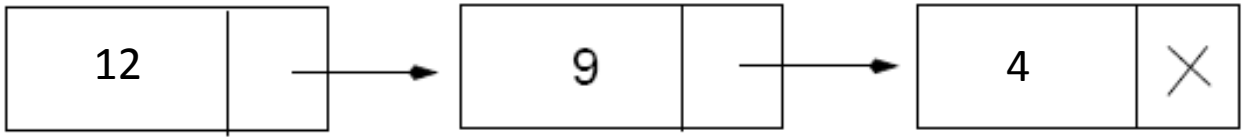
n-1 times

- Running time: $O(n\lg n)$ --- Can be shown to be $\Theta(n\lg n)$

# Priority Queues

**Properties**

- Each element is associated with a value (priority)
- The key with the highest (or lowest) priority is extracted first



# Operations on Priority Queues

- Max-priority queues support the following operations:
  - INSERT($S$, $x$): inserts element $x$ into set $S$
  - EXTRACT-MAX($S$): removes and returns element of $S$ with largest key
  - MAXIMUM($S$): returns element of $S$ with largest key
  - INCREASE-KEY($S$, $x$, $k$): increases value of element $x$'s key to $k$ (Assume $k \geq x$'s current key value)

# HEAP-MAXIMUM

Goal:

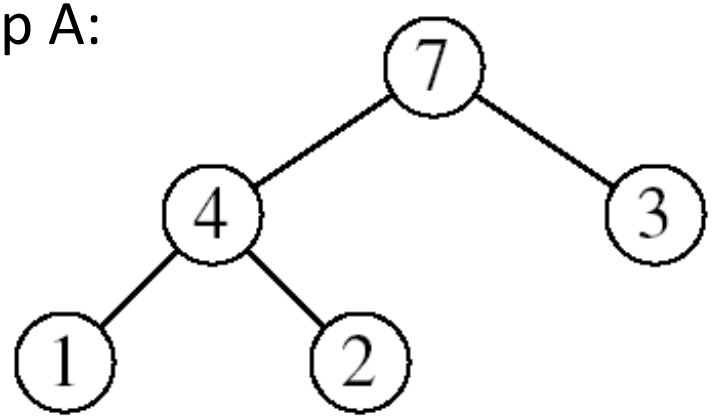
– Return the largest element of the heap

Running time: $O(1)$

*Alg:* HEAP-MAXIMUM($A$)

1. **return** $A[1]$

Heap A:




Heap-Maximum(A) returns 7

# HEAP-EXTRACT-MAX

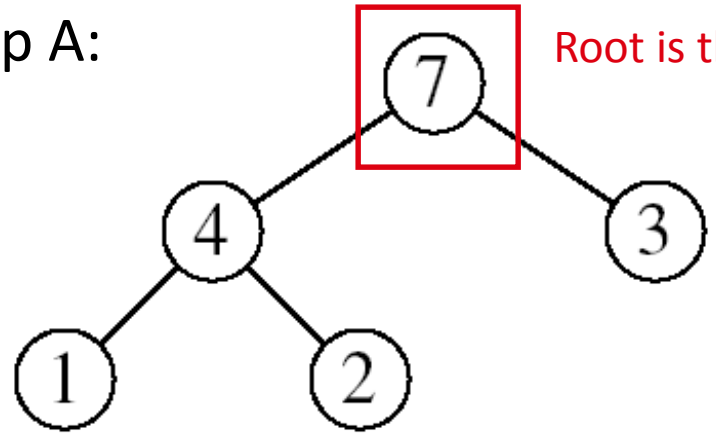
Goal:

- Extract the largest element of the heap (i.e., return the max value and also remove that element from the heap

Idea:

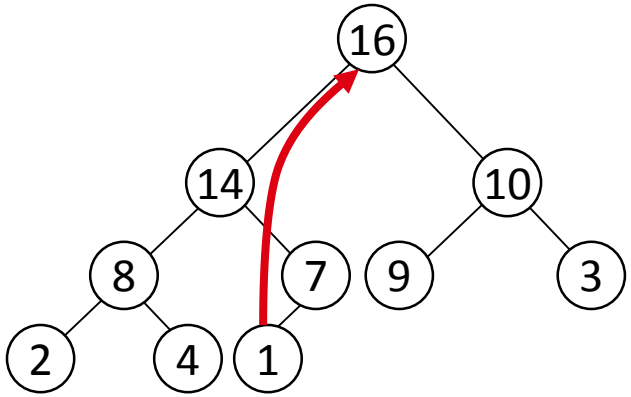
- Exchange the root element with the last
- Decrease the size of the heap by 1 element
- Call MAX-HEAPIFY on the new root, on a heap of size n-1

Heap A:

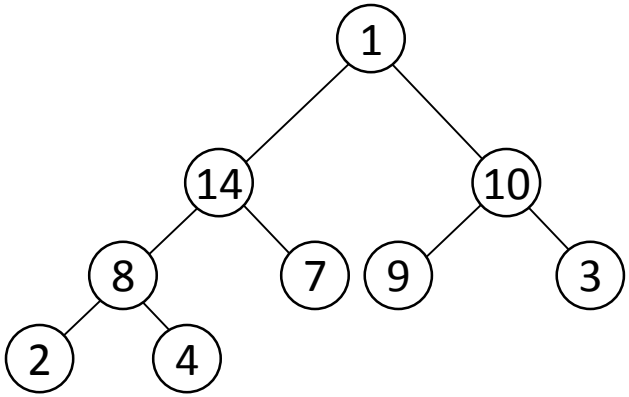



Root is the largest element

# Example: HEAP-EXTRACT-MAX

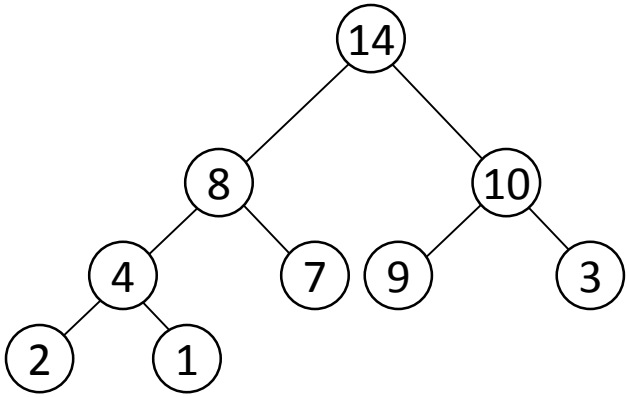



max = 16




Heap size decreased with 1

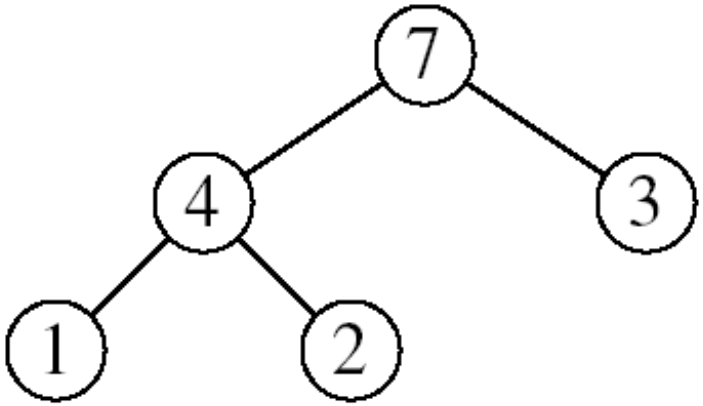
Call MAX-HEAPIFY(*A*, 1, *n*-1)



# HEAP-EXTRACT-MAX

*Alg:* HEAP-EXTRACT-MAX($A$, n)

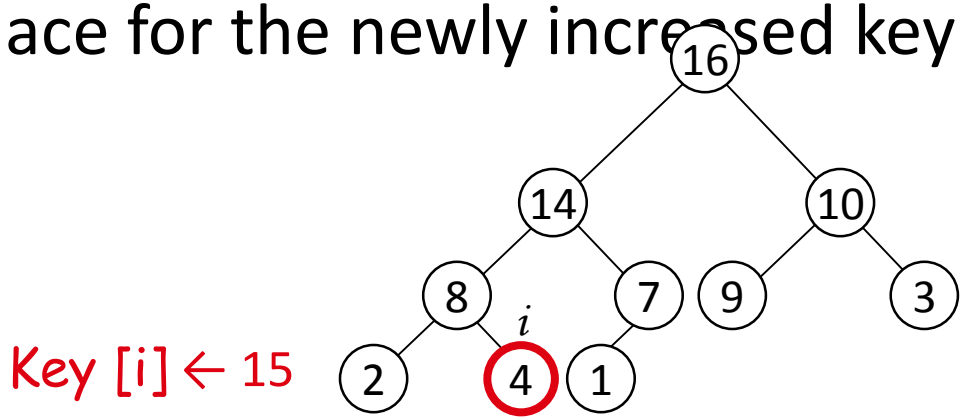
1. **if** n < 1
2. **then error** "heap underflow"
3. max ← $A[1]$
4. $A[1] \leftarrow A[n]$
5. MAX-HEAPIFY($A$, 1, n-1) ▷ remakes heap
6. **return** max



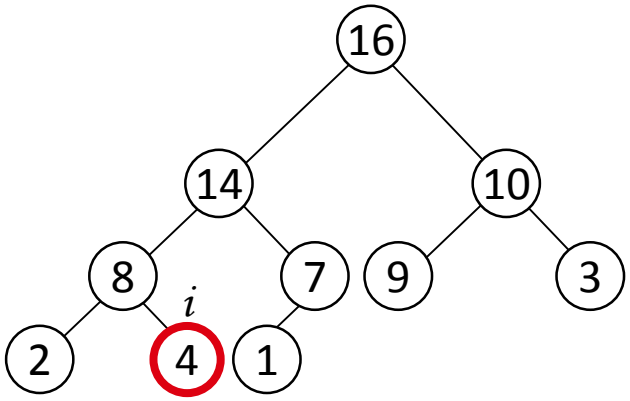

Running time: $O(\lg n)$

# HEAP-INCREASE-KEY

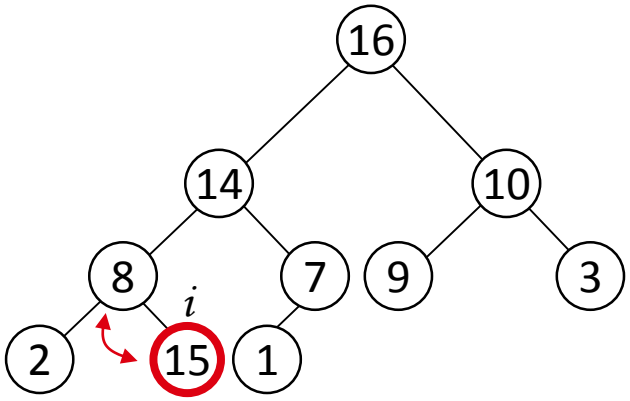
- Goal:
  - Increases the key of an element i in the heap
- Idea:
  - Increment the key of $A[i]$ to its new value
  - If the max-heap property does not hold anymore: traverse a path toward the root to find the proper place for the newly increased key

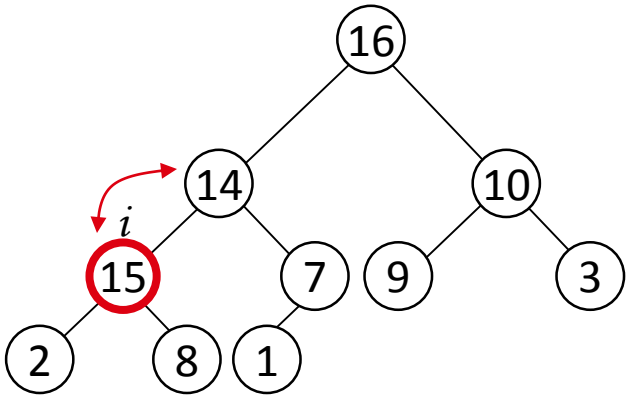


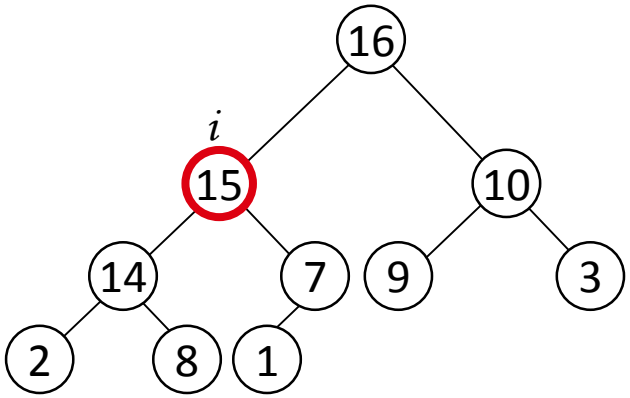
# Example: HEAP-INCREASE-KEY




$Key[i] \leftarrow 15$



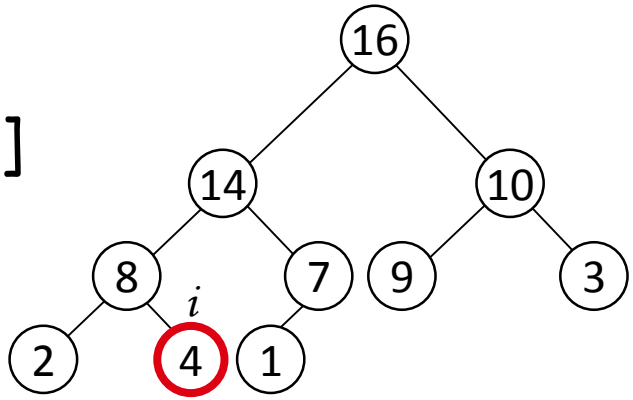



# HEAP-INCREASE-KEY

*Alg:* HEAP-INCREASE-KEY(A, i, key)

```
if key < A[i]
   then error "new key is smaller than current key"
A[i] ← key
while i > 1 and A[PARENT(i)] < A[i]
    do exchange A[i] ↔ A[PARENT(i)]
        i ← PARENT(i)
```

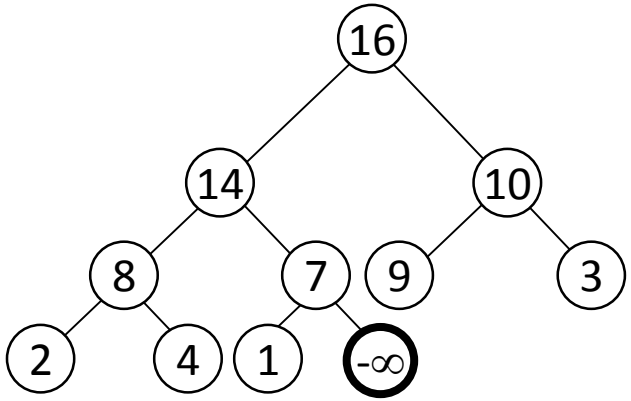


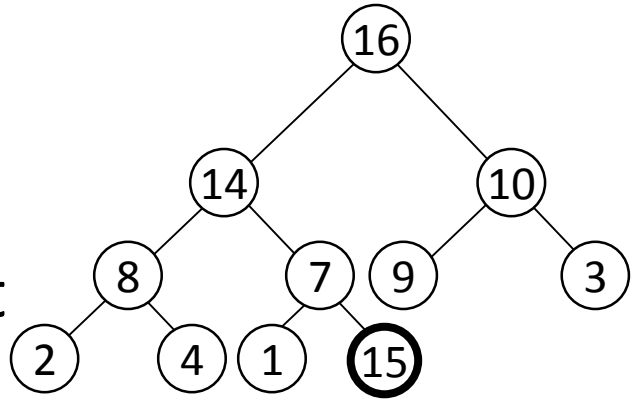

Key [i] ← 15

- Running time: $O(\lg n)$

# MAX-HEAP-INSERT

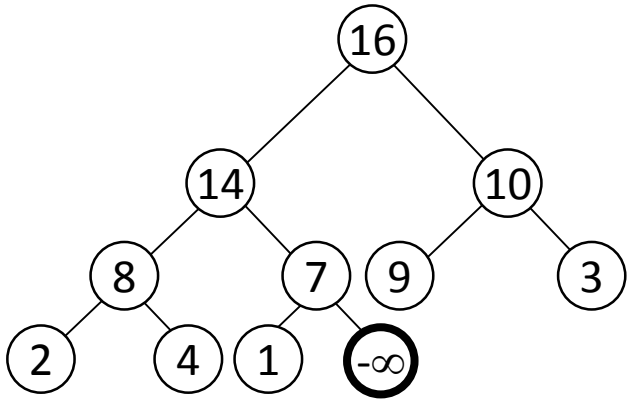
- Goal:
  - Inserts a new element into a max-heap
- Idea:
  - Expand the max-heap with a new element whose key is -∞
  - Calls HEAP-INCREASE-KEY to set the key of the new node to its correct value and maintain the max-heap property



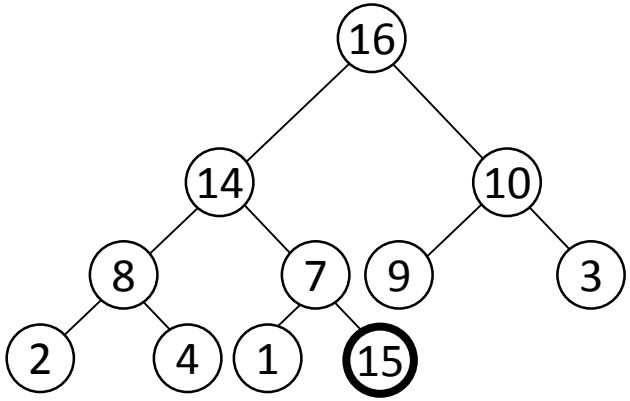



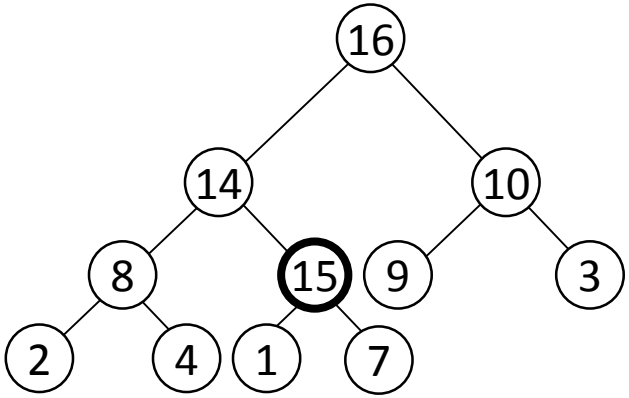
# Example: MAX-HEAP-INSERT

Insert value 15:
- Start by inserting -∞

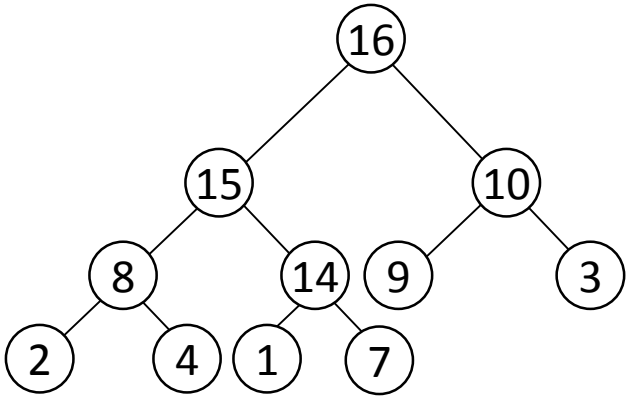



Increase the key to 15
Call HEAP-INCREASE-KEY on A[11] = 15



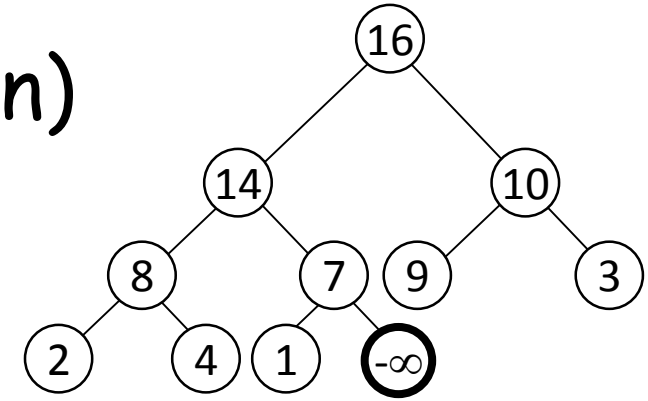

The restored heap containing the newly added element



# MAX-HEAP-INSERT

*Alg:* MAX-HEAP-INSERT(*A*, key, n)

1. heap-size[*A*] ← n + 1
2. *A*[n + 1] ← -∞
3. HEAP-INCREASE-KEY(*A*, n + 1, key)



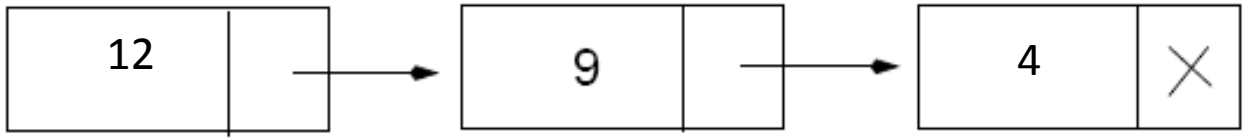

Running time: *O*(lg*n*)

# Summary

- We can perform the following operations on heaps:
  - MAX-HEAPIFY $O(\lg n)$
  - BUILD-MAX-HEAP $O(n)$
  - HEAP-SORT $O(n \lg n)$
  - MAX-HEAP-INSERT $O(\lg n)$
  - HEAP-EXTRACT-MAX $O(\lg n)$
  - HEAP-INCREASE-KEY $O(\lg n)$
  - HEAP-MAXIMUM $O(1)$

Average $O(\lg n)$

# Priority Queue Using Linked List




**Remove a key: O(1)**

**Insert a key: O(n)**

Increase key: O(n)

Extract max key: O(1)

Average: O(n)

# Experimental Procedure

- Analyse the problem statement
- Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
- Implement the algorithm in C language
- Compile the C program
- Design test cases and test the implemented program
- Document the Results
- Analyse and discuss the outcomes of your experiment

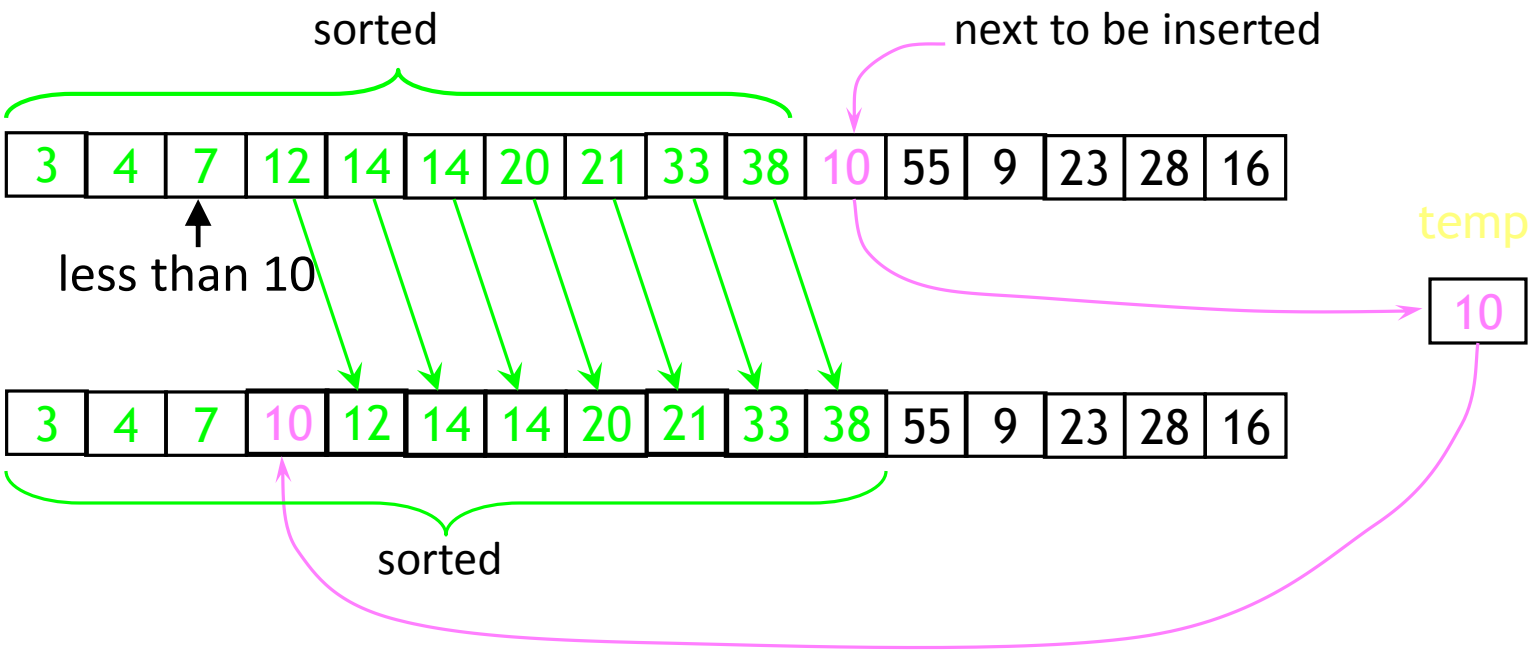
# Review: Insertion sort

- The outer loop of insertion sort is:

  for (outer = 1; outer < a.length; outer++) {...}

- The invariant is that all the elements to the left of outer are sorted with respect to one another
  - For all i < outer, j < outer, if i < j then a[i] <= a[j]
  - This does *not* mean they are all in their final correct place; the remaining array elements may need to be inserted
  - When we increase outer, a[outer-1] becomes to its left; we must keep the invariant true by inserting a[outer-1] into its proper place
  - This means:
    - Finding the element's proper place
    - Making room for the inserted element (by shifting over other elements)
    - Inserting the element

# One step of insertion sort




- This *one step* takes O(n) time
- We must do it n times; hence, insertion sort is $O(n^2)$

# The idea of shellsort

- With insertion sort, each time we insert an element, other elements get nudged *one step* closer to where they ought to be
- What if we could move elements a *much longer distance* each time?
- We could move each element:
  - A long distance
  - A somewhat shorter distance
  - A shorter distance still
- This approach is what makes shellsort so much faster than insertion sort

# Shellsort

- Founded by Donald Shell and named the sorting algorithm after himself in 1959.
- 1st algorithm to break the quadratic time barrier but few years later, a sub quadratic time bound was proven
- Shellsort works by comparing elements that are distant rather than adjacent elements in an array or list where adjacent elements are compared.

# Shellsort

- Shellsort uses a sequence $h_1, h_2, \ldots, h_t$ called the ***increment sequence***. Any increment sequence is fine as long as $h_1 = 1$ and some other choices are better than others.

# Shellsort

- Shellsort makes multiple passes through a list and sorts a number of equally sized sets using the insertion sort.

# Shellsort

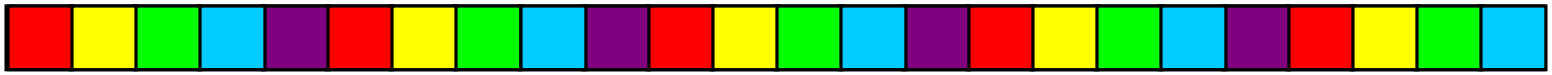
- Shellsort improves on the efficiency of insertion sort by *quickly* shifting values to their destination.

# Shellsort

- Shellsort is also known as ***diminishing increment sort***.
- The distance between comparisons decreases as the sorting algorithm runs until the last phase in which adjacent elements are compared

# Sorting nonconsecutive subarrays

- Here is an array to be sorted (numbers aren't important)



- Consider just the red locations
- Suppose we do an insertion sort on *just these numbers,* as if they were the only ones in the array?
- Now consider just the yellow locations
- We do an insertion sort on just these numbers
- Now do the same for each additional group of numbers
- The resultant array is sorted *within groups,* but not overall

# C Code: Shellsort

```
void sort(Item a[], int sequence[], int start, int stop)
   {
   int step, i;

   for (int step = 0; sequence[step] >= 1; step++)
      {
      int inc = sequence[step];

      for (i = start + inc; i <= stop; i++)
         {
         int j = i;
         Item val = a[i];

         while ((j >= start + inc) && val < a[j - inc])
            {
            a[j] = a[j - inc];
            j -= inc;
            }

         a[j] = val;
         }
      }
   }
```

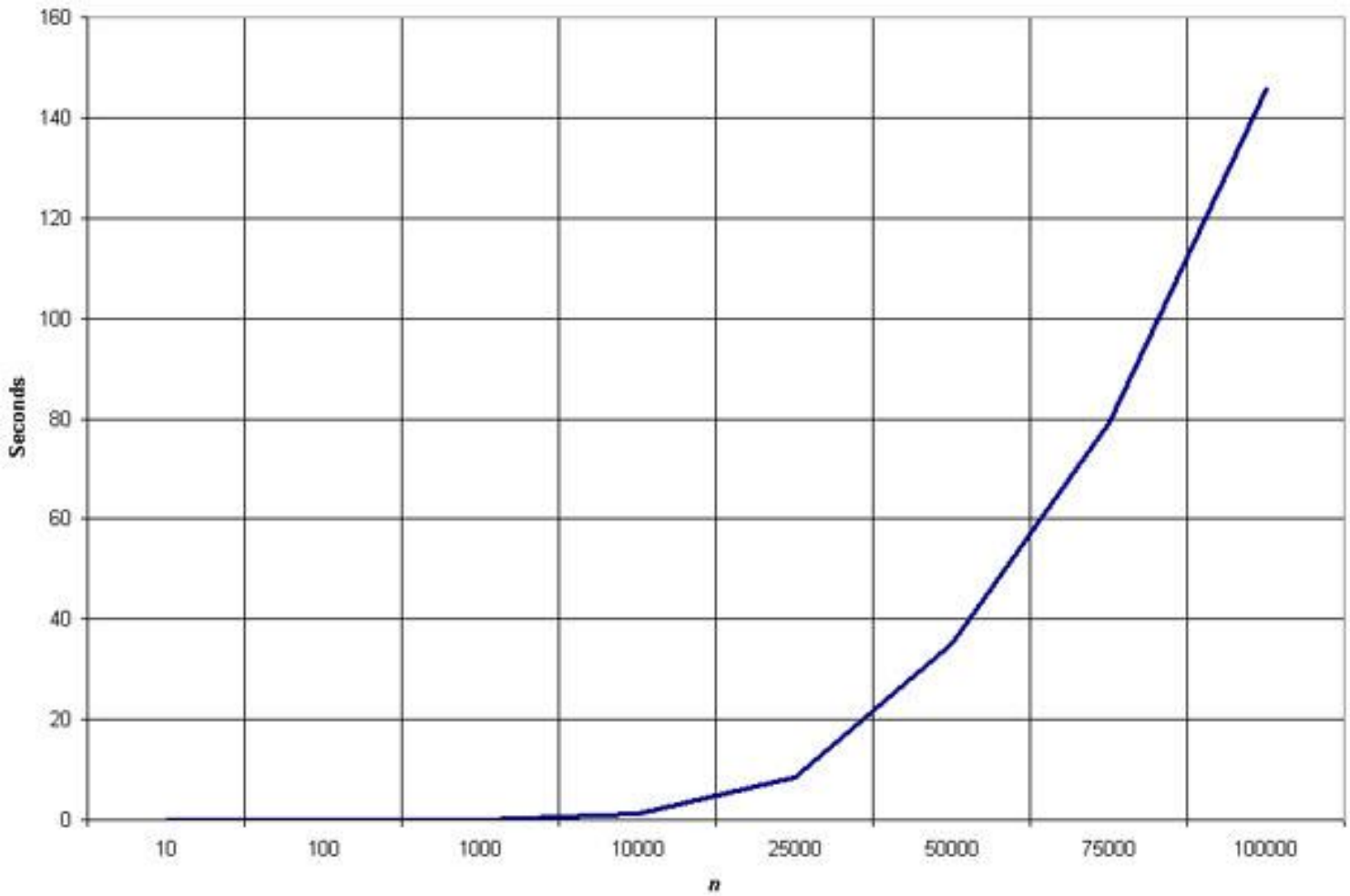
# C Code: Using a Shell Sort

```
#include "stdlib.h"
#include "stdio.h"

#define Item int

void sort(Item a[], int sequence[], int start, int stop);

int main(int argc, char * argv[])
   {
   printf("This program uses shell sort to sort a random array\n\n");
   printf("  Parameters: [array-size]\n\n");

   int size = 100;
   if (argc > 1) size = atoi(argv[1]);

   int sequence[] = { 364, 121, 40, 13, 4,  1, 0};
   int * array = (int *) malloc(sizeof(int) * size);

   srand(123456);
   printf("Generating %d random elements …\n", size);
   for (int i = 0; i < size; i++)
       array[i] = rand();

   printf("Sorting elements …\n", size);
   sort(array, sequence, 0, size - 1);

   printf("The sorted array is …\n");
   for (int i = 0; i < size; i++)
      printf("%d ", array[i]);
   printf("\n");
   free(array);
   }
```

# Shellsort

- After each phase and some increment $h_k$, for every *i*, we have a[ *i* ] ≤ a [ *i* + $h_k$ ] all elements spaced $h_k$ apart are sorted.
- The file is said to be $h_k$ – sorted.

# Empirical Analysis of Shellsort




Source: http://linux.wku.edu/~lamonml/algor/sort/shell.html

# Shellsort Best Case

- Best Case: The best case in the shell sort is when the array is already sorted in the right order. The number of comparisons is less.

# Shellsort Worst Case

- The running time of Shellsort depends on the choice of increment sequence.
- The problem with Shell's increments is that pairs of increments are not necessarily relatively prime and smaller increments can have little effect.

# Exercise

- Given a randomly ordered set of n numbers, design and develop an algorithm to sort them into non-descending order using heap sort and shell sort compare their efficiency. Tabulate the output for various inputs and verify against expected values. Analyse the efficiency of both the algorithms. Describe your learning along with the limitations of both, if any. Suggest how these can be overcome.

# Key factors for discussion and analysis

- Implement heap sort and sort random integers
- Implement shell sort and sort random integers
- Analyse the performance of both
- List out the advantages and disadvantages of both

# Results and Presentations

- Calculations/Computations/Algorithms

  The calculations/computations/algorithms involved in each program has to be presented

- Presentation of Results

  The results for all the valid and invalid cases have to be presented

- Analysis and Discussions

  how the data is manipulated or transformed, what are the key operations involved. Errors encounters and how they are resolved.

- Conclusions

  Summary

# Comments

- Limitations of Experiments

  Outline the loopholes in the program, data structures or solution approach.

- Limitations of Results

  Present the test cases; justify if the program is tested correctly considering all the outcomes. Mention what is not tested, if any.

- Learning happened

  What is the overall learning happened

- Conclusions

  Summary

# References


- Gilberg, R. F., and Forouzan, B. A. (2007): A Pseudocode Approach With C, 2nd edn. Cengage Learning