Introduction to Computers and Programming

Prof. I. K. Lundqvist

Lecture 11
April 12 2004

Binary Search

- Can be performed on
  - Sorted arrays
  - Full and balanced BSTs
- Compares and cuts half the work
  - We cut work in ½ each time
  - How many times can we cut in half?

Binary search is **O(Log N)**

Binary Search

11	17	19	42	43	47	53	67

11 found
3 comparisons
3 = Log (8)

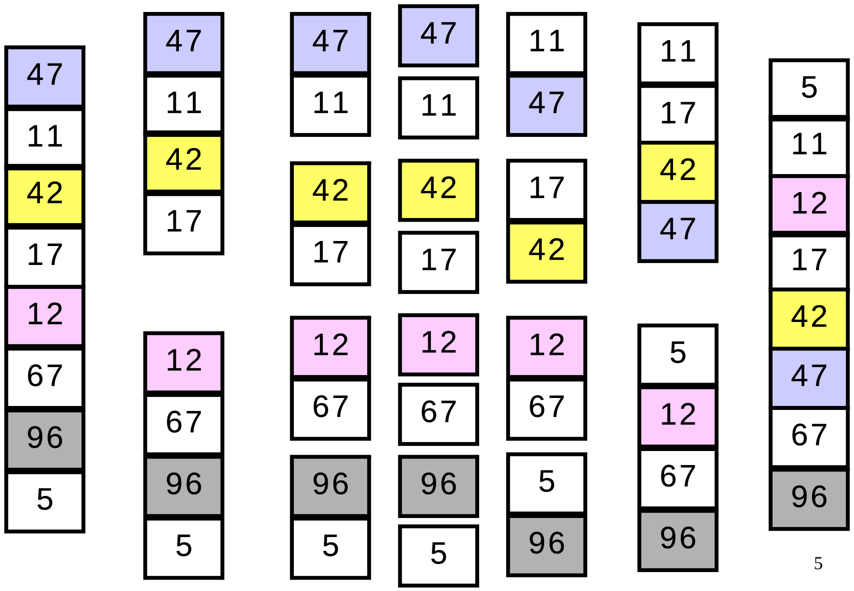
55 not found
4 comparisons
4 = Log (8) +1


```
The Binary Search Algorithm
Input:   Array to search, element to search for
Output:  Index if element found, -1 otherwise

Statement                                            Work   T(n)

set Return_Index to -1;                              -- c1
loop                                                 -- c2 (log n +1)
   set Current_Index to ⌊(UB + LB) /2⌋               -- c3 (log n +1)
   if the LB > UB                                    -- c4 (log n +1)
      exit;                                          -- c5
   if Input_Array(Current_Index) = element           -- c6 log n
      Return_Index := Current_Index                  -- c7
      exit;                                          -- c8
   if Input_Array(Current_Index) < element           -- c9  log n
      LB := Current_Index +1                         -- c10 log n
   else                                              -- c11 log n
      UB := Current_Index – 1                        -- c12 log n
return Return_Index                                  -- c13
```

T(n) = (c1+c2+c3+c4+c5+c7+c8+c13) + (c2+c3+c4+c6+c9+c10+c11+c12) log n
= c' + c''log(n)

= O(log (n))

Merge Sort



Merge Sort Analysis Analysis

Statement	Work
MergeSort(A, left, right)	T(n)
if (left < right)	O(1)
mid := (left + right) / 2;	O(1)
MergeSort(A, left, mid);	T(n/2)
MergeSort(A, mid+1, right);	T(n/2)
Merge(A, left, mid, right);	O(n)

$$T(n) = \begin{cases} O(1) & \text{when } n = 1, \\ 2T(n/2) + O(n) & \text{when } n > 1 \end{cases}$$

Recurrence Equation

Solving Recurrences: Iteration

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- T(n) =
  $aT(n/b) + cn$
  $a(aT(n/b/b) + cn/b) + cn$
  $a^2T(n/b^2) + cna/b + cn$
  $a^2T(n/b^2) + cn(a/b + 1)$
  $a^2(aT(n/b^2/b) + cn/b^2) + cn(a/b + 1)$
  $a^3T(n/b^3) + cn(a^2/b^2) + cn(a/b + 1)$
  $a^3T(n/b^3) + cn(a^2/b^2 + a/b + 1)$
  …
  $a^kT(n/b^k) + cn(a^{k-1}/b^{k-1} + a^{k-2}/b^{k-2} + \ldots + a^2/b^2 + a/b + 1)$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So we have
  - $T(n) = a^k T(n/b^k) + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$
- For $n = b^k$
  - $T(n) = a^k T(1) + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$
    $= a^k c + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$
    $= ca^k + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$
    $= cna^k/b^k + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$
    $= cn\ (a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if a = b?
  - $T(n) = cn(k + 1)$
    $= cn(\log_b n + 1)$

$$= O(n \log n)$$

Heapsort: Best of two worlds

- Merge sort
  - Advantage: O(n log n)
- Insertion sort
  - Advantage: Can sort in place, and efficient for nearly sorted arrays

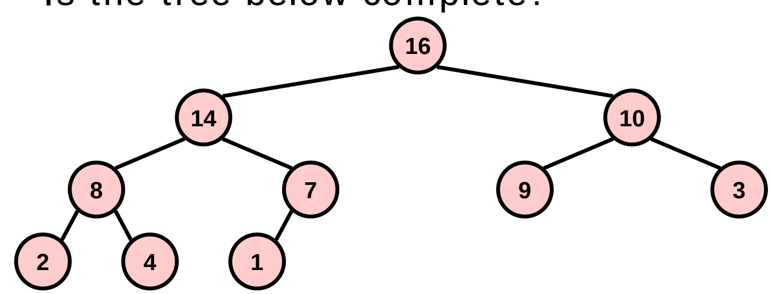
Heapsort

Combines the advantage of Merge and Insertion sort

Heaps

A **complete binary tree** is a full binary tree that has all leaves at the same depth

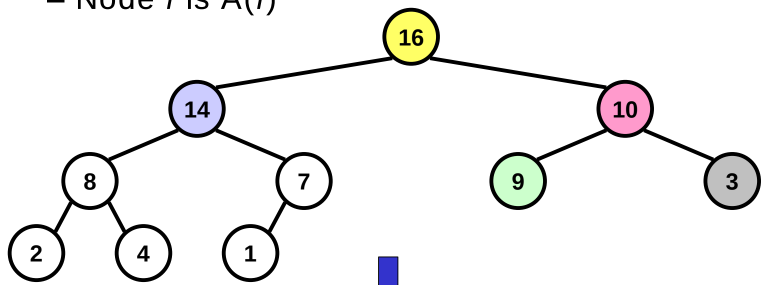
- Is the tree below complete?



Heaps are **nearly** *complete binary trees*

Binary Tree → Array

- Use Breadth-First-Search
  - The root node is A(1)
  - Node *i* is A(*i*)

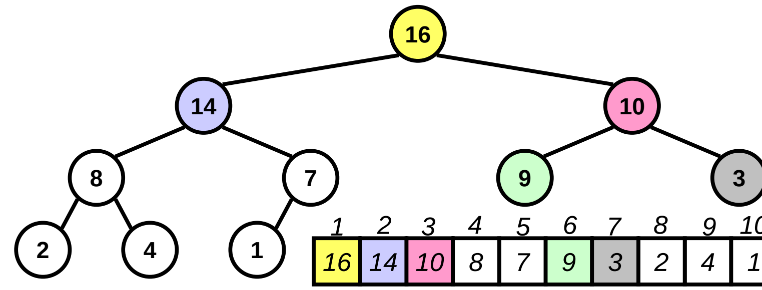


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Heap_Array

Manipulating Heaps

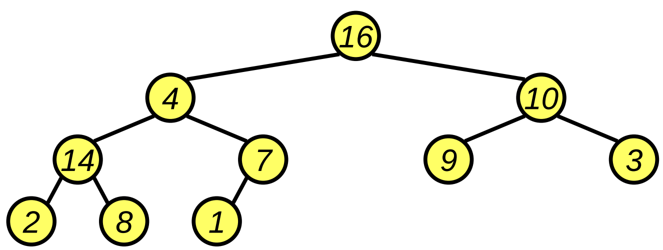
- Parent(i) – return ⌊i/2⌋
- Left(i) – return (2*i)
- Right(i) – return (2*i + 1)



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Heap property: A(Parent(i)) ≥ A(i)

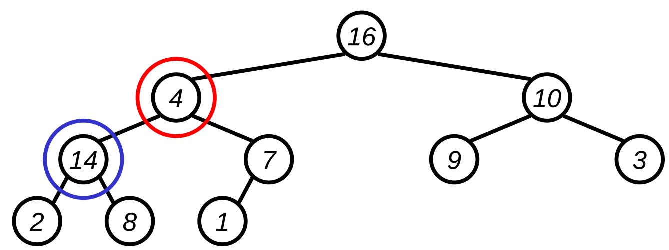
Is the tree a Heap?



Heap_Array =

16	4	10	14	7	9	3	2	8	1
1	2	3	4	5	6	7	8	9	10

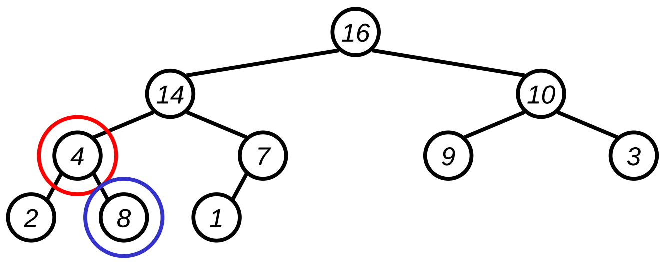
Is the tree a Heap?



Heap_Array =

16	4	10	14	7	9	3	2	8	1
1	2	3	4	5	6	7	8	9	10

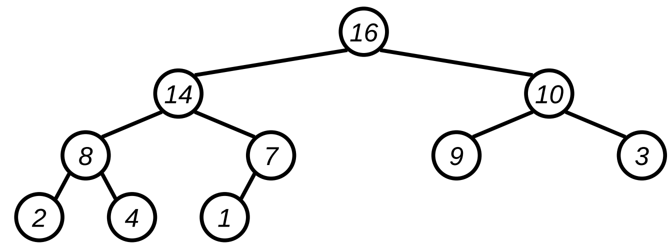
Is the tree a Heap?



Heap_Array =

16	14	10	4	7	9	3	2	8	1
1	2	3	4	5	6	7	8	9	10

Now it is a heap



Heap_Array =

16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

Heapify

Move an element in location *i* to satisfy heap property

```
Heapify (A, i)
  lchild := Left(i)
  rchild := Right(i)
  if (lchild <= heap_size and A[lchild] > A[i]) then
      largest := lchild
  else
      largest := i

  if (rchild <= heap_size and A[rchild] > A[largest]) then
      largest := rchild

  if (largest /= i) then
      Swap(A, i, largest)
      Heapify(A, largest)
```


Creating a heap

- Given an unsorted array

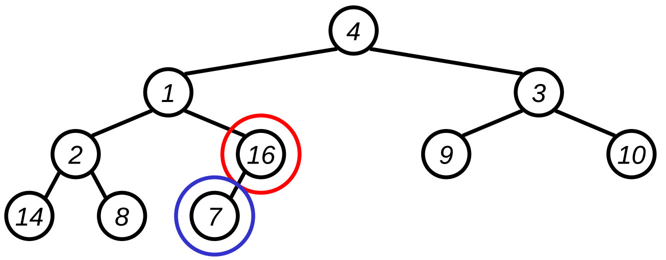
```
build_heap (A, Size)
   heap_size(A) := Size;
   for i in ⌊Size/2⌋  downto 1
        Heapify(A, i);
```

Build_Heap

Heapify(Heap_Array, 5) – Does nothing

Heap_Array =

1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

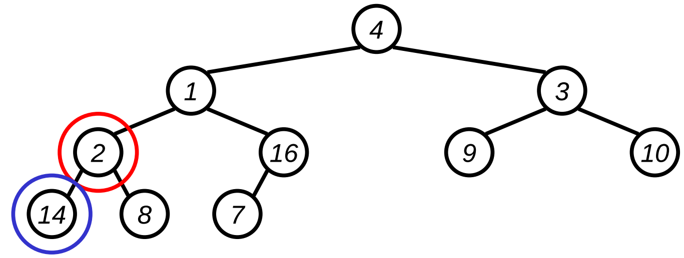


Build_Heap

Heapify(Heap_Array,4)

Heap_Array =

1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

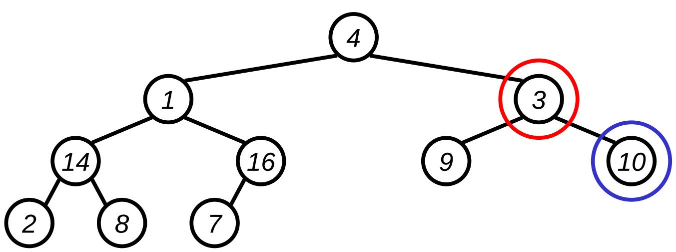


Build_Heap

Heapify(Heap_Array,3)

Heap_Array =

1	2	3	4	5	6	7	8	9	10
4	1	3	14	16	9	10	2	8	7

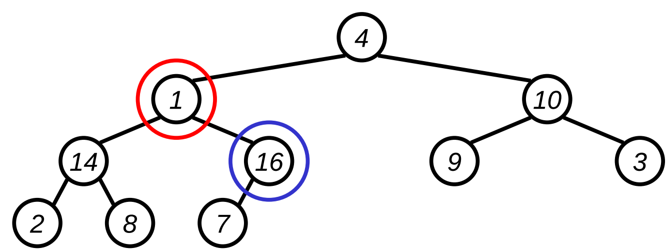


Build_Heap

Heapify(Heap_Array,2)

Heap_Array =

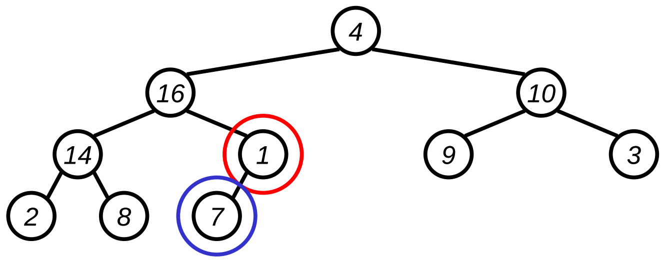
1	2	3	4	5	6	7	8	9	10
4	1	10	14	16	9	3	2	8	7



Build_Heap

Recursive Heapify(Heap_Array, 5)

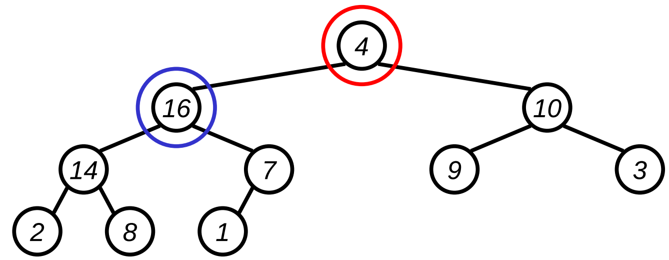
	1	2	3	4	5	6	7	8	9	10
Heap_Array =	4	16	10	14	1	9	3	2	8	7



Build_Heap

Heapify(Heap_Array, 1)

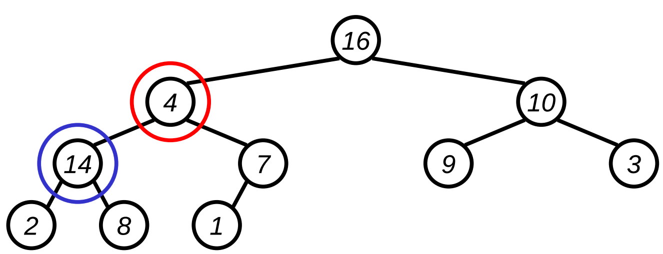
	1	2	3	4	5	6	7	8	9	10
Heap_Array =	4	16	10	14	7	9	3	2	8	1



Build_Heap

Recursive Heapify(Heap_Array, 2)

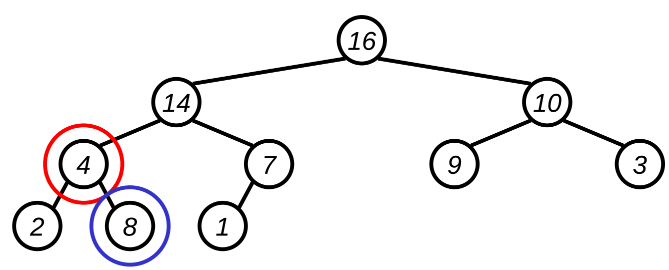
	1	2	3	4	5	6	7	8	9	10
Heap_Array =	16	4	10	14	7	9	3	2	8	1



Build_Heap

Recursive Heapify(Heap_Array, 4)

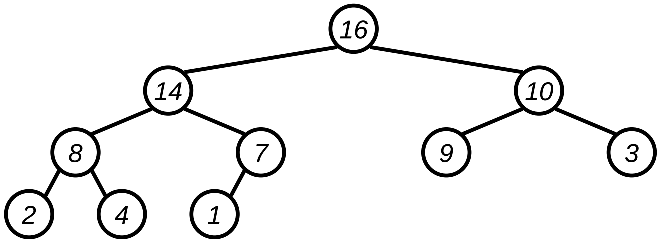
	1	2	3	4	5	6	7	8	9	10
Heap_Array =	16	14	10	4	7	9	3	2	8	1



Heap

Heap_Array =

1	2	3	4	5	6	7	8	9	10
16	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*1*



Heap Sort

```
BuildHeap(A);
for i in size downto 2
    Swap(A[1], A[i])
    heap_size:= heap_size -1;
    Heapify(A, 1);
```