STUDY OF SORTING ALGORITHMS

HARIKRISHNA MALAYAPPAN

KUMAR KRIS RAMANUJAM

THE UNIVERSITY OF TEXAS AT ARLINGTON

Sorting – Historic Introduction

- First intensively studied computer science problem.
- Developed during 1960s when large scale commercial data processing System was automated on a large scale
- Transfer of data between slow storage (tape or disk) and main memory was a major issue.
- Most files does not fit inside the memory
- Main Memory was limited (in 100 K bytes).
- Files to be sorted were in large magnitude.
- Algorithm to sort these files was a major focus among companies.

Motivation

- Practical use since often used
- Working with large sets of data in computers is facilitated when the data is sorted.
- Different perspective towards the same problem
  - e.g. How to improve the worst case time of the Quick Sort?
- Minimum amount of work specified by the algorithms to get the optimal solution to the problem.
- How much extra space does the algorithm use to optimize the memory usage
- If the amount of extra space is constant with respect to the input size, the algorithm is said to work IN PLACE.

Motivation contd.

- MOORE'S LAW: Processing Power Doubles every 18 months
  - memory capacity doubles every 18 months
  - problem size expands to fill memory
- Sedgewick's Corollary: Need Faster Sorts every 18 months!

  Sorts take longer to complete on new processors

  old: N lg N

  new: (2N lg 2N)/2 = N lg N + N
- Other compelling reasons to study sorting
  - cope with new languages and machines
  - rebuild obsolete libraries
  - address new applications
  - intellectual challenge of basic research

Overview

Simple Sorting Algorithms
e.g. insertion sort, bubble sort

ADV
→ Easy to implement

DisADV
→ too slow – $O(n^2)$

Hence, hardly used in practice

Fast Sorting Algorithms
e.g. quick sort, heap sort, shell sort

ADV
→ Fast – O(n logn)

DisADV
→ Difficult to implement

Used for large data sets

Sorting Algorithms – Important Aspects

- Which sorting algorithm shall I use ?
  - depends on application profile
- Criteria:
  - performance – applications prefer a fast sorting algorithm
  - worst case performance – critical for military or engineering applications
- Stability – important for applications working with arrays that are almost sorted
- Implementation simple <-> difficult (creation of additional data structures e.g. heapsort)
- Performance study
  - Comparing Elements
    - Number of comparisons (counting loops)
  - Moving or replacing elements
    - Number of Exchanges (counting loops)

Summary of Sorting Algorithms

Sorting Methods	Worst Case	Best Case	Average Case	Applications
InsertSort	n^2	n	n^2	Very fast when n < 50
BubbleSort	n^2	n	n^2	Very fast when n < 50
MergeSort	nlogn	nlogn	nlogn	Need extra space; good for external sort
HeapSort	nlogn	nlogn	nlogn	Good for real-time app.
QuickSort	n^2	nlogn	nlogn	Practical and fast
BinSort	n+s	n+s	n+s	small, fixed range
RadixSort	k(n+s)	k(n+s)	k(n+s)	Need extra spaces

QUICK SORT

QUICK SORT

Invented by **C.A.R. Hoare** in **1960**

Fast Sorting Algorithm

What Makes it fast??
Divide and Conquer principle

- **Divide** a problem into smaller independent subproblems
  Solve these Sub problems individually
- **Conquer:** Solve the main problem by combining the solutions of sub problems

Quick Sort Algorithm

- Given an array of n values, randomly pick an element (*pivot*) in the array to partition.
- Then compare the rest of the elements to this value.
- If they are greater, put them to the "right" of the partition element
- If they are less, put them to the "left" of the partition element.
- When done with the partition, the position of the pivot element is fixed.
- Sort the left partition, using Quick Sort (Recursive!)
- Sort the right part of the array, using Quick Sort. (Recursive!)

Quick Sort Partition Algorithm

```
int partition1( int *a, int low, int high ) {
    int left, right, pivot_item;
    pivot_item = a[low];
    left = low;
    right = high;
    while ( left < right ) {
        /* Move left while item < pivot */
        while( a[left] <= pivot_item ) left++;
        /* Move right while item > pivot */
        while( a[right] >= pivot_item ) right--;
        if ( left < right ) SWAP(a,left,right);
    }
    /* right is final position for the pivot */
    SWAP(a, low, right);
    return right;
}
```


QUICK SORT contd..

A simple Animation

QUICK SORT Analysis

Focus of analysis: Selecting Pivot element / partition

Quick Sort Running Time: T(n) = n-1 + T(i-1) + T(n-i) ---→ (1)

BESTCASE: Split in the middle, pivot = n/2
T(n) = 2T(n/2) + O(n),
Solving recursive function gives **O(nlogn)**, the ideal case of QuickSort.

WORST CASE:
- Pivot = last or first element of array
- T(n) = n-1 + T(0) + T(n-1) from(1) sub i = 1 or n
  T(n) = $O(n^2)$

AVERAGE CASE : Split at random position
T(n) = 2(T(0)+. . .+T(n-1))/n + O(n)

QUICK SORT Analysis

Improving Worst case

- Choose random partitioning element
- Randomized QuickSort by Hoare 1960
- Randomized QuickSort keeps running time independednt of Input

<u>Other ways – Median of three elements as pivot</u>

- Randomly pick three elements in the array to be sorted
- Choose the middle value of these three elements to be the partition
- Complexity – Less if size of array to be sorted is large.
- Gain: Good partition element

3 way Radix Quick Sort

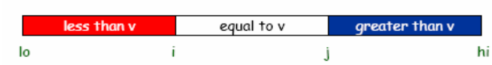
3 way Radix Quick Sort

- Keep all duplicates together in partitioning step.
  - Best for Quick Sort with equal keys
    - n - # of distinct keys
    - N – total # of elements
    - 3 way radix sort is efficient for
      N >> n i.e. efficient for array with equal keys

3 way partitioning

A way to deal with equal keys.

Partition elements into 3 parts:
- elements between i and j equal to partition element v
- larger elements to the right of i
- smaller elements to the left of j

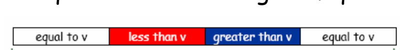


Aspects of 3 way partitioning
- Not much code.
- In-place.
- Linear if keys are all equal.
- Small overhead if no equal keys.

Dutch National Flag Problem

Problem: Re-arrange an array of elements in to top, middle and bottom

An Elegant solution to Dutch national flag problem.
Partition elements into 4 parts:
- no larger elements to left of m
- no smaller elements to right of m
- equal elements to left of p
- equal elements to right of q



Then, swap equal keys to center of array.

3 way Radix Quick Sort Algorithm

```
private static void quicksortX(String a[], int lo, int hi, int d) {
if (hi - lo <= 0) return;
int i = lo-1, j = hi, p = lo-1, q = hi;
char v = a[hi].charAt(d);
while (i < j) {
while (a[++i].charAt(d) < v) ;
while (v < a[--j].charAt(d))
if (j == lo) break;
if (i > j) break;
exch(a, i, j);
if (a[i].charAt(d) == v) { p++; exch(a, p, i); }
if (a[j].charAt(d) == v) { q--; exch(a, j, q); }
}
if (p == q) {
if (v != '\0') quicksortX(a, lo, hi, d+1);
return;
}
if (a[i].charAt(d) < v) i++;
for (int k = lo; k <= p; k++, j--) exch(a, k, j);
for (int k = hi; k >= q; k--, i++) exch(a, k, i);
quicksortX(a, lo, j, d);
if ((i == hi) && (a[i].charAt(d) == v)) i++;
if (v != '\0') quicksortX(a, j+1, i-1, d+1);
quicksortX(a, i, hi, d);
}
```


Some sample Functions

- Records with equal keys together.
- Finding collinear points.
- Sort population by age.
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

COUNTING SORT

COUNTING SORT

→ Works by counting the occurrences of each data value in the array

→ n data items in the range from 0 to k for some integer k.

→ Algorithm can then determine, for each input element, the amount of elements less than i

Arrays

→ A[1…….n] is the input array

→ B[1……..n] has the sorted array in the end

→ C[1….k] for some constant k working array to hold the number of occurrences in the original array A[1…..n]

→ No Comparison made in the counting sort

COUNTING SORT ALGORITHM

```
Counting-Sort ( A, B, k )

for i 1 to k
     do C[i] ¬ 0
for j ¬ 1 to length[A]
    do C[A[j]] ¬ C[A[j]] + 1

// C[i] now contains the number of elements equal to k.
for i ¬ 2 to k
    do C[i] ¬ C[i] + C[i-1]
// C[i] now contains the number of elements less than or
    equal to i
for j ¬ length[A] downto 1
   do B[C[A[j]]] ¬ A[j]
        C[A[j]] ¬ C[A[j]] - 1
```


Counting Sort Demo

A	3	6	4	1	3	4	1	4	C	2	0	2	3	0	1
									C	2	2	4	7	7	8
B							4		C	2	2	4	6	7	8
B		1					4		C	1	2	4	6	7	8
B		1				4	4		C	2	2	4	5	7	8
B	1	1	3	3	4	4	4	6							

M E R G E S O R T

Description of Merge Sort

Recursive sorting procedure
- uses $O(n \log n)$ comparisons in the worst case

Algorithm

- If n < 2 then the array is already sorted. Stop now.
-
- Otherwise, n>1, and we perform the following three steps in sequence:

  1. Sort the left half of the the array.
  2. Sort the right half of the the array.
  3. Merge the now-sorted left and right halves.

Comparisons in Merger Sort

Worst-case number of comparisons

$T(n) = T(n/2) + T(n/2) + n$, if $n>1$.

- **First term** - number of comparisons used to sort the left half of the array
- **Second term** - number of comparisons used to sort the left half of the array
- **Third term** - an upper bound on the number of comparisons used to merge two sorted arrays

Comparisons in Merge Sort contd.

Suppose we want to sort 16 elements, T(16) = ?

T(16) = 2T(8) + 16
T(8) = 2T(4) + 8
T(4) = 2T(2) + 4
T(2) = 2T(1) + 2
T(1) = 0

i.e. T(1) = 0
T(2) = 2T(1) + 2 = 0 + 2
T(4) = 2T(2) + 4 = 4 + 4 = 8
T(8) = 2T(4) + 8 = 16 + 8 = 24
T(16) = 2T(8) + 16 = 48 + 16 = 64

So MergeSort requires at most 64 comparisons to sort 16 elements.

MERGE SORT OVERVIEW

- Good running time for most data sets.
- Based on divide and conquer approach.
- Chops partition half way by dividing (left + right)/2
- Successive pair of elements are used to combine when we merge
- This algorithm works in two sorted input lists of unequal size.
- Merging takes n/2 comparisons.
- No more than (n – 1) comparisons.

MERGE SORT (CONT)

→Merge Sort uses the divide and conquer approach.

→ Divides the element into two sub lists n/2.

→ If there are n elements, two sub arrays are created such that

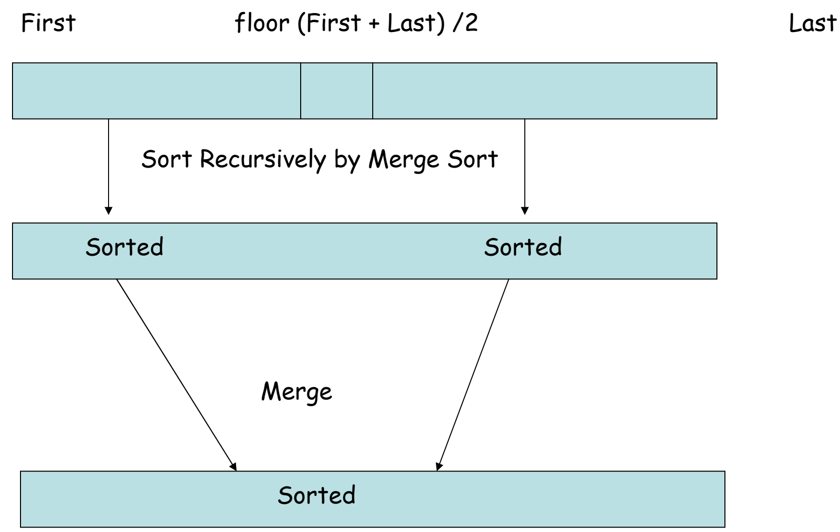
Left Sub Array [1…….r] Right Sub Array [r +1, n]

→Both the sub Arrays are sorted accordingly.

→ Once the Arrays are sorted they are combined into a single array for A[1….n]. (Conquer Approach)

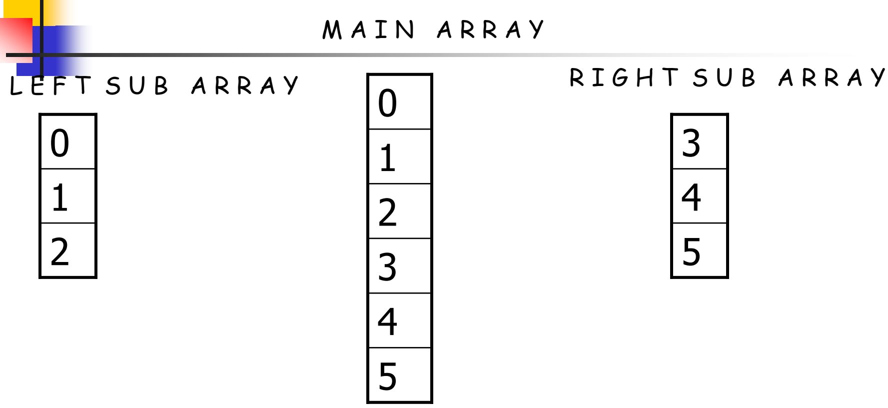
→Sorting is done in the Merge itself.

→ Conquering is done recursively using the Merge Sort.



Diagrammatic representation - Merge Sort

MERGE EXAMPLE ON TWO SORTED SUB ARRAYS



A comparison is made between the first element of the left sub array And the right sub array. Which ever is greater goes into the first place of The main Array. The appropriate index of the array is increased.

MERGE EXAMPLE ON TWO SORTED SUB ARRAYS

LEFT SUB ARRAY

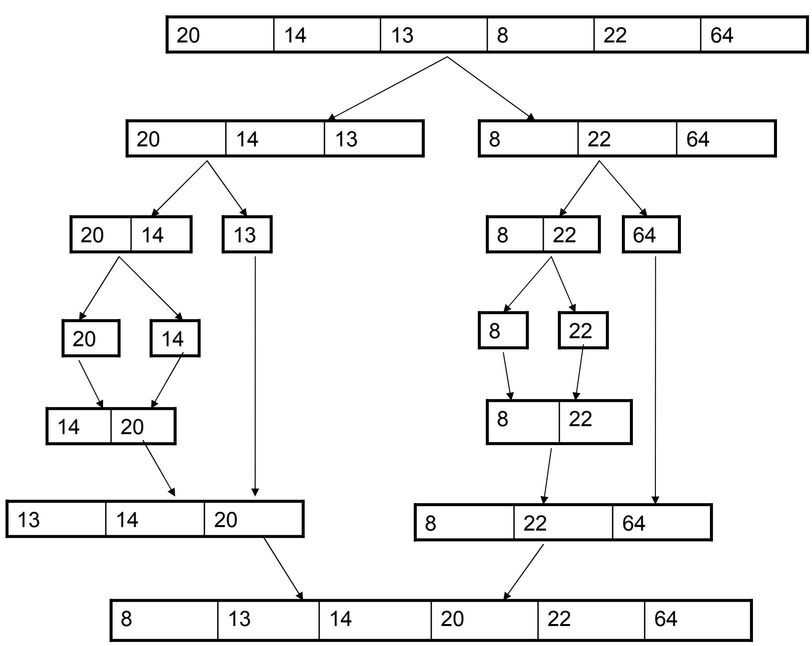
0	30
1	35
2	40

0	10
1	20
2	30
3	30
4	35
5	40

RIGHT SUB ARRAY

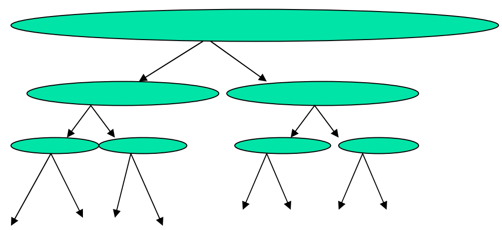
3	10
4	20
5	30

Left[0] is compared with Right[0]. R[0] is greater than L[0]
R[0] is placed in the MainArray at 0.
R index is increased by 1.



MERGE SORT ANALYSIS

Depth	# sequences	size
0	1	n
1	2	n/2
2	2^2	$n/2^2$
3	2^3	$n/2^3$
i	2^i	$n/2^i$



Calculation of Running Time Of Merge Sort

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

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 4T(n/4) + 2n \\ &= 8T(n/8) + 3n \\ &\vdots \\ &= 2^k T(n/2^k) + kn \\ &\vdots \\ &= nT(1) + n\log_2 n \\ &= n + n\log_2 n \end{aligned}$$

Source: http://www.brpreiss.com/books/opus4/html/page513.html

MERGE SORT EFFICIENCY

→ Operations in O(n lg n) time.

→ No best -case and worst case as Quick Sort where the worst case is $O(n^2)$

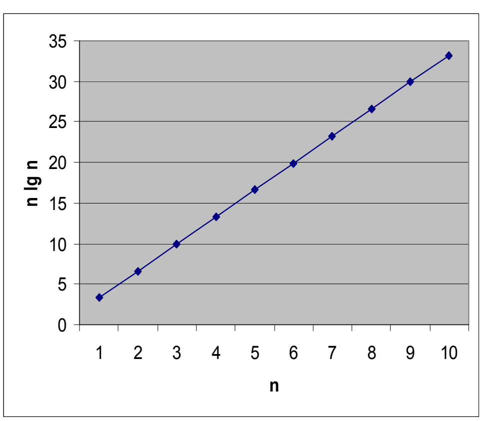
→Needs a duplicate copy of the array being sorted.

→ Merge Sort is used mainly for small size of n.

→ If n is large it is very impractical to apply the algorithm.

→ Merge Sort is fast for small n.

MERGE SORT ANALYSIS



n	n lg n
10	3.321928
100	6.643856
1000	9.965784
10000	13.28771
100000	16.60964
1000000	19.93157
10000000	23.2535
100000000	26.57542
1000000000	29.89735
10000000000	33.21928

n	log n	n log n	n^2 Quick Sort (Worst Case)
10	3.3219	33.2193	100.0000
50	**5.6439**	**282.1928**	**2500.0000**
100	**6.6439**	**664.3856**	**10000.0000**
110	6.7814	745.9496	12100.0000
120	6.9069	828.8269	14400.0000
130	7.0224	912.9078	16900.0000
140	7.1293	998.0996	19600.0000
150	**7.2288**	**1084.3228**	**22500.0000**
160	7.3219	1171.5085	25600.0000
170	7.4094	1259.5965	28900.0000
180	7.4919	1348.5336	32400.0000
190	7.5699	1438.2726	36100.0000
200	**7.6439**	**1528.7712**	**40000.0000**
300	**8.2288**	**2468.6456**	**90000.0000**
400	**8.6439**	**3457.5425**	**160000.0000**
500	**8.9658**	**4482.8921**	**250000.0000**

QUICK SORT VS MERGE SORT

→ Quick Sort does not require an extra array to work whereas Merge Sort needs an extra array to merge the two sorted sub lists into the main array.

→ Merge Sort has better worst case behavior than Quick Sort

→ A general sorting algorithm cannot do better than a average case of O (n log n).

→ Merge Sort is a sorting algorithm which is not in-place when dealing with arrays.

HEAP SORT

HEAP SORT

→ Heap Sort is one of the best general-purpose algorithms

→ Part of Selection Sort Family

→ Very good running time performance on randomly ordered arrays.

→ Worst case is same as the average case performance.

→ Needs only a fixed amount of extra storage space.

→ In-Place algorithm.

HEAP SORT

→ Heap Sort basically works with an array.

→ There are two types of heaps.

<u>→ Min Heaps</u>

Smallest element is
At the root.

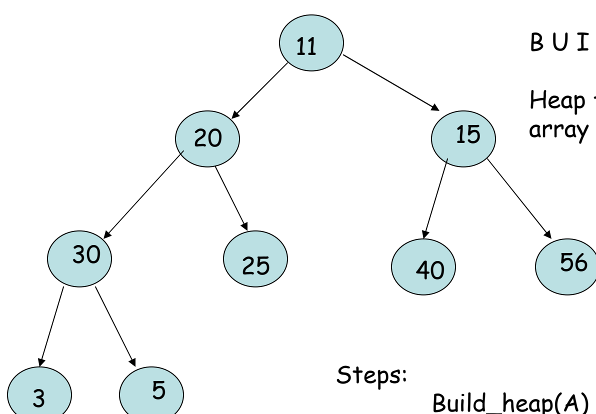
<u>→ Max Heaps.</u>

Largest Element is at the root.

→ Every new element taken from the array has to be inserted into the correct place.

→ Once inserted into an heap we got to address the locality of the element.

Input: 11, 20, 15, 30, 25, 40, 56, 3, 5 (Max Heap)



BUILD HEAP

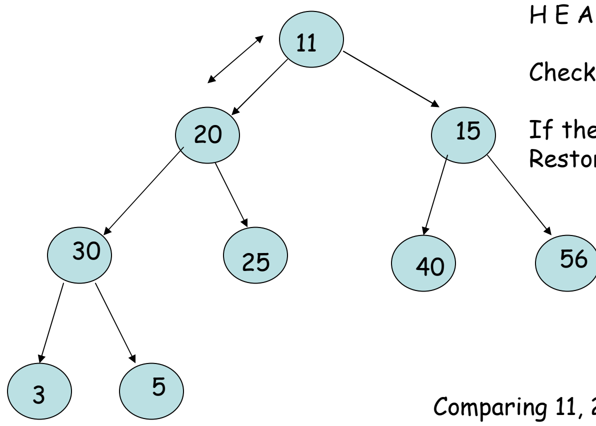
Heap from a unordered input array

Steps:

Build_heap(A)
heap_size(A) = |A|
for i = (A/2) down to 1
Heapify (A, i)

Running time is $\sum_{i=1}^{\log n} n/2^i = O(n)$

Input: 11, 20, 15, 30, 25, 40, 56, 3, 5



HEAPIFY

Checks for heap property.

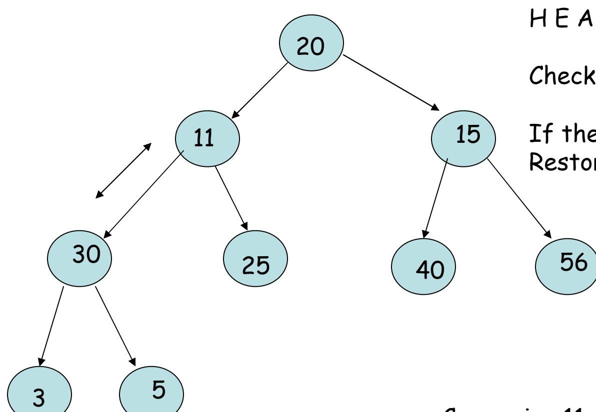
If the heap property is violated
Restores the heap

Comparing 11, 20 and 15

20 is greater than 11 and 15.

Move 20 to root.

Input: 11, 20, 15, 30, 25, 40, 56, 3, 5



HEAPIFY

Checks for heap property.

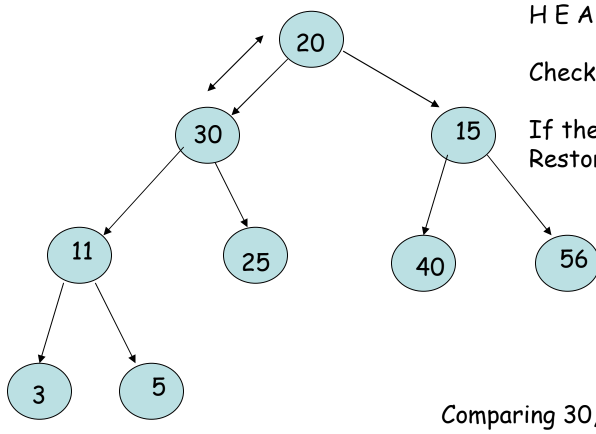
If the heap property is violated
Restores the heap

Comparing 11, 30 and 25

30 is greater than 11 and 25.

Swap 30 with 11

Input: 11, 20, 15, 30, 25, 40, 56, 3, 5



HEAPIFY

Checks for heap property.

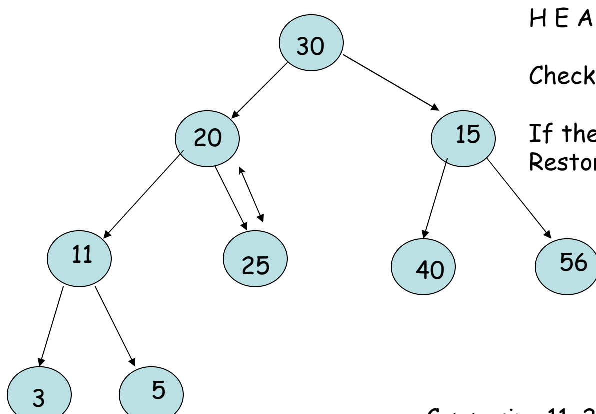
If the heap property is violated
Restores the heap

Comparing 30, 20 and 15

30 is greater than 20 and 15.

Move 30 in the place of 20.

Input: 11, 20, 15, 30, 25, 40, 56, 3, 5



HEAPIFY

Checks for heap property.

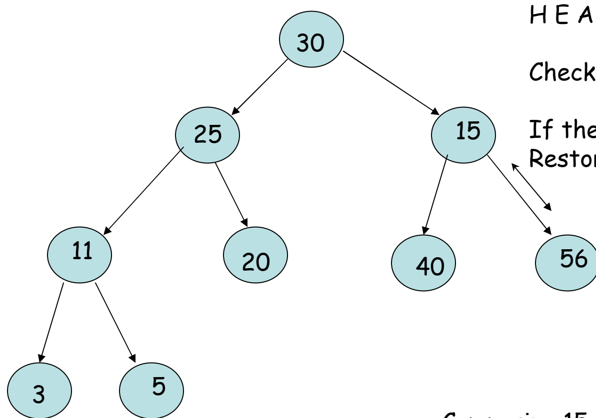
If the heap property is violated
Restores the heap

Comparing 11, 20 and 25

25 is greater than 20 and 11.

Move 25 in the place of 20.

Input: 11, 20, 15, 30, 25, 40, 56, 3, 5



HEAPIFY

Checks for heap property.

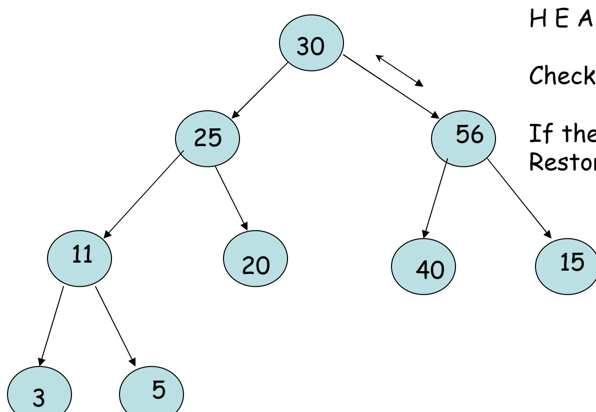
If the heap property is violated
Restores the heap

Comparing 15, 40 and 56

56 is greater than 15 and 40.

Move 56 in the place of 15.

Input: 11, 20, 15, 30, 25, 40, 56, 3, 5



HEAPIFY

Checks for heap property.

If the heap property is violated
Restores the heap

Comparing 56, 25 and 30

56 is greater than 25 and 30.

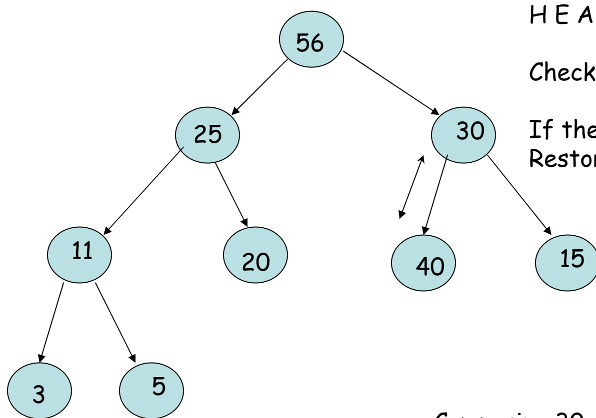
Move 56 in the place of 30.

Input: 11, 20, 15, 30, 25, 40, 56, 3, 5

HEAPIFY

Checks for heap property.

If the heap property is violated
Restores the heap



Comparing 30, 40 and 15

40 is greater than 30 and 15.

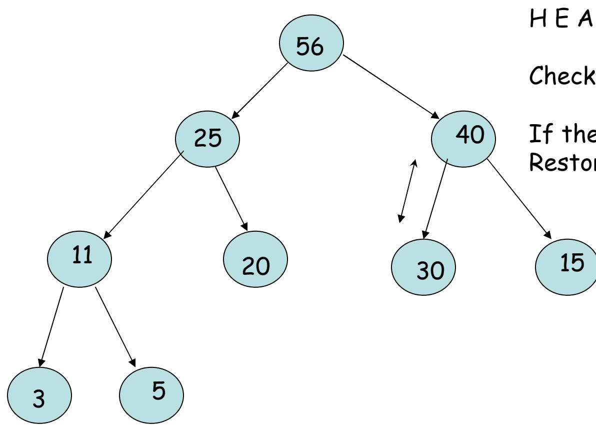
Move 40 in the place of 30.

Input: 11, 20, 15, 30, 25, 40, 56, 3, 5

HEAPIFY

Checks for heap property.

If the heap property is violated
Restores the heap



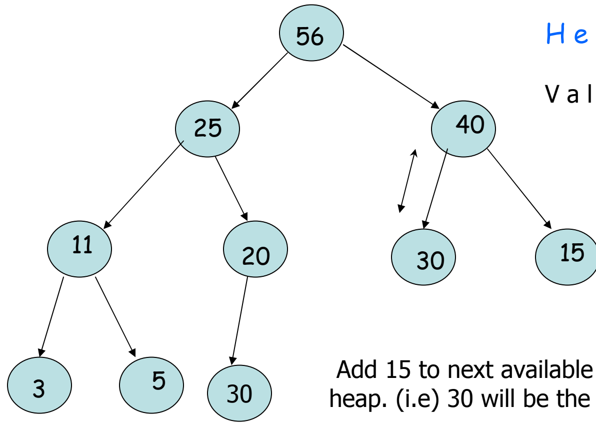
HeapIfy algorithms

Done for the given Input

Input: 11, 20, 15, 30, 25, 40, 56, 3, 5

Heap Insertion

Value = 30



Add 15 to next available position in the heap. (i.e) 30 will be the child of 20.

Do the Heapify Algorithm again to restore heap property

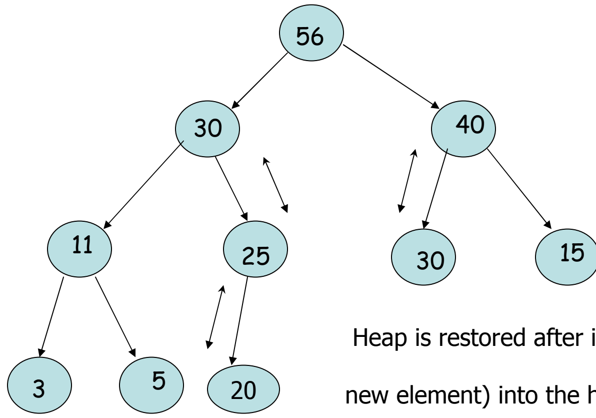
Procedure For InsertIon Into Heap

Algorithm $\text{HEAPINSERT}(H, n, \preceq, x)$
Input: A heap $(H, \preceq)$ (represented as an array) containing n values and a new value x to be inserted into H
Output: H and n, with x inserted and the heap property preserved

$n \leftarrow n + 1$
$H[n] \leftarrow x$
$child \leftarrow n$
$parent \leftarrow n$ div 2
while $parent \geq 1$
  if $H[child] \preceq H[parent]$ **then**
    $\text{SWAP}(H[parent], H[child])$
    $child \leftarrow parent$
    $parent \leftarrow parent$ div 2
  else $parent \leftarrow 0$

Source: http://planetmath.org/encyclopedia/HeapInsertionAlgorithm.html

Input: new element 30 into the heap

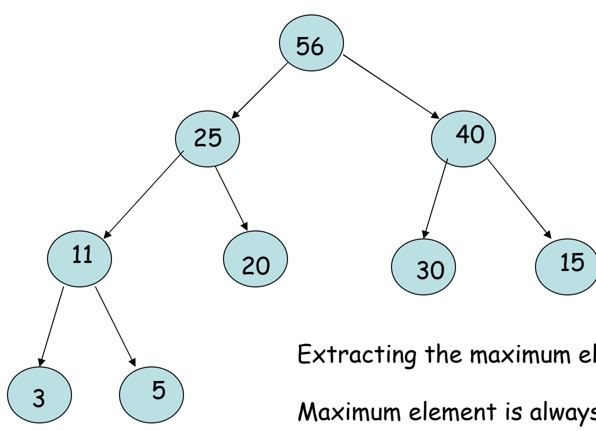


Heap is restored after inserting 30(the new element) into the heap.

Time Complexity O (log n)

We need to traverse in the heap till we find a position for the new element in the heap. (Position Found = Insert the element there

HEAP EXTRACT MAX



Extracting the maximum element

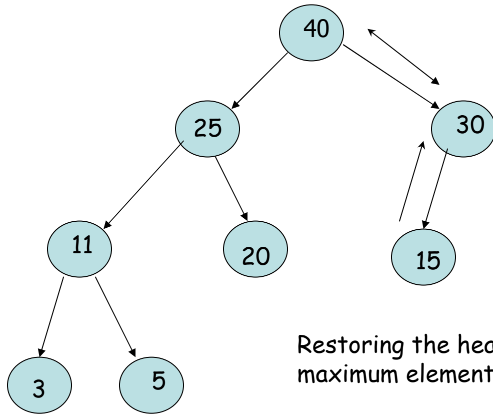
Maximum element is always at the root.

In this example it is 56.

Extract 56 and restore the heap property.

Input: 11, 20, 15, 30, 25, 40, 56, 3, 5

HEAP EXTRACT MAX



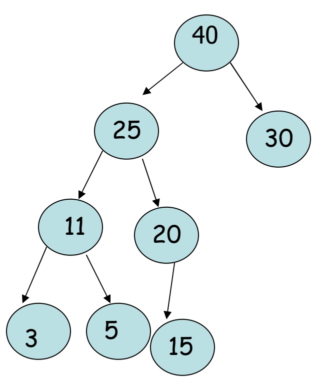
Restoring the heap after extracting the maximum element.

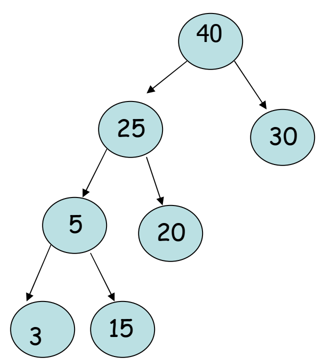
40 is moved to the root

30 is moved in the place of 40.

Running Time is O (lg n)

HEAP DELETION



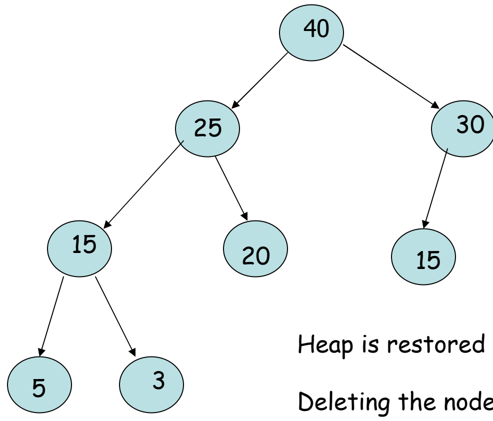


Deleting the node 11.

After Deleting the node 11. We move 5 to the top and move 15 to the right child of 5

Do the heapify algorithm to restore heap property

HEAP DELETION (Contd)



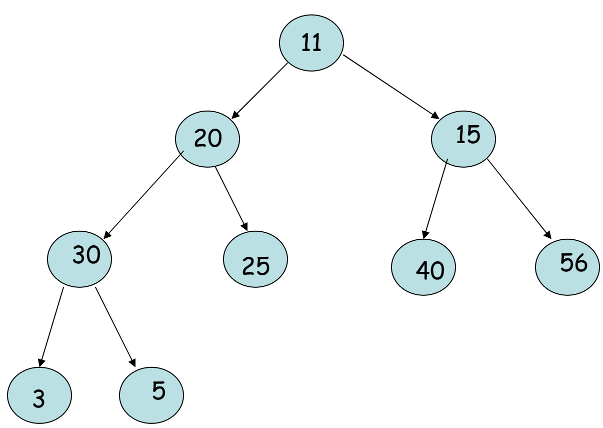
Heap is restored after

Deleting the node 11 from the heap.

Deletion takes O(log n) time to delete the node

And call the heapify algorithm

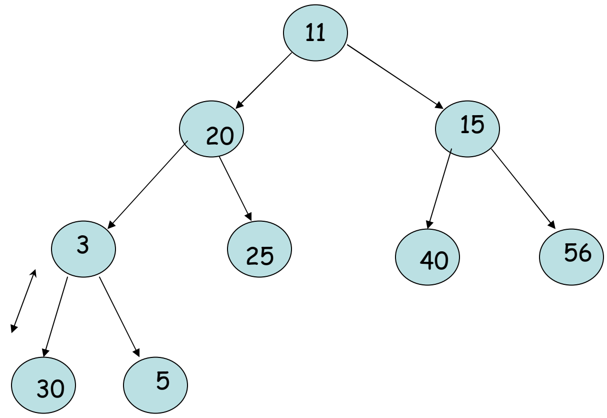
Input: 11, 20, 15, 30, 25, 40, 56, 3, 5



Min Heap

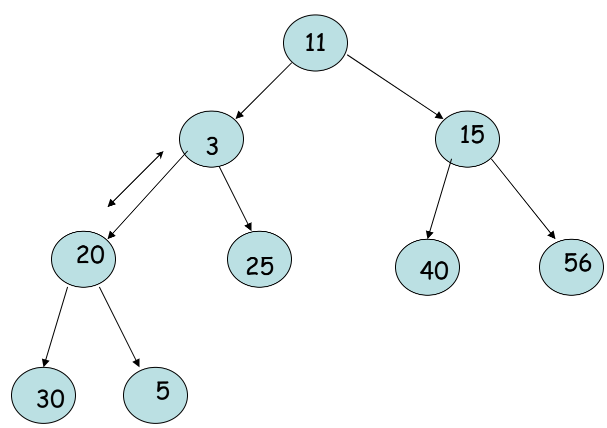
Heap Constructed from the array

Input: 11, 20, 15, 30, 25, 40, 56, 3, 5



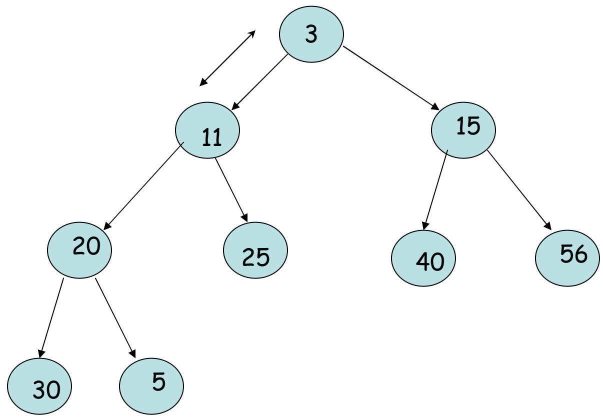
Min Heap
Comparing 30,5 and 3.
3 is less than 30 and 5.
Swap 30 with 3

Input: 11, 20, 15, 30, 25, 40, 56, 3, 5



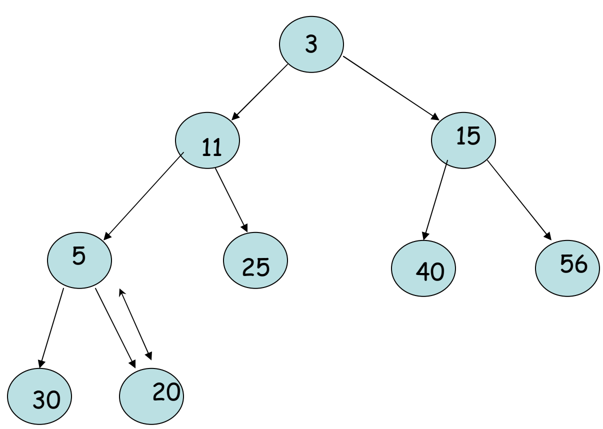
Min Heap
Comparing 3,20 and 25.
3 is less than 20 and 25.
Swap 20 with 3.

Input: 11, 20, 15, 30, 25, 40, 56, 3, 5



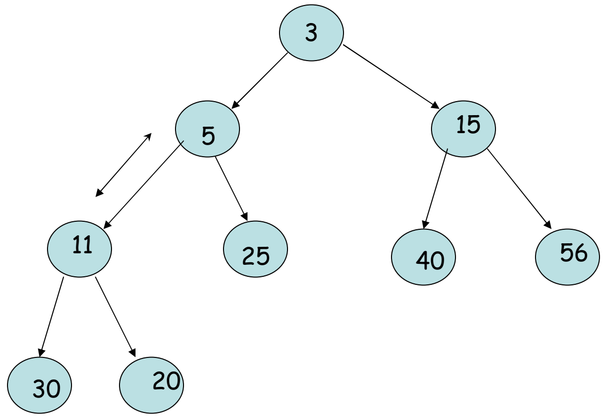
Min Heap
Comparing 3,11 and 15.
3 is less than 11 and 15.
Swap 3 with 11.

Input: 11, 20, 15, 30, 25, 40, 56, 3, 5



Min Heap
Comparing 5,20 and 30.
5 is less than 20 and 30.
Swap 5 with 20.

I n p u t : 11, 20, 15, 30, 25, 40, 56, 3, 5

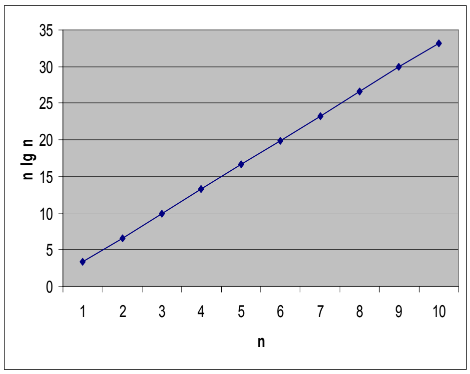


Min Heap
Comparing 5,11 and 25.
5 is less than 11 and 25.
Swap 5 with 11.
Heap Property Restored

HEAP SORT (TIME COMPLEXITY)

Build – Max Heap	O(n)
Max – Heapify	O(lg n)
Heap Sort	O(n lg n)
Heap – Maximum	O(1)
Heap – Extract Max	O(lg n)
Heap- Increase Key	O(lg n)
Max-Heap-Insert	O(lg n)
Heap Search	O(n)
Heap Delete	O(lg n)

HEAP SORT ANALYSIS



n	n lg n
10	3.321928
100	6.643856
1000	9.965784
10000	13.28771
100000	16.60964
1000000	19.93157
10000000	23.2535
100000000	26.57542
1000000000	29.89735
10000000000	33.21928

STRENGTHS AND WEAKNESS OF HEAP SORT

STRENGTHS

All data types.

Medium Complexity

WEAKNESS

Not a stable sort since every insert or delete operation heapify must be done.

HEAP SORT VS QUICK SORT

Worst case running time of Quick Sort is O(n2) while the Worst Case running time of Heap Sort is same as the best and average case O(n lg n).

→ Heap Sort algorithm is used where there is unacceptable complexity for the running time of the algorithm

→ Embedded systems with real-time constraints often uses Heap Sort algorithm

RUNNING TIME COMPLEXITY

	Insertion Sort	Selection Sort	Bubble Sort	Merge Sort
Avg. Compares	e.g. $O(n^2)$	$O(n^2)$	$O(n^2)$	O (n log n)
Avg. Copies	$O(n^2)$	O (n)	$O(n^2)$	O (n log n)
Worst Compares	$O(n^2)$	$O(n^2)$	$O(n^2)$	O (n log n)
Worst Copies	$O(n^2)$	O (n)	$O(n^2)$	O (n log n)
Best Compares	O (n)	$O(n^2)$	$O(n^2)$or O (n)	O (n log n)
Best Copies	O(1)	O(1)	O(1)	O (n log n) or O(1)
When Worst	Reverse sorted	Reverse sorted	Reverse Sorted	---
When Best	Already sorted	Already sorted	Already sorted	---

Source: http://ccl.northwestern.edu/tisue/cs311/spring-01/sq2/answers.html

KEY POINTS

Sorting in O (n log n)

→Most of the sorting algorithm will take at least O(n lg n) steps to sort n items.

→ Reason: Comparison of the data items in the array

→ Moving the data items from one position to another

→ Working with extra space (Sometimes in Quick Sort Worst Case)

Bibliography

http://www.devx.com/vb2themax/Article/19900

http://www.fearme.com/misc/alg/node42.html

http://www.informatik.uni-stuttgart.de/ipvr/sgs/lehre/seminare/famous_algorithms_ws03_04/01_abstract.html

http://www.cs.princeton.edu/~rs/talks/Montreal.pdf

http://www.cs.colorado.edu/~karl/2270.fall03/sorting.html

http://www.cs.rit.edu/~jdb/cs2/qsortAndMSort.pdf

http://www.cs.princeton.edu/courses/archive/spring04/cos226/lectures/radix.4up.pdf

R.L. Wainwright, *A Class of Sorting Algorithms based on Quicksort*, Communications of the ACM, Vol. 28, No. 4, April 1985, pgs. 396-402.

C.R. Cook, and Kim D.J, *Best sorting algorithm for nearly sorted lists*, Communications of the ACM, Vol. 23, No. 11, Nov. 1980, pgs. 620-624.