Advanced Analysis of Algorithms

Department of Computer Science
Swat College of Science & Technology

CS Course : Advanced Analysis of Algorithms
Course Instructor : Muzammil Khan

Chapter 8

Sorting Algorithms
(Advanced Sorting Algorithms II)

Heap Sort

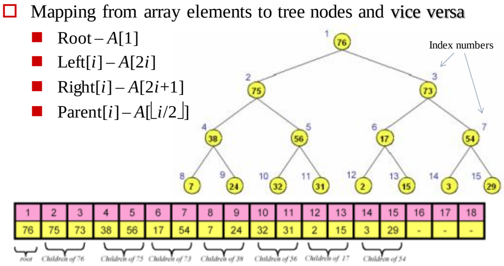
- Combines the better attributes of merge sort and insertion sort
  - Like merge sort but unlike insertion sort
    - The running time is *O(n lg n)*
  - Like insertion sort but unlike merge sort
    - Sorts *in place*
- Heap sort is *always O(n log n)*
  - Quick sort is usually *O(n log n)* but in the worst case slows to $O(n^2)$
  - Quick sort is generally faster, but Heap sort has the guaranteed O(n log n) time
    - Can be used in *time-critical* applications

Heap

- Introduces an algorithm design technique
  - Create data structure (*heap*) to manage information during the execution of an algorithm
- The *heap* has other applications beside sorting
  - Priority Queues (may discuss later)
- Array viewed as a nearly complete binary tree
  - Physically – linear array
  - Logically – binary tree
    - Complete filled on all levels(except lowest one) left to right
  - length[*A*] – number of elements in array *A*
  - heap-size[*A*] – number of elements in heap stored in *A*
    - heap-size[*A*] ≤ length[*A*]

Heap Construction

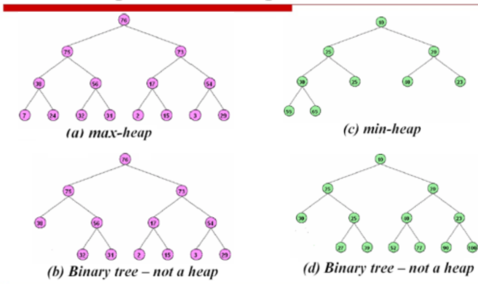
- Mapping from array elements to tree nodes and vice versa
  - Root – *A*[1]
  - Left[*i*] – *A*[2*i*]
  - Right[*i*] – *A*[2*i*+1]
  - Parent[*i*] – *A*[⌊*i*/2⌋]



Heap Properties (Max & Min Heaps)

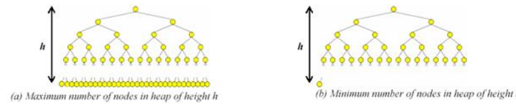
- Max-Heap
  - For every node excluding the root, value is at most that of its parent
    - *A*[*parent*[*i*]] ≥ *A*[*i*]
  - Largest element is stored at the root
    - In any subtree, no values are larger than the value stored at subtree root
- Min-Heap
  - For every node excluding the root, value is at least that of its parent
    - *A*[*parent*[*i*]] ≤ *A*[*i*]
  - Smallest element is stored at the root
    - In any subtree, no values are smaller than the value stored at subtree root

Max-Heap & Min-Heap



Height Theorem

- Theorem: The height of a *heap* with *n* nodes is $\lfloor lg(n) \rfloor$
- Proof : Let *h* be the height of a heap as shown



  - 2^0 nodes at level 0, 2^1 nodes at level 1, and so on…
- In case (a)
  - $2^0 + 2^1 + 2^2 + . . . + 2^h = 2^{h+1}-1$
- In case (b)
  - $2^0 + 2^1 + 2^2 + . . . + 2^{h-1} + 1= 2^h$

Height Theorem (Cont...)

- Since ***n*** is number of node in the heap, so
  - ***n*** can be equal to Maximum, Minimum or Lie in between
  - Therefore $2^h \le n \le 2^{h+1} -1$ Or. $2^h \le n < 2^{h+1}$
  - Taking *log to the base 2*, implies that;
    - $h \le lg\ n < h+1$
  - Using property of the floor function
    - $\lfloor lg\ (n) \rfloor$
- Concludes
  - Minimum number of elements in *heap of height* ***h*** is 2^h
  - Maximum number of elements in *heap of height* ***h*** is 2^{h+1}
  - An ***n****-element* heap has height $\lfloor lg\ (n) \rfloor$

Heap's - Basic Operations

- The basic operations on heap data structure, *implemented as an array*, are;
  - Parent(i)
    - Returns index of the ***parent of the node*** identify by index ***i***
  - Left(i)
    - Returns index of the ***left child of the node*** identify by index ***i***
  - Right(i)
    - Returns index of the ***right child of the node*** identify by index ***i***
  - Build-heap(A)
    - Convert an ***unsorted array A*** into ***a heap***
  - Heapify(A, i)
    - Restores ***heap order property*** by adjusting the key stored in a node with index ***i***

Heap's - Basic Operations (Cont...)

- Parent Procedure

```
PARENT( i )
k ← ⌊ i / 2 ⌋
return k
```

- Left Child Procedure

```
LEFT( i )
k ← 2i
return k
```

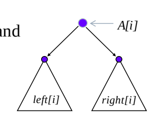
- Right Child Procedure

```
RIGHT( i )
k ← 2i + 1
return k
```

  - All procedures require fixed amount of time to do computations on array, thus running time is constant i.e. $\theta(1)$

Maintaining the Heap Property

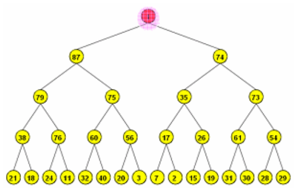
- *Max-Heapify* is a sub-routine for manipulating max-heaps
- *Max-Heapify* function
  - Fix a node *A[i]* that violates the max-heap property
    - That is; a node that is smaller than its children
- When *Max-Heapify* is called
  - It is assumed that trees rooted at *Left[i]* and *Right[i]* are max-heaps



- *Max-Heapify* lets the value at the node *A[i]* "float" down in the max-heap

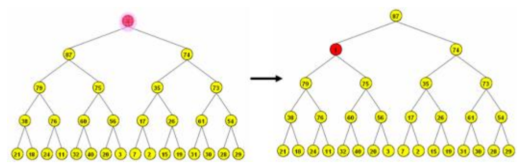
Maintaining the Heap Property (Cont...)

- Let Consider the following *heap tree*,
  - But, *does not maintain heap property*



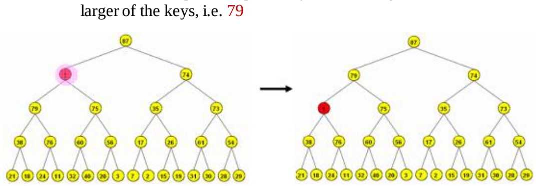
Maintaining the Heap Property (Cont...)

- The key "1" at the root violates the heap order property
  - It is smaller than both the keys, 87 and 74, at child nodes
  - In order to fix up the heap, the key 1 is exchanged with the larger of the keys, i.e. 87

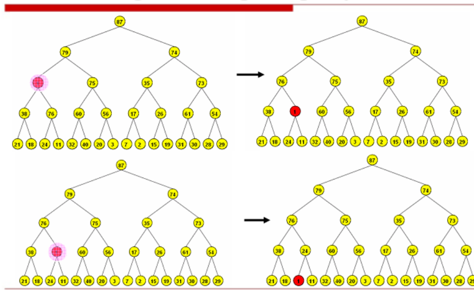


Maintaining the Heap Property (Cont...)

- The key "1" at the new position violates the heap order property
  - It is smaller than both the keys, 79 and 75, at child nodes
  - In order to fix up the heap, the key 1 is exchanged with the larger of the keys, i.e. 79



Maintaining the Heap Property (Cont...)



Heap in Sorting

- Use max-heaps for sorting (Max-heap array is unsorted)
- Steps in sorting
  - Convert the given array of size ***n*** to a max-heap (*BuildMaxHeap*)
  - Swap the first and last elements of the array
    - Now, the largest element is in the last position, where it belongs
    - That leaves ***n* – *1*** elements to be placed in their appropriate locations
    - However, the array of first ***n* – *1*** elements is no longer a max-heap
    - Float the element at the root down one of its subtrees so that the array remains a max-heap (MaxHeapify)
  - Repeat step 2 until the array is sorted

Heap Sort Procedure

- Sort by
  - Maintaining the unsorted elements as a max-heap
- Start by building a max-heap on all elements in *A*
  - Maximum element is in the root, *A*[1]
- Move the maximum element to its correct final position
  - Exchange *A*[1] with *A*[*n*]
- Discard *A*[*n*] – it is now sorted
  - Decrement heap-size[*A*]
- Restore the max-heap property on *A*[1..*n*–1]
  - Call *MaxHeapify*(*A*, 1)
- Repeat until heap-size[*A*] is reduced to 2

Heap Sort Pseudocode

HeapSort(A)

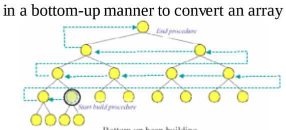
1. Build-Max-Heap(A)
2. **for** $i \leftarrow$ *length*[A] **downto** 2
3. **do** exchange $A[1] \leftrightarrow A[i]$
4. *heap-size*[A] $\leftarrow$ *heap-size*[A] – 1
5. *MaxHeapify*(A, 1)

Build-Max-Heap(A) Pseudocode

BuildMaxHeap(A)

1. *heap-size*[A] $\leftarrow$ *length*[A]
2. **for** $i \leftarrow \lfloor$*length*[A]/2$\rfloor$ **downto** 1
3. **do** *MaxHeapify(A, i)*

- Max-heapify can be used in a bottom-up manner to convert an array into a max-heap



Bottom up heap building

MaxHeapify(A, i) Pseudocode

MaxHeapify(A, i)

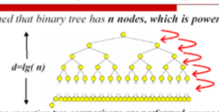
1. $l \leftarrow$ left(i) ► ***l*** stores the index of left child of parent identify by index ***i***
2. $r \leftarrow$ right(i) ► ***r*** stores the index of left child of parent identify by index ***i***
3. **if** $l \le$ *heap-size*[A] and $A[l] > A[i]$ ► check if key at left child is larger than parent node
4. **then** *largest* $\leftarrow l$ ► if yes, *key at left* is saved in variable ***largest***
5. **else** *largest* $\leftarrow i$ ► else, *key at right* is saved in variable ***largest***
6. **if** $r \le$ *heap-size*[A] **and** $A[r] > A[largest]$ ► larger of the keys at left and right is chosen
7. **then** *largest* $\leftarrow r$
8. **if** *largest* $\neq i$
9. **then** exchange $A[i] \leftrightarrow A[largest]$ ► elements pointed by ***largest*** and ***i*** are exchanged
10. *MaxHeapify*(A, *largest*) ► ***MaxHeapify*** procedure called recursively to the key at new location
11. **Return A** ► Heap is fixed up

Analysis of HeapSort Algorithm

- Build-Max-Heap takes $O(n)$ and each of the n-1 calls to MaxHeapify takes time $O(\lg n)$
- That is
  - MaxHeapify $O(\lg n)$
  - BuildMaxHeap $O(n)$
- Therefore HeapSort $T(n) = O(n \lg n)$

Analysis of Heap Building ($O(n)$)

➢ *It is assumed that binary tree has **n nodes, which is power of 2**, and **d** is **tree depth***



- *In heapifying operation **two comparisons are performed** among the parent and child nodes*
- *In **worst case all the keys** would be processed and **shifted down to leaf level***
- *The table summarizes the number of comparisons on different levels*

Depth	Number of nodes	Number of comparisons
0	2^0	2(d-1)
1	2^1	2(d-2)
2	2^2	2(d-3)
--	--	--
k	2^k	2(d-k-1)
---	---	--
d-1	2^{d-1}	0 (Except first node)

Analysis of Heap Building (Cont...)

- *Total number of comparisons* $= 2\sum_{k=0}^{d-1} 2^k (d-k-1) + 2$
- $T_{build} = 2\sum_{k=0}^{d-1} 2^k (d-k-1) + 2$, *where* $d = \lg(n)$

$$= 2(d-1)\sum_{k=0}^{d-1} 2^k - 2\sum_{k=0}^{d-1} k.2^k + 2$$

$$\sum_{k=0}^{d-1} 2^k = 2^d - 1 \quad \textit{(sum of geometric series)}$$

$$\sum_{k=0}^{d-1} k.2^k = (d-2)2^d + 2 \quad \textit{(sum of arithmetic-geometric series)}$$

- $T_{build} = 2(2^d - 1) = 2(2^{\lg n} - 1) = 2(n-1) = \theta(n)$

➢ The heap build procedure runs in $\theta(n)$ time

Comparison of Sorting Algorithms

Algorithm	Worst Case	Average Case
Insertion Sort	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \lg n)$	$O(n \lg n)$
Quick Sort	$O(n^2)$	$O(n \lg n)$
Heap Sort	$O(n \lg n)$	$O(n \lg n)$

- The sorted order determined by these algorithms is based only on a comparison between input elements

Priority Queue

- Popular & important application of heaps
- Max and min priority queues
- Maintains a *dynamic* set *S* of elements
- Each set element has a *key* – an associated value
- Goal is
  - To support insertion and extraction efficiently
- Applications
  - Ready list of processes in operating systems by their priorities – the list is highly dynamic
  - In event-driven simulators to maintain the list of events to be simulated in order of their time of occurrence

Priority Queue (Cont...)

- Is a data collection of
  - Items and associated priority, called Keys
- Example

data →	A	X	Y	C	Z	H	I	D
priorities →	10	10	1	3	2	9	7	3

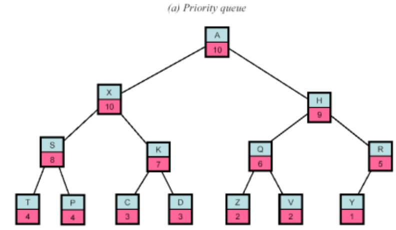
  - Characters "A" & "X" will be retrieved first, because of high priority

- A Priority Queue is
  - Naturally implemented as *heap*
  - In which *keys* are the *queue priorities*

Priority Queue (Cont...)

Data →	A	X	Y	C	Z	H	K	P	Q	R	S	T	V	D
Priorities →	10	10	1	3	2	9	7	4	6	5	8	4	2	3

(a) Priority queue



Basic Operations on Priority Queue

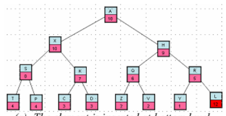
- Operations on a ***max-priority queue***
  - Insert(S, x) - inserts the element x into the set S
    - $S \leftarrow S \cup \{x\}$
  - Maximum(S) - returns the element of S with the largest key
  - Extract-Max(S) - removes and returns the element of S with the largest key
  - Increase-Key(S, x, k) – increases the value of element x's key to the new value k
- ***Min-priority queue*** supports Insert, Minimum, Extract-Min, and Decrease-Key
- Heap gives a good compromise between fast insertion but slow extraction and vice versa

Priority Queue Insertion Procedure

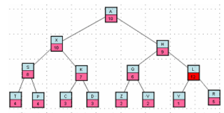
- The insertion algorithm adds an element to a priority queue
- It proceeds as follows
- Step # 1
  - Insert the new element at the last position in the heap
- Step # 2
  - Increase the heap size by one
- Step # 3
  - Use HEAPFY operation to fix the new element
    - Means; maintain heap property

Priority Queue Insertion

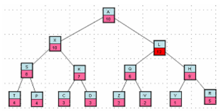
A new data element 'L' with priority 12 is inserted. The insertion procedure is illustrated by figures (a)-(d)



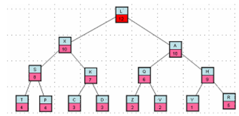
(a) The element is inserted at bottom level. The heap order property is violated



(b) The key 12 being larger, it is swapped with the key 5 at the parent node



(c) The key 12 being larger, it is swapped with the key 9 at the parent node



(d) Finally, new key is moved to root. The heap order property is restored

Priority Queue Insertion Algorithm

Heap-Insert(A, key)

```
heap-size[A] ← heap-size[A] + 1
i ← heap-size[A]
while i > 1 and A[Parent(i)] < key
    do A[i] ← A[Parent(i)]
        i ← Parent(i)
A[i] ← key
```

- Running time is $O(\lg n)$
  - Because the path from the last leaf to the root has length O(lgn)

Priority Queue Increase-Key Algorithm

Heap-Increase-Key(A,i, key)

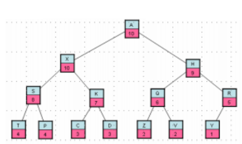
```
If key < A[i]
   then error "new key is smaller than the current key"
A[i] ← key
while i > 1 and A[Parent[i]] < A[i]
      do exchange A[i] ↔ A[Parent[i]]
           i ← Parent[i]
```


Priority Queue Extraction Procedure

- The *extraction algorithm* removes an element with highest priority
- It proceeds as follows
- Step # 1
  - Remove element at the root
- Step # 2
  - Move element in last position to the root
- Step # 3
  - Decrease the heap size by one
- Step # 4
  - Use HEAPFY operation to fix the new element at the root

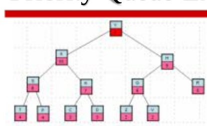
Priority Queue Extraction

- An element with highest priority is extracted from the queue
  - The last element in the heap is moved to the root
  - The resulting tree is converted into heap by fixing up procedure. The procedure is illustrated in figures (a)-(e)

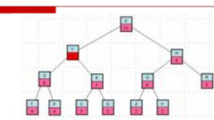


(a) Element 'A' with priority 10 is extracted from the queue

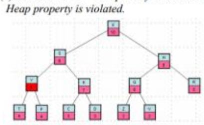
Priority Queue Extraction (Cont...)



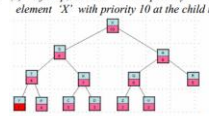
(b) Last element 'Y' with priority 1 is moved to root Heap property is violated.



(c) To fix up, element 'Y' with priority 1 is swapped with element 'X' with priority 10 at the child node



(d) Element 'Y' with priority 1 is swapped with element 'S' at the child node



(e) Element 'Y' with priority 1 is swapped with element 'T' at the child node. The heap property is now restored.

Priority Queue Extraction Algorithm

Heap-Extract-Max(A)

1. if *heap-size*[A] < 1
2. then error "heap underflow"
3. $max \leftarrow A[1]$
4. $A[1] \leftarrow A[heap\text{-}size[A]]$
5. $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
6. MaxHeapify(A, 1)
7. return max

- Running time is $O(\lg n)$
  - Dominated by the running time of MaxHeapify

End of Chapter

- Your may have quiz next week