Lecture 4: Priority Queues, Sorting, and Heaps

Pure Consciousness is a Field of Perfect Order

1

Wholeness Statement

The Priority Queue ADT stores any kind of object as a *key object* pair, but the *keys* must be objects that have a *total order relation (*or *linear ordering)*. *Science of Consciousness:* Each individual has access to the source of thought which is a field perfect order and balance. By opening our awareness to this field, we grow in the qualities of order and balance.

2

Overview

- Simple In-place Sorting
  - Selection sort
  - Insertion sort
- Priority Queue ADT
- Sorting with a Priority Queue
- Heap Data Structure

3

Analysis of Simple Sorting Algorithms

- SelectionSort and InsertionSort are among the simplest sorting methods and have straight forward analysis of running time. For each, we will consider best case, worst case, and average case running times.

4

Selection Sort Example

9	4	12	7	2
2	4	12	7	9
2	4	12	7	9
2	4	7	12	9
2	4	7	9	12

5

In-place SelectionSort

```
Algorithm SelectionSort (arr)
 Input Array arr
 Output elements in arr are in sorted order
    last ← arr.length - 1
    for i ← 0 to last  do
          nextMin ← findNextMinIndex(arr, i, last)
          swapElements(arr, i, nextMin)

//find index of minimum element between indices first and last
Algorithm findNextMinIndex(arr, first, last)
    min ← arr[first]
    minIndex ← first
   for i ← first + 1 to last do
         if arr[i] < min then
               min ← arr[i]
               minIndex ← i
   return  minIndex
```


6

Analysis of SelectionSort

- If n is the number of items in the array, there are n-1 comparisons on the first pass, n-2 on the second, and so on. The formula for the sum of such a series is: $(n–1) + (n–2) + (n–3) + ... + 1 = n*(n–1)/2$
  Thus, the algorithm makes $O(n^2)$ comparisons.

- With n items, SelectionSort performs no more than n swaps. For large values of n, the comparison times will dominate, so we have to say that the SelectionSort runs in $O(n^2)$ time.

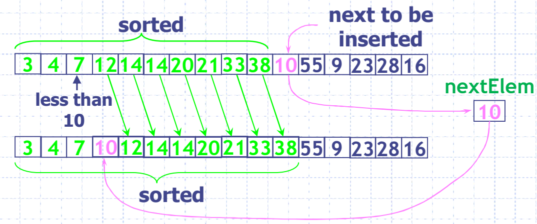
"Every-Case" Analysis

- Because there are two loops, nested, depending on n, it is $\Theta(n^2)$.
- Running time is the same for the best and worst cases.

Insertion Sort Example

9	4	12	7	2
4	9	12	7	2
4	9	12	7	2
4	7	9	12	2
2	4	7	9	12

Inner While-Loop of InsertionSort



- This one step, the inner while-loop, could make O(i) shifts in the worst case

In-place InsertionSort

Algorithm *InsertionSort(arr)*
Input Array *arr*
Output elements in *arr* are in sorted order

```
for i ← 1 to arr.length - 1 do
    j ← i
    nextElem ← arr[i]
    while j > 0 ∧ nextElem < arr[j - 1] do
        arr[j] ← arr[j - 1]      // shift array element to right
        j ← j – 1
    arr[j] ← nextElem             // place element in sorted position
```


Analysis of InsertionSort

- How many comparisons and copies does this algorithm require? On the first pass, it compares a maximum of one item. On the second pass, it's a maximum of two items, and so on, up to a maximum of n-1 comparisons on the last pass. This is $1 + 2 + 3 + … + n-1 = n*(n-1)/2$. However, because on each pass an average of only half of the maximum number of items are actually compared before the insertion point is found, we can divide by 2, which gives $n^2/4$ on average.
- The number of shifts is approximately the same as the number of comparisons. However, a shift/move operation isn't as expensive as a swap. In any case, like selection sort, the insertion sort runs in $O(n^2)$ time for random data.

Analysis of InsertionSort

- **Best-Case Analysis.** The best case for InsertionSort occurs when the input array is already sorted. In this case, the condition in the inner while loop always fails, so the code inside the loop never executes. The result is that execution time inside each outer loop is constant, and so running time is O(n).

13

13

Analysis of InsertionSort

- **Worst-Case Analysis.** Since there are two loops, nested, even in the worst case, the running time is only $O(n^2)$. The worst case for InsertionSort occurs when the input array is reverse-sorted. In this case, in pass #i of the outer for loop the inner while loop must execute all its statements i times approximately, and so execution time is proportional to $1+2+\ldots+n-1=O(n^2)$. Therefore, worst-case running time is $O(n^2)$.

14

14

Analysis of InsertionSort

- **Average-Case Analysis.** It is reasonable to expect that typically, the inner while loop will not work as hard as it does in the worst-case. As mentioned earlier, on average there are $n^2/4$ comparisons. So on average, InsertionSort runs in $O(n^2)$.

15

15

Comparing Performance of Simple Sorting Algorithms

- Swaps are more expensive than shifts/moves. Notice that swaps involve roughly eight primitive operations. This is more costly than shifting (which takes about four).
- Also, insertion sort does, on average, half as many key comparisons. Demos give empirical data for comparison.
- BubbleSort performs (on average) $O(n^2)$ swaps whereas SelectionSort performs only O(n) swaps, and InsertionSort does not perform any swaps at all (it shifts right which takes about half as much time as a swap). This difference explains why BubbleSort is so much slower than the other two. (Empirical studies show BubbleSort is 5 times slower than InsertionSort and 40% slower than SelectionSort and that InsertionSort is 3.5 times faster that SelectionSort on average.)

Simple Sorting 16

16

Main Point

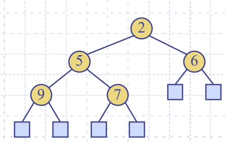
1. Using the tools of asymptotic analysis, we show that, in the worst case, SelectionSort and InsertionSort run in $O(n^2)$, so performance of both algorithms is about the same (i.e., asymptotically equivalent). A finer analysis computes the number of comparisons and swaps (SelectionSort) versus comparisons and shifts (InsertionSort) performed by each algorithm, and thus explains why, on average, InsertionSort is 3.5 times faster than SelectionSort.

Science of Consciousness: This analysis illustrates the principle that deeper levels of intelligence enable one to have greater insight and greater mastery over the more expressed values of life.

Simple Sorting 17

17

Priority Queues



Priority Queues & Sorting 18

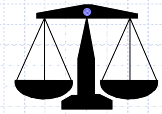
18

Priority Queue ADT (§ 2.4.1)

- A priority queue stores a collection of items
- An item is a pair (key, element)
- Main methods of the Priority Queue ADT
  - insertItem(k, e) inserts an item with key k and element e
  - removeMin() removes the item with smallest key and returns its element
- Additional methods
  - minKey() returns, but does not remove, the smallest key of an item
  - minElement() returns, but does not remove, the element of an item with smallest key
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

19

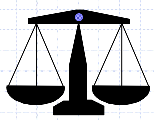
Total Order Relation



- Keys in a priority queue can be arbitrary objects on which a total order is defined
- Two distinct items in a priority queue can have the same key
- Also, called a *Linear Order*
- Mathematical concept of total order relation $\le$
  - **Reflexive** property: $x \le x$
  - **Antisymmetric** property: $x \le y \wedge y \le x \Rightarrow x = y$
  - **Transitive** property: $x \le y \wedge y \le z \Rightarrow x \le z$
  - **Totality** property: $x \le y \vee y \le x$

20

Partial Order Relation



- This is just to contrast the distinction between total order and partial order relations
- In a total order even equal elements are comparable
- Mathematical concept of partial order relation $<$
  - **Anti-Reflexive** property: $!\,(x < x)$, no element is related to itself, thus not every element is related (partial)
  - **Asymmetric** property: $x < y \Rightarrow !\,(y < x)$
  - **Transitive** property: $x < y \wedge y < z \Rightarrow x < z$

21

Comparator ADT (§ 2.4.1)

- A comparator encapsulates the action of comparing two objects according to a given total order relation
- A generic priority queue uses an auxiliary comparator
- The comparator is external to the keys being compared
- When the priority queue needs to compare two keys, it uses its comparator
- The comparator defines a total order on the keys
- Methods of the Comparator ADT, all with Boolean return type
  - isLessThan(x, y)
  - isLessThanOrEqualTo(x,y)
  - isEqualTo(x,y)
  - isGreaterThan(x, y)
  - isGreaterThanOrEqualTo(x,y)
  - isComparable(x)

22

Sorting with a Priority Queue (§ 2.4.2)

- We can use a priority queue to sort a set of comparable elements
  - Insert the elements one by one with a series of insertItem(e, e) operations
  - Remove the elements in sorted order with a series of removeMin() operations
- The running time of this sorting method depends on the priority queue implementation

Algorithm *PQ-Sort(S, C)*
  Input sequence S, comparator C for the elements of S
  Output sequence S sorted in increasing order according to C
  $P \leftarrow$ new priority queue using C
  while ! *S.isEmpty* () **do**
    $e \leftarrow$ *S.remove* (*S. first* ())
    P.insertItem(*e*, *e*)
  while ! *P.isEmpty*() **do**
    $e \leftarrow$ *P.removeMin*()
    S.insertLast(*e*)

23

Sequence-based Priority Queue

- Implementation with an unsorted list

  4 — 1 — 2 — 5 — 3

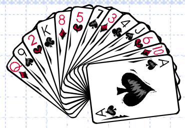
- Performance:
  - insertItem takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin, minKey and minElement take $O(n)$ time since we have to traverse the entire sequence to find the smallest key
- Implementation with a sorted list

  1 — 2 — 3 — 4 — 5

- Performance:
  - insertItem takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin, minKey and minElement take $O(1)$ time since the smallest key is at the beginning of the sequence

24

Selection-Sort



- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence

  4 — 5 — 2 — 3 — 1

- Running time of Selection-sort:
  - Inserting the elements into the priority queue with n insertItem operations takes $O(n)$ time
  - Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

    $$n + \ldots + 2 + 1$$
- Selection-sort runs in $O(n^2)$ time

Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence

  1 — 2 — 3 — 4 — 5

- Running time of Insertion-sort:
  - Inserting the elements into the priority queue with n insertItem operations takes time proportional to

    $$1 + 2 + \ldots + n$$
  - Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

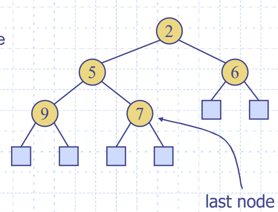
Main Point

2. Insertion sort starts with an initial list with one element, then inserts each new element such that the resulting sequence is also in order. Selection sort selects the smallest element each iteration from an unsorted list and inserts it at the end of the target list. Neither of these algorithms is optimal.
   Science of Consciousness: In contrast, pure intelligence always follows the optimal law of least action. By opening our awareness to pure intelligence, we grow in the qualities of efficiency and spontaneous right action.

The Heap Data Structure

What is a heap (§2.4.3)

- A heap is a binary tree storing keys at its internal nodes and satisfying the following properties:
  - Heap-Order: for every internal node v other than the root, $key(v) \geq key(parent(v))$
  - Complete Binary Tree: let h be the height of the heap
    - for $i = 0, \ldots, h - 1$, there are 2^i nodes of depth i
    - at depth $h - 1$, the internal nodes are to the left of the external nodes
- The last node of a heap is the rightmost internal node of depth $h - 1$



Heap-Order Property

- For all internal nodes v (except the root):

  $$key(v) \geq key(parent(v))$$

  - That is, the key of every child node is greater than or equal to the key of its parent node

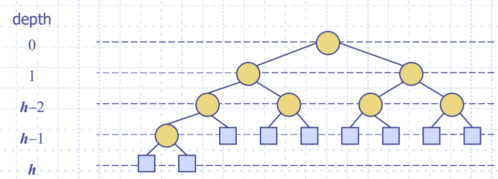
Other Properties of a Heap

- A heap is a binary tree whose values are in ascending order on every path from root to leaf
- Values are stored in internal nodes only
- A heap is a binary tree whose root contains the minimum value and whose subtrees are heaps

31

Heap

- All leaves of the tree are on two adjacent levels
- The binary tree is complete on every level except the deepest level.



32

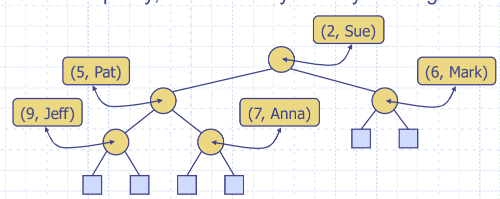
Adding Nodes to a Heap

- New nodes must be added left to right at the lowest level, i.e., the level containing internal and external nodes or containing all external nodes

33

Heaps and Priority Queues

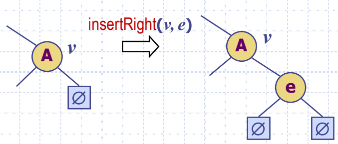
- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node
- For simplicity, we show only the keys in diagrams



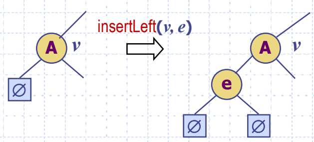
34

Insert Methods

insertRight(v, e)
Right child must be external.



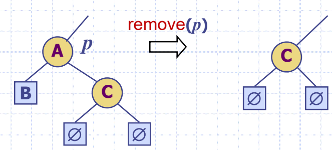
insertLeft(v, e)
Left child must be external.

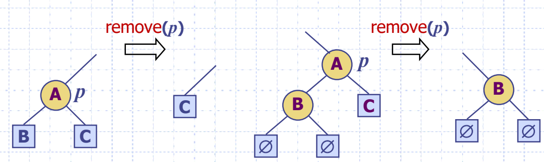


35

Remove Method

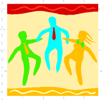
remove(p)
Either the left or right child must be external!
We can only remove the node above an external node.



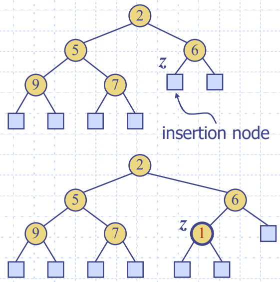


36

Insertion into a Heap (§2.4.3)



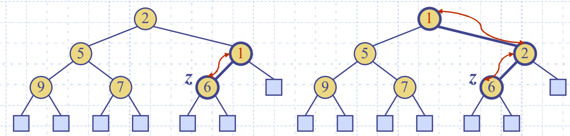
- Method insertItem of the priority queue ADT corresponds to the insertion of a key k into the heap
- The insertion algorithm consists of three steps
  1. Find the insertion node z (the new last node)
  2. Store k at z and expand z into an internal node
  3. Restore the heap-order property



37

Upheap

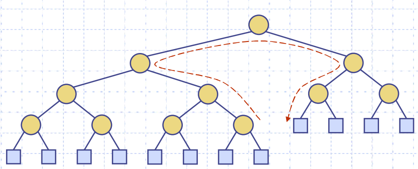
- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time



38

Finding the Insertion Node and Updating the Last Node

- The insertion node can be found by traversing a path of $O(\log n)$ nodes
  - While the current node is a right child, go to the parent node
  - If the current node is a left child, go to the right child
  - While the current node is internal, go to the left child
- Similar algorithm for updating the last node after a removal



39

Exercise

- Insert the following keys into a heap represented as a Tree:
  9, 6, 5, 14, 4, 12, 15, 3, 2

40

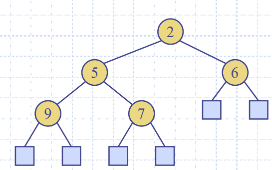
Efficient Representation of A Heap

Use an Array, Vector, or Sequence with efficient random access

41

Vector (or Array) based Heap Implementation (§2.4.3)

- We can represent a heap with n keys by means of a vector of length $n + 1$
- For the node at rank i
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$
- Links between nodes are not explicitly stored
- The leaves are not represented
- The cell at rank 0 is not used
- Operation insertItem corresponds to inserting at rank $n + 1$
- Operation removeMin corresponds to removing at rank n
- Yields in-place heap-sort



	2	5	6	9	7
0	1	2	3	4	5

42

Exercise

- Insert the following keys into a heap represented as a Vector/Array:
  9, 6, 5, 14, 4, 12, 15, 3, 2

43

Implementation of upHeap

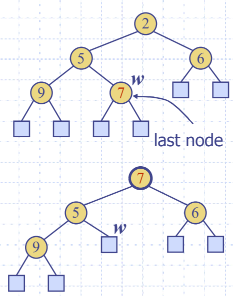
```
Algorithm upHeap(H, i)
  Input Array H representing a heap and index i of an element in the heap
  Output H with the heap property restored

  parent ← i / 2
  if 1 ≤ parent ∧ H[parent] > H[i] then
     temp ← H[parent]
     H[parent] ← H[i]          {swap elements}
     H[i] ← temp
     upHeap(H, parent)
```


44

Removal from a Heap (§2.4.3)

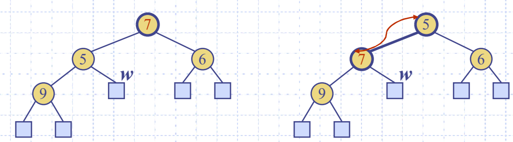
- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Compress w and its children into a leaf
  - Restore the heap-order property



45

Downheap

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time



46

Exercise:

- Write the pseudocode for downHeap
  - You can have any interface you wish
  - You will need an extra argument, i.e., the size of the heap (Why?)
  - The interface for a recursive version of upHeap was upHeap(H, i)
    - So the interface of a recursive version would be downHeap(H, i, size)
    - An iterative version would have interface downHeap(H, size)

47

Recursive Version

```
Algorithm downHeap(H, size, r)
  Input Array H representing a heap, rank r of an element in H,
        and the size of the heap H
  Output H with the heap property restored

  smallest ← rankOfMin(H, size, r)      {min of r and its children}
  if smallest ≠ r then
     temp ← H[smallest]
     H[smallest] ← H[r]                 {swap elements}
     H[r] ← temp
     downHeap(H, size, smallest)
```


48

Helper for downHeap Algorithm

```
Algorithm rankOfMin(A, size, r)
    Input array A, a rank r (containing an element of A), and size of the
          heap stored in A
    Output the rank of element in A containing the smallest value

    smallest ← r
    left ← 2*r
    right ← left + 1
    if left ≤ size ∧ A[left] < A[smallest] then
        smallest ← left
    if right ≤ size ∧ A[right] < A[smallest] then
        smallest ← right
    return smallest
```


Iterative Version

```
Algorithm downHeap(H, size)
    Input Array H representing a heap and the size of H (size ≥ 1)
    Output H with the heap property restored
    property ← false
    i ← 1
   while property = false do
       smallest ← rankOfMin(H, i, size)
       if smallest ≠ i then
           temp ← H[smallest]
           H[smallest] ← H[i]              {swap elements}
           H[i] ← temp
           i ← smallest
       else
           property ← true

    return H
```


Analysis of Heap Operations

- Upheap()
- Downheap()

Analysis of Heap-based Priority Queue

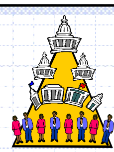
- insertItem(k, e)
- removeMin()
- minKey()
- minElement()
- size()
- isEmpty()

Analysis of Sorting with a Heap-based Priority Queue

- What is the running time of this sorting method if the priority queue is implemented as a Heap?

```
Algorithm PQ-Sort(S, C)
    Input sequence S, comparator C
    for the elements of S
    Output sequence S sorted in
    increasing order according to C
    P ← priority queue with
            comparator C
    while ! S.isEmpty () do
        e ← S.remove (S. first ())
        P.insertItem(e, e)
    while ! P.isEmpty() do
        e ← P.removeMin()
        S.insertLast(e)
```

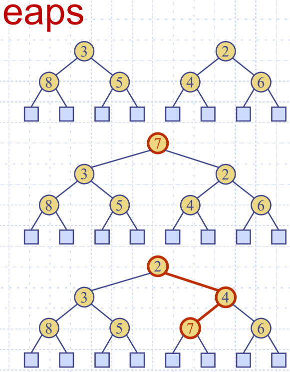

Analysis of Heap-Based Priority Queue (§2.4.4)



- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods insertItem and removeMin take $O(\log n)$ time
  - methods size, isEmpty, minKey, and minElement take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

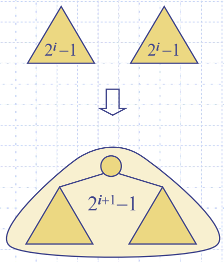
Merging Two Heaps

- We are given two heaps and a key k
- We create a new heap with the root node storing k and with the two heaps as subtrees
- We call downHeap to restore the heap-order property



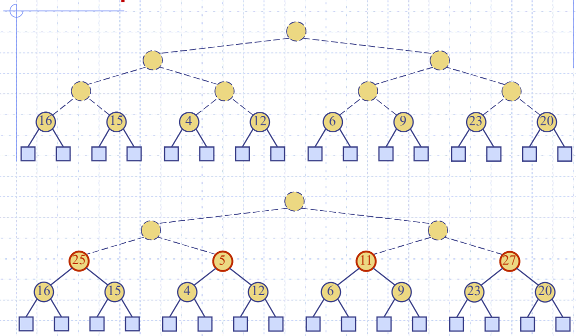
Bottom-up Heap Construction (§2.4.4)

- We can construct a heap storing n given keys using a bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with 2^i-1 keys are merged into heaps with $2^{i+1}-1$ keys



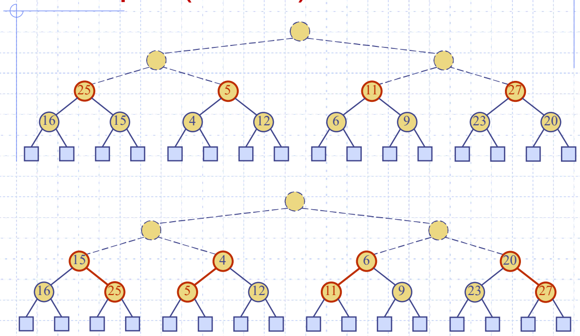
55

Example



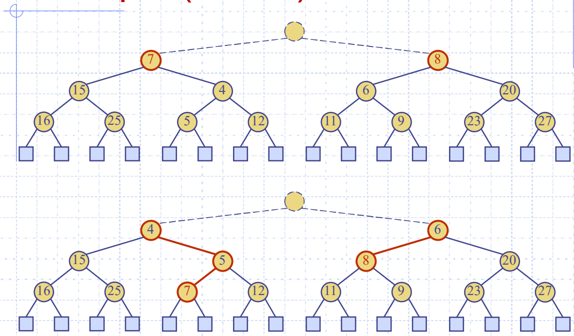
56

Example (contd.)



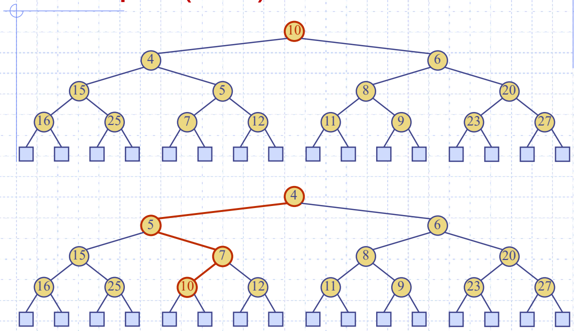
57

Example (contd.)



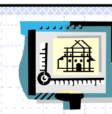
58

Example (end)

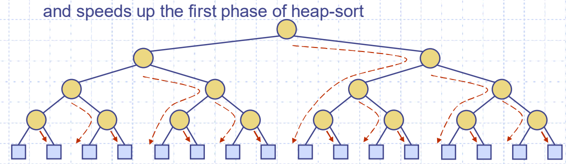


59

Analysis of Bottom-up Heap Construction



- We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path)
- Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy paths is $O(n)$
- Thus, bottom-up heap construction runs in $O(n)$ time
- Bottom-up heap construction is faster than n successive insertions and speeds up the first phase of heap-sort



60

HeapSort

Algorithm *heapsort*(arr)
Input Array *arr*
Output elements in *arr* are in sorted order

```
    buildHeap (arr)  // build the heap from the bottom up O(n)

    end ← arr.length-1
    while end > 0 do
        swapElements(arr, 0, end) // move max to end of heap
        end ← end - 1  // decrease size of the heap
        downHeap(arr, 0, end) // restore heap-order O(nlog n)

Algorithm swapElements(H, j, k)
    temp ← H[j]
    H[j] ← H[k]
    H[k] ← temp
```


Build the Heap from bottom

Algorithm *buildHeap*(arr)
Input Array *arr*
Output *arr* is a heap built from the bottom up in O(n) time with the root at index 0 (instead of 1)

```
    last ← arr.length-1;
    next ← last;
    while (next > 0) do
        downHeap(arr, last, parent(next));
        next ← next - 2;

Algorithm parent(i)
    return floor((i – 1)/ 2)
```


Iterative Version of downHeap

Algorithm *downHeap*(*H, last, i*)
Input Array *H* containing a heap and *last* (index of last element of *H*)
Output H with the heap order property restored

```
property ← false
while ! property do
    maxIndex ← indexOfMax(H, i, last)  // returns i or one of its children
    if maxIndex ≠ i then
        swapElements(H, maxIndex, i)  // swaps larger to parent i
        i ← maxIndex    // move down the tree/heap to max child
    else
        property ← true
```


Helper for downHeap Algorithm

Algorithm *indexOfMax*(*A, r, last*)
Input array *A*, an index *r* (referencing an element of A), and *last*, the index of the last element of the heap stored in *A*
Output index of element in *A* containing the largest of *r* or *r*'s children

```
largest ← r
left ← 2*r +1
right ← left + 1
if left ≤ last ∧ A[left] > A[largest] then
    largest ← left
if right ≤ last ∧ A[right] > A[largest] then
    largest ← right
return largest
```


Main Point

2. A heap is a binary tree that stores *key object* pairs at each internal node and maintains *heap-order* and is *complete*. Heap-order means that for every node *v* (except the root), *key(v)≥key(parent(v))*. Pure consciousness is the field of wholeness, perfectly orderly, and complete.

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	◆ slow ◆ in-place ◆ for small data sets (< 1K)
insertion-sort	$O(n^2)$	◆ slow ◆ in-place ◆ for small data sets (< 1K)
heap-sort	$O(n \log n)$	◆ fast ◆ in-place ◆ for large data sets (1K — 1M)
PQ-sort	$O(n \log n)$	◆ fast ◆ NOT in-place, but is simple ◆ for large data sets (1K — 1M)

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. Sorting with a Priority Queue is a simple process of inserting the elements in the queue and removing them using the *removeMin* operation.

2. How the Priority Queue is implemented determines its efficiency when used in a sort, i.e., if implemented as a Heap, then the sorting algorithm is optimal, $O(n \log n)$.

3. Transcendental Consciousness is the unbounded field of pure order and efficiency.
4. **Impulses within Transcendental Consciousness**: The laws of nature are non-changing and universal which provide a reliable basis for the integrity of the universe.
5. **Wholeness moving within itself** : In Unity Consciousness, life is spontaneously lived in accord with natural law for maximum achievement with minimum effort.