**IDEAS**

* start with an activity - try and solve a problem (get them to touch upon a heap by themselves)
* PowerPoint (diagrams, charts)
* Draw diagrams on whiteboard
* Socrative QUESTIONS:
  + To wrap up

TO DO

|  |  |
| --- | --- |
| ALL | * finish research |
| ALL | * everyone understand what a heap is * watch videos:   + <https://www.youtube.com/watch?v=c1TpLRyQJ4w>   + Part 1, 2, 3, etc. |
| JORDAN | * prepare implementation - with comments, be able to explain it   + Operations you can perform on the heap |
| IOANA | * create lesson plan   + Components in powerpoint |
| JEMIMA | * make powerpoint   + Ensure that the components on the rubric are included |
| IOANA | * create Socrative questions/quiz - prize for winner (glory) |
|  |  |

**WHAT IS A HEAP DATA STRUCTURE**

**(How are the pieces of information organized (structure)?)**

**TREE** - hierarchy. The definition of a tree is recursive. A tree may contain no nodes, in which case it is an empty tree, or it contains a node and 0 or more links to subordinate trees (subtrees). These subtrees are trees themselves. This means that if you have an algorithm that operates on a tree, you can use the same algorithm to operate on the subtrees pointed to by the root’s links.

Because a tree’s constructor includes its 2 children, it is easiest to build the tree from the bottom up.

**NODES** - the individual elements of the tree. Each node contains data and 0 or more links to other nodes

**ROOT** - topmost node in the hierarchy -> MUST EITHER BE THE HIGHEST OR THE LOWEST VALUE

**CHILD NODE** - a node that is subordinate to another node

Efficiency of heaps:

Whenever you work with a heap, most of the time taken by the algorithm will be spent upheaping and downheaping. As it happens, the max number of levels of a complete tree is log(n) + 1, n is the # of nodes in the tree. Adding or removing from a heap is O(log n), as you can make the swaps log(n) times, or one less time than the number of levels in the tree.

HEAP:

* region of computer’s memory
* free-floating region of memory, not as tightly managed
* to allocate memory on heap,
* must use: malloc() or calloc()
* once memory has been allocated, responsible for using free()
* no size restrictions (unlike stacks)
* slightly slower to be read from & written to : must use pointers to access memory
* unlike stack: variable created on heap can be accessed by any function, anywhere in program
* Use it whenever you need quick access to the largest/smallest item, because that item will always be the first element in the array or at the root of the tree.
* Can implement a heap as a tree or an array
* BUILDING a heap:
  + Start with top node, fill in top to bottom, left to right.
  + Don’t move on to row below until row above is completely full

Classification of heaps:

MAX HEAP:

* The values/keys in each internal node are greater than or equal to the values in the children and the lowest value of that node.
* In other words: PARENT is always greater than CHILD
* WILL ALWAYS HAVE GREATEST VALUE AT THE ROOT (top)

MIN HEAP:

* The values/keys in each internal node are less than or equal to the values in the children and the highest value of that node
* In other words: PARENT is always less than CHILD
* WILL ALWAYS HAVE SMALLEST VALUE AT THE ROOT (top)
* What happens when you need to add an item that violates this rule?
  + Normally you have to add in the leftmost available spot
  + When you add a new item, it compares the item to its parent and runs a check if it satisfies the max/min rule (P>C or P<C)

Reference: <http://gribblelab.org/CBootcamp/7_Memory_Stack_vs_Heap.html#sec-3>

**WHAT IS THE PURPOSE OF THIS DATA STRUCTURE?**

**OPERATIONS YOU CAN PERFORM ON A HEAP**

**(What are the functions (services) of your data structure and how do they work?)**

Upheap: The upheap process is used to add a node to a heap. When you upheap a node, you compare its value to its parent node; if its value is less than its parent node, then you switch the two nodes and continue the process.

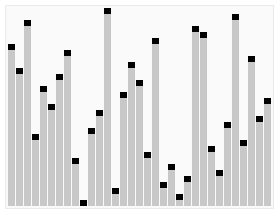
Downheap: The downheap process is similar to the upheaping process. When you downheap a node, you compare its value with its two children. If the node is less than both of its children, it remains in place; otherwise, if it is greater than one or both of its children, then you switch it with the child of lowest value, thereby ensuring that of the three nodes being compared, the new parent node is lowest.

Heap Sort:

<http://codereview.stackexchange.com/questions/32606/implementation-of-heap-sort>

Implementation Video:

<https://www.youtube.com/watch?v=LhhRbRXhB40>



Max Heap: <http://www.sanfoundry.com/java-program-implement-max-heap/>

**EXAMPLE PROGRAM OF A HEAP**

**OTHER SCENARIOS WHERE YOU MIGHT USE A HEAP**

**(What are the practical uses of this data structure (aka how are they an improvement on basic variables and arrays)?)**

HEAP SORT

* Comparison-based sorting algorithm
* An improved selection sort
* Divides the input into sorted and unsorted
* Iteratively shrinks the unsorted by taking the largest elements and moving it to the sorted elements
* IMPROVEMENT: uses the heap data structure rather than a linear-time search to find the maximum
* Not a stable sort

**HEAP SORT CODE:**

1. ***/\****
2. ***\* Java Program to Implement Heap Sort***
3. ***\*/***
5. **import java.util.Scanner;**
7. ***/\* Class HeapSort \*/***
8. **public class HeapSort**
9. **{**
10. **private static int N;**
11. ***/\* Sort Function \*/***
12. **public static void sort(int arr[])**
13. **{**
14. **heapify(arr);**
15. **for (int i = N; i > 0; i--)**
16. **{**
17. **swap(arr,0, i);**
18. **N = N-1;**
19. **maxheap(arr, 0);**
20. **}**
21. **}**
22. ***/\* Function to build a heap \*/***
23. **public static void heapify(int arr[])**
24. **{**
25. **N = arr.length-1;**
26. **for (int i = N/2; i >= 0; i--)**
27. **maxheap(arr, i);**
28. **}**
29. ***/\* Function to swap largest element in heap \*/***
30. **public static void maxheap(int arr[], int i)**
31. **{**
32. **int left = 2\*i ;**
33. **int right = 2\*i + 1;**
34. **int max = i;**
35. **if (left <= N && arr[left] > arr[i])**
36. **max = left;**
37. **if (right <= N && arr[right] > arr[max])**
38. **max = right;**
40. **if (max != i)**
41. **{**
42. **swap(arr, i, max);**
43. **maxheap(arr, max);**
44. **}**
45. **}**
46. ***/\* Function to swap two numbers in an array \*/***
47. **public static void swap(int arr[], int i, int j)**
48. **{**
49. **int tmp = arr[i];**
50. **arr[i] = arr[j];**
51. **arr[j] = tmp;**
52. **}**
53. ***/\* Main method \*/***
54. **public static void main(String[] args)**
55. **{**
56. **Scanner scan = new Scanner( System.in );**
57. **System.out.println("Heap Sort Test\n");**
58. **int n, i;**
59. ***/\* Accept number of elements \*/***
60. **System.out.println("Enter number of integer elements");**
61. **n = scan.nextInt();**
62. ***/\* Make array of n elements \*/***
63. **int arr[] = new int[ n ];**
64. ***/\* Accept elements \*/***
65. **System.out.println("\nEnter "+ n +" integer elements");**
66. **for (i = 0; i < n; i++)**
67. **arr[i] = scan.nextInt();**
68. ***/\* Call method sort \*/***
69. **sort(arr);**
70. ***/\* Print sorted Array \*/***
71. **System.out.println("\nElements after sorting ");**
72. **for (i = 0; i < n; i++)**
73. **System.out.print(arr[i]+" ");**
74. **System.out.println();**
75. **}**
76. **}**