Trees 1: introduction to Binary Trees & Heaps
Basic terminology

- Finite set of nodes (may be empty -- 0 nodes), which contain data
- First node in tree is called the root
Basic terminology

- Each node may be linked to 0, 1 or 2 child nodes (or children)
- A node with children is a parent; a node with no children is a leaf

Nodes d and e are children of b.
Node c is parent of f and g.
More botany & sociology

- The root node has no parent; it is the **ancestor** of all other nodes.
- All nodes in the tree are **descendants** of the root node.
- Nodes with the same parent are **siblings**.
- Any node with descendants (any parent node) is the root node of a **subtree**.
But wait, there’s more!

- Tree **depth** is the maximum number of steps through ancestor nodes from the deepest leaf back to root
  - tree with single node (root) has depth of 0
  - empty tree has depth of -1

Depth of entire tree is 2; depth of b’s subtree is 1; depth of f’s subtree is 0
Tree shape terminology

- **Full** binary tree
  - every parent has 2 children,
  - every leaf has equal depth
Tree shape terminology

- **Complete** binary tree
  - every level is full except possibly the deepest level
  - if the deepest level isn’t full, leaf nodes are as far to the left as possible
Some examples

Complete binary tree

Neither

Full and complete

Both

Complete

Neither full nor complete
Array representation of a binary tree

- Complete binary tree is easily represented as an array
- Root data is stored at index 0
- Root’s children are stored at index 1 and 2
Array representation of a binary tree

• In general:
  – root is tree[0]
  – for any node at tree[n], the parent node will be found at index \((n-1)/2\)
  – for any node at tree[n], the children of n (if any) will be found at tree[2n + 1] and tree[2n + 2]
Array representation of a binary tree

• Since these formulas apply to any complete tree represented as an array, can create a tree class with 2 private members:
  – an array that stores the data
  – a counter to keep track of the number of nodes in the tree

• Relationships between nodes can always be determined from the given formulas
Array representation of a binary tree

- What if tree isn’t complete?
  - Can still use array, but problem is more complicated -- have to keep track of which children actually exist
  - One possibility would be to place a known dummy value in empty slots
  - Another is to add another private member to the class -- an array of booleans paired to the data array -- index value is true if data exists, false if not
A heap is a binary tree in which the data entries can be compared using total order semantics.

Defining qualities of a heap:
- The data entry contained in any node is greater than or equal to the data entry contained in either of its children.
- The tree is always complete.
Heap Example

45

32 41

19 7 28 40

10 12
Heap Applications

• Heap can be used to implement a sorting algorithm called heapsort
• Heap is also a handy way to implement a priority queue -- we’ll use this application to illustrate how heaps work
Heap Implementation

• Because a heap is by definition a complete tree, it is easy to use an array-based implementation.

• Heap operations \texttt{reHeapUp} and \texttt{reHeapDown} are used to maintain the other defining quality of a heap -- each node’s data entry \(\geq\) data entries of its children.
ReHeapUp operation

When a new node is added to the tree, it is always added at the leftmost open position in the bottom row.

This maintains completeness of tree, but can violate the other defining characteristic of a heap.
ReHeapUp operation

ReHeapUp restores the second heap condition: a parent node’s data is always greater than or equal to the data in either of its child nodes.

The operation is accomplished by swapping the new node’s data entry with that of its parent.

Parent/child data swapping continues until the heap condition is restored.
In a priority queue, the highest-priority item is always dequeued first -- this item would be the top item of the heap.

Since this is going to be an array implementation, we’ll perform the usual trick of swapping the last array entry with the first, so we can minimize the amount of copying to be done.
Now we can effectively remove the item by simply diminishing the “used” portion of our array by 1 (already done here)

We are once again faced with the same problem -- the heap is the right shape, but the data values are in the wrong positions.

We solve this problem, as before, by swapping data between nodes. In this case, we swap the parent node’s data with the largest data entry of the two children, continuing until the heap is restored.
Example: Priority Queue with heap implementation

- Priority queue has all operations associated with a queue: enqueue, dequeue and helper functions
- Heap is a good fit for priority queue because highest-priority item should always be dequeued first, and this item would always be found at the top of a heap
Code for Priority Queue

public class PQueue {
    Heap queue;

    public PQueue (int cap) {
        queue = new Heap(cap);
    }

    public void enqueue (int item) {
        queue.addEntry(item);
    }

    public int dequeue () {
        return queue.getTop();
    }
}
Code for Heap class

public class Heap {
    int [] data;
    int numItems;

    public Heap (int size) {
        numItems = 0;
        data = new int [size];
    }
}
Code for Heap class

```java
public int parent (int n) {
    return (n-1)/2;
}

public int leftChild (int n) {
    return 2*n + 1;
}

public int rightChild (int n) {
    return 2*n + 2;
}
```
Code for Heap class

```java
public void addEntry (int entry) {
    if (numItems < data.length) {
        data[numItems] = entry;
        reHeapUp(numItems);
        numItems++;
    }
}
```
public void reHeapUp(int n) {
    int x = n;
    while (x > 0 && data[x] > data[parent(x)]) {
        int tmp = data[x];
        data[x] = data[parent(x)];
        data[parent(x)] = tmp;
        x = parent(x);
    }
}
public int getTop () {
    int value=data[0]; // save return value
    numItems--;
    data[0]=data[numItems];  // swap top & bottom
    reHeapDown(numItems); // restore heap
    return value;
}
public void reHeapDown (int n) {
    int current = 0, bigChild;
    boolean heapOK = false;
    while (!heapOK && (leftChild(current) < n))
    {
        if (rightChild(current) >= n)
            bigChild = leftChild(current);
        else if (data[leftChild(current)] > data[rightChild(current)])
            bigChild = leftChild(current);
        // continued next slide
    }
Code for Heap class

else
    bigChild = rightChild(current);
if (data[current] < data[bigChild]) {
    int tmp = data[current];
    data[current] = data[bigChild];
    data[bigChild] = tmp;
    current = bigChild;
}
else
    heapOK = true;
} // end of while loop
} // end of method
} // end of class
A good sorting algorithm is hard to find ...

- Quadratic sorting algorithms (with running times of $O(N^2)$, such as Selectionsort & Insertionsort are unacceptable for sorting large data sets
- Mergesort and Quicksort are both better alternatives, but each has its own set of problems
A good sorting algorithm is hard to find ...

• Both Mergesort and Quicksort have average running times of $O(N \log N)$, but:
  – Quicksort’s worst-case running time is quadratic
  – Mergesort uses dynamic memory and may fail for large data sets because not enough memory is available to complete the sort
Heapsort to the rescue!

- Combines the time efficiency of Mergesort with the storage efficiency of Quicksort
- Uses an element interchange algorithm similar to Selectionsort
- Works by transforming the array to be sorted into a heap
How Heapsort Works

• Begin with an array of values to be sorted
• Heapsort algorithm treats the array as if it were a complete binary tree
• Values are rearranged so that the other heap condition is met: each parent node is greater than or equal to its children
## Heapsort in Action

<table>
<thead>
<tr>
<th>42</th>
<th>19</th>
<th>33</th>
<th>8</th>
<th>12</th>
<th>97</th>
<th>54</th>
<th>85</th>
<th>29</th>
<th>60</th>
<th>26</th>
<th>71</th>
</tr>
</thead>
</table>

### Original array

![Binary tree diagram](image)

- **42**
  - **19**
    - **8**
    - **29**
    - **60**
    - **26**
    - **71**
  - **33**
    - **97**
    - **54**

... arranged as binary tree (but not yet heap)
Heapsort in Action

97  85  71  29  60  42  54  8  19  12  26  33

... rearranged as heap
Heapsort in Action

The goal of the sort algorithm is to arrange entries in order, smallest to largest. Since the root element is by definition the largest element in a heap, that largest element can be placed in its final position by simply swapping it with whatever element happens to be in the last position:

| 33 | 85 | 71 | 29 | 60 | 42 | 54 | 8 | 19 | 12 | 26 | 97 |

As a result, the shaded area can now be designated the sorted side of the array, and the unsorted side is almost, but not quite, a heap.
Heapsort in Action

The next step is to restore the heap condition by rearranging the nodes again; the root value is swapped with its largest child, and may be swapped further down until the heap condition is restored:

\[
\begin{array}{cccccccccccc}
85 & 60 & 71 & 29 & 33 & 42 & 54 & 8 & 19 & 12 & 26 & 97 \\
\end{array}
\]

Once the heap is restored, the previous operation is then repeated: the root value, which is the largest on the unsorted side, is swapped with the last value on the unsorted side:

\[
\begin{array}{cccccccccccc}
26 & 60 & 71 & 29 & 33 & 42 & 54 & 8 & 19 & 12 & 85 & 97 \\
\end{array}
\]
Heapsort in Action

Again, the unsorted side is almost a heap, and must be restored to heap condition - this is accomplished the same way as before:

The process continues: root is swapped with the last value and marked sorted; the unsorted portion is restored to heap condition.
Heapsort in Action

And so on ...

54  33  42  29  19  12  26  8  60  71  85  97

And so forth ...

42  33  26  29  19  12  8  54  60  71  85  97

Etc. ...

8  12  19  26  29  33  42  54  60  71  85  97

Tada!
Pseudocode for Heapsort

• Convert array of n elements into heap
• Set index of unsorted to n (unsorted = n)
• while (unsorted > 1)
  unsorted--;
  swap (array[0], array[unsorted]);
  reHeapDown
makeHeap function

- Two common ways to build initial heap
- First method builds heap one element at a time, starting at front of array
- Uses reHeapUp process (last seen in priority queue implementation) to enforce heap condition - children are $\leq$ parents
Alternative makeHeap method

• Uses a function (subHeap) that creates a heap from a subtree of the complete binary tree
• The function takes three arguments: an array of items to be sorted, the size of the array (n), and a number representing the index of a subtree.
• Function subHeap is called within a loop in the makeHeap function, as follows:

```java
for (int x = (n/2); x>0; x--)
    subHeap(data, x-1, n);
```
Alternative makeHeap in Action

| 42 | 19 | 33 | 8  | 12 | 97 | 54 | 85 | 29 | 60 | 26 | 71 |

Original array

First call to subHeap, in the given example:
subHeap (data, 5, 12);

Algorithm examines subtree to determine if any change is necessary; since the only child of element 5 (97) is element 11 (71), no change from this call.
Second call to subHeap

The second call to subHeap looks at the subtree with root index 4.

The children of this node are found at indexes 9 and 10.

Both children are greater than the root node, so the root node’s data entry is swapped with the data entry of the larger child.
Third call to subHeap

Continuing to work backward through the array, the next root index is 3

Again, the children of this node are examined, and the root data entry is swapped with the larger of its two children
The process continues ...

In this case, a further swap is necessary because the data entry’s new position has a child with a larger data entry
The process continues ... 

| 42 | 85 | 97 | 29 | 60 | 71 | 54 | 8 | 19 | 12 | 26 | 33 |

Again, a further swap operation is necessary
Last call to subHeap

We now have a heap:

![Heap Diagram]
ReHeapDown function

• Both the original HeapSort function and the subHeap function will call reHeapDown to rearrange the array

• The reHeapDown method swaps an out-of-place data entry with the larger of its children
Time analysis for HeapSort

- The time required for a HeapSort is the sum of the time required for the major operations:
  - building the initial heap: $O(N \log N)$
  - removing items from the heap: $O(N \log N)$

- So total time required is $O(2N \log N)$, but since constants don’t count, the worst-case (and average case) for HeapSort is $O(N \log N)$