# 16 Trees, Heaps, and Heapsort

## 16.1 Introduction

Trees are the basis for several important data types and data structures. There are also several sorting algorithms based on trees. One of these algorithms is heapsort, which uses a complete binary tree represented in an array for fast in-place sorting.

## 16.2 Basic Terminology

A trees is a special type of graph.

> **Graph**: A collection of *vertices*, or *nodes*, and *edges* connecting the vertices. An edge may be thought of as a pair of vertices. Formally, a graph is an ordered pair <*V*,*E*> where *V* is a set of vertices, and *E* is a set of pairs of elements of *V*.
>
> **Simple path**: A list of distinct vertices such that successive vertices are connected by edges.
>
> **Tree**: A graph with a distinguished vertex *r*, called the *root*, such that there is exactly one simple path between each vertex in the tree and *r*.

We usually draw trees with the root at the top and the vertices and edges descending below. Figure 1 illustrates a tree.

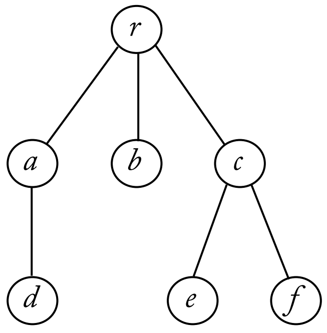

**Figure 1:** A Tree

Vertex *r* is the root. The root has three *children*: *a*, *b*, and *c*. The root is the *parent* of these vertices. These vertices are also *siblings* of one another because they have the same parent Vertex *a* has child *d* and vertex *c* has children *e* and *f*. The ancestors of a vertex are the vertices on the path between it and the root; the *descendents* of a vertex are all the vertices of which it is an ancestor. Thus vertex *f* has ancestors *f*, *c*, and *r*, and *c* has descendents *c*, *e*, and *f*. A vertex without children is a *terminal vertex* or a *leaf*; those with children are *non-terminal* vertices or *internal* vertices. The tree in Figure 1 has three internal vertices (*r*, *a*, and *c*) and four leaf vertices (*b*, *d*, *e*, and *f*). The graph consisting of a vertex in a tree, all its descendents, and the edges connecting them, is a *sub-tree* of the tree.

A graph consisting of several trees is a *forest*. The *level* of a vertex in a tree is the number of vertices in the path from the vertex to the root, not including itself. In Figure 1, vertex *r* is at level zero, vertices *a*, *b*, and *c* are at level one, and vertices *d*, *e*, and *f* are at level two. The *height* of a tree is the maximum level in the tree. The height of the tree in Figure 1 is two.

An *ordered tree* is one in which the order of the children of each vertex is specified. Ordered trees are not drawn in any special way—some other mechanism must be used to specify whether a tree is ordered.

## 16.3 Binary Trees

Binary trees are especially important for making data structures.

> **Binary tree**: An ordered tree whose vertices have at most two children. The children are distinguished as the *left child* and *right child*. The sub-tree whose root is the left (right) child of a vertex is the *left (right) sub-tree* of that vertex.

A **full binary tree** is one in which every level is full except possibly the last. A **complete binary tree** is a full binary tree in which only the right-most vertices at the bottom level are missing. Figure 2 illustrates these notions.

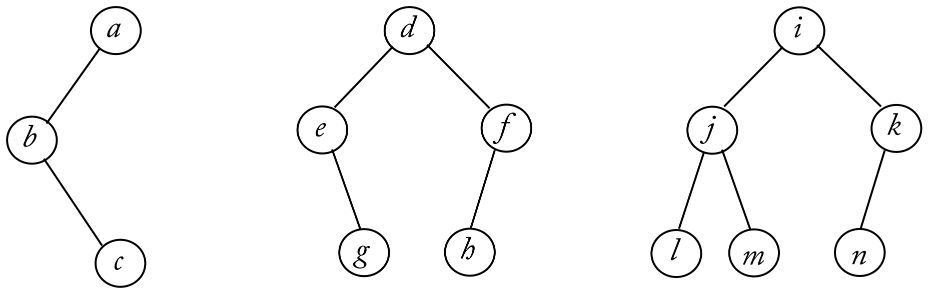

**Figure 2:** Binary Trees

The trees in Figure 2 are binary trees. In the tree on the left, vertex *a* has a left child, vertex *b* has a right child, and vertex *c* has no children. This tree is neither full nor complete. The middle tree is full but not complete, and the right tree is complete.

Trees have several interesting and important properties, the following among them.

- A tree with *n* vertices has *n*-1 edges.
- A complete binary tree with *n* internal vertices has either *n* or *n*+1 leaves.
- The height of a full binary tree with *n* vertices is *floor*(lg *n*).

## 16.4 Heaps

A vertex in a binary tree has the *heap-order property* if the value stored at the vertex is greater than or equal to the values stored at its descendents.

> **Heap**: A complete binary tree whose every vertex has the heap-order property.

An arbitrary complete binary tree can be made into a heap as follows:

- Every leaf already has the heap-order property, so the sub-trees whose roots are leaves are heaps.
- Starting with the right-most internal vertex *v* at the next-to-last level, and working left across levels and upwards in the tree, do the following: if vertex *v* does not have the heap-order property, swap its value with the largest of its children, then do the same with the modified vertex, until the sub-tree rooted at *v* is a heap.

It is fairly efficient to make complete binary trees into heaps because each sub-tree is made into a heap by swapping its root downwards in the tree as far as necessary. The height of a complete binary tree is *floor*(lg *n*), so this operation cannot take very long.

Heaps can be implemented in a variety of ways, but the fact that they are complete binary trees makes it possible to store them very efficiently in contiguous memory locations. Consider the numbers assigned to the vertices of the complete binary tree in Figure 3. Note that numbers are assigned left to right across levels, and from top to bottom of the tree.\

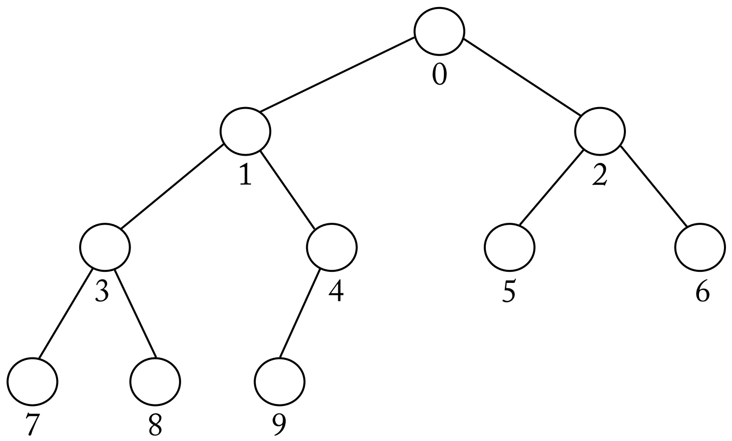

**Figure 3:** Numbering Vertices for Contiguous Storage

This numbering scheme can be used to identify each vertex in a complete binary tree: vertex zero is the root, vertex one is the left child of the root, vertex two is the right child of the root, and so forth. Note in particular that

- The left child of vertex *k* is vertex 2*k*+1.
- The right child of vertex *k* is vertex 2*k*+2.

- The parent vertex k is vertex *floor*((*k*-1)/2).
- If there are *n* vertices in the tree, the last one with a child is vertex *floor*(*n*/2) - 1.

If we let these vertex numbers be array indices, then each array location is associated with a vertex in the tree, and we can store the values at the vertices of the tree in the array: the value of vertex *k* is stored in array location *k*. The correspondence between array indices and vertex locations thus makes it possible to represent complete binary trees in arrays. The fact that the binary tree is complete means that every array location stores the value at a vertex, so no space is unused in the array.

## 16.5 Heapsort

We now have all the pieces we need to for an efficient and interesting sorting algorithm based on heaps. Suppose we have an array to be sorted. We can consider it to be a complete binary tree stored in an array as explained above. Then we can

- Make the tree into a heap as explained above.
- The largest value in a heap is at the root, which is always at array location zero. We can swap this value with the value at the end of the array and pretend the array is one element shorter. Then we have a complete binary tree that is almost a heap—we just need to sift the root value down the tree as far as necessary to make it one. Once we do, the tree will once again be a heap.
- We can then repeat this process again and again until the entire array is sorted.

This sorting algorithm, called *heapsort*, is shown in the Ruby code in Figure 4 below.

```
def heapify(array, i, max_index)
  tmp = array[i]
  j = 2*i + 1
  while j <= max_index
    j += 1 if j < max_index && array[j] < array[j+1]
    break if array[j] <= tmp
    array[i] = array[j]
    i, j = j, 2*i + 1
  end
  array[i] = tmp
end

def heap_sort(array)
  # make the entire array into a heap
  max_index = array.size-1
  ((max_index-1)/2).downto(0).each do | i |
    heapify(array,i,max_index)
  end

  # repeatedly remove the root and remake the heap
  loop do
    array[0],array[max_index] = array[max_index],array[0]
    max_index -= 1
    break if max_index <= 0
    heapify(array, 0, max_index)
  end
  return array
end
```

**Figure 4:** Heapsort

A complex analysis that we will not reproduce here shows that the number of comparisons done by heapsort in both the best, worst, and average cases are all in $O(n \lg n)$. Thus heapsort joins Shell sort, merge sort, and quicksort in our repertoire of fast sorting algorithms. Empirical studies have shown that while heapsort is not as fast as quicksort, it is not much slower than Shell sort and merge sort, with the advantages that it does not use any extra space,and it does not have bad worst case complexity.

## 16.6 Summary and Conclusion

A tree is a special sort of graph that is important in computing. One application of trees is for sorting: an array can be treated as a complete binary tree and then transformed into a heap. The heap can then be manipulated to sort the array in place in $O(n \lg n)$ time. This algorithm is called heapsort and it is a good algorithm to use when space is at a premium and respectable worst case complexity is required.

## 16.7 Review Questions

1. In Figure 1, what are the descendents of *r*? What are the ancestors of *r*?
2. How can you tell from a diagram whether a tree is ordered?
3. Is every full binary tree a complete binary tree? Is every complete binary tree a full binary tree?
4. Where is the largest value in a heap?
5. Using the heap data structure numbering scheme, which vertices are the left and right children of vertex 27? Which vertex is the parent of vertex 27?
6. What is the worst case behavior of heapsort?

## 16.8 Exercises

1. Represent the three trees in Figure 2 as sets of ordered pairs according to the definition of a graph.
2. Could the graph in Figure 1 still be a tree if *b* was its root? If so, redraw the tree in the usual way (that is, with the root at the top) to make clear the relationships between the vertices.
3. Draw a complete binary tree with 12 vertices, placing arbitrary values at the vertices. Use the algorithm discussed in the chapter to transform the tree into a heap, redrawing the tree at each step.
4. Suppose that we change the definition of the heap-order property to say that the value stored at a vertex is less than or equal to the values stored at its descendents. If we use the heapsort algorithm on trees that are heaps according to this definition, what will be the result?
5. In the heapsort algorithm in Figure 4, the `heapify()` operation is applied to vertices starting at `max_index-1`. Why does the algorithm not start at `max_index`?
6. Draw a complete binary tree with 12 vertex, placing arbitrary values at the vertices. Use the heapsort algorithm to sort the tree, redrawing the tree at each step, and placing removed values into a list representing the sorted array as they are removed from the tree.
7. Write a program to sort arrays of various sizes using heapsort, Shell sort, merge sort, and quicksort. Time your implementations and summarize the results.
8. Introspective sort is a quicksort-based algorithm recently devised by David Musser. introspective sort works like quicksort except that it keeps track of the depth of recursion (or of the stack), and when recursion gets too deep (about $2 \cdot \lg n$ recursive calls or stack elements), it switches to heapsort to sort sub-lists. This algorithm does $O(n \lg n)$ comparisons even in the worst case, sorts in place, and usually runs almost as fast as quicksort on average. Write an introspective sort function, time your implementation against standard quicksort, and summarize the results.

## 16.9 Review Question Answers

1. In Figure 1 the descendents of *r* are all the vertices in the tree. Vertex *r* has no ancestor except itself.
2. You can't tell from a diagram whether a tree is ordered; there must be some other notation to indicate that this is the case.
3. Not every full binary tree is complete because all the leaves of a tree might be on two levels, making it full, but some of the missing leaves at the bottom level might not be on the right, meaning that it is not complete. Every complete binary tree must be a full binary tree, however.
4. The largest value in a heap is always at the root.
5. Using the heap data structure numbering scheme, the left child of vertex 27 is vertex (2*27)+1= 55, the right child of vertex 27 is vertex (2*27)+2 = 56, and the parent of vertex 27 is vertex *floor*((27-1)/2) = 13.
6. The worst, best, and average case behavior of heapsort is in *O*(*n* lg *n*).