

This lab has 1 checkpoints. You have until the end of lab to complete checkpoints for full credit.  
This assignment's **repo name** is l09-HeapTrace

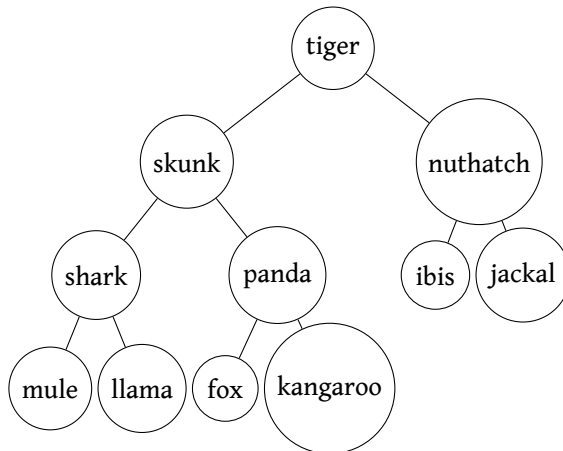
## Learning Outcomes

Upon completing this lab, students should be able to

- **Trace** the *heapsort* algorithm.
- **Translate** a heap between a tree view and an array view.

## Introduction

1. Consider a *max-heap* data structure containing `String` objects ordered lexicographically but stored in a `String[]`.  
(a) Translate the following *tree view* of a heap into its corresponding *array view*:



**Solution:**

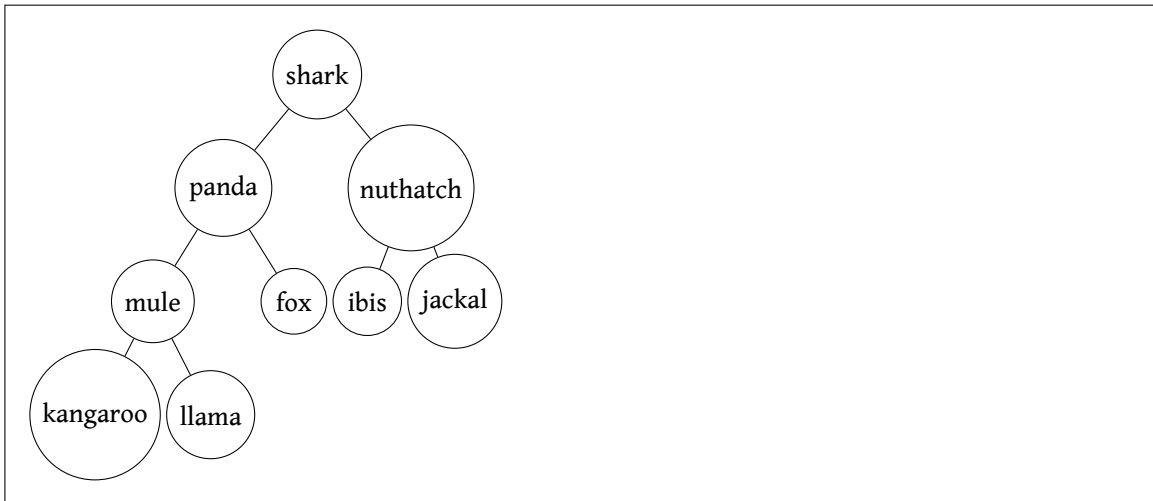
	0	1	2	3	4	5	6	7	8	9	10
A	tiger	skunk	nuthatch	shark	panda	ibis	jackal	mule	llama	fox	kangaroo

- (b) After two steps of *heapsort* where the root of the heap is swapped with the element in the furthest node still in the heap, heap size is reduced, and the root is bubbled-down to reestablish the heap property, the *array view* is (dark cells are already in sorted order):

	0	1	2	3	4	5	6	7	8	9	10
A	shark	panda	nuthatch	mule	fox	ibis	jackal	kangaroo	llama	skunk	tiger

Convert the *heap* at this point into a tree view.

**Solution:**



- (c) **After** swapping the correct element into array index 8 but **before** running `bubbleDown`, what does the *whole* array (heap and sorted elements) look like? Be sure to indicate the line between the heap and the sorted portions of the array.

**Solution:**

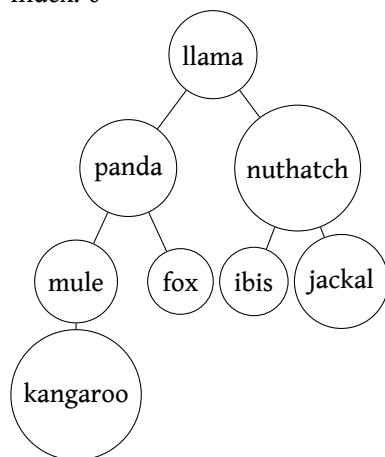
0	1	2	3	4	5	6	7	8	9	10
llama	panda	nuthatch	mule	fox	ibis	jackal	kangaroo	shark	skunk	tiger

- (d) `bubbleDown` will be called with the **index** number of the node that *might* be out of order; it will also have access to the array holding the heap/sorted entries. The method will be called *recursively* until the heap property is restored.

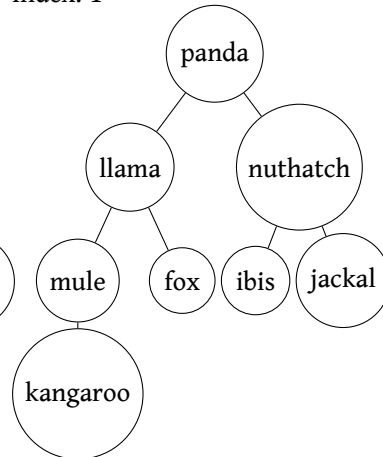
Give the **index** parameter and the *tree* representation of the heap portion of the array as `bubbleDown` starts while restoring the heap property from the point you captured in the previous question. Notice that the first heap value is just what you answered there.

**Solution:**

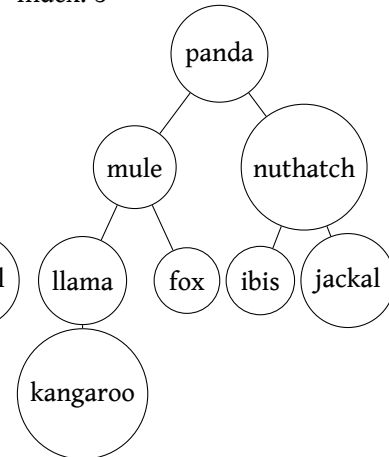
Index: 0



Index: 1



Index: 3



- ✓ (1) Show the lab instructor your drawings of the data structures.

```
1  private final int NULL = -1; // no such element
2  private String [] A; // set and filled
3
4  private int left(int index) {
5      int candidate = 2 * index + 1;
6      if (candidate >= heapEnd)
7          candidate = NULL;
8      return candidate;
9  }
10
11 private int right(int index) {
12     int candidate = 2 * index + 2;
13     if (candidate >= heapEnd)
14         candidate = NULL;
15     return candidate;
16 }
17
18 private int parent(int index) {
19     int candidate = NULL;
20     if (index > 0)
21         candidate = (index - 1) / 2;
22     return candidate;
23 }
24 private void bubbleUp(int index) {
25     if (index > 0) {
26         if (A[index].compareTo(A[parent(index)]) > 0) {
27             swap(index, parent(index));
28             bubbleUp(parent(index));
29         }
30     }
31 }
32
33 private void bubbleDown(int index) {
34     if (left(index) != NULL) { // at least one
35         int iBiggestChild = left(index);
36         if (right(index) != NULL) { // must test (two children)
37             int iRight = right(index);
38             if (A[iBiggestChild].compareTo(A[iRight]) < 0) // right child bigger
39                 iBiggestChild = iRight;
40         } // iBiggestChild is valid reference to the largest child of A[index]
41         if (A[iBiggestChild].compareTo(A[index]) > 0) {
42             swap(index, iBiggestChild);
43             bubbleDown(iBiggestChild);
44         }
45     }
```