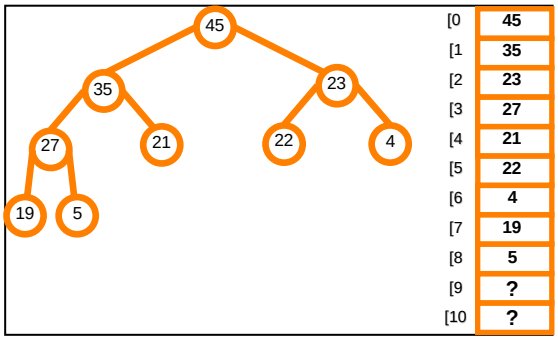
Heap and Priority Queue

1. **Assignment: Implement the Heap class from the UML code and pseudo-code in this worksheet (30 points).**
2. **Complete Heap Sort through the heapSort method. Use partialReheapifyDown method to complete the sorting. Create a HeapDriver to test with an ArrayList of 1,000 random elements. Print out results of sort. (20 points).**

A *priority queue* is a collection of data designed to make it easy to find the element with highest priority. Such a data structure might be used, for example, in a hospital emergency room to schedule an operating table. A *heap* is an efficient technique to implement a *priority queue*.



A heap stores values in a **complete binary tree.** The value of each node in the heap is greater than or equal to each of its child nodes. Notice that a heap is partially ordered, but not completely. In particular, the largest element is always at the root. Although we will continue to think of the heap as a tree, the internal representation will be in an ArrayList.

A complete binary tree can be efficiently stored as an ArrayList. The children of the node at index i are found at positions 2i+1 and 2i+2 in the ArrayList, the parent is stored at (i-1)/2.

Heap
- list: ArrayList<Integer>
+ Heap(): //constructor create new instance of list + enqueue(int entry):void + dequeue( ): int + size(): int - reheapifyUp(currentSpot: int): void - reheapifyDown(currentSpot: int): void + heapSort(ArrayList<Integer> data):void //data to be sorted - partialReheapifyDown(currentSpot: int, stop: int): void

Insert a new element into a Priority Queue (enqueue)

To insert a new value into a heap the value is pushed onto the back of the ArrayList list. This preserves the complete binary tree property, but not the heap ordering. To fix the ordering, the new value is swapped (*reheapify up)* into its new position. It is compared to its parent node. If larger, the node and the parent are exchanged. This continues until either the root is reached, or the new value finds its correct position less than its parent. Because this process follows a path up a complete binary tree, it is limited in steps to the height of the complete binary tree.

Heap and Priority Queue

Complete the code for swapping an element up the tree if its parent has a smaller value.

```
//Precondition: currentSpot is the index in list ArrayList of element to reheapify
//Postcondition: The heap properties are restored.
reheapifyUp(int currentSpot){
      Find the index of your parent node to node at currentSpot
      while element is not root and greater than parent
            swap element at currentSpot with parent
            currentSpot = parent
            parent = (currentSpot-1)/2
}
```

Complete the code for enqueue (or inserting) an element into a heap. The enqueue method puts the new element at the end of the ArrayList and then reheapifyUp is called to restore the heap property.

```
enqueue(int entry){
      push entry onto back of ArrayList list
      call reheapifyUp with the index of the newly inserted key
}
```

Remove Highest Priority Element (dequeue())

The largest (highest priority) value is always found at the root. But when this value is removed it leaves a "hole." Filling this hole with the last element in the heap restores the complete binary tree property, but not the heap order property. To restore the heap, order the element brought up to replace the root must *reheapify down* into position by swapping down the tree with its maximum child to maintain the heap property.

```
//Precondition: currentSpot is the index in list ArrayList of element to reheapify
//Postcondition: The heap properties are restored.
reheapifyDown(int currentSpot){
  int leftChild <- currentSpot*2+1; //Declare child indices
  int rightChild <- currentSpot*2+2; //Declare child indices
  while (leftChild < size())  //While not at a leaf node

        int largest; //Find largest of two children
        if (rightChild <size()&& list[leftChild]<list[rightChild])
              largest <- rightChild;
        else
              largest <- leftChild;
        if (list[currentSpot] >= list[largest])
              break;      // key is >= children, can stop
        //swap currentSpot and largest child elements
        int temp <- list[currentSpot];
        list[currentSpot] <- list[largest];
        list[largest] <- temp;
        currentSpot <- largest; //Reset currentSpot
        leftChild <- currentSpot*2+1; //children indices
        rightChild <- currentSpot*2+2;  //children indices
}
```

```
//Precondition: heap properties in place
//Postcondition: Maximum value returned. The heap properties are restored.
int dequeue()
{
  Store root element in a temp variable
  copy last element in list into first (root) location
  remove last element in list
  reheapifyDown(0) //the new root element down the tree
  return  stored temp element
}
```

Sort an ArrayList using a heap data structure (heapSort())

By transforming an array (ArrayList) into a heap in O(n lg n) time and then dequeing all elements in O(n lg n) time, we have a transform and conquer algorithm to sort data in O(n lg n) time. By storing the dequeued value in the same ArrayList, we can complete this sort with no additional memory required. It is an in place sorting algorithm.

```
//Precondition: data is an unsorted ArrayList
//Postcondition: ArrayList data is sorted. Heap properties are not maintained.
heapSort(ArrayList<Integer> data)
{
  list <- data
  //convert list to a heap structure
  For each element 0..n-1 in list
        perform re-heapification upward

  unsorted <- size()

  while (unsorted > 1)
        Swap list[0] with last unsorted element (data[unsorted-1]).
        partialReheapifyDown(0, unsorted-1)//maintain heap
        unsorted <- unsorted -1
}
```

Special version of reheapify down that stops at the end of heap and before sorted data.

```
//Precondition: Heap properties maintained from 0 to stop. From stop +1 to size data sorted.
//Postcondition:The heap properties restored from 0 to stop-1. From stop to size data sorted.
void partialReheapifyDown(int currentSpot, int stop)
//reheapify down logic, replace size() with stop in both locations
```

Heap and Priority Queue

Application of the Heap (Review):

1. What is the difference between a priority queue and a queue?

2. Where is the largest value found in a heap?

3. What are the two heap properties (tree and node order properties)?

4. Where is the 2nd largest element in a heap? The third largest element?

5. Show that unless other information is maintained, finding the *minimum* value in a heap must be an O(n) operation. Hint: How many elements in the heap could potentially be the minimum?

6. What is the height of a heap that contains 100 elements?

7. Most of the work is done by reheapifyUp and reheapifyDown. What is the worst-case running time for each of these methods? Can you prove it?

8. Running time (Big-0) to use a Heap to order an array of unordered elements?