Worksheet 33: Heaps and Priority Queues[SOLUTION]

In preparation: Read Chapter 10 to learn more about trees, and chapter 11 to learn more about the Priority Queue data type. In you have not done so already, complete Worksheets 29 and 30 to explore the idea of a binary tree.

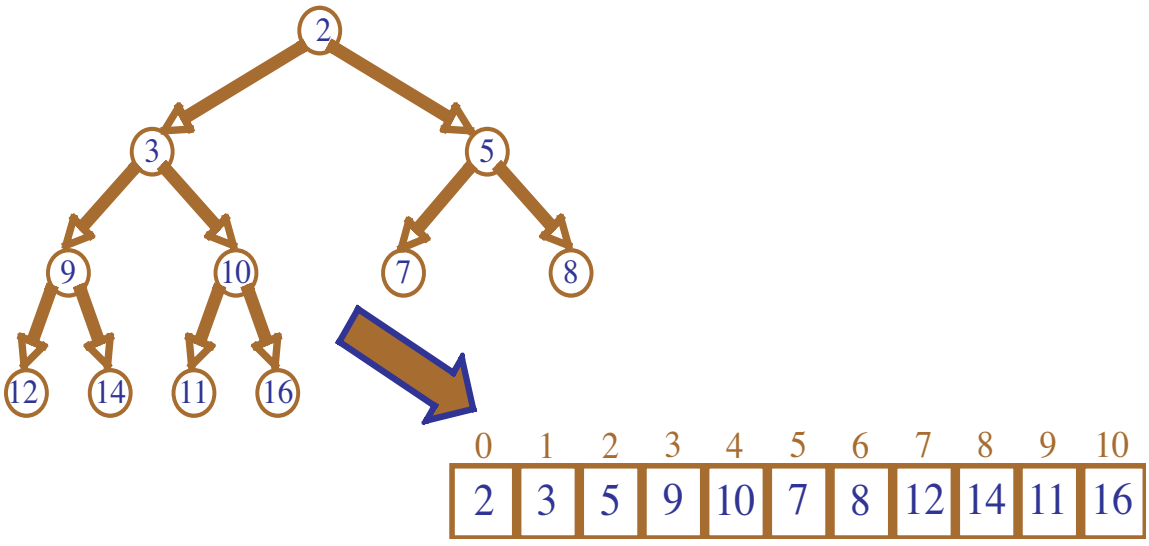
Recall that a *priority queue* is a collection designed to make it easy to find the element with highest priority. Such a data structure might be used, for example, in a hospital emergency room to schedule an operating table. The conceptual interface for this data type is shown at right.

```
Priority Queue Conceptual Interface

void add (TYPE newValue);
TYPE getFirst ();
void removeFirst ();
```

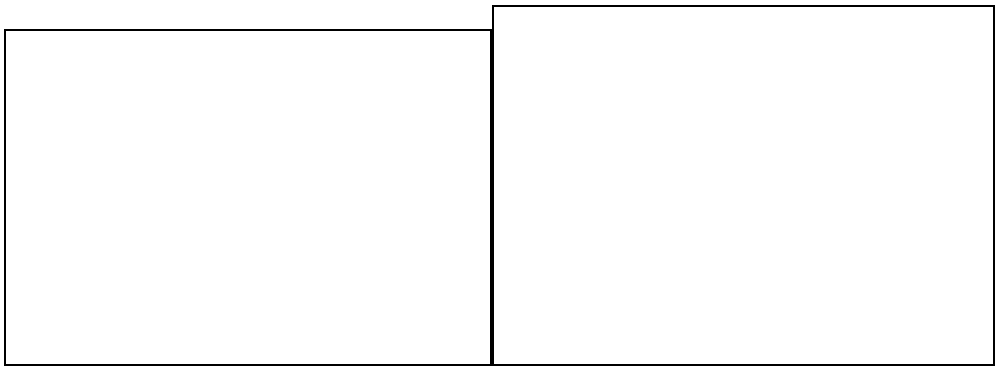
A *heap* is an efficient technique to implement a Priority Queue. A heap stores values in a complete binary tree in which the value of each node is less than or equal to each of its child nodes. This is termed the *heap order property*.

Notice that a heap is partially ordered, but not completely. In particular, the smallest element is always at the root. Although we will continue to think of the heap as a tree, the internal representation will be in a dynamic array. Recall that a complete binary tree can be efficiently stored as a dynamic array. The children of node i are found at positions 2i+1 and 2i+2, the parent at (i-1)/2.

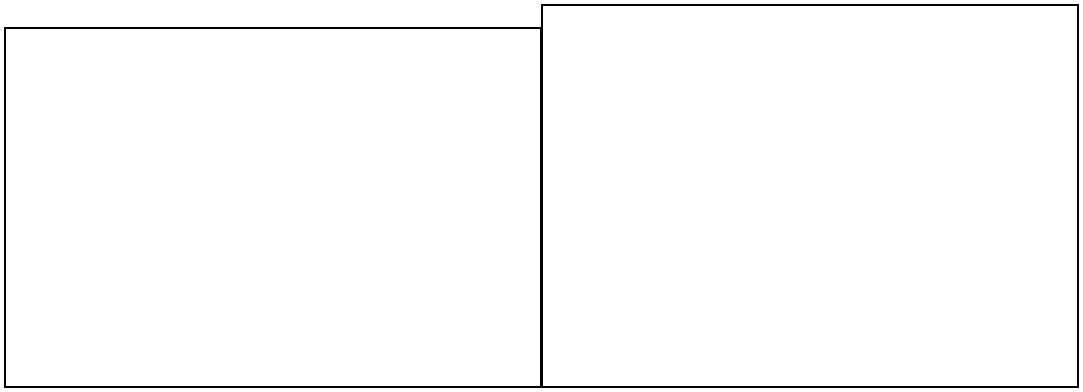


To insert a new value into a heap the value is first added to the end. This preserves the complete binary tree property, but not the heap ordering. To fix the ordering, the new value is *percolated* into position. It is compared to its parent node. If smaller, the node and the parent are exchanged. This continues until the root is reached, or the new value

finds its correct position. Because this process follows a path in a complete binary tree, it is O(log n).



The smallest 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 new value must *percolate down* into position.



We abstract the restoring heap property into a new routine termed adjustHeap:

```
void HeapRemoveFirst(struct dyArray *heap) {
     int last = dyArraySize(heap) - 1;
     assert (last != 0); /* make sure we have at least one element */
                /* Copy the last element to the first  position */
     dyArrayPut(heap, 0, dyArrayrGet(heap, last));
     dyArrayRemoveAt(heap, last);       /* Remove last element.*/
     adjustHeap(heap, last-1, 0);/* Rebuild heap */
  }
```

AdjustHeap can easily be written as a recursive routine:

```
void _adjustHeap (struct dyArray * heap, int max, int pos)
  int leftChild = 2*pos + 1; int rightChild = 2 * pos + 2;
  if (rightChild < max) { /* we have two children */
     get index of smallest child
```

```
    if value at pos < value of smallest child
       swap with smallest child,  call adjustHeap (max, index of smallest child)
 else if (leftchild < max) { /* we have one child */
    if value at pos < value of child
      swap with smallest child, call adjustHeap (max, index of left child) /* Is this adjust
necessary? */
  /* else no children, done */
```

The process follows a path in the complete tree, and hence is O(log n).

Two internal routines help in the completion of both these routines. The function *swap* , which you wrote in an earlier worksheet, will exchange two dynamic array positions. The function *indexSmallest* takes two index values, and returns the position of the smaller of the two.

Using these ideas, complete the implementation of the Heap data structure:

```
void swap (struct dyArray * v, int i, int j) {  /* swap elements i j */
    TYPE temp = dyArrayGet(v, i);
    dyArrayPut(v, i, dyArrayGet(v, j));
    dyArrayPut(v, j, temp);
  }

 int indexSmallest (struct dyArray * v, int i, int j) {
    /* return index of smallest element */
    if (LT(dyArrayGet(v, i), dyArrayGet(v, j))
       return i;
    return j;
  }

 TYPE heapGetFirst (struct dyArray *heap) {
    assert(dyArraySize(heap) > 0);
    return dyArrayGet(heap, 0);
  }

 void heapRemoveFirst(struct dyArray *heap) {
    int last = dyArraySize(heap)-1;
    if (last != 0)       /* Copy the last element to the first */
       dyArrayPut(heap, 0, dyArrayGet(heap, last)); /* position */
    dyArrayRemoveAt(heap, last);         /* Remove last element. */
    _adjustHeap(heap, last - 1, 0);/* Rebuild heap */
  }
```

```
/*      Add a node to the heap

        param: heap    pointer to the heap
        param: node    node to be added to the heap
        pre:   heap is not null
        post:  node is added to the heap
*/
void addHeap(DynArr *heap, TYPE val)
{
  int parent;
  int pos = sizeDynArr(heap);

  addDynArr(heap, val);

  while(pos != 0)
    {
        parent = (pos-1)/2;
        if(compare(getDynArr(heap, pos), getDynArr(heap, parent)) == -1)
      {
            swapDynArr(heap, parent, pos);
            pos = parent;
      } else return;
    }
}

/*      Adjust heap to maintain heap property

        param: heap    pointer to the heap
        param: max     max index of the heap nodes in the dynamic array
        param: pos     position index where the adjustment starts
        pre:   max < size
        post:  heap property is maintained for nodes from index pos to index max
*/
void _adjustHeap(DynArr *heap, int max, int pos)
{
  int left, right, small;
  assert(max < sizeDynArr(heap));
  left = 2*pos + 1;
  right = 2*pos + 2;
  if(right <= max) /* Two children */
    {
        small = _smallerIndexHeap(heap, left, right);
        if(compare(getDynArr(heap, small), getDynArr(heap, pos)) == -1)
      {
            swapDynArr(heap, pos, small);
            _adjustHeap(heap, max, small);
      }
    }
  else if(left <= max) /* One child */
    {
```

```
            if(compare(getDynArr(heap, left), getDynArr(heap, pos)) == -1)
        {
                    swapDynArr(heap, pos, left);
                    _adjustHeap(heap, max, left);
        }
    }
}
```