Worksheet 33: Heaps and Priority Queues

Group 11

In preparation: Read Chapter 10 to learn more about trees, and chapter 11 to learn more about the Priority Queue data type. If 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.
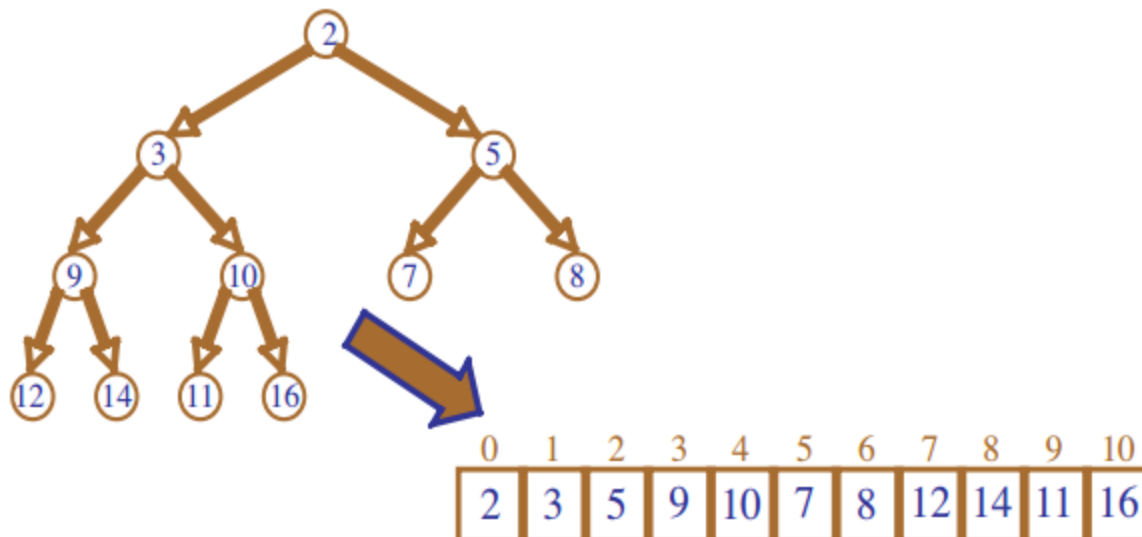
To insert a new value into a heap the value is first added to the end. This preserves the complete binary tree property, but not the heap ordering. To fix the ordering, the new value is percolated into position. It is compared to its parent node. If smaller, the node and the parent are exchanged. This 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).
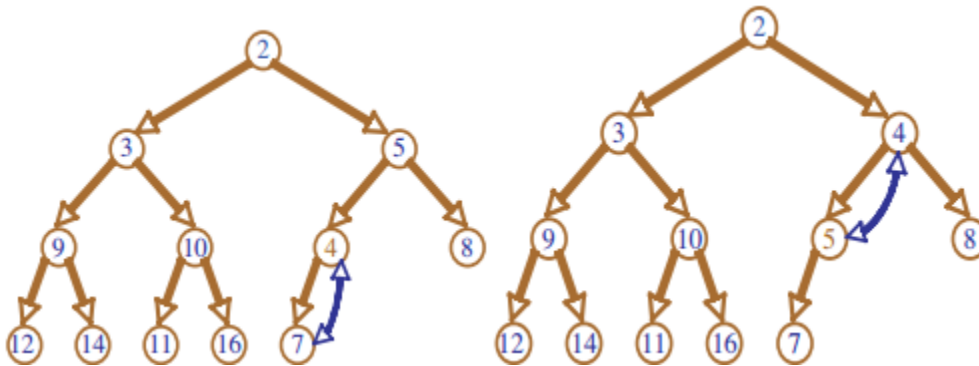
The smallest value is always found at the root. But when this value is removed it leaves a "hole." Filling this hole with the last element in the heap restores the complete binary tree property, but not the heap order property. To restore the heap order the new value must percolate down into position.
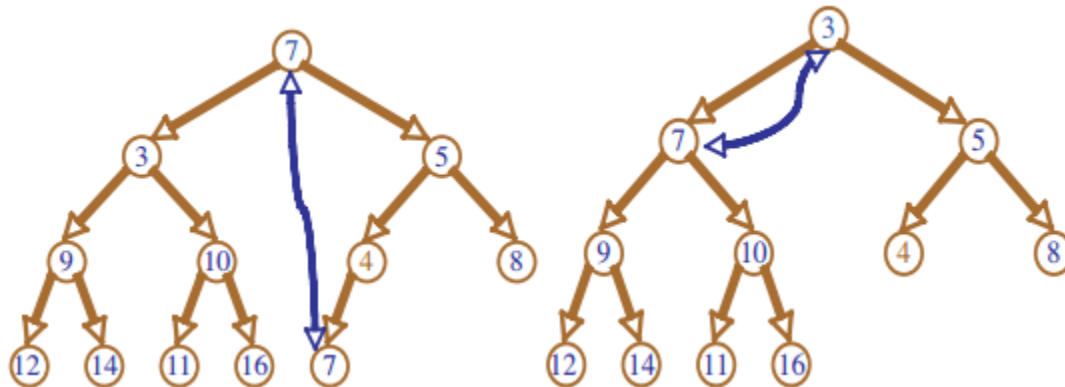
We abstract the restoring heap property into a new routine termed adjustHeap:

```
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, 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)
  /* 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;
    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, 0);/* Rebuild heap */
  }


void _adjustHeap (struct dyArray * heap, int max, int pos) {

int leftIdx = pos * 2 + 1; // finding left child
int rghtIdx = pos * 2 + 2; // finding  right child

        if ( rghtIdx < max ) {

        int smallest = indexSmallest ( heap, leftIdx, rghtIdx ) // returns the smallest child of
the position passed
                    if ( smallest < pos ){ // compares the smallest and pos

swap ( heap, smallest, pos ); // if smaller it swaps

adjustHeap ( heap, max, smallest ); } then calls the adjust recursively to compare further

        }

        else if ( leftIdx < max ){  // else if one child is found

        If ( leftIdx < pos ){ // compare the left with the pos

        Swap ( heap, leftIdx, pos ); // swap if left is smaller

        adjustHeap ( heap, max, leftIdx ); // call adjust recursively to compare further

        }

        }

        /* else no children, done */

}
```

//element is added at in the next open slot (elements are added left to right)

//while the new values is less than a parent swap value with the parent

```
void heapAdd (struct dyArray * heap, TYPE newValue) {
        int parent;
        int pos = sizeDynArr(heap); // determine the size of the heap ( size of the array)
        dyArrayAdd(heap, newValue); /* adds to end – now need to adjust position */
        while(pos != 0) { //continue percolating up until we reach root node
                parent = (pos-1)/2;//parent of a node
                if(compare(getDynArr(heap, pos), getDynArr(heap, parent)) == -1){
                        swapDynArr(heap, parent, pos);
                        pos = parent;
                }
                else return;
        }
}
```

Compare function is a function from Assignment #4

```
int compare(TYPE left, TYPE right) {
   struct Data *L = (struct Data *)left;
   struct Data *R = (struct Data *)right;

   if(L->number < R->number){
      return 1;
   }
   else if(L->number > R->number){
      return -1;
   }
   else{
      return 0;
   }
}
```