Heap Implementation
Goals

- Heap Representation
- Heap Priority Queue ADT Implementation
Complete binary tree has structure that is efficiently implemented with a **DynArr**:

- Children of node $i$ are stored at $2i + 1$ and $2i + 2$
- Parent of node $i$ is at $\text{floor}(\frac{(i - 1)}{2})$

**Why is this a bad idea if tree is not complete?**
If the tree is not complete (it is thin, unbalanced, etc.), the **DynArr** implementation will be full of holes

Big gaps where the level is not filled!
Heap Implementation: \textit{add}

- Where does the new value get placed to maintain completeness?
- How do we guarantee the heap order property?
  - How do we compute a parent index?
  - When do we ‘stop’
- Complete Worksheet #33 – heapAdd()
Heap Implementation: `removeMin`

```c
void removeMinHeap(DynArr *heap){
    int last;
    assert(sizeDynArr(heap) > 0);
    last = sizeDynArr(heap) - 1;
    putDynArr(heap, 0, getDynArr(heap, last));  /* Copy the last element to the first */
    removeAtDynArr(heap, last);                /* Remove last element. */
    _adjustHeap(heap, last, 0);               /* Rebuild heap */
}
```

Percolates down from Index 0 to last (not including last...which is one beyond the end now!)
Heap Implementation: removeMin
Heap Implementation: removeMin (cont.)

last = sizeDynArr(heap) – 1;
putDynArr(heap, 0, getDynArr(heap, last));
/* Copy the last element to the first */
removeAtDynArr(heap, last);
_adjustHeap(heap, last, 0);
Heap Implementation: \_adjustHeap

\_adjustHeap(heap, upTo, start);
\_adjustHeap(heap, last, 0);

Smallest child (min = 3)
Heap Implementation: \_adjustHeap

current is less than smallest child so \_adjustHeap exits and removeMin exits
Recursive _adjustHeap

```c
void _adjustHeap(struct DynArr *heap, int max, int pos) {
    int leftIdx = pos * 2 + 1;
    int rightIdx = pos * 2 + 2;

    if (rightIdx < max) {
        /* Have two children? */
        /* Get index of smallest child (_minIdx). */
        /* Compare smallest child to pos. */
        /* If necessary, swap and call _adjustHeap(max, minIdx). */
    }
    else if (leftIdx < max) {
        /* Have only one child. */
        /* Compare child to parent. */
        /* If necessary, swap and call _adjustHeap(max, leftIdx). */
    }
    /* Else no children, we are at bottom → done. */
}
```
void swap(struct DynArr *arr, int i, int j) {
    /* Swap elements at indices i and j. */
    TYPE tmp = arr->data[i];
    arr->data[i] = arr->data[j];
    arr->data[j] = tmp;
}

int minIdx(struct DynArr *arr, int i, int j) {
    /* Return index of smallest element value. */
    if (compare(arr->data[i], arr->data[j]) == -1)
        return i;
    return j;
}
**Priority Queues: Performance Evaluation**

<table>
<thead>
<tr>
<th></th>
<th>SortedVector</th>
<th>SortedList</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>(O(n)) \n Binary search \n Slide data up</td>
<td>(O(n)) \n Linear search</td>
<td>(O(\log n)) \n Percolate up</td>
</tr>
<tr>
<td><strong>getMin</strong></td>
<td>(O(1)) \n get(0)</td>
<td>(O(1)) \n Returns firstLink val</td>
<td>(O(1)) \n Get root node</td>
</tr>
<tr>
<td><strong>removeMin</strong></td>
<td>(O(n)) \n Slide data down \n (O(1)) : Reverse Order</td>
<td>(O(1)) \n removeFront()</td>
<td>(O(\log n)) \n Percolate down</td>
</tr>
</tbody>
</table>

So, which is the best implementation of a priority queue?
Priority Queues: Performance Evaluation

• Recall that a priority queue’s main purpose is rapidly accessing and removing the smallest element!

• Consider a case where you will insert (and ultimately remove) \( n \) elements:
  
  — ReverseSortedVector and SortedList:
    
    Insertions: \( n \times n = n^2 \)
    
    Removals: \( n \times 1 = n \)
    
    Total time: \( n^2 + n = O(n^2) \)
    
  — Heap:
    
    Insertions: \( n \times \log n \)
    
    Removals: \( n \times \log n \)
    
    Total time: \( n \times \log n + n \times \log n = 2n \log n = O(n \log n) \)

How do they compare in terms of space requirements?
Your Turn

• Complete Worksheet #33 - _adjustHeap( .. )