Priority Queue ADT & Heaps
Goals

• Introduce the Priority Queue ADT
• Heap Data Structure Concepts
• Not really a FIFO queue – misnomer!!

• Associates a “priority” with each element in the collection:
  – First element has the highest priority (typically, lowest value)

• Applications of priority queues:
  – To do list with priorities
  – Active processes in an OS
Priority Queue ADT: Interface

- Next element returned has highest priority

```c
void add(newValue + priority);
TYPE getMin();
void removeMin();
```
Heap: has 2 completely different meanings

1. Classic data structure used to implement priority queues
2. Memory space used for dynamic allocation

We will study the data structure (not dynamic memory allocation)
Binary Heap data structure: a \textit{complete} binary tree in which every node’s value is less than or equal to the values of its children (min heap)

Review: a \textit{complete} binary tree is a tree in which
1. Every node has at most two children (binary)
2. The tree is entirely filled except for the bottom level which is filled from left to right (complete)
   – Longest path is $\text{ceiling}(\log n)$ for $n$ nodes
Min-Heap: Example

Root = Smallest element

Next open spot

Last filled position
(not necessarily the last element added)
Maintaining the Heap: Addition

Add element: 4

Place new element in next available position, then fix it by “percolating up”
Maintaining the Heap: Addition (cont.)

Percolating up:
while new value is less than parent,
swap value with parent

After first iteration (swapped with 7)

After second iteration (swapped with 5)
New value not less than parent → Done
Maintaining the Heap: Removal

- Since each node’s value is less than or equal to the values of its children, the root is always the smallest element.

- Thus, the operations \texttt{getMin} and \texttt{removeMin} access and remove the root node, respectively.

- Heap removal (\texttt{removeMin}): What do we replace the root node with? Hint: How do we maintain the completeness of the tree?
Heap removal *(removeMin)*:

1. Replace root with the element in the last filled position
2. Fix heap by “percolating down”
removeMin:
1. Move element in last filled pos into root
2. Percolate down

Root = Smallest element

Last filled position
Maintaining the Heap: Removal (cont.)

Percolating down:
while greater than smallest child
swap with smallest child

Root value removed
(16 copied to root and last node removed)

After first iteration (swapped with 3)
Maintaining the Heap: Removal (cont.)

Percolating down:
while greater than smallest child
swap with smallest child

After second iteration (moved 9 up)

After third iteration (moved 12 up)
Reached leaf node → Stop percolating
Maintaining the Heap: Removal (cont.)

Root = New smallest element

New last filled position
Practice

Insert the following numbers into a min-heap in the order given: 54, 13, 32, 42, 52, 12, 6, 28, 73, 36
Remove the minimum value from the min-heap
Your Turn

• Complete Worksheet: Heaps Practice
Heap Implementation
Goals

- Heap Representation
- Heap Priority Queue ADT Implementation
Complete binary tree has structure that is efficiently represented with an array (or dynamic array)

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

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( )
Write : addHeap

```c
void addHeap (struct dyArray * heap, TYPE newValue) {

}
```
Heap Implementation: removeMin

```c
void removeMinHeap(DynArr *heap){
    int last;
    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: \textit{removeMin}

```
Heap data: 2 4 8 3 5 9 10 14 12 11 16
```

```
Binary heap: 2 3 4 9 10 5 8 14 12 11 16
```

```
min: 2
```

```
last: 11
```
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`

```plaintext
_adjustHeap(heap, upTo, start);
_adjustHeap(heap, last, 0);
```

Diagram:

```
  7
 /|
3  4
 |
  v
  3
  |
  v
  9
 /|/
14 12 11
 |
  v
  16
 |
  v
10
```

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

\_adjustHeap(heap, last, 1);
Heap Implementation: _adjustHeap

current is less than smallest child so _adjustHeap exits and removeMin exits
void _adjustHeap (struct dyArray * heap, int max, int pos) {

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

So, which is the best implementation of a priority queue?

*(What if wrote a getMin() for AVLTree?)*
• 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( .. )
BuildHeap and Heapsort
Goals

• Build a heap from an array of arbitrary values
• HeapSort algorithm
• Analysis of HeapSort
• How do we build a heap from an arbitrary array of values???
BuildHeap: Is this a proper heap?

Are any of the subtrees \textit{guaranteed} to be proper heaps?
BuildHeap: Leaves are proper heaps

Size = 11
Size/2 - 1 = 4

First non-leaf
• How can we use this information to build a heap from a random array?
• _adjustHeap: takes a binary tree, rooted at a node, where all subtrees of that node are proper heaps and percolates down from that node to ensure that it is a proper heap

```c
void _adjustHeap(struct DynArr *heap, int max, int pos)
```

Adjust up to (not inclusive)  
Adjust from
BuildHeap

- Find the first non-leaf node, i, (going from bottom to top, right to left)
- adjust heap from it to max
- Decrement i and repeat until you process the root
BuildHeap: Leaves are proper heaps

Size = 11
Size/2 – 1 = 4
HeapSort

- BuildHeap and _adjustHeap are the keys to an **efficient, in-place**, sorting algorithm
  - in-place means that we don’t require any extra storage for the algorithm
- Any ideas???
HeapSort

1. BuildHeap – turn arbitrary array into a heap

2. Swap first and last elements

3. Adjust Heap (from 0 to the last...not inclusive!)

4. Repeat 2-3 but decrement last each time through
Size = 6
i = Size/2 – 1 = 2

i = 2
_adjustHeap(heap, 6, 2)
HeapSort Simulation: BuildHeap

Size = 6
i=1

_adjustHeap(heap,6,1)
HeapSort Simulation: BuildHeap

Size = 6
i = 0

i=0
_adjustHeap(heap,6,0)
HeapSort Simulation: BuildHeap

\[
\begin{array}{c|c|c|c|c|c}
0 & 1 & 2 & 3 & 4 & 5 \\
\hline
3 & 4 & 7 & 9 & 5 & 8 \\
\end{array}
\]

\[
i=-1
\]

Done...with BuildHeap ....now let’s sort it!
HeapSort

1. BuildHeap

2. Swap first and last

3. Adjust Heap (from 0 to the last)

4. Repeat 2-3 but decrement last
HeapSort Simulation: Sort in Place

Iteration 1

0 1 2 3 4 5

| 3 | 4 | 7 | 9 | 5 | 8 |

i=5

Swap(v, 0, i)

_adjustHeap(v, i, 0);
i=5
Swap(v, 0, i)
_adjustHeap(v, i, 0);
HeapSort Simulation: Sort in Place

i=4
\textit{Swap(v, 0, i)}
\texttt{_adjustHeap(v, i, 0)};
i=4
Swap(v, 0, i)
_adjustHeap(v, i, 0);
HeapSort Simulation: *Sort in Place*

Iteration 3

```plaintext
i = 3
Swap(v, 0, i)
_adjustHeap(v, i, 0);
```
HeapSort Simulation: Sort in Place

Iteration 3

i=3
Swap(v, 0, i)
_adjjustHeap(v, i, 0);
i=2
\[\text{Swap}(v, 0, i)\]
\[\text{_adjustHeap}(v, i, 0);\]
i=2
Swap(v, 0, i)
_adjustHeap(v, i, 0);
i=1
_Swp(v, 0, i)_
_adjustHeap(v, i, 0);
HeapSort Simulation: Sort in Place

Iteration 5

i = 1
Swap(v, 0, i)
_adjustHeap(v, i, 0);
HeapSort Simulation: Sort in Place

0 1 2 3 4 5
9 8 7 5 4 3

i=0
DONE
HeapSort Performance

• Build Heap:
  
  \( \frac{n}{2} \) calls to \_adjustHeap = \( O(n \log n) \)

• HeapSort:
  
  \( n \) calls to swap and adjust = \( O(n \log n) \)

• Total:
  
  \( O(n \log n) \)
Your Turn

• Worksheet 34 – BuildHeap and Heapsort