CS 261 – Data Structures

Priority Queue ADT & Heaps
Priority Queue ADT

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

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

```java
void add(newValue);

TYPE getFirst();

void removeFirst();
```
Priority Queues: **Implementations**

<table>
<thead>
<tr>
<th></th>
<th>SortedVector</th>
<th>SortedList</th>
<th>LinkedList</th>
</tr>
</thead>
</table>
| **add**        | O(n)  
|                | Binary search  
|                | Slide data up     | O(n)  
|                | Linear search   | O(1)  
|                | addLast(obj)      |                     |
| **getFirst**   | O(1)  
|                | elementAt(0)     | O(1)  
|                | Returns head.obj |                     |
| **removeFirst**| O(n)  
|                | Slide data down  
|                | O(1)  
|                | Reverse Order    | O(n)  
|                | head.remove()    |                     |
|                | Linear search then 
|                | remove smallest  |                     |

We can definitely do better than these!!!

What about a Skip List?
Heap Data Structure

Heap: has 2 completely different meanings:

1. Data structure for priority queues
2. Memory space for dynamic allocation

We will study the data structure

(not dynamic memory allocation)
Heap Data Structure

Heap = Complete binary tree in which every node’s value ≤ the children values
Heap: Example

Root = Smallest element

Last filled position
(not necessarily the last object added)
Complete Binary Tree

1. Every node has at most two children (binary)

2. Children have an arbitrary order

3. Completely filled except for the bottom level, which is filled from left to right (complete)

4. Longest path is \( \text{ceiling}(\log n) \) for \( n \) nodes
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 < parent, swap value with parent

After first iteration (swapped with 7)

After second iteration (swapped with 5)
Maintaining the Heap: **Removal**

The root is always the smallest element

→ `getFirst` and `removeFirst` access and remove the root node
Maintaining the Heap: **Removal**

1. Replace the root with the element in the last filled position

2. Fix heap by “percolating down”
Maintaining the Heap: Removal

`removeMin`:
1. Move value in last element to root
2. Percolate down

Root = Smallest element

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

Percolating down:
while > smallest child
swap with smallest child

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

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

After second iteration (moved 9 up)

After third iteration (moved 12 up)

Percolating down: while > smallest child swap with smallest child
Maintaining the Heap: **Removal (cont.)**

Root = New smallest element

New last filled position
Heap Representation as Dynamic Array

- Children of a node at index $i$ are stored at indices $2i + 1$ and $2i + 2$

- Parent of node at index $i$ is at index $\text{floor}((i - 1) / 2)$
Creating New Abstraction

```c
struct Heap {
    /* encapsulation */
    struct DynArr arr;
};

OR

struct DynArr *heap;
```
Heap Implementation: Add 4 to the heap

```c
void addHeap(struct DynArr *heap, TYPE val) {
    int pos = sizeDynArr(heap); /* next open spot. */
    int pidx = (pos - 1)/2; /* Get parent index */
    TYPE parentVal;
    ...
}
```

Parent position \(\textbf{pidx}\)  Next open spot \(\textbf{pos}\)

```
0 1 2 3 4 5 6 7 8 9 10 11
 2 3 5 9 10 7 8 14 12 11 16 4
```
Heap Implementation: add (cont.)

```c
void addHeap(struct DynArr *heap, TYPE val) {
    int pos = sizeDynArr(heap);
    int pidx = (pos - 1) / 2;
    TYPE parentVal;
    /* Make room for new element (but don’t add it yet). */
    if (pos >= heap->cap)
        _setCapDynArr(heap, 2 * heap->cap);
    if (pos > 0) parentVal = heap->data[pidx];
    ...
}
```

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>10</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
void addHeap(struct DynArr *heap, TYPE val) {
    
    /* While not at root and new value is less than parent. */
    while (pos > 0 && LT(val, parentVal)) {
        /* New value is smaller \rightarrow move parent down. */
        heap->data[pos] = parentVal;
        pos = pidx;
        pidx = (pos - 1) / 2; /* percolate up */
    }
}
Heap Implementation: add (cont.)

```c
void addHeap(struct DynArr *heap, TYPE val) {
    ...  
    /* While not at root and new value is less than parent. */
    while (pos > 0 && LT(val, parentVal)) {
        /* New value is smaller → move parent down. */
        heap->data[pos] = parentVal;
        pos = pidx;
        pidx = (pos - 1) / 2; /* percolate up */
        if (pos > 0) /* Get parent (if not at root). */
            parentVal = heap->data[pidx];
    }
```

![Heap Implementation Diagram](image-url)
Heap Implementation: add (cont.)

```c
void addHeap(struct DynArr *heap, TYPE val) {
    ... 
    while (pos > 0 && LT(val, parentVal)) {
        SWAP
    } /* reached root or parent is smaller than new value. */
    heap->data[pos] = val; /* Now add new value. */
}
```

Second part of while test fails: stop iteration and add new value

```
<table>
<thead>
<tr>
<th>pidx</th>
<th>pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>7</td>
</tr>
</tbody>
</table>
```
Heap Implementation: add (cont.)

```
  2
 / \
3   5
 /   /\n9   10 7 \
 /   /   / \
14 12 11 16 4
```

```
0 1 2 3 4 5 6 7 8 9 10 11
2 3 5 9 10 7 8 14 12 11 16 4
```
Heap Implementation: add (cont.)

```
 2
 /\  
|  3  |
|   \ |
|  9  |
 |   \|
| 14  |

 0 1 2 3 4 5 6 7 8 9 10 11
 2 3 5 9 10 4 8 14 12 11 16 7
```
Heap Implementation: add (cont.)

The diagram shows a binary heap with nodes labeled from 2 to 16. The nodes are connected to form a complete binary tree. The root node is 2, and the tree is populated with values 14, 12, 11, 16, 7, 5, 3, 4, 9, 10, 8. The diagram also includes labels 'pidx' and 'pos' to indicate specific positions in the tree.
Heap Implementation: add (cont.)

```
  2
 /\  \\
3-+ 4-+ 5
 |   |   |
|   |   |
|   v   v
9-+ 10-+ 11-+ 16
 |     |     |
|     v     v
|     14-+ 12-+ 11-+ 16
|         |         |
|         v         v
|         0-+ 1-+ 2-+ 3-+ 4-+ 5-+ 6-+ 7-+ 8-+ 9-+ 10-+ 11
|         |   |   |   |   |   |   |   |   |   |   |   |
  2   3   4   9   10   5   8   14  12  11  16  7
```
Heap Implementation: removeMin

```c
void removeMinHeap(struct DynArr *heap) {
    int last = sizeDynArr(heap) - 1;
    if (last != 0) /* Copy the last element to the first */
        heap->data[0] = heap->data[last];
}
```
Heap Implementation: removeMin

```c
void removeMinHeap(struct DynArr *heap) {
    int last = sizeDynArr(heap) - 1;
    if (last != 0)
        heap->data[0] = heap->data[last];
    removeAtDynArr(heap, last); /* Remove last */
    /* Rebuild heap property. */
    _adjustHeap(heap, last, 0);
}
```
Heap Implementation: removeMin (cont.)
Heap Implementation: removeMin (cont.)
Heap Implementation: removeMin (cont.)

- val > min \(\rightarrow\) Copy min to parent spot (idx)
- Move idx to child

Smallest child (min = 3)

val = 7
Heap Implementation: \texttt{removeMin (cont.)}

\texttt{val > min} \to \text{Copy min to parent spot (idx)}

\text{Move idx to child}

\texttt{val = 7}
Heap Implementation: **removeMin** (cont.)

val < min → Copy val into idx spot
Done

Smallest child (min = 9)
Heap Implementation: **removeMin** (cont.)

val < min $\rightarrow$ Copy val into idx spot

Done

val = 7

idx

max

max

heap

0 1 2 3 4 5 6 7 8 9 10 11
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;
}
Useful Routines: minIdx

```c
int minIdx(struct DynArr *arr, int i, int j)
{
    /* Return index of smallest element value. */
    if (LT(arr->data[i], arr->data[j])
        return i;
    return j;
}
```
Heap Representation as Dynamic Array

- Children of a node at index $i$ are stored at indices $2i + 1$ and $2i + 2$

- Parent of node at index $i$ is at index $\text{floor}((i - 1) / 2)$
void _adjustHeap(struct DynArr *heap,
    int maxIdx, int pos) {
    int leftIdx = pos * 2 + 1;
    int rghtIdx = pos * 2 + 2;
    if (rghtIdx < maxIdx) {
        /* If necessary, swap with the smallest child */
        /* and recursively call _adjustHeap() */
    }
    else if (leftIdx < max) {
        /* If necessary, swap */
        /* and recursively call _adjustHeap() */
    }
}
adjustHeap

void _adjustHeap(struct DynArr *heap, int maxIdx, int pos) {
    int leftIdx = 2*pos + 1;
    int rightIdx = 2*pos + 2;
    int smallIdx;
    if(rightIdx <= maxIdx) /*2 children */ {
        smallIdx = minIdx(leftIdx, rightIdx);
        if(LT(heap->data[smallIdx], heap->data[pos]){
            swap(heap->data, pos, smallIdx);
            _adjustHeap(heap, maxIdx, smallIdx);
        }
    } else if(leftIdx <= maxIdx) /* One child */
    if(LT(heap->data[leftIdx], heap->data[pos]) {
        swap(heap->data, pos, leftIdx);
        _adjustHeap(heap, maxIdx, leftIdx);
    }
## Priority Queues: Performance Evaluation

<table>
<thead>
<tr>
<th></th>
<th>SortedVector</th>
<th>SortedList</th>
<th>Heap</th>
<th>SkipList</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td></td>
<td>Binary search</td>
<td>Linear search</td>
<td>Percolate up</td>
<td>Add to all lists</td>
</tr>
<tr>
<td><strong>getMin</strong></td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td>get(0)</td>
<td>Returns head.val</td>
<td>Get root node</td>
<td>Get first link</td>
</tr>
<tr>
<td><strong>removeMin</strong></td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td></td>
<td>Slide data down</td>
<td>O(1): Reverse Order</td>
<td>Percolate down</td>
<td>Remove from all lists</td>
</tr>
<tr>
<td></td>
<td></td>
<td>removeFront</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

• Remember 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: $n^2 + n = \mathcal{O}(n^2)$

  – Heap:
    
    Insertions: $n \times \log n$
    
    Removals: $n \times \log n$
    
    Total time: $n \times \log n + n \times \log n = 2n \log n = \mathcal{O}(n \log n)$
Priority Queue Application: Simulation

• Original, and one of most important, applications

• Discrete event driven simulation:
  
  – Actions represented by “events” – things that have (or will) happen at a given time
  
  – Priority queue maintains list of pending events ➔ Highest priority is next event
  
  – Event pulled from list, executed ➔ often spawns more events, which are inserted into priority queue
  
  – Loop until everything happens, or until fixed time is reached
Priority Queue Applications: Example

- **Example**: Ice cream store
  - People arrive
  - People order
  - People leave

- **Simulation algorithm**:
  1. Determine time of each event using random number generator with some distribution
  2. Put all events in priority queue based on when it happens
  3. Simulation framework pulls minimum *(next to happen)* and executes event