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

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>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>( O(n) ) Binary search</td>
<td>( O(n) ) Linear search</td>
<td>( O(1) ) \texttt{addLast(obj)}</td>
</tr>
<tr>
<td></td>
<td>Slide data up</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>getFirst</strong></td>
<td>( O(1) ) \texttt{elementAt(0)}</td>
<td>( O(1) ) \texttt{head.obj}</td>
<td>( O(n) ) Linear search for smallest value</td>
</tr>
<tr>
<td></td>
<td>Slide data down ( O(1) ) \rightarrow Reverse Order</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>removeFirst</strong></td>
<td>( O(n) )</td>
<td>( O(1) ) \texttt{head.remove()}</td>
<td>( O(n) ) Linear search then remove smallest</td>
</tr>
<tr>
<td></td>
<td>Slide data down</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Which node becomes the root?
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 new root > smallest child
swap with smallest child

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

1st iteration
Maintaining the Heap: Removal (cont.)

After second iteration (moved 9 up)

Percolating down

After third iteration (moved 12 up)
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 DynArr {
    TYPE *data;
    int size;
    int capacity;
};

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

```
2
/ \    /
3   5  \\
|  / \  |
| 9 10  |
  14  11 16
```

```
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.)

The diagram shows a binary heap with the numbers 2, 3, 5, 9, 10, 4, 8, 14, 12, 11, 16, and 7. The numbers are arranged in a way that satisfies the heap property:

- The parent node is greater than or equal to its children.
- The value of each node is stored at its index (pidx) in the heap array, and the index is calculated as floor(log2(pos + 1)).
Heap Implementation: add (cont.)

Diagram of a heap with nodes and their positions in the array.
Heap Implementation: add (cont.)

```
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: Add

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

Parent position: \textbf{pidx}  
Next open spot: \textbf{pos}
Heap Implementation: add (cont.)

```c
void addHeap(struct DynArr *heap, TYPE val) {
    ...
    /* Make room for the new element */
    if (pos >= heap->cap)
        _doubleCapacity(heap, 2 * heap->cap);
    ...
}
```
Heap Implementation: add (cont.)

```c
void addHeap(struct DynArr *heap, TYPE val) {
    ...
    /* While not at root and new value is less than parent */
    if (pos > 0) parentVal = heap->data[pidx];
    while (pos > 0 && LT(val, parentVal)) {
        /* Percolate upwards */
        heap->data[pos] = parentVal;
        pos = pidx;
        pidx = (pos - 1) / 2; /* new parent index */
        if (pos > 0) parentVal = heap->data[pidx];
    }
    ...
    
    0  1  2  3  4  5  6  7  8  9  10  11
    2  3  5  9 10  4  8 14 12 11 16  7
    pidx  pos
```
Heap Implementation: add (cont.)

```c
void addHeap(struct DynArr *heap, TYPE val) {
    ...
    while (pos > 0 && LT(val, parentVal)) {
        /* percolate upwards */
    }
    heap->data[pos] = val; /* Now add new value */
    heap->size++;
}
```

```
\begin{array}{c|c}
\text{pidx} & \text{pos} \\
\hline
0 & 1 \downarrow \\
1 & 2 \downarrow \\
2 & 3 \\
3 & 4 \\
4 & 9 \\
5 & 10 \\
6 & 5 \\
7 & 8 \\
8 & 14 \\
9 & 12 \\
10 & 11 \\
11 & 16 \\
12 & 7 \\
\end{array}
```
Heap Implementation: removeMin (cont.)

The heap is represented as a binary tree, and the removal of the minimum element is illustrated by swapping the last element (7) with the root (2) and then heapifying the tree.

The array representation of the heap is shown below, with the root element highlighted and the last element marked for removal.
Heap Implementation: removeMin (cont.)

The diagram shows the process of removing the minimum element from a heap. The root node initially contains the value 7. After removing the minimum, the new root node is selected. The new root node is chosen to maintain the heap property, ensuring that the parent node is always less than or equal to its children if it's a min heap.

The array below the heap illustrates the values before and after the removal, with the new root marked by an arrow pointing downward.

New root
**Heap Implementation: removeMin (cont.)**

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

Smallest child (min = 3)

val = 7
Heap Implementation: `removeMin` (cont.)

**val > min** → Copy min to parent spot (`idx`)  
Move `idx` to child

```
val = 7
```

```
max
```

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

val < min \( \rightarrow \) Copy val into idx spot
Done

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

val < min → Copy val into idx spot
Done

val = 7

idx
Heap Implementation: removeMin

```c
void removeMinHeap(struct DynArr *heap) {
    assert(heap);
    int last = heap->size - 1;
    if (last != 0) /* Copy the last element to the first */
        heap->data[0] = heap->data[last];
    heap->size--; /* Remove last */
    ...
```

First element

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>10</td>
<td>5</td>
<td>8</td>
<td>14</td>
<td>12</td>
<td>11</td>
<td>16</td>
<td>7</td>
</tr>
</tbody>
</table>

Last element
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 $\lfloor (i - 1) / 2 \rfloor$
Recursive \texttt{adjustHeap} \\

\begin{verbatim}
void _adjustHeap(struct DynArr *heap,
                 int maxIdx, int pos) {
    assert(heap);
    int leftIdx = pos * 2 + 1; // left child index
    int rightIdx = pos * 2 + 2; // right child index
    if (rightIdx < maxIdx) {  // there are 2 children
        /* swap with the smaller child if it is smaller than me */
    }
    else if (leftIdx < maxIdx) {  // there is 1 child
        /* swap if the left child is smaller than me */
    }
    /* else no children, done */
}
\end{verbatim}
_adjustHeap

```c
void _adjustHeap(struct DynArr *heap,
                 int maxIdx, int pos) {
    ...
    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 */
    ...
}
```
void _adjustHeap(struct DynArr *heap,

    int maxIdx, int pos) {

    ...

    if (rightIdx <= maxIdx) { /* 2 children */
        ...
    }

    } else if (leftIdx <= maxIdx) { /* One child */
    
        if (LT(heap->data[leftIdx], heap->data[pos])){
            _swap(heap->data, pos, leftIdx);
            _adjustHeap(heap, maxIdx, leftIdx);
        }

    }

}
void _swap(struct DynArr *da, int i, int j)
{
    /* Swap elements at indices i and j */
    TYPE tmp = da->data[i];
    da->data[i] = da->data[j];
    da->data[j] = tmp;
}
Useful Routines: minIdx

int _minIdx(struct DynArr *da, int i, int j)
{
    /* Return index of the smaller element */
    if (LT(da->data[i], da->data[j]))
        return i;
    return j;
}
## Priority Queues: Performance Evaluation

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

Which implementation is the best for 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: \( n^2 + n = O(n^2) \)
  
  – Heap:
    
    Insertions: \( n \times \log n \)
    
    Removals: \( n \times \log n \)
    
    Total time: \( n \times \log n + n \times \log n = 2n \log n = O(n \log n) \)
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 \(\rightarrow\) Highest priority is next event

  – Event pulled from list, executed \(\rightarrow\) 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