Priority Queues

Sorting

cs206

April 20
Upheap

- Restore heap order
  - swap upwards
  - stop when finding a smaller parent
  - or reach root
- $O(\log n)$
Downheap

- Restore heap order
  - swap downwards
  - swap with smaller child
  - stop when finding larger children
  - or reach a leaf
- $O(\log n)$
Numbering a Heap

- Can easily associate a number with each position in heap
- Root = 0
- Row 1
  - left = 1
  - right = 2
- Row 2
  - leftmost = 3
  - rightmost = 6
- Row 3
  - leftmost = $2^3 - 1 = 7$
  - rightmost = $2^4 - 2 = 14$
- Row N
  - leftmost = $2^{N-1}$
  - rightmost = $2^{(N+1)} - 2$
Heaps are built on Arrays

Numbering directly leads to storage position in array

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>9</td>
<td>7</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Locations of Parents and children are in strict mathematical relationship

- Parent from child
  - suppose child is at location childLoc in array
    - parentLoc = (childLoc-1)/2
- Child from Parent
  - suppose parent is at parentLoc in array
    - leftChild = parentLoc*2+1
    - rightChild = parentLoc*2+2

- Parent from child
  - child at loc 4 (value 7)
    - parent is at (4-1)/2 = 1 (value 5)
- Child from Parent
  - parent at loc 2 (value 6)
    - leftChild =2*2+1 = 5 (value 1)
    - rightChild = 2*2+2 = 6 (value — not used)
public class PriorityQHeap<K extends Comparable<K>, V> extends AbstractPriorityQueue<K, V> {

    private static final int CAPACITY = 1032;
    private Pair<K, V>[] backArray;
    private int size;

    public PriorityQHeap() {
        this(CAPACITY);
    }

    public PriorityQHeap(int capacity) {
        size=0;
        backArray = new Pair[capacity];
    }

    @Override
    public int size()
    {
        return size;
    }

    @Override
    public boolean isEmpty()
    {
        return size==0;
    }
}
public boolean offer(K key, V value) {
    if (size >= (backArray.length - 1))
        return false;
    // put new item in at end data items
    int loc = size++;
    backArray[loc] = new Pair<K, V>(key, value);
    // up heap
    int upp = (loc - 1) / 2; // the location of the parent
    while (loc != 0) {
        if (0 > backArray[loc].compareTo(backArray[upp])) {
            // swap and climb
            Pair<K, V> tmp = backArray[upp];
            backArray[upp] = backArray[loc];
            backArray[loc] = tmp;
            loc = upp;
            upp = (loc - 1) / 2;
        } else {
            break;
        }
    }
    return true;
}
@Override
public V poll() {
    if (isEmpty())
        return null;
    Entry<K,V> tmp = backArray[0];
    removeTop();
    return tmp.theV;
}

@Override
public V peek() {
    if (isEmpty())
        return null;
    return backArray[0].theV;
}
private void removeTop()
{
    backArray[0] = backArray[size-1];
    backArray[size-1]=null;
    size--;
    int upp=0;
    while (true)
    {
        int dwn;
        int dwn1 = upp*2+1;
        if (dwn1>size) break;
        int dwn2 = upp*2+2;
        if (dwn2>size) {  dwn=dwn1;
                        } else {
            int cmp = backArray[dwn1].compareTo(backArray[dwn2]);
            if (cmp<=0)  dwn=dwn1;
            else  dwn=dwn2;
        }
        if (0 > backArray[dwn].compareTo(backArray[upp]))
        {
            Pair<K,V> tmp = backArray[dwn];
            backArray[dwn] = backArray[upp];
            backArray[upp] = tmp;
            upp=dwn;
        } else {  break;  }
    }
}
General Removal

- swap with last node
- delete last node
- may need to upheap or downheap

Heap:

1 (root) -> 5 (parent) -> 9 (left child) 11 (right child)

1 (root) -> 6 (left child) 8 (right child) -> 15 (right child)

1 (root) -> 9 (left child) 11 (right child) -> 17 (right child)

1 (root) -> 22 (left child) 33 (right child) -> 27 (right child)

Delete node 5 and replace with last node 9.

Delete node 33 and replace with last node 22.
## Complexity Analysis

<table>
<thead>
<tr>
<th></th>
<th>Unordered</th>
<th>Ordered (using SAL)</th>
<th>Heap Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>offer</td>
<td>O(1)</td>
<td>O(N)</td>
<td></td>
</tr>
<tr>
<td>peek</td>
<td>O(N)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>poll</td>
<td>O(N)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>Offer N then poll N</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sorting

- public void sort(Comparable[] arra)
  - change the order of the items in arra
  - All examples will use integers but same statements apply to any Comparable object
  - ideally, do this “in place”.
    - That is do not use any extra memory

- First 3 sort techniques we have already discussed
• PQ on unordered == Selection Sort
• PQ on ordered == Insertion Sort
• PQ on Heap == Heap Sort
Selection Sort

- **Selection-sort:**
  - in place algorithm given an array with N items:
    - step 1: find the min from 0..(N-1) in array and swap with item in position 0
    - step 2: find min from 1..(N-1) in array and swap with item in position 1.
    - etc
  - priority queue implemented with an unsorted array / arrayList / ...

- **Time:**
  - $O(n^2)$
    - In terms of priority $Q$, can split this into two phases
      - insertion == $O(N)$
      - polling == $O(N^2)$
Example

Phase 1 — Inserting

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>(7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td>4</td>
<td>(7,4)</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(g)</td>
<td>()</td>
<td>(7,4,8,2,5,3,9)</td>
</tr>
</tbody>
</table>

Phase 2 — Polling

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>(7,4,8,5,3,9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(2)</td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td>(2,3)</td>
<td>(7,4,8,5,9)</td>
</tr>
<tr>
<td>(c)</td>
<td>(2,3,4)</td>
<td>(7,8,5,9)</td>
</tr>
<tr>
<td>(d)</td>
<td>(2,3,4,5)</td>
<td>(7,8,9)</td>
</tr>
<tr>
<td>(e)</td>
<td>(2,3,4,5,7)</td>
<td>(8,9)</td>
</tr>
<tr>
<td>(f)</td>
<td>(2,3,4,5,7,8)</td>
<td>(9)</td>
</tr>
<tr>
<td>(g)</td>
<td>(2,3,4,5,7,8,9)</td>
<td>()</td>
</tr>
</tbody>
</table>
Insertion Sort

- Insertion-sort
  - in-place algorithm
    - Step 0: start with item in position 0. Now the items in positions 0..0 are sorted
    - Step 1: look at item in position 1. Compare it to item in 0. If p1 is smaller, then swap. The items in position 0..1 are sorted with respect to each other
    - Step 2: determine where item in p2 should go in sorted list 0..N. If needed, for instance, bigger than 0 but smaller than 1. Make a space: save p1 into tmp. Shifting p1 into p2. Then put tmp into p1. Now the item in 0..2 are sorted.
    - Step N:
      - insert/swap the element into the correct sorted position
  - Priority queue implemented with a sorted array/ArrayList / ...
  - Time:
    - $O(n^2)$
    - In terms of PQ
      - Add: $O(n^2)$
      - Remove: $O(n)$
    - Faster than selection sort
**Example**

<table>
<thead>
<tr>
<th>Phase 1 — Inserting</th>
<th></th>
<th>Phase 2 — polling</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) 7</td>
<td>(7)</td>
<td>(a) (2) (3,4,5,7,8,9)</td>
</tr>
<tr>
<td>(b) 4</td>
<td>(4,7)</td>
<td>(b) (2,3) (4,5,7,8,9)</td>
</tr>
<tr>
<td>(c) 8</td>
<td>(4,7,8)</td>
<td></td>
</tr>
<tr>
<td>(d) 2</td>
<td>(2,4,7,8)</td>
<td></td>
</tr>
<tr>
<td>(e) 5</td>
<td>(2,4,5,7,8)</td>
<td></td>
</tr>
<tr>
<td>(f) 3</td>
<td>(2,3,4,5,7,8)</td>
<td></td>
</tr>
<tr>
<td>(g) 9</td>
<td>(2,3,4,5,7,8)</td>
<td></td>
</tr>
</tbody>
</table>

|   |   |   |
|   |   |   |
|   |   |   |

|   |   |   |
| (g) (2,3,4,5,7,8,9) |   | (2,3,4,5,7,8,9) | () |
Heap Sort

- Heap-sort:
  - Insertion — no more than \( \log_2(n) \) steps per insertion
  - Deletion — no more than \( \log_2(n) \) steps per deletion

- Priority queue is implemented with a heap

- Time:
  - Add: \( O(n \times \log_2(n)) \) — under some assumptions \( O(n) \).
  - Remove: \( O(n \times \log_2(n)) \)
  - Note: with a lot of work can do this without an additional array.
## Example

Phase 1 — Inserting

| (a) | 7  | (7)   |
| (b) | 4  | (4,7) |
| (c) | 8  | (4,7,8) |
| (d) | 2  | (2,4,8,7) |
| (e) | 5  | (2,4,8,7,5) |
| (f) | 3  | (2,4,3,7,5,8) |
| (g) | 9  | (2,4,3,7,5,8,9) |

Phase 2 — polling

| (a) | (2) | (3,4,7,5,8,9) |
| (b) | (2,3) | (4,5,7,9,8) |
| .. | .. | .. |
| (g) | (2,3,4,5,7,8,9) | () |
## Timing

<table>
<thead>
<tr>
<th>size</th>
<th>selection</th>
<th>Insertion</th>
<th>Insertion</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>16</td>
<td>15</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>2000</td>
<td>8</td>
<td>12</td>
<td>26</td>
<td>3</td>
</tr>
<tr>
<td>4000</td>
<td>24</td>
<td>23</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>8000</td>
<td>96</td>
<td>95</td>
<td>81</td>
<td>10</td>
</tr>
<tr>
<td>16000</td>
<td>370</td>
<td>378</td>
<td>315</td>
<td>17</td>
</tr>
<tr>
<td>32000</td>
<td>1585</td>
<td>1359</td>
<td>1218</td>
<td>36</td>
</tr>
<tr>
<td>64000</td>
<td>5771</td>
<td>5590</td>
<td>4605</td>
<td>77</td>
</tr>
<tr>
<td>128000</td>
<td>23087</td>
<td>21547</td>
<td>19849</td>
<td>161</td>
</tr>
<tr>
<td>256000</td>
<td></td>
<td></td>
<td></td>
<td>345</td>
</tr>
<tr>
<td>512000</td>
<td></td>
<td></td>
<td></td>
<td>1128</td>
</tr>
<tr>
<td>1024000</td>
<td></td>
<td></td>
<td></td>
<td>1973</td>
</tr>
<tr>
<td>2048000</td>
<td></td>
<td></td>
<td></td>
<td>3225</td>
</tr>
<tr>
<td>4096000</td>
<td></td>
<td></td>
<td></td>
<td>7577</td>
</tr>
<tr>
<td>8192000</td>
<td></td>
<td></td>
<td></td>
<td>18586</td>
</tr>
</tbody>
</table>

10000==1 second
Practice

- Show the array after each step of selection sort, insertion sort and heap sort for the following list
- for insertion and selection sort, use the in-place algorithms discussed
- For heap sort, show the array after each addition to the heap and each removal from the heap.
- Count the number of operations you did for each

9, 4, 1, 3, 12, 17, 2, 8, 20, 5
Breakpoints and using them in VSC

- Better than Print statements!!!!!!
- Idea: set a place (or places) when your program will pause.
- When paused:
  - look at variable values
  - look at the stack
    - continue
    - stepwise
Lab

- Open VSC on some program you wrote
- Set a breakpoint
- Run program
- When hit breakpoint
  - Get screen to show variable values and the stack
  - Take picture and send the picture to me