Introduction to Priority Queues

• Want to maintain a collection of $n$ items to support $\text{findMin}$, $\text{insert}$ and $\text{deleteMin}$ in $O(\log n)$ time
  – Note that we are $\textit{not}$ interested in doing arbitrary $\text{remove()}$ or $\text{find()}$
  – A linked list or array requires that some operations take $O(n)$ time
  – An unbalanced binary search tree does $\textit{not}$ have a good worst case. A balanced tree (or even splay trees) is too much work

• A binary heap is implemented like an array and in addition supports $\text{insert}$ in $O(1)$ expected time, and $\text{findMin}$ in worst-case $O(1)$ time.

• Excellent for general hierarchical searches, and possibly sorting
What Is A Heap?

- A heap is a complete binary tree that stores keys in internal nodes with the order property $\text{key(parent)} \leq \text{key(child)}$.

- Height of the tree is logarithmic.
- With $n$ keys, a heap can be constructed in $O(n)$ time (stored in an array of size $n$).
Insertion Into Heap: Percolate Up

1. Create a hole in the next available location. If key can be placed in the hole, done.
2. Otherwise, percolate the key into its parent and recurse (till root).
3. Efficient algorithm by placing $-\infty$ at position 0 of the heap to avoid testing of root.
Example

![Diagram](image.png)
Example

\[
\begin{array}{c}
\text{(a)} \\
\begin{array}{c}
14 \\
13 \\
16 \\
24 \\
21 \\
19 \\
68 \\
65 \\
26 \\
32 \\
31 \\
\end{array}
\end{array}
\quad \quad \quad
\begin{array}{c}
\text{(b)} \\
\begin{array}{c}
14 \\
13 \\
16 \\
24 \\
21 \\
19 \\
68 \\
65 \\
26 \\
32 \\
31 \\
\end{array}
\end{array}
\]
Remove Minimum From Heap: Percolate Down

- Key at root is removed creating a hole
- Conceptually move last key in array to root
- Percolate down so that order property is satisfied
Example

Min = 13

13

14 16

19 19 68

65 26 32 31 65 26 32 31
Example
Example
Application of Heaps: Nearest Neighbor

• Given hierarchical data structure containing points (in general data objects) and a query object
• We want to output the data object closest to the query object
• Approach: Recursive travel of the data structure using a priority queue to prune the search space
  – Q contains objects with a lower bound
  – Want to insert and remove items from Q
  – We stop searching if current cost is less than the lower bound stored in Q
Nearest Neighbor: Pruned version

```java
float lowerbound(Rectangle r, Point p) {
    if (r.inside(p)) return 0;
    if (r.left(p)) return r.minX - p.x;
    if (r.right(p)) return p.x - r.maxX;
    if (r.southEastCorner(p)) ...
}
```

```java
Result process(KDNode k, int cd, Rectangle r, Result res) {
    if (k == null) return res;
    if (lowerbound(r, q) >= res.distance) return res;
    float dist = distance(node.data, query);
    if (dist < res.distance) {
        result.point = node.data;
        result.distance = distance(node.data, query);
    }
    if (q[cd] < k.data[cd]) ... else ...
    return res;
}
```
Nearest Neighbor: Pruned version

Result process(KDNode k, int cd, Rectangle r, Result res) {
    float dist = distance(node.data, query);
    if (dist < res.distance) {
        result.point = node.data;
        result.distance = distance(node.data, query);
    }
    if (q[cd] < k.data[cd]) {
        res = process(k.left, cd+1, r.trimLeft(cd, k.data), res);
        res = process(k.right, cd+1, r.trimRight(cd, k.data), res);
    } else {
        res = process(k.right, cd+1, r.trimRight(cd, k.data), res);
        res = process(k.left, cd+1, r.trimLeft(cd, k.data), res);
    }
    return res;
}
Which Nodes Are Processed?

![Graph with nodes labeled with coordinates such as (35,90), (10,75), (70,80), (25,10), (80,40), (50,90), (20,50), (70,30), (90,60), (50,25), (60,10).]
Any Improvement Possible?

- Nine nodes examined. Can we do better?
- Yes, if decision to pick a node is based on dynamic changing costs instead of initial left-right decision.
- Use a priority queue. (Seven nodes examined instead of nine)
General Structure Of Algorithm

- Elements in Q are pairs (node, lowerbound)
- Initialize:
  - Insert (root,0) into Q
  - Nearest object = NULL, and distance = $\infty$
- Body of the algorithm: While Q is not empty and $\text{findmin}(Q).\text{cost} < \text{result.distance}$
  1. Node = deleteMin(Q);
  2. If (isInternal(node)) insertQ(child, lowerbound(child))
  3. Otherwise, process(node)
Nearest Neighbor: Priority Q Version

Q contents

- \([35, 90), 0]\); !
- \([70, 80), 0],
  \([10, 75), 5]\);
- \([80, 40), 0],
  \([10, 75), 5],
  \([50, 90), 30]\);
- \([70, 30), 0], \ldots ,
  \([90, 60), 40]\); !
- \([10, 75), 5],
  \([50, 25, 20], \ldots ;
- \([25, 10), 5], \ldots ,
- \([20, 50), 15], \ldots , !
Nearest Neighbor: Q version

```
insertQ(KDNode n, float cost, int cd) {
    // insert into a priority Q based on cost.
    // store cutting dimension and node too
}
init() {
    result.distance = infinity; result.point = null
    insertQ(T.root, 0, 0);
    process(Q, result);
}
Result processQ(Queue Q, Result res) {
    if (null(Q)) return res;
    int cost = Q.extractMin().cost; int cd = Q.extractMin().cd;
    KDNode node = Q.deleteMin();
    if (cost > res.distance) return res;
    ...
}
```
Nearest Neighbor: Q version

```java
Result processQ(Queue Q, Result res) {
    KDNode node = Q.deleteMin();
    if (cost < res.distance) {
        result.point = node.data;
        result.distance = distance(node.data, query);
    }
    left.region = intersect(node.region, lefthalf(node.data), cd);
    cost = lowerbound(left.region, query);
    Q.insert(node.left, cost, cd+1);
    right.region = intersect(node.region, righthalf(node.data), cd);
    cost = lowerbound(right.region, query);
    Q.insert(node.right, cost, cd+1);
    return processQ(Q, res);
}
```