Heaps, Heapsort, and Priority Queues

Analysis of Algorithms
Binary Trees

Definition: a binary tree $T$ is **full** if each node is either a leaf or possesses exactly two child nodes.

Definition: a binary tree $T$ with $n$ levels is **complete** if all levels except possibly the last are completely full, and the last level has all its nodes to the left side.
Binary Heaps

- A heap is a complete binary tree
  - Perfectly balanced, except perhaps for the bottom level
  - The height of complete tree with $N$ nodes is $\lfloor \log N \rfloor$
  - The height only increases when $N$ is a power of 2
Binary Heaps

• The *heap order property*:
  – For each subtree, the key stored at the root has higher priority than all of the other keys stored in the subtree

• There is no ordering between the children of any node other than that they are both lower priority than their parent

• The highest priority key stored in a heap will be in the root and the lowest priority key will be in one of the leaves
Binary Heaps

• What are heaps used for?
  – The typical use of a heap is as a data structure for *quickly* and *repeatedly* finding the maximum (or minimum) value in an array
  – Heaps are used to implement *priority queues*

• For example, consider this sorting algorithm:
  – Insert all elements into an array
  – Repeat for all elements:
    • Extract the maximum priority element from a list
    • Place this maximum element at the beginning of another list
Binary Heaps

- A heap is implemented as an array, usually with the first valid index located at position 1.
- For a key at location $k$, its left child will be in location $(2k)$ and its right child will be in location $(2k)+1$.
- The parent of $k$ will be in location floor($k/2$).
- If $(2k)$ is greater than the number of elements, then the key at location $k$ is a leaf.
An Example of a Heap

Number of elements = 15
Promotion in a Heap

- It is a violation of the heap order property if a child has higher priority than its parent.
- To eliminate this violation:
  - Exchange child with parent
  - Repeat until heap order is restored
  - This is called a “swim” up operation

```java
private void swim(int k)
{
    while (k > 1 && less(k/2, k))
    {
        exch(k, k/2);
        k = k/2;
    }
}
```
Demotion in a Heap

- It is also a violation of the heap order property if the parent has lower priority than either of its children
  - Exchange parent with the higher priority child
  - Repeat until heap order restored
  - This is a “sink” down operation

```java
private void sink(int k) {
    while (2*k <= N) {
        int j = 2*k;
        if (j < N && less(j, j+1)) j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```
Insertion into a Heap

• To add a node to a heap:
  – Insert the node at end of array (rightmost leaf), then “swim” it up

• Cost:
  – At most $1 + \lceil \log N \rceil$ compares

```java
public void insert(Key x) {
    pq[++N] = x;
    swim(N);
}
```
Deletion from a Heap

• To remove the highest priority node from the heap:
  – Remove the root (it always has the highest priority), replace it with the last node and sink down

• Cost:
  – At most $\lceil \log N \rceil$ compares
Building a Heap

- To create a heap from an array ("heapify" the array), start by looking at the left half of the array (the non-leaves):
Building a Heap

• Of those nodes, look at the right half (these are the nodes one level up from the leaves):
Building a Heap

• Of these nodes, swap with child node if a child has a higher priority (here, high values correspond to a high priority):

After 3 swaps:
Building a Heap

• Repeat with the next higher level:
Building a Heap

- Swapping as necessary:

```
   9  29  25  19  21  18  17  13  6  5  2  4  12  16  8
```

```
0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
```

```
1
 /\     /
2 29  19 21
 /\     /
3 4
 /\ 5
19
 /\ 6
8 9
```

```
1
 /\     /
3 25  17
 /\     /
4 6
 /\ 7
18
 /\ 8
13
```

```
2
 /\ 11
10
```

```
5
 /\ 12
2
```

```
13
```

```
14
```

```
15
```
Building a Heap

- Finally, consider the root:
Building a Heap

- And again, recursively swapping as necessary:
Building a Heap

- This is now a valid heap:
Analysis of Build-Heap

• Building a heap of size $N$ using this bottom-up method takes time proportional to $N$
  – This is because most of the nodes are at low levels in the tree (near the leaves), so not a lot of swaps
Heapsort

- This is a valid heap, but the array is not sorted
- Can we sort an array using heap operations?
Heapsort

1. Start with a valid heap
2. Repeatedly swap the root with the last element in the array
3. Fix up the heap so that it is valid again
Heapsort
Heapsort

The diagram illustrates a heap, which is a complete binary tree where each node is greater than or equal to its children. The root node is 21, and the tree is colored to show the order in which elements are removed during heapsort. The numbers in the diagram represent the order in which elements are removed, starting with 21 and ending with 29.
Heapsort

```
0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  8 19 18 16  9 12 17 13  6  5  2  4 21 25 29
```

Diagram:

```
Heapsort

0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  8 19 18 16  9 12 17 13  6  5  2  4 21 25 29

1

2

19

3

18

4  5  6  7

16  9 12 17

8  9 10 11

13  6  5  2

8  9 10 11

12  13  14  15

21  25  29
```
Heapsort
Heapsort
Heapsort
Heapsort
Heapsort
Heapsort
Heapsort
Heapsort
Heapsort
Heapsort
Heapsort

12 9 5 8 6 2 4 13 16 17 18 19 21 25 29

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

12

2

9

4 5

8 6

13 16 17 18 19 21 25 29

3

5

6

2

12 9 5 8 6 2 4 13 16 17 18 19 21 25 29

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Heapsort
Heapsort
Heapsort
Heapsort
Heapsort
Heapsort
Heapsort
Heapsort

A heap is a nearly complete binary tree in which each node's value is greater than or equal to (in a max heap) or less than or equal to (in a min heap) the values of its children. Heapsort is a comparison-based sorting algorithm that uses a binary heap data structure.

The algorithm begins by constructing a max heap from the input data. Then, the largest element is swapped with the last element, and the heap size is reduced by one. The new root is then heapified to maintain the heap property. This process is repeated until the heap size is reduced to one, resulting in a sorted array.

The diagram above shows a binary heap with values in increasing order from top to bottom and left to right. The root node contains the largest value, and the heap property is maintained throughout the tree.
Heapsort
Heapsort
Heapsort
Heapsort
Heapsort Implementation

```java
public class Heap {
    public static void sort(Comparable[] a) {
        int N = a.length;
        for (int k = N/2; k >= 1; k--)
            sink(a, k, N);
        while (N > 1) {
            exch(a, 1, N);
            sink(a, 1, --N);
        }
    }

    private static void sink(Comparable[] a, int k, int N) {
        /* as before */
    }

    private static boolean less(Comparable[] a, int i, int j) {
        /* as before */
    }

    private static void exch(Comparable[] a, int i, int j) {
        /* as before */
    }
}
```

but convert from 1-based indexing to 0-base indexing
Analysis of Heapsort

• The bottom-up construction of a heap with $N$ items takes time proportional to $N$

• Sorting array takes time proportional to $N \log N$

• Heapsort is an **in-place** sorting algorithm
  - Mergesort: not in-place, needs linear extra space
  - Quicksort: in-place, quadratic time in worst case

• Heapsort is optimal for both time and space, but
  - The inner loop takes longer than quicksort’s
  - Makes poor use of cache memory (not good for large arrays)
  - Heapsort is not a stable sort
The Priority Queue

• Goal: Find the largest $M$ items in a stream of $N$ items
  – Example: Fraud detection (isolate $$ transactions)
  – Example: File maintenance (find biggest files or directories)

• Constraint: What if you do not have enough memory to store all $N$ items?
  – Sorting will be difficult if you cannot store all items
  – Solution: store only the *largest* items in a priority queue using a binary heap ($N \log N$ time and $M$ space)
Priority Queue Considerations

• Issues:
  – Checking for underflow and overflow
    • Underflow: throw exception if deleting from empty PQ
    • Overflow: add no-arg constructor and resize the array
  – Minimum-oriented priority queue
    • Replace less() with greater()
  – Updating priorities
    • How can we change the priority of an item (*update its priority*) in light of new information?
Analysis of Priority Queue

• The efficiency of updating an item or adding a new item to a priority queue that has N items varies according to how it is implemented:

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Insert</th>
<th>Delete Max</th>
<th>Find Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unordered array</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Ordered array</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Binary heap</td>
<td>log N</td>
<td>log N</td>
<td>1</td>
</tr>
<tr>
<td>d-ary heap</td>
<td>log_d N</td>
<td>d log_d N</td>
<td>1</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>1</td>
<td>log N</td>
<td>1</td>
</tr>
<tr>
<td>Impossible</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Prim Using a Priority Queue

• Recall that for Prim’s algorithm we repeatedly have to select the next vertex that is closest (has the smallest edge weight) to the tree built so far.

• Therefore we need some sort of data structure that enables us to associate a value (the distance from the vertex to the tree under construction) with each vertex, and then select the vertex with the lowest value (the highest priority).

• The solution is to use a priority queue implemented as a min heap.
Prim Using a Priority Queue

• What is the current priority of \( v \)?
• What is the current priority of \( w \)?
• Which vertex (\( v \) or \( w \)) will now join the MST?
• What is the priority of \( w \) after \( v \) joins the MST?
Prim Using a Priority Queue

• It is now easy to see how to implement Prim’s algorithm
• First, initialize a priority queue (an array) to contain every vertex with equal priority, $\infty$
• Then a source vertex, $s$, is chosen and its priority is changed to 0
• Remember: we want low key values to represent high priorities, so we should use a min heap instead of a max heap
Prim Using a Priority Queue

At each stage of the algorithm we extract the vertex $u$ with the highest priority (that is, the lowest weight). We then examine the neighbors of $u$. For each neighbor $v$, there are two possibilities:

1. If $v$ is not in the queue, then it is already in the spanning tree being constructed and do not consider it further.

2. If $v$ is currently in the queue, then check whether this new edge $(u,v)$ will cause an update in the priority of $v$. If the weight of $(u,v)$ is less than the current priority of $v$, then change the priority of $v$ to be the weight of $(u,v)$ and set the parent$(v) = u$. 
Prim Using a Priority Queue

Insert all vertices into the priority queue, PQ
Initialize all priorities to \( \infty \), select a source, and change priority of source to 0
heapify the PQ (i.e., create a min heap)
while PQ ≠ empty
    \( u = \text{deleteMin}(PQ) \)
    for each \( v \) adjacent to \( u \) do
        if \( v \in PQ \) and weight \((u,v)\) < \( v.\text{getPriority}(\) )
            \( v.\text{parent} = u \)
            \( v.\text{setPriority}(\text{weight}\ (u,v)) \)
        end if
    end for
re-heapify the PQ
end while

* It is important to notice how the parent array is managed:
  for every vertex \( v \in PQ \) with a finite key value, parent[\( v \)] is the vertex not in the PQ that was responsible for the key of \( v \) reaching the highest priority (lowest weight) it has currently reached.
Prim Using a Priority Queue

Suggestion: for each vertex $v \in PQ$, store:

1. The ID number of $v$
2. The weight of the smallest edge (the priority) from a vertex $u$ in the tree to $v$, and
3. The tree vertex, $u$, that is on the other side of this edge

For a sparse graph, it is important to keep track of where every vertex is in the heap, so that updating its priority will be fast.
Prim Using a Priority Queue

• For fast update, keep a separate array QP, of locations in the heap.
• Example: Suppose you have 12 vertices with IDs from 0 – 11.

Prim tree (red) so far:

Heap:

PQ

QP

ID

0                  1                  2                  3                   4                 5                   6                 7                   8

Update to (10, 4, 5)
V₁₀ is in PQ[QP[10]]
Event-Driven Simulation Using a Priority Queue

• Goal: Simulate the motion of $N$ moving particles that behave according to the laws of elastic collision

• Collision prediction:
  • Given the position, velocity, and radius of a particle, when will it collide next with a wall or another particle?

• Collision resolution:
  • When a collision occurs, \textit{update} the colliding particles according to the laws of elastic collisions
Event-Driven Simulation Using a Priority Queue

• Discretize the time in quanta of size $dt$
• Update the position of each particle after every $dt$ units of time, and check for overlaps
• If there is an overlap (a collision), roll back the clock to the time of the collision, update the velocities of the colliding particles, and continue the simulation
Event-Driven Simulation Using a Priority Queue

• Detecting a collision:
  – $N^2/2$ overlap checks per time quantum, $dt$
  – If $dt$ is very small, the simulation will be very slow
  – If $dt$ is too large, some collisions might be missed

• Solution: use a priority queue to detect collisions and update the simulation after a collision
Event-Driven Simulation Using a Priority Queue

• Change state only when something happens
  – Assume that between collisions, particles move in straight-line trajectories
  – Maintain PQ of collision events, prioritized by time
    • The PQ is a list of all the possible collisions that could happen in the future, based on current trajectories
    • Removing the min value from the PQ is equivalent to simulating the collision
    • When a collision does occur, deal with that one (roll-back), predict any future collisions that result, and insert these new events into the priority queue
Event-Driven Simulation Using a Priority Queue

• Initialization:
  – Fill up the PQ with all potential particle-wall and particle-particle collisions (quadratic time – done only once)
  – A “priority” in the PQ represents the time to collision (along with the IDs of the two particles involved in the collision)

• Main loop:
  – Delete the impending collision (highest priority) from PQ
  – Update the velocities of the colliding particles along with the priorities of other particles they might collide with
  – Advance all particles to time $t$, on a straight-line trajectory