Heaps and Priority Queues

Priority Queue

- A priority queue (PQ) is a collection in which each item has an associated number known as a priority.
  - use a higher priority for items that are "more important"

- Example application: scheduling a shared resource like the CPU
  - give some processes/applications a higher priority, so that they will be scheduled first and/or more often

- Key operations:
  - insert: add an item (with a position based on its priority)
  - remove: remove the item with the highest priority

- One way to implement a PQ efficiently is using a type of binary tree known as a heap.
Complete Binary Trees

- A binary tree of height $h$ is complete if:
  - levels 0 through $h - 1$ are fully occupied
  - there are no "gaps" to the left of a node in level $h$

- Complete:

- Not complete (○ = missing node):

Representing a Complete Binary Tree

- A complete binary tree has a simple array representation.
  - The nodes of the tree are stored in the array in the order in which they would be visited by a level-order traversal (i.e., top to bottom, left to right).

- Examples:
Navigating a Complete Binary Tree in Array Form

- The root node is in $a[0]$
- **Given the node in $a[i]$:**
  - its left child is in $a[2*i + 1]$
  - its right child is in $a[2*i + 2]$
  - its parent is in $a[(i - 1)/2]$ (using integer division)

- Examples:
  - the left child of the node in $a[1]$ is in $a[2*1 + 1] = a[3]$
  - the right child of the node in $a[3]$ is in $a[2*3 + 2] = a[8]$
  - the parent of the node in $a[4]$ is in $a[(4 - 1)/2] = a[1]$
  - the parent of the node in $a[7]$ is in $a[(7 - 1)/2] = a[3]$

What is the left child of 24?

- Assume that the following array represents a complete tree:

```
  0 1 2 3 4 5 6 7 8
  26 12 32 24 18 28 47 10 9
```
Heaps

- Heap: a complete binary tree in which each interior node is greater than or equal to its children

Examples:

- The largest value is always at the root of the tree.
- The smallest value can be in any leaf node - there's no guarantee about which one it will be.
- We're using max-at-top heaps.
  - can also have a min-at-top heap, with every interior node <= its children

Which of these is a heap?

- A. B. C. D. more than one (which ones?)
- E. none of them
How to Compare Objects

• We need to be able to compare items in the heap.

• If those items are objects, we can't just do something like this:
  
  ```java
  if (item1 < item2)
  ```

  Why not?

• Instead, we need to use a method to compare them.

An Interface for Objects That Can Be Compared

• The `Comparable` interface is a built-in generic Java interface:

  ```java
  public interface Comparable<T> {
      public int compareTo(T other);
  }
  ```

• It is used when defining a class of objects that can be ordered.

• Examples from the built-in Java classes:

  ```java
  public class String implements Comparable<String> {
      ...
      public int compareTo(String other) {
          ...
      }
  }
  public class Integer implements Comparable<Integer> {
      ...
      public int compareTo(Integer other) {
          ...
      }
  }
  ```
An Interface for Objects That Can Be Compared (cont.)

```java
public interface Comparable<T> {
    public int compareTo(T other);
}
```

- `item1.compareTo(item2)` should return:
  - a negative integer if `item1` "comes before" `item2`
  - a positive integer if `item1"comes after" item2`
  - 0 if `item1` and `item2` are equivalent in the ordering

- These conventions make it easy to construct appropriate method calls:

```
numeric comparison                          comparison using compareTo
item1 < item2                                item1.compareTo(item2) < 0
item1 > item2                                item1.compareTo(item2) > 0
item1 == item2                               item1.compareTo(item2) == 0
```

Heap Implementation

```java
public class Heap<T extends Comparable<T>> {
    private T[] contents;
    private int numItems;
    public Heap(int maxSize) {
        contents = (T[])new Comparable[maxSize];
        numItems = 0;
    }
}
```

- Heap is another example of a generic collection class.
  - as usual, `T` is the type of the elements
  - extends `Comparable<T>` specifies `T` must implement `Comparable<T>`
  - must use `Comparable` (not `Object`) when creating the array
Heap Implementation (cont.)

public class Heap<T extends Comparable<T>> {
    private T[] contents;
    private int numItems;
    ...
}

The picture above is a heap of integers:
Heap<Integer> myHeap = new Heap<Integer>(20);
• works because Integer implements Comparable<Integer>
• could also use String or Double

Removing the Largest Item from a Heap

• Remove and return the item in the root node.
• In addition, we need to move the largest remaining item to the root, while maintaining a complete tree with each node >= children
• Algorithm:
  1. make a copy of the largest item
  2. move the last item in the heap to the root
  3. “sift down” the new root item until it is >= its children (or it’s a leaf)
  4. return the largest item

sift down the 5:
Sifting Down an Item

- To sift down item x (i.e., the item whose key is x):
  1. compare x with the larger of the item's children, y
  2. if x < y, swap x and y and repeat

- Other examples:
sift down the 10:

```
  10  
 /   \
 7   18
/     \
3  5   8 6
```

sift down the 7:

```
  7   
 /   \ 
26  23
/     \ 
15  18 10
```

siftDown() Method

```java
class SiftDown {
    private void siftDown(int i) {
        T toSift = contents[i];
        int parent = i;
        int child = 2 * parent + 1;
        while (child < numItems) {
            // If the right child is bigger, set child to be its index.
            if (child < numItems - 1 && contents[child].compareTo(contents[child + 1]) < 0) {
                child = child + 1;
            }
            if (toSift.compareTo(contents[child]) >= 0) {
                break; // we're done
            }
            contents[parent] = contents[child];
            parent = child;
            child = 2 * parent + 1;
        }
        contents[parent] = toSift;
    }
}
```

- We don't actually swap items. We put the sifted item in place at the end.
The `remove()` method is defined as follows:

```java
public T remove() {
    T toRemove = contents[0];
    contents[0] = contents[numItems - 1];
    numItems--;
    siftDown(0);
    return toRemove;
}
```

**Inserting an Item in a Heap**

- **Algorithm:**
  1. put the item in the next available slot (grow array if needed)
  2. "sift up" the new item
     until it is <= its parent (or it becomes the root item)

- **Example:** insert 35
  
  - **Put it in place:**
    - `20`
      - `16`
        - `5`
        - `8`
      - `12`
    
    - **Sift it up:**
      
      ```
      28          5
     /   \
    20     12  20
   /   |   \
  16   8   16
 |     |   |
5 8    5 8
      ```

**insert() Method**

```java
public void insert(T item) {
    if (numItems == contents.length) {
        // code to grow the array goes here...
    }
    contents[numItems] = item;
    siftUp(numItems);
    numItems++;
}
```

**Time Complexity of a Heap**

- A heap containing $n$ items has a height $\leq \log_2 n$. Why?

- Thus, removal and insertion are both $O(\log n)$.
  - remove: go down at most $\log_2 n$ levels when sifting down; do a constant number of operations per level
  - insert: go up at most $\log_2 n$ levels when sifting up; do a constant number of operations per level

- This means we can use a heap for a $O(\log n)$-time priority queue.
Using a Heap for a Priority Queue

• Recall: a priority queue (PQ) is a collection in which each item has an associated number known as a priority.
  • ("Ann Cudd", 10), ("Robert Brown", 15), ("Dave Sullivan", 5)
  • use a higher priority for items that are "more important"

• To implement a PQ using a heap:
  • order the items in the heap according to their priorities
    • every item in the heap will have a priority \( \geq \) its children
    • the highest priority item will be in the root node
  • get the highest priority item by calling `heap.remove()`!

• For this to work, we need a "wrapper" class for items that we put in the priority queue.
  • will group together an item with its priority
  • with a `compareTo()` method that compares priorities!

A Class for Items in a Priority Queue

```java
public class PQItem implements Comparable<PQItem> {
    // group an arbitrary object with a priority
    private Object data;
    private int priority;
    ...

    public int compareTo(PQItem other) {
        // error-checking goes here...
        return (priority - other.priority);
    }
}
```

• Example: `PQItem item = new PQItem("Dave Sullivan", 5);`
  • Its `compareTo()` compares PQItems based on their priorities.
  • `item1.compareTo(item2)` returns:
    • a negative integer if `item1` has a lower priority than `item2`
    • a positive integer if `item1` has a higher priority than `item2`
    • 0 if they have the same priority
Using a Heap for a Priority Queue

- Sample client code:
  ```java
  Heap<PQItem> pq = new Heap<PQItem>(50);
pq.insert(new PQItem("Dave", 5));
pq.insert(new PQItem("Ann", 10));
pq.insert(new PQItem("Bob", 15));
  
PQItem mostImportant = pq.remove(); // will get Bob!
  ```

Using a Heap to Sort an Array

- Recall selection sort: it repeatedly finds the smallest remaining element and swaps it into place:

  ```text
  0 1 2 3 4 5 6
  5 16 8 14 20 1 26
  0 1 2 3 4 5 6
  1 16 8 14 20 5 26
  0 1 2 3 4 5 6
  1 5 8 14 20 16 26
  ...
  ```

- It isn't efficient, because it performs a linear scan to find the smallest remaining element ($O(n)$ steps per scan).

- Heapsort is a sorting algorithm that repeatedly finds the largest remaining element and puts it in place.

- It is efficient, because it turns the array into a heap.
  - it can find/remove the largest remaining in $O(\log n)$ steps!
Converting an Arbitrary Array to a Heap

To convert an array (call it \textit{contents}) with \( n \) items to a heap:
1. start with the parent of the last element: \( \text{contents}[i] \), where \( i = \frac{(n-1)-1}{2} = \frac{n-2}{2} \)
2. sift down \( \text{contents}[i] \) and all elements to its left

\textbf{Example:}

\begin{center}
\begin{tabular}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
5 & 16 & 8 & 14 & 20 & 1 & 26
\end{tabular}
\end{center}

Last element's parent = \( \text{contents}[\frac{7-2}{2}] = \text{contents}[2] \).
Sift it down:

\begin{center}
\begin{tabular}{cccc}
5 \\
16 & 8 \\
14 & 20 & 1 & 26
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{ccc}
5 \\
16 & 26 \\
14 & 20 & 1 & 8
\end{tabular}
\end{center}

Converting an Array to a Heap (cont.)

Next, sift down \( \text{contents}[1] \):

\begin{center}
\begin{tabular}{cccc}
5 \\
16 & 26 \\
14 & 20 & 1 & 8
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{ccc}
5 \\
20 & 26 \\
14 & 16 & 1 & 8
\end{tabular}
\end{center}

Finally, sift down \( \text{contents}[0] \):

\begin{center}
\begin{tabular}{cccc}
5 \\
20 & 26 \\
14 & 16 & 1 & 8
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{cc}
26 \\
20 & 5
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{cc}
26 \\
8
\end{tabular}
\end{center}
Heapsort

• Pseudocode:
  
  ```java
  heapSort(arr) {
    // Turn the array into a max-at-top heap.
    heap = new Heap(arr);
    endUnsorted = arr.length - 1;
    while (endUnsorted > 0) {
      // Get the largest remaining element and put it
      // at the end of the unsorted portion of the array.
      largestRemaining = heap.remove();
      arr[endUnsorted] = largestRemaining;
      endUnsorted--;
    }
  }
  ```

Heapsort Example

• Sort the following array: 0 1 2 3 4 5 6
  13 6 45 10 3 22 5

• Here's the corresponding complete tree:

```
  13
  / \
 6   45
 / \
10  3  22  5
```

• Begin by converting it to a heap:
Heapsort Example (cont.)

• Here’s the heap in both tree and array forms:

```
45
\  /\
10 22
/   \
6   13
```  

• Remove the largest item and put it in place:

```
45
10
6
3
13
22
```

```
0 1 2 3 4 5 6
45 10 22 6 3 13 5
```

done

endUnsorted: 6

largestRemaining: 45

```
toRemove: 45
remove()
copies 45; moves 5 to root
```

```
45
10
6
3
13
22
```

```
0 1 2 3 4 5 6
45
```

endUnsorted: 6

largestRemaining: 45

```
heapSort() puts 45 in place; decrements endUnsorted
```

```
toRemove: 22
remove()
sifts down 5; returns 45
```

```
22
10
6
3
13
```

```
0 1 2 3 4 5 6
22 10 13 6 3 5 45
```

done

endUnsorted: 5

largestRemaining: 22

```
copy 22; move 5 to root
```

```
22
10
6
3
5
```

```
0 1 2 3 4 5 6
22 10 13 6 3 5 45
```

done

endUnsorted: 5

largestRemaining: 22

```
put 22 in place
```

```
10
5
6
3
```

```
0 1 2 3 4 5 6
13 10 5 6 3 22 45
```

done

endUnsorted: 4

largestRemaining: 22

```
copy 13; move 3 to root
```

```
13
10
6
3
5
```

```
0 1 2 3 4 5 6
13 10 5 6 3 22 45
```

done

endUnsorted: 4

largestRemaining: 22

```
sift down 5; return 22
```

```
13
10
6
3
5
```

```
0 1 2 3 4 5 6
13 10 5 6 3 22 45
```

done

endUnsorted: 4

largestRemaining: 22

```
put 13 in place
```

```
10
5
6
3
```

```
0 1 2 3 4 5 6
10 6 5 3 13 22 45
```

done

endUnsorted: 3

largestRemaining: 13

```
toRemove: 13
```

```
10
5
6
3
```

```
0 1 2 3 4 5 6
10 6 5 3 13 22 45
```

done

endUnsorted: 3

largestRemaining: 13
Heapsort Example (cont.)

copy 10; move 3 to root

6
   5
3

sift down 3; return 10

3
   5

put 10 in place

6

endUnsorted: 3
largestRemaining: 10

toRemove: 10

6 3 5 3 13 22 45

Heapsort Example (cont.)

copy 6; move 5 to root

3
   5

sift down 5; return 6

3

put 6 in place

5

endUnsorted: 2
largestRemaining: 6

toRemove: 6

5 3 5 10 13 22 45

Heapsort Example (cont.)

copy 5; move 3 to root

3

sift down 3; return 5

3

put 5 in place

3

endUnsorted: 1
largestRemaining: 5

toRemove: 5

3 3 6 10 13 22 45

5 3 10 13 22 45

endUnsorted: 0
Time Complexity of Heapsort

- Time complexity of creating a heap from an array?

- Time complexity of sorting the array?

How Does Heapsort Compare?

<table>
<thead>
<tr>
<th>algorithm</th>
<th>best case</th>
<th>avg case</th>
<th>worst case</th>
<th>extra memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertion sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Shell sort</td>
<td>$O(n \log n)$</td>
<td>$O(n^{1.5})$</td>
<td>$O(n^{1.5})$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>bubble sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>mergesort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><strong>heapsort</strong></td>
<td><strong>$O(n \log n)$</strong></td>
<td><strong>$O(n \log n)$</strong></td>
<td><strong>$O(n \log n)$</strong></td>
<td><strong>$O(1)$</strong></td>
</tr>
</tbody>
</table>

- Heapsort matches mergesort for the best worst-case time complexity, but it has better space complexity.
- Insertion sort is still best for arrays that are almost sorted.
  - heapsort will scramble an almost sorted array before sorting it
- Quicksort is still typically fastest in the average case.
State-Space Search Revisited

- Earlier, we considered three algorithms for state-space search:
  - breadth-first search (BFS)
  - depth-first search (DFS)
  - iterative-deepening search (IDS)
- These are all uninformed search algorithms.
  - always consider the states in a certain order
  - do not consider how close a given state is to the goal

- 8 Puzzle example:

  ![8 Puzzle Diagram]

  one step away from the goal, but the uninformed algorithms won't necessarily consider it next

Informed State-Space Search

- Informed search algorithms attempt to consider more promising states first.
- These algorithms associate a priority with each successor state that is generated.
  - base priority on an estimate of nearness to a goal state
  - when choosing the next state to consider, select the one with the highest priority
- Use a priority queue to store the yet-to-be-considered search nodes.
State-Space Search: Estimating the Remaining Cost

• As mentioned earlier, informed search algorithms associate a priority with each successor state that is generated.

• The priority is based in some way on the remaining cost – i.e., the cost of getting from the state to the closest goal state.
  • for the 8 puzzle, remaining cost = # of steps to closest goal

• For most problems, we can’t determine the exact remaining cost.
  • if we could, we wouldn’t need to search!

• Instead, we estimate the remaining cost using a heuristic function $h(x)$ that takes a state $x$ and computes a cost estimate for it.
  • heuristic = rule of thumb

• To find optimal solutions, we need an admissible heuristic – one that never overestimates the remaining cost.

Heuristic Function for the Eight Puzzle

• Manhattan distance = horizontal distance + vertical distance
  • example: For the board at right, the Manhattan distance of the 3 tile from its position in the goal state = 1 column + 1 row = 2

• Use $h(x) =$ sum of the Manhattan distances of the tiles in $x$ from their positions in the goal state
  • for our example:

  $h(x) = 1 + 1 + 2 + 2 + 1 + 0 + 1 + 1 = 9$

• This heuristic is admissible because each of the operators (move blank up, move blank down, etc.) moves a single tile a distance of 1, so it will take at least $h(x)$ steps to reach the goal.
**Greedy Search**

- Priority of state $x$, $p(x) = -1 \cdot h(x)$
  - mult. by $-1$ so states closer to the goal have higher priorities

- Greedy search would consider the highlighted successor before the other successors, because it has the highest priority.

- Greedy search is:
  - incomplete: it may not find a solution
    - it could end up going down an infinite path
  - not optimal: the solution it finds may not have the lowest cost
    - it fails to consider the cost of getting to the current state

**A* Search**

- Priority of state $x$, $p(x) = -1 \cdot (h(x) + g(x))$
  - where $g(x)$ = the cost of getting from the initial state to $x$

- Incorporating $g(x)$ allows A* to find an optimal solution – one with the minimal total cost.
Characteristics of A*

- It is complete and optimal.
  - provided that \( h(x) \) is admissable, and that \( g(x) \) increases or stays the same as the depth increases

- Time and space complexity are still typically exponential in the solution depth, \( d \) – i.e., the complexity is \( O(b^d) \) for some value \( b \).

- However, A* typically visits far fewer states than other optimal state-space search algorithms.

<table>
<thead>
<tr>
<th>solution depth</th>
<th>iterative deepening</th>
<th>A* w/ Manhattan dist. heuristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>112</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>6384</td>
<td>25</td>
</tr>
<tr>
<td>12</td>
<td>364404</td>
<td>73</td>
</tr>
<tr>
<td>16</td>
<td>did not complete</td>
<td>211</td>
</tr>
<tr>
<td>20</td>
<td>did not complete</td>
<td>676</td>
</tr>
</tbody>
</table>

- Memory usage can be a problem, but it’s possible to address it.

Implementing Informed Search

- Add new subclasses of the abstract Searcher class.

- For example:
  ```java
  public class GreedySearcher extends Searcher {
    private Heap<PQItem> nodePQueue;
    
    public void addNode(SearchNode node) {
      nodePQueue.insert(new PQItem(node, -1 * node.getCostToGoal()));
    }
  }
  ```


The numbers shown are the average number of search nodes visited in 100 randomly generated problems for each solution depth.

The searches do not appear to have excluded previously seen states.