Heaps

Not the heap we discuss in the context of C++.

A heap is a complete binary tree such that each node in the tree is greater than or equal to all its children/descendants.

Adding An Entry to A Heap

42 > 21, swap them

42 > 35, swap them

42 < 45, stop
**Pseudocode for Adding An Entry**

Step-1. Place the new entry in the heap in the first available location.

- How do we find that location?

Step-2. Swap the new entry with its parent until the new entry is smaller than or equal to its parent.

**Removing The Largest from A Heap**

1. **Promote**
   - 21 < max(children of 21)
   - swap 21 and 42

2. **Promote**
   - 21 < max(children of 21)
   - swap 21 and 35
Pseudocode for Removing The Largest

Step-1. Copy the last entry in the deepest level to the root, and delete this last node. This entry is called the “out-of-place” entry.

Step-2. Swap the out-of-place entry with its largest child until it is greater than both of its children.

How do we find the last entry in the deepest level?

Array Representation of Complete Binary Trees

![Binary Tree Diagram]

Level 1 | Level 2 | Level 3 | Level 4
---|---|---|---
45 | 42 | 23 | 27
45 | 42 | 23 | 27
45 | 42 | 23 | 27
Given a tree node stored as the $i$-th element of the array,

- its parent is stored at location $(i - 1)/2$, using integer division
- its left child is stored at location $2i + 1$, and
- its right child is stored at location $2i + 2$.

In the insert() and remove() methods, how do you find the last entry in the deepest level?

---

```cpp
template <class T>
class Heap {
public:
    Heap ();  ~Heap();
    void insert (T& entry);
    void remove_largest (T& largest);
private:
    static const int MAXIMUM = 1000;
    int data_count;
    T data[MAXIMUM];
};
```

You may want to use dynamic arrays to remove the capacity limitation of this implementation.
Implementing The insert() Method

template <class T>
void Heap::insert (T& entry)
{

}

Applications of Heaps

Application 1: Priority Queues

- A priority queue is a data structure that stores entries along with a priority for each entry.
- Entries are removed in order of priorities — the highest priority entry is removed first.
Heap Implementation of Priority Queues

- Store the entries of a priority queue in a heap.
- Compare tree nodes/entries by their priorities.
- The entry with the highest priority will always be the root of the tree.

How do we enforce the first-come-first-serve discipline among equal-priority entries?

Application 2: Heapsort

Phase-1. One by one, add all the data to a heap.

Phase-2. One by one, remove all the entries from the heap.

Who get out of the heap first?
Who is the second?
Who is the third?
A Tricky Implementation of Heapsort

Phase-1: Insertions

<table>
<thead>
<tr>
<th>Insertion</th>
<th>Array</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>add 21</td>
<td>21 35 22 27 23 45 42 19 4 5</td>
<td>5</td>
</tr>
<tr>
<td>add 35</td>
<td>21 35 22 27 23 45 42 19 4 5</td>
<td>5</td>
</tr>
<tr>
<td>swap 35 and 21</td>
<td>35 21 22 27 23 45 42 19 4 5</td>
<td>5</td>
</tr>
<tr>
<td>add 22</td>
<td>35 21 22 27 23 45 42 19 4 5</td>
<td>5</td>
</tr>
<tr>
<td>add 27</td>
<td>35 21 22 27 23 45 42 19 4 5</td>
<td>5</td>
</tr>
<tr>
<td>swap 27 and 21</td>
<td>35 27 22 21 23 45 42 19 4 5</td>
<td>5</td>
</tr>
</tbody>
</table>
add 23

add 45

swap 45 and 22

swap 45 and 35

add 42

swap 42 and 35
Phase-2: Deletions

add 19

add 4

add 5

delete 45

swap 5 and 42

swap 5 and 35
delete 42

swap 4 and 35

swap 4 and 22

remove 35

swap 19 and 27

swap 19 and 23
remove 27

\[5 \ 23 \ 22 \ 21 \ 19 \ 4 \ 27 \ 35 \ 42 \ 45\]

\[\begin{array}{c}
\text{swap 5} \\
\text{and 23}
\end{array}\]

\[23 \ 5 \ 22 \ 21 \ 19 \ 4 \ 27 \ 35 \ 42 \ 45\]

\[\begin{array}{c}
\text{swap 5} \\
\text{and 21}
\end{array}\]

\[23 \ 21 \ 22 \ 5 \ 19 \ 4 \ 27 \ 35 \ 42 \ 45\]

remove 23

\[4 \ 21 \ 22 \ 5 \ 19 \ 23 \ 27 \ 35 \ 42 \ 45\]

\[\begin{array}{c}
\text{swap 4} \\
\text{and 22}
\end{array}\]

\[22 \ 21 \ 4 \ 5 \ 19 \ 23 \ 27 \ 35 \ 42 \ 45\]
remove 22

\[
\begin{array}{c}
\text{19 21 4 5 22 23 27 35 42 45}
\end{array}
\]

swap 19 and 21

\[
\begin{array}{c}
\text{21 19 4 5 22 23 27 35 42 45}
\end{array}
\]

remove 21

\[
\begin{array}{c}
\text{5 19 4 21 22 23 27 35 42 45}
\end{array}
\]

swap 5 and 19

\[
\begin{array}{c}
\text{19 5 4 21 22 23 27 35 42 45}
\end{array}
\]

remove 19

\[
\begin{array}{c}
\text{4 5 19 21 22 23 27 35 42 45}
\end{array}
\]

swap 4 and 5

\[
\begin{array}{c}
\text{5 4 19 21 22 23 27 35 42 45}
\end{array}
\]

remove 5

\[
\begin{array}{c}
\text{4 5 19 21 22 23 27 35 42 45}
\end{array}
\]

remove 4

\[
\begin{array}{c}
\text{4 5 19 21 22 23 27 35 42 45}
\end{array}
\]

Array sorted
Implementation

void heapsort (int data[], int N) {
    // Phase-1, N heap insertions
    for (int i=0; i<N; i++) {
        // i is the index of the last entry in the heap
        int j;
        for (j=i; j>0 && data[j]>data[(j-1)/2]; j=(j-1)/2)
            swap (data[j], data[(j-1)/2]);
    }
}

// Phase-2, N heap deletions
for (int i=N; i>0; i--) {
    swap (data[0], data[i-1]);
    // Now the heap has only i-1 elements
    int j=0;
    while ((2*j+1) < i-1) {
        int max_child = 2*j+1;
        if ((2*j+2)<i-1 && data[max_child]<data[2*j+2])
            max_child = 2*j+2;
        if (data[j] < data[max_child])
            swap(data[j], data[max_child]);
        else
            break;
        j = max_child;
    }
}