Topic 5:
- Heap
- Data structure
- Heap sort
- Priority queue
Sorting Revisited

- Insertion sort
  - *quadratic*, but in-place sorting
- Merge sort
  - *nlog n*, but not in-place

Heapsort combines best of two
Heap Introduction

- An array $A$ that can be viewed as a *nearly complete* binary tree
- An array $A$ representing a heap:  
  - $\text{length}(A)$: size of the array
  - $\text{heap-size}(A) (\leq \text{length}(A))$:  
    - Only $A[1, ..., \text{heap-size}(A)]$ contain elements of the heap.
### Array? Tree?

<table>
<thead>
<tr>
<th>Index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>14</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Left(i)**
  - Return \(2i\)

- **Right(i)**
  - Return \(2i + 1\)

- **Parent(i)**
  - Return \(\left\lfloor i/2 \right\rfloor\)

Only nodes on bottom right of tree are missing.
Let $n = \text{heap-size}(A)$

Depth of $i$'th element:
- $\lceil \lg i \rceil$

Height of the tree:
- $\lceil \lg n \rceil$

Leaves:
- $\lceil n/2 \rceil + 1 \rightarrow n$
Max-heap

(Max-)heap property:

- $A[\text{Parent}(i)] \geq A[i]$
Max-heap

- (Max-)heap property:
  - \( A[\text{Parent}(i)] \geq A[i] \)

For any node \( i \), \( A[i] \) is the maximum among all the values in the subtree rooted at \( i \).
Max-heap

(Max-)heap property:
- $A[\text{Parent}(i)] \geq A[i]$
Heap Operations

- Max-Heapify
- Build-MaxHeap
- HeapSort
Max-Heapify

- **Input:**
  - For a node $i$, left and right subtrees of $i$ are max-heaps
  - $A[i]$ may be smaller than its children (violates max-heap property)

- **Output:**
  - The subtree rooted at $i$ is a max-heap.

---

Crucial in maintain heap property
Max-heapify Operation

- Max-heapify( $A$, $i$, $n$ ) :

Ex:
Max-heapify( $A$, 2 )

in-place operation
Pseudo-code

MAX-HEAPIFY(A, i, n)

\[
\begin{align*}
l &= \text{LEFT}(i) \\
r &= \text{RIGHT}(i) \\
\text{if } l &\leq n \text{ and } A[l] > A[i] \\
&\quad \text{largest} = l \\
\text{else } &\quad \text{largest} = i \\
\text{if } r &\leq n \text{ and } A[r] > A[\text{largest}] \\
&\quad \text{largest} = r \\
\text{if } \text{largest} \neq i \\
&\quad \text{exchange } A[i] \text{ with } A[\text{largest}] \\
&\quad \text{MAX-HEAPIFY}(A, \text{largest}, n)
\end{align*}
\]

Writing recurrence is not always the best approach.

Running time:
\[O(\text{height of } i) = O(\log n)\]
An Non-recursive Algorithm

```
procedure MaxHeapify(A[], size, i)
1   jnext ← i;
2   repeat
3       j ← jnext;
4       L ← Left(j);
5       R ← Right(j);
6       largest ← j;
7       if (L ≤ size) and (A[L] > A[largest]) then largest ← L;
8       if (R ≤ size) and (A[R] > A[largest]) then largest ← R;
9       if (j ≠ largest) then Swap (A[j], A[largest]);
10      jnext ← largest;
11     until (j = largest);
```
Build Max-heap Operation

- **Build-Max-Heap** \((A, n)\)
  
  \(n = \text{length}(A)\);
  
  for \(i = \lceil n/2 \rceil\) downto 1 do
  
  Max-Heapify \((A, i, n)\)

- **Time complexity:**
  - Easy bound: \(O(n \log n)\)
  - Tight bound: \(\Theta(n)\)
Heapsort

- Input array: $A[1..n]$
- Wish to be in-place sorting
- First: build heap

$A[1..n]$

$A[1..n-1]$

$A[1..n-2]$
- **Heapsort** ($A, n$)
  
  Built-Max-Heap ($A, n$)

  for $i=n$ downto 2
  

  Max-Heapify ($A, 1, i-1$)

- **Time complexity:**
  
  - $O(n \lg n)$
Priority Queue

- Operations:
  - Init( S )
  - Insert ( S, key )
  - Maximum ( S )
  - Extract-Max ( S )
  - Increase-key ( S, i, key )

A priority queue can be implemented using various data structures, heap is one of them.
Using Linked List for Priority Queue

- Operations:
  - Init( $S$ ) $O(1)$
  - Insert ( $S, key$ ) $O(1)$
  - Maximum ( $S$ ) $O( \text{sizeof}(S) )$
  - Extract-Max ( $S$ ) $O( \text{sizeof}(S) )$
  - Increase-key ( $S, i, key$ ) $O(1)$
Heap Implementations

**Heap-Extract-Max** $(A, n)$

```plaintext
if $n < 1$
    error “heap underflow”
max = $A[1]$
n = $n - 1$
Max-Heapify $(A, 1, n)$ // remakes heap
return max
```

Time complexity: $O(\lg n)$
Heap Implementation -- cont.

- Increase-key (S, i, key)
Increase-Key

\[
\text{HEAP-INCREASE-KEY}(A, i, key) \\
\text{if } key < A[i] \\
\quad \text{error "new key is smaller than current key"} \\
\quad A[i] = key \\
\text{while } i > 1 \text{ and } A[\text{PARENT}(i)] < A[i] \\
\quad \text{exchange } A[i] \text{ with } A[\text{PARENT}(i)] \\
\quad i = \text{PARENT}(i)
\]

- Time complexity: $O(\log n)$
Increase-Key

\[
\text{HEAP-INCREASE-KEY}(A, i, \text{key})
\]

\[
\text{if } \text{key} < A[i] \\
\text{error “new key is smaller than current key”}
\]

\[
A[i] = \text{key}
\]

\[
\text{while } i > 1 \text{ and } A[\text{PARENT}(i)] < A[i] \\
A[\text{PARENT}(i)] = A[i]
\]

How about

Decrease-key (S, i, key)

- Time complexity: \(O(lg \ n)\)
Heap Implementation -- cont.

Max-Heap-Insert \((A, key, n)\)

\[
\begin{align*}
n & = n + 1 \\
A[n] & = -\infty \\
\text{Heap-Increase-Key} \ (A, n, key)
\end{align*}
\]

Time complexity:
\[O (\lg n)\]
Priority Queue – Heap Implementation

- Operations:
  - Insert \((S, key)\) \(O(lg n)\)
  - Maximum \((S)\) \(O(1)\)
  - Extract-Max \((S)\) \(O(lg n)\)
  - Increase-key \((S, i, key)\) \(O(lg n)\)

A different way to build heap:
Insertion-Build-Heap
Another Heap-sort

- Time complexity: $O(n \ lg \ n)$
Example of Alg. Analysis

* Analyze the running time of the following algorithm, where the priority P is implemented by a Max-heap.

```
Program1(A, n)
  /* A is an array of n elements */
  P.Initialize();
  for i ← 1 to n do
    P.Insert(A[i]);
  end
  while P.Size() ≠ 0 do
    x ← P.ExtractMax();
    Print x;
  end
```
Another Example

- Analyze the running time of the following algorithm, where the priority $P$ is implemented by a Max-heap

```
Program2(A, n)
/* A is an array of n elements */
1 P.Initialize();
2 for i ← 1 to n do
3     j ← 1;
4     while j ≤ n do
5         P.Insert(A[i] * A[j]);
6         j ← j * 2;
7     end
8 end
```
Summary

- Heap
  - Data structure
  - Heapsort

- Priority queue
  - Implemented by heap
Lower Bound for Sorting

- **Model**
  - What types of operations are allowed
  - E.g: partial sum
    - Both addition and subtraction
    - Addition-only model

- For sorting:
  - Comparison-based model
Decision Tree

\[ a_i > a_j \]

- yes
  - \[ a_k > a_m \]
  - ...

- no
  - \[ a_s > a_t \]
  - ...

CSE 2331/5331
Example

- For insertion sort with 3 elements
Decision Tree

- Not necessary same height
- Worst case complexity:
  - Longest root-leaf path
- Each leaf:
  - A possible outcome
    - I.e., a permutation of input
- Every possible outcome should be some leaf
- \#leaves \( \geq n! \)
Lower Bound

- Any binary tree of height $h$ has at most $2^h$ leaves
  - A binary tree with $m$ leaves is of height at least $lg m$

- Worst case complexity for any algorithm sorting $n$ elements is:
  - $\Omega(lg (n!)) = \Omega(n \ lg \ n)$
  - (by Stirling approximation)
Non-comparison Based Sorting

- Assume inputs are integers $\in [1, \ldots k]$
  - $k = O(n)$

- Count-Sort ($A$, $n$) (simplified version)
  
  Initialize array $C[1,\ldots k]$ with $C[i] = 0$
  for $i = 1$ to $n$ do
    $C[A[i]]++$
  output based on $C$

- Time and space complexity
  - $O(k) = O(n)$