Topic 5:
Heap
data structure
heap sort
Priority queue
Sorting Revisited

- Insertion sort
  - \textit{quadratic}, but in-place sorting

- Merge sort
  - \textit{nlg 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 $A[1, \ldots, \text{heap-size}(A)]$ contain elements of the heap.
Array? Tree?

Index: 1 2 3 4 5 6 7 8 9 10

16 14 10 8 7 9 3 2 4 1

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

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

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

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:
  \( \lceil n/2 \rceil + 1 \to n \)
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)

l = LEFT(i)
r = RIGHT(i)

if l ≤ n and A[l] > A[i]
    largest = l
else largest = i

if r ≤ n and A[r] > A[largest]
    largest = r

if largest ≠ i
    exchange A[i] with A[largest]
MAX-HEAPIFY(A, largest, n)

Writing recurrence
is not always the best
approach.

Running time:
O(height of i) = O(lg n)
An Non-recursive Algorithm

```plaintext
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: \( O(n \lg n) \)
  - Tight bound: \( \Theta(n) \)

leaves from \( n/2 \) to \( n \)

Still in-place!
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 \(2\)
    
    Exchange \(A[1]\) with \(A[i]\)
    
    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

Algorithm: *Heap-Extract-Max* \((A, n)\)

```plaintext
if \(n < 1\)
    error “heap underflow”
\[
\begin{align*}
    max &= A[1] \\
    n &= n - 1 \\
    \text{Max-Heapify}(A, 1, n) & \quad // \text{remakes heap}
\end{align*}
\]
return \(max\)
```

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

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

**HEAP-INCREASE-KEY** \((A, i, key)\)

if \(key < A[i]\)
   error "new key is smaller than current key"

\(A[i] = key\)

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

\(i = \text{PARENT}(i)\)

- Time complexity: \(O(lg n)\)
Increase-Key

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

- if \(key < A[i]\)
  - error “new key is smaller than current key”
  - \(A[i] = key\)
- while \(i > 1\) 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

```
procedure HeapSort(B[], n)
1  size ← 0;
2  for i ← 1 to n do
3      MaxHeapInsert (A, size, B[i]);
4    end
5  for i ← n downto 1 do
6      B[i] ← HeapExtractMax(A, size);
7  end
```

- 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 */
    1 P.Initialize();
    2 for i ← 1 to n do
    3     P.Insert(A[i]);
    4 end
    5 while P.Size() ≠ 0 do
    6     x ← P.ExtractMax();
    7     Print x;
    8 end
```
Another Example

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

```plaintext
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
  - \[ a_k > a_m \]
  - ... [continued]
- no
  - \[ a_s > a_t \]
  - ... [continued]
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 ≥ \( n! \)
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
  - $O(k) = O(n)$