CS473-Algorithms I

Lecture 8

Heapsort
Introduction

- $O(n \lg n)$ worst case
- Sorts in place
- Another design paradigm
  - Use of a data structure (heap) to manage information during execution of algorithm
Heap Data Structure

- Nearly complete binary tree
  - Completely filled on all levels, except possibly the lowest level
  - Lowest level is filled from left to right
  - Each node of the tree stores an element

- **Height** of a node
  - Length of the longest simple downward path from the node to a leaf

- **Depth** of a node
  - Length of the simple downward path from the root to the node
Heap Property

- For every node \( i \) other than root
  - Max-Heap: \( A[\text{parent}(i)] \geq A[i] \)
  - Min-Heap: \( A[\text{parent}(i)] \leq A[i] \)

  Where \( A[i] \) denotes the element stored at node \( i \)

- Will discuss Max-Heap

  Fact: Largest element in a subtree of a heap is at the root of the subtree.
Example

A:

1 2 3 4 5 6 7 8 9 10

16 14 10 8 7 9 3 2 4 1
Heap Data Structure

• Store a heap in an array with implicit links
  – Left child: \( \text{left}(i) = 2i \)
  – Right child: \( \text{right}(i) = 2i + 1 \)

Computing \( 2i \) is fast: \text{left shift} \text{ in binary}

– Parent of \( i \) is: \( \text{parent}(i) = \lfloor i/2 \rfloor \)

Computing \( \lfloor i/2 \rfloor \) is fast: \text{right shift} \text{ in binary}

• \( A[1] \): element stored at the root

• Array has two attributes
  – \text{length}[A]: number of elements in A
  – \text{heap-size}[A] = \text{n}: number of elem. in heap stored in A

\[
n \leq \text{length}[A]
\]
Heap Operations

**EXTRACT-MAX(A, n)**

max ← A[1]
n ← n – 1
**HEAPIFY(A, 1, n)**
return max

O(1) + heapify time
Heap Operations

Maintaining heap property:

Subtrees rooted at left[i] and right[i] are already heaps.

But, A[i] may violate the heap property (i.e., may be smaller than its children)

Idea: Float down the value at A[i] in the heap so that subtree rooted at i becomes a heap.

HEAPIFY(A, i, n)

if 2i ≤ n and A[2i] > A[i]
then largest ← 2i
else largest ← i

if 2i +1 ≤ n and A[2i+1] > A[largest]
then largest ← 2i +1
if largest ≠ i then
HEAPIFY(A, largest, n)
else return
Maintaining Heap

HEAPIFY(A, 2, 10)

HEAPIFY(A, 4, 10)
Intuitive Analysis of HEAPIFY

• Consider HEAPIFY(A, i, n)
  – let $h(i)$ be the height of node $i$
  – at most $h(i)$ recursion levels
    • Constant work at each level: $\Theta(1)$
  – Therefore $T(i) = O(h(i))$

• Heap is almost-complete binary tree
  ▶ $h(i) = O(\lg n)$

• Thus $T(n) = O(\lg n)$
Formal Analysis of HEAPIFY

- Worst case occurs when last row of the subtree $S_i$ rooted at node $i$ is half full

- $T(n) \leq T(|S_{L(i)}|) + \Theta(1)$

- $S_{L(i)}$ and $S_{R(i)}$ are complete binary trees of heights $h(i) - 1$ and $h(i) - 2$, respectively
Formal Analysis of HEAPIFY

- Let \( m \) be the number of leaf nodes in \( S_{L(i)} \)
- \( |S_{L(i)}| = m + (m - 1) = 2m - 1 \);
  - ext \quad \text{int}
- \( |S_{R(i)}| = \frac{m}{2} + (\frac{m}{2} - 1) = m - 1 \)

- \( |S_{L(i)}| + |S_{R(i)}| + 1 = n \)
  
  \( (2m - 1) + (m - 1) + 1 = n \Rightarrow m = \frac{n+1}{3} \)
  \( |S_{L(i)}| = 2m - 1 = 2\left(\frac{n+1}{3}\right) - 1 = (\frac{2n}{3} + \frac{2}{3}) - 1 = \frac{2n}{3} - 1/3 \leq \frac{2n}{3} \)

- \( T(n) \leq T(\frac{2n}{3}) + \Theta(1) \Rightarrow T(n) = O(\log n) \)

By case 2 of Master Thm
Maintaining Heap Property: Efficiency Issues

Recursion vs iteration:

- In the absence of tail recursion iterative version is in general more efficient.

Because of the pop/push operations to/from stack at each level of recursion.

HEAPIFY(A, i, n)

\[
\begin{align*}
  j & \leftarrow i \\
  \text{while true do} & \\
  & \quad \text{if } 2j \leq n \text{ and } A[2j] > A[j] \\
  & \quad \quad \text{then largest } \leftarrow 2j \\
  & \quad \text{else largest } \leftarrow j \\
  & \quad \text{if } 2j + 1 \leq n \text{ and } A[2j+1] > A[\text{largest}] \\
  & \quad \quad \text{then largest } \leftarrow 2j + 1 \\
  & \quad \text{if largest } \neq j \text{ then} \\
  & \quad \quad \text{exchange } A[j] \leftrightarrow A[\text{largest}] \\
  & \quad \quad j \leftarrow \text{largest} \\
  & \quad \text{else return}
\end{align*}
\]
Building Heap

• Use HEAPIFY in a bottom-up manner
  – This processing order guarantees that $S_{L(i)}$ and $S_{R(i)}$ are already heaps when HEAPIFY is run on node $i$

Lemma: last $\lceil n/2 \rceil$ nodes of a heap are all leaves

Proof:

$m = 2^{d-1}$: # nodes at level $d - 1$

$f$: # nodes at level $d$ (last level)
Proof of Lemma

• # of leaves = \( f + (m - \left\lfloor \frac{f}{2} \right\rfloor) \)

\[
= m + \left\lfloor \frac{f}{2} \right\rfloor
\]

\[
m + (m - 1) + f = n
\]

\[
2m + f = n + 1
\]

\[
\left\lfloor \frac{1}{2}(2m + f) \right\rfloor = \left\lfloor \frac{1}{2}(n + 1) \right\rfloor
\]

\[
\left\lfloor m + \frac{f}{2} \right\rfloor = \left\lfloor \frac{n}{2} \right\rfloor
\]

\[
m + \left\lfloor \frac{f}{2} \right\rfloor = \left\lfloor \frac{n}{2} \right\rfloor
\]

• # of leaves = \( \left\lfloor \frac{n}{2} \right\rfloor \)

Q.E.D
Building Heap

**BUILD-HEAP**(A, n)

for \( i \leftarrow \lfloor n/2 \rfloor \) downto 1 do

**HEAPIFY**(A, i, n)

Running time analysis

- Get simple \( O(n \lg n) \) bound
  - \( n \) calls to **HEAPIFY** each of which takes \( O(\lg n) \) time
  - Loose bound
  - A good approach in general
    - Start by proving easy bound
    - Then, try to tighten it
Build-Heap: Example
Build-Heap: Example (cont')
If the heap is complete binary tree then \( h_\ell = d - \ell \)

Otherwise, nodes at a given level do not all have the same height

But we have \( d - \ell - 1 \leq h_\ell \leq d - \ell \)
Assume that all nodes at level $\ell = d - 1$ are processed

$$T(n) = \sum_{\ell=0}^{d-1} n_\ell O(h_\ell) = O(\sum_{\ell=0}^{d-1} n_\ell h_\ell)$$

$$\begin{cases} n_\ell = 2^\ell & \text{# of nodes at level } \ell \\ h_\ell = \text{height of nodes at level } \ell \end{cases}$$

$$\therefore T(n) = O\left( \sum_{\ell=0}^{d-1} 2^\ell (d - \ell) \right)$$

Let $h = d - \ell \Rightarrow \ell = d - h$ (change of variables)

$$T(n) = O\left( \sum_{h=1}^{d} h 2^{d-h} \right) = O\left( \sum_{h=1}^{d} h 2^{d/2^h} \right) = O\left( 2^d \sum_{h=1}^{d} h \left(1/2\right)^h \right)$$

but $2^d = \Theta(n) \Rightarrow T(n) = O\left( n \sum_{h=1}^{d} h \left(1/2\right)^h \right)$
Build-Heap: **tighter** running time analysis

\[
\sum_{h=1}^{d} h(1/2)^h \leq \sum_{h=0}^{d} h(1/2)^h \leq \sum_{h=0}^{\infty} h(1/2)^h
\]

recall infinite decreasing geometric series

\[
\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \quad \text{where} \quad |x| < 1
\]

differentiate both sides

\[
\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}
\]
Build-Heap: **tighter** running time analysis

\[ \sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2} \]

then, multiply both sides by \( x \)

\[ \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \]

in our case: \( x = 1/2 \) and \( k = h \)

\[ \therefore \sum_{h=0}^{\infty} h(1/2)^h = \frac{1/2}{(1-1/2)^2} = 2 = \Theta(1) \]

\[ \therefore T(n) = \Theta(n\sum_{h=1}^{d} h(1/2)^h) = \Theta(n) \]
Heapsort Algorithm

The HEAPSORT algorithm

(1) Build a heap on array $A[1\ldots n]$ by calling $\text{BUILD-HEAP}(A, n)$
(2) The largest element is stored at the root $A[1]$
(3) Discard node $n$ from the heap
(4) Subtrees ($S_2$ & $S_3$) rooted at children of root remain as heaps
    but the new root element may violate the heap property
    Make $A[1\ldots n - 1]$ a heap by calling $\text{HEAPIFY}(A, 1, n - 1)$
(5) $n \leftarrow n - 1$
(6) Repeat steps 2–4 until $n = 2$
Heapsort Algorithm

HEAPSORT(A, n)
BUILD-HEAP(A, n)
for i ← n downto 2 do
    HEAPIFY(A, 1, i −1)
Heapsort: Example
Heapsort: Example
Heapsort: Example
Heapsort Run Time Analysis

- **BUILD-HEAP** takes $O(n)$ time
- $i$-th iteration of for loop takes $O(\lg(n - i))$ time

$$T(n) = \sum_{i=1}^{n-1} O(\lg(n-i)) = \sum_{k=1}^{n-1} O(\lg k) = O\left(\sum_{k=1}^{n-1} \lg k\right) = O(n \lg n)$$

- **Heapsort** is a very good algorithm but, a good implementation of **quicksort** always beats **heapsort** in practice
- However, **heap data structure** has many popular applications, and it can be efficiently used for implementing **priority queues**
Data structures for Dynamic Sets

• Consider sets of records having key and satellite data
Operations on Dynamic Sets

• Queries: Simply return info; Modifying operations: Change the set

  - INSERT\((S, x)\): (Modifying) \(S \leftarrow S \cup \{x\}\)
  - DELETE\((S, x)\): (Modifying) \(S \leftarrow S - \{x\}\)
  - MAX\((S)\) / MIN\((S)\): (Query) return \(x \in S\) with the largest/smallest \(key\)
  - EXTRACT-MAX\((S)\) / EXTRACT-MIN\((S)\) : (Modifying) return and delete \(x \in S\) with the largest/smallest \(key\)
  - SEARCH\((S, k)\): (Query) return \(x \in S\) with \(key[x] = k\)
  - SUCCESSOR\((S, x)\) / PREDECESSOR\((S, x)\) : (Query) return \(y \in S\) which is the next larger/smaller element after \(x\)

• Different data structures support/optimize different operations
Priority Queues (PQ)

• Supports
  – INSERT
  – MAX / MIN
  – EXTRACT-MAX / EXTRACT-MIN

• One application: Schedule jobs on a shared resource
  – PQ keeps track of jobs and their relative priorities
  – When a job is finished or interrupted
    Highest priority job is selected from those pending using EXTRACT-MAX
  – A new job can be added at any time using INSERT
Priority Queues

• **Another application**: Event-driven simulation
  – Events to be simulated are the items in the **PQ**
  – Each event is associated with a time of occurrence which serves as a *key*
  – Simulation of an event can cause other events to be simulated in the future
  – Use **EXTRACT-MIN** at each step to choose the next event to simulate
  – As new events are produced insert them into the **PQ** using **INSERT**
Implementation of Priority Queue

• **Sorted linked list**: Simplest implementation
  – **INSERT**
    – O($n$) time
    – Scan the list to find place and splice in the new item
  – **EXTRACT-MAX**
    – O(1) time
    – Take the first element

> Fast extraction but slow insertion.
Implementation of Priority Queue

- **Unsorted linked list**: Simplest implementation
  - **INSERT**
    - $O(1)$ time
    - Put the new item at front
  - **EXTRACT-MAX**
    - $O(n)$ time
    - Scan the whole list
  
  ▶ Fast insertion but slow extraction

  Sorted linked list is better on the average
  - **Sorted list**: on the average, scans $n/2$ elem. per insertion
  - **Unsorted list**: always scans $n$ elem. at each extraction
Heap Implementation of PQ

- **INSERT** and **EXTRACT-MAX** are both $O(\log n)$
  - good compromise between fast insertion but slow extraction and vice versa
- **EXTRACT-MAX**: already discussed **HEAP-EXTRACT-MAX**

**INSERT**: Insertion is like that of Insertion-Sort.

- Traverses $O(\log n)$ nodes, as **HEAPIFY** does but makes fewer comparisons and assignments
  - **HEAPIFY**: compares parent with both children
  - **HEAP-INSERT**: with only one

**HEAP-INSERT**($A$, $key$, $n$)

- $n \leftarrow n + 1$
- $i \leftarrow n$
- while $i > 1$ and $A[\lfloor i/2 \rfloor] < key$ do
  - $A[i] \leftarrow A[\lfloor i/2 \rfloor]$
  - $i \leftarrow \lfloor i/2 \rfloor$
- $A[i] \leftarrow key$
HEAP-INSERT \((A, 15)\)
Heap Increase Key

- Key value of $i$-th element of heap is increased from $A[i]$ to $key$

```
HEAP-INCREASE-KEY(A, i, key)
    if key < A[i] then
        return error
    while $i > 1$ and $A\lfloor i/2 \rfloor < key$ do
        $A[i] \leftarrow A\lfloor i/2 \rfloor$
        $i \leftarrow \lfloor i/2 \rfloor$
    $A[i] \leftarrow key$
```
HEAP-INCREASE-KEY(A, 9, 15)
Heap Implementation of PQ

<table>
<thead>
<tr>
<th>key</th>
<th>data</th>
<th>H-ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>d</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>g</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>h</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>i</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>k</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>l</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>n</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

Diagram of heap structure: