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
  - **Height** of the tree: height of the root
- **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: left(i)=2i
  – Right child: right(i)= 2i+1
  Computing 2i is fast: left shift in binary
  – Parent of i is: parent(i)=\lfloor i/2 \rfloor
  Computing \lfloor i/2 \rfloor is fast: right shift in binary

• A[1]: element stored at the root
• Array has two attributes
  – length[A]: number of elements in A
  – heap-size[A]=n: number of elem. in heap stored in A

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

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

\[
\begin{align*}
\text{max} &\leftarrow A[1] \\
n &\leftarrow n - 1 \\
\text{HEAPIFY}(A, 1, n) \\
\text{return max}
\end{align*}
\]

\(O(1) + \text{heapify time}\)
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)

\[
\begin{align*}
\text{if} & \ 2i \leq n \ \text{and} \ A[2i] > A[i] \\
& \text{then} \ largest \leftarrow 2i \\
\text{else} & \ largest \leftarrow i \\
\text{if} & \ 2i + 1 \leq n \ \text{and} \ A[2i+1] > A[\text{largest}] \\
& \text{then} \ largest \leftarrow 2i + 1 \\
\text{if} & \ largest \neq i \ \text{then} \\
& \text{exchange} \ A[i] \leftrightarrow A[\text{largest}] \\
& \text{HEAPIFY}(A, \text{largest}, n) \\
\text{else} & \text{return}
\end{align*}
\]
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
  \( \geq h(i) = O(lg n) \)

• 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 $S_{L(i)}$

- $|S_{L(i)}| = m + (m - 1) = 2m - 1$;

- $|S_{R(i)}| = m/2 + (m/2 - 1) = m - 1$

- $|S_{L(i)}| + |S_{R(i)}| + 1 = n$

  $(2m - 1) + (m - 1) + 1 = n \Rightarrow m = (n+1)/3$

- $|S_{L(i)}| = 2m - 1 = 2(n+1)/3 - 1 = (2n/3 + 2/3) - 1 = 2n/3 - 1/3 \leq 2n/3$

- $T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\lg n)$

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

Recursion vs iteration:

• In the absence of tail recursion, the 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)

\[ j \leftarrow i \]

while true do

if 2\( j \leq n \) and A[2\( j \)] > A[\( j \)]

then largest \( \leftarrow 2 \( j \) \)

else largest \( \leftarrow j \)

if 2\( j + 1 \) \( \leq n \) and A[2\( j + 1 \)] > A[largest]

then largest \( \leftarrow 2 \( j + 1 \) \)

if largest \( \neq j \) then

exchange A[\( j \)]\( \leftrightarrow A[largest] \)

\[ j \leftarrow \text{largest} \]

else return
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 $\left\lceil n/2 \right\rceil$ 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 = \( f + (m - \lceil f/2 \rceil) \)

\[
= m + \lfloor f/2 \rfloor
\]

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

\( 2m + f = n + 1 \)

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

\[
\lfloor m + f/2 \rfloor = \lceil n/2 \rceil
\]

\[
m + \lfloor f/2 \rfloor = \lceil n/2 \rceil
\]

• # of leaves = \( \lceil n/2 \rceil \)

Q.E.D
Building Heap

\texttt{BUILD-HEAP}(A, n)

\begin{align*}
\text{for } i \leftarrow \left\lfloor \frac{n}{2} \right\rfloor \text{ downto } 1 \text{ do} \\
\text{HEAPIFY}(A, i, n)
\end{align*}

Running time analysis

- Get simple $O(n \lg n)$ bound
  - $n$ calls to $\text{HEAPIFY}$ 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’)

Diagram showing the process of building a heap.
Build-Heap: tighter running time analysis

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

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

$$\quad \left\{ \begin{array}{l}
 n_\ell = 2^\ell = \# \text{ of nodes at level } \ell \\
 h_\ell = \text{height of nodes at level } \ell
\end{array} \right.$$ 

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

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

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

but $2^d = \Theta(n) \Rightarrow T(n) = \mathcal{O}\left(n \sum_{h=1}^{d} h \left(1/2\right)^h\right)$
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

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

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

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

![Diagram showing a record with a key and satellite data]
Operations on Dynamic Sets

- **Queries:** Simply return info; **Modifying operations:** Change the set

  - $\text{INSERT}(S, x)$: (Modifying) $S \leftarrow S \cup \{x\}$
  - $\text{DELETE}(S, x)$: (Modifying) $S \leftarrow S \setminus \{x\}$
  - $\text{MAX}(S) / \text{MIN}(S)$: (Query) return $x \in S$ with the largest/smallest $key$
  - $\text{EXTRACT-MAX}(S) / \text{EXTRACT-MIN}(S)$: (Modifying) return and delete $x \in S$ with the largest/smallest $key$
  - $\text{SEARCH}(S, k)$: (Query) return $x \in S$ with $\text{key}[x] = k$
  - $\text{SUCCESSOR}(S, x) / \text{PREDECESSOR}(S, x)$: (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

```plaintext
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
```

Traverses $O(\log n)$ nodes, as **HEAPIFY** does but makes fewer comparisons and assignments
HEAP-INSERT(A, 15)
Heap Increase Key

• Key value of \( i \)-th element of heap is increased from \( A[i] \) to \( key \)

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

\[
\begin{align*}
\text{if } & \text{ key } < A[i] \text{ then} \\
\text{return } & \text{ error} \\
\text{while } & i > 1 \text{ and } A[\lfloor i/2 \rfloor] < \text{ key do} \\
& A[i] \leftarrow A[\lfloor i/2 \rfloor] \\
& i \leftarrow \lfloor i/2 \rfloor \\
& A[i] \leftarrow \text{key}
\end{align*}
\]
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></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
14 4
1   10
10  3
16  1
*   *
9   6
2   8
15  2
*   *
3   7
7   5
*   *
8   9
*   *
*   *
```