Today’s Lecture

- Continue with Recurrences
  - Master Theorem
  - Quicksort
- First Data-Structure:
  - Heap
  - Its Application in Sorting
Iterative recurrences

- Example:
  \[ T(n) = 4T(n/4) + n \]
  \[ = n + 4(n/4 + 4T(n/16)) \]
  \[ = n + nn + 16T(n/16) \]
  \[ = n + 2n + 16[4/4 + 4T(n/8)] \]
  \[ = n + 2n + 4n + 4T(n/8) \]
  \[ = n + 2n + 4n + ... \]
  \[ = n + \log_{4}\varepsilon-1 \]
  \[ = n \sum_{i=0}^{\log_{4}\varepsilon-1} 2^i + 4^{\log_{4}\varepsilon}T(1) \]
  \[ = \Theta(n^2) + \Theta(n^2) \]

- Disadvantage:
  - Tedious
  - Error-Prone

- Use to generate initial guess, and then prove by induction.

---

Master Theorem

- Let \( a \) and \( b \) be constants, and let \( f(n) \) be a nonnegative function defined on integral powers of \( b \). Let \( T(n) \) be defined on the integral powers of \( b \) as
  \[ T(n) = \begin{cases} 
  & \Theta(1) \quad \text{if } n = 1 \\
  & aT\left(\frac{n}{b}\right) + f(n) \quad \text{if } n = b^k
  \end{cases} \]

Then we have:

- If \( f(n) = O\left(n^{\log_b a - \varepsilon}\right) \) for some constant \( \varepsilon > 0 \), then \( T(n) = O\left(n^{\log_b a}\right) \)
- If \( f(n) = \Theta\left(n^{\log_b a}\right) \) for some constant \( \varepsilon > 0 \), then \( T(n) = O\left(n^{\log_b a} \log n\right) \)
- If \( f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right) \) for some constant \( \varepsilon > 0 \), and if \( n \geq b \Rightarrow af(n/b) \leq cf(n) \) for some positive constant \( c \geq 0 \), then \( T(n) = \Theta\left(f(n)\right) \)
Example

- Consider the recurrence \( T(n) = T(n/2) + 1 \) (binary search)
  Then \( a = 1, b = 2 \) and \( f(n) = 1 = n^{\log_b 1} \), so by case 2 of Master Theorem
  \( T(n) = \Theta(n^{\log_b 1} \log n) = \Theta(\log n) \)
- Consider the recurrence \( T(n) = 2T(n/2) + n \) (merge sort)
  Then \( a = 2, b = 2 \) and \( f(n) = n = n^{\log_b 2} \), so by case 2 of Master Theorem
  \( T(n) = \Theta(n \log n) \)
- Consider the recurrence \( T(n) = T(n/4) + n^{1/2} \)
  Then \( a = 1, b = 4 \) and \( f(n) = n^{1/2} = \Omega(n^{\log_b 4}) \),
  and \( af(n/b) = (n/4)^{1/2} = n^{1/2} / 2 = 0.5 f(n) \).
  So by case 3 of Master Theorem \( T(n) = \Theta(n^{1/2}) \)

Build recursive tree

The tree:

\[
\begin{array}{c|c|c|c|c|c}
 & f(n) & af(n/b) & a^2 f(n/b^2) & f(n/b^3) & f(n/b^4) \\
\hline
f(n) & f(n/b) & f(n/b) & f(n/b) & f(n/b) & f(n/b) \\
\end{array}
\]

Last row: \( \Theta(a^{\log_b n}) = \Theta(n^{\log_b a}) \) elements, each one \( \Theta(1) \).

Total: \( \Theta(n^{\log_b a}) + \sum_{i=1}^{\log_b n - 1} a^i f(n/b^i) \)

Which term dominates?

\[\sum_{i=1}^{\log_b n - 1} a^i f(n/b^i)\]
Back to Algorithms

- Quick Sort
  - Sort in place
  - Very practical
  - Divide-and-conquer

- Algorithm
  - Divide into two arrays around the first element
  - Recursively sort each array
  - Merge/combine-trivial

Partition Routine

Partition\( (A,p,r) \)

\[
x = A(r) \\
i = p - 1 \\
\text{for } j = p \text{ to } r - 1 \\
\quad \text{if } A(j) \leq x \text{ then} \\
\quad \quad i++ \\
\quad \quad \text{exchange}(A(i),A(j)) \\
\text{exchange}(A(i+1),A(r)) \\
\text{return}(i+1)
\]

\[ \begin{array}{ccc}
\leq x & > x & ?? \\
\hline
p & i & j & r
\end{array} \]
Quick Sort

Quicksort(A,p,r)
while (p<r)
    q=partition(A,p,r)
    Quicksort(A,p,q-1)
    Quicksort(A,q+1,r)
end

- To simplify, assume distinct elements:
  - Lucky always an even element: \( T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n) \)
  - Unlucky: \( T(n) = 2T(0) + T(n-1) + \Theta(n) = \Theta(n^2) \)

- How to avoid bad case?
  - Partition around middle element (does not work!)
  - Idea: Partition around a random element!

Quicksort (cont’d.)

- Partition around a Randomly chosen element and let \( T(n) \) be the expected time to sort.

- Consider the case where the partition is \( (k,n-k-1) \). In this case, the expected time to terminate is:

\[
T(k) + T(n-k-1) + \Theta(n)
\]

- Condition on \( k \) being a specific value, note that any value of \( k \) from 0 to \( n-1 \) is equally likely:

\[
T(n) = \sum_{k} \Pr[(k,n-k-1) \text{ split}] T(n \mid (k,n-k-1) \text{ split})
\]
\[
= \sum_{k} \left[ T(k) + T(n-k-1) + \Theta(n) \right]
\]
\[
= \frac{1}{n} \sum_{k=0}^{n-1} \left[ T(k) + T(n-k-1) + \Theta(n) \right]
\]
\[
= \frac{1}{n} \sum_{k=0}^{n-1} [T(k) + \Theta(n)]
\]
Solving the recurrence

- Next:
  We try to prove that $T(n) \leq an \log n + b$

First, Choose $b$ large enough to satisfy $T(1) \leq a \log 1 + b = b$

Inductive step:

$$T(n) = \frac{2}{n} \sum_{k=1}^{n-1} T(k) + \Theta(n) \leq \frac{2}{n} \sum_{k=1}^{n-1} ak \log k + b + \Theta(n)$$

$$= \frac{2a}{n} \sum_{k=1}^{n-1} k \log k + \frac{2}{n} b + \Theta(n)$$

Need to prove this is $\leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2$

$$= \frac{2a}{n} \left( \frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + 2b + \Theta(n)$$

$$= an \log n + b + \left( \Theta(n) + b - \frac{an}{4} \right)$$

---

Technical Lemma

- We need to show $n^2 \log n$ bound is true.

$$\sum_{k=1}^{n-1} k \log k = \sum_{k=1}^{\left\lfloor \frac{n}{2} \right\rfloor} k \log k + \sum_{k=\left\lfloor \frac{n}{2} \right\rfloor}^{n-1} k \log k$$

$$\leq \log n \left( \sum_{k=1}^{n-1} k - \sum_{k=\left\lfloor \frac{n}{2} \right\rfloor}^{n-1} k \right)$$

$$\leq \log n \left( \frac{n(n-1)}{2} - \frac{n}{2} \left( \frac{n}{2} - 1 \right) \right)$$

$$\leq \frac{1}{2} n^2 \log n - \frac{n^2}{8}$$
Heaps, Priority Queues and Heap Sort

Priority Queue

- Handles a collection of items, called keys.
- There exists a way to compare keys to each other. This is called an order relation.
- The result of these comparisons determines the priority of the keys.
- Operations supported:
  - insert a key
  - Remove the largest key
Applications

- Scheduling
- Operating systems
- Keeping track of largest $n$ elements in a sequence
- Sorting

Methods of a Priority Queue

- Initialize: initialize the structure
- Insert (key): insert a new key
- Remove Max: return and remove largest key

PQ-Sort in procedural pseudocode

- (sorting an array with using a priority queue)
  - Initialize
  - for $i = 1$ to $n$
    - Insert ($a[i]$)
  - for $i := n$ downto 1
    - $a[i] :=$ RemoveMax
How to Implement a Priority Queue

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Insert</th>
<th>Remove Max</th>
<th>Delete</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Array or Linked List</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted Array or Linked List</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Heap</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

Heap

- A heap is a binary tree storing keys, with the following properties:
  - partial order:
    - key (child) < key(parent)
  - left-filled levels:
    - the last level is left-filled
    - the other levels are full
Logarithmic Height

- A heap with \( n \) keys has height: \( H(n) = \log_2 n \)
- \textbf{Proof:}
  
  Let \( n \) be the number of keys, and \( H(n) \) be the height. We have:
  \[
  2^{H(n)-1} \leq n \leq 2^{H(n)}
  \]
  Taking logarithm of both sides; the result will follow.

---

Heap Representations

- left_child(i) = 2i
- right_child(i) = 2i + 1
- parent(j) = j \div 2

<table>
<thead>
<tr>
<th>X</th>
<th>T</th>
<th>O</th>
<th>G</th>
<th>S</th>
<th>M</th>
<th>N</th>
<th>A</th>
<th>E</th>
<th>R</th>
<th>B</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
Heap Insertion

- Add the key in the next available spot in the heap.
- Upheap checks if the new node is greater than its parent. If so, it switches the two.
- Upheap continues up the tree.
Heap Insertion

- **Upheap** terminates when new key is less than the key of its **parent** or the **top of the heap** is reached.
- \((\text{total } \#\text{switches}) \leq (\text{height of tree} - 1) = \log n\)

Heapify Algorithm

- Assumes L and R sub-trees of \(i\) are already Heaps and makes It's sub-tree a Heap:

  \[
  \text{Heapify}(A, i, n) \\
  \text{If } (2i \leq n) \text{ & } (A[2i] > A[i]) \text{ Then} \\
  \text{largest} = 2i \\
  \text{Else} \text{largest} = i \\
  \text{If } (2i+1 \leq n) \text{ & } (A[2i+1] > A[\text{largest}]) \text{ Then} \\
  \text{largest} = 2i+1 \\
  \text{If } (\text{largest } \neq i) \text{ Then} \\
  \text{Exchange } (A[i], A[\text{largest}]) \\
  \text{Heapify}(A, \text{largest}, n) \\
  \text{Endif} \\
  \text{End Heapify}
  \]
Extracting the Maximum from a Heap:

- Here is the algorithm:
  
  \[
  \text{Heap-Extract-Max}(A) \\
  \text{Remove } A[1] \\
  n=n-1 \\
  \text{Heapify}(a, 1, n) \\
  \text{End } \text{Heap-Extract-Max}
  \]

Building a Heap

- Builds a heap from an unsorted array:
  
  \[
  \text{Build_Heap}(A, n) \\
  \text{For } i=\text{floor}(n/2) \text{ down to } 1 \text{ do} \\
  \text{Heapify}(A, i, n) \\
  \text{End } \text{Build_Heap}
  \]

- Example:

  \[
  A = \begin{bmatrix} 4 & 1 & 3 & 2 & 16 & 9 & 10 & 14 & 8 & 7 \end{bmatrix}
  \]
Building a Heap (cont’d.)
Running time of Building a Heap

- $O(n \log n)$ is trivial: $n$ calls of Heapify, each of cost $O(\log n)$
- Tighter Bound: $O(n)$
  - The cost of “Heapify” is proportional to the number of levels visited (height of node $i$)
  - Assume $n=2^k-1$ (complete binary tree):
    - For each leaf node, the number of levels visited is 1,
    - For each node at next level is 2,
    - 3 for next level, etc.

Total # of levels visited = \[ \sum_{i=0}^{\log(n+1)} \frac{i}{2^i} \]

Using Induction, it is easy to see that

\[ \sum_{i=0}^{\log(n+1)} \frac{i}{2^i} = O(1) \]

Implying:

$T(n) = \text{Total # of levels visited} = O(n)$
Heapsort

Heapsort\((A, n)\)

**Build-Heap\((A, n)\)**

For \(i = n\) downto 2 do


Heapify\((A, 1, i)\)

End For

End Heapsort

---

Heapsort (cont’d.)

- First build the corresponding Heap:

\[ A \]

\[
\begin{array}{cccccccc}
16 & 14 & 8 & 7 & 9 & 3 & 2 & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
16 & 14 & 8 & 7 \\
10 & 9 & 3 \\
2 & 4 & 1 \\
\end{array}
\]
Heapsort (cont’d.)

Exchange Heapify\((A, 1, n-1)\)

Heapsort (cont’d.)

Exchange Heapify\((A, 1, n-2)\)
Heapsort (cont’d.)

\[ \text{Exchange Heapify} \left( A, 1, n-3 \right) \]

Heapsort (cont’d.)

\[ \text{Exchange Heapify} \left( A, 1, n-4 \right) \]
Heapsort (cont’d.)

Finally:

Heapsort (cont’d.)
### Running Time

**Heapsort**\((A,n)\)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Build-Heap</strong>((A,n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>For (i=n) downto 2 do</td>
<td>(n−1)Times</td>
</tr>
<tr>
<td>Exchange (A[1] &amp; A[i])</td>
<td>(O(1))</td>
</tr>
<tr>
<td><strong>Heapify</strong>((A,i))</td>
<td>(O(\log n))</td>
</tr>
</tbody>
</table>

End Heapsort

- Total Running time: \(O(n \log n)\)