CSE5311 – Module 2

• This Class
  – Heaps and Heapsort?
  – QuickSort
  – Mergesort
  – Other Sorting Algorithms

• At the end of the class
  ▪ Binary trees
  ▪ Priority queues and heaps
  ▪ Quicksort
    ▪ Worstcase
    ▪ Bestcase
  ▪ Mergesort
  ▪ Recurrences for Quicksort and Mergesort

Further Reading
Reference books on Algorithms

Course Syllabus

• Review of Asymptotic Analysis and Growth of Functions, Recurrences

• Sorting Algorithms
  • Graphs and Graph Algorithms.
  • Greedy Algorithms:
    ▪ Minimum spanning tree, Union-Find algorithms, Kruskal's Algorithm,
    ▪ Clustering,
    ▪ Huffman Codes, and
    ▪ Multiphase greedy algorithms.
  • Dynamic Programming:
    ▪ Shortest paths, negative cycles, matrix chain multiplications, sequence alignment, RNA secondary structure, application examples.

• Network Flow:
  ▪ Maximum flow problem, Ford-Fulkerson algorithm, augmenting paths, Bipartite matching problem, disjoint paths and application problems.

• NP and Computational tractability:
  ▪ Polynomial time reductions; The Satisfiability problem; NP-Complete problems; and Extending limits of tractability.

• Approximation Algorithms, Local Search and Randomized Algorithms

8/30/2009 M KUMAR CSE5311 1

8/30/2009 M KUMAR CSE5311 2
SORTING ALGORITHMS

Heaps and Heapsort

Priority Trees
Building Heaps
Maintaining Heaps
Heapsort Algorithm
Analysis of Heapsort Algorithm

Priority Queues

What is a priority queue?
A priority queue is an abstract data type which consists of a set of elements. Each element of the set has an associated priority or key. Priority is the value of the element or value of some component of an element.

Example:
S : {(Brown, 20), (Gray, 22), (Green, 21)} priority based on name
{(Brown, 20), (Green, 21), (Gray, 22)} priority based on age

Each element could be a record and the priority could be based on one of the fields of the record.
Example

A Student's record:

Attributes: Name Age Sex Student No. Marks
Values: John Brown 21 M 94XYZ23 75

Priority can be based on name, age, student number, or marks

Operations performed on priority queues,
- inserting an element into the set
- finding and deleting from the set an element of highest priority

Priority Queues

Priority queues are implemented on partially ordered trees (POTs)
- POTs are labeled binary trees
- the labels of the nodes are elements with a priority
- the element stored at a node has at least as large a priority as the elements stored at the children of that node
- the element with the highest priority is at the root of the tree
Example

HEAPS

The heap is a data structure for implementing POT's
Each node of the heap tree corresponds to an element of the array that stores the value in the node
The tree is filled on all levels except possibly the lowest, which are filled from left to right up to a point.

An array $A$ that represents a heap is an object with two attributes
- $\text{length}[A]$, the number of elements in the array and
- $\text{heap-size}[A]$, the number of elements in the heap stored within the array $A$

$\text{heap-size}[A] \leq \text{length}[A]$
HEAPS (Contd)


<table>
<thead>
<tr>
<th>Position</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>Value</td>
<td>24</td>
<td>21</td>
<td>19</td>
<td>13</td>
<td>14</td>
<td>3</td>
<td>10</td>
<td>2</td>
<td>7</td>
<td>11</td>
</tr>
</tbody>
</table>

Given node with index \( i \),

\[
\text{PARENT}(i) \text{ is the index of parent of } i; \text{PARENT}(i) = \left\lfloor \frac{i}{2} \right\rfloor
\]

\[
\text{LEFT CHILD}(i) \text{ is the index of left child of } i; \text{LEFT CHILD}(i) = 2i;
\]

\[
\text{RIGHT CHILD}(i) \text{ is the index of right child of } i; \text{ and RIGHT CHILD}(i) = 2i + 1
\]

Heap Property

THE HEAP PROPERTY
\[ A[\text{PARENT}(i)] \geq A[i] \]

The heap is based on a binary tree
The height of the heap (as a binary tree) is the number of edges on the longest simple downward path from the root to a leaf.

The height of a heap with \( n \) nodes is \( O(\log n) \).

All basic operations on heaps run in \( O(\log n) \) time.
n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1

Number of nodes at different levels in a Binary Tree

Heap Algorithms

- HEAPIFY
- BUILD_HEAP
- HEAPSORT
- HEAP_EXTRACT_MAX
- HEAP_INSERT
HEAPIFY

The HEAPIFY algorithm checks the heap elements for violation of the heap property and restores heap property.

Procedure HEAPIFY (A, i)
Input: An array A and index i to the array. i = 1 if we want to heapify the whole tree. Subtrees rooted at LEFT_CHILD(i) and RIGHT_CHILD(i) are heaps
Output: The elements of array A forming subtree rooted at i satisfy the heap property.

1. \( l \leftarrow \) LEFT_CHILD \((i)\);
2. \( r \leftarrow \) RIGHT_CHILD \((i)\);
3. if \( l \leq \text{heap\_size}[A] \) and \( A[l] > A[i] \)
   then largest \( \leftarrow l; \)
   else largest \( \leftarrow i; \)
4. if \( r \leq \text{heap\_size}[A] \) and \( A[r] > A[\text{largest}] \)
   then largest \( \leftarrow r; \)
5. if largest \( \neq i \)
6. then exchange \( A[i] \leftrightarrow A[\text{largest}] \)
7. HEAPIFY (A, largest)
Procedure HEAPSORT(A)
Input: Array A[1...n], n = length[A]
Output: Sorted array A[1...n]
1. BUILD_HEAP[A]
2. for i ← length[A] down to 2
4. heap_size[A] ← heap_size[A]+1;
5. HEAPIFY(A,1)

Procedure BUILD_HEAP (A)
Input: An array A of size n = length [A], heap_size[A]
Output: A heap of size n
1. heap_size[A] ← length[A]
2. for i ← [length[A]/2] downto 1
3. HEAPIFY(A,i)
Running time of Build_heap

1. Each call to Heapify takes $O(\log n)$ time
2. There are $O(n)$ such calls
3. Therefore the running time is at most $O(n \log n)$

However the complexity of Build_heap is $O(n)$

Proof:
In an $n$ element heap there are at most $\lceil n/2^{h+1} \rceil$ nodes of height $h$

The time required to heapify a subtree whose root is at a height $h$ is $O(h)$

(this was proved in the analysis for Heapify)

So the total time taken for Build_heap is given by,

$$
\sum_{h=0}^{\left\lfloor \log_2 n \right\rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot h
\leq \frac{n}{2} \cdot \sum_{h=0}^{\left\lfloor \log_2 n \right\rfloor} \frac{h}{2^h}
$$

We know that

$$
\sum_{h=0}^{\infty} \frac{h}{2^h} = 2 = O(n)
$$

Thus the running time of Build_heap is given by, $O(n)$
The HEAPSORT Algorithm

Procedure HEAPSORT(A)
Input : Array A[1…n], n = length[A]
Output : Sorted array A[1…n]
1. BUILD_HEAP[A]
2. for i ← length[A] down to 2
4. heap_size[A] ← heap_size[A]-1;
5. HEAPIFY(A,1)

Example : To be given in the lecture

Running Time:
Step 1 BUILD_HEAP takes O(n) time,
Steps 2 to 5 : there are (n-1) calls to HEAPIFY
which takes O(log n) time
Therefore running time takes  O (n log n)
HEAP_EXTRACT_MAX

Procedure HEAP_EXTRACT_MAX(A[1…n])

Input : heap(A)

Output : The maximum element or root, heap (A[1…n-1])

1. if heap_size[A] ≥ 1
2. max ← A[1];
4. heap_size[A] ← heap_size[A]-1;
5. HEAPIFY(A,1)
6. return max

Running Time : O (log n) time

8/30/2009 M KUMAR CSE5311

HEAP_INSERT

Procedure HEAP_INSERT(A, key)

Input : heap(A[1…n]), key - the new element

Output : heap(A[1…n+1]) with k in the heap

1. heap_size[A] ← heap_size[A]+1;
2. i ← heap_size[A];
3. while i > 1 and A[PARENT(i)] < key
5. i ← PARENT(i);
6. A[i] ← key

Running Time : O (log n) time

8/30/2009 M KUMAR CSE5311
Questions:

- What is a heap?
- What are the running times for heap insertion and deletion operations?
- Did you understand HEAPIFY AND and HEAPSORT algorithms?
- Can you write a heapsort algorithm for arranging an array of numbers in descending order?

Quicksort

- Quicksort algorithm
- Quicksort performance
- Quicksort analysis
Quicksort

- The worst case running time of Quicksort algorithm is $O(n^2)$
- However, its expected running time is $O(n \log n)$
- Three-step divide-and-conquer process for sorting a subarray $A[l..r]$
  
  **Divide**: partition the array $A[l..r]$ into two nonempty subarrays $A[l..q]$ and $A[q+1,r]$ such that each element of $A[l..q]$ is less than or equal to each element of $A[q+1,..,r]$
  
  **Conquer**: sort the two subarrays $A[l..q]$ and $A[q+1..r]$ by recursive calls to Quicksort
  
  **Combine**: the subarrays are already sorted in place. No work is needed to combine them

---

Example

| 13 | 02 | 18 | 26 | 76 | 87 | 98 | 11 | 93 | 77 | 65 | 43 | 38 | 09 | 65 | 06 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 13 | 02 | 06 | 26 | 76 | 87 | 98 | 11 | 93 | 77 | 65 | 43 | 38 | 09 | 65 | 18 |
| 13 | 02 | 06 | 09 | 76 | 87 | 98 | 11 | 93 | 77 | 65 | 43 | 38 | 26 | 65 | 18 |
| 13 | 02 | 06 | 09 | 11 | 87 | 98 | 76 | 93 | 77 | 65 | 43 | 38 | 26 | 65 | 18 |
| 11 | 02 | 06 | 09 | 13 | 87 | 98 | 76 | 93 | 77 | 65 | 43 | 38 | 26 | 65 | 18 |
| 11 | 02 | 06 | 09 | 13 | 87 | 98 | 76 | 93 | 77 | 65 | 43 | 38 | 26 | 65 | 18 |
| 09 | 02 | 06 | 11 | 13 | 87 | 98 | 76 | 93 | 77 | 65 | 43 | 38 | 26 | 65 | 18 |
| 09 | 02 | 06 | 11 | 13 | 87 | 98 | 76 | 93 | 77 | 65 | 43 | 38 | 26 | 65 | 18 |
| 06 | 02 | 09 | 11 | 13 | 87 | 98 | 76 | 93 | 77 | 65 | 43 | 38 | 26 | 65 | 18 |
| 06 | 02 | 09 | 11 | 13 | 87 | 98 | 76 | 93 | 77 | 65 | 43 | 38 | 26 | 65 | 18 |
| 02 | 06 | 09 | 11 | 13 | 87 | 98 | 76 | 93 | 77 | 65 | 43 | 38 | 26 | 65 | 18 |
| 02 | 06 | 09 | 11 | 13 | 87 | 98 | 76 | 93 | 77 | 65 | 43 | 38 | 26 | 65 | 18 |
| 02 | 06 | 09 | 11 | 13 | 87 | 98 | 76 | 93 | 77 | 65 | 43 | 38 | 26 | 65 | 98 |
| 02 | 06 | 09 | 11 | 13 | 87 | 98 | 76 | 65 | 77 | 65 | 43 | 38 | 26 | 93 | 98 |
| 02 | 06 | 09 | 11 | 13 | 26 | 18 | 76 | 65 | 77 | 65 | 43 | 38 | 87 | 93 | 98 |
| 02 | 06 | 09 | 11 | 13 | 18 | 26 | 76 | 65 | 77 | 65 | 43 | 38 | 87 | 93 | 98 |
| 02 | 06 | 09 | 11 | 13 | 18 | 26 | 76 | 65 | 38 | 65 | 43 | 77 | 87 | 93 | 98 |
| 02 | 06 | 09 | 11 | 13 | 18 | 26 | 43 | 65 | 38 | 65 | 76 | 77 | 87 | 93 | 98 |
| 02 | 06 | 09 | 11 | 13 | 18 | 26 | 43 | 38 | 65 | 65 | 76 | 77 | 87 | 93 | 98 |
| 02 | 06 | 09 | 11 | 13 | 18 | 26 | 38 | 43 | 65 | 65 | 76 | 77 | 87 | 93 | 98 |
| 02 | 06 | 09 | 11 | 13 | 18 | 26 | 38 | 43 | 65 | 65 | 76 | 77 | 87 | 93 | 98 |
Procedure `Quicksort(A,l,r)`

**Input:** Unsorted Array `(A,l,r)`

**Output:** Sorted subarray `A(0..r)`

To sort the entire array `A`, `l = 1` and `r = length[A]`

- if `l < r` then `q ← PARTITION(A,l,r)`
  - QUICKSORT(A,l,q-1)
  - QUICKSORT(A,q+1,r)

---

Procedure `PARTITION(A,l,r)`

**Input:** Array `A(l .. r)`


- `x ← A[l]; i ← l; j ← r;`
  - while `i < j` do
    - while `A[i] ≤ x` and `i ≤ r` do `i ← i + 1;`
    - while `A[j] > x` and `j ≥ l` do `j ← j - 1;`
    - if `i < j` then
      - exchange `A[i] ↔ A[j];`
      - `q ← j;`
Running Time of Quicksort

\[ T(n) = n-1 + T(i-1) + T(n-i) \]

It takes \( n-1 \) comparisons for the partition
Then we sort smaller sequences of size \( i-1 \) and \( n-i \)
Each element has the same probability of being selected as the pivot,
The average running time is given by,

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

Detailed Analysis – ? Discussion

Mergesort

Like Quicksort, Mergesort algorithm also is based on the divide-and-conquer principle.

Divide: This step computes the middle of the array, takes constant time, \( \Theta(1) \)

Conquer: Two subproblems, each of size \( n/2 \) are recursively solved.
Each subproblem contributes \( 2T(n/2) \) to the running time.

Combine: Two sorted sequences are merged, this takes \( \Theta(n) \) time
Procedure MERGESORT(A, l, r)
Input: A an array in the range 1 to n.
Output: Sorted array A.
    if l < r
        then q ← \lfloor (l+r)/2 \rfloor;
        MERGESORT(A, l, q)
        MERGESORT(A, q+1, r)
        MERGE (A, l, q, r)
Procedure \textsc{Merge}(A, l, q, r)

\textbf{Inputs:} two sorted subarrays \(A(l, q)\) and \(A(q+1, r)\)

\textbf{Output:} Merged and sorted array \(A(l, r)\)

\begin{itemize}
  \item \(i \leftarrow l;\)
  \item \(j \leftarrow q+1;\)
  \item \(k \leftarrow 0;\)
  \item while \((i \leq q)\) and \((j \leq r)\) do
    \begin{itemize}
      \item \(k \leftarrow k+1;\)
      \item if \(A[i] \leq A[j]\) then
        \begin{itemize}
          \item \(\text{TEMP}[k] \leftarrow A[i];\)
          \item \(i \leftarrow i +1;\)
        \end{itemize}
      \item else
        \begin{itemize}
          \item \(\text{TEMP}[k] \leftarrow A[j];\)
          \item \(j \leftarrow j +1;\)
        \end{itemize}
    \end{itemize}
  \item if \(j > r\) then
    \begin{itemize}
      \item for \(t \leftarrow 0\) to \(q - i\) do
        \begin{itemize}
          \item \(A[r-t] \leftarrow A[q-t];\)
        \end{itemize}
      \item for \(t \leftarrow 0\) to \(k-1\) do
        \begin{itemize}
          \item \(A[l+t] \leftarrow \text{TEMP}[t];\)
        \end{itemize}
    \end{itemize}
\end{itemize}

Runtime Complexity of Mergesort

\begin{itemize}
  \item \(n\) \(\rightarrow\) \(n/2\)
  \item \(n/2\) \(\rightarrow\) \(n/4\)
  \item \(n/4\) \(\rightarrow\) \(n/8\)
  \item ... \(\rightarrow\) \(n/2^n\)
  \item \(n/2^n\) \(\rightarrow\) \(n/2^{n-1}\)
  \item ... \(\rightarrow\) \(n/2^k\)
  \item \(n/2^k\) \(\rightarrow\) \(n/2^{k-1}\)
  \item ... \(\rightarrow\) \(n/2^1\)
  \item \(n/2^1\) \(\rightarrow\) \(n/2^0\)
  \item \(n/2^0\) \(\rightarrow\) \(n\)
\end{itemize}
Runtime Complexity of Mergesort

\[ T(n) = 2 T(n/2) + \Theta(n) \]
\[ T(n/2) = 2 T(n/4) + n/2 \]
\[ T(n/4) = 2 T(n/8) + n/4 \]
\[ T(n) = 2^3 T(n/4) + 3n \]
\[ \ldots \]
\[ T(n) = 2^k T(n/2^k) + k n \text{ if } n = 2^k \text{ then } , k = \log n \]

Therefore, \( T(n) = 2^k T(1) + n \log n = n \Theta(1) + n \log n \)
\[ T(n) = O(n \log n) \]

Questions

- What is a pivot element?
- Do you understand divide-and-conquer?
- What is running time if pivot element is at the center of the array?
- How is Mergesort different from Quicksort?
- Trace the algorithm with the help of an example?
- What is the use of TEMP array?
- Do you know why we execute steps 13 and 14 in the MERGE algorithm?
- What happens if \( i > q \) after the while loop (4-11)?
Home Work

- Find applications for selection sort, heapsort, quicksort, and mergesort sorting algorithms?
- Which of these problems are suitable for different types of sorting operations?
- Identify applications for which each sorting algorithm works best

Homework - Reference Books

- Insertion Sort
- Counting Sort
- Find the Maximum
- Find the Minimum
The input is a set $S$ containing $n$ real numbers, and a real number $x$. Design an algorithm to determine whether there are two elements of $S$ whose sum is exactly $x$. The algorithm should run in $O(n \log n)$ time. Suppose now that the set $S$ is given in a sorted order. Design an algorithm to solve the above problem in time $O(n)$.

1. Sort the numbers in $O(n \log n)$ time
2. for each number ($x_1$) search the BST to check if you can find another ($x_2$) so that $x_1 + x_2 = x$ - takes $O(n \log n)$ time

```
While i < j
    IF $S[i] + S[j] < x$ THEN j = j - 1
    else if $S[i] + S[j] > x$ THEN i = i + 1
DONE
```

$T(n) = T(n-1) + O(1)$

The input is a sequence of $n$ integers with many duplications such that the number of distinct integers in the sequence is $O(\log n)$. Design a sorting algorithm to sort such sequences using at most $O(\log \log n)$ comparisons in the worst case.

*Insert each element into a balanced binary search tree*

*Each node of the tree has a key and a pointer to a linked list of all the elements with the same key.*

*There are $O(\log n)$ distinct integers in the input sequence, therefore there are $O(\log n)$ nodes in the tree.*

*A height of the tree = $\log$ (number of nodes in the tree) *

*For our tree, height = $O(\log \log n)$*

*We can append the different linked lists (nodes of the tree) by performing an inorder traversal of the tree.*
The input is a sequence of elements \( x_1, x_2, \ldots, x_n \), given one at a time. Design an \( O(n) \) algorithm to compute the \( k \)th smallest element using only \( O(k) \) memory spaces. The value of \( k \) is known ahead of time but the value of \( n \) is not known until the last element is seen.

The input is a sequence of elements, given one at a time, inserted into a fixed memory space \( C \). In the \( i \)th input step \( x_i \) is put into \( C \) (and \( C \)'s previous content is erased). You can perform any computation between two input steps (including, moving the content of \( C \) to another memory location).

Create a heap, \( H \) of size \( k \) after \( k \) items have arrived.
[you have to write the procedure for constructing this heap]

The largest of the \( k \) items is at root of the heap, that is \( H[1] \).

When a new element arrives, compare it with

- If \( C[\text{element}] \geq H[1] \) then discard it
- Else remove Max element from \( H \) and insert \( C[\text{element}] \) into \( H \)

The above step is repeated for every new arrival and stops when the last item has arrived.

The root of the heap contains the \( k \)th smallest element.

The input is \( d \) sequences of elements such that each sequence is already sorted, and there is a total of \( n \) elements. Design an \( O(n \log d) \) algorithm to merge all the sequences into one sorted sequence.

There are \( d \) sorted sequences

Place all \( d \) minimal elements in heap, that is elements \( A_1[1], A_2[1], \ldots, A_d[1] \) are placed in the heap.
[you have to write the procedure for constructing this heap]

In each step remove the minimal element in the heap and insert the next element from corresponding sequence into the heap. For example \( A_3[1] \) is the minimal element in the heap, insert \( A_3[2] \) into the heap after the extract-min.
[you have to write procedures for extracting the minimum item from the heap and inserting a new element in the heap]
You are given a collection of \( n \) bolts of different widths and \( n \) corresponding nuts. You are allowed to try a nut and bolt together, from which you can determine whether the nut is larger than the bolt, smaller than the bolt, or matches the bolt exactly. However there is no way to compare two nuts together or two bolts together. The problem is to match each bolt to its nut. Design an algorithm for this problem with average case efficiency of \( \Theta(n \log n) \).

Given an array of integers \( A[1..n] \), such that, for all \( i, 1 \leq i < n \), we have \( |A[i] - A[i+1]| \leq 1 \). Let \( A[1] = x \) and \( A[n] = y \), such that \( x < y \). Design an efficient search algorithm to find \( j \) such that \( A[j] = z \) for a given value \( z, x \leq z \leq y \). What is the maximal number of comparisons to \( z \) that your algorithm makes?