COMP 3761: Algorithm Analysis and Design

Shidong Shan

BCIT

Overview: **Transform-and-conquer**

- Methods of transformation:
  1. Instance simplification
  2. Representation change
  3. Problem reduction
- Presorting
- Heap and heapsort
- Horner's rule
- Problem reduction

Transform and Conquer

This technique solves a problem's instance by a **transformation**:

- Instance simplification: transform one instance to a simpler and more convenient instance of the same problem
- Representation change: transform one instance to a different representation of the same instance
- Problem reduction: one problem is reduced to another
  i.e., transform one problem into an entirely different problem for which an algorithm is already available.

Presorting

Main idea: Presort the list to simplify the problem instance.

- Many problems involving lists are easier to solve when the list is sorted
- Searching
- Selection problem
- Element uniqueness problem: checking if all elements are distinct

- Presorting is used in many geometric algorithms.

How fast can we sort ?

- Efficiency of algorithms involving sorting depends on the efficiency of sorting
- **Theorem** (more in Section 11.2):
  To sort a list of size n by any comparison-based algorithm, $\lceil log_2 n! \rceil \approx n \log_2 n$ comparisons are necessary in the worst case.
- Note: About $n \log_2 n$ comparisons are also sufficient to sort an array of size n (by mergesort).

Searching with presorting

Problem: Search for a given K in $A[0..n-1]$

- Presorting-based algorithm:
  1. Sort the array by an efficient sorting algorithm (e.g., mergesort)
  2. Apply binary search.
- Efficiency:

$$\Theta(n \log n) + O(\log n) = \Theta(n \log n).$$

- Good or bad?

  Why do we have our dictionaries, telephone directories, etc. sorted?

Element uniqueness with presorting

Problem: Determine if all the elements in a given array are distinct.

- Presorting-based algorithm:
  1. Sort the array by an efficient sorting algorithm (eg., Mergesort)
  2. Scan array to check pairs of **adjacent** elements
- Efficiency:

$$\Theta(n \log n) + O(n) = \Theta(n \log n).$$

- Brute-force algorithm:
  - See Section 2.3 Example 2
  - Compare all pairs of elements
  - Efficiency: $O(n^2)$

Binary trees

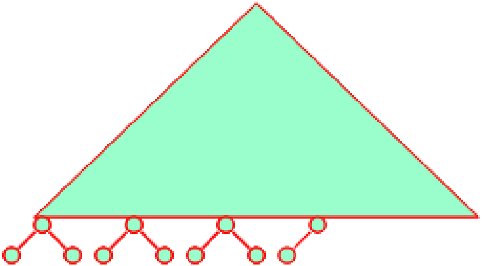
- **Ordered tree**: a rooted tree in which all the children of each vertex are ordered.
- **Binary tree**: an ordered tree in which every vertex has no more than two children: a **left child** and/or **right child**
- Important inequality for the height (h) of a binary tree with n nodes:

$$\lfloor \log_2 n \rfloor \leq h \leq n - 1.$$

Definition

A **heap** is a binary tree with keys at its nodes (one key per node) such that:

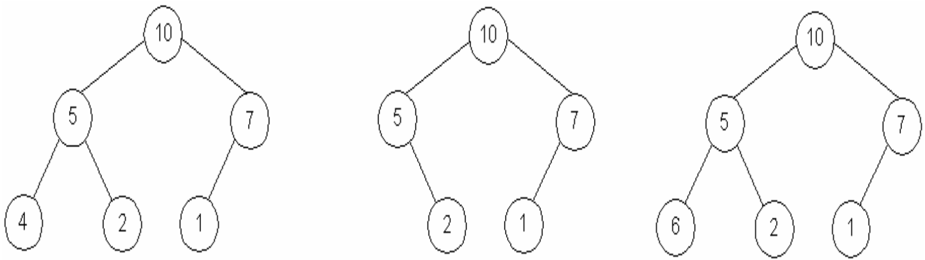
- It is essentially **complete**, i.e., all its levels are full except possibly the last level, where only some rightmost keys may be missing



- The key at each node is $\geq$ keys at its children

Illustration of the heap's definition

Which of the following trees is a heap?



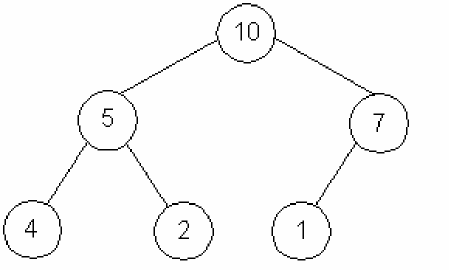
- Heaps elements are ordered top down (along any path down from its root)
- They are not ordered left to right.

Some important properties of a heap

- Given n, there exists a unique binary tree with n nodes that is essentially complete, with $h = \lfloor \log_2 n \rfloor$
- The root contains the largest key (or smallest key for a "Min-heap"
- The subtree rooted at any node of a heap is also a heap
  A heap is also a divide-and-conquer ready structure
- A heap can be represented as an array

Heap's array representation

- Store heaps elements in an array (whose elements indexed, for convenience, 1 to n) in top-down left-to-right order
- Example:



- Left child of node j is at $2j$
- Right child of node j is at $2j + 1$
- Parent of node j is at $\lfloor j/2 \rfloor$
- Parental nodes are represented in the first $\lfloor n/2 \rfloor$ locations

Heap Construction (Bottom-up)

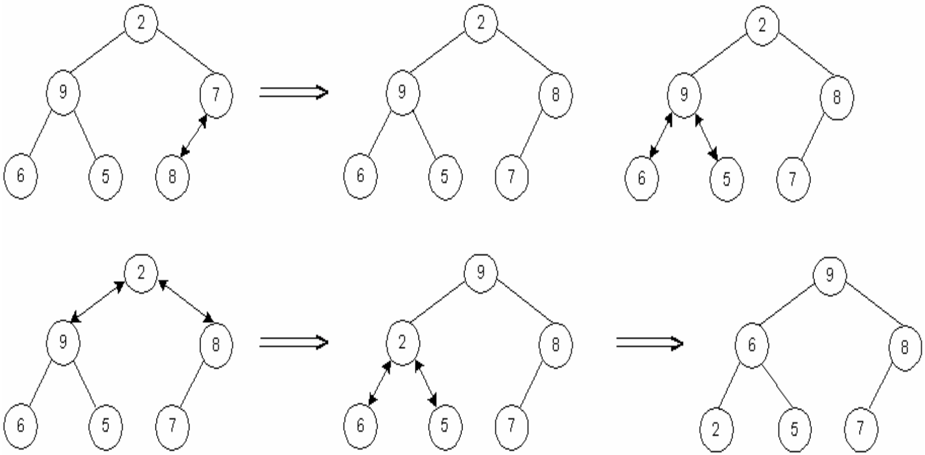
Step 0 Initialize the structure with keys in the order given

Step 1 Starting with the last (rightmost) parental node, fix the heap rooted at it, if it doesnt satisfy the heap condition:
keep exchanging it with its largest child until the heap condition holds

Step 2 Repeat Step 1 for the preceding parental node

Example of Heap Construction

Construct a heap for the list 2, 9, 7, 6, 5, 8



Algorithm of Bottom-up Heap Construction

```
Algorithm HeapBottomUp(H[1..n])
//Constructs a heap from the elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]
    heap ← false
    while not heap and 2 * k ≤ n do
          j ← 2 * k
          if j < n   //there are two children
             if H[j] < H[j + 1]   j ← j + 1
          if v ≥ H[j]
                heap ← true
          else H[k] ← H[j];   k ← j
    H[k] ← v
```

Heapsort

tage 1 heap construction: Construct a heap for a given list of n keys

tage 2 maximum deletion: Repeat operation of root removal $n - 1$ times:

- Exchange keys in the root and in the last (rightmost) leaf
- Decrease heap size by 1
- If necessary, swap new root with larger child until the heap condition holds (heapification)

Example of sorting by Heapsort

Example: Sort the list 2, 9, 7, 6, 5, 8 by heapsort

Analysis of Heapsort

- Stage 1: Build heap for a given list of n keys in the worst case:

$$C_{worst}(n) = \Sigma_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)).$$

- Note: If the heap's tree is full, the number of nodes at level i is 2^i; the number of key comparisons involving a key on level i is $2(h-i)$.
- Stage 2: Repeat operation of root removal $n-1$ times (fix heap)

$$C_{worst}(n) = \Sigma_{i=1}^{n-1} 2\log_2 i = \Theta(n \log n).$$

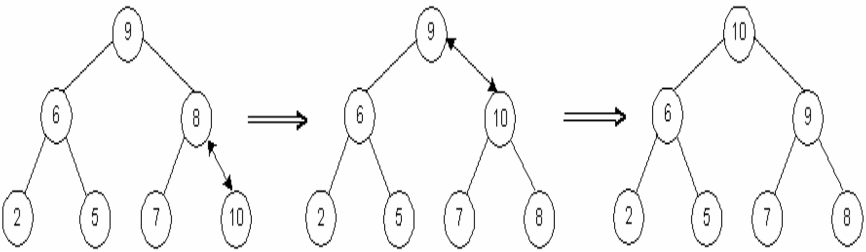
- Both worst-case and average-case efficiency: $\Theta(nlogn)$
- In-place sorting: yes
- Stable sorting: no (e.g., 1, 1)

Priority Queue

- A priority queue is the ADT of a set of elements with numerical priorities
- A priority queue has the following operations:
    1. find element with highest priority
    2. delete element with highest priority
    3. insert element with assigned priority
- Heap is a very efficient way for implementing priority queues
- Two ways to handle priority queue:
  highest priority = smallest number

Inserting a new element into a heap

- Insert the new element at last position in heap
- Compare it with its parent and, if it violates heap condition, exchange
- Continue comparing the new element with nodes up the tree until the heap condition is satisfied
- Example: Insert key 10 into the heap



- Efficiency: $O(\log n)$

Polynomial evaluation (revisited)

Problem: compute the value of the polynomial at a given point x

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0.$$

- Recall: two brute-force algorithms with different efficiency classes
- evaluate from the highest to lowest term: $O(n^2)$
- evaluate from the lowest to highest term: $O(n)$

Horner's rule

Evaluate a polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

- represent $p(x)$ by a different formula

$$p(x) = (\ldots(a_n x + a_{n-1})x + \ldots)x + a_0$$

- For example:

$$p(x) = 2x^4 - x^3 + 3x^2 + x - 5.$$

- Horner's rule uses the formula:

$$p(x) = x(x(x(2x - 1) + 3) + 1) - 5.$$

- The change of formula leads to a faster algorithm

Evaluation by Horner's rule

- Evaluate $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$ at $x = 3$
- Same sequence of computations are obtained by simply arranging the coefficient in a table and proceeding as follows:

coefficients	2	-1	3	1	-5
x = 3	2	5	18	55	160

Horner's rule Pseudocode

ALGORITHM *Horner*($P[0..n]$, x)

//Evaluates a polynomial at a given point by Horner's rule
//Input: An array $P[0..n]$ of coefficients of a polynomial of degree n
// (stored from the lowest to the highest) and a number x
//Output: The value of the polynomial at x
$p \leftarrow P[n]$
for $i \leftarrow n - 1$ **downto** 0 **do**
    $p \leftarrow x * p + P[i]$
return p

Efficiency of Horner's Rule:

- number of multiplications $M(n) = n$
- number of additions $A(n) = n$.

Problem Reduction

- Solve a problem by a transforming it into a different problem for which an algorithm is already available
- To be of practical value, the combined time of the transformation and solving the other problem should be smaller than solving the problem as given by another method

Examples of solving problems by reduction

- transforming a maximization problem to a minimization problem and vice versa
  e.g. $\max f(x) = \min -f(x)$
- min-heap construction vs. max-heap construction
- computing the Least Common Multiple $\mathrm{lcm}(m, n)$ via computing $\gcd(m, n)$
  $$lcm(m, n) = \frac{m * n}{gcd(m, n)}$$
- reduction to graph problems
  e.g., solving puzzles via state-space graphs

Lab Exercises

Section 6.1: 1, 2, 4

Section 6.4: 1, 8

Section 6.5: 1, 2, 3, 4