Data Structures

A Supplement
(Based on [Manber 1989])

Yih-Kuen Tsay

Department of Information Management
National Taiwan University

Heads

🌀 A (max) heap is a binary tree whose keys satisfy the heap property:
the key of every node is greater than or equal to the key of any of its children.

🌀 It supports the two basic operations of a priority queue:

Heaps

🌐 A (max) heap is a binary tree whose keys satisfy the heap property:
the key of every node is greater than or equal to the key of any of its children.

🌐 It supports the two basic operations of a priority queue:

☀ *Insert(x)*: insert the key x into the heap.

☀ *Remove()*: remove and return the largest key from the heap.

🌑 A binary tree can be represented implicitly by an array A as follows:

1. The root is stored in $A[1]$.
2. The left child of $A[i]$ is stored in $A[2i]$ and the right child is stored in $A[2i + 1]$.

Heaps (cont.)

Algorithm Remove_Max_from_Heap (A, n);
begin
    if $n = 0$ **then** print "the heap is empty"
    else *Top_of_the_Heap* := $A[1]$;
        $A[1] := A[n]$; $n := n - 1$;
        parent := 1; *child* := 2;
        while *child* $\leq n - 1$ **do**
            if $A[child] < A[child + 1]$ **then**
            child := *child* + 1;
            if $A[child] > A[parent]$ **then**
            swap($A[parent], A[child]$);
            parent := *child*;
            child := 2 * *child*
            else *child* := *n*
end

Heaps (cont.)

Algorithm Insert_to_Heap (A, n, x);
begin

         $n := n + 1$;

         $A[n] := x$;

         child $:= n$;

         parent $:= n$ *div* 2;

         while *parent* ≥ 1 **do**

                 if $A[parent] < A[child]$ **then**

                    swap$(A[parent], A[child])$;

                    child $:=$ *parent*;

                    parent $:=$ *parent* *div* 2

                 else *parent* $:= 0$

end

AVL Trees

Definition

An AVL tree is a binary search tree such that, for every node, the difference between the heights of its left and right subtrees is at most 1 (the height of an empty tree is defined as 0).

This definition guarantees a maximal height of $O(\log n)$ for any AVL tree of n nodes.
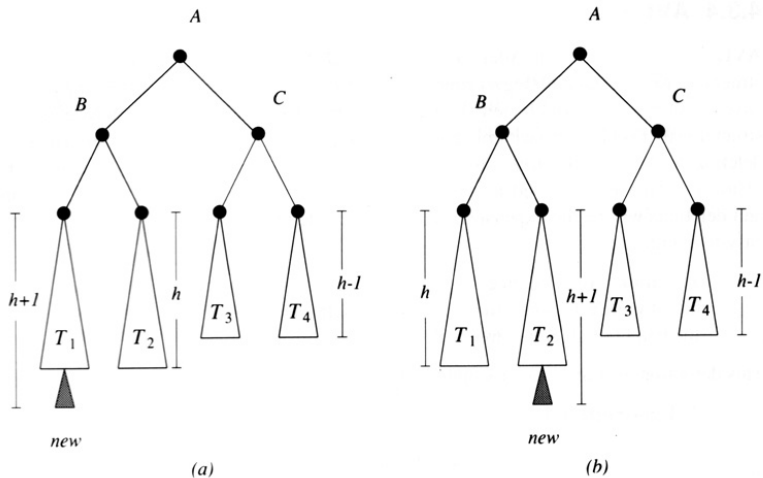
AVL Trees (cont.)

Figure 4.13 Insertions that invalidate the AVL property.
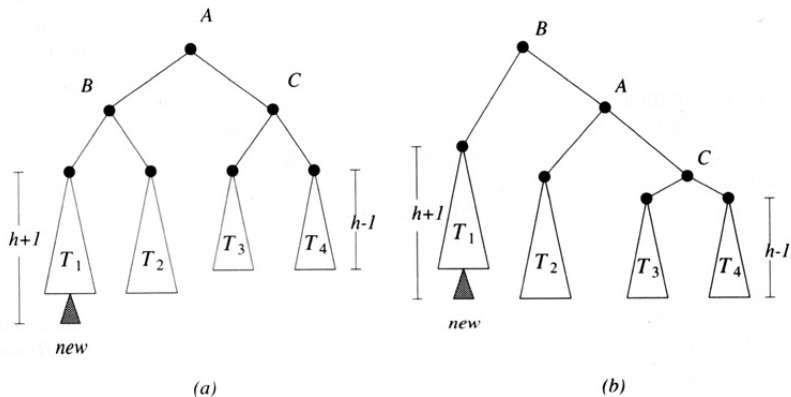
AVL Trees (cont.)

Figure 4.14 A single rotation: (a) Before. (b) After.

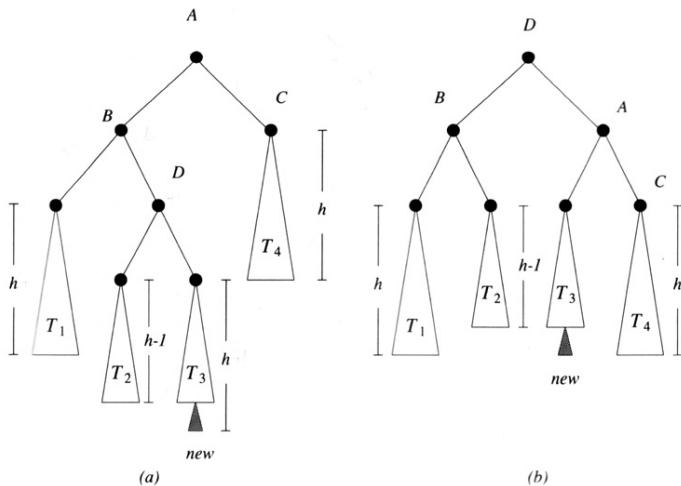Source: [Manber 1989].

AVL Trees (cont.)

Figure 4.15 A double rotation: (a) Before. (b) After.

Source: [Manber 1989].

Union-Find

- There are *n* elements $x_1, x_2, \cdots, x_n$ divided into groups. Initially, each element is in a group by itself.

- Two operations on the elements and groups:
  - *find(A)*: returns the name of A's group.
  - *union(A, B)*: combines A's and B's groups to form a new group with a unique name.

- To tell if two elements are in the same group, one may issue a find operation for each element and see if the returned names are the same.
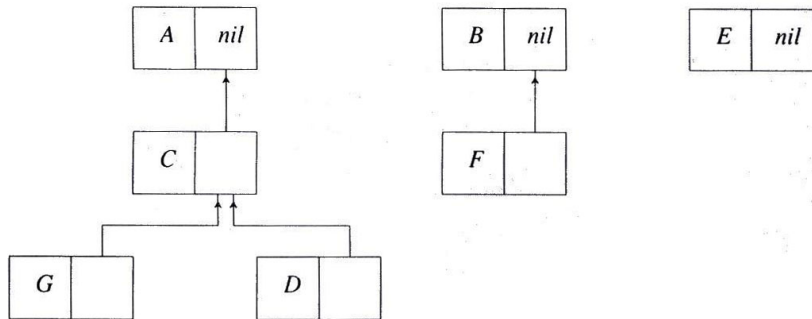
Union-Find (cont.)

Figure 4.16 The representation for the union-find problem.

Source: [Manber 1989].

🔵 The root also stores the number of elements in (i.e., the size of) its group.

🔵 To *balance* the tree resulted from a union operation, *let the smaller group join the larger group* and update the size of the larger group accordingly.

Theorem (Theorem 4.2)

If balancing is used, then any tree of height h must contain at least 2^h elements.

🔵 Any sequence of m find or union operations (where $m \geq n$) takes $O(m \log n)$ steps.
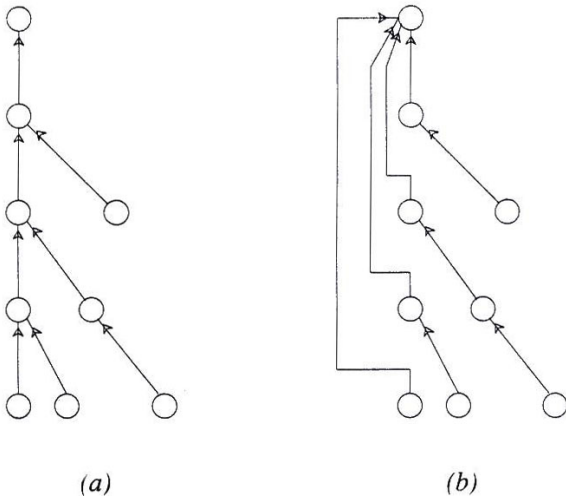
Union-Find (cont.)

(a) (b)

Figure 4.17 Path compression: (a) Before. (b) After.

Theorem (Theorem 4.3)

If both balancing and path compression are used, any sequence of m find or union operations (where $m \geq n$) takes $O(m \log^* n)$ steps.

The value of $\log^* n$ intuitively equals the number of times that one has to apply log to n to bring its value down to 1.

Code for Union-Find

```
Algorithm Union_Find_Init(A,n);
begin
  for i := 1 to n do
      A[i].parent := nil;
      A[i].size := 1
end

Algorithm Find(a);
begin
  if A[a].parent <> nil then
      A[a].parent := Find(A[a].parent);
      Find := A[a].parent;
  else
      Find := a
end
```

Code for Union-Find (cont.)

```
Algorithm Union(a,b);
begin
  x := Find(a);
  y := Find(b);
  if x <> y then
     if A[x].size > A[y].size then
        A[y].parent := x;
        A[x].size := A[x].size + A[y].size;
     else
        A[x].parent := y;
        A[y].size := A[y].size + A[x].size
end
```