Advanced Sets (II)
Balanced Trees

- Time taken by operations on binary search trees
  - Proportional to the average depth of that tree
  - Rearrange the tree after each insertion and deletion
    - To avoid the worst case, make the tree complete

![Complete Binary Search Trees, before and after INSERT(T, 1)]
Balanced Trees – 2-3 tree

- 2-3 tree
  - Each interior node has two or three children
  - Each path from the root to a leaf has the same length
  - Special cases: tree with zero node or one node
  - Each element in a set is placed at a leaf

- Each interior node keeps one or two keys
  - The smallest element that is a descendant of the second child
  - The smallest element that is a descendant of the third child, if the third child exists
Balanced Trees – 2-3 trees

- Divide descendants into two or three groups
  - \( x < \text{key}_{\text{first}} \): subtree descending from the 1\(^{\text{st}}\) child
  - \( \text{key}_{\text{first}} \leq x < \text{key}_{\text{second}} \): subtree descending from the 2\(^{\text{nd}}\) child
  - \( \text{key}_{\text{second}} \leq x \): subtree descending from the 3\(^{\text{rd}}\) child

```
  7   16
 /   /
5   8 12
 |
2  7 8 12
 |
 2 5 7 8 12 16 19
```

```
2-3 Trees – operations

- **INSERT** \((x, T)\)
  - Elements are all in the leaf nodes
  - Descending the tree \(T\), reach at a node which has leaves as children and \(x\) should be inserted as a child
    - If that node has only two children, add \(x\) as a child
      - Place \(x\) in appropriate order
      - Then, adjust the keys in the interior node
    - If three children, split that interior node.
      - Two smaller elements stay with the original node and two larger elements move to a new node.
      - Then the new node is inserted to the parent of that interior node
      - **INSERT** is performed recursively up to the root, if necessary
2-3 Trees - operations

- INSERT(18, T)
2-3 Trees - operations

- INSERT(10, T)
2-3 Trees - operations

- DELETE \((x, T)\)
  - After deletion of a leaf containing \(x\), its parent node \(node\) may have only one child
    - If \(node\) is root, delete \(node\) and make its lone child be the new root
    - Otherwise, look for \(p\), parent of \(node\). If any children adjacent to \(node\) to the right or left has three children, transfer proper one to \(node\)
    - If all adjacent nodes have two children, transfer the lone child of \(node\) to an adjacent sibling of \(node\), and delete \(node\)
2-3 Trees - operations

- DELETE (10, T)
2-3 Trees - operations

- **DELETE (7, T)**
Sets with MERGE and FIND

- In certain problem, we start with sets
  - Combine sets in some order, and
  - Ask which set a particular object contains

- MERGE and FIND
  - MERGE (A, B, C)
    - $C = A \cup B$, if A and B are disjoint (each is called component)
    - Undefined, otherwise
  - FIND (x)
    - Returns the set of which $x$ is a member
    - Undefined, if $x$ is in two or more sets, or in no set
Simple Implementation of MFSET

- **MERGE-FIND ADT called MFSET**
  - **MERGE (A, B)** – union of A and B, result in either A and B
  - **FIND (x)** – name of component of which x is a member
  - **INITIAL (A, x)** – create component A that contains only x

```plaintext
const
    n = { number of elements };
type
    MFSET = array[1..n] of integer;

array[subrange of members] of (type of set names);
```
Simple Implementation of MFSET

- **MERGE (A, B)**
  - The result of merge is called as A
  - *components* of type MFSET
    - *components*[x] holds the name of the set currently containing x

```pascal
procedure MERGE ( A, B: integer; var C: MFSET )
var
  x: 1..n;
begin
  for x:= 1 to n do
    if C[x] = B then  { C[ ] is components[ ] }
      C[x] := A
  end;  { MERGE }
```
Tree Implementation of MFSET

- Use trees with pointers to parents
  - Nodes of trees correspond to set members
  - Each node, except the root of each tree, has a pointer to its parent
  - The root hold the name of the set as well as an element
Tree Implementation of MFSET
Tree Implementation of MFSET - operations

- **MERGE**
  - Make the root of one tree be a child of the root of the other
  - Indiscriminate merging could result in a single chain of members
    - FIND on each element takes $O(n^2)$
  - Keep count at each root and merge the smaller tree to be the child of the larger tree
    - FIND on each element takes $O(n \log n)$
Tree Implementation of MFSET - operations

- Path compression
  - During FIND, when following a path from some node to the root, make each node encountered along the path be a child of the root

FIND(7) with Path Compression