Parma University, Math. Deptartment

Algorithms and Data Structures
Dynamic Balanced Trees

Marco Pellegrini

Summary of lecture 13

- This lecture:
  Balanced trees: 2-3-Trees
  Balanced trees: B-Trees (overview)
- Handout from Aho, Hopcroft and Ullman pages 169-180. for 2-3 trees
- Chapter 19 of CLR for B-trees.

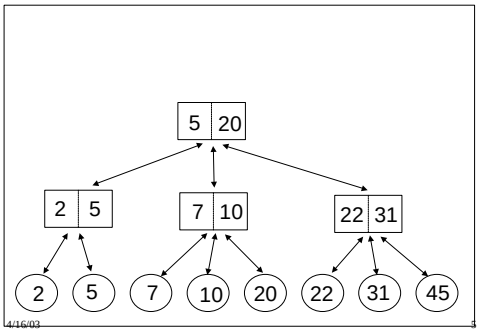
Balanced Dynamic Trees.

- We use the pointer-linked structure to build Balanced Dynamic Trees, which allow the following basic operations:
  INSERT(x,T)
  DELETE(x,T)
  MEMBER(x,T)
- We will see a type of Tree called 2-3-TREE.
- There are many other types of trees with similar properties (AVL-Trees, Red-Black-Trees, etc..).

2-3-Trees

- Each node that is not a leaf has either 2 or 3 children.
- All the leaves are at the same level.
- All the data is stored at the leaves in sorted order from left to right.
- At the root and at any intermediate node we store two numbers:
  1) The largest value in the left sub-tree
  2) The largest value in the middle sub-tree
- The right subtree might not exist.

Examples.



Exercise

- Create 3 different 2-3-trees on the following data: 1,5,6,7,9,10,13,14,17,20,30,40,55.
- Create a 2-3-tree on the above data with the smallest height.

Observations

- If a 2-3-tree has height h, it can have at least 2^h leaves, and at most 3^h leaves.
- Conversely if a 2-3-tree has N leaves then the height is between $\log_2(N)$ and $\log_3(N)$
- Thus operations that take constant time at each level, take overall O(log n) time.
- We will see next how to perform MEMBER, INSERTION and DELETION, each operation will take time O(log n).

Searching in a (2-3)-Tree.

It is similar to Binary search in an array, except that we follow pointers instead of computing indices.

```
SEARCH(x: key, v: node)
    IF Leaf(v) AND key(v)=x THEN RETURN True
    IF Leaf(v) AND key(v)≠x THEN RETURN False
    L = left-child(v); M = middle-child(v); R = right-child(v)
    IF x ≤ Left-key(v) THEN
       RETURN SEARCH(x,L)
    IF x ≤ middle-key(v) THEN
       RETURN SEARCH(x,M)
    IF R ≠ NIL THEN RETURN SEARCH(x,R)
          ELSE RETURN false
```

first call: MEMBER(x,T) = SEARCH(x, root(T))

Exercise

- Mark the nodes of the tree T visited during the call of MEMBER(5,T), where T is the tree of smallest height built in the previous exercise.

Insertion in a 2-3-tree T

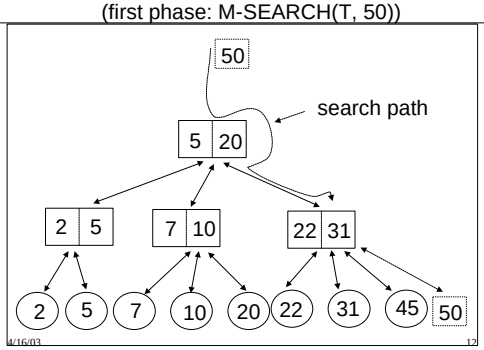
- Easy case: T has only 1 node (a leaf).
  Let a be the new value to be inserted and b the value in T. Generate root-node r.
  Make a the left-child of r and b the right-child if a < b.
  Make a the right-child and b the left-child if a >b.
  Update the values stored in r.
- If T has more than 1 node.
  We modify SEARCH to return the node of the last level before the leaves where the value a should be (ideally) placed (allowing for the time being to violate the 2-3-rule).
  Let us call such procedure M-SEARCH.

Insertion in a 2-3-tree, 2

- Now we call a procedure whose task is to restore the 2-3-property (RESTORE) if the property was violated.
- RESTORE takes as input a node A with 4 children C1,C2,C3 and C4 in left to right order. It generates a node B. Assigns C1, C2 as children of A, assigns C3, C4 as children of B.
  Makes B as the right sibling of A at the parent of A.
- If the parent of A has 3 children stop.
  Otherwise invoke RESTORE on the father of A
- If A is the root. Generate a new root as parent of A

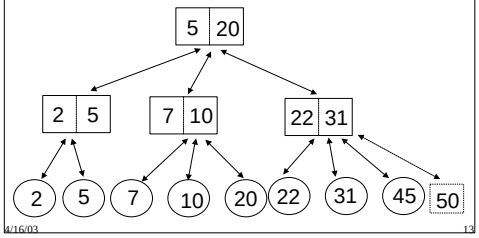
Example of Insertion.

(first phase: M-SEARCH(T, 50))



Example of Insertion.

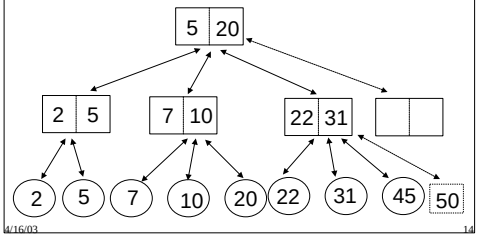
(second phase: RESTORE)



4/16/03 13

Example of Insertion.

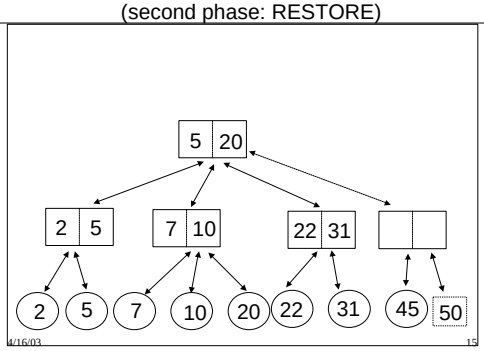
(second phase: RESTORE)



4/16/03 14

Example of Insertion.

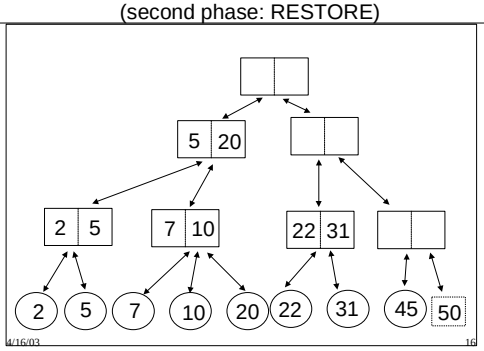
(second phase: RESTORE)



4/16/03 15

Example of Insertion.

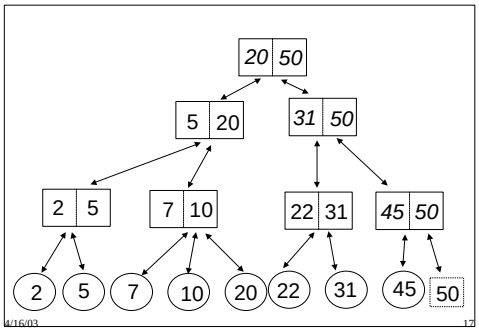
(second phase: RESTORE)



4/16/03 16

Example of Insertion.

(third phase: UPDATE DATA IN INTERMEDIATE NODES)



4/16/03 17

Exercise

- Show how the tree with minimum height of 2 exercise before is modified when the value 50 is inserted. Show the intermediate states.

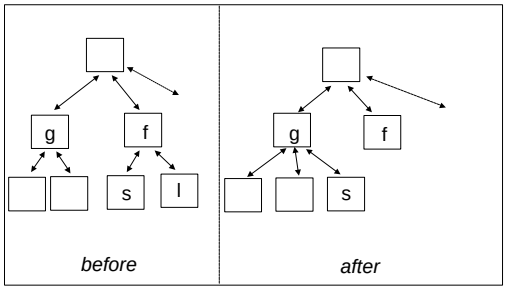
4/16/03 18

DELETION from a 2-3-tree.

- To Delete an element A from a 2-3-tree we first find A using SEARCH. Let l be the leaf holding the value A.
- If l is the root, remove l.
- If l is the child of a node with 3 children, remove l.
- If l is the child of a node f with 2 children (call them l and s).
  If f is the root: delete l and f, make s the root.
  If f is not the root: Let g be a brother of f, to its left, with 2 sons: make s the rightmost child of g, remove l, call DELETION on f.
  IF g has 3 children, make the rightmost child of g a child of f, delete l.

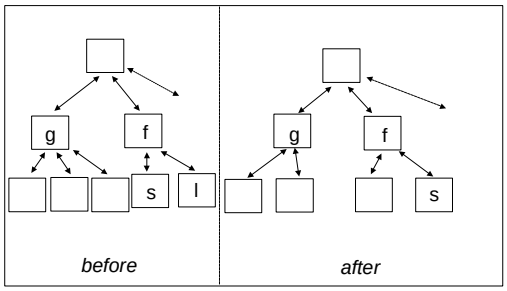
DELETION, example of subrule 1

(g, brother of f has 2 children)



DELETION, example of subrule 2

(g, brother of f has 3 children)

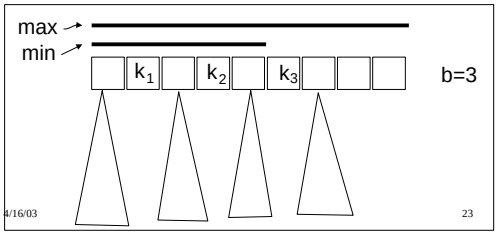


Conclusions on 2-3-trees

- For irregular trees we use linked structures.
- 2-3-tress allow to perform INSERTION, DELETION, MEMBER in time O(log n), where n is the number of current leaves.
- 2-3-trees are more flexible, but somewhat more complicated than HEAPS, etc.

B-Trees

- Every node in a B-tree has at least B children and at most 2B-1 children. (Note for B=2 these numbers are 2 and 3).
- Every node has between B-1 and 2B-2 keys.



B-Trees, cntd.

- We can relax the rule for the root that can have fewer than b-1 children.
- We want to tell a leaf from an internal node, to do this we can either keep a flag-bit on the node or check that all pointers to children are NIL.
- All the leaves are at the same level.
- To simplify algorithms we may want to store in the node also the current number of children.
- If the value of B becomes very large one can start thinking of using binary search on the node to find out the child holding a particular key, instead of a sequential search we used (implicitly) in the B=2 case.

Conclusions

- We have seen a scheme of balanced search tree called 2-3-trees.
- The basic operations of searching, inserting and deleting elements take time O(log n) in the worst case.
- B-trees are generalization of 2-3-trees which are suitable especially for data stored in external memory.