Parma University, Math. Deptartment

Algorithms and Data Structures
Binary search trees

Marco Pellegrini

Summary of lecture 12

This Lecture (CLR 13.1-3).

Binary search trees

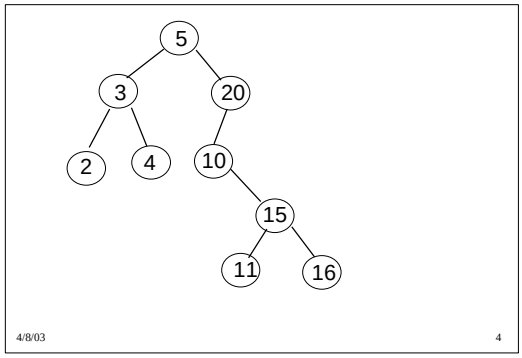
inorder visit.

Search, successor-search, insert, delete

Binary search trees.

- Hash tables can support operations: insert, delete, search.
- But operations like: find-minimum, find-maximum, list-in-sorted-order, predecessor, successor are difficult to implement.
- All of the above operations are easy on binary search trees.
- The Binary-search-tree property is:
  If y is a node in the left-subtree of x, key[y] ≤ key[x]
  If y is a node in the right-subtree of x, key[y] ≥ key[x]

Example



Listing elements in sorted order.

- We can extract all the elements in a search-tree in sorted order by a method called in-order walk.
- The idea is to visit recursively the whole tree printing the left-subtree, the node, the right subtree.
- ```
  Inorder-walk(x)
    IF x ≠ NIL
      THEN Inorder-walk(left[x])
           PRINT key[x]
           Inorder-walk(right[x])
   END IF
  ```
- First call is Inorder-walk(root(T)).
- Exercise: simulate inorder walk on the tree of the previous slide, showing the value of x at each call and the order of the calls.

Searching on a binary tree.

- The searching strategy on a binary-tree is somewhat similar to binary-search in a sorted array. By comparing k with the key at a node we know where we have to keep searching.
- ```
  Tree-search(x,k)
    IF x = NIL or k = key[x]
      THEN RETURN x END IF
    IF k < key[x]
      THEN RETURN Tree-search(left[x],k)
      ELSE RETURN Tree-search(right[x],k)
   END IF
  ```
- The time of Tree-search is O(h) where h is the height of the tree= the length of the longest path from a leaf to the root.

Exercise

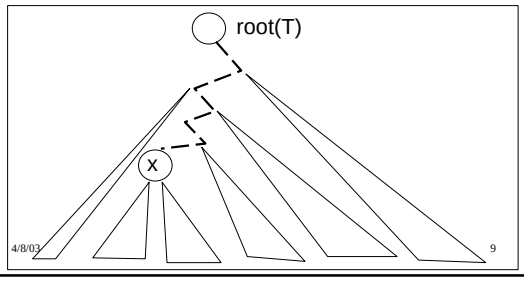
- Write a procedure Iterative-tree-search(x,k) that searches for k in the tree rooted at x without recursive calls.

Minimum and maximum

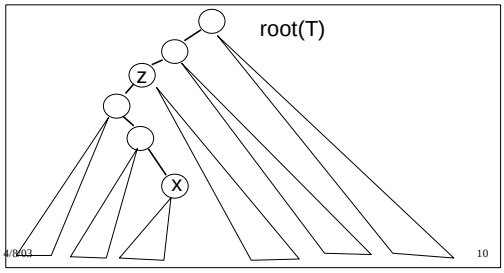
- The smallest element is found following the left-pointers.
- ```
  Tree-min(x)
    WHILE left[x] ≠ NIL DO
     x := left[x]
    END WHILE
    RETURN x
  ```
- The largest element is found following the right pointers
- ```
  Tree-max(x)
    WHILE right[x] ≠ NIL DO
      x := right[x]
    END WHILE
    RETURN x
  ```
- What is the time of Tree-max and Tree-min?

Successor-search

- Suppose x is a node in T, we look for the node that holds the key after key[x] in the sorted order.
- Case 1: x has a right-subtree.



- Case 2: x has no right subtree.
  In this case our answer is the lowest ancestor of x on which we turn left going down the tree.



Successor code

- ```
  Tree-successor(x)
   IF right[x] ≠ NIL            (the code)
     THEN RETURN Tree-minimum(right[x])
   END IF
   y := parent[x]
   WHILE y ≠ NIL AND x = right[y]
     DO  x := y
         y := parent[x]
   END WHILE
   RETURN y
  ```
- HW3.2 Write the code for Predecessor(x) where x is a node in T.
- HW3.3 Write code for Key-Successor(k) and Key-Predecessor(k). k is a key not necessarily in T. Use previous subroutines.

Insertion

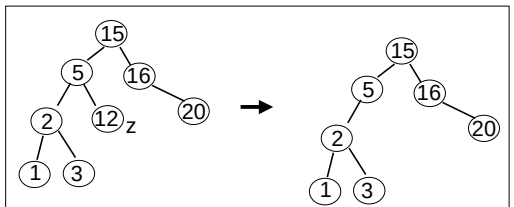
- We want to insert a new key k in T so that the binary-tree property continues to hold.
- The idea is to do like in Tree-search, but to use two pointers to walk down th tree, one pointing to a node, the other to the node's parent.
- When we cannot go down any further, then we attach the new key under the last parent found.
- One can distinguish 2 cases, when the tree T is empty (that is root(T) = NIL), and when it is not empty.
- To insert key k we create node (k,NIL,NIL,NIL) and a pointer z to it.
  ```

Tree-insert: code

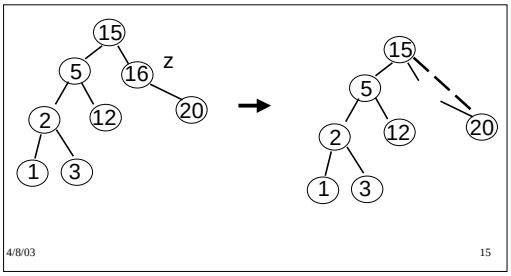
```
Tree-Insert(T,z)
    IF root(T) = NIL
      THEN root(T) := z
      ELSE x := root(T)
           y := NIL
           WHILE x ≠ NIL
             DO y := x
             IF key[z] < key[x]
               THEN x := left[x]
               ELSE x := right[x]
             END IF
           END WHILE
           parent[z] := y
           IF key[z] < key[y]
             THEN left[y] := z
             ELSE right[y] := z
           END IF
    END IF
```


Tree-delete(z)

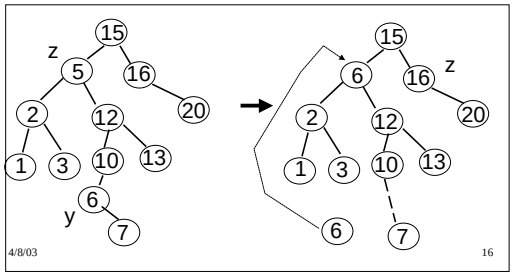
- We have a pointer z to a node in T to be deleted.
- Case 1: z has no children. Then just remove it.



- We have a pointer z to a node in T to be deleted.
- Case 2: z has 1 child. Then remove it and by-pass.



- Case 3: z has two children. Find y = successor(z). y has at most 1 child. Bypass y and remove y. Replace z with y.



Homework

- HW3.3 (CLR 13.3-6)
  For any binary tree T and any nodes x and y in T if we call Tree-delete(T,x) and then Tree-delete(T,y) we get the same final tree as calling Tree-delete(T,y) and then Tree-delete(T,x)?
  Argue why this is true or give a counter-example.

Conclusions

- Binary search tree can store keys and allow for the following operations:
  Search, insert, delete, successor, predecessor, minimum, maximum in time O(h), where h is the height of the tree.
- Binary search tree allow to find the elements in sorted order in time linear in the number of the elements in the tree.
- The height of a binary tree h is a number between $\lfloor \log_2 n \rfloor$ and n-1.
- We will see methods for forcing a tree to be balanced so that it has a small height.