CH-231-A

Algorithms and Data Structures

ADS

Lecture 22

Dr. Kinga Lipskoch

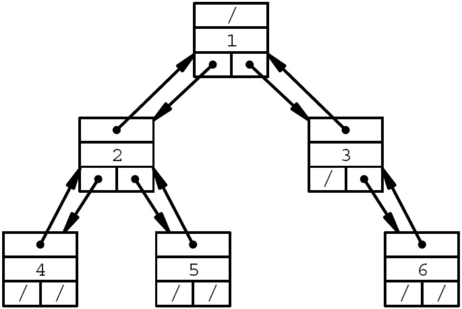
Spring 2020

Representing Rooted Trees

- Traversing a rooted tree requires us to know about the hierarchical relationships of their nodes.
- Similar to linked list implementations, such relationships can be stored by using pointers.

Binary Tree

- Binary trees T have an attribute $T.root$.
- They consist of nodes x with attributes $x.parent$ (short $x.p$), $x.left$, and $x.right$ in addition to $x.key$.



d-ary Trees

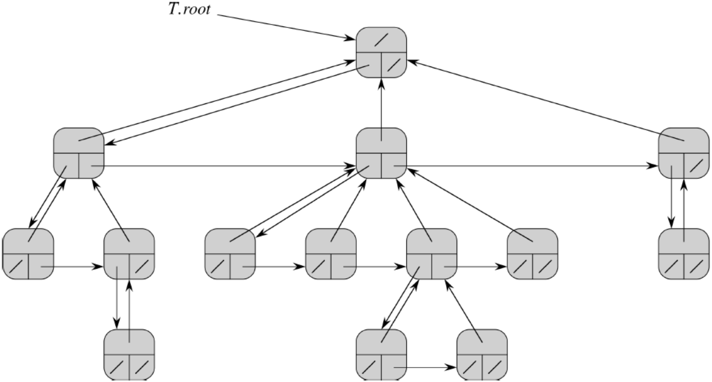
- d-ary trees are rooted trees with at most d children per node.
- They can be handled analogously to binary trees.

```
struct node {
    int val;
    node* parent;
    node* child[d];
};

typedef node* tree;
```

Rooted Trees with Arbitrary Branching

Rooted trees T with arbitrary branching consist of nodes x with attributes $x.p$, $x.leftmost\text{-}child$, and $x.right\text{-}sibling$ in addition to $x.key$.



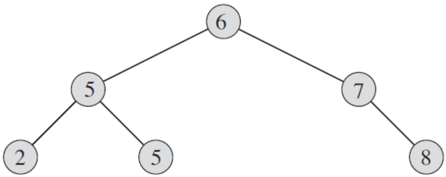
Discussion

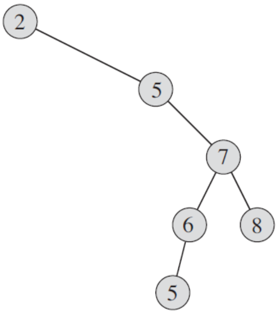
- Representing trees with pointers allows for a simple and intuitive representation.
- It also allows for a dynamic data management.
- Modifying operations can be implemented efficiently.
- However, extra memory requirements exist for storing the pointers.

Binary Search Tree: Definition

- A binary search tree (BST) is a binary tree with the following property:
  - Let x be a node of the BST.
  - If y is a node in the left subtree of x, then $y.key \leq x.key$.
  - If y is a node in the right subtree of x, then $x.key \leq y.key$.
- The idea of a BST data structure is to support efficient dynamic set operations, many in $O(h)$, where h is the tree's height.

Binary Search Tree: Examples





Query: In Order Visit

- Visit all nodes in order and execute an operation:

```
Function DFS-Inorder-Visit(Node n)
if n = NIL then return;
DFS-Inorder-Visit(n.left) ;
n.Operation() ;
DFS-Inorder-Visit(n.right) ;
```

- The operation could, e.g., be printing the key.
- This tree traversal is also referred to as in-order tree walk.
- Time complexity (n = number of nodes):
  $O(nk)$ when assuming that the operation is in $O(k)$.

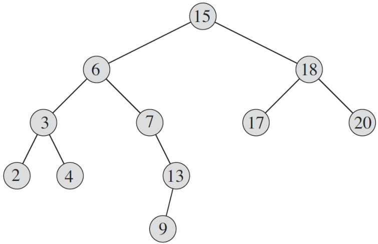
Query: Searching

- Recursive tree search:

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

- Iterative tree search:

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```



Time complexity: $O(h)$

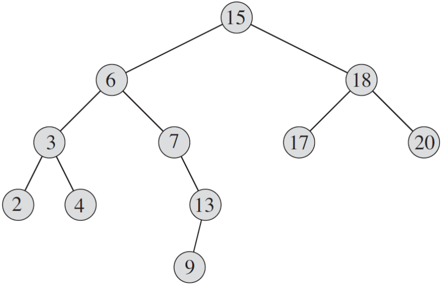
Query: Finding Minimum / Maximum

TREE-MINIMUM(x)

```
while x.left ≠ NIL
    x = x.left
return x
```

TREE-MAXIMUM(x)

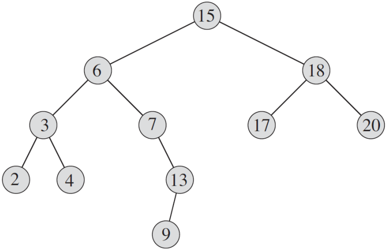
```
while x.right ≠ NIL
    x = x.right
return x
```



Time complexity: $O(h)$

Query: Finding Successor (In Order)

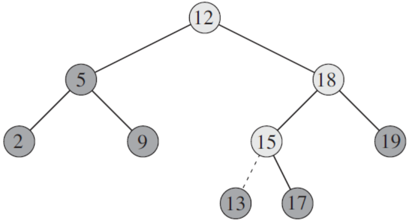
```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```



Time complexity: $O(h)$

Modify Operation: Insertion (In Order)

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```



Time complexity: $O(h)$

Modify Operation: Transplant

Replaces a subtree rooted at node u with a subtree rooted at node v.

```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

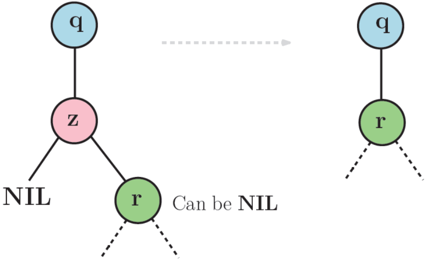
Remarks:

- $u.p$ can be nil.
- v can be nil.
- Time complexity: O(1)

Modify Operation: Deletion (1)

Case 1:

Deleted node z has no or only right child.

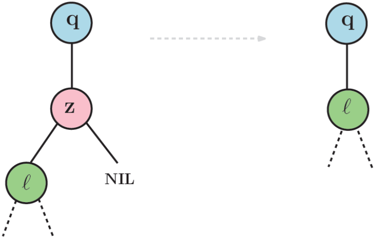



```
if z.left == NIL
    TRANSPLANT(T, z, z.right)
```

Modify Operation: Deletion (2)

Case 2:

Deleted node z has only left child.



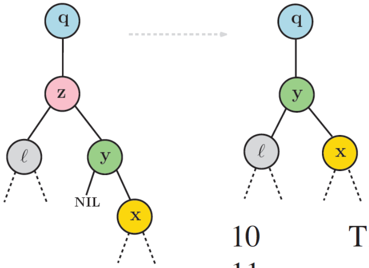

```
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
```

Remark: For both cases, it does not matter whether z is *q.left* or *q.right*.

Modify Operation: Deletion (3)

Case 3a:

Deleted node z has both children and $Successor(z) = z.right$.

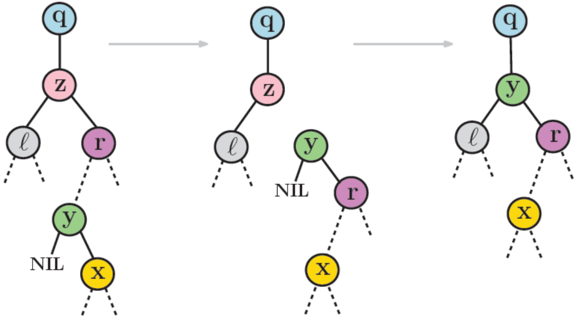



```
TRANSPLANT(T, z, y)
y.left = z.left
y.left.p = y
```

Modify Operation: Deletion (4)

Case 3b:

Deleted node z has both children and $Successor(z) = y \neq z.right$.



Modify Operation: Deletion

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

Time complexity: $O(h)$

Binary Search Tree: Summary

- BST provides all basic dynamic set operations in $O(h)$ running time, including:
  - Search
  - Minimum
  - Maximum
  - Predecessor
  - Successor
  - Insert
  - Delete
- Hence, BST operations are fast if h is small, i.e., if the tree is balanced. Then, $O(h) = O(\lg n)$.