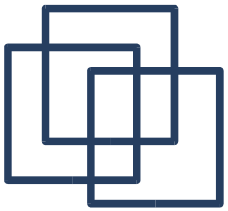
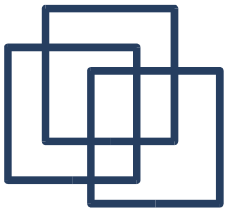
Binary Search Trees



Introduction

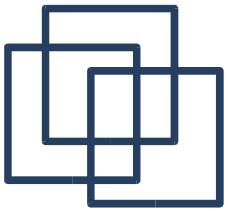
- Support for many operations, e.g., search, min, max, predecessor, successor, insert, delete.
- Suitable as dictionaries and priority queues.
- Important parameter: *height of the tree*.
  - basic operations proportional to the height
  - *E[height]* of random binary trees $\Theta(\lg n)$.
- Binary tree: every node in the tree has ≤2 children. See trees from last week.



Binary Search Trees

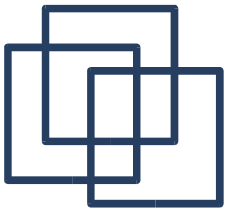
- Keys stored satisfied the following property:
  - for a node x and a node y in the *left* subtree of x, $key[y] \leq key[x]$
  - for a node x and a node y in the *right* subtree of x, $key[y] \geq key[x]$.
- Very similar to quicksort.
- **inorder_tree_walk(x):**

  if $x \neq$NIL **then** inorder_tree_walk(left(x))
  print key(x)
  inorder_tree_walk(right(x))



Searching

- **iterative_search(*x*,*k*)**:
  while $x \neq$NIL **and** $k \neq$key(x) **do**
    x:= k<key(x) ? left(x) : right(x)
  return x
- Running time is *O(height)*.
- **tree_min(*x*)**: **while** left(x)≠NIL **do** x:=left(x)
  return x
- **tree_max(*x*)**: **while** right(x)≠NIL do x:=right(x)
  return x



Successors (Sorted Enumeration)

- **tree_successor(*x*):**

  if right(*x*)≠NIL **then**

  return tree_min(right(*x*))

  y:=parent(*x*)

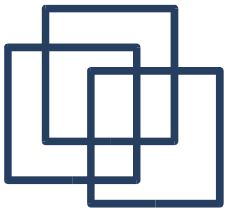
  while *y*≠NIL **and** *x*==right(*y*) do

  x:=*y*

  y:=parent(*y*)

  return *y*

- Again *O(height)*



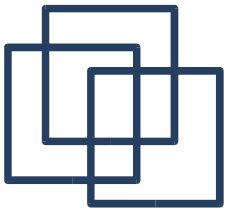
Insertion

- Change the tree, but keep the BST property!
- **tree_insert(*T*,*z*)**: // simplified

  y:=search_leaf(T,x)

  parent(z):=y

  fix_child(y,z)
- Add a new leaf everytime.



Deletion

- Change the tree, but keep the BST property!
- delete(T,z): // simplified

  if z is a leaf **then** remove z

  else if z has one child **then** splice out z

  else if z has two children **then**

    y:=tree_successor(T,z)

    remove y

    replace z by y
- Optimized in the book, again *O(height)*.