CS5101: Design and Analysis of Algorithms

Search Trees



Arijit Mondal

Dept of CSE

arijit@iitp.ac.in

https://www.iitp.ac.in/~arijit/

Sorted arrays

- Search – return a pointer to an object with the key k

Sorted arrays

- Search – return a pointer to an object with the key k
- Min/Max – return a pointer to the object with the smallest/largest key

Sorted arrays

- Search – return a pointer to an object with the key k
- Min/Max – return a pointer to the object with the smallest/largest key
- Predecessor/Successor – given a pointer to an object in the data structure, return a pointer to the object with the next-smallest (respectively, next-largest) key. If the given object has the minimum (respectively, maximum) key, report "none"

Sorted arrays

- Search – return a pointer to an object with the key k
- Min/Max – return a pointer to the object with the smallest/largest key
- Predecessor/Successor – given a pointer to an object in the data structure, return a pointer to the object with the next-smallest (respectively, next-largest) key. If the given object has the minimum (respectively, maximum) key, report "none"
- Sorted output – output the objects in the data structure one by one in order of their keys

Sorted arrays

- Search – return a pointer to an object with the key k
- Min/Max – return a pointer to the object with the smallest/largest key
- Predecessor/Successor – given a pointer to an object in the data structure, return a pointer to the object with the next-smallest (respectively, next-largest) key. If the given object has the minimum (respectively, maximum) key, report "none"
- Sorted output – output the objects in the data structure one by one in order of their keys
- Select – return a pointer to the object in the data structure with the ith-smallest key

Sorted arrays

- Search – return a pointer to an object with the key k
- Min/Max – return a pointer to the object with the smallest/largest key
- Predecessor/Successor – given a pointer to an object in the data structure, return a pointer to the object with the next-smallest (respectively, next-largest) key. If the given object has the minimum (respectively, maximum) key, report "none"
- Sorted output – output the objects in the data structure one by one in order of their keys
- Select – return a pointer to the object in the data structure with the ith-smallest key
- Rank – given a key k, return the number of objects in the data structure with key at most k

Sorted arrays: unsupported operations

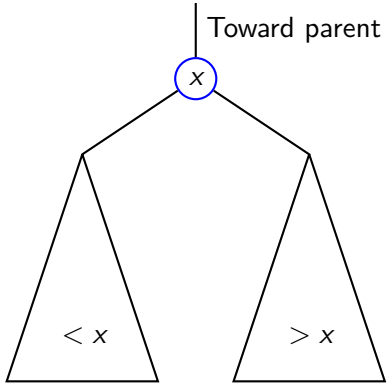
- Insert – given a new object x, add x to the data structure
- Delete – for a key k, delete an object with key k from the data structure, if one exists

Search trees

Operation	Sorted array	Balanced search trees
Search	$O(\lg n)$	$O(\lg n)$
Min	$O(1)$	$O(\lg n)$
Max	$O(1)$	$O(\lg n)$
Predecessor	$O(\lg n)$	$O(\lg n)$
Successor	$O(\lg n)$	$O(\lg n)$
Sorted output	$O(n)$	$O(n)$
Select	$O(1)$	$O(\lg n)$
Rank	$O(\lg n)$	$O(\lg n)$
Insert	$O(n)$	$O(\lg n)$
Delete	$O(n)$	$O(\lg n)$

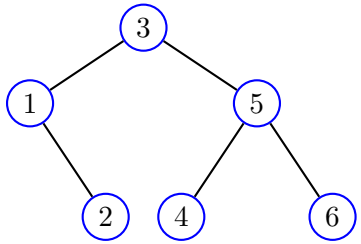
Search tree property

- For every object x, objects in x's left subtree have keys smaller than that of x
- For every object x, objects in x's right subtree have keys larger than that of x



Search tree pointers

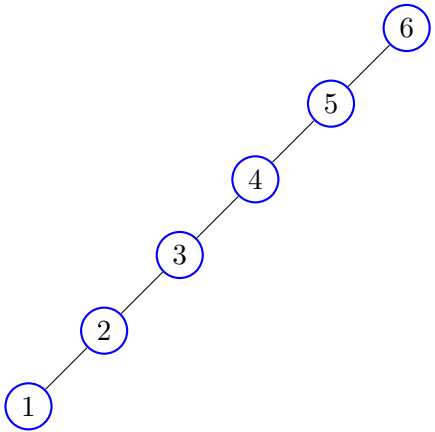
- Typically, for a given node we store a key and left, right, parent pointers
- 'parent' pointer is optional. We may store many additional information depending on the requirements



node	parent	left	right
1	3	null	2
2	1	null	null
3	null	1	5
4	5	null	null
5	3	4	null
6	5	null	null

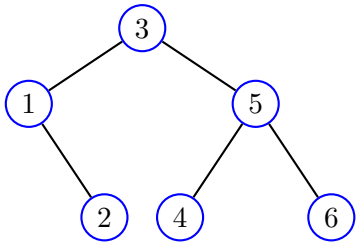
Search tree height

- The height of a tree is defined as the length of a longest path from its root to a leaf
- Height: $\approx \lg n$ for perfectly balanced tree, $(n-1)$ for chain



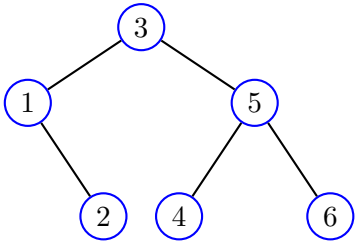
Search operation

- For a key k, return a pointer to an object in the data structure with key k (or report that no such object exists)
- Steps:
  1. Start at root node
  2. Repeatedly traverse left and right child pointers, as appropriate (left if k is less than the current node's key, right if k is bigger)
  3. Return a pointer to an object with key k (if found) or "none" (upon reaching a null pointer)



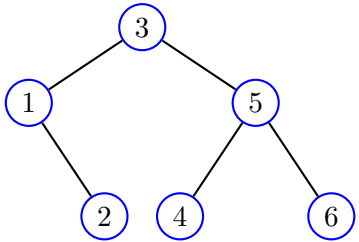
Min & Max operation

- Min (Max): return a pointer to the object in the data structure with the smallest (respectively, largest) key



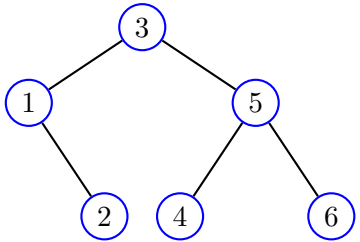
Min & Max operation

- Min (Max): return a pointer to the object in the data structure with the smallest (respectively, largest) key
- Steps:
  1. Start at the root node
  2. Traverse left child pointers (right child pointers) as long as possible, until encountering a null pointer
  3. Return a pointer to the last object visited
- Time complexity: $O(height)$



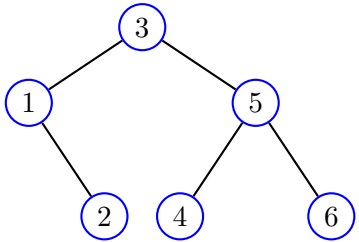
Predecessor

- Given a pointer to an object in the data structure, return a pointer to the object with the next-smallest key. (If the object has the minimum key, report none)



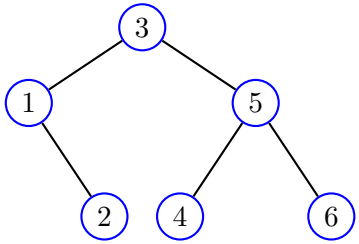
Predecessor

- Given a pointer to an object in the data structure, return a pointer to the object with the next-smallest key. (If the object has the minimum key, report none)
- Steps:
  1. If x's left subtree is non-empty, return the result of Max applied to this subtree
  2. Otherwise, traverse parent pointers upward toward the root. If the traversal visits consecutive nodes y and z with y a right child of z, return a pointer to z
  3. Otherwise report none
- Time complexity: $O(height)$



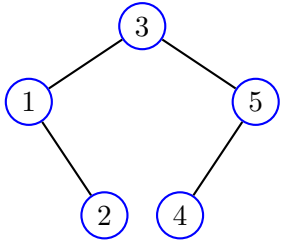
Sorted output

- Output the objects in the data structure one by one in order of their keys
- Steps:
  1. Recursively call print on the root's left sub tree
  2. Output the object at the root
  3. Recursively call print on the root's right sub tree
- Time complexity: $O(n)$
- This is inorder traversal. Explore – preorder, postorder traversal.



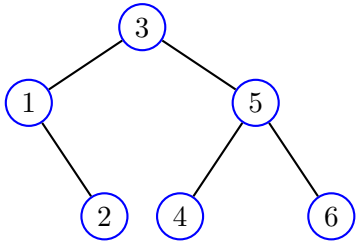
Insert

- Given a new object x, add x to the data structure
- Steps:
  1. Start at the root node
  2. Repeatedly traverse left and right child pointers, as appropriate (left if k is at most the current node's key, right if it's bigger), until a null pointer is encountered
  3. Replace the null pointer with one to the new object. Set the new node's parent pointer to its parent, and its child pointers to null
- Time complexity: $O(\mathit{height})$

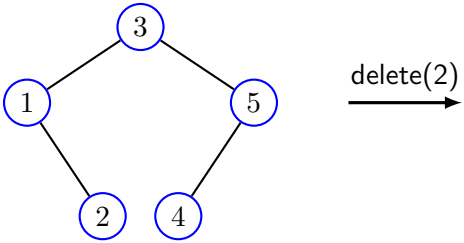


Insert

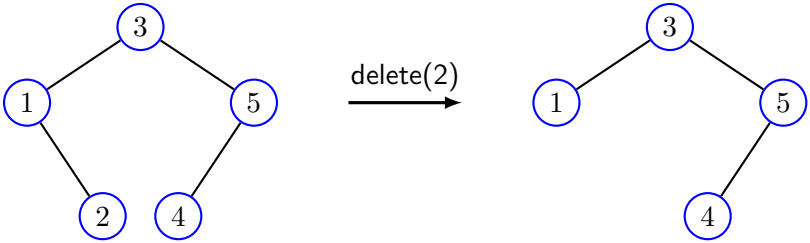
- Given a new object x, add x to the data structure
- Steps:
  1. Start at the root node
  2. Repeatedly traverse left and right child pointers, as appropriate (left if k is at most the current node's key, right if it's bigger), until a null pointer is encountered
  3. Replace the null pointer with one to the new object. Set the new node's parent pointer to its parent, and its child pointers to null
- Time complexity: $O(height)$



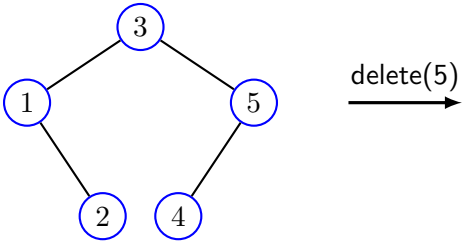
Delete-1



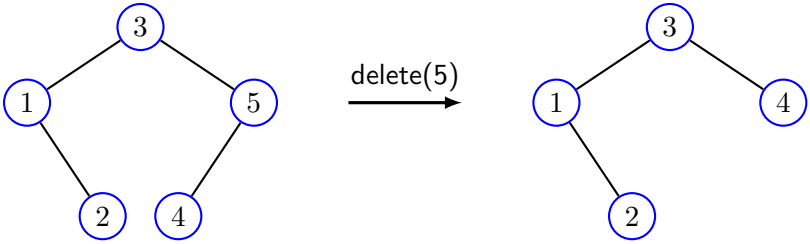
Delete-1



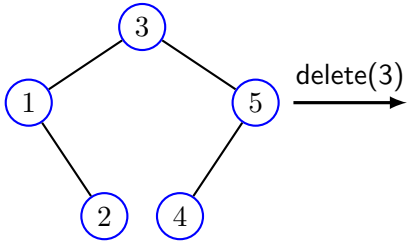
Delete-2



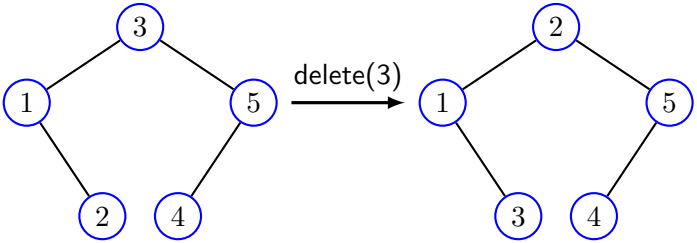
Delete-2



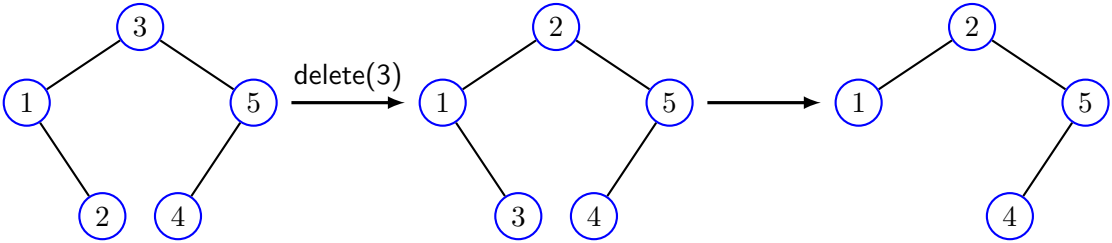
Delete-3



Delete-3



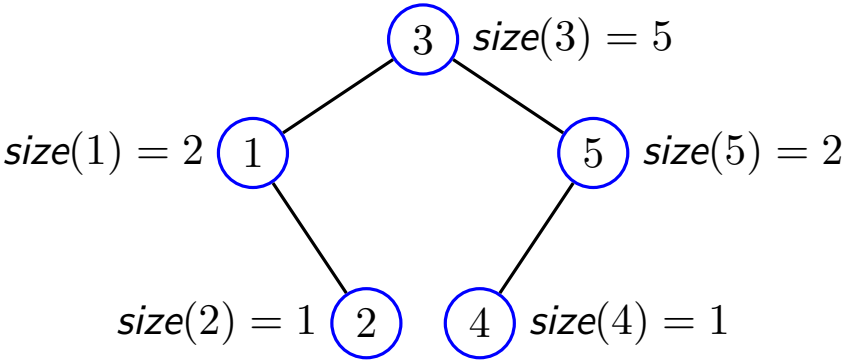
Delete-3



Delete

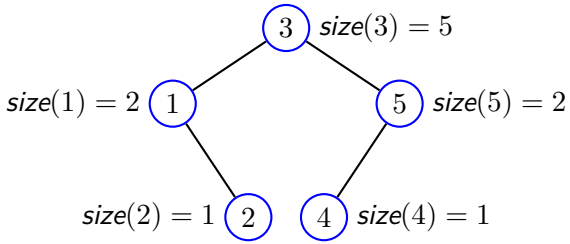
- For a key k, delete an object with key k from the search tree, if one exists
- Steps:
  1. Use Search to locate an object x with key k. (If no such object exists, halt.)
  2. If x has no children, delete x by setting the appropriate child pointer of x's parent to null. (If x was the root, the new tree is empty.)
  3. If x has one child, splice x out by rewiring the appropriate child pointer of x's parent to x's child, and the parent pointer of x's child to x's parent. (If x was the root, its child becomes the new root.)
  4. Otherwise, swap x with the object in its left subtree that has the biggest key, and delete x from its new position (where it has at most one child)
- Time complexity: $O(\mathit{height})$

Select: Augmentation



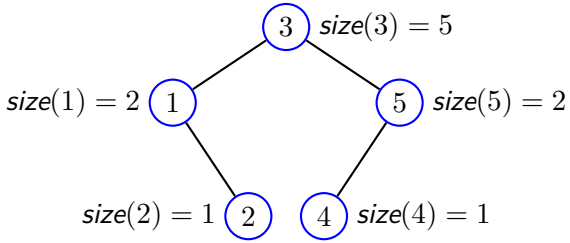
Select using Augmentation

- Given a number i, between 1 and the number of objects, return a pointer to the object in the data structure with the *i*th-smallest key



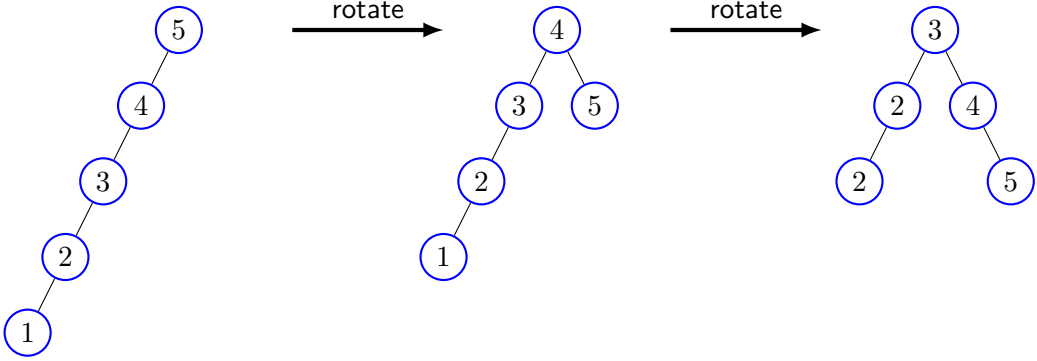
Select using Augmentation

- Given a number i, between 1 and the number of objects, return a pointer to the object in the data structure with the ith-smallest key
- Steps:
  1. Start at the root and let j be the size of its left subtree. (If it has no left child pointer, then $j = 0$)
  2. If $i = j + 1$, return a pointer to the root
  3. If $i < j + 1$, recursively compute the ith-smallest key in the left subtree
  4. If $i > j + 1$, recursively compute the $(i - j - 1)$th smallest key in the right subtree

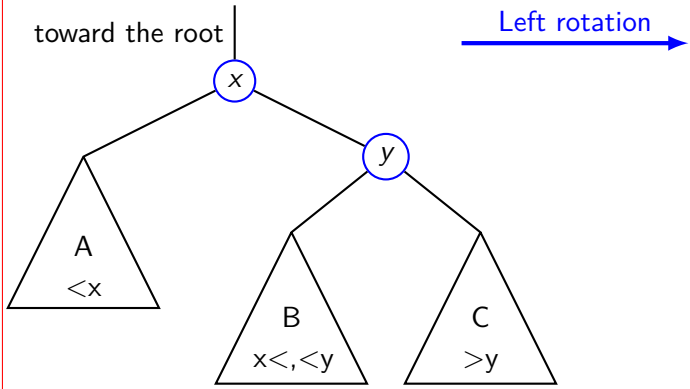


Rotation

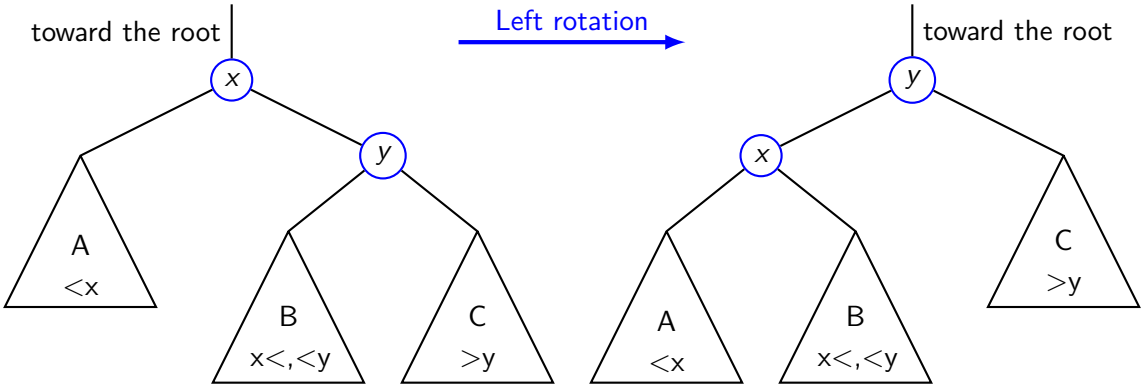
- Rotation can help improving the height of the tree
- Balanced tree can be constructed



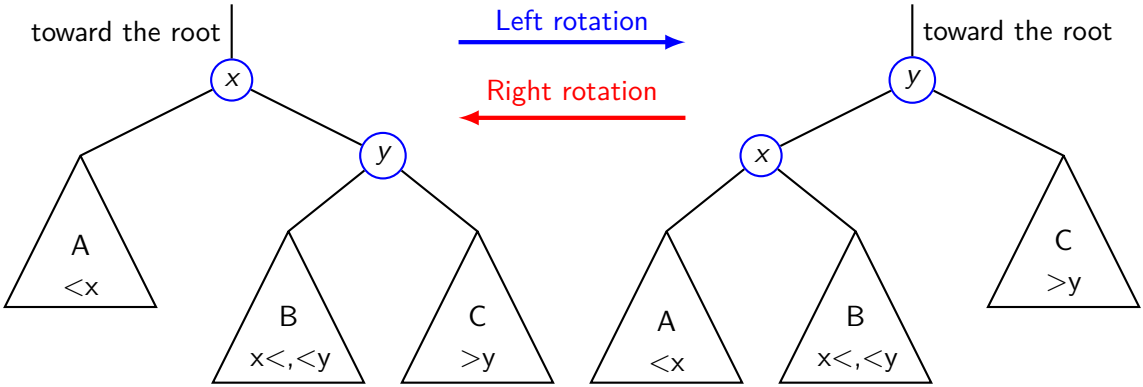
Left Rotation



Left Rotation



Left Rotation



Average case complexity

Thank you!