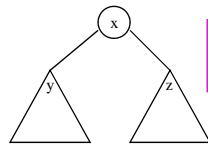


## Binary Search Trees

- In addition to insert/delete:
  - » Heaps supported **min/max**.
  - » Hashing supported **search**.
  - » What if we want both **min/max/search**, and also **pred/succ**?
- Binary Search trees:

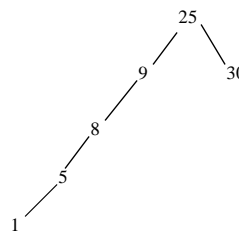
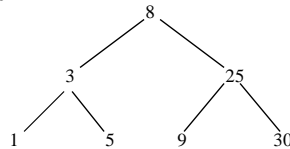


$\forall a \in \text{left tree}$	$key(a) \leq key(x)$
$\forall a \in \text{right tree}$	$key(a) \geq key(x)$

113

## Examples

- Legal Binary Search Trees:



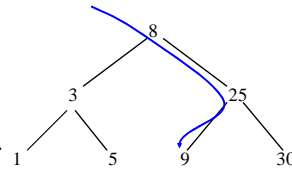
- In-Order traversal:
  - InOrder(left(x))
  - print(x)
  - InOrder(right(x))

- Note that given Binary tree, can output sorted in  $O(n)$  time!  
Gives **lower bound** on constructing Binary Tree.  
(Compare with building a Heap)

114

## Searching in Binary Tree

- Check if "current node" is =x or =NIL.  
Equal to x -- return,  
Equal to NIL -- return "not found"
- else:  
if  $x \leq$  current key ---- search in the left tree  
otherwise ---- search in the right tree.

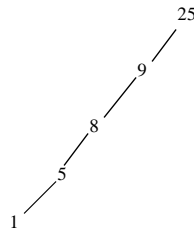


- "Go down the tree, turning right/left as appropriate..."
- Running time:  $\Theta(h)$ ,  $h$ =height of the tree.
- Note that this was impossible to do with a heap

115

## Inserting into Binary Tree

- Insertion: search for key, and put it in the first empty space.
- Insertion takes  $\Theta(h)$ .
- Sort:
  - » Insert item-by-item,
  - » in-order walk.
  - »  $\Theta(n^2)$ ...



116

## Relation to Quicksort

- Randomly permute input.
- Consider example: **3 1 8 2 6 7 5**
  - » **Quicksort** chooses 3, then compares 1,8,2,6,7,5 to 3.  
Then chooses 1, compares to 2  
chooses 8, compares 6,7,5 to 8.
  - » **Binary Tree**: chooses 3, places as root  
Then chooses 1, compares with 3, put in place.  
chooses 8, compares with 3, put in place  
etc...
  - » Overall, **same comparisons, only different order** !!

117

## Successor/Predecessor

- **Min/max** - go all the way left or all the way right.
- **Successor**:
 

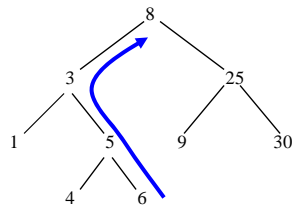
```

      if right(x) != NIL, return TreeMin(right(x))
      y=parent(x)
      while y != NIL & x=right(y)
          x=y
          y=parent(x)
      return y
      
```

}

← Easy case

← Go up the tree to the LEFT



- Successor of 6 is 8. Predecessor of 8 is 6.
- What if we go "up left" and reach root, i.e. cannot ever go "up right" ?

118

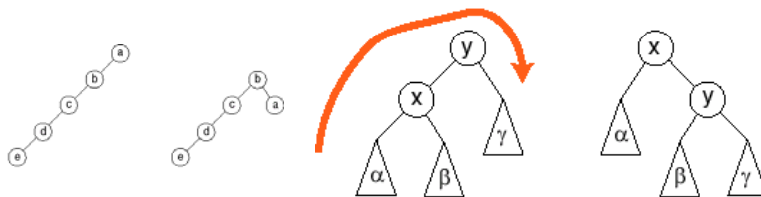
## Deletion

- 3 Cases:
  - » No children – simple delete, replace pointer from parent with null
  - » 1 child – delete, redirect pointer from parent to point to child
  - » 2 children – cannot simply delete
- 3rd case: put successor(x) instead of x.
  - » Binary Tree property satisfied.
  - » Delete “hole” using case 1 or 2.
  - » Why ??
  - » (successor does not have left child !)

119

## Summary

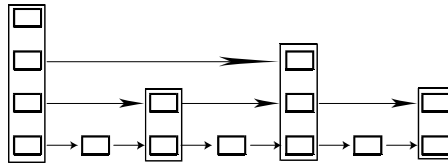
- Data structure implements insert/delete/search/pred/successor
- Speed depends on height of the tree, in general can be large
- Need a way to “rebalance the tree”
- Possible to rebalance so that height  $< 2 \log n$
- Basic operation: “Rotation”



120

## Skip List

- Simple data structure, *easier* to implement than red/black trees.
- Sorted linked list with “skip” pointers.
- Construction:
  - » Every element in the “bottom list” (list 0)
  - » Element passed to list 1 with *probability 1/2*.
  - » In general, element in list  $i$  is promoted/added to list  $i+1$  with prob.  $1/2$ .
  - » Stop promoting after a node is promoted  $2 \log n$  times



127