CS2010: ALGORITHMS AND DATA STRUCTURES

Lecture 14: Binary Search Trees (2)

Vasileios Koutavas

School of Computer Science and Statistics
Trinity College Dublin

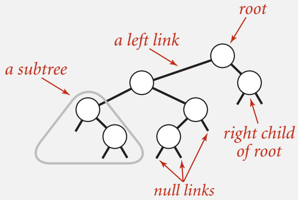
http://algs4.cs.princeton.edu

3.2 Binary Search Trees

- *BSTs*
- *ordered operations*
- *deletion*

Binary search trees

Definition. A BST is a binary tree in symmetric order.

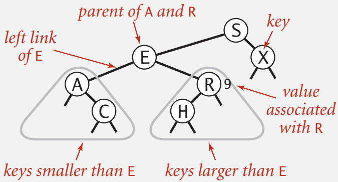


A binary tree is either:

- Empty.
- Two disjoint binary trees (left and right).

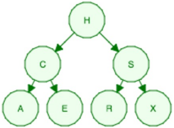
Symmetric order. Each node has a key, and every node's key is:

- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.

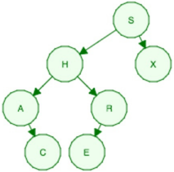


Q: Which of the following are Binary Search Trees? Why?

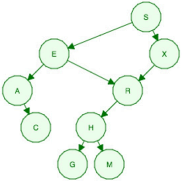
(1)



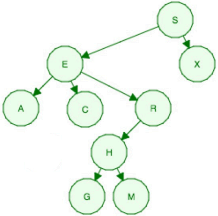
(2)



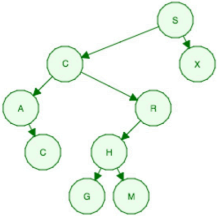
(3)



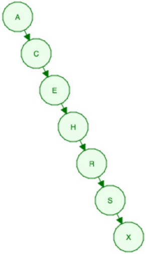
(4)



(5)



(6)



ST implementations: summary

implementation	guarantee		average case		operations on keys
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	$\frac{1}{2} N$	N	`equals()`
binary search (ordered array)	$\lg N$	N	$\lg N$	$\frac{1}{2} N$	`compareTo()`
BST	N	N	$1.39 \lg N$	$1.39 \lg N$	`compareTo()`

Why not shuffle to ensure a (probabilistic) guarantee of 4.311 ln N?

http://algs4.cs.princeton.edu

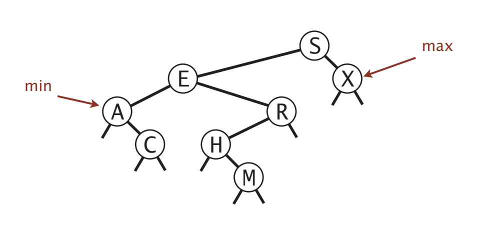
3.2 Binary Search Trees

- *BSTs*
- *ordered operations*
- *deletion*

Minimum and maximum

Minimum. Smallest key in table.

Maximum. Largest key in table.

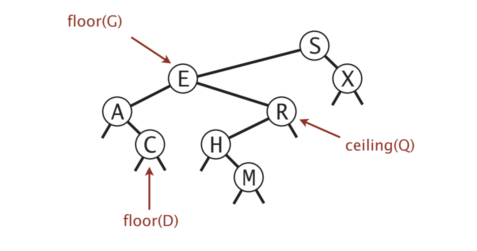


Q. How to find the min / max?

Floor and ceiling

Floor. Largest key ≤ a given key.

Ceiling. Smallest key ≥ a given key.



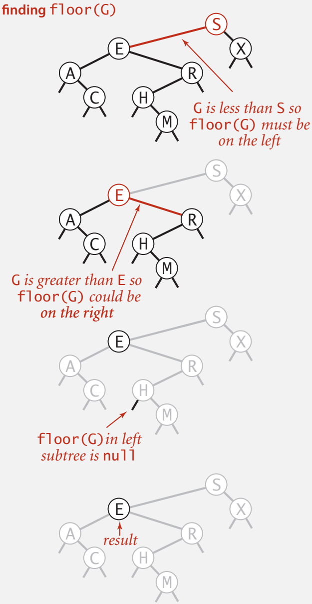
Q. How to find the floor / ceiling?

Computing the floor

Case 1. [k equals the key in the node]
The floor of k is k.

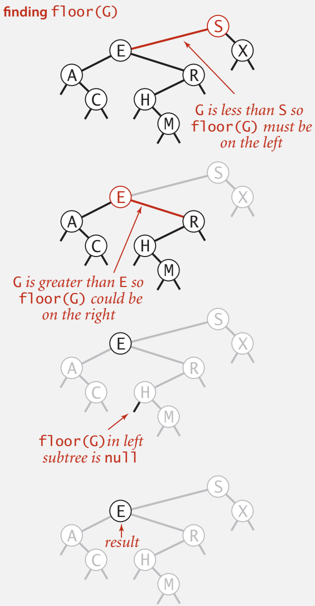
Case 2. [k is less than the key in the node]
The floor of k is in the left subtree.

Case 3. [k is greater than the key in the node]
The floor of k is in the right subtree
(if there is any key $\leq k$ in right subtree);
otherwise it is the key in the node.

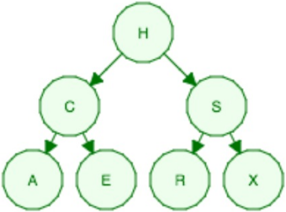


Computing the floor

```
public Key floor(Key key)
{
   Node x = floor(root, key);
   if (x == null) return null;
   return x.key;
}
private Node floor(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);

   if (cmp == 0) return x;

   if (cmp < 0)  return floor(x.left, key);

   Node t = floor(x.right, key);
   if (t != null) return t;
   else           return x;

}
```

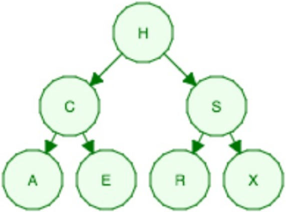


- **rank(Key k)**: how many keys less than k?
- **select(int n)**: what key has rank n?



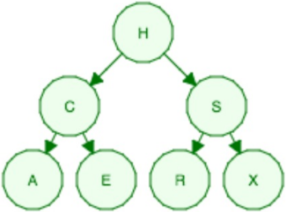
- **Q**: what is the rank of 'S'?

- **rank(Key k)**: how many keys less than k?
- **select(int n)**: what key has rank n?



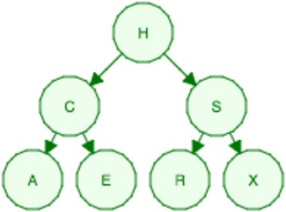
- **Q**: what is the rank of 'S'? 5 (5 keys less than 'S' in the tree)

- **rank(Key k)**: how many keys less than k?
- **select(int n)**: what key has rank n?



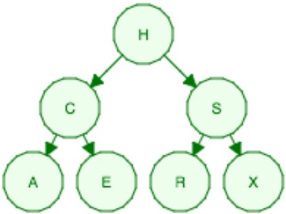
- **Q**: what is the rank of 'S'? 5 (5 keys less than 'S' in the tree)
- **Q**: what key has rank 4?

- **rank(Key k)**: how many keys less than k?
- **select(int n)**: what key has rank n?



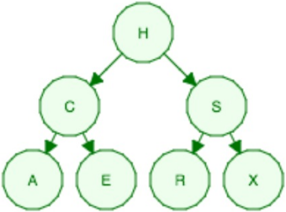
- **Q**: what is the rank of 'S'? 5 (5 keys less than 'S' in the tree)
- **Q**: what key has rank 4? 'R' (4 keys less than 'R' in the tree)

- **rank(Key k)**: how many keys less than k?
- **select(int n)**: what key has rank n?



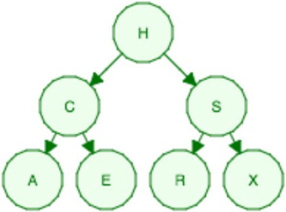
- **Q**: what is the rank of 'S'? 5 (5 keys less than 'S' in the tree)
- **Q**: what key has rank 4? 'R' (4 keys less than 'R' in the tree)
- **Q**: what is the rank of 'Q' ('Q' is not in the tree)?

- **rank(Key k)**: how many keys less than k?
- **select(int n)**: what key has rank n?



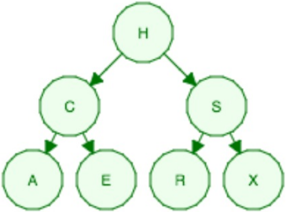
- **Q**: what is the rank of 'S'? 5 (5 keys less than 'S' in the tree)
- **Q**: what key has rank 4? 'R' (4 keys less than 'R' in the tree)
- **Q**: what is the rank of 'Q' ('Q' is not in the tree)? 4 (4 keys in the tree are less than 'Q')

- **rank(Key k)**: how many keys less than k?
- **select(int n)**: what key has rank n?



- **Q**: what is the rank of 'S'? 5 (5 keys less than 'S' in the tree)
- **Q**: what key has rank 4? 'R' (4 keys less than 'R' in the tree)
- **Q**: what is the rank of 'Q' ('Q' is not in the tree)? 4 (4 keys in the tree are less than 'Q')
- **Q**: what key has rank 7?

- **rank(Key k)**: how many keys less than k?
- **select(int n)**: what key has rank n?

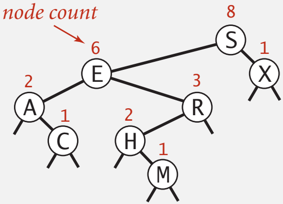


- **Q**: what is the rank of 'S'? 5 (5 keys less than 'S' in the tree)
- **Q**: what key has rank 4? 'R' (4 keys less than 'R' in the tree)
- **Q**: what is the rank of 'Q' ('Q' is not in the tree)? 4 (4 keys in the tree are less than 'Q')
- **Q**: what key has rank 7? no key in the tree has this rank (no key has 7 keys smaller than it in the tree)

Rank and select

Q. How to implement `rank()` and `select()` efficiently?

A. In each node, we store the number of nodes in the subtree rooted at that node; to implement `size()`, return the count at the root.



BST implementation: subtree counts

```
private class Node
{
   private Key key;
   private Value val;
   private Node left;
   private Node right;
   private int count;
}
```

number of nodes in subtree

```
public int size()
{  return size(root);  }
```

```
private int size(Node x)
{
   if (x == null) return 0;
   return x.count;
}
```

ok to call when x is null

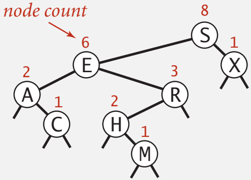
```
private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val, 1);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) x.left  = put(x.left,  key, val);
   else if (cmp  > 0) x.right = put(x.right, key, val);
   else if (cmp == 0) x.val = val;
   x.count = 1 + size(x.left) + size(x.right);
   return x;
}
```

initialize subtree count to 1

Rank

Rank. How many keys $< k$?

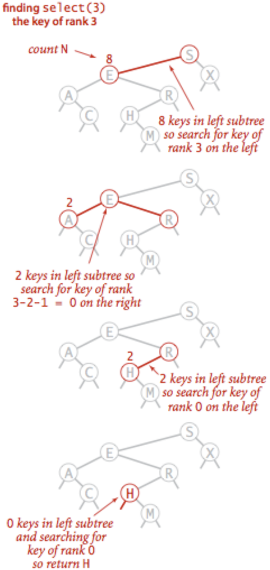
Easy recursive algorithm (3 cases!)




```
public int rank(Key key)
{  return rank(key, root);  }

private int rank(Key key, Node x)
{
   if (x == null) return 0;
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) return rank(key, x.left);
   else if (cmp  > 0) return 1 + size(x.left) + rank(key, x.right);
   else if (cmp == 0) return size(x.left);
}
```

Select. Find the key with rank n.

```
public Key select(int n) {
    if (n < 0 || n >= size()) return null;
    Node x = select(root, n);
    return x.key;
}

private Node select(Node x, int n) {
    if (x == null) return null;
    int t = size(x.left);
    if      (t > n) return select(x.left,  n);
    else if (t < n) return select(x.right, n-t-1);
    else            return x;
}
```



Selection in a BST

Task: Process all nodes of the tree.

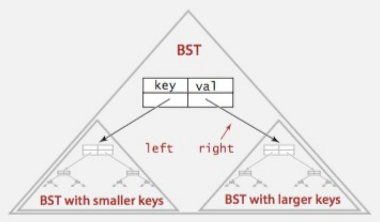
Purpose: To print all nodes, to add all nodes in a datastructure (e.g. queue), etc.

Three kinds of traversals:

→ inorder: for each node:
  1. traverse the left subtree
  2. **process the node**
  3. traverse the right subtree

→ preorder: for each node:
  1. **process the node**
  2. traverse the left subtree
  3. traverse the right subtree

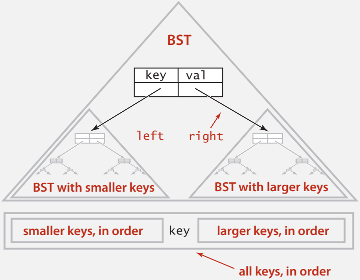
→ postorder: for each node:
  1. traverse the left subtree
  2. traverse the right subtree
  3. **process the node**



Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

```
public Iterable<Key> keys()
{
    Queue<Key> q = new Queue<Key>();
    inorder(root, q);
    return q;
}

private void inorder(Node x, Queue<Key> q)
{
   if (x == null) return;
   inorder(x.left, q);
   q.enqueue(x.key);
   inorder(x.right, q);
}
```



Property. Inorder traversal of a BST yields keys in ascending order.

BST: ordered symbol table operations summary

	sequential search	binary search	BST
search	N	$\lg N$	h
insert	N	N	h
min / max	N	1	h
floor / ceiling	N	$\lg N$	h
rank	N	$\lg N$	h
select	N	1	h
ordered iteration	$N \log N$	N	N

h = height of BST
(proportional to log N
if keys inserted in random order)

Worst case: h = O(N)

order of growth of running time of ordered symbol table operations

http://algs4.cs.princeton.edu

3.2 Binary Search Trees

- BSTs
- ordered operations
- *deletion*

ST implementations: summary

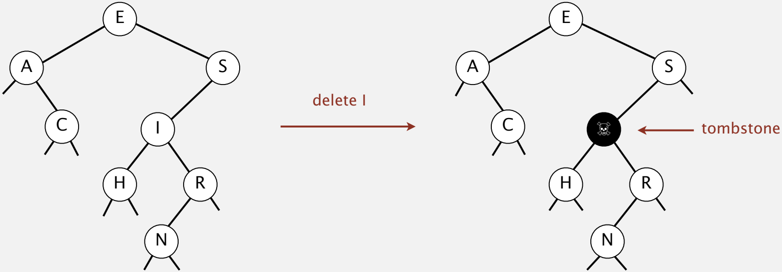
implementation	guarantee			average case			ordered ops?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	½ N	N	½ N		`equals()`
binary search (ordered array)	lg N	N	N	lg N	½ N	½ N	✔	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	? ? ?	✔	`compareTo()`

Next. Deletion in BSTs.

BST deletion: lazy approach

To remove a node with a given key:

- Set its value to `null`.
- Leave key in tree to guide search (but don't consider it equal in search).



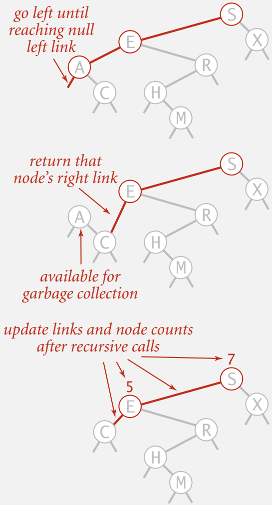
Cost. $\sim 2 \ln N'$ per insert, search, and delete (if keys in random order), where N' is the number of key-value pairs ever inserted in the BST.

Unsatisfactory solution. Tombstone (memory) overload.

Deleting the minimum

To delete the minimum key:

- Go left until finding a node with a null left link.
- Replace that node by its right link.
- Update subtree counts.

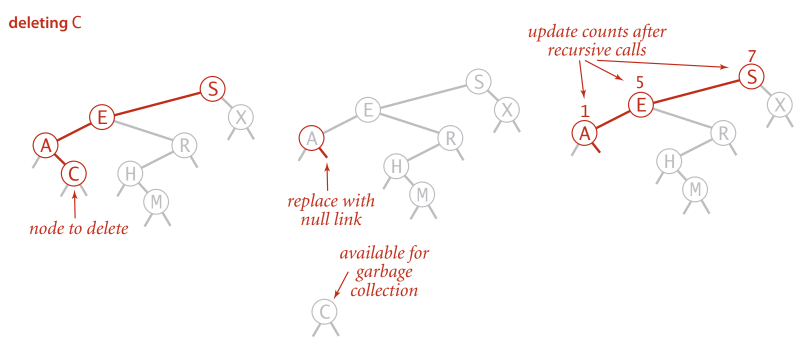



```
public void deleteMin()
{  root = deleteMin(root);  }

private Node deleteMin(Node x)
{
   if (x.left == null) return x.right;
   x.left = deleteMin(x.left);
   x.count = 1 + size(x.left) + size(x.right);
   return x;
}
```

Hibbard deletion

To delete a node with key `k`: search for node `t` containing key `k`.

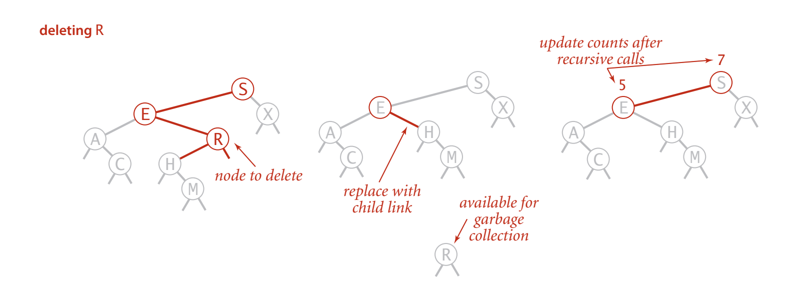
Case 0. [0 children] Delete `t` by setting parent link to null.



Hibbard deletion

To delete a node with key `k`: search for node `t` containing key `k`.

Case 1. [1 child] Delete `t` by replacing parent link.

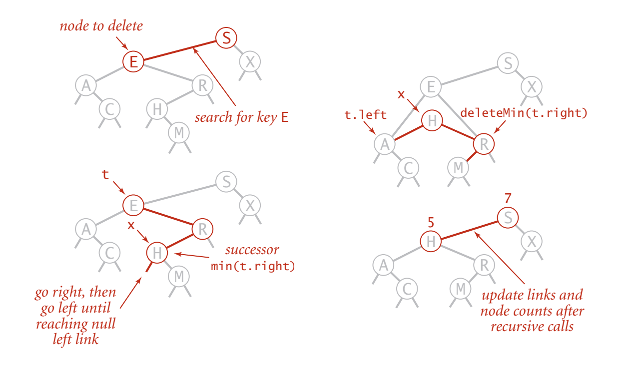


Hibbard deletion

To delete a node with key `k`: search for node `t` containing key `k`.

Case 2. [2 children]

- Find successor `x` of `t`. ← x has no left child
- Delete the minimum in `t`'s right subtree. ← but don't garbage collect x
- Put `x` in `t`'s spot. ← still a BST



Hibbard deletion: Java implementation

```
public void delete(Key key)
{  root = delete(root, key);  }

private Node delete(Node x, Key key) {
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = delete(x.left,  key);
   else if (cmp > 0) x.right = delete(x.right, key);
   else {
      if (x.right == null) return x.left;
      if (x.left  == null) return x.right;

      Node t = x;
      x = min(t.right);
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.count = size(x.left) + size(x.right) + 1;
   return x;
}
```

search for key

no right child

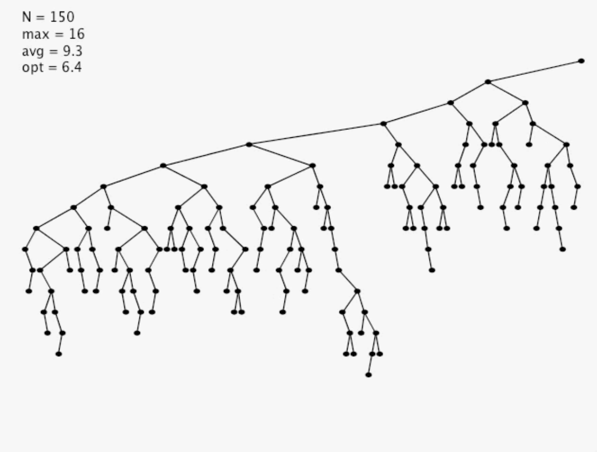
no left child

replace with successor

update subtree counts

Hibbard deletion: analysis

Unsatisfactory solution. Not symmetric.



Surprising consequence. Trees not random (!) $\Rightarrow$ $\sqrt{N}$ per op.

Longstanding open problem. Simple and efficient delete for BSTs.

ST implementations: summary

implementation	guarantee			average case			ordered ops?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	$½\,N$	N	$½\,N$		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$½\,N$	$½\,N$	✔	`compareTo()`
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	`compareTo()`

other operations also become $\sqrt{N}$ if deletions allowed

Next lecture. Guarantee logarithmic performance for all operations.