Binary Search Trees

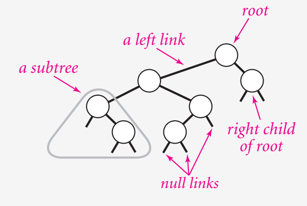
- binary search tree
- ordered operations
- deletion

References:
Algorithms in Java, Chapter 12
http://www.cs.princeton.edu/algs4/42bst

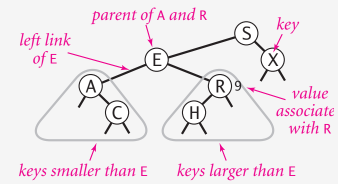
Binary search trees

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

A binary tree is either:
- Empty.
- A key-value pair and two disjoint binary trees.



Symmetric order. Every node's key is:
- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.



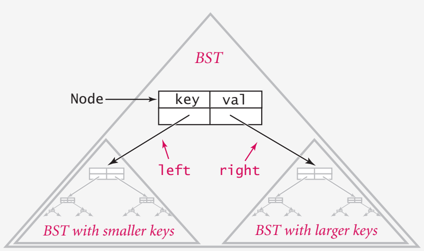
BST representation in Java

A BST is a reference to a root node.

A Node is comprised of four fields:
- A Key and a Value.
- A reference to the left and right subtree.

smaller keys　larger keys

```
private class Node
{
   private Key key;
   private Value val;
   private Node left, right;
   public Node(Key key, Value val)
   {
      this.key = key;
      this.val = val;
   }
}
```



Key and Value are generic types; Key is Comparable

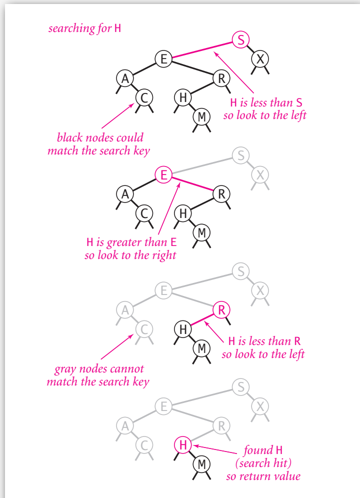
BST implementation (skeleton)

```
public class BST<Key extends Comparable<Key>, Value>
{
   private Node root;

   private class Node
   {  /* see previous slide */  }

   public void put(Key key, Value val)
   {  /* see next slides */  }

   public Value get(Key key)
   {  /* see next slides */  }

   public void delete(Key key)
   {  /* see next slides */  }

   public Iterable<Key> iterator()
   {  /* see next slides */  }

}
```

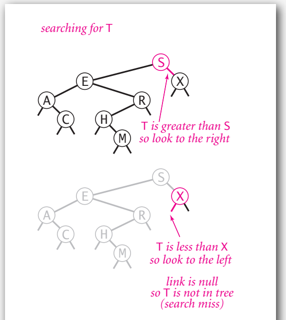
root of BST

BST search

Get. Return value corresponding to given key, or `null` if no such key.



Successful search in a BST



Unsuccessful search in a BST

BST search: Java implementation

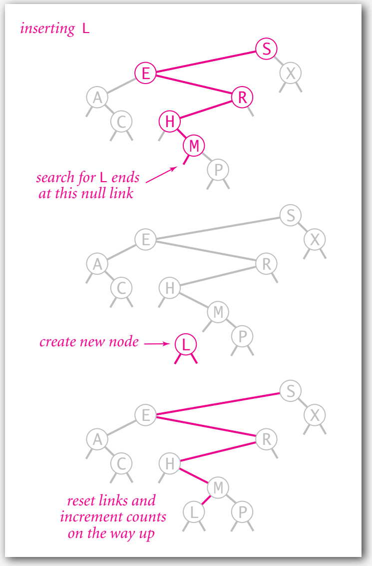
Get. Return value corresponding to given key, or `null` if no such key.

```
public Value get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if      (cmp  < 0) x = x.left;
      else if (cmp  > 0) x = x.right;
      else if (cmp == 0) return x.val;
   }
   return null;
}
```

Running time. Proportional to depth of node.

BST insert

Put. Associate value with key.



BST insert: Java implementation

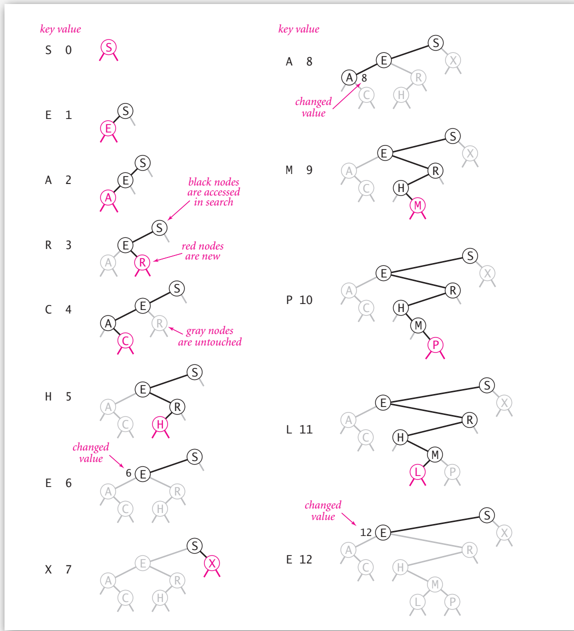
Put. Associate value with key.

```
public void put(Key key, Value val)
{  root = put(root, key, val);  }

private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   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;
   return x;
}
```

concise, but tricky, recursive code; read carefully!

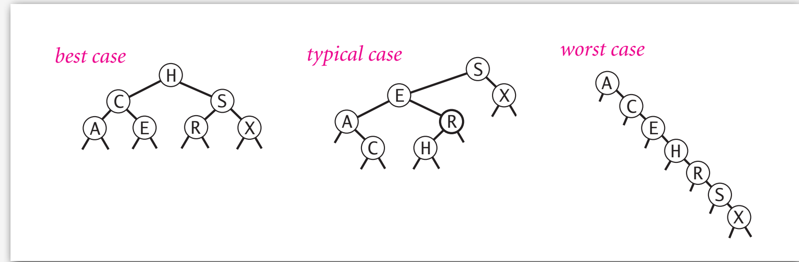
Running time. Proportional to depth of node.

BST trace: standard indexing client



Tree shape

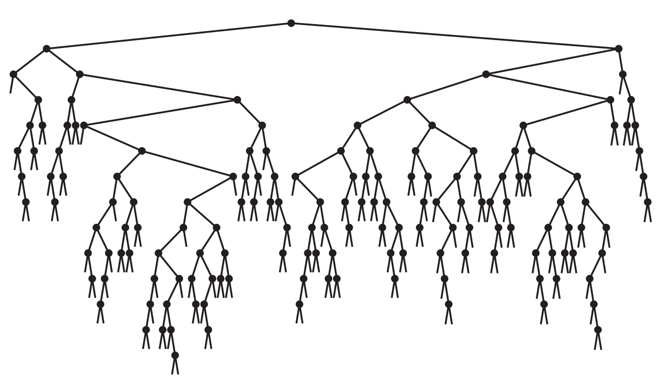
- Many BSTs correspond to same set of keys.
- Cost of search/insert is proportional to depth of node.



Remark. Tree shape depends on order of insertion.

BST insertion: random order

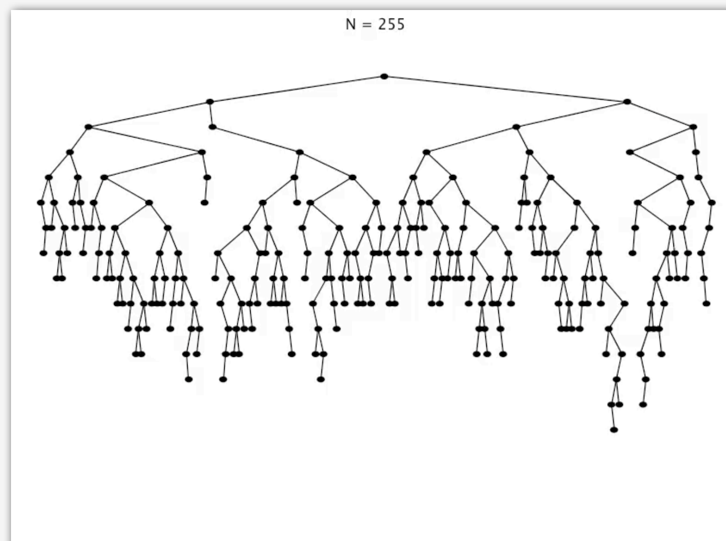
Observation. If keys inserted in random order, tree stays flat.



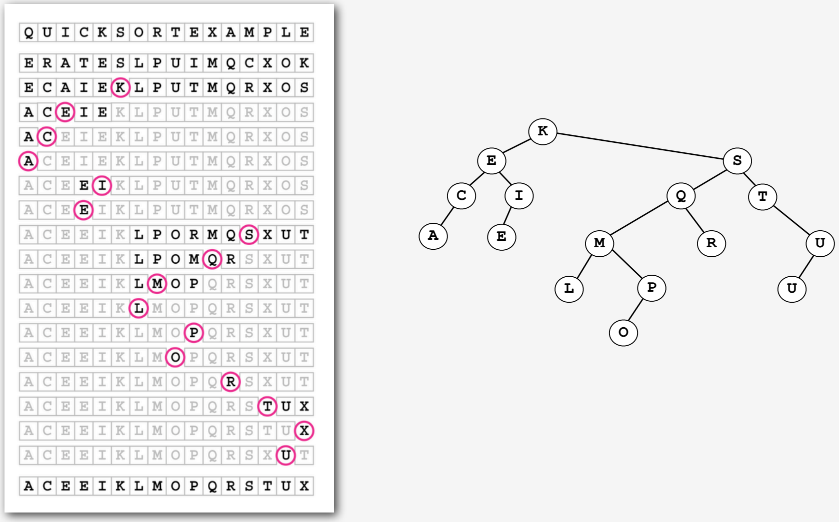
Typical BST built from random keys (N = 256)

BST insertion: random order visualization

Ex. Insert keys in random order.



Correspondence between BSTs and quicksort partitioning



Remark. Correspondence is 1-1 if no duplicate keys.

BSTs: mathematical analysis

Proposition. If keys are inserted in random order, the expected number of compares for a search/insert is ~ 2 ln N.

Pf. 1-1 correspondence with quicksort partitioning.

Proposition. [Reed, 2003] If keys are inserted in random order, expected height of tree is ~ 4.311 ln N.

But... Worst-case for search/insert/height is N.
(exponentially small chance when keys are inserted in random order)

ST implementations: summary

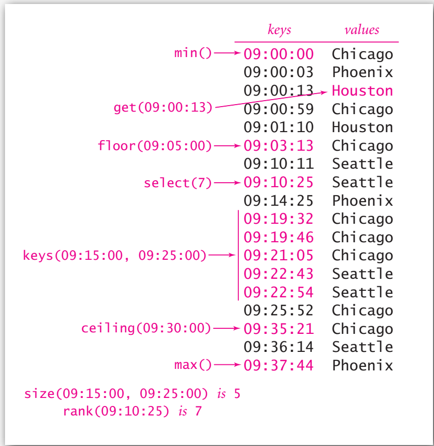
implementation	guarantee		average case		ordered ops?	operations on keys
	search	insert	search hit	insert		
unordered array	N	N	N/2	N	no	`equals()`
unordered list	N	N	N/2	N	no	`equals()`
ordered array	lg N	N	lg N	N/2	yes	`compareTo()`
ordered list	N	N	N/2	N/2	yes	`compareTo()`
BST	N	N	1.39 lg N	1.39 lg N	?	`compareTo()`

Next challenge. Ordered symbol tables ops.

- basic implementations
- randomized BSTs
- **ordered symbol table ops**

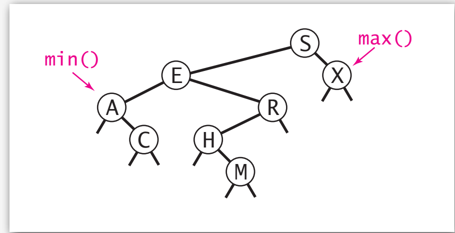
Ordered symbol table operations

Minimum. Smallest key in table.
Maximum. Largest key in table.
Floor. Largest key ≤ to a given key.
Ceiling. Smallest key ≥ to a given key.
Rank. Number of keys < than given key.
Select. Key of given rank.
Size. Number of keys in a given range.
Iterator. All keys in order.



Minimum and maximum

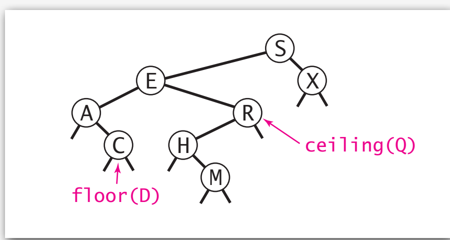
Minimum. Smallest key in table.
Maximum. Largest key in table.



Q. How to find the min / max.
A.

Floor and ceiling

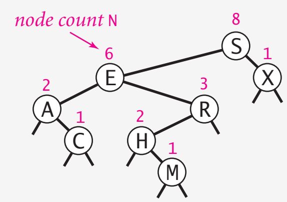
Floor. Largest key ≤ to a given key.
Ceiling. Smallest key ≥ to a given key.



Q. How to find the floor /ceiling.
A.

Rank

Rank. How many keys < k ?

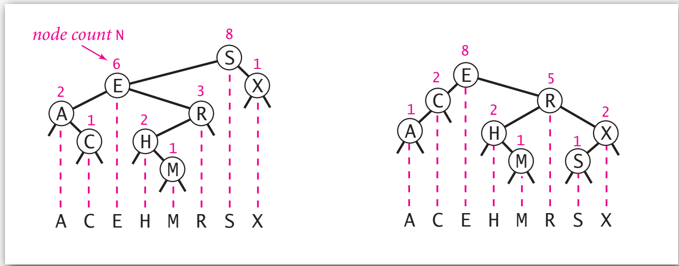



```
public int rank(Key key)
{  return rank(key, root);  }

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              return size(x.left);
}
```

how to implement size() efficiently?

Subtree counts

In each node, we store the number of nodes in the subtree rooted at that node.



Remark. This facilitates efficient implementation of `rank()` and `select()`.

BST implementation: subtree counts

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

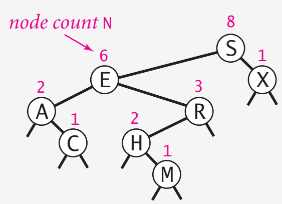
nodes in subtree

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

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

```
private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   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.N = 1 + size(x.left) + size(x.right);
   return x;
}
```


Range count

Range count. How many keys between `lo` and `hi`?



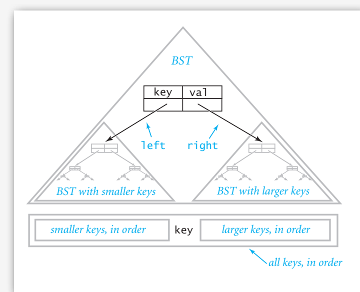

```
public int size(Key lo, Key hi)
{
   if (contains(hi)) return rank(hi) - rank(lo) - 1;
   else              return rank(hi) - rank(lo);
}
```

number of keys < hi

Inorder traversal

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

```
public Iterable<Key> allKeys()
{
    Queue<Key> q = new Queue<Key>();
    inorder(root, queue);
    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.

Inorder traversal

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

```
visit(S)
  visit(E)
    visit(A)
      enqueue A
      visit(C)
        enqueue C
    enqueue E
    visit(R)
      visit(H)
        enqueue H
        visit(M)
          enqueue M
      print R
  enqueue S
  visit(X)
    enqueue X
```

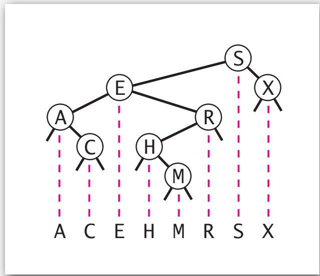
recursive calls

```
A
C
E
H
M
R
S
X
```

queue

```
S
S E
S E A
S E A C
S E R
S E R H
S E R H M
S X
```

function call stack



ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	`compareTo()`

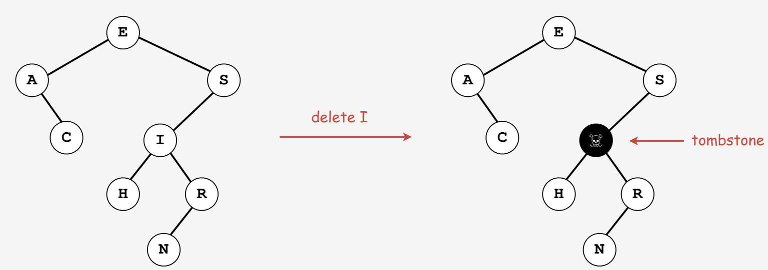
Next lecture. Can we guaranteed performance?

- basic implementations
- randomized BSTs
- **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 searches (but don't consider it equal to search key).



Cost. O(log N') per insert, search, and delete (if keys in random order), where N' is the number of elements ever inserted in the BST.

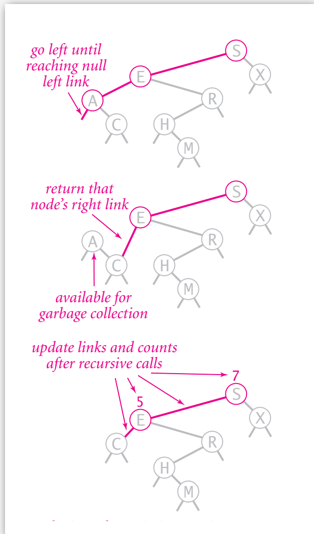
Unsatisfactory solution. Tombstone overload.

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.

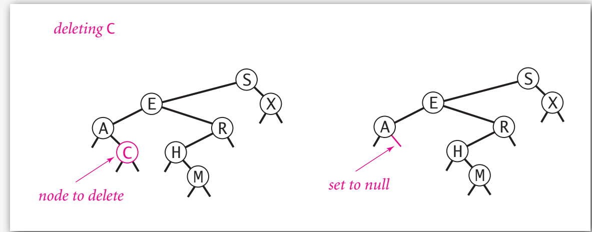
```
public void deleteMin()
{  root = deleteMin(root);  }

private Node deleteMin(Node x)
{
   if (x.left == null) return x.right;
   x.left = deleteMin(x.left);
   x.N = 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.

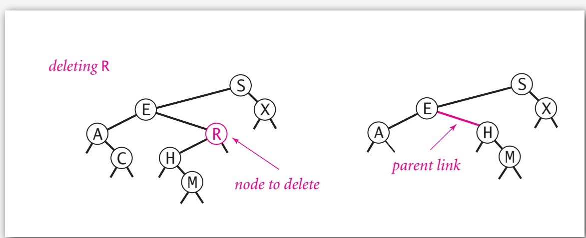
Case 0. [0 children] Delete t by setting parent link to null.



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.

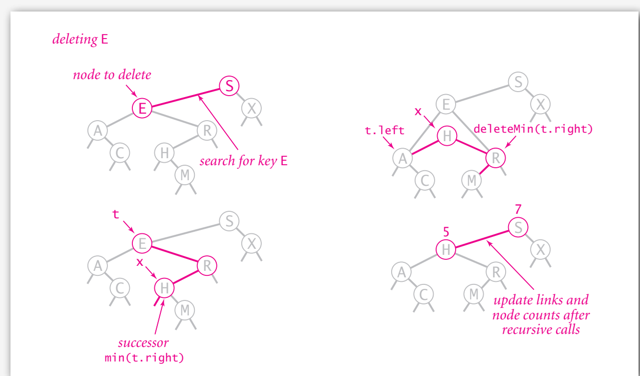


Hibbard deletion

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;

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

search for key

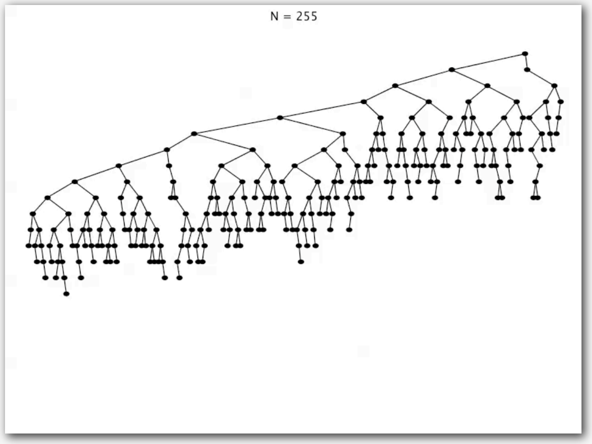
no right child

replace with successor

update subtree counts

Hibbard deletion: analysis

Unsatisfactory solution. Not symmetric.



Surprising consequence. Trees not random (!) ⇒ sqrt(N) per op.

Longstanding open problem. Simple and efficient delete for BSTs.

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	$\sqrt{N}$	yes	`compareTo()`

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

Next lecture. Guarantee logarithmic performance for all operations.