LECTURE 6 OF 12

Binary Search Trees

Data Structures & Algorithms

Based on Robert Sedgewick & Kevin Wayne

Algorithms, 4th Edition (Addison-Wesley, 2011)

Chapters 3.1 & 3.2

Duration: 4 hours (with breaks)

Learning Objectives

By the end of this lecture, students will be able to:

1. Define the symbol table abstraction and distinguish between ordered and unordered symbol table APIs, identifying the operations each supports.
2. Describe the binary search tree data structure, its invariant properties, and how the shape of a BST depends on the order of key insertions.
3. Implement core BST operations — search, insert, min, max, floor, ceiling, rank, select, and range queries — and trace their execution on concrete examples.
4. Explain and implement Hibbard deletion in BSTs, identifying the three cases (leaf, one child, two children) and tracing the algorithm step by step.
5. Analyze the expected and worst-case costs of BST operations, relating the expected height of a random BST to quicksort partitioning.
6. Compare BST performance against elementary symbol table implementations (unordered list, ordered array) and articulate when each is appropriate.

Section 1: Symbol Tables (~25 minutes)

1.1 The Symbol Table Abstraction

A **symbol table** is one of the most fundamental abstractions in computer science. At its core, it is a data structure that associates *keys* with *values*. You put key-value pairs into the structure, and later you retrieve a value by specifying its key. This pattern appears everywhere:

Application	Key	Value
Dictionary	word	definition
DNS lookup	domain name	IP address
Compiler	variable name	type and address
File system	filename	file contents location
Genomics	codon (e.g., ACG)	amino acid
Phone book	name	phone number
Bank	account number	balance
Web search	keyword	list of web pages

The symbol table abstraction captures all of these use cases in a single, clean API.

1.2 Basic Symbol Table API

DEFINITION 1.1 — SYMBOL TABLE. A *symbol table* is a collection of key-value pairs supporting the following operations:

```
public class ST<Key, Value>
---------------------------------------------------------------------------
         void  put(Key key, Value val)    // insert key-value pair
        Value  get(Key key)               // value associated with key
                                          // (null if key absent)
         void  delete(Key key)            // remove key and its value
      boolean  contains(Key key)          // is there a value for key?
      boolean  isEmpty()                  // is the table empty?
          int  size()                     // number of key-value pairs
Iterable<Key>  keys()                     // all the keys in the table
```

We adopt the following **conventions**, which simplify both the API semantics and implementations:

1. **No duplicate keys.** If a client inserts a key-value pair with a key that already exists, the new value replaces the old one. Think of this as an associative array: `st[key] = value` overwrites any previous association.

2. **No null values.** We use `null` in `get()` to signal that a key is absent. Consequently, we disallow `null` values — calling `put(key, null)` is equivalent to `delete(key)`.

3. **Keys must be non-null.** Passing a `null` key to any method throws an exception.

These conventions give us a clean invariant: a key is in the symbol table if and only if `get(key)` returns a non-null value.

1.3 Ordered Symbol Table API

When keys are **Comparable** (they support a total order), we can provide a much richer set of operations:

DEFINITION 1.2 — ORDERED SYMBOL TABLE. An *ordered symbol table* extends the basic symbol table with operations that exploit the ordering of keys:

```
public class OrderedST<Key extends Comparable<Key>, Value>
---------------------------------------------------------------------------
          Key  min()                      // smallest key
          Key  max()                      // largest key
          Key  floor(Key key)             // largest key <= given key
          Key  ceiling(Key key)           // smallest key >= given key
          int  rank(Key key)              // number of keys < given key
          Key  select(int k)             // key of rank k
          int  size(Key lo, Key hi)      // number of keys in [lo..hi]
Iterable<Key>  keys(Key lo, Key hi)      // keys in [lo..hi], in order
```

These operations are enormously useful. For example, `floor` and `ceiling` let you find the nearest match when an exact match does not exist — think of finding the closest stock price to a target, or the nearest scheduled departure time.

The `rank` and `select` operations are inverses of each other:

- `rank(key)` answers: "How many keys are strictly less than `key`?"
- `select(k)` answers: "Which key has exactly `k` keys less than it?"

Therefore `rank(select(k)) = k` for all valid `k`, and `select(rank(key)) = key` for all keys in the table.

1.4 Elementary Implementations

Before we get to BSTs, let us consider two elementary implementations to establish baselines.

Unordered Linked List (Sequential Search)

Maintain key-value pairs in a linked list, in no particular order.

- **Search:** Walk through the list, comparing the search key against each node's key. Cost: **N** compares in the worst case.
- **Insert:** Search for the key first (to check for duplicates). If found, update the value. If not found, add a new node at the front. Cost: **N** compares in the worst case.

```
Sequential search in an unordered linked list:

 [S,0] -> [E,1] -> [A,2] -> [R,3] -> [C,4] -> [H,5] -> null

 get("R"):  compare S, E, A, R -- found! (4 compares)
 get("Z"):  compare S, E, A, R, C, H -- not found (6 compares)
```

Ordered Array (Binary Search)

Maintain two parallel arrays — one for keys (in sorted order) and one for values. Use binary search to find keys.

- **Search:** Binary search on the sorted key array. Cost: **lg N** compares.
- **Insert:** Find the insertion point with binary search, then shift all larger entries to the right. Cost: **lg N** compares + **N** array accesses for shifting.

```
Binary search in an ordered array:

 keys[]:   A  C  E  H  R  S
 vals[]:   2  4  1  5  3  0

 rank("H"):
   lo=0, hi=5, mid=2 -> keys[2]="E" < "H" -> lo=3
   lo=3, hi=5, mid=4 -> keys[4]="R" > "H" -> hi=3
   lo=3, hi=3, mid=3 -> keys[3]="H" = "H" -> found at rank 3
```

1.5 Cost Summary for Elementary Implementations

Implementation	Search (worst)	Insert (worst)	Search (avg)	Insert (avg)	Ordered ops?
Unordered list	N	N	N/2	N	No
Ordered array	lg N	N	lg N	N/2	Yes

The unordered list gives fast insert but slow search. The ordered array gives fast search but slow insert (due to array shifting). Neither achieves efficient performance for both operations simultaneously. This is precisely the gap that binary search trees fill.

[5-minute break]

Section 2: Binary Search Tree Basics (~50 minutes)

2.1 BST Definition

DEFINITION 2.1 — BINARY SEARCH TREE. A *binary search tree* (BST) is a binary tree in which each node has a key (and an associated value), and the key in each node is:
- **Greater than** all keys in its **left** subtree, and
- **Less than** all keys in its **right** subtree.

This property — called the **BST invariant** or **symmetric order** — must hold for every node in the tree, not just the root.

```
A valid BST:

              S
             / \
            E   X
           / \
          A   R
           \  /
            C H

Invariant check at node S: left subtree keys {A,C,E,H,R} < S < X {right subtree}
Invariant check at node E: left subtree keys {A,C} < E < {H,R} right subtree
Invariant check at node R: left subtree keys {H} < R and right subtree is empty
Invariant check at node A: left subtree empty, right subtree keys {C} > A
```

2.2 Node Structure

Each node in a BST contains five fields:

```
private class Node {
    Key key;           // the key
    Value val;         // the associated value
    Node left;         // link to left subtree
    Node right;        // link to right subtree
    int size;          // number of nodes in subtree rooted here
}
```

The `size` field is crucial — it maintains the count of all nodes in the subtree rooted at this node (including the node itself). This count enables efficient `rank` and `select` operations. We maintain the invariant:

```
size(x) = size(x.left) + size(x.right) + 1
```

where `size(null) = 0`.

2.3 Search

Searching for a key in a BST is elegant: start at the root and follow links left or right depending on comparison results.

```
Algorithm: BST Search

get(Node x, Key key):
    if x is null:
        return null           // key not found
    compare = key.compareTo(x.key)
    if compare < 0:
        return get(x.left, key)     // search left subtree
    else if compare > 0:
        return get(x.right, key)    // search right subtree
    else:
        return x.val                // found!
```

Trace: Searching for "H" in the BST above.

```
            S            <-- compare "H" with "S": H < S, go left
           / \
          E   X          <-- compare "H" with "E": H > E, go right
         / \
        A   R            <-- compare "H" with "R": H < R, go left
         \  /
          C H            <-- compare "H" with "H": equal, FOUND! return value
```

The search follows the path: S -> E -> R -> H. Number of compares: **4**.

Trace: Searching for "G" (not present).

```
            S            <-- compare "G" with "S": G < S, go left
           / \
          E   X          <-- compare "G" with "E": G > E, go right
         / \
        A   R            <-- compare "G" with "R": G < R, go left
         \  /
          C H            <-- compare "G" with "H": G < H, go left
                         <-- H.left is null, return null (NOT FOUND)
```

The search follows the path: S -> E -> R -> H -> null. Number of compares: **4** (with one null check).

The key insight: a search examines nodes along a single path from the root to a leaf (or to the node containing the key). Therefore the cost is bounded by the height of the tree plus one.

2.4 Insert

Insertion is a natural extension of search. We search for the key: if we find it, we update its value; if we reach a null link, we create a new node there.

```
Algorithm: BST Insert

put(Node x, Key key, Value val):
    if x is null:
        return new Node(key, val, 1)    // create leaf node with size 1
    compare = key.compareTo(x.key)
    if compare < 0:
        x.left = put(x.left, key, val)
    else if compare > 0:
        x.right = put(x.right, key, val)
    else:
        x.val = val                      // key exists, update value
    x.size = 1 + size(x.left) + size(x.right)   // update count
    return x
```

Note the pattern: `x.left = put(x.left, key, val)`. This recursive structure both searches for the insertion point and stitches the new node back into the tree. On the way back up the recursion, it updates the `size` field of every ancestor.

Trace: Building a BST by inserting S, E, A, R, C, H, X in order.

```
Insert S:       S

Insert E:       S
              /
             E

Insert A:       S
              /
             E
           /
          A

Insert R:       S
              /
             E
           / \
          A   R

Insert C:       S
              /
             E
           / \
          A   R
           \
            C

Insert H:       S
              /
             E
           / \
          A   R
           \  /
            C H

Insert X:       S
              / \
             E   X
           / \
          A   R
```

```
             \  /
              C H
```

2.5 BST Shape Depends on Insertion Order

This is one of the most important points about BSTs: **the shape of a BST is entirely determined by the order in which keys are inserted** (assuming no deletions).

Consider inserting the same seven keys in different orders:

Order 1: H, C, S, A, E, R, X (median first, then balanced)

```
              H
             / \
            C   S
           / \ / \
          A  E R  X

Height = 2 (balanced!)
```

Order 2: A, C, E, H, R, S, X (sorted order)

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

Height = 6 (degenerate — essentially a linked list!)
```

Order 3: S, E, A, R, C, H, X (our original example)

```
              S
             / \
            E   X
           / \
          A   R
           \  /
            C H

Height = 3 (reasonably balanced)
```

All three trees contain exactly the same keys, but their heights — and therefore their search performance — differ dramatically.

2.6 Height Analysis

DEFINITION 2.2 — HEIGHT. The *height* of a BST is the length of the longest path from the root to a null link (counting the number of links, or equivalently, the number of nodes minus one on that path).

The height determines the worst-case cost of search and insert operations.

- **Best case:** The tree is perfectly balanced. Height = floor(lg N). Every search costs at most lg N + 1 compares.
- **Worst case:** The tree is completely skewed (degenerate). Height = N - 1. Every search costs up to N compares.
- **Average case:** For a tree built from N randomly ordered distinct keys, the expected height is approximately **4.311 ln N**.

THEOREM 2.1. *If N distinct keys are inserted into an initially empty BST in uniformly random order, the expected number of compares for a search hit is approximately 2 ln N (about 1.39 lg N), and the expected number of compares for an insertion is approximately 2 ln N.*

Proof sketch. The proof relies on a remarkable correspondence with quicksort. Consider searching for a key in a BST built by random insertions. The comparisons made during the search are exactly the same as the comparisons made to the search key during quicksort partitioning (where the root plays the role of the pivot). The analysis of quicksort (Lecture 3) established that the average number of compares to sort N items is ~2N ln N, so the average number of compares involving any particular element is ~2 ln N.

More precisely, let C_N denote the expected internal path length of a random BST with N nodes divided by N (the average depth). Then:

```
C_N = 2(N+1)/N * H_N - 2

where H_N = 1 + 1/2 + 1/3 + ... + 1/N (the N-th harmonic number)
```

Since H_N ~ ln N + 0.5772..., we get C_N ~ 2 ln N ~ 1.39 lg N.

This means a random BST is only about 39% worse than a perfectly balanced BST for search operations — quite good!

PROPOSITION 2.1. *The expected height of a random BST with N keys is approximately 4.311 ln N.*

This result, proved by Reed (2003), is much harder than the average search cost. The constant 4.311 is 1/c where c is the unique solution to 2c * e^(1-2c) = 1 with c > 1, giving c approximately 0.2320.

The gap between expected search cost (~2 ln N) and expected height (~4.311 ln N) is explained by the fact that most searches do not go all the way to the deepest leaf — they terminate at various depths.

[10-minute break]

Section 3: BST Operations (~45 minutes)

3.1 Minimum and Maximum

Finding the minimum key in a BST is trivial: follow left links from the root until you reach a node with no left child. That node contains the smallest key.

```
Algorithm: BST Minimum

min(Node x):
    if x.left is null:
        return x
    return min(x.left)
```

Symmetrically, the maximum follows right links.

```
Trace: Finding min in our BST:

              S
             / \
            E   X
           / \
          A   R
           \  /
            C H

S -> E -> A  (A.left is null, so min = A)
```

Cost: proportional to the height of the tree.

3.2 Floor and Ceiling

The **floor** of a key k is the largest key in the BST that is less than or equal to k. The **ceiling** is the smallest key greater than or equal to k.

```
Algorithm: BST Floor

floor(Node x, Key key):
    if x is null:
        return null
    compare = key.compareTo(x.key)
    if compare == 0:
        return x                          // exact match
    if compare < 0:
        return floor(x.left, key)         // floor must be in left subtree
    // compare > 0: floor might be x, or might be in right subtree
    t = floor(x.right, key)
    if t is not null:
        return t                          // found a closer floor in right subtree
    else:
        return x                          // x itself is the floor
```

The logic is subtle in the third case. When `key > x.key`, we know that `x.key` is a candidate for the floor (it is less than `key`). But there might be a larger key in the right subtree that is still <= `key`. So we check the right subtree first; if we find something there, it is a better (larger) floor. Otherwise, `x` itself is the floor.

Trace: floor("G") in our BST.

```
             S    (8)
            / \
           E    X
          / \
         A   R
          \  /
           C H

floor(S, "G"):  "G" < "S", recurse on left subtree
  floor(E, "G"):  "G" > "E", E is a candidate. Check right subtree.
    floor(R, "G"):  "G" < "R", recurse on left subtree
      floor(H, "G"):  "G" < "H", recurse on left subtree
        floor(null, "G"):  return null
      return null (H has no suitable floor in its left subtree)
    return null (no closer floor found in R's left subtree)
  t = null, so return E itself
return E

Result: floor("G") = "E"
```

Trace: floor("R") in our BST.

```
floor(S, "R"):  "R" < "S", recurse on left subtree
  floor(E, "R"):  "R" > "E", E is a candidate. Check right subtree.
    floor(R, "R"):  "R" == "R", exact match!
    return R
  t = R (not null), return R
return R

Result: floor("R") = "R"
```

The ceiling operation is perfectly symmetric (swap left/right and </> comparisons).

3.3 Rank and Select

Rank answers: "How many keys in the BST are strictly less than a given key?"

```
Algorithm: BST Rank

rank(Node x, Key key):
    if x is null:
        return 0
    compare = key.compareTo(x.key)
    if compare < 0:
        return rank(x.left, key)
    else if compare > 0:
        return 1 + size(x.left) + rank(x.right, key)
    else:
        return size(x.left)
```

The reasoning:
- If `key < x.key`: all keys less than `key` are in the left subtree.
- If `key > x.key`: all left-subtree keys (there are `size(x.left)` of them) are less than `key`, plus the node `x` itself, plus however many in the right subtree are less than `key`.
- If `key == x.key`: exactly `size(x.left)` keys are less than `key`.

Select answers: "What key has rank k?" (i.e., what is the k-th smallest key, 0-indexed?)

```
Algorithm: BST Select

select(Node x, int k):
    if x is null:
        return null
    leftSize = size(x.left)
    if k < leftSize:
        return select(x.left, k)
    else if k > leftSize:
        return select(x.right, k - leftSize - 1)
    else:
        return x.key
```

Trace: select(3) in our BST (find the key of rank 3, i.e., the 4th smallest).

Let us first annotate the tree with subtree sizes:

```
              S (7)
             / \
            E (5) X (1)
           / \
          A (2) R (2)
           \    /
            C (1) H (1)
```

```
select(S, 3):  size(S.left) = size(E) = 5.  3 < 5, recurse left.
  select(E, 3):  size(E.left) = size(A) = 2.  3 > 2, recurse right with k = 3 - 2 - 1 = 0.
    select(R, 0):  size(R.left) = size(H) = 1.  0 < 1, recurse left.
      select(H, 0):  size(H.left) = 0.  0 == 0, return "H".

Result: select(3) = "H"
```

Verification: the sorted keys are A, C, E, H, R, S, X. Index 3 (0-based) is indeed "H".

3.4 Range Search and Range Count

A **range search** returns all keys in a given interval [lo, hi]. A **range count** returns how many keys fall in that interval.

Range count is simple using rank:

```
size(lo, hi) = rank(hi) - rank(lo) + (contains(hi) ? 1 : 0)
```

For range search, we perform an in-order traversal but prune branches that cannot contain keys in the range:

```
Algorithm: Range Search

keys(Node x, Queue queue, Key lo, Key hi):
    if x is null:
        return
    compareLo = lo.compareTo(x.key)
    compareHi = hi.compareTo(x.key)
    if compareLo < 0:
        keys(x.left, queue, lo, hi)       // some left-subtree keys might be in range
    if compareLo <= 0 and compareHi >= 0:
        queue.enqueue(x.key)              // this key is in range
    if compareHi > 0:
        keys(x.right, queue, lo, hi)      // some right-subtree keys might be in range
```

3.5 Tree Traversals

A BST can be traversed in several standard orders, each useful for different purposes.

In-Order Traversal (Left, Root, Right)

Visits all keys in **sorted order**. This is the most natural traversal for a BST.

```
Algorithm: In-Order Traversal

inorder(Node x):
    if x is null: return
    inorder(x.left)
    visit(x)
    inorder(x.right)
```

Trace on our BST:

```
             S
            / \
           E   X
          / \
         A   R
          \  /
           C H

In-order visit sequence: A, C, E, H, R, S, X   (sorted!)
```

The call tree:

```
inorder(S)
  inorder(E)
    inorder(A)
      inorder(null) -- return
      visit A
      inorder(C)
        inorder(null) -- return
        visit C
        inorder(null) -- return
    visit E
    inorder(R)
      inorder(H)
        inorder(null) -- return
        visit H
        inorder(null) -- return
      visit R
      inorder(null) -- return
  visit S
  inorder(X)
    inorder(null) -- return
    visit X
    inorder(null) -- return
```

Pre-Order Traversal (Root, Left, Right)

Visits the root before its subtrees. Useful for **serialization** and **copying** a tree.

```
Pre-order on our BST: S, E, A, C, R, H, X
```

If you insert keys in pre-order sequence into an empty BST, you reconstruct the original tree exactly.

Post-Order Traversal (Left, Right, Root)

Visits the root after its subtrees. Useful for **deleting a tree** (free children before parent) and **evaluating expression trees**.

```
Post-order on our BST: C, A, H, R, E, X, S
```

Level-Order Traversal (BFS)

Visits nodes level by level, from top to bottom, left to right within each level. Implemented using a queue.

```
Algorithm: Level-Order Traversal

levelorder(Node root):
    queue = new Queue()
    queue.enqueue(root)
    while queue is not empty:
        x = queue.dequeue()
        visit(x)
        if x.left is not null:  queue.enqueue(x.left)
        if x.right is not null: queue.enqueue(x.right)
```

```
Level-order on our BST:

Level 0:  S
Level 1:  E, X
Level 2:  A, R
Level 3:  C, H

Visit order: S, E, X, A, R, C, H
```

[10-minute break]

Section 4: Deletion in BSTs (~30 minutes)

Deletion is the most complex BST operation. We present two approaches.

4.1 Lazy Deletion

The simplest approach: do not physically remove the node. Instead, mark it as a "tombstone" — a deleted entry that still occupies space in the tree.

```
Lazy delete: set the value to null but leave the key in the tree.

put(key, null)    // marks key as deleted
```

Advantages:
- Extremely simple to implement.
- Does not disturb the tree structure.
- Works well when deletions are rare.

Disadvantages:
- Memory is never reclaimed; the tree only grows.
- Over time, search performance degrades because searches must examine deleted nodes.
- The `size` field becomes inaccurate unless we track tombstones separately.

Lazy deletion is acceptable as a temporary measure, but for a robust implementation we need actual node removal.

4.2 Hibbard Deletion

Thomas Hibbard proposed the standard BST deletion algorithm in 1962. It handles three cases based on the number of children of the node to be deleted.

DEFINITION 4.1 — HIBBARD DELETION. To delete a node x from a BST:

Case 1: x has no children (leaf node). Simply remove x by setting the parent's link to null.

Case 2: x has exactly one child. Replace x with its only child, connecting the child to x's parent.

Case 3: x has two children. Replace x with its **in-order successor** (the smallest key in x's right subtree). This preserves the BST invariant because the successor is greater than everything in x's left subtree and less than everything else in x's right subtree.

```
Algorithm: Hibbard Deletion

delete(Node x, Key key):
    if x is null:
        return null
    compare = key.compareTo(x.key)
    if compare < 0:
        x.left = delete(x.left, key)
    else if compare > 0:
        x.right = delete(x.right, key)
    else:
        // Found the node to delete
        if x.right is null:
            return x.left           // Cases 1 & 2 (no right child)
        if x.left is null:
            return x.right          // Case 2 (no left child)
        // Case 3: two children
        Node t = x                  // save reference to node
        x = min(t.right)            // x now points to the successor
        x.right = deleteMin(t.right)  // remove successor from right subtree
        x.left = t.left             // attach left subtree
    x.size = 1 + size(x.left) + size(x.right)
    return x
```

The helper `deleteMin` removes the minimum node from a subtree:

```
deleteMin(Node x):
    if x.left is null:
        return x.right              // this node is the min; replace with right child
    x.left = deleteMin(x.left)
    x.size = 1 + size(x.left) + size(x.right)
    return x
```

4.3 Deletion Traces

Let us trace each case on our example BST:

```
Starting BST:
              S (7)
             / \
            E (5) X (1)
           / \
          A (2) R (2)
           \    /
            C (1) H (1)
```

Case 1: Delete "X" (leaf node)

```
delete(S, "X"): "X" > "S", recurse right
  delete(X, "X"): found! X has no children (X.left=null, X.right=null).
    X.right is null, return X.left (which is null).
  S.right = null

Result:
              S (6)
             /
            E (5)
           / \
          A (2) R (2)
           \    /
            C (1) H (1)
```

Case 2: Delete "A" (one child — right child C)

Starting from the original tree:

```
delete(S, "A"): "A" < "S", recurse left
  delete(E, "A"): "A" < "E", recurse left
    delete(A, "A"): found! A.right = C, A.left = null.
      A.right is not null and A.left is null: return A.right (which is C).
    E.left = C

Result:
              S (6)
             / \
            E (4) X (1)
           / \
          C (1) R (2)
                /
               H (1)
```

Case 3: Delete "E" (two children)

Starting from the original tree:

```
delete(S, "E"): "E" < "S", recurse left
  delete(E, "E"): found! E has two children (left=A, right=R).
    Step 1: Find successor = min(E.right) = min(R) -> follows left to H.
            Successor is H.
    Step 2: Remove H from right subtree: deleteMin(R).
            deleteMin(R): R.left = H.
              deleteMin(H): H.left is null, return H.right (null).
            R.left = null. R.size updated to 1.
            Return R.
    Step 3: New node H takes E's place.
            H.right = R (the right subtree minus H)
            H.left = A (E's original left subtree)

Result:
              S (6)
             / \
            H (4) X (1)
           / \
          A (2) R (1)
           \
            C (1)
```

Let us verify the BST invariant: at H, left subtree {A, C} < H and right subtree {R} > H. At S, left subtree {A, C, H, R} < S and right subtree {X} > S. Valid!

4.4 Problems with Hibbard Deletion

Hibbard deletion has a subtle but serious flaw: it introduces **asymmetry** that degrades performance over time.

PROPOSITION 4.1. *After a sequence of random insertions and Hibbard deletions, the expected height of the resulting BST is proportional to sqrt(N), not log N.*

This is because the deletion algorithm always replaces with the in-order successor (from the right subtree), never the predecessor (from the left subtree). This systematic bias makes the left subtrees tend to be taller than the right subtrees over many operations.

Consider this experiment: start with a random BST of N nodes, then perform N^2 random insert/delete pairs (inserting a random key, then deleting a random key, keeping the tree size at approximately N). After this process, the height grows to be proportional to sqrt(N) rather than the log N we would expect for a random BST.

```
Performance degradation with Hibbard deletion:

  N          Expected height     Expected height
             (random BST)        (after many Hibbard deletions)
  -------    ----------------    --------------------------------
  100        ~20                 ~10  (still OK)
  10,000     ~40                 ~100
  1,000,000  ~60                 ~1,000
```

This is one of the great unsolved problems in the analysis of algorithms: **no one has found a simple, efficient, symmetric deletion algorithm for BSTs that preserves logarithmic height.** This problem has been open for over 60 years.

The practical solution is to use a **balanced BST** (such as a red-black tree or AVL tree, which we will study in Lecture 7), which guarantees logarithmic height regardless of the operation sequence.

Section 5: BST Analysis (~20 minutes)

5.1 Summary of Costs

THEOREM 5.1. *In a BST, the costs of search, insert, floor, ceiling, rank, select, delete, min, and max are all proportional to the height of the tree.*

This gives us the following cost table:

Operation	Best case (balanced)	Worst case (skewed)	Average (random)
Search	lg N	N	~1.39 lg N
Insert	lg N	N	~1.39 lg N
Delete	lg N	N	~sqrt(N)*
Min/Max	lg N	N	lg N
Floor/Ceiling	lg N	N	~1.39 lg N
Rank/Select	lg N	N	~1.39 lg N
Range search	lg N + R	N + R	~1.39 lg N + R

*After many Hibbard deletions (see Section 4.4). R = number of keys in range.

5.2 Relationship to Quicksort

The connection between BSTs and quicksort is deep and illuminating.

PROPOSITION 5.1. *Building a BST from N randomly ordered keys uses exactly the same comparisons as quicksort would use to sort those keys (with the first element as pivot at each stage).*

Consider the keys 5, 2, 8, 1, 3, 7, 9 inserted into an empty BST:

```
BST construction:                 Quicksort partition:

Insert 5 (root):                  Pivot = 5:
   5                              [2,1,3]  5  [8,7,9]

Insert 2 (compare with 5):       Left partition, pivot = 2:
   5                              [1]  2  [3]
  /
 2

Insert 8 (compare with 5):       Right partition, pivot = 8:
   5                              [7]  8  [9]
  / \
 2   8
```

Each node in the BST corresponds to a pivot in quicksort. The left subtree contains elements that went to the left partition, and the right subtree contains elements that went to the right partition. The comparisons are identical!

This correspondence means that every result about random BSTs translates to a result about quicksort, and vice versa.

5.3 Comprehensive Comparison

Let us place BSTs in context by comparing them against all the symbol table implementations we have seen:

Implementation	Search (worst)	Insert (worst)	Delete (worst)	Search (avg)	Insert (avg)	Ordered ops	Space
Unordered list	N	N	N	N/2	N	No	N
Ordered array	lg N	N	N	lg N	N/2	Yes	N
BST	N	N	N	1.39 lg N	1.39 lg N	Yes	N
Balanced BST*	lg N	lg N	lg N	lg N	lg N	Yes	N

*Red-black trees, AVL trees — covered in Lecture 7.

The BST improves dramatically over ordered arrays for insertion (from linear to logarithmic on average) while maintaining logarithmic search (even improving the constant factor: 1.39 lg N vs lg N for ordered arrays, where the lg N for ordered arrays refers to binary search which also makes about lg N compares in the average case). The weakness is the N worst case, which balanced BSTs eliminate.

5.4 Why Random BSTs Work Well in Practice

Even though the worst case is linear, BSTs perform well in many practical applications:

1. **Random or pseudo-random insertion order:** If keys arrive in a random order (or are hashed before use), the resulting BST has expected height ~4.311 ln N and expected search cost ~1.39 lg N.
2. **Self-organizing property:** Frequently accessed keys tend to be near the root of the search path, giving good cache behavior.
3. **Simple implementation:** BSTs are straightforward to code and debug compared to balanced trees.
4. **In-place operations:** No auxiliary arrays or complex restructuring needed (unlike balanced trees).

However, the worst case is a real concern. Sorted or nearly-sorted input is common in practice (reading from a sorted file, inserting timestamps in chronological order), and this produces the degenerate case. For applications where worst-case guarantees matter, balanced BSTs are essential.

[10-minute break]

Section 6: Tree Traversal Algorithms in Depth (~15 minutes)

6.1 Iterative In-Order Traversal Using an Explicit Stack

The recursive in-order traversal is elegant but uses O(h) space on the call stack, where h is the tree height. We can make this explicit:

```
Algorithm: Iterative In-Order Traversal

iterativeInorder(Node root):
    stack = new Stack()
    current = root
    while current is not null OR stack is not empty:
        // Go as far left as possible
        while current is not null:
            stack.push(current)
            current = current.left
        // Process the node
        current = stack.pop()
        visit(current)
        // Move to right subtree
        current = current.right
```

Trace on our BST:

```
              S
             / \
            E   X
           / \
          A   R
           \  /
            C H

Step  | current | Stack (top->bottom) | Action        | Output
------|---------|---------------------|---------------|-------
  1   |    S    |                     | push S, go L  |
  2   |    E    | S                   | push E, go L  |
  3   |    A    | E,S                 | push A, go L  |
  4   |  null   | A,E,S              | pop A, visit  | A
  5   |    C    | E,S                 | push C, go L  |
  6   |  null   | C,E,S              | pop C, visit  | C
  7   |  null   | E,S                 | pop E, visit  | E
  8   |    R    | S                   | push R, go L  |
  9   |    H    | R,S                 | push H, go L  |
 10   |  null   | H,R,S              | pop H, visit  | H
 11   |  null   | R,S                 | pop R, visit  | R
 12   |  null   | S                   | pop S, visit  | S
 13   |    X    |                     | push X, go L  |
 14   |  null   | X                   | pop X, visit  | X
 15   |  null   |                     | done          |

Output: A, C, E, H, R, S, X  (sorted!)
```

6.2 Morris Traversal (Constant Space)

Morris traversal achieves in-order traversal using **O(1) extra space** (no stack, no recursion) by temporarily modifying the tree. It uses **threaded binary tree** ideas.

The key insight: before moving to the left subtree of a node, create a temporary link (thread) from the rightmost node of the left subtree back to the current node. This thread allows us to return to the parent without a stack.

```
Algorithm: Morris In-Order Traversal

morrisInorder(Node root):
    current = root
    while current is not null:
        if current.left is null:
            visit(current)
            current = current.right     // follow right link (or thread)
        else:
            // Find in-order predecessor
            predecessor = current.left
            while predecessor.right is not null AND predecessor.right != current:
                predecessor = predecessor.right

            if predecessor.right is null:
                // Create thread
                predecessor.right = current
                current = current.left
            else:
                // Thread exists -- we have returned; remove thread
                predecessor.right = null
                visit(current)
                current = current.right
```

Morris traversal is clever but modifies the tree temporarily (and restores it). It is used in situations where stack space is extremely constrained, such as embedded systems. The trade-off: it is harder to understand and debug, and the tree must not be accessed by other threads during traversal.

6.3 Applications of Each Traversal Order

Traversal	Order	Primary Applications
In-order	Left, Root, Right	Sorted output; BST validation (check keys are in order); range queries
Pre-order	Root, Left, Right	Tree serialization (can reconstruct BST from pre-order); tree copying; prefix expression evaluation
Post-order	Left, Right, Root	Safe tree deletion (children before parent); postfix expression evaluation; computing subtree properties (size, height) bottom-up
Level-order	Level by level (BFS)	Finding shortest path to a node; printing tree level by level; serialization for complete/near-complete trees

BST Validation using In-Order Traversal:

A binary tree is a valid BST if and only if its in-order traversal produces keys in strictly increasing order. This gives us a simple O(N) validation algorithm:

```
Algorithm: Validate BST

isValidBST(Node x):
    return validate(x, null, null)

validate(Node x, Key min, Key max):
    if x is null: return true
    if min is not null AND x.key <= min: return false
    if max is not null AND x.key >= max: return false
    return validate(x.left, min, x.key) AND validate(x.right, x.key, max)
```

Section 7: Worked Examples and Practice Problems (~30 minutes)

Problem 1: Construct a BST from Insertion Sequence

Problem: Draw the BST that results from inserting the following keys (in order) into an initially empty BST: `P, Q, E, B, G, F, M, N`

Solution:

```
Insert P:       P

Insert Q:       P
                 \
                  Q

Insert E:       P
               / \
              E   Q

Insert B:       P
               / \
              E   Q
             /
            B

Insert G:       P
               / \
              E   Q
             / \
            B   G

Insert F:       P
               / \
              E   Q
             / \
            B   G
               /
              F

Insert M:       P
               / \
              E   Q
             / \
            B   G
               / \
              F   M

Insert N:       P
```

```
                / \
               E   Q
              / \
             B   G
                / \
               F   M
                    \
                     N
```

The final BST has height 4. In-order traversal: B, E, F, G, M, N, P, Q.

Problem 2: Trace floor("D") and ceiling("D")

Using the BST from Problem 1.

floor("D"):

```
floor(P, "D"): "D" < "P", go left
  floor(E, "D"): "D" < "E", go left
    floor(B, "D"): "D" > "B", B is candidate. Check right.
      floor(null, "D"): return null
    t = null, return B
  return B
return B

Answer: floor("D") = "B"
```

ceiling("D"):

```
ceiling(P, "D"): "D" < "P", P is candidate. Check left.
  ceiling(E, "D"): "D" < "E", E is candidate. Check left.
    ceiling(B, "D"): "D" > "B", go right.
      ceiling(null, "D"): return null
    return null
  t = null, return E
t = E (not null), return E

Answer: ceiling("D") = "E"
```

Verification: sorted keys are B, E, F, G, M, N, P, Q. The largest key <= "D" is "B", and the smallest key >= "D" is "E". Correct!

Problem 3: Compute rank and select

Using the BST from Problem 1 (annotated with sizes):

```
              P (8)
             / \
          E (6)  Q (1)
         / \
      B (1)  G (4)
             / \
          F (1)  M (2)
                   \
                    N (1)
```

(a) What is rank("G")?

```
rank(P, "G"): "G" < "P", recurse left
  rank(E, "G"): "G" > "E", return 1 + size(E.left) + rank(E.right, "G")
                = 1 + size(B) + rank(G, "G")
                = 1 + 1 + rank(G, "G")
    rank(G, "G"): "G" == "G", return size(G.left) = size(F) = 1
  = 1 + 1 + 1 = 3
= 3

Answer: rank("G") = 3
```

Verification: keys less than "G" are {B, E, F} -- exactly 3. Correct!

(b) What is select(5)?

```
select(P, 5):  size(P.left) = size(E) = 6.  5 < 6, recurse left.
  select(E, 5):  size(E.left) = size(B) = 1.  5 > 1, recurse right with k = 5 - 1 - 1 = 3.
    select(G, 3):  size(G.left) = size(F) = 1.  3 > 1, recurse right with k = 3 - 1 - 1 = 1.
      select(M, 1):  size(M.left) = 0.  1 > 0, recurse right with k = 1 - 0 - 1 = 0.
        select(N, 0):  size(N.left) = 0.  0 == 0, return "N".

Answer: select(5) = "N"
```

Verification: sorted keys are B(0), E(1), F(2), G(3), M(4), N(5), P(6), Q(7). Key at rank 5 is "N". Correct!

Problem 4: Delete "E" from the BST in Problem 1

Solution: Node "E" has two children (B and G). Apply Hibbard deletion Case 3.

Step 1: Find successor of E = min(E.right) = min(G) = follow left links from G: G -> F. Successor is **F**.

Step 2: Delete F from E's right subtree. F is a leaf (no children), so just remove it.

```
After removing F from G's left:

        G (3)
         \
          M (2)
           \
            N (1)
```

Step 3: Replace E with F. F takes E's position with E's left subtree (B) and the modified right subtree.

```
Final BST after deleting E:

              P (7)
             / \
          F (5)  Q (1)
         / \
      B (1)  G (3)
               \
                M (2)
                 \
                  N (1)
```

Verification of BST invariant:
- At F: left {B} < F < {G, M, N} right. Valid.
- At P: left {B, F, G, M, N} < P < {Q} right. Valid.
- At G: left empty, right {M, N} > G. Valid.

Problem 5: Different Insertion Orders, Same Keys

Problem: Give two different insertion orders of the keys {1, 2, 3, 4, 5} that produce (a) a balanced BST and (b) a maximally skewed (worst-case) BST.

Solution:

(a) Balanced BST — insert median first, then medians of halves:

Insertion order: **3, 1, 5, 2, 4** (or **3, 2, 4, 1, 5**, etc.)

```
              3
             / \
            1   5
             \ /
             2 4

Height = 2
```

(b) Maximally skewed — insert in sorted order:

Insertion order: **1, 2, 3, 4, 5**

```
          1
           \
            2
             \
              3
               \
                4
                 \
                  5

Height = 4  (degenerate: linked list)
```

Or in reverse sorted order: 5, 4, 3, 2, 1 (left-skewed).

Problem 6: In-Order Predecessor and Successor

Problem: In the BST from Problem 1, what are the in-order predecessor and successor of "G"?

Solution:

The in-order traversal is: B, E, **F**, **G**, **M**, N, P, Q.

- **Predecessor of G** = F (the largest key in G's left subtree; follow left, then right as far as possible: G -> F, and F has no right child, so F is the predecessor).
- **Successor of G** = M (the smallest key in G's right subtree; follow right, then left as far as possible: G -> M, and M has no left child, so M is the successor).

Problem 7: BST Property Verification

Problem: Is the following binary tree a valid BST?

```
    10
   /  \
  5    15
 / \   / \
3   8 12  20
   /    \
  7     13
```

Solution: Perform an in-order traversal and check that keys are in strictly increasing order.

In-order: 3, 5, 7, 8, 10, 12, 13, 15, 20

Each key is strictly greater than the previous: 3 < 5 < 7 < 8 < 10 < 12 < 13 < 15 < 20. **Yes, this is a valid BST.**

Alternatively, verify at each node that it respects the min/max constraints from its ancestors:
- Node 10: no constraints. Valid.
- Node 5: must be < 10. Valid.
- Node 15: must be > 10. Valid.
- Node 3: must be < 5. Valid.
- Node 8: must be > 5 and < 10. Valid.
- Node 7: must be > 5 and < 8. Valid. (7 is in 8's left subtree, and 8 is in 5's right subtree, which is in 10's left subtree.)
- Node 12: must be > 10 and < 15. Valid.

- Node 13: must be > 12 and < 15. Valid.
- Node 20: must be > 15. Valid.

Problem 8: Sequence of Operations

Problem: Starting from an empty BST, perform the following operations and show the tree after each:

1. put(5, "A")
2. put(3, "B")
3. put(7, "C")
4. put(1, "D")
5. put(4, "E")
6. delete(3)
7. put(6, "F")
8. delete(5)

Solution:

After put(5,"A"), put(3,"B"), put(7,"C"), put(1,"D"), put(4,"E"):

```
        5
       / \
      3   7
     / \
    1   4
```

Step 6: delete(3). Node 3 has two children (1 and 4).
- Successor of 3 = min(3.right) = 4 (4 has no left child).
- Delete 4 from 3's right subtree (it is a leaf, so just remove).
- Replace 3 with 4.

```
        5
       / \
      4   7
     /
    1
```

Step 7: put(6, "F"). Compare with 5: 6 > 5, go right. Compare with 7: 6 < 7, go left. 7.left is null, insert here.

```
        5
       / \
      4   7
     /   /
    1   6
```

Step 8: delete(5). Node 5 has two children (4 and 7).
- Successor of 5 = min(5.right) = min(7) = 6 (follow left: 7 -> 6, and 6 has no left child).
- Delete 6 from 5's right subtree.
- deleteMin(7): 7.left = 6. deleteMin(6): 6.left is null, return 6.right (null). So 7.left = null.
- Right subtree becomes just 7.
- Replace 5 with 6. 6.left = 4-subtree, 6.right = 7.

```
        6
       / \
      4   7
     /
    1
```

Verification: in-order traversal gives 1, 4, 6, 7. Valid BST.

Summary

Topic	Key Takeaway
Symbol tables	Key-value abstraction; ordered tables support floor, ceiling, rank, select
BST definition	Each node's key is greater than all left-subtree keys, less than all right-subtree keys
BST shape	Determined by insertion order; random order gives good balance
Search/Insert cost	Proportional to tree height; ~1.39 lg N average for random trees
BST height	Best: lg N, Worst: N, Average (random): ~4.311 ln N
Hibbard deletion	Replace with in-order successor; causes sqrt(N) height degradation over time
Traversals	In-order (sorted), pre-order (serialization), post-order (cleanup), level-order (BFS)
BST vs. quicksort	Building a BST uses the same comparisons as quicksort partitioning
Practical verdict	BSTs are simple and efficient on average, but need balancing for worst-case guarantees

Looking ahead: In Lecture 7, we will study **balanced search trees** (2-3 trees, left-leaning red-black trees, and AVL trees) that guarantee O(lg N) height regardless of the insertion and deletion order, eliminating the worst-case linear behavior of standard BSTs.

References

1. **Sedgewick, R. & Wayne, K.** *Algorithms*, 4th Edition. Addison-Wesley, 2011. Chapters 3.1 (Symbol Tables) and 3.2 (Binary Search Trees).

2. **Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C.** *Introduction to Algorithms*, 4th Edition. MIT Press, 2022. Chapter 12 (Binary Search Trees).

3. **Hibbard, T.N.** "Some Combinatorial Properties of Certain Trees with Applications to Searching and Sorting." *Journal of the ACM*, 9(1):13-28, 1962. (Original paper on BST deletion.)

4. **Reed, B.** "The Height of a Random Binary Search Tree." *Journal of the ACM*, 50(3):306-332, 2003. (Proof that expected height is ~4.311 ln N.)

5. **Knuth, D.E.** *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Edition. Addison-Wesley, 1998. Section 6.2.2 (Binary Tree Searching).

6. **Sedgewick, R. & Flajolet, P.** *An Introduction to the Analysis of Algorithms*, 2nd Edition. Addison-Wesley, 2013. Chapter 6 (Trees). (Rigorous analysis of random BST properties.)

7. **Morris, J.H.** "Traversing Binary Trees Simply and Cheaply." *Information Processing Letters*, 9(5):197-200, 1979. (Original Morris traversal paper.)

8. **Booksite:** https://algs4.cs.princeton.edu/32bst/ — Sedgewick & Wayne's companion website with Java implementations, visualizations, and exercises.