# Algorithms

Review for Final

# Final Exam

- Coverage: the whole semester
- Goal: doable in 2 hours
- Cheat sheet: you are allowed *two* 8’11” sheets, both sides

# Final Exam: Study Tips

- Study tips:
  - Study each lecture
  - Study the homework and homework solutions
  - Study the midterm exams
- Re-make your previous cheat sheets
  - I recommend handwriting or typing them
  - Think about what you should have had on it the first time…cheat sheets is about *identifying* important concepts

# Graph Representation

- Adjacency list
- Adjacency matrix
- Tradeoffs:
  - *What makes a graph dense?*
  - *What makes a graph sparse?*
  - *What about planar graphs?*

# Basic Graph Algorithms

- Breadth-first search
  - *What can we use BFS to calculate?*
  - A: shortest-path distance to source vertex
- Depth-first search
  - Tree edges, back edges, cross and forward edges
  - *What can we use DFS for?*
  - A: finding cycles, topological sort

# Topological Sort, MST

- Topological sort
  - Examples: getting dressed, project dependency
  - *To what kind of graph does topological sort apply?*
- Minimum spanning tree
  - Optimal substructure
  - Min edge theorem (enables greedy approach)

# MST Algorithms

- Prim’s algorithm
  - *What is the bottleneck in Prim’s algorithm?*
  - A: priority queue operations
- Kruskal’s algorithm
  - *What is the bottleneck in Kruskal’s algorithm?*
  - Answer: depends on disjoint-set implementation
    - As covered in class, disjoint-set union operations
    - As described in book, sorting the edges

# Single-Source Shortest Path

- Optimal substructure
- Key idea: relaxation of edges
- *What does the Bellman-Ford algorithm do?*
  - *What is the running time?*
- *What does Dijkstra's algorithm do?*
  - *What is the running time?*
  - *When does Dijkstra's algorithm not apply?*

# Disjoint-Set Union

- We talked about representing sets as linked lists, every element stores pointer to list head
- *What is the cost of merging sets A and B?*
  - A: O(max(|A|, |B|))
- *What is the maximum cost of merging n 1-element sets into a single n-element set?*
  - A: $O(n^2)$
- *How did we improve this? By how much?*
  - A: always copy smaller into larger: $O(n \lg n)$

# Amortized Analysis

- Idea: worst-case cost of an operation may overestimate its cost over course of algorithm
- Goal: get a tighter *amortized bound* on its cost
  - Aggregate method: total cost of operation over course of algorithm divided by # operations
    - Example: disjoint-set union
  - Accounting method: “charge” a cost to each operation, accumulate unused cost in bank, never go negative
    - Example: dynamically-doubling arrays

# Dynamic Programming

- Indications: optimal substructure, repeated subproblems
- *What is the difference between memoization and dynamic programming?*
- A: same basic idea, but:
  - *Memoization*: recursive algorithm, looking up subproblem solutions after computing once
  - *Dynamic programming*: build table of subproblem solutions bottom-up

# LCS Via Dynamic Programming

- *Longest common subsequence* (*LCS*) problem:
  - Given two sequences x[1..m] and y[1..n], find the longest subsequence which occurs in both
- Brute-force algorithm: $2^m$ subsequences of x to check against $n$ elements of y: $O(n\,2^m)$
- Define c[i,j] = length of LCS of x[1..i], y[1..j]
- Theorem:

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

# Greedy Algorithms

- Indicators:
  - Optimal substructure
  - *Greedy choice property*: a locally optimal choice leads to a globally optimal solution
- Example problems:
  - Activity selection: Set of activities, with start and end times. Maximize compatible set of activities.
  - Fractional knapsack: sort items by $/lb, then take items in sorted order
  - MST

# NP-Completeness

- *What do we mean when we say a problem is in* ***P****?*
  - A: A solution can be found in polynomial time
- *What do we mean when we say a problem is in* ***NP****?*
  - A: A solution can be verified in polynomial time
- *What is the relation between* ***P*** *and* ***NP****?*
  - A: $\mathbf{P} \subseteq \mathbf{NP}$, but no one knows whether $\mathbf{P} = \mathbf{NP}$

# Review: NP-Complete

- *What, intuitively, does it mean if we can reduce problem P to problem Q?*
  - P is "no harder than" Q
- *How do we reduce P to Q?*
  - Transform instances of P to instances of Q in polynomial time s.t. Q: "yes" iff P: "yes"
- *What does it mean if Q is NP-Hard?*
  - Every problem $P \in \mathbf{NP} \leq_p Q$
- *What does it mean if Q is NP-Complete?*
  - Q is NP-Hard and $Q \in \mathbf{NP}$

# Review: Proving Problems NP-Complete

- *What was the first problem shown to be NP-Complete?*
- A: Boolean satisfiability (*SAT*), by Cook
- *How do we usually prove that a problem R is NP-Complete?*
- A: Show R $\in$ **NP**, and reduce a known NP-Complete problem Q to R

# Review: Reductions

- Review the reductions we've covered:
  - Directed hamiltonian cycle → undirected hamiltonian cycle
  - Undirected hamiltonian cycle → traveling salesman problem
  - 3-CNF → *k*-clique
  - *k*-clique → vertex cover
  - Homework 7

# Next: Detailed Review

- Up next: a detailed review of the first half of the course
  - The following 100+ slides are intended as a resource for your studying
  - Since you probably remember the more recent stuff better, I just provide this for the early material

# Review: Induction

- Suppose
  - S(k) is true for fixed constant k
    - Often k = 0
  - S(n) $\Rightarrow$ S(n+1) for all n >= k
- Then S(n) is true for all n >= k

# Proof By Induction

- Claim:S(n) is true for all n >= k
- Basis:
  - Show formula is true when n = k
- Inductive hypothesis:
  - Assume formula is true for an arbitrary n
- Step:
  - Show that formula is then true for n+1

# Induction Example: Gaussian Closed Form

- Prove $1 + 2 + 3 + \ldots + n = n(n+1) / 2$
  - Basis:
    - If $n = 0$, then $0 = 0(0+1) / 2$
  - Inductive hypothesis:
    - Assume $1 + 2 + 3 + \ldots + n = n(n+1) / 2$
  - Step (show true for n+1):

$$1 + 2 + \ldots + n + n+1 = (1 + 2 + \ldots + n) + (n+1)$$

$$= n(n+1)/2 + n+1 = [n(n+1) + 2(n+2)]/2$$

$$= (n+1)(n+2)/2 = (n+1)(n+1 + 1) / 2$$

# Induction Example: Geometric Closed Form

- Prove $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for all a != 1
  - Basis: show that $a^0 = (a^{0+1} - 1)/(a - 1)$

    $a^0 = 1 = (a^1 - 1)/(a - 1)$
  - Inductive hypothesis:
    - Assume $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$
  - Step (show true for n+1):

    $a^0 + a^1 + \ldots + a^{n+1} = a^0 + a^1 + \ldots + a^n + a^{n+1}$

    $= (a^{n+1} - 1)/(a - 1) + a^{n+1} = (a^{n+1+1} - 1)(a - 1)$

# Review: Analyzing Algorithms

- We are interested in *asymptotic analysis*:
  - Behavior of algorithms as problem size gets large
  - Constants, low-order terms don't matter

# An Example: Insertion Sort

| 30 | 10 | 40 | 20 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

$i = \varnothing \quad j = \varnothing \quad key = \varnothing$

$A[j] = \varnothing \qquad A[j+1] = \varnothing$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| | | | |
|---|---|---|---|
| *30* | *10* | *40* | *20* |
| 1 | 2 | 3 | 4 |

i = 2  j = 1  key = 10

A[j] = 30  A[j+1] = 10

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 30 | 30 | 40 | 20 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

i = 2    j = 1    key = 10
A[j] = 30    A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| | | | |
|---|---|---|---|
| *30* | *30* | *40* | *20* |
| 1 | 2 | 3 | 4 |

i = 2    j = 1    key = 10
A[j] = 30    A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 30 | 30 | 40 | 20 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

$i = 2 \quad j = 0 \quad key = 10$

$A[j] = \varnothing \quad A[j+1] = 30$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 30 | 30 | 40 | 20 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

$i = 2$ $\quad j = 0$ $\quad key = 10$

$A[j] = \varnothing$ $\quad A[j+1] = 30$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 10 | 30 | 40 | 20 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

i = 2    j = 0    key = 10

A[j] = ∅    A[j+1] = 10

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 10 | 30 | 40 | 20 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

$i = 3 \quad j = 0 \quad key = 10$

$A[j] = \varnothing \quad A[j+1] = 10$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 10 | 30 | 40 | 20 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

$i = 3$ $j = 0$ $key = 40$

$A[j] = \varnothing$ $A[j+1] = 10$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| | | | |
|---|---|---|---|
| *10* | *30* | *40* | *20* |
| 1 | 2 | 3 | 4 |

i = 3    j = 0    key = 40
A[j] = ∅    A[j+1] = 10

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 10 | 30 | 40 | 20 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

i = 3   j = 2   key = 40
A[j] = 30   A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 10 | 30 | 40 | 20 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

i = 3    j = 2    key = 40

A[j] = 30    A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| | | | |
|---|---|---|---|
| *10* | *30* | *40* | *20* |
| 1 | 2 | 3 | 4 |

$i = 3 \quad j = 2 \quad key = 40$

$A[j] = 30 \quad A[j+1] = 40$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 10 | 30 | 40 | 20 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

i = 4  j = 2  key = 40
A[j] = 30  A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 10 | 30 | 40 | 20 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

i = 4     j = 2     key = 20
A[j] = 30     A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

# An Example: Insertion Sort

| 10 | 30 | 40 | 20 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

$i = 4$ $j = 2$ $key = 20$

$A[j] = 30$ $A[j+1] = 40$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

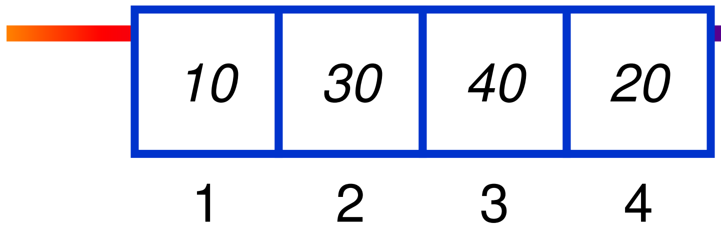
# An Example: Insertion Sort

| 10 | 30 | 40 | 20 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

i = 4   j = 3   key = 20
A[j] = 40   A[j+1] = 20

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
             A[j+1] = A[j]
             j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort




i = 4     j = 3     key = 20
A[j] = 40     A[j+1] = 20

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 10 | 30 | 40 | 40 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 10 | 30 | 40 | 40 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

$i = 4 \quad j = 3 \quad key = 20$

$A[j] = 40 \quad A[j+1] = 40$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 10 | 30 | 40 | 40 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 10 | 30 | 40 | 40 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

i = 4   j = 2   key = 20
A[j] = 30   A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 10 | 30 | 40 | 40 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

i = 4  j = 2  key = 20
A[j] = 30  A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 10 | 30 | 30 | 40 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

$i = 4 \quad j = 2 \quad key = 20$

$A[j] = 30 \quad A[j+1] = 30$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 10 | 30 | 30 | 40 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

i = 4  j = 2  key = 20
A[j] = 30  A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 10 | 30 | 30 | 40 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

i = 4    j = 1    key = 20
A[j] = 10    A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| | | | |
|---|---|---|---|
| *10* | *30* | *30* | *40* |
| 1 | 2 | 3 | 4 |

i = 4  j = 1  key = 20
A[j] = 10  A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 10 | 20 | 30 | 40 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

i = 4    j = 1    key = 20
A[j] = 10    A[j+1] = 20

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

# An Example: Insertion Sort

| 10 | 20 | 30 | 40 |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

$i = 4 \quad j = 1 \quad key = 20$

$A[j] = 10 \quad A[j+1] = 20$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

*Done!*

# Insertion Sort

| Statement | Effort |
|---|---|
| `InsertionSort(A, n) {` | |
| `  for i = 2 to n {` | $c_1 n$ |
| `    key = A[i]` | $c_2(n-1)$ |
| `    j = i - 1;` | $c_3(n-1)$ |
| `    while (j > 0) and (A[j] > key) {` | $c_4 T$ |
| `      A[j+1] = A[j]` | $c_5(T-(n-1))$ |
| `      j = j - 1` | $c_6(T-(n-1))$ |
| `    }` | 0 |
| `    A[j+1] = key` | $c_7(n-1)$ |
| `  }` | 0 |
| `}` | |

$T = t_2 + t_3 + \ldots + t_n$ where $t_i$ is number of while expression evaluations for the $i^{th}$ for loop iteration

# Analyzing Insertion Sort

- $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 T + c_5(T - (n-1)) + c_6(T - (n-1)) + c_7(n-1)$

  $= c_8 T + c_9 n + c_{10}$
- What can T be?
  - Best case -- inner loop body never executed
    - $t_i = 1 \Rightarrow T(n)$ is a linear function
  - Worst case -- inner loop body executed for all previous elements
    - $t_i = i \Rightarrow T(n)$ is a quadratic function
  - *If T is a quadratic function, which terms in the above equation matter?*

# Upper Bound Notation

- We say InsertionSort's run time is $O(n^2)$
  - Properly we should say run time is *in* $O(n^2)$
  - Read O as "Big-O" (you'll also hear it as "order")
- In general a function
  - f(n) is O(g(n)) if there exist positive constants $c$ and $n_0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$
- Formally
  - $O(g(n)) = \{ f(n): \exists$ positive constants $c$ and $n_0$ such that $f(n) \leq c \cdot g(n) \; \forall \; n \geq n_0$

# Big O Fact

- A polynomial of degree k is $O(n^k)$
- Proof:
  - Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$
    - Let $a_i = | b_i |$
  - $f(n) \leq a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$

$$\leq \quad n^k \sum a_i \frac{n^i}{n^k} \quad \leq \quad n^k \sum a_i \quad \leq \quad cn^k$$

# Lower Bound Notation

- We say InsertionSort's run time is $\Omega(n)$
- In general a function
  - $f(n)$ is $\Omega(g(n))$ if $\exists$ positive constants $c$ and $n_0$ such that $0 \leq c \cdot g(n) \leq f(n) \;\; \forall\, n \geq n_0$

# Asymptotic Tight Bound

- A function f(n) is $\Theta(g(n))$ if $\exists$ positive constants $c_1$, $c_2$, and $n_0$ such that

$$c_1\, g(n) \le f(n) \le c_2\, g(n)\ \forall\ n \ge n_0$$

# Other Asymptotic Notations

- A function f(n) is o(g(n)) if $\exists$ positive constants $c$ and $n_0$ such that

  $$f(n) < c\ g(n)\ \forall\ n \geq n_0$$

- A function f(n) is ω(g(n)) if $\exists$ positive constants $c$ and $n_0$ such that

  $$c\ g(n) < f(n)\ \forall\ n \geq n_0$$

- Intuitively,
  - o() is like <
  - ω() is like >
  - Θ() is like =
  - O() is like ≤
  - Ω() is like ≥

# Review: Recurrences

- Recurrence: an equation that describes a function in terms of its value on smaller functions

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

# Review: Solving Recurrences

- Substitution method
- Iteration method
- Master method

# Review: Substitution Method

- Substitution Method:
  - Guess the form of the answer, then use induction to find the constants and show that solution works
  - Example:
    - $T(n) = 2T(n/2) + \Theta(n) \rightarrow T(n) = \Theta(n \lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor) + n \rightarrow$ ???

# Review: Substitution Method

- Substitution Method:
  - Guess the form of the answer, then use induction to find the constants and show that solution works
  - Examples:
    - $T(n) = 2T(n/2) + \Theta(n) \rightarrow T(n) = \Theta(n \lg n)$
    - $T(n) = 2T(\lfloor n/2 \rfloor) + n \rightarrow T(n) = \Theta(n \lg n)$
  - We can show that this holds by induction

# Substitution Method

- Our goal: show that
  $T(n) = 2T(\lfloor n/2 \rfloor) + n = O(n \lg n)$
- Thus, we need to show that $T(n) \leq c\, n \lg n$ with an appropriate choice of $c$
  - Inductive hypothesis: assume
    $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor$
  - Substitute back into recurrence to show that
    $T(n) \leq c\, n \lg n$ follows, when $c \geq 1$
    (show on board)

# Review: Iteration Method

- Iteration method:
  - Expand the recurrence $k$ times
  - Work some algebra to express as a summation
  - Evaluate the summation

# Review: $s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$

- s(n) =

  c + s(n-1)

  c + c + s(n-2)

  2c + s(n-2)

  2c + c + s(n-3)

  3c + s(n-3)

  …

  kc + s(n-k) = ck + s(n-k)

# Review: $s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$

- So far for n >= k we have
  - s(n) = ck + s(n-k)
- What if k = n?
  - s(n) = cn + s(0) = cn

# Review: $T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$

- T(n) =

  2T(n/2) + c

  2(2T(n/2/2) + c) + c

  $2^2T(n/2^2) + 2c + c$

  $2^2(2T(n/2^2/2) + c) + 3c$

  $2^3T(n/2^3) + 4c + 3c$

  $2^3T(n/2^3) + 7c$

  $2^3(2T(n/2^3/2) + c) + 7c$

  $2^4T(n/2^4) + 15c$

  …

  $2^kT(n/2^k) + (2^k - 1)c$

# Review: $T(n) = \begin{cases} c & n = 1 \\ 2T\left(\dfrac{n}{2}\right) + c & n > 1 \end{cases}$

- So far for n > 2k we have
  - $T(n) = 2^k T(n/2^k) + (2^k - 1)c$
- What if k = lg n?
  - $T(n) = 2^{\lg n}\, T(n/2^{\lg n}) + (2^{\lg n} - 1)c$

    $= n\, T(n/n) + (n - 1)c$

    $= n\, T(1) + (n-1)c$

    $= nc + (n-1)c = (2n - 1)c$

# Review: The Master Theorem

- Given: a *divide and conquer* algorithm
  - An algorithm that divides the problem of size $n$ into $a$ subproblems, each of size $n/b$
  - Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function $f(\mathrm{n})$
- Then, the Master Theorem gives us a cookbook for the algorithm's running time:

# Review: The Master Theorem

- if T(n) = aT(n/b) + f(n) then

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \\ \Theta\left(n^{\log_b a} \log n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \\ \Theta\left(f(n)\right) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \text{ AND} \\ & af(n/b) < cf(n) \text{ for large } n \end{cases} \left.\vphantom{\begin{matrix} a \\ b \\ c \\ d \end{matrix}}\right\} \begin{matrix} \varepsilon > 0 \\ c < 1 \end{matrix}$$

# Review: Merge Sort

```
MergeSort(A, left, right) {
  if (left < right) {
      mid = floor((left + right) / 2);
      MergeSort(A, left, mid);
      MergeSort(A, mid+1, right);
      Merge(A, left, mid, right);
  }
}

// Merge() takes two sorted subarrays of A and
// merges them into a single sorted subarray of A.
// Merge()takes O(n) time, n = length of A
```

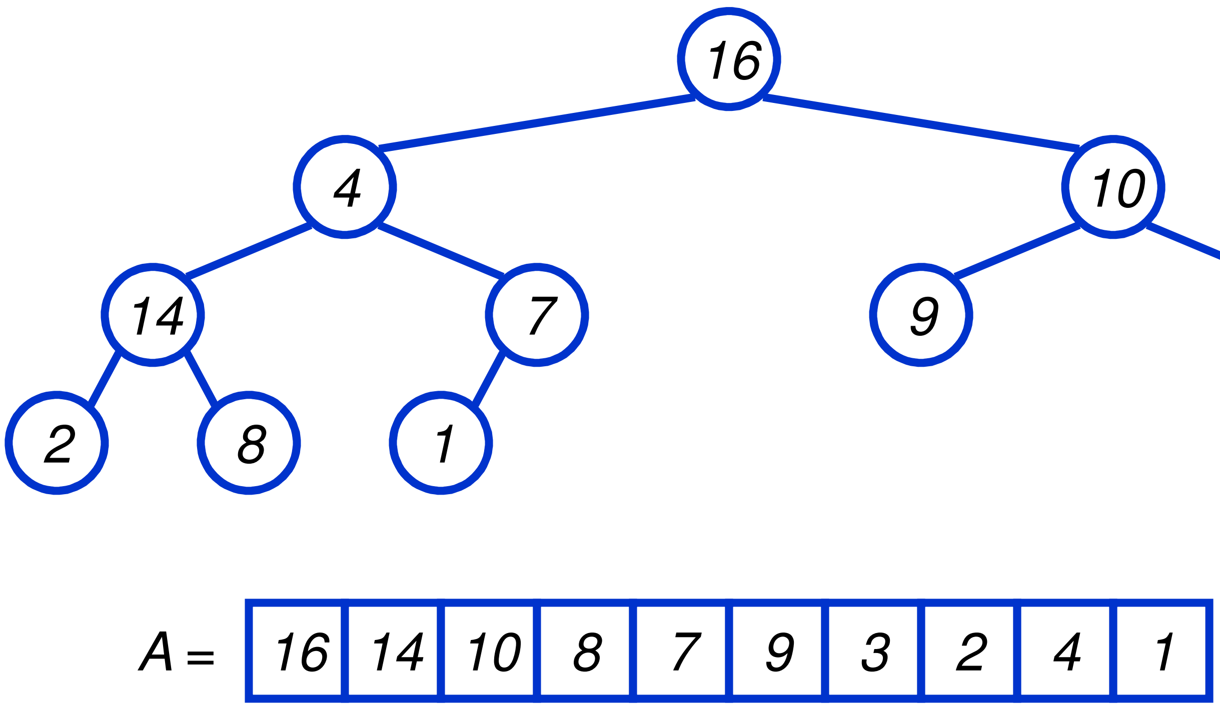
# Review: Analysis of Merge Sort

| Statement | Effort |
|---|---|
| `MergeSort(A, left, right) {` | `T(n)` |
| `    if (left < right) {` | `Θ(1)` |
| `        mid = floor((left + right) / 2);` | `Θ(1)` |
| `        MergeSort(A, left, mid);` | `T(n/2)` |
| `        MergeSort(A, mid+1, right);` | `T(n/2)` |
| `        Merge(A, left, mid, right);` | `Θ(n)` |
| `    }` | |
| `}` | |

- So T(n) = $\Theta(1)$ when n = 1, and
  $2T(n/2) + \Theta(n)$ when n > 1
- Solving this recurrence (*how?*) gives T(n) = n lg n

# Review: Heaps

- A *heap* is a “complete” binary tree, usually represented as an array:



# Review: Heaps

- To represent a heap as an array:

```
Parent(i) { return ⌊i/2⌋; }
Left(i) { return 2*i; }
right(i) { return 2*i + 1; }
```

# Review: The Heap Property

- Heaps also satisfy the *heap property*:

  A[*Parent*($i$)] $\geq$ A[$i$] for all nodes $i > 1$

  - In other words, the value of a node is at most the value of its parent
  - The largest value is thus stored at the root (A[1])
- Because the heap is a binary tree, the height of any node is at most $\Theta(\lg n)$

# Review: Heapify()

- **`Heapify()`**: maintain the heap property
  - Given: a node *i* in the heap with children *l* and *r*
  - Given: two subtrees rooted at *l* and *r*, assumed to be heaps
  - Action: let the value of the parent node "float down" so subtree at *i* satisfies the heap property
    - If A[i] < A[l] or A[i] < A[r], swap A[i] with the largest of A[l] and A[r]
    - Recurse on that subtree
  - Running time: O(*h*), h = height of heap = O(lg *n*)

# Review: BuildHeap()

- **`BuildHeap()`**: build heap bottom-up by running **`Heapify()`** on successive subarrays
  - Walk backwards through the array from n/2 to 1, calling **`Heapify()`** on each node.
  - Order of processing guarantees that the children of node *i* are heaps when *i* is processed
- Easy to show that running time is $O(n \lg n)$
- Can be shown to be $O(n)$
  - Key observation: most subheaps are small

# Review: Heapsort()

- **`Heapsort()`**: an in-place sorting algorithm:
  - Maximum element is at A[1]
  - Discard by swapping with element at A[n]
    - Decrement heap_size[A]
    - A[n] now contains correct value
  - Restore heap property at A[1] by calling **`Heapify()`**
  - Repeat, always swapping A[1] for A[heap_size(A)]
- Running time: O($n$ lg $n$)
  - **`BuildHeap`**: O($n$), **`Heapify`**: $n$ * O(lg $n$)

# Review: Priority Queues

- The heap data structure is often used for implementing *priority queues*
  - A data structure for maintaining a set $S$ of elements, each with an associated value or *key*
  - Supports the operations **`Insert()`**, **`Maximum()`**, and **`ExtractMax()`**
  - Commonly used for scheduling, *event simulation*

# Priority Queue Operations

- **Insert(S, x)** inserts the element x into set S
- **Maximum(S)** returns the element of S with the maximum key
- **ExtractMax(S)** removes and returns the element of S with the maximum key

# Implementing Priority Queues

```
HeapInsert(A, key)     // what's running time?
{
    heap_size[A] ++;
    i = heap_size[A];
    while (i > 1  AND  A[Parent(i)] < key)
    {
        A[i] = A[Parent(i)];
        i = Parent(i);
    }
    A[i] = key;
}
```

# Implementing Priority Queues

```
HeapMaximum(A)
{
    // This one is really tricky:

    return A[i];
}
```

# Implementing Priority Queues

```
HeapExtractMax(A)
{
    if (heap_size[A] < 1) { error; }
    max = A[1];
    A[1] = A[heap_size[A]]
    heap_size[A] --;
    Heapify(A, 1);
    return max;
}
```

# Example: Combat Billiards

- Extract the next collision $C_i$ from the queue
- Advance the system to the time $T_i$ of the collision
- Recompute the next collision(s) for the ball(s) involved
- Insert collision(s) into the queue, using the time of occurrence as the key
- Find the next overall collision $C_{i+1}$ and repeat

# Review: Quicksort

- Quicksort pros:
  - Sorts in place
  - Sorts O(n lg n) in the average case
  - Very efficient in practice
- Quicksort cons:
  - Sorts $O(n^2)$ in the worst case
  - Naïve implementation: worst-case = sorted
  - Even picking a different pivot, some particular input will take $O(n^2)$ time

# Review: Quicksort

- Another divide-and-conquer algorithm
  - The array A[p..r] is *partitioned* into two non-empty subarrays A[p..q] and A[q+1..r]
    - Invariant: All elements in A[p..q] are less than all elements in A[q+1..r]
  - The subarrays are recursively quicksorted
  - No combining step: two subarrays form an already-sorted array

# Review: Quicksort Code

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q);
        Quicksort(A, q+1, r);
    }
}
```

# Review: Partition Code

```
Partition(A, p, r)
    x = A[p];
    i = p - 1;
    j = r + 1;
    while (TRUE)
        repeat
            j--;
        until A[j] <= x;
        repeat
            i++;
        until A[i] >= x;
        if (i < j)
            Swap(A, i, j);
        else
            return j;
```

**`partition()`** *runs in O(n) time*

# Review: Analyzing Quicksort

- *What will be the worst case for the algorithm?*
  - Partition is always unbalanced
- *What will be the best case for the algorithm?*
  - Partition is perfectly balanced
- *Which is more likely?*
  - The latter, by far, except...
- *Will any particular input elicit the worst case?*
  - Yes: Already-sorted input

# Review: Analyzing Quicksort

- In the worst case:

  $T(1) = \Theta(1)$

  $T(n) = T(n - 1) + \Theta(n)$

- Works out to

  $T(n) = \Theta(n^2)$

# Review: Analyzing Quicksort

- In the best case:

  $T(n) = 2T(n/2) + \Theta(n)$

- Works out to

  $T(n) = \Theta(n \lg n)$

# Review: Analyzing Quicksort

- Average case works out to $T(n) = \Theta(n \lg n)$
- Glance over the proof (lecture 6) but you won't have to know the details
- Key idea: analyze the running time based on the expected split caused by Partition()

# Review: Improving Quicksort

- The real liability of quicksort is that it runs in $O(n^2)$ on already-sorted input
- Book discusses two solutions:
  - Randomize the input array, OR
  - *Pick a random pivot element*
- *How do these solve the problem?*
  - By insuring that no particular input can be chosen to make quicksort run in $O(n^2)$ time

# Sorting Summary

- Insertion sort:
  - Easy to code
  - Fast on small inputs (less than ~50 elements)
  - Fast on nearly-sorted inputs
  - $O(n^2)$ worst case
  - $O(n^2)$ average (equally-likely inputs) case
  - $O(n^2)$ reverse-sorted case

# Sorting Summary

- Merge sort:
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort subarrays
    - Linear-time merge step
  - O(n lg n) worst case
  - Doesn't sort in place

# Sorting Summary

- Heap sort:
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children's keys
  - O(n lg n) worst case
  - Sorts in place
  - Fair amount of shuffling memory around

# Sorting Summary

- Quick sort:
  - Divide-and-conquer:
    - Partition array into two subarrays, recursively sort
    - All of first subarray < all of second subarray
    - No merge step needed!
  - $O(n \lg n)$ average case
  - Fast in practice
  - $O(n^2)$ worst case
    - Naïve implementation: worst case on sorted input
    - Address this with randomized quicksort

# Review: Comparison Sorts

- Comparison sorts: O(n lg n) at best
  - Model sort with decision tree
  - Path down tree = execution trace of algorithm
  - Leaves of tree = possible permutations of input
  - Tree must have n! leaves, so O(n lg n) height

# Review: Counting Sort

- Counting sort:
  - Assumption: input is in the range 1..k
  - Basic idea:
    - Count number of elements $k \leq$ each element $i$
    - Use that number to place $i$ in position $k$ of sorted array
  - No comparisons! Runs in time O(n + k)
  - Stable sort
  - Does not sort in place:
    - O(n) array to hold sorted output
    - O(k) array for scratch storage

# Review: Counting Sort

```
CountingSort(A, B, k)
      for i=1 to k
            C[i]= 0;
      for j=1 to n
            C[A[j]] += 1;
      for i=2 to k
            C[i] = C[i] + C[i-1];
      for j=n downto 1
            B[C[A[j]]] = A[j];
            C[A[j]] -= 1;
```

# Review: Radix Sort

- Radix sort:
  - Assumption: input has $d$ digits ranging from 0 to $k$
  - Basic idea:
    - Sort elements by digit starting with *least* significant
    - Use a stable sort (like counting sort) for each stage
  - Each pass over $n$ numbers with $d$ digits takes time $O(n+k)$, so total time $O(dn+dk)$
    - When $d$ is constant and $k=O(n)$, takes $O(n)$ time
  - Fast! Stable! Simple!
  - Doesn't sort in place

# Review: Binary Search Trees

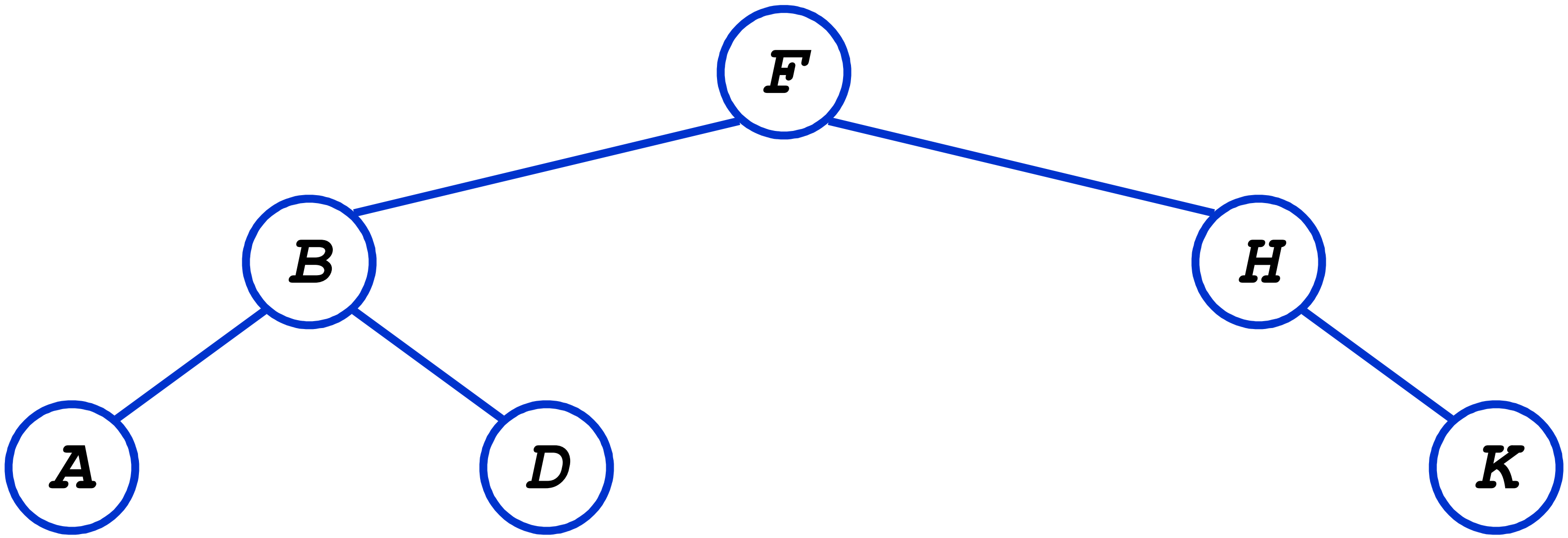
- *Binary Search Trees* (BSTs) are an important data structure for dynamic sets
- In addition to satellite data, elements have:
  - *key*: an identifying field inducing a total ordering
  - *left*: pointer to a left child (may be NULL)
  - *right*: pointer to a right child (may be NULL)
  - *p*: pointer to a parent node (NULL for root)

# Review: Binary Search Trees

- BST property:

  key[left(x)] ≤ key[x] ≤ key[right(x)]

- Example:



# Review: Inorder Tree Walk

- An *inorder walk* prints the set in sorted order:

```
TreeWalk(x)
    TreeWalk(left[x]);
    print(x);
    TreeWalk(right[x]);
```

  - Easy to show by induction on the BST property
  - *Preorder tree walk*: print root, then left, then right
  - *Postorder tree walk*: print left, then right, then root

# Review: BST Search

```
TreeSearch(x, k)
    if (x = NULL  or  k = key[x])
        return x;
    if (k < key[x])
        return TreeSearch(left[x], k);
    else
        return TreeSearch(right[x], k);
```

# Review: BST Search (Iterative)

```
IterativeTreeSearch(x, k)
    while (x != NULL  and  k != key[x])
        if (k < key[x])
            x = left[x];
        else
            x = right[x];
    return x;
```

# Review: BST Insert

- Adds an element x to the tree so that the binary search tree property continues to hold
- The basic algorithm
  - Like the search procedure above
  - Insert x in place of NULL
  - Use a "trailing pointer" to keep track of where you came from (like inserting into singly linked list)
- Like search, takes time $O(h)$, $h$ = tree height

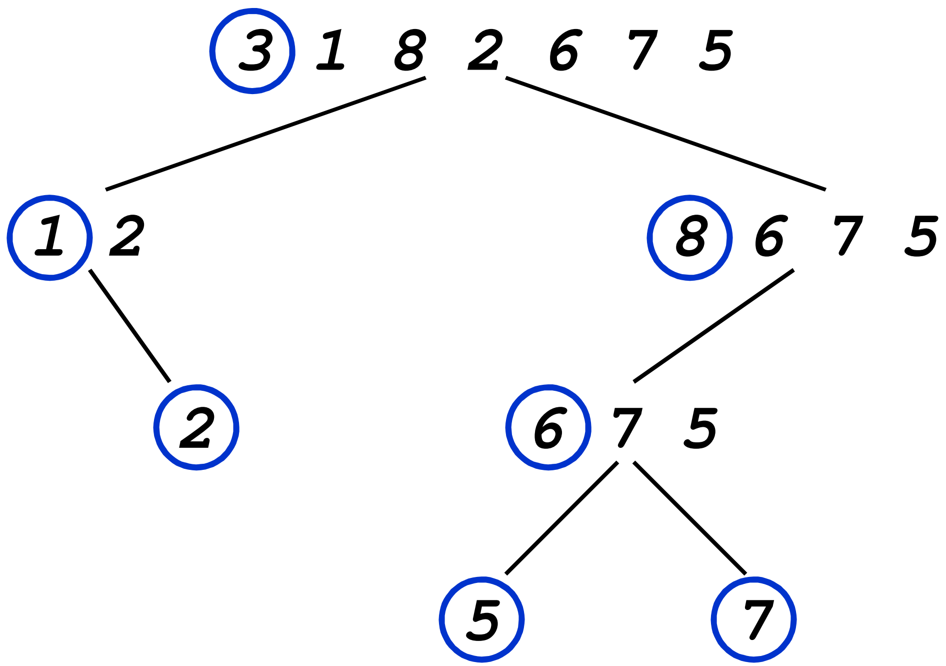
# Review: Sorting With BSTs

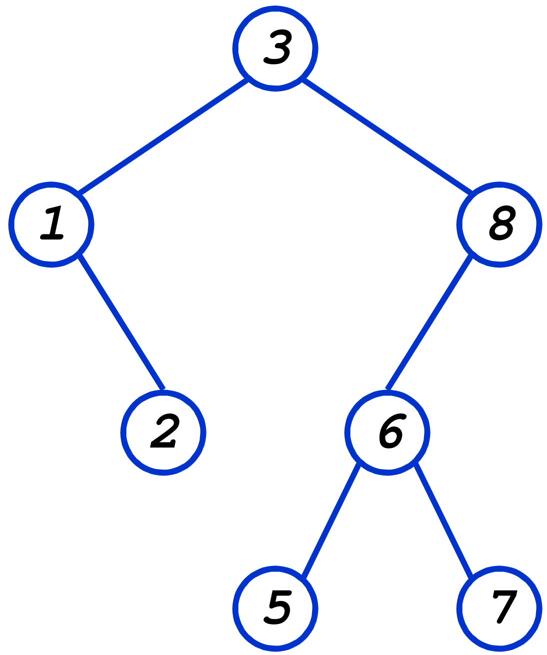
- Basic algorithm:
  - Insert elements of unsorted array from $1..n$
  - Do an inorder tree walk to print in sorted order
- Running time:
  - Best case: $\Omega(n \lg n)$ (it's a comparison sort)
  - Worst case: $O(n^2)$
  - Average case: $O(n \lg n)$ (it's a quicksort!)

# Review: Sorting With BSTs

- Average case analysis
  - It's a form of quicksort!

```
for i=1 to n
    TreeInsert(A[i]);
InorderTreeWalk(root);
```



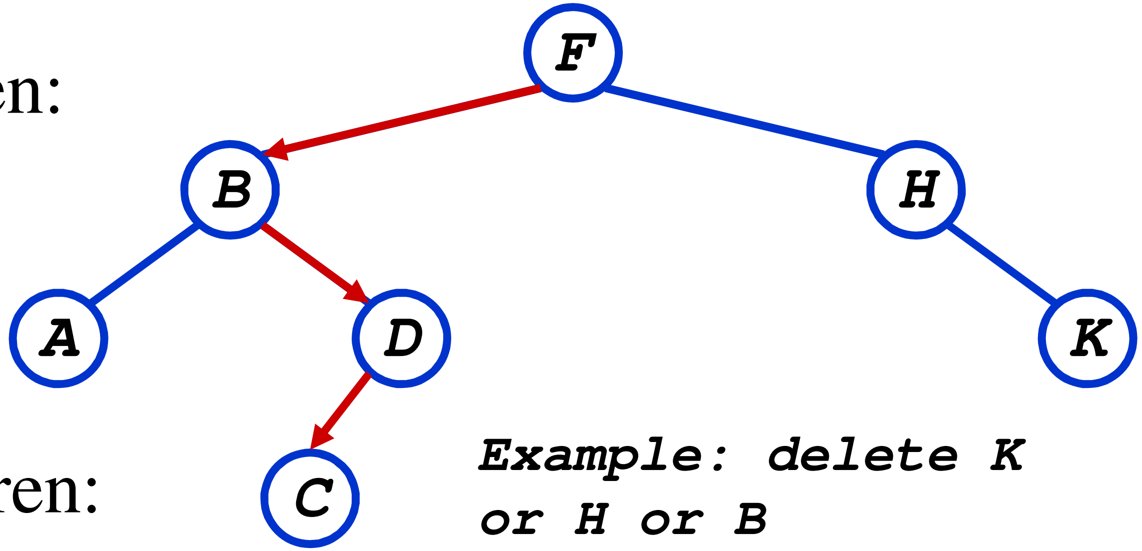



# Review: More BST Operations

- Minimum:
  - Find leftmost node in tree
- Successor:
  - x has a right subtree: successor is minimum node in right subtree
  - x has no right subtree: successor is first ancestor of x whose left child is also ancestor of x
    - Intuition: As long as you move to the left up the tree, you're visiting smaller nodes.
- Predecessor: similar to successor

# Review: More BST Operations

- Delete:
  - x has no children:
    - Remove x
  - x has one child:
    - Splice out x
  - x has two children:
    - Swap x with successor
    - Perform case 1 or 2 to delete it




*Example: delete K or H or B*

# Review: Red-Black Trees

- *Red-black trees*:
  - Binary search trees augmented with node color
  - Operations designed to guarantee that the height $h = O(\lg n)$

# Red-Black Properties

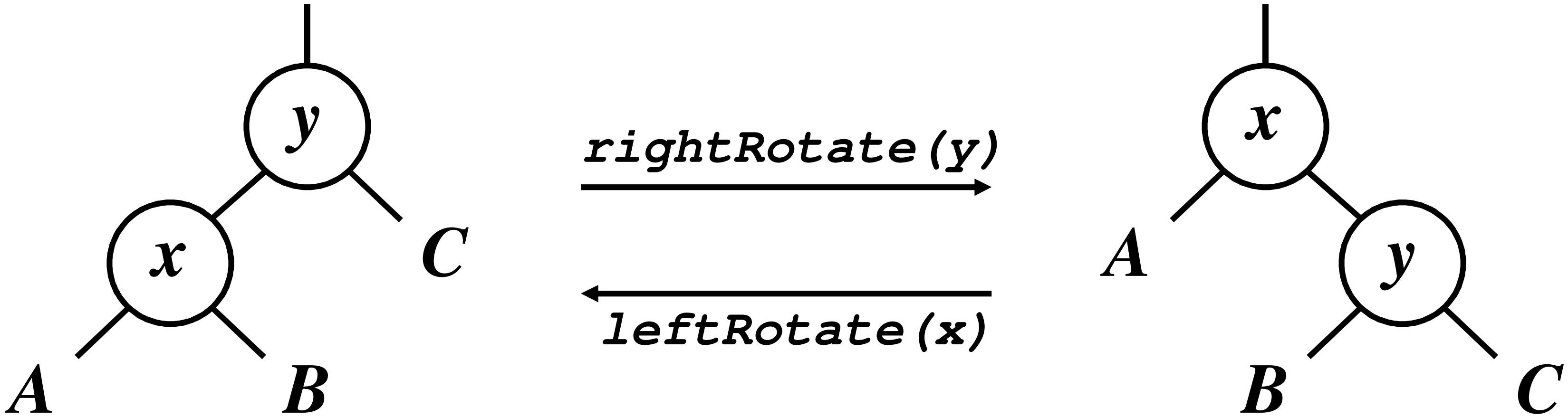
- The *red-black properties*:
  1. Every node is either red or black
  2. Every leaf (NULL pointer) is black
     - Note: this means every "real" node has 2 children
  3. If a node is red, both children are black
     - Note: can't have 2 consecutive reds on a path
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black
- *black-height*: # black nodes on path to leaf
  - Lets us prove RB tree has height $h \leq 2 \lg(n+1)$

# Operations On RB Trees

- Since height is O(lg $n$), we can show that all BST operations take O(lg $n$) time
- Problem: BST Insert() and Delete() modify the tree and could destroy red-black properties
- Solution: restructure the tree in O(lg $n$) time
  - You should understand the basic approach of these operations
  - Key operation: *rotation*

# RB Trees: Rotation

- Our basic operation for changing tree structure:



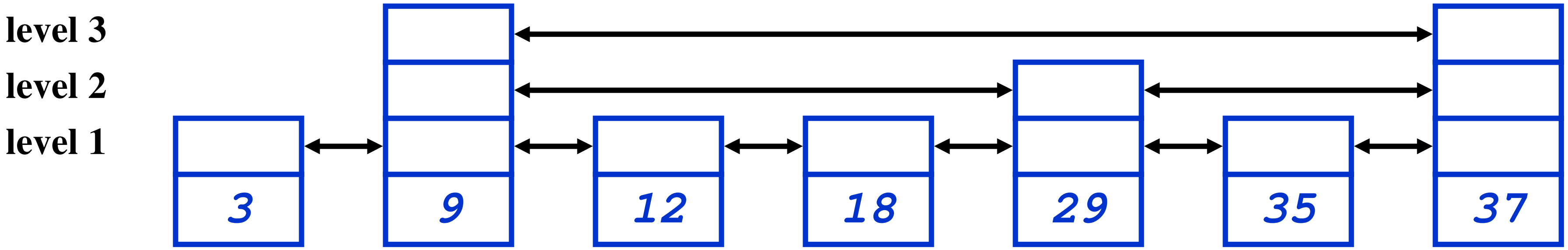

- Rotation preserves inorder key ordering
- Rotation takes O(1) time (just swaps pointers)

# Review: Skip Lists

- A relatively recent data structure
  - "A probabilistic alternative to balanced trees"
  - A randomized algorithm with benefits of r-b trees
    - O(lg $n$) expected search time
    - O(1) time for Min, Max, Succ, Pred
  - *Much* easier to code than r-b trees
  - Fast!

# Review: Skip Lists

- The basic idea:




- Keep a doubly-linked list of elements
  - Min, max, successor, predecessor: O(1) time
  - Delete is O(1) time, Insert is O(1)+Search time
- Add each level-$i$ element to level $i$+1 with probability $p$ (e.g., $p$ = 1/2 or $p$ = 1/4)

# Review: Skip List Search

- To search for an element with a given key:
  - Find location in top list
    - Top list has O(1) elements with high probability
    - Location in this list defines a range of items in next list
  - Drop down a level and recurse
- O(1) time per level on average
- O(lg $n$) levels with high probability
- Total time: O(lg $n$)

# Review: Skip List Insert

- Skip list insert: analysis
  - Do a search for that key
  - Insert element in bottom-level list
  - With probability $p$, recurse to insert in next level
  - Expected number of lists = $1 + p + p^2 + \ldots = ???$

    $= 1/(1-p) = O(1)$ if $p$ is constant
  - Total time = Search + O(1) = O(lg $n$) expected
- Skip list delete: O(1)

# Review: Skip Lists

- O(1) expected time for most operations
- O(lg $n$) expected time for insert
- O($n^2$) time worst case
  - But random, so no particular order of insertion evokes worst-case behavior
- O($n$) expected storage requirements
- Easy to code

# Review: Hashing Tables

- Motivation: symbol tables
  - A compiler uses a *symbol table* to relate symbols to associated data
    - Symbols: variable names, procedure names, etc.
    - Associated data: memory location, call graph, etc.
  - For a symbol table (also called a *dictionary*), we care about search, insertion, and deletion
  - We typically don’t care about sorted order

# Review: Hash Tables

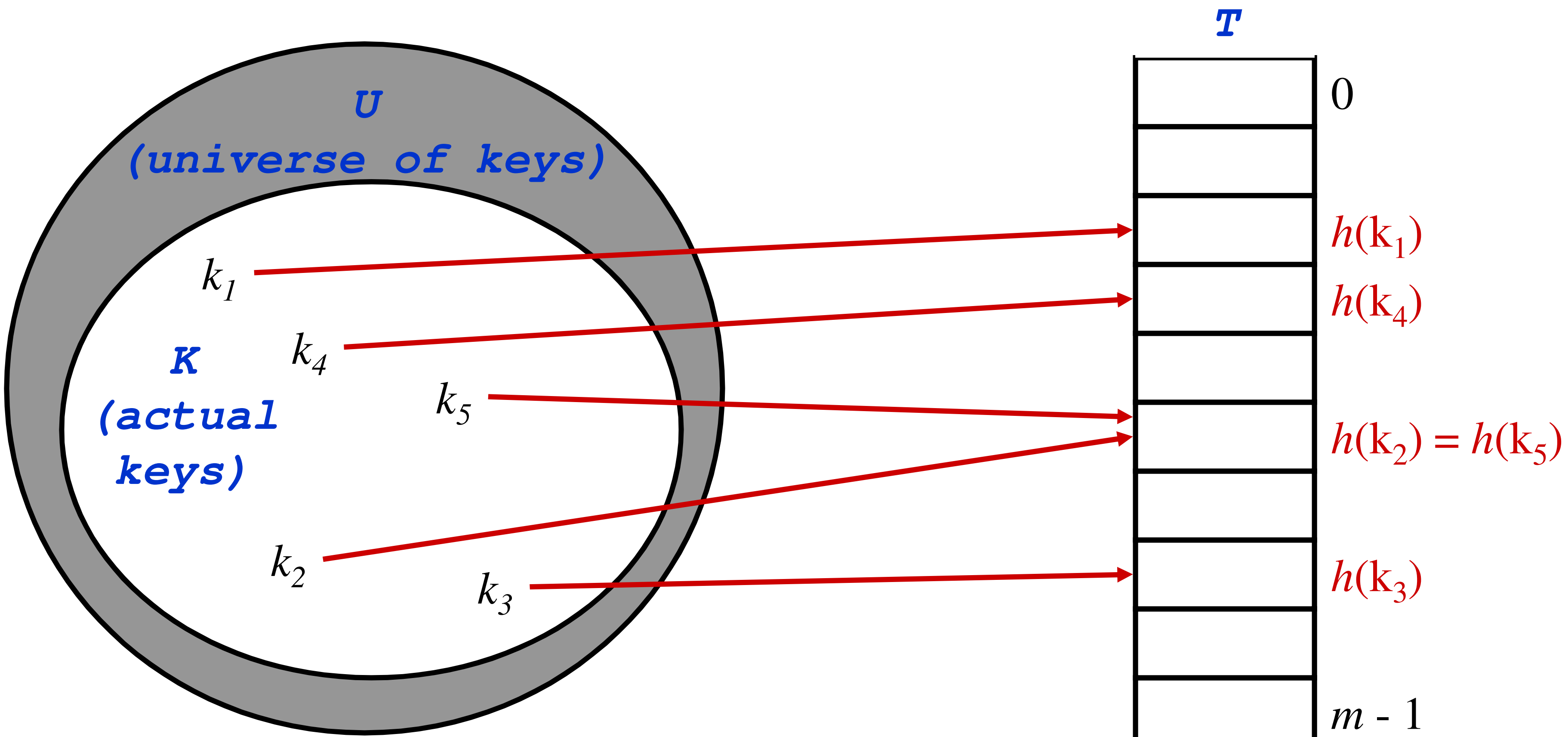
- More formally:
  - Given a table $T$ and a record $x$, with key (= symbol) and satellite data, we need to support:
    - Insert ($T$, $x$)
    - Delete ($T$, $x$)
    - Search($T$, $x$)
  - Don't care about sorting the records
- *Hash tables* support all the above in O(1) expected time

# Review: Direct Addressing

- Suppose:
  - The range of keys is 0..$m$-1
  - Keys are distinct
- The idea:
  - Use key itself as the address into the table
  - Set up an array T[0..m-1] in which
    - T[$i$] = $x$ if $x \in T$ and key[$x$] = $i$
    - T[$i$] = NULL otherwise
  - This is called a *direct-address table*

# Review: Hash Functions

- Next problem: *collision*

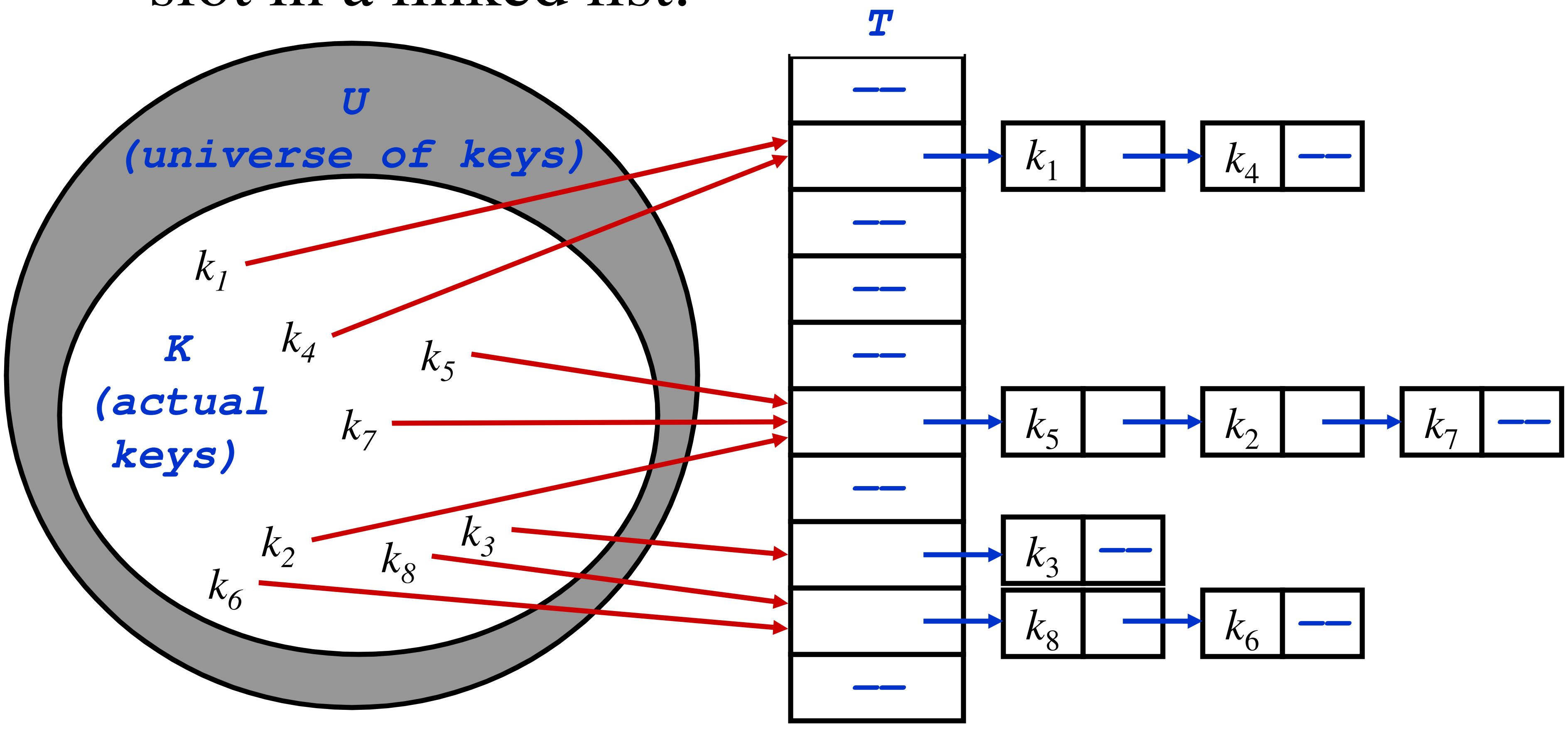


# Review: Resolving Collisions

- *How can we solve the problem of collisions?*
- *Open addressing*
  - To insert: if slot is full, try another slot, and another, until an open slot is found (*probing*)
  - To search, follow same sequence of probes as would be used when inserting the element
- *Chaining*
  - Keep linked list of elements in slots
  - Upon collision, just add new element to list

# Review: Chaining

- Chaining puts elements that hash to the same slot in a linked list:



# Review: Analysis Of Hash Tables

- *Simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- *Load factor* $\alpha = n/m$ = average # keys per slot
  - Average cost of unsuccessful search = $O(1+\alpha)$
  - Successful search: $O(1+ \alpha/2) = O(1+ \alpha)$
  - If $n$ is proportional to $m$, $\alpha = O(1)$
- So the cost of searching = O(1) if we size our table appropriately

# Review: Choosing A Hash Function

- Choosing the hash function well is crucial
  - Bad hash function puts all elements in same slot
  - A good hash function:
    - Should distribute keys uniformly into slots
    - Should not depend on patterns in the data
- We discussed three methods:
  - Division method
  - Multiplication method
  - Universal hashing

# Review: The Division Method

- $h(k) = k \bmod m$
  - In words: hash $k$ into a table with $m$ slots using the slot given by the remainder of $k$ divided by $m$
- Elements with adjacent keys hashed to different slots: good
- If keys bear relation to $m$: bad
- Upshot: pick table size $m$ = prime number not too close to a power of 2 (or 10)

# Review: The Multiplication Method

- For a constant $A$, $0 < A < 1$:
- $h(k) = \lfloor m \underbrace{(kA - \lfloor kA \rfloor)}_{\textit{Fractional part of } kA} \rfloor$
- Upshot:
  - Choose $m = 2^P$
  - Choose $A$ not too close to 0 or 1
  - Knuth: Good choice for $A = (\sqrt{5} - 1)/2$

# Review: Universal Hashing

- When attempting to foil an malicious adversary, randomize the algorithm
- *Universal hashing*: pick a hash function randomly when the algorithm begins (*not* upon every insert!)
  - Guarantees good performance on average, no matter what keys adversary chooses
  - Need a family of hash functions to choose from

# Review: Universal Hashing

- Let $\varsigma$ be a (finite) collection of hash functions
  - …that map a given universe $U$ of keys…
  - …into the range $\{0, 1, \ldots, m - 1\}$.
- If $\varsigma$ is *universal* if:
  - for each pair of distinct keys $x, y \in U$, the number of hash functions h $\in \varsigma$ for which $h(x) = h(y)$ is $|\varsigma|/m$
  - In other words:
    - With a random hash function from $\varsigma$, the chance of a collision between $x$ and $y$ ($x \neq y$) is exactly $1/m$

# Review: A Universal Hash Function

- Choose table size $m$ to be prime
- Decompose key $x$ into $r$+1 bytes, so that $x = \{x_0, x_1, \ldots, x_r\}$
  - Only requirement is that max value of byte < $m$
  - Let $a = \{a_0, a_1, \ldots, a_r\}$ denote a sequence of $r$+1 elements chosen randomly from $\{0, 1, \ldots, m - 1\}$
  - Define corresponding hash function $h_a \in \varsigma$:

$$h_a(x) = \left( \sum_{i=0}^{r} a_i x_i \right) \bmod m$$

  - With this definition, $\varsigma$ has $m^{r+1}$ members

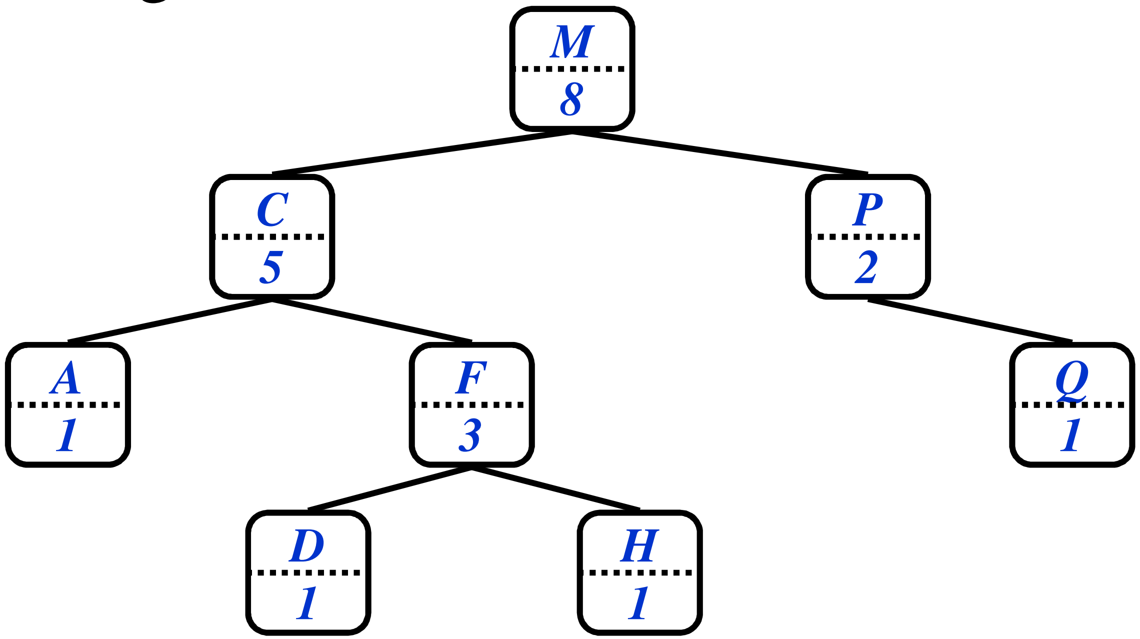
# Review: Dynamic Order Statistics

- We've seen algorithms for finding the $i$th element of an unordered set in $O(n)$ time
- *OS-Trees*: a structure to support finding the $i$th element of a dynamic set in $O(\lg n)$ time
  - Support standard dynamic set operations (`Insert()`, `Delete()`, `Min()`, `Max()`, `Succ()`, `Pred()`)
  - Also support these order statistic operations:

```
void OS-Select(root, i);
int OS-Rank(x);
```

# Review: Order Statistic Trees

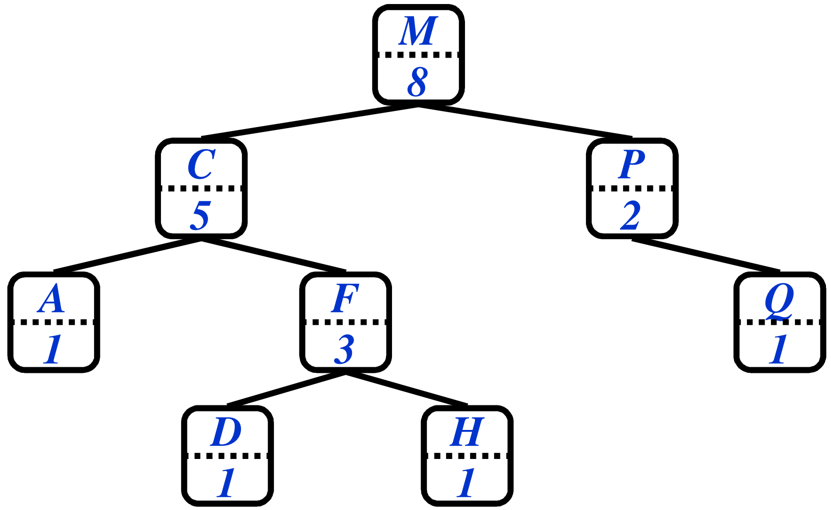
- OS Trees augment red-black trees:
  - Associate a *size* field with each node in the tree
  - **`x->size`** records the size of subtree rooted at **`x`**, including **`x`** itself:



# Review: OS-Select

- Example: show OS-Select(*root*, 5):

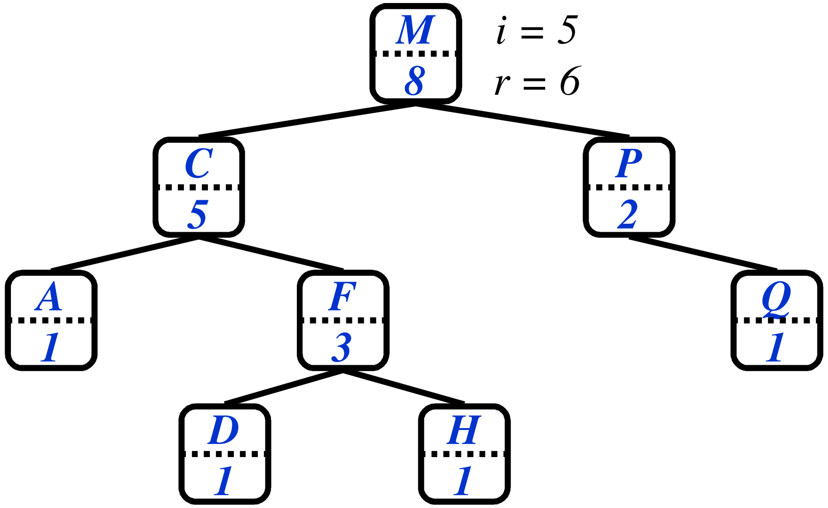
```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```



# Review: OS-Select

- Example: show OS-Select(*root*, 5):

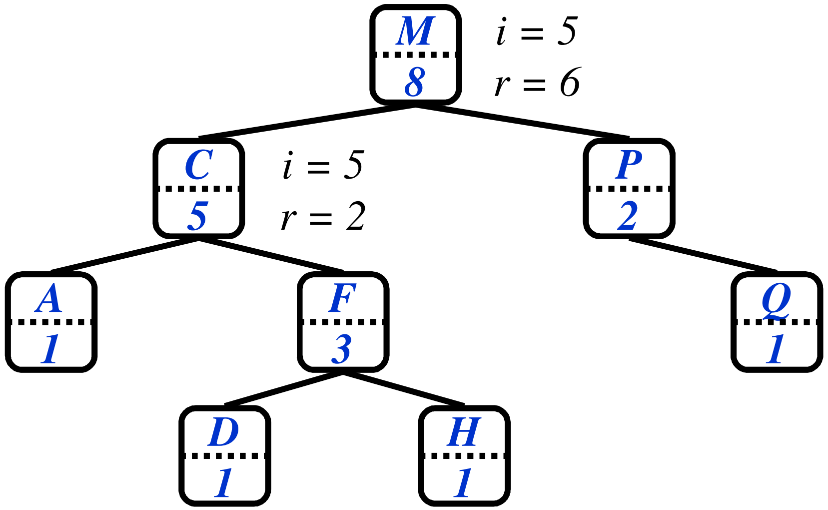
```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```



# Review: OS-Select

- Example: show OS-Select(*root*, 5):

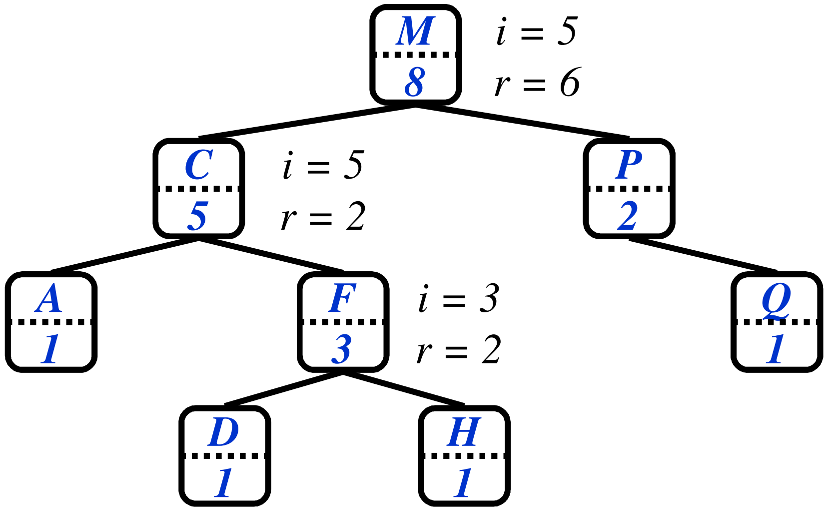
```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```



# Review: OS-Select

- Example: show OS-Select(*root*, 5):

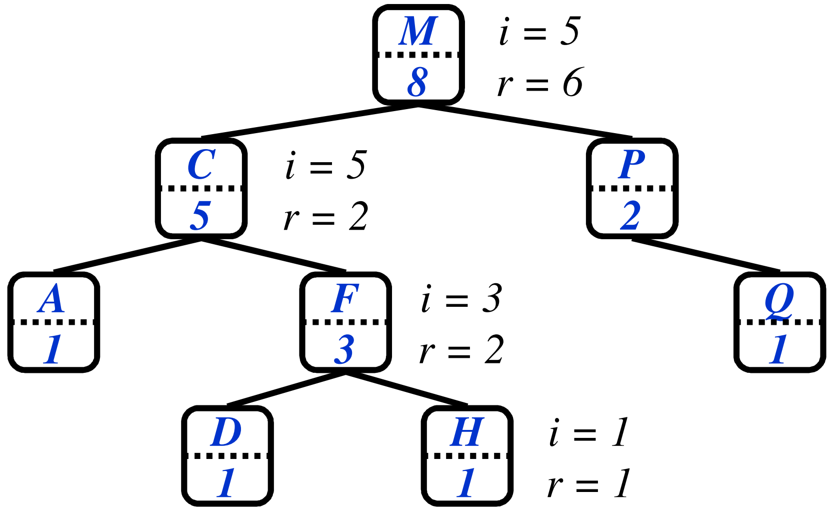
```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```



# Review: OS-Select

- Example: show OS-Select(*root*, 5):

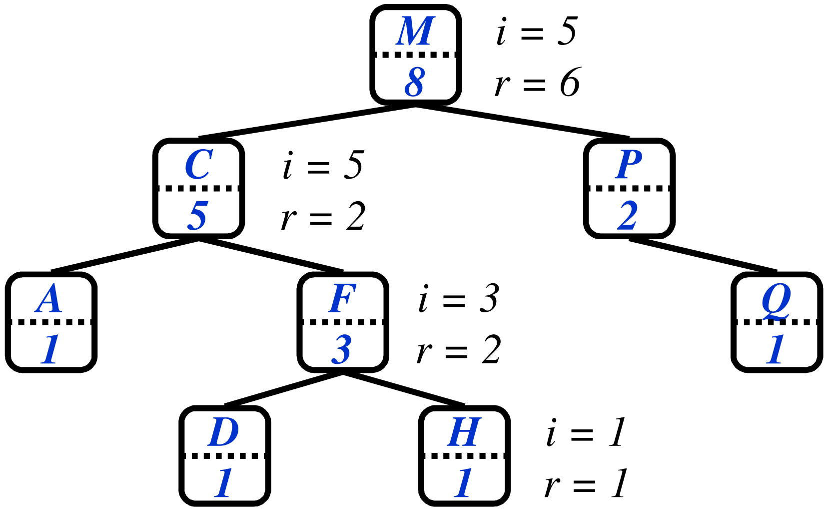
```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```



# Review: OS-Select

- Example: show OS-Select(*root*, 5):

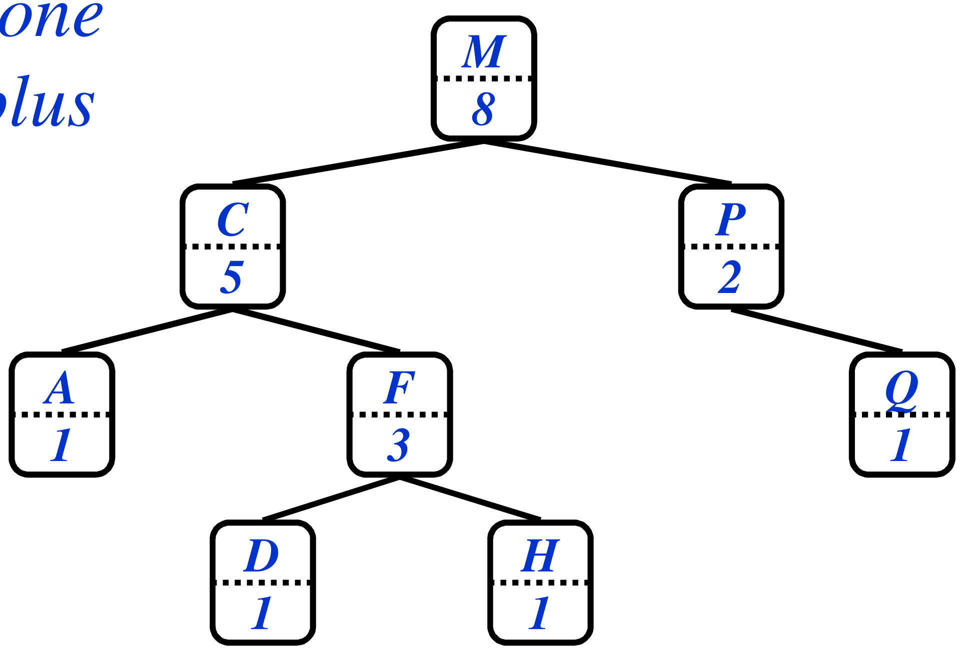
```
OS-Select(x, i)
{
  r = x->left->size + 1;
  if (i == r)
    return x;
  else if (i < r)
    return OS-Select(x->left, i);
  else
    return OS-Select(x->right, i-r);
}
```




*Note: use a sentinel NIL element at the leaves with size = 0 to simplify code, avoid testing for NULL*

# Review: Determining The Rank Of An Element

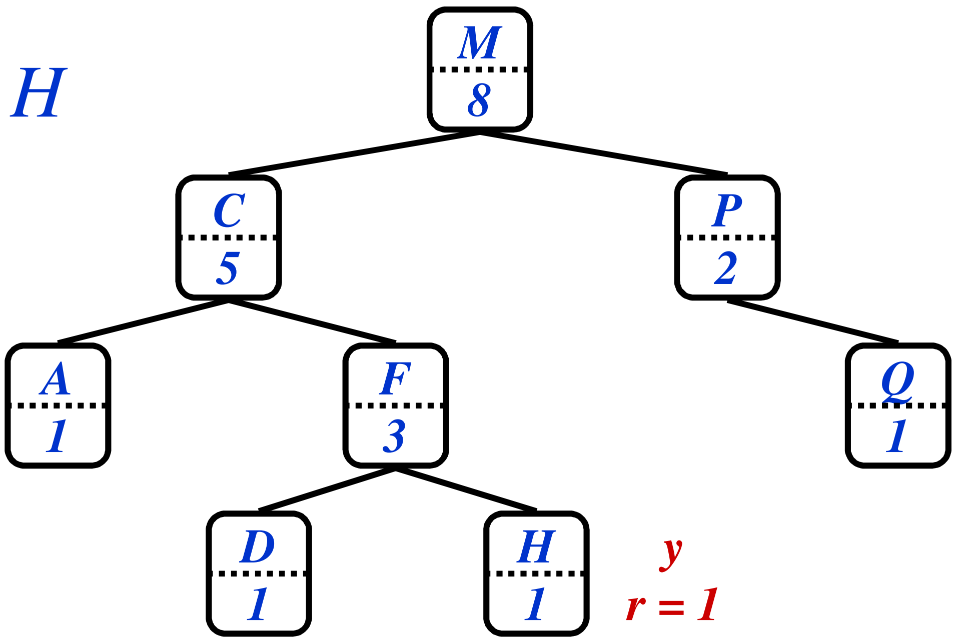
*Idea: rank of right child x is one more than its parent's rank, plus the size of x's left subtree*




```
OS-Rank(T, x)
{
    r = x->left->size + 1;
    y = x;
    while (y != T->root)
        if (y == y->p->right)
            r = r + y->p->left->size + 1;
        y = y->p;
    return r;
}
```

# Review: Determining The Rank Of An Element

*Example 1:*
*find rank of element with key H*

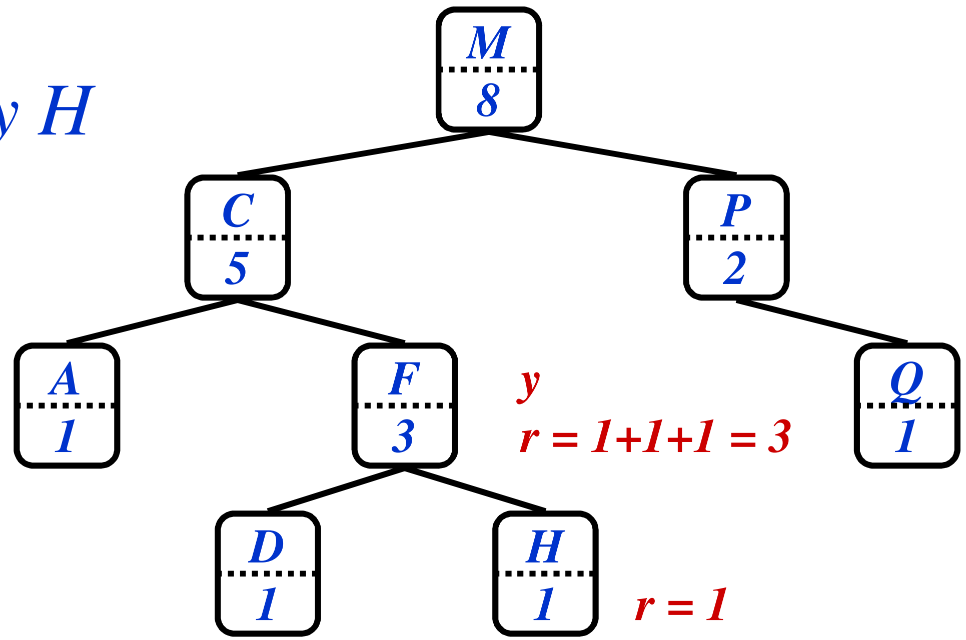



```
OS-Rank(T, x)
{
    r = x->left->size + 1;
    y = x;
    while (y != T->root)
        if (y == y->p->right)
            r = r + y->p->left->size + 1;
        y = y->p;
    return r;
}
```

# Review: Determining The Rank Of An Element

*Example 1:*

*find rank of element with key H*

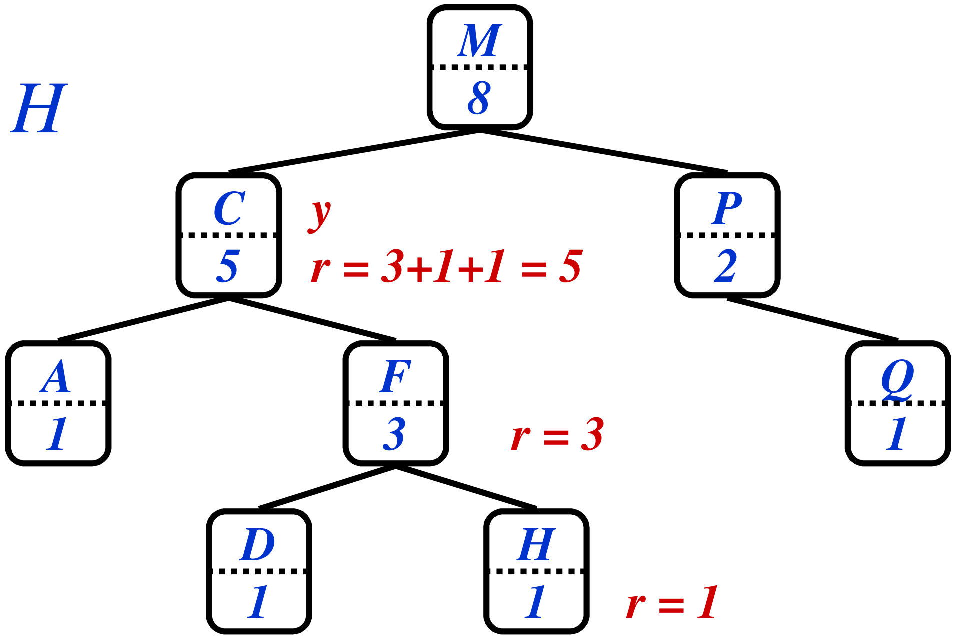



```
OS-Rank(T, x)
{
    r = x->left->size + 1;
    y = x;
    while (y != T->root)
        if (y == y->p->right)
            r = r + y->p->left->size + 1;
        y = y->p;
    return r;
}
```

# Review: Determining The Rank Of An Element

*Example 1:*

*find rank of element with key H*

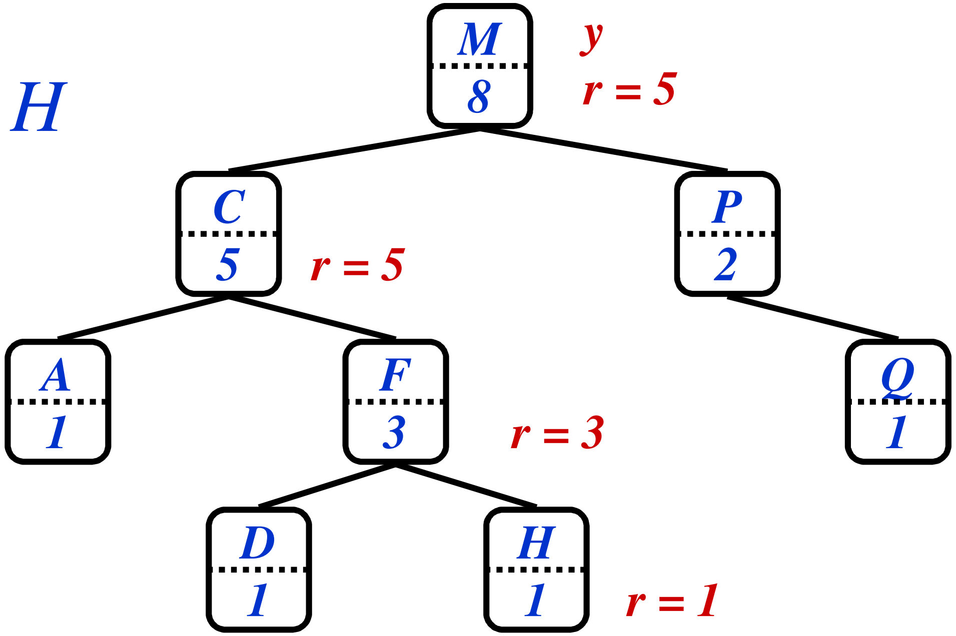



```
OS-Rank(T, x)
{
    r = x->left->size + 1;
    y = x;
    while (y != T->root)
        if (y == y->p->right)
            r = r + y->p->left->size + 1;
        y = y->p;
    return r;
}
```

# Review: Determining The Rank Of An Element

*Example 1:*

*find rank of element with key H*

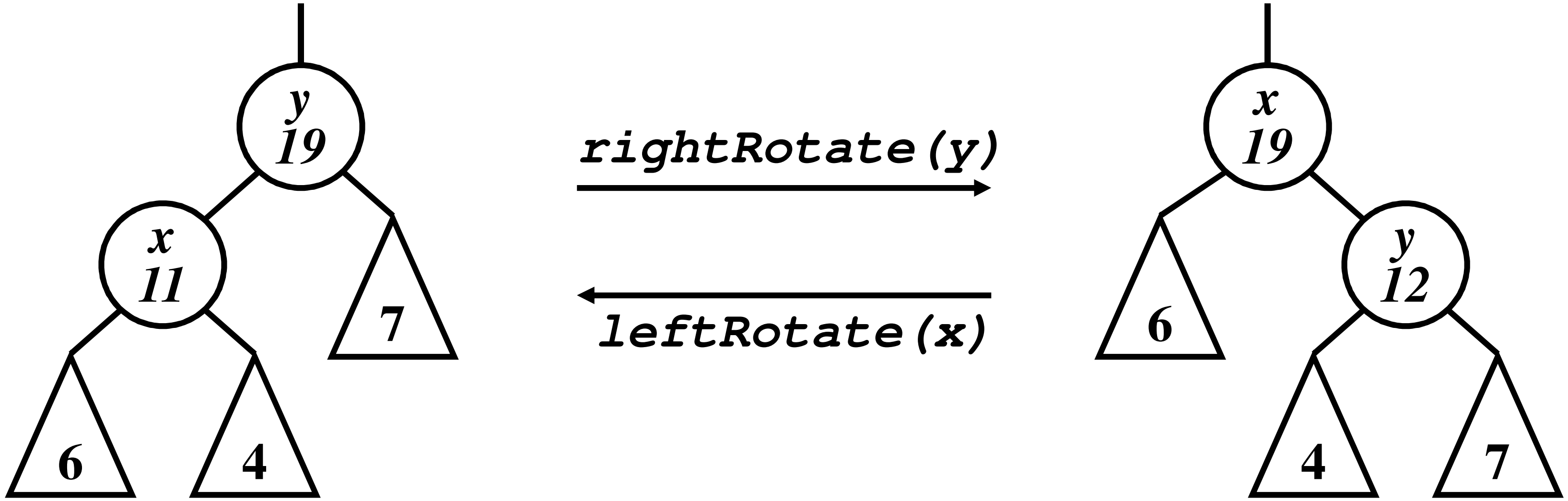



```
OS-Rank(T, x)
{
    r = x->left->size + 1;
    y = x;
    while (y != T->root)
        if (y == y->p->right)
            r = r + y->p->left->size + 1;
        y = y->p;
    return r;
}
```

# Review: Maintaining Subtree Sizes

- So by keeping subtree sizes, order statistic operations can be done in O(lg n) time
- Next: maintain sizes during Insert() and Delete() operations
  - Insert(): Increment size fields of nodes traversed during search down the tree
  - Delete(): Decrement sizes along a path from the deleted node to the root
  - Both: Update sizes correctly during rotations

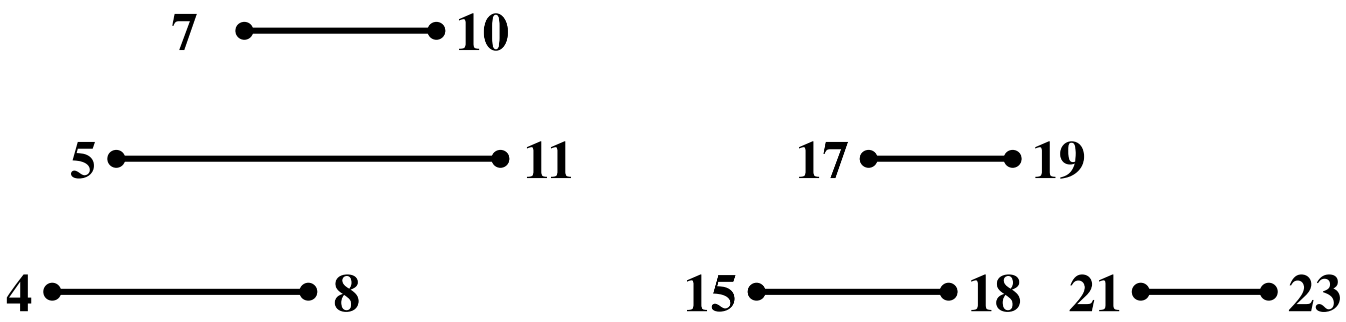
# Reivew: Maintaining Subtree Sizes




- Note that rotation invalidates only *x* and *y*
- Can recalculate their sizes in constant time
- Thm 15.1: can compute any property in O(lg n) time that depends only on node, left child, and right child

# Review: Interval Trees

- The problem: maintain a set of intervals
  - E.g., time intervals for a scheduling program:




  - Query: find an interval in the set that overlaps a given query interval
    - [14,16] → [15,18]
    - [16,19] → [15,18] or [17,19]
    - [12,14] → NULL

# Interval Trees

- Following the methodology:
  - Pick underlying data structure
    - Red-black trees will store intervals, keyed on $i \rightarrow low$
  - Decide what additional information to store
    - Store the maximum endpoint in the subtree rooted at $i$
  - Figure out how to maintain the information
    - Insert: update max on way down, during rotations
    - Delete: similar
  - Develop the desired new operations

# Searching Interval Trees

```
IntervalSearch(T, i)
{
    x = T->root;
    while (x != NULL && !overlap(i, x->interval))
        if (x->left != NULL && x->left->max ≥ i->low)
            x = x->left;
        else
            x = x->right;
    return x
}
```

- Running time: O(lg n)

# Review: Correctness of IntervalSearch()

- Key idea: need to check only 1 of node's 2 children
  - Case 1: search goes right
    - Show that ∃ overlap in right subtree, or no overlap at all
  - Case 2: search goes left
    - Show that ∃ overlap in left subtree, or no overlap at all

# Review: Correctness of IntervalSearch()

- Case 1: if search goes right, ∃ overlap in the right subtree or no overlap in either subtree
  - If ∃ overlap in right subtree, we're done
  - Otherwise:
    - $x \rightarrow left = \text{NULL}$, or $x \rightarrow left \rightarrow max < x \rightarrow low$ (*Why?*)
    - Thus, no overlap in left subtree!

```
while (x != NULL && !overlap(i, x->interval))
        if (x->left != NULL && x->left->max ≥ i->low)
            x = x->left;
        else
            x = x->right;
    return x;
```

# Review: Correctness of IntervalSearch()

- Case 2: if search goes left, ∃ overlap in the left subtree or no overlap in either subtree
  - If ∃ overlap in left subtree, we're done
  - Otherwise:
    - i →low ≤ x →left →max, by branch condition
    - x →left →max = y →high for some y in left subtree
    - Since i and y don't overlap and i →low ≤ y →high, i →high < y →low
    - Since tree is sorted by low's, i →high < any low in right subtree
    - Thus, no overlap in right subtree

```
while (x != NULL && !overlap(i, x->interval))
        if (x->left != NULL && x->left->max ≥ i->low)
            x = x->left;
        else
            x = x->right;
    return x;
```