Analysis of Algorithms
CS 477/677

Instructor: Monica Nicolescu

Lecture 14

Augmenting Data Structures

- Let’s look at two new problems:
  - Dynamic order statistic
  - Interval search
- It is unusual to have to design all-new data structures from scratch
  - Typically: store additional information in an already known data structure
  - The augmented data structure can support new operations
- We need to correctly maintain the new information without loss of efficiency

Dynamic Order Statistics

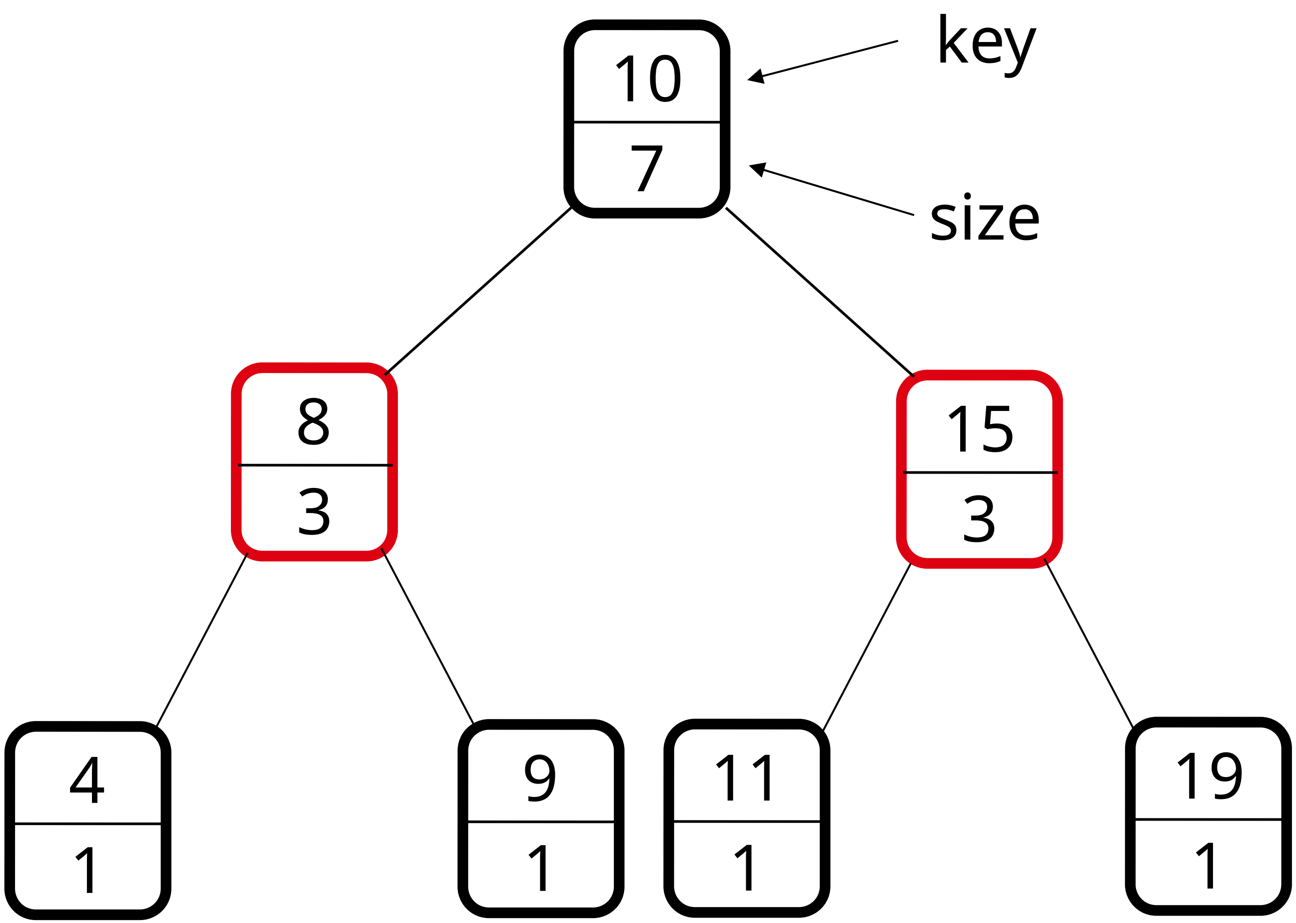
- Def.: the i-th order statistic of a set of n elements, where $i \in \{1, 2, \ldots, n\}$ is the element with the i-th smallest key.
- We can retrieve an order statistic from an unordered set:
  - Using: RANDOMIZED-SELECT
  - In: O(n) time
- We will show that:
  - With red-black trees we can achieve this in O(lgn)
  - Finding the **rank** of an element takes also O(lgn)

Order-Statistic Tree

- Def.: **Order-statistic tree:** a red-black tree with additional information stored in each node
- Node representation:
  - Usual fields: key[x], color[x], p[x], left[x], right[x]
  - Additional field: size[x] that contains the number of (internal) nodes in the subtree rooted at x (including x itself)
- For any internal node of the tree:

size[x] = size[left[x]] + size[right[x]] + 1

Example: Order-Statistic Tree



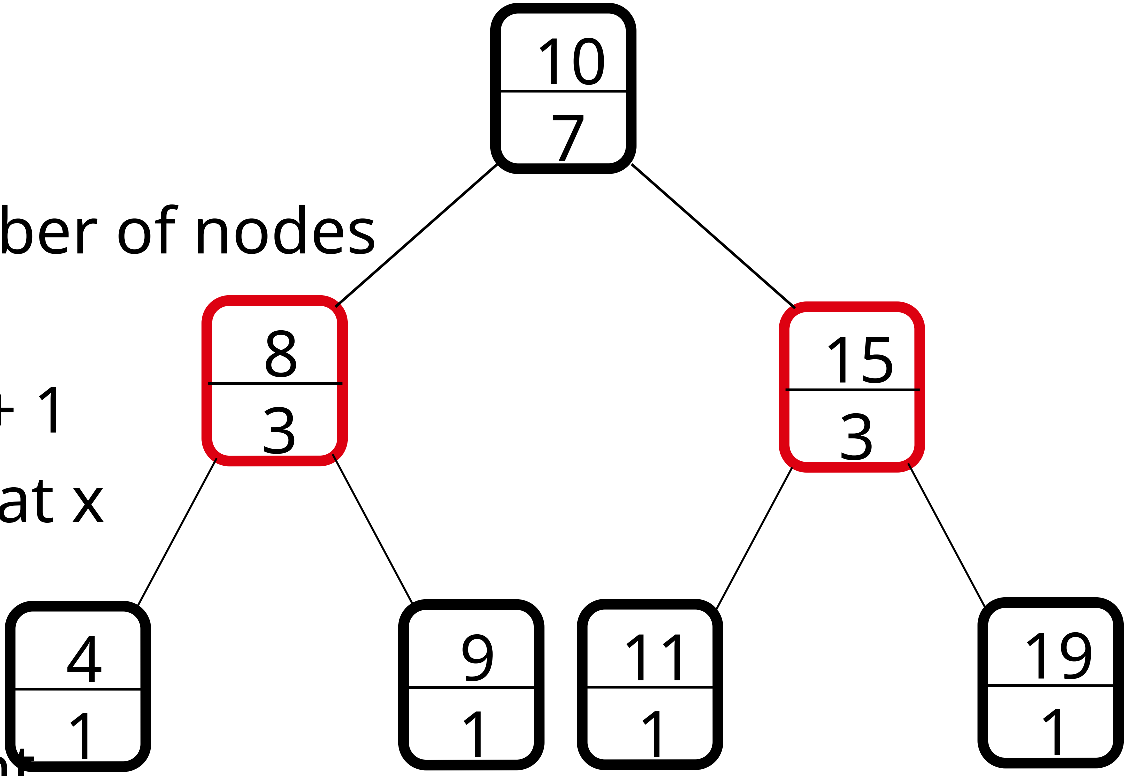
OS-SELECT

Goal:

- Given an order-statistic tree, return a pointer to the node containing the i-th smallest key in the subtree rooted at x

Idea:

- size[left[x]] = the number of nodes that are smaller than x
- rank'[x] = size[left[x]] + 1 in the subtree rooted at x
- If i = rank'[x] Done!
- If i < rank'[x]: look left
- If i > rank'[x]: look right



OS-SELECT(x, i)

r ← size[left[x]] + 1 ► compute the rank of x within the subtree rooted at x

if i = r

  then return x

elseif i < r

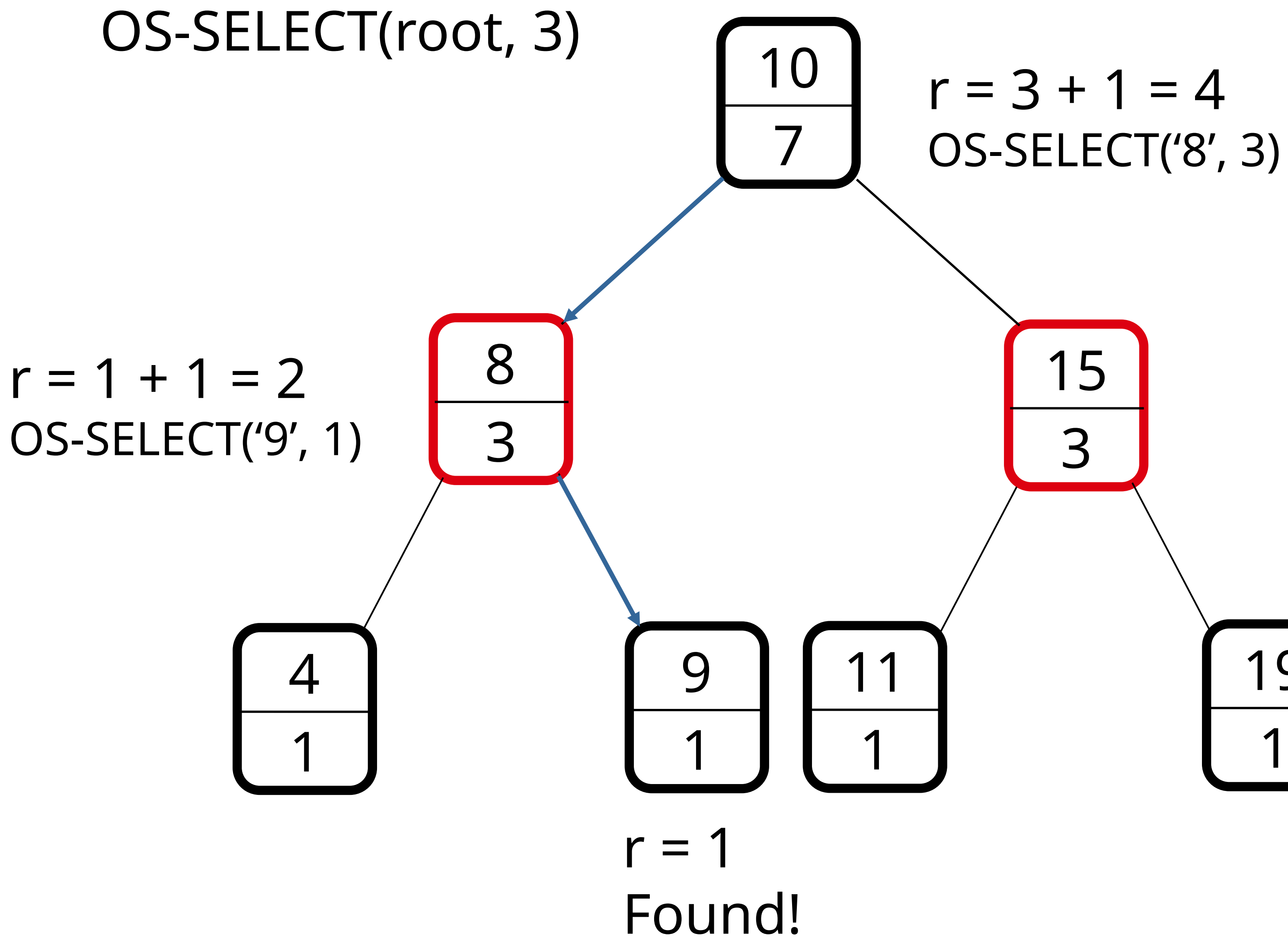
  then return OS-SELECT(left[x], i)

else return OS-SELECT(right[x], i - r)

Initial call: OS-SELECT(root[T], i)

Running time: O(lgn)

Example: OS-SELECT



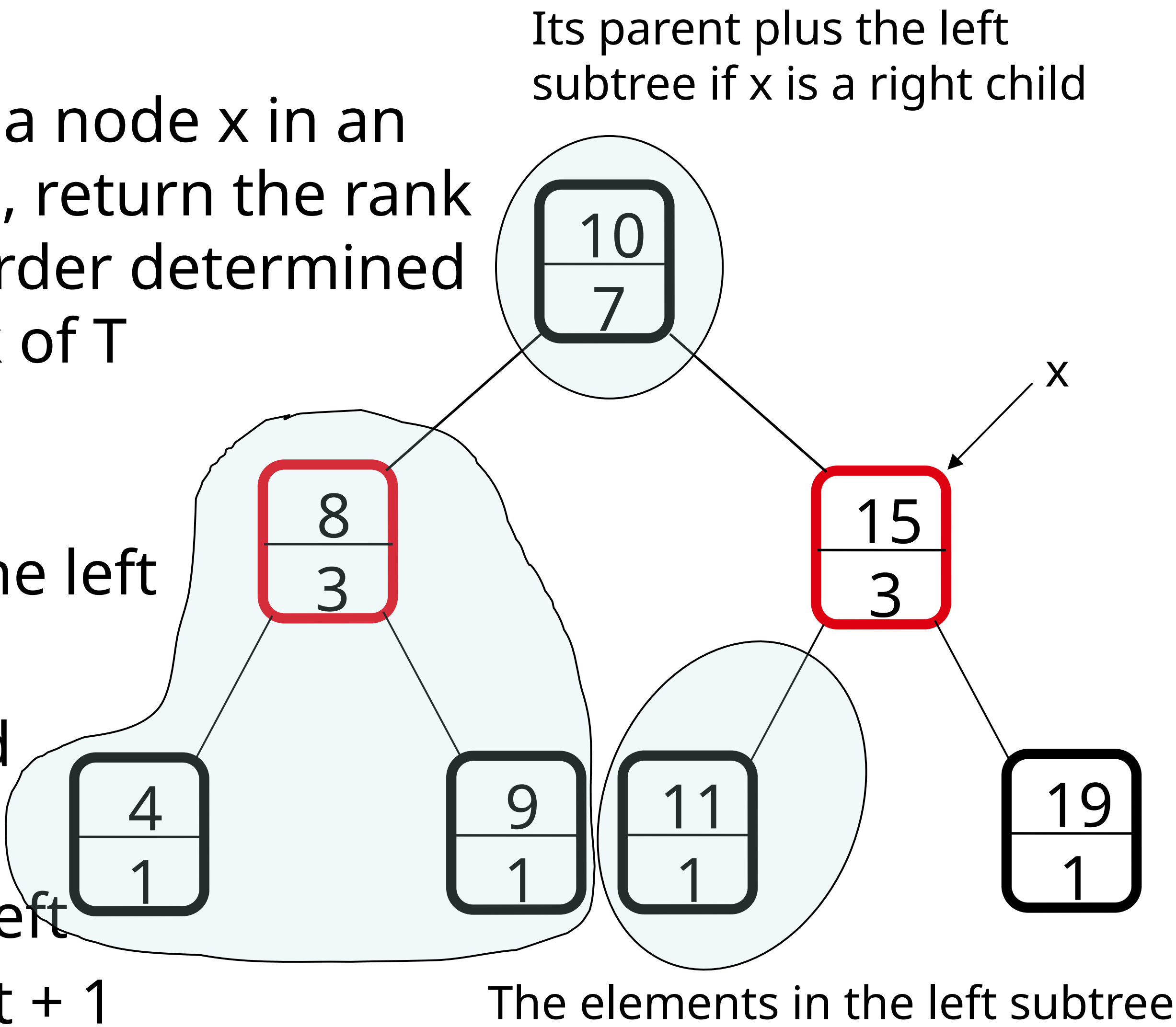
OS-RANK

Goal:

- Given a pointer to a node x in an order-statistic tree, return the rank of x in the linear order determined by an inorder walk of T

Idea:

- Add elements in the left subtree
- Go up the tree and if a right child: add the elements in the left subtree of the parent + 1



OS-RANK(T, x)

```
r ← size[left[x]] + 1
y ← x
while y ≠ root[T]
    do if y = right[p[y]]
        then r ← r + size[left[p[y]]] + 1
    y ← p[y]
return r
```

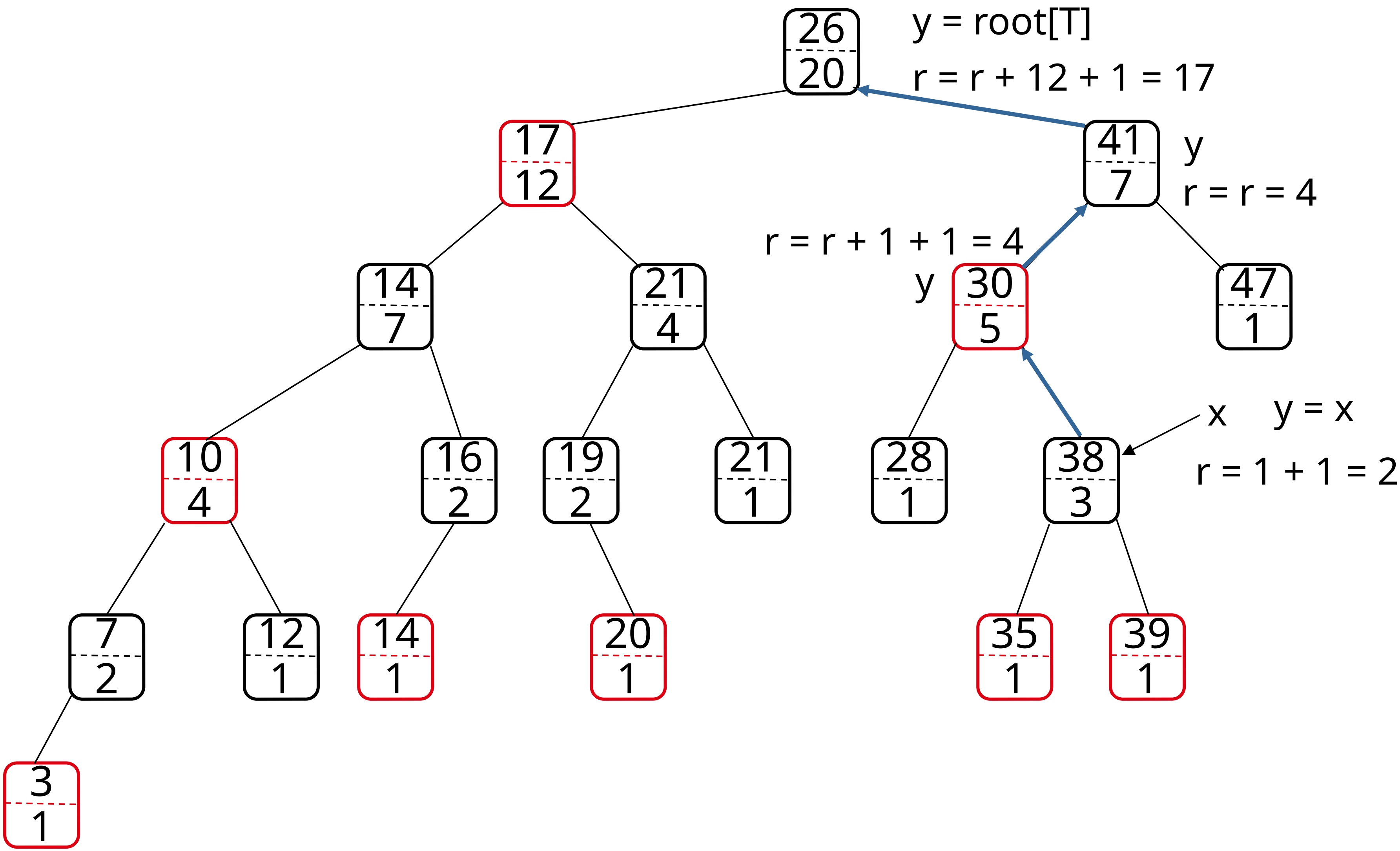
Add to the rank the elements in its left subtree + 1 for itself

Set y as a pointer that will traverse the tree

If a right child add the size of the parent's left subtree + 1 for the parent

Running time: O(lgn)

Example: OS-RANK



Maintaining Subtree Sizes

- We need to maintain the size field during INSERT and DELETE operations
- Need to maintain them efficiently
- Otherwise, might have to recompute all size fields, at a cost of $\Omega(n)$

Maintaining Size for OS-INSERT

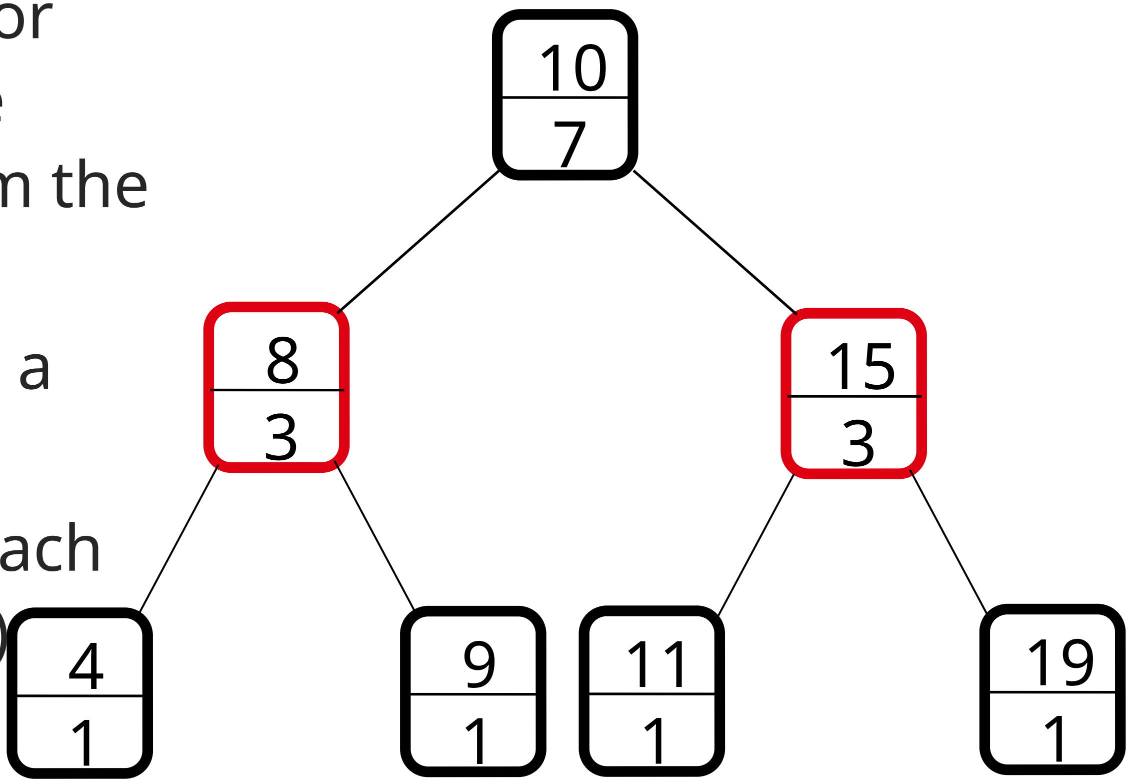
- Insert in a red-black tree has two stages
  1. Perform a binary-search tree insert
  2. Perform rotations and change node colors to restore red-black tree properties

OS-INSERT

Idea for maintaining the size field during insert

Phase 1 (going down):

- Increment size[x] for each node x on the traversed path from the root to the leaves
- The new node gets a size of 1
- Constant work at each node, so still O(lgn)



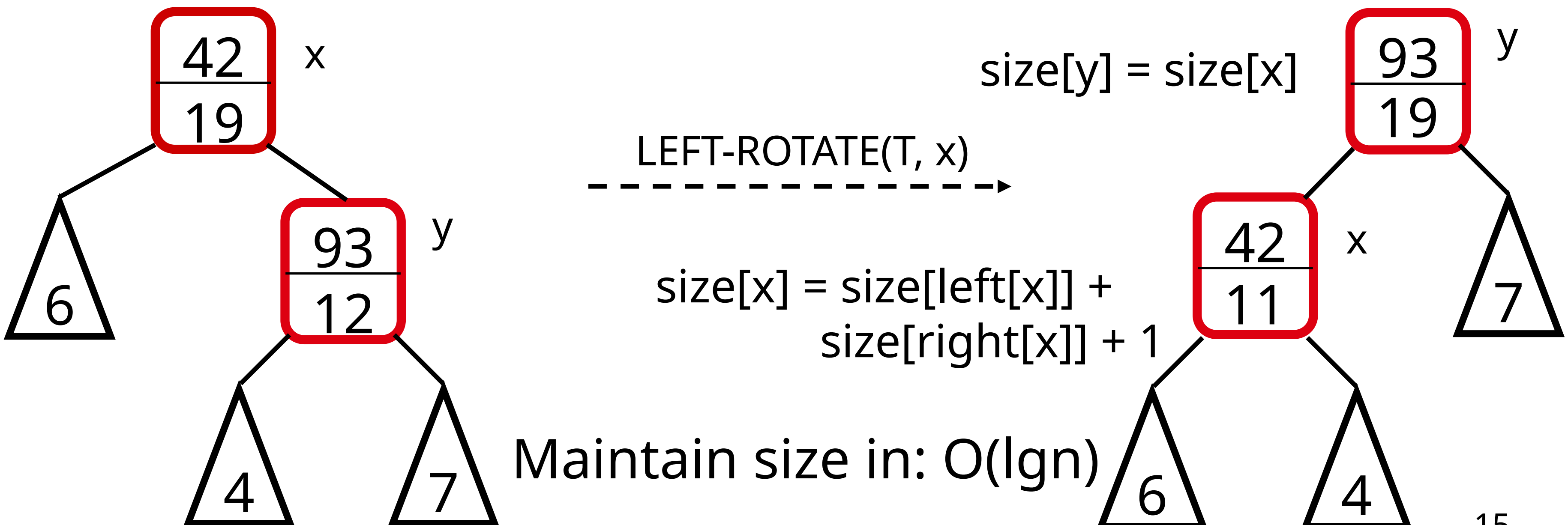
OS-INSERT

Idea for maintaining the size field during insert

Phase 2 (going up):

- During RB-INSERT-FIXUP there are:
  - O(lgn) changes in node colors
  - At most two rotations

Rotations affect the subtree sizes !!



Maintain size in: O(lgn)

Augmenting a Data Structure

1. Choose an underlying data structure

   ⇒ Red-black trees

2. Determine additional information to maintain

   ⇒ size[x]

3. Verify that we can maintain additional information for existing data structure operations

   ⇒ Shown how to maintain size during modifying operations

4. Develop new operations

   ⇒ Developed OS-RANK and OS-SELECT

Augmenting Red-Black Trees

Theorem: Let f be a field that augments a red-black tree. If the contents of f for a node can be computed using only the information in x, left[x], right[x] ⇒ we can maintain the values of f in all nodes during insertion and deletion, without affecting their O(lgn) running time.

Examples

1. Can we augment a RBT with size[x]?

   Yes: size[x] = size[left[x]] + size[right[x]] + 1

2. Can we augment a RBT with height[x]?

   Yes: height[x] = 1 + max(height[left[x]], height[right[x]])

3. Can we augment a RBT with rank[x]?

   No, inserting a new minimum will cause all n rank values to change

Interval Trees

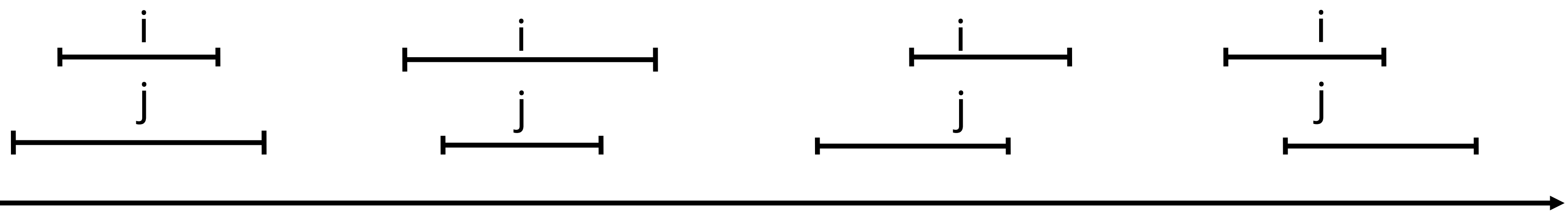
Def.: **Interval tree** = a red-black tree that maintains a dynamic set of elements, each element x having associated an interval int[x].

- Operations on interval trees:
  - INTERVAL-INSERT(T, x)
  - INTERVAL-DELETE(T, x)
  - INTERVAL-SEARCH(T, i)

Interval Properties

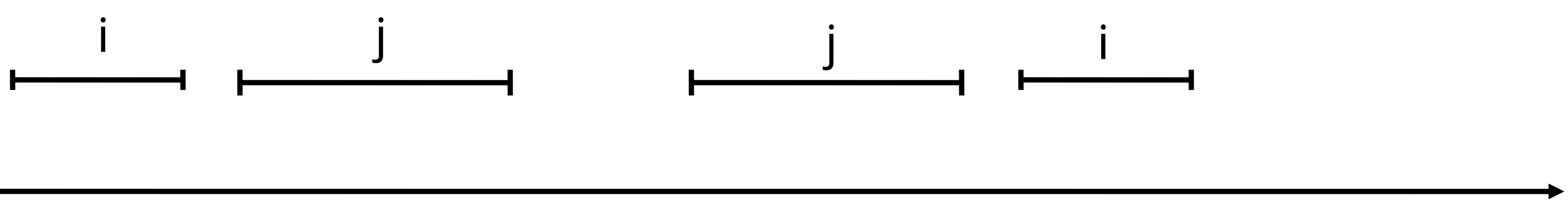
- Intervals i and j overlap iff:

  $low[i] \leq high[j]$ and $low[j] \leq high[i]$



- Intervals i and j do not overlap iff:

  $high[i] < low[j]$ or $high[j] < low[i]$



Interval Trichotomy

- Any two intervals i and j satisfy the **interval trichotomy**: exactly one of the following three properties holds:

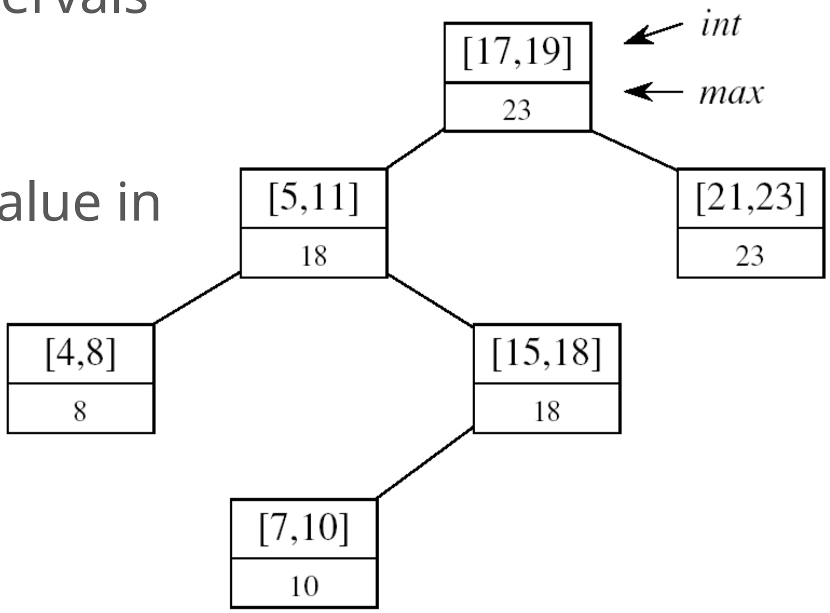
a) i and j overlap,

b) i is to the left of j ($high[i] < low[j]$)

c) i is to the right of j ($high[j] < low[i]$)

Designing Interval Trees

1. Underlying data structure
   - Red-black trees
   - Each node x contains: an interval int[x], and the key: low[int[x]]
   - An inorder tree walk will list intervals sorted by their low endpoint
2. Additional information
   - max[x] = maximum endpoint value in subtree rooted at x
3. Maintaining the information



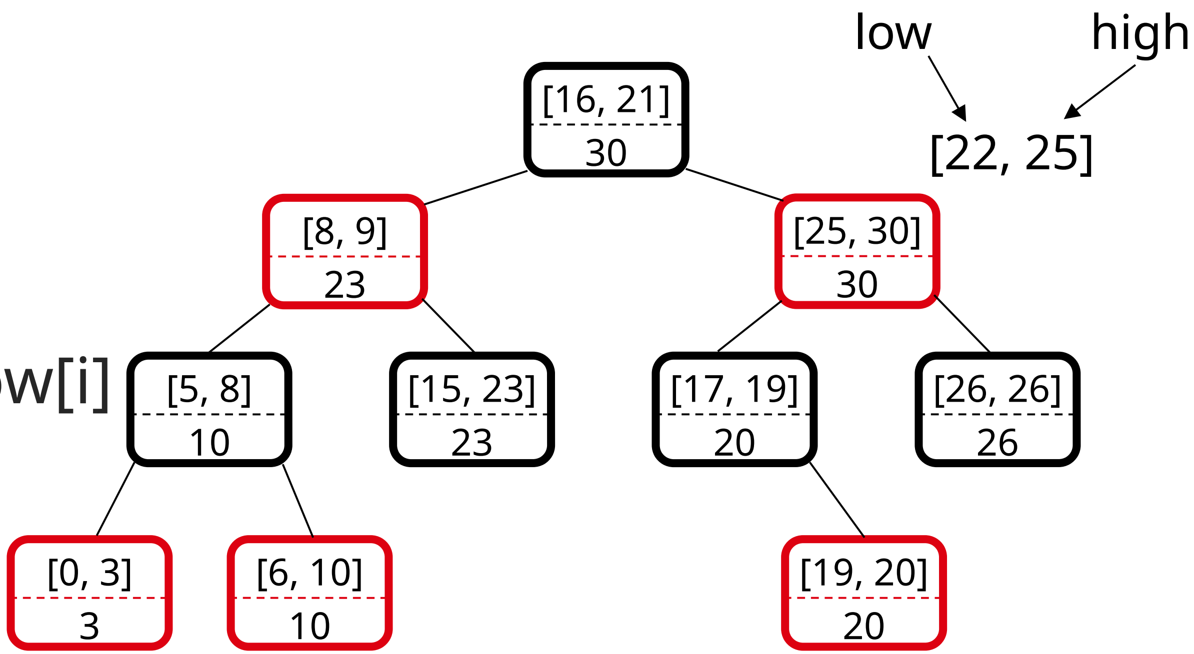
$$\max[x] = \max \begin{cases} high[int[x]] \\ \max[left[x]] \\ \max[right[x]] \end{cases}$$

Constant work at each node, so still O(lgn) time

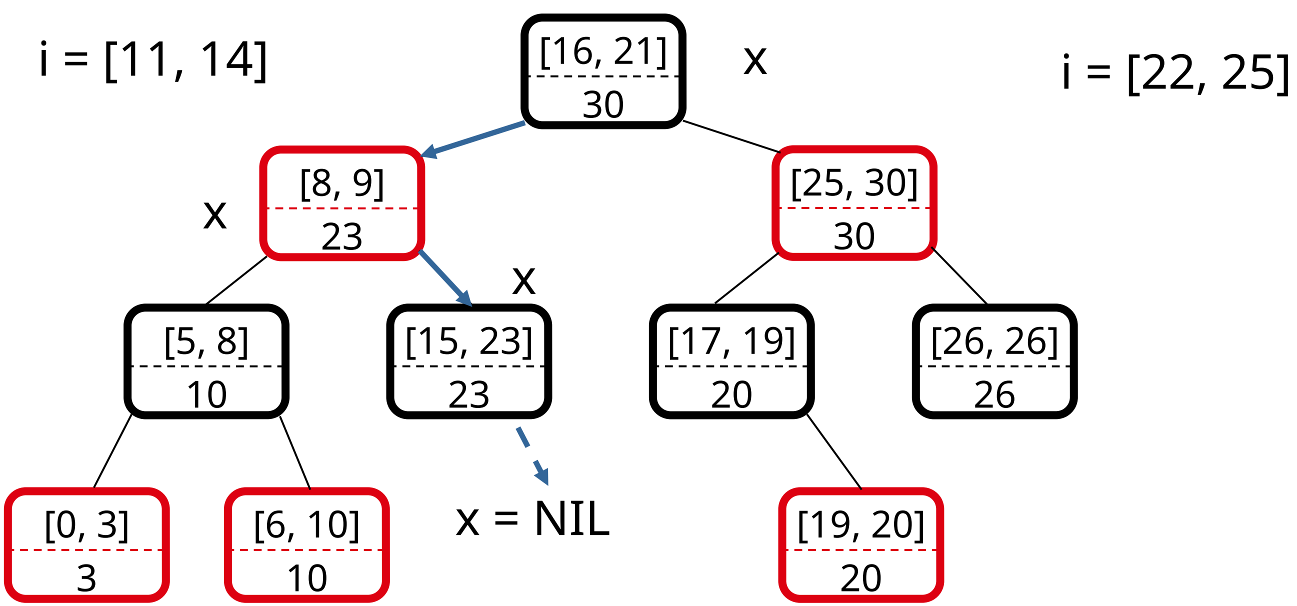
Designing Interval Trees

4. Develop new operations

- INTERVAL-SEARCH(T, i):
  - Returns a pointer to an element x in the interval tree T, such that int[x] overlaps with i, or NIL otherwise
- Idea:
- Check if int[x] overlaps with i
- Max[left[x]] ≥ low[i]
  - Go left
- Otherwise, go right



Example



i = [11, 14]
[16, 21]
30
x
i = [22, 25]
[8, 9]
23
x
[25, 30]
30
x
[5, 8]
10
[15, 23]
23
[17, 19]
20
[26, 26]
26
[0, 3]
3
[6, 10]
10
x = NIL
[19, 20]
20

INTERVAL-SEARCH(T, i)

1. x ← root[T]
2. **while** x ≠ nil[T] and i does not overlap int[x]
3. **do if** left[x] ≠ nil[T] and max[left[x]] ≥ low[i]
4. **then** x ← left[x]
5. **else** x ← right[x]
6. **return** x

Theorem

At the execution of interval search: if the search goes right, then either:

  - There is an overlap in right subtree, or
  - There is no overlap in either subtree

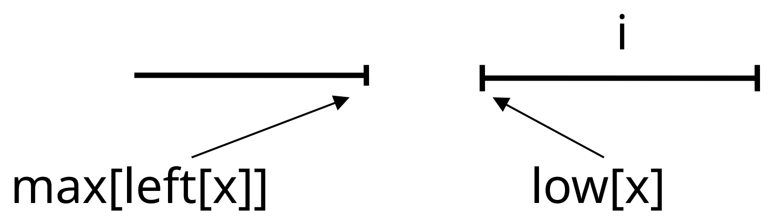
- Similar when the search goes left
- It is safe to always proceed in only one direction

Theorem

- **Proof:** If search goes right:
  - If there is an overlap in right subtree, done
  - If there is no overlap in right ⇒ show there is no overlap in left
  - Went right because:

left[x] = nil[T] ⇒ no overlap in left, or

max[left[x]] < low[i] ⇒ no overlap in left

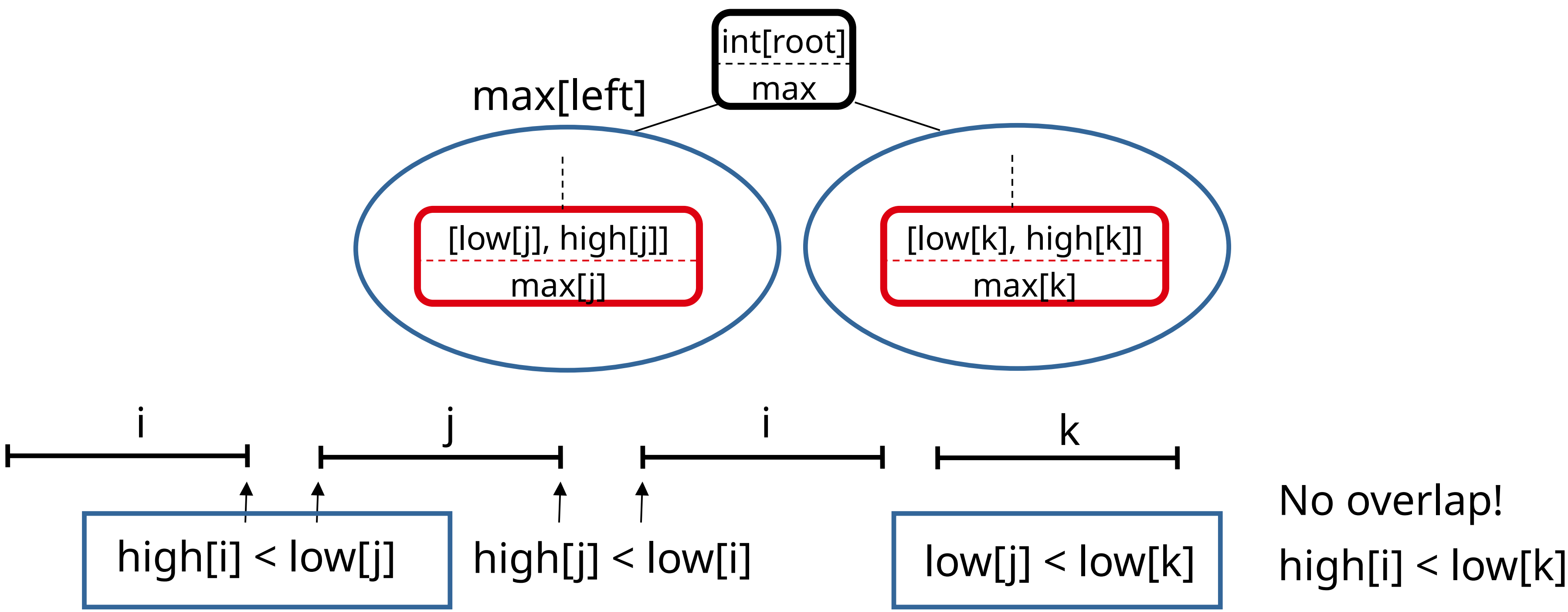


Theorem - Proof

If search goes left:

- If there is an overlap in left subtree, done
- If there is no overlap in left, show there is no overlap in right
- Went left because:

$$low[i] \le max[left[x]] = high[j] \text{ for some } j \text{ in left subtree}$$



Material up to this point included in the second midterm

MID-TERM 2

Second Midterm Exam

- Tuesday, April 2 in class
- 75 minutes
- Exam structure:
  - TRUE/FALSE questions
  - short questions on the topics discussed in class
  - homework-like problems

Topics

- All topics from midterm 1 up to dynamic programming
  - Randomized quicksort
  - Probability background
  - The selection problem
  - Sorting in linear time
  - Heaps
  - Augmenting data structures (RBT, OS-Trees, interval trees)

General Advice for Study

- **Understand** how the algorithms are working
  - Work through the examples we did in class
  - "Narrate" for yourselves the main steps of the algorithms in a few sentences
- Know **when** or **for what problems** the algorithms are applicable
- **Do not memorize** algorithms

Dynamic Programming

- An algorithm design technique used for **optimization problems**
  - Find a solution with the **optimal value** (minimum or maximum)
  - A set of **choices** must be made to get an optimal solution
  - There may be multiple solutions that return the optimal value: we want to find one of them

Dynamic Programming

- Similar to divide and conquer, but with one key difference
  - Subproblems are **not independent:** subproblems share subsubproblems
- Divide and conquer
  - Partition the problem into **independent** subproblems
  - Solve the subproblems recursively
  - Combine the solutions to solve the original problem

Dynamic Programming

- Applicable when subproblems are **not independent**
  - Subproblems share subsubproblems

E.g.: Fibonacci numbers:

- Recurrence: F(n) = F(n-1) + F(n-2)
- Boundary conditions: F(1) = 0, F(2) = 1
- Compute: F(5) = 3, F(3) = 1, F(4) = 2

  - A divide and conquer approach would repeatedly solve the common subproblems
  - Dynamic programming solves every subproblem just once and stores the answer in a table

Dynamic Programming Algorithm

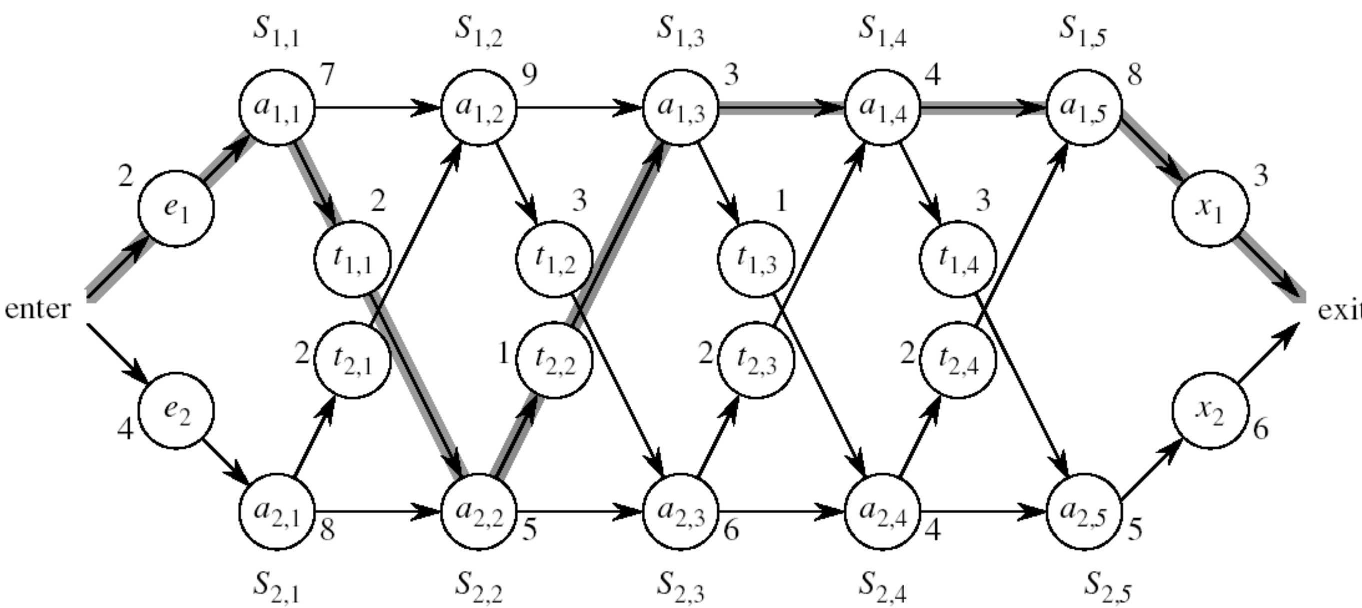
1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution in a bottom-up fashion
4. Construct an optimal solution from computed information

Elements of Dynamic Programming

- Optimal Substructure
  - An optimal solution to a problem contains within it an optimal solution to subproblems
  - Optimal solution to the entire problem is built in a bottom-up manner from optimal solutions to subproblems
- Overlapping Subproblems
  - If a recursive algorithm revisits the same subproblems again and again $\Rightarrow$ the problem has overlapping subproblems

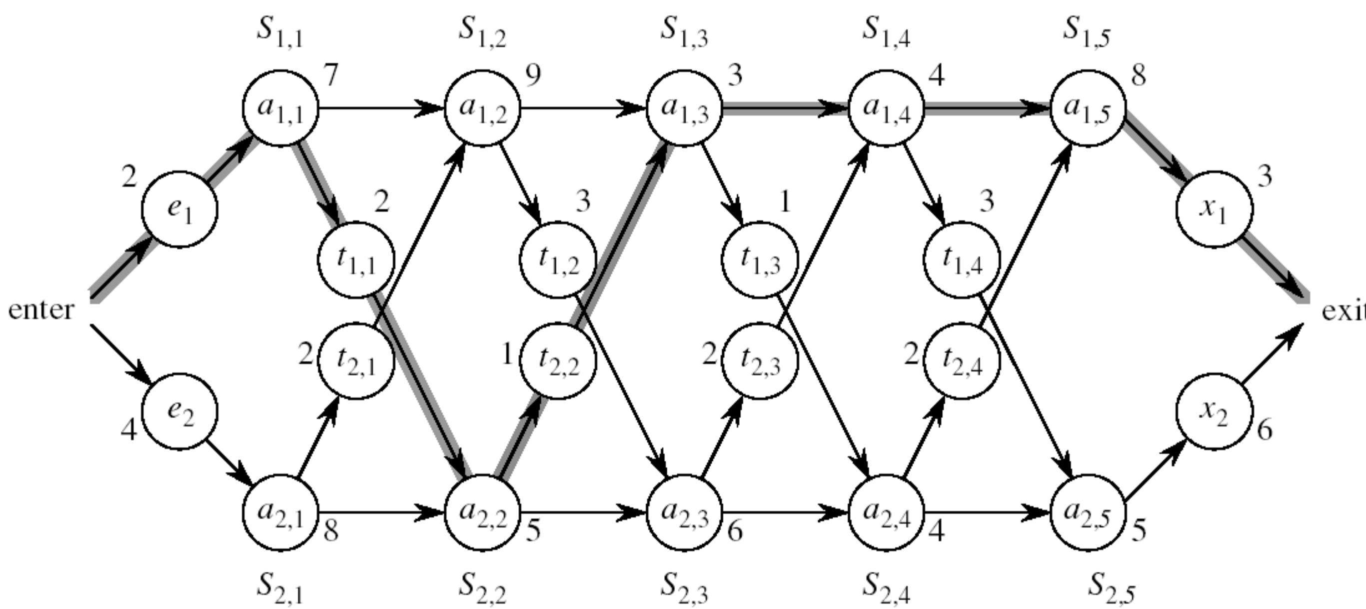
Assembly Line Scheduling

- Automobile factory with two assembly lines
  - Each line has n stations: $S_{1,1}, \ldots, S_{1,n}$ and $S_{2,1}, \ldots, S_{2,n}$
  - Corresponding stations $S_{1,j}$ and $S_{2,j}$ perform the same function but can take different amounts of time $a_{1,j}$ and $a_{2,j}$
  - Times to enter are e_1 and e_2 and times to exit are x_1 and x_2



Assembly Line

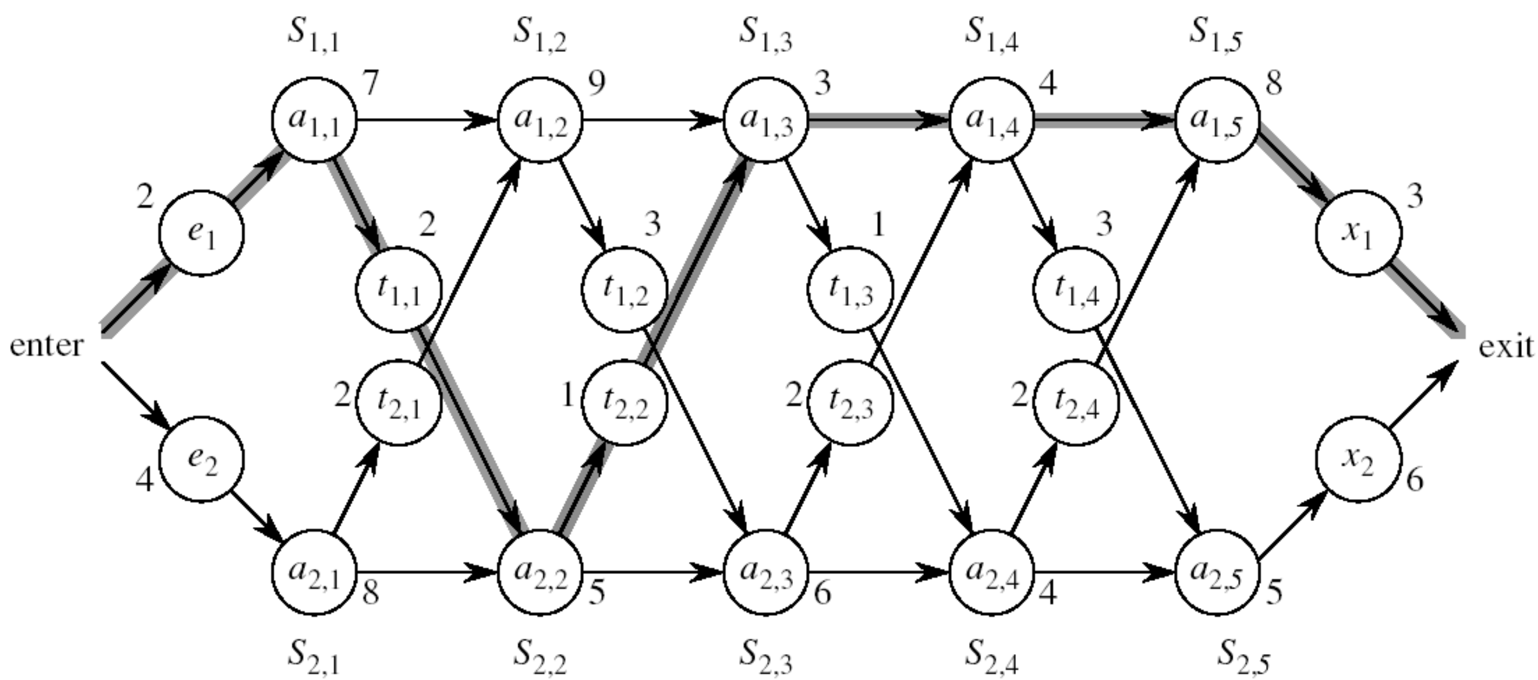
- After going through a station, the car can either:
  - stay on same line at no cost, or
  - transfer to other line: cost after $S_{i,j}$ is $t_{i,j}$, i = 1, 2, j = 1, . . . , n-1



Assembly Line Scheduling

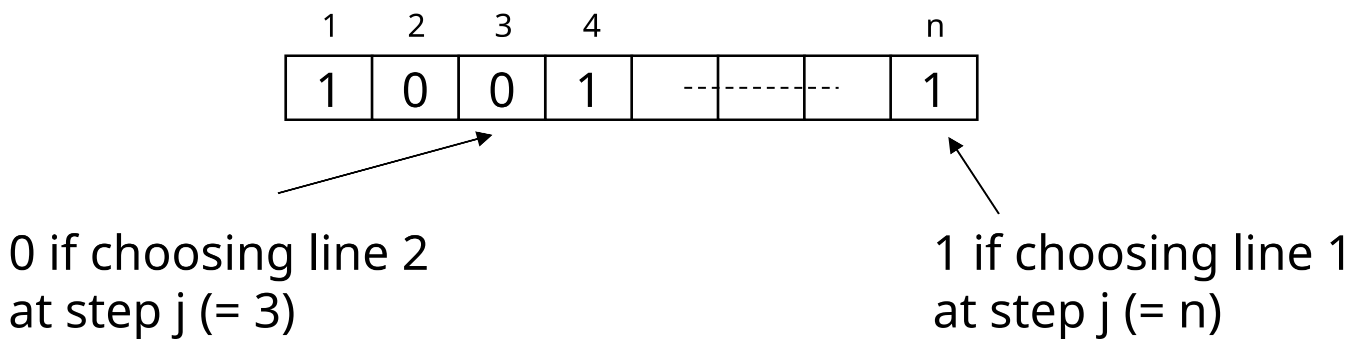
- Problem:

  What stations should be chosen from line 1 and what from line 2 in order to minimize the total time through the factory for one car?



One Solution

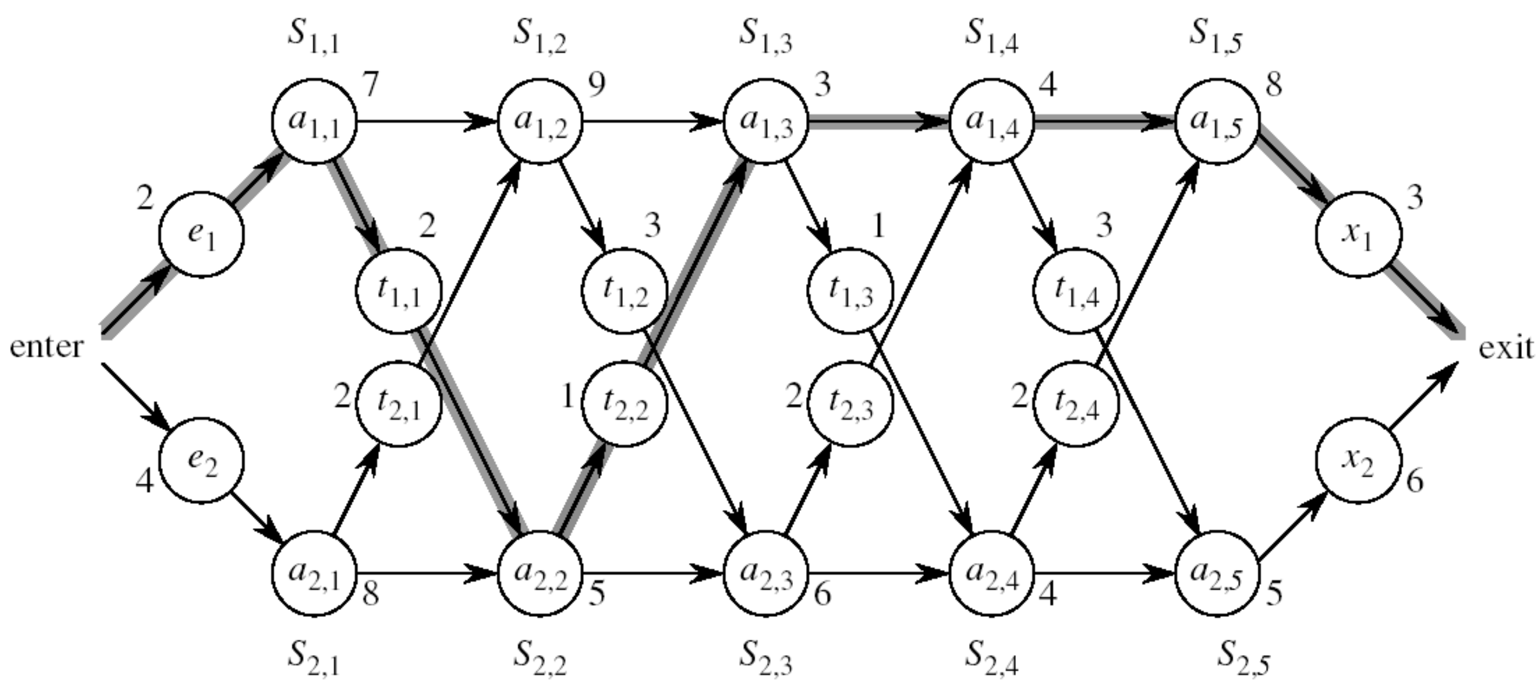
- Brute force
  - Enumerate all possibilities of selecting stations
  - Compute how long it takes in each case and choose the best one



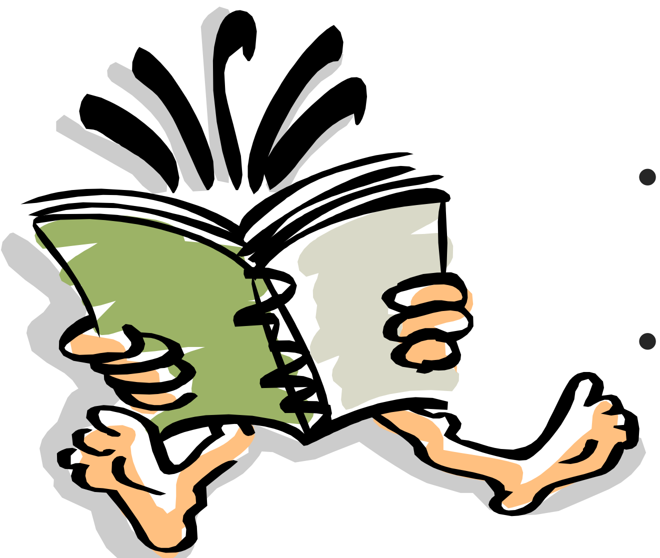
  - There are 2^n possible ways to choose stations
  - Infeasible when n is large

1. Structure of the Optimal Solution

- How do we compute the minimum time of going through the station?



Readings



- For this lecture
  - Chapter 17, 14
- Coming next
  - Sections 14.2-14.4