# CS420: Analysis of algorithms

Ewan Davies

Colorado State University

Spring 2026

# Data Structures

# Why data structures?

- We now shift from algorithms to data structures, abstract data types, and rigorous time analysis
- A medium-term goal is to develop data structures that support fast algorithms for problems like bipartite matching and max-flow
- We start with simple structures and build toward more sophisticated trees
- Background references:
  - [CLRS09, Chapter 10] for basic structures
  - [Tarj83, Chapters 3–5] and [CLRS09, Chapters 12–13] for search trees

# Abstract data types

- An **abstract data type** (ADT) specifies the operations we want without committing to an implementation
- We care about:
  - what operations are supported
  - what guarantees each implementation can achieve
  - how the cost depends on the current size $n$
- The same ADT can have very different running times under different implementations

# A basic list ADT

# List operations

| Operation | Meaning |
|---|---|
| `Peek(L, i)` | Return the element at index *i* |
| `Prepend(L, x)` | Insert *x* at the start of the list |
| `Append(L, x)` | Insert *x* at the end of the list |

- We could add more operations such as `Delete`, `Pop`, or `Insert`
- We use *n* for the current number of items in the list

# Singly linked lists

- A singly linked list is a path through memory:
  - we keep a pointer to the head
  - each item points to the next item
  - the last item points to `nil`
- Standard costs for a list of length $n$:
  - `Prepend` is $\Theta(1)$
  - `Peek` is $\Theta(n)$
  - `Append` is $\Theta(n)$ unless we also store a tail pointer
- With both head and tail pointers, `Append` becomes $\Theta(1)$

# Dynamic arrays

- Another list implementation uses a contiguous block of memory
- Most appends are cheap: place the new item in the next free slot
- Occasionally an append is expensive:
  - allocate a larger block (we assume this takes $\Theta(1)$ time)
  - copy all old items (this is $\Theta(n)$)
  - then append the new item $\Theta(1)$
- This is the first place where **amortized analysis** becomes useful

# Amortized analysis via aggregate analysis

- The worst-case cost an operation may be large, but the average over a long sequence can still be small
- If the $i$th append has cost $c_i$, then the amortized cost over $N$ appends is

$$\frac{1}{N}\sum_{i=1}^{N} c_i$$

- This is **aggregate analysis**: add the total cost, then divide by the number of operations

# Dynamic arrays with additive growth

- Suppose the array capacity starts at $g$
- When we need more space, grow from $n$ to $n + g$ elements
- Then among $N$ appends there are about $\lfloor N/g \rfloor$ expensive ones
- The expensive appends have costs on the order of $g, 2g, 3g, \ldots$
- The amortized cost is

$$\frac{1}{N}\left(\sum_{i=1}^{\lfloor N/g \rfloor} \Theta(ig) + O(N)\right) = \Theta(N)$$

- Additive growth leads to slow appends

# Dynamic arrays with multiplicative growth

- Now suppose we grow capacity by $n \mapsto \lceil gn \rceil$ for some fixed $g > 1$
- Expensive appends happen only about $\Theta(\log_g N)$ times during $N$ appends
- The expensive appends cost on the order of $g, g^2, g^3, \ldots$
- The amortized cost is

$$\frac{1}{N}\left(\sum_{i=1}^{\log_g N} \Theta(g^i) + O(N)\right) = O(1)$$

- Since each append costs at least $\Omega(1)$ anyway, the amortized cost is $\Theta(1)$

# Takeaway from dynamic arrays

- `Append` on a dynamic array is:
  - $O(n)$ in the worst case
  - $O(1)$ amortized under multiplicative resizing
- `Peek` is naturally $O(1)$ in the worst case
- `Prepend` is expensive because every item may need to move
- The point is not just the implementation, but the style of reasoning: worst-case versus amortized guarantees

# Binary search trees

# Search trees

- **Search trees** are a family of data structures that allow fast insertion, deletion, and search
- We assume our trees are **full binary trees**: nodes have 0 or 2 children
- We call nodes with 2 children **internal** and nodes with 0 children **external**
- See CLRS Section 12 and Tarjan Chap. 4
- Linked lists and dynamic arrays give us $\Omega(n)$ worst-case time complexity for at least one of these operations, and this can be avoided by using trees
- We start by defining an abstract data type for a **set ordered by key**

# Ordered sets ADT

- Consider the problem of maintaining one or more sets of objects, each with a distinct key from a totally ordered universe

| Operation | Meaning |
|---|---|
| `Key(x)` | Return the key of an element `x` |
| `Access(S, k)` | Return the element in set `S` with key `k` (null if it doesn't exist) |
| `Insert(S, x)` | Insert element `x` into set `S` |
| `Delete(S, x)` | Delete element `x` from set `S` |

# Ordered sets ADT

| Operation | Meaning |
|---|---|
| `Join(S1, y, S2)` | Return the ordered set representing the union `S1` ∪ `{y}` ∪ `S2` under the assumption that `Key(x) <= Key(y) <= Key(z)` for all `x` in `S1` and `z` in `S2` |
| `Split(y, S)` | Return `(S1, S2)` where `S1` is the subset of `S` of elements with keys `< Key(y)` and `S2` is the subset of `S` with keys `> Key(y)` |

# Binary search trees

- A **binary search tree** stores one key per internal node
- It satisfies the **symmetric-order property**: for every internal node with key $k$
  - every key in the left subtree is smaller than $k$
  - every key in the right subtree is larger than $k$
- Lookup behaves like binary search, except on a tree instead of an array
- Typical node fields:
  - parent pointer
  - left child pointer
  - right child pointer

# Binary search tree example

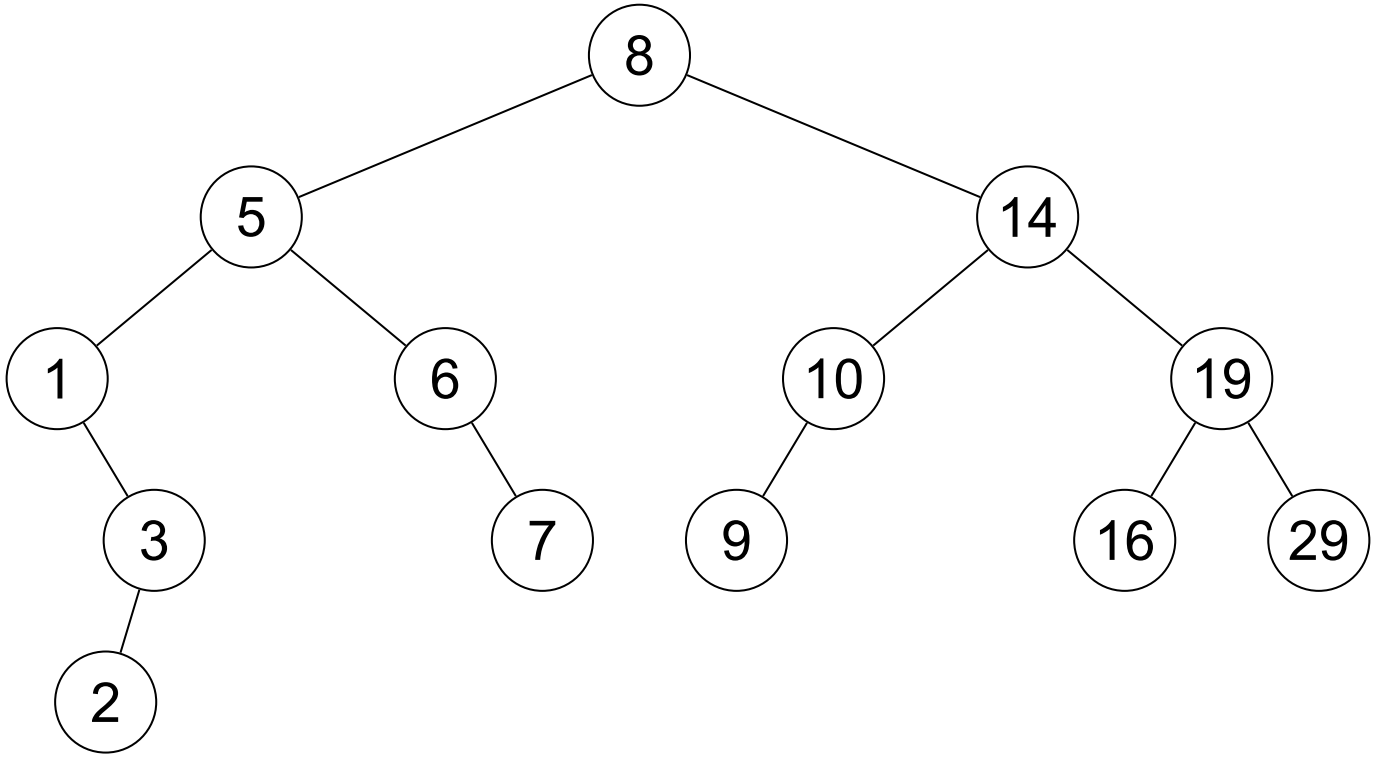


Figure 1: A small binary search tree

# Binary search tree example

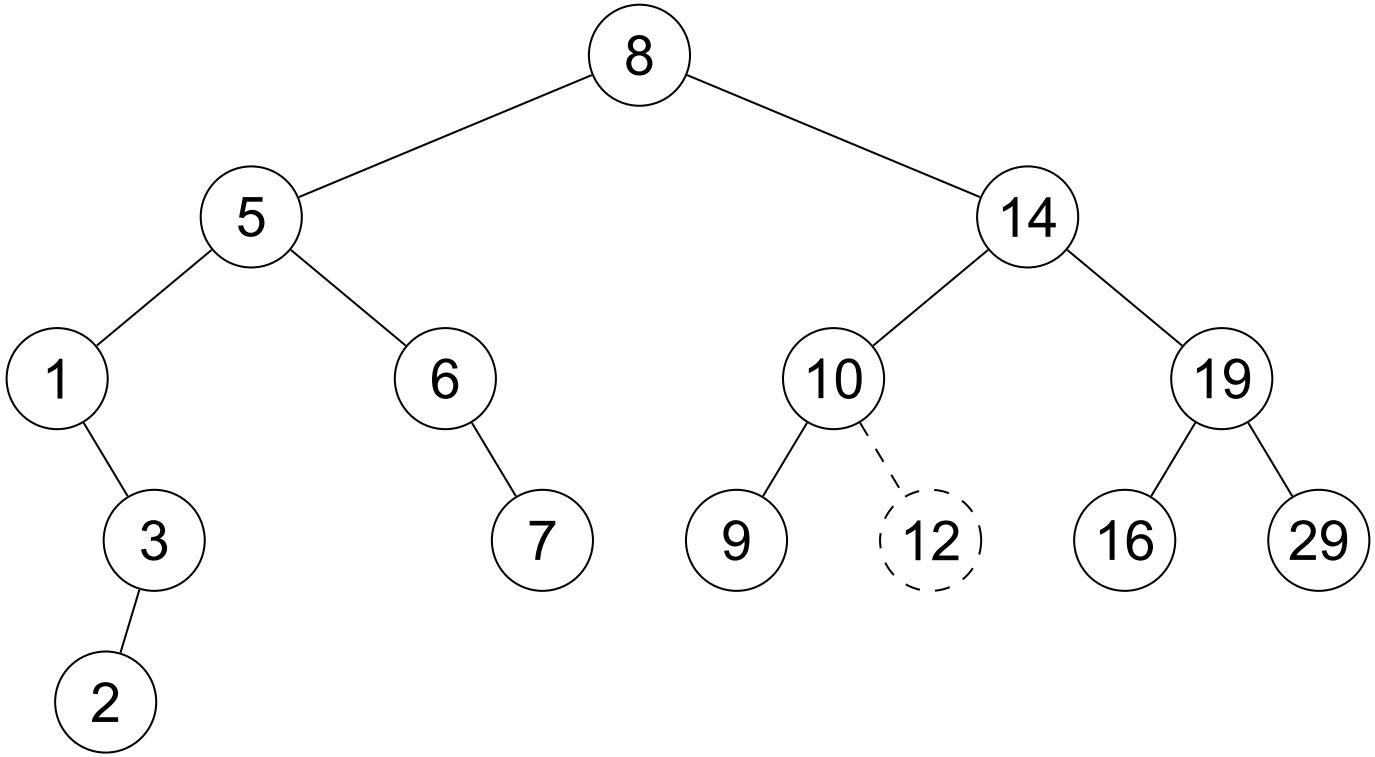


Figure 2: Insertion of a new key

# Binary search tree example

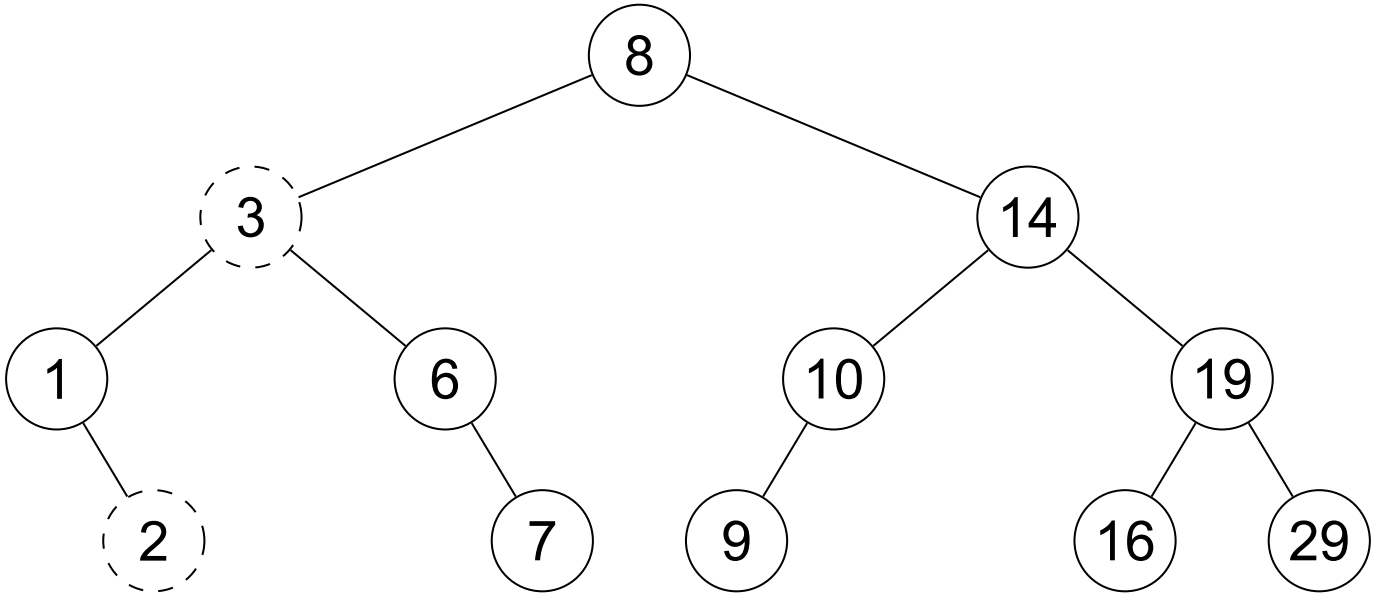


Figure 3: Deletion of a key: replace the deleted key with its **predecessor** (the largest key in its left subtree) to preserve symmetric order

# Binary search tree operations

- Lookup: descend from the root comparing the target with the current key
- Insert: follow the lookup path to a null leaf, then add a new node there
- Delete is more subtle:
  - if a node has a null child, replace it by its non-null child or by null
  - otherwise replace it by its predecessor
  - you need to **recursively** replace predecessors
- The running time is proportional to the (maximum) depth reached during the operation

# Why balance matters

- If a binary search tree with $n$ nodes has the best-case depth $O(\log n)$, then lookup, insert, and delete are fast in the worst case
- If the tree degenerates into a path-like shape, the depth becomes $\Omega(n)$ and the operations are all slow
- The major challenge is to keep the order property while preventing the tree from becoming too deep
- We will see a few ways of meeting this challenge

# Red-black trees

# Red-black trees

- A **red-black tree** is a binary search tree augmented with a particular balancing discipline
- We study a formulation based on a **rank function** with constraints on parents, grandparents, and leaves
- Write $R(x)$ for the rank and $P(x)$ for the parent of node $x$

# Red-black rank properties

- The **red-black rank properties** are
  - (i) If $x$ is any node with a parent then $R(x) \leq R(P(x)) \leq R(x) + 1$
  - (ii) If $x$ is any node with a grandparent then $R(x) < R(P(P(x)))$
  - (iii) If $x$ is external then $R(x) = 0$ and if $x$ also has a parent then $R(P(x)) = 1$
- We will see how these constraints force logarithmic depth
- These properties are easy to reason about as they are **local**

# Red-black color properies

- An alternative formulation uses **colors** with the following properties

(a) every node is red or black

(b) external nodes are black and red nodes have two black children

(c) for all nodes $x$, every path from $x$ to an external node has the same number of black nodes

- Intuitively these force some kind of balance
- It is not obvious that you can easily decide the colors of nodes
- There can be some advantages to the rank formulation
- Property (c) is hard to reason about because it is **global**

# Equivalence

- In a homework assignment you will prove that the formulations are equivalent
- Given a valid rank function we let a node be
  - **black** if it has no parent or its parent has greater rank
  - **red** otherwise
- We will move forward using the most convenient formulation where we need it

# Red-black tree example

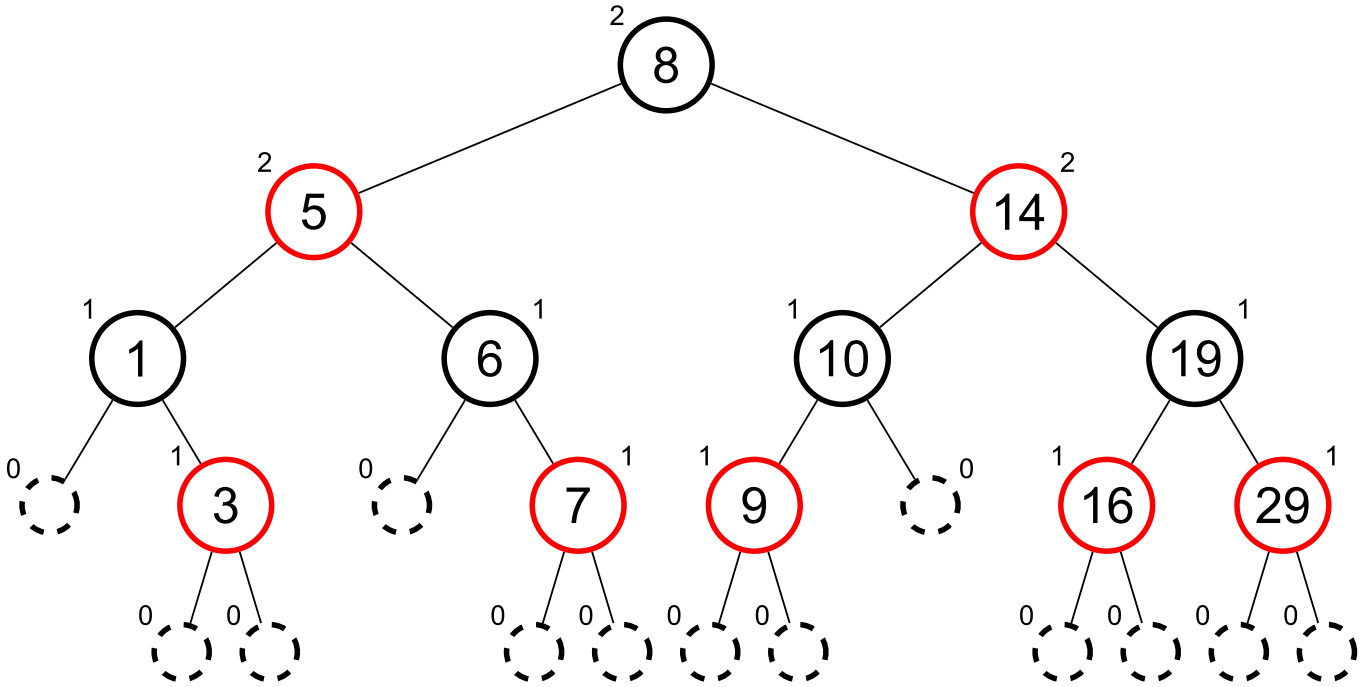


Figure 4: A small red-black tree

# Red-black tree height bound

**Lemma:** In a red-black tree

(a) A node of rank $k$ has height at most $2k$

(b) A subtree rooted at a node of rank $k$ has at least $2^{k+1} - 1$ nodes (including external)

**Corollary:** A red-black tree with $n$ internal nodes has depth at most

$$2\lfloor \log_2(n+1) \rfloor$$

and so insert/lookup/delete have worst-case time complexity $O(\log n)$

# Proof of corollary

- Let the depth of the tree be $d$, the rank of the root be $r$, and suppose that we have $n$ *internal* nodes.
- Then $d \leq 2r$ by (a) and the tree has at least $N = 2^{r+1} - 1$ nodes including external
- A binary tree of $N$ nodes cannot have more than $(N + 1)/2$ leaves, so

$$n \geq \frac{N - 1}{2} \geq 2^r - 1 \geq 2^{d/2} - 1.$$

- Then $d \leq 2 \log_2(n + 1)$ as required. □

# Proof of lemma

- We proceed by induction on the rank $k$.
- Base case:
  - By (i)–(iii) nodes have rank zero if and only if they are external
  - External nodes have no descendants, so their height is zero and their subtree contains $2^0 - 1 = 1$ nodes as required.
- Induction step:
  - Let $x$ be a node of rank $k \geq 1$, and let $y$ and $z$ be the children of $x$.
  - $y, z$ have rank $k$ or $k - 1$ by (i)

# Proof of lemma

- If a child of $x$ has rank $k$, its children have rank $k - 1$ by (ii) and we can apply the induction hypothesis
- If a child of $x$ has rank $k - 1$, we can apply the induction hypothesis there
- For height, the worst case is that $x$ has grandchildren of rank $k - 1$ to which we apply the induction hypothesis, so these grandchildren have height at most $2k - 2$
- But then x has height at most $2k$ as required
- For size of subtree, the worst case is that the children of x have rank $k - 1$
- Then the subtree rooted at $x$ has at least $1 + 2(2^k - 1) = 2^{k+1} - 1$ nodes. □

# Rotations

- The tool for rebalancing binary trees is **rotation**
- Rotations preserve symmetric order while changing the shape of the tree
- For red-black trees we also use rank adjustment known as **promotion** and **demotion**
- Insertion and deletion may trigger a small sequence of local repairs:
  - single rotation
  - double rotation (two carefully chosen single rotations)
  - rank adjustment (equivalently, color changes)

# Rotations

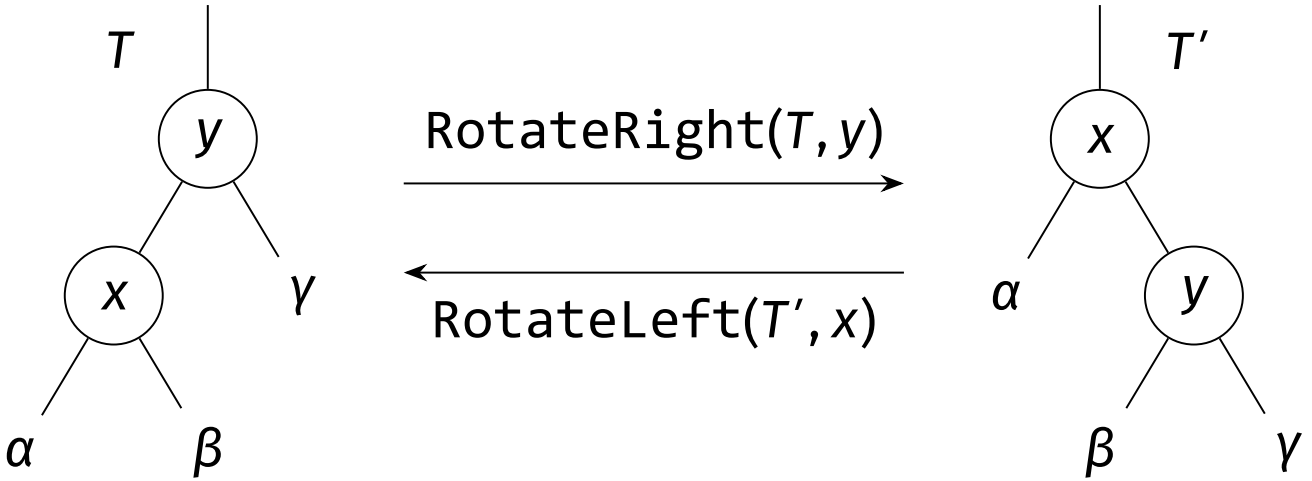


Figure 5: Single rotation

# Insertion in red-black trees

- The effect of insertion is to replace a node of rank 0 with a node of rank 1
- The only problem this can cause is a chain of three nodes of equal rank, say $x-y-z$ where z is the “current node” and is farthest from the root
- This violates (ii) (or that red nodes have two black children)
- To handle this we try to promote the top of the chain
- If the top of the chain $x$ has two red children then we can promote $x$ and look for chains with $x$ at the bottom now
- If $x$ has a black child then it cannot be promoted without violating (i), and a single or double rotation completes the rebalancing

# Insertion in red-black trees

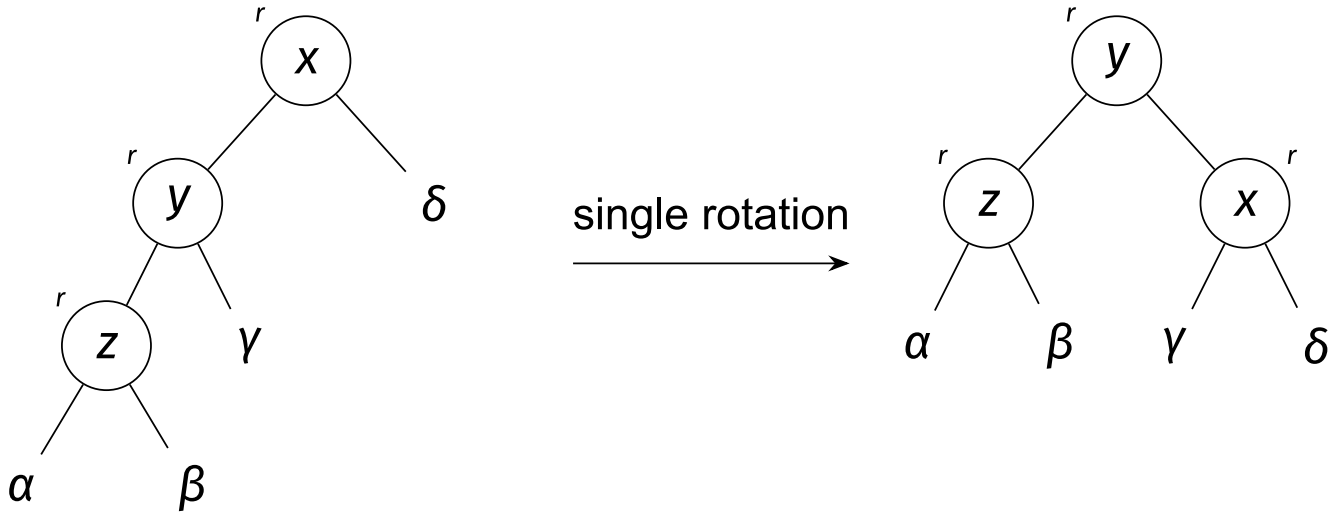


Figure 6: Single rotation: when z is the left-child of its parent.

# Insertion in red-black trees

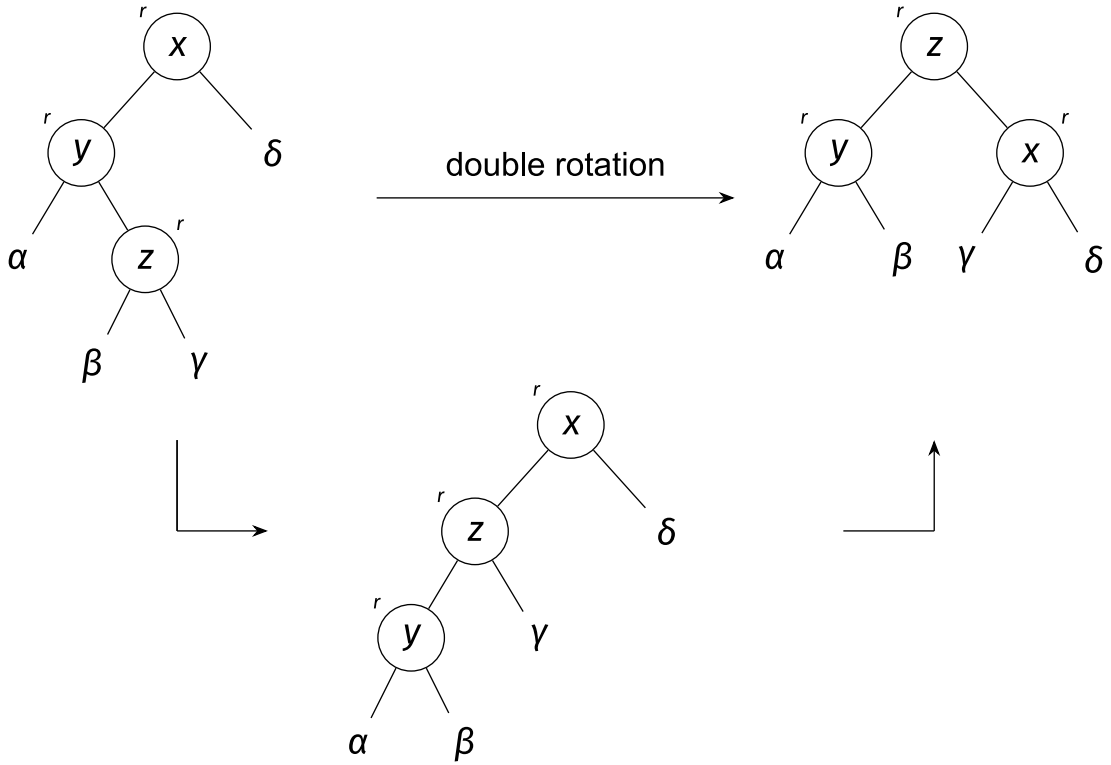


Figure 7: Double rotation: when z is the right-child of its parent. Start with a left-rotation at y then perform a right-rotation at x.

# Exploring examples

# Deletion in red-black trees

- Deletion replaces some node in the tree with one, say $x$, of lower rank
- This means we are potentially violating (i), and we can correct this as follows:

(1) The sibling $y$ of $x$ is black.

(a) Both children of $y$ are black, then it is allowed to demote the parent of $x$ and check for violations of (i) at the parent.

(b) The child of $y$ farthest from xis red. This is fixed by a single rotation at the parent of $x$, and we can now stop.

(c) The child of $y$ nearest to $x$ is red and its sibling is black. This is fixed by a double rotation at the parent of $x$, and we can now stop.

# Deletion in red-black trees

(2) The sibling $y$ of $x$ is red. Then it must be that both children of $y$ are black, and hence after a single rotation at the parent of $x$ we are in case (1). Note that when applying case (1) in this situation, subcase (1a) cannot cause a new violation and so all subcases are terminal.

- There will be a homework exercise figuring out the details: see [Tarj83, Figure 4.8] and [CLRS09, Section 13.4] for guidance

# The time complexity of rebalancing

- Analyzing the rebalancing steps and using the height bounds, we know that insertion and deletion take time $O(\log n)$ in the worst case for red-black trees, even taking into account the cost of any rebalancing that may occur
- This is great, but the implementation of such trees is a little complex and the storage of the bits indicating color or rank needs to be taken into account

# Red-black tree performance

- Red-black trees guarantee:
  - lookup in worst-case $O(\log n)$
  - insert in worst-case $O(\log n)$
  - delete in worst-case $O(\log n)$
- This is excellent asymptotically
- The tradeoff is implementation complexity:
  - extra metadata per node
  - more cases to handle correctly

# References

[CLRS09] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2009). *Introduction to algorithms*. MIT Press, 3rd ed. Catalog link. Library link.

[Tarj83] Tarjan, R. E. (1983). *Data structures and network algorithms*. CBMS-NSF regional conference series in applied mathematics, Society for Industrial and Applied Mathematics. Catalog link.