Amortized Analysis

Princeton University • COS 423 • Theory of Algorithms • Spring 2001 • Kevin Wayne

Beyond Worst Case Analysis

Worst-case analysis.

- Analyze running time as function of worst input of a given size.

Average case analysis.

- Analyze average running time over some distribution of inputs.
- Ex: quicksort.

Amortized analysis.

- Worst-case bound on sequence of operations.
- Ex: splay trees, union-find.

Competitive analysis.

- Make quantitative statements about online algorithms.
- Ex: paging, load balancing.

Amortized Analysis

Amortized analysis.

- Worst-case bound on sequence of operations.
  - no probability involved
- Ex: union-find.
  - sequence of m union and find operations starting with n singleton sets takes $O((m+n)\ \alpha(n))$ time.
  - single union or find operation might be expensive, but only $\alpha(n)$ on average

Dynamic Table

Dynamic tables.

- Store items in a table (e.g., for open-address hash table, heap).
- Items are inserted and deleted.
  - too many items inserted $\Rightarrow$ copy all items to larger table
  - too many items deleted $\Rightarrow$ copy all items to smaller table

Amortized analysis.

- Any sequence of n insert / delete operations take O(n) time.
- Space used is proportional to space required.
- Note: actual cost of a single insert / delete can be proportional to n if it triggers a table expansion or contraction.

Bottleneck operation.

- We count insertions (or re-insertions) and deletions.
- Overhead of memory management is dominated by (or proportional to) cost of transferring items.

Dynamic Table: Insert

Dynamic Table Insert

```
Initialize table size m = 1.

INSERT(x)
   IF (number of elements in table = m)
      Generate new table of size 2m.
      Re-insert m old elements into new table.
      m ← 2m

   Insert x into table.
```

Aggregate method.

- Sequence of n insert ops takes O(n) time.
- Let c_i = cost of i^{th} insert.

$$c_i = \begin{cases} i & \text{if } i - 1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

$$\begin{aligned} \sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\log_2 n} 2^j \\ &= n + (2n - 1) \\ &< 3n \end{aligned}$$

Dynamic Table: Insert

Accounting method.

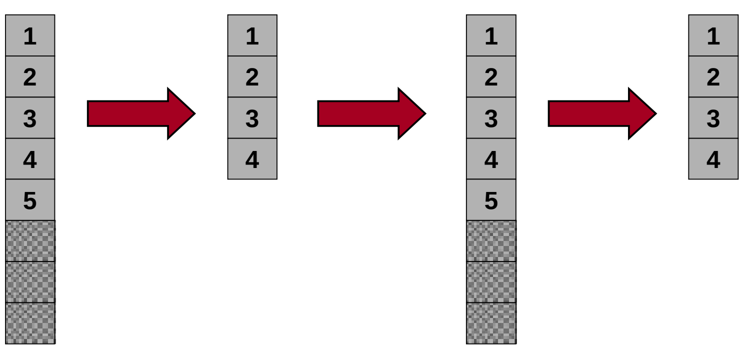
- Charge each insert operation \$3 (amortized cost).
  - use \$1 to perform immediate insert
  - store \$2 in with new item
- When table doubles:
  - \$1 re-inserts item
  - \$1 re-inserts another old item

Dynamic Table: Insert and Delete

Insert and delete.

- Table overflows ⇒ double table size.
- Table ≤ ½ full ⇒ halve table size.

✎ Bad idea: can cause thrashing.



Dynamic Table: Insert and Delete

Insert and delete.

- Table overflows ⇒ double table size.
- Table ≤ ¼ full ⇒ halve table size.

Dynamic Table Delete

```
Initialize table size m = 1.

DELETE(x)
   IF (number of elements in table ≤ m / 4)
      Generate new table of size m / 2.
      m ← m / 2
      Reinsert old elements into new table.

   Delete x from table.
```

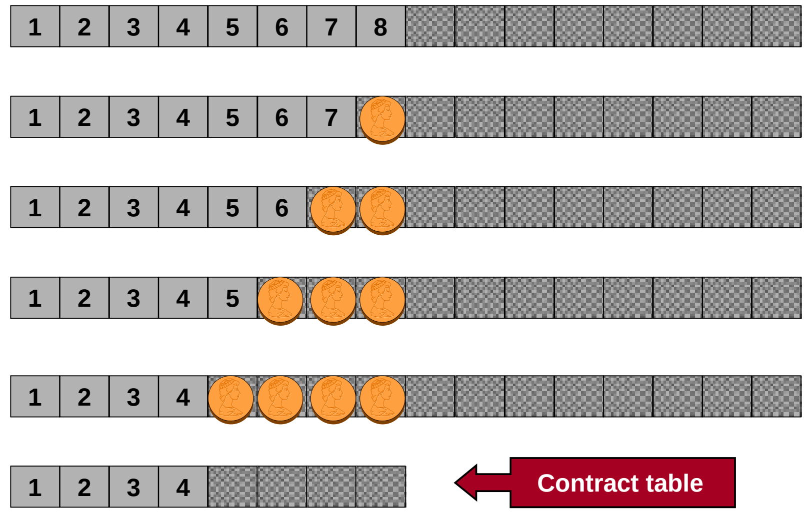
Dynamic Table: Insert and Delete

Accounting analysis.

- Charge each insert operation \$3 (amortized cost).
  - use \$1 to perform immediate insert
  - store \$2 with new item
- When table doubles:
  - \$1 re-inserts item
  - \$1 re-inserts another old item

- Charge each delete operation \$2 (amortized cost).
  - use \$1 to perform delete
  - store \$1 in emptied slot
- When table halves:
  - \$1 in emptied slot pays to re-insert a remaining item into new half-size table

Dynamic Table: Delete



Dynamic Table: Insert and Delete

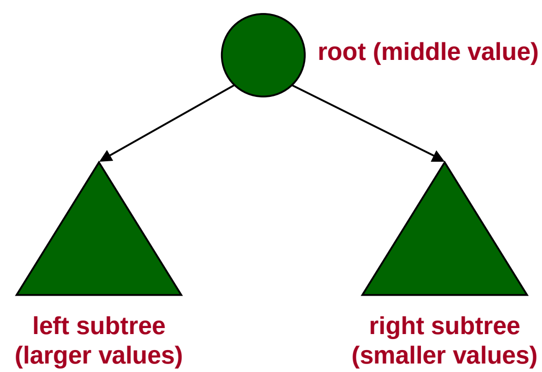
Theorem. Sequence of n inserts and deletes takes O(n) time.

- Amortized cost of insert = \$3.
- Amortized cost of delete = \$2.

Binary Search Tree

Binary tree in "sorted" order.

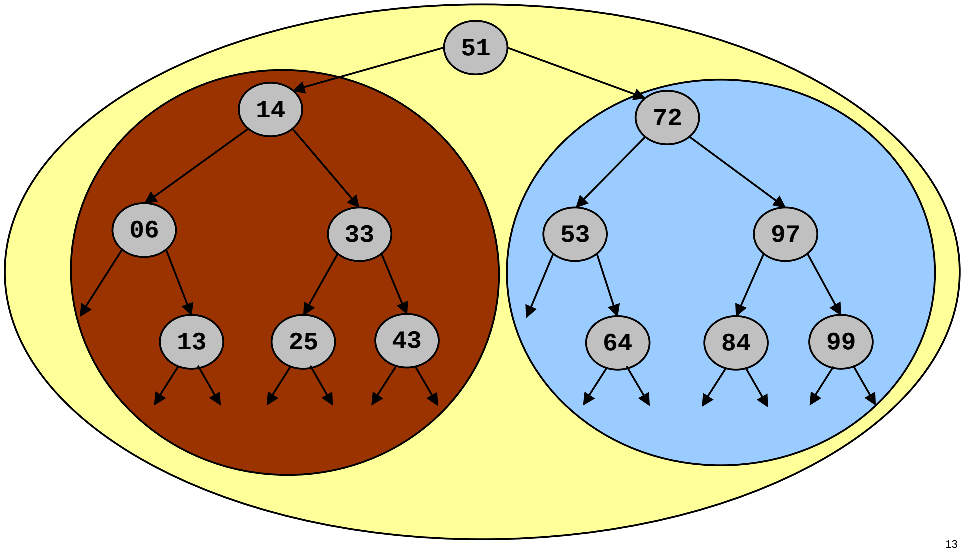
- Maintain ordering property for ALL sub-trees.



Binary Search Tree

Binary tree in "sorted" order.

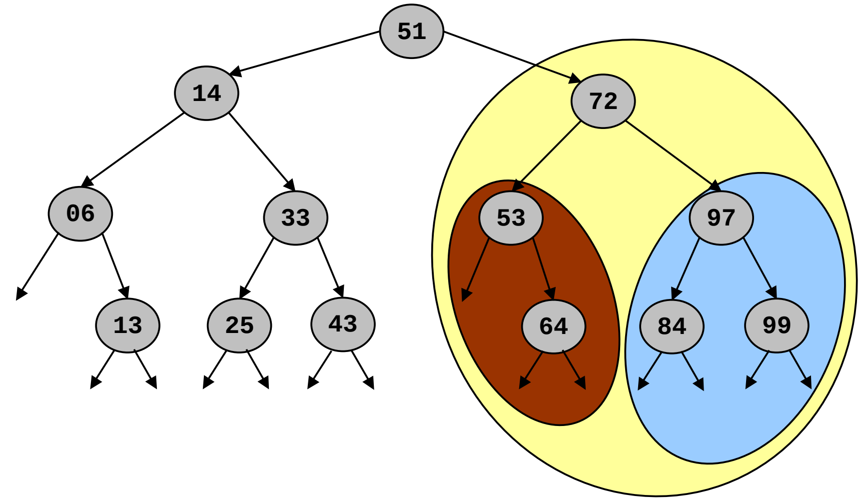
- Maintain ordering property for ALL sub-trees.



Binary Search Tree

Binary tree in "sorted" order.

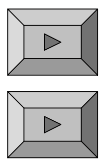
- Maintain ordering property for ALL sub-trees.



Binary Search Tree

Insert, delete, find (symbol table).

- Amount of work proportional to height of tree.
- O(N) in "unbalanced" search tree.
- O(log N) in "balanced" search tree.



Search

Insert

Types of BSTs.

- AVL trees, 2-3-4 trees, red-black trees.
- Treaps, skip lists, splay trees.

BST vs. hash tables.

- Guaranteed vs. expected performance.
- Growing and shrinking.
- Augmented data structures: order statistic trees, interval trees.

Splay Trees

Splay trees (Sleator-Tarjan, 1983a). Self-adjusting BST.

- Most frequently accessed items are close to root.
- Tree automatically reorganizes itself after each operation.
  - no balance information is explicitly maintained
- Tree remains "nicely" balanced, but height can potentially be n - 1.
- Sequence of m ops involving n inserts takes O(m log n) time.

Theorem (Sleator-Tarjan, 1983a). Splay trees are as efficient (in amortized sense) as static optimal BST.

Theorem (Sleator-Tarjan, 1983b). Shortest augmenting path algorithm for max flow can be implemented in O(mn log n) time.

- Sequence of mn augmentations takes O(mn log n) time!
- Splay trees used to implement dynamic trees (link-cut trees).

Splay

Find(x, S): Determine whether element x is in splay tree S.
Insert(x, S): Insert x into S if it is not already there.
Delete(x, S): Delete x from S if it is there.
Join(S, S'): Join S and S' into a single splay tree, assuming that $x < y$ for all $x \in S$, and $y \in S'$.

All operations are implemented in terms of basic operation:

Splay(x, S): Reorganize splay tree S so that element x is at the root if $x \in S$; otherwise the new root is either $\max \{ k \in S : k < x \}$ or $\min \{ k \in S : k > x \}$.

Implementing Find(x, S).

- Call Splay(x, S).
- If x is root, then return x; otherwise return NO.

Splay

Implementing Join(S, S').

- Call Splay(+∞, S) so that largest element of S is at root and all other elements are in left subtree.
- Make S' the right subtree of the root of S.

Implementing Delete(x, S).

- Call Splay(x, S) to bring x to the root if it is there.
- Remove x: let S' and S'' be the resulting subtrees.
- Call Join(S', S'').

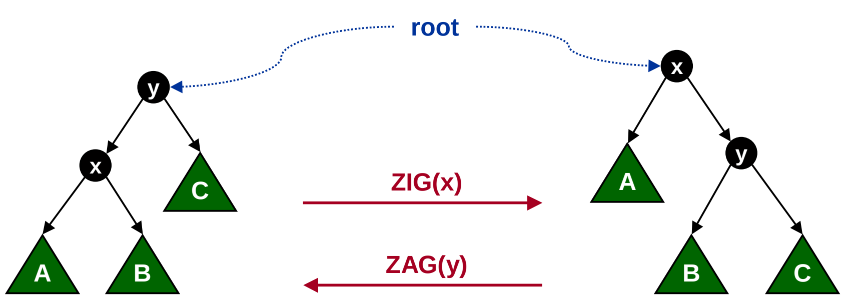
Implementing Insert(x, S).

- Call Splay(x, S) and break tree at root to form S' and S''.
- Call Join(Join(S', {x}), S'').

Implementing Splay(x, S)

Splay(x, S): do following operations until x is root.

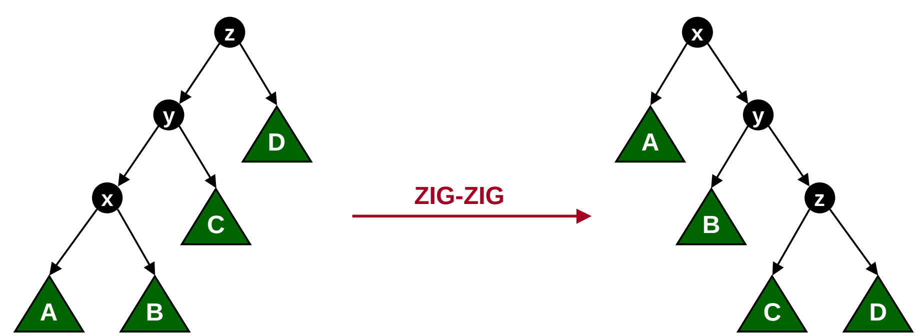
- ZIG: If x has a parent but no grandparent, then rotate(x).
- ZIG-ZIG: If x has a parent y and a grandparent, and if both x and y are either both left children or both right children.
- ZIG-ZAG: If x has a parent y and a grandparent, and if one of x, y is a left child and the other is a right child.



Implementing Splay(x, S)

Splay(x, S): do following operations until x is root.

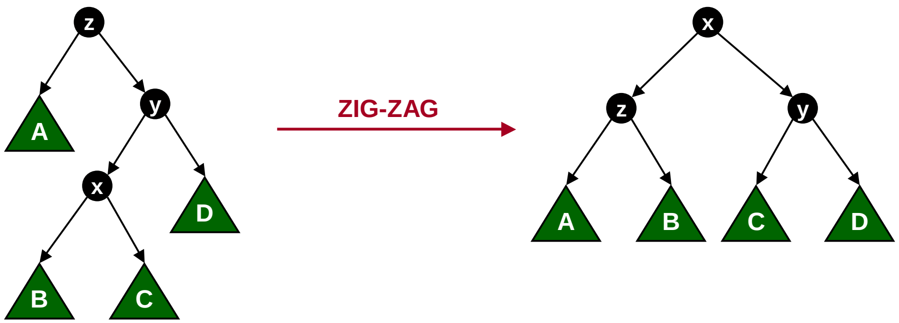
- ZIG: If x has a parent but no grandparent.
- ZIG-ZIG: If x has a parent y and a grandparent, and if both x and y are either both left children or both right children.
- ZIG-ZAG: If x has a parent y and a grandparent, and if one of x, y is a left child and the other is a right child.



Implementing Splay(x, S)

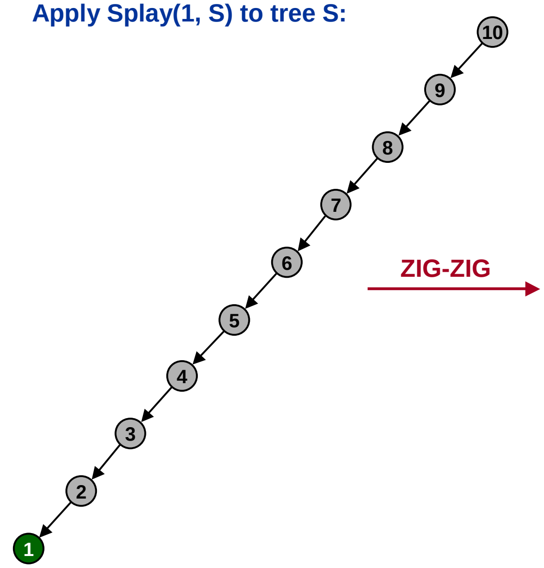
Splay(x, S): do following operations until x is root.

- ZIG: If x has a parent but no grandparent.
- ZIG-ZIG: If x has a parent y and a grandparent, and if both x and y are either both left children or both right children.
- **ZIG-ZAG: If x has a parent y and a grandparent, and if one of x, y is a left child and the other is a right child.**



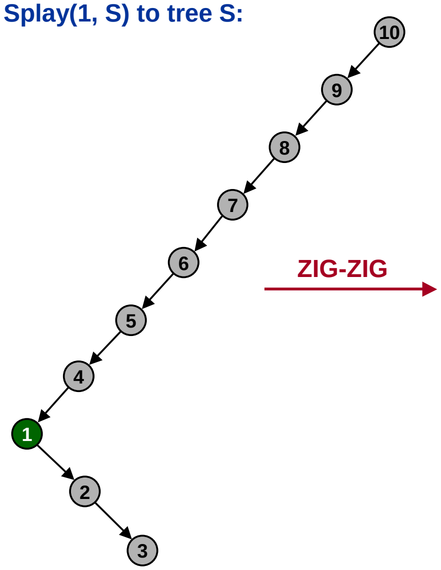
Splay Example

Apply Splay(1, S) to tree S:



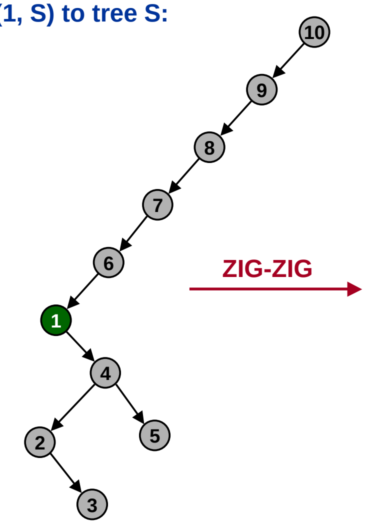
Splay Example

Apply Splay(1, S) to tree S:



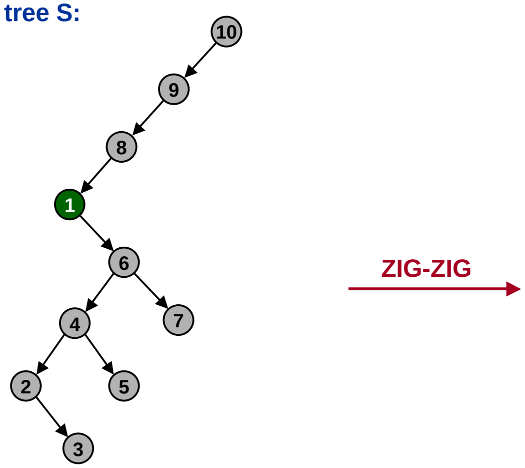
Splay Example

Apply Splay(1, S) to tree S:



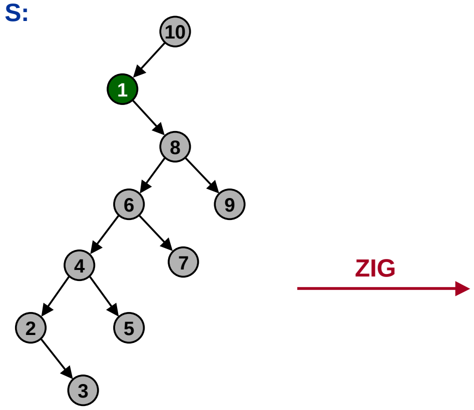
Splay Example

Apply Splay(1, S) to tree S:



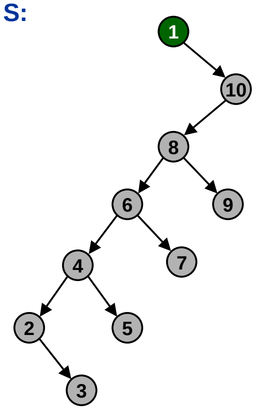
Splay Example

Apply Splay(1, S) to tree S:



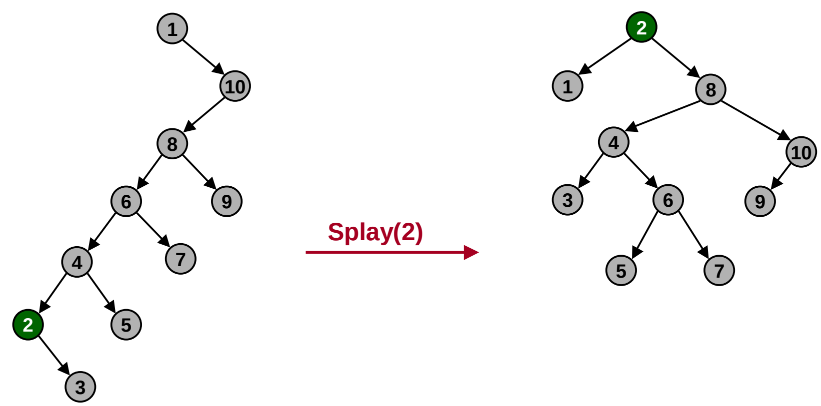
Splay Example

Apply Splay(1, S) to tree S:



Splay Example

Apply Splay(2, S) to tree S:

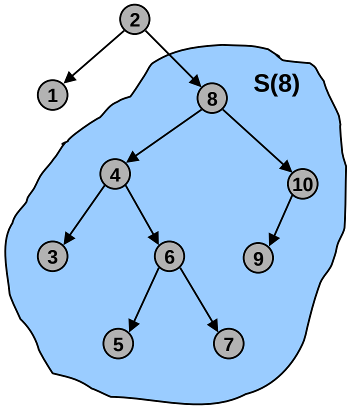


Splay Tree Analysis

Definitions.

- Let S(x) denote subtree of S rooted at x.
- |S| = number of nodes in tree S.
- $\mu(S)$ = rank = $\lfloor \log |S| \rfloor$.
- $\mu(x) = \mu(S(x))$.

|S| = 10
$\mu(2) = 3$
$\mu(8) = 3$
$\mu(4) = 2$
$\mu(6) = 1$
$\mu(5) = 0$



Splay Tree Analysis

Splay invariant: node x always has at least $\mu(x)$ credits on deposit.

Splay lemma: each splay(x, S) operation requires $\leq 3(\mu(S) - \mu(x)) + 1$ credits to perform the splay operation and maintain the invariant.

Theorem: A sequence of m operations involving n inserts takes O(m log n) time.

Proof:

- $\mu(x) \leq \lfloor \log n \rfloor \Rightarrow$ at most $3\lfloor \log n \rfloor + 1$ credits are needed for each splay operation.
- Find, insert, delete, join all take constant number of splays plus low-level operations (pointer manipulations, comparisons).
- Inserting x requires $\leq \lfloor \log n \rfloor$ credits to be deposited to maintain invariant for new node x.
- Joining two trees requires $\leq \lfloor \log n \rfloor$ credits to be deposited to maintain invariant for new root.

Splay Tree Analysis

Splay invariant: node x always has at least $\mu(x)$ credits on deposit.

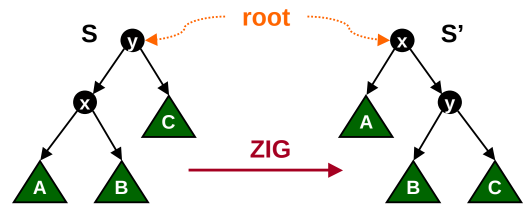
Splay lemma: each splay(x, S) operation requires $\leq 3(\mu(S) - \mu(x)) + 1$ credits to perform the splay operation and maintain the invariant.

Proof of splay lemma: Let $\mu(x)$ and $\mu'(x)$ be rank before and single ZIG, ZIG-ZIG, or ZIG-ZAG operation on tree S.

- We show invariant is maintained (after paying for low-level operations) using at most:
  - $3(\mu(S) - \mu(x)) + 1$ credits for each ZIG operation.
  - $3(\mu'(x) - \mu(x))$ credits for each ZIG-ZIG operation.
  - $3(\mu'(x) - \mu(x))$ credits for each ZIG-ZAG operation.
- Thus, if a sequence of of these are done to move x up the tree, we get a telescoping sum $\Rightarrow$ total credits $\leq 3(\mu(S) - \mu(x)) + 1$.

Splay Tree Analysis

Proof of splay lemma (ZIG): It takes $\leq 3(\mu(S) - \mu(x)) + 1$ credits to perform a ZIG operation and maintain the splay invariant.



- In order to maintain invariant, we must pay:

$$\begin{aligned} \mu'(x) + \mu'(y) - \mu(x) - \mu(y) &= \mu'(y) - \mu(x) \\ &\leq \mu'(x) - \mu(x) \\ &\leq 3(\mu'(x) - \mu(x)) \\ &= 3(\mu(S) - \mu(x)) \end{aligned}$$

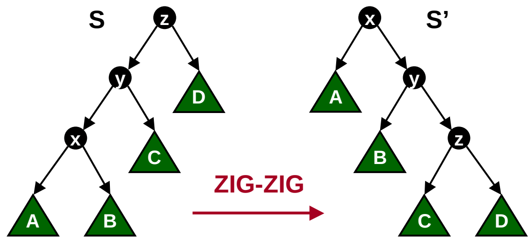
$\mu(y) = \mu'(x)$

$\mu'(x) = \mu(S)$

- Use extra credit to pay for low-level operations.

Splay Tree Analysis

Proof of splay lemma (ZIG-ZIG): It takes $\leq 3(\mu'(x) - \mu(x))$ credits to perform a ZIG-ZIG operation and maintain the splay invariant.



$$\begin{aligned}\mu'(x)+\mu'(y)+\mu'(z)-\mu(x)-\mu(y)-\mu(z) &= \mu'(y)+\mu'(z)-\mu(x)-\mu(y)\\ &= (\mu'(y)-\mu(x))+(\mu'(z)-\mu(y))\\ &\leq (\mu'(x)-\mu(x))+(\mu'(x)-\mu(x))\\ &= 2(\mu'(x)-\mu(x))\end{aligned}$$

- If $\mu'(x) > \mu(x)$, then can afford to pay for constant number of low-level operations and maintain invariant using $\leq 3(\mu'(x) - \mu(x))$ credits.

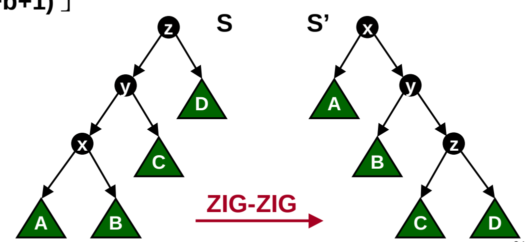
Splay Tree Analysis

Proof of splay lemma (ZIG-ZIG): It takes $\leq 3(\mu'(x) - \mu(x))$ credits to perform a ZIG-ZIG operation and maintain the splay invariant.

- Nasty case: $\mu(x) = \mu'(x)$.
- We show in this case $\mu'(x) + \mu'(y) + \mu'(z) < \mu(x) + \mu(y) + \mu(z)$.
  - don't need any credit to pay for invariant
  - 1 credit left to pay for low-level operations

  so, for contradiction, suppose $\mu'(x) + \mu'(y) + \mu'(z) \geq \mu(x) + \mu(y) + \mu(z)$.
- Since $\mu(x) = \mu'(x) = \mu(z)$, by monotonicity $\mu(x) = \mu(y) = \mu(z)$.
- After some algebra, it follows that $\mu(x) = \mu'(z) = \mu(z)$.
- Let $a = 1 + |A| + |B|$, $b = 1 + |C| + |D|$, then $\lfloor \log a \rfloor = \lfloor \log b \rfloor = \lfloor \log (a+b+1) \rfloor$
- WLOG assume $b \geq a$.

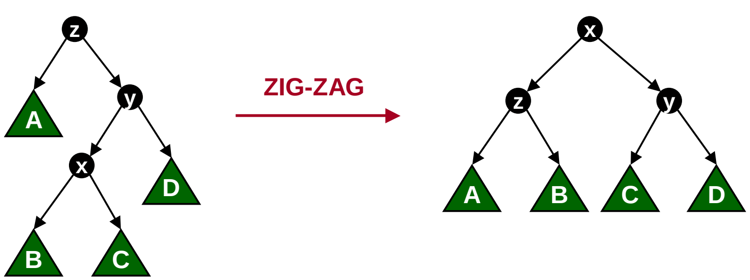
$$\begin{aligned}\lfloor \log(a+b+1) \rfloor &\geq \lfloor \log(2a) \rfloor\\ &= 1+\lfloor \log a \rfloor\\ &> \lfloor \log a \rfloor\end{aligned}$$



Splay Tree Analysis

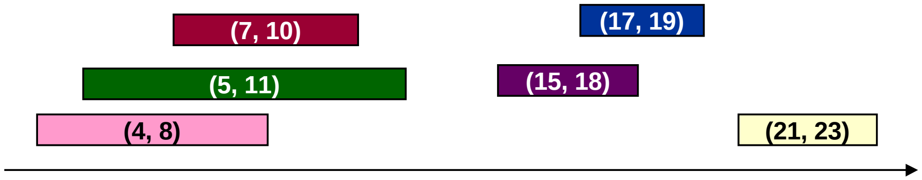
Proof of splay lemma (ZIG-ZAG): It takes $\leq 3(\mu'(x) - \mu(x))$ credits to perform a ZIG-ZAG operation and maintain the splay invariant.

- Argument similar to ZIG-ZIG.



Augmented Search Trees

Interval Trees



Support following operations.

Interval-Insert(i, S): Insert interval i = (ℓ_i, r_i) into tree S.

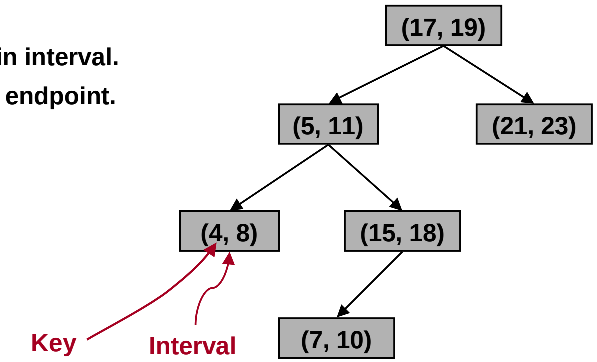
Interval-Delete(i, S): Delete interval i = (ℓ_i, r_i) from tree S.

Interval-Find(i, S): Return an interval x that overlaps i, or report that no such interval exists.

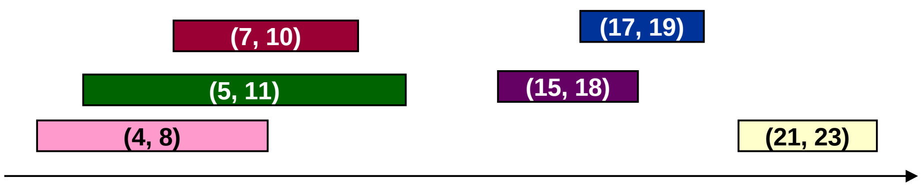
Interval Trees

Key ideas:

- Tree nodes contain interval.
- BST keyed on left endpoint.

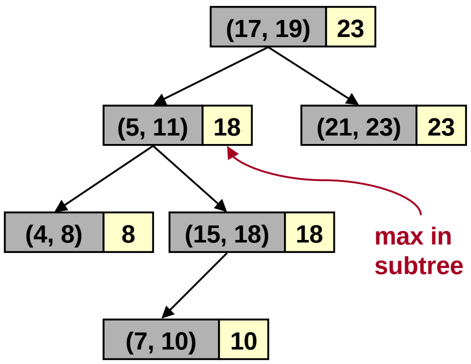


Interval Trees



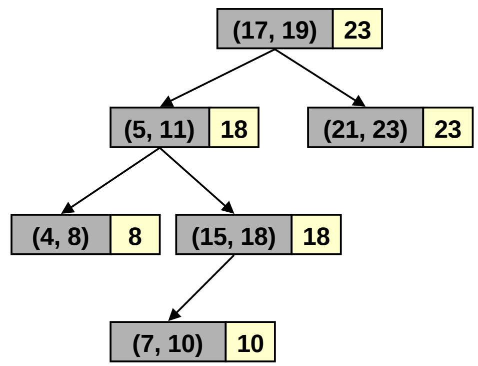
Key ideas:

- Tree nodes contain interval.
- BST keyed on left endpoint.
- Additional info: store max endpoint in subtree rooted at node.



Finding an Overlapping Interval

Interval-Find(i, S): return an interval x that overlaps i = (ℓ_i, r_i), or report that no such interval exists.



Interval-Find (i, S)

```
x ← root(S)

WHILE (x != NULL)
   IF (x overlaps i)
      RETURN t
   IF (left[x] = NULL OR
       max[left[x]] < ℓi)
      x ← right[x]
   ELSE
      x ← left[x]
RETURN NO

Splay last node on path
traversed.
```

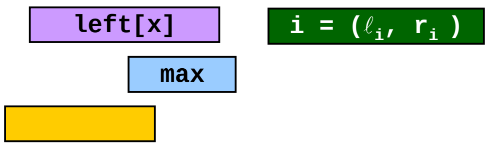
Finding an Overlapping Interval

Interval-Find(i, S): return an interval x that overlaps $i = (\ell_i, r_i)$, or report that no such interval exists.

Case 1 (right). If search goes right, then there exists an overlap in right subtree or no overlap in either.

Proof. Suppose no overlap in right.

- `left[x] = NULL` ⇒ no overlap in left.
- `max[left[x]]` $< \ell_i$ ⇒ no overlap in left.



Interval-Find (i, S)

```
x ← root(S)

WHILE (x != NULL)
   IF (x overlaps i)
      RETURN t
   IF (left[x] = NULL OR
       max[left[x]] < ℓi)
      x ← right[x]
   ELSE
      x ← left[x]
RETURN NO

Splay last node on path
traversed.
```

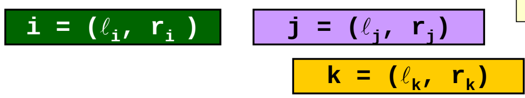

Finding an Overlapping Interval

Interval-Find(i, S): return an interval x that overlaps $i = (\ell_i, r_i)$, or report that no such interval exists.

Case 2 (left). If search goes left, then there exists an overlap in left subtree or no overlap in either.

Proof. Suppose no overlap in left.

- $\ell_i \leq$ `max[left[x]]` $= r_j$ for some interval j in left subtree.
- Since i and j don't overlap, we have $\ell_i \leq r_i \leq \ell_j \leq r_j$.
- Tree sorted by ℓ ⇒ for any interval k in right subtree: $r_i \leq \ell_j \leq \ell_k$ ⇒ no overlap in right subtree.



Interval-Find (i, S)

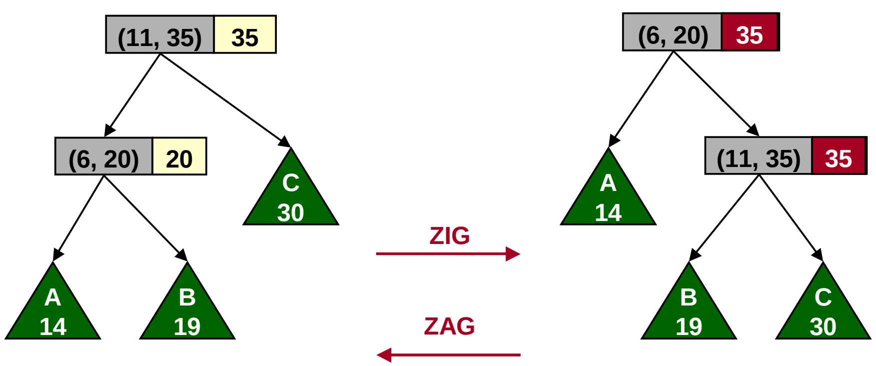
```
x ← root(S)

WHILE (x != NULL)
   IF (x overlaps i)
      RETURN x
   IF (left[x] = NULL OR
       max[left[x]] < ℓi)
      x ← right[x]
   ELSE
      x ← left[x]
RETURN NO

Splay last node on path
traversed.
```


Interval Trees: Running Time

Need to maintain augmented data structure during tree-modifying ops.

- Rotate: can fix sizes in O(1) time by looking at children:

$$\max[x] = \max \begin{cases} \max[\mathit{left}[x]] \\ \max[\mathit{right}[x]] \\ r_x \end{cases}$$



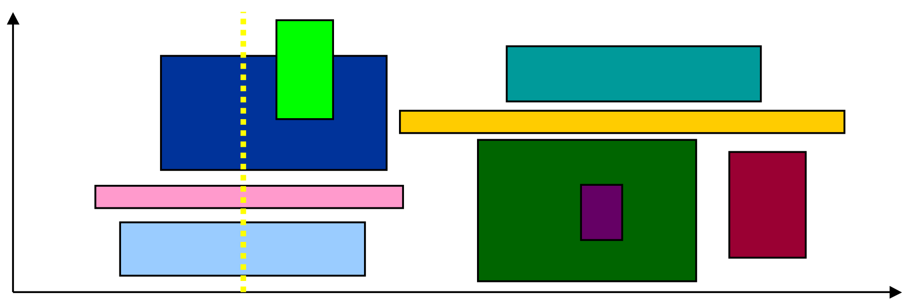
VLSI Database Problem

VLSI database problem.

- Input: integrated circuit represented as a list of rectangles.
- Goal: decide whether any two rectangles overlap.

Algorithm idea.

- Move a vertical "sweep line" from left to right.
- Store set of rectangles that intersect the sweep line in an interval search tree (using y interval of rectangle).



VLSI Database Problem

VLSI (r_1, r_2 ,..., r_N)

```
Sort rectangle by x coordinate (keep two copies of
rectangle, one for left endpoint and one for right).

FOR i = 1 to 2N
   IF (r_i is "left" copy of rectangle)
      IF (Interval-Find(r_i, S))
         RETURN YES
      ELSE
          Interval-Insert(r_i, S)
   ELSE (r_i is "right" copy of rectangle)
      Interval-Delete(r_i, S)
```

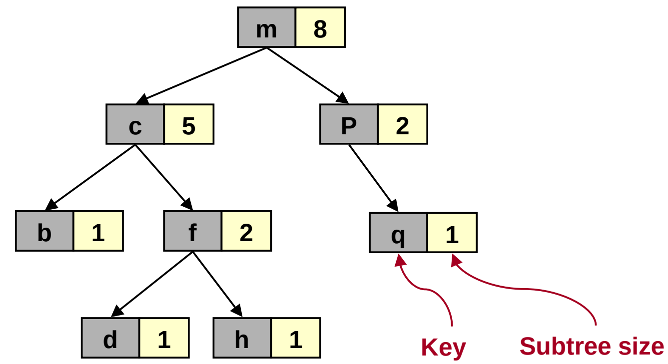

Order Statistic Trees

Add following two operations to BST.

Select(i, S): Return i^{th} smallest key in tree S.

Rank(i, S): Return rank of x in linear order of tree S.

Key idea: store size of subtrees in nodes.



Order Statistic Trees

Need to ensure augmented data structure can be maintained during tree-modifying ops.

- **Rotate: can fix sizes in O(1) time by looking at children.**