## 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

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

\[
\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\log_2 n} 2^j = n + (2^n - 1) < 3n
\]

**Insert** ($x$)
- IF (number of elements in table = $m$)
  - Generate new table of size $2m$.
  - Re-insert $m$ old elements into new table.
  - $m \leftarrow 2m$
- Insert $x$ into table.

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 $\Rightarrow$ double table size.
- Table $\leq \frac{1}{2}$ full $\Rightarrow$ halve table size.

! Bad idea: can cause thrashing.

Dynamic Table: Insert and Delete

**Dynamic Table Delete**

Initialize table size $m = 1$.

DELETE ($x$)
- IF (number of elements in table $\leq m / 4$)
  - Generate new table of size $m / 2$.
  - $m \leftarrow m / 2$
  - 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

---

Binary Search Tree

**Theorem.** Sequence of $n$ inserts and deletes takes $O(n)$ time.
- Amortized cost of insert = $3$.
- Amortized cost of delete = $2$.

**Binary tree in "sorted" order.**
- Maintain ordering property for ALL sub-trees.

- Root (middle value)
- Left subtree (larger values)
- Right subtree (smaller values)
Binary Search Tree

Binary tree in "sorted" order.
- Maintain ordering property for ALL sub-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 ∈ S, and y ∈ 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 ∈ S; otherwise the new root is either max { k ∈ S : k < x} or min { k ∈ S : k > x}.

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

### 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.

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:

ZIG-ZIG

Splay Example

Apply Splay(1, S) to tree S:

ZIG

Splay Example

Apply Splay(1, S) to tree S:

Splay(2)

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 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

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:
  \[
  \mu'(x) + \mu'(y) - \mu(x) - \mu(y) = \mu'(y) - \mu(x) \\
  \leq \mu'(x) - \mu(x) \\
  \leq 3(\mu'(x) - \mu(x)) \\
  = 3(\mu(S) - \mu(x))
  \]
- Use extra credit to pay for low-level operations.

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

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

- Argument similar to ZIG-ZIG.
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.

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

Interval Trees

Interval Trees

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.

**Algorithm**

```plaintext
x ← root(S)
WHILE (x ≠ NULL)
    IF (x overlaps i)
        RETURN x
    IF (left[x] = NULL OR max[left[x]] < \(l_i\))
        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 = (l_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 \implies no overlap in left.
- max[left[x]] < l_i \implies no overlap in left.

\[ i = (l_i, r_i) \]

\[ \text{max} \]

\[ \text{left}[x] \]

Splay last node on path traversed.

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.

- \[ \ell_1 \leq \text{max}[\text{left}[x]] = r_j \] for some interval j in left subtree.
- Since i and j don't overlap, we have \[ \ell_1 \leq r_i \leq \ell_2 \leq r_j \].
- Tree sorted by \( \ell \) \implies for any interval k in right subtree: \[ r_i \leq \ell_3 \leq \ell_k \implies \text{no overlap in right subtree} \]

\[ i = (l_i, r_i) \]

\[ j = (l_i, r_i) \]

\[ k = (l_k, r_k) \]

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:

\[ \text{max}[x] = \max \begin{cases} \text{max}[\text{left}[x]] \\ \text{max}[\text{right}[x]] \end{cases} \]

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

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.