Topic 09: Splay Trees

- Amortized time
- Splay Trees
Amortized Time

- **Amortized** time gives long-term time information
  - Individual operations maybe be far more costly than stated
  - But the total time of many operations
Scanner in = ... ;
String[] words = new String[0];

while (in.hasNext())  \(\Theta(n)\)
{
    words = resize(words,
                   words.length+1);
    words[words.length-1] = in.next();
}
Scanner in = ... ;
String[] words = new String[0];

while (in.hasNext())
{
    words = resize(words, words.length+1);
    words[words.length-1] = in.next();
}

Asymptotic time cost of one iteration: $O(n)$
Scanner in = ... ;
String[] words = new String[0];

while (in.hasNext()) {
  words = resize(words, words.length + 1);
  words[words.length - 1] = in.next();
}

Asymptotic time cost of entire loop: $O(n^2)$
Amortized Time

Scanner in = ... ;
String[] words = new String[1];
int count = 0;

while (in.hasNext())
{
    if (count == words.length)
    {
        words = resize(words,
                        words.length*2);
    }

    words[count] = in.next();
    count++;
}

Scanner in = ...;
String[] words = new String[1];
int count = 0;

while (in.hasNext())
{
    if (count == words.length)
    {
        words = resize(words,
                        words.length*2);
    }

    words[count] = in.next();
    count++;
}
Scanner in = ... ;
String[] words = new String[1];
int count = 0;

while (in.hasNext())
{
    if (count == words.length)
    {
        words = resize(words, words.length*2);
    }

    words[count] = in.next();
    count++;
}

These parts total to: $O(1)$
Amortized Time

Scanner in = ... ;
String[] words = new String[];
int count = 0;

while (in.hasNext())
{
    if (count == words.length)
    {
        words = resize(words,
                        words.length*2);
    }
    words[count] = in.next();
    count++;
}

This is:
O(n)

But it's very rare!
Amortized Time

Total cost for the loop:

\[ cn + x \left( 1 + 2 + 4 + 8 + 16 + \ldots + n \right) \]

\[ cn + x \left( 2n - 1 \right) \]

\[ n \left( c + 2x \right) - x \]

- \( c \) – cost of the \( O(1) \) part of the loop
- \( x \) – cost of the array duplication
Scanner in = ...;
String[] words = new String[1];
int count = 0;

while (in.hasNext())
{
    if (count == words.length)
    {
        words = resize(words,
                        words.length*2);
    }

    words[count] = in.next();
    count++;
}
Calculating Amortized Time

\[ t_{amortized} = \frac{t_{total}}{ops} \]
Scanner in = ...;
String[] words = new String[1];
int count = 0;

while (in.hasNext())
{
    if (count == words.length)
    {
        words = resize(words, words.length*2);
    }

    words[count] = in.next();
    count++;
}
Proving Amortized Time

- Total / operations
  - This is actually the **definition** of amortized time

- Accounting method

- Potential method
Accounting Method

- Declare that constant-time actions take 1 credit each
- Assign a budget to each operation
  - You get “paid” for operations
- You must pay for each operation you perform

- Do you ever run out of credits?
Accounting Method

Our 1-credit operations:

- The $O(1)$ parts of the loop (1 credit total)
- Extending the array (1 credit per element in the new size)

- Budget: 3 credits per insert
  - 1 to pay for the $O(1)$ parts
  - 2 to pay for future expansion
Potential Method

- Find some function $\Phi$, which maps your state to a positive value
- $\Phi$ is the “potential” stored up in the system
  - Like potential energy in physics!
- For each op, calculate change in potential:
  \[ \Delta \Phi = \Phi( S_{\text{end}} ) - \Phi( S_{\text{start}} ) \]
- Use potential to “pay for” expensive operations:
  \[ T_{\text{amortized}} = T_{\text{actual}} + \Delta \Phi \]
Potential Method

Our Potential Function:

\[ \Phi = 4 \times \text{count} - 2 \times \text{words.length} + 2 \]

Key Properties:
- \( \geq 0 \) at the init state
- Increases by 4 on most insertions
- Drops when we expand the array
- Never goes negative
Potential Method

$\Phi = 4 \cdot \text{count} - 2 \cdot \text{words.length} + 2$

• When we expand the array:
  - count increases by 1
  - words.length doubles (to twice the old count)

$$\Delta \Phi_{\text{expand}} = \Phi_{\text{new}} - \Phi_{\text{old}} = 4 - 2 \cdot \text{count}_{\text{old}}$$
Potential Method

\[ \Phi = 4 \times \text{count} - 2 \times \text{words.length} + 2 \]

Budget: 5 per insert

- Most insertions:
  - 1 unit to pay for \( O(1) \) costs
  - 4 units to store into potential

- Expansions:
  - 1 unit to pay for \( O(1) \) costs
  - \( 2 \times \text{count}_{\text{old}} \) to pay for array expansion
  - \( 4 - 2 \times \text{count}_{\text{old}} \) taken from potential
Amortized Time (Summary)

- Amortized time = total / ops
  - Some operations cheap, save up for later
  - Some operations expensive, use up stored value

- Three ways to prove it
  - Calculate total time
  - Accounting method
  - Potential method
Amortized != Average

- Amortized time is **not** average time!

- Average time:
  - Average over all possible inputs
  - Some inputs may lead to poor worst case

- Amortized time:
  - Average over many steps
    - Some steps cheap, others expensive
  - But works with **any possible input!**
Topic 09: Splay Trees

- Amortized time
- Splay Trees
Splay Trees

• Splay trees are not balanced!

• Instead, we “splay” a node to the root any time it is referenced
  – Insert
  – Search

• Insight: Recently used nodes are likely to be used again, soon.
Splay Trees

Here's where it gets weird:

- Splay trees have $O(n)$ height, but...
- Splay trees still display $O(\log n)$ amortized performance
Splaying (on Insert)

1. Start with an ordinary tree
2. Insert new node
3. Splay the node up to the root
Splaying (on Search)

1. Start with an ordinary tree
2. Find the node in the tree
3. Splay the node up to the root
Splaying Rotations

• Three rotations to know:
  “Zig” - one level rotation (only used at the root)
  “Zig-Zig” - two level rotation, both in same direction
  “Zig-Zag” - two level rotation, in opposite directions

• Start at the node, and do 2-level rotations toward the root; do a 1-level at the end if necessary
NOTES:
• A will always represent the node we're splaying
• We'll leave off all of the sub-trees for simplicity
Order matters here. Rotate at C first. Otherwise, we cannot guarantee $O(\log n)$ performance.
Zig-Zag
Splay Trees In Practice: Insertion

Insert 1 into the tree.
Splay Trees In Practice: Insertion

Insert 2 into the tree.

Normally, 2 would be a child of 1...but we splay it up to the root.
Splay Trees In Practice: Insertion

The process continues as we insert more nodes...it never balances.
Splay Trees In Practice: Insertion
Splay Trees In Practice: Insertion
Splay Trees In Practice: Insertion
Splay Trees In Practice: Insertion
So what happens if we search for 1 here?

Let's track the steps of splaying it up to the root...
Splay Trees In Practice: Search

This is a zig-zig step.

This part of the subtree is reversed...things look bad, don't they?
This is also a zig-zig step.

But you see how the tree is starting to collapse?
Splay Trees In Practice: Search

This is also a zig-zig step.

Do you see how the tree is starting to collapse?
The last step is a zig step, since we're at the root.
Splay Trees In Practice: Search

But if we search for 2 next, and then 3 and 4...it can get bad again!
We arrived at this configuration by doing:

- Lots of **very cheap** insertions
- A single expensive search

**Intuition:**
Cheap operations “save up” for later expensive ones.

Thus, we use **amortized time.**
Splay Trees: Discussion

- In what sense is a splay tree fast?
- In what sense is it slow?

- What sorts of programs would work well with a splay tree?
- What sorts of programs would not work well with a splay tree?