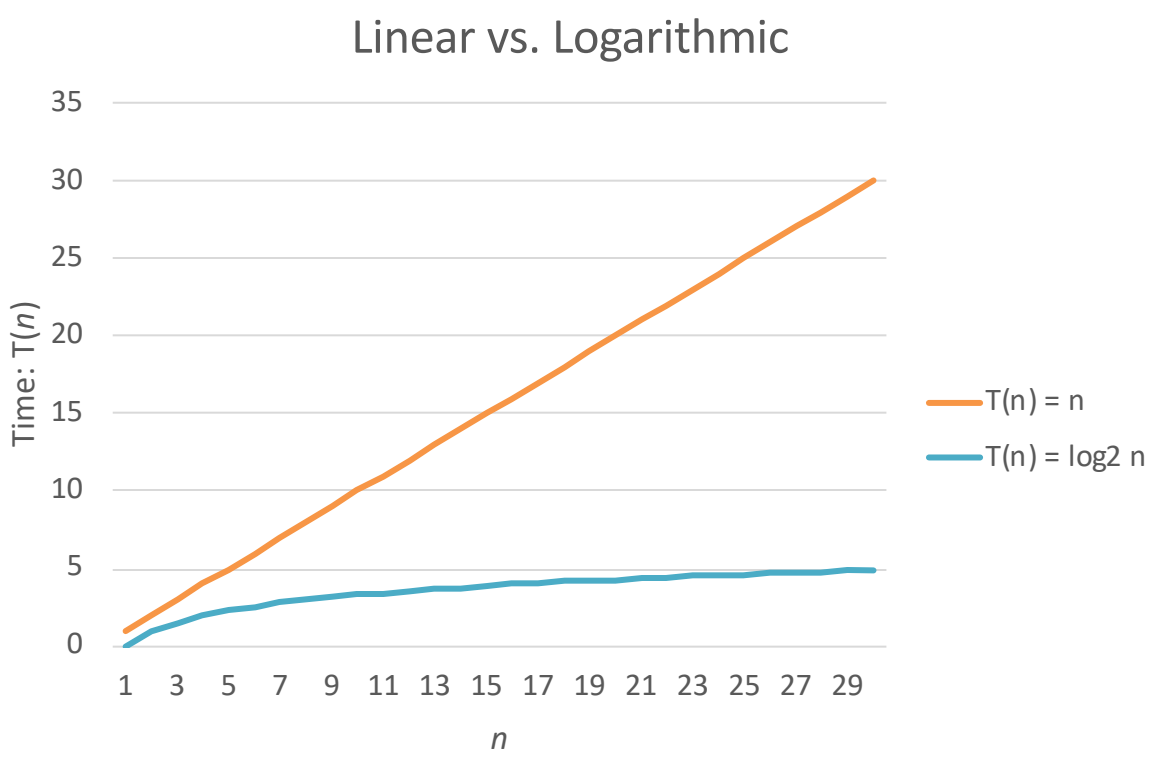
Sort then Search?

Recall: Linear and Binary Search

- Find an element of an array
- Linear search: $O(n)$ time
- Binary search: $O(\log n)$ time
  - Requires sorted array



Searching an Array: Sort then Search?

Linear Search

Total: $O(n)$

Sorting is expensive!
(compared to searching)

Sort + Binary Search

Total: $\Theta(n \log n) + O(\log n) = \Theta(n \log n)$

Searching an Array: Sort then Search?

2 x Linear Search

Total: $2 \cdot O(n) = O(n)$

Sort + 2 x Binary Search

Total: $\Theta(n \log n) + 2 \cdot O(\log n) = \Theta(n \log n)$

Per Search: $\frac{\Theta(n \log n)}{2} = \Theta(n \log n)$

Not "real", but useful!

Searching an Array: Sort then Search?

n x Linear Search

Total: $n \cdot O(n) = O(n^2)$

Sort + n x Binary Search

Total: $\Theta(n \log n) + n \cdot O(\log n) = \Theta(n \log n)$

Per Search: $\frac{\Theta(n \log n)}{n} = \Theta(\log n)$

Sorting is worth it if you need to search a lot!

Side Note: Maintaining a Sorted Array?

- What if you want to insert and remove things?
  - Find where to insert/remove: $O(\log n)$
  - Inserting/removing in the middle is expensive: $O(n)$
- Use a linked list?
  - Insert/remove: $O(1)$
  - Binary search relies on random access: $O(n)$
- Binary search tree!
  - Maintains sorted order (in-order traversal)
  - Search: $O(\log n)$
  - Insert/remove: $O(\log n)$

Use binary search!

BST search is basically binary search!

Summary

- Is it worth it to sort before searching?
  - Only if you do lots of searches after sorting
  - (Lots = $\Theta(n)$)
- Sometimes you need to spend time to save time
  - Do an expensive operation to make future operations cheaper
- Main ideas:
  - Analyze total cost of a *sequence* of operations
  - Report cost per operation (a useful fiction!)

Coming Up…

Amortized Analysis

Amortized Analysis

Array-Backed List

- A list structure that stores elements in an array
- See:
  - C++ std::vector
  - Python list
  - Java ArrayList
- To append:
  - If enough space, simply put item in array ($\Theta(1)$)
  - If out of space… ($\Theta(n)$)
    - Allocate a bigger array ($\Theta(1)$)
    - Copy the contents ($\Theta(n)$)
    - Delete old array ($\Theta(1)$)

$O(n)$

Complexity Analysis

Worst-case time of appending m items?

```
vector<size_t> v;
for (size_t i = 0; i < m; ++i) {
  v.push_back(i);
}
```

Are we really going to resize for every append??

Worst case: resize ($\Theta(n)$)

$n = i = 0, 1, 2, \ldots, m-1$

Complexity: $O(1 + 2 + \cdots + m - 1) = O\left(\frac{(m-1)m}{2}\right) = O(m^2)$

Complexity Analysis

- Say that when we resize, we add k slots (start with k slots)
  - Then we resize every k inserts
- Complexity of m appends?
  - $m\left(\frac{k-1}{k}\right)$ easy appends: $m\left(\frac{k-1}{k}\right)\cdot\Theta(1)=\Theta(m)$
  - $m\frac{1}{k}$ resize appends: $\Theta\left(k+2k+\cdots+\frac{m}{k}k\right)=\Theta\left(k\left(1+2+\cdots+\frac{m}{k}\right)\right)$

$$=\Theta\left(k\frac{\frac{m}{k}\left(\frac{m}{k}+1\right)}{2}\right)=\Theta(m^2)$$

  - Overall: $\Theta(m)+\Theta(m^2)=\Theta(m^2)$ Comparable to resizing for every append!

Complexity Analysis

- Say that when we resize, we double the slots (start with 1 slot)
  - Then we resize when there are $1, 2, 4, 8, \ldots = 2^0, 2^1, 2^2, 2^3 \ldots$ items
- Complexity of m appends?
  - How many resizes?
    - How many powers of 2 are less than m?
    - $2^x \leq m \rightarrow x \leq \log_2 m$
    - So $\Theta(\log m)$ resizes
  - $\Theta(m - \log m)$ easy appends: $\Theta(m - \log m) \cdot \Theta(1) = \Theta(m) \cdot \Theta(1) = \Theta(m)$
  - $\Theta(\log m)$ resize appends: $O(2^0 + 2^1 + 2^2 + \cdots + 2^{\log_2 m}) = O(2^{\log_2 m}) = O(m)$
  - Overall: $\Theta(m) + O(m) = \Theta(m)$

$O(a^0 + a^1 + a^2 + \cdots a^n) = O(a^n)$

O because of this $\leq$

Better than resizing for every append!

Amortized Analysis

- An *amortized* time is a bound that promises
  - The worst case time of a sequence of operations (starting from “empty”) is bounded by the sum of the amortized times of the individual operations
- The amortized time is *not* how long an operation actually takes!
  - It is the *per operation* time of a sequence of operations

Amortized Analysis

- Array-Backed List (adding capacity)
  - Cost of appending n items: $\Theta(n^2)$
  - Amortized $\frac{\Theta(n^2)}{n} = \Theta(n)$ insert time
- Array-Backed List (doubling capacity)
  - Cost of appending n items: $\Theta(n)$
  - Amortized $\frac{\Theta(n)}{n} = \Theta(1)$ insert time

Amortized Analysis

- Amortized times only make sense when starting from "empty"

```
void f(std::vector<int>& v) {
  v.push_back(0);
}
```

Can't promise $O(1)$ for 1 append...
v might be huge and about to resize!

- Our sort + search analysis was not quite an amortized analysis
  - Only promise $O(\log n)$ per search for one sort and a *lot* of searches
  - Amortized times bound the worst case for any sequence!

Average-Case Analysis

- Contrast: Average-case analysis
  - For finding “typical” complexity
  - Assign probabilities to different cases
  - Get expected time complexity, given those probabilities
- Amortized analysis
  - For finding aggregate *worst-case* complexities
  - Consider sequences of operations
  - Get time per operation

Summary

- Array-backed list
  - Store a list in an array, resize as necessary
- Worst-case analysis is misleading
  - The worst case is resizing, but we don't always resize!
- Amortized analysis
  - Rather than summing up individual worst-case complexities…
  - …Analyze entire sequence of operations (starting from "empty")
  - Report average time per operation
- Amortized time of an operation
  - Time complexity of sequence is bounded by sum of amortized times
- Array-backed lists: amortized $\Theta(1)$ append

Coming Up…

Splay Trees

Splay Trees: Intro

Splay Trees

- A splay tree is a self-balancing binary search tree, like
  - Randomized BSTs
  - Red-black trees (and 2-3-4 trees)
- Complexities (n nodes):
  - Worst-case $O(n)$ insert and search
  - Amortized $O(\log n)$ insert and search

Splay Trees

- What is the worst-case time of the following sequence?
  - Insert m distinct items into an empty tree
  - Look up each of the m items

- Tempting:
  - Worst-case $O(n)$ for each operation so…
  - Inserts: $O(1 + 2 + \cdots + m)$
  - Lookups: $O(m + m + \cdots + m)$
  - Total worst-case complexity: $O(m^2)$

True!

But too conservative/loose
It's not going to take this long!

Splay Trees

- What is the worst-case time of the following sequence?
  - Insert m distinct items into an empty tree
  - Look up each of the m items

- Better:
  - Amortized $O(\log n)$ for each operation so…
  - Inserts: $O(\log 1 + \log 2 + \cdots + \log m)$
  - Lookups: $O(\log m + \log m + \cdots + \log m)$
  - Total worst-case complexity: $O(m \log m)$

A tighter bound!

Important: this is not an amortized bound
It's an actual worst-case bound!

Splay Trees

- https://www.cs.usfca.edu/~galles/visualization/SplayTree.html
- Sort of like insert-at-root
  - But it happens every time
  - Not randomized!
- Changes the tree when you search!
  - Recently found items move to the root
- Overall principle
  - There might be a long branch
  - But if you insert/search there, you make it shorter

Splay Trees Pros?

- Good worst-case bounds on op. sequences (amortized $O(\log n)$)
  - BSTs and randomized BSTs have amortized $O(n)$ operations)
- No extra bookkeeping
  - Red-black trees and randomized BSTs have more space overhead
- Relatively simple to implement
  - Simpler than red-black trees
  - Slightly more complicated than randomized BSTs
- Locality
  - Recently/frequently found items are cheap to find again

Splay Tree Cons?

- Worst-case $O(n)$ time of individual operations
  - Variation in runtime for each insert/search
  - Red-black trees guarantee $O(\log n)$ worst-case complexity
- Overhead of rotations on every operation
  - BST has no re-balancing overhead
  - Randomized BST and red-black trees only re-balance on insert/delete
- Lookup can change the tree!
  - Seems weird
  - In C++, we expect `exists(const item_t& x)` to be `const`, but it can't be!
  - (Actually it can be – look up the mutable keyword, if you are interested)

Summary

- Splay Trees
  - Self-balancing binary search tree
  - Worst-case $O(n)$ insert and search
  - Amortized $O(\log n)$ insert and search
  - Low space overhead
  - Especially good when there are common/repetitive lookups
- Demo
  - Inserted items move to root
  - Found items move to root
  - Individual operations might be expensive
    - But long branches get pulled up

Coming Up…

Splay Tree Details

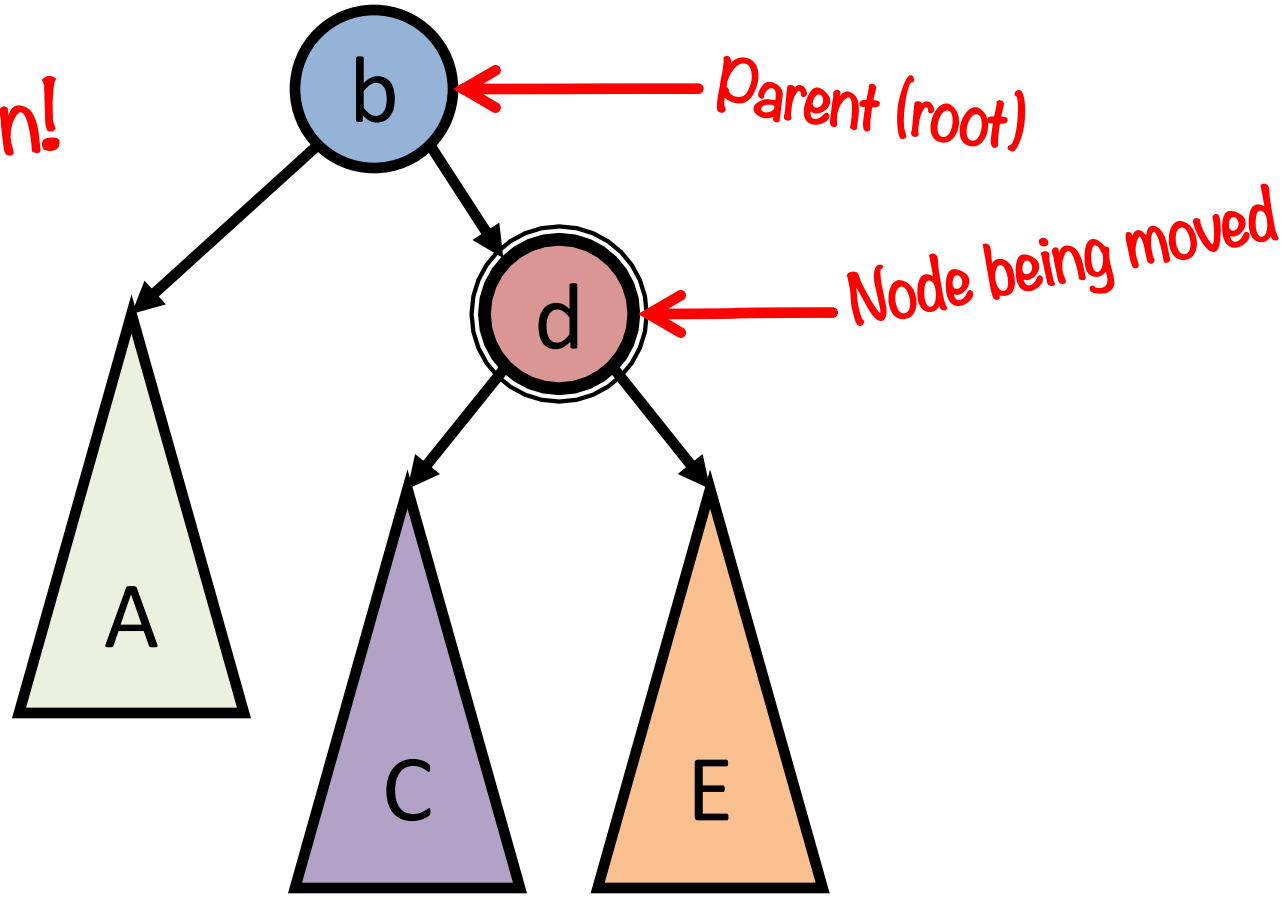
Splay Trees: Details

Splay Tree Insertion

- Big picture
  - Perform normal BST insertion
  - Then percolate the new node to the root
    - "Splay" operation
- Splay operation
  - Move new node up 2 levels at a time (double rotations!)
  - When possible, use a special double rotation

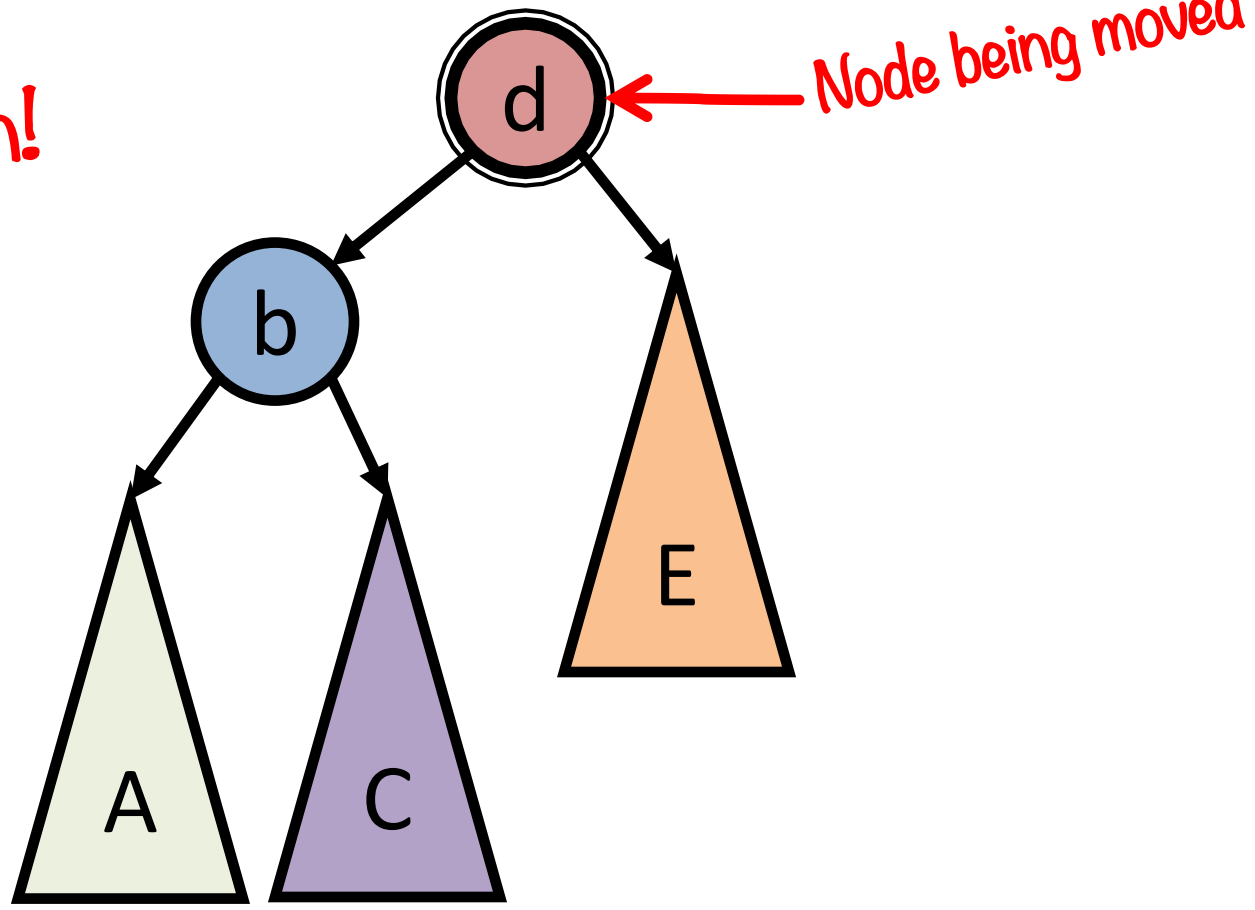
Node's parent is root: "zag" operation

Just a regular rotation!



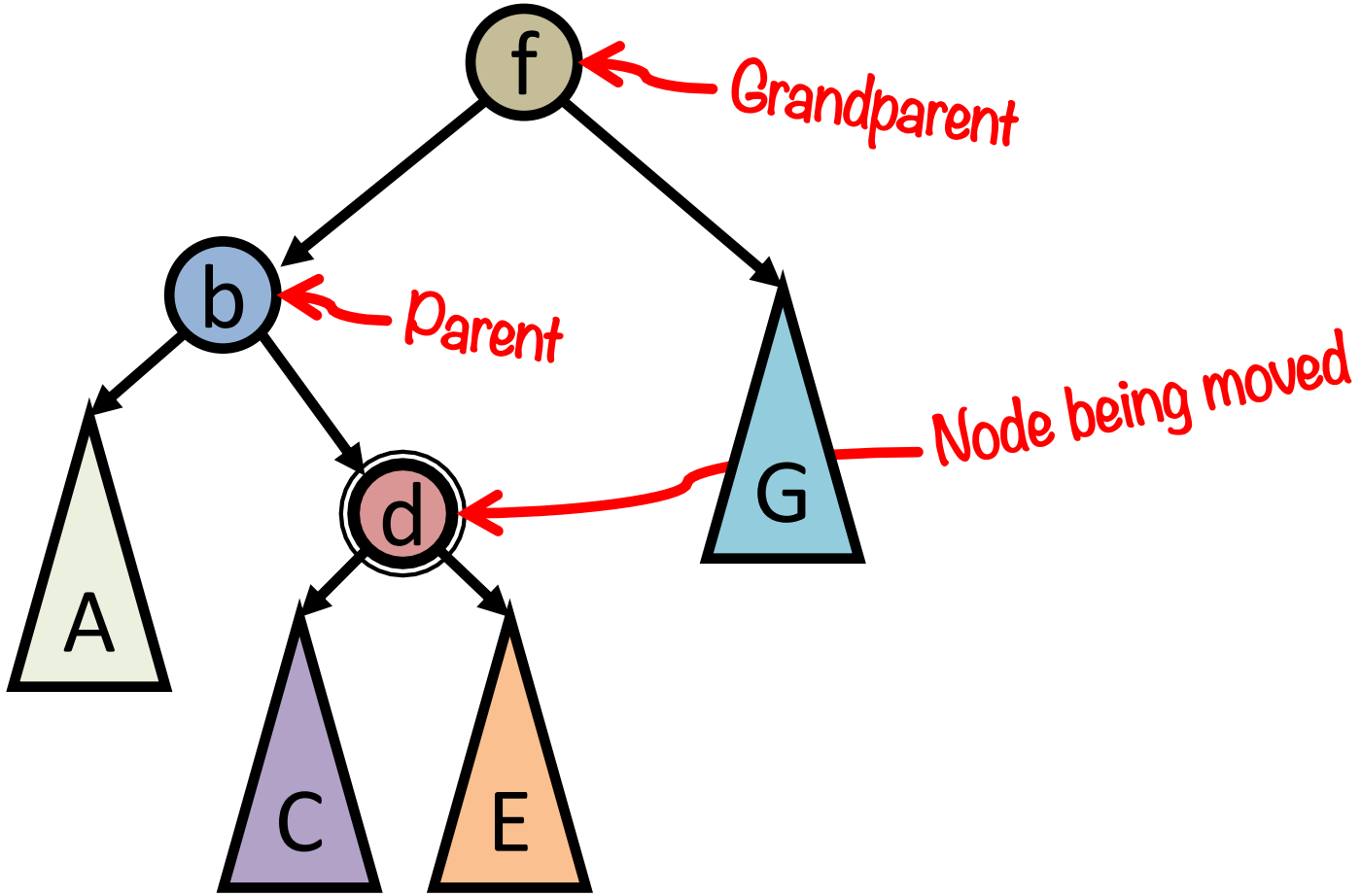
Node's parent is root: "zag" operation

Just a regular rotation!



Right child of left child: “zag-zig” operation

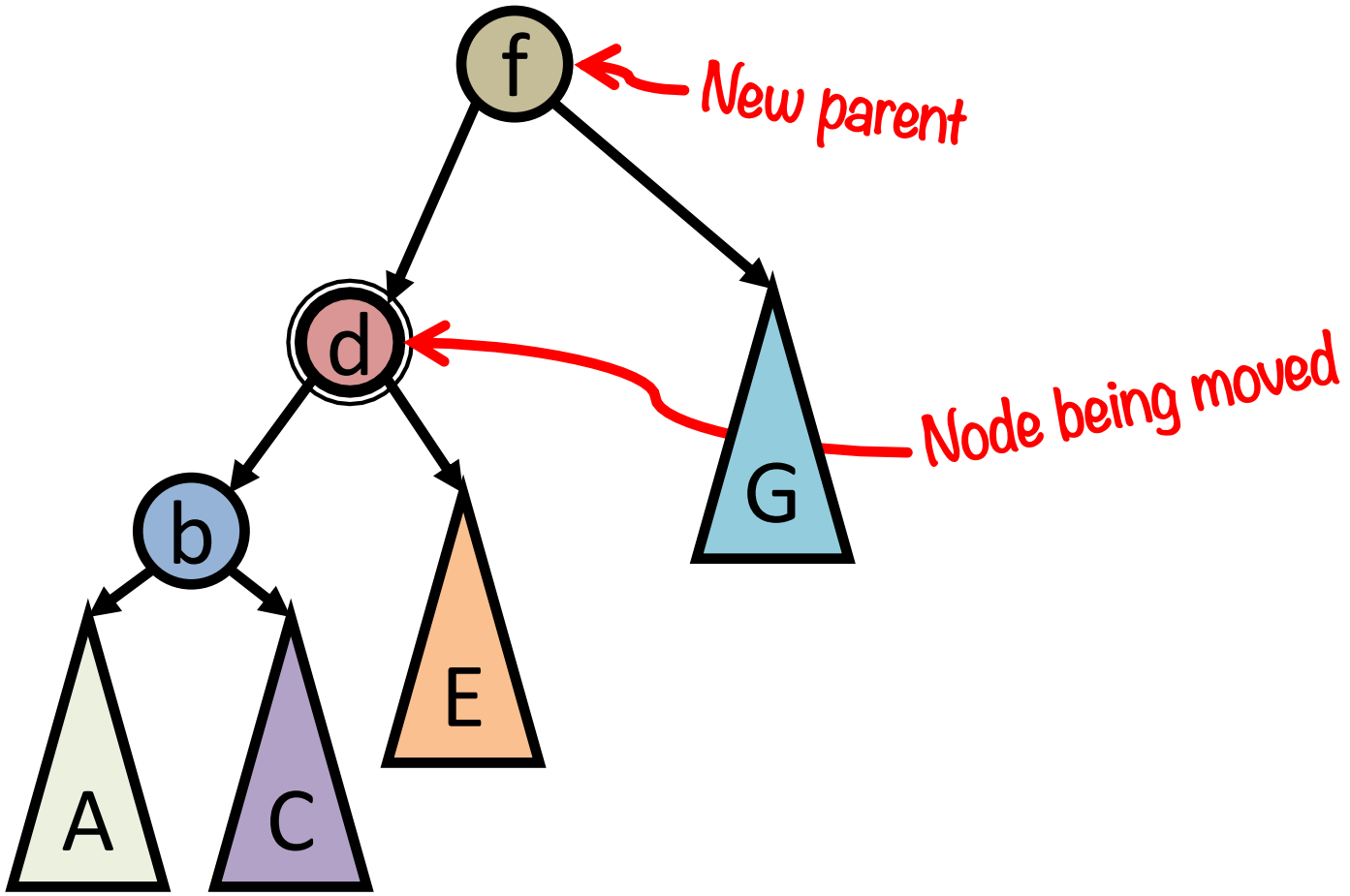
Just two regular rotations!



Right child of left child: “zag-zig” operation

Just two regular rotations!

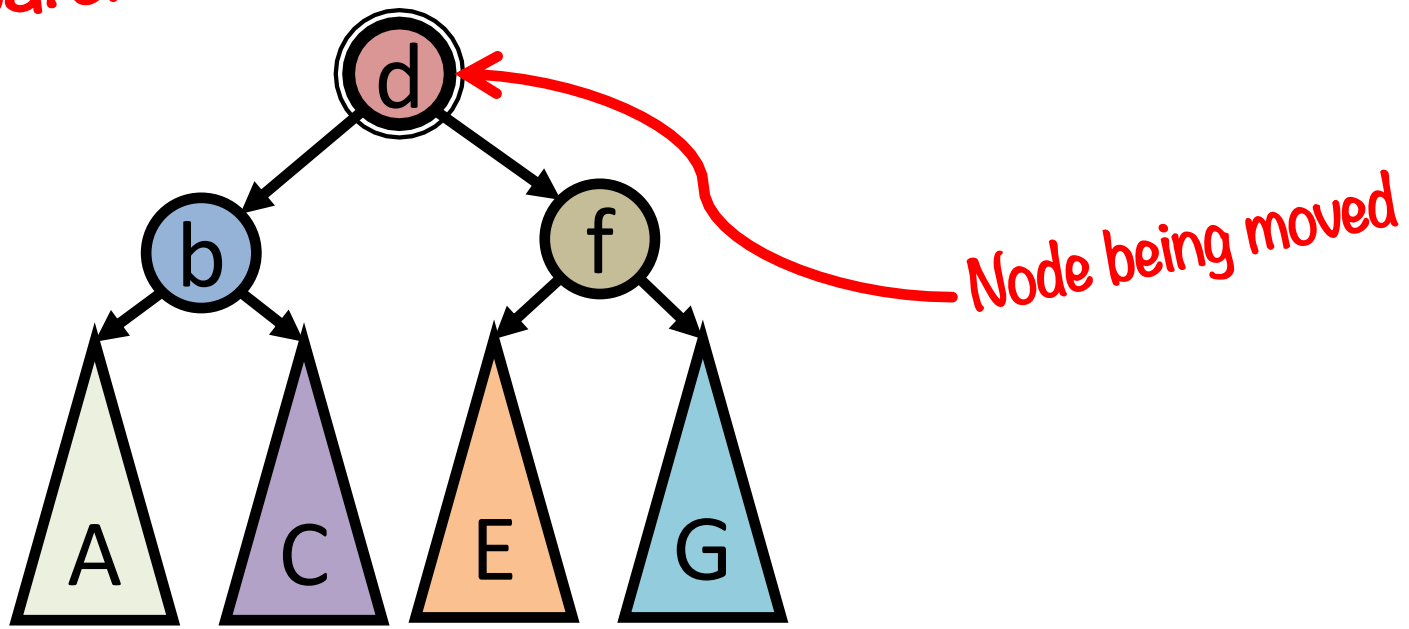
1. Rotate on parent



Right child of left child: “zag-zig” operation

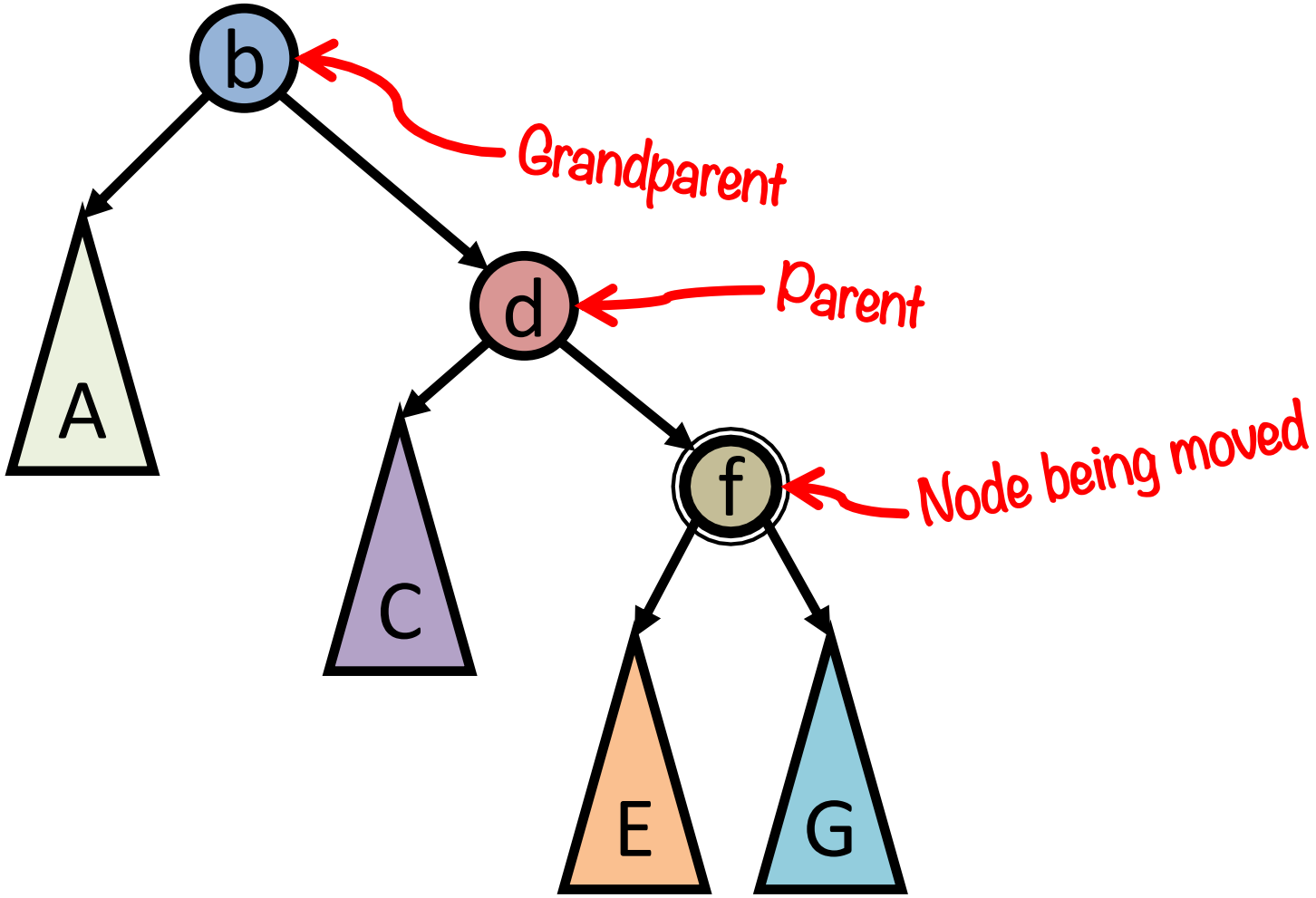
Just two regular rotations!

1. Rotate on parent
2. Rotate on new parent



Right child of right child: “zag-zag” operation

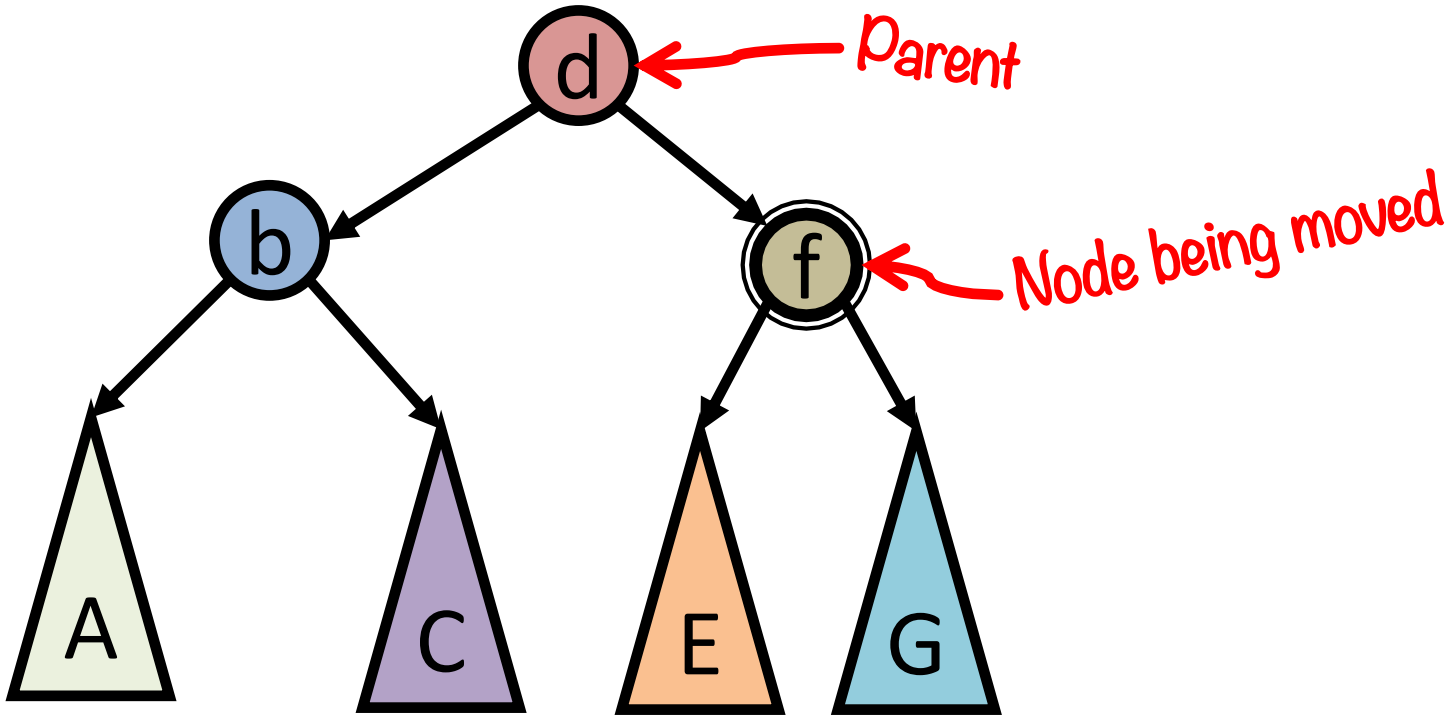
Special to splay trees



Right child of right child: “zag-zag” operation

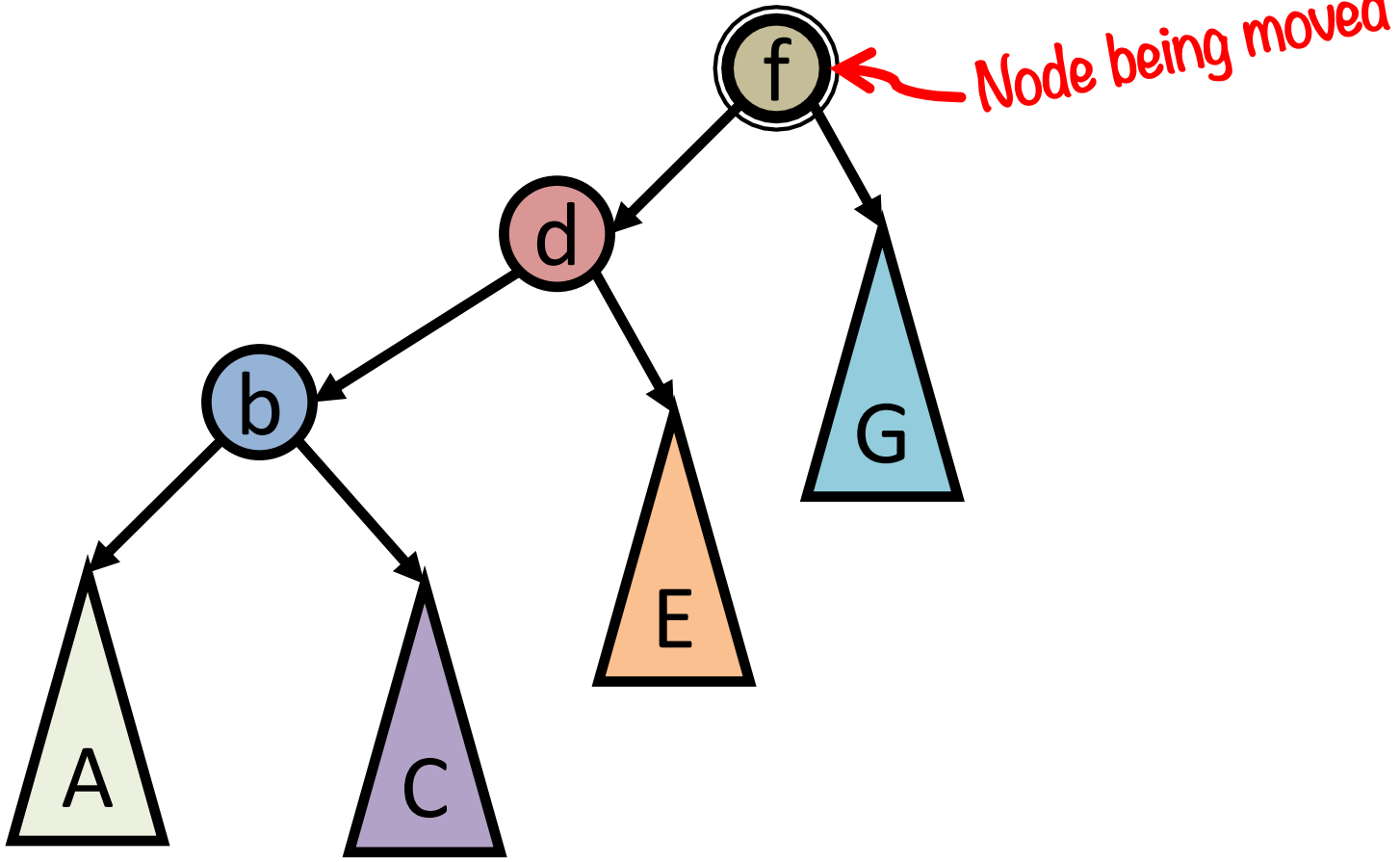
Special to splay trees

1. Rotate on grandparent



Right child of right child: “zag-zag” operation

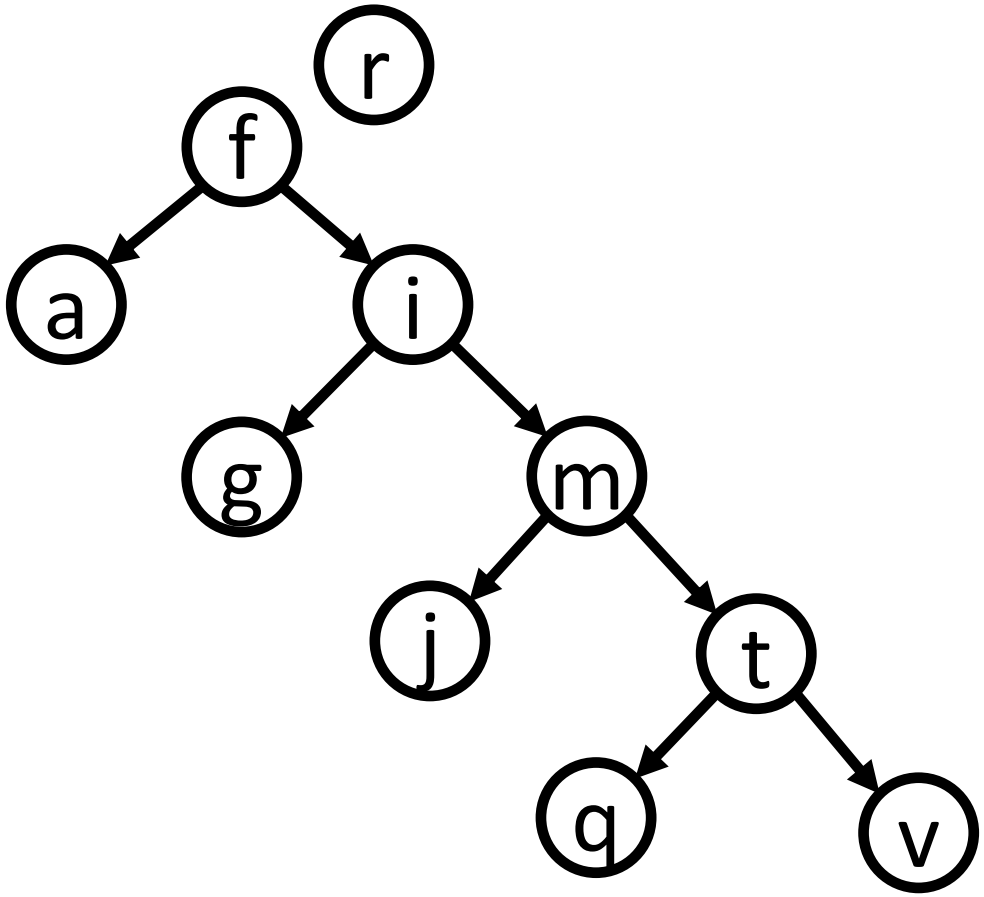
Special to splay trees
1. Rotate on grandparent
2. Rotate on parent



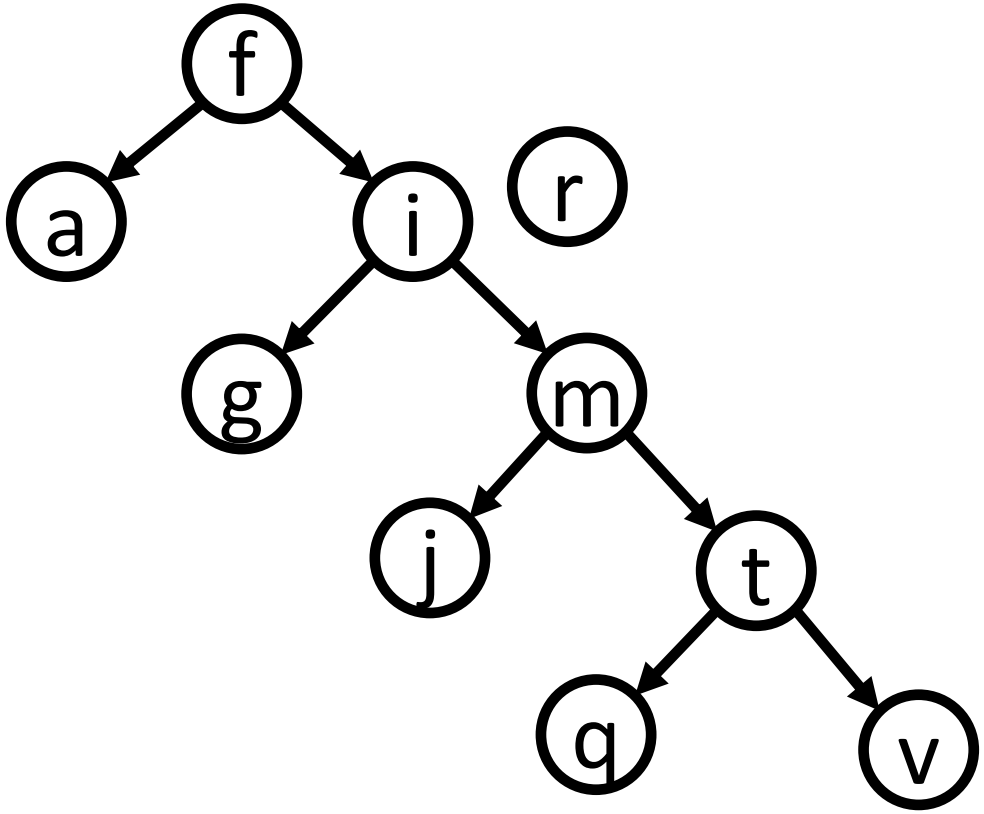
Splay Tree Operations

- Splay Operation
  - Move the given node to the root
  - Use zig-zig and zag-zag rotations when possible
- Insert
  - Insert as usual
  - Splay new node
- Search
  - Search as usual
  - Splay found node

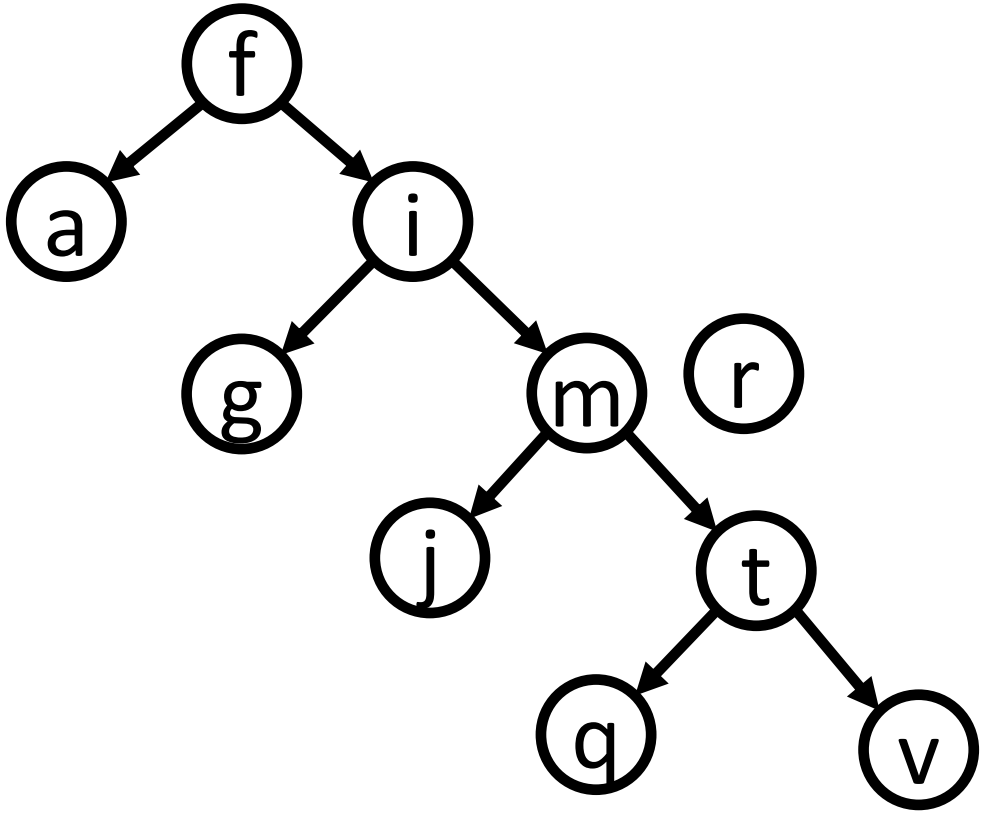
Example: Insert



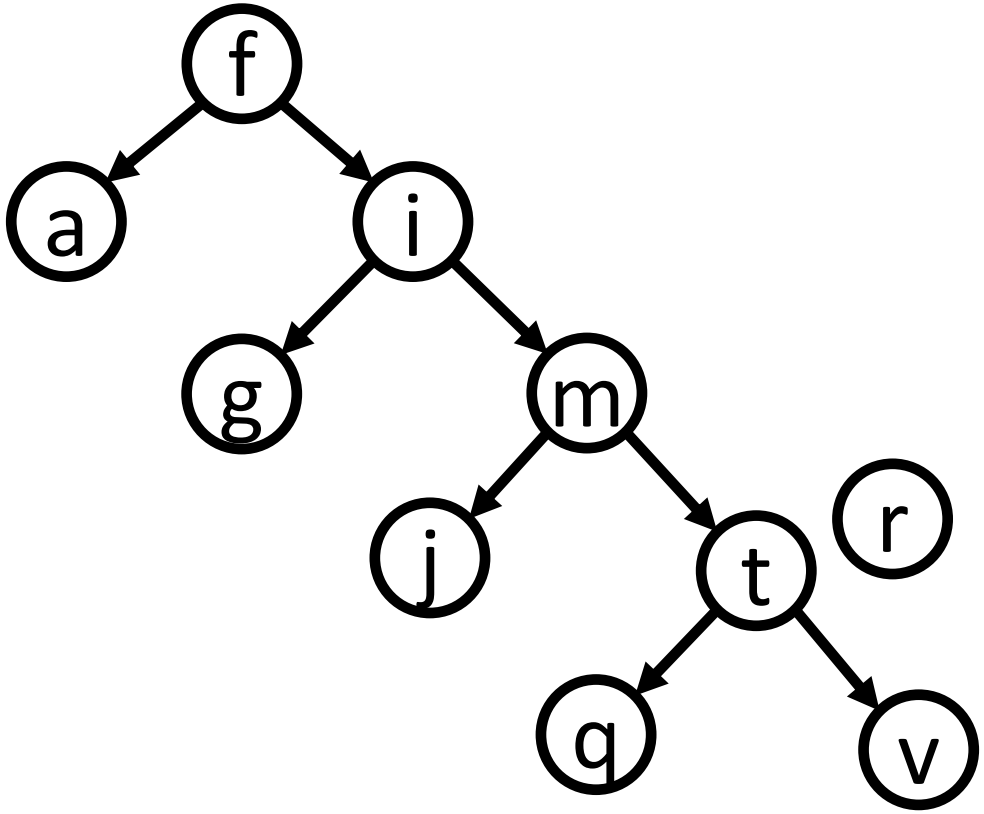
Example: Insert



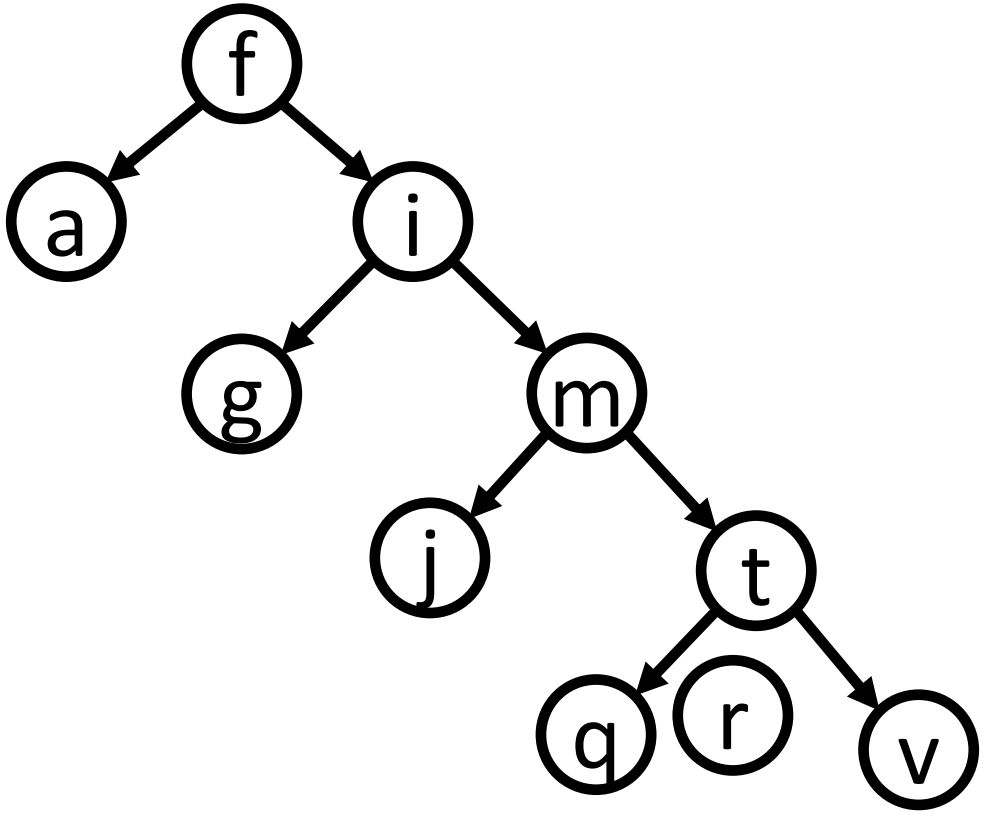
Example: Insert



Example: Insert

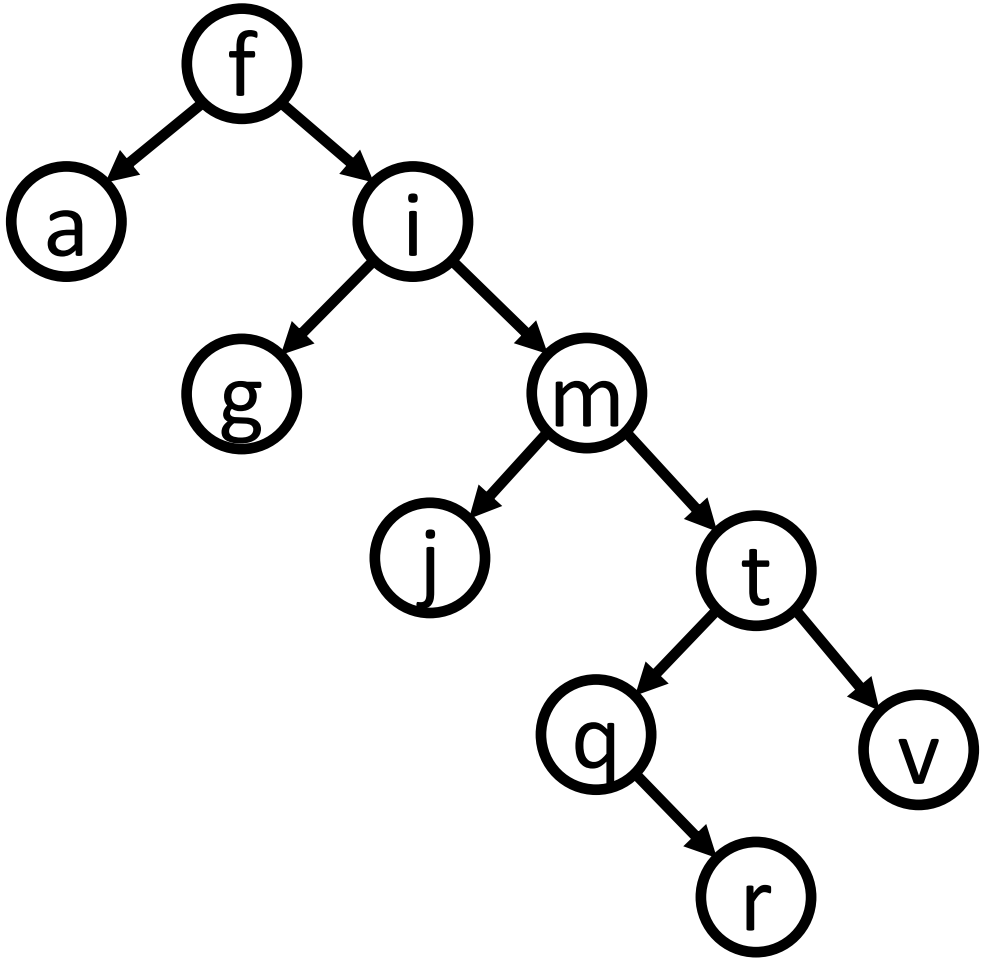


Example: Insert



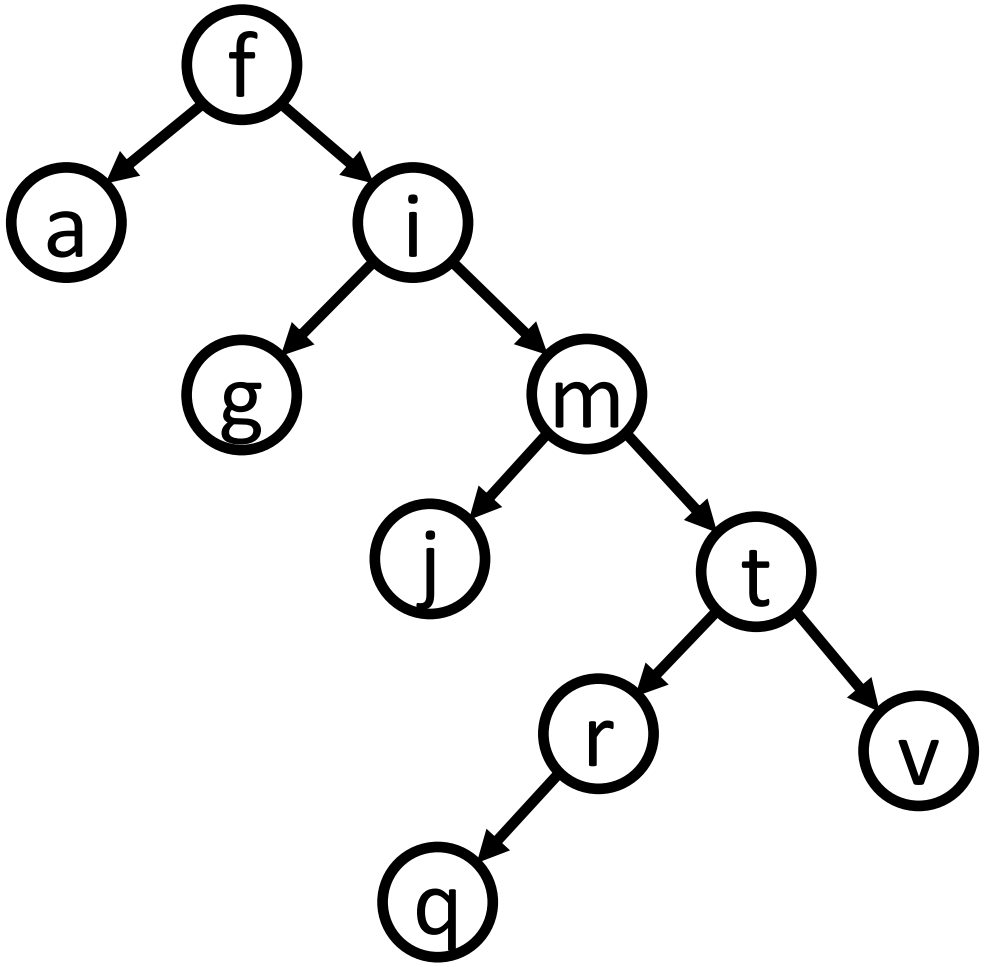
Example: Insert

Zag-zig
1. Rotate on parent
2. Rotate on new parent



Example: Insert

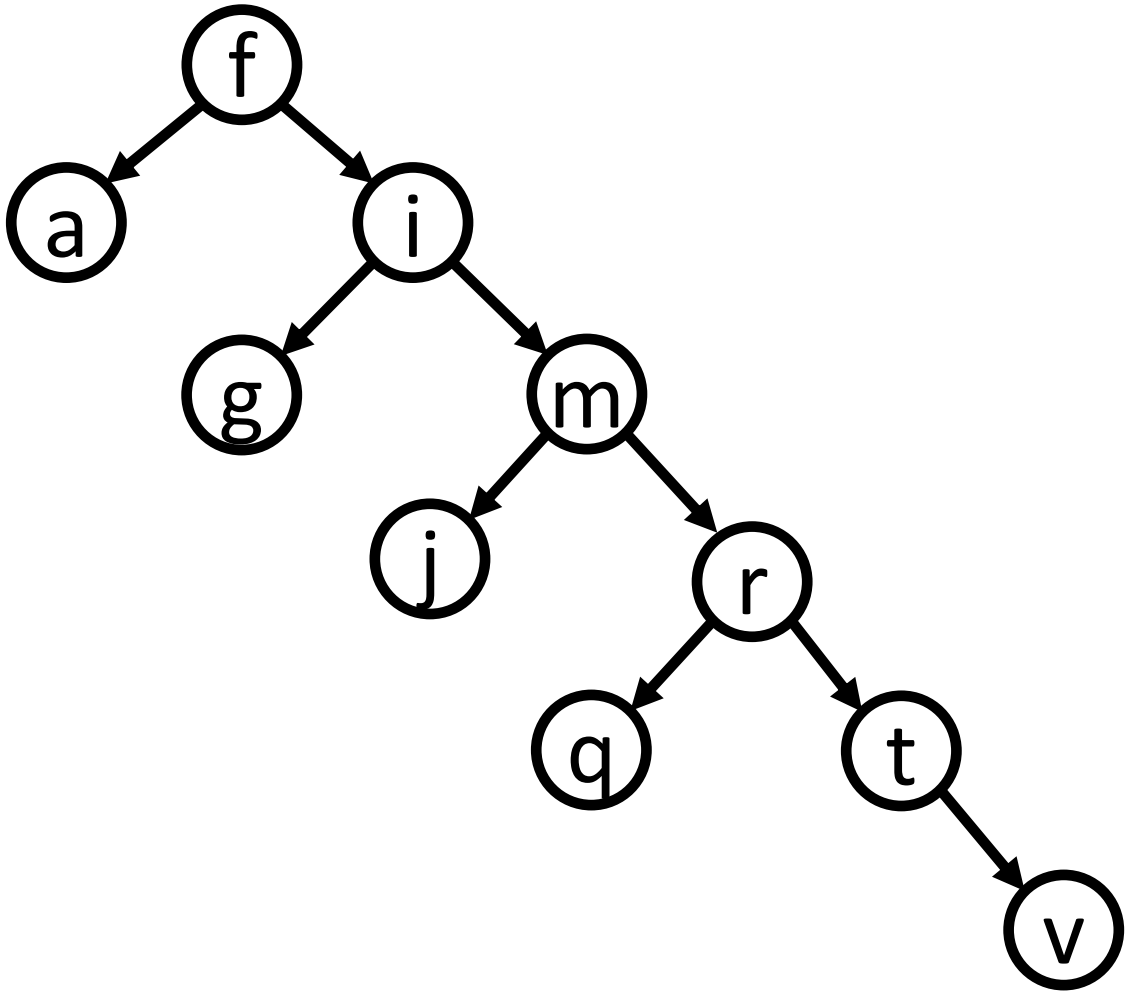
Zag-zig
1. ~~Rotate on parent~~
2. Rotate on new parent



Example: Insert

Zag-zig
1. ~~Rotate on parent~~
2. ~~Rotate on new parent~~

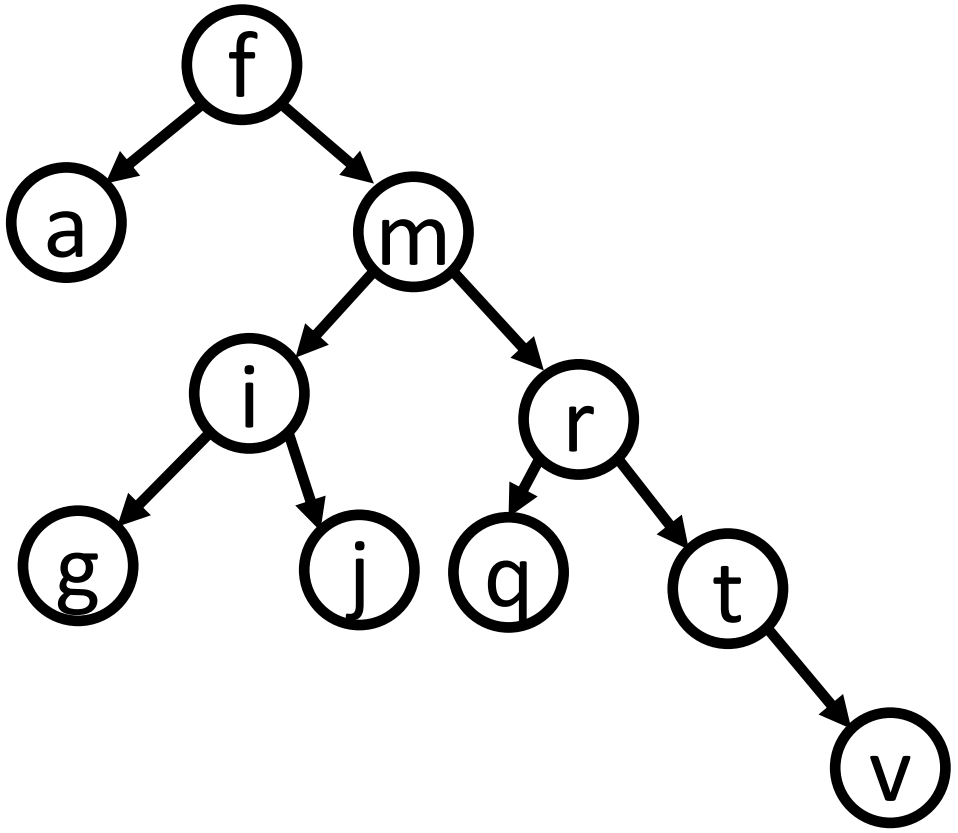
Zag-zag
1. Rotate on grandparent
2. Rotate on parent



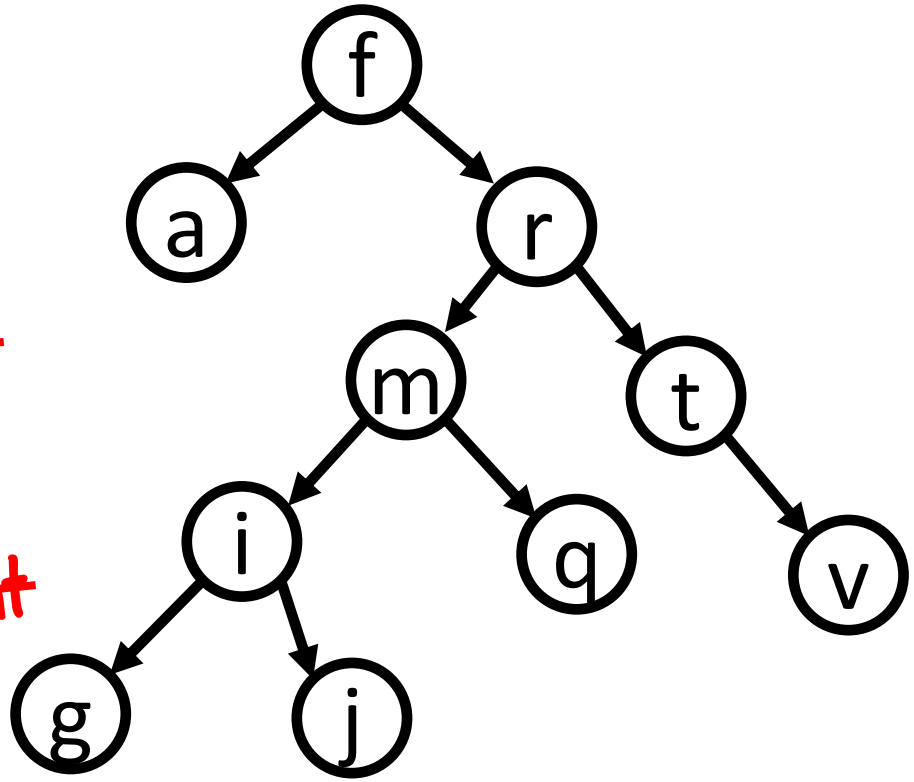
Example: Insert

Zag-zig
1. ~~Rotate on parent~~
2. ~~Rotate on new parent~~

Zag-zag
1. ~~Rotate on grandparent~~
2. Rotate on parent



Example: Insert



Zag-zig

1. ~~Rotate on parent~~
2. ~~Rotate on new parent~~

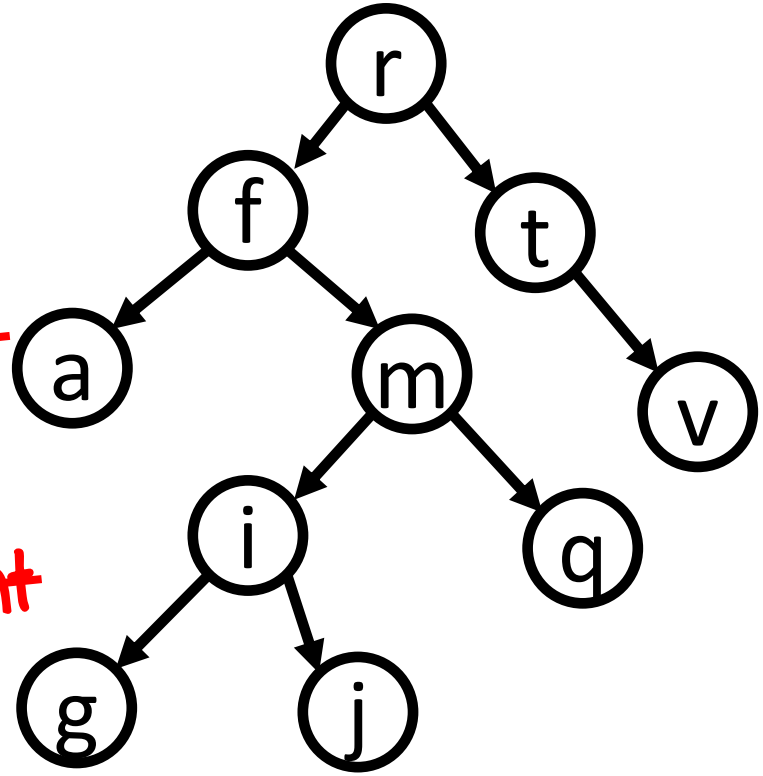
Zag-zag

1. ~~Rotate on grandparent~~
2. ~~Rotate on parent~~

Zag

1. Rotate on parent

Example: Insert



Zag-zig

1. ~~Rotate on parent~~
2. ~~Rotate on new parent~~

Zag-zag

1. ~~Rotate on grandparent~~
2. ~~Rotate on parent~~

Zag

1. ~~Rotate on parent~~

Summary

- Splay Operation
  - Move the given node to the root
  - Use zig-zig and zag-zag rotations when possible
- Insert
  - Insert as usual
  - Splay new node
- Search
  - Search as usual
  - Splay found node

Coming Up…

Quinn Returns!

Banker’s Method: Intro

Recall: Array-Backed List

- Amortized $O(1)$ append
- How did we prove it?
  - Analyzed worst-case complexity of m appends
  - Divided by m to find time per operation

Proving Amortized Bounds

- Aggregate Method
  - Analyze worst-case complexity of m operations
  - Divide by m for get time per operation
- Limitations
  - What if appending and popping in an array-backed list?
  - What if inserting and searching in a splay tree?
  - Hard to analyze every possible sequence of operations.

Proving Amortized Bounds

- Aggregate Method
  - Analyze worst-case complexity of m operations
  - Divide by m for get time per operation
- Banker's Method (Accounting Method)
  - Treat time like money – allowed to save it for later
  - "Overcharge" for cheap operations, and save the extra
  - Show that there is always enough "in the bank" to pay for expensive operations
- Physicist's Method (Potential Method)
  - Maybe when you're older…

Revenge of the Train: The Quinnening

- The train is just like an array-backed list!
- Each constant-time step costs 1 coin
  - “Loading” a “box” (putting an item in the array)
  - “Moving” a “box” (copying an item to the new array)
- How much should Quinn charge each “customer” (call to append)?
  - 1 coin per append
    - Pays for loading the box, but not for copying it
  - 2 coins per append
    - Pays for loading the box and copying the first time, but not the second time!
  - 3 coins per append
    - Seems to be enough…

Summary

- Aggregate method
  - Analyze sequence of operations
  - Find per-operation time
- Not always easy when there are many possible sequences
- Banker's method
  - "Overcharge" for cheap operations, save the extra
  - Show that there is always enough "money" for expensive operations
- Train example
  - Charging 3 coins per append seems to be enough
  - Constant price means amortized $O(1)$ time!

Coming Up…

Proving that 3 coins per append is enough

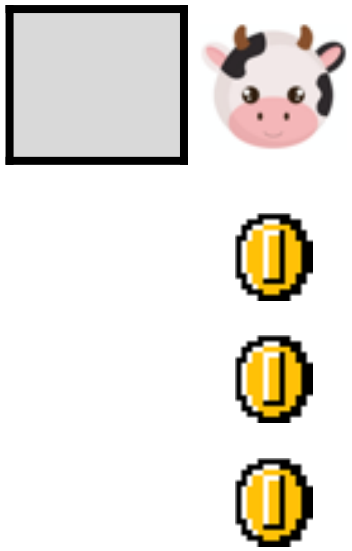
Banker’s Method: Details

Recall: Proving Amortized Bounds

- Aggregate Method
  - Analyze worst-case complexity of m operations
  - Divide by m for get time per operation
- Banker's Method (Accounting Method)
  - Treat time like money – allowed to save it for later
  - "Overcharge" for cheap operations, and save the extra
  - Show that there is always enough "in the bank" to pay for expensive operations
- Physicist's Method (Potential Method)
  - Maybe when you're older…

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough



Paying for operations on item i:

- "Load"
  - Use first coin

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough

Paying for operations on item i:

- "Load"
  - Use first coin

Banker's Method: Append/Expansion

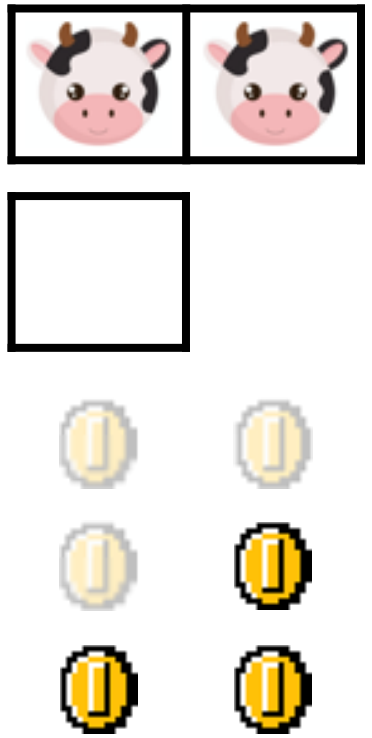
- Show that charging 3 "coins" per append is enough

Paying for operations on item i:

- "Load"
  - Use first coin

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough

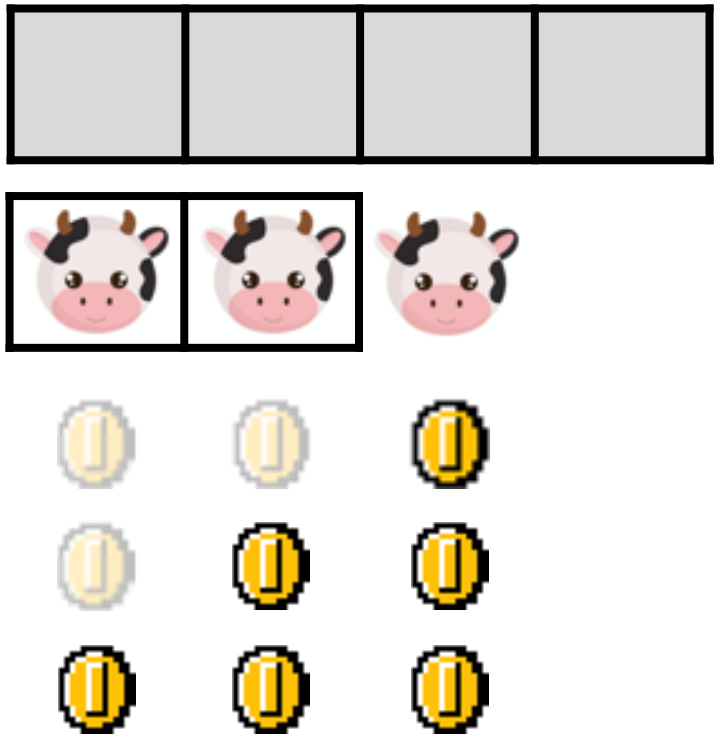


Paying for operations on item i:

- "Load"
  - Use first coin
- 1st "Move"
  - Use second coin

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough

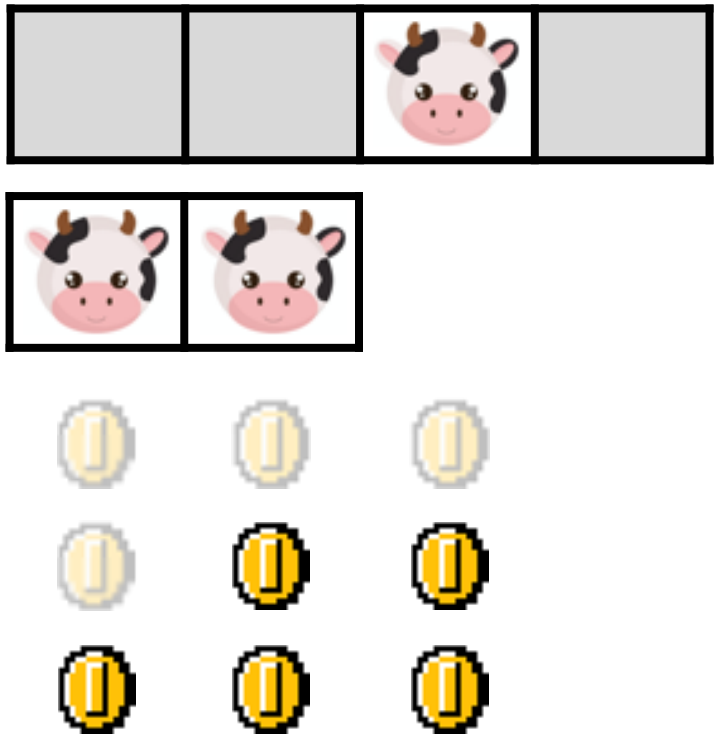


Paying for operations on item i:

- "Load"
  - Use first coin
- 1st "Move"
  - Use second coin

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough

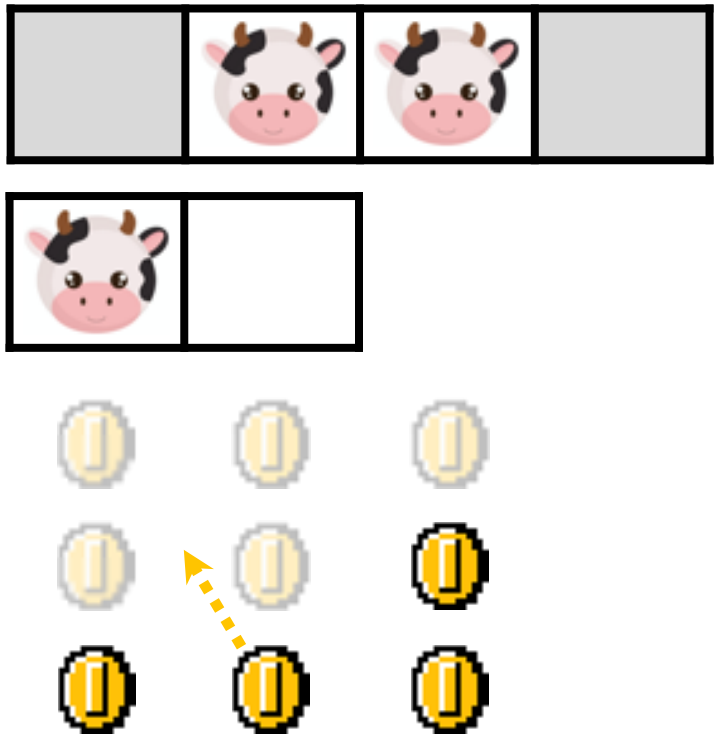


Paying for operations on item i:

- "Load"
  - Use first coin
- 1st "Move"
  - Use second coin

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough

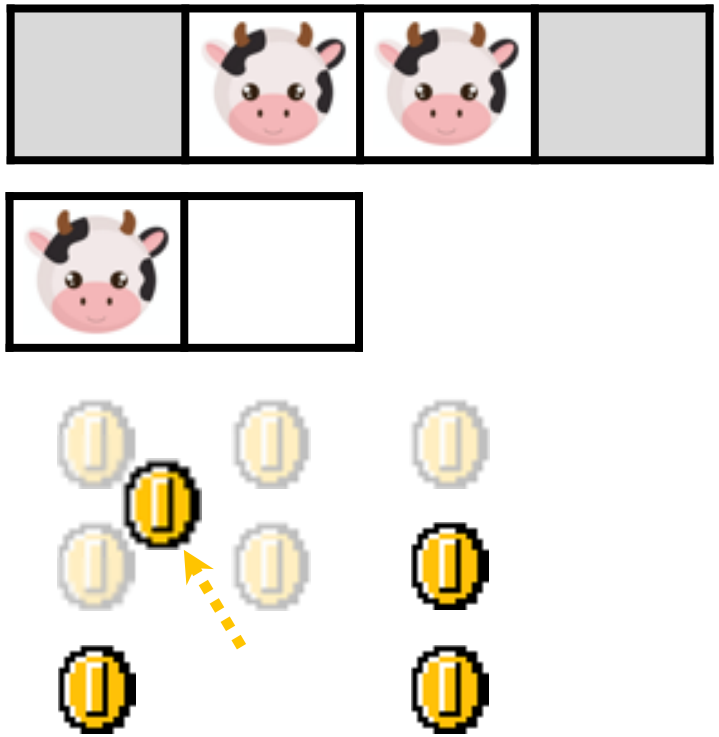


Paying for operations on item i:

- "Load"
  - Use first coin
- 1st "Move"
  - Use second coin

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough



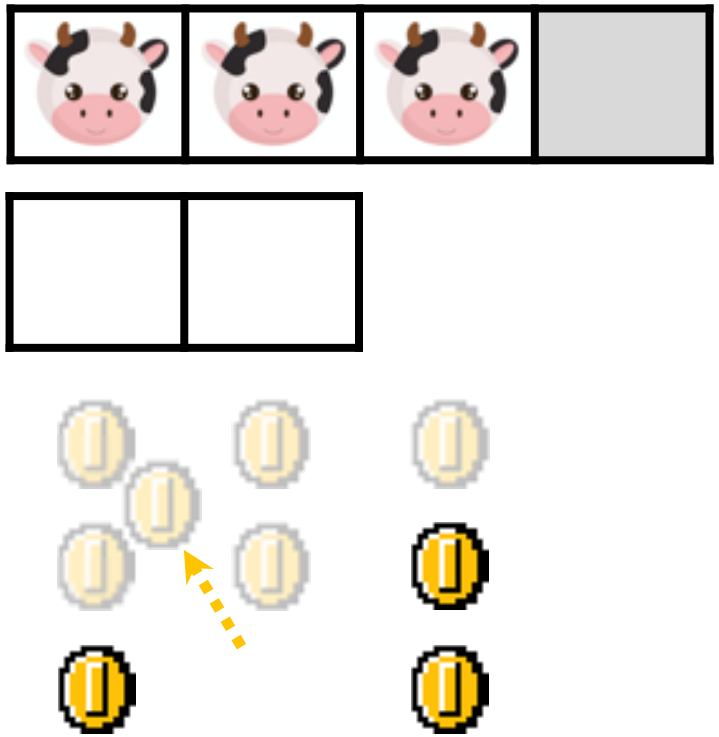
Paying for operations on item i:

- "Load"
  - Use first coin
- 1st "Move"
  - Use second coin
- All other "Moves"
  - Use third coin from item $n/2 + i$

This coin has been deposited because $n/2$ items have been appended since last resize!

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough

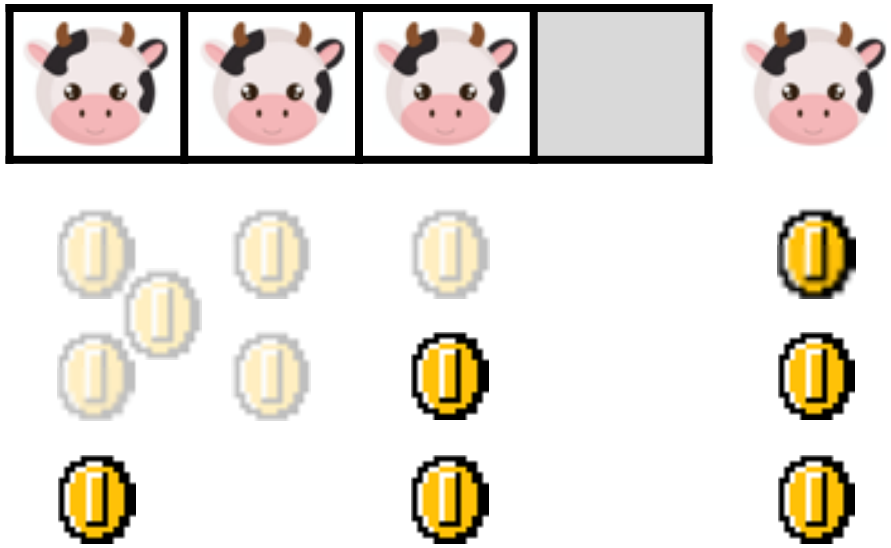


Paying for operations on item i:

- "Load"
  - Use first coin
- 1st "Move"
  - Use second coin
- All other "Moves"
  - Use third coin from item $n/2 + i$

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough

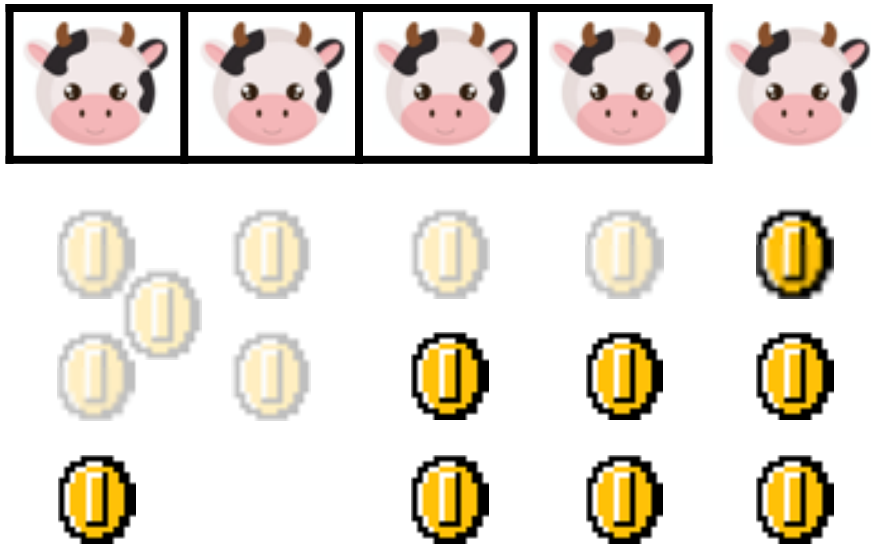


Paying for operations on item i:

- "Load"
  - Use first coin
- 1st "Move"
  - Use second coin
- All other "Moves"
  - Use third coin from item $n/2 + i$

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough

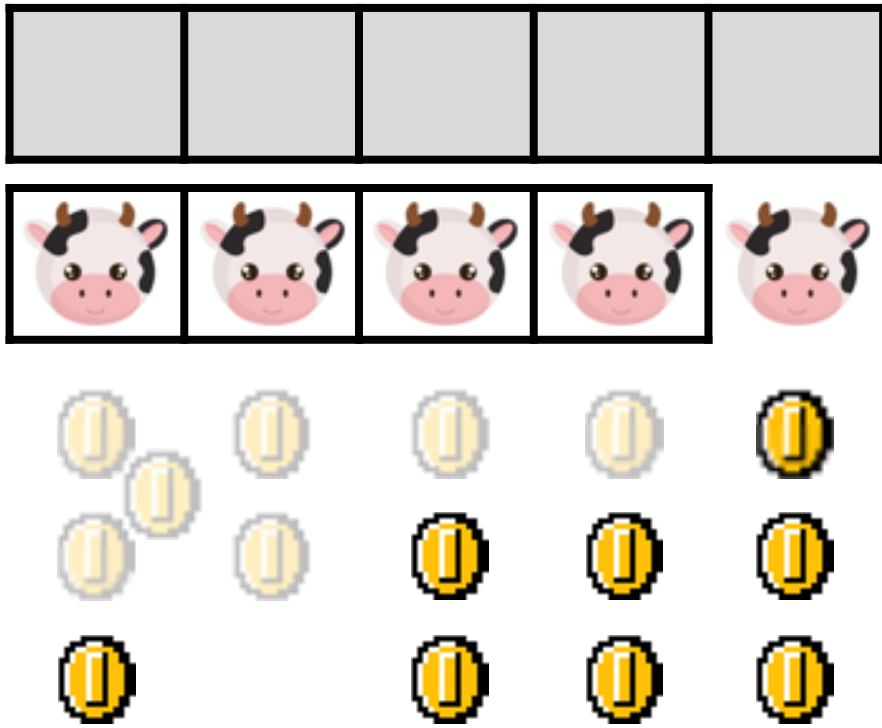


Paying for operations on item i:

- "Load"
  - Use first coin
- 1st "Move"
  - Use second coin
- All other "Moves"
  - Use third coin from item $n/2 + i$

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough

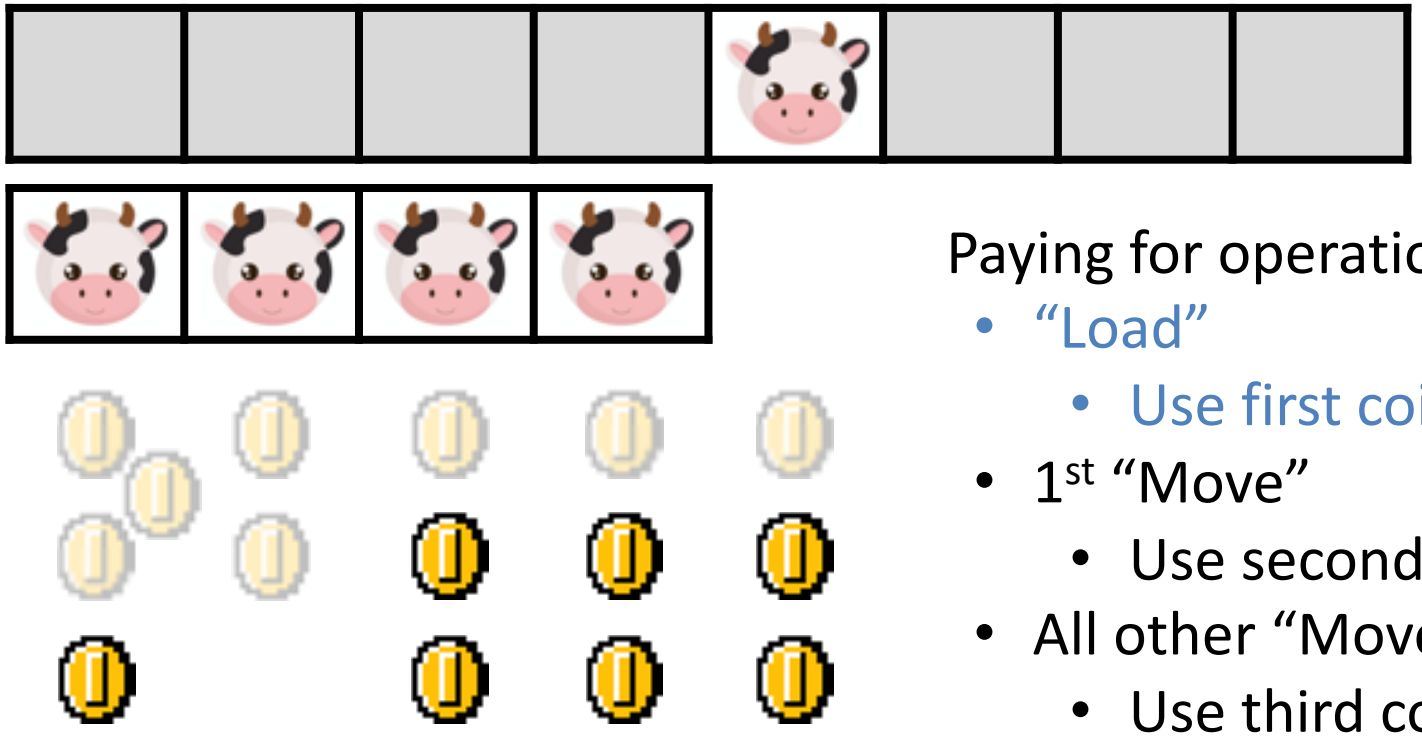


Paying for operations on item i:

- "Load"
  - Use first coin
- 1st "Move"
  - Use second coin
- All other "Moves"
  - Use third coin from item $n/2 + i$

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough

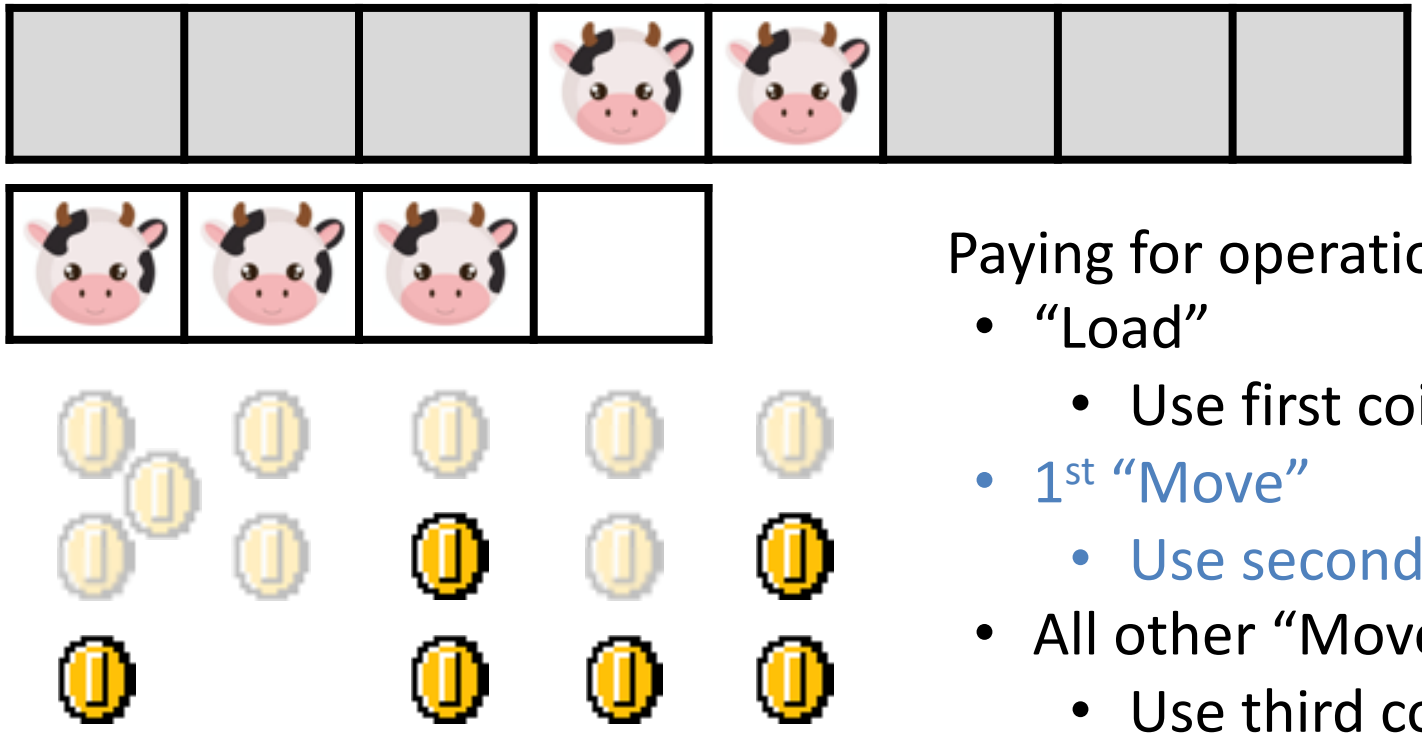


Paying for operations on item i:

- "Load"
  - Use first coin
- 1st "Move"
  - Use second coin
- All other "Moves"
  - Use third coin from item $n/2 + i$

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough

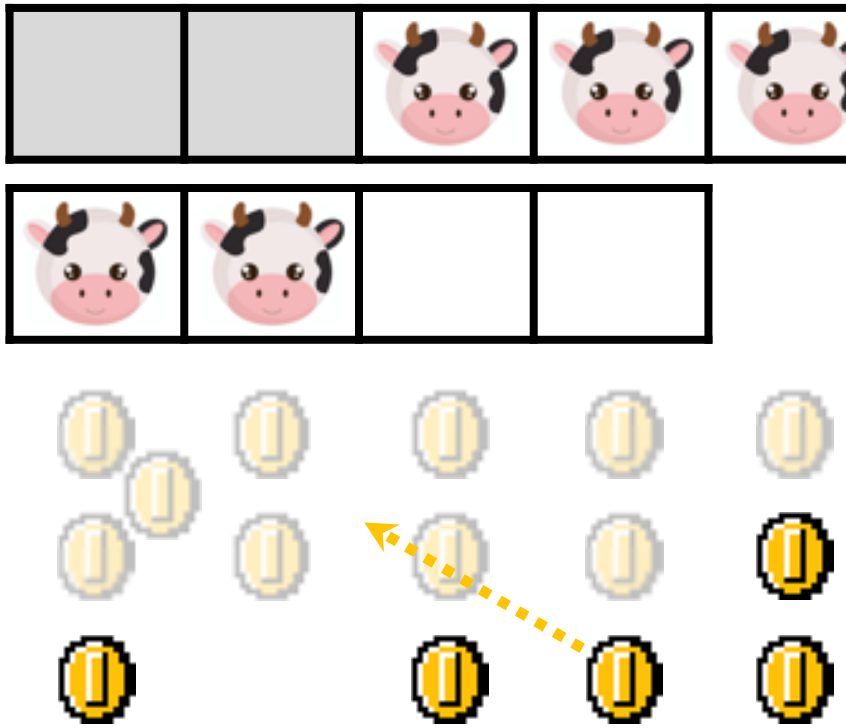


Paying for operations on item i:

- "Load"
  - Use first coin
- 1st "Move"
  - Use second coin
- All other "Moves"
  - Use third coin from item $n/2 + i$

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough

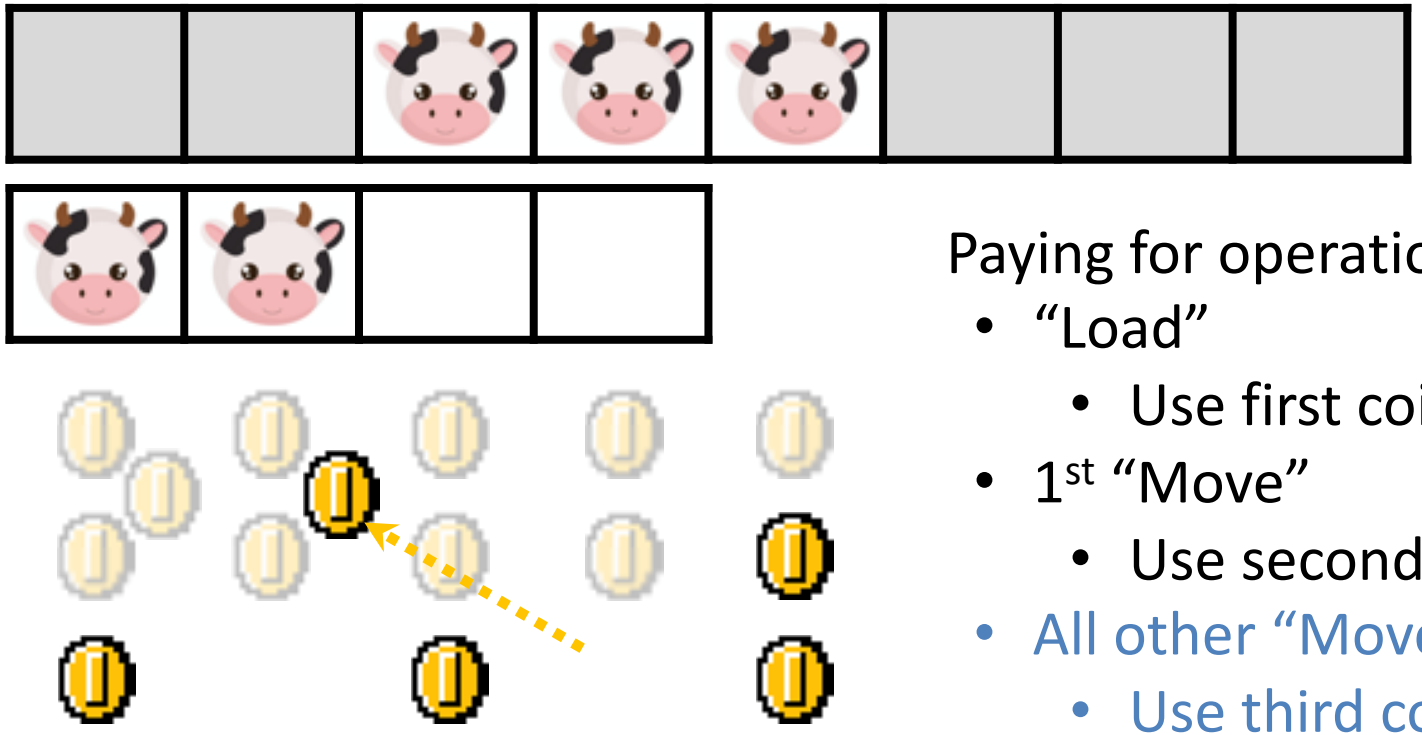


Paying for operations on item i:

- "Load"
  - Use first coin
- 1st "Move"
  - Use second coin
- All other "Moves"
  - Use third coin from item $n/2 + i$

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough

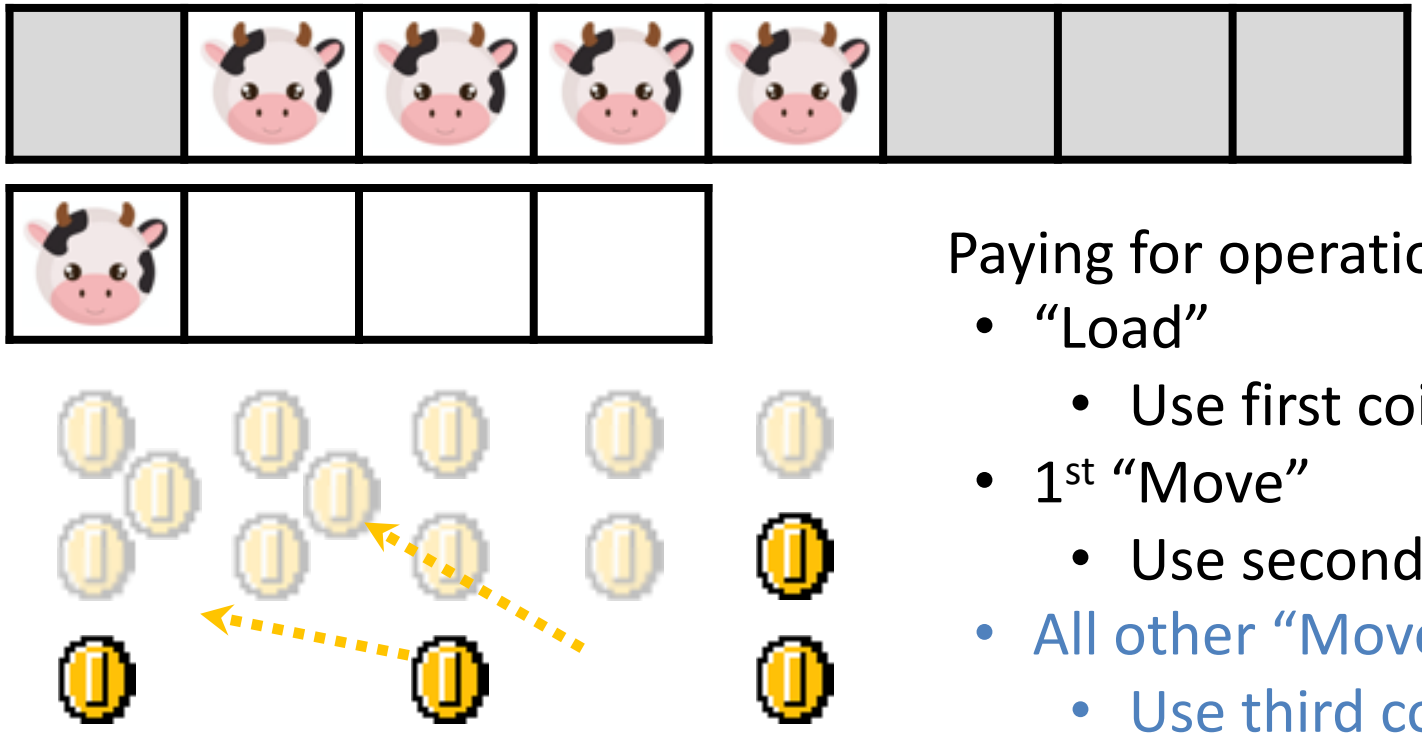


Paying for operations on item i:

- "Load"
  - Use first coin
- 1st "Move"
  - Use second coin
- All other "Moves"
  - Use third coin from item $n/2 + i$

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough

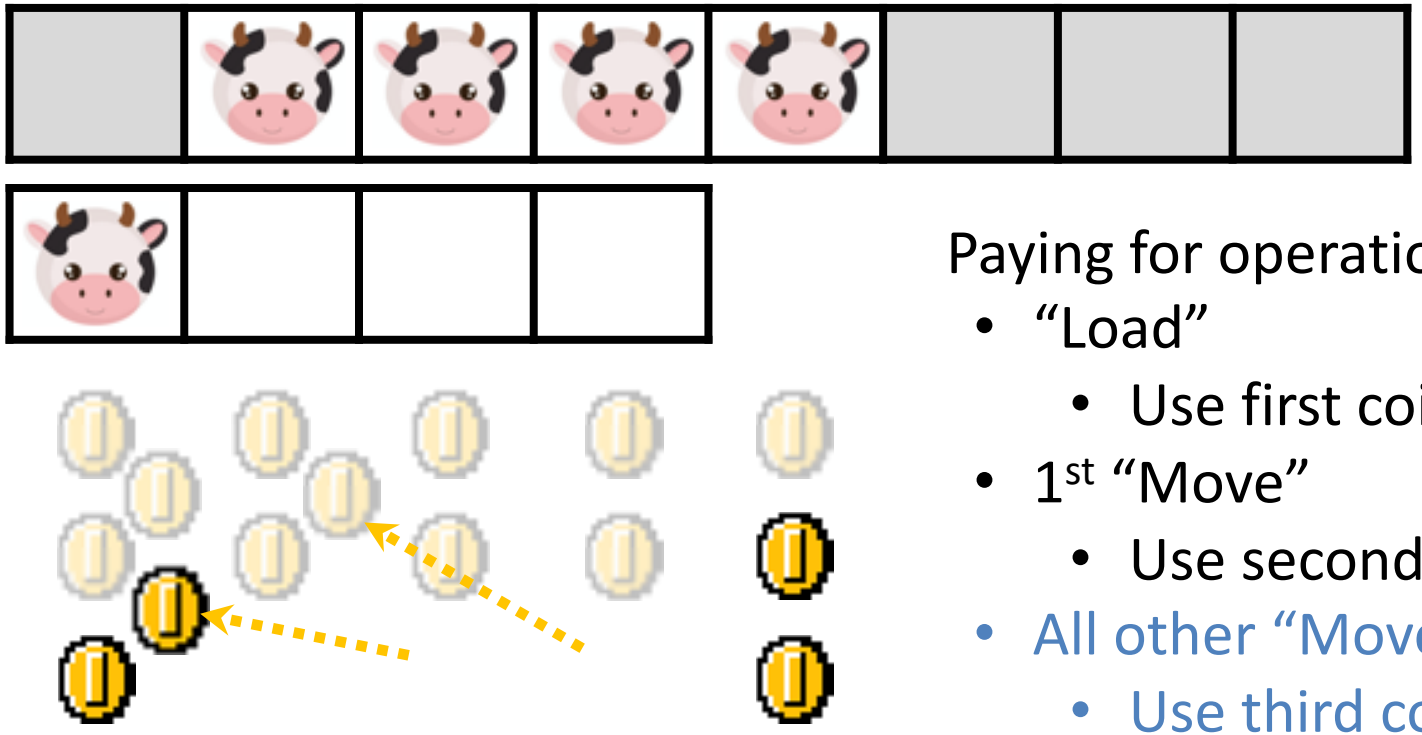


Paying for operations on item i:

- "Load"
  - Use first coin
- 1st "Move"
  - Use second coin
- All other "Moves"
  - Use third coin from item $n/2 + i$

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough

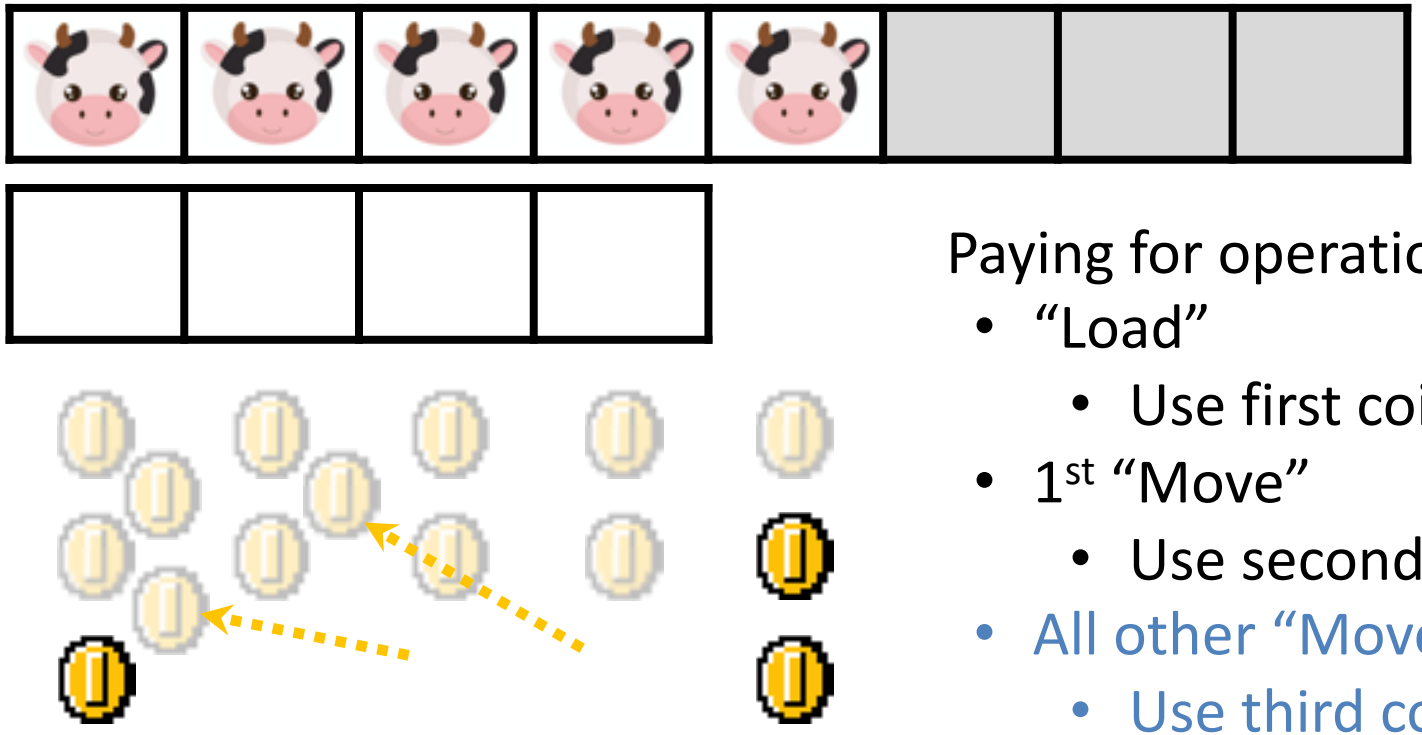


Paying for operations on item i:

- "Load"
  - Use first coin
- 1st "Move"
  - Use second coin
- All other "Moves"
  - Use third coin from item $n/2 + i$

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough

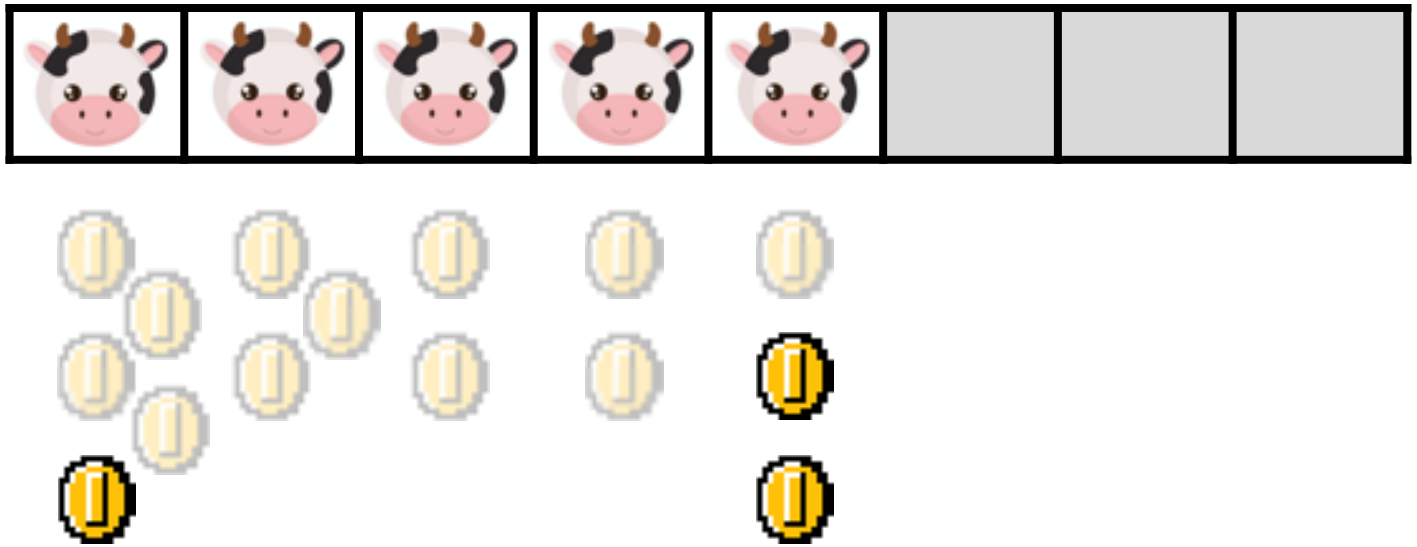


Paying for operations on item i:

- "Load"
  - Use first coin
- 1st "Move"
  - Use second coin
- All other "Moves"
  - Use third coin from item $n/2 + i$

Banker's Method: Append/Expansion

- Show that charging 3 "coins" per append is enough
  - Since we can always pay for loading and moving each item, 3 "coins" is enough
  - Since we can charge a constant price for each append, append has amortized $O(1)$ complexity



Banker's Method: Pop/Shrink

- What about popping from the back, and shrinking the array?
  - Recall: we shrink when $size = capacity/4$
- How much should we charge for a pop operation?
  - Charge 2 coins for each pop
  - Paying for operations
    - Remove last item: use the first coin from the pop
    - Copy item i to smaller array: use the second coin from popping item $n + i$
  - Why can we guarantee that the copy coin has been deposited?
    - When there have been no pops since last resize, $size \geq capacity/2$
    - So when we shrink, we must have popped at least half of the items since the last resize
    - If size at shrinking is n, then we have popped at least n items.
  - Since we can charge $O(1)$ coins, pop has $O(1)$ amortized time

Splay Trees?

- High-level:
  - For each splay (insert/search) charge a price that is logarithmic in n
  - Show that you can always pay for every step of a splay operation
- We won't go through the argument in CS70
  - But you can find it onli
  - (e.g. https://www.cs.c s/lec04.pdf)

Summary

- Using the Banker's method
  - Set a "price" for each operation
    - The price may depend on n!
  - Show how to pay for each $O(1)$ step
    - Typically using "coins" saved up from earlier operations
  - The price gives the amortized time for each operation
  - The operations can be interleaved in any sequence because they are "paid for"!
- Array-backed lists
  - Append and pop (from back) both have amortized $O(1)$ complexity
- Splay trees
  - Insert and search both have amortized $O(\log n)$ complexity