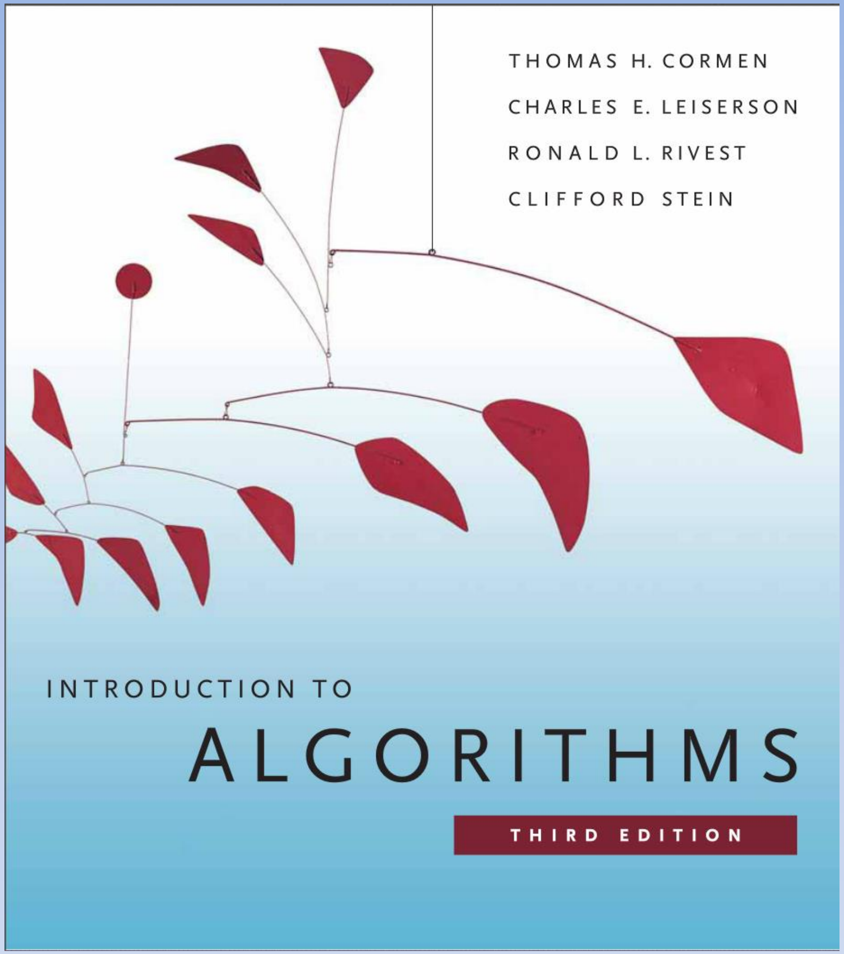


174
CH17: Amortized Algorithms

- Tighter bounds by looking at a sequence of operations and averaging

Chapter 17

IV Advanced Design and Analysis Techniques

Stacks w/ Augmented Operation

- Consider a Stack:
  - Push(S, x): pushes object x onto stack S
  - Pop(S): pops the top of the stack S and returns the popped object (returning error if stack is empty).
- Each stack operation runs in O(1) time.
- We can think of these ops as having cost 1.
- Cost of a sequence of n Push and Pop operations is thus n.

Stacks

- push(S, 2)

S:

- 2

Stacks

- push(S, 2)
- push(S, 1)

S:

- 1
- 2

Stacks

- push(S, 2)
- push(S, 1)
- push(S, 3)

S:

- 3
- 1
- 2

Stacks

- push(S, 2)
- push(S, 1)
- push(S, 3)
- Pop(S)

S:

- 1
- 2

Stacks

- push(S, 2)
- push(S, 1)
- push(S, 3)
- Pop(S)
- Push(S, 7)

S:

- 7
- 1
- 2

Stacks

- push(S, 2)
- push(S, 1)
- push(S, 3)
- Pop(S)
- Push(S, 7)
- Pop(S)
- Pop(S)

S:

- 2

Stacks
How much work have we done?

- push(S, 2)
- push(S, 1)
- push(S, 3)
- Pop(S)
- Push(S, 7)
- Pop(S)
- Pop(S)

S:

- 2

Stacks
How much work have we done?

- push(S, 2)
- push(S, 1)
- push(S, 3)
- Pop(S)
- Push(S, 7)
- Pop(S)
- Pop(S)

- **7 Units of Work Total**

S:

- 2

Multipop(S, k)

- Now we augment our stack with a new operation:
- Multipop(S, k): removes the k top objects of stack S. If stack has fewer than k object, the stack is emptied.

Multipop(S, k)

MULTIPOP(S, k)

```
while not STACK-EMPTY(S) and k > 0
    POP(S)
    k = k - 1
```

a) Multipop(S, 4)
b) Multipop(S, 7)

Multipop(S, k)

```
MULTIPOP(S, k)
while not STACK-EMPTY(S) and k > 0
    POP(S)
    k = k - 1
```

- Cost of Multipop(S, k) = min(s, k)
  - s=# of objects on stack

Augmented Stack w/ Worst-Case Analysis

- Push & Pop are O(1)
- How about Multipop(S, k)???
- Worst-Case w/ Multipop:
  - n items in stack
  - n items pop from stack with Multipop(S, n)
  - Cost of one Multipop is O(n)
- THEREFORE Worst-Case w/ Augmented Stack is $O(n^2)$
  - n operations of Multipop worst-case cost of n

Worst-Case w/ Observation

- Worst-Case requires n Multipop operations, with each popping n items.
  - Where did these n^2 items come from????
- NOTE: each item can only be popped once per time its pushed.
  - ≤ n Push's => ≤ n Pop's
    - Including those in Multipop

- Although our worst-case analysis is correct,

  The $O(n^2)$ result, obtained by considering worst-case cost of each operation individually,

  IS NOT TIGHT!

Augmented Stack w/ Aggregate Analysis

- Aggregate Analysis , we show that for all n, a sequence of n operations takes worst-case time T(n) in total!
  - Here we are looking at a block of operations rather than operations individually.
- Amortized Cost is calculated per operation as T(n)/n
  - Total cost averaged over the number of operations!

Augmented Stack w/ Aggregate Analysis

- Remember: each item can only be popped once per time its pushed.
  - ≤ n Push's => ≤ n Pop's
    - Including those in Multipop
- So the maximum cost possible w/ n operations is n.
  - T(n) = n
- So the amortized cost per operation is:
  - T(n)/n = O(1)!!!
    - NOT O(n) like worst-case for Multipop

Aggregate Analysis

- Aggregate Analysis assigns amortized cost of each operation as the average cost.
  - $T(n) / n$
- BUT: this is not a type of probabilistic analysis.
- $T(n)$ is a worst-case bound on n operations.

Incrementing a Binary Counter

- Implement a k-bit binary counter!
- array A[0..k-1]
  - A.length = k
  - A[0] is low-order bit
  - A[k-1] is high-order bit

$$x = \sum_{i=0}^{k} A[i] * 2^i$$

Binary Function w/ Python

```
>>> print '\n'.join(
    ["(%i,%s)"%(i,bin(i)) for i in range(11)])
(0,0b0)
(1,0b1)
(2,0b10)
(3,0b11)
(4,0b100)
(5,0b101)
(6,0b110)
(7,0b111)
(8,0b1000)
(9,0b1001)
(10,0b1010)
```

- NOTE: Low-order bit is A[n]

Binary Function w/ Python

- To Match Textbook use a few utilities:

```
btoa = lambda x:''.join(x[::-1])
btoi = lambda x:int(''.join(x[::-1]),2)
itob = lambda x:bin(x)[:1:-1]
```

Increment(A)

```
INCREMENT(A)
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

- Loop 2-4 wants to add 1 to into position i
- If A[i] is already 1, then adding one flips it to 0.
  - Requiring carry add 1 into position i+1
- If A[i] is 0 loop exits w/ 1 inserted into A[i] in line 6.
  - Otherwise now A[i] = 0 for all i<A.length & x=0

Increment(A) w/

bincount.py - D:\Documents\GitHub\ics1293\Code\py\bincount.py

File Edit Format Run Options Windows Help

```python
#binary operations w/ binary counter
btoa = lambda x:''.join(x[::-1])
btoi = lambda x:int(''.join(x[::-1]),2)
itob = lambda x:bin(x)[:1:-1]

def incrementBinaryCounter(A):
    ''' High Order Bit in A[0]'''
    i = 0
    while i < len(A) and A[i]=='1':
        A[i] = '0'
        i += 1
    if i < len(A):
        A[i] = '1'

A=list('00000000')

for i in range(20):
    print "%2i = %s (%2i)"%(i,btoa(A),btoi(A))
    incrementBinaryCounter(A)
```

Ln: 17 Col: 17

```
 0 = 00000000 ( 0)
 1 = 00000001 ( 1)
 2 = 00000010 ( 2)
 3 = 00000011 ( 3)
 4 = 00000100 ( 4)
 5 = 00000101 ( 5)
 6 = 00000110 ( 6)
 7 = 00000111 ( 7)
 8 = 00001000 ( 8)
 9 = 00001001 ( 9)
10 = 00001010 (10)
11 = 00001011 (11)
12 = 00001100 (12)
13 = 00001101 (13)
14 = 00001110 (14)
15 = 00001111 (15)
16 = 00010000 (16)
17 = 00010001 (17)
18 = 00010010 (18)
19 = 00010011 (19)
```

Binary Counter w/ Worst-Case Analysis

- In the worst case, Increment needs to flip all k bits.
  - ‘11111111’ => ‘00000000’
- So worst-case cost for n Increments in O(nk)
- Clearly, here again, this is not possible!

- We can Tighten our Analysis!

```
 0 = 00000000 ( 0)
 1 = 00000001 ( 1)
 2 = 00000010 ( 2)
 3 = 00000011 ( 3)
 4 = 00000100 ( 4)
 5 = 00000101 ( 5)
 6 = 00000110 ( 6)
 7 = 00000111 ( 7)
 8 = 00001000 ( 8)
 9 = 00001001 ( 9)
10 = 00001010 (10)
11 = 00001011 (11)
12 = 00001100 (12)
13 = 00001101 (13)
14 = 00001110 (14)
15 = 00001111 (15)
16 = 00010000 (16)
17 = 00010001 (17)
18 = 00010010 (18)
19 = 00010011 (19)
```

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Increment w/ Observation

- Not every bit flips everytime!

Bit	Flips how often	Times in n Increments
0	Every time	N
1	½ the time	$\lfloor n/2 \rfloor$
2	¼ the time	$\lfloor n/2^2 \rfloor$
i	$^1/_{2^i}$ the time	$\lfloor n/2^i \rfloor$
i ≥ k	never	0

Increment w/ Observation

- Therefore, total # of flips =

$$\sum_{i=0}^{k-1} \lfloor n/2^i \rfloor$$

$$n \sum_{i=0}^{\infty} 1/2^i$$

- n * (1 / (1 - ½)) = 2n
- SO: T(n) = 2n
  - so average cost per operation is O(1)

Exercise 17.1-1

- If the set of stack operations included a Multipush operation, which pushes k items onto the stack, would the O(1) bound on the amortized cost of stack operations continue to hold?

Exercise 17.1-2

- Show that if a Decrement operation were included in the k-bit counter example, n operations could cost as much as $\Theta(nk)$ time

Exercise 17.1-3

- Suppose we perform a sequence of n operations on a data structure in which the ith operation costs i if i is an exact power of 2, and 1 otherwise.
- Use aggregate analysis to determine the amortized cost per operation.

Exercise 17.1-3

- Let c_i = cost of the ith operation
- c_i =
  - i if i is an exact power of 2
  - 1 otherwise

Operation	Cost
1	1
2	2
3	1
4	4
5	1
6	1
7	1
8	8
9	1
10	1

Exercise 17.1-3

- How many operations between 1 and n will be an exact power of 2?

- How many powers of 2 between 1 and n?

- $2^0, 2^1, 2^2, \ldots, 2^{\lg n}$

- $? = 2^0 + 2^1 + 2^2 + \ldots + 2^{\lg n}$
- Total Cost $T(n) = n + (2n-1) < 3n$

Exercise 17.1-3

- How many operations between 1 and n will be an exact power of 2?
- How many powers of 2 between 1 and n?
- $2^0, 2^1, 2^2, \ldots, 2^{\lg n}$
- $? = 2^0 + 2^1 + 2^2 + \ldots + 2^{\lg n}$
- $= 2^{\lg n+1} - 1 = 2 * 2^{\lg n} - 1 = 2n-1$
- Total Cost $T(n) = n + (2n-1) < 3n$

Amortized Analysis w/ Accounting Method

- Different charges assigned to different operations
  - Some are charged more than their actual cost
  - Some are charged less
- Amortized cost is the amount charged

Accounting Method w/ Amortized Costs

- Amortized Costs when exceeding actual costs generate credit.
- Credits are stored and used later to pay for operations that are more expensive than their amortized cost.
- Credit must never go negative!

Amortized Costs

- Amortized Costs must be chosen carefully!
- Must ensure that total amortized cost of a sequence of operations provides an upper bound on total actual cost.
- MOREOVER: Total Amortized Cost must be an upper bound for all possible sequences!

Amortized Costs
Equation 17.1

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

- c_i : actual cost of the i^{th} operation
- $\hat{c}_i$: amortized cost of the i^{th} operation

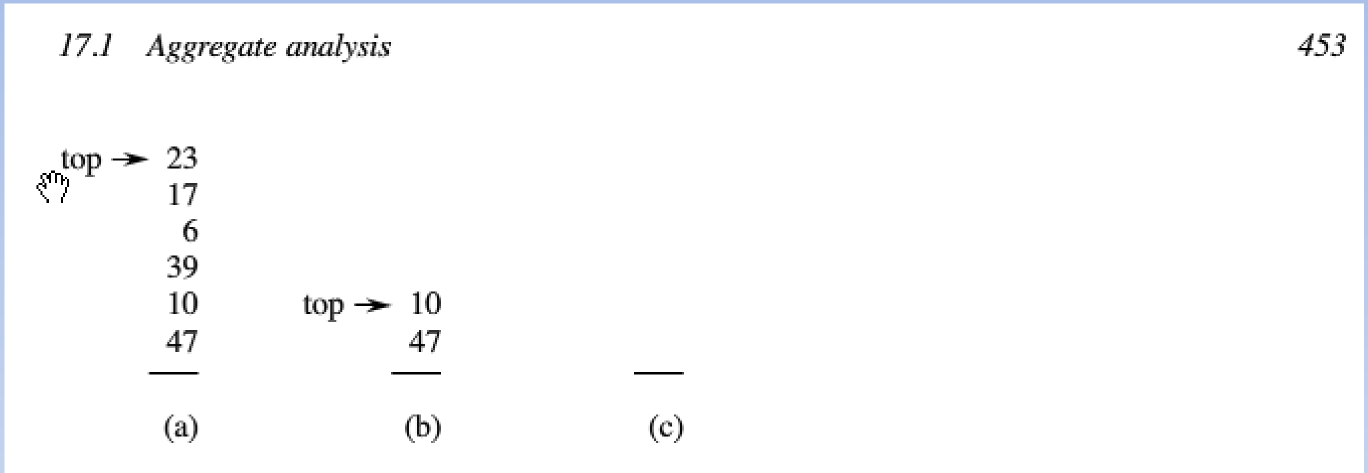
Actual cost can never exceed Amortized Cost

$$\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$$

- c_i : actual cost of the i^{th} operation
- $\hat{c}_i$: amortized cost of the i^{th} operation

- If credit is negative, algorithm is in debt!
- While the algorithm is in debt, Amortized Cost is not an upper bound on performance!

Remember Stack w/ Multipop!



- (a) Multipop(S, 4); (b) Multipop(S, 7)
- Actual Costs:
  - Push=1, Pop=1, Multipop=min(s,k)
    - s=stack size, k=multipop parameter
- Let’s utilize the following Amortized Costs
  - Push=2, Pop=0, Multipop=0

Plates @ Cafeteria

- Represent our Stack w/ Plates in dispenser.
- Each time we add a plate we spend 1 dollar and put 1 dollar on the plate!

- The dollar on the plate is prepayment for the cost of later popping it!

Plates @ Cafeteria

- The most our n operations will cost is 2n.
- No sequence of operations can ever create a debt.

- So we can conclude that Total Amortized Cost T(n) = O(n) is an upper bound.

Bin Counter

- Running time is proportional to the number of flipped bits!
- Dollar Bills will represent unit cost!

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Bin Counter w/ Accounting Method

- Amortized Costs:
  - Each time we set a bit to 1 we'll charge \$2 !
    - Actual Cost is One Dollar!
    - One Dollar of the Two dollars charged used to cover actual cost!
    - Second dollar stored on bit as credit.
  - Setting a bit to 0 has cost 0!
    - When bit set 0 we'll use the One Dollar stored on bit.

Bin Counter w/ Accounting Method

- NOTE: Initially all bits are 0.
- At any point in time, every 1 in the counter has a dollar of credit on it.
- Thus, we can charge nothing to reset a bit to 0
  - we just pay for the reset with bill on bit!!

Increment(A)

```
INCREMENT(A)
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

- Increment Calls set at most one bit (line 6)!
  - Costing 2 dollars !!
- The number of 1s in the counter never becomes negative
  - thus the amount of credit stays nonnegative.
- Thus n Increment operations have Total Amortized Cost $T(n) < 2n = O(n)$

Exercise 17.2-1

- Suppose we perform a sequence of stack operations on a stack whose size never exceeds k.
  - After every k operations, we make a copy of the entire stack for backup purposes.
- Show that the cost of n stack operations, including copy the stack, is O(n) by assigning suitable amortized costs to the various stack operations.

Exercise 17.2-1

- Amortized Costs:
  - Push: \$2
  - Pop: \$2
- For each Push and Pop:
  - one dollar used to pay for operation.
  - one dollar saved in stack.
  - After k operations there will always be \$k to pay for copy!

Exercise 17.2-2

- Redo Exercise 17.1-3 using accounting method of analysis.

Exercise 17.2-2

- Let c_i = cost of the ith operation
- c_i =
  - i if i is an exact power of 2
  - 1 otherwise

Operation	Cost
1	1
2	2
3	1
4	4
5	1
6	1
7	1
8	8
9	1
10	1

Exercise 17.2-2

- How many operations between 1 and n will be an exact power of 2?

- How many powers of 2 between 1 and n?

- $2^0, 2^1, 2^2, \ldots, 2^{\lg n}$

- $? = 2^0 + 2^1 + 2^2 + \ldots + 2^{\lg n}$
- $= 2^{\lg n+1} - 1 = 2 * 2^{\lg n} - 1 = 2n-1$
- Total Cost $T(n) = n + (2n-1) < 3n$

Exercise 17.2-2

- Charge \$3 for each operation
  - if i is not an exact power of 2, pay one dollar
  - if i is an exact power of 2, pay i dollars using stored credit!

Operation	Cost	Actual Cost	Credit Remaining
1	3	1	2
2	3	2	3
3	3	1	5
4	3	4	4
5	3	1	6
6	3	1	8
7	3	1	10
8	3	8	5
9	3	1	7
10	3	1	9

Exercise 17.2-2

- Charge \$3 for each operation
  - if i is not an exact power of 2, pay one dollar
  - if i is an exact power of 2, pay i dollars using stored credit!

- Total Amortized Cost for n operation $T(n)=3n$
- Total Actual Cost $T(n) = n + (2n-1) < 3n$
- Since :
  - the amortized cost of each operation is $O(1)$,
  - and the amount of credit never goes negative,
  - the total cost of n operations is $O(n)$

Amortized Analysis w/ Potential Method

- Like accounting method, but credit thought of as potential (like physics) stored with the entire data structure.
  - Accounting method stores credit w/ specific objects
    - plates in stack
    - bits in counter
  - Potential method stores potential in data structure as a whole.
    - Potential can be released to pay for future operations
    - Most Flexible of amortized analysis methods!

Potential Method

- D_i = Data Structure after the i^{th} operation
- D_0 = the initial data structure
- c_i = actual cost of the i^{th} operation
- $\widehat{c}_i$ = amortized cost of the i^{th} operation

- Potential Function: $\Phi(D_i) \rightarrow \mathbb{R}$
  - $\Phi(D_i)$ is the potential associated w/ data structure D_i

Potential Method

- $\widehat{c_i} = c_i + \Phi(Di) - \Phi(Di_{-1})$
- $\Delta\Phi(Di) = \Phi(Di) - \Phi(Di_{-1})$
  - Increase (or decrease) in potential due to i^th operation

Total Amortized Cost T(n) w/ Potential Method

$$T(n) = \sum_{i=1}^{n} \widehat{c}_i$$

$$= \sum_{i=1}^{n} (c_i + \Phi(\mathrm{D}_i) - \Phi(\mathrm{D}_{i-1}))$$

Telescoping sum

$$= \sum_{i=1}^{n} c_i + \Phi(\mathrm{D}_n) - \Phi(\mathrm{D}_0)$$

- Φ(Dn) $\geq \Phi(\mathrm{D}_0)$, for all i
  - Amortized Cost an upper bound

Augmented Stack w/ Potential Method

- $\Phi(Di)$ = is the number of items in the stack w/ i^{th} operation.
- $\Phi(D_0) = 0$
- $\Phi(Di) \geq 0$

Augmented Stack w/ Potential Method

Operation	Actual Cost	$\Delta\Phi(D_i)$	Amortized Cost
Push	1	$(s+1) - s = 1$	$1 + 1 = 2$
Pop	1	$(s-1) - s = -1$	$1 - 1 = 0$
Multipop(S, k)	$k' = \min(k,s)$	$(s-k') - s = -k'$	$k' - k' = 0$

- Amortized cost of n operations
  - $T(n) < 2n = O(n)$

Bin Counter w/ Potential Method

- Now lets look @ Bin Counter again.
- $\Phi(D_i) = b_i$ is the number of ones in counter after i^{th} Increment Operation.
- Now using this Phi Function we can look at the amortized cost of an Increment Operation.

Bin Counter w/ Potential Method

- $\Phi(D_i)=b_i$ is the number of ones in counter after i^{th} Increment Operation.
- Say i^{th} increment operation resets t_i bits
  - $c_i = t_i + 1$
    - Since in addition to resetting the t_i bits it also potentially sets one bit.
- If $b_i = 0$, all bits are 0 (no 1's)
  - THEN b_{i-1} must have just reset all k bits
  - $b_{i-1} = t_i = k$ (all 1's)

Bin Counter w/ Potential Method

- If b_i>0, Then some number of bits are set at time i.
  - THEN b_i =
    - b_{i-1}: the number of bits set before this operation
    - $-t_i$: decreased by the number of bits that were just reset.
    - +1: Plus one more for the possible bit just set.
- So for both cases:

$$b_i \leq b_{i-1} - t_i + 1$$

Bin Counter w/ Potential Method

- So for both cases:

$$b_i \leq b_{i-1} - t_i + 1$$

- $\Delta\Phi(\mathrm{D}_i) = (b_{i-1} - t_i + 1) - b_{i-1}$
- SO: $\Delta\Phi(\mathrm{D}_i) = 1 - t_i$

- $\widehat{c_i} = c_i + \Delta\Phi(\mathrm{D}_i)$

$$\leq (t_i + 1) + (1 - t_i)$$

$$\leq 2$$

Bin Counter w/ Potential Method

- $\widehat{c_i} \leq 2$
- Since:
  - $\Phi(D_0)=0$
  - $\Phi(D_i) \geq 0$, for all i
  - THEN, the total amortized cost (2n) is an upper bound on the total actual cost
  - $T(n) = O(n)$

Dynamic Tables

- "We do not always know in advance how many objects some application will store in a table."
- Dynamically expanding and contracting tables deal with this problem.
- Amortized Analysis allows us to show we can solve this problem in O(1)
  - Providing Dynamic Tables as efficiently as Static Tables.

Dynamic Tables w/ C++

- <vector>'s allow dynamic arrays in c++
- Array.push_back(it)
  - Adds new item
- Array.back()
  - Returns last item.
- Array.pop_back()
  - Removes last item.

```cpp
#include <iostream>     // Enables use of IO
#include <vector>       // Enables use of vector
#include <cstdlib>      // Enables use of rand()
using namespace std;

int main(){
  vector<int> dynamicArray;
  int i, randomSize;

  /* random size of vector */
  randomSize = rand()%11;
  for (i=0;i<randomSize;++i){
    dynamicArray.push_back(i);
  }
  /* print out new vector */
  while (dynamicArray.size()>0) {
    cout << dynamicArray.back() <<" ";
    dynamicArray.pop_back();
  }
  cout <<endl;
}
```

Dynamic Tables

- As it fills, it must reallocate with a larger size
  - copying all objects into the new, larger table
- As it contracts, it might want to reallocate with a smaller size
  - freeing unused space!

Operations w/ Dynamic Table

- Table-Insert: item inserted into table occupying a single **slot**!
  - SLOT: space for one
- Table-Delete: removes an item from the table
  - thereby freeing a slot
- GOAL:
  - Insertion & Deletion in O(1)!
  - Unused space never exceeds a constant fraction of total space!

Load Factor w/ Dynamic Table

- Load Factor $\alpha(T)$

$$\alpha(\text{Empty Table}) = 0$$

$$\alpha(\text{Table}) = \frac{\text{Items in Table}}{\text{Size of Table (number of slots)}}$$

- GOAL: Load Factor Bound Below by Constant
  – Insures unused space is never more than a constant fraction of total space!

First Step w/ Insertion

- Consider only Insertion
  - Table-Insert
- When the table fills up (no more slots) then a larger new table must be allocated.
  - Common to double the current size.
  - Contents from old table slots must be inserted into slots in larger new table!
    - Table must be contiguous (for O(1) access).
  - Guarantees that $\alpha(T) \geq \frac{1}{2}$

Table-Insert(T, x)

```
TABLE-INSERT(T, x)
if T.size == 0
    allocate T.table with 1 slot
    T.size = 1
if T.num == T.size
    allocate new-table with 2 · T.size slots
    insert all items in T.table into new-table
    free T.table
    T.table = new-table
    T.size = 2 · T.size
insert x into T.table
T.num = T.num + 1
```

Table-Insert(T,x) w/ C++

```cpp
DynamicArray::DynamicArray(){
  size = 0;
  num = 0;
};

void TableInsert(DynamicArray& T, int x){
  int* newT;
  cout << "Table Insert In (size,num): "<<T.size<<"," ;
  cout << T.num << endl;
  if (T.size==0){
    cout << "Setting size to 1." <<endl;
    T.table = new int[1];
    T.size=1;
  }
  if (T.num==T.size){
    cout << "Setting size to "<<2*T.size<<".
    newT = new int[2*T.size];
    for (int i=0;i<T.num;++i){
      newT[i]=T.table[i];
    }
    free(T.table);
    T.table=newT;
    T.size=2*T.size;
  }
  T.table[T.num] = x;
  T.num+=1;
  cout << "Table Insert Out (size,num): "<<T.size<<"," ;
  cout << T.num << endl;
}
```

TABLE-INSERT(T, x)

if $T.size == 0$
    allocate $T.table$ with 1 slot
    $T.size = 1$
if $T.num == T.size$
    allocate *new-table* with $2 \cdot T.size$ slots
    insert all items in $T.table$ into *new-table*
    free $T.table$
    $T.table = new\text{-}table$
    $T.size = 2 \cdot T.size$
insert x into $T.table$
$T.num = T.num + 1$

Two "Insertion" Procedures & Expansion

- Elementary Insertion:
  - Insertion into table in lines 6 & 10
  - Constant Time: O(1)
- Table-Insert(T, x) Procedure
  - Elementary insertion w/ partially full array.
  - Expansion occurs w/ Lines 5-9
  - Expansion cost dominated by elementary insertions into new array.

Running Time

- Charge 1 per elementary insertion
- Count only elementary insertions
- c_i = actual cost of the i^{th} operation
  - $c_i = 1$ w/ partially full array
  - If full $c_i=i$:
    - i-1 items in table at start of i^{th} operation.
    - i-1 items inserted into new array
      - Assume allocation time for new array is constant.
    - i^{th} item inserted into slot of new array

Running Time

- Worst case cost per operation then $c_i = O(n)$
- Worst Case Cost w/ n operations = $O(n^2)$
- Still we don't always expand!

Running Time

- Worst case cost per operation then $c_i = O(n)$
- Worst Case Cost w/ n operations = $O(n^2)$
- Still we don't always expand!

$$c_i = \begin{cases} i : if\ i-1\ is\ exact\ power\ of\ 2 \\ 1 : \qquad\qquad otherwise \end{cases}$$

$$Total\ Cost = \sum_{i=1}^{n} c_i$$

Running Time

- Worst case cost per operation then c_i=O(n)
- Worst Case Cost w/ n operations = O(n^2)
- Still we don't always expand!

$$c_i = \begin{cases} i : if\ i - 1\ is\ exact\ power\ of\ 2 \\ 1 : \qquad otherwise \end{cases}$$

$$Total\ Cost = \sum_{i=1}^{n} c_i \le n + \sum_{j=1}^{\lfloor \lg n \rfloor} 2^j$$

Running Time

- Worst case cost per operation then c_i=O(n)
- Worst Case Cost w/ n operations = O(n^2)
- Still we don't always expand!

$$c_i = \begin{cases} i : if\ i - 1\ is\ exact\ power\ of\ 2 \\ 1 : \qquad\qquad\qquad\qquad otherwise \end{cases}$$

$$Total\ Cost = \sum_{i=1}^{n} c_i \le n + \sum_{j=1}^{\lfloor \lg n \rfloor} 2^j = n + \frac{2^{\lfloor \lg n \rfloor} - 1}{2 - 1}$$

Running Time

- Worst case cost per operation then c_i=O(n)
- Worst Case Cost w/ n operations = O(n^2)
- Still we don't always expand!

$$c_i = \begin{cases} i : if\ i-1\ is\ exact\ power\ of\ 2 \\ 1 : \qquad\qquad\qquad\qquad otherwise \end{cases}$$

$$Total\ Cost = \sum_{i=1}^{n} c_i \le n + \sum_{j=1}^{\lfloor \lg n \rfloor} 2^j = n + \frac{2^{\lfloor \lg n \rfloor} - 1}{2-1}$$

$$< n + 2n = 3n$$

Running Time

- Worst case cost per operation then c_i=O(n)
- Worst Case Cost w/ n operations = O(n^2)
- Aggregate Analysis say amortized cost per operation = 3

$$c_i = \begin{cases} i : if\ i-1\ is\ exact\ power\ of\ 2 \\ 1 : \qquad\qquad\qquad\qquad otherwise \end{cases}$$

$$Total\ Cost = \sum_{i=1}^{n} c_i \le n + \sum_{j=1}^{\lfloor \lg n \rfloor} 2^j = n + \frac{2^{\lfloor \lg n \rfloor} - 1}{2-1}$$

$$< n + 2n = 3n$$

Dynamic Array w/ Accounting Method

- Each insertions cost 3 dollars
  - 1 dollar pays for the elementary insertion
  - 1 dollar pays for the 1 later insertion into slot of new larger table.
  - 1 dollar pays for the later insertion of one other existing item into the new larger table.
- Items continually insert into new tables as the array expands.
- But:
  - The table is never less than half full
  - Half the items (the half copied during the last expansion) have no money on them.
  - The other half of the items have 2 dollars each
    - just enough to cover the items with no money!

Dynamic Array w/ Accounting Method (2^{nd})

- Suppose we've just expanded
  - size=m before the expansion
  - size=2m after the expansion
- The expansion will use up all the credit.
  - So after the expansion there is no remaining credit.
- m insertions will occur before the next expansion.
  - Each insertion will put 1 dollar on the item
  - Each insertion will put 1 dollar on one of the m items in slots just after expansion
- 2m credits by the next expansion
  - When there'll be 2m items!

Dynamic Array w/ Potential Method

- Define a potential function that's 0 after an expansion, but builds to table size by the time the table is full
  - enabling expansion w/ potential payment

Dynamic Array w/ Potential Method

$$\Phi(T) = 2 \cdot T.num - T.size$$

- Immediately after expansion :
  - $T.num = T.size/2$

$$\Phi(T) = 2 \cdot \frac{T.size}{2} - T.size = 0$$

Dynamic Array w/Potential Method

$$\Phi(T) = 2 \cdot T.num - T.size$$

- Since the table is always at least half full:

$$\frac{T.num}{T.size} \geq \frac{1}{2}$$

$$2 \cdot T.num \geq \mathrm{T.size}$$

$$\Phi(T) = 2 \cdot T.num - T.size \geq 0$$

- Thus: the sum of the amortized costs of n Table-Insert ops gives an upper bound on the sum of the actual costs!

Amortized Cost i^{th} op w/ no expansion

$$\widehat{c_i} = c_i + \Phi(T_i) - \Phi(Ti_{-1})$$

$= 1 + (2*num_i - size_i) - (2*num_{i-1} - size_{i-1})$

Since no expansion

$= 1 + (2*num_i - size_i) - (2*\mathbf{(num_i - 1)} - \mathbf{size_i})$

$= 1 + 2*num_i - 2*num_i + 2 = 3$

- $\widehat{c_i}$ is O(1)

Amortized Cost i^{th} op w/ Expansion

$$\widehat{c_i} = c_i + \Phi(T_i) - \Phi(Ti_{-1})$$

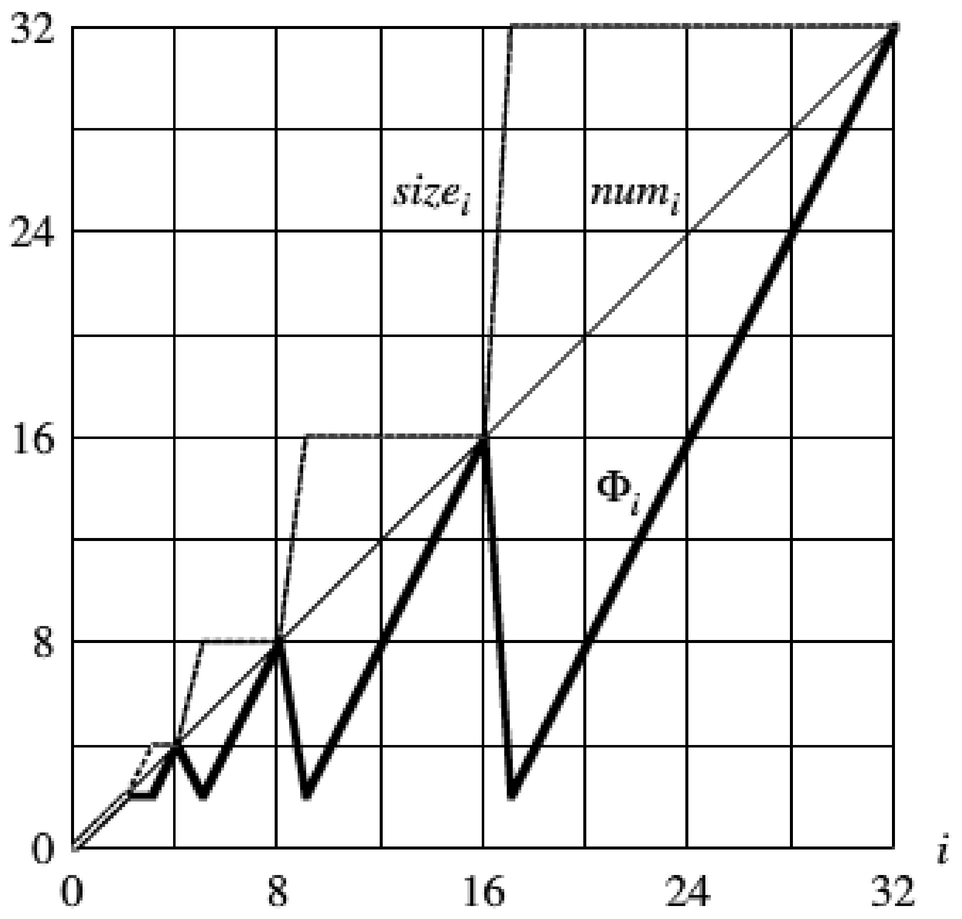
- Since expansion
  - $size_i = 2*size_{i-1}$
  - $size_{i-1} = num_{i-1} = num_i-1$
  - $c_i = num_{i-1} + 1 = num_i$

$= num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1})$

$= num_i + (2num_i - 2(num_i-1)) - (2(num_i-1)-(num_i-1))$

$= num_i + 2 - (num_i - 1) = 3$

- $\widehat{c_i}$ is O(1)



32
24
16
8
0
0
8
16
24
32
i
$size_i$
num_i
Φ_i

Table Expansion and Contraction

- Now need implementation for TableDelete
  - Table delete removes an item from table.
- Removing the item is ez!
- We need to contract table when load factor become too small.
  - When number of items drops too low
  - Allocate new smaller table
  - Copy items from old to new table
  - Free space for old table.

Like Two Properties Preserved

- Load factor of the dynamic table bound below by a positive constant!
- Amortized cost of a table operation is bounded above by a constant!
- Cost measured by elementary insertions & deletions

Expansion & Contraction w/ First Idea

- Expand by doubling when full.
- Contract by halving when half full.
- Guarantees that $\frac{1}{2} \leq \alpha(T) \leq 1$

Expansion & Contraction w/ First Idea – Bad Idea

- T.size = 11
- Insert, Insert, Insert, Insert, Insert, Insert
- Insert, Insert, Insert, Insert, Insert, Insert -> Expand
- T.size = 22 & T.num=12
- Delete, Delete => Contract
- T.num=10 & T.size=11
- Insert, Insert => Expand
- T.size = 22 & T.num = 12
- Delete, Delete => Contract
- T.num=10 & T.size=11
- .
- .
- .

Expansion & Contraction w/ Second Idea

- Expansion was working alright, but we seem to be to anxious to contract!
- Improvement: Let $\alpha(T)$ decrease all the way down to ¼ (not just ½) before contracting by ½.
  - After expansion or contraction the array is half full w/ $\alpha(T)$ = ½ !

Expansion & Contraction w/ Second Idea’s Intuition

- We Want to make sure that we perform enough operations between consecutive expansions/contractions to pay for the change in table size.
- Need to delete have the items before contraction.
- Need to double number of items before expansion.
- Either way, number of operations between expansions/contractions is at least a constant fraction of number of items copied.

Expansion & Contraction Analysis w/ Potential Method

$$\Phi(\mathrm{T}) = \begin{cases} 2 \cdot T.num - T.size, & if\, \alpha(\mathrm{T}) \geq \frac{1}{2} \\ \frac{T.size}{2} - T.num, & if\, \alpha(\mathrm{T}) < \frac{1}{2} \end{cases}$$

- When exactly ½ full, potential is 0.
- As the array shrinks under ½ it prepares for contraction.
- As the array expands beyond ½ it prepares for expansion.

Prove Potential Positive @ any point

$$\Phi(\mathrm{T}) = \begin{cases} 2 \cdot T.num \; - T.size, & if\,\alpha(\mathrm{T}) \;\geq \dfrac{1}{2} \\ \dfrac{T.size}{2} - T.num, & if\,\alpha(\mathrm{T}) \;< \dfrac{1}{2} \end{cases}$$

- T empty $\Phi(\mathrm{T})=0$
- $\alpha(\mathrm{T}) \;\geq \frac{1}{2} \Rightarrow$
  - num $\geq \frac{size}{2} \Rightarrow 2 \cdot num \geq size \Rightarrow \Phi(\mathrm{T}) \geq 0$
- $\alpha(\mathrm{T}) \;< \frac{1}{2} \Rightarrow$
  - num $< \frac{size}{2} \Rightarrow \Phi(\mathrm{T}) \geq 0$

Additional Intuition

- Φ(T) measures how far from α(T) $= \frac{1}{2}$ we are.
- α(T) = $\frac{1}{2}$ ⇒Φ(T) = 2 · $num - 2$ · $num \; = 0$
- α(T) =1⇒Φ(T) = 2 · $num - num \; = num$
  - We have potential to cover expansion
- α(T) = $\frac{1}{4}$ ⇒Φ(T) = size/2 · $num - num$

  $= 4 \cdot num/2 - num \; = num$
  - Leaving potential to pay for contraction

Additional Intuition

- SO: When we double or have, have enough potential to pay for moving all num items in array.
- Potential increases linearly between $\alpha(T) = \frac{1}{2}$ and $\alpha(T) = 1$
- Potential also increases linearly between $\alpha(T) = \frac{1}{2}$ and $\alpha(T) = \frac{1}{4}$

Additional Intuition

- α(T) moves from ½ to 1 and from ½ to ¼.
  - Depending on the direction the distance to travel differs.
- Size=20, Num=10, α(T)=1/2
  - Distance from 10 to 20 is 10
  - Distance from 10 to 5 is 5!
- For α(T) to go from ½ to 1, num increase from size/2 to size.
  - For a total increase of size/2.
  - Explaining the coefficient of 2 on the T.num term in Φ formula when $\alpha(T) \geq \frac{1}{2}$
- For α(T) to go from ½ to 1/4, num decreases from size/2 to size/4 .
  - For a total decrease of size/4.
  - Explaining the coefficient of -1 on the T.num term in Φ formula when $\alpha(T) < \frac{1}{2}$