Amortized Analysis

Andreas Klappenecker

[partially based on the slides of Prof. Welch]

People who analyze algorithms have double happiness. First of all they experience the sheer beauty of elegant mathematical patterns that surround elegant computational procedures. Then they receive a practical payoff when their theories make it possible to get other jobs done more quickly and more economically.

Don Knuth

Analyzing Calls to a Data Structure

- Some algorithms involve repeated calls to one or more data structures.

- When analyzing the running time of an algorithm, one needs to sum up the time spent in all the calls to the data structure.

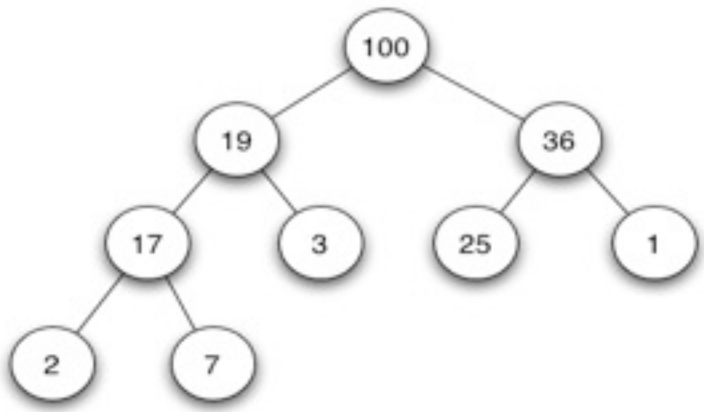
- Problem: If different calls take different times, how can we accurately calculate the total time?

Max-Heap

A max-heap is an nearly complete binary tree

(i.e., all levels except the deepest level are completely filled and the last level is filled from the left)

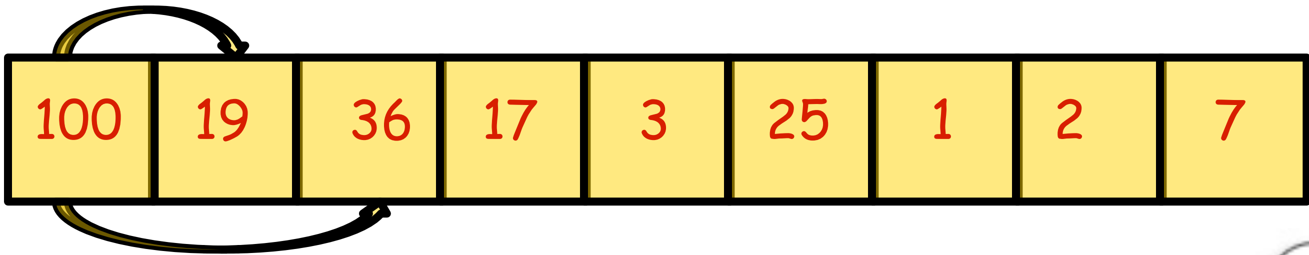
satisfying the heap property: if B is a child of a node A, then key[A] >= key[B].

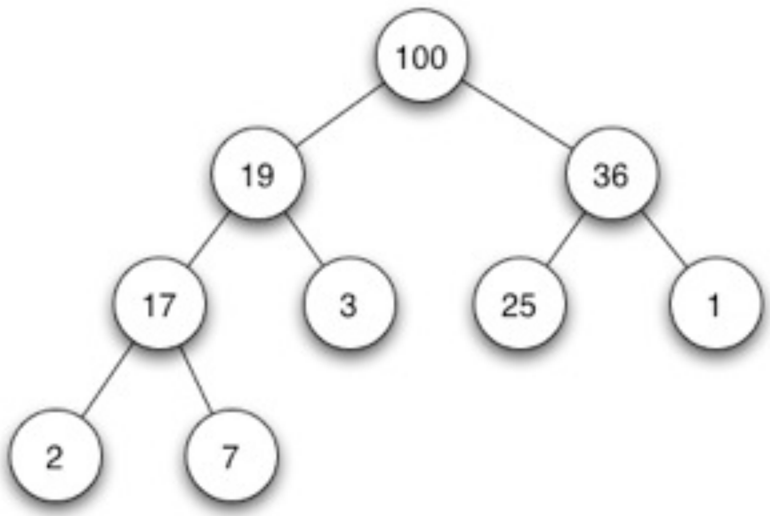


[Picture courtesy of Wikipedia.]

Heap Implementation

We can store a heap in an array:





If the array is indexed a[1..n],

then a[i] has children a[2i] and a[2i+1]:

a[1] has children a[2], a[3],

a[2] has children a[4], a[5],

a[3] has children a[6], a[7], ...

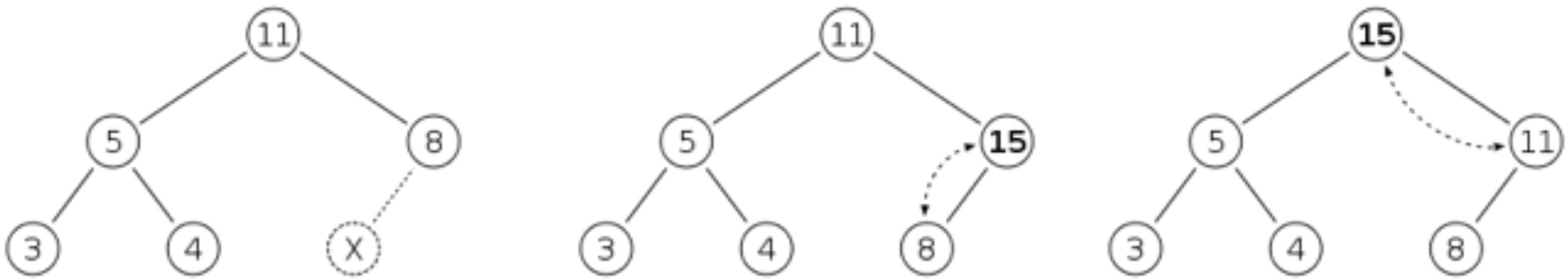
Adding an Element to a Heap

An element can be added to the heap as follows:

1. Add the element on the bottom level of the heap.
2. Compare the added element with its parent; if they are in the correct order, stop.
3. If not, swap the element with its parent and return to the previous step.

Adding an Element: Example

Adding 15 to a max-heap. Insert at position x, compare with parent, swap, compare with parent, swap.



What is the time-complexity of adding an element to a heap with n elements?

Constructing a Heap

Let us form a heap of n elements from scratch.

First idea:

- Use n times add to form the heap.
- Each addition to the heap operates on a heap with at most n elements.
- Adding to a heap with n elements takes O(log n) time
- Total time spent doing the n insertions is O(n log n) time

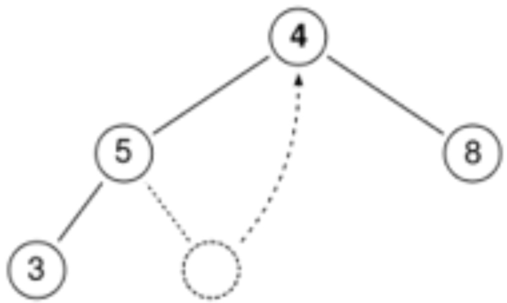
Constructing a Heap (2)

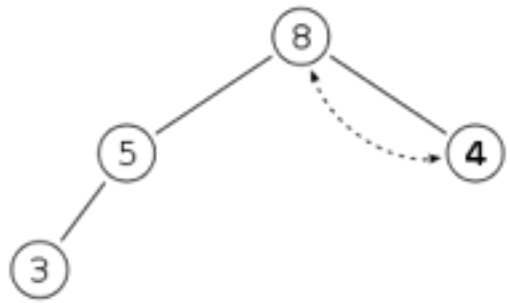
Two question arise:

- Does our analysis overestimate the time?
  The different insertions take different amounts of time, and many are on smaller heaps. (=> leads to amortized analysis)
- Is this the optimal way to create a heap?
  Perhaps simply adding n times is not the best way to form a heap.

Deleting the Maximal Element from a Max-Heap

Deleting the maximal element from a max-heap starts by replacing it with the last element from the lowest level. Then restore the heap property (using Max-Heapify) by swapping with largest child, and repeat same process on the next level, etc.





Constructing a Heap (3)

Second idea:

Place elements in an array, interpret as a binary tree. Look at subtrees at height h (measured from lowest level). If these trees have been heapified, then subtrees at height h+1 can be heapified by sending their roots down.

Initially, the trees at height 0 are all heapified.

Constructing a Heap (4)

Array of length n. Number of nodes at height h is at most floor($n/2^{h+1}$). Cost to heapify a tree at height h+1 if all subtrees have been heapified: O(h) swaps. Total cost:

$$\begin{aligned}\sum_{h=0}^{\lceil \lg n \rceil} \frac{n}{2^{h+1}} O(h) &= O\left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^h}\right) \\ &\le O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O(n)\end{aligned}$$

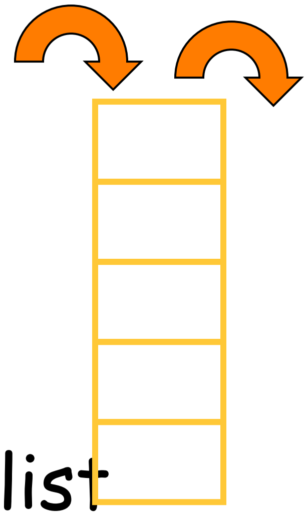
Amortized Analysis

Amortized Analysis

- Purpose is to accurately compute the total time spent in executing a sequence of operations on a data structure
- Three different approaches:
  - aggregate method: brute force
  - accounting method: assign costs to each operation so that it is easy to sum them up while still ensuring that the result is accurate
  - potential method: a more sophisticated version of the accounting method (omitted here)

Running Example #1:

- Operations are:
  - Push(S,x)
  - Pop(S)
  - Multipop(S,k) - pop the top k elements
- Implement with either array or linked list
  - time for Push is O(1)
  - time for Pop is O(1)
  - time for Multipop is O(min(|S|,k))



Running Example #2:

- Operation:
  - increment(A) - add 1 (initially 0)
- Implementation:
  - k-element binary array
  - use grade school ripple-carry algorithm

Aggregate Method

Aggregate Method

- Show that a sequence of n operations takes T(n) time
- We can then say that the amortized cost per operation is T(n)/n
- Makes no distinction between operation types

Augmented Stack:

- In a sequence of n operations, the stack never holds more than n elements.
- Thus, the cost of a multipop is $O(n)$
- Therefore, the worst-case cost of any sequence of n operations is $O(n^2)$.
- But this is an over-estimate!

Aggregate Method for

- Key idea: total number of pops (or multipops) in the entire sequence of operations is at most the total number of pushes

- Suppose that the maximum number of Push operations in the sequence is n.
- So time for entire sequence is O(n).
- Amortized cost per operation: O(n)/n = O(1).

Aggregate Method for k-Bit

- Worst-case time for an increment is O(k). This occurs when all k bits are flipped
- But in a sequence of n operations, not all of them will cause all k bits to flip:
  - bit 0 flips with every increment
  - bit 1 flips with every 2nd increment
  - bit 2 flips with every 4th increment …
  - bit k flips with every 2^k-th increment

Aggregate Method for k-Bit

- Total number of bit flips in n increment operations is
  - $n + n/2 + n/4 + \ldots + n/2^k < n(1/(1-1/2)) = 2n$
- So total cost of the sequence is $O(n)$.
- Amortized cost per operation is $O(n)/n = O(1)$.

Accounting Method

Accounting Method

- Assign a cost, called the "amortized cost", to each operation
- Assignment must ensure that the sum of all the amortized costs in a sequence is at least the sum of all the actual costs
  - remember, we want an upper bound on the total cost of the sequence

Accounting Method

- For each operation in the sequence:
  - if amortized cost > actual cost then store extra as a credit with an object in the data structure
  - if amortized cost < actual cost then use the stored credits to make up the difference
- Never allowed to go into the red! Must have enough credit saved up to pay for

Accounting Method vs.

- Aggregate method:
  - first analyze entire sequence
  - then calculate amortized cost per operation
- Accounting method:
  - first assign amortized cost per operation
  - check that they are valid (never go into the red)

Accounting Method for

- Assign these amortized costs:
  - Push - 2
  - Pop - 0
  - Multipop - 0
- For Push, actual cost is 1. Store the extra 1 as a credit, associated with the pushed element.
- Pay for each popped element (either from

Accounting Method for

- There is always enough credit to pay for each operation (never go into red).
- Each amortized cost is $O(1)$
- So cost of entire sequence of n operations is $O(n)$.

Accounting Method for k-Bit

- Assign amortized cost for increment operation to be 2.
- Actual cost is the number of bits flipped:
  - a series of 1's are reset to 0
  - then a 0 is set to 1
- Idea: 1 is used to pay for flipping a 0 to 1. The extra 1 is stored with the

Accounting Method for k-Bit Counter

Accounting Method for k-Bit Counter

0	0	0	0	0

Accounting Method for k-Bit Counter

0	0	0	0	0

1

0	0	0	0	1

Accounting Method for k-Bit Counter

0	0	0	0	0

1

0	0	0	0	1

1

0	0	0	1	0

Accounting Method for k-Bit Counter

0	0	0	0	0

				1
0	0	0	0	1

			1	
0	0	0	1	0

			1	1
0	0	0	1	1

Accounting Method for k-Bit Counter

0	0	0	0	0

				1
0	0	0	0	1

			1	
0	0	0	1	0

			1	1
0	0	0	1	1

		1		
0	0	1	0	0

Accounting Method for k-Bit Counter

0	0	0	0	0

				1
0	0	0	0	1

			1	
0	0	0	1	0

			1	1
0	0	0	1	1

		1		
0	0	1	0	0

		1		1
0	0	1	0	1

Accounting Method for k-Bit Counter

0	0	0	0	0

				1
0	0	0	0	1

			1	
0	0	0	1	0

			1	1
0	0	0	1	1

		1		
0	0	1	0	0

		1		1
0	0	1	0	1

		1	1	
0	0	1	1	0

Accounting Method for k-Bit Counter

0	0	0	0	0
				1
0	0	0	0	1
			1	
0	0	0	1	0
			1	1
0	0	0	1	1

		1		
0	0	1	0	0
		1		1
0	0	1	0	1
		1	1	
0	0	1	1	0
		1	1	1
0	0	1	1	1

Accounting Method for k-Bit

- All changes from 1 to 0 are paid for with previously stored credit (never go into red)
- Amortized time per operation is $O(1)$
- total cost of sequence is $O(n)$

Conclusions

Amortized Analysis allows one to estimate the cost of a sequence of operations on data structures.

The method is typically more accurate than worst case analysis when the data structure is dynamically changing.

Dynamic Table

Tables

Goal: Create a table that is small, but large enough so that it will not overflow.

Difficulty: Proper size might not be known in advance.

Solution: Dynamic tables

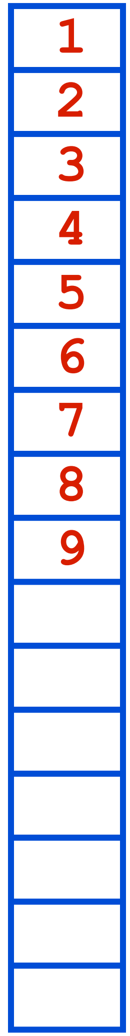
Basic idea: whenever the table overflows, simply grow it to a larger table (that is, for instance, twice the size).

Table Insert

```
Insert(T,x)
  if (size[T]==0) then      // allocate initial table
     allocate table[T] with 1 slot; size[T]=1; num[T]=0;
  endif;
  if (num[T] == size[T]) then      // overflow, grow table
     allocate new_table with 2*size[T] slots;
     insert all items in table[T] into new_table; free table[T];
     table[T] := new_table; size[T] := 2*size[T];
  endif;
  insert x into table[T]; num[T] := num[T] + 1;
```

Example

Operation	Table Size	Cost
Insert(1)	1	1
Insert(2)	2	1 + 1
Insert(3)	4	1 + 2
Insert(4)	4	1
Insert(5)	8	1 + 4
Insert(6)	8	1
Insert(7)	8	1
Insert(8)	8	1
Insert(9)	16	1 + 8



Challenge

Use aggregate analysis to find the amortized cost of a dynamic table, assuming that n elements are inserted one after another.

Aggregate Analysis

Cost c_i for the i-th insert is

$1 + 2^k$ if $i-1 = 2^k$ for some $k >= 0$

1 otherwise

Thus, n insert operations cost:

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

Amortized cost: <= 3n/n = 3

Challenge

Use the accounting method to find the amortized cost of a dynamic table, assuming that n elements are inserted one after another.

Accounting Analysis

- Charge \$3 for each operation
- The operation costs \$1
- We have \$2 left over to pay for future doubling of the table
- Why \$2? Assume after growing the table, all previous elements have already paid their dues and no money is left over. Each element in the new half of the array has to pay for its future move and for the move of one element of the old half.