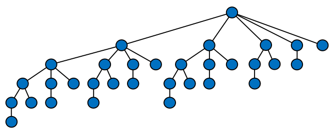
Lecture 2. Amortized Analysis

CpSc 8400: Algorithms and Data Structures

Brian C. Dean

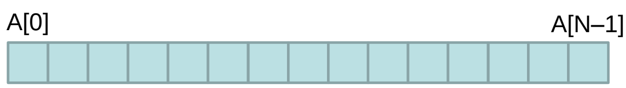


School of Computing
Clemson University
Spring, 2021

1

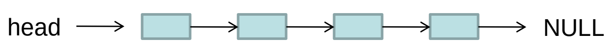
Warm-Up: Fundamental Data Structures

Arrays



- Retrieve or modify any element in O(1) time.
- Insert or delete in middle of list: O(N) time. ☹
- Insert or delete from ends: O(1) time
  - Be careful not to run over end of allocated memory

Linked Lists



(sometimes doubly linked, or ending with a sentinel instead of NULL)

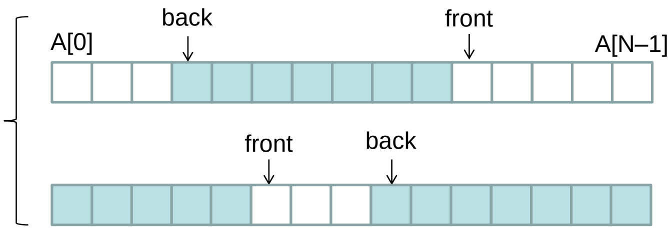
- Seek to any position in list: O(N) time. ☹
- Then insert or delete element: O(1) time.
- Insert or delete from ends: O(1) time.

2

2

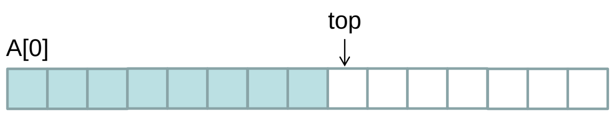
Fundamental Data Structures

Queues



- First-In, First-Out (FIFO).
- O(1) enqueue & dequeue, implemented using arrays or linked lists (often implemented using <u>circular</u> arrays).

Stacks



- Last-In, First-Out (LIFO).
- O(1) push & pop, implemented using arrays or linked lists.

3

3

An Important Distinction…

<u>Specification</u> of a data structure in terms of the operations it needs to support.

(sometimes called an *abstract data type)*

A concrete approach for **<u>implementation</u>** of the data structure that fulfills these requirements.

4

4

Example: Queues

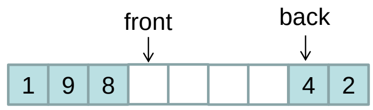
Abstract data type:
queue

Must support these operations:

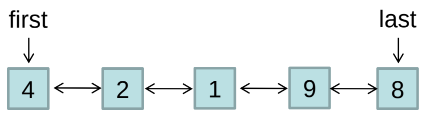
- *Insert*(k) a new key *k* into the structure.
- *Remove* the least-recently-inserted key from the structure.

(so FIFO behavior)

Choices for concrete implementation:



Circular array



(Doubly) linked list

Enforcing Abstraction in Code

Abstract data type:
queue

Concrete implementation:
queue.cpp

```
queue.h:

class Queue {
  private:
    int *A;
    int front, back, N;

  public:
    Queue();
    ~Queue();
    void insert(int key);
    int remove(void);
};
```

Enforcing Abstraction in Code

Abstract data type:
queue

Concrete implementation:
queue.cpp

```
queue.h:

class Queue {
private:
    int *A;
    int front, back, N;
public:
    Queue();
    ~Queue();
    void insert(i
    int remove(vo
};
```

```
Queue q;
q.insert(6);
x = q.remove();
```

```
int Queue::remove(void)
{
    int result = A[back];
    back = (back+1) % N;
    return result;
}
```

7

Today's Useful Analysis Technique: Adding Things in Smart Ways

- Lots of analyses get easier when you add things together after re-grouping them in smart ways.

8

Example: Think about an Algorithm from the Perspective of a Data Element…

Elements of Data

Steps Taken by Algorithm (in chronological order…)

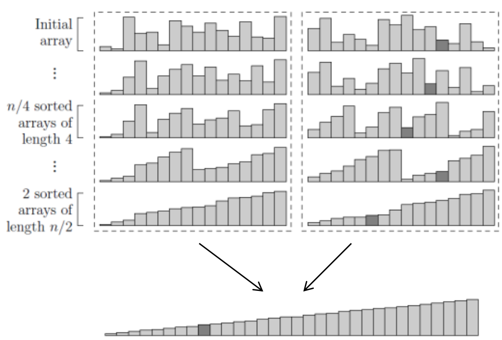
	Elem 1	Elem 2	Elem 3	Elem 4	Elem 5	
Step 1		1	1			*"Swap elements 2 and 3": O(1)*
Step 2						
Step 3				1		*"Modify element 4": O(1)*
Step 4						
Step 5	1	1	1	1	1	*"Copy elements into a new array": O(n)*

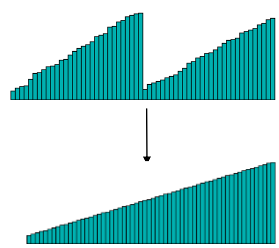
Time/work done to each element at each major "step" of our algorithm's execution

(that is, sum the columns first, not the rows…)

Example: Merge Sort

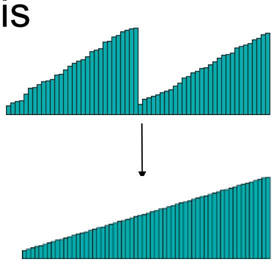
- It takes Θ(n) time to merge two sorted lists of combined length n.
- How much time does it take to do a full merge sort then?





Example: Merge Sort

- It takes Θ(n) time to merge two sorted lists of combined length n.
- Think of this as O(1) time per element taking part in the merge.
- Now look at a particular array element e. The total "work" we spend on e is equal to the number of merges in which e takes part: O(log n).
  - Why O(log n)? Each merge doubles the size of the sorted subarray containing e.
- So O(n log n) total work.



11

11

Useful Analysis Technique: Think about an Algorithm from the Perspective of a Data Element…

- Figure out how much work / running time is spent on a single generic element of data during the course of the algorithm.
- Add this up to get the total running time.

(compared to adding up the time spent on each "operation", summed over each operation in chronological order)

12

12

Example: Enumerating Subsets

counter = 0

For all subsets $S \subseteq \{1, 2, 3, \ldots, n\}$

  Increment counter

What is the value of counter at the end of execution?

13

13

Example: Enumerating Subsets

counter = 0

For all subsets $S \subseteq \{1, 2, 3, \ldots, n\}$

  For all subsets $T \subseteq S$

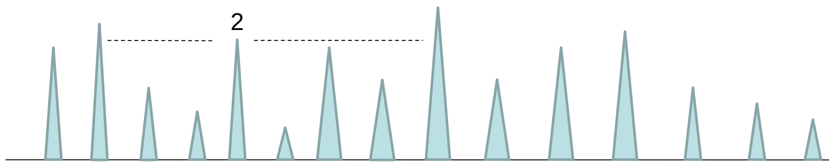
    Increment counter

What is the value of counter at the end of execution?

14

14

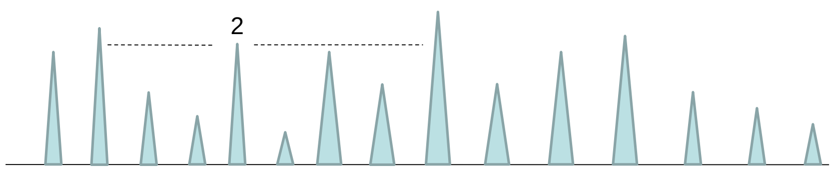
Example: Domination Radius



- Given the heights of N individuals standing in a line.
- **Goal:** find the domination radius of each individual.

15

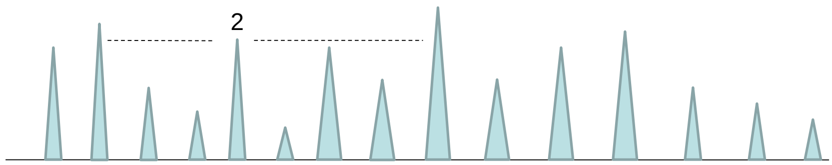
Example: Domination Radius



- Given the heights of N individuals standing in a line.
- **Goal:** find the domination radius of each individual.
- Simple algorithm: from each element, scan left until blocked, then scan right until blocked.

16

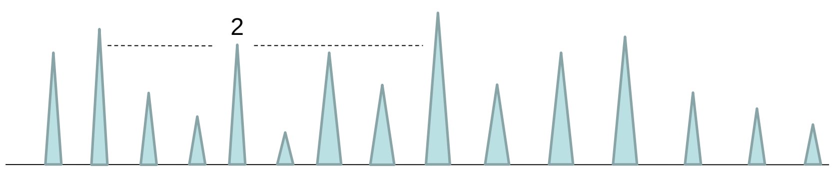
Example: Domination Radius



- Given the heights of N individuals standing in a line.
- **Goal:** find the domination radius of each individual.
- Simple algorithm: from each element, scan left until blocked, then scan right until blocked. ($O(N^2)$ worst-case)

17

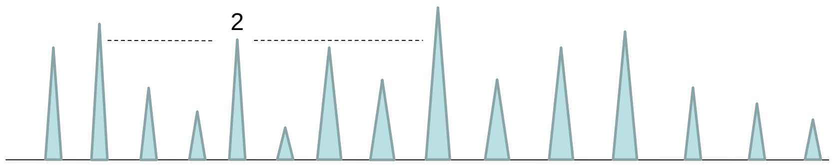
Example: Domination Radius



- Given the heights of N individuals standing in a line.
- **Goal:** find the domination radius of each individual.
- Simple algorithm: from each element, scan left until blocked, then scan right until blocked. ($O(N^2)$ worst-case)
- Refinement: from each element, scan left and right simultaneously until blocked.

18

Example: Domination Radius



- Given the heights of N individuals standing in a line.
- **Goal:** find the domination radius of each individual.
- Simple algorithm: from each element, scan left until blocked, then scan right until blocked. ($O(N^2)$ worst-case)
- Refinement: from each element, scan left and right simultaneously until blocked.
  - Add up running time by grouping together “high work” elements separately from “low work” elements…

19

Recall: Think about an Algorithm from the Perspective of a Data Element…

Elements of Data

Steps Taken by Algorithm (in chronological order…)

	Elem 1	Elem 2	Elem 3	Elem 4	Elem 5	
Step 1		1	1			*“Swap elements 2 and 3”*
Step 2						
Step 3				1		*“Modify element 4”*
Step 4						
Step 5	1	1	1	1	1	*“Copy elements into a new array”*

Time/work done to each element at each major “step” of our algorithm’s execution

20

20

Re-Sizing Memory Blocks

- Since memory blocks often cannot expand after allocation, what do we do when a memory block fills up?
- For example, suppose we allocate 100 words of memory space for a stack (implemented as an array), but then realize we have more than 100 elements to push onto the stack!

(yes, use of a linked list would have solved this problem, but suppose we really want to use arrays instead…)

21

21

Memory Allocation : Successive Doubling

- A common technique for block expansion: whenever our current block fills up, allocate a new block of twice its size and transfer the contents to the new block.
- Unfortunately, now some of our push operations will be quite slow!
  - Most push operations take only O(1) time.
  - However, a push operation resulting in an expansion (and a copy of the n elements currently in the stack) will take Θ(n) time.

22

22

How to Describe the Running Time of Push…?

- Push has a somewhat non-uniform running time profile:
  - O(1) almost always
  - Except Θ(N) every now and then.
- But just saying the running time is "Θ(N) in the worst case" doesn't tell the whole story…
  - Doesn't do the structure justice.
  - People might be scared to use it for large input sizes…

23

How Expensive is Your Car to Maintain…?

Jan: $10

Feb: $10

…

Nov: $10

Dec: $130 = $10 + yearly $120 tune-up

- Same problem: saying it's "$130/month in the worst case" doesn't tell the complete story…

24

Amortization of Costs

Actual Cost	Amortized Cost
Jan: $10	Jan: $20
Feb: $10	Feb: $20
…	…
Nov: $10	Nov: $20
Dec: $130	Dec: $20

- So our cost is "$20/month, amortized".
- This is a simpler, more accurate description of our cost structure.
- Compare with actually paying $20/month…

25

Back to Push…

- How much does each push actually cost?

Op#:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	…
Insert:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…
Copy:		1	2		4				8								16		…
Total:	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17	1	…
Cumulative:	1	3	6	7	12	13	14	15	24	25	26	27	28	29	30	31	48	49	…

26

Back to Push…

- How much does each push actually cost?

Op#:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	…
Insert:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…
Copy:		1	2		4				8								16		…
Total:	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17	1	…
Cumulative:	1	3	6	7	12	13	14	15	24	25	26	27	28	29	30	31	48	49	…

- What about if we charge ourselves 3 units of work per operation instead…?

Total:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	…
Cumulative:	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	…

27

27

Back to Push…

- How much does each push actually cost?

k

Op#:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	…
Insert:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…
Copy:		1	2		4				8								16		…
Total:	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17	1	…
Cumulative:	1	3	6	7	12	13	14	15	24	25	26	27	28	29	30	31	48	49	…

- What about if we charge ourselves 3 units of work per operation instead…?

Total:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	…
Cumulative:	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	…

"True" cumulative cost after any sequence of k operations is upper bounded by "fictitious" cumulative cost of 3k…

28

28

Back to Push…

- How much does each push actually cost?

k

Op#:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	…
Insert:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…
Copy:		1	2		4				8								16		…
Total:	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17	1	…
Cumulative:	1	3	6	7	12	13	14	15	24	25	26	27	28	29	30	31	48	49	…

- What about if we charge ourselves 3 units of work per operation instead…?

Total:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	…
Cumulative:	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	…

Total insert cost = k

Total copy cost ≤ 2k

29

29

Back to Push…

- How much does each push actually cost?

k

Op#:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	…
Insert:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…
Copy:		1	2		4				8 +2	+2	+2	+2	+2	+2	+2	+2	~~16~~		…
Total:	1	2	3	1	5	1	1	1		1	1	1	1	1	1	1	17	1	…
Cumulative:	1	3	6	7	12	13	14	15	24	25	26	27	28	29	30	31	48	49	…

- What about if we charge ourselves 3 units of work per operation instead…?

Total:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	…
Cumulative:	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	…

Total insert cost = k

Total copy cost ≤ 2k

30

30

Back to Push…

- How much does each push actually cost?

k

Op#:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	…
Insert:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…
Copy:		1	2		4	+2	+2	+2	~~8~~	+2	+2	+2	+2	+2	+2	+2	~~16~~		…
Total:	1	2	3	1	+2	1	1	1	+2	1	1	1	1	1	1	1	17	1	…
Cumulative:	1	3	6	7	12	13	14	15	24	25	26	27	28	29	30	31	48	49	…

- What about if we charge ourselves 3 units of work per operation instead…?

Total:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	…
Cumulative:	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	…

Total insert cost = k

Total copy cost ≤ 2k

31

31

Back to Push…

- How much does each push actually cost?

Op#:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	…
Insert:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…
Copy:		1	2		4				8								16		…
Total:	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17	1	…
Cumulative:	1	3	6	7	12	13	14	15	24	25	26	27	28	29	30	31	48	49	…

- What about if we charge ourselves 3 units of work per operation instead…?

Total:	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	…
Cumulative:	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	…

- So how different is our version of push from a version that takes 3 units in the worst case?

32

32

Amortized Analysis

- Any sequence of k pushes takes O(k) worst-case time, so we say that push takes O(1) **amortized** time.
- "On average", over the entire sequence, each individual push therefore takes O(1) time.
- In general, an operation runs in O(f(n)) amortized time if any sequence of k such operations runs in O(k f(n)) time.

33

Amortized Analysis : Motivation

- Amortized analysis is an ideal way to characterize the worst-case running time of operations with highly non-uniform performance.
- **It is still worst-case analysis, just averaged over an arbitrary sequence of operations.**
- It gives us a much clearer picture of the true performance of a data structure that more faithfully describes the true performance.
  - E.g., "Θ(N) worst case vs. O(1) amortized".

34

Amortized Analysis : Motivation

- Suppose we have 2 implementations of a data structure to choose from:
  A: O(log n) worst-case time / operation.
  B: O(log n) amortized time / operation.
- There is **no difference** if we use either A or B as part of a larger algorithm. For example, if our algorithm makes n calls to the data structure, the running time is O(n log n) in either case.
- The choice between A and B only matters in a "real-time" setting when the response time of an individual operation is important.