Exam 2 review

William Hendrix

Lecture 13

Exam topics

- Stacks and queues
- Dictionaries
  - Sorted/unsorted arrays and lists
  - Binary search trees (including balanced BSTs)
  - Hash tables
    - Expected case analysis
- Priority Queues/Heaps
- Union-Find
- Divide-and-conquer algorithms
- Master Theorem

Question types

- List operations and/or complexity for data structure
- Compare and contrast data structures
  - Complexity, space, cache coherency
- Definitions of expected case or amortized complexity
  - Will not need to derive amortized complexity
- Describe data structure after some operations
- Describe algorithm output based on data structure
- Algorithm design
- Divide-and-conquer algorithms
- Master Theorem

Exam topics

- Stacks and queues
- Dictionaries
  - Sorted/unsorted arrays and lists
  - Binary search trees (including balanced BSTs)
  - Hash tables
    - Expected case analysis
- Priority Queues/Heaps
- Union-Find
- Divide-and-conquer algorithms
- Master Theorem

Back

Master Theorem

- Powerful theorem for proving complexity of divide-and-conquer algorithms

Master Theorem. If $T(n) = aT(n/b) + f(n)$,

$$T(n) = \begin{cases} \Theta(n^c), & \text{if } f(n) = O(n^{c-\epsilon}) \text{ for some } \epsilon > 0 \\ \Theta(n^c \lg n), & \text{if } f(n) = \Theta(n^c) \\ \Theta(f(n)), & \text{if } f(n) = \Omega(n^{c+\epsilon}) \text{ for some } \epsilon > 0 \\ & \underline{\text{and}}\ af(n/b) < f(n) \text{ for large } n \end{cases},$$

<u>Four steps to solve:</u>

1. Identify a, b, and $f(n)$
2. Calculate $c = \log_b(a)$
3. Decide case: $f(n)$ vs. n^c: $O(n^{c-\varepsilon})$, $\Theta(n^c)$, $\Omega(n^{c+\varepsilon})$
4. Apply Master Theorem (test regularity if case 3)

Back

Master Theorem exercises

- What is *c* for the following recurrences?
- What case does *f(n)* fall under?
- What is the asymptotic complexity for the following recurrences? Write "n/a" if the Master Theorem does not apply.

1. $T(n) = 2T(n/2) + \Theta(n^2)$
2. $U(n) = 4U(n/2) + \Theta(n^2)$
3. $V(n) = 9V(n/9) + \Theta(n)$
4. $W(n) = 3W(n/3) + \Theta(n^2)$
5. $X(n) = 2X(n/4) + \Theta(n)$
6. $Y(n) = 3Y(n/9) + \Theta(1)$
7. $Z(n) = Z(n/2) + \Theta(1)$

	c	Case	Complexity
$T(n)$			
$U(n)$			
$V(n)$			
$W(n)$			
$X(n)$			
$Y(n)$			
$Z(n)$			

Back

Master Theorem exercises

- What is *c* for the following recurrences?
- What case does *f(n)* fall under?
- What is the asymptotic complexity for the following recurrences? Write "n/a" if the Master Theorem does not apply.

1. $T(n) = 2T(n/2) + \Theta(n^2)$
2. $U(n) = 4U(n/2) + \Theta(n^2)$
3. $V(n) = 9V(n/9) + \Theta(n)$
4. $W(n) = 3W(n/3) + \Theta(n^2)$
5. $X(n) = 2X(n/4) + \Theta(n)$
6. $Y(n) = 3Y(n/9) + \Theta(1)$
7. $Z(n) = Z(n/2) + \Theta(1)$

	c	Case	Complexity
$T(n)$	1		
$U(n)$	2		
$V(n)$	1		
$W(n)$	1		
$X(n)$	0.5		
$Y(n)$	0.5		
$Z(n)$	0		

Back

Master Theorem exercises

- What is *c* for the following recurrences?
- What case does *f(n)* fall under?
- What is the asymptotic complexity for the following recurrences? Write "n/a" if the Master Theorem does not apply.

1. $T(n) = 2T(n/2) + \Theta(n^2)$
2. $U(n) = 4U(n/2) + \Theta(n^2)$
3. $V(n) = 9V(n/9) + \Theta(n)$
4. $W(n) = 3W(n/3) + \Theta(n^2)$
5. $X(n) = 2X(n/4) + \Theta(n)$
6. $Y(n) = 3Y(n/9) + \Theta(1)$
7. $Z(n) = Z(n/2) + \Theta(1)$

	c	Case	Complexity
$T(n)$	1	$\Omega(n^{c+\epsilon})$	
$U(n)$	2	$\Theta(n^c)$	
$V(n)$	1	$\Theta(n^c)$	
$W(n)$	1	$\Omega(n^{c-\epsilon})$	
$X(n)$	0.5	$\Omega(n^{c+\epsilon})$	
$Y(n)$	0.5	$O(n^{c-\epsilon})$	
$Z(n)$	0	$\Theta(n^c)$	

Back

Master Theorem exercise solutions

- What is *c* for the following recurrences?
- What case does *f(n)* fall under?
- What is the asymptotic complexity for the following recurrences? Write "n/a" if the Master Theorem does not apply.

1. $T(n) = 2T(n/2) + \Theta(n^2)$
2. $U(n) = 4U(n/2) + \Theta(n^2)$
3. $V(n) = 9V(n/9) + \Theta(n)$
4. $W(n) = 3W(n/3) + \Theta(n^2)$
5. $X(n) = 2X(n/4) + \Theta(n)$
6. $Y(n) = 3Y(n/9) + \Theta(1)$
7. $Z(n) = Z(n/2) + \Theta(1)$

	c	Case	Complexity
$T(n)$	1	$\Omega(n^{c+\epsilon})$	$\Theta(n^2)$
$U(n)$	2	$\Theta(n^c)$	$\Theta(n^2 \lg n)$
$V(n)$	1	$\Theta(n^c)$	$\Theta(n \lg n)$
$W(n)$	1	$O(n^{c-\epsilon})$	$\Theta(n^2)$
$X(n)$	0.5	$\Omega(n^{c+\epsilon})$	$\Theta(n)$
$Y(n)$	0.5	$O(n^{c-\epsilon})$	$\Theta(\sqrt{n})$
$Z(n)$	0	$\Theta(n^c)$	$\Theta(\lg n)$

Back

Stacks and queues

- **Stacks**
  - Support *push()* and *pop()* operations
  - Last-In, First-Out (LIFO) order
- **Queues**
  - Support *enqueue()* and *dequeue()* operations
  - First-In, First-Out (FIFO) order

- **Deques** ("decks")
  - Support all 4 operations
- All three implemented using dynamic arrays
- All operations *O(1)*
  - *enqueue()* and *push() O(1) amortized* time

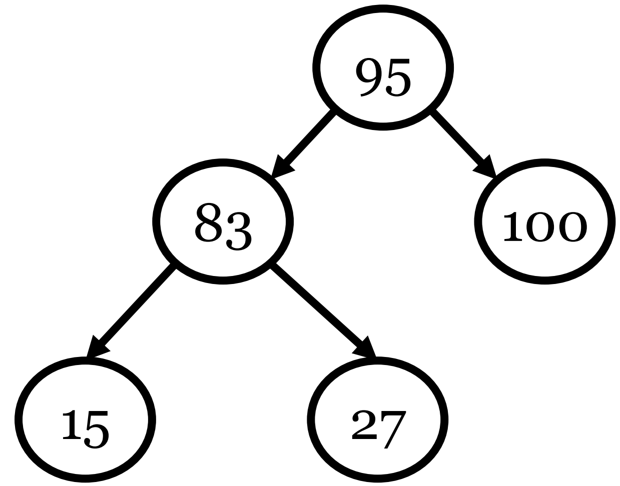
Back

Stack/queue exercise

- Consider the following algorithm for iterating through the elements of a Binary Search Tree:

```
  Input: tree: a BST
1 Algorithm: Iterate
2 nodes = {};
3 Add tree.root to nodes;
4 while nodes ≠ ∅ do
5 |   Print all the elements of nodes;
6 |   t = next element of nodes;
7 |   Add t.left to nodes, unless it's NIL;
8 |   Add t.right to nodes, unless it's NIL;
9 end
```

T

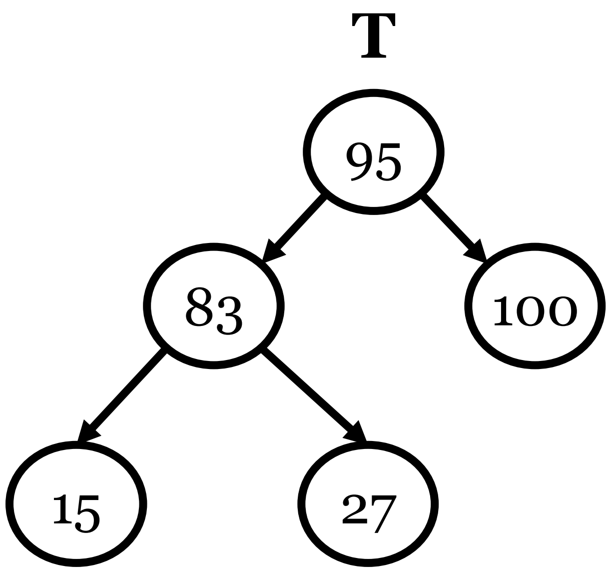


- Assume that Line 5 prints the node values in the order they would be removed
  - I.e., first value is next node to be popped/dequeued

1. What is printed by Iterate(T) if nodes is a stack?
2. What is printed by Iterate(T) if nodes is a queue?

Stack/queue exercise solution

Back



Iteration	Stack	Queue
1	95	95
2	100, 83	83, 100
3	83	100, 15, 25
4	27, 15	15, 25
5	15	25

Back

Dictionaries

- 3 main operations: Insert(x), Delete(x), Search(x)
- 4/5 secondary operations: Max(), Min(), Successor(x), Predecessor(x), Build
- Seven main implementations with various pros/cons
  - Unsorted array
  - Sorted array
  - Unsorted doubly-linked list
  - Sorted doubly-linked list
  - Balanced binary search tree
  - Hash table (expected case)
  - Bit vector
  - Time complexity, time coefficient (e.g., caching), space (e.g., links vs. no links, empty cells)
- No singly-linked lists or unbalanced BSTs
- Hash tables: separate chaining vs. open addressing

Dictionary complexity

Operation	Unsorted array	Unsorted DLL	Sorted array	Sorted DLL	BBST	Hash table	Bit vector
Search(x)	$O(n)$	$O(n)$	$O(\lg n)$	$O(n)$	$O(\lg n)$	$O(1)$†	$O(1)$
Delete(x)	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(\lg n)$	$O(1)$†	$O(1)$
Insert(x)	$O(1)$*	$O(1)$	$O(n)$	$O(n)$	$O(\lg n)$	$O(1)$†	$O(1)$
Build	n/a	n/a	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n)$†	$O(n+r)$
Min()	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(\lg n)$	$O(n)$†	$O(r)$
Max()	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(\lg n)$	$O(n)$†	$O(r)$
Pred(x)	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(\lg n)$	$O(n)$†	$O(r)$
Succ(x)	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(\lg n)$	$O(n)$†	$O(r)$

* Amortized time
† Expected case

Back

Dictionary exercise

1. Create a table with the worst-case complexity of the algorithm below using 7 different dictionary implementations:
   - Sorted and unsorted array, sorted and unsorted doubly-linked list, balanced BST, hash table (expected), bit vector

```
Input: data: array of positive integers
Input: n: number of integers in data
Output: set of unique elements in data
Algorithm: Unique
dict = Dictionary();
for i = 1 to n do
    if dict.Search(data[i]) = NIL then
        dict.Insert(data[i]);
    end
end
return dict;
```

2. How long would it take to print out all of the elements in these dictionaries?

Dictionary exercise solution

Back

1. Unique will perform:
   - n calls to Search()
   - Up to n calls to Insert()
   - $O(n)$ other operations

Implementation	n Searches	$\leq n$ Inserts	Total time
Unsorted array	$O(n^2)$	$O(n)$	$O(n^2)$
Sorted array	$O(n\lg n)$	$O(n^2)$	$O(n^2)$
Unsorted DLL	$O(n^2)$	$O(n)$	$O(n^2)$
Sorted DLL	$O(n^2)$	$O(n^2)$	$O(n^2)$
Balanced BST	$O(n\lg n)$	$O(n\lg n)$	$O(n\lg n)$
Hash table	$O(n)^\dagger$	$O(n)^\dagger$	$O(n)^\dagger$
Bit vector	$O(n)$	$O(n)$	$O(n)$

2. Arrays or DLLs: $O(n)$

- BSTs: $O(n)$
- Hash table: $O(m)$, or $O(n)$ expected
- Bit vector: $O(r)$, where r is data range

Back

Priority queues and heaps

- **Priority Queue**
  - Abstract data structure that supports extracting max/min element
  - Main operations (max): Max(), DeleteMax(), Insert(x)
- **Heap:** primary implementation for Priority Queue
  - Array-based complete BST
  - *Heap property*: all children are smaller (larger) than their parent
  - Parent of i is at $i/2$, children are at $2i$ and $2i+1$
  - Helper operations: PercolateUp(i), PercolateDown(i)
    - Shift a value up or down in the tree to satisfy heap property
    - Both: $O(\lg n)$

Operation	Heap
Insert(x)	$O(\lg n)$
Max()	$O(1)$
DeleteMax()	$O(\lg n)$
Build	$O(n)$

- No Fibonacci heaps on exam!

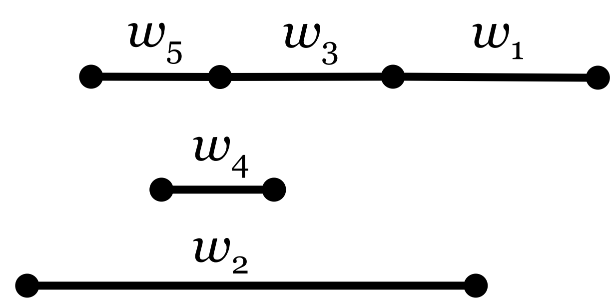
Back

Heap exercise

- Consider the following greedy algorithm for optimizing workshop attendance at a conference:

```
   Input: ws: set of workshops, with start and end times
   Input: n: number of workshops in W
   Output: W: largest set of workshops that do not overlap
1  Algorithm: GreedyWorkshops
2  W = Queue();
3  heap = MinHeap(n);
   // heap compares workshops according to end time
4  for i = 1 to n do
5  |   heap.Insert(ws[i]);
6  end
7  last = 0;
8  for i = 1 to n do
9  |   w = heap.DeleteMin();
10 |   if w.start ≥ last then
11 |   |   W.Enqueue(w);
12 |   |   last = w.end;
13 end
14 return W;
```

Workshop	Start	End
w_1	7	10
w_2	1	8
w_3	4	7
w_4	3	5
w_5	2	4

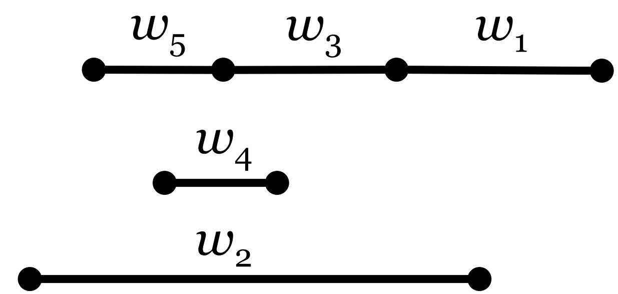


1. Draw the contents of *heap* on the set of workshops above:
   a) after each iteration of the for loop in lines 4-6.
   b) after each iteration of the for loop in lines 8-13.

Back

Heap exercise solution

Workshop	w_1	w_2	w_3	w_4	w_5
Start	7	1	4	3	2
End	10	8	7	5	4



Lines 4-6	i=1:		w_1				
	2:		w_2	w_1			
	3:		w_3	w_1	w_2		
	4:		w_4	w_3	w_2	w_1	
	5:		w_5	w_4	w_2	w_1	w_3
Lines 8-13	i=1:		w_4	w_3	w_2	w_1	
	2:		w_3	w_1	w_2		
	3:		w_2	w_1			
	4:		w_1				
	5:						

Back

Union-Find operations

- **Initialize**
  - Assigns every element to its own partition
  - $O(n)$
- **Find(x)**
  - Follow links to partition ID (root)
  - Recursively point to root
  - $O(\alpha(n))$
  - Generally less than 5 for conceivable n
- **Union(a, b)**
  - Find root of both sides
  - Point to max root to min
  - $O(\alpha(n))$

```
Algorithm: UnionFind(n)
unionfind = Array(n);
for i = 1 to n do
    unionfind[i] = i;
end
return unionfind;
```

```
Algorithm: Find(x)
if unionfind[x] ≠ x then
    id = Find(unionfind[x]);
    unionfind[x] = id;
end
return unionfind[x];
```

```
Algorithm: Union(a, b)
ra = Find(a);
rb = Find(b);
if ra > rb then
    Swap ra and rb;
end
unionfind[ra] = rb;
```

Back

Union-Find exercise

- **Problem:** blob counting
- **Input:** an n by n matrix of integers 1-4
- **Output:** number of contiguous regions of the same integer
  - Contiguous: cells adjacent horizontally or vertically
- **Example:** n = 5, 4 blobs

1	1	3	3	3
1	2	1	3	3
2	2	1	1	3
2	2	1	3	3
2	1	1	1	3

1. Design an algorithm to count blobs
2. Analyze its complexity

- *Hint:* number your "pixels":
  - A[r, c] -> rn + c

$$\begin{array}{ccccc} 0 & 1 & 2 & \dots & n-1 \\ n & n+1 & n+2 & \dots & 2n-1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \dots & \dots & \dots & \dots & n^2-1 \end{array}$$

Back

Union-Find exercise solution

- **Main idea:** use Union-Find to keep track of blobs
- **Pseudocode**
  - Initialize Union-Find
  - Iterate through all n^2 cells
  - Union with cells above, below, left, and right if they have same color
    - More clever: just check right and down (or up and left)
  - Afterwards, count number of distinct partition IDs
    - More clever: if they were distinct before, Union reduces the number of blobs by 1
    - Count backwards from n^2
- **Analysis**
  - Initialize: $O(n^2)$
  - First loop: n^2 iterations, $O(\alpha(n^2))$ time -> $O(n^2\alpha(n^2))$
  - Second loop: $O(n^2\alpha(n^2))$, if using bitmap
  - Total: $O(n^2\alpha(n^2)) = O(n^2\alpha(n))$

Back

Union-Find algorithm

Input: n: size of input matrix
Input: A: $n \times n$ matrix in which to count blobs
Output: the number of blobs in A
Algorithm: CleverBlobCount

```
uf = UnionFind(n^2);
blobs = n^2;
for r = 0 to n − 1 do
    for c = 0 to n − 1 do
        x = rn + c;
        if r < n − 1 then
            right = rn + c + 1;
            if A[r, c] = A[r, c + 1] and uf.Find(x) ≠ uf.Find(right) then
                uf.Union(x, right);
                blobs = blobs − 1;
        if c < n − 1 then
            down = (r + 1)n + c;
            if A[r, c] = A[r + 1, c] and uf.Find(x) ≠ uf.Find(down) then
                uf.Union(x, down);
                blobs = blobs − 1;
    end
end
return blobs;
```

Complexity: $O(n^2\alpha(n^2)) = O(n^2\alpha(n))$

Back

Divide-and-conquer

- *Intuition:* combining solutions is sometimes easier than solving directly

- Solve small problems directly (*base case*)
- Divide large problem into one or more subproblems
  - E.g., split array into 2 halves, 3 thirds, etc.
- Solve subproblems recursively
- Combine solutions to subproblems into solution for full problem

- Easy to prove correctness via strong induction
- Good for parallel algorithms
- Doesn’t work if you can’t solve problem by combining partial solutions

Back

Divide-and-conquer exercise

- **Problem:** matrix multiplication (square matrices)
  - Naïve algorithm: $O(n^3)$
- **Divide-and-conquer algorithm:** Strassen's algorithm
  - Split both matrices into 4 quarters: $\begin{array}{|c|c|}\hline A_1 & B_1 \\ \hline C_1 & D_1 \\ \hline \end{array}$, $\begin{array}{|c|c|}\hline A_2 & B_2 \\ \hline C_2 & D_2 \\ \hline \end{array}$
  - Calculate the following matrices:
    - 7 multiplications
    - 6 additions
    - 4 subtractions

$$\begin{aligned}
M_1 &= (A_1 + D_1)(A_2 + D_2) \\
M_2 &= (C_1 + D_1)A_2 \\
M_3 &= A_1(B_2 - D_2) \\
M_4 &= D_1(C_2 - A_2) \\
M_5 &= (A_1 + B_1)D_2 \\
M_6 &= (C_1 - A_1)(A_2 + B_2) \\
M_7 &= (B_1 - D_1)(C_2 + D_2)
\end{aligned}$$

  - Calculate the 4 quarters of the result:
    - 6 additions
    - 2 subtractions

$$\begin{aligned}
A_3 &= M_1 + M_4 - M_5 + M_7 \\
B_3 &= M_3 + M_5 \\
C_3 &= M_2 + M_4 \\
D_3 &= M_1 - M_2 + M_3 + M_6
\end{aligned}$$

Back

Divide-and-conquer exercise

- *S(n)*: time to multiply two *n* by *n* matrices

1. Write a recurrence for *S(n)*
   - Split matrices into 4 quarters
   - Calculate 7 intermediate products
     - 7 multiplications
     - 10 addition/subtraction
   - Calculate 4 quarters of result
     - 8 addition/subtraction

2. Solve the recurrence for *S(n)*
   a) Identify a, b, and $f(n)$
   b) Calculate $c = \log_b(a)$
   c) Compare $f(n)$ to n^c
   d) Apply Master Theorem

Back

Divide-and-conquer exercise solution

- *S(n)*: time to multiply two *n* by *n* matrices

1. Write a recurrence for *S(n)*
   - Split matrices into 4 quarters $\Theta(n^2)$ (copy) OR $\Theta(1)$ (offsets)
   - Calculate 7 intermediate products
     - 7 multiplications $7S(n/2)$
     - 10 addition/subtraction $\Theta(n^2)$
   - Calculate 4 quarters of result $\Theta(n^2)$
     - 8 addition/subtraction

   $$S(n) = 7S(n/2) + \Theta(n^2)$$

2. Solve the recurrence for *S(n)*
   a) Identify *a*, *b*, and *f(n)* $a = 7,\ b = 2,\ f(n) = \Theta(n^2)$
   b) Calculate $c = \log_b(a)$ $c = \log_b(a) = \lg(7) \approx 2.81$
   c) Compare *f(n)* to n^c $f(n) = O(n^{\lg 7 - 0.8})$
   d) Apply Master Theorem $S(n) = \Theta(n^{\lg 7}) \approx \Theta(n^{2.81})$

Coming up

- **Exam 2** will be Tuesday
  - Practice Exam 2 sample solution posted on Canvas
- Exam review: Monday at 5 (CHE 100)
- After exam: sorting algorithms
- **Project 1** will be due Oct. 18

- **Practice problems:** 3-26, 3-29 (p 102), 4-43 (p. 144)
- **Recommended readings (Thursday):** Sections 4.9, 4.6, and 4.7