

CS430 Lecture 01 Activities

Opening Questions

1. Algorithm A takes 5 seconds to sort 1000 records, and Algorithm B takes 10 seconds to sort 1000 records. You have the code for both algorithms. When deciding which algorithm to use to sort up to 1,000,000 records, why might Algorithm B be the better choice?
2. Why is it helpful to sometimes define a problem in its most basic mathematical terms?
3. In your own words explain what a loop invariant is.
4. What are the three kinds of growth in run time analysis we may do on an algorithm.
5. For recursive algorithms, what do we need to define and solve to do the runtime analysis?

Sorting as a Case Study for Algorithm Analysis

1. Write pseudocode for one of these iterative sorts: InsertionSort, BubbleSort, SelectionSort. Then draw pictures for a sample run on 5 random numbers showing comparisons/swaps.

2. Write the loop invariant for your sort and prove your sort is correct by proving Initialization, Maintenance, and Termination.

An important property of sorting algorithms is whether or not it is **stable**: numbers with the same value appear in the output array in the same order as they do in the input array.

3. Is your sort stable?

Resource (memory or runtime) Use Analysis – Resource use analysis usually depends on the size of the input to the algorithm. You can write a function $T(n)$ that matches the behavior of the resource use of the algorithm.

NOTE: For recursive algorithms we develop and solve a recurrence relation to find the $T(n)$, the resource use function.

4. For the iterative sort you wrote above, construct a run time analysis function $T(n)$ by assigning a different constant (c_1 , c_2 , c_3 , etc) to each type of statement (the run time for statements of that type), and counting how many times each statement executes for an input size n . Then sum up the constants times the execution counts. You may need to define variables other than n if there are event controlled iterations with an unknown number of loops.

For a more descriptive analysis we need to consider the various generic execution flows and input structures that result in those generic execution flows. There are 3 more descriptive resource use analyses: 1) worst case (usually used) 2) average case (sometimes used) 3) best case (hardly ever used)

5. For your $T(n)$ function from above determine the best case, worst case and average case $T(n)$ s.

CS430 Lecture 02 Activities

Asymptotic Analysis – To simplify comparing the resource usage of different algorithms for the same problem

- ignore machine dependent constants; look at the growth of $T(n)$ as $n \rightarrow \infty$
- as you double n , what does $T(n)$ do?? double?? square??

Theta Notation (more details in future lectures)

- Drop lower order terms;
- Ignore leading constants
- Concentrates on the growth

1. For your best case, average case, worst case $T(n)$ functions from above give the asymptotic function (Theta notation)

2. Given the problem sizes and worst case runtime for one of the problem sizes, and what you know about each algorithm, predict the missing runtimes.

	n=100	n=200	n=400	n=800
Linear search	10 seconds			
Binary search		8 seconds		
Insertion Sort			320 seconds	

Opening Questions - Average Case Runtime

3. How did we approach average case runtime analysis of iterative algorithms previously? How can we improve on this?

Expectation of a Random Variable

A random variable is a variable that maps an outcome of a random process to a number. Examples:

- Flipping a coin, If heads $X=1$, if tails $X=0$
- Y =sum of 7 rolls of a fair die
- Z =in insertion sort, the number of swaps needed to move the i th item to its correct position in items 1 thru $(i-1)$

The expected value of a random variable X is sum over all outcomes of the value of the outcome times the probability of the outcome.

$$E(X) = \sum_{s \in S} X(s)p(s)$$

4. What is the expected outcome when you roll a fair die once? What about a loaded die where the probability of a side coming up is the value of the side divided by 21?

5. Calculate the expected outcome when you roll a fair die twice and sum the results. Do this two different ways.

Now let's use expectation of a random variable to improve our average case runtime for insertion sort (similar for bubble sort or selection sort).

- Sort n distinct elements using insertion sort
- X_i is the random variable equal to the number of comparisons used to insert a_i into the proper position after the first $i-1$ elements have already been sorted. $1 \leq X_i \leq i-1$

$E(X_i)$ is expected number of comparisons to insert a_i into the proper position after the first $i-1$ elements have been sorted.

$E(X) = E(X_2) + E(X_3) + \dots + E(X_n)$ is the expected number of comparisons to complete the sort (our new average case runtime function).

6. Write equations for the following and simplify.

$E(X_i)$

$E(X)$

7. What if the data is not random?

CS430 Lecture 03 Activities

Asymptotic Analysis (more details)

BIG-O Notation – Upper bound on growth of a runtime function

$$f(n) \in O(g(n)) \quad \text{"f(n) is big-O of g(n)"}$$

If there exists C, n_0 such that

$$0 < f(n) < Cg(n) \quad \text{when } n > n_0$$

1a. Use the definition of big-O to show $2n^2$ is big-O n^3 (find a C and n_0 that works in the above)

1b. Use the definition of big-O to show $T(n)=3n^3-4n^2+3\lg n-n = O(n^3)$

Omega Ω – lower bound

$$f(n) = \Omega(g(n))$$

$$0 \leq cg(n) \leq f(n)$$

$$c? \quad n_0 < n$$

2. Use the definition of omega to show $n^{1/2} = \Omega(\log n)$

THETA θ Notation – Strict Bound

$$f(n) = \theta(g(n))$$

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

$$c_1 ? < c_2 ?$$

3a. Use the definition of theta to show $3n^3 - 4n^2 + 37n = \Theta(n^3)$

3b. Use the definition of theta to show $n^2 + 3n^3 = \Theta(n^3)$

Recursive Sorting – Mergesort

- divide and conquer (and combine) approach, recursive algorithm
- key idea: you can merge two sorted lists of total length n in $\Theta(n)$ linear time
- base case: a list of length one element is sorted

1. Demonstrate how you can merge two sorted sub-lists total n items with n compares/copies. How much memory do we need to do this? Write pseudocode to do this.

2 3 7 8 1 4 5 6

--	--	--	--	--	--	--	--

```
Mergesort(A, p, r) { // initial call Mergesort (A, 1, n)
  if (p<r) {
    q = (p+r)/2    // integer division
    Mergesort(A, p, q) // recursively sort 1st half
    Mergesort(A, q+1, r) // recursively sort 2nd half
    Merge(A, p, q, r) // merge 2 sorted sub-lists
  }
}
```

2. Demonstrate Mergesort on this data

3	41	52	26	38	57	09	49
---	----	----	----	----	----	----	----

CS430 Lecture 04 Activities

1. A recurrence relation describes runtime function recursively for a recursive algorithm. Write a recurrence relation for the Mergesort algorithm. HINT: try to count the number of executions of each statement and the cost of each statement.

Solving Recurrence Relations – Recurrence Tree Method

We solve a recurrence relation to get a function in its closed (non-recursive) form. The recurrence tree method is a visual method of repeatedly substituting in the recurrence relation for $T(n)$ on smaller and smaller “ n ” until you reach the base case, and then summing up all the nodes in the tree.

2. Draw the recurrence tree for Mergesort

Divide and Conquer Algorithms

- Divide – divide the problem into sub-problems that can be solved independently
- Conquer – recursively solve each sub-problem
- Combine – possibly necessary, combine solutions to sub-problems

Not all problems can be solved with the divide and conquer approach. Maybe sub-problems are not independent, or solutions to sub-problems cannot be combined to find solution to main problem.

3. Write a recursive algorithm for Binary Search. Write and solve its recurrence relation.

4. Write a recursive algorithm for Selection Sort (or insertion sort or bubble sort). Write and solve its recurrence relation.

5. Describe an efficient divide and conquer algorithm to count the number of times a character appears in a string of length n .

Inductive Proofs (needed in next lecture to prove a solution to a recurrence relation)

6. What are the three steps in an inductive proof?

7. Use an inductive proof to show the sum of the first n integers is $(n)(n+1)/2$

CS430 Lecture 05 Activities

Opening Questions

1. Define Big-O, Omega and Theta notation.

2. In your own words explain what a recurrence relation is, what do we use recurrence relations for, why do we solve recurrence relations?

Recurrence Relation Solution Approach - Guess and prove by induction

Guess (or are given Hint) at form of solution, prove it is the solution

- Using definition of BIG-O or θ
- Using Induction
 - Prove Base Case (if boundary condition given)
 - Assume true for some “n”
 - Prove true for a larger “n”

EXAMPLE

$T(n) = 4T(n/2) + n$ guess $T(n) = O(n^3)$??

Assume $T(k) \leq ck^3$ for some $k < n$, use assumption with $k = n/2$, then prove it for $k = n$

$T(n/2) \leq c(n/2)^3$ merge with recurrence

$T(n) \leq 4c(n/2)^3 + n$

$T(n) \leq c/2(n)^3 + n$

$T(n) \leq cn^3 - (c/2(n)^3 - n)$

$(c/2(n)^3 - n) > 0$ if $c \geq 2$ and $n > 1$

$T(n) \leq cn^3 - (\text{something positive})$

$T(n) \leq cn^3$

$T(n) = O(n^3)$

1. $T(n) = 4T(n/2) + n^3$ guess $T(n) = \Theta(n^3)$??

2. $T(n) = 4T(n/2) + n$ guess $T(n) = O(n^2)$??

Recurrence Relation Solution Approach - Iteration Method (repeated substitution)

Convert the recurrence relation to summation using repeated substitution (Iterations)

- Keys to Iteration Method
 - # of times iterated to get $T(1)$
 - Find the pattern in terms and simplify to summation

EXAMPLE

$$\begin{aligned}T(n) &= 4T\left(\frac{n}{2}\right) + n \\T(n) &= n + 4T\left(\frac{n}{2}\right) \\&= n + 4\left(4T\left(\frac{n}{4}\right) + \frac{n}{2}\right) \\&= n + 4\left(\frac{n}{2} + 4\left(\frac{n}{4} + 4T\left(\frac{n}{8}\right)\right)\right) \\&= n + 2n + 4n + 64T\left(\frac{n}{8}\right) \\T(n) &= n + 2n + 4n + \dots \dots \dots 4^{\lg_2 n} T(1) \\&= n \sum_{k=0}^{\lg_2 n - 1} 2^k + \theta(n^2) \qquad \qquad \qquad 2^{\lg_2 n} = n \\&= n \left(\frac{2^{\lg_2 n} - 1}{2 - 1} \right) \qquad \qquad \qquad 4^{\lg_2 n} = n^2 \\T(n) &= n^2 - n = O(n^2)\end{aligned}$$

3. $T(n) = 3T(n/3) + \lg n$ proof by iteration/repeated substitution

4.
$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$

5.
$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + n^2$$

6. $T(n) = T(n-1) + n$

Recurrence Relation Solution Approach - Master Method

For solving recurrences of the form $T(n) = aT(n/b) + f(n)$

- Compare growth of $f(n)$ to $n^{\log_b a}$
 - Case1 $f(n) \leq cn^{\log_b a}$ $T(n) = \Theta(n^{\log_b a})$
 - Case2 $c_1 n^{\log_b a} \leq f(n) \leq c_2 n^{\log_b a}$ $T(n) = \Theta(n^{\log_b a} \lg n)$
 - Case3 $f(n) > cn^{\log_b a}$ $T(n) = \Theta(f(n))$

7.
$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

8.
$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

9.
$$T(n) = 4T\left(\frac{n}{2}\right) + n^3$$

CS430 Lecture 06 Activities

Opening Questions

1. Mergesort is $\Theta(n \lg n)$ runtime in best case, worst case and average case. How much memory is needed for Mergesort on input size n ?
2. Mergesort does all the work of sorting items in the Merge function, after recursively splitting the collection down to the base case. Briefly explain the difference with Quicksort.

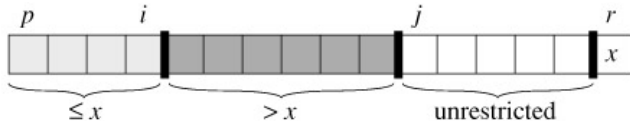
Quicksort

A recursive divide and conquer sorting algorithm.

- base case: a list of length one element is sorted
- Divide "Partition" array into 2 sub-arrays with small #'s in first, large #'s in second and known index dividing them
- Conquer - recursively sort each sub-array
- Combine - Nothing to do

1. Write recursive pseudocode for Quicksort (similar to Mergesort).

Partition idea (you should be able to do this in place): pick the last element in the current array as the "pivot", the number used to decide large or small. Then make a single pass of the array to move the "small" numbers before the "large" numbers and keep the "large" numbers after the "small" numbers. Then put the "pivot" between the two subarrays and return the location of the pivot.



2. Write iterative pseudocode for Partition. How much memory is needed?

3. Demonstrate Partition on this array.

2	8	7	1	3	5	6	4
---	---	---	---	---	---	---	---

4. What do you think the best possible outcome would be for a call to Partition, and why? What about worst possible outcome?

5. Write (and solve) recurrence relations for Quicksort in the best case partition and worst case partition.

6. What if there is a pretty bad, but not awful, partition at every call. Try always a 9 to 1 split from partition. Write and solve recurrence relation.

With all the other sorts we could describe a particular input order that would yield worst case run time.

7. How can we avoid a particular input order yielding worst case run time for quicksort?

Visual Sorting Software By A. Alegoz, previous CS430 student (1.8Mb zipped, Win only)

<http://www.cs.iit.edu/~cs430/IITSort.zip>

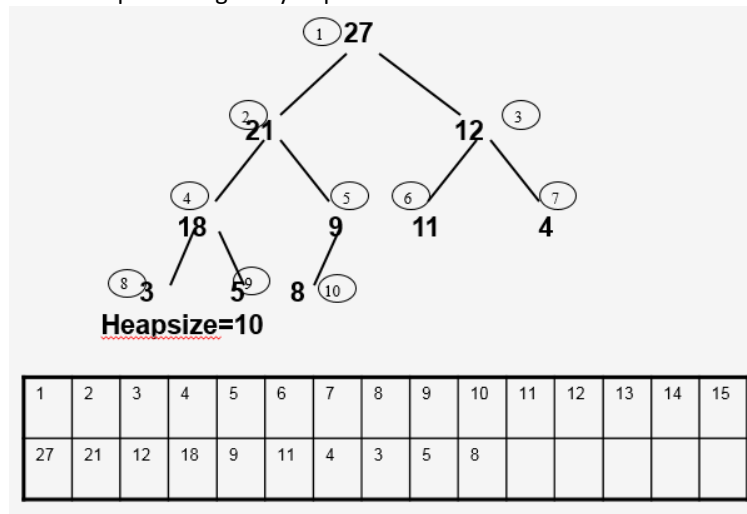
CS430 Lecture 07 Activities

Opening Questions

1. Explain the difference between the Binary Search Tree Parent-Child value relationship and the Heap Property Parent-Child value relationship.
2. Binary search trees are a dynamic data structure that uses left-child and right-child pointers to represent the tree. How is this different in a heap?

Heaps

Since a heap is a nearly complete binary tree and will always grow and shrink in the rightmost bottom leaf you can implement a heap with an array instead of needing pointers (as is needed for a binary search tree which can grow/shrink at any node. Example of a MaxHeap showing array implementation:



1. At what index position is the largest element in a MaxHeap? We have to know how to easily move around a Heap. Can you devise a formula to relate the index of a parent to the indexes of its children? How about a formula for the index of a child to the index of its parent?

2. If a heap was one larger, where does tree have to gain a node from when done? If a heap was one smaller, where does tree have to lose a node when done?

3. Considering your answer to #2, try to devise a way to ExtractMax from this maxheap. What is the runtime in terms of heapsize?

21
18 12
8 9 11 4
3 5

Heapsize=9 array A

21	18	12	8	9	11	4	3	5					
----	----	----	---	---	----	---	---	---	--	--	--	--	--

4. Considering your answer to #2, try to devise a way to Insert(20) into this maxheap. What is the runtime in terms of heapsize?

21
18 12
8 9 11 4
3 5

Heapsize=9 array A

21	18	12	8	9	11	4	3	5					
----	----	----	---	---	----	---	---	---	--	--	--	--	--

Both the above extractMax and Insert assumed we already had a heap. To efficiently build a heap we put all the items to insert in an array. Call MaxHeapify (walk value down) from index $\text{heapsize}/2$ up to root (index 1).

5. Write pseudocode for this BuildHeap algorithm and demonstrate on this data. What is the runtime in terms of n , the size of B ?
 $B=[15 \ 8 \ 4 \ 9 \ 3 \ 16]$

6. Write the loop invariant for BuildHeap and prove that it works.

7. How can we use a maxHeap and extractMax to sort?

CS430 Lecture 08 Activities

Opening Questions

1. If we have 17 distinct items to sort, what is the height of the decision tree that represents all possible orderings of the 17 items?

Comparison Sorts

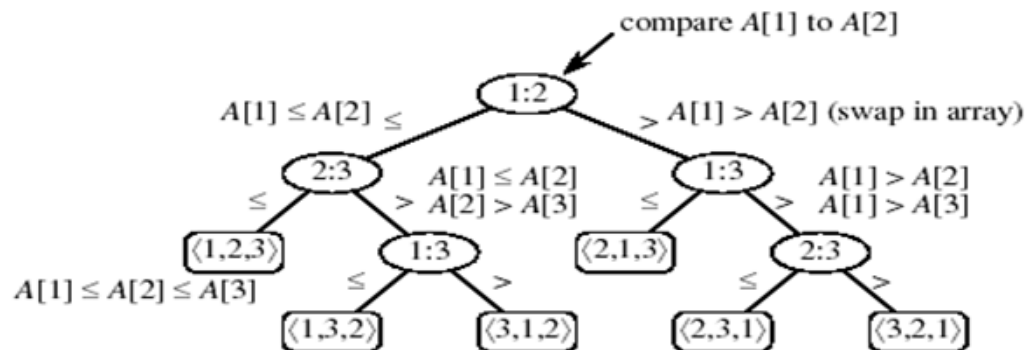
The sorted order they determine is based only on comparisons between the input elements. That is, given two elements a_i and a_j , we perform one of the tests $a_i < a_j$, $a_i = a_j$, $a_i = a_j$, or $a_i > a_j$ to determine their relative order.

Claim: We must make at least $\Omega(n \lg n)$ comparisons in the general case

The decision-tree model

- Abstraction of any comparison sort.
- Represents comparisons made by a specific sorting algorithm on inputs of a given size.
- Abstracts away everything else: control and data movement.
- View the tree as if the algorithm splits in two at each node, based on the information it has determined up to that point; The tree models all possible execution traces

Each internal node is labeled by indices of array elements **from their original positions**. Each leaf is labeled by the permutation of orders that the algorithm determines.



1. How many possible permutations of n items are there (at least one of them must be sorted)? In the decision tree model how many leaves are there for n items? For a binary tree to have that many leaves how many levels does it need? Why do we care how many levels are in the decision tree?

Linear Time Sorting – Counting Sort

If we do not use comparison of keys to sort data how can we sort data? It seems comparing items is necessary for sorting.

- Counting sort assumes that each of the n input elements is an integer in the range 0 to k , for some integer k . When $k = O(n)$, the sort runs in $T(n)$ time (best if $k \ll n$)
- For each input element “ x ”, count how many elements are equal to “ x ” then use this to count how many elements are \leq “ x ”. This information can be used to place element x directly into its position in the output array.
- Does not sort in place, needs 2nd array size “ n ” and 3rd array size “ k ”

2. Write pseudocode for counting sort.

3. Demo counting sort on this data:

2	5	3	0	2	3	0	3
---	---	---	---	---	---	---	---

Counting sort is best on large amounts of data where the range of the data is small. Like many single digit numbers. However we can use counting sort multiples times to sort multiple digit numbers, starting with the right most digit, etc. This is called Radix Sort

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

An important property of counting sort is that it is **stable**: numbers with the same value appear in the output array in the same order as they do in the input array. That is, it breaks ties between two numbers by the rule that whichever number appears first in the input array appears first in the output array. Normally, the property of stability is important only when satellite data are carried around with the element being sorted. Counting sort’s stability is important for another reason: counting sort is often used as a subroutine in radix sort. As we shall see in the next section, in order for radix sort to work correctly, counting sort must be stable.

4. Which of these sorts can be stable InsertionSort, SelectionSort, BubbleSort, MergeSort, QuickSort, HeapSort?

CS430 Lecture 09 Activities

Opening Questions

1. Order Statistics: Select the i th smallest of n elements (the element with rank i)

$i = 1$: minimum;

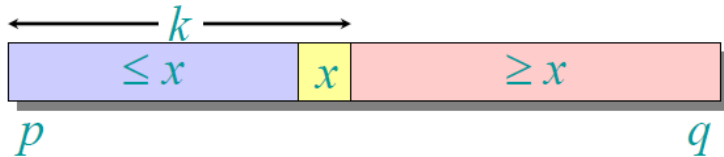
$i = n$: maximum;

$i = \text{floor}((n+1)/2)$ or $\text{ceiling}((n+1)/2)$: median

How fast can we solve the problem for various i values?

Randomized Algorithm for finding the i th Element

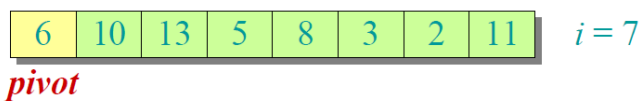
1. Think about partition (with a random choice of the pivot) from quicksort. Can you think of a way to use that and comparing the final location of the pivot to i , and then divide and conquer?



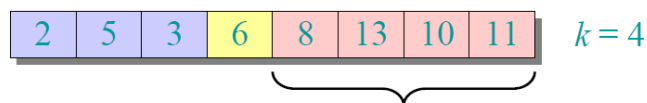
2. Demonstrate on this array to find i -th smallest element

6	10	13	5	8	3	2	11
---	----	----	---	---	---	---	----

Select the $i = 7$ th smallest:



Partition:



Select the $7 - 4 = 3$ rd smallest recursively.

3. What is the worst case running time if you find the i th smallest element?

Is there an algorithm to find the i th smallest element that runs in linear time in the worst case?

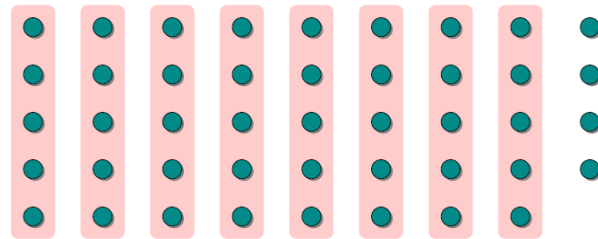
SELECT(i, n)

1. Divide the n elements into groups of 5. Find the median of each 5-element group by hand.
2. Recursively **SELECT** the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.
3. Partition around the pivot x . Let $k = \text{rank}(x)$.
4. **if** $i = k$ **then return** x
 elseif $i < k$
 then recursively **SELECT** the i th smallest element in the lower part
 else recursively **SELECT** the $(i-k)$ th smallest element in the upper part

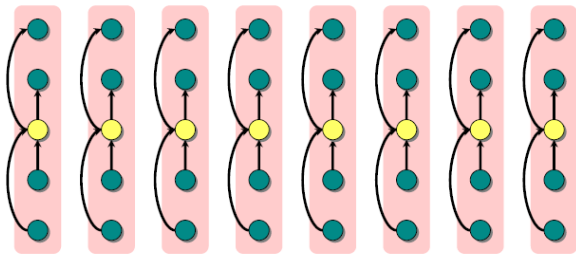
Choosing the pivot



Choosing the pivot

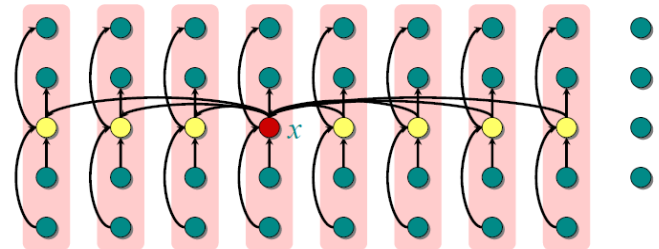


1. Divide the n elements into groups of 5.



1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.

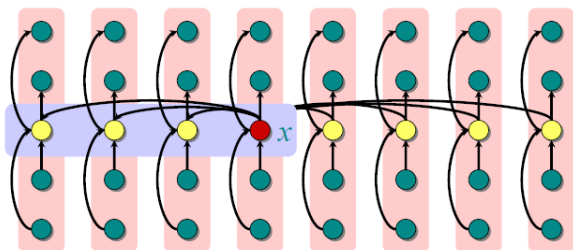
lesser
greater



1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.
2. Recursively **SELECT** the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.

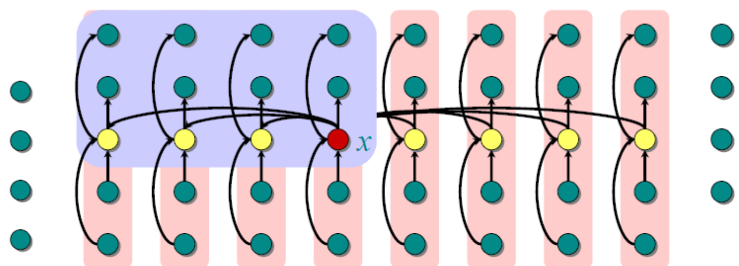
lesser
greater

Analysis



At least half the group medians are $\leq x$, which is at least $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$ group medians.

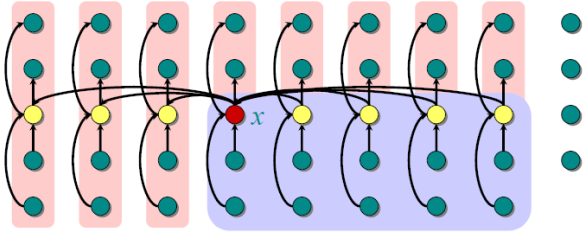
Analysis



At least half the group medians are $\leq x$, which is at least $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$ group medians.
 • Therefore, at least $3 \lfloor n/10 \rfloor$ elements are $\leq x$.

Developing the recurrence

Analysis



At least half the group medians are $\leq x$, which is at least $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$ group medians.

- Therefore, at least $3\lfloor n/10 \rfloor$ elements are $\leq x$.
- Similarly, at least $3\lfloor n/10 \rfloor$ elements are $\geq x$.

$$\begin{array}{l}
 \underline{T(n)} \quad \text{SELECT}(i, n) \\
 \Theta(n) \quad \left\{ \begin{array}{l} 1. \text{ Divide the } n \text{ elements into groups of } 5. \text{ Find} \\ \text{the median of each } 5\text{-element group by rote.} \end{array} \right. \\
 T(n/5) \quad \left\{ \begin{array}{l} 2. \text{ Recursively SELECT the median } x \text{ of the } \lfloor n/5 \rfloor \\ \text{group medians to be the pivot.} \end{array} \right. \\
 \Theta(n) \quad \left\{ \begin{array}{l} 3. \text{ Partition around the pivot } x. \text{ Let } k = \text{rank}(x). \\ 4. \text{ if } i = k \text{ then return } x \\ \text{elseif } i < k \\ \text{then recursively SELECT the } i\text{th} \\ \text{smallest element in the lower part} \\ \text{else recursively SELECT the } (i-k)\text{th} \\ \text{smallest element in the upper part} \end{array} \right. \\
 T(7n/10)
 \end{array}$$

Solving the recurrence

$$T(n) = T\left(\frac{1}{5}n\right) + T\left(\frac{7}{10}n\right) + \Theta(n)$$

Substitution:

$$T(n) \leq cn$$

$$T(n) \leq \frac{1}{5}cn + \frac{7}{10}cn + \Theta(n)$$

$$= \frac{18}{20}cn + \Theta(n)$$

$$= cn - \left(\frac{2}{20}cn - \Theta(n)\right)$$

$$\leq cn$$

if c is chosen large enough to handle the $\Theta(n)$.

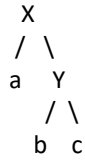
In practice, this algorithm runs slowly, because the constant in front of n is large.

Would we use this approach to find the median to partition around in Quicksort, and achieve in worst-case Theta ($n \log n$) time?

CS430 Lecture 10 Activities

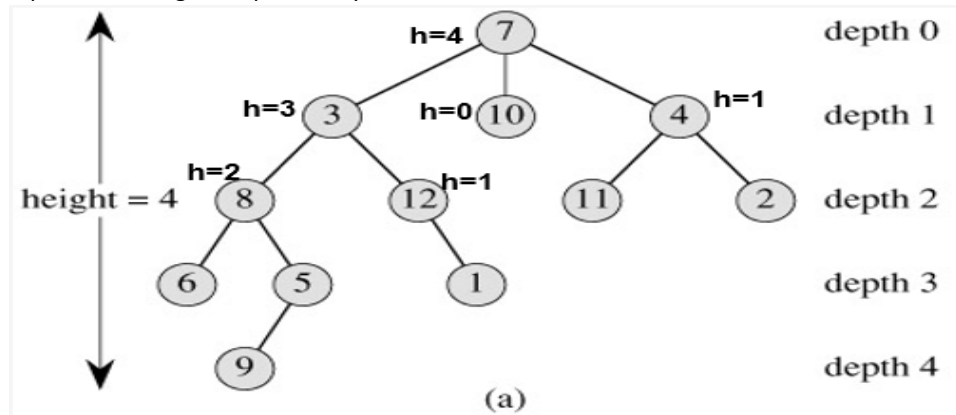
Opening Questions

1. In your own words explain how you insert a new key in a binary search tree.
2. Give an example of a series of 5 keys inserted one at a time into a binary search tree that will yield a tree of height 5.
3. If we do a left rotate on node X below, explain which left and/or right child links need to be changed



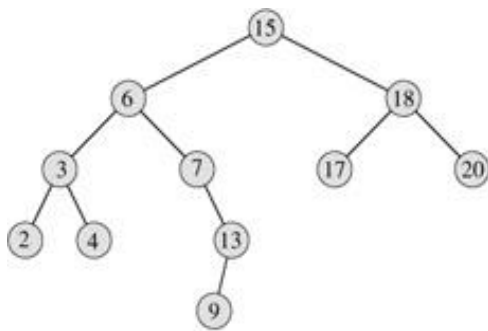
Tree depth vs Tree height

The length of the path from the root r to a node x is the depth of x in T . The height of a node in a tree is the number of edges on the longest simple downward path from the node to a leaf, and the height of a tree is the height of its root. The height of a tree is also equal to the largest depth of any node in the tree.

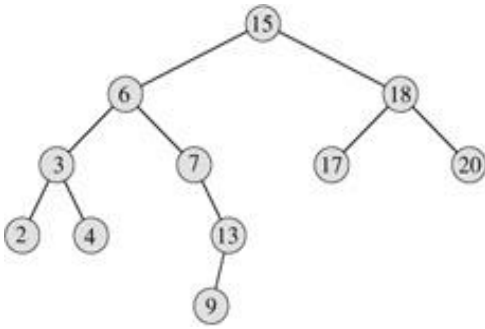


BST Operations

1. Write pseudocode for BST Successor (or Predecessor). Demonstrate on the below tree from node 15 and then node 13.



2. Write pseudocode for BST Insert. Demonstrate on the below tree to insert 5 and then 19.



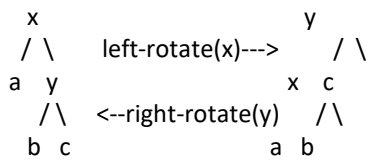
3. What are the three possible cases when deleting a node from a BST?

4. Write pseudocode for BST delete (assume you already have a pointer to the node)

BST Rotations

Local operation in a search tree that maintains the BST property and possibly alters the height of the BST.

x and y are nodes; a, b, c are sub trees



5. Write pseudocode for LeftRotate (or RightRotate). What is the worst case runtime?

CS430 Lecture 11 Activities

Opening Questions

1. What is the main problem with Binary Search Trees that Red-Black Trees correct? Explain briefly (2-3 sentences) how Red-Black Trees correct this problem with Binary Search Trees.
2. For the balanced binary search trees, why is it important that we can show that a rotation at a node in a is $O(1)$ (i.e. not dependent on the size the BST)

Red-Black Trees

Red-Black Properties

1. Every node is colored either red or black
2. The root is black
3. Every null pointer descending from a leaf is considered to be a null black leaf node
4. If a node is red, then both its children are black
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes (black height)

black-height of a node $bh(x)$ - The number of black nodes on any path from, but not including, a node "x" down to a null black leaf node

1. If a node x has $bh(x)=3$, what is its largest and smallest possible height (distance to farthest leaf) in the BST?
2. Prove using induction and red-black tree properties. A red-black tree with n internal nodes (n key values) has height at most $2\lg(n+1)$

Part A - First show the sub-tree rooted at node "x " has at least $2^{bh(x)}-1$ internal nodes. Use induction

Part B – Let “h” be height of R-B Tree, by property 4 at least half the nodes on path from root to leaf are black

$bh(\text{root}) \geq h/2$

Use that and part A to show $h \leq 2\log(n+1)$

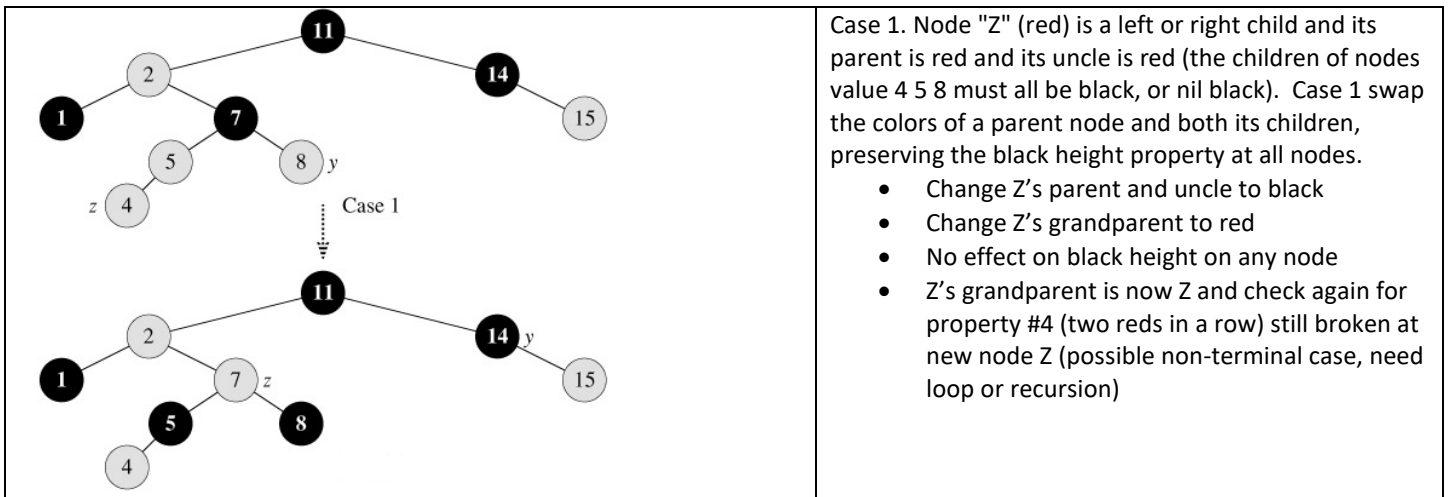
3. Which BST operations change for a red-black tree and which do not change? What do the operations that change need to be aware of and why?

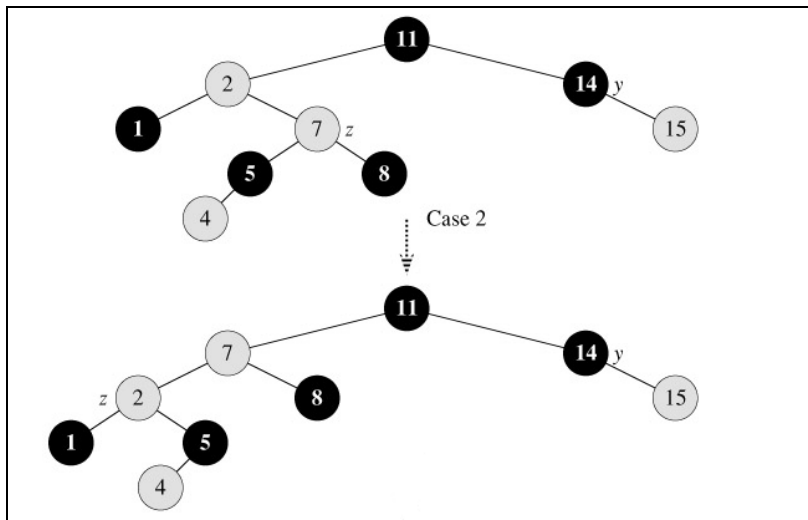
search, insert, delete, predecessor, successor, minimum, maximum, rotations

Red-Black Tree Insert - Similar to BST insert, assume we start with a valid red-black tree.

1. Locate leaf position to insert new node
2. Color new node red and create 2 new black nil leaves below newly inserted red node
3. If parent of new insert was _____ (fill in the blank, black or red), then done. ELSE procedure to recolor nodes and perform rotations to maintain red-black properties.

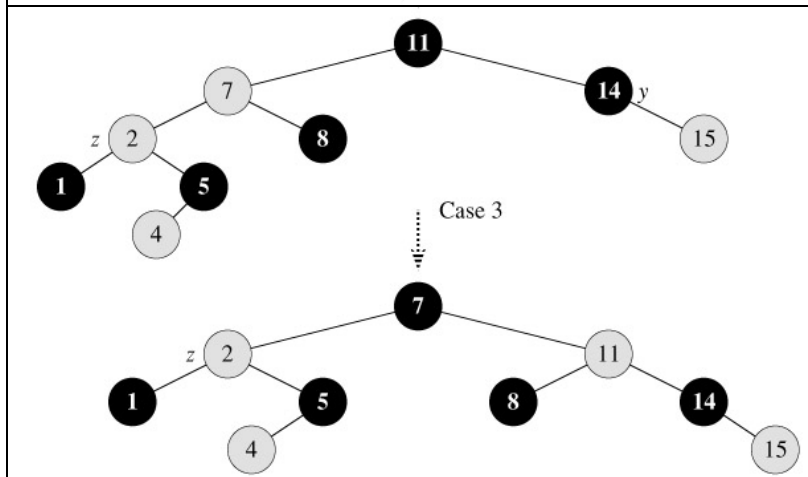
There are three cases if R-B Property #4 broken when insert a red node "Z" (or changed color of a node to red) and its parent is also red.





Case 2. Node "Z" is a right child and its parent is red and its uncle is NOT red. Case 2 does a single rotation, preserving the black height property at all nodes.

- Rotate left on parent of Z
- Re-label old parent of Z as Z and continue to case #3



Case 3. Node "Z" is a left child and its parent is red and its uncle is NOT red. Case 3 does a single rotation and swaps the colors of a parent node and both its children, preserving the black height property at all nodes.

- Rotate right on grandparent of Z
- Color old parent of Z black
- Color old grandparent of Z red

CS430 Lecture 12 Activities

Opening Questions

1. What do you think the issue we need to handle when deleting a node from a red-black tree? How does red-black delete differ from a BST delete?

Red-Black Tree Delete

Think of V as having an “extra” unit of blackness. This extra blackness must be absorbed into the tree (by a red node), or propagated up to the root and out of the tree. There are four cases – our examples and “rules” assume that V is a left child. There are symmetric cases for V as a right child

Terminology in Examples

- The node just deleted was U
- The node that replaces it is V, which has an extra unit of blackness
- The parent of V is P
- The sibling of V is S



Black Node



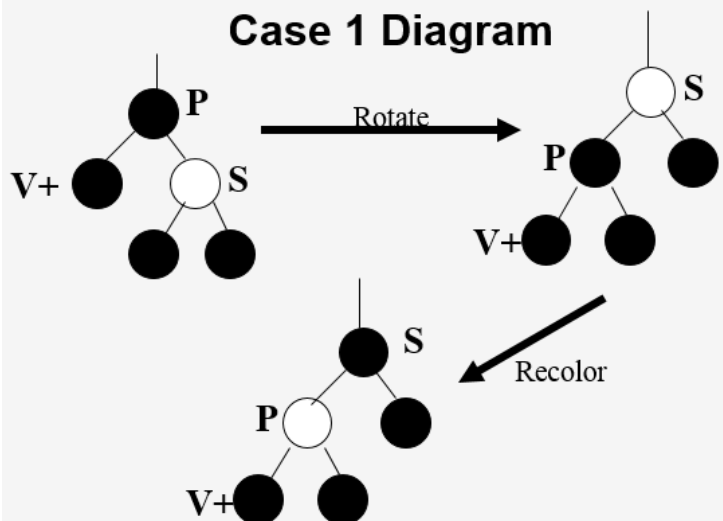
Red or Black and don't care

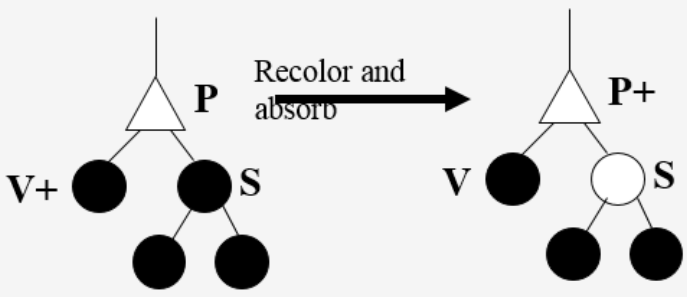
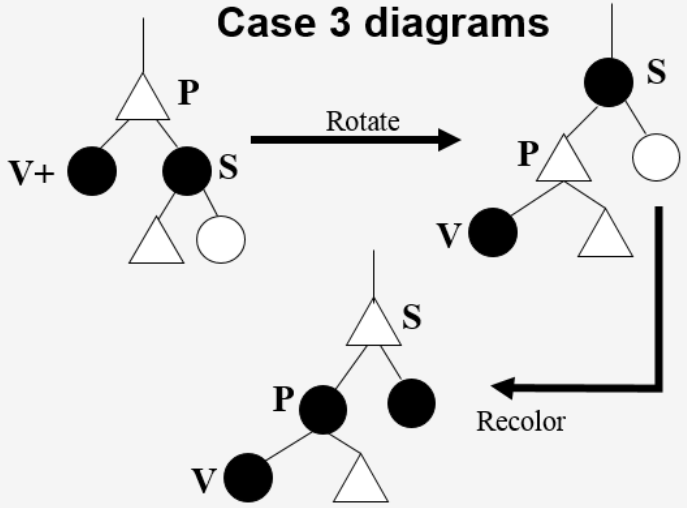
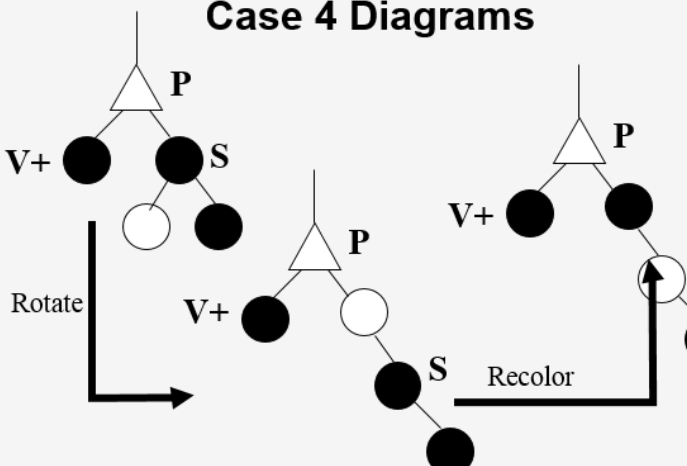


Red Node

- V's sibling, S, is Red
 - Rotate S around P and recolor S & P
- NOT a terminal case – One of the other cases will now apply
- All other cases apply when S is Black

Case 1 Diagram



<ul style="list-style-type: none"> ▪ V's sibling, S, is black and has <u>two black children</u>. <ul style="list-style-type: none"> – Recolor S to be Red – P absorbs V's extra blackness <ul style="list-style-type: none"> • If P was Red, make it black, we're done • If P was Black, it now has extra blackness and problem has been propagated up the tree 	<h3>Case 2 diagram</h3>  <p>Either extra black absorbed by P or P now has extra blackness</p>
<ul style="list-style-type: none"> ▪ S is black ▪ S's RIGHT child is RED (Left child either color) <ul style="list-style-type: none"> – Rotate S around P – Swap colors of S and P, and color S's Right child Black ▪ This is the terminal case – we're done 	<h3>Case 3 diagrams</h3> 
<ul style="list-style-type: none"> ▪ S is Black, S's right child is Black and S's left child is Red <ul style="list-style-type: none"> – Rotate S's left child around S – Swap color of S and S's left child – Now in case 3 	<h3>Case 4 Diagrams</h3> 

Red Black Visualization - <http://gauss.ececs.uc.edu/RedBlack/redblack.html>
<https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>

AVL Trees

An AVL tree is a special type of binary tree that is always "partially" balanced. The criteria that is used to determine the "level" of "balanced-ness" is the difference between the heights of sub-trees of every node in the tree. The "height" of tree is the "number of levels" in the tree.

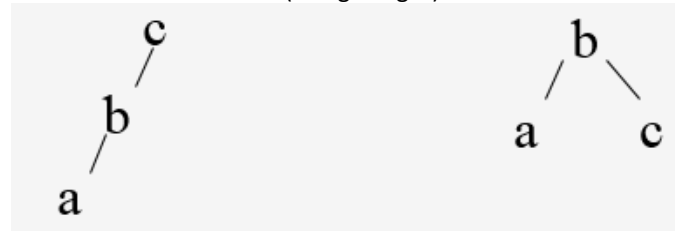
An AVL tree is a binary tree in which the difference between the height of the right and left sub-trees (of any node) is never more than one.

1. How do you think we could keep track of the height of the right and left sub-trees of every node?

2. If we find an imbalance, how can we correct it without adding any significant cost to the insert or delete?

Single Rotations

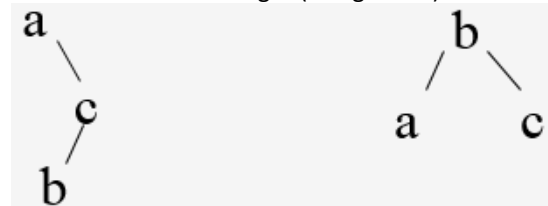
The imbalance is left-left (or right-right)



Perform single right rotation at "c" (R-rotation)
Similar idea for single left rotation (L-Rotation)

Double Rotations

The imbalance is left-right (or right-left)



Perform right rotation at "c" then left rotation at "a" (RL-rotation)
Similar idea for left rotation then right rotation (LR-Rotation)

CS430 Lecture 13 Activities

Opening Questions

Previously we discussed the order-statistic selection problem: given a collection of data, find the k th largest element. We saw that this is possible in $O(n)$ time for each k th element you are trying to find. The naïve method would just be to sort the data in $O(n \lg n)$ and then access each k th element in $O(1)$ time.

1. Which of these two methods would you use if you knew you would be asked to find multiple k th largest elements from a set of static data?

2. What if our collection of data is changing (dynamic), would either these approaches work efficiently for a collection of data that has inserts and deletes happening?

Augmenting Data Structures

For particular applications, it is useful to modify a standard data structure to support additional functionality. Sometimes the modification is as simple as by storing additional information in it, and by modifying the ordinary operations of the data structure to maintain the additional information.

Dynamic Order-Statistic Trees (Augmenting Balanced Trees)

1. What can we do with a binary search tree (and more efficiently with a balanced binary search tree)?

2. Consider the naïve method of finding the k th largest item is to sort the array and then access the k th item in $O(1)$ time. Can we do this with a (balanced) BST? What if we augment it (HINT: recall how counting sort worked)?

3. What is the recursive formula to find the size of a subtree at node x

4. Discuss in detail how would you keep the size at a node correct when you insert a new node, and possibly rotate to keep the tree balanced?

TREE-INSERT(T, z)	LEFT-ROTATE(T, x)
1 $y = \text{NIL}$	1 $y = x.\text{right}$ // set y
2 $x = T.\text{root}$	2 $x.\text{right} = y.\text{left}$ // turn y 's left subtree into x 's right subtree
3 while $x \neq \text{NIL}$	3 if $y.\text{left} \neq T.\text{nil}$
4 $y = x$	4 $y.\text{left}.p = x$
5 if $z.\text{key} < x.\text{key}$	5 $y.p = x.p$ // link x 's parent to y
6 $x = x.\text{left}$	6 if $x.p == T.\text{nil}$
7 else $x = x.\text{right}$	7 $T.\text{root} = y$
8 $z.p = y$	8 elseif $x == x.p.\text{left}$
9 if $y == \text{NIL}$	9 $x.p.\text{left} = y$
10 $T.\text{root} = z$ // tree T was empty	10 else $x.p.\text{right} = y$
11 elseif $z.\text{key} < y.\text{key}$	11 $y.\text{left} = x$ // put x on y 's left
12 $y.\text{left} = z$	12 $x.p = y$
13 else $y.\text{right} = z$	

5. Discuss in general how would you keep the size at a node correct when you delete a node, and possibly rotate to keep the tree balanced?

6. How can we use the augmented data at each node (size) in a balanced binary search tree to solve the kth largest item problem?

7. How can we use the augmented data at each node (size) in a balanced binary search tree so when given a pointer to a node in the tree we can determine its rank (the index position of the node of the tree data in sorted order)?

8. Why not just use this approach always (not just with dynamically changing data) instead of the $O(n)$ one?

9. Can we use this approach on a regular BST?

CS430 Lecture 14 Activities

Opening Questions

1. Briefly explain what two properties a problem must have so that a dynamic programming algorithm approach will work.
2. Previously we have learned that divide-and-conquer algorithms partition a problem into independent sub-problems, solve each sub-problem recursively, and then combine their solutions to solve the original problem. Briefly, how are dynamic programming algorithms similar and how are they different from divide-and-conquer algorithms?
3. Why does it matter how we parenthesize a chain of matrix multiplications? We get the right answer any way we associate the matrices for multiplication. i.e. If A, B and C are matrices of correct dimensions for multiplication
 $(A B) C = A (B C)$

Dynamic Programming

Dynamic Programming Steps

1. Define structure of optimal solution, including what are the largest sub-problems.
2. Recursively define optimal solution
3. Compute solution using table bottom up
4. Construct Optimal solution

Optimal Matrix Chain Multiplication (optimal parenthesization)

1. How many ways are there to parenthesize (two at a time multiplication) 4 matrices $A*B*C*D$?
2. Step 1: Generically define the structure of the optimal solution to the Matrix Chain Multiplication problem.
The optimal way to multiply n matrices A_1 through A_n is:
 3. Step 2: Recursively define the optimal solution. Assume $P(1,n)$ is the optimal cost answer. Make sure you include the base case.
4. Use proof by contradiction to show that Matrix Chain Multiplication problem has optimal substructure, i.e. the optimal answer to problem must contain optimal answers to sub-problems.

5. Step 3: Compute solution using a table bottom up for the Matrix Chain Multiplication problem. Use your answer to question 3 above. Note the overlapping sub-problems as you go. Step 4: Construct Optimal solution

A	B	C	D
2*4	4*6	6*3	3*7

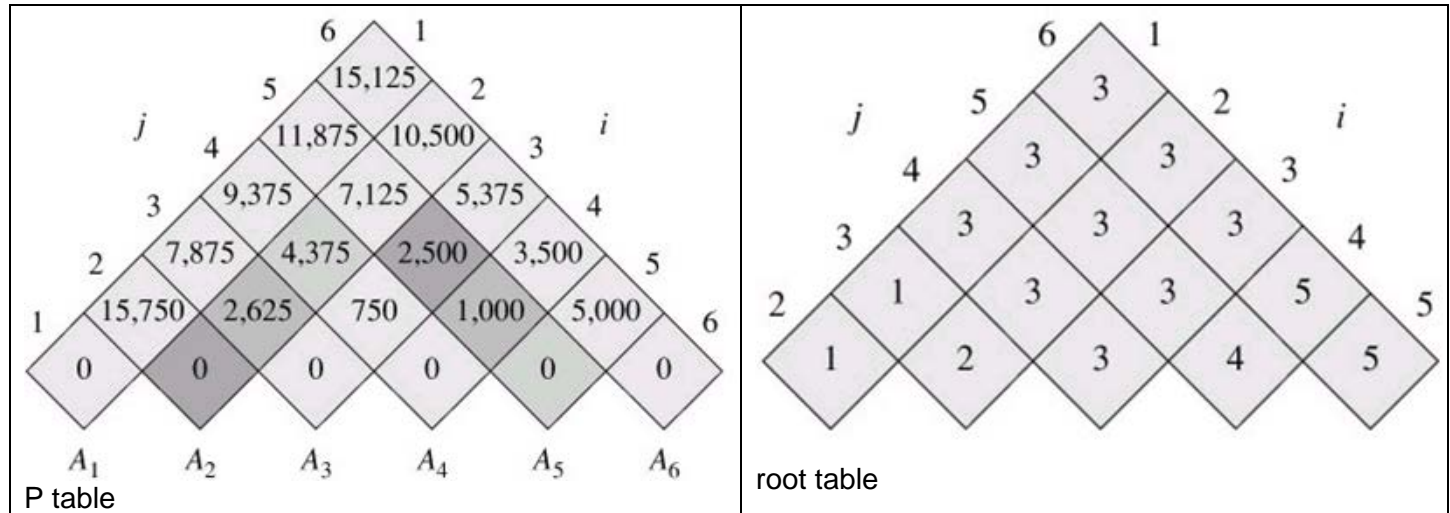
CS430 Lecture 15 Activities

Opening Questions

1. Why are optimal solutions to sub-problems stored in a table in dynamic programming solutions?

Constructing the answer for the Optimal Matrix Chain Multiplication (optimal parenthesization) from the dynamic programming table. See the solution to the example problem

A1 A2 A3 A4 A5 A6
30x35 35x15 15x5 5x10 10x20 20x25



2. Write the optimal parenthesization.

3. Write pseudocode to use the root table to print the optimal parenthesization. Then write pseudocode to use the root table to actually perform the multiplications in the optimal parenthesization.

Longest Common Subsequence

Example: X[1.....m] Y[1.....n]

X : ABCBDAB Y : BDCABA

Length of LCS = 4 BCBA or BCAB

1. The brute force approach would be to find all subsequences of one input, see if each exist in other input. How many are there?

2. Step 1: Generically define the structure of the optimal solution to the Longest Common Subsequence problem.

The longest common subsequence of sequence $X[1.....m]$ and sequence $Y[1.....n]$ is:

3. Step 2: Recursively define the optimal solution. Assume $C(i,j)$ is the optimal answer for up to position i in X and position j in Y . Make sure you include the base case.

4. Use proof by contradiction to show that Longest Common Subsequence problem has optimal substructure, i.e. the optimal answer to problem must contain optimal answers to sub-problems.

5. Step 3: Compute solution using a table bottom up for the Longest Common Subsequence problem. Use your answer to question 3 above. Note the overlapping sub-problems as you go. .Step 4: Construct Optimal solution

X = ABCBDAB

Y = BDCABA

<https://www.cs.usfca.edu/~galles/visualization/DPLCS.html>

CS430 Lecture 16 Activities

Opening Questions

1. In the Optimal Binary Search Tree problem we are not just trying to balance the tree, instead we are trying to minimize what?
2. What is different about the 0-1 knapsack problem as compared to the other problems we solved with dynamic programming?

Optimal Binary Search Tree

The Optimal Binary Search Tree problem is a special case of a BST where the data is static (no inserts or deletes) and we know the probability of each key in the data being searched for. We want to minimize total expected search time for the BST. Recall expectation

$$E(X) = \sum_{s \in S} X(s)p(s)$$

1. Apply the expectation formula to the Optimal Binary Search Tree problem with n keys and $C(i)$ is how deep item " i " is. $P(i)$ is probability of search for item " i "
2. The brute force approach would be to find all possible BSTs, find the expected search time of each and pick the minimum one. How many BSTs are there?
3. Step 1: Generically define the structure of the optimal solution to the Optimal Binary Search Tree problem. The optimal binary search tree with n keys n keys and $C(i)$ is how deep item " i " is and $P(i)$ is probability of search for item " i " is:
4. Step 2: Recursively define the optimal solution. Assume $A(i,j)$ is the optimal answer for keys i to j . Make sure you include the base case.

5. Use proof by contradiction to show that Optimal Binary Search Tree problem has optimal substructure, i.e. the optimal answer to problem must contain optimal answers to sub-problems.

6. Step 3: Compute solution using a table bottom up for the Optimal Binary Search Tree problem. Use your answer to question 4 above. Note the overlapping sub-problems as you go. .Step 4: Construct Optimal solution

A	B	C	D	E	F
.2	.16	.08	.22	.21	.13

Optimal Binary Search Tree <http://www.cse.yorku.ca/~aaw/Gubarenko/BSTAnimation.html>

CS430 Lecture 17 Activities

When can you use Dynamic Programming?

DP computes recurrences efficiently by storing partial results. Thus DP can only be efficient when there are not too many partial results to compute. There are $n!$ permutations of an n -element set – we cannot use DP to store the best solution to each sub-permutation. There are 2^n subsets of an n -element set, we cannot use dynamic programming to store the best solution for each subset.

Dynamic Programming works best on objects which are linearly ordered and cannot be rearranged, so the number of partial results is not exponential. Characters in a string, matrices in a chain, the left-to-right order of the leaves in a BST. One commonality to all the dynamic programming solutions we explored is that all the problems had some sort of ordering restriction. Here is an example that does not. Because of the constraint on the total weight limit, it is not an exponential enumerate all the subsets of an n -element set.

0-1 Knapsack Problem

Given N items and a total weight limit W , v_i is the value and w_i is the weight of item i , maximize the total value of items taken.

The dynamic programming algorithm computes entries for a matrix $C[0..N, 0..W]$

$C[i, j]$ = the optimal value of the items put into the knapsack from among items $\{1, 2, \dots, i\}$ with total weight $\leq j$

with $C[0, ?] = C[?, 0] = 0$

1. When you think about calculating $C[i, j]$ there are two options. The i th item is in that optimal answer or is not. Write the recurrence relation.

2. Write pseudocode to fill in the $C[i, j]$ matrix, use your answer from #7.

Opening Questions - Greedy Algorithms

3. Briefly explain what two properties a problem must have so that a greedy algorithm approach will work.
4. A good cashier gives change using a greedy algorithm to minimize the number of coins they give back. Explain this greedy algorithm.
5. For what types of optimization problems does optimal substructure fail?

Activity Selector Problem

Given a set S of n activities each with start S_i , Finish F_i , find the maximum set of Compatible Activities (non-overlapping).
(or could be jobs scheduled with fixed start and end times instead of meetings)

6. The brute force approach would be to find all possible subsets of n activities, eliminate the ones with non-compatible meetings, and find the largest subset. How many subsets are there?
7. Prove the Activity Selector Problem has optimal substructure (using similar proof by contradiction approach that we used for dynamic programming: assume you have an optimal answer, remove something to get to the largest sub-problem, show that the sub-problem must also be solved optimally).

CS430 Lecture 18 Activities

Proving a Greedy Choice Property

To prove a Greedy Choice Property for a problem (that local optimal choice leads to global optimal solution) use the following “cut and paste” proof.

1. Assume you have an optimal answer that does not contain the greedy choice you are trying to prove.
2. Show that you can “cut” something out of that optimal answer and “paste” in the greedy choice you are trying to prove. Therefore either the assumed optimal answer already contained the greedy choice, or you can make the assumed optimal contain the greedy choice.
3. Therefore there is always an optimal answer that contains the greedy choice (can be continued like an inductive proof).

1. Try various “common sense” greedy approaches that divide the problem into a sub-problem(s) and try to come up with counter-examples or prove the greedy choice is correct.

Does every Optimization Problem exhibit Optimal Substructure?

Consider the following two problems in which we are given a directed graph $G = (V, E)$ and vertices $u, v \in V$.

1. Un-weighted shortest path: Find a path from u to v consisting of the fewest edges.
2. Un-weighted longest simple path: Find a simple path from u to v consisting of the most edges.

2. Try to prove Optimal Substructure for the above two problems.

3. Why does optimal substructure fail for some optimization problems?

CS430 Lecture 19 Activities

Fractional Knapsack Problem

n items w_i weight v_i values Total weight limit = W
Maximize value Total weight $\leq W$ Fractional amounts of items are allowed

1. Prove that the Fractional Knapsack Problem has optimal substructure.
2. Try various “common sense” greedy approaches that divide the problem into a sub-problem(s) and try to come up with counter-examples or prove the greedy choice is correct using the “cut and paste” proof.

Huffman Codes Problem

Data Encoding Background

- Data is a sequence of characters
- Fixed Length – Each character is represented by a unique binary string. Easy to encode, concatenate the codes together. Easy to decode, break off 3-bit codewords and decode each one.
- Variable Length – Give frequent characters shorter codewords, infrequent characters get long codewords. However, how do we decode if the length of the codewords are variable?
- Prefix Codes - No codeword is a prefix of another codeword. Easy to encode, concatenate the codes together. Easy to decode, since no codeword is a prefix of another, strip off one bit at a time and match to unique prefix code

Huffman Codes

- Are a Data Compression technique using a greedy algorithm to construct an optimal variable length prefix code
- Use frequency of occurrence of characters to build an optimal way to represent each character as a binary string
- Use a Binary tree method – 0 means go to left child, 1 means go to right child (not a binary search tree).
- Cost of Tree in bits $B(T) = \sum_{\text{for all } c \in C} \text{freq}(c) * \text{depth}(c)$

Example

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 16.3 A character-coding problem. A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated. If each character is assigned a 3-bit codeword, the file can be encoded in 300,000 bits. Using the variable-length code shown, the file can be encoded in 224,000 bits.

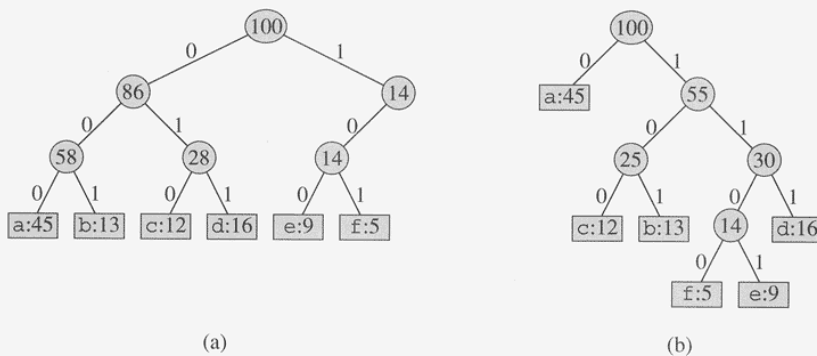
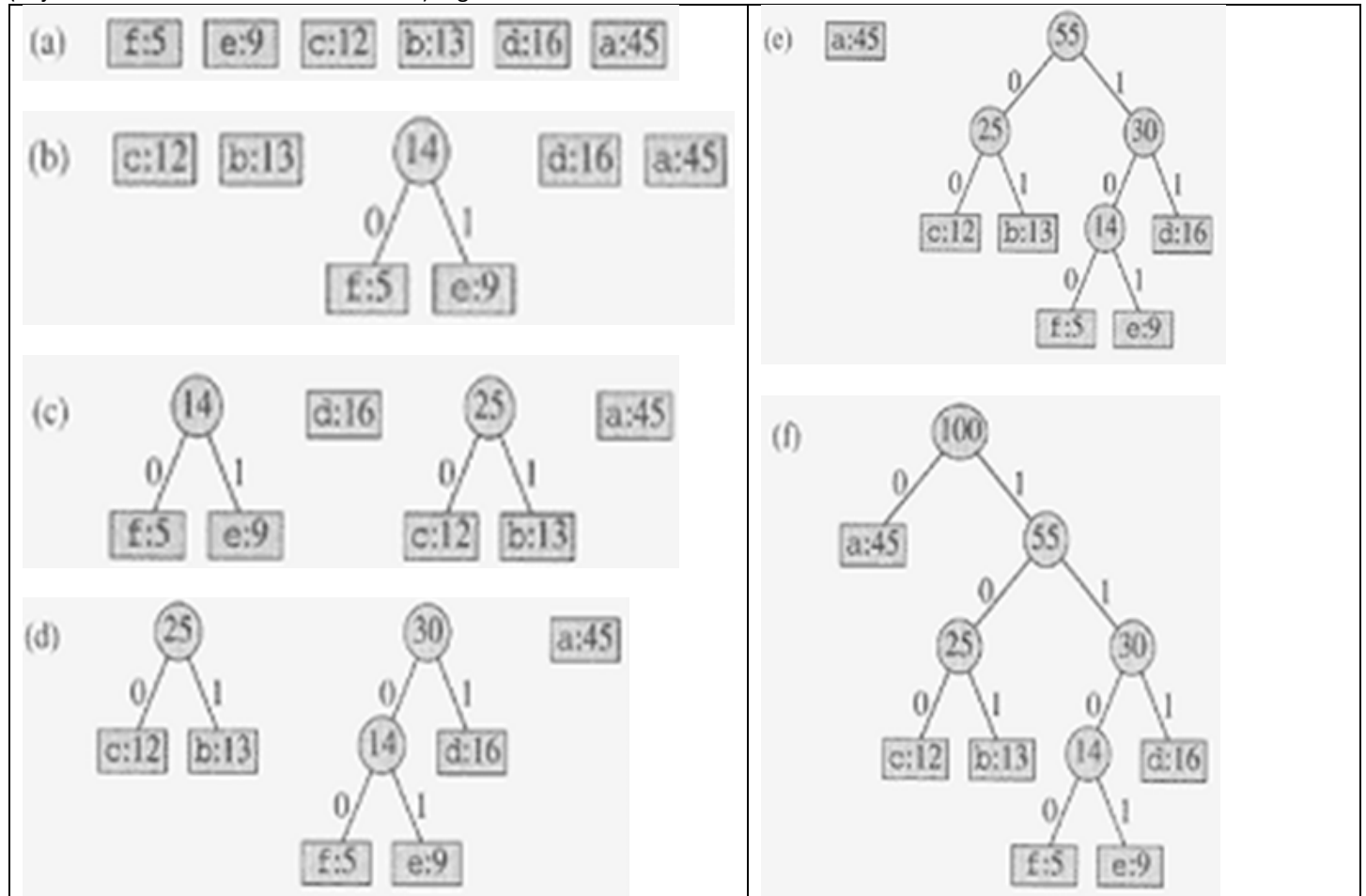


Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. (a) The tree corresponding to the fixed-length code $a = 000, \dots, f = 101$. (b) The tree corresponding to the optimal prefix code $a = 0, b = 101, \dots, f = 1100$.

3. Prove that the Huffman Codes Problem has optimal substructure.

The greedy approach that works is: Build the tree bottom up by using a minimum priority queue to merge 2 least frequent objects (objects are leaf nodes or other subtrees) together into new subtree.



Huffman <http://www.cs.auckland.ac.nz/software/AlgAnim/huffman.html>

4. Proof this greedy approach leads to an optimal Huffman Code tree: Build the tree bottom up by using a minimum priority queue to merge 2 least frequent objects (objects are leaf nodes or other subtrees) together into new subtree.

CS430 Lecture 20 Activities

Opening Questions

1. How do you think the allocated size growth of a dynamic array like Java's ArrayList is implemented? How much bigger does it grow when needed? What is the runtime for a sequence of n insertions starting from a default size of 10 considering the worst individual insert?

Amortized (to pay off gradually) Analysis

So far, we have analyzed best and worst case running times for an operation without considering its context. With amortized analysis, we study a sequence of operations rather than individual operations. An amortized analysis is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive.

Aggregate Method of Amortized Analysis

1. Can we do a better analysis by amortizing the cost over all inserts? Starting with a table size one and doubling the size when necessary make a table showing the first 10 inserts and determine a formula for $\text{cost}(i)$ for the cost of the i th insert. Then aggregate *add up" all the costs and divide by n (aggregate analysis).

Accounting Method of Amortized Analysis

Figure out a specific amortized cost to be allocated to each operation to ensure you have enough “balance” to handle the bad operations.

Charge i th operation a fictitious amortized cost \hat{c}_i , where \$1 pays for 1 unit of work (i.e., time).

- This fee is consumed to perform the operation.
- Any amount not immediately consumed is stored in the bank for use by subsequent operations.
- The bank balance must not go negative! We must ensure that for all n

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

Thus, the total amortized costs provide an upper bound on the total true costs.

2. For the previous ArrayList example determine the amortized cost \hat{c}_i necessary.

3. Consider, as a second example, a binary counter that is being implemented in hardware.

Assume that the machine on which it is being run can flip a bit as its basic operation. We now want to analyze the cost of counting up from 0 to n (using k bits).

What is the naive worst-case analysis for how many bits we need to flip?

Decimal	Binary
1	000001
2	000010
3	000011
4	000100
5	000101
\vdots	\vdots
n	100110

4. Use the aggregate method to perform a more careful analysis for n increments of a binary counter.

5. Use the accounting method to perform a more careful analysis for n increments of a binary counter.

CS430 Lecture 21 Activities

Opening Questions

1. Review the operations on a min (or max) binary heap and their runtimes
2. What if we needed a function to union two min binary heaps? How would you do it and what is the run time?
3. Why did we use an array for a binary heap? Does a heap need to be binary?

Mergeable (Min)Heaps

Consider the following operations on heaps. Our goal is to support all of these operations in no worse than $\Theta(\log n)$ and not be constrained to use an array or a binary tree.

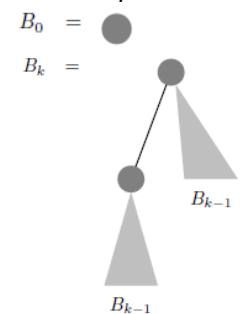
- Make-Heap: creates a new empty heap
- Insert: inserts a new element into a heap
- Minimum: returns the minimum element in a heap
- Extract-Min: returns the minimum element in a heap and removes it from the heap
- Union: creates a new heap consisting of the elements of two existing heaps

Two types of mergeable heaps are Binomial heaps and Fibonacci heaps. They also support these operations.

- Decrease-Key: changes the value of some element in a heap to a smaller value
- Delete: removes an element from a heap

Binomial Heaps (utilizing binomial trees)

A binomial tree is an ordered tree defined recursively:



1. How many nodes does B_k have, what is its height, and how many children does its root have?

2. Why are they called binomial trees?

A binomial heap is a collection (linked list) of binomial trees in which each binomial tree is heap-ordered: each node is greater than or equal to its parent. Also, in a binomial heap at most one instance of B_i may occur for any i .

3. How many binomial trees are needed at most in the linked list of roots to make a binomial heap of n nodes?

4. See <https://www.cs.usfca.edu/~galles/visualization/BinomialQueue.html> to help describe how each operation is done, and its run time:

Make-Heap:

Minimum:

Union:

Insert:

Extract-Min:

Decrease-Key:

Delete:

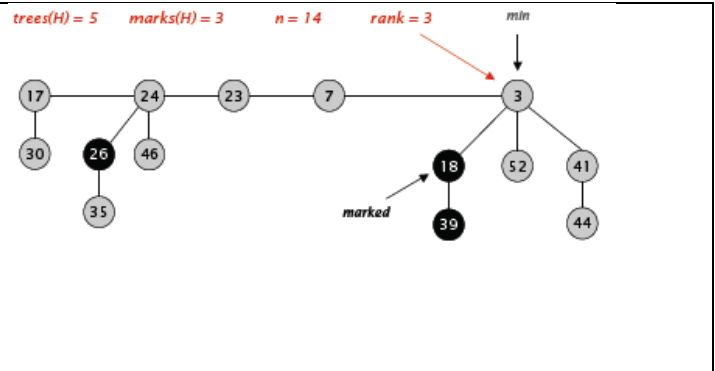
CS430 Lecture 22 Activities

Fibonacci Heaps

Fibonacci heaps which support heap operations that do not delete elements in constant amortized time. From a theoretical standpoint, Fibonacci heaps are especially desirable when the number of EXTRACT-MIN and DELETE operations is small relative to the number of other operations performed. This situation arises in many graph algorithms.

In essence, a Fibonacci heap is a “lazy” binomial heap in which the necessary housekeeping is delayed until the last possible moment: deletion.

- Set of Heap ordered trees (each parent smaller than children)
- Maintain pointer to minimum element (find-min takes $O(1)$ time)
- Set of marked nodes (if one of its children has been removed)
- n – number of nodes in the heap
- $\text{rank}(x)$ – number of children of node x
- $\text{rank}(H)$ – max rank of any node in heap H
- $\text{trees}(H)$ – number of trees in heap H
- $\text{marks}(H)$ – number of marked nodes in H



1. See <https://www.cs.usfca.edu/~galles/JavascriptVisual/FibonacciHeap.html> and <https://www.cs.princeton.edu/~wayne/cs423/fibonacci/FibonacciHeapAnimation.html> to help describe how each operation is done, and a rough estimate on its run time:
Make-Heap

Insert:

Minimum:

Union:

Extract-Min:

Decrease-Key:

Delete:

Operation	Binary heap	Binomial heap	Fibonacci heap	Note that the times indicated for the Fibonacci heap are amortized times while the times for binary and binomial heaps are worst-case per-operation times.
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	
INSERT	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$	
MINIMUM	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	
EXTRACT-MIN	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	
UNION	$\Theta(n)$	$O(\log n)$	$\Theta(1)$	
DECREASE-KEY	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$	
DELETE	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	

CS430 Lecture 23 Activities

Disjoint-set Data Structure

It is useful in many applications to have a structure that handles groups of disjoint sets. In particular, we support the following three operations:

- **Make-Set(x):** Creates a new set consisting of a single element x . Since the sets are disjoint, we assume that this element is not already contained in any set.
- **Union(x,y):** Given elements in two different sets, forms the union of two existing sets into one new set.
- **Find-Set(x):** Returns a pointer to the representative of the set containing element x .

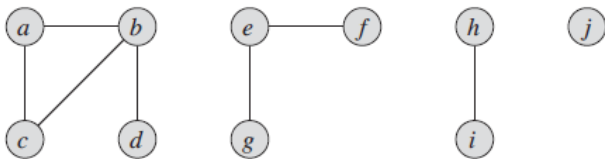
To identify a set, we return a pointer to any element in the set. The only constraint we make on the element chosen is that if the set does not change between calls to Find-Set, we must return the same representative.

We analyze the algorithms implementing these operations in terms of n , the number of Make-Set operations (all Make-Sets are usually assumed to run first), and m , the total number of Make-Set, Union, and Find-Set operations ($n \leq m$)

1. Give then above, how many possible Union operations might there be?

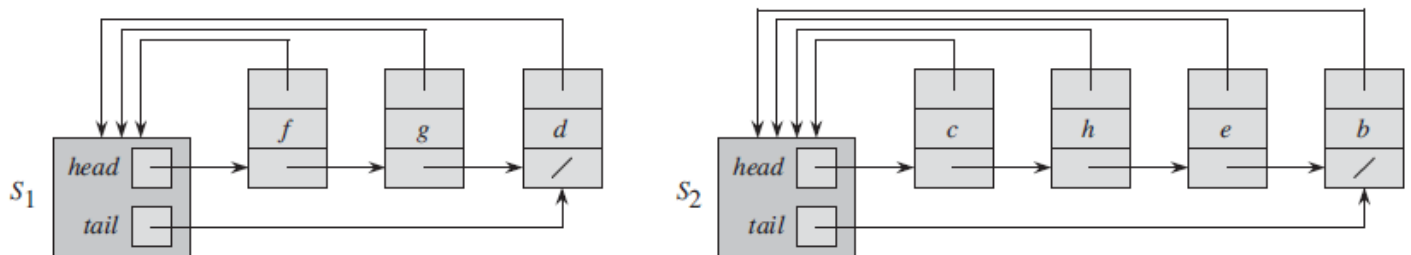
Disjoint-set Application

A graph data structure is a set of vertices and edges between those vertices, and supports problems where there can be relationships between any two items (vertices)



1. It is easy to see the disjoint sets (connected components) of the graph. Given a graph $G=(V, E)$ write an algorithm using Make-Set, Union, Find-Set to find the connected components of any undirected graph

Linked-list representation



One simple approach is to represent a set as an unordered linked list. Each element contains two pointers, one to the next element (as in a simple linked list) and one to the head of the list. The head serves as the set representative.

2. Describe the algorithms for Make-Set(x) and Find-Set(x) including runtime. For Find-Set(x) assume you already have a reference to x .

3. How do you think $\text{Union}(x,y)$ is implemented?

4. For n total elements, what is the maximum number of Make-Set and Union operations that would need to be called in the worst case to get all the elements in one set? What is the maximum amortized runtime of each union?

5. For n total elements, what is the minimum number of Make-Set and Union operations that would need to be called in the best case to get all the elements in one set? What is the lower bound, worst case amortized runtime of each union?

CS430 Lecture 24 Activities

Opening Questions

1. Give an example NOT discussed in the video lecture of a problem that can be represented by a graph.
2. If there is a path in a graph from a vertex back to itself that is called a _____
3. Which representation of a graph, adjacency-list and adjacency-matrix, usually uses more memory and why?

Graphs

1. Draw the graph: A directed graph $G=(V,E)$, where $V=\{1,2,3,4,5,6\}$ and $E=\{(1,2),(2,2),(2,4),(2,5),(4,1),(4,5),(5,4),(6,3)\}$. What is edge $(2,2)$ called?

2. Draw the graph: An undirected graph $G=(V, E)$, where $V=\{1,2,3,4,5,6\}$ and $E=\{\{1,2\},\{1,5\},\{2,5\},\{3,6\}\}$. What is vertex 4 called? What is different about how the edge set E is denoted for an undirected graph? Are self-loops allowed in an undirected graph?

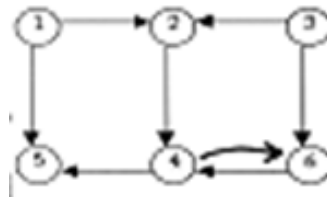
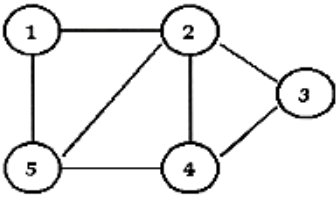
3. Define these terms:

- vertex v is adjacent to vertex u in an undirected graph
- vertex v is adjacent to vertex u in a directed graph
- the degree of a vertex in an undirected graph
- the degree of a vertex in a directed graph
- a path in an undirected graph
- a path in a directed graph
- the length of a path

- v is reachable from u
- a simple path
- a cycle in an undirected graph. What about a simple cycle?
- a cycle in a directed graph. What about a simple cycle?
- Acyclic graph
- Connected undirected graph
- Connected directed graph
- Bipartite Graph

Graph Implementations

4. What is the adjacency list implementation of these two graphs?



5. What is the adjacency matrix implementation of the above two graphs?

6. How do the two implementations handle a weighted graph?

7. Two different representations of the graph data structure are discussed in the book, adjacency-list and adjacency-matrix. Please briefly discuss the runtime (in terms of $|V|$ and $|E|$) of these graph operations/algorithms using each implementation. Assume vertices are labeled as integers.

- What is the worst-case big-O runtime for checking to see if an edge from vertex u to vertex v exists?
- How long does it take to compute the out-degree of every vertex of a directed graph?
- How long does it take to compute the in-degree of every vertex of a directed graph?

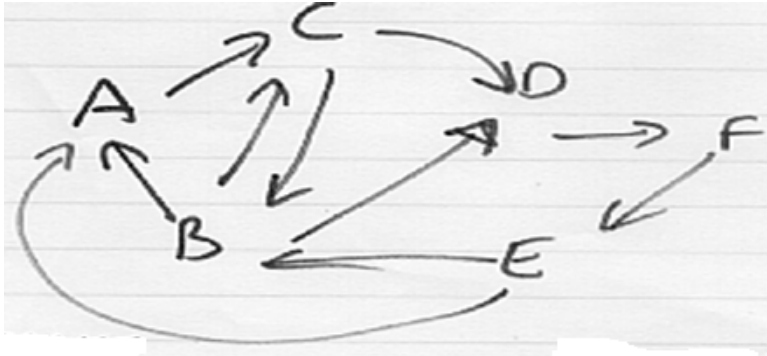
Graph Traversals

A way to search / visit all the vertices in a graph. There is not a unique answer usually.

- Undirected graph - if connected, all vertices will be visited
- Directed graph - Must be strongly connected to be able to visit all vertices

Breadth first - visit vertices one edge from a given (or random) source, two edges from source, etc. Uses a queue and some way to mark a vertex as visited (white initially, gray when first visited and put in queue, black when out of queue), label a vertex with how far from the source, and label a vertex with how its predecessor vertex was during the traversal.

8. Perform a breadth first search on this graph.



CS430 Lecture 25 Activities

Opening Questions

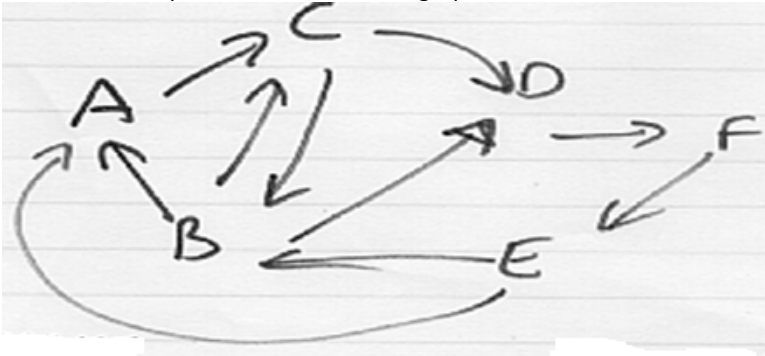
1. What is the runtime for breadth first search (if you restart the search from a new source if everything was not visited from the first source)?
2. Does a breadth first search always reach all vertices?
3. How can you use a breadth first search to find the shortest path (minimum number of edges) from a given source vertex to all other vertices?
4. If you look at the predecessor edges which were used to connect to an unvisited vertex, what do these predecessor edges form? Is it unique for a graph?

Depth First Search

https://www.reddit.com/r/dataisbeautiful/comments/7b7aa0/visualizing_the_depthfirst_search_recursive/?st=j9OUDR00&sh=5b671c59

As we visit a vertex we try to move to a new adjacent vertex that hasn't yet been visited, until nowhere else to go, then backtrack. Uses a stack and some way to mark a vertex as visited (white initially, gray when first visited and put in stack, black when out of stack), label a vertex with a counter for first time seen, and another counter for last time seen (we will see why later), and label a vertex with how its predecessor vertex was during the traversal.

1. Perform a depth first search on this graph.



```

DFS(G)
1  for each vertex  $u \in V[G]$ 
2      do  $color[u] \leftarrow WHITE$ 
3       $\pi[u] \leftarrow NIL$ 
4   $time \leftarrow 0$ 
5  for each vertex  $u \in V[G]$ 
6      do if  $color[u] = WHITE$ 
7          then DFS-VISIT( $u$ )

DFS-VISIT( $u$ )
1   $color[u] \leftarrow GRAY$   $\triangleright$  White vertex  $u$  has just been discovered.
2   $time \leftarrow time + 1$ 
3   $d[u] \leftarrow time$ 
4  for each  $v \in Adj[u]$   $\triangleright$  Explore edge  $(u, v)$ .
5      do if  $color[v] = WHITE$ 
6          then  $\pi[v] \leftarrow u$ 
7              DFS-VISIT( $v$ )
8   $color[u] \leftarrow BLACK$   $\triangleright$  Blacken  $u$ ; it is finished.
9   $f[u] \leftarrow time \leftarrow time + 1$ 

```

2. What is the runtime for depth first search (if you restart the search from a new source if everything was not visited from the first source)?

Another interesting property of depth-first search is that the search can be used to classify the edges of graph G based on how they are traversed

- **Tree edges** are edges in the depth-first forest G_π . Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v) .
- **Back edges** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Self-loops, which may occur in directed graphs, are considered to be back edges.
- **Forward edges** are those non-tree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
- **Cross edges** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

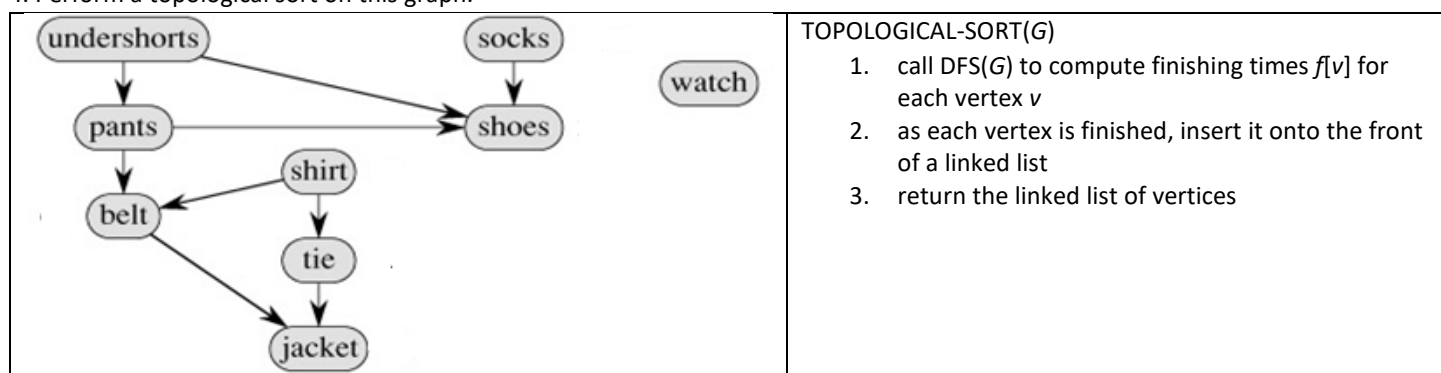
3. If a graph has no back edges when completing a depth first search what does that tell us about the graph?

demo BFS/DFS <http://www3.cs.stonybrook.edu/~skiena/combinatorica/animations/search.html>

Topological sort (a DFS application)

- A topological sort of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v) , then u appears before v in the ordering. (If the graph is not acyclic, then no linear ordering is possible.)
- A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right. Topological sorting is thus different from the usual kind of "sorting" studied earlier.
- Directed acyclic graphs are used in many applications to indicate precedence among events
- A depth-first search can be used to perform a topological sort of a directed acyclic graph, or a "dag" as it is sometimes called.

4. Perform a topological sort on this graph.



5. Why does the topological sort work? What is its runtime?

The Parenthesis Theorem

The parenthesis theorem tells us that, for two vertices $u, v \in V$, it cannot be the case that $d[u] < d[v] < f[u] < f[v]$; that is, the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are either disjoint or nested. This is a simple consequence of the depth-first nature of DFS. If the algorithm discovers u and then discovers v , it cannot later back out of u without first backing out of v .

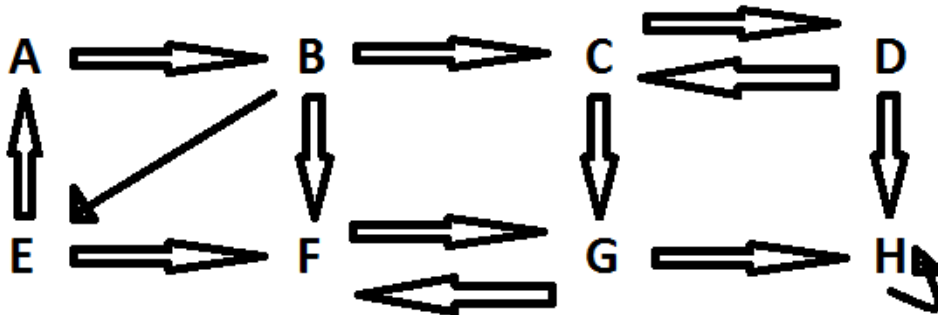
Strongly Connected Components (a DFS application)

A graph is said to be strongly connected if every vertex is reachable from every other vertex. The strongly connected components of an arbitrary directed graph form a partition into subgraphs that are themselves strongly connected. It is possible to test the strong connectivity of a graph, or to find its strongly connected components, in linear time.

STRONGLY-CONNECTED-COMPONENTS (G)

1. call DFS (G) to compute finishing times $f[u]$ for each vertex u
2. compute G^T (the transpose of the graph)
3. call DFS (G^T), but in the main loop of DFS, consider the vertices in order of decreasing $f[u]$ (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

6. Find the strongly connected components.



6. Discuss: G and G^T will have the same strongly connected components.

7. Discuss: The component with the latest finish time vertex will have no edges in the transpose to any other component.

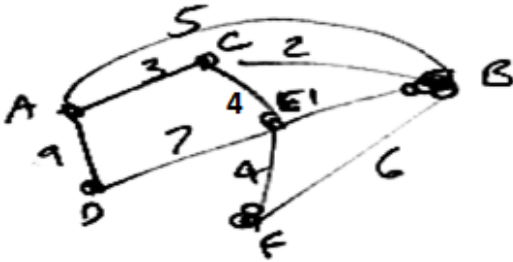
CS430 Lecture 26 Activities

Opening Questions

1. What is the difference between a tree and a graph?
2. Give a recursive definition for a tree.
3. In a weighted undirected graph, what is the difference between a minimum spanning tree and a shortest path in a graph?
4. Since shortest paths contain shortest sub-paths (optimal substructure), name an algorithmic approach that we might try to find a shortest path in a graph.

Minimum Spanning Trees

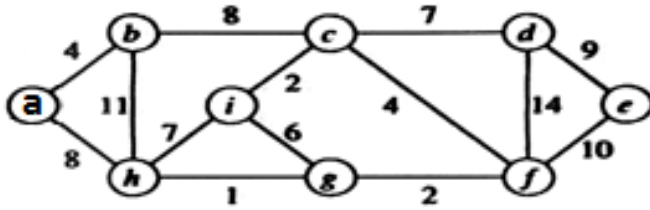
1. Give a definition of a Minimum Spanning Tree, and find a MST of the below graph.



2. Prove a Minimum Spanning Tree has optimal substructure.

3. What are some possible greedy approaches to find a Minimum Spanning Tree? Prove correct or show counterexample

4. Demonstrate your MST algorithm on the following graph and write pseudocode.



How to find the shortest route between two points on a map.

Input:

- Directed graph $G = (V, E)$
- Weight function $w : E \rightarrow \mathbf{R}$

Weight of path $p = \langle v_0, v_1, \dots, v_k \rangle$

$$= \sum_{i=1}^k w(v_{i-1}, v_i)$$

= sum of edge weights on path p .

Shortest-path weight u to v :

$$\delta(u, v) = \begin{cases} \min \{w(p) : u \xrightarrow{p} v\} & \text{if there exists a path } u \rightsquigarrow v, \\ \infty & \text{otherwise.} \end{cases}$$

Shortest path u to v is any path p such that $w(p) = \delta(u, v)$.

Variants

- Single-source: Find shortest paths from a given source vertex $s \in V$ to every vertex $v \in V$.
- Single-destination: Find shortest paths to a given destination vertex.
- Single-pair: Find shortest path from u to v . No way known that's better in worst case than solving single-source.
- All-pairs: Find shortest path from u to v for all $u, v \in V$. We'll see algorithms for all-pairs in the next chapter.

Negative-weight edges - OK, as long as no negative-weight cycles are reachable from the source.

- If we have a negative-weight cycle, just keep going around it, and get $w(s, v) = -\infty$ for all v on the cycle.
- But OK if the negative-weight cycle is not reachable from the source.
- Some algorithms work only if there are no negative-weight edges in the graph.

1. What would the brute force approach be to solve the shortest path problem, and what is its run time?

2. Prove optimal substructure for the shortest path problem

Output of single-source shortest-path algorithm

For each vertex $v \in V$:

- $d[v] = \delta(s, v)$, Initially, $d[v] = \infty$, Reduces as algorithms progress. But always maintain $d[v] \geq \delta(s, v)$. Call $d[v]$ a *shortest-path estimate*.
- $\pi[v]$ = predecessor of v on a shortest path from s , If no predecessor, $\pi[v] = \text{NIL}$, π induces a tree—*shortest-path tree*

Initialization - All the shortest-paths algorithms start with INIT-SINGLE-SOURCE.

INIT-SINGLE-SOURCE(V, s)

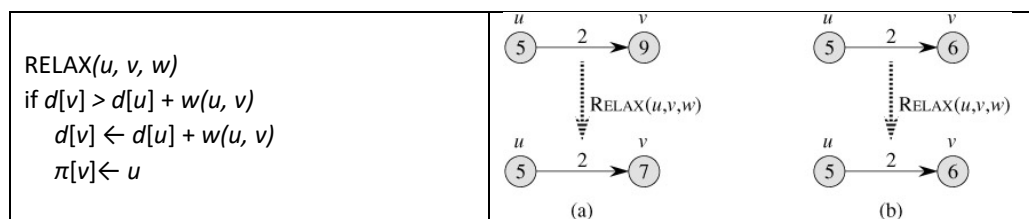
for each $v \in V$

$d[v] \leftarrow \infty$

$\pi[v] \leftarrow \text{NIL}$

$d[s] \leftarrow 0$

Relaxing an edge (u, v) - Can we improve the shortest-path estimate (best seen so far) from the source s to v by going through u and taking edge (u, v) ?



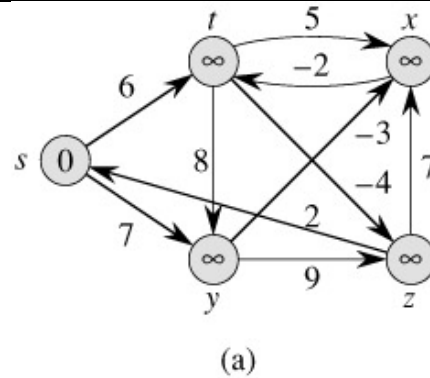
The algorithms differ in the order and how many times they relax each edge.

Shortest Path Algorithm - Bellman-Ford

The most straightforward of the “relax an edge” algorithms. Relaxes the edges in a fixed order (any fixed order) $|V|-1$ times. Not a greedy algorithm.

- Allows negative-weight edges.
- Computes $d[v]$ and $\pi[v]$ for all $v \in V$.
- Returns TRUE if no negative-weight cycles reachable from s , FALSE otherwise.

```
BELLMAN-FORD( $V, E, w, s$ )
INIT-SINGLE-SOURCE( $V, s$ )
for  $i \leftarrow 1$  to  $|V|-1$ 
  for each edge  $(u, v) \in E$ 
    RELAX( $u, v, w$ )
for each edge  $(u, v) \in E$  // all edges, in any order, same
order each time
  if  $d[v] > d[u] + w(u, v)$ 
    then return FALSE
return TRUE
```



3. Execute Bellman-Ford on the above graph from source s for this edge order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. Update the $d[v]$ and $\pi[v]$ values for each iteration.

4. What is the runtime of Bellman-Ford?

5. Prove Bellman-Ford is correct.

Values you get on each pass and how quickly it converges depends on order of relaxation. But guaranteed to converge after $|V|-1$ passes, assuming no negative-weight cycles.

CS430 Lecture 28 Activities

Opening Questions

1. We saw the Bellman-Ford algorithm found the shortest path from a source to all other vertices by “brute force” relaxing every edge in the graph in a fixed order $|V|-1$ times. Why did it need to do this $|V|-1$ times? And with this in mind could we improve on the Bellman-Ford for certain graphs?

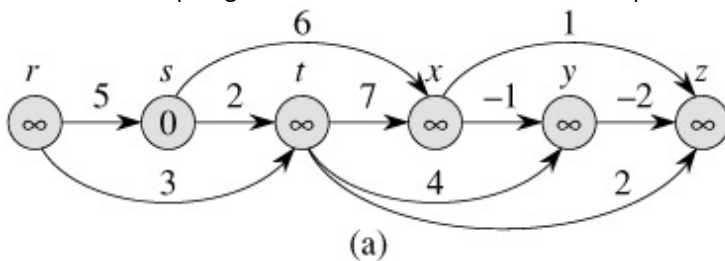
DAG Shortest Path Algorithm

By relaxing the edges of a weighted DAG (directed acyclic graph) $G = (V, E)$ in topological sort order of its vertices, we can compute shortest paths from a single source. Shortest paths are always well defined in a DAG, since even if there are negative-weight edges, no negative-weight cycles can exist.

DAG-SHORTEST-PATHS(G, w, s)

- 1 topologically sort the vertices of G
- 2 INITIALIZE-SINGLE-SOURCE(G, s)
- 3 **for** each vertex u , taken in topologically sorted order
- 4 **do for** each vertex $v \in \text{Adj}[u]$
- 5 **do** RELAX(u, v, w)

1. Here is the topological sort on a DAG. Find the shortest path from s to every other vertex.



2. What is the runtime for DAG Shortest Path?

3. Discuss why DAG Shortest Path is correct.

4. If we restrict the graph to having no negative edges, given a source s , what is the shortest path from s to one of its adjacent vertexes?

Dijkstra's Shortest Path Algorithm

- No negative-weight *edges*.
- Essentially a weighted version of breadth-first search.
 - Instead of a FIFO queue, uses a priority queue.
 - Keys are shortest-path weight estimates ($d[v]$).
- Have two sets of vertices:
 - S = vertices whose final shortest-path weights are determined,
 - Q = priority queue = $V - S$.
- Dijkstra's algorithm can be viewed as greedy, since it always chooses the "lightest" ("closest") vertex in $V - S$ to add to S .

DIJKSTRA(V, E, w, s)

INIT-SINGLE-SOURCE(V, s)

$S \leftarrow$ empty set

$Q \leftarrow V$ // i.e., insert all vertices into Q by " d " values

while Q not empty

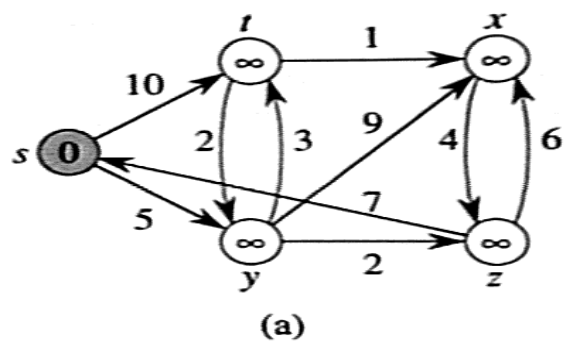
$u \leftarrow$ EXTRACT-MIN(Q)

$S \leftarrow S \cup \{u\}$

for each vertex $v \in \text{Adj}[u]$

 RELAX(u, v, w) // possibly updates a short path estimate " d " value and moves the vertex forward in the queue

5. Here is a graph with no negative edges. Find the shortest path from s to every other vertex using Dijkstra's algorithm.



6. What is the runtime for Dijkstra's algorithm?

7. Prove the greedy choice in Dijkstra's algorithm (pick the vertex with the smallest shortest path estimate, not including the vertices we are done with) leads to an optimal solution.

<https://www.youtube.com/watch?v=wtdtkJgcYUM>

<https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html>

CS430 Lecture 29 Activities

All-Pairs Shortest Paths Problem

- Given a directed graph $G = (V, E)$, weight function $w : E \rightarrow \mathbb{R}$, $|V| = n$.
- Goal: create an $n \times n$ matrix of shortest-path distances from every vertex to every other vertex $\delta(u, v)$.
- Could run BELLMAN-FORD once from each vertex:
 - $O(V^2E)$ which is $O(V^4)$ if the graph is *dense* ($E \sim V^2$).
- If no negative-weight edges, could run Dijkstra's algorithm once from each vertex:
 - $O(V E \lg V)$ with binary heap— $O(V^3 \lg V)$ if dense
- We'll see how to do in $O(V^3)$ in all cases with dynamic programming (we have already shown the shortest path problem has optimal substructure).

The formal problem statement:

- Assume that G is given as adjacency matrix of weights: $W = (w_{ij})$, with vertices numbered 1 to n .

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ \text{weight of } (i, j) & \text{if } i \neq j, (i, j) \in E, \\ \infty & \text{if } i \neq j, (i, j) \notin E. \end{cases}$$

- Output is the shortest path matrix $D = (d_{ij})$, where $d_{ij} = \delta(i, j)$.

Dynamic Programming Steps

1. Define structure of optimal solution, including what are the largest sub-problems.
2. Recursively define optimal solution
3. Compute solution using table bottom up
4. Construct Optimal solution

To help us develop the first dynamic programming approach we can restate the All-Pairs Shortest Paths problem as follows.

Find the shortest path from every vertex to every other vertex considering at most paths of $|V|-1$ edges (longest simple path for $|V|$ vertices).

1. Define structure of optimal solution.

2. Recursively define optimal solution

Slow All-Pairs Shortest Paths Algorithm

Compute a solution bottom-up: Compute $L^{(1)} = W$, then $L^{(2)}$ from $L^{(1)}$, etc. . . , $L^{(n-1)}$

EXTEND(L, W, n)

create L' an $n \times n$ matrix

for $i \leftarrow 1$ to n

for $j \leftarrow 1$ to n

$l'_{ij} \leftarrow \infty$

for $k \leftarrow 1$ to n

$l_{ij} \leftarrow \min(l'_{ij}, l_{ik} + w_{kj})$

return L'

SLOW-APSP(W, n)

$L^{(1)} \leftarrow W$

for $m \leftarrow 2$ to $n-1$

$L^{(m)} \leftarrow \text{EXTEND}(L^{(m-1)}, W, n)$

return $L^{(n-1)}$

3. What is the runtime of EXTEND and SLOW-APSP?

Improving on SLOW-APSP

Note the code to multiply two $n \times n$ matrixes ($A * B$) together to get C , an $n \times n$ matrix

for $i \leftarrow 1$ to n

for $j \leftarrow 1$ to n

$c_{ij} \leftarrow 0$

for $k \leftarrow 1$ to n

$c_{ij} \leftarrow c_{ij} + a_{ik} * b_{kj}$

4. How does this matrix multiply code compare to the EXTEND code? Why do we care?

Faster All-Pairs Shortest Paths Algorithm

Compute a solution bottom-up: Compute $L^{(1)} = W$, then $L^{(2)}$ from $L^{(1)}$, then $L^{(4)}$ from $L^{(2)}$ etc. . . , $L^{(n-1)}$

FASTER-APSP(W, n)

$L^{(1)} \leftarrow W$

$m \leftarrow 1$

while $m < n-1$

$L^{(2m)} \leftarrow \text{EXTEND}(L^{(m)}, L^{(m)}, n)$

$m \leftarrow 2m$

return $L^{(m)}$

5. What is the runtime of FASTER-APSP?

Floyd-Warshall Algorithm

To help us develop another dynamic programming approach we can restate the All-Pairs Shortest Paths problem as follows:

Find the shortest path from every vertex to every other vertex considering at most all other vertices intermediate on the paths.

6. Define structure of optimal solution.

7. Recursively define optimal solution and write pseudocode.

8. What is the run time of Floyd-Warshall?

9. Demonstrate Floyd-Warshall.

