COS 226 Midterm Review Spring 2015

Ananda **Guna**wardena
(guna)
guna@cs.princeton.edu
guna@princeton.edu

Time and location:

- The midterm is during lecture on
  - **Wednesday, March 11 from 11-12:20pm.**
- The exam will start and end promptly, so please do arrive on time.
- The midterm room is either McCosh 10 or McDonnell A02, depending on your precept date.
  - Friday Precepts: McCosh 10.
  - Thursday Precepts: McDonnell A02.
- Failure to go to the right room can result in a serious deduction on the exam. There will be no makeup exams except under extraordinary circumstances, which must be accompanied by the recommendation of a Dean.

Rules

- Closed book, closed note.
- You may bring one 8.5-by-11 sheet (one side) with notes in your own handwriting to the exam.
- No electronic devices (including calculators, laptops, and cell phones).

Materials covered

- *Algorithms, 4th edition, Sections 1.3–1.5, Chapter 2, and Chapter 3.*
- *Lectures 1–10.*
- *Programming assignments 1–4.*

concepts (so far) in a nutshell

Intro · Union Find	Quick-union · Union-by-size
Analysis of Algorithms	Binary search
Stacks and Queues	Dijkstra 2-stack
Elementary Sorts	Selection · Insertion · Shuffle
Mergesort	Merging
Quicksort	Partitioning · Quick-select
Priority Queues	Heap · Heapsort
Elementary Symbol Tables · BSTs	BST ops · Inorder traversal
Balanced Search Trees	2-3 tree · Red-black BST
Hash Tables	Linear probing

List of algorithms and data structures:

quick-find	quick-union	weighted quick-union
resizing arrays	linked lists	stacks and queues
insertion sort	selection sort	Knuth shuffle
mergesort	bottom-up mergesort	
quicksort	3-way quicksort	
binary heaps	heapsort	
sequential search	binary search	BSTs
2-3 trees	left-leaning red-black BSTs	
separate chaining	linear probing	

- Recall as much as possible about each of the above topics
- Write down up to 5 important things about each one

Analysis of Algorithms

Question

True or False

I. Tilde notation includes the coefficient of the highest order term. ✓

II. Tilde notation provides both an upper bound and a lower bound on the growth of a function. ✓

III. Big-Oh notation suppresses lower order terms, so it does not necessarily accurately describe the behavior of a function for small values of N. ?

Count operations

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      for (int k = j+1; k < N; k++)
         if (a[i] + a[j] >= a[k]) count++;
```

$i=0$ $\quad j = 1 .. n-1$ $\quad k = 2 .. N-1$ $\quad N^2$

Suppose that it takes 1 second to execute this code fragment when $N = 1000$. Using tilde notation, formulate a hypothesis for the running time (in seconds) of the code fragment as a function of N.

$T(n) \sim a n^3$ — 1000

Analysis of Algorithms

- Estimate the performance of an algorithm using
  - Θ, Ω, O, θ, order of growth
  - comparisons, array accesses, exchanges, memory requirements

$\Omega(\mathrm{lk})\ \Omega(n) \leftarrow \boxed{\Omega(n^2) \le T(n) = 2n^2 + n + 1 \rightarrow O(n^2)}$

$O(n^2) \rightarrow O(n^3) \rightarrow O(2^n)$

$\Theta(n^2)$

- best, worst, average
  - Performance measure based on some specific inputs

More formally….

- tilda notation

$f(N) \sim g(N)$ means $\lim_{N \to \infty} \frac{f(N)}{g(N)} = 1$

notation	provides	example	shorthand for	used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ $\vdots$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$ $\vdots$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$ $\vdots$	develop lower bounds

---

Amortized analysis

- Measure of average performance over a series of operations
  - Some good, few bad

```
for (int i=1; i<n; i++)
    InsertionSort(A[0..i])
```

$\frac{1}{2} i^2$

?

$$\sum_{i=1}^{n} \frac{1}{2} i^2 = \frac{1}{2} \sum_{i=1}^{n} i^2 = \frac{1}{2} \cdot \frac{n}{6}(n+1)(2n+1) \sim \frac{1}{6} n^3$$

Analysis of Algorithms

- Techniques:
  - count operations
    - Operations ➔ reads/writes, compares

    expensive
  - Derive mathematically
    - exploit the property of the algorithm
    - solve a recurrence formula to reach a closed form solution.
    - Obtain upper/lower bounds

Count operations

Example 1

```
for (int i = 1; i < N; i++) {
     for (int j = i; j > 0; j--) {
          if (genomes[j-1].length() > genomes[j].length())
               exch(genomes, j-1, j);
          else break;
     }
}
```

2

4 array acces

assign

compares	Array accesses	assignments	External method calls
$\frac{1}{2}n^2$	$6 \cdot \frac{1}{2}n^2$	$3 \cdot \frac{1}{2}n^2$	$1 \cdot \frac{1}{2}n^2$

temp = A[i]
A[i] = A[j]
A[j] = temp

Count operations

Example 2

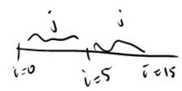
```
int uniqueCount = 0;
Arrays.sort(a); //assume that this line takes N log N time.
int i = 0;

while (i < N) {
   int current = a[i];
   uniqueCount += 1;
   int j = i + 1;
   while (j < N) {
      if (a[j] != current)
         break;
      j++;
   }
   i = j;
}
```

N log N

i++

compare



compares	Array accesses	assignments	External method calls
n	n	n	0

Runtime complexity

This is a method of describing behavior of an algorithm using runtime Observations. Runtime of an algorithm depends on many factors including language, compiler, input size, memory, optimizations etc

```
int N = Integer.parseInt(args[0]);
String[] genomes = new String[N];
for (int i = 0; i < N; i++) {
     In gfile = new In("genomeFile" + i + ".txt");
     genomes[i] = gfile.readString();
}
```

The following runtimes were observed from an algorithm that reads a file of strings and sort them using insertion sort. The runtime analysis seems to suggest the algorithm is linear. Is this correct?

N	Time (s)
1	0.15
2	0.14
4	0.19
8	0.41
16	0.85
32	1.66
64	3.38

$T(N) = a N^b$ ← power law

$b = 1$

Important when counting

- Do not assume two nested loops always give you n^2
  - Always read the code to see what it does
- When doubling or halving loop control variable, it can lead to log N performance
  - But analyze carefully
- Sometimes the sum of operations can be approximated by an integral
  - $\sum f(n) \sim \int f(n)$

n

$\sum_{i=1}^{n} \log i \sim \int_{i=1}^{n} \log i$

useful formulas

$1 + 2 + \ldots + N.$	$\sum_{i=1}^{N} i \sim \int_{x=1}^{N} x\,dx \sim \frac{1}{2}N^2$
$1^k + 2^k + \ldots + N^k.$	$\sum_{i=1}^{N} i^k \sim \int_{x=1}^{N} x^k\,dx \sim \frac{1}{k+1}N^{k+1}$
$1 + 1/2 + 1/3 + \ldots + 1/N.$	$\sum_{i=1}^{N} \frac{1}{i} \sim \int_{x=1}^{N} \frac{1}{x}\,dx = \ln N$
3-sum triple loop.	$\sum_{i=1}^{N}\sum_{j=i}^{N}\sum_{k=j}^{N} 1 \sim \int_{x=1}^{N}\int_{y=x}^{N}\int_{z=y}^{N} dz\,dy\,dx \sim \frac{1}{6}N^3$

$1 + ½ + ¼ + ⅛ + \ldots$

$\sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = 2$

$\int_{x=0}^{\infty} \left(\frac{1}{2}\right)^x dx = \frac{1}{\ln 2} \approx 1.4427$

Mathematically speaking

- write recurrences for many of the standard algorithms
  - linear search ➔ T(n) = 1 + T(n-1) ⟶ $T(n) \sim n$
  - binary search ➔ T(n) = 1 + T(n/2) ⟶ $T(n) \sim \log n$
  - merge sort ➔ T(n) = 2T(n/2) + n
  - quicksort ➔ T(n) = T(n-i-1) + T(i) + n
  - insertion sort ➔ T(n) = i + T(n-i-1)
- solve them using many of the techniques discussed
  - Repeated application with base case like T(0) = 1 or 0
  - T(n) = 1 + T(n-1) = 1 + (1 + T(n-2)) = ………

counting memory

- standard data types (int, bool, double)
- object overhead – 16 bytes
- array overhead – 24 bytes
- references – 8 bytes
- Inner class reference – 8 bytes

```
public class TwoThreeTree<Key extends Comparable<Key>, Value> {
   private Node root;

   private class Node {
      private int count;              // subtree count
      private Key key1, key2;         // the one or two keys
      private Value value1, value2;   // the one or two values
      private Node left, middle, right;   // the two or three subtrees
   }
   ...
}
```

4 16 16 24

node = 60 + 16 + 8 = 84 + 4 = 88

padding

- How much memory is needed for a 2-3 tree object that holds N nodes?

88N + 8 + 16 ~ 88n

Data Structure Performance estimates (worst or amortized)

	find	Insert	Delete	update
unordered array	n	1	n	n
ordered array	log n	n	n	log n
resizable unordered array	n	1	n	n
linked list	n	1	n	n
ordered linked list				
queue				
stack				
binary heap				
BST				
LLRB				

Stacks and Queues

Stack and queues

- Amortized constant time operations
- implementation using
  - linked lists ✓
  - resizable arrays
- many variations of stacks and queues asked in design questions
  - design a queue that allows removing a random element (in addition to deque)
  - design a queue using a resizable array
  - design a queue using two stacks

Resizing arrays

- Arrays are static, simple, random access data structures
- Arrays can be used in many applications
  - If resizing can be done efficiently
  - resizing by 1 is a bad idea (why?)
  - doubling the array is a good idea (why?) ✓
  - can we get amortized constant performance in arbitrary insertion into an array?

cost is n not 1

Using resizable arrays

- Implement a stack
  - amortized constant time : pop and push
- Implement a queue with circular array
  - amortized constant time: enque and deque

Resizable array questions

- resizing array by one gives amortized linear time per item (bad)
- resizing array by doubling/halving gives amortized constant time (good)
- What if instead of doubling the size of the array, we triple the size? good or bad?
- Resizing also includes shrinking the array by ½ . When do we do that? When the array is less than half full or ¼ full? What is a sequence of operations to justify your claim?

Possible/impossible questions

Possible/impossible questions

- We can build a heap in linear time. Is it possible to build a BST in linear time? yes No
- is it possible to find the max or min of any list in log N time? No
- Is it possible to create a collection where an item can be stored or found in constant time Hash table
- Is it possible to design a max heap where find max, insertions and deletions can be done in constant time?

No violate the n log n lower bound

Possible/impossible questions

- is it possible to sort a list of n keys in linear time, where only d (some small constant) distinct keys exists among n keys? ~dn

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

- Is it possible to find median in an unsorted list in linear time?

index = 4

3 1 2 4 5 6 2

≤3 | 3 | ≥3   n=10

Possible/impossible questions

- is it possible to implement a FIFO queue using a single array, still have amortized constant time for enque and deque? yes
- Is it possible to solve the 3-sum problem in n log n time? don't know

$n\log n < n^{1+\varepsilon}$

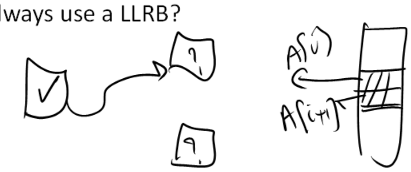
$n^3 \leftarrow a[i] + a[j] + a[k] = 0$

$n^2 \leftarrow a[i] + a[j] = -a[k]$

Why?

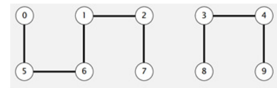
order

- Why do we ever use a BST when we can always use a hash table?
- Why do we ever use arrays when we can use linked lists?
- Why do we ever use a heap when we can always use a LLRB?



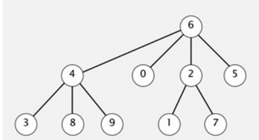
Union-find

quick-union and quick-find



	0	1	2	3	4	5	6	7	8	9
id[]	1	1	1	8	8	1	1	1	8	8

Weighted quick-union



	0	1	2	3	4	5	6	7	8	9
id[]	6	2	6	4	6	6	6	2	4	4

logarithmic union and find performance

- **Maintain heuristics**
  - when merging two trees, smaller one gets attached to larger one – height does not increase
  - Height only increase when two trees are the same size

Weighted Union-find question

Circle the letters corresponding to `id[]` arrays that *cannot* possibly occur during the execution of the weighted quick union algorithm.

		0	1	2	3	4	5	6	7	8	9
A.	a[i]:	8	0	4	0	0	4	0	4	2	0
B.	a[i]:	4	1	8	2	1	5	1	1	4	5
C.	a[i]:	3	3	6	9	3	6	3	4	1	9
D.	a[i]:	2	1	1	1	1	1	1	2	1	7

- What is the right approach to solving this?

Answer to union-find question

Circle the letters corresponding to `id[]` arrays that *cannot* possibly occur during the execution of the weighted quick union algorithm.

		0	1	2	3	4	5	6	7	8	9
A.	a[i]:	8	0	4	0	0	4	0	4	2	0
B.	a[i]:	4	1	8	2	1	5	1	1	4	5
C.	a[i]:	3	3	6	9	3	6	3	4	1	9
D.	a[i]:	2	1	1	1	1	1	1	2	1	7

A B C

A. The `id[]` array contains a cycle: $8 \to 2 \to 4 \to 0 \to 8$.

B. The height of the forest is $4 > \lg(10)$.

C. The size of tree rooted at the parent of 3 is less than twice the size of tree rooted at 3.

D. The following sequence of union operations would create the given `id[]` array:
`2-0 1-8 7-9 0-9 8-5 4-1 1-9 3-8 5-6`

Sorting

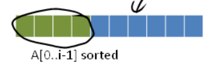
Typical question

lynx	bass	lion	bass	bass	bass	bass	gnat	wren	bass
bass	bear	frog	bear	bear	bear	bear	bass	worm	bear
bear	crab	mole	clam	crab	clam	clam	bear	oryx	clam
crab	lion	hawk	crab	lynx	crab	crab	crab	swan	crab
lion	goat	wren	frog	frog	frog	crow	lion	wolf	crow
goat	duck	lynx	gnat	goat	goat	deer	goat	mule	deer
mole	frog	crab	goat	lion	hawk	dove	duck	mole	dove
frog	dove	swan	hawk	mole	lion	duck	frog	puma	duck
swan	clam	bear	lion	clam	lynx	frog	dove	seal	frog
clam	hawk	clam	lynx	hawk	mole	gnat	clam	deer	gnat
hawk	deer	bass	lynx	swan	swan	goat	hawk	lion	goat
wren	crow	goat	mole	wren	wren	hawk	deer	goat	hawk
mule	gnat	mule	mule	gnat	gnat	mule	crow	bear	lion
oryx	lynx	oryx	oryx	lynx	lynx	oryx	lynx	lynx	lynx
gnat	lynx	gnat	swan	mule	mule	lynx	lynx	gnat	lynx
lynx	puma	lynx	wren	oryx	oryx	lynx	oryx	lynx	mole
puma	worm	puma	puma	crow	puma	puma	puma	frog	mule
worm	seal	worm	worm	puma	worm	worm	worm	crab	oryx
seal	oryx	seal	seal	seal	crow	seal	seal	bass	puma
crow	mule	crow	crow	worm	deer	lion	mule	crow	seal
deer	wolf	deer	deer	deer	dove	wren	wren	clam	swan
wolf	wren	wolf	wolf	dove	duck	wolf	wolf	hawk	wolf
dove	swan	dove	dove	duck	seal	mole	swan	dove	worm
duck	mole	duck	duck	wolf	wolf	swan	mole	duck	wren
----	----	----	----	----	----	----	----	----	----
0									1

Use the invariants to identify the sort algorithm

basic sorts

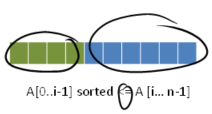
original



- *insertion sort*
  - invariant: A[o..i-1] is sorted
  - perform well in practice for almost sorted data
  - can be used in quicksort and merge sort to speed things up

basic sorts

- *selection sort*
  - Invariant: A[0...i-1] is sorted and are the smallest elements in the array
  - not used in practice much

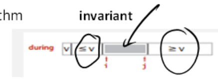


Linearithmic sorts

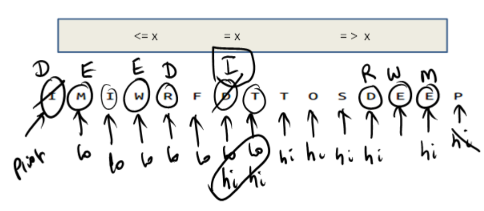
Standard or 2-way Quick sort

- randomize the array
- find a pivot (A[lo] usually)
- partition the array to find a pivot position j such that *A[j] = pivot*
  - *A[lo...j-1] < A[j] and A[j] < A[j+1...hi-1]*
  - Pointers stop and swap on equal keys to pivot
- recurse on subarrays leaving the pivot in-place
- properties
  - good general purpose $n \log_2 n$ algorithm
    - partitioning takes linear time
  - not stable
  - in-place
  - ideal for parallel implementations
  - choosing a bad pivot can lead to quadratic performance
  - Works well when no duplicates

invariant

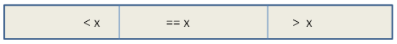


Demo of 2-way quick sort

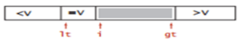


3-way quick sort

- same as 2-way quicksort
- works well with duplicate keys
- same process
  - choose a pivot, say x
  - partition the array as follows

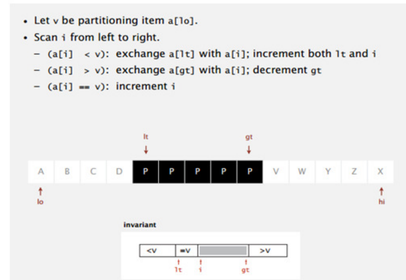


  - Invariant



- uses Dijkstra's 3-way partitioning algorithm

3-way partitioning demo

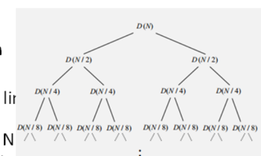


Demo of 3-way quick sort

I M I W R F D T T O S D E E P

Top-down merge sort

- facts
  - recursive
  - merging is the main operation
- performance
  - merging 2-sorted arrays takes linear time
  - merge sort tree is of height lg N
  - consistent linearithmic algorithm
- other properties
  - uses extra linear space
  - Stable
    - equal keys retain relative position in subsequent sorts



Properties
- Left most items get sorted first
- Look for 2-sorted, 4-sorted etc

bottom-up merge sort

- facts
  - iterative
  - merges sub-arrays of size 2, 4, 8 (lg N times) to finally get a sorted array
- performance
  - merging all sub arrays takes linear time in each step
  - merge continues lg N times
  - consistent linearithmic algorithm
- other properties
  - no extra space
  - stable
    - merge step retains the position of the equal keys

Properties
- Look for 2-sorted, 4-sorted, 8-sorted etc

Heap Sort

- build a max/min heap
- delete max/min and insert into the end of the array (if heap is implemented as an array) until heap is empty
- performance is linearithmic
- is heap sort stable?

Knuth shuffle

- Generates random permutations of a finite set
- algorithm

```
for (int i=n-1; i > 0; i--) {
    j = random(0..i);
    exch(a[j], a[i]);
}
```

sorting question

- Suppose you are sorting n-equal keys labeled $k_1\ k_2\ k_3\ k_4 \ldots k_n$
- Identify the number of compares (in terms of n) required when applying the following algorithms
  - Insertion sort
  - Selection sort
  - 2-way quicksort
  - 3-way quicksort
  - mergesort
  - heapsort

Problem 3 – sort matching

Sort Invariants

- Insertion sort – A[0..i] is sorted, A[i+1..n-1] is the original
- Selection sort – A[0..i] sorted and A[0..i] <= A[i+1..n-1]
- 2-way quicksort – first element is the pivot p and array is divided as A[<= p | p | >= p]
- 3-way quicksort - A[< p | == p | > p]
- Mergesort (bottom-up) – pairs of elements (2's, 4's, 8's etc get sorted. Working on whole array at once)
- Mergesort (top-down) – pairs of elements (2's, 4's, 8's etc) get sorted. Working from left to right
- Heapsort - A[1.. i] is a max heap and A[i+1..n-1] are sorted and are the largest elements
- Knuth shuffle – A[0..i] get shuffled first and display random form.

lynx	bass	lion	bass	bass	bass	bass	gnat	wren	bass
bass	bear	frog	bear	bear	bear	bear	bass	worm	bear
bear	crab	mole	clam	crab	clam	clam	bear	oryx	clam
crab	lion	hawk	crab	lynx	crab	crab	crab	swan	crab
lion	goat	wren	frog	frog	frog	crow	lion	wolf	crow
goat	duck	lynx	gnat	goat	goat	deer	goat	mule	deer
mole	frog	crab	goat	lion	hawk	dove	duck	mole	dove
frog	dove	swan	hawk	mole	lion	duck	frog	puma	duck
swan	clam	bear	lion	clam	lynx	frog	dove	seal	frog
clam	hawk	clam	lynx	hawk	mole	gnat	clam	deer	gnat
hawk	deer	bass	lynx	swan	swan	goat	hawk	lion	goat
wren	crow	goat	mole	wren	wren	hawk	deer	goat	hawk
mule	gnat	mule	mule	gnat	gnat	mule	crow	bear	lion
oryx	lynx	oryx	oryx	lynx	lynx	oryx	lynx	lynx	lynx
gnat	lynx	gnat	swan	mule	mule	lynx	lynx	gnat	lynx
lynx	puma	lynx	wren	oryx	oryx	lynx	oryx	lynx	mole
puma	worm	puma	puma	crow	puma	puma	puma	frog	mule
worm	seal	worm	worm	puma	worm	worm	worm	crab	oryx
seal	oryx	seal	seal	seal	crow	seal	seal	bass	puma
crow	mule	crow	crow	worm	deer	lion	mule	crow	seal
deer	wolf	deer	deer	deer	dove	wren	wren	clam	swan
wolf	wren	wolf	wolf	dove	duck	wolf	wolf	hawk	wolf
dove	swan	dove	dove	duck	seal	mole	swan	dove	worm
duck	mole	duck	duck	wolf	wolf	swan	mole	duck	wren
----	----	----	----	----	----	----	----	----	----
0	6	9	7	4	3	2	5	8	1

(0) Original input
(1) Sorted
(2) Selection sort
(3) Mergesort (*top-down*)
(4) Mergesort (*bottom-up*)
(5) Quicksort (*standard, no shuffle*)
(6) Quicksort (*3-way, no shuffle*)
(7) Insertion sort
(8) Heapsort
(9) Knuth shuffle

Priority Queues

Binary heaps

- Invariant
  - for each node N
    - Key in N >= key in left child and key in right child
- good logarithmic performance for
  - insert
  - remove max
  - find max (constant)
- heap building
  - bottom-up ➔ linear time (sink each level)
  - top-down ➔ linearithmic (insert and swim)

Heap questions

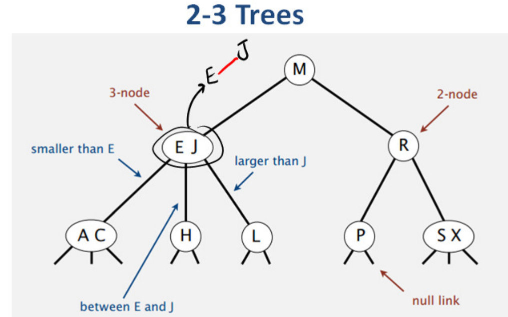
- Given a heap, find out which key was inserted last?
  - it must be along the path of the right most leaf node in the tree
  - We always delete the root by exchanging that with the last leaf node
- Build a heap
  - Bottom-up
  - Top-down
- Applications
  - can be used in design questions where delete, insert takes logarithmic time and find max takes constant time

Ordered Symbol Tables

	sequential search	binary search	BST
search	N	$\log N$	h
insert	N	N	h
min / max	N	1	h
floor / ceiling	N	$\log N$	h
rank	N	$\log N$	h
select	N	1	h
ordered iteration	$N \log N$	N	N

Balanced Trees

2-3 Trees

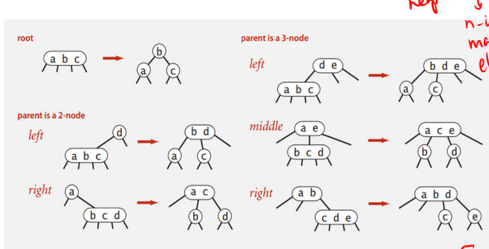


Two invariants

- Balance invariant – each path from root to leaf nodes have the same length
- Order invariant – an inorder traversal of the tree produces an ordered sequence

2-3 Tree operations

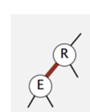
i
max heap
n-i
max elem

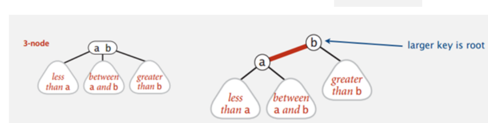


X 5 2 3 1 1 6 7 8 9

Red-black trees

- How to represent 3-nodes?
  - Regular BST with red "glue" links.

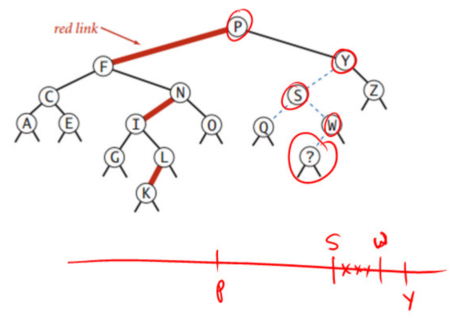




Red-black tree properties

- A BST such that
  - No node has two red links connected to it
  - Every path from root to null link has the same number of **black links**
  - Red links lean left.

examples



Red-black tree questions

- add or delete a key to/from a red-black tree and show how the tree is rebalanced
- Determining the value of an unknown node
  - Less than M, greater than G, less than L
- Know all the operations
  - Left rotation, right rotation, color flip
  - Know how to build a LLRB using operations
- Know how to go from 2-3 tree to a red-black tree and vice versa

Symbol Tables

hashing

- simple idea
- given a key, find a hash function H(key) that computes an integer value.
- create a table of size M and use H(key)%M to find a place.
- hard to avoid collisions
  - separate chaining
  - linear probing
- choose a good hash function
  - Easy to compute
  - Avoid collisions
  - Keep chain lengths to be Θ (log N / log log N) using a random distribution of keys

Hashing type questions

- Given a set of keys, which table could result in?
  - Look for keys that are in the table corresponding to their hash values
    - They were inserted first
  - There must be at least one key that is in the position of the hash value (first key inserted)
- Know the value of a good hash function
- Know how collisions are resolved using
  - Separate chaining
  - Linear probing
- Know when to resize the hash table

Algorithm and Data Structure Design

Covered in details in design session.
See design notes on midterm site

Design problems

- Typically challenging
- There can be many possible solutions
  - partial credit awarded
- Usually it is a data structure design to meet certain performance requirements for a given set of operations
  - Example, create a data structure that meets the following performance requirements
    - findMedian in ~1, insert ~ lg n, delete ~ lg n
  - Example: A leaky queue that can remove from any point, that can insert to end and delete from front, all in logarithmic time or better
- Typical cues to look for
  - *log n* time may indicate that you need a sorted array or balanced BST or some sort of a heap
  - Amortized time may indicate, you can have some costly operations once in a while, but on average, it must perform as expected

design problem #1

- Design a randomizedArray structure that can insert and delete a random Item from the array. Need to guarantee amortized constant performance
  - Insert(Item item)
  - delete()

design problem #2

15 | 12 | 11

dog

An ExtrinsicMaxPQ is a priority queue that allows the programmer to specify the priority of an object independent of the intrinsic properties of that object. This is unlike the MaxPQ from class, which assumed the objects were comparable and used the compare method to establish priority. You may assume the Items are comparable.

```
public class ExtrinsicMaxPQ<Item extends Comparable<Item>> {
    ExtinsicMaxPQ()                    //do not implement
    void put(Item x, int priority)
    Item delMax()
}
```

cat, 12
dog, 10 15
swimp, 11

If an Item already exists in the priority queue, then its priority is changed instead of adding another item. All operations should complete in **amortized logarithmic time in the worst case**. Your ExtrinsicMaxPQ should use **memory proportional to the number of items. For a <u>small</u> amount of partial credit, you may assume that no Item's priority is ever changed (i.e. no item is inserted twice).**

dog, 15
dog 15
cat, 12
swimp

Example:

```
put("cat", 12)       // cat is inserted with priority 12
put("dog", 10)
put("swimp is a raccoon who enjoys fries and does not like to eat dirt", 11)
put("dog", 15)       // dog's priority is changed to 15
delMax()             // deletes dog, which has priority 15, cat is now max
put("fish", 20)      // fish is inserted, and is now max
put("fish", 11)      // fish's priority is reduced to 11, cat is again max
delMax()             // removes cat (priority 12), either swimp or fish now max
delMax()             // removing either swimp or fish is OK, both priority 11
```

Design Problem #3

`public class MoveToFront<Item>`

`MoveToFront()`	*create an empty move-to-front data structure*
`void add(Item item)`	*add the item at the front (index 0) of the sequence (thereby increasing the index of every other item)*
`Item itemAtIndex(int i)`	*the item at index i*
`void mtf(int i)`	*move the item at index i to index 0 (thereby increasing the index of items 0 through i – 1)*

All operations should take time proportional to $\log N$ *in the worst case, where* N *is the number of items in the data structure.*

Key choices for design problems

- Choice of data structure
  - LLRB
    - insert, delete, rank, update, find, max, min, rank (logarithmic time)
  - Hash table
    - insert, find, update (constant time)
  - Heap
    - delMax, insert (logarithmic)
  - symbol table
    - ordered (same as LLRB), unordered (same as hash table)
  - LIFO Stack and FIFO queue
    - inserts and deletes in amortized constant time

1
lg n
n lg n
n^2 lg n