COS 226
Midterm Review
Fall 2015

guna@cs.princeton.edu

Time and location:

- The midterm is during lecture on
  - **Tuesday, Oct 27th from 11-12:20pm.**
- The exam will start and end promptly, so please do arrive on time.
- The midterm room is
  - McDonnell A01 - P03 P03A - all 11am precepts
  - Jadwin 10 - P01 P05 - 9am and 1:30pm precepts
  - Friend 101 - P02 P02A P04 P04A - all the even precepts (10am and 12:30pm)
- 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 1–11*
- *Programming assignments 1–5.*

concepts (so far) in a nutshell

List of algorithms and data structures:

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

Sorting Invariants

An Invariant

- An invariant is a statement that is TRUE throughout the execution of an algorithm
  - Insertion sort
    - A[0..0] is sorted at the beginning (pre-condition)
    - A[0..i] sub-array is sorted at any intermediate point (loop-invariant)
    - A[0..N-1] sorted at the end (post-condition)

Question?

1. Which sorting algorithms have the invariant A[0…i] sub-array is sorted after i-steps?
   1. Insertion sort ✓
   2. Selection sort ✓
   3. Top-down mergesort ✗
   4. Bottom-up mergesort ✗
   5. 2-way quicksort ✗
   6. 3-way quicksort ✗
   7. Heapsort ✗
   8. Knuth Shuffle ✗

Question?

2. Which sorting algorithms have the invariant A[i+1…N-1] sub-array is the original after i-steps?

1. Insertion sort
2. Selection sort
3. Top-down mergesort
4. Bottom-up mergesort
5. 2-way quicksort
6. 3-way quicksort
7. Heapsort
8. Knuth Shuffle

Question?

3. Which sorting algorithms have the invariant A[0…i] are the smallest in the array after i-steps?

1. Insertion sort
2. Selection sort
3. Top-down mergesort
4. Bottom-up mergesort
5. 2-way quicksort
6. 3-way quicksort
7. Heapsort
8. Knuth Shuffle

Question?

4. Which sorting algorithms have the invariant A[0] is the largest in the array after i-steps?

1. Insertion sort
2. Selection sort
3. Top-down mergesort
4. Bottom-up mergesort
5. 2-way quicksort
6. 3-way quicksort
7. Heapsort
8. Knuth Shuffle

Question?

5. Which sorting algorithms have the invariant A[0..i] is a rearrangement and A[i+1..N-1] is the original in the array after i-steps?

1. Insertion sort ✓
2. Selection sort ✗
3. Top-down mergesort ✓
4. Bottom-up mergesort ✗
5. 2-way quicksort ✗
6. 3-way quicksort ✗
7. Heapsort ✗
8. Knuth Shuffle ✓

Insertion, selection or mergesort?

null	find	find	exch	exch
type	hash	hash	fifo	fifo
null	heap	heap	find	find
hash	lifo	link	hash	hash
null	link	list	heap	heap
heap	list	null	leaf	leaf
sort	null	null	left	left
link	null	null	less	less
list	null	push	lifo	lifo
push	null	sort	link	link
find	null	swap	list	list
swap	push	type	next	next
null	root	null	null	node
null	sort	null	null	null
root	swap	root	root	null
lifo	type	lifo	null	null
swap	exch	swap	swap	null
leaf	fifo	leaf	null	null
tree	leaf	tree	tree	null
path	left	path	path	null
node	less	node	node	null
left	next	left	sort	path
less	node	less	push	push
exch	null	exch	null	root
null	null	null	null	sink
sink	null	sink	sink	sort
swim	path	swim	swim	swap
null	sink	null	null	swap
next	swap	next	swap	swap
swap	swap	swap	swap	swim
fifo	swim	fifo	type	tree
null	tree	null	null	type
----	----	----	----	----
0	4	2	3	1

Further invariants:

2. Insertion: A[i+1…N-1] is the original

3. Selection: A[0…i] are the smallest in the array and are in the final position

4. Mergesort: A[0..i] is sorted and I = 2^k for some k

2-way and 3-way quicksort

The pivot (first entry) is placed in the final position

2-way - All elements to the left are <= and right are >=

3-way - All elements equal to pivot are in the middle. Left is < and right is >

- How can one recognize 2-way from 3-way if first entry has no duplicates?
  - **2-way** does not swap a lot and do one swap of the pivot at the end. That is, all elements <= pivot must stay in their original position and all elements >= must stay in their original position
  - **3-way** takes the pivot with it and making swaps. All elements < have moved to the left as pivot moves forward

2-way or 3-way?

≤

null	hash	fifo	exch
type	heap	next	fifo
null	fifo	null	find
hash	link	hash	hash
null	list	null	heap
heap	next	heap	leaf
sort	find	exch	left
link	lifo	link	less
list	exch	list	lifo
push	leaf	less	link
find	less	find	list
swap	left	left	next
null	node	node	node
null	null	leaf	null
root	null	lifo	null
lifo	null	null	null
swap	null	swap	null
leaf	null	null	null
tree	null	tree	null
path	null	path	null
node	null	null	null
left	path	swap	path
less	tree	push	push
exch	swap	sort	root
null	sink	null	sink
sink	swim	sink	sort
swim	root	swim	swap
null	swap	null	swap
next	swap	type	swap
swap	push	swap	swim
fifo	sort	null	tree
null	type	root	type
----	----		----
0	6	5	1

5. 2-way quicksort

6. 3-way quicksort

navy	corn	mist	bark
plum	mist	coal	blue
coal	coal	jade	cafe
jade	jade	blue	coal
blue	blue	cafe	corn
pink	cafe	herb	dusk
rose	herb	gray	gray
gray	gray	leaf	herb
teal	leaf	dusk	jade
ruby	dusk	mint	leaf
mint	mint	lime	lime
lime	lime	bark	mint
silk	bark	corn	mist
corn	navy	navy	navy
bark	silk	wine	palm
wine	wine	silk	pine
dusk	ruby	ruby	pink
leaf	teal	teal	plum
herb	rose	sage	rose
sage	sage	rose	ruby
cafe	pink	pink	sage
mist	plum	pine	silk
pine	pine	palm	teal
palm	palm	plum	wine
----	----	----	----
0	5	6	1

Heapsort and Knuth Shuffle

null	type	exch
type	tree	fifo
null	swim	find
hash	swap	hash
null	push	heap
heap	swap	leaf
sort	swap	left
link	null	less
list	list	lifo
push	path	link
find	less	list
swap	null	next
null	sink	node
null	null	null
root	sort	null
lifo	null	null
swap	link	null
leaf	leaf	null
tree	hash	null
path	null	null
node	node	null
left	left	path
less	find	push
exch	exch	root
null	null	sink
sink	heap	sort
swim	null	swap
null	null	swap
next	next	swap
swap	root	swim
fifo	fifo	tree
null	lifo	type
----	----	----
0	7	1

max

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

Invariants

7. Heapsort :

A[1..i] is a max-heap and A[i+1..N] is sorted and are the max elements in the array. Note that I = N

8. Knuth shuffle

A[0..i] is randomized and A[i+1..N-1] is the original

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++;
```

Assume T = a. N^3
1 = a. (1000)^3
a=1/10^9

T = (1/10^9)* N^3

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.

Analysis of Algorithms

- Performance of an algorithm is measured using
  - comparisons, array accesses, exchanges,
  - memory requirements
- best, worst, average
  - Performance measure based on some specific input type
    - Eg: sorted, random, reverse sorted. Almost sorted

Examples

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

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
  - Overall good (or not bad)
- Array resizing
  - Resize by 1 as extra space is needed
  - Resize by doubling the size as extra space is needed
  - Resize by tripling the size as extra space is needed

Amortized analysis

- Resize by 1 as needed
  - Start with N=1, 1W (new)
  - N=2, 1W (to copy), 1W(new)
  - N=3, 2W (to copy), 1W (new)
  - Total writes (to write N elements)
    - 1+2+… + (N-1) + (1+ 1 + 1 + ..+1) ~ N^2
    - Cost per operation ~ N (expensive)
- Resize by doubling
  - Start with N=1, 1W (new)
  - N=2, 1W (to copy), 1W(new)
  - N=4, 2W (to copy), 2W (new)
  - N=8, 4W (to copy), 4W (new)
  - Also each array copy require equal number of reads. Hence we double the sum
  - Total writes (to write N = 2^k elements for some k)
    - 2(1+2+… + 2^{k-1})+ (1+2+… + 2^{k-1}) ~ 3N
    - Cost per operation ~ 3 (cheap)

Use 1 + 2 + 4 + .. 2^(k-1) = 2^k -1
Assume that N = 2^k for some k
Hence we have the doubling sum to
= N

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.	$\sum_{i=1}^{N}\sum_{j=i}^{N}\sum_{k=j}^{N} 1 \sim \int_{x=1}^{N}\int_{y=x}^{N}\int_{z=y}^{N} dz\,dy\,dx \sim \frac{1}{6}N^3$

$1 + ½ + ¼ + ⅛ + \ldots$

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

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

counting memory

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

```
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
   }
   ...
}
```

- How much memory is needed for a 2-3 tree object that holds N nodes? Express the answer in tilde notation, big O notation
  - For each node we need= 4+16+16+24 = 60
  - Add overhead and inner class bytes = 60 +16 + 8 = 84 + 4(padding) = 88
  - Total = 8(root) + 16(overhead) + 88N

Algorithm and Data Structure Design

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()

Solution: Any resizable array would do. To delete a random item, just swap with the last one and delete. All can now be done in amortized constant time

design problem #2

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>> {
        ExtrinsicMaxPQ()                  //do not implement
        void put(Item x, int priority)
        Item delMax()
}
```

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).**

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 #2 solution

- The key here is that we can use a max-heap, but updating the priority in a max-heap is linear time.
- Hence we need a better way to update. The trick is to have a ST of (Item, reference) where reference is a node in the item max-heap
  - Put(item, prio) – check if the item is in the ST. If so follow the pointer and update the priority of the node . Sink or swim based on new priority
  - If the item is not in ST, add the item to the ST, add a node to max-heap, record the reference of the node in ST and make the connection.

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.*

#3 solution

- This is a tricky one. You need to maintain a counter that indicates the last one inserted.
  - Add(item) - decrement the counter and add the node to a LLRB with the new counter. The new one is the smallest and supposed to be the left most element in LLRB
  - Decrementing the priority of all others is implicit. Explicit do not work as it would require N time to do this.
  - Move to front(index i) – find the entry with the index (BST search in log N), delete it, insert as a new node. This will now become the new node
  - We probably need to maintain counters of smallest (to insert) and largest (one that is the oldest)

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

Bonus Slides

Possible/impossible questions

Possible/impossible questions

- We can build a heap in linear time. Is it possible to build a BST in linear time?
  - No. if we can, then we can sort in N time. Not possible
- is it possible to find the max or min of any list in log N time?
  - This would mean that we do not look at all the elements in the list. Hence a lower bound may be N for this operation
- Is it possible to create a collection where an item can be stored or found in constant time
  - Yes, we can use a hashtable to do this
- Is it possible to design a max heap where find findMax, insert and delMax can be done in constant time?
  - If this is possible, we can build a heap in N time, do delMax n times and then have a heapsort in N time. This is not possible

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?
  - Yes, this is possible with 3-way quicksort
  - It would require ~ dN operations. As long as d is not a function of N, we can do the sorting in linear time
- Is it possible to find median in an unsorted list in linear time?
  - Yes, if we use quickselect, then this can be done in N time

Possible/impossible questions

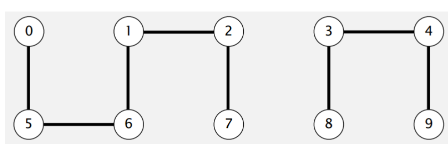
- is it possible to implement a FIFO queue using a single array, still have amortized constant time for enque and deque?
  - Yes, use a resizable array two stacks (or circular array), where one array is enque and other is deque
- Is it possible to solve the 3-sum problem in n log n time?
  - We do know it can be solved in N^2 time.
  - Others have obtained lower bounds.
  - But it is not yet known the 3-sum can be done in N^(2-epsilon) for some epsilon

Why?

- Why do we ever use a BST when we can always use a hash table?
  - BST provide more order operations
- Why do we ever use arrays when we can use linked lists?
  - Arrays are random access and simple to implement
- Why do we ever use a heap when we can always use a LLRB?
  - LLRB and heaps do totally different things. LLRB gives efficient order operations
- Why do we ever use a LLRB when we can always randomize and get a balance BST?
  - If data is not all available at the time of building BST , this would not work. So we need a dynamic balance condition

Union-find

quick-union and quick-find



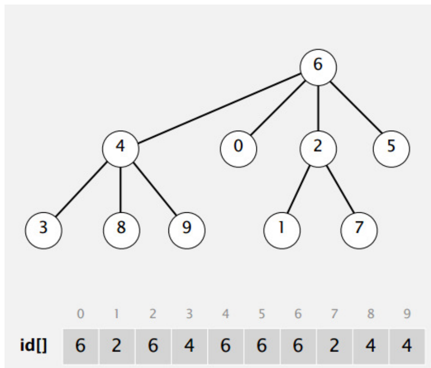
Given above structure, fill in the id[] table. Assume that 1 and 8 are the roots

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

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

The roots of the tree are 1 and 8. Assuming we start with a forest of trees (with one node) we can give some order of union operations Union(2,7), union(2,1), union(1,

Weighted quick-union



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?
  - Be sure that it is a tree (no cycles)
  - Be sure that the size of the tree is lg N

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`

Extra Slides

Demo of 2-way quick sort

<= x	= x	=> x

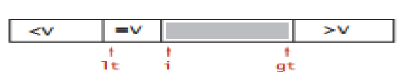
I	M	I	W	R	F	D	T	T	O	S	D	E	E	P
I	E												M	
I	E	E	D								W	I	M	
I	E			D		R	T	T	O	S			M	
F	E	E	D	D	I	R	T	T	O	S	W	I	M	P

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

< x	== x	> x

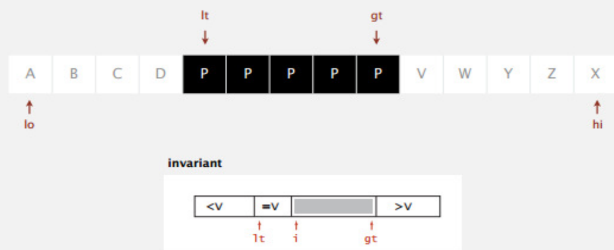
  - Invariant



- uses Dijkstra's 3-way partitioning algorithm

3-way partitioning demo

- Let v be partitioning item a[lo].
- Scan i from left to right.
  - (a[i] < v): exchange a[lt] with a[i]; increment both lt and i
  - (a[i] > v): exchange a[gt] with a[i]; decrement gt
  - (a[i] == v): increment i

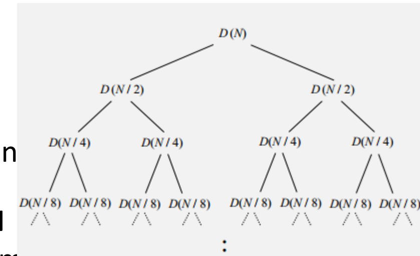


Demo of 3-way quick sort

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

Top-down merge sort

- facts
  - recursive
  - merging is the main operation
- performance
  - merging 2-sorted arrays takes lin 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 heap
- delete max and insert into the end of the array (if heap is implemented as an array) until heap is empty
- performance is linearithmic

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]);
}
```

Priority Queues

Binary heaps

- Invariant
  - for each node N
    - Key in N >= key in left child and key in right child (order invariant)
    - A complete binary tree – all levels are full except perhaps the last level. Elements are added to the last level from L to R
- good logarithmic performance for
  - Insert(Item item)
  - delMax()
  - max()
- heap building
  - bottom-up → linear time (sink each level)
  - top-down → linearithmic (insert and swim)

Heap questions

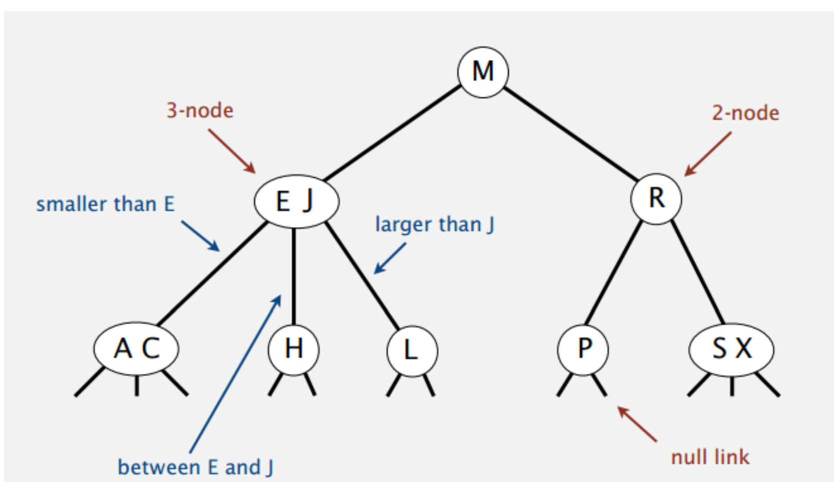
- Given a heap, find out which key was inserted last?
  - it must be along the path of the right most leaf node in the tree
  - We always delete the root by exchanging that with the last leaf node
- Build a heap
  - Bottom-up
  - Top-down
- Applications
  - can be used in design questions where delete,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

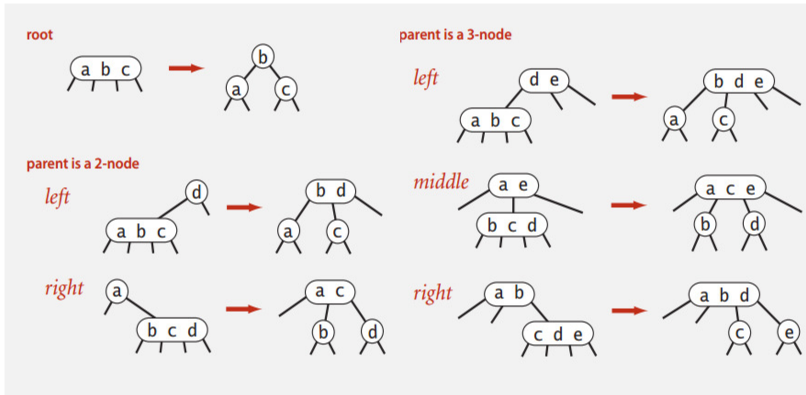
2-3 Trees



Two invariants

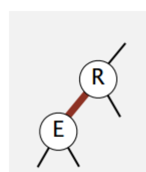
- Balance invariant – each path from root to leaf nodes have the same length
- Order invariant – an inorder traversal of the tree produces an ordered sequence

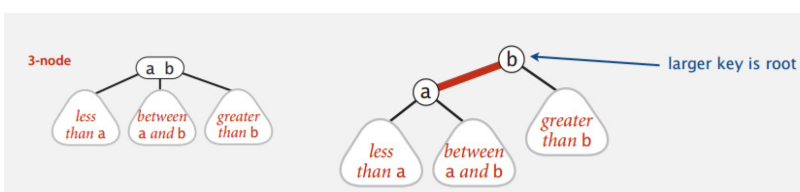
2-3 Tree operations



Red-black trees

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

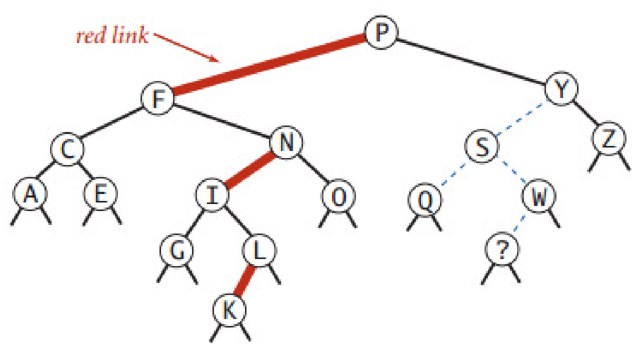




Red-black tree properties

- 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