Week 1 - Dynamic Connectivity

March 4, 2017 1:24 PM

Union - connect two objects together
Connected - check if two objects are connected

Applications

When programming, associate objects with integers 0 - N-1

- can insert into an array and access information easily (indexes)
- supresses details

Connections

- Reflective
  - p is connected to p
- Symmetric
  - if p is connected to q then q is connected to p
- Transitive
  - if p is connected to q and q is connected to r p is connected to r

Connective components

- maximal set of objects that are mutually connected
  - {0}{123}{456789}
- can gain efficiency using this method

Find Query

- check if two objects are in the same components

Union Command

- Replace components containing two objects with their union

Union-Find Data Type (API)

- Specification of methods
  - public class UF
    - UF(int N)
    - void union (int p, int q)
    - boolean connected (int p, int q)
    - int find(int p)
    - int count()

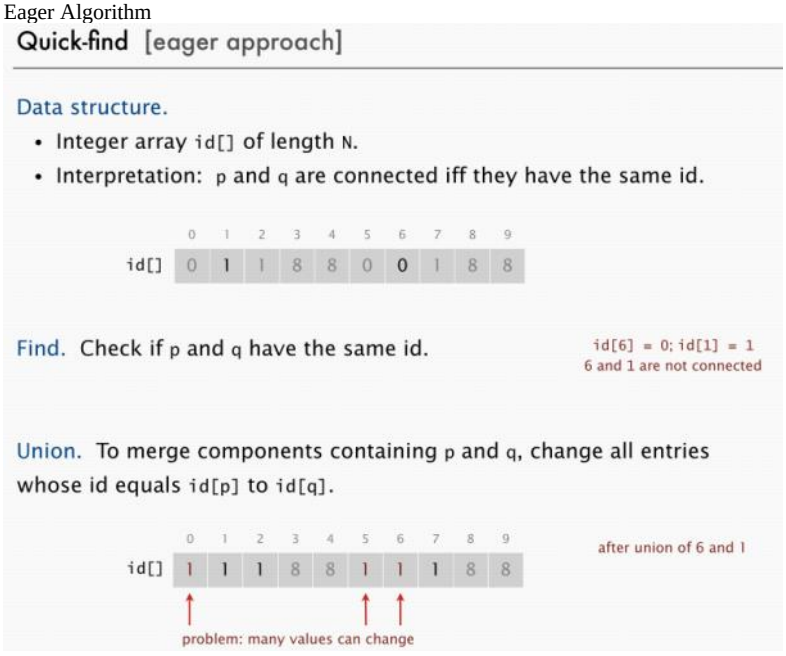
Dynamic-connectivity client

- Check data type API design
- building a client that uses the data type
- read information from standard input and read 2 integers and connect them

Week 1 - Quick Find

March 4, 2017 5:54 PM

Eager Algorithm



Data structure

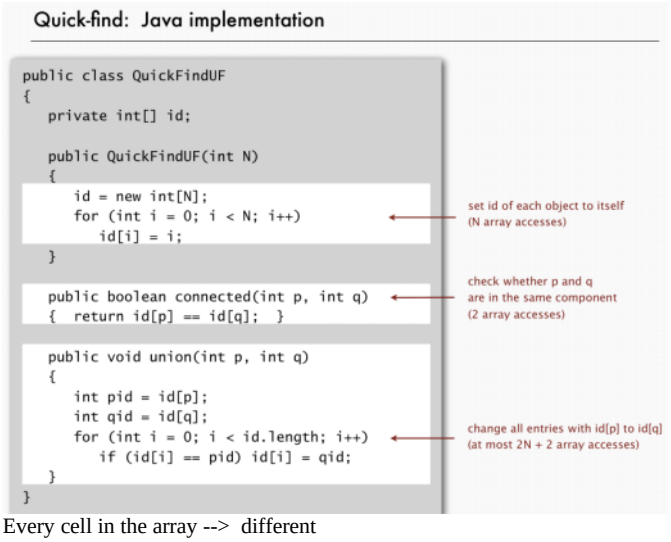
- integer array indexed by objects
- interpretation: p and q are connected if and only if their entry in the array are the same

To find: check the array value of indexes p and q (if they are the same)

To Union: change the entries of p -> q

- id[p] -> id[q]
- problem with large number of objects

Code:



Every cell in the array --> different

Implementation of Union trick

- make variables for id[p] and [id[q] before implementing.

Effectiveness:

number of times the code has to access the array

Algorithm	Initialize	Union	Find
Quick Find	N	N	1

Union command too expensive in memory and time.

- not scalable
- quadratic algorithm = bad

Screen clipping taken: 14/05/2017 5:06 PM

Week 1 - Quick Union

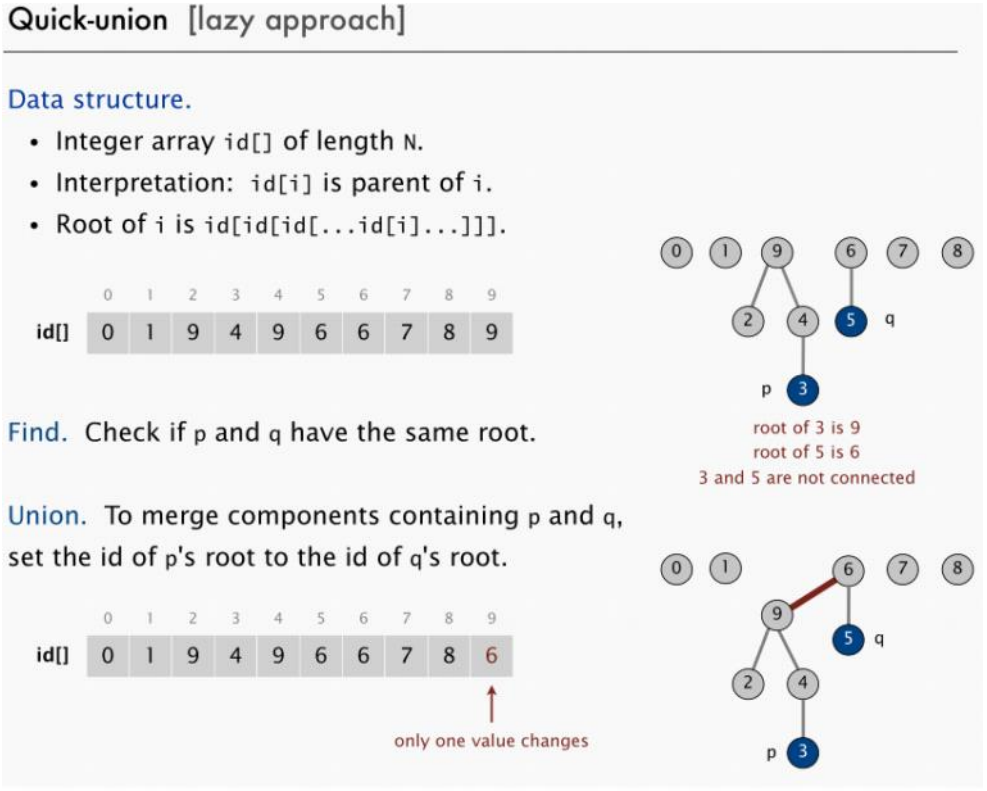
March 4, 2017 6:08 PM

(Lazy Approach) - avoid doing work until we have to

Same data structure as quick find

Different representation

- set of trees called forest
- each entry in the array will contain reference to parent in the tree.

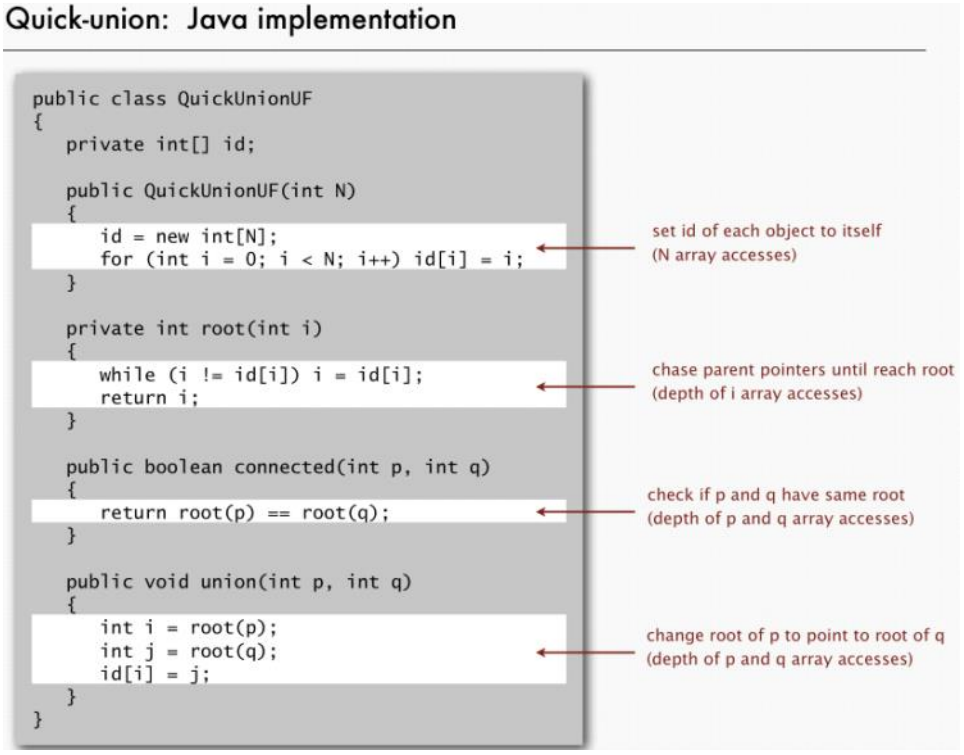


Find: check if p and q have the same root

Union: to merge set id of p's root's to id of q

- id[p's root] = q

Code Implementation



follow id[i] until id[i] = I

- private method

Effectiveness

Algorithm	Initialize	Union	Find
Quick Find	N	N	1
Quick Union	N	N ?	N

- Quick Union - worst case scenario.

Defect

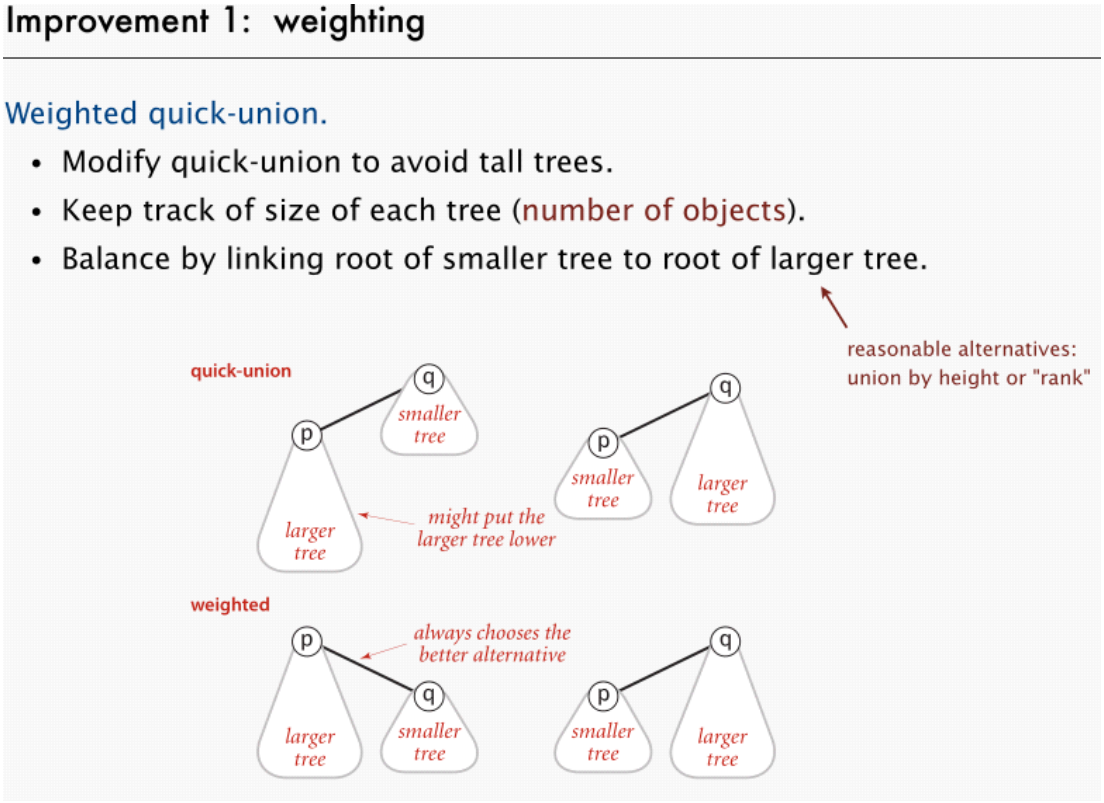
- trees can get too tall
    - find operation too expensive

Week 1 - Quick Union Improvements

May 15, 2017 2:42 PM

Weighting

- take steps to avoid having tall trees
- avoid putting the large tree lower
    - keep track of the number of objects in each tree
    - connect the root of the smaller tree to the root of the larger tree
- make the trees flat instead of long
- SMALLER TREE GOES BELOW



Screen clipping taken: 15/05/2017 3:17 PM

Java implementation

- same data structure as quick union
- need an extra array to gives the number of items rooted at I
- FIND = identical
- UNION = check sizes and then link root of smaller to the larger
    - change the size array

Weighted quick-union: Java implementation

Data structure. Same as quick-union, but maintain extra array `sz[i]` to count number of objects in the tree rooted at `i`.

Find. Identical to quick-union.

```
return root(p) == root(q);
```

Union. Modify quick-union to:

- Link root of smaller tree to root of larger tree.
- Update the `sz[]` array.

```
int i = root(p);
int j = root(q);
if (i == j) return;
if  (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
else                { id[j] = i; sz[i] += sz[j]; }
```

Screen clipping taken: 15/05/2017 3:18 PM

Running time

- find - proportional to depth of the tree
    - depth is lg N - lg = log base 2

Algorithm	Initialize	Union	Connected
quick-find	N	N	1
quick-union	N	N	N
weighted	N	lg N	lg N

Can improve method - path compression

Path Compression

- after computing the root of any sub tree, set the id of the sub tree have that root
- one extra line of code
    - second loop to root() to set the id of the selected node to the original root

No such algorithm that can be linear - only algorithms that are linear in practice

Week 1 - Union-Find Applications

May 15, 2017 3:19 PM

Percolation

- model for many physical systems
  - nxn grid
  - each square is a site
  - each site is has a probability p
  - a system is percolated if the top and the bottom are connected by open sites
    - electricity
      - open conductor closed insulator
    - water flow
    - social network
  - open with a given probability (p)
    - if site vacancy is low then does not percolate
    - if site vacancy is high then it does percolate
    - in the middle - phase transition
      - threshold between percolate and not percolate is sharp
        - there is a value to determine if a system will or will not percolate p*
        - run simulations to determine p*
  - Monte Carlo simulation
    - the nxn square is all blocked
    - randomly open squares until the system percolates
      - calculate vacancy percentage
      - run many times and find average
  - Dynamic connectivity example with percolation
    - create a virtual top and bottom site
    - to check we check if the virtual top and bottom are connected
  - To open a site connect to all open adjacent squares
    - calls to union

Steps to developing a usable algorithm

1. Model the problem
2. Find an algorithm to solve
3. Fast and fits in memory?
   a. if not figure out why
4. Find a way to address the problem
5. Iterate until satisfied

Week 1 - Programming Assignment Takeaways

June 2, 2017 10:01 PM

Do not redeclare instance variables in the constructor - will overwrite the instance variables.

Backwash - two bottom or top sites are both connected to the bottom virtual site but not to each other.

- connecting them to the virtual site connects them to each other inadvertently

Fix

- make another weighted quick union without the virtual sites - eliminates backwash - only use for isFull()

Week 1- Analysis of Algorithms- Observations

June 3, 2017 6:52 PM

3 Sum problem

- n distinct integers how many triples sum to exactly 0
- runs a loop that checks each triple

How long will this program take to run?

- Empirical analysis
  - Stopwatch class
  - run program and measure running time
- Data analysis
  - Plot running time versus input size
    - can also plot as a log-log plot
      - may yield straight line - get the power
  - Regression will yield run time - get hypothesis
- Effects on runtime
  - what algorithm is being used
  - Input data
- System dependent effects
  - hardware
  - software
  - system

Week 1- Analysis of Algorithms- Mathematical Models

June 4, 2017 1:12 PM

Total running time = sum of cost x frequency for all operations

- cost depends on machine
- frequency depends on algorithm

In principle accurate mathematical models are plausible

Cost Table - from slides

operation	example	nanoseconds †
integer add	a + b	2.1
integer multiply	a * b	2.4
integer divide	a / b	5.4
floating-point add	a + b	4.6
floating-point multiply	a * b	4.2
floating-point divide	a / b	13.5
sine	Math.sin(theta)	91.3
arctangent	Math.atan2(y, x)	129.0
...	...	...

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM

Screen clipping taken: 04/06/2017 1:16 PM

operation	example	nanoseconds †
variable declaration	`int a`	c_1
assignment statement	`a = b`	c_2
integer compare	`a < b`	c_3
array element access	`a[i]`	c_4
array length	`a.length`	c_5
1D array allocation	`new int[N]`	c_6 N
2D array allocation	`new int[N][N]`	c_7 N^2
string length	`s.length()`	c_8
substring extraction	`s.substring(N/2, N)`	c_9
string concatenation	`s + t`	c_{10} N

Screen clipping taken: 04/06/2017 1:16 PM

- made from experiments
- some constant
- some operations are proportional to N

Example : 0 sum problem

Q. How many instructions as a function of input size N?

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

operation	frequency
variable declaration	2
assignment statement	2
less than compare	N + 1
equal to compare	N
array access	N
increment	N to 2 N

Screen clipping taken: 04/06/2017 1:19 PM

- may be tedious to get exact counts as algorithms get more complicated

Simplification 1
- count the ones which are most expensive
    - most expensive or most frequent

Simplification 2
- ignore low order terms
- only the highest order in a polynomial counts
- as N --> infinity low order terms are negligible

Estimating a discrete sum
- replace sum with integral

Ex 1. $1 + 2 + \ldots + N.$

$$\sum_{i=1}^{N} i \sim \int_{x=1}^{N} x\,dx \sim \frac{1}{2} N^2$$

Ex 2. $1^k + 2^k + \ldots + N^k.$

$$\sum_{i=1}^{N} i^k \sim \int_{x=1}^{N} x^k dx \sim \frac{1}{k+1} N^{k+1}$$

Ex 3. $1 + 1/2 + 1/3 + \ldots + 1/N.$

$$\sum_{i=1}^{N} \frac{1}{i} \sim \int_{x=1}^{N} \frac{1}{x} dx = \ln N$$

Ex 4. 3-sum triple loop.

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

Screen clipping taken: 04/06/2017 1:27 PM

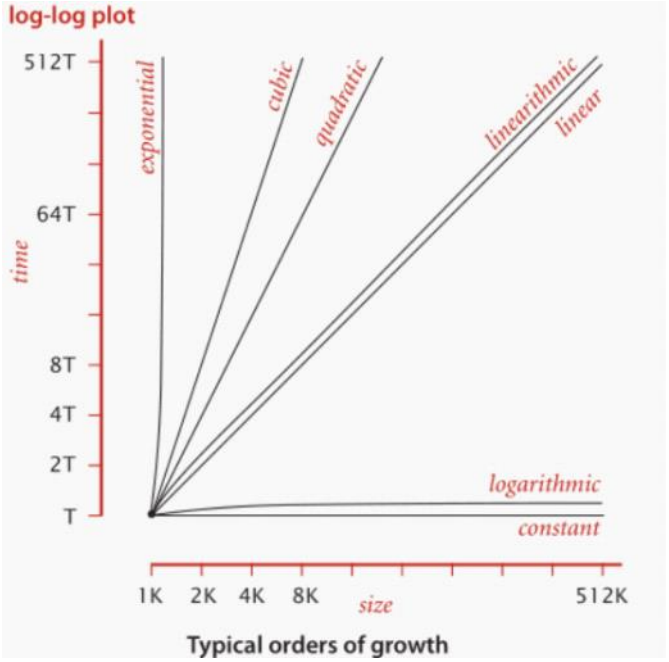
In theory precise mathematical models are possible but practically, approximations are better

Week- Analysis of Algorithms- Order of Growth Classifications

June 4, 2017 1:29 PM

Not many different fucntions in algorithms

- only few functions turn up
- log N, N NlogN, N^2, N^3, N^3, 2^N



Screen clipping taken: 04/06/2017 1:36 PM

- no leading constant

Make sure no algorithms are quadratic or higher

order of growth	name	typical code framework	description	example	T(2N) / T(N)
1	constant	`a = b + c;`	statement	add two numbers	1
log N	logarithmic	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search	~ 1
N	linear	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum	2
N log N	linearithmic	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	quadratic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }`	double loop	check all pairs	4
N^3	cubic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }`	triple loop	check all triples	8
2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	T(N)

Screen clipping taken: 04/06/2017 1:39 PM

Binary Search

- sorted array
- given a key - find the key in the array

- compare key against middle entry
  - too small go left
  - too big go right

```
public static int binarySearch(int[] a, int key)
{
   int lo = 0, hi = a.length-1;
   while (lo <= hi)
   {
       int mid = lo + (hi - lo) / 2;
       if      (key < a[mid]) hi = mid - 1;
       else if (key > a[mid]) lo = mid + 1;
       else return mid;
   }
   return -1;
}
```

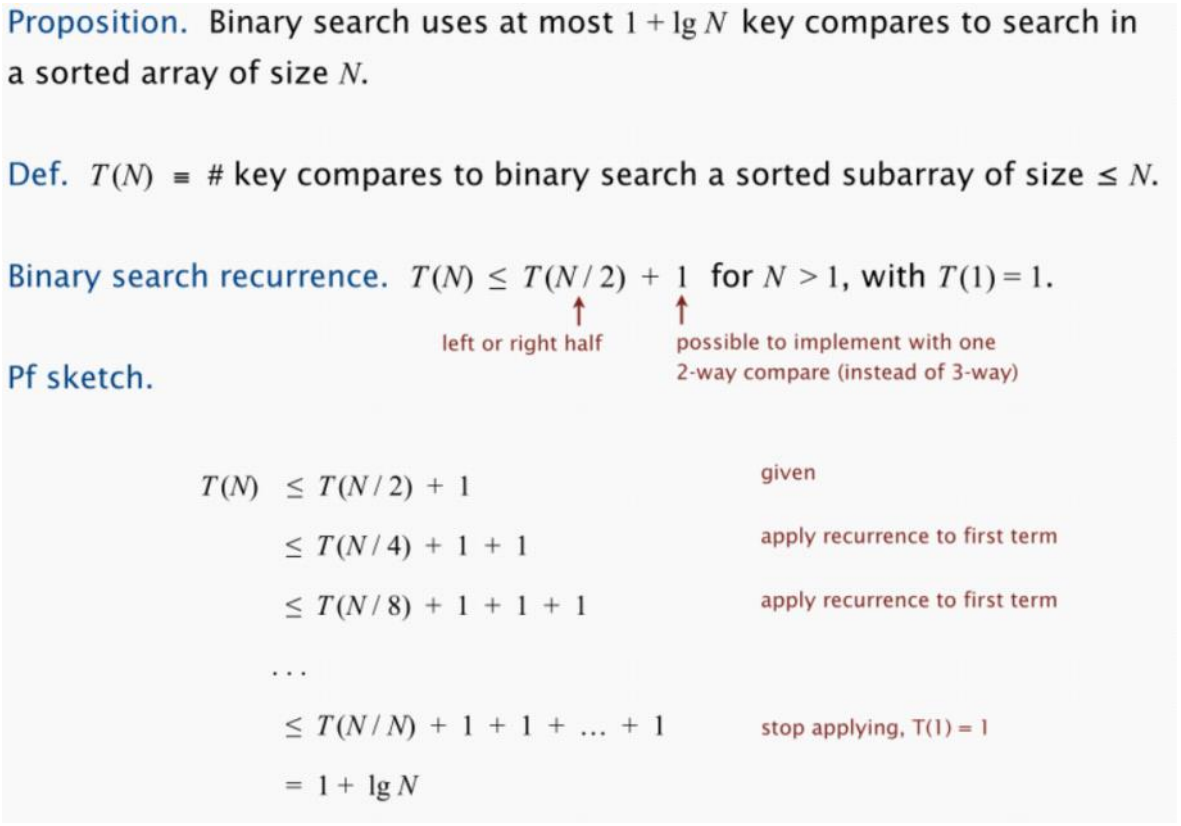
one "3-way compare"

Invariant. If key appears in the array a[], then a[lo] ≤ key ≤ a[hi].

Screen clipping taken: 04/06/2017 1:45 PM

Since binary search is recursive, the relation is also recurrent

- telescoping



Screen clipping taken: 04/06/2017 2:50 PM

- only holds if N is a power of 2.
  - can still prove its logarithmic

Better 3 sum algorithm

- sort array
- each pair of numbers - binary search -(a[i] + a[j])
- Makes it N^2 logN

yields faster programs

- want faster order of growth

Week 1- Analysis of Algorithms- Theory of Algorithms

June 4, 2017 3:18 PM

Type of Analyses

Best Case	Worst Case	Average
easiest input	most difficult input	random input
goal for all inputs	guarantee for all inputs	way to predict performance

Can either:

- design for the worst case
  - ideal
- randomize
  - depend on probability

Theory of Algorithms

- have a problem
- want to know the difficulty of algorithm

Approach

- supress details
  - get to a common factor (one term)
  - focus on worst case design

Optimal Algorithm

- can guarantee certain performance for any input
- no algorithm can provide a better performance

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

- Tilde - provide approximate model
    - will focus on tilde in this course

Example 1- One Sum

Goals
- establish difficulty
- develop the optimal algorithm

Upper bound (O(N))
- brute force
- provides upper bound for run time
- every array entry

Lower bound (Ω(N))
- every array entry
    - need to check every square

Therefore, the optimal algorithm has to be the same of upper bound or lower bound

Brute force algorithm for one sum is optimal.

Example 2 - Three sum

- no one knows the optimal algorithm for 3 sum
- upper bound is known but not lower bound

Design Approach
- develop algorithm
- prove lower bound
    - difficult for complex algorithms

- lower upper bound (new algorithm)
- raise lower bound (difficult for complex problems
    - difficult to prove

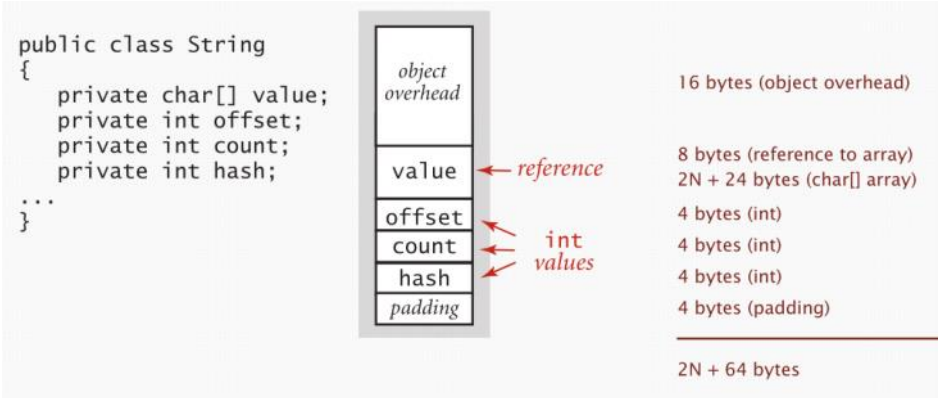
Week 1- Analysis of Algorithms- Memory

June 5, 2017 9:51 PM

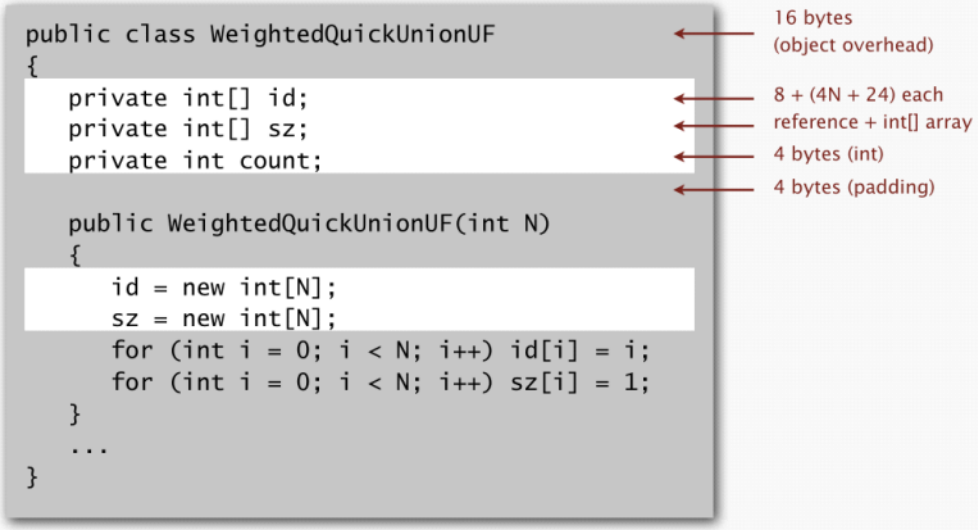
Current machines are 64 bit with 8 byte pointers

Memory usage of primitive types

Type	Bytes
boolean	1
byte	1
char	2
int	4
float	4
long	8
double	8
char[]	2N+24
int[]	4N+24
double[]	8N +24
char[]	~2MN
int[][]	~4MN
double[][]	~8MN
Object overhead	16
Reference	8
Padding	8
String	~2N



Q. How much memory does `WeightedQuickUnionUF` use as a function of N? Use tilde notation to simplify your answer.



A. $8N + 88 \sim 8N$ bytes.

Screen clipping taken: 05/06/2017 10:00 PM

Week 2- Introduction

Tuesday, June 6, 2017 10:15 AM

Fundamental data types

- Bags ques stacks

>collection of objects that we want to maintain

- Operations
  - Insert
  - Remove
  - Iterate
  - Test if empty
- Stack
  - Takes out the item that was most recently added
  - Last in first out

- Queue
  - Takes out item least recently added
  - First in first out

Modular Programming

- Completely separate interface and implementation
- Many clients can use the same implementations
  - Modular reusable libraries

Week 2- Stacks

Tuesday, June 6, 2017 10:20 AM

1st Implementations

- Linked lists
  - Consists of nodes with strings and references to the next string
  - Push operation
    - New node at the beginning
  - Pop
    - Remove first node

Use inner class:

- Stack class will contain node objects,
  - Contains the string and the reference for another node

- Pop operation
  - Return first item
  - Make the current first node the reference from the returned first item

- Push operation
  - Add a new node at the beginning
  - Make node equal to the old first node
    - This will become the second node
  - Create new node
  - Set the reference to the second node

```
public class LinkedStackOfStrings
{
   private Node first = null;

   private class Node
   {
      String item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(String item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public String pop()
   {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

private inner class
(access modifiers don't matter)

Screen clipping taken: 16/06/2017 6:35 PM

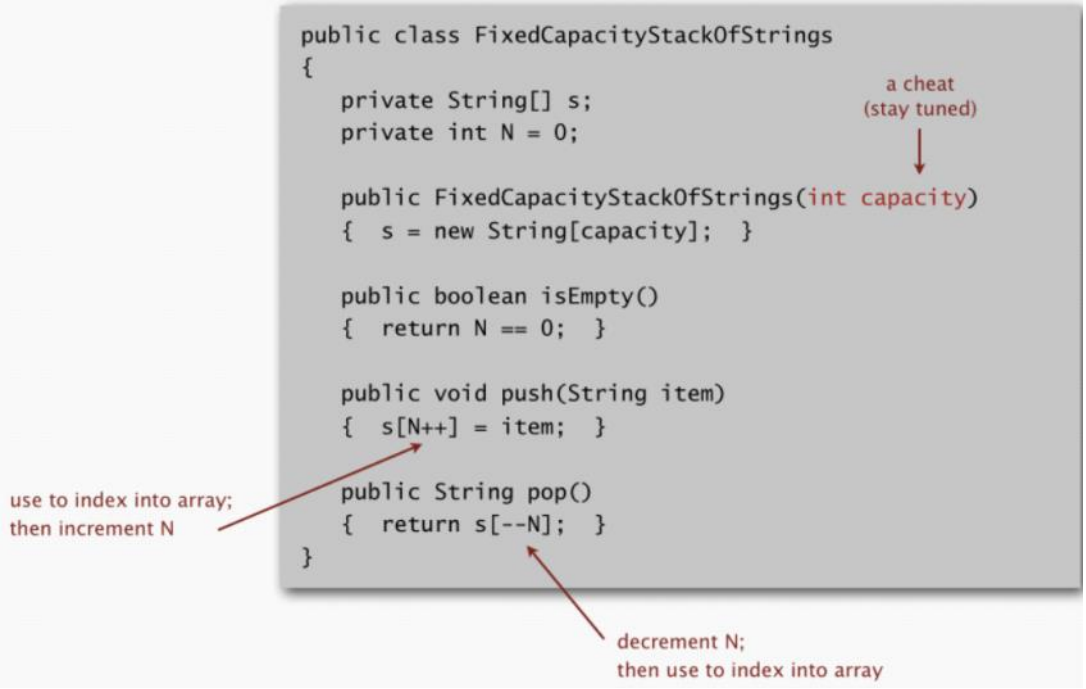
- Every operation takes constant time in the worst case
- Memory
  - ~40N BYTES

2nd Implementation

- Array structure
  - Stores N items on stack

- Pop operation
  - Remove string at N
  - Decrement N

- Push operation
  - Add string at N
  - Increment N



Screen clipping taken: 16/06/2017 6:36 PM

Stack Considerations

- Pop from empty stack
- Use resizing array for array implenatation

- We allow clients to insert null items

- Loitering
  - Have references to something we aren't using
  - When popping objects, make the popped object null
    - Memory considerations
    -

Week 2- Resizing Arrays

Wednesday, June 7, 2017 11:08 AM

We need a way such that the client does not need to provide the size of the array when initializing the stack

We could resize the array when the client pushes or pops from the stack

- Too expensive
- Need to copy the entire array into the new array
  - Time ~N^2/2

How to grow array?

- If the array is full, create a new array twice the size

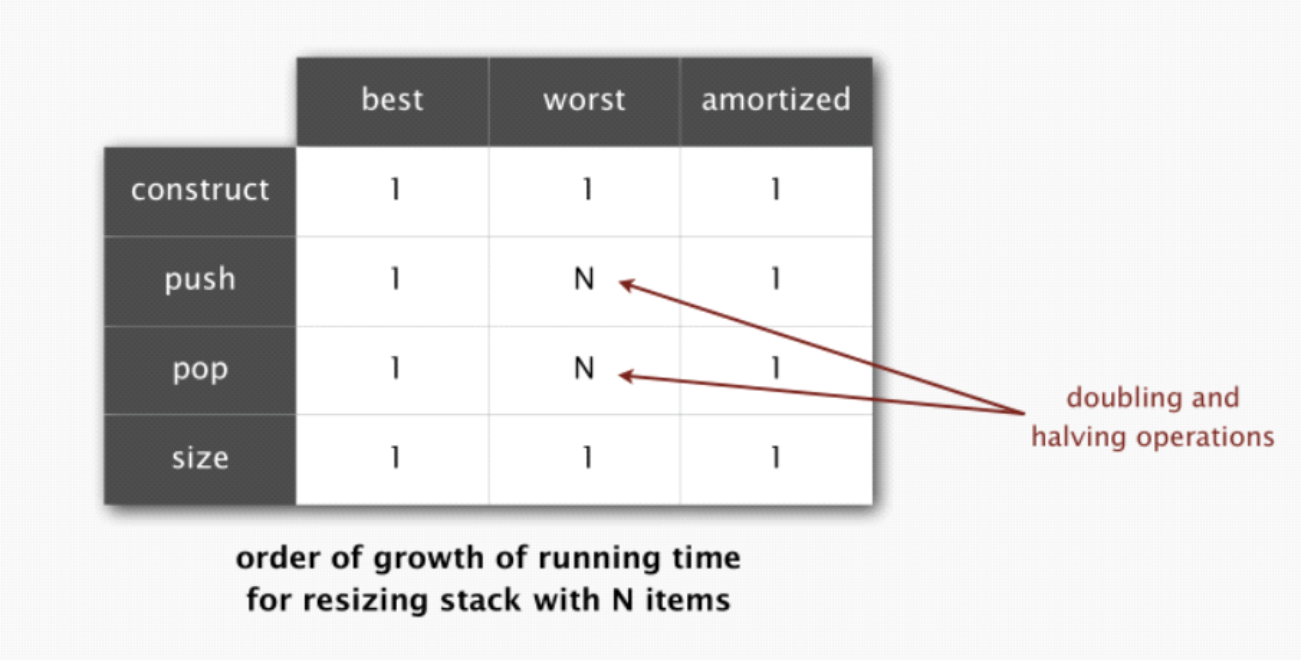
- Proportional to 3N not 1/2N^2

How to shrink array?

- Don't want array to get too empty
- Half the array every time it gets less than half
- Thrashing
  - If the client push pop push pop
  - Will half the array and then double
  - Quadratic time

- Half the array everytime the array is **one quarter** full

Performance



Screen clipping taken: 16/06/2017 6:37 PM

Resizing array vs Linked List

Resizing Array

- Every operation takes amortized constant time
  - Long time when resizing
- Less wasted space

Linked List

- Every operation takes constant time in the worst case
- Extra time and space to deal with links
- slower

Week 2- Queues

Wednesday, June 7, 2017 11:25 AM

Using the same API as stacks – but different names

Linked List

Need to maintain 2 pointers

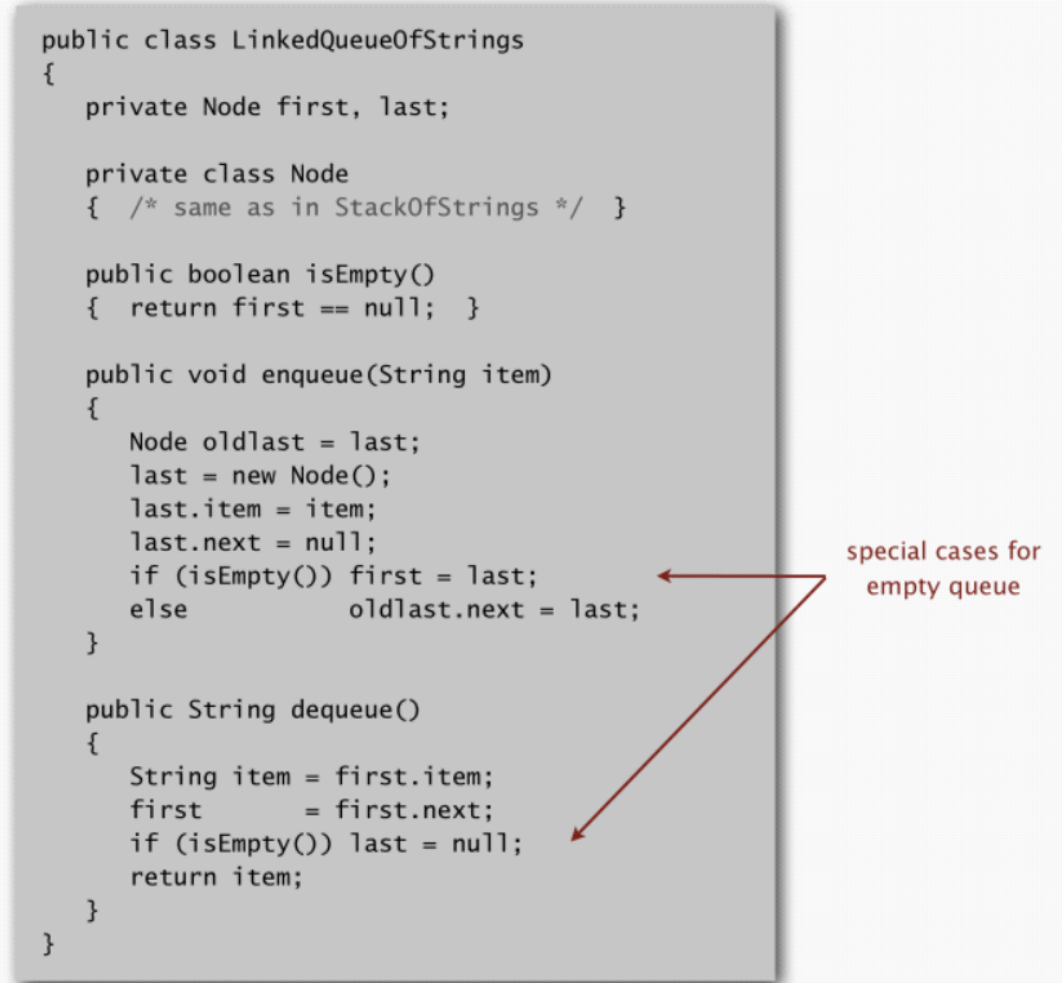
- One to the beginning and one to the end

Dequeue

- Same as stack

Enqueue

- Copy old node
- Create new node with new string and null node
- Make old node's pointer the new node



Screen clipping taken: 16/06/2017 6:38 PM

- If empty then make last node equal to first node

Array Implementation

- Have two pointers: head and tail
  - Insert new nodes at tail
  - Remove nodes at head
- When the array resize, make head at zero

Week 2 – Generics

Wednesday, June 7, 2017 11:34 AM

Implementations so far have been only good for strings

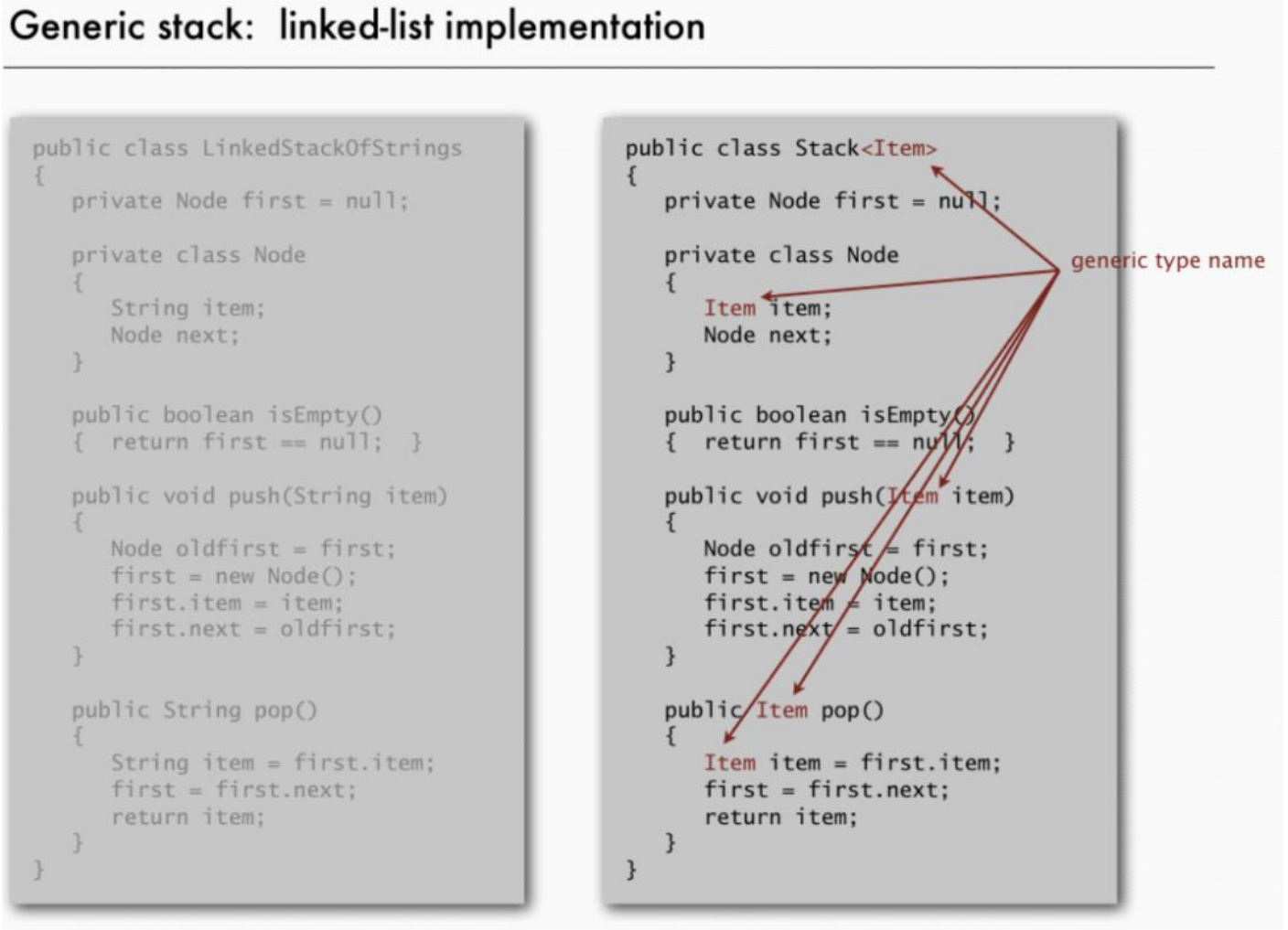
Attempt one – old approach
- Implement a separate stack class for each type
- Maintaining is difficult

Attempt 2 –
- Use casting
- Make the stack of object
  - Cast the final answer when the client calls for it
- Client's code has to do it
  - Unsatisfactory

Attempt 3- Generics
- Avoid casting in client
- Discover mismatch errors at compile time

- Replace string with generic type name

Linked List:



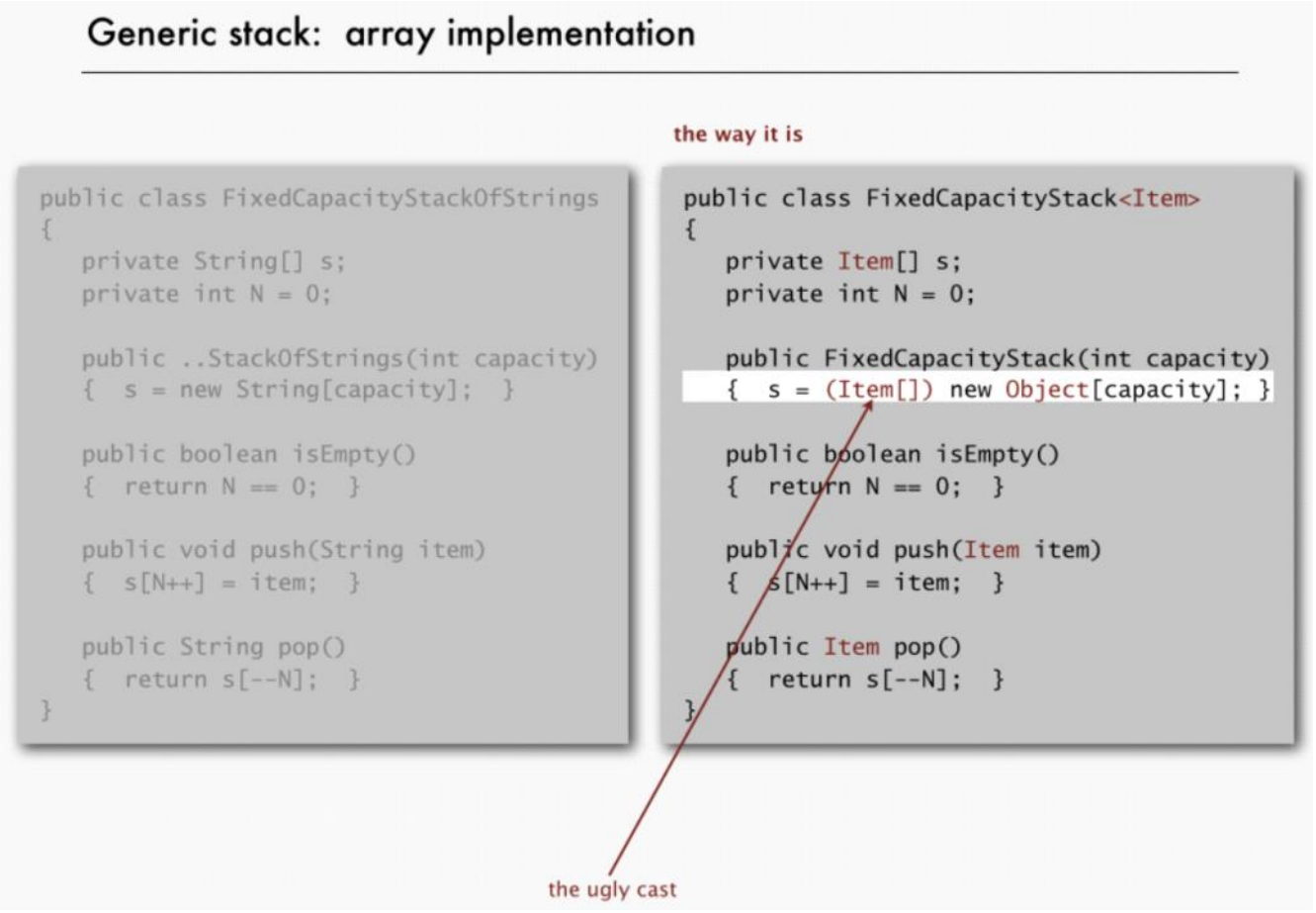
Screen clipping taken: 16/06/2017 6:44 PM

Arrays:

- More difficult than linked lists
- We could make generic arrays
  - Java does not allow this
- Create an array of objects
  - Casts these objects to items

(Casting is bad form but in this case, no choice)

- Undesirable feature



Screen clipping taken: 16/06/2017 6:45 PM

- When compiling, will get a warning for unchecked or unsafe operation
  - This is because of the uncheck casts

Primitive types

- We need to use Java's wrapper type
  - Integer is wrapper type for int

- Autoboxing – automatic cast between primitive type and its wrapper

Week 2- Iterators

Wednesday, June 7, 2017 11:44 AM

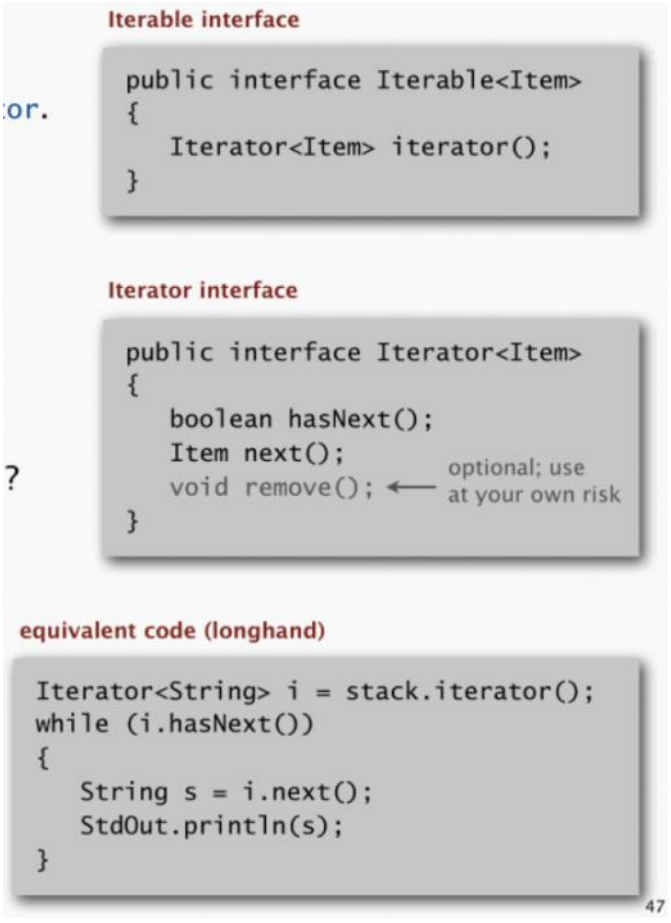
- Makes client code very elegant and compact

- Allow the client to iterate through the objects in the stack
  - Don't want them to know if we're using a linked list or array

Iteration
- A class that has a method that returns an iterator

Iterator
- Has methods hasNext() and next() - remove() is bad

If a data structure is inerrable, we can use for each statement



Screen clipping taken: 16/06/2017 6:50 PM

Linked List Iteration

Implements iterable
- Means that has a method that returns an iterable
- Has to implement hasNext() and next()

HasNext()
- If we're done returns false
- If not returns true

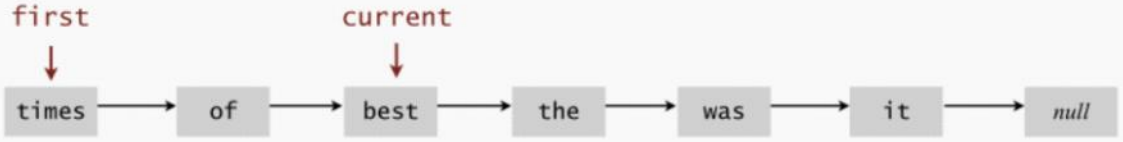
Next()
- Give next item in iteration

Stack iterator: linked-list implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator() { return new ListIterator(); }

    private class ListIterator implements Iterator<Item>
    {
        private Node current = first;

        public boolean hasNext() {  return current != null;  }
        public void remove()     {  /* not supported */      }
        public Item next()
        {
            Item item = current.item;
            current   = current.next;
            return item;
        }
    }
}
```

throw UnsupportedOperationException

throw NoSuchElementException if no more items in iteration



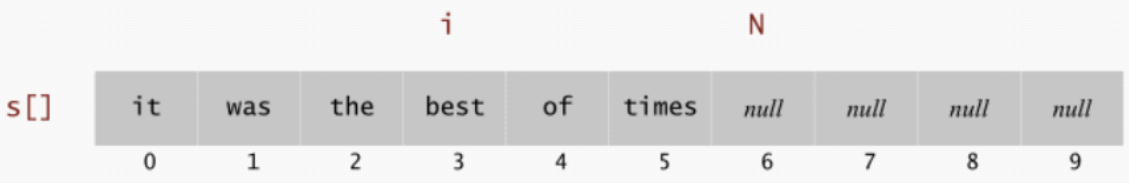
Screen clipping taken: 16/06/2017 6:50 PM

Array Iterator Implementation

- Goes from beginning of stack
  - Need to go from the end of the array to the beginning

Stack iterator: array implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    …
    public Iterator<Item> iterator()
    { return new ReverseArrayIterator(); }

    private class ReverseArrayIterator implements Iterator<Item>
    {
        private int i = N;

        public boolean hasNext() {  return i > 0;         }
        public void remove()     {  /* not supported */  }
        public Item next()       {  return s[--i];        }
    }
}
```



Screen clipping taken: 16/06/2017 6:51 PM

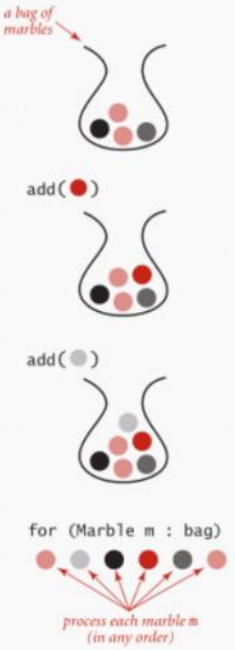
Data structure called a bag

- Adding items to a collection and iterating

Bag API

Main application. Adding items to a collection and iterating (when order doesn't matter).

public class Bag<Item> implements Iterable<Item>	
Bag()	*create an empty bag*
void add(Item x)	*insert a new item onto bag*
int size()	*number of items in bag*
Iterable<Item> iterator()	*iterator for all items in bag*



Implementation. Stack (without pop) or queue (without dequeue).

Screen clipping taken: 16/06/2017 6:50 PM

Week 2- Introduction to Sorts

June 17, 2017 2:24 PM

Rules to follow

Item = a record to the information that we are going to sort
A piece of the record is called the key

- this is the part that we sort

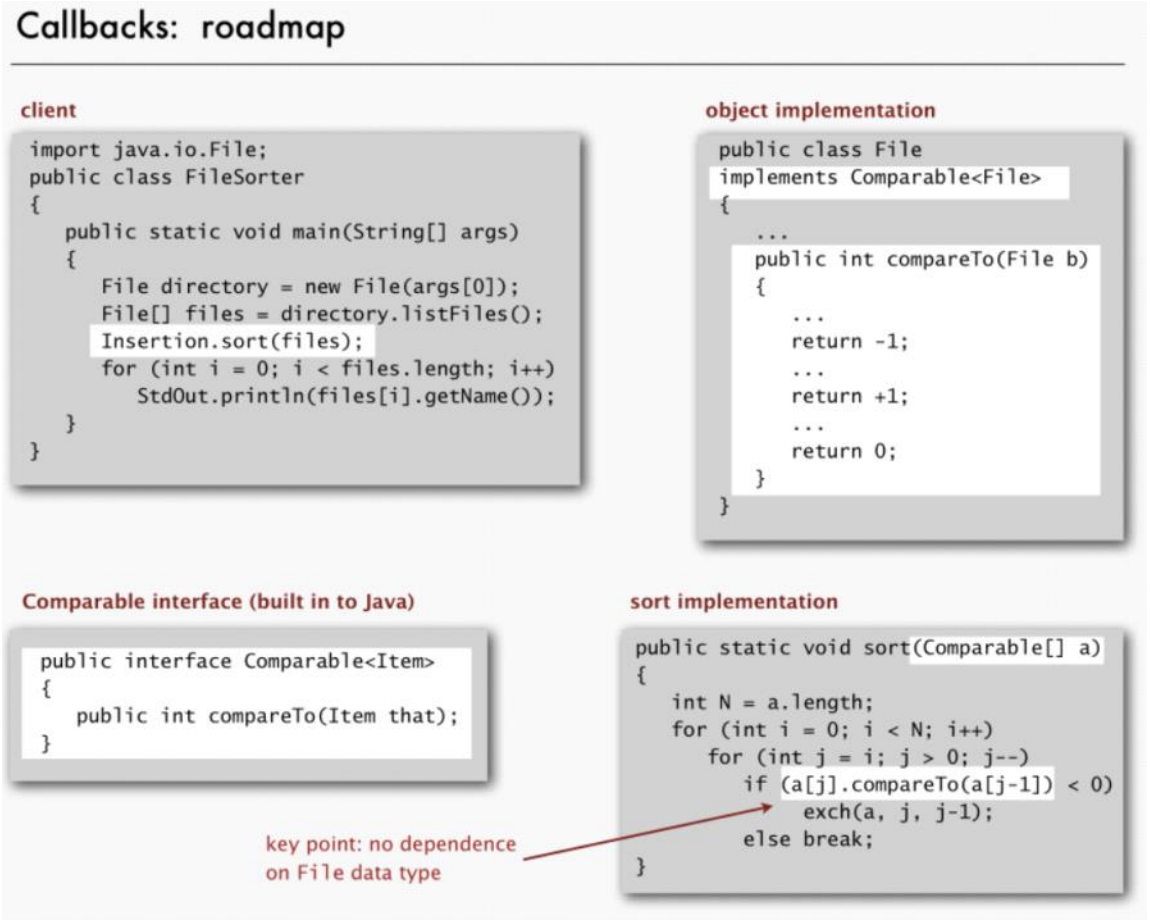
Goal - Sort any type of data

Callback

- How can sort() know how to compare data of double, string and java.io.file without any information about the type of the key
- call back is a reference to executable code
- the sort function calls back object's compareTo() method
    - the program compares code using the object's compare to method

Comparable Interface

- specification that a type that implements comparable will have a compareTo method



Screen clipping taken: 17/06/2017 6:09 PM

Total order

A total order is a binary relation $\leq$ that satisfies:

- Antisymmetry: if $v \leq w$ and $w \leq v$, then $v = w$.
- Transitivity: if $v \leq w$ and $w \leq x$, then $v \leq x$.
- Totality: either $v \leq w$ or $w \leq v$ or both.

Screen clipping taken: 17/06/2017 6:11 PM

Comparable API

implement compareTo so v.compareTo(w) is a total order

- if null returns exception

Built in

- integer
- double
- string

Make ourselves

- need to implement Comparable

```
public class Date implements Comparable<Date>
{
   private final int month, day, year;

   public Date(int m, int d, int y)
   {
      month = m;
      day   = d;
      year  = y;
   }

   public int compareTo(Date that)
   {
      if (this.year  < that.year ) return -1;
      if (this.year  > that.year ) return +1;
      if (this.month < that.month) return -1;
      if (this.month > that.month) return +1;
      if (this.day   < that.day  ) return -1;
      if (this.day   > that.day  ) return +1;
      return 0;
   }
}
```

only compare dates to other dates

Screen clipping taken: 17/06/2017 6:14 PM

Helper functions - refer to data throguh compares and exchanges - static
Only use these functions when sorting to ensure accuracy

- less
  - is v less than w
- Exchange
  - swamp item in a[] at i with j

Week 2 – Selection Sort

Wednesday, July 5, 2017 4:36 PM

Start out with unsorted array

- Find smallest remaining entry
- Swap a[i] and a[min]
- Increment i

Invariants – things that are true in the algorithm

- The entries to the left are in ascending order
- No entry to the right of the arrow are smaller than any value on the left side

Selection sort inner loop

To maintain algorithm invariants:

- Move the pointer to the right.

```
i++;
```

- Identify index of minimum entry on right.

```
int min = i;
for (int j = i+1; j < N; j++)
   if (less(a[j], a[min]))
      min = j;
```

- Exchange into position.

```
exch(a, i, min);
```

Screen clipping taken: 05/07/2017 10:00 PM

```
public class Selection
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
      {
         int min = i;
         for (int j = i+1; j < N; j++)
            if (less(a[j], a[min]))
               min = j;
         exch(a, i, min);
      }
   }

   private static boolean less(Comparable v, Comparable w)
   {  /* as before */  }

   private static void exch(Comparable[] a, int i, int j)
   {  /* as before */  }
}
```

Screen clipping taken: 05/07/2017 10:01 PM

Mathematical Analysis

- N^2/2 compares and N exchanges
- Quadratic time – even if input is sorted

Minimal data movement – minimum number of exchanges

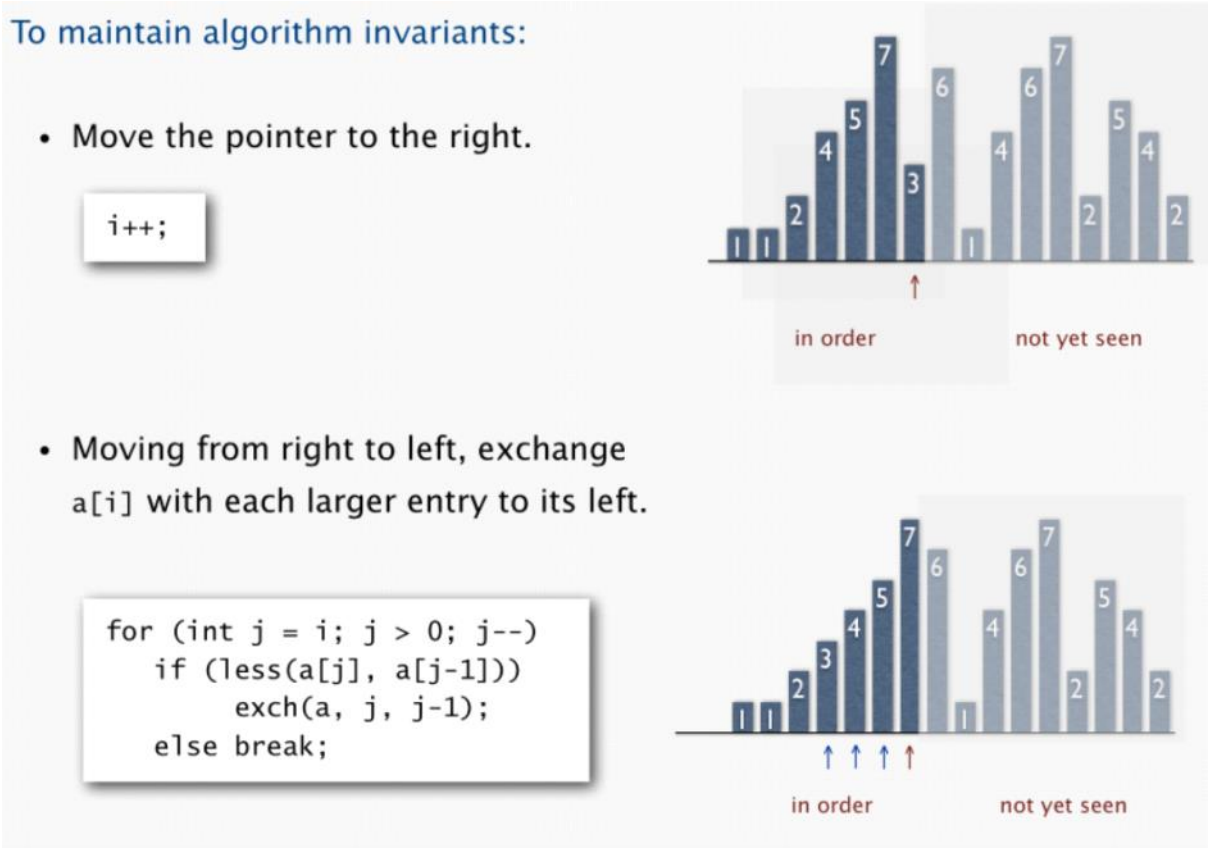
Week 2 – Insertion Sort

Wednesday, July 5, 2017 4:43 PM

Has two variables, j and i, exchange to the left until there is no next value or the card is less than j's value. (j is the position of the value being switched while i is the value of the first value.

Invariants

- Entries to the left of the arrow are sorted
- Entries to the right have not been seen



Screen clipping taken: 05/07/2017 10:01 PM

```
public class Insertion
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
         for (int j = i; j > 0; j--)
            if (less(a[j], a[j-1]))
               exch(a, j, j-1);
            else break;
   }

   private static boolean less(Comparable v, Comparable w)
   {  /* as before */  }

   private static void exch(Comparable[] a, int i, int j)
   {  /* as before */  }
}
```

Screen clipping taken: 05/07/2017 10:01 PM

Mathematical Analysis
1/4 N^2 compares
1/4 N^2 exchanges

- depends on how sorted the original array was sorted
- we can expect each value to be moved half the length of the array

Best Case
- already sorted
- N-1 compares
- 0 exchanges
- validates that the array is sorted

Worst Case
- In descending order
- every element goes all the way to the front
- 1/2 N^2 compares and exchanges.

Partially Sorted arrays
- an inversion
    - a pair of keys that are just switched
- if a number of inversions is < c*N where c is any constant
- Insertion time is linear time for partially sorted arrays

Week 2 - Shell sort

July 5, 2017 10:13 PM

With shell sort, we move entries more than one position at a time

h-sorting
- only sorts every h entries
- starts at entry h+1
    - compares with entries h entries back
    - repeats until array is combed through

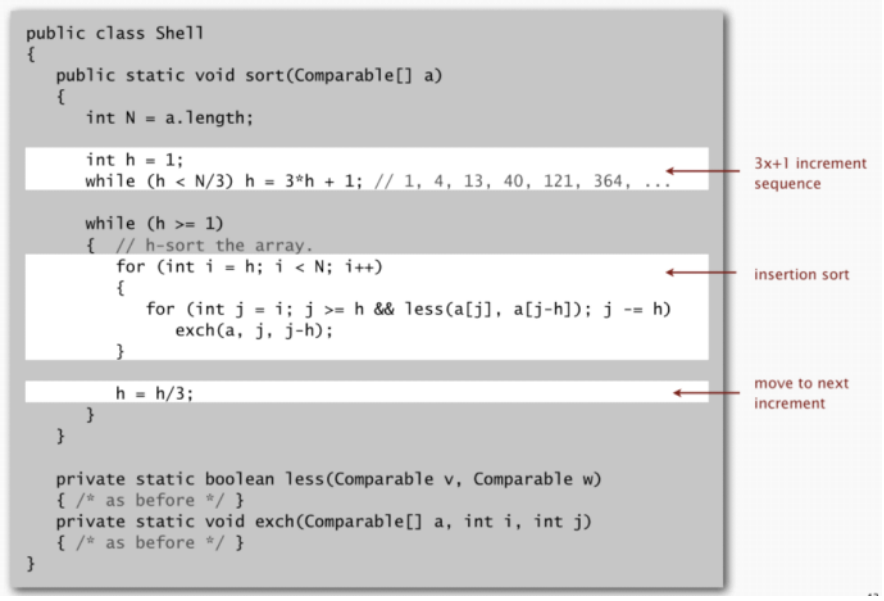
Use insertion sort, but instead of comparing with the adjacent entry, we compare with the h^{th} entry back

Big increment -> small subarray
Small increment -> array will be partially sorted
- can then use insertion sort

Use increments - ie (7,3,1)
- 1 sort is essentially insertion sort

If you g sort an array and then k sort,
- the final array will be both k sorted and g sorted

What increment system should we use?
- powers of 2?
    - does not compare even positions with odd positions
- powers of 2-1
    - ok
- 3x+1
    - easy to compute
- **we find the largest number that is smaller than the array size**



Screen clipping taken: 05/07/2017 10:24 PM

not much difference for partially sorted arrays

Mathematical Analysis
- worst case is O(N^3/2)
- nobody knows an accurate model for shellsort

Very useful in practice
- unless the array size is huge
- not a lot of code

Week 2 - Shuffling (Sorting application)

July 5, 2017 11:40 PM

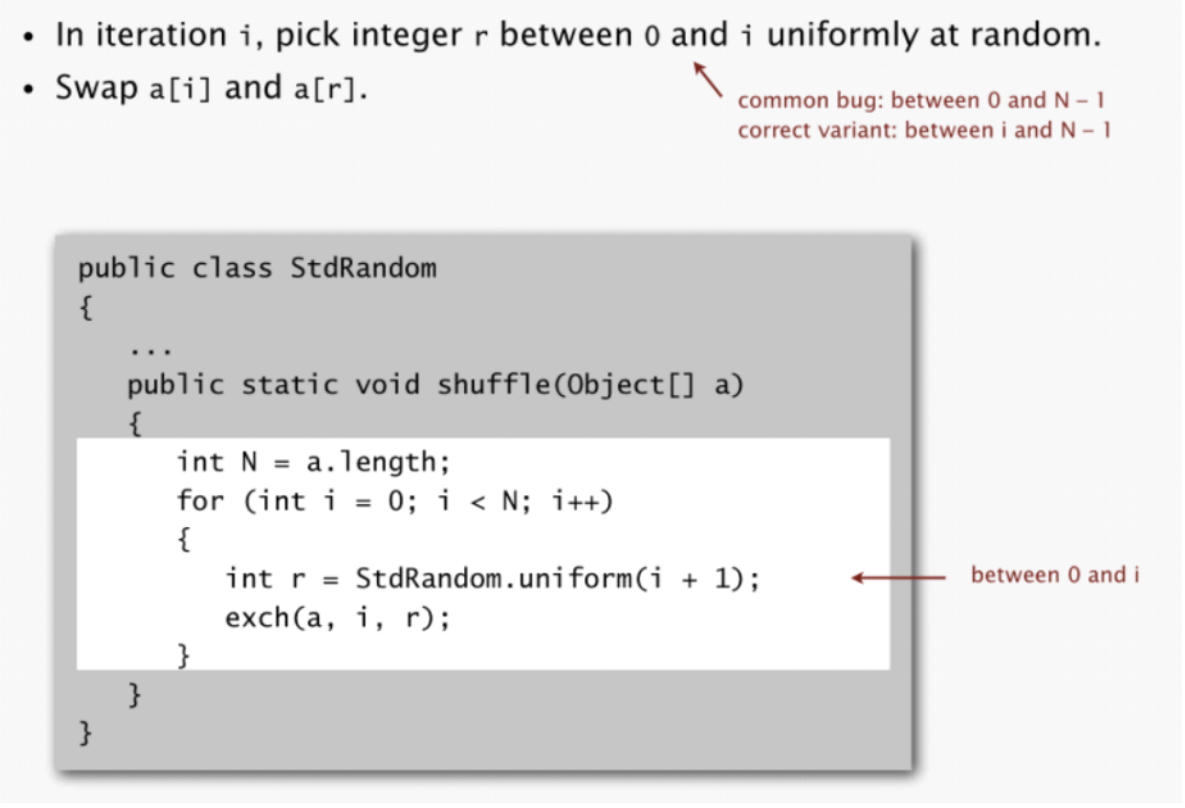
Application of sorting - shuffling

Generate a random real number for each entry

- sort the numbers

Can we have a faster way to shuffle?

- linear time
- pass through array - swap index at i with a random index r
- only swap cards with already seen indexes
- the values left of i are shuffled, right of i are not yet seen
    - Knuth Shuffle



Screen clipping taken: 06/07/2017 12:15 AM

Week 2 - Convex Hull (Sorting Application)

July 6, 2017 12:19 AM

For a set of points, the convex hull is the smallest perimeter that can enclose all points.

- vertices need to be points in the set

We want a program that will give us the convex hull of a set of points

- outputs a sequence of vertices in clockwise order

Convex hull application:

- find the shortest path from s to t such that it avoids the polygonal obstacle
- going around - will be the convex hull of the set of points

Second application,

- find the two sets of points furthest from each other
- both points will be on the convex hull

Properties of the convex hull

- can transverse convex hull by only making counter clockwise turns
- the vertices of convex hull appear in increasing order of polar angle with respect to the lowest y component

Graham Scan Algorithm

- choose point p with the smallest y coordinate
- sort points by polar angle
- consider points in order, discard unless it creates a counter clockwise turn.

Implementation Challenges

- point with smallest y coordinate
  - define total order - compare y component
- How to sort points by polar angle
  - define a total order for each point p
  - total order will be in the next lecture
- Determine if a CCW turn - next
- Sorting efficiently
  - merge sort sorts in NlogN time
  - a good sorting algorithm gives a good convex hull algorithm

- Multiple points on the same line

Implementing CCW turns

- how to determine if a turn is CCW?
- is c to the left of ray a-b
- look at slopes
  - special cases - collinear and infinity/0 slope
- Compute cross product
  - if vector is positive - CCW
  - if vector is negative - clockwise
  - if vector is 0, - they are collinear

Immutable point data type

```
public class Point2D
{
   private final double x;
   private final double y;

   public Point2D(double x, double y)
   {
      this.x = x;
      this.y = y;
   }

   ...

   public static int ccw(Point2D a, Point2D b, Point2D c)
   {
      double area2 = (b.x-a.x)*(c.y-a.y) - (b.y-a.y)*(c.x-a.x);
      if      (area2 < 0) return -1;  // clockwise
      else if (area2 > 0) return +1;  // counter-clockwise
      else                return  0;  // collinear
   }
}
```

danger of floating-point roundoff error

Screen clipping taken: 06/07/2017 12:34 AM

Graham scan: implementation

Simplifying assumptions. No three points on a line; at least 3 points.

```
Stack<Point2D> hull = new Stack<Point>();

Arrays.sort(p, Point2D.Y_ORDER);          ← p[0] is now point with lowest y-coordinate
Arrays.sort(p, p[0].BY_POLAR_ORDER);      ← sort by polar angle with respect to p[0]

hull.push(p[0]);  ← definitely on hull
hull.push(p[1]);

for (int i = 2; i < N; i++) {
   Point2D top = hull.pop();
   while (Point2D.ccw(hull.peek(), top, p[i]) <= 0)
      top = hull.pop();
   hull.push(top);
   hull.push(p[i]);  ← add p[i] to putative hull
}
```

discard points that would create clockwise turn

Running time. $N \log N$ for sorting and linear for rest.

Pf. $N \log N$ for sorting; each point pushed and popped at most once.

Screen clipping taken: 06/07/2017 12:35 AM

Week 2 - Programming Assignemnt

July 10, 2017 7:44 PM

For linked lists - need to bypass the intial null value

- when adding the first node, set first and last to that node
- when removing the last node, set first and last to null

When writing while loops

- what can cause the loop to run forever

Don't just copy their code

Week 3 - Merge Sort

July 11, 2017 1:41 PM

How to mergesort

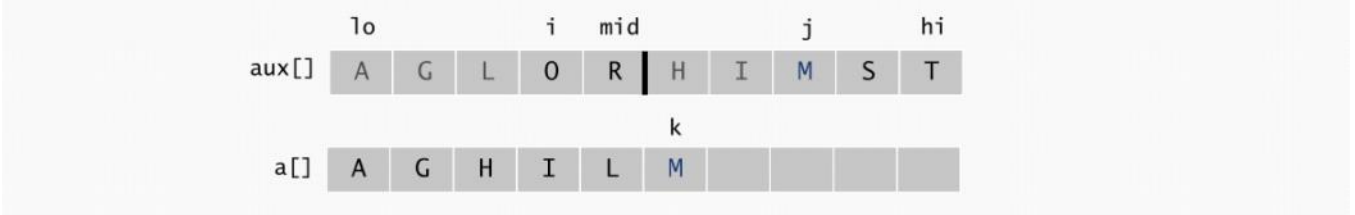
- divide array into two halves
- recursively sort into two halves
- merge two halves

To get two sorted halves into one array (abstract in-place merge)

- keep 3 indices, one at the first of the large array and 2 at the start of the smaller arrays
- compare the two indices in the sub arrays and insert the smallest one where the index in the large array is

Merging: Java implementation

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
   assert isSorted(a, lo, mid);    // precondition: a[lo..mid]   sorted
   assert isSorted(a, mid+1, hi);  // precondition: a[mid+1..hi] sorted

   for (int k = lo; k <= hi; k++)                                          copy
      aux[k] = a[k];

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)                                          merge
   {
      if      (i > mid)              a[k] = aux[j++];
      else if (j > hi)               a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                           a[k] = aux[i++];
   }

   assert isSorted(a, lo, hi);     // postcondition: a[lo..hi] sorted
}
```



Assertion

- helps detect logic bug
- test assumption about the program
- documents code

Assert statement

- throws exception unless boolean condition is true
  - assert is Sorted(a,lo,hi);

Can enable or disable at runtime

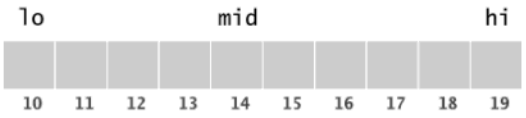
java -eq MyProgram // enable assertion

java -da MyProgram // disable assertion (default)

best practice - to test logic not to test inputs

Mergesort: Java implementation

```
public class Merge
{
   private static void merge(...)
   {  /* as before */  }

   private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
   {
      if (hi <= lo) return;
      int mid = lo + (hi - lo) / 2;
      sort(a, aux, lo, mid);
      sort(a, aux, mid+1, hi);
      merge(a, aux, lo, mid, hi);
   }

   public static void sort(Comparable[] a)
   {
      aux = new Comparable[a.length];
      sort(a, aux, 0, a.length - 1);
   }
}
```



Screen clipping taken: 11/07/2017 6:42 PM

- don’t create aux array in recursive method - takes too much memory

Running Time

- efficent algorithm
- NlogN time (insertion sort is N^2)

Proposition

- uses at most NlgN compares and 6NlgN array accesses with size N array

Pf Sketch (recurrence relation)

- the nuber of compares C(N) and array access A(N) to mergesort an array of size N

Mergesort: number of compares and array accesses

Proposition. Mergesort uses at most $N \lg N$ compares and $6 N \lg N$ array accesses to sort any array of size N.

Pf sketch. The number of compares $C(N)$ and array accesses $A(N)$ to mergesort an array of size N satisfy the recurrences:

$$C(N) \le C(\lceil N/2 \rceil) + C(\lfloor N/2 \rfloor) + N \quad \text{for } N > 1, \text{ with } C(1) = 0.$$

left half right half merge

$$A(N) \le A(\lceil N/2 \rceil) + A(\lfloor N/2 \rfloor) + 6N \quad \text{for } N > 1, \text{ with } A(1) = 0.$$

We solve the recurrence when N is a power of 2. ← result holds for all N

$$D(N) = 2\, D(N/2) + N, \text{ for } N > 1, \text{ with } D(1) = 0.$$

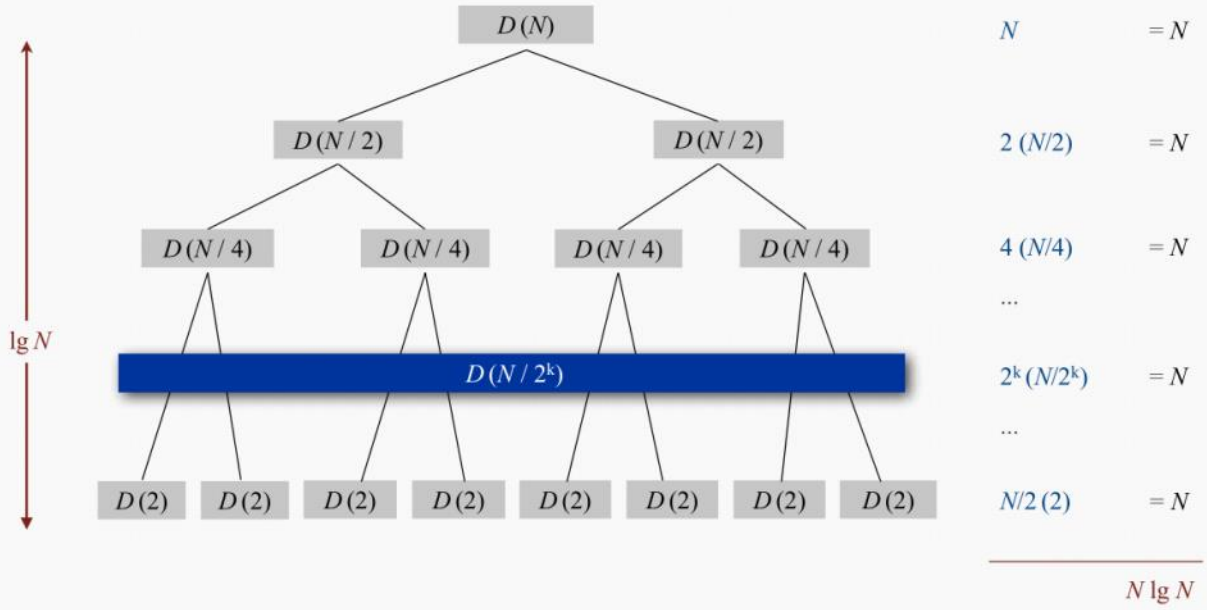
Screen clipping taken: 11/07/2017 6:42 PM

we can prove by induction

Divide-and-conquer recurrence: proof by picture

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 1. [assuming N is a power of 2]



Screen clipping taken: 11/07/2017 6:41 PM

- we can also prove by graphical analysis
  - there are lg N times you can split D(N)
  - every time yoou split it is N
  - therefore time takes NlgN

Divide-and-conquer recurrence: proof by expansion

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 2. [assuming N is a power of 2]

$D(N) = 2\,D(N/2) + N$	given
$D(N) / N = 2\,D(N/2) / N + 1$	divide both sides by N
$= D(N/2) / (N/2) + 1$	algebra
$= D(N/4) / (N/4) + 1 + 1$	apply to first term
$= D(N/8) / (N/8) + 1 + 1 + 1$	apply to first term again
$\ldots$	
$= D(N/N) / (N/N) + 1 + 1 + \ldots + 1$	stop applying, D(1) = 0
$= \lg N$	

Screen clipping taken: 11/07/2017 6:43 PM

Induction:

Divide-and-conquer recurrence: proof by induction

Proposition. If $D(N)$ satisfies $D(N) = 2\,D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 3. [assuming N is a power of 2]

- Base case: $N = 1$.
- Inductive hypothesis: $D(N) = N \lg N$.
- Goal: show that $D(2N) = (2N) \lg (2N)$.

$D(2N) = 2\,D(N) + 2N$	given
$= 2\,N \lg N + 2N$	inductive hypothesis
$= 2\,N\,(\lg (2N) - 1) + 2N$	algebra
$= 2\,N \lg (2N)$	QED

Screen clipping taken: 11/07/2017 6:43 PM

Memory
uses space proportional to N

- extra auxiliary array

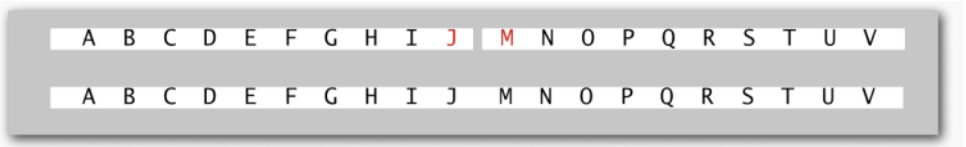
Practical Improvements

1. Use insertion sort for small subarrays
    a. too much overhead for recursive calls
    b. cutoff to insertion sort for 7 items (insertion sort is efficent for small arrays)

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo + CUTOFF - 1)
   {
      Insertion.sort(a, lo, hi);
      return;
   }
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   merge(a, aux, lo, mid, hi);
}
```

Screen clipping taken: 11/07/2017 6:51 PM

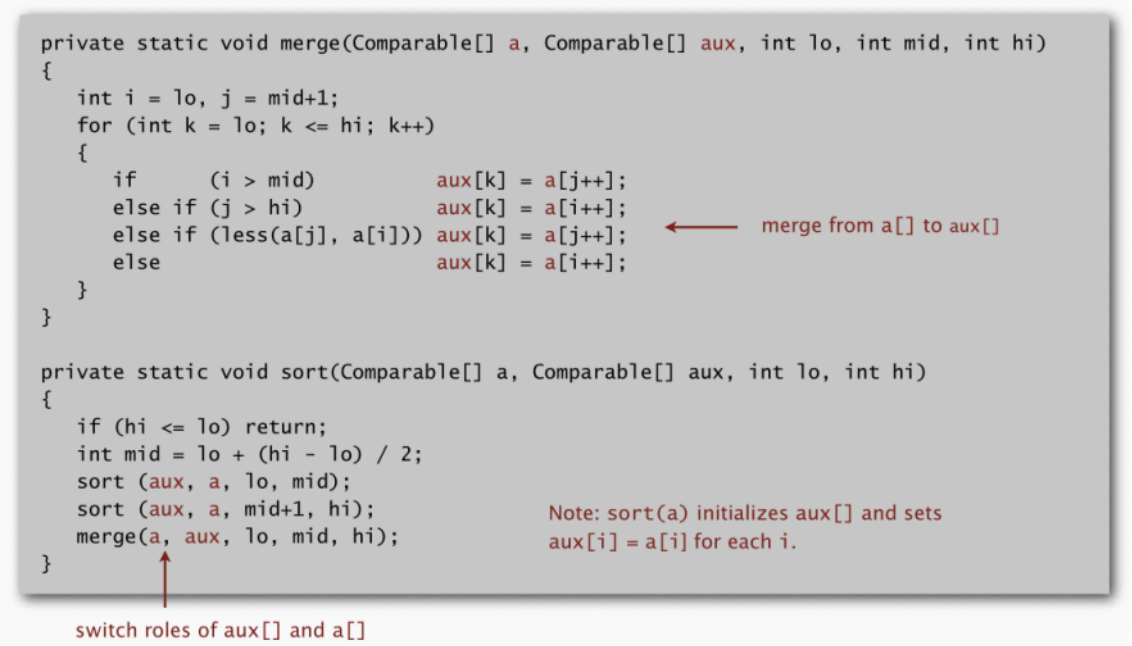
2. Stop if already sorted
    a. The biggest element in the first half is smaller than the smallest item in the second half




```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   if (!less(a[mid+1], a[mid])) return;
   merge(a, aux, lo, mid, hi);
}
```

Screen clipping taken: 11/07/2017 6:51 PM

3. Eliminate the copy to the aux array
    a. switch role of input and aux array in each recursive call



Screen clipping taken: 11/07/2017 6:52 PM

Week 3 - Bottom Up Mergesort

July 11, 2017 6:55 PM

Each array is subarrays of sorted arrays of size 1

- merge subarrays of size 1
- repeat for 2,4,8,16…..

No recursion needed

```
public class MergeBU
{
   private static void merge(...)
   {  /* as before */  }

   public static void sort(Comparable[] a)
   {
      int N = a.length;
      Comparable[] aux = new Comparable[N];
      for (int sz = 1; sz < N; sz = sz+sz)
         for (int lo = 0; lo < N-sz; lo += sz+sz)
            merge(a, aux, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
   }
}
```

Screen clipping taken: 11/07/2017 6:56 PM

- uses extra space proportional to size of array

Week 3 - Sorting Complexity

July 11, 2017 7:00 PM

Framework to study efficiency of algorithms

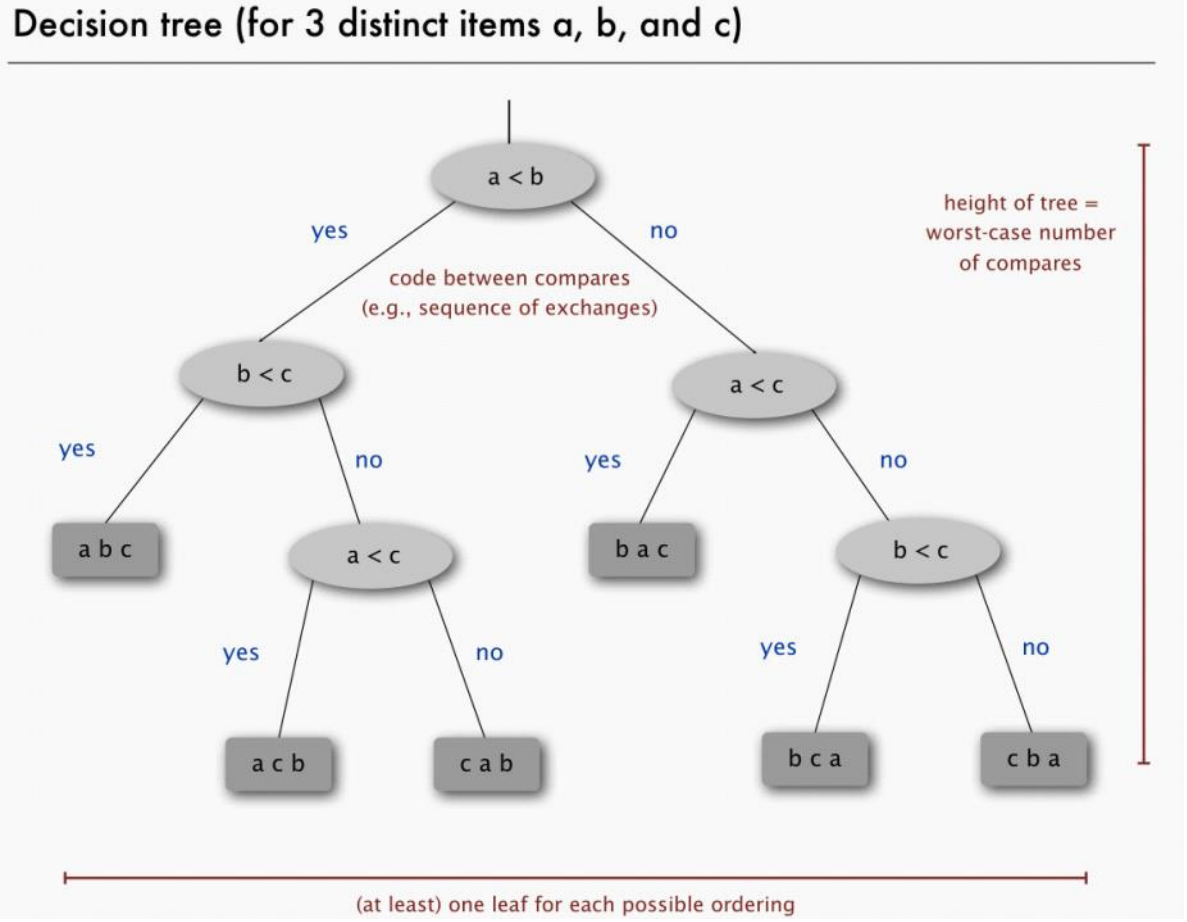
Model of Computation

- operations that the algorithm performs
- Cost model - operation costs
- Upper bound - guaranteed cost provided by some algorithm
  - known algorithm
- Lower bound - proven limit on cost guaranteed on all algorithms
  - no algorithm can do better
- Optimal algorithm
  - lower bound = upper bound

ie. Sorting

- decision tree
- all we can use are compares
- number of compares
- mergesort - NlgN

Decision Tree:

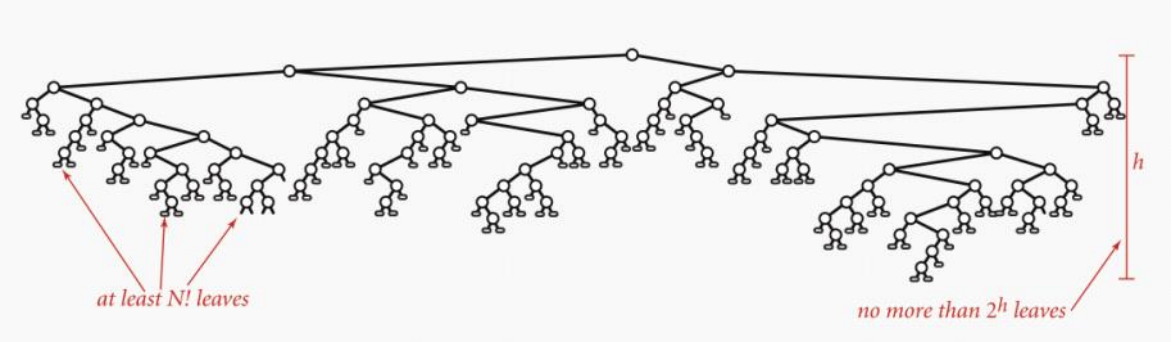


Screen clipping taken: 11/07/2017 9:23 PM

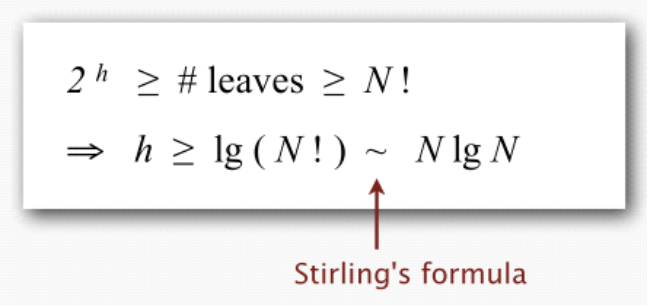
- Lower bound finds the lowest height of tree possible

Any compare-based sorting algorithm must use at least lg(N!) ~ NlgN compares

- binary tree of height h has 2^h
    - no more than 2^h
- at least N! leaves



Screen clipping taken: 11/07/2017 9:26 PM



Screen clipping taken: 11/07/2017 9:27 PM

Therefore the lower bound is NlgN

- mergesort is the optimal algorithm

Mergesort is optimal to number of compares

- not optimal for memory

Lessons from sorting complexity

- use theory as guide
    - don’t design an algorithm with 1/2 NlgN compares
        - won't work
- The lower bound is only with one case (using only compares)
    - ie. if the array is partially sorted insertion sort will be linear time

Week 3 - Comparators

July 11, 2017 9:33 PM

Java's way to compare different values

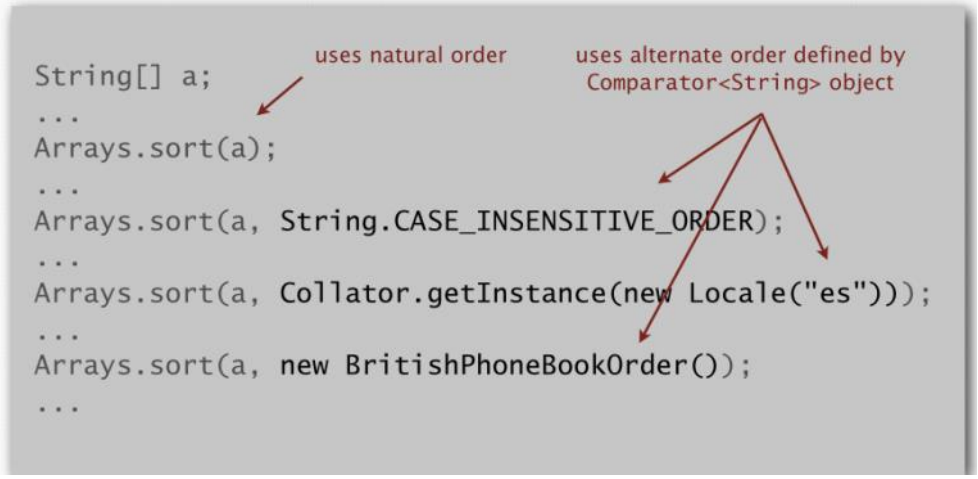
Comparable interface -> used before

Comparator interface - sort using an alternate order

- can have different keys
  - natural order
  - case insensitive
  - etc

Java system sort

- create comparator object
- Pass as second argument to Arrays.sort()



Screen clipping taken: 11/07/2017 10:00 PM

Defining how to sort data can be done separately from when you define the data type

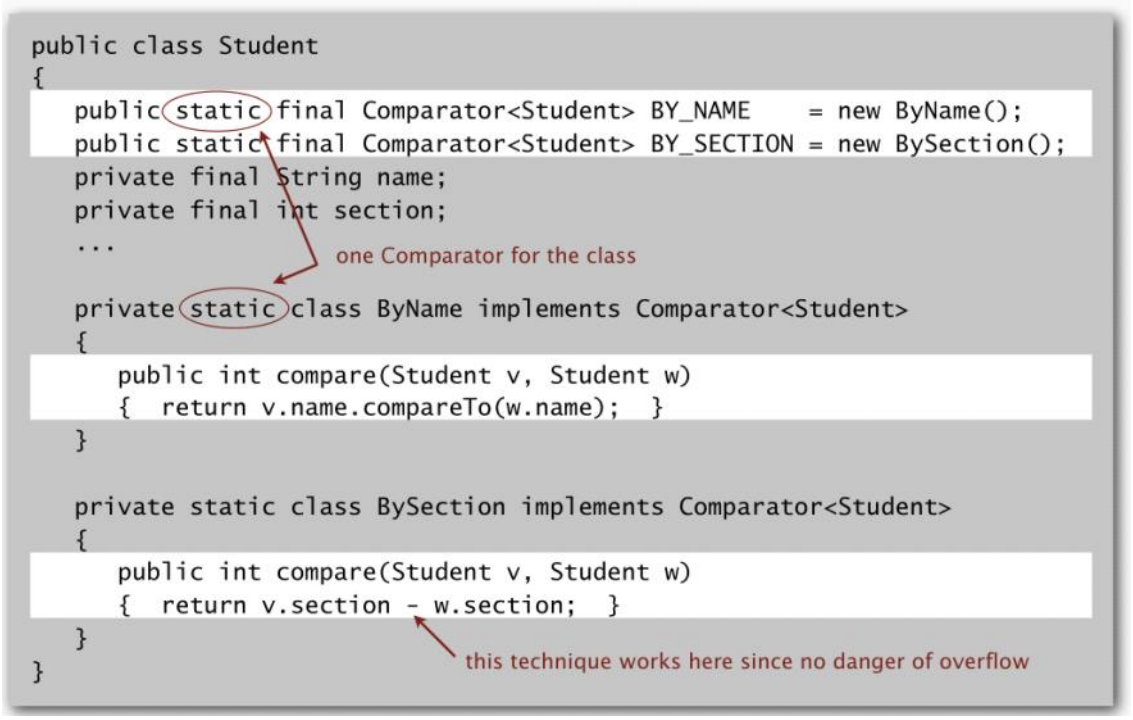
We can change our implementations to support comparators

- use object instead of Comparable
  - Object[] a instead of Comparable[] a
- pass Comparable to sort() and less() and use it in less()

```
public static void sort(Object[] a, Comparator comparator)
{
   int N = a.length;
   for (int i = 0; i < N; i++)
      for (int j = i; j > 0 && less(comparator, a[j], a[j-1]); j--)
         exch(a, j, j-1);
}

private static boolean less(Comparator c, Object v, Object w)
{  return c.compare(v, w) < 0;   }

private static void exch(Object[] a, int i, int j)
{  Object swap = a[i]; a[i] = a[j]; a[j] = swap;  }
```

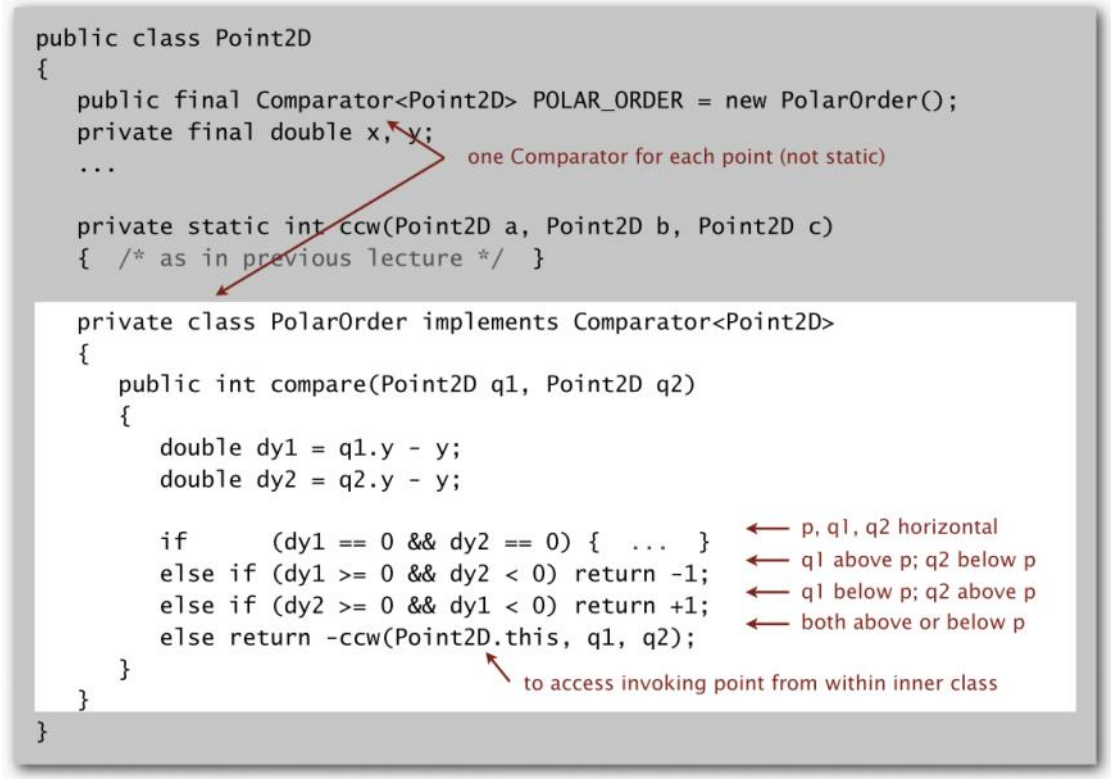
Screen clipping taken: 11/07/2017 10:04 PM

Implementing comparators



Screen clipping taken: 11/07/2017 10:04 PM

Example: Comparator for 2d points (CCW polar angles)



Screen clipping taken: 11/07/2017 10:09 PM

- the compare() method returns a positive value of obj 1 is greater than obj2

- 0 for equal
- negative for obj2 greater than obj1

Week 3 - Stability

July 11, 2017 10:14 PM

A stable sort preserves the relative order of items with equal keys

Which sorts are stable?
- insertion
- merge

Not stable
- selection
- shell

What makes sorts stable?
- equal items never move past each other

Long distance exchanges often makes sorts unstable

Mergesort's stability depends on its merge() method
- if the two keys are equal will take from the lower array - items will remain sorted

Week 3 - Quicksort

July 12, 2017 11:51 AM

Quicksort is the Java sort for primitive data types

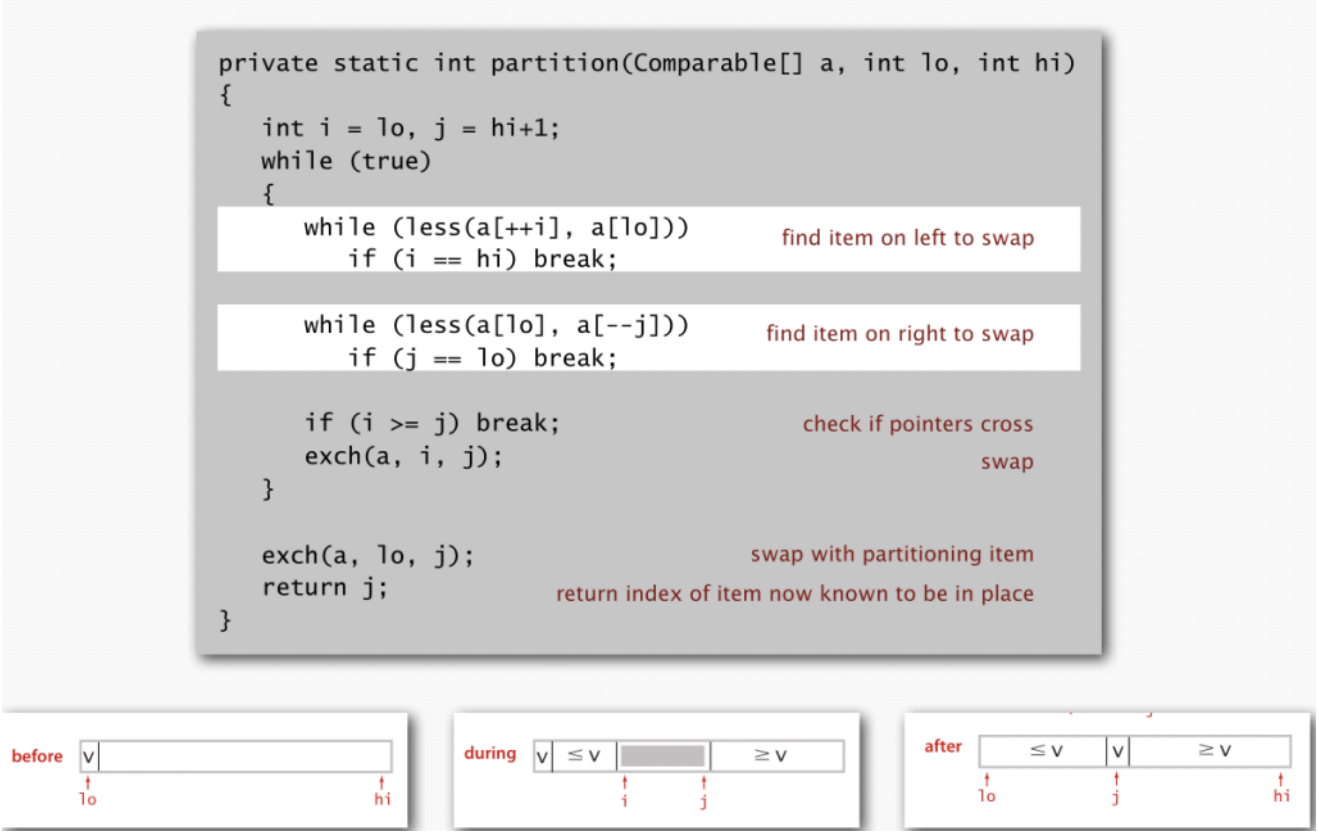
Quicksort

- Recursive method

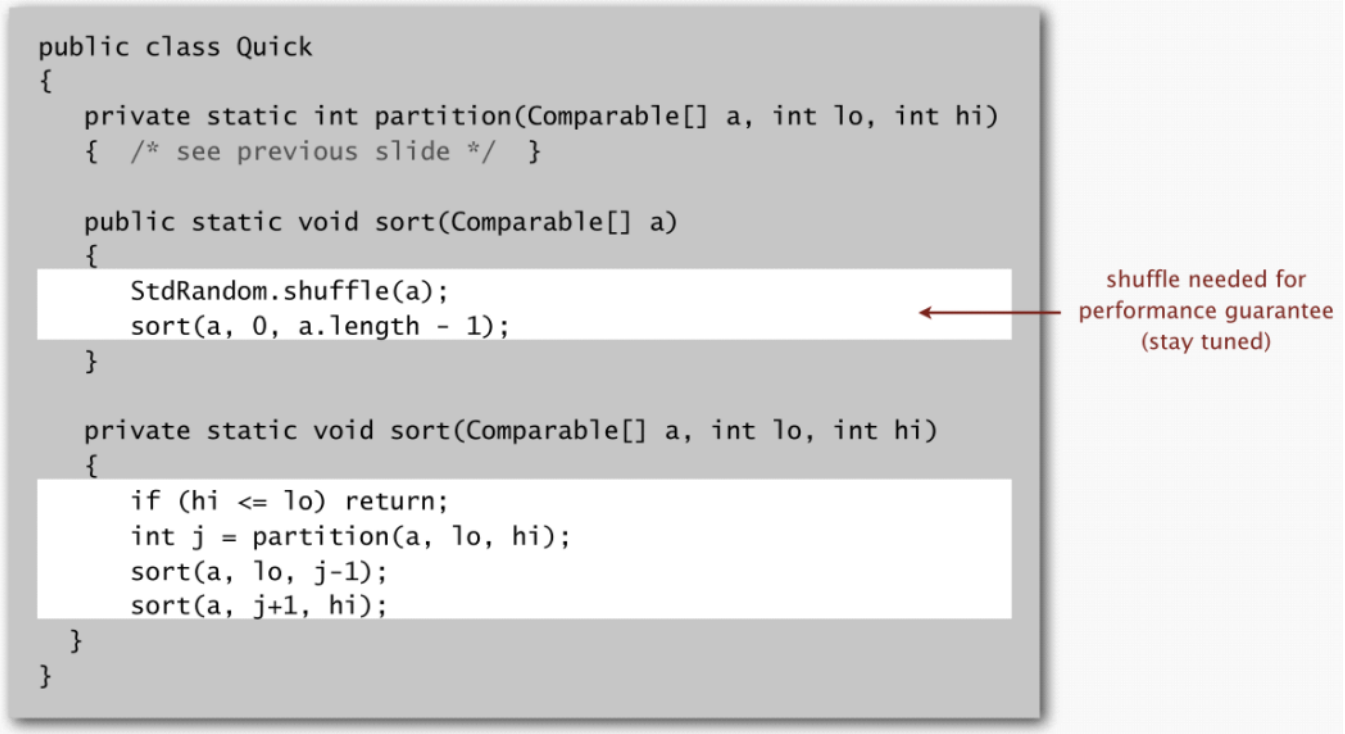
- shuffle the array
- partition such that a value will be in place
  - no value left of it is greater than the value
  - no value right of it is less than the value
- sort each side recursively

Demo

1. randomly select the partition
2. have 2 pointers i,j
   a. scan from left to right for i
      i. stop if the value is greater than the partition value
   b. scan from right to left for j
      i. stop if the value is smaller than the partition value
   c. when both i and j are stopped, exchange i and j
3. exchange j with the partition element



Screen clipping taken: 12/07/2017 2:54 PM



Screen clipping taken: 12/07/2017 2:56 PM

Implementation Details

- partitions in place - doesn't take extra space
- testing when pointers cross is trickier - duplicate keys
- (j == lo) is redundant - j will hit the partitioning element
- (i=hi) is not redundant - there is room for i to run off the array

Preserving randomness

- shuffling is needed for performance guarantee

Equal keys

- when the pointers hit values that are equal to the partition, they stop

Running Time

- faster than mergesort
- Best Case
    - divides everything by half
        - NlgN
- Worst Case
    - when the partition is the first value
        - 1/2 N^2
        - unlikely to happen
- Average case

Pf. C_N satisfies the recurrence $C_0 = C_1 = 0$ and for $N \geq 2$:

$$C_N = \underset{\text{partitioning}}{(N+1)} + \left(\frac{C_0 + C_{N-1}}{N}\right) + \left(\frac{\overset{\text{left}}{C_1} + \overset{\text{right}}{C_{N-2}}}{N}\right) + \ldots + \left(\frac{C_{N-1} + C_0}{N}\right)$$

partitioning probability

- Multiply both sides by N and collect terms:

$$NC_N = N(N+1) + 2(C_0 + C_1 + \ldots + C_{N-1})$$

- Subtract this from the same equation for $N - 1$:

$$NC_N - (N-1)C_{N-1} = 2N + 2C_{N-1}$$

- Rearrange terms and divide by $N(N+1)$:

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}$$

Screen clipping taken: 12/07/2017 3:03 PM

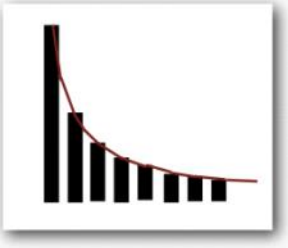
- Repeatedly apply above equation:

$$\begin{aligned}
\frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \quad \leftarrow \text{substitute previous equation} \\
&= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \ldots + \frac{2}{N+1}
\end{aligned}$$

previous equation

- Approximate sum by an integral:

$$\begin{aligned}
C_N &= 2(N+1)\left(\frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots \frac{1}{N+1}\right) \\
&\sim 2(N+1)\int_3^{N+1} \frac{1}{x}\,dx
\end{aligned}$$



- Finally, the desired result:

$$C_N \sim 2(N+1)\ln N \approx 1.39 N \lg N$$

Screen clipping taken: 12/07/2017 3:04 PM

- 39% more compares than mergesort
- faster because of less data movement
- shuffle guarantees against worst case
- easily understandable mathematically

Practical Improvements

- insertion sort for small subarrays
  - cutoff ~ 10 items (10-20 also works)

    - another option is to leave small subarrays until the end and then run a insertion sort on the sub arrays
- Take the median
    - sample the items
    - estimate median of samples
        - take the median of 3 random items

Week 3 - Selection

July 12, 2017 3:18 PM

Goal - given an array of N items - find the kth largest

Use theory as a guide

- we can solve selection in NlgN
  - sort the array - find the kth
- If k is small - proportional to N
  - for k = 0 just find min
  - for k = 1 need to run through twice
- Lower bound is N - must look at all items

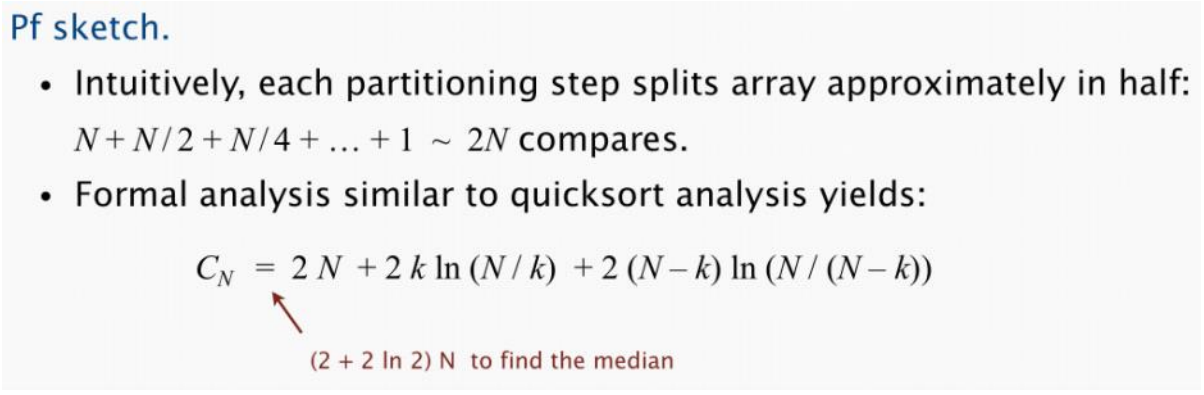
Quick-Select

Partition array such that

- a[j] is in place
- no entry smaller left
- no entry larger right
- repeat in one subarray depending on the values of k and j
  - if k is larger than k - only look at right subarray
  - if k is smaller than k - only look at left subarray
  - if equal - found the kth digit

Mathematical Analysis

- only takes linear time on average



Screen clipping taken: 12/07/2017 3:23 PM

Quick Select is only on average linear time

There is an algorithm that is worst case linear time

- takes too much memory to be practical
- still worthwhile to find a practical linear time algorithm

Week 3 - Duplicate Keys

July 12, 2017 3:26 PM

The purpose of sorting is often to bring keys of equal keys together

Quicksort

- goes quadratic time unless partition stops on equal keys

Mistake: put all items equal to partition items on one side

- if all the keys are equal
  - then the algorithm will become quadratic
- if you stop when keys are equal
  - divides the array in half
  - NlgN compares
- If put all items in place
  - desirable

3 way partitioning

- partition array into three parts
- 1 array smaller than partition
- 1 array equal to partition
- 1 array larger than partition

3 constants

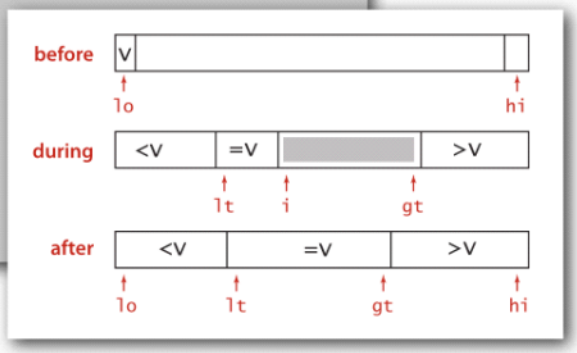
- lt - less than partitioning elemetn
- gt - greater than partitioning element
- i - everything right of i we haven't seen yet
- between lt and i - equal to v

Screen clipping taken: 12/07/2017 3:33 PM

- i scans from left tor ight
  - if i is less than partition, swap i and lt - increment i and lt
  - if i is greater than partition, swap i and gt - decrement gt
  - if i is equal to partition increment i
  - stop with i and gt cross

3-way quicksort: Java implementation

```
private static void sort(Comparable[] a, int lo, int hi)
{
   if (hi <= lo) return;
   int lt = lo, gt = hi;
   Comparable v = a[lo];
   int i = lo;
   while (i <= gt)
   {
      int cmp = a[i].compareTo(v);
      if      (cmp < 0) exch(a, lt++, i++);
      else if (cmp > 0) exch(a, i, gt--);
      else              i++;
   }

   sort(a, lo, lt - 1);
   sort(a, gt + 1, hi);
}
```



Screen clipping taken: 12/07/2017 3:37 PM

Sorting Lower Bound

Sorting lower bound. If there are n distinct keys and the i^{th} one occurs x_i times, any compare-based sorting algorithm must use at least

$$\lg \left(\frac{N!}{x_1!\, x_2! \cdots x_n!} \right) \sim -\sum_{i=1}^{n} x_i \lg \frac{x_i}{N}$$

$N \lg N$ when all distinct; linear when only a constant number of distinct keys

Screen clipping taken: 12/07/2017 3:39 PM

- number of compares is equal to lower bound
- depends on the distribution of keys
  - entropy optimal
- when using 3 way partitioning
  - many applications when run times become linear

Week 3 - System Sorts

July 12, 2017 3:41 PM

Applications

Java - arrays.sort()
- with primitive type
- implements comparable and comparator

Another implementation that does not require making the array random
- Tukey's ninther
  - Median of the median of 3 samples each of 3 entries
  - approximates the median of 9 values with only 12 compares
  - don’t like change of state and also less costly

What is the most important sorting algorithm?
- has diverse attributes
  - stability
  - parallel
  - equal keys
  - key types
  - large or small items
  - guaranteed vs random performance

System sort is usually good enough but not always

	inplace?	stable?	worst	average	best	remarks
selection	✔		$N^2 / 2$	$N^2 / 2$	$N^2 / 2$	*N* exchanges
insertion	✔	✔	$N^2 / 2$	$N^2 / 4$	N	use for small *N* or partially ordered
shell	✔		?	?	N	tight code, subquadratic
merge		✔	N lg N	N lg N	N lg N	*N* log *N* guarantee, stable
quick	✔		$N^2 / 2$	2 N ln N	N lg N	*N* log *N* probabilistic guarantee fastest in practice
3-way quick	✔		$N^2 / 2$	2 N ln N	N	improves quicksort in presence of duplicate keys
???	✔	✔	N lg N	N lg N	N	holy sorting grail

Screen clipping taken: 12/07/2017 3:53 PM

Week 3 programming assignment

July 20, 2017 12:26 PM

- watch for null values
- careful with pointers in arrays - at the end of array
    - double check every time you call pointers - what happens with them at the end of arrays

Week 4 - APIs and Elementary Implementations

July 21, 2017 4:05 PM

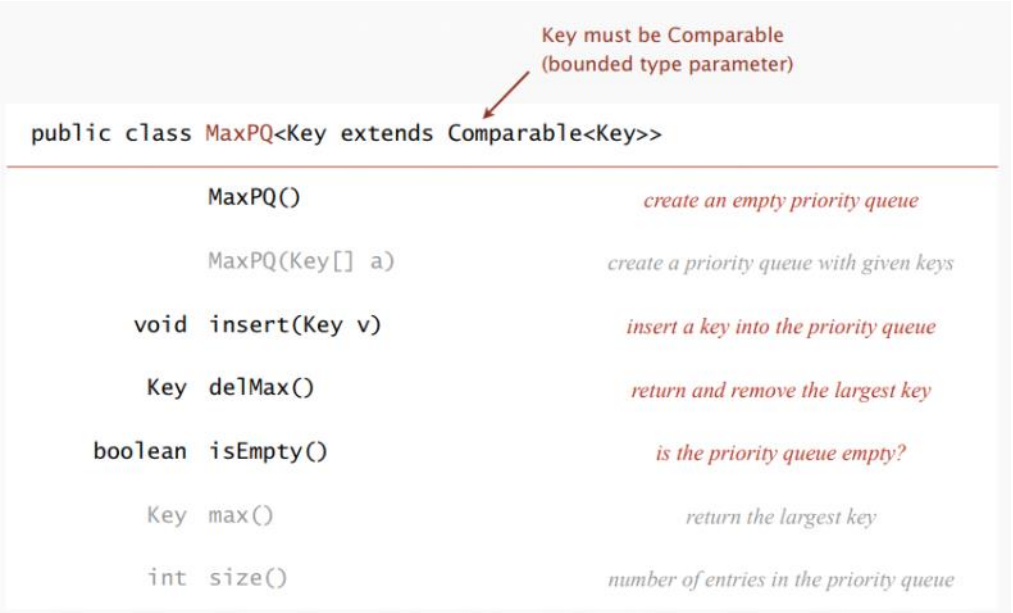
Priority Queues

- like sorting but more flexablility

Priority Queue

- remove the largest or smallest item

API:



Screen clipping taken: 21/07/2017 7:01 PM

Items must be comparable

Lots of applications:

- event driven simulation
- numberical computation
- data compression
- graph searching
- etc

Example: Store the largest M values in a very large N

- memory restrictions
- (credit card fraud detection - only want to look at largest amounts)

To keep the largest M,

- add a transaction
- then delete the lowest value if there are more values than M in the queue

Compared to other methods:

implementation	time	space
sort	N log N	N
elementary PQ	M N	M
binary heap	N log M	M
best in theory	N	M

Screen clipping taken: 21/07/2017 7:11 PM

- this technique too slow
    - binary heap is better(future lesson)

Two ways to set up a prioritized queue

- ordered stack
    - easy to remove
    - hard to add
- unordered stack
    - opposite

Unordered Example:

```
public class UnorderedMaxPQ<Key extends Comparable<Key>>
{
   private Key[] pq;   // pq[i] = ith element on pq
   private int N;      // number of elements on pq

   public UnorderedMaxPQ(int capacity)
   {  pq = (Key[]) new Comparable[capacity];  }

   public boolean isEmpty()
   {  return N == 0; }

   public void insert(Key x)
   {  pq[N++] = x;  }

   public Key delMax()
   {
      int max = 0;
      for (int i = 1; i < N; i++)
         if (less(max, i)) max = i;
      exch(max, N-1);
      return pq[--N];
   }
}
```

no generic array creation

less() and exch() similar to sorting methods

null out entry to prevent loitering

Screen clipping taken: 21/07/2017 7:14 PM

Running Time

order of growth of running time for priority queue with N items

implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
goal	log N	log N	log N

Screen clipping taken: 21/07/2017 7:15 PM

Week 4 - Binary Heaps

July 21, 2017 7:17 PM

Binary Tree

- empty or node with links to left an right binary tree

Complete Tree

- perfectly balanced except for bottom level
- height of tree with N nodes is floor of lgN

Binary Heap

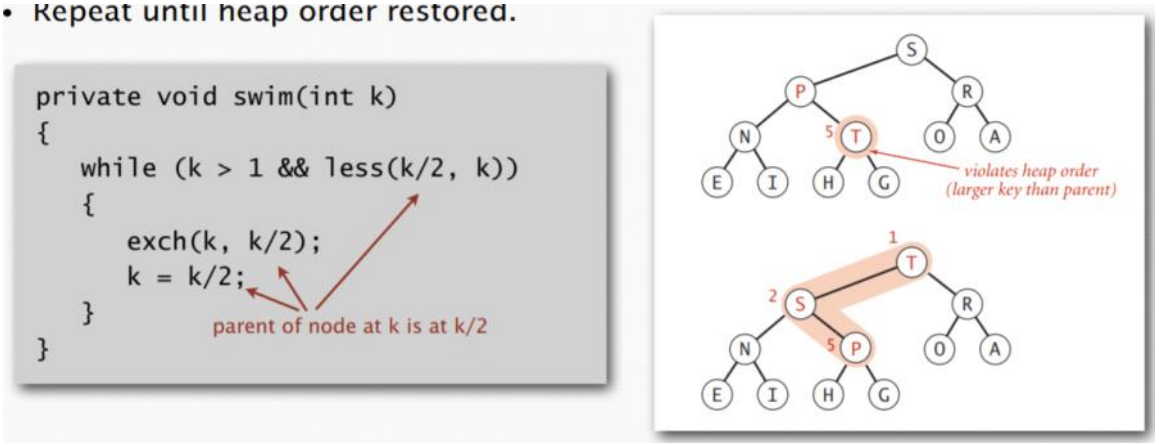
- array implementation of a heap ordered binary tree
  - keys in nodes
  - Parent's key no smaller than children's keys
- indices start at one
- nodes in level order
  - no explicit links needed

- a[1] will be the largest key
- parent of node k is at k/2
- children of node k are at 2k and 2k+1

Promotion

- violates invariants above but then fixes it later
- if a child's key becomes larger than its parent's
  - a value change
- exchange the keys in parent and child
  - keep moving up the tree until the invariant is restored

code (the child swims to the top)



Screen clipping taken: 21/07/2017 10:12 PM

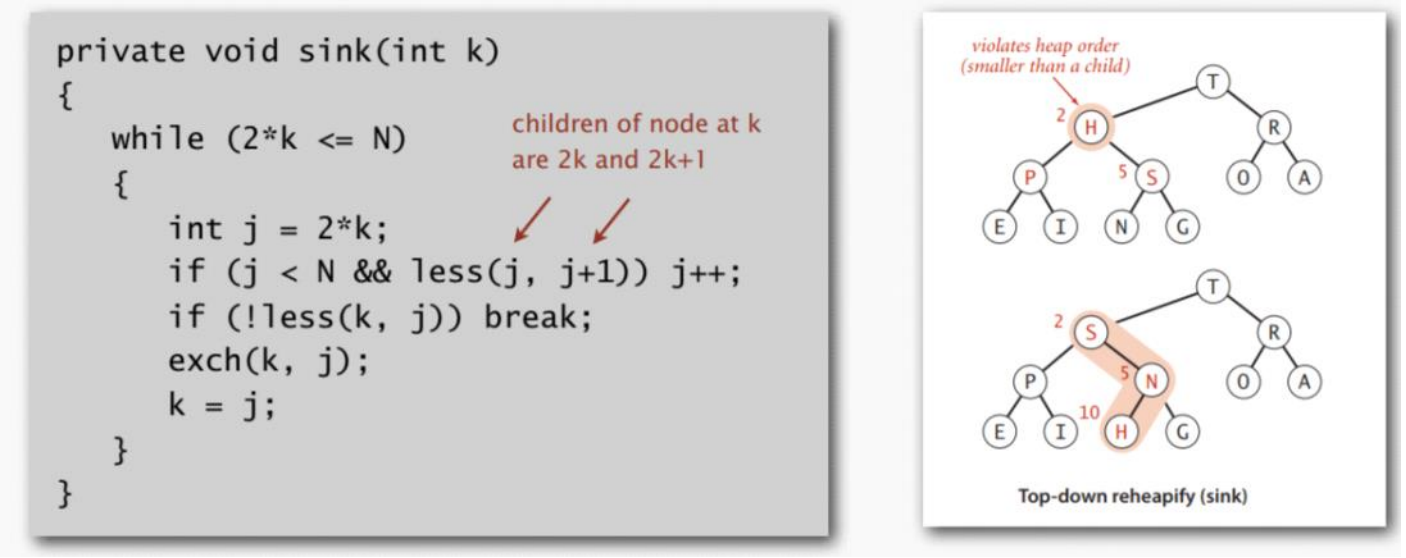
This allows us to insert an element in the heap

- add a new element
- let the element swim up

Demotion

- if a parent is smaller than its children
- swap with the larger child
- keep moving until invariant is restored

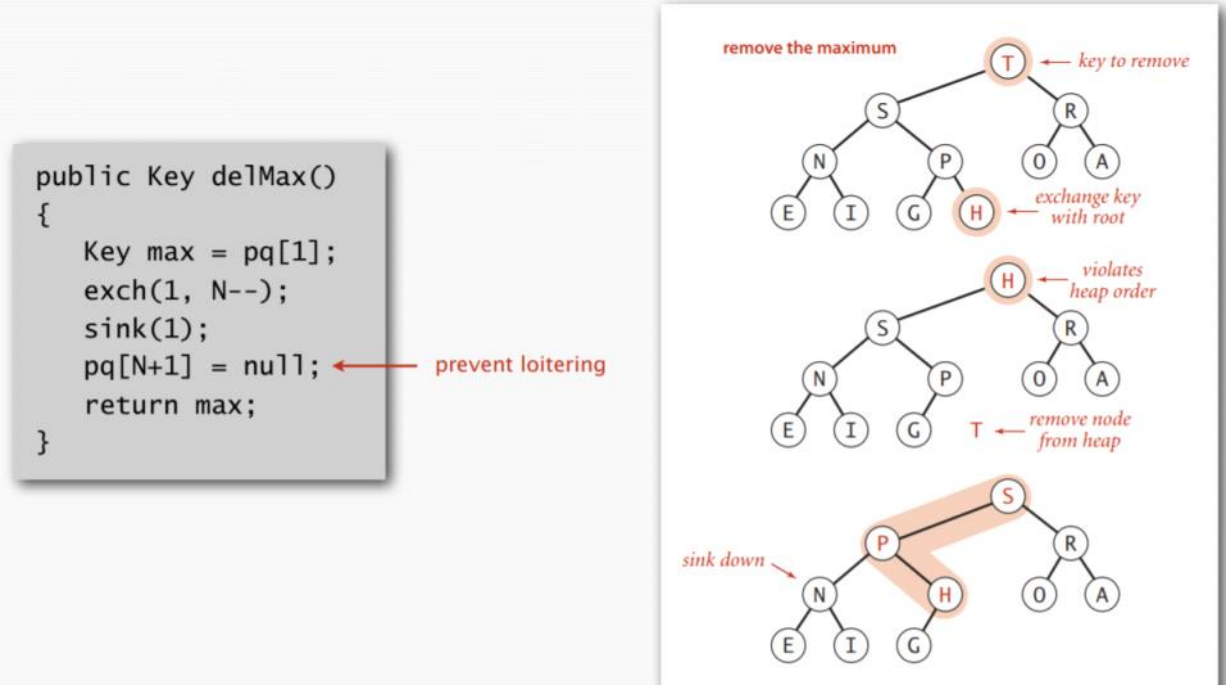
code (the parent sinks to the bottom)



Screen clipping taken: 21/07/2017 10:27 PM

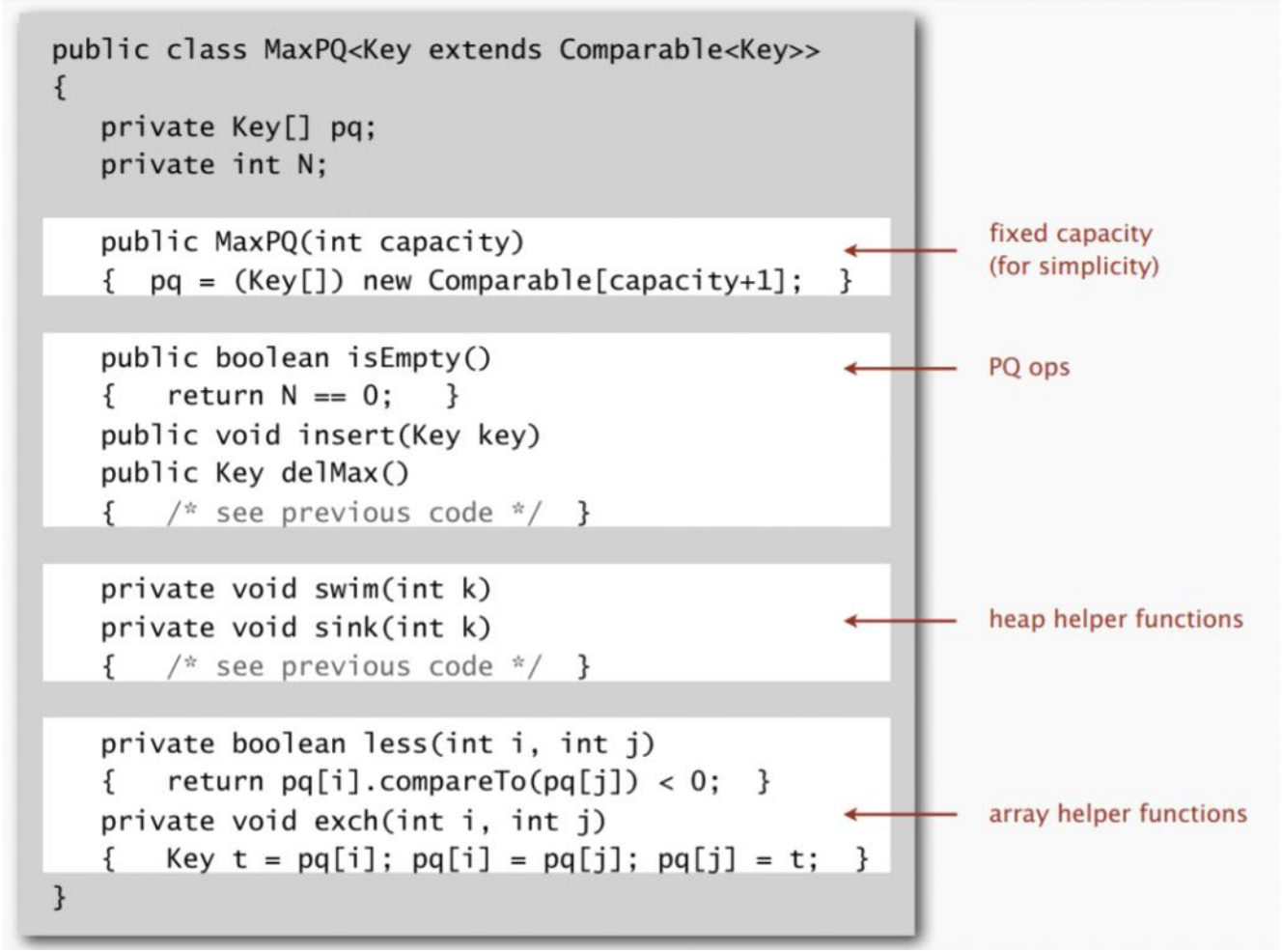
Delete the maximum in a heap

- exchange root with node at end - let it sink down



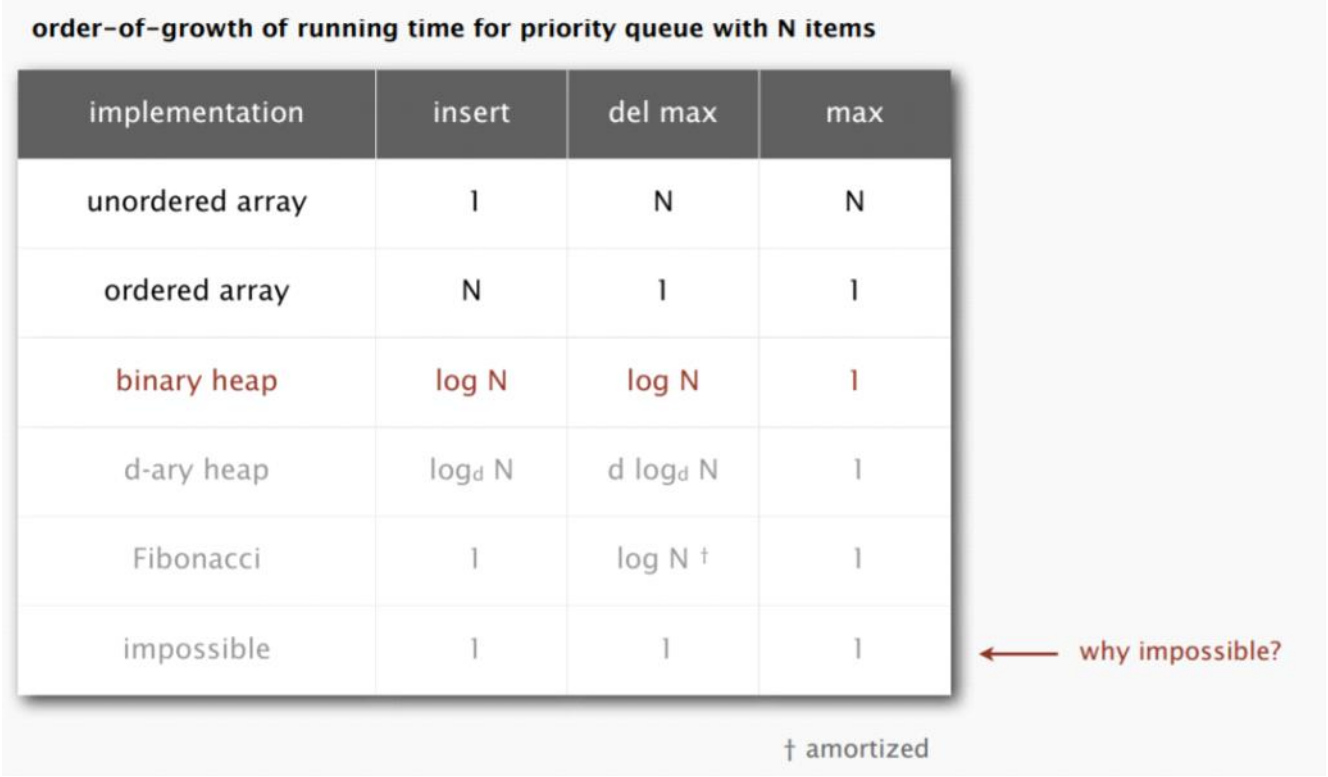
Screen clipping taken: 21/07/2017 10:29 PM

Complete Implementation



Screen clipping taken: 21/07/2017 10:38 PM

Cost Summary



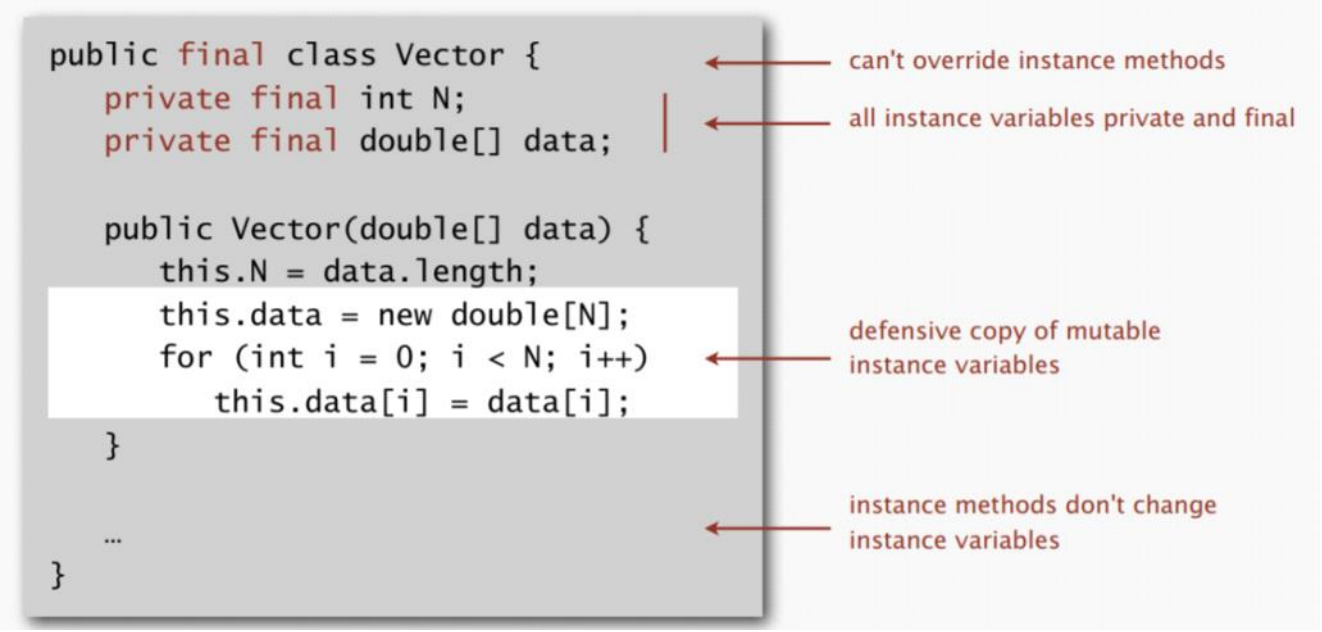
Screen clipping taken: 21/07/2017 10:39 PM

Heap Considerations

- Immutability of keys
  - assuming that client does not change keys while they're using the priority queue
    - **use immutable keys**
- Underflow and Overflow
  - throw exception if deleting from empty PQ
  - add resizing array
- minimum oriented priority queue
- Other options
  - remove arbitrary options
  - Change the priority of an item

Immutablility
- everything in java is a data type
- immutable - can't change the data type once created

Example:



Screen clipping taken: 21/07/2017 10:43 PM

Advantages:
- simplifies debugging
- Safer in presence in hostile code

Disadvantages
- need to create a new object for each data type value

Week 4 - Heapsort

July 21, 2017 11:08 PM

Heapsort

- have n keys in array
  - build a mas heap
- remove the largest value
- in place sort

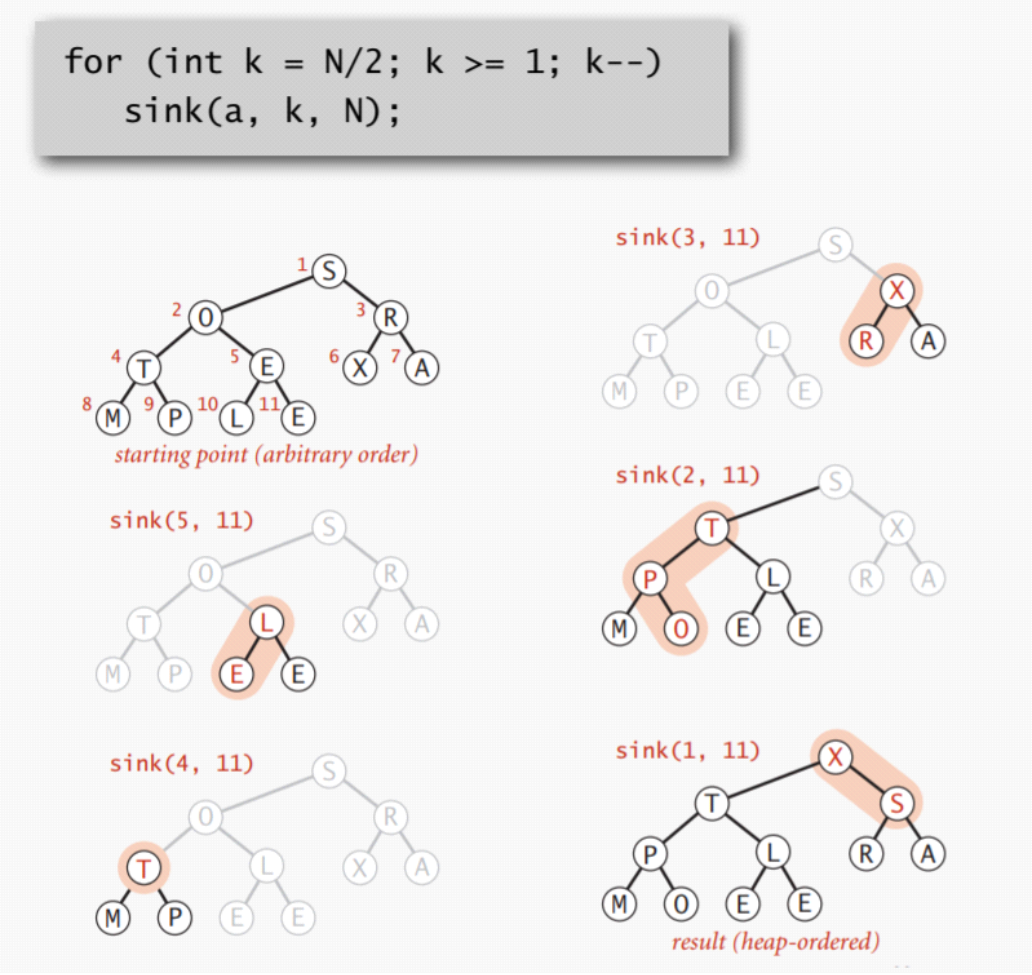
Heap construction

- bottom up construction
  - Start with 1 node heaps at bottom
  - sink the parent of the two nodes at the bottom
  - repeat till you reach the top of the heap

Sort with heaps

- exchange first and last element
- sink the new first element
- keep going until heap is emptied

Heap Construction Code



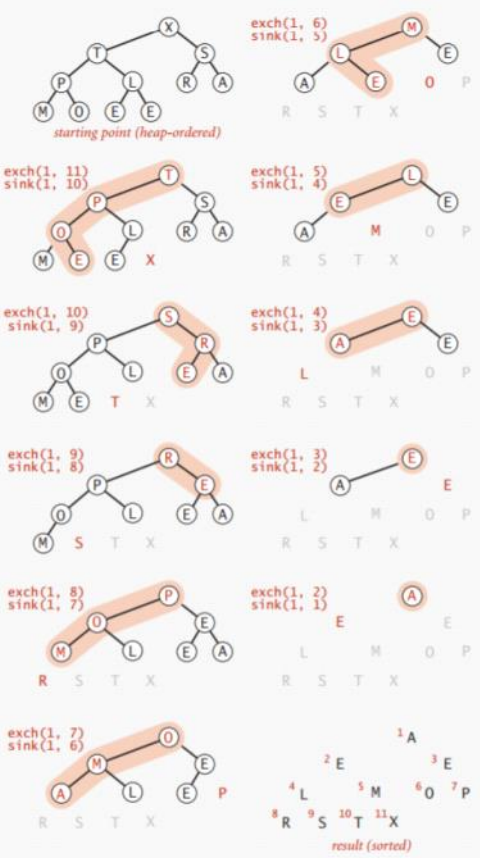
Screen clipping taken: 21/07/2017 11:58 PM

Removing the maximum

Second pass.

- Remove the maximum, one at a time.
- Leave in array, instead of nulling out.

```
while (N > 1)
{
   exch(a, 1, N--);
   sink(a, 1, N);
}
```



Screen clipping taken: 21/07/2017 11:59 PM

Total java Implementation

```
public class Heap
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int k = N/2; k >= 1; k--)
         sink(a, k, N);
      while (N > 1)
      {
         exch(a, 1, N);
         sink(a, 1, --N);
      }
   }

   private static void sink(Comparable[] a, int k, int N)
   {  /* as before */  }

   private static boolean less(Comparable[] a, int i, int j)
   {  /* as before */  }

   private static void exch(Comparable[] a, int i, int j)
   {  /* as before */  }

}
```

but convert from 1-based indexing to 0-base indexing

Screen clipping taken: 22/07/2017 12:01 AM

Mathematical Analysis

- consturction uses 2N compares and exchanges
- sort uses 2NlgN compares and exchanges

First in place sorting algorithm with NlgN compares in the worst case

Heapsort is optimal for time and space BUT

1. inner loop is longer than quicksort
2. poor use in cache memory
    a. accesses information far away - quicksort will access information closer
3. not stable

Summary

	inplace?	stable?	worst	average	best	remarks
selection	x		$N^2 / 2$	$N^2 / 2$	$N^2 / 2$	N exchanges
insertion	x	x	$N^2 / 2$	$N^2 / 4$	N	use for small N or partially ordered
shell	x		?	?	N	tight code, subquadratic
quick	x		$N^2 / 2$	2 N ln N	N lg N	N log N probabilistic guarantee fastest in practice
3-way quick	x		$N^2 / 2$	2 N ln N	N	improves quicksort in presence of duplicate keys
merge		x	N lg N	N lg N	N lg N	N log N guarantee, stable
heap	x		2 N lg N	2 N lg N	N lg N	N log N guarantee, in-place
???	x	x	N lg N	N lg N	N lg N	holy sorting grail

Screen clipping taken: 22/07/2017 12:05 AM

Week 4 - Symbol Table APIs

July 22, 2017 12:25 AM

Symbol Tables

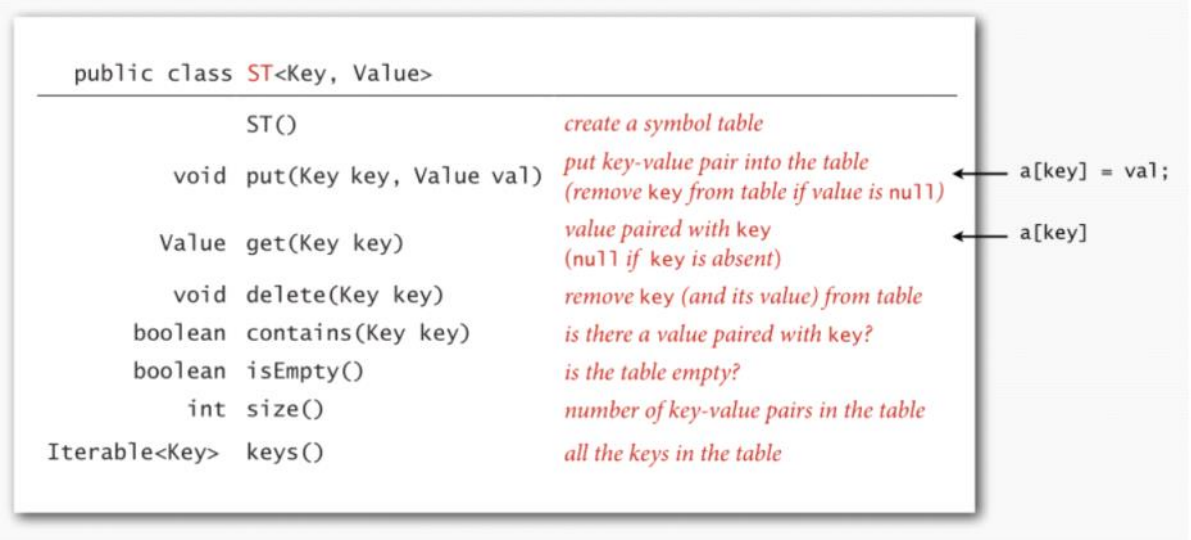
- insert a value with a specific key
- given a key, search for the corresponding value
    - key-value pair

Ex.
DNS lookup.

- insert UFL with IP adress
- Given URL find the IP adress

Basic Symbol Table API

- associative array abstraction
- Associate one value with each key
    - the key is the index - value is the value
        - key may not be an interger however - it is generic in our case



Screen clipping taken: 22/07/2017 12:56 AM

Conventions:

- Values are not null
- get() returns null if key not present
- put() overwrites old value with new value

- makes it so the contains() is the same
    - reutrn get(key) != null
- implement lazy version of delete()
    - put(key.null)

Values: any generic type

Key: - natural assumptions

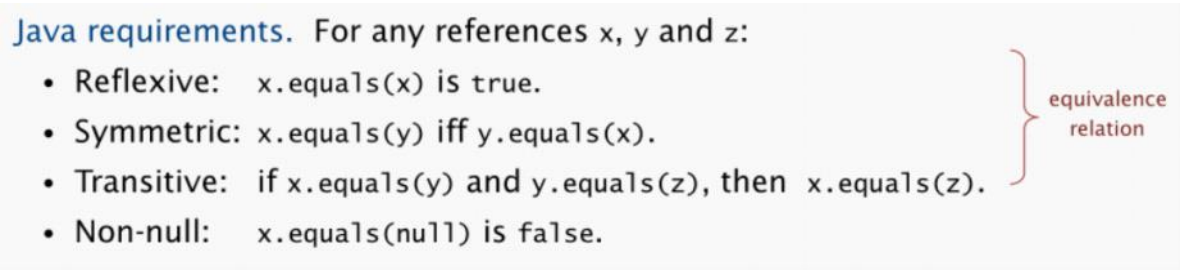
- are comparable
- if cant be comparable
    - any generic type - use equals() to test equality
        - use hash code to scramble key (stay tuned)

Best Practice

- immutable types for symbol table keys

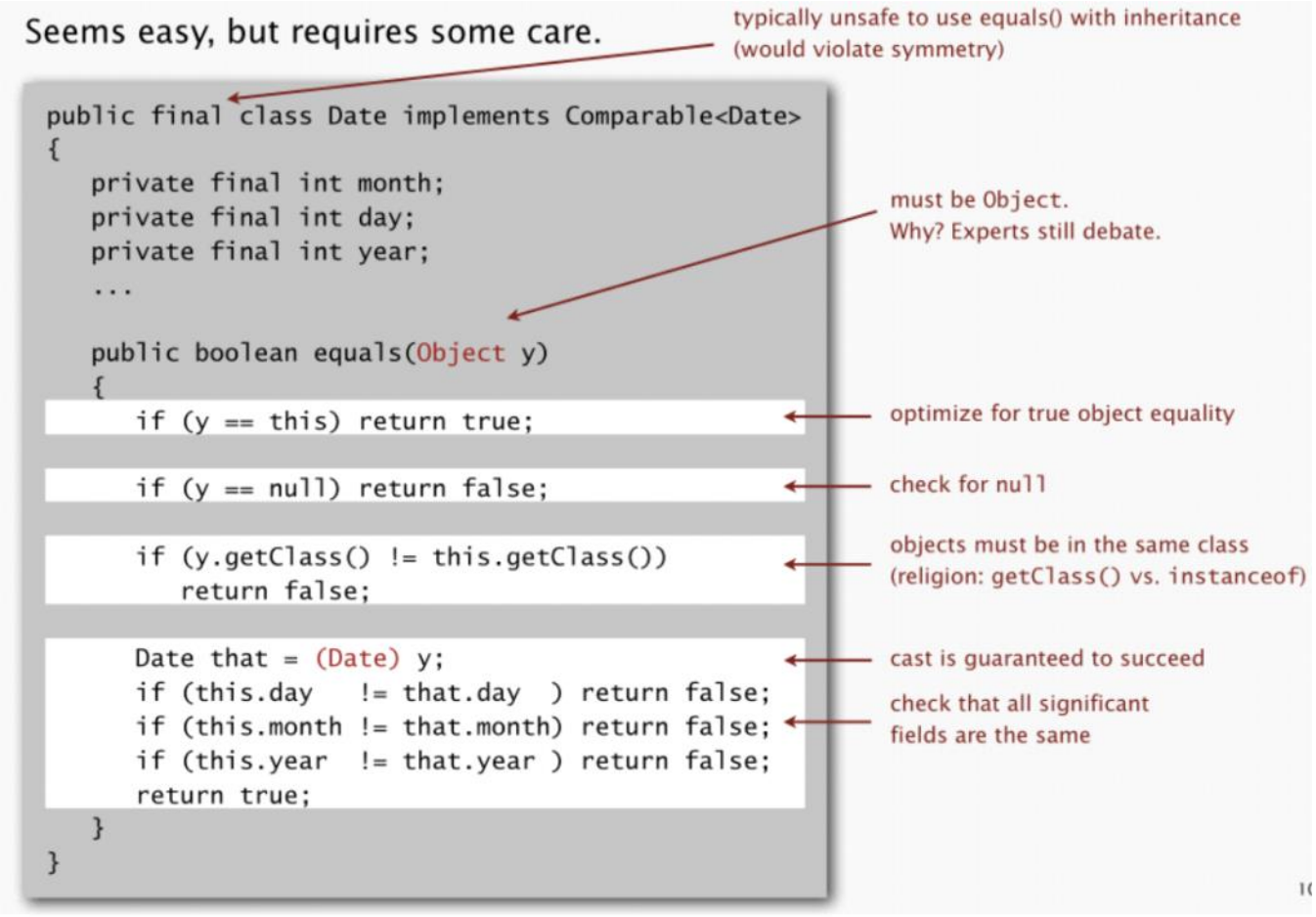
Equality test

- all Java classes inherit a method equals()
- in Java this method is defaulted to is the references equal
    - not practical for general uses



Screen clipping taken: 22/07/2017 1:16 AM

Implementation



Screen clipping taken: 22/07/2017 1:18 AM

Equals Design (same as picture above)

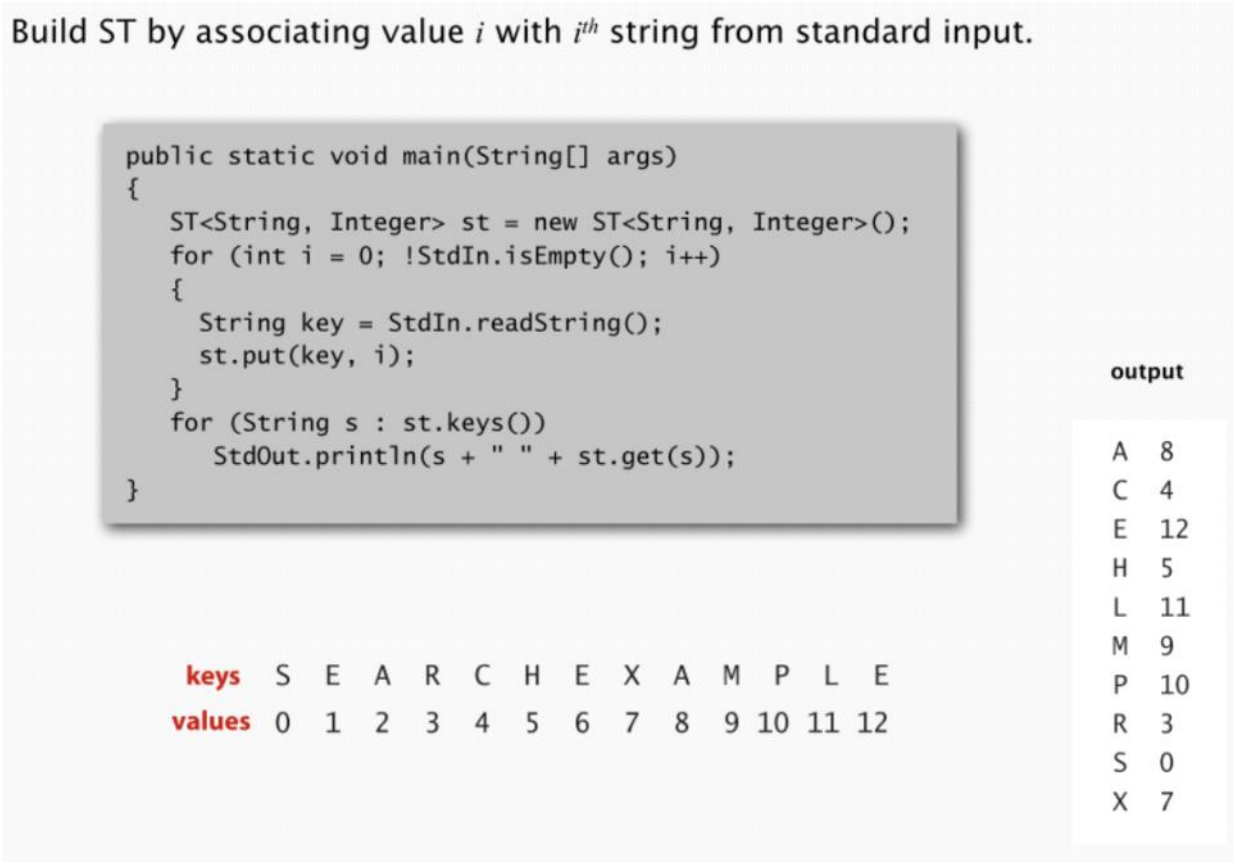
1. optimization for reference
2. check against null
3. two objects are the same type and cast
4. Compare each significant field
    a. if field is primitive type use ==
    b. if it is an object use equals()
        i. apply rule recursively
    c. array

i. apply to each entry

Best Practices

- fields that are most likely to differ compare first
- make compareTo() comply with equals()

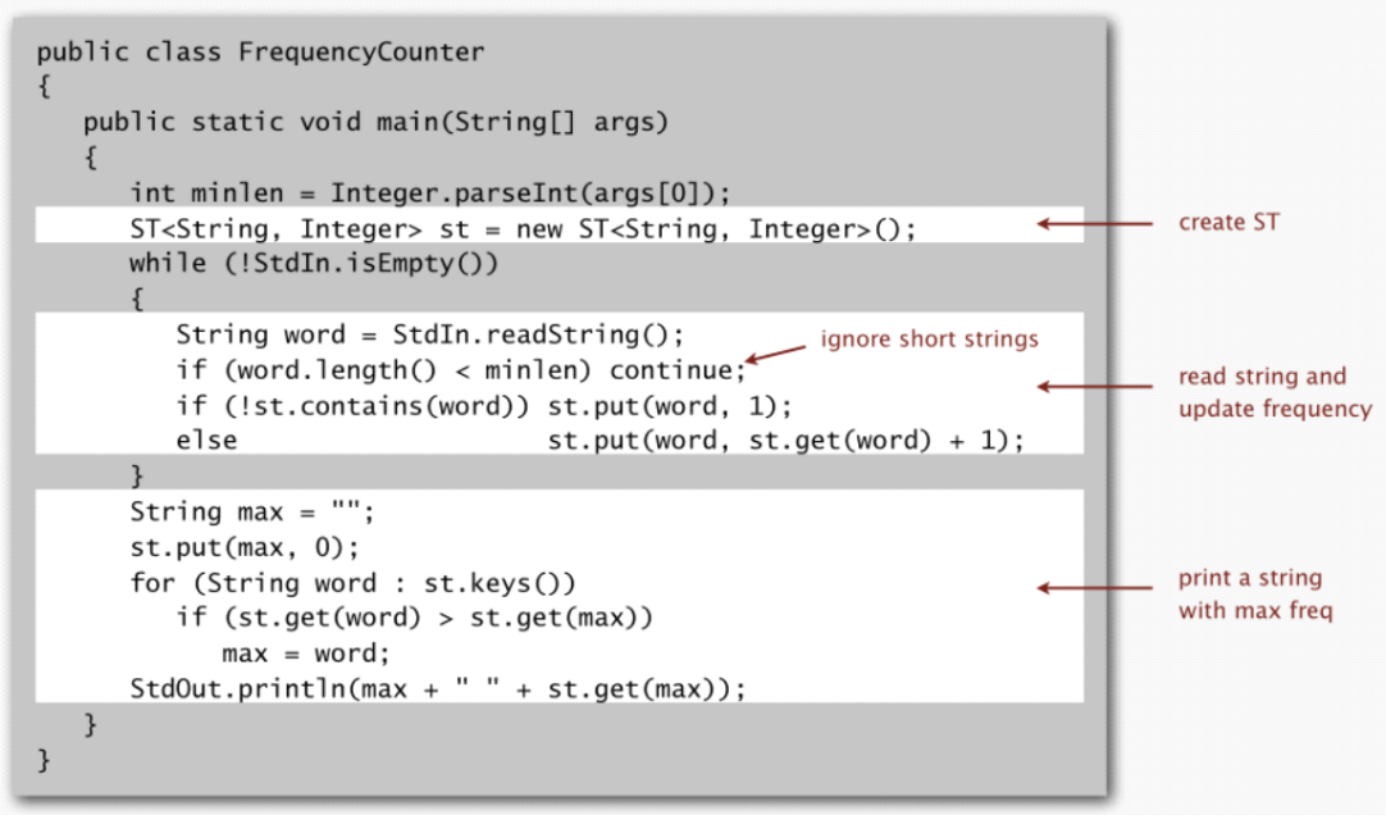
Implementation - indexing client



Screen clipping taken: 22/07/2017 1:38 AM

Frequency Client

- read a sequence of strings from standard input
  - print the one that occurs with highest frequency



Screen clipping taken: 22/07/2017 1:41 AM

Week 4 - Elementary Symbol Table Implementations

July 23, 2017 1:51 PM

1. Maintain an unordered linked list of key value pairs
    a. Search: Scan through all keys to find a match
    b. Insert: Scan through all keys to find a match - if no match add to front
2. Binary search in an ordered array
    a. Have parallel arrays of keys and values
    b. maintain keys in ordered order
        i. get index via binary search and use the index to get value
    c. Rank help function
        i. how many keys are less than k?
    d. Code:

```
public Value get(Key key)
{
   if (isEmpty()) return null;
   int i = rank(key);
   if (i < N && keys[i].compareTo(key) == 0) return vals[i];
   else return null;
}
```

```
private int rank(Key key)                    number of keys < key
{
   int lo = 0, hi = N-1;
   while (lo <= hi)
   {
       int mid = lo + (hi - lo) / 2;
       int cmp = key.compareTo(keys[mid]);
       if      (cmp  < 0) hi = mid - 1;
       else if (cmp  > 0) lo = mid + 1;
       else if (cmp == 0) return mid;
   }
   return lo;
}
```

Screen clipping taken: 23/07/2017 2:55 PM

- rank is binary search
- to insert a key we need to shift everything over
    - anything you do with the key array has to be done with the vals array

Summary of the two methods:

ST implementation	worst-case cost (after N inserts)		average case (after N random inserts)		ordered iteration?	key interface
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N / 2	N	no	equals()
binary search (ordered array)	log N	N	log N	N / 2	yes	compareTo()

Screen clipping taken: 23/07/2017 2:58 PM

- we need an efficient implementation of both sort and insert

Week 4 - Ordered Operations

July 23, 2017 3:51 PM

When keys are comparable and ordered

- has more operations available

Wider Interface:

public class ST<Key extends Comparable<Key>, Value>	
ST()	*create an ordered symbol table*
void put(Key key, Value val)	*put key-value pair into the table (remove* key *from table if value is* null*)*
Value get(Key key)	*value paired with* key *(*null *if* key *is absent)*
void delete(Key key)	*remove* key *(and its value) from table*
boolean contains(Key key)	*is there a value paired with* key*?*
boolean isEmpty()	*is the table empty?*
int size()	*number of key-value pairs*
Key min()	*smallest key*
Key max()	*largest key*
Key floor(Key key)	*largest key less than or equal to* key
Key ceiling(Key key)	*smallest key greater than or equal to* key
int rank(Key key)	*number of keys less than* key
Key select(int k)	*key of rank* k
void deleteMin()	*delete smallest key*
void deleteMax()	*delete largest key*
int size(Key lo, Key hi)	*number of keys in* [lo..hi]
Iterable<Key> keys(Key lo, Key hi)	*keys in* [lo..hi]*, in sorted order*
Iterable<Key> keys()	*all keys in the table, in sorted order*

Screen clipping taken: 23/07/2017 3:59 PM

Costs of all the operations:

	sequential search	binary search
search	N	lg N
insert / delete	N	N
min / max	N	1
floor / ceiling	N	lg N
rank	N	lg N
select	N	1
ordered iteration	N lg N	N

Screen clipping taken: 23/07/2017 4:01 PM

- sequential search is linked list

We need a faster way to insert/delete values - Binary Search Trees

Week 4 - Binary Search Trees

July 23, 2017 4:03 PM

Allows us to perform efficient symbol table operations

Binary Search Tree

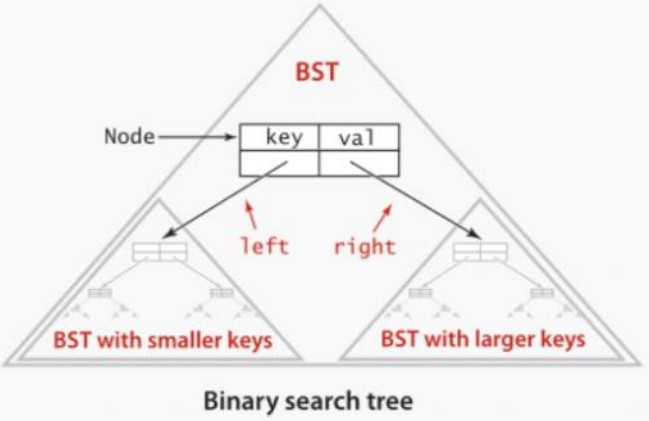
- explicit tree structure (implicit is for example, heap trees)
- binary tree in symmetric order
  - each node has a key and every node's key is
    - larger than all keys in its left subtree
    - smaller than all the keys in its right subtree
  - the parent is the value in-between each subtree

Representation in Java

- extend linked list implementation
  - BST is a reference to a root node
    - key
    - value
    - reference to a right and left subtree

```
private class Node
{
   private Key key;
   private Value val;
   private Node left, right;
   public Node(Key key, Value val)
   {
      this.key = key;
      this.val = val;
   }
}
```

Key and Value are generic types; Key is Comparable

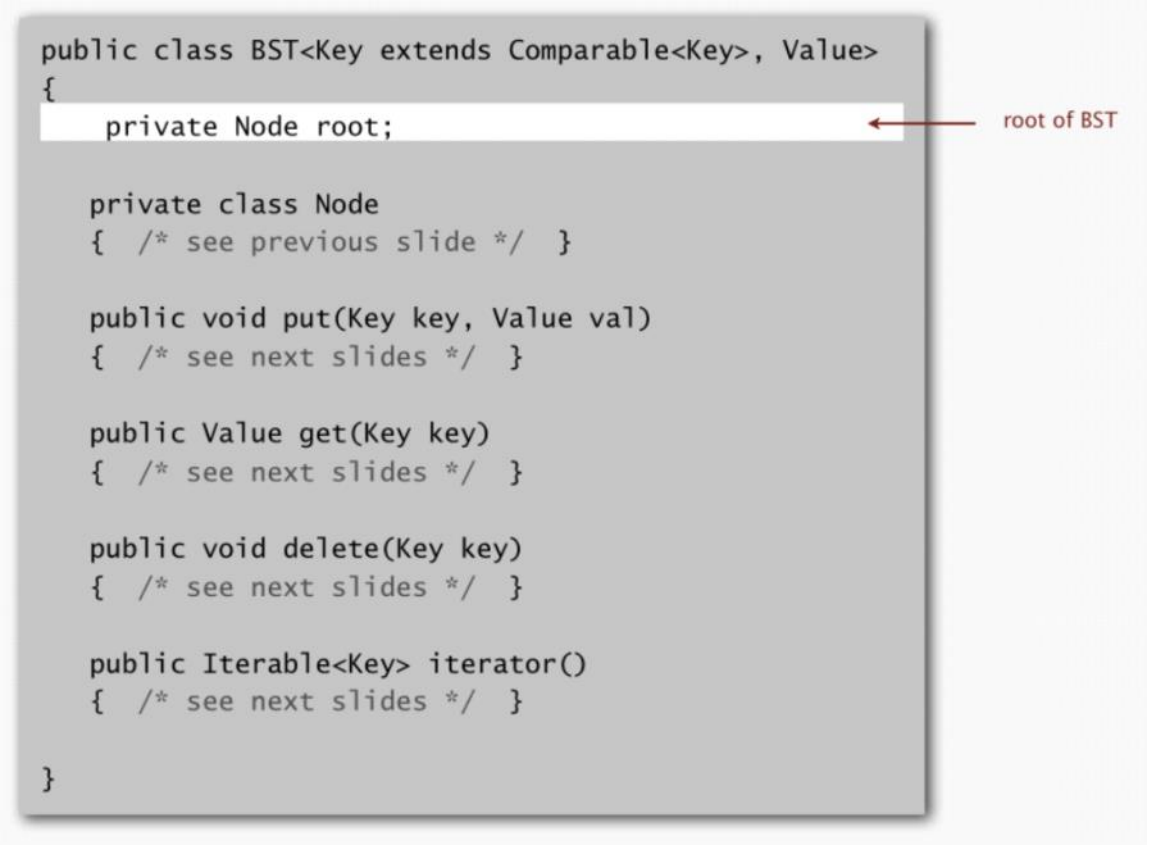


Binary search tree

Screen clipping taken: 23/07/2017 4:08 PM

- left and right nodes are initialized as null

Implementation:



Screen clipping taken: 23/07/2017 4:09 PM

Search (get()):

- if less go left
- if more go right
- if equal then the search is complete

- for an unsuccessful search,
    - no more tree
        - search miss

Insert

- search tree until you get to the null link
    - replace null link with the inserted node

Get Implementation:

```
public Value get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if      (cmp  < 0) x = x.left;
      else if (cmp  > 0) x = x.right;
      else if (cmp == 0) return x.val;
   }
   return null;
}
```

Screen clipping taken: 23/07/2017 4:13 PM

Put:

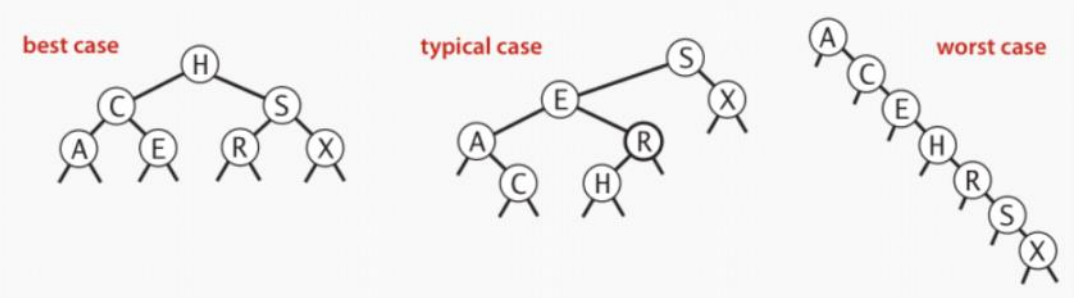
- if key is in tree - reset value
- if key is not in tree - add new node
- Recursive implementation
    - as you go down the tree, return a link to higher up in the tree
    - to fix the links

```
public void put(Key key, Value val)
{  root = put(root, key, val);  }

private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0)
      x.left  = put(x.left,  key, val);
   else if (cmp  > 0)
      x.right = put(x.right, key, val);
   else if (cmp == 0)
      x.val = val;
   return x;
}
```

concise, but tricky, recursive code; read carefully!

Screen clipping taken: 23/07/2017 4:17 PM

- Many different binary search trees that have the same keys
    - depend on the order of insertions



Screen clipping taken: 23/07/2017 4:23 PM

Binary search tree and quicksort partitioning

- 1-1 correspondence if no duplicates
- helps with analysis

Mathematical Analysis

- 2lnN - average
    - same as quicksort
- worst case = 4lnN
    - **keep in mind that worst case for binary search is N**
        - no random

implementation	guarantee		average case		ordered ops?	operations on keys
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N/2	N	no	equals()
binary search (ordered array)	lg N	N	lg N	N/2	yes	compareTo()
BST	N	N	1.39 lg N	1.39 lg N	next	compareTo()

Screen clipping taken: 23/07/2017 4:32 PM

- next refers to next few lectures

Week 4 - Ordered Operations in Binary Search Trees

July 23, 2017 4:39 PM

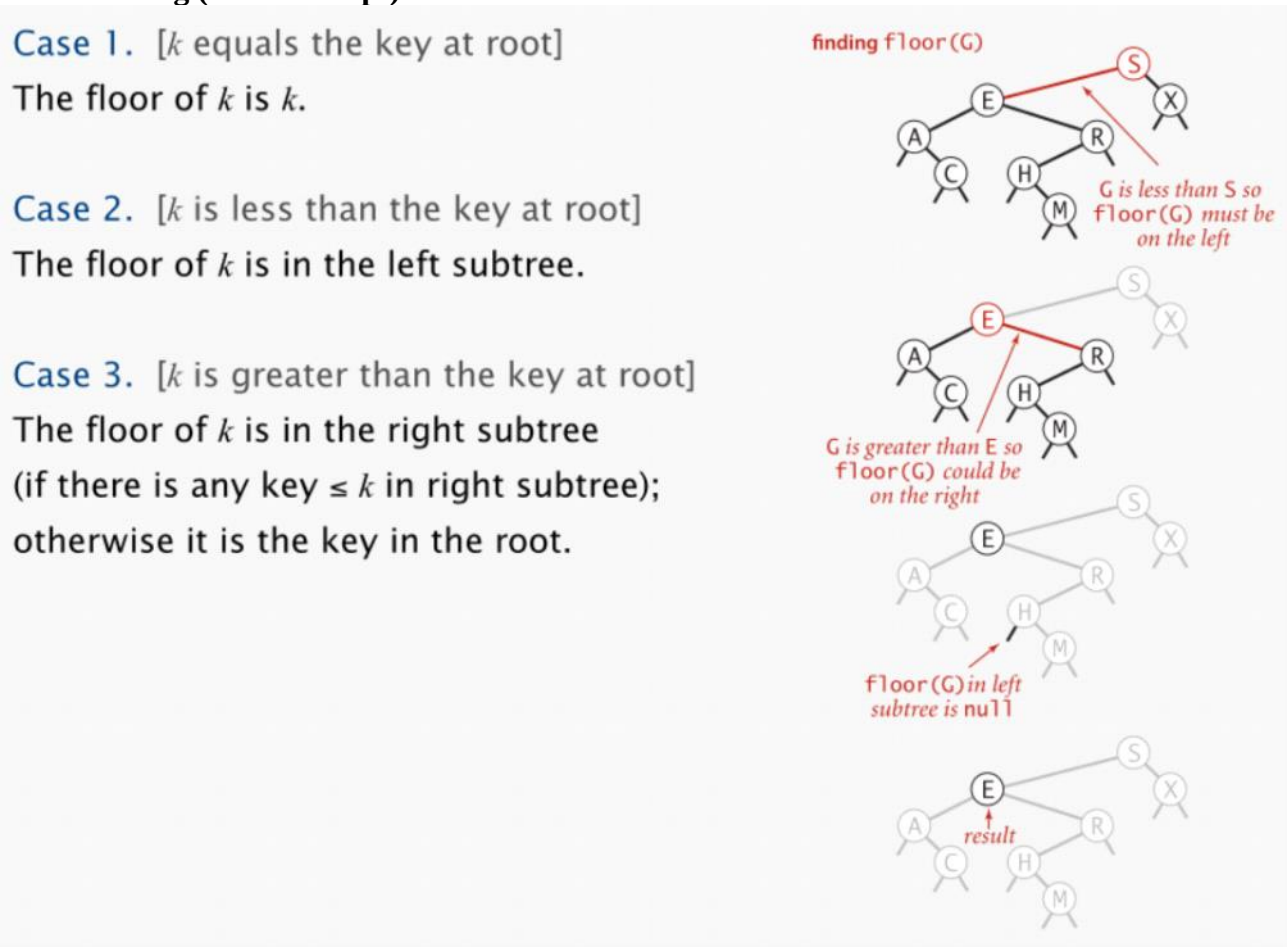
Minimum Key

- the most left key

Maximum Key

- the most right key

Floor/ Ceiling (same concept)



Screen clipping taken: 23/07/2017 4:41 PM

Code:

```
public Key floor(Key key)
{
   Node x = floor(root, key);
   if (x == null) return null;
   return x.key;
}
private Node floor(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);

   if (cmp == 0) return x;

   if (cmp < 0)  return floor(x.left, key);

   Node t = floor(x.right, key);
   if (t != null) return t;
   else           return x;

}
```

Screen clipping taken: 23/07/2017 4:42 PM

Rank/Select

- subtree counts
  - keep a field which is the number of subtrees that it contains (including itself)
- size - count at root

Implementation (subtree count):

```
private class Node
{
   private Key key;
   private Value val;
   private Node left;
   private Node right;
   private int count;
}
```

number of nodes in subtree

```
public int size()
{  return size(root);  }

private int size(Node x)
{
   if (x == null) return 0;
   return x.count;
}
```

ok to call when x is null

```
private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val, 1);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) x.left  = put(x.left,  key, val);
   else if (cmp  > 0) x.right = put(x.right, key, val);
   else if (cmp == 0) x.val = val;
   x.count = 1 + size(x.left) + size(x.right);
   return x;
}
```

Screen clipping taken: 23/07/2017 4:46 PM

Rank with subtree counts

- recursive algorithm
- to find number of keys less than k
  - if key is equal to key at current node
    - size of left subtree
  - if key is less than the root
    - return the rank of the left subtree
  - if key is greater
    - return 1+ size of left subtree and the rank of the right subtree

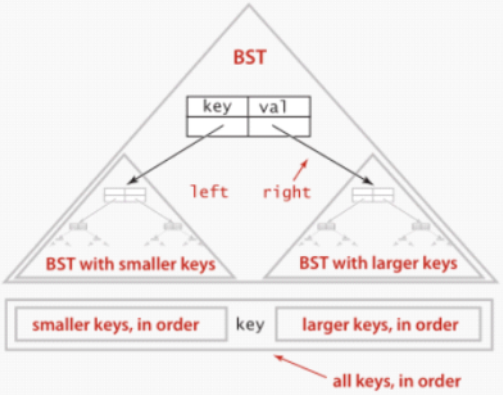
```
public int rank(Key key)
{  return rank(key, root);  }

private int rank(Key key, Node x)
{
   if (x == null) return 0;
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) return rank(key, x.left);
   else if (cmp  > 0) return 1 + size(x.left) + rank(key, x.right);
   else if (cmp == 0) return size(x.left);
}
```

Screen clipping taken: 23/07/2017 6:17 PM

Iteration (inorder traversal)

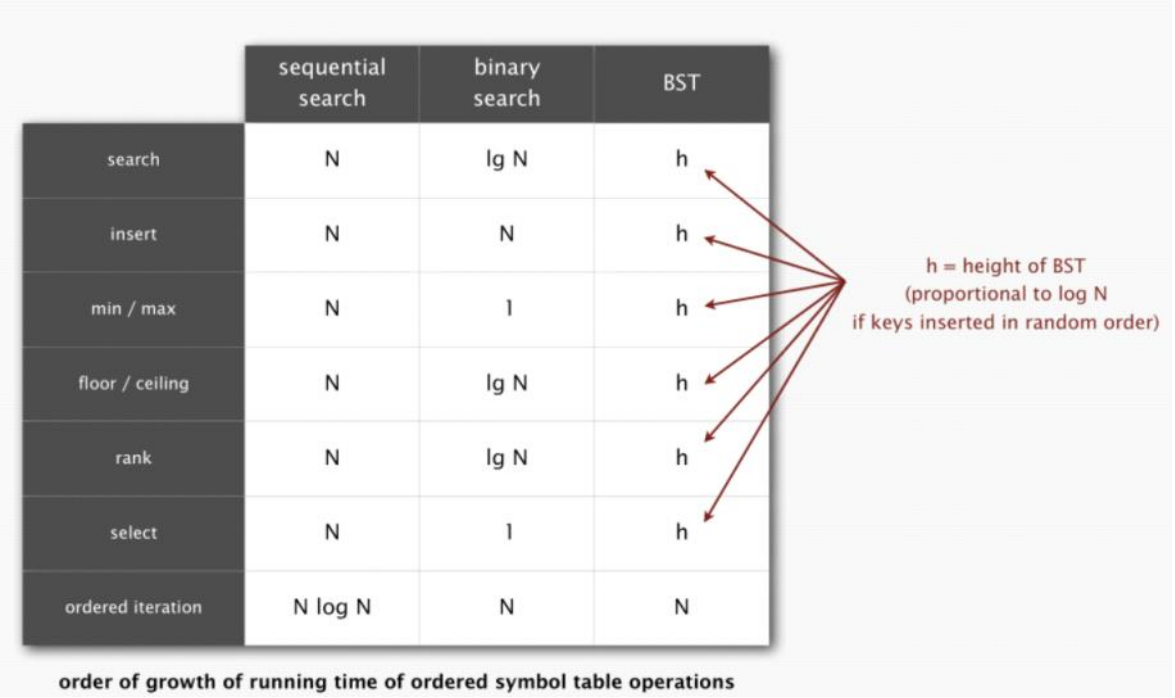
- traverse left subtree
- enqueue key
- Traverse right subtree

```
public Iterable<Key> keys()
{
    Queue<Key> q = new Queue<Key>();
    inorder(root, q);
    return q;
}

private void inorder(Node x, Queue<Key> q)
{
   if (x == null) return;
   inorder(x.left, q);
   q.enqueue(x.key);
   inorder(x.right, q);
}
```



Screen clipping taken: 23/07/2017 6:18 PM

Summary



Screen clipping taken: 23/07/2017 6:19 PM

Week 4 - Deletion in Binary Search Tables

July 23, 2017 6:20 PM

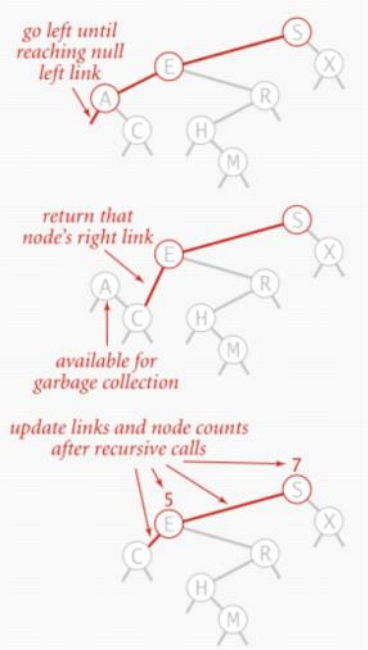
Deletion : Lazy Approach

- set value to null
- leave key in tree to guide searches
- will keep insert search and delete logrithmic

➢ memory overload

Deleting the Minimum

- go left until a node with a null left link
- replace that node with its right link
- update counts



Screen clipping taken: 23/07/2017 6:25 PM

Code:

```
public void deleteMin()
{  root = deleteMin(root);  }

private Node deleteMin(Node x)
{
   if (x.left == null) return x.right;
   x.left = deleteMin(x.left);
   x.count = 1 + size(x.left) + size(x.right);
   return x;
}
```

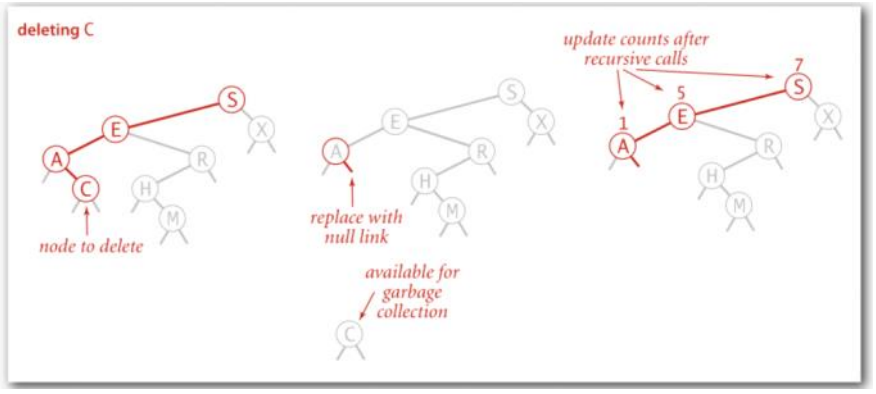
Screen clipping taken: 23/07/2017 6:26 PM

- also works for max

Hibbard Deletion

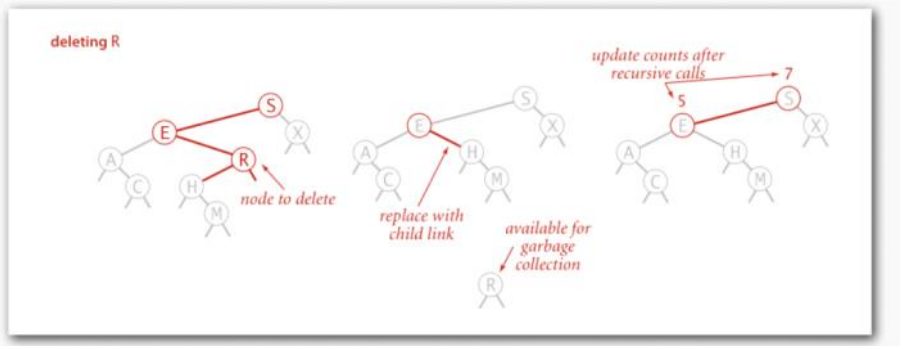
- to delete node with key k
  - search for node t containing k

- Case 1 : 0 Children
  - delete t by setting the link to it to null
  - update counts



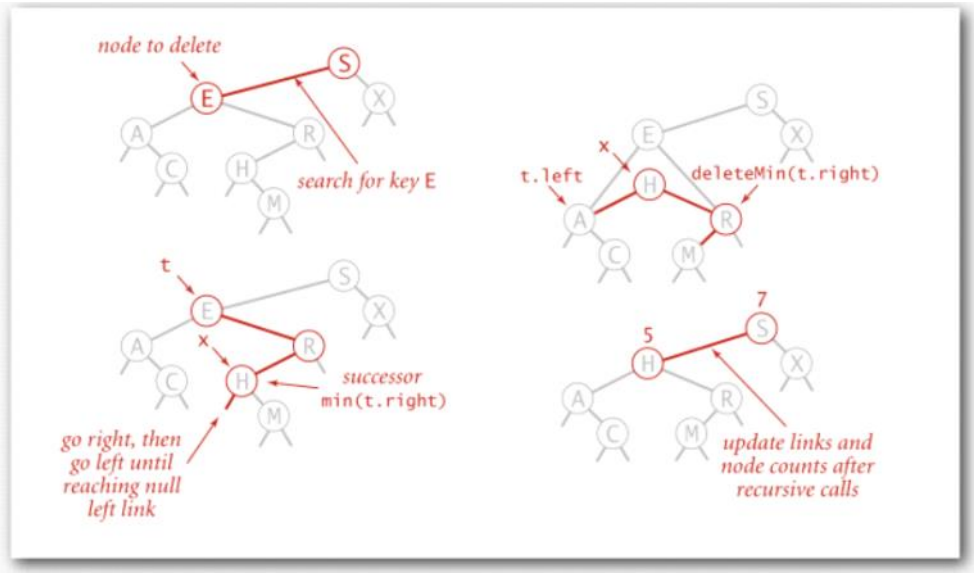
Screen clipping taken: 23/07/2017 6:42 PM

- Case 2: 1 Child
  - replace parent link
    - bypass the node by setting the link to the child to t's child instead of t
  - update counts



Screen clipping taken: 23/07/2017 6:43 PM

- Case 3: 2 Children
  - Find the next smallest node in the right subtree
    - (x)
    - store the value of the next smallest node
    - delete that node
    - replace t with that node
  - Update counts



Screen clipping taken: 23/07/2017 6:46 PM

Code:

```
public void delete(Key key)
{  root = delete(root, key);  }

private Node delete(Node x, Key key) {
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = delete(x.left,  key);
   else if (cmp > 0) x.right = delete(x.right, key);
   else {
      if (x.right == null) return x.left;
      if (x.left  == null) return x.right;

      Node t = x;
      x = min(t.right);
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.count = size(x.left) + size(x.right) + 1;
   return x;
}
```

search for key

no right child

no left child

replace with successor

update subtree counts

Screen clipping taken: 23/07/2017 6:46 PM

Disadvantages:

- the trees become not balanced
  - because we are only taking from the right
- after a long sequence of removes and inserts
  - height becomes sqrt(N)

Summary:

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	equals()
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	compareTo()
BST	N	N	N	1.39 lg N	1.39 lg N	√N	yes	compareTo()

other operations also become √N if deletions allowed

Screen clipping taken: 23/07/2017 6:49 PM

Week 4- Programming Assignment

July 25, 2017 5:14 PM

- when a method returns an iterable object
    - array list or linkedlist
        - array list is better for searching - not as good for when adding or removing a lot of elements
        - linked list is better for removing and adding elements but not for searchign

- when working with size() in a for loop make sure the size stays consistent or store the size in a value beforehand

Week 5 - Balanced Search Trees

July 28, 2017 9:40 PM

Binary search trees are average logarithmic

- we want a data structure that is guaranteed logarithmic
  - 2-3 trees
  - left leaning red-black BST
  - B-trees

2-3 Tree

- allow 1 or 2 keys per node
  - two node - one key two children
  - three node - two key three children
    - less
    - between
    - greater
- Perfect Balance
  - every path from root to the null link has the same length
- Symmetric Order
  - in order transversal yields keys in ascending order

Search

- compare search key against keys in node
- find interval containing search key
- follow associated link

Insert (2- node at bottom)

- search for key
- replace two node with three node

Insert (3- node at bottom)

- add new key to 3-node to make temp 4-node
- move middle key in 4- node into parent
  - leaving two 2-nodes
- If the node at the root is a 3-node add to make 4-node and split into three 2-nodes
  - the middle value in the 4-node becomes the root of a 2-node
  - height of tree increases by 1

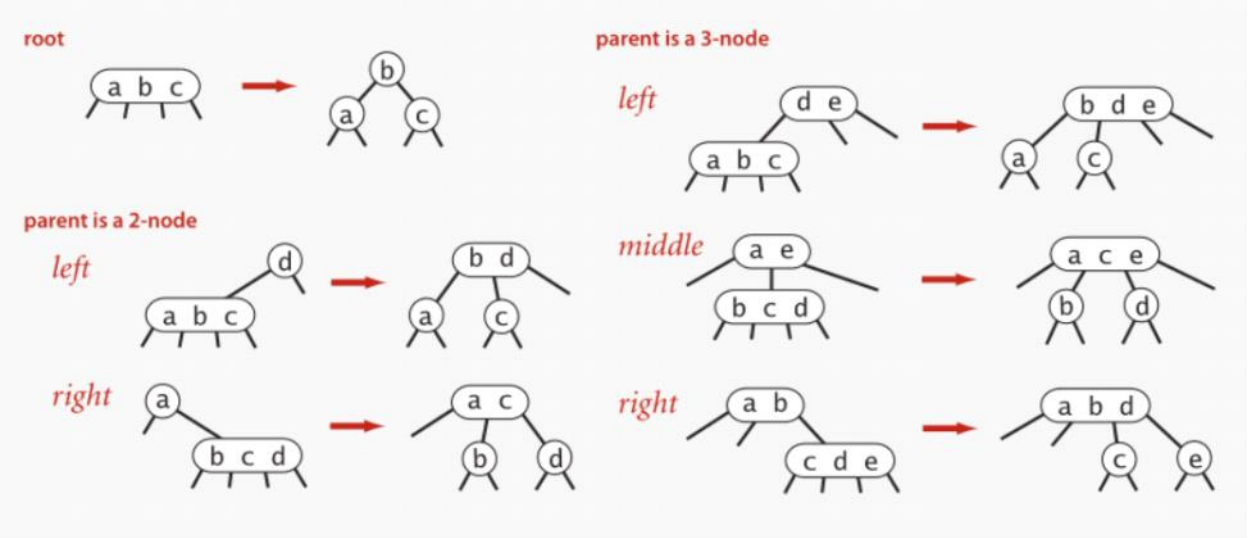
Local Transformations

- splitting a 4-node is a local transformation
  - constant number of operations
- we don't touch anything above or below the nodes that are undergoing the transformation

Global Properties

- Invariants
  - maintains symmetric order and perfect balance
  - all operations maintains symmetric order and perfect balance

All Operations:

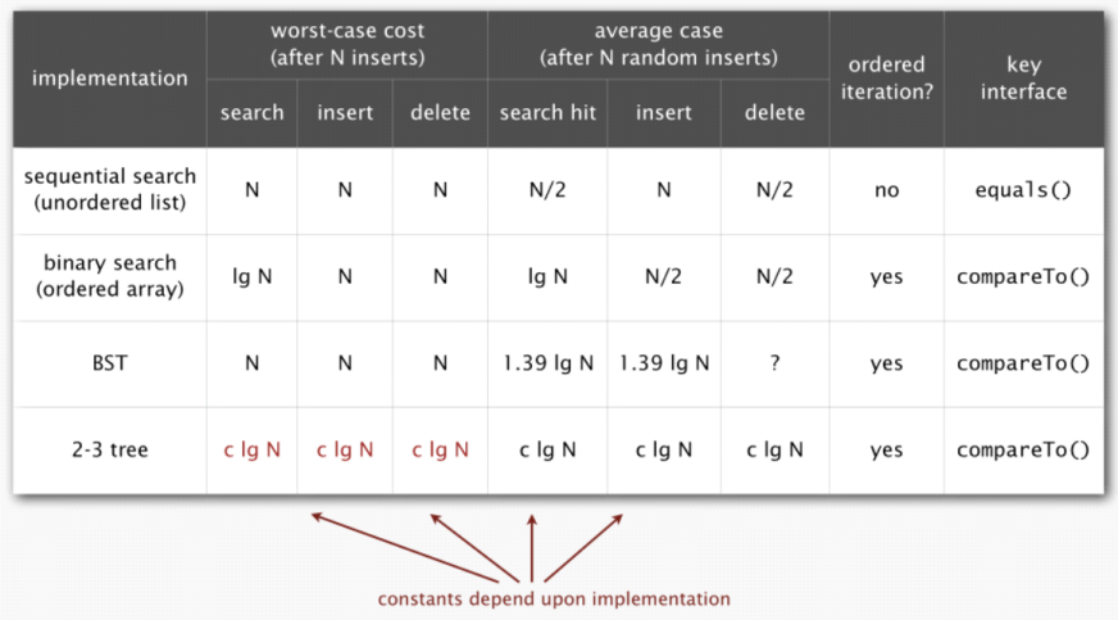


Screen clipping taken: 28/07/2017 10:35 PM

All operations have time proportional to the height of the tree - since the height is the same for all nodes,

- we have guaranteed logarithmic performance for search and insert

Summary:



implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	equals()
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	compareTo()
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	compareTo()
2-3 tree	c lg N	c lg N	c lg N	c lg N	c lg N	c lg N	yes	compareTo()

Screen clipping taken: 28/07/2017 10:44 PM

Implementation:

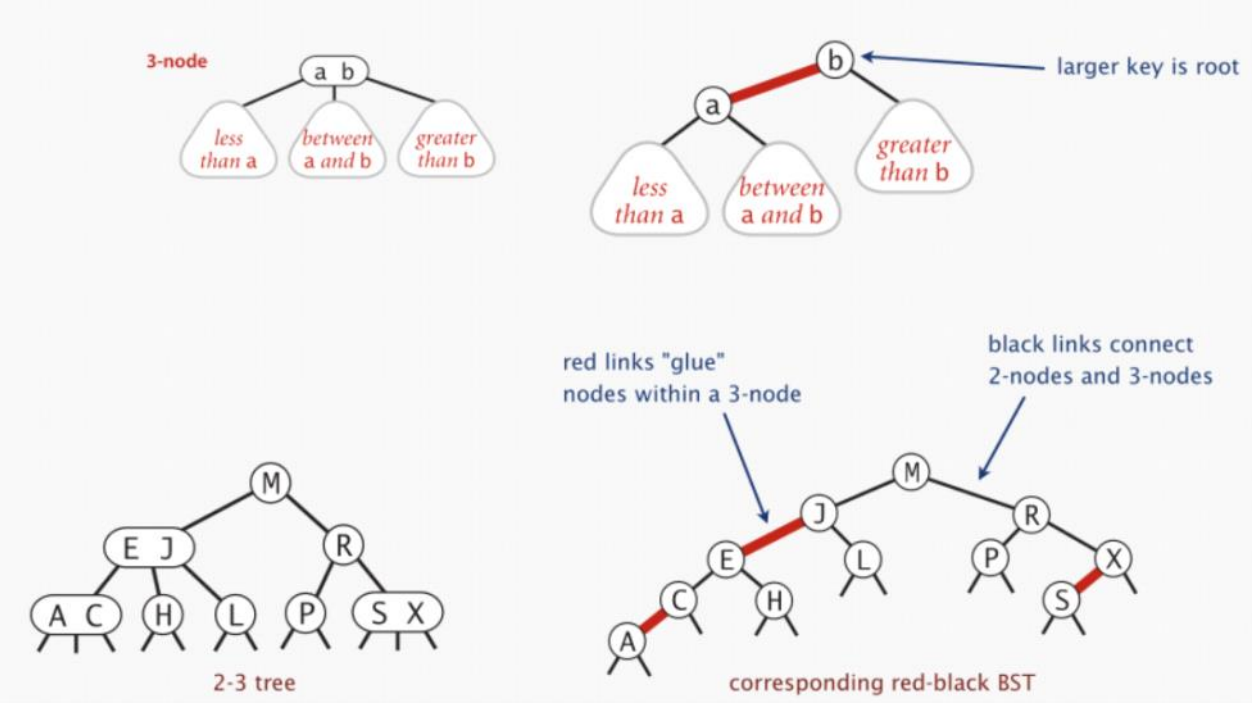
- direct implementation is complicated
  - lots of cases to consider
- possible but much easier way is possible
- is used as a model

Week 5 - Red Black BST's

July 29, 2017 11:22 AM

Allows us to implement 2-3 trees using existing binary search tree code

- represent 2-3 tree BST
  - left leaning
- need simple representation for 3-nodes
  - use internal left leaning links as "glue" for 3-nodes



Screen clipping taken: 29/07/2017 11:25 AM

Definition:

- no node has two red links
- every path from root ahs same amount of black links
  - perfect black balance
- red links lean left

Search

- search is the same as elementary BST
  - ignore color
  - runs faster because of better balance

```
public Val get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if      (cmp  < 0) x = x.left;
      else if (cmp  > 0) x = x.right;
      else if (cmp == 0) return x.val;
   }
   return null;
}
```

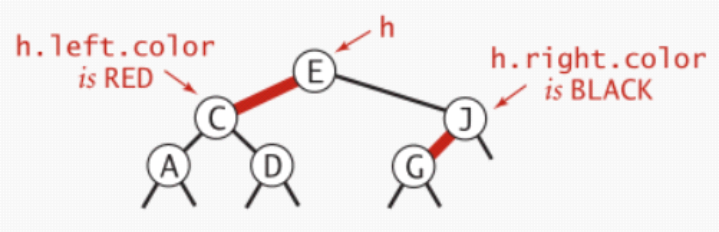
Screen clipping taken: 29/07/2017 11:27 AM

Most of the operations are the same for elementary BST

- but runs faster because of balance

Red-Black Representation

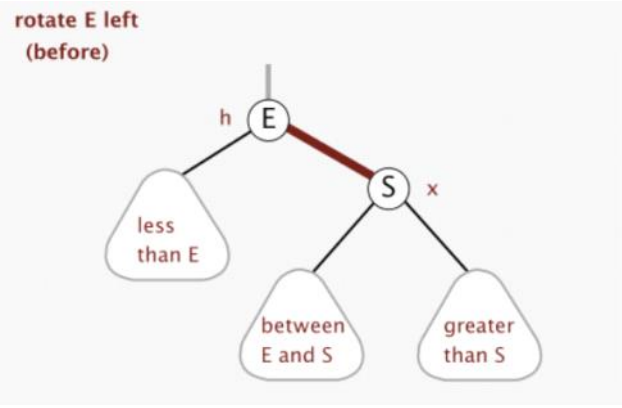
- each node is pointed by one link from its parent
- can encode color in the child node
- null links are black



Screen clipping taken: 29/07/2017 11:29 AM

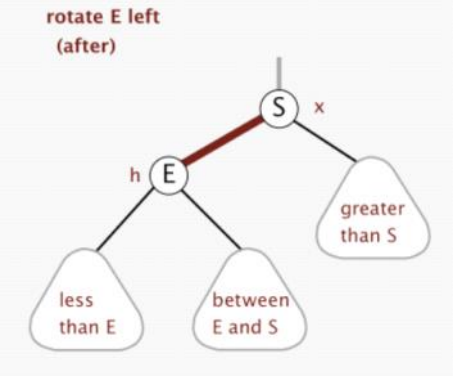
Rotations:

- sometimes red links lean in the wrong direction(right)
  - we need to perform a left rotation
  - make a right leaning red to a left leaning




```
private Node rotateLeft(Node h)
{
   assert isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Screen clipping taken: 29/07/2017 11:31 AM



Screen clipping taken: 29/07/2017 11:32 AM

- all local transformations

Right Leaning Rotations

We may sometimes need to take a left leaning link and make it lean right

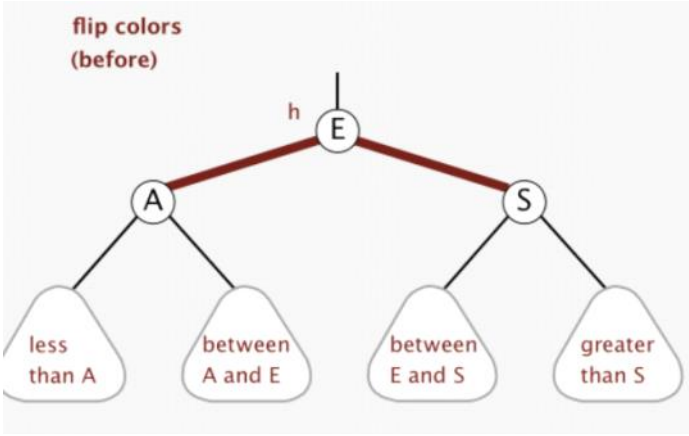
- temporarily for insertion

```
private Node rotateRight(Node h)
{
   assert isRed(h.left);
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Screen clipping taken: 29/07/2017 12:58 PM

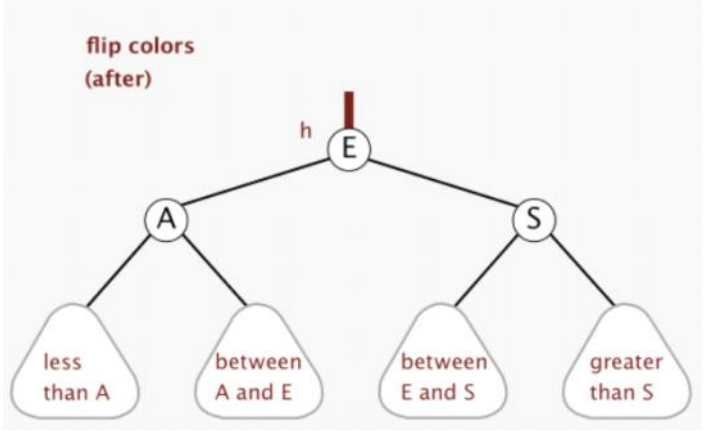
Color Flip

- node with two red links
  - temporary 4-node
- split and pass center node to the root
- only change colors




```
private void flipColors(Node h)
{
   assert !isRed(h);
   assert isRed(h.left);
   assert isRed(h.right);
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

Screen clipping taken: 29/07/2017 1:05 PM



Screen clipping taken: 29/07/2017 1:06 PM

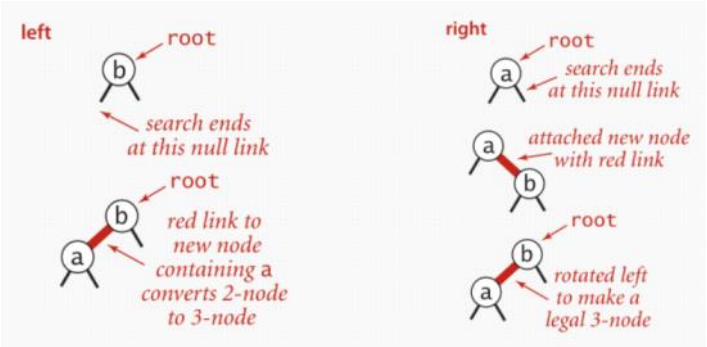
Basic Strategy

- maintain 1-1 correspondence with 2-3 trees by applying red black BST operations(above)

Insert

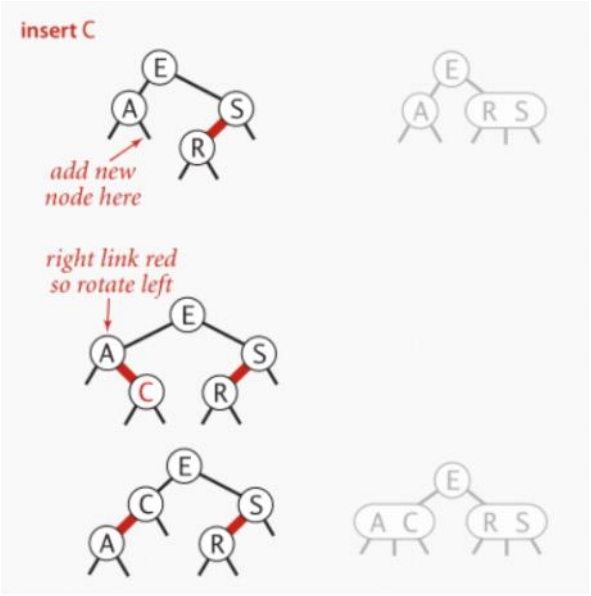
When inserting, we insert with red link to the parent

Insert into tree with only one node



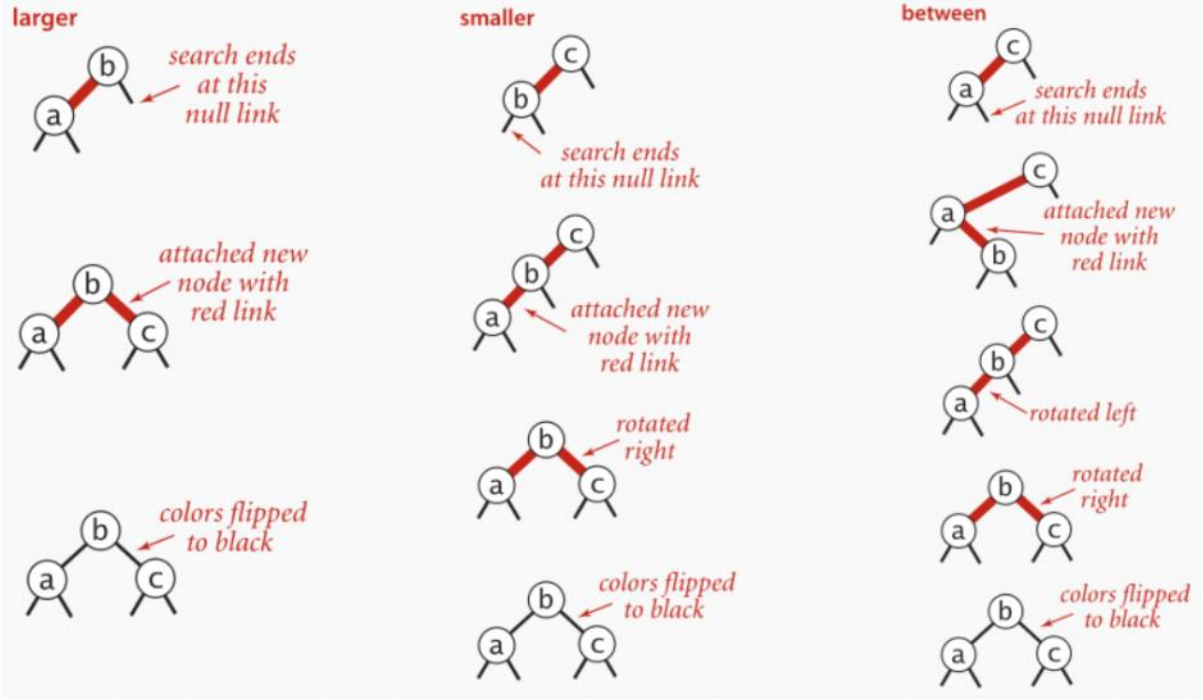
Screen clipping taken: 29/07/2017 1:24 PM

Insert into a 2-node at bottom



Screen clipping taken: 29/07/2017 1:24 PM

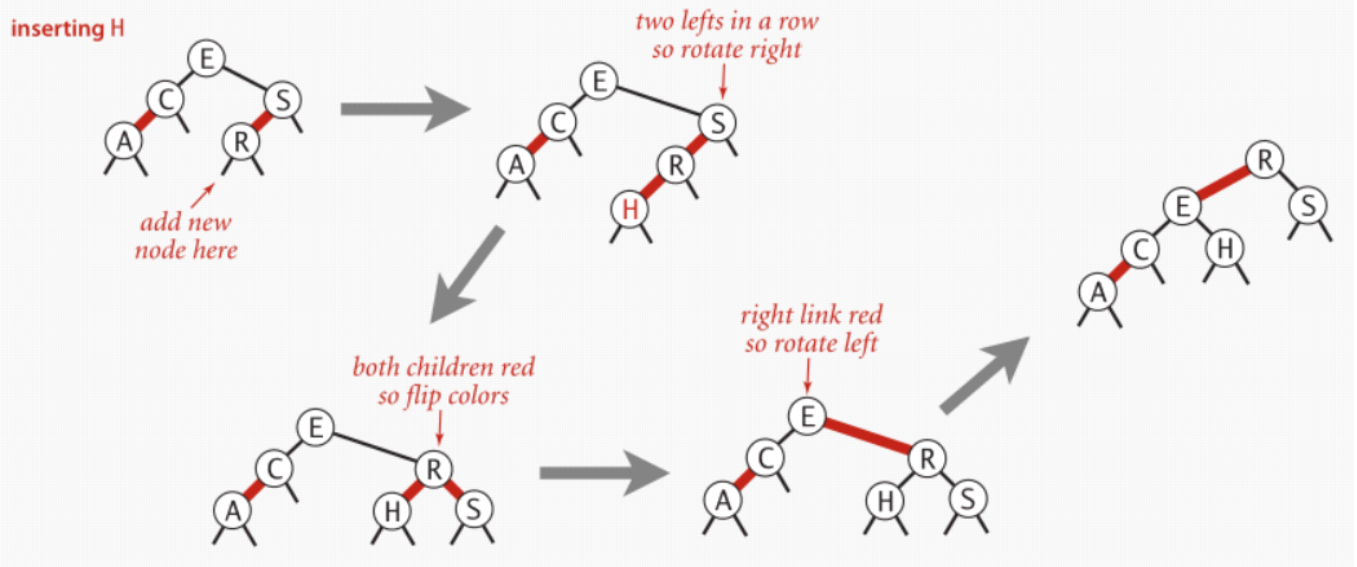
Insert into tree with 2 nodes



Screen clipping taken: 29/07/2017 1:25 PM

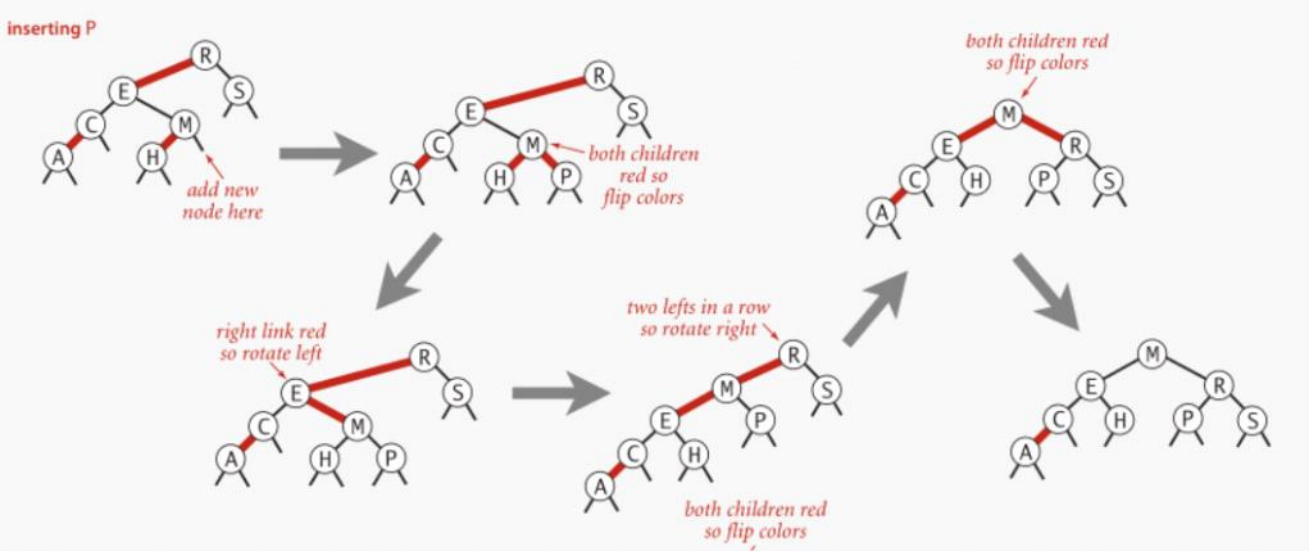
Insert into a 3 node at bottom

- standard BST insert (red link)
- rotate to balance 4-node (if needed)
- flip colors to pass red link
- rotate to make lean left (if needed)
- We passed a red link up
    - repeat case 1 or 2 up the tree (if needed)
        - (passing a red link up into a 3-node



Screen clipping taken: 29/07/2017 1:39 PM

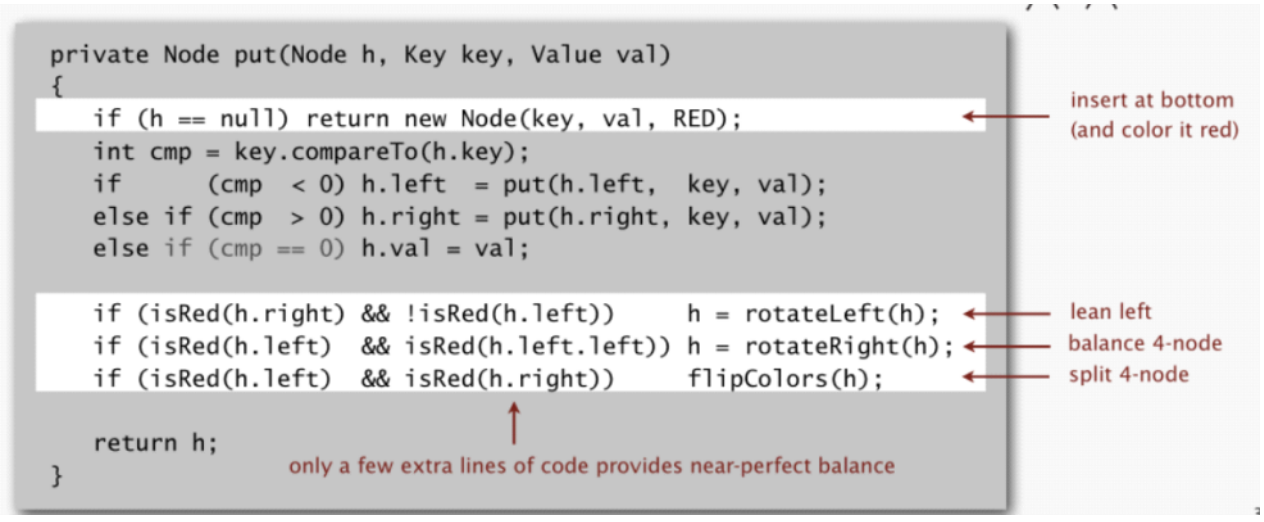
Passing red links up in the tree



Screen clipping taken: 29/07/2017 1:41 PM

Insert Implementation

- Same code handles all cases
  - right child red, left child black
    - rotate left
  - left child red, left left gradchild red
    - rotate right
  - both children red
    - flip



Screen clipping taken: 29/07/2017 2:04 PM

Height of tree is guaranteed 2lgN in the worst case

- height of tree is ~lgN in typical applications

Summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	equals()
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	compareTo()
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	compareTo()
2-3 tree	c lg N	c lg N	c lg N	c lg N	c lg N	c lg N	yes	compareTo()
red-black BST	2 lg N	2 lg N	2 lg N	1.00 lg N *	1.00 lg N *	1.00 lg N *	yes	compareTo()

* exact value of coefficient unknown but extremely close to 1

Screen clipping taken: 29/07/2017 2:12 PM

Week 5 - B trees (red black applications)

July 29, 2017 2:35 PM

B-trees

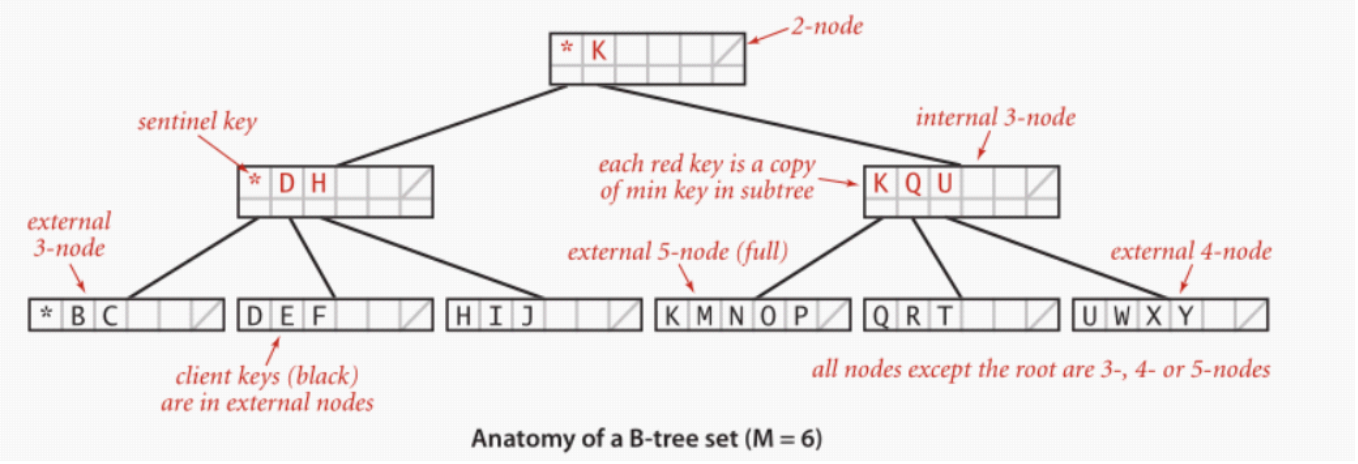
- general version

File System

- we have large data files
- Probing in the hard drive is the bottleneck operation
  - once first accessed we can access information in the page very fast
- the cost model is based on the number of probes
- the goal of file systems is to access data using minimum number of probes

B-Tree

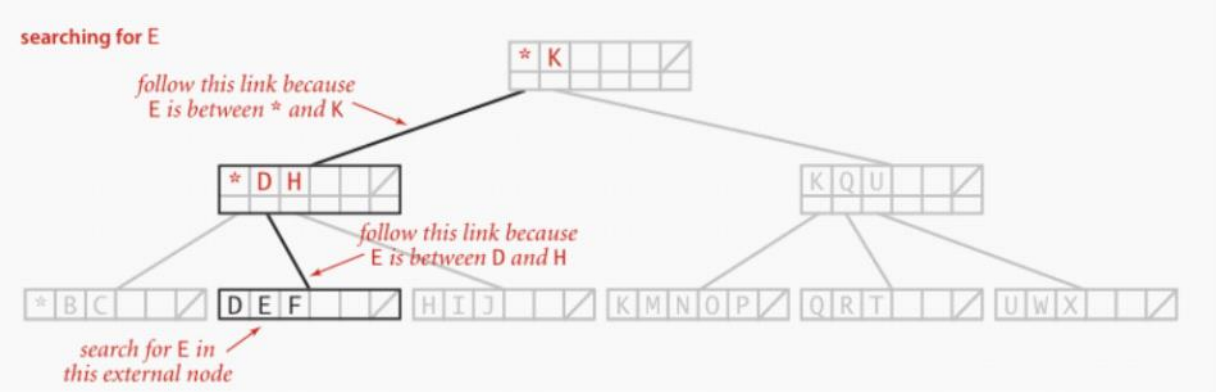
- generalize 2-3 trees by allowing M-1 key link pairs per node
  - at least 2 key link pairs at root
  - at least M/2 key link pairs in other nodes
  - external nodes contain client keys
    - sorted order
    - when inserting into a full external node the node splits
      - splits in two - M/2 - M link pairs
  - internal nodes contain copies of keys to guide search



Screen clipping taken: 29/07/2017 2:48 PM

Search

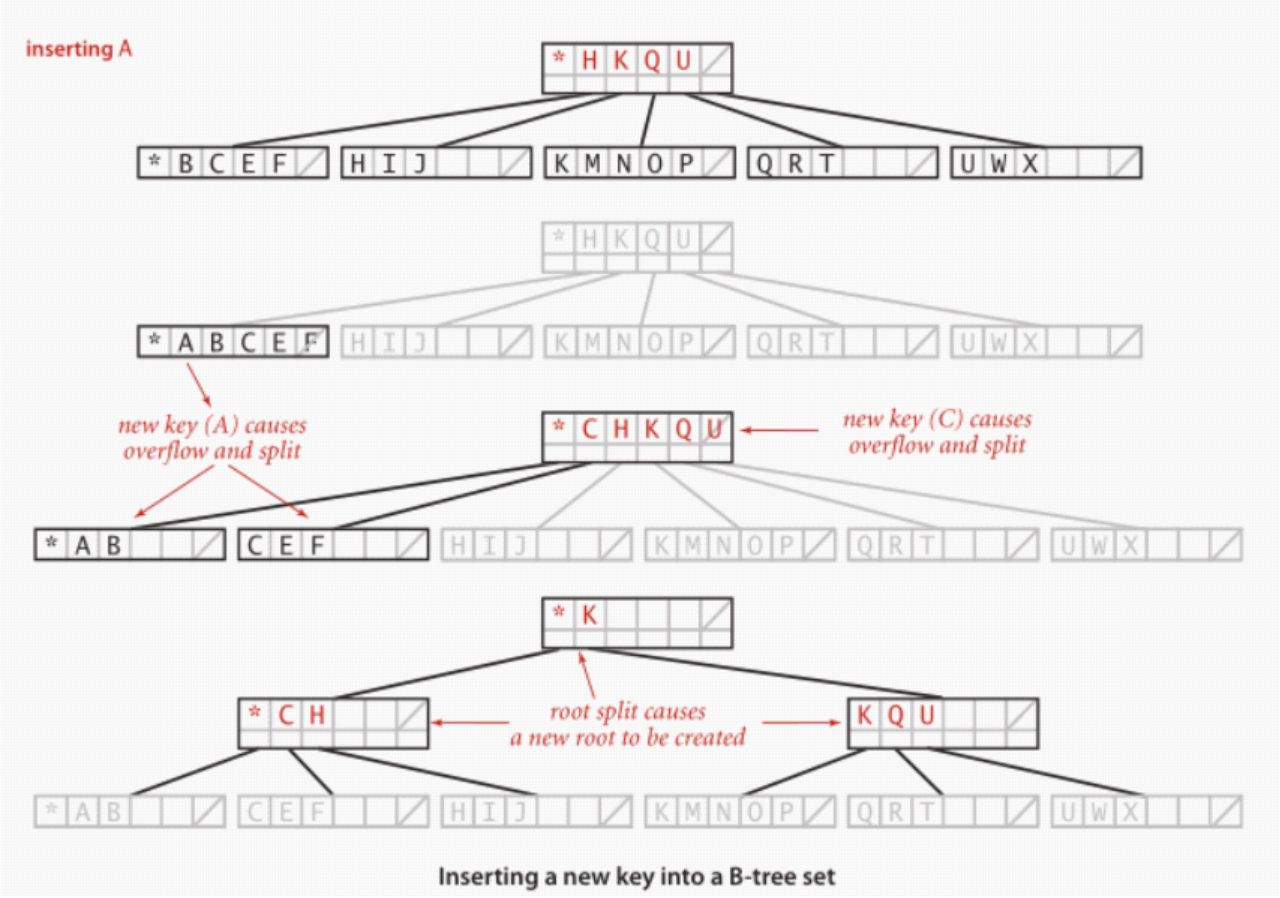
- start at root
- find interval and follow link
- search terminates at an external node



Screen clipping taken: 29/07/2017 2:50 PM

Insert

- search for new key
- insert at bottom
- split nodes with M keys on the way up the tree



Screen clipping taken: 29/07/2017 2:51 PM

Balance in B-trees

- a search or insertion in a B tree with order M with N keys requires between log_M-1(N) and log_M/2(N)
- all internal nodes have between M/2 and M-1 links
- In practice, max probes are 4 or 5

Week 5 - 1d Range Search

July 29, 2017 3:07 PM

Intersections among geometric objects

- how many points in rectangle
- how many intersections between rectangles
- etc

Binary search trees and more are very efficient solutions

Extension of ordered symbol table

- insert key-value pair
- search for key k
- delete key k
- Range search
  - find all keys between k1 and k2
- Range count
  - number of keys between k1 and k2

Geometric Interpretation

- keys are points on line
  - find/count points on the line

Keep keys in a BST

- use rank function
  - **we can compute how many keys between k1 and k2**

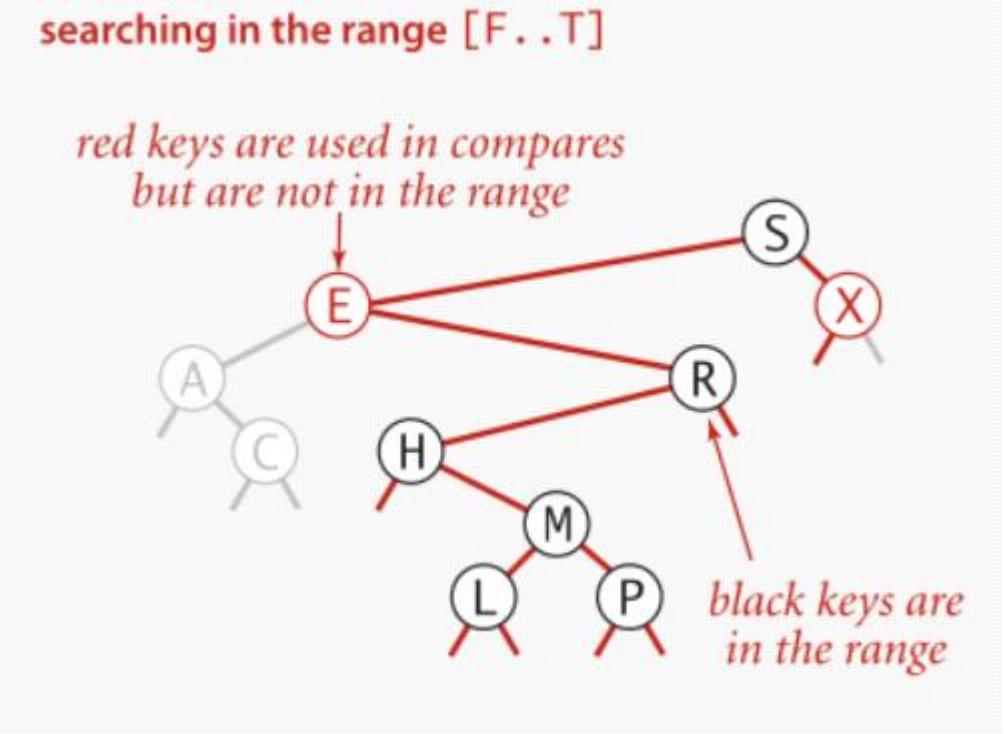
```
public int size(Key lo, Key hi)
{
   if (contains(hi)) return rank(hi) - rank(lo) + 1;
   else              return rank(hi) - rank(lo);
}
```

number of keys < hi

Screen clipping taken: 29/07/2017 3:25 PM

1d Range search

- find all keys between lo and hi
  - recursively find all keys in left subtree(if any fall in range)
  - check key in current node
  - recursively find all keys in right subtree(if any fall in range)



Screen clipping taken: 29/07/2017 3:45 PM

proportional to R+log N

Week 5 - Line Segment Intersection

July 29, 2017 3:54 PM

Orthogonal Line Segment Intersection Search

- all lines are either vertical or horizontal
- find all places where they intersect

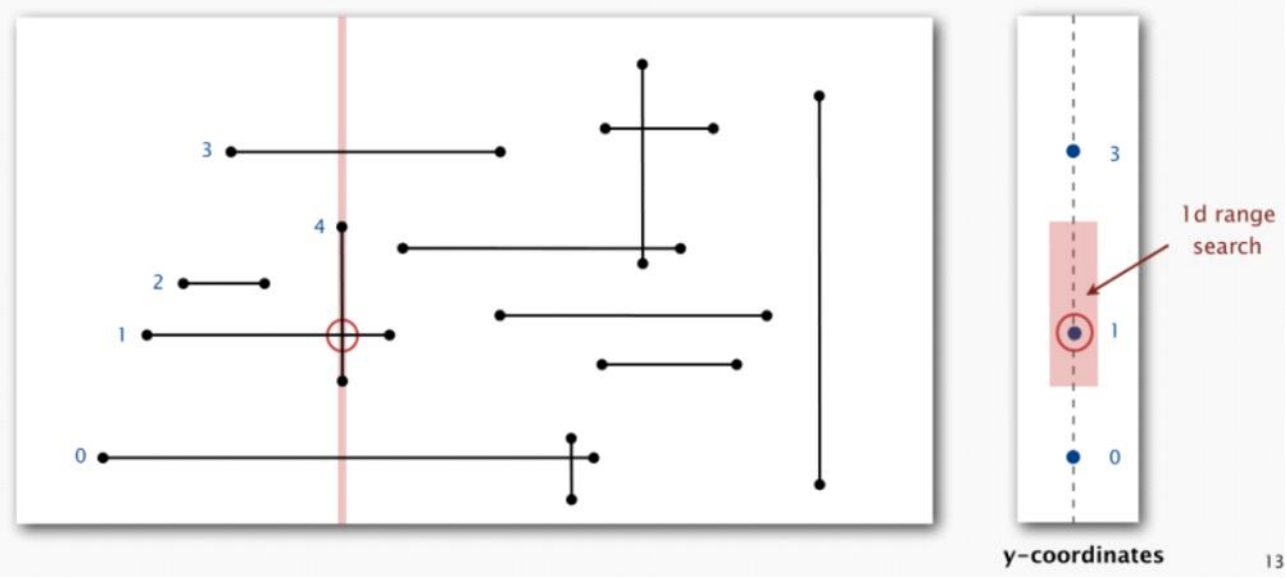
Brute force - quadratic algorithm

- check all pairs of line segments for intersections

*Assumption! all x and y coordinates are distinct

Sweep Line Algorithm

- sweep vertical line from left to right
    - x- coordinates define evets
    - horizontal segment(left endpoint)
        - insert y coordinate into BST
    - horizontal segment(right endpoint)
        - remove y coordinate from BST
    - vertical segment
        - range search for y endpoints



Screen clipping taken: 29/07/2017 4:05 PM

Week 5 - kd Trees

July 29, 2017 4:07 PM

2-d Orthogonal range search

- insert a 2d key
- search for a 2d key
- range search
- range count

Geometric Interpretation

- keys are points
- find/count points in a h-v rectangle

Grid Implementation

- divide space into MxM grid of squares
- create list of points contained in each square
- 2d array to directly index square
- add(x,y) to the corresponding square
- Range search examine only squares that intersect 2d range query
  - deal only with the perimeter of the rectangle
  - all squares fully inside will be counted as in range
- Space - M^2 + N
- Time 1+ N/M^2
  - per square
- We need to manage the size of M
  - M = sqrt(N)

Running Time (assuming data is random)

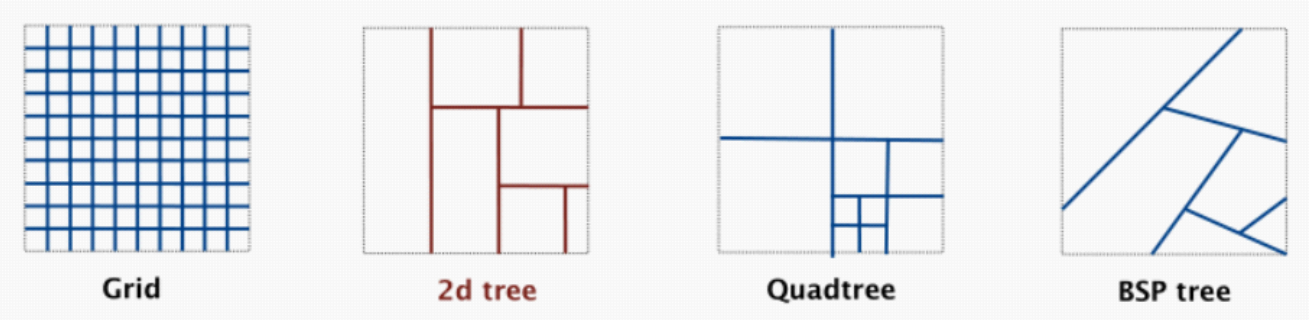
- initialize -N
- Insert - 1
- Range Search - 1 per point in range

Good although often times the points are clustered

- list of points in each square will be too big

Use a tree to represent a recursive subdivision of 2d space

- space partitioning trees

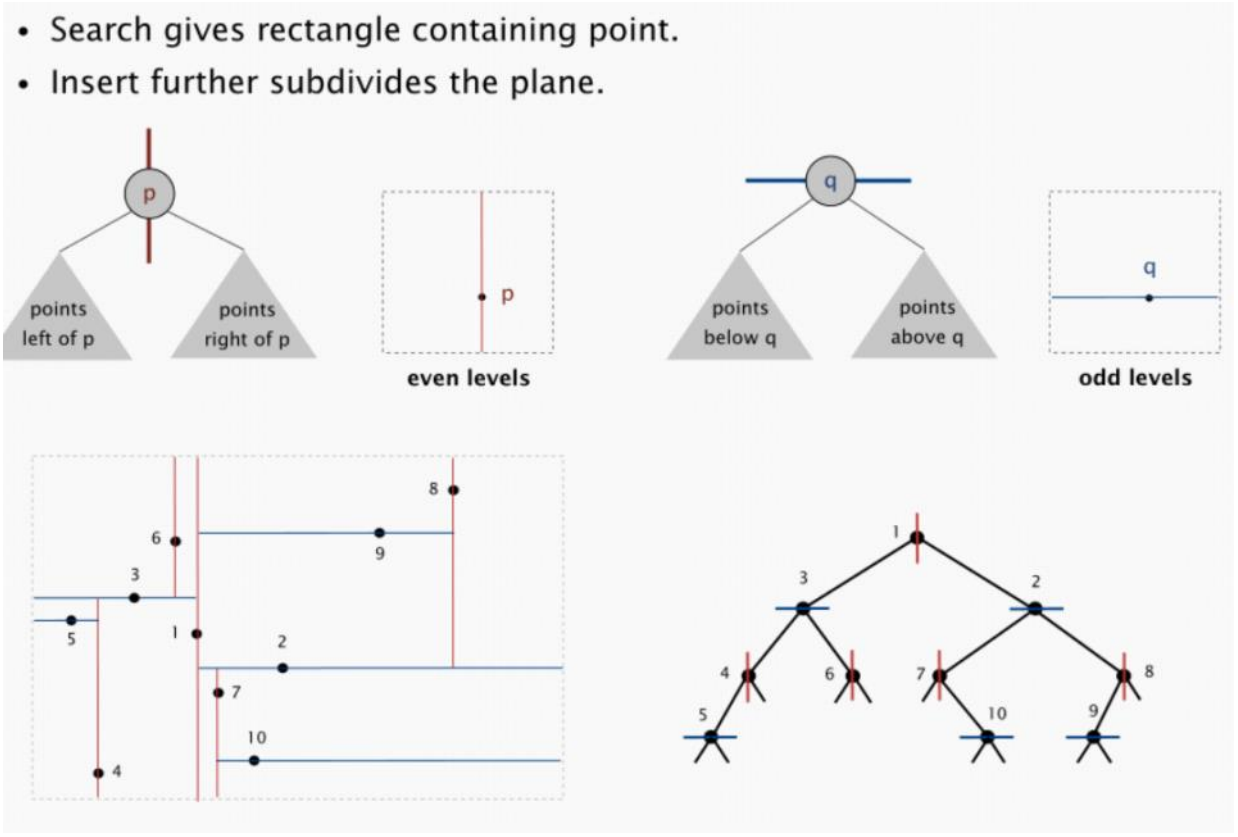


Screen clipping taken: 29/07/2017 6:06 PM

2d Tree

- recursively partitioning the plane
- all the points to the left(tree wise) of a point are left/below(geometrically) and vise versa
- The first node is vertically partitioned
  - 2nd level is horizontally partitioned
  - odd = vertical even = horizontal

Left/below = left child --------- Right/above = right child

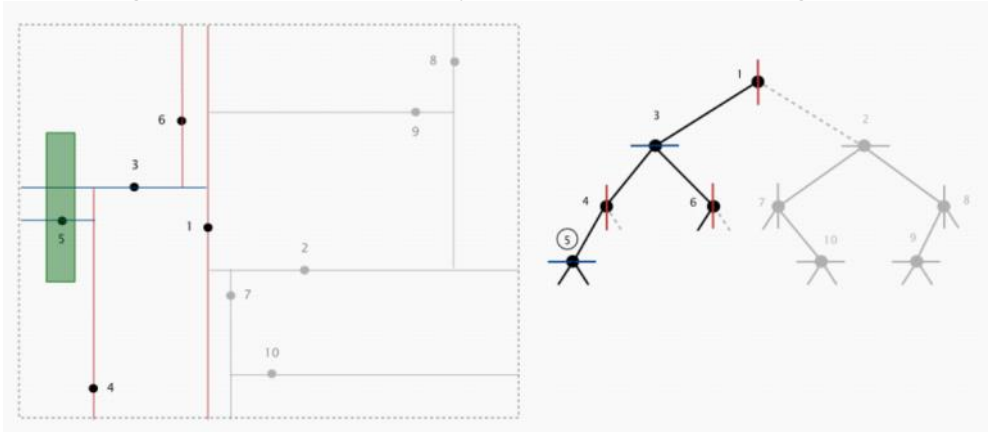


Screen clipping taken: 29/07/2017 6:12 PM

Range Search

- check if point in node lies in rectangle
- recursively search left/bottom
- recursively search right/top

- If the rectangle **does not intersect** a partitioning line (given by the points)
  - we move down the tree
    - if the rectangle is in the area left, we move to the left child
    - if to the right, we move to the right child
- If the rectangle **does intersect** a partitioning line,
  - we have to search both subtrees (start with left)
  - goes down the tree recursively until hits null links following the rules above



Screen clipping taken: 29/07/2017 6:18 PM

Running Time

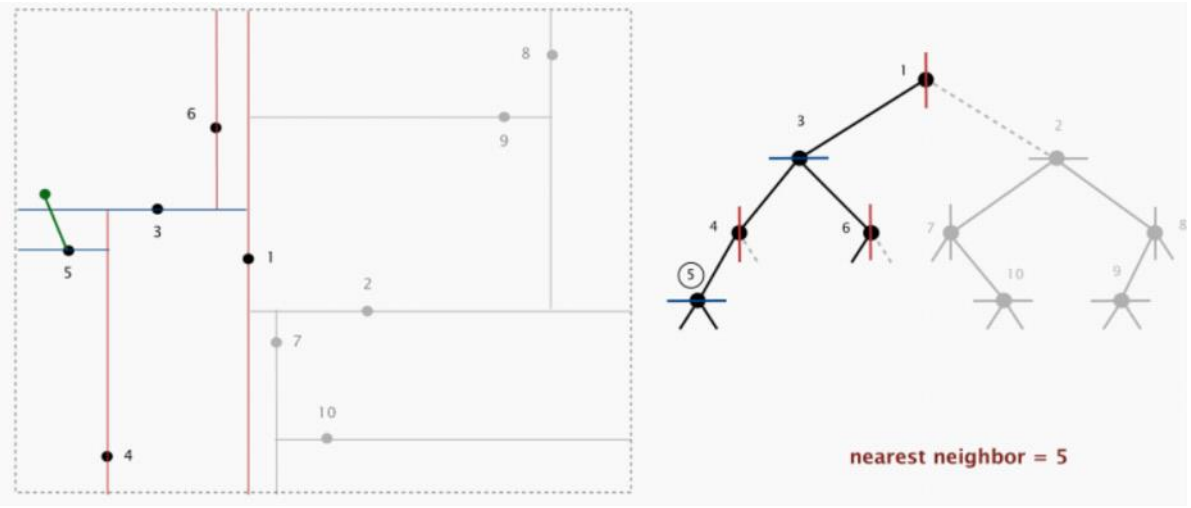
- proportional to R+logN (typically)
- R+sqrt(N) (worst case)

Nearest Neighbor Search

- find closest point to query point

- check distance from point to query point
- recursively search left/bottom
- recursively search right/top
- method begins by searching for query point

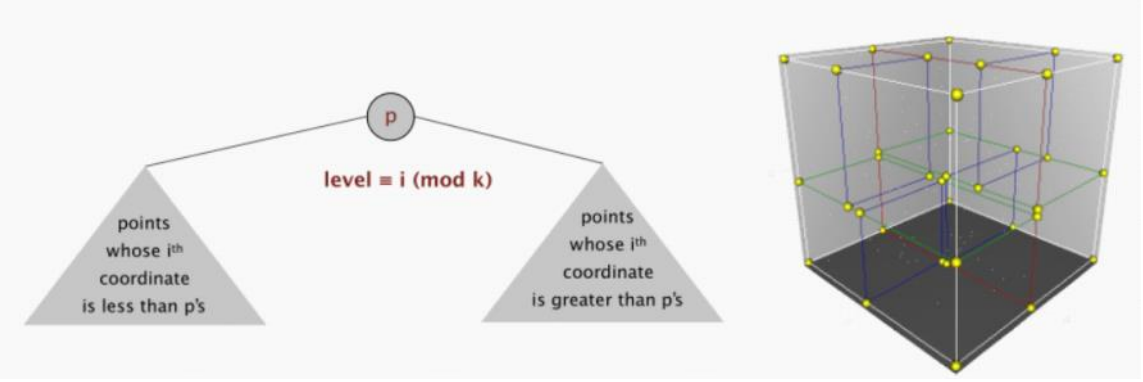
- we move toward the query point first
- if a new closest point is found
    - it becomes the new "champion"



Screen clipping taken: 29/07/2017 6:48 PM

Running Time
- Typical Case
    - log N
- Worst Case
    - N

Expands to multiple dimensions (kd tree - k dimensions)
- partition k dimensions
    - 1 dimension at a time



nm

Screen clipping taken: 29/07/2017 6:56 PM

Week 5 - Interval Search Trees

July 29, 2017 7:01 PM

1d Interval search

- instead of point objects we have intervals

We need to support the operations:

- insert
- search
- delete
- interval intersection query
    - given an interval find all intervals that intersects that interval

*Assumption!

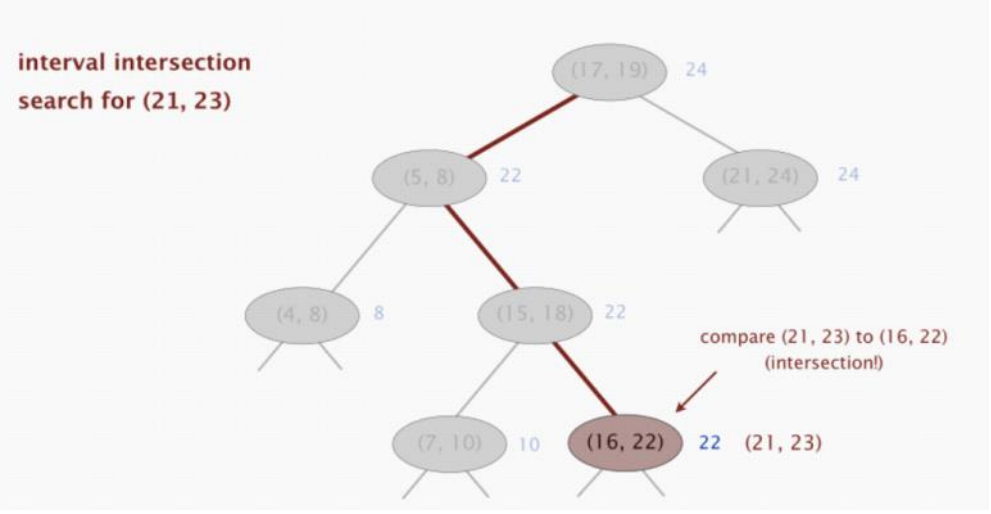
- no two intervals have the same left endpoint

Search Tree Data Structure

- use left endpoint as BST key
- store max endpoint in subtree rooted at node

Insert

- insert using lo as key
- update max in each node



Screen clipping taken: 29/07/2017 7:08 PM

Search (any interval that intersects query)

- if interval in node intersects query interval
    - return
- if left subtree is null go right
- if max endpoint in left subtree is less than lo go right
- else go left
- until hit null at right

Implementation

```
Node x = root;
while (x != null)
{
   if      (x.interval.intersects(lo, hi)) return x.interval;
   else if (x.left == null)                x = x.right;
   else if (x.left.max < lo)               x = x.right;
   else                                    x = x.left;
}
return null;
```

Screen clipping taken: 29/07/2017 7:14 PM

Analysis - use red black to guarantee performance

operation	brute	interval search tree	best in theory
insert interval	1	log N	log N
find interval	N	log N	log N
delete interval	N	log N	log N
find any one interval that intersects (lo, hi)	N	log N	log N
find all intervals that intersects (lo, hi)	N	R log N	R + log N

order of growth of running time for N intervals

Screen clipping taken: 29/07/2017 7:16 PM

Week 5- Rectangle Intersection

July 29, 2017 7:16 PM

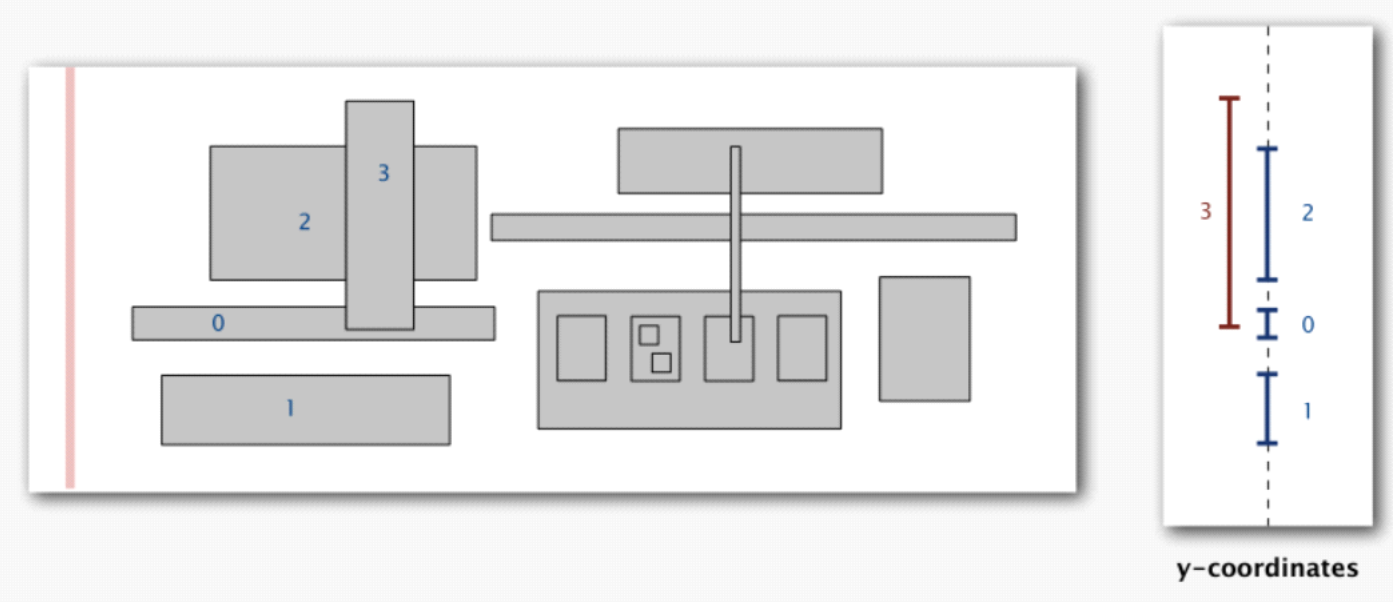
Goal: find all intersections among a set of N orthogonal rectangles

- Brute force
  - quadratic

*Assumption! - all coordinates are district

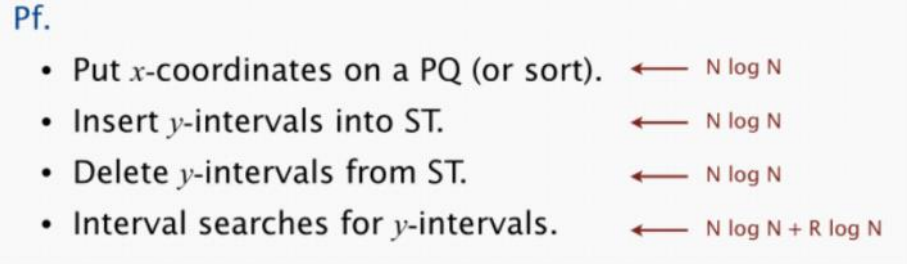
Sweep vertical line from left to right

- x coordinate of interval defines events
- maintain set of rectangles the intersect the sweep line in an interval search tree(using y coordinate)
- left endpoint (vertical line)
  - interval search for for y-interval of rectangle
  - insert y- interval
- right endpoint(vertical line)
  - remove y interval



Screen clipping taken: 29/07/2017 7:25 PM

Analysis



Screen clipping taken: 29/07/2017 7:26 PM

Week 5 - Programming assignment

August 3, 2017 11:19 AM

in kdtree if we define being colinear as greater, make sure to make this consistent through the program

- in this case it was the put method

check for an already existing point being inserted again

Week 6 - Hash Tables

August 4, 2017 12:13 PM

Way to implement symbol tables

- we can do better than R-B tree
  - different access
  - no ordered operations

Hashing

- Save items in a key indexed table (index is a function of the key)
- Hash function - method for computing array from key

Issues

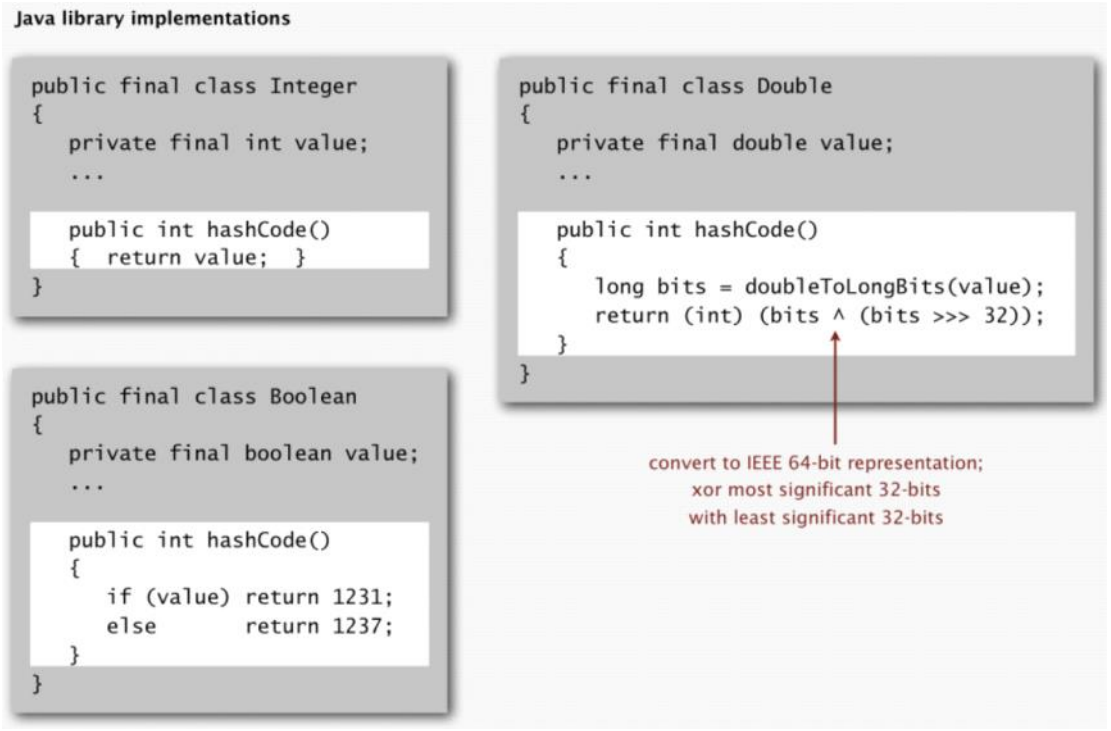
- computing the has function
- equality test - method for checking whether two keys are equal
- Collision Resolution
  - two values hash to the same array index

Implementation

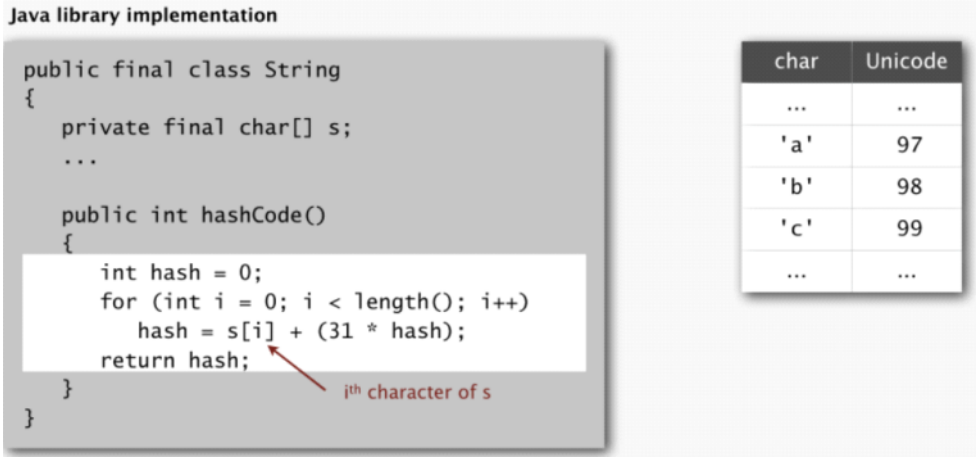
- ideally
  - scramble keys uniformly to produce a table index
- efficiently computable
- each table equally likely for each key
- need a different approach for each key type

- all java classes have a method hashCode()
  - returns a 32 bit int
- REQUIREMENT
  - if e.equals(y) then (x.hashCode() == y.hashCode()
- Highly Desirable
  - if !x.equals(y) then (x.hashCode() != y.hashCode())
- Default implementation
  - Memory address

Implementing Hash Code: integers, booleans and doubles



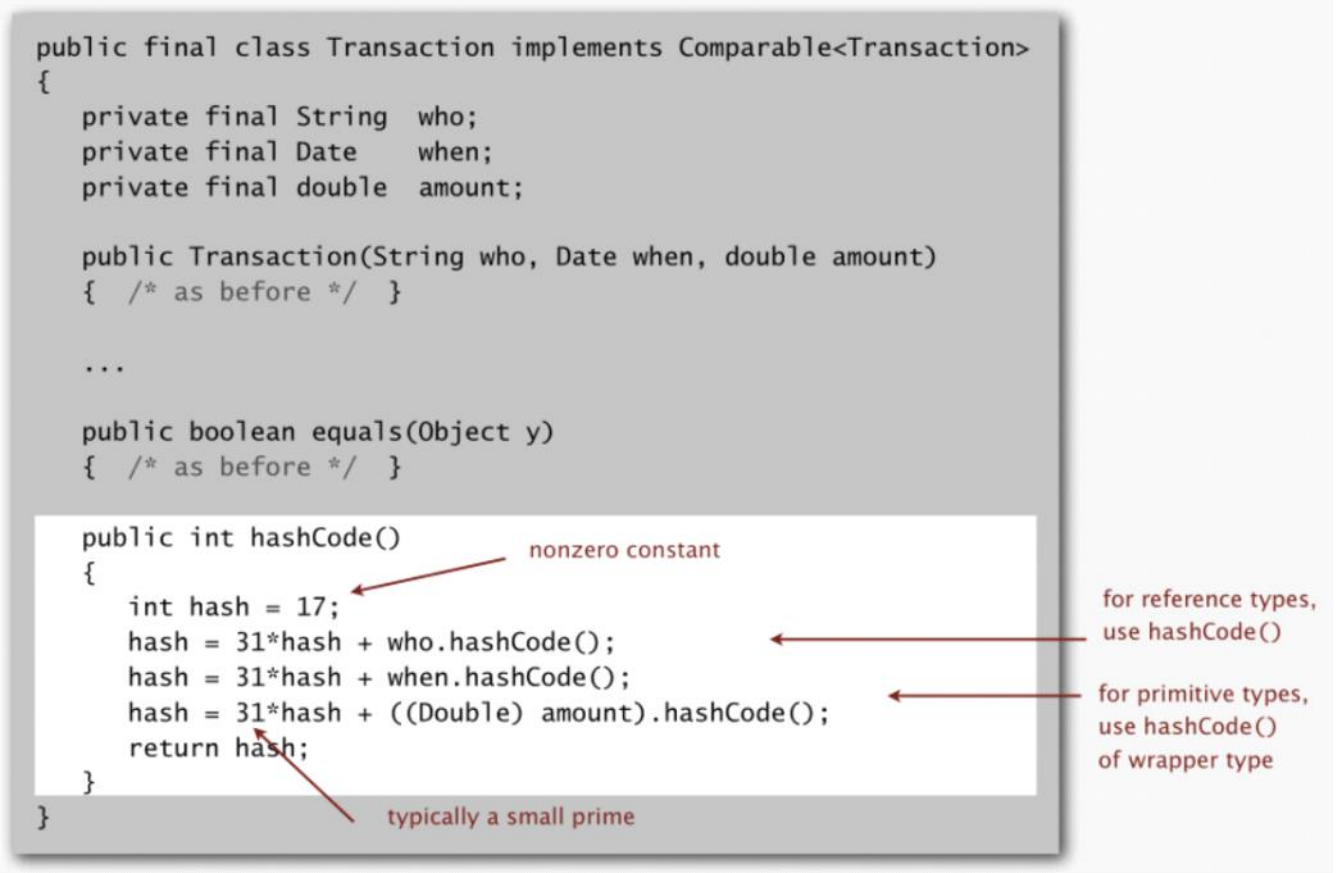
Screen clipping taken: 04/08/2017 12:40 PM



Screen clipping taken: 04/08/2017 12:41 PM

- treats string as large number
- Horner's method
  - h = s[0] * 31^L-1 + … s[L-3] * 31^2 + s[L-2]*31 +s[L-1] * 31^0
- you want to involve all the characters in the string
- if the object is immutable, store the value as an instance variable
  - don't have to recomputed

Example of User defined types



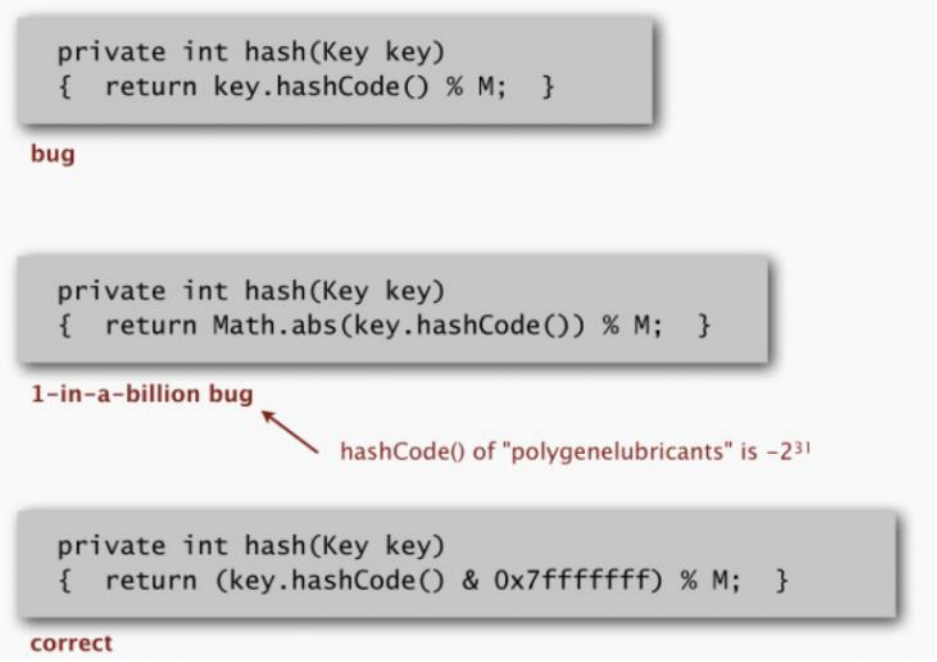
Screen clipping taken: 04/08/2017 12:44 PM

- standard recipe
  - 31x+y rule to compute fields
  - if primitive use wrapper type hashCode()
  - null - return 0
  - reference type - use hashCode() - applies the method recursively
  - array - use Arrays.deepHashCode()

In theory, - universal hash functions exist
- equal chance of each position

Basic Rule - Need to use the whole key to compute hashcode
- consult an expert for state of the art codes

Modular Hashing
- hash code gives an integer from -2^31 and (2^31)-1
- a hash function needs an int from 0 - M-1 (for table sized M)
  - M is typically a power of 2 or a prime number
  - this is because how we get a hash function from a hash code is to do mod(M)
    - need to take abs(hashCode) - can't have a negative array index
    - still has a bug



Screen clipping taken: 04/08/2017 2:52 PM

- use hashCode make positive and then mod - abs value has a small bug

Uniform Hashing Assumption

- each key is equally likely to hash to an integer between 0 - M-1
- able to come close to this

Bins and Balls Example

- throw balls uniformly at random into M bins
- Birthday problem
  - subset of this problem
  - expect two balls in the same bin after sqrt(pi*M/2) tosses
- Coupon Collector
  - expect every bin to have at least 1 ball after M lnM tosses
- Load Balancing
  - after M tosses most loaded bin has (logM/log(logM)) balls

Week 6 - Separate Chaining

August 4, 2017 3:29 PM

Collision Resolution

- Collision - two distinct keys hashing to the same index
- Birthday problem- we will have collisions

The challenge is to deal with challenges efficiently

Separate Chaining

- build a linked list for each of the table positions
  - hash - map key to integer i between 0-(M-1)
  - Insert put in front of i^{th} chain
  - search - only need to search the i^{th} chain

Implementation

```
public class SeparateChainingHashST<Key, Value>
{
   private int M = 97;                // number of chains
   private Node[] st = new Node[M];   // array of chains

   private static class Node
   {
      private Object key;
      private Object val;
      private Node next;
      ...
   }

   private int hash(Key key)
   {  return (key.hashCode() & 0x7fffffff) % M;  }

   public Value get(Key key) {
      int i = hash(key);
      for (Node x = st[i]; x != null; x = x.next)
         if (key.equals(x.key)) return (Value) x.val;
      return null;
   }

}
```

array doubling and halving code omitted

no generic array creation (declare key and value of type Object)

Screen clipping taken: 05/08/2017 3:00 AM

- another possibility is to make the hash table bigger as more values get inserted
  - simple implementation
  - not discussed here

Insertion Implementation

```
public void put(Key key, Value val) {
   int i = hash(key);
   for (Node x = st[i]; x != null; x = x.next)
      if (key.equals(x.key)) { x.val = val; return; }
   st[i] = new Node(key, val, st[i]);
}
```

Screen clipping taken: 05/08/2017 3:02 AM

Analysis

- using uniform hashing assumption,
  - number of keys in list is within a constant factor of N/M
    - extremely close to 1
- Make M = N/5
  - M too large - too many empty chains
  - M too small - chains too long

Summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	equals()
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	compareTo()
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	compareTo()
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	compareTo()
separate chaining	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	equals() hashCode()

* under uniform hashing assumption

Screen clipping taken: 05/08/2017 3:05 AM

Week 6 - Linear Probing

August 5, 2017 3:05 AM

- another popular collision detection method

- open addressing
  - when a new key collides find next empty slot and put it there
  - array size will need to be larger than the amount of keys

Insert
- put at table index i if free
  - if not try i+1, i+2, etc
  - until hit empty position
- if run to the end of the array
  - run off to index 0

Search
- search table index i
  - if free search hit
  - if not try i+1 i+2 etc
  - until hit empy position

Array size must be greater than the number of key-value pairs
- need array resizing
- at least twice as big

Implementation

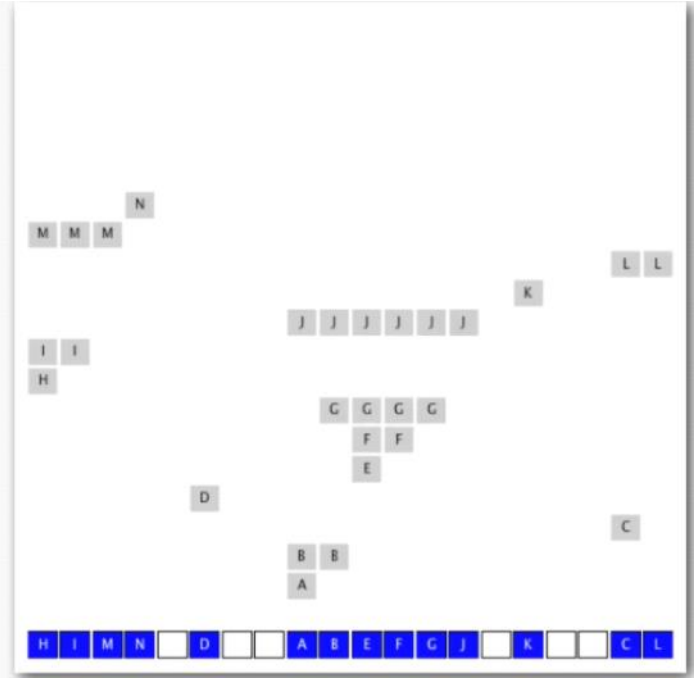
```
public class LinearProbingHashST<Key, Value>
{
   private int M = 30001;
   private Value[] vals = (Value[]) new Object[M];
   private Key[]   keys = (Key[])   new Object[M];

   private int hash(Key key) {  /* as before */  }

   public void put(Key key, Value val)
   {
      int i;
      for (i = hash(key); keys[i] != null; i = (i+1) % M)
         if (keys[i].equals(key))
             break;
      keys[i] = key;
      vals[i] = val;
   }

   public Value get(Key key)
   {
      for (int i = hash(key); keys[i] != null; i = (i+1) % M)
         if (key.equals(keys[i]))
             return vals[i];
      return null;
   }
}
```

array doubling and halving code omitted

Screen clipping taken: 05/08/2017 3:15 AM

Cluster

- new keys likely to hash into middle of big clusters



Screen clipping taken: 05/08/2017 3:31 AM

- as more values are inserted, large clusters form and clusters merge together

Knuth's parking problem
cars show up at random times to find a spot- if they can not find one, they look to the next spot (models clustering)

- with M/2 spaces mean displacement is 3/2
- M cars mean displacement is sqrt(pi(m)/8)

Proposition. Under uniform hashing assumption, the average # of probes in a linear probing hash table of size M that contains $N = \alpha M$ keys is:

$$\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \quad \text{search hit}$$

$$\sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right) \quad \text{search miss / insert}$$

Screen clipping taken: 05/08/2017 3:35 AM

typical choice alpha = N/M = 0.5

Summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	equals()
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	compareTo()
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	compareTo()
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	compareTo()
separate chaining	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	equals() hashCode()
linear probing	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	equals() hashCode()

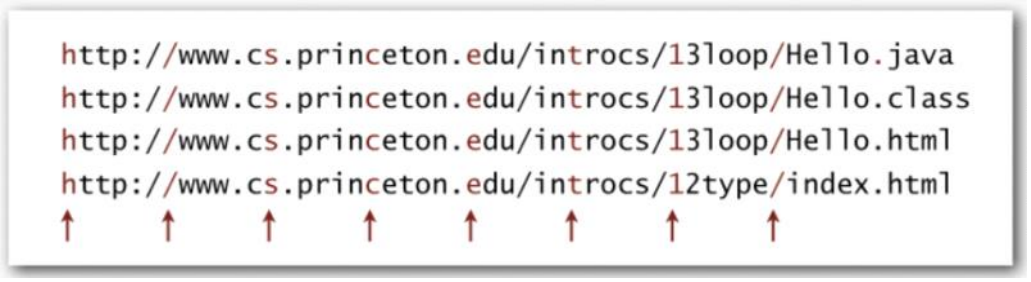
* under uniform hashing assumption

Screen clipping taken: 05/08/2017 3:37 AM

Week 6 - Hash Table Context

August 5, 2017 3:40 AM

- originally in java, they only used every 8th key when computing the hash function
  - makes it easier to hash long strings
  - great potential for bad collision patterns
  - some strings have the same character every 8th characther (ie. URLs)



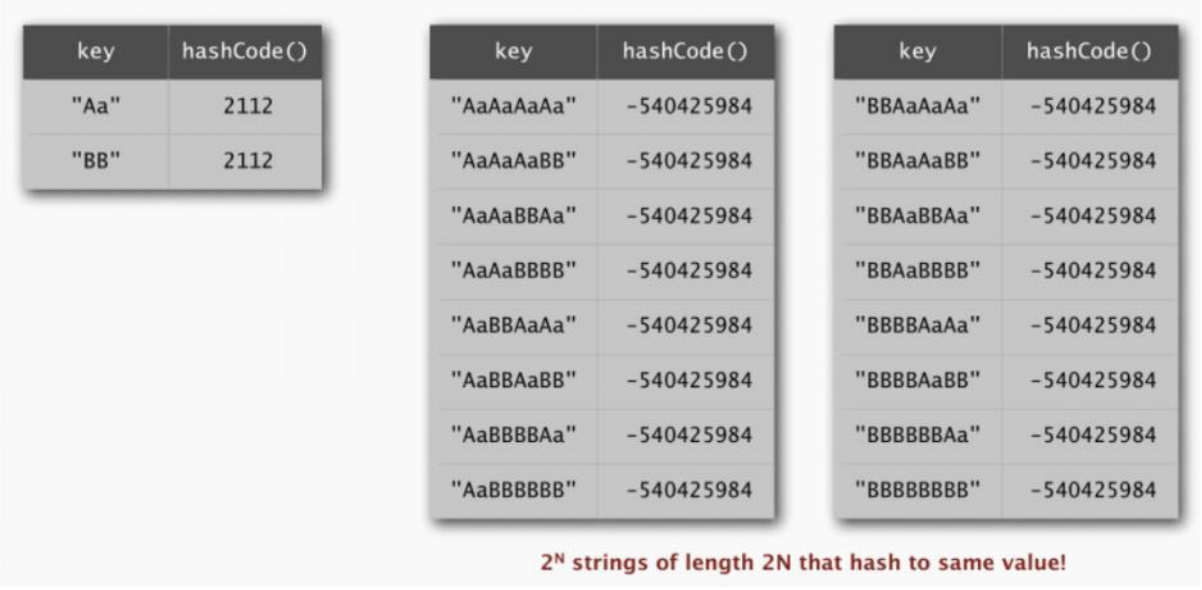
Screen clipping taken: 05/08/2017 3:42 AM

Important to use every character when computing hash functions

Uniform hashing assumption is important when we need to guarantee performance

- sometimes R-B search trees is better if you can't guarantee uniform hashing assumptions

- hackers can DDOS attack your service
  - if they know your hash code - they can exploit cracks in your hash function



key	hashCode()
"Aa"	2112
"BB"	2112

key	hashCode()
"AaAaAaAa"	-540425984
"AaAaAaBB"	-540425984
"AaAaBBAa"	-540425984
"AaAaBBBB"	-540425984
"AaBBAaAa"	-540425984
"AaBBAaBB"	-540425984
"AaBBBBAa"	-540425984
"AaBBBBBB"	-540425984

key	hashCode()
"BBAaAaAa"	-540425984
"BBAaAaBB"	-540425984
"BBAaBBAa"	-540425984
"BBAaBBBB"	-540425984
"BBBBAaAa"	-540425984
"BBBBAaBB"	-540425984
"BBBBBBAa"	-540425984
"BBBBBBBB"	-540425984

2^N strings of length 2N that hash to same value!

Screen clipping taken: 05/08/2017 3:47 AM

One way hash functions

- "hard" to find a key that will hash to a desired value (or two keys to the same value)

```
String password = args[0];
MessageDigest sha1 = MessageDigest.getInstance("SHA1");
byte[] bytes = sha1.digest(password);

/* prints bytes as hex string */
```

Screen clipping taken: 05/08/2017 3:48 AM

Separate Chaining vs Linear Probing

Separate Chaining	Linear Probing
easier to implement delete	less wasted space
performance degrades gracefully	better cache performance
clustering less sensitive to poorly designed hash function	

Improved Versions

- two probe hashing
  - hash to two positions - insert in shorter of the two chains
  - reduces expected length of the longest chain to log log N
- double hashing
  - linear probing - skip a variable amount
    - wipes out clustering
  - can allow table to become nearly full
  - more difficult to implement delete
- cuckoo hashing
  - hash to two positions
    - insert key into either position
      - if occupied reinsert displaced key into its alternative position
  - constant worst time for search

Hash Tables vs Balanced Search Trees

Hash Tables	Balanced Search Trees
Simpler to code	Stronger performance guarantee
no effective alternative for unordered keys	Support for ordered ST operations
faster for simple keys (simple arithmetic vs log N compares)	Easier to implement compareTo than equals and hashcode
better system support in Java for strings	

Java includes Both

- R-B search trees
  - java.util.TreeMap
  - java.util.TreeSet
- Hash Tables
  - java.util.HashMap
  - java.util.IdentifyHashMap

Week 6 - Symbol Table Applications - Sets

August 5, 2017 4:00 AM

Mathematical Set

- collection of distinct keys

API

public class SET<Key extends Comparable<Key>>	
SET()	*create an empty set*
void add(Key key)	*add the key to the set*
boolean contains(Key key)	*is the key in the set?*
void remove(Key key)	*remove the key from the set*
int size()	*return the number of keys in the set*
Iterator<Key> iterator()	*iterator through keys in the set*

Screen clipping taken: 05/08/2017 4:01 AM

Implementation

- take symbol table implementation
  - remove anything that deals with values

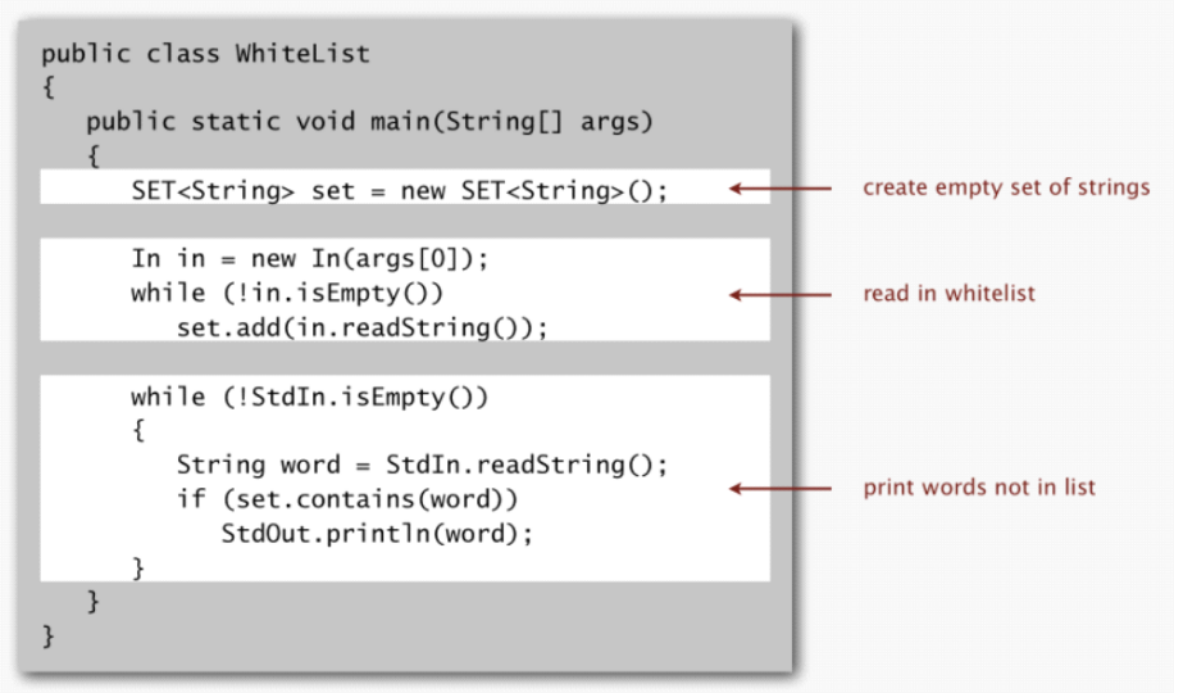
Examples of Sets

- Exception Filter
  - read in a list of words from one file
    - print out all words from standard input that are not in(or in) the list
  - whitelist
    - prints all the words that are in the list
  - black list
    - prints the words not in the list

application	purpose	key	in list
spell checker	identify misspelled words	word	dictionary words
browser	mark visited pages	URL	visited pages
parental controls	block sites	URL	bad sites
chess	detect draw	board	positions
spam filter	eliminate spam	IP address	spam addresses
credit cards	check for stolen cards	number	stolen cards

Screen clipping taken: 05/08/2017 4:07 AM

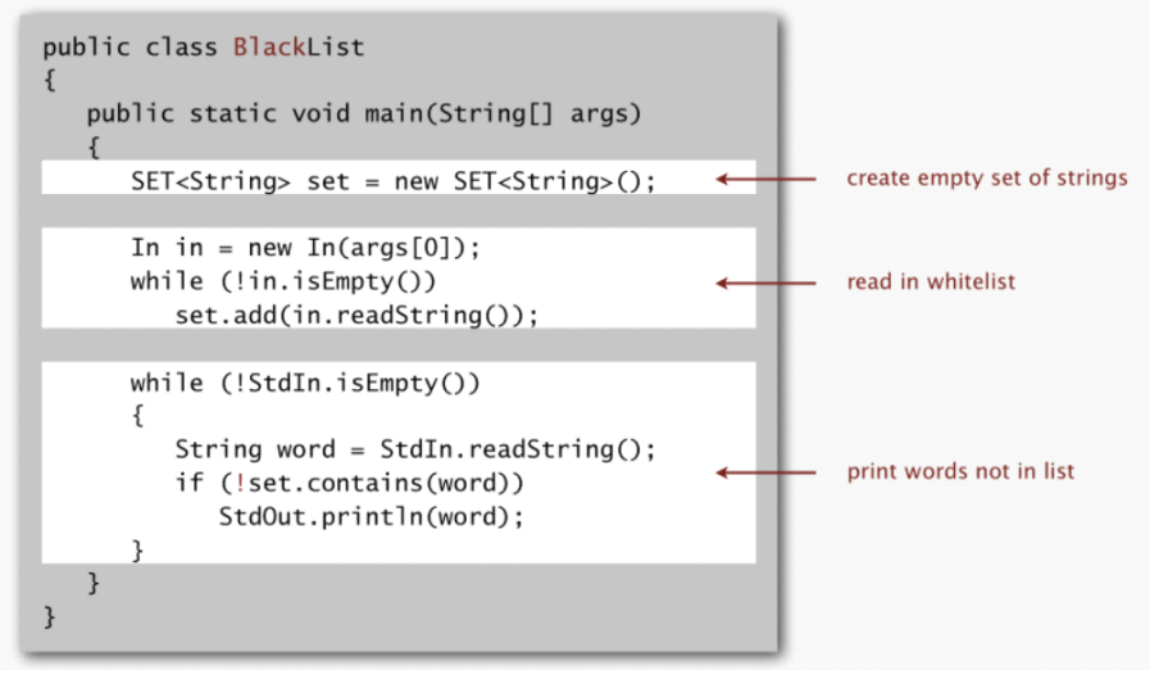
Implementation (whitelist)



Screen clipping taken: 05/08/2017 4:08 AM

- whitelist prints all words that are in the exceptional list

Implementation(blacklist)



Screen clipping taken: 05/08/2017 4:09 AM

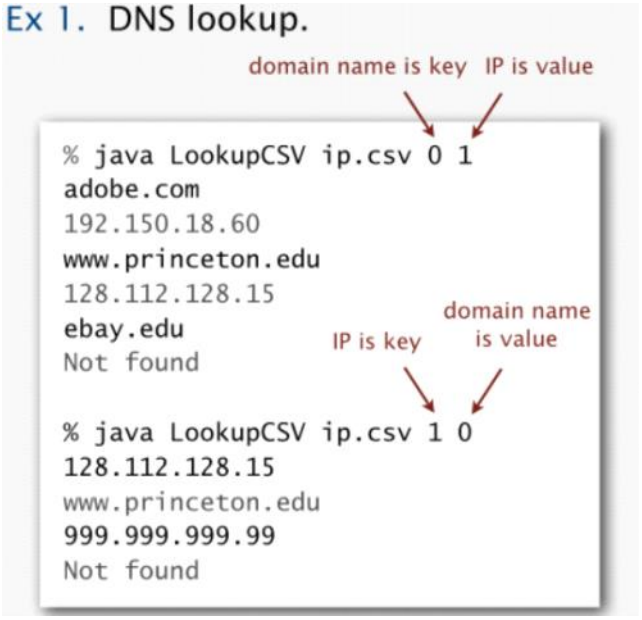
- black list prints all words that are not in the exceptional list

Week 6 - Symbol Table Applications - Dictionary Client

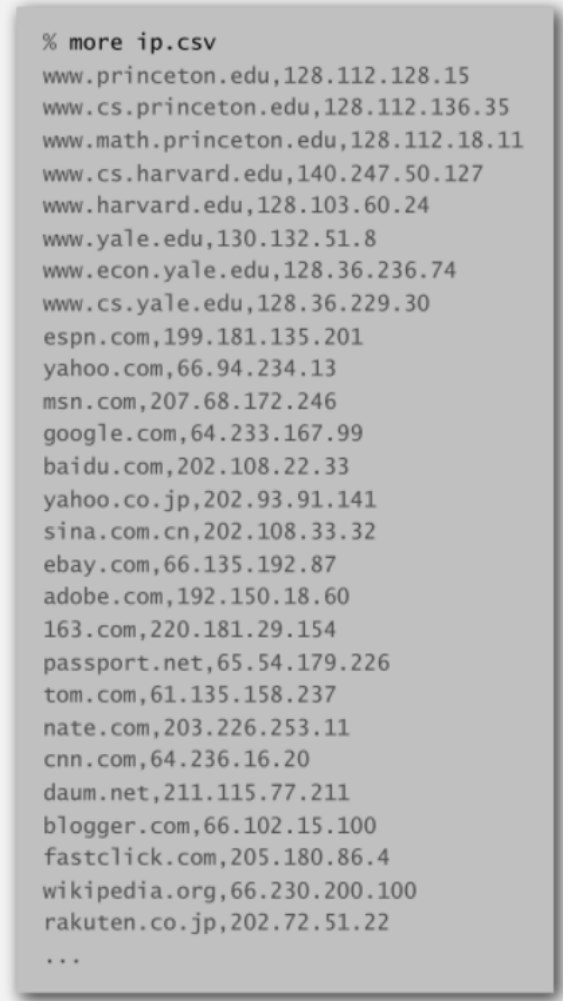
August 5, 2017 4:10 AM

Command line arguments

- CSV file
- key field
- value field

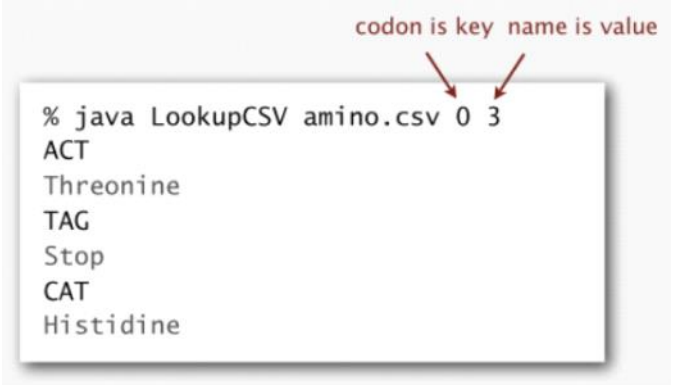


Screen clipping taken: 05/08/2017 4:12 AM



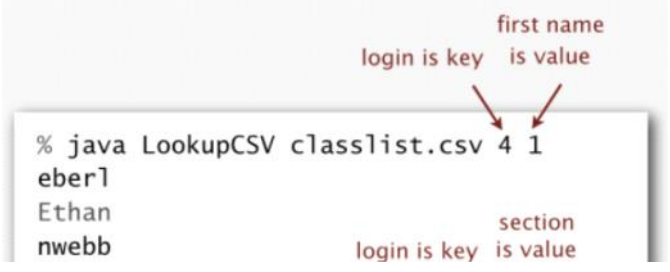
Screen clipping taken: 05/08/2017 4:13 AM

Ex. 2 Amino Acids

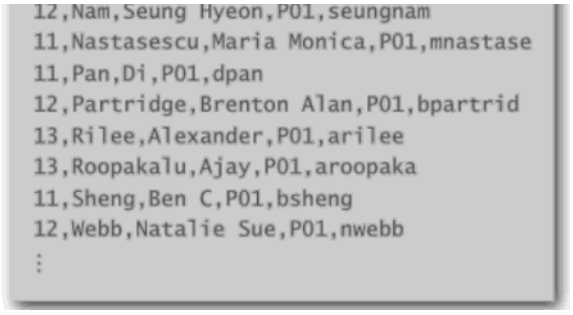


Screen clipping taken: 05/08/2017 4:14 AM

Ex. 3 Class List

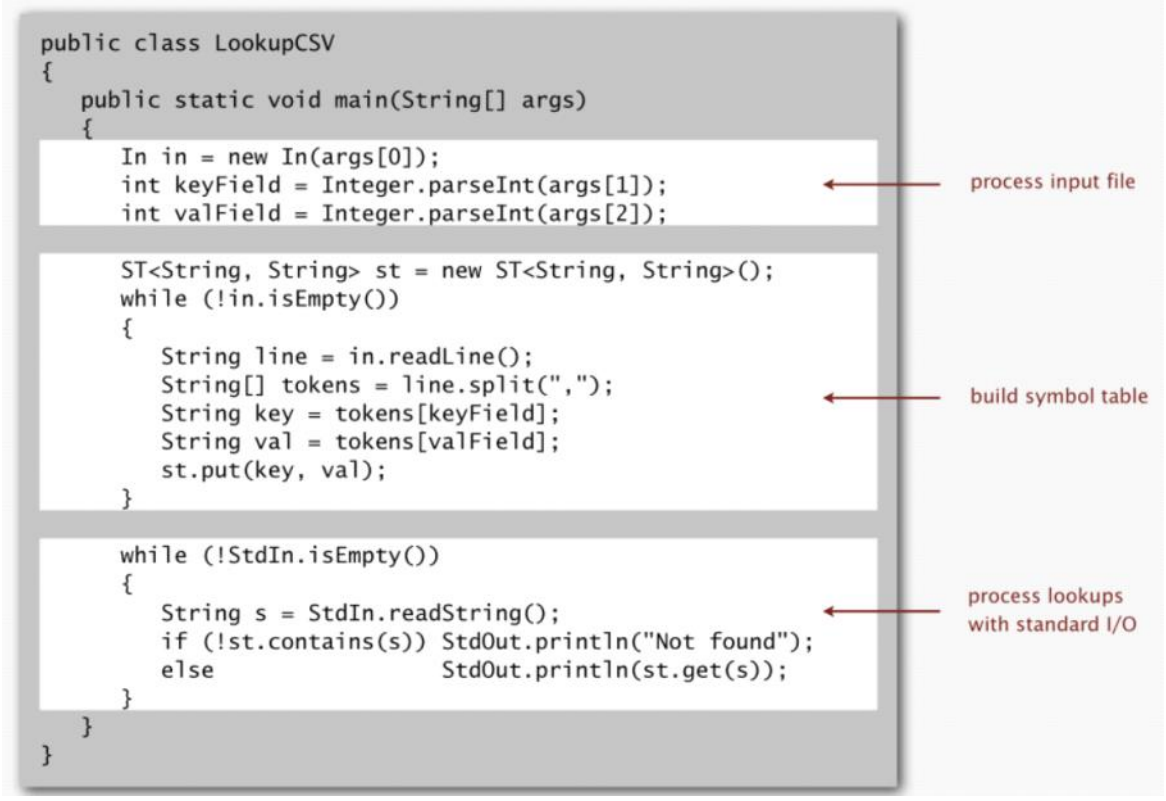


Screen clipping taken: 05/08/2017 4:15 AM



Screen clipping taken: 05/08/2017 4:15 AM

Implementation



Screen clipping taken: 05/08/2017 4:16 AM

Week 6 - Symbol Table Applications - Indexing Clients

August 5, 2017 4:17 AM

We may need to index our pc

- allows us to search files in our computer

Goal: Given a list of files specified, create an index so that you can efficiently find all files containing a given query string

```
% ls *.txt
aesop.txt magna.txt moby.txt
sawyer.txt tale.txt

% java FileIndex *.txt

freedom
magna.txt moby.txt tale.txt

whale
moby.txt

lamb
sawyer.txt aesop.txt
```

```
% ls *.java
BlackList.java Concordance.java
DeDup.java FileIndex.java ST.java
SET.java WhiteList.java

% java FileIndex *.java

import
FileIndex.java SET.java ST.java

Comparator
null
```

Solution. Key = query string; value = set of files containing that string.

Screen clipping taken: 05/08/2017 4:19 AM

Implementation

```
import java.io.File;
public class FileIndex
{
   public static void main(String[] args)
   {
      ST<String, SET<File>> st = new ST<String, SET<File>>();

      for (String filename : args) {
         File file = new File(filename);
         In in = new In(file);
         while (!in.isEmpty())
         {
            String key = in.readString();
            if (!st.contains(key))
               st.put(key, new SET<File>());
            SET<File> set = st.get(key);
            set.add(file);
         }
      }

      while (!StdIn.isEmpty())
      {
         String query = StdIn.readString();
         StdOut.println(st.get(query));
      }
   }
}
```

symbol table

list of file names from command line

for each word in file, add file to corresponding set

process queries

18

Screen clipping taken: 05/08/2017 4:21 AM

Concordance

- get context for words before and after search word (in books and text)

ie

```
% java Concordance tale.txt
cities
tongues of the two *cities* that were blended in

majesty
their turnkeys and the *majesty* of the law fired
me treason against the *majesty* of the people in
of his most gracious *majesty* king george the third

princeton
no matches
```

Screen clipping taken: 05/08/2017 4:24 AM

Implementation

```
public class Concordance
{
   public static void main(String[] args)
   {
      In in = new In(args[0]);
      String[] words = in.readAllStrings();
      ST<String, SET<Integer>> st = new ST<String, SET<Integer>>();
      for (int i = 0; i < words.length; i++)
      {
         String s = words[i];
         if (!st.contains(s))
            st.put(s, new SET<Integer>());
         SET<Integer> set = st.get(s);
         set.add(i);
      }

      while (!StdIn.isEmpty())
      {
         String query = StdIn.readString();
         SET<Integer> set = st.get(query);
         for (int k : set)
             // print words[k-4] to words[k+4]
      }
   }
}
```

read text and build index

process queries and print concordances

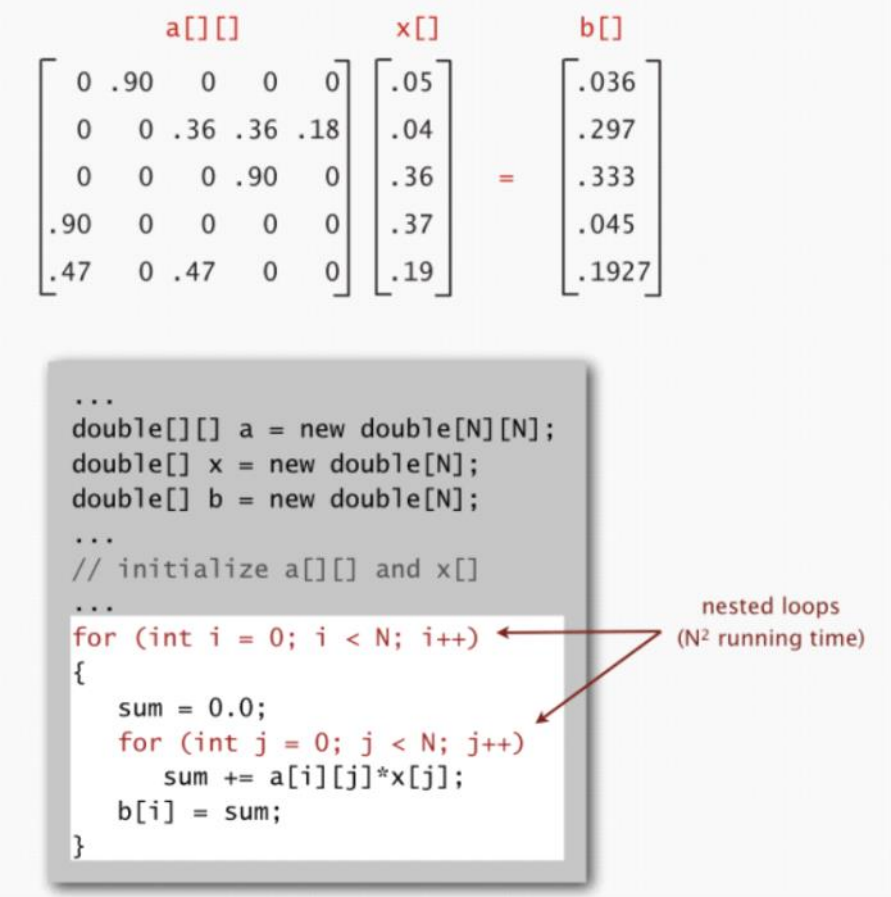
Screen clipping taken: 05/08/2017 4:24 AM

Week 6 - Symbol Table Applications - Sparse Vectors

August 5, 2017 4:26 AM

Matrix- Vector Multiplication

- standard implementation:



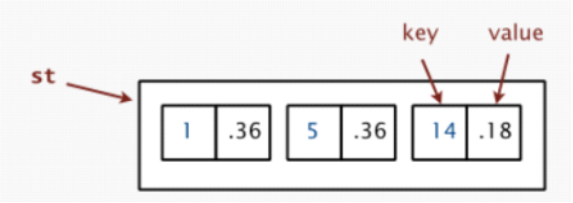
Screen clipping taken: 05/08/2017 4:28 AM

This implementation is fine for full/almost full matrices but not efficient for sparse matrices(because of the nested for loop)

Symbol tables solve this

Standard implementation of vectors

- array of size N - where N is the dimensions of the vector
- instead, symbol table representation
  - key = index, value = entry
  - efficient iterator
  - space proportional to number of nonzero



Screen clipping taken: 05/08/2017 4:32 AM

Implementation:

```
public class SparseVector
{
   private HashST<Integer, Double> v;

   public SparseVector()
   {  v = new HashST<Integer, Double>();  }

   public void put(int i, double x)
   {  v.put(i, x);  }

   public double get(int i)
   {
      if (!v.contains(i)) return 0.0;
      else return v.get(i);
   }

   public Iterable<Integer> indices()
   {  return v.keys();  }

   public double dot(double[] that)
   {
       double sum = 0.0;
       for (int i : indices())
           sum += that[i]*this.get(i);
       return sum;
   }
}
```

HashST because order not important

empty ST represents all 0s vector

a[i] = value

return a[i]

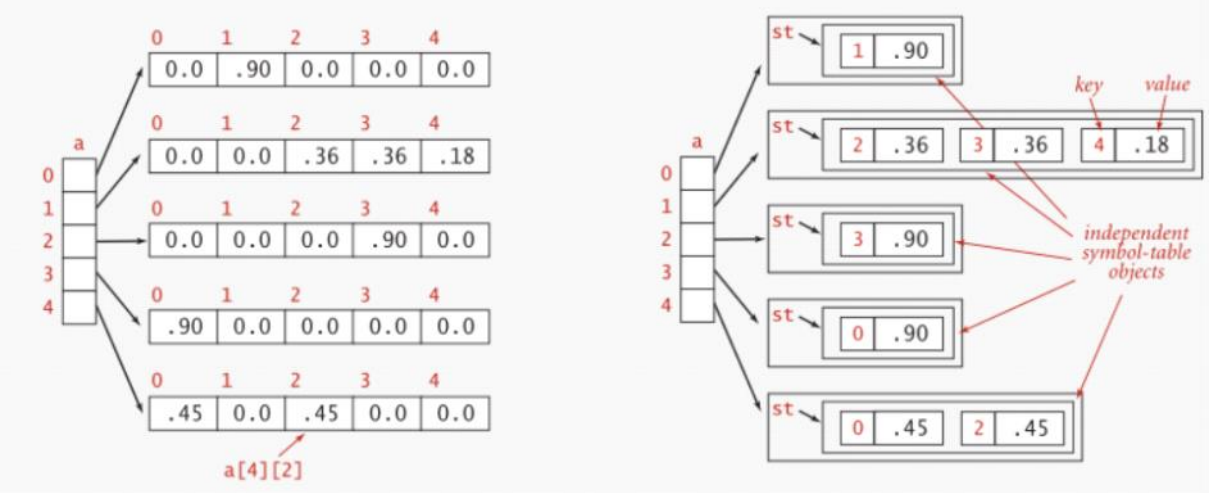
dot product is constant time for sparse vectors

Screen clipping taken: 05/08/2017 4:32 AM

Normal matrix representation - 2D array

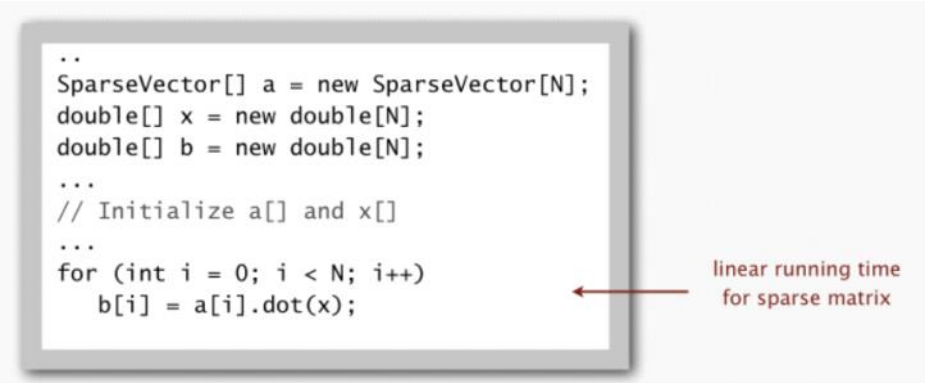
Sparse matrix representation

- each row is a sparse vector



Screen clipping taken: 05/08/2017 4:35 AM

- more efficient running time for matrix multiplication



Screen clipping taken: 05/08/2017 4:35 AM