Tuesday, November 17, 2015
3:24 PM

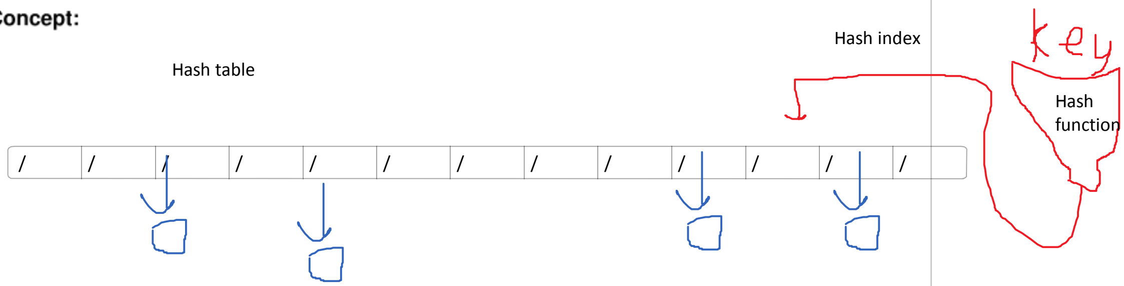
outlineW11
T
Inserted from: <file://C:\Users\SpencerFricke\Downloads\outlineW11T.pdf>

Hashing

Goal:

Do fast then O(Log(n)) number time complexity for lookup, insert, remove on a value oriented collection of size N

Concept:



hash table

Array that stores the collection of items

table size (TS)

Length of the array

load factor (LF)

Number of items / table size

key

Info in item that uniquely identifies it used to compute a hash index for where the item goes in the table

hash function

Converts the key into its hash index

Ideal Hashing

Assume

- store 150 students records
- table is an array of student records
- null is sentinel value meaning element is unused
- key is student id number in format: zipcode + 3 digit number

  53706000, 53706001, 53706002, … 53706148, 53706149

→ What would be a good hash function to use on the id number?

```
int hash(K key) {
    Return key - 53706000
}
```

Perfect Hash Function:

```
void insert(K key, D data) { table[hash(key)] data;}

D lookup(K key) { return Table[hash(key)];}

void delete(K key) {table[hash(key)] = null;}
```

The UW uses 10 digit ID numbers: 9012345789 9012345432 9023456789

→ Is a perfect hash function possible for these id numbers?

Yes, use the trivial hash function hash index is the key, but this requires a HUGE table that wastes a lot of space

Collision:

When the hash function returns the same hash index for different keys

Key Issues:

- Designing the hash function
- Choosing the table size
- Handling collisions

Designing a Hash Function

Good Hash Functions:

1) Must be deterministic
   a. same key always generates the same hash index
2) Achieve uniformity
   a. Keys are distributed evenly across the table
3) Fast/Easy to compute
   a. Use only parts of key that distinguish items
4) Minimize collisions

Java Hash Function Steps:

1. Generate a Hash Code item.hashCode() converts the items key to an integer

2. Compress the hash code into a valid Hash Index for the table 0 to TableSize - 1
   HashCode(key) % TableSize

   *Beware of negative hash codes
   ABS(hashCode) % TableSize
   (HashCode % TableSize) + TableSize if negative

   Since % is slow, using bit shifting for fast * or / of powers of 2
   On a TableSize that's a power of 2 is becoming popular

outlineW11R

Inserted from: <file://C:\Users\SpencerFricke\Downloads\outlineW11R.pdf>

CS 367 Announcements
Thursday, November 19, 2015

Midterm Exam 2

- Tuesday, November 24th, 5:00 pm
- Lec 1: B10 Ingraham Hall
- Lec 2: 6210 Social Sciences Building
- Exam information posted
- Sample questions on Learn@UW
- UW IDs are required

Homework h9 due 10 pm, tomorrow, November 20th

Program p4 due 10 pm, Sunday, November 29th

Last Time

Heap Data Structure
- insert
- removeMax

Hashing
- terminology
- designing a good hash function

Today

Hashing
- designing a good hash function (from last time)
- choosing table size
- expanding a hash table
- handling collisions

Next Time

- exam 2 Q&A

Techniques for Generating Hash Codes

Extraction Breaking the key into parts and then only using those parts that distinguish items.

Weighting Emphasizing some parts over others

Folding

123	456	789

123 * 10 + 456*100 + 789 *1

Folding is combing parts through operations like addition and bitwise operations

Handling Non-Integer Keys

Strings

(hash code)
$C_0 * 31^{n-1} + C_1 * 31^{n-2} .....+ C_{n-2} * 31^1 + C_{n-1} * 31^0$
C_i is ASCII value for char at position "i" in a string of length N

For a variable sized string

$((((C_0 * 31) + C_1) * 31) + C_2) * 31....$

Complexity O(1) with respects too size of the collection
Complexity O(N) with respects too size (length) of the string

Doubles

64 bit IEEE Floating Points

|
V

32 bit unsigned integer
Idea: extract upper and lower 32 bits form together using Java's bitwise exclusive or

Ex: 8 bit double -> 4 bit integer

01101011 -> bit pattern for key
XOR <u>00000110</u> -> key shifted right by 4 bits
0110**1101** **->** extract as 4 bit integer

In Java
Long bits = Double.doubleToLongBits(key);
Int hashcode = (int) (bits ^ (bits>>32));

Choosing the Table Size

Table Size and Collisions

Assume 100 items with random keys in the range 0 – 9999 are being stored in a hashtable.
Also assume the hash function is simply %tablesize.

→ How likely would a collision occur if the table had 10000 elements? 1000? 100?

As the load factor increases, so do collisions

Choose a table size that leaves some extra space
Load factor .7 to .8

Table Size and Distribution

Assume 50 items are stored in a hashtable.
Also assume the hashCode function returns multiples of some value x.
For example, if x = 20 then hashCode returns 20, 40, 60, 80, 100, ...

→ How likely would a collision occur if the table had 60 elements? 50? 37?

This can result in poor distribution

Only m buckets will be used

M = table size / GCF(x, table size)

60/ GCF(20,60) = 60/20 = 3
60/ GCF(20,50) = 50/10 = 5

To avoid this prob use a prime table size

Backup: pick a table size that isn't divisable

Resizing the Hash Table

- Resize when table becomes "too full" LF >= .75
  Naïve approach
  1. Make a new table twice size of old
  2. Copy items from old to new

     Fails since the tableSize affects the hash index and we wanted to spread the items across the new larger table

Steps

Steps for rehashing

1.
   1. "Double" table size to nearest prime backup
      a. (2 * oldTableSize) + 1
      b. Allocate new table of that size

2. 2. Rehash items from old table into the new table

Complexity

Costly O(N) where N is size of the collection, if possible carefully select the initial table size to avoid rehashing

Hash(key) = key % TableSize

TableSize	Hash(63)	Hash Index
19	63 % 9	

Collision Handling using Open Addressing

- Each elements in the table stores one item
- If collision?
    - Search for an unused element
    - Else where in the table
- Idea: use a "probe sequence" with wrap around to search making sure that insert, lookup, delete use same probe sequence

Linear Probing

Step size of 1
Probe Sequence: H_k , $H_k + 1$, ...

Results in clusters of used elements which drops the efficiency or hashing due to long probe sequence

Hash(key) = key % tableSize

better

440	166	266	124	263			337	359		351
0	1	2	3	4	5	6	7	8	9	10

166
359
263

Key	H_k	$H_k + 1$	$H_k + 5$
166	1		
359	7	8	
263	10	11-> 0	4

Quadratic Probing

Probe sequence: H_k, $H_k + 1^2$, $H_k + 2^2$, $H_k + 3^2$, ...

Double Hashing

Probe sequence: H_k, $H_k + 1 * step$, $H_k + 2 * step$, $H_k + 3 * step$,

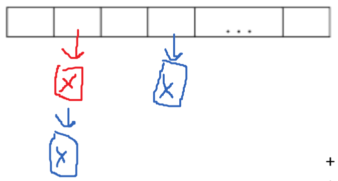
Step = hash2(key)

Preferred for open addressing

Collision Handling using Buckets

- Each element can store more than one item, it's a bucket!
- If collision? Just throw that collided item in bucket
- Typically leave the buckets unsorted

"Chained" Buckets



+ easy implemented
+ buckets are dynamically sized (grow/shrink)

Chains of listnodes, insert is O(1)
Lookup/delete O(1) on average - reasonable table size and good hash function
O(N) for worse case

Array Buckets

Typically a bucket size of 3 works well - reasonable table size and good hash function

Tree Buckets

Balanced search tree is overly complicated given the buckets stores only a few items

Java API Support for Hashing

`hashCode` method

- method of `Object` class
- returns an `int`
- default hash code is BAD - computed from object's memory address

Guidelines for overriding `hashCode`:

*Remember that it must be deterministic (same value sends same hash code)

- If your item class implements/overrides .equals() then it should also implement/overrides .hashCode()

`Hashtable<K,V>` and `HashMap<K,V>` class

- in `java.util` package
- implement `Map<K,V>` interface
- K Type parameter for the key
  `V` Type parameter for the associated value
  operations:
  V get (K key) {} //*lookup*
  V put (K key, V value) {} //*insert*
  Boolean remove (K key) {}
  V remove (K key, V value) {}
- constructors allow you to set
  initial capacity (default = 16 for HashMap, 11 for HashTable)
  load factor (default = 0.75)
- handles collisions with chained buckets
- HashMap only: Allows null for both keys and values
- Hashtable only: Is synchronized

TreeMap vs HashMap

	TreeMap	HashMap
Underlying Date Structure	RBT	Hashtable with chained buckets
Complexity of basic operations	O(LogN)	O(1) average case O(N) worst case
Iterating over the keys	Ascending order	No particular order
Complexity of iterating Over values	O(N) Tree Traversal	O(Table size + N) worse case

If hashing is so fast why don't we always use it for storing a key-value ordered collection

outlineW13T

Inserted from: <file://C:\Users\SpencerFricke\Downloads\outlineW13T.pdf>

CS 367 Announcements
Tuesday, December 1, 2015

Program p5 due 10 pm, Tuesday, December 15th

Last Time
- exam mechanics
- sample questions

Today

Finish Hashing (prior lecture)
ADTs/Data Structures Revisited
Graphs
- terminology
- implementation issues
- edge representations
- traversals

Return Exam 2

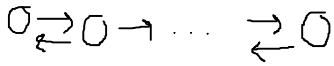
Next Time

Read: continue *Graphs*
- traversals
- applications of BFS/DFS
- more terminology
- topological ordering

ADTs/Data Structures

Linear

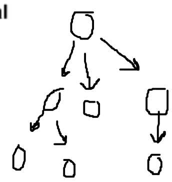
first last



- predecessors: at most 1 Prev
- successors: at most 1 Next

Hierarchical

Top Root

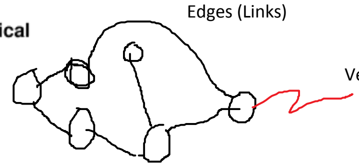


"Bottom" Leaves

- predecessors: at most 1 Parent
- successors: 0 or more - general tree, at most two - binary tree Children

Graphical

Edges (Links)



Vertex (Node)

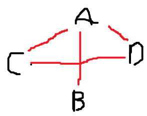
- predecessors:
- successors: Any Number

*Represent pairwise relationships/processes between items in the collection.
Edge indicates some relationship exists

*No clear first/last or top/bottom so we need to specify where to start/stop
Ueter Efficiently

Graph Terminology

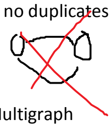
undirect



"grapa"

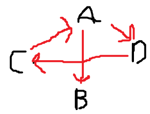
Self-edge

no duplicates

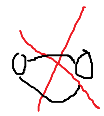


Multigraph

directed



"Digraph"



Degree: number of edges for a specified vertex

Path: sequence of connected vertexes
    Example: c, a, b - acyclic
        c, a, d, c - cyclic

In-Degree: numbers of incoming edges
Out-Degree: numbers of outgoing edges
Degree: in-degree + out-degree

Path: sequence of vertexes in the direction of their connections
    Example: A, D, B - acyclic
        D, B, D - cyclic

Source   target

$A \longrightarrow B$

A pred B
B succ A

Order: number of vertexes
Size: Number of edges

Implementing Graphs

Graph ADT Ops

Constructor (Initially empty)
Insert vertex, insert edge(between existing vertexes)
Deleted vertex (its edges), delete edge
Lookup vertex (Might return associated data)
Lookup edge
isEmpty, degree, size, order

+ others

Graph Class

```
Public class Graph<T> {
        Private list<Graphnode<T>> nodes;
        //could use map instead of list (TreeMap, HashMap) duplicates are ok
        //set (TreeSet, HashSet) no duplicates
```

Graphnode Class

```
Class Graphnode<T> {
        Private T data;
```

Representing Edges

Adjacency Matrix

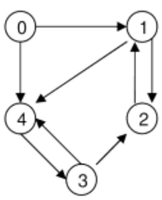
Store edges in a 2D array

I → j == (I, j) == AM[I][j]

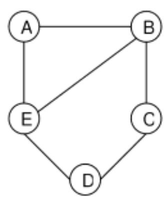
Need to map node's key to its AM inde
Add to graph class
Private boolean[][] edges;

Given the following graphs:

Graph 1



Graph 2



→ Show the adjacency matrix representation of the edges for each of the graphs:

Graph 1

Target To

Source From

	0	1	2	3	4
0		T			T
1			T		T
2		T			
3			T		T
4				T	

Graph 2

	A	B	C	D	E
A		T			T
B	T		T		T
C		T		T	
D			T		T
E	T	T			

Undirected mirror image

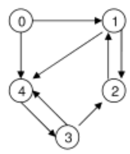
Representing Edges

Adjacency Lists

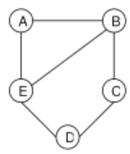
-Stores a list of successor in each graphnode
-add to graphnode class
    Private list<Graphnode<T>> edges

Given the following graphs:

Graph 1



Graph 2



→ **Show an adjacency list representation of the edges for each of the graphs:**

	Graph 1		Graph 2
0:	1,4	A:	
1:	2,4	B:	
2:	1	C:	
3:	2,4	D:	
4:	3	E:	

Using Edge Representations

→ **Write the code to be added to a `Graph` class that computes the degree of a given node in an undirected graph.**

1. Adjacency list:

```
public int degree( Graphnode<T> n) {

    Return n.getEdges().size();
```

2. Adjacency matrix:

```
public int degree( Graphnode<T> n) {

        Int I = map n's key to its AM index
        Int degree = 0;
        For (int s = 0; s < #nodes;  s++)
            If (edges[i][s])
                Degree++;

        Return degree;
```


Comparison of Edge Representations

Ease of Implementation

Both easy
AM requires mapping key -> index

Space (memory)

AM $O(N^2)$ always

AL $O(N)$ Average case - sparse graphs
$O(N^2)$ Worse case = complete graphs

Time (complexity of ops)

Depends on the applications operation

node's degree?

AM $O(N)$

AL $O(1)$

edge exit between two given nodes?

AM check AM[A][B] = $O(1)$

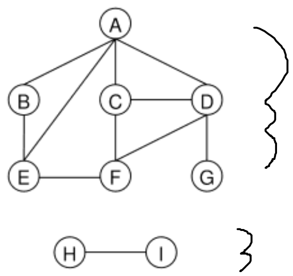
AL Search AL = $O(N)$ Worse case

Searches and Traversals

Search Look through a collection stopping at the first item that matches the search criteria

Traversal Visit each item in the collection exactly once

Graph: specify start vertex and visit those vertexes that are reachable



Connected component

→ What is the length of the longest path starting at A?

Visit each node exactly once
Problem: want avoid cycles
Solution: Mark vertexes as they are visited

*IN 367 pick unvisited successors in increasing numerical order or alphabetical order

outlineW13R

Inserted from: <file://C:\Users\SpencerFricke\Downloads\outlineW13R.pdf>

CS 367 Announcements
Thursday, December 3, 2015

Homework h10 assigned 12/6

Program p5 due 10 pm, Tuesday, December 15th

Last Time

Finish Hashing
ADTs/Data Structures Revisited
Graphs
- terminology
- implementation issues
- edge representations

Return Exam 2

Today

Graphs
- traversals
- applications of BFS/DFS
- more terminology

Next Time

Read: finish *Graphs*, start *Sorting*
- topological ordering
- Dijkstra's algorithm

Sorting Intro
Basic Sorts
- bubble sort
- insertion sort
- selection sort

Depth-First Search (DFS)

- Assume all vertexes initially marked unvisited
- Relies on a stack, we'll use the call stac with recursion

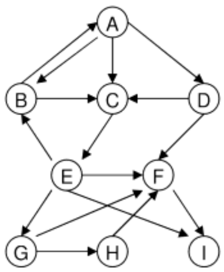
Algorithm

```
DFS(v)
    Mark v as visited
    For each unvisited successor s that is adjacent to v
        DFS(s)
```

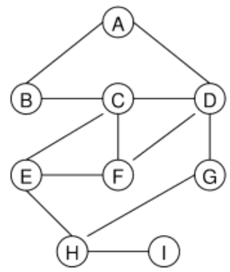
*Equivalent to preorder traversal

DFS Practice

Graph 1



Graph 2



→ Give the order that vertexes are visited for depth-first search (DFS) starting at A

Graph 1: A, B, C, E, F, I, G, H, D

Graph 2: A, B, C, D, F, E, H, G, I

→ Give the DFS spanning tree starting at A

Graph 1:

A - D
|
B
|
C
|
E - G - H
|
F
|
i

Graph 2:

A -
|
B
|
C
|
D
|
F
|
E
|
H - G - i

Breadth-First Search (BFS)

- Assume all vertexes are initially marked unvisited
- Relies on a queue

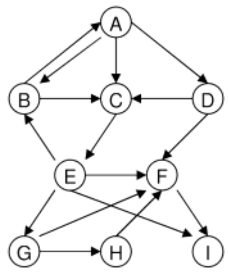
Algorithm

```
BFS(v)
    Q = new Queue()
    Mark v as visited
    Q.enqueue(v)
    While (!Q.isEmpty())
        C = Q.dequeue()
        For each unvisited successors adjacent to c
        Mark s as visited
        q.enqueue(s)
```

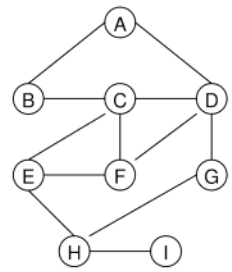
*Equivalent to a level-order traversal

BFS Practice

Graph 1



Graph 2



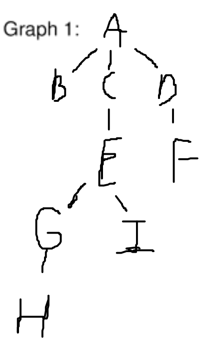
→ Give the order that vertexes are visited for breadth-first search (BFS) starting at A.

Graph 1: A, B, C, D, E, F, G, I, H

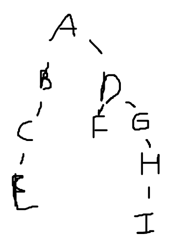
Graph 2: A, B, D, C, F, G, E, H, I

Give the BFS spanning tree starting at A.

Graph 1:



Graph 2:



Applications of DFS/BFS

Is the graph connected starting at some start vertexes?
What vertexes are reachable from some start vertexes?

Path Detection

Is there a path from Start to X?
What is a path from start to x?
Idea: Do DFS/BFS modified to keep a predecessor list that is used to reconstruct the path. (Later)
What is the shortest path from start to x?
Idea: (unweighted) use BFS to find a path with fewest edges
Idea: (weighted) use Dijkstra's algorithm to find the path with the smallest total edge weights (later)

Cycle Detection

Is there a cycle from start back to start (excluding simple cycles such as A,B,A), we want a cycle of 3 or more vertexes.
Is there a cycle anywhere in the graph (see readings for intro about using 3 marks included "in progress")

More Graph Terminology

Weighted Graph: Edge's are assigned a value representing a cost, time, distance, force

Network: weighted digraph - where edge weights are non-negative

Complete Graph: An edge exists between every pair of vertexes
N = # of vertexes
#edges = N(N-1)/2

Connected Graph:
- Undirected: a **path** exists between every pair of vertexes
- Directed:
  - Weakly: a path exists between every pair of vertexes ignoring edge directions
  - Strongly: a path exists between every pair of vertexes respecting edge direction

Tree: Direct acyclic graph

outlineW14T

Inserted from: <file://C:\Users\SpencerFricke\Downloads\outlineW14T.pdf>

CS 367 Announcements
Tuesday, December 8, 2015

Homework h10 due 10 pm, Friday, December 11th

Program p5 due 10 pm, Tuesday, December 15th

Last Time

Graphs
- more terminology
- traversals
- applications of BFS/DFS

Today

Graphs
- applications of BFS/DFS (from last time)
- topological ordering
- Dijkstra's algorithm

Sorting Intro

Basic Sorts
- bubble sort
- insertion sort
- selection sort

Next Time

Read: continue *Sorting*

Finish Basic Sorts

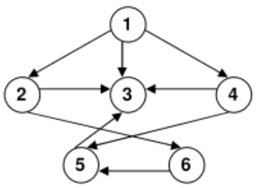
Better Sorts
- heap sort
- merge sort
- quick sort

Or Topological Sorting, or Topological Numbering

Topological Ordering

Idea: come up with a list of vertexes such that each vertex in the list comes before any of its successors

→ **Arrange the vertices below in a list such that when the edges are added none of the arrows point to the left:**



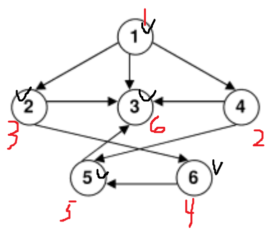
1, 2, 4, 6, 5, 3

Topological Ordering Algorithm

Iterative Algorithm We will use a stack, see readings for recursive implementation

```
Num = number of vertexes
ST = new stack();
Mark all vertexes as unvisited
For each vertex V with no predecessors
        Mark V as visited
        ST.push(V);
While (!ST.isEmpty())
        V = ST.peek()
        If all successors of V are marked visited
                ST.pop()
                Give it the value of Num
                Num--
        Else select one unvisited successor U adjacent to V
                Mark U as visitied
                ST.push(U)
```

Example



Num = 6

ST

1, 4, 2, 6, 5, 3

Dijkstra's Algorithm

- Finds shortest path (smallest total edge weights) in a network
- Edge weights must be non-negative
- Edge weights can be unbounded
- Must specify a single start vertex
- Can be used on undirected graphs by converting edges $A - B \Rightarrow A \rightleftarrows B$

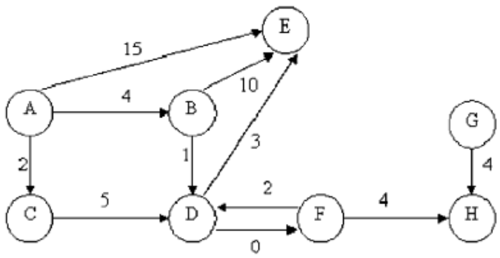
```
for each vertex V
   initialize V's visited mark to false
   initialize V's total weight (tw) to infinity
   initialize V's predecessor to null
set start_vertex's total weight (tw) to 0

create new priority queue pq
put (0, start_vertex) on pq

while !pq.isEmpty()
   (V's tw, V) = pq.removeMin()
   set V's visited mark to true

   for each unvisited successor S adjacent to V
      if S's total weight can be reduced
         update S's total weight to: V's tw + edge weight from V to S
         change S's predecessor to: V
         put (S's tw, S) on pq (or just update S's tw if already on pq)
```

Dijkstra's Practice



Iteration	Priority Queue min max
0	0,A
1	2,C 4,B 15,E
2	4,B 7,D 15,E
3	5,D 14,E
4	5,F 8,E
5	8,E 9,H
6	9,H
7	

Underlined means changed during that iteration

Vertex	Visited	Total Weight (tw)	Predecessor
A	~~F~~ T	~~∞~~ 0	\
B	~~F~~ T	~~∞~~ 0+4 = 4	~~\~~ A
C	~~F~~ T	∞ 2	\ A
D	~~F~~ T	∞ ~~7~~ 5	\ ~~C~~ B
E	~~F~~ T	∞ ~~15~~ ~~14~~ 8	\ ~~A~~ ~~B~~ D
F	~~F~~ T	∞ 5	\ D
H	~~F~~ T	∞ 9	\ F
G	F	∞	\

Reconstruct shortest path from A to F

Trick is to start from destination
F, D, B, A => A, B, D, F => cost - 5

Sorting

Problem
Arrange a collection of items into some prescribed order well do increasing numerical order

Solution
Comparison sort
Compare pairs of items to determine
Determine their relative order - in Java use comparable items with compareTo() we will us < > =

Complexity
Best comparison sorts are O(N Log(N)) for the worst case time complexity where N is number of items in the collection

2 Dominant operations - comparison and Data move (shifts/swaps)

In-Place Sorts

Just use one array to store and sort the collection

Basic In-Place Comparison Sorts

Idea: array is divided into sorted and unsorted parts
-Each pass through the array moves one item from unsorted to sorted
-Sorting the entire array requires at most N-1 passes

Bubble Sort

Idea Each pass through the unsorted part "Bubbles" the next smallest item to the back of the sorted part

*Started entire array at unsorted .

Psuedocode

```
int passes = A.length-1;
Boolean swapsDown = true;
for (int i = 0; i < passes && swapsDown ; i++) {      <- passes

   for (int j = A.length-1; j > i; j--) {   } Bubbling
      if (A[j] < A[j-1]) {
         swap(A[j], A[j-1]);

      }
   }
}
```

Analysis (modified code)

	best case	worst case
kind of array	Sorted	Reverse Sorted
# comparisons	$\cancel{O(N^2)}$ O(N)	$O(N^2)$
# swaps	0	$O(N^2)$
total	O(N)	$O(2N^2)$ -> $O(N^2)$

Insertion Sort

Idea Start with first item in sorted part

Each pass <u>insert</u> next unsorted item into the sorted part

Psuedocode (linear insertion)

```
for (int i = 1; i < A.length; i++) {      <- passes
   int temp = A[i];

   int j;
   for (j = i-1; j >= 0 && A[j] > temp; j--)      <- Inserting
      A[j+1] = A[j];

   A[j+1] = temp;
}
```

Analysis

	best case	worst case
kind of array	Sorted	Reverse Sorted
# comparisons	O(N)	$O(N^2)$
# shifts	0	$O(N^2)$
total	O(N)	$O(2N^2)$ -> $O(N^2)$

Selection Sort

Idea

-Start entire array as unsorted
-Each pass select smallest from unsorted part and swap it wit the first unsorted

Psuedocode

```
int passes = A.length-1;
for (int i = 0; i < passes; i++) {
   int minIndex = i;

   for (int j = i+1; j < A.length; j++) {
      if (A[j] < A[minIndex])
         minIndex = j;
   }

   swap(A[minIndex],A[i]);
}
```

Analysis

	best case	worst case
kind of array	Sorted	Not reverse sorted N, 1, 2, 3, … N-1
# comparisons	$O(N^2)$	$O(N^2)$
# swaps	0 (excluding self swap)	O(N)
total	$O(N^2)$	$O(N^2)$

outlineW14R

Inserted from: <file://C:\Users\SpencerFricke\Downloads\outlineW14R.pdf>

CS 367 Announcements
Thursday, December 10, 2015

Final Exam

- Wednesday, December 23rd, 7:45 to 9:45 am (morning)
- Exam information posted
- Sample questions on Learn@UW
- UW IDs are required

Homework h10 due by 10 pm tomorrow, Friday, December 11th

- make sure your file is a pdf but not pdf scan of written work or pdf of a screen shot
- make sure you use the name h10.pdf
- submit to your `in` handin directory
- remember homeworks are to be done individually
- remember that late work is not accepted

Program p5 due 10 pm, Tuesday, December 15th

Last Time

Graphs

- applications of BFS/DFS
- topological ordering
- Dijkstra's algorithm

Today

Sorting Intro

Basic Sorts

- bubble sort
- insertion sort
- selection sort

Better Sorts

- heap sort
- merge sort
- quick sort

Next Time

Read: finish *Sorting*

Finish Better Sorts

Stable Sorts

Sorting in Java

Heap Sort

Idea Naïve version
- Insert each item from original array into a min heap
- Remove min each item from the min heap filling the original array from left to right

Analysis (Naïve)
- N * O(log(N)) = O(N log(N))
- N * O(log(N)) = O(N log(N))

= O(2N log(N)) -> O(N log(N))

Space for Naïve
Requires 2N Memory

Doing 'In Place' would be better
- use a max heap reheapifying then do remove max filling the array from left to right

Merge Sort

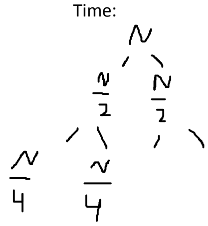
Idea Divide and conquer recursive algorithm

1. Divide the array in half then recursively merge sort each half
2. Merge the 2 sorted halves into 1 sorted list

Analysis

Merge work at each level * numbers of levels

Time:



1 * N = O(N)
2 * N/2 = O(N)
…. = O(N)

Tree height is O(Log(N))

O(N) * O(log(N)) = O(N log(N))

Space requires 2N memory

In-place can get to N memory

Quick Sort

Divide and conquer recursive algorithm

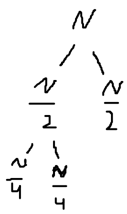
Idea

1. Select a value from the array (called the pivot) and partition the array into

<= P	P	>= P
Left Part		Right Part

2. Recursively quick sort on the left and right parts

Analysis Time for Best case



Partition work at each level * numbers of levels

2 * N/2 = O(N)
4 * N/4 = O(N)

Tree height = O(log(N))

O(N) * O(log(N)) = O(N log(N))
Best and Average Cases

Worse case

O(N) * O(N) = O(N^2)

Quick Sort (cont.)

Choosing a Good Pivot

Bad - Pivot is A[First]
  If array is sorted/revered sorted then the pivot will be smallest/largest resulting in everything going into 1 part
Good - "Median of 3"
  Pick the middle value of A[first], A[Middle], A[Last]

Partitioning the Array

6 1 5 9 3 5 4 3 7 6 2 8 2 Pivot Selection
2 L   8   R 4 6
 L   R
 2   5
 L   R R R
Stop when R crosses L
2 1 2 3 3 5 8 9 7 6 5 4 6
  4   5
2 1 2 3 3 <u>4</u> 8 9 7 6 5 5 6
Quick sort left and right of <u>4</u>

Partitioning unknown values

Left increases to value > P or it crosses over R
Right decreases to value < P or it crosses over L

outlineW15T

Inserted from: <file://C:\Users\SpencerFricke\Downloads\outlineW15T.pdf>

CS 367 Announcements
Tuesday, December 14, 2015

Final Exam

- Wednesday, December 23rd, 7:45 to 9:45 am (morning)
- Lec 1: B10 Ingraham Hall
- Lec 2: 145 Birge Hall
- UW ID REQUIRED
- Bring #2 pencils
- Exam information posted
- Sample questions on Learn@UW

Program p5 due 10 pm, Tonight, December 15th

Verify your scores are correctly entered on Learn@UW

Last Time

Sorting Intro
Basic Sorts

- bubble sort
- insertion sort
- selection sort

Better Sorts

- heap sort
- merge sort
- quick sort

Today

Finish Better Sorts (from last lecture)
Stable Sorts
Sorting in Java
Course Overview Sheets
Final Exam Info
Evaluations – Skrentny, CS 367, lecture

Stable Sorts

→ What do you notice about the sorting of the following three lists of names?

Unsorted	sorted by first name	sorted by last name
Jane Jetson	Barney Rubble	Stewie Griffin
Elroy Jetson	Elroy Jetson	Elroy Jetson
Homer Simpson	George Jetson	George Jetson
Marge Simpson	Homer Simpson	Jane Jetson
Stewie Griffin	Jane Jetson	Judy Jetson
Judy Jetson	Judy Jetson	Barney Rubble
George Jetson	Marge Simpson	Homer Simpson
Barney Rubble	Stewie Griffin	Marge Simpson

*used on composite items (have multiple keys)
*Stable sort preserve the relative ordering for duplicate keys

Stable	Unstable
Bubble	Selection
Insertion	Heap
Merge	

Sorting in Java API

In `java.util`

`Collections.sort(List)`

Uses a modified merge sort since it is stable an the items in the collection are likely to be composites

`Arrays.sort(`*`array_to_sort`*`)`

Overloaded

If array of references

If array or primitives tuned quicksort

Median of 3	Median of 3	Median of 3

Median of 3

If Subarray is small insertion sort is used

Abstract Data Types (ADTs) and Data Structures (DS)

ADT
DS

Layout of Collection

- Linear

```
List                array, SimpleArrayList, shadow array
                    chain of nodes, Listnode, SimpleLinkedList
                         tail, header, doubly linked, circularly
linked
Stack
Queue
Deque               circular array
```

- hierarchical

```
                    general tree, Treenode
                    binary tree, BinaryTreenode
                    binary search tree, BSTnode
                    balanced search tree
                    red-black tree
PriorityQueue       heap
```

- graphical

```
Graph               Graphnode
                    adjacency matrix
                    adjacency list
```

Orientation of Operations

- position oriented - operations occur at a specified position

  list, stack (top), queue (front/rear), deque ("double ended")

- value oriented - operations occur at position determined by item's key value

```
                    sorted list
                    search trees
Map                 hash table
```

- hybrid?

```
PriorityQueue       heap
                    hash table
```

Algorithms

Operations on ADTs/data structures

insert, lookup, remove

Recursion

vs. iteration
rules, guiding questions
call stack trace
execution tree trace

Traversing

list			
tree	level	pre/in/post	
graph	DFS (stack)	BFS (queue)	spanning trees

Searching

linear O(N)
binary O(logN)

Hashing

hash function: hash code (extracting, weighting, folding) → hash index (compressing)
table size: prime size, load factor, rehashing
collisions: open addressing, buckets

Graphs

topological ordering
Dijkstra's (priority queue)

Sorting

basic O(N^2): bubble, insertion, selection

better O(NlogN): heap, merge, quick

stable sorts

Complexity

Complexity

1, logN, N, NlogN, N^2, N^3, 2^N, N!

time: abstract, dominant ops

space: memory

worst/average/best-case

big-O

Determining Complexity

informal
- constant
- linear
- quadratic

code
- loops
- method calls

time equation
- simplify

recurrence equations
- base T( ) =
- recursive T(N) = + T ( )
- equations → table, guess solution → verify → complexity

Caveats

small problem size
same complexity

Java Concepts

Primitives vs. References

Command-line Arguments

Exceptions

`throw`
`try`/`catch`/`finally`
`throws` (checked vs unchecked)
defining

Programming for Generality

`Object`
generics

Interfaces

`Comparable`, `compareTo`
ADTs

Iterators

`Iterable: iterator()`
`Iterator: hasNext(), next()`

indirect
direct

Package Visibility

Java Collections Framework

`Iterable<T>`, `Iterator<E>`
`List<T>: ArrayList<T>`, `LinkedList<T>`
`Vector<E>`, `Stack<E>`
`Hashtable<K,V>`
`Map<K,V>: TreeMap<K,V>`, `HashMap<K,V>`
`Set<E>: TreeSet<E>`, `HashSet<E>`