CSE 326 Data Structures

Dave Bacon

Final Review

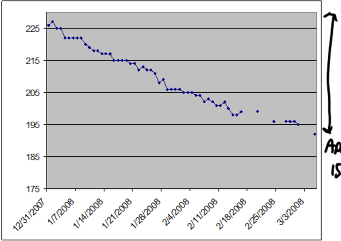
Stay on Target….Stay on Target

Logisitics

- Hand in Homework 7
- Friday: Games and NP completeness

- **Final for Section A:**

Thursday March 15, 8:30-10:20 MGH 231


225
215
205
195
185
175
12/31/2007
1/7/2008
1/14/2008
1/21/2008
1/28/2008
2/4/2008
2/11/2008
2/18/2008
2/25/2008
3/3/2008

April 15th

Final Logisitics

- Example Final ~~Example~~ (up soon)
- Final Exam Review Material (up soon)
- Homework 7 will not be returned before final, but homework solution will be posted shortly

- Regular office hours next week, plus, I'll be in my office (CSE 460) 9-5. Stop by or email for a good time to meet.

Final Material

- “Everything is fair game”
- BUT ~~80~~-90% of the material will come from material covered after the midterm

splay tree's

- This means: Splay trees onward
- This means: Up to Krustkal’s

Floyd-Warshall

Huffman Coding.

Final Material Rough Map

- Stuff before the midterm
- Splay Trees, B-Trees, Memory Hierarchy
- Hashing
- Disjoint Sets
- Sorting
- Graph Algorithms

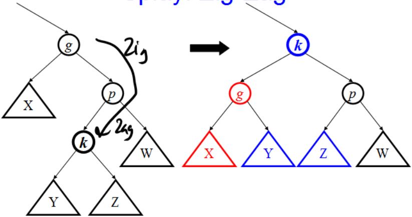
Splay Trees

self adjusting.

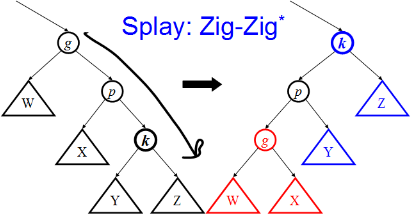
- Blind adjusting version of AVL trees
  - Why worry about balances? Just rotate anyway!
- *Amortized* time per operations is O(log *n*)
- Worst case time per operation is O(*n*)
  - But guaranteed to happen rarely

Insert/Find always rotate node *to the root*!

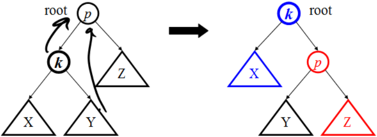
Splay: Zig-Zag*



Splay: Zig-Zig*



Special Case for Root: Zig



Splay Operations: Find

- Find the node in normal BST manner
- Splay the node to the root
  - if node not found, splay what would have been its parent

o splay

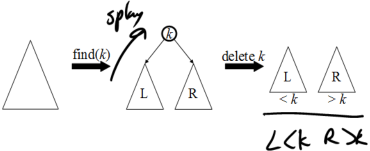
nothing

when in doubt splay

Splay Operations: Insert

- Insert the node in normal BST manner
- Splay the node to the root

Splay Operations: Remove



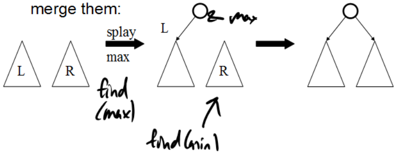
Now what?

~~Array~~ Join

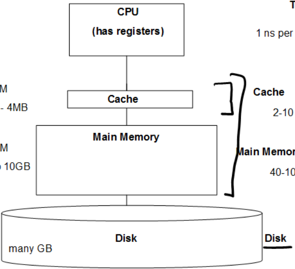
Join

Join(L, R):

given two trees such that (stuff in L) < (stuff in R), merge them:



Splay on the maximum element in L, then attach R



CPU

(has registers)

Cache

Main Memory

Disk

SRAM

8KB - 4MB

DRAM

up to 10GB

many GB

Time to access:

1 ns per instruction

Cache

2-10 ns

Main Memory

40-100 ns

Disk

a few *milli*seconds

(5-10 *Million* ns)

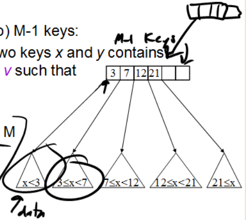
Solution: B*-Trees

- specialized *M*-ary search trees

~~binary~~

- Each **node** has (up to) M-1 keys:
  - subtree between two keys *x* and *y* contains leaves with *values v* such that $x \le v < y$

- Pick branching factor M such that each node takes one full {*page, block*} of memory



B$^+$-Tree Properties ‡

- Data is stored at the leaves ✓
- All leaves are at the same depth and contains between $\lceil L/2 \rceil$ and L data items
- Internal nodes store up to M-1 keys mostly full
- Internal nodes have between $\lceil M/2 \rceil$ and M children
- Root (special case) has ✓ between 2 and M children (or root could be a leaf)

‡These are technically B$^+$-Trees

Insertion Algorithm

1. Insert the key in its leaf
2. If the leaf ends up with L+1 items, **overflow**!
   - Split the leaf into two nodes:
     - original with $\lceil (L+1)/2 \rceil$ items
     - new one with $\lfloor (L+1)/2 \rfloor$ items
   - Add the new child to the parent
   - If the parent ends up with $M+1$ items, **overflow**!
3. If an internal node ends up with M+1 items, **overflow**!
   - Split the node into two nodes:
     - original with $\lceil (M+1)/2 \rceil$ items
     - new one with $\lfloor (M+1)/2 \rfloor$ items
   - Add the new child to the parent
   - If the parent ends up with $M+1$ items, **overflow**!
4. Split an overflowed root in two and hang the new nodes under a new root

This makes the tree deeper!

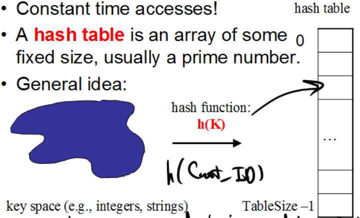
Deletion Algorithm

1. Remove the key from its leaf

2. If the leaf ends up with fewer than $\lceil L/2 \rceil$ items, **underflow**!
   - Adopt data from a sibling; update the parent
   - If adopting won't work, delete node and merge with neighbor
   - If the parent ends up with fewer than $\lceil M/2 \rceil$ items, **underflow**!

Hash Tables

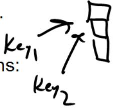
- Constant time accesses!
- A **hash table** is an array of some fixed size, usually a prime number.
- General idea:



designing good hash functions

Collision Resolution

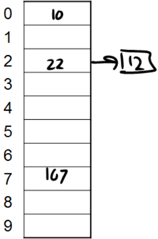
Collision: when two keys map to the same location in the hash table.



Two ways to resolve collisions:

1. Separate Chaining
2. Open Addressing (linear probing, quadratic probing, double hashing)

“Separate Chaining”

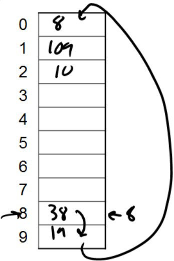


Insert:
10
22
107
12
42

- **Separate chaining**: All keys that map to the same hash value are kept in a list (or “bucket”).

Open Addressing

Find (8)



Insert:
38
19
8
109
10

- **Linear Probing**: after checking spot h(k), try spot h(k)+1, if that is full, try h(k)+2, then h(k)+3, etc.

Terminology Alert!

“**Open** Hashing”
equals
“Separate Chaining”

Weiss

4

“Closed Hashing”
equals
“**Open** Addressing”

4

Load Factor in Linear Probing

- For *any* $\lambda < 1$, linear probing *will* find an empty slot

load factor $\lambda = \frac{\# \text{keys}}{\text{size table}}$

- Expected # of probes (for large table sizes)
  - successful search: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$

~~2.5~~ $\lambda = \frac{1}{2}$ 1.5 probes

  - unsuccessful search: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$

$\lambda = \frac{1}{2}$ 2.5 probes

- Linear probing suffers from ***primary clustering***
- Performance quickly degrades for $\lambda > 1/2$

Quadratic Probing

Less likely
to encounter
Primary
Clustering

$f(i) = i^2$

- Probe sequence:

  0th probe = h(k) mod TableSize

  1th probe = (h(k) + 1) mod TableSize

  2th probe = (h(k) + 4) mod TableSize

  3th probe = (h(k) + 9) mod TableSize

  . . .

  ith probe = $(h(k) + i^2)$ mod TableSize

Quadratic Probing: Success guarantee for $\lambda < \frac{1}{2}$

less than half full.

- If size is prime and $\lambda < \frac{1}{2}$, then quadratic probing will find an empty slot in size/2 probes or fewer.
  - show for all $0 \leq i,j \leq size/2$ and $i \neq j$

    $(h(x) + i^2) \bmod size \neq (h(x) + j^2) \bmod size$
  - by contradiction: suppose that for some $i \neq j$:

    $(h(x) + i^2) \bmod size = (h(x) + j^2) \bmod size$

    $\Rightarrow i^2 \bmod size = j^2 \bmod size$

    $\Rightarrow (i^2 - j^2) \bmod size = 0$

    $\Rightarrow [(i + j)(i - j)] \bmod size = 0$

BUT size does not divide $(i-j)$ or $(i+j)$

Double Hashing

$f(i) = i * g(k)$

where g is a second hash function

- Probe sequence:

  0th probe = h(k) mod TableSize

  1th probe = (h(k) + g(k)) mod TableSize

  2th probe = (h(k) + 2*g(k)) mod TableSize

  3th probe = (h(k) + 3*g(k)) mod TableSize

  . . .

  ith probe = (h(k) + i*g(k)) mod TableSize

Rehashing

Disjoint Sets

Chapter 8

Disjoint Union - Find

{1},{2},...

- Maintain a set of pairwise disjoint sets.
  - {3,5,7} , {4,2,8}, {9}, {1,6}
- Each set has a unique name, one of its members
  - {3,5,7} , {4,2,8}, {9}, {1,6}

Find(4) → 8

Union

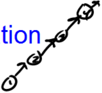
- Union(x,y) – take the union of two sets named x and y
  - {3,5,7} , {4,2,8}, {9}, {1,6}
  - Union(5,1)

    {3,5,7,1,6}, {4,2,8}, {9},

Find

- Find(x) – return the name of the set containing x.
  - {3,5,7,1,6}, {4,2,8}, {9},
  - Find(1) = 5 ←
  - Find(4) = 8

Simple Implementation

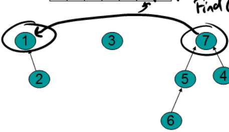


- Array of indices

	1	2	3	4	5	6	7
up	0	1	0	7	7	5	~~0~~ 1

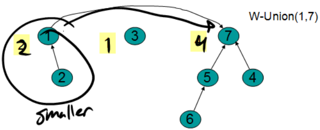
Up[x] = 0 means x is a root.

Find(6) = 7



Weighted Union

- Weighted Union
  - Always point the *smaller* (total # of nodes) tree to the root of the larger tree

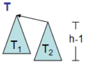


Analysis of Weighted Union

With weighted union an up-tree of height h has weight *at least* 2^h.

- Proof by induction
  - **Basis**: h = 0. The up-tree has one node, $2^0 = 1$
  - **Inductive step**: Assume true for all h' < h.

Minimum weight up-tree of height h formed by weighted unions



$W(T_1) \geq W(T_2) \geq 2^{h-1}$

Weighted union

Induction hypothesis

$W(\mathbf{T}) \geq 2^{h-1} + 2^{h-1} = 2^h$

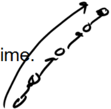
Analysis of Weighted Union (cont)

Let T be an up-tree of weight n formed by weighted union. Let h be its height.

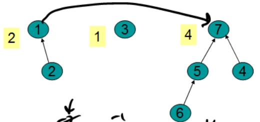
$$n \geq 2^h$$

$$\log_2 n \geq h$$

- Find(x) in tree T takes O(log n) time.



Array Implementation



~~-2~~ 7 -1 -4-2=-6

	1	2	3	4	5	6	7
up	-1	1	-1	7	7	5	-1
weight	2		1				4

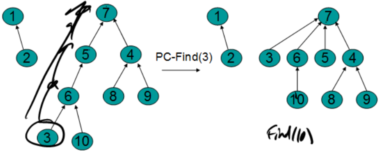
Nifty Storage Trick

- Use the same array representation as before
- Instead of storing `-1` for the root, simply store `-size`

[Read section 8.4, page 276]

Path Compression

- On a Find operation point all the nodes on the search path directly to the root.



Complex Complexity of Union-by-Size + Path Compression

Tarjan proved that, with these optimizations, p union and find operations on a set of n elements have worst case complexity of $O(p \cdot \underline{\alpha(p, n)})$

For *all practical purposes* this is amortized constant time:

$O(p \cdot 4)$ for p operations!

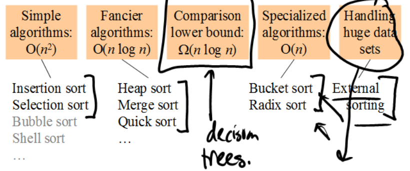
$O(4p)$ amortize $\quad \frac{O(4p)}{p} = O(1)$

- Very complex analysis – worse than splay tree analysis etc. that we skipped! disjoint sets

Sorting: *The* ***Big*** *Picture*

Given *n* <u>comparable</u> elements in an array, sort them in an increasing (or decreasing) order.

bound violate bound



Simple algorithms: $O(n^2)$

- Insertion sort
- Selection sort
- Bubble sort
- Shell sort
- …

Fancier algorithms: $O(n \log n)$

- Heap sort
- Merge sort
- Quick sort
- …

Comparison lower bound: $\Omega(n \log n)$

decision trees.

Specialized algorithms: $O(n)$

- Bucket sort
- Radix sort

Handling huge data sets

- External sorting

Insertion Sort: Idea

- At the *k*th step, put the *k*th input element in the correct place among the first *k* elements
- Result: After the *k*th step, the first *k* elements are sorted.

Runtime:

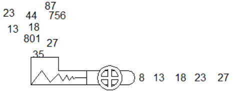
worst case	:	$O(n^2)$
best case	:	$O(n)$
average case	:	$O(n^2)$

Selection Sort: idea

- Find the smallest element, put it 1st
- Find the next smallest element, put it 2nd
- Find the next smallest, put it 3rd
- And so on …

$O(n^2)$

HeapSort: Using Priority Queue ADT (heap)



Build = N

Shove all elements into a priority queue, take them out smallest to largest.

Delete min = log N

Runtime: O(N log N

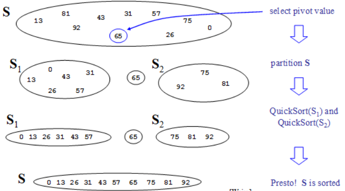
Merge Sort

```
MergeSort (Array [1..n])
1. Split Array in half
2. Recursively sort each half
3. Merge two halves together
```

"The 2-pointer method"

```
Merge (a1[1..n],a2[1..n])
i1=1, i2=1
While (i1<n, i2<n) {
      if (a1[i1] < a2[i2]) {
            Next is a1[i1]
            i1++
      } else {
            Next is a2[i2]
            i2++
      }
}
Now throw in the dregs…
```

The steps of QuickSort



select pivot value

partition S

QuickSort(S_1) and QuickSort(S_2)

Presto! S is sorted

[Weiss]

picking pivot, threshold

BucketSort (aka BinSort)

If all values to be sorted are *known* to be between 1 and *K*, create an array `count` of size *K*, **increment** counts while traversing the input, and finally output the result.

Example *K*=5. Input = (5,1,3,4,3,2,1,1,5,4,5)

`count` array	
1	
2	
3	
4	
5	

2^{32}

Running time to sort n items?

$O(N+K)$

Fixing impracticality: RadixSort

- Radix = “The base of a number system”
  - We’ll use 10 for convenience, but could be anything

- Idea: BucketSort on each **digit**, least significant to most significant (lsd to msd)

Internal versus External Sorting

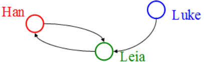
- Need sorting algorithms that minimize disk/tape access time
- **External sorting** – Basic Idea:
  - – Load chunk of data into RAM, sort, store this “run” on disk/tape
  - – Use the Merge routine from Mergesort to merge runs
  - – Repeat until you have only one run (one sorted chunk)
  - – Text gives some examples

Graphs

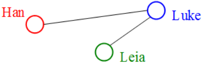
Chapter 9 in Weiss

Graph Definitions

In *directed* graphs, edges have a specific direction:



In *undirected* graphs, they don't (edges are two-way):



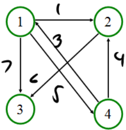
`v` is *adjacent* to **`u`** if **`(u,v)`** $\in$ **`E`**

Representation

- adjacency **matrix**:

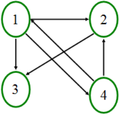
$$A[u][v] = \begin{cases} \text{weight} & , \text{ if } (u, v) \in E \\ 0 & , \text{ if } (u, v) \notin E \end{cases}$$

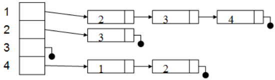
	1	2	3	4
1	0	1	7	0
2				
3				
4				



Representation

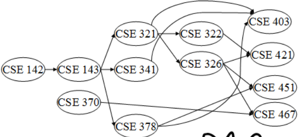
- adjacency **list:**





Application: Topological Sort

Given a directed graph, **`G = (V,E)`**, output all the vertices in **`V`** such that no vertex is output before any other vertex with an edge to it.



DAG

Is the output unique?

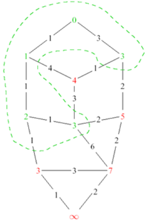
Graph Traversals

- Breadth-first search (and depth-first search) work for arbitrary (directed or undirected) graphs - not just mazes!
  - Must mark visited vertices so you do not go into an infinite loop!
- Either can be used to determine connectivity:
  - Is there a path between two given vertices?
  - Is the graph (weakly) connected?
- Which one:
  - Uses a queue?
  - Uses a stack?
  - Always finds the shortest path (for unweighted graphs)?

Single Source Shortest Paths (SSSP)

Given a graph *G*, edge costs $c_{i,j}$, and vertex *s*, find the shortest paths from *s* to all vertices in G.

Dijkstra's Algorithm: Idea

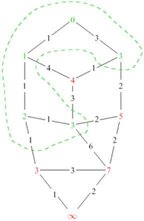


Adapt BFS to handle weighted graphs

Two kinds of vertices:

- Finished or known vertices
  - Shortest distance has been computed
- Unknown vertices
  - Have tentative distance

Dijkstra’s Algorithm: Idea



At each step:

1) Pick closest unknown vertex
2) Add it to known vertices
3) Update distances

Dijkstra’s Algorithm: Pseudocode

```
Initialize the cost of each node to $\infty$

Initialize the cost of the source to 0

While there are unknown nodes left in the graph
    Select an unknown node $b$ with the lowest cost
    Mark $b$ as known
    For each node $a$ adjacent to $b$
        $a$’s cost = min($a$’s old cost,  $b$’s cost + cost of ($b$, $a$))
```

Dijkstra’s Algorithm: a Greedy Algorithm

Greedy algorithms always make choices that *currently* seem the best

- Short-sighted – no consideration of long-term or global issues
- Locally optimal - does not always mean globally optimal!!

Minimum Spanning Trees

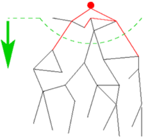
Given an undirected graph **G**=(**V**,**E**), find a graph **G'=(V, E')** such that:

- E' is a subset of E
- |E'| = |V| - 1
- G' is connected
- $\sum_{(u,v)\in E'} c_{uv}$ is minimal

G' is a **minimum spanning tree**.

Applications: wiring a house, power grids, Internet connections

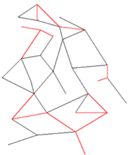
Two Different Approaches



Prim's Algorithm

Almost identical to Dijkstra's

greedy node



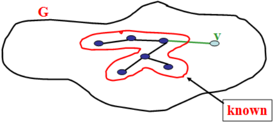
Kruskals's Algorithm

Completely different!

greedy edges

Prim's algorithm

Idea: Grow a tree by adding an edge from the "known" vertices to the "unknown" vertices. Pick the ***edge with the smallest weight.***



Prim's Algorithm for MST

A *node-based* greedy algorithm

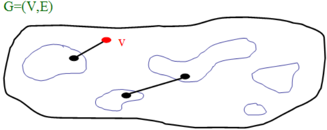
Builds MST by greedily adding nodes

1. Select a node to be the "root"
   - mark it as known
   - Update cost of all its neighbors
2. While there are unknown nodes left in the graph
   a. Select an unknown node *b* with the smallest cost from some *known* node *a*
   b. Mark *b* as known
   c. Add (*a*, *b*) to MST
   d. Update cost of all nodes adjacent to *b*

Note: cost from some a, ***not*** from root

Kruskal's MST Algorithm

Idea: Grow a forest out of edges that do not create a cycle. Pick an ***edge with the smallest weight.***



Kruskal's Algorithm for MST

An *edge-based* greedy algorithm

Builds MST by greedily adding edges

1. Initialize with
   - empty MST
   - all vertices marked unconnected
   - all edges unmarked
2. While there are still unmarked edges
   a. Pick the lowest cost edge `(u,v)` and mark it
   b. If `u` and `v` are not already connected, add `(u,v)` to the MST and mark `u` and `v` as connected to each other

Doesn't it sound familiar?