CSE 326
Lecture 18

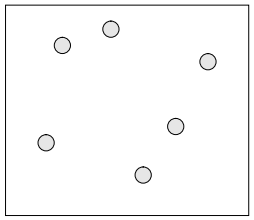
Spring Quarter 1995

Summary of Dictionary Data Structures

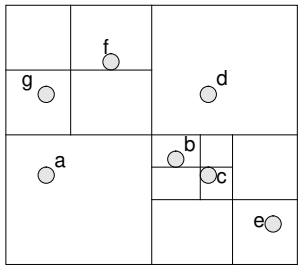
	Find	Insert	Delete	Find-min	List-all	n Finds
Sorted Array	log n	n	n	1	n	n log n
Ordered List	n	n	n	1	n	n^2
Binary Search Tree	n	n	n	n	n	n^2
AVL Tree	log n	log n	log n	log n	n	n log n
Splay Tree	n	n	n	n	n	n log n

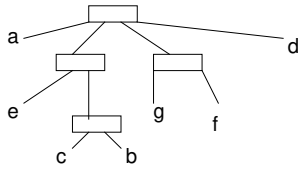
Multidimensional Search Trees

- Example: two dimensions. We want to store a set of ordered pairs (x_1,y_1), (x_2,y_2), (x_3,y_3), ... , (x_n,y_n). There is no specific order to the points as there is with say integers or character strings.

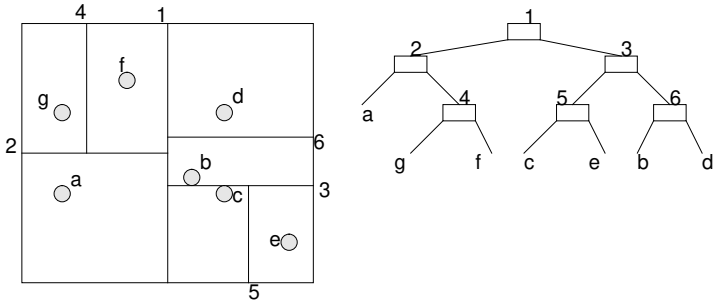


Quad Trees

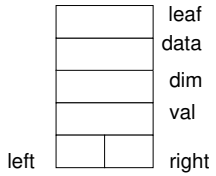




k-d Trees



K-d Trees



- Unlike Quad Trees, the decision has to be made in which dimension to split the data, and where within that dimension the split line should lie. This depends on the application somewhat but a good policy is:
  1. Choose the dimension with the widest spread.
  2. Split the points approximately in half.

Find in K-d Trees

- Find(x,y,n) finds the node which has the pair (x,y) in it and returns null if there is no such node.
- Find(x,y,n) {

```
if n.leaf = true then
   if n.data = (x,y) then return(n) else return null
else
   case
    n.dim =1: if x < n.val then return(Find(x,y,n.left))
                 else return(Find(x,y,n.right))
    n.dim =2: if y < n.val then return(Find(x,y,n.left))
                 else return(Find(x,y,n.right))
   endcase }
```


Hashing

- Hashing is a family of data structures used to efficiently support the operations of insert, delete, and find.
- It cannot be used efficiently for other operations where the order of the data is important. No list-all, range queries, successor, predecessor.
- General Idea:

  We have a key space of size M and we wish to store N keys where N << M

  Examples:

  1. keys are identifiers in a program and we need to store them in a "symbol table" during compilation.

  2. keys are student names and we wish to store student records using their names to look them up.

Simple Hash Table

T

0	
1	
2	**John Smith**
3	
4	**Judy Jones**
5	
6	**Peter Doe**
7	**Henry Ford**
8	
9	

- Hash function

  h: set of possible keys to {0,1,...,HSize-1}

  h("name") is called the hash value of "name"
- Example:

  h(Judy Jones) = 4
  h(Henry Ford) = 7
- To find the entry for "name" we simply calculate T(h("name")).

Hashing Properties

- Generally, hash tables have some unused entries.

$$LoadFactor = \lambda = \frac{N}{HSize}$$

- A good quality hash function distributes the data as evenly as possible over the hash table.
- Upon insertion we may have a collision, h(inserted key) = h(existing key).

  There are different strategies for dealing with collision.

Some Good Hash Functions

- Integers:

  Choose HSize to be a prime and h(n) = n mod HSize
- Character Strings:

  Think of the character string $x=a_0a_1a_2...a_m$ as the integer int(x) = $a_0 + a_1128 + a_2128^2 + ... + a_m128^{m-1}$. We calculate h(x) = int(x) mod HSize.

  This is done using Horner's rule with out computing large integers.
- Computing h(x)

  h := 0;
  for i = m to 0 by -1 do h := a_i + 128h mod HSize;
  return(h)

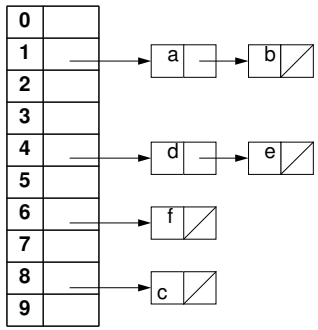
A Bad Hash Function

- Keys able1, able2, able3,..., able4.
- HSize = 128

  int(ablex) mod 128 = int(a)

  Thus, h(ablex) = h(abley) for all x and y.

Open Hashing or Hashing using Chaining



- Collisions h(a) = h(b), h(d) = h(e)
- Choose ordered or unordered chain where appropriate. Common case is the unordered chain because equality checks are easier and there is little advantage with the ordered chain.

Open Hashing Properties

- Load Factor λ

  Unsuccessful search costs λ comparisons on average

  Successful search costs 1 + λ/2 comparisons on average
- Generally, comparisons are to be avoided so choosing λ between 1/2 and 1 is a wise choice.