3137 Data Structures and Algorithms in C++

Lecture 5

July 19 2006

Shlomo Hershkop

Announcements

- midpoint of semester today
  - make sure you are ok with the material covered so far

Outline

- more trees
  - b trees again
  - compression
- Priority queues
  - heaps
- Hash table DS
  - background
  - design
  - applications

Question

- so what is the point of a tree structure ??
- what is the point of a binary search tree ??

B Tree

- great when can’t fit the data in memory
- type of search tree
- trying to make getting to data fast
  - increasing width will make getting to data faster
- M
  - maximum number of children per internal node (we did 3)
- L
  - maximum number of data items per leaf

Rules

1. All data stored at leaf level
2. non leaf nodes store M-1 keys
3. root is either leaf or has between 2 and M children
4. All internal nodes have between M/2 and M children
   1. restricts branching factor
5. All leaves are on the same depth and have between L/2 and L children

- Key
  - the internal node key represents the smallest value on the i+1th subtree
- Leafs can be any DS you choose

- Any ideas on the advantage of the leaf system here ??

- is everyone comfortable with working with B+ trees

Compression

- Many times we need to compress information
  - scaling factor
  - resource allocation
  - over promise
- lossy compression
  - JPEG
  - PNG
- lossless compression
  - when would this be important ?
  - TIF
  - BMP
  - .zip

side note

- is everyone familiar with the tar program??
  - usage
  - how it works ?

- ASCII encodes each character as a 7 bit value
- the idea of compression, is to find a better way of representing your information
  - idea: instead of using uniform length codes for everything, use less bits for higher occurring information parts
- Huffman trees allow you to create very good lossless compression tables to be able to quickly compress text

- Huffman algorithm
  - idea

Hoffman compression

1. Create a frequency count of each of your characters in your file
2. Start to build a binary tree always combining 2 lowest frequencies into one tree the resulting frequency is the combined frequencies
3. Going left is 0, going right is 1

Example

- If I counted:
- E = 29
- A = 14
- T = 10
- B = 4
- D = 2
- C = 1

decompression

- So seeing a code, we simply run down the tree
- As soon as we hit a leaf, translate to that character

Compressing text

- How would you use Huffman to compress text??
- what about decompressing

Priority Queue DS

- DS to keep track of priority
  - can say lowest number is highest priority
- insert
- findmin

Heap

- implementation of priority queue
- Heap order property:
  - any parent is as small or smaller than its children
- can use array representation:
  - no links to manage
  - but need to estimate largest size ahead of time

- used everywhere
  - service priority
  - Operating systems – juggling threads, processes and processors

Example

- When we are interested in k^{th} smallest number in a large number set
- can sort and then pick it out
  - what is the run time ??
- can create BST
  - how will this help
  - what is the run time ?
- can create heap and do k findmin
  - what is the run time ??

percolate up

- insert operation
- idea: bubble up till we reach correct location on the tree

1. create hole in tree to hold value
2. Does it fit ….done
3. else: switch with parent and try again

Runtime for this ??

findmin

- so now we need to find min….
- what is the run time for finding the min ??
- what if we want to find and delete ??

percolate down

- pull out the root
- put hole
- swap with smaller child
- bubble down the hole

run time

- how long would it take me to find an item of specific priority ??
- any ideas on how to help this ?

building a heap

- given a set of N items what is the fastest way of building the heap ??

easy solution

- just do N inserts
  - worst case will be O(n log n)
- when heaps invented that was best way
- they actually have a linear time algorithm….any ideas ?

linear time build

- start in middle
- work way back up

- why it works ?

D-heaps

- we were doing binary heaps, but no reason cant have larger branching factor

- issues:
  - when would be the best time to use heaps ?
  - how to structure it if cant fit entire heap in memory ??

change of pace

- so can we summarize the runtimes for the DS we have covered ??

Question

- if we have 10,000 items
  - how would you store it to quickly support find?
- now what if you only had 20 items
  - how would this be different

Hash Table DS

- This data structure is for organizing an unordered set of items

- find
- insert
- delete

Comparison of average runtime

- Best Tree:
  - AVL
    - find
    - insert
    - delete

- Hash Table
    - find
    - insert
    - delete

- Hash Function
  - mapping function between items and locations in the hashtable

- let me do a graphical example with a bunch of names

Issues

- What hash function to use ?

- What do you do about collisions??

Example

- Lets say you need a dictionary
- For each word insert in hash table
  - runtime ?
- when I need to look up a word call find on hash table
  - runtime ?

hash functions

- The truth is that hash functions should be based on the data
- lets step through some examples

Option 1: integral keys

- items are numbers
- can use them directly to compute hash
- Hash(key) = key % Tablesize
- Example
- Question : why not use randomness to make sure to avoid collisions ?

Option 2: String key

- Hash(key) = sum of ascii values
- Hash(abc) = 97 + 98 + 99
- any idea if this will work ?

- Counter example:
  - dictionary
  - tablesize 40,000
  - what is the maximum word size
  - what would be the max value returned by the hash ??

Option 3: power

- lets add some spread to the summation
- Hash(ley) = key[1]*26^0 + key[1]*26^1 * ..key[i]*26^i

issues

- non uniform distribution of characters in the english language
- only 28% of your table will actually be reached
- collisions!

Option 4: Adjusted power

- Hash(ley) = (key[1]*37^0 + key[1]*37^1 * ..key[i]*37^i) % tablesize

- need to make sure it will be positive
- java uses 31^i
- performs well on general strings

- ok so now we know how to get things into the table
- what do you do when 2 things map to same array location ??

Option 1: Separate Chaining

- At each array location have a linked list
  - how would the insert in the LL work ?
- how do you perform a find on the hash table ?

Option 2: open addressing

- if collision occurs, will try to find alternate cell in the array to store item
- lets see how this works

strategy

- first try hash(x)
- if full
  - try Hash(x) + f(i) % tablesize to locate
  - f is used to move around the array to find a location to use
  - different options, any ideas ?

Linear probing

- f(i) = i
- Example
- can you think of any issues ?

clustering

- linear probing suffers from a problem called clustering
- domino affect

Quadratic probing

- $f(i) = i^2$
- how will this affect clusters ?

Theorem

- if quadratic probing is used and table size is prime, and table is at least half empty then we will always find a spot for a new element

Option 3: Double Hashing

- Apply a second hash function H_2 and probe at distance i * $hash_2(x)$

- f(i) = rehash(i)

- hash(x) + i*$f_i(x)$

- Note:
  1. can't return 0
  2. entire table must be addressable

Load factor

- number of element
- divided by
- table size

growing

- So how do you resize a hash ??

deletion

- how would deletion work
- any issues?

Extendible Hashing

- setup similar to B+ tree
- hashing routine which has growth built in
- use partial bits for keys
- when need to grow will use more bits

question

- from the data structures we have covered which is the most space efficient ??

Wrapping up

- Say you want to add a new operation to heaps
- DecreasePriority (p,d)
  - want to subtract d from priority p
  - any ideas on run time ??

Next time

- Reading
  - chapter 5, chapter 7