Lecture 10: Big O

Wednesday, February 13, 2019 11:14 AM

introduction

1. basics of big O
    a. measures how fast it takes for an algorithm to run
        i. bad way to measure algorithm speed is to use the time or to use the number of instructions it takes
    b. better way to measure is the number of instructions used by an algorithm as a function of the size of the input data
    c. worst case scenario

computing big-o

1. process
    a. determine number of operations an algorithm performs f(n)
        i. operations:
            1) accessing an item
            2) evaluating a mathematical expression
            3) traversing a single link in a linked list, etc...
    b. keep most significant term of that function and throw rest away
    c. remove an constant multipliers from the function
    d. this is your Big-O
2. tips
    a. just look at the most frequently occuring operations to save time
3. strategy
    a. loocate all loops that don't run for a fixed number of iterations and determine max number of iterations
    b. turn these loops into fixed number of iterations using max possible iteration count
    c. then do Big-O count
4. multi-input algorithms
    a. algorithm that operate on two or more *independent* data sets, make sure to include both
        i. you don't know which one is bigger
    b. must still eliminate constants and coefficients for invididual variables

STL and Big O

STL and Big Oh Cheat Sheet

When describing the Big-O of each operation (e.g. insert) on a container (e.g., a vector) below, we assume that the container holds n items when the operation is performed.

Name: list
Purpose: Linked list
Usage: list<int> x; x.push_back(5);
Inserting an item (top, middle*, or bottom): O(1)
Deleting an item (top, middle*, or bottom): O(1)
Accessing an item (top or bottom): O(1)
Accessing an item (middle): O(n)
Finding an item: O(n)
*But to get to the middle, you may have to first iterate through X items, at cost O(x)

Name: vector
Purpose: A resizable array
Usage: vector<int> v; v.push_back(42);
Inserting an item (top, or middle): O(n)
Inserting an item (bottom): O(1)
Deleting an item (top, or middle): O(n)
Deleting an item (bottom): O(1)
Accessing an item (top, middle, or bottom): O(1)
Finding an item: O(n)

Name: set
Purpose: Maintains a set of unique items
Usage: set<string> s; s.insert("Ack!");
Inserting a new item: $O(\log_2 n)$
Finding an item: $O(\log_2 n)$
Deleting an item: $O(\log_2 n)$

Name: map
Purpose: Maps one item to another
Usage: map<int,string> m; m[10] = "Bill";
Inserting a new item: $O(\log_2 n)$
Finding an item: $O(\log_2 n)$
Deleting an item: $O(\log_2 n)$

Name: queue and stack
Purpose: Classic stack/queue

If instead of holding n items, a container holds

Usage: queue<long> q; q.push(5);
Inserting a new item: O(1)
Popping an item: O(1)
Examining the top: O(1)

p items, then just replace "n" with "p" when you do your analysis.

54

Hash Tables vs. Binary Search Trees

	Hash Tables	Binary Search Trees
Speed	O(1) regardless of # of items	$O(\log_2 N)$
Simplicity	Easy to implement	More complex to implement
Max Size	Closed: Limited by array size Open: Not limited, but high load impacts performance	Unlimited size
Space Efficiency	Wastes a lot of space if you have a large hash table holding few items	Only uses as much memory as needed (one node per item inserted)
Ordering	No ordering (random)	Alphabetical ordering

1. determining big O
   a. first determine the max number of items each container could possibly hold
   b. assume it always holds *n* items
2. for a vector< set<int> > v
   a. finding a 7 in any set: $N\log_2(Q)$ because it takes N steps to search a vector and $\log_2(Q)$ to search through a set

Sorting (inefficient)

1. details
   a. items, what are we sorting
   b. rules, how do we order them
   c. constraints, RAM/disk, array/linked list?
   d. cannot be log(N)
      i. fastest will always be nlog(N), usually
2. rules
   a. don't choose sorting algorithm until you understand requirements of the problem
   b. choose simplest sorting algorithm that meets requirements
3. stable vs. unstable sort
   a. stable sort takes into account initial ordering, maintaining order of similar-valued items
   b. unstable sorting re-orders items without taking into account their initial ordering

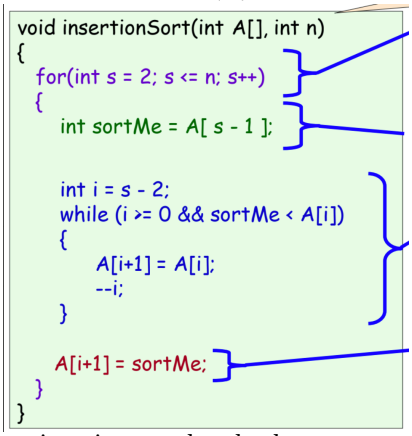
selection sort

1. *after one iteration*
   a. leaves the smallest value at the bottom
2. sorting from smallest to biggest in ascending order
3. go through and find the smallest value
4. swap it with the first index and move index up one
5. find next smallest, swap, and move on and continue
6. steps
   a. N steps, 1 step, n-1 steps, 1 step, n-2 steps, 1 step
   b. N swaps happen
7. big O
   a. $O(N^2)$
8.

```
void selectionSort(int A[], int n)
{
  for (int i = 0; i < n; i++)
  {
    int minIndex = i;
    for (int j = i+1; j < n; j++)
    {
      if (A[j] < A[minIndex])
        minIndex = j;
    }
    swap(A[i], A[minIndex]);
  }
}
```

insertion sort

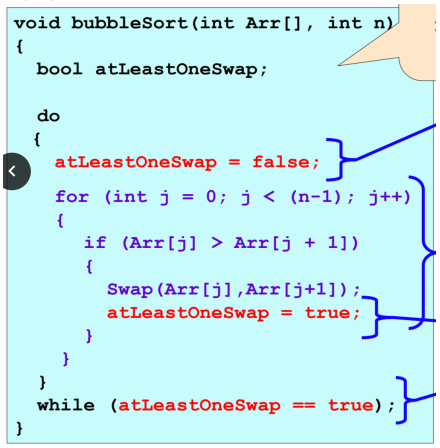
1. *after one iteration*

a. first portion (after the marker) will be sorted
2. insert each element into proper position within the first portion of the array and shift all following books to the right
3. worst case scenario: $O(n^2)$
4. 
```
void insertionSort(int A[], int n)
{
  for(int s = 2; s <= n; s++)
  {
    int sortMe = A[ s - 1 ];

    int i = s - 2;
    while (i >= 0 && sortMe < A[i])
    {
        A[i+1] = A[i];
        --i;
    }

    A[i+1] = sortMe;
  }
}
```
5. use insertion sort when the elements are mostly in order already
6. unstable sort!

bubble sort

1. *after one iteration*
   a. largest will be at the end
   b. first portion will be sorted
2. compare two elements at a time, swap if they're out of order, move to next two elements. and keep iterating through array until no swaps occur
3. $O(N^2)$
4. 
```
void bubbleSort(int Arr[], int n)
{
  bool atLeastOneSwap;

  do
  {
    atLeastOneSwap = false;
    for (int j = 0; j < (n-1); j++)
    {
      if (Arr[j] > Arr[j + 1])
      {
        Swap(Arr[j],Arr[j+1]);
        atLeastOneSwap = true;
      }
    }
  }
  while (atLeastOneSwap == true);
}
```
5. stable sort!

notes

1. with one iteration through the array, bubble sort would have the largest element at the front. insertion sort has first 2 elements sorted. insertion sort has smallest integer in front

shell sort

1. uses h-sorting
   a. algorithm:
      i. pick value of h
      ii. if a[i] and a[i+h] are out of order
         1) swap two elements
      iii. if swapped any elements during last pass, repeat process with the same h value
   b. h-sorting with h = 1 is bubble sort

Lecture 11: Sorting/Trees

Monday, February 25, 2019 11:08 AM

note

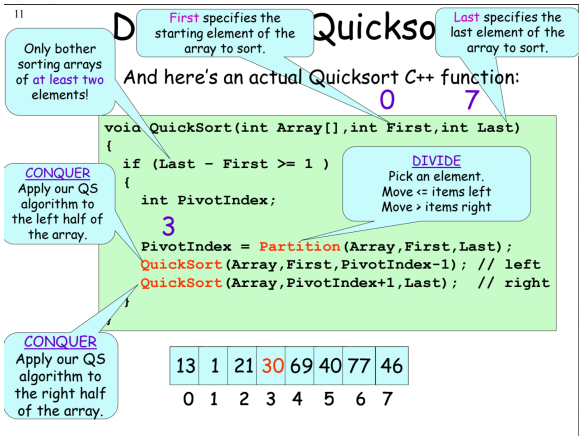
1. know whether it's log(p) + n or if it's nlog(p)
2. log(p) + n occurs when you're searching through a vector/list of a key just once

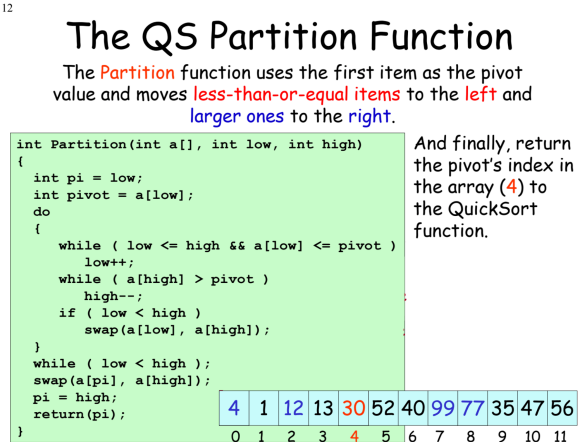
advanced sorting

1. divide elements into two groups of roughly equal size
2. sort each of these smaller groups
3. combine sorted groups into one large sorted list
4. usually uses recursion

quicksort

1. *after one iteration*
   a. all values after a certain point (pivot) will be smaller than all the values on the right side of the pivot
2. algorithm
   a. if array contains 0 or 1 element, return
   b. select arbitrary element *p* from array (pivot: usually first element)
   c. move all elements less than or equal to *p* to left of the array and all elements greater than *p* to the right (partition)
   d. recursively repeat process on the left sub-array and then right sub-array
3. code
   a.





4. big-O
   a. worst case is nlog(n)
5. details
   a. avoid for arrays that are already sorted
   b. requires N^2 for arrays that are mostly in order

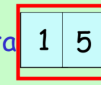
c. unstable; can be parallelized, varied ram, does not work on a singly linked list (yes on doubly linked lists)

merge sort

1. takes to pre-sorted arrays as inputs and outputs a combined, third sorted array
2. basic idea
   a. initialize counter variables i1, i2 to zero
   b. while there are more items to copy
      i. if a1[i1] less than a2[i2]
         1) copy a1[i1] to output array b and i1++
      ii. else
         1) copy a2[i1] to output array b and i2++
   c. if either array runs out, copy entire contents of other array over
3. code
   a.

```
void merge(int data[],int n1, int n2)
{
  int i=0, j=0, k=0;
  int *temp = new int[n1+n2];
  int *sechalf = data + n1;

  while (i < n1 || j < n2)
  {
    if (i == n1)
       temp[k++] = sechalf[j++];
    else if (j == n2)
       temp[k++] = data[i++];
    else if (data[i] <= sechalf[j])
       temp[k++] = data[i++];
    else
       temp[k++] = sechalf[j++];
  }
  for (i=0;i<n1+n2;i++)
    data[i] = temp[i];
  delete [] temp;
}
```



4. algorithm
   a. if array has one element, return
   b. split array into two equal sections
   c. recursively call mergesort function on left half
   d. recursively call mergesort function on right half
   e. merge two halves using merge function
5. big-o
   a. $n\log_2 n$ (keep dividing in half) until just 1 book left
6. details
   a. yes, you can apply it to linked lists (you aren't jumping around)
   b. merge sort is usually not a stable sort
   c. can be parallelized (broken up across cores)

sorting cheat sheet

Sort Name	Stable/ Non-stable	Notes
Selection Sort	Unstable	Always $O(n^2)$, but simple to implement. Can be used with linked lists. Minimizes the number of item-swaps (important if swaps are slow)
Insertion Sort	Stable	$O(n)$ for already or nearly-ordered arrays. $O(n^2)$ otherwise. Can be used with linked lists. Easy to implement.
Bubble Sort	Stable	$O(n)$ for already or nearly-ordered arrays (with a good implementation). $O(n^2)$ otherwise. Can be used with linked lists. Easy to implement. Rarely a good answer on an interview!
Shell Sort	Unstable	$O(n^{1.25})$ approx. OK for linked lists. Used in some embedded systems (eg, in a car) instead of quicksort due to fixed RAM usage.
Quick	Unstable	$O(n \log n)$ average. $O(n^2)$ for already/mostly/reverse ordered arrays or

Quick Sort	Unstable	$O(n \log_2 n)$ average, $O(n^2)$ for already/mostly reverse ordered arrays of arrays with the same value repeated many times. Can be used with linked lists. Can be parallelized across multiple cores. Can require up $O(n)$ slots of extra RAM (for recursion) in the worst case, $O(\log_2 n)$ avg.
Merge Sort	Stable	$O(n \log_2 n)$ always. Used for sorting large amounts of data on disk (aka "external sorting"). Can be used to sort linked lists. Can be parallelized across multiple cores. Downside: Requires n slots of extra memory/disk for merging – other sorts don't need xtra RAM.
Heap Sort	Unstable	$O(n \log_2 n)$ always. Sometimes used in low-RAM embedded systems because of its performance/low memory req'ts.

Lecture 12: Binary Trees

Wednesday, February 27, 2019 10:58 AM

trees

1. all log(n) for binary trees
2. details
   a. the top node is called the root node
   b. every node can have zero or more children nodes
   c. every node has its sub-trees
   d. removing whole sections of trees is called "pruning"
   e. adding whole sections to trees is called "grafting"
3. traversing the tree
   a. always start with the root node
   b. four common ways to traverse a tree
      i. pre-order traversal
         1) process current node
         2) process nodes in left sub-tree
         3) process nodes in right-sub tree
         4)

```
void PreOrder(Node *cur)
{
   if (cur == NULL)           // if empty, return...
      return;

   cout << cur->value;       // Process the current node.

   PreOrder(cur->left);     // Process nodes in left sub-tree.
   PreOrder(cur-> right);  // Process nodes in left sub-tree.
}
```

      ii. in-order traversal
         1) process the nodes in the left sub-tree
         2) process the current node
         3) process the nodes in the right sub-tree
         4)

```
void InOrder(Node *cur)
{
   if (cur == NULL)           // if empty, return...
      return;

   InOrder(cur->left);     // Process nodes in left sub-tree.

   cout << cur->value;      // Process the current node.

   InOrder(cur->right);  // Process nodes in right sub-tree.
}
```

      iii. post-order traversal
         1) process the nodes in the left sub-tree
         2) process the nodes in the right sub-tree
         3) process current node

```
void PostOrder(Node *cur)
```

4)

```
{
   if (cur == NULL)            // if empty, return...
      return;
   PostOrder(cur->left);       // Process nodes in left sub-tree.
   PostOrder(cur-> right);     // Process nodes in right sub-tree.
   cout << cur->value;         // Process the current node.
}
```

iv. level-order traversal
   1) visit each leve's nodes from left to right before visiting nodes in the next level
   2) use temp pointer and queue of node pointers
   3) insert root node pointer into queue
   4) while queue is not empty
      a) dequeue the top node
      b) process the node
      c) add nodes children to queue if not null

4. how to remember
   a. starting just above left of root node, draw a loop counter-clockise around all of the nodes and draw dots (represent order of traversal)
   b. pre-order
      i. draw a dot next on the left side of the dot
   c. in order
      i. draw a dot under the node
   d. post-order
      i. draw dot on right
   e. level-order
      i. draw horizontal lines, goes from left to right
5. expression evaluation
   a.
      1. If the current node is a number, return its value.
      2. Recursively evaluate the left subtree and get the result.
      3. Recursively evaluate the right subtree and get the result.
      4. Apply the operator in the current node to the left and right results; return the result.

binary search tree

1. details
   a. for every node x, every node in the left sub-tree is less than x
   b. every node in the right sub-tree is greater than x
2. operations
   a. determine if tree is empty
   b. search binary search tree for a value
      i. log(n)
   c. insert an item in the binary search tree
      i. log(n)
   d. delete an item from the binary search tree
   e. find the numebr of nodes and leaves in the binary search tree
   f. traverse the binary search tree
   g. free the memory used by the binary search tree

searching a binary search tree

1. algorithm

start at root of the tree
keep going until hit a null pointer
    if v is equal to current node value, return found
    if v is less than current node's value, go left
    if v is greater than current node's value, go right
if we hit a null pointer, not found

2. recursive search
   a.
```
bool Search(int V, Node *ptr)
{
  if (ptr == NULL)
    return(false);  // nope
  else if (V == ptr->value)
    return(true);  // found!!!
  else if (V < ptr->value)
    return(Search(V,ptr->left));
  else
    return(Search(V,ptr->right));
}
```
3. iterative search
   a.
```
bool Search(int V,Node *ptr)
{
  while (ptr != NULL)
  {
    if (V == ptr->value)
      return(true);
    else if (V < ptr->value)
      ptr = ptr->left;
    else
      ptr = ptr->right;
  }
  return(false); // nope
}
```
4. big-o
   a. if mostly balanced, then you eliminate about half of your tree every time you go down a link
      i. $\log_2(n)$
   b. worst case
      i. n

inserting items

1. algorithm
   a. if tree is empty, allocate new node and put v into it
      i. point root pointer to our new node
   b. start at root of the tree
   c. while we're not done:
      i. if v is equal to current node's value, done!
      ii. if v is less than current node's value
         1) if there is a left child, go left
         2) otherwise allocate a new node and put v into it and set current node's left pointer to new node. done!
      iii. if v is greater than current node's value
         1) if there is a right child, then go right
         2) otherwise allocate a new node and put v into it
            a) set current node's right pointer to new node
2. code
   a.
```
void insert(const std::string &value)
{
  if (m_root == NULL)
    {  m_root = new Node(value);  return; }
  Node *cur = m_root;
  for (;;)
  {
    if (value == cur->value)  return;
    if (value < cur->value)
    {
      if (cur->left != NULL)
        cur = cur->left;
      else
      {
        cur->left = new Node(value);
        return;
      }
    }
    else if (value > cur->value)
    {
      if (cur->right != NULL)
```

```
            cur = cur->right;
        else
        {
            cur->right = new Node(value);
            return;
        }
    }
  }
}
```

b. note: for (;;) is faster than while(true)!

finding min & max

1. min -- you just go all the way left
2. max -- you just go all the way right
3.

```
int GetMin(node *pRoot)
{
  if (pRoot == NULL)
    return(-1);  // empty

  while (pRoot->left != NULL)
     pRoot = pRoot->left;

  return(pRoot->value);
}
```

```
int GetMax(node *pRoot)
{
  if (pRoot == NULL)
    return(-1);  // empty

  while (pRoot->right != NULL)
     pRoot = pRoot->right;

  return(pRoot->value);
}
```

4. log(n)

printing a BST in order

1. in-order traversal

freeing the whole tree

1. recursively free the entire tree
2. post order traversal
3.

```
void FreeTree(Node *cur)
{
   if (cur == NULL)          // if empty, return...
       return;

   FreeTree(cur->left);     // Delete nodes in left sub-tree.

   FreeTree (cur->right);  // Delete nodes in right sub-tree.

  delete cur;              // Free the current node
}
```

4.

Lecture 13: More Binary Search Trees

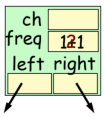
Monday, March 4, 2019 10:23 AM

deleting node from BST

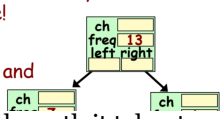
1. case 1: zero children
    a. sub-case 1: target node is not the root node
        i. unlink parent node from target node by setting parent's link to nullptr
        ii. delete target node
    b. sub-case 2: target node is the root node
        i. set root pointer to null and delete the target node
2. case 2: one child
    a. sub-case #1: target node is not the root node
        i. relink parent node to target node's (only) child
        ii. delete the target node
    b. sub-case #2: target node is the root node
        i. set root pointer to the only child
        ii. delete the target node
3. case 3: two children
    a. look at left and right children
    b. we don't actually delete node *k*
    c. we replace the value of *k* with the value of ***either***
        i. left subtree's largest valued child (child all the way to the right)
        ii. right subtree's smallest-valued child

huffman encoding

1. compressing files
2. for each of the characters (letters)...
3. A. Create a binary tree leaf node for each entry in our table, but don't insert any of these into a tree!
   B. While we have more than one node left:
   1. Find the two nodes with lowest freqs.
   2. Create a new parent node.
   3. Link the parent to each of the children.
   4. Set the parent's total frequency equal to the sum of its children's frequencies.
   5. Place the new parent node in our grouping.



4. after that, we create a new bit-encoding for each character
5. A. Create a binary tree leaf node for each entry in our table, but don't insert any of these into a tree!
   B. Build a binary tree from the leaves.
   C. Now label each left edge with a "0" and each right edge with a "1"



    a. the new bit encoding is the path it takes to get to the node
    b. for instance, getting to a character in the second level is either 01 or 10 or even 11
6. decoding
7. 1. Extract the encoding scheme from the compressed file.
   2. Build a Huffman tree (a binary tree) based on the encodings.
   3. Use this binary tree to convert the compressed file's contents back to the original characters.
   4. Save the converted (uncompressed) data to a file.

how to get depth

1. code

```
int getDepth(Node* head)
{
    if (head != nullptr)
```

```
            return 0;
        else
        {
            depthLeft = getDepth(head->left);
            depthRight = getDepth(head->right);
            if (depthLeft > depthRight)
                return 1 + depthLeft;
            else
                return 1 + depthRight;
        }
    }
```

balanced search trees

1. 2-3 trees, red-black trees, and avl trees
2. avl tree
    a. keeps track of height for sub-trees and if height of the subtrees under any node is different by more than one level, the avl algorithm shifts nodes around to maintain balance
3. just know that a balanced BST are always O(log n)
4. always ask interviewer if the BST is balanced
    a. could make or break your interview

Lecture 14: Hash Tables

Monday, March 4, 2019 11:41 AM

hash table basics

1. most efficient ways to search for data
2. modulo operator %
    a. used to divide two numbers and obtain the remainder
    b. if we modulo by n, the results will be between 0 to n-1
3. each slot in the hash table is called a "bucket"

inserting into hash tables

1. use a hash function
    a. in this case, we use the modulo division operator that converts an extremely large number into a smaller number
2. collision
    a. when a hash function produces the same output for two values
3. fixing collisions
    a. closed hash table with "linear probing"
    b. open hash table

closed hash table with "linear probing"

1. if the target bucket is empty, store whole value
2. if bucket is occupied, scan from that bucket until we hit the first open bucket
    a. if there are no buckets available, do not insert (fixed size array)
3. if we hit the bottom bucket and it's filled, we start at the top until we hit the original bucket
4. searching for values
    a. if we don't find out value, we probe down our array until we find our array or until we hit an empty bucket
    b. if there's an empty bucket or we hit the original bucket, then we know it's not there
5. insertion/searching time complexity
    a. usually O(1)
    b. worst case is O(N)
6. deleting values from hash table
    a. can't just set it to false because then we will not see items
    b. do not use linear probing closed hash tables for no deletions

open hash table

1. insertion
    a. instead of storing values directly in array, each array bucket points to a linked list of values
    b. as usual, compute bucket using hash function
    c. add new value to linked list at array[bucket]
2. searching
    a. if it's in the tree, it's true
3. deleting
    a. just remove node from the linked list
    b. if the linked list is empty, set the pointer to null in your array

closed hash table efficiency

1. (nearly) empty hash table
    a. insertion/searching
        i. O(1)
2. (nearly) full
    a. insertion/searching
        i. worst case: O(N)
        ii. average case: O(1)
3. load factor
    a. maximum number of values you intend to add divided by number of buckets in the array
    b. higher load means more full; max is 1

c. Given a particular load L for a Closed Hash Table w LP, it's easy to compute the average # of tries it'll take you to insert/find an item:

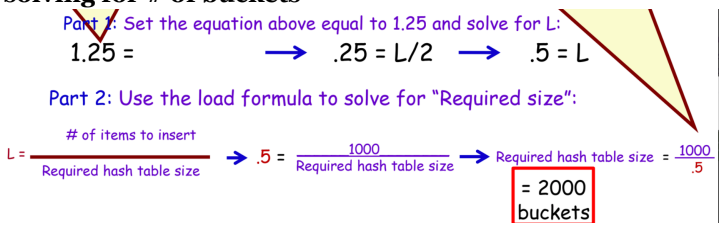
Average # of Tries = $\frac{1}{2}(1 + 1/(1-L))$ for L < 1.0

open hash table efficiency

1. load factor equation
2. L = load = *number/total*
3. Given a particular load L for an Open Hash Table, it's also easy to compute the average # of tries to insert/find an item:

Average # of Checks = 1 + L/2

solving for # of buckets

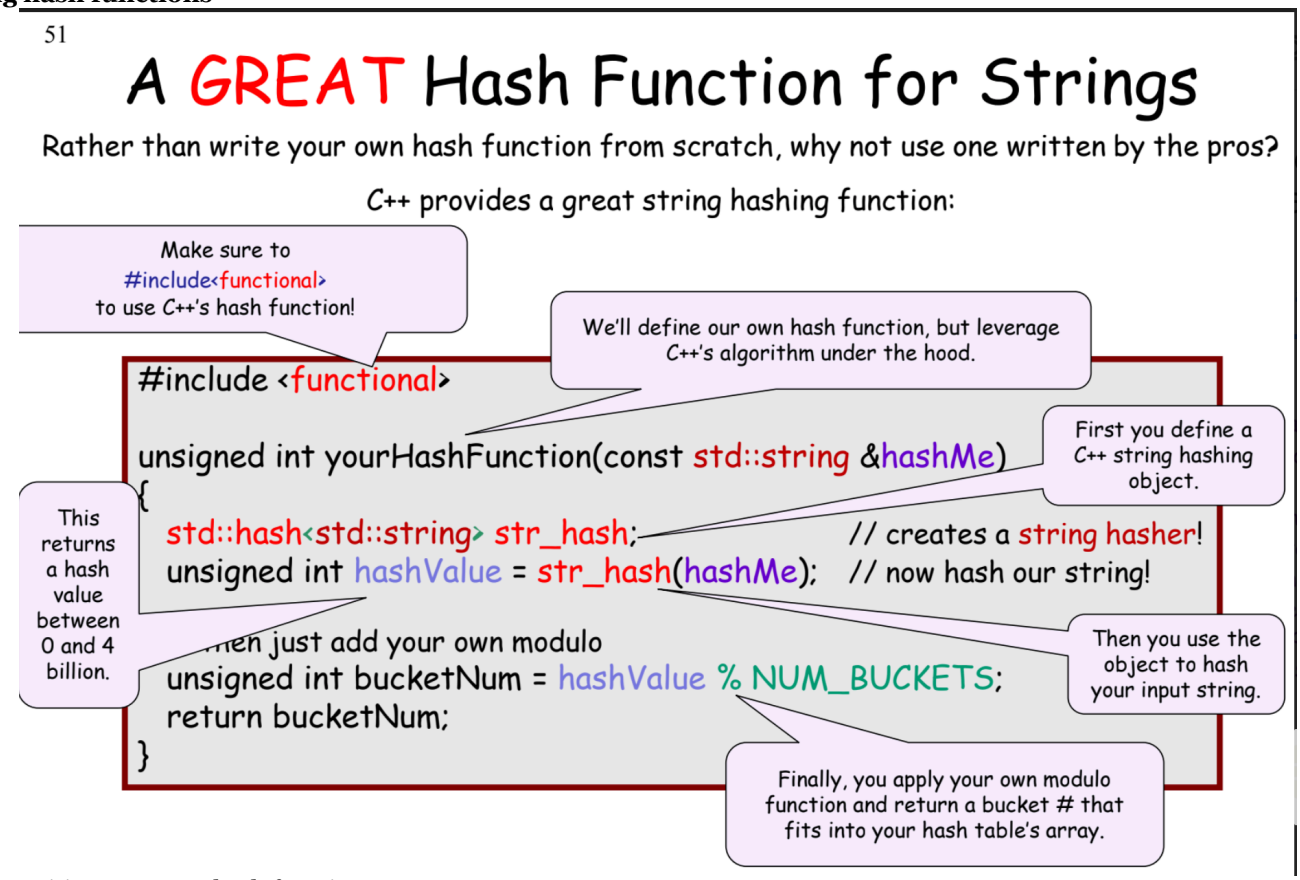


details

1. efficiency trade off
   a. really big hash table with too many buckets and get fast searches but you waste a lot of memory; small hash tables will be slower
2. always try to choose prime number of buckets! reduces collisions

writing hash functions

1. 51



2. writing your own hash function
   a. understand the "nature" of the data
   b. design algorithm, analyze it, iterate
   c. hash function must always give us the same bucket # for a given input value

unordered map

The unordered_map: A hash-based version of a map

```
#include <unordered_map>
#include <iostream>
#include <string>
using namespace std::tr1; // required for a hash-based map
using namespace std;

int main( )
{
  unordered_map <string,int> hm;              // define a new U_M
  unordered_map <string,int>::iterator iter; // define an iterator for a U_M
  hm["Carey"] = 10;                          // insert a new item into the U_M
  hm["David"] = 20;
  iter = hm.find("Carey");                   // find Carey in the hash map
  if (iter == hm.end())                      // did we find Carey or not?
     cout << "Carey was not found!";         // couldn't find "Carey" in the hash map
  else
  {
      cout << "When we look up " << iter->first;       // "When we look up Carey"
      cout << " we find " << iter->second;            // "we find 10"
  }
}
```

hash tables vs. binary search trees

24

Hash Tables vs. Binary Search Trees

	Hash Tables	Binary Search Trees
Speed	O(1) regardless of # of items	$O(\log_2 N)$
Simplicity	Easy to implement	More complex to implement
Max Size	Closed: Limited by array size Open: Not limited, but high load impacts performance	Unlimited size
Space Efficiency	Wastes a lot of space if you have a large hash table holding few items	Only uses as much memory as needed (one node per item inserted)
Ordering	No ordering (random)	Alphabetical ordering

tables

1. a group of related data is called a "record"
   a. struct
   b. class
2. each record has "fields" that can hold values
3. if we have a bunch of records, this is a table
4. key field
   a. unique fields
5. creating tables
   a. array/vector of struct
6. making table more efficient *called indexes*
   a. use vector to hold records
   b. data structure associating name with slot #
   c. another data structure associating ID # with slot #
   d. you always have to add to the indexes when you do anything

Chapter 15: Heaps

Monday, March 11, 2019 10:35 AM

priority queues

1. details
    a. each item has a priority rating indicating how important it is
    b. when you dequeue, it dequeues the item with the highest priority
    c. usually will have a 98% probability of processing the highest priority and 2% of the time will take the lower priority
        i. this way lower priority items don't get stuck
2. operations
    a. insert
    b. get
    c. remove
    d. specify what the priority is
3. data structures
    a. three queues
    b. if we have a ton of priorities, use heap!

heaps

1. binary tree but *not binary search tree*
2. use a "complete" binary tree
    a. top n-1 levels are filled with nodes
    b. all nodes on bottom most level must be as far left as possible
3. maxheap
    a. quickly insert a new item into the heap
    b. quickly retrieve the largest item from the heap
4. minheap
    a. quickly insert a new item into the heap
    b. quickly retrieve the smallest item from the heap

maxheap/minheap

1. max
    a. value in a node is always greater than or equal to values of the node's children
    b. tree is complete binary tree
2. min
    a. value in a node is always less than or equal to values of its two children
    b. also a complete binary tree

extracting items

1. if tree is empty, return error
2. otherwise, top item is biggest value
3. if heap has one node, delete it and return saved value
4. copy value from the right-most node in bottom-most row to root node
5. delete that node
    a. to find the bottom-most right-most node, use an array
6. repeatedly swap the just-moved value with larger of its two children until value is greater than or equal to both of its children
    a. sifting down
7. return saved value to the user

adding nodes

1. if empty, create new root node and return
2. otherwise insert new node in the bottom-most left-most position of the tree
3. compare new value with parent value
4. if new value is greater than parent value, then swap htem
5. repeat steps until proper place

a. while loop

array

1. size
   a. we know each level has twice number of nodes as previous level
   b. copy each level over
2. int variable
   a. holds how many items are in the heap
3. properties
   a. root value is heap[0]
   b. bottom-most, right-most node is heap[count - 1]
   c. add or remove by setting heap[count] = value and updating count
4. Challenge: Come up with a formula to locate a node's children

   Hint: It's of the form leftChild(parent) = 2 * parent + 1
   rightChild(parent) = 2 * parent + 2
   a. parent = (child - 1) / 2
      i. integer math!!!
5.

```
class HeapHelper
{
  HeapHelper()             { num = 0;  }
  int GetRootIndex()       { return(0); }
  int LeftChildLoc(int i)  { return(2*i+1); }
  int RightChildLoc(int i) { return(2*i+2); }
  int ParentLoc(int i)     { return((i-1)/2); }
  int PrintVal(int i)      { cout << a[i]; }
  void AddNode(int v)      { a[num] = v; ++num;}
private:
  int a[MAX_ITEMS];
  int num;
};
```

a[0]	123
[1]	42

extracting from a maxheap -- array version

1. If the count == 0 (it's an empty tree), return error.
2. Otherwise, heap[0] holds the biggest value. Remember it for later.
3. If the count == 1 (that was the only node) then set count=0 and return the saved value.
4. Copy the value from the right-most, bottom-most node to the root node: heap[0] = heap[count-1]
5. Delete the right-most node in the bottom-most row: count = count - 1
6. Repeatedly swap the just-moved value with the larger of its two children:
   Starting with i=0, compare and swap:
   heap[i] with heap[2*i+1] and heap[2*i+2]
7. Return the saved value to the user.

adding to a maxheap -- array version

1. Insert a new node in the bottom-most, left-most open slot:
   heap[count] = value
   count = count + 1;
2. Compare the new value heap[i] with its parent's value: heap[(i-1)/2]
3. If the new value is greater than its parent's value, then swap them.
4. Repeat steps 2-3 until the new value rises to its proper place or we reach the top of the array.

complexity

1. worst case is log(n) to insert a new item
2. guaranteed to be log(n) levels deep

"naive" heapsort

1. insert all numbers into a new maxheap
2. while there are numbers left in the heap
    a. remove biggest value from heap
    b. place it in the last open slot of the array

"efficient" heapsort

1. convert our input array into a maxheap
    a. algorithm:

```
for(curNode = lastNode through rootNode)
        focus on subtree rooted at curNode
        consider the subtree has a maxHeap
        shift top value down until subtree becomes maxHeap


/* Faster version. */
startNode = N/2 - 1 // ignore the bottom children
for(curNode = lastNode through rootNode)
        focus on subtree rooted at curNode
        consider the subtree has a maxHeap
        shift top value down until subtree becomes maxHeap
```

2. while numbers left in heap
    a. remove biggest value
    b. reheapify (see algorithm below)
    c. place in last open slot of array
3. time complexity
    a. part 1: O(n)
    b. part 2: O(nlogn)
    c. thus the total complexity is O(nlogn)

reheapification algorithm

```
copy value from right-most node in bottom-most node to the root node
delete right-most node in the bottom-most row
repeated swap just-moved value with larger of two children until the value is greater than or equal to both
of its children
```

Chapter 16: Graphs

Wednesday, March 13, 2019 10:22 AM

graphs

1. ADT that stores set of entities and keeps track of relationships between them
2. two items
    a. vertices (nodes)
    b. edges (arcs)
        i. connects vertices
3. types of graphs
    a. directed graph
        i. one vertex goes to another
    b. undirected graph
        i. bidirectional

adjacency matrix

1. when to use
    a. when there are fewer bidirectional relationships
2. double-dimensional array
    a. size of both dimensions is number of verticles in graph
        i. bool graph[*n*][*n*]
    b. each element holds whether or not there is an edge between the two vertices from *i* to *j* (adjacency matrix)
    c. to bidirectionally connect vertices, set both [*i*][*j*] and [*j*][*i*] to true
3. matrix multiplication
    a. if you multiply the matrix by itself, you can find vertices that are two edges apart (represented by a *truthy* value in the new matrix)
    b. if you multiply the old matrix by the new matrix, you can find vertices that are three edges apart

adjacency list

1. definition
    a. array of *n* linked lists that represents a directed graph of *n* vertices
        i. list<int>graph[*n*]
    b. adding *j* to list number *i* means that there is edge from *i* to *j*
2. when to use
    a. when there are many bidirectional relationships

graph traversals

1. depth-first
    a. keep going until dead end or previously visited vertex
    b. backtrack and try another path
    c.

```
Depth-First-Traversal(curVertex)
{
   If we've already visited  the current vertex
      Return

   Otherwise
      Mark the current vertex as visited
      Process the current vertex (e.g., print it out)

      For each edge leaving the current vertex
         Determine which vertex the edge takes us to
         Call Depth-First-Traversal on that vertex
}
```

    d.

```
Depth-First-Search-With-Stack(start_room)
  Push start_room on the stack
  While the stack is not empty
        Pop the top item off the stack and put it in variable c
        If c hasn't been visited yet
            Drop a breadcrumb (we've visited the current room)
```

```
        For each door d leaving the room
        If the room r behind door d hasn't been visited
            Push r onto the stack.
```

2. breadth-first
    a. concentric circles
    b. explore all vertices *n* away from start
    c.

```
Breadth-First-Search (startVertex)
{
  Add the starting vertex to our queue
  Mark the starting vertex as "discovered"
  While the queue is not empty
     Dequeue the top vertex from the queue and place in c
     Process vertex c (e.g., print its contents out)
     For each vertex v directly reachable from c
          If v has not yet been "discovered"
             Mark v as "discovered"
             Insert vertex v into the queue
}
```

weighted edges

1. edge has a cost associated with it
2. shortest path has lowest total cost of edges between two vertices
3. finding shortest path between two nodes
    a. dijkstra's algorithm

dijkstra's algorithm

1. basic idea
    a. splits vertices in two distinct sets: set of unsettled vertices and set of settled vertices
    b. unsettled vertices
        i. don't know optimal distance to it from starting vertex s
    c. settled vertex
        i. we havel earned starting vertex s