Data Structures and Algorithms

CS2800

4 Credit Course

Introduction

- Informally, an algorithm is any well defined computational procedure that takes some value or a set of values as input and produces some value or set of values as output. An algorithm is thus a sequence of computational steps that transform the input into output.
- Efficiency: Different algorithms devised to solve the same problem often differ dramatically. Efficiency can be taken as the time taken for the algorithm.
  - Time for a program = (No. of instructions)/(No. of instructions executed per second)
- Pseudo code conventions
- Insertion sort :
  - Input: A sequence of n nos. <a1, a2, ……. , an>
  - Output : A permutation of the input sequence <a11,a21, ……. , an1> such that

    a11 <= a21 <= ……. <=an1.

  - Pseudo code :
  - for j = 2 to A.length

    Key = A[j]

    //insert A[j] into the sorted sequence A[1 …….. j-1]

    i = j-1

    while i>0 and A[i]>key

    A[i+1] = A[i]

    i = i-1

    A[i+1] = key

- Loop invariants and correctness of insertion sort.
  - Initialization
  - Maintenance
  - Termination
- Analysis of algorithms
  - Worst case – The worst case running time of an algorithm gives us an upper bound on the running time of the input.

        - Average case – The average case is often as bad as the worst case. In some particular cases the technique of probabilistic analysis is applied to find the average case running time.
- Divide and Conquer approach
    - Divide
    - Conquer
    - Combine
- Merge Sort
    - cnlogn steps
    - Analysis of merge sort
- Asymptotic Notation
    - Θ notation : Θ(g(n)) = { f(n) : there exists positive constants c1, c2, and n0 such that 0 <= c1g(n) <= f(n) <= c2g(n) for all n>=n0 }
    - O notation : O(g(n)) = { f(n) : there exists positive constants c and n0 such that 0 <= f(n) <= cg(n) for all n>=n0 }
    - Ω notation : Ω(g(n)) = { f(n) : there exists positive constants c and n0 such that 0 <= cg(n) <= f(n) for all n >= n0 }
    - o notation : o(g(n)) = { f(n) : for any positive constant c > 0, there exists a constant n0 > 0 such that 0 <= f(n) < cg(n) for all n > n0 }
    - ω notation : ω (g(n)) = { f(n) : for any positive constant c > 0, there exists a constant n0 > 0 such that 0 <= cg(n) < cg(n) for all n > n0 }
- Comparing functions
    - Transitivity
    - Reflexivity
    - Symmetry
    - Transpose symmetry
    - Trichotomy
- Standard notations and common functions: Standard mathematical functions and notations.
    - Monotonicity
    - Floors and ceilings
    - Modular arithmetic
    - Polynomials
    - Exponentials
    - Logarithms
    - Factorials
    - Functional Iteration
    - Fibonacci numbers
- Divide and Conquer
    - Divide the problem into a number of sub-problems that are smaller instances of the same problem.
    - Conquer the sub-problems by solving them recursively. If the sub-problem sizes are small enough, however, just solve the sub-problems in a straight forward manner.
    - Combine the solutions to the sub-problems into the solution for the original problem.

- Recurrences: A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. Three methods to solve recurrences – that is for obtaining asymptotic "Θ" or " O" bounds on the solution are :
  - Substitution method – Guess the bound and then use mathematical induction to prove the guess correct
  - Recursion-tree method – Convert the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion.
  - Master method – Provides bounds for recurrences of the form

    T(n) = aT(n/b) + f(n), where a>=1, b>=1 and f(n) is a given function.

- Master Theorem

$$T(n) = O(n^d), \quad d > log_{b^a}$$

$$= O(n^d \, log_{b^n}), \quad d = log_{b^a}$$

$$= O(n^{log_{b^a}}), \quad d = log_{b^a}$$

- Basic proof of Master Theorem
- Divide and conquer method used in multiplication of two complex numbers
- Indicator random variables
  - I[A] = 1, if A occurs

    0, if A does not occur

  - This is used to find average case running time of many algorithms.
  - Expectation of running time is found using probabilistic methods.

Sorting and Order Statistics

- Algorithms that solve sorting problem.
- Decision tree model.
- HeapSort
  - Heaps – The binary heap data structure is an array object that we can view as a nearly complete binary tree. Each node of the tree corresponds to an element of the array. An array A that represents a heap is an object with 2 attributes: A.length and A.heap-size. The root of the tree is A[1].
  - Parent (i) – floor(i/2)
  - Left (i) – 2i
  - Right (i) – 2i + 1
  - Max-heap property – For every node I other than the root, A[Parent(i)] >= A[i]
  - Max-heapify – procedure which runs in O(lg n) time, is the key to maintaining the max-heap property.
  - Build-max-heap – procedure, which runs in linear time, produces a max-heap from an unordered input array.

- Heap sort – procedure which runs in O(n lg n) time, sorts an array in place.
- Priority queues – A priority queue is a data structure for maintaining a set S of elements, each with an associated value called key.
- A max-priority queue has the following functions
  - Insert(S,x) – inserts element x into the set S
  - Maximum(S) – returns element of S with largest key.
  - Extract-Max(S) – removes and returns the element of S with largest key.
  - Increase-Key(S,x,k) – increases the value of element x's key to k.

o Quicksort
- Quicksort applies the divide and conquer paradigm. There is a 3 step divide-and-conquer process for sorting a subarray A[p …….. r]
  - Divide: Partition the array A[p ……. R] into two subarrays A[p …….. q-1] and A[q+1 ……. R] such that each element of A[p …….. q-1] is less than or equal to A[q] which in turn is less than or equal to each element of A[q+1 ……. R].
  - Conquer: Sort the two subarrays A[q+1 ……. R] and A[p …….. q-1] by recursive calls to quicksort.
  - Combine: Because the subarrays are already sorted, no work is needed to combine them.
- QUICKSORT(A,p,r)
  - If p < r
    - Q = PARTITION(A,p,r)
    - QUICKSORT(A,p,q-1)
    - OUICKSORT(A,q+1,r)
- Partition procedure rearranges the subarray A[p ……. R] in place.
- Performance of quicksort :
  - Worst case partition: T(n) = T(n-1) + T(0) + Θ(n). Hence T(n) = Θ(n^2)
  - Best case partition: T(n) = 2T(n/2) + Θ(n). Hence T(n) = Θ(n lg n)
  - Balanced partition
- Randomized version of quicksort.
- Analysis of quicksort – Expected running time.

o Counting sort
- Counting sort assumes that each of the n input elements is an integer in the range 0 to k for some integer k. When k = O(n), the sort runs in Θ(n) time.

o Selection in expected linear time
- The algorithm Randomized-Select is modeled after quicksort algorithm. Unlike quicksort, Randomized-Select works only on 1 side of the partition. Running time is Θ(n).
- Selection in worst case linear time – analysis using medians

DataStructures

- Stacks
  - In stack, the element deleted from the set is the one most recently inserted : the stack implements a last-in first-out or LIFO policy.
  - Insert – The insert operation on a stack is called PUSH. O(1) time.
  - Delete – The delete operation on stack is called POP. O(1) time.
  - When S.top = 0, the stack contains no elements and is empty.
- Queue
  - In queue, the element deleted is always the one that has been in the set for longest time: the queue implements a first-in-first-out or FIFO policy.
  - Insert – Insert operation in queue is called ENQUEUE. O(1) time.
  - Delete - Delete operation in queue is called DEQUEUE. O(1) time.
- Linked Lists
  - A linked list is a datastructure in which the objects are arranged in a linear order.
  - There are two types of linked lists :
    - Singly linked list: Each object has a 'key' and a pointer to the next object.
    - Doubly linked list: Each object has a 'key' and two pointers 'next' and 'previous'.
  - Functions implemented on linked list :
    - Inset
    - Delete
    - Search
- Hash Tables
  - A hash table is an effective data structure for implementing dictionaries. Under reasonable assumptions, the average time to search for an element in a hash table is O(1).
  - Direct-address tables: This is a simple technique that works well when the universe U = {0,1, …….. m-1 } of keys is reasonably small. To represent the dynamic set, we use an array, or direct-address table, denoted by T[0 …… m-1], in which each portion or slot corresponds to a key in the universe U. SEARCH, INSERT and DELETE take O(1) time.
  - Hash tables: When the set K of keys stored in the dictionary is much smaller than the universe U, a hash table requires much less storage than a direct-address table. With hashing, an element with key k is stored in a slot h(k); that is we use a hash function h to compute the slot.
    - Two keys may hash to the same slot. This is called collision. Methods used to resolve collision are :
      - Chaining
      - Open addressing
    - Chaining: We place all the elements that hash to the same slot into the same linked list. Worst case running time for insert is O(1). For search, the worst case running time is the length of the list.
      - Theorem: In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time

Θ(1+α), under the assumption of simple hashing where α is n/m (n slots and m elements).

- Theorem: In a hash table in which collisions are resolved by chaining, an successful search takes average-case time Θ(1+α), under the assumption of simple hashing.

- Hash functions :
  - A good hash function satisfies the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to.
  - Division method – we map a key k into one of the m slots by taking the remainder of k divided by m. h(k) = k mod m.
  - Multiplication method – First we multiply the key k by a constant A in the range 0 < A < 1 and extract the fractional part of kA. Then we multiply this value by, and take the floor of the result.
- Open addressing :
  - In open addressing, all elements occupy the hash table itself. When searching for an element, we systematically examine table slots until either we find the desired element or we have ascertained that the element is not in the table.
    - Linear probing
    - Quadratic probing
    - Double hashing
  - Theorem: Given an open-address hash table with load factor α = n/m <1, the expected number of probes in an unsuccessful search is at most 1/(1 – α), assuming uniform hashing.

- Binary Search Trees
  - The search tree data structure supports many dynamic-set operations, including – search, minimum, maximum, predecessor, successor, insert and delete. Basic operations on a binary search tree take time proportional to the height of the tree.
  - A binary search tree is organized in a binary tree. We can represent the tree by a linked data structure in which each node is an object. In addition to key and satellite data, each node contains attributes 'left', 'right', and 'parent'. The keys in a binary search tree are always stored in such a way as to satisfy the binary-search-tree property :
    - Let x be a node in a binary search tree. If y is a node in the left sub-tree of x , then y.key <= x.key. If y is a node in the right sub-tree of x, then y.key >= x.key.
  - In-order walk: This algorithm prints the key of the root of the sub-tree between printing the values in its left sub-tree and printing those in its right sub-tree.
  - Pre-order walk : Prints the root before the values in either sub-tree.
  - Post-order walk: Prints the root after the values in its sub-trees.
  - Search: Takes O(h) time.
  - Minimum and maximum both refer to the left most node and the right most node of the tree. Both run in time O(h).
  - Predecessor and successor
  - Insert

- Transplant: replaces one sub-tree as the child of its parent with another sub-tree.
- Delete
- Randomly built binary search trees :
    - Theorem: The expected height of a randomly built binary search tree on n distinct keys is O(lg n).

Advanced Design and Analysis Techniques

- Dynamic Programming
    - Dynamic programming like the divide and conquer method solves problems by combining the solutions to sub-problems. We apply dynamic programming to optimization problems.
    - When developing a dynamic programming algorithm, we follow sequence of 4 steps :
        - Characterize the structure of an optimal solution
        - Recursively define the value of an optimal solution
        - Compute the value of an optimal solution, typically in a bottom-up fashion
        - Construct an optimal solution from computed information
    - Rod cutting problem
        - Top down method
        - Bottom up method
        - Memoized method
    - Sub problem graphs: When we think about a dynamic programming problem, we should understand the set of sub problems involved and how they depend on one another. The sub problem graph embodies exactly this information.
    - Matrix-chain multiplication
    - Longest common subsequence problem
- Greedy Algorithms
    - A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
    - Activity selection problem
    - Theorem: Consider any non empty subproblem S_k and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k .
    - Sequence of steps to design greedy algorithm
        - Case the optimization problem as one in which we make a choice and are left with one more sub problem to solve.
        - Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
        - Demonstrate optimal sub structure by showing that, having made the greedy choice, what remains is a sub problem with the property that if we combine an optimal solution to the sub problem with the greedy

choice we have made, we arrive at an optimal solution to the original problem.
        - Elements of the greedy strategy
            - Greedy choice property
            - Optimal substructure
    - Amortized analysis
        - Aggregate analysis
        - Accounting method

Advanced DataStructures

- Disjoint sets
    - Linked list representation of disjoint sets
        - Implementation of union
        - Weighted union heuristic
    - Disjoint set forests
    - Heuristics to improve running time
        - Union by rank
        - Path compression

Graph Algorithms

- Representation of graphs
    - Adjacency list representation
    - Adjacency matrix representation
- Breadth First Search
- Depth First Search
- Topological Sort
- Minimum Spanning Tree
    - Kruskal's Algorithm
    - Prim's Algorithm
- Single Source Shortest Paths
    - Negative weight edges and cycles
    - Bellman-Ford Algorithm
    - Dijkstra's Algorithm