Data Structures (IT205)
Lab Manual
Core Course, B.tech.
Semester III (2^{nd} Year)

Prepared by

Rahul Muthu

Course Instructor (IT205)

[2011-12, 2012-13, 2013-14, 2014-15, 2015-16]

mgc.jpg

July 2015

Contents

Part I

Basic lab exercises

This part of this manual consists of such programming exercises as implementing the most standard routines associated with data structures taught in the course. The questions are by and large direct implementations and the goal of this is to ensure the student is able to translate into a real working program the data structures and associated algorithms taught in an abstract or mathematical manner. It is expected that each student has thoroughness as far as the programming involved in this part is concerned. The lab exercises here are classified according to topics covered in the course. The occurrence of a question in a particular assignment as well as the timing of the assignment varies as it needs to be synchronised with the material covered in the lectures. The purpose of this manual is to provide a fairly good idea of all the programming problems a student is expected to be conversant with and is not necessarily the exact order in which assignments are given. This does not follow a precise timeline.

Chapter 1

Arrays, Linked Lists, Pointer Based Data Structures, Stacks & Queues

1. Write code for creating a linked list, adding elements to the linked list at the tail, searching for an element with a particular key and also deleting an element with a particular key if it exists.

2. Implement stacks and queues using the linked list data-structure.

3. Implement stacks and queues using arrays.

Chapter 2

Sorting & Searching

1. Implement Binary Search (recursive and iterative).
2. Implement bubble sort, selection sort, insertion sort, merge sort, quick sort and heap sort routines.
3. Write code to find the rank of an element in a list of elements from a totally ordered set.
4. Write code to find an element of a given rank in a list of elements from a totally ordered set.

Chapter 3

Heaps

1. Write a routine for recovering the parent, left-child and right child of a node in a binary heap.
2. Implement the root-down and the leaf-upward variants of the build heap routing.
3. Implement Heap Sort (including the necessary subroutines)
4. Implement the various routines of the binary heap data structure for priority queues (Insert, delets, key value increase/decrease and restoring the heap property).

Chapter 4

Binary Search Trees

1. Write routines (recursive and iterative) to implement in-order, pre-order and post-order tree traversals. Also implement level-order traversals, and zig-zag traversals.
2. Write procedures to compute the height of a node, depth of a node and size of a subtree dynamically.
3. Write code for the query operations (find/search, maximum, minimum, successor and predecessor in a binary search tree).
4. Write code for inserting nodes into and deleting nodes from binary search trees.

Chapter 5

Balanced BSTs

1. Write code for the Left- and Right- Rotate operations which take as arguments a Binary Search Tree (used to locate the ROOT variable) and a node x within it.
2. Write routines for balancing binary search trees using rotations. Do this at every insertion and deletion operation that results in an imbalanced tree. This is an AVL tree.
3. Write a program to find a node (if any) in a Binary Search Tree, such that the maximum depth of its left and right subtrees differ by more than 1, and perform a rotate to reduce the difference.

Chapter 6

Graph Algorithms

1. Implement Breadth First Search (BFS) for undirected graphs.
2. Implement Depth First Search (DFS) for undirected graphs.
3. Implement Dijkstra's algorithm for a positive edge weighted graph.
4. Implement Kruskal's and Prim's algorithms for finding MSTs for an undirected graph.
5. Implement the all pairs shortest paths algorithm based on matrix multiplication ideas and Floyd Warshal Algorithm.

Chapter 7

Hashing

1. Implement direct addressing in a table.
2. Implement hashing with chaining.
3. Implement hashing by probing using different schemes. Take care of deleted slots and differentiate them from slots never occupied.

Part II

Advanced lab exercises

This part of this manual consists of advanced programming exercises which may require a certain degree of creativity and adaptation of standard data structures in appropriate manners. The questions in this section are primarily drawn from the lab exams I have conducted for this course over the last four years, and also variants of questions which I set as a part of my theory exams. The questions in this part are by and large more difficult than those appearing in the first part, though the classification on this basis is not unwavering.

Chapter 1

Arrays, Linked Lists, Pointer Based Data Structures, Stacks & Queues

1. Write a code for implementing a stack, a queue, and a queue simulated by two stacks and a stack simulated by two queues, all using prespec- ified size arrays. You need to implement the push, pop, enqueue, dequeue operations and error checks.
2. Write code for reversing a singly linked list.
3. (a) Implement routines for exchanging two elements of a stack. You may use two extra stacks for temporary storage.
   (b) Implement a routine for reversing a contiguous subsequence of a stack of elements. Again you may use two auxilliary stacks for temporary storage.
   (c) Write a routine for exchanging two elements of a queue using two extra queues for temporary storage.
   (d) Write routine for reversing a subsequence of contiguous elements of a queue, using two auxilliary queues.
4. Write code to translate an arbitrary rooted tree into a binary tree, using the left-child, right-sibling representation. Also write a routine to decode it and recover the original tree.
5. Write a routine for returning all elements with a particular key value, without deleting them from the list. The list of elements needs to be returned in the form of a duplicate linked list.
6. Write an efficient routine for reversing a sublist of elements represented in a linked list.
7. Write an efficient routine for reversing a subsequence of a linked list.
8. Consider a list of records each of which have 5 different key fields from a totally ordered set. The search condition for a record is based on specific values of each of the five key fields. Create a linked list of the largest set of data items all of which match at least three specific key values. This needs to be done preserving the order of elements in the original list and also without destroying the original list.
9. (a) Write a routine for changing a sequence of elements in a queue according to a specified input permutation using two temporary queues. Do the same using a single temporary queue.

   (b) Write a routine for changing a sequence of elements in a stack according to a specified permutation using two temporary stacks.

10. Implement a list data structure which behaves as a queue or a stack (conceptually meaning FIFO or LIFO) depending on the size of the data structure. For concreteness implement it such that deletion happens according to the logic of a queue if the size of the data structure at the time of deletion is ≤ 8 and according to the logic of stack when the size is > 8 at the time of deletion.

11. Write a routine for transferring the elements of a stack to a second stack in the same order, using a third stack as working space. The extra restriction is that the relative order of no two elements can be reverse of their original on any stack at any stage of the procedure.

Chapter 2

Sorting & Searching

1. Consider a circular array A of n elements which are in increasing order beginning at some position. However it is not known that the beginning position is index 1, and the minimum element could be at any position from 1 to n. The elements are in increasing order starting at that index and traversing the array in a circular fashion. Using ideas similar to binary search find the location of the maximum/minimum element in time $\theta(\log n)$. Note this is better than the $\theta(n)$ time needed in the case of an unsorted array and worse than the $\theta(1)$ time needed in a conventional sorted array.

   Example: $50, 58, 79, 80, 84, 100, 105, 256, 7, 14, 28, 42, 48$. Here the returned value should be the positions 8 and 9 respectively containing 256 (maximum) and 7 (minimum).

2. Consider a two dimensional $n \times n$ array A containing the distinct integers $\{1, \ldots, n^2\}$. The arrangement is such that $A[i, j] < A[i+1, j]$ and $A[i, j] < A[i, j+1]$, for all $1 <= i, j <= n$. Write a code to find the possible locations of the element of a given rank as well as the rank of an element at a given location.

   Example:

1	3	12	15	21	30	33	34
1	3	12	15	21	30	33	34
1	3	12	15	21	30	33	34
1	3	12	15	21	30	33	34
1	3	12	15	21	30	33	34
1	3	12	15	21	30	33	34
1	3	12	15	21	30	33	34
1	3	12	15	21	30	33	34

3. Write a routine for finding a maximum/minimum element in an unsorted array without resorting to sorting.

4. Write a routine for simultaneously finding the maximum and minimum elements in an unsorted array without sorting, and more effeciently than finding the two of them individually by the method of the previous question.

5. Write a routine that finds the second largest/ second smallest element in an unsorted array without sorting. This should be significantly more efficient than finding the maximum (minimum) and then finding the maximum (minimum) among the remaining elements.

6. (a) Now consider a set of n numbers stored as a sequence in an array A, which needs to be sorted. One possibility is to sort the first f fraction of the sequence, and then sort the last f fraction of the sequence and then to sort the first f fraction again, in each case using the algorithm you

stated above as a subroutine to sort the selected fractional subarray. What is the smallest value of f for which this procedure will correctly sort any input sequence? What is the running time of this modified sorting procedure in terms of f and the running time expression you wrote earlier (for the subroutine)? Solve it and express your answer in closed form asymptotic notation.

(b) Consider an extended version of the above procedure, where you repeat the alternating procedures of sorting the first and last f fraction of the array until the sequence is sorted. What is the smallest value of f, for which this procedure will eventually sort the sequence. Compute the number of iterations required by your algorithm as a function of f.

(c) Is the running time of these sorting algorithms smaller or larger, asymptotically, than the standard algorithms? If it is larger, then suggest some reason why this algorithm might be useful.

7. (a) Now consider a 2-dimensional $n \times n$ array. In each row the ele- ments are in increasing order, and in each column also the elements are in increasing order. Develop an algorithm similar to binary search, and argue that it is correct. Write an expression for its running time as a recurrence relation and solve the recurrence to get an estimate on the time required for this search algorithm.

   (b) Generalise this idea to a k-dimensional array, with size n on each dimension.

   (c) How long does it take given a set of n numbers, in arbitrary (unsorted) order to place in these data-structures (2 or higher dimensional arrays) satisfying the requirements of relative values? Is this better or worse than sorting the elements into a long single dimensional array, asymptotically and in absolute terms?

8. (a) Write a program to implement counting sort.

   (b) Write code similar to counting sort to arrange n elements from a fixed set $\{1, \ldots, k\}$ according to the following description. If the frequencies of the elements are $\{f_1, \ldots, f_k\}$, then let f_{min} be the minimum non-zero frequency. The initial $k_0 f_k$ elements of the output array must contain f_{min} occurrences of 1 (assuming the frequency of 1 is not zero), followed by f_{min} occurrences of 2 (again, assuming the frequency of 2 is not zero) and so on upto f_{min} occurrences of element k. Here k_0 is the number of elements among $\{1, \ldots, k\}$ which have non-zero frequency. For elements of frequency zero, there is nothing to place in the output array. After this, reduce the frequency f_{min} from the original frequency of each of the k elements and repeat the procedure using the new value of f_{min} and the new set of non-zero frequency elements to continue placing them in the next subsequent positions of the output array till all the elements are placed. The algorithm must be stable for equal value elements.

9. (a) Write code for radix sort.

   (b) Write a variant of radix sort where each digit is from the elements $\{1, \ldots, k\}$ but for the i^{th} digit, the increasing order of elements is beginning from element value i using cyclic/modulo arithematic. Thus for the right-most position the minimum element is 1 as usual. The order is $1, \ldots, k$. For the second position we have $2 < 3 < \cdots < k < 1$ for the third position we have $3 < \cdots < k < 1 < 2$. Can the idea used to translate counting sort to radix sort be adapted to this routine?

10. All sorting algorithms work by reducing the number of inversions until there are none. Bubble sort is a special sorting algorithm in which every data movement reduces the number of inversions by exactly one. Given an input of n distinct numbers and a number k between 0 and n(n-1)/2, modify bubble sort to stop with exactly k inversions. If the original number of inversions is less than k, then that number should be increased to k. The number of inversions should change monotonically during the course of the algorithm (either keeps increasing or keeps decreasing; both not allowed). The algorithm should resemble bubble sort in terms of data movement, in the sense that you cannot exchange data except

adjacent entries. You may compare non-adjacent elements, but exchange can happen only between adjacent elements.

11. Implement Quick-Sort and also output the number of comparisons (between input data) and the number of exchanges (of input data) your code performs during the execution.

Chapter 3

Heaps

1. Write code for finding the parent and child indices for a k-ary heap.
2. Write a program to build a treap from a given set of ordered pair values.
3. Write a program to buils an alternating heap of n elements.
4. Write a program to compute the number of valid arrangements of an array of n distinct elements into a maximum heap.
5. A continuous stream of jobs arrive and are enlisted for processing on a machine. Each of the jobs has a priority number associated with it. After finishing a job, the machine selects the earliest arriving job among the pending jobs of highest priority. Write a program making use of two queues and one temporary variable, to enforce this protocol of job-scheduling. The temporary variable can be used to store the priority of the least priority job waiting to be executed.
6. Suppose you are given an array of n distinct numbers A which may structurally be treated as a heap. It is possible that the key value property is not satisfied for max heaps and is also not satisfied for min heaps. Without altering key values or moving/exchanging data, find the largest subtree of the given tree which is also a heap (either max or min whichever provides a heap with the maximum number of nodes). In this question you may take a modified meaning of subtree where you break the link of the root from its parent and are also free to ignore some descendants as long as the selected subset of nodes form a heap structurally.
7. Given a binary min heap on n nodes all containing distinct key values, write a program to compute the number of pairs of nodes such that exchanging the key values of these nodes will not affect the binary min heap key value property.
8. A binary minimum heap is stored in an array as you are aware. An inversion in an array of distinct elements is the number of pairs of elements which are stored in the array with the larger element appearing before the smaller element. Given a positive integer n what is the smallest and largest possible number of inversions in a binary minimum heap on n distinct elements. Write an algorithm and program to compute this value.

Chapter 4

Binary Search Trees

1. Write procedures to compute the height of a node, depth of a node and size of a subtree and rank of an element, dynamically. Use these subroutines to write code to determine if there is a subtree consisting of precisely the nodes with ranks from $\{i, \ldots, j\}$.

   A routine to find the range of ranks in a subtree was discussed in the last lecture. Essentially the terms local rank, base rank and overall rank were introduced. This is one possible way to solve this.

2. Given a BST (root and pointers to parent and children), compute the number of inversions if you print the keys of the BST according to the level order traversal. You may assume the keys are distinct. Your code should work on the basis of an algorithm that does not need to read explicit key values, but infer which node has a smaller or larger key value, using their relative positions in the BST.

   An inversion in a set of distinct key values is when a smaller element appears later than a larger element.

   Solution idea:
   Here, you may compute all subtree sizes using the standard recursive algorithm, beginning from the root. Modify this routine so as to store subtree size -1, instead of subtree size, at each node.

   Now perform level order traversal. At each node compute its number of inversions with nodes appearing after it. This can be done by adding up the subtree size-1 values of all preceeding nodes on the same level and then adding the size of the left subtree of the current node. (note you should add 1 to subtree size minus one to get subtree size! :P). Thus, a tempoorary variable can be created which is the cumulative values of subtree size minus 1 of all nodes in the current level. This must be reset to zero when we change level. This can be detected by assigning a level number attribute to the nodes (as is done while enqueuing for level-order-traversal similar to breadth first search). When one dequeues an element, and enqueues its children, the children are assigned level number of the dequeued node plus 1.

   This algorithm runs in $\theta(n)$ time.

3. Consider a binary search tree on n nodes which contain the elements $1, \ldots, n$. Write a code to locate a pair of links, such that the number of nodes in the three fragments resulting from breaking these links are of approximately equal size.

   Solution idea:
   Here compute the subtree sizes of all nodes using the standard recursive procedure beginning at the root. Cut off the link of a node with subtree size approximately $\frac{n}{3}$ with its parent. Thus the residual fragment is approximately $\frac{2n}{3}$ in size. Now locate a link to a node such that the subtree size is approximately $\frac{n}{3}$. All this effectively uses the subtree sizes information computed earlier. The key idea is to find a subtree whose size is as close to $\frac{n}{3}$ as possible.

4. Write a program to compute the number of structurally different binary trees of height h, for which the level order traversal and zigzag traversal result in the same sequence of nodes.

   Solution idea:
   Here, the height is fixed as h and hence the number of levels is exactly $h+1$. The level 0 (root level) clearly has only one node. Now the next level (level 1) also has only one node since all odd level numbers can have only one node if level-order coincides with zig-zag order because at odd levels, level order moves from left to right and zig-zag moves from right to left. Thus if there is more than one node at such levels, the orders will be different.

   Since each odd level has exactly one node, the next even level can have either one node or two nodes (since the tree is binary).

   Thus the number of even levels (apart from the root) is $\lfloor \frac{h}{2} \rfloor$. At each such level we hve a choice of incorporating either one node or two nodes. This choice made at each level dictates the number of configurations of placing the unique node at the next level.

   Hence the computation can be described as follows.

   Select the subset of even levels which will have two nodes. (the remaining have one node each). The possibilities for doing this are

   $$\binom{\lfloor \frac{h}{2} \rfloor}{i}$$

   Here i lies between 0 and $\lfloor \frac{h}{2} \rfloor$. For each such level, the ways of appending the node at the next odd level is 4 (provided such a next level exists. It wont exist for the last even level if it is also the last level overall). For the remaining even levels, the unique node of that level can be appended as child to its parent in two ways and its unique child in the next level (if one exists) can be appended in 2 ways independent of the parent link. Thus here as well there are four ways to complete the linking. There are thus approximately

   $$4^{\lfloor \frac{h}{2} \rfloor}$$

   ways to complete the links in any configuration. This number is precise when the last level is odd. When the number of levels is even, the last level there is only a 2 factor or a 1 factor depending on whether the level has one node or two nodes respectively and not a 4 factor. This is because there is a parent link but no children link.

   We have to sum this number over all the

   $$2^{\lfloor \frac{h}{2} \rfloor}$$

   choices of subset of even levels having two nodes.

5. Write a program to compute the number of orderings of n distinct keys which will result in a particular n node binary search tree. You may assume that you are given as input the tree structure including the root and parent, left-child and right-child pointers at each node.

   Solution idea:
   Here, one can translate the given binary tree into a set of distinct binary strings labelling the path of each node from the root with 0 for left child and 1 for right child. This set of strings is closed under the prefix closure operation.

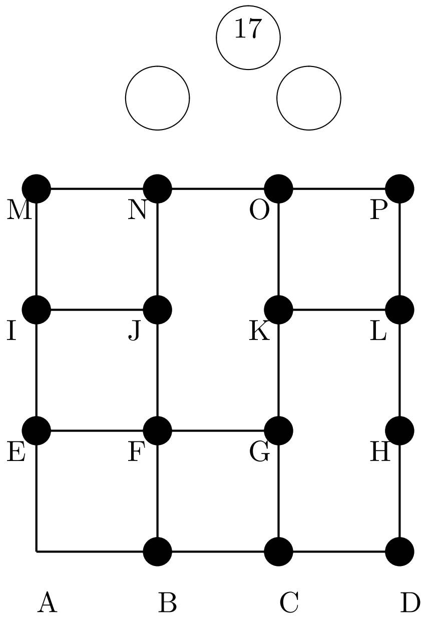
   The root is labelled with the zero length string ϵ. The number of nodes in the left subtree is equal to the number of strings beginning with 0. The number of nodes in the right subtree is the number of strings beginning with 1. These nodes can be interleaved in any order across the subtrees as long as they respect the order restrictions among the ones in the same subtree. The number of interleavings is given as

$$\binom{n(T_L) + n(T_R)}{n(T_L)}$$

.

Now solve the problem of orderings of T_L and T_R recursively and multiply those individual answers to eachother and the above term to get the final answer. To translate the subproblems of the left and right subtrees into the string parlance, simply take all the strings beginning with 0 (left subtree nodes) and exclude the initial 0 and do the analogous thing for the right subtree. Thus we have a recursive approach. When we have a tree with one node (empty string ϵ) the number of insert orders is just 1. This is the bottom of the recursion tree.

6. Given a BST on n nodes with distinct key values, find the largest value k, such that the nodes with the k smallest key values form a subtree and the nodes with the k largest values also form a subtree. $k < n$, i.e. we are excluding from consideration the entire tree and only considering nontrivial subtrees. If no such value of k occurs, then return not possible.



7. Suppose you have a modified definition of Binary Search Trees, which require that all nodes with key value less than the key value of the root must go into the left subtree and all nodes with key value greater than the key value of the root go into the right subtree (as in the normal BST). However, the rule gets flipped at level 1 (children of the root), meaning the nodes of the left subtree with key less than the key of the left child of the root go into its right subtree and those with key values greater go into its left subtree. Same rule applies at the right child of the root. At level 2, the rule is again back to that at level 0 (i.e same as for root). The rule alternates at every level. Assuming only distinct keys are present:

   (a) Write a routine for inserting a node into such a tree satisfying this modified search tree condition.

   (b) Write routines for searching for a key in this tree and also for finding the successor of an element and the maximum element in the tree.

Chapter 5

Balanced BSTs

1. Write code for left-rotation of a BST at a node, and also compute the change in average node-depth as a result of this rotation.

2. Consider a binary search tree given as input with only the structure and the key values satisfying the BST property are not given explicitly. However, as we know rotations preserve the key value property. A cost associated with each node is provided as an input and this cost field is not the key value field satisfying the BST PROPERTY. Write code to perform a series of rotations such that the sum of the products of the cost of each node with its depth (after the series of rotations) is minimised.

3. Implement the Red-Black Tree Insert and Delete Operations, including the restoring of the red-black tree properties after inserting or deleting of a node. In particular, you need to have a single nil variable to store all the leaves and the parent of the root. The root is always black. In addition to the fields you used to program the standard binary search tree, you need to have a field for the colour of each node.

4. Implement insertion and deletion in Red-Black trees and 2-3-4 trees.

Chapter 6

Graph Algorithms

1. Write a program to find a cut vertex in a graph if any.
2. Write a program to determine the connected components of an undirected graph.
3. Adapt BFS and DFS on directed graphs and undirected graphs to classify the edges as tree edges, cross edges, forward edges and back edges.

Chapter 7

Hashing

1. Implement dynamic tables with the expand corresponding to full table with a doubling in size for a new table, and the threshold for reducing to a table of half the size is $\frac{1}{3}$ occupancy.

Chapter 8

Cross topics

1. You are given a sequence of stacks $S_1, \ldots, S_t$, containing $n_1, \ldots, n_t$ elements. The elements in all the stacks are numbers from the set $\{0, \ldots, 9\}$. The bottom most element of a stack is not allowed to be 0. The i^{th} stack can thus be thought of as decimal integer of ni digits where the most significant digit is at the bottom of the stack and the least significant digit is at the top. You need to sort the stacks in numerical order. You may use two extra stacks of sufficient capacity (ignore overflow error) to decide if a particular stack is less than another stack (according to the previous definition) and then write the output in an array A of length t. This array will indicate the permutation of the input stacks according to sorted order.

   For example Stack 5 is the minimum, Stack 1 is the second minimum, Stack t is the third minimum etc. Write an efficient algorithm for this and analyse its running time in terms of the input sizes given. Notice that you can use any standard optimal algorithm like merge sort or heap sort. Just focus on the subroutine for comparing two stacks (whose running time is now a function of n_i and n_j , and not $O(1)$). For the overall algorithm find the total cost according to the underlying sorting algorithm you select. At the end of the algorithm the stacks must contain the original elements in the same order and only the stack index permutation must be written into array A.

2. The students may look at questions set in theory exams over the years (located in the past exam papers folder within this course's lecture folder) for further ideas of implementable data structures to hone their skills.