|  |
| --- |
|  |
| NOTE: Detailed algorithm or C code is acceptable for answers. State any assumptions made. Calculator needed | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | a) | Write a recursive method to determine whether a singly linked list is sorted in descending order or not. Assume Node contains data and next pointer. Declare Node structure and write the method. The method should return 1 if list is sorted and 0 otherwise.  int isLinkListSorted (NODE \*linklist) // write this recursive method and declare NODE  typedef struct node { int data; struct node \*next; } NODE;  int isLinkListSorted (NODE \*linklist) {  If (linklist == null) or (linklist  next == null) return 1;  if (linklist  data >= linklistnextdata // in order  return (isLinkListSorted (linklistnext)  else  return 0; | 2  2  2 | |
| b) | Given an integer doubly linked list, with Head and Tail pointers, print the list so that alternate elements from the beginning and end of the list are printed. 12345 should print 15243 and 123456 should print 162534. Your method should handle border conditions.  void AlternatePrint(DNODE \*Head,, DNODE \*Tail); // write this method  typedef struct dnode { int data; struct dnode \*prev; struct dnode \*next} DNODE;  void AlternatePrint(DNODE \*Head,, DNODE \*Tail) {  DNODE \*p = Head;  DNODE \*q = Tail;  if (p == null) return; // empty list  while (pnext != q) || (p != q) {  print (pdata, qdata);  p = pnext;  q = qprev;  }  print (pdata);  if (p != q)  print(qdata);  } | 1  1  2  2 | |
|  | c) | X and Y are two singly linked lists containing integer document ids. Write a function (or algorithm) called “Merge” that takes X and Y and returns a third list Z which contains only doc ids that are in X and NOT in Y. Eg. X = {1,3,5,6,7} and Y = {1,2,3,4,6} then Z should be {5,7}  MERGE(x, y) {  z = ( ) // null list  while( x !=NULL and y !=NULL)  if (xdata == y>data)  then x = xnext  y = ynext  else if (x>data< ydata)  then  insert xdata to end of z list  x = xnext  else  y= ynext  }  if (x != NULL)  while (x!= NULL) {  insert xdata to end of z list  x = xnext  }  return(z)  } | 2  2  2  2 | |
|  | | | | |
| 2. | a) | Convert the following infix expression to its equivalent postfix and prefix expressions. The symbol (^) denotes the exponent operator.  ( (A+B) /C ) ^ ( D\*( E-F ) )  Postfix : AB + C / DEF - \* ^  Prefix : ^ / + ABC \* D - EF | 3  3 | |
| b) | Write a method in C to print individual digits of a number separated by hyphen. Eg. 17852 should print 1-7-8-5-2. You can use a stack with the standard stack operations (push, pop, top, isEmpty, size). You do not have to write the Stack methods.  void digitizer( int decimal) // should print individual digits of decimal separated by hyphen  void digitizer(int decimal) {  int r;  while(decimal!=0) {  r = decimal%10;  decimal = decimal/10;  push(r);  }  while ( ! isEmpty())  print(pop() + “-“);  } | 4  2 | |
| c) | Implement enqueue and dequeue operations of a circular queue using an array Q of size N. Assume f is front of the queue and r is next available cell in the array.  int size()  return (N-f+r) %N  bool isEmpty()  return (f==r)  void enqueue(int item) {  if (size() == N-1) then  printf ("Q is full")  else  Q[r] = item;  r = (r + 1) %N  }  int dequeue() {  if (isEmpty()) then  printf("Q is Empty")  else {  res = Q[f];  f = (f+1) %N  return (res)  } | 1  3  1  3 | |
|  | | | | |
| 3. | a) | Write a method called AlternateSplit to split an integer Queue into two queues which contain alternate elements of the original queue. Use only queue methods that are usually defined in the queue ADT (enqueuer, dequeue etc) For eg. if original queue is (1,2,3,4,5) then Q1 is (1,3,5) and Q2 is (2,4). You do not have to write enqueue and dequeuer methods.  Void AlternateSplit(Node \*Q, Node \*\*Q1, Node \*\*Q2) {  While (!isEmpty(Q)) {  enqueue (Q1, dequeue(Q));  If (!isEmpty(Q))  enqueue (Q2,dequeue(Q));  }  } | 2  4 | |
| b) | Draw a Binary Search Tree with these numbers.  14, 17, 7, 11, 23, 4, 13, 16  Redraw after 7 is removed from the BST.  14  / \  7 17  / \ / \  4 11 16 23  \  13  After 7 is removed:, one of the following ( 4 or 11 replaces 7) | 3  3 | |
| c) | Suppose that you have a balanced binary search tree data structure declared as follows:  struct tree {  int key;  int value;  struct tree \*left, \*right;  };  Write a function sumTree that, given a pointer to the root of the tree, computes the sum of the  value fields in all nodes in the tree.  int sumTree(struct tree \*root)  int sumTree(const struct tree \*root)  {  int sum;  if(root = NULL) return 0;  sum = root->value;  if (root->left != NULL)  sum += sumTree(root->left);  if (root->right != NULL)  sum += sumTree(root->right);  return sum;  } | 2  4  2 | |
|  | | | | |
| 4. | a) | Given a integer array S[MAX], use the functions of a heap data structure to sort the elements in S in descending order. Declare the HEAP data structure and write the function Sort\_Descend(). You can use the heap functions Insert and RemoveMin, without defining them.  void Insert (HEAP \*h, int i); // assume this function is already defined  int RemoveMin(HEAP \*h); // assume this function is already defined  void Sort\_Descend (int S, int size); // write this function that takes S array and sorts it using the above heap functions  typedef struct heap {  int capacity;  int element[MAX\_HEAP];  } HEAP;  void Sort\_Descend (int S, int size) {  HEAP \*h = malloc (sizeof (HEAP));  int i;  for (i=0; i<MAX; ++i) Insert (h, S[i]);  for (i=MAX-1; i>=0; --i) S[i] = RemoveMin(h);  } | 2  2  2 | |
| b)` | In a min-Heap, the data value of the parent node is smaller than that of its children. Draw a min-heap with the following data values. Redraw the min-heap after removeMin() operation is performed. You do NOT have to show all the in between heap structures.  40,30,70,20,50,60,10,90  10  / \  30 20  / \ / \  40 50 60 70  /  90 | 3  3 | |
| c) | Insert the following elements into an initially empty AVL tree. Show the rotations and all the in between trees. (RVenkatesan p181)  73,11,56,97,88,45,55,61 | 8 | |
|  | | | | |
| 5. | a) | Insert the following letters into what is originally an empty B-tree of order 5:  C N G A H E K Q M F W L T Z D PR X Y S  Order 5 means that a node can have a maximum of 5 children and 4 keys. All nodes other than the root must have a minimum of 2 keys. Assume keys are in alphabetical order A<B<C…<Z | | 8 |
| b) | Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101. Give the hash table contents resulting from Linear Probing.   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 50 | 700 | 76 | 85 | 92 | 73 | 101 | | 1 | 0 | 6 | 1 | 1 | 3 | 3 |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | | 700 | 50 | 85 | 92 | 73 | 101 | 76 | | | 6 |
| c) | Give the contents of the hash table that results when you insert items with the keys  R E P U B L I C A N in that order into an initially empty table of *N* = 5 lists, using separate chaining with unordered lists.  Use the hash function 11\* *k* mod *N* to transform the *k*th letter of the alphabet into a table index, e.g., hash(I) = hash(9) = 99 % 5 = 4.   |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | R | E | P | U | B | L | I | C | A | N | | k | 18 | 5 | 16 | 21 | 2 | 12 | 9 | 3 | 1 | 14 | | 11\*k | 198 | 55 | 176 | 231 | 22 | 132 | 99 | 33 | 11 | 154 | | 11\*mod 5 | 3 | 0 | 1 | 1 | 2 | 2 | 4 | 3 | 1 | 4 |  |  |  | | --- | --- | | Index | Elements | | 0 | E | | 1 | A, U, P | | 2 | L, B | | 3 | C, R | | 4 | N, I |   Order of elements in hash table should not matter. | | 6 |