|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Register Number |  |  |  |  |  |  |  |  |  |

|  |  |
| --- | --- |
| **Sri Sivasubramaniya Nadar College of Engineering, Kalavakkam – 603 110**  (An Autonomous Institution, Affiliated to Anna University, Chennai) | |
| Department of Computer Science and Engineering  **Continuous Assessment Test – I**  **Question Paper** |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Degree & Branch** | BE (CSE) | | | | **Semester** | III |
| **Subject Code & Name** | UCS 1302 Data Structures | | | | **Regulation:** | **2018** |
| **Academic Year** | 2021-2021 ODD | **Batch** | 2020-2024 | **Date** | **18.10.2021** | **FN** |
| **Time: 90 Minutes** | **Answer KEY** | | | | **Maximum: 50 Marks** | |

**Part – A (6×2 = 12 Marks)**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| <KL2> | 1. 1. Given the following functions for the running time of different algorithms:   f1(n) = n log10 n + 1000n  f2(n) = 10n + 20  f3(n) = n(n - 1)/2  Infer whether each function is of the time complexity log n, n, n log n or n2  by writing Yes/No in each cell of the table given below.  **Solution**   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | O(log n) | O(n) | O(n log n) | O(n2) | | f1(n) | No | No | yes | yes | | f2(n) | No | yes | Yes | Yes | | f3(n) | No | No | No | yes | | <CO1> |
| <KL2> | 2. Outline the main advantages of the linked-list based implementation and array-based implementation of the List ADT.  Linked List  The amount of space used is proportional to the amount needed. . Certain operations are quicker. Specifically: Construction and destruction may be quicker when the pointer to the node is given. Insertion and deletion require only pointer jumps and not moves of the data values.  Arrays  The retrieve operation is fast as it uses direct addressing.   |  |  |  | | --- | --- | --- | | Operation  add to front  add to back  add at given index/given node  find index of an object  remove front element  remove back element  remove at given index/node | Array list  n  1  n  n  n  1  n | Linked list  1  n  1  n  1  n  1 | | <CO1> |
| <KL2> | 3. Show a C function that takes an array of integers and creates a new array that contains all the integers from input array with all negative integers at the end of the array.  Example: Input : a[ ] = {2,3,-8.7,-9,6,-4.-2,0}  Output: b[ ] = {2,3,7,6,0,-8,-9,-4,-2}  **Solution:**  void pushNegative(int a[], int size)  {  Int i=0, k=0, result[size];    while(i<size)  {  if(a[i]>0)  result[k++] = a[i];  i++;  }  while(i<size)  {  if(a[i]<0)  res[k++] = a[i];  i++;  }  } | <CO1> |
| <KL2> | 4. Write the algorithm for deleteFirst(\*CLL) that deletes the first node in the circular linked list where CLL is the pointer to the last node.  Algorithm deleteFirst(\*cll)  Input: circular linked list whose last node is pointed by cll  Output: Input list with first node deleted; returns deleted data  DS: node   |  |  | | --- | --- | | Data | next |   Var: node \*temp, int deleted  1. temp ← cll →next  2. (cll → next ) ← (cll → next → next)  3. deleted ← (temp→data)  3. free(temp)  4. return deleted | <CO1> |
| <KL2> | 5. Write the algorithm for inserting an element into the stack implemented with an array  Algorithm push(int stack[], int\* top, int item)  // To push ‘item’ into ‘stack’ of size N in the position \*top+1  1. if (isFull(stack[ ]) <> 1) //(top <= N-1)  then \*top = \*top+1  stack[\*top] = item  2. return | <CO2> |
| <KL2> | 6. Why do you need circular arrays to implement a Queue? Explain with an example. (explanation 1 +eg 1) | <CO2> |

**Part – B (3×6 = 18 Marks)**

|  |  |  |
| --- | --- | --- |
| <KL3> | 7. Build a function in C to print all the *loosers* in the array. An element is a looser if it is lesser than all the elements to its right side. The rightmost element in the array is always looser.  ***Example: Input: a[ ] = {10, 8, 4, 3, 15, 28,23}***  ***Output: loosers are 3, 15 and 23***  **Solution:**  void printLoosers(int arr[], int size)  {      for (int i = 0; i < size; i++)      {          int j;          for (j = i+1; j < size; j++)          {              if (arr[i] >=arr[j])                  break;          }          if (j == size) // the loop didn't break              printf(“%d “, arr[i]);    }  } | <CO1> |
| <KL2> | 8. Define polynomial ADT and outline the algorithm for difference of two polynomials. Make use of a linked list. Trace your algorithm with the following example. Support it with diagrams. (Trace 3, function -3)  Input:  poly1 = -8x6+7x5+4x2  poly2= 3x6-9x2-8  Output:  Poly3 = -11 x6+7x5+13x2-8  **Solution:**  poly1, poly2, poly3 // pointers to polynomial linked lists 1, 2 & 3 respectively  p1=poly1->next  p2=poly2->next  while(p1!=NULL && p2!=NULL)  {  if(p1->exp == p2->exp)  insert(poly3, p1->coeff-p2->coeff,p1->exp)  p1=p1->next  p2=p2->next  }  if(p1->exp>p2->exp)  insert(poly3, p1->coeff,p1->exp)  else  insert(poly3, p2->coeff,p2->exp)    if(p1==NULL)  while(p2!=NULL)  insert(poly3, p2->coeff,p2->exp)  if(p2==NULL)  while(p1!=NULL)  insert(poly3, p1->coeff,p1->exp) | <CO1> |
| <KL2> | 9. Outline a function in C, *EvalExprn(str \*exp)* that takes a postfix expression, evaluates it and returns the value. Assume that Stack ADT is available.  Input: 546\*+  Output: 29  Solution:  int evalExp(exp)  {  Int first, second, result;  n=len(exp)  create a stack S  for i=0 to n-1  {  if exp[i] is operand  push (S, (int)exp[i])  else  {  second= (int)pop(S)  first= (int)pop(S)  result=first+second  switch(exp[i])  {  case ‘+’: result=second+first  case ‘-’: result=second-first  case ‘\*’: result=second\*first  case ‘/’: result=second/first  }  push(S,result)  }  } | <CO2> |

**Part – C (2×10 = 20 Marks)**

|  |  |  |
| --- | --- | --- |
| <KL3> | 10. Develop a C function *Merge(Input1[], Input2[], Result[])* which takes two arrays of integers in sorted order and store the merged sorted data in a third array.  Input:  Input1[] = {4,7,11,45}  Input2[] = {2,8,9,16,21,71}  Output:  Result[] = {2,4,7,8,9,11,16,21,45,71}  **Solution:**  i=0, j=0, k=0  size1 // size of array 1  size2 // size of array 2  while(in1[i]<size1 && in2[i]<size2)  if(in1[i]<in2[j])  {  res[k++]=in1[i]  i++  }  else  {  res[k++]=in2[j]  j++  }  if(j>size)  while(i<size1)  res[k++]=in1[i++]  if(i>size)  while(j<size2)  res[k++]=in2[j++] | <CO1> |
| (OR) | | |
| <KL3> | 11. Suppose you have a pointer to the head of singly linked list. Normally, each node in the list only has a pointer to the next element, and the last node’s pointer is NULL. Unfortunately, your list might have been corrupted by a bug in somebody else’s code, so that the last node has a pointer back to some other node in the list instead.  Chart, box and whisker chart  Description automatically generated  A Normal Linked List  Chart, box and whisker chart  Description automatically generated  Corrupted Linked List  Develop an iterative algorithm that determines whether the linked list is corrupted or not. Your algorithm must not modify the list. Write the algorithm in O(n2) time and evolve another version in O(n). bruteforce method – 5 and O(N) method - 5  **Solution:**  /\* Link list node \*/  **struct** Node {  **int** data;  **struct** Node\* next;  };    /\* Function to detect loop. \*/  **void** detectLoop(**struct** Node\*, **struct** Node\*);    /\* This function detects loop in the list    If loop was there in the list then it returns 1,    otherwise returns 0 \*/  **int** detectLoop(**struct** Node\* list)  {  **struct** Node \*slow\_p = list, \*fast\_p = list;        // Iterate and find if loop exists or not  **while** (slow\_p && fast\_p && fast\_p->next) {          slow\_p = slow\_p->next;          fast\_p = fast\_p->next->next;            /\* If slow\_p and fast\_p meet at some point then there             is a loop \*/  **if** (slow\_p == fast\_p)                /\* Return 1 to indicate that loop is found \*/  **return** 1;          }      }        /\* Return 0 to indicate that there is no loop\*/  **return** 0;  } | <CO1> |
| <KL3> | 12. Write an algorithm to read a string of characters and determine whether it is a palindrome. A palindrome is a sequence of characters that reads the same forward and backward. Eg. Madam. The character “.” is end of string. Write “Yes” if the input is a palindrome; else write “No”. Use a Stack and a Queue data structure. Write all the required sub algorithms of Stack and Queue operations. Stack operations – 3 Q operations – 3 Palindrome function -4    **Solution:**  Stack S  Queue Q  i=0  while(i<=strlen(str))  {  enqueue(Q,str[i++])  push(S,str[i++])  }  while(isEmpty(Q))  if(dequeue(Q)!=top(S)  return 0  return 1 | <CO2> |
| (OR) | | |
| <KL3> | 13. Construct a Data Structure *SpecialStack* that supports operations such as *pushS(), popS(), isEmptyS(), isFullS()* and an additional operation *getMin()* which should return minimum element from the *SpecialStack. SpecialStack* uses a two traditional stacks; first one to stack the given elements and second one to keep track of the minimum of the elements currently in the first stack. It means that while pushing an element in the first stack, push the minimum in the second stack; while popping out from the first stack, pop out from second stack too. All these operations of *SpecialStack* must be O(1). Build *SpecialStack* ADT using a standard *Stack* ADT and write the algorithms for its operations  **Operations push – 4 pop – 2 getMin 2 overflow/underflow - 2**  **Solution:**  Assume that both stacks are initially empty and 18, 19, 29, 15 and 16 are inserted to the SpecialStack.   |  |  | | --- | --- | | When we insert 18, both stacks change to following. | | | Actual Stack  18 <--- top | Auxiliary Stack  18 <---- top | | When 19 is inserted, both stacks change to following. | | | Actual Stack  19 <--- top  18 | Auxiliary Stack  18 <---- top  18 | | When 29 is inserted, both stacks change to following. | | | Actual Stack  29 <--- top  19  18 | Auxiliary Stack  18 <---- top  18  18 | | When 15 is inserted, both stacks change to following. | | | Actual Stack  15 <--- top  29  19  18 | Auxiliary Stack  15 <---- top  18  18  18 | | When 16 is inserted, both stacks change to following. | | | Actual Stack  16 <--- top  15  29  19  18 | Auxiliary Stack  15 <---- top  15  18  18  18 | | <CO2> |

**--------------**

**Guidelines**

1. The question paper should be set in accordance with Bloom’s Taxonomy given in APPENDIX – A. The questions in a desired knowledge level must contain the respective action verbs.
2. The Knowledge level (Eg. <K2>) and the course outcome (Eg. <CO2>) should be mentioned against each question and subdivisions in the respective columns.
3. Both the questions in “either or” type must be set in the same knowledge level and must be from same CO.
4. In the case of “either or” type questions, the keyword (OR) must be in a separate row.
5. In the case of sub-divisions in a question, it is preferable to have the same knowledge level.
6. The marks assigned to each question in the case of subdivisions should be mentioned clearly in the end of the question within brackets and with a keyword Marks. (Eg. (5 Marks) ).
7. Add the keyword “Options” before the choices of an objective type question in Part A.
8. Once the question paper is set, its adherence to the guidelines in terms of knowledge levels and marks distribution can be assessed using the software QPAS (Question Paper Auditing System).

**APPENDIX – A**

**Bloom’s Taxonomy Action Verbs**

|  |  |  |
| --- | --- | --- |
| **K Level** | **Bloom’s Definition** | **Action Verbs** |
| K1  Remember | Exhibit memory of  Previously learned  Material by recalling  facts, terms, basic  concepts, and answers. | Choose, Define, Find, How, Label, List, Match, Name, Omit, Recall, Relate, Show, Spell, Tell, What, When, Where, Which, Who, Why. |
| K2  Understand | Demonstrate understanding  of facts and ideas by organizing, comparing, translating, interpreting,  giving descriptions and stating main ideas. | Classify, Compare, Contrast, Demonstrate, Explain, Extend, Illustrate, Infer, Interpret, Outline, Relate, Rephrase, Show, Summarize, Translate |
| K3  Apply | Solve problems to new situations by applying acquired knowledge, facts, techniques, and rules in a different way. | Apply, Build, Construct, Develop, Experiment with, Identify, Interview, Make use of, Model, Organize, Plan, Select, Solve, Utilize |
| K4  Analyse | Examine and break information into parts by identifying motives or causes. Make inferences and find evidence to support generalizations | Analyze, Assume, Categorize, Conclusion, Discover, Dissect, Distinguish, Divide, Examine, Function, Inference, Inspect, Motive, Relationships, Simplify, Survey, Take part in, Test for, Theme |
| K5  Evaluate | Present and defend opinions by making judgments about  information, validity of ideas, or quality of work based on a set of criteria. | Agree, Appraise, Assess, Award, Choose, Compare, Conclude, Criteria, Criticize, Decide, Deduct, Defend, Determine, Disprove, Estimate, Evaluate, Explain, Importance, Influence, Interpret, Judge, Justify, Mark, Measure, Opinion, Perceive, Prioritize, Prove, Rate, Recommend, Rule on, Select, Support, Value |
| K6  Create | Compile information together in a different way by combining elements in a new pattern or proposing alternative solutions. | Adapt, Build, Change, Choose, Combine, Compile, Compose, Construct, Create, Delete, Design, Develop, Discuss, Elaborate, Estimate, Formulate, Happen, Imagine, Improve, Invent, Make up, Maximize, Minimize, Modify, Original, Originate, Plan, Predict, Propose, Solution, Solve, Suppose, Test, Theory |