Massey University
ALBANY CAMPUS

Mid-Semester Test
159201
Semester One - 2013

Time Allowed: 45 Mins

INSTRUCTIONS

Attempt **ALL** questions.
Circle ONE answer for each question on this paper.

Write your ID number below.

ID No:_____

This test contributes 20% to the final assessment
(1 mark per question)

Calculators are permitted.

Turn over to pg. 2...

1. The following code snippet shows a function that *searches* elements (Nodes) in a linked-list. Unfortunately, the list is implemented as a ***circular list***. When this function is used and the number is not found, it enters in an infinite loop.

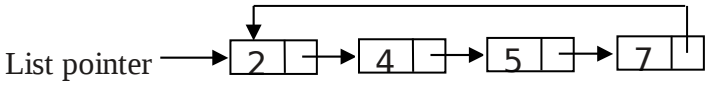


Figure 1: a circular linked-list.

```
...
struct Node {  //declaration
  int number;
  Node *next;
};
Node *A;
...
1  void Search(Node *listpointer, int a) {
2   Node *current; int i=0;
3   current = listpointer;
4   while (current!=NULL){
6     if (current->number == a) {
7       printf("number %d is at position %d",a,i);
8       return;
9     }
10     current = current->next;
11     i++;
12   }
13   printf("The searched number is not in the list");
14 }
...
```

Circle the option that best describes the change to the function that **would stop the infinite loop** when the number is not in the list:

a) change line 4 to: `while (current->next!= NULL)`
b) change line 4 to: `while (current==listpointer)`
c) change line 4 to: `while (current->next!=listpointer)`
d) change line 4 to: `while (current->number!=x)`

2. After changing line 4 of the code above, the programmer tries to find number 7, the last element. It does not print number "7", instead it prints "the search number is not in the list". Circle the answer that best describes **how to fix this problem** (add this code after line 12):

a)
```
if (current->number == a) { printf(number %d is at position %d",a,i);}
```

b)
```
if (current->number == a) {
    printf(number %d is at position %d",a,i);
    return;
  }
```

Turn over to pg. 3...

c) `if (current->number != a) {printf(number %d is at position %d",a,i);}`

d)
```
if (current == a) {
     printf(number %d is at position %d",a,i);
     return;
   }
```

3. The following function receives two linked-list pointers as parameters, and carries out some operation.

```
void Mystery_function(Node * &listpointer1, Node * listpointer2) {
  Node *current, *prev;
  current = listpointer1;
  prev = NULL;
  while (current != NULL) {
      prev = current;
      current = current->next;
  }
  prev->next=listpointer2;
}
```

Circle the option that best describes the end result of the function above:

a) The two linked-lists are compared
b) The two linked-lists are deleted
c) The two linked-lists are joined onto linked-list 1
d) Linked-list 1 is copied onto linked-list 2

4. Circle the correct option to complete the following statement about the order of adding and deleting elements for queues and stacks:
Queues are _____ and **stacks** are _____ .

a) FIFO FIFO
b) LIFO FIFO
c) LIFO LIFO
d) FIFO LIFO

NOTE: FIFO = first in, first out LIFO = last in, first out

5. Mark the option that best describes the basic operation for a standard **queue** (ADT) called push, where one parameter (value) is passed, e.g., **Push(value)**:

a) it pushes an element out of the queue
b) it adds a new element to the queue with the value passed as a parameter
c) it searches for value in the entire queue
d) it deletes the entire queue

6. Mark the option that best describes the basic operation for a standard **stack** (ADT) called **Top ( )** :

Turn over to pg. 4...

a) it returns the *first* element that was added to the stack (FIFO)
b) it returns the *last* element that was added to the stack (LIFO)
c) it returns the middle element that was added to the stack
d) it does not returns any element because as Top() is not a stack operation.

7. A possible solution for the "**creeping problem**" on queues implemented with arrays is:

a) delete the whole array every time it gets filled up.
b) the "creeping problem" does not occur in queues implemented with arrays.
c) use a circular array.
d) make the array bigger

8. Evaluate and circle the answer for the following Reverse Polish Notation (RPN):

```
27 3 4 6 * + / 8 +
```

a) 8
b) 10
c) 9
d) 1

9. Given the fragment of the code below, which line is ***correct*** when referring to the values of variables a, b or c?

```
...
struct Setofnumbers{
      int a;
      float b;
      double c;
};
…
main(){
Setofnumbers set1;
Setofnumbers *set2;
…
//do something with the values of a, b or c from set1 or set2
…
}
```

a) set1.a = set2.a
b) set2.b = set1->b
c) set2->c = set1.c
d) set1->a = set2->a

Turn over to pg. 5...

The following pseudo-code is relevant to questions **10** and **11**:

```
1     Stack S;
2     string expr;
3     int op1, op2, result;
4     read
5     while (there is still char in expr)
6           if (is a number)  push number onto S;
7           if (is an operator) {
8             op2 = top of S;
9             pop S;
10            op1 = top of S;
11            pop S;
12            result = apply operator to op1 and op2;
13            push result onto S;
14          }
15    }
16    display top of S;
```

10. Given the pseudo-code for evaluating a RPN expression, circle the answer that best describes how to find the condition "the expression has too many numbers":

a) before lines 8 and 10, test if S is empty. If it **is empty**, print "too many numbers".
b) after line 15, test if S is empty. If it **is empty**, print "too many numbers".
c) after line 15, test if S is empty. If it **is NOT empty**, print "too many numbers".
d) before lines 8 and 10, test if S is empty. If it **is NOT empty**, print "too many numbers".

11. In the pseudo-code above, circle the answer that best describes how to find the condition "the expression has too many operators"

a) before lines 8 and 10, test if S is empty. If it **is empty**, print "too many operators".
b) after line 15, test if S is empty. If it **is empty**, print "too many operators".
c) before lines 8 and 10, test if S is empty. If it **is NOT empty**, print "too many operators".
d) after line 15, test if S is empty. If it **is NOT empty**, print "too many operators".

12. A queue (ADT) implemented as a C++ class with linked-lists need to have its size (number of elements) evaluated frequently. Circle the option that would give the best performance:

a) traverse the linked-list every time size is needed
b) add a counter variable to the queue class and update it when using isEmpty()
c) add a counter variable to the queue class and update it when using Leave() and Join()
d) add a pointer previous to the node of the linked-list

13. About a **list** (ADT), which of the following options best describe its characteristics:

a) lists cannot be extended, can be accessed via index, and are very good for random access.
b) lists can be extended, but elements have to be accessed sequentially.
c) lists can be extended, can be accessed via index, and are very good for random access.
d) lists cannot be extended, and elements have to be accessed sequentially.

Turn over to pg. 6...

14. Consider the binary tree in the figure below (figure 2). What is the result of traversing the tree using **pre-order** traversal?

a) A B D E C F G
b) D E B F G C A
c) D B E A F C G
d) A B C D E F G

15. Consider the binary tree in the figure below (figure 2). What is the result of traversing the tree using **in-order** traversal?

a) A B D E C F G
b) D E B F G C A
c) D B E A F C G
d) A B C D E F G

16. Consider the binary tree in the figure below (figure 2). What is the result of traversing the tree using **post-order** traversal?

a) A B D E C F G
b) D E B F G C A
c) D B E A F C G
d) A B C D E F G

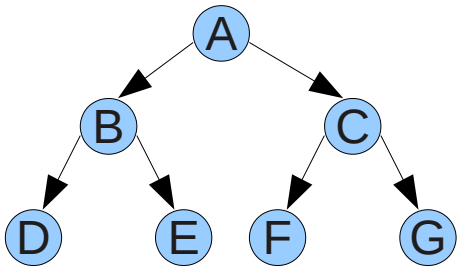


Figure 2: A binary tree.

17. Consider using vectors implemented in the STL. Suppose you declare a vector<int> v, and have a variable int num. Circle the statement that is ***INCORRECT:***

We can use the statement: v[i] = num;
a) with **any** i, as long as v was resized with: v.resize(maxsize);
b) with **i<maxsize**, as long as v was resized with: v.resize(maxsize);
c) with **any i** as the vector v can be resized at any point in the program.
d) with **i<maxsize**, as long as maxsize numbers were pushed into v with: v.push_back(num);

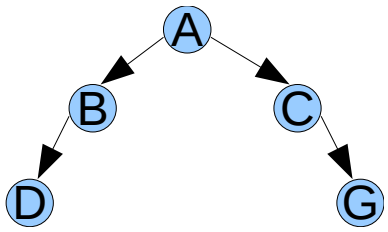
Turn over to pg. 7...

18. Consider the following code snippet for a binary tree:

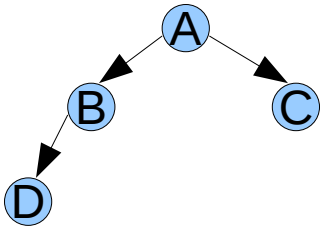
```
class Tree {
private:
  char data;
  Tree *leftptr, *rightptr;

public:
  Tree(char newthing, Tree* L, Tree* R);
  ~Tree() { }
};

Tree::Tree(char newthing, Tree* L, Tree* R) {
  data = newthing;
  leftptr = L;
  rightptr = R;
}

Tree *T1, *T2, *T3, *T4, *T5, *T6, *T7, *R1;

int main() {
     T1 = new Tree('D', NULL, NULL);
     T2 = new Tree('C', NULL, NULL);
     T3 = new Tree('B', T1, NULL);
     T4 = new Tree('E', NULL, NULL);
     T5 = new Tree('G', NULL, NULL);
     T6 = new Tree('F', T4, T5);
     T7 = new Tree('A', T3, T2);
     R1=T7;
}
```

When we run the code above, which diagram is the best representation of the tree at 'R1'?

a)

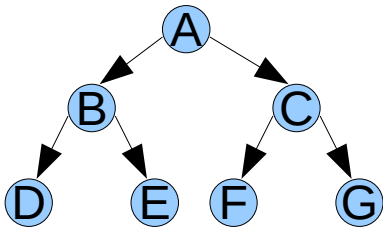


b)

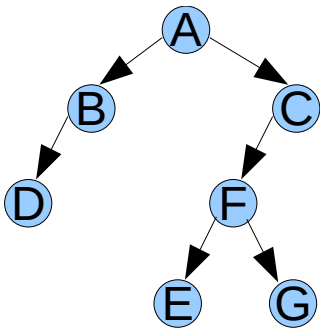


Turn over to pg. 8...

c)



d)



19. The following C++ code implements a method in a class called List.

```
template <class T>
  bool List<T>::MysteryMethod() {
       Node * temp1, *temp2;
       current = front->next;
       temp2=front;
       while (current != NULL){
         temp1=current->next;
         current->next = front;
         front = current;
         current=temp1;
       }
       temp2->next=NULL;
  }
```

Circle the option that best describes the outcome for the list after running the method:
a) repeatedly copies the values of the elements to current
b) inverts the head with the tail without affecting other elements
c) the elements are in reverse order
d) delete all the values while keeping empty nodes

Turn over to pg. 9...

20. The following C function is used to traverse a Tree based on storing tree pointers in a stack. Which traverse is this code implementing?

```
void SomeOrderIterative(Tree *root) {
  stack<Tree*> nodeStack;
  Tree *curr = root;
  while (1) {
        if (curr != NULL) {
              nodeStack.push(curr);
              curr = curr->Left();
              continue;
        }
        if (nodeStack.size() == 0) {
              return;
        }
        curr = nodeStack.top();
        nodeStack.pop();
        printf("%c ",curr->RootData());
        curr = curr->Right();
  }
}
```

a) Pre-order
b) In-order
c) Post-order
d) None of the above

+ + + + + + + + + +

Turn over to pg. ...