Suggested Solutions to Midterm Problems

1. Explain the following terms:

   (a) abstract data type (ADT)

   Solution. An ADT is a collection of data and a set of operations on the data. □

   (b) precondition and postcondition (of a function/method)

   Solution. A precondition is a statement of the conditions on the input at the beginning of a function. A postcondition is a statement of the conditions on the result/output at the end of a function. They form the most important part of the contract for a function. A function is implemented correctly, if the following holds: whenever the function is invoked with the precondition being true, it will terminate and return a result that satisfies the postcondition. □

2. Consider the ADT polynomial for integer polynomials over a single variable x, which includes the following operations:

   - `+degree(): int`
     Returns the degree of a polynomial
   - `+coefficient(n: int): int`
     Returns the coefficient of $x^{\mathtt{n}}$.
   - `+changeCoefficient(newCoeff: int, n: int): bool`
     Replaces the coefficient of $x^{\mathtt{n}}$ with `newCoeff`, and returns *true* if the operation is carried out successfully and *false* otherwise.

   (a) Using these operations, write statements (in pseudocode) to compute the sum of two polynomials.

   Solution. See the slides for TA Session #4 on 12/28. □

   (b) Can one find out the highest possible degree of a polynomial supported by a particular implementation of the ADT? How?

   Solution. See the slides for TA Session #4 on 12/28. □

3. The following is a recursive, but simplified, variant of Euclid's algorithm for computing the greatest common divisor of two positive integers.

```
int gcd(int a, int b)
{
  if (a == 0)
    return b;
  if (b == 0)
    return a;
  if (a > b)
    return gcd(a - b, b);
  else
    return gcd(a, b - a);
}
```

(a) What are the base cases? Why are they appropriate?

Solution. The base cases are when $a = 0$ or $b = 0$. They are appropriate, as under either of the conditions, the computation may stop with the correct greatest common divisor (represented by the other non-zero argument). This means that the algorithm terminates correctly if it ever will.

To be completed. □

(b) Will the base cases always be reached? Why?

Solution. Yes, one of the base cases will eventually be reached, assuming that both of the original input numbers are positive. This means that the algorithm will terminate.

To be completed. □

4. Consider the array-based implementation of the ADT bag, the C++ class `ArrayBag`. We have implemented the function `remove()` with a static array as follows:

```
template<typename ItemType>
bool ArrayBag::remove(const typename& anEntry)
{
  int locatedIndex = getIndexOf(anEntry);
  bool canRemoveItem = (locatedIndex > -1);
  if(canRemoveItem)
  {
    itemCount--;
    items[locatedIndex] = items[itemCount];
  }
  return canRemoveItem;
}
```

Suppose that we now want to implement `ArrayBag` with dynamic memory allocation. The class definition has been changed to

```
template<typename ItemType>
class ArrayBag: public BagInterface
{
private:
  static const int DEFAULT_CAPACITY = 6;
  ItemType* items;
  int itemCount;
  int maxItems;
  int getIndexOf(const ItemType& target) const;
public:
  // public member functions are omitted
};
```

where `items` is a pointer pointing to the dynamic array. We now want to modify `remove()` and add the following feature: *When the number of items is no greater than half of the current bag capacity, cut down the bag capacity to its half.* Rewrite `remove()` to add this feature.

Note. You may (1) assume that the bag capacity will always be an odd number, and (2) use the function `getIndexOf()` to obtain the array index of a given item if it exists or -1 otherwise.

Solution. See the slides for TA Session #4 on 12/28. □

5. Consider the link-based implementation of the ADT bag, the C++ class `LinkedBag`. We have implemented the function `add()`, which adds an item at the beginning position of our bag, as follows:

```
template<class ItemType>
bool LinkedBag<ItemType>::add(const ItemType& newEntry)
{
  Node<ItemType>* newNodePtr = new Node<ItemType>();
  newNodePtr->setItem(newEntry);
  newNodePtr->setNext(headPtr);
  headPtr = newNodePtr;
  itemCount++;
  return true;
}
```

Rewrite this function to add an item at the *ending* position of our bag.

Solution. See the slides for TA Session #4 on 12/28. □

6. When we implemented the classes `LinkedBag`, we used C++ templates to make our classes capable of storing all types of items. With this in mind, please answer the following questions.

   (a) What will be the output of the following program?

```
#include <iostream>
using namespace std;

template<typename A, typename B>
void f(A a, B b)
{
  cout << a + b << endl;
}

int main()
{
  f<double, double>(1.23, 10.3);
  f<double, int>(1.23, 10.3);
  f<int, double>(1.23, 10.3);
  f<char, char>(1.23, 10.3);
  return 0;
}
```

   Solution. See the slides for TA Session #4 on 12/28. □

(b) Suppose that we have two classes `A` and `B`, and want to use one bag to store items of both classes. Propose a way to modify `LinkedBag` or `A` and `B` to complete the task.

Solution. See the slides for TA Session #4 on 12/28. □

7. Convert the infix expression `(a / (b / c)) + d - e * f` to the postfix form. Please follow the conversion algorithm discussed in class (see the appendix), and show the status of the stack and the current postfix expression after each character of the infix expression is processed.

Solution. See the slides for TA Session #4 on 12/28. □

8. Consider again the infix to postfix conversion algorithm. Suppose we want to allow the exponent operator `^`, which has higher precedence than `*` and `/` and is associated to the right, i.e., `a ^ b ^ c` equals `a ^ (b ^ c)`. Please modify the conversion algorithm to allow the additional `^`.

Solution.

```
for (each character ch in the infix expression) {
  switch (ch) {
    ...
    case ’+’, ’*’:  // These are left-associative operators
      // Process operators of higher or equal precedence
      while (!aStack.isEmpty() and
             aStack.peek() is not a ’(’ and
             precedence(ch) <= precedence(aStack.peek())) {
        postfixExp = postfixExp + aStack.peek()
        aStack.pop()
      }
      aStack.push(ch)  // Save the operator
      break
    case ’^’:          // ^ is right-associative
      aStack.push(ch)  // Save, to be processed later
      break
   ...
  }
}
// Append to postfixExp the operators remaining in the stack
while (!aStack.isEmpty()) {
  postfixExp = postfixExp + aStack.peek()
  aStack.pop()
}
```

The exponentiation operator `^` is right-associative and has the highest precedence, so we always push it on to the stack whenever we encounter one. For example, we would push three `^`’s if we do encounter three of them consecutively. They will be popped out when another operator of lower precedence or a ’)’ is encountered.

□

9. How are stacks and recursion related? Please explain the relationship by considering solutions to the task of determining the existence of a path on a directed graph.

Solution. The activities of invocation and return during the execution of a recursive function may be emulated by operations a stack, as the activities exhibit a last-in-first-out property just like a stack. (In fact, the invocation and return of any function, recursive or non-recursive, is implemented with the help of a stack.) A recursive call can be emulated by a push and a return by a pop.

To be completed. ☐

10. Design an algorithm (in pseudocode) to sort a stack of integers in increasing order with the smallest element on the top. You may use additional stacks and integer variables, but no other data structures (including arrays). You should try to use as few additional stacks as possible.

    Solution. See the slides for TA Session #4 on 12/28. ☐

Appendix

- Below is the algorithm discussed in class for converting an infix expression to a postfix expression.

```
for (each character ch in the infix expression) {
  switch (ch) {
    case operand:  // Append operand to the end of postfixExp
      postfixExp = postfixExp + ch
      break
    case '(':      // Save '(' on the stack aStack
      aStack.push(ch)
      break
    case operator: // Process operators of higher precedence
      while (!aStack.isEmpty() and
             aStack.peek() is not a '(' and
             precedence(ch) <= precedence(aStack.peek())) {
             postfixExp = postfixExp + aStack.peek()
             aStack.pop()
      }
      aStack.push(ch)  // Save the operator
      break
    case ')': // Pop the stack until matching '('
      while (aStack.peek() is not a '(') {
        postfixExp = postfixEP + aStack.peek()
        aStack.pop()
      }
      aStack.pop()  // Remove the matching '('
      break
  }
}
// Append to postfixExp the operators remaining in the stack
while (!aStack.isEmpty()) {
  postfixExp = postfixExp + aStack.peek()
  aStack.pop()
}
```