Data Structure Lab Assignments

March 24, 2017
Assignment 3

Definition of an Abstract Class

Define the following abstract class "List".

```cpp
template <typename E>
class List {
    void operator = (const List&); {}
    List(const List&) {}

    public:
        // constructors and destructors
        List() {}
        virtual ~List() {};

        // Return the number of elements in the List
        virtual int NumberOfElements() const = 0;

        // Return the Location of current element
        virtual int CurrentLocation() const = 0;

        // Return the data of current element
        virtual const E& getData() const = 0;

        // clear all the data from the List
        virtual void clear() = 0;

        // insert a data in the List at current location
        virtual void insert(const E& data) = 0;

        // insert a data in the end of the List
        virtual void append(const E& data) = 0;

        // delete a data at the current Location
        virtual E remove() = 0;

        // other methods and members...
    }
```
//set the current Location to the start of the List
virtual void setListLocationToStart() = 0;

//set the current Location to the end of the List
virtual void setListLocationToEnd() = 0;

//set the current Location to the element on left side
//of current element
virtual void previousElement() = 0;

//set the current Location to the element on left side
//of current element
virtual void nextElement() = 0;

//Set current Location to a new Location
virtual int setToNewLocation(int location) = 0;
};

Implementation of derived class based on Array List

Implement the following derived class name "arrayList" which inherits the abstract class "List" and implements its member functions.

template<typename E>
class arrayList : public List<E>
{
    int arraySize; // maximum size of the list
    int currentSize; // number of data elements stored
    int currentLocation; // location pointer of the list
    E* dataArray; // Array for storing the data

public:
    //constructors and destructors
    arrayList(int size)
    {
        arraySize = size;
        currentSize = currentLocation = 0;
        dataArray = new E[arraySize];
    }

    ~arrayList()
    {
        delete [] dataArray;
    }
}
Assignment 4

Implementation of derived class based on Link List

Implement the following derived class name "linkList" which inherits the abstract class "List" in Assignment 3 and implements its member functions.

```cpp
template<typename E>
class linkList : public List<E>
{
    node<E>* firstElement;
    node<E>* lastElement;
    node<E>* currentElement;
    int sizeOfList;

public:
    // constructors and destructors
    linkList()
    {
        firstElement = lastElement = currentElement = new node<E>;
        sizeOfList = 0;
    }

    ~linkList()
    {
        for (currentElement = firstElement; firstElement != NULL; firstElement = firstElement->nextElement)
        {
            delete currentElement;
        }
    }
};
```

The above implementation needs the following class "node" for its definition.

```cpp
template<typename E>
class node
{
    // ...
public:
    E data; // stores the data for the current node
    node *nextElement; // points to the next node

    // constructor
    node(const E& dataValue, node *nextElementPointer = NULL):
    data(dataValue), nextElement(nextElementPointer) {}
Assignment 5

Definition of an Abstract Class

Define the following abstract class "Stack".

```cpp
template <typename E>
class Stack {
    void operator = (const Stack&) {}
    Stack(const Stack&) {}

public:
    // constructors and destructors
    Stack() {}
    virtual ~Stack() {}  

    // Return the length of the stack
    virtual int length() const = 0;

    // Return the Location of current element
    virtual void push(const E& element) = 0;

    // Clear the current data element from stack
    virtual E pop() = 0;

    // Return a copy of the current element
    virtual const E& currentElement() const = 0;

    // Remove all the elements from the stack
    // and free the occupied memory without causing
    // memory leak
    virtual void clearStack() = 0;
};
```
Implementation of derived class "StackArrayList" based on ArrayList

Implement the following derived class name "StackArrayList" which inherits the abstract class "Stack" and implements its member functions.

template <typename E>
class StackArrayList: public Stack<E>
{
    int stacksize; // size of the stack
    int currentLocation; // location pointer keeping track of current element
    arrayList<E>* StackArray; // Array for storing the data

public:
    // constructors and destructors
    StackArrayList(int size)
    {
        stacksize = size;
        StackArray = new arrayList<E>(size);
        currentLocation = 0;
    }
    ~StackArrayList()
    {
        delete StackArray;
    }
};
Assignment 6

Definition of an Abstract Class

Define the following abstract class "Queue".

```cpp
template <typename E>
class Queue
{
    void operator = (const Queue&) {}
    Queue(const Queue&) {}

    public:
        // constructors and destructors
        Queue() {}
        virtual ~Queue() {};

        // Insert an element at the end of the queue
        virtual void insert(const E&) = 0;

        // Remove the first element of the queue
        virtual E remove() = 0;

        // Return the length of the queue
        virtual int length() const = 0;

        // Return a copy of the first element
        virtual const E& firstElement() const = 0;

        // Remove all the elements from the Queue
        // and free the occupied memory without causing memory leak
        virtual void clearQueue() = 0;
};
```
Implementation of derived class "QueueArrayList" based on Array List

Implement the following derived class name "QueueArrayList" which inherits the abstract class "Queue" and implements its member functions. The data storage is based on previously implemented "ArrayList" class.

template <typename E>
class QueueArrayList : public Queue<E>
{
    int maximumSize; //maximum size of the array list
    int firstElement; //index of the first element
    int lastElement; //index of the last element
    ArrayList<E>* QueueArray; // Array for storing the data

    public:
    // constructors and destructors
    QueueArrayList(int size)
    {
        maximumSize = size;
        QueueArray = new ArrayList<E>(size);
        firstElement = lastElement = 0;
    }

    QueueArrayList()
    {
    }

    ~QueueArrayList()
    {
        delete QueueArray;
    }
};

Implementation of derived class "QueueLinkList" based on Array List

Implement the following derived class name "QueueLinkList" which inherits the abstract class "Queue" and implements its member functions. The data storage is based on previously implemented "LinkList" class.
template <typename E>
class QueueLinkList: public Queue<E>
{
    // size of the link list
    int size;
    // Array for storing the data
    linkList<E>* QueueArray;

public:
    // constructors and destructors
    QueueLinkList()
    {
        QueueArray = new linkList<E>();// for the next statement ensure that you have implemented
        // destructor in the linkList class.
        delete QueueArray;
    }

};
Assignment 7

Implement the C++ functions for the following sorting algorithms operating on the ArrayList class (defined in Assignment 3):

1. Selection Sort
2. Bubble Sort
3. Merge Sort
4. Quick Sort

These functions are standalone functions and need not be a member function of any class.

Suggestion (optional): Try to use "List" class pointer as a function argument which can be used to send either "ArrayList" or "linkList" data to these implemented functions. This way your implemented function can be reused for "linkList" class in future if the need arises.
Assignment 8

Implement a class called "binaryHeap" using templates as shown below:

```cpp
template <typename E>
class binaryHeap
{
    E* heap; // pointer to the heap array
    int maxsize; // maximum size of the array
    int numberOfElements; // current number of elements in the heap

    public:
        // constructor.
        heap(E* e, int max, int num);
        // returns current size of heap
        int sizeOfHeap() const;
        // Check if the given position is a leaf node.
        bool LeafNode(int position);
        // returns the position of right child.
        int rightChildNode(int position);
        // returns the position of right child.
        int leftChildNode(int position);
        // returns the position of parent.
        int parent(int position);
        // insert an element into heap.
        void insert(const E& element);
        // removes the element from the heap.
        E remove(int position);
    }
```

Define all the functions listed above. You are free to define your own helper functions in case the need arises.
Assignment 9

Modify appropriately the class ”binaryHeap” in the previous assignment with minimum changes to implement a priority queue.
Assignment 10

The C++ class "BinaryTreeNode" is defined as:

template <typename Key, typename E>
class BinaryTreeNode : public BinNode<E>
{
  E value;  // The node’s value
  BinaryTreeNode* leftChild;  // Pointer to left child
  BinaryTreeNode* rightChild;  // Pointer to right child

public:
  // constructors and destructor
  BinaryTreeNode() { leftChild = rightChild = NULL; }
  BinaryTreeNode(E value, BinaryTreeNode* l =NULL, BinaryTreeNode* r =NULL) { value = e; leftChild = l; rightChild = r; }
  ~BinaryTreeNode() // Destructor
  { delete leftChild; delete rightChild; }

  // Functions to set and return the value and key
  E& getValue() { return value; }
  void setValue(const E& e) { value = e; }

  // Functions to set and return the children
  inline BinaryTreeNode* left() const { return leftChild; }
  void setLeft(BinaryTreeNode<E>* b) { leftChild = b; }
  inline BinaryTreeNode* right() const { return rightChild; }
  void setRight(BinaryTreeNode<E>* b) { rightChild = b; }

  // Return true if it is a leaf, false otherwise
  bool isLeaf() { return (leftChild == NULL) && (rightChild == NULL); }
};

Implement a binary tree using the above C++ class "BinaryTreeNode". Implement C++ functions for inserting and deleting nodes in the binary tree. Next, implement C++ functions for inorder, preorder and post-order traversals of the nodes of the implemented binary tree.