Swinburne University of Technology

Faculty of Science, Engineering and Technology

ASSIGNMENT COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures & Patterns
Assignment number and title:	7 – NTree Tree Traversal
Due date:	May 28, 2019, 10:30
Lecturer:	Dr. Markus Lumpe

Your name:____________________________ **Your student id:**________________

Check Tutorial	Wed 08:30	Wed 10:30	Wed 12:30	Wed 14:30	Thurs 08:30	Thurs 10:30	Thurs 12:30	Thurs 14:30	Fri 10:30

Marker's comments:

Problem	Marks	Obtained
1 – depth-first traversal	24	
2 – breath-first traversal	38	
3 – left linear representation (optional)	30	
Total	62(92)	

Extension certification:

This assignment has been given an extension and is now due on ____________

Signature of Convener:_________________________

Problem Set 7: NTree Tree Traversal

Using the template class `NTree` studied in tutorial 10 and the template class `Queue` defined in problem set 6, add tree traversal to class `NTree`:

```
#pragma once

#include <stdexcept>

#include "TreeVisitor.h"

template<class T, int N>
class NTree
{
private:
  T fKey;                     // T() for empty NTree
  NTree<T,N>* fNodes[N];      // N subtrees of degree N

  NTree();                    // sentinel constructor

public:
  static NTree<T,N> NIL;      // sentinel

  NTree( const T& aKey );     // a simple NTree with key and N subtrees

  NTree( const NTree& aOtherNTree );             // copy constructor
  virtual ~NTree();                              // destructor
  NTree& operator=( const NTree& aOtherNTree );  // assignment operator

  virtual NTree* clone();                        // clone a tree

  bool isEmpty() const;       // is tree empty
  const T& key() const;       // get key (node value)

  // indexer (allow for result modification by client - no const in result)
  NTree& operator[]( unsigned int aIndex ) const;

  // tree manipulators (using constant references)
  void attachNTree( unsigned int aIndex, const NTree<T,N>& aNTree );
  const NTree& detachNTree( unsigned int aIndex );

  // depth-first traversal
  void traverseDepthFirst( const TreeVisitor<T>& aVisitor ) const;

  // breadth-first traversal
  void traverseBreadthFirst( const TreeVisitor<T>& aVisitor ) const;
};
```

Use "`TreeVisitor.h`" available on Canvas as a start to implement the tree traversal.

Problem 1

Implement "depth-first traversal". You can follow the approach shown in class. Please note that there is no in-order traversal for NTree.

Test harness:

```
void testDepthFirstTraversal()
{
  string A( "A" );
  string A1( "AA" );
  string A2( "AB" );
  string A3( "AC" );
  string AA1( "AAA" );
  string AB1( "ABA" );
  string AB2( "ABB" );

  typedef NTree<string,3> NS3Tree;

  NS3Tree lTree( A );

  lTree.attachNTree( 0, *(new NS3Tree( A1 )) );
  lTree.attachNTree( 1, *(new NS3Tree( A2 )) );
  lTree.attachNTree( 2, *(new NS3Tree( A3 )) );

  lTree[0].attachNTree( 0, *(new NS3Tree( AA1 )) );
  lTree[1].attachNTree( 0, *(new NS3Tree( AB1 )) );
  lTree[1].attachNTree( 1, *(new NS3Tree( AB2 )) );

  cout << "Depth-first traversal:" << endl;

  lTree.traverseDepthFirst( PreOrderVisitor<string>() );

  cout << endl << "Success." << endl;
}
```

Result:

```
Depth-first traversal:
A AA AAA AB ABA ABB AC
Success.
```

Problem 2

Implement "breadth-first traversal". You can follow the approach shown in class. You need a local queue variable in `traverseBreadthFirst`. To avoid unwanted copying, use a pointer to **const** `NTree<T,N>` as type for the required `Queue` value object. That is, specify `Queue<const NTree<T,N>*> lQueue`, if `lQueue` is the local queue object in `traverseBreadthFirst`.

Test harness:

```
void testBreadthFirstTraversal()
{
  string A( "A" );
  string A1( "AA" );
  string A2( "AB" );
  string A3( "AC" );
  string AA1( "AAA" );
  string AB1( "ABA" );
  string AB2( "ABB" );

  typedef NTree<string,3> NS3Tree;

  NS3Tree lTree( A );

  lTree.attachNTree( 0, *(new NS3Tree( A1 )) );
  lTree.attachNTree( 1, *(new NS3Tree( A2 )) );
  lTree.attachNTree( 2, *(new NS3Tree( A3 )) );

  lTree[0].attachNTree( 0, *(new NS3Tree( AA1 )) );
  lTree[1].attachNTree( 0, *(new NS3Tree( AB1 )) );
  lTree[1].attachNTree( 1, *(new NS3Tree( AB2 )) );

  cout << "Breadth-first traversal:" << endl;

  lTree.traverseBreadthFirst( TreeVisitor<string>() );

  cout << endl << "Success." << endl;
}
```

Result:

```
Breadth-first traversal:
A AA AB AC AAA ABA ABB
Success.
```

Problem 3

Implement "left linear representation". This can be achieved by using a depth-first tree traversal using a visitor that responds to both the `preVisit()` and the `postVisit()` methods. More precisely, you need to define a new visitor, called `LeftLinearVisitor`, that is a subclass of `TreeVisitor`, which overrides the corresponding virtual methods to obtain a left-linear representation of `NTree`.

Test harness:

```
void testLinearRepresentation()
{
  string A( "A" );
  string A1( "AA" );
  string A2( "AB" );
  string A3( "AC" );
  string AA1( "AAA" );
  string AB1( "ABA" );
  string AB2( "ABB" );

  typedef NTree<string,3> NS3Tree;

  NS3Tree lTree( A );

  lTree.attachNTree( 0, *(new NS3Tree( A1 )) );
  lTree.attachNTree( 1, *(new NS3Tree( A2 )) );
  lTree.attachNTree( 2, *(new NS3Tree( A3 )) );

  lTree[0].attachNTree( 0, *(new NS3Tree( AA1 )) );
  lTree[1].attachNTree( 0, *(new NS3Tree( AB1 )) );
  lTree[1].attachNTree( 1, *(new NS3Tree( AB2 )) );

  cout << "Linear representation:" << endl;

  lTree.traverseDepthFirst( LeftLinearVisitor<string>() );

  cout << endl << "Success." << endl;
}
```

Result:

```
Linear representation:
[A[AA[AAA]][AB[ABA][ABB]][AC]]
Success.
```

Submission deadline: Tuesday, May 28, 2019, 10:30.

Submission procedure: on paper (NTree traversal, LinearTreeVisitor).