Swinburne University of Technology

Faculty of Science, Engineering and Technology

FINAL EXAM COVER SHEET

Subject Code: COS30008
Subject Title: Data Structures & Patterns
Due date: June 3, 2021, 13:00
Lecturer: Dr. Markus Lumpe

Your name: Zoie Tad-y **Your student id:** 102884743

Check Tutorial	Wed 08:30	Wed 10:30	Wed 16:30	Thurs 08:30	Thurs 10:30	Thurs 14:30	Thurs 16:30	Fri 08:30	Fri 10:30	Fri 14:30
		X								

Marker's comments:

Problem	Marks	Time Estimate in minutes	Obtained
1	50	20	
2	54	15	
3	42	10	
4	60	15	
5	8+128=136	60	
Total	342	120	

This test requires approx. 2 hours and accounts for 50% of your overall mark.

Problem 1 (50 marks)

Answer the following questions in one or two sentences:

a. How can we construct a tree where all subtrees have the same degree? (4 marks)

1a) Predefine pointers to N number of nodes for each node created for a tree and make sure that the tree is strictly balanced. Degree being the number of child nodes for a specific node, setting a predefined definition that each node created will always point to either N number of nodes or 0 will ensure that each subtree will have have the same degree.

b. What are reference data members and how do we initialize them? (2 marks)

1b) Reference data members are "variables" that serve as an alias to an existing value stored in memory. We can intialise a reference member by using the type followed by "&" and the name of the variable reference, and equating them to a variable.

c. What is the difference between l-value and r-value references? (6 marks)

1c) L value references point towards or reference to a specific section in memory that may contain value, r-value references are simply references to value that is short lived and is not stored in memory. In most reference and even earlier early c++ discussion, r value references are simply referred to as the opposite of l value references. Typically l value references are in the left hand side of the assignment operator, and r value references are found in the right hand side of assignment operator. However l values may also appear on the right hand side of the when it is being assigned to another l value reference. R value references can never be modified or assigned with another value.

d. What is an object adapter? (6 marks)

1d) An Object adapter is a wrapper around existing objects that allows them to collaborate. Certain definitions of certain objects prevent them from working with the functionality of other objects, in the event that we need that objects functionality to work with another object we create adapters to transform or format one or two of the objects to make them seemingly compatible.

e. What is a key concept of an abstract data types? (4 marks)

1e) Abstract data types are data types that defined by their functionality, not strictly how they are constructed declaratively. Being abstract means that it exists as a thought or idea, and being an ADT is just that, for example two stacks that a programtically defined differently are still considered stacks as long as they adhere to the abstract definition of what a stack is and can do.

f. How do we define mutual dependent classes in C++? (4 marks)

1f)

We can use forward declaration, declare at least one of the classes in the definition of the other through references or pointers.

g. What must a value-based data type define in C++? (2 marks)

1g)

Value based data types must define copy constructors.

h. What is the difference between copy constructor and assignment operator and how do we guarantee safe operation? (8 marks)

1h)

Copy constructors creates new instances of the a given object with the same information that object has through assigning a spearate memory block for the new object, assignment operators assigns the entirety of one object to another another object without allocating new memory space. To ensure safe operation deleting the old instances, or freeing the memory space allocated from the old instance should be done to prevent memory leaks.

i. What is the best-case, average-case, and worse-case for a lookup in a binary tree? (6 marks)

1i)

Worst case is O(n) where each node is visited, best case is O(1) where the node being looked up is the root, and average case if O(log n) where approx half the nodes are visited in completing the look up.

j. You are given a set of n-1 numbers out of n numbers. How do we find the missing number n_k, $1 \leq k \leq n$, in linear time? (8 marks)

1j)

We can convert both sets, set one being the given n-1 numbers, and the other being the general set of n numbers, to its canonical form by sorting which is O(n log n) and then traverse both at the same time, when the values do not match the that means the value at that index on the general set of n numbers is the missing number. Linear traversal is O(n) so the entire process is O(nlogn) + O(n), drop O (nlogn) since O(n) is larger, hence O(n)

```
 1
 2  // COS30008, Final Exam, 2021
 3
 4  #pragma once
 5
 6  #include <stdexcept>
 7
 8  template<typename T>
 9  class TTreePostfixIterator;
10
11  template<typename T>
12  class TTree
13  {
14  private:
15
16      T fKey;
17      TTree<T>* fLeft;
18      TTree<T>* fMiddle;
19      TTree<T>* fRight;
20
21      TTree() : fKey(T())     // use default constructor to initialize fKey
22      {
23          fLeft = &NIL;       // loop-back: The sub-trees of a TTree object with
24          fMiddle = &NIL;     //            no children point to NIL.
25          fRight = &NIL;
26      }
27
28      void addSubTree( TTree<T>** aBranch, const TTree<T>& aTTree )
29      {
30          if ( !(*aBranch)->empty() )
31          {
32              delete *aBranch;
33          }
34
35          *aBranch = const_cast<TTree<T>*>(&aTTree);
36      }
37
38  public:
39
40      using Iterator = TTreePostfixIterator<T>;
41
42      static TTree<T> NIL;    // sentinel
43
44      // getters for subtrees
45      const TTree<T>& getLeft() const { return *fLeft; }
46      const TTree<T>& getMiddle() const { return *fMiddle; }
47      const TTree<T>& getRight() const { return *fRight; }
48
49      // add a subtree
50      void addLeft( const TTree<T>& aTTree ) { addSubTree( &fLeft, aTTree ); }
51      void addMiddle( const TTree<T>& aTTree ) { addSubTree( &fMiddle, aTTree ); }
52      void addRight( const TTree<T>& aTTree ) { addSubTree( &fRight, aTTree ); }
```

```
53
54      // remove a subtree, may through a domain error
55      const TTree<T>& removeLeft() { return removeSubTree( &fLeft ); }
56      const TTree<T>& removeMiddle() { return removeSubTree( &fMiddle ); }
57      const TTree<T>& removeRight() { return removeSubTree( &fRight ); }
58
59  // Problem 1: TTree Basic Infrastructure
60
61  private:
62
63      // remove a subtree, may through a domain error
64      const TTree<T>& removeSubTree(TTree<T>** aBranch)
65      {
66          if (*aBranch == &NIL)
67          {
68              throw std::domain_error("Branch does not exist");
69          }
70
71          *aBranch = &NIL;
72      }
73
74  public:
75
76      // TTree l-value constructor
77      TTree(const T& aKey) :fKey(aKey), fLeft(&NIL), fMiddle(&NIL), fRight(&NIL)
78      {}
79
80      // destructor (free sub-trees, must not free empty trees)
81      ~TTree()
82      {
83          if (empty())
84          {
85              throw domain_error("Empty TTree");
86          }
87          delete fLeft;
88          delete fMiddle;
89          delete fRight;
90
91      }
92
93
94      // return key value, may throw domain_error if empty
95      const T& operator*() const
96      {
97          if (empty())
98          {
99              throw std::domain_error("Empty Tree");
100         }
101
102         return fKey;
103     }
104
```

```
105     // returns true if this TTree is empty
106     bool empty() const
107     {
108         return this == &NIL;
109     }
110
111     // returns true if this TTree is a leaf
112     bool leaf() const
113     {
114         return fLeft->empty() && fMiddle->empty() && fRight->empty();
115     }
116
117 // Problem 2: TTree Copy Semantics
118
119     // copy constructor, must not copy empty TTree
120     TTree(const TTree<T>& aOtherTTree)
121     {
122         if (aOtherTTree.empty())
123         {
124             throw domain_error("Empty Tree.");
125         }
126
127         if (!aOtherTTree.empty())
128         {
129             fKey = aOtherTTree.fKey;
130
131             addRight(aOtherTTree.getRight());
132             addMiddle(aOtherTTree.getMiddle());
133             addLeft(aOtherTTree.getLeft());
134         }
135     }
136
137     // copy assignment operator, must not copy empty TTree
138     TTree<T>& operator=(const TTree<T>& aOtherTTree)
139     {
140         if (aOtherTTree.empty())
141         {
142             throw domain_error("Empty Tree.");
143         }
144
145         if (this != &aOtherTTree)
146         {
147             delete fLeft;
148             delete fMiddle;
149             delete fRight;
150
151             fKey = aOtherTTree.fKey;
152
153             addRight(aOtherTTree.getRight());
154             addMiddle(aOtherTTree.getMiddle());
155             addLeft(aOtherTTree.getLeft());
156
```

```
157         }
158         return *this;
159     }
160
161     // clone TTree, must not copy empty trees
162     TTree<T>* clone() const
163     {
164         if (!empty())
165         {
166             return new TTree(*this);
167         }
168     }
169
170 // Problem 3: TTree Move Semantics
171
172     // TTree r-value constructor
173     TTree(T&& aKey) :fKey(aKey), fLeft(&NIL), fMiddle(&NIL), fRight(&NIL)
174     {}
175
176     // move constructor, must not copy empty TTree
177     TTree(TTree<T>&& aOtherTTree)
178     {
179         if (aOtherTTree.empty())
180         {
181             throw domain_error("Empty Tree.");
182         }
183
184         if (this != &aOtherTTree)
185         {
186             fKey = aOtherTTree.fKey;
187             fRight = std::move((aOtherTTree.getRight()));
188             fMiddle = std::move((aOtherTTree.getMiddle()));
189             fLeft = std::move((aOtherTTree.getLeft()));
190
191         }
192         return *this;
193     }
194
195     // move assignment operator, must not copy empty TTree
196     TTree<T>& operator=(TTree<T>&& aOtherTTree)
197     {
198         if (aOtherTTree.empty())
199         {
200             throw domain_error("Empty Tree.");
201         }
202
203         if (this != &aOtherTTree)
204         {
205             delete fLeft;
206             delete fMiddle;
207             delete fRight;
208
```

```
209             fKey = aOtherTTree.fKey;
210
211             fRight = std::move((aOtherTTree.getRight()));
212             fMiddle = std::move((aOtherTTree.getMiddle()));
213             fLeft = std::move((aOtherTTree.getLeft()));
214
215         }
216         return *this;
217     }
218
219 // Problem 4: TTree Postfix Iterator
220
221     // return TTree iterator positioned at start
222     Iterator begin() const
223     {
224         return Iterator(&this);
225     }
226
227     // return TTree iterator positioned at end
228     Iterator end() const
229     {
230         return begin().end();
231     }
232 };
233
234 template<typename T>
235 TTree<T> TTree<T>::NIL;
236
```

```
 1
 2  // COS30008, Final Exam, 2021
 3
 4  #pragma once
 5
 6  #include "TTree.h"
 7
 8  #include <stack>
 9
10  template<typename S>
11  struct TTreeFrontier
12  {
13      size_t stage;                          // frontier stages: 0, 1, 2
14      const TTree<S>* node;                  // frontier TTree node
15
16      TTreeFrontier( const TTree<S>* aNode ) :
17          node(aNode),                       // TTree node
18          stage(0)                           // 0 - start right
19      {}
20  };
21
22  template<typename T>
23  class TTreePostfixIterator
24  {
25  private:
26      const TTree<T>* fTTree;                // 3-way tree
27      std::stack<TTreeFrontier<T>> fStack;   // DFS traversal stack
28
29      using Frontier = TTreeFrontier<T>;
30
31      // push subtree starting with aNode
32      void push_nodes(const TTree<T>* aNode)
33      {
34          fStack.push(aNode);
35      }
36
37   public:
38
39      using Iterator = TTreePostfixIterator<T>;
40
41      Iterator operator++(int)
42      {
43          Iterator old = *this;
44
45          ++(*this);
46
47          return old;
48      }
49
50      bool operator!=( const Iterator& aOtherIter ) const
51      {
52          return !(*this == aOtherIter);
```

```
53      }
54
55      // iterator constructor
56      TTreePostfixIterator(const TTree<T>* aTTree)
57      {
58          fTTree = aTTree;
59
60          constTTree<T>* lNode = fTTree;
61
62          push_nodes(lNode);
63
64          while (!lNode.empty())
65          {
66
67              if (!lNode.GetRight()->empty())
68                  push_nodes(lNode.GetRight());
69              if (!lNode.GetMiddle()->empty())
70                  push_nodes(lNode.GetMiddle());
71              if (!lNode.GetLeft()->empty())
72                  push_nodes(lNode.GetLeft());
73
74              lNode = lNode.GetLeft();
75          }
76
77      }
78
79      // iterator dereference
80      const T& operator*() const
81      {
82          return *(fStack.top());
83      }
84
85      // prefix increment
86      Iterator& operator++()
87      {
88          fStack.pop();
89          return *this;
90      }
91
92      // iterator equivalence
93      bool operator==(const Iterator& aOtherIter) const
94      {
95          return fTTree == aOtherIter.fTTree;
96      }
97
98      // auxiliaries
99      Iterator begin() const
100     {
101         return Iterator(fTTree);
102     }
103     Iterator end() const
104     {
```

```
105         Iterator iter = *this;
106
107         iter.fStack = std::stack<const TTree<T>*>();
108
109         return iter;
110     }
111 };
112
```