

Swinburne University of Technology

School of Science, Computing and Engineering Technologies

FINAL EXAM COVER SHEET

Subject Code: COS30008

Subject Title: Data Structures & Patterns

Due date: Nov 29, 2023

Lecturer: Dr. James Jackson

Your name: Tran Hoang Hai Anh **Your student id:** 104177513

Check Tutorial	Mon 10:30	Mon 14:30	Tues 08:30	Tues 10:30	Tues 12:30	Tues 14:30	Tues 16:30	Wed 08:30	Wed 10:30	Wed 12:30	Wed 14:30

Marker's comments:

Problem	Marks	Time Estimate in minutes	Obtained
1	132	30	
2	56	10	
3	60	15	
4	10+88=98	45	
5	50	20	
Total	396	120	

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

```

86 // Problem 1: TernaryTree Basic Infrastructure
87
88 private:
89
90 // remove a subtree, may throw a domain error [22]
91 const TTree& removeSubTree(size_t aSubtreeIndex)
92 {
93     if (aSubtreeIndex > 2)
94     {
95         throw out_of_range("Invalid aSubtree Index");
96     }
97     const TTree& subtree = *fSubTrees[aSubtreeIndex];
98     if (subtree.empty())
99     {
100         throw domain_error("Subtree is NIL");
101     }
102     fSubTrees[aSubtreeIndex] = &NIL;
103     return subtree;
104 };
105
106 // add a subtree; must avoid memory leaks; may throw domain error [18]
107 void addSubTree(size_t aSubtreeIndex, const TTree& aTree)
108 {
109     if (aSubtreeIndex > 2)
110     {
111         throw out_of_range("Invalid aSubtree Index");
112     }
113     if (!fSubTrees[aSubtreeIndex] -> empty())
114     {
115         throw domain_error("Subtree is not NIL");
116     }
117     fSubTrees[aSubtreeIndex] = const_cast<TTree*>(&aTree);
118 };
119
120 public:
121
122 // TernaryTree l-value constructor [18]
123 TernaryTree(const T& aKey) : fKey(aKey)
124 {
125     fill(fSubTrees, fSubTrees + 3, &NIL);
126 };
127
128 // destructor (free sub-trees, must not free empty trees) [14]
129 ~TernaryTree()
130 {
131     if (!empty())
132     {
133         for (size_t i = 0; i < 3; i++)

```

```

134 // TernaryTree l-value constructor [18]
135 TernaryTree(const T& aKey) : fKey(aKey)
136 {
137     fill(fSubTrees, fSubTrees + 3, &NIL);
138 };
139
140 // destructor (free sub-trees, must not free empty trees) [14]
141 ~TernaryTree()
142 {
143     if (!empty())
144     {
145         for (size_t i = 0; i < 3; i++)
146         {
147             if (!fSubTrees[i] -> empty())
148             {
149                 delete fSubTrees[i];
150             }
151         }
152     }
153 };
154
155 // return key value, may throw domain_error if empty [2]
156 const T& operator*( ) const
157 {
158     if (empty())
159     {
160         throw domain_error("Empty ternary tree is encountered");
161     }
162     else
163     {
164         return fKey;
165     }
166 };
167
168 // returns true if this ternary tree is empty [4]
169 bool empty() const
170 {
171     if (this == &NIL)
172     {
173         return true;
174     }
175     else
176     {
177         return false;
178     }
179 };
180
181 // returns true if this ternary tree is a leaf [10]
182 bool leaf() const
183 {
184     return all_of(fSubTrees, fSubTrees + 3, [](const TTree* subTree)

```

```

159     }
160     size_t fSubTreesHeight[3] = {};
161     for (size_t i = 0; i < 3; i++)
162     {
163         if (!fSubTrees[i] -> empty())
164         {
165             fSubTreesHeight[i] = fSubTrees[i] -> height();
166         }
167         else
168         {
169             fSubTreesHeight[i] = 0;
170         }
171     }
172     const auto maxHeight = *max_element(fSubTreesHeight, fSubTreesHeight + 3);
173     return maxHeight + 1;
174 };
175
176 // Problem 2: TernaryTree Copy Semantics
177
178 // copy constructor, must not copy empty ternary tree
179 TernaryTree(const TTree& aOtherTree)
180 {
181     for (size_t i = 0; i < 3; i++)
182     {
183         fSubTrees[i] = &NIL;
184     }
185     if (!aOtherTree.empty())
186     {
187         *this = aOtherTree;
188     }
189 };
190
191 // copy assignment operator, must not copy empty ternary tree
192 // may throw a domain error on attempts to copy NIL
193 TTree& operator=(const TTree& aOtherTTree)
194 {
195     if (this != &aOtherTTree)
196     {
197         if (aOtherTTree.empty())
198         {
199             throw domain_error("NIL as source not permitted.");
200         }
201         else
202         {
203             this->~TernaryTree();
204             fKey = aOtherTTree.fKey;
205             for (size_t i = 0; i < 3; i++)
206             {
207                 fSubTrees[i] = aOtherTTree.fSubTrees[i] -> empty() ? &NIL : aOtherTTree.fSubTrees[i] -> clone();
208             }
209         }
210     }
211 }

```

```

212 }
213
214 // Problem 4: TernaryTree Prefix Iterator
215
216 private:
217     // push subtree of aNode [30]
218     void push_subtrees(const TTree* aNode)
219     {
220         if (!aNode->getRight().empty()) fStack.push(const_cast<TTreeNode*>(&aNode->getRight()));
221         if (!aNode->getMiddle().empty()) fStack.push(const_cast<TTreeNode*>(&aNode->getMiddle()));
222         if (!aNode->getLeft().empty()) fStack.push(const_cast<TTreeNode*>(&aNode->getLeft()));
223     };
224
225 public:
226     // iterator constructor [12]
227     TernaryTreePrefixIterator(const TTree* aTTree) : fTTree(aTTree), fStack()
228     {
229         if (!fTTree->empty()) fStack.push(const_cast<TTreeNode*>(&fTTree));
230     };
231
232     // iterator dereference [8]
233     const T& operator*() const
234     {
235         return **fStack.top();
236     };
237
238     // prefix increment [12]
239     Iterator& operator++()
240     {
241         TTreeNode poppedNode = const_cast<TTreeNode*>(&fStack.top());
242         fStack.pop();
243         push_subtrees(&poppedNode);
244         return *this;
245     };
246
247     // iterator equivalence [12]
248     bool operator==(const Iterator& aOtherIter) const
249     {
250         if (fStack.size() == aOtherIter.fStack.size() && fTTree == aOtherIter.fTTree)
251         {
252             return true;
253         }
254         else
255         {
256             return false;
257         }
258     };
259 };

```