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: June 7, 2022, 18:00
Lecturer: Dr. Markus Lumpe

Your name: Ta Quang Tung **Your student id:** 104222196

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.

TernaryTree.h

```

// COS30008, Final Exam, 2022

#pragma once

#include <stdexcept>
#include <algorithm>

template<typename T>
class TernaryTreePrefixIterator;

template<typename T>
class TernaryTree
{
public:

    using TTree = TernaryTree<T>;
    using TSubTree = TTree*;

private:

    T fKey;
    TSubTree fSubTrees[3];

    // private default constructor used for declaration of NIL
    TernaryTree() :
        fKey(T())
    {
        for ( size_t i = 0; i < 3; i++ )
        {
            fSubTrees[i] = &NIL;
        }
    }

public:

    using Iterator = TernaryTreePrefixIterator<T>;

    static TTree NIL;           // sentinel

    // getters for subtrees
    const TTree& getLeft() const { return *fSubTrees[0]; }
    const TTree& getMiddle() const { return *fSubTrees[1]; }
    const TTree& getRight() const { return *fSubTrees[2]; }

    // add a subtree
    void addLeft( const TTree& aTTree ) { addSubTree( 0, aTTree ); }
    void addMiddle( const TTree& aTTree ) { addSubTree( 1, aTTree ); }
    void addRight( const TTree& aTTree ) { addSubTree( 2, aTTree ); }

    // remove a subtree, may through a domain error
    const TTree& removeLeft() { return removeSubTree( 0 ); }
    const TTree& removeMiddle() { return removeSubTree( 1 ); }
    const TTree& removeRight() { return removeSubTree( 2 ); }

////////////////////////////////////////////////////////////////////////
// Problem 1: TernaryTree Basic Infrastructure
```

```
58
59  private:
60
61      // remove a subtree, may throw a domain error [22]
62      const TTree& removeSubTree( size_t aSubtreeIndex ) {
63          if (aSubtreeIndex > 2) {
64              throw std::domain_error("Invalid subtree index!");
65          }
66
67          if (fSubTrees[aSubtreeIndex] == &NIL) {
68              throw std::domain_error("Subtree is NIL");
69          }
70
71          TSubTree tmp = fSubTrees[aSubtreeIndex];
72          fSubTrees[aSubtreeIndex] = &NIL;
73          return *tmp;
74      }
75
76      // add a subtree; must avoid memory leaks; may throw domain error [18]
77      void addSubTree( size_t aSubtreeIndex, const TTree& aTTree ) {
78          if (aSubtreeIndex > 2) {
79              throw std::domain_error("Invalid subtree index!");
80          }
81
82          if (fSubTrees[aSubtreeIndex] != &NIL) {
83              throw std::domain_error("Subtree is not NIL");
84          }
85
86          fSubTrees[aSubtreeIndex] = const_cast<TSubTree>(&aTTree);
87      }
88
89  public:
90
91      // TernaryTree l-value constructor [10]
92      TernaryTree( const T& aKey ) : fKey(aKey) {
93          for ( size_t i = 0; i < 3; i++ )
94          {
95              fSubTrees[i] = &NIL;
96          }
97      }
98
99      // destructor (free sub-trees, must not free empty trees) [14]
100     ~TernaryTree() {
101         if (!empty()) {
102             for (size_t i = 0; i < 3; i++) {
103                 if (fSubTrees[i] != &NIL) delete fSubTrees[i];
104             }
105         }
106     }
107
108     // return key value, may throw domain_error if empty [2]
109     const T& operator*() const {
110         if (empty()) {
111             throw std::domain_error("Tree is empty.");
112         }
113
114         return fKey;
115     }
116
117     // returns true if this ternary tree is empty [4]
```

```
118     bool empty() const {
119         return this == &NIL;
120     }
121 
122     // returns true if this ternary tree is a leaf [10]
123     bool leaf() const {
124         return fSubTrees[0]->empty() && fSubTrees[1]->empty() && fSubTrees[2]->empty();
125     }
126 
127     // return height of ternary tree, may throw domain_error if empty [48]
128     size_t height() const {
129         if (this == &NIL) {
130             throw std::domain_error("Operation not supported");
131         }
132 
133         int height = 0;
134 
135         if (!fSubTrees[0]->empty()) {
136             int maxLeft = fSubTrees[0]->height() + 1;
137             if (maxLeft > height) height = maxLeft;
138         }
139 
140         if (!fSubTrees[1]->empty()) {
141             int maxMid = fSubTrees[1]->height() + 1;
142             if (maxMid > height) height = maxMid;
143         }
144 
145         if (!fSubTrees[2]->empty()) {
146             int maxRight = fSubTrees[2]->height() + 1;
147             if (maxRight > height) height = maxRight;
148         }
149 
150         return height;
151     }
152 
153 ////////////////////////////////////////////////////////////////////////
154 // Problem 2: TernaryTree Copy Semantics
155 
156     // copy constructor, must not copy empty ternary tree
157     TernaryTree( const TTree& aOtherTTree ) {
158         if (aOtherTTree.empty()) {
159             throw std::domain_error("NIL as source not permitted");
160         }
161 
162         fKey = aOtherTTree.fKey;
163 
164         for ( size_t i = 0; i < 3; i++ )
165         {
166             fSubTrees[i] = &NIL;
167         }
168 
169         if (!aOtherTTree.fSubTrees[0]->empty()) {
170             fSubTrees[0] = aOtherTTree.fSubTrees[0]->clone();
171         }
172 
173         if (!aOtherTTree.fSubTrees[1]->empty()) {
174             fSubTrees[1] = aOtherTTree.fSubTrees[1]->clone();
175         }
176 
177         if (!aOtherTTree.fSubTrees[2]->empty()) {
```

```
178             fSubTrees[2] = aOtherTTree.fSubTrees[2]->clone();
179         }
180     }
181 
182     // copy assignment operator, must not copy empty ternary tree
183     // may throw a domain error on attempts to copy NIL
184     TTree& operator=( const TTree& aOtherTTree ) {
185         if (aOtherTTree.empty()) {
186             throw std::domain_error("NIL as source not permitted");
187         }
188 
189         if (this != &aOtherTTree) {
190             this->~TernaryTree();
191 
192             fKey = aOtherTTree.fKey;
193 
194             for ( size_t i = 0; i < 3; i++ )
195             {
196                 fSubTrees[i] = &NIL;
197             }
198 
199             if (!aOtherTTree.fSubTrees[0]->empty()) {
200                 fSubTrees[0] = aOtherTTree.fSubTrees[0]->clone();
201             }
202 
203             if (!aOtherTTree.fSubTrees[1]->empty()) {
204                 fSubTrees[1] = aOtherTTree.fSubTrees[1]->clone();
205             }
206 
207             if (!aOtherTTree.fSubTrees[2]->empty()) {
208                 fSubTrees[2] = aOtherTTree.fSubTrees[2]->clone();
209             }
210         }
211 
212         return *this;
213     }
214 
215     // clone ternary tree, must not copy empty trees
216     TSubTree clone() const {
217         if (empty()) return const_cast<TSubTree>(this);
218 
219         return new TTree(*this);
220     }
221 
222 ////////////////////////////////////////////////////////////////////////
223 // Problem 3: TernaryTree Move Semantics
224 
225     // TTree r-value constructor
226     TernaryTree( T&& aKey ) {
227         fKey = std::move(aKey);
228 
229         for ( size_t i = 0; i < 3; i++ )
230         {
231             fSubTrees[i] = &NIL;
232         }
233     }
234 
235     // move constructor, must not copy empty ternary tree
236     TernaryTree( TTree&& aOtherTTree ) {
237         if (aOtherTTree.empty()) {
```

```
238             throw std::domain_error("NIL as source not permitted");
239         }
240 
241         for ( size_t i = 0; i < 3; i++ )
242         {
243             fSubTrees[i] = &NIL;
244         }
245 
246         *this = std::move(aOtherTTree);
247     }
248 
249     // move assignment operator, must not copy empty ternary tree
250     TTree& operator=( TTree&& aOtherTTree ) {
251         if (aOtherTTree.empty()) {
252             throw std::domain_error("NIL as source not permitted");
253         }
254 
255         if (this != &aOtherTTree) {
256             this->~TernaryTree();
257 
258             for ( size_t i = 0; i < 3; i++ )
259             {
260                 fSubTrees[i] = &NIL;
261             }
262 
263             fKey = std::move(aOtherTTree.fKey);
264 
265             if (!aOtherTTree.fSubTrees[0]->empty()) {
266                 fSubTrees[0] = const_cast<TSubTree>(&aOtherTTree.removeLeft());
267             }
268 
269             if (!aOtherTTree.fSubTrees[1]->empty()) {
270                 fSubTrees[1] = const_cast<TSubTree>(&aOtherTTree.removeMiddle());
271             }
272 
273             if (!aOtherTTree.fSubTrees[2]->empty()) {
274                 fSubTrees[2] = const_cast<TSubTree>(&aOtherTTree.removeRight());
275             }
276         }
277 
278         return *this;
279     }
280 
281 ////////////////////////////////////////////////////////////////////////
282 // Problem 4: TernaryTree Prefix Iterator
283 
284     // return ternary tree prefix iterator positioned at start
285     Iterator begin() const {
286         return Iterator(this);
287     }
288 
289     // return ternary prefix iterator positioned at end
290     Iterator end() const {
291         return Iterator(this).end();
292     }
293 };
294 
295 template<typename T>
296 TernaryTree<T> TernaryTree<T>::NIL;
297 
```

TernaryTreePrefixIterator.h

```
1
2  // COS30008, Final Exam, 2022
3
4  #pragma once
5
6  #include "TernaryTree.h"
7
8  #include <stack>
9
10 template<typename T>
11 class TernaryTreePrefixIterator
12 {
13 private:
14     using TTree = TernaryTree<T>;
15     using TTreeNode = TTree*;
16     using TTreeStack = std::stack<const TTree*>;
17
18     const TTree* fTTree;                // ternary tree
19     TTreeStack fStack;                  // traversal stack
20
21 public:
22
23     using Iterator = TernaryTreePrefixIterator<T>;
24
25     Iterator operator++(int)
26     {
27         Iterator old = *this;
28
29         ++(*this);
30
31         return old;
32     }
33
34     bool operator!=( const Iterator& aOtherIter ) const
35     {
36         return !(*this == aOtherIter);
37     }
38
39 ////////////////////////////////////////////////////////////////////////
40 // Problem 4: TernaryTree Prefix Iterator
41
42 private:
43
44     // push subtree of aNode [30]
45     void push_subtrees( const TTree* aNode ) {
46         if (aNode->empty()) {
47             throw std::domain_error("Node is NIL");
48         }
49
50         if (!aNode->getRight().empty()) fStack.push(const_cast<const TTree*>(&aNode->
   getRight()));
51         if (!aNode->getMiddle().empty()) fStack.push(const_cast<const TTree*>(&aNode->
   getMiddle()));
52         if (!aNode->getLeft().empty()) fStack.push(const_cast<const TTree*>(&aNode->
   getLeft()));
53     }
54
55 public:
```

```
56
57      // iterator constructor [12]
58      TernaryTreePrefixIterator( const TTree* aTTree ) : fTTree(aTTree) {
59          fStack = TTreeStack();
60
61          if (fTTree != &TTree::NIL) {
62              fStack.push(fTTree);
63          }
64      }
65
66      // iterator dereference [8]
67      const T& operator*() const {
68          return **(fStack.top());
69      }
70
71      // prefix increment [12]
72      Iterator& operator++() {
73          TTree* node = const_cast<TTree*>(fStack.top());
74          fStack.pop();
75          push_subtrees(node);
76
77          return *this;
78      }
79
80      // iterator equivalence [12]
81      bool operator==( const Iterator& aOtherIter ) const {
82          return fTTree == aOtherIter.fTTree && fStack == aOtherIter.fStack;
83      }
84
85      // auxiliaries [4,10]
86      Iterator begin() const {
87          Iterator beginIterator = *this;
88
89          beginIterator.fStack = std::stack<const TTree*>();
90          if (beginIterator.fTTree != &TTree::NIL) {
91              beginIterator.fStack.push(beginIterator.fTTree);
92          }
93
94          return beginIterator;
95      }
96
97      Iterator end() const {
98          Iterator endIterator = *this;
99          endIterator.fStack = std::stack<const TTree*>();
100         return endIterator;
101     }
102 };
103
```

Main.cpp

```cpp

// COS30008, Final Exam, 2022

#include <iostream>
#include <string>
#include <stdexcept>

using namespace std;

#define P1
#define P2
#define P3
#define P4

// Test keys
string s1( "This" );
string s2( "is" );
string s3( "a" );
string s4( "ternary" );
string s5( "tree" );
string s6( "in" );
string s7( "action." );
string s8( "It" );
string s9( "works!" );

#ifdef P1

#include "TernaryTree.h"

void runP1()
{
    cout << "Test Problem 1:" << endl;

    using S3Tree = TernaryTree<string>;

    cout << "Setting up ternary tree..." << endl;

    S3Tree root( s1 );
    S3Tree* nA = new S3Tree( s2 );
    S3Tree* nB = new S3Tree( s5 );
    S3Tree* nC = new S3Tree( s7 );
    S3Tree nAA( s3 );
    S3Tree nAAC( s4 );
    S3Tree nBB( s6 );
    S3Tree nCB( s8 );
    S3Tree nCC( s9 );

    nAA.addRight( nAAC );
    nA->addLeft( nAA );

    nB->addMiddle( nBB );

    nC->addMiddle( nCB );
    nC->addRight( nCC );

    root.addLeft( *nA );
    root.addMiddle( *nB );
```

```
58      root.addRight( *nC );
59
60      try
61      {
62          root.addRight( *nC );
63
64          cerr << "Error: Non-empty subtree overridden." << endl;
65      }
66      catch (std::domain_error e)
67      {
68          cout << "Successfully caught: " << e.what() << endl;
69      }
70
71      cout << "Testing basic ternary tree logic ..." << endl;
72
73      cout << "Is NIL empty? " << (S3Tree::NIL.empty() ? "Yes" : "No") << endl;
74      cout << "Is root empty? " << (root.empty() ? "Yes" : "No") << endl;
75
76      try
77      {
78          cout << "Height of root is: " << root.height() << endl;
79
80          S3Tree::NIL.height();
81
82          cerr << "Error: NIL has no height." << endl;
83      }
84      catch (std::domain_error e)
85      {
86          cout << "Successfully caught: " << e.what() << endl;
87      }
88
89      cout << "Tearing down ternary tree..." << endl;
90
91      nC->removeRight();
92      nC->removeMiddle();
93      nB->removeMiddle();
94      nAA.removeRight();
95      nA->removeLeft();
96
97      try
98      {
99          nA->removeLeft();
100
101         cerr << "Error: Empty subtree removed." << endl;
102     }
103     catch (std::domain_error e)
104     {
105         cout << "Successfully caught: " << e.what() << endl;
106     }
107
108     cout << "Nodes nA, nB, nC get destroyed by destructor." << endl;
109
110     cout << "Test Problem 1 complete." << endl;
111 }
112
113 #endif
114
115 #ifdef P2
116
117 #include "TernaryTree.h"
```

```
118
119 void runP2()
120 {
121     cout << "Test Problem 2:" << endl;
122
123     using S3Tree = TernaryTree<string>;
124
125     S3Tree root( s1 );
126     S3Tree* nA = new S3Tree( s2 );
127     S3Tree* nB = new S3Tree( s5 );
128     S3Tree* nC = new S3Tree( s7 );
129     S3Tree* nAA = new S3Tree( s3 );
130     S3Tree* nAAC = new S3Tree( s4 );
131     S3Tree* nBB = new S3Tree( s6 );
132     S3Tree* nCB = new S3Tree( s8 );
133     S3Tree* nCC = new S3Tree( s9 );
134
135     nAA->addRight( *nAAC );
136     nA->addLeft( *nAA );
137
138     nB->addMiddle( *nBB );
139
140     nC->addMiddle( *nCB );
141     nC->addRight( *nCC );
142
143     root.addLeft( *nA );
144     root.addMiddle( *nB );
145     root.addRight( *nC );
146
147     S3Tree copy = root;
148
149     const S3Tree* lLeft;
150     const S3Tree* lRight;
151
152     lLeft = &copy.getLeft().getLeft().getRight();
153     lRight = &root.getLeft().getLeft().getRight();
154
155     if ( lLeft == lRight )
156     {
157         cerr << "Error: Shallow copy detected." << endl;
158     }
159     else
160     {
161         cout << "Copy constructor appears to work properly." << endl;
162     }
163
164     lLeft = &copy.getMiddle().getLeft();
165     lRight = &root.getMiddle().getRight();
166
167     if ( lLeft != lRight )
168     {
169         cerr << "Error: Copy does not preserve tree structure." << endl;
170     }
171     else
172     {
173         if ( !lLeft->empty() )
174         {
175             cerr << "Error: NIL not preserved." << endl;
176         }
177         else
```

```
178         {
179             cout << "Copy constructor preserves tree structure." << endl;
180         }
181     }
182 
183     root = copy;
184 
185     lLeft = &copy.getLeft().getLeft().getRight();
186     lRight = &root.getLeft().getLeft().getRight();
187 
188     if ( lLeft == lRight )
189     {
190         cerr << "Error: Shallow copy detected." << endl;
191     }
192     else
193     {
194         cout << "Assignment appears to work properly." << endl;
195     }
196 
197     lLeft = &copy.getMiddle().getLeft();
198     lRight = &root.getMiddle().getRight();
199 
200     if ( lLeft != lRight )
201     {
202         cerr << "Error: Assignment does not preserve tree structure." << endl;
203     }
204     else
205     {
206         if ( !lLeft->empty() )
207         {
208             cerr << "Error: NIL not preserved." << endl;
209         }
210         else
211         {
212             cout << "Assignment preserves tree structure." << endl;
213         }
214     }
215 
216     try
217     {
218         root = S3Tree::NIL;
219 
220         cerr << "Error: Copy of NIL! You should not see this message." << endl;
221     }
222     catch (domain_error e)
223     {
224         cout << "Successfully caught: " << e.what() << endl;
225     }
226 
227     S3Tree* clone = root.clone();
228 
229     lLeft = &clone->getLeft().getLeft().getRight();
230     lRight = &root.getLeft().getLeft().getRight();
231 
232     if ( lLeft == lRight )
233     {
234         cerr << "Error: Shallow copy detected." << endl;
235     }
236     else
237     {
```

```
238         cout << "Clone appears to work properly." << endl;
239     }
240 
241     delete clone;
242 
243     cout << "Trees root and copy get deleted next." << endl;
244     cout << "Test Problem 2 complete." << endl;
245 }
246 
247 #endif
248 
249 #ifdef P3
250 
251 #include "TernaryTree.h"
252 
253 void runP3()
254 {
255     cout << "Test Problem 3:" << endl;
256 
257     using S3Tree = TernaryTree<string>;
258 
259     S3Tree root( string( "This" ) );
260     S3Tree* nA = new S3Tree( s2 );
261     S3Tree* nB = new S3Tree( s5 );
262     S3Tree* nC = new S3Tree( s7 );
263     S3Tree* nAA = new S3Tree( s3 );
264     S3Tree* nAAC = new S3Tree( s4 );
265     S3Tree* nBB = new S3Tree( s6 );
266     S3Tree* nCB = new S3Tree( "It" );
267     S3Tree* nCC = new S3Tree( s9 );
268 
269     nAA->addRight( *nAAC );
270     nA->addLeft( *nAA );
271 
272     nB->addMiddle( *nBB );
273 
274     nC->addMiddle( *nCB );
275     nC->addRight( *nCC );
276 
277     root.addLeft( *nA );
278     root.addMiddle( *nB );
279     root.addRight( *nC );
280 
281     S3Tree copy = std::move(root);
282 
283     if ( root.leaf() )
284     {
285         cout << "std::move makes root a leaf node." << endl;
286     }
287     else
288     {
289         cerr << "Error: You should not see this message as root must become a leaf node."
    << endl;
290     }
291 
292     cout << "The payload of tree: " << *copy << endl;
293     cout << "The payload of tree.getLeft().getLeft().getRight():\t" << *copy.getLeft()
    .getLeft().getRight() << endl;
294     cout << "The payload of tree.getRight():\t" << *copy.getRight() << endl;
295 
```

```
296     root = std::move(copy);
297 
298     if ( copy.leaf() )
299     {
300         cout << "std::move makes copy a leaf node." << endl;
301     }
302     else
303     {
304         cerr << "Error: You should not see this message as copy must become a leaf node."
    << endl;
305     }
306 
307     cout << "The payload of tree: " << *root << endl;
308     cout << "The payload of tree.getLeft().getLeft().getRight():\t" << *root.getLeft()
    .getLeft().getRight() << endl;
309     cout << "The payload of tree.getRight():\t" << *root.getRight() << endl;
310 
311     try
312     {
313         root = std::move(S3Tree::NIL);
314 
315         cerr << "Error: Move of NIL! You should not see this message." << endl;
316     }
317     catch (domain_error e)
318     {
319         cout << "Successfully caught: " << e.what() << endl;
320     }
321 
322     cout << "Test Problem 3 complete." << endl;
323 }
324 
325 #endif
326 
327 #ifdef P4
328 
329 #include "TernaryTreePrefixIterator.h"
330 
331 void runP4()
332 {
333     cout << "Test Problem 4:" << endl;
334 
335     using S3Tree = TernaryTree<string>;
336 
337     S3Tree root( s1 );
338     S3Tree* nA = new S3Tree( s2 );
339     S3Tree* nB = new S3Tree( s5 );
340     S3Tree* nC = new S3Tree( s7 );
341     S3Tree* nAA = new S3Tree( s3 );
342     S3Tree* nAAC = new S3Tree( s4 );
343     S3Tree* nBB = new S3Tree( s6 );
344     S3Tree* nCB = new S3Tree( s8 );
345     S3Tree* nCC = new S3Tree( s9 );
346 
347     nAA->addRight( *nAAC );
348     nA->addLeft( *nAA );
349 
350     nB->addMiddle( *nBB );
351 
352     nC->addMiddle( *nCB );
353     nC->addRight( *nCC );
```

```
354
355     root.addLeft( *nA );
356     root.addMiddle( *nB );
357     root.addRight( *nC );
358
359     cout << "Test prefix iterator:";
360
361     for ( const string& k : root )
362     {
363         cout << ' ' << k;
364     }
365
366     cout << endl;
367
368     cout << "Test Problem 4 complete." << endl;
369 }
370
371 #endif
372
373 int main()
374 {
375 #ifdef P1
376     runP1();
377 #endif
378
379 #ifdef P2
380     runP2();
381 #endif
382
383 #ifdef P3
384     runP3();
385 #endif
386
387 #ifdef P4
388     runP4();
389 #endif
390
391     return 0;
392 }
393
394
```

Problem 5 (50 marks)

Answer the following questions in one or two sentences:

a. How can we construct a tree where all nodes have the same degree? [4]

5a)

We use an n-ary tree.

b. What is the difference between l-value and r-value references? [6]

5b)

l-values have defined storage locations whereas r-values do not.

c. What is a key concept of an abstract data types? [4]

5c)

An abstract data type is defined independently of its representation. In other words, its interface is separated from its actual implementation.

d. How do we define mutual dependent classes in C++? [4]

5d)

We use a forward declaration when defining mutal dependent classes.

e. What must a value-based data type define in C++? [2]

5e)

It must define the types of data it stores.

f. What is an object adapter? [6]

5f)

An object adapter converts the interface of an existing class into another interface that is compatible with other classes.

g. What is the difference between copy constructor and assignment operator and how do we guarantee safe operation? [8]

5g)

When a copy constructor is called, all instance variables are uninitialized. However, when an assignment operator is called, they are already initialized and care must be taken to release old memory and prevent accidental suicide (if an object is assigned to itself).

h. What is the best-case, average-case, and worse-case for a lookup in a binary tree? [6]

5h)

Best case is O(1) when the target element is at the root.
Average case is O(log n).
Worst case is O(n) when the tree is shaped like a linked list (for example: 2 3 4 5 6 7 where 2 is at the root, 3 is on the right of 2, 4 on the right of 3, etc.) and the target element is the last element (in the example worst case would be looking for 7)

i. What are reference data members and how do we initialize them? [2]

5i)

A reference data member contain a reference to another variable outside the object. They must be initialized with a member initializer, before the body of the constructor.

j. You are given 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]

5j)

Assuming the list of number is already sorted in ascending order and there are no repeated values:

If the list of numbers is already sorted in ascending order, we loop through it starting from the second number, comparing each number with the one before it. If current - prev != 1, the missing number is current - 1. At the end of the list, we have another check to see if current == n. If current != n, n is missing.

In the case where only one number k is given (n = 2), only a simple comparison k == 1 is needed. If it is true, the missing number is 2, otherwise it is 1.