

CSC 216 Portfolio 3

Nicolas Nytko

Dec 15, 2016

Contents

1	Paper	3
1.1	Perky Time Alotment [sic]	3
2	Homework	17
2.1	Search Trees	17
2.1.1	Problem R-10.19ae (0.5)	17
2.1.2	Problem R-10.24 (0.5)	17
2.1.3	Problem C-10.7 (1)	17
2.1.4	Problem C-10.12,13 (1.5)	17
2.1.5	Problem C-10.21 (1.5)	17
3	Labs	19
3.1	Bucket Sort	19
3.1.1	Compiler Environment	19
3.1.2	Source	19
3.1.3	Compiler Output	21
3.1.4	Program Output	21
3.2	Leprechaun Simulator	23
3.2.1	Compiler Environment	23
3.2.2	Source	23
3.2.3	Compiler Output	39
3.2.4	Program Output	39

1 Paper

1.1 Perky Time Alotment [sic]

Start with a background section to explain where PERT (Program Evaluation and Review Technique) came from. Next explain how PERT works and what it has to do with graphs and graph algorithms. Then expand to discuss how it is used today. Some suggestions are: construction, software development, etc. What does PERT have to do with CPM? What is CPM?

The following is a list of formatting requirements your paper(s) must follow:

- 2-5 pages:
 - single spaced with 35% (or less) of the 'body' space taken up by pictures/figures
 - double spaced with no pictures/figures
 - page count does NOT include a title page or list of references (both of which must be provided, however)
 - page count does NOT include a table of contents nor an index (either of which may be provided, if you wish)
- margins:
 - not more than 1.25 inches on a side
 - not less than 0.5 inches on a side
- text fonts:
 - something I can read!
 - not less than 8 point
 - not more than 12 point
- section/heading fonts:
 - something I can read!
 - not less than the text font
 - not more than 4 more than the text font
- style:
 - readable is preferred (spelling and grammar count!)
 - must have an introduction, body, and conclusion

The Pertinence of PERT

PERT And CPM And Why They *Pertain* to You

Nicolas Nytko

December 15, 2016

1 History and Background

When working on large-scale projects, it is important to be able to keep track of time and resources. Project management tools, notably PERT or CPM, were developed in order to keep track of these resources and to keep track of project milestones. Many times when working on large-scale projects, certain tasks must be done before others because they are dependent on each other, and these tasks or milestones must be done by a certain time or else the whole project must be forfeited.

Project management tools are not a modern concept, and some can be traced all the way back to ancient civilizations. Take a look at the ancient Egyptians and how they built the *Great Pyramid of Giza*. Spanning a 20 year building period, over two million blocks of stone, each weighing about 2 tons, were dragged in and placed to build the great pyramid. Looking at ancient records, archaeologists can infer that thousands of workers were managed by splitting them into four groups, one for each side of the pyramid. Sophisticated planning, management, and organization was required in order to find the correct stones, then cut, move, and set them into place.

A more recent example of project management was the creation of the *Gantt* chart by American mechanical engineers Henry Gantt and Frederick Taylor. A Gantt chart is a bar chart where activities are displayed by horizontal bars, each having a length proportional to approximately how long the item should take to complete. Gantt charts were first used in World War I in some famous projects at the time such as the Hoover Dam, and later the U.S. interstate highway network. They are still used today because they are simple and easy to understand by the entire workforce. However, one of the major shortcomings is that the relationships between activities and their dependencies are not shown on this chart, which is where PERT comes in.

The *program evaluation and review technique*, PERT for short, is a tool used

to analyze and represent the tasks and procedures needed to complete a program or project. PERT was originally developed by the United States Navy's Special Projects Office, along with Lockheed Missile Systems in the late 1950's to help measure and estimate progress for several missile projects, the most notable of which was the *UGM-27 Polaris* submarine-launched missile. PERT was designed to manage the over 3,000 contractors employed on the *Polaris* program by essentially providing a project road map that identified major milestones and how they were all dependent on each other.

One important thing to note was that the only constraint that PERT was created to deal with was time. Since this system was developed by the U.S. Navy, one could easily deduce why other factors such as cost or quality control were not factored in. The development of PERT was driven by a political need for the United States to compete with the Soviet Union during the cold war. PERT was used to ensure that the Polaris project was completed during a time when the United States Government was worried about the Soviet Union's increasing stockpile of nuclear arms.

2 How Does PERT Work?

PERT charts are used to schedule, organize, and manage tasks and milestones in a project or program. A PERT chart begins with one initial task or node that signifies the start of the project. From this node, arrows are drawn to other nodes and this depicts the sequence of tasks in the project. These tasks that are linked in order are called *dependent* or *serial* tasks. Concurrent sequences of tasks can be going on at the same time, these are called *parallel* or *concurrent* tasks. If a task has multiple arrows leading to it, then all those previous tasks must be completed before that task can be done, these are considered to have *task dependency*. Nodes

that have task dependency cannot be done before their dependent tasks.

Numbers are placed along the arrows to denote how much time is allotted to complete the task. For tasks that don't take any time to complete but must be done before others, they are often depicted with a dotted arrow line. These are called *dummy activities*. An example of a dummy activity in a software development scenario is when system files must be converted before more tasks can be completed, but relative to the project timeline the time needed to complete this task is negligible.

For each activity, three different time estimations must be defined in order to calculate the final estimation for the entire project. The first is the *optimistic time*, which is the minimum time required to accomplish an item or activity assuming that everything goes better than expected. The second one is the *pessimistic time*, which is the opposite of the optimistic time where the project is expected to go slower than usual; everything that can go wrong will go wrong in this estimation. It is the absolute maximum amount of time something should take. In this estimation, everything is assumed to go wrong except for major catastrophes. The third estimation is the *most likely time*, where it is the time required to go through a path assuming everything goes through normally. From these three estimations you can find the *expected time*, which accounts for the fact that some things don't always proceed normally. This is calculated by taking a weighted average of all three previous time estimations with the likely time estimation being 4 times more heavily averaged. This is based on an approximation of the *Beta distribution*.

$$E = \frac{T_{\text{optimistic}} + (4 \times T_{\text{likely}}) + T_{\text{pessimistic}}}{6}$$

Along with the expected time, something called the possible variance of this estimate

is also calculated.

$$V = \frac{(T_{pessimistic} - T_{likely})^2}{36}$$

To calculate the expected time of the full project, E and V are summed up for each project activity or item.

$$E_{project} = \sum_{k=1}^n E_k \quad V_{project} = \sum_{k=1}^n V_k$$

The sum of every E is the project expected time. The sum of every V is the variation of the entire project's expected time. The standard deviation can then be calculated as well, it is equal to the square root of the variance, \sqrt{V} .

A measure of excess time and resources on a project is called the *float*, or alternatively the *slack*. In a project, the amount of slack time is the amount of excess time that any particular item or activity can be delayed by and not affect subsequent tasks (free float), or affect the entire project (total float). A project that has positive float or positive slack would indicate that the project is ahead of schedule, negative slack would indicate behind schedule, and no slack would indicate that the project is simply on schedule.

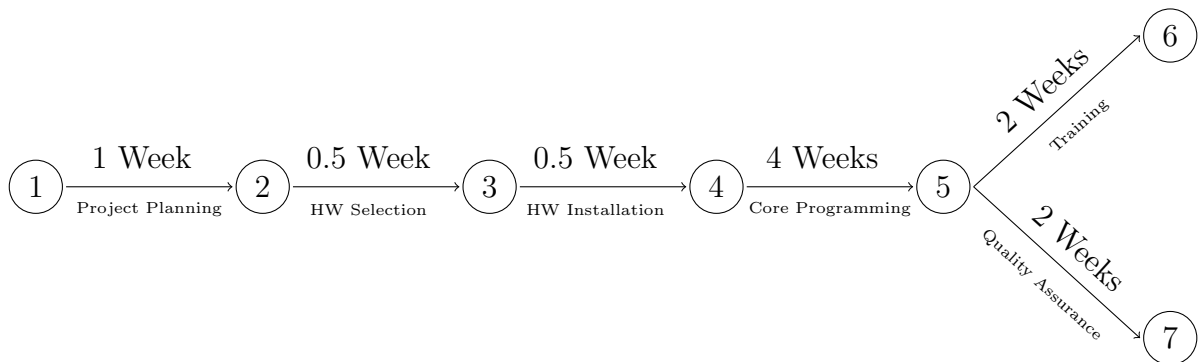
3 Graphing With PERT

Creating a PERT graph is not difficult, but there are some things that must be done before the graph is created. The first is to create a task list of each thing that must be done for the project. This list should describe things about each task, such as the approximate duration and which tasks (if any) that it is dependent on in order to be started. Here is an example of a task list table that can be created:

#	Task	Length	Dependence	Type
a.	Project Planning	1 Week	N/A	Sequential
b.	Hardware Selection	0.5 Week	a	Sequential
c.	Hardware Installation	0.5 Week	b	Sequential
d.	Core Systems Programming	4 Weeks	c	Sequential
e.	Core Systems Training	2 Weeks	d	Parallel
f.	Core Systems Quality Assurance	2 Weeks	d	Parallel

This example table has both sequential and parallel project tasks. This will be important when creating the PERT graph because it will be displayed as separate paths that can be taken. It is a good idea to make this task table as detailed as possible so nothing is forgotten when the graph is created.

To draw the graph, start with the first task that does not have any task dependency. This will be the start of your new PERT chart. To draw the task, pick an arbitrary shape to represent tasks/nodes. Good shapes to choose are circles, rectangles, or rounded rectangles. Bad shapes to choose are stars, flowers, the batman symbol, or anything overly complicated. Label this first node with a “1”, because it is your first task. After this first node, go down the list in order of dependency and draw each node with its number drawn inside of it. Once every task has been drawn, start connecting them in order with arrows. These arrows should be labeled with the activity name and duration.



The above illustration details the previously shown task chart converted into a PERT graph. It begins with the first node, which is project planning and lasts for 1 week. It then continues sequentially until the core programming node, where it branches into two parallel tasks. These two tasks can be done at the same time and are not dependent on each other to be completed.

Due to its relative simplicity, PERT graphs can be created using any graphing or image editing software, though specialized programs do exist. A quick google search shows that PERT graphing programs can range from being completely free to about \$350 USD. For those that are more technically inclined, graphs can be created using graphing programs or markup languages such as GraphViz, TikZ/PGF, etc. (The figure above was created with the TikZ package for LaTeX). Or if you're desperate enough, you could even just fire up a copy of MS Paint and start drawing it by hand.

4 CPM

Critical Path Method, CPM, is a project management technique similar to PERT that was also developed in the 1950's. CPM began development in 1956 by the *DuPont* chemical company and computing firm *Remington Rand Univac*. The precursors to CPM, however, were originally developed and practiced by DuPont as early as 1940 and helped contribute to the success of the *Manhattan Project*. Like PERT, it too was devised as a way to manage activity interrelationships in a project. The Critical Path Method was named after its usage of a *Critical Path*, a sequence of tasks or activities to be finished so that a project can be completed. Items or activities on the critical path cannot be done until previous activities have been completed.

CPM was first used in 1958 to construct a new DuPont chemical plant, and then

used again in 1959 to manage the shutdown of another DuPont plant. DuPont reportedly saved about 25% of the costs on shutdowns by using CPM by using it to efficiently plan shutdowns instead of flooding the project with labor. CPM was dropped by DuPont once the management team responsible for it was changed. Even though it was short-lived within DuPont, another company named Mauchly Associates started to use it. This company helped to commercialize CPM by having it focus on scheduling efficiencies rather than cost savings. Mauchly also conducted CPM training and transformed it into a new way of running businesses for construction companies. However, since computer systems were still relatively expensive back in the early 1960's, CPM had a high barrier of entry for many. It wasn't a coincidence that CPM started becoming widely popular once computers became small enough to fit on top of every office desk.

The generation of a CPM graph requires the differentiation between critical and non-critical activities. Critical activities are identified because if any of these activities are delayed, then the whole project suffers. The *Critical Path* in CPM follows this sequence of critical activities. The critical path is the absolutely most optimized path that goes through each critical activity and has the smallest time and resource expense.

CPM keeps track of both time and resources for each individual project item or activity. This allows for something called *project crashing*. Let's explain it with a simple analogy: imagine you are a project manager told to finish a certain project milestone in 7 weeks, and have a budget allotment of \$10,000. Later, you are told that the 7 weeks for the project will push back the project ship date, and must be done in 5 weeks. With this smaller amount of time, more resources must be dedicated to the project. Many times, this is done by modeling the increase with a simple linear relationship, however, it is up to the project manager to determine how much the

resources are increased by.

While CPM and PERT sound very similar, there are some key differences to keep in mind. When scheduling a project using, a PERT chart is created using uncertain activities, while a CPM chart is created using very well-defined activities. When using CPM, the durations of project items are very well defined and rigid, they cannot be changed and failure to stick to those deadlines may jeopardize the entire project. When looking at these factors, CPM can be considered to be deterministic, i.e., not accounting for any randomness or error. Therefore, CPM only gives one time estimate. PERT, on the other hand, is probabilistic and so gives multiple estimates: optimistic time, most likely time, and pessimistic time. An additional difference between PERT and CPM activities is that CPM will process critical and non-critical activities differently, while PERT does not differentiate between critical and non-critical activities and all tasks are assigned the same priority.

When scheduling a project using PERT, the only factor accounted for is the time duration of each activity or item. This is not true for CPM, which aims to minimize both the time used, but also to minimize the costs of each project item. As stated before, this is due to the developing companies of these task methods and what their requirements were at the times. Another interesting thing to note is that because PERT is milestone based, it is best used for projects where the activities or items are non-repetitive and do not keep showing up repeatedly. CPM is the opposite, and is best suited for repeated activities. Because of this, PERT is more suited for research and development projects, while CPM is best suited for large-scale construction projects where the same thing may be done over and over again.

5 PERT and CPM Today

An example of PERT/CPM usage today is in construction projects. Depending on the project's individual needs either PERT or CPM can be used. (However, more often than not CPM is used). Project times can be easily estimated based on previous knowledge and experience of other construction projects. Most construction project milestones and events are well-defined and have a rigid time duration, making CPM an excellent candidate for project management. For a construction project to be able to use either PERT or CPM, it must contain events that can be done independently of each other, and these events must have an order to them. Continuous processes such as several types of manufacturing or oil refining will not be able to be managed using PERT or CPM because they do not contain independent jobs to have done.

PERT is often used to schedule research and development projects. Since in these kinds of projects the nodes are not very well defined and time durations are only approximates, PERT will work well because it does not require extremely precise numbers, nor does it require extremely rigid deadlines. Researchers can set general project milestones given approximations for previous projects or from research and then use this to create a general time expectation for the project to be finished.

In software development, PERT is sometimes used alongside of Agile. Agile is a set of development principles that focuses on rapid development and feedback. In an agile development environment, work is divided into small parts and given to cross-functional teams that work on all aspects of the iteration: planning, analysis, development, etc. After each iteration, a working prototype is showed to shareholders or to upper management and feedback is recorded. This allows for minimal risk and easy adaptation because each small step must be scrutinized. In this case, PERT can be used to plan the entire project and divide it into subintervals. Each interval is an

activity on the PERT chart that takes up a duration of time, and some are dependent on others to be finished. PERT can be used to identify which parts of the project will take the most time, and then allow project managers to optimize that item.

6 Conclusion

PERT is a critical tool for project management and is very commonly used nowadays for all kinds of projects. It is very well suited for identifying key project milestones and relating them together in a specific order that maximizes project efficiency, this certain order being the critical path of the project. Using PERT, it is very easy to get high-accuracy project duration estimates and expensive software is not necessary to calculate it. It is simple to understand and can help efficiently manage projects and activities.

References

- [1] Stauber, B. Ralph et al. *Federal Statistical Activities*. The American Statistician, vol. 13, no. 2, 1959, pp. 9-12.
www.jstor.org/stable/2682310.
- [2] Rouse, Margaret. *PERT chart (Program Evaluation Review Technique)*. Search-SoftwareQuality. WhatIs.com, May 2007. Web. 01 Dec. 2016.
<http://searchsoftwarequality.techtarget.com/definition/PERT-chart>.
- [3] Mind Tools. *Critical Path Analysis and PERT Charts: Planning and Scheduling More Complex Projects*. Critical Path Analysis and PERT Charts. Mind Tools, n.d. Web. 01 Dec. 2016.
<https://www.mindtools.com/critpath.html>.
- [4] Kielmas, Maria. *History of the Critical Path Method*. History of the Critical Path Method. Chron, n.d. Web. 01 Dec. 2016.
<http://smallbusiness.chron.com/history-critical-path-method-55917.html>.
- [5] TutorialsPoint. *PERT Estimation Technique*. www.tutorialspoint.com. Tutorials Point, n.d. Web. 01 Dec. 2016.
https://www.tutorialspoint.com/management_concepts/pert_estimation_technique.htm.
- [6] S, Surbhi. *Difference Between PERT and CPM (with Comparison Chart) - Key Differences*. Key Differences. Key Differences, 27 Aug. 2016. Web. 1 Dec. 2016.
<http://keydifferences.com/difference-between-pert-and-cpm.html>.

- [7] Berman, Craig. *History of the Critical Path Method*. History of the Critical Path Method. Azcentral, n.d. Web. 1 Dec. 2016.
<http://yourbusiness.azcentral.com/history-critical-path-method-24351.html>.

2 Homework

2.1 Search Trees

2.1.1 Problem R-10.19ae (0.5)

Consider a tree T storing 100,000 entries. What is the worst-case height of T in the following cases?

1. T is an AVL tree.
2. T is a binary search tree.

The maximum height of an AVL tree is $2 \log n + 2 = 12$. The maximum height of a binary search tree is $O(n)$, $O(100,000) = 100,000$.

2.1.2 Problem R-10.24 (0.5)

Explain why you would get the same output in an inorder listing of the entries in a binary search tree, T , independent of whether T is maintained to be an AVL tree, splay tree, or red-black tree.

The whole purpose of a binary search tree is to have elements sorted so that an in-order traversal will output items in the same order every time. If a tree does not do this then it is not considered to be a binary search tree.

2.1.3 Problem C-10.7 (1)

If we maintain a reference to the position of the left-most internal node of an AVL tree, then operation `first` (Section 9.3) can be performed in $O(1)$ time. Describe how the implementation of the other map functions needs to be modified to maintain a reference to the left-most position.

Any function that changes the positioning of the nodes must also update the left-most node reference. The insert, delete, and update functions must update the reference. The insert and delete functions must check the position because something smaller than the left-most node can be inserted or the smallest node can be deleted. The update function must also set the node reference because the nodes may be swapped around and positions may be exchanged.

2.1.4 Problem C-10.12,13 (1.5)

Show that the nodes that become unbalanced in an AVL tree during an insert operation may be nonconsecutive on the path from the newly inserted node to the root. Show that at most one node in an AVL tree becomes unbalanced after operation `removeAboveExternal` is performed within the execution of a erase map operation.

The balance factor of an AVL tree is equal to $-h_{left} + h_{right}$. If a node is inserted such that its parent becomes balanced (or simply not unbalanced enough to get updated), a node higher up may become unbalanced because the left side of the tree becomes higher. This would be an example of a node on a nonconsecutive path becoming unbalanced. If a node is removed and its parent has only one child after that, then only the grandparent will become unbalanced.

2.1.5 Problem C-10.21 (1.5)

The mergeable heap ADT consists of operations `insert(k, x)`, `removeMin()`, `unionWith(h)`, and `min()`, where the `unionWith(h)` operation performs a union of the mergeable heap h with the present one, destroying the old versions of both. Describe a concrete implementation

of the mergeable heap ADT that achieves $O(\log n)$ performance for all its operations.

An $O(\log n)$ implementation of a mergeable heap can be achieved by using a *binomial heap*. A binomial heap is a collection of binomial trees, a binomial tree being defined by the following rules:

- A binomial tree with 0 order is just a single node
- A binomial tree with k order has k children, with those children being binomial trees of orders $k - 1, k - 2, \dots, 1, 0$. The children go in that order from left to right.

Merging a binomial tree of same order is very simple. Find the tree that is larger (or smaller, depending on the heap property). The other tree will become a child of that first tree, and the first tree will increase its order to $k + 1$. A binomial heap can have either 1 or 0 binomial trees of each order. These trees each satisfy the heap property.

Operation	Procedure	Complexity
<code>unionWith(h)</code>	Merge any binomial trees that have the same order. For any trees that don't match, just place them in the heap.	$O(\log n)$
<code>min()</code>	Search through each binomial tree and find the smallest root.	$O(\log n)$
<code>removeMin()</code>	Find the minimum element as before. Remove the whole tree from the heap. Transform the sub-binomial trees into a binomial heap and merge it back.	$O(\log n)$
<code>insert</code>	Create a separate heap with just the key. Merge this single heap into the heap.	$O(1)$

3 Labs

3.1 Bucket Sort

Name: Nicolas Nytko

Course: CSC216

Activity: Bucket Sort

Level: 2

Description: Design and implement a version of the bucket-sort algorithm for sorting a linked list of n entries (for instance, a list of type `std::list<int>`) with integer keys taken from the range $[0, N]$, for $N \geq 2$. The algorithm should run in $O(n + N)$ time.

3.1.1 Compiler Environment

Listing 1: environment

```
1 tex git:(master) pwd
2 /Users/nicolas/Git/portfolio2/tex
3 tex git:(master) uname -a
4 Darwin Nicolass-MacBook-Pro.local 16.1.0 Darwin Kernel Version 16.1.0: Thu Oct 13
   21:26:57 PDT 2016; root:xnu-3789.21.3~60/RELEASE_X86_64 x86_64
5 tex git:(master) clang --version
6 Apple LLVM version 8.0.0 (clang-800.0.42.1)
7 Target: x86_64-apple-darwin16.1.0
8 Thread model: posix
9 InstalledDir: /Library/Developer/CommandLineTools/usr/bin
10 tex git:(master) harper_cpp --version
11 This is harper_cpp version 1.221 executing under perl v5.18.2 and compiling with:
12
13 Configured with: --prefix=/Library/Developer/CommandLineTools/usr --with-gxx-include
   -dir=/usr/include/c++/4.2.1
14 Apple LLVM version 8.0.0 (clang-800.0.42.1)
15 Target: x86_64-apple-darwin16.1.0
16 Thread model: posix
17 InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

3.1.2 Source

Listing 2: ../lab/bucketsort/main.cpp

```
1 #include <iostream>
2 #include <list>
3 #include <stack>
4 #include <cstdio>
5
6 std::list<int> bucketSort( const std::list<int>& pList, int nMin, int nMax )
7 {
8     std::stack<int>* pBuckets = new std::stack<int>[static_cast<unsigned int>(nMax -
   nMin) + 1];
9     std::list<int> pReturnList;
10
11     for ( auto i = pList.begin(); i != pList.end(); i++ )
12     {
13         pBuckets[*i - nMin].push( *i );
14     }
15
16     for ( int i=nMin; i <= nMax; i++ )
```

```

17     {
18         while ( !pBuckets[i-nMin].empty( ) )
19         {
20             pReturnList.push_back( pBuckets[i-nMin].top( ) );
21             pBuckets[i-nMin].pop( );
22         }
23     }
24
25     delete [] pBuckets;
26
27     return pReturnList;
28 }
29
30 std::list<int> bucketSort( std::list<int>& pList )
31 {
32     int nMin = pList.front( ), nMax = pList.front( );
33
34     for ( auto i = ++pList.begin( ); i != pList.end( ); i++ )
35     {
36         if ( *i < nMin )
37         {
38             nMin = *i;
39         }
40
41         if ( *i > nMax )
42         {
43             nMax = *i;
44         }
45     }
46
47     return bucketSort( pList, nMin, nMax );
48 }
49
50 template<typename T>
51 void printList( const std::list<T>& pList )
52 {
53     std::cout << "Printing list of size " << pList.size( ) << std::endl;
54
55     for ( auto i = pList.begin( ); i != pList.end( ); i++ )
56     {
57         std::cout << *i << " ";
58     }
59
60     std::cout << std::endl;
61 }
62
63 int main( )
64 {
65     srand( static_cast<unsigned int>( time( nullptr ) ) );
66
67     std::list<int> pList;
68
69     for ( size_t i=0; i < 15; i++ )
70     {
71         pList.push_back( rand( ) % 50 );
72     }
73
74     std::cout << "Pre sorted list:" << std::endl;
75

```

```

76     printList( pList );
77     pList = bucketSort( pList );
78
79     std::cout << "After sorting: " << std::endl;
80     printList( pList );
81
82     return 0;
83 }

```

3.1.3 Compiler Output

Listing 3: ../lab/bucketsort/compilerout

```

1     bucketsort git:(master)    make CC=harper_cpp
2 harper_cpp -std=c++14 main.cpp -o bucket.out
3 main.cpp***

```

3.1.4 Program Output

Listing 4: ../lab/bucketsort/progout

```

1     bucketsort git:(master)    ./bucket.out
2 Pre sorted list:
3 Printing list of size 15
4 5 36 16 42 16 42 21 47 7 37 3 17 19 38 15
5 After sorting:
6 Printing list of size 15
7 3 5 7 15 16 16 17 19 21 36 37 38 42 42 47
8     bucketsort git:(master)    ./bucket.out
9 Pre sorted list:
10 Printing list of size 15
11 12 35 42 3 49 14 15 28 33 49 43 32 14 30 2
12 After sorting:
13 Printing list of size 15
14 2 3 12 14 14 15 28 30 32 33 35 42 43 49 49
15     bucketsort git:(master)    ./bucket.out
16 Pre sorted list:
17 Printing list of size 15
18 19 34 18 11 29 39 9 9 9 11 33 0 6 22 39
19 After sorting:
20 Printing list of size 15
21 0 6 9 9 9 11 11 18 19 22 29 33 34 39 39
22     bucketsort git:(master)    ./bucket.out
23 Pre sorted list:
24 Printing list of size 15
25 26 33 41 22 12 11 3 37 32 23 26 15 1 14 26
26 After sorting:
27 Printing list of size 15
28 1 3 11 12 14 15 22 23 26 26 26 32 33 37 41
29     bucketsort git:(master)    ./bucket.out
30 Pre sorted list:
31 Printing list of size 15
32 26 33 41 22 12 11 3 37 32 23 26 15 1 14 26
33 After sorting:
34 Printing list of size 15
35 1 3 11 12 14 15 22 23 26 26 26 32 33 37 41
36     bucketsort git:(master)    ./bucket.out

```

```

37 Pre sorted list:
38 Printing list of size 15
39 33 32 17 30 42 33 47 18 8 35 16 33 46 6 13
40 After sorting:
41 Printing list of size 15
42 6 8 13 16 17 18 30 32 33 33 33 35 42 46 47
43     bucketsort git:(master)      ./bucket.out
44 Pre sorted list:
45 Printing list of size 15
46 33 32 17 30 42 33 47 18 8 35 16 33 46 6 13
47 After sorting:
48 Printing list of size 15
49 6 8 13 16 17 18 30 32 33 33 33 35 42 46 47
50     bucketsort git:(master)      ./bucket.out
51 Pre sorted list:
52 Printing list of size 15
53 40 31 43 41 25 5 41 49 34 47 6 48 41 48 0
54 After sorting:
55 Printing list of size 15
56 0 5 6 25 31 34 40 41 41 41 43 47 48 48 49
57     bucketsort git:(master)      ./bucket.out
58 Pre sorted list:
59 Printing list of size 15
60 4 32 42 10 38 2 29 8 33 21 39 31 31 32 24
61 After sorting:
62 Printing list of size 15
63 2 4 8 10 21 24 29 31 31 32 32 33 38 39 42
64     bucketsort git:(master)      ./bucket.out
65 Pre sorted list:
66 Printing list of size 15
67 4 32 42 10 38 2 29 8 33 21 39 31 31 32 24
68 After sorting:
69 Printing list of size 15
70 2 4 8 10 21 24 29 31 31 32 32 33 38 39 42
71     bucketsort git:(master)      ./bucket.out
72 Pre sorted list:
73 Printing list of size 15
74 11 31 18 18 18 24 23 39 9 33 32 49 23 24 11
75 After sorting:
76 Printing list of size 15
77 9 11 11 18 18 18 23 23 24 24 31 32 33 39 49
78     bucketsort git:(master)      ./bucket.out
79 Pre sorted list:
80 Printing list of size 15
81 11 31 18 18 18 24 23 39 9 33 32 49 23 24 11
82 After sorting:
83 Printing list of size 15
84 9 11 11 18 18 18 23 23 24 24 31 32 33 39 49
85     bucketsort git:(master)      ./bucket.out
86 Pre sorted list:
87 Printing list of size 15
88 11 31 18 18 18 24 23 39 9 33 32 49 23 24 11
89 After sorting:
90 Printing list of size 15
91 9 11 11 18 18 18 23 23 24 24 31 32 33 39 49
92     bucketsort git:(master)      ./bucket.out
93 Pre sorted list:
94 Printing list of size 15
95 18 30 44 26 1 46 17 17 32 45 22 14 18 16 1

```

```

96 After sorting:
97 Printing list of size 15
98 1 1 14 16 17 17 18 18 22 26 30 32 44 45 46

```

3.2 Leprechaun Simulator

Name: Nicolas Nytko

Course: CSC216

Activity: Jumping Leprechauns

Level: 3

Description: Write a program that performs a simple n-body simulation, called Jumping Leprechauns. This simulation involves n leprechauns, numbered 1 to n . It maintains a gold value g_i for each leprechaun i , which begins with each leprechaun starting out with a million dollars worth of gold, that is, $g_i = 1000000$ for each $i = 1, 2, \dots, n$. In addition, the simulation also maintains, for each leprechaun, i , a place on the horizon, which is represented as a double-precision floating point number, x_i . In each iteration of the simulation, the simulation processes the leprechauns in order. Processing a leprechaun i during this iteration begins by computing a new place on the horizon for i , which is determined by the assignment

$$x_i \leftarrow x_i + r g_i$$

where r is a random floating-point number between 0 and 1. The leprechaun i then steals half the gold from the nearest leprechauns on either side of him and adds this gold to his gold value, g_i . Write a program that can perform a series of iterations in this simulation for a given number, n , of leprechauns. You must maintain the set of horizon positions using an ordered map data structure described in this chapter.

3.2.1 Compiler Environment

Listing 5: environment

```

1  tex git:(master)      pwd
2  /Users/nicolas/Git/portfolio2/tex
3  tex git:(master)      uname -a
4  Darwin Nicolass-MacBook-Pro.local 16.1.0 Darwin Kernel Version 16.1.0: Thu Oct 13
   21:26:57 PDT 2016; root:xnu-3789.21.3~60/RELEASE_ARM_T8020 x86_64
5  tex git:(master)      clang --version
6  Apple LLVM version 8.0.0 (clang-800.0.42.1)
7  Target: x86_64-apple-darwin16.1.0
8  Thread model: posix
9  InstalledDir: /Library/Developer/CommandLineTools/usr/bin
10 tex git:(master)      harper.cpp --version
11 This is harper.cpp version 1.221 executing under perl v5.18.2 and compiling with:
12
13 Configured with: --prefix=/Library/Developer/CommandLineTools/usr --with-gxx-include
   -dir=/usr/include/c++/4.2.1
14 Apple LLVM version 8.0.0 (clang-800.0.42.1)
15 Target: x86_64-apple-darwin16.1.0
16 Thread model: posix
17 InstalledDir: /Library/Developer/CommandLineTools/usr/bin

```

3.2.2 Source

Listing 6: ../lab/leprechaun/main.cpp

```

1  #include <iostream>
2  #include <cstdlib>
3  #include "orderedmap.hpp"
4
5  typedef double Position;
6  struct LeprechaunData
7  {
8      double nGold;
9      bool bIterated;
10 };
11 typedef OrderedMap<Position ,LeprechaunData> LepMap;
12 typedef MapKey<Position ,LeprechaunData> LepMapKey;
13
14 void iterateLeprechauns( LepMap& pMap )
15 {
16     LepMapKey* i = pMap.first( );
17
18     //for ( LepMapKey* i=pMap.first( ); i != nullptr; i = pMap.getNext( i ) )
19     while ( i != nullptr )
20     {
21         if ( i->nValue.bIterated == false )
22         {
23             double r = static_cast<double>( rand( ) % 2000 - 1000 ) / 1000.0;
24             i->nKey += r * i->nValue.nGold;
25
26             Position p = i->nKey;
27             LeprechaunData d = i->nValue;
28             d.bIterated = true;
29
30             pMap.remove( i );
31
32             auto* pHigher = pMap.getHigher( p );
33             auto* pLower = pMap.getLower( p );
34
35             if ( pHigher != nullptr )
36             {
37                 d.nGold += pHigher->nValue.nGold / 2;
38                 pHigher->nValue.nGold /= 2;
39             }
40
41             if ( pLower != nullptr )
42             {
43                 d.nGold += pLower->nValue.nGold / 2;
44                 pLower->nValue.nGold /= 2;
45             }
46
47             pMap.insert( p, d );
48             i = pMap.first( );
49         }
50         else
51         {
52             i = pMap.getNext( i );
53         }
54     }
55
56     for ( LepMapKey* i=pMap.first( ); i != nullptr; i = pMap.getNext( i ) )
57     {
58         i->nValue.bIterated = false;
59     }

```



```

60 }
61
62 void printLeprechauns( LepMap& pMap )
63 {
64     std::cout << "Contents of leprechaun map: " << std::endl;
65
66     int nIter = 1;
67     for ( LepMapKey* i=pMap.first( ); i != nullptr; i = pMap.getNext( i ) )
68     {
69         std::cout << nIter++ << " @ " << i->nKey << " w/ " << i->nValue.nGold << "
            gold" << std::endl;
70     }
71 }
72
73 int main( )
74 {
75     LepMap pLeprechauns;
76
77     std::srand( static_cast<unsigned int>( time( nullptr ) ) );
78     size_t nLeprechauns = std::rand( ) % 4 + 3;
79
80     std::cout << "There are " << nLeprechauns << " leprechauns." << std::endl;
81
82     for ( size_t i=0; i < nLeprechauns; i++ )
83     {
84         pLeprechauns.insert( std::rand( ) % 100, { 1000000, false } );
85     }
86
87     printLeprechauns( pLeprechauns );
88
89     for ( int i=0; i < 5; i++ )
90     {
91         iterateLeprechauns( pLeprechauns );
92     }
93
94     std::cout << "Iterated leprechauns 5 times" << std::endl;
95
96     printLeprechauns( pLeprechauns );
97 }

```

Listing 7: ../lab/leprechaun/btree.hpp

```

1  #ifndef BTREE_HPP
2  #define BTREE_HPP
3
4  #include <functional>
5
6  template<typename Data>
7  class BinaryTreeNode
8  {
9  private:
10     BinaryTreeNode<Data>* pLeft,* pRight,* pParent;
11     Data pData;
12
13 public:
14     /* big three */
15
16     BinaryTreeNode( ): pLeft( nullptr ), pRight( nullptr ), pParent( nullptr ),
        pData( ) { }

```

```

17
18 BinaryTreeNode( const Data& pDataNew ): pLeft( nullptr ), pRight( nullptr ),
    pParent( nullptr ), pData( pDataNew ) { }
19
20 BinaryTreeNode( const BinaryTreeNode& pNode ): pLeft( nullptr ), pRight( nullptr
    ), pParent( nullptr ), pData( )
21 {
22     operator=( pNode );
23 }
24
25 BinaryTreeNode& operator=( const BinaryTreeNode& pNode )
26 {
27     if ( pNode.pLeft != nullptr )
28     {
29         pLeft = new BinaryTreeNode( *pNode.pLeft );
30     }
31
32     if ( pNode.pRight != nullptr )
33     {
34         pRight = new BinaryTreeNode( *pNode.pRight );
35     }
36
37     pParent = pNode.pParent;
38
39     return *this;
40 }
41
42 ~BinaryTreeNode( )
43 {
44     if ( pLeft != nullptr )
45     {
46         delete pLeft;
47     }
48
49     if ( pRight != nullptr )
50     {
51         delete pRight;
52     }
53 }
54
55 Data& getData( )
56 {
57     return pData;
58 }
59
60 BinaryTreeNode*& getLeft( )
61 {
62     return pLeft;
63 }
64
65 BinaryTreeNode*& getRight( )
66 {
67     return pRight;
68 }
69
70 BinaryTreeNode* getLeftNode( )
71 {
72     return pLeft;
73 }

```

```

74
75 BinaryTreeNode* getRightNode( )
76 {
77     return pRight;
78 }
79
80 BinaryTreeNode* getParent( )
81 {
82     return pParent;
83 }
84
85 void setParent( BinaryTreeNode* pNewParent )
86 {
87     pParent = pNewParent;
88 }
89
90 bool isRightChild( )
91 {
92     if ( getParent( ) == nullptr )
93     {
94         return false;
95     }
96     else
97     {
98         if ( getParent( )->getRightNode( ) == this )
99         {
100             return true;
101         }
102         else
103         {
104             return false;
105         }
106     }
107 }
108
109 bool isLeftChild( )
110 {
111     return !isRightChild( );
112 }
113
114 Data* getDataPtr( )
115 {
116     return &pData;
117 }
118
119 Data setData( const Data& pDataNew )
120 {
121     pData = pDataNew;
122 }
123
124 /* create children */
125
126 BinaryTreeNode* createLeftNode( )
127 {
128     if ( pLeft )
129     {
130         delete pLeft;
131         pLeft = nullptr;
132     }

```

```

133
134     pLeft = new BinaryTreeNode;
135     pLeft->pParent = this;
136
137     return pLeft;
138 }
139
140 BinaryTreeNode* createLeftNode( const Data& pDataNew )
141 {
142     if ( pLeft )
143     {
144         delete pLeft;
145         pLeft = nullptr;
146     }
147
148     pLeft = new BinaryTreeNode( pDataNew );
149     pLeft->pParent = this;
150
151     return pLeft;
152 }
153
154 BinaryTreeNode* createRightNode( )
155 {
156     if ( pRight )
157     {
158         delete pRight;
159         pRight = nullptr;
160     }
161
162     pRight = new BinaryTreeNode;
163     pRight->pParent = this;
164
165     return pRight;
166 }
167
168 BinaryTreeNode* createRightNode( const Data& pDataNew )
169 {
170     if ( pRight )
171     {
172         delete pRight;
173         pRight = nullptr;
174     }
175
176     pRight = new BinaryTreeNode( pDataNew );
177     pRight->pParent = this;
178
179     return pRight;
180 }
181
182 void removeLeftNode( )
183 {
184     delete pLeft;
185     pLeft = nullptr;
186 }
187
188 void removeRightNode( )
189 {
190     delete pRight;
191     pRight = nullptr;

```

```

192     }
193
194     /* checks to see if children exist */
195
196     bool hasLeft( ) const
197     {
198         return ( pLeft ? true : false );
199     }
200
201     bool hasRight( ) const
202     {
203         return ( pRight ? true : false );
204     }
205
206     bool hasChildren( ) const
207     {
208         return ( ( pRight && pLeft ) ? true : false );
209     }
210
211     bool hasAnyChildren( ) const
212     {
213         return ( ( pRight || pLeft ) ? true : false );
214     }
215
216     /* traversals */
217
218     void preorderTraversal( const std::function<void(BinaryTreeNode<Data>*)>&
219                             pTraverseFunction )
220     {
221         pTraverseFunction( this );
222
223         if ( pLeft != nullptr )
224         {
225             pLeft->preorderTraversal( pTraverseFunction );
226         }
227
228         if ( pRight != nullptr )
229         {
230             pRight->preorderTraversal( pTraverseFunction );
231         }
232     }
233
234     void postorderTraversal( const std::function<void(BinaryTreeNode<Data>*)>&
235                             pTraverseFunction )
236     {
237         if ( pLeft != nullptr )
238         {
239             pLeft->postorderTraversal( pTraverseFunction );
240         }
241
242         if ( pRight != nullptr )
243         {
244             pRight->postorderTraversal( pTraverseFunction );
245         }
246
247         pTraverseFunction( this );
248     }
249
250     void inorderTraversal( const std::function<void(BinaryTreeNode<Data>*)>&

```

```

    pTraverseFunction )
249 {
250     if ( pLeft != nullptr )
251     {
252         pLeft->inorderTraversal( pTraverseFunction );
253     }
254
255     pTraverseFunction( this );
256
257     if ( pRight != nullptr )
258     {
259         pRight->inorderTraversal( pTraverseFunction );
260     }
261 }
262 };
263
264 template<typename Data>
265 class BinaryTree
266 {
267 protected:
268     BinaryTreeNode<Data>* pRoot;
269
270 public:
271     /* big three */
272
273     BinaryTree( ): pRoot( nullptr ) { }
274
275     BinaryTree( const BinaryTree& pTree ): pRoot( nullptr )
276     {
277         if ( pTree.pRoot != nullptr )
278         {
279             pRoot = new BinaryTreeNode<Data>( *pTree.pRoot );
280         }
281     }
282
283     BinaryTree& operator=( const BinaryTree& pTree )
284     {
285         if ( pRoot != nullptr )
286         {
287             delete pRoot;
288             pRoot = nullptr;
289         }
290
291         if ( pTree.pRoot != nullptr )
292         {
293             pRoot = new BinaryTreeNode<Data>( *pTree.pRoot );
294         }
295
296         return *this;
297     }
298
299     ~BinaryTree( )
300     {
301         if ( pRoot != nullptr )
302         {
303             delete pRoot;
304         }
305     }
306

```

```

307 BinaryTreeNode<Data>*& getRoot( )
308 {
309     return pRoot;
310 }
311
312 BinaryTreeNode<Data>* getRoot( ) const
313 {
314     return pRoot;
315 }
316
317 BinaryTreeNode<Data>* createRoot( )
318 {
319     if ( pRoot )
320     {
321         delete pRoot;
322         pRoot = nullptr;
323     }
324
325     pRoot = new BinaryTreeNode<Data>;
326
327     return pRoot;
328 }
329
330 BinaryTreeNode<Data>* createRoot( const Data& pData )
331 {
332     if ( pRoot )
333     {
334         delete pRoot;
335         pRoot = nullptr;
336     }
337
338     pRoot = new BinaryTreeNode<Data>( pData );
339
340     return pRoot;
341 }
342
343 void deleteRoot( )
344 {
345     delete pRoot;
346 }
347
348 void preorderTraversal( const std::function<void(BinaryTreeNode<Data>*)>&
349     pTraverseFunction )
350 {
351     if ( pRoot != nullptr )
352     {
353         pRoot->preorderTraversal( pTraverseFunction );
354     }
355 }
356
357 void postorderTraversal( const std::function<void(BinaryTreeNode<Data>*)>&
358     pTraverseFunction )
359 {
360     if ( pRoot != nullptr )
361     {
362         pRoot->postorderTraversal( pTraverseFunction );
363     }
364 }

```

```

364     void inorderTraversal( const std::function<void(BinaryTreeNode<Data>*)>&
                           pTraverseFunction )
365     {
366         if ( pRoot != nullptr )
367         {
368             pRoot->inorderTraversal( pTraverseFunction );
369         }
370     }
371 };
372
373 #endif

```

Listing 8: ../lab/leprechaun/orderedmap.hpp

```

1  #ifndef ORDERED_MAP_HPP
2  #define ORDERED_MAP_HPP
3
4  #include "btree.hpp"
5
6  template<typename Key, typename Value>
7  struct MapKey
8  {
9      Key nKey;
10     Value nValue;
11
12     BinaryTreeNode< struct MapKey<Key, Value> >* pNode;
13 };
14
15 template<typename Key, typename Value>
16 class OrderedMap
17 {
18 private:
19     typedef MapKey<Key, Value> MapNode;
20     BinaryTree<MapNode> pSearchTree;
21
22     MapNode* getMaxUnderValue( const Key& nKey, BinaryTreeNode<MapNode>* pNode )
23     {
24         if ( nKey <= pNode->getData( ).nKey &&
25             pNode->hasLeft( ) )
26         {
27             return getMaxUnderValue( nKey, pNode->getLeftNode( ) );
28         }
29         if ( nKey > pNode->getData( ).nKey &&
30             pNode->hasRight( ) )
31         {
32             MapNode* pNodeTemp =
33                 getMaxUnderValue( nKey, pNode->getRightNode( ) );
34
35             if ( pNodeTemp != nullptr )
36             {
37                 Key nTemp = pNodeTemp->nKey;
38
39                 return ( nTemp > pNode->getDataPtr( )->nKey ) ? pNodeTemp : pNode->
                     getDataPtr( );
40             }
41             else
42             {
43                 return pNode->getDataPtr( );
44             }
45         }
46     }
47
48     MapNode* insert( const Key& nKey, const Value& nValue )
49     {
50         MapNode* newNode = new MapNode( nKey, nValue );
51         BinaryTreeNode<MapNode>* pNewNode = new BinaryTreeNode<MapNode>( newNode );
52         pNewNode->hasLeft( ) = false;
53         pNewNode->hasRight( ) = false;
54         pNewNode->dataPtr = newNode;
55         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
56         if ( pSearchNode == nullptr )
57         {
58             pSearchTree->setRootNode( pNewNode );
59             return newNode;
60         }
61         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
62         while ( true )
63         {
64             if ( nKey < pCurrentNode->getData( ).nKey )
65             {
66                 if ( pCurrentNode->hasLeft( ) )
67                     pCurrentNode = pCurrentNode->getLeftNode( );
68                 else
69                     pCurrentNode->hasLeft( ) = true;
70             }
71             else if ( nKey > pCurrentNode->getData( ).nKey )
72             {
73                 if ( pCurrentNode->hasRight( ) )
74                     pCurrentNode = pCurrentNode->getRightNode( );
75                 else
76                     pCurrentNode->hasRight( ) = true;
77             }
78             else
79                 return pCurrentNode->dataPtr;
80         }
81         pCurrentNode->insert( nKey, nValue );
82         return newNode;
83     }
84
85     void erase( const Key& nKey )
86     {
87         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
88         if ( pSearchNode == nullptr )
89             return;
90         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
91         while ( true )
92         {
93             if ( nKey < pCurrentNode->getData( ).nKey )
94             {
95                 if ( pCurrentNode->hasLeft( ) )
96                     pCurrentNode = pCurrentNode->getLeftNode( );
97                 else
98                     return;
99             }
100            else if ( nKey > pCurrentNode->getData( ).nKey )
101            {
102                if ( pCurrentNode->hasRight( ) )
103                    pCurrentNode = pCurrentNode->getRightNode( );
104                else
105                    return;
106            }
107            else
108            {
109                BinaryTreeNode<MapNode>* pLeftNode = pCurrentNode->getLeftNode( );
110                BinaryTreeNode<MapNode>* pRightNode = pCurrentNode->getRightNode( );
111                MapNode* pNewNode = new MapNode( pCurrentNode->dataPtr->nKey,
112                                                  pCurrentNode->dataPtr->nValue );
113                BinaryTreeNode<MapNode>* pNewNodeTree = new BinaryTreeNode<MapNode>( pNewNode );
114                pNewNodeTree->hasLeft( ) = false;
115                pNewNodeTree->hasRight( ) = false;
116                pNewNodeTree->dataPtr = pNewNode;
117                if ( pLeftNode )
118                    pNewNodeTree->hasLeft( ) = true;
119                if ( pRightNode )
120                    pNewNodeTree->hasRight( ) = true;
121                pCurrentNode->dataPtr = pNewNode;
122                pCurrentNode->setLeftNode( pLeftNode );
123                pCurrentNode->setRightNode( pRightNode );
124                return;
125            }
126        }
127    }
128
129     void clear()
130     {
131         pSearchTree->clear();
132     }
133
134     const Key& getKey() const
135     {
136         return pCurrentNode->getData( ).nKey;
137     }
138
139     const Value& getValue() const
140     {
141         return pCurrentNode->getData( ).nValue;
142     }
143
144     void setKey( const Key& nKey )
145     {
146         pCurrentNode->dataPtr->nKey = nKey;
147     }
148
149     void setValue( const Value& nValue )
150     {
151         pCurrentNode->dataPtr->nValue = nValue;
152     }
153
154     void print() const
155     {
156         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
157         if ( pSearchNode == nullptr )
158             return;
159         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
160         while ( true )
161         {
162             if ( pCurrentNode->hasLeft( ) )
163                 pCurrentNode = pCurrentNode->getLeftNode( );
164             else
165                 break;
166         }
167         while ( true )
168         {
169             pCurrentNode->dataPtr->print();
170             if ( pCurrentNode->hasRight( ) )
171                 pCurrentNode = pCurrentNode->getRightNode( );
172             else
173                 break;
174         }
175     }
176
177     void printTree() const
178     {
179         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
180         if ( pSearchNode == nullptr )
181             return;
182         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
183         while ( true )
184         {
185             if ( pCurrentNode->hasLeft( ) )
186                 pCurrentNode = pCurrentNode->getLeftNode( );
187             else
188                 break;
189         }
190         while ( true )
191         {
192             pCurrentNode->dataPtr->print();
193             if ( pCurrentNode->hasRight( ) )
194                 pCurrentNode = pCurrentNode->getRightNode( );
195             else
196                 break;
197         }
198     }
199
200     void printMap() const
201     {
202         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
203         if ( pSearchNode == nullptr )
204             return;
205         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
206         while ( true )
207         {
208             if ( pCurrentNode->hasLeft( ) )
209                 pCurrentNode = pCurrentNode->getLeftNode( );
210             else
211                 break;
212         }
213         while ( true )
214         {
215             pCurrentNode->dataPtr->print();
216             if ( pCurrentNode->hasRight( ) )
217                 pCurrentNode = pCurrentNode->getRightNode( );
218             else
219                 break;
220         }
221     }
222
223     void printMapTree() const
224     {
225         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
226         if ( pSearchNode == nullptr )
227             return;
228         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
229         while ( true )
230         {
231             if ( pCurrentNode->hasLeft( ) )
232                 pCurrentNode = pCurrentNode->getLeftNode( );
233             else
234                 break;
235         }
236         while ( true )
237         {
238             pCurrentNode->dataPtr->print();
239             if ( pCurrentNode->hasRight( ) )
240                 pCurrentNode = pCurrentNode->getRightNode( );
241             else
242                 break;
243         }
244     }
245
246     void printMapTree2() const
247     {
248         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
249         if ( pSearchNode == nullptr )
250             return;
251         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
252         while ( true )
253         {
254             if ( pCurrentNode->hasLeft( ) )
255                 pCurrentNode = pCurrentNode->getLeftNode( );
256             else
257                 break;
258         }
259         while ( true )
260         {
261             pCurrentNode->dataPtr->print();
262             if ( pCurrentNode->hasRight( ) )
263                 pCurrentNode = pCurrentNode->getRightNode( );
264             else
265                 break;
266         }
267     }
268
269     void printMapTree3() const
270     {
271         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
272         if ( pSearchNode == nullptr )
273             return;
274         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
275         while ( true )
276         {
277             if ( pCurrentNode->hasLeft( ) )
278                 pCurrentNode = pCurrentNode->getLeftNode( );
279             else
280                 break;
281         }
282         while ( true )
283         {
284             pCurrentNode->dataPtr->print();
285             if ( pCurrentNode->hasRight( ) )
286                 pCurrentNode = pCurrentNode->getRightNode( );
287             else
288                 break;
289         }
290     }
291
292     void printMapTree4() const
293     {
294         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
295         if ( pSearchNode == nullptr )
296             return;
297         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
298         while ( true )
299         {
300             if ( pCurrentNode->hasLeft( ) )
301                 pCurrentNode = pCurrentNode->getLeftNode( );
302             else
303                 break;
304         }
305         while ( true )
306         {
307             pCurrentNode->dataPtr->print();
308             if ( pCurrentNode->hasRight( ) )
309                 pCurrentNode = pCurrentNode->getRightNode( );
310             else
311                 break;
312         }
313     }
314
315     void printMapTree5() const
316     {
317         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
318         if ( pSearchNode == nullptr )
319             return;
320         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
321         while ( true )
322         {
323             if ( pCurrentNode->hasLeft( ) )
324                 pCurrentNode = pCurrentNode->getLeftNode( );
325             else
326                 break;
327         }
328         while ( true )
329         {
330             pCurrentNode->dataPtr->print();
331             if ( pCurrentNode->hasRight( ) )
332                 pCurrentNode = pCurrentNode->getRightNode( );
333             else
334                 break;
335         }
336     }
337
338     void printMapTree6() const
339     {
340         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
341         if ( pSearchNode == nullptr )
342             return;
343         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
344         while ( true )
345         {
346             if ( pCurrentNode->hasLeft( ) )
347                 pCurrentNode = pCurrentNode->getLeftNode( );
348             else
349                 break;
350         }
351         while ( true )
352         {
353             pCurrentNode->dataPtr->print();
354             if ( pCurrentNode->hasRight( ) )
355                 pCurrentNode = pCurrentNode->getRightNode( );
356             else
357                 break;
358         }
359     }
360
361     void printMapTree7() const
362     {
363         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
364         if ( pSearchNode == nullptr )
365             return;
366         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
367         while ( true )
368         {
369             if ( pCurrentNode->hasLeft( ) )
370                 pCurrentNode = pCurrentNode->getLeftNode( );
371             else
372                 break;
373         }
374         while ( true )
375         {
376             pCurrentNode->dataPtr->print();
377             if ( pCurrentNode->hasRight( ) )
378                 pCurrentNode = pCurrentNode->getRightNode( );
379             else
380                 break;
381         }
382     }
383
384     void printMapTree8() const
385     {
386         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
387         if ( pSearchNode == nullptr )
388             return;
389         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
390         while ( true )
391         {
392             if ( pCurrentNode->hasLeft( ) )
393                 pCurrentNode = pCurrentNode->getLeftNode( );
394             else
395                 break;
396         }
397         while ( true )
398         {
399             pCurrentNode->dataPtr->print();
400             if ( pCurrentNode->hasRight( ) )
401                 pCurrentNode = pCurrentNode->getRightNode( );
402             else
403                 break;
404         }
405     }
406
407     void printMapTree9() const
408     {
409         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
410         if ( pSearchNode == nullptr )
411             return;
412         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
413         while ( true )
414         {
415             if ( pCurrentNode->hasLeft( ) )
416                 pCurrentNode = pCurrentNode->getLeftNode( );
417             else
418                 break;
419         }
420         while ( true )
421         {
422             pCurrentNode->dataPtr->print();
423             if ( pCurrentNode->hasRight( ) )
424                 pCurrentNode = pCurrentNode->getRightNode( );
425             else
426                 break;
427         }
428     }
429
430     void printMapTree10() const
431     {
432         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
433         if ( pSearchNode == nullptr )
434             return;
435         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
436         while ( true )
437         {
438             if ( pCurrentNode->hasLeft( ) )
439                 pCurrentNode = pCurrentNode->getLeftNode( );
440             else
441                 break;
442         }
443         while ( true )
444         {
445             pCurrentNode->dataPtr->print();
446             if ( pCurrentNode->hasRight( ) )
447                 pCurrentNode = pCurrentNode->getRightNode( );
448             else
449                 break;
450         }
451     }
452
453     void printMapTree11() const
454     {
455         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
456         if ( pSearchNode == nullptr )
457             return;
458         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
459         while ( true )
460         {
461             if ( pCurrentNode->hasLeft( ) )
462                 pCurrentNode = pCurrentNode->getLeftNode( );
463             else
464                 break;
465         }
466         while ( true )
467         {
468             pCurrentNode->dataPtr->print();
469             if ( pCurrentNode->hasRight( ) )
470                 pCurrentNode = pCurrentNode->getRightNode( );
471             else
472                 break;
473         }
474     }
475
476     void printMapTree12() const
477     {
478         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
479         if ( pSearchNode == nullptr )
480             return;
481         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
482         while ( true )
483         {
484             if ( pCurrentNode->hasLeft( ) )
485                 pCurrentNode = pCurrentNode->getLeftNode( );
486             else
487                 break;
488         }
489         while ( true )
490         {
491             pCurrentNode->dataPtr->print();
492             if ( pCurrentNode->hasRight( ) )
493                 pCurrentNode = pCurrentNode->getRightNode( );
494             else
495                 break;
496         }
497     }
498
499     void printMapTree13() const
500     {
501         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
502         if ( pSearchNode == nullptr )
503             return;
504         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
505         while ( true )
506         {
507             if ( pCurrentNode->hasLeft( ) )
508                 pCurrentNode = pCurrentNode->getLeftNode( );
509             else
510                 break;
511         }
512         while ( true )
513         {
514             pCurrentNode->dataPtr->print();
515             if ( pCurrentNode->hasRight( ) )
516                 pCurrentNode = pCurrentNode->getRightNode( );
517             else
518                 break;
519         }
520     }
521
522     void printMapTree14() const
523     {
524         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
525         if ( pSearchNode == nullptr )
526             return;
527         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
528         while ( true )
529         {
530             if ( pCurrentNode->hasLeft( ) )
531                 pCurrentNode = pCurrentNode->getLeftNode( );
532             else
533                 break;
534         }
535         while ( true )
536         {
537             pCurrentNode->dataPtr->print();
538             if ( pCurrentNode->hasRight( ) )
539                 pCurrentNode = pCurrentNode->getRightNode( );
540             else
541                 break;
542         }
543     }
544
545     void printMapTree15() const
546     {
547         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
548         if ( pSearchNode == nullptr )
549             return;
550         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
551         while ( true )
552         {
553             if ( pCurrentNode->hasLeft( ) )
554                 pCurrentNode = pCurrentNode->getLeftNode( );
555             else
556                 break;
557         }
558         while ( true )
559         {
560             pCurrentNode->dataPtr->print();
561             if ( pCurrentNode->hasRight( ) )
562                 pCurrentNode = pCurrentNode->getRightNode( );
563             else
564                 break;
565         }
566     }
567
568     void printMapTree16() const
569     {
570         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
571         if ( pSearchNode == nullptr )
572             return;
573         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
574         while ( true )
575         {
576             if ( pCurrentNode->hasLeft( ) )
577                 pCurrentNode = pCurrentNode->getLeftNode( );
578             else
579                 break;
580         }
581         while ( true )
582         {
583             pCurrentNode->dataPtr->print();
584             if ( pCurrentNode->hasRight( ) )
585                 pCurrentNode = pCurrentNode->getRightNode( );
586             else
587                 break;
588         }
589     }
590
591     void printMapTree17() const
592     {
593         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
594         if ( pSearchNode == nullptr )
595             return;
596         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
597         while ( true )
598         {
599             if ( pCurrentNode->hasLeft( ) )
600                 pCurrentNode = pCurrentNode->getLeftNode( );
601             else
602                 break;
603         }
604         while ( true )
605         {
606             pCurrentNode->dataPtr->print();
607             if ( pCurrentNode->hasRight( ) )
608                 pCurrentNode = pCurrentNode->getRightNode( );
609             else
610                 break;
611         }
612     }
613
614     void printMapTree18() const
615     {
616         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
617         if ( pSearchNode == nullptr )
618             return;
619         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
620         while ( true )
621         {
622             if ( pCurrentNode->hasLeft( ) )
623                 pCurrentNode = pCurrentNode->getLeftNode( );
624             else
625                 break;
626         }
627         while ( true )
628         {
629             pCurrentNode->dataPtr->print();
630             if ( pCurrentNode->hasRight( ) )
631                 pCurrentNode = pCurrentNode->getRightNode( );
632             else
633                 break;
634         }
635     }
636
637     void printMapTree19() const
638     {
639         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
640         if ( pSearchNode == nullptr )
641             return;
642         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
643         while ( true )
644         {
645             if ( pCurrentNode->hasLeft( ) )
646                 pCurrentNode = pCurrentNode->getLeftNode( );
647             else
648                 break;
649         }
650         while ( true )
651         {
652             pCurrentNode->dataPtr->print();
653             if ( pCurrentNode->hasRight( ) )
654                 pCurrentNode = pCurrentNode->getRightNode( );
655             else
656                 break;
657         }
658     }
659
660     void printMapTree20() const
661     {
662         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
663         if ( pSearchNode == nullptr )
664             return;
665         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
666         while ( true )
667         {
668             if ( pCurrentNode->hasLeft( ) )
669                 pCurrentNode = pCurrentNode->getLeftNode( );
670             else
671                 break;
672         }
673         while ( true )
674         {
675             pCurrentNode->dataPtr->print();
676             if ( pCurrentNode->hasRight( ) )
677                 pCurrentNode = pCurrentNode->getRightNode( );
678             else
679                 break;
680         }
681     }
682
683     void printMapTree21() const
684     {
685         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
686         if ( pSearchNode == nullptr )
687             return;
688         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
689         while ( true )
690         {
691             if ( pCurrentNode->hasLeft( ) )
692                 pCurrentNode = pCurrentNode->getLeftNode( );
693             else
694                 break;
695         }
696         while ( true )
697         {
698             pCurrentNode->dataPtr->print();
699             if ( pCurrentNode->hasRight( ) )
700                 pCurrentNode = pCurrentNode->getRightNode( );
701             else
702                 break;
703         }
704     }
705
706     void printMapTree22() const
707     {
708         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
709         if ( pSearchNode == nullptr )
710             return;
711         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
712         while ( true )
713         {
714             if ( pCurrentNode->hasLeft( ) )
715                 pCurrentNode = pCurrentNode->getLeftNode( );
716             else
717                 break;
718         }
719         while ( true )
720         {
721             pCurrentNode->dataPtr->print();
722             if ( pCurrentNode->hasRight( ) )
723                 pCurrentNode = pCurrentNode->getRightNode( );
724             else
725                 break;
726         }
727     }
728
729     void printMapTree23() const
730     {
731         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
732         if ( pSearchNode == nullptr )
733             return;
734         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
735         while ( true )
736         {
737             if ( pCurrentNode->hasLeft( ) )
738                 pCurrentNode = pCurrentNode->getLeftNode( );
739             else
740                 break;
741         }
742         while ( true )
743         {
744             pCurrentNode->dataPtr->print();
745             if ( pCurrentNode->hasRight( ) )
746                 pCurrentNode = pCurrentNode->getRightNode( );
747             else
748                 break;
749         }
750     }
751
752     void printMapTree24() const
753     {
754         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
755         if ( pSearchNode == nullptr )
756             return;
757         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
758         while ( true )
759         {
760             if ( pCurrentNode->hasLeft( ) )
761                 pCurrentNode = pCurrentNode->getLeftNode( );
762             else
763                 break;
764         }
765         while ( true )
766         {
767             pCurrentNode->dataPtr->print();
768             if ( pCurrentNode->hasRight( ) )
769                 pCurrentNode = pCurrentNode->getRightNode( );
770             else
771                 break;
772         }
773     }
774
775     void printMapTree25() const
776     {
777         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
778         if ( pSearchNode == nullptr )
779             return;
780         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
781         while ( true )
782         {
783             if ( pCurrentNode->hasLeft( ) )
784                 pCurrentNode = pCurrentNode->getLeftNode( );
785             else
786                 break;
787         }
788         while ( true )
789         {
790             pCurrentNode->dataPtr->print();
791             if ( pCurrentNode->hasRight( ) )
792                 pCurrentNode = pCurrentNode->getRightNode( );
793             else
794                 break;
795         }
796     }
797
798     void printMapTree26() const
799     {
800         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
801         if ( pSearchNode == nullptr )
802             return;
803         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
804         while ( true )
805         {
806             if ( pCurrentNode->hasLeft( ) )
807                 pCurrentNode = pCurrentNode->getLeftNode( );
808             else
809                 break;
810         }
811         while ( true )
812         {
813             pCurrentNode->dataPtr->print();
814             if ( pCurrentNode->hasRight( ) )
815                 pCurrentNode = pCurrentNode->getRightNode( );
816             else
817                 break;
818         }
819     }
820
821     void printMapTree27() const
822     {
823         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
824         if ( pSearchNode == nullptr )
825             return;
826         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
827         while ( true )
828         {
829             if ( pCurrentNode->hasLeft( ) )
830                 pCurrentNode = pCurrentNode->getLeftNode( );
831             else
832                 break;
833         }
834         while ( true )
835         {
836             pCurrentNode->dataPtr->print();
837             if ( pCurrentNode->hasRight( ) )
838                 pCurrentNode = pCurrentNode->getRightNode( );
839             else
840                 break;
841         }
842     }
843
844     void printMapTree28() const
845     {
846         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
847         if ( pSearchNode == nullptr )
848             return;
849         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
850         while ( true )
851         {
852             if ( pCurrentNode->hasLeft( ) )
853                 pCurrentNode = pCurrentNode->getLeftNode( );
854             else
855                 break;
856         }
857         while ( true )
858         {
859             pCurrentNode->dataPtr->print();
860             if ( pCurrentNode->hasRight( ) )
861                 pCurrentNode = pCurrentNode->getRightNode( );
862             else
863                 break;
864         }
865     }
866
867     void printMapTree29() const
868     {
869         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
870         if ( pSearchNode == nullptr )
871             return;
872         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
873         while ( true )
874         {
875             if ( pCurrentNode->hasLeft( ) )
876                 pCurrentNode = pCurrentNode->getLeftNode( );
877             else
878                 break;
879         }
880         while ( true )
881         {
882             pCurrentNode->dataPtr->print();
883             if ( pCurrentNode->hasRight( ) )
884                 pCurrentNode = pCurrentNode->getRightNode( );
885             else
886                 break;
887         }
888     }
889
890     void printMapTree30() const
891     {
892         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
893         if ( pSearchNode == nullptr )
894             return;
895         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
896         while ( true )
897         {
898             if ( pCurrentNode->hasLeft( ) )
899                 pCurrentNode = pCurrentNode->getLeftNode( );
900             else
901                 break;
902         }
903         while ( true )
904         {
905             pCurrentNode->dataPtr->print();
906             if ( pCurrentNode->hasRight( ) )
907                 pCurrentNode = pCurrentNode->getRightNode( );
908             else
909                 break;
910         }
911     }
912
913     void printMapTree31() const
914     {
915         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
916         if ( pSearchNode == nullptr )
917             return;
918         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
919         while ( true )
920         {
921             if ( pCurrentNode->hasLeft( ) )
922                 pCurrentNode = pCurrentNode->getLeftNode( );
923             else
924                 break;
925         }
926         while ( true )
927         {
928             pCurrentNode->dataPtr->print();
929             if ( pCurrentNode->hasRight( ) )
930                 pCurrentNode = pCurrentNode->getRightNode( );
931             else
932                 break;
933         }
934     }
935
936     void printMapTree32() const
937     {
938         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
939         if ( pSearchNode == nullptr )
940             return;
941         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
942         while ( true )
943         {
944             if ( pCurrentNode->hasLeft( ) )
945                 pCurrentNode = pCurrentNode->getLeftNode( );
946             else
947                 break;
948         }
949         while ( true )
950         {
951             pCurrentNode->dataPtr->print();
952             if ( pCurrentNode->hasRight( ) )
953                 pCurrentNode = pCurrentNode->getRightNode( );
954             else
955                 break;
956         }
957     }
958
959     void printMapTree33() const
960     {
961         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
962         if ( pSearchNode == nullptr )
963             return;
964         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
965         while ( true )
966         {
967             if ( pCurrentNode->hasLeft( ) )
968                 pCurrentNode = pCurrentNode->getLeftNode( );
969             else
970                 break;
971         }
972         while ( true )
973         {
974             pCurrentNode->dataPtr->print();
975             if ( pCurrentNode->hasRight( ) )
976                 pCurrentNode = pCurrentNode->getRightNode( );
977             else
978                 break;
979         }
980     }
981
982     void printMapTree34() const
983     {
984         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
985         if ( pSearchNode == nullptr )
986             return;
987         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
988         while ( true )
989         {
990             if ( pCurrentNode->hasLeft( ) )
991                 pCurrentNode = pCurrentNode->getLeftNode( );
992             else
993                 break;
994         }
995         while ( true )
996         {
997             pCurrentNode->dataPtr->print();
998             if ( pCurrentNode->hasRight( ) )
999                 pCurrentNode = pCurrentNode->getRightNode( );
1000             else
1001                 break;
1002         }
1003     }
1004
1005     void printMapTree35() const
1006     {
1007         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
1008         if ( pSearchNode == nullptr )
1009             return;
1010         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
1011         while ( true )
1012         {
1013             if ( pCurrentNode->hasLeft( ) )
1014                 pCurrentNode = pCurrentNode->getLeftNode( );
1015             else
1016                 break;
1017         }
1018         while ( true )
1019         {
1020             pCurrentNode->dataPtr->print();
1021             if ( pCurrentNode->hasRight( ) )
1022                 pCurrentNode = pCurrentNode->getRightNode( );
1023             else
1024                 break;
1025         }
1026     }
1027
1028     void printMapTree36() const
1029     {
1030         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
1031         if ( pSearchNode == nullptr )
1032             return;
1033         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
1034         while ( true )
1035         {
1036             if ( pCurrentNode->hasLeft( ) )
1037                 pCurrentNode = pCurrentNode->getLeftNode( );
1038             else
1039                 break;
1040         }
1041         while ( true )
1042         {
1043             pCurrentNode->dataPtr->print();
1044             if ( pCurrentNode->hasRight( ) )
1045                 pCurrentNode = pCurrentNode->getRightNode( );
1046             else
1047                 break;
1048         }
1049     }
1050
1051     void printMapTree37() const
1052     {
1053         BinaryTreeNode<MapNode>* pSearchNode = pSearchTree->getRootNode( );
1054         if ( pSearchNode == nullptr )
1055             return;
1056         BinaryTreeNode<MapNode>* pCurrentNode = pSearchNode;
1057         while ( true )
1058         {
1059             if ( pCurrentNode->hasLeft( ) )
1060                 pCurrentNode = pCurrentNode->getLeftNode( );
1061             else
1062                 break;
1063         }
1064         while ( true )
1065         {
1066             pCurrentNode->dataPtr->print();
1067             if ( pCurrentNode->hasRight( ) )
1068                 pCurrentNode = pCurrentNode->getRightNode( );
1069             else
1070                 break;
1071         }
1072     }
1073
1074     void printMapTree38()
```



```

45     }
46     else if ( nKey > pNode->getDataPtr( )->nKey )
47     {
48         return pNode->getDataPtr( );
49     }
50
51     return nullptr;
52 }
53
54 MapNode* getMinOverValue( const Key& nKey, BinaryTreeNode<MapNode>* pNode )
55 {
56     if ( nKey >= pNode->getData( ).nKey &&
57         pNode->hasRight( ) )
58     {
59         return getMinOverValue( nKey, pNode->getRightNode( ) );
60     }
61     if ( nKey < pNode->getData( ).nKey &&
62         pNode->hasLeft( ) )
63     {
64         MapNode* pNodeTemp =
65             getMinOverValue( nKey, pNode->getLeftNode( ) );
66
67         if ( pNodeTemp != nullptr )
68         {
69             Key nTemp = pNodeTemp->nKey;
70
71             return ( nTemp < pNode->getDataPtr( )->nKey ) ? pNodeTemp : pNode->
                getDataPtr( );
72         }
73         else
74         {
75             return pNode->getDataPtr( );
76         }
77     }
78     else if ( nKey < pNode->getDataPtr( )->nKey )
79     {
80         return pNode->getDataPtr( );
81     }
82
83     return nullptr;
84 }
85
86 public:
87     OrderedMap( ): pSearchTree( ) { }
88     OrderedMap( const OrderedMap& pMap ): pSearchTree( pMap.pSearchTree ) { }
89     OrderedMap& operator=( const OrderedMap& pMap )
90     {
91         pSearchTree = pMap.pSearchTree;
92     }
93
94     /* Returns a reference to the smallest key value */
95
96     MapNode* first( )
97     {
98         if ( pSearchTree.getRoot( ) == nullptr )
99         {
100             throw std::runtime_error( "Calling first() when map is empty." );
101         }
102

```

```

103     BinaryTreeNode<MapNode>* pLeft = pSearchTree.getRoot( );
104
105     while ( pLeft->getLeftNode( ) != nullptr )
106     {
107         pLeft = pLeft->getLeftNode( );
108     }
109
110     return pLeft->getDataPtr( );
111 }
112
113 /* Returns a reference to the largest key value */
114
115 MapNode* last( )
116 {
117     if ( pSearchTree.getRoot( ) == nullptr )
118     {
119         throw std::runtime_error( "Calling last() when map is empty." );
120         return nullptr;
121     }
122
123     BinaryTreeNode<MapNode>* pRight = pSearchTree.getRoot( );
124
125     while ( pRight->getRightNode( ) != nullptr )
126     {
127         pRight = pRight->getRightNode( );
128     }
129
130     return pRight->getDataPtr( );
131 }
132
133 MapNode* find( const Key& nKey )
134 {
135     if ( pSearchTree.getRoot( ) == nullptr )
136     {
137         throw std::runtime_error( "Calling find() when map is empty." );
138         return nullptr;
139     }
140
141     BinaryTreeNode<MapNode>* pCurrent = pSearchTree.getRoot( );
142     bool bLooping = true;
143
144     while ( bLooping && pCurrent->getDataPtr( )->nKey != nKey )
145     {
146         if ( pCurrent->hasRight( ) )
147         {
148             if ( nKey > pCurrent->getRightNode( )->getDataPtr( )->nKey )
149             {
150                 pCurrent = pCurrent->getRightNode( );
151             }
152             else
153             {
154                 bLooping = false;
155             }
156         }
157         else if ( pCurrent->hasLeft( ) )
158         {
159             if ( nKey < pCurrent->getLeftNode( )->getDataPtr( )->nKey )
160             {
161                 pCurrent = pCurrent->getLeftNode( );

```

```

162         }
163         else
164         {
165             bLooping = false;
166         }
167     }
168 }
169
170 if ( pCurrent->getDataPtr( )->nKey == nKey )
171 {
172     return pCurrent->getDataPtr( );
173 }
174 else
175 {
176     return nullptr;
177 }
178 }
179
180 /* Returns the key that has the greatest value less than pKey */
181
182 MapNode* getLower( const Key& nKey )
183 {
184     return getMaxUnderValue( nKey, pSearchTree.getRoot( ) );
185 }
186
187 /* Returns the key that has the smallest value greater than pKey */
188
189 MapNode* getHigher( const Key& nKey )
190 {
191     return getMinOverValue( nKey, pSearchTree.getRoot( ) );
192 }
193
194 MapNode* getNext( const MapNode* pKey )
195 {
196     if ( pKey == last( ) )
197     {
198         return nullptr;
199     }
200
201     BinaryTreeNode<MapNode>* pCurrent = pKey->pNode;
202
203     if ( pCurrent->getParent( ) != nullptr )
204     {
205         if ( pCurrent->hasRight( ) )
206         {
207             /* If we are a middle node, go to the right. */
208
209             pCurrent = pCurrent->getRightNode( );
210
211             while ( pCurrent->hasLeft( ) )
212             {
213                 pCurrent = pCurrent->getLeftNode( );
214             }
215         }
216         else if ( pCurrent->getParent( )->getLeftNode( ) == pCurrent )
217         {
218             /* If we are a left node, go up to the middle node. */
219
220             pCurrent = pCurrent->getParent( );

```

```

221     }
222     else
223     {
224         /* If we are a right node, we have to traverse the tree
225            until we get to a suitable middle node */
226
227         while ( pKey->nKey >
228                pCurrent->getParent( )->getData( ).nKey )
229         {
230             pCurrent = pCurrent->getParent( );
231         }
232
233         pCurrent = pCurrent->getParent( );
234     }
235 }
236 else
237 {
238     if ( pCurrent->hasRight( ) )
239     {
240         pCurrent = pCurrent->getRightNode( );
241
242         while ( pCurrent->hasLeft( ) )
243         {
244             pCurrent = pCurrent->getLeftNode( );
245         }
246     }
247     else
248     {
249         return nullptr;
250     }
251 }
252
253 return pCurrent->getDataPtr( );
254 }
255
256 MapNode* insert( const Key& nKey, const Value& nValue )
257 {
258     BinaryTreeNode<MapNode>* pNode;
259
260     if ( pSearchTree.getRoot( ) == nullptr )
261     {
262         /* If our search tree is empty just stick it at the top */
263
264         pNode = pSearchTree.createRoot( );
265     }
266     else
267     {
268         /* Otherwise loop through and find a good place to put it. */
269
270         BinaryTreeNode<MapNode>* pCurrent = pSearchTree.getRoot( );
271         bool bLooping = true;
272
273         while ( bLooping )
274         {
275             if ( nKey < pCurrent->getDataPtr( )->nKey )
276             {
277                 /* Go left if smaller */
278
279                 if ( pCurrent->hasLeft( ) )

```

```

280         {
281             pCurrent = pCurrent->getLeftNode( );
282         }
283         else
284         {
285             pNode = pCurrent->createLeftNode( );
286             bLooping = false;
287         }
288     }
289     else if ( nKey >= pCurrent->getDataPtr( )->nKey )
290     {
291         /* Go right if larger */
292
293         if ( pCurrent->hasRight( ) )
294         {
295             pCurrent = pCurrent->getRightNode( );
296         }
297         else
298         {
299             pNode = pCurrent->createRightNode( );
300             bLooping = false;
301         }
302     }
303 }
304
305 MapNode* pData = pNode->getDataPtr( );
306 pData->pNode = pNode;
307 pData->nKey = nKey;
308 pData->nValue = nValue;
309
310 return pData;
311 }
312
313 void remove( MapNode* pNode )
314 {
315     if ( pNode == nullptr )
316     {
317         return;
318     }
319
320     BinaryTreeNode<MapNode>* pTreeNode = pNode->pNode;
321
322     if ( !pNode->pNode->hasAnyChildren( ) )
323     {
324         /* No children */
325
326         if ( pTreeNode->getParent( ) != nullptr )
327         {
328             if ( pTreeNode->isRightChild( ) )
329             {
330                 pTreeNode->getParent( )->removeRightNode( );
331             }
332             else
333             {
334                 pTreeNode->getParent( )->removeLeftNode( );
335             }
336         }
337     }
338     else if ( pSearchTree.getRoot( ) == pTreeNode )

```

```

339         {
340             pSearchTree.deleteRoot( );
341         }
342     }
343     else if ( !pTreeNode->hasChildren( ) && pTreeNode->hasAnyChildren( ) )
344     {
345         /* Only one child */
346
347         BinaryTreeNode<MapNode>* pChild;
348
349         if ( pTreeNode->hasLeft( ) )
350         {
351             pChild = pTreeNode->getLeft( );
352         }
353         else
354         {
355             pChild = pTreeNode->getRight( );
356         }
357
358         if ( pTreeNode->getParent( ) != nullptr )
359         {
360             if ( pTreeNode->isRightChild( ) )
361             {
362                 pTreeNode->getParent( )->getRight( ) = pChild;
363             }
364             else
365             {
366                 pTreeNode->getParent( )->getLeft( ) = pChild;
367             }
368
369             pChild->setParent( pTreeNode->getParent( ) );
370         }
371         else
372         {
373             pSearchTree.getRoot( ) = pChild;
374             pChild->setParent( nullptr );
375         }
376
377         pTreeNode->getLeft( ) = nullptr;
378         pTreeNode->getRight( ) = nullptr;
379         delete pTreeNode;
380     }
381     else
382     {
383         /* Has both children
384            Find the smallest node on the right side */
385
386         BinaryTreeNode<MapNode>* pSmallest = pTreeNode->getRight( );
387
388         while ( pSmallest->hasLeft( ) )
389         {
390             pSmallest = pSmallest->getLeft( );
391         }
392
393         pTreeNode->getData( ) = pSmallest->getData( );
394         pTreeNode->getData( ).pNode = pTreeNode;
395
396         remove( pSmallest->getDataPtr( ) );
397     }

```

```

398     }
399 };
400
401 #endif

```

3.2.3 Compiler Output

Listing 9: ../lab/leprechaun/compilerout

```

1      leprechaun git:(master)      make CC=harper_cpp
2 harper_cpp -std=c++14 main.cpp -o leprechaun.out
3 main.cpp***

```

3.2.4 Program Output

Listing 10: ../lab/leprechaun/progout

```

1      leprechaun git:(master)      ./leprechaun.out
2 There are 4 leprechauns.
3 Contents of leprechaun map:
4 1 @ 11 w/ 1e+06 gold
5 2 @ 23 w/ 1e+06 gold
6 3 @ 29 w/ 1e+06 gold
7 4 @ 32 w/ 1e+06 gold
8 Iterated leprechauns 5 times
9 Contents of leprechaun map:
10 1 @ -1.1785e+06 w/ 894958 gold
11 2 @ 206579 w/ 947357 gold
12 3 @ 333358 w/ 962387 gold
13 4 @ 555869 w/ 1.1953e+06 gold
14      leprechaun git:(master)      ./leprechaun.out
15 There are 3 leprechauns.
16 Contents of leprechaun map:
17 1 @ 22 w/ 1e+06 gold
18 2 @ 58 w/ 1e+06 gold
19 3 @ 72 w/ 1e+06 gold
20 Iterated leprechauns 5 times
21 Contents of leprechaun map:
22 1 @ -403878 w/ 499268 gold
23 2 @ 50139.7 w/ 848755 gold
24 3 @ 2.4141e+06 w/ 1.65198e+06 gold
25      leprechaun git:(master)      ./leprechaun.out
26 There are 6 leprechauns.
27 Contents of leprechaun map:
28 1 @ 21 w/ 1e+06 gold
29 2 @ 22 w/ 1e+06 gold
30 3 @ 42 w/ 1e+06 gold
31 4 @ 65 w/ 1e+06 gold
32 5 @ 80 w/ 1e+06 gold
33 6 @ 84 w/ 1e+06 gold
34 Iterated leprechauns 5 times
35 Contents of leprechaun map:
36 1 @ -1.16275e+06 w/ 380127 gold
37 2 @ -862824 w/ 1.73828e+06 gold
38 3 @ 580376 w/ 1.03174e+06 gold
39 4 @ 723471 w/ 1.70831e+06 gold
40 5 @ 1.59557e+06 w/ 392792 gold

```

```

41 6 @ 1.69344e+06 w/ 748749 gold
42     leprechaun git:(master)      ./leprechaun.out
43 There are 5 leprechauns.
44 Contents of leprechaun map:
45 1 @ 25 w/ 1e+06 gold
46 2 @ 57 w/ 1e+06 gold
47 3 @ 70 w/ 1e+06 gold
48 4 @ 91 w/ 1e+06 gold
49 5 @ 94 w/ 1e+06 gold
50 Iterated leprechauns 5 times
51 Contents of leprechaun map:
52 1 @ -860354 w/ 523193 gold
53 2 @ -634719 w/ 1.05164e+06 gold
54 3 @ 338756 w/ 587860 gold
55 4 @ 385943 w/ 1.92767e+06 gold
56 5 @ 419655 w/ 909637 gold
57     leprechaun git:(master)      ./leprechaun.out
58 There are 5 leprechauns.
59 Contents of leprechaun map:
60 1 @ 25 w/ 1e+06 gold
61 2 @ 57 w/ 1e+06 gold
62 3 @ 70 w/ 1e+06 gold
63 4 @ 91 w/ 1e+06 gold
64 5 @ 94 w/ 1e+06 gold
65 Iterated leprechauns 5 times
66 Contents of leprechaun map:
67 1 @ -860354 w/ 523193 gold
68 2 @ -634719 w/ 1.05164e+06 gold
69 3 @ 338756 w/ 587860 gold
70 4 @ 385943 w/ 1.92767e+06 gold
71 5 @ 419655 w/ 909637 gold
72     leprechaun git:(master)      ./leprechaun.out
73 There are 4 leprechauns.
74 Contents of leprechaun map:
75 1 @ 8 w/ 1e+06 gold
76 2 @ 19 w/ 1e+06 gold
77 3 @ 49 w/ 1e+06 gold
78 4 @ 83 w/ 1e+06 gold
79 Iterated leprechauns 5 times
80 Contents of leprechaun map:
81 1 @ -96671.6 w/ 745056 gold
82 2 @ 3398.62 w/ 2.22748e+06 gold
83 3 @ 113228 w/ 264465 gold
84 4 @ 113354 w/ 763000 gold
85     leprechaun git:(master)      ./leprechaun.out
86 There are 4 leprechauns.
87 Contents of leprechaun map:
88 1 @ 8 w/ 1e+06 gold
89 2 @ 19 w/ 1e+06 gold
90 3 @ 49 w/ 1e+06 gold
91 4 @ 83 w/ 1e+06 gold
92 Iterated leprechauns 5 times
93 Contents of leprechaun map:
94 1 @ -96671.6 w/ 745056 gold
95 2 @ 3398.62 w/ 2.22748e+06 gold
96 3 @ 113228 w/ 264465 gold
97 4 @ 113354 w/ 763000 gold

```