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 *Pert*ain 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 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 than 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*.

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

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.

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

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.

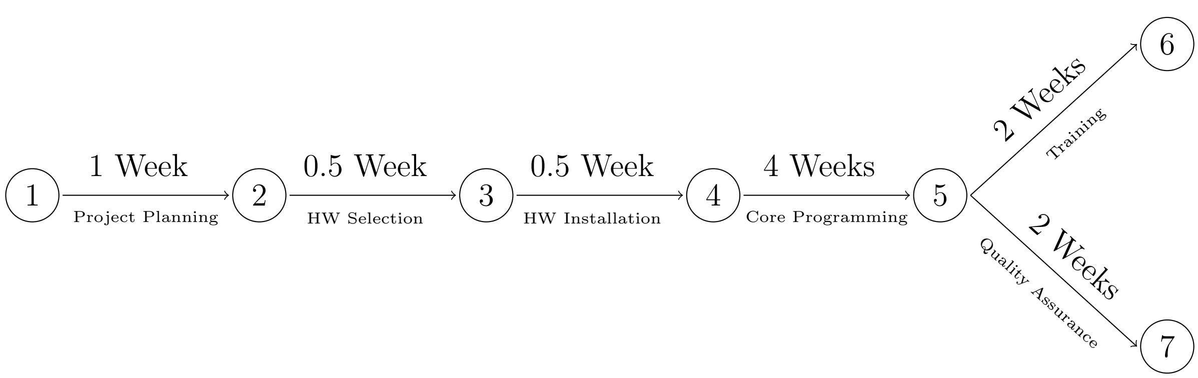
3 Graphing With PERT

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. Continous 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)*. SearchSoftwareQuality. 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 $k-1$, $k-2$, …, 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**
`unionWith(h)`	Merge any binomial trees that have the same order. For any trees that don't match, just place them in the heap.	$O(\log n)$
`min()`	Search through each binomial tree and find the smallest root.	$O(\log n)$
`removeMin()`	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.	$O(\log n)$
`insert`	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 $[0, N1]$, for $N \geq 2$. The algorithm should run in $O(n + N)$ time.	

3.1.1 Compiler Environment

Listing 1: environment

```
    tex git:(master)    pwd
/Users/nicolas/Git/portfolio2/tex
    tex git:(master)    uname -a
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
    tex git:(master)    clang --version
Apple LLVM version 8.0.0 (clang-800.0.42.1)
Target: x86_64-apple-darwin16.1.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
    tex git:(master)    harper_cpp --version
This is harper_cpp version 1.221 executing under perl v5.18.2 and compiling with:

Configured with: --prefix=/Library/Developer/CommandLineTools/usr --with-gxx-include-dir=/usr/include/c++/4.2.1
Apple LLVM version 8.0.0 (clang-800.0.42.1)
Target: x86_64-apple-darwin16.1.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

3.1.2 Source

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

```cpp
#include <iostream>
#include <list>
#include <stack>
#include <cstdio>

std::list<int> bucketSort( const std::list<int>& pList, int nMin, int nMax )
{
    std::stack<int>* pBuckets = new std::stack<int>[static_cast<unsigned int>(nMax - nMin) + 1];
    std::list<int> pReturnList;

    for ( auto i = pList.begin( ); i != pList.end( ); i++ )
    {
        pBuckets[*i - nMin].push( *i );
    }

    for ( int i=nMin; i <= nMax; i++ )
```

```
    {
        while ( !pBuckets[i-nMin].empty( ) )
        {
            pReturnList.push_back( pBuckets[i-nMin].top( ) );
            pBuckets[i-nMin].pop( );
        }
    }

    delete[] pBuckets;

    return pReturnList;
}

std::list<int> bucketSort( std::list<int>& pList )
{
    int nMin = pList.front( ), nMax = pList.front( );

    for ( auto i = ++pList.begin( ); i != pList.end( ); i++ )
    {
        if ( *i < nMin )
        {
            nMin = *i;
        }

        if ( *i > nMax )
        {
            nMax = *i;
        }
    }

    return bucketSort( pList, nMin, nMax );
}

template<typename T>
void printList( const std::list<T>& pList )
{
    std::cout << "Printing list of size " << pList.size( ) << std::endl;

    for ( auto i = pList.begin( ); i != pList.end( ); i++ )
    {
        std::cout << *i << " ";
    }

    std::cout << std::endl;
}

int main( )
{
    srand( static_cast<unsigned int>( time( nullptr ) ) );

    std::list<int> pList;

    for ( size_t i=0; i < 15; i++ )
    {
        pList.push_back( rand( ) % 50 );
    }

    std::cout << "Pre sorted list:" << std::endl;

```

```
    printList( pList );
    pList = bucketSort( pList );

    std::cout << "After sorting: " << std::endl;
    printList( pList );

    return 0;
}
```

3.1.3 Compiler Output

Listing 3: ../lab/bucketsort/compilerout

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

3.1.4 Program Output

Listing 4: ../lab/bucketsort/progout

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

```
Pre sorted list:
Printing list of size 15
33 32 17 30 42 33 47 18 8 35 16 33 46 6 13
After sorting:
Printing list of size 15
6 8 13 16 17 18 30 32 33 33 33 35 42 46 47
    bucketsort git:(master)    ./bucket.out
Pre sorted list:
Printing list of size 15
33 32 17 30 42 33 47 18 8 35 16 33 46 6 13
After sorting:
Printing list of size 15
6 8 13 16 17 18 30 32 33 33 33 35 42 46 47
    bucketsort git:(master)    ./bucket.out
Pre sorted list:
Printing list of size 15
40 31 43 41 25 5 41 49 34 47 6 48 41 48 0
After sorting:
Printing list of size 15
0 5 6 25 31 34 40 41 41 41 43 47 48 48 49
    bucketsort git:(master)    ./bucket.out
Pre sorted list:
Printing list of size 15
4 32 42 10 38 2 29 8 33 21 39 31 31 32 24
After sorting:
Printing list of size 15
2 4 8 10 21 24 29 31 31 32 32 33 38 39 42
    bucketsort git:(master)    ./bucket.out
Pre sorted list:
Printing list of size 15
4 32 42 10 38 2 29 8 33 21 39 31 31 32 24
After sorting:
Printing list of size 15
2 4 8 10 21 24 29 31 31 32 32 33 38 39 42
    bucketsort git:(master)    ./bucket.out
Pre sorted list:
Printing list of size 15
11 31 18 18 18 24 23 39 9 33 32 49 23 24 11
After sorting:
Printing list of size 15
9 11 11 18 18 18 23 23 24 24 31 32 33 39 49
    bucketsort git:(master)    ./bucket.out
Pre sorted list:
Printing list of size 15
11 31 18 18 18 24 23 39 9 33 32 49 23 24 11
After sorting:
Printing list of size 15
9 11 11 18 18 18 23 23 24 24 31 32 33 39 49
    bucketsort git:(master)    ./bucket.out
Pre sorted list:
Printing list of size 15
11 31 18 18 18 24 23 39 9 33 32 49 23 24 11
After sorting:
Printing list of size 15
9 11 11 18 18 18 23 23 24 24 31 32 33 39 49
    bucketsort git:(master)    ./bucket.out
Pre sorted list:
Printing list of size 15
18 30 44 26 1 46 17 17 32 45 22 14 18 16 1
```

```
After sorting:
Printing list of size 15
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 $i = 1, 2, \ldots, n$. 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

$$x_i \leftarrow x_i + rg_i$$

where r is a random floating-point number between 1 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

```
    tex git:(master)    pwd
/Users/nicolas/Git/portfolio2/tex
    tex git:(master)    uname -a
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
    tex git:(master)    clang --version
Apple LLVM version 8.0.0 (clang-800.0.42.1)
Target: x86_64-apple-darwin16.1.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
    tex git:(master)    harper_cpp --version
This is harper_cpp version 1.221 executing under perl v5.18.2 and compiling with:

Configured with: --prefix=/Library/Developer/CommandLineTools/usr --with-gxx-include-dir=/usr/include/c++/4.2.1
Apple LLVM version 8.0.0 (clang-800.0.42.1)
Target: x86_64-apple-darwin16.1.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

3.2.2 Source

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

```cpp
#include <iostream>
#include <cstdlib>
#include "orderedmap.hpp"

typedef double Position;
struct LeprechaunData
{
    double nGold;
    bool bIterated;
};
typedef OrderedMap<Position,LeprechaunData> LepMap;
typedef MapKey<Position,LeprechaunData> LepMapKey;

void iterateLeprechauns( LepMap& pMap )
{
    LepMapKey* i = pMap.first( );

    //for ( LepMapKey* i=pMap.first( ); i != nullptr; i = pMap.getNext( i ) )
    while ( i != nullptr )
    {
        if ( i->nValue.bIterated == false )
        {
            double r = static_cast<double>( rand( ) % 2000 - 1000 ) / 1000.0;
            i->nKey += r * i->nValue.nGold;

            Position p = i->nKey;
            LeprechaunData d = i->nValue;
            d.bIterated = true;

            pMap.remove( i );

            auto* pHigher = pMap.getHigher( p );
            auto* pLower = pMap.getLower( p );

            if ( pHigher != nullptr )
            {
                d.nGold += pHigher->nValue.nGold / 2;
                pHigher->nValue.nGold /= 2;
            }

            if ( pLower != nullptr )
            {
                d.nGold += pLower->nValue.nGold / 2;
                pLower->nValue.nGold /= 2;
            }

            pMap.insert( p, d );
            i = pMap.first( );
        }
        else
        {
            i = pMap.getNext( i );
        }
    }

    for ( LepMapKey* i=pMap.first( ); i != nullptr; i = pMap.getNext( i ) )
    {
        i->nValue.bIterated = false;
    }
```

```
}

void printLeprechauns( LepMap& pMap )
{
    std::cout << "Contents of leprechaun map: " << std::endl;

    int nIter = 1;
    for ( LepMapKey* i=pMap.first( ); i != nullptr; i = pMap.getNext( i ) )
    {
        std::cout << nIter++ << " @ " << i->nKey << " w/ " << i->nValue.nGold << "
            gold" << std::endl;
    }
}

int main( )
{
    LepMap pLeprechauns;

    std::srand( static_cast<unsigned int>( time( nullptr ) ) );
    size_t nLeprechauns = std::rand( ) % 4 + 3;

    std::cout << "There are " << nLeprechauns << " leprechauns." << std::endl;

    for ( size_t i=0; i < nLeprechauns; i++ )
    {
        pLeprechauns.insert( std::rand( ) % 100, { 1000000, false } );
    }

    printLeprechauns( pLeprechauns );

    for ( int i=0; i < 5; i++ )
    {
        iterateLeprechauns( pLeprechauns );
    }

    std::cout << "Iterated leprechauns 5 times" << std::endl;

    printLeprechauns( pLeprechauns );
}
```

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

```
#ifndef BTREE_HPP
#define BTREE_HPP

#include <functional>

template<typename Data>
class BinaryTreeNode
{
private:
    BinaryTreeNode<Data>* pLeft,* pRight,* pParent;
    Data pData;

public:
    /* big three */

    BinaryTreeNode( ): pLeft( nullptr ), pRight( nullptr ), pParent( nullptr ),
        pData( ) { }
```

```cpp

    BinaryTreeNode( const Data& pDataNew ): pLeft( nullptr ), pRight( nullptr ), pParent( nullptr ), pData( pDataNew ) { }

    BinaryTreeNode( const BinaryTreeNode& pNode ): pLeft( nullptr ), pRight( nullptr ), pParent( nullptr ), pData( )
    {
        operator=( pNode );
    }

    BinaryTreeNode& operator=( const BinaryTreeNode& pNode )
    {
        if ( pNode.pLeft != nullptr )
        {
            pLeft = new BinaryTreeNode( *pNode.pLeft );
        }

        if ( pNode.pRight != nullptr )
        {
            pRight = new BinaryTreeNode( *pNode.pRight );
        }

        pParent = pNode.pParent;

        return *this;
    }

    ~BinaryTreeNode( )
    {
        if ( pLeft != nullptr )
        {
            delete pLeft;
        }

        if ( pRight != nullptr )
        {
            delete pRight;
        }
    }

    Data& getData( )
    {
        return pData;
    }

    BinaryTreeNode*& getLeft( )
    {
        return pLeft;
    }

    BinaryTreeNode*& getRight( )
    {
        return pRight;
    }

    BinaryTreeNode* getLeftNode( )
    {
        return pLeft;
    }
```

```

    BinaryTreeNode* getRightNode( )
    {
        return pRight;
    }

    BinaryTreeNode* getParent( )
    {
        return pParent;
    }

    void setParent( BinaryTreeNode* pNewParent )
    {
        pParent = pNewParent;
    }

    bool isRightChild( )
    {
        if ( getParent( ) == nullptr )
        {
            return false;
        }
        else
        {
            if ( getParent( )->getRightNode( ) == this )
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }

    bool isLeftChild( )
    {
        return !isRightChild( );
    }

    Data* getDataPtr( )
    {
        return &pData;
    }

    Data setData( const Data& pDataNew )
    {
        pData = pDataNew;
    }

    /* create children */

    BinaryTreeNode* createLeftNode( )
    {
        if ( pLeft )
        {
            delete pLeft;
            pLeft = nullptr;
        }
```

```
        pLeft = new BinaryTreeNode;
        pLeft->pParent = this;

        return pLeft;
    }

    BinaryTreeNode* createLeftNode( const Data& pDataNew )
    {
        if ( pLeft )
        {
            delete pLeft;
            pLeft = nullptr;
        }

        pLeft = new BinaryTreeNode( pDataNew );
        pLeft->pParent = this;

        return pLeft;
    }

    BinaryTreeNode* createRightNode( )
    {
        if ( pRight )
        {
            delete pRight;
            pRight = nullptr;
        }

        pRight = new BinaryTreeNode;
        pRight->pParent = this;

        return pRight;
    }

    BinaryTreeNode* createRightNode( const Data& pDataNew )
    {
        if ( pRight )
        {
            delete pRight;
            pRight = nullptr;
        }

        pRight = new BinaryTreeNode( pDataNew );
        pRight->pParent = this;

        return pRight;
    }

    void removeLeftNode( )
    {
        delete pLeft;
        pLeft = nullptr;
    }

    void removeRightNode( )
    {
        delete pRight;
        pRight = nullptr;
```

```
    }

    /* checks to see if children exist */

    bool hasLeft( ) const
    {
        return ( pLeft ? true : false );
    }

    bool hasRight( ) const
    {
        return ( pRight ? true : false );
    }

    bool hasChildren( ) const
    {
        return ( ( pRight && pLeft ) ? true : false );
    }

    bool hasAnyChildren( ) const
    {
        return ( ( pRight || pLeft ) ? true : false );
    }

    /* traversals */

    void preorderTraversal( const std::function<void(BinaryTreeNode<Data>*)>&
        pTraverseFunction )
    {
        pTraverseFunction( this );

        if ( pLeft != nullptr )
        {
            pLeft->preorderTraversal( pTraverseFunction );
        }

        if ( pRight != nullptr )
        {
            pRight->preorderTraversal( pTraverseFunction );
        }
    }

    void postorderTraversal( const std::function<void(BinaryTreeNode<Data>*)>&
        pTraverseFunction )
    {
        if ( pLeft != nullptr )
        {
            pLeft->postorderTraversal( pTraverseFunction );
        }

        if ( pRight != nullptr )
        {
            pRight->postorderTraversal( pTraverseFunction );
        }

        pTraverseFunction( this );
    }

    void inorderTraversal( const std::function<void(BinaryTreeNode<Data>*)>&
```

```
        pTraverseFunction )
    {
        if ( pLeft != nullptr )
        {
            pLeft->inorderTraversal( pTraverseFunction );
        }

        pTraverseFunction( this );

        if ( pRight != nullptr )
        {
            pRight->inorderTraversal( pTraverseFunction );
        }
    }
};

template<typename Data>
class BinaryTree
{
protected:
    BinaryTreeNode<Data>* pRoot;

public:
    /* big three */

    BinaryTree( ): pRoot( nullptr ) { }

    BinaryTree( const BinaryTree& pTree ): pRoot( nullptr )
    {
        if ( pTree.pRoot != nullptr )
        {
            pRoot = new BinaryTreeNode<Data>( *pTree.pRoot );
        }
    }

    BinaryTree& operator=( const BinaryTree& pTree )
    {
        if ( pRoot != nullptr )
        {
            delete pRoot;
            pRoot = nullptr;
        }

        if ( pTree.pRoot != nullptr )
        {
            pRoot = new BinaryTreeNode<Data>( *pTree.pRoot );
        }

        return *this;
    }

    ~BinaryTree( )
    {
        if ( pRoot != nullptr )
        {
            delete pRoot;
        }
    }

```

```
    BinaryTreeNode<Data>*& getRoot( )
    {
        return pRoot;
    }

    BinaryTreeNode<Data>* getRoot( ) const
    {
        return pRoot;
    }

    BinaryTreeNode<Data>* createRoot( )
    {
        if ( pRoot )
        {
            delete pRoot;
            pRoot = nullptr;
        }

        pRoot = new BinaryTreeNode<Data>;

        return pRoot;
    }

    BinaryTreeNode<Data>* createRoot( const Data& pData )
    {
        if ( pRoot )
        {
            delete pRoot;
            pRoot = nullptr;
        }

        pRoot = new BinaryTreeNode<Data>( pData );

        return pRoot;
    }

    void deleteRoot( )
    {
        delete pRoot;
    }

    void preorderTraversal( const std::function<void(BinaryTreeNode<Data>*)>&
        pTraverseFunction )
    {
        if ( pRoot != nullptr )
        {
            pRoot->preorderTraversal( pTraverseFunction );
        }
    }

    void postorderTraversal( const std::function<void(BinaryTreeNode<Data>*)>&
        pTraverseFunction )
    {
        if ( pRoot != nullptr )
        {
            pRoot->postorderTraversal( pTraverseFunction );
        }
    }

```

```
    void inorderTraversal( const std::function<void(BinaryTreeNode<Data>*)>&
        pTraverseFunction )
    {
        if ( pRoot != nullptr )
        {
            pRoot->inorderTraversal( pTraverseFunction );
        }
    }
};

#endif
```

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

```
#ifndef ORDERED_MAP_HPP
#define ORDERED_MAP_HPP

#include "btree.hpp"

template<typename Key, typename Value>
struct MapKey
{
    Key nKey;
    Value nValue;

    BinaryTreeNode< struct MapKey<Key,Value> >* pNode;
};

template<typename Key, typename Value>
class OrderedMap
{
private:
    typedef MapKey<Key,Value> MapNode;
    BinaryTree<MapNode> pSearchTree;

    MapNode* getMaxUnderValue( const Key& nKey, BinaryTreeNode<MapNode>* pNode )
    {
        if ( nKey <= pNode->getData( ).nKey &&
             pNode->hasLeft( ) )
        {
            return getMaxUnderValue( nKey, pNode->getLeftNode( ) );
        }
        if ( nKey > pNode->getData( ).nKey &&
             pNode->hasRight( ) )
        {
            MapNode* pNodeTemp =
                getMaxUnderValue( nKey, pNode->getRightNode( ) );

            if ( pNodeTemp != nullptr )
            {
                Key nTemp = pNodeTemp->nKey;

                return ( nTemp > pNode->getDataPtr( )->nKey ) ? pNodeTemp : pNode->
                    getDataPtr( );
            }
            else
            {
                return pNode->getDataPtr( );
            }
```

```
        }
        else if ( nKey > pNode->getDataPtr( )->nKey )
        {
            return pNode->getDataPtr( );
        }

        return nullptr;
    }

    MapNode* getMinOverValue( const Key& nKey, BinaryTreeNode<MapNode>* pNode )
    {
        if ( nKey >= pNode->getData( ).nKey &&
             pNode->hasRight( ) )
        {
            return getMinOverValue( nKey, pNode->getRightNode( ) );
        }
        if ( nKey < pNode->getData( ).nKey &&
             pNode->hasLeft( ) )
        {
            MapNode* pNodeTemp =
                getMinOverValue( nKey, pNode->getLeftNode( ) );

            if ( pNodeTemp != nullptr )
            {
                Key nTemp = pNodeTemp->nKey;

                return ( nTemp < pNode->getDataPtr( )->nKey ) ? pNodeTemp : pNode->
                    getDataPtr( );
            }
            else
            {
                return pNode->getDataPtr( );
            }
        }
        else if ( nKey < pNode->getDataPtr( )->nKey )
        {
            return pNode->getDataPtr( );
        }

        return nullptr;
    }

public:
    OrderedMap( ): pSearchTree( ) { }
    OrderedMap( const OrderedMap& pMap ): pSearchTree( pMap.pSearchTree ) { }
    OrderedMap& operator=( const OrderedMap& pMap )
    {
        pSearchTree = pMap.pSearchTree;
    }

    /* Returns a reference to the smallest key value */

    MapNode* first( )
    {
        if ( pSearchTree.getRoot( ) == nullptr )
        {
            throw std::runtime_error( "Calling first() when map is empty." );
        }

```

```
        BinaryTreeNode<MapNode>* pLeft = pSearchTree.getRoot( );

        while ( pLeft->getLeftNode( ) != nullptr )
        {
            pLeft = pLeft->getLeftNode( );
        }

        return pLeft->getDataPtr( );
    }

    /* Returns a reference to the largest key value */

    MapNode* last( )
    {
        if ( pSearchTree.getRoot( ) == nullptr )
        {
            throw std::runtime_error( "Calling last() when map is empty." );
            return nullptr;
        }

        BinaryTreeNode<MapNode>* pRight = pSearchTree.getRoot( );

        while ( pRight->getRightNode( ) != nullptr )
        {
            pRight = pRight->getRightNode( );
        }

        return pRight->getDataPtr( );
    }

    MapNode* find( const Key& nKey )
    {
        if ( pSearchTree.getRoot( ) == nullptr )
        {
            throw std::runtime_error( "Calling find() when map is empty." );
            return nullptr;
        }

        BinaryTreeNode<MapNode>* pCurrent = pSearchTree.getRoot( );
        bool bLooping = true;

        while ( bLooping && pCurrent->getDataPtr( )->nKey != nKey )
        {
            if ( pCurrent->hasRight( ) )
            {
                if ( nKey > pCurrent->getRightNode( )->getDataPtr( )->nKey )
                {
                    pCurrent = pCurrent->getRightNode( );
                }
                else
                {
                    bLooping = false;
                }
            }
            else if ( pCurrent->hasLeft( ) )
            {
                if ( nKey < pCurrent->getLeftNode( )->getDataPtr( )->nKEy )
                {
                    pCurrent = pCurrent->getLeftNode( );
```

```
                }
                else
                {
                    bLooping = false;
                }
            }
        }

        if ( pCurrent->getDataPtr( )->nKey == nKey )
        {
            return pCurrent->getDataPtr( );
        }
        else
        {
            return nullptr;
        }
    }

    /* Returns the key that has the greatest value less than pKey */

    MapNode* getLower( const Key& nKey )
    {
        return getMaxUnderValue( nKey, pSearchTree.getRoot( ) );
    }

    /* Returns the key that has the smallest value greater than pKey */

    MapNode* getHigher( const Key& nKey )
    {
        return getMinOverValue( nKey, pSearchTree.getRoot( ) );
    }

    MapNode* getNext( const MapNode* pKey )
    {
        if ( pKey == last( ) )
        {
            return nullptr;
        }

        BinaryTreeNode<MapNode>* pCurrent = pKey->pNode;

        if ( pCurrent->getParent( ) != nullptr )
        {
            if ( pCurrent->hasRight( ) )
            {
                /* If we are a middle node, go to the right. */

                pCurrent = pCurrent->getRightNode( );

                while ( pCurrent->hasLeft( ) )
                {
                    pCurrent = pCurrent->getLeftNode( );
                }
            }
            else if ( pCurrent->getParent( )->getLeftNode( ) == pCurrent )
            {
                /* If we are a left node, go up to the middle node. */

                pCurrent = pCurrent->getParent( );
```

```
                }
                else
                {
                    /* If we are a right node, we have to traverse the tree
                       until we get to a suitable middle node */

                    while ( pKey->nKey >
                            pCurrent->getParent( )->getData( ).nKey )
                    {
                        pCurrent = pCurrent->getParent( );
                    }

                    pCurrent = pCurrent->getParent( );
                }
            }
            else
            {
                if ( pCurrent->hasRight( ) )
                {
                    pCurrent = pCurrent->getRightNode( );

                    while ( pCurrent->hasLeft( ) )
                    {
                        pCurrent = pCurrent->getLeftNode( );
                    }
                }
                else
                {
                    return nullptr;
                }
            }

            return pCurrent->getDataPtr( );
        }

        MapNode* insert( const Key& nKey, const Value& nValue )
        {
            BinaryTreeNode<MapNode>* pNode;

            if ( pSearchTree.getRoot( ) == nullptr )
            {
                /* If our search tree is empty just stick it at the top */

                pNode = pSearchTree.createRoot( );
            }
            else
            {
                /* Otherwise loop through and find a good place to put it. */

                BinaryTreeNode<MapNode>* pCurrent = pSearchTree.getRoot( );
                bool bLooping = true;

                while ( bLooping )
                {
                    if ( nKey < pCurrent->getDataPtr( )->nKey )
                    {
                        /* Go left if smaller */

                        if ( pCurrent->hasLeft( ) )
```

```
                    {
                        pCurrent = pCurrent->getLeftNode( );
                    }
                    else
                    {
                        pNode = pCurrent->createLeftNode( );
                        bLooping = false;
                    }
                }
                else if ( nKey >= pCurrent->getDataPtr( )->nKey )
                {
                    /* Go right if larger */

                    if ( pCurrent->hasRight( ) )
                    {
                        pCurrent = pCurrent->getRightNode( );
                    }
                    else
                    {
                        pNode = pCurrent->createRightNode( );
                        bLooping = false;
                    }
                }
            }
        }

        MapNode* pData = pNode->getDataPtr( );
        pData->pNode = pNode;
        pData->nKey = nKey;
        pData->nValue = nValue;

        return pData;
    }

    void remove( MapNode* pNode )
    {
        if ( pNode == nullptr )
        {
            return;
        }

        BinaryTreeNode<MapNode>* pTreeNode = pNode->pNode;

        if ( !pNode->pNode->hasAnyChildren( ) )
        {
            /* No children */

            if ( pTreeNode->getParent( ) != nullptr )
            {
                if ( pTreeNode->isRightChild( ) )
                {
                    pTreeNode->getParent( )->removeRightNode( );
                }
                else
                {
                    pTreeNode->getParent( )->removeLeftNode( );
                }
            }
            else if ( pSearchTree.getRoot( ) == pTreeNode )
```

```
            {
                pSearchTree.deleteRoot( );
            }
        }
        else if ( !pTreeNode->hasChildren( ) && pTreeNode->hasAnyChildren( ) )
        {
            /* Only one child */

            BinaryTreeNode<MapNode>* pChild;

            if ( pTreeNode->hasLeft( ) )
            {
                pChild = pTreeNode->getLeft( );
            }
            else
            {
                pChild = pTreeNode->getRight( );
            }

            if ( pTreeNode->getParent( ) != nullptr )
            {
                if ( pTreeNode->isRightChild( ) )
                {
                    pTreeNode->getParent( )->getRight( ) = pChild;
                }
                else
                {
                    pTreeNode->getParent( )->getLeft( ) = pChild;
                }

                pChild->setParent( pTreeNode->getParent( ) );
            }
            else
            {
                pSearchTree.getRoot( ) = pChild;
                pChild->setParent( nullptr );
            }

            pTreeNode->getLeft( ) = nullptr;
            pTreeNode->getRight( ) = nullptr;
            delete pTreeNode;
        }
        else
        {
            /* Has both children
               Find the smallest node on the right side */

            BinaryTreeNode<MapNode>* pSmallest = pTreeNode->getRight( );

            while ( pSmallest->hasLeft( ) )
            {
                pSmallest = pSmallest->getLeft( );
            }

            pTreeNode->getData( ) = pSmallest->getData( );
            pTreeNode->getData( ).pNode = pTreeNode;

            remove( pSmallest->getDataPtr( ) );
        }
```

```
    }
};

#endif
```

3.2.3 Compiler Output

Listing 9: ../lab/leprechaun/compilerout

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

3.2.4 Program Output

Listing 10: ../lab/leprechaun/progout

```
     leprechaun git:(master)      ./leprechaun.out
There are 4 leprechauns.
Contents of leprechaun map:
1 @ 11 w/ 1e+06 gold
2 @ 23 w/ 1e+06 gold
3 @ 29 w/ 1e+06 gold
4 @ 32 w/ 1e+06 gold
Iterated leprechauns 5 times
Contents of leprechaun map:
1 @ -1.1785e+06 w/ 894958 gold
2 @ 206579 w/ 947357 gold
3 @ 333358 w/ 962387 gold
4 @ 555869 w/ 1.1953e+06 gold
     leprechaun git:(master)      ./leprechaun.out
There are 3 leprechauns.
Contents of leprechaun map:
1 @ 22 w/ 1e+06 gold
2 @ 58 w/ 1e+06 gold
3 @ 72 w/ 1e+06 gold
Iterated leprechauns 5 times
Contents of leprechaun map:
1 @ -403878 w/ 499268 gold
2 @ 50139.7 w/ 848755 gold
3 @ 2.4141e+06 w/ 1.65198e+06 gold
     leprechaun git:(master)      ./leprechaun.out
There are 6 leprechauns.
Contents of leprechaun map:
1 @ 21 w/ 1e+06 gold
2 @ 22 w/ 1e+06 gold
3 @ 42 w/ 1e+06 gold
4 @ 65 w/ 1e+06 gold
5 @ 80 w/ 1e+06 gold
6 @ 84 w/ 1e+06 gold
Iterated leprechauns 5 times
Contents of leprechaun map:
1 @ -1.16275e+06 w/ 380127 gold
2 @ -862824 w/ 1.73828e+06 gold
3 @ 580376 w/ 1.03174e+06 gold
4 @ 723471 w/ 1.70831e+06 gold
5 @ 1.59557e+06 w/ 392792 gold
```

```
6 @ 1.69344e+06 w/ 748749 gold
     leprechaun git:(master)     ./leprechaun.out
There are 5 leprechauns.
Contents of leprechaun map:
1 @ 25 w/ 1e+06 gold
2 @ 57 w/ 1e+06 gold
3 @ 70 w/ 1e+06 gold
4 @ 91 w/ 1e+06 gold
5 @ 94 w/ 1e+06 gold
Iterated leprechauns 5 times
Contents of leprechaun map:
1 @ -860354 w/ 523193 gold
2 @ -634719 w/ 1.05164e+06 gold
3 @ 338756 w/ 587860 gold
4 @ 385943 w/ 1.92767e+06 gold
5 @ 419655 w/ 909637 gold
     leprechaun git:(master)     ./leprechaun.out
There are 5 leprechauns.
Contents of leprechaun map:
1 @ 25 w/ 1e+06 gold
2 @ 57 w/ 1e+06 gold
3 @ 70 w/ 1e+06 gold
4 @ 91 w/ 1e+06 gold
5 @ 94 w/ 1e+06 gold
Iterated leprechauns 5 times
Contents of leprechaun map:
1 @ -860354 w/ 523193 gold
2 @ -634719 w/ 1.05164e+06 gold
3 @ 338756 w/ 587860 gold
4 @ 385943 w/ 1.92767e+06 gold
5 @ 419655 w/ 909637 gold
     leprechaun git:(master)     ./leprechaun.out
There are 4 leprechauns.
Contents of leprechaun map:
1 @ 8 w/ 1e+06 gold
2 @ 19 w/ 1e+06 gold
3 @ 49 w/ 1e+06 gold
4 @ 83 w/ 1e+06 gold
Iterated leprechauns 5 times
Contents of leprechaun map:
1 @ -96671.6 w/ 745056 gold
2 @ 3398.62 w/ 2.22748e+06 gold
3 @ 113228 w/ 264465 gold
4 @ 113354 w/ 763000 gold
     leprechaun git:(master)     ./leprechaun.out
There are 4 leprechauns.
Contents of leprechaun map:
1 @ 8 w/ 1e+06 gold
2 @ 19 w/ 1e+06 gold
3 @ 49 w/ 1e+06 gold
4 @ 83 w/ 1e+06 gold
Iterated leprechauns 5 times
Contents of leprechaun map:
1 @ -96671.6 w/ 745056 gold
2 @ 3398.62 w/ 2.22748e+06 gold
3 @ 113228 w/ 264465 gold
4 @ 113354 w/ 763000 gold
```