Contents

1.0 Corrections and Changes

- Insertion of data item 9 into the tree in the figure below is incorrect. It should have been added as the right child of the node holding data item 8. (10/24/2012)
- We have decided to show you how to read data files directly rather than using a pipe since using a pipe is almost as complicated as just reading the file from a hard drive. (10/24/2012)
- Section 4.1 Experiment 1 part 5. You should only try to speed up the getData() for the sorted array not the unsorted array. The idea is that you will be able to make use of the sorted property to find an element much faster than you could if the array were unsorted. (11/01/2012)

2.0 Terms

You should know these terms by the end of the project.

- Recursive data-structure
- Tree
- Sorted Linked List
- Edge (Link)
- Node (Vertex)
- Root
- Binary Tree
- Sorted Binary Tree
- Pipe
- Leaf
- Wall Time
- Interior Node

3.0 Description

An important part of studying computation is understanding how quickly a program you have written can solve a particular problem. In this project you will perform experiments to determine how *fast* we can perform two simple operations using three different data-structures.

You will implement the data-structures, perform experiments with different datasets, measure the performance of your implementation, and write a report in which you present and explain your results.

Two of these data-structures you have worked with already: arrays and linked lists.

Linked lists belong to a set called *recursive data-structures*. Recursion is a process which refers to itself and is a common method of solving problems in computer science. The linked list you wrote in the previous lab was built by repeatedly adding nodes. The fractals you created in the first project were also recursive since they were formed by repeatedly evaluating a function that referred to itself.

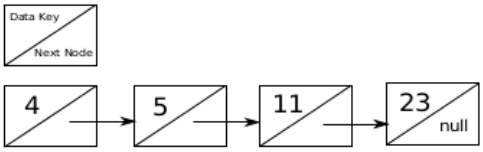
Here is a useful wikipedia article with many examples of recursion: http://en.wikipedia.org/wiki/Recursion.

Unlike your previous linked list the one you write for this project will keep the elements you enter in order. The student with the least key will always be at the head, the next smallest element comes next etc. This sorted property must not depend on the order in which you add data to the list. A linked list with this property is called a *sorted linked list*.

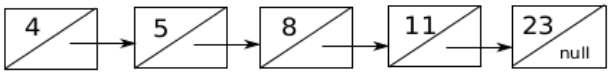
The third data-structure you will analyze in this Project is a *binary tree*, which is another recursive data-structure. Trees are an extension of linked lists where instead of each node leading to one other node we will allow each node to point to more than one other node. The binary tree structure has the following rules: one node is designated the *root* (this is analogous to the head of the linked list), every node can point to two other nodes, there are no cycles allowed. Counter-intuitively computer scientists draw trees upside down so the root is at the top and the *leaves* are at the bottom.

We are going to use the binary tree to store data just as we used the array and linked list to store data. As you will discover using a tree structure has advantages over using linked lists and and over arrays.

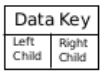
Sorted Linked List

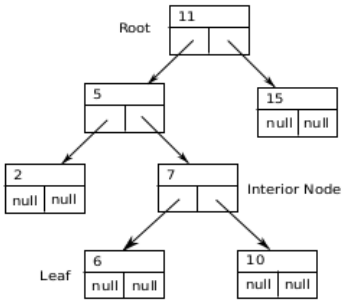


Insert Data Item with Key 8

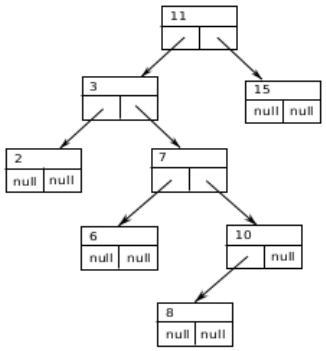


Sorted Binary Tree

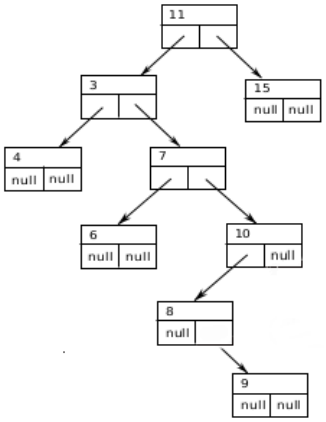




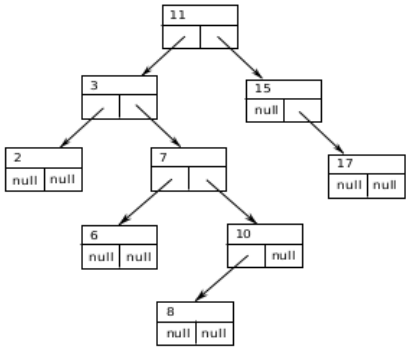
Insert Data Item with Key 8



Insert Data Item with Key 9



Insert Data Item with Key 17



4.0 Experiments

All the experiments will analyze the speed with which the three data-structures carry out two operations. The operations are:

- addData(key, data)
- getData(key)

You will perform the following experiments using a separate program you have written for each data-structure.

The programs will each take a list of n elements to put into their data-structure via a pipe and print four values: 1) The average number of comparisons that it took to add the items that were piped in, 2) The average number of comparisons it took to get m items out of the data-structure, 3) The average amount of wall time that it took to add the items that were piped in, 4) The average amount of wall time it took to get m items out of the data-structure.

You will use these four output values to create the figures for your report.

We will use the student class you were shown in lecture. Recall that each student has a name and an ID and a compareTo() method. You will need to use the compareTo() method in your experiments.

In each experiment you are going to keep track of two measures of speed. 1) Record how many comparisons your software makes by counting the number of times compareTo() gets called. 2) Keep track of the actual time that passes from the start of each experiment to the end (this is called the wall time).

Repeat each experiment at least 10 times choosing values of n and m between 1 and 5,000 so you can clearly see how the number of comparisons scales with the number of data elements your data-structures hold.

4.1 Experiment 1

1. Create an array of size n
2. Fill the array with data elements using an addData() method you write. Make sure the student IDs are not in sequence in the array.
3. Call getData() m times with random keys.
4. Sort the array (you may use methods provided by Java to sort the array).
5. Repeat step 3. Find a way to significantly increase the speed of your getData() for *the sorted array* vs. the unsorted array.

4.2 Experiment 2

1. Create a Sorted Linked List (write your own).
2. Fill the list with n data elements using an addData() method you write.
3. Call getData() m times with random keys.

4.3 Experiment 3

1. Create a binary tree (write your own). This binary tree will store items in sorted order just like the sorted linked list. The tree will maintain the following property: for each node containing a data item with some key (call it k) all the keys of data-elements in the left subtree will be less than k and all keys of the data-elements in the right subtree will greater than k.
2. Add n data items to the tree using an addData() method you write.
3. Call getData() m times with random keys.
4. Discover what property of the tree determines how quickly data items can be retrieved. Notice the worst case and the best case for the speed with which a data element can be retrieved. (Use the two provided datasets for this).

4.4 Experiment 4

Write a method called optimizeTree() that takes advantage of your observation in part 4 of Experiment 3 so that getData() tries to avoid the worst case of the regular binary tree. You do not have to keep track of any comparisons you use in optimizeTree(). That is the you get the work done by optimizeTree() for free without having it count against the cost of getData(). You are allowed to use extra storage space in your solution if you need it.

5.0 Report

Write a report detailing the results of your experiments. The report should be no longer than 2 pages. Add an appendix with your source code (the appendix doesn't count against your two pages).
Your report will include the following figures:

5.1 Figure 1

A plot comparing the speed of getData() for the unsorted and sorted arrays. The x-axis is the number of items added and the y-axis will be the number of comparisons getData() made.

5.2 Figure 2

A plot comparing the speed of addData() for your three data-structures (sorted array, sorted list, sorted binary tree). The x-axis is the number of data items added and the y-axis will be the *wall time* addData() took. (Extra Credit: include your extra credit tree data on the plot).

5.3 Figure 3

A plot comparing the speed of addData() for your three data-structures (sorted array, sorted list, sorted binary tree). The x-axis is the number of data items added and the y-axis will be the *comparisons* addData() made. (Extra Credit: include your extra credit tree data on the plot).

5.4 Figure 4

A plot comparing the speed of getData() for your three data-structures (sorted array, sorted list, sorted binary tree). The x-axis is the number of data items added and the y-axis will be the *wall time* getData() took. (Extra Credit: include your extra credit tree data on the plot).

5.5 Figure 5

A plot comparing the speed of getData() for your three data-structures (sorted array, sorted list, sorted binary tree). The x-axis is the number of data items added and the y-axis will be the *comparisons* getData() made. (Extra Credit: include your extra credit tree data on the plot).

Use prose to compare the difference in getData() for the sorted and unsorted arrays.

Use prose to compare the differences and similarities in the speed of getData() and addData() for the sorted array, sorted list, and sorted tree. Discuss the best and worst cases for how quickly an item can be retrieved for each of the data-structures. Try to characterize the relationship between the number of elements in your data-structures and the speed with which you can add and retrieve data using simple mathematical functions. This does not need to be formal just try to match the data in your plots to functions you know from algebra class.

Extra credit: Explain in prose what you noticed about the speed of getData() and addData() that helped you decide how the tree needed to be organized so it would reduced the occurrences of worst case running time for getData() and addData().

Appendix:

Include your source code.

6.0 Grading

There are a 100 points available for this project + 10 points of extra credit.

Report:

- Inclusion of results from Experiment 1 (10 points).
- Inclusion of results from Experiment 2 (20 points).
- Inclusion of results from Experiment 3 (30 points).
- Accuracy of comparison based results (10 points).
- Overall quality of report:
  Clarity of prose and figures, insight into issues. (20 points)
- Extra Credit: 10 points.

Source Code: Code Style: (10 points)

7.0 Submission

Zip up your three programs and the source code and mail it to cs251f12@gmail.com.

Print your report and turn it in during lecture on Friday Nov 9th.