Program #1
BCS Ranking Program

Nathan Balon
CIS 350
Data Structures and Algorithms
1/18/2005

Description of the Problem

There is a problem with the rankings of teams in college football. The BCS currently has a ranking system but it has been found to be inadequate. A new system is to be developed that uses the bookies of America, to rank the top college football teams. Each bookie will give there rankings of what they feel are the top 6 teams in the country.

A program needs to be developed to read in the ranking from the bookies of the top six teams and then to determine what the median ranking for the teams are based on the input given. The median ranking then will be used establish the national ranking of top teams. The program will accept a number of rankings given by bookies as input. Then the program will compute the median ranking and display it to the user.

Input Specification

The program will read multiple sets of input. In a set the first input that will be given to the program is the number of rankings `n`. The variable `n` will be no larger than 10,000. After the number of rankings is established the rankings will be read into the program. The sets of rankings will be a string of six characters, consisting of the characters {A, B, C, D, E, F} with none being repeated and no spaces in between the characters. The strings read in will have different permutations of these characters. The program will continue to read the strings of rankings into the program until it reads all rankings of the number `n`. After a set of rankings is read in the program attempts to read in another set of rankings. If `n` equals 0 is found when reading in the number of rankings, then the end of input is found and the program terminates.

An example of the input that is used for the program is given in figure 1.

```
4
ABDCEF
BACDEF
CADBFE
ABCDEF
0
```

Figure 1

Output Specification

After the program has calculated the median rank from the set of ranking it was given as input the median rank will be displayed to the user. If there is more than one median ranking with the same value, then the ranking that comes first alphabetically will be displayed to the user. The output of the program will be a string in the form "**`ranking`** `is the median ranking with value` **`value`**." In this case **`ranking`** and **`value`** with be replaced with the correct output from the program. **`Ranking`** will be the correct ranking and **`value`** will be the value of the ranking. The program will continue to display the median ranking for each set of input rankings given by the user.

An example of the output that could be displayed by the program is given in figure 2.

ABDCEF is the median ranking with a value of 4.

Figure 2

Data structure Implementation and Discussion

The data structure `Ranking` is the data structure that is used by the program to determine the median ranking from a set of input rankings. `Ranking` uses a vector to store a set of rankings that were given as input to the program. At first I was going to use a pointer was used to avoid copying all of the values in the vector when a `Ranking` object is created. In the end, I decided against this option. While it would have improved the performance of the program it is possible that the vector that is pointed to by the object could be deleted, which would cause unexpected results. Because of this I changed the implementation of the class to pass a vector by value when creating an object. The class contains the data member's `_inputRankings`, `_medianRanking`, `_rankValue` and `_intialPermutation`. First, `_inputRanking` is a vector of string used to store all the ranking in a set that was given to the program as input. Second, `_medianRanking` contains a string which is to store the median ranking. Third, `_rankValue` contains the value of the rank. Last, `_intialPermutation`, contains a string that is to be used for the initial permutation that is used to calculate a median ranking.

The class `Ranking` has one constructor. To use the constructor the user supplies a vector of strings which contain the ranking which were entered as input. The class also contains a destructor that is used to deallocate memory that was allocated for the vector in the constructor.

The `Ranking` class has five member functions. The functions are `createPermutationString`, `distanceBetweenStrings`, `calcCandidateRank`, `calcRank`, and `displayMedianRanking`. To Begin, `createPermutationString` is called by the constructor when a `Ranking` object is created. Initial I created a constant string to store the value of the intial permutation the problem with that approach is that it did not allow the program to be run on set of input that varied in length. Next, `distanceBetweenStrings` is a private member function that is used to calculate the distance between two strings. The function was made private because it does not change or display any state information of an object and it is used just to process the difference between two strings. The member function `calcCandidateRank` uses `distanceBetweenStrings` to determine the distance between to strings when calculating the median rank. Also, `calcCandidateRank` is another private function which used to compute the sum of the distance from candidate ranking to all of the voted rankings. The next member function is `calcRank` which is used to determine which candidate ranking is the median ranking for the set of voted rankings. Fourth, displayMedianRanking sends median ranking to standard output. These are the member functions used by the class `Ranking`.

The main of the program uses two global functions `getNumberOfRankings` and `getInputRankings`. These two functions are used to get the input of the program from the user. The function `getNumberOfRankings` reads in the number of rankings

that are contained in a set of ranks which is to have it median ranking calculated. Next, the function `getInputRankings` reads in a number of strings containing the rankings. The number of strings which are read in is equal to the value returned from the function `getNumberOfRankings`. After a string of rankings is read in it is added to a vector which is returned from the function. These two functions were not necessary to have it would have been possible to have implemented the functionality of these functions in main but it makes the program easier to read by creating separate functions.

The class that was created to hold the rankings could add some additional methods to give the user of the class more functionality. Methods could be created to add new rankings to an object. Also, the functionality to remove a ranking from an object could also be added. There are a few other things that could be added to class to improve its functionality. As it stands the class can not be used to be stored in STL container since it doesn't implement a copy constructor, the assignment operator and overload the less than operator. These features were left out of the program since they wouldn't have any effect on the assignment at hand. For this class to be used as a general purpose class, these additional features could be added.

It would have also been possible to program the assignment using procedural programming such as how a c program would be written. For this program there isn't any interaction with other objects. The one benefit of creating a class to store the data structure is it provides encapsulation of the data. Also, in the future it would be easy to provide additional functionality to this class.

The diagram below shown below in figure 3 contains the class diagram of `Ranking`.

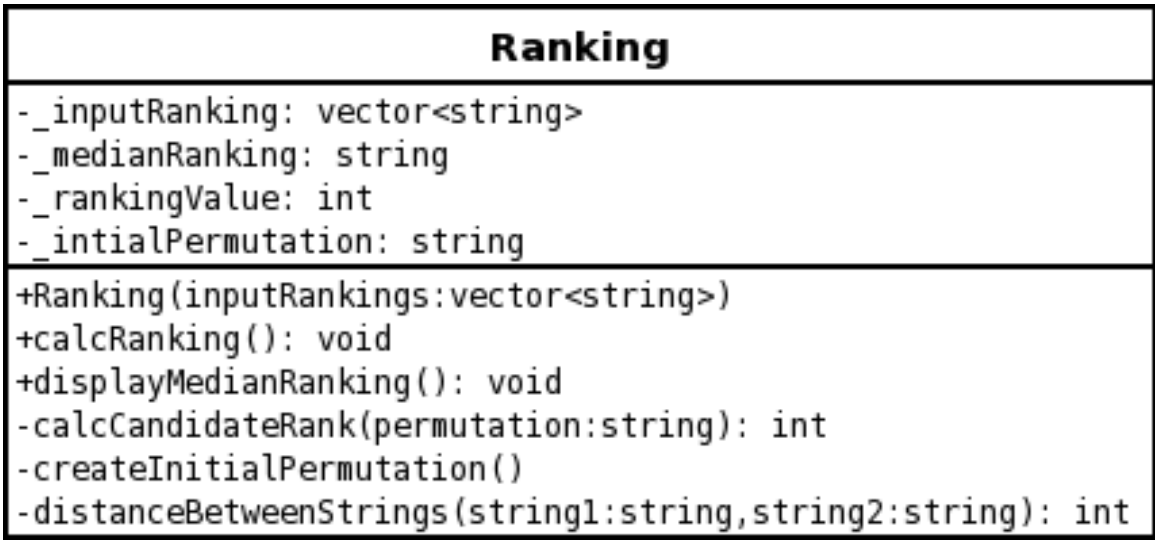

Figure 3 `Ranking` class diagram

Algorithm Descriptions

The algorithms for the program will be described using UML and specifically activity diagrams will be used. A few of the more trivial functions were left out of the algorithm description because the analysis of there algorithms would add much to the analysis of the program. One example of a function that was left out is `displayMedianRanking`, because it simply sends a string to standard output.

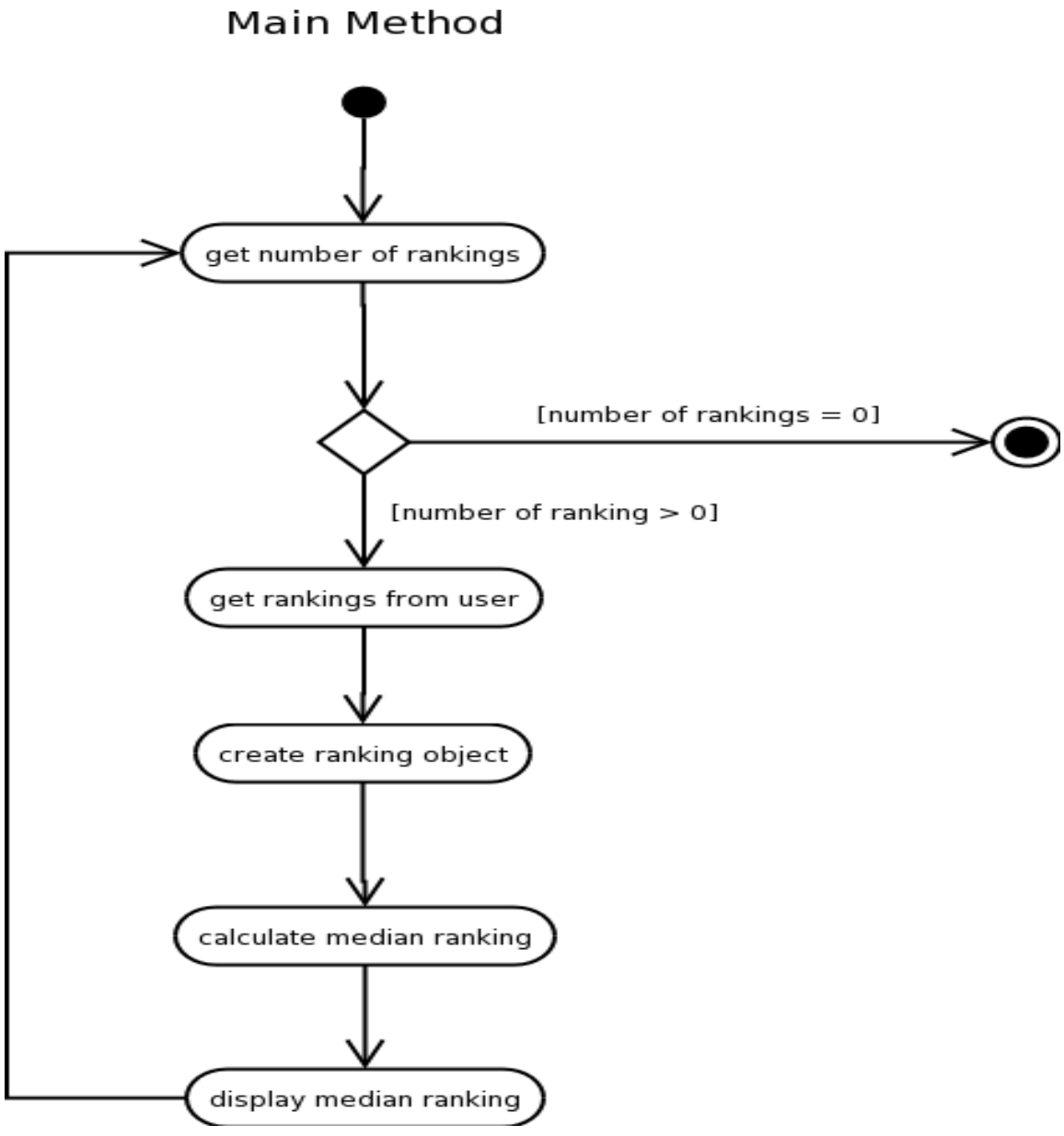

Figure 4 main

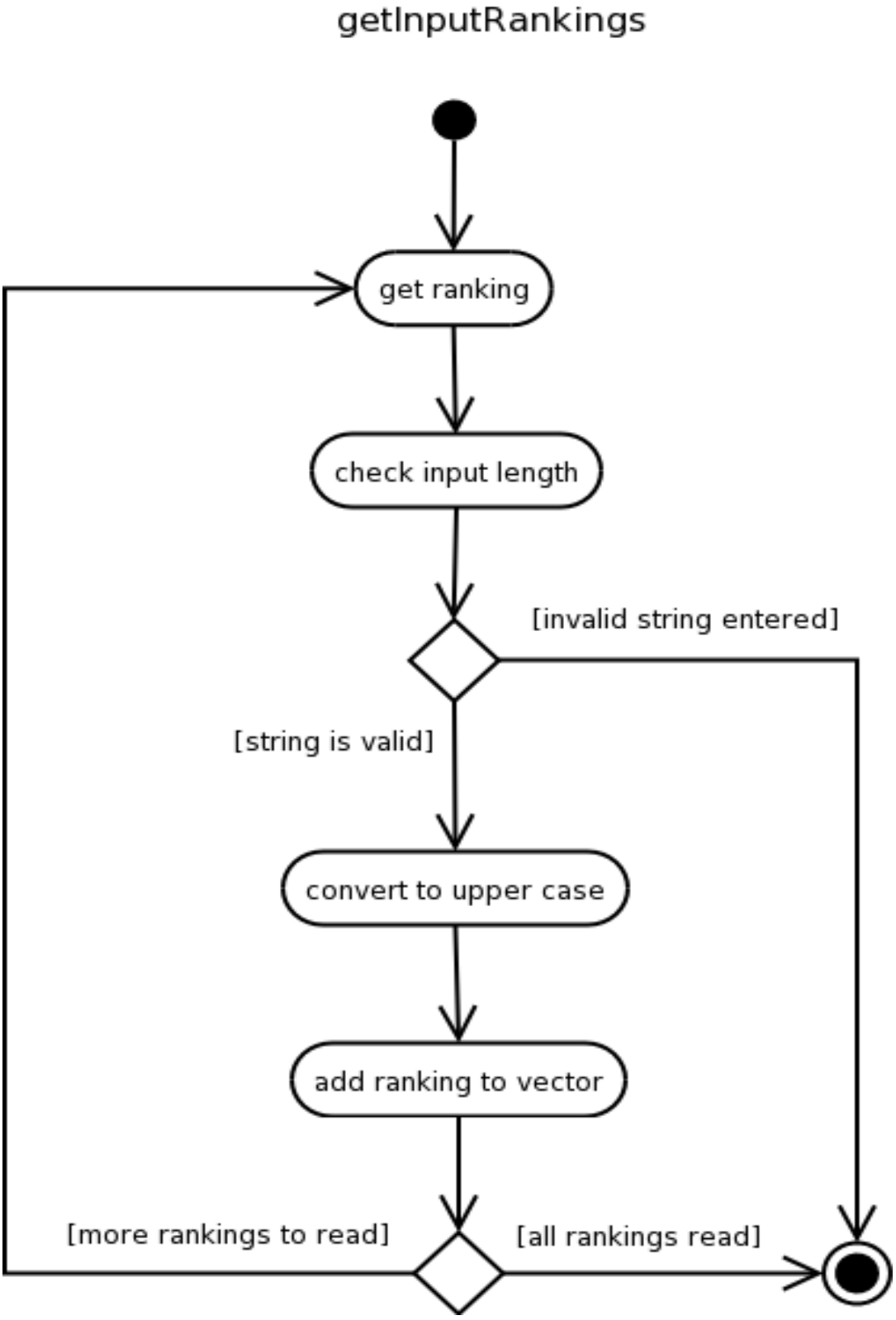

Figure 5 getInputRankings

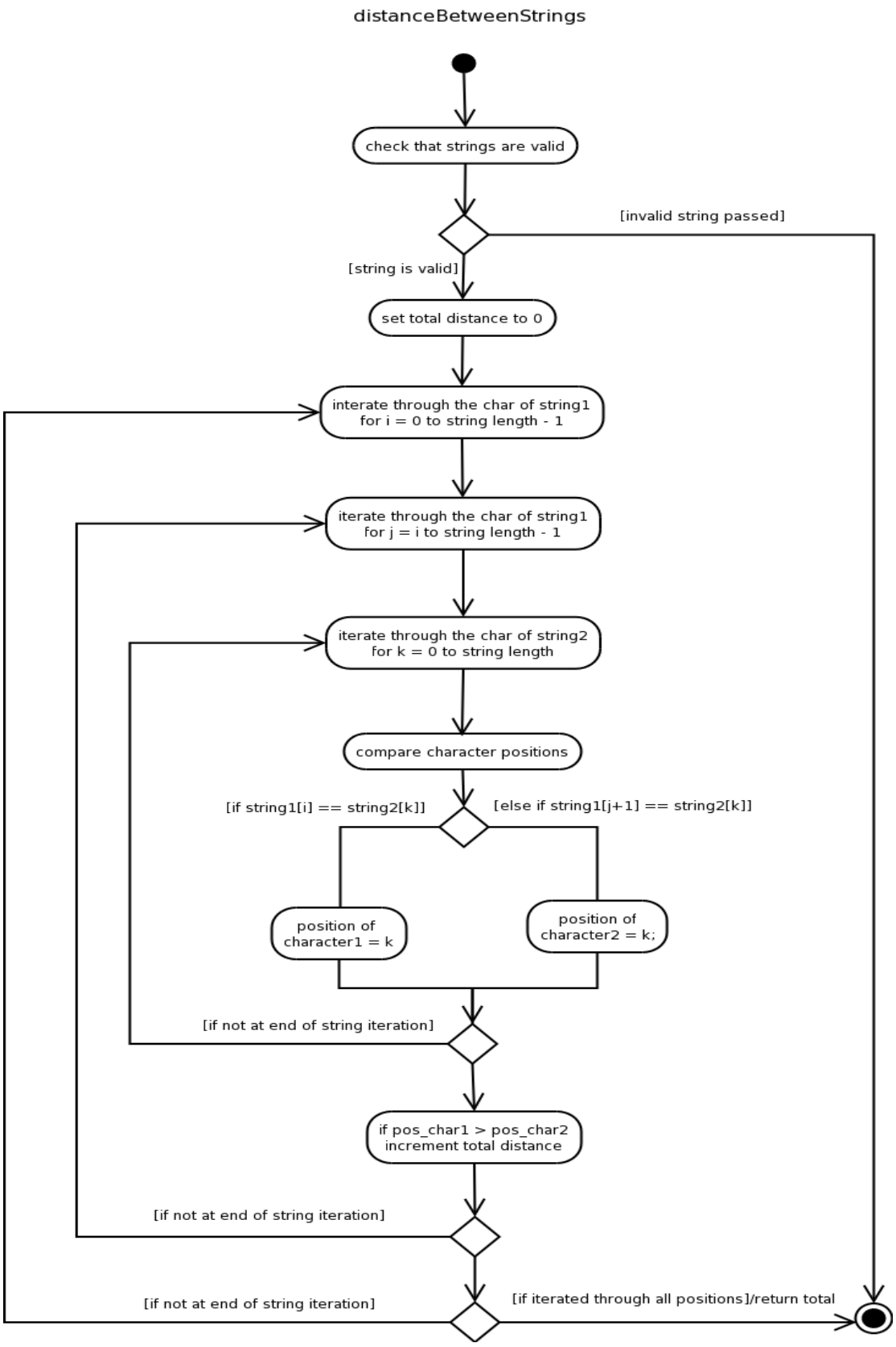

Figure 6 distanceBetweenStrings

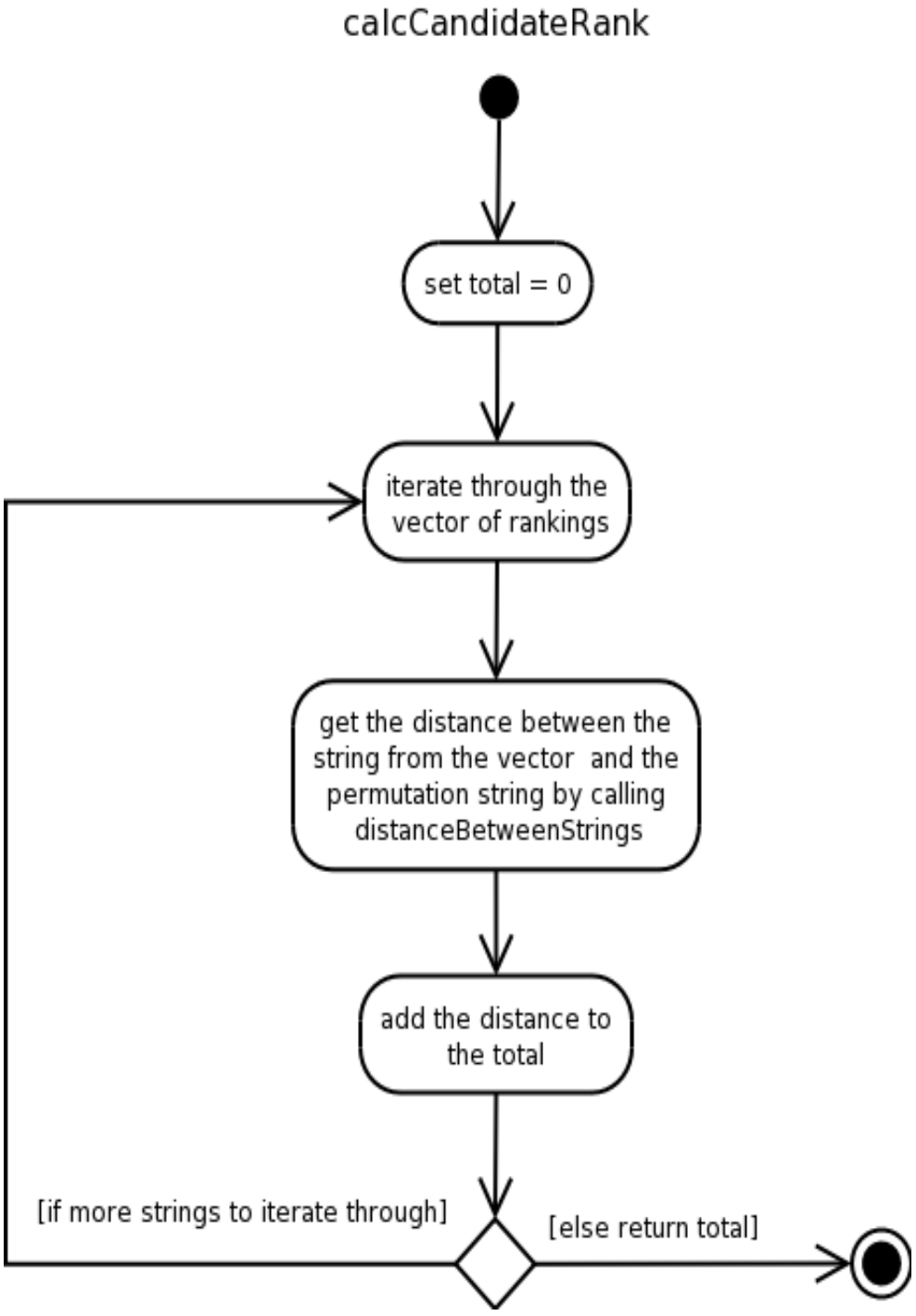

Figure 7 calCandidateRank

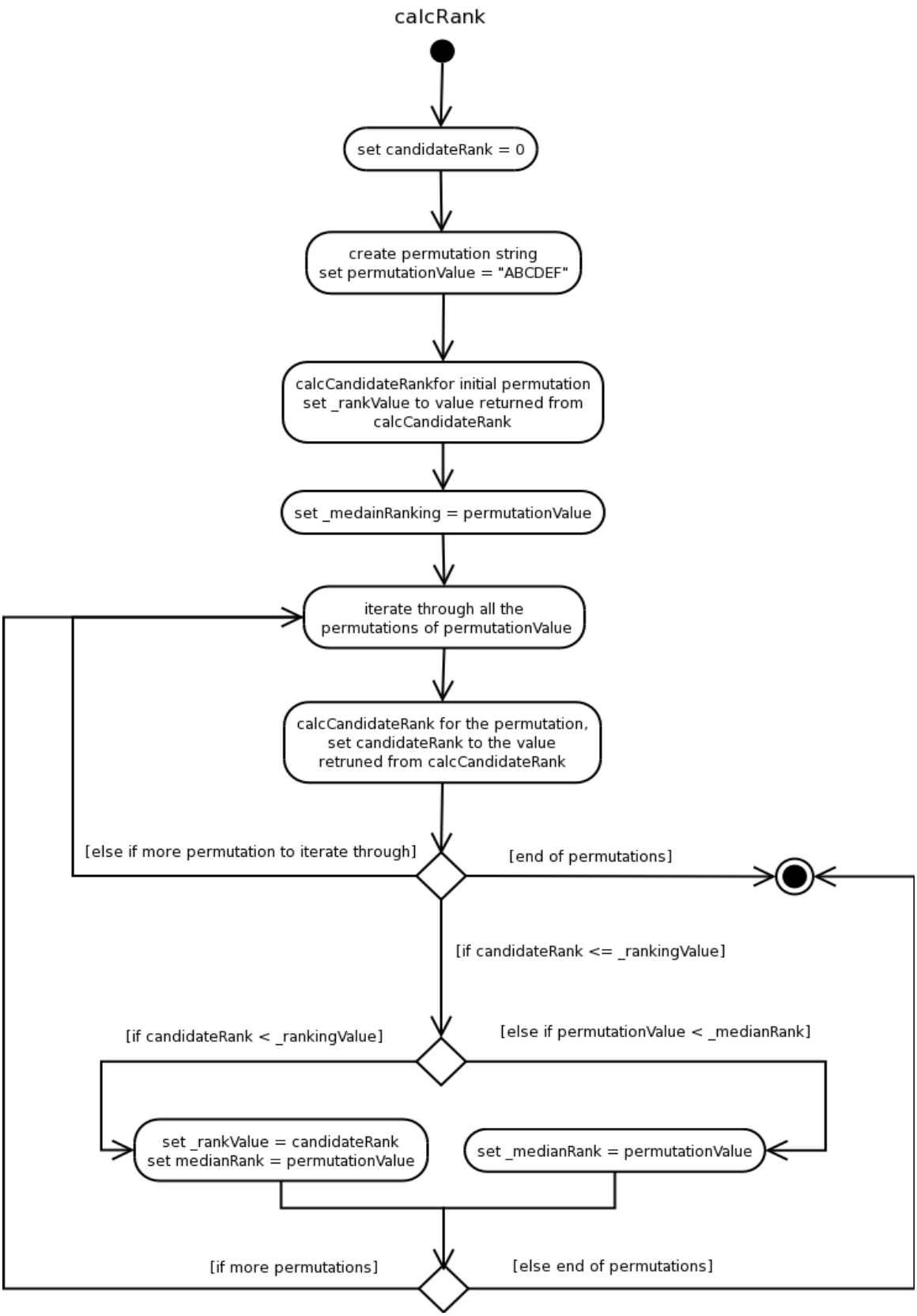

Figure 8 calcRank

Analysis

1) How many rankings are possible?

P(6,6) = 6*5*4*3*2*1 = 720

So there are 720 different permutations that are possible.

2) If the computer did 1/sec how long?

The maximum number of permutations that is possible is 10,000. So to determine the worst case run time for the program 10,000 permutations should be inputted into the program. If the computer could only calculate one the value of permutation per second, it would take 10,000 seconds to run the program. Which would take 2.77 hours to run the program in the worst case that all 10,000 sets of permutations were supplied?

3) Given a 1 gig machine what is the largest problem that you can solve?

A 1 gig machine is able to execute 1,000,000,000 instructions per second. It is difficult to give an exact amount of time that program would run in. In modern operating systems a number of processes are switched back and forth from, so it is unlikely that a process will get to run from start to finish with out being interrupted. Also, the amount of time need for IO to read in a file from the disk will vary but it shouldn't take that long to read in 10,000 short strings that contain the rankings.

Because of the mentioned problems, the analyses of the runtime will then just focus on the key operations that are used by the program. Computing the ranks is the most intensive part of the program the number of operations required to compute one ranking is (n-1)n/2. The number n in the case of the program will be six since the strings of six characters are read into the program and the maximum number of ranking that are read in is 10,000. Computing the rank will take 15 * 10,000 = 150,000 operations. If each operation it was estimated that 100 clock cycles were necessary the program would take 150,000 * 100 = 15,000,000 clock cycles. It would take approximately .015 seconds to calculate the median ranking from the input given.

After the median rank has been calculated the rank must be displayed to the user. One problem is that it is possible that there is more than one ranking that has the same median rank. One solution to this problem is to use a multi-map from the standard template library. When the value of the rank is calculated in the previous step it could be added to a multi-map. The key would be the value of the ranking and the value would be the string that was read in. The benefit of using a map is that the internal structure of the map is a tree to it allows for the quick retrieval of elements. After the median was calculated, the value that matches could be retrieved from the map in logarithmic amount of time. It is possible that there will be 720 keys so in the worst case to remove an element for the map would take 10 operations of traversing the tree. Also, some time will have to be accounted for the construction of the multi-map.

The described problem should easily be able to be run on a 1 gig machine. It should take less than a second to compute the median ranking and then to display it to the user.

If the number of letters used for the ranking was increased it would also have an impact on the performance of the program. The letter m will represent the number of teams that are given to be ranked. In order to determine the median ranking all permutations of the letters of m would have to be calculated. For instance, if the number of teams to be ranked was increased to 10, then 10! permutations would need to be compared to determine the median value. When m equals 10 teams to be ranked then 3,628,800 permutations would need to be compared. As the number m increases the number of permutations that needed to be calculated dramatically increases. The largest problem that could reasonably be solved on a one gig machine is 13. The teams could be ranked, which would cause 13! permutations to be calculated. If it was very important that 14 teams were used it would be possible but the user would have to wait hours for the median ranking to be calculated. For all practical purpose 13 is the largest number of teams that could be ranked. If more than 13 teams had to be ranked another method would need to be used to calculate the rankings.

Runtime and Storage Analysis

The worst case runtime and storage is used to determine the performance of the ranking program. The worst case runtime gives an upper bound on the amount of time needed to run the program. The worst case storage analysis gives on upper bound on the amount of memory that that will be needed to store the program.

For the ranking program the performance of the program depends on the length of the string of ranking and the number of rankings used to compute the median value. The variable ***n*** will represent the number of ranking in the input set of rankings. The variable ***m*** will represent the number of characters in a ranking string. A third variable will be introduced ***o***, which represents the number of set of input given to the program. The tables below give the worst case runtime and storage for the program.

Worst Case Runtime	
Function	***Worst Case Runtime***
Ranking	O(1)
~Ranking	O(1)
getNumberOfRankings	O(1)
getInputRankings	O(n * m)

Worst Case Runtime	
createPermutationString	O(m)
distanceBetweenStrings	O(m^3)
calcCandidateRank	O(m^3 * n)
calcRank	O(m! * m^3 * n)
displayRank	O(1)
main	O(m! * m^3 * n * o)

Table 1 Worst case runtime

Worst Case Storage	
Function	***Worst Case Storage***
Ranking	O(n*m)
~Ranking	O(1)
getNumberOfRankings	O(1)
getInputRankings	O(n * m)
createPermutationString	O(1)
distanceBetweenStrings	O(1)
calcCandidateRank	O(1)
calcRank	O(1)
displayRank	O(1)
main	O(n*m)

Table 2 Worst case storage

Test Plan

A number of sets of input for the program will be created to test that the program is working correctly.

The input sets will be similar to the one given in the program description. All the values

for the sets of rankings will be determined before the tests are run. The test will use a different variation of data to determine if the program is running correctly. If any problems arise and the program does not output the correct values for the sample input the program will be debugged to correct the problem.

The table below contains the input sets that will be used to test the program. The column input gives the set of input that is to be entered into the program. Expected output column gives the results that are expected when the test is run. Reason explains why the set of input was chosen to test the program.

Test #	Input	Expected Output	Reason
1	4 ABDCEF BACDEF ABCEDF ACBDEF	Median rank = ABCDEF Rank value = 4	To test the sample case given by the Instructor.
2	2 ABC ACB	Median rank = ABC Rank value = 1	To test the program when a different size of input is used, instead of strings of the length 6.
3	2 ABC CBA	Median rank = ABC Rank value = 3	To test two strings that can easily have there values calculated.
4	4 ABC ABC CBA CBA	Median rank = ABC Rank value = 6	To use the previous test and to test that rank value changes correctly, when more strings were entered with the intension of change the rank value.
5	5 ABC ABC CBA CBA BCA	Median rank = BCA Rank value = 6	To use the previous test and add another ranking so that the median rank will change.
6	3 ABCDEF ACBDEF ACBDEF	Median rank = ACBDEF Rank value = 1	To test that the median ranking will change.
7	3 FDECAB FDECAB FDECAB	Median rank = FDECAB Rank value = 0	To test that the program works correctly when given all equal strings.
8	ACBEFD ABCDEF ACBDFE	Median rank = ACBDEF Rank value = 4	To test another set of rankings similar to those given by the instructor.
9	ABCDEF ABC	The program should display and error and terminate	To test that strings of different length are not compared.
10	4 ABDCEF BACDEF ABCEDF acbdef	Median rank = ABCDEF Rank value = 4	To check that the program ignores the case of the letters when calculating the median rank.

Table 3 Test plan

Test Results

The tests given in the test plan were run to determine if the program was operating correctly. The table below gives the results of the test. All of the tests that were created in the test plan produced the correct output.

Test #	Input	Expected Output	Test Results
1	4 ABDCEF BACDEF ABCEDF ACBDEF	Median rank = ABCDEF Rank value = 4	Median rank = ABCDEF Rank value = 4
2	2 ABC ACB	Median rank = ABC Rank value = 1	Median rank = ABC Rank value = 1
3	2 ABC CBA	Median rank = ABC Rank value = 3	Median rank = ABC Rank value = 3
4	4 ABC ABC CBA CBA	Median rank = ABC Rank value = 6	Median rank = ABC Rank value = 6
5	5 ABC ABC CBA CBA BCA	Median rank = BCA Rank value = 6	Median rank = BCA Rank value = 6
6	3 ABCDEF ACBDEF ACBDEF	Median rank = ACBDEF Rank value = 1	Median rank = ACBDEF Rank value = 1
7	3 FDECAB FDECAB FDECAB	Median rank = FDECAB Rank value = 0	Median rank = FDECAB Rank value = 0
8	ACBEFD ABCDEF ACBDFE	Median rank = ACBDEF Rank value = 4	Median rank = ACBDEF Rank value = 4
9	4 ABDCEF BACDEF ABCEDF acbdef	Median rank = ABCDEF Rank value = 4	Median rank = ABCDEF Rank value = 4
10	ABCDEF ABC	The program should display an error and terminate	The program displayed an error and then terminated

Table 4 Test results

Sample Runs

```
4
ABDCEF
BACDEF
ABCEDF
ACBDEF
ABCDEF is the median ranking with a value of 4
2
ABC
ACB
ABC is the median ranking with a value of 1
2
ABC
CBA
ABC is the median ranking with a value of 3
4
ABC
ABC
CBA
CBA
ABC is the median ranking with a value of 6
5
ABC
ABC
CBA
CBA
BCA
BCA is the median ranking with a value of 6
3
ABCDEF
ACBDEF
ACBDEF
ACBDEF is the median ranking with a value of 1
3
FDECAB
FDECAB
FDECAB
FDECAB is the median ranking with a value of 0
3
ACBEFD
ABCDEF
ACBDFE
ACBDEF is the median ranking with a value of 4
4
ABDCEF
BACDEF
ABCEDF
acbdef
ABCDEF is the median ranking with a value of 4
2
ABCDEF
ABC
All strings must have the same length
Press any key to continue . . .
```