Project-1

(CSE-220-Data Structure and Algorithm)

Title: *The Sieve of Eratosthenes- A classic approach to finding Primes.*

Made By:
Ankit Vadehra(12BCE0282)
Poorva Arora(12BCE0277)

Abstract:
Implementing the Sieve of Eratosthenes, which is an efficient method to calculate primes, in different data structures, Namely: Linked List and Binary Search Tree. And, calculating the time comparison of the algorithm, and the 2 Data Structures.

Sieve of Eratosthenes, what is it ?

As wikipedia states :

“

In mathematics, the **Sieve of Eratosthenes**, is a simple, ancient algorithm for finding all prime numbers up to any given limit. It does so by iteratively marking as composite (i.e. not prime) the multiples of each prime, starting with the multiples of 2.

”

The sieve uses a different approach to find prime numbers, hence discarding the traditional functional approach that checked numbers inside 2-for-loops, and after checking each number individually it returns whether the number is a prime or not. At higher values this functional approach tends to take a significant amount of time, as well as ending up in taking a lot of process speed.

Instead of this the sieve strikes out the multiples of the initial primes, hence making out the process of finding out the remaining primes real easy.
Eg:

To find all the prime numbers less than or equal to 30, proceed as follows.

First generate a list of integers from 2 to 30:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

First number in the list is 2; cross out every 2nd number in the list after it (by counting up in increments of 2), i.e. all the multiples of 2:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~ 21 ~~22~~ 23 ~~24~~ 25 ~~26~~ 27 ~~28~~ 29 ~~30~~

Next number in the list after 2 is 3; cross out every 3rd number in the list after it (by counting up in increments of 3), i.e. all the multiples of 3:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8 9 10~~ 11 ~~12~~ 13 ~~14 15 16~~ 17 ~~18~~ 19 ~~20 21 22~~ 23 ~~24~~ 25 ~~26 27 28~~ 29 ~~30~~

Next number not yet crossed out in the list after 3 is 5; cross out every 5th number in the list after it (by counting up in increments of 5), i.e. all the multiples of 5:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8 9 10~~ 11 ~~12~~ 13 ~~14 15 16~~ 17 ~~18~~ 19 ~~20 21 22~~ 23 ~~24 25 26 27 28~~ 29 ~~30~~

Next number not yet crossed out in the list after 5 is 7; the next step would be to cross out every 7th number in the list after it, but they are all already crossed out at this point, as these numbers (14, 21, 28) are also multiples of smaller primes because 7*7 is greater than 30. The numbers left not crossed out in the list at this point are all the prime numbers below 30:

2 3 5 7 11 13 17 19 23 29

Algorithm :

To find all the prime numbers less than or equal to a given integer *n* by Eratosthenes' method:

1. Create a list of consecutive integers from 2 to *n*: (2, 3, 4, ..., *n*).
2. Initially, let *p* equal 2, the first prime number.
3. Starting from *p*, count up in increments of *p* and mark each of these numbers greater than *p* itself in the list. These will be multiples of *p*: 2*p*, 3*p*, 4*p*, etc.; note that some of them may have already been marked.
4. Find the first number greater than *p* in the list that is not marked. If there was no such number, stop. Otherwise, let *p* now equal this number (which is the next prime), and repeat from step 3.

When the algorithm terminates, all the numbers in the list that are not marked are prime.

Data Structure Use :

Using a proper Data Structure for the process of finding out primes using sieve of eratosthenes is very important. Each function brings with itself a different way of storing and extracting data and causing a significant change in the time taken to get to the result.
We need to check whether a candidate prime is the least element in the table (thereby finding it to be composite) , or find that the least element in the table is greater than our candidate prime, revealing that our candidate actually is prime.
Given the needs there are many Data Structures that can be used to solve the Sieve. We will give a comparitive analysis of the Sieve in Different Data-Structures namely a Binary Search Tree and Linked List.

Process:

Both the implementations use the same methodology to ask users for the Uper_Bound value and then fill in the data structure with values from 2-to-Upper_Bound.
Hence our Data Struture is filled with Natural-Numbers: (2,3,4,5,6....,N)[1] .
After this each Data Structure uses ddifferent techniques to facilitate the data, and t allow data manipulation tasks, and printing the list of primes.

[1] :Since the numbers 0 and 1 are considered neither prime nor composite we eclude them from the list of numbers.

Linked-list Implementation :

LOGIC:
1) Structure has Data to store a list of natural numbers
2) It has info to seperate primes from composites.(By default set as 0)
3) Create a Fucton in main to create a linked list
4)The Function takes in the number till which primes have to be tested.
"max"-the upper bound.
5)The list is created with naural number from 2 to max-1.
(0 and 1 are cosidered neither prime nor composite.)
6)The sieve of eratothenes algorithm is performed in main(), and all te multiples of
2,3,4.....,max are passed to the function prime()
7)the Prime() function takes in each mutiple value by call-by-value and traverses the list,
and searches for it.
8)When found the prime() fuction changes the value of info to -1. Thus telling us at a later stage that
the number is a composite.
9)The pop function is called next, and it traverses the list searching for the info variable and it shows all the
data which has info=0.
10)Thus we can take in any number, reate a dynamic list. work on it and then show the prime numbers,
11)Also the program has display() and display2() functions to show at any stage the state of the
list and data and info variables.

Class/Functions Description:

S. No	Class-Function/Function	Description
1	`void create(int number);`	Takes in the Upper-Bound of the main function passes it in the variable **number** and creates an ordered list starting from the number **2 to (number-1).** Also it assigns the default value of variable **info as 0.**
2	`void prime(int number);`	The numbers which are composites are passed

		from the main function and the fuction traverses the list and wherever the **data** variable is found having the composite number, the value of **info** is changed from **0 to -1** .
3	`void pop();`	The numbers having **info** value as 0, are all primes, and hece the **data** variable crrespondig to **0-info** value can be printed by general traversal.
4	`void display();`	Displays the **data elements,** ie. Th whole list.
5	`void display2();`	Displays **info** field, and hence at any point we can see th state of the assigning composites with **-1** value.
6	`int main();`	Takes in the boundary value for numer list, and passes it to **void create();** also the Sieve-algorithm runs in it, and passes composites to **void prime();**

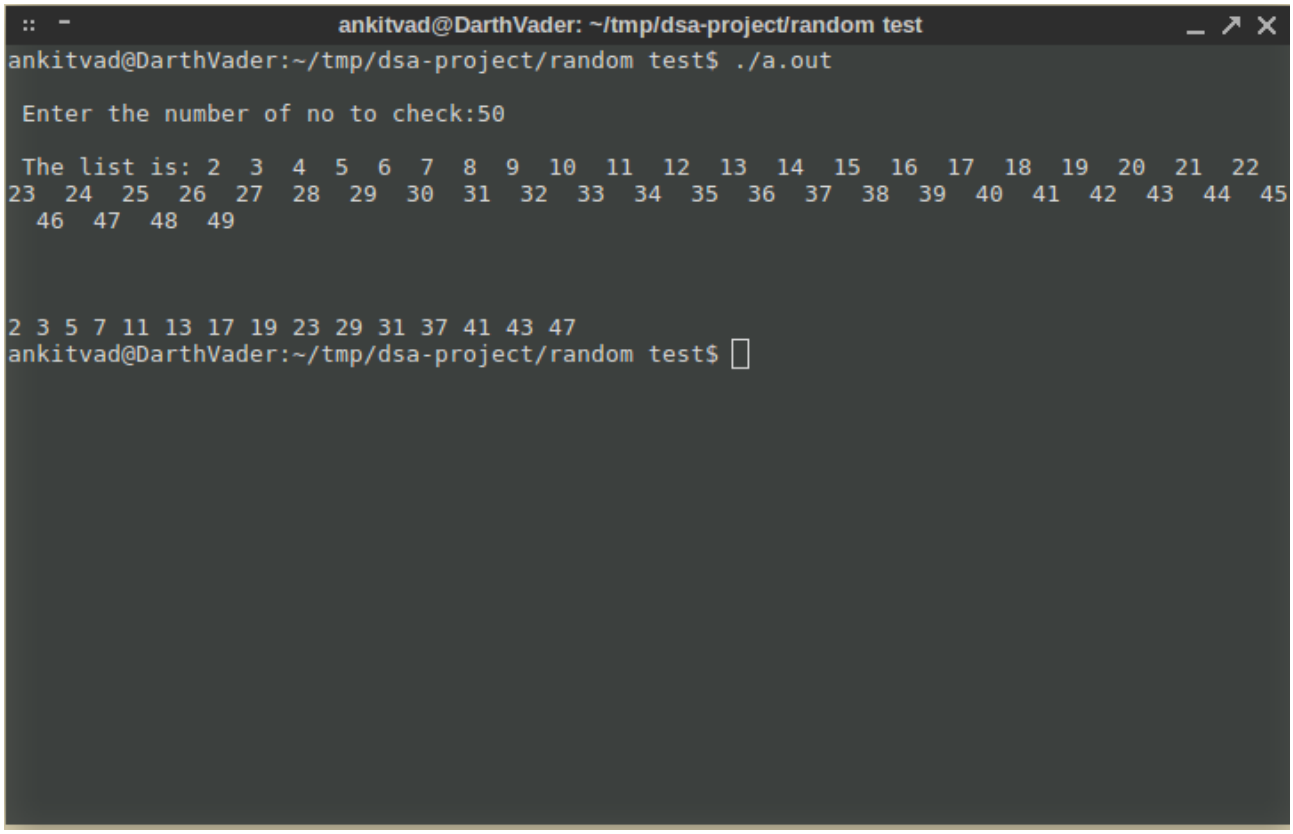
Code(Linked-List Implementation):

```
#include <iostream>
#include <stdlib.h>
using namespace std;

struct node
{
        int data;//Will store the Natural Numbers in succession
        int info;//a flag variable to store 0 and -1 to tell the type of number.
        node *next;
} *newptr = NULL, *head = NULL, *temp = NULL, *temp2 = NULL;

void create(int number)//Takes in the upper-bound from th main function
{int count=2; //Since 0 and 1 are not prime we start from 2.
        while((number-1)>=count)
        {
                newptr = new(node);
                //checking for memory
                newptr = new(node);
                if(newptr == NULL)
                {
                        cout<<"\n short of memory! \n EXITING";
                        return;
                }
                //Storing the value of natural numbers in data-variable
                newptr->data=number-1;
                newptr->info=0;//By default all numbers are taken prime.
                        newptr->next = NULL;
                        if(head==NULL)                  // no element in the
list
                            {
                                head = newptr;
```

```
                        }
                        else        //i.e. insert in beginning
                            {
                                newptr->next=head;
                                head=newptr;
                        }



         number=number-1;
        }


}

void prime(int number)
{       //traversing list to change info value(deleting-value), depending upon
the type.
          temp = head;
          while(temp!=NULL)
        {
                if(temp->data==number)
                  {
                        temp->info=-1;
                        break;
                     }
                temp=temp->next;
        }
}

void pop()
/*since the list tarts from numbers 2 and 3,
which are always prime we need not worry about the beggining conditions,
ie, changing head to the next. List won't become empty, EVER*/
{
        temp = head;
        while(temp !=NULL)              // by the end of this loop the temp
will point to node prior to that to be deleted
        {
                if(temp->info==0)
                {
                        cout<<temp->data<<" ";
                        //array[index]=temp2->data;
                //index++;

                }
            temp=temp->next;
    }


}

void display()
{cout<<"\n The list is: ";
        temp = head;
        while(temp != NULL)
        {
                cout<<temp->data<<"  ";
                temp = temp->next;
        }

}
```

```
//To Show info :
/*
void display2()
{temp = head;
        while(temp != NULL)
        {
                cout<<temp->info<<"  ";
                temp = temp->next;
        }
}
*/

int main()
{int max;
cout<<"\n Enter the number of no to check:";
cin>>max;
create(max);
display();
cout<<"\n \n";
//multiplier to be popped, later.
for (int p=2; p<=max; p++)
{int c=2;
                int mul = p*c;
                while(mul<max)
                        {   //cout<<mul<<"";
                                prime(mul);
                                //Passing value to list for update.
                                    c++;
                           mul = p*c;
                        }


}
//display();
cout<<"\n";
//display2();
cout<<"\n";
pop();
cout<<"\n";

//display();

return 0;
}
```

Output:



Explanation:

The Upper_Bound(max) Number entere by the user was : 50
The create list function created the list and the display function is used to view the list.
After that the main passes all the composites, which are subsequently marked -1 in the info.
The prime() function is called, which then prnts all the list->data elements, whose ->info value is 0, hence being prime.

Binary Search Tree:

In the binary search tree we have not used 2 variables, instead only a single variable to hold data that is the normal value of the number.
Then we use a loop to handle the Sieve of Eratothenes to pass multiples f numbers to the obj.delete() function of the Binary Search Tree to traverse the tree and then pop/delete out the multiples. In the end we use the functions to print the modified tree in a postorder, inorder and preorder format.

Class/Functions Description:

S. No	Class-Function/Function	Description
1	`void createTree(node** Tree, int element);`	Takes in the Upper-Bound of the main function passes it in the variable **number** and creates an ordered tree starting from the number **2 to (number-1).** It checks in all conditions of the tree, nodes and element magnitude greater or smaller than parent node and creates an ordered tree.
2	`void removeTree(node** Tree);`	It passes the created tree at the end of the program and deletes all the element and nodes, so as to free up memory and remve pointers.
3	`void preorder();`	It passes the tree and uses recursion technique to print the tree element in the **NODE-LEFT-RIGHT** format.
4	`void postorder();`	It passes the tree uses recurso techniques to print the tree elemets in **LEFT-RIGHT-NODE** format.
5	`void inorder();`	t passes the tree uses recurso techniques to print the tree elemets in **LEFT-NODE-RIGHT** format.
6	`void deleteNode();`	It takes in the composites from the sieve algorithm in function-**main()** and deletes the composites, realligning the tree checking all 3-base cases.
7	`int main();`	Takes in the boundary value and passes it to the create function to make the tree, also it passes the composites to the deleteNode funtions and the composites are deleted and the revised tree can be viewed as it is.

Code(Binary-Search-Tree Implementation):

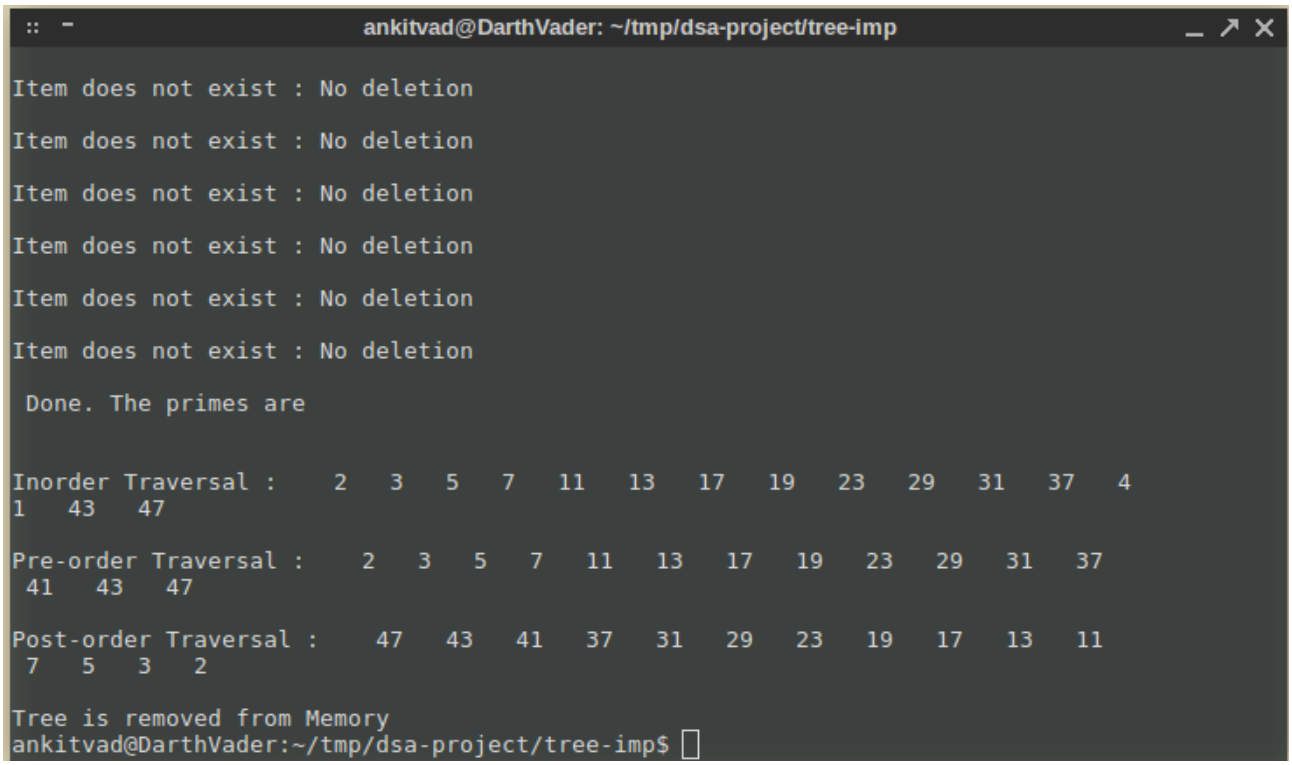
```
//Sieve of Eratothenes
//Tree Implementataion

#include <iostream>
#include <stdlib.h>
using namespace std;


struct node
{
        int data;
        struct node *left;
    struct node *right;
};

class BST{
    public:
    node *tree;
    BST()
    {
        tree=NULL;
    }
    void createTree(node **,int item);    //For Building Tree
    void removeTree(node **); //Remove tree from memory.
    void preOrder(node *);     //For Tree Traversal
    void inOrder(node *);
    void postOrder(node *);
    void deleteNode(int);
};

//it is used for inseting an single element in//a tree, but if calls more than
once will create tree.
void BST :: createTree(node **tree,int item)
{
    if(*tree == NULL){
        *tree = new node;
        (*tree)->data = item;
        (*tree)->left = NULL;
        (*tree)->right = NULL;
    }
    else{
        if( (*tree)->data > item)
            createTree( &((*tree)->left),item);
        else
            createTree( &((*tree)->right),item);
    }
}


void BST :: preOrder(node *tree){
    if( tree!=NULL){
        cout<<"   "<< tree->data;
        preOrder(tree->left);
        preOrder(tree->right);
    }
}
```

```
void BST :: inOrder(node *tree){
    if( tree!=NULL){
        inOrder( tree->left);
        cout<<"   "<< tree->data;
        inOrder(tree->right);
    }
}


void BST :: postOrder(node *tree){
    if( tree!=NULL){
        postOrder( tree->left);
        postOrder( tree->right);
        cout<<"   "<<tree->data;
    }
}

void BST :: removeTree(node **tree){
    if( (*tree) != NULL){
        removeTree( &(*tree)->left );
        removeTree( &(*tree)->right );
        delete( *tree );
    }
}

node * find_Insucc(node *curr)
{
    node *succ=curr->right; //Move to the right sub-tree.
    if(succ!=NULL)
    {
        while(succ->left!=NULL)    //If right sub-tree is not empty.
            succ=succ->left; //move to the left-most end.
    }
    return(succ);
}
void BST :: deleteNode(int item)
{
    node *curr=tree,*succ,*pred;
    int flag=0,delcase;
    //step to find location of node
    while(curr!=NULL && flag!=1)
    {
        if(item < curr->data)
        {
            pred = curr;
            curr = curr->left;
        }
        else if(item > curr->data)
        {
            pred = curr;
            curr = curr->right;
        }
        else
        { //curr->data = item
            flag=1;
        }
    }

    if(flag==0){
        cout<<"\nItem does not exist : No deletion\n";
        goto end;
    }
```

```
    //Decide the  case of deletion
    if(curr->left==NULL && curr->right==NULL)
        delcase=1; //Node has no child
    else if(curr->left!=NULL && curr->right!=NULL)
        delcase=3; //Node contains both the child
    else
        delcase=2; //Node contains only one child
        //Deletion Case 1
    if(delcase==1)
    {
        if(pred->left == curr) //if the node is a left child
            pred->left=NULL; //set pointer of its parentelse
            pred->right=NULL;
        delete(curr); //Return deleted node to the memory bank.
    }

    //Deletion Case 2
    if(delcase==2)
    {
        if(pred->left==curr)
        { //if the node is a left child
                if(curr->left==NULL)
                pred->left=curr->right;
            else
                pred->left=curr->left;
        }
        else
        { //pred->right=curr
                if(curr->left==NULL)
                pred->right=curr->right;
            else
                pred->right=curr->left;
        }
        delete(curr);
    }

    //Deletion case 3
    if(delcase==3)
    {
        succ = find_Insucc(curr); //Find the in_order successor//of the node.
        int item1 = succ->data;
        deleteNode(item1);  //Delete the inorder successor
        curr->data = item1; //Replace the data with the data in order successor.
    }
 end: ;
}


int main()
{
        BST obj;
    int choice;
    int height=0,total=0,n,item;
    node **tmp;
    cout<<"\nHow many nodes do you want to enter : ";
    cin>>n;
    for(int i=2;i<n;i++)
    {
       obj.createTree(&obj.tree,i);
    }
```

```
 for (int p=2; p<=n; p++)
 {int c=2;
 int mul = p*c;
  while(mul<n)
  {
    obj.deleteNode(mul);
    //Passing value to list for update.
        c++;
    mul = p*c;
   }
  }

cout<<"\n Done. The primes are \n";
cout<<"\n\nInorder Traversal : ";
obj.inOrder(obj.tree);

cout<<"\n\nPre-order Traversal : ";
obj.preOrder(obj.tree);

cout<<"\n\nPost-order Traversal : ";
obj.postOrder(obj.tree);


obj.removeTree(&obj.tree);
cout<<"\n\nTree is removed from Memory";
cout<<"\n";
return 0;
}
```

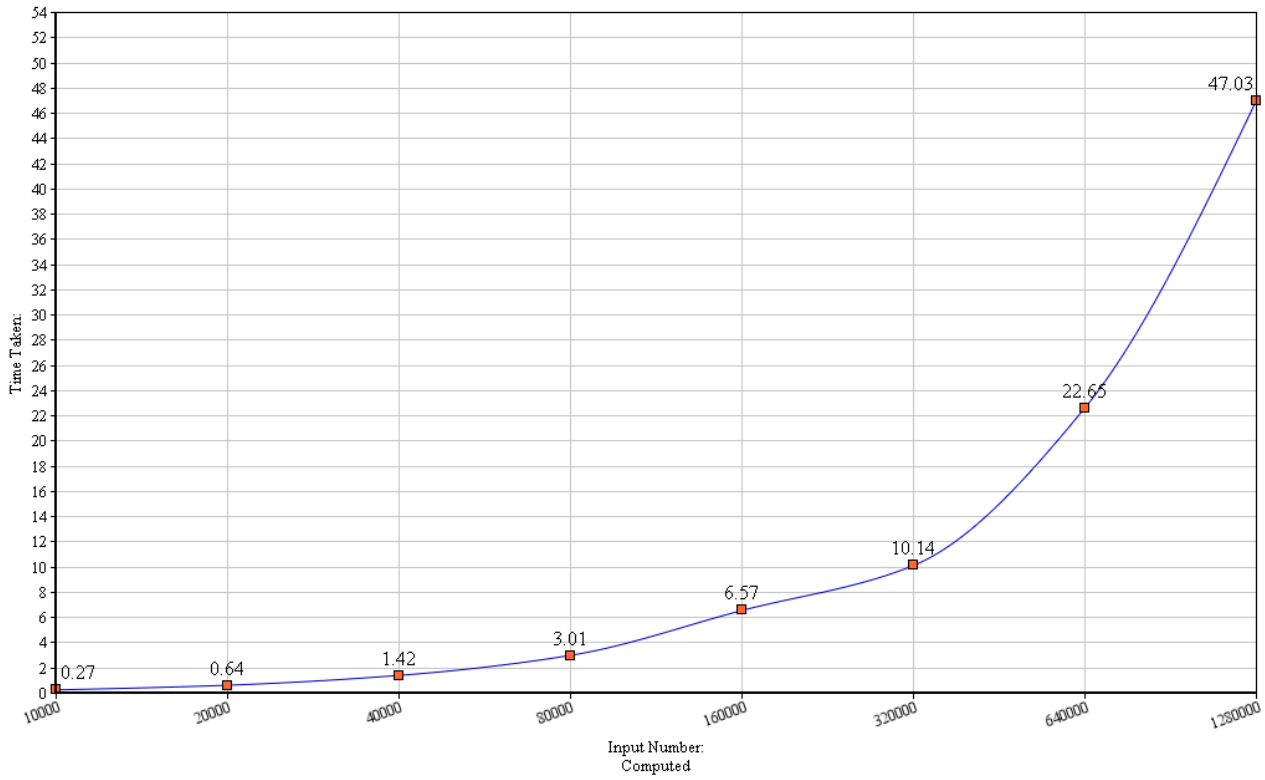
Output:



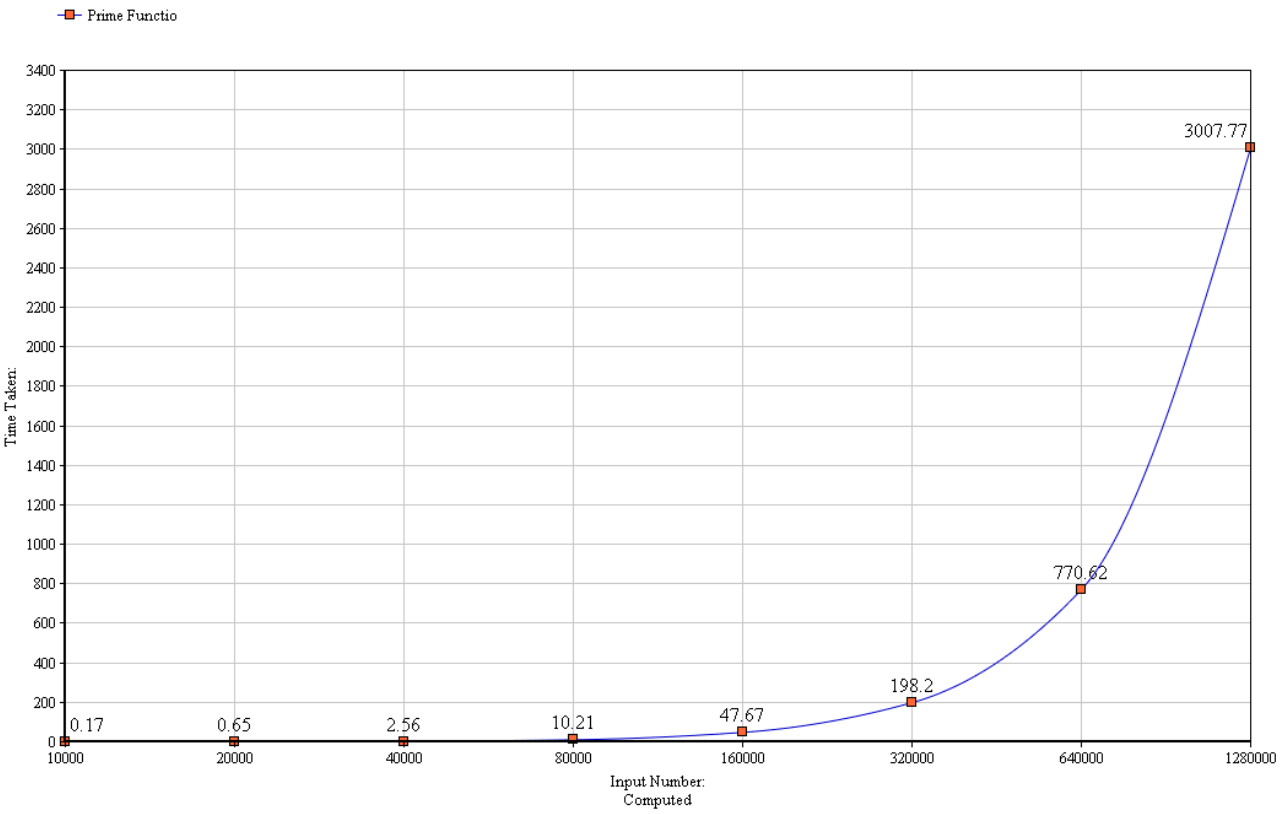
Time Comparison's:

1) Sieve Algorithm vs Prime-Functional-Approach :(~1000000)

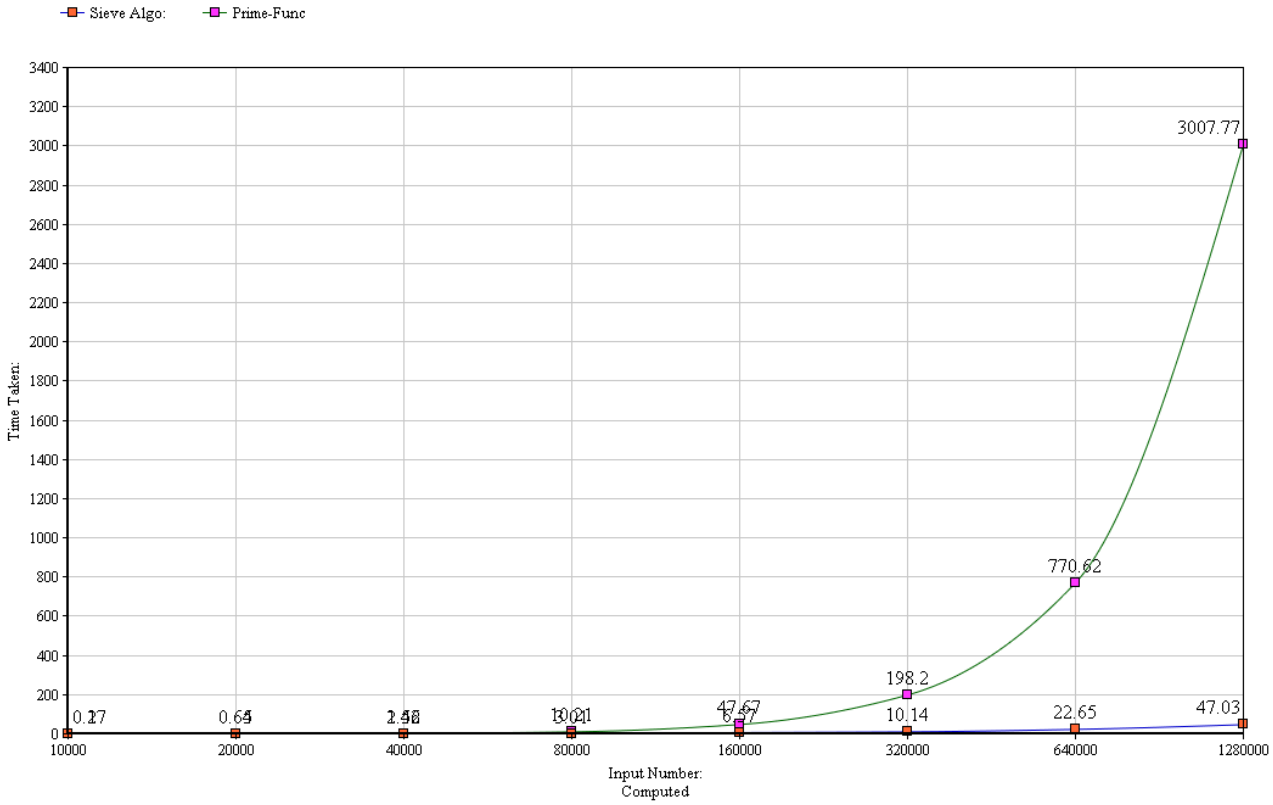
Input Value	Sieve Algorithm(Time seconds)	2-For loop functional approach(Time secods)
10000	0.27	0.17
20000	0.64	0.65
40000	1.42	2.56
80000	3.01	10.21
160000	6.57	47.67
320000	10.14	198.2
640000	22.65	770.62
1280000	47.03	3007.77



(Sieve Algorithm)



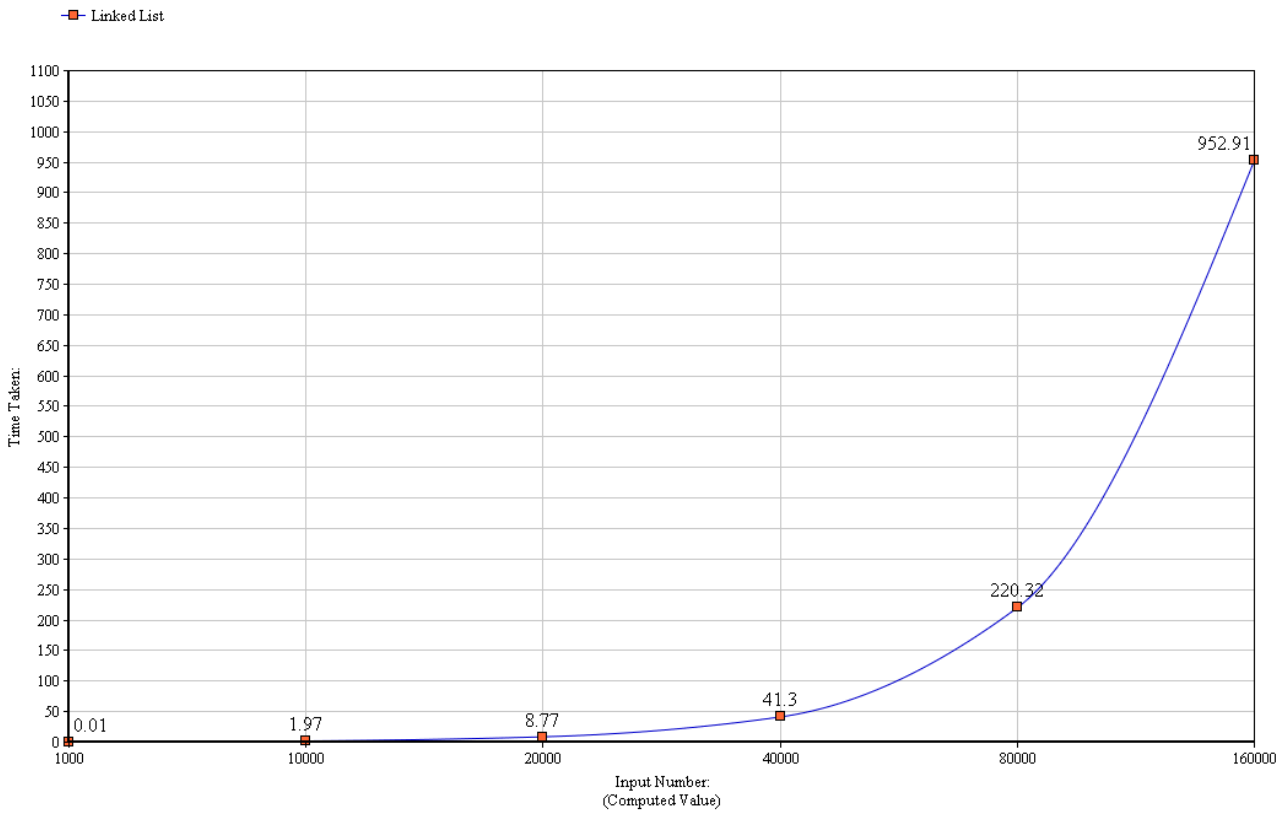
(Prime Function Algorithm)



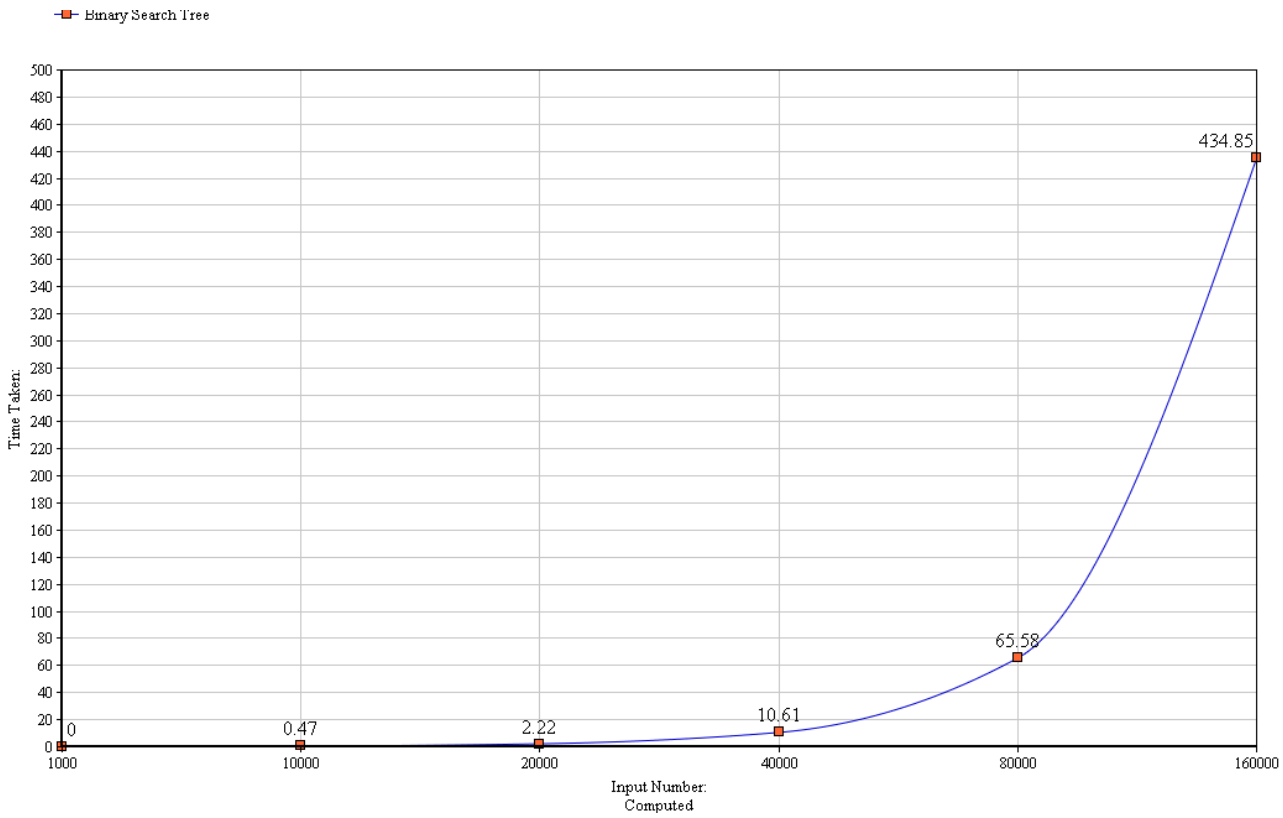
(Prime vs Sieve Algo)

2) Linked-List Implementation vs Binary Search Tree

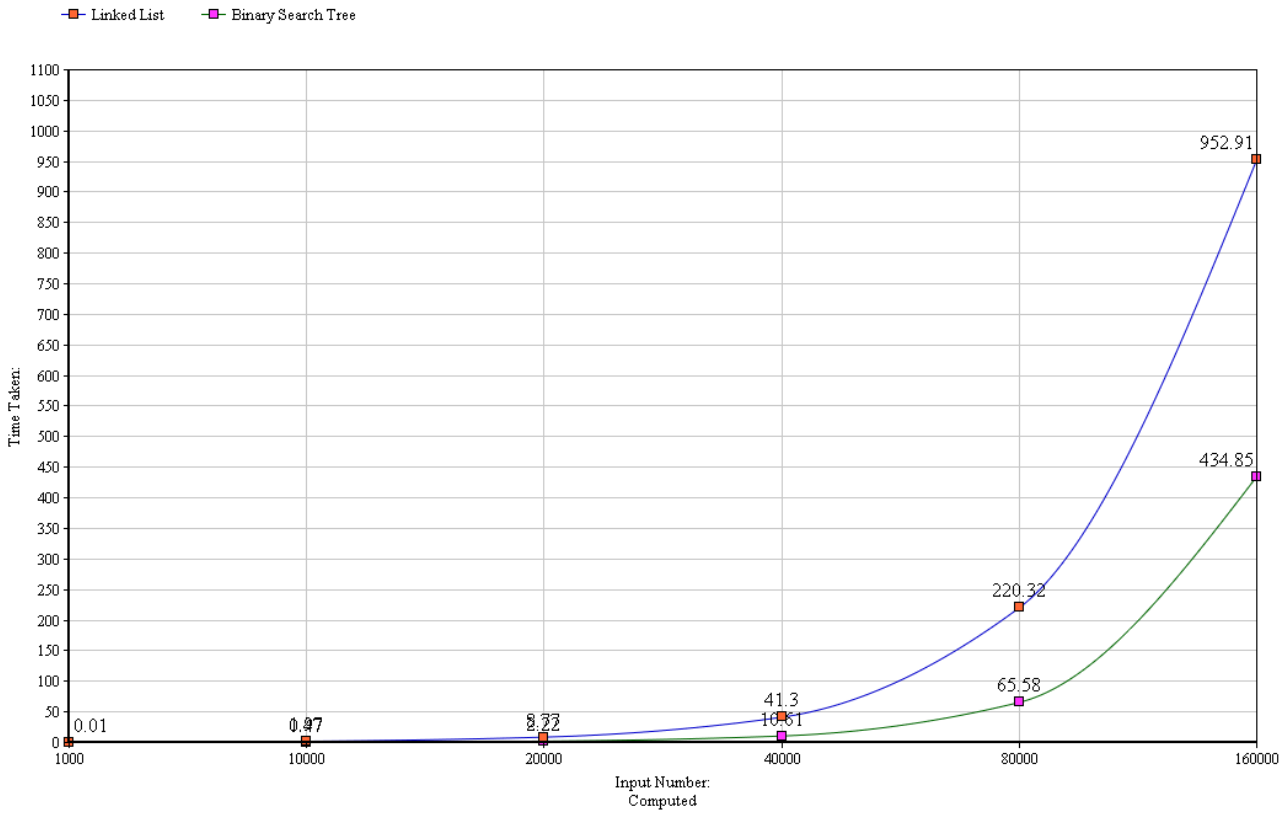
Input Value	Linked List	Binary Search Tree
1000	0.01	0.00
10000	1.97	0.47
20000	8.77	2.22
40000	41.3	10.61
80000	220.32	65.58
160000	952.91	434.85



(Linked List Implementation)



(Binary Search Tree)



(Linked List vs BST)

Refereneces:

1. Wikipedia [http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes]
2. http://www.cs.tufts.edu/~nr/comp150fp/archive/melissa-oneill/Sieve-JFP.pdf
3. http://stackoverflow.com/
4. http://pages.cs.wisc.edu/~vernon/cs367/notes/3.COMPLEXITY.html
5. Introduction To Algorithm, 3rd Edition, CLRS