**Advance Algorithm Laboratory Work**

Submitted By

**Junaid Altaf Girkar**

**60004190057**

**A/A4**

**TE Computer Engineering**

**Implementation of a Framework to identify the data structure and perform Amortized Analysis on Any Selected Program.**

**Theory**:

In computer science, amortized analysis is a method for analyzing a given algorithm's complexity, or how much of a resource, especially time or memory, it takes to execute. The motivation for amortized analysis is that looking at the worst-case run time can be too pessimistic. Instead, amortized analysis averages the running times of operations in a sequence over that sequence.  As a conclusion: “Amortized analysis is a useful tool that complements other techniques such as worst-case and average-case analysis.“

For a given operation of an algorithm, certain situations (e.g., input parametrizations or data structure contents) may imply a significant cost in resources, whereas other situations may not be as costly. The amortized analysis considers both the costly and less costly operations together over the whole sequence of operations. This may include accounting for different types of input, length of the input, and other factors that affect its performance.

Amortized analysis requires knowledge of which series of operations are possible. This is most commonly the case with data structures, which have a state that persists between operations. The basic idea is that a worst-case operation can alter the state in such a way that the worst case cannot occur again for a long time, thus "amortizing" its cost.

METHODS OF AMORTIZED ANALYSIS

There are generally three methods for performing amortized analysis: the aggregate method, the accounting method, and the potential method. All of these give correct answers; the choice of which to use depends on which is most convenient for a particular situation.[3]

* **Aggregate** analysis determines the upper bound T(n) on the total cost of a sequence of n operations, then calculates the amortized cost to be T(n) / n.
* The **accounting** method is a form of aggregate analysis which assigns to each operation an amortized cost which may differ from its actual cost. Early operations have an amortized cost higher than their actual cost, which accumulates a saved "credit" that pays for later operations having an amortized cost lower than their actual cost. Because the credit begins at zero, the actual cost of a sequence of operations equals the amortized cost minus the accumulated credit. Because the credit is required to be non-negative, the amortized cost is an upper bound on the actual cost. Usually, many short-running operations accumulate such credit in small increments, while rare long-running operations decrease it drastically.
* The **potential** method is a form of the accounting method where the saved credit is computed as a function (the "potential") of the state of the data structure. The amortized cost is the immediate cost plus the change in potential.

**Code**:

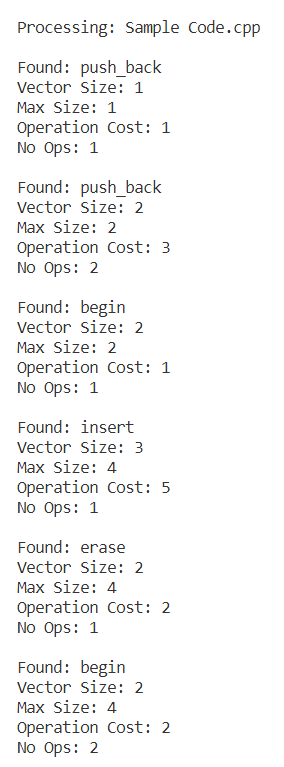
Amortized.cpp

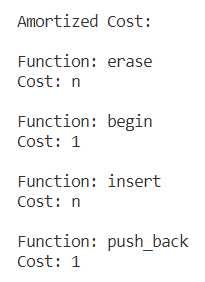
| #include <iostream> #include <string> #include <cstring> #include <fstream> #include <cstdlib> #include <unordered\_map>  bool searchLine(std::string line, std::string key) {  return line.find(key) != std::string::npos; }  class DataStructure {  public:  virtual void process(std::ifstream &file, std::string &line) {}  virtual void printCost() {} };  class stackAnalyzer : public DataStructure { };  class queueAnalyzer : public DataStructure { };  class mapAnalyzer : public DataStructure { };  class vectorAnalyzer : public DataStructure {  private:  std::string variable\_name;   int constant\_cost = 1;  int vector\_size = 0;  int max\_size = 1;   // Change in number of elements  // 1: Add  // -1: Remove  // 0: No change  std::unordered\_map<std::string, int> function\_calls{  {"push\_back", 1},  {"insert", 1},  {"erase", -1},  {"find", 0},  {"begin", 0},  };   // Cost per function  std::unordered\_map<std::string, int> incurred\_costs{  {"push\_back", 0},  {"insert", 0},  {"erase", 0},  {"find", 0},  {"begin", 0},  };   // No of times function is called  std::unordered\_map<std::string, int> function\_ops{  {"push\_back", 0},  {"insert", 0},  {"erase", 0},  {"find", 0},  {"begin", 0},  };   // Time Complexity  std::unordered\_map<std::string, int \*> function\_cost{  {"push\_back", &this->constant\_cost},  {"insert", &this->vector\_size},  {"erase", &this->vector\_size},  {"find", &this->vector\_size},  {"begin", &this->constant\_cost},  };   public:  vectorAnalyzer(std::string var\_name = "vec") {  this->variable\_name = var\_name;  }   void incurCost(std::pair<std::string, int> function) {  if (this->vector\_size > max\_size) {  this->incurred\_costs[function.first] += this->max\_size;  this->max\_size = this->max\_size \* 2;  }  this->incurred\_costs[function.first] += \*this->function\_cost[function.first];  }   void search(std::string line) {  for (auto function : function\_calls) {  if (searchLine(line, function.first)) {  this->vector\_size += function.second;  this->incurCost(function);  this->function\_ops[function.first]++;  std::cout << "\nFound: " << function.first << std::endl;  std::cout << "Vector Size: " << this->vector\_size << std::endl;  std::cout << "Max Size: " << this->max\_size << std::endl;  std::cout << "Operation Cost: " << this->incurred\_costs[function.first] << std::endl;  std::cout << "No Ops: " << this->function\_ops[function.first] << std::endl;  }  }  }   void process(std::ifstream &file, std::string &line) {  while (getline(file, line)) {  this->search(line);  }  }   void printCost() {  for (auto function : function\_ops) {  if (function.second) {  std::cout << "\nFunction: " << function.first << std::endl;   int val = this->incurred\_costs[function.first] / function.second;  std::cout << "Cost: " << (val / this->vector\_size ? "n" : "1") << std::endl;  }  }  } };  DataStructure \*identifyDataStructure(std::ifstream &file, std::string &line) {  // can also use regex for better search + getting variable name  while (getline(file, line)) {  if (searchLine(line, "vector")) return new vectorAnalyzer();  if (searchLine(line, "stack")) return new stackAnalyzer();  if (searchLine(line, "queue")) return new queueAnalyzer();  if (searchLine(line, "map")) return new mapAnalyzer();  }  return nullptr; }  int main(int argc, char \*\*argv) {  std::cout << "\nProcessing: " << argv[1] << std::endl;   std::ifstream file(argv[1]);  std::string line;   DataStructure \*ds = identifyDataStructure(file, line);  if (ds) {  ds->process(file, line);  std::cout << "\nAmortized Cost: " << std::endl;  ds->printCost();  } else {  std::cout << "Data Structure Not Supported" << std::endl;  }   file.close();  delete ds; } |
| --- |

SampleCode.cpp

| #include <iostream> #include <vector>  int main() {  std::vector<int> vec;   vec.push\_back(1);  vec.push\_back(2);   std::cout << vec.front() << std::endl;   vec.insert(vec.begin() + 1, 4);  vec.erase(vec.begin(), vec.begin() + 1); } |
| --- |

Output:





Observations:

The amortized cost of the SampleCode.cpp file is lesser than the worst case scenario cost.

Conclusion:

Thus amortized analysis gives us detailed insights of the data structures and the amortized cost is right and is always lesser than the cost that would incur in the worst case scenario.

References:

<https://www.geeksforgeeks.org/analysis-algorithm-set-5-amortized-analysis-introduction/>

<https://www.tutorialspoint.com/Amortized-Analysis>

<https://en.wikipedia.org/wiki/Amortized_analysis#:~:text=In%20computer%20science%2C%20amortized%20analysis,time%20can%20be%20too%20pessimistic>.

**Implementation of Amortized Analysis (Aggregate Method)**

**Theory**:

Amortized analysis is a method of analyzing algorithms that can help us determine an upper bound on the complexity of an algorithm. This is particularly useful when analyzing operations on data structures, when they involve slow, rarely occurring operations and fast, more common operations. With this disparity between each operations’ complexity, it is difficult to get a tight bound on the overall complexity of a sequence of operations using worst-case analysis. Amortized analysis provides us with a way of averaging the slow and fast operations together to obtain a tight upper bound on the overall algorithm runtime. Here we will consider a simplified version of the hash table problem, and show that a sequence of n insert operations has overall time O(n).

## Aggregate Method

The aggregate method is used to find the total cost. If we want to add a bunch of data, then we need to find the amortized cost by this formula.

For a sequence of n operations, the cost is −



Let *ci* be the cost of the *i*-th insertion:

| **ci = i if i−1 is a power of 2  1 otherwise** |
| --- |

Let's consider the size of the table *si* and the cost *ci* for the first few insertions in a sequence:

| **i 1 2 3 4 5 6 7 8 9 10 si 1 2 4 4 8 8 8 8 16 16 ci 1 2 3 1 5 1 1 1 9 1** |
| --- |

Alternatively we can see that *ci*=1+*di* where *di* is the cost of doubling the table size. That is

| **di = i−1 if i−1 is a power of 2  0 otherwise** |
| --- |

Then summing over the entire sequence, all the 1's sum to *O*(*n*), and all the *di* also sum to *O*(*n*). That is,

| Σ1≤i≤n ci ≤ n + Σ0≤j≤m 2j−1 |
| --- |

where *m* = log(*n* − 1). Both terms on the right hand side of the inequality are *O*(*n*), so the total running time of *n* insertions is *O*(*n*).

Code:



| #include <iostream> #include <bits/stdc++.h> using namespace std;  void print(int arr[], int n){  for(int i = 0; i<n; i++){  cout<<arr[i]<<" ";  }  cout<<endl; }  int main(){   int size = 1;  int count = 0;  int arr[size];  int \*p = arr;  int cost = 0;    while(1){  int n;  cout<<"Enter the number you wish to insert in the dynamic table: ";  cin>>n;  if(count<size){  \*(p + count) = n;  count+=1;  cout<<p<<endl;  print(p, count);  cost+=1;  }else{  //double array  cout<<"Double"<<endl;  int \*new\_arr = new int[size\*2];  for(int i=0; i<count; i++){  // cout<<\*(p+i)<<endl;  new\_arr[i] = \*(p+i);  }  cost+=count+1;  size\*=2;  new\_arr[count] = n;  count+=1;  p = new\_arr;  cout<<p<<endl;  print(p, count);  }  cout<<"Ammortized Cost: "<<cost<<endl;  cout<<endl;  }  return 0; } |
| --- |

**Output**:



| Enter the number you wish to insert in the dynamic table: 5 0x7fff6e35bc10 5  Ammortized Cost: 1  Enter the number you wish to insert in the dynamic table: 10 Double 0x55c2abae66d0 5 10  Ammortized Cost: 3  Enter the number you wish to insert in the dynamic table: 2 Double 0x55c2abae66f0 5 10 2  Ammortized Cost: 6  Enter the number you wish to insert in the dynamic table: 19 0x55c2abae66f0 5 10 2 19  Ammortized Cost: 7  Enter the number you wish to insert in the dynamic table: 5 Double 0x55c2abae6710 5 10 2 19 5  Ammortized Cost: 12  Enter the number you wish to insert in the dynamic table: 23 0x55c2abae6710 5 10 2 19 5 23  Ammortized Cost: 13 |
| --- |

**Observations**:

The aggregate method of amortized analysis is used to find the amortized cost which it does by calculating the average of individual costs.

**Conclusion**:

The aggregate method of amortized analysis gives the amortized cost which might be higher or lower than the individual cost of some steps but is overall lesser than the total worst case scenario and the sequence of n insert operations has overall time O(n).

**References**:

<https://www.cs.cornell.edu/courses/cs3110/2012sp/lectures/lec21-amortized/lec21.html>

<https://www.tutorialspoint.com/Amortized-Analysis>

**Implementation of Amortized Analysis (Accounting Method)**

**Theory**:

Amortized analysis is a method of analyzing algorithms that can help us determine an upper bound on the complexity of an algorithm. This is particularly useful when analyzing operations on data structures, when they involve slow, rarely occurring operations and fast, more common operations. With this disparity between each operations’ complexity, it is difficult to get a tight bound on the overall complexity of a sequence of operations using worst-case analysis. Amortized analysis provides us with a way of averaging the slow and fast operations together to obtain a tight upper bound on the overall algorithm runtime. Here we will consider a simplified version of the hash table problem, and show that a sequence of n insert operations has overall time O(n).

### Accounting (Banker's) Method

The aggregate method directly seeks a bound on the overall running time of a sequence of operations. In contrast, the accounting method seeks to find a *payment* of a number of extra time units charged to each individual operation such that the sum of the payments is an upper bound on the total actual cost. Intuitively, one can think of maintaining a bank account. Low-cost operations are charged a little bit more than their true cost, and the surplus is deposited into the bank account for later use. High-cost operations can then be charged less than their true cost, and the deficit is paid for by the savings in the bank account. In that way we spread the cost of high-cost operations over the entire sequence. The charges to each operation must be set large enough that the balance in the bank account always remains positive, but small enough that no one operation is charged significantly more than its actual cost.

We emphasize that the extra time charged to an operation does not mean that the operation really takes that much time. It is just a method of accounting that makes the analysis easier.

If we let *c'i* be the charge for the *i*-th operation and *ci* be the true cost, then we would like

| *Σ1≤i≤n ci ≤ Σ1≤i≤n c'i* |
| --- |

for all *n*, which says that the ***amortized time*** Σ*1≤i≤n c'i* for that sequence of *n* operations is a bound on the true time Σ*1≤i≤n ci*.

Considering the example of the extensible array. Say it costs 1 unit to insert an element and 1 unit to move it when the table is doubled. Clearly a charge of 1 unit per insertion is not enough, because there is nothing left over to pay for the moving. A charge of 2 units per insertion again is not enough, but a charge of 3 appears to be:

| **i 1 2 3 4 5 6 7 8 9 10 si 1 2 4 4 8 8 8 8 16 16 ci 1 2 3 1 5 1 1 1 9 1 c'i 3 3 3 3 3 3 3 3 3 3 bi 2 3 3 5 3 5 7 9 3 4** |
| --- |

where bi is the balance after the i-th insertion.

In fact, this is enough in general. Let m refer to the m-th element inserted. The three units charged to m are spent as follows:

* One unit is used to insert m immediately into the table.
* One unit is used to move m the first time the table is grown after m is inserted.
* One unit is donated to element m − 2k, where 2k is the largest power of 2 not exceeding m, and is used to move that element the first time the table is grown after m is inserted.

Now whenever an element is moved, the move is already paid for. The first time an element is moved, it is paid for by one of its own time units that was charged to it when it was inserted; and all subsequent moves are paid for by donations from elements inserted later.

In fact, we can do slightly better, by charging just 1 for the first insertion and then 3 for each insertion after that, because for the first insertion there are no elements to copy. This will yield a zero balance after the first insertion and then a positive one thereafter.

**Code**:

| #include <iostream> #include <bits/stdc++.h> using namespace std;  void print(int arr[], int n){  for(int i = 0; i<n; i++){  cout<<arr[i]<<" ";  }  cout<<endl; }  int main(){  int size = 1;  int count = 0;  int arr[size];  int \*p = arr;  int account = 0;   cout<<"Initial account balance = "<<account<<endl<<endl;   while(1){  int n;  cout<<"Enter the number you wish to insert in the dynamic table: ";  cin>>n;  account += 3; //adding 3 in account   if(count<size){  \*(p+count) = n;  count+=1;  print(p, count);  account-=1;  }else{  cout<<"Double"<<endl;  int \*new\_arr = new int[size\*2];  for(int i =0; i<count; i++){  new\_arr[i] = \*(p+i);  }  account-=count;  p = new\_arr;  \*(p+count) = n;  size\*=2;  count+=1;  account-=1;  print(p, count);  }  cout<<"Account balance: "<<account<<endl;  cout<<endl;  }   return 0; } |
| --- |

Output:

| Initial account balance = 0  Enter the number you wish to insert in the dynamic table: 5 5  Account balance: 2  Enter the number you wish to insert in the dynamic table: 10 Double 5 10  Account balance: 3  Enter the number you wish to insert in the dynamic table: 2 Double 5 10 2  Account balance: 3  Enter the number you wish to insert in the dynamic table: 19 5 10 2 19  Account balance: 5  Enter the number you wish to insert in the dynamic table: 5 Double 5 10 2 19 5  Account balance: 3  Enter the number you wish to insert in the dynamic table: 23 5 10 2 19 5 23  Account balance: 5 |
| --- |

**Observations**:

The account balance did not fall below zero even once while we entered values in the dynamic table.

**Conclusion**:

Intuitively, we can consider the accounting method as “saving for a rainy day.” The idea is to allocate a fixed cost d for each step of the algorithm. Low-cost calls will accrue “money” to be able to pay for more expensive calls. the amount in “the bank” must never drop below 0, during any step of the algorithm. As long as this holds, we know that our amortized cost of ˆci per operation is a valid amortized cost of the operationand the sequence of n insert operations has overall time O(n).

**References**:

<https://www.cs.cornell.edu/courses/cs3110/2012sp/lectures/lec21-amortized/lec21.html>

<http://www2.hawaii.edu/~nodari/teaching/s16/notes/notes01.pdf>

**Implementation of Amortized Analysis (Potential Method)**

**Theory**:

Amortized analysis is a method of analyzing algorithms that can help us determine an upper bound on the complexity of an algorithm. This is particularly useful when analyzing operations on data structures, when they involve slow, rarely occurring operations and fast, more common operations. With this disparity between each operations’ complexity, it is difficult to get a tight bound on the overall complexity of a sequence of operations using worst-case analysis. Amortized analysis provides us with a way of averaging the slow and fast operations together to obtain a tight upper bound on the overall algorithm runtime. Here we will consider a simplified version of the hash table problem, and show that a sequence of n insert operations has overall time O(n).

### Potential (Physicist's) Method

Suppose we can define a ***potential function*** Φ (read "Phi") on states of a data structure with the following properties:

* Φ(*h*0) = 0, where *h*0 is the initial state of the data structure.
* Φ(*ht*) ≥ 0 for all states *ht* of the data structure occurring during the course of the computation.

Intuitively, the potential function will keep track of the precharged time at any point in the computation. It measures how much saved-up time is available to pay for expensive operations. It is analogous to the bank balance in the banker's method. But interestingly, it depends only on the current state of the data structure, irrespective of the history of the computation that got it into that state.

We then define the ***amortized time*** of an operation as

| c + Φ(h') − Φ(h), |
| --- |

where *c* is the actual cost of the operation and *h* and *h*' are the states of the data structure before and after the operation, respectively. Thus the amortized time is the actual time plus the change in potential. Ideally, Φ should be defined so that the amortized time of each operation is small. Thus the change in potential should be positive for low-cost operations and negative for high-cost operations.

Now consider a sequence of *n* operations taking actual times *c*0, *c*1, *c*2, ..., *cn*−1 and producing data structures *h*1, *h*2, ..., *hn* starting from *h*0. The total amortized time is the sum of the individual amortized times:

(*c*0 + Φ(*h*1) − Φ(*h*0)) + (*c*1 + Φ(*h*2) − Φ(*h*1)) + ... + (*cn*−1 + Φ(*hn*) − Φ(*hn*−1))

= *c*0 + *c*1 + ... + *cn*−1 + Φ(*hn*) − Φ(*h*0)

= *c*0 + *c*1 + ... + *cn*−1 + Φ(*hn*).

Therefore the amortized time for a sequence of operations overestimates of the actual time by Φ(*hn*), which by assumption is always positive. Thus the total amortized time is always an upper bound on the actual time.

Code:

| # include <iostream> # include <bits/stdc++.h> using namespace std;  void print(int \*arr, int n){  for(int i=0; i<n; i++){  cout<<arr[i]<<" ";  }  cout<<endl; }  int potential(int count, int size){  return (2\*count - size); }  int main(){  int size = 1;  int count = 0;   int arr[size];  int \*p = arr;   while(1){  int n;  cout<<"Enter the number you want to add in the dynamic array: ";  cin>>n;  if(count<size){  \*(p+count) = n;  count+=1;  cout<<"Potential: "<<potential(count, size)<<endl;  print(p,count);  }else{  //double  cout<<"DOUBLE"<<endl;  int \*new\_arr = new int[size\*2];  for(int i=0; i<count; i++) new\_arr[i] = \*(p+i);  size\*=2;  p = new\_arr;  \*(p+count) = n;  cout<<"Potential: "<<potential(count, size)<<endl;  count+=1;  print(p,count);  }  cout<<endl;  }   return 0; } |
| --- |

Output:

| Enter the number you want to add in the dynamic array: 5 Potential: 1 5   Enter the number you want to add in the dynamic array: 10 DOUBLE Potential: 0 5 10   Enter the number you want to add in the dynamic array: 2 DOUBLE Potential: 0 5 10 2   Enter the number you want to add in the dynamic array: 19 Potential: 4 5 10 2 19   Enter the number you want to add in the dynamic array: 5 DOUBLE Potential: 0 5 10 2 19 5   Enter the number you want to add in the dynamic array: 23 Potential: 4 5 10 2 19 5 23 |
| --- |

**Observations**:

The amortized time for a sequence of operations overestimates of the actual time by Φ(hn), which by assumption is always positive. Thus the total amortized time is always an upper bound on the actual time.

**Conclusion**:

The key to amortized analysis with the physicist's method is to define the right potential function. The potential function needs to save up enough time to be used later when it is needed. But it cannot save so much time that it causes the amortized time of the current operation to be too high and the sequence of n insert operations has overall time O(n).

**References**:

<https://www.cs.cornell.edu/courses/cs3110/2012sp/lectures/lec21-amortized/lec21.html>

**Implementation of Hiring Problem using multiple position scenario**

Theory:

The hiring problem is a simple model of decision-making under uncertainty

It is closely related to the well-known Secretary Problem

A sequence of n candidates is to be interviewed to fill a post. For each interviewed candidate we only learn about his/her relative rank among the candidates we've seen so far. After each interview, hire and finish, or discard and interview a new candidate. The nth candidate must be hired if we have reached that far. The goal: devise an strategy that maximizes the probability of hiring the best of the n candidates

Hire-Assistant(n)  
  1  best = 0                // fictional least qualified candidate  
  2  for i = 1 to n  
  3    interview candidate i // paying cost  ci  
  4    if candidate i is better than candidate best  
  5      best = i  
  6      hire candidate i    // paying cost ch

Code:

#include <iostream>  
#include <bits/stdc++.h>  
using namespace std;  
  
class Interviewee{  
    public:      
        string name;  
        int talent;  
};  
  
int compute\_cost(Interviewee I[], int n, int ci, int ch){  
    int best = -1;  
    int cost = 0;  
    for(int i=0; i<n; i++){  
        cost+=ci;  
        if(I[i].talent > best){  
            best = I[i].talent;  
            cost+=ch;  
        }  
    }  
    return cost;  
}  
  
int compute\_randomized\_cost(Interviewee I[], int n, int ci, int ch){  
    random\_shuffle(I, I + n);  
    int best = -1;  
    int cost = 0;  
    for(int i=0; i<n; i++){  
        cost+=ci;  
        if(I[i].talent > best){  
            best = I[i].talent;  
            cost+=ch;  
        }  
    }  
    return cost;  
}  
  
  
int main(){  
    cout<<"HIRING PROBLEM"<<endl;  
    cout<<endl;  
  
    int interview\_cost, hiring\_cost, n;  
  
    cout<<"enter interview cost: ";  
    cin>>interview\_cost;  
  
    cout<<"enter hiring cost: ";  
    cin>>hiring\_cost;  
  
    cout<<"enter number of applicants: ";  
    cin>>n;  
  
    cout<<endl;  
    Interviewee I[n];  
    for(int i=0; i<n; i++){  
        Interviewee j;  
        cout<<"Enter name of applicant "<<i<<": ";  
        cin>>j.name;  
        cout<<"Enter talent of applicant "<<i<<": ";  
        cin>>j.talent;  
        I[i] = j;  
        cout<<endl;  
    }  
  
    int cost = compute\_cost(I, n, interview\_cost, hiring\_cost);  
    int randcost = compute\_randomized\_cost(I, n, interview\_cost, hiring\_cost);  
  
    cout<<"Cost: "<<cost<<endl;  
    cout<<"Randomized Cost: "<<randcost<<endl;  
  
    return 0;  
}

Output:

HIRING PROBLEM         
   
enter interview cost: 2  
enter hiring cost: 5  
enter number of applicants: 5  
   
Enter name of applicant 0: Ram  
Enter talent of applicant 0: 10  
   
Enter name of applicant 1: Rahim  
Enter talent of applicant 1: 15  
   
Enter name of applicant 2: Sai  
Enter talent of applicant 2: 8  
   
Enter name of applicant 3: Arjun  
Enter talent of applicant 3: 5  
   
Enter name of applicant 4: Gopal  
Enter talent of applicant 4: 18  
   
Cost: 25  
Randomized Cost: 15

**Observations**:

If the list of people applying for interviews are randomly placed based on their talent value then shuffling their position and then taking the interview will help in minimizing the cost of hiring

**Conclusion**:

Applying randomization over the pool of candidates will help to minimize the cost

before hiring cost is O(m\*ch) where ch=cost of hiring

After randomization it is O(ln n).

**References**:

<https://www.cs.upc.edu/~conrado/research/talks/sem-UCT-hiring.pdf>

<http://www2.hawaii.edu/~suthers/courses/ics311f20/Notes/Topic-05.html>

**Implementation of Birthday Paradox**

Theory:

The birthday problem (also called the birthday paradox) deals with the [probability](https://brilliant.org/wiki/uniform-probability/#probability-by-outcomes) that in a set of n randomly selected people, at least two people share the same birthday.

Though it is not technically a [paradox](https://brilliant.org/wiki/paradox/), it is often referred to as such because the probability is counter-intuitively high.

The birthday problem is an answer to the following question: In a set of n randomly selected people, what is the probability that at least two people share the same birthday?   
What is the smallest value of n where the probability is at least 50% or 99%?

Let p(n) be the probability that at least two of a group of nn randomly selected people share the same birthday. By the [pigeonhole principle](https://brilliant.org/wiki/pigeonhole-principle-definition/), since there are 366 possibilities for birthdays (including February 29), it follows that when n ≥ 367, p(n)=100%. The counterintuitive part of the answer is that for smaller n, the relationship between n and p(n) is (very) non-linear.

In fact, the thresholds to surpass 50% and 99% are quite small: 50% probability is reached with just 23 people and 99% with just 70 people.

Code:

| #include <iostream> using namespace std;     int main(){     // Assuming non-leap year     float num = 365;     float denom = 365;     float pr;     int n = 0;     cout << "Probability to find : ";     cin >> pr;     float p = 1;     while (p > pr){         p \*= (num/denom);         num--;         n++;     }     cout << "Total no. of people out of which there is: " << n << endl;     cout << "The probability that two of them have same birthdays is: "  << p << endl;     return 0; } |
| --- |

Output:

| D:\Files\Engineering\Semester - 6\AA\Experiments\Experiment 6>BirthdayParadox Probability to find : 0.5 Total no. of people out of which there is: 23 The probability that two of them have same birthdays is: 0.492703 |
| --- |

Observations:

We observe that the probability of 50% for two people having the same birthday requires just 23 people and the probability of 99% for two people having the same birthday requires just 99 people.

Conclusion:

The more the number of people, the more the number of people, and thus, more the probability of two people having their birthday on the same day.

References:

<https://betterexplained.com/articles/understanding-the-birthday-paradox/>

<https://www.geeksforgeeks.org/birthday-paradox/>

**Implementation of QuickSort and Randomized QuickSort with comparative analysis.**

Theory:

QuickSort is a[Divide and Conquer algorithm](https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/). It picks an element as pivot and partitions the given array around the picked pivot. There are two partition schemes in quick sort:

* Lomuto’s Partition Scheme: This algorithm works by assuming the pivot element as the last element. If any other element is given as a pivot element then swap it first with the last element. Now initialize two variables i as low and j also low,  iterate over the array and increment i when arr[j] <= pivot and swap arr[i] with arr[j] otherwise increment only i. After coming out from the loop swap arr[i] with arr[hi]. This i stores the pivot element.
* Hoare’s Partition Scheme: [Hoare’s Partition Scheme](https://en.wikipedia.org/wiki/Quicksort#Hoare_partition_scheme) works by initializing two indexes that start at two ends, the two indexes move toward each other until an inversion is (A smaller value on the left side and greater value on the right side) found. When an inversion is found, two values are swapped and the process is repeated.

Code:

Normal Hoare Partition

| #include <stdio.h> #include <iostream> #include <time.h> #include <chrono>  using namespace std; using namespace std::chrono;  int partition(int arr[], int low, int high) {     int pivot = arr[low];     int i = low - 1, j = high + 1;      while (true)     {         do         {             i++;         } while (arr[i] < pivot);          do         {             j--;         } while (arr[j] > pivot);          if (i >= j)             return j;          swap(arr[i], arr[j]);     } }  void quickSort(int arr[], int low, int high) {     if (low < high)     {         int pi = partition(arr, low, high);         quickSort(arr, low, pi);         quickSort(arr, pi + 1, high);     } }  void printArray(int arr[], int size) {     int i;     for (i = 0; i < size; i++)         cout << arr[i] << ' ';     cout << endl; }  int main() {     int arr[] = {99, 81, 56, 63, 17, 6, 84, 87, 62, 5, 7, 38, 67, 22, 10, 37, 90, 85, 25, 1, 88, 42, 16, 2, 91, 95, 78, 50, 97, 82, 52, 69, 32, 43, 40, 53, 73, 61, 30, 26, 33, 70, 74, 35, 31, 65, 11, 24, 27, 54, 57, 15, 66, 98, 44, 19, 49, 13, 60, 39, 8, 80, 3, 86, 18, 34, 59, 72, 93, 71, 21, 4, 23, 64, 9, 29, 47, 55, 46, 89, 79, 51, 100, 36, 75, 58, 68, 41, 94, 28, 83, 20, 48, 45, 77, 14, 12, 96, 76, 92};     int n = sizeof(arr) / sizeof(arr[0]);     cout << "Unsorted Array:" << endl;     printArray(arr, n);     auto start = high\_resolution\_clock::now();     quickSort(arr, 0, n - 1);     auto stop = high\_resolution\_clock::now();     cout << "Sorted Array:" << endl;     printArray(arr, n);     auto duration = duration\_cast<microseconds>(stop - start);     cout << "Time taken by function: " << duration.count() << " microseconds" << endl;     return 0; } |
| --- |

Normal Lomuto Partition

| #include <stdio.h> #include <iostream> #include <time.h> #include <chrono>  using namespace std; using namespace std::chrono;  int partition(int arr[], int low, int high) {     int pivot = arr[high];     int i = (low - 1);      for (int j = low; j <= high - 1; j++)     {         if (arr[j] <= pivot)         {             i++;             swap(arr[i], arr[j]);         }     }     swap(arr[i + 1], arr[high]);     return (i + 1); }  void quickSort(int arr[], int low, int high) {     if (low < high)     {         int pi = partition(arr, low, high);         quickSort(arr, low, pi - 1);         quickSort(arr, pi + 1, high);     } } void printArray(int arr[], int size) {     int i;     for (i = 0; i < size; i++)         cout << arr[i] << " ";     cout << endl; }  int main() {     int arr[] = {99, 81, 56, 63, 17, 6, 84, 87, 62, 5, 7, 38, 67, 22, 10, 37, 90, 85, 25, 1, 88, 42, 16, 2, 91, 95, 78, 50, 97, 82, 52, 69, 32, 43, 40, 53, 73, 61, 30, 26, 33, 70, 74, 35, 31, 65, 11, 24, 27, 54, 57, 15, 66, 98, 44, 19, 49, 13, 60, 39, 8, 80, 3, 86, 18, 34, 59, 72, 93, 71, 21, 4, 23, 64, 9, 29, 47, 55, 46, 89, 79, 51, 100, 36, 75, 58, 68, 41, 94, 28, 83, 20, 48, 45, 77, 14, 12, 96, 76, 92};     int n = sizeof(arr) / sizeof(arr[0]);     cout << "Unsorted Array:" << endl;     printArray(arr, n);     auto start = high\_resolution\_clock::now();     quickSort(arr, 0, n - 1);     auto stop = high\_resolution\_clock::now();     cout << "Sorted Array" << endl;     printArray(arr, n);     auto duration = duration\_cast<microseconds>(stop - start);     cout << "Time taken by function: " << duration.count() << " microseconds" << endl;     return 0; } |
| --- |

Randomized Hoare Partition

| #include <stdio.h> #include <iostream> #include <time.h> #include <chrono> using namespace std; using namespace std::chrono;  int partition(int arr[], int low, int high) {     int pivot = arr[low];     int i = low - 1, j = high + 1;      while (true)     {         do         {             i++;         } while (arr[i] < pivot);          do         {             j--;         } while (arr[j] > pivot);          if (i >= j)             return j;          swap(arr[i], arr[j]);     } }  int random\_partition(int arr[], int low, int high) {     int r = low + rand() % (high - low);     swap(arr[r], arr[low]);     return partition(arr, low, high); }  void quickSort(int arr[], int low, int high) {     if (low < high)     {         int pi = random\_partition(arr, low, high);         quickSort(arr, low, pi);         quickSort(arr, pi + 1, high);     } }  void printArray(int arr[], int size) {     int i;     for (i = 0; i < size; i++)         cout << arr[i] << ' ';     cout << endl; } int main() {     int arr[] = {99, 81, 56, 63, 17, 6, 84, 87, 62, 5, 7, 38, 67, 22, 10, 37, 90, 85, 25, 1, 88, 42, 16, 2, 91, 95, 78, 50, 97, 82, 52, 69, 32, 43, 40, 53, 73, 61, 30, 26, 33, 70, 74, 35, 31, 65, 11, 24, 27, 54, 57, 15, 66, 98, 44, 19, 49, 13, 60, 39, 8, 80, 3, 86, 18, 34, 59, 72, 93, 71, 21, 4, 23, 64, 9, 29, 47, 55, 46, 89, 79, 51, 100, 36, 75, 58, 68, 41, 94, 28, 83, 20, 48, 45, 77, 14, 12, 96, 76, 92};     int n = sizeof(arr) / sizeof(arr[0]);     auto start = high\_resolution\_clock::now();     quickSort(arr, 0, n - 1);     auto stop = high\_resolution\_clock::now();     cout << "Sorted Array" << endl;     printArray(arr, n);     auto duration = duration\_cast<microseconds>(stop - start);     cout << "Time taken by function: " << duration.count() << " microseconds" << endl;     return 0; } |
| --- |

Randomized Lomuto Partition

| #include <stdio.h> #include <iostream> #include <time.h> #include <chrono> using namespace std; using namespace std::chrono; // Lomuto Partitioning int partition(int arr[], int low, int high) {     int pivot = arr[high];     int i = (low - 1);      for (int j = low; j <= high - 1; j++)     {         if (arr[j] <= pivot)         {             i++;             swap(arr[i], arr[j]);         }     }     swap(arr[i + 1], arr[high]);     return (i + 1); } int random\_partition(int arr[], int low, int high) {     int r = low + rand() % (high - low);     swap(arr[r], arr[high]);     return partition(arr, low, high); }  void quickSort(int arr[], int low, int high) {     if (low < high)     {         int pi = random\_partition(arr, low, high);         quickSort(arr, low, pi - 1);         quickSort(arr, pi + 1, high);     } } void printArray(int arr[], int size) {     int i;     for (i = 0; i < size; i++)         cout << arr[i] << ' ';     cout << endl; }  int main() {     int arr[] = {99, 81, 56, 63, 17, 6, 84, 87, 62, 5, 7, 38, 67, 22, 10, 37, 90, 85, 25, 1, 88, 42, 16, 2, 91, 95, 78, 50, 97, 82, 52, 69, 32, 43, 40, 53, 73, 61, 30, 26, 33, 70, 74, 35, 31, 65, 11, 24, 27, 54, 57, 15, 66, 98, 44, 19, 49, 13, 60, 39, 8, 80, 3, 86, 18, 34, 59, 72, 93, 71, 21, 4, 23, 64, 9, 29, 47, 55, 46, 89, 79, 51, 100, 36, 75, 58, 68, 41, 94, 28, 83, 20, 48, 45, 77, 14, 12, 96, 76, 92};     int n = sizeof(arr) / sizeof(arr[0]);     cout << "Unsorted Array" << endl;     printArray(arr, n);     auto start = high\_resolution\_clock::now();     quickSort(arr, 0, n - 1);     auto stop = high\_resolution\_clock::now();     cout << "Sorted Array" << endl;     printArray(arr, n);     auto duration = duration\_cast<microseconds>(stop - start);     cout << "Time taken by function: " << duration.count() << " microseconds" << endl;     return 0; } |
| --- |

Output:

Normal Hoare Partition

| D:\Files\Engineering\Semester - 6\AA\Experiments\Experiment 7>g++ -Wall -std=c++14 normal\_hoare.cpp -o normal\_hoare  D:\Files\Engineering\Semester - 6\AA\Experiments\Experiment 7>normal\_hoare Unsorted Array: 99 81 56 63 17 6 84 87 62 5 7 38 67 22 10 37 90 85 25 1 88 42 16 2 91 95 78 50 97 82 52 69 32 43 40 53 73 61 30 26 33 70 74 35 31 65 11 24 27 54 57 15 66 98 44 19 49 13  60 39 8 80 3 86 18 34 59 72 93 71 21 4 23 64 9 29 47 55 46 89 79 51 100 36 75 58 68 41 94 28 83 20 48 45 77 14 12 96 76 92 Sorted Array: 1 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 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 Time taken by function: 13 microseconds |
| --- |

Normal Lomuto Partition

| D:\Files\Engineering\Semester - 6\AA\Experiments\Experiment 7>g++ -Wall -std=c++14 normal\_lomuto.cpp -o normal\_lomuto  D:\Files\Engineering\Semester - 6\AA\Experiments\Experiment 7>normal\_lomuto Unsorted Array: 99 81 56 63 17 6 84 87 62 5 7 38 67 22 10 37 90 85 25 1 88 42 16 2 91 95 78 50 97 82 52 69 32 43 40 53 73 61 30 26 33 70 74 35 31 65 11 24 27 54 57 15 66 98 44 19 49 13  60 39 8 80 3 86 18 34 59 72 93 71 21 4 23 64 9 29 47 55 46 89 79 51 100 36 75 58 68 41 94 28 83 20 48 45 77 14 12 96 76 92 Sorted Array 1 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 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 Time taken by function: 14 microseconds |
| --- |

Random Hoare Partition

| D:\Files\Engineering\Semester - 6\AA\Experiments\Experiment 7>g++ -Wall -std=c++14 random\_hoare.cpp -o random\_hoare  D:\Files\Engineering\Semester - 6\AA\Experiments\Experiment 7>random\_hoare Unsorted Array 99 81 56 63 17 6 84 87 62 5 7 38 67 22 10 37 90 85 25 1 88 42 16 2 91 95 78 50 97 82 52 69 32 43 40 53 73 61 30 26 33 70 74 35 31 65 11 24 27 54 57 15 66 98 44 19 49 13  60 39 8 80 3 86 18 34 59 72 93 71 21 4 23 64 9 29 47 55 46 89 79 51 100 36 75 58 68 41 94 28 83 20 48 45 77 14 12 96 76 92 Sorted Array 1 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 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 Time taken by function: 12 microseconds |
| --- |

Random Lomuto Partition’

| D:\Files\Engineering\Semester - 6\AA\Experiments\Experiment 7>g++ -Wall -std=c++14 random\_lomuto.cpp -o random\_lomuto  D:\Files\Engineering\Semester - 6\AA\Experiments\Experiment 7>random\_lomuto Unsorted Array 99 81 56 63 17 6 84 87 62 5 7 38 67 22 10 37 90 85 25 1 88 42 16 2 91 95 78 50 97 82 52 69 32 43 40 53 73 61 30 26 33 70 74 35 31 65 11 24 27 54 57 15 66 98 44 19 49 13  60 39 8 80 3 86 18 34 59 72 93 71 21 4 23 64 9 29 47 55 46 89 79 51 100 36 75 58 68 41 94 28 83 20 48 45 77 14 12 96 76 92 Sorted Array 1 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 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 Time taken by function: 15 microseconds |
| --- |

Observations:

Comparing Hoare partitioning with Lomuto partitioning, Hoare partitioning is more efficient as it takes 13 microseconds while Lomuto partitioning takes 14 microseconds. Randomized Hoare partitioning is even more efficient as it takes 12 microseconds while randomized Lomuto partitioning is less efficient with a runtime of 15 microseconds.

Conclusion:

Thus randomized Hoare partitioning is the most efficient partitioning method amongst all the methods that we implemented. Hoare partitioning is more efficient than Lomuto partitioning because it does three times fewer swaps on an average. It also creates efficient partitions even when all values are equal.

References:

<https://www.geeksforgeeks.org/hoares-vs-lomuto-partition-scheme-quicksort/>

**Jayesh Kavedia**

**60004190053**

**TE COMPS A**

**Implementation of Advanced Data Structure for Red-Black Tree Insertion Rules**

**Theory**

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions.

Algorithm for Insertion of a Node

Let x be the newly inserted node.

* Perform standard BST insertion and make the colour of newly inserted node as RED.
* If x is the root, change the colour of x as BLACK.
* Do the following if the color of x’s parent is not BLACK and x is not the root.
  + If x’s uncle is RED
    - Change the colour of parent and uncle as BLACK.
    - Colour of a grandparent as RED.
    - Change x = x’s grandparent, repeat steps 2 and 3 for new x.
  + If x’s uncle is BLACK, then there can be four configurations for x, x’s parent (p) and x’s grandparent (g)
    - Left Left Case (p is left child of g and x is left child of p) : Perform Right Rotation and Swap Color
    - Left Right Case (p is left child of g and x is the right child of p) : Perform RL Rotation and Swap Color
    - Right Right Case (Mirror of case i) : Perform Left Rotation and Swap Color
    - Right Left Case (Mirror of case ii) : Perform LR Rotation and Swap Color

**Code**

#include <bits/stdc++.h>  
using namespace std;  
   
enum Color {RED, BLACK};  
   
struct Node  
{  
    int data;  
    bool color;  
    Node \*left, \*right, \*parent;  
      
    Node(int data)  
    {  
       this->data = data;  
       left = right = parent = NULL;  
       this->color = RED;  
    }  
};  
   
class RBTree  
{  
private:  
    Node \*root;  
protected:  
    void rotateLeft(Node \*&, Node \*&);  
    void rotateRight(Node \*&, Node \*&);  
    void fixViolation(Node \*&, Node \*&);  
public:  
    RBTree() { root = NULL; }  
    void insert(const int &n);  
    void inorder();  
    void levelOrder();  
    int blackdepth();  
};  
  
int blackdepthHelper(Node \*root){  
    Node\* ptr = root;  
    int count = 0;  
    while(ptr!=NULL){  
        if (ptr->color ==1){  
            count++;  
        }  
        ptr= ptr->left;  
    }  
    return count;  
}  
   
void inorderHelper(Node \*root)  
{  
    if (root == NULL)  
        return;  
   
    inorderHelper(root->left);  
    cout << root->data << "  "<< root->color << endl;  
    inorderHelper(root->right);  
}  
   
Node\* BSTInsert(Node\* root, Node \*pt)  
{  
    if (root == NULL)  
       return pt;  
   
    if (pt->data < root->data)  
    {  
        root->left  = BSTInsert(root->left, pt);  
        root->left->parent = root;  
    }  
    else if (pt->data > root->data)  
    {  
        root->right = BSTInsert(root->right, pt);  
        root->right->parent = root;  
    }  
    return root;  
}  
   
void levelOrderHelper(Node \*root)  
{  
    if (root == NULL)  
        return;  
   
    std::queue<Node \*> q;  
    q.push(root);  
   
    while (!q.empty())  
    {  
        Node \*temp = q.front();  
        cout << temp->data << "  "<< temp->color << endl;  
        q.pop();  
   
        if (temp->left != NULL)  
            q.push(temp->left);  
   
        if (temp->right != NULL)  
            q.push(temp->right);  
    }  
}  
   
void RBTree::rotateLeft(Node \*&root, Node \*&pt)  
{  
    Node \*pt\_right = pt->right;  
   
    pt->right = pt\_right->left;  
   
    if (pt->right != NULL)  
        pt->right->parent = pt;  
   
    pt\_right->parent = pt->parent;  
   
    if (pt->parent == NULL)  
        root = pt\_right;  
   
    else if (pt == pt->parent->left)  
        pt->parent->left = pt\_right;  
   
    else  
        pt->parent->right = pt\_right;  
   
    pt\_right->left = pt;  
    pt->parent = pt\_right;  
}  
   
void RBTree::rotateRight(Node \*&root, Node \*&pt)  
{  
    Node \*pt\_left = pt->left;  
   
    pt->left = pt\_left->right;  
   
    if (pt->left != NULL)  
        pt->left->parent = pt;  
   
    pt\_left->parent = pt->parent;  
   
    if (pt->parent == NULL)  
        root = pt\_left;  
   
    else if (pt == pt->parent->left)  
        pt->parent->left = pt\_left;  
   
    else  
        pt->parent->right = pt\_left;  
   
    pt\_left->right = pt;  
    pt->parent = pt\_left;  
}  
   
void RBTree::fixViolation(Node \*&root, Node \*&pt)  
{  
    Node \*parent\_pt = NULL;  
    Node \*grand\_parent\_pt = NULL;  
   
    while ((pt != root) && (pt->color != BLACK) &&  
           (pt->parent->color == RED))  
    {  
   
        parent\_pt = pt->parent;  
        grand\_parent\_pt = pt->parent->parent;  
   
        if (parent\_pt == grand\_parent\_pt->left)  
        {  
   
            Node \*uncle\_pt = grand\_parent\_pt->right;  
   
            if (uncle\_pt != NULL && uncle\_pt->color == RED)  
            {  
                grand\_parent\_pt->color = RED;  
                parent\_pt->color = BLACK;  
                uncle\_pt->color = BLACK;  
                pt = grand\_parent\_pt;  
            }  
   
            else  
            {  
                if (pt == parent\_pt->right)  
                {  
                    rotateLeft(root, parent\_pt);  
                    pt = parent\_pt;  
                    parent\_pt = pt->parent;  
                }  
                rotateRight(root, grand\_parent\_pt);  
                swap(parent\_pt->color,  
                           grand\_parent\_pt->color);  
                pt = parent\_pt;  
            }  
        }  
        else  
        {  
            Node \*uncle\_pt = grand\_parent\_pt->left;  
            if ((uncle\_pt != NULL) && (uncle\_pt->color == RED))  
            {  
                grand\_parent\_pt->color = RED;  
                parent\_pt->color = BLACK;  
                uncle\_pt->color = BLACK;  
                pt = grand\_parent\_pt;  
            }  
            else  
            {  
                if (pt == parent\_pt->left)  
                {  
                    rotateRight(root, parent\_pt);  
                    pt = parent\_pt;  
                    parent\_pt = pt->parent;  
                }  
                rotateLeft(root, grand\_parent\_pt);  
                swap(parent\_pt->color,  
                         grand\_parent\_pt->color);  
                pt = parent\_pt;  
            }  
        }  
    }  
    root->color = BLACK;  
}  
   
void RBTree::insert(const int &data)  
{  
    Node \*pt = new Node(data);  
   
    root = BSTInsert(root, pt);  
   
    fixViolation(root, pt);  
}  
   
void RBTree::inorder()     {  inorderHelper(root);}  
void RBTree::levelOrder()  {  levelOrderHelper(root); }  
int RBTree::blackdepth()  {  blackdepthHelper(root);  }  
   
int main()  
{  
    RBTree tree;  
   
    tree.insert(8);  
    tree.insert(18);  
    tree.insert(5);  
    tree.insert(15);  
    tree.insert(17);  
    tree.insert(25);  
    tree.insert(40);  
    tree.insert(80);  
      
    cout << "1 -> Black Node 0 -> Red Node\n\n";  
   
    cout << "Inorder Traversal of Created Tree\n";  
    tree.inorder();  
   
    cout << "\n\nLevel Order Traversal of Created Tree\n";  
    tree.levelOrder();  
      
    int bd = tree.blackdepth();  
    cout<< "\n\nBlack depth of RB Tree excluding NULL node " << bd;  
    cout<< "\n\nBlack depth of RB Tree including NULL node " << bd+1;  
   
    return 0;  
}

**Output**

1 -> Black Node 0 -> Red Node  
  
Inorder Traversal of Created Tree  
5  1  
8  0  
15  1  
17  1  
18  1  
25  0  
40  1  
80  0  
  
  
Level Order Traversal of Created Tree  
17  1  
8  0  
25  0  
5  1  
15  1  
18  1  
40  1  
80  0  
  
  
Black depth of RB Tree excluding NULL node 2  
  
Black depth of RB Tree including NULL node 3

**Observations**

* Insertion follows BST insertion property along with a fixup method to balance the tree in case of any property violation.
* In case of Red Node followed by Red Node there are three possible cases to be handled.
* Red-black trees offer logarithmic average and worst-case time complexity for insertion.
* Rebalancing has an average time complexity of O(1) and worst-case complexity of O(log n).

**Conclusion**

Red Black Tree is a self-balanced tree similar to BST with one extra bit of storage for the color value. While Inserting a node in RB Tree the node must be inserted such that no violation of RB Tree Property takes place. The Average Case Time Complexity for RB Tree Insertion is O (log n).

**References**

* [https://www.baeldung.com/cs/red-black-trees#:~:text=5.-,Complexity,to%20bulk%20and%20parallel%20operations.](https://www.baeldung.com/cs/red-black-trees%23:~:text=5.-,Complexity,to%20bulk%20and%20parallel%20operations.)
* <https://www.ijsrd.com/articles/IJSRDV6I90249.pdf>
* <https://www.cs.umanitoba.ca/~hacamero/Research/RBTreesKim.pdf>

**Jayesh Kavedia**

**60004190053**

**TE COMPS A**

**Implementation of Advanced Data Structure for Red-Black Tree Deletion Rules**

**Theory**

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions.

RB Tree Deletion Algorithm

1) Perform standard BST delete. When we perform standard delete operation in BST, we always end up deleting a node which is an either leaf or has only one child. So we only need to handle cases where a node is leaf or has one child. Let v be the node to be deleted and u be the child that replaces v.

2) Simple Case: If either u or v is red, we mark the replaced child as black.

3) If Both u and v are Black.

3.1) Color u as double black. Now our task reduces to convert this double black to single black.

3.2) Do following while the current node u is double black, and it is not the root. Let sibling of node be s.

(a): If sibling s is black and at least one of sibling’s children is red, perform rotation(s). Let the red child of s be r. This case can be divided in four subcases depending upon positions of s and r.

(i) Left Left Case (s is left child of its parent and r is left child of s or both children of s are red).

(ii) Left Right Case (s is left child of its parent and r is right child).

(iii) Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)

(iv) Right Left Case (s is right child of its parent and r is left child of s)

(b): If sibling is black and its both children are black, perform recoloring, and recur for the parent if parent is black. In this case, if parent was red, then we didn’t need to recur for parent, we can simply make it black.

(c): If sibling is red, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black.This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two subcases.

(i) Left Case (s is left child of its parent). We right rotate the parent p.

(ii) Right Case (s is right child of its parent). We left rotate the parent p.

3.3) If u is root, make it single black and return.

**Code**

#include <iostream>  
#include <queue>  
using namespace std;  
   
enum COLOR { RED, BLACK };  
   
class Node {  
public:  
  int val;  
  COLOR color;  
  Node \*left, \*right, \*parent;  
   
  Node(int val) : val(val) {  
    parent = left = right = NULL;  
    color = RED;  
  }  
   
  Node \*uncle() {  
    if (parent == NULL or parent->parent == NULL)  
      return NULL;  
   
    if (parent->isOnLeft())  
      return parent->parent->right;  
    else  
      return parent->parent->left;  
  }  
   
  bool isOnLeft() { return this == parent->left; }  
   
  Node \*sibling() {  
    if (parent == NULL)  
      return NULL;  
   
    if (isOnLeft())  
      return parent->right;  
   
    return parent->left;  
  }  
   
  void moveDown(Node \*nParent) {  
    if (parent != NULL) {  
      if (isOnLeft()) {  
        parent->left = nParent;  
      } else {  
        parent->right = nParent;  
      }  
    }  
    nParent->parent = parent;  
    parent = nParent;  
  }  
   
  bool hasRedChild() {  
    return (left != NULL and left->color == RED) or  
           (right != NULL and right->color == RED);  
  }  
};  
   
class RBTree {  
  Node \*root;  
   
  void leftRotate(Node \*x) {  
    Node \*nParent = x->right;  
    if (x == root)  
      root = nParent;  
   
    x->moveDown(nParent);  
   
    x->right = nParent->left;  
    if (nParent->left != NULL)  
      nParent->left->parent = x;  
    nParent->left = x;  
  }  
   
  void rightRotate(Node \*x) {  
    Node \*nParent = x->left;  
   
    if (x == root)  
      root = nParent;  
   
    x->moveDown(nParent);  
   
    x->left = nParent->right;  
    if (nParent->right != NULL)  
      nParent->right->parent = x;  
   
    nParent->right = x;  
  }  
   
  void swapColors(Node \*x1, Node \*x2) {  
    COLOR temp;  
    temp = x1->color;  
    x1->color = x2->color;  
    x2->color = temp;  
  }  
   
  void swapValues(Node \*u, Node \*v) {  
    int temp;  
    temp = u->val;  
    u->val = v->val;  
    v->val = temp;  
  }  
   
  void fixRedRed(Node \*x) {  
    if (x == root) {  
      x->color = BLACK;  
      return;  
    }  
   
    Node \*parent = x->parent, \*grandparent = parent->parent,  
         \*uncle = x->uncle();  
   
    if (parent->color != BLACK) {  
      if (uncle != NULL && uncle->color == RED) {  
        parent->color = BLACK;  
        uncle->color = BLACK;  
        grandparent->color = RED;  
        fixRedRed(grandparent);  
      } else {  
        if (parent->isOnLeft()) {  
          if (x->isOnLeft()) {  
            swapColors(parent, grandparent);  
          } else {  
            leftRotate(parent);  
            swapColors(x, grandparent);  
          }  
          rightRotate(grandparent);  
        } else {  
          if (x->isOnLeft()) {  
            rightRotate(parent);  
            swapColors(x, grandparent);  
          } else {  
            swapColors(parent, grandparent);  
          }  
          leftRotate(grandparent);  
        }  
      }  
    }  
  }  
   
  Node \*successor(Node \*x) {  
    Node \*temp = x;  
   
    while (temp->left != NULL)  
      temp = temp->left;  
   
    return temp;  
  }  
  Node \*BSTreplace(Node \*x) {  
    if (x->left != NULL and x->right != NULL)  
      return successor(x->right);  
    if (x->left == NULL and x->right == NULL)  
      return NULL;  
    if (x->left != NULL)  
      return x->left;  
    else  
      return x->right;  
  }  
  void deleteNode(Node \*v) {  
    Node \*u = BSTreplace(v);  
    bool uvBlack = ((u == NULL or u->color == BLACK) and (v->color == BLACK));  
    Node \*parent = v->parent;  
   
    if (u == NULL) {  
      if (v == root) {  
        root = NULL;  
      } else {  
        if (uvBlack) {  
          fixDoubleBlack(v);  
        } else {  
          if (v->sibling() != NULL)  
            v->sibling()->color = RED;  
        }  
   
        if (v->isOnLeft()) {  
          parent->left = NULL;  
        } else {  
          parent->right = NULL;  
        }  
      }  
      delete v;  
      return;  
    }  
   
    if (v->left == NULL or v->right == NULL) {  
      if (v == root) {  
        v->val = u->val;  
        v->left = v->right = NULL;  
        delete u;  
      } else {  
        if (v->isOnLeft()) {  
          parent->left = u;  
        } else {  
          parent->right = u;  
        }  
        delete v;  
        u->parent = parent;  
        if (uvBlack) {  
          fixDoubleBlack(u);  
        } else {  
          u->color = BLACK;  
        }  
      }  
      return;  
    }  
   
    swapValues(u, v);  
    deleteNode(u);  
  }  
   
  void fixDoubleBlack(Node \*x) {  
    if (x == root)  
      return;  
   
    Node \*sibling = x->sibling(), \*parent = x->parent;  
    if (sibling == NULL) {  
      fixDoubleBlack(parent);  
    } else {  
      if (sibling->color == RED) {  
        parent->color = RED;  
        sibling->color = BLACK;  
        if (sibling->isOnLeft()) {  
          rightRotate(parent);  
        } else {  
          leftRotate(parent);  
        }  
        fixDoubleBlack(x);  
      } else {  
        if (sibling->hasRedChild()) {  
          if (sibling->left != NULL and sibling->left->color == RED) {  
            if (sibling->isOnLeft()) {  
              sibling->left->color = sibling->color;  
              sibling->color = parent->color;  
              rightRotate(parent);  
            } else {  
              sibling->left->color = parent->color;  
              rightRotate(sibling);  
              leftRotate(parent);  
            }  
          } else {  
            if (sibling->isOnLeft()) {  
              sibling->right->color = parent->color;  
              leftRotate(sibling);  
              rightRotate(parent);  
            } else {  
              sibling->right->color = sibling->color;  
              sibling->color = parent->color;  
              leftRotate(parent);  
            }  
          }  
          parent->color = BLACK;  
        } else {  
          sibling->color = RED;  
          if (parent->color == BLACK)  
            fixDoubleBlack(parent);  
          else  
            parent->color = BLACK;  
        }  
      }  
    }  
  }  
   
  void levelOrder(Node \*x) {  
    if (x == NULL)  
      return;  
   
    queue<Node \*> q;  
    Node \*curr;  
   
    q.push(x);  
   
    while (!q.empty()) {  
      curr = q.front();  
      q.pop();  
   
      cout << curr->val << " ";  
   
      if (curr->left != NULL)  
        q.push(curr->left);  
      if (curr->right != NULL)  
        q.push(curr->right);  
    }  
  }  
   
  void inorder(Node \*x) {  
    if (x == NULL)  
      return;  
    inorder(x->left);  
    cout << x->val << " ";  
    inorder(x->right);  
  }  
   
public:  
  RBTree() { root = NULL; }  
   
  Node \*getRoot() { return root; }  
   
  Node \*search(int n) {  
    Node \*temp = root;  
    while (temp != NULL) {  
      if (n < temp->val) {  
        if (temp->left == NULL)  
          break;  
        else  
          temp = temp->left;  
      } else if (n == temp->val) {  
        break;  
      } else {  
        if (temp->right == NULL)  
          break;  
        else  
          temp = temp->right;  
      }  
    }  
   
    return temp;  
  }  
   
  void insert(int n) {  
    Node \*newNode = new Node(n);  
    if (root == NULL) {  
      newNode->color = BLACK;  
      root = newNode;  
    } else {  
      Node \*temp = search(n);  
   
      if (temp->val == n) {  
        return;  
      }  
      newNode->parent = temp;  
   
      if (n < temp->val)  
        temp->left = newNode;  
      else  
        temp->right = newNode;  
   
      fixRedRed(newNode);  
    }  
  }  
   
  void deleteByVal(int n) {  
    if (root == NULL)  
      return;  
   
    Node \*v = search(n), \*u;  
   
    if (v->val != n) {  
      cout << "No node found to delete with value:" << n << endl;  
      return;  
    }  
   
    deleteNode(v);  
  }  
   
  void printInOrder() {  
    cout << "Inorder: " << endl;  
    if (root == NULL)  
      cout << "Tree is empty" << endl;  
    else  
      inorder(root);  
    cout << endl;  
  }  
   
  void printLevelOrder() {  
    cout << "Level order: " << endl;  
    if (root == NULL)  
      cout << "Tree is empty" << endl;  
    else  
      levelOrder(root);  
    cout << endl;  
  }  
};  
   
int main() {  
  RBTree tree;  
   
    tree.insert(8);  
    tree.insert(18);  
    tree.insert(5);  
    tree.insert(15);  
    tree.insert(17);  
    tree.insert(25);  
    tree.insert(40);  
    tree.insert(80);  
  
   
  tree.printInOrder();  
  tree.printLevelOrder();  
   
  cout<<endl<<"Deleting 8"<<endl;  
  tree.deleteByVal(8);  
  tree.printLevelOrder();  
  cout<<endl<<"Deleting 80"<<endl;  
  tree.deleteByVal(80);  
  tree.printLevelOrder();  
  cout<<endl<<"Deleting 5"<<endl;   
  tree.deleteByVal(5);  
  tree.printLevelOrder();  
  return 0;  
}

**Output**

// Original Tree

Inorder:   
5 8 15 17 18 25 40 80   
Level order:   
17 8 25 5 15 18 40 80   
  
Deleting 8  
Level order:   
17 15 25 5 18 40 80   
  
Deleting 80  
Level order:   
17 15 25 5 18 40   
  
Deleting 5  
Level order:   
17 15 25 18 40

**Observations**

* Deletion follows BST deletion property and finds successor node, along with a fixup method to balance the tree in case of any property violation.
* In case of Double Black Node there are eleven possible cases to be handled.
* Red-black trees offer logarithmic average and worst-case time complexity for deletion.
* Rebalancing has a time complexity of O(1) and worst-case complexity of O(log n).

**Conclusion**

Red Black Tree is a self-balanced tree similar to BST with one extra bit of storage for the color value. While Deleting a node in RB Tree the deleted node’s successor needs to be found and fixed to avoid any property violations. The Average Case Time Complexity for RB Tree Deletion is O (log n).

**References**

* [https://www.baeldung.com/cs/red-black-trees#:~:text=5.-,Complexity,to%20bulk%20and%20parallel%20operations.](https://www.baeldung.com/cs/red-black-trees%23:~:text=5.-,Complexity,to%20bulk%20and%20parallel%20operations.)
* <https://www.cs.umanitoba.ca/~hacamero/Research/RBTreesKim.pdf>
* “Introduction to Algorithms, Second Edition,” by Thomas H. Cormen, Charles E. Leiserson, Ronald L.Rivest and Clifford Stein.

**Implementation of Network Flow Algorithm using Ford-Fulkerson Method**

Theory:

Residual Graph of a flow network is a graph which indicates additional possible flow. If there is a path from source to sink in residual graph, then it is possible to add flow. Every edge of a residual graph has a value called residual capacity which is equal to original capacity of the edge minus current flow. Residual capacity is basically the current capacity of the edge. . Residual capacity is 0 if there is no edge between two vertices of residual graph. We can initialize the residual graph as original graph as there is no initial flow and initially residual capacity is equal to original capacity. To find an augmenting path, we can either do a BFS or DFS of the residual graph. Using BFS, we can find out if there is a path from source to sink. BFS also builds parent[] array. Using the parent[] array, we traverse through the found path and find possible flow through this path by finding minimum residual capacity along the path. We later add the found path flow to overall flow

Code:

#include <iostream>  
#include <bits/stdc++.h>  
using namespace std;  
  
#define V 6  
  
bool bfs(int graph[V][V], int s, int t, int parent[]){  
    bool visited[V];  
    for(int i = 0; i<V; i++) visited[i] = 0;  
  
    queue<int> q;  
    visited[s] = true;  
    q.push(s);  
    parent[s] = -1;  
    // cout<<s<<" ";  
  
    while(!q.empty()){  
        int u = q.front();  
        q.pop();  
  
        for(int v = 0; v<V; v++){  
            if(visited[v]==false && graph[u][v]>0){  
                visited[v] = true;  
                q.push(v);  
                parent[v] = u;  
                // cout<<v<<" ";  
            }  
        }  
    }  
    return (visited[t]==true);  
}  
  
int ford\_fulkerson(int graph[V][V], int s, int t){  
    int rgraph[V][V];  
    for(int u=0; u<V; u++){  
        for(int v=0; v<V; v++){  
            rgraph[u][v] = graph[u][v];  
        }  
    }  
  
    int max\_flow = 0;  
    int parent[V];  
     
    bool bb = bfs(rgraph, s, t, parent);  
  
    while(bfs(rgraph, s, t, parent)){  
        int path\_flow = INT\_MAX;  
        for(int v=t; v!=s; v=parent[v]){  
            int u = parent[v];  
            path\_flow = min(path\_flow, rgraph[u][v]);  
        }  
  
        for(int v=t; v!=s; v=parent[v]){  
            int u = parent[v];  
            rgraph[u][v]-=path\_flow;  
            rgraph[v][u]+=path\_flow;  
        }  
        // cout<<path\_flow<<endl;  
        max\_flow+=path\_flow;  
    }  
  
    return max\_flow;  
}  
  
  
int main(){  
  
    int graph[6][6] = {{0, 8, 0, 0, 3, 0},  
                    {0, 0, 9, 0 ,0 ,0},  
                    {0, 0, 0, 0, 7, 2},  
                    {0, 0, 0, 0, 0, 5},  
                    {0, 0, 7, 4, 0, 0},  
                    {0, 0, 0, 0, 0, 0}};  
     
    int max\_flow = ford\_fulkerson(graph, 0, 5);  
    cout<<"max flow: "<<max\_flow<<endl;  
    return 0;  
}

Output:

max flow: 6

Observations:

The above implementation uses adjacency matrix representation though where BFS takes O(V2) time, the time complexity of the above implementation is O(EV3)

Conclusion:

Flow network is a graph which indicates additional possible flow. If there is a path from source to sink in residual graph

Residual capacity is 0 if there is no edge between two vertices of residual graph

References:

<https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/>

**Department of Computer Engineering**

**Academic Year 2021-2022**

**Group Members**

*Jayesh Kavedia – 60004190053*

*Jigar Shah – 60004190055*

*Junaid Girkar – 60004190057*

**Dictionary of Words using RB TREE**

* **PROBLEM STATEMENT**

Dictionary can store data of any type and can be used later for accessing data in an efficient way. A Dictionary can be made using various possible data structures available like AVL tree, Trie, Red Black Tree, B Tree etc. But since a dictionary can contain huge amount of data it is necessary to consider the time and space taken to store and retrieve data from the dictionary. Red Black Tree have an advantage over Trie and AVL Tree and are very much similar to B Tree. This project aims to build a dictionary of words using Red Black Tree.

* **INTRODUCTION**

**Need**

One can use a dictionary to look up the meaning of any words that one doesn’t understand. A dictionary is one of the most important tools during studying at a university. A good dictionary can help one understand the subject better, improve one’s communication by making sure you are using words correctly. A monolingual dictionary has lots of different information about every word in English. These dictionaries have lots of information about grammar and pronunciation. To make this dictionary available online to everyone for use, we need to store it in some data structure for efficient management of data from the dictionary. As a dictionary can save n number of words it is necessary to maintain the storage aspect as well as retrieval of data. Thus, need of a data structure to store words of a dictionary is essential.

**Working**

The project consists of a pre built English Dictionary words which can be loaded into the Red Black Tree. The program reads the text file line by line, inserts each word in the Red Black Tree using BST Insertion Property and fixes the insertion so that there are no violations of the Red Black Tree properties. Through the program option has been provided to insert a new word in the dictionary. When the user chooses insert a word option he is prompted to insert a word, the program reads the word and inserts that word in the dictionary i.e., in the RB Tree using the BST Insertion and Fixup Code to avoid any violations of properties. The program provides an option to find a particular word in the dictionary. When the user chooses the option to find a word he is prompted to insert the word which is needed to be found. The search operation takes place through the normal BST Search Property. If the word is found the user gets a message that the word has been found. If the word is not found then user gets a message that the word entered is not present in the dictionary. The program also provides an option to see the size of the dictionary. When the user chooses the size option, the program counts all the nodes of the RB Tree and displays the sum of all the nodes, which represents the size of the dictionary. The user can also see the height of the tree formed for the dictionary. When the user chooses this option the program counts the number of nodes present in one side of the tree and returns the count which represents the height of the tree.

**Applications**

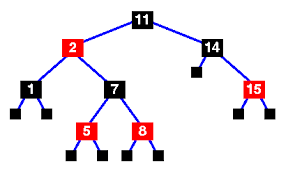
The project can be used as a standard dictionary which contains words and their meanings. It provides fast storage and access of data with the help of Red Black Trees. The project can be used to load a predefined set of words in the dictionary. It can also be used to add new words in the dictionary. It can also be used to find whether a word is present in the dictionary or not. It provides all these operations in minimum time and space possible.

* **ADVANCED DATA STRUCTURE**

**Theory/Working**

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree. Properties of RB Tree are :

* Every node has a color either red or black.
* The root of the tree is always black.
* There are no two adjacent red nodes (A red node cannot have a red parent or red child).
* Every path from a node (including root) to any of its descendants’ NULL nodes has the same number of black nodes.
* All leaf nodes are black nodes



Black height is the number of black nodes on a path from the root to a leaf. Leaf nodes are also counted black nodes.

**Insertion of Node in RB Tree**

Algorithm for Insertion of a Node

Let x be the newly inserted node.

Perform standard BST insertion and make the colour of newly inserted node as RED.

If x is the root, change the colour of x as BLACK.

Do the following if the color of x’s parent is not BLACK and x is not the root.

If x’s uncle is RED

Change the colour of parent and uncle as BLACK.

Colour of a grandparent as RED.

Change x = x’s grandparent, repeat steps 2 and 3 for new x.

If x’s uncle is BLACK, then there can be four configurations for x, x’s parent (p) and x’s grandparent (g)

Left Left Case (p is left child of g and x is left child of p) : Perform Right Rotation and Swap Color

Left Right Case (p is left child of g and x is the right child of p) : Perform RL Rotation and Swap Color

Right Right Case (Mirror of case i) : Perform Left Rotation and Swap Color

Right Left Case (Mirror of case ii) : Perform LR Rotation and Swap Color

**Searching a Node in RB Tree**

We start at the root, and then we compare the value to be searched with the value of the root, if it’s equal we are done with the search if it’s smaller we know that we need to go to the left subtree because in a binary search tree all the elements in the left subtree are smaller and all the elements in the right subtree are larger. Searching an element in the binary search tree is basically this traversal, at each step we go either left or right and at each step we discard one of the sub-trees

**Level Order Traversal**

* Traversing every node level wise
* Nodes at level i are traversed before level i+1

**Applications**

* Most of the self-balancing BST library functions like map, multiset, and multimap in C++ ( or java pacakages like java.util.TreeMap and java.util.TreeSet ) use Red-Black Trees.
* It is used to implement CPU Scheduling Linux. Completely Fair Scheduler uses it.
* It is also used in the K-mean clustering algorithm in machine learning for reducing time complexity.
* Moreover, MySQL also uses the Red-Black tree for indexes on tables in order to reduce the searching and insertion time.

**Complexity Analysis**

**Insertion**

There are three phases to inserting a key into a non-empty tree. The binary search tree insert operation is conducted in the first phase. Because a red-black tree is balanced, the BST insert operation is O(height of tree), which is O(log n). The new node is then colored red in the second stage. This step is O(1) since it only involves changing the value of one node's color field. In the third stage, we restore any red-black characteristics that have been violated. Changing the colors of nodes takes O(1) time. However, we may need to deal with a double-red issue farther along the route from the inserted node to the root. In the worst-case scenario, we wind up fixing a double-red condition all the way from the inserted node to the root. In the worst-case scenario, the recoloring performed during insertion is O(log n) i.e., time for one recoloring x maximum number of recoloring performed. As a result, restoring red-black characteristics takes O(log n), and the overall time for insert is O(log n).

Best Case: In the best case, there is no rotation. Only recoloring takes place. The time complexity is O(log n). Consider the following example.

Worst case: RB trees require a constant (at most 2 for insert) number of rotations. So in the worst case, there will be 2 rotations while insertion. The time complexity is O(log n).

Average Case: Since the average case is the mean of all possible cases, the time complexity of insertion in this case too is O(log n).

**Space Complexity of RB Tree**

The average and worst space complexity of a red-black tree is the same as that of a Binary Search Tree and is determined by the total number of nodes: O(n) because we don't need any extra space to hold duplicate data structures. We arrive to this conclusion because each node has three pointers: left child, right child, and parent. Each node takes up O(1) space. As a result, if the tree has n total nodes, the space complexity is n times O(1), which is O(n). Because there are just two colors, monitoring the color of each node takes only one bit of information per node. Because the tree contains no extra data that distinguishes it as a red-black tree, its memory footprint is nearly comparable to that of a conventional binary search tree. In many circumstances, the extra bit of data may be stored with no extra memory cost.

* **IMPLEMENTATION**

**Important screen shots**

What do you want to do?  
1- Load "EN-US-Dictionary"      2- Print size of the Dictionary  
3- Insert Word                  4- Look-up a Word  
5- Print Tree Height            6- Print Black Height of the Tree  
7- Exit  
> 2  
EN-US-Dictionary currently has 0 words!  
  
What do you want to do?  
1- Load "EN-US-Dictionary"      2- Print size of the Dictionary  
3- Insert Word                  4- Look-up a Word  
5- Print Tree Height            6- Print Black Height of the Tree  
7- Exit  
> 3  
Enter the word you want to insert: Junaid  
"Junaid" inserted Successfully  
  
What do you want to do?  
1- Load "EN-US-Dictionary"      2- Print size of the Dictionary  
3- Insert Word                  4- Look-up a Word  
5- Print Tree Height            6- Print Black Height of the Tree  
7- Exit  
> 3  
Enter the word you want to insert: Jayesh  
"Jayesh" inserted Successfully  
  
What do you want to do?  
1- Load "EN-US-Dictionary"      2- Print size of the Dictionary  
3- Insert Word                  4- Look-up a Word  
5- Print Tree Height            6- Print Black Height of the Tree  
7- Exit  
> 3  
Enter the word you want to insert: Jigar  
"Jigar" inserted Successfully  
  
What do you want to do?  
1- Load "EN-US-Dictionary"      2- Print size of the Dictionary  
3- Insert Word                  4- Look-up a Word  
5- Print Tree Height            6- Print Black Height of the Tree  
7- Exit  
> 2  
EN-US-Dictionary currently has 3 words!  
  
What do you want to do?  
1- Load "EN-US-Dictionary"      2- Print size of the Dictionary  
3- Insert Word                  4- Look-up a Word  
5- Print Tree Height            6- Print Black Height of the Tree  
7- Exit  
> 4  
Enter the word you want to look-up: Junaid  
FOUND "Junaid"!  
  
What do you want to do?  
1- Load "EN-US-Dictionary"      2- Print size of the Dictionary  
3- Insert Word                  4- Look-up a Word  
5- Print Tree Height            6- Print Black Height of the Tree  
7- Exit  
> 4        
Enter the word you want to look-up: Hello  
"Hello" DOES NOT EXIST IN THE DICTIONARY  
  
What do you want to do?  
1- Load "EN-US-Dictionary"      2- Print size of the Dictionary  
3- Insert Word                  4- Look-up a Word  
5- Print Tree Height            6- Print Black Height of the Tree  
7- Exit  
> 5  
2  
  
What do you want to do?  
1- Load "EN-US-Dictionary"      2- Print size of the Dictionary  
3- Insert Word                  4- Look-up a Word  
5- Print Tree Height            6- Print Black Height of the Tree  
7- Exit  
> 6  
1  
  
What do you want to do?  
1- Load "EN-US-Dictionary"      2- Print size of the Dictionary  
3- Insert Word                  4- Look-up a Word  
5- Print Tree Height            6- Print Black Height of the Tree  
7- Exit  
> 1  
EN-US-Dictionary loaded successfully!  
  
What do you want to do?  
1- Load "EN-US-Dictionary"      2- Print size of the Dictionary  
3- Insert Word                  4- Look-up a Word  
5- Print Tree Height            6- Print Black Height of the Tree  
7- Exit  
> 2  
EN-US-Dictionary currently has 97465 words!  
  
What do you want to do?  
1- Load "EN-US-Dictionary"      2- Print size of the Dictionary  
3- Insert Word                  4- Look-up a Word  
5- Print Tree Height            6- Print Black Height of the Tree  
7- Exit  
> 6  
10  
  
What do you want to do?  
1- Load "EN-US-Dictionary"      2- Print size of the Dictionary  
3- Insert Word                  4- Look-up a Word  
5- Print Tree Height            6- Print Black Height of the Tree  
7- Exit  
> 5  
20  
  
What do you want to do?  
1- Load "EN-US-Dictionary"      2- Print size of the Dictionary  
3- Insert Word                  4- Look-up a Word  
5- Print Tree Height            6- Print Black Height of the Tree  
7- Exit  
> 7  
Thank you for using our application! :)

**Code of important functions**

DataStructure.py

| class Node:     # if color = 0 -> red     # if color = 1--> black     def \_\_init\_\_(self, key):  # Constructor         self.key = key  # Node needs a key to be initialized         self.parent = None         self.right = None         self.left = None         self.color = 0   class RedBlackTree:     def \_\_init\_\_(self):  # Constructor         self.nil = Node(None)         self.nil.color = 1  # The root and the nil are black         self.root = self.nil         self.number\_of\_nodes = 0      def search(self, key):         node = self.root          while node != self.nil:  # as long as we didn't reach the end of the tree             if node.key == key:                 return True             elif key < node.key:                 node = node.left             else:                 node = node.right         return False      def insert(self, key):         newNode = Node(str(key).lower())         newNode.left = self.nil         newNode.right = self.nil         node = self.root         parent = None  # TBD          while node != self.nil:  # Find the appropriate parent             parent = node             if newNode.key < node.key:                 node = node.left             else:                 node = node.right         newNode.parent = parent          if parent is None:  # Inserted node is the first node             newNode.color = 1             self.root = newNode             self.number\_of\_nodes += 1             return         elif newNode.key < parent.key:             parent.left = newNode         else:             parent.right = newNode          if newNode.parent.parent is None:  # Parent is the root             self.number\_of\_nodes += 1             return          self.insertFix(newNode)  # Handle cases         self.number\_of\_nodes += 1      # This method handles cases of RB-tree insertions     def insertFix(self, newNode):         while newNode != self.root and newNode.parent.color == 0:  # Loop until we reach the root or parent is black              parentIsLeft = False  # Parent is considered left child by default              # Assign uncle to appropriate node             if newNode.parent == newNode.parent.parent.left:                 uncle = newNode.parent.parent.right                 parentIsLeft = True             else:                 uncle = newNode.parent.parent.left              # Case 1: Uncle is red -> Reverse colors of uncle, parent and grandparent             if uncle.color == 0:                 newNode.parent.color = 1                 uncle.color = 1                 newNode.parent.parent.color = 0                 newNode = newNode.parent.parent              # Case 2: Uncle is black -> check triangular or linear and rotate accordingly             else:                 # Left-right condition (triangular)                 if parentIsLeft and newNode == newNode.parent.right:                     newNode = newNode.parent  # Take care as we made the new node the parent                     self.leftRotate(newNode)                 # Right-Left condition (triangular)                 elif not parentIsLeft and newNode == newNode.parent.left:                     newNode = newNode.parent                     self.rightRotate(newNode)                 # Left-left condition (linear)                 if parentIsLeft:                     newNode.parent.color = 1  # the new parent                     newNode.parent.parent.color = 0  # the new grandparent will be red                     self.rightRotate(newNode.parent.parent)                 # Right-right condition (linear)                 else:                     newNode.parent.color = 1                     newNode.parent.parent.color = 0                     self.leftRotate(newNode.parent.parent)          self.root.color = 1  # Set root to black      def leftRotate(self, node):          y = node.right         node.right = y.left  # connect node to c         if y.left != self.nil:  # connect c to node             y.left.parent = node          y.parent = node.parent  # connect y to node's parent          if node.parent is None:  # connect node's parent to y             self.root = y         elif node == node.parent.left:             node.parent.left = y         else:             node.parent.right = y          y.left = node  # connect y to node         node.parent = y  # connect node to y      def rightRotate(self, node):                 y = node.left         node.left = y.right  # connect node to d         if y.right != self.nil:  # connect d to node             y.right.parent = node         y.parent = node.parent  # connect y to node's parent          if node.parent is None:  # connect b parent to a's parent             self.root = y         elif node == node.parent.left:             node.parent.left = y         else:             node.parent.right = y          y.right = node  # connect y to node         node.parent = y  # connect node to y      # This method returns the height of the tree     def heightOfTree(self, node, sumval):         if node is self.nil:             return sumval         return max(self.heightOfTree(node.left, sumval + 1), self.heightOfTree(node.right, sumval + 1))      # This method returns the black-height of the tree     def getBlackHeight(self):         node = self.root         bh = 0         while node is not self.nil:             node = node.left             if node.color == 1:                 bh += 1         return bh      # Function to print used in debugging     def \_\_printCall(self, node, indent, last):         if node != self.nil:             print(indent, end=' ')  # the default end character is new line             if last:                 print("R----", end=' ')                 indent += "     "             else:                 print("L----", end=' ')                 indent += "|    "              s\_color = "RED" if node.color == 0 else "BLACK"             print(str(node.key) + "(" + s\_color + ")")             self.\_\_printCall(node.left, indent, False)             self.\_\_printCall(node.right, indent, True)      # Function to call print     def print\_tree(self):         self.\_\_printCall(self.root, "", True) |
| --- |

**Main.py**

| import DataStructure as ds  tree = ds.RedBlackTree()  # initialize RB-tree DICTIONARY\_NAME = "EN-US-Dictionary"   def readFile(fileName):     file = open(fileName, "r")     for i in file:         if not tree.search(i.rstrip('\n')):             tree.insert(i.rstrip('\n'))     file.close()   while True:     print("What do you want to do?")     option = input(         "1- Load \"" + DICTIONARY\_NAME + "\"\t2- Print size of the Dictionary\n"         "3- Insert Word             \t4- Look-up a Word\n"         "5- Print Tree Height       \t6- Print Black Height of the Tree\n"         "7- Exit\n"         "> ")      if option == '1':         readFile(DICTIONARY\_NAME + ".txt")         print(DICTIONARY\_NAME + " loaded successfully!")      elif option == '2':         print(DICTIONARY\_NAME + ' currently has ' + str(tree.number\_of\_nodes) + ' words!')      elif option == '3':         s = str(input("Enter the word you want to insert: ")).strip()         if tree.search(s.lower()):             print("\"" + s + "\" is already in the dictionary!")         elif len(s) > 0 and not s.isspace():             tree.insert(s.lower())             print('\"' + s + '\" inserted Successfully')         else:             print('Invalid entry')      elif option == '4':         s = str(input("Enter the word you want to look-up: ")).strip()         if tree.search(s.lower()):             print("FOUND \"" + s + '\"!')         else:             print("\"" + s + '\" DOES NOT EXIST IN THE DICTIONARY')      elif option == '5':         print(tree.heightOfTree(tree.root, 0))      elif option == '6':         print(tree.getBlackHeight())      elif option == '7':         print("Thank you for using our application! :)")         break      print() |
| --- |

EN-US-Dictionary.txt

| Rockne's wimpiest loop's Fargo Eastwood's treat . . . length aloof Mattie's |
| --- |

* **COMPLEXITY ANALYSIS**

Insertion of a word in the Dictionary

Insertion of a word in the dictionary takes place through the RB Tree Insertion Process. Insertion takes place in three phases where the BST insertion takes O (log n). Coloring of node is done in O (1). The Fixup code of Insertion takes best case of O(1) when only color swapping takes place and takes O (log n) if any type of rotation is to be performed. Thus the overall average case time complexity of insertion of a word comes out to be O (log n) which is faster as compared to other data structures.

Searching a word in the Dictionary

To search a word in the Dictionary the program uses Search Operation of Binary Search Tree. The search takes place as follows : if value to be searched is less than the value of the root then the left sub tree of the tree is traversed to find the word, if value to be searched is greater than the value of the root the right subtree is searched. The process is repeated until the node with value is found. If node evaluates to NULL then the word is not found. This process is similar to Binary Search thus the complexity for searching a word is O (log n).

Height of the Tree doesn’t exceeds O (log n).

The Space Complexity for the Dictionary is equivalent to the space complexity of Red Black Tree which is O (n).

* **CONCLUSION**

The project implements a dictionary of words using Red Black Tree which provides fast insertion, searching of words in the dictionary. Red Black Tree is a self balancing tree similar to BST with one added property of color to maintain balancing. Insertion of word takes O (log n), Searching of word takes O (log n) and the space complexity of the dictionary is O (n). Thus the Dictionary is capable of storing and retrieving huge amount of words in an efficient manner.

**Comparison of Operations on Dictionary**

| **OPERATION** | **AVERAGE CASE** | **WORST CASE** |
| --- | --- | --- |
| Space | O(n) | O(n) |
| Search | O(log n) | O(log n) |
| Insert | O(log n) | O(log n) |