**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**:

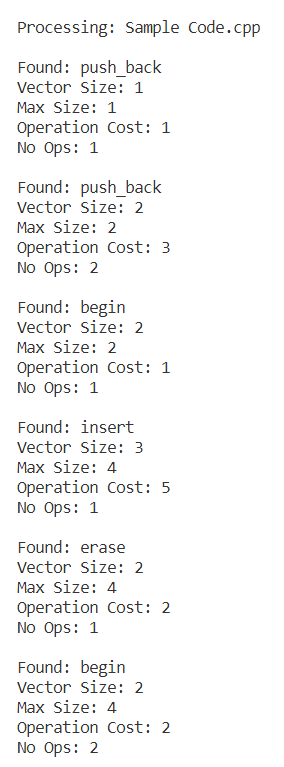
Amortized.cpp

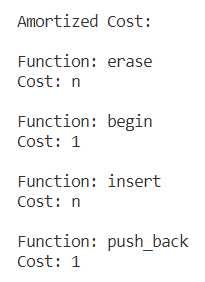
| #include <iostream> #include <string> #include <cstring> #include <fstream> #include <cstdlib> #include <unordered\_map>  bool searchLine(std::string line, std::string key) {  return line.find(key) != std::string::npos; }  class DataStructure {  public:  virtual void process(std::ifstream &file, std::string &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>.