FINAL PROJECT

GROUP MEMBERS
PHURILAP KITLERTPAISAN 6680251
ACHIRA PORNPUTTICHAI 6680996
CHAIYAPAT TRIYANAM 6680730

PRESENTED BY GROUP 10

OVERVIEW

Employee Management System

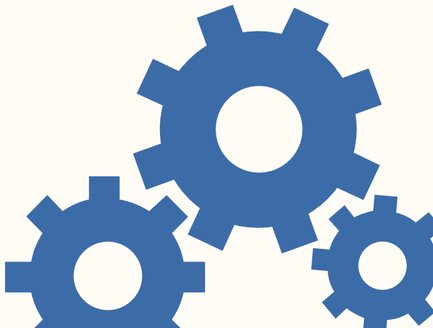
-Stores employee data using array pointer
-Uses BST to search for employee data
-Queue for hiring candidates(FIFO)
-Sort employees by salary
-Update employee salary

HOW IT HELPS SDG 8

- **Employee Management:**
  - The Employee class stores key details (name, salary, experience) for clear records and fair treatment.
  - The Company class manages these records, including adding new employees and updating their information.

- **Fair Hiring Process:**
  - The Queue class handles job candidates in a first-in, first-out manner, ensuring every applicant has an equal opportunity.

- **Efficient Data Handling:**
  - Sorting methods (like bubble sort) help organize employee records for easy review.
  - Other data structures, such as a Binary Search Tree (BST), can further optimize searching and sorting, enhancing performance as data grows.



REQUIREMENTS

1) ARRAY & POINTERS

2) OVERLOADING OPERATORS

3) SORTING

4) BST

5) QUEUE

```
11  class Company {
12  private:
13      Employee** employees;
14      CompanyBST employeeBST;
15      Queue candidateQueue;
16      int capacity;
17      int count;
18
19  public:
20      Company(int cap = 10) : capacity(cap), count(0) {
21          employees = new Employee*[capacity];
22      }
23
24      ~Company() {
25          for(int i = 0; i < count; i++){
26              delete employees[i];
27          }
28          delete[] employees;
29      }
```

ARRAY & POINTERS

The program uses Employee** employees to create an array of pointer. Each one will point to an employee object.

When company is constructed, it creates space for 10 pointers.

When destructor is called, it first deletes the employee object inside the array. Once the array is empty, it will then delete the whole array. This will ensure everything is deleted and no memory leak.

```
void Employee::operator++(){
    experience++;
}

void Employee::operator+=(double amt){
    salary+= amt;
}
```

OPERATOR OVERLOADING

The program has 2 operator overloading:

- operator++ is for increasing experience by 1
- operator += is used to increase the salary by a specific amount.

SORTING.H

```
void bubbleSort(Employee* employees[], int count) {
    int n = count;
    int swapped;

    do {
        swapped = 0; //0 mean no swap
        for (int i = 0; i < n - 1; i++) {
            if (employees[i]->getSalary() > employees[i + 1]->getSalary()) {
                Employee* temp = employees[i];
                employees[i] = employees[i + 1];
                employees[i + 1] = temp;
                swapped = 1;
            }
        }
        n--;
    } while (swapped != 0);
}

#endif
```

Sorting algorithm:Bubble sort

- Sorts Employee by their salaries in ascending order.
- Compares salaries of the employee using getSalary().
- Swaps the employees using temporary variable(temp) to places the higher salary after the lower employee
- Sorts within the original array.

BST.H

```
struct TreeNode {
    Employee* employee;
    TreeNode* left;
    TreeNode* right;

    TreeNode(Employee* emp) : employee(emp), left(nullptr), right(nullptr) {}
};

class CompanyBST {
private:
    TreeNode* root;

    void insertNode(TreeNode*& node, Employee* emp){
        if(!node){
            node = new TreeNode(emp);
        } else if(emp->getName() < node->employee->getName()){
            insertNode(node->left, emp);
        } else{
            insertNode(node->right, emp);
        }
    }
```

Structure (TreeNode):

- The TreeNode struct defines the building blocks of the BST. Each node holds an Employee pointer, a left child pointer, and a right child pointer.
- The constructor TreeNode(Employee* emp) initializes a new node with a given employee and sets both child pointers to nullptr

Insertion (insertNode):

- The insertNode function recursively adds new employees to the BST.

```
Employee* findEmployee(TreeNode* node, const std::string& name) const{
    if(!node) return nullptr;
    if(name == node->employee->getName()) return node->employee;
    if(name < node->employee->getName()) return findEmployee(node->left, name);
    return findEmployee(node->right, name);
}

void displayEmployees(TreeNode* node) const{
    if(node){
        displayEmployees(node->left);
        node->employee->display();
        displayEmployees(node->right);
    }
}

void freeTree(TreeNode* node){
    if(node){
        freeTree(node->left);
        freeTree(node->right);
        delete node;
    }
}
```

```
52      public:
53          CompanyBST() : root(nullptr) {}
54          ~CompanyBST() { freeTree(root); }
55
56          void addEmployee(Employee* emp){
57              insertNode(root, emp);
58          }
59
60          Employee* findEmployee(const std::string& name) const{
61              return findEmployee(root, name);
62          }
63
64          void displayEmployees() const{
65              displayEmployees(root);
66          }
67      };
68
69      #endif
```

BST.H

Searching (findEmployee):

- The findEmployee function searches for an employee by name.

Memory Management (freeTree):

- The freeTree function recursively deallocates all nodes in the BST, preventing memory leaks.
- It uses post order traversal to delete the child nodes before the parent node.

QUEUE.H

```
#ifndef QUEUE_H
#define QUEUE_H

#include "employee.h"
using namespace std;

class Queue {
private:
    Employee** candidates;
    int capacity;
    int front;
    int rear;
    int count;

public:
    Queue(int cap = 10){
        capacity = cap;
        candidates = new Employee*[capacity];
        front = 0;
        rear = 0;
        count = 0;
    }

```

- Create a header file **(queue.h)** that defines a Queue class for managing job candidates.
- The Queue stores pointers to Employee objects

- It uses a fixed-size dynamic array with a default capacity The queue follows a FIFO order: the first candidate added is the first one removed

- Include basic methods:
  - enqueue: Add a candidate to the queue.
  - dequeue: Remove and return the first candidate in the queue.
  - isEmpty: Check if the queue has no candidates.

- Use header guards (#ifndef, #define, #endif) to prevent multiple inclusions of the file.

QUEUE.H

- Includes a destructor that frees the dynamically allocated array of Employee*.

- Provides an enqueue(Employee* candidate) method:
  - Adds a candidate if the queue isn't full.
  - Updates the rear index (with modular arithmetic) and increments count.
  - Displays a message if the queue is full.
- Offers a dequeue() method:
  - Removes and returns the front candidate if the queue isn't empty.
  - Updates the front index (also with modular arithmetic) and decrements count.
  - Prints a message and returns nullptr if the queue is empty.
- Implements an isEmpty() const method that returns true if count == 0.

```
24          ~Queue(){
25              delete[] candidates;
26          }
27
28          void enqueue(Employee* candidate){
29              if(count < capacity){
30                  candidates[rear] = candidate;
31                  rear = (rear + 1) % capacity;
32                  count++;
33              } else{
34                  cout<<"Queue is full."<<endl;
35              }
36          }
37
38          Employee* dequeue(){
39              if(count > 0){
40                  Employee* candidate = candidates[front];
41                  front = (front + 1) % capacity;
42                  count--;
43                  return candidate;
44              } else{
45                  cout<<"Queue is empty."<<endl;
46                  return nullptr;
47              }
48          }
49
50          bool isEmpty() const{
51              return count == 0;
52          }
53      };
54
55      #endif
```

MAIN.CPP (FEATURES)

```
int main(int argc, char* argv[]){
    Company myCompany;

    for(int i = 1; i < argc; i += 3){
        string name = argv[i];
        int experience = atoi(argv[i+1]);
        double salary = atof(argv[i+2]);
        myCompany.addEmployee(name, experience, salary);
    }
```

- argc/argv is used to take command-line arguments in the terminal, the loop process input in triples(name, exp, sal) from the i+=3 in for loop.
- the addEmployee function will store input in the Company array. which can be used later for other functions like in BST and queue.

MAIN.CPP (FEATURES)

```
//Display in BST order
cout<<"\nEmployees BST order:\n";
myCompany.displayEmployeesBST();

//Display original array
cout<<"\nEmployees original order:\n";
myCompany.displayEmployeesArray();

//Sort and display by salary
myCompany.sortEmployeesBySalary();
cout<<"\nEmployees sorted by salary:\n";
myCompany.displayEmployeesArray();
```

- The displayEmployeesBST() function will print out the data using the In Order(Alphabetically A-Z)
- displayEmployeesArray() displays using the insertion order. So whatever we input, it will be put to an array, and will print that exact order.
- Sort and display by Salary: first we use sortEmployeesBySalary() function calls the bubble sort function. After sorting, the program will then display the sorted array.

MAIN.CPP (FEATURES)

```
//Employee update
Employee* emp = myCompany.getEmployee(1);
if(emp){
    cout<<"\nUpdating experience and salary for "<<emp->getName()<<endl;
    *emp += 4500;
    ++(*emp);
    emp->display();
}
```

EMPLOYEE UPDATE

- getEmployee(1) gets the employee in index 1(2nd employee)
- uses getName function to tell us which employee is getting updated
- operator+= is used to increase salary by a set amount(4500 baht for this case)
- operator++ is called for increasing experience by 1 year
- finally, it display the updated array of the employee

```
//BST search
string searchName = argv[1];
Employee* found = myCompany.findEmployee(searchName);
if(found){
    cout<<"\nFound via BST search:\n";
    found->display();
}
```

BST SEARCH

- uses findEmployee() to search for argv[1]. if it exists, it will cout “Found via BST search:” and then the details of that employee.

MAIN.CPP (FEATURES)

```
//Process queue
cout<<"\nProcessing candidate queue:\n";
while(myCompany.hasCandidates()){
    Employee* candidate = myCompany.processNextCandidate();
    cout<<"Processed: "<<candidate->getName()<<endl;
}
```

- hasCandidates() is for checking if there's still candidates in the array. While it is still true, it will call processNextCandidate
- processNextCandidate will use the dequeue() function and then display what candidate is dequeued.
- Queue is First in First Out(FIFO) so the first candidate that got enqueue will be dequeued first.

TEST CASES

Normal input:
./final "John" 5 50000 "Alice" 3 45000 "Bob" 7 60000

Maximum:
./final "E1" 1 10000 "E2" 2 20000 "E3" 3 30000 "E4" 4 40000 "E5" 5 50000 "E6" 6 60000 "E7" 7 70000 "E8" 8 80000 "E9" 9 90000 "E10" 10 100000

Overcapacity:
./final "E1" 1 10000 "E2" 2 20000 "E3" 3 30000 "E4" 4 40000 "E5" 5 50000 "E6" 6 60000 "E7" 7 70000 "E8" 8 80000 "E9" 9 90000 "E10" 10 100000 "E11" 11 110000

THANK YOU