```
#include<iostream>
#include<string>

using namespace std;

class user {
   long int mno;
   string name;
   float bill;

   user() {
      mno = 0;
      name = " ";
      bill = 0;
   }

   friend class record;
};

class record {
   user u[10];
public:

   int n;
   void accept();
   void heapsort();
   void adjust(int i, int n);
   void quickSort(int low, int high);
   int partition(int low, int high);
   void binarysearch(int low,int high);
   void binaryrec(int low, int high);
```

```
    void linearSearch();
    void display();
};

void record::accept() {
    cout << "Number of members you want to add? " << endl;
    cin >> n;
    for (int i = 0; i < n; i++) {
        cout << "Mobile number of user " << i + 1 << endl;
        cin >> u[i].mno;
        cout << "Name of user " << i + 1 << endl;
        cin >> u[i].name;
        cout << "Bill amount of user " << i + 1 << endl;
        cin >> u[i].bill;
    }
}

void record::adjust(int i, int n) {
    while (2 * i + 1 <= n) {
        int j = 2 * i + 1;
        if (j + 1 <= n && u[j + 1].bill > u[j].bill) {
            j = j + 1;
        }
        if (u[i].bill >= u[j].bill) {
            break;
        } else {
            user temp = u[i];
            u[i] = u[j];
            u[j] = temp;
            i = j;
        }
    }
}

void record::heapsort() {
    for (int i = (n / 2) - 1; i >= 0; i--) {
        adjust(i, n - 1);
    }
    for (int i = n - 1; i > 0; i--) {
        user t = u[0];
        u[0] = u[i];
        u[i] = t;
        adjust(0, i - 1);
    }
}

int record::partition(int low, int high) {
    long int pivot = u[high].mno;
    int i = (low - 1);

    for (int j = low; j < high; j++) {
        if (u[j].mno <= pivot) {
            i++;

            user temp = u[i];
            u[i] = u[j];
            u[j] = temp;
        }
    }

    user temp = u[i + 1];
    u[i + 1] = u[high];
    u[high] = temp;
    return (i + 1);
}

void record::quickSort(int low, int high) {
    if (low < high) {

        int pi = partition(low, high);



        quickSort(low, pi - 1);
        quickSort(pi + 1, high);
    }
}

void record::binarysearch(int low,int high) {
    int x;
    cout << "Enter mobile number you want to search: " << endl;
    cin >> x;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (u[mid].mno == x) {
            cout << "USER FOUND" << endl;
            cout << "Name: " << u[mid].name << endl;
            cout << "Mobile: " << u[mid].mno << endl;
            cout << "Bill: " << u[mid].bill << endl;
            return;
        }
    if (u[mid].mno < x)
      low = mid + 1;
    else
      high = mid - 1;
  }

  cout << "User not found!" << endl;
}

void record::binaryrec( int low, int high) {
  int x;
  cout << "Enter mobile number you want to search: " << endl;
  cin >> x;

  if (low <= high) {
    int mid = low + (high - low) / 2;

    if (u[mid].mno == x) {
      cout << "USER FOUND" << endl;
      cout << "Name: " << u[mid].name << endl;
      cout << "Mobile: " << u[mid].mno << endl;
      cout << "Bill: " << u[mid].bill << endl;
      return;
    }

    if (u[mid].mno < x)
       return binaryrec(mid + 1, high);
    else
     return binaryrec(low, mid - 1);
  }
```

```
    cout << "User not found!" << endl;
}

void record::linearSearch() {
    long int target;
    bool flag = false;

    cout << "Enter mobile number to search for: ";
    cin >> target;

    for (int i = 0; i < n; i++) {
        if (u[i].mno == target) {
            cout << "USER FOUND" << endl;
            cout << "Name: " << u[i].name << endl;
            cout << "Mobile: " << u[i].mno << endl;
            cout << "Bill: " << u[i].bill << endl;
            flag = true;
            break;
        }
    }

    if (!flag) {
        cout << "User not found!" << endl;
    }
}

void record::display() {
    cout << "ENTERED DATA: " << endl;
    cout << "Mobile no. for user\t\tName for user\t\tBill for user" << endl;
    for (int i = 0; i < n; i++) {
        cout << "---------------------------------------------------" << endl;
        cout << u[i].mno << "\t\t" << u[i].name << "\t\t" << u[i].bill << endl;
    }
}

int main() {
    record r;
    int choice;

    do {
        cout << "\nMENU:" << endl;
        cout << "1. Accept Data" << endl;
        cout << "2. Display Data" << endl;
        cout << "3. Sort Data by Bill (HeapSort)" << endl;
        cout << "4. Sort Data by Mobile Number (QuickSort)" << endl;
        cout << "5. Binary Search (by mobile number)" << endl;
        cout << "6. Binary Search Recursive (by mobile number)" << endl;
        cout << "7. Linear Search (by mobile number)" << endl;
        cout << "8. Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                r.accept();
                break;
            case 2:
                r.display();
                break;
            case 3:
                r.heapsort();  // Sort the data by bill using HeapSort
                cout << "Data after sorting by bill:" << endl;
                r.display();
                break;
            case 4:
                r.quickSort(0, r.n - 1);  // Sort the data by mobile number using QuickSort
                cout << "Data after sorting by mobile number:" << endl;
                r.display();
                break;
            case 5:
                r.binarysearch(0, r.n - 1);
                break;
            case 6:
                r.binaryrec(0, r.n - 1);
                break;
            case 7:
                r.linearSearch();  // Perform linear search for mobile number
                break;
            case 8:
                cout << "Exiting program..." << endl;
                break;
            default:
                cout << "Invalid choice, please try again!" << endl;
        }

    } while (choice != 7);

    return 0;
}
```

```
#include<iostream>
#include<string>
using namespace std;

class node {
  int id;
  string name;
  node *next;
  friend class graph;
  friend class stack;
};

class queue {
  int q[20];
  int front;
  int rear;
  friend class graph;

public:
  queue() {
    front = 0;
    rear = -1;
  }

  void push_q(int);
  int pop_q();
  int empty_q();
};

int queue::empty_q() {
  return front > rear;
}

void queue::push_q(int temp) {
    rear++;
    q[rear] = temp;
}

int queue::pop_q() {
    if (front > rear) return -1;
    return q[front++];
}

class stack {
    int st[20];
    int top;
    friend class graph;

public:
    stack() {
        top = -1;
    }

    void push(int);
    int pop();
    int empty();
};

void stack::push(int temp) {
    top++;
    st[top] = temp;
}
```

```
int stack::pop() {
    int temp = st[top];
    top--;
    return temp;
}

int stack::empty() {
    if (top == -1) {
        return 0;
    }
    else
        return 1;
}

class graph {
    node* head[20];
    int visited[20];

public:
    graph() {
        cout << "Enter the number of vertices: ";
        cin >> n;
        for (int i = 0; i < n; i++) {
            head[i] = new node();
            cout << "Enter the name of user " << i << ": ";
            cin >> head[i]->name;
            head[i]->id = i;
            head[i]->next = NULL;
        }
    }
    void create_adj_list();
    void display();
    void DFT();
    void DFT_rec(int v);
    void DFT_non_rec();
    void BFS_non_rec();
    int n;
};

void graph::create_adj_list() {
    int v;
    node* curr;

    for (int i = 0; i < n; i++) {
        //node* temp = head[i];
        cout << "\nEnter vertices connected to user " << i << " (vertex ID): \n";

        do {
            cout << "Enter the connected vertex ID (or -1 to stop): ";
            cin >> v;

            if (v == -1) {
                break;
            }

            if (v < 0 || v >= n) {
                cout << "Invalid vertex ID! Please enter a valid vertex ID between 0 and " << n - 1 << ".\n";
                continue;
            }
            if (i == v) {
                cout << "Self loop not allowed!" << endl;
            } else {
                curr = new node();
                curr->id = v;
                curr->name = head[v]->name;
                curr->next = NULL;

                node* adjListTemp = head[i];
                while (adjListTemp->next != NULL) {
                    adjListTemp = adjListTemp->next;
                }
                adjListTemp->next = curr;
            }

            cout << "Do you want to add more adjacent vertices for user " << i << "? (y/n): ";
            char ch;
            cin >> ch;
            if (ch != 'y') break;

        } while (true);
    }
}

void graph::display() {
    for (int i = 0; i < n; i++) {
        if (head[i] == NULL) {
            cout << "Vertex " << i << " has no data.\n";
            continue;
        }
    node *temp = head[i]->next;
    cout << "\nThe connections of user " << head[i]->id << " (" << head[i]->name << ") are:\n";

    if (temp != NULL) {
      while (temp != NULL) {
        cout << "User ID: " << temp->id << ", User Name: " << temp->name << endl;
        temp = temp->next;
      }
    } else {
      cout << "No connections.\n";
    }
  }
}

void graph::DFT() {
  int v;
  cout << "Enter starting vertex for DFT: ";
  cin >> v;

  for (int i = 0; i < n; i++) {
    visited[i] = 0;
  }

  cout << "DFS traversal: ";
  visited[v] = 1;
  cout << head[v]->name << " ";
  DFT_rec(v);
  cout << endl;
}

void graph::DFT_rec(int v) {
    node* temp = head[v]->next;
    while (temp != NULL) {
        if (visited[temp->id] == 0) {
            visited[temp->id] = 1;
            cout << temp->name << " ";
            DFT_rec(temp->id);
        }
        temp = temp->next;
    }
}

void graph::DFT_non_rec() {
    stack s;
    int v;
    cout << "Enter start vertex for DFT(non recursive): ";
    cin >> v;

    for (int i = 0; i < n; i++) {
        visited[i] = 0;
    }
    visited[v] = 1;
    s.push(v);

    while (s.empty() != 0) {
        v = s.pop();
        cout << head[v]->name << " ";
        node* temp = head[v]->next;
        while (temp != NULL) {
            if (!visited[temp->id]) {
                visited[temp->id] = 1;
                s.push(temp->id);
            }
            temp = temp->next;
        }
    }
}

void graph::BFS_non_rec() {
    queue q;
    int v;
    cout << "Enter start vertex for BFS (non-recursive): ";
    cin >> v;

    for (int i = 0; i < n; i++) {
        visited[i] = 0;
    }

    visited[v] = 1;
    q.push_q(v);
    cout << "BFS Traversal: ";

    while (!q.empty_q()) {
        v = q.pop_q();
        cout << head[v]->name << " ";
        node* temp = head[v]->next;
        while (temp != NULL) {
            if (!visited[temp->id]) {
                visited[temp->id] = 1;
                q.push_q(temp->id);
            }
            temp = temp->next;
        }
```

```
  }
  cout << endl;
}

int main() {
  graph g;
  int choice;

  do {
    cout << "\nGraph Operations Menu:\n";
    cout << "1. Create Graph\n";
    cout << "2. Display Graph\n";
    cout << "3. Recursive Depth-First Traversal (DFT)\n";
    cout << "4. Non-Recursive Depth-First Traversal (DFT)\n";
    cout << "5. Non-Recursive Breadth-First Traversal (BFS)\n";
    cout << "6. Exit\n";
    cout << "Enter your choice: ";
    cin >> choice;

    switch (choice) {
      case 1:
        cout << "Creating Graph...\n";
        g.create_adj_list();
        break;

      case 2:
        cout << "Displaying Graph...\n";
        g.display();
        break;

      case 3:
        cout << "Recursive DFT...\n";
        g.DFT();
        break;

      case 4:
        cout << "Non-Recursive DFT...\n";
        g.DFT_non_rec();
        break;

      case 5:
        cout << "Non-Recursive BFS...\n";
        g.BFS_non_rec();
        break;

      case 6:
        cout << "Exiting program...\n";
        break;

      default:
        cout << "Invalid choice! Please enter a valid option.\n";
    }
  } while (choice != 6);

  return 0;
}
```

```
#include <iostream>
using namespace std;

class avl_node {
    string word, meaning;
    avl_node *left, *right;
public:
    friend class avlTree;
};

class avlTree {
    avl_node *root;
public:
    avlTree() { root = NULL; }

    int height(avl_node *);
    int diff(avl_node *);
    avl_node *rr_rotation(avl_node *);
    avl_node *ll_rotation(avl_node *);
    avl_node *rl_rotation(avl_node *);
    avl_node *lr_rotation(avl_node *);
    avl_node *balance(avl_node *);
    avl_node *insert(avl_node *, avl_node *);
    void insert();
    void display(avl_node *);
    void display() { display(root); }
};

avl_node *avlTree::ll_rotation(avl_node *parent) {
    avl_node *temp = parent->left;
    parent->left = temp->right;
    temp->right = parent;
    return temp;
}

avl_node *avlTree::rr_rotation(avl_node *parent) {
    avl_node *temp = parent->right;
    parent->right = temp->left;
    temp->left = parent;
    return temp;
}

avl_node *avlTree::lr_rotation(avl_node *parent) {
    avl_node *temp = parent->left;
    parent->left = rr_rotation(temp);
    return ll_rotation(parent);
}

avl_node *avlTree::rl_rotation(avl_node *parent) {
    avl_node *temp = parent->right;
    parent->right = ll_rotation(temp);
    return rr_rotation(parent);
}

int avlTree::height(avl_node *temp) {
    if (!temp) return 0;
    int l_height = height(temp->left);
    int r_height = height(temp->right);
    return max(l_height, r_height) + 1;
}

int avlTree::diff(avl_node *temp) {
    return height(temp->left) - height(temp->right);
}

avl_node *avlTree::balance(avl_node *temp) {
    int bal_factor = diff(temp);
    if (bal_factor > 1) {
        if (diff(temp->left) > 0) {
            temp = ll_rotation(temp);
        } else {
            temp = lr_rotation(temp);
        }
    } else if (bal_factor < 0) {
        if (diff(temp->right) > 0) {
            temp = rl_rotation(temp);
        } else {
            temp = rr_rotation(temp);
        }
    }
    return temp;
}

avl_node *avlTree::insert(avl_node *root, avl_node *temp) {
    if (!root) {
        root = new avl_node;
        root->word = temp->word;
        root->meaning = temp->meaning;
        root->left = root->right = NULL;
        return root;
    }
    if (temp->word < root->word) {
        root->left = insert(root->left, temp);
    } else if (temp->word > root->word) {
        root->right = insert(root->right, temp);
    }
    return balance(root);
}

void avlTree::insert() {
    avl_node *temp = new avl_node;
    cout << "Enter word: ";
    cin >> temp->word;
    cout << "Enter meaning: ";
    cin.ignore();
    getline(cin, temp->meaning);
    temp->left = temp->right = NULL;
    root = insert(root, temp);
    cout << "Word inserted successfully!\n";
}

void avlTree::display(avl_node *temp) {
    if (temp) {
        display(temp->left);
        cout << temp->word << " : " << temp->meaning << endl;
        display(temp->right);
    }
}

int main() {
    avlTree tree;
    int choice;
    do {
        cout << "\nDictionary AVL Tree";
        cout << "\n1. Insert Word";
        cout << "\n2. Display Dictionary";
        cout << "\n3. Exit";
        cout << "\nEnter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                tree.insert();
                break;
            case 2:
                tree.display();
                break;
            case 3:
                cout << "Exiting...\n";
                break;
            default:
                cout << "Invalid choice! Try again.\n";
        }
    } while (choice != 3);
    return 0;
}
```

```
#include <iostream>
#define MAX 10

using namespace std;

class HashTable {
  int table[MAX];

public:
  HashTable() {
    for (int i = 0; i < MAX; i++)
      table[i] = -1;
  }

  void insert_linear_prob(int key) {
    int loc = key % MAX;
    int i = loc;

    if (table[loc] == -1) {
      table[loc] = key;
      cout << "Inserted " << key << " at index " << loc << endl;
      return;
    }

    i = (loc + 1) % MAX;
    while (i != loc) {
      if (table[i] == -1) {
        table[i] = key;
        cout << "Inserted " << key << " at index " << i << endl;
        return;
      }
```

```
            i = (i + 1) % MAX;
        }

        cout << "Hash is full! Cannot insert " << key << endl;
    }

    void insert_linear_prob_with_replacement(int key) {
        int loc = key % MAX;
        int i = loc;

        if (table[loc] == -1) {
            table[loc] = key;
            cout << "Inserted " << key << " at index " << loc << endl;
            return;
        }

        if (table[loc] % MAX != loc) {
            swap(table[loc], key);
            cout << "Replaced index " << loc << " with " << table[loc] << " and reinserted " << key <<
endl;
        }

        i = (loc + 1) % MAX;
        while (i != loc) {
            if (table[i] == -1) {
                table[i] = key;
                cout << "Inserted " << key << " at index " << i << endl;
                return;
            }
            i = (i + 1) % MAX;
        }
        cout << "Hash is full! Cannot insert " << key << endl;
    }



    void display() {
        cout << "\nFinal Hash Table:" << endl;
        for (int i = 0; i < MAX; i++)
        cout << "Index " << i << " : " << table[i] << endl;


    }
};

int main() {
    HashTable ht;
    int n, key, choice;

    cout << "Enter the number of keys to insert: ";
    cin >> n;

    cout << "Choose insertion method:\n1. Linear Probing without Replacement\n2. Linear
Probing with Replacement\nEnter choice: ";
    cin >> choice;

    for (int i = 0; i < n; i++) {
        cout << "Enter key " << i + 1 << ": ";
        cin >> key;

        if (choice == 1) {
            ht.insert_linear_prob(key);
        } else if (choice == 2) {
            ht.insert_linear_prob_with_replacement(key);
        } else {
            cout << "Invalid choice!" << endl;
            return 0;
        }
    }

    ht.display();
    return 0;
}
```

```
#include <vector>

#include <iostream>

using namespace std;


int n;

vector<int> x;

int Ncount = 0;


class nqclass{

public:


bool place(int k, int i)

{

  for (int j = 1; j < k; j++)

  {

    if (x[j] == i || abs(x[j] - i) == abs(j - k))

    {

      return false;

    }

  }

  return true;

}

void printSolution()

{

  for (int i = 1; i <= n; i++)

  {

    for (int j = 1; j <= n; j++)

    {

      if (x[i] == j)

        cout << "Q ";
```

```
            else
                cout << ". ";
        }
        cout << endl;
    }
    cout << endl;
}

void nQueens(int k, int n)
{

    for (int i = 1; i <= n; i++)
    {
        if (place(k, i))
        {
            x[k] = i;
            if (k == n)
            {
                Ncount++;
                printSolution();
            }
            else
            {
                nQueens(k + 1, n);
            }
        }
    }
}
};

int main()
```

```
{
   nqclass q;
   cout << "Enter the number of queens: ";
   cin >> n;
   x.resize(n + 1);
   q.nQueens(1, n);
   cout << "Number of solutions are: " << Ncount;
   return 0;
}
```

```
#include <iostream>
using namespace std;

class graph {
    int cost[10][10], nearest[10], t[10][10];
    int n, i, j, k, startv;

public:
    void create();
    void display();
    void prims();
};

void graph::create() {
    char ch;
    cout << "Enter number of vertices in the graph: " << endl;
    cin >> n;

    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            cost[i][j] = 999;
        }
    }

    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n; j++) {
            cout << "Is there a connection between " << i << " and " << j << "? (y/n): ";
            cin >> ch;

            if (ch == 'y') {
                cout << "Enter the distance between " << i << " and " << j << ": ";
                cin >> cost[i][j];
                cost[j][i] = cost[i][j];
            } else {
                cost[i][j] = 999;
                cost[j][i] = 999;
            }
        }
    }
}

void graph::display() {
    cout << "Adjacency Matrix:" << endl;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            cout << cost[i][j] << '\t';
        }
        cout << endl;
    }
}

void graph::prims() {
    int mincost = 0, min, r = 0, j;

    cout << "Enter the start vertex: ";
    cin >> startv;
    nearest[startv] = -1;

    for (i = 0; i < n; i++) {
        if (i != startv) {
            nearest[i] = startv;
        }
    }

    for (i = 0; i < n - 1; i++) {
        min = 999;

        for (k = 0; k < n; k++) {
            if (nearest[k] != -1 && cost[k][nearest[k]] < min) {
                j = k;
                min = cost[k][nearest[k]];
            }
        }

        t[r][0] = nearest[j];
        t[r][1] = j;
        t[r][2] = min;
        r++;

        mincost += cost[j][nearest[j]];
        nearest[j] = -1;

        for (k = 0; k < n; k++) {
            if (nearest[k] != -1 && cost[k][nearest[k]] > cost[k][j]) {
                nearest[k] = j;
            }
        }
    }

    cout << "Minimum spanning tree cost: " << mincost << endl;
}

int main() {
    graph g1;

    g1.create();

    cout << "Displaying the matrix: " << endl;

    g1.display();

    g1.prims();

    return 0;

}
```

```
#include <iostream>

#include <vector>

using namespace std;


void findSelectedItems(vector<vector<int>> B, vector<int> wt, int n, int W);


int knapsack(int W, vector<int> wt, vector<int> val, int n) {

    vector<vector<int>> B(n + 1, vector<int>(W + 1, 0));


    for (int i = 1; i <= n; i++) {

        for (int w = 0; w <= W; w++) {

            if (wt[i - 1] <= w) {

                B[i][w] = max(val[i - 1] + B[i - 1][w - wt[i - 1]], B[i - 1][w]);

            } else {

                B[i][w] = B[i - 1][w];

            }

        }

    }


    cout << "Maximum value in Knapsack = " << B[n][W] << endl;


    findSelectedItems(B, wt, n, W);


    return B[n][W];

}


void findSelectedItems(vector<vector<int>> B, vector<int> wt, int n, int W) {

    int i = n, k = W;

    vector<int> selectedItems;


    while (i > 0 && k > 0) {
        if (B[i][k] != B[i - 1][k]) {
            selectedItems.push_back(i);
            k -= wt[i - 1];
        }
        i--;
    }

    cout << "Selected items: ";
    for (int item : selectedItems) {
        cout << item << " ";
    }
    cout << endl;
}

int main() {
    int n, W;
    cout << "Enter number of items: ";
    cin >> n;

    vector<int> val(n), wt(n);

    cout << "Enter weights of items: ";
    for (int i = 0; i < n; i++) {
        cin >> wt[i];
    }

    cout << "Enter values of items: ";
    for (int i = 0; i < n; i++) {
        cin >> val[i];
    }
    cout << "Enter knapsack capacity: ";
    cin >> W;

    knapsack(W, wt, val, n);

    return 0;
}
```