NAME: ABDUR REHMAN AZIZ
S/O: M.AZIZ
SEAT NO: B20102009

2ND YEAR SEC-B

(402)
DATA STRUCTURES
(CODING ASSIGNMENT)

JAGGED ARRAYS /

N-DIMENSION ARRAYS

1. SIR'S CODE

myCodes > nDimensionArray > jaggedArraySir.cpp > ...

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


int main() {

    int c[4][4];
    //int** ja = new int*[4];
    int **a,**b;
    a = new int*[4];
    b = new int*[4];

    printf("&a = %u a = %u , &b = %u , b = %u \n",&a,a,&b,b);
    // int C[4]={3,6,2,5};
    int C[4] = {4, 4, 4, 4};
    for (int i=0; i<4; i++) {
        a[i] = new int(C[i]);
        b[i] = new int(C[i]);
    }
    //  int a[4][4],b[4][4]

    for (int x=0; x<4; x++) {
        for (int y=0; y<C[x]; y++) {
            a[x][y] = b[x][y] = x * C[x] + y;
        }
    }

    for (int x=0; x<4; x++) {
        printf("&a[%d]= %u , %u , &b[%d] = %u , %u \n",x,&a[x],a[x],x,&b[x],b[x]);
    }

    for (int x=0; x<4; x++) {
```

`

```
    for (int x=0; x<4; x++) {
        printf("&a[%d]= %u , %u , &b[%d] = %u , %u \n",x,&a[x],a[x],x,&b[x],b[x]);
    }

    for (int x=0; x<4; x++) {
        for (int y=0; y<C[x]; y++) {
            printf("a[%d][%d] = %d @ %u\n",x,y,a[x][y],&a[x][y]);
        }
    }

    for (int x=0; x<4; x++) {
        for (int y=0; y<4; y++) {
            printf("c[%d][%d] = %d @ %u\n",x,y,c[x][y],&c[x][y]);
        }
    }

    for (int x=0; x<4; x++) {
        for (int y=0; y<C[x]; y++) {
            printf("b[%d][%d] = %d @ %u\n",x,y,b[x][y],&b[x][y]);
        }
    }
    return 0;
}
```

`

2. ROW MAJOR

myCodes > nDimensionArray > rowMajor.cpp > main()

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

int main() {

    int N;
    cout << "Enter number of dimensions: ";
    cin >> N;
    int *S = new int[N];

    for (int i = 0; i < N; i++) {
        cout << "Enter size of Dimension " << i+1 << ": ";
        cin >> S[i];
    }

    int totalValues = 1;
    for (int i = 0; i < N; i++) {
        totalValues = totalValues * S[i];
    }

    int e;
    int* I = new int[N];
    for (int k = 0; k < N; k++) {
        cout << "Enter Index of Dimension: " << k+1 << endl;
        cin >> e;
        I[k] = e;
    }

    int s = 1;
    int alpha = 0; // require index

    for (int i=0 ; i<N; i++) {
```

`

```
    for (int i=0 ; i<N; i++) {
        for (int j=i+1 ; j<N; j++) {
            s = s * S[j];
        }
        alpha += I[i] * s;
        s = 1;
    }

    int *linearArray = new int[totalValues];
    cout << "Size of Linear Array: " << totalValues << endl;

    int *baseAddress = &linearArray[0];
    const int size_dt = sizeof(linearArray[0]);

    int address = int(baseAddress) + (alpha * size_dt);
    cout << "Base Address: " << baseAddress << endl;
    cout << "Address of given indexes " << address << endl;

    delete S;
    S = nullptr;
    delete I;
    I = nullptr;
    delete linearArray;
    linearArray = nullptr;


    return 0;
}
```

`

3. COLUMN MAJOR

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

int main() {

    int N;
    cout << "Enter number of dimensions: ";
    cin >> N;
    int *S = new int[N];

    for (int i = 0; i < N; i++) {
        cout << "Enter size of Dimension " << i+1 << ": ";
        cin >> S[i];
    }

    int totalValues = 1;
    for (int i = 0; i < N; i++) {
        totalValues = totalValues * S[i];
    }

    int e;
    int* I = new int[N];
    for (int k = 0; k < N; k++) {
        cout << "Enter Index of Dimension: " << k+1 << endl;
        cin >> e;
        I[k] = e;
    }

    int s = 1;
    int alpha = 0; // require index

    for (int i=0 ; i<N; i++) {
```

`

```
    for (int i=0 ; i<N; i++) {
        for (int j=0 ; j<i; j++) {
            s = s * S[j];
        }
        alpha += I[i] * s;
        s = 1;
    }

    int *linearArray = new int[totalValues];
    cout << "Size of Linear Array: " << totalValues << endl;

    int *baseAddress = &linearArray[0];
    const int size_dt = sizeof(linearArray[0]);

    int address = int(baseAddress) + (alpha * size_dt);
    cout << "Base Address: " << baseAddress << endl;
    cout << "Address of given indexes " << address << endl;

    delete S;
    S = nullptr;
    delete I;
    I = nullptr;
    delete linearArray;
    linearArray = nullptr;


    return 0;
}
```

`

LINKED LIST

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

struct Node {
    int key;
    Node *next;
};

void insert(Node *hn, int value) {
    Node *newNode = (Node*) malloc(sizeof(Node));
    newNode->key = value;
    newNode->next = hn;
    hn = newNode;
}


void deleteNode(Node *hn, int value) {
    Node *current;
    Node *previous;

    while (current->next != NULL) {
        if (current->key==value) {
            previous->next = current->next;
            free(current);
            return;
        }

        previous = current;
        current = current->next;
    }
```

`

```
30          current = current->next;
31      }
32
33      cout << value << " not found." << endl;
34  }
35
36  void display(Node* hn) {
37      while (hn!=NULL) {
38          cout << hn->key << "->";
39          hn = hn->next;
40      }
41
42      cout << "NULL" << endl;
43  }
44
45  int main() {
46      int values[] = {4, 7, 1, 5, 9, 0, 1, 3, 77};
47      int len = sizeof(values)/sizeof(values[0]);
48      Node *HeadNode;
49
50      for (int i=0; i<len; i++) {
51          insert(HeadNode, values[i]);
52          display(HeadNode);
53      }
54
55      deleteNode(HeadNode, 9);
56      display(HeadNode);
57      deleteNode(HeadNode, 999);
58      display(HeadNode);
59
60      return 0;
61  }
```

`

STACK USING ARRAY

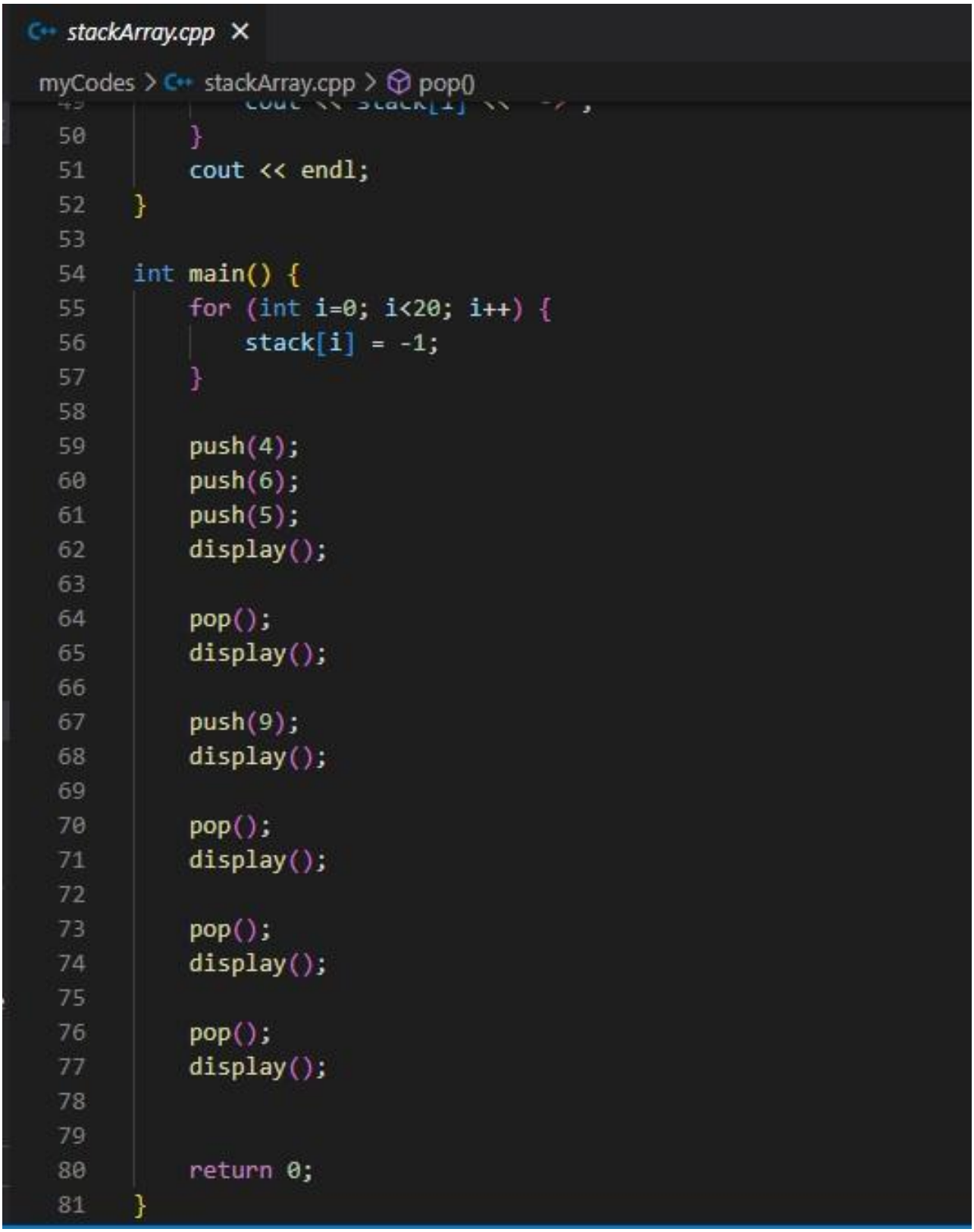
myCodes > stackArray.cpp > ...

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

int stack[20], top=0, numOfValues=0;

bool isEmpty() {
    if (numOfValues==0) {
        return true;
    }
    return false;
}

bool isFull() {
    if (numOfValues==20) {
        return true;
    }
    return false;
}

void push(int value) {
    if (isFull()) {
        cout << "Stack Overflow" << endl;
        return;
    }

    stack[top++] = value;
    numOfValues++;
}

int pop() {
    if (isEmpty()) {
        cout << "Stack Underflow" << endl;
```

`

```
    }

    stack[top++] = value;
    numOfValues++;
}

int pop() {
    if (isEmpty()) {
        cout << "Stack Underflow" << endl;
        return -1;
    }

    int value = stack[top];
    stack[top--] = -1;
    numOfValues--;
    return value;
}

int count() {
    return numOfValues;
}

void display() {
    cout << "Stack: ";
    for (int i=top-1; i>=0; i--) {
        cout << stack[i] << "->";
    }
    cout << endl;
}

int main() {
    for (int i=0; i<20; i++) {
        stack[i] = -1;
```

`


stackArray.cpp
myCodes > stackArray.cpp > pop()
50 }
51 cout << endl;
52 }
53
54 int main() {
55 for (int i=0; i<20; i++) {
56 stack[i] = -1;
57 }
58
59 push(4);
60 push(6);
61 push(5);
62 display();
63
64 pop();
65 display();
66
67 push(9);
68 display();
69
70 pop();
71 display();
72
73 pop();
74 display();
75
76 pop();
77 display();
78
79
80 return 0;
81 }

`

STACK USING LINKED LIST

myCodes > stackLL.cpp > ...

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

struct SNode {
    int data;
    SNode *next;
};

SNode *top=NULL;
int numOfValues = 0;

bool isEmpty() {
    if (top==NULL) {
        return true;
    }
    return false;
}

void push(int value) {
    SNode *newTop = new SNode();
    newTop->data = value;
    newTop->next = top;
    top = newTop;
    numOfValues++;
}

int pop() {
    if (isEmpty()) {
        cout << "Stack Underflow" << endl;
        return -1;
    }

```

`

```
}

int pop() {
    if (isEmpty()) {
        cout << "Stack Underflow" << endl;
        return -1;
    }

    SNode *temp = top;
    int value = temp->data;
    top = top->next;
    delete temp;
    numOfValues--;

    return value;
}

void display() {
    SNode *temp = top;
    cout << "Stack: ";
    while (temp!=NULL) {
        cout << temp->data << "->";
        temp = temp->next;
    }
    cout << endl;
}

int count() {
    return numOfValues;
}
```

`

```
56
57  int main() {
58
59      push(4);
60      push(6);
61      push(5);
62      display();
63
64      pop();
65      display();
66
67      push(9);
68      display();
69
70      pop();
71      display();
72
73      pop();
74      display();
75
76      pop();
77      display();
78
79      return 0;
80  }
```

`

QUEUE USING ARRAY

myCodes > C++ queueArray.cpp > top()

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

int queue[20]={0}, rear = -1, front = -1, capacity = 20, count = 0;

bool isFull() {
    if (capacity == count) {
        return true;
    }

    return false;
}

bool isEmpty() {
    if (front==-1 && rear==-1) {
        return true;
    }

    return false;
}

void enqueue(int value) {
    if (isFull()) {
        cout << "Queue Overflow" << endl;
    }

    else {
        if (front == - 1) front++;
        queue[rear++] = value;
        count++;
    }
}
```

`

```
34  int dequeue() {
35      int val;
36      if (front == - 1 || front > rear) {
37          cout << "Queue Underflow " << endl;
38      }
39
40      else {
41          val = queue[front];
42          queue[front] = 0;
43
44          if(front==rear) {
45              front = -1, rear = -1;
46          }
47          else {
48              front++;
49          }
50
51          count--;
52      }
53
54      return val;
55  }
56
57
58  int top() {
59      if (isEmpty()) {
60          cout << "Queue is empty." << endl;
61          return -99999;
62      }
63
64      else {
65          return queue[front];
```

`

```
    }

    else {
        return queue[front];
    }
}

void display() {
    if (front == - 1) {
        cout << "Queue is empty." << endl;
    }

    else {
        cout << "Queue elements are: ";
        for (int i=front; i<=rear; i++) {
            cout << queue[i] << ", ";
        }
        cout << endl;
    }
}

int main() {

    int choice;
    cout << "1) Insert an element" << endl;
    cout << "2) Delete first element" << endl;
    cout << "3) Display all the elements" << endl;
    cout << "4) Get top element" << endl;
    cout << "5) Exit" << endl;

    do {
        cout << "Enter your choice: ";
```

`

```
91
92      do {
93          cout << "Enter your choice: ";
94          cin >> choice;
95
96          switch (choice) {
97              case 1:
98                  int value;
99                  cout << "Enter value: " << endl;
100                 cin >> value;
101                 enqueue(value);
102                 break;
103             case 2:
104                 dequeue();
105                 break;
106             case 3:
107                 display();
108                 break;
109             case 4:
110                 cout << "Top value: " << top() << endl;
111                 break;
112             case 5:
113                 cout << "Exit" << endl;
114                 break;
115             default:
116                 cout << "Invalid choice" << endl;
117         }
118     }
119     while(choice!=5);
120
121     return 0;
122 }
```

`

QUEUE USING LINKED LIST

myCodes > C++ queueLL.cpp > ...

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


struct QNode {
    int data;
    QNode* next;
    QNode(int d) {
        data = d;
        next = NULL;
    }
};

QNode *front=NULL, *rear=NULL;

void enqueue(int value) {
    QNode *newNode = new QNode(value);
    if (rear==NULL) {
        front = newNode;
        rear = newNode;
    }

    rear->next = newNode;
    rear = newNode;
}

int dequeue() {
    if (front==NULL) {
        return 0;
    }

    QNode *temp = front;
```

`

myCodes > C++ queueLL.cpp > ...

```
26
27  int dequeue() {
28      if (front==NULL) {
29          return 0;
30      }
31
32      QNode *temp = front;
33
34      front = front->next;
35      if (front==NULL) {
36          rear = NULL;
37      }
38
39      int value = temp->data;
40      delete temp;
41
42      return value;
43  }
44
45  void display() {
46      QNode *temp = front;
47      while (temp!=NULL) {
48          cout << temp->data << "->";
49          temp = temp->next;
50      }
51      cout << "NULL" << endl;
52  }
53
54
55  int main() {
56
57      enqueue(10);
```

`

```
54
55  int main() {
56
57      enqueue(10);
58      enqueue(20);
59      display();
60
61      dequeue();
62      display();
63
64      dequeue();
65      display();
66
67      enqueue(30);
68      enqueue(40);
69      enqueue(50);
70      display();
71
72      dequeue();
73      display();
74
75      return 0;
76  }
```

`

SEARCHING ALGORITHMS

(LINEAR & BINARY SEARCH)

myCodes > searchingAndSorting > C++ search.cpp > main()

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

int liearSearch(int values[], int size, int value) {

    for (int i=0; i<size; i++) {
        if (values[i]==value) {
            return i;
        }
    }
    return -1;
}

int binarySearch(int values[], int size, int value) {

    int low=0, high=size-1;

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

        if (values[mid] == value)
            return mid;

        if (values[mid] < value)
        low = mid + 1;

        else
        high = mid - 1;
    }

    return -1;
}
```

`

SORTING ALGORITHMS

1. MERGE SORT

myCodes > searchingAndSorting > C++ mergeSort.cpp > merge(int [], int, int, int)

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


void merge(int arr[], int lower, int mid, int upper) {

    int n1 = mid - lower + 1;
    int n2 = upper - mid;

    int L[n1], M[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[lower + i];

    for (int j = 0; j < n2; j++)
        M[j] = arr[mid + 1 + j];

    int i, j, k;
    i = 0;
    j = 0;
    k = lower;

    while (i < n1 && j < n2) {
        if (L[i] <= M[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = M[j];
            j++;
        }
        k++;
```

`

```
        }
        k++;
    }

    while (i < n1) {
        arr[k++] = L[i++];
    }

    while (j < n2) {
        arr[k++] = M[j++];
    }
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {

        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}

> void printArray(int arr[], int size) {···
```

`

2. QUICK SORT

myCodes > searchingAndSorting > quickSort.cpp > main()

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

int partition(int arr[], int start, int end) {

    int pivot = arr[start];

    int count = 0;
    for (int i = start + 1; i <= end; i++) {
        if (arr[i] <= pivot)
            count++;
    }

    int pivotIndex = start + count;
    swap(arr[pivotIndex], arr[start]);

    int i = start, j = end;

    while (i < pivotIndex && j > pivotIndex) {

        while (arr[i] <= pivot) i++;

        while (arr[j] > pivot) j--;

        if (i < pivotIndex && j > pivotIndex) {
            swap(arr[i++], arr[j--]);
        }
    }

    return pivotIndex;
}
```

`

```
32
33  void quickSort(int arr[], int start, int end) {
34
35      if (start<end) {
36          int p = partition(arr, start, end);
37          quickSort(arr, start, p - 1);
38          quickSort(arr, p + 1, end);
39      }
40  }
41
42  void printArray(int arr[], int size) {
43      for (int i = 0; i < size; i++) {
44          cout << arr[i] << " ";
45      }
46      cout << endl;
47  }
48
49  int main() {
50      cout << "Quick Sort" << endl;
51      int values[] = {23, 1, 7, 3, 12, 5, 22, 11, 9};
52      int size = sizeof(values) / sizeof(values[0]);
53      cout << "Before Sort: "; printArray(values, size);
54
55      quickSort(values, 0, size - 1);
56
57      cout << "After Sort: "; printArray(values, size);
58
59      return 0;
60  }
61
```

`

3. BUBBLE SORT

myCodes > searchingAndSorting > bubbleSort.cpp > main()

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

void bubbleSort(int arr[], int size) {

    int i, j;
    for (i=0; i < size-1; i++) {
        for (j=0; j < size-i-1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    cout << "Bubble Sort" << endl;
    int values[] = {23, 1, 7, 3, 12, 5, 22, 11, 9};
    int size = sizeof(values) / sizeof(values[0]);
    cout << "Before Sort: "; printArray(values, size);

    bubbleSort(values, size);

    cout << "After Sort: "; printArray(values, size);

```

`

4. INSERTION SORT

myCodes > searchingAndSorting > insertionSort.cpp > ...

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

void insertionSort(int arr[], int size) {
    int temp, j;
    for (int i=1; i < size; i++) {
        temp = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > temp) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = temp;
    }
}

void printArray(int arr[], int size) {...

int main() {
    cout << "Insertion Sort" << endl;
    int values[] = {23, 1, 7, 3, 12, 5, 22, 11, 9};
    int size = sizeof(values) / sizeof(values[0]);
    cout << "Before Sort: "; printArray(values, size);

    insertionSort(values, size);

    cout << "After Sort: "; printArray(values, size);

    return 0;
}
```

`

5. SELECTION SORT

myCodes > searchingAndSorting > selectionSort.cpp > main()

```cpp
void selectiontionSort(int arr[], int size) {

    for (int i=0; i < size-1; i++) {
        int minimumIndex = i;
        for (int j = i+1; j < size; j++) {
            if (arr[j] < arr[minimumIndex])
                minimumIndex = j;
        }

        if (minimumIndex!=i)
            swap(arr[minimumIndex], arr[i]);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    cout << "Selection Sort" << endl;
    int values[] = {23, 1, 7, 3, 12, 5, 22, 11, 9};
    int size = sizeof(values) / sizeof(values[0]);
    cout << "Before Sort: "; printArray(values, size);

    selectiontionSort(values, size);

    cout << "After Sort: "; printArray(values, size);

    return 0;
```

`

EXPRESSION PARSING

myCodes > expression_parsing > C++ expParsing.cpp > ...

```cpp
#include <iostream>
#include "bits/stdc++.h"
#include <string.h>
#include <stack>
#include <cmath>
#include <map>
using namespace std;


int isOperator(char op) {
    if (op=='(') return 1;

    else if (op==')') return -1;

    else if (op=='+' || op=='-') return 2;

    else if (op=='/' || op=='*') return 3;

    else if (op=='^') return 4;

    else return 0;
}


double operation(double a, double b, char op) {
    if (op=='+') return a+b;

    else if (op=='-') return a-b;

    else if (op=='*') return a*b;

    else if (op=='/') return a/b;
```

`

```
31
32      else if (op=='/') return a/b;
33
34      else return pow(a, b);
35  }
36
37
38  char* InfixToPostfix(char* infix) {
39      char *postfix;
40      postfix = (char*) malloc(100 * sizeof(char));
41      stack<char> s;
42      int len = strlen(infix);
43
44      int j = 0;
45
46      for (int i=0; i<len; i++) {
47
48          char current = infix[i];
49          int op = isOperator(current);
50
51          if (op==-1) {
52              while (s.top()!='(') {
53                  postfix[j++] = s.top();
54                  s.pop();
55              }
56              s.pop();
57              continue;
58          }
59
60          if (op==0) {
61              postfix[j++] = current;
62          }
63
```

`

```
63
64          else {
65
66              if ((s.empty()) || op==1)   s.push(current);
67
68              else {
69                  while (op <= isOperator(s.top())) {
70                      postfix[j++] = s.top();
71                      s.pop();
72                      if (s.empty()) break;
73                  }
74                  s.push(current);
75              }
76          }
77      }
78
79      while (!s.empty()) {
80          postfix[j++] = s.top();
81          s.pop();
82      }
83      postfix[j] = '\0';
84
85      return postfix;
86  }
87
88
89  double postfixEvaluation(char* postfix, map<char, double> &values) {
90      stack<char> solution;
91      double ans;
92      int length = strlen(postfix);
93      for (int k=0; k<length; k++) {
94          char current = postfix[k];
```

`

```
    double ans;
    int length = strlen(postfix);
    for (int k=0; k<length; k++) {
        char current = postfix[k];

        if (isOperator(current)==0) {
            solution.push(current);
        }

        else {
            double b = values[solution.top()];
            solution.pop();
            double a = values[solution.top()];
            solution.pop();

            ans = operation(a, b, current);
            // cout << a << current << b << " = " << ans << endl;
            values[char(ans)] = ans;
            solution.push(char(ans));
        }
    }
    solution.pop();

    return ans;
}


int main() {

    char infix[100] = "A+B*C/(E-F)*(A^(B-C/D))";
    // cout << "Enter an expression \n";
    // char infix[100];
```

`

```
int main() {

    char infix[100] = "A+B*C/(E-F)*(A^(B-C/D))";
    // cout << "Enter an expression \n";
    // char infix[100];
    // cin >> infix;

    int len = strlen(infix);
    map<char, double> values;

    for (int z=0; z<len; z++) {
        char c = infix[z];
        double a;
        if (isOperator(c)==0 && values.count(c)==0) {
            cout << "Enter value of " << c << ": ";
            cin >> a;
            values[c] = a;
        }
    }

    cout << endl << "Infix: " << infix << endl << endl;

    char* postfix = InfixToPostfix(infix);
    cout << "Postfix: " << postfix << endl << endl;

    double ans = postfixEvaluation(postfix, values);
    cout << "Solution: "<< ans << endl;
}
```

`

TREES

1. EXPRESSION TREE

myCodes > trees > C++ expTree.cpp > ...

```
1   #include <iostream>
2   #include "infToPos.h"
3   using namespace std;
4
5
6   struct Node {
7       char data;
8       Node *left;
9       Node *right;
10  };
11
12
13  Node* createExpressionTree(char *postfix) {
14
15      stack<struct Node*> s;
16      struct Node *n, *l, *r;
17
18      int len = strlen(postfix);
19      for (int i=0; i<len; i++) {
20
21          char current = postfix[i];
22          n = new Node;
23          n->data = current;
24          n->left = NULL;
25          n->right = NULL;
26
27
28          if (isOperator(current)!=0) {
29              r = s.top();
30              s.pop();
31              l = s.top();
32              s.pop();
```

`

```
            s.pop();

            n->right = r;
            n->left = l;
        }

        s.push(n);
    }

    struct Node *ETRoot = new Node;
    ETRoot = s.top();
    s.pop();

    return ETRoot;
}


void traverseInOrder(struct Node *temp) {
    if (temp!=NULL) {
        traverseInOrder(temp->left);
        cout << temp->data;
        traverseInOrder(temp->right);
    }
}


void traversePreOrder(struct Node *temp) {
    if (temp!=NULL) {
        cout << temp->data;
        traversePreOrder(temp->left);
        traversePreOrder(temp->right);
    }
}
```

`

```
void traversePostOrder(struct Node *temp) {
    if (temp!=NULL) {
        traversePostOrder(temp->left);
        traversePostOrder(temp->right);
        cout << temp->data;
    }
}


int main() {

    // char infix[100] = "a+b";
    char infix[100] = "A+B*C/(E-F)*(A^(B-C/D))";
    // inputExpression(infix);
    cout << infix << endl;

    char postfix[100];
    infixToPostfix(infix, postfix);
    cout << postfix << "\n\n" << endl;


    struct Node *ETRoot = new Node;
    ETRoot = createExpressionTree(postfix);

    cout << "Pre-Order: ";  traversePreOrder(ETRoot); cout << endl;
    cout << "Post-Order: "; traversePostOrder(ETRoot); cout << endl;
    cout << "In-Order: ";   traverseInOrder(ETRoot); cout << endl;

    return 0;
}
```

`

2. BINARY SEARCH TREE

myCodes > trees > bst.cpp > deleteNode(Node *, int)

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

struct Node {
    int key;
    Node *left, *right;
};


Node* newNode(int item) {
    Node *temp = (Node*)malloc(sizeof(Node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Inorder Traversal
void inorder(Node *root) {
  if (root != NULL) {
    inorder(root->left);
    cout << root->key << " -> ";
    inorder(root->right);
  }
}

// Preorder Traversal
void preorder(Node *root) {
  if (root != NULL) {
    cout << root->key << " -> ";
    inorder(root->left);
    inorder(root->right);
  }
```

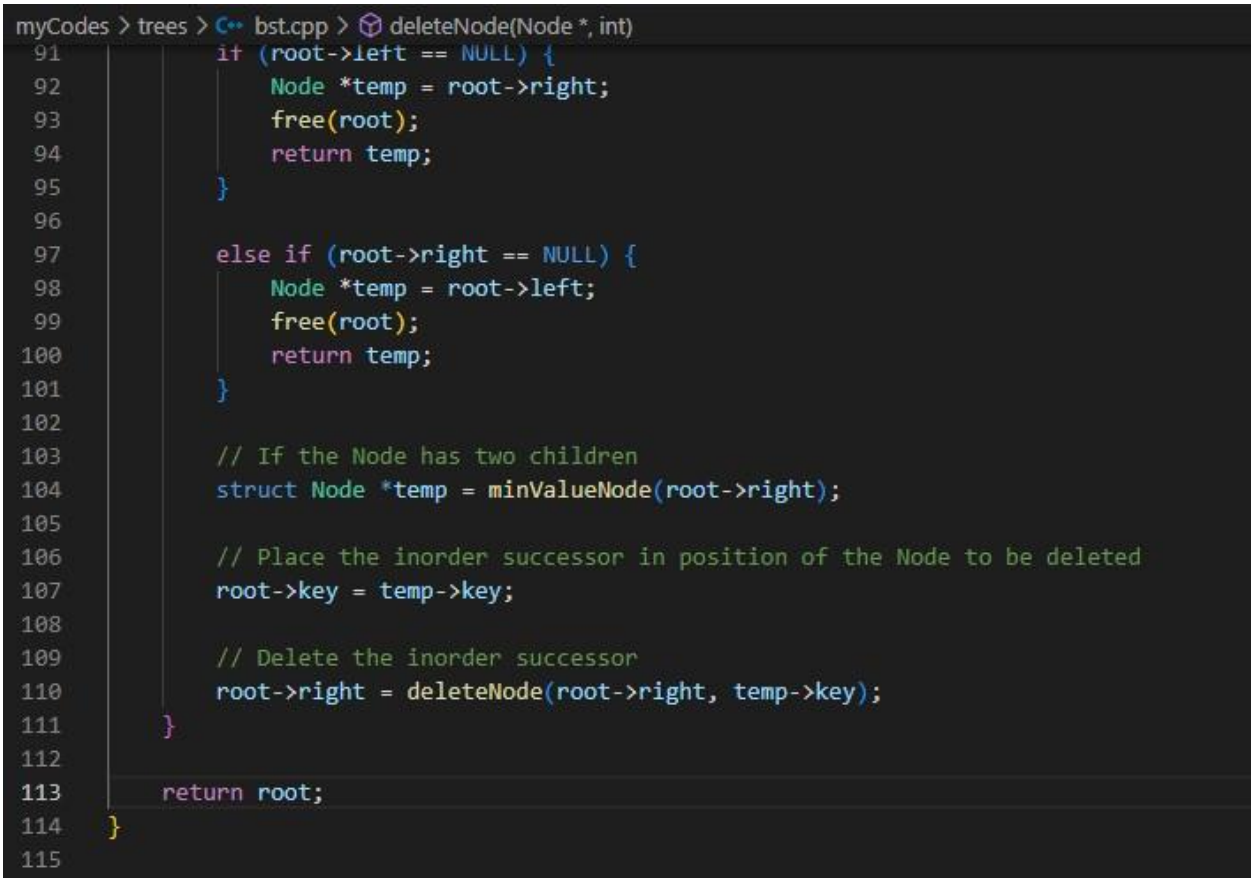
`

```
30      inorder(root->left);
31      inorder(root->right);
32    }
33  }
34
35  // Postorder Traversal
36  void postorder(Node *root) {
37    if (root != NULL) {
38      inorder(root->left);
39      inorder(root->right);
40      cout << root->key << " -> ";
41    }
42  }
43
44
45  Node* insert(Node *Node, int value) {
46
47      if (Node==NULL) return newNode(value);
48
49      if (value < Node->key)
50          Node->left = insert(Node->left, value);
51      else
52          Node->right = insert(Node->right, value);
53
54      return Node;
55  }
56
57
58  Node* minValueNode(Node *HeadNode) {
59      Node *current = HeadNode;
60
61      while (current && current->left != NULL)
62          current = current->left;
```

`

```
61      while (current && current->left != NULL)
62          current = current->left;
63
64      return current;
65  }
66
67  Node* maxValueNode(Node *HeadNode) {
68      Node *current = HeadNode;
69
70      while (current && current->right != NULL)
71          current = current->right;
72
73      return current;
74  }
75
76  Node* deleteNode(Node *root, int key) {
77      // Return if the tree is empty
78      if (root == NULL) return root;
79
80      // Find the Node to be deleted
81      if (key < root->key) {
82          root->left = deleteNode(root->left, key);
83      }
84
85      else if (key > root->key) {
86          root->right = deleteNode(root->right, key);
87      }
88
89      else {  // key == root->key
90          // If the Node is with only one child or no child
91          if (root->left == NULL) {
92              Node *temp = root->right;
```

`


myCodes > trees > bst.cpp > deleteNode(Node *, int)
91 if (root->left == NULL) {
92 Node *temp = root->right;
93 free(root);
94 return temp;
95 }
96
97 else if (root->right == NULL) {
98 Node *temp = root->left;
99 free(root);
100 return temp;
101 }
102
103 // If the Node has two children
104 struct Node *temp = minValueNode(root->right);
105
106 // Place the inorder successor in position of the Node to be deleted
107 root->key = temp->key;
108
109 // Delete the inorder successor
110 root->right = deleteNode(root->right, temp->key);
111 }
112
113 return root;
114 }
115

`

```
int main() {
    struct Node *root = NULL;
    root = insert(root, 8);
    root = insert(root, 3);
    root = insert(root, 1);
    root = insert(root, 6);
    root = insert(root, 7);
    root = insert(root, 10);
    root = insert(root, 14);
    root = insert(root, 4);

    cout << "Preorder traversal: ";
    preorder(root);
    cout << endl;

    cout << "Inorder traversal: ";
    inorder(root);
    cout << endl;

    cout << "Postorder traversal: ";
    postorder(root);
    cout << endl;

    root = deleteNode(root, 3);
    cout << endl << "After deleting 3" << endl;
    cout << "Inorder traversal: ";
    inorder(root);
    cout << endl;
}

```

`

3. BINARY HEAP / PRIORITY QUEUE

myCodes > trees > PQ.cpp > BinaryHeap

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

class BinaryHeap {
    private:
        int* PQ;
        int count;
        int capacity;
        int n;

    public:
        BinaryHeap() {
            capacity = 5;
            PQ = (int*) malloc(capacity*sizeof(int));
            count = 0;
            n = 1;
        }

        void display(){
            cout << "count = " << count << ": ";
            for (int i=1; i<count+1; i++) {
                cout << PQ[i] << " ";
            }
            cout << endl;
        }

        void heapify() {

            int i = count;

            while (i!=1) {
                int k = int(i/2);
```

`

```
30
31          while (i!=1) {
32              int k = int(i/2);
33              if (PQ[i]<PQ[k]) {
34
35                  int temp = PQ[k];
36                  PQ[k] = PQ[i];
37                  PQ[i] = temp;
38                  i = k;
39              }
40
41              else break;
42          }
43      }
44
45      void enqueue(int data) {
46          if (count == capacity) {
47
48              capacity *= 2;
49              int *temp = (int*) malloc(capacity*sizeof(int));
50
51              for (int i=1; i<count+1; i++) {
52                  temp[i] = PQ[i];
53              }
54
55              free(PQ);
56              PQ = temp;
57          }
58
59          PQ[++count] = data;
60          heapify();
61      }
```

`

```
55                  free(PQ);
56                  PQ = temp;
57              }
58
59              PQ[++count] = data;
60              heapify();
61          }
62
63          void dequeue() {
64              if ((2*n <= count) && ((2*n)+1 <= count)) {
65
66                  if (PQ[2*n] < PQ[(2*n)+1]) {
67                      PQ[n] = PQ[2*n];
68                      n *= 2;
69                  }
70
71                  else {
72                      PQ[n] = PQ[(2*n)+1];
73                      n = (2*n)+1;
74                  }
75
76                  dequeue();
77              }
78
79              else {
80                  PQ[n] = PQ[count];
81                  count--;
82              }
83              n = 1;
84          }
85  };
```

`

```
int main() {
    BinaryHeap PriorityQueue;

    PriorityQueue.enqueue(8);
    PriorityQueue.enqueue(13);
    PriorityQueue.enqueue(5);
    PriorityQueue.enqueue(10);
    PriorityQueue.enqueue(3);
    PriorityQueue.enqueue(9);
    PriorityQueue.enqueue(2);
    PriorityQueue.enqueue(-1);
    PriorityQueue.enqueue(6);
    PriorityQueue.enqueue(4);

    PriorityQueue.display();

    PriorityQueue.dequeue();
    PriorityQueue.display();

    PriorityQueue.dequeue();
    PriorityQueue.display();

    PriorityQueue.dequeue();
    PriorityQueue.display();
    return 0;
}

```

`

HASHING

1. OPEN HASHING

myCodes > hashing > openHash.cpp > ...

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

bool isPrime(int n) {
    // Corner cases
    if (n <= 1)  return false;
    if (n <= 3)  return true;

    // This is checked so that we can skip
    // middle five numbers in below loop
    if (n%2 == 0 || n%3 == 0) return false;

    for (int i=5; i*i<=n; i=i+6)
        if (n%i == 0 || n%(i+2) == 0)
            return false;

    return true;
}

int nextPrime(int N) {

    // Base case
    if (N <= 1)
        return 2;

    int prime = N;
    bool found = false;

    // Loop continuously until isPrime returns
    // true for a number greater than n
    while (!found) {
        prime++;
```

`

```
31      while (!found) {
32          prime++;
33
34          if (isPrime(prime))
35              found = true;
36      }
37
38      return prime;
39  }
40
41
42  struct HNode {
43      int key;
44      HNode* next;
45
46      HNode(int x) {
47          this->key = x;
48          next = NULL;
49      }
50  };
51
52
53  HNode** openHashing(int &HS, int *values, int totalValues) {
54
55      HNode **HT = (HNode**) malloc(HS * sizeof(HNode*));
56      int numOfValues = 0;
57      float load = 0;
58
59      for (int i=0; i<HS; i++) {
60          HT[i] = NULL;
61      }
62
```

`

```
63      int HI;
64      for (int i=0; i<totalValues; i++) {
65
66          if (load >= 1) {
67
68              int tempValues[numOfValues];
69              for (int j=0; j<numOfValues; j++) {
70                  tempValues[j] = values[j];
71              }
72
73              free(HT);
74              HS = nextPrime(HS * 2);
75              HT = openHashing(HS, tempValues, numOfValues);
76          }
77
78          int key = values[i];
79          HNode *hn = new HNode(key);
80          HI = key % HS;
81
82          if (HT[HI]==NULL) {
83              HT[HI] = hn;
84          }
85
86          else {
87              HNode *temp = HT[HI];
88              HT[HI] = hn;
89              HT[HI]->next = temp;
90          }
91
92          numOfValues++;
93          load = float(numOfValues)/float(HS);
```

`

```
92          numOfValues++;
93          load = float(numOfValues)/float(HS);
94
95      }
96
97      return HT;
98  }
99
100 void printHash(HNode** array, int len) {
101     for (int i=0; i<len; i++) {
102         cout << "Index " << i << ": ";
103         if (array[i]==NULL) {
104             cout << "NULL" << endl;
105         }
106
107         else {
108             HNode *temp = array[i];
109             while (temp!=NULL) {
110                 cout << temp->key << "->";
111                 temp = temp->next;
112             }
113             cout << endl;
114         }
115     }
116 }
117
118 void printList(int* list, int len) {
119     cout << "[";
120     for (int i=0; i<len; i++) {
121         cout << list[i] << ", ";
122     }
```

`

```
    }
    cout << "]" << endl;
}

bool exists(HNode **HT, int HS, int val) {

    int HI = val % HS;

    HNode *temp = HT[HI];
    while (temp!=NULL) {
        if (temp->key==val) return true;
        temp = temp->next;
    }

    return false;
}

int main() {

    int sizeOfArray = 10;
    int values[sizeOfArray] = {5, 13, 2, 44, 100, 15, 33, 66, 88, 99};
    cout << "Array: "; printList(values, sizeOfArray);

    int hs = 5;
    struct HNode **ht = openHashing(hs, values, sizeOfArray);

    cout << endl << "HS: " << hs << endl; printHash(ht, hs);

    cout << endl << "searching" << endl;
    cout << exists(ht, hs, 44) << endl;

    return 1;
```

`

2. CLOSED HASING

myCodes > hashing > C++ closedHash.cpp > ...

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

bool isPrime(int n) {
    // Corner cases
    if (n <= 1)  return false;
    if (n <= 3)  return true;

    // This is checked so that we can skip
    // middle five numbers in below loop
    if (n%2 == 0 || n%3 == 0) return false;

    for (int i=5; i*i<=n; i=i+6)
        if (n%i == 0 || n%(i+2) == 0)
            return false;

    return true;
}

int nextPrime(int N) {

    // Base case
    if (N <= 1)
        return 2;

    int prime = N;
    bool found = false;

    // Loop continuously until isPrime returns
    // true for a number greater than n
    while (!found) {
```

`

```
30      // Loop continuously until isPrime returns
31      // true for a number greater than n
32      while (!found) {
33          prime++;
34
35          if (isPrime(prime))
36              found = true;
37      }
38
39      return prime;
40  }
41
42
43  void printList(int* list, int len) {
44      cout << "[";
45      for (int i=0; i<len; i++) {
46          cout << list[i] << ", ";
47      }
48      cout << "]" << endl;
49  }
50
51
52
53  int* closedHashing(int &HS, int *values, int totalValues) {
54
55      int *HT = (int*) malloc(HS * sizeof(int));
56
57      for (int i=0; i<HS; i++) {
58          HT[i] = -1;
59      }
60
61      int no_of_values = 0;
```

`

```
61      int no_of_values = 0;
62      float load = 0;
63
64      int HI, key;
65      for (int i=0; i<totalValues; i++) {
66
67          if (load >= 0.5) {
68              // rehashing
69              int val[no_of_values];
70
71              for (int k = 0; k < no_of_values; k++)
72              {
73                  val[k] = values[k];
74              }
75
76              free(HT);
77              HS = nextPrime(HS * 2);
78              HT = closedHashing(HS, val, no_of_values);
79          }
80
81          key = values[i];
82          HI = key % HS;
83
84          if (HT[HI] == -1) {
85              HT[HI] = key;
86          }
87
88          else {
89              // linear probing
90              int j = HI;
```

`

```
88          else {
89              // linear probing
90              int j = HI;
91              while (HT[j] != -1) {
92                  j = (j+1) % HS;
93              }
94
95              // quadratic probing
96              // int x = 1;
97              // while (HT[j] != -1) {
98              //     j = (HI + (x*x)) % HS;
99              //     x++;
100             // }
101
102             HT[j] = key;
103         }
104
105         no_of_values++;
106         load = float(no_of_values)/float(HS);
107     }
108
109     return HT;
110 }
111
112
113 bool exists(int* HT, int HS, int val) {
114     int HI = val % HS;
115
116     // linear probing
117     while (HT[HI]!=-1) {
118         if (HT[HI]==val) return true;
119         HI = (HI++) % HS;
```

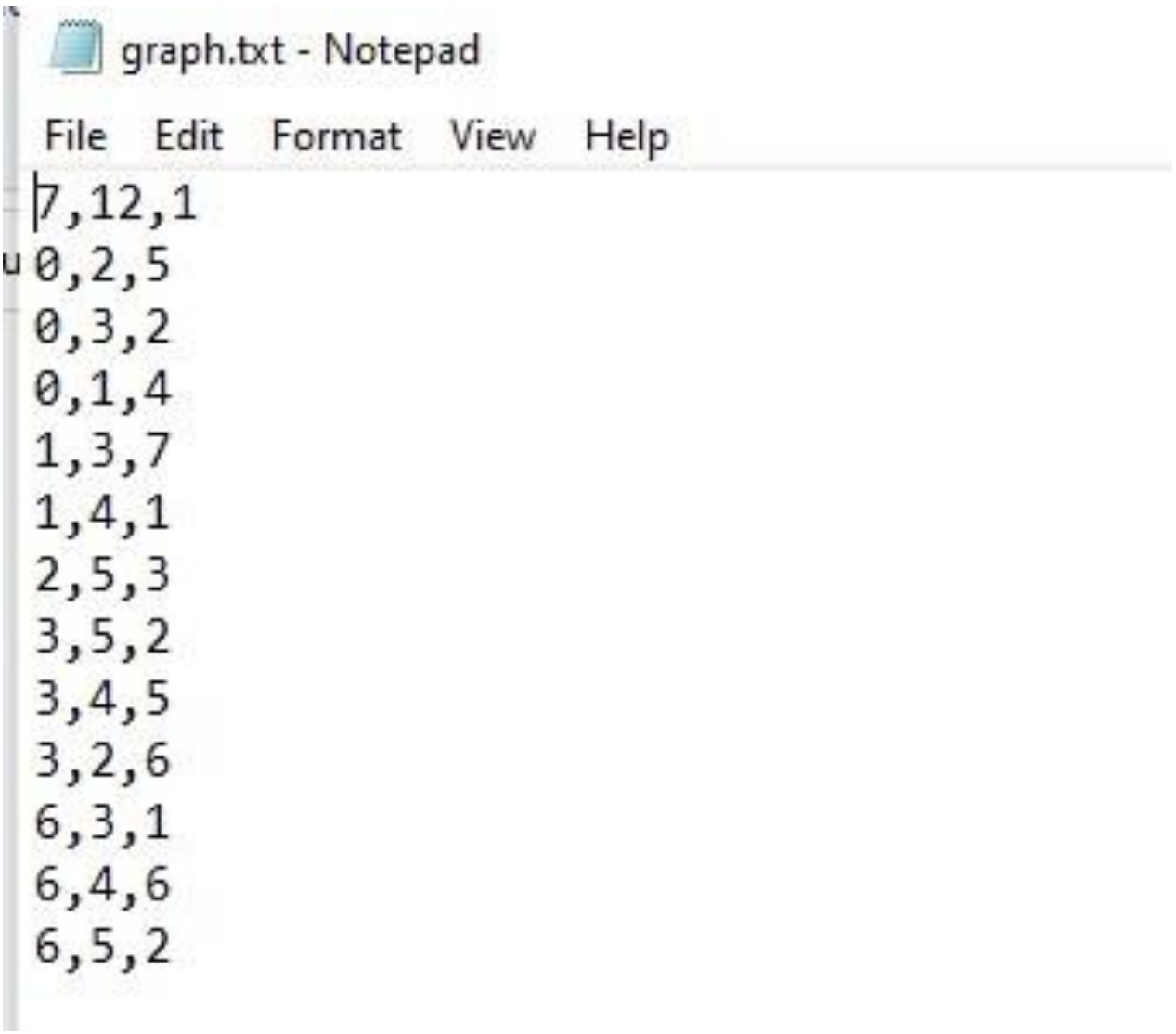
`

```
118         if (HT[HI]==val) return true;
119         HI = (HI++) % HS;
120     }
121
122     // quadratic probing
123     // int x = 1, j = HI;
124     // while (HT[j] != -1) {
125     //     if (HT[j]==val) return true;
126     //     j = (HI + (x*x)) % HS;
127     //     x++;
128     // }
129
130     return false;
131 }
132
133 int main() {
134     int sizeOfArray = 10;
135     int values[sizeOfArray] = {5, 13, 2, 44, 100, 15, 33, 66, 88, 99};
136     cout << "Array: "; printList(values, sizeOfArray);
137
138     int hs = 10, *ht;
139
140     ht = closedHashing(hs, values, sizeOfArray);
141
142     cout << endl << "Size of Linear Array: " << hs << endl;
143     cout << "Linear Probation: "; printList(ht, hs);
144
145     cout << "Exists" << endl;
146     cout << exists(ht, hs, 55555) << endl;
147     cout << exists(ht, hs, 13) << endl;
148
149     return 0;
```

`

GRAPHS

TEXT FILE



`

1. IN AND OUT DEGREE

i. ADJACENY LIST FUNCTIONS

myCodes > graphs > h adjListFunctions.h > AdjNode > vertex

```
1   #include <iostream>
2   #include <fstream>
3   #include <sstream>
4   #include <vector>
5   #include <map>
6
7   using namespace std;
8
9   // --------ADJACENCY LIST----------
10
11
12  struct AdjNode {
13      int vertex, weight;
14      AdjNode *next;
15      AdjNode(int ver, int w) {
16          this->vertex = ver;
17          this->weight = w;
18          this->next = NULL;
19      }
20  };
21
22  struct AdjList {
23      AdjNode **List;
24      int vertices, edges, direction;
25  };
26
27  void addNode(AdjNode **List, int v1, int v2, int weight, int direction) {
28      AdjNode *node = new AdjNode(v2, weight);
29
30      if(List[v1]==NULL) {
31          List[v1] = node;
32      }
```

`

```
30      if(List[v1]==NULL) {
31          List[v1] = node;
32      }
33      else {
34          AdjNode *temp = List[v1];
35          List[v1] = node;
36          node->next = temp;
37      }
38
39      if (direction==0) {
40          node = new AdjNode(v1, weight);
41          if(List[v2]==NULL) {
42              List[v2] = node;
43          }
44          else {
45              AdjNode *temp = List[v2];
46              List[v2] = node;
47              node->next = temp;
48          }
49      }
50  }
51
52  AdjList adjacencyList(string filename) {
53
54      int vertices, edges, direction;
55
56      ifstream file;
57      file.open(filename);
58
59      string graphInfo;
60      getline(file, graphInfo);
61      stringstream ved(graphInfo);
```

`

```
61      stringstream ved(graphInfo);
62
63      string n;
64      getline(ved, n, ',');
65      vertices = stoi(n);
66
67      getline(ved, n, ',');
68      edges = stoi(n);
69
70      getline(ved, n, ',');
71      direction = stoi(n);
72
73      AdjList info;
74      info.vertices = vertices;
75      info.edges = edges;
76      info.direction = direction;
77
78      AdjNode **List = (AdjNode**) malloc(vertices*sizeof(AdjNode*));
79
80      for (int i=0; i<vertices; i++)
81          List[i] = NULL;
82
83
84      for (int i=0; i<edges; i++) {
85          int v1, v2, weight;
86          getline(file, graphInfo);
87          stringstream data(graphInfo);
88          string value;
89
90          getline(data, value, ',');
91          v1 = stoi(value);
92
```

`

```
90          getline(data, value, ',');
91          v1 = stoi(value);
92
93          getline(data, value, ',');
94          v2 = stoi(value);
95
96          getline(data, value, ',');
97          weight = stoi(value);
98
99          addNode(List, v1, v2, weight, direction);
100     }
101
102     file.close();
103     info.List = List;
104
105     return info;
106 }
107
108 int outDegree(AdjNode **List, int vertex) {
109     int degree = 0;
110     AdjNode *temp = List[vertex];
111     while (temp!=NULL) {
112         degree++;
113         temp = temp->next;
114     }
115
116     return degree;
117 }
118
119 int inDegree(AdjNode **List, int vertex, int vertices) {
120     int degree = 0;
121     for (int i=0; i<vertices; i++) {
```

`

```
118
119  int inDegree(AdjNode **List, int vertex, int vertices) {
120      int degree = 0;
121      for (int i=0; i<vertices; i++) {
122          AdjNode *temp = List[i];
123          while (temp!=NULL) {
124              if (temp->vertex==vertex)
125                  degree++;
126
127              temp = temp->next;
128          }
129      }
130
131      return degree;
132  }
133
134  void printList(AdjNode **List, int vertices) {
135      cout << "Adjacency List:" << endl << endl;
136      for (int i=0; i<vertices; i++) {
137          cout << "Vertex " << i << ": ";
138          AdjNode *temp = List[i];
139          while (temp!=NULL) {
140              cout << temp->vertex << "->";
141              temp = temp->next;
142          }
143
144          cout << endl;
145      }
146      cout << endl;
147  }
148
```

`

ii. ADJACENCY MATRIX

myCodes > graphs > h adjMatFunctions.h > ...

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <map>

using namespace std;


// -------ADJACENCY MATRIX--------

int** squareMatrix(int n) {
    int **matrix = new int*[n];
    for (int i=0; i<n; i++) {
        matrix[i] = new int[n];
    }

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

    return matrix;
}

void printMatrix(int **matrix, int rows, int columns) {
    cout << "Adjacency Matrix" << endl << endl;

    cout << "  \t";
    for (int k=0; k<columns; k++) {
        cout << k << "\t";
```

`

```
        cout << k << "\t";
    }
    cout << endl << endl;

    for (int i=0; i<rows; i++) {
        cout << i << ":\t";
        for (int j = 0; j < columns; j++) {
            cout << matrix[i][j] << "\t";
        }
        cout << endl;
    }
    cout << endl;
}

int inDegree(int **Matrix, int vertices, int vertex) {

    int degree = 0;
    for (int i=0; i<vertices; i++) {
        if (Matrix[i][vertex] != 0) {
            degree += 1;
        }
    }

    return degree;
}

int outDegree(int **Matrix, int vertices, int vertex) {
    int degree = 0;
    for (int i=0; i<vertices; i++) {
        if (Matrix[vertex][i] != 0) {
            degree += 1;
        }
```

`

```
63          }
64      }
65
66      return degree;
67  }
68
69  struct AdjMatrix {
70      int **Matrix;
71      int vertices, edges, direction;
72  };
73
74  AdjMatrix adjacencyMatrix(string filename) {
75      int vertices, edges, direction;
76
77      ifstream file;
78      file.open(filename);
79
80      string graphInfo;
81      getline(file, graphInfo);
82      stringstream ved(graphInfo);
83
84      string n;
85      getline(ved, n, ',');
86      vertices = stoi(n);
87
88      getline(ved, n, ',');
89      edges = stoi(n);
90
91      getline(ved, n, ',');
92      direction = stoi(n);
93
94      AdjMatrix info;
```

`

```
    info.vertices = vertices;
    info.edges = edges;
    info.direction = direction;

    int **Matrix = squareMatrix(vertices);

    for (int i=0; i<edges; i++) {
        int v1, v2, weight;
        getline(file, graphInfo);
        stringstream data(graphInfo);
        string value;

        getline(data, value, ',');
        v1 = stoi(value);

        getline(data, value, ',');
        v2 = stoi(value);

        getline(data, value, ',');
        weight = stoi(value);

        Matrix[v1][v2] = weight;
        if (direction==0) {
            Matrix[v2][v1] = weight;
        }
    }

    file.close();
    info.Matrix = Matrix;

    return info;
}
```

`

iii. IN AND OUT DEGREE DRIVER

myCodes > graphs > C++ inOutDegree.cpp > main()

```cpp
#include "adjMatFunctions.h"
#include "adjListFunctions.h"


int main() {
    // ----------MATRIX------------
    int vertices, edges, direction;

    AdjMatrix info = adjacencyMatrix("graph.txt");
    int **matrix = info.Matrix;
    vertices = info.vertices;
    edges = info.edges;
    direction = info.direction;

    printMatrix(matrix, vertices, vertices);

    for (int i=0; i<vertices; i++) {
        cout << "Indegree of Vertex " << i << ": " << inDegree(matrix, vertices, i) << endl;
        cout << "Outdegree of Vertex " << i << ": " << outDegree(matrix, vertices, i) << endl;
        cout << endl;
    }
```

```cpp

    // --------LIST--------------
    int vertices1, edges1, direction1;

    AdjList info1 = adjacencyList("graph.txt");

    AdjNode **list = info1.List;
    vertices1 = info1.vertices;
    edges1 = info1.edges;
    direction1 = info1.direction;

    printList(list, vertices1);
    for (int i=0; i<vertices; i++) {
        cout << "Indegree of Vertex " << i << ": " << inDegree(list, i, vertices1) << endl;
        cout << "Outdegree of Vertex " << i << ": " << outDegree(list, i) << endl;
        cout << endl;
    }

    return 0;
}
```

`

2. TOPOLOGICAL SORT

myCodes > graphs > topoSort.cpp > ...

```cpp
#include "adjMatFunctions.h"


bool allVisited(bool vertices[], int size) {
    for (int i=0; i<size; i++) {
        if (vertices[i]==false)
            return false;
    }
    return true;
}

int* topologicalSort(int **matrix, int vertices) {
    int ID[vertices];

    for (int i=0; i<vertices; i++)
        ID[i] = inDegree(matrix, vertices, i);

    bool visited[vertices] = {false};
    int *topSortList = new int[vertices], k=0;

    bool allVis = false;
    while (!allVis) {
        int minVertex = 0;
        for (int i=0; i<vertices; i++) {
            // finding minimum value which is not visited
            if ( ID[i]<=ID[minVertex] && visited[i]==false )
                minVertex = i;
        }

        visited[minVertex] = true;
        topSortList[k++] = minVertex;

```

`

```
        for (int j=0; j<vertices; j++) {
            if (matrix[minVertex][j]!=0)
                ID[j]--;
        }

        allVis = allVisited(visited, vertices);
    }

    return topSortList;
}



int main() {

    int vertices, edges, direction;
    AdjMatrix info = adjacencyMatrix("graph.txt");
    int **matrix = info.Matrix;
    vertices = info.vertices;
    edges = info.edges;
    direction = info.direction;

    int *sortedVertices = topologicalSort(matrix, vertices);

    cout << "Sorted Vertices w.r.t Independence:" << endl;
    for (int i=0; i<vertices; i++) {
        cout << sortedVertices[i] << " | ";
    }
    cout << endl;

    return 0;
}
```

`

3. DIJKSTRA ALGORITHM

myCodes > graphs > dijkstraAlgo.cpp

```cpp
#include "adjMatFunctions.h"

struct Vertex {
    Vertex *previousVertex;
    int id;
    int distance;
    bool visited;
    bool connected;
};

Vertex* createNewVertex(int id) {

    Vertex *newVertex = (Vertex*) malloc(sizeof(Vertex));
    newVertex->previousVertex = NULL;
    newVertex->visited = false;
    newVertex->connected = false;
    newVertex->id = id;
    newVertex->distance = 2147483647;

    return newVertex;
}

bool allVisited(Vertex **vertices , int size) {
    for (int i=0; i<size; i++) {
        if (vertices[i]->visited == false)
            return false;
    }
    return true;
}

int *connectedVertices(int **Matrix, int numOfVertices, int vertex) {

```

`

```
31  int *connectedVertices(int **Matrix, int numOfVertices, int vertex) {
32
33      int out_degree = outDegree(Matrix, numOfVertices, vertex);
34      int *connected_vertices = new int[out_degree];
35
36      int j=0;
37      for (int i=0; i<numOfVertices; i++)
38          if (Matrix[vertex][i] != 0)
39              connected_vertices[j++] = i;
40
41      return connected_vertices;
42
43  }
44
45  Vertex* nextVertex(Vertex **vertices, int size) {
46
47      Vertex* next_vertex = vertices[0];
48
49      int j;
50      for (j=1; j<size; j++) {
51          if (!vertices[j]->visited) {
52              next_vertex = vertices[j++];
53              break;
54          }
55      }
56
57      if (j==size) return NULL;
58
59      for (int i=j; i<size; i++) {
60          if (!vertices[i]->visited) {
61              if (vertices[i]->distance < next_vertex->distance) {
62                  next_vertex = vertices[i];
```

`

```
                    next_vertex = vertices[i];
                }
            }
        }

        return next_vertex;
    }

    Vertex** dijkstraAlgorithm(int **matrix, int vertices) {

        Vertex **allVertices = (Vertex**) malloc(vertices * sizeof(Vertex*));

        for (int i=0; i<vertices; i++)
            allVertices[i] = createNewVertex(i);

        Vertex *current_vertex = allVertices[0];
        current_vertex->distance = 0;
        current_vertex->connected = true;
        current_vertex->previousVertex = NULL;


        bool visit = false;
        while (!visit) {

            int *connected_vertices = connectedVertices(matrix, vertices, current_vertex->id);
            int num_of_connected_vertices = outDegree(matrix, vertices, current_vertex->id);

            for (int i=0; i<num_of_connected_vertices; i++) {

                int conVert = connected_vertices[i];
                Vertex *connected_vertex = allVertices[conVert];

```

`

```
            int current_edge_weight = matrix[current_vertex->id][connected_vertex->id];
            int current_vertex_weight = current_vertex->distance;

            int totalWeight = current_vertex_weight + current_edge_weight;

            if (!connected_vertex->connected) {
                connected_vertex->connected = true;
                connected_vertex->previousVertex = current_vertex;
                connected_vertex->distance = totalWeight;
            }

            else if (totalWeight < connected_vertex->distance) {
                connected_vertex->distance = totalWeight;
                connected_vertex->previousVertex = current_vertex;
            }
        }
        current_vertex->visited = true;
        visit = allVisited(allVertices, vertices);
        if (!visit)
            current_vertex = nextVertex(allVertices, vertices);
    }
    return allVertices;
}

void display(Vertex **vertices, int size) {
    cout << "(Vertex, Distance) <- Previous Vertex" << endl;
    for (int i=0; i<size; i++) {
        Vertex *current = vertices[i];
        while (current != NULL) {
            cout << "(" << current->id << ", " << current->distance << ") <- ";
            current = current->previousVertex;
        } cout << endl;
```

`

```
            current = current->previousVertex;
        } cout << endl;
    }
}

int main() {

    int vertices, edges, direction;
    AdjMatrix info = adjacencyMatrix("graph.txt");
    int **matrix = info.Matrix;
    vertices = info.vertices;
    edges = info.edges;
    direction = info.direction;

    Vertex **singlePathGraph = dijkstraAlgorithm(matrix, vertices);

    display(singlePathGraph, vertices);

    return 0;
}
```

`

4. PRIM'S ALGORITHM

myCodes > graphs > primsAlgo.cpp > primsAlgorithm(int **, int)

```cpp
#include "adjMatFunctions.h"

#define infinity 2147483647;

int** primsAlgorithm(int **Matrix, int num_of_vertices) {
    int **newMatrix = squareMatrix(num_of_vertices);
    int total_cost = 0;
    bool selected[num_of_vertices];
    selected[0] = true;

    int edgeNumber;
    for (edgeNumber=0; edgeNumber < num_of_vertices-1; edgeNumber++) {
        int minimum = infinity;
        int x, y;

        for (int i = 0; i < num_of_vertices; i++) {
            if (selected[i]) {
                for (int j=0; j<num_of_vertices; j++) {
                    if (!selected[j] && Matrix[i][j]!=0) {
                        if (Matrix[i][j]<minimum) {
                            minimum = Matrix[i][j];
                            x = i;
                            y = j;
                        }
                    }
                }
            }
        }

        newMatrix[x][y] = minimum;

        printf("%d - %d : %d\n", x, y, minimum);
```

`

```
31
32          printf("%d - %d : %d\n", x, y, minimum);
33
34          total_cost += minimum;
35          selected[y] = true;
36      }
37
38      printf("Total Cost = %d",total_cost);
39      return newMatrix;
40  }
41
42  int main() {
43
44      int vertices, edges, direction;
45      AdjMatrix info = adjacencyMatrix("graph.txt");
46      int **matrix = info.Matrix;
47      vertices = info.vertices;
48      edges = info.edges;
49      direction = info.direction;
50
51      int **newMatrix = primsAlgorithm(matrix, vertices);
52      printMatrix(newMatrix, vertices, vertices);
53
54      return 0;
55  }
```

5. KRUSKAL'S ALGORITHM

To covert Prim's Algo into Kruskal's, just remove the condition on line 17.

`