2K22/CO/236

classmate
Date
Page

DS - Assignment -1

Matrix Addition

```
void addMatrices (int m1[3][3], int m2[3][3], int res[3][3], int rows, int columns) {
    for (int i = 0; i<rows ; i++) {
        for (int j = 0 ; j<columns; j++) {
            res [i][j] = m1 [i][j] + m2[i][j];
        }
    }
}
```

Matrix Transpose

```
void transposeMatrix (int mat [3][3], int rows, int columns) {
    int res [columns][rows];
    for (int i = 0; i<rows ; i++) {
        for (int j = 0 ; j<columns ; j++) {
            res [i][j] = mat [j][i];
        }
    }
}
```

Matrix Multiplication

```
void multiplyMatrices (int m1[3][3], int m2 [3][3], int res[3][3], int r1, int c1, int c2) {
    for (int i = 0; i< r1; i++) {
        for (int j = 0 ; j < c2; j++) {
            res [i][j] = 0;
            for (int k = 0; k < c1; k++) {
                res[i][j] += m1[i][k] * m2[k][j];
            }
        }
    }
}
```

classmate
Date
Page

Sparse Matrix Addition

```
struct Sparse *add (struct Sparse * s1, struct Sparse * s2) {
      struct Sparse *sum;
      sum = (struct Sparse *) malloc (sizeof (struct Sparse));
      sum->e = (struct Element *) malloc ((s1->num + s2->num)* sizeof (struct Element));
      int i=0, j=0, k=0;
      while ( i< s1->e[i].i && j<s2->e[j].j) {
            if (s1->e[i].i < s2->e[j].i) { sum->e[k++] = s1->e[i++];}
            else if (s1->e[i].i > s2->e[j]i) {sum->e[k++] = s2->e[j++];}
            else {
                  if (s1->e[i].j < s2->e[j].j) { sum->e[k++] = s1->e[i++];}
                  else if (s1->e[i].j > s2->e[j].j) {sum->e[k++] = s2->e [j++];}
                  else {
                        sum->e[k] = s1->e[i];
                        sum->e[k].x = s1->e[i++].x + s2->e[j++].x;
                  }
            }
      }
      for(; i< s1->num; i++) { sum->e[k++] = s1->e[i]; }
      for(; j<s2->num; j++) { sum->e[k++] = s2->e[j]; }
      sum->m = s1->m;
      sum->n = s1->n;
      sum->num = k      // k stores no. of non-zero elements
      return sum;
}
```

2K22/CO/236

classmate
Date
Page

Sparse Matrix Transpose

```
struct Sparse *transpose (struct Sparse *s) {
    struct Sparse *result = (struct Sparse *) malloc (sizeof (struct Sparse));
    result→m = s→n
    result→n = s→m
    result→num = s→num
    result→e = (struct Element *) malloc (result→num * sizeof (struct Element));
    int colCount [s→n];
    for (int i=0; i<s→n; i++) { colCount[i] = 0;}
    for (int i=0; i<s→num; i++) { colCount[s→e[i].j]++; }
    int colPosition [s→n];
    colPosition [0] = 0;
    for (int i=1; i<s→n; i++){
        colPosition [i] = colPosition [i-1] + colCount[i-1];
    }
    for (int i=0; i<s→num; i++) {
        int j = s→e[i].j;
        result→e [colPosition[j]].i = s→e [i].j ;
        result→e [colPosition[j]].j = s→e [i].i ;
        result→e [colPosition[j]].x = s→e [i].x ;
        colPosition [j]++;
    }
    return result;
}
```

Sparse Matrix Multiplication

```
struct Sparse *multiply (struct Sparse *s1, struct Sparse *s2) {
        if (s1→n != s2→m) { printf("Multiplication not possible"); return NULL; }
        struct Sparse *result;
        result = (struct Sparse *) malloc (sizeof (struct Sparse));
        result→m = s1→m;
        result→n = s2→n;
        result→num = 0;
        result→e = (struct Element *) malloc (result→num * sizeof (struct Element));
        int sum, k=0;
        for (int i=0 ; i<s1→m ; i++) {
            for (int j=0 ; j<s2→n ; j++) {
                sum=0;
                for (int l=0; l<s1→num; l++) {
                    for (int m=0; m<s2→num; m++) {
                        if (s1→e[l].j == s2→e[m].i && s1→e[l].i == i && s2→e[m].j == j) {
                            sum += s1→e[l].x * s2→e[m].x;
                        }
                    }
                }
                if (sum != 0) {
                    result→num++;
                    result→e = (struct Element*) realloc (result→e, result→num * sizeof(struct Element));
                    result→e[k].i = i;
                    result→e[k].j = j;
                    result→e[k].x = sum;
                    k++;
                }
            }
        }
        return result;
}
```

2k22/CO/236

Polynomial Addition

```
struct Poly *add (struct Poly *p1, struct Poly *p2){
        struct Poly *sum;
        sum = (struct Poly *) malloc (sizeof (struct Poly));
        sum→terms = (struct Term*) malloc( (p1→n + p2→n) * sizeof (struct Term));
        int i=0, j=0, k=0;
        while (i<p1→n && j<p2→n) {
            if (p1→terms [i].exp > p2→terms [j].exp) {sum→terms [k++] = p1→terms [i++];
            else if (p1→terms [i].exp < p2→terms [j].exp){ sum→terms [k++] = p2→terms [j++];
            else {
                sum→terms [k]. exp = p1→ terms [i].exp;
                sum→terms [k++]. coeff = p1→terms [i++]. coeff + p2→terms [j++]. coeff;
            }
        }
        for (; i<p1→n ; i++) { sum→terms [k++] = p1→terms [i]; }
        for (; j<p2→n ; j++) { sum→terms [k++] = p2→terms [j]; }
        sum→n = k      // k stores no. of elements in sum
        return sum;
}
```

Polynomial Multiplication

```
struct Poly *multiply (struct Poly *p1, struct Poly *p2) {
    struct Poly *result = (struct Poly *)malloc (sizeof (struct Poly));
    result→n = p1→n + p2→n;
    result→terms = (struct Term *) malloc (result→n * sizeof (struct Term));
    int k = 0;
    for (int i = 0; i < p1→n; i++) {
        for (int j = 0; j < p2→n; j++) {
            result→terms [k]. coeff = p1→terms [i]. coeff * p2→terms [j]. coeff ;
            result→terms [k]. exp = p1→terms [i]. exp + p2→terms [j]. exp ;
            k++;
        }
    }
    for (int i = 0; i < result→n - 1; i++) {
        for (int j = i+1; j < result→n ; j++) {
            if (result→terms [i]. exp == result→terms [j]. exp) {
                result→terms [i]. coeff += result→terms [j]. coeff;
                for (int l = j ; l < result→n -1; l++) { result→terms [l] = result→terms [l+1]; }
                result→ n--;
                j--;
            }
        }
    }
    return result;
}
```

2K22/CO/236

Using Stack, check for Palindrome string

```
bool isPalindrome (const char* str) {
    struct Stack stack;
    initialize (&stack);
    int length = strlen (str);
    for (int i=0; i< length; i++) { push (&stack, str [i]); }
    for (int i=0; i<length; i++) {
        if (pop (&stack) != str [i]) { return false; }
    }
    return true;
}
```

Application of Heaps as priority queues

1. Initialization: Create empty heap data structure. This can be binary min-heap or binary max-heap, depending on whether you want to implement min or max-priority queue.
2. Insertion: To insert an element with a priority value into priority queue
   - Add element to heap
   - Re-heapify the heap to maintain heap property (min-heap or max-heap property)
3. Extraction: To extract an element with highest (max-priority queue) or lowest (min-priority queue) priority from queue:
   - Retrieve root element of Heap (one with Highest or Lowest priority)
   - Replace root with last element in Heap.
   - Remove last element
   - Re-heapify heap to maintain heap property
4. Peek: To peek at element with highest (max-priority queue) or lowest (min-priority queue) priority without removing it:
   - Return root element of Heap