**Veri Yapıları 2.Ödev**

**Ad : Younes Rahebi**

**Öğrenci No : B221210588**

**Grup : 1.Öğretim B Grubu**

**Öğrendiklerim:**

Dinamik Dizi İşlemleri: Kod, bir yığın dizisi ve AVL ağaçları oluşturmak için kendi dinamik dizilerini kullanır. Bu, diziler için ihtiyaç duyuldukça bellek ayırma, yeniden boyutlandırma ve nesnelerin yaşam süreçlerini yapılandırıcılar ve yıkıcılar ile kontrol etme anlamına gelir.

AVL Ağacı Kurulumu: AVL ağacı kodu, ekleme sonrası ağacın dengeli kalmasını sağlamak için dengeleme teknikleri hakkında bilgi verir. Burada ağaç rotasyonlarını kavramak ve uygulamak çok önemlidir.

Yığın Kullanımı: Yığın (Stack) yapısını baştan yazmak, Last-In-First-Out (LIFO) mantığını, dinamik yeniden boyutlandırma işlemini ve C++'da bellek yönetimini öğretir.

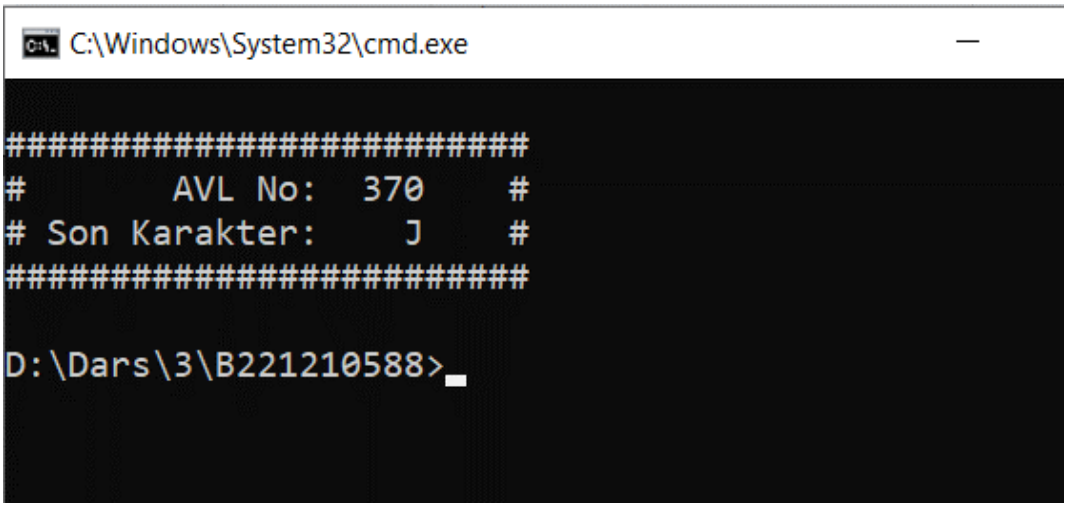
**Ödevde Yaptıklarım:**

Dosya Okuma: Programın, bir dosyadan veri alıp, bu verileri AVL ağaçlarına yerleştirmek için bir fonksiyonu vardır.

Ağaç İşlemleri: Uygulama, AVL ağaçlarında yaprak olmayan düğümlerin toplam değerini ve bu değerlerin karakter karşılıklarını bulma yeteneğine sahiptir.

Yığın ile Postorder Gezinme: AVL ağacının postorder sıralamasını yapar ve sadece yaprak düğümlerini bir yığına ekler.

AVL'de Silme İşlemi: AVL sınıfındaki searchAndDelete metodu, kendini dengeli tutan ağaçlarda silme işlemleri hakkında bilgi verir.



**Zorlandığım Kısımlar:**

Bellek Yönetimi: StackDynamicArray ve AVLDynamicArray'ın yıkıcılarında doğru bellek yönetimi yapmak zorluklardan biriydi. Bellek kaçaklarını önlemek ve nesneleri uygun sırada silmek, tanımsız davranışları engellemek için hayatiydi.

AVL Dönüşlerinin Uygulanması: AVL dönüşlerini doğru şekilde uygulamak dikkatli bir değerlendirme gerektirdi, çünkü hatalı uygulamalar dengesiz bir ağaca ve verimsiz işlemlere neden olabilirdi.

main.cpp

Miscellaneous Files (Global Scope)

```
38  while (avlTrees.count > 1) {
39
40      StackDynamicArray stackLeaves;
41
42      for (int i = 0; i < avlTrees.count; ++i) {
43          AVL* currentAVLTree = avlTrees.getAVL(i);
44          if (currentAVLTree != nullptr) {
45              Stack* leavesStack = new Stack;
46              currentAVLTree->postorderWithStack(currentAVLTree->root, *leavesStack);
47              stackLeaves.addStack(leavesStack);
48          }
49      }
50
51      bool removeSmallest = true;
52
53      while (stackLeaves.count > 1) {
54          int targetIndex = -1;
55      int targetValue = removeSmallest ? INT_MAX : INT_MIN;
56
57          for (int i = 0; i < stackLeaves.count; ++i) {
58              if (stackLeaves.getStack(i)->empty()) {
59                  continue;
60              }
61
62              int currentTop = stackLeaves.getStack(i)->top();
63              bool isTarget = removeSmallest ? (currentTop < targetValue) : (currentTop > targetValue);
64
65              if (isTarget) {
66                  targetValue = currentTop;
67                  targetIndex = i;
68              }
69          }
70
71          if (targetIndex == -1) {
72              break;
73          }
74
75          stackLeaves.getStack(targetIndex)->pop();
76          removeSmallest = !removeSmallest;
77
78          if (stackLeaves.getStack(targetIndex)->empty()) {
79              avlTrees.removeAt(targetIndex);
80              stackLeaves.removeAt(targetIndex);
81
82              for (int i = 0; i < avlTrees.count; ++i) {
83                  AVL* currentAVL = avlTrees.getAVL(i);
84                  if (currentAVL != nullptr) {
85                      int nonLeafTotalValue = currentAVL->calculateNonLeafTotalValue(currentAVL->root);
86                      int asciiValue = nonLeafTotalValue % (90 - 65 + 1) + 65;
87                      std::cout << static_cast<char>(asciiValue);
88                  }
89              }
```