//Ödev 1

//Ogrenci1 No: 1030520793

//Ogrenci1 İsmi: Buse ÇEŞMECİ

//Ogrenci2 No: 1030520725

//Ogrenci2 İsmi: Çağrı KUZULU

//Ogrenci3 No: 1030520759

//Ogrenci3 İsmi: Musa BOZ

//Ogrenci4 No: 1030520755

//Ogrenci4 İsmi: Şeyma TEZCAN

//Ders kodu: BZ205

//Dosya ismi: C:\Users\User\Desktop\veriyapilariproje2018tümproje

# FONKSİYONLARIN ÇALIŞMA MANTIĞI

Öncelikle class node yapısı oluşturduk ve bunlara da bizim daha sonraki fonksiyonlarda ihtiyacımız olacak verileri ekledik, burada b+ ağacının yapısı için gerekli olan 3 adet veriyi de aynı class yapısında tutmaya karar verdik.

```
class node
{
```

```
        public:
                double numara;
                char* ad;
                char* soyad;
                char* bolum;
                char* alinanDers;
                node* sonraki;
                node* onceki;

        //B+ ağacının attributeleri
        vector<double> Keys;
        node *Parent;
        bool isLeaf;
};
```

Aşağıdaki class List yapısında ise ihtiyacımız olacak bağlantılı liste(constructor) ve yıkım için gerekli olacak(destructor) fonksiyonunu ekliyoruz, daha sonra ise fonksiyonlarımızı teker teker tanımlıyoruz. Burada 2 tane private eleman var(node* head,node* last) ve bunlara dışardan direk erişim sağlanamıyor.Bu 2 yapıyı kullanmak için List tanımı yapılması gerekiyor.

```
class List
{
        public:
     List(void){head = 0; last = NULL;} //  constructor
     ~List(void); //  destructor
            bool IsEmpty()
       {
        return head == NULL;
       }
      //Bağlantılı listede tutulan düğümler üzerinde yapılan işlemler
                node* insertNode(int index,double numara,char* ad,char* soyad,char* bolum);
                int findNode(double veri);
                int deleteNode(double veri);
                void displayList(void);
        private:
                node* head;
                node*last;
};
```

Burada da 2. Soruda gerekli olan ders işlemleri için ayrı bir class List2 yapısı oluşturmaya karar verdik. Buradaki

fonksiyonlar ise Dersler ile ilgili ve sıralama algoritmalarıyla ilgilidir.

```
//2. liste tanımı ve fonksiyonları
class List2
{
	public:
		List2(void)
		{
			head=0;
			last=NULL;
		}
		~List2(void);
		bool IsEmpty()
		{
			return head == NULL;
		}
		//List2'de kullanılacak fonksiyonlarımız
		node* dersEkle(int index,double numara,char* ad,char* soyad,char*
bolum,char* alinanDers);
		int dersSil(double input);
		int dersArama(double input);
		void sadeceProgramlamaDersi(void);
		void programlamaDersiAGrubu(void);
		void programlamaDersiBGrubu(void);
		void displayList2(void);
		void heapify(int dizi[],int n,int i);
		void kumelemeSiralama(int dizi[],int n);
		private:
			node* head;
			node* last;
};
```

insertNode fonksiyonunda 5 adet parametremiz var(index,numara,ad,soyad,bolum). Bu fonksiyonun çalışma mantığı; fonksiyon 5 parametreyi dışarıdan alır ve işleme başlar, ilk başta index'in 0'dan küçük olup olmadığına bakar eğer index 0'dan küçükse return NULL dönderir, 0'dan büyük ise node* currNode=head; değerini alır, burada bu işlemi kullanmamızın sebebi currNode değerini baştan başlatmaktır daha sonra while döngüsüne girer ve sona gelene kadar o düğümün currIndexini artırır. Daha sonra if ile currNode'un 0 olup olmadığına bakar, eğer bu if komutu doğru ise return 0

dönderir, değilse yeni bir newNode oluşturur ve bunun içine parametrelerimizi atar. Bu işlemlerden sonra ise if ve else komutlarını kullanarak düğümlerin adreslerini sıralar. En son ise return newNode komutunu dönderir.

```
node* List::insertNode(int index,double numara,char* ad,char* soyad,char* bolum)
{
	//Eğer index 0'dan küçükse NULL değerini dönderir
	if(index<0)
		return NULL;
	int currIndex=1;
	node* currNode=head;
	while(currNode&&index>currIndex)
	{
		currNode = currNode->sonraki;
		currIndex++;
	}
	if(index >0&&currNode==0)
	return 0;
	//Düğümdeki verileri parametrelere eşitliyoruz.
	node* newNode = new node;
	newNode->numara=numara;
	newNode->ad=ad;
	newNode->soyad=soyad;
	newNode->bolum=bolum;

	if(index==0)
	{
		newNode->sonraki=head;
		last=head=newNode;
		newNode->onceki=NULL;
	}
	else
	{
		newNode->sonraki=currNode->sonraki;
		currNode->sonraki=newNode;
		newNode->onceki=currNode;
		last=newNode;
	}
	return newNode;
}
```

İnt List::findNode(double input) fonksiyonu dışardan aldığı öğrenci numarası ile fonksiyon içinde arama yapar. Burada node* currNode=head ifadesini kullanmamızın sebebi diğer fonksiyonlarda da olduğu gibi baştan işleme başlamasıdıri bu

sayede tüm listeyi gezebilir. Eğer girilen öğrenci numarası o düğümdeki herhangi bir elemana(currNode->numara) eşitse o elemanın indexini return currIndex komutuyla dönderir, eşitlik sağlanmazsa return 0 değerini alır.

```
int List::findNode(double input)
{
	node* currNode=head;
	int currIndex=1;
	while(currNode && currNode->numara!=input)
	{
		currNode=currNode->sonraki;
		currIndex++;
	}
	//Öğrenci numarası düğümlerdeki numaraların herhangi birine eşit ise indexinin dönderir
	if(currNode)
	return currIndex;
	//Öğrenci numarası düğümlerdeki numaraların herhangi birine eşit değil ise 0 değerini dönderir.
	return 0;
}
```

int List::deleteNode(double input) fonksiyonu double tipinde bir öğrenci numarası parametresi alır ve daha sonra 2 adet düğüm içi eleman ve 1 adet int currIndex elemanlarını tanımlar. Daha sonra while döngüsünün içinde düğüm oldukça ve düğümün içindeki numara o parametreye eşit olmadıkça döngü devam eder ve her seferinde currIndex elemanı 1 artar. O düğüm bulunduğu anda(currNode) while'dan çıkar ve if döngüsüne dahil olur. Burada ki işlemlerden sonra return currIndex komutuna gelir. Eğer girilen parametre o düğümlerdeki herhangi bir öğrenci numarasına eşit olmazsa bu fonksiyon return 0 değerini dönderir.

```
int List::deleteNode(double input)
{
```

```
	node* prevNode=NULL;
	node* currNode=head;
	int currIndex=1;
	while(currNode && currNode->numara!=input)
	{
		prevNode = currNode;
		currNode = currNode->sonraki;
		currIndex++;
	}
	if(currNode)
	{
		if(prevNode)
		{
			prevNode->sonraki=currNode->sonraki;
			delete currNode;
		}
		else
		{
			head=currNode->sonraki;
			delete currNode;
		}
		return currIndex;
	}
	return 0;
}
```

void List::displayList() fonksiyonu listedeki öğrencileri listeler. Bu fonksiyonun çalışma mantığı ise düğüm head'den başlar ve while döngüsüne girer, bu döngü düğümün sonuna gelene kadar(currNode!=NULL), her bir düğümdeki verileri ekrana yazdırır.

```
void List::displayList()
{
	int num=0;
	//Burada currNode'u heade eşitlememizin sebebi düğüme baştan başlamak
	node* currNode=head;
	//currNode NULL yani sona gelene kadar bu işlemleri yapar.
	while(currNode!=NULL)
	{
```

```
                cout<<""<<currNode->numara<<" "<<currNode->ad<<" "<<currNode->soyad<<" "<<currNode->bolum<<endl;
                currNode=currNode->sonraki;
                num++;
        }
        cout<<"Listedeki Ogrenci Sayilari: "<<num<<endl;
}
```

List2::dersEkle(int index,double numara,char*ad,char*soyad,char*bolum,char*alinanDers) fonksiyonu, aynı insertNode fonksiyonu mantığında çalışır, buradaki tek fark ise alıinanDers parametresi olmasıdır ve her düğüme öğrenci bilgileriyle birlikte alinanDersi de eklemesidir.

```
node* List2::dersEkle(int index,double numara,char* ad,char* soyad,char* bolum,char* alinanDers)
{
        if(index<0)
                return NULL;
        int currIndex=1;
        node* currNode=head;
        while(currNode&&index>currIndex)
        {
                currNode = currNode->sonraki;
                currIndex++;
        }
        if(index >0&&currNode==0)
        return 0;
        node* newNode = new node;
        newNode->numara=numara;
        newNode->ad=ad;
        newNode->soyad=soyad;
        newNode->bolum=bolum;
        newNode->alinanDers=alinanDers;

        if(index==0)
        {
                newNode->sonraki=head;
                last=head=newNode;
                newNode->onceki=NULL;
```

```
	}
	else
	{
		newNode->sonraki=currNode->sonraki;
		currNode->sonraki=newNode;
		newNode->onceki=currNode;
		last=newNode;
	}
	return newNode;
}
```

List2::dersSil(double input) fonksiyonu aynı deleteNode fonksiyonunda olduğu gibi bir double input parametresi alır(öğrenci numarası) ve aynı mantıkla çalışır.

```
int List2::dersSil(double input)
{
	node* prevNode=NULL;
	node* currNode=head;
	int currIndex=1;
	while(currNode && currNode->numara!=input)
	{
		prevNode = currNode;
		currNode = currNode->sonraki;
		currIndex++;
	}
	if(currNode)
	{
		if(prevNode)
		{
			prevNode->sonraki=currNode->sonraki;
			delete currNode;
		}
		else
		{
			head=currNode->sonraki;
			delete currNode;
		}
		return currIndex;
	}
	return 0;
}
```

List2::dersArama(double input) fonksiyonu findNode() fonksiyonunda olduğu gibi double input(öğrenci numarası) parametresini alıyor.Burada findNode() fonksiyonundan farklı olan nokta ise o düğümün isim,soyisim,bölüm ve alınan ders bilgilerini ekrana yazdırıyor.

```
int List2::dersArama(double input)
{
	node* currNode=head;
	int currIndex=1;
	while(currNode && currNode->numara!=input)
	{
		currNode=currNode->sonraki;
		currIndex++;
	}
	if(currNode)
	cout<<currNode->ad<<currNode->soyad<<currNode->bolum<<currNode->alinanDers<<endl;
	return 0;
}
```

List2::sadeceProgramlamaDersi() fonksiyonu ilk olarak bir sayaç değeri alıyor.Daha sonra node*currNode=head komutuyla beraber while döngüsüne devam ediyor.Her düğümü teker teker geziyor ve o düğümde alınan ders “BilgisayarProgramlama” ya eşit olduğu anda o öğrencinin bilgilerini ekrana yazdırıyor.Düğümün sonuna geldiği anda (currNode!=NULL) döngüden çıkıyor ve burada da tek bir ekrana yazma işlemini yapıyor daha sonra ise işlem sona eriyor.

```
void List2::sadeceProgramlamaDersi()
{
	int sayac=0;
	node* currNode=head;
	while(currNode!=NULL)
	{
```

```
		if(currNode->alinanDers=="BilgisayarProgramlama")
		{
			cout<<""<<currNode->numara<<" "<<currNode->ad<<" "<<currNode->soyad<<" "<<currNode->bolum<<" "<<endl;
			sayac++;
		}
		currNode=currNode->sonraki;
	}
	cout<<"Bilgisayar Programlama Dersini Alan Toplam Ogrenci Sayisi: "<<sayac<<endl;
}
```

List2::programlamaDersiAGrubu() fonksiyonu bir soyad dizgisi alıyor ve while döngüsüne giriyor. Döngünün içinde her seferinde o düğümdeki öğrencinin soyismini dizgiye ekliyor ve daha sonra bu öğrencinin alınan dersi bilgisayar programlamaya eşit olduğu anda, soyismin baş harfinin alfabetik durumunu(A-K) kontrol ediyor ve kontrol komutu sağlanırsa ekrana yazıyor.

```
void List2::programlamaDersiAGrubu()
{
	char *soyadi=new char[20];
	int sayac=0;
	node* currNode=head;
	while(currNode!=NULL)
	{
		soyadi=currNode->soyad;
		if(currNode->alinanDers=="BilgisayarProgramlama")
			if(soyadi[0]>='A'&&soyadi[0]<='K')
				{
					cout<<""<<currNode->numara<<" "<<currNode->ad<<" "<<currNode->soyad<<" "<<currNode->bolum<<" "<<endl;
					sayac++;
				}
		currNode=currNode->sonraki;
	}
	cout<<"Bilgisayar Programlama Dersini Alan Ogrencilerin Soyisimlerinin A-K Arasi Olanlarin Sayisi: "<<sayac<<endl;
}
```

List2::programlamaDersiBGrubu() fonksiyonu bir soyad dizgisi alıyor ve while döngüsüne giriyor. Döngünün içinde her seferinde o düğümdeki öğrencinin soyismini dizgiye ekliyor ve daha sonra bu öğrencinin alınan dersi bilgisayar programlamaya eşit olduğu anda, soyismin baş harfinin alfabetik durumunu(L-Z) kontrol ediyor ve kontrol komutu sağlanırsa ekrana yazıyor.

```
void List2::programlamaDersiBGrubu()
{
	char *soyadi=new char[20];
	int sayac=0;
	node* currNode=head;
	while(currNode!=NULL)
	{
		soyadi=currNode->soyad;
		if(currNode->alinanDers=="BilgisayarProgramlama")
			if(soyadi[0]>='L'&&soyadi[0]<='Z')
			{
				cout<<""<<currNode->numara<<" "<<currNode->ad<<" "<<currNode->soyad<<" "<<currNode->bolum<<" "<<endl;
				sayac++;
			}
			currNode=currNode->sonraki;
	}
	cout<<"Bilgisayar Programlama Dersini Alan Ogrencilerin Soyisimlerinin L-Z Arasi Olanlarin Sayisi: "<<sayac<<endl;
}
```

List2::displayList2() fonksiyonu düğümün başlangıcından başlar ve düğüm sonuna gelene kadar o öğrencinin bilgilerini ve aldığı dersi ekrana teker teker yazar.

```
void List2::displayList2()
{
	int sayac=0;
	node* currNode=head;
	while(currNode!=NULL)
	{
```

```
                cout<<""<<currNode->numara<<" "<<currNode->ad<<" "<<currNode->soyad<<" "<<currNode->bolum<<" "<<currNode->alinanDers<<endl;
                currNode=currNode->sonraki;
                sayac++;
        }
        cout<<"Ders Alan Ogrenci Sayilari: "<<sayac<<endl;
}
```

List2::heapify(int dizi[],int n,int i) fonksiyonu başlangıçta kökü en büyük olarak eşitler. Eğer kökün sol tarafı kökten büyükse solu en büyük olarak, sağ tarafı büyükse sağı en büyük olarak eşitler. Yapının en büyük elemanı kök değilse swap komutuyla en büyük ile takas eder.

```
void List2::heapify(int dizi[],int n,int i)
{
        int enBuyuk=i;//başlangıçta kökü en buyuk olarak eşitliyoruz.
        int sol=2*i+1;//kökün sol tarafı
        int sag=2*i+2;//kökün sağ tarafı

        //eğer kökün sol tarafı kökten büyükse bu işlemler uygulanmalı
        if(sol<n && dizi[sol]>dizi[enBuyuk])
        enBuyuk=sol;

        //eğer kökün sağ tarafı kökten daha büyükse bu işlemler uygulanmalı
        if(sag<n && dizi[sag]>dizi[enBuyuk])
        enBuyuk=sag;

        //eger dizinin en buyuk elemanı kök değilse takas işlemi uygulanmalı
        if(enBuyuk!=i)
        {
                swap(dizi[i],dizi[enBuyuk]);

                heapify(dizi,n,enBuyuk);
        }
```

List2::kumelemeSiralama(int dizi[],int n) fonksiyonu main fonksiyonu tarafından çağrıldığında, düğümü baştan sona kontrol edip, o düğümde bilgisayar programlama dersi alan

öğrenci varsa o öğrencinin numarasını diziye ekleyip, eklenen diziden sıralama yapar.

```
void List2::kumelemeSiralama(int dizi[],int n)
{
	//burada yapmamız gereken düğümü baştan sona kontrol edip o düğümde BilgisayarProgramlama dersini alan bir öğrenci varsa
	//o öğrencinin numarasını diziye ekleyip, eklediğimiz diziden sıralama yapmaktır.
	node *currNode=head;
	n=0;
	while(currNode!=NULL)
	{
		if(currNode->alinanDers=="BilgisayarProgramlama")
		{
		dizi[n]=currNode->numara;
		n++;
		}
		currNode=currNode->sonraki;
	}

	//diziyi yeniden düzenleme
	for(int i=n/2-1;i>=0;i--)
		heapify(dizi,n,i);

	//heap den bire bir eleman çekme
	for(int i=n-1;i>=0;i--)
	{
		swap(dizi[0],dizi[i]);

		heapify(dizi,i,0);
	}

	cout<<"Kumeleme Siralama Algoritmasi\n"<<endl;
	for(int i=0;i<n;++i)
		cout<<dizi[i]<<" "<<endl;
	cout<<"\n";
}
```

List:: List::~List(void) fonksiyonu oluşturulan bağlantılı listenin yıkım fonksiyonudur. Başlangıçta currNode=head ve

nextNode=NULL atıyor daha sonra currNode!=NULL(son düğüme gelene kadar) olana kadar nodeları siliyor.

```
List::~List(void)
{
	node* currNode=head;
	node* nextNode=NULL;

	while(currNode!=NULL)
	{
		nextNode=currNode->sonraki;
		//destroy the current Node
		delete currNode;
		currNode=nextNode;
	}
}
```

List2::~List2(void) fonksiyonu aynı ilk List'de olduğu gibi yıkıcı rolü üstleniyor.

```
List2::~List2(void)
{
	node* currNode=head;
	node* nextNode=NULL;

	while(currNode!=NULL)
	{
		nextNode=currNode->sonraki;
		delete currNode;
		currNode=nextNode;
	}
```

B+ ağacı başlangıcı

Class NonLeafNode:public node,B+ yapısının yapraksız düğümü için bir sınıf tanımlar.

```
class NonLeafNode : public node {
	/*
		Hedef: B+ tree'nin yapraksız düğümü için bir sınıf tanımlar
```

```
                Yaklaşım: Node sınıfı düğüm yapısını tanımlar
        */
public:
  vector<node *> Children;
  NonLeafNode() {
    isLeaf = false;
    Parent = NULL;
  }
};
```

# Class LeafNode:public node B+ yapısının yapraklı düğümü için bir sınıf tanımlar.

```
class LeafNode : public node {
        /*
                Hedef: B+ tree'nin yapraklı düğümü için bir sınıf tanımlanır
                Yaklaşım: Bir yaprak düğümü (Genel düğümden inherit eder)
        */
public:
  node *Next;
  vector<string> Value;
  LeafNode() {
    isLeaf = true;
    Next = NULL;
    Parent = NULL;
  }
};

class BPTree {
  /*
                Hedef: B+ tree sınıfı oluşturma
                Yaklaşım: sınıf tanımlama
        */
private:
  node *head;
  int n;      // ağacın derecesi

public:
  BPTree(int);
  void insert(double, string);
  void remove(double);
  string find(double);
  void printKeys();
```

```
  void printValues();
  ~BPTree();
};
```

B+ tree sınıfı, ağacı oluşturmayı hedefler.

```
BPTree::BPTree(int degree) {
  // Hedef: B+ tree sınıfının yapıcı fonksiyonu, ağacı oluşturmayı hedefler.

  n = degree;
  head = new LeafNode();
}
```

B+ ağacının yıkıcı fonksiyonu ağaç silmeyi hedefler.
Doğru olmayan tüm ağaç silinmelidir.

```
BPTree::~BPTree() {
  // Hedef: B+ ağacının yıkıcı fonksiyonu ağacı silmeyi hedefler.
        // Yapılacaklar: Doğru olmayan, bütün ağacı silmem lazım, sadece kök düğümü değil

  if (head) {
    delete head;
  }
}
```

void BPTree::insert(double key,string value) fonksiyonu B+ ağacına eleman ekliyor.Eleman için uygun yeri bulup gerekli işlemleri yapıyor.

```
void BPTree::insert(double key, string value) {
        // Hedef: B+ ağacına eleman eklemeyi hedefler
        // Eleman için uygun yeri bul

         node *t = this->head; // kökten başlama


        while (!t->isLeaf) {  // bir yaprak düğümüne ulaşana kadar
    int flag = 0;
    // anahtar (key) dizisinde daha büyük bir değer bulunana kadar, ileriye devam et
```

```
  for (int i = 0; i < t->Keys.size(); ++i) {
   // daha büyük değerde eleman bulundu
    if (t->Keys[i] > key) {
      // Yapraksız düğüm olarak şeklini değiştir ve i. çocuğa git
      t = ((NonLeafNode *)t)->Children[i];
      flag = 1;
      break;
    }
  }
  // eğer hiçbir anahtar daha büyük değilse, en büyük anahtar odur
  if (!flag) {
        // smartly put value  ?? check
    t = ((NonLeafNode *)t)->Children[t->Keys.size()];
  }
 }


 // eğer anahtar sonuncudan sonra ise (ayrıca anahtar güncel olarak 0 anahtara sahipse ele
alınır)
 if (t->Keys.size() == 0 || key > t->Keys.back()) {
  t->Keys.push_back(key);
  ((LeafNode *)t)->Value.push_back(value);

 } else {
  for (int i = 0; i < t->Keys.size(); ++i) { // yaprak düğümüne ekleme
    if (t->Keys[i] == key) {
      cout << "Iki tane ayni olan anahtar ekleyemezsiniz!" << endl;
      return;
    } else if (t->Keys[i] > key) {
      t->Keys.insert(t->Keys.begin() + i, key);
      ((LeafNode *)t)
        ->Value.insert(((LeafNode *)t)->Value.begin() + i, value);
      break;
    }
  }
 }
 if (t->Keys.size() > this->n) { // yaprak düğümünü ayırma
  node *tnew = new LeafNode();
  tnew->Parent = t->Parent;

  // // ikinci yarıyla yeni yaprak yapma - anahtarları ekleme
  tnew->Keys.insert(tnew->Keys.begin(), t->Keys.begin() + ceil((n + 1) / 2), t->Keys.end());
  // // ikinci yarıyla yeni yaprak yapma - değerleri ekleme
  ((LeafNode *)tnew)
```

```
      ->Value.insert(((LeafNode *)tnew)->Value.begin(),
                     ((LeafNode *)t)->Value.begin() + ceil((n + 1) / 2),
                     ((LeafNode *)t)->Value.end());

   // ikinci yarım anahtarları ve değerleri orjinalden silme
   t->Keys.erase(t->Keys.begin() + ceil((n + 1) / 2), t->Keys.end());
   ((LeafNode *)t)
      ->Value.erase(((LeafNode *)t)->Value.begin() + ceil((n + 1) / 2),
                     ((LeafNode *)t)->Value.end());

   // yeni yapılmış yaprak noktalarına NULL değeri veriliyor
   ((LeafNode *)tnew)->Next = ((LeafNode *)t)->Next;
   // eski yaprak noktalarını yeni yaprak düğümüne
   ((LeafNode *)t)->Next = tnew;


   key = t->Keys[ceil((n + 1) / 2) - 1];

   // t'nin ebeveyni yapraksız düğümdür
   while (t->Parent != NULL) {
     //şimdi yeni olan t=t->parent dır, çünkü bütün sürecin ebeveyn üzerinde tamamlanması
gerekir.
     t = t->Parent;

     for (int i = 0; i < t->Keys.size(); ++i) {

       // eğer güncel anahtar varolan anahtarlar içerisinde en büyüğüyse
       if (key > t->Keys.back()) {
         // because it is largest, push it at last çünkü o en büyüğü, onu en sona at
         t->Keys.push_back(key);
         // yeni ayrılmış yaprak düğümünü ebeveyninin çocuk dizisine at
         ((NonLeafNode *)t)->Children.push_back(tnew);
         break;
       }
       // eğer güncel anahtar en büyüğü değilse
       else if (t->Keys[i] > key) {
         // t çocuğunu onun çocuk dizisine ekle
         t->Keys.insert(t->Keys.begin() + i, key);
         // tnew çocuğunu onun çocuğunun dizisine ekleme
         ((NonLeafNode *)t) ->Children.insert(((NonLeafNode *)t)->Children.begin() + i + 1,
tnew);
         break;
       }
     }
```

```
    // eğer ebeveynlerin ayrıca ayrılmaya ihtiyacı varsa
    if (t->Keys.size() > this->n) {
      // yeni düğüm yap
      node *nright = new NonLeafNode();
      nright->Parent = t->Parent;
      // ikiye ayır
      nright->Keys.insert(nright->Keys.begin(),
                          t->Keys.begin() + floor((n + 2) / 2),
                          t->Keys.end());
      ((NonLeafNode *)nright)
         ->Children.insert(((NonLeafNode *)nright)->Children.begin(),
                           ((NonLeafNode *)t)->Children.begin() +
                              floor((n + 2) / 2),
                           ((NonLeafNode *)t)->Children.end());
      for (int i = floor((n + 2) / 2);
           i < ((NonLeafNode *)t)->Children.size(); ++i) {
        ((NonLeafNode *)t)->Children[i]->Parent = nright;
      }
      key = t->Keys[floor((n + 2) / 2) - 1];
      t->Keys.erase(t->Keys.begin() + floor((n + 2) / 2) - 1, t->Keys.end());
      ((NonLeafNode *)t)
         ->Children.erase(((NonLeafNode *)t)->Children.begin() +
                             floor((n + 2) / 2),
                          ((NonLeafNode *)t)->Children.end());
      tnew = nright;
    } else {
      tnew->Parent = t;
      return;
    }
  }
  // köke eriştiğimizde
  if (t->Parent == NULL) {
    // yapraksız düğüm oluştur
    t->Parent = new NonLeafNode();

    // t ve tnew'i bu yeni yapraksız düğümün çocuk dizisine ekle
    ((NonLeafNode *)t->Parent) -> Children.insert(((NonLeafNode *)t->Parent)-
>Children.begin(), t);
    ((NonLeafNode *)t->Parent) -> Children.insert(((NonLeafNode *)t->Parent)-
>Children.begin() + 1, tnew);

    if (t->isLeaf) {
      // yaprak düğümünün son anahtarını ebeveyn düğümünün başına taşı
      (t->Parent)->Keys.insert((t->Parent)->Keys.begin(), t->Keys.back());
```

```
    } else {
      // t son çocuğunun son anahtarını t'nin ebeveyninin başlangıcına taşı
      (t->Parent)->Keys.insert(
          (t->Parent)->Keys.begin(),
          ((NonLeafNode *)t)->Children.back()->Keys.back());
    }
    tnew->Parent = t->Parent;
    head = t->Parent;
   }

  } else {
   return;
  }
}
```

string BPTree::find(double key) fonksiyonu B+ ağacı içinde eleman arar. Aranan eleman ağaç içinde yer almıyorsa,yer almadığını belirten ifadeyi return eder.

```
string BPTree::find(double key) {
        // Hedef: fonksiyon, B+ ağacının içerisinde elemanları aramayı hedefler

  node *t = this->head;
  while (!t->isLeaf) { // doğru yeri bul
   int flag = 0;
   for (int i = 0; i < t->Keys.size(); ++i) {
    if (t->Keys[i] >= key) {
     t = ((NonLeafNode *)t)->Children[i];
     flag = 1;
     break;
    }
   }
   if (!flag) {
    t = ((NonLeafNode *)t)->Children[t->Keys.size()];
   }
  }
  for (int i = 0; i < t->Keys.size(); ++i) {
   if (t->Keys[i] == key) {
    return ((LeafNode *)t)->Value[i];
   }
  }
  return "Bu anahtar B+ agacinin icerisinde yer almiyor!";
```

```
}
```

# void BPTree::printKeys() fonksiyonu B+ ağacı içindeki tüm anahtarları yazdırır.

```
void BPTree::printKeys() {
        //      Hedef: fonksiyon, B+ ağacının içerisindeki bütün anahtarları yazdırmayı
hedefler

  if (head->Keys.size() == 0) {
    cout << "[]" << endl;
    return;
  }
  vector<node *> q;
  q.push_back(head);
  while (q.size()) {
    unsigned long size = q.size();
    for (int i = 0; i < size; ++i) {
      if (!q[i]->isLeaf) {
        for (int j = 0; j < ((NonLeafNode *)q[i])->Children.size(); ++j) {
          q.push_back(((NonLeafNode *)q[i])->Children[j]);
        }
      }
      cout << "[";
      int nk = 0;
      for (nk = 0; nk < q[i]->Keys.size() - 1; ++nk) {
        cout << q[i]->Keys[nk] << ",";
      }
      cout << q[i]->Keys[nk] << "] ";
    }
    q.erase(q.begin(), q.begin() + size);
    cout << endl;
  }
}
```

# void BPTree::printfValues() fonksiyonu B+ ağacı içindeki tüm string değerleri yazdırır.

```
void BPTree::printValues() {
  // Hedef: fonksiyon, B+ içerisindeki bütün string değerleri yazdırmayı hedefler
```

```
  node *t = this->head;
  while (!t->isLeaf) {
    t = ((NonLeafNode *)t)->Children[0];
  }
  while (t != NULL) {
    for (int i = 0; i < t->Keys.size(); ++i) {
      cout << ((LeafNode *)t)->Value[i] << endl;
    }
    t = ((LeafNode *)t)->Next;
  }
}
```

# void BPTree::remove(double key) fonksiyonu girilen anahtarın düğümünü bulup siler.

```
void BPTree::remove(double key) {
  node *t = this->head;

  // Girilen anahtarın düğümünü bul
  while (!t->isLeaf) { // Yerini bul
    int flag = 0;
    for (int i = 0; i < t->Keys.size(); ++i) {
      if (t->Keys[i] >= key) {
        t = ((NonLeafNode *)t)->Children[i];
        flag = 1;
        break;
      }
    }
    if (!flag) {
      t = ((NonLeafNode *)t)->Children[t->Keys.size()];
    }
  }

  // anahtarı ve değerini sil
  int flag = 0;
  for (int i = 0; i < t->Keys.size(); ++i) {
    if (t->Keys[i] == key) {
      t->Keys.erase(t->Keys.begin() + i);
      ((LeafNode *)t)->Value.erase(((LeafNode *)t)->Value.begin() + i);
      flag = 1;
```

```
    break;
  }
 }
 // bulunamazsa
 if (!flag) {
  cout << "Silmek istediğiniz anahtar yoktur!" << endl;
  return;
 }
        // ayarlama sadece gerekenden daha az düğüm olduğunda gerekir
 if (((LeafNode *)t)->Value.size() < ceil((n + 1) / 2) && t->Parent != NULL) {

  node *Rsibling;
  node *Lsibling;
  Rsibling = Lsibling = NULL;

  // iç düğüm
  int Child_num = -1;

  // anahtar dizisinin içerisinde, anahtarın pozisyonunu bul
  for (int i = 0; i < ((NonLeafNode *)t->Parent)->Children.size(); ++i) {
   if (((NonLeafNode *)t->Parent)->Children[i] == t) {
    Child_num = i;
    break;
   }
  }

  // bulunanın solu SOL KARDEŞ
  if (Child_num - 1 >= 0) {
   Lsibling = ((NonLeafNode *)t->Parent)->Children[Child_num - 1];
  }
  // bulunanın sağı SAĞ KARDEŞ
  if (Child_num + 1 < ((NonLeafNode *)t->Parent)->Children.size()) {
   Rsibling = ((NonLeafNode *)t->Parent)->Children[Child_num + 1];
  }

  // kardeşler şimdi biliniyor

  // eğer sağ kardeş yeterli düğüme sahipse, buradan (sağdan) değer götür
  if (Rsibling != NULL && ((LeafNode *)Rsibling)->Value.size() - 1 >= ceil((n + 1) / 2)) {

   // sağ kardeşin ilk anahtarını al ve bu anahtarı şu anki düğüme ekle
   t->Keys.push_back(Rsibling->Keys.front());
   ((LeafNode *)t)->Value.push_back(((LeafNode *)Rsibling)->Value.front());
```

```
    // bu anahtarı sağ çocuktan sil
    Rsibling->Keys.erase(Rsibling->Keys.begin());
    ((LeafNode *)Rsibling)->Value.erase(((LeafNode *)Rsibling)->Value.begin());

    // bu değişikliklerin ayrıca güncel düğümün ebeveynine yansıtılması gerekir
    t->Parent->Keys[Child_num] = t->Keys.back();
    return;
   }

  // eğer sağ kardeş yeterli düğüme sahipse, buradan(soldan) götür
  else if (Lsibling != NULL && ((LeafNode *)Lsibling)->Value.size() - 1 >= ceil((n + 1) /
2)) {

    // sol kardeşin son anahtarını al ve şu anki düğüme ekle
    t->Keys.insert(t->Keys.begin(), Lsibling->Keys.back());
    ((LeafNode *)t)->Value.insert(((LeafNode *)t)->Value.begin(),
                  ((LeafNode *)Lsibling)->Value.back());

    // bu anahtarı sol çocuktan sil
    Lsibling->Keys.erase(Lsibling->Keys.end() - 1);
    ((LeafNode *)Lsibling) ->Value.erase(((LeafNode *)Lsibling)->Value.end() - 1);

    // bu değişiklikler ayrıca şu anki düğümün ebeveynine yansıtılması gerekir.
    t->Parent->Keys[Child_num - 1] = Lsibling->Keys.back();
    return;
   }

  // kardeşler yardım edemezse, güçlendirmek için ayrılması gerekir
  else {
    // sağ çocuk gerekenden daha azsa
    if (Rsibling != NULL && ((LeafNode *)Rsibling)->Value.size() - 1 < ceil((n + 1) / 2)) {

      // sağ kardeşin tüm anahtarlarını şu anki düğüme böl
      t->Keys.insert(t->Keys.end(), Rsibling->Keys.begin(), Rsibling->Keys.end());
      // ve ayrıca değerleri
      ((LeafNode *)t) ->Value.insert(((LeafNode *)t)->Value.end(),
                    ((LeafNode *)Rsibling)->Value.begin(),
                    ((LeafNode *)Rsibling)->Value.end());

      // şimdi sağ kardeşin silinmesi gerekli

      // current->next = right sibling -> next yap
      ((LeafNode *)t)->Next = ((LeafNode *)Rsibling)->Next;
```

```
                // şimdi güvenilir bir şekilde sağ kardeşi silebiliriz


        // ilk olarak değişiklikleri onun ilk ebeveynine yansıt

        t->Parent->Keys.erase(t->Parent->Keys.begin() + Child_num);
        // bütün sağ kardeş düğümünü sil
        ((NonLeafNode *)t->Parent)->Children.erase(((NonLeafNode *)t->Parent)-
>Children.begin() + Child_num + 1);

        // şu anki pointer yerinde (sağ kardeşin solunda) ve sağ kardeş silindi
      }
      // sağ kardeş istenilenden az
      else if (Lsibling != NULL && ((LeafNode *)Lsibling)->Value.size() - 1 < ceil((n + 1) /
2)) {

        // sol kardeşin bütün anahtarlarını şu anki düğüme böl
        Lsibling->Keys.insert(Lsibling->Keys.end(), t->Keys.begin(), t->Keys.end());

        // ayrıca değerlerini de
        ((LeafNode *)Lsibling) ->Value.insert(((LeafNode *)Lsibling)->Value.begin(),
                        ((LeafNode *)t)->Value.begin(),
                        ((LeafNode *)t)->Value.end());


        // şimdi sol kardeşin silinmesi gerekli

        // left sibling -> next = current->next yap

        ((LeafNode *)Lsibling)->Next = ((LeafNode *)t)->Next;

        // şimdi sol kardei güvenilir bir şekilde silebiliriz

        // ilk olarak değişiklikler onun ebeveynine yansıtılır
        t->Parent->Keys.erase(t->Parent->Keys.begin() + Child_num - 1);

        // bütün sol kardeş düğümünü sil
        ((NonLeafNode *)t->Parent) ->Children.erase(((NonLeafNode *)t->Parent)-
>Children.begin() + Child_num);

        // şu anki pointer orijinal sola değiştirildi ve şu anki silindi
        t = Lsibling;
      }
```

```
    // şu anki düğüm süreci başa erişmiyorsa, tekrarlı bir şekilde bütün düğümlerin arasında işlemden geçir
    while (t->Parent != this->head) {

      Rsibling = Lsibling = NULL;
      // işlemi başlatmak için t yi t->parent e taşı
      t = t->Parent;

      // eğer yeterli çocuğa sahipse, işlenecek bir şey yok
      if (((NonLeafNode *)t)->Children.size() >= floor((n + 2) / 2)) {
        return;
      }

      //şu anki düğümü bul ve düğümün pozisyonunu child_num'da tut
      int Child_num = -1;
      for (int i = 0; i < ((NonLeafNode *)t->Parent)->Children.size(); ++i) {
        if (((NonLeafNode *)t->Parent)->Children[i] == t) {
          Child_num = i;
          break;
        }
      }

      // şu ankinin solu
      if (Child_num - 1 >= 0) {
        Lsibling = ((NonLeafNode *)t->Parent)->Children[Child_num - 1];
      }

      // şu ankinin sağı
      if (Child_num + 1 < ((NonLeafNode *)t->Parent)->Children.size()) {
        Rsibling = ((NonLeafNode *)t->Parent)->Children[Child_num + 1];
      }

      // eğer sağ kardeş yeterli düğümlere sahipse, buradan al
      if (Rsibling != NULL && ((NonLeafNode *)Rsibling)->Children.size() - 1 >= floor((n + 2) / 2)) {

        // sağ kardeşten ilk çocuğu al ve yukarıdakine benzer olarak işlemleri yap
        ((NonLeafNode *)t) ->Children.push_back(((NonLeafNode *)Rsibling)->Children.front());
        t->Keys.push_back(t->Parent->Keys[Child_num]);
        t->Parent->Keys[Child_num] = Rsibling->Keys.front();
        ((NonLeafNode *)Rsibling) ->Children.erase(((NonLeafNode *)Rsibling)->Children.begin());
        Rsibling->Keys.erase(Rsibling->Keys.begin());
```

```
        ((NonLeafNode *)t)->Children.back()->Parent = t;
        return;
      }

      // eğer sol kardeş yeterli düğümlere sahipse
      else if (Lsibling != NULL && ((NonLeafNode *)Lsibling)->Children.size() - 1 >=
floor((n + 2) / 2)) {

        // sol kardeşin son çocuğunu al ve yukarıdakine benzer olarak işlemleri yap
        ((NonLeafNode *)t) ->Children.insert(((NonLeafNode *)t)->Children.begin(),
((NonLeafNode *)Lsibling)->Children.back());
        t->Keys.insert(t->Keys.begin(), t->Parent->Keys[Child_num - 1]);
        t->Parent->Keys[Child_num] = Lsibling->Keys.back();
        ((NonLeafNode *)Lsibling) ->Children.erase(((NonLeafNode *)Lsibling)-
>Children.end() - 1);
        Lsibling->Keys.erase(Lsibling->Keys.end() - 1);
        ((NonLeafNode *)t)->Children.front()->Parent = t;
        return;
      }

      // eğer sağ tarafında yeterli değilse, bölünecek
      else if (Rsibling != NULL && ((NonLeafNode *)Rsibling)->Children.size() - 1 <
floor((n + 2) / 2)) {
        ((NonLeafNode *)Rsibling) ->Children.insert(((NonLeafNode *)Rsibling)-
>Children.begin(),
                        ((NonLeafNode *)t)->Children.begin(),
                        ((NonLeafNode *)t)->Children.end());
        Rsibling->Keys.insert(Rsibling->Keys.begin(),
                        t->Parent->Keys[Child_num]);
        Rsibling->Keys.insert(Rsibling->Keys.begin(), t->Keys.begin(),
                        t->Keys.end());
        for (int i = 0; i < ((NonLeafNode *)t)->Children.size(); ++i) {
          ((NonLeafNode *)t)->Children[i]->Parent = Rsibling;
        }
        t->Parent->Keys.erase(t->Parent->Keys.begin() + Child_num);
        ((NonLeafNode *)t->Parent)
           ->Children.erase(((NonLeafNode *)t->Parent)->Children.begin() +
                        Child_num);
        t = Rsibling;
      }

      // eğer sol tarafında yeterli değilse, bölünecek

      else if (Lsibling != NULL &&
```

```
            ((NonLeafNode *)Lsibling)->Children.size() - 1 <
                floor((n + 2) / 2)) {
        ((NonLeafNode *)Lsibling)
          ->Children.insert(((NonLeafNode *)Lsibling)->Children.end(),
                    ((NonLeafNode *)t)->Children.begin(),
                    ((NonLeafNode *)t)->Children.end());
        Lsibling->Keys.insert(Lsibling->Keys.end(),
                    t->Parent->Keys[Child_num - 1]);
        Lsibling->Keys.insert(Lsibling->Keys.end(), t->Keys.begin(),
                    t->Keys.end());
        for (int i = 0; i < ((NonLeafNode *)t)->Children.size(); ++i) {
          ((NonLeafNode *)t)->Children[i]->Parent = Lsibling;
        }
        t->Parent->Keys.erase(t->Parent->Keys.begin() + Child_num - 1);
        ((NonLeafNode *)t->Parent)
          ->Children.erase(((NonLeafNode *)t->Parent)->Children.begin() +
                    Child_num);
        t = Lsibling;
      }
    }

    // başa erişildi ve başın anahtarı yok
    if (t->Parent == this->head && this->head->Keys.size() == 0) {
      // şu anki düğüm baş oldu
      this->head = t;
      return;
    }
   }
  }
}
```

int main(void) fonksiyonu projemizin ana fonksiyonudur ve bu tüm diğer fonksiyonlar bu ana fonksiyonda yürütülür. Listemiz için gerekli olan düğümleri bu fonksiyonda ekledik ve bir do-while döngüsünün içine bir switch case yapısı kullandık kullanıcı 0 tuşuna basana kadar do-while döngüsü içinde bu işlemler yapılır. Burada bağlantılı liste ve b+ ağacı için istenilen işlemler kullanılabilir.

```
int main(void)
{
	BPTree t(3);
	List list;
	List2 list2;
	list.insertNode(0,1857,"Abdullah","Esen","BilgisayarMuhendisligi");
	list.insertNode(1,1977,"Deniz","Zeybek","BiyomedikalMuhendisligi");
	list.insertNode(2,8574,"Esra","Ayhan","BilgisayarMuhendisligi");
	list.insertNode(3,1050,"Ilayda","Kahraman","BilgisayarMuhendisligi");
	list.insertNode(4,2530,"Salih","Akin","BiyomedikalMuhendisligi");
	list.insertNode(5,6742,"Burak","Karatoprak","BilgisayarMuhendisligi");
	list.insertNode(6,5782,"Yeliz","Mutlu","BiyomedikalMuhendisligi");
	list.insertNode(7,7213,"Kerem","Pullu","BilgisayarMuhendisligi");
	list.insertNode(8,8430,"Feyza","Kuleli","BilgisayarMuhendisligi");
	list.insertNode(9,9474,"Rana","Gedik","BilgisayarMuhendisligi");
	list.insertNode(10,3567,"Feraye","Can","BiyomedikalMuhendisligi");
	list.insertNode(11,4678,"Faruk","Bilgili","BilgisayarMuhendisligi");
	list.insertNode(12,2563,"Selin","Demir","BilgisayarMuhendisligi");
	list.insertNode(13,1560,"Merve","Gunduz","BiyomedikalMuhendisligi");
	list.insertNode(14,5981,"Kadir","Bayri","BilgisayarMuhendisligi");
	list2.dersEkle(0,1857,"Abdullah","Esen","BilgisayarMuhendisligi","BilgisayarProgra
mlama");
	list2.dersEkle(1,1977,"Deniz","Zeybek","BiyomedikalMuhendisligi","BilgisayarProgr
amlama");
	list2.dersEkle(2,8574,"Esra","Ayhan","BilgisayarMuhendisligi","BilgisayarProgramla
ma");
	list2.dersEkle(3,1050,"Ilayda","Kahraman","BilgisayarMuhendisligi","VeritabaniYon
etimSistemleri");
	list2.dersEkle(4,2530,"Salih","Akin","BiyomedikalMuhendisligi","BilgisayarPrograml
ama");
	list2.dersEkle(5,6742,"Burak","Karatoprak","BilgisayarMuhendisligi","VeritabaniYon
etimSistemleri");
	list2.dersEkle(6,5782,"Yeliz","Mutlu","BiyomedikalMuhendisligi","VeritabaniYoneti
mSistemleri");
	list2.dersEkle(7,7213,"Kerem","Pullu","BilgisayarMuhendisligi","BilgisayarPrograml
ama");
	list2.dersEkle(8,8430,"Feyza","Kuleli","BilgisayarMuhendisligi","BilgisayarPrograml
ama");
	list2.dersEkle(9,9474,"Rana","Gedik","BilgisayarMuhendisligi","VeritabaniYonetimS
istemleri");
	list2.dersEkle(10,3567,"Feraye","Can","BiyomedikalMuhendisligi","BilgisayarProgra
mlama");
	list2.dersEkle(11,4678,"Faruk","Bilgili","BilgisayarMuhendisligi","VeritabaniYoneti
mSistemleri");
```

```
	list2.dersEkle(12,2563,"Selin","Demir","BilgisayarMuhendisligi","VeritabaniYonetim
Sistemleri");
	list2.dersEkle(13,1560,"Merve","Gunduz","BiyomedikalMuhendisligi","BilgisayarPro
gramlama");
	list2.dersEkle(14,5981,"Kadir","Bayri","BilgisayarMuhendisligi","BilgisayarPrograml
ama");
	int
dizi[]={1857,1977,8574,1050,2530,6742,5782,7213,8430,9474,3567,4678,2563,1560,5981};
	string
dizgi[]={"Abdullah","Deniz","Esra","Ilayda","Salih","Burak","Yeliz","Kerem","Feyza","Ran
a","Feraye","Faruk","Selin","Merve","Kadir"};
	int index1=0;
	while(dizi[index1]!=dizi[15])
	{
		t.insert(dizi[index1],dizgi[index1]);
		index1++;
	}
	string satir;
	double numara,index=15;
	int sayi;
	do
	{
	cout<<"\n1-->Listeye Eleman Ekleme\n2-->Listeden Eleman Silme\n3-->Liste
Uzerinden Arama Yapma\n4-->Listeyi Goruntuleme"<<endl;
	cout<<"5-->Ogrenciye Ders ekleme\n6-->Ders Alan Ogrenci Kaydini Silme\n7--
>Ders Alan Ogrenciler Arasindan Arama Yapma"<<endl;
	cout<<"8-->Ders Alan Ogrencileri Goruntuleme\n9-->Sadece Programlama Dersini
Alan Ogrenci Listesini Goruntuleme\n10-->Programlama Dersi A Grubunu
Listeleme"<<endl;
	cout<<"11-->Programlama Dersi B Grubu Listeleme\n12-->Bilgisayar Programlama
Dersini Alan Ogrencileri Kumeleme Siralama Algoritmasina Gore Listeleme"<<endl;
	cout<<"13-->B+ Agacina Ogrenci Ekleme\n14-->B+ Agacindaki Ogrenciyi
Bulma\n15-->B+ Agacindaki Numaralari Yazma ve Siralama"<<endl;
	cout<<"16-->B+ Agacindaki Ogrencileri Yazma ve Siralama\n17-->B+ Agacindaki
Ogrenciyi Silme\n0-->Cikis Islemi"<<endl;
	cin>>sayi;
	switch(sayi)
	{
		case 1:
		{
			cout<<"Eklemek istediginiz ogrencinin numarasini, adini, soyadini ve
bolumunu giriniz. "<<endl;
			char *ad=new char[20];
			char *soyad=new char[20];
```

```
                    char *bolum=new char[30];
                    cin>>numara>>ad>>soyad>>bolum;
        list.insertNode(index,numara,ad,soyad,bolum);
        index++;
        cout<<ad<<" isimli ogrenci dugume eklendi"<<endl;
                    break;
            }
            case 2:
            {
                    cout<<"Silmek istediginiz ogrencinin numarasini giriniz."<<endl;
                    cin>>numara;
                    list.deleteNode(numara);
                    cout<<numara<<" numarali ogrencinin kaydi silindi."<<endl;
                    break;
            }
            case 3:
            {
                    cout<<"Aramak istediginiz ogrencinin numarasini giriniz. "<<endl;
                    cin>>numara;
                    cout<<"Numarasini girdiginiz ogrencinin index\'i :
"<<list.findNode(numara)<<endl;
                    break;
            }
            case 4:
            {
                    list.displayList();
                    break;
            }
            case 5:
            {
                    index=15;
                    char *adi=new char[20];
                    char *soyadi=new char[20];
                    char *bolumu=new char[20];
                    char *alinanDers=new char[30];
                    cout<<"Ders eklemek istediginiz ogrencinin numarasini, adini,
soyadini, bolumunu ve hangi dersi eklemek istediginizi giriniz."<<endl;
                    cin>>numara>>adi>>soyadi>>bolumu>>alinanDers;
                    list.insertNode(index,numara,adi,soyadi,bolumu);
                    list2.dersEkle(index,numara,adi,soyadi,bolumu,alinanDers);
                    index++;
                    cout<<"Ogrencinin bilgileri ve alacagi ders dugume eklendi"<<endl;
                    break;
            }
```

```
case 6:
{
	cout<<"Silmek istediginiz ogrencinin numarasini giriniz."<<endl;
	cin>>numara;
	list.deleteNode(numara);
	list2.dersSil(numara);
	cout<<numara<<" numarali ogrencinin kaydi silindi"<<endl;
	break;
}
case 7:
{
	cout<<"Aramak istediginiz ogrencinin numarasini giriniz."<<endl;
	cin>>numara;
	cout<<list2.dersArama(numara)<<endl;
	break;
}
case 8:
{
	list2.displayList2();
	break;
}
case 9:
{
	list2.sadeceProgramlamaDersi();
	break;
}
case 10:
{
	list2.programlamaDersiAGrubu();
	break;
}
case 11:
{
	list2.programlamaDersiBGrubu();
	break;
}
case 12:
{
	int n;
	int dizimiz[10];
	list2.kumelemeSiralama(dizimiz,n);
	break;
}
case 13:
```

```
		{
			double k;
			string s;
			cout<<"Numarayi ve Isimi Giriniz "<<endl;
			cin>>k>>s;
			t.insert(k,s);
			cout<<"B+ agacina eleman eklendi"<<endl;
			break;
		}
		case 14:
		{
			double x;
		cout<<"Aranacak ogrencinin numarasini giriniz: ";
		cin>>x;
		cout<<t.find(x)<<endl;
			break;
		}
		case 15:
		{
			t.printKeys();
			break;
		}
		case 16:
		{
			t.printValues();
			break;
		}
		case 17:
		{
			double x;
		cout<<"Silinecek numarayi giriniz: "<<endl;
		cin>>x;
		t.remove(x);
		cout<<x<<" numarali anahtar b+ agacindan silindi."<<endl;
			break;
		}
	}
}while(sayi);



return 0;
}
```