Veri Yapıları

Hazır Veri Yapısı Kütüphaneleri

Hazır Veri Yapısı Kütüphaneleri

- Birçok programlama dilleri (C++, Java vb.), hazır veri yapıları ve algoritmalar içermektedirler.

- Programcılar, bu hazır veri yapılarını, kullanacakları veri yapılarının ekleme/silme vb. kodlarından bağımsız olarak, istedikleri problemleri çözmekte rahatça kullanabilirler.

- C++ da hazır veri yapıları Standart Şablon Kütüphanesi 'nde (Standard Template Library = STL) bulunurlar.

Hazır Veri Yapısı Kütüphaneleri

- STL'in ana fikri veri yapılarının ekle/sil, vb. gibi tekrarlayan kodlarının hazır hale getirilmesi ve programcıya kolay kullanılabilir bir arayüz sunulmasıdır.

- Örneğin bir tamsayı yığını kullanmak isteyen bir programcı, aşağıdaki kodu yazarak basit bir şekilde bu yığını tanımlayabilir:

```
stack <int>  y;
```

- Açı parantezleri < > şablon parametrelerini belirtmek için kullanılır.

- Daha sonra STL'nin bu veri yapısı için sağladığı metodları (fonksiyon) kullanarak istediği işlemleri gerçekleştirir:

```
y.push(3);
```

Genel olarak Template tanımlama

- Şablonlar (templates=containers) parametereli veri türleridir.
- Programcı kendi fonksiyon, struct, veya class'ını template olarak tanımladığında farklı veri türleri ile çalışma olanağı getirir.

- Deklarasyon için genel sentaks:

şablon_ismi <veri_türü> değişken_ismi;

- Örnekler:

```
stack <int>          s1;
stack <float>        s2;
stack <char *>       s3;
stack <int *>        s4;
stack <struct Ogrenci>  s5;
```

Örnek: Template struct tanımlama

```
template  <typename   veri_turu>
struct Sablon {
  veri_turu   veri;
  void yaz() { cout << veri << endl;}
};
```

```
int main() {
  Sablon <int>        a;
  Sablon <float>    b;
  Sablon <char *>  c;

  a.veri = 5;
  a.yaz();

  b.veri = 3.14;
  b.yaz();

  c.veri = "Ahmet";
  c.yaz();

  return 0;
}
```

Hazır Veri Yapısı Kütüphaneleri

- Kaplar (containers) nesne-tabanlı programlamada kullanılan bir kavramdır.

- STL kapları farklı kütüphaneler (C++ header dosyaları) içinde tanımlıdırlar.

- Bir kabı kullanmak için ilgili kütüphane #include direktifi ile program koduna eklenmelidir.

- Örnek: Yandaki header dosyaları C:\Dev-Cpp\Include\C++\3.4.2\ isimli klasör altında yer almaktadır.

```
#include  <vector>
#include  <list>
#include <deque>
#include <queue>
#include <stack>
#include <map>
#include <set>
#include <valarray>
#include <bitset>
#include <iterator>
```

STL'de bulunan veri yapıları 3 grup altında toplanabilir:

- **Ardışıl kaplar *(sequence containers)***
  (Vektör, bağlantılı liste gibi lineer veri yapılarıdır)
  - Vector
  - List
  - Double-Ended Queue

- **Kap Uyarlayıcıları *(container adapters)***
  (Lineer veri yapılarının, bazı özellikleri kısıtlanmış halleridir)
  - Stack
  - Queue
  - Priority Queue

- **Birlik Kapları *(associative containers)***
  (Elemanlarına arama anahtarları kullanarak doğrudan erişim sağlayan veri yapılarıdır)
  - Set , Multiset ,Bitset
  - Map , Multimap

Ardışıl Kaplar

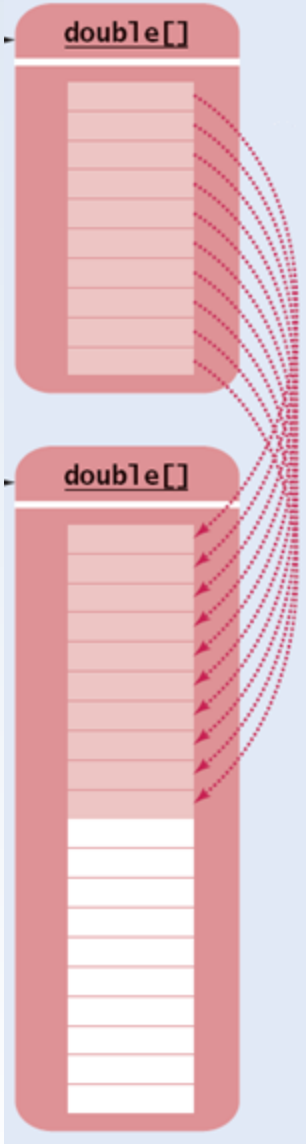
Ardışıl Kaplar

- C++'da 3 adet ardışıl kap bulunur:
  - `vector` - normal diziye göre daha kullanışlı bir dizi yapısıdır
  - `list` - bağlantılı liste
  - `deque` - double ended queue (bir çeşit dizi yapısı)

- Ardışıl kaplardaki ortak metodlar (fonksiyonlar)
  - `front` → ilk elemanın referansını döndürür
  - `back` → son elemanın referansını döndürür
  - `push_back` → sona bir eleman ekler
  - `pop_back` → son elemanı çıkartır

Vector

- Güvenilir ve gelişmiş bir dizi gerçeklemesidir.
- Normal dizilere benzer şekilde, bellekte ard arda gelen elemanlardan oluşur. [] operatörü kullanılabilir. Verilere hızlı erişim sağlanır.
- Boyutunu dinamik olarak arttırabilir:
  - Daha büyük boyutlu bir diziye ihtiyaç duyulduğunda, bellekten daha büyük bir alan alınarak, eski dizinin içeriği bu alana otomatik olarak kopyalanır.
  - Yeni kapasite 2'nin katları şeklinde otomatik olarak artar. (2, 4, 8, 16, 32, 64, 128, 256, . . .)
  - Eski dizi için bellekten alınan alan işletim sistemine otomatik olarak geri iade edilir.



Vector Örneği 1: (int)

```
#include <iostream>
#include <stdlib.h>
#include <vector>
using namespace std;
int main() {
   vector <int>  A;
   int i;

   // Add data to vector:
   for (i=0;i<5;i++) A.push_back( (i+1)*10 );
   // Array-wise printing:
   for (i=0;i<5;i++) cout<<A[i]<<" ";

   int eski_boyut = A.size();
   int eski_kapasite = A.capacity();

   cout<<endl<<"Vektorun boyutu="
       << eski_boyut <<endl;
   cout<<"Vektorun kapasitesi="
        << eski_kapasite <<endl;

   cout<<endl<<"Vektore kapasitesinden
        fazla yeni veriler ekleniyor"<<endl;
   for (i=eski_boyut;i<2*eski_kapasite;i++)
        A.push_back( (i+1)*10 );

   cout<<"Vektorun yeni boyutu="
        << A.size() <<endl;
   cout<<"Vektorun yeni kapasitesi="
       << A.capacity() <<endl;

   cout<<endl<<"Vektore bir veri daha
   ekleniyor"<<endl;
   A.push_back( (i+1)*10 );

   cout<<"Vektorun son boyutu="
        << A.size() <<endl;
   cout<<"Vektorun son kapasitesi="
        << A.capacity() <<endl<<endl;

   for (i=0;i<A.size();i++) cout<<A[i]<<" ";

    // Finally clear the vector:
   while (!A.empty())
          A.pop_back();

   system("pause");
   return 0;
}
```

EKRAN ÇIKTISI:

```
0 1 2 3 4
Vektorun boyutu=5
Vektorun kapasitesi=8

Vektore kapasitesinden fazla veri ekleniyor
Vektorun boyutu=16
Vektorun kapasitesi=16

Vektore bir veri ekleniyor
Vektorun boyutu=17
Vektorun kapasitesi=32

17 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

Vector Örneği 2: (char *)

```
#include <iostream>
#include <stdlib.h>
#include <vector>
using namespace std;

int main() {
    char * Ogr[] = {"Bülent", "Metin", "Fatih"};

   vector <  char * >  A;
   int i;

   // Copy data from array Ogr to vector A:
   for (i=0;i<3;i++) A.push_back( Ogr[i] );

   // Array-wise printing:
   for (i=0;i<3;i++)
        cout<< A[i] << endl;

   system("pause");
   return 0;
}
```

EKRAN ÇIKTISI:

```
Bülent
Metin
Fatih
```

Alternatif yöntem:
(Constructor initialization)

```
vector<char *>  A(Ogr, Ogr+3);
```

Vector Örneği 3: (struct)

```
#include <iostream>
#include <stdlib.h>
#include <vector>
using namespace std;

typedef struct Ogrenci {
     int num;
     char isim[20];
};

Ogrenci  Ogr[] = {
   {4369, "Bülent"},
   {7026, "Metin"},
   {2184, "Fatih"}
};

int main() {
   vector <Ogrenci>  A;
   int i;
```

```
// Copy data from isimler to vector A:
   for (i=0;i<3;i++) A.push_back( Ogr[i] );

   // Array-wise printing:
   for (i=0;i<3;i++)
        cout<< A[i].num << " " << A[i].isim << endl;

   system("pause");
   return 0;
}
```

EKRAN ÇIKTISI:

```
4369 Bülent
7026 Metin
2184 Fatih
```

Vector - Üye Fonksiyonlar

- Ortak metodların yanı sıra, kullanılabilecek daha birçok metodu vardır:

- Kapasite ile ilgili metodlar: size, max_size, resize, capacity, empty, reserve

- Elemanlara erişim ile ilgili metodlar: [] , at, front, back

- Vektörde değişikliğe neden olan metodlar: assign, push_back, pop_back, insert, erase, swap, clear

Kaplara ait metodların kullanım detayları için STL dökümantasyonlarına bakılabilir.

List

- List STL kabı bağlantılı liste için verimli bir ekleme çıkarma gerçeklemesi sunar.
- Eğer ekleme ve çıkarma işemleri genel olarak listenin sonuna yapılıyorsa "deque" kabı daha uygundur.
- List kabı çift yönlü bağlantılı liste olarak gerçeklenmiştir. Dolayısıyla liste üzerinde hızlı bir şekilde çift yönde ilerlenebilir.
- Elemanlara erişim:
  - İndis operatörü [] ve at() fonksiyonu list'lerde kullanılmaz!
  - Bunların yerine, bir iterator nesnesi tanımlanarak liste dolaşılabilir.

List Örneği (Iterator kullarak dolaşma)

```
#include <iostream>
#include <stdlib.h>
#include <list>
#include <iterator>

using namespace std;

int main() {
    list <int> A;

   // Add data to vector:
   for(int i=0;i<5;i++)
          A.push_back( (i+1)*10 );

   // Traversing and printing with an iterator:
   list  <int>::iterator  tara;
   for (tara=A.begin(); tara != A.end(); tara++)
          cout << *tara << endl;

  system("pause");
  return 0;
}
```

```
EKRAN ÇIKTISI:

10
20
30
40
50
```

Yineleyiciler (İteratörler)

- Veri Yapıları için C++'ın hazır kütüphaneleri kullanıldığında, bazı veri yapılarının elemanları arasında dolaşmak için iteratör adı verilen yapılar kullanılır.

- İteratörler işaretçilere benzetilebilir.
- Bir bağlantılı liste üzerinde dolaşılırken işaretçiler kullanılır. Örneğin:

```
dugum *ptr=bas;
while (ptr!=NULL)
        ptr = ptr->sonraki;
```

- Benzer şekilde, STL'nin list yapısı üzerinde dolaşılırken de iteratörler kullanılır.

İteratörler

- İteratörler aşağıdakiler için kullanılabilir:
  - Ardışıl kaplar (Vector, List, Deque)
  - Birlik kapları (Map, Set, Multimap, Multiset, Bitset)

- İteratörleri, Kap adaptörleri (Stack, Queue, Priority Queue) üzerinde kullanmak mümkün değildir. Çünkü bunların elemanlarına nasıl erişileceği adaptör yapısı ile önceden tanımlanmış durumdadır.
- Örnek: Bir yığının elemanları arasında dolaşmak mümkün değildir. Çünkü yığına sadece tek noktadan erişim mümkündür, bu da yığın yapısının kendi metodlarıyla gerçekleştirilir.

İteratör tipleri	Yön	İşlem
iterator	ileri	Okuma/yazma
const_iterator	ileri	Okuma
reverse_iterator	geri	Okuma/yazma
reverse_const_iterator	geri	Okuma

Reverse İteratör Örneği

```
#include <iostream>
#include <list>
#include <iterator>

using namespace std;

int main()
{
  list <int> A;
  for(int i=0;i<5;i++)
       A.push_back(i+1)*10);

  list <int>::reverse_iterator tara;
  for (tara = A.rbegin(); tara != A.rend(); tara++)
       cout << *tara << endl;

  return 0;
}
```

```
EKRAN ÇIKTISI:

50
40
30
20
10
```

Deque (Double-ended queue)

- vector ve liste yapısının iyi yönlerini bir arada kullanabilmek için tasarlanmış bir yapıdır.
- Vector'deki gibi elemanlara [] indeks ile hızlı erişim sağlar (dizi üzerinde gerçeklenmiştir.)
- Aynı zamanda listedeki gibi önüne ve arkasına hızlı eleman ekleme ve çıkarma gibi işlemler verimli bir şekilde gerçeklenmiştir. (Listede ortaya ekleme de yapmak da mümkündür.)
- Bir kuyruk yapısı için varsayılan (default) veri alt yapısıdır.
- Vectordeki temel operasyonların yanısıra, push_front ve pop_front fonksiyonları içerir.
- Dikkat! Push_back bütün kaplar için vardır ancak push_front sadece list ve deque'da vardır.

Kap Uyarlayıcıları

Kap uyarlayıcıları

- C++'da 3 adet kap uyarlayıcısı bulunur:
  - `stack`- yığın
  - `queue` - kuyruk
  - `priority_queue` - öncelik kuyruğu

Stack - Yığın üye fonksiyonlar

- empty true → bosmu()
- pop → cikar() metodumuzdan farkı geri dönüş değeri olmamasıdır. void pop(); Yığının üstündeki elemanı çıkarır ancak elemana erişim sağlamaz.
- push → ekle(…)
- size → yığındaki eleman sayısını döndürür
- top → yığındaki en üst elemana erişim sağlar. Ancak eleman yığından çıkarılmaz.

Stack Örneği

```
#include <iostream>
#include <stdlib.h>
#include <stack>
using namespace std;

int main() {
  stack <int> yigin;

  for ( int i=1; i <= 5; i++)
      yigin.push(i*10); // Yığına ekle

 // Poping and printing:
  while (!yigin.empty() )
  {
     cout << "Yiginin tepe elemani : "
          << yigin.top() << endl;
     yigin.pop(); // Yığından çıkart
  }
return 0; }
```

```
SCREEN  OUTPUT:

Yiginin tepe elemani : 50
Yiginin tepe elemani : 40
Yiginin tepe elemani : 30
Yiginin tepe elemani : 20
Yiginin tepe elemani : 10
```

Queue – Kuyruk üye fonksiyonlar

- back→ kuyruğun son elemanına erişim sağlar
- empty → bosmu()
- front → kuyruğun ilk elemanına erişim sağlar
- pop → cikar() metodumuzdan farkı geri dönüş değeri olmamasıdır. void pop(); kuyruğun ilk elemanı çıkarır ancak elemana erişim sağlamaz.
- push → ekle()
- size kuyruktaki eleman sayısını döndürür

Queue Örneği

```
#include <iostream>
#include <stdlib.h>
#include <queue>
using namespace std;

int main(){
   queue <int> kuyruk;

   for(int i=1;i<=5;i++)
         kuyruk.push(i*10); // Kuyruğa ekle

    // Poping and printing:
   while(!kuyruk.empty()){
         cout << "Kuyrugun front elemani : "
              << kuyruk.front() << endl;
         kuyruk.pop();
   }

  system("pause");
  return 0;
}
```

```
SCREEN  OUTPUT:

Kuyrugun front elemani : 10
Kuyrugun front elemani : 20
Kuyrugun front elemani : 30
Kuyrugun front elemani : 40
Kuyrugun front elemani : 50
```

Priority Queue - Öncelik kuyruğu üye fonksiyonlar

- Default olarak kuyruk büyükten küçüğe sıralı tutulur.
- Büyük olan önce işlenir.
- Kendi öncelik türünüzü belirlemek mümkündür, ancak daha ileri programlama bilgilerine ihtiyaç vardır.

- empty
- size
- top ilk sıradakine erişim sağlar
- push önceliğe göre eleman ekler
- pop ilk sıradakini çıkarır.

Priority_queue Örneği

```
#include <iostream>
#include <stdlib.h>
#include <queue>
using namespace std;

int main(){
   priority_queue <int> A;
   // Öncelikli Kuyruga ekle:
   A.push(30);
   A.push(100);
   A.push(25);
   A.push(40);

  // Elements in this queue has been ordered.
  // Poping and printing:
  while (!A.empty()) {
        cout << " " << A.top() << endl;
        A.pop();
  }
system("pause");
  return 0;
}
```

```
SCREEN  OUTPUT:

100
40
30
25
```

Sıralama kriterinin değiştirilmesi

- Default olarak sıralama rezerve kelimesi «less» ‘dir.
- «greater» yazarak sıralama kriteri değiştirilebilir.

```
priority_queue <int, vector<int>, greater<int> >  A;
```

```
EKRAN ÇIKTISI:

25
30
40
100
```

Alternatif : Kap uyarlayıcısı gerçekleştirimleri

- Bir kap uyarlayıcısının avantajı, kullanıcının alt veri yapısını seçebilmesidir.
- Örneğin yığın yapısının default alt veri yapısı deque'dur. (Yani Stack, aslında Deque kullanılarak gerçeklenmiştir.)
- Eğer yığının deque yerine, bağlantılı liste üzerinde gerçeklenmiş halini kullanmak istersek:

```
stack <int> yigin;                    yerine
stack <int, list <int> > yigin;       yazılır
```

- Eğer vektör kullanılarak yapmak istenirse

```
stack <int, vector<int> > yigin;      yazılır
```

Stack Örneği (Bağlantılı liste kullanarak)

```
#include <iostream>
#include <stdlib.h>
#include <stack>
#include <list>

using namespace std;

int main()
{
    stack <int, list <int> > yigin;

   for(int i=1;i<=5;i++)
          yigin.push(i*10); // Yığına ekle

    // Poping and printing:
   while(!yigin.empty()){
          cout << yigin.top() << endl;
          yigin.pop();
   }

   system("pause");
   return 0;
}
```

```
SCREEN  OUTPUT:

50
40
30
20
10
```

Birlik Kapları

Birlik Kapları

- Anahtar veriler kullanarak elemanlara doğrudan erişim sağlarlar. Arama anahtarlarını sıralı şekilde tutarlar.

- 5 adet birlik kabı vardır:
  - Multiset – sadece anahtarları tutar, veri tekrarı mümkündür
  - Set – sadece anahtarları tutar, tekrar mümkün değildir
  - Bitset – bit işlemleri için kullanılan küme
  - Multimap – anahtarları ve bunlarla ilgili değerleri tutar, tekrar mümkündür.
  - Map – anahtarları ve bunlarla ilgili değerleri tutar, tekrar mümkün değildir.

- Ortak fonksiyonlar:
  - `find`, `lower_bound`, `upper_bound`, `count`

Map Örneği : Anahtar kullanımı

```
#include <iostream>
#include <stdlib.h>
#include <stack>
#include <list>

using namespace std;

int main()
{
    stack <int, list <int> > yigin;

   for(int i=1;i<=5;i++)
          yigin.push(i*10); // Yığına ekle

    // Poping and printing:
   while(!yigin.empty()){
          cout << yigin.top() << endl;
          yigin.pop();
   }

   system("pause");
   return 0;
}
```

```
EKRAN ÇIKTISI:

Carsamba 3
Cuma 5
Cumartesi 6
Pazar 7
Pazartesi 1
Persembe 4
Sali 2
```

Örnek : Kelime sıklıkları

İngilizce düz metin içeren bir dosyadaki kelimelerin kullanım sıklığını ölçmek için "map" veri yapısını kullanan bir program yazılacaktır.

Konular: STL ve sıralı erişilebilir dosya okuma (sequential access file)

Program çıktı olarak kelimeleri ve sıklıklarını gösterecektir.

Örnek 1: Map kullanımı

- Program dosyadan sırayla kelimeleri okur ve bir map yapısına ekler.
- Böylece her kelimenin sıklığı hesaplanır.

- Ekran çıktısı alfabetik olarak sıralı olur. :
  - Map veri yapısı daima anahtara göre sıralanır (bu örnekte kelimeler anahtardır) .
  - Değere göre sıralama yoktur (bu örnekte sıklıklar değerdir) .

Örnek 1: Map kullanımı

```
#include <iostream>
#include <stdlib.h>
#include <map>
#include <string>
using namespace std;

int main() {
   FILE *myfile= fopen( "english.txt", "r" );
   if(!myfile){
          cerr << "Dosya acilamadi" << endl;
          return -1;
   }
   char word[100];
   map <string,int> freq;
   while(!feof(myfile)){
          fscanf(myfile,"%s",word);

          freq[word]++;
   }
   fclose(myfile);
```

```
   map<string,int>::iterator tara;

   int count;

   for (tara=freq.begin();
        tara != freq.end();
        tara++)
   {
          cout << tara->first
               << "\t"
               << tara->second << endl;
   }

   system("pause");
   return 0;
}
```

Örnek 1: Map kullanımı

- Kelime sıklıkları alfabetik sırayla yazdırılır.

```
EKRAN ÇIKTISI:

a               10
above           1
academic        1
accepted        1
addition        1
adhere          1
aims            1
albeit          1
algorithms      9
all             5
along           1
already         1
also            3
an              5
analysts        1
and             23
any             2
applications    2
. . .
```

Örnek 2: Map ve Multimap birlikte kullanımı

- Kelimeleri sıklıklarına göre büyükten küçüğe göre göstyermek için, elde edilen map başka bir map'a atanır. (ikinci map'ta sıklıklar anahtar olmalıdır.)

- Aynı sıklık değerine sahip 1'den fazla kelime olabilceği için, veri yapısı multimap olmalıdır.

- Yeni multimap'de değerler default olarak küçükten büyüğe sıralo öolur.

- Ekran çıktısının sıklıklara göre büyükten küçüğe olması isteniyor. En yüksek sıklığa sahip ilk 10 kelimenin ekranda görüntülenmesi için veri yapısının en sondaki 10 kayıtı gösterilecektir.

Örnek 2: Map ve Multimap birlikte kullanımı

```
#include <iostream>
#include <stdlib.h>
#include <map>
#include <string>
#include <ctype.h>
using namespace std;

// Stringin tamamını küçük harfe çevirir.
void change_to_lower(char * s) {
 while (*s != NULL) {
    *s = tolower(*s);
    s++;
  }
}

int main() {
   FILE *myfile= fopen( "english.txt", "r" );
   if(!myfile){
          cerr << "Dosya acilamadi" << endl;
          return -1;
   }
   char word[100];
```

(devam)

```
map <string,int> freq; // İlk yazılan (string) anahtardır.

while(!feof(myfile)){
        fscanf(myfile,"%s",word);
        change_to_lower(word);
        freq[word]++;
}
fclose(myfile);

 // Değer alanları (sıklık) anahtar olacak şekilde
 // başka bir map'e atanma yapılır.

multimap <int,string > freq_rev;

map<string,int>::iterator tara1;

for(tara1=freq.begin();tara1!=freq.end();tara1++)
        freq_rev.insert(make_pair(tara1->second,tara1->first));
```

(devam)

```
 //En sondaki 10 kayıtı yazdır.

   multimap <int,string>::reverse_iterator tara2;
   int count;

   for (tara2=freq_rev.rbegin(),count=0;count<10;tara2++,count++)
          cout<<tara2->second<<" "<<tara2->first<<endl;

   system("pause");
   return 0;
}
```

Örnek 2: Map ve Multimap birlikte kullanımı

- Kelimeler sıklıklarına göre azalan sıradadır.

```
EKRAN ÇIKTISI:

the    31
to     26
and    23
of     20
in     18
you    16
is     14
a      10
...
```

Uygulamada Yapılacaklar

- STL örnekleri