به نام خدا

برنامه‌سازی پیشرفته

آرش شفیعی

ساختارداده‌ها و الگوریتم‌های استاندارد

ساختارداده‌های استاندارد

- در پیاده‌سازی سیستم‌های نرم‌افزاری در بیشتر مواقع به مجموعه‌ای از داده‌ها نیاز داریم که باید به نحوی ذخیره‌سازی کرده و بر روی آنها عملیات انجام دهیم.
- کلاسی را که یک مجموعه از داده‌ها را نگهداری می‌کند، معمولا یک ظرف [1] می‌نامیم.
- معمولا در یک برنامه باید ظرف‌هایی که نیاز داریم را با انواع توابع مورد نیازها پیاده‌سازی کنیم. در کتابخانهٔ استاندارد سی++ بسیاری از ساختارداده‌های معمول پیاده‌سازی شده‌اند.

[1] container

ساختارداده‌های استاندارد

- ظرف‌های استاندارد را می‌توان به چهار دستهٔ ظرف‌های ترتیبی [1]، ظرف‌های مبدل [2]، ظرف‌های رابطه‌ای [3] تقسیم کرد.

[1] sequence containers
[2] container adaptors
[3] associative containers

ساختارداده‌های استاندارد

- ظرف‌های ترتیبی به ظرف‌هایی گفته می‌شود که دسترسی به عناصر آنها به صورت ترتیبی صورت می‌گیرد.
- این ظرف‌ها شامل آرایه (`array`)، وکتور یا حامل (`vector`)، صف دوطرفه (`deque`)، و لیست یا لیست پیوندی (`list`) می‌شوند.
- آرایه ظرفی است که اندازهٔ آن در زمان اجرای برنامه غیرقابل تغییر است.
- وکتور آرایه‌ای است که اندازهٔ آن در زمان اجرا قابل افزایش و کاهش است و علاوه بر آن پیاده‌سازی آن برای دسترسی تصادفی به عناصر وکتور بهینه‌سازی شده است.
- صف‌دوطرفه یک صف است که از ابتدا و انتها می‌توان به عناصر آن دسترسی پیدا کرد. به علاوه دسترسی تصادفی به عناصر آن به صورت بهینه انجام می‌شود.
- لیست یک لیست پیوندی دوطرفه است.

ساختارداده‌های استاندارد

- ظرف‌های مبدل به ظرف‌هایی گفته می‌شود که یک سازوکار جدید برای پیاده‌سازی ظرف‌ها ارائه نمی‌کنند، بلکه تنها از ظرف‌های ترتیبی به عنوان زیرساخت استفاده کرده و امکانات و قابلیت‌های جدید به آنها اضافه می‌کنند.
- این ظرف‌ها شامل صف (`queue`)، پشته (`stack`)، و صف اولویت (`priority_queue`) می‌شوند.

ساختارداده‌های استاندارد

- پشته ساختارداده‌ای است که در آن آخرین عنصری که وارد می‌شود، اولین عنصری است که خارج می‌شود. به عبارت دیگر برای اضافه کردن یک عنصر به پشته به بالای آن عنصری را اضافه و برای حذف از پشته آخرین عنصر اضافه شده به پشته اولین عنصری است که حذف می‌شود. پشته مفهوم آخرین–ورودی، اولین–خروجی [1] را پیاده‌سازی می‌کند.
- صف ساختار داده‌ای است که در آن اولین عنصری که وارد می‌شود، اولین عنصری است که خارج می‌شود. پس وقتی یک عنصر به صف وارد می‌شود در آخر صف قرار می‌گیرد و اولین عنصری که به صف وارد شده اولین عنصری است که خارج می‌شود. پشته مفهوم اولین–ورودی، اولین–خروجی [2] را پیاده‌سازی می‌کند.
- در یک صف اولویت، هر عنصر دارای یک اولویت است و عناصری که اولویت بیشتری دارند زودتر از صف خارج می‌شوند.

[1] last in، first out (LIFO)
[2] first in، first out (FIFO)

ساختارداده‌های استاندارد

- ظرف‌های رابطه‌ای به ظرف‌هایی گفته می‌شود دسترسی به عناصر آنها بر اساس یک رابطه یا یک نگاشت است.
- ظرف‌های رابطه‌ای شامل نگاشت (`map`)، مجموعه (`set`)، چندنگاشت (`multimap`)، چندمجموعه (`multiset`)، نگاشت نامرتب (`unordered_map`) و مجموعهٔ نامرتب (`unordered_set`) می‌شوند.
- نگاشت ظرفی است که توسط آن یک کلید به یک مقدار نسبت داده می‌شود، پس دسترسی به هر عنصر توسط کلید یکتای آن عنصر است. کلیدهای یک نگاشت به صورت مرتب در درون آن قرار گرفته‌اند.
- در یک مجموعه هر عنصر تنها یک بار تکرار می‌شود و دسترسی به هر عنصر مجموعه توسط یک کلید یکتا صورت می‌گیرد. عناصر یک مجموعه به صورت مرتب در آن قرار گرفته‌اند.

ساختارداده‌های استاندارد

- چندنگاشت، نگاشتی است که در آن کلید یکتا نیست و چند عنصر می‌توانند کلید یکسان داشته باشند. همچنین چندمجموعه، مجموعه‌ای است که در عناصر می‌توانند تکرار شوند.
- نگاشت و مجموعهٔ نامرتب شبیه نگاشت و مجموعه هستند با این تفاوت که عناصر درونی آنها مرتب‌نشده است.

ساختارداده‌های استاندارد

- هر ظرف شامل یک پیمایشگر [1] است. یک پیمایشگر شبیه یک اشاره‌گر است که به یک عنصر از یک ظرف اشاره می‌کند. پیمایشگرهای ظرف‌های مختلف به صورت‌های مختلف تعریف شده‌اند.
- همچنین برای هر ظرف الگوریتم‌های متفاوتی تعریف شده است که می‌توان از این الگوریتم‌ها به صورت بهینه برای عملیات متفاوت بر روی ظرف‌ها از جمله مرتب‌سازی و جستجو استفاده کرد.

[1] iterator

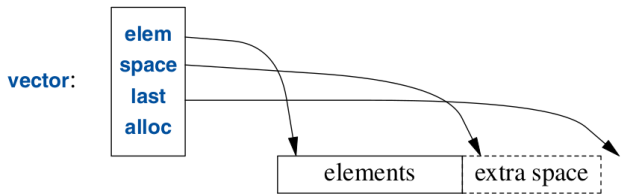
وکتورها

- وکتور یا حامل یکی از پراستفاده‌ترین ظرف‌هاست. یک وکتور دنباله‌ای از عناصر با یک نوع معین است. عناصر وکتور به صورت به‌هم‌پیوسته [1] مجاور یکدیگر در حافظه قرار گرفته‌اند.
- در یک وکتور تعدادی عنصر قرار گرفته و تعدادی فضای خالی برای عناصری که در آینده وارد وکتور خواهند شد وجود دارد. وقتی فضای خالی پر شد، وکتور مجددا فضای جدیدی در حافظه تخصیص می‌دهد.

[1] contiguous

وکتورها

- به عبارت دیگر یک وکتور شامل اندازه‌ای [1] است که تعداد عناصر درون آن را تعیین می‌کند و دارای ظرفیتی [2] است که حداکثر تعداد عناصری که می‌توانند درون آن قرار بگیرند را تعیین می کند. وقتی اندازه بیشتر به ظرفیت برسد، وکتور باید برای عناصر جدید فضای بیشتر تخصیص دهد و ظرفیت وکتور را افزایش دهد.



[1] size
[2] capacity

وکتورها

- وکتور توسط قالب پیاده‌سازی شده است و آن را می‌توان با استفاده از عناصری از هر نوع داده‌ای ساخت. در مقداردهی اولیه می‌توان اندازهٔ وکتور و مقدار پیش‌فرض عناصر آن را تعیین کرد.

```
vector<int> v1 = {1, 2, 3, 4}; // size is 4
vector<string> v2;  // size is 0
vector<Shape*> v3(23); // size is 23; initial element value: nullptr
vector<double> v4(32,9.9); // size is 32; initial element value: 9.9
```

- عملگر زیرنویس برای وکتور سربارگذاری شده است، بنابراین می‌توانیم به عناصر آن به صورت تصادفی دسترسی پیدا کنیم.

```
vector<int> myvector (10);   // 10 zero-initialized elements
vector<int>::size_type sz = myvector.size();
// assign some values:
for (unsigned i=0; i<sz; i++) myvector[i]=i;
```

وکتورها

- می‌توانیم وکتوری به صورت زیر بسازیم:

```
struct Entry { string name; int number; };

vector<Entry> phone_book = {
    {"Mr. X",123},
    {"Mrs. Y", 456}
};
```

- از آنجایی که عملگر درج را برای Entry تعریف کرده‌ایم می‌توانیم به صورت زیر عناصر وکتور را چاپ کنیم.

```
void print_book(const vector<Entry>& book) {
    for (int i = 0; i!=book.size(); ++i)
        cout << book[i] << '\n';
}
```

وکتورها

- همچنین توسط حلقهٔ تکرار بر روی دامنه می‌توانیم به صورت زیر به عناصر وکتور دسترسی پیدا کنیم.

```
۱ void print_book(const vector<Entry>& book) {
۲     for (const auto& x : book)
۳         cout << x << '\n';
۴ }
```

وکتورها

- یکی از توابع وکتور تابع push_back است که توسط آن می‌توان یک عنصر به وکتور افزود. با فرض اینکه عملگر استخراج برای نوع دادهٔ Entry تعریف شده باشد، می‌توانیم به صورت زیر عمل کنیم.

```
1 void input() {
2     for (Entry e; cin>>e; )
3         phone_book.push_back(e);
4 }
```

- تابع push_back به گونه‌ای طراحی شده است که تا وقتی که اندازهٔ وکتور به ظرفیت آن نرسیده است عنصر را به وکتور اضافه می‌کند و پس از اینکه اندازه به ظرفیت رسید، ظرفیت وکتور را می‌افزاید و همچنین ممکن است فضایی جدید در حافظه برای عناصر خود تخصیص دهد و عناصر در حافظهٔ قبلی را در حافظهٔ جدید کپی کند.

- با استفاده از تابع reserve() می‌توان فضایی را در حافظه تخصیص داد و بدین‌گونه از تخصیص مجدد حافظه جلوگیری کرد، اما باید دانست که وکتور از راه‌های اکتشافی یا هیوریستیک‌هایی استفاده می‌کند که توسط آن مقدار بهینهٔ ظرفیت را پیش‌بینی می‌کند.

وکتورها

- با استفاده از تابع pop_back() می‌توانیم عنصری را از یک وکتور حذف کنیم.

```
۱  vector<int> myvector;
۲  int sum (0);
۳  myvector.push_back (100);
۴  myvector.push_back (200);
۵  myvector.push_back (300);
۶
۷  while (!myvector.empty()) {
۸      sum+=myvector.back();
۹      myvector.pop_back();
۱۰ }
```

وکتورها

- دسترسی به عناصر خارج از محدوده در وکتور توسط عملگر زیرنویس [] خطایی تولید نمی‌کند.

```
vector<Entry> book;
int i = book[book.size()].number;
// i gets a random value
}
```

- وکتور در دسترسی خارج از محدوده استثنایی ارسال نمی‌کند، چرا که با پیاده‌سازی استثنا برای آن سربار اضافی تحمیل شده و از بهره‌وری وکتور کاسته می‌شود.

وکتورها

- دسترسی به عناصر خارج از محدوده در وکتور توسط عملگر زیرنویس [] خطایی تولید نمی‌کند.

```
vector<Entry> book;
int i = book[book.size()].number;
// i gets a random value
}
```

- از طرف دیگر می‌توان با استفاده از تابع at به اعضای یک وکتور دسترسی پیدا کرد که در صورت دسترسی خارج از محدوده این تابع یک استثنا ارسال می‌کند.

وکتورها

- می‌توانیم کلاسی تعریف کنیم که از کلاس وکتور ارث‌بری می‌کند و بدین ترتیب عملگر زیرنویس را سربارگذاری کرده به نحوی که دسترسی به عناصر استثنا ارسال کند.

```
template<typename T>
class Vec : public std::vector<T> {
public:
    // use the constructors from vector (under the name Vec)
    using vector<T>::vector;
    T& operator[](int i) { return vector<T>::at(i); } // range check
    const T& operator[](int i) const {
        return vector<T>::at(i); // range check const objects
    }
};
```

وکتورها

- می‌توانیم کلاسی تعریف کنیم که از کلاس وکتور ارث‌بری می‌کند و بدین ترتیب عملگر زیرنویس را سربارگذاری کرده به نحوی که دسترسی به عناصر استثنا ارسال کند.

```
۱ void checked(Vec<Entry>& book) {
۲     try {
۳         book[book.size()] = {"Joe",999999};
۴         // ...
۵     } catch (out_of_range&) {
۶         cerr << "range error\n";
۷     }
۸ }
```

وکتورها

- بهتر است همیشه در بدنهٔ اصلی برنامه همهٔ استثناها را مدیریت کنیم تا اگر تابعی یک استثنا را مدیریت نکرد با توقف برنامه روبرو نشویم.

```
int main() {
    // ...
    try {
        // code using Vec
    } catch (out_of_range&) {
        cerr << "range error\n";
    } catch (...) {
        cerr << "unknown exception thrown\n";
    }
    // ...
}
```

وکتور

- معمولا در استفاده از ظرف‌ها می‌خواهیم عناصر ظرف را پیمایش کنیم. برای چنین کاری معمولا از یک پیمایشگر [1] استفاده می‌کنیم.
- یک پیمایشگر شبیه یک اشاره‌گر است که به یکی از عناصر یک ظرف اشاره می‌کند و می‌توان با استفاده از آن عناصر را پیمایش کرد. هر ظرف برای خود به نحوی متفاوت پیمایشگری پیاده‌سازی کرده است ولی نحوهٔ استفاده از آنها یکسان است.

[1] iterator

وکتور

- هر ظرف در کتابخانهٔ استاندارد دو تابع `begin()` و `end()` فراهم کرده است. تابع `begin()` پیمایشگری به اولین عنصر ظرف و تابع `end()` پیمایشگری به عنصر ماقبل‌آخر ظرف بازمی‌گرداند.
- اگر `p` یک پیمایشگر باشد، `p*` مقدار عنصری است که آن پیمایشگر به آن اشاره می‌کند، و `p++` پیمایشگر را یک عنصر به جلو حرکت می‌دهد. همچنین اگر `p` به کلاسی اشاره کند که یکی از اعضای آن `m` است، آنگاه `p->m` نشان دهندهٔ آن عضو از کلاس است که معادل با `(*p).m` است.

```
1 vector<int> vec(10,100);
2 for (vector<int>::iterator it = vec.begin(); it != vec.end(); ++it)
3     cout << ' ' << *it;
```

وکتور

- در کلاس وکتور می‌توان پیمایشگر را به صورت زیر تعریف کرد.

```
template <typename T>
class Vector {
public:
    typedef T * iterator;
};
auto Vector<int>::iterator iter;
```

وکتورها

- همچنین با استفاده از پیمایشگرها می‌توانیم وکتور را مقداردهی اولیه کنیم.

```
// empty vector of ints
vector<int> first;
// four ints with value 100
vector<int> second (4,100);
// iterating through second
vector<int> third (second.begin(),second.end());
// a copy of third
vector<int> fourth (third);
// the iterator constructor can also be used to construct from arrays:
int myints[] = {16,2,77,29};
vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );
```

وکتورها

- از تابع insert می‌توانیم برای درج عناصری در میانهٔ وکتور استفاده کنیم.

```
۱  vector<int> myvector (3,100);
۲  vector<int>::iterator it;
۳  it = myvector.begin();
۴  it = myvector.insert ( it , 200 );
۵  myvector.insert (it,2,300);
۶  // "it" no longer valid, get a new one:
۷  it = myvector.begin();
۸  vector<int> anothervector (2,400);
۹  myvector.insert (it+2,anothervector.begin(),anothervector.end());
۱۰ int myarray [] = { 501,502,503 };
۱۱ myvector.insert (myvector.begin(), myarray, myarray+3);
```

وکتورها

- تابع reserve برای اختصاص دادن حافظه و تغییر ظرفیت وکتور به کار برده می‌شود.

```
۱  vector<int> bar;
۲  sz = bar.capacity();
۳  bar.reserve(100);   // change the capacity to 100
۴  cout << "making bar grow:\n";
۵  for (int i=0; i<200; ++i) {
۶      bar.push_back(i);
۷      if (sz!=bar.capacity()) {
۸        sz = bar.capacity();
۹        cout << "capacity changed: " << sz << '\n';
۱۰     }
۱۱ }
```

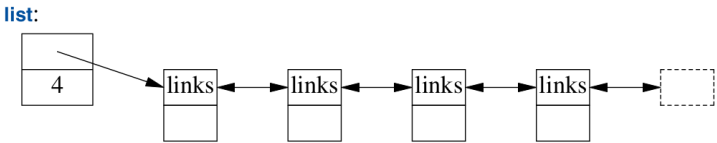
صف دوطرفه

- یکی از ساختارداده‌های ترتیبی صف دوطرفه [1] است که شبیه وکتور است با این تفاوت که پیاده‌سازی آن به نحوی است که درج و حذف از ابتدای وکتور را نیز فراهم می‌کند.
- اضافه کردن این رفتار سرباری نیز دارد به طوری که بهره‌وری آن از وکتور کمتر است ولی در جایی که به اضافه و درج در ابتدای وکتور نیاز است می‌تواند این کار را بهینه‌تر انجام دهد.

[1] double ended queue

لیست

- یکی از ساختارداده‌های ترتیبی لیست است که در واقع پیاده‌سازی یک لیست پیوندی دوطرفه [1] است.



- معمولا در جایی که به ظرفی از عناصر نیاز داریم از وکتور استفاده می‌کنیم. وکتور همچنین الگوریتم‌های جستجو و مرتب‌سازی بهینه‌تری دارد. اما وقتی نیاز به درج و حذف تعداد زیادی از عناصر در میانهٔ ظرف داریم از لیست استفاده می‌کنیم.

[1] doubly-linked list

لیست

- هنگامی از لیست استفاده می‌کنیم که می‌خواهیم در لیست درج و حذف کنیم، بدون اینکه عناصر دیگر لیست را جابجا کنیم.
- یک لیست را می‌توانیم شبیه یک وکتور مقداردهی اولیه و از آن استفاده کنیم.

```
list<Entry> phone_book = { {"Mr. X",123}, {"Mrs. Y", 456} };
int get_number(const string& s) {
    for (const auto& x : phone_book)
        if (x.name==s)
            return x.number;
    return 0; // use 0 to represent "number not found"
}
```

لیست

- همچنین با استفاده از یک پیمایشگر می‌توانیم عناصر یک لیست را پیمایش کنیم.

```
int get_number(const string& s) {
    for (auto p = phone_book.begin(); p!=phone_book.end(); ++p)
    if (p->name==s)
        return p->number;
    return 0; // use 0 to represent "number not found"
}
```

- پیمایشگر لیست از نوع پیمایشگر دوطرفه (bidirectional_iterator) است، یعنی تنها به جلو و عقب حرکت می‌کند. دسترسی به یک عنصر در میانهٔ لیست به صورت تصادفی توسط عملگر زیرنویس امکان پذیر نیست.

لیست

- با استفاده از توابع `insert` و `erase` می‌توانیم عنصری به لیست بیافزاییم و عنصری از لیست حذف کنیم.

```
۱ void f(const Entry& ee,
۲          list<Entry>::iterator p, list<Entry>::iterator q) {
۳     // add ee before the element referred to by p
۴     phone_book.insert(p,ee);
۵     // remove the element referred to by q
۶     phone_book.erase(q);
۷ }
```

- تابع `insert(p, elem)` عنصری که یک کپی از `elem` است را به لیست، قبل از عنصری که `p` به آن اشاره می‌کند می‌افزاید.
- همچنین `erase(p)` عنصری را که `p` به آن اشاره می‌کند، از لیست حذف می‌کند.

لیست

- برای لیست دو تابع `push_front` و `pop_front` نیز تعریف شده‌اند که می‌توان با استفاده از آنها عنصری را به ابتدای لیست افزود یا از ابتدای لیست عنصری را حذف کرد.
- دلیل این که این عملیات در لیست وجود دارند ولی در وکتور وجود ندارند این است که برای اضافه کردن یک عنصر در ابتدای لیست تنها لازم است عنصری را در حافظه ساخته و اشاره‌گری به اولین عنصر لیست فعلی در آن قرار داد. اما برای اضافه کردن یک عنصر به ابتدای یک وکتور باید همهٔ عناصر وکتور را جابجا کرد.
- تابع `sort` در لیست عناصر آن را مرتب‌سازی می‌کند.
- با استفاده از تابع `remove` می‌توان یک عنصر را با استفاده از مقدار آن از لیست حذف کرد.

```
۱ int myints[]= {17,89,7,14};
۲ list<int> mylist (myints,myints+4);
۳ mylist.remove(89);
```

لیست

- دو لیست مرتب‌شده را می‌توان با استفاده از تابع merge ادغام کرد.

```
۱ int first[] = {5,10,15,20,25};
۲ int second[] = {50,40,30,20,10};
۳ vector<int> v(10);
۴ sort (first,first+5);
۵ sort (second,second+5);
۶ merge (first,first+5,second,second+5,v.begin());
```

پیچیدگی

- پیچیدگی درج در یک وکتور بسته به اینکه وکتور ظرفیت داشته باشد، یا ظرفیت آن تمام شده باشد و نیاز به تخصیص حافظهٔ جدید داشته باشد $O(1)$ یا $O(n)$ است، اما پیچیدگی درج یک عنصر در یک لیست $O(1)$ است.
- البته باید توجه داشت در حالتی که وکتور ظرفیت داشته باشد، برای درج در آن نیاز به تخصیص حافظه نیست، اما در لیست در هر بار اضافه کردن یک عنصر باید یک فضای به اندازه همان یک عنصر در حافظه تخصیص داد.

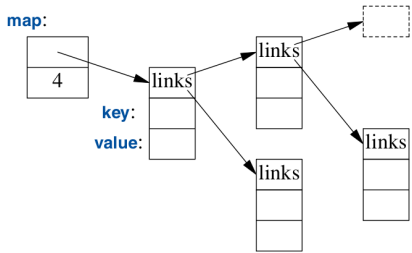
نگاشت

- نگاشت [1] یکی از ساختارهای داده است که توسط آن می‌توان تعدادی زوج را ذخیره سازی کرد. هر زوج شامل یک کلید و یک مقدار است. می‌گوییم یک ظرفِ نگاشت هر کلید را به یک مقدار نگاشت می‌کند و ارتباطی بین یک مجموعه از کلیدها و یک مجموعه از مقادیر را نشان می‌دهد.
- یک نگاشت را آرایهٔ رابطه‌ای یا لغت‌نامه یا دیکشنری [2] نیز می‌نامیم.
- در یک نگاشت هر کلید یکتاست و دو کلید یکسان را نمی‌توان در یک نگاشت درج کرد. از طرفی در ساختار دادهٔ چندنگاشت [2] می‌توان دو کلید یکسان درج کرد.

[1] map
[2] dictionary
[2] multi-map

نگاشت

- علاوه بر نگهداری زوج‌های کلید و مقدار، یک نگاشت، کلیدها را توسط یک درخت جستجوی دودویی [1] ذخیره‌سازی می‌کند. در درخت جستجوی دودویی، مقادیر همهٔ رئوس زیردرخت سمت چپ در یک رأس پدر، کوچکتر از مقدار رأس پدر است و همچنین مقادیر همهٔ رئوس زیردرخت سمت راست بزرگتر از رأس پدر است. بنابراین جستجو در کلیدهای یک نگاشت با پیچیدگی زمانی کمتر و در نتیجه بهینه‌تر انجام می‌شود.



- پیچیدگی جستجو برای یک نگاشت با n عضو $O(\log n)$ است.

[1] binary search tree

نگاشت

- می‌توانیم از یک نگاشت بدین صورت استفاده کنیم.

```
map<string,int> phone_book { {"Mr. X",123}, {"Mrs. Y", 456} };

int get_number(const string& s) {
    return phone_book[s];
}
```

- یک نگاشت از رشته‌ها و به اعداد صحیح تعریف می‌کنیم. پس در اینجا کلیدها رشته‌ها و مقادیر اعداد صحیح هستند.

نگاشت

- پس یک نگاشت از کلاس map مجموعه‌ای است از زوج‌ها که توسط کلاس زوج pair تعریف شده‌اند.
- یک زوج توسط یک قالب با دو متغیر تعریف شده است. متغیر اول نوع داده‌ای کلید و متغیر دوم نوع داده‌ای مقدار را تعیین می‌کند.

```
۱ pair <string,double> product1;  // default constructor
۲ pair <string,double> product2 ("tomatoes",15.5);   // value init
۳ pair <std::string,double> product3 (product2); // copy constructor
۴
۵ map<string,double> mymap;
۶ mymap.insert (product1);
۷ mymap.insert (product2);
۸ mymap.insert (product3);
۹ mymap.insert (pair <std::string,double> ("potatoes", 10.5));
```

نگاشت

- عملگر زیرنویس [1] برای نگاشت تعریف شده است، بنابراین می‌توانیم به صورت زیر نیز کلیدها و مقادیر را در آن درج کنیم.

```
map<char,string> mymap;

mymap['a']="an element";
mymap['b']="another element";
mymap['c']=mymap['b'];
```

- پس برای درج یک کلید در یک نگاشت توسط عملگر زیرنویس، نیازی نیست آن کلید در نگاشت وجود داشته باشد. با استفده از عملگر زیرنویس، در صورتی که یک کلید در نگاشت وجود نداشته باشد آن کلید در نگاشت درج می‌شود و در صورتی که آن کلید در نگاشت وجود داشته باشد، مقدار آن تغییر می‌کند.

[1] subscript operator

نگاشت

- سازندهٔ نگاشت می‌تواند توسط پیمایشگر یک نگاشت دیگر نیز نگاشت را بسازد.

```
۱ map<char,int> first;
۲ first['a']=10; first['b']=30;
۳ first['c']=50; first['d']=70;
۴ map<char,int> second (first.begin(),first.end());
۵ map<char,int> third (second);
```

- همانند وکتور، عملگر زیرنویس در نگاشت نیز استثنا ارسال نمی‌کند. می‌توان از تابع at برای دسترسی به عناصر نگاشت استفاده کرد. این تابع در صورت خطا استثنا ارسال می‌کند.

نگاشت

- با استفاده از تابع `find` می‌توان در یک نگاشت جستجو کرد. اگر جستجو موفقیت‌آمیز بود، این تابع یک پیمایشگر به کلید یافته شده بازمی‌گرداند. اگر کلید مورد نظر یافته نشود، تابع مقدار پیشمایگر `()end` را بازمی‌گرداند که در واقع به بعد از آخرین عنصر در نگاشت اشاره می‌کند.
- پیمایشگر در یک نگاشت دو عضو `first` و `second` دارد که به کلید و مقدار در یک زوج اشاره می‌کنند.

نگاشت

- در مثال زیر با دریافت یک حرف الفبای انگلیسی معادل عددی آن را در یک نگاشت جستجو می‌کنیم.

```
map<char, int> m;
for(int i = 0; i <26; i++) {
    m.insert(pair<char, int>('A'+i, 65+i));
}
char ch;
cout << "Enter key: ";
cin >> ch;
map<char, int>::iterator p;
p = m.find(ch);
if(p != m.end())
    cout << "The ASCII value of" << p->first << " is " << p->second;
  else
    cout << "Key not in map.\n";
```

نگاشت

- تابع درج (`insert`) یک زوج بازمی‌گرداند. متغیر دوم در این زوج یک متغیر بولی است. در صورتی که تابع `insert` موفقیت‌آمیز انجام شود، متغیر دوم در زوج بازگردانده شده، مقدار درست را بازمی‌گرداند و در غیراینصورت مقدار نادرست را بازمی‌گرداند.
- همچنین متغیر اول در زوج بازگردانده شده یک پیمایشگر است.
- در صورتی که کلید مورد نظر برای درج در نگاشت وجود داشت، تابع درج پیمایشگری به کلید یافته شده بازمی‌گرداند و در صورتی که کلید مورد نظر در نگاشت وجود نداشت، کلید و مقدار را درج کرده و پیمایشگری به کلید تازه درج شده بازمی‌گرداند.

نگاشت

```
map<char,int> mymap;

// first insert function version (single parameter):
mymap.insert ( std::pair<char,int>('a',100) );
mymap.insert ( std::pair<char,int>('z',200) );

pair<std::map<char,int>::iterator,bool> ret;
ret = mymap.insert ( std::pair<char,int>('z',500) );
if (ret.second==false) {
    cout << "element 'z' already existed";
    cout << " with a value of " << ret.first->second << '\n';
}
```

نگاشت

- همچنین با استفاده از تابع erase می‌توان تعدادی از زوج‌ها را در یک نگاشت حذف کرد.

```
map<char,int> mymap;
map<char,int>::iterator it;

// insert some values:
mymap['a']=10; mymap['b']=20; mymap['c']=30;
mymap['d']=40; mymap['e']=50; mymap['f']=60;

it=mymap.find('b');
mymap.erase (it); // erasing by iterator
mymap.erase ('c'); // erasing by key
it=mymap.find ('e');
mymap.erase ( it, mymap.end() );    // erasing by range
```

- حذف می‌تواند توسط یک مقدار یا توسط یک پیمایشگر برای یک عضو یا بازه‌ای از اعضا انجام شود.

نگاشت

- پیمایشگر نگاشت را شبیه به پیمایشگر وکتور و لیست می‌توان استفاده کرد و اعضای نگاشت را پیمایش کرد.

```
// show content:
for (it=mymap.begin(); it!=mymap.end(); ++it)
    cout << it->first << " => " << it->second << '\n';
```

چندنگاشت

- چندنگاشت شبیه یک نگاشت است با این تفاوت که کلیدها می‌توانند تکرار شوند.
- تابع درج پیمایشگری به کلید درج شده بازمی‌گرداند.
- تابع `erase` نیز همهٔ زوج‌ها با یک کلید معین را حذف می‌کند.
- تابع شمارش (`count`) تعداد کلیدها را در نگاشت می‌شمارد.
- تابع `equal_range` یک کلید را دریافت کرده و بازه‌ای با شروع از اولین تکرار کلید و پایان با آخرین تکرار کلید مورد نظر را بازمی‌گرداند.

چندنگاشت

- مقدار بازگردانده شده توسط تابع equal_range یک دوتایی است که مقدار اول آن پیمایشگری به ابتدای محدودهٔ یافت شده و مقدار دوم آن پیمایشگری به بعد از انتهای محدودهٔ یافت شده است.

```
std::multimap<char,int> mm;
mm.insert(pair<char,int>('a',10)); mm.insert(make_pair('b',20));
mm['b'] = 30; mm['b'] = 40; mm['c'] = 50; mm['c'] = 60; mm['d'] = 60;
cout << "mm contains:\n";
for (char ch='a'; ch<='d'; ch++) {
    pair <multimap<char,int>::iterator,
                  multimap<char,int>::iterator> ret;
    ret = mm.equal_range(ch);
    cout << ch << " =>";
    for (multimap<char,int>::iterator it=ret.first;
             it!=ret.second; ++it)
      cout << ' ' << it->second;
    cout << '\n';
}
```

چندنگاشت

```
multimap<char,int> mm;
mm.insert(make_pair('x',50));
mm.insert(make_pair('y',100));
mm['y']=150; mm['y']=200; mm['z']=250; mm['z']=300;

for (char c='x'; c<='z'; c++) {
    cout << "There are " << mm.count(c)
           << " elements with key " << c << ":";
    multimap<char,int>::iterator it;
    for (it=mm.equal_range(c).first;
          it!=mm.equal_range(c).second; ++it)
        cout << ' ' << (*it).second;
    cout << '\n';
}
```

چندنگاشت

- تابع lower_bound پیمایشگری بازمی‌گرداند به اولین کلیدی که از کلید داده شده توسط تابع بزرگتر یا مساوی است. تابع upper_bound پیمایشگری بازمی‌گرداند به اولین کلیدی که از کلید داده شده توسط تابع بزرگتر است.

```
multimap<char,int> mm;
multimap<char,int>::iterator it,itlow,itup;
mm.insert(std::make_pair('a',10));
mm.insert(std::make_pair('b',121));
mm['c']=1001; mm['c']=2002;
mm['d']=11011; mm['e']=44;

itlow = mm.lower_bound ('b');  // itlow points to b
itup = mm.upper_bound ('d');   // itup points to e (not d)

// print range [itlow,itup):
for (it=itlow; it!=itup; ++it)
    cout << (*it).first << " => " << (*it).second << '\n';
```

چندنگاشت

از آنجایی که کلیدها در یک نگاشت مرتب شده‌اند، بنابراین اگر بخواهیم از یک کلاس دلخواه به عنوان کلید برای یک نگاشت استفاده کنیم، عملگر کوچکتر < باید برای آن کلاس سربارگذاری شده باشد.

```
1  class student {
2      int id;
3      string name;
4  public:
5      bool operator<(const student& s) { return (id < s.id); }
6      // ...
7  };
8  map<student, double> grades;
```

- به جز کلید، مقدار در یک نگاشت نیز می‌تواند از یک کلاس دلخواه و تعریف‌شده توسط کاربر باشد. اما نیازی نیست عملگر کوچکتر بر روی آن کلاس تعریف شده باشد.

مجموعه

- مجموعه (`set`) شبیه نگاشت (`map`) است با این تفاوت که به جای نگهداری یک زوج کلید و مقدار، فقط یک کلید را نگهداری می‌کند.
- پس یک مجموعه ساختار داده‌ای است برای نگهداری کلیدهای یکتا که در یک درخت جستجوی دودویی ذخیره شده‌اند.
- چندمجموعه نیز مانند مجموعه است با این تفاوت که کلیدها می‌توانند تکرار شوند.

پشته

- یک پشته ساختار داده‌ای است که داده‌ها را می‌توان به انتهای آن اضافه کرد یا از انتهای آن حذف کرد.
- پشته را می‌توان توسط یک وکتور یا صف‌دوطرفه ساخت. پس زیرساخت پشته ساختارداده‌ٔ جدیدی نیست بلکه تنها عملگرهای مورد نیاز برای آن تعریف شده‌اند.
- با استفاده از تابع push در پشته درج و با استفاده از تابع pop از پشته مقداری را حذف می‌کنیم.

```
۱ deque<int> mydeque (3,100);  // deque with 3 elements
۲ vector<int> myvector (2,200);  // vector with 2 elements
۳ stack<int> first;  // empty stack
۴ stack<int> second (mydeque);  // stack initialized to copy of deque
۵ stack<int, vector<int> > third;  // empty stack using vector
۶ stack<int, vector<int> > fourth (myvector);
```

صف

- یک صف ساختار داده‌ای است که داده‌ها را می‌توان به انتهای آن اضافه کرد یا از ابتدای آن حذف کرد. اولین داده‌ای که وارد صف می‌شود، اولین داده‌ای است که از صف خارج می‌شود.
- صف را می‌توان توسط یک لیست یا صف‌دوطرفه ساخت. پس زیرساخت صف نیز ساختارداده‌ٔ جدیدی نیست بلکه تنها تابع push برای درج و تابع pop برای حذف برای آن تعریف شده‌اند. به طور پیش‌فرض صف با استفاده از صف‌دوطرفه ساخته می‌شود.

```
deque<int> mydeck (3,100);  // deque with 3 elements
list<int> mylist (2,200);  // list with 2 elements
queue<int> first;  // empty queue
queue<int> second (mydeck);  // queue initialized to copy of deque
queue<int,list<int> > third; // empty queue with
     // list as underlying container
queue<int,list<int> > fourth (mylist);
```

صف

- نوع خاصی از صف، صف اولویت است که توسط کلاس priority_queue پیاده‌سازی شده است. داده‌ها در صف اولولیت با استفاده از اولویتشان خارج می‌شود. عنصری که بیشترین اولویت را دارد، به عنوان اولین عنصر از صف خارج می‌شود. بنابراین عملگر مقایسه‌ای کوچکتر باید برای آنها تعریف شده باشند.
- برای مثال اگر صفی با مقادیر صحیح داشته باشیم، بزرگترین عدد اول از همه از صف خارج می‌شود.

```
priority_queue<int> mypq;
mypq.push(30); mypq.push(100); mypq.push(25); mypq.push(40);

cout << "Popping out elements...";
while (!mypq.empty()) {
     cout << ' ' << mypq.top();
     mypq.pop();
} // Popping out elements... 100 40 30 25
```

الگوریتم

- در کتابخانهٔ <algorithm> الگوریتم‌هایی تعریف شده‌اند که از آنها می‌توان برای همهٔ ساختارداده‌های کتابخانهٔ استاندارد استفاده کرد.
- برای مثال all_of بررسی می‌کند آیا برای همهٔ عناصر یک محدوده از یک ظرف (که با دو پیمایشگر ابتدا و انتها تعیین شده است) شرطی برقرار است یا خیر. تابع any_of بررسی می‌کند آیا برای یکی از عناصر یک محدوده شرطی برقرار است یا خیر.
- تابع for_each تابعی را بر روی همهٔ عناصر یک محدوده اعمال می‌کند.
- تابع find عنصری را در یک محدوده جستجو می‌کند و تابع find_if عنصری را جستجو می‌کند در صورتی که برای آن عنصر شرطی برقرار باشد.
- توابع count برای شمارش، search برای جستجوی یک زیردنباله، copy برای کپی یک محدوده replace برای جایگزین کردن یک محدوده، remove برای حذف مقادیری در یک محدوده، remove_if برای حذف مقادیری در یک محدوده در صورت وجود یک شرط، sort برای مرتب سازی، و merge برای الحاق دو محدوده از مقادیر به کار می‌روند.

الگوریتم

- در بسیاری از توابع کتابخانهٔ الگوریتم از اشاره‌گر به تابع استفاده شده است.
- می‌توان به جای استفاده از اشاره‌گر به تابع از فانکتور یا تابع لامبدا نیز استفاده کرد.

الگوریتم

- برای مثال برای شمردن عناصری از یک مجموعه که همگی فرد هستند به صورت زیر عمل می‌کنیم. تابعی که فرد بودن یک عنصر را بررسی می‌کند باید به عنوان یک ورودی به تابع count_if به عنوان یک اشاره‌گر به تابع وارد شود.

```
۱ bool IsOdd (int i) { return ((i%2)==1); }
۲
۳ vector<int> myvector;
۴ for (int i=1; i<10; i++) myvector.push_back(i);
۵ // myvector: 1 2 3 4 5 6 7 8 9
۶ int mycount = count_if (myvector.begin(), myvector.end(), IsOdd);
۷ cout << "myvector contains " << mycount  << " odd values.\n";
۸ // myvector contains 5 odd values.
```

الگوریتم

- می‌توانیم از توابع لامبدا نیز برای وارد کردن یک تابع به تابع دیگر استفاده کنیم.

```
۱ int main () {
۲   array<int,7> foo = {0,1,-1,3,-3,5,-5};
۳
۴   if ( any_of(foo.begin(), foo.end(), [](int i){return i<0;}) )
۵       cout << "There are negative elements in the range.\n";
۶
۷   return 0;
۸ }
```

الگوریتم

- همچنین به جای استفاده از اشاره‌گر به تابع یا توابع لامبدا می‌توانیم از فانکتورها استفاده کنیم.

```
۱  class myclass {           // function object type:
۲  public:
۳    void operator() (int i) {std::cout << ' ' << i;}
۴  };
۵  vector<int> myvector;
۶  myvector.push_back(10); myvector.push_back(20);
۷  myvector.push_back(30);
۸  myclass op;
۹  cout << "myvector contains:";
۱۰ for_each (myvector.begin(), myvector.end(), op);
```

الگوریتم

- رشته‌ها در زبان سی++ با استفاده از وکتور پیاده‌سازی شده‌اند.
- بنابراین همهٔ توابعی که برای وکتور وجود دارند را می‌توان برای رشته‌ها نیز استفاده کرد.
- از پیمایشگرها برای پیمایش رشته‌ها می‌توان استفاده کرد و همچنین از همهٔ توابع کتابخانهٔ الگوریتم نیز می‌توان برای کار با رشته‌ها استفاده کرد.

الگوریتم

- در کتابخانهٔ استاندارد بسیاری از توابع مورد نیاز برای کار با ظرف‌ها از قبیل مرتب‌سازی و جستجو و کپی عناصر وجود دارد.
- برای مثال می‌توانیم به سادگی عناصر یک وکتور را مرتب و سپس عناصر آن را در یک لیست کپی کنیم.

```
1 sort(vec.begin(),vec.end()); // use < for order
2 unique_copy(vec.begin(),vec.end(),lst.begin()); // don't copy adjacent
```

- برای مرتب‌سازی یک وکتور شامل عناصری از کلاس Entry عملگر < باید برای کلاس تعریف شده باشد.

```
1 bool operator<(const Entry& x, const Entry& y) { // less than
2     return x.name<y.name; // order Entries by their names
3 }
```

الگوریتم

- در صورتی که بخواهیم عناصری را به یک وکتور اضافه کنیم از تابع back_inserter استفاده می‌کنیم.

```
sort(vec.begin(),vec.end());
unique_copy(vec.begin(),vec.end(),back_inserter(lst));
```

- با فراخوانی back_inserter فضایی در پایان یک ظرف افزوده می‌شود.

الگوریتم

- از تابع `find` می‌توان برای جستجو در یک ظرف استفاده کرد. در صورتی که عنصر مورد نظر در یک ظرف یافته نشود، پیمایشگر `()end` بازگردانده می‌شود.

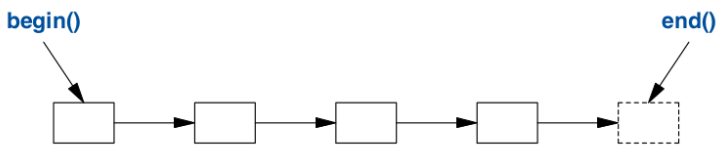
```
auto p = find(s.begin(),s.end(),c);
if (p!=s.end()) return true;
else return false;
```

- کد بالا را به صورت زیر نیز می‌توانیم بنویسیم.

```
return find(s.begin(),s.end(),c)!=s.end();
```

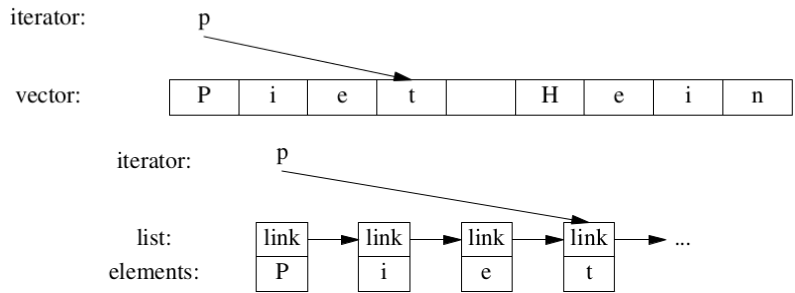
الگوریتم

- از آنجایی که همهٔ الگوریتم‌ها در کتابخانهٔ استاندارد با بازه‌های نیمه‌باز (بازه‌های بسته–باز) کار می‌کنند، پیمایشگر `end()` نیز به بعد از عنصر پایانی در یک ظرف اشاره می‌کند.



الگوریتم

- یک پیمایشگر به یک عنصر در یک ظرف اشاره می‌کند. برای وکتور پیمایشگر می‌تواند توسط یک اشاره‌گر تعریف شود، ولی برای ظرف‌های پیچیده تر پیمایشگر متناسب با کلاس ظرف مربوطه پیاده‌سازی می‌شود. برای پیمایشگرها عملگرهای ++ و * تعریف شده‌اند.



الگوریتم

- خلاصه‌ای از الگوریتم‌های کتابخانهٔ الگوریتم به صورت زیر است.

```
// For each element x in [b:e) do f(x)
f=for_each(b,e,f)
// p is the first p in [b:e) so that *p==x
p=find(b,e,x)
// p is the first p in [b:e) so that f(*p)
p=find_if(b,e,f)
// n is the number of elements *q in [b:e) so that *q==x
n=count(b,e,x)
// n is the number of elements *q in [b:e) so that f(*q)
n=count_if(b,e,f)
// Replace elements *q in [b:e) so that *q==v with v2
replace(b,e,v,v2)
// Replace elements *q in [b:e) so that f(*q) with v2
replace_if(b,e,f,v2)
```

الگوریتم

- خلاصه‌ای از الگوریتم‌های کتابخانهٔ الگوریتم به صورت زیر است.

```
۱  // Copy [b:e) to [out:p)
۲  p=copy(b,e,out)
۳  // Copy elements *q from [b:e) so that f(*q) to [out:p)
۴  p=copy_if(b,e,out,f)
۵  // Move [b:e) to [out:p)
۶  p=unique_copy(b,e,out)
۷  // Copy [b:e) to [out:p); 'dont copy adjacent duplicates
۸  p=move(b,e,out)
۹  // Sort elements of [b:e) using < as the sorting criterion
۱۰ sort(b,e)
۱۱ // Sort elements of [b:e) using f as the sorting criterion
۱۲ sort(b,e,f)
```

الگوریتم

- خلاصه‌ای از الگوریتم‌های کتابخانهٔ الگوریتم به صورت زیر است.

```
// [p1:p2) is the subsequence of the sorted sequence [b:e)
// with the value v; basically a binary search for v
(p1,p2)=equal_range(b,e,v)
// Merge two sorted sequences [b:e) and [b2:e2) into [out:p)
p=merge(b,e,b2,e2,out)
// Merge two sorted sequences [b:e) and [b2:e2) into [out:p)
// using f as the comparison
p=merge(b,e,b2,e2,out,f)
```

الگوریتم

- با استفاده از کتابخانهٔ استاندارد امکان اجرای الگوریتم‌ها به صورت موازی نیز وجود دارد.

```
sort(v.begin(),v.end()); // sequential
sort(seq,v.begin(),v.end()); // sequential (same as the default)
sort(par,v.begin(),v.end()); // parallel
```