Chapter 4
Data structures

Sebuah **data structure** atau struktur data, adalah cara menyimpan data di dalam memori komputer. Penting untuk memilih data structure yang tepat untuk suatu masalah, karena setiap data structure memiliki kelebihan dan kekurangannya masing-masing. Pertanyaan krusialnya adalah: operasi mana yang dapat dilakukan secara efisien dalam data structure yang dipilih?

Bab ini memperkenalkan data structure paling penting dalam *C++ Standard Library*. Sebaiknya gunakan *standard library* jika memungkinkan, karena ini akan menghemat banyak waktu. Di bagian selanjutnya dari buku ini, kita akan mempelajari data structure yang lebih canggih yang tidak tersedia dalam *standard library*.

Dynamic arrays

Dynamic arrays atau **Array dinamis** adalah array yang ukurannya dapat diubah selama eksekusi program. Struktur dynamic arrays yang paling populer di C++ adalah `vector`, yang dapat digunakan hampir seperti array biasa.

Kode berikut membuat sebuah vector kosong dan menambahkan tiga elemen ke dalamnya:

```
vector<int> v;
v.push_back(3); // [3]
v.push_back(2); // [3,2]
v.push_back(5); // [3,2,5]
```

Setelah ini, elemen-elemen dapat diakses seperti dalam array biasa:

```
cout << v[0] << "\n"; // 3
cout << v[1] << "\n"; // 2
cout << v[2] << "\n"; // 5
```

Fungsi `size` mengembalikan jumlah elemen dalam vector. Kode berikut mengiterasi vector dan mencetak semua elemennya:

```
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << "\n";
}
```

Cara yang lebih singkat untuk mengiterasi melalui sebuah vector adalah sebagai berikut:

```
for (auto x : v) {
    cout << x << "\n";
}
```

Fungsi `back` mengembalikan elemen terakhir dalam vector, dan fungsi `pop_back` menghapus elemen terakhir:

```
vector<int> v;
v.push_back(5);
v.push_back(2);
cout << v.back() << "\n"; // 2
v.pop_back();
cout << v.back() << "\n"; // 5
```

Kode berikut membuat sebuah vector dengan lima elemen:

```
vector<int> v = {2,4,2,5,1};
```

Cara lain untuk membuat vector adalah dengan memberikan jumlah elemen dan nilai awal untuk setiap elemen:

```
// ukuran 10, nilai awal 0
vector<int> v(10);
```

```
// ukuran 10, nilai awal 5
vector<int> v(10, 5);
```

Implementasi internal dari sebuah vector menggunakan array biasa. Jika ukuran vector bertambah dan array menjadi terlalu kecil, maka sebuah array baru dialokasikan dan semua elemen dipindahkan ke array baru. Namun, hal ini tidak sering terjadi, dan kompleksitas waktu rata-rata dari `push_back` adalah $O(1)$.

Struktur `string` juga merupakan sebuah *dynamic array* yang dapat digunakan hampir seperti vector. Selain itu, ada sintaks khusus untuk string yang tidak tersedia dalam data structure lainnya. String dapat digabungkan menggunakan simbol $+$. Fungsi `substr`(k, x) mengembalikan substring yang dimulai pada posisi k dan memiliki panjang x, serta fungsi `find`(t) menemukan posisi kemunculan pertama dari substring t.

Kode berikut menunjukkan beberapa operasi pada string:

```
string a = "hatti";
string b = a+a;
cout << b << "\n"; // hattihatti
b[5] = 'v';
cout << b << "\n"; // hattivatti
string c = b.substr(3,4);
cout << c << "\n"; // tiva
```

Set Structure

Struktur **set** adalah sebuah data structure yang menyimpan sekumpulan elemen. Operasi dasar dari *set* meliputi penyisipan elemen (*insertion*), pencarian (*search*), dan penghapusan (*removal*).

Pustaka standar C++ menyediakan dua implementasi set: Struktur set berbasis *balanced binary tree* dan operasinya berjalan dalam waktu $O(log\ n)$. Sementara itu, struktur `unordered_set` menggunakan *hashing*, sehingga operasinya berjalan dalam waktu $O(1)$ secara rata-rata.

Pemilihan implementasi set yang akan digunakan sering kali bergantung pada preferensi pengguna. Keuntungan dari struktur set adalah kemampuannya dalam menjaga urutan elemen serta menyediakan fungsi-fungsi yang tidak tersedia dalam `unordered_set`. Di sisi lain, `unordered_set` bisa lebih efisien dalam beberapa kasus.

Kode berikut membuat sebuah set yang berisi bilangan bulat dan menunjukkan beberapa operasi yang dapat dilakukan. Fungsi `insert` menambahkan elemen ke dalam set, fungsi `count` mengembalikan jumlah kemunculan suatu elemen dalam set, dan fungsi `erase` menghapus elemen dari set.

```
set<int> s;
s.insert(3);
s.insert(2);
s.insert(5);
cout << s.count(3) << "\n"; // 1
cout << s.count(4) << "\n"; // 0
s.erase(3);
s.insert(4);
cout << s.count(3) << "\n"; // 0
cout << s.count(4) << "\n"; // 1
```

Sebuah set dapat digunakan hampir seperti sebuah vector, tetapi tidak memungkinkan untuk mengakses elemen menggunakan notasi [ ]. Kode berikut membuat sebuah set, mencetak jumlah elemen di dalamnya, lalu mengiterasi semua elemennya:

```
set<int> s = {2,5,6,8};
cout << s.size() << "\n"; // 4
for (auto x : s) {
    cout << x << "\n";
}
```

Salah satu sifat penting dari set adalah bahwa semua elemennya *unik*. Oleh karena itu, fungsi `count` selalu mengembalikan 0 (jika elemen tidak ada dalam set) atau 1 (jika elemen ada dalam set), dan fungsi `insert` tidak akan menambahkan elemen jika elemen tersebut sudah ada dalam set. Kode berikut mengilustrasikan hal ini:

```
set<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 1
```

C++ juga memiliki struktur `multiset` dan `unordered_multiset` yang bekerja seperti `set` dan `unordered_set`, tetapi mereka dapat menyimpan beberapa instance dari sebuah elemen. Misalnya, dalam kode berikut, ketiga instance dari angka 5 ditambahkan ke dalam sebuah multiset:

```
multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 3
```

Fungsi `erase` menghapus semua instance dari sebuah elemen dalam multiset:

```
s.erase(5);
cout << s.count(5) << "\n"; // 0
```

Sering kali, hanya satu instance yang harus dihapus, yang dapat dilakukan sebagai berikut:

```
s.erase(s.find(5));
cout << s.count(5) << "\n"; // 2
```

Map structures

Sebuah **map** adalah sebuah array yang digeneralisasi dan terdiri dari pasangan *key-value* (*kunci-nilai*). Dalam sebuah array biasa, kunci selalu berupa bilangan bulat berurutan $0,1,\ldots,n-1$, di mana n adalah ukuran array. Namun, dalam sebuah *map*, kunci dapat berupa tipe data apa pun dan tidak harus berupa nilai yang berurutan.

Pustaka standar C++ memiliki dua implementasi *map* yang sesuai dengan implementasi *set*: struktur map didasarkan pada balanced binary tree, di mana akses elemen memerlukan waktu $O(log\ n)$, sementara struktur `unordered_map` menggunakan hashing, sehingga akses elemen memerlukan waktu $O(1)$ secara rata-rata.

Kode berikut membuat sebuah map di mana kunci (*keys*) adalah string dan nilainya (*values*) adalah bilangan bulat (*integers*):

```
map<string,int> m;
m["monkey"] = 4;
m["banana"] = 3;
m["harpsichord"] = 9;
cout << m["banana"] << "\n"; // 3
```

Jika nilai dari sebuah kunci diminta tetapi map tidak memilikinya, kunci tersebut akan otomatis ditambahkan ke dalam map dengan nilai default. Sebagai contoh, dalam kode berikut, kunci "aybabtu" dengan nilai 0 akan ditambahkan ke dalam map:

```
map<string,int> m;
cout << m["aybabtu"] << "\n"; // 0
```

Fungsi `count` memeriksa apakah sebuah kunci (*key*) ada dalam sebuah map:

```
if (m.count("aybabtu")) {
    // kunci ada
}
```

Kode berikut mencetak semua kunci dan nilai dalam sebuah map:

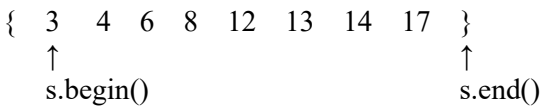
```
for (auto x : m) {
    cout << x.first << " " << x.second << "\n";
}
```

Iterators and ranges

Banyak fungsi dalam pustaka standar C++ beroperasi dengan *iterator*. Sebuah **iterator** adalah variabel yang menunjuk ke sebuah elemen dalam suatu data structure.

Iterator yang sering digunakan, `begin` dan `end`, mendefinisikan sebuah rentang (*range*) yang berisi semua elemen dalam sebuah data structure. Iterator `begin` menunjuk ke elemen pertama dalam data structure, sedangkan iterator `end` menunjuk ke posisi *setelah elemen terakhir*.

Situasinya terlihat sebagai berikut:



Perhatikan ketidaksimetrisan dalam iterator: `s.begin()` menunjuk ke sebuah elemen dalam data structure, sedangkan `s.end()` menunjuk ke luar data structure. Dengan demikian, rentang yang didefinisikan oleh iterator bersifat *half-open* (setengah terbuka).

Bekerja dengan Rentang (*Ranges*)

Iterator digunakan dalam fungsi pustaka standar C++ yang menerima rentang elemen dalam suatu data structure. Biasanya, kita ingin memproses semua elemen dalam data structure, sehingga iterator `begin` dan `end` diberikan ke fungsi tersebut.

Sebagai contoh, kode berikut mengurutkan sebuah *vector* menggunakan fungsi `sort`, kemudian membalik urutan elemennya dengan `reverse`, dan akhirnya mengacak urutannya dengan `random_shuffle`:

```
sort(v.begin(), v.end());
reverse(v.begin(), v.end());
random_shuffle(v.begin(), v.end());
```

Fungsi-fungsi ini juga dapat digunakan dengan array biasa. Dalam hal ini, fungsi diberikan pointer ke array sebagai pengganti iterator:

```
sort(a, a+n);
reverse(a, a+n);
random_shuffle(a, a+n);
```

Iterator pada set

Iterator sering digunakan untuk mengakses elemen dalam sebuah set. Kode berikut membuat sebuah iterator `it` yang menunjuk ke elemen terkecil dalam set:

```
set<int>::iterator it = s.begin();
```

Cara yang lebih pendek untuk menuliskan kode tersebut adalah sebagai berikut:

```
auto it = s.begin();
```

Elemen yang ditunjuk oleh iterator dapat diakses menggunakan simbol $*$ (*asterisk*). Sebagai contoh, kode berikut mencetak elemen pertama dalam set:

```
auto it = s.begin();
cout << *it << "\n";
```

Iterator dapat digerakkan menggunakan operator ++ (maju) dan −− (mundur), yang berarti iterator akan berpindah ke elemen berikutnya atau sebelumnya dalam set.

Kode berikut mencetak semua elemen dalam urutan naik:

```
for (auto it = s.begin(); it != s.end(); it++) {
    cout << *it << "\n";
}
```

Kode berikut mencetak elemen terbesar dalam set:

```
auto it = s.end(); it--;
cout << *it << "\n";
```

Fungsi `find(`x`)` mengembalikan sebuah iterator yang menunjuk ke elemen yang bernilai x. Namun, jika set tidak mengandung x, iterator yang dikembalikan akan bernilai `end`.

```
auto it = s.find(x);
if (it == s.end()) {
    // x tidak ditemukan
}
```

Fungsi `lower_bound(`x`)` mengembalikan iterator ke elemen terkecil dalam set yang nilainya *setidaknya* x, sedangkan fungsi `upper_bound(`x`)` mengembalikan iterator ke elemen terkecil dalam set yang nilainya *lebih besar dari* x. Jika tidak ada elemen yang memenuhi kondisi tersebut, nilai yang dikembalikan adalah `end`.

Fungsi-fungsi ini tidak didukung oleh struktur `unordered_set` yang tidak mempertahankan urutan elemen.

Sebagai contoh, kode berikut menemukan elemen yang paling dekat dengan x:

```
auto it = s.lower_bound(x);
if (it == s.begin()) {
    cout << *it << "\n";
} else if (it == s.end()) {
    it--;
    cout << *it << "\n";
} else {
    int a = *it; it--;
    int b = *it;
    if (x - b < a - x) cout << b << "\n";
    else cout << a << "\n";
}
```

Kode ini mengasumsikan bahwa set tidak kosong dan mengevaluasi semua kemungkinan kasus menggunakan sebuah iterator `it`. Pertama, iterator menunjuk ke elemen terkecil yang nilainya setidaknya x. Jika `it` sama dengan `begin`, elemen yang bersesuaian adalah yang paling dekat dengan x. Jika `it` sama dengan `end`, elemen terbesar dalam set adalah yang paling dekat dengan x. Jika tidak ada kasus sebelumnya yang berlaku, elemen yang paling dekat dengan x adalah elemen yang bersesuaian dengan `it` atau elemen sebelumnya.

Atau berikut penjelasan yang lebih mudah:

Kode ini mengasumsikan bahwa *set* tidak kosong dan mengevaluasi semua kemungkinan kasus menggunakan iterator it.

1. **Langkah pertama**, iterator menunjuk ke elemen terkecil dalam *set* yang nilainya setidaknya x.
2. **Jika iterator sama dengan `begin()`**, artinya elemen pertama dalam *set* adalah yang paling dekat dengan x, sehingga langsung dicetak.
3. **Jika iterator sama dengan `end()`**, artinya tidak ada elemen yang nilainya setidaknya x, sehingga elemen terbesar dalam *set* yang paling dekat dengan x akan dicetak.
4. **Jika tidak termasuk dua kasus di atas**, maka nilai terdekat dengan x bisa jadi adalah elemen yang ditunjuk oleh `it` atau elemen sebelumnya (`it-1`). Maka, kita membandingkan kedua elemen tersebut dan mencetak yang paling dekat dengan x.

Struktur Lainnya

Bitset

Sebuah **bitset** adalah sebuah array di mana setiap nilainya adalah 0 atau 1. Sebagai contoh, kode berikut membuat sebuah bitset yang berisi 10 elemen:

```
bitset<10> s;
s[1] = 1;
s[3] = 1;
s[4] = 1;
s[7] = 1;
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

Keuntungan menggunakan bitset adalah bahwa ia membutuhkan lebih sedikit memori dibandingkan dengan array biasa, karena setiap elemen dalam bitset hanya menggunakan satu bit memori. Sebagai contoh, jika n bit disimpan dalam sebuah array `int`, maka $32n$ bit memori akan digunakan, tetapi bitset yang sesuai hanya membutuhkan n bit memori. Selain itu, nilai-nilai dalam bitset dapat dimanipulasi secara efisien menggunakan operator bit, yang memungkinkan optimalisasi algoritma menggunakan bitset.

Kode berikut menunjukkan cara lain untuk membuat *bitset* yang sama seperti di atas:

```
bitset<10> s(string("0010011010")); // dari kanan ke kiri
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

Fungsi `count` mengembalikan jumlah angka satu dalam bitset:

```
bitset<10> s(string("0010011010"));
cout << s.count() << "\n"; // 4
```

Kode berikut menunjukkan contoh penggunaan operasi bit:

```
bitset<10> a(string("0010110110"));
bitset<10> b(string("1011011000"));
cout << (a & b) << "\n"; // 0010010000
cout << (a | b) << "\n"; // 1011111110
cout << (a ^ b) << "\n"; // 1001101110
```

Deque

Sebuah **deque** adalah dynamic arrays yang ukurannya dapat diubah secara efisien di kedua ujungnya. Seperti vector, deque menyediakan fungsi `push_back` dan `pop_back`, tetapi juga memiliki fungsi `push_front` dan `pop_front` yang tidak tersedia dalam vector.

Sebuah deque dapat digunakan sebagai berikut:

```
deque<int> d;
d.push_back(5);  // [5]
d.push_back(2);  // [5,2]
d.push_front(3); // [3,5,2]
d.pop_back();    // [3,5]
d.pop_front();   // [5]
```

Implementasi internal deque lebih kompleks dibandingkan vector, dan karena alasan ini, deque lebih lambat dibandingkan vector. Meskipun demikian, baik penambahan maupun penghapusan elemen memerlukan waktu rata-rata $O(1)$ di kedua ujungnya.

Stack

Sebuah **stack** adalah data structure yang menyediakan dua operasi dengan kompleksitas waktu $O(1)$: menambahkan elemen ke atas (*push*) dan menghapus elemen dari atas (*pop*). Hanya elemen teratas yang dapat diakses dalam sebuah stack.

Kode berikut menunjukkan bagaimana stack dapat digunakan:

```
stack<int> s;
s.push(3);
s.push(2);
s.push(5);
cout << s.top(); // 5
s.pop();
cout << s.top(); // 2
```

Queue

Sebuah **queue** juga menyediakan dua operasi dengan kompleksitas waktu $O(1)$: menambahkan elemen ke akhir (*push*) dan menghapus elemen pertama (*pop*). Hanya elemen pertama dan terakhir yang dapat diakses dalam sebuah queue.

Kode berikut menunjukkan bagaimana queue dapat digunakan:

```
queue<int> q;
q.push(3);
q.push(2);
q.push(5);
cout << q.front(); // 3
q.pop();
cout << q.front(); // 2
```

Priority Queue

Sebuah **priority queue** menyimpan sekumpulan elemen. Operasi yang didukung adalah penyisipan (*insertion*) dan, tergantung pada jenis queue, pengambilan serta penghapusan elemen minimum atau maksimum. Penyisipan dan penghapusan memerlukan waktu $O(log\ n)$, sedangkan pengambilan elemen (*retrieval*) memerlukan waktu $O(1)$.

Meskipun ordered set secara efisien mendukung semua operasi dalam priority queue, keuntungan menggunakan priority queue adalah faktor konstanta yang lebih kecil. Priority queue biasanya diimplementasikan menggunakan struktur *heap*, yang jauh lebih sederhana dibandingkan balanced binary tree yang digunakan dalam ordered set.

Secara default, elemen dalam priority queue C++ diurutkan dalam urutan menurun, sehingga memungkinkan menemukan dan menghapus elemen terbesar dalam queue. Kode berikut mengilustrasikan penggunaannya:

```
priority_queue<int> q;
q.push(3);
q.push(5);
q.push(7);
q.push(2);
cout << q.top() << "\n"; // 7
q.pop();
cout << q.top() << "\n"; // 5
q.pop();
q.push(6);
cout << q.top() << "\n"; // 6
q.pop();
```

Jika kita ingin membuat priority queue yang mendukung pencarian dan penghapusan elemen terkecil, kita dapat melakukannya sebagai berikut:

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

Policy-Based Data Structures

Kompiler *g++* juga mendukung beberapa data structure yang bukan bagian dari pustaka standar C++. Struktur seperti ini disebut *policy-based* data structures. Untuk menggunakan struktur ini, baris berikut harus ditambahkan ke dalam kode:

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
```

Setelah itu, kita dapat mendefinisikan data structure `indexed_set`, yang mirip dengan set tetapi dapat diindeks seperti array. Definisi untuk nilai bertipe `int` adalah sebagai berikut:

```
typedef tree<int, null_type, less<int>, rb_tree_tag,
tree_order_statistics_node_update> indexed_set;
```

Sekarang kita dapat membuat set seperti berikut:

```
indexed_set s;
s.insert(2);
s.insert(3);
s.insert(7);
s.insert(9);
```

Keistimewaan dari set ini adalah kita dapat mengakses indeks yang akan dimiliki elemen dalam array yang terurut. Fungsi `find_by_order` mengembalikan iterator ke elemen pada posisi tertentu:

```
auto x = s.find_by_order(2);
cout << *x << "\n"; // 7
```

Dan fungsi `order_of_key` mengembalikan posisi suatu elemen:

```
cout << s.order_of_key(7) << "\n"; // 2
```

Jika elemen tidak ada dalam set, fungsi ini mengembalikan posisi tempat elemen tersebut seharusnya berada:

```
cout << s.order_of_key(6) << "\n"; // 2
cout << s.order_of_key(8) << "\n"; // 3
```

Kedua fungsi ini berjalan dalam waktu *logaritmik*.

Perbandingan dengan Sorting

Sering kali, sebuah masalah dapat diselesaikan menggunakan baik data structure maupun sorting. Terkadang, ada perbedaan mencolok dalam efisiensi aktual dari pendekatan-pendekatan ini, yang mungkin tidak terlihat hanya dari kompleksitas waktunya.

Mari kita pertimbangkan sebuah masalah di mana kita diberikan dua daftar A dan B, masing-masing berisi n elemen. Tugas kita adalah menghitung jumlah elemen yang terdapat di kedua daftar tersebut.

Sebagai contoh, untuk daftar berikut:

$$A = [5,2,8,9,4] \quad \text{dan} \quad B = [3,2,9,5]$$

Jawabannya adalah 3, karena angka 2, 5, dan 9 terdapat di kedua daftar.

Pendekatan langsung (*brute force*) untuk masalah ini adalah dengan memeriksa semua pasangan elemen, yang memerlukan waktu $O(n^2)$. Namun, kita akan membahas algoritma yang lebih efisien.

Algoritma 1

Kita membuat sebuah set yang berisi elemen-elemen dalam A, lalu kita mengiterasi elemen-elemen dalam B dan memeriksa apakah elemen tersebut juga terdapat dalam A.

Pendekatan ini efisien karena elemen dalam A disimpan dalam set. Dengan menggunakan struktur set, kompleksitas waktu algoritma ini adalah $O(n \log n)$.

Algoritma 2

Sebenarnya, kita tidak perlu menyimpan elemen dalam set terurut (*ordered set*), sehingga kita dapat menggunakan `unordered_set` sebagai gantinya. Pendekatan ini meningkatkan efisiensi hanya dengan mengganti data structure yang digunakan. Kompleksitas waktu algoritma ini menjadi $O(n)$.

Algoritma 3

Alih-alih menggunakan data structure, kita bisa menggunakan sorting.

1. Pertama, kita mengurutkan kedua daftar A dan B.
2. Setelah itu, kita melakukan iterasi secara bersamaan pada kedua daftar untuk menemukan elemen yang sama.

Kompleksitas waktu untuk sorting adalah $O(n \log n)$, sedangkan pencarian elemen yang sama berjalan dalam $O(n)$. Sehingga, total kompleksitas waktu algoritma ini adalah $O(n \log n)$.

Perbandingan Efisiensi

Berikut adalah tabel yang menunjukkan seberapa efisien algoritma di atas ketika n bervariasi dan elemen daftar adalah bilangan bulat acak antara 1 ... 10^9:

n	Algoritma 1	Algoritma 2	Algoritma 3
10^6	1.5 s	0.3 s	0.2 s
$2 \cdot 10^6$	3.7 s	0.8 s	0.3 s
$3 \cdot 10^6$	5.7 s	1.3 s	0.5 s
$4 \cdot 10^6$	7.7 s	1.7 s	0.7 s
$5 \cdot 10^6$	10.0 s	2.3 s	0.9 s

Algoritma 1 dan 2 sama kecuali dalam penggunaan struktur *set* yang berbeda. Dalam masalah ini, pilihan tersebut memiliki pengaruh penting terhadap waktu eksekusi, karena Algoritma 2, 4–5 kali lebih cepat dibandingkan Algoritma 1.

Namun, algoritma yang paling efisien adalah Algoritma 3, yang menggunakan *sorting*. Algoritma ini hanya memerlukan setengah waktu dibandingkan dengan Algoritma 2. Menariknya, meskipun kompleksitas waktu Algoritma 1 dan Algoritma 3 sama, yaitu $O(n \log n)$, Algoritma 3 sepuluh kali lebih cepat.

Hal ini dapat dijelaskan oleh fakta bahwa *sorting* adalah prosedur yang sederhana dan hanya dilakukan sekali di awal Algoritma 3, sementara sisanya berjalan dalam waktu *linear*. Sebaliknya, Algoritma 1 harus mempertahankan *balanced binary tree* yang kompleks selama keseluruhan eksekusi algoritma.