C++プログラミング II

第 6 回 リスト, 連想コンテナ

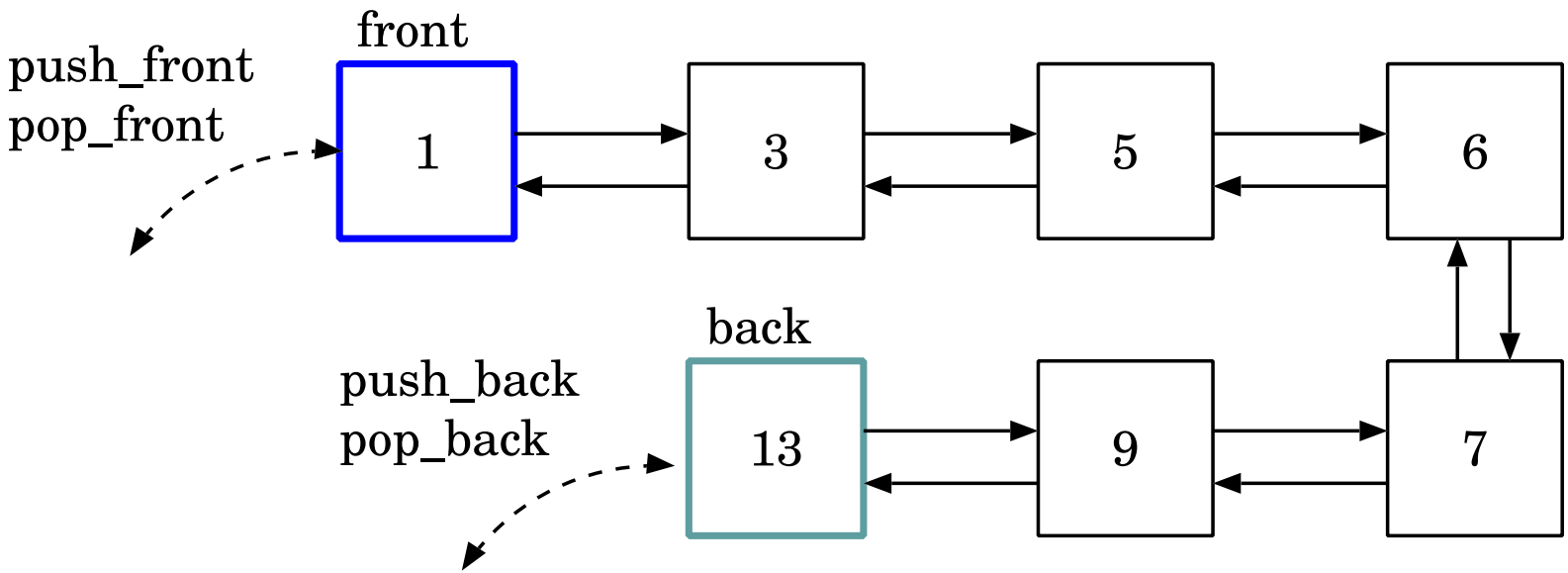
岡本秀輔

成蹊大学理工学部

std::list

リストとは

- 英語の list: 一覧表, 羅列
- データ構造としての特徴
  - データ数によらず挿入削除の時間が一定
  - std::list は双方向のリスト
    - ある要素はその前後のリンク情報を持つ
- `std::list` の主な操作
  - `push_front`, `push_back`, `pop_front`, `pop_back`, `front`, `back`, `empty`, `size`
- `#include<list>`ヘッダファイル



push/popの例

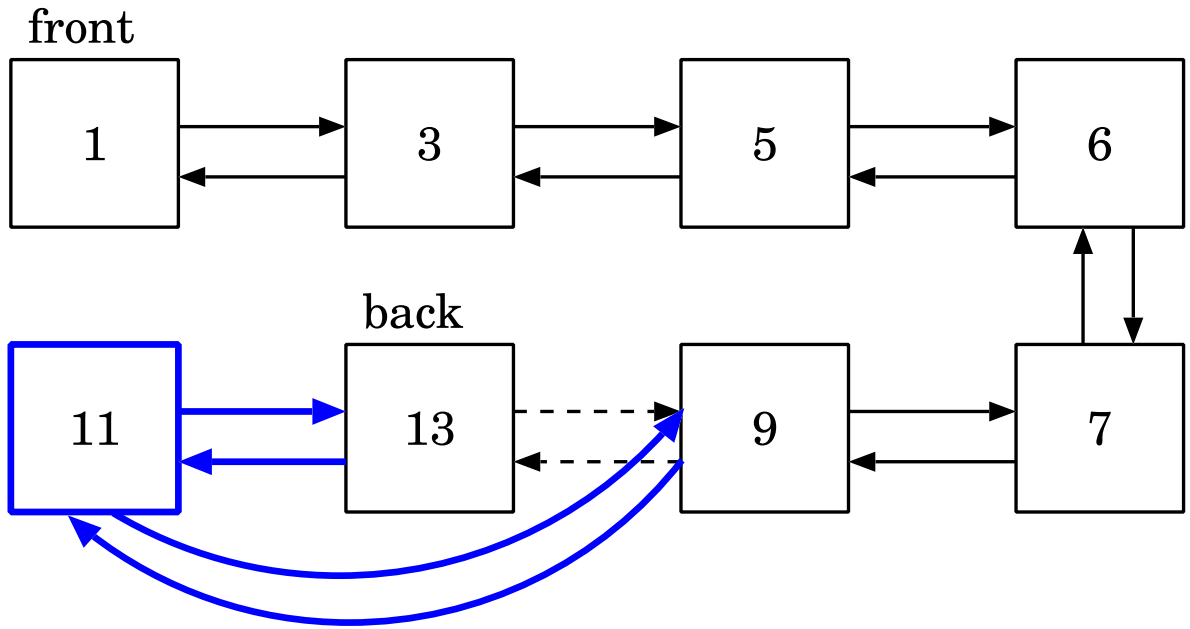
```
#include <iostream>
#include <list>
using std::cout;

void print(std::list<int>& x) {
    cout << x.size() <<": ";
    for (auto e : x) cout << e <<" ";
    cout <<"\n";
}

int main() {
    std::list<int> a{ 3, 5, 6, 7, 9};
    a.push_front(1);
    a.push_back(13);
    print(a);      // 7: 1 3 5 6 7 9 13
    a.pop_back();
    print(a);      // 6: 1 3 5 6 7 9
}
```

std::list のデータ挿入

- 挿入操作
  - 挿入時に他の要素の移動が発生しない
  - 関係する要素の前後のリンク情報を変更するのみ
- 要素 11 を 9 と 13 の間に挿入した場合：
  - 11 の場所を割り当てて、13 と 9 のリンクを更新



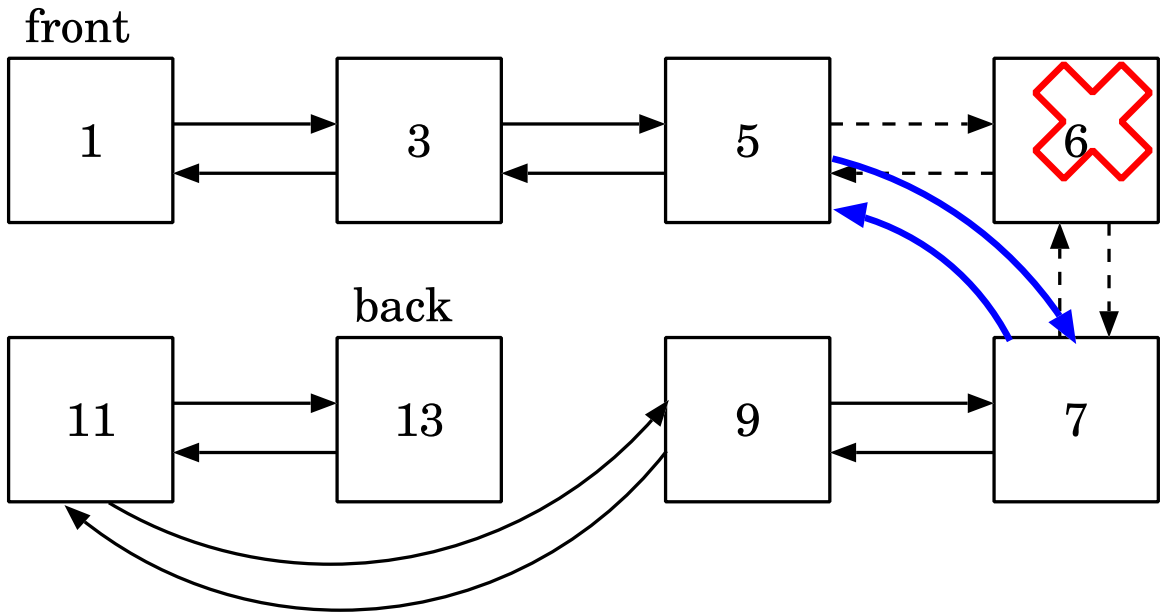
std::find と insert の例

```cpp
#include <iostream>
#include <algorithm>  // find
#include <list>
using std::cout;
void print(std::list<int>& x) {
    cout << x.size() <<": ";
    for (auto e : x) cout << e <<" ";
    cout <<"\n";
}

int main() {
    std::list<int> a{ 3, 5, 6, 7, 9, 13};
    auto itr {std::find(a.begin(), a.end(), 13)};
    std::cout << *itr <<"\n"; // 13
    a.insert(itr, 11); // 13 の前に挿入
    print(a);      // 7: 3 5 6 7 9 11 13
}
```

std::list のデータ削除

- 削除操作
  - 削除対象以外でデータの移動はない
  - 関係する要素の前後のリンク情報を変更するのみ
- 6 の削除
  - 5 と 7 の要素のリンクを更新し、6 の割り当てを解除



erase と remove の例

```
#include <iostream>
#include <algorithm>  // find
#include <list>
using std::cout;
void print(std::list<int>& x) {
   cout << x.size() <<": ";
   for (auto e : x) cout << e <<" ";
   cout <<"\n";
}

int main() {
   std::list<int> a{ 3, 5, 6, 7, 6, 9, 7, 11, 13};
   auto itr {std::find(a.begin(), a.end(), 6)};
   if (itr != a.end())
      a.erase(itr); // 最初の 6 を削除（イテレータ指定）
   print(a);         // 8: 3 5 7 6 9 7 11 13
   a.remove(7);      // 7 をすべて削除（値指定）
   print(a);         // 6: 3 5 6 9 11 13
}
```

sort と merge の例

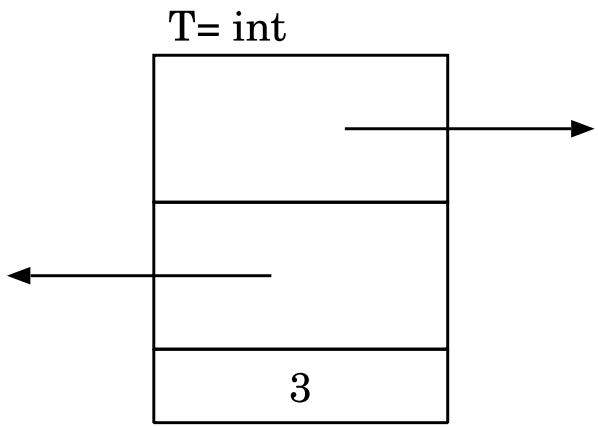
- sort() は整列し, merge() は二つの整列リストを融合する

```
#include <iostream>
#include <list>
using std::cout;
void print(std::list<int>& x) {
    cout << x.size() <<": ";
    for (auto e : x) cout << e <<" ";
    cout <<"\n";
}

int main() {
    std::list<int> a{4,2,6,0}, b{5,1,3,7};
    a.sort();           // a を整列
    print(a);           // 4: 0 2 4 6
    b.sort();           // b を整列
    print(b);           // 4: 1 3 5 7
    a.merge(b);         // a にまとめて, b は空に
    print(a);           // 8: 0 1 2 3 4 5 6 7
}
```

注意点

- 添字を使った要素アクセスができない
- 要素一つのメモリ使用量が多い
  - すべての要素が前後の情報を持つ
  - 例:`sizeof(int*)`=8, `sizeof(int)`=4
- 目的の要素を探すのは時間がかかる
  - `begin()`, `end()` の両端からたどるのみ
  - メモリ上にデータが散らばるので各要素のアクセス効率が悪い



std::list が使われる場面

- 要素の置かれる場所を覚えておきアクセスする
  - 要素の場所情報の保持：
    イテレータ, ポインタ, リファレンス
- コンテナごとの要素の場所情報の特徴
  - `vector`
    - 挿入時に無効となる可能性あり
    - 削除では対象要素より後ろ側がすべて無効
  - `deque`
    - 先頭末尾以外の挿入削除で全体が無効
  - `list`
    - 挿入時に無効となる要素はない
    - 削除では対象要素のみが無効
- 他のコンテナとの組み合わせ
  - 場所情報を `vector` や `deque` に保存する
  - 間接的に場所が不変の要素にアクセスできる

例1: イテレータの保存

- org に最初の点を記憶しておく

```
#include <iostream>
#include <list>
using std::cout;
struct Point { int x, y; };
int main() {
    std::list<Point> a{ {3,7} };
    auto org { a.begin() }; // 場所を記憶

    a.push_front( {1,5} );  // 適当に値を挿入
    a.insert(org, {2, 6} ); // org は無効ではない
    a.push_back( {4, 8} );
    for (auto& p : a) cout <<"{"<<p.x<<", "<<p.y<<"} ";

    // org は利用可能
    cout <<"\n{"<< org->x <<", "<< org->y <<"}\n";
}
```

例2: ポインタの保存

- org にポインタで場所と順序を保存

```
#include <iostream>
#include <vector>
#include <list>
void print(std::list<int>& x) {
   for (auto e : x) std::cout << e <<" ";
   std::cout <<"\n";
}

int main() {
   std::list<int> a{4,2,6,0}, b{1,3,5,7};
   std::vector<int*> org;
   for (auto& p: a) org.push_back(&p); // 保存
   a.sort();          // a を整列
   a.merge(b);        // a にまとめて，b は空に
   print(a);          // 0 1 2 3 4 5 6 7
   for (auto p : org) std::cout << *p <<" "; // 4 2 6 0
   std::cout <<"\n";
}
```

連続コンテナのまとめ

コンテナの選択

- 通常は `std::vector` を使用する
- スタックやキューのような専用の扱いがあるならば
  - `std::stack`, `std::queue`, `std::deque` を使う
  - `stack/queue` は内部で `std::deque` を使っている可能性が高い
- 要素の場所を覚えておきたい場合には `std::list` を他と組み合わせる
  - `std::vector` と `std::deque` は要素の挿入削除で場所情報が無効になる

メンバ関数の比較 (一部)

- 共通：コンテナを入れ替えても動作する

	vector	deque	list	array
empty/size/swap	○	○	○	○
begin/end(範囲 for 文)	○	○	○	○
front/back	○	○	○	○
operator[]/at	○	○	×	○
push_front/pop_front	×	○	○	×
push_back/pop_back	○	○	○	×
insert/erase/clear	○	○	○	×
resize	○	○	○	×

- 固有操作 (標準アルゴリズムより性能が良い)

  list : merge, splice, remove, remove_if, reverse, unique, sort

  array : fill

連想配列としてのstd::map

std::mapの基本

- 連想配列とは添字に整数以外の値も利用できる配列
- キーと値の対の関係を保持するデータ構造
- std::map<K,V>はテンプレート引数二つのコンテナ
  - K は < 演算子を持つ型, V は任意の型
  - #include <map>ヘッダファイル

```
#include <iostream>
#include <map>
using std::cout, std::string;

int main() {
    std::map<string,int> price;

    price["orange"] = 150;
    price["banana"] = 300;
    price["pineapple"] = 850;

    cout << price["banana"] <<"\n";
}
```

初期化

- map 内部では std::pair<K,V>形式で要素が保存される
- 初期化: キーと値の対を{}に入れて並べる

```
std::map<string,int> price {
    {"orange",150},
    {"banana",300},
    {"pineapple",450}
};
cout << price.size() <<"\n"; // 3 要素数
```

全要素の取得

- 範囲 for 文ではキーで整列した結果が得られる
- std::pair<K,V>は first と second のデータメンバに持つ構造体テンプレート
- 構造化バインディングを使って要素を取得しても良い

```
for (auto& p: price) // 要素は std::pair, 整列済み
   cout <<"["<< p.first <<":"<< p.second << "]";
cout <<"\n";

for (auto& [n,p]: price) // 要素対を各々取得
   cout <<"["<< n <<":"<< p << "]";
cout <<"\n";
```

```
[banana:300][orange:150][pineapple:450]
[banana:300][orange:150][pineapple:450]
```

検索

- find() メンバ関数でキーを指定してイテレータを得る
- 見つからない場合には end() 関数の値
- 結果イテレータの特徴
  - *it が std::pair 型
  - it->first でキー, it->second で値を得る
  - 構造化バインディングでキー／値の対を取得しても良い

```
std::map<string,int> price {
    {"orange",150},{"banana",300},{"pineapple",450} };

// 検索
if (auto it{price.find("banana")}; it!=price.end()){
    auto& [k, v] {*it};
    cout << k <<": " << v <<"\n";
}
```

読み出しの注意点

- 未挿入のキーで参照するとデフォルト値が挿入される
  - データを読み出す
- 検索には `find()` を使う
- この例は `price["tomato"]` を読み出している
  `int` のデフォルト値は 0

```
std::map<string,int> price
     { {"orange",150},{"banana",300},{"pine",450} };

if (price["tomato"] == 0)  // 注意：挿入される
   cout << "always true\n";
cout << price.size() <<"\n"; // 4 ：tomato 分がプラス
```

削除

- 要素の削除にはイテレータが必要
- C++20 から条件指定の複数削除 が可能となる
  - erase_if

```
std::map<std::string,int> price {
   {"orange",150},
   {"tomato",120},
   {"banana",300},
   {"pineapple",450} };

// 削除にはイテレータが必要
if (auto it{price.find("banana")}; it!=price.end())
   price.erase(it);

std::cout << price.size() <<"\n"; // 3
```

連想コンテナ

連想コンテナとは

- 要素を素早く探すための入れ物
- 保存する要素の値（キー）を使って対象要素を探す

	整列	順序なし
キー	`set`	`unordered_set`
キー/値の対	`map`	`unordered_map`
キー (重複あり)	`multiset`	`unordered_multiset`
キー/値の対 (重複あり)	`multimap`	`unordered_multimap`

- `map` と `unordered_map` のみ `[]` 演算子を持つ
- `[]` 演算子以外はほぼ共通の操作
  - 基本操作：Ctor の形式, 代入, 範囲 `for` 文, `empty`, `size`, `insert`, `erase`, `swap`, `clear`, `count`, `find`
  - 応用操作
    - `equal_range`, `lower_bound`, `upper_bound`,
    - `extract`, `merge`,
    - `key_comp`, `value_comp`

set の基本操作

- erase, find は iterator 型の値を返す。
- insert は std::pair<iterator,bool>を返す。
  - 同一キーが既にある場合は false で失敗する。
  - 重複の有無の確認に利用できる。

```
#include <iostream>
#include <set>
using std::cout, std::string;

int main() {
   std::set<string> a{"nn","ab","yu","ss"};

   if (auto [it,flag] {a.insert("nn")}; flag)
      cout << *it <<" is inserted\n";  // 出力なし
   cout << a.count("nn") <<"\n";       // 1

   if (auto it {a.find("nn")}; it != a.end())
      a.erase(it);
}
```

multiset の基本操作

- insert, erase, find が iterator 型の値を返す
  - insert は基本的に成功する。
  - メモリ不足などのエラーは別の方法で通知される

```
#include <iostream>
#include <set> // set と multiset は共通
using std::cout, std::string;

int main() {
   std::multiset<string> a{"nn","ab","yu","ss"};

   auto it{a.insert("nn")}; // 基本的に成功する
   cout << *it <<" is inserted\n";
   cout << a.count("nn") <<"\n";   // 2

   // もっと良いやり方は（次回）
   while ((it = a.find("nn")) != a.end())
      a.erase(it);
}
```

mapの基本操作

- Ctor, insert, iteratorがstd::pairであることを除き、setと同じ使い方

```
#include <iostream>
#include <map>
using std::cout, std::string;
int main() {
    std::map<string,int> a
        {{"nn",2}, {"ab",1}, {"yu",2}, {"ss",4}};

    if (auto [it,flag] {a.insert({"nn",3})}; flag) {
        auto [s,i] {*it}; // std::pair<string,int>
        cout << s <<" "<< i <<"\n";
    }
    cout << a.count("nn") <<"\n";                  // 1

    if (auto it {a.find("nn")}; it != a.end())
        a.erase(it);
}
```

multimapの基本操作

- Ctor, insert, iteratorがstd::pairであることを除き、multisetと同じ使い方

```
#include <iostream>
#include <map>
using std::cout, std::string;
int main()
{
    std::multimap<string,int> a
        {{"nn",2},{"ab",1},{"yu",2},{"ss",4}};

    auto it {a.insert({"nn", 3})};
    auto [s, i] {*it};
    cout << s <<" "<< i <<"\n";
    cout << a.count("nn") <<"\n";          // 2

    // もっと良いやり方は（次回）
    while ((it = a.find("nn")) != a.end())
        a.erase(it);
}
```

順序なし連想コンテナの基本操作

- <unordered_set>と<unordered_map>ヘッダファイル
- 使い方は set や map などと同じ（並ばないだけ)

```cpp
#include <iostream>
#include <unordered_map>
using std::cout, std::string;

int main() {
    std::unordered_map<string,int> a
        {{"nn",2}, {"ab",1}, {"yu",2}, {"ss",4}};

    if (auto [it,flag] {a.insert({"nn",3})}; flag)
        cout << it->first << it->second <<"\n";
    cout << a.count("nn") <<"\n";                    // 1

    if (auto it {a.find("nn")}; it != a.end())
        a.erase(it);
}
```

プログラム例

重複なし乱数の列

- [1,n] の範囲で n 個の重複のない乱数を生成する
  - std::vector<bool>の方が高速

```
#include <iostream>
#include <set>
#include "random.hpp" // 第 5 回講義
int main(int argc, char *argv[]) {
   const int n { argc > 1 ? std::atoi(argv[1]):10 };
   std::set<int> chk;
   UniDist r{1,n};
   for (int x{0}, i{0}; i < n; i++) {
      while (true) {
         x = r.get();                          // 乱数生成
         if (auto [it,f] {chk.insert(x)}; f) // 確認
            break;
      }
      std::cout << x <<" ";
   }
   std::cout <<"\n";
}
```

入力時の整列

- multiset が適している例
  - vector に入力して sort しても良いが。。。

```
#include <iostream>
#include <set>

template<typename T>
void print(const T& x) {
    for (auto& e: x) std::cout << e <<" ";
    std::cout <<"\n";
}

int main() {
    std::multiset<std::string> x;
    for (std::string a; std::cin >> a; )
        x.insert(a);
    print(x);
}
```

単語の出現頻度

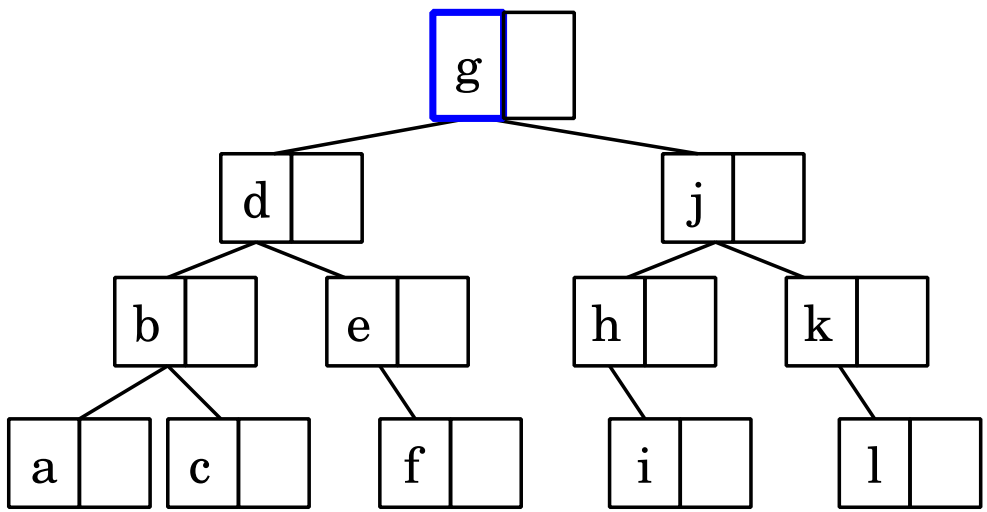
- map で数えるのみ
- 実用化では正規化 (normalize) が鍵を握る

```cpp
#include <iostream>
#include <cctype> // isalpha
#include <map>
auto normalize(std::string s) {
    if (!std::isalpha(s.back()))
        s = s.substr(0,s.size()-1);
    return s;
}
int main() {
    std::map<std::string,int> words;
    for (std::string s; std::cin >> s; )
        ++ words[ normalize(s) ];
    for (auto [w,c]: words)
        std::cout << w <<": "<< c <<"\n";
}
```

補足

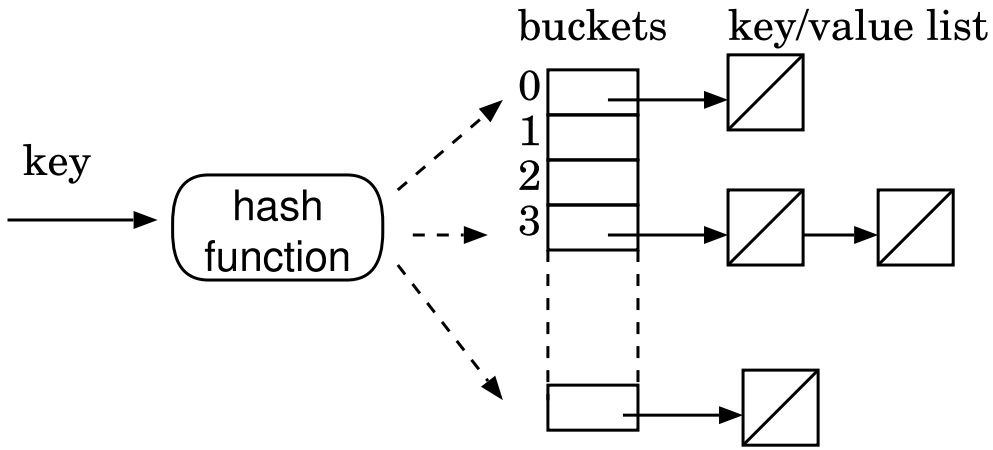
整列連想コンテナ

- 特徴
  - 要素がキーの値の順序を考えて保存されている
  - キーが $<$ 演算子を持つ必要がある
  - 要素数 N の対数時間 $(\log_2(N))$ で検索
  - 挿入削除でイテレータなどが無効にならない
- バランスした二分探索木が想定される
  - バランス: 根から葉までが深さがほぼ一定
  - 二分探索木：子の数が 2 以下、左の子は小、右の子は大



順序なし連想コンテナ

- 特徴
  - 要素がキー値の大小と無関係に保存されている
  - 要素数とは無関係にほぼ一定時間で検索可能
  - 整列版より高速だが要素が整列していない
  - 挿入でイテレータなどが無効
  - 削除ではイテレータは保たれる
- ハッシュテーブルが想定される
  - キーをハッシュ関数にかけて保存用配列の添字を得る
  - 配列は要素のリストで、そこで線形探索を行う



考慮事項

類似コンテナとどちらを使うかの選択する場面がある

- set<T>と map<T,bool>の関係
  - set の方がメモリ使用が少ないはず
  - すべて map に統一した方が面倒が少ない？
- set<int>と vector<bool>の関係
  - vector<bool>は省メモリ用に特殊化されている
  - 使用する整数の範囲とライブラリの実装しだいで vector<bool>の方が高速では？
- map<int,T>と vector<T>の関係
  - 使用する整数の範囲しだい