Program nauczania w ramach Olimpiady Informatycznej

Wojciech Baranowski

26 lutego 2024

Spis treści

1 Wstęp

W tej sekcji umieszczone będą pojęcia, których zrozumienie oraz znajomość praktyczna są niezbędne do dalszej kontynuacji nauki.

1.1 Git

Git - system kontroli wersji opracowany przez Linusa Torvalds'a w 2005 roku. Służy on do utrzymywania historii zmian danego zbioru plików i katalogów w potencjalnie rozdystrybuowanym środowisku. Pozwala na zrównoleglenie pracy wielu deweloperów nad jednym projektem, poprzez utrzymywanie nieliniowej struktury zmian.

1.1.1 Git - przykład liniowy:

Alice rozwija aplikację, jednak przydałaby się jej możliwość potencjalnego powrotu do poprzedniej (działającej) wersji programu, jeśli nowa zmiana pójdzie nie po jej myśli. Decyduje się więc skorzystać z git'a. Każdorazowo, gdy jej program znajdzie się w dostatecznie stabilnym stanie, Alice tworzy commit'a utworzonych przez nią zmian. Dzięki temu w dowolnym momencie może wrócić do dowolnego zacommitowanego punktu czasowego.

1.1.2 Git - w praktyce:

Mamy gotowy program, który liczy średnią arytmetyczną dwóch liczb (póki co całkowitych, zwracany wynik także jest całkowity).

1. inicjalizujemy repozytorium git'a: **git init**,
2. tworzymy gałąź (linię czasową) dla naszego programu: **git checkout -b nazwa**,
3. dodajemy nasz program do zbioru plików, które chcemy zacommitować: **git add program.cpp**,
4. commitujemy nasze zmiany: **git commit -m "initial commit"**.

Następnie przerabiamy program tak, by obsługiwał liczby zmiennoprzecinkowe.

1. dodajemy nową wersję pliku do zbioru plików, które chcemy zacommitować: **git add program.cpp**,
2. commitujemy nasze nowe zmiany: **git commit -m "floating points"**,
3. możemy zobaczyć historię naszych commitów przy pomocy następującej komendy: **git reflog**.

Załóżmy, że chcemy się cofnąć do poprzedniego commita, w którym program działał jedynie dla liczb całkowitych, a następnie zmienić jego działanie tak, że tylko wynik byłby liczbą zmiennoprzecinkową.

1. wylistowywujemy historię commitów: **git reflog**,
2. znajdujemy hash interesującego nas commita, do którego chcemy wrócić,
3. resetujemy program do wersji z wybranego commita: **git reset --hard hash-commita**,

W ten sposób możemy rozpocząć pracę na wcześniejszej wersji programu, a następnie zacommitować alternatywne rozwiązanie.

1.1.3 Repozytorium

Repozytorium gita pozwala na przechowywanie historii wersji w chmurze. Najpopularniejszym serwisem świadczącym takie usługi jest **github.com**. W celu skorzystania z githuba niezbędne jest utworzenie konta w serwisie. Następnie w ramach danego projektu należy utworzyć odpowiadające mu repozytorium, z którym następnie można zsynchronizować lokalnego gita.

1.1.4 Repozytorium - przykład

Dotychczasowo Alice rozwijała swoją aplikację na jednym urządzeniu. Ostatnio stwierdziła, że dostęp do projektu na drugim urządzeniu znacznie ułatwił by jej pracę. Decyduje się więc na zastosowanie repozytorium w serwisie github.com. Alice tworzy w serwisie repozytorium dla jej aplikacji, a następnie synchronizuje z nim swojego gita. Od tej pory Alice może w dowolnym momencie umieścić (wypchnąć) jej lokalne zmiany na githuba, a następnie pobrać (sklonować) na drugie urządzenie.

1.1.5 Repozytorium - klucze

Aby móc uwierzytelnić naszego klienta konsolowego w serwisie github, a co za tym idzie uzyskać dostęp do wypychania lokalnych zmian do repozytorium, musimy najpierw skonfigurować klucze dostępu dla danego urządzenia.

1. generujemy klucze w naszym systemie, które posłużą do uwierzytelniania w githubie **ssh-keygen -t rsa -b 4096 -C "adres-email-użyty-na-githubie"**,
2. domyślnie klucze dostępne są w katalogu **/home/nazwa-użytkownika/.ssh**,
3. kopiujemy zawartość klucza publicznego (klucz z rozszerzeniem **.pub**),
4. dodajemy klucz na githubie w zakładce **settings/SSH-and-GPG-keys** jako nowy klucz SSH.

Ponadto po stronie lokalnego gita wymagana będzie autoryzcja użytkownika. W konfiguracji gita należy ustawić nazwę oraz email użytkownika:

- git config --global user.name "IMIE NAZWISKO"
- git config --global user.email "EMAIL"

1.1.6 Repozytorium - w praktyce

Założyliśmy wcześniej konto w serwisie github.com. Teraz chcemy udostępnić w nim nasz projekt.

1. tworzymy nowe repozytorium na githubie, które będzie przeznaczone dla naszej aplikacji,
2. łączymy lokalne repozytorium gita z githubem: **git remote add origin adres-repozytorium**,
3. wypychamy lokalne zmiany do repozytorium: **git push origin nazwa-gałęzi**.

Następnie chcemy sklonować projekt z githuba na innym urządzeniu.

1. wchodzimy do katalogu, w którym chcemy przechowywać nasz projekt,
2. klonujemy projekt za pomocą linku: **git clone link-do-sklonowania-repozytorium**.

W przypadku, gdy wprowadziliśmy zmiany na jedym urządzeniu i chcemy zaktualizować stan projektu na innym urządzeniu, wykonujemy następujące czynności.

1. wchodzimy do katalogu projektu,
2. aktualizujemy gita o informacje odnośnie stanu repozytorium na githubie: **git fetch**,
3. aktualizujemy lokalnego gita: **git pull**.

W przypadku wystąpienia niezgodnych wersji wersjonowania pomiędzy gitem lokalnym oraz githubem, wystąpić mogą tzw. konflikty, czyli miejsca w projekcie, gdzie programista musi zadecydować jaką wersję części aplikacji git ma uznać za poprawną. W zależności od narzędzie konflikty można rozwiązywać na różne sposoby, dlatego przytoczę jedynie scenariusze, w których konieczne może okazać się zastosowanie działań forsownych.

1. Jeśli próba aktualizacji gita lokalnego się nie powiedzie, natomiast interesująca nas wersja programu znajduje się na githubie, najprościej jest po prostu usunąć lokalny katalog z projektem, a następnie ponownie go sklonować.
2. Jeśli próba wypchnięcia zmian na githuba się nie powiedzie, natomiast interesująca nas wersja programu znajduje się na lokalnym gicie, to możemy dokonać wypchnięcia siłowego: **git push --force origin nazwa-gałęzi**.

1.2 Złożoność obliczeniowa

Złożoność obliczeniowa jest miarą wzrostu liczby operacji wykonywanych przez algorytm w miarę wzrostu rozmiaru danych. W praktyce miara ta zakłada jedynie asymptotyczną ocenę wzrostu, pomijając czynniki mniej znaczące lub stałe. Stosowana jest głównie z uwagi na niezależność od środowiska wykonującego algorytm oraz trudności w szacowaniach opartych na fizycznych aspektach obliczeń takich jak częstotliwość cyklów zegarowych lub opóźnienia na poszczególnych komponentach jednostki wykonującej operacje.

1.2.1 Symbol O

Niech $g : R_{\geq 0} \to R_{\geq 0}$ będzie funkcją zmiennej x. Oznaczmy przez $O(g)$ zbiór funkcji $f : R_{\geq 0} \to R$ takich, że dla pewnego $c \in R_{\geq 0}$ oraz $x_0 \in R_{\geq 0}$ mamy $f(x) \leq cg(x)$ dla wszystkich $x \geq x_0$. Równoważnie:

$$f(x) = O(g(x)) \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = c \geq 0.$$

1.2.2 Symbol O - w praktyce

- $f(x) = 2x + 100 = O(x)$,
- $f(x) = 3x^2 + 16x = O(x^2)$,
- $f(x) = 5xlog_4(x) = O(xlog(x))$,
- $f(x) = \frac{1}{x^2-4} = O(1)$,
- $f(x) = 72 = O(1)$,
- $f(x) = x + log_2(x) + 120log_4(log_2(x)) = O(x)$,
- $f(x) = x^x + x! = O(x^x)$.

1.2.3 Symbol o

Niech $g : R_{\geq 0} \to R_{\geq 0}$ będzie funkcją zmiennej x. Oznaczmy przez $o(g)$ zbiór funkcji $f : R_{\geq 0} \to R$ takich, że dla dowolnego $c \in R_{>0}$ istnieje $x_0 \in R_{\geq 0}$ takie, że $f(x) < cg(x)$ dla wszystkich $x \geq x_0$. Równoważnie:

$$f(x) = o(g(x)) \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = 0.$$

1.2.4 Symbol o - w praktyce

- $f(x) = 2x + 100 = o(x^2)$,
- $f(x) = 3x^2 + 16x = o(x^8)$,
- $f(x) = 5xlog_4(x) = o(x^2)$,
- $f(x) = \frac{1}{x^2-4} = o(1)$,
- $f(x) = 72 = o(n)$,
- $f(x) = x + log_2(x) + 120log_4(log_2(x)) = o(xlog(x))$,
- $f(x) = x^x + x! = o(x^{x!})$.

1.2.5 Symbol Ω

Niech $g : R_{\geq 0} \to R_{\geq 0}$ będzie funkcją zmiennej x. Oznaczmy przez $\Omega(g)$ zbiór funkcji $f : R_{\geq 0} \to R$ takich, że dla pewnego $c \in R_{\geq 0}$ oraz $x_0 \in R_{\geq 0}$ mamy $g(x) \leq cf(x)$ dla wszystkich $x \geq x_0$. Równoważnie:

$$f(x) = \Omega(g(x)) \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = c > 0 \vee \lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty.$$

1.2.6 Symbol Ω - w praktyce

- $f(x) = 2x + 100 = \Omega(x)$,
- $f(x) = 3x^2 + 16x = \Omega(x^2)$,
- $f(x) = 5xlog_4(x) = \Omega(log(x))$,
- $f(x) = \frac{1}{x^2-4} = \Omega(\frac{1}{x^{100}})$,
- $f(x) = 72 = \Omega(1)$,
- $f(x) = x + log_2(x) + 120log_4(log_2(x)) = \Omega(1)$,
- $f(x) = x^x + x! = \Omega(x!)$.

1.2.7 Symbol ω

Niech $g : R_{\geq 0} \to R_{\geq 0}$ będzie funkcją zmiennej x. Oznaczmy przez $\omega(g)$ zbiór funkcji $f : R_{\geq 0} \to R$ takich, że dla dowolnego $c \in R_{\geq 0}$ istnieje $x_0 \in R_{\geq 0}$ takie, że $g(x) < cf(x)$ dla wszystkich $x \geq x_0$. Równoważnie:

$$f(x) = \omega(g(x)) \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty.$$

1.2.8 Symbol ω - w praktyce

- $f(x) = 2x + 100 = \omega(log(x))$,
- $f(x) = 3x^2 + 16x = \omega(x)$,
- $f(x) = 5xlog_4(x) = \omega(log(x))$,
- $f(x) = \frac{1}{x^2-4} = \omega(\frac{1}{x^3})$,
- $f(x) = 72 = \omega(\frac{1}{x})$,
- $f(x) = x + log_2(x) + 120log_4(log_2(x)) = \omega(log(x))$,
- $f(x) = x^x + x! = \omega(x^{100})$.

1.2.9 Symbol Θ

Niech $g : R_{\geq 0} \to R_{\geq 0}$ będzie funkcją zmiennej x. Oznaczmy przez $\Theta(g)$ zbiór funkcji $f : R_{\geq 0} \to R$ takich, że dla pewnych $c_1, c_2 \in R_{\geq 0}$ oraz $x_0 \in R_{\geq 0}$ mamy $c_1 g(x) \leq f(x) \leq c_2 g(x)$ dla wszystkich $x \geq x_0$. Równoważnie:

$$f(x) = \Theta(g(x)) \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = c > 0.$$

1.2.10 Symbol Θ - w praktyce

- $f(x) = 2x + 100 = \Theta(x)$,
- $f(x) = 3x^2 + 16x = \Theta(x^2)$,
- $f(x) = 5xlog_4(x) = \Theta(xlog(x))$,
- $f(x) = 72 = \Theta(1)$,
- $f(x) = log(log(x)) = \Theta(log(log(x))$,
- $f(x) = 2^{x+1} = \Theta(2^x)$,
- $f(x) = 2^{1000000} = \Theta(1)$.

1.3 Szacowanie wzorcowej złożoności

1.3.1 Koncept

Popularną strategią przydatną w procesie projektowania wzorcowego rozwiązania danego problemu algorytmicznego zadanego na Olimpiadzie Informatycznej jest szacowanie pożądanej złożoności. Każde zadanie posiada jasno sprecyzowane dane wejściowe, a w szczególności ich rozmiar, co może pozwolić na określenie złożoności rozwiązania wzorcowego, poprzez oszacowanie maksymalnej liczby operacji jaką potencjalnie może wykonać program, skonfrontowanie jej z limitami czasowymi i szybkością procesora, a następnie dopasowanie do jednej ze standardowych postaci funkcji wyrażającej złożoność.

Dla przykładu podane są najpopularniejsze złożoności wzorcowych rozwiązań zadań z Olimpiady Informatycznej (podane w kolejności rosnącej):

- $O(n)$
- $O(nlog(n))$
- $O(n\sqrt(n))$
- $O(n^2)$
- $O(n^3)$
- $O(2^n)$

1.3.2 Szacowanie

Biorąc pod uwagę specyfikację procesorów używanych do wykonywania zadań na środowiskach olimpiady w celu ich testowania (taktowanie 2GHz), Możemy założyć, że w przeciągu jednej sekundy będzie on w stanie wykonać około 2 miliardy poleceń. Z uwagi na architekturę procesorów, w szczególności fakt, że instrukcje znane nam z języka C++ często składają się z wielu podstawowych instrukcji procesora, bezpieczniej jest przyjąć, że procesor będzie w stanie przetworzyć parędziesiąt milionów instrukcji języka C++ na sekundę. Idąc dalej, jeśli chcielibyśmy zastosować powyższe szacowanie w kontekście złożoności obliczeniowej, zależnej od wielkości danych wejściowych, trzeba wziąć pod uwagę wszelakie stałe, które pojęcie złożoności pomija, tak więc ostatecznym szacowaniem, które należałoby zastosować jest kilka milionów operacji na sekundę.

1.3.3 Przykład

Dla przykładu weźmy pod uwagę zadanie, w którym wielkość danych określona jest przez $n \leq 200000$, natomiast algorytm powinien się wykonywać co najwyżej 5 sekund. Opierając się na powyższych założeniach, możemy więc przyjąć, że w przeciągu 5 sekund będzie on w stanie wykonać około kilkanaście milionów operacji. Spróbujmy dopasować liczbę operacji w przypadku największej możliwej wartości n ($n = 200000$) do jednej z typowych funkcji wyrażających złożoność:

- $O(n^2)$: $200000^2 = 4 * 10^{10}$, co znacznie wychodzi poza zakres kilkunastu milionów.
- $O(n\sqrt{n})$: $200000 * \sqrt{200000} \approx 450 * 200000 = 9 * 10^7$, co daje wynik o niecały rząd wielkości za duży.
- $O(nlog(n))$: $200000 * log(200000) \approx 15 * 200000 = 3 * 10^6$, co mieści się w naszych szacowaniach.
- $O(n)$, równe po prostu 200000 jest niedoszacowaniem o około 2 rzędy wielkości.

Po powyższej analizie można wywnioskować, że docelową złożonością rozwiązania, której prawdopodobnie spodziewają się autorzy zadania jest złożoność $O(nlog(n))$.

1.3.4 W praktyce

W praktyce ograniczenia czasowe zadań zazwyczaj mieszczą się w przedziale od jednej do kilkunastu sekund. Oznacza to, że dla poszczególnych rozmiarów danych wejściowych możemy przyporządkować złożoności, które z dużym prawdopodobieństwem mogą okazać się złożonościami docelowymi:

- $[1 - 10] : O(n!)$
- $[10 - 15] : O(3^n)$ / $O(2^n)$
- $[15 - 20] : O(2^n)$
- $[20 - 100] : O(n^4)$
- $[100 - 300] : O(n^3)$
- $[300 - 1000] : O(n^2 log(n))$
- $[1000 - 5000] : O(n^2)$
- $[5000 - 50000] : O(n\sqrt{n})$ / $O(nlog^2(n))$
- $[50000 - 500000] : O(nlog(n))$
- $[500000 - 1000000] : O(nlog(n))$ / $O(n)$
- $[1000000+] : O(n)$

Warto zaznaczyć, że jeśli rozwiązanie posiada nienaturanie niskie albo wysokie stałe, to powyższe szacowania mogą okazać się błędne. Należy więc polegać przede wszystkim na swojej własnej intuicji.

2 STL

STL - Standard Template Library jest biblioteką będącą zbiorem gotowych szablonów przydatnych algorytmów, struktur danych oraz innych narzędzi, które przydają się w pisaniu kodu. Narzędzie to umożliwia pominięcie implementacji niekiedy bardzo złożonego kodu, oferując gotowe rozwiązania, stworzone oraz rozwijane przez specjalistów w tej dziedzinie.

Aby skorzystać z całości STLa należy dołączyć do programu bibliotekę **bits/stdc++.h** poprzez umieszczenie w programie dyrektywy #**include**<**bits/stdc++.h**> W ramach uproszczenia przestrzeni nazw zaleca się także dodanie linii **using namespace std;**, dzięki której uniknąć można każdorazowego pisania przedrostka **std::** przed nazwą struktury bądź funkcji.

2.1 Przydatne narzędzia

2.1.1 Auto

Auto jest słowem kluczowym, którego można użyć w zastępstwie za typ zmiennej, pod warunkiem, że w momencie deklaracji następuje także przypisanie wartości. Spowoduje to wydedukowanie typu zmiennej przez kompilator na podstawie kontekstu. Przykładowo:

```
auto pewnaLiczbaCalkowita = 311;
```

Dzięki auto możemy nieprzejmować się jakiego typu danymi operujemy; w szczególności gdy nie jesteśmy tego typu pewni na przykład w momencie, gdy używamy nieznanej nam dotychczas funkcji języka. Minusem tego rozwiązania jest zmniejszona czytelność kodu, ponieważ czytając kod, każdorazowo musimy zastanawiać się jaki typ faktycznie kryje się pod słowem kluczowym auto. Szczególne zastosowanie auto znajduje w procesie iterowania po strukturach danych, które przedstawione będą w dalszej części rozdziału. Przykładowo chcą przeiterować się po wszystkich elementach danej struktury **nazwaStruktury**, możemy posłużyć się nastepującą pętlą:

```
for(auto elementStruktury : nazwaStruktury) {
    //elementStruktury jest elementem naszej struktury rozpatrywanym w danej iteracji
}
```

2.1.2 Komparator

Komparatory nie są stricte elementem STL'a, jednakże znajdują szerokie zastosowanie w połączeniu ze strukturami oraz funkcjami udostępnianymi przez STLa. Formalnie jest on funkcją typu bool, która na wejściu otrzymuje dwa elementy, natomiast na wyjściu zwraca informację czy element pierwszy jest mniejszy od drugiego. Dla przykładu komparator porządkujący liczby według ich wartości modulo 3:

```
bool komparatorMod3(int a, int b) {
    return (a % 3) < (b % 3);
}
```

Aby użyć komparatorów, najczęściej należy podać je tworzonej strukturze w postaci argumentu w operatorze diamentowym (<**..., komparator**>) albo jako argument do przyjmującej go funkcji (**(..., komparator)**). Istnieją także inne sposoby tworzenia komparatorów, takie jak wyrażenia lambda czy struktury z przeciążonymi operatorami, jednakże nam w zupełności wystarczy powyższa forma. Ciekawskich ponownie odsyłam do innych źródeł.

2.1.3 Iterator

Iterator sam w sobie nie ma zastosowania. Struktura ta znajduje zastosowanie dopiero w połączeniu z odpowiednimi strukturami danych. Iterator jest swojego rodzaj wskaźnikiem, który możemy poruszać między elementami struktury liniowej, a także odczytywać wartość elementu, na który aktalnie wskazuje. Iterator dla danej struktury danych deklarujemy następująco:

```
struktura::iterator nazwa;
```

Dla przykładu:

```
set<int>::iterator nazwaIteratora;
```

Aby zastosować iterator na wybranej strukturze danych, należy ustawić go na jednej z pozycji, do których wybrane struktury oferują dostęp; przykładowo dla kolejki będzie to **front()**, natomiast dla zbioru **begin()** oraz **end()**. Przykładowo chcąc ustawić iterator na początek zbioru:

```
set<int>::iterator nazwaIteratora = zbior.begin();
```

Iteratorem poruszamy tak samo jak klasycznym wskaźnikiem; dodając albo odejmując od niego wartość równą liczbie elementów o którą chcemy go poruszyć. Aby odczytać wartość elementu, na którym obecnie zatrzymał się iterator należy odwołać się do jego wartości identycznie jak w przypadku wskaźnika. Niech zbior będzie zbiorem o strukturze (1, 2, 3):

```
set<int>::iterator nazwaIteratora = zbior.begin();
*nazwaIteratora // odczytanie wartości 1
nazwaIteratora++;
*nazwaIteratora // odczytanie wartości 2
nazwaIteratora--;
*nazwaIteratora; odczytanie wartości 1
nazwaIteratora+=2;
*nazwaIteratora; odczytanie wartości 3
```

2.2 Struktury nieuporządkowane

2.2.1 Pair

Para jest strukturą danych oferującą możliwość agregacji dwóch elementów, niekoniecznie tego samego typu. Samą parę deklaruje się w następujący sposób:

```
pair<typ1, typ2> nazwa;
```

Na przykład:

```
pair<int, string> nazwaPary;
```

Aby odwołać się do poszczególnych elementów pary, należy odnieść się do elementów **first** oraz **second** w następujący sposób:

```
nazwaPary.first = 4;
nazwaPary.second = "cztery";
```

Ponadto w tworzeniu par przydatna może okazać się metoda **make_pair** działająca jako konstruktor:

```
pair<int, string> nazwaPary = make_pair(4, "cztery");
```

Dozwolona jest także inicjalizacja przy użyciu krotki rzędu 2:

```
pair<int, string> nazwaPary = {1, "cztery"};
```

W szczególności stworzyć można pary kaskadowe, które mogą pomieścić więcej elementów, kosztem zagłębienia struktury. Poniżej przykład implementacji oraz użycia pary pozwalającej na przechowywanie czterech elementów, przydatnej chociażby w implementacji niektórych struktur grafowych:

```
pair<pair<int, double>, pair<bool, string> > czteroPara;
czteroPara = make_pair(make_pair(1, 2.5), make_pair(true, "tak"));
pair.first.first = 2;
pair.first.second = 3.5;
pair.second.first = false;
pair.second.second = "nie";
```

Pary pożna porównywać poprzez użycie standardowych komparatorów. Priorytetowo porównane zostaną pierwsze elementy, a następnie drugie. Przykładowo:

```
{1, 3} > {1, 2}
{2, 1} > {1, 10}
{5, 8} < {6, 1}
{1, 2} = {1, 2}
```

2.2.2 Vector

Vector jest strukturą zbliżoną do klasycznej tablicy. Oferuje on możliwość zapisu oraz odczytu wartości dowolnego elementu. W przeciwieństwie do tablicy jego rozmiar jest elastyczny - vector potrafi dynamicznie zmieniać rozmiar w zależności od zapotrzebowania, a także na życzenie, poprzez wykorzystaniem odpowiednich komend. Wadą vectora jest jego wydajność, która jest mniejsza, niż w przypadku klasycznej tablicy. Aby zadeklarować vector należy posłużyć się następującą składnią:

```
vector<typ> nazwa;
```

Na przykład:

```
vector<int> nazwaVectora;
```

W celu inicjalizacji vectora o danym rozmiarze początkowym należy sprecyzować rozmiar w nawiasach na końcu deklaracji:

```
vector<int> nazwaVectora(128);
```

Odwoływanie się do danego elementu vectora przebiega identycznie jak w przypadku tablicy:

```
nazwaVectora[12] = 21;
```

Dodatkowo możliwe jest dodawanie oraz usuwanie nowych elementów na początku, jak i końcu vectora, za pomocą komend **push_back**, **pop_back** oraz **push_front**, **pop_front**. Przykładowo dla vectora o wstępnej strukturze [1, 2, 3, 4] po wykonaniu komend

```
nazwaVectora.push_back(5)
nazwaVectora.pop_front()
nazwaVectora.pop_back()
nazwaVectora.push_back(6)
nazwaVectora.push_front(0)
```

otrzymamy następujący vector: [0, 2, 3, 4, 6]. Co więcej vector oferuje możliwość odczytu aktualnej liczby elementów, które się na nim znajdują - komenda **size()**, a także zapytanie czy jest on w danym momencie pusty - komenda **empty()**. Możliwe jest też usunięcie wszystkich elementów vectora komendą **clear()**, jednakże usuwa ona tylko same wartości, a nie pamięć przydzieloną vectorowi. W celu usunięcia pamięci przydzielonej vectorowi służy komenda **shrink_to_fit()**. Warta uwagi jest także funkcja **reverse()** umożliwiająca odwrócenie kolejności elementów w vectorze.

2.2.3 Queue

Queue, a właściwie kolejka jest strukturą opartą na zasadzie FIFO (first in - first out). Oznacza to, że jak w przypadku klasycznej kolejki, element który trafił do niej na początku, opuści ją jako pierwszy. Struktura oparta jest o listę wiązaną, co oznacza, że jesteśmy w stanie odwoływać się jedynie do jej pierwszego i ostatniego elementu. Zaletą tego rozwiązania jest dynamiczność struktury, która w dowolnym momencie użytkowania zajmuje jedynie tyle pamięci, ile w danej chwili potrzebuje, adaptując się każdorazowo po zmianie liczby elementów w niej zawartej. Kolejkę deklarujemy w następujący sposób:

```
queue<typ> nazwa;
```

Na przykład:

```
queue<int> nazwaKolejki;
```

Aby dodawać oraz usuwać elementy z kolejki, stosuje się komendy **push()** oraz **pop()**. Ponadto kolejka pozwala na odczyt elementu aktualnie znajdującego się na jej początku, przy pomocy komendy **front()**:

```
queue<int> nazwaKolejki;
nazwaKolejki.push(1);
nazwaKolejki.push(2);
nazwaKolejki.push(3);
nazwaKolejki.front(); // zwraca wartość 1
nazwaKolejki.pop();
nazwaKolejki.front(); // zwraca wartość 2
nazwaKolejki.push(4)
nazwaKolejki.front(); // zwraca wartość 2
```

Ponadto tak, jak w przypadku vectora, kolejka także pozwala na dostęp do aktualnie przechowywanej liczby elementów - komenda **size()** oraz zapytanie czy struktura jest pusta - komenda **empty()**.

2.2.4 Stack

Stack znany także jako stos jest strukturą bardzo zbliżoną do kolejki. Jego struktura oparta jest także na koncepcie listy wiązanej, przez co dostępny jest jedynie jego element wierzchni. Sam stos charakteryzuje się architekturą LIFO (last in, first out), dzięki której, jak w przypadku chociażby stosu książek, element który został dodany do stosu jako ostatni, opuści strukturę jako pierwszy. Stos deklarujemy w następujący sposób:

```
stack<typ> nazwa;
```

Na przykład:

```
stack<int> nazwaStosu;
```

Podobnie jak w przypadku kolejki, stos udostępnia metody pozwalające na dodanie do niego elementu - komenda **push()** oraz na usunięcie - komenda **pop()**. Odpowiednikiem komendy front, znanej z kolejki, jest w tym przypadku komeda **top()**, pozwalająca na odczyt elementu znajdującego się na szczycie stosu. Przykładowo:

```
stack<int> nazwaStosu;
nazwaStosu.push(1);
nazwaStosu.push(2);
nazwaStosu.push(3);
nazwaStosu.top(); // zwraca wartość 3
nazwaStosu.pop();
nazwaStosu.top(); // zwraca wartość 2
nazwaStosu.push(4);
nazwaStosu.top(); // zwraca wartość 4
```

Identycznie jak w przypadku kolejki oraz vectora, mamy dostęp do liczby elementów aktualnie wchodzących w skład stosu - komenda **size()** oraz zapytania czy struktura jest pusta - komenda **empty()**.

2.2.5 Deque

Deque jest strukturą zachowującą się dokładnie tak samo jak kolejka, z tą różnicą, że operacje dodawania, usuwania oraz odczytu elementu można wykonywać na obydwu jej końcach. Bikolejkę deklaruje się w następujący sposób:

```
deque<typ> nazwa;
```

Dla przykładu:

```
deque<int> nazwaBikolejki;
```

Operacje dodania, usunięcia oraz odczytu elementu z przodu bikolejki dokonuje się kolejno komendami **push_front()**, **pop_front()** oraz **front()**. Operacje wykonywane na końcu bikolejki noszą analogiczne nazwy, tylko z wykorzystaniem słowa back; **push_back()**, **pop_back()** oraz **back()**. Na przykład:

```
deque<int> nazwaBikolejki;
nazwaBikolejki.push_back(1); // (1)
nazwaBikolejki.push_front(2); // (1, 2)
nazwaBikolejki.push_back(3); // (3, 1, 2)
nazwaBikolejki.push_front(4); // (3, 1, 2, 4)
nazwaBikolejki.front(); // zwraca wartość 4
nazwaBikolejki.pop_back(); // (1, 2, 4)
nazwaBikolejki.pop_front(); // (1, 2);
nazwaBikolejki.back(); // zwraca wartość 1
```

Identycznie jak w przypadku kolejki klasycznej, dla bikolejki dostępne są także funkcje **size()** oraz **empty()**.

2.2.6 Bitset

Bitset jest strukturą będącą tablicą o stałym rozmiarze przechowywującą jedynie zera i jedynki. W przeciwieństwie do tablicy przechowywującej wartości typu bool, bitset zapisuje każdą wartość na jednym bicie, a nie bajcie, dzięki czemu zajmuje on 8 razy mniej miejsca oraz wykonuje operacje 8 razy szybciej. Bitset deklaruje się następująco:

```
bitset<rozmiar> nazwa;
```

Na przykład:

```
bitset<32> nazwaBitsetu;
```

Co więcej, podczas deklaracji bitsetu można przypisać do niego wartość początkową równą binarnej reprezentacji danej zmiennej:

```
int nazwaZmiennej = 311;
bitset<32> bitsetZeZmiennej(nazwaZmiennej);
```

Możliwa jest także operacja odwrotna - aby zamienić bitset na zmienną zapisaną w systemie dziesiątkowym należy zastosować funkcję **to_ulong()**, która zamienia zawartość bitsetu na zmeinną typu unsigned long:

```
bitset<32> nazwaBitsetu(311);
int wartoscBitsetu = (int) nazwaBitsetu.to_ulong(); // dodatkowo castujemy wynik na inta.
```

Proces odwołania do wybranego bitu struktury przebiega analogicznie jak w przypadku klasycznej tablicy. Przykładowo, aby odwołać się do bitu drugiego (odpowiadającego wartości 2^2 w zapisie dziesiątkowym), należy zastosować następujacą składnię:

```
nazwaBitsetu[2];
```

Ponadto bitset oferuje także metodę **test(i)**, która zwraca wartość typu bool w zależności, czy i-ty bit jest w tym momencie zapalony. Podobna składniowo jest także metoda **flip(i)**, zmieniająca wartość i-tego bitu na odwrotną:

```
bitset<3> nazwaBitsetu(0); // początkowo wszystkie bity są zgaszone
nazwaBitsetu.flip(1); // 010
nazwaBitsetu.flip(2); // 110
nazwaBitsetu.test(1); // true
nazwaBitsetu.test(0); // false
nazwaBitsetu.to_ulong(); // 6
```

Przydatna może się okazać także funkcja **count()**, która zwraca liczbę aktualnie zapalonych (mających wartość 1) bitów w bitsecie:

```
bitset<8> nazwaBitsetu(100) // 01100100
nazwaBitsetu.count() // 3
```

Prawdziwa potęga bitsetów ukazuje się w momencie stosowania na nich operacji logicznych. Istnieje bowiem możliwość przeprowadzenia logicznego ORa, ANDa oraz XORa na bitsecie w czasie ośmiokrotnie mniejszym, niż w przypadku klasycznej tablicy:

```
bitset<8> bitset1(120); // 01111000
bitset<8> bitset2(167); // 10100111
bitset1 | bitset2 // OR daje wynik 11111111
bitset1 & bitset2 // AND daje wynik 00100000
bitset1 ^ bitset2 // XOR daje wynik 11011111
```

2.3 Struktury uporządkowane

2.3.1 Priority queue

Priority queue, czyli kolejka priorytetowa jest strukturą zbliżoną działaniem do kolejki klasycznej. Jedyna różnica polega na tym, że kolejkę każdorazowo opuszcza element o największej wartości, a nie element najwcześniej dodany. Wewnętrznie struktura kolejki priorytetowej znacznie różni się od kolejki klasycznej, gdyż zamiast listy wiązanej, priority queue opiera się na kopcu binarnym. Deklaracja kolejki priorytetowej przebiega następująco:

```
priority_queue<typ> nazwa;
```

Przykładowo:

```
priority_queue<int> nazwaKolejkiPriorytetowej;
```

W związku z architekturą kopca biranrego operacje dodania elementu - komenda **push()** oraz usunięcia elementu - komenda **pop()** obarczone są złożonością rzędu **O(log(n))**, gdzie n to liczba elementów obecnie znajdująca się w strukturze. Wartość elementu największego dostępna jest przy pomocy komendy **top()**, a jej złożoność, podobnie jak w przypadku operacji na poprzednich strukturach, jest stała. Na przykład:

```
priority_queue<int> nazwaKolejkiPriorytetowej;
nazwaKolejkiPriorytetowej.push(1);
nazwaKolejkiPriorytetowej.push(11);
nazwaKolejkiPriorytetowej.push(3);
nazwaKolejkiPriorytetowej.top(); // zwraca wartość 11
nazwaKolejkiPriorytetowej.pop();
nazwaKolejkiPriorytetowej.top(); // zwraca wartość 3
nazwaKolejkiPriorytetowej.push(4);
nazwaKolejkiPriorytetowej.top(); // zwraca wartość 4
```

Analogicznie do poprzednich struktur, kolejka priorytetowa także oferuje metody **size()** oraz **empty()**. W przypadku, gdy potrzebujemy kolejki priorytetowej, która priorytetyzuje elementy najmniejsze zamiast największych, możemy wykorzystać pewną właściwość matematyczną. Niech A będzie zbiorem wartości znajdujących się w strukturze, natomiast A^* zbiorem zawierającym elmenty przeciwne do elementów zbioru A (np. dla $A = (-1, 2, 3)$, $A^* = (1, -2, -3)$). Prawdziwe jest następujące zdanie logiczne:

$$a = max(A) \iff a = min(A^*)$$

Oznacza to, że w sytuacji, gdy chcemy priorytetyzować elementy o najmniejszej wartości, wystarczy, że do kolejki będziemy dodawać elementy do nich przeciwne, natomiast odczytując wartości z kolejki, ponownie je negować. Dla przykładu chcemy umieścić na kolejce następujące elementy: $(4, 5, 2, 1)$, aby następnie móc je pozdejmować w kolejności niemalejącej:

```
priority_queue<int> kolejkaMinimalizujaca;
kolejkaMinimalizujaca.push(-4); // negujemy umieszczane wartości
kolejkaMinimalizujaca.push(-5);
kolejkaMinimalizujaca.push(-2);
```

```
kolejkaMinimalizujaca.push(-1);
kolejkaMinimalizujaca.top() * -1; // zwraca wartość 1
kolejkaMinimalizujaca.pop();
kolejkaMinimalizujaca.top() * -1; // zwraca wartość 2
kolejkaMinimalizujaca.pop();
kolejkaMinimalizujaca.top() * -1; // zwraca wartość 4
kolejkaMinimalizujaca.pop();
kolejkaMinimalizujaca.top() * -1; // zwraca wartość 5
```

Możliwe jest także zastosowanie własnego komparatora, dzięki któremu możliwe będzie porównywanie elementów w dowolny sposób:

```
priority_queue<int, vector<int>, komparator> kolejkaZKomparatorem;
```

Warto zaznaczyć, że użycie komparatora wymusza na nas podanie enigmatycznego argumentu **vector<int>**, jednakże nie jest to istotne w danym momencie. Ciekawskich odsyłam do dokumentacji biblioteki STL.

2.3.2 Set

Set jest strukturą danych odzwierciedlającą klasyczny zbiór znany z matematyki. Jego główna właściwość polega na tym, że każdy element znajdujący się w zbiorze musi mieć unikatową wartość. Oznacza to, że zestaw elementów **(1, 2, 3)** jest poprawnym zbiorem, natomiast **(1, 2, 2)** już nie. Ponadto elementy zbioru ułożone są w ustalonej kolejności; domyślnie tworzą ciąg rosnący. Set deklaruje się następujaco:

```
set<typ> nazwa;
```

Na przykład:

```
set<int> nazwaZbioru;
```

Główną zaletą zbioru jest jego architektura, która pozwala na wykonywanie większości działań na strukturze w czasie logarytmicznym względem jego złożoności, czyli **O(nlog(n)**. Elementy dodaje się do zbioru za pomocą komendy **insert()**, natomiast usuwa komendą **erase(element)**. Co więcej element możemy usunąć także poprzez iterator; **erase(iterator)**. Dla przykładu:

```
set<int> nazwaZbioru;
nazwaZbioru.insert(1); // zbiór: {1}
nazwaZbioru.insert(2); // zbiór: {1, 2}
nazwaZbioru.insert(2); // zbiór: {1, 2}
nazwaZbioru.insert(3); // zbiór: {1, 2, 3}
nazwaZbioru.erase (2); // zbiór: {1, 3}
set<int>::iterator iteratorZbioru = nazwaZbioru.begin(); // iterator na początek zbioru
nazwaZbioru.erase(iteratorZbioru); // zbiór: {3}
```

Set oferuje także możliwość wyszukiwania elementów, które się na nim znajdują. Komenda **find(wartość)** zwraca iterator do elementu zbioru o podanej wartości. W przypadku, gdy nie uda się znaleźć elementu, zwracany jest iterator na koniec seta (**nazwaZbioru.end()**). Ponadto istnieją także komendy **lower_bound(wartość)** oraz **upper_bound(wartość)**, które w analogiczny sposób zwracają iterator do pierwszego eementu nie mniejszego, niż zadana wartość oraz pierwszego elementu większego od podanej wartości. Podobnie jak w przypadku **find()**, jeśli wyszukiwanie się nie powiedzie, zwrócony zostanie iterator na koniec seta. Przykładowo:

```
set<int> nazwaZbioru; // niech zbiór prezentuje się następująco: {1, 3, 5, 7, 9}.
nazwaZbioru.find(3) // zwróci iterator na trzeci element zbioru
*nazwaZbioru.find(3) // zwróci wartość 5
*nazwaZbioru.upper_bound(6) // zwróci wartość 7
*nazwaZbioru.upper_bound(5) // zwróci wartość 7
*nazwaZbioru.lower_bound(4) // zwróci wartość 5
*nazwaZbioru.lower_bound(5) // zwróci wartość 5
nazwaZbioru.find(12) // zwróci iterator na koniec zbioru
```

Oczywiście jak we wszystkich powyższych strukturach, set także dysponuje metodami **size()** oraz **empty()**. Dodatkowo posiada także metodę **clear()**, która usuwa ze zbioru wszystkie elementy.

Zbiór umożliwia także zastosowanie własnego komparatora, dzięki czemu można trzymać na nim elementy według interesującej nas, niestandardowej kolejności:

```
set<typ, nazwaKomparatora> niestandardowoUporzadkowanyZbior;
```

2.3.3 Multiset

Multiset jest strukrą niemalże identyczną do zbioru, z tą różnicą że multizbiór potrafi przechowywać wiele elementów o tej samej wartości. Dlatego też zarówno **(1, 2, 3)**, jak i **(1, 2, 2)** są poprawnymi multizbiorami. Udostępnia on metody analogiczne, co zwykły set. Deklaruje się go następująco:

```
multiset<typ> nazwa;
```

Na przykład:

```
multiset<int> nazwaMultizbioru;
```

Warto zaznaczyć, że komenda **erase(wartość)** w tym przypadku usunie z multisetu wszystkie elementy o podanej wartości. W przypadu, gdy interesuje nas usunięcie jedynie jednego elementu, musimy wspomóc się usuwaniem elementu po iteratorze:

```
multiset<int> nazwaMultizbioru; // niech multizbiór prezentuje się następująco: {1, 2, 2, 2, 3}
// ustalmy, że chcemy usunąć tylko jeden element o wartości dwa.
multiset<int>::iterator iteratorMultizbioru = nazwaMultizbioru.find(2); // deklarujemy iterator
// wskazujący na pierwszy z brzegu element dwa
nazwaMultizbioru.erase(iteratorMultizbioru); // usuwamy element 2, na który wskazuje iterator
// teraz multizbiór prezentuje się następujaco: {1, 2, 2, 3}
```

2.3.4 Map

Ostatnią rodziną omawianych struktur będą mapy. Mapa znana jest także pod nazwą słownika albo tabeli asocjacji, gdyż jej zadaniem jest przyporządkowywanie elementów jednego zbioru do elementów innego zbioru. Dla przykładu mapa pozwala nam na przyporządkowanie każdemu elementowi typu **string** przyporządkować element typu **int** odpowiadający jego długości. Podobnie jak pozostałe struktury uporządkowane, jej rozmiar zmienia się dynamicznie, w zależności od liczby elementów, które zawiera, co czyni ją strukturą efektywną pamięciowo. Operacje na niej wykonywane charakteryzują się złożnością **O(log(n))** względem jej rozmiaru. Mapę deklaruje się następująco:

```
map<typKluczy, typWartosci> nazwaMapy;
```

Przykładowo:

```
map<string, int> nazwaMapy;
```

Dodawanie oraz odczytywanie elementów z mapy odbywa się analogicznie, jak w przypadku klasycznej tablicy. Jedyna różnica jest taka, że zamiast podawania indeksu elementu podajemy jego klucz, który zresztą niekoniecznie musi być liczbą. Dla przykładu:

```
map<string, int> nazwaMapy;
nazwaMapy["cztery"] = 4;
nazwaMapy["dwa"] = 2;
nazwaMapy["cztery"] // zwraca wartość 4
```

Ponadto możemy także usuwać elementy z naszej mapy za pomocą komendy **erase()** oraz pytać się czy istnieje element o danym kluczu poprzez metodę **contains()**. Na przykład:

```
map<string, int> nazwaMapy; // niech będzie to mapa z poprzedniego przykładu
nazwaMapy.contains("cztery") // zwraca wartość true
nazwaMapy.contains("trzy") // zwraca wartość false
nazwaMapy.erase("cztery")
nazwaMapy.contains("cztery") // zwraca wartość true
```

2.3.5 Unordered map

Unordered map znana także jako haszmapa jest mapą nieuporządkowaną. Struktura ta swoim działaniem odzwierciedla klasyczną mapę, jednakże jej implementacja różni się znacząco. Zamiast drzew czerwono-czarnych używa on tablicy haszującej, przez co umożliwia ona przeprowadzanie na niej operacji w średnim czasie stałym $\Theta(1)$. Haczyk polega na tym, że w przypadku podania nieuporządkowanej mapie złośliwych danych, złożoność ta moze wzrosnąć do złożoności liniowej względem jej rozmiaru - $O(n)$. Należy więc używać jej z rozwagą. Deklaruje się ją następująco:

```
unordered_map<typKluczy, typWartosci> nazwa;
```

Przykładowo:

```
unordered_map<string, int> nazwaHaszmapy;
```

Zainteresowanych działaniem mapy nieuporządkowanej zachęcam do zapoznania się z tematem haszowania oraz tablicy haszującej.

2.4 Przydatne funkcje

2.4.1 Sort

Jedną z najbardziej przydatnych funkcji, które oferuje STL jest bez wątpienia funkcja **sort()**. Oferuje ona możliwość posortwania tablicy elementów według ustalonej reguły sortowania. Funkcja **sort()** Domyślnie sortuje elementy niemalejąco. Używa się jej następująco:

```
sort(tablica, tablica + rozmiarTablicy);
//ewentualnie w przypadku użycia komparatora
sort(tablica, tablica + rozmiarTablicy, komparator);
```

Dla przykładu:

```
// niech komparatorNierosnacy będzie komparatorem porządkującym elementy nierosnąco
int tab[n]; // niech tab będzie tablicą, którą chcemy posortować, natomiast n jej rozmiarem
sort(tab, tab + n); // funkcja posortuje niemalejąco tablicę tab
sort(tab, tab + n, komparatorNierosnacy) // funkcja posortuje nierosnąco tablicę tab
```

Sort potrafi także sortować między innymi wektory. W tym wypadku składnia jest nieco inna:

```
vector<int> wektor; // niech wektor będzie wektorem, który chcemy posortować
sort(wektor.begin(), wektor.end());
```

2.4.2 Permutation

STL posiada także mechanizm do generowania kolejnych permutacji danego ciągu. Do generowania następnej leksykograficznie permutacji tablicy należy użyć funkcji **next_permutation()**, natomiast do permutacji poprzedniej; komendy **prev_permutation)**. Przykładowo:

```
int tab[5] = {2, 5, 3, 7, 1};
prev_permutation(tab, tab + 5) // po wywołaniu funkcji, tab przyjmie postać {2, 5, 3, 1, 7}
next_permutation(tab, tab + 5) // po wywołaniu funkcji, tab przyjmie postać {2, 5, 7, 1, 3}
```

Funkcje te posiadają także wartość zwracaną. W przypadku, gdy istnieje permutacja poprzednia / następna, to zwracają one wartość true, w przeciwnym wypadku false. Na przykład:

```
int tab[5] = {1, 2, 3, 5, 4};
next_permutation(tab, tab + 5) // zwróci true
//w tym momencie tab prezentuje się następująco: {1, 2, 3, 4, 5}
next_permutation(tab, tab + 5) // zwróci false, ponieważ nie ma następnej permutacji
```

Ciekawostką jest, że istnieje także trzecia komenda **random_shuffle()**, która dokonuje losowej permutacji ciągu wartości.