Notatki z Algorytmów i Struktur Danych

2024-10-08 – 2025-01-21

Spis treści

Przedmowa

To są notatki z przedmiotu algorytmy i struktury danych prowadzonego na kierunku IAD w 2024/2025 roku przez dr Tomasza Krajkę. Treści obejmują 14 wykładów.

Ponieważ kod podany na wykładach nie kompiluje się, to był poprawiony bez zmian logiki wewnętrznej funkcji/klas. Głównym powodem dla błędów kompilacji kodu doktora był magiczny uniwersalny typ `TypKlucza` który niby jest używany ale nie był w żadnym miejscu zdefiniowany. Wszędzie gdzie ten typ pojawia się są używane szablony, jednak można zamienić ten typ na konkretny (np. `int`) i usunąć wszystkie szablony.

[1] https://creativecommons.org/publicdomain/zero/1.0/deed.pl

2024-10-08

1 Podstawy programistyczne: Instrukcje i typy danych

Techniki programowania dzielą się na dwie podstawowe grupy: strukturalne i obiektowe.

1.1 Techniki Strukturalne

Oparte o trzy podstawowe instrukcje:

1. sekwencja (konkatenacja instrukcji)
2. selekcja (instrukcja warunkowa)
3. iteracja (pętla)

oraz dane i struktury danych na których one operują.

1.2 Techniki Obiektowe

Następuje w nich połączenie danych z operacjami wykonywanymi na nich.

1.3 Strukturalne techniki programowania

Oparte są o zdania rozkazujące(instrukcje), których złożenia tworzą algorytmy Böhm i Jacopini udowodnili, że do napisania czegokolwiek wystarczy trzech instrukcji.

1.3.1 Sekwencja (konkatenacja)

Wykonaj instrukcję 1, a następnie wykonaj instrukcję 2:

```
instrukcja 1;
instrukcja 2;
```

1.3.2 Selekcja

Sprawdź prawdziwość wyrażenia logicznego(warunku) i w przypadku jego prawdziwości wykonaj instrukcję 1, w przeciwnym razie instrukcję 2

```
if (warunek) {
  instrukcja 1;
} else {
  instrukcja 2;
}
```

1.3.3 Iteracja

Sprawdź prawdziwość wyrażenia logicznego (warunku) i w przypadku jego fałszywości zakończ zakończ instrukcję w przeciwnym razie wykonaj instrukcje i ponownie wykonaj całą iterację i ewentualne wykonanie instrukcji. Instrukcja ta jest wykonywane dopóty, dopóki warunek pozostaje prawdziwy.

```
while (warunek) {
  instrukcja;
}
```

1.3.4 Struktury danych

W strukturalnych technikach programowania algorytmy operują na danych. Ponieważ przechowywane informacje mogą mieć różną postać, dlatego w językach programowania wprowadzono różne typy danych. Nie istnieje uniwersalny typ danych umożliwiający przechowywanie każdego rodzaju informacji.

Aby wyczerpująco scharakteryzować typ danych należy oprócz zbioru przyjmowanych przez ten typ wartości (Ω) określić także zbiór operacji jakie można wykonywać na danym typie danych Φ oraz dopuszczonych relacji Ψ.

Typ danych = (Ω, Φ, Ψ)

Typy danych dzielą się na

1. typy proste: stanowią informacje jednorodną co do swojej natury i niepodzielną,
2. typy strukturalne: składają się z wielu danych zorganizowanych w pewną strukturę,
3. typy wskaźnikowe: stanowią jedynie adres/odwołanie do pewnego obszaru w pamięci w którym przechowywana jest faktyczna informacja.

1.4 Proste typy danych

1. logiczny bool =
   $(\{\text{true}, \text{false}\}, \{!, ||, \&\&, \ldots\}, \varnothing)$

2. całkowity int =
   (podzbiór $\mathbb{Z}, \{+, *, -, /, ++, --, \&, |, \sim, \gg, \ll\}, \{<, >, !=, ==, \leq, \geq\}$)
3. rzeczywisty float =
   (Podzbiór $\mathbb{Q}, \{+, *, -, /\}, \{<, >, !=, ==, \leq, \geq\}$)
4. znakowy char =
   ($\{\mathbb{Z}_{256}$ lub więcej np. dla wchar$\}, \{+, *, -, /, ++, --, \&, |, \sim, \gg, \ll\}, \{<, >, !=, ==, \leq, \geq\}$)
5. typ wyliczeniowy enum =
   (podzbiór $\mathbb{Z}, \{+, *, -, /, ++, --, \&, |, \sim, \gg, \ll\}, \{<, >, !=, ==, \leq, \geq\}$)

   ```
   enum liczby = {
     zero,
     jeden,
     piec = 5,
     szesc,
     dziesiec = 10
   }
   ```

6. typ pusty void =
   $(\varnothing, \varnothing, \varnothing)$

2024-10-15

1.5 Strukturalne typy danych

1.5.1 Tablice statyczne

Służą do przechowywania wielu danych tego samego typu. Tablice statyczne są strukturami jednorodnymi oraz indeksowanymi. Do elementów tablicy odwołujemy się za pomocą `operator[]`. Tablice statyczne są tworzone w momencie ich deklaracji.

Z matematycznego punktu widzenia tablica stanowi odwzorowanie typu indeksowego w typ składowy

$$\text{tablica} : D \to P$$

gdzie

- D – zbiór indeksowy
- P – typ wartości

Zbiorem wartości takiej tablicy jest zbiór wszystkich takich funkcji, którego moc jest równa $|P^D|$

```
int tablica1[5];
int tablica2[4][5]; // <-- takich deklaracji się unika według doktora

// dzielimy sobie w głowie tę tablicę
// wyznaczamy wiersz przez resztę z dzielenie i kolumnę przez dzielenie
int tablica3[20];
tablica1[3] = 4;
cout << tablica2[2];
```

Za pomocą tablic można przechowywać także łańcuchy znaków

```
char napis[14] = "To jest napis";
//                           ^^^
// łańcuch kończy się znakiem zwanym «null character»
// ten znak jest niewidoczny, oznaczany jest \0

char napis[14] = {'T', 'o', ' ', 'j', 'e', 's', 't',
                  ' ', 'n', 'a', 'p', 'i', 's', '\0'};
```

1.5.2 Rekordy (struktury)

Rekordy służą do przechowywania informacji złożonej, niejednorodnej, ale o stałej długości. Ich składowe nie są ponumerowane, lecz ponazywane (identyfikatorowane). Dostęp do składowych daje operator kropka «.». Z matematycznego punktu widzenia rekord stanowi iloczyn kartezjański jego typów składowych $T_1 \times T_2 \times \cdots \times T_n$

```
struct Osoba {
  int wiek;
  bool kobieta;
  double wzrost;
};

Osoba Jan;
Jan.wiek = 20;
Jan.kobieta = false;
Jan.wzrost = 175.2;
```

W języku C++ wprowadzono typy obiektowe, które stanowią znaczne rozszerzenie struktur, w związku z czym są coraz rzadziej stosowane.

1.5.3 Obiekty statyczne

Typ obiektowy, podobnie jak typ rekordowy może przechowywać informacje niejednorodne co do swojej struktury, ale dodatkowo udostępnia możliwość zdefiniowania operacji jakie można wywoływać na danych tego typu (poprzez jego metody). Obiekty statyczne tworzone są w chwili deklaracji poprzez wywołania specjalnej jego metody(konstruktora).

1.6 Wskaźnikowe typy danych

1.6.1 Tablice tworzone dynamicznie

Tak jak tablice statyczne mogą przechowywać informację jednorodną co do struktury, jednak nie muszą być tworzone w momencie deklaracji zmiennej wskaźnikowej. Co więcej pozwalają na tworzenie i usuwanie wielu różnych tablic o różnych rozmiarach przypisanych do tej samej zmiennej wskaźnikowej.

1.6.2 Wskaźniki

Przechowują adres zmiennej w której znajduje się informacja. Zmienne wskaźnikowe mogą przechowywać adresy różnych typów. Przy korzystaniu ze zmiennych wskaźnikowych istotną rolę pełnią operatory

- * – operator wyłuskania zmiennej
- & – operator pozyskania adresu

Zmiennej wskaźnikowej możemy nadać wartość specjalną `NULL` – pusty adres.

Zmiennej wskaźnikowej można nadać adres zmiennej istniejącej już w pamięci albo utworzyć pod wolnym adresem który zostanie przypisany do tej zmiennej, nową zmienną (zmienną utworzoną dynamicznie)

Do dynamicznego tworzenie zmiennej służą operatory `new` i `delete`, a w przypadku tworzenia i usuwania zmiennych tablicowych operatory `new[]` i `delete[]`

```
int Zmienna1 = 5;

int *Addr1;
int *Addr2;
int *Tablica;

Addr1 = &Zmienna1;
```

```
*Addr1 = 3;

Addr2 = new int;
*Addr2 = 3;
// Po usunięciu adres nadal jest taki jak był do usunięcia lecz my
// już nie mamy uprawnień do zapisywania pod tym adresem
delete Addr2;

Tablica = new int[7];
Tablica[4] = 3;

delete[] Tablica;
```

1.6.3 Typy plikowe

Zmienne wskazujące na obszar w pamięci poza pamięcią operacyjną – na nośniku zewnętrznym. Do obsługiwania typu plikowego obiekty klas `ifstream` (odczyt), `ofstream` (zapis), `fstream` (odczyt oraz zapis);

1.6.4 Typy obiektowe dynamiczne

Zmienne przechowujące wskaźniki do zmiennych, które podobnie jak obiekty statyczne mogą przechowywać informacje, i wykonywać na tych informacjach operacje. Do tworzenia i usuwania obiektów dynamicznych służą konstruktory i destruktory za pomocą operatorów `new` i `delete`. Do składowych zmiennej obiektowej utworzonej dynamicznie odwołujemy się za pomocą operatora `->`.

```
class Kl {
public:
  int a;
  Kl(void);
  ~Kl(void);
};

Kl *wskOb;
wskOb = new Kl;
wskOb->a = 5; // LUB
(*wskOb).a = 3;

cout << wskOb->a;
delete wskOb;
```

2024-10-22

2 Podstawowe techniki programowania

2.1 Technika TOP-DOWN (technika zstępująca)

Polega na rozkładaniu podstawowego problemu na mniejsze podproblemy z których każdy ponownie rozkłada się na mniejszy podproblemy aż ostatecznie dochodzimy do pojedynczej instrukcji rozwiązującej dany podproblem.

Na przykład program rozwiązujący równanie kwadratowe można rozdzielić na część pobierającą dane, część wyznaczającą wyróżnik równania, część określającą liczbę pierwiastków i część wyznaczającą te pierwiastki i część wydającą wynik.

Jeszcze jedną podstawową jednostką w tej metodzie jest algorytm (istotna jest reguła rozwiązująca dany problem, a nie wykorzystane dane i struktury danych)

Technika TOP-DOWN stosuje następujące elementy:

i) Dekompozycja – rozkładanie problemu na podproblemy z dokładnym ich wyspecyfikowaniem (określeniem, co dokładnie ma robić ta część programu i przy wykorzystaniu jakich danych)
ii) Kodowanie - rozwiązanie danej części zadania programistycznego za pomocą pojedynczej instrukcji albo zestawu instrukcji
iii) Odraczanie – pozostawienie danego problemu do późniejszego rozwiązania. W tym celu można wykorzystać zaślepkę (procedurę/funkcję, która choć nie rozwiązuje danego problemu, to daje jednak pozornie poprawne wyniki, pozwalające przetestować poprawność reszty programu)

2.2 Technika BOTTOM-UP (technika wstępująca)

Budowanie rozwiązania problemu poprzez składanie pojedynczych instrukcji rozwiązujących pewną część zadania programistycznego. Podstawową jednostką w tej technice jest model danych (dziedzina danej, oraz operacje i relacje które można na niej wykonywać). Z tego względu w tej technice często stosuje się obiektowe metody programowania

2.3 Programowanie obiektowe

W programowaniu strukturalnym dane były jedynie przedmiotami na których operował algorytm. W obiektowym metodach programowania dane są podmiotami, mogącymi wykonywać na sobie pewne (odpowiednie dla nich) działania i operacje – obiekty poza polami przechowującymi dane

posiadają także metody, czyli procedury i funkcje wykonujące operacje na obiekcie. Programowanie obiektowe zwiększa poziom abstrakcji dostosowuje rozwiązanie problemu do języka programisty, a nie komputery.

```
class Klasa {
  int pole1;
  bool pole2;
  string pole3;
public:
  Klasa(int a);
  Klasa(bool b);
  ~Klasa();
  void metoda1(int a, bool b);
  bool metoda2(int* c);
};
```

Obiekty posiadają specjalne metody tzw. konstruktory, o nazwie takiej, jak nazwa klasy, a po jego utworzeniu wykonuje instrukcje zawarte w definicji konstruktora; oraz destruktory które najpierw wykonują instrukcje zawarte w definicji destruktora, po czym usuwają obiekt tej klasy. Najważniejszymi elementami programowania zorientowanego obiektowo są

- enkapsulacja
- hermetyzacja
- dziedziczenie
- polimorfizm

1. Enkapsulacja – obiekty łączą w sobie zarówno informację zawarte w polach klasy jak i operacje które możemy na nich wykonywać. Takie połączenie nazywamy enkapsulacją
2. Hermetyzacja – dostęp do składowych obiektów(ich pól i metod może być ograniczany przez programistę. Zapobiega to przypadkowym i niepożądanym zmianom składowych obiektu). W celu zapewnienia hermetyzacji w języku C++ mamy trzy specyfikatory dostępu:
   i) private – dana składowa jest dostępna jedynie dla metod danej klasy. Stanowi najmocniejsze zabezpieczenie przed niepożądanym dostępem. Private jest domyślnym specyfikatorem dostępu
   ii) protected – składowe są dostępne tak jak przy specyfikatorze private, ale dodatkowo dla składowych obiektu mają dostęp także składowe klas potomnych
   iii) public – dostęp do składowych jest nieograniczony
3. Dziedziczenie – umożliwia tworzenie klas potomnych które rozszerzają definicję klasy, z której odbywa się dziedziczenie. Rozszerzenie to może odbywać się poprzez dodanie nowych pól, do-

danie nowych metod lub zmianę działania metod występujących w klasie przodka (polimorfizm). W języku C++ przy dziedziczeniu określa się również specyfikator dostępu do klasy przodka

4. Polimorfizm – pozwala programiście na uzależnienie zachowania się obiektu od faktycznej jego klasy. Mechanizm ten znacząco ułatwia dodawanie nowej funkcjonalności do programu, oraz czyni jego konstrukcję logiczną z punktu widzenia koncepcyjnego. Zamiast tworzyć program w oparciu o liczne instrukcji warunkowe dostosowujące działanie programu do konkretnej sytuacji, sposobem wykonania programu kieruje typ obiektów podlegających polimorfizmowi. Praktyczna realizacja polimorfizmu odbywa się w oparciu o metody wirtualne, zmieniające swoje zachowanie w zależności od klasy wywołującego ją obiektu. Możliwe jest również tworzenie klas abstrakcyjnych (poprzez tworzenie metod czysto wirtualnych), których obiektów nie można tworzyć bezpośrednio, a jedynie poprzez utworzenie obiektów klas potomnych. Taka klasa abstrakcyjna stanowi szablon dla klas potomnych informując, jakie cechy powinny być zaimplementowane w klasach potomnych.

```
class Figure {
  // ...
public:
  Figure();
  virtual ~Figure();
  virtual void draw();
  // ...
};

class Prostokat : public Figure {
  // ...
public:
  Prostokat();
  virtual ~Prostokat();
  void draw();
  // ...
};

class Elipsa : public Figure {
  // ...
public:
  Elipsa();
  virtual ~Elipsa();
  void draw();
  // ...
};

int main() {
```

```
  Figure *fig;
  fig = new Prostokat();
  fig->draw(); // rysuje prostokąt
  delete fig;
  fig = new Elipsa();
  fig->draw(); // rysuje elipsę
  delete fig;
}
```

Dzięki dziedziczeniu do wskaźnika `fig` możemy przypisać obiekt klasy `Figure` jak również dowolny obiekt z klas dziedziczących. Jeśli dana metoda (`draw`) zadeklarowana jest jako wirtualna, wówczas jej zachowanie zależne jest od faktycznej klasy obiektu. Metoda `draw` w przypadku obiektu klasy `Prostokat` rysuje prostokąt, w przypadku obiektu klasy `Elipsa` rysuje elipsę, a w przypadku klasy `Figure` wykonałby instrukcję zawarte w tej metodzie w klasie `Figure`. W ten sposób zamiast korzystać wielokrotnie z instrukcji warunkowych uzależniających zachowanie programu od wybranej figury można ujednolicić kod programu.

W podanym przykładzie dla obiektu klasy `Figure` metoda `draw` chciałaby narysować ogólną metodę. Ponieważ nie wiemy jaką figurę należy wówczas narysować, to chcielibyśmy nie wywoływać tej metody, a jedynie wywoływać ją w klasach dziedziczących po klasie `Figure`. Można wówczas zadeklarować metodę `draw` jako czysto wirtualną

```
class Figure {
  // ...
public:
  virtual void draw() = 0;
  // ...
};
```

Taka deklaracja oznacza, że ta metoda nie będzie definiowana w klasie `Figure`, a jedynie w klasach potomnych. Konsekwencją jest fakt, że klasa `Figure` staje się klasą abstrakcyjną – nie możemy tworzyć obiektów tej klasy(bo nie moglibyśmy wywołać dla tego obiektu metody `draw`). Taka klasa staje się wówczas szablonem dla kolejno definiowanych klas(figur) potomnych, dając nam informację, że każdą figurę będzie można narysować, czyli że konkretne figury będą miały metodę `draw`. Oczywiście jeśli w którejś z klas potomnych metoda `draw` nie będzie przedefiniowana, to ta klasa automatycznie również stanie się klasą abstrakcyjną.

2024-10-29

3 Złożoność obliczeniowa

Konstruując algorytmy chcielibyśmy mieć możliwość ich oceny. Oczywiście podstawowym kryterium jakości algorytmu jest odpowiedź na pytanie «czy dany algorytm poprawnie rozwiązuje zadany mu problem?» (kryterium poprawności)

Jeśli jednak mamy do dyspozycji kilka algorytmów które poprawnie rozwiązują rozważany problem należy zastosować kolejne kryterium oceny algorytmów, jakim jest stopień wykorzystania zasobów.

W informatyce istnieją dwa podstawowe zasoby, których stopień zużycia determinuje jakość algorytmu. Są to **czas** i **pamięć**.

Jako, że pamięć jest zasobem stosunkowo tanim (choć nie nieograniczonym), to jako główny zasób uznaje się czas działania algorytmu. Oczywiście, aby ocenić sam algorytm należy zastosować kryterium niezależne od maszyny na której został on uruchomiony (słaby algorytm na dobrym komputerze może być szybszy, niż dobry algorytm na słabym komputerze)

Z tego względu czas działania algorytmu mierzymy liczbą wykonywanych w nim prostych, dominujących w tym algorytmie operacji. Liczba wykonywanych operacji zależy oczywiście również od rozmiaru problemu, dlatego będziemy przyjmowali, że złożoność czasowa jest funkcją rozmiaru problemu $T(n)$.

3.1 Notacja asymptotyczna

Ponieważ dla problemów małych rozmiarów, czas poświęcony na ich rozwiązanie przez algorytm jest z reguły mały, za najważniejsze w ocenie złożoności czasowej algorytmu uznaje się jego zachowanie asymptotyczne, czyli jak szybko rośnie liczba dominujących operacji w miarę wzrostu rozmiaru danych wejściowych. Do takiej oceny stosuje się notację asymptotyczną. Oceną złożoności czasowej algorytmów można wykonać dla różnych przypadków danych wejściowych:

- Złożoność optymistyczna – złożoność w przypadku danych wejściowych najlepszych z punktu widzenia algorytmu
- Złożoność średnia (oczekiwana) – złożoność w przypadku przeciętnych danych wejściowych
- Złożoność pesymistyczna – złożoność w przypadku danych wejściowych najgorszych z punktu widzenia danego algorytmu

1. Notacja Θ – złożoność algorytmu $T(n)$ jest rzędu $g(n)$

$$T(n) = \Theta(g(n)) = \{f(n) : \bigvee_{c_1,c_2 \in \mathbb{R}_+} \bigvee_{n_0 \in \mathbb{N}} \bigwedge_{n \geq n_0} 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

2. Notacja O – złożoność algorytmu $T(n)$ jest co najwyżej rzędu $g(n)$

$$T(n) = O(g(n)) = \{f(n) : \bigvee_{c \in \mathbb{R}_+} \bigvee_{n_0 \in \mathbb{N}} \bigwedge_{n \geq n_0} 0 \leq f(n) \leq cg(n)\}$$

3. Notacja o – złożoność algorytmu $T(n)$ jest mniejszego rzędu niż $g(n)$

$$T(n) = o(g(n)) = \{f(n) : \bigwedge_{c \in \mathbb{R}_+} \bigvee_{n_0 \in \mathbb{N}} \bigwedge_{n \geq n_0} 0 \leq f(n) < cg(n)\}$$

4. Notacja Ω – złożoność $T(n)$ jest co najwyżej rzędu $g(n)$

$$T(n) = \Omega(g(n)) = \{f(n) : \bigvee_{c \in \mathbb{R}_+} \bigvee_{N_0 \in \mathbb{N}} \bigwedge_{n \geq n_0} 0 \leq cg(n) \leq f(n)\}$$

5. Notacja ω – złożoność $T(n)$ jest wyższego rzędu niż $g(n)$

$$T(n) = \omega(g(n)) = \{f(n) : \bigwedge_{c \in \mathbb{R}_+} \bigvee_{n_0 \in \mathbb{N}} \bigwedge_{n \geq n_0} 0 \leq cg(n) < f(n)\}$$

3.2 Klasy złożoności

1. $\Theta(1)$ – złożoność stała
2. $\Theta(\log n)$ – złożoność logarytmiczna
3. $\Theta(n)$ – złożoność liniowa
4. $\Theta(n \log n)$ – złożoność liniowo-logarytmiczna
5. $\Theta(n^k)$ – złożoność wielomianowa ($\mathcal{P}$)
6. $\Theta(n^{\log n})$ – złożoność podwykładnicza
7. $\Theta(k^n)$, gdzie $k \in \mathbb{N} \setminus \{0, 1\}$ – złożoność wykładnicza
8. $\Theta(n!)$ – złożoność wykładnicza typu silnia

Dla dużych danych wejściowych w praktyce rozwiązywalne są problemy o maksymalnej złożoności wielomianowej. W algorytmice występuje specjalna klasa problemów $\mathcal{NP}$ (non-deterministic polynomial), dla których zostały znalezione algorytmy rozwiązujące je w czasie wykładniczym, a nawet niedeterministycznie wielomianowym, ale nie wiadomo czy istnieją algorytmy rozwiązujące je w czasie wielomianowym na zwykłym komputerze. Wśród problemów $\mathcal{NP}$ znajduję się grupa problemów zwanych $\mathcal{NP}$-zupełnymi które charakteryzują się tym, że jeśli istnieje rozwiązania któregokolwiek z tej grupy w czasie wielomianowym, to ponieważ każdy problem $\mathcal{NP}$ da się sprowadzić do problemu $\mathcal{NP}$-zupełnego, zatem każdy problem $\mathcal{NP}$ miałby rozwiązanie w czasie wielomianowym.

Istnieją również problemy $\mathcal{NP}$-trudne, które mogą być sprowadzone do problemów $\mathcal{NP}$–zupełnych w czasie wielomianowym, a więc bez podwyższania klasy złożoności rozwiązania

---

2024-11-05

4 Rekurencja, rekursja i problemy z nimi związane

4.1 Rekurencja

Rekurencja zależność zdefiniowana poprzez odwołanie się do samej siebie, np.:

$$n! = \begin{cases} 1, & n = 0 \\ n \cdot (n-1)!, & n \in \mathbb{N}_+ \end{cases}$$

Praktyczną realizację rekurencji w programach komputerowych jest rekursja:

```
int Silnia(int n) {
  if (n == 0) {
    return 1;
  }
  return n * Silnia(n - 1);
}
```

Rekursywne wywołanie funkcji jest kosztowne. Wymaga wykorzystanie pamięci (stosu) w celu umieszczenia tam wartości zmiennych, oraz czasu związanego z odkładaniem ich na stosie.

Z punktu widzenia całego algorytmu nie musi to jednak być koszt najistotniejszy i nie musi wiązać się ze zwiększeniem złożoności algorytmu. Z tego względu choć unikanie rekursji jest wskazane ze względu na szybkość działania algorytmu, to szczególnie w sytuacjach w których problem jest typowo rekurencyjny, wykorzystanie rekursji nie jest błędem. Zastosowanie rekursji może wiązać się jednak z pewnymi dodatkowymi problemami.

Przykład 4.1

Zdefiniujemy ciąg, którego kolejny wyraz jest sumą 2 poprzednich, zwany w matematyce ciągiem Fibonacciego

$$F(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F(n-1) + F(n-2), & n \in \mathbb{N} \setminus \{0, 1\} \end{cases}$$

Stosując w rozwiązaniu rekursywnym bezpośrednio podany wzór otrzymamy funkcję postaci

```
int ZleFibo(int n) {
  if (n == 0)
    return 0;
  if (n == 1)
    return 1;
  return ZleFibo(n - 1) + ZleFibo(n - 2);
}
```

Zobaczmy, jak działa ta funkcja:

```
ZleFibo(n) =
ZleFibo(n-1) + ZleFibo(n-2) =
ZleFibo(n-2) + ZleFibo(n-3) + ZleFibo(n-2) =
ZleFibo(n-2) + ZleFibo(n-3) + ZleFibo(n-3) + ZleFibo(n-4) = ...
               ^^^^^^^^^^^^   ^^^^^^^^^^^^
```

Widzimy, że w przypadku tego algorytmu następuje wielokrotne wyznaczanie wartości tych samych funkcji dla tych samy elementów, co sprawia, że algorytm jest bardzo nieefektywny. Aby temu zaradzić musimy w procedurze wyznaczania ciągu Fibonacciego uwzględnić także wyraz poprzedni

```
void FF(int n) {
  n = 3;
}

int main() {
  int a = 4;
  FF(a);
  cout << n; // wydrukuje 4
}


void FF(int* n) {
  *n = 3;
}
```

```
int main() {
  int *a;
  a = new int;
  *a = 4;
  FF(a);
  cout << *a; // wydrukuje 3
}

void FiboPar(int n, int *F0, int *F1) {
  if(n == 0 || n == 1) {
    *F0 = 0;
    *F1 = 1;
  } else {
    FiboPar(n-1, F1, F0); // F0 = F(n-1), F1 = F(n-2)
    *F1 = *F0 + *F1; // F0 = F(n-1), F1 = F(n)
  }
}

int Fibo(int n) {
  int R0, R1;
  FiboPar(n, &R0, &R1);
  if(n == 0)
    return R0;
  return R1;
}
```

4.2 Rekursywne znajdowanie największego wspólnego dzielnika

W celu znalezienia największego wspólnego dzielnika można posłużyć się następującym wzorem rekurencyjnym będącym realizacją tak zwanego zmodyfikowanego algorytmu Euklidesa

$$\gcd(a,b) = \begin{cases} \gcd(|a|,|b|), & a < 0 \vee b < 0 \\ a, & b = 0 \\ \gcd(b, a \bmod b), & \text{w przeciwnym razie} \end{cases}$$

```
int NWD(int a, int b) {
  if(a < 0 || b < 0)
    return NWD(abs(a), abs(b));
  if (b == 0)
```

```
    return a;
  return NWD(b, a % b);
}
```

4.3 Sortowanie przez scalanie

Rekursję można także zastosować w problemie sortowania. Przykładem zastosowania rekursywnego algorytmu sortowania jest algorytm sortowania przez scalanie (merge sort). W algorytmie tym korzystamy z następujących zasad:

1. Tablica 1-elementowa jest posortowana
2. Sortując tablicę dzielimy ją na 2 podtablice równej (lub prawie równej) długości
3. Sortujemy każdą z podtablic osobno
4. Łączymy dwie posortowane podtablice w 1 posortowaną tablicę

2024-11-12

4.3.1 Implementacja `Merge`

Procedura łącząca 2 posortowane podtablice tablicy A, gdzie p jest początkowym indeksem pierwszej podtablicy, q jest końcowym indeksem pierwszej podtablicy, q+1 jest początkowym elementem drugiej podtablicy, a r jest końcowym indeksem drugiej podtablicy (czyli mamy podtablice A[p..q] i A[q+1..r]) ma postać:

```
template<class Typ>
void Merge(Typ* A, int p, int q, int r) {
  int n1, n2, k, l, m;
  Typ* Pom1;
  Typ* Pom2;
  n1 = q - p + 1; // długość pierwszej podtablicy
  n2 = r - q;     // długość drugiej podtablicy
  Pom1 = new Typ[n1]; // pierwsza podtablica
  Pom2 = new Typ[n2]; // druga podtablica
  for(int i = 0; i < n1; i++) {
    Pom1[i] = A[p + i];
  }
  for(int i = 0; i < n2; i++) {
    Pom2[i] = A[q + 1 + i];
```

```
  }
  k = 0; // indeks w pierwszej podtablicy
  l = 0; // indeks w drugiej podtablicy
  m = p; // indeks w tablicy A
  while(n1 > 0 && n2 > 0) {
    if(Pom1[k] < Pom2[l]) {
      A[m] = Pom1[k];
      k++;
      n1--;
    } else {
      A[m] = Pom2[l];
      l++;
      n2--;
    }
    m++;
  }
  if (n1 == 0) {
    for(int i = 1; i <= n2; i++) {
      A[m] = Pom2[l];
      l++;
      m++;
    }
  } else {
    for(int i = 1; i <= n1; i++) {
      A[m] = Pom1[k];
      k++;
      m++;
    }
  }
  // Dbamy o pamięć
  delete[] Pom1;
  delete[] Pom2;
}
```

4.3.2 Ostateczny `MergeSort`

Ostatecznie rekursywna procedura sortowania przez scalanie ma postać:

```
template<class Typ>
void MergeSort(Typ *a, int p, int r) {
  int q;
  if(p < r) {
    q = (p + r) / 2;
```

```
    MergeSort(a, p, q);
    MergeSort(a, q + 1, r);
    Merge(a, p, q, r);
  }
}
```

Wykonując sortowanie przez scalanie na każdym poziome rekursji n wstawień elementów łącząc każdą parę podtablic. Ponieważ każdy podział tablicy na dwie podtablice zmniejsza nam rozmiar problemu problemu dwukrotnie (aż dojdziemy do problemu rozmiaru 1), czyli liczbę kroków `i` po których zakończymy podział jest równe:

$$\frac{n}{2^i} = 1 \qquad n = 2^i \qquad i = \log_2 n$$

Złożoność obliczeniowa algorytmu `MergeSort` zatem rzędu $\Theta(n \log_2 n) = \Theta(n \ln n)$

Algorytm sortowania przez scalanie jest przykładem zastosowania techniki programistycznej «divide and conquer», która polega na zamianie rozwiązanego problemu na prostsze problemy tego samego typu.

4.4 Bisekcja

Innym przykładem zastosowania tej techniki jest bisekcyjne znajdowanie miejsca zerowego funkcji ciągłej z zadaną dokładnością ε. Niech będzie zaimplementowana funkcja `double fun(double x);` zwracająca wartość funkcji, której miejsca zerowego poszukujemy dla zadanego argumentu `x`. W metodzie tej korzysta się z własności funkcji ciągłej polegającej na tym, że jeśli na końcach badanego przedziału wartość funkcji przyjmuje przeciwne znaki to wewnątrz tego przedziału musi istnieć miejsce zerowe. Zdefiniujemy funkcje rekurencyjnie rozwiązującą ten problem:

```
#include <cmath>

double Mz(double xl, double xp,
          double fl, double fp, double eps) {
  double xs, fs;
  if(std::abs(xp - xl) < eps) {
    return (xp + xl) / 2;
  }
  xs = (xp + xl) / 2.0;
  fs = fun(xs);
  if(fl * fs < 0) {
    return Mz(xl, xs, fl, fs, eps);
  }
```

```
  return Mz(xs, xp, fs, fp, eps);
}
```

Zastosowana metoda bisekcji w każdym rekursywny kroku dzieli przeszukiwany przedział na pół i do dalszych obliczeń wybiera tę połowę, w której znajduje się miejsce zerowe.

W programie głównym (zakładając, że a i b są końcami przedziału w którym szukamy miejsca zerowego, a eps zadaną dokładnością) należy umieścić instrukcję:

```
void glowny(double a, double b, double eps) {
  bool znaleziono;
  double MZerowe, fa, fb;
  fa = fun(a);
  fb = fun(b);
  znaleziono = true;
  if(fa * fb > 0) {
    znaleziono = false;
  } else {
    MZerowe = Mz(a, b, fa, fb, eps);
  }
}
```

Chociaż rekursja jest wygodną i do rozwiązania niektórych problemów idealną techniką, to ze względu na dodatkowy koszt czasowy i pamięciowy należy dążyć do unikania jej stosowania w sytuacjach w których badany problem da się rozwiązać w inny sposób.

4.5 Silnia bez rekursji

Jedną z metod usuwania rekursji jest wykorzystanie pojedynczej pętli

```
int SilniaNieRek(int n) {
  int a = 1;
  for(int i = 1; i > 1; i++) {
    a *= i;
  }
  return a;
}
```

4.6 Liczby Fibonacciego bez rekursji

W eliminowaniu rekursji można także wykorzystać metodę tablicową. Wymaga ona utworzenia dodatkowej tablicy w której umieszczamy kolejne wyniki pośrednie pozwalające w kolejnych obrotach

pętli wyznaczyć ostateczne rozwiązanie. Metoda ta jest szczególnie efektywna w przypadkach w których wyznaczenie ostatecznego rozwiązania wymaga wyznaczenia wartości dla większości indeksów pośrednich.

```
int FiboNieRek(int n) {
  int *TabPom;
  int wyn;
  if (n == 0) {
    return 0;
  }
  TabPom = new int[n+1];
  TabPom[0] = 0;
  TabPom[1] = 1;
  for (int i = 2; i <= n; i++) {
    TabPom[i] = TabPom[i - 1] + TabPom[i - 2];
  }
  wyn = TabPom[n]; // Potrzebujemy tego elementu ze względu na to, że jeśli
                   // zrobimy `return TabPom[n]`, to już nie będziemy mogli
                   // usunąć pamięci po wywołaniu return'a
  delete[] TabPom;
  return wyn;
}
```

2024-11-19

5 Struktury danych

Struktura danych stanowi sposób zorganizowania zbioru danych. W zależności od tego jakie są relacje pomiędzy poszczególnymi elementami zbioru mamy do czynienia z różnymi strukturami danych.

Najczęstszymi sposobami reprezentacji struktur danych w pamięci komputera są reprezentacje:

1. Listowa (dowiązaniowa)
2. Tablicowa

Przechowywane elementy tworzące strukturę danych posiadają następujące pola:

1. Klucz – wyróżnione informacje (pola/pola) na podstawie której dany element jest identyfikowany
2. Dane uzupełniające (dodatkowe) – pozostałe pola charakteryzujące przechowywaną informacje

3. Dane wskaźnikowe – stanowią informacje o roli elementu i relacji z innymi elementami w strukturze danych

5.1 Lista dowiązaniowa

Lista dowiązaniowa jest strukturą danych w której elementy są ułożone w liniowym porządku wyznaczonym przez wskaźniki związane z każdym elementem listy.

W zależności od liczby kierunków po których możemy poruszać się po elementach listy wyróżniamy 2 rodzaje list: listy jednokierunkowe i listy dwukierunkowe.

Listy pozwalają na wykonywanie na jej elementach 3 operacji:

1. Dodawanie elementu (insert)
2. Usuwanie elementu (delete)
3. Odświeżanie elementu o podobnym kluczu (search)

Załóżmy, że przechowywane na liście elementy mają następującą deklarację:

```
template<class TypKlucza>
class Element {
private:
  TypKlucza Key; // Klucz
  //...          // Inne pola
  Element* Next; // Wskaźnik do następnego elementu
  Element* Prev; // Wskaźnik do poprzedniego elementu
public:
  Element(); // Konstruktor
  ~Element(); // Destruktor
  bool EqualKey(TypKlucza k);

  Element* GetNext();
  void SetNext(Element* n);

  Element* GetPrev();
  void SetPrev(Element* n);
  // ... // inne metody
};
```

A sama lista jest listą dwukierunkową zadeklarowaną następująco:

```
template<class TypKlucza>
class List2Side {
private:
```

```
  Element<TypKlucza>* head;
  Element<TypKlucza>* tail;
public:
  List2Side();
  ~List2Side();
  void Insert(Element<TypKlucza>* el);
  Element<TypKlucza>* Search(TypKlucza k);
  void Delete(Element<TypKlucza>* el);
};
```

Wówczas wspomniane 3 operacje dla listy dwukierunkowej będą miały następujące definicje:

```
#include <cstddef> // NULL pochodzi z tego nagłówku

template<class TypKlucza>
void List2Side<TypKlucza>::Insert(Element<TypKlucza>* el) {
  el->SetNext(head);
  if(head != NULL) {
    head->SetPrev(el);
  } else {
    tail = el;
  }
  head = el;
  el->SetPrev(NULL);
}

template<class TypKlucza>
Element<TypKlucza>* List2Side<TypKlucza>::Search(TypKlucza k) {
  Element<TypKlucza>* x = head;
  while(x != NULL && !x->EqualKey(k)) {
    x = x->GetNext();
  }
  return x;
}

template<class TypKlucza>
void List2Side<TypKlucza>::Delete(Element<TypKlucza>* el) {
  if(el->GetPrev() != NULL) {
    el->GetPrev()->SetNext(el->GetNext());
  } else {
    head = el->GetNext();
  }

  if(el->GetNext() != NULL) {
```

```
    el->GetNext()->SetPrev(el->GetPrev());
  } else {
    tail = el->GetPrev();
  }

  delete el;
}

template<class TypKlucza>
List2Side<TypKlucza>::~List2Side() {
  while(head != NULL) {
    Delete(head);
  }
}
```

5.2 Problem słownika

Listy pozwalają na efektywną realizację problemu słownika. Załóżmy, że mamy pewien zbiór słów S. Słownikiem nazywamy strukturę danych umożliwiającą wykonywanie następujących operacji:

1. `Construct()` – tworzy nowy zbiór słów $S = \varnothing$
2. `Search(v)` – sprawdza czy słowo v należy do słownika S i jeśli tak, to zwraca do niego referencję (wskaźnik)
3. `Insert(v)` – wstawienie słowa v do słownika S, tzn. $S = S \cup \{v\}$, a jeśli v było w słowniku, to zbiór S nie ulega zmianie
4. `Delete(v)` – usuwanie słowa ze v słownika S, tzn. $S = S \setminus \{v\}$, Jeśli $v \notin S$, to zbiór S nie ulega zmianie

5.3 Listy samoorganizujące się (self-organizing lists)

Ponieważ przy odszukiwaniu elementu listy, lista jest zawsze przeglądana od początku, to zarówno złożoność pesymistyczna, jak i oczekiwana odszukiwania jest $\Theta(n)$ w większości przypadków zdarza się jednak, że niektóre elementy są odszukiwane częściej niż inne. Umieszczanie ich bliżej początku listy zmniejsza zatem złożoność czasową odszukiwania. Realizowane jest to w listach samoorganizujących się zmieniających strukturę listy w zależności od częstości dostępu do jej elementów.

Istnieją zatem różne strategie samoorganizowania się list:

1. Strategia B (Basic) – brak samoorganizacji listy
2. Strategia MF (Move to front) – odszukany element jest przenoszony na początek listy

3. Strategia TL (Transposition) – odszukany element jest przenoszony o jedną pozycje w kierunku początku listy
4. Strategia FC (Frequency counter) – w każdym elemencie listy występuje dodatkowe pole – licznik odwołań do elementu zwiększany o 1 przy każdym odwołaniu się do tego elementu. A sama lista jest sortowana malejąco względem tego pola

---

2024-11-26

5.4 Kolejki

Kolejki są strukturami danych typu FIFO (first in first out) w których pierwszy dodany element jest również przetwarzany jako pierwszy. Struktura danych typu FIFO składa się z następującego zbioru wartości, operacji, relacji.

$$\text{FIFO} = (\{\mathbb{U}, \mathbb{Q}\}, \{\text{newqueue}, \text{front}, \text{attach}, \text{detach}\}, \{\text{emptyqueue}\})$$

Gdzie

- $\mathbb{U}$ – zbiór elementów kolejki (“staczy w kolejce”)
- $\mathbb{Q}$ – kolejka (queue)
- newqueue : $\varnothing \to \mathbb{Q}$ jest to operacje tworzenia nowej kolejki
- front : $\mathbb{Q} \rightharpoonup \mathbb{U}$ – operacja pobrania elementu z początku kolejki [2]
- attach : $\mathbb{U} \times \mathbb{Q} \to \mathbb{Q}$ – operacja dodania nowego elementu na koniec kolejki
- detach : $\mathbb{Q} \rightharpoonup \mathbb{Q}$ – operacja usunięcia elementu z początku kolejki (jak poprzednio funkcja częściowa)
- emptyqueue : $\mathbb{Q} \to$ Bool

5.4.1 Aksjomaty struktury danych typu FIFO

1. $\text{emptyqueue}(\text{newqueue})$
2. $\neg\text{emptyqueue}(\text{attach}(u, q))$
3. $\text{front}(\text{newqueue}) = ?$
4. $\text{front}(\text{attach}(u, \text{newqueue})) = u$
5. $\neg\text{emptyqueue}(q) \Rightarrow \text{front}(\text{attach}(u, q)) = \text{front}(q)$

[2] Tu jest taka strzałka bo to jest funkcja częściowa, pusta kolejka nie ma pierwszego elementu

6. detach(attach(u, newqueue)) = newqueue
7. detach(newqueue) =?
8. ¬emptyqueue(q) ⇒ attach(u, detach(q)) = detach(attach(u, q))

Niech klasa Element będzie zdefiniowana tak jak poprzednio

```
template<class TypKlucza>
class Element {
private:
  TypKlucza Key; // Klucz
  //...          // Inne pol
  Element* Next; // Wskaźnik do następnego elementu
  Element* Prev; // Wskaźnik do poprzedniego elementu
public:
  Element(); // Konstruktor
  ~Element(); // Destruktor
  bool EqualKey(TypKlucza k);

  Element* GetNext();
  void SetNext(Element* n);

  Element* GetPrev();
  void SetPrev(Element* n);
  // ... // inne metody
};
```

Wówczas kolejka Queue może być zdefiniowana następująco:

```
#include <cstddef> // NULL pochodzi z tego nagłówku

template<class TypKlucza>
class Queue {
private:
  Element<TypKlucza> *head;
  Element<TypKlucza> *tail;
public:
  Queue();
  ~Queue();
  Element<TypKlucza> *Front();
  void Attach(Element<TypKlucza>* el);
  void Detach();
  bool Emptyqueue();
};

template<class TypKlucza>
```

```
Queue<TypKlucza>::Queue() {
  head = NULL;
  tail = NULL;
}

template<class TypKlucza>
Queue<TypKlucza>::~Queue() {
  while(!Emptyqueue())
    Detach();
}

template<class TypKlucza>
Element<TypKlucza>* Queue<TypKlucza>::Front() {
  return head;
}

template<class TypKlucza>
void Queue<TypKlucza>::Attach(Element<TypKlucza>* el) {
  if (Emptyqueue()) {
    head = el;
    el->SetNext(NULL);
    el->SetPrev(NULL);
  } else {
    el->SetPrev(tail);
    el->SetNext(NULL);
    tail->SetNext(el);
  }
  tail = el;
}

template<class TypKlucza>
void Queue<TypKlucza>::Detach() {
  Element<TypKlucza>* el = head;
  if(Emptyqueue()) {
    return;
  }

  if(el->GetNext() == NULL) {
    head = NULL;
    tail = NULL;
  } else {
    head = el->GetNext();
    head->SetPrev(NULL);
```

```
  }

  delete el;
}

template<class TypKlucza>
bool Queue<TypKlucza>::Emptyqueue() {
  return head == NULL;
}
```

Kolejkę można także reprezentować przy wykorzystaniu tablicy, jakkolwiek taka reprezentacja narzuca a priori ograniczenia na liczbę elementów w kolejce.

W reprezentacji tablicowej przechowujemy indeks głowy Kolejki (pierwszego elementu do przetworzenia w kolejce) i ogona (pierwszego wolnego miejsca do wstawienia do kolejki następnego elementu). W reprezentacji tablicowej ma następującą definicję:

Achtung! 1

Przez to, że dla przechowywanie elementów kolejki używamy tablicy statycznej wskaźników (jakby bezsensowne to nie było), musimy wiedzieć rozmiar tablicy podczas kompilacji, czyli nie możemy przekazać rozmiar wewnętrznej tablicy podczas uruchamiania programu, więc musimy *statycznie* wskazać rozmiar. Dlatego drugi argument szablonu to właśnie ten rozmiar. Bardziej szczegółowe informacje: https://en.cppreference.com/w/cpp/language/template_parameters#Template_non-type_arguments

```
template<class TypKlucza, int n>
class QueueT {
private:
  Element<TypKlucza>* Tablica[n];
  int head;
  int tail;
public:
  QueueT();
  ~QueueT();
  Element<TypKlucza>* Front();
  void Attach(Element<TypKlucza>* el);
  void Detach();
  bool Emptyqueue();
};

template<class TypKlucza, int n>
QueueT<TypKlucza, n>::QueueT() {
```

```
  tail = 0;
  head = 0;
}

template<class TypKlucza, int n>
QueueT<TypKlucza, n>::~QueueT() {
  while(!Emptyqueue())
    Detach();
}

template<class TypKlucza, int n>
Element<TypKlucza>* QueueT<TypKlucza, n>::Front() {
  if(!Emptyqueue())
      return Tablica[head];
  return NULL;
}

template<class TypKlucza, int n>
void QueueT<TypKlucza, n>::Attach(Element<TypKlucza>* el) {
  if(((tail + 1) % n) == head) {
    /* jakaś funkcja BladDodawania() która nam załatwi raportowanie błędu*/
    return;
  }
  Tablica[tail] = el;
  tail = (tail + 1) % n;
}

template<class TypKlucza, int n>
void QueueT<TypKlucza, n>::Detach() {
  if(Emptyqueue())
    return;
  delete Tablica[head];
  head = (head + 1) % n;
}

template<class TypKlucza, int n>
bool QueueT<TypKlucza, n>::Emptyqueue() {
  return head == tail;
}
```

2024-12-03

5.5 Stosy

Stosy są strukturami danych typu LIFO (last in first out) w których ostatni dodany element jest przetwarzany jako pierwszy, czyli przetwarzanie elementów odbywa się w porządku odwrotnym, niż napływanie elementów do stosu. Struktura typu LIFO składa się z następującego zbioru wartości/operacji/relacji:

$$\text{LIFO} = (\{\mathbb{U}, \mathbb{S}\}, \{\text{newstack}, \text{top}, \text{push}, \text{pop}\}, \{\text{emptystack}\})$$

gdzie

- $\mathbb{U}$ – zbiór elementów odkładanych na stos
- $\mathbb{S}$ – stos (ang. stack)
- newstack $:\to \mathbb{S}$ – operacja tworzenia nowego stosu
- top : $\mathbb{S} \rightharpoonup \mathbb{U}$ – operacja pobrania elementu z wierzchu stosu
- push : $\mathbb{U} \times \mathbb{S} \to \mathbb{S}$ – operacja odłożenia nowego elementu na stos
- pop : $\mathbb{S} \rightharpoonup \mathbb{S}$ – operacja zdjęcia elementu ze stosu
- emptystack : $\mathbb{S} \to \mathbb{B}$ – relacja informująca, czy stos jest pusty

5.5.1 Aksjomaty

Aksjomaty struktury danych typu LIFO:

1. $\text{emptystack}(\text{newstack})$
2. $\neg\text{emptystack}(\text{push}(u, s))$
3. $\text{top}(\text{newstack}) =?$
4. $\text{top}(\text{push}(u, s)) = u$
5. $\text{pop}(\text{newstack}) =?$
6. $\text{pop}(\text{push}(u, s)) = s$

5.5.2 Implementacja wskaźnikowa

Niech klasa `Element` będzie zdefiniowana tak jak poprzednio:

```
template<class TypKlucza>
class Element {
private:
  TypKlucza Key; // Klucz
  //...          // Inne pol
  Element* Next; // Wskaźnik do następnego elementu
```

```
  Element* Prev; // Wskaźnik do poprzedniego elementu
public:
  Element(); // Konstruktor
  ~Element(); // Destruktor
  bool EqualKey(TypKlucza k);

  Element* GetNext();
  void SetNext(Element* n);

  Element* GetPrev();
  void SetPrev(Element* n);
  // ... // inne metody
};
```

Wówczas stos `Stack` może być zdefiniowany następująco:

```
#include <cstddef> // NULL pochodzi z tego nagłówku
template<class TypKlucza>
class Stack {
private:
  Element<TypKlucza>* TTop;
public:
  Stack();
  ~Stack();
  Element<TypKlucza>* Top();
  void Push(Element<TypKlucza>* el);
  void Pop();
  bool EmptyStack();
};

template<class TypKlucza>
Stack<TypKlucza>::Stack() : TTop(NULL) {};

template<class TypKlucza>
Stack<TypKlucza>::~Stack() {
  while(!EmptyStack()) {
    Pop();
  }
};

template<class TypKlucza>
Element<TypKlucza>* Stack<TypKlucza>::Top() {
  return TTop;
}
```

```
template<class TypKlucza>
void Stack<TypKlucza>::Push(Element<TypKlucza>* el) {
  if(!EmptyStack()) {
    TTop->SetPrev(el);
  }
  el->SetNext(TTop);
  el->SetPrev(NULL);
  TTop=el;
}

template<class TypKlucza>
void Stack<TypKlucza>::Pop() {
  Element<TypKlucza>* el = TTop;
  if(EmptyStack())
    return;
  if(TTop->GetNext() == NULL) {
    TTop = NULL;
  } else {
    TTop = el->GetNext();
    TTop->SetPrev(NULL);
  }
  delete el;
}

template<class TypKlucza>
bool Stack<TypKlucza>::EmptyStack() {
  return TTop == NULL;
}
```

5.5.3 Implementacja za pomocą tablicy statycznej

Podobnie jak w przypadku kolejki, stos także można reprezentować przy wykorzystaniu tablicy, jakkolwiek taka reprezentacja narzuca a priori ograniczenie na liczbę elementów n odkładanych na stosie.

W reprezentacji tablicowej przechowujemy indeks wierzchu stosu (elementu ostatnio odłożonego na stos).

W reprezentacji tablicowej stos ma następującą definicję:

```
template<class TypKlucza, int n>
class StackT {
```

```
private:
  // Nie używamy tu Element<TypKlucza>* bo to bez sensu
  TypKlucza* tablica[n];
  size_t TTop;
public:
  StackT();
  ~StackT();
  TypKlucza* Top();
  void Push(TypKlucza* el);
  void Pop();
  bool EmptyStack();
};

template<class TypKlucza, int n>
StackT<TypKlucza, n>::StackT()
  :TTop(-1) {};

template<class TypKlucza, int n>
StackT<TypKlucza, n>::~StackT() {
  while(!EmptyStack())
    Pop();
}

template<class TypKlucza, int n>
TypKlucza* StackT<TypKlucza, n>::Top() {
  if(!EmptyStack()) {
    return tablica[TTop];
  }

  return NULL;
}

template<class TypKlucza, int n>
void StackT<TypKlucza, n>::Push(TypKlucza* el) {
  if(TTop == n-1) {
    /*BladDodawania();*/
    return;
  }
  TTop++;
  tablica(TTop) = el;
}

template<class TypKlucza, int n>
```

```
void StackT<TypKlucza, n>::Pop() {
  if(!EmptyStack()) {
    delete tablica[TTop];
    TTop--;
  }
}

template<class TypKlucza, int n>
bool StackT<TypKlucza, n>::EmptyStack() {
  return n < 0;
}
```

6 Odwrotna notacja polska

Przykładowym zagadnieniem w którym wykorzystywane są kolejki i stosy jest przekształcanie wyrażeń ze zwykłej notacji infiksowej, w której operatory działań znajdują się pomiędzy ich argumentami (operandami), na notacje postfiksową, czyli beznawiasowa, zwaną również «Odwrotną notacją polską», w której operatory działań występuje po operandach.

Sposób konwersji z notacji infiksowej na notację postfiksową:

1. Jeśli E jest zmienną lub stałą, to E jest wyrażeniem w notacji postfiksowej
2. Jeśli mamy wyrażenie infiksowej $E_1 \circ E_2$, gdzie E_1, E_2 są wyrażeniami w notacji infiksowej, a $\circ$ jest operatorem, to wyrażenie ma postać $E_1'\ E_2'\ \circ$, gdzie $E_1' \equiv E_1$, $E_2' \equiv E_2$. Na przykład $a + b \equiv a\ b\ +$(z 1 i 2 reguły)
3. Jeśli (E_1) jest wyrażeniem w notacji infiksowej, to w notacji postfiksowej to wyrażenie będzie miało postać E_1', gdzie $E_1' \equiv E_1$(reguła pozbywania się nawiasów)

Przykład 6.1

Wyrażenie w postaci infiksowej $a \cdot (b + c)$ w notacji postfiksowej ma postać $a\ b\ c\ +\ \cdot$

6.1 Zastosowanie

Odwrotna notacja polska upraszcza obliczenia które dzięki niej da się łatwo zautomatyzować. Dzięki wykorzystaniu kolejek i stosów da się również łatwo zautomatyzować proces konwersji z wyrażenia w postaci infiksowej na wyrażenie w postaci postfiksowej.

W procesie konwersji wyrażenia w postaci infiksowej na wyrażenie w postaci postfiksowej zakładamy, iż na wejściu mamy kolejkę której elementami są składowe wyrażenia w postaci infiksowej czytane od

lewej do prawej, na wyjściu zaś chcemy uzyskać kolejkę, której składowymi są elementy wyrażenia w postaci postfiksowej. W procesie konwersji korzysta się z pomocniczej struktury danych – stosu.

Przykład 6.2

Wyrażenie w postaci infiksowej $a \cdot (b + c)$ w kolejce wejściowej ma postać
`a; *; (; b; +; c; )`

2024-12-10

6.2 Konwersja postfix -> infix

Pierwszym krokiem przy wykonywaniu konwersji jest przypisanie priorytetów operatorom i nawiasom

Priorytet	Operatory i nawiasy
0	`(`
1	`+`, `-`, `)`
2	`*`, `/`, `div`, `mod`, `~`
3	`^`, funkcje 1-argumentowe (`sin`, `cos`, ...)

Gdzie «-» jest zwykłym odejmowaniem, natomiast «~» jest negacją

Reguły konwersji z notacji infiksowej na postfiksową:

Każdy argument (operand) z kolejki wejściowej zostaje przepisany do kolejki wyjściowej, zaś operatory i nawiasy są dopisywane na stos według następujących zasad:

1. Nawias `(` przepisujemy na stos bezwarunkowo
2. Operator możemy zapisać na stos tylko wtedy, gdy jego priorytet jest wyższy niż priorytet elementu na wierzchu stosu, albo gdy stos jest pusty, to przepisujemy ze stosy do kolejki wyjściowej wszystkie elementy, aż zajdzie jeden z warunków umożliwiających odłożenie operatora na stos.
3. Nawias zamykający nie jest dopisywany na stos, a jego pojawienie się w kolejce wejściowej powoduje, że ze stosu do kolejki wejściowej przepisywane są wszystkie operatory, aż do napotkania nawiasu `(`, który jest zdejmowany ze stosy, ale nie jest przepisywany na wyjście.
4. W momencie, gdy kolejka wejściowa stanie się pusta, wówczas należy przepisać ze stosu na wyjście wszystkie operatory aż do opróżnienia stosu

Przykład 6.3

Skonwertować wyrażenie w postaci infiksowej do postaci postfiksowej zamieszczone w następującej kolejce
`(; a; *; (; c; +; d; ); -; b; ^; 7; ); +; 5; /; (; a; -; d; +; b; )`

Krok	Kolejka wejściowa	stos	kolejka wyjściowa
1	`a; *; (; c; +; d; ); -; b; ^; 7; ); +; 5; /; (; a; -; d; +; b; )`	`(`	
2	`*; (; c; +; d; ); -; b; ^; 7; ); +; 5; /; (; a; -; d; +; b; )`	`(`	`a`
3	`(; c; +; d; ); -; b; ^; 7; ); +; 5; /; (; a; -; d; +; b; )`	`(*`	`a`
4	`c; +; d; ); -; b; ^; 7; ); +; 5; /; (; a; -; d; +; b; )`	`(*(`	`a`
5	`+; d; ); -; b; ^; 7; ); +; 5; /; (; a; -; d; +; b; )`	`(*(`	`a c`
6	`d; ); -; b; ^; 7; ); +; 5; /; (; a; -; d; +; b; )`	`(*(+`	`a c`
7	`); -; b; ^; 7; ); +; 5; /; (; a; -; d; +; b; )`	`(*(+`	`a c d`
8	`-; b; ^; 7; ); +; 5; /; (; a; -; d; +; b; )`	`(*`	`a c d +`
9	`b; ^; 7; ); +; 5; /; (; a; -; d; +; b; )`	`(-`	`a c d + *`
10	`^; 7; ); +; 5; /; (; a; -; d; +; b; )`	`(-`	`a c d + * b`
11	`7; ); +; 5; /; (; a; -; d; +; b; )`	`(-^`	`a c d + * b`
12	`); +; 5; /; (; a; -; d; +; b; )`	`(-^`	`a c d + * b 7`

```
13                          +; 5; /; (; a; -; d; +; b; )          a c d + * b 7 ^ -
14                             5; /; (; a; -; d; +; b; )  +       a c d + * b 7 ^ -
15                                /; (; a; -; d; +; b; )  +       a c d + * b 7 ^ - 5
16                                   (; a; -; d; +; b; )  +/      a c d + * b 7 ^ - 5
17                                      a; -; d; +; b; )  +/(     a c d + * b 7 ^ - 5
18                                         -; d; +; b; )  +/(     a c d + * b 7 ^ - 5 a
19                                            d; +; b; )  +/(-    a c d + * b 7 ^ - 5 a
20                                               +; b; )  +/(-    a c d + * b 7 ^ - 5 a d
21                                                  b; )  +/(+    a c d + * b 7 ^ - 5 a d -
22                                                     )  +/(+    a c d + * b 7 ^ - 5 a d -
                                                                  b
23                                                        +/     a c d + * b 7 ^ - 5 a d -
                                                                  b +
24                                                               a c d + * b 7 ^ - 5 a d -
                                                                  b + / +
```

Przy obliczaniu wyrażenia w postaci postfiksowej na wejściu mamy kolejkę stanowiącą wyrażenie w postaci postfiksowej, przy obliczeniu wartości wyrażenia korzystamy z pomocniczej struktury danych – stosu(na który odkładamy wartości liczbowe), zaś na wyjściu otrzymujemy wartości wyrażenia.

Reguły wyznaczania wartości wyrażenia w postaci postfiksowej:

1. Jeśli w kolejce wejściowej pojawia się argument, wówczas przypisujemy na stos wartość tego argumentu
2. Jeśli w kolejce wejściowej pojawia się operator 1-argumentowy, wówczas zdejmujemy ze stosu wartość, wyznaczamy wartość wyrażenia odpowiadającego dla argumentu równemu równego wartości argumentu zdjętego ze stosu, zaś jego wynik odkładamy na stosie
3. Jeśli w kolejce wejściowej pojawia się operator 2-argumentowy, to zdejmujemy ze stosu dwie wartości, wyznaczamy wartość wyrażenia odpowiadającego temu operatorowi dla argumentów zdjętych ze stosu(w odwrotnej kolejności niż kolejność zdjęcia ze stosu), zaś jego wynik odkładamy na stosie
4. Po wywołaniu tych czynności, gdy kolejka zostanie opróżniona, na stosie powinien znajdować się dokładnie jeden argument który jest wartością wyrażenia

Przykład 6.4

Wyznaczyć wartość wyrażenia w postaci postfiksowej
`3 2 4 + * 1 7 ^ - 5 3 4 - 2 + / +`

Krok	Kolejka wejściowa	stos
1	`2 4 + * 1 7 ^ - 5 3 4 - 2 + / +`	`3`
2	`4 + * 1 7 ^ - 5 3 4 - 2 + / +`	`3 2`
3	`+ * 1 7 ^ - 5 3 4 - 2 + / +`	`3 2 4`
4	`* 1 7 ^ - 5 3 4 - 2 + / +`	`3 6`
5	`1 7 ^ - 5 3 4 - 2 + / +`	`18`
6	`7 ^ - 5 3 4 - 2 + / +`	`18 1`
7	`^ - 5 3 4 - 2 + / +`	`18 1 7`
8	`- 5 3 4 - 2 + / +`	`18 1`
9	`5 3 4 - 2 + / +`	`17`
10	`3 4 - 2 + / +`	`17 5`
11	`4 - 2 + / +`	`17 5 3`

```
12                          - 2 + / +  17 5 3 4
13                            2 + / +  17 5 -1
14                              + / +  17 5 -1 2
15                                / +  17 5 1
16                                  +  17 5
17                                     22
```

7 Drzewa

Drzewo Drzewa są strukturami danych, w których każdym z elementów z elementów (zwanych wierzchołkami albo węzłami) poza przechowywania klucza i ewentualnie innymi informacjami posiada $n+1$ wskaźników, z których jeden wskazuje na element wskazuje na element zwany ojcem, zaś pozostałe n wskaźników określa n wierzchołków zwanych synami.

Korzeń Drzewa posiadają jeden wyróżniony wierzchołek, nie posiadający ojca i zwany korzeniem (root).

Liść Wierzchołki, które nie posiadają żadnych synów nazywamy liśćmi

W zależności od tego, ilu synów może maksymalnie mieć dowolnych wierzchołek wyróżniamy różne rodzaje drzew:

Drzewo binarne W szczególności drzewa w których każdy wierzchołek może mieć maksymalnie 2 synów nazywamy(czyli każdy element w drzewie przechowuje 3 wskaźniki – do rodzica i 2 synów), nazywamy drzewami binarnymi.

Mówimy, że 2 węzły w drzewie: u i v są odległe o $k > 0$, gdy u jest przodkiem v albo v jest przodkiem u, zaś k jest najmniejszą długością drogi między u i v

Poziom drzewa Poziomem w drzewie nazywamy zbiór węzłów jednakowo odległych od korzenia. Poziomy numerujemy kolejnymi liczbami całkowitymi od 0, gdzie 0 jest poziomem zawierającym jedynie korzeń, 1 poziomem zawierającym jedynie synów korzenia, itd.

2024-12-17

Wierzchołki (węzły) wierzchołki drzewa binarnego binarnego mogą mieć następującą deklarację:

```
template<class TypKlucza>
class Element {
private:
  TypKlucza Key;
  Element* Parent;
  Element* Left;
  Element* Right;
public:
  Element();
  ~Element();
  Element* GetParent();
  void SetParent(Element* p);
  Element* GetLeft();
  void SetLeft(Element* l);
  Element* GetRight();
  void SetRight(Element* r);
  TypKlucza GetKey();
  void SetKey(TypKlucza k);
  // ... // Inne metody
};
```

W przypadku reprezentacji drzew mających wielu synów, wygodnie jest reprezentować wskaźniki do synów w postaci tablicy wskaźników.

Przedstawione reprezentacje mogą być stosowane w sytuacjach w których maksymalna liczba synów wierzchołka jest stała, albo przynajmniej ograniczona pewną liczbą całkowitą. W przypadku, gdy liczba synów wierzchołka może być dowolna i nieograniczoną żadna wartością całkowitą, taka reprezentacja jest jednak niewystarczająca. Można wówczas stosować reprezentację «Na lewo syn, na prawo brat» polegającą na tym, że wykorzystujemy reprezentację taką jak w przypadku drzew binarnych, jednak jedynie wskaźnik do lewego syna jest wskaźnikiem do faktycznego syna, zaś korzystając z «prawego» wskaźnika uzyskujemy dostęp do listy synów tego samego ojca(braci), po której przechodzimy jak po zwykłej liście, korzystając z «prawego» wskaźnika

7.1 Drzewo BST

Drzewo BST (Binary Search Tree) drzewo binarne w którym między kluczami można wprowadzić relację porządku i wszyscy «lewi» potomkowie dowolnego wierzchołka mają klucze nie późniejsze (nie większe) niż wartość klucza w tym wierzchołku, a wszyscy «prawi» potomkowie dowolnego wierzchołka mają klucze nie wcześniejsze (nie mniejsze) niż wartość klucza w tym wierzchołku

Jeśli y jest dowolnym wierzchołkiem, x_l jego lewym potomkiem, a x_p jego prawym potomkiem, to

zachodzi następująca własność:

$$\text{key}[x_l] \leq \text{key}[y] \leq \text{key}[x_p]$$

Wierzchołki drzewa binarnego możemy przetwarzać w następujących porządkach:

1. In-order – przetwarzamy rekursywnie lewe poddrzewo danego wierzchołka, następnie przetwarzamy sam wierzchołek, a na koniec rekursywnie przetwarzamy prawe poddrzewo wierzchołka.
2. Pre-order – najpierw przetwarzamy sam wierzchołek, następnie rekursywnie przetwarzamy lewe poddrzewo wierzchołka, a na koniec rekursywnie przetwarzamy prawe poddrzewo tego wierzchołku
3. Post-order – najpierw rekursywnie przetwarzamy lewe poddrzewo wierzchołka, następnie rekursywnie przetwarzamy prawo poddrzewo tego wierzchołka, a następnie przetwarzamy sam wierzchołek

Algorytmy przechodzenia drzewa zgodnie z tymi porządkami mają następującą postać:

```
#include <cstddef> // Dla NULL

template<class TypKlucza>
void InorderWalk(Element<TypKlucza>* x,
                 void Przetworz(Element<TypKlucza>*)) {
  if (x != NULL) {
    InorderWalk(x->GetLeft());
    Przetworz(x);
    InorderWalk(x->GetRight());
  }
}

template<class TypKlucza>
void PrerderWalk(Element<TypKlucza>* x,
                 void Przetworz(Element<TypKlucza>*)) {
  if (x != NULL) {
    Przetworz(x);
    PrerderWalk(x->GetLeft());
    PrerderWalk(x->GetRight());
  }
}

template<class TypKlucza>
void PostrderWalk(Element<TypKlucza>* x,
                 void Przetworz(Element<TypKlucza>*)) {
  if (x != NULL) {
```

```
    PostrderWalk(x->GetLeft());
    PostrderWalk(x->GetRight());
    Przetworz(x);
  }
}
```

Na drzewach BST można wykonywać następujące operacje:

1. `Minimum` – znajdowanie wierzchołka przechowującego najmniejsza wartość klucza w drzewie
2. `Maximum` – znajdowanie wierzchołka przechowującego największą wartość klucza w drzewie
3. `Successor` – Znajdowanie wierzchołka przechowującego następną w porządku rosnącym wartość klucza w drzewie.
4. `Predecessor` – znajdowanie wierzchołka przechowującego poprzednią (w porządku rosnącym) wartość klucza w drzewie.
5. `Search` – znajdowanie wierzchołka o podanej wartości klucza w drzewie
6. `Insert` – wstawienie wierzchołka do drzewa
7. `Delete` – usunięcie wierzchołka z drzewa

Niech klasa `Element` będzie zdefiniowana tak jak poprzednio. Wówczas drzewo BST może być zdefiniowana następująco:

```
#include <cstddef>
template <class TypKlucza>
class Element {
 private:
  TypKlucza Key;
  Element* Parent;
  Element* Left;
  Element* Right;

 public:
  Element();
  ~Element();
  Element* GetParent();
  void SetParent(Element* p);
  Element* GetLeft();
  void SetLeft(Element* l);
  Element* GetRight();
  void SetRight(Element* r);
  TypKlucza GetKey();
  void SetKey(TypKlucza k);
  // ... // Inne metody
};
```

```
template <class TypKlucza>
class BST {
 private:
  Element<TypKlucza>* root;

 public:
  BST();
  ~BST();
  Element<TypKlucza>* Minimum();
  Element<TypKlucza>* Maximum();
  Element<TypKlucza>* Successor(Element<TypKlucza>*);
  Element<TypKlucza>* Predecessor(Element<TypKlucza>*);
  Element<TypKlucza>* Search(TypKlucza Key);
  void Insert(Element<TypKlucza>* el);
  void Delete(Element<TypKlucza>* el);
};

template <class TypKlucza>
BST<TypKlucza>::BST() {
  root = NULL;
}

template <class TypKlucza>
BST<TypKlucza>::~BST() {
  while (root != NULL) {
    Delete(root);
  }
}

template <class TypKlucza>
Element<TypKlucza>* BST<TypKlucza>::Minimum() {
  Element<TypKlucza>* el;
  el = root;
  if (el == NULL) return NULL;
  while (el->GetLeft() != NULL) el = el->GetLeft();
  return el;
}

template <class TypKlucza>
Element<TypKlucza>* BST<TypKlucza>::Maximum() {
  Element<TypKlucza>* el;
  el = root;
```

```
  if (el == NULL) return NULL;
  while (el->GetRight() != NULL) el = el->GetRight();
  return el;
}

template <class TypKlucza>
Element<TypKlucza>* BST<TypKlucza>::Search(TypKlucza Key) {
  Element<TypKlucza>* x;
  while ((x != NULL) && (Key != x->GetKey()))
    if (Key < x->GetKey())
      x = x->GetLeft();
    else
      x = x->GetRight();
  return x;
}

template <class TypKlucza>
Element<TypKlucza>* BST<TypKlucza>::Successor(Element<TypKlucza>* x) {
  Element<TypKlucza>* y;
  Element<TypKlucza>* z;

  if (x->GetRight() != NULL) {
    y = x->GetRight();
    while (y->GetLeft() != NULL) y = y->GetLeft();
    return y;
  }

  z = x;
  y = x->GetParent();
  while ((y != NULL) && (z == y->GetRight())) {
    z = y;
    y = y->GetParent();
  }
  return y;
}

template <class TypKlucza>
Element<TypKlucza>* BST<TypKlucza>::Predecessor(Element<TypKlucza>* x) {
  Element<TypKlucza>* y;
  Element<TypKlucza>* z;

  if (x->GetLeft() != NULL) {
    y = x->GetLeft();
```

```
    while (y->GetRight() != NULL) y = y->GetRight();
    return y;
  }

  z = x;
  y = x->GetParent();
  while ((y != NULL) && (z == y->GetLeft())) {
    z = y;
    y = y->GetParent();
  }
  return y;
}

template <class TypKlucza>
void BST<TypKlucza>::Insert(Element<TypKlucza>* el) {
  Element<TypKlucza>* x;
  Element<TypKlucza>* y;
  y = NULL;
  x = root;
  while (x != NULL) {
    y = x;
    if (el->GetKey() < x->GetKey()) {
      x = x->GetLeft();
    } else {
      x = x->GetRight();
    }
  }
  el->SetParent(y);
  el->SetLeft(NULL);
  el->SetRight(NULL);
  if (y == NULL) {
    root = el;
  } else if (el->GetKey() < y->GetKey()) {
    y->SetLeft(el);
  } else {
    y->SetRight(el);
  }
}
```

2025-01-07

Usuwanie węzła z drzewa można podzielić na trzy przypadki w zależności od liczby synów usuwanego wierzchołka:

1. Usuwanie węzła bez synów – musimy wówczas jedynie zaktualizować informacje w węźle rodzica, że jego syn został usunięty
2. Usuwanie węzła z jednym synem – musimy wówczas połączyć ojca usuwanego węzła z jego jedynym synem
3. Usuwanie węzła z dwoma synami – musimy wówczas znaleźć węzeł w drzewie przechowujący kolejną wartość (względnie poprzednią wartość). Węzeł ten na pewno nie ma lewego syna (prawego syna – w przypadku poprzednika). Następnie wymieniamy przechowywaną przez niego wartość z wartością faktycznie usuwanego węzła i usuwamy ten węzeł (posiadający co najwyżej jednego syna)

```
void BST::Delete(Element* el) {
  Element* x;
  Element* y;
  if(el->GetLeft() == NULL || el->GetRight() == NULL) {
    y = el;
  } else {
    y = Successor(el);
  }
  if (y->GetLeft() != NULL) {
    x = y->GetLeft();
  }
  if (x!=NULL) {
    x->SetParent(y->GetParent());
  }
  if(y->GetParent() == NULL) {
    root = x;
  } else if (y == y->GetParent()->GetLeft()) {
    y->GetParent()->SetLeft(x);
  } else {
    y->GetParent()->SetRight(x);
  }
  if(y != el) {
    el->SetKey(y->GetKey());
    // ... kopiowanie innych pól
  }
  delete y;
}
```

7.2 Drewa Splay

Drzewa Splay są w zasadzie drzewa BST których implementacja opiera się o operację `Splay` (pochylenie).

Operacja `Splay(l, S)`, gdzie l jest kluczem, a S – binarnym drzewem poszukiwań, przekształca drzewo S w drzewo S', reprezentujące tę sam zbiór kluczy co S, ale w korzeniu drzewa S' znajduję się klucz l, w przypadku gdy ten klucz był w drzewie S, w przeciwnym razie w korzeniu drzewa S' znajduje się węzeł o kluczu l', który jest najbliższy leksykograficznie kluczowi l.

Zaimplementowanie operacji `Splay` znacząco ułatwia operacji `Search`, `Insert` i `Delete`

1. `Search(l)` – wystarczy w konać operację `Splay(l, S)` i sprawdzić jaki węzeł znajduję się w korzeniu nowo powstałego drzewa S'. Jeśli węzeł w korzeniu ma klucz równy l, to ten klucz znajduje się w drzewie i jest jego korzeniem, w przeciwnym razie takiego klucza nie było w drzewie
2. `Insert(l)` – wykonujemy operację `Splay(l, S)`, w korzeniu S' znajdzie się najbliższy leksykograficznie klucz l'. Pozostaje ustalić czy l' jest predecesorem czy sukcesorem
3. `Delete(l)` – :(

Implementacja operacji `Splay(l, S)` składa się z następujących kroków:

1. Wyznaczenie nowego korzenia w drzewie(węzła l, albo najbliższego mu leksykograficznie)
2. Wykonanie ciągu operacji przekształcających drzewo tak, aby węzeł z punktu pierwszego stał się korzeniem drzewa, przy zachowaniu własności drzewa BST

Przekształcenia te nazywamy **obrotami**, rodzaje obrotów:

1. Pojedynczy obrót w prawo/w lewo
2. Podwójny obrót w prawo
3. Złożenie obrotu w lewo i w prawo

7.3 Drzewa dokładnie wyważone

Drzewo dokładnie wyważone Drzewo nazywamy dokładnie wyważonym jeśli dla każdego węzła w drzewie liczby węzłów w lewym i prawym poddrzewie różnią się co najwyżej o 1.

Złożoność obliczeniowa przy wykonywaniu operacji odszukania elementu w dokładnie wyważonym drzewie BST jest rzędu $\Theta(\log n)$ – w każdym kroku przechodząc do kolejnego syna odrzucamy połowę drzewa (poddrzewo zakorzenione w drugim synu). Taka sytuacja ma jednak miejsce tylko w przypadku, gdy drzewo jest drzewem dokładnie wyważonym (węzły są rozłożone równomiernie po prawych i lewych synach kolejnych węzłów)

W najgorszym przypadku, gdy każdy węzeł w drzewie nie ma prawego (względnie lewego) syna ta złożoność rośnie do $O(n)$. Konstruując drzewa BST należy zatem, aby drzewo miało możliwie małą wysokość.

2025-01-14

7.4 Drzewa AVL

Drzewo wyważone Drzewo nazywamy wyważonym, gdy dla każdego węzła wysokości jego poddrzew różnią się co najwyżej o 1

Drzewa realizujące tę zasadę (zrównoważone binarne drzewa poszukiwań) to drzewa AVL (Adelson-Velsky i Landis). Z każdym wierzchołkiem drzewa AVL związana jest pewna liczba całkowita $bf(x)$ (tzw. balance factor) równa różnicy wysokości jego lewego i prawego poddrzewa. W drzewach AVL wartość $bf(x)$ dla każdego wierzchołka x może być równa jedynie $-1, 0, 1$

$$bf(x) = h_l(x) - h_p(x) \in \{-1, 0, 1\}$$

Wykonując wstawienie nowego wierzchołka do drzewa AVL cofamy się od wstawionego wierzchołka idą w kierunku korzenia (zgodnie ze wskaźnikiem `parent`), ewentualnie wykonując obroty naprawiające balance factor $bf(x)$.

Cofając się w kierunku korzenia dla każdego węzła y wyznaczamy $bf(y)$, aż do napotkania węzła y, który:

1. y jest korzeniem i $bf(y) \in \{-1, 0, 1\}$ lub
2. $bf(y) = 0$ lub
3. $bf(y) \in \{-2, 2\}$ (w tym wypadku dokonujemy obrotu naprawiającego własność drzew AVL)

8 Sortowanie

Problem sortowania jest problemem szczególnie często występującym w praktyce programistycznej. Co więcej algorytmy (lub ich fragmenty) tworzone w celu rozwiązania tego problemu znajdują zastosowania także w wielu innych problemach. Wprowadzane w problemie sortowania struktury danych znajdują zastosowania także w wielu innych problemach i pogłębiają wiedzę programistyczną

Rozwiązanie problemu sortowania pomaga przy rozwiązywaniu wielu innych zagadnień jak na przykład problemy wyszukiwania i odszukiwania[3] kluczy o zadanej wartości.

Zdefiniujemy zatem sam problem sortowania: Niech będzie dany ciąg kluczy: $a_1, a_2, \ldots, a_n$, na których określona relacja porządku $\leq$.

Sortowaniem nazywamy procedurę której uzyskujemy permutację tego ciągu wejściowego $a'_1, a'_2, \ldots, a'_n$ taką, że $a'_1 \leq a'_2, \leq \ldots \leq a'_n$

Do tej pory poznaliśmy trzy metody sortowania:

1. sortowanie bąbelkowe (bubble sort)
2. sortowanie przez wstawianie (insertion sort)
3. sortowanie przez scalanie (merge sort)

Aby przedstawić kolejną metodę sortowania wprowadzimy nową strukturę danych – kopiec binarny

8.1 Kopiec

Kopiec binarny jest drzewem binarnym, w którym każdy węzeł od korzenia aż do przed-przed-ostatniego ostatniego poziomu ma dokładnie dwóch synów. Ostatni poziom w drzewie jest wypełniony węzłami od strony lewej do prawej

Kopce binarne występują w dwóch odmianach: w kopcach typu “max” wartość klucza przechowywanego w każdym węźle je nie mniejsza od wartości kluczy przechowywanych w jego synach, zaś w kopcach typu “min” wartość klucza przechowywanego w każdym węźle jest nie większa od wartości kluczy przechowywanego w jego synach

W przypadku kopców binarnych wygodną reprezentacją jest reprezentacja tablicowa. W tej reprezentacji element o indeksie 0 jest to element który jest przechowywany w korzeniu kopca, kolejnymi elementami w tablicy są elementy przechowywane na kolejnych poziomach kopce w porządku od lewej do prawej. W takiej reprezentacji poza tablicą przechowującą elementy kopca (albo wskaźniki do tych elementów) istotne są również 2 pola

1. length – określające rozmiar tablicy
2. `HeapSize` – określające liczbę elementów znajdujących się w kopcu

   Oczywiście HeapSize $\leq$ length

Kopce binarne typu “max” pozwalają na wykonywanie na ich elementach następujących operacji:

[3] oznaczenia doktora Krajki: Odszukiwanie = szukanie jednego, Wyszukiwanie = szukanie wielu

1. `MaxHeapify` – przywraca własność typu “max”(w każdym węźle ma być przechowywany klucz o wartości nie mniejszej niż kluczy synów tego węzła)
2. `BuildMaxHeap` – tworzy z dowolnej tablicy kopiec typu “max”
3. `HeapSort` – operacja sortowania przez kopcowanie elementów tablicy

Deklaracja klasy reprezentującej kopiec może wyglądać następująco:

```
template <class TypKlucza>
class Element {
 private:
  TypKlucza Key;
  Element* Parent;
  Element* Left;
  Element* Right;

 public:
  Element();
  ~Element();
  Element* GetParent();
  Element* GetLeft();
  Element* GetRight();
  TypKlucza GetKey();
  void SetParent(Element* p);
  void SetLeft(Element* l);
  void SetRight(Element* r);
  void SetKey(TypKlucza k);
  bool Wiekszy(Element*);
  // ... // Inne metody
};

template <class TypKlucza>
class Heap {
 private:
  Element<TypKlucza>** tablica;
  int Length;
  int HeapSize;

 public:
  Heap(int len, int hs, Element<TypKlucza>** tab);
  ~Heap();
  int Parent(int i);
  int Left(int i);
  int Right(int i);
  void MaxHeapify(int i);
```

```
  void BuildMaxHeap();
  void HeapSort();
};
```

Natomiast definicja tej klasy może wyglądać następująco:

```
template <class TypKlucza>
Heap<TypKlucza>::Heap(int len, int hs, Element<TypKlucza>** tab) {
  Length = len;
  HeapSize = hs;
  tablica = new Element<TypKlucza>*[len];
  for (int i = 0; i < Length; i++) {
    tablica[i] = tab[i];
  }
}

template <class TypKlucza>
Heap<TypKlucza>::~Heap() {
  for (int i = 0; i < Length; i++) {
    delete tablica[i];
  }
  delete[] tablica;
}

template <class TypKlucza>
int Heap<TypKlucza>::Parent(int i) {
  return (i + 1) / 2 - 1;
}

template <class TypKlucza>
int Heap<TypKlucza>::Left(int i) {
  return 2 * i + 1;
}

template <class TypKlucza>
int Heap<TypKlucza>::Right(int i) {
  return 2 * i + 2;
}
```

Metoda `MaxHeapify` zakłada, że zarówno lewe jak i prawe poddrzewo węzła o indeksie i są kopcami i jedynie sam węzeł i może naruszać własność kopca. Metoda ta wybiera największą wartość klucza spośród wartości przechowywanych w węźle i oraz w jego synach i w przypadku gdy ta wartość znajduje się w jednym z synów wymienia wartości węzłów i odpowiedniego syna, a następnie rekursywnie wywołuje metodę `MaxHeapify` dla tego syna.

```
template <class TypKlucza>
void Heap<TypKlucza>::MaxHeapify(int i) {
  int l = Left(i);
  int r = Right(i);
  int max;

  Element<TypKlucza>* pom;

  if (l < HeapSize && tablica[l]->Wiekszy(tablica[i]))
    max = l;
  else
    max = i;

  if (r < HeapSize && tablica[r]->Wiekszy(tablica[max])) max = r;

  if (max != i) {
    pom = tablica[i];
    tablica[i] = tablica[max];
    tablica[max] = pom;

    MaxHeapify(max);
  }
}
```

2025-01-21

Achtung! 2: Informacje odnośnie egzaminu

- terminy już ustalone
- jedno zadanie zrobione całościowe wystarczy na zaliczenie
- obecność na wykładach daje od 1 do 3 punktów na egzaminie
- na egzaminie będą dwa zadania
  - jedno z materiału z wykładu, wg doktora łatwiejsze
  - drugie modyfikacja tego co było na wykładzie

Metoda `BuildMaxHeap` buduje kopiec rozpoczynając od liści. Ponieważ pojedynczy element jest kopcem, zatem w tej metodzie wystarczy przechodzić do wcześniejszych elementów tablicy wywołując dla tych kolejnych elementów metodę `MaxHeapify`, jako że jedynie w tym nowo dodawanym do kopca węźle własność kopca może nie być spełniona.

```
template <class TypKlucza>
void Heap<TypKlucza>::BuildMaxHeap() {
  HeapSize = Length;
  for (int i = HeapSize / 2 - 1; i >= 0; i--) {
    MaxHeapify(i);
  }
}
```

Zauważmy, że w korzeniu kopca zawsze znajduje się element, którego klucz jest największy. Można zatem zamienić element w korzeniu, z ostatnim elementem w tablicy reprezentującej kopiec, zmniejszymy rozmiar kopca o 1, a następnie przywrócimy własność kopca wywołując metodę `MaxHeapify` dla elementu który pojawił się w korzeniu. Postępując w ten sposób z kopca będziemy wyłączali kolejne największe klucze, co w konsekwencji doprowadzi do posortowania tablicy reprezentującej kopiec. Przedstawiony algorytm stanowi istotę sortowanie przez kopcowanie (`HeapSort`), którego implementacja wygląda następująco:

```
template <class TypKlucza>
void Heap<TypKlucza>::HeapSort() {
  Element<TypKlucza>* pom;
  BuildMaxHeap();
  for (int i = Length - 1; i > 0; i--) {
    pom = tablica[i];
    tablica[i] = tablica[0];
    tablica[0] = pom;
    HeapSize--;
    MaxHeapify(0);
  }
}
```

Można pokazać, że czas działania algorytmu sortowanie przez kopcowanie ma złożoność $O(n \log n)$. Algorytm ten ma jednak tę przewagę nad algorytmem sortowania przez scalanie, że sortowanie odbywa się "w miejscu", czyli przy algorytmie `HeapSort` nie jest potrzebna dodatkowa pamięć o zmiennej wielkości (taka pamięć była potrzebna w algorytmie `MergeSort` podczas łączenia dwóch posortowanych tablic).

8.2 Kolejki priorytetowe

Kolejka priorytetowa jest strukturą danych, do implementacji której można wykorzystać kopiec binarny. W związku z tym, kolejki priorytetowe również występują w 2 odmianach: typu "min" i typu "max".

Kolejki priorytetowe pozwalają na wykonywanie następujących operacji:

1. `Insert(x)` – wstawia element x do kolejki priorytetowej

2. `Maximum()` – zwraca element kolejki o największym kluczu
3. `ExtractMax()` – zwraca i usuwa z kolejki element o największym kluczu
4. `IncreaseKey(x, k)` – zmienia wartość klucza przechowywanego w elemencie x na nową nie mniejszą od poprzedniej wartość k

Zmodyfikujmy zatem poprzednią definicję kopca dodając do niego następujące metody realizujące te operacje:

```
#include <cstddef> // dla NULL
template <class TypKlucza>
class Heap {
 private:
  // ...
 public:
  // ...
  void Insert(Element<TypKlucza>* x, TypKlucza k);
  Element<TypKlucza>* Maximum();
  Element<TypKlucza>* ExtractMax();
  void IncreaseKey(int i, TypKlucza k);
};

template <class TypKlucza>
Element<TypKlucza>* Heap<TypKlucza>::Maximum() {
  return tablica[0];
}

template <class TypKlucza>
Element<TypKlucza>* Heap<TypKlucza>::ExtractMax() {
  Element<TypKlucza>* m;
  if (HeapSize < 1) {
    return NULL;
  }
  m = tablica[0];
  tablica[0] = tablica[HeapSize - 1];
  HeapSize--;
  MaxHeapify(0);
  return m;
}

template <class TypKlucza>
void Heap<TypKlucza>::IncreaseKey(int i, TypKlucza k) {
  Element<TypKlucza>* pom;
  if (tablica[i]->Wiekszy(k)) {
    return;
```

```
  }
  tablica[i]->SetKey(k);
  while (i > 0 && tablica[i]->Wiekszy(tablica[Parent(i)->GetKey()])) {
    pom = tablica[i];
    tablica[i] = tablica[Parent(i)];
    tablica[Parent(i)] = pom;
    i = Parent(i);
  }
}

template <class TypKlucza>
void Heap<TypKlucza>::Insert(Element<TypKlucza>* x, TypKlucza k) {
  HeapSize++;
  tablica[HeapSize - 1] = x;
  x->SetKey(k);
  IncreaseKey(HeapSize - 1, k);
}
```

Kolejki priorytetowe są wygodną strukturą danych we wszystkich zastosowaniach, w których mamy do czynienia z elementami posiadającymi określone priorytety i chcemy je przetwarzać w kolejności rosnących albo malejących wartości tych priorytetów.

8.3 Sortowanie szybkie (QuickSort)

Algorytm sortowania szybkiego jest algorytmem rekursywnym (podobnie jak MergeSort). Istotą algorytmu jest podział sortowanej tablicy na 2 podtablicy oraz element rozdzielający, z których w pierwszej podtablicy znajdują się elementy o nie większej wartości klucza, niż wartość w elemencie rozdzielającym, a w drugiej podtablicy znajdują wartości o nie mniejszej wartości klucza niż wartość w elemencie rozdzielającym. Następnie obie te podtablice są rekursywnie sortowane.

Rekursywna procedura `QuickSort` korzysta z funkcji `Partition` dzielącej tablicę na dwie podtablice i zwracającej indeks elementu rozdzielającego.

```
template <class TypTablicy>
int Partition(TypTablicy* tab, int p, int k) {
  TypTablicy x, pom;
  int i;
  x = tab[k];
  i = p - 1;
  for (int j = p; j < k; j++) {
    if (tab[j] <= x) {
      i++;
      pom = tab[j];
```

```
      tab[j] = tab[i];
      tab[i] = pom;
    }
  }
  pom = tab[i + 1];
  tab[i + 1] = x;
  x = pom;
  return i + 1;
}

template <class TypTablicy>
void QuickSort(TypTablicy* tab, int p, int k) {
  int q;
  if (p < k) {
    q = Partition(tab, p, k);
    QuickSort(tab, p, q - 1);
    QuickSort(tab, q + 1, k);
  }
}
```