Algorytmy i struktury danych
Wstęp
Listy, kolejki, stosy
IO, WIMiIP

Danuta Szeliga

AGH Kraków

2021/2022

Spis treści I

Pojęcia podstawowe

- Dane w komputerze przechowywane są w postaci binarnej — dla komputera to jednolita masa bitów
- Człowiekowi wygodnie jest używać abstrakcyjnego modelu części świata rzeczywistego
- Potrzebna jest zatem metodologia ustrukturalizowania i precyzyjnego zdefiniowania informacji, które następnie mogą być przechowywane i przetwarzane $\rightarrow$ "nałożenie" abstrakcyjnej struktury

Abstrakcyjne typy danych

- Abstrakcyjne Typy Danych (ATD) to modele matematyczne uogólniajace pewna kategorie obiektów, wykazujacych określone zachowanie i posiadajacych określoną strukturę
- Zbiór operacji podstawowych, które mozna wykonać na określonym ATD musi zawierać co najmniej jeden element
- Argumenty i wyniki operacji na ATD mogą być danymi tego ATD lub innych
- Abstrakcyjne typy danych mogą być wzajemnie w sobie zagnieżdżane
- Implementacja ATD polega na zdefiniowaniu jego odpowiednika w konkretnym jezyku programowania
- Przykłady ATD
  Liczby rzeczywiste: zdefiniowane sa operacje dodawania, odejmowania, mnożenia itd.
  Liczby zespolone: zagnieżdżenie dwóch egzemplarzy danych innego ATD; zdefiniowane są operacje dodawania, odejmowania itd.
  Kolejka: zdefiniowane są operacje: `dodaj`, `pobierz`, `czyPusta`

Typ danych

Typ danej definiuje

- Zbiór możliwych wartości, które może przyjmować obiekt
- Sposób kodowania informacji i przechowywania w pamięci
- Możliwe operacje, które mogą być na obiekcie wykonywane

Typ danej/obiektu opisuje pewną podklasę informacji, które mogą być wyrażane i przechowywane za pomocą tej danej

- Typ możemy traktować abstrakcyjnie i wtedy z reguły jest określany przez operacje działające na obiekcie danego typu
- W językach programowania możemy używać ściśle określonych typów
  - Typów, które dostarcza nam kompilator → typy wbudowane (podstawowe i złożone)
  - Typów, które możemy sami konstruować, używając tych, które są już zdefiniowane → strukturalizacja informacji i tworzenie hierarchii typów

Pojęcia podstawowe

Założenie

Każdy obiekt (stała lub zmienna), wyrażenie i funkcja jest pewnego typu

- Stała to obiekt który nie zmienia swojej wartości
- Zmienna to obiekt, który może zmieniać swoją wartość
- Struktura danych to szczegółowe rozwiązanie implementacyjne sposobu przechowywania danych pewnego typu — zbiór obiektów określonych typów, posiadający swoistą organizację i związany z nią sposób wykorzystania

Pojęcia podstawowe

Struktura danych jest spójna

jeżeli dla każdych dwóch różnych jej obiektów A i B istnieje ciąg obiektów rozpoczynający się w A i kończących w B, a dla każdych dwóch kolejnych obiektów w ciągu pierwszy z nich jest następnikiem drugiego lub drugi jest następnikiem pierwszego

Struktura danych jest liniowa

gdy ma jedną funkcję określającą następnika tak, że w strukturze występuje dokładnie jeden obiekt początkowy i dokładnie jeden końcowy (beznastępnikowy), bądź też wszystkie obiekty są początkowe

Struktura danych jest drzewiasta

gdy posiada dokładnie jeden obiekt początkowy, a dla każdego obiektu poza początkowym istnieje w strukturze dokładnie jeden poprzednik

Grafową strukturą danych

jest dowolna struktura danych

Pojęcia podstawowe

Statyczna struktura danych

nie zmienia swojego rozmiaru ani struktury w trakcie działania algorytmu

- Większość języków programowania ma wbudowane mechanizmy wspierające tworzenie statycznych typów danych
- Najczęściej są to
  - tablice
  - rekordy (struktury)
  - pliki (ciągi)

Dynamiczna struktura danych

może zmieniać swój rozmiar i strukturę w trakcie działania algorytmu

- Dynamiczne struktury udostępniane są najczęściej w bibliotekach lub wymagają implementacji
- Najczęściej są to
  - listy
  - drzewa
  - grafy

Przykłady struktur danych

- Liniowe struktury danych
  - lista/wektor
    - tablica (statyczna, dynamiczna, rzadka, macierz)
    - lista z dowiązaniami (jedno- i dwukierunkowa)
    - stos
    - kolejka (jedno- i dwukierunkowa, priorytetowa)
  - tablica asocjacyjna/słownik/mapa
- Nieliniowe struktury danych
  - grafowe struktury danych
    - macierz sąsiedztwa
    - listy sąsiedztwa
    - stos o strukturze grafowej
    - baza danych
  - drzewiaste struktury danych
    - B-drzewa
    - drzewa binarne (BST, AVL, Red-black)
    - kopce

Tablica

Tablica

to struktura danych

- jednorodna, składa się z obiektów tego samego typu
- o dostępie swobodnym, wszystkie składowe mogą być wybrane w dowolnej kolejności i są jednakowo dostępne
- składowe są dostępne przez indeksowanie

W większości języków programowania tablica zajmuje ciągły obszar pamięci, a mechanizm obsługi tablic jest wbudowany w język

C/C++

```
float x[10];
float y[5][5];
z = x[2]+y[2][3];
```

C#

```
float[] x=new float[10];
float[,] y=new float[5,5];
z = x[2]+y[4,3];
```

Java

```
float[] x=new float[10];
float[][] y=new float[5][5];
z = x[2]+y[2][3];
```

Rekord/struktura

Rekord

to struktura danych

- niejednorodna, grupuje kilka powiązanych logicznie ze sobą danych, które mogą być różnych typów
- o dostępie swobodnym
- dane stanowią pola rekordu

W większości języków programowania rekord (jako całość) zajmuje ciągły obszar pamięci, choć ze względu na różne rozmiary danych składowych, czasami poszczególne składowe rekordu nie są składowane w sposób ciągły (*wyrównywanie do granicy słowa*)

C/C++

```
struct Person{
string name;
short age; } Mike;
Mike.age = 10;
```

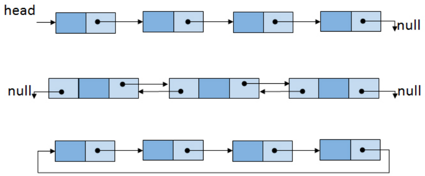
C#

```
struct Person{
public string name;
public short age; } Mike;
Mike.age = 10;
```

Lista z dowiązaniami (Linked list)

Lista

to liniowa struktura danych, zbudowana z sekwencji węzłów (*nodes*), zawierających dane oraz co najmniej jeden odnośnik (link, referencję) do kolejnego węzła ($\rightarrow$ lista jednokierunkowa,singly-linked list); węzeł może zawierać również odnośnik do węzła poprzedniego ($\rightarrow$ lista dwukierunkowa)



- W porównaniu do tablicy, logiczna kolejność elementów listy może być inna od kolejności fizycznej (w pamięci)
- Lista nie zapewnia swobodnego dostępu do jej elementów (z wyjątkiem pierwszego (`head`)) lecz dostęp sekwencyjny
- Implementacja: tablicowa lub wskaźnikowa

Lista

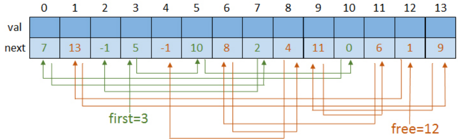
Podstawowe operacje

- Sprawdzenie, czy lista jest pusta
- Wstawienie elementu na początek listy
- Wstawianie elementu wewnątrz listy
- Usuwanie elementu z listy
- Przeglądanie listy
- Wyszukiwanie elementu w liście

Lista

Implementacja tablicowa listy jednokierunkowej

```
const int size = 100;
    struct NODE {
    T val;                // warto Ż
    int next;    // indeks nast Źpnego elementu listy
} list[size]; // lista o max. rozmiarze = size
```



- Lista zaimplementowana w ten sposób opiera się na tablicy obiektów (lub rekordów) danego typu
- Można również zaimplementować taką listę na dwóch tablicach (jedna dla wartości, druga dla wskaźników)
- Niewykorzystane pola tablicy łączone są w postaci osobnej listy dla łatwiejszego ich wykorzystania

\+ Jedyny sposób zaimplementowania listy w językach, które nie posiadają wskaźników

– Ograniczona elastyczność (stały rozmiar tablicy)

Lista (implementacja tablicowa)

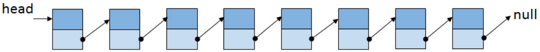
Wstawianie na początek listy

```
bool insertHead(NODE lista[], int& first, int& free, T x){
    if(first == -1) { // je eli lista jest pusta
    lista[first = free].val = x;
    free = lista[free].next;
    list[first].next = -1;
    return true;
    }
    if(free == -1) return false; // brak miejsca
    tmp = lista[free].next;
    lista[free].val = x;
    lista[free].next = first;
    first = free;
    free = tmp;
    return true; // wstawiono
}
```

Lista

Implementacja wskaźnikowa listy jednokierunkowej

```
struct NODE{
    T val; // warto Ż
    NODE* next; // wska nik do nast Źpnego elementu listy
} *head = null; // pocz tkowo lista jest pusta
```



- Lista zaimplementowana w ten sposób opiera się na kolekcji powiązanych ze sobą obiektów utworzonych dynamicznie

+ Duża elastyczność (ograniczona jedynie ilością dostępnej pamięci)

– Kolejne elementy listy nie muszą znajdować się w kolejnych komórkach pamięci

Lista (implementacja wskaźnikowa)

Wstawianie na początek listy

```
insertHead(NODE* & head, T x){
    NODE* tmp = new NODE;
    tmp->val = x;
    tmp->next = head;
    head = tmp;
}
```

Uwagi:

- Wskaźnik do `head` będzie modyfikowany $\rightarrow$ przesyłany do funkcji przez referencję

Lista (implementacja wskaźnikowa)

Wstawianie elementu do listy

- Wstawianie PO elemencie

```
insertAfter(NODE* p, T x){
    NODE* tmp = new NODE;
    tmp->val = x;
    tmp->next =p->next;
    p->next = tmp;
}
```

- Wstawianie PRZED elementem
  Ponieważ nie mamy wskaźnika na poprzedni element, nowy element wstawiamy PO `p`, a następnie podmieniamy wartości

```
insertBefore(NODE* p, T x){
    NODE* tmp = new NODE;
    tmp->val = p->val;
    tmp->next = p->next;
    p->val = x;
    p->next = tmp;
}
```

Lista (implementacja wskaźnikowa)

Usuwanie elementu z listy

- Usuwanie następnika p - nie dla ostatniego elementu:

```
bool deleteAfter(NODE* p){
    NODE* tmp = p->next;
    if(tmp == null) return false;
    p->next = tmp->next;
    delete tmp;
    return true;
}
```

Lista (implementacja wskaźnikowa)

Usuwanie elementu z listy cd.

- Usuwanie p
  Ponieważ nie mamy wskaźnika na poprzedni element, możemy to zrobić, ale nie dla ostatniego elementu listy.

```
bool deleteThis(NODE* p){
    NODE* tmp = p->next;
    if(tmp != null) { // czy nie ostatni element?
        // kopiowanie wartoŻci nastŹpnika
        p->val = tmp->val;
        // kopiowanie wska nika nastŹpnika
        p->next = tmp->next;
        delete tmp;
        return true; // sukces
    }
    return false; // pora ka
}
```

Lista

Przeglądanie/operacja wykonywana na wszystkich elementach listy

- Implementacja tablicowa

```
Visit(NODE list[], int first){
    while(first > -1) {
        do something with list[first].val;
        first = list[first].next;
    }
}
```

- Implementacja wskaźnikowa

```
Visit(NODE* head) {
    NODE* tmp = head;
    while(tmp != null) {
        do something with tmp->val;
        tmp = tmp->next;
    }
}
```

Lista

Wyszukiwanie elementu o wartości x

- Implementacja tablicowa

```
int find(NODE list[], int first, T x){
    while(first > -1) {
        if(list[first].val == x)
            return first;
        else
            first = list[first].next;
    }
    return -1; // nie znaleziono
}
```

- Implementacja wskaźnikowa

```
NODE* find(NODE* head, T x){
    NODE* tmp = head;
    while(tmp != null) {
        if(tmp->val == x)
            return tmp;
        else
            tmp = tmp->next;
    }
    return null; // nie znaleziono
}
```

Lista

Warianty

- Lista dwukierunkowa (*doubly-linked list*)
  - każdy element musi posiadać dodatkowy wskaźnik `prev`
  - dodatkowy wskaźnik pokazujący na ostatni element listy `tail`
  - łatwiejsze operacje wstawiania i usuwania elementów, większe zapotrzebowanie na pamięć
- Lista cykliczna (*circularly-linked list*)
  - pierwszy i ostatni węzeł listy są połączone (są sąsiadami)
  - może być jedno- lub dwukierunkowa
- *Unrolled linked list* — przechowuje wiele wartości w jednym węźle → zwiększenie lokalności danych
- Wartownik (*sentinel*) to sztuczny węzeł, który powala uprościć warunki brzegowe dotyczące ogona i głowy listy
  - Z reguły `wartownik->next == wartownik`
  - Użycie wartowników nie prowadzi zwykle do poprawy asymptotycznej złożoności operacji wykonywanych na liście lecz często prowadzi do zmniejszenia stałych współczynników

Lista

Przyspieszanie wyszukiwania w liście

Poszukiwanie elementu w liście jest mało efektywne, ponieważ

- przeszukiwanie listy może być prowadzone tylko sekwencyjnie
- trzeba odwiedzić wszystkie elementy listy
- bezpośrednie zastosowanie szybszych metod lokalizowania elementów (np. przeszukiwanie binarne) przydatnych w strukturach danych o dostępie dowolnym są nieefektywne dla list

Podstawową metodą przyspieszania wyszukiwania elementów na liści jest wykorzystanie uporządkowania listy względem wybranego klucza

- Listy uporządkowane
- Listy uporządkowane z przeskokami

Lista uporządkowana

- Mniej operacji: zamiast `insertAfter` i `insertBefore` mamy tylko operację `insert`
- Najdogodniejszym sposobem utworzenia listy posortowanej jest zagwarantowanie, że po każdej operacji `insert` lista pozostaje uporządkowana
- + Wyszukiwanie elementu: lista jest przeglądana tylko tak daleko, jak długo elementy mają klucz mniejszy od poszukiwanego. W pesymistycznym przypadku trzeba odwiedzić wszystkie elementy listy.

Lista uporządkowana

Dodawanie elementu

Dane wejściowe: lista posortowana względem danego klucza
Uwagi: lista pusta lub jednoelementowa jest posortowana
Dane wejściowe: lista posortowana względem danego klucza

Wariant 1 Wstawienie na właściwą pozycję

1. Znajdź pozycję, na której powinien pojawić się nowy element by lista pozostała posortowana
2. Wstaw element na tę pozycję

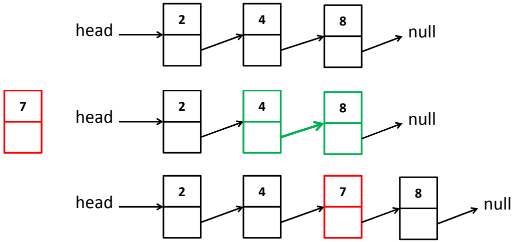
Wariant 2 Przywrócenie uporządkowania

1. Wstaw nowy element na początek listy
2. Przesuwaj ten element dalej tak długo jak następnik istnieje i jego klucz jest mniejszy od klucza elementu dodawanego

Lista uporządkowana

Dodawanie elementu – cd.

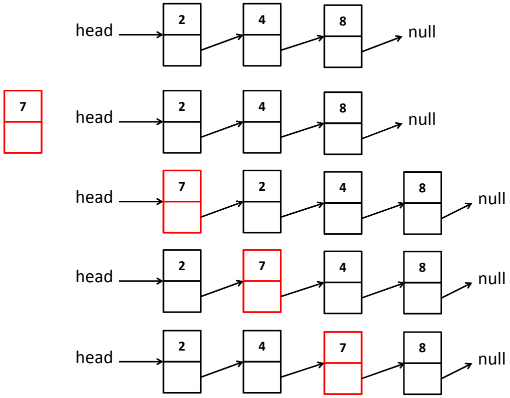
Wariant 1: Przykład



Lista uporządkowana

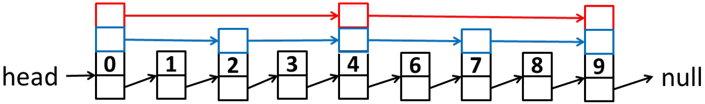
Dodawanie elementu – cd.

Wariant 2: Przykład



Listy uporządkowane z przeskokami

- Wprowadza się dodatkowe poziomy odnośników, pozwalających na przemieszczanie się po liście o więcej niż jeden element.
- Zaleta: przeszukiwanie listy nie musi być sekwencyjne



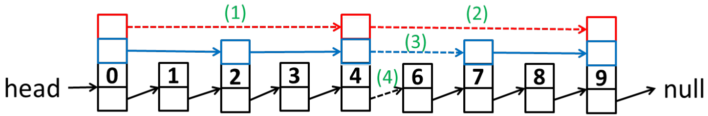
Listy uporządkowane z przeskokami

Poszukiwanie elementu

Poszukiwanie elementu

- W pierwszej kolejności przeszukiwanie prowadzone jest z wykorzystaniem najwyższego poziomu odnośników
- Jeśli osiągnięto koniec listy lub napotkano element z kluczem większym niż poszukiwany, wówczas poszukiwanie ponawiane jest od węzła poprzedzającego, ale z wykorzystaniem wskaźników poziomu o jeden niższego.
- Szukanie trwa aż do znalezienia elementu lub wykorzystania wszystkich poziomów poszukiwań

Przykład: wyszukiwanie elementu z kluczem 5:



Lista

Pozostałe uwagi

Usprawnienia w implementacji listy

- Przydatna są operacje `isEmpty` oraz `size`. Operacja `size` może bazować na liczniku wstawień i usunięć
- Wprowadzenie dodatkowego wskaźnika `tail` przyspiesza dodawanie elementów na koniec listy
- Do usprawnienia przeglądania listy wygodnie jest wprowadzić sztuczny element znajdujący się za ostatnim elementem listy (porównaj rozwiązanie z wartownikiem). Wówczas dobrze jest wprowadzić dwie operacje na liście: `last` oraz `end`

Porównanie listy i tablicy

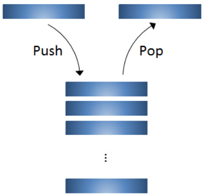
Operacja	Tablica	Lista	
		jednk.	dwuk.
Rozmiar	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Dostęp do elem. brzegowego	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Dostęp do elem. wewnętrznego	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Dostęp do elem. następnego	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Dostęp do elem. poprzedniego	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Wstawianie/usuwanie na początku	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Wstawianie/usuwanie na końcu	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Wstawianie/usuwanie wewnątrz	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Lokalność danych	b. duża	mała	mała

- Dostęp sekwencyjny jest dużo szybszy w przypadku tablic (lokalność danych)
- Lista potrzebuje więcej pamięci (wskaźniki) do przechowywania tej samej ilości danych
- Rozmiar listy może się zmieniać dynamicznie

Stos

Stos (stack)

Stos - Last In First Out (LIFO)
Def. to liniowa struktura danych, w której dane dokładane są na wierzchołek stosu (operacja push) oraz są pobierane (operacja pop) również z wierzchołka stosu



- Aby ściągnąć element ze stosu, należy najpierw po kolei ściągnąć wszystkie elementy znajdujące się nad nim
- Zastosowania:
  - Obliczenia — odwrotna notacja polska (RPN)
  - Pamięć programu (zmienne automatyczne, wywołania funkcji)
  - Algorytmy parsingu, grafowe, ...

Stos

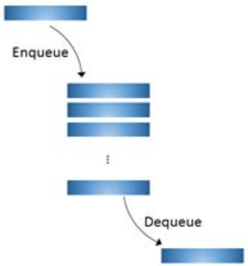
Operacje

- Implementacja tablicowa
  - pierwszy element tablicy = dno stosu
  - ostatni aktywny element tablicy = wierzchołek stosu
  - potrzebna dodatkowa zmienna przechowująca indeks wierzchołka
- Implementacja listowa
  - głowa listy = wierzchołek stosu
  - ogon listy = dno stosu
- Podstawowe operacje
  - `isEmpty()` — $\mathcal{O}(1)$
  - `T pop()` — $\mathcal{O}(1)$
  - `push(T)` — $\mathcal{O}(1)$

Kolejka (*queue*)

Kolejka = First In First Out

to liniowa struktura danych, w której dane dodawane są na końcu (*tail*) kolejki (operacja *enqueue*), a są usuwane (operacja *dequeue*) z początku (*head*) kolejki



- Zastosowania
  - Obsługa zdarzeń
  - Procesy kolejkowe
  - Algorytmy grafowe, ...
- Wariant: kolejka dwustronna (*dequeue*), kolejka priorytetowa (*priority queue*)

Kolejka (*queue*)

- Implementacja tablicowa
  - zorganizowana jako bufor cykliczny
  - potrzebne dwie dodatkowe zmienne do przechowywania indeksów początku i końca kolejki
- Implementacja listowa
  - głowa listy = początek kolejki
  - ogon listy = koniec kolejki
- Podstawowe operacje
  - `isEmpty()` — $\mathcal{O}(1)$
  - `T dequeue()` — $\mathcal{O}(1)$
  - `enqueue(T)` — $\mathcal{O}(1)$
- Kolejka priorytetowa
  - służy do przechowywania elementów zbioru, na którym określono relację porządku
  - najczęściej implementowano jako kopiec lub tablica asocjacyjna