PROJEKTOWANIE I ANALIZA ALGORYTMÓW

TABLICE, LISTY, STOS

Wykład 2

dr inż. Łukasz Jeleń

Na podstawie wykładów dr. T. Fevensa

SPRAWY ORGANIZACYJNE

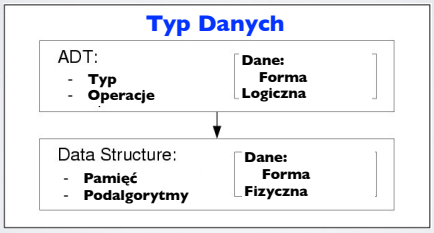
- Egzamin: 28.06.2023, godz. 9:00, sala: TBA
- Konsultacje:
  - czwartki godz. 11:00 - 13:00, s. 230, c-3

POPRZEDNIO

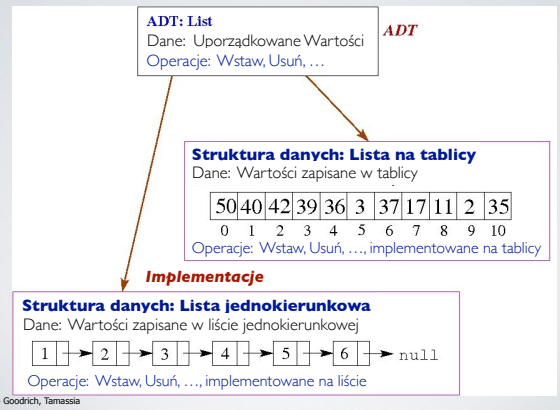
- Definicja Problemu
  - Zbiór danych wraz z poleceniem wykonania
- Definicja Algorytmu
  - Jest zbiorem dobrze zdefiniowanych zasad niezbędnych do rozwiązania problemu
- Struktury Danych
  - Typ danych wraz z polami, zależnościami między nimi oraz operacji na strukturze danych, które pozwalają na manipulację tych pól
  - ADT
    - Definicja typu danych tylko za pomocą wartości i operacji na tym typie danych

STRUKTURA DANYCH VS. ADT

- Struktura danych jest konkretną implementacją ADT



PRZYKŁAD: SD VS. ADT

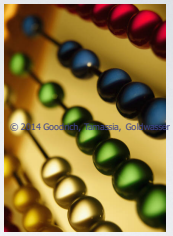


WYBÓR IMPLEMENTACJI ADT

- Tylko po dokładnej analizie charakterystyki problemu możemy wybrać najlepszą strukturę danych dla danego zadania
- Przykład z banku:
  - Otwarcie konta: kilka minut
  - Transakcje: kilka sekund
  - Zamknięcie konta: doba

TABLICE

arrays



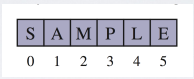
DEFINICJA

- Tablica jest sekwencyjnym zbiorem(kolekcją) zmiennych tego samego typu.
- Każda tablica posiada:
  - komórkę/pole
  - indeks - unikalna referencja do wartości zapisanej na polu/w komórce.
    - 0, 1, 2, …, n
  - każda wartość zapisana w tablicy nazywana jest elementem

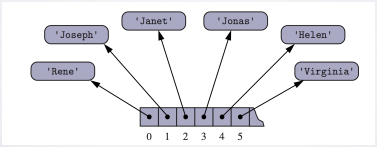
- Deklaracja tablicy:
  - $elementType[]\ arrayName = \{initialValue_0, initialValue_1, \ldots, initialValue_{N-1}\};$
  - operator **new**:
    - new elementType[rozmiar]
      - zwraca referencje do nowej tablicy, która jest zazwyczaj zapisywana w zmiennej tablicowej

TABLICA ZNAKÓW VS. REFERENCJE DO OBIEKTÓW

- Tablica może przechowywać prymitywne elementy - znaki:

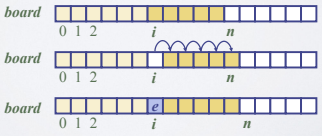


- może też przechowywać referencje do obiektów



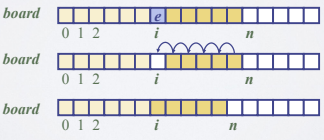
PRZYKŁAD

- Tablica przechowująca nazwiska graczy wraz z ich najlepszymi punktami w grze
- problem - dodawanie do tablicy
  - w celu dodania wpisu „e" do tablicy „board" na indeksie „i", musimy zrobić dla niego miejsce poprzez przesunięcie **n-1** wpisów w tablicy



PRZYKŁAD

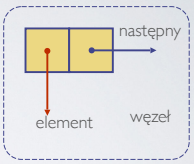
- problem - usunięcie danych z tablicy
  - w celu usunięcia wpisu „e" z tablicy „board" na indeksie „i", musimy przesunąć **n-1** wpisów w tablicy w celu załatania dziury…

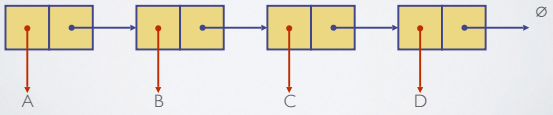


LISTY

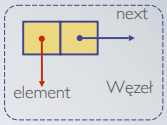
LISTY JEDNOKIERUNKOWE

- Lista jednokierunkowa jest konkretną strukturą danych składającą się z sekwencji węzłów
- Każdy węzeł przechowuje:
  - element
  - link do następnego węzła





DEFINICJA WĘZŁA DLA ELEMENTÓW LISTY




```
template <typename E>
class SNode{                          //węzeł listy jednokier.
private:
  E elem;                             //wartość elementu listy
  SNode<E>* next                      //następny elem. listy
  friend class SLinkedList<E>         //dostęp dla listy
};
```

```
//metody dostępu do danych:

public E getElement()
    {return elem;}

public SNode getNext()
    {return next;}
```

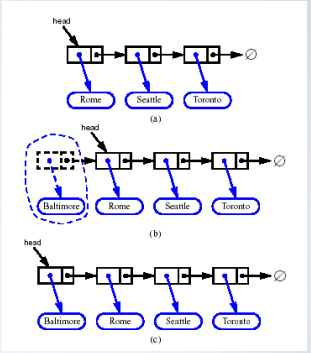
```
//metody modyfikujące:

public void setElement(E newE)
    {elem = newE;}

public void setNext(Node<E> newN)
    {next = newN;}
```


INTERFEJS LISTY JEDNOKIERUNKOWEJ W C++

```
template <typename E>
class SLinkedList{                    //lista jednokierunkowa
public:
  SLinkedList();                      //konstruktor
  ~SLinkedList();                     //destruktor
  bool empty() const;                 //sprawdzanie czy lista jest pusta
  const E& front() const;             //zwraca pierwszy element
  void addFront(const E& e)           //dodawanie na początek listy
  void removeFront()                  //usuń pierwszy element listy
private:
  SNode<E>* head;                     //początek listy
};
```


DODAWANIE NA POCZĄTKU

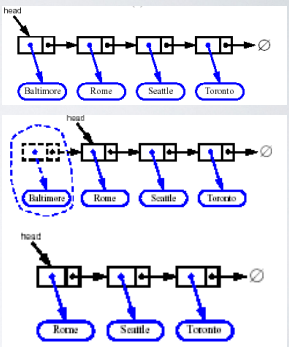
1. Alokacja nowego węzła
2. Dodanie nowego elementu
3. Dodać powiązanie tak, aby nowy węzeł wskazywał na początek listy (stary *head*)
4. Uaktualnić *head* tak, aby wskazywało na nowy węzeł



Przykład realizacji w materiałach na stronie

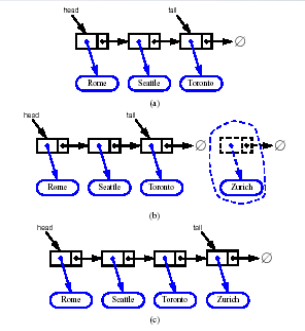
USUWANIE Z PRZODU LISTY

1. Uaktualnić *head* tak, aby wskazywało na kolejny element w liście
2. Usunąć pierwszy węzeł

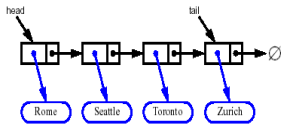


DODAWANIE Z TYŁU LISTY

1. Alokacja nowego węzła
2. Dodanie nowego elementu
3. Dodać powiązanie tak, aby nowy węzeł wskazywał na ***null***
4. Dodać powiązanie tak, aby ostatni element wskazywał na nowy węzeł
5. Uaktualnić *tail* tak, aby wskazywało na nowy węzeł



USUWANIE Z TYŁY LISTY



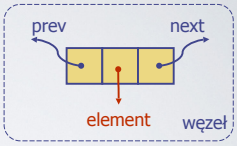
Usuwanie elementów z tyłu listy jednokierunkowej nie jest wydajne!

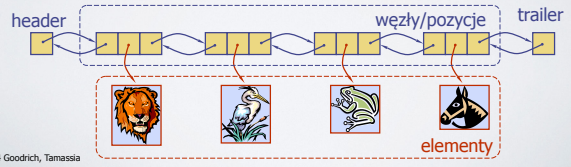
Nie da się przeprowadzić uaktualnienia *tail* tak, aby wskazywał na węzeł poprzedzający w **stałym** czasie

LISTY DWUKIERUNKOWE

LISTA DWUKIERUNKOWA

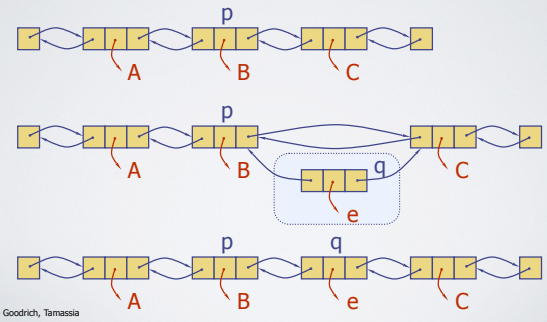
- Lista dwukierunkowa składa się z połączonych węzłów zawierających:
  - element
  - link do poprzedniego węzła
  - link do następnego węzła
- Zawiera dwa dodatkowe węzły
  - header
  - trailer





WSTAWIANIE

Wizualizacja metody addAfter(p, e) zwracająca pozycję q



WSTAWIANIE - ALGORYTM

Algorytm addAfter(p,e):

Stwórz nowy węzeł v

v.setElement(e)

v.setPrev(p) {link v do poprzednika}

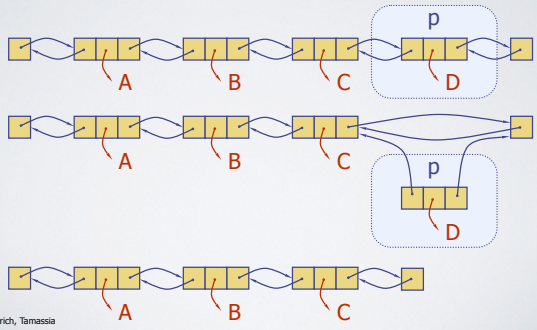
v.setNext(p.getNext()) {link v do następcy}

(p.getNext()).setPrev(v) {link następcy do v}

p.setNext(v) {link p do nowego następcy, v}

USUWANIE

Wizualizacja remove(p), gdzie p = last()



USUWANIE - ALGORYTM

Algorytm remove(p):

t ← p.element {tymczasowa zmienna do przechowywania zwracanej wartości}

(p.getPrev()).setNext(p.getNext()) {usuwanie linków do p}

(p.getNext()).setPrev(p.getPrev())

p.setPrev(null) {usuwanie linków z p}

p.setNext(null)

return t

WYDAJNOŚĆ

- W implementacji listy dwukierunkowej stwierdzimy, że:
  - Wykorzystane przez listę miejsce dla n elementów jest w O(n)
  - Miejsce wykorzystane przez każdy element listy jest w O(1)

SEKWENCJA

- Sekwencja jest strukturą danych rozszerzającą definicję listy
- Zawiera funkcje pozwalające na dostęp do elementu poprzez podanie jego indeksu (jak dla wektora)
- Interfejs składa się z operacji na liście oraz:
  - atIndex(i): Zwraca element na pozycji i.
  - indexOf(p): Zwraca indeks elementu p.

```
class NodeSequence :  public NodeList {
 public:
   Iterator atIndex( int i)  const;
                              // zwraca element na pozycji i

   int indexOf( const Iterator& p)  const;
                              // zwraca indeks elementu p
};
```


STOSY

ABSTRACT DATA TYPES (ADT)

Abstract data type (ADT) jest abstrakcją struktury danych

ADT definiuje:
- Przechowywane dane
- Operacje na danych
- Definicję możliwych błędów tych operacji

Przykład: ADT modelujące prosty system obrotu papierami wartościowymi
- Przechowywane dane to zamówienia kupna/sprzedaży
- Dostępne operacje:
  - zamówienie kupna(akcja, udziały, cena)
  - zamówienie sprzedaży(akcja, udziały, cena)
  - void cancel(zamówienie)
- Wyjątki:
  - Kupno/Sprzedaż nieistniejąca akcja
  - Cancel nieistniejące zamównienie

STOS ADT

- ADT dla stosu przechowuje dowolne obiekty
- Dodawanie i usuwanie wykonywane jest na zasadzie LIFO - Last-in First-out
- Można przyrównać do zabawki dozującej cukierki (PEZ)
- Operacje na stosie:
  - push(element): dodawanie elementu
  - element pop(): usuwa i zwraca element z wierzchu stosu
- Dodatkowe operacje na stosach:
  - object top(): zwraca ostatni element umieszczony na stosie bez jego usuwania
  - integer size(): zwraca ilość przechowywanych elementów
  - boolean isEmpty(): mówi, czy są jakieś elementy przechowywane na stosie

WYJĄTKI

- Próba wywołania operacji ADT może czasami prowadzić do wystąpienia błędów, które nazywamy wyjątkami
  - Exceptions
- Wyjątki są „wyrzucane" (thrown) przez operacje i nie mogą być wykonywane
- Dla stosu operacje pop i top nie mogą zostać wykonane, jeśli stos jest pusty
- Próba wywołania metody pop i top dla pustego stosu wyrzuci wyjątek EmptyStackException

ZASTOSOWANIA STOSÓW

Zastosowania bezpośrednie

Historia odwiedzanych stron w przeglądarce internetowej
Sekwencja operacji „cofnij" w edytorze tekstu
Łańcuch wywołania metod w Wirtualnej maszynie Javy

Zastosowania pośrednie

Pomocnicza struktura danych dla algorytmów
Składowa innych struktur danych

STOS BAZUJĄCY NA TABLICY

- Prostym sposobem implementacji Stosu jest zastosowanie tablicy
- Dodajemy elementy od lewej do prawej
- Dodatkowa zmienna kontrolująca indeks elementu na wierzchu stosu

```
Algorytm size()
  return t + 1

Algorytm pop()
  if isEmpty() then
    throw EmptyStackException
  else
    t ← t − 1
    return S[t + 1]
```

STOS BAZUJĄCY NA TABLICY

- Tablica przechowująca elementy stosu może się zapełnić
- Operacja push() powinna zatem wyrzucać wyjątek FullStackException
  - Ograniczenie związane z implementacją bazującą na tablicy
  - Nie zawsze występujące przy implementacji stosu

```
Algorytm push(e)
  if t = S.length − 1 then
    throw FullStackException
  else
    t ← t + 1
    S[t] ← e
```

WYDAJNOŚĆ I OGRANICZENIA

Wydajność

Niech n będzie ilością elementów na stosie
Wykorzystane miejsce będzie równe O(n)
Każda operacja będzie działała w czasie O(1)

Ograniczenia

Maksymalny rozmiar stosu musi zostać zdefiniowany *a priori* i nie może zostać zmieniony
Próba dodania nowego elementu do pełnego stosu powoduje wyjątki typowe dla danej implementacji

STOS BAZUJĄCY NA POWIĘKSZANEJ TABLICY

- Wykonując operację push, kiedy tablica jest pełna, zamiast wyrzucania wyjątku, możemy możemy zastąpić ją większą tablicą
- Jak duża powinna być nowa tablica?
  - Strategia inkrementalna: zwiększ rozmiar o stałą c
  - Strategia podwajania: podwój rozmiar tablicy

```
Algorytm push(e)
  if t = S.length − 1 then
    A ← nowa tablica o rozmiarze…
    for i ← 0 to t do
      A[i] ← S[i]
    S ← A
  t ← t + 1
  S[t] ← e
```


PORÓWNANIE STRATEGII

- Porównamy strategię inkrementalną i podwajania poprzez analizę całkowitego czasu **T(n)** niezbędnego do wykonania n operacji push()
- Zakładamy, że rozpoczynamy od pustego stosu reprezentowanego przez tablicę o rozmiarze 1
- Czasem średnim będzie średni czas operacji push niezbędny do wykonania serii operacji, np.: T(n)/n

STRATEGIA INKREMENTALNA

- Tablica zostanie zastąpiona k = n/c razy
- Całkowity czas T(n) wykonania n operacji push jest proporcjonalny do:

$$n + c + 2c + 3c + 4c + \ldots + kc =$$

$$n + c(1 + 2 + 3 + \ldots + k) =$$

$$n + ck(k + 1)/2$$

- Ponieważ c jest stałą, T(n) jest w $O(n + k^2)$, tj. $O(n^2)$
- Średni czas operacji push jest w O(n)

STRATEGIA PODWAJANIA

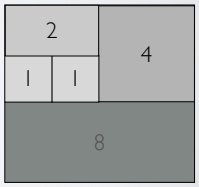
- Tablica zostanie zastąpiona $k = \log_2 n$ razy
- Całkowity czas T(n) wykonania n operacji push jest proporcjonalny do:

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$

$$n + 2^{k+1} - 1 = 2n - 1$$

- T(n) jest w O(n)
- Średni czas operacji push jest w O(1)

ciąg geometryczny



INTERFEJS STOSU W C++

- Interfejs jest odniesieniem do ADT stosu
- Wymaga zdefiniowania klasy EmptyStackException
- Najbardziej zbliżoną konstrukcją jest `vector`

```
template <typename Object>
class Stack{
public:
   int size();
   bool isEmpty();
   Object& top()
         throw(EmptyStackException);
   void push(Object o);
   Object pop()
         throw(EmptyStackException);
}
```


SPRAWDZANIE DOPASOWANIA NAWIASÓW

Każdy nawias "(", "{", lub "[" musi być sparowany z odpowiadającym mu nawiasem ")", "}", lub "]"

prawidłowo: ( )(( )){([( )])}
prawidłowo: (( )(( )){([( )])})
błędnie:)(( )){([( )])}
błędnie: ({[ ])}
błędnie: (

LISTA JEDNOKIERUNKOWA JAKO STOS

- Możemy zaimplementować stos za pomocą listy jednokierunkowej
  - Górny element jest przechowywany w pierwszym węźle listy
- Wykorzystanie miejsca to O(n), a każda operacja stosu jest wykonywana w czasie O(1)