W. 4.1
Lista jednokierunkowa
(Singly linked list)

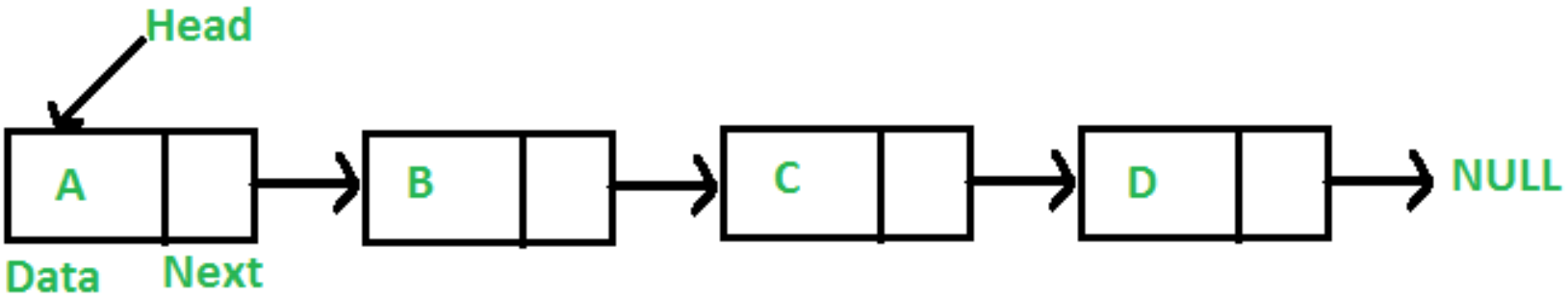
AST (ADT)

- AST = Abstrakcyjne Struktury Danych (ADT = Abstract Data Types) są to struktury danych, do których podany jest jedynie interfejs, t.j. sposób dostępu do tej struktury przez funkcje/metody.
- Ponieważ podanie interfejsu (sposobu obsługi) nie determinuje implementacji tej struktury (jak ona rzeczywiście działa), to sprawia, że nie możemy mówić o złożoności obliczeniowej danej funkcji z interfejsu. Możemy to powiedzieć jedynie o konkretnej implementacji AST!!!

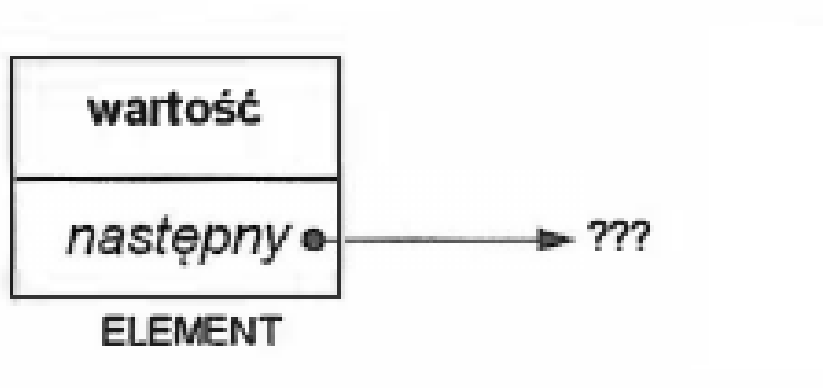
Problem

- Potrzebujemy struktury danych, która gromadzi w sposób uporządkowany (kolejność) elementy.
- Taka struktura nazywa się listą.
- **Lista jednokierunkowa** to struktura, którą łatwiej możemy przeglądać tylko w określonym kierunku – od początku do końca.

Idea list jednokierunkowej



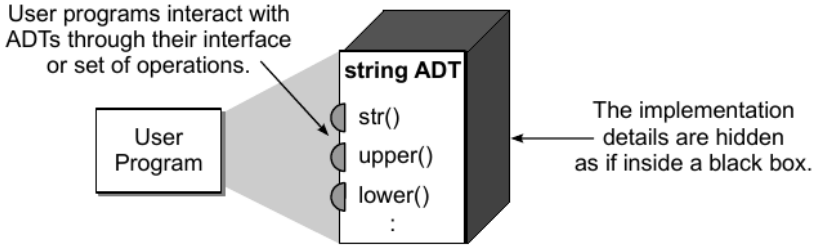
Czego potrzebujemy?



- Lista jednokierunkowa składa się z węzłów (nodes) które zawierają dwa pola:
  - Wartość – wartość składowana w liście.
  - Następny – wskaźnik do kolejnego elementu listy.

Interfejs - ogólnie

- Interfejs listy określa sposób dostępu i manipulacji strukturą programistyczną.
- Interfejs pozawala na oddzielenie użycia danej struktury danych od jej implementacji.
- Określając interfejsy możemy podzielić zadania programistyczne na niezależne zespoły programistyczne. Wówczas nie jest ważne jak dana struktura została zaimplementowana przez zespół (chociaż na ogół musi spełniać pewne założenia). Jeżeli interfejs jest zaimplementowany poprawnie wówczas integracja struktur od różnych zespołów programistycznych w jeden program nie będzie stanowiła problemu. Dzięki temu praca nad dużymi projektami informatycznymi jest łatwiejsza i możliwa do podzielenia na zespoły
- O wytwarzaniu oprogramowania w dużej skali (podział na moduły, zarządzanie i praca w grupie) zajmiemy się na kursie Inżynierii systemów (tj. Inżynierii oprogramowania).



- Przykład interfejsu do typu string -napis.
- Implementacja jest ukryta dla użytkownika – programu.

Interfejs listy

- Podstawowy interfejs AST:
  - push(value) -> Add a node in the beginning
  - append(value) -> Add a node in the end
  - pop() -> Remove a node from the end
  - popFirst() -> Remove a node from the beginning
  - head() -> Return the first node
  - tail() -> Return the last node
  - remove(Node) -> Remove Node from the list
- Nazwy mogą się różnić i nie wszystkie muszą funkcje muszą być implementowane.
- Pamiętaj, że na tym etapie nie możemy jeszcze mówić o złożoności obliczeniowej funkcji z interfejsu – do tego potrzebne są konkretne implementacje tych funkcji!!!

Węzeł - Node

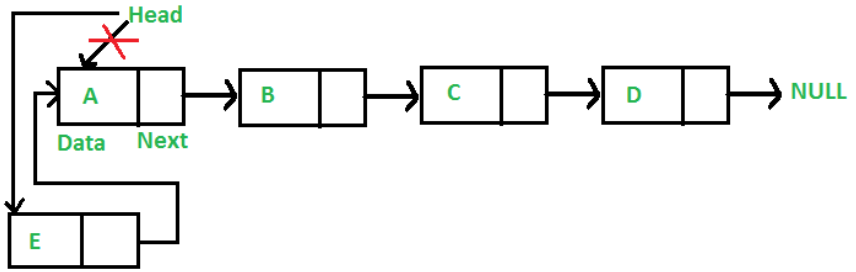
- class Node
- {
- public:
- int data;
- Node *next;
- };

- Atrybuty klasy Node:
  - Data – dane zapisane w węźle listy;
  - Next – wskaźnik do kolejnego węzła

Wstawianie elementów

Push – wstawiamy element na początek listy

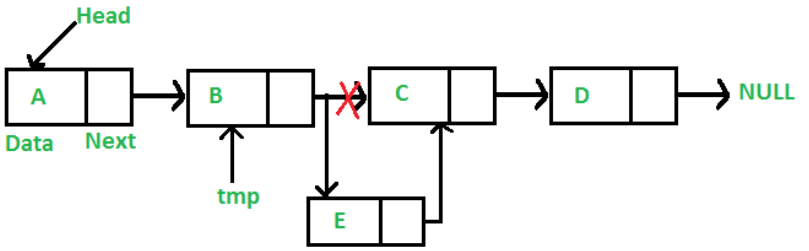
- void push(Node** head_ref, int new_data)
- {
- /* 1. allocate node */
- Node* new_node = new Node();
-
- /* 2. put in the data */
- new_node->data = new_data;
-
- /* 3. Make next of new node as head */
- new_node->next = (*head_ref);
-
- /* 4. move the head to point to the new node */
- (*head_ref) = new_node;
- }



- Node ** head_ref – wskaźnik do wskaźnika pokazującego na głowę/początek(head) listy. Robimy tak, gdyż ta wartość będzie zmieniana w funkcji.
- Rozpisz sekwencję działań tej funkcji na kartce.

InsertAfter – wstaw za węzłem

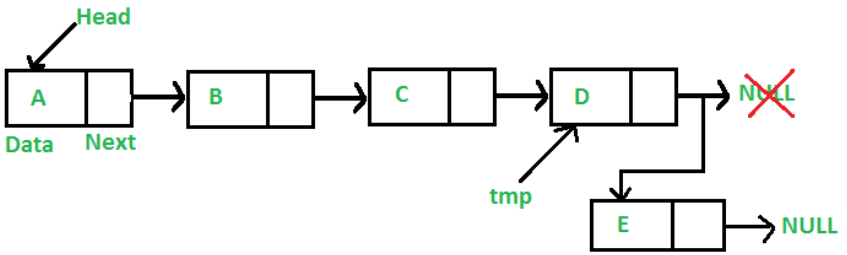
```
void insertAfter(Node* prev_node, int new_data)
{
/*1. check if the given prev_node is NULL */
if (prev_node == NULL)
{
   cout<<"the given previous node cannot be NULL";
   return;
}

/* 2. allocate new node */
Node* new_node = new Node();

/* 3. put in the data */
new_node->data = new_data;

/* 4. Make next of new node as next of prev_node */
new_node->next = prev_node->next;

/* 5. move the next of prev_node as new_node */
prev_node->next = new_node;

}
```



- Podajemy element do wstawienia i węzeł (jego wskaźnik) za który wstawiamy.

Append – wstaw na koniec

```
void append(Node** head_ref, int new_data)
{
/* 1. allocate node */
Node* new_node = new Node();
Node *last = *head_ref; /* used in step 5*/

/* 2. put in the data */
new_node->data = new_data;

/* 3. This new node is going to be the last node, so make next of
it as NULL*/
new_node->next = NULL;

/* 4. If the Linked List is empty,  then make the new node as head */
if (*head_ref == NULL)
{  *head_ref = new_node;       return;  }

/* 5. Else traverse till the last node */
while (last->next != NULL)
last = last->next;

/* 6. Change the next of last node */
last->next = new_node;
return; }
```



- Zauważ, że w kroku 5 używamy pętli while, aby przejść do końcowego węzła (tego który ma next=NULL.

Wypisz listę

- void printList(struct Node *node)
- {
- **while (node != NULL)** {
- printf(" %d ", node->data);
- node = node->next; }
- }

Użycie

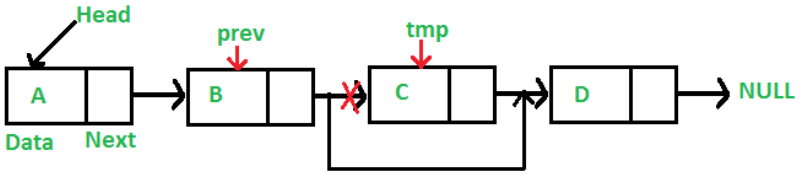
```
int main()
{
/* Start with the empty list */
Node* head = NULL;
// Insert 6. So linked list becomes 6->NULL
append(&head, 6);
// Insert 7 at the beginning.
// So linked list becomes 7->6->NULL
push(&head, 7);
// Insert 1 at the beginning.
// So linked list becomes 1->7->6->NULL
push(&head, 1);
// Insert 4 at the end. So
// linked list becomes 1->7->6->4->NULL
append(&head, 4);
// Insert 8, after 7. So linked
// list becomes 1->7->8->6->4->NULL
insertAfter(head->next, 8);
cout<<"Created Linked list is: ";
printList(head);
return 0;  }
```

- Na początku tworzymy pustą listę:
  - Node* head = NULL;
- Ten wskaźnik następnie jest używany do wskazywania na głowę listy.
- Nie możesz „zgubić” tego wskaźnika, bo stracisz dostęp do listy i w pamięci zostanie zajęta struktura danych do której nie można się odwołać.

Usuwanie

Usuwanie elementu

```
void deleteNode(struct Node **head_ref, int key)
{ // Store head node
struct Node* temp = *head_ref, *prev;
// If head node itself holds the key to be deleted
if (temp != NULL && temp->data == key)
{
*head_ref = temp->next; // Changed head
free(temp);           // free old head
return;
}
// Search for the key to be deleted, keep track of the
// previous node as we need to change 'prev->next'
while (temp != NULL && temp->data != key)
{ prev = temp;
temp = temp->next; }
// If key was not present in linked list
if (temp == NULL) return;
// Unlink the node from linked list
prev->next = temp->next;
free(temp); // Free memory  }
```



- Zauważ, że linijka:
  - struct Node* temp = *head_ref, *prev;
- oznacza:
  - struct Node *temp = *head_ref;
  - struct Node *prev;

Usuwanie elementu o danym indeksie

```
void deleteNode(struct Node **head_ref, int position)
{  // If linked list is empty
if (*head_ref == NULL)
return;
// Store head node
struct Node* temp = *head_ref;
// If head needs to be removed
if (position == 0) {
 *head_ref = temp->next; // Change head
free(temp);           // free old head
return; }
// Find previous node of the node to be deleted
for (int i=0; temp!=NULL && i<position-1; i++)
temp = temp->next;
// If position is more than number of ndoes
if (temp == NULL || temp->next == NULL)
return;
// Node temp->next is the node to be deleted
// Store pointer to the next of node to be deleted
struct Node *next = temp->next->next;
// Unlink the node from linked list
free(temp->next); // Free memory
temp->next = next; // Unlink the deleted node from list  }
```

- Usuwanie elementu o indeksie **k** polega na:
  - Przesunięciu się do węzła o indeksie **k-1;**
  - Zapamiętaniu wskaźnika do elementu **k+1;**
  - Usunięciu węzła **k**
  - Powiązaniu węzłów **k-1** oraz **k+1**;

Usuwanie całej listy

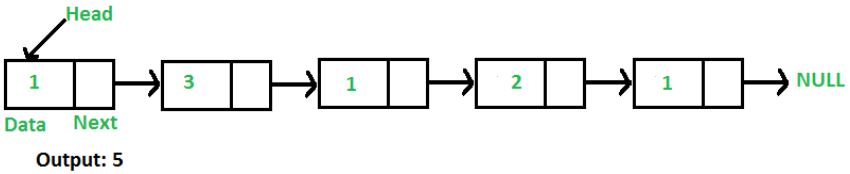
- void deleteList(Node** head_ref)
- { /* deref head_ref to get the real head */
- Node* current = *head_ref;
- Node* next;
- **while (current != NULL)**
- **{ next = current->next;**
- **free(current);**
- **current = next; }**
- /* deref head_ref to affect the real head back in the caller. */
- *head_ref = NULL; }

- Zauważ, że usuwane jest zawarte w pętli while – nie musimy znać liczby elementów listy.

Długość listy

Długość listy iteracyjnie (liczba elementów)

- int getCount(Node* head)
- {
- int count = 0; // Initialize count
- Node* current = head; // Initialize current
- **while (current != NULL)**
- **{ count++;**
- **current = current->next; }**
- return count;
- }



- Ponownie używamy pętli while do iteracji po liście.
- Spróbuj napisać wersję rekurencyjną getCount!
  - getCount musi wywoływać samą siebie.

Rozwiązanie

- int **getCount**(struct Node* head)
- { // Base case
- if (head == NULL)
- return 0;
-
- // count is 1 + count of remaining list
- return 1 + **getCount**(head->next);
- }

Przeszukiwanie listy

Szukanie iteracyjnie

- bool search(Node* head, int x)
- {
- Node* current = head; // Initialize current
- while (current != NULL)
- {
- if (current->key == x)
- return true;
- current = current->next;
- }
- return false;
- }

- Funkcja search zwraca prawdę gdy element x jest na liście lub fałsz w przeciwnym wypadku.
- Nie musimy znać indeksu elementu, gdyż usuwanie nie potrzebuje indeksu elementu.
- Uwaga:
  - Zamiana elementu x na y polega na usuwaniu x, a następnie dodaniu elementu y.
  - Proszę napisać taką funkcję replace(x,y); Sprawdź, czy element istnieje przed jego zamianą!!!
- Spróbuj zaimplementować search rekurencyjnie.

Szukanie rekurencyjne

```
bool search(struct Node* head, int x)
{  // Base case
    if (head == NULL)
        return false;

    // If key is present in current node, return true
    if (head->key == x)
        return true;

    // Recur for remaining list
    return search(head->next, x);  }
```

Element o danym indeksie

Element o indeksie n - iteracja

```
int GetNth(Node* head, int index)
{
Node* current = head;
// the index of the  node we're currently looking at
int count = 0;
while (current != NULL)
{  if (count == index)
return(current->data);
count++;
current = current->next; }

/* if we get to this line, the caller was asking
for a non-existent element  so we assert fail */
assert(0);
}
```

- Po liście iterujemy pętlą while, ale jednocześnie w pętli sprawdzamy, czy dotarliśmy do węzła o danym indeksie.
- Napisz GetNth rekurencyjnie.

Element o indeksie n - rekurencja

- int GetNth(struct Node *head,int n)
- { int count = 1;
- //if count equal too n return node->data
- if(count == n)
- return head->data;
- //recursively decrease n and increase head to next pointer
- return GetNth(head->next, n-1);
- }

Zadanie

- Napisz funkcję podającą n-ty element od końca:
  - Oblicz długość listy -> N
  - Dojdź do elementu N-n od głowy.
- Napisz funkcję łączącą dwie listy dodając jedną na koniec drugiej:
  - Przejdź do końcowego węzła pierwszej listy.
  - Ustaw next dla pierwszej listy na head drugiej listy.

Python

Węzel

- class ListNode:
- def __init__(self, data) :
- self.data = data
- self.next = None

- Tworzenie listy:
  - a=ListNode(11)
  - a.next=ListNode(2)
  - a.next.next=ListNode(5)

Wyświetlanie listy

- def traversal(head):
- curNode = head
- while curNode is not None :
- print curNode.data
- curNode = curNode.next

Szukanie elementu

- def unorderedSearch(head, target):
- curNode = head
- while curNode is not None and curNode.data != target :
- curNode= curNode.next
- return curNode is not None

Push/prepend

- # Given the head pointer,
- #create node
- prepe newNode = ListNode(item)
- # set next of new node to head of the list
- newNode.next = head
- #set head to new node
- head = newNode

Zadanie

- Napisz pozostałe procedury:
  - Usuwanie – python automatycznie zarządza pamięcią i odśmiecacz (Garbage collector) usuwa nieużywane obiekty, ale możemy wymusić usunięcie przez:
    - **del** Zmienna/Obiekt
  - Dodawanie elementów.

Złożoność obliczeniowa

Wybrane metody

- Oblicz złożoności:
  - Push/preprend jest klasy O(1)
  - Append jest klasy O(n)
  - Search jest średnio (i w najgorszym wypadku) klasy O(n)
  - Usuwanie elementu jest średnio (i w najgorszym wypadku) klasy O(n)
  - Obliczanie długości jest klasy O(n)

Literatura dodatkowa

- https://www.geeksforgeeks.org/data-structures/linked-list/#singlyLinkedList
- Rozdział 5 - Algorytmy. Struktury danych i techniki programowania.
- Rozdziały 10 – T. Cormen, ‘Wprowadzenie do algorytmów’, PWN
- Rozdział 6.2, 2 - Data Structures and Algorithms using Python.

Koniec