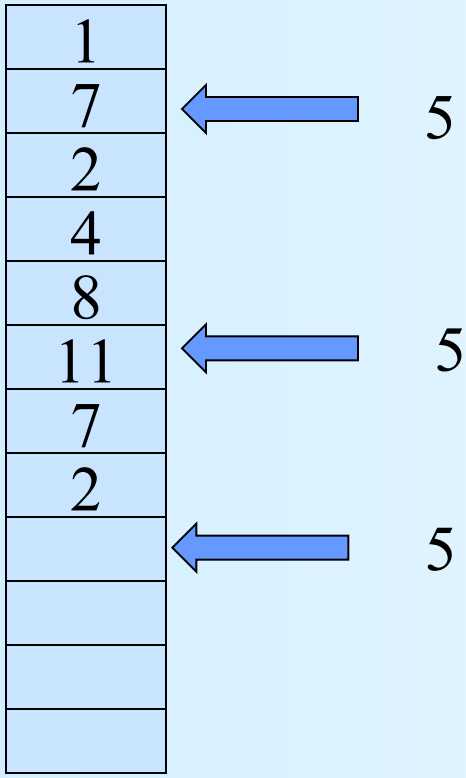
Algorytmy i struktury danych

struktury rekurencyjne

tablice, listy,

stosy, kolejki

Tablica – struktura o swobodnym dostępie



C/C++

```
const int MAX = 12;
int Data[MAX] =
  {1,7,2,4,8,11,7,2};
```

Pseudokod

```
Data = [1,7,2,4,8,11,7,2,0,0,0,0]
```

 # defacto w Pythonie nazywa się to lista

```
Data[5]  = 5
int i = 1
Data[i] = 5
Data[i+2] = 5
```

Jeszcze o pseudokodzie

- ✓ zakres w Pythonie jest listą i jest indeksowany od 0
- ✓ czas dostępu jest jak dla listy (tj. o(n))
- ✓ sprawdzanie długości możliwe jest funkcją len
- ✓ iterowanie

```
for element in A : .....
```

- ✓ do definiowania zakresów służy funkcja **range**

`range(1,10)` zwraca zakres 1,2,3,4,5,6,7,8,9
`range(1,9,4)` zwraca zakres 1, 5
`range(5,3)` zwraca zakres []
`range(5,3,-1)` zwraca zakres [5, 4]

Kod działa, ale w Pythonie podane dla tablic czasy nie są prawdziwe gdyż w Pythonie zapis [] oznacza listę

Jeszcze o pseudokodzie

- ✓ zmiany parametrow w Pythonie nie "wydostaja sie poza funkcje". Mimo to w pseudokodzie unikać będziemy ich zmian.
- ✓ W Pythonie nie trzeba zwalniać pamięci i w związku z tym pseudokod kod nie będzie obejmować takich operacji

Tablica – podstawowe operacje

- ✓ Struktura o dostępie swobodnym

Operacje

- ✓ Zapis - Θ(1) :
  - `A[i] = newValue`
- ✓ Odczyt - Θ(1) :
  - `value = A[i]`
  - `if value == A[i]`
- ✓ Przeszukiwanie - O(n) lub O(log n) w zależności od uporządkowania
  - `for e in A : ...`
  - ```
    for i in range(0,len(a)) :
        if A[i] == s : ...
    ```
  - `element = find(A,pattern)`
    ```

Tablica – dodatkowe operacje

- ✓ Wstawianie - O(n) – n = liczba elementów przed

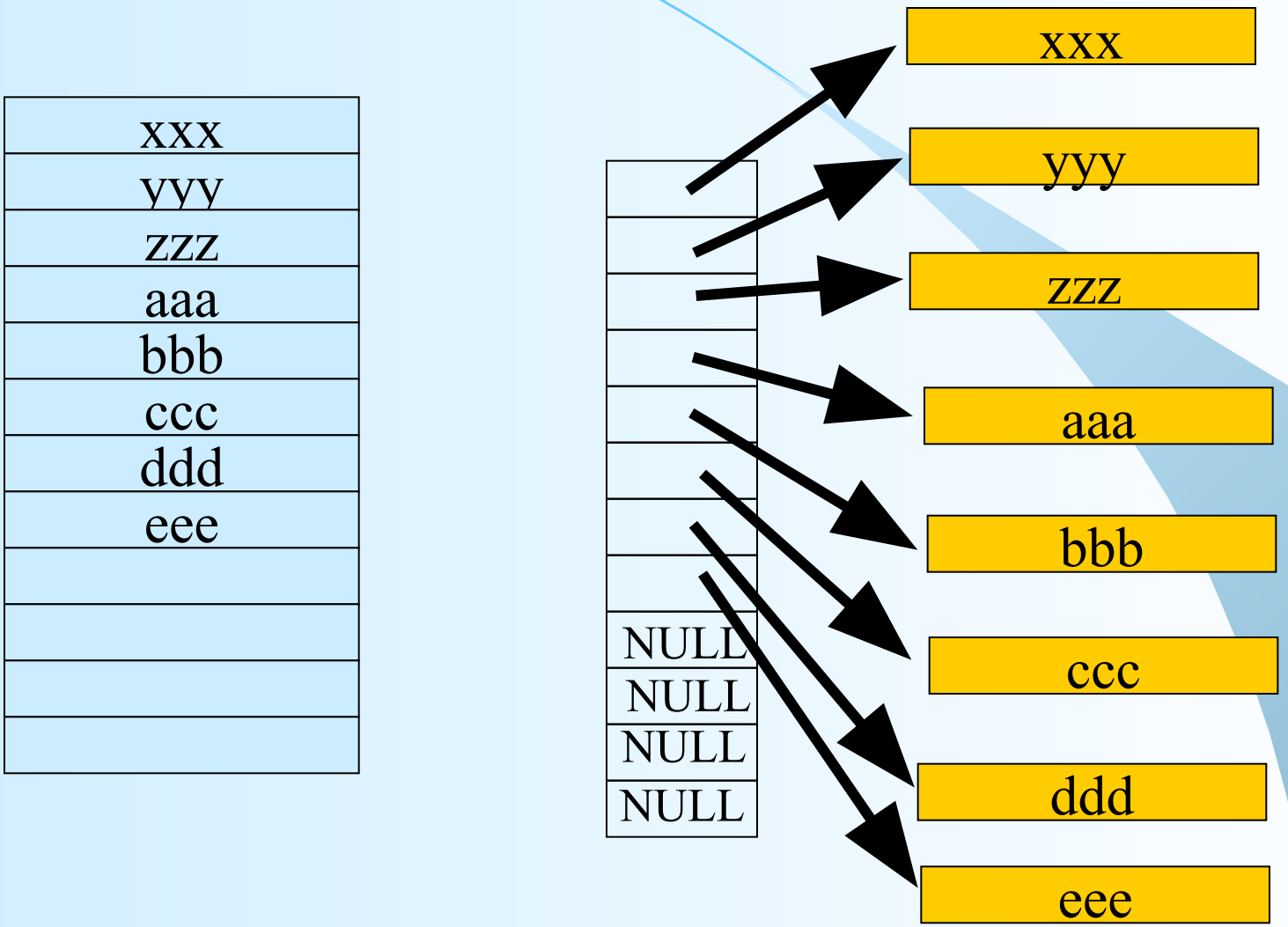
```
for i in range(n-1, k, -1) : A[i+1] = A[i]
A[k] = newElement
```

- ✓ Zmiana rozmiaru - Θ(n) :

```
C/C++ :
Resize(A, newSize)
{
    tmp = alokuj_nowa_tablice
    kopiuj_elementy_z_A_do_nowej_tablicy
    zwolnij_A
    return tmp
}
```

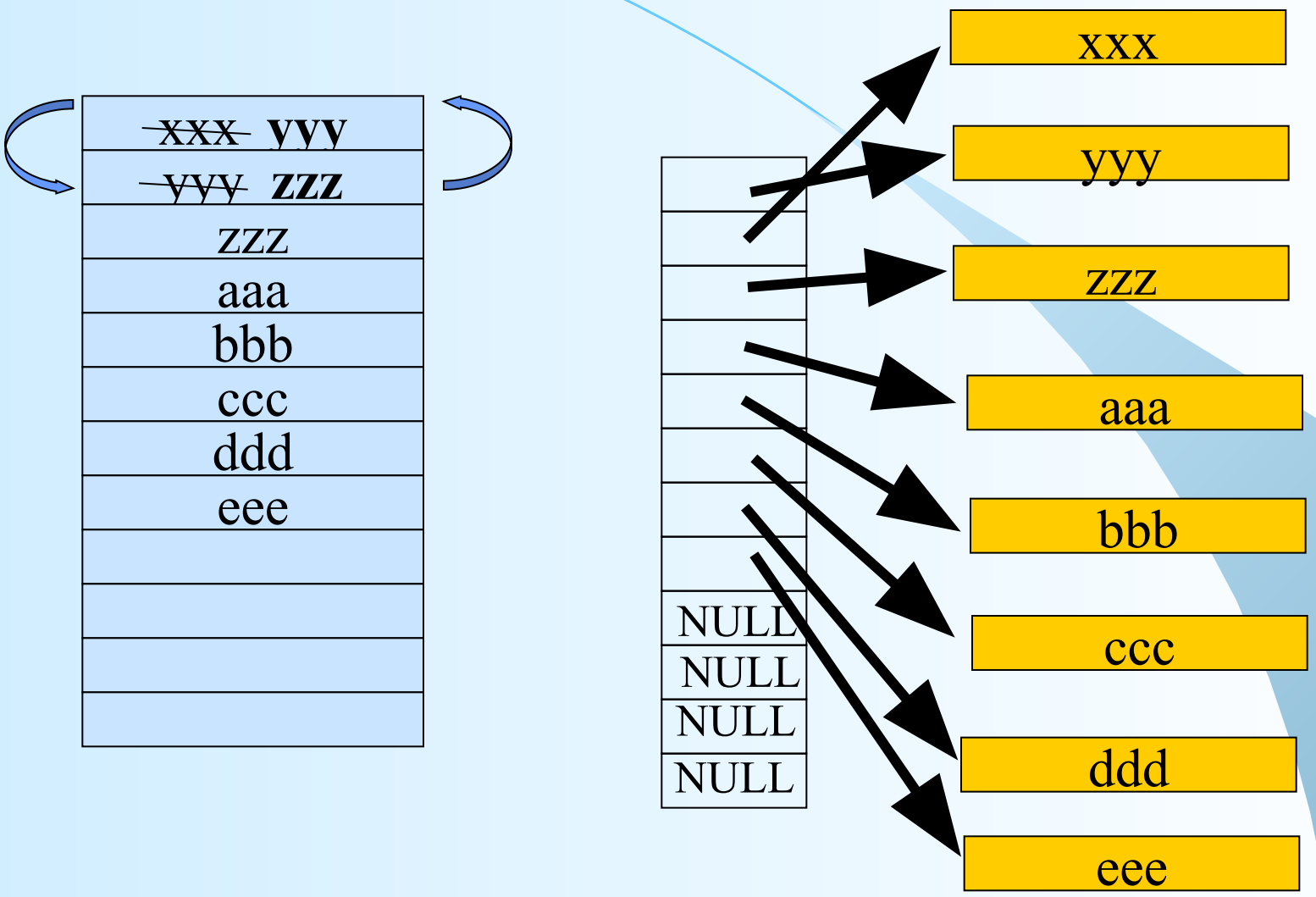
Tablica wskaźników

`C/C++  ELEMENT Array[MAX];` `ELEMENT *Array[MAX];`



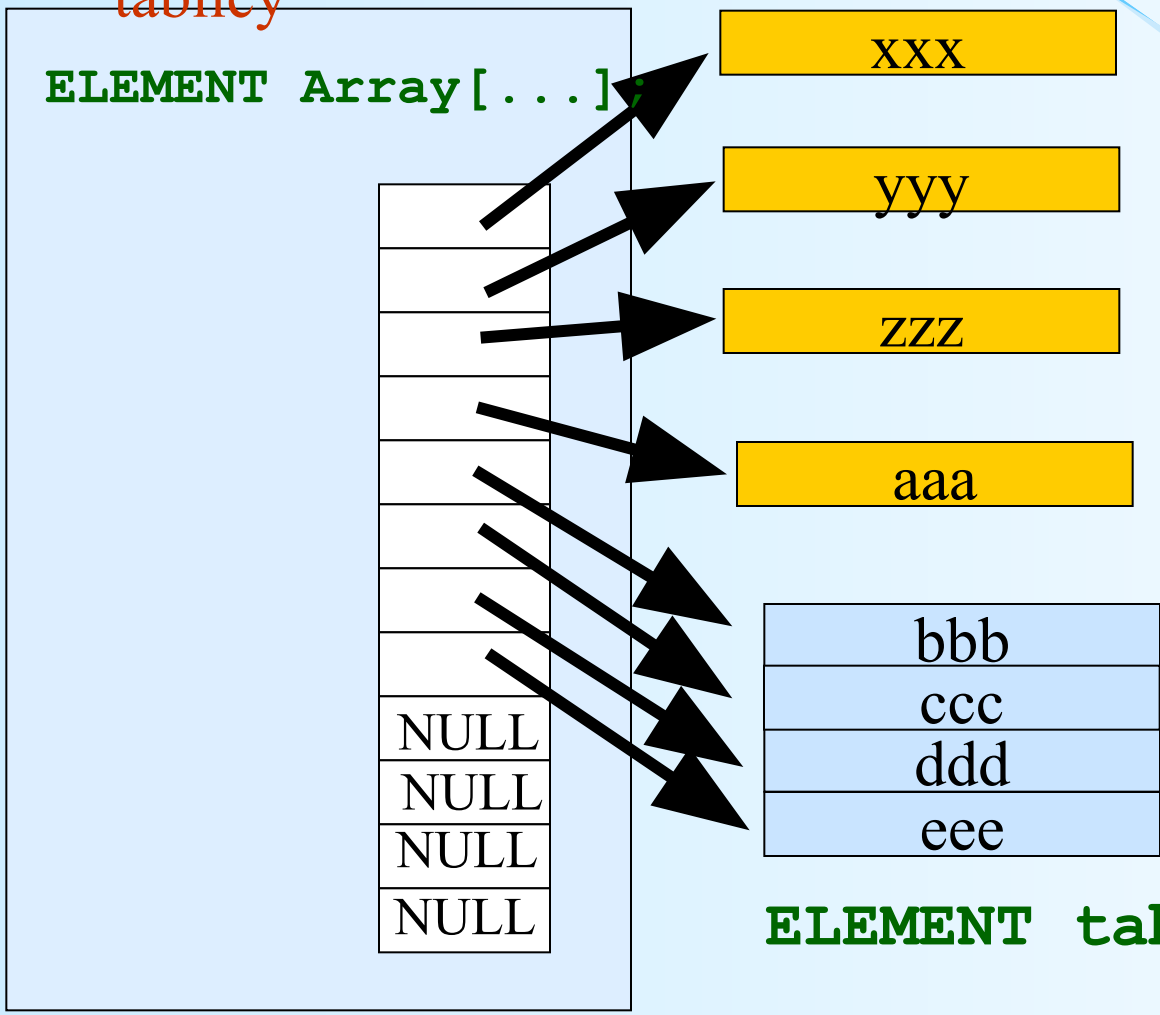
W jęz. Python, .Net, Java obiekty klas mają charakter wkaźników

Zamiana elementów



UWAGA

Musimy mieć na co pokazywać, można alokować pojedyncze elementy lub skorzystać z dodatkowej tablicy



C/C++:

```
Element *Array[MAX];
for (i.... )
   A[i] = new Element;
```

```
for (....)
   Array[i] = &tab[j]
```

```
ELEMENT tab[?];
```

!! Należy wyzerować nieużywane komórki !!

Rekurencja danych

C/C++

```
typedef int DATA;
typedef struct
  SLNODE {
      DATA data;
      SLNODE *next;
} *PSLNODE;
SLNODE OneElement;
```

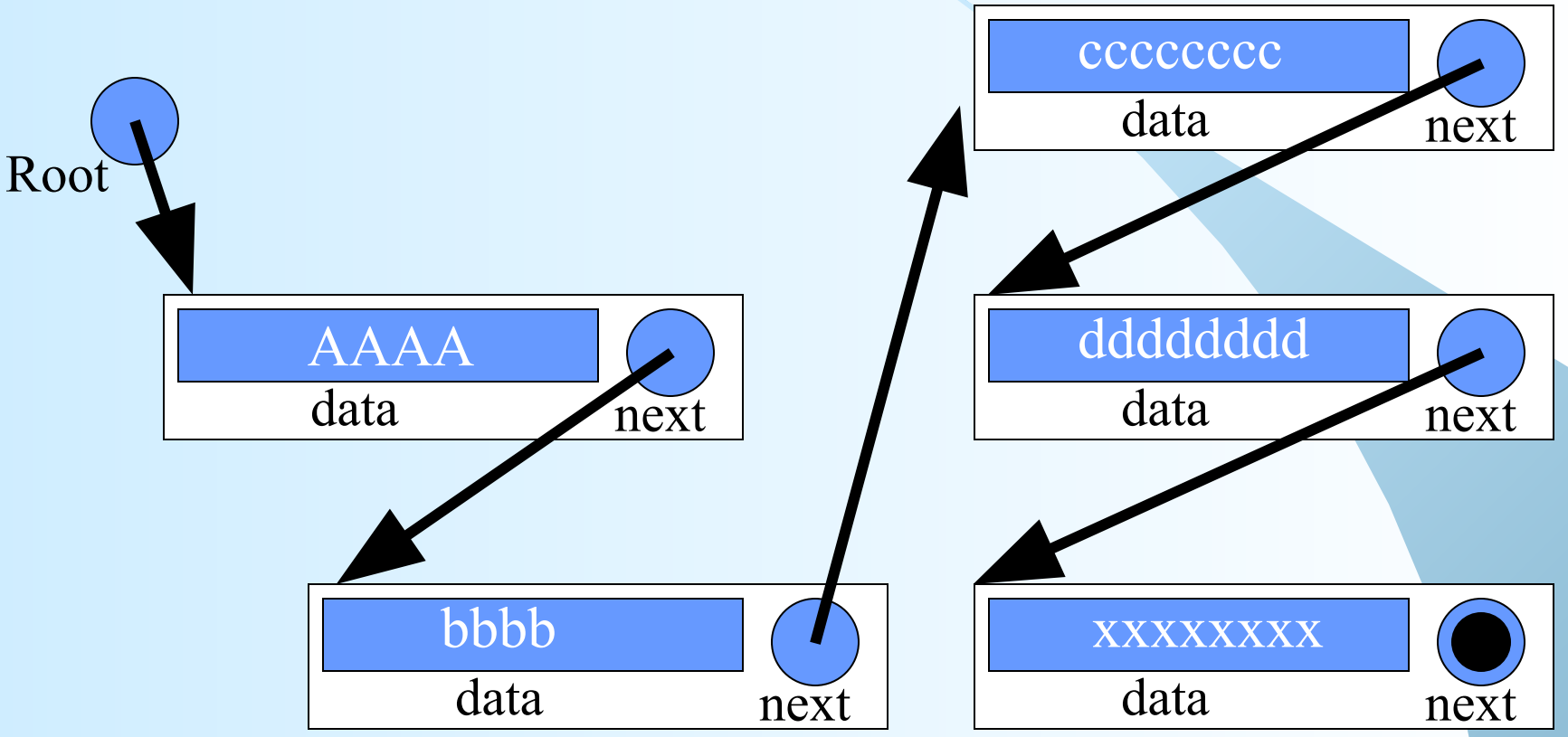
C/C++ - inaczej

```
typedef int DATA;
struct SLNODE;
typedef SLNODE *
  PSLNODE;
struct NODE {
      DATA data;
      PSLNODE next;
} OneElement;
```

Pseudokod :

```
class SLNODE :
    data = None
    next = None
```

Lista jednostronnie wiązana



Lista jednostronnie wiązana

```
class NODE :
    data = None
    next = None
```

```
def PrintList(firstNode) :
    tmp = firstNode
    while tmp != None :
        print(tmp.data)
        tmp = tmp.next
```

```
def GetListLen(firstNode) :
    cnt = 0
    tmp = firstNode
    while tmp != None :
        cnt = cnt+1
        tmp = tmp.next
    return cnt
```

Lista jednostronnie wiązana

```
def GetFirst(firstNode) :
    if firstNode == None :
        return None
    return firstNode


def GetLast(firstNode) :
    if firstNode == None :
        return None
    tmp = firstNode
    while tmp.next != None :
        tmp = tmp.next
    return tmp
```

Użycie :

```
lastNode = GetLast(list)
```

Lista jednostronnie wiązana

```
def AddFirst(firstNode, newNode) :
    newNode.next = firstNode
   return newNode
```

```
def AddLast(firstNode, newNode) :
    newNode.next = None
   if firstNode == None :
    return newNode
   tmp = firstNode
   while tmp.next != None :
    tmp = tmp.next
   tmp.next = newNode
   return firstNode
```

Użycie :

```
list = None
newNode = Node()
newNode.data = "abc"
list = AddLast(list,newNode)
```

Lista jednostronnie wiązana

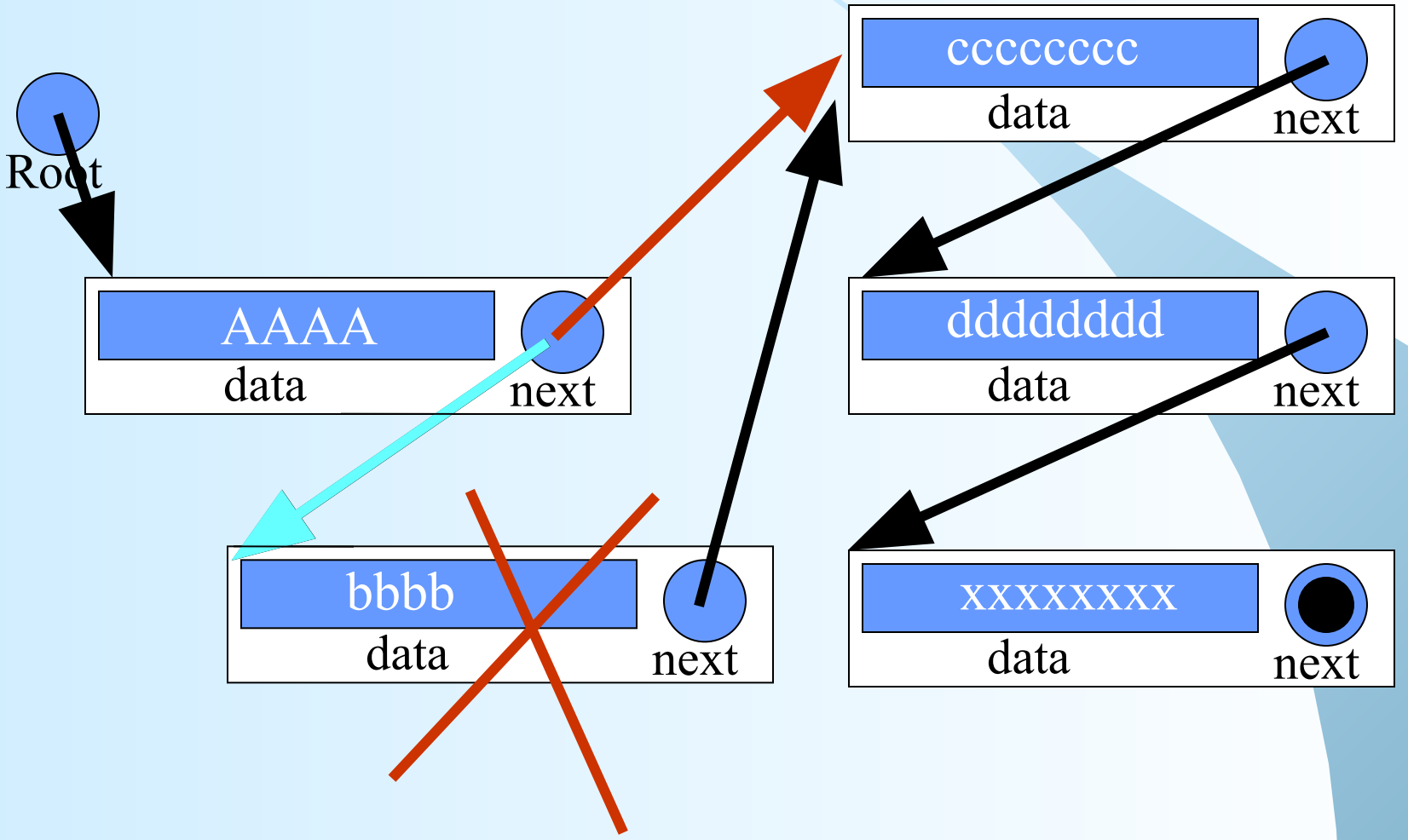
```
def RemoveFirst(firstNode) :
    if firstNode == None :
        return None
    return firstNode.next


def RemoveLast(firstNode) :
    if firstNode == None :
        return None
    if firstNode.next == None :
        return None
    tmp = firstNode
    while tmp.next.next != None :
        tmp = tmp.next
    tmp.next = None
    return firstNode
```

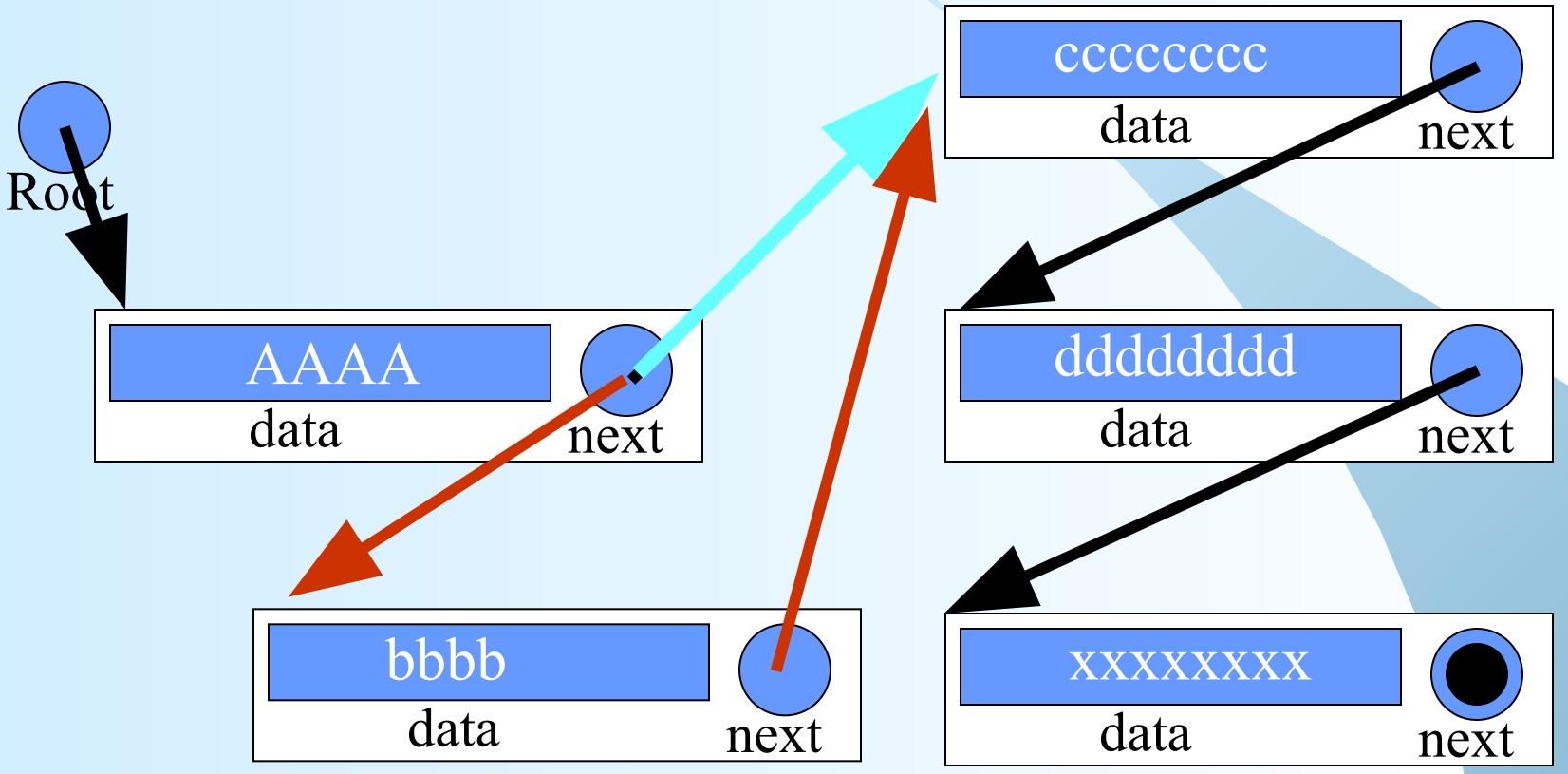
Użycie :

```
list = RemoveFirst(list)
```

Lista jednostronnie wiązana- usuwanie elementu



Lista jednostronnie wiązana- dodawanie elementu



Lista jednostronnie wiązana

```
def FindNode(firstNode, dataPattern) :
    tmp = firstNode
   while tmp != None :
      if tmp.data == dataPattern :
         return tmp
    tmp = tmp.next
   return None
```

```
def GetAtPos(firstNode, pos) :
   #funkcja zwraca element na pozycji pos liczac od 0
   #jesli nie ma tylu elementow zwraca None
   tmp = firstNode
   while tmp != None :
      if pos == 0 :
         return tmp
      pos = pos - 1
         tmp = tmp.next
   return None
```

Lista jednostronnie wiązana

```
def InsertAfter(newNode, node) :
    if node!= None :
        newNode.next = node.next
        node.next = newNode
def RemoveAfter(node) :
    if node!= None :
        node.next = node.next.next

        #POMIJAMY USUWANIE Z PAMIECI!
```

```
#InsertBefore i RemoveNode - wymagaja
# modyfikacji wezla wcześniejszego
#Czyli szukać również trzeba węzła
# wcześniejszego
```

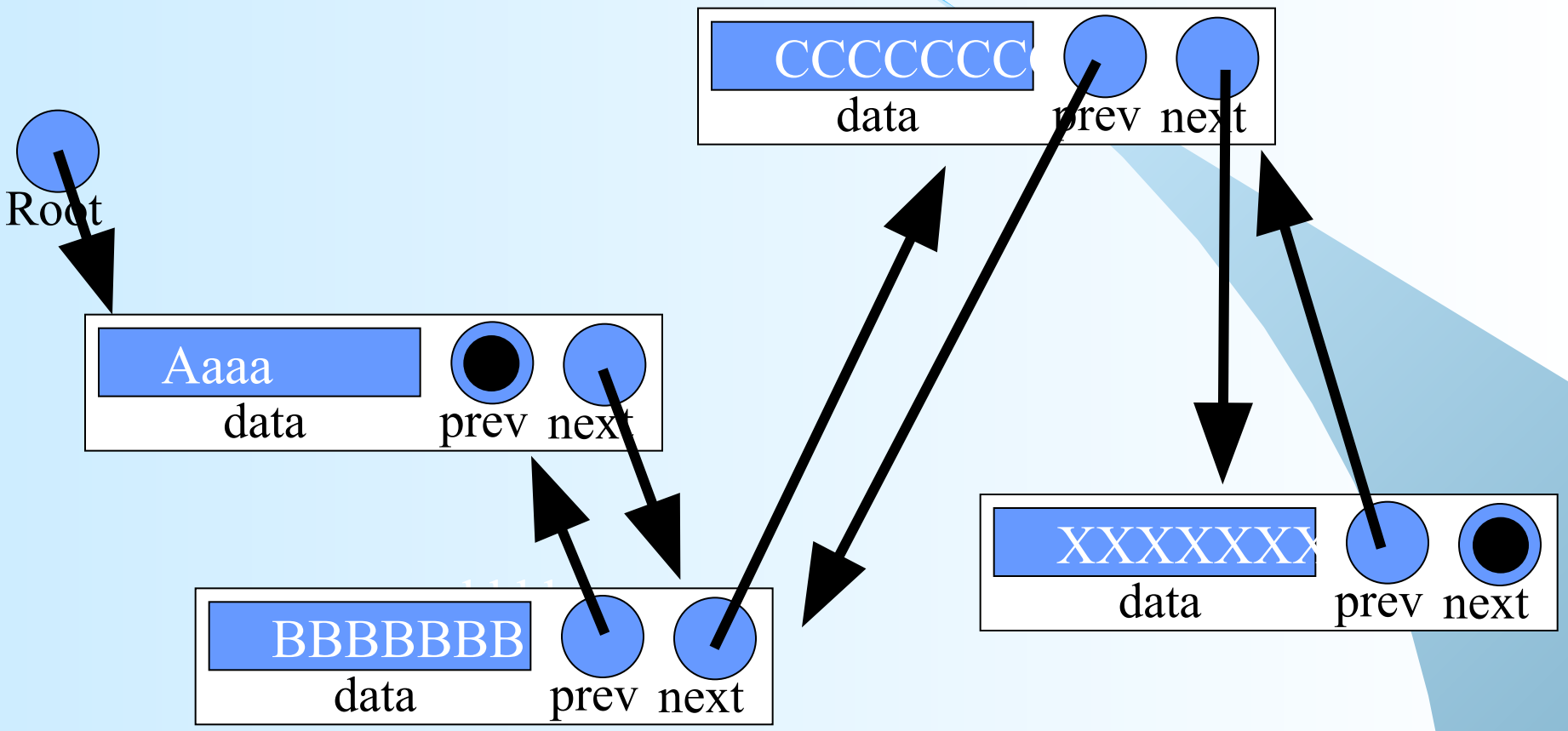
Lista jednostronnie wiązana

```
def InsertBefore(firstNode, newNode, dataPattern) :
    if firstNode == None :
        return firstNode
    if firstNode.data == dataPattern :
        newNode.next = firstNode
        return newNode
    tmp = firstNode
    while tmp.next != None :
            if tmp.next.data == dataPattern :
                newNode.next = tmp.next
                tmp.next = newNode
                break
            tmp = tmp.next
    return firstNode
```

Lista jednostronnie wiązana

```
def RemoveNode(firstNode, dataPattern) :
    if firstNode == None :
      return firstNode
  if firstNode.data == dataPattern :
      return firstNode.next
    tmp = firstNode
    while tmp.next != None :
          if tmp.next.data == dataPattern :
             tmp.next = tmp.next.next
             break
          tmp = tmp.next
    return firstNode
```

Lista dwustronnie wiązana



Lista dwustronnie wiązana

```
class DLNODE :
   data = None
   next = None
   prev = None
```

Identyczne będą funkcje :

```
PrintList(list)
GetListLen(list)
GetFirst(list)
GetLast(list)
GetAtPos(list, pos)
```

Lista dwustronnie wiązana

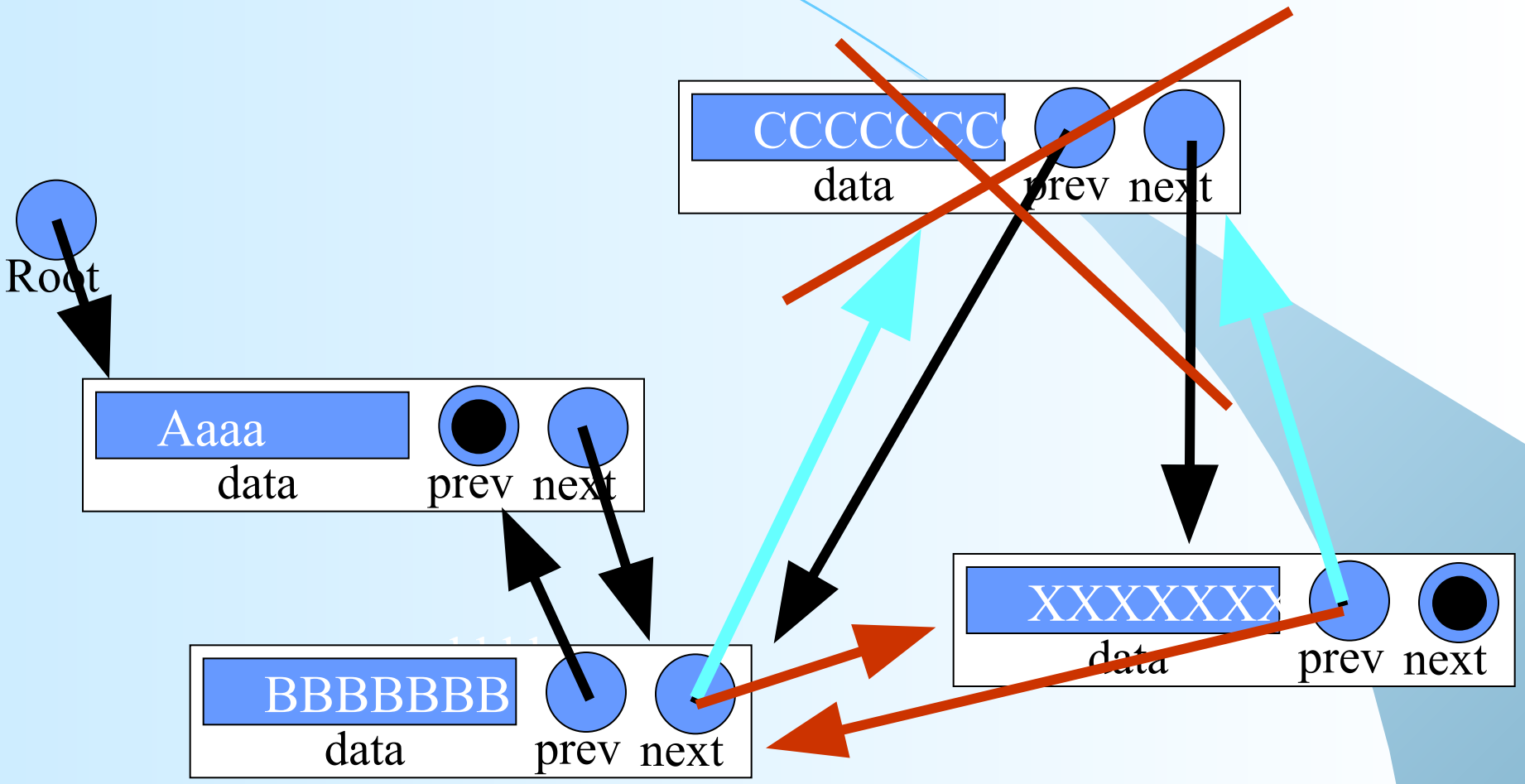
```
def AddFirst(firstNode, newNode) :
    newNode.prev = None
    newNode.next = firstNode
    if firstNode != None :
        firstNode.prev = newNode
    return newNode
```

```
def AddLast(firstNode, newNode) :
    last = GetLast(firstNode)
    newNode.prev = last
    if last == None :
        return newNode
    last.next = newNode
    return firstNode
```

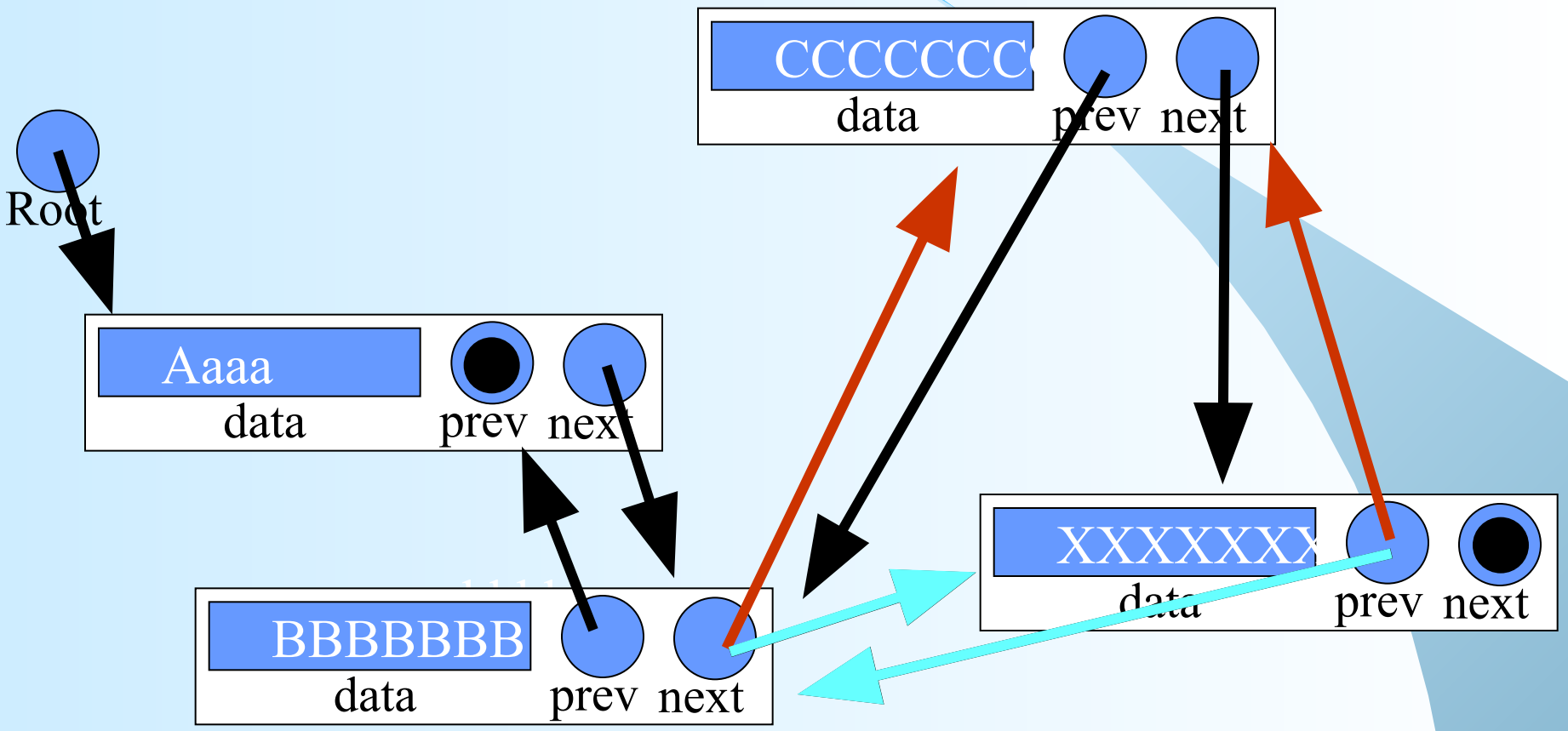
Użycie :

```
newNode = DLNODE()
newNode.data = "ABC"
list = AddLast(list,newNode)
```

Lista dwustronnie wiązana usuwanie elementu



Lista dwustronnie wiązana wstawianie elementu



Lista dwustronnie wiązana

```
def InsertAfter(newNode, node) :
    if node == None :
        return
    newNode.next = node.next
    newNode.prev = node
    if node.next != None :
        node.next.prev = newNode
    node.next = newNode


def RemoveAfter(node) :
    if node == None :
        return
    if node.next == None :
        return
    node.next = node.next.next
    if node.next != None :
        node.next.prev = node
```

Lista dwustronnie wiązana

```
def InsertBefore (firstNode, newNode, node) :
  if node == None :
     return firstNode
  newNode.next = node
  newNode.prev = node.prev
  node.prev = newNode
  if newNode.prev == None : #dodajemy wezel na pocz.
     return newNode
  newNode.prev.next = newNode
  return firstNode
```

Użycie :

```
node = FindNode(list, whatToFind)
list = InsertBefore(list, newNode, node)
```

Lista dwustronnie wiązana

```
def RemoveNode(firstNode, node) :
  if node == None :
     return firstNode
  if node.next != None :
     node.next.prev = node.prev
  if node.prev == None : #tzn. firstNode==node
     return node.next
  node.prev.next = node.next
  return firstNode
```

Użycie :

```
node = FindNode(list, whatToFind)
list = RemoveNode(list, node)
```

Lista dwustronnie wiązana

```
def RemoveFirst(firstNode) :
   return RemoveNode(firstNode, firstNode)

def RemoveLast(firstNode) :
   last = GetLast(firstNode)
   return RemoveNode(firstNode, last)
```

Użycie :

```
list = RemoveFirst(list)
```

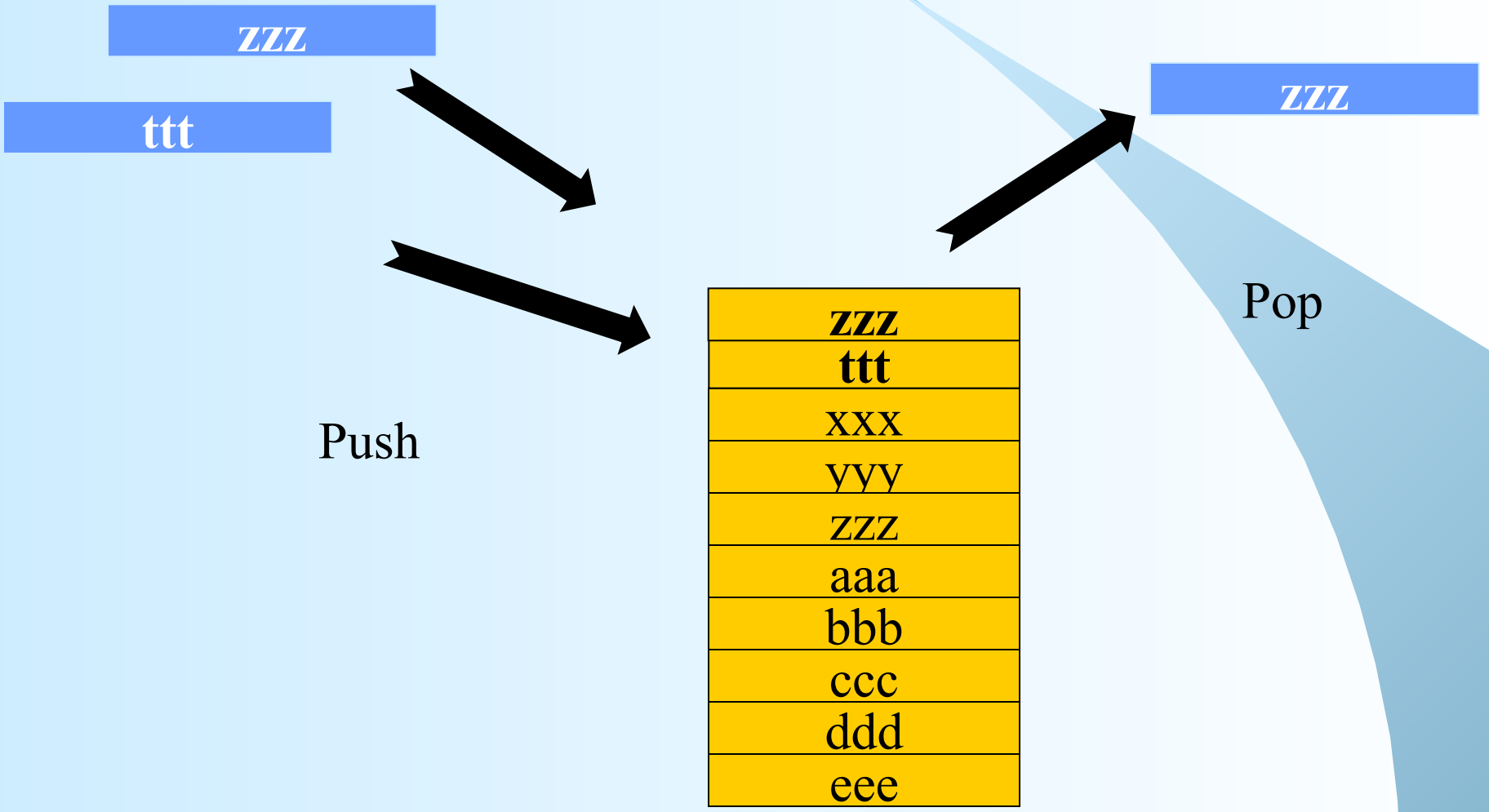
Lista - struktura o dostępie sekwencyjnym

Operacje :

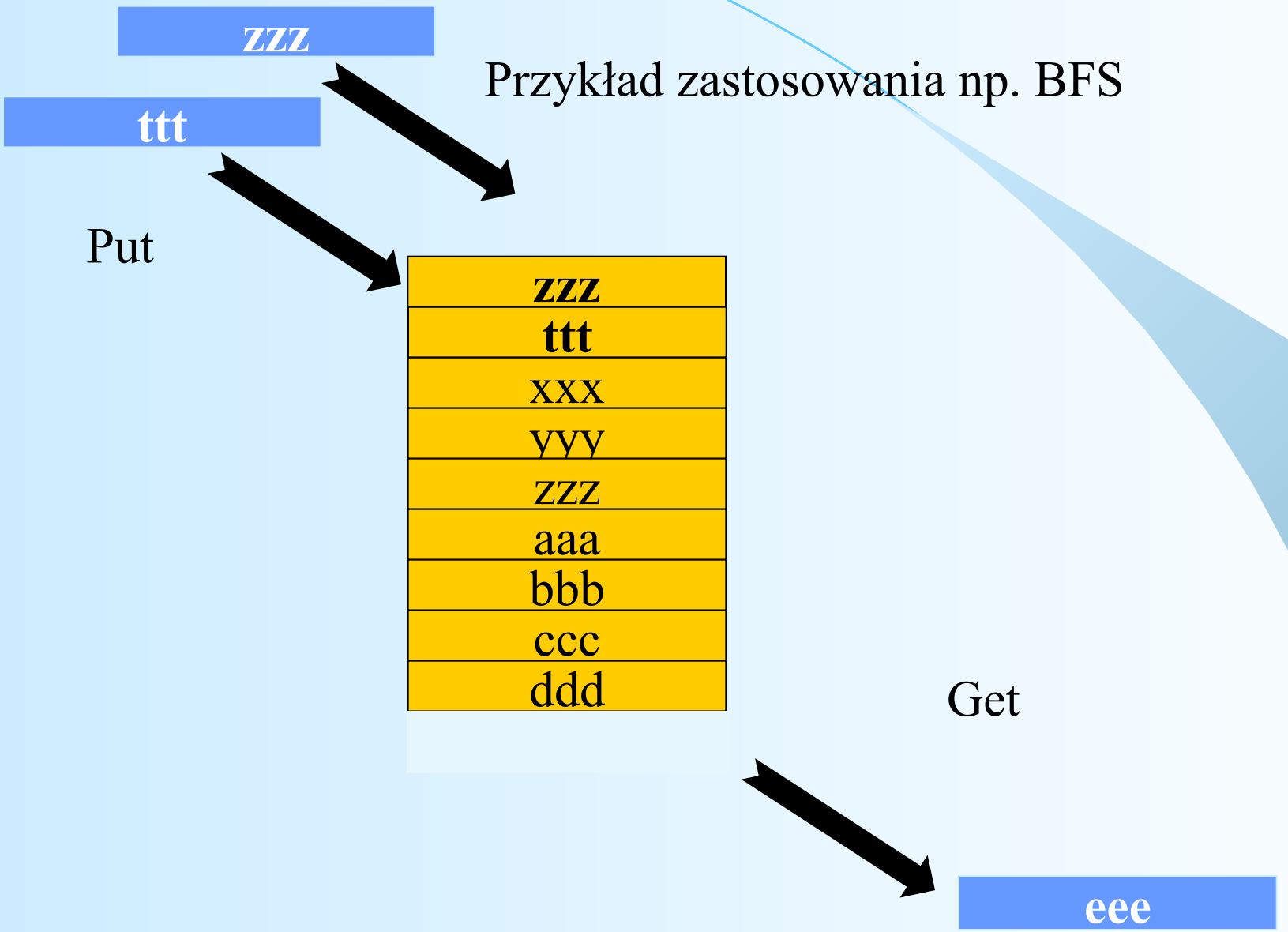
- ✓ Przeszukiwanie - O(n) (zależy ? od uporządkowania)
- ✓ Wstawianie na poczatek - Θ(1)
- ✓ Usuwanie z poczatku - Θ(1)
- ✓ Zapis na i-ta pozycję - O(n) : GetAt+InsertAfter
- ✓ Odczyt - O(n) : GetAt(list,i-1)
- ✓ Wstawianie na koniec Θ(n) ??
- ✓ Usuwanie z końca Θ(n) ??

Stos - Last In First Out

Przykład zastosowania np. DFS

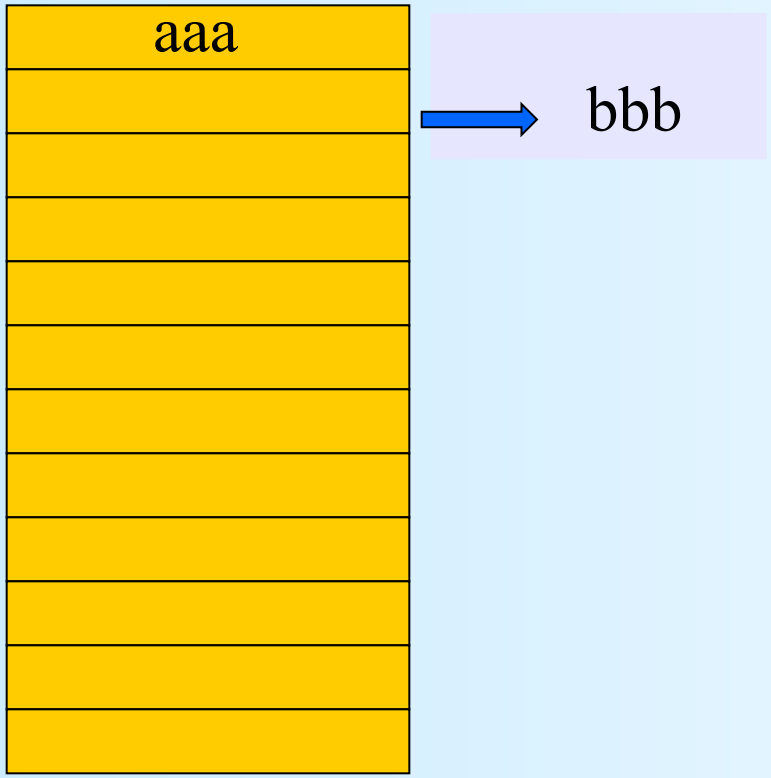


Kolejka - First In First Out



Tablicowa realizacja stosu

array[MAX];



cnt

1

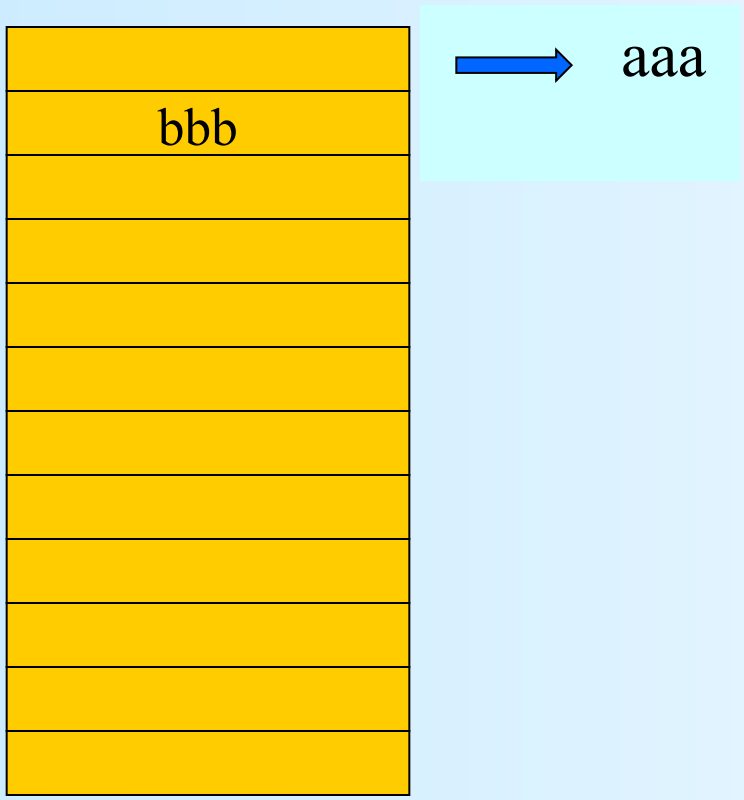
```
cnt=0
def Push(element) :
    if cnt < MAX
      array[cnt] = element
      cnt = cnt+1
    else :
      ERROR("przepelnienie")


def Pop()
    if cnt <= 0 :
       ERROR("stos pusty")
    else :
       cnt = cnt-1
       return array[cnt]
```

Problem :Ograniczona pojemność

Tablicowa realizacja kolejki

ELEMENT array[MAX];



Problem :

- ograniczona pojemność
- wędrujące elementy

Realizacja

- realokacji elementów po osiągnięciu końca buforu
- śledzenie początku i końca (skomplikowane) – tzw. bufor cykliczny

Listowa realizacja stosu/kolejki

- ✓ Push -> AddFirst() => Θ(1)
- ✓ Put -> AddLast() => Θ(n) lub O(1)
- ✓ Pop/Get -> GetFirst() + RemoveFirst() => Θ(1)

Poniższy Push, Put same alokują nowy węzeł listy (tj. jako parametr otrzymują dane do umieszczenia na stosie/w kolejce

```
def Push(stack, dataToStore) :
    node = SLNODE()
    node.data = dataToStore
    return AddFirst(stack, node)

def Put(queue, dataToStore) :
    node = DLNODE()
    node.data = dataToStore
    return AddLast(queue, node)
```

Listowa realizacja kolejki/stosu

Poniższe Get, Pop zwracają dane a nie węzły listy

Z : węzeł nie wymaga dealokacji (w C/C++ trzeba dodać)

```
def Pop(stack) :
    ret = GetFirst(stack)
    if ret == None :
            ERROR("Struktura jest pusta")
    return RemoveFirst(stack), ret.data


def Get(queue) :
    ret = GetFirst(queue)
    if ret == None :
            ERROR("Kolejka jest pusta")
    return RemoveFirst(queue), ret.data
```

Użycie :

```
stack,value = Pop(stack)
```

Listowa realizacja kolejki w czasie stałym

```
def Put(firstNode, lastNode, dataToStore) :
    node = DLNODE()
    node.data = dataToStore
    node.next = None
    node.prev = lastNode
    if lastNode == None :   # kolejka byla pusta
       return node, node

    lastNode.next = node
    return firstNode, node
```

Użycie :

```
start,end = Put(start, end, "abc")
```

Listowa realizacja kolejki w czasie stałym

```
def Get(firstNode, lastNode) :
    if firstNode == None :
       ERROR("Kolejka jest pusta")
   elif firstNode == lastNode :
       return None, None, firstNode.data

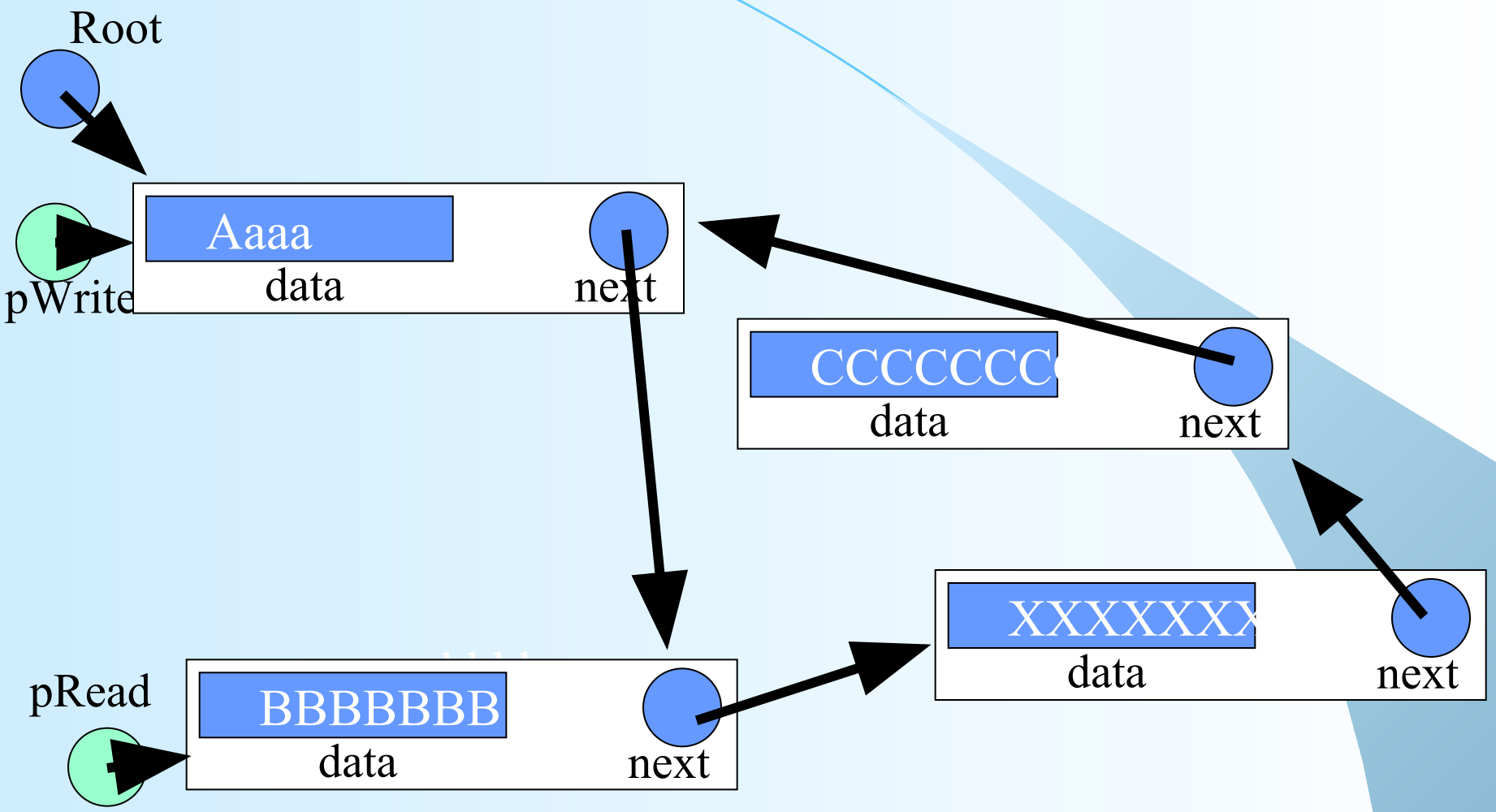
   if (firstNode.next == None) :
          return None, None, firstNode.data

   ret = firstNode.data
   return RemoveFirst(firstNode), lastNode, ret
```

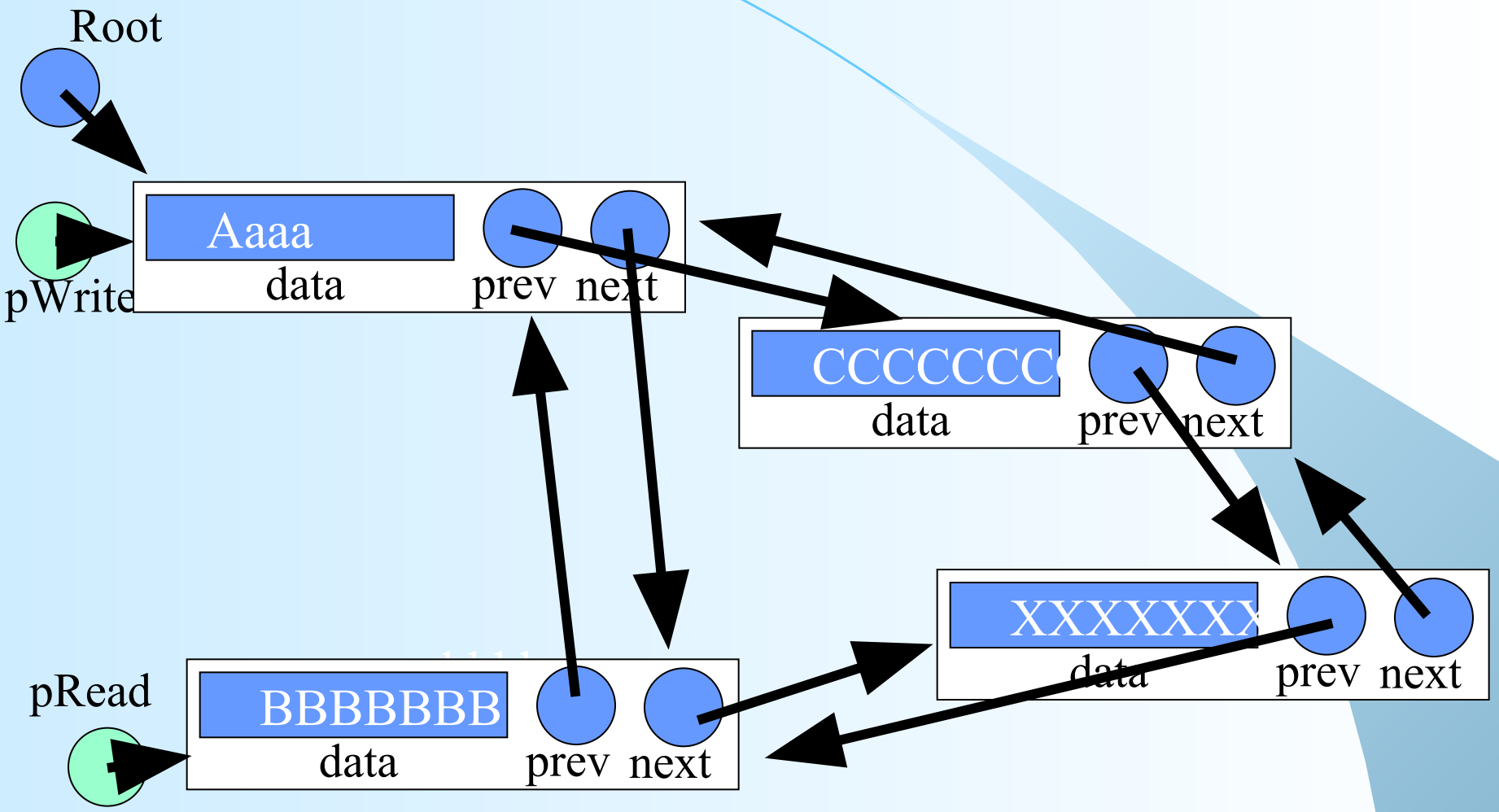
Użycie :

```
start,end,value = Get(start,end)
```

Lista cykliczna (jednostronna)



Lista cykliczna (dwustronna)



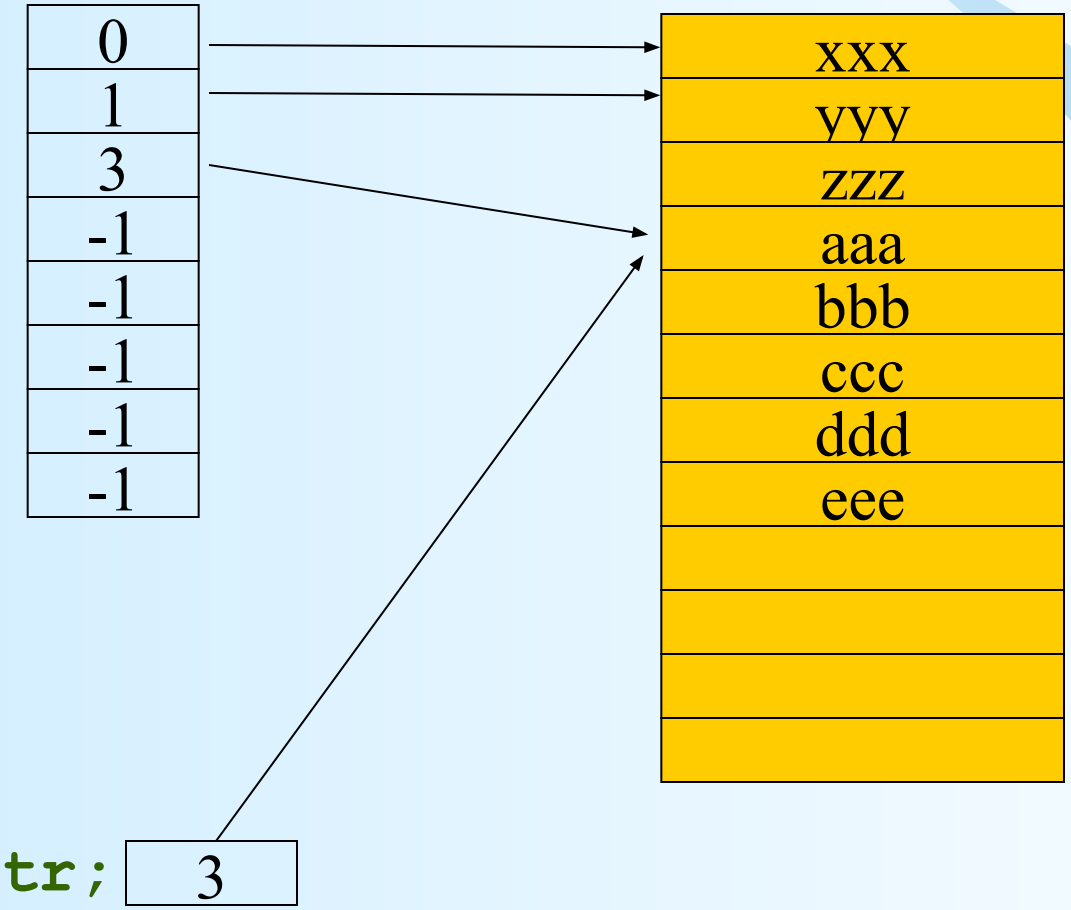
Lista cykliczna

- ✓ Ostatni element pokazuje na pierwszy
- ✓ W przypadku cyklicznej listy dwukierunkowej – również pierwszy na ostatni
- ✓ Wstaw, usuń są analogiczne do listy zwykłej
- ✓ Szukanie – należy zapamiętać początek poszukiwań aby wiedzieć kiedy je zakończyć

- ✓ Czas sklejania dwóch list ?

Struktury wskaźnikowe bez wskaźników

```
int arrOfPtrs[MAXPTR];   ELEMENT elements[MAXEL];
```



Przykład użycia stosu

- ✓ odwiedzanie pomieszczen – DFS

```
def VisitRooms(startRoom) :
    stack = []
    stack = Push(stack, startRoom)
    while stack != None :
        stack, current = Pop(stack)
        neighbours = get_neighbours(current)
        for room in neighbours :
            if not already_visited(room) :
                mark_visited(room)
                stack = Push (stack, room)
```

- ✓ odwiedzanie pomieszczen BFS :
  Push -> Put
  Pop -> Get

Odwrotna Notacja Polska

- ✓ Rozwinięcie notacji zaproponowanej przez Jana Łukasiewicza
- ✓ Pozwala wykonywać obliczenia bez nawiasów, bez potrzeby stosowania priorytetów
- ✓ a+b*c przedstawiane jako a b c * +
- ✓ a+b*c-7 przedstawiane jako a b c * + 7 -
- ✓ b*c-(d+e)*4 przedstawiane jako b c * d e + 4 * -

ONP nie jest jednoznaczna tj. wyrażenie może mieć kilka legalnych reprezentacji

ONP - obliczenia

```
def ONPCalc(onp) :
    '''Funkcja oblicza wyniki wyrazenia w ONP
   W implementacji obslugiwane są operatory 2
   argumentowe. Przyklad : 3 4 * 2 +'''

   stack = None
   while onp != None :
      onp,token = get_next_token(onp)
      if is_operand(token) :
         stack = Push(stack, token)
      else :
         stack, op1 = Pop(stack)
         stack, op2 = Pop(stack)
         result = do_calculation(token,op1,op2)
         stack = Push(stack, result)
   stack, result = Pop(stack)
   return result
```

ONP - obliczenia

Obliczenia :

- **weź element wyrażenia**
- **jeżeli to operand połóż go na stosie**
- **jeżeli to operator**
  - **zdejmij ze stosu odpowiednią liczbę operandów,**
  - **wykonaj obliczenia**
  - **odłóż wynik na stos**
- **wynik znajduje się na górze stosu**

ONP – przykład obliczeń

Wejście	Bieżący	Stos po b. operacji
2, 3, *, 5, +, 7, *		
3, *, 5, +, 7, *	2	2
*, 5, +, 7, *	3	2,3
5, +, 7, *	*	6
+, 7, *	5	6, 5
7, *	+	11
*	7	11, 7
	*	77

ONP - konwersja

Konwersja :

- **Pobierz kolejny element wyrażenia**
- **Jeśli jest to operand przepisz go na wyjście**
- **Jeśli jest to operator "(" połóż go na stosie**
- **Jeśli jest to operator ")" zdejmij ze stosu wszystkie operatory aż do nawiasu otwierającego włącznie**
- **Jeśli to inny niż nawiasy**
  - **zdejmij ze stosu wszystkie operatory o priorytecie wyższym (ważniejsze) lub równym do bieżącego i**
  - **połóż bieżący operator na stosie**
- **Po przetworzeniu wyrażenia przepisz stos na wyjście**

ONP - konwersja

```
def ONPConv(expression) :
    stack = None
    onp = None
    while expression != None :
        expression, token = get_next_token(expression)
        if is_operand(token):
            onp = Put(onp, token)
        elif token == "(" :
            stack = Push(stack, token)
        elif token == ")" :
            while stack != None :
                stack,token = Pop(stack)
                if token == "(" : break
                else : onp = Put(onp, token)
        else :
            priority = get_priority(token)
            while stack != None :
                stack,top = Pop(stack)
                if top=="(" or get_priority(top)<priority :
                    stack = Push(stack,top)
                    break
                onp = Put(onp, top)
            stack = Push(stack,token)
    while stack != None :
        stack, item = Pop(stack)
        onp = Put(onp, item)
    return onp
```

ONP – przykład konwersji

Wejście	Wyjście	Stos
b*c-(d+e)*4		
*c-(d+e)*4	b	
c-(d+e)*4	b	*
-(d+e)*4	b c	*
(d+e)*4	b c *	-
d+e)*4	b c *	- (
+e)*4	b c * d	- (
)*4	b c * d	- (+
* 4	b c * d e	- (+
4	b c * d e +	- *
	b c * d e + 4	- *
	b c * d e + 4 *	-
	b c * d e + 4 * -	