Wykład 13: Struktury danych.

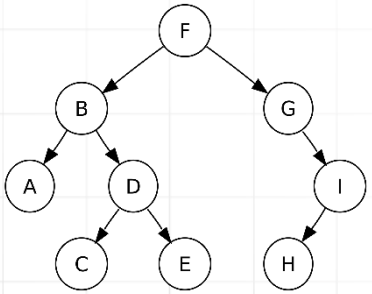
dr inż. Andrzej Stafiniak

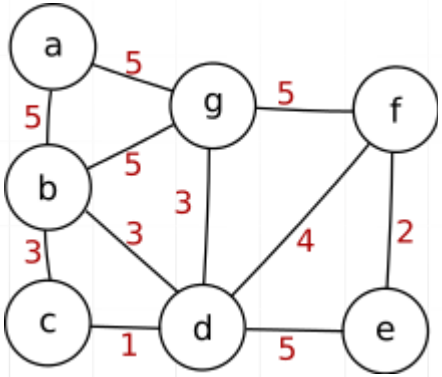
Wrocław 2023

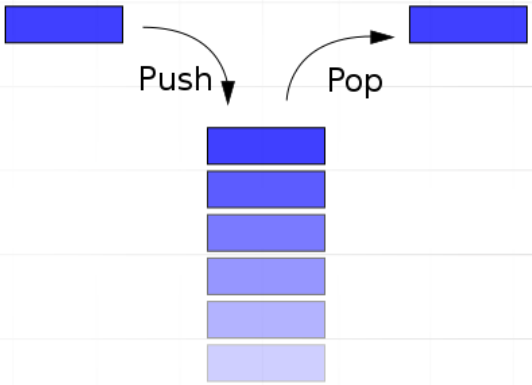
Jeszcze bardziej złożone struktury danych

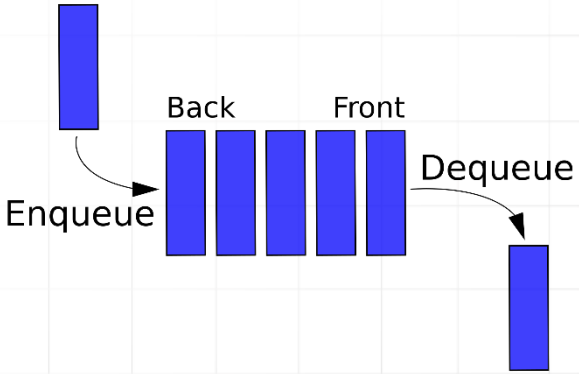
Na podstawie prostych typów oraz omówionych złożonych typów danych można budować bardziej złożone formy takie jak:

- Lista jedno – dwukierunkowa,
- Bufor LIFO (Stos),
- Bufor FIFO (Kolejka),
- Graf,
- Drzewo,







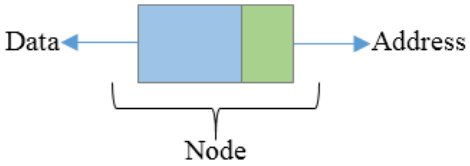


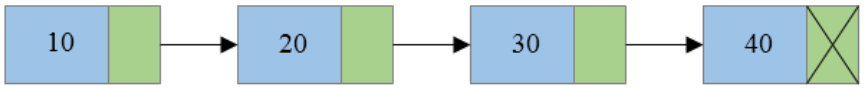
wikipedia.org

Lista jako struktura danych

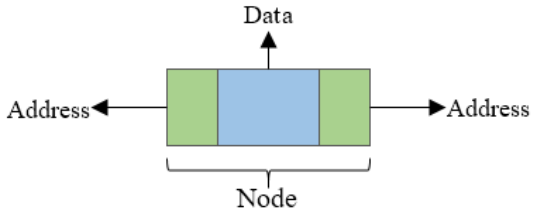
Lista to przykład **struktury danych**, której elementy ułożone są w porządku liniowym, ale w odróżnieniu od **tablicy** nie muszą zajmować **ciągłego obszaru pamięci**.

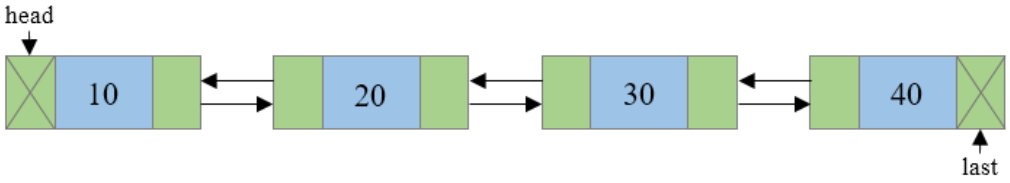
- Lista jednokierunkowa (*single linked list*)





- Lista dwukierunkowa (*doubly linked list*)

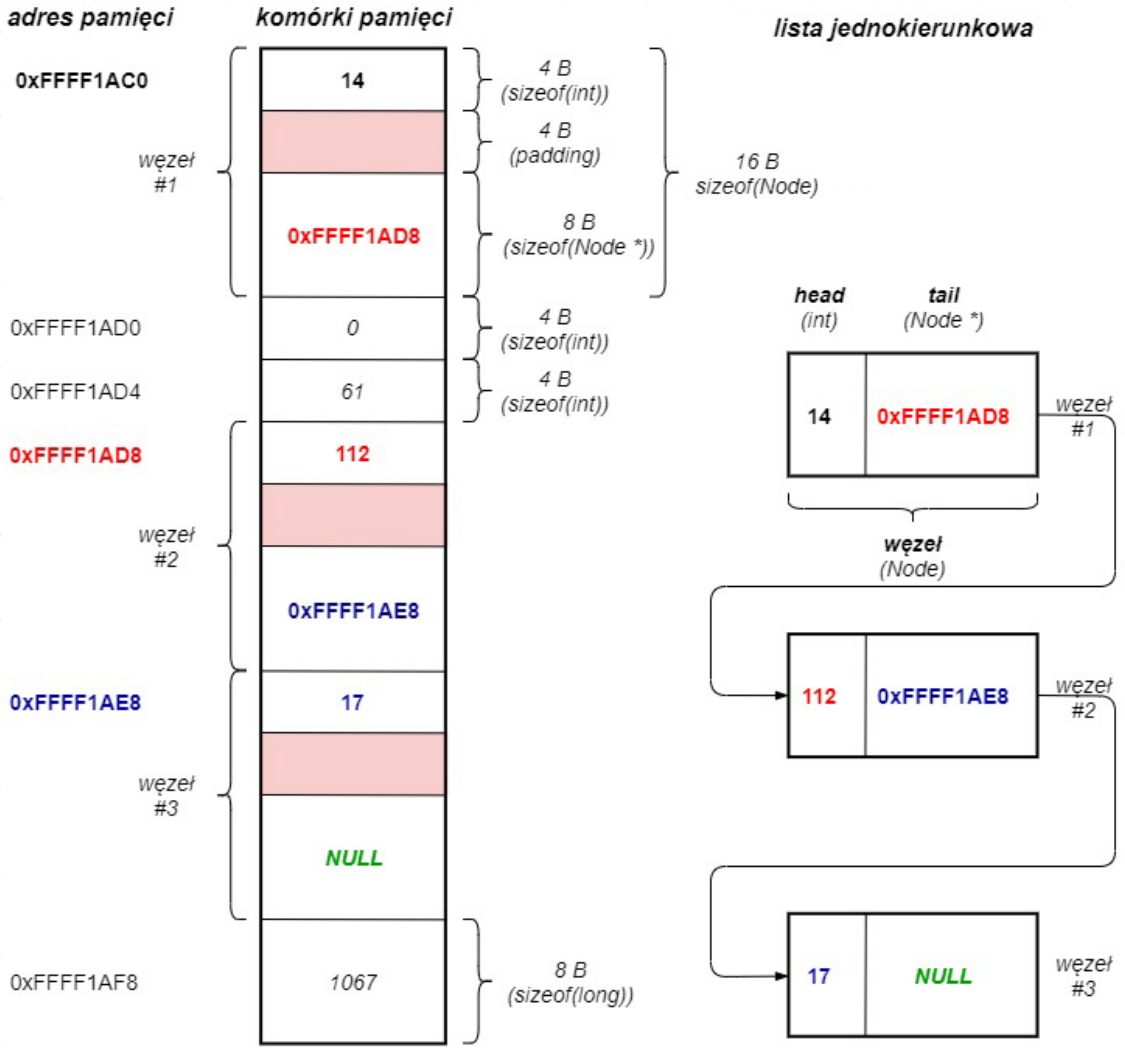




Element listy, zwany **węzłem** (node), składa się z danych które gromadzi lista oraz ze **wskaźnika** do następnego elementu listy. Wskaźnik ostatniego węzła wskazuje na zerowy adres NULL.

https://codeforwin.org/2015/10/doubly-linked-list-data-structure-in-c.html

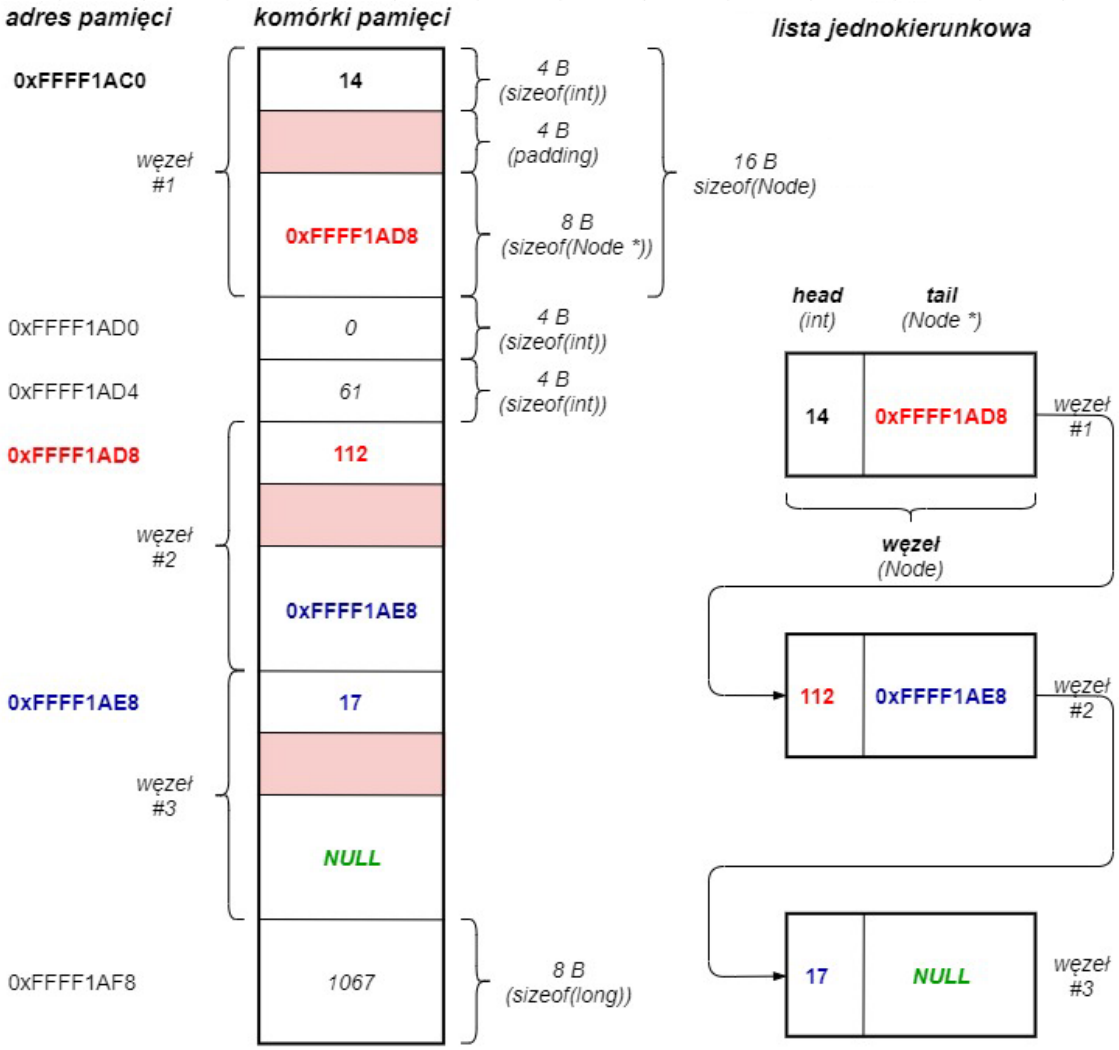
Lista jako struktura danych



Dzięki takiej implementacji możliwe jest przechodzenie po węzłach listy podobnie, jak podczas iteracji po elementach tablicy, ale różnica jest taka, że poszczególne węzły listy **nie muszą zajmować ciągłego obszaru pamięci**.

Lista różni się ponadto od tablicy tym, że możemy ją **modyfikować tzn. dodawać lub usuwać elementy w dowolnym miejscu**. Realizowane jest to za pomocą zmiany wartości wskaźnika wskazującego następny element listy (oraz poprzedni element listy w listach dwukierunkowych).

Lista jako struktura danych



Dostęp do kolejnych węzłów listy jest sekwencyjny, oznacza to, że z jednego elementu listy możemy skoczyć do kolejnego: następnego lub poprzedniego.

Dojście do konkretnego elementu listy wymaga przejścia sekwencyjnego przez wszystkie węzły od pierwszego do docelowego.

Wynika to z samej budowy listy.

Lista jako struktura danych

```
typedef struct Student {
    int index;
    char name[16];
    struct Student * next;
    struct Student * prev;
}StudentPwr;
```

node:

head

tail

W liście, **węzeł** (**node**) może być pojedynczym egzemplarzem struktury, która składa się:

- z **głowy – head**, czyli danych, które ma przechowywać lista
- oraz z **ogona – tail**, czyli pola, które wskazuje kolejny węzeł listy za pomocą wskaźników:
  - **`next`** – wskazuj następny węzeł list,
  - **`prev`** – wskazuje poprzedni węzeł listy.

Dla listy jednokierunkowej węzeł zawiera tylko jeden wskaźnik `next`, dla listy dwukierunkowej dwa `next` i `prev`.

Lista jako struktura danych

```
typedef struct Student {
    int index;
    char name[16];
    struct Student * next;
    struct Student * prev;
}StudentPwr;
```

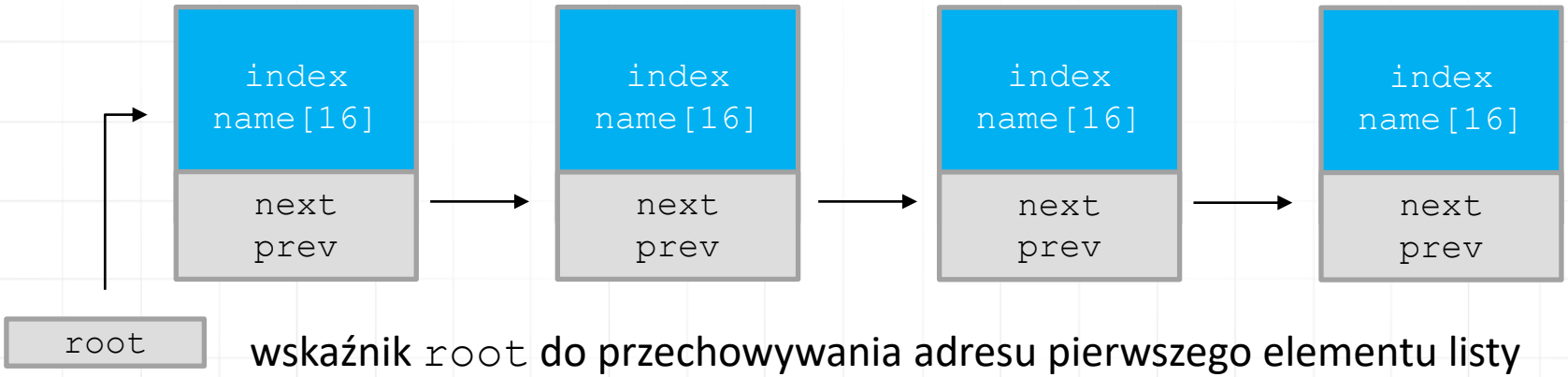
node:

head

tail

Adresy zapisane w wskaźnikach pozwalają powiązać listę w jedną całość (ciąg), ale w zależności od implementacji możemy rozróżnić kilka rodzajów list:

- lista jednokierunkowa – wykorzystywany tylko jeden wskaźnik `next`



wskaźnik `root` do przechowywania adresu pierwszego elementu listy

Lista jako struktura danych

```
typedef struct Student {
    int index;
    char name[16];
    struct Student * next;
    struct Student * prev;
}StudentPwr;
```

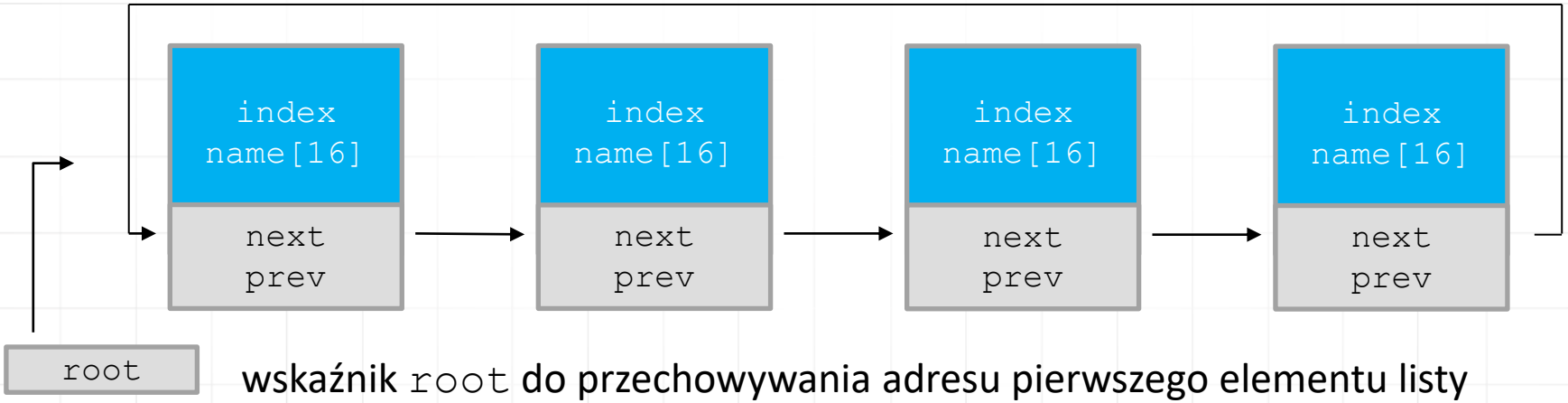
node:

head

tail

Adresy zapisane w wskaźnikach pozwalają powiązać listę w jedną całość (ciąg), ale w zależności od implementacji możemy rozróżnić kilka rodzajów list:

- lista jednokierunkowa cykliczna



wskaźnik `root` do przechowywania adresu pierwszego elementu listy

Lista jako struktura danych

```
typedef struct Student {
    int index;
    char name[16];
    struct Student * next;
    struct Student * prev;
}StudentPwr;
```

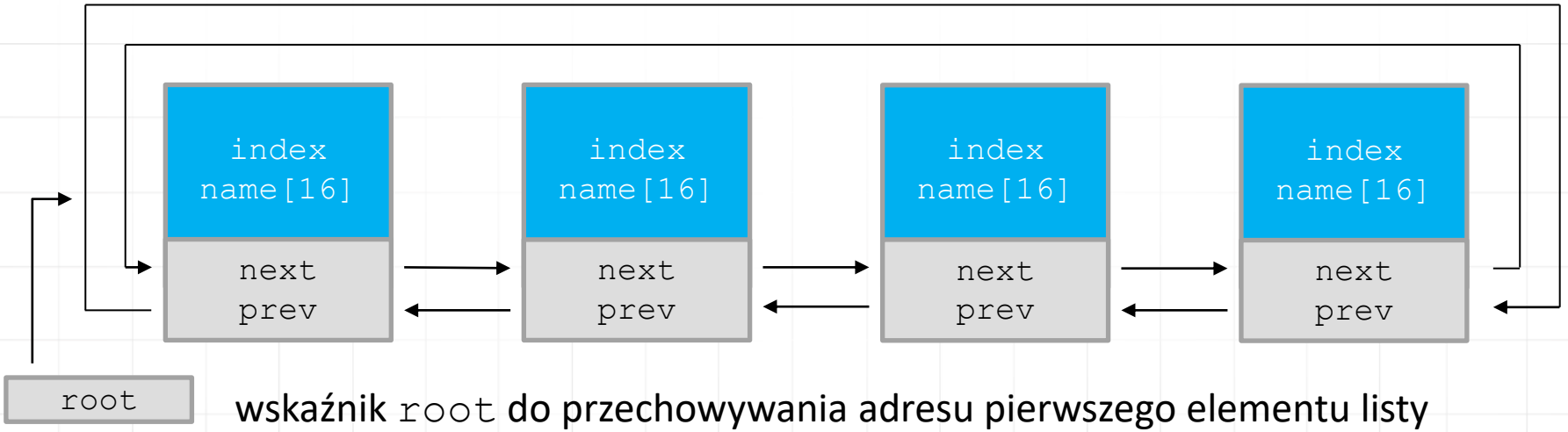
node:

head

tail

Adresy zapisane w wskaźnikach pozwalają powiązać listę w jedną całość (ciąg), ale w zależności od implementacji możemy rozróżnić kilka rodzajów list:

- lista dwukierunkowa cykliczna



wskaźnik `root` do przechowywania adresu pierwszego elementu listy

Lista jako struktura danych

```
typedef struct Student {
    int index;
    char name[16];
    struct Student * next;
    struct Student * prev;
}StudentPwr;
```

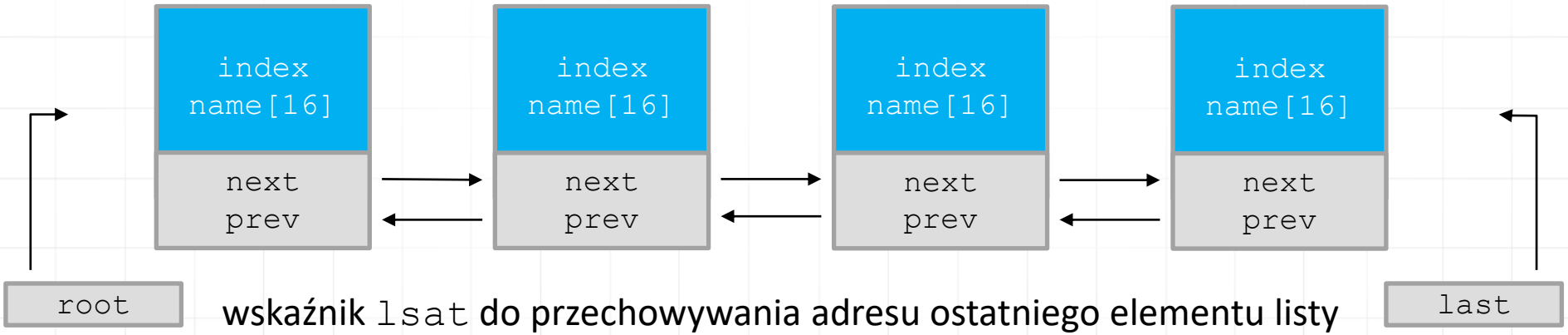
node:

head

tail

Adresy zapisane w wskaźnikach pozwalają powiązać listę w jedną całość (ciąg), ale w zależności od implementacji możemy rozróżnić kilka rodzajów list:

- lista dwukierunkowa



wskaźnik `lsat` do przechowywania adresu ostatniego elementu listy

Lista jako struktura danych

```
typedef struct Student {
    int index;
    char name[16];
    struct Student * next;
    struct Student * prev;
}StudentPwr;
```

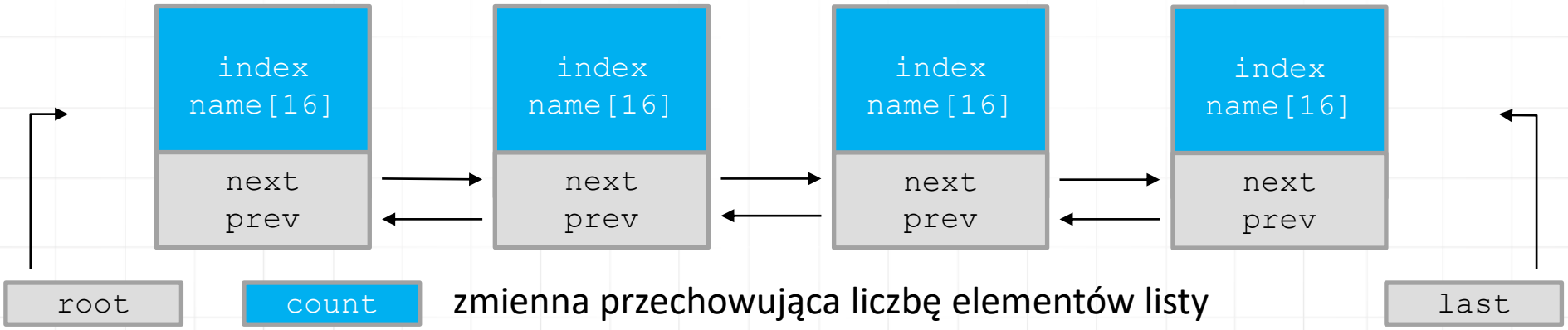
node:

head

tail

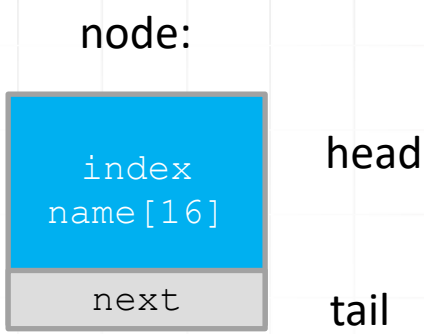
Adresy zapisane w wskaźnikach pozwalają powiązać listę w jedną całość (ciąg), ale w zależności od implementacji możemy rozróżnić kilka rodzajów list:

- lista dwukierunkowa



Lista jednokierunkowa

```
typedef struct Student {
    int index;
    char name[16];
    struct Student * next;
}StudentPwr;
```



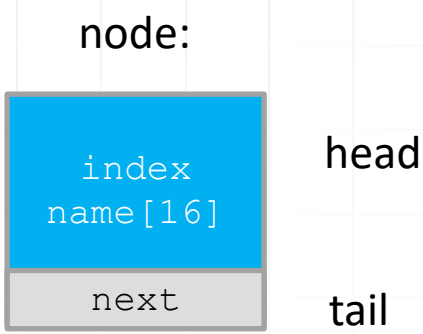
- lista dwukierunkowa

Struktura danych typu lista, najczęściej wykorzystywana jest do tworzenia baz danych, które swobodnie można modyfikować w czasie działania programu, dlatego aby tworzyć nowe węzły lub usuwać węzły, musimy dynamicznie przedzielać lub zwalniać pamięć.

```
StudentPwr * node = (StudentPwr *) malloc(sizeof(StudentPwr));
node->index = 123456;
node->name = "Kowalski";  // inicjalizacja wezla
node->next = NULL;
```

Lista jednokierunkowa

```
typedef struct Student {
   int index;
   char name[16];
   struct Student * next;
}StudentPwr;
```

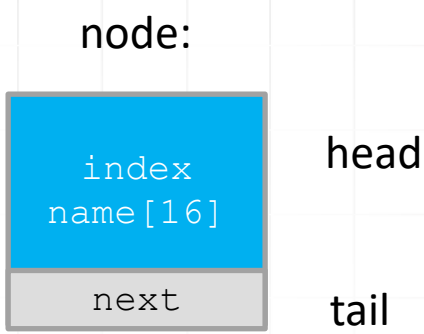


Aby uniwersalnie tworzyć nowe węzły, oddelegujmy to zadanie funkcji:

```
StudentPwr * createNode(int index , char * name, StudentPwr * next){
        //Przydziel dynamicznie adres na wezel
        StudentPwr * node = (StudentPwr *) malloc(sizeof(StudentPwr));

        node->index = index;                    // inicjalizacja wezla
        strncpy(node->name, name, 16);
        node->next = next;

        return node;                    // zwroc adres wezla
}
```

Lista jednokierunkowa

```
typedef struct Student {
   int index;
   char name[16];
   struct Student * next;
}StudentPwr;
```



Następnie stwórzmy pierwszy element listy:

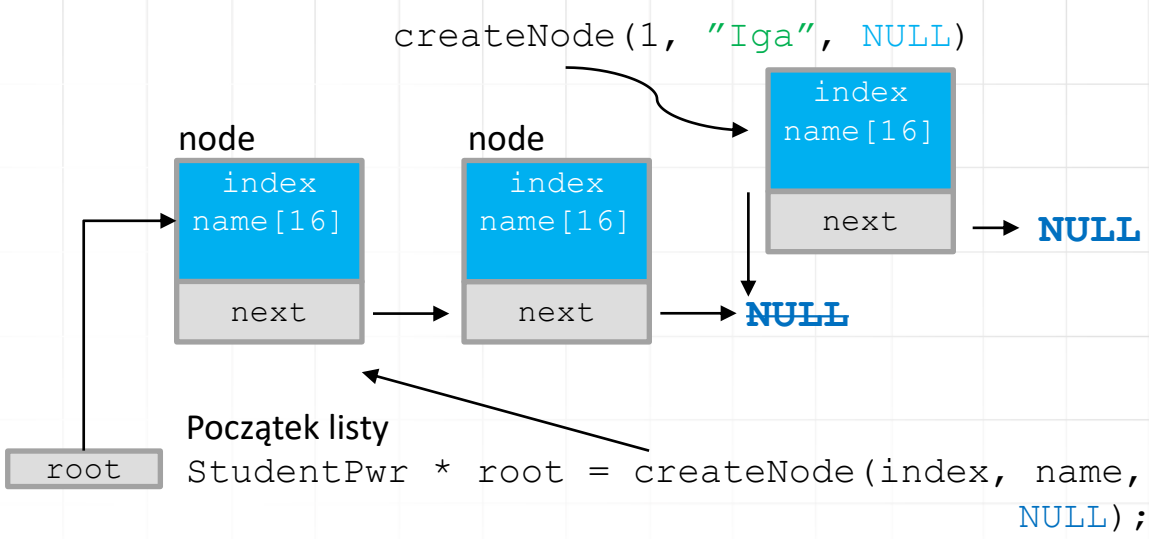
```
StudentPwr * root = createNode(266666, "Kowalski", NULL)
```

Wskaźnik `root` przechowuje adres początku listy.

Ponieważ jest to pierwszy element listy, jest zarazem ostatnim elementem listy, czyli nie ma na kogo wskazywać.

Lista jednokierunkowa

```
typedef struct Student {
    int index;
    char name[16];
    struct Student * next;
}StudentPwr;
```



Nowe węzły należy gdzieś dodać, np. na początku lub na końcu listy:

Jako `root` przekazujemy adres na początek listy.

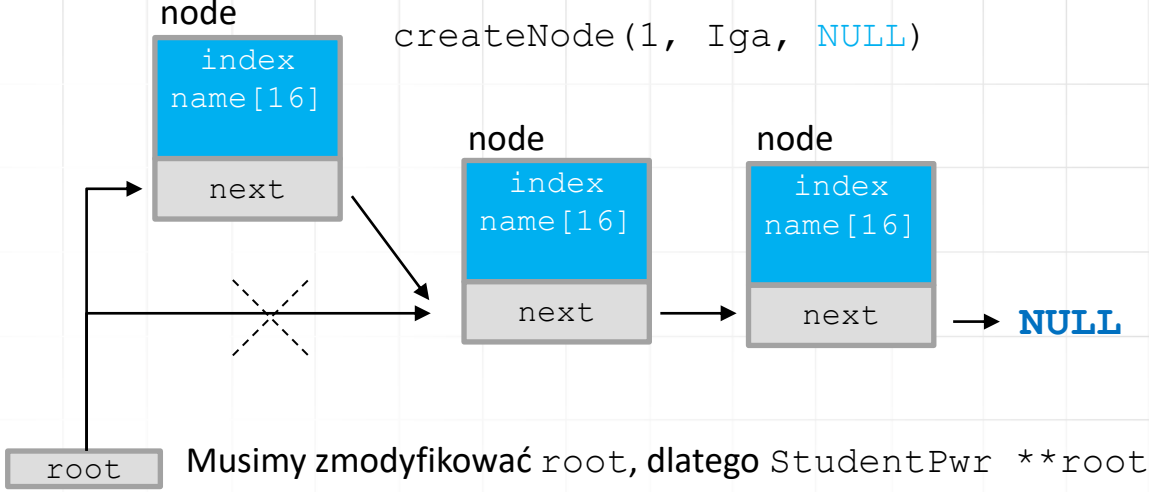
```
void pushBack (StudentPwr * root , int index , char * name) {

    StaudentPwr * currentNode = root;
    if (currentNode != NULL){
        while ( currentNode->next != NULL )
            currentNode = currentNode->next; // Przewin na koniec listy
        currentNode->next = createNode(index, name, NULL);
    }
}
```

Pomocnicza zmienna wskaźnikowa, która pozwoli nam na przewijanie listy.

Dodaj nowy element na koniec listy i ustawa wskaźnik `next` na `NULL`.

Lista jednokierunkowa

```
typedef struct Student {
    int index;
    char name[16];
    struct Student * next;
}StudentPwr;
```



Musimy zmodyfikować `root`, dlatego `StudentPwr **root`

`*root = node;`

Nowe węzły należy gdzieś dodać, np. na początku lub na końcu listy:

Przekazujemy adres `root` czyli adres czegoś, co wskazuje początek listy, aby móc go zmodyfikować.

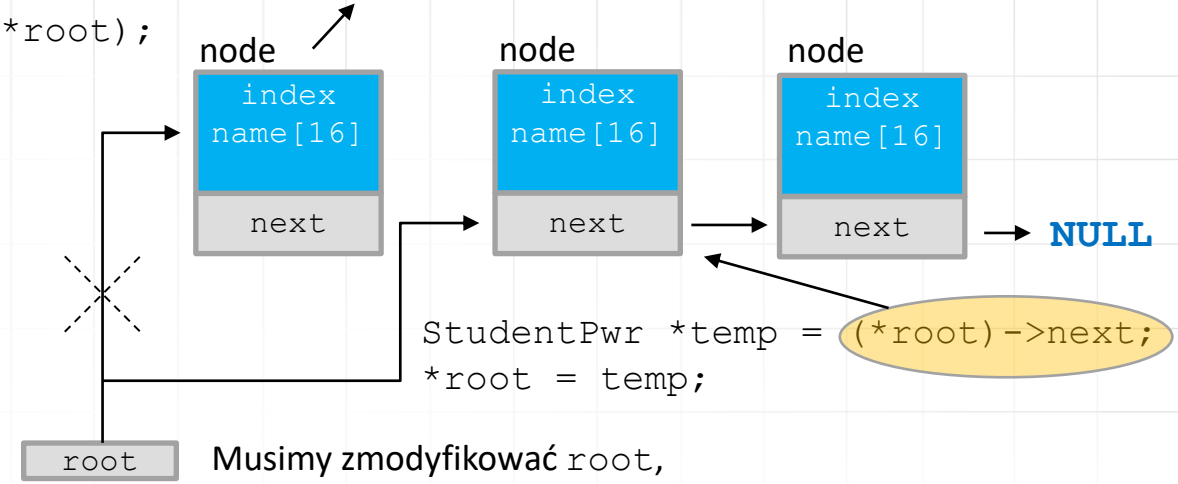
```
void pushFront (StudentPwr ** root , int index , char * name) {

    if ( root != NULL ) {
        StudentPwr * node = createNode(index, name, *root);
        *root = node;
    }
}
```

Zmodyfikuj i ustaw nowy adres listy.

Dodaj nowy element na początek listy i ustawa wskaźnik `next` na adres początku listy przekazanej do funkcji .

Lista jednokierunkowa

```
free (*root);
```




```
typedef struct Student {
    int index;
    char name[16];
    struct Student * next;
}StudentPwr;
```

Musimy zmodyfikować `root`, dlatego `StudentPwr **root`

Możemy również usunąć węzeł, np. pierwszy:

Przekazujemy adres `root` czyli adres czegoś, co wskazuje początek listy, aby móc go zmodyfikować.

```
void popFront ( StudentPwr ** root ) {

    if(root != NULL && *root != NULL){
        StudentPwr * temp = (*root)->next;
        free (*root);
        *root = temp; }
}
```

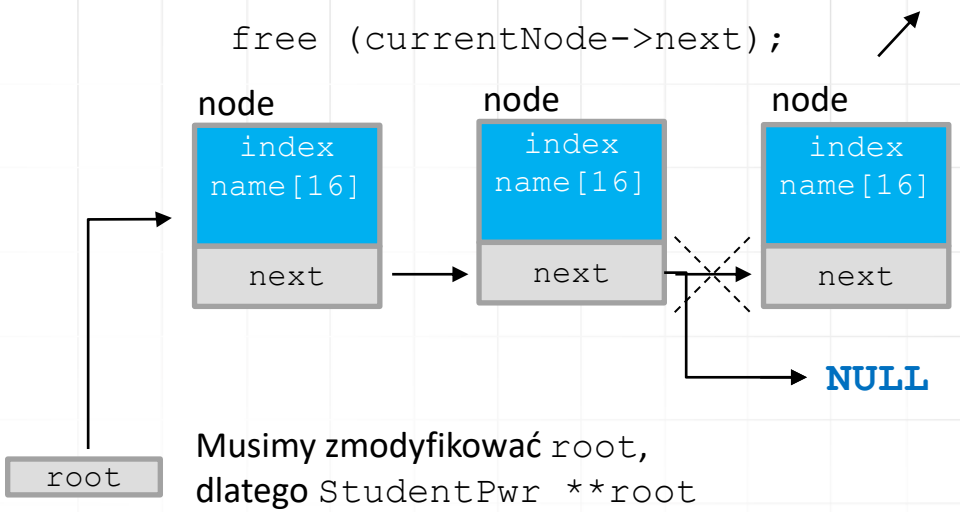
Tymczasowo zapamiętaj adres drugiego elementu w zmiennej wskaźnikowej `temp`.

Zwolnij to co wskazuje `root`.

Pod wskaźnik wskazujący na pierwszy element przypisz adres drugiego elementu

Lista jednokierunkowa

```
typedef struct Student {
   int index;
   char name[16];
   struct Student * next;
}StudentPwr;
```



Musimy zmodyfikować `root`, dlatego `StudentPwr **root`

Możemy również usunąć węzeł, np. ostatni:

```
void popBack( StudentPwr ** root ) {
   if(root != NULL && *root != NULL){
      StudentPwr * currentNode = *root;
      if(currentNode->next == NULL) {
         free(currentNode);
         *root = NULL ;}
      else{
         while(currentNode->next->next != NULL)
            currentNode = currentNode -> next;
         free (currentNode->next);
         currentNode->next =NULL ;
      }
   }
}
```

Przekazujemy adres `root` czyli adres czegoś, co wskazuje początek listy, aby móc go zmodyfikować.

Pomocnicza zmienna wskaźnikowa, która pozwoli nam na przewijanie listy .

W sytuacji gdy lista ma jeden element.

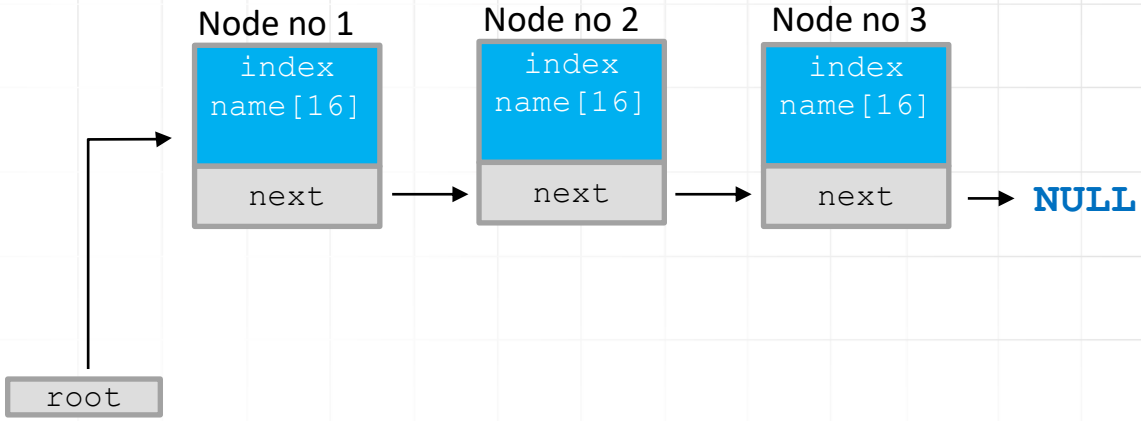
Przewiń do przedostatniego elementu.

Zwolnij adres ostatniego elementu.

Ustaw przedostatni element ostatnim.

Lista jednokierunkowa

```
typedef struct Student {
   int index;
   char name[16];
   struct Student * next;
}StudentPwr;
```

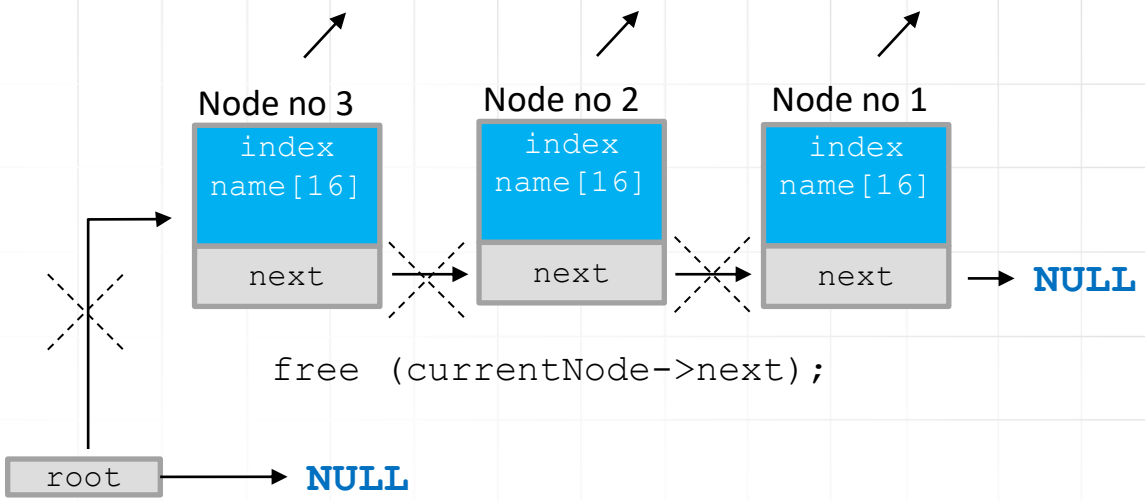


Możemy zdefiniować funkcję w celu wyświetlenia danych z listy:

```
void showList(StudentPwr * root) {
   int i = 1;
   StudentPwr * currentNode = root;
   if ( currentNode != NULL ) {
      // Przejdz na koniec listy
      while (currentNode->next != NULL){
         printf("Node no %i, Student: %s, index = %i \n", i, currentNode->name,
         currentNode->index);
         currentNode = currentNode -> next ;
         i++;
      }
      printf("Node no %i, Student: %s, index = %i \n", i, currentNode->name,
      currentNode->index);   }
   else
      printf("No list to show");
}
```

Lista jednokierunkowa

```
typedef struct Student {
    int index;
    char name[16];
    struct Student * next;
}StudentPwr;
```

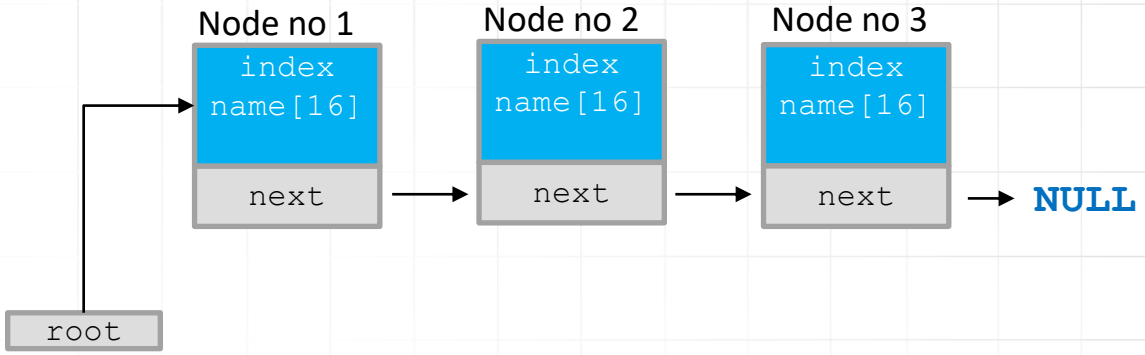


Na koniec wypada zdefiniować funkcję do zwolnienia pamięci po liście:

```
void deleteList(StudentPwr ** root) {
    int i = 1;
    while (*root != NULL){
        printf("Delete node no %i\n", i);
        popBack(root);
        i++;
    }
}
```

Lista jednokierunkowa

```
typedef struct Student {
   int index;
   char name[16];
   struct Student * next;
}StudentPwr;
```



Przykładowe nadgłówki funkcji do obsługi listy jednokierunkowej:

```
StudentPwr * createNode(int index , char * name, StudentPwr * next)

void pushBack(StudentPwr * root, int index, char * name)
void pushFront(StudentPwr ** root, int index, char * name);

void popFront(StudentPwr ** root);
void popBack(StudentPr ** root);

void showList(StudentPwr * root);
void deleteList(StudentPwr ** root);

bool isEmpty(StudentPwr * root);
```

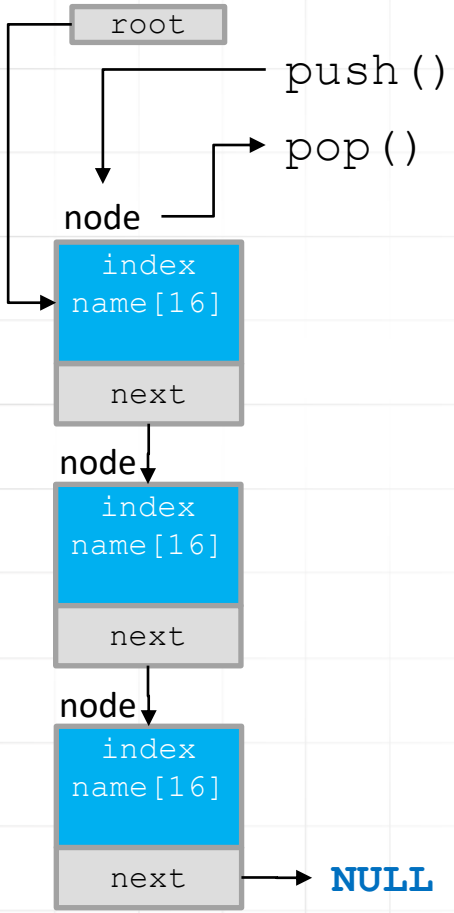
Przy usuwaniu elementu z listy jako argumenty można dodać wskaźniki na konkretny typ danych aby uzyskać wartości przechowywane w usuwanym węźle .

Warto ustawić typ zwracany przez funkcję na `bool` w celu zabezpieczenia przed nieprawidłowym wykonaniu instrukcji w funkcji. Można ustalić aby zwrócona została jakaś wartość z węzła lub adres węzła, lub np. ilość węzłów.

Lista jednokierunkowa - bufor LIFO

Na bazie listy jednokierunkowej można stworzyć:

- bufor LIFO (Stos)



Operacje na stosie mogą być realizowane na podstawie znanych już funkcji:

- `push()` - zapis na stos (np. `pushFront()`)
- `pop()` - usunięcie ze stosu (np. `popFront()`)
- `empty()` - sprawdzenie czy stos jest pusty (np.

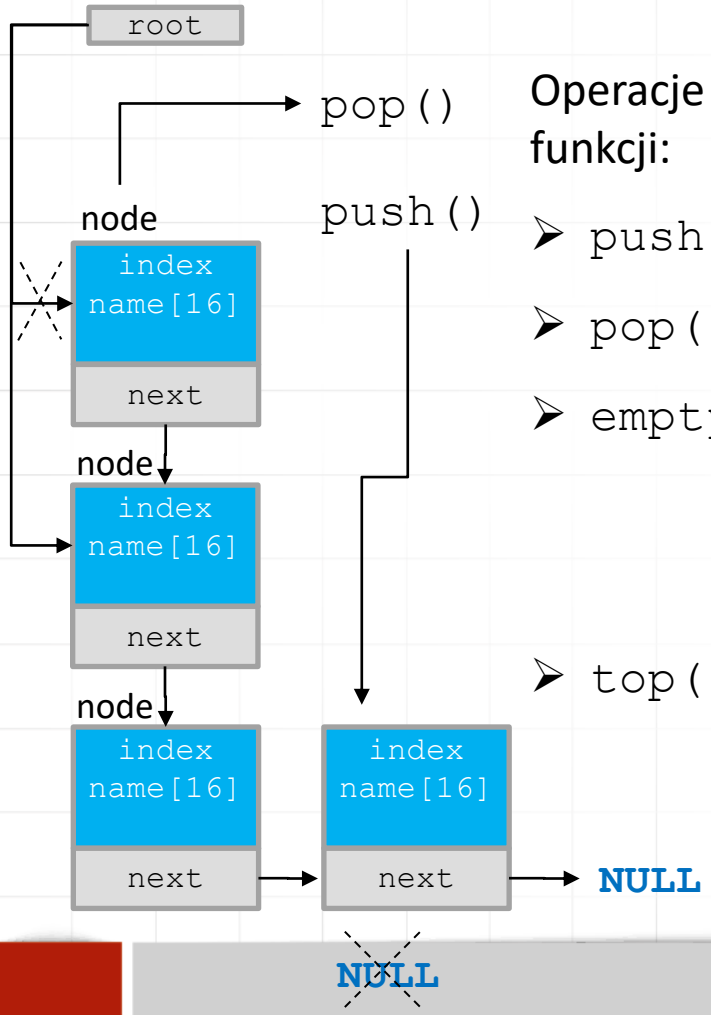
```
isEmpty(StudentPwr * root){
        return !root;
})
```

- `top()` - odczyt ze szczytu stosu, np. zwrócenie wskaźnika na szczyt stosu.

Lista jednokierunkowa - bufor FIFO

Na bazie listy jednokierunkowej można stworzyć:

- bufor FIFO (Kolejka)



Operacje na kolejce mogą być realizowane na podstawie znanych już funkcji:

- `push()` - zapis elementu na koniec kolejki (np. `pushBack()`)
- `pop()` - usunięcie elementu z początku kolejki (np. `popFront()`)
- `empty()` - sprawdzenie czy kolejka jest pusta (np.

```
isEmpty(StudentPwr * root){
        return !root;
})
```

- `top()` - odczyt z początku kolejki, np. zwrócenie wskaźnika na początek kolejki lub wartości z początku kolejki.