

Wykład 13: Struktury danych.

dr inż. Andrzej Stafiniak

Wrocław 2024



HR EXCELLENCE IN RESEARCH

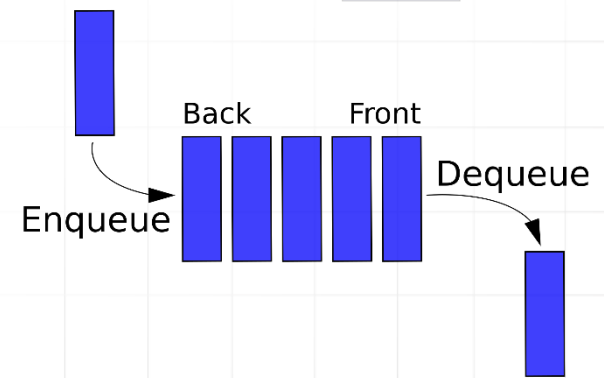
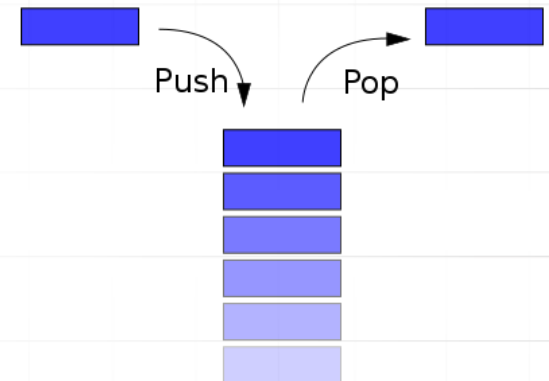
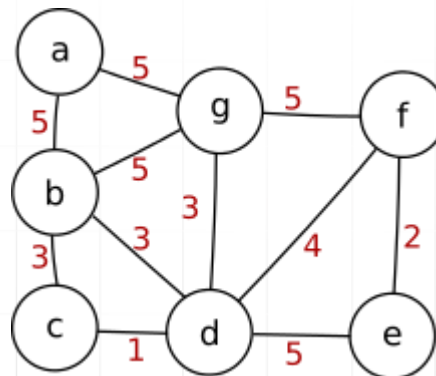
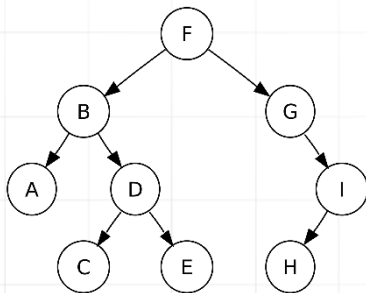


Politechnika Wroclawska

Jeszcze bardziej złożone struktury danych

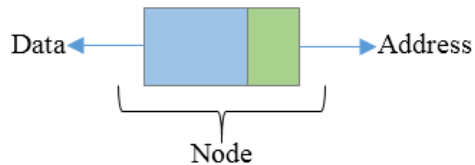
Na podstawie prostych typów oraz omówionych złożonych typów danych można budować bardziej złożone formy (Abstract Data Types) takie jak:

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



Lista jako struktura danych

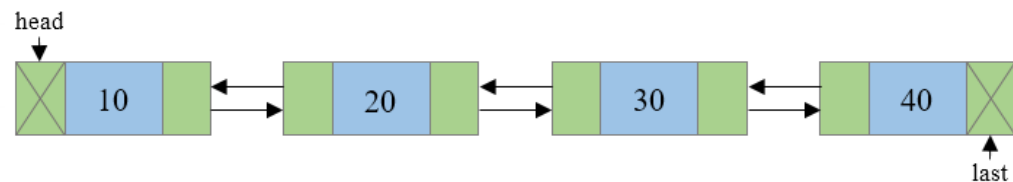
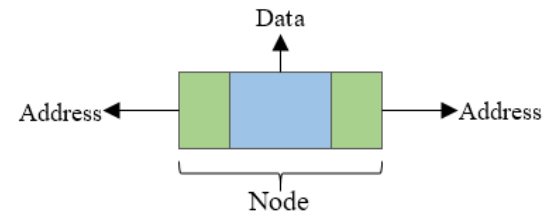
- Lista jednokierunkowa (*single linked list*)



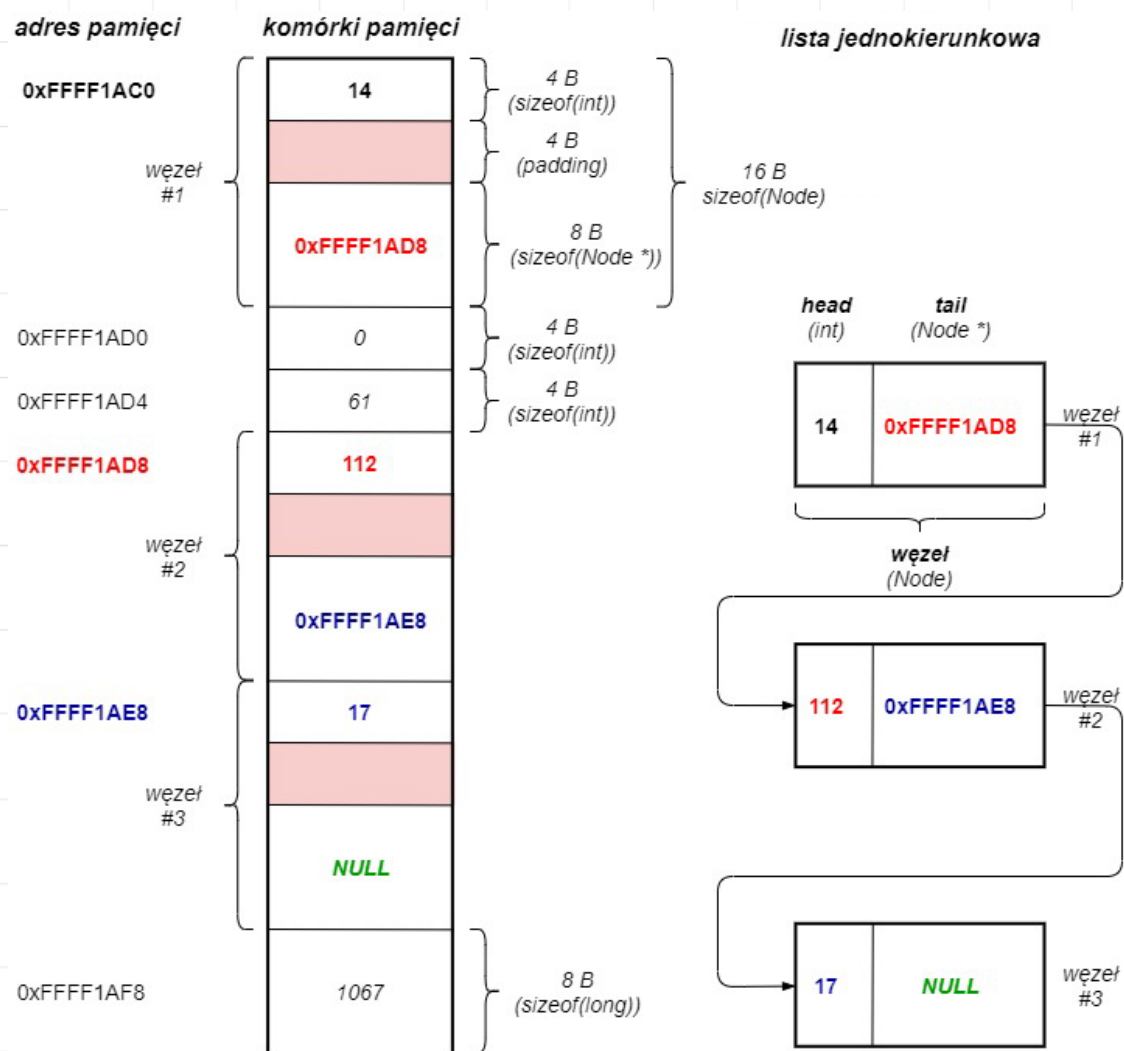
Element listy, zwany **węzłem** (node), składa się z danych które gromadzi lista oraz ze **wskaźnika** do następnego elementu listy. Wskaźnik ostatniego węzła wskazuje na zerowy adres NULL.

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

- Lista dwukierunkowa (*doubly linked list*)



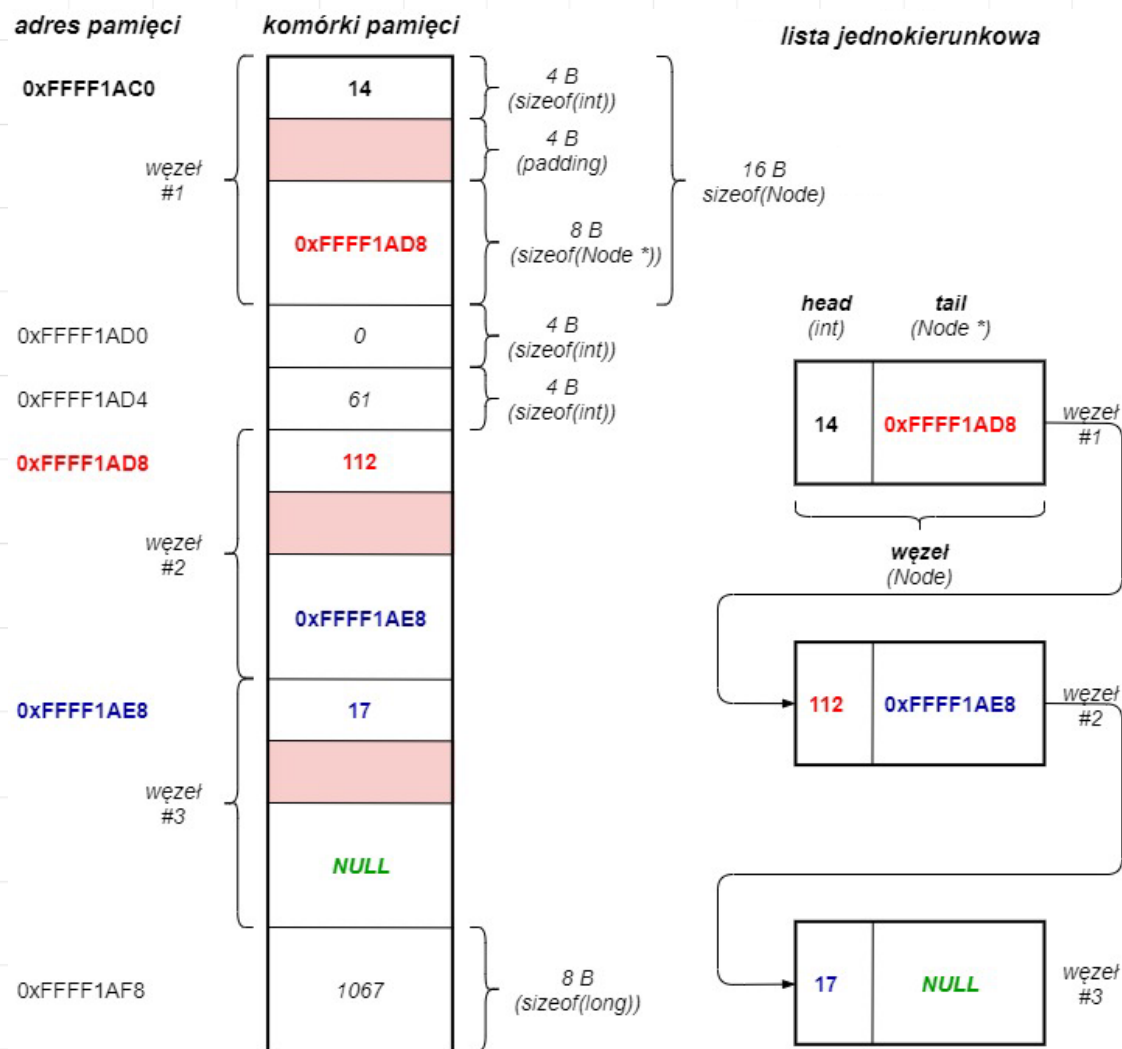
Lista jako struktura danych



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 vs. Tablica - podsumowanie

- Tablice statyczne – z góry zdefiniowany rozmiar, brak możliwości modyfikacji.
- Tablice dynamiczne – rozmiar może być modyfikowany, zwiększany, zmniejszany, ale wstawienie dodatkowego elementu w konkretne miejsce w tablicy wymaga już szerszego zabiegu z kopiowaniem poszczególnych pól tablicy. W obu przypadkach sprowadza się do tego, że na ogół tworzy się tablice z zapasem miejsca.
- Listy wprowadzają znaczne lepsze wykorzystanie (i oszczędność?) pamięci. Dzięki dynamicznej alokacji pamięci pod węzeł listy, list mogą rozrastać się lub kurczyć w czasie działania programu, w dowolnym miejscu tej listy, co jednak:
 - wymaga dodatkowej pamięci pod wskaźnik na kolejny węzeł listy,
 - zwiększa ryzyko wycieków pamięci.
- Elementy tablicy zajmują liniową i ciągłą przestrzeń adresową, co zawsze oznacza praktycznie natychmiastowy dostęp do nich.
- Elementy listy powiązane są wskaźnikami, co wymaga sekwencyjnego poruszania się po niej, w celu odczytu lub usunięcia danego elementu.

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

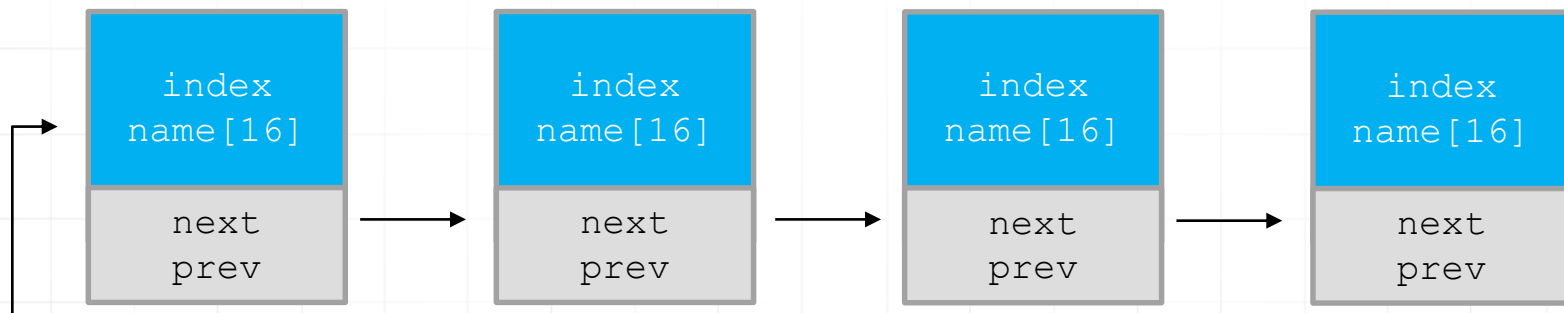
```
typedef struct Student {  
    int index;  
    char name[16];  
    struct Student * next;  
    struct Student * prev;  
} StudentPwr;
```

node:



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

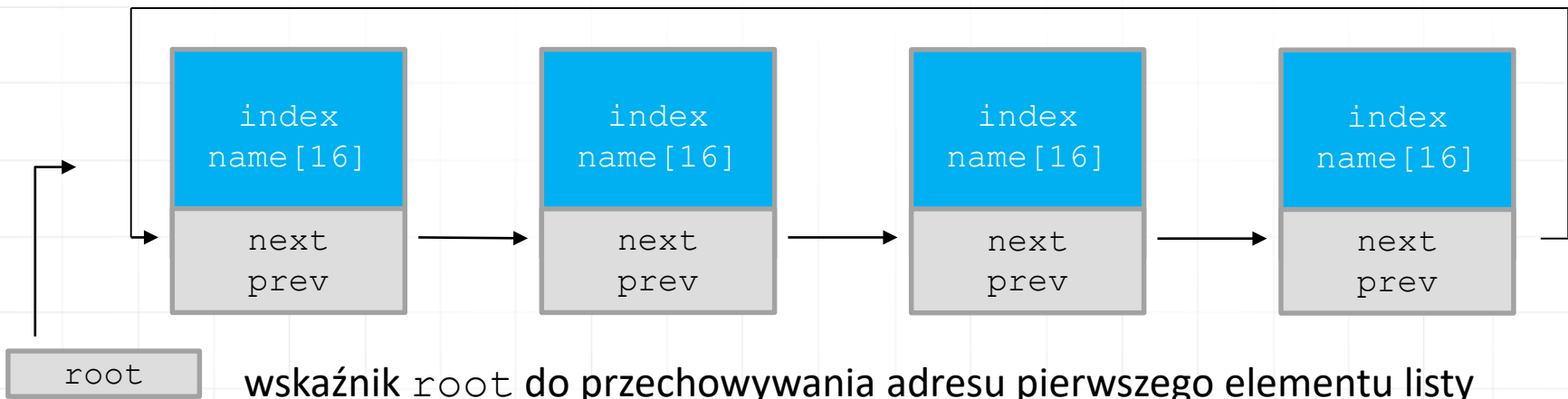
```
typedef struct Student {  
    int index;  
    char name[16];  
    struct Student * next;  
    struct Student * prev;  
} StudentPwr;
```

node:



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

➤ lista jednokierunkowa cykliczna



Lista jako struktura danych

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

node:

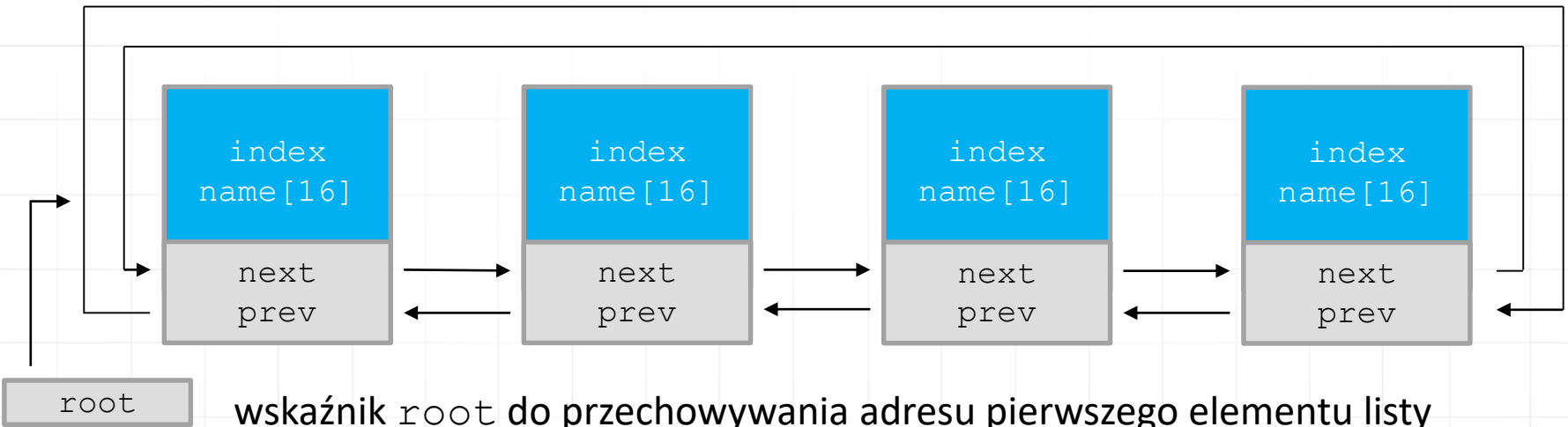


head

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

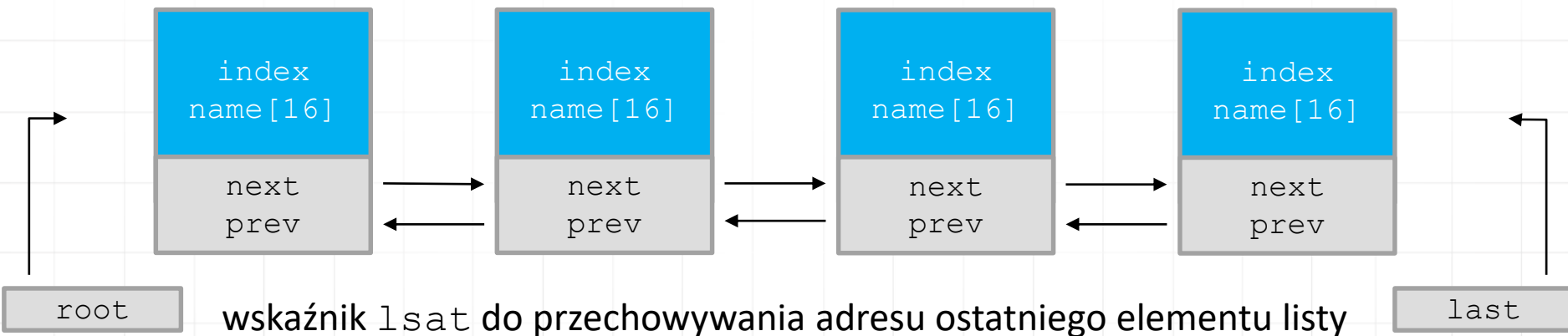
```
typedef struct Student {  
    int index;  
    char name[16];  
    struct Student * next;  
    struct Student * prev;  
} StudentPwr;
```

node:



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

➤ lista dwukierunkowa



Lista jako struktura danych

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

node:

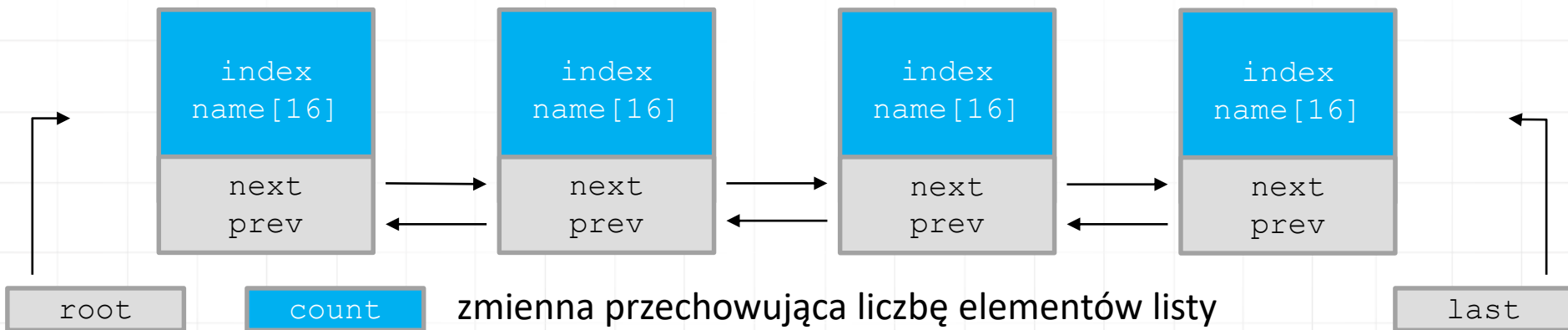


head

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;
```

node:



head

tail

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;
```

node:



head

tail

Następnie stwórzmy pierwszy element listy:

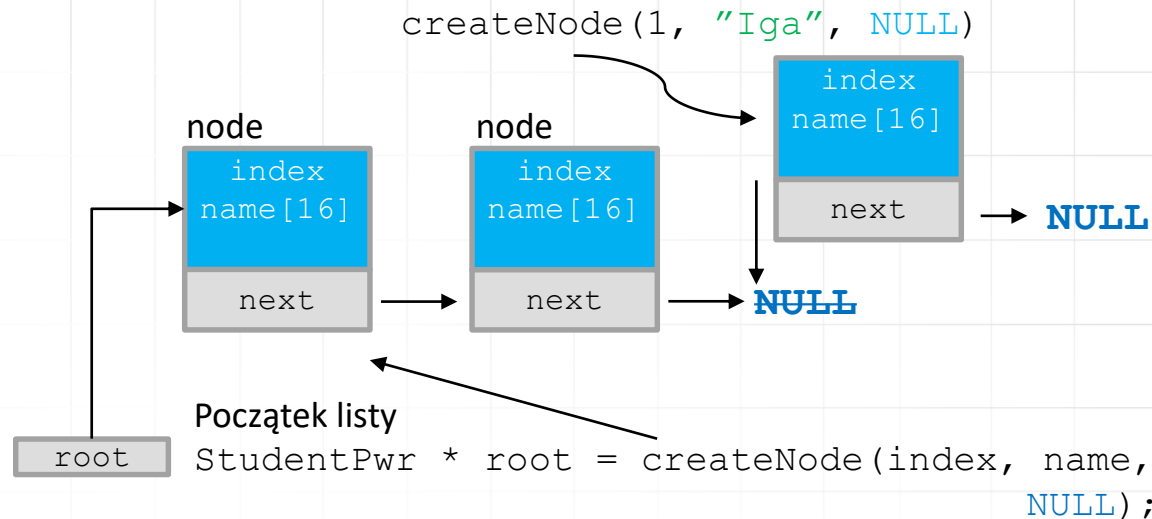
```
StudentPwr * root = createNode(266666, "Kowalski", NULL);
```

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.

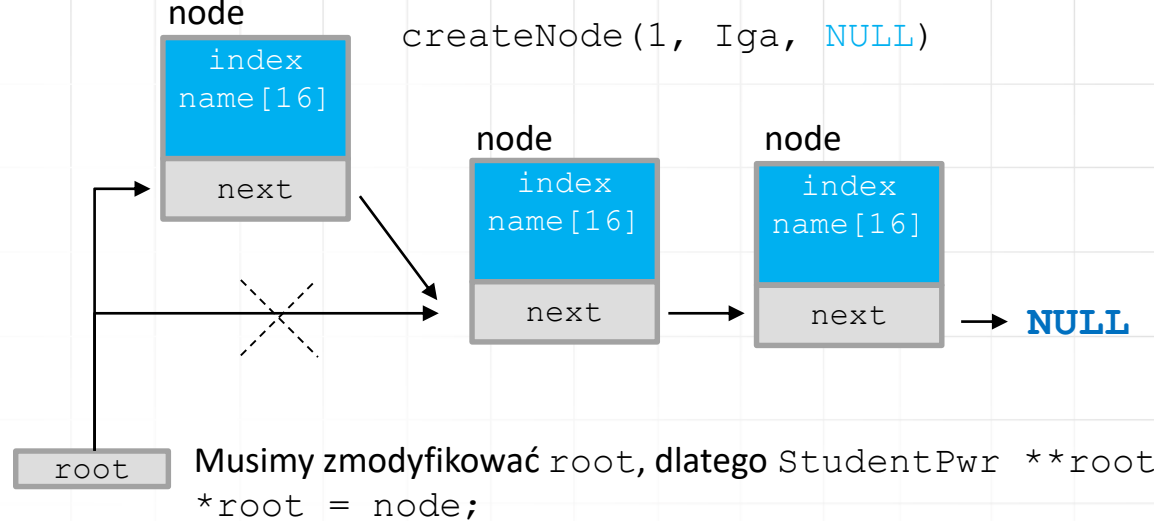
```
void pushBack (StudentPwr ** root , int index , char * name) {  
    StudentPwr * currentNode = *root;  
    if ( root != NULL && *root != NULL ) {  
        while ( currentNode->next != NULL )  
            currentNode = currentNode->next; // Przewin na koniec listy  
        currentNode->next = createNode(index, name, NULL);  
    } else {  
        //jezeli to pierwszy element  
        StudentPwr * node = createNode(index, name, NULL);  
        *root = node;  
    }  
}
```

Pomocnicza zmienna wskaźnikowa, która pozwoli nam na przewijanie listy.

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

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:

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

```
    if ( root != NULL && *root != NUUL ) {
```

```
        StudentPwr * node = createNode(index, name, *root);
```

```
        *root = node;
```

```
    } else { //jeżeli to pierwszy element
```

```
        StudentPwr * node = createNode(index, name, NULL);
```

```
        *root = node;
```

```
    }
```

```
}
```

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

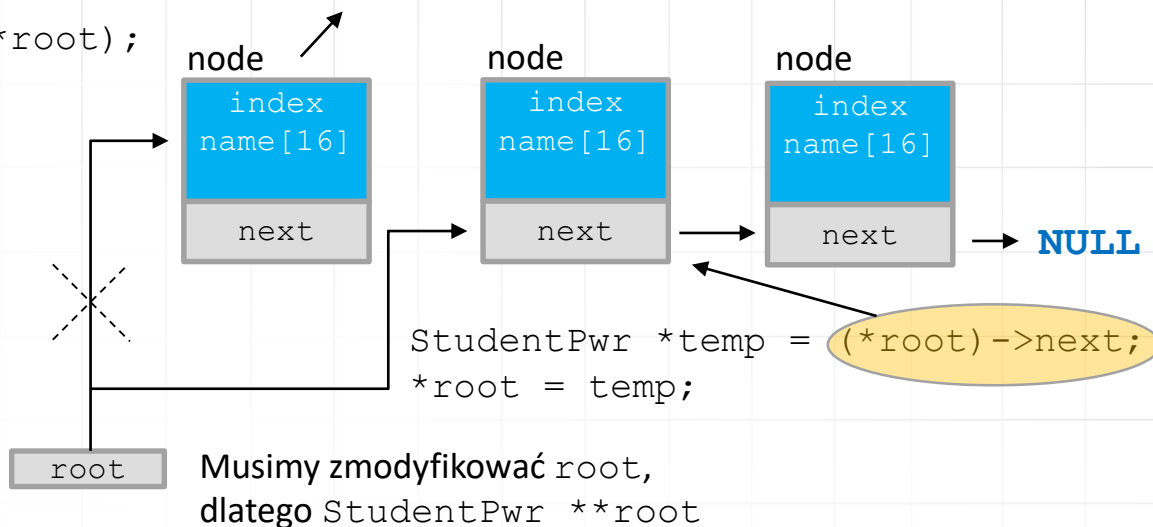
Zmodyfikuj i ustaw nowy adres listy.

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

Lista jednokierunkowa

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

free (*root);



Możemy również usunąć węzeł, np. pierwszy:

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

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

Tymczasowo zapamiętaj adres drugiego elementu w zmiennej wskaźnikowej temp.

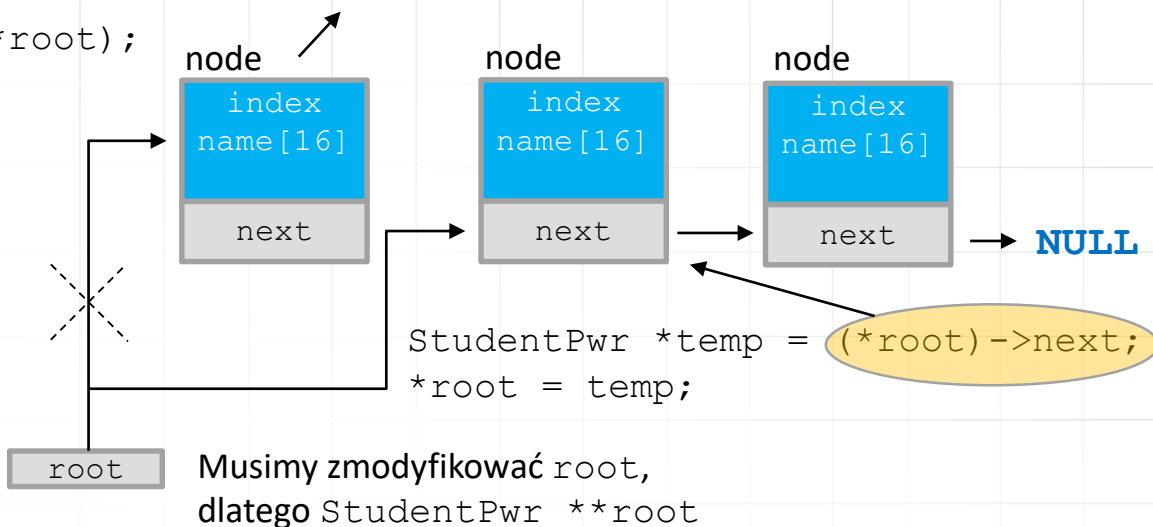
Zwolnij to co wskazuje root.

Pod wskaźnik wskazujący na pierwszy element przypisz adres drugiego elementu

Lista jednokierunkowa

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

free (*root);



Możemy również usunąć węzeł, np. pierwszy, ale zwrócić również go jako wartość:

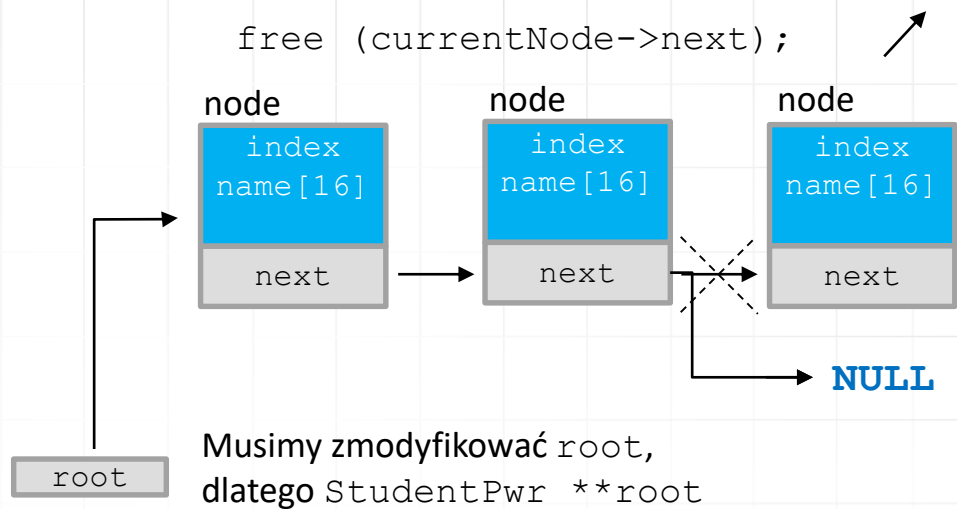
```
StudentPwr popFront ( StudentPwr ** root ) {
```

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

Pomocnicza zmienna strukturalna, która pozwoli nam przechowanie wartości z węzła po tym jak go usuniemy.

Lista jednokierunkowa

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



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.

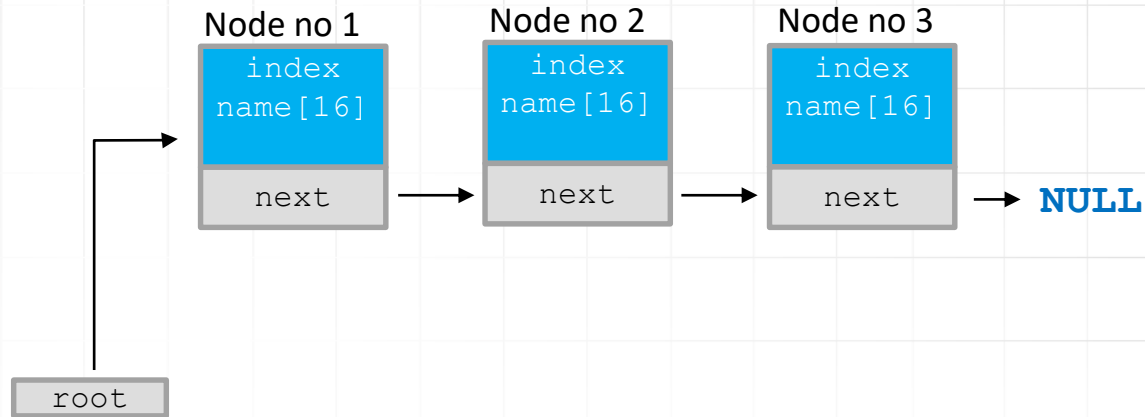
Przeviń do przedostatniego elementu.

Zwolnij adres ostatniego elementu.

Ustaw przedostatni element ostatnim.

Lista jednokierunkowa

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

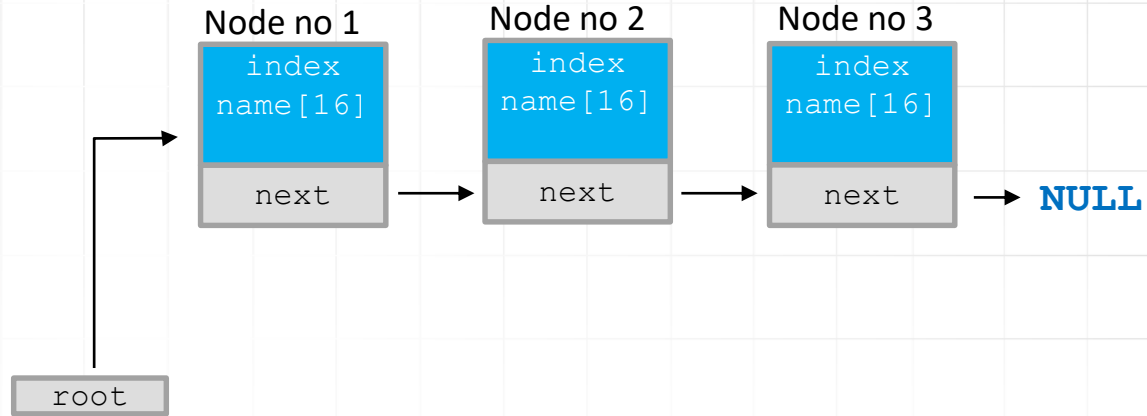


Możemy zdefiniować funkcję w celu wyświetlenia danych z listy:

```
void showList(StudentPwr * root) {  
    int i = 1;  
    StudentPwr * currentNode = root;  
    if (currentNode != NULL) {  
        // Przejdź 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;
```

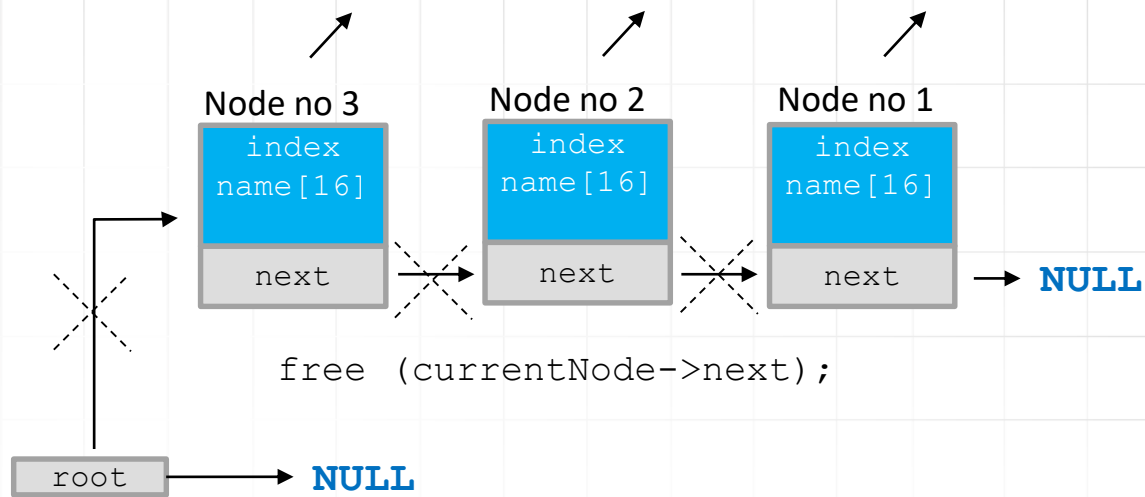


Możemy zdefiniować funkcję w celu zwrócenia ilości węzłów listy:

```
int listSize(StudentPwr * root) {  
    int i = 1;  
    StudentPwr * currentNode = root;  
  
    if ( currentNode != NULL ) {  
        // Przejdź na koniec listy  
        do{  
            currentNode = currentNode -> next ;  
            i++;  
        } while (currentNode->next != NULL)  
  
        return i;  
    }  
}
```

Lista jednokierunkowa

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

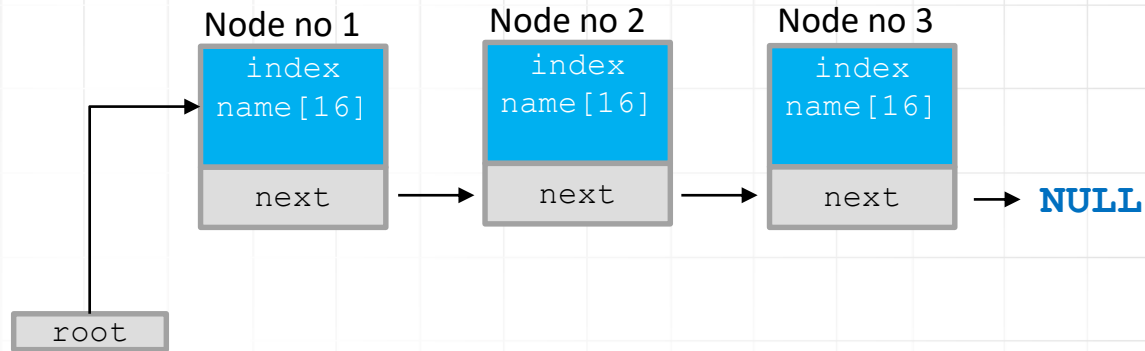


Na koniec wypada zdefiniować funkcję do zwolnienia pamięci po liście:

```
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 nagłówki 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, dane przechowywane w węźle lub adres węzła, lub np. ilość węzłów.

Lista jednokierunkowa - bufor LIFO

Na bazie listy jednokierunkowej można stworzyć:

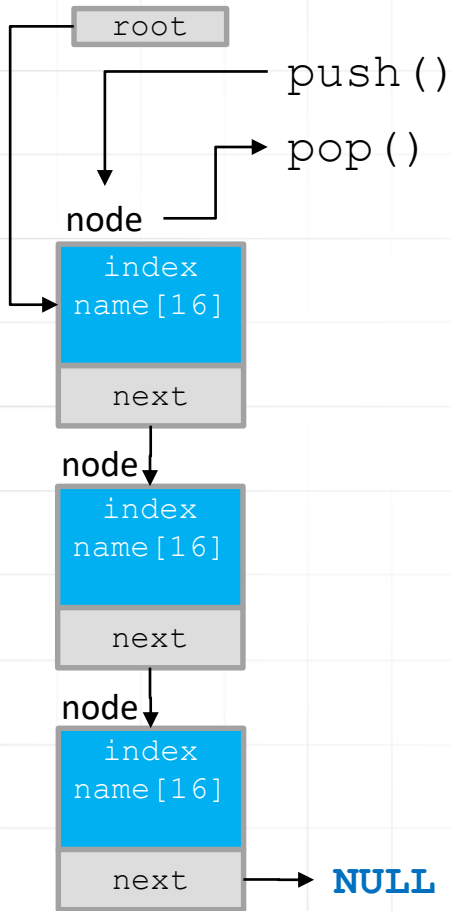
- bufor LIFO (Stos)

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.

```
bool 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:

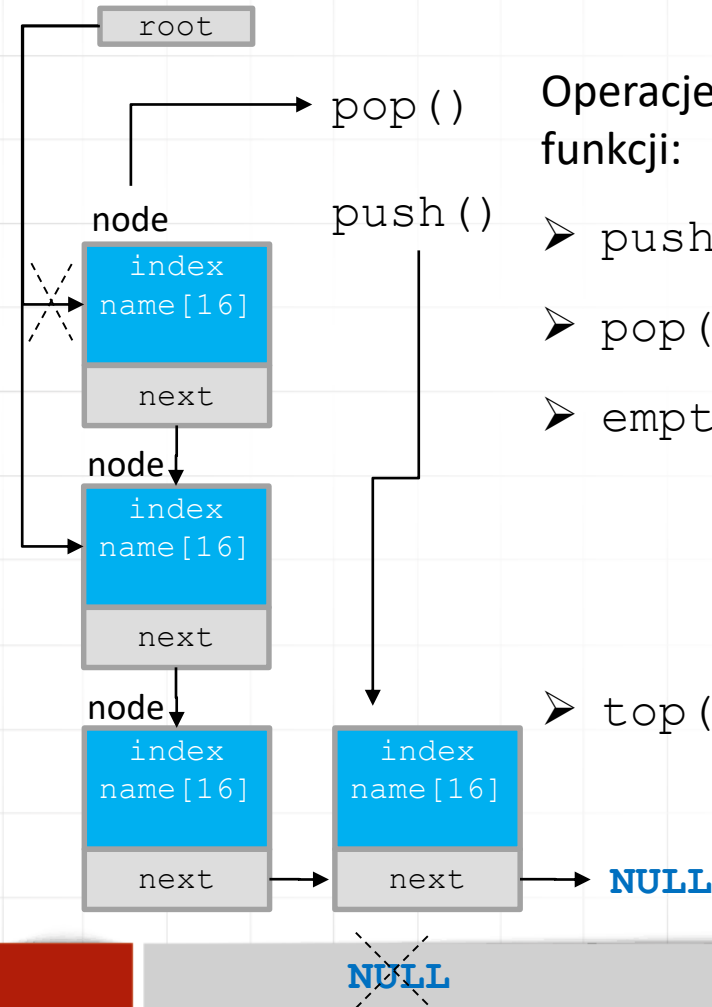
- bufor FIFO (Kolejka)

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.

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

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



Lista dwukierunkowa

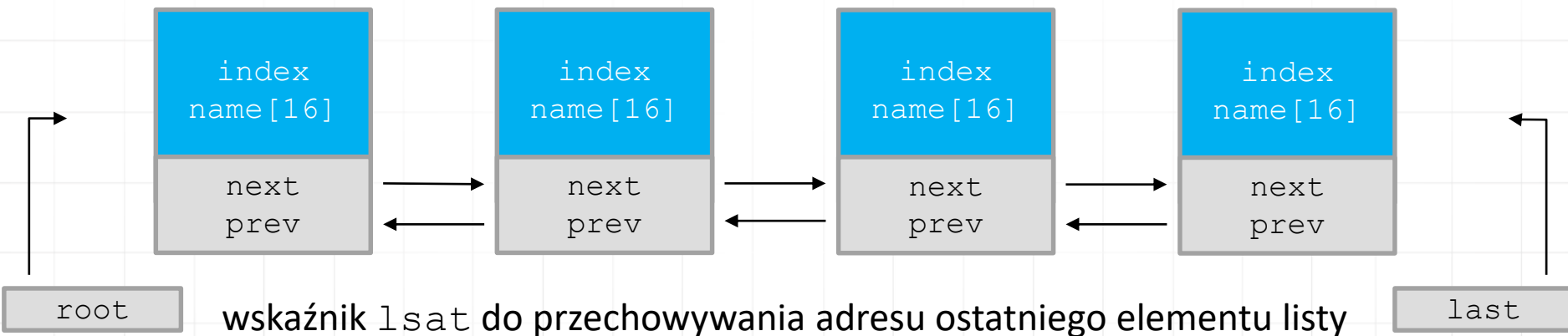
```
typedef struct Student {  
    int index;  
    char name[16];  
    struct Student * next;  
    struct Student * prev;  
} StudentPwr;
```

node:



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

➤ lista dwukierunkowa



Lista dwukierunkowa

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



Funkcja do tworzenia nowego węzła:

```
StudentPwr * createNode(int index , char * name, StudentPwr * next,  
StudentPwr * prev){  
    //Przydziel dynamicznie adres na wezel  
    StudentPwr * node = (StudentPwr *) malloc(sizeof(StudentPwr));  
  
    node->index = index; // inicjalizacja wezla  
    strncpy(node->name, name, 16);  
    node->next = next;  
    node->prev = prev;  
  
    return node; // zwroc adres wezla  
}
```

Lista dwukierunkowa

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

node:



head

tail

Funkcja do wstawiania węzła na początek listy:

```
void pushFront (StudentPwr ** root , int index , char * name) {  
    if ( *root != NULL ) {  
        StudentPwr * node = createNode(index, name, *root, NULL);  
        (*root)->prev = node;  
        *root = node;  
    }  
    else { //jezeli to pierwszy element  
        StudentPwr * node = createNode(index, name, NULL, NULL);  
        *root = node;  
    }  
}
```

Lista dwukierunkowa

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

node:



head

tail

Funkcja do wstawiania węzła na początek listy:

```
StudentPwr * pushBack (StudentPwr ** root , int index , char * name) {  
    StudentPwr * currentNode = *root;  
  
    if (currentNode != NULL){  
        while ( currentNode->next != NULL )  
            currentNode = currentNode->next; // Przewin na koniec listy  
        currentNode->next = createNode(index, name, NULL, currentNode);  
        return currentNode->next;  
    } else { //jezeli to pierwszy element  
        StudentPwr * node = createNode(index, name, NULL, NULL);  
        *root = node;  
        return node;  
    }  
}
```

Lista dwukierunkowa

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

node:



head

tail

Wskaźniki na początek i koniec listy oraz inicjalizacja listy:

```
StudentPwr * root = NULL;  
StudentPwr * last = NULL;
```

```
root = last = createNode(123456, "Kowalski", NULL, NULL);
```

```
pushFront(&root, 123457, "Kowalski");
```

```
last = pushBack (&root , 123458, "Wozniak");
```