ALGORYTMY
I STRUKTURY DANYCH

Temat 4:

Realizacje dynamicznych struktur danych.

Wykładowca: dr inż. Zbigniew TARAPATA

e-mail: Zbigniew.Tarapata@isi.wat.edu.pl

http://www.tarapata.strefa.pl/p_algorytmy_i_struktury_danych/

Współautorami wykładu są: G.Bliźniuk, A.Najgebauer, D.Pierzchała

Wykład : Realizacje dynamicznych struktur danych

- Dynamiczne realizacje struktur listowych (definiowanie elementu listy, dołączanie i usuwanie elementu listy, wyszukiwanie elementu w liście, przestawianie elementów w liście);
- Dynamiczne realizacje struktur drzewiastych (definiowanie elementu drzewa, dołączanie i usuwanie elementu drzewa, wyszukiwanie elementu w drzewie);

Statyczna a dynamiczna struktura danych

Statyczna struktura danych:

- Posiada z góry ustalony rozmiar (bez możliwości jego zmiany);
- Jej deklaracja poprzedza uruchomienie głównej procedury programu;
- Liczba zmiennych o typie statycznych musi być z góry znana;
- Pamięć jest przydzielana na wstępie a oddawana po zakończeniu programu;

Statyczna a dynamiczna struktura danych

Dynamiczną strukturę danych odróżnia sposób przydziału pamięci operacyjnej:

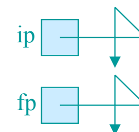
- ilość wymaganej pamięci przez struktury danych nie musi być znana przed uruchomieniem programu;
- przydział pamięci następuje dynamicznie w czasie realizacji odpowiedniej części programu;
- po wykonaniu zadań struktury danych powinny być usunięte a przydzielona pamięć dynamicznie zwolniona;
- zaleta tego podejścia:
  - możliwość dynamicznego tworzenia struktur danych o różnych „kształtach”;
  - „brak” ograniczeń na wielkość struktury;
- wada: złożone operacje dodawania i usuwania elementów struktury;

Arytmetyka wskaźników – przypomnienie

- Utworzenie struktury dynamicznej możliwe jest poprzez zastosowanie wskaźników, adresów, referencji;
- Czy potrafimy odróżnić pojęcie i właściwości:
  - zmiennej,
  - wskaźnika,
  - adresu,
  - referencji?

Arytmetyka wskaźników – przypomnienie

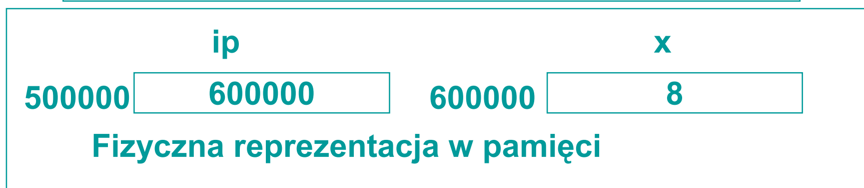
- Deklaracja zmiennej ‘var’ o typie ‘type’:
  - type var;
  - Przykład:
    - int x = 8;
- Deklaracja zmiennej wskaźnikowej ‘name’ do typu ‘type’ :
  - type *name;
  - Przykład:
    - int *ip; /*wskaźnik na typ integer*/
    - float *fp; /*wskaźnik na typ float */



- Przypisanie adresu zmiennej x do wskaźnika ip;
  - ip = &x;
    - & oznacza ‘adres’ i zwraca adres stojącej przy nim operandy;
    - ip teraz wskazuje na x;

Arytmetyka wskaźników – przypomnienie

- Pobranie wartości wskazywanej przez wskaźnik :
  - *int y;*
  - *y = *ip;*
  - y teraz posiada wartość równą x;



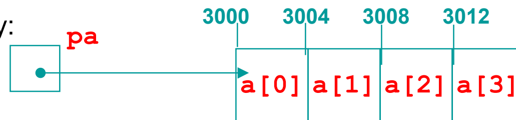
Arytmetyka wskaźników – przypomnienie

- Przeanalizuj i wyjaśnij poniższy fragment kodu:

```
int x=1, y=2, z[10];
int *ip, *iq;
ip = &x;        /* ip wskazuje na x     */
y = *ip;        /* y = 1                */
*ip = 0;        /* x przypisano 0       */
*ip += 1;       /* x zwiększone o 1     */
y = *ip+2;      /* y = 3                */
ip = &z[0];     /* ip wskazuje na z[0]  */
iq = ip;        /* iq wskazuje na z[0]  */
```

Arytmetyka wskaźników – przypomnienie

- Przykład operacji dla typu prostego:
  - *int *px;*
  - *px += 2;*
- px = (adres px) + 2 * rozmiar obiektu wskazywanego przez px;
- dla px równe jest 300 i rozmiaru 4 dla int:
  - 3000 + 2 * 4 = 3008;
- Przykład dla tablicy:
  - *int a[4];*
  - *int *pa;*
  - *pa = a;* /* lub *pa=&a[0];* */
    - pa ++; /*pa wskazuje na a[1] */
    - pa =+ 2; /*pa wskazuje na a[3] */
    - pa --; /*pa wskazuje na a[2] */



Dynamiczne zarządzanie pamięcią operacyjną

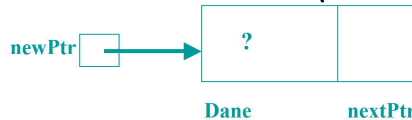
- Dynamiczny przydział pamięci (alokacja):
  - *malloc()* /* z biblioteki stdlib.h */
  - Przykład:
    - *void * malloc(size_t size);*
  - ‘void’ pozwala na wskazanie danych dowolnego typu;
  - argumentem funkcji jest rozmiar przydzielanej pamięci;
  - pamięć przydzielana jest z obszaru zwanego *kopcem*;
  - *sizeof()* – operator do określenia wielkości pamięci:
    - *sizeof(int)* – zwróci rozmiar pamięci potrzebny dla przechowania danych typu integer;
    - *sizeof(struct node)* – zwróci rozmiar pamięci potrzebny dla przechowania danych typu zadeklarowanej struktury ‘node’;

Dynamiczne zarządzanie pamięcią – polecenia

- Przykład użycia *malloc()* i *sizeof()*:
  - (1) *int x=1, y;*
    y = sizeof x;
    lub *y = sizeof (int);*
  - (2)

```
struct node {
      char data;
      struct node *nextPtr };
struct node *newPtr;
newPtr = malloc(sizeof(struct node));
```

rekursywna deklaracja struktury



Dynamiczne zarządzanie pamięcią – polecenia

- Dynamiczne zwolnienie pamięci (na kopiec):
  - *free()* /* z biblioteki stdlib.h */
- Przykład:

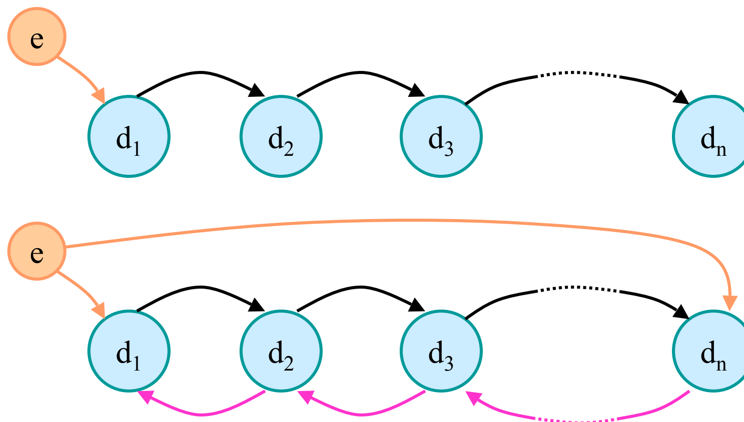
```
struct node {
      char data;
      struct node *nextPtr };
struct node *newPtr;
newPtr = malloc(sizeof(struct node));
…
/* operacje na danych*/
…
free (newPtr);
```

Dynamiczne realizacje struktur listowych

- Definiowanie elementu listy;
- Dołączanie i usuwanie elementu listy;
- Wyszukiwanie elementu w liście;
- Przestawianie elementów w liście;

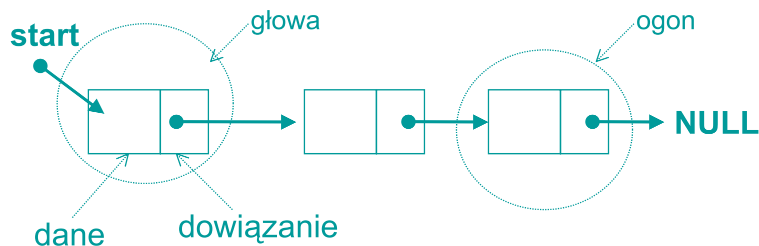
Dynamiczne realizacje struktur listowych

- Model grafowy listy jednokierunkowej (dwukierunkowej)



Dynamiczne realizacje struktur listowych

- Element listy zawiera:
  - Dane elementarne,
  - Odwzorowanie relacji następstwa – informacja o dowiązaniu do innych elementów;



Dynamiczne realizacje struktur listowych

- Deklaracja elementu listy:

```
struct Node {
        char data;
        struct Node *next;
        };
```

dane elementarne

dowiązanie do kolejnego elementu

```
typedef struct Node *NodePtr;
    /* pomocniczy typ wskaźnikowy do struktury 'Node' */
    /* zmienna 'start' tego typu wskazywać będzie głowę listy */
```

Dynamiczne realizacje struktur listowych

- Podstawowe operacje na listach:
  - dołączanie elementu do listy,
  - wyszukiwanie elementu w liście,
  - usuwanie elementu z listy,
  - przestawianie elementów w liście (=>*więcej na wykładzie dotyczącym sortowania list*);

Dynamiczne realizacje struktur listowych

- Algorytm ***dołączania*** elementu do listy ***jednokierunkowej*** (1):
  - Cel:
    - Dodanie nowego elementu do listy;
  - Dane wejściowe:
    - Dowiązanie głowy listy ‘startPtr’;
    - Nowa dana elementarna;

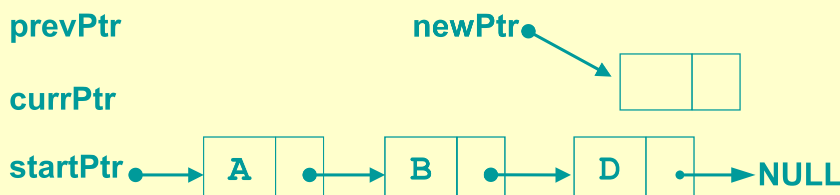
Dynamiczne realizacje struktur listowych

- Algorytm ***dołączania*** elementu do listy ***jednokierunkowej*** (2):
  - Utwórz element i ustal dane elementarne;
  - Znajdź miejsce wstawienia elementu w liście;
  - Wstaw element do listy:
    - Wstaw element jako pierwszy w liście;
    - (lub) Wstaw element we wskazane miejsce w liście;

Dynamiczne realizacje struktur listowych

- Algorytm ***dołączania*** elementu do listy ***jednokierunkowej*** (3):
  - Utwórz element i ustal dane elementarne

```
int insert (NodePtr *startPtr, char nazwa)
{
  Node *newPtr, *currPtr, *prevPtr;
  int retcode=1;
  /* Utwórz element Node */
  newPtr = (Node *)malloc(sizeof(Node));
```



Dynamiczne realizacje struktur listowych

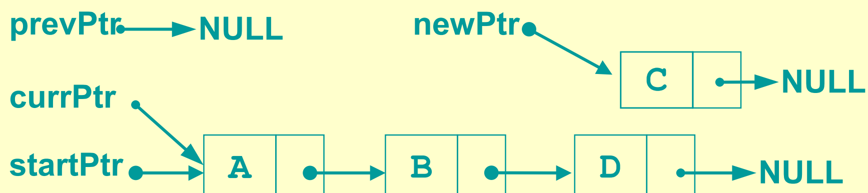
- Algorytm ***dołączania*** elementu do listy ***jednokierunkowej*** (4):
  - Utwórz element i ustal dane elementarne

```
int insert (NodePtr *startPtr, char nazwa)
{
  Node *newPtr, *currPtr, *prevPtr;
  int retcode = 1;
  /* Utwórz element Node */
  newPtr = (Node *)malloc(sizeof(Node));
  if (newPtr == NULL)  /* weryfikacja przydzielonej pamięci*/
     retcode = 0;
  else
  {  /* Ustal dane elementarne w Node */
     newPtr -> data = nazwa;
     newPtr -> next = NULL;
     /* Inicjalizacja wskaźników pomocniczych */
     currPtr = *startPtr;   /* ustaw wskaźnik na głowę listy */
     prevPtr = NULL;
```


Dynamiczne realizacje struktur listowych

- Algorytm ***dołączania*** elementu do listy ***jednokierunkowej*** (4):
  - Utwórz element i ustal dane elementarne

```
int insert (NodePtr *startPtr, char nazwa)
{
  Node *newPtr, *currPtr, *prevPtr;
  int retcode = 1;
  /* Utwórz element Node */
  newPtr = (Node *)malloc(sizeof(Node));
  if (newPtr == NULL)  /* weryfikacja przydzielonej pamięci*/
     retcode = 0;
  else
```

Dynamiczne realizacje struktur listowych

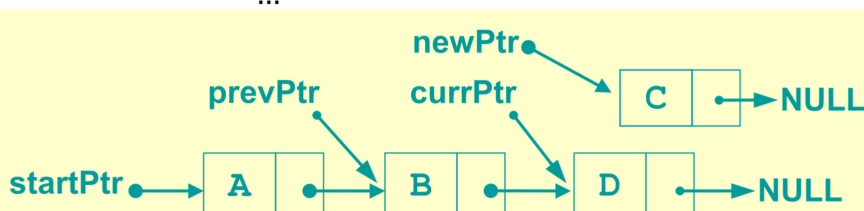
- Algorytm ***dołączania*** elementu do listy ***jednokierunkowej*** (5):
  - Utwórz element i ustal dane elementarne
  - Znajdź miejsce wstawienia elementu w liście
    - Dopóki *currPtr* jest różny od *NULL* oraz dane elementu wstawianego są 'większe' od *currPtr->data:*
      - Ustaw *prevPtr* na *currPtr;*
      - Przesuń *currPtr* na następny element listy;

```
...
/* Znajdź miejsce wstawienia */
while ((currPtr != NULL) && (nazwa > currPtr->data))
{
        prevPtr = currPtr;
        currPtr = currPtr -> next;
}
```


Dynamiczne realizacje struktur listowych

- Algorytm ***dołączania*** elementu do listy ***jednokierunkowej*** (5):
  - Utwórz element i ustal dane elementarne
  - Znajdź miejsce wstawienia elementu w liście
    - Dopóki *currPtr* jest różny od *NULL* oraz dane elementu wstawianego są 'większe' od *currPtr->data:*
      - Ustaw *prevPtr* na *currPtr;*
      - Przesuń *currPtr* na następny element listy;

...



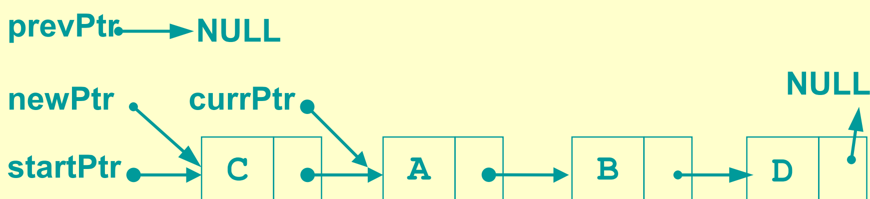
Dynamiczne realizacje struktur listowych

- Algorytm ***dołączania*** elementu do listy ***jednokierunkowej*** (6):
  - Utwórz element i ustal dane elementarne
  - Znajdź miejsce wstawienia elementu w liście
    - Zainicjalizuj *currPtr* na *start* listy a *prevPtr* na *NULL;*
    - Dopóki *currPtr* jest różny od *NULL* i wartość wstawiana jest większa od *currPtr->data:*
      - Ustaw *prevPtr* na *currPtr;*
      - Przesuń *currPtr* na następny element listy;
  - Wstaw element do listy:
    - Wstaw element jako pierwszy w liście:
      - Ustaw pole *next* elementu wstawianego na pierwszy element listy;
      - Ustaw wskaźnik do listy na element wstawiony;
    - (lub) Wstaw element we wskazane miejsce w liście:
      - Ustaw pole *next* elementu *prevPtr* na element wstawiany;
      - Ustaw pole *next* elementu wstawianego na element *currPtr;*

Dynamiczne realizacje struktur listowych

- Algorytm ***dołączania*** elementu do listy ***jednokierunkowej*** (7):

```
/* Wstaw element */
if (prevPtr == NULL)
/* Wstaw element jako pierwszy w liście */
{
  newPtr -> next = *startPtr;
  *startPtr = newPtr;
}
```



Dynamiczne realizacje struktur listowych

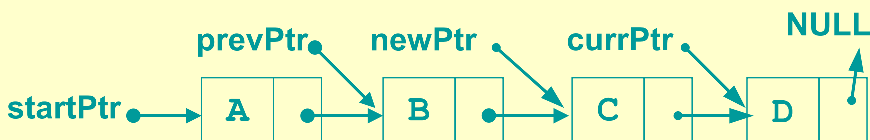
- Algorytm ***dołączania*** elementu do listy ***jednokierunkowej*** (8):

```
/* Wstaw element */
…
/* Wstaw element w miejsce między prevPtr a currPtr */
else
{
  newPtr->next = currPtr;
  prevPtr->next = newPtr;
}
  retcode = 1;
}
return retcode;
}
```


Dynamiczne realizacje struktur listowych

- Algorytm ***dołączania*** elementu do listy ***jednokierunkowej*** (8):

```
/* Wstaw element */
…
/* Wstaw element w miejsce między prevPtr a currPtr */
else
{
  newPtr->next = currPtr;
```

Dynamiczne realizacje struktur listowych

- Uwagi do ***dołączania*** elementu do listy ***dwukierunkowej***:
  - każdy element posiada dodatkowe pole (dowiązanie) *prev*, które dla pierwszego elementu listy jest równe *NULL*;
  - lista może być przeglądana w obydwu kierunkach;
  - często pamięta się dowiązania do pierwszego i ostatniego elementu;
  - należy zawsze uaktualniać dowiązania w obydwu kierunkach (wartości czterech pól);

Czy potrafisz dostosować zaprezentowany algorytm do list dwukierunkowych?

Dynamiczne realizacje struktur listowych

- Uwagi do ***dołączania*** elementu do listy ***cyklicznej***:
  - brak dowiązań wskazujących na NULL;
  - w liście jednoelementowej dowiązania wskazują na ten sam element;
  - aby uniknąć pętli nieskończonej podczas przeglądania listy, należy zastosować 'strażnika' – dowiązanie do pierwszego elementu (umownego);

Czy potrafisz dostosować zaprezentowany algorytm do list cyklicznych?

Dynamiczne realizacje struktur listowych

- Algorytm ***szukania*** elementu w liście ***jednokierunkowej*** (1):
  - Cel:
    - Wyszukanie elementu w liście;
  - Dane wejściowe:
    - Dowiązanie głowy listy 'startPtr';
    - Kryterium poszukiwania, np. wartość danej elementarnej;
  - Uwagi:
    - W skrajnym przypadku należy przejrzeć wszystkie elementy (złożoność $O(n)$);

Dynamiczne realizacje struktur listowych

- Algorytm ***szukania*** elementu w liście ***jednokierunkowej*** (2):
  - Rozpocznij od pierwszego elementu listy;
  - Dopóki aktualny element listy jest różny od NULL oraz dane szukane są różne od danych aktualnego elementu, to przemieść się do następnego elementu listy;
  - Daj dowiązanie do aktualnego elementu;

Dynamiczne realizacje struktur listowych

- Algorytm ***szukania*** elementu w liście ***jednokierunkowej*** (3):

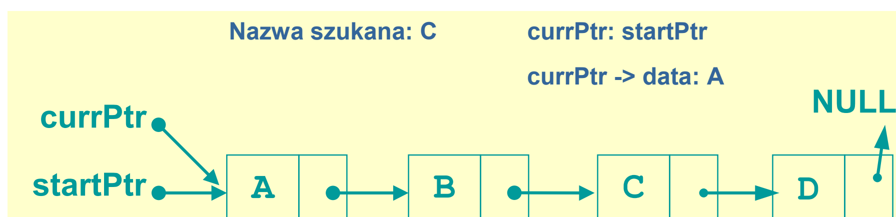
```
Node *Find (NodePtr *startPtr, char nazwa)
{
   NodePtr currPtr = *startPtr;      /* pierwszy element listy
   */
   while ((currPtr != NULL) && (currPtr -> data !=
   nazwa))
      currPtr = currPtr -> next;
   return currPtr;
}
```


Dynamiczne realizacje struktur listowych

- Algorytm ***szukania*** elementu w liście ***jednokierunkowej*** (4):

- Przykład:

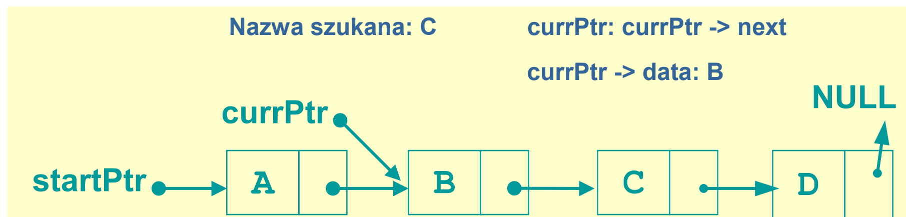
  Node *ele = Find (startPtr, ‘C’);



Dynamiczne realizacje struktur listowych

- Algorytm ***szukania*** elementu w liście ***jednokierunkowej*** (5):
- Przykład:

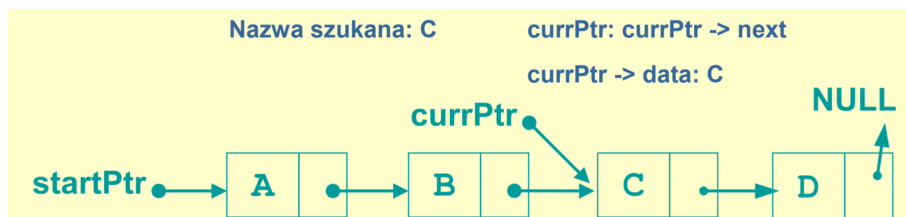
  Node *ele = Find (startPtr, 'C');



Dynamiczne realizacje struktur listowych

- Algorytm ***szukania*** elementu w liście ***jednokierunkowej*** (6):
- Przykład:

  Node *ele = Find (startPtr, 'C');



Dynamiczne realizacje struktur listowych

- Algorytm ***usuwania*** elementu z listy ***jednokierunkowej*** (1):
  - Cel:
    - Usunięcie elementu z listy;
  - Dane wejściowe:
    - Dowiązanie głowy listy 'startPtr';
    - Opis elementu, np. wartość danej elementarnej;

Dynamiczne realizacje struktur listowych

- Algorytm ***usuwania*** elementu z listy ***jednokierunkowej*** (2):
  - Jeżeli dane są zgodne z danymi pierwszego elementu listy
    - Usuń pierwszy element listy;
  - W p.p. znajdź element do usunięcia w liście;
  - Jeżeli nie znaleziono elementu, generuj komunikat;
  - W p.p. usuń znaleziony element;

Dynamiczne realizacje struktur listowych

- Algorytm ***usuwania*** elementu z listy ***jednokierunkowej*** (3):
  - Jeżeli dane są zgodne z danymi pierwszego elementu listy
    - Usuń pierwszy element listy;

```
int delete (NodePtr *startPtr, char nazwa)
   {
     NodePtr prevPtr, currPtr, tempPtr;
     int retcode;
     if (*startPtr == NULL)                /* Lista pusta */
       retcode = 0;
     else
     {
       if (nazwa == (*startPtr)->data)   /* Usuń pierwszy element listy */
       {
              tempPtr = *startPtr;
              *startPtr = (*startPtr)->next;
              free (tempPtr);
              retcode = 1;
       }
```


Dynamiczne realizacje struktur listowych

- Algorytm ***usuwania*** elementu z listy ***jednokierunkowej*** (4):
  - Jeżeli dane są zgodne z danymi pierwszego elementu listy
    - Usuń pierwszy element listy;
  - W p.p. znajdź w liście element do usunięcia:

```
…
else
   {  /* znajdź w liście element do usunięcia */
     prevPtr = *startPtr;                 /* początek listy */
     currPtr = (*startPtr)->next;         /* drugi element*/

   while (currPtr != NULL && currPtr -> data != nazwa)
     {
              prevPtr = currPtr;
              currPtr = currPtr -> next;
     }
```

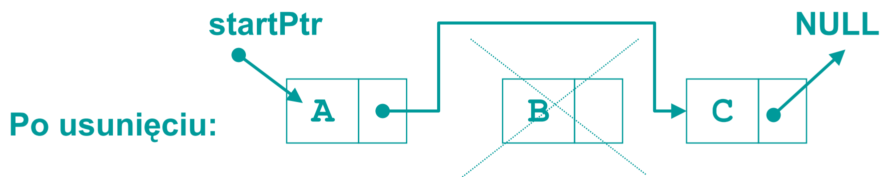
Dynamiczne realizacje struktur listowych

- Algorytm ***usuwania*** elementu z listy ***jednokierunkowej*** (5):
  - Jeżeli dane są zgodne z danymi pierwszego elementu listy
    - Usuń pierwszy element listy;
  - W p.p. znajdź element do usunięcia w liście:
    - Jeżeli nie znaleziono elementu, generuj komunikat;
    - W p.p. usuń znaleziony element;

```
...
if (currPtr == NULL)
   retcode = 0;     /* node not found */
else
{  /* Usuń znaleziony element */
   tempPtr = currPtr;
   prevPtr->next = currPtr->next;
   free (tempPtr);
   retcode = 1;
} } }
return retcode;
```


Dynamiczne realizacje struktur listowych

- Algorytm ***usuwania*** elementu z listy ***jednokierunkowej*** (6):

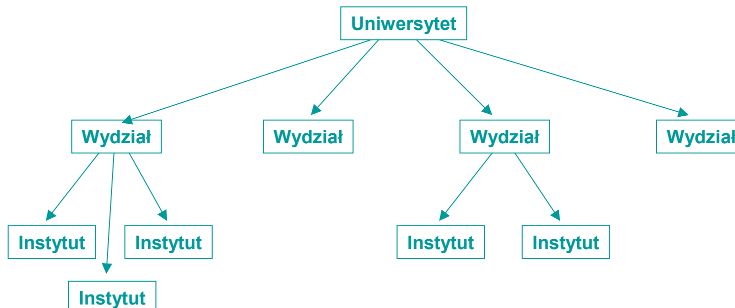


Dynamiczne realizacje struktur drzewiastych

- Definiowanie elementu drzewa;
- Dołączanie i usuwanie elementu drzewa;
- Wyszukiwanie elementu w drzewie;

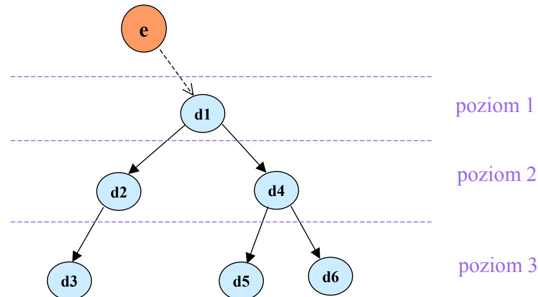
Dynamiczne realizacje struktur drzewiastych

- Drzewiastą strukturą danych nazywamy strukturę danych ***S=(D, R, e)***, w której relacja porządkująca ***N*** opisuje kolejne, rekurencyjne powiązania pomiędzy danymi elementarnymi drzewa, tworzącymi kolejne „poddrzewa”.
- Przykład struktury drzewiastej



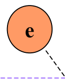
Dynamiczne realizacje struktur drzewiastych

- Drzewo binarne (dwójkowe):
  - Drzewo o stopniu 2 (każdy węzeł ma co najwyżej dwóch ***następników – potomków***: lewego i prawego);
  - Ostatnimi potomkami są ***liście*** (elementy, które nie mają potomków);



Dynamiczne realizacje struktur drzewiastych

- Drzewo binarne (dwójkowe):
  - Drzewo o stopniu 2 (każdy węzeł ma co najwyżej dwóch ***następników – potomków***: lewego i prawego);
  - Ostatnimi potomkami są ***liście*** (elementy, które nie mają potomków);



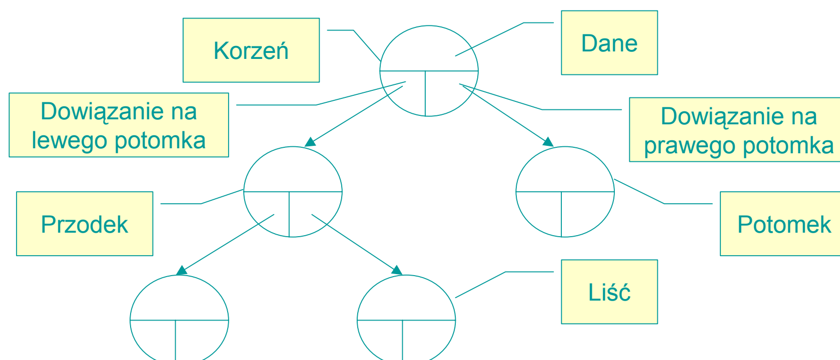
Czy drzewo z poprzedniego slajdu (Uniwersytet) jest drzewem binarnym?

Dynamiczne realizacje struktur drzewiastych

- Zupełne drzewo binarne (dwójkowe):
  - Każdy węzeł, z wyjątkiem liści, ma dokładnie dwóch potomków;
- Drzewo poszukiwań binarnych (BST):
  - Dla każdego węzła (nie liścia) wszystkie wartości przechowywane w lewym poddrzewie są mniejsze od jego wartości oraz przeciwnie dla drzewa prawego;
- Drzewo AVL (1962 – Adelson-Velskii, Landis)
  - Drzewo BST jest drzewem AVL wtedy, kiedy dla każdego wierzchołka wysokości dwóch jego poddrzew różnią się o co najwyżej 1 poziom;
- Kopiec
  - Wartości przechowywane w potomkach każdego węzła są mniejsze od wartości węźle;
  - Drzewo jest szczelnie wypełniane od lewego poddrzewa;
  - Liście leżą na co najwyżej dwóch sąsiednich poziomach;

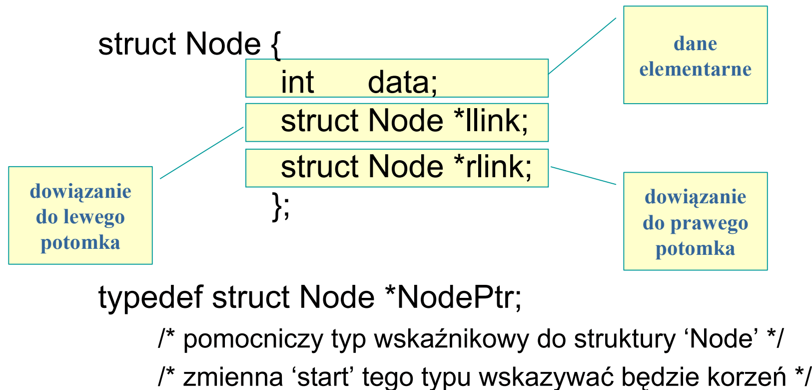
Dynamiczne realizacje struktur drzewiastych

- Element drzewa zawiera:
  - Dane elementarne,
  - Realizację relacji następstwa – dowiązania do następników;



Dynamiczne realizacje struktur drzewiastych

- Deklaracja elementu drzewa binarnego:




```
typedef struct Node *NodePtr;
    /* pomocniczy typ wskaźnikowy do struktury 'Node' */
    /* zmienna 'start' tego typu wskazywać będzie korzeń */
```


Dynamiczne realizacje struktur drzewiastych

- Podstawowe operacje na drzewach ***binarnych***:
  - szukanie elementu w drzewie,
  - przeglądanie drzewa;
  - dołączanie elementu do drzewa,
  - usuwanie elementu z drzewa,
- Uwaga:
  - Operacje te często są realizowane rekurencyjnie;

Dynamiczne realizacje struktur drzewiastych

- Algorytm ***szukania*** elementu w drzewie ***binarnym*** (1):
  - Cel:
    - uzyskanie dowiązania do węzła;
    - można je interpretować jako identyfikację węzła;
  - Dane wejściowe:
    - Dowiązanie do korzenia drzewa 'Root';
    - Kryterium poszukiwania, np. wartość danej elementarnej;
  - Uwagi:
    - kolejność przeszukiwania dowolna – w skrajnym przypadku należy przejrzeć wszystkie węzły w drzewie (złożoność $O(n)$);
    - stosowane rozwiązania: pętla lub rekurencja;

Dynamiczne realizacje struktur drzewiastych

- Algorytm ***szukania*** elementu w drzewie ***binarnym*** (2):
  - Ustaw aktualne dowiązanie na korzeń drzewa;
  - Dopóki aktualne dowiązanie jest różne od NULL:
    - Jeżeli wartość szukana jest mniejsza od danej aktualnego węzła, to szukaj w jego lewym poddrzewie;
    - Jeżeli wartość szukana jest większa od danej aktualnego węzła, to szukaj w jego prawym poddrzewie;
    - Jeżeli wartość szukana jest równa danej aktualnego węzła, to koniec – daj dowiązanie do węzła;

Dynamiczne realizacje struktur drzewiastych

- Algorytm ***szukania*** elementu w drzewie ***binarnym*** (3):
  - Wersja procedury z pętlą:

```
NodePtr find (int inValue, NodePtr node)
{
  while (node) {
    if (inValue == node -> data)
      return node;
    else if (inValue < node -> data)
      node = node -> llink;
    else if (inValue > node -> data)
      node = node -> rlink;
    return NULL;
}
```

przejście do lewego poddrzewa

przejście do prawego poddrzewa

Dynamiczne realizacje struktur drzewiastych

- Algorytm ***szukania*** elementu w drzewie ***binarnym*** (4):
  - Wersja procedury rekurencyjnej:

```
NodePtr find (int inValue, NodePtr node)
{
  if (node) {
    if (inValue == node -> data)
      return node;
    else if (inValue < node -> data)
      return find ( inValue, node > llink);
    else if (inValue > node -> data)
      return find ( inValue, node > rlink);
  else return NULL;
}
```

wywołanie procedury dla lewego poddrzewa

wywołanie procedury dla prawego poddrzewa

Dynamiczne realizacje struktur drzewiastych

- Algorytm ***przeglądanie*** drzewa ***binarnego*** (1):
  - Cel:
    - jednokrotne „odwiedzenie” każdego elementu drzewa;
    - można je interpretować jako umieszczenie wszystkich węzłów w jednej linii – linearyzacja drzewa;
  - Dane wejściowe:
    - Dowiązanie do korzenia drzewa ‘Root’;
  - Uwagi:
    - kolejność przejścia dowolna – liczba możliwych ścieżek w drzewie o n węzłach wynosi n! (permutacja);
    - najczęściej stosowane sposoby: wszerz i w głąb;

Dynamiczne realizacje struktur drzewiastych

- Algorytm ***przeglądanie*** drzewa ***binarnego w głąb*** (2):
  - Wersja „inorder” – LVR
    - Przejście do lewego poddrzewa (L);
    - Odwiedzenie węzła (V);
    - Przejście do lewego poddrzewa (R);
  - Wersja „preorder” – VLR
    - Odwiedzenie węzła (V);
    - Przejście do lewego poddrzewa (L);
    - Przejście do lewego poddrzewa (R);
  - Wersja „postorder” – LRV
    - Przejście do lewego poddrzewa (L);
    - Przejście do lewego poddrzewa (R);
    - Odwiedzenie węzła (V);

Dynamiczne realizacje struktur drzewiastych

- Algorytm ***przeglądanie*** drzewa ***binarnego w głąb*** (3):
  - Wersja procedury „inorder" – LVR

```
void inorder (NodePtr node)
{
  if (node)
  {
    inorder (node -> llink);
    visit (node);
    inorder (node -> rlink);
  }
}
```


Dynamiczne realizacje struktur drzewiastych

- Algorytm ***przeglądanie*** drzewa ***binarnego w głąb*** (4):
  - Wersja procedury „preorder" – VLR

```
void preorder (NodePtr node)
{
  if (node)
  {
    visit (node);
    preorder (node -> llink);
    preorder (node -> rlink);
  }
}
```

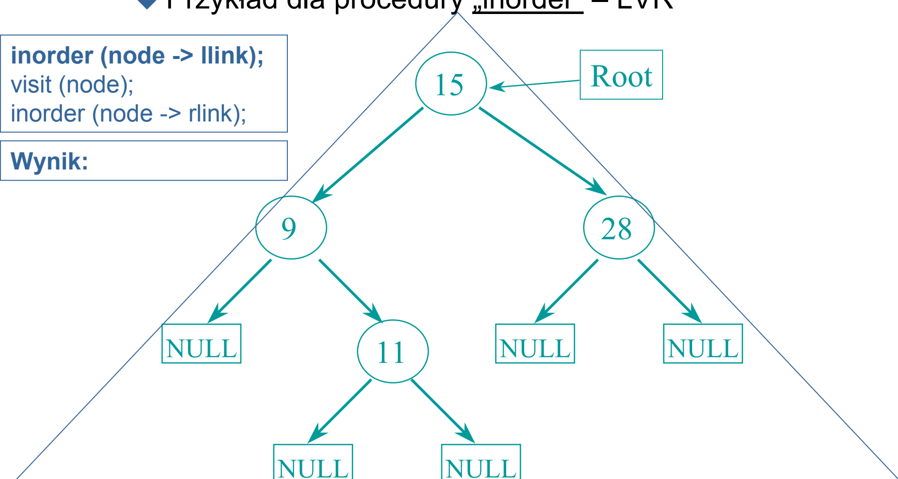
Dynamiczne realizacje struktur drzewiastych

- Algorytm ***przeglądanie*** drzewa ***binarnego w głąb*** (5):
  - Wersja procedury „postorder” – LRV

```
void postorder (NodePtr node)
{
 if (node)
 {
   postorder (node -> llink);
   postorder (node -> rlink);
   visit (node);
 }
}
```


Dynamiczne realizacje struktur drzewiastych

- Algorytm ***przeglądanie*** drzewa ***binarnego w głąb*** (6):
  - Przykład dla procedury „inorder” – LVR

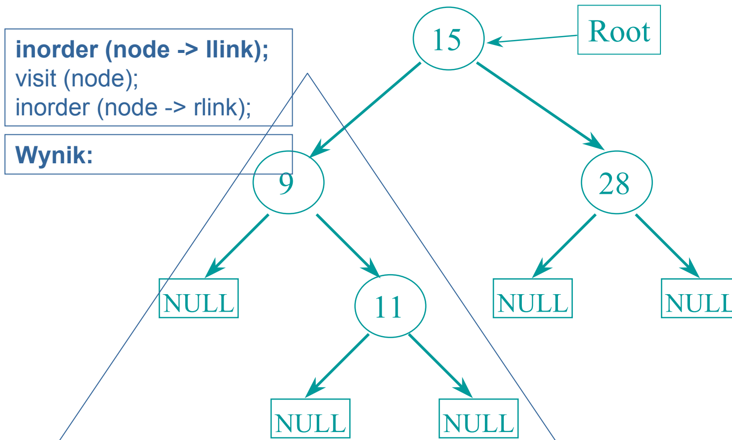


Dynamiczne realizacje struktur drzewiastych

- Algorytm ***przeglądanie*** drzewa ***binarnego w głąb*** (7):
  - Przykład dla procedury „inorder" – LVR

inorder (node -> llink);
visit (node);
inorder (node -> rlink);

Wynik:

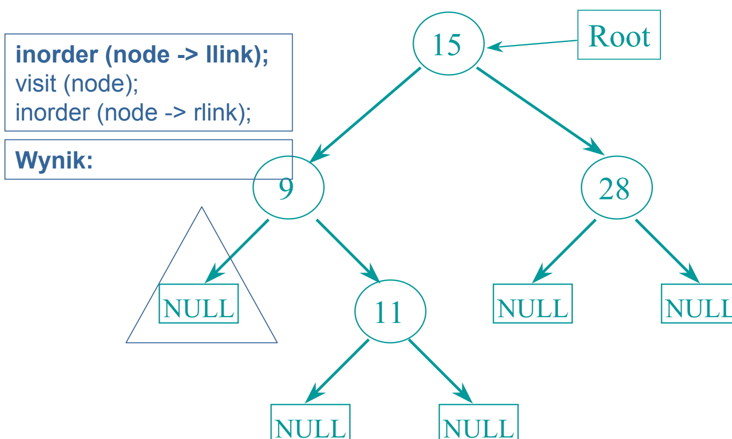


Dynamiczne realizacje struktur drzewiastych

- Algorytm ***przeglądanie*** drzewa ***binarnego w głąb*** (8):
  - Przykład dla procedury „inorder" – LVR

inorder (node -> llink);
visit (node);
inorder (node -> rlink);

Wynik:

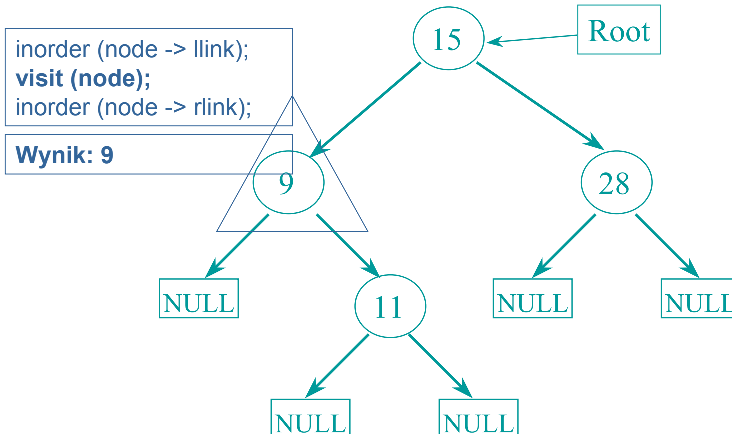


Dynamiczne realizacje struktur drzewiastych

- Algorytm ***przeglądanie*** drzewa ***binarnego w głąb*** (9):
  - Przykład dla procedury „inorder" – LVR

```
inorder (node -> llink);
visit (node);
inorder (node -> rlink);
```

Wynik: 9

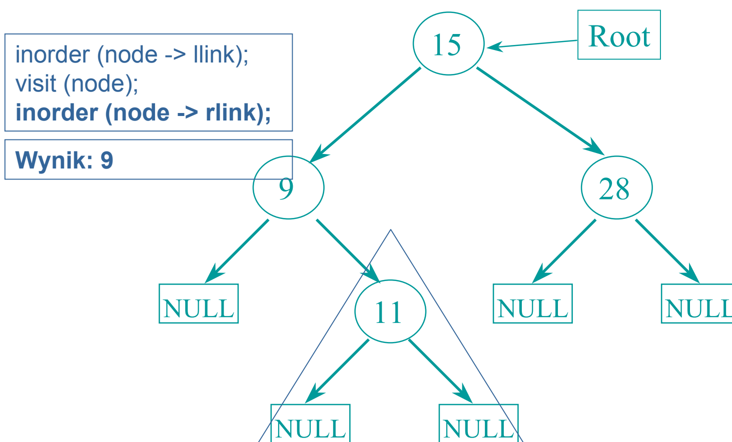


Dynamiczne realizacje struktur drzewiastych

- Algorytm ***przeglądanie*** drzewa ***binarnego w głąb*** (10):
  - Przykład dla procedury „inorder" – LVR

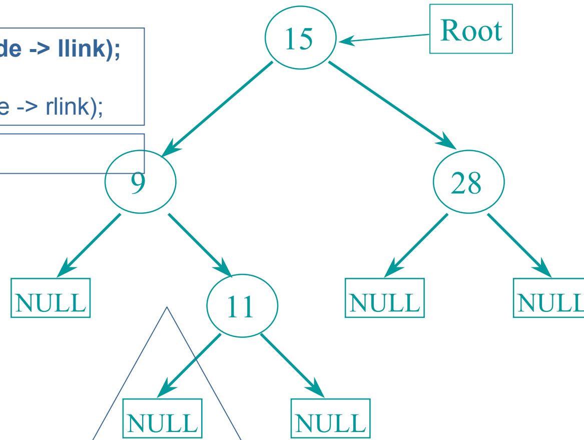
```
inorder (node -> llink);
visit (node);
inorder (node -> rlink);
```

Wynik: 9



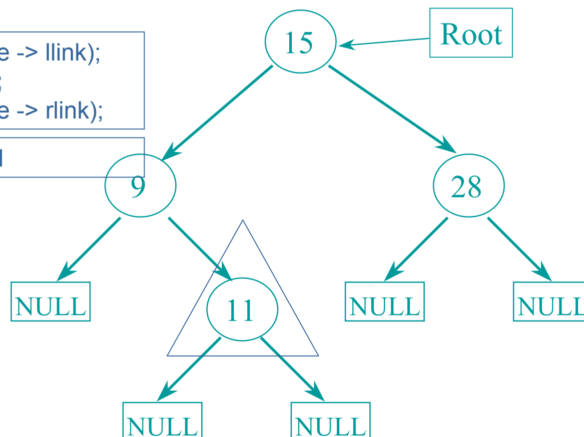
Dynamiczne realizacje struktur drzewiastych

- Algorytm ***przeglądanie*** drzewa ***binarnego w głąb*** (11):
  - Przykład dla procedury „inorder” – LVR



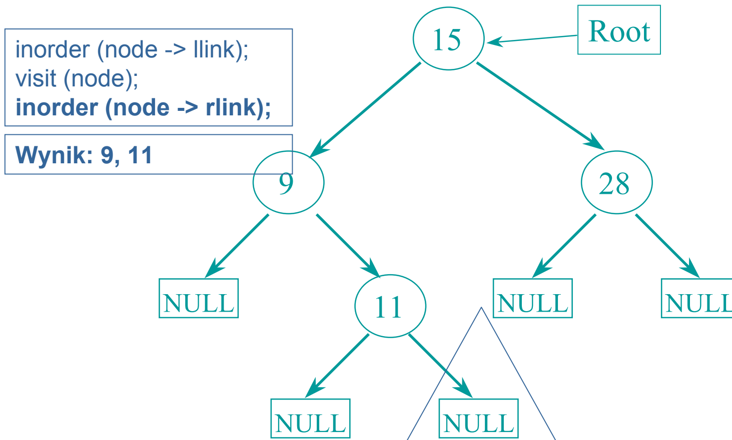
Dynamiczne realizacje struktur drzewiastych

- Algorytm ***przeglądanie*** drzewa ***binarnego w głąb*** (12):
  - Przykład dla procedury „inorder” – LVR



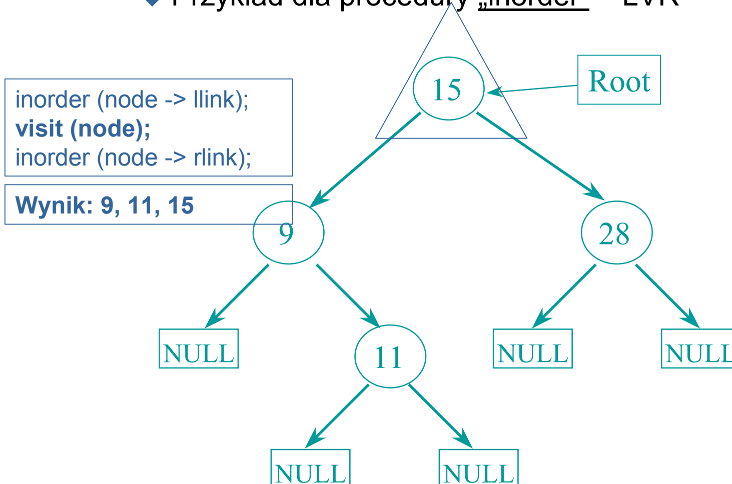
Dynamiczne realizacje struktur drzewiastych

- Algorytm ***przeglądanie*** drzewa ***binarnego w głąb*** (13):
  - Przykład dla procedury „inorder" – LVR



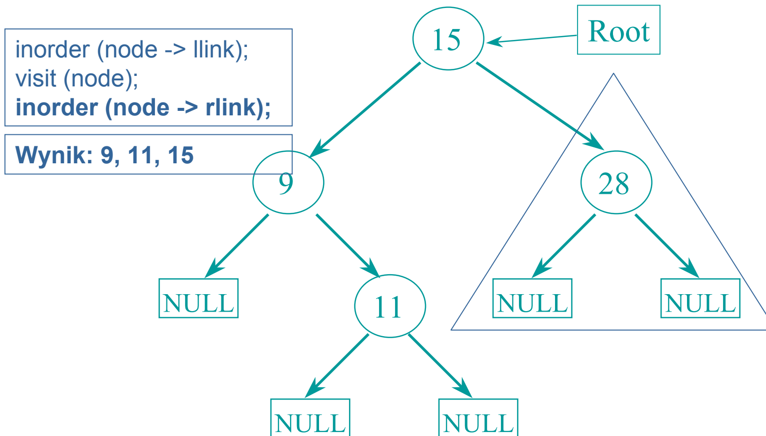
Dynamiczne realizacje struktur drzewiastych

- Algorytm ***przeglądanie*** drzewa ***binarnego w głąb*** (14):
  - Przykład dla procedury „inorder" – LVR



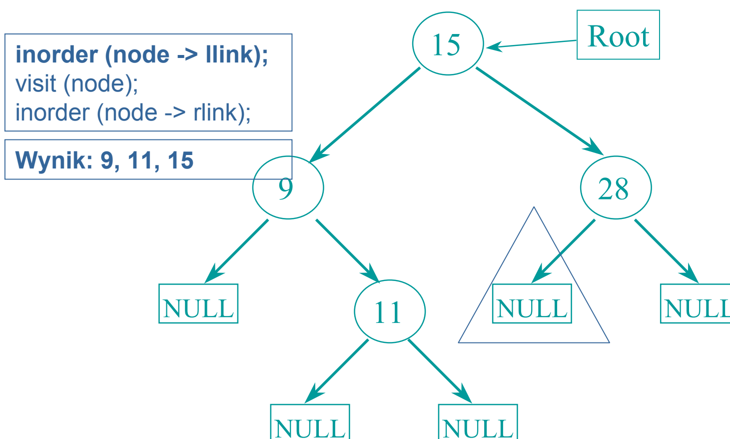
Dynamiczne realizacje struktur drzewiastych

- Algorytm ***przeglądanie*** drzewa ***binarnego w głąb*** (15):
  - Przykład dla procedury „inorder" – LVR



Dynamiczne realizacje struktur drzewiastych

- Algorytm ***przeglądanie*** drzewa ***binarnego w głąb*** (16):
  - Przykład dla procedury „inorder" – LVR

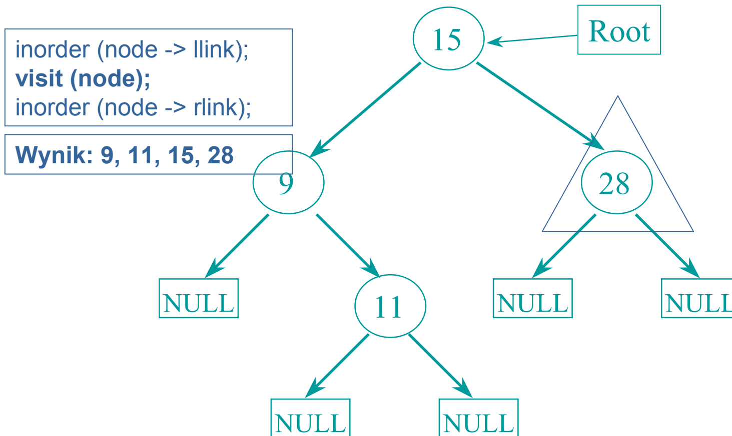


Dynamiczne realizacje struktur drzewiastych

- Algorytm ***przeglądanie*** drzewa ***binarnego w głąb*** (17):
  - Przykład dla procedury „inorder” – LVR

```
inorder (node -> llink);
visit (node);
inorder (node -> rlink);
```

Wynik: 9, 11, 15, 28

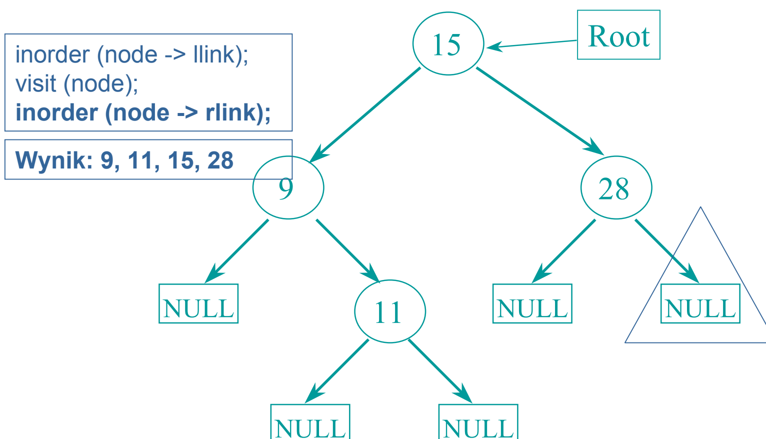


Dynamiczne realizacje struktur drzewiastych

- Algorytm ***przeglądanie*** drzewa ***binarnego w głąb*** (18):
  - Przykład dla procedury „inorder” – LVR

```
inorder (node -> llink);
visit (node);
inorder (node -> rlink);
```

Wynik: 9, 11, 15, 28



Dynamiczne realizacje struktur drzewiastych

- Algorytm ***dołączania*** elementu do drzewa ***binarnego*** (1):
  - Cel:
    - dodanie nowego elementu do drzewa;
  - Dane wejściowe:
    - Dowiązanie do korzenia drzewa 'Root';
    - Nowa dana elementarna;

Dynamiczne realizacje struktur drzewiastych

- Algorytm ***dołączania*** elementu do drzewa ***binarnego*** (2):
  - Utwórz element i ustal dane elementarne;
  - Znajdź miejsce wstawienia elementu w drzewie;
  - Wstaw element do drzewa:
    - Wstaw element jako pierwszy w drzewie;
    - (lub) Wstaw element we wskazane miejsce w drzewie;

Dynamiczne realizacje struktur drzewiastych

- Algorytm ***dołączania*** elementu do drzewa ***binarnego*** (3):

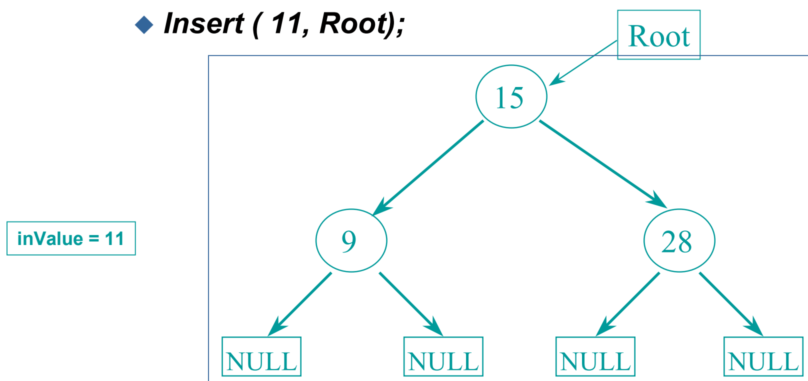
```
void Insert (int inValue, NodePtr &next) {
    if  (next == NULL ) {
        next = (Node *)malloc(sizeof(Node));
        next -> llink = NULL;
        next -> rlink = NULL;
        next -> data = inValue;
    }
    else if ( inValue  <   next -> data )
        Insert( inValue, next -> llink ) ;
    else if ( inValue  >   next -> data )
        Insert( inValue, next -> rlink );
}
```

rekurencja

Dynamiczne realizacje struktur drzewiastych

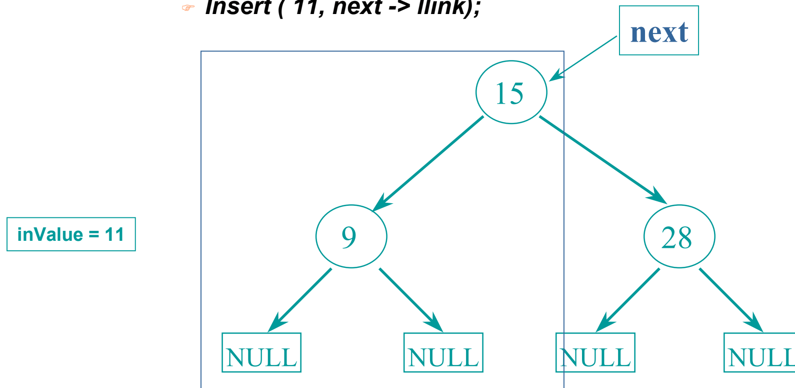
- Algorytm ***dołączania*** elementu do drzewa ***binarnego*** (4):
  - Wstawienie do drzewa elementu z wartością ’11’;
  - ***Insert (11, Root);***

inValue = 11



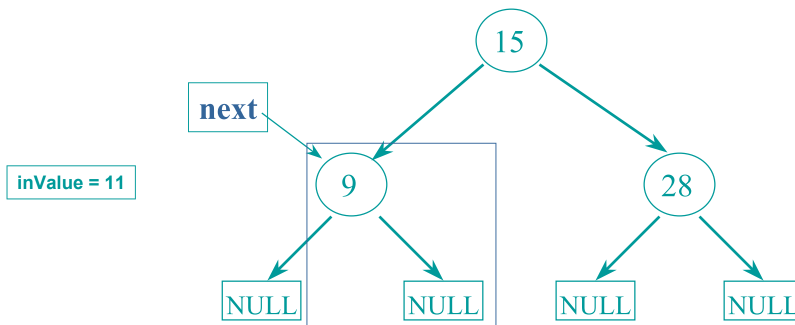
Dynamiczne realizacje struktur drzewiastych

- Algorytm ***dołączania*** elementu do drzewa ***binarnego*** (5):
  - Dopóki ***(inValue < next -> data)***
    - ***Insert (11, next -> llink);***



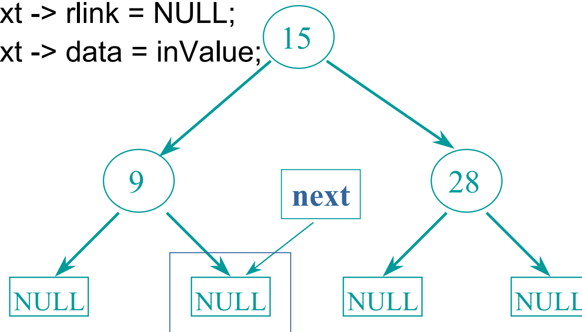
Dynamiczne realizacje struktur drzewiastych

- Algorytm ***dołączania*** elementu do drzewa ***binarnego*** (6):
  - Dopóki ***(inValue > next -> data)***
    - ***Insert (11, next -> rlink);***



Dynamiczne realizacje struktur drzewiastych

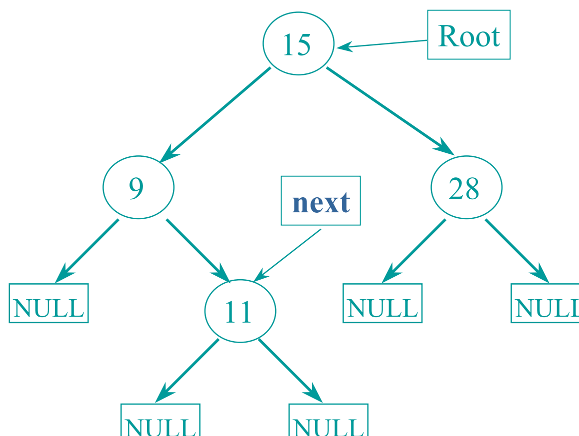
- Algorytm ***dołączania*** elementu do drzewa ***binarnego*** (7):
  - if (next == NULL) {
  - next = (Node *)malloc(sizeof(Node));
  - next -> llink = NULL;
  - next -> rlink = NULL;
  - next -> data = inValue;
  - }



inValue = 11

Dynamiczne realizacje struktur drzewiastych

- Algorytm ***dołączania*** elementu do drzewa ***binarnego*** (8):
  - Drzewo po wstawieniu elementu z wartością ’11’;



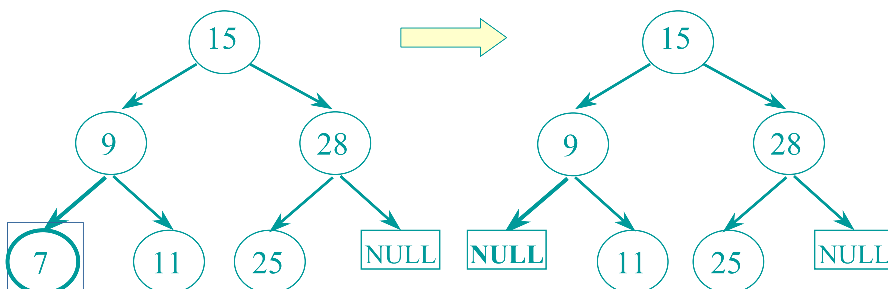
inValue = 11

Dynamiczne realizacje struktur drzewiastych

- Algorytm ***usuwania*** elementu z drzewa ***binarnego*** (1):
  - Cel:
    - Usunięcie węzła z drzewa;
  - Dane wejściowe:
    - Dowiązanie do korzenia drzewa 'Root';
    - Opis elementu usuwanego, np. wartość danej elementarnej;
  - Uwagi:
    - Przypadek 1: węzeł jest liściem;
    - Przypadek 2: węzeł ma jednego potomka;
    - Przypadek 3: węzeł ma dwóch potomków;

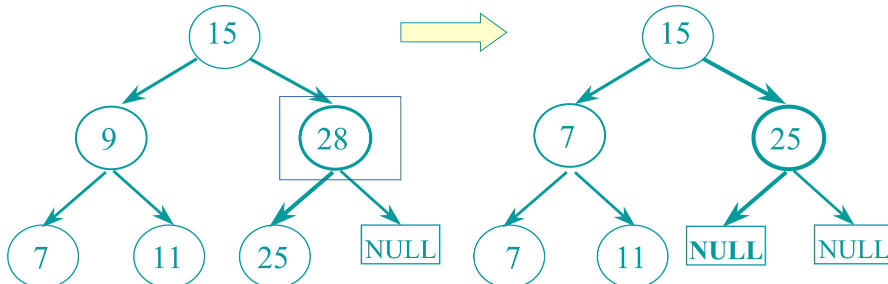
Dynamiczne realizacje struktur drzewiastych

- Algorytm ***usuwania*** elementu z drzewa ***binarnego*** (2):
  - Przypadek 1: węzeł jest liściem:
    - Znajdź element w drzewie;
    - Usuń węzeł z drzewa;



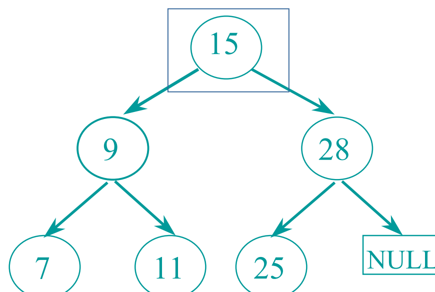
Dynamiczne realizacje struktur drzewiastych

- Algorytm ***usuwania*** elementu z drzewa ***binarnego*** (3):
  - Przypadek 2: węzeł ma jednego potomka:
    - Znajdź element w drzewie;
    - Usuń węzeł z drzewa;
    - Zastąp węzeł usunięty jego potomkiem (zmiana dowiązania w przodku węzła usuwanego)



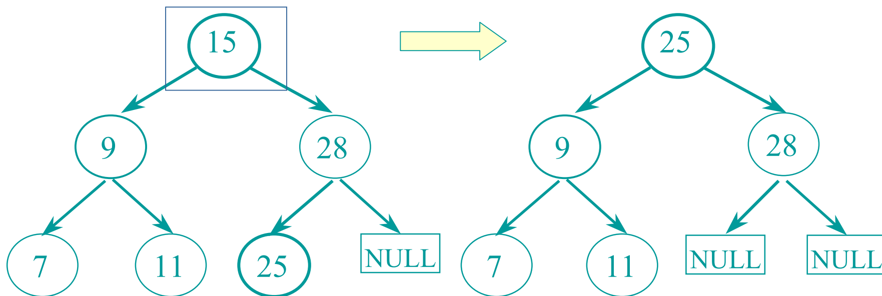
Dynamiczne realizacje struktur drzewiastych

- Algorytm ***usuwania*** elementu z drzewa ***binarnego*** (4):
  - Przypadek 3: węzeł ma dwóch potomków:
    - Znajdź element w drzewie;
    - Usuń węzeł z drzewa;
    - Zastąp węzeł usunięty: najmniejszym z prawego poddrzewa lub największym z lewego poddrzewa;



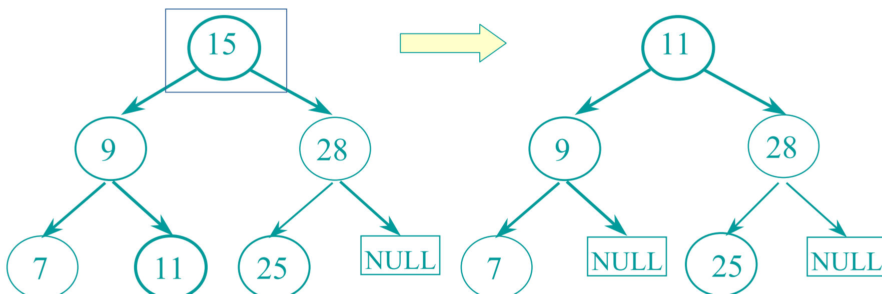
Dynamiczne realizacje struktur drzewiastych

- Algorytm ***usuwania*** elementu z drzewa ***binarnego*** (5):
  - Przypadek 3: węzeł ma dwóch potomków:
    - Wersja z przesunięciem najmniejszego elementu z prawego poddrzewa (skrajnie lewy wierzchołek);



Dynamiczne realizacje struktur drzewiastych

- Algorytm ***usuwania*** elementu z drzewa ***binarnego*** (6):
  - Przypadek 3: węzeł ma dwóch potomków:
    - Wersja z przesunięciem największego elementu z lewego poddrzewa (skrajnie prawy wierzchołek);



Dynamiczne realizacje struktur drzewiastych

- Algorytm ***usuwania*** elementu z drzewa ***binarnego*** (7):
  - Jeżeli w „Przypadku 3” przesuwany:
    - skrajnie prawy węzeł (największy element) z lewego poddrzewa posiada potomka lewego;
    - skrajnie lewy węzeł (najmniejszy element) z prawego poddrzewa posiada potomka prawego;
  - to należy zastosować dla węzła przesuwanego dodatkowo algorytm usuwania z „Przypadku 2” (usuwanie węzła z jednym potomkiem);
- Inne rozwiązanie dla operacji usuwania zobaczymy na wykładzie dotyczącym działań na rodzajach drzew binarnych;

Dziękuję za uwagę