Wykład 8
Abstrakcyjne typy danych. Struktury wiązane
Lista dwukierunkowa uporządkowana.
Drzewa. Drzewa poszukiwań binarnych: wstawianie liści, wstawianie do korzenia, obroty w węzłach, wyważanie drzewa, przejście przez drzewo
Złożoność obliczeniowa tablic, drzew i list

1. Lista dwukierunkowa uporządkowana.

Etap 1 - Opis ADT

Nazwa typu - Lista elementów

Własności typu: Potrafi przechować ciąg elementów o dowolnym rozmiarze

Dostępne działania: Inicjalizacja listy
Określenie, czy lista jest pusta
Dodanie elementu do listy,
Wyszukanie miejsca na liście przez większym lub równym elemencie na liście lub na końcu listy
Usuwanie z listy,

Etap 2 - Budowa interfejsu

```
typedef int dane;                          // dane umieszczone liście
typedef struct ELEMENTD* listanp;          //nazwa wskaźnika na element listy
struct ELEMENTD
 { dane Dane;
   listanp Nastepny, Poprzedni;
 };
struct listadw
 { listanp Poczatek;
   listanp Gdzie;
 };
```

Abstrakcyjny algorytm listy uporządkowanej zrealizowanej w postaci listy dwukierunkowej: lista → listadw

void Inicjalizacja(lista & Lista);
{*działanie:* inicjuje liste
warunki wstępne: Lista wskazuje na pierwszy element
warunki końcowe: Lista zostaje zainicjowana jako pusta}

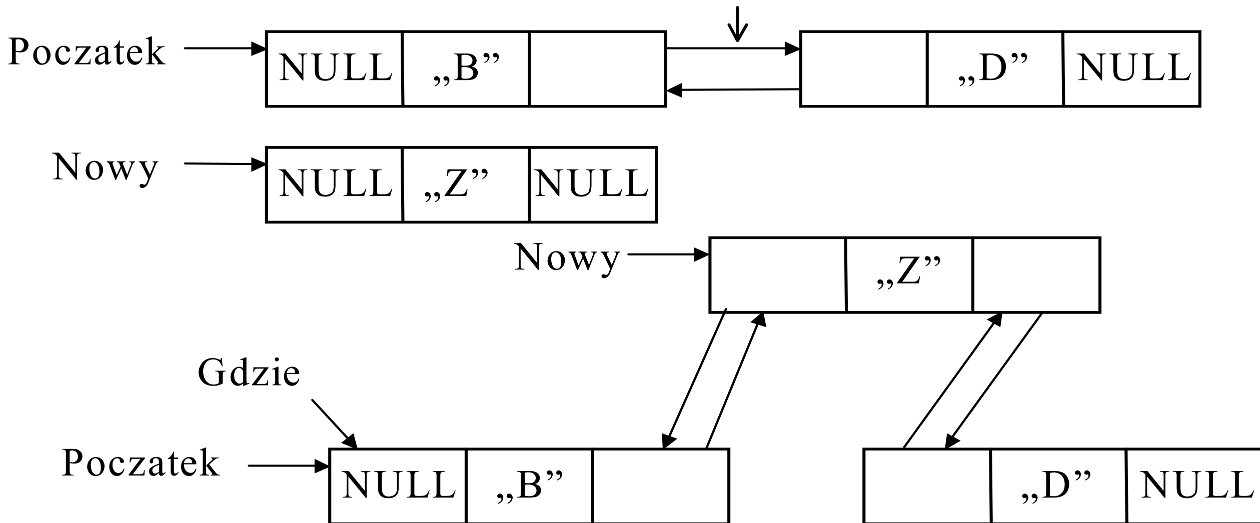
inline int Pusty(lista Lista) { **return** Lista.Poczatek==**NULL**;}
{*działanie*: określa, czy lista jest pusta; typ **inline**, bo często wywoływana
warunki wstępne: *Lista* jest zainicjowana,
warunki końcowe: funkcja zwraca 1, jeśli lista pusta, w przeciwnym wypadku 0}

int Szukaj(lista& Lista, dane Dana);
{ *działanie:* szuka elementu na liście
warunki początkowe: Lista wskazuje na zainicjowaną listę,
warunki końcowe: funkcja zwraca wskazanie *Lista.Gdzie* na element wskazujący na element o wartości większej (wtedy funkcja zwraca wartość równą 3) lub równej wartości *Dana* (wtedy funkcja zwraca wartość równą 2) lub na końcu listy, gdy nie znaleziono większego lub równego elementu na liście – wtedy funkcja zwraca wartość równą 1 i wartość *Lista.Gdzie* wskazuje na ostatni element. Jeśli znaleziono miejsce na początku listy (pierwszy element jest większy lub równy *Dana*), Lista.Gdzie jest równe NULL. Gdy lista jest pusta, wtedy funkcja zwraca 0.}

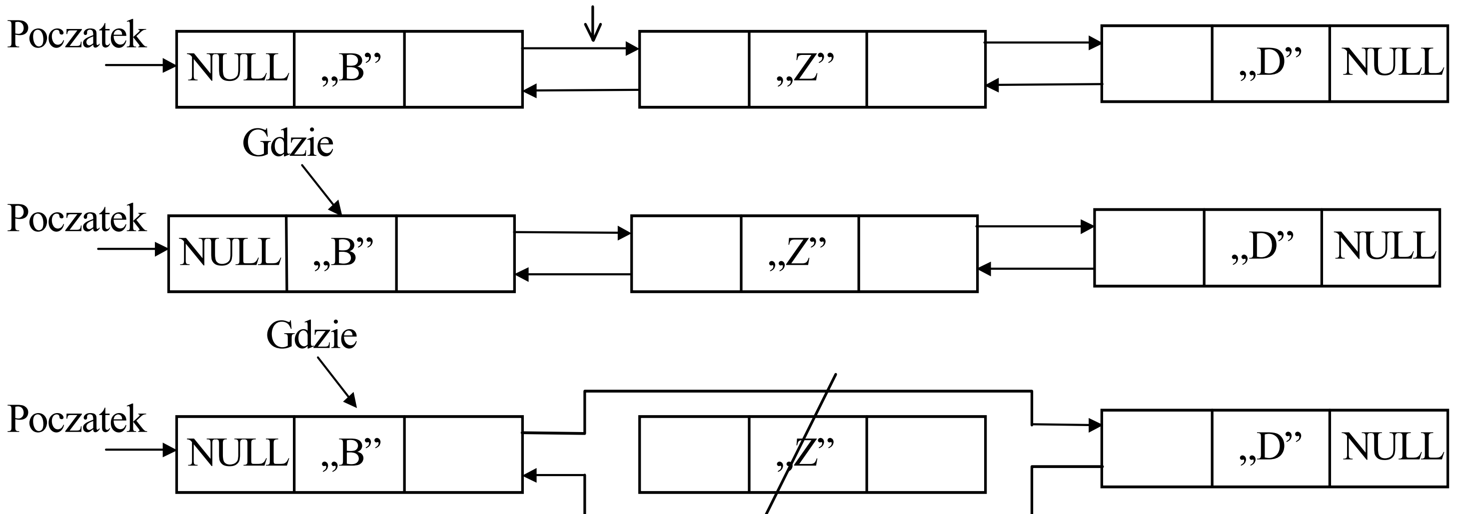
int Wstaw(lista& Lista, dane Dana);
{ *działanie:* dodaje element w dowolne miejsce ciągu umieszczonego na liście
warunki początkowe: Dane jest daną do wstawienia na miejscu pośrednio wskazywanym przez *Lista.Gdzie* zainicjowanej listy *Lista*.
warunki końcowe: funkcja dodaje daną *Dana* na miejscu określonym przez funkcję *Szukaj,* jeśli zwróci wartość 0 (wstawianie do pustej listy), lub wartość 1 (wstawianie na końcu listy) lub wartość 2 i 3 (wstawianie wewnątrz listy, odpowiednio przed równym lub większym elementem lub na początku listy) i funkcja *Wstaw* zwraca 1, w przeciwnym wypadku zwraca 0}

int Usun(lista& Lista);
{ *działanie:* usuwa element na dowolnym miejscu w ciągu wstawionym do listy
warunki początkowe: Lista jest zainicjowaną listą, *Lista.Gdzie* jest pośrednim wskazaniem na element usuwany określony przez funkcję *Szukaj,* gdy zwróci wartość 2
warunki końcowe: funkcja usuwa z listy element określony przez funkcję *Szukaj*, gdy zwróci ona wartość 2 oraz wraca dane umieszczone na usuwanym elemencie }

- Wyszukiwanie i wstawianie w dowolnym miejscu listy niepustej oraz jako pierwszy na pustej



- usuwanie elementu w dowolnym miejscu listy niepustej



Etap 3. Implementacja w postaci listy dwukierunkowej nieuporządkowanej

```
#include <conio.h>
#include <stdio.h>
  //1. interfejs ADT listy uporządkowanej
 typedef int dane;                               // dane umieszczone liście
 typedef struct ELEMENTD* listanp;        //nazwa wskaźnika na element listy

 struct ELEMENTD
  {  dane Dane;
    listanp Nastepny, Poprzedni;
  };
                //funkcje ADT listy
 struct listadw
  { listanp Poczatek;
    listanp Gdzie;
  };
 void Inicjalizacja(listadw& Lista);
 inline int Pusty(listadw Lista);
 int Szukaj(listadw& Lista, dane Klucz);     //przeciążenie nazw funkcji
 int Szukaj(listadw& Lista, long Miejsce);  // dzięki różnej liście parametrów
 int Wstaw(listadw& Lista, dane Dana);
 dane Usun(listadw& Lista);

        //2. funkcje we/wy dla danych umieszczonych na liście
 void Pokaz_dane (dane Dana);
 dane Dane(char* s);

        //3. funkcje ogólnego przeznaczenia
  void Komunikat(char*);
  char Menu(const int ile, char *Polecenia[]);

        //4. elementy programu
 const int Esc=27;
 const int POZ=4;
 char * Tab_menu[POZ] =
  { "1 : Wstawianie do posortowanej listy ",
    "2 : Usuwanie z posortowanej listy",
    "3 : Wydruk listy wraz z jej usuwaniem",
    " >Esc Koniec programu"};
```

//5. funkcje klienta korzystające ze listy

```
//typedef umozliwia funkcjom klienta na uniezaleznie sie
//od zmiany implementaci listy uporzadkowanej
//w stosunku do listy uporzadkowanej jednokierunkowej
typedef listadw lista;
void Wstaw_do_listy(lista& Lista);
void Usun_z_listy(lista& Lista);
void Wyswietl_usun_z_listy(lista& Lista);

void main(void)
{ lista Lista;
  char Wybor;
  clrscr();
  Inicjalizacja(Lista);
  do
   {   Wybor= Menu(POZ, Tab_menu);
       switch (Wybor)
         { case '1' : Wstaw_do_listy(Lista);                 break;
           case '2' : if (Pusty(Lista))
                        Komunikat("\nLista pusta\n");
                      else (Usun_z_listy(Lista));            break;
           case '3' : if (Pusty(Lista))
                          Komunikat("\nLista pusta\n") ;
                      else Wyswietl_usun_z_listy(Lista);     break;
    }
   } while (Wybor !=Esc );
  }
```

```
//*************funkcje klienta korzystające z listy***********
 void Wstaw_do_listy(lista& Lista)
  { dane  Dana= Dane("Podaj dane do wstawienia: ");
    Szukaj(Lista, Dana);
    if (!Wstaw(Lista, Dana))    Komunikat("\nBrak pamieci");
    else Komunikat("\nWstawiono do listy"); }

  void Usun_z_listy(lista& Lista)
   { dane  Dana= Dane("Podaj dane do usuniecia: ");
     if (Szukaj(Lista, Dana)==2)
      {Usun(Lista);
       Komunikat("\n Usunieto z listy");}
     else Komunikat("\nNie znaleziono danej");
    }

  void Wyswietl_usun_z_listy(lista& Lista)
    { dane d;
      while (!Pusty(Lista))
        {  Szukaj(Lista, 1L);
           d=Usun(Lista);
           Pokaz_dane(d);}
      }
//*******funkcje we/wy dla danych umieszczonych na liście*********
 dane Dane(char* s)
  { int a;
   do
    { fflush(stdin);
      printf("\n\n%s",s);
    } while (scanf("%d",&a)!=1);
   return a;
  }

 void Pokaz_dane(dane Dana)
  { printf("\nNumer: %d\n", Dana);
   printf("Nacisnij dowolny klawisz...\n"); getch();}

//********funkcje ogólnego przeznaczenia***********
char Menu(const int ile, char *Polecenia[])
 {  clrscr();
    for (int i=0; i<ile;i++) printf("\n%s",Polecenia[i]);
    return getch(); }

 void Komunikat(char* s)
   {   printf(s); getch();  }
```

```
//*********interfejs ADT listy************
 void Inicjalizacja(listadw& Lista)  { Lista.Poczatek = NULL;}

 inline int Pusty(listadw Lista)      { return Lista.Poczatek==NULL;  }

 int Szukaj(listadw& Lista, dane Klucz) //wyszukiwanie klucz na liście
  { int wynik=0;
    if (Pusty(Lista))
      { Lista.Gdzie = Lista.Poczatek;
        return 0;  }
   if (Lista.Gdzie==NULL)   Lista.Gdzie= Lista.Poczatek;
   if (Lista.Gdzie->Dane<Klucz)        //szukaj na prawo, jeśli klucz jest większy
    while (Lista.Gdzie->Dane< Klucz && Lista.Gdzie->Nastepny!=NULL)
         Lista.Gdzie = Lista.Gdzie->Nastepny;
   else
    if (Lista.Gdzie->Dane>Klucz)      //szukaj na lewo, jeśli klucz jest mniejszy
      while (Lista.Gdzie->Dane>Klucz && Lista.Gdzie->Poprzedni!=NULL)
          Lista.Gdzie = Lista.Gdzie->Poprzedni;
    //znaleziono w liście element mniejszy od klucza, Gdzie wskazuje na element mniejszy od klucza
   if (Lista.Gdzie->Dane < Klucz)   return 1;
         //znaleziono w liście element większy lub równy kluczowi
   if (Lista.Gdzie->Dane == Klucz)    wynik= 2;
   else    wynik= 3;
        //Gdzie powinien wskazywać element przed większym lub równym elementem
   Lista.Gdzie= Lista.Gdzie->Poprzedni;
   return wynik;}

 int Szukaj(listadw& Lista, long Miejsce)  //wyszukiwanie miejsca na liście
   { long  Numer= 1;
    Lista.Gdzie = Lista.Poczatek;
    if (Pusty(Lista)) return 0;
    while (Lista.Gdzie->Nastepny!=NULL && Miejsce != Numer)
     { Lista.Gdzie= Lista.Gdzie->Nastepny;
       Numer++; }
    if (Miejsce == Numer)
      { Lista.Gdzie= Lista.Gdzie->Poprzedni;
        return 2;}
    else
      if ( Miejsce == Numer+1)  return 1;
      else return 3;}
```

```
int Wstaw(listadw& Lista, dane Dana)
{ listanp Nowy = new ELEMENTD;
  if (Nowy !=NULL)    Nowy->Dane=Dana;
  else return 0;                             //nie wstawiono, za mało miejsca w pamięci
  if (Lista.Gdzie==NULL)                     //wstaw na początku listy
   { Nowy->Nastepny= Lista.Poczatek;         //podłącz z prawej na początku
      if (!Pusty(Lista))
          Lista.Poczatek->Poprzedni= Nowy;
      Lista.Poczatek= Nowy;                  //podłącz z lewej na początku
    }
  else                                       //wstaw wewnątrz listy lub na końcu
   {Nowy->Nastepny= Lista.Gdzie->Nastepny;//podłącz z prawej w środku lub na końcu
     if (Lista.Gdzie->Nastepny!=NULL)
         Lista.Gdzie->Nastepny->Poprzedni= Nowy;  //podłącz z prawej w środku
     Lista.Gdzie->Nastepny= Nowy;            //podłącz z lewej w środku lub na końcu
    }
     Nowy->Poprzedni= Lista.Gdzie;           //podłącz z lewej - zawsze
     return 1;
}

dane Usun(listadw& Lista)
{ listanp Pom=Lista.Gdzie;
  if (Lista.Gdzie==NULL)
    Lista.Gdzie= Lista.Poczatek;             //korekcja wskaźnika Gdzie
  else
    Lista.Gdzie= Lista.Gdzie->Nastepny;
  if (Lista.Gdzie->Poprzedni!=NULL)          //odłącz na lewo
    Lista.Gdzie->Poprzedni->Nastepny=Lista.Gdzie->Nastepny; //odłącz w środku
  else Lista.Poczatek= Lista.Gdzie->Nastepny;              //odłącz na początku
  if (Lista.Gdzie->Nastepny!=NULL)           //odłącz na prawo
    Lista.Gdzie->Nastepny->Poprzedni=Lista.Gdzie->Poprzedni;//odłącz w środku
  dane d=Lista.Gdzie->Dane;
  delete Lista.Gdzie;
  Lista.Gdzie=Pom;
  return d;
 }
```

2. Drzewa

1. Definicje drzew

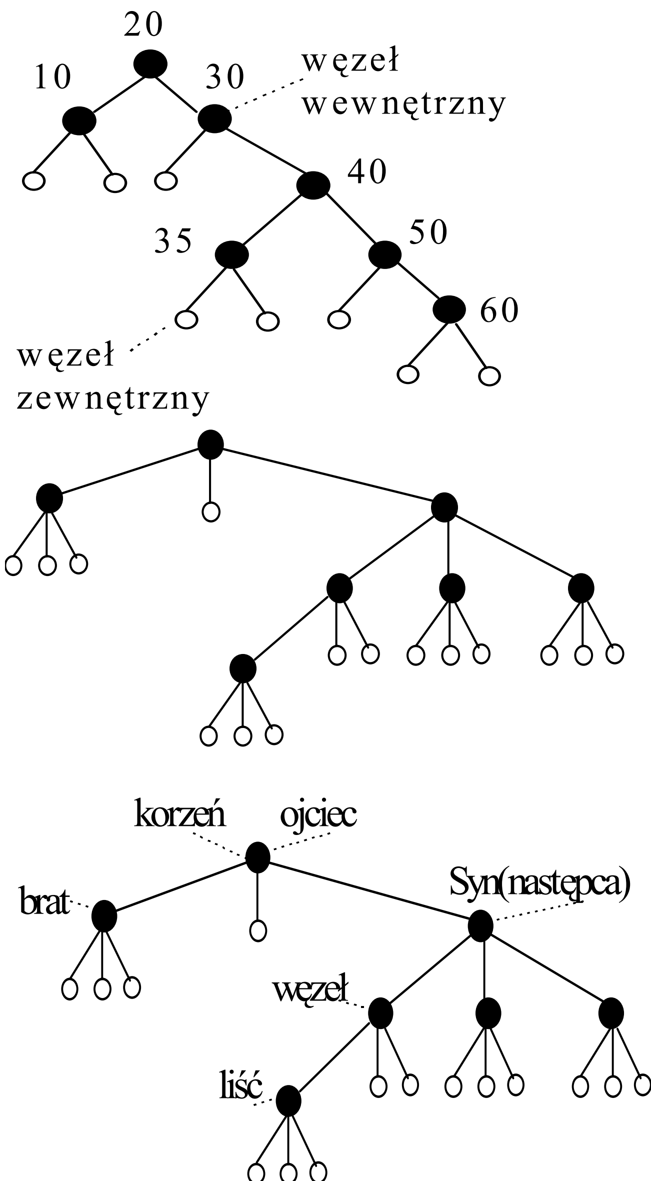
Definicja 1:

Drzewa to matematyczna abstrakcja, która umożliwia:

- opisywanie własności algorytmów (np algorytmy: dziel i zwyciężaj” , kopcowanie)
- opisywanie konkretnych struktur danych, które są realizacjami drzew (np. drzewa poszukiwań binarnych, drzewa czerwono-czarne, B-drzewa).

Klasyfikacja drzew:

- drzewa binarne i m-drzewa
- drzewa z korzeniem
- drzewa uporządkowane
- drzewa swobodne



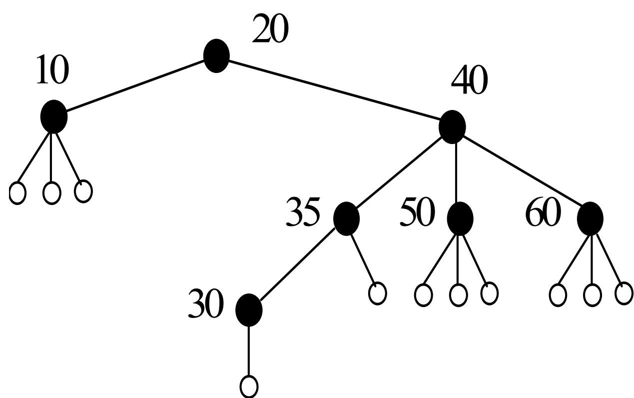
Definicja 2:

Drzewo binarne to węzeł zewnętrzny lub dołączony do pary drzew binarnych, które nazywa się odpowiednio lewym i prawym poddrzewem tego węzła.

Definicja 3:

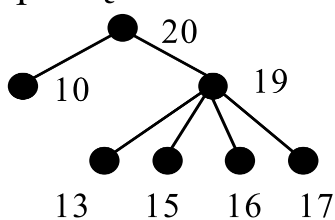
M-drzewo to węzeł zewnętrzny lub węzeł wewnętrzny dołączony do uporządkowanego ciągu drzew, które są również m-drzewami.

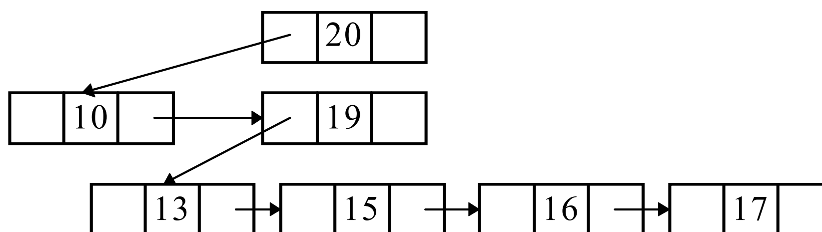
Definicja 4: *Drzewo z korzeniem* (lub drzewo nieuporządkowane) to węzeł (nazywany korzeniem) połączony z wielozbiorem drzew z korzeniem.
Taki wielozbiór nazywamy jest lasem nieuporządkowanym.
Drzewa nieuporządkowane są często reprezentowane w programach jako drzewa uporządkowane.

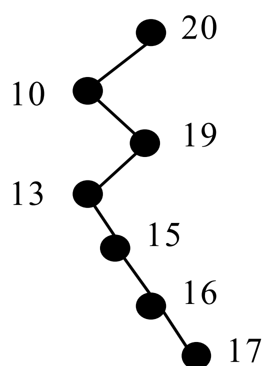


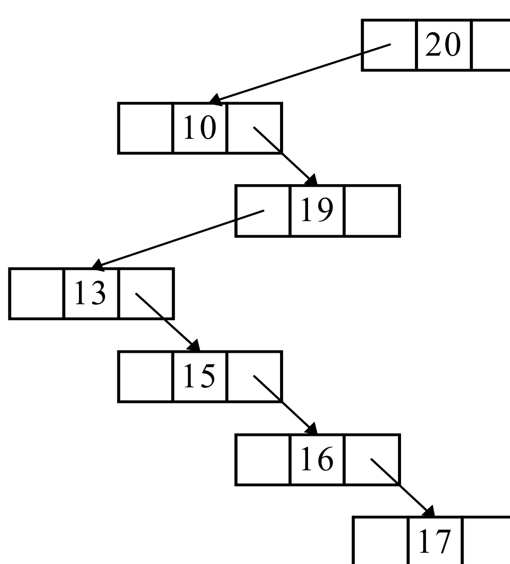
Definicja 5: *Drzewo uporządkowane* to węzeł (nazywany korzeniem) dołączony do ciągu rozłącznych drzew. Taki ciąg nazywany jest lasem. W przeciwieństwie do m-drzew może mieć dowolną liczbę synów (następców), lecz ustala się ich kolejność (porządek).

Definicja 6: Istnieje jednoznaczne *odwzorowanie* drzew binarnych na lasy uporządkowane i na odwrót.









Definicja 7: *Drzewo swobodne* czyli drzewo bez korzenia to graf, który zawiera *n-1* krawędzi i nie ma cykli, jest spójny, każde dwa wierzchołki łączy dokładnie jedna ścieżka prosta.

Graf jest parą zbiorów węzłów i krawędzi łączących po dwa różne węzły, przy czym każde dwa węzły łączy co najwyżej jedna krawędź.

Ścieżka prosta to ciąg krawędzi prowadzący od jednego węzła do innego, w którym nie powtarza się żaden węzeł dwukrotnie.

Graf jest *spójny*, jeśli jego dowolne dwa węzły można połączyć ścieżką prostą.

Ścieżka jest *cyklem* wtedy, gdy różni się od ścieżki prostej tylko tym, że pierwszy i ostatni węzeł to ten sam węzeł.

2. Drzewo binarne poszukiwań
2.1. Podstawowe definicje

- nazwa typu wskaźnika na funkcję dla funkcji przejścia przez drzewo

```
typedef  void(* zrob)(OSOBA&);
```

- definicja elementu drzewa

```
typedef struct ELEMENTD* PELEMENTD;
 struct OSOBA
   { int Numer;
    char Nazwisko[DL];
   };

struct ELEMENTD
   { OSOBA Dane;
     PELEMENTD Lewy, Prawy;
   };
```

2. 2. Budowa interfejsu

```
void Inicjalizacja (PELEMENTD &Wezel);
```

{ *działanie:* inicjuje drzewo
warunki wstępne: Wezel wskazuje na pierwszy element, zwany korzeniem
warunki końcowe: drzewo zostaje zainicjowane jako puste}

```
PELEMENTD Szukaj(PELEMENTD Wezel, char* Klucz);
```

{ *działanie:* szuka elementu w drzewie
warunki początkowe: Wezel wskazuje na zainicjowane drzewo, *Klucz* jest poszukiwanym elementem,
warunki końcowe: jeśli to możliwe, funkcja szuka elementu w drzewie równego *Kluczowi* idąc na lewo każdego z węzłów, jeśli element w węźle jest większy i na prawo, jeśli element w węźle jest mniejszy, natomiast jeśli znajdzie węzeł równy *Kluczowi,* zwraca wskazanie na ten element drzewa, w przeciwnym wypadku zwraca adres pusty}

int Wstaw(PELEMENTD &Wezel, PELEMENTD Pozycja);
{*działanie:* dodaje element do drzewa
warunki początkowe: Pozycja jest elementem z danymi wstawianym do zainicjowanego drzewa, *Wezel* wskazuje na korzeń drzewa poszukiwań binarnych
warunki końcowe: jeśli to możliwe, funkcja dodaje element *Pozycja* do drzewa idąc na lewo każdego z węzłów, jeśli dana elementu w węźle jest większa od danej w *Pozycja* i na prawo, jeśli dana elementu w węźle jest mniejsza aż do osiągnięcia węzła z wolnym łączem *Wezel*, który po wstawieniu wskazuje na element *Pozycja* i zwraca wynik równy 0, natomiast jeśli znajdzie węzeł wskazywany przez *Wezel* z równym elementem, to kończy poszukiwania, usuwa element *Pozycja* i zwraca wynik równy 1}

void Us(PELEMENTD &Gdzie, PELEMENTD &Usuwany);
{*działanie:* usuwa element z drzewa
warunki początkowe: Gdzie jest węzłem początkowym będącym lewym następcą elementu *Usuwany*, *Usuwany* jest węzłem z usuwanymi danymi
warunki końcowe: jeśli jest to możliwe, funkcja szuka elementu największego idąc w prawo od węzła *Gdzie*, następnie wstawia kopię wartości znalezionego elementu do usuwanego węzła *Usuwany* i ustawia w *Usuwany* wskazanie na element *Gdzie* ze znalezionym elementem oraz podłącza lewego następcę znalezionego elementu *Gdzie* na jego miejsce w drzewie}

int Usun(PELEMENTD &Wezel, **char*** Klucz);
{*działanie:* usuwa element z drzewa
warunki początkowe: Wezel jest zainicjowanym drzewem, *Klucz* zawiera dane poszukiwane w drzewie
warunki końcowe: jeśli jest to możliwe, funkcja szuka elementu w drzewie równego *Kluczowi* idąc na lewo każdego z węzłów, jeśli element w węźle jest większy i na prawo, jeśli element w węźle jest mniejszy, natomiast jeśli znajdzie węzeł równy *Kluczowi*, to jeśli ma on tylko jednego następcę (prawego lub lewego), zostaje nim zastąpiony, następnie usunięty i zwraca wynik równy 0, a w przeciwnym przypadku (węzeł równy *Kluczowi* nie jest liściem) funkcja szuka elementu największego za pomocą funkcji *Us,* przekazując jej lewego następcę usuwanego węzła. Funkcja *Us* wyszukuje największy element drzewa, idąc na prawo, który może zastąpić element usuwany; po znalezieniu kopiuje jego wartość do elementu usuwanego, następnie podłącza do drzewa w miejsce skopiowanego elementu jego lewego następcę i przekazuje wskazanie na ten skopiowany element do funkcji *Usun.* Funkcja *Usun* usuwa ten element i zwraca wynik równy 0. W przypadku, kiedy nie znaleziono elementu, funkcja zwraca wynik równy 1}

void Usun_pamiec (PELEMENTD Wezel);
{ *działanie:* usuwa elementy z drzewa
warunki początkowe: *Wezel* jest zainicjowanym drzewem
warunki końcowe: funkcja przechodzi przez wszystkie gałęzie aż do osiągnięcia liści i usuwa je z pamięci zaczynając zawsze od lewego liścia. Każdy węzeł z usuniętymi liśćmi staje się liściem. Po zakończeniu usuwania liczba elementów w drzewie jest równa 0}

void Usun_drzewo**(**PELEMENTD &Wezel**);**
{ *działanie:* usuwa elementy z drzewa i inicjuje drzewo jako puste
warunki początkowe: *Wezel* jest zainicjowanym drzewem
warunki końcowe: po wykonaniu funkcji *Usun_pamiec,* przekazując jej wskazanie *Wezel,* inicjuje wskazanie *Wezel* jako puste }

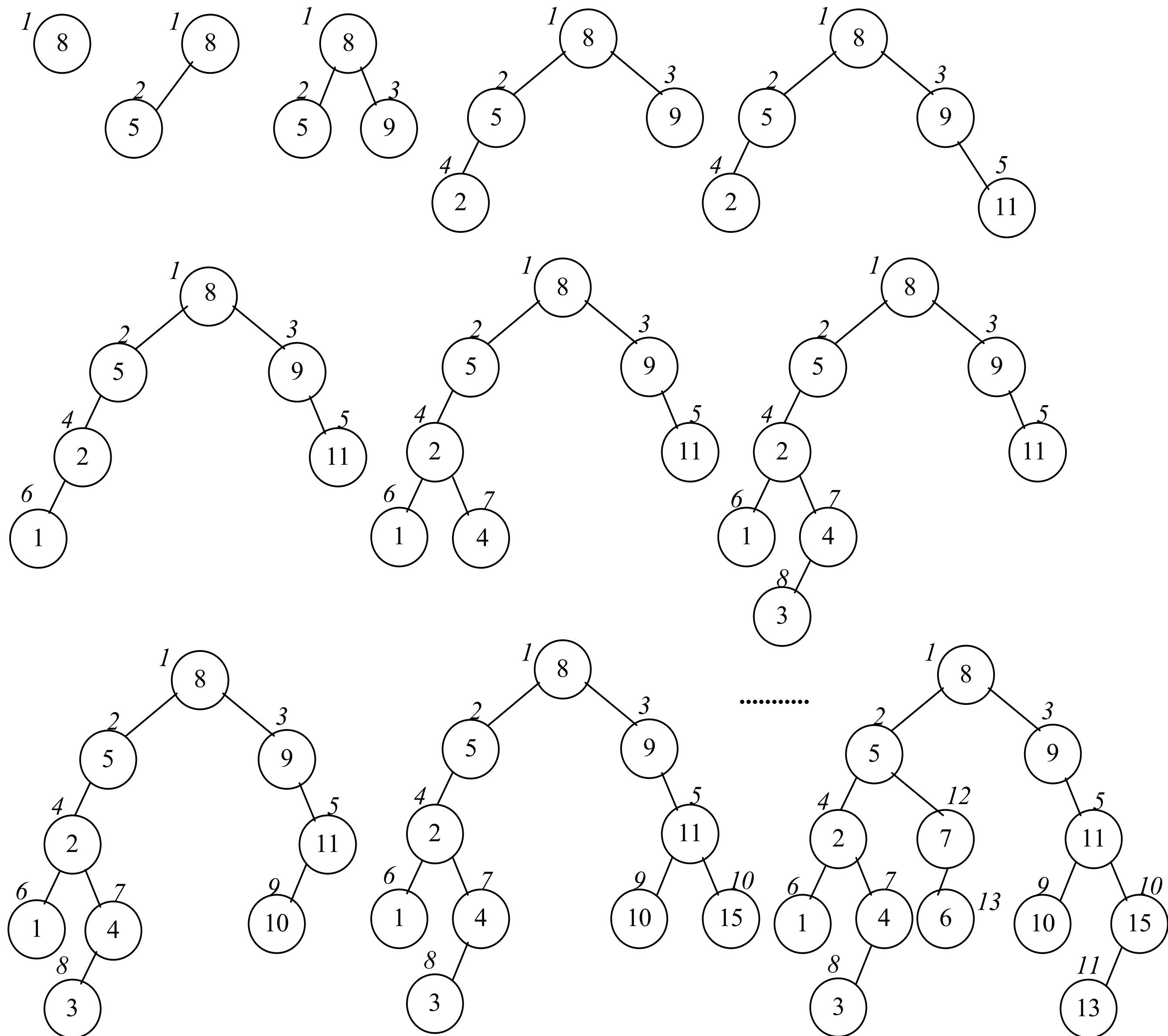
void Dla_kazdego (PELEMENTD Wezel, zrob funkcja);
{*działanie:* wykonuje funkcje na każdym wstawionym elemencie do drzewa
warunki początkowe: *Węzeł* jest zainicjowanym drzewem, *zrob* jest typem funkcji, która pobiera element drzewa i nie zwraca wartości
warunki końcowe: jeśli jest to możliwe, funkcja typu *zrób* jest wykonywana tylko raz dla każdego elementu wstawionego do drzewa zaczynając od najmniejszego elementu }

void Dla_jednego (PELEMENTD Wezel, **char*** Klucz, zrob funkcja);
{ *działanie*: wykonuje *funkcja* na elemencie wyszukanym w drzewie
warunki początkowe: *Wezel* jest zainicjowanym drzewem, *zrob* jest typem funkcji, która pobiera element z drzewa i nie zwraca wartości
warunki końcowe: funkcja typu *zrób* jest wykonywana tylko raz dla elementu z drzewa *Wezel* o wartości *Klucza*, jeśli zostanie wyszukany przez funkcję *Szukaj*}

2.3. Implementacja podstawowych funkcji

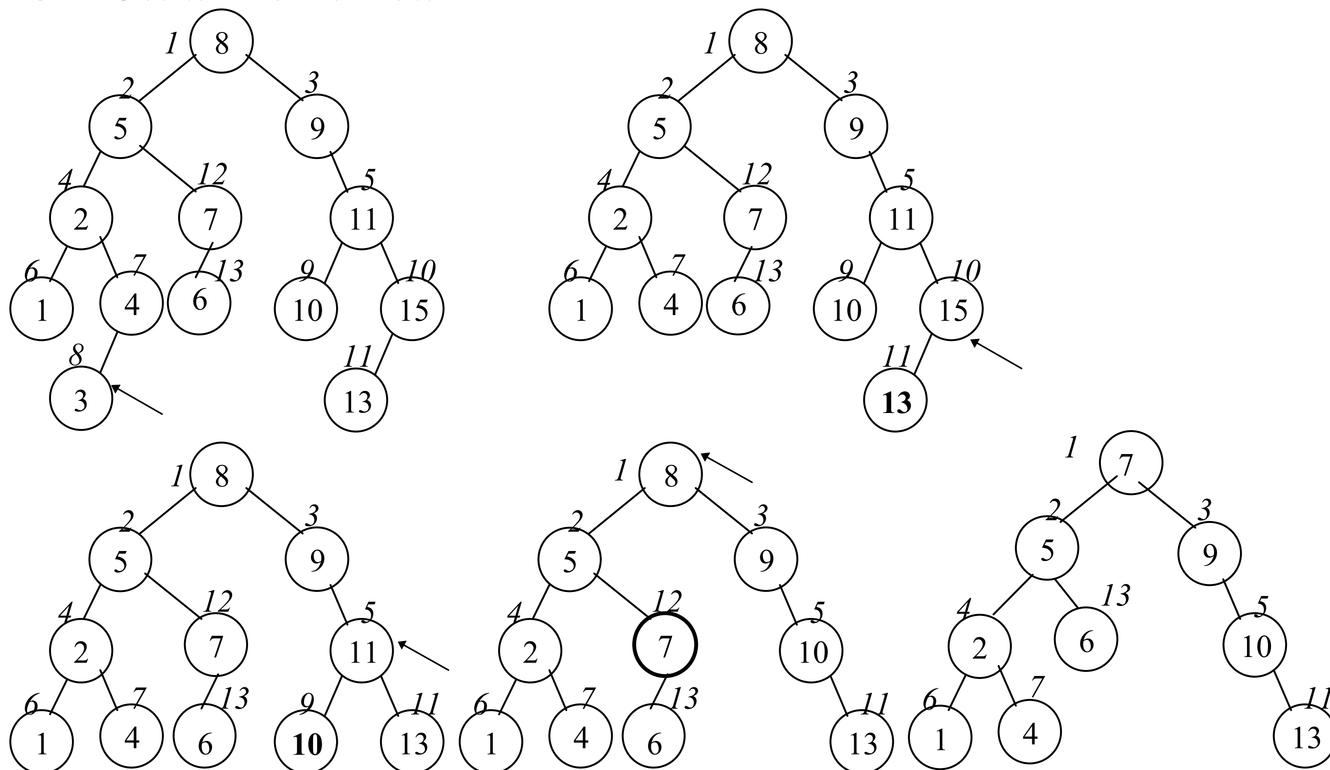
2.3.1. Wstawianie do drzewa

Należy wstawić do drzewa ciąg: 8, 5, 9, 2, 11, 1, 4, 3, 10, 15, 13, 7 , 6



```
int Wstaw(PELEMENTD &Wezel, PELEMENTD Pozycja)
  { if (Wezel== NULL) {Wezel= Pozycja; return 0;}
     if (strcmp(Pozycja->Dane.Nazwisko, Wezel->Dane.Nazwisko)<0)
         return Wstaw(Wezel->Lewy, Pozycja);
     else
      if (strcmp(Pozycja->Dane.Nazwisko, Wezel->Dane.Nazwisko)>0)
        return Wstaw(Wezel->Prawy, Pozycja);
      else
       { delete Pozycja;
        return 1;}
    }
```

2.3.2. Usuwanie z drzewa

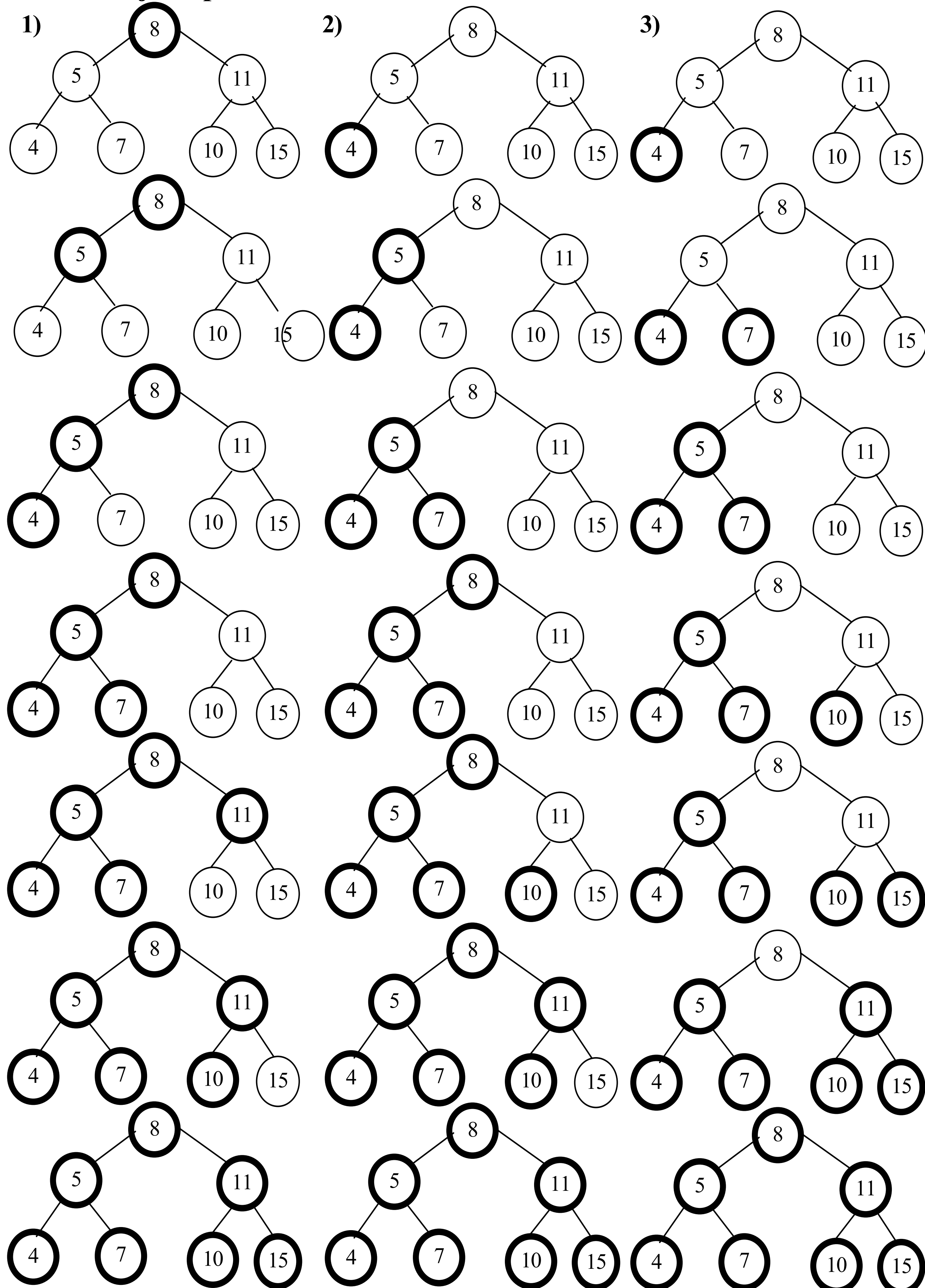


```
void Us(PELEMENTD &Gdzie,PELEMENTD &Usuwany)
  { if (Gdzie->Prawy != NULL) Us(Gdzie->Prawy, Usuwany); //szukanie liścia
   else                                                  //wymiana danych usuwanych z liściem
     { strcpy(Usuwany->Dane.Nazwisko, Gdzie->Dane.Nazwisko);
      Usuwany= Gdzie;
      Gdzie= Gdzie->Lewy;} }

 int Usun(PELEMENTD &Wezel, char *Klucz)
  { PELEMENTD Usuwany;
   if (Wezel != NULL)
     if (strcmp(Klucz, Wezel->Dane.Nazwisko)<0)
       return Usun(Wezel->Lewy, Klucz);
     else
       if (strcmp(Klucz, Wezel->Dane.Nazwisko)>0)
         return Usun(Wezel->Prawy, Klucz);
        else                                     // znaleziono element do usunięcia
         { Usuwany= Wezel;
           if (Usuwany->Prawy == NULL) Wezel= Usuwany->Lewy;
           else if (Usuwany->Lewy == NULL) Wezel= Usuwany->Prawy;
                   else Us(Usuwany->Lewy, Usuwany);
           delete Usuwany;
          return 0;
         }
     return 1;  }
```

2.3.3. Przeszukiwanie w drzewie

```
PELEMENTD Szukaj(PELEMENTD Wezel, char *Klucz)
  {
   if (Wezel == NULL) return NULL;
   if (strcmp(Klucz, Wezel->Dane.Nazwisko)== 0) return Wezel;
   else
    if (strcmp(Klucz, Wezel->Dane.Nazwisko) < 0)
      return Szukaj(Wezel->Lewy, Klucz);
    else
      return Szukaj(Wezel->Prawy, Klucz);
  }

void Dla_jednego (PELEMENTD Wezel, char* Klucz, zrob funkcja)
  {
   PELEMENTD Gdzie;
   Gdzie= Szukaj(Wezel, Klucz);
   if (Gdzie!= NULL) funkcja(Gdzie->Dane);
   }
```

2.3.4. Przejście przez drzewo



1) Przejście przedrostkowe przez drzewo

```
 void Dla_kazdego (PELEMENTD Wezel, zrob funkcja)
   {
    if (Wezel!= NULL)
      { funkcja(Wezel->Dane);
       Dla_kazdego(Wezel->Lewy, funkcja);
        Dla_kazdego(Wezel->Prawy, funkcja);  }}
```

2) Przejście uporządkowane przez drzewo -np. wyświetlenie posortowanych danych

```
  void Dla_kazdego (PELEMENTD Wezel, zrob funkcja)
   {
    if (Wezel!= NULL)
      {Dla_kazdego(Wezel->Lewy, funkcja);
       funkcja(Wezel->Dane);
       Dla_kazdego(Wezel->Prawy, funkcja);}}
```

3) Przejście przyrostkowe przez drzewo

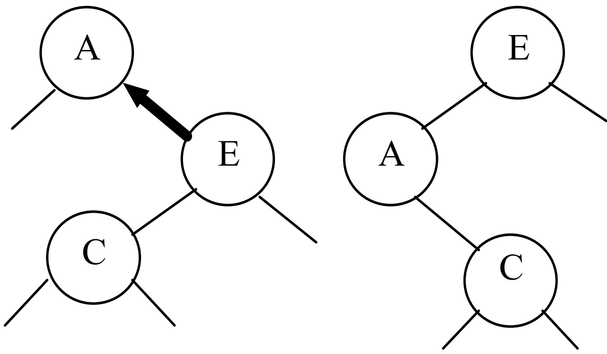
```
 void Dla_kazdego (PELEMENTD Wezel, zrob funkcja)
   {
    if (Wezel!= NULL)
      {Dla_kazdego(Wezel->Lewy, funkcja);
       Dla_kazdego(Wezel->Prawy, funkcja);
       funkcja(Wezel->Dane);} }
```

Przykład - usunięcie drzewa

```
void Usun_pamiec(PELEMENTD Wezel)
 {
   if (Wezel != NULL)
    {    Usun_pamiec((Wezel)->Lewy);
         Usun_pamiec((Wezel)->Prawy);
         delete Wezel;   }}

void Usun_drzewo(PELEMENTD &Wezel)
 { Usun_pamiec(Wezel);
   Wezel=NULL;
  }
```

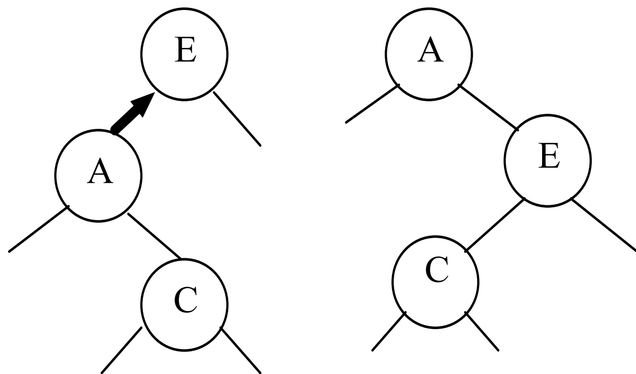
2.3.5. Obroty w węzłach

- obrót w lewo




```
void Obrot_L(PELEMENTD& Wezel)
{   PELEMENTD P;
    P = Wezel->Prawy;
    Wezel->Prawy = P->Lewy;
    P->Lewy = Wezel;
    Wezel = P;
}
```

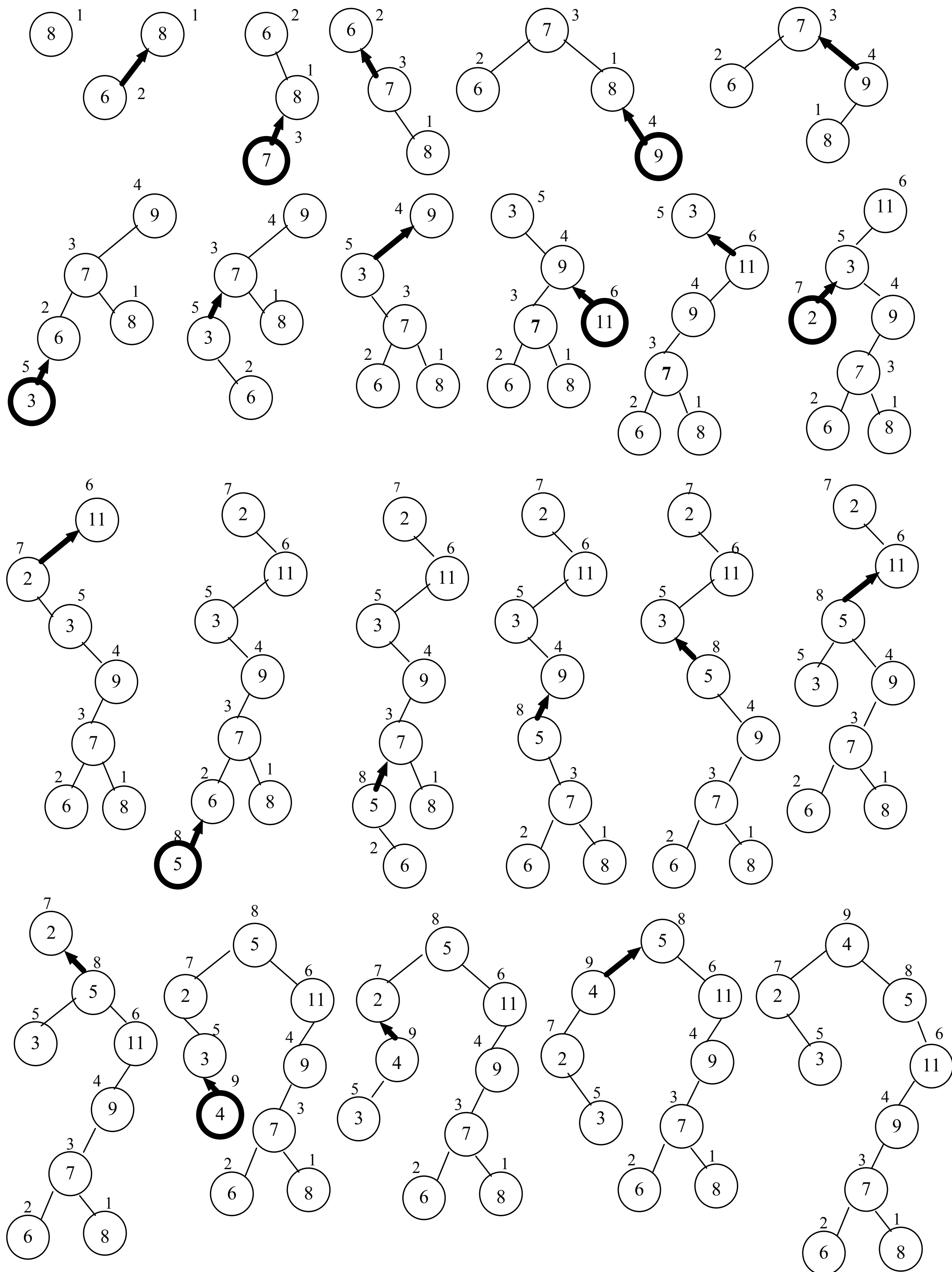
- obrót w prawo

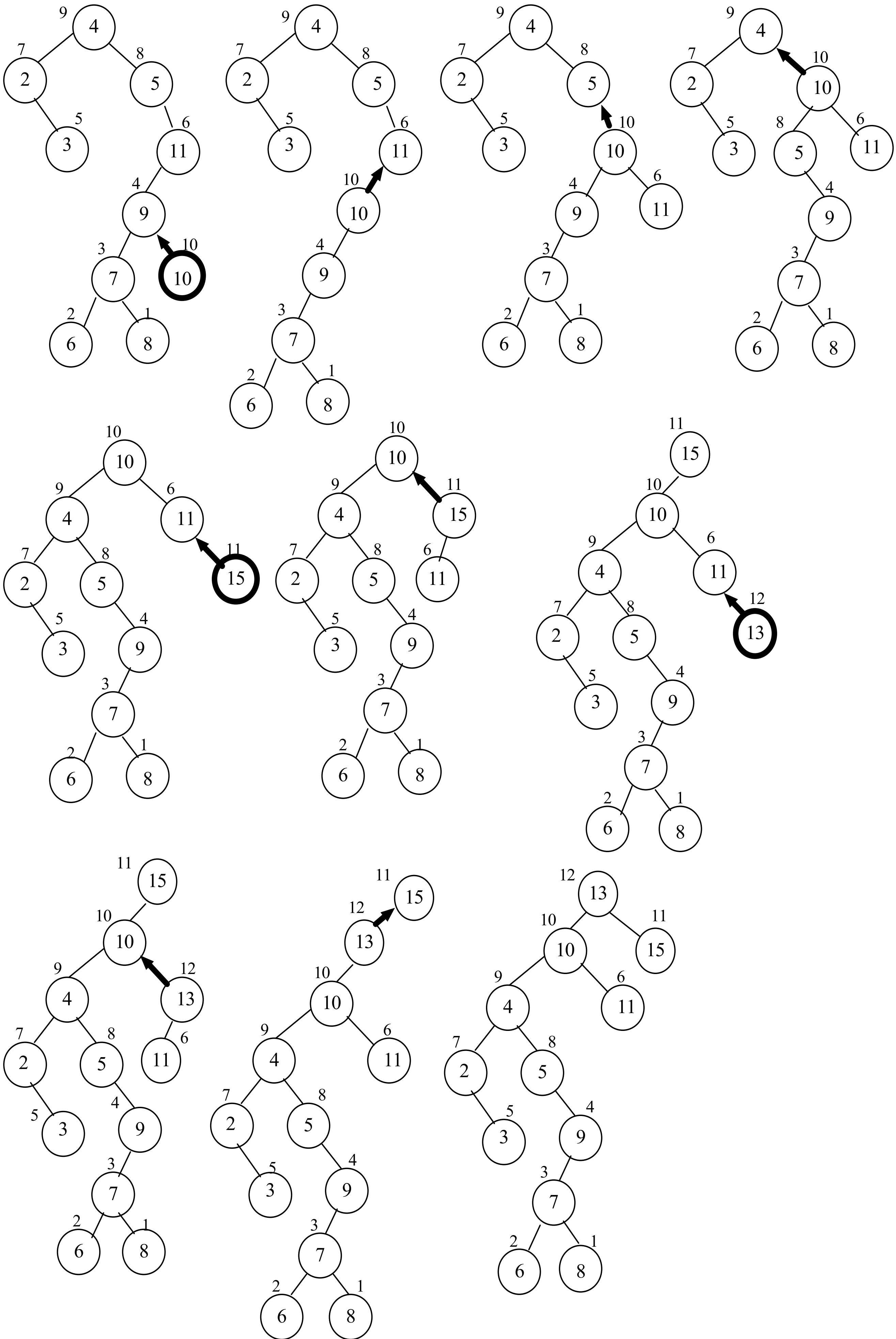



```
void Obrot_P(PELEMENTD& Wezel)
 { PELEMENTD P;
   P = Wezel->Lewy;
   Wezel->Lewy = P->Prawy;
   P->Prawy = Wezel;
   Wezel = P;
•   }
```

2.3.6. Wstawianie do korzenia drzewa

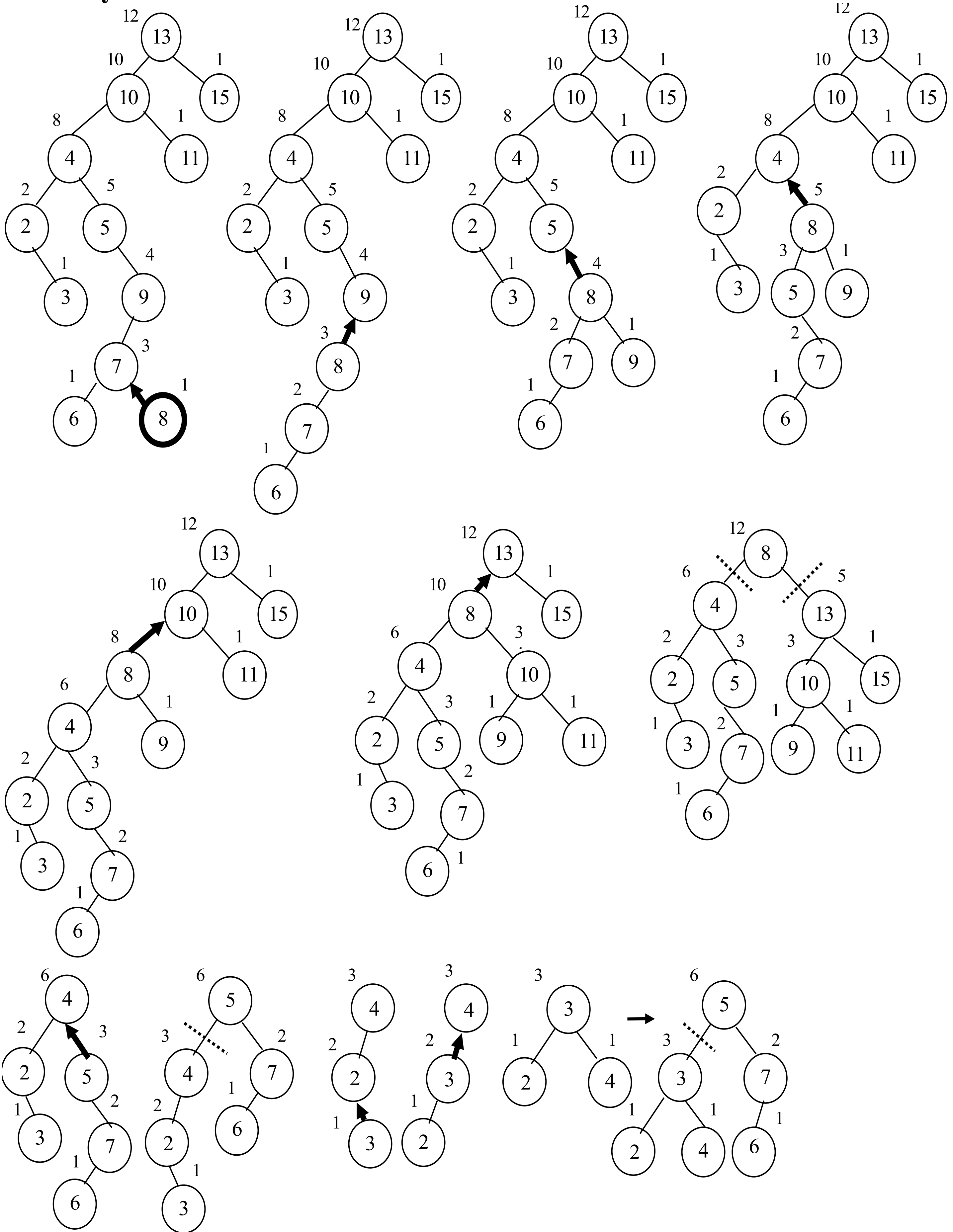
- Wstaw następujący ciąg: 8, 6, 7, 9, 3, 11, 2, 5, 4, 10, 15, 13 wstawiając do korzenia drzewa

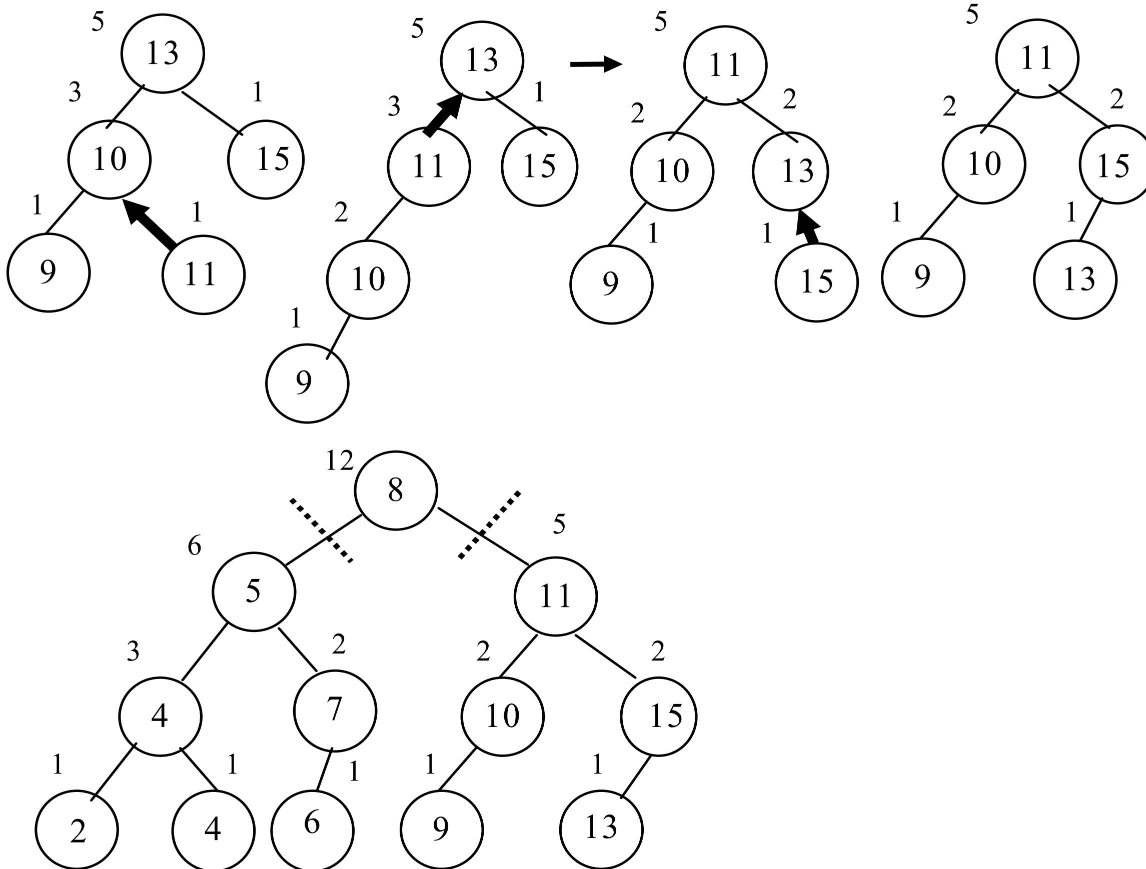




```
void Wstaw_korzen(PELEMENTD& Wezel,PELEMENTD Pozycja)
 {
   if (Wezel == NULL)
       Wezel = Pozycja;
   else
    if (strcmp(Pozycja->Dane.Nazwisko,Wezel->Dane.Nazwisko)<0)
     {
        Wstaw_korzen(Wezel->Lewy, Pozycja);
        Obrot_P(Wezel);
     }
    else
      if (strcmp(Pozycja->Dane.Nazwisko,Wezel->Dane.Nazwisko)>0)
       {
         Wstaw_korzen(Wezel->Prawy, Pozycja);
         Obrot_L(Wezel);
        }
     else
       {
        delete Pozycja;
        Pozycja=NULL;
       }
 }
```

2.3.7. Wyważanie drzewa






```
int Wywaz_drzewo( PELEMENTD& Wezel)
 {   if (Wezel == NULL || Wezel->Licznik == 1) return 0;
      Podzial(Wezel,Wezel->Licznik / 2);
      Wywaz_drzewo(Wezel->Lewy);
      Wywaz_drzewo(Wezel->Prawy);
      return 1; }

void Podzial( PELEMENTD& Wezel, int Liczba)
 { int il;
   if ( Wezel->Lewy == NULL) il =0;
   else il= Wezel->Lewy->Licznik;
   if (il > Liczba)
     { Podzial(Wezel->Lewy, Liczba);
       Obrot_P(Wezel );
       Oblicz_wezly(Wezel); }
   else
    if (il < Liczba)
     {  Podzial(Wezel->Prawy, Liczba-il-1);
        Obrot_L(Wezel);
        Oblicz_wezly(Wezel);}
  }
```

```
int Oblicz_wezly(PELEMENTD Wezel)
 { if (Wezel != NULL)
    { if (Wezel->Prawy == NULL && Wezel->Lewy == NULL) Wezel->Licznik=1;
      else   Wezel->Licznik= Oblicz_wezly(Wezel->Lewy)+
                             Oblicz_wezly(Wezel->Prawy)+1;
      return Wezel->Licznik; }
   else return 0; }

void Obrot_LL(PELEMENTD& Wezel)
 { PELEMENTD P;
   long x1,x2,x3;
   x1=x2=x3=0L;
   if (Wezel->Lewy != NULL)              x1= Wezel->Lewy->Licznik;
   if (Wezel->Prawy->Lewy !=NULL)     x2= Wezel->Prawy->Lewy->Licznik;
   if (Wezel->Prawy->Prawy != NULL) x3= Wezel->Prawy->Prawy->Licznik;
   P = Wezel->Prawy;   Wezel->Prawy = P->Lewy;       P->Lewy = Wezel;
   Wezel = P;
   Wezel->Lewy->Licznik= x1+x2;
   Wezel->Licznik= Wezel->Lewy->Licznik+x3;}

void Obrot_PP(PELEMENTD& Wezel)
 { PELEMENTD P;
   long x1,x2,x3;
   x1=x2=x3=0L;
   if (Wezel->Prawy != NULL)             x1= Wezel->Prawy->Licznik;
   if (Wezel->Lewy->Prawy != NULL)   x2= Wezel->Lewy->Prawy->Licznik;
   if (Wezel->Lewy->Lewy != NULL) x3= Wezel->Lewy->Lewy->Licznik;
   P = Wezel->Lewy;       Wezel->Lewy = P->Prawy;       P->Prawy = Wezel;
   Wezel = P;
   Wezel->Prawy->Licznik= x1+x2;
   Wezel->Licznik= Wezel->Prawy->Licznik+x3;}

 void Podzial( PELEMENTD& Wezel, int Liczba)
 {  int il;
    if ( Wezel->Lewy == NULL) il =0;
    else il= Wezel->Lewy->Licznik;
    if (il > Liczba)
      {    Podzial(Wezel->Lewy, Liczba);
           Obrot_PP(Wezel);  }          //zamiast Obrot_P(Wezel);
                                        //Oblicz_wezly(Wezel);
   else
     if (il < Liczba)
      { Podzial(Wezel->Prawy, Liczba-il-1);
        Obrot_LL(Wezel); }             //zamiast Obrot_L(Wezel);
 }                                       //Oblicz_wezly(Wezel);
```

```
#include <conio.h>
#include <string.h>
#include <stdio.h>
#include "mdrzewow.h"
#include "dodatki.h"
#include "we_wy.h"
 char *Polecenia[]={  "1 : Wstawianie do korzenia drzewa ",
                      "2 : Wstawianie jako lisc drzewa ",
                        "3 : Usuwanie z drzewa",
                       "4 : Wyswietlenie elementu drzewa",
                        "5 : Wywazanie drzewa",
                        "6 : Wydruk drzewa",
                        "7 : Usun drzewo",
                        "Esc - Koniec programu"};

 void Podaj_klucz(char *Klucz);
 void Wstaw_(PELEMENTD &Korzen, int jak);
 void Usun_(PELEMENTD &Korzen);
 void Wywaz_drzewo_(PELEMENTD &Korzen);
 void Dla_jednego_(PELEMENTD &Korzen);

void main(void)
 { char Co;
   PELEMENTD Korzen;
  Inicjalizacja(Korzen);
   do
     { Co = Menu(8,Polecenia);
       switch(Co)
        {
          case '1'   : Wstaw_(Korzen, 0); break;
          case '2'   : Wstaw_(Korzen, 1); break;
          case '3'   : Usun_(Korzen); break;
          case '4'   : Dla_jednego_(Korzen); break;
          case '5'   : Wywaz_drzewo_(Korzen); break;
          case '6'   : Dla_kazdego(Korzen, Pokaz_dane); break;
          case '7'   : Usun_drzewo(Korzen);break;
          case 27  : Komunikat("\nKoniec programu");break ;
          default   :  Komunikat("\nZla opcja");
       }
    } while (Co!=27);
 }
```

```
void Podaj_klucz(char* Klucz)
  {  char bufor[DL+2];     bufor[0]=DL;
     printf("\nPodaj klucz: ");
     strcpy(Klucz,cgets(bufor)); }

 void Wstaw_(PELEMENTD &Korzen, int jak)
  { OSOBA Dana; PELEMENTD Nowy;
    Dana= Dane();
    if ((Nowy= Nowy_element(Dana)) == NULL)
         Komunikat("\nBrak pamieci");
    else
       {   if (jak==0) Wstaw_korzen(Korzen, Nowy);
           else Wstaw(Korzen, Nowy);
       } }

 void Usun_(PELEMENTD &Korzen)
  { char Klucz[DL];
    if (Korzen==NULL)
      { Komunikat("\nDrzewo puste");    return; }
   Podaj_klucz(Klucz);
   if  (Usun(Korzen,Klucz))  Komunikat("\nNie znaleziono elementu");
   else                      Komunikat("\nUsunieto element");
  }

void Dla_jednego_(PELEMENTD &Korzen)
 {  char Klucz[DL];
    if (Korzen==NULL)
     { Komunikat("\nDrzewo puste");   return;}
    Podaj_klucz(Klucz);
    Dla_jednego(Korzen,Klucz,Pokaz_dane);
 }

 void Wywaz_drzewo_(PELEMENTD &Korzen)
  { if (Korzen==NULL)
      { Komunikat("\nDrzewo puste");  return; }
   Oblicz_wezly(Korzen);
   if (Wywaz_drzewo(Korzen))      Komunikat("\nWywazono drzewo");
   else   Komunikat("\nNie wywazono drzewo-1 element");
  }
```

3. Złożoność obliczeniowa tablic, drzew i list (wg R.Sedgewick, Algorytmy w C++)

Typ struktury danych	Przypadek najgorszy-N / Przypadek średni-Ś					
i algorytmu	**wstawianie**		**wyszukiwanie**			**wybór**
	N	**Ś**	**N**	**Ś**		**N**
				traf.	**chyb.**	
tablica indeksowana kluczem	1	1	1	1	1	m
tablica uporządkowana	n	n/2	n	n/2	n/2	1
tablica nieuporządkowana	1	1	n	n/2	n	n lg n
lista uporządkowana	n	n/2	n	n/2	n/2	n
lista nieuporządkowana	1	1	n	n/2	n	n lg n
wyszukiwanie binarne (tablica)	n	n/2	lg n	lg n	lg n	1
binarne drzewo poszukiwań	n	lg n	n	lg n	lg n	n
drzewo czerwono-czarne	lg n	lg n	lg n	lg n	lg n	lg n