Algorytmy i struktury danych

wykład 5

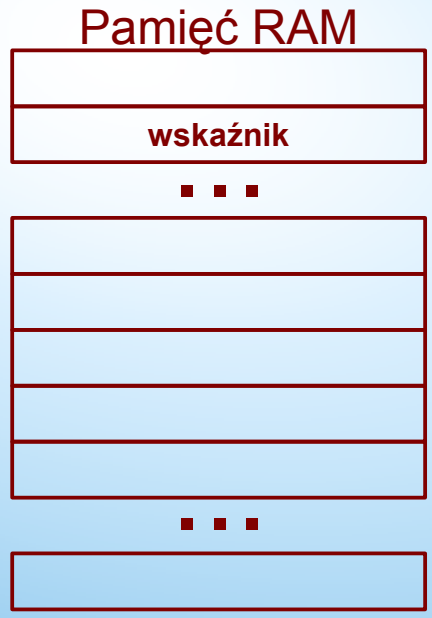
Plan wykładu:

- Wskaźniki.
- Złożone struktury danych:
  - listy,
  - drzewa,
  - kopce.

Wskaźniki

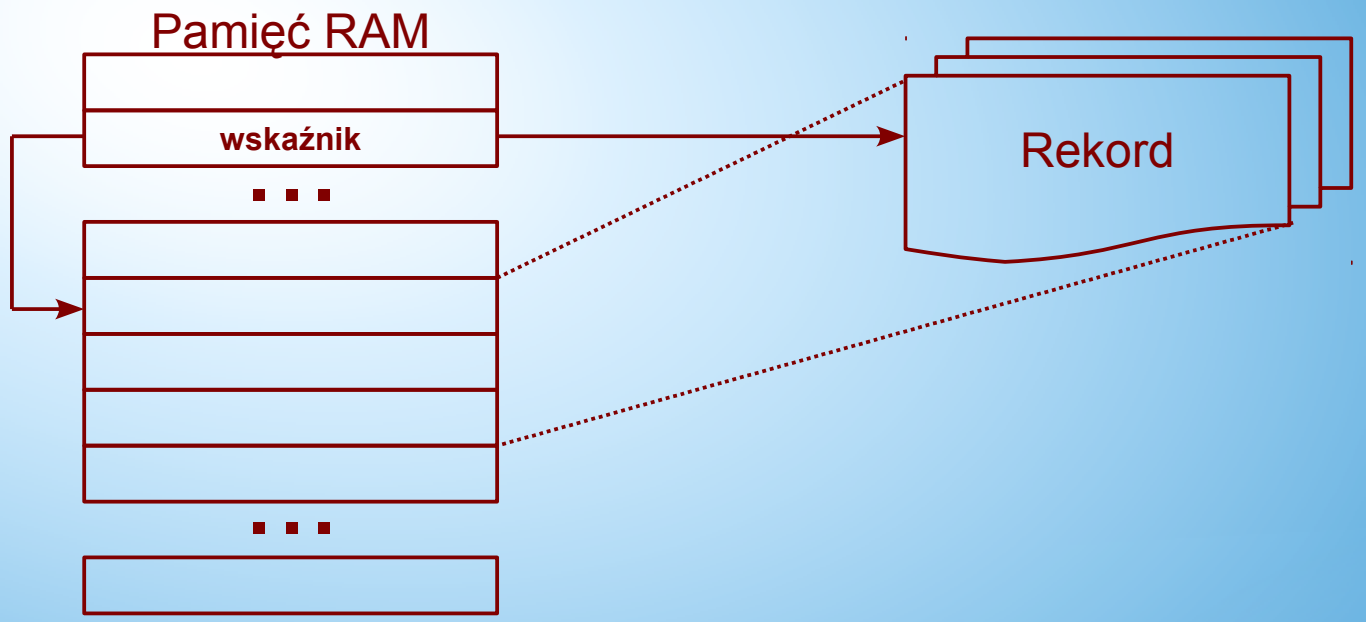
Wskaźnik – jest to liczba lub symbol który w ogólności wskazuje adres komórki pamięci. W językach wysokiego poziomu wskaźniki mogą również wskazywać na abstrakcyjne struktury danych.

Wskaźnik – jest to liczba lub symbol który w ogólności wskazuje adres komórki pamięci. W językach wysokiego poziomu wskaźniki mogą również wskazywać na abstrakcyjne struktury danych.

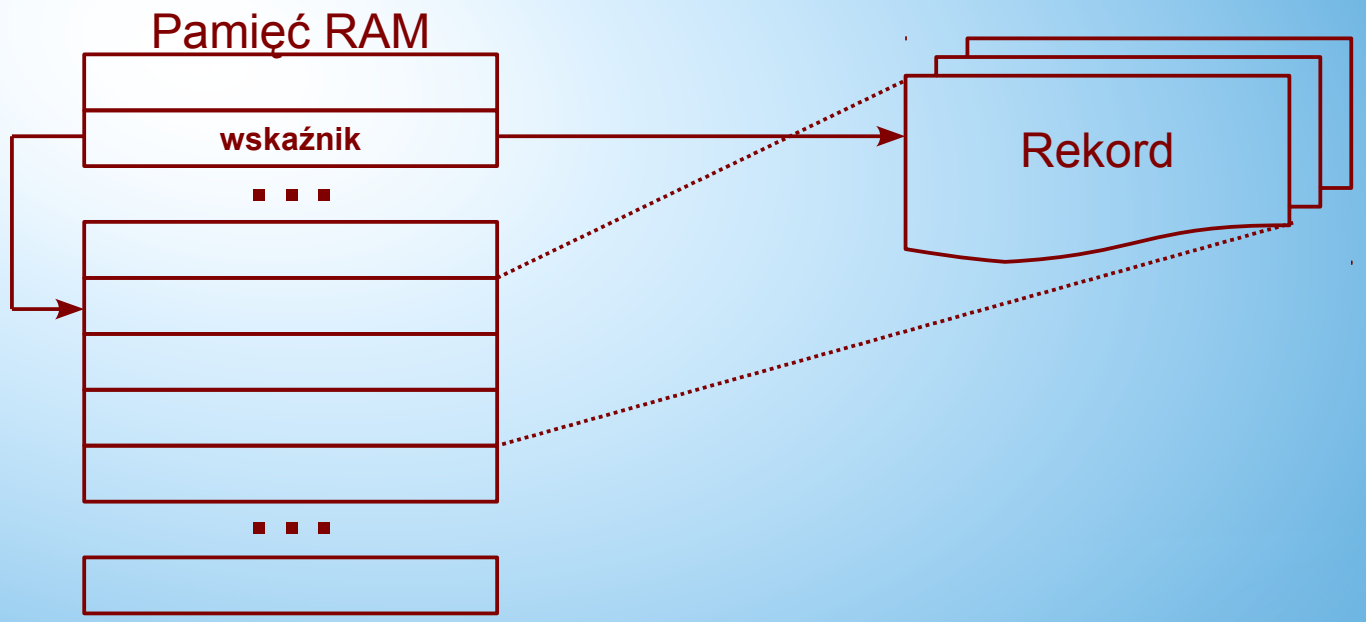


Wskaźnik – jest to liczba lub symbol który w ogólności wskazuje adres komórki pamięci. W językach wysokiego poziomu wskaźniki mogą również wskazywać na abstrakcyjne struktury danych.

Wskaźnik – jest to liczba lub symbol który w ogólności wskazuje adres komórki pamięci. W językach wysokiego poziomu wskaźniki mogą również wskazywać na abstrakcyjne struktury danych.



Wskaźnik – jest to liczba lub symbol który w ogólności wskazuje adres komórki pamięci. W językach wysokiego poziomu wskaźniki mogą również wskazywać na abstrakcyjne struktury danych.



Uwagi:

- wskaźnik to adres danej lub danych w pamięci,
- wskaźnik może wskazywać na inny wskaźnik,
- rozmiar wskaźnika jest równy rozmiarowi adresu fizycznego pamięci obowiązującym w danym systemie komputerowym.

Zmienne wskaźnikowe w językach programowania:

	Język Pascal	Język C
▪ deklaracja:	**type** PWsk = ^Wsk; **var** W: PWsk; W: ^Wsk;	TWsk *Wsk;
▪ wskaźnik na zmienną:	@nazwa_zmiennej;	&nazwa_zmiennej;
▪ wartość zmiennej wskazywanej:	nazwa_zmiennej^	∗nazwa_zmiennej
▪ wskazanie puste:	**nil**	NULL
▪ alokacja i zwalnianie pamięci:	New(nazwa_zmiennej); Dispose(nazwa_zmiennej); **...** GetMem(nazwa_zmiennej, rozmiar); FreeMem(nazwa_zmiennej, rozmiar);	typ ∗nazwa_zmiennej; nazwa_zmiennej = (typ ∗)mallok(rozmiar); **...** free(nazwa_zmiennej);

Dynamiczne struktury danych

Węzeł – struktura w pamięci, zawierająca dane oraz zmienne wskazujące, pozwalające na określenie położenia innego węzła w pamięci.

Uwagi:

- węzeł powinien zawierać co najmniej jedno pole wskaźnikowe,
- węzeł może wskazywać na samego siebie,
- węzeł nie musi zawierać pól danych,
- konstrukcja węzła wpływa na wydajność algorytmów.

Listy

Lista – jest to liniowa struktura będąca kontenerem danych, w której węzły wskazują co najwyżej jeden element następujący i co najwyżej jeden element poprzedzający dany węzeł.

Rodzaje list:

- jednokierunkowa prosta,
- jednokierunkowa cykliczna,
- dwukierunkowa prosta,
- dwukierunkowa cykliczna.

Lista – jest to liniowa struktura będąca kontenerem danych, w której węzły wskazują co najwyżej jeden element następujący i co najwyżej jeden element poprzedzający dany węzeł.

Rodzaje list:

- jednokierunkowa prosta,
- jednokierunkowa cykliczna,
- dwukierunkowa prosta,
- dwukierunkowa cykliczna.

Każdy węzeł wskazuje na następny, ostatni węzeł nie wskazuje na nic. Przeglądanie listy może odbywać się tylko w jednym kierunku.

Lista – jest to liniowa struktura będąca kontenerem danych, w której węzły wskazują co najwyżej jeden element następujący i co najwyżej jeden element poprzedzający dany węzeł.

Rodzaje list:

- jednokierunkowa prosta,
- jednokierunkowa cykliczna,
- dwukierunkowa prosta,
- dwukierunkowa cykliczna.

Każdy węzeł wskazuje na następny, ostatni węzeł nie wskazuje na nic. Przeglądanie listy może odbywać się tylko w jednym kierunku.

Każdy węzeł wskazuje na następny, ostatni węzeł wskazuje na początek listy. Przeglądanie listy może odbywać się tylko w jednym kierunku.

Lista – jest to liniowa struktura będąca kontenerem danych, w której węzły wskazują co najwyżej jeden element następujący i co najwyżej jeden element poprzedzający dany węzeł.

Rodzaje list:

- jednokierunkowa prosta,
- jednokierunkowa cykliczna,
- dwukierunkowa prosta,
- dwukierunkowa cykliczna.

Każdy węzeł wskazuje na następny, ostatni węzeł nie wskazuje na nic. Przeglądanie listy może odbywać się tylko w jednym kierunku.

Każdy węzeł wskazuje na następny, ostatni węzeł wskazuje na początek listy. Przeglądanie listy może odbywać się tylko w jednym kierunku.

Każdy węzeł wskazuje na następny i poprzedni węzeł. Pierwszy węzeł nie wskazuje na żaden poprzedni, ostatni węzeł nie wskazuje na żaden następny. Przeglądanie listy może odbywać się w dwóch kierunkach.

Lista – jest to liniowa struktura będąca kontenerem danych, w której węzły wskazują co najwyżej jeden element następujący i co najwyżej jeden element poprzedzający dany węzeł.

Rodzaje list:

- jednokierunkowa prosta,
- jednokierunkowa cykliczna,
- dwukierunkowa prosta,
- dwukierunkowa cykliczna.

Każdy węzeł wskazuje na następny, ostatni węzeł nie wskazuje na nic. Przeglądanie listy może odbywać się tylko w jednym kierunku.

Każdy węzeł wskazuje na następny, ostatni węzeł wskazuje na początek listy. Przeglądanie listy może odbywać się tylko w jednym kierunku.

Każdy węzeł wskazuje na następny i poprzedni węzeł. Pierwszy węzeł wskazuje na koniec listy, ostatni węzeł wskazuje na jej początek. Przeglądanie listy może odbywać się w dwóch kierunkach.

Każdy węzeł wskazuje na następny i poprzedni węzeł. Pierwszy węzeł nie wskazuje na żaden poprzedni, ostatni węzeł nie wskazuje na żaden następny. Przeglądanie listy może odbywać się w dwóch kierunkach.

Lista – jest to liniowa struktura będąca kontenerem danych, w której węzły wskazują co najwyżej jeden element następujący i co najwyżej jeden element poprzedzający dany węzeł.

Rodzaje list:

- jednokierunkowa prosta,
- jednokierunkowa cykliczna,
- dwukierunkowa prosta,
- dwukierunkowa cykliczna.

Każdy węzeł wskazuje na następny, ostatni węzeł nie wskazuje na nic. Przeglądanie listy może odbywać się tylko w jednym kierunku.

Każdy węzeł wskazuje na następny, ostatni węzeł wskazuje na początek listy. Przeglądanie listy może odbywać się tylko w jednym kierunku.

Każdy węzeł wskazuje na następny i poprzedni węzeł. Pierwszy węzeł wskazuje na koniec listy, ostatni węzeł wskazuje na jej początek. Przeglądanie listy może odbywać się w dwóch kierunkach.

Każdy węzeł wskazuje na następny i poprzedni węzeł. Pierwszy węzeł nie wskazuje na żaden poprzedni, ostatni węzeł nie wskazuje na żaden następny. Przeglądanie listy może odbywać się w dwóch kierunkach.

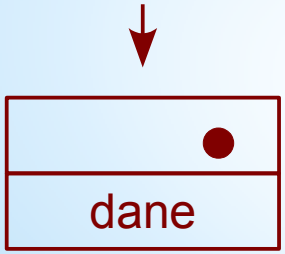
Uwagi:

- pierwszy węzeł w liście nazywany jest jej początkiem,
- lista może składać się z tylko jednego węzła,
- listy mogą być uporządkowane.

Lista jednokierunkowa

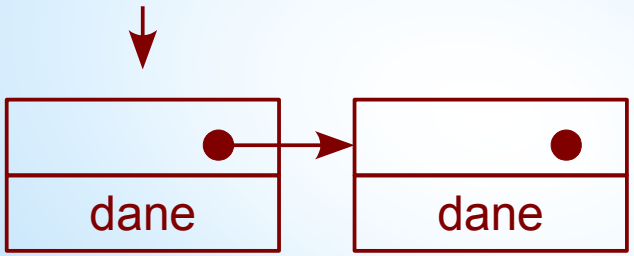
Struktura listy jednokierunkowej:

Początek listy



Struktura listy jednokierunkowej:

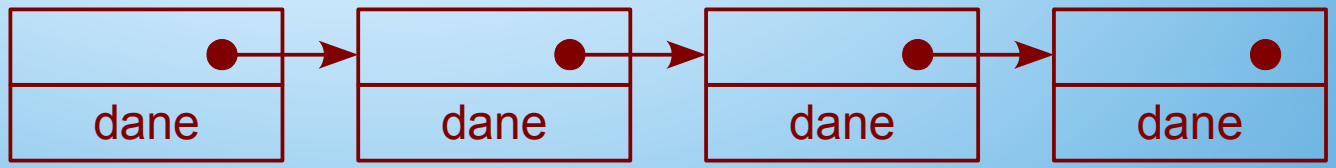
Początek listy



Struktura listy jednokierunkowej:

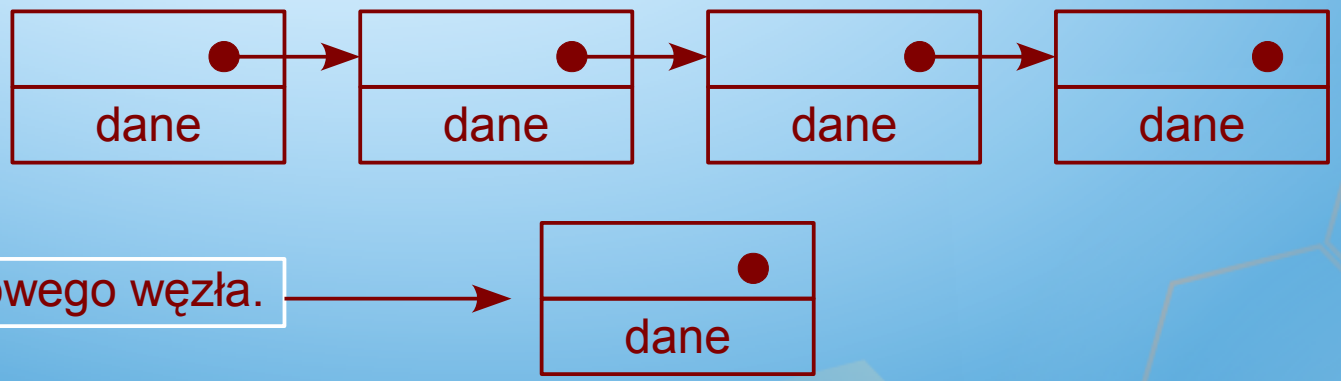
Struktura listy jednokierunkowej:

Dodawanie węzła:



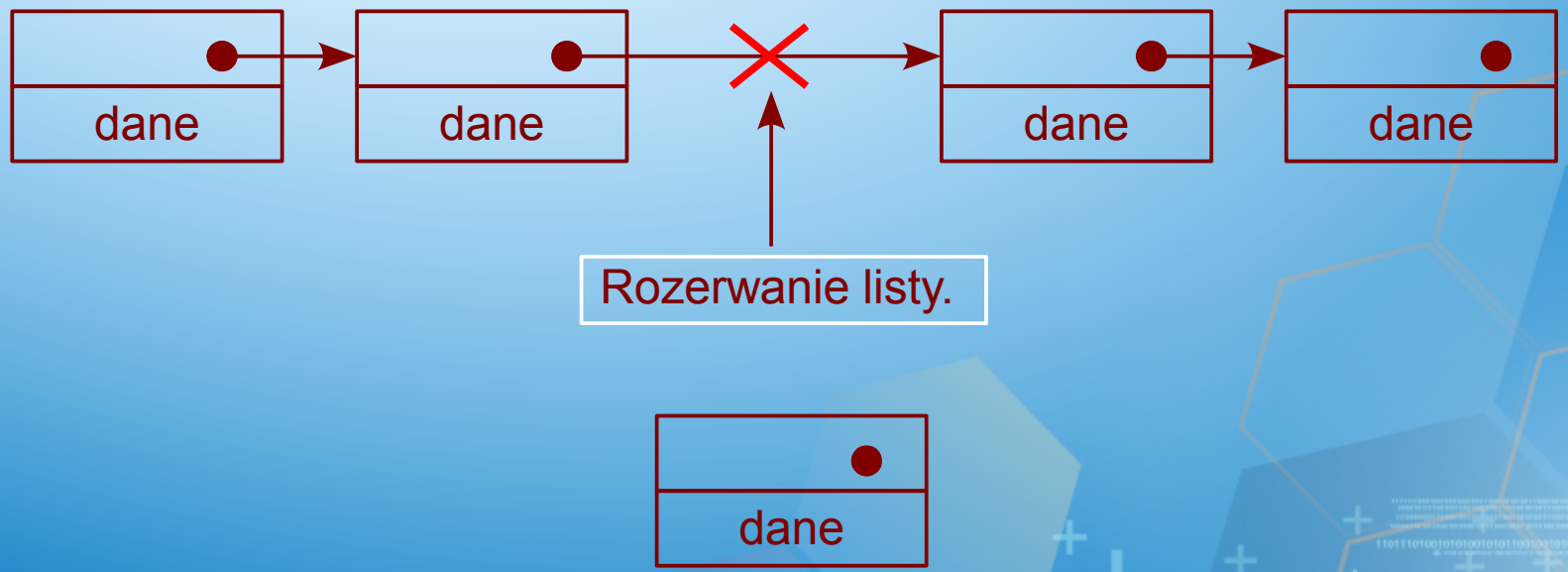
Struktura listy jednokierunkowej:

Dodawanie węzła:



Struktura listy jednokierunkowej:

Dodawanie węzła:

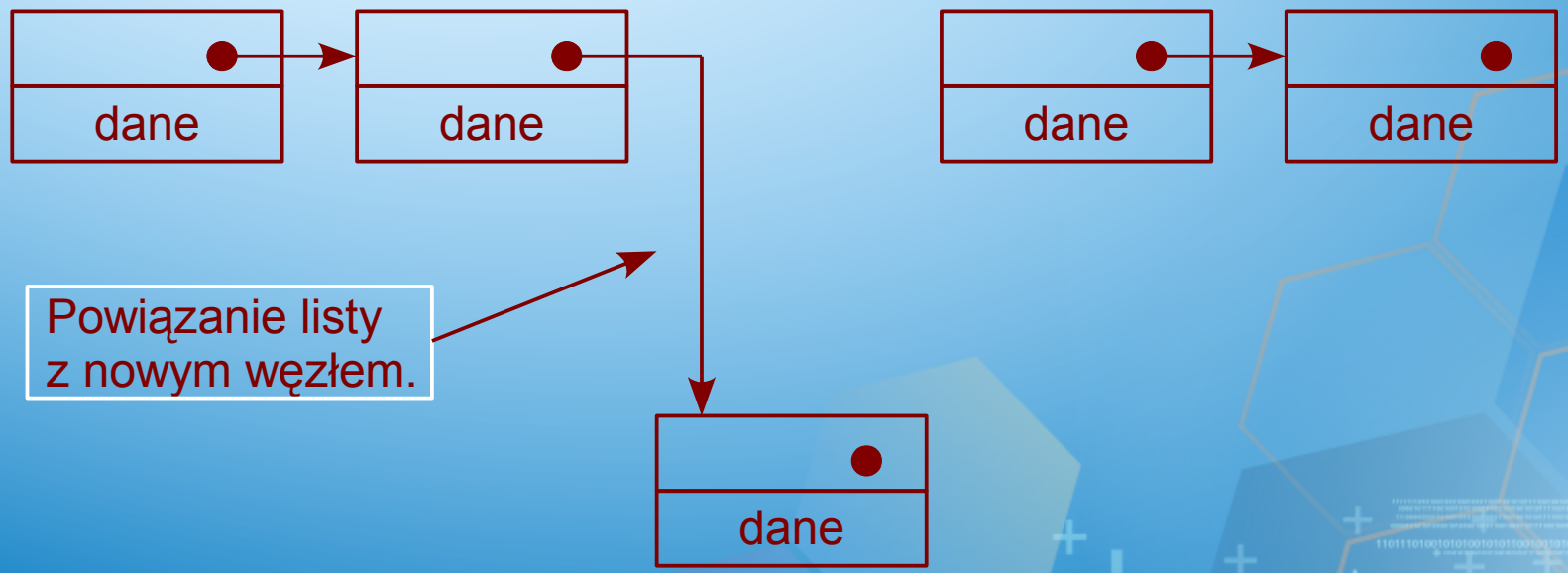


Struktura listy jednokierunkowej:

Dodawanie węzła:

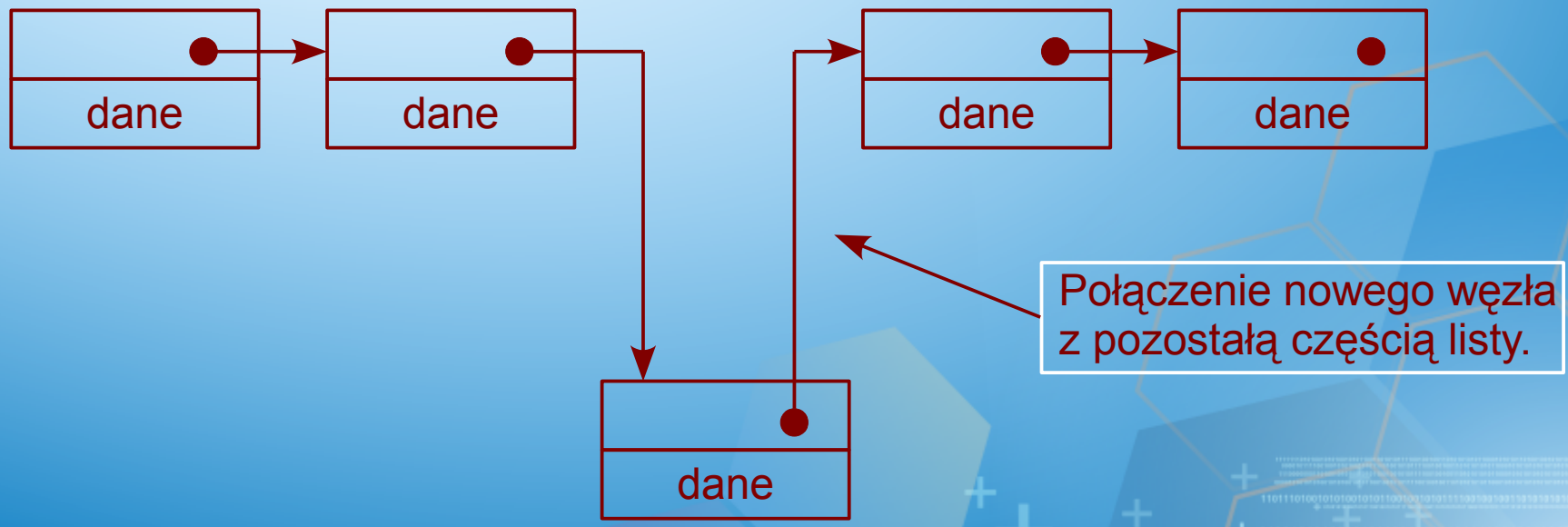
Struktura listy jednokierunkowej:

Dodawanie węzła:



Struktura listy jednokierunkowej:

Dodawanie węzła:

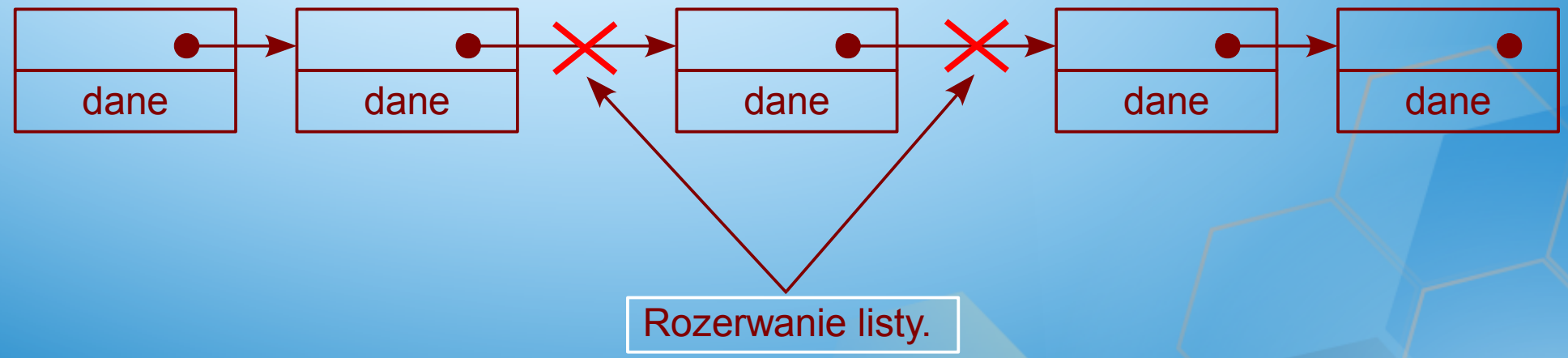


Struktura listy jednokierunkowej:

Usuwanie węzła:

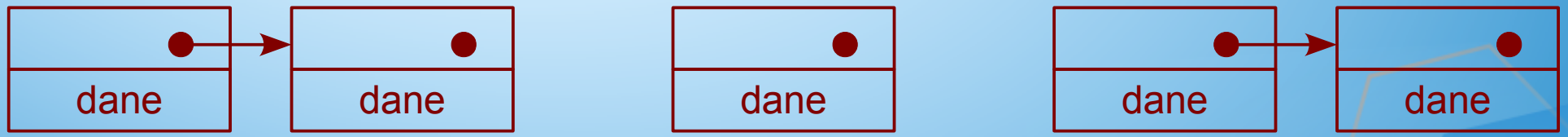
Struktura listy jednokierunkowej:

Usuwanie węzła:



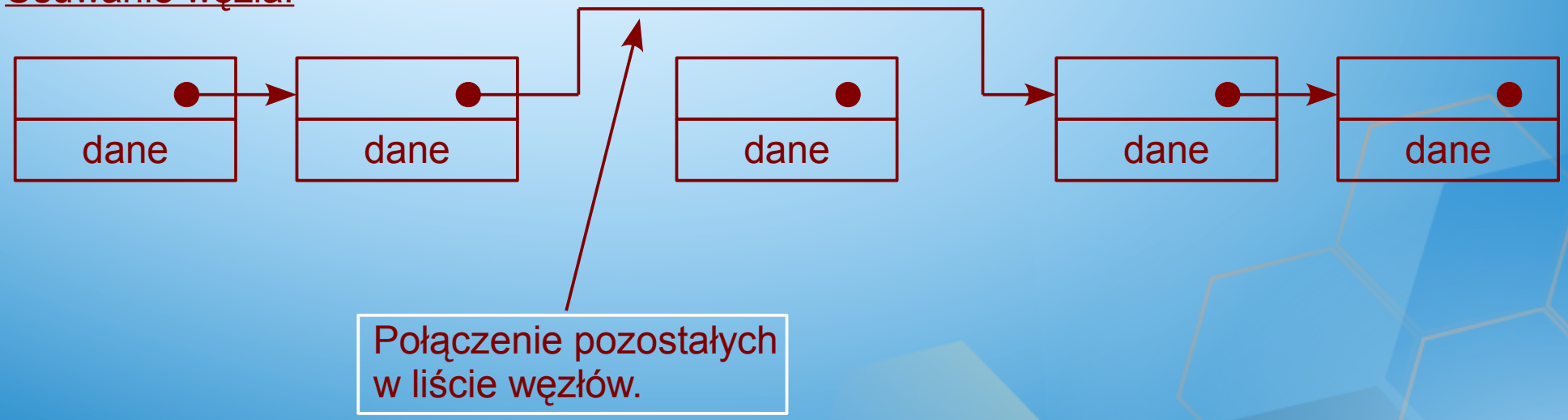
Struktura listy jednokierunkowej:

Usuwanie węzła:



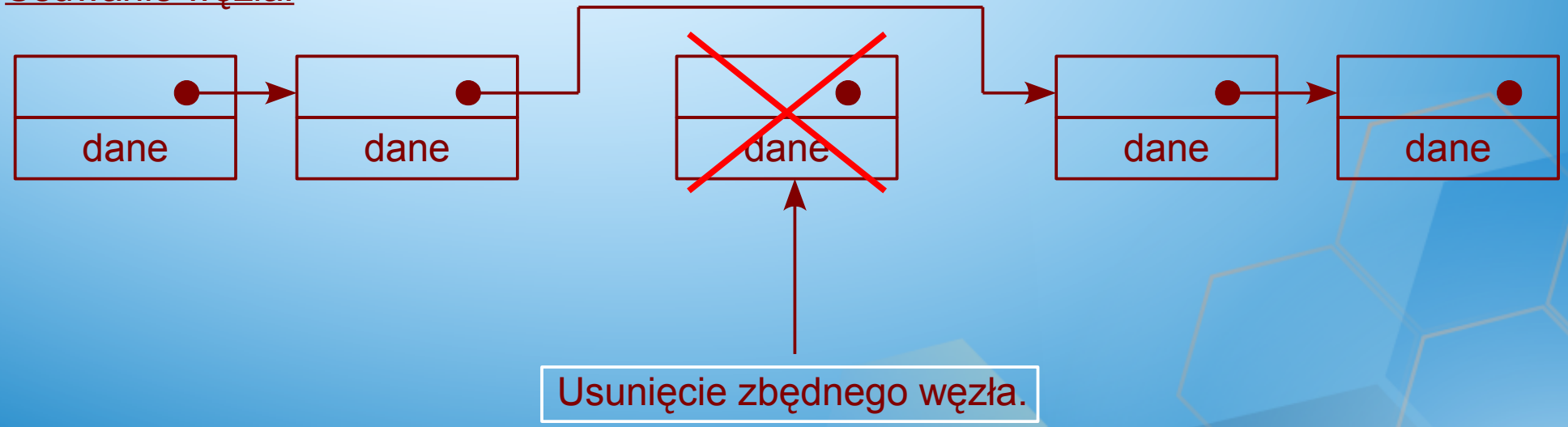
Struktura listy jednokierunkowej:

Usuwanie węzła:



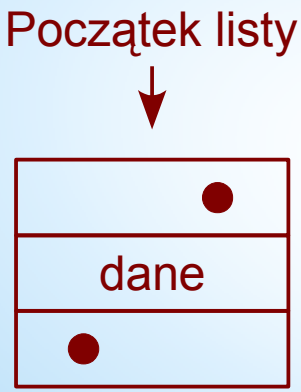
Struktura listy jednokierunkowej:

Usuwanie węzła:

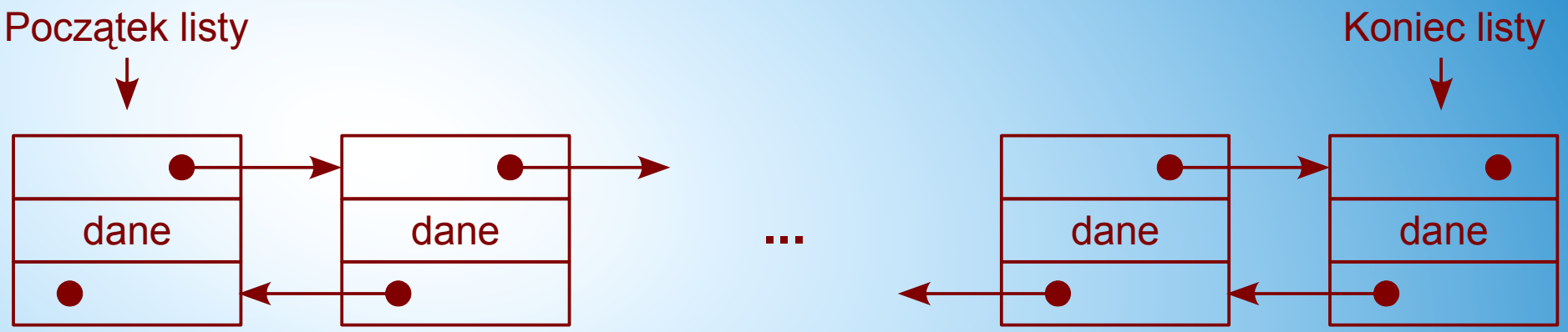


Lista dwukierunkowa

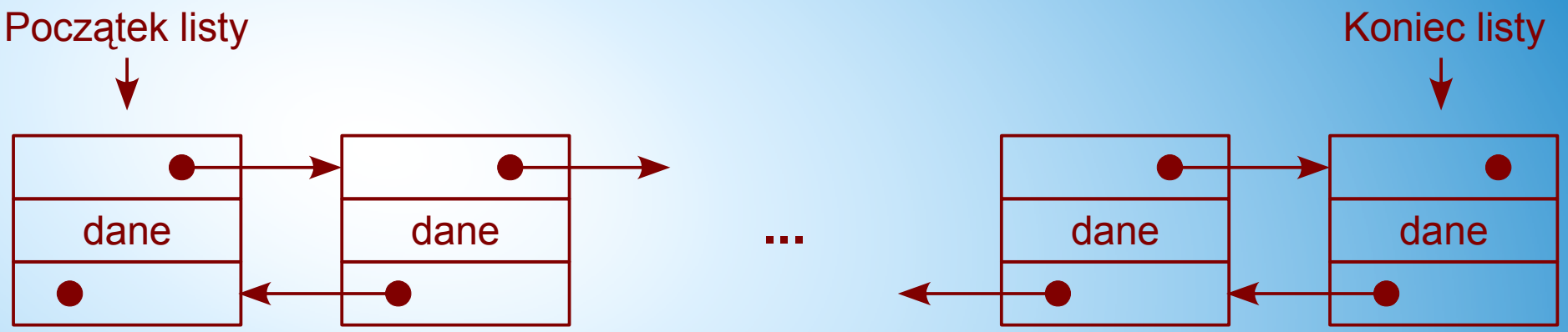
Struktura listy dwukierunkowej:



Struktura listy dwukierunkowej:



Struktura listy dwukierunkowej:

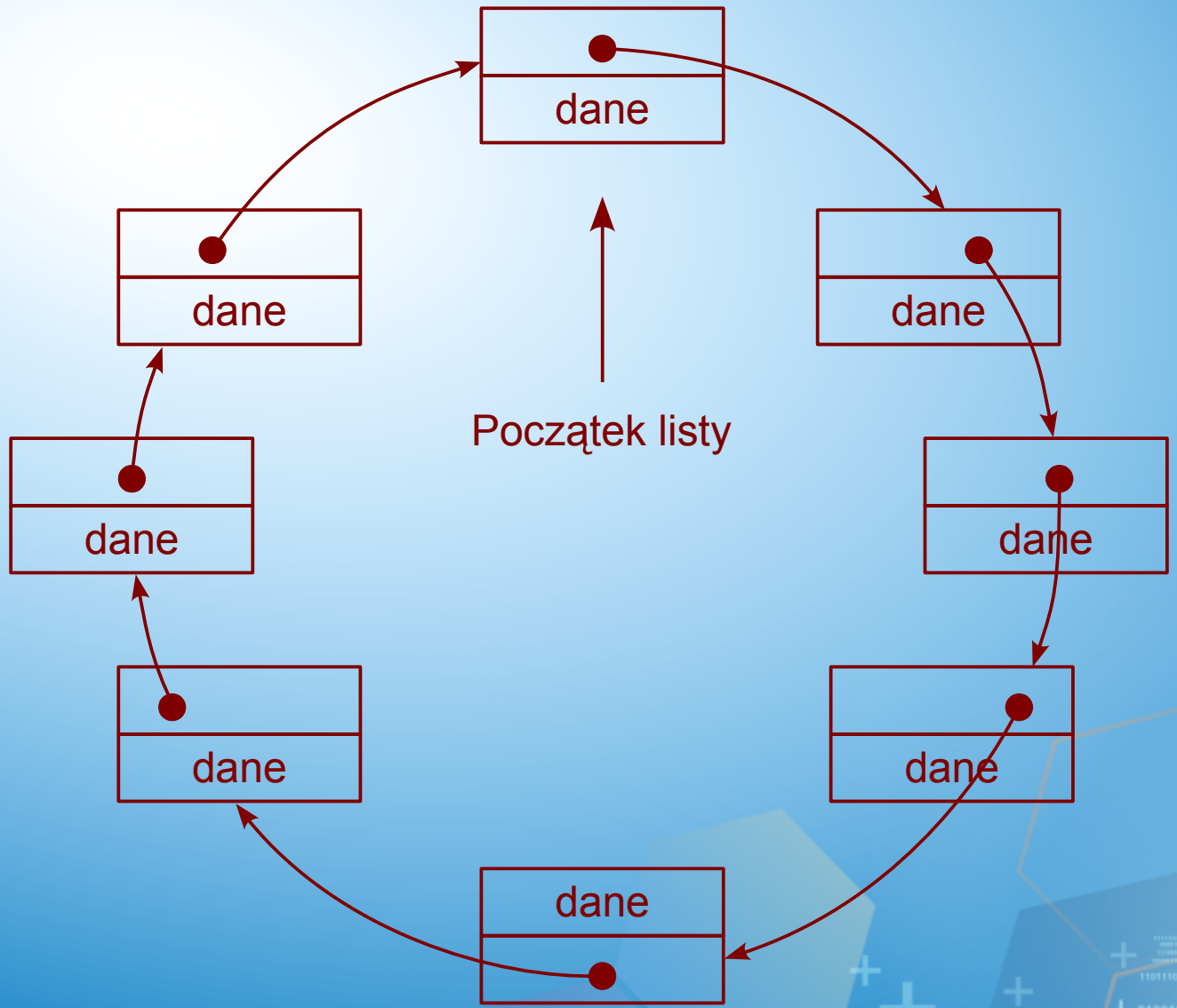


Uwagi:

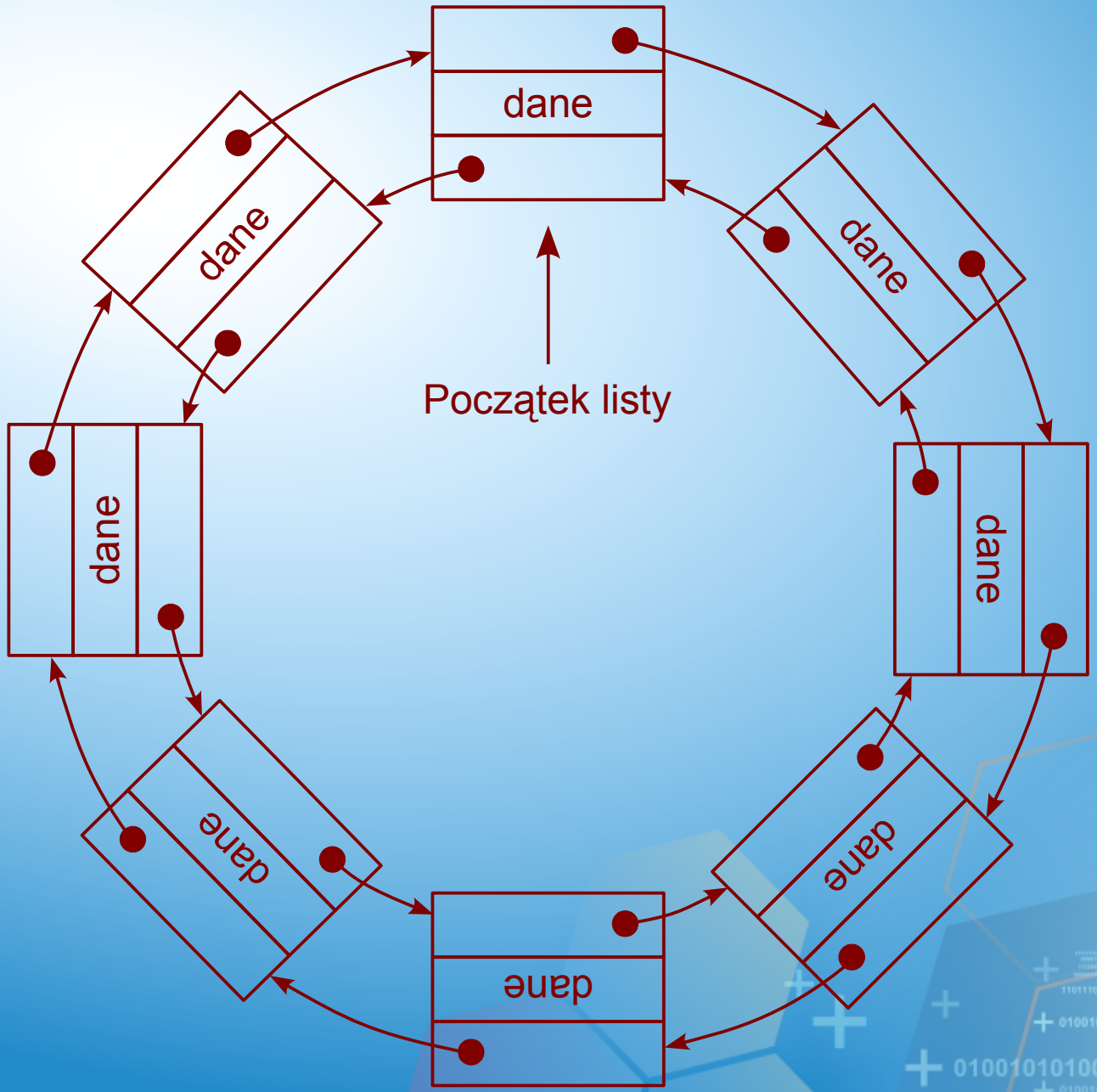
- dodawanie elementu jest podobne do czynności związanych z listą jednokierunkową – wymaga rozerwania listy, ale dodatkowo należy uzupełnić wskazania w sąsiednich węzłach,
- usuwanie elementu z listy także wymaga uzupełnienia wskazań w węźle poprzedzającym i węźle następującym po węźle usuwanym.

Listy cykliczne

Struktura listy jednokierunkowej cyklicznej:



Struktura listy dwukierunkowej cyklicznej:



Listy w językach programowania

Lista w językach programowania:

- deklaracja typu listy jednokierunkowej prostej:

```
type    PElementListy = ^TElementListy;
        TElementListy = record
            wartosc: Integer;
            nast_elem: PElementListy;
        end;
```

Lista w językach programowania:

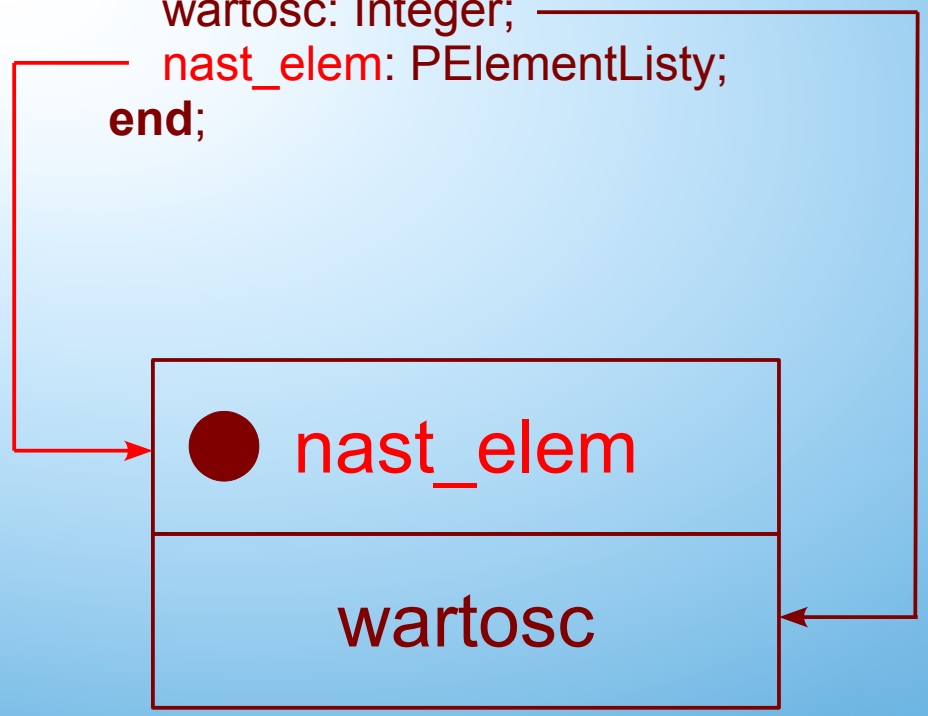
- deklaracja typu listy jednokierunkowej prostej:

```
type    PElementListy = ^TElementListy;
        TElementListy = record
            wartosc: Integer;
            nast_elem: PElementListy;
        end;
```

Lista w językach programowania:

- deklaracja typu listy jednokierunkowej prostej:

```
type    PElementListy = ^TElementListy;
        TElementListy = record
            wartosc: Integer;
            nast_elem: PElementListy;
        end;
```



- dodanie elementu do listy jednokierunkowej prostej na jej końcu:

```
var       Lista: PElementListy;
...
function dodaj_element_do_listy(Lista: PElementListy;
                                wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   New(L);
   L^.wartosc := wartosc;
   L^.nast_elem := nil;
   if Lista <> nil then begin
      P := Lista;
      while P^.nast_elem <> nil do P := P^.nast_elem;
      P^.nast_elem := L;
      dodaj_element_do_listy := Lista
   end else dodaj_element_do_listy := L
end;
```

- dodanie elementu do listy jednokierunkowej prostej na jej końcu:

```
var       Lista: PElementListy;
...
function dodaj_element_do_listy(Lista: PElementListy;
                                wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   New(L);
   L^.wartosc := wartosc;
   L^.nast_elem := nil;
   if Lista <> nil then begin
      P := Lista;
      while P^.nast_elem <> nil do P := P^.nast_elem;
      P^.nast_elem := L;
      dodaj_element_do_listy := Lista
   end else dodaj_element_do_listy := L
end;
```

Deklaracja zmiennej przechowującej początek listy.

- dodanie elementu do listy jednokierunkowej prostej na jej końcu:

```
var       Lista: PElementListy;
...
function dodaj_element_do_listy(Lista: PElementListy;
                                wartosc: Integer): PElementListy;

var       L, P: PElementListy;
begin
   New(L);
   L^.wartosc := wartosc;
   L^.nast_elem := nil;
   if Lista <> nil then begin
      P := Lista;
      while P^.nast_elem <> nil do P := P^.nast_elem;
      P^.nast_elem := L;
      dodaj_element_do_listy := Lista
   end else dodaj_element_do_listy := L
end;
```

Deklaracja funkcji dodającej węzeł do listy. Parametr Lista jest wskaźnikiem na początek listy, parametr wartosc jest daną, która ma być zapamiętana w węźle listy.

- dodanie elementu do listy jednokierunkowej prostej na jej końcu:

```
var       Lista: PElementListy;
...
function dodaj_element_do_listy(Lista: PElementListy;
                                wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   New(L);
   L^.wartosc := wartosc;
   L^.nast_elem := nil;
   if Lista <> nil then begin
      P := Lista;
      while P^.nast_elem <> nil do P := P^.nast_elem;
      P^.nast_elem := L;
      dodaj_element_do_listy := Lista
   end else dodaj_element_do_listy := L
end;
```

Zmienne w funkcji będą wykorzystane podczas dodawania węzła do listy.

- dodanie elementu do listy jednokierunkowej prostej na jej końcu:

```
var       Lista: PElementListy;
...
function dodaj_element_do_listy(Lista: PElementListy;
                                wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   New(L);
   L^.wartosc := wartosc;
   L^.nast_elem := nil;
   if Lista <> nil then begin
      P := Lista;
      while P^.nast_elem <> nil do P := P^.nast_elem;
      P^.nast_elem := L;
      dodaj_element_do_listy := Lista
   end else dodaj_element_do_listy := L
end;
```

Utworzenie nowego, pustego węzła ...

- dodanie elementu do listy jednokierunkowej prostej na jej końcu:

```
var       Lista: PElementListy;
...
function dodaj_element_do_listy(Lista: PElementListy;
                                wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   New(L);
   L^.wartosc := wartosc;
   L^.nast_elem := nil;
   if Lista <> nil then begin
      P := Lista;
      while P^.nast_elem <> nil do P := P^.nast_elem;
      P^.nast_elem := L;
      dodaj_element_do_listy := Lista
   end else dodaj_element_do_listy := L
end;
```

... oraz wprowadzenie danej i zainicjowanie adresu następnego węzła na wskazanie puste.

- dodanie elementu do listy jednokierunkowej prostej na jej końcu:

```
var       Lista: PElementListy;
...
function dodaj_element_do_listy(Lista: PElementListy;
                                wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   New(L);
   L^.wartosc := wartosc;
   L^.nast_elem := nil;
   if Lista <> nil then begin
      P := Lista;
      while P^.nast_elem <> nil do P := P^.nast_elem;
      P^.nast_elem := L;
      dodaj_element_do_listy := Lista
   end else dodaj_element_do_listy := L
end;
```

Jeśli lista istnieje (czyli Lista jest różna od nil) to nastąpi odszukanie końca listy.

- dodanie elementu do listy jednokierunkowej prostej na jej końcu:

```
var       Lista: PElementListy;
...
function dodaj_element_do_listy(Lista: PElementListy;
                                wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   New(L);
   L^.wartosc := wartosc;
   L^.nast_elem := nil;
   if Lista <> nil then begin
      P := Lista;
      while P^.nast_elem <> nil do P := P^.nast_elem;
      P^.nast_elem := L;
      dodaj_element_do_listy := Lista
   end else dodaj_element_do_listy := L
end;
```

Po wykonaniu wyróżnionego fragmentu funkcji zmienna P wskazuje na koniec listy.

- dodanie elementu do listy jednokierunkowej prostej na jej końcu:

```
var       Lista: PElementListy;
...
function dodaj_element_do_listy(Lista: PElementListy;
                                wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   New(L);
   L^.wartosc := wartosc;
   L^.nast_elem := nil;
   if Lista <> nil then begin
      P := Lista;
      while P^.nast_elem <> nil do P := P^.nast_elem;
      P^.nast_elem := L;
      dodaj_element_do_listy := Lista
   end else dodaj_element_do_listy := L
end;
```

P wskazuje na koniec listy, zatem dodanie nowego węzła do listy będzie polegało na dopisaniu jego adresu do aktualnego końca listy. Wstawiany element został już wcześniej poprawnie zainicjowany.

- dodanie elementu do listy jednokierunkowej prostej na jej końcu:

```
var       Lista: PElementListy;
...
function dodaj_element_do_listy(Lista: PElementListy;
                                wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   New(L);
   L^.wartosc := wartosc;
   L^.nast_elem := nil;
   if Lista <> nil then begin
      P := Lista;
      while P^.nast_elem <> nil do P := P^.nast_elem;
      P^.nast_elem := L;
      dodaj_element_do_listy := Lista
   end else dodaj_element_do_listy := L
end;
```

Po dodaniu elementu do listy funkcja zwraca jako wynik wskaźnik na początek listy.

- dodanie elementu do listy jednokierunkowej prostej na jej końcu:

```
var       Lista: PElementListy;
...
function dodaj_element_do_listy(Lista: PElementListy;
                                wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   New(L);
   L^.wartosc := wartosc;
   L^.nast_elem := nil;
   if Lista <> nil then begin
      P := Lista;
      while P^.nast_elem <> nil do P := P^.nast_elem;
      P^.nast_elem := L;
      dodaj_element_do_listy := Lista
   end else dodaj_element_do_listy := L
end;
```

Jeśli lista nie istniała (czyli Lista posiadała wartość nil), to funkcja zwraca jako wynik węzeł utworzony na początku funkcji. Jednocześnie będzie to początek listy.

- dodanie elementu do listy jednokierunkowej prostej na jej końcu:

```
var       Lista: PElementListy;
...
function dodaj_element_do_listy(Lista: PElementListy;
                                wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   New(L);
   L^.wartosc := wartosc;
   L^.nast_elem := nil;
   if Lista <> nil then begin
      P := Lista;
      while P^.nast_elem <> nil do P := P^.nast_elem;
      P^.nast_elem := L;
      dodaj_element_do_listy := Lista
   end else dodaj_element_do_listy := L
end;
```

Funkcja dodająca węzeł do listy powinna mieć możliwość jego dodania w dowolnym miejscu listy.

- usuwanie elementu z listy jednokierunkowej prostej:

```
function usun_element_z_listy(Lista: PElementListy;
                              wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   L := Lista; P := Lista;
   while (P <> nil) and (P^.wartosc <> wartosc) do
   begin
      L := P;
      P := P^.nast_elem;
   end;
   if P <> nil then
   begin
      if P = L then begin
         L := L^.nast_elem;
         usun_element_z_listy := L
      end else begin
         L^.nast_elem := P^.nast_elem;
         usun_element_z_listy := Lista
      end;
      Dispose(P);
   end
end;
```

- usuwanie elementu z listy jednokierunkowej prostej:

```
function usun_element_z_listy(Lista: PElementListy;
                              wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   L := Lista; P := Lista;
   while (P <> nil) and (P^.wartosc <> wartosc) do
   begin
      L := P;
      P := P^.nast_elem;
   end;
   if P <> nil then
   begin
      if P = L then begin
         L := L^.nast_elem;
         usun_element_z_listy := L
      end else begin
         L^.nast_elem := P^.nast_elem;
         usun_element_z_listy := Lista
      end;
      Dispose(P);
   end
end;
```

Funkcja będzie usuwała z listy wskazanej przez zmienną Lista pierwszy znaleziony węzeł z daną równą zmiennej wartosc.

- usuwanie elementu z listy jednokierunkowej prostej:

```
function usun_element_z_listy(Lista: PElementListy;
                              wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   L := Lista; P := Lista;
   while (P <> nil) and (P^.wartosc <> wartosc) do
   begin
      L := P;
      P := P^.nast_elem;
   end;
   if P <> nil then
   begin
      if P = L then begin
         L := L^.nast_elem;
         usun_element_z_listy := L
      end else begin
         L^.nast_elem := P^.nast_elem;
         usun_element_z_listy := Lista
      end;
      Dispose(P);
   end
end;
```

Na początku trzeba odszukać węzeł, który ma zostać usunięty. Zmienna L będzie wskazywała węzeł poprzedni, zmienna P będzie wskazywała węzeł bieżący.

- usuwanie elementu z listy jednokierunkowej prostej:

```
function usun_element_z_listy(Lista: PElementListy;
                              wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   L := Lista; P := Lista;
   while (P <> nil) and (P^.wartosc <> wartosc) do
   begin
      L := P;
      P := P^.nast_elem;
   end;
   if P <> nil then
   begin
      if P = L then begin
         L := L^.nast_elem;
         usun_element_z_listy := L
      end else begin
         L^.nast_elem := P^.nast_elem;
         usun_element_z_listy := Lista
      end;
      Dispose(P);
   end
end;
```

Funkcja uwzględnia przypadek, gdy lista nie istnieje tj. P = nil. Ten sam warunek powoduje zakończenie wyszukiwania, gdy zostanie osiągnięty koniec listy.

- usuwanie elementu z listy jednokierunkowej prostej:

```
function usun_element_z_listy(Lista: PElementListy;
                              wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   L := Lista; P := Lista;
   while (P <> nil) and (P^.wartosc <> wartosc) do
   begin
      L := P;
      P := P^.nast_elem;
   end;
   if P <> nil then
   begin
      if P = L then begin
         L := L^.nast_elem;
         usun_element_z_listy := L
      end else begin
         L^.nast_elem := P^.nast_elem;
         usun_element_z_listy := Lista
      end;
      Dispose(P);
   end
end;
```

Odnajdywanie właściwego węzła zakończy się również, gdy zostanie taki znaleziony. Jeśli P nie wskazuje tego węzła, to kolejne wykonanie pętli powoduje przypisanie do P następnego węzła z listy. Zmienna L będzie wtedy wskazywać węzeł poprzedzający P.

- usuwanie elementu z listy jednokierunkowej prostej:

```
function usun_element_z_listy(Lista: PElementListy;
                              wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   L := Lista; P := Lista;
   while (P <> nil) and (P^.wartosc <> wartosc) do
   begin
      L := P;
      P := P^.nast_elem;
   end;
   if P <> nil then
   begin
      if P = L then begin
         L := L^.nast_elem;
         usun_element_z_listy := L
      end else begin
         L^.nast_elem := P^.nast_elem;
         usun_element_z_listy := Lista
      end;
      Dispose(P);
   end
end;
```

Jeśli P jest różne od nil, to oznacza, że właściwy węzeł został znaleziony i można przystąpić do jego usunięcia.

- usuwanie elementu z listy jednokierunkowej prostej:

```
function usun_element_z_listy(Lista: PElementListy;
                              wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   L := Lista; P := Lista;
   while (P <> nil) and (P^.wartosc <> wartosc) do
   begin
      L := P;
      P := P^.nast_elem;
   end;
   if P <> nil then
   begin
      if P = L then begin
         L := L^.nast_elem;
         usun_element_z_listy := L
      end else begin
         L^.nast_elem := P^.nast_elem;
         usun_element_z_listy := Lista
      end;
      Dispose(P);
   end
end;
```

Jeśli P = L to oznacza, że usuwany będzie pierwszy węzeł listy. Zatem funkcja zwróci nowy początek listy, będący następnym elementem po węźle P (L).

- usuwanie elementu z listy jednokierunkowej prostej:

```
function usun_element_z_listy(Lista: PElementListy;
                              wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   L := Lista; P := Lista;
   while (P <> nil) and (P^.wartosc <> wartosc) do
   begin
      L := P;
      P := P^.nast_elem;
   end;
   if P <> nil then
   begin
      if P = L then begin
         L := L^.nast_elem;
         usun_element_z_listy := L
      end else begin
         L^.nast_elem := P^.nast_elem;
         usun_element_z_listy := Lista
      end;
      Dispose(P);
   end
end;
```

Jeśli P <> L to oznacza, że usuwany będzie węzeł z środka lub końca listy.

- usuwanie elementu z listy jednokierunkowej prostej:

```
function usun_element_z_listy(Lista: PElementListy;
                              wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   L := Lista; P := Lista;
   while (P <> nil) and (P^.wartosc <> wartosc) do
   begin
      L := P;
      P := P^.nast_elem;
   end;
   if P <> nil then
   begin
      if P = L then begin
         L := L^.nast_elem;
         usun_element_z_listy := L
      end else begin
         L^.nast_elem := P^.nast_elem;
         usun_element_z_listy := Lista
      end;
      Dispose(P);
   end
end;
```

Jeśli P <> L to oznacza, że usuwany będzie węzeł z środka lub końca listy: rozerwanie listy i jednoczesne połączenie listy z pominięciem wyłączonego węzła P.

- usuwanie elementu z listy jednokierunkowej prostej:

```
function usun_element_z_listy(Lista: PElementListy;
                              wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   L := Lista; P := Lista;
   while (P <> nil) and (P^.wartosc <> wartosc) do
   begin
      L := P;
      P := P^.nast_elem;
   end;
   if P <> nil then
   begin
      if P = L then begin
         L := L^.nast_elem;
         usun_element_z_listy := L
      end else begin
         L^.nast_elem := P^.nast_elem;
         usun_element_z_listy := Lista
      end;
      Dispose(P);
   end
end;
```

Ponieważ początek listy nie uległ zmianie, to funkcja zwróci ten sam początek, jaki otrzymała podczas wywołania.

- usuwanie elementu z listy jednokierunkowej prostej:

```
function usun_element_z_listy(Lista: PElementListy;
                              wartosc: Integer): PElementListy;
var       L, P: PElementListy;
begin
   L := Lista; P := Lista;
   while (P <> nil) and (P^.wartosc <> wartosc) do
   begin
      L := P;
      P := P^.nast_elem;
   end;
   if P <> nil then
   begin
      if P = L then begin
         L := L^.nast_elem;
         usun_element_z_listy := L
      end else begin
         L^.nast_elem := P^.nast_elem;
         usun_element_z_listy := Lista
      end;
      Dispose(P);
   end
end;
```

Usunięcie z pamięci wyłączonego węzła P.

Czynności dodatkowe funkcji obsługujących listy:

- funkcja dodająca elementy do listy powinna ją także tworzyć,
- funkcja dodająca elementy może porządkować listę lub wstawiać elementy w wybrane miejsca listy,
- funkcja usuwająca elementy z listy powinna także usuwać listę, jeśli to konieczne,
- w listach dwukierunkowych mechanika rozrywania listy wymaga większej liczby operacji,
- w listach cyklicznych dodawanie węzłów wymaga uwzględnienia specyfiki listy,
- w listach cyklicznych usuwanie węzłów również wymaga uwzględnienia specyfiki listy.

Czynności dodatkowe funkcji obsługujących listy:

- funkcja dodająca elementy do listy powinna ją także tworzyć,
- funkcja dodająca elementy może porządkować listę lub wstawiać elementy w wybrane miejsca listy,
- funkcja usuwająca elementy z listy powinna także usuwać listę, jeśli to konieczne,
- w listach dwukierunkowych mechanika rozrywania listy wymaga większej liczby operacji,
- w listach cyklicznych dodawanie węzłów wymaga uwzględnienia specyfiki listy,
- w listach cyklicznych usuwanie węzłów również wymaga uwzględnienia specyfiki listy.

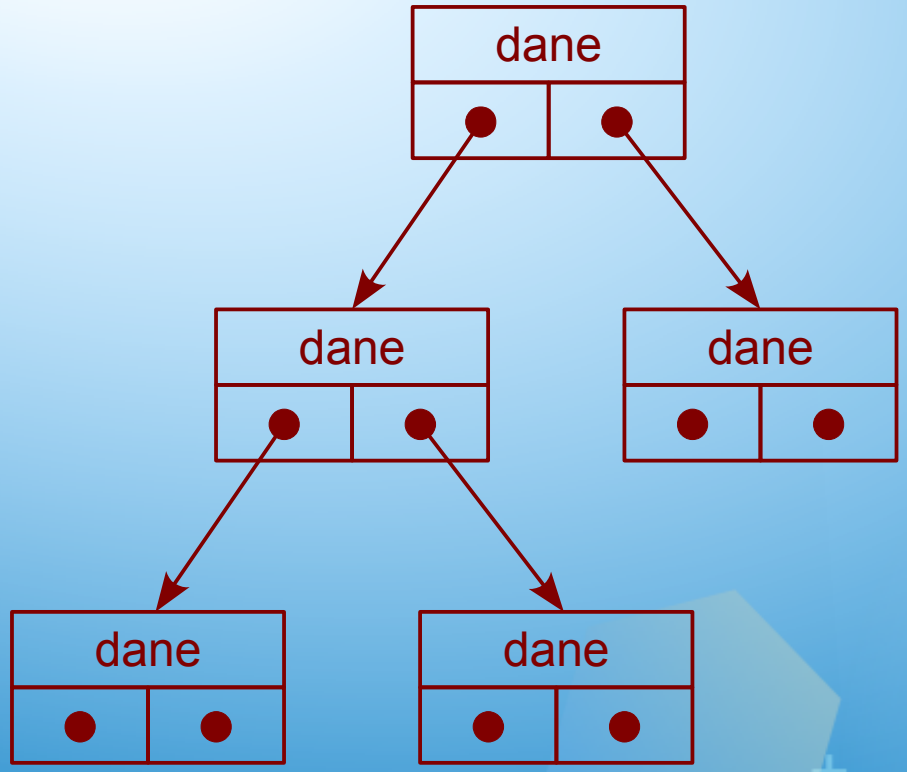
Należy pamiętać, że powyższe funkcje wykonane nieprawidłowo – bez uwzględnienia przypadków brzegowych – mogą powodować tzw. efekt **memory leak**.

Drzewa

Drzewo – to rodzaj kontenera danych, którego struktura logiczna jest odzwierciedleniem struktury drzewa matematycznego.

Drzewo – to rodzaj kontenera danych, którego struktura logiczna jest odzwierciedleniem struktury drzewa matematycznego.

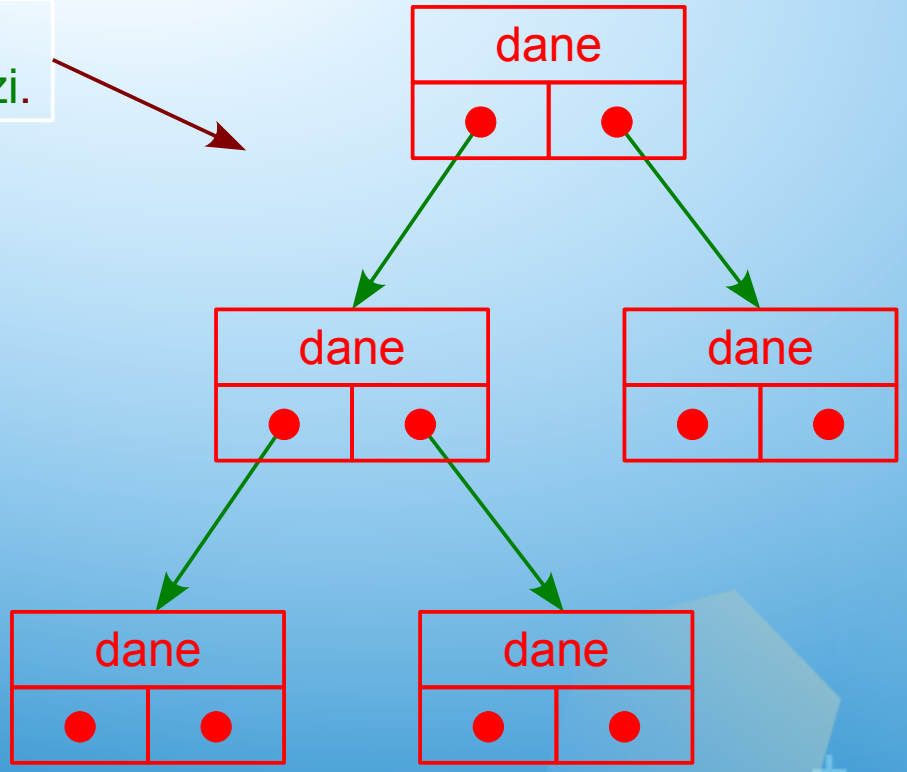
Budowa drzewa:



Drzewo – to rodzaj kontenera danych, którego struktura logiczna jest odzwierciedleniem struktury drzewa matematycznego.

Budowa drzewa:

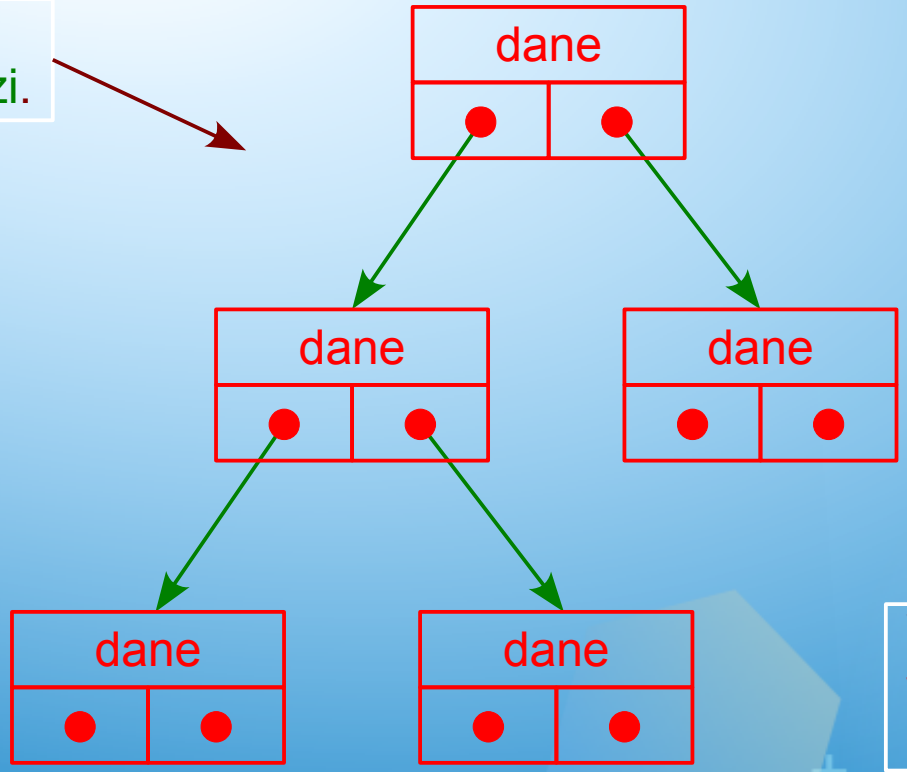
Drzewo,składa się z węzłów i krawędzi.



Drzewo – to rodzaj kontenera danych, którego struktura logiczna jest odzwierciedleniem struktury drzewa matematycznego.

Budowa drzewa:

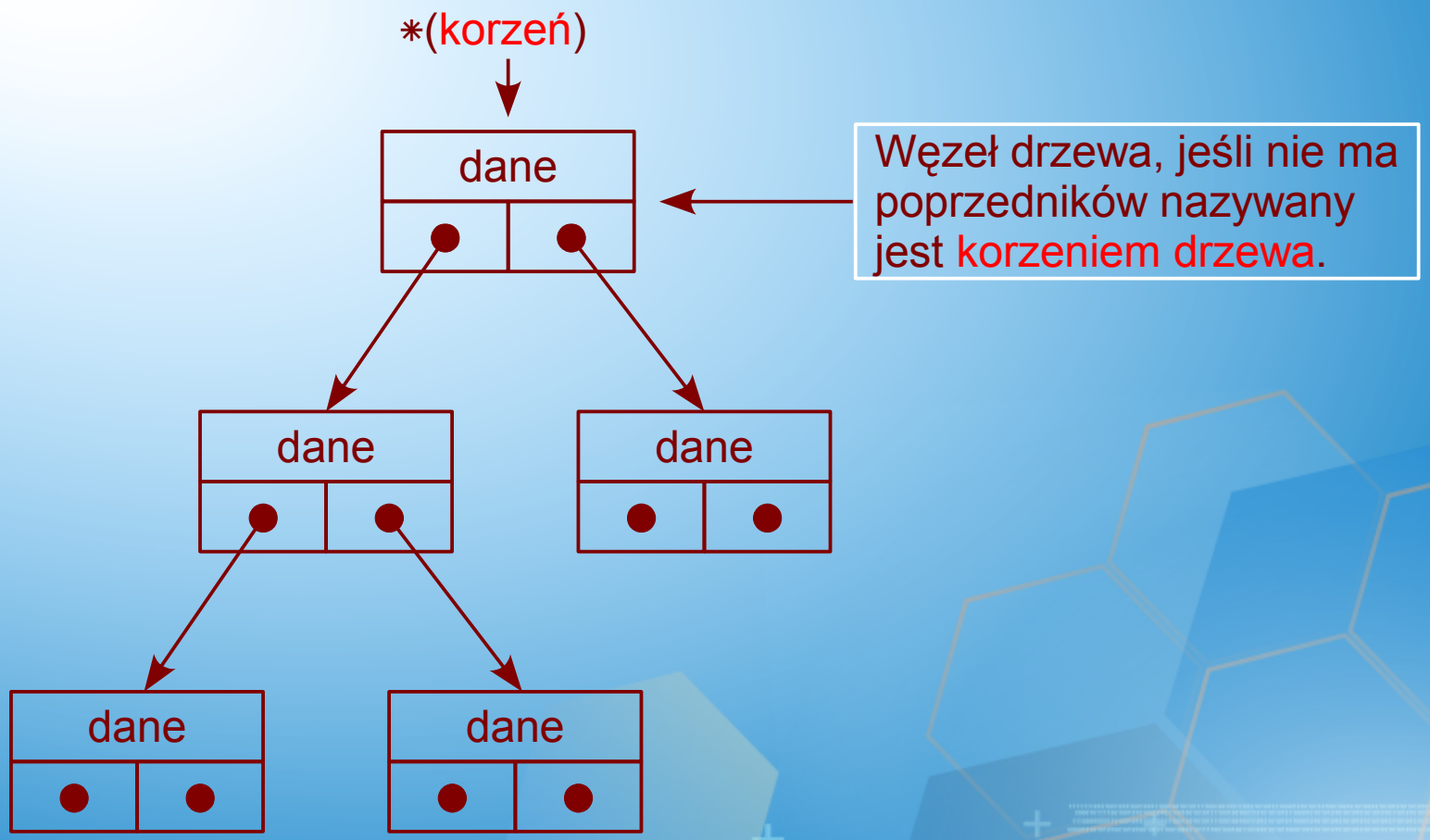
Drzewo,składa się z węzłów i krawędzi.



W ogólnym przypadku węzeł może wskazywać od **2** do ***n*** następników.

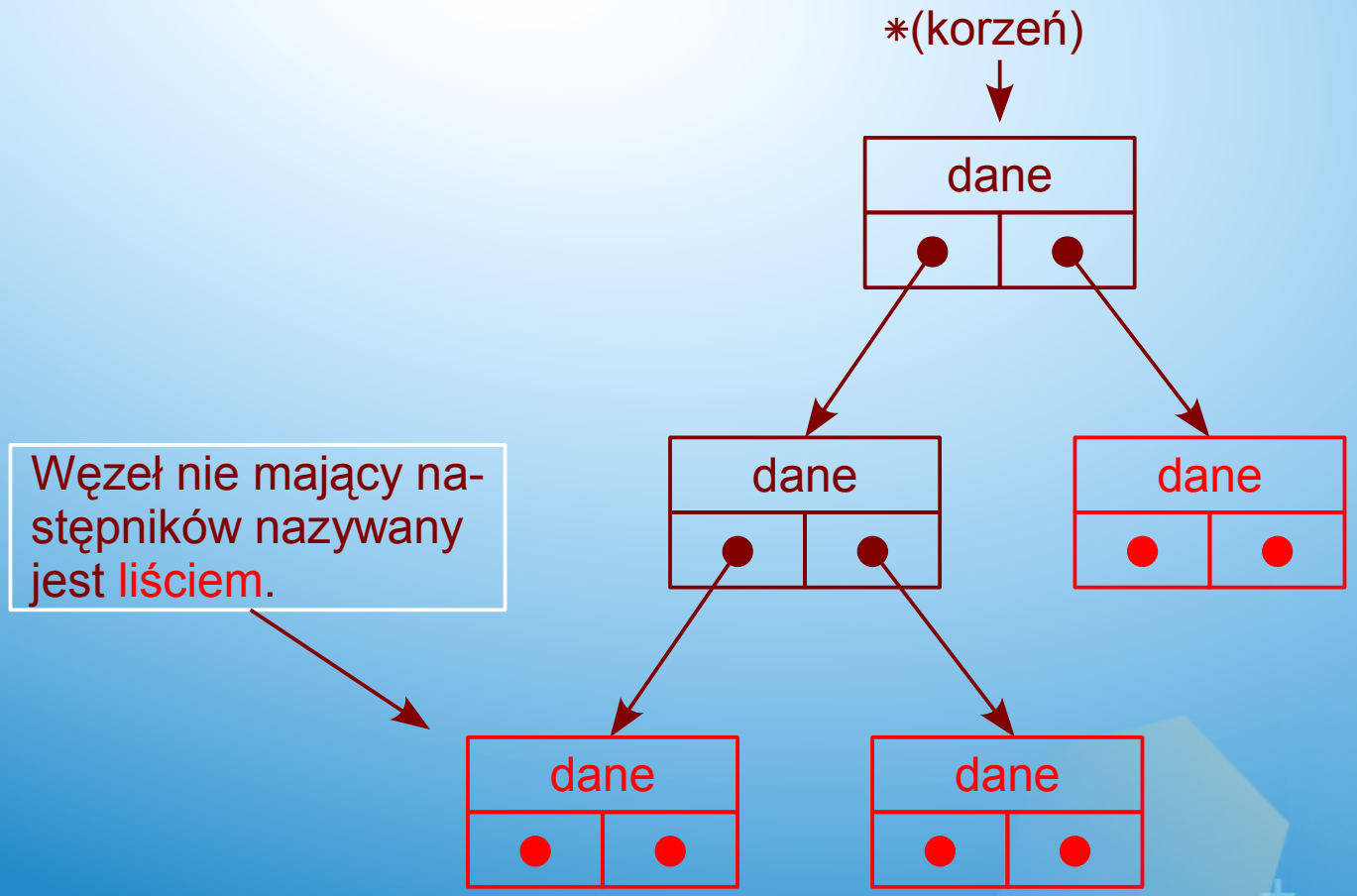
Drzewo – to rodzaj kontenera danych, którego struktura logiczna jest odzwierciedleniem struktury drzewa matematycznego.

Budowa drzewa:



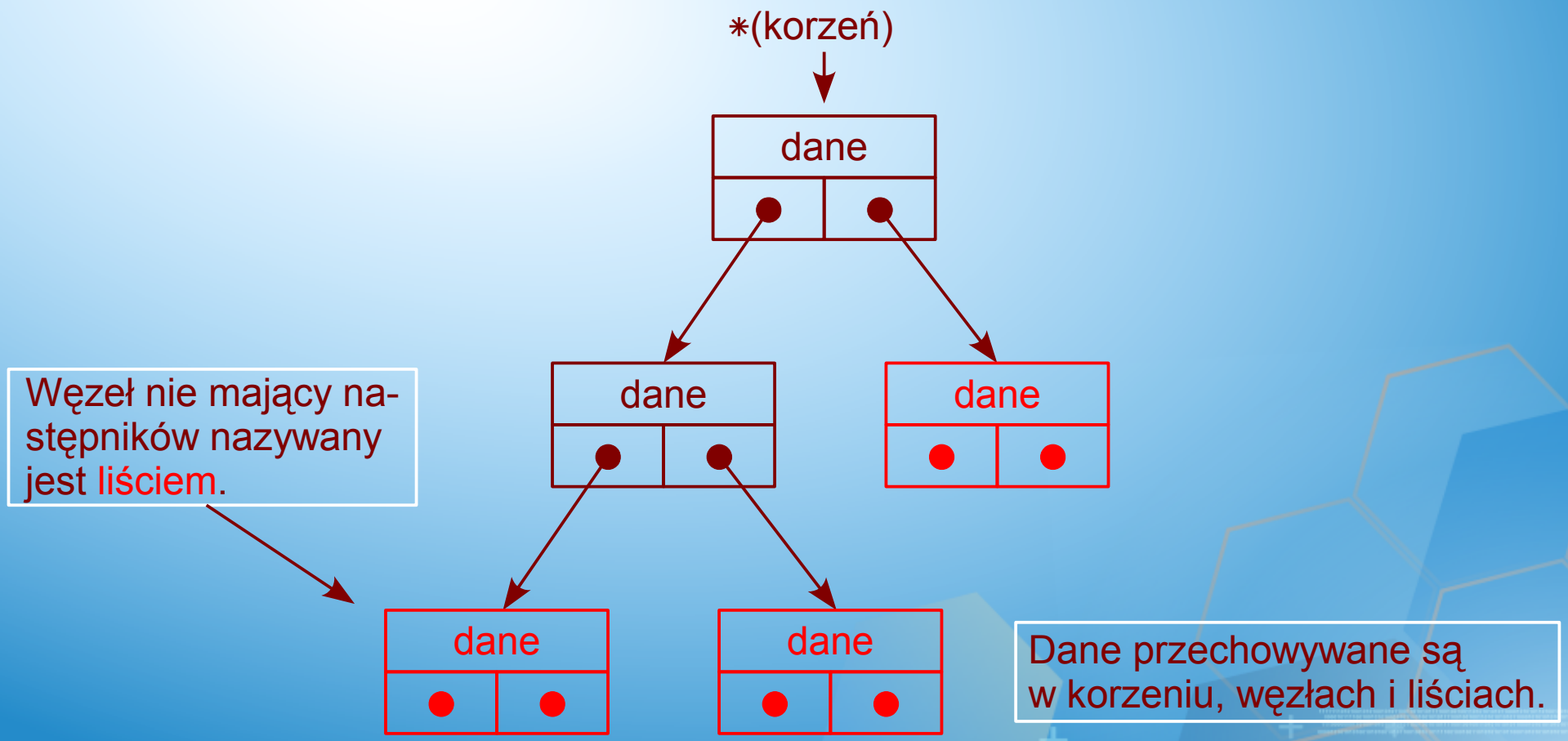
Drzewo – to rodzaj kontenera danych, którego struktura logiczna jest odzwierciedleniem struktury drzewa matematycznego.

Budowa drzewa:



Drzewo – to rodzaj kontenera danych, którego struktura logiczna jest odzwierciedleniem struktury drzewa matematycznego.

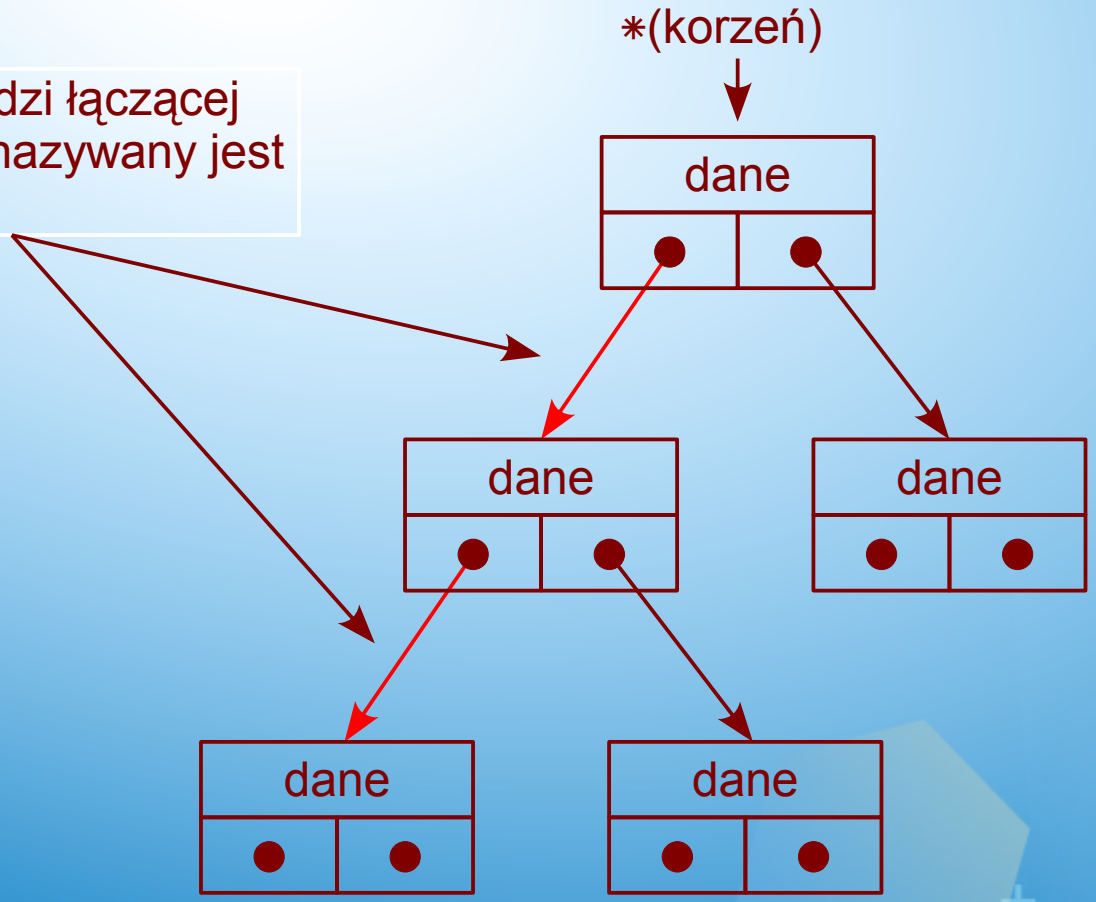
Budowa drzewa:



Drzewo – to rodzaj kontenera danych, którego struktura logiczna jest odzwierciedleniem struktury drzewa matematycznego.

Budowa drzewa:

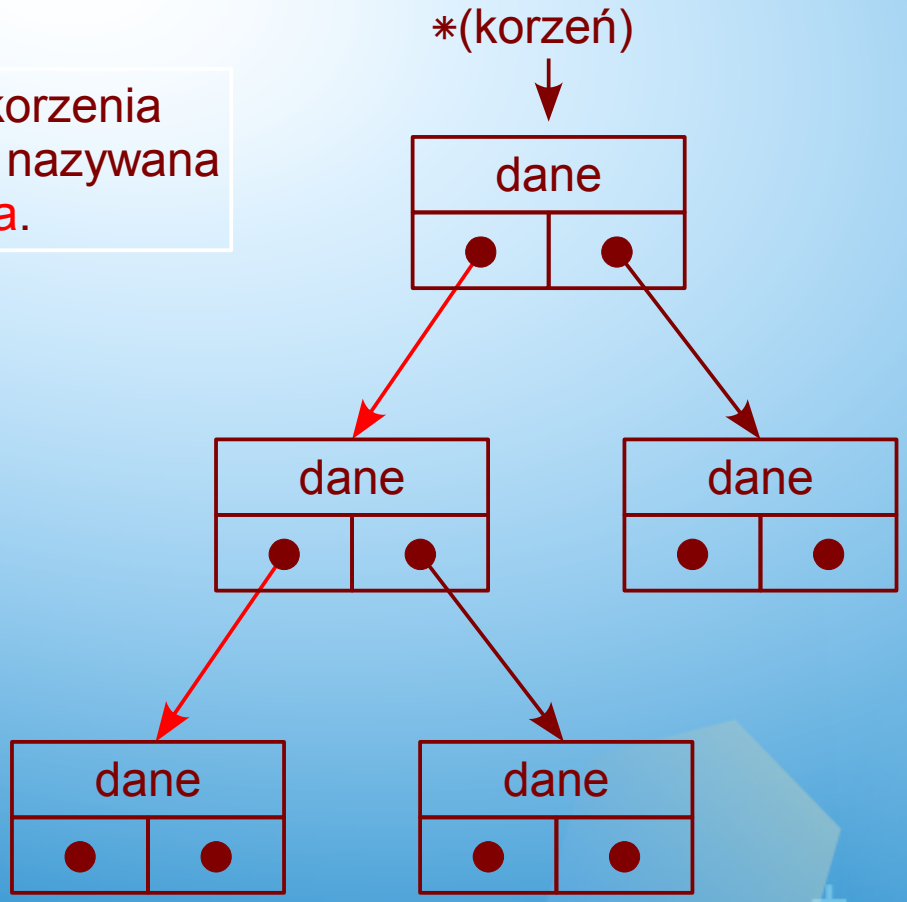
Ciąg krawędzi łączącej dwa węzły nazywany jest ścieżką.



Drzewo – to rodzaj kontenera danych, którego struktura logiczna jest odzwierciedleniem struktury drzewa matematycznego.

Budowa drzewa:

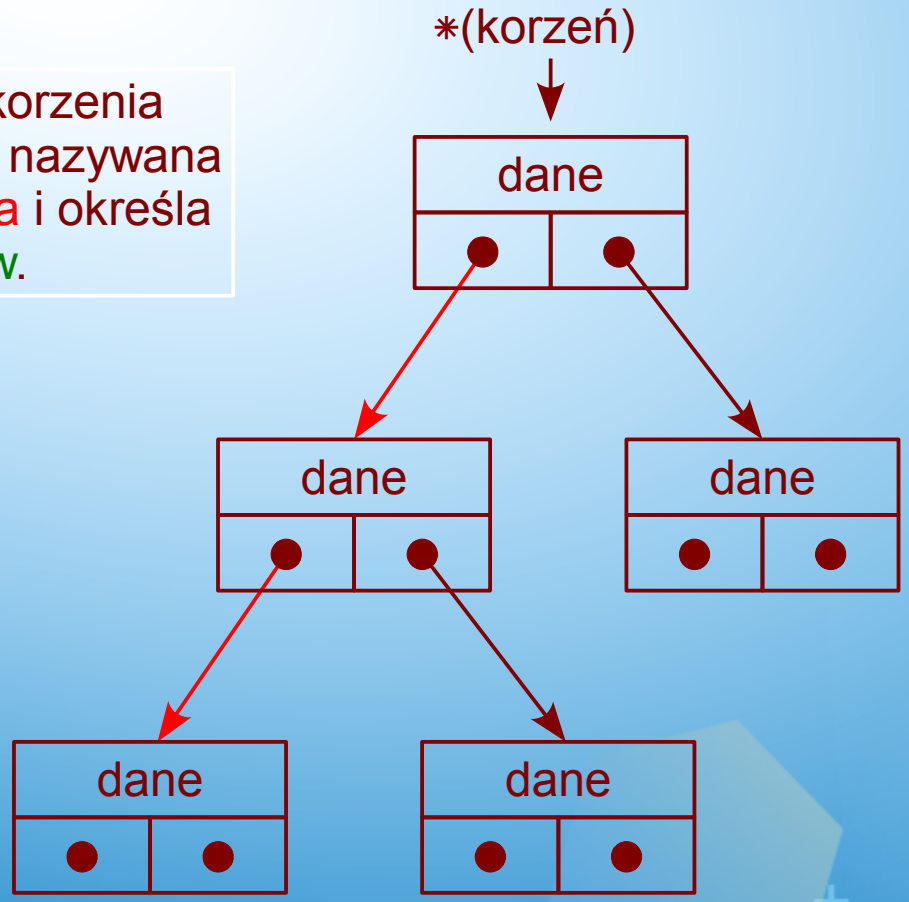
Liczba krawędzi od korzenia do dowolnego węzła nazywana jest długością drzewa.



Drzewo – to rodzaj kontenera danych, którego struktura logiczna jest odzwierciedleniem struktury drzewa matematycznego.

Budowa drzewa:

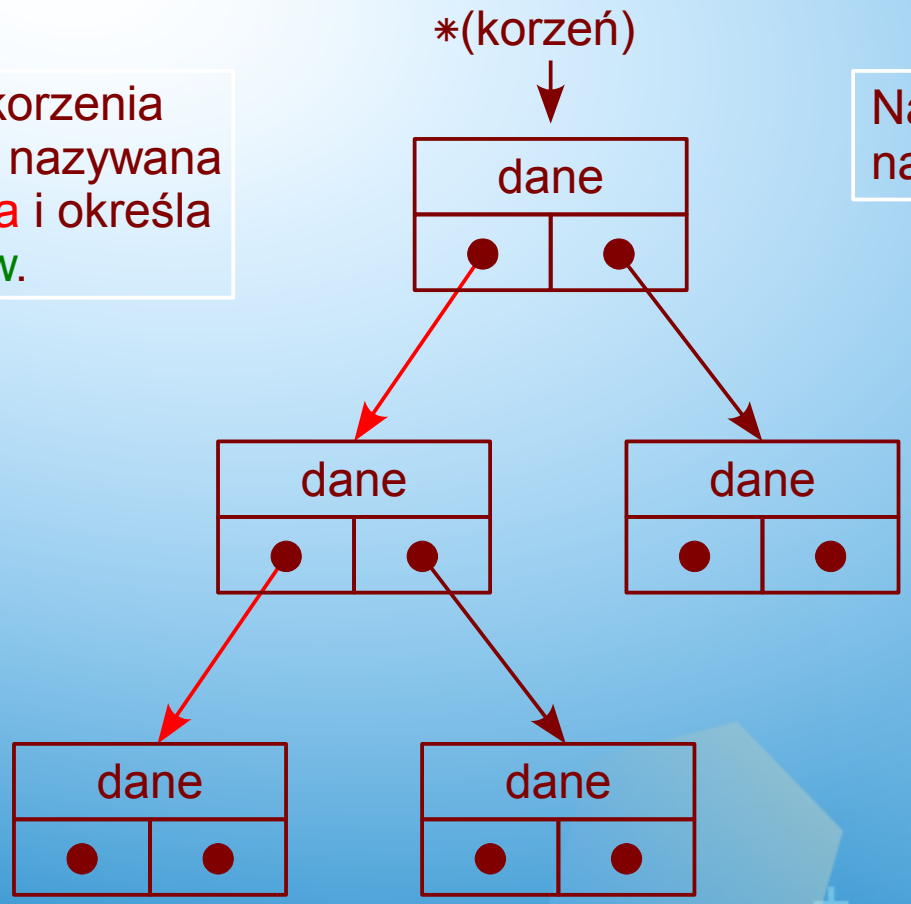
Liczba krawędzi od korzenia do dowolnego węzła nazywana jest długością drzewa i określa liczbę jego poziomów.



Drzewo – to rodzaj kontenera danych, którego struktura logiczna jest odzwierciedleniem struktury drzewa matematycznego.

Budowa drzewa:

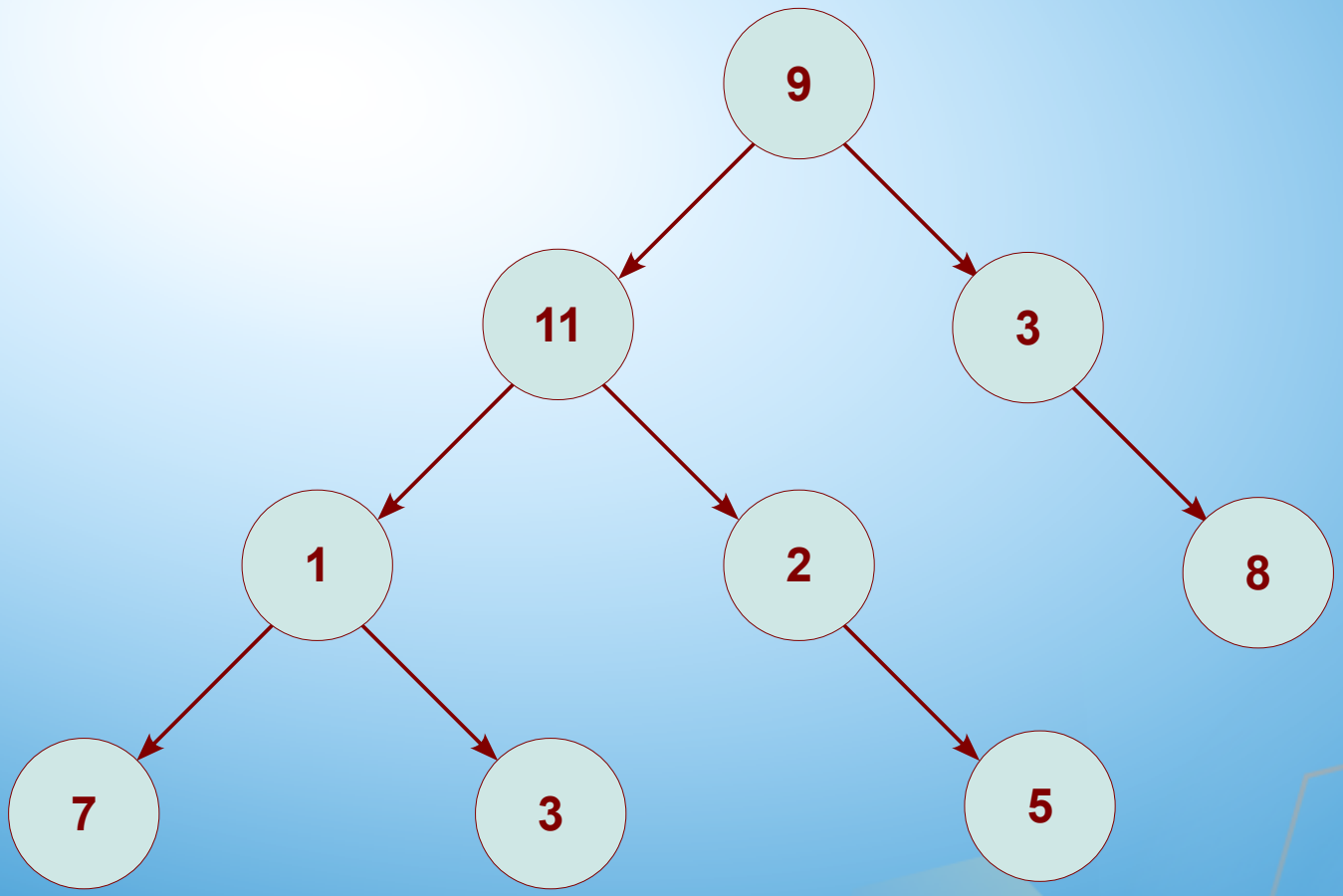
Liczba krawędzi od korzenia do dowolnego węzła nazywana jest długością drzewa i określa liczbę jego poziomów.



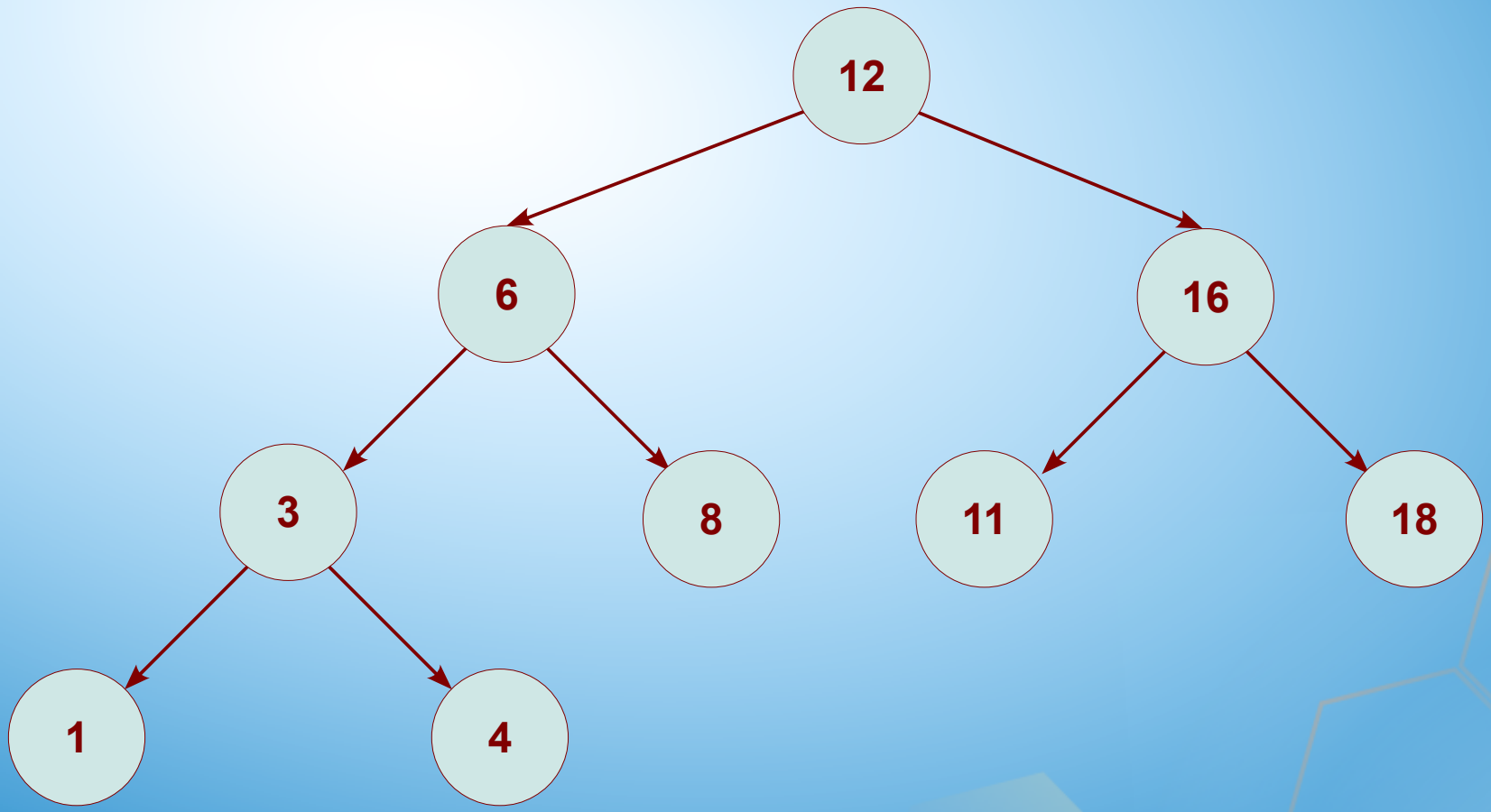
Największa długość drzewa nazywana jest **wysokością**.

Rodzaje drzew w informatyce

Drzewo binarne – jest to drzewo w którym stopień każdego węzła nie jest większy niż 3.



Drzewo binarne poszukiwań – jest to drzewo binarne w którym wartość klucza w węzłach lewego poddrzewa jest mniejsza niż w prawym poddrzewie.

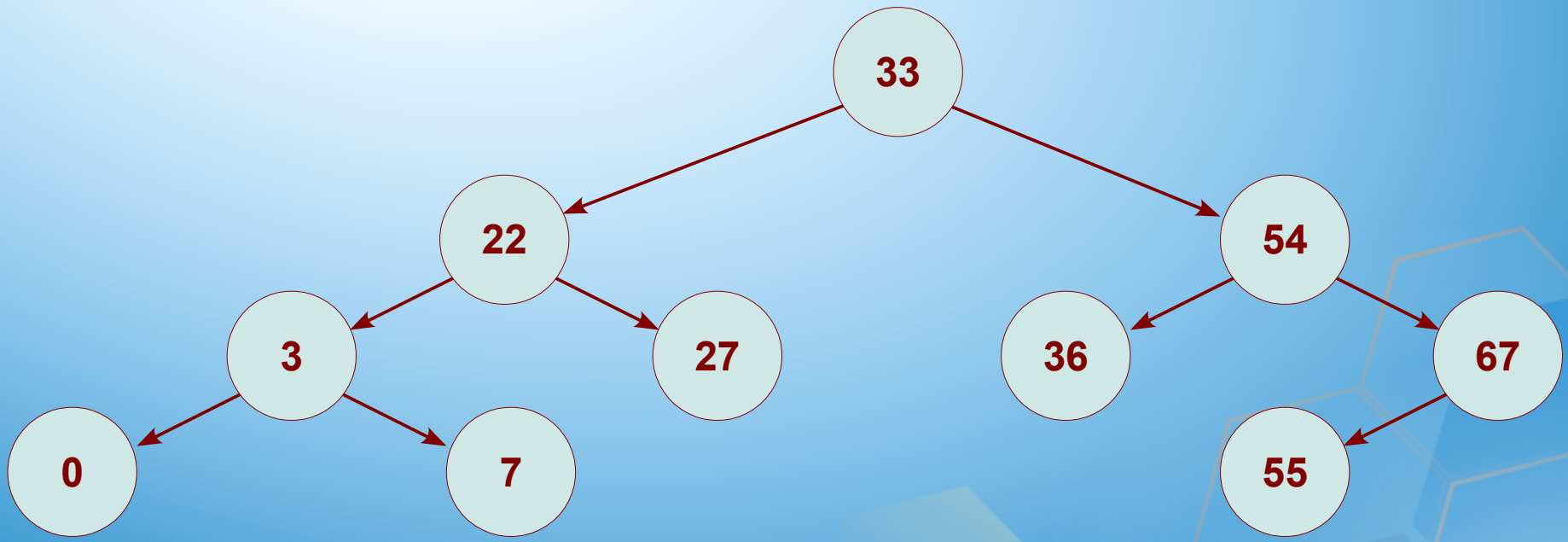


Złożoność wyszukiwania: O(log *n*)

Drzewo AVL – jest to zrównoważone drzewo binarne poszukiwań.

Właściwości:

- każdy węzeł posiada współczynnik wyważenia,
- różnica wysokości między lewym i prawym poddrzewem musi wynosić +1, 0 lub -1,
- jeśli dodanie lub usunięcie węzła spowoduje, że współczynnik wyważenia przyjmie wartość niedozwoloną, to wykonuje się rotacje węzłów, która przywraca zrównoważenie,
- pesymistyczna złożoność wyszukiwania wynosi $1.44(\log_2 n)$.



Zrównoważanie drzewa BST można uzyskać stosując algorytm DSW.

Drzewo czerwono-czarne – jest to samoorganizujące się drzewo binarne poszukiwań.

Zasady tworzenia:

- każdy węzeł jest czerwony lub czarny,
- korzeń jest czarny i każdy liść jest czarny (wskazanie puste traktuje się jak liść),
- jeśli węzeł jest czerwony, to jego potomkowie muszą być czarni,
- każda ścieżka z korzenia do wybranego liścia zawiera tyle samo czarnych węzłów.

Drzewo czerwono-czarne – jest to samoorganizujące się drzewo binarne poszukiwań.

Zasady tworzenia:

- każdy węzeł jest czerwony lub czarny,
- korzeń jest czarny i każdy liść jest czarny (wskazanie puste traktuje się jak liść),
- jeśli węzeł jest czerwony, to jego potomkowie muszą być czarni,
- każda ścieżka z korzenia do wybranego liścia zawiera tyle samo czarnych węzłów.

Właściwości:

- drzewo BR jest strukturą danych powiązaną z odpowiednimi algorytmami,
- operacje elementarne wymagają złożoności równej O(log *n*),
- dla *n* węzłów głębokość drzewa *h* wyniesie maksymalnie 2 log(*n*+1),
- **przywrócenie właściwości drzewa wymaga co najwyżej dwóch operacji rotacji**.

Drzewo czerwono-czarne – jest to samoorganizujące się drzewo binarne poszukiwań.

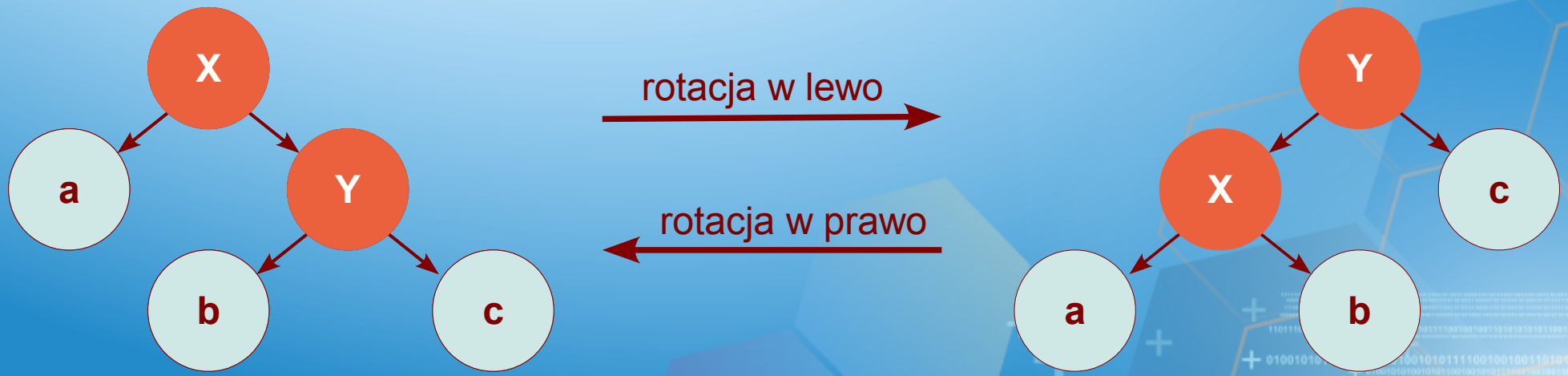
Zasady tworzenia:

- każdy węzeł jest czerwony lub czarny,
- korzeń jest czarny i każdy liść jest czarny (wskazanie puste traktuje się jak liść),
- jeśli węzeł jest czerwony, to jego potomkowie muszą być czarni,
- każda ścieżka z korzenia do wybranego liścia zawiera tyle samo czarnych węzłów.

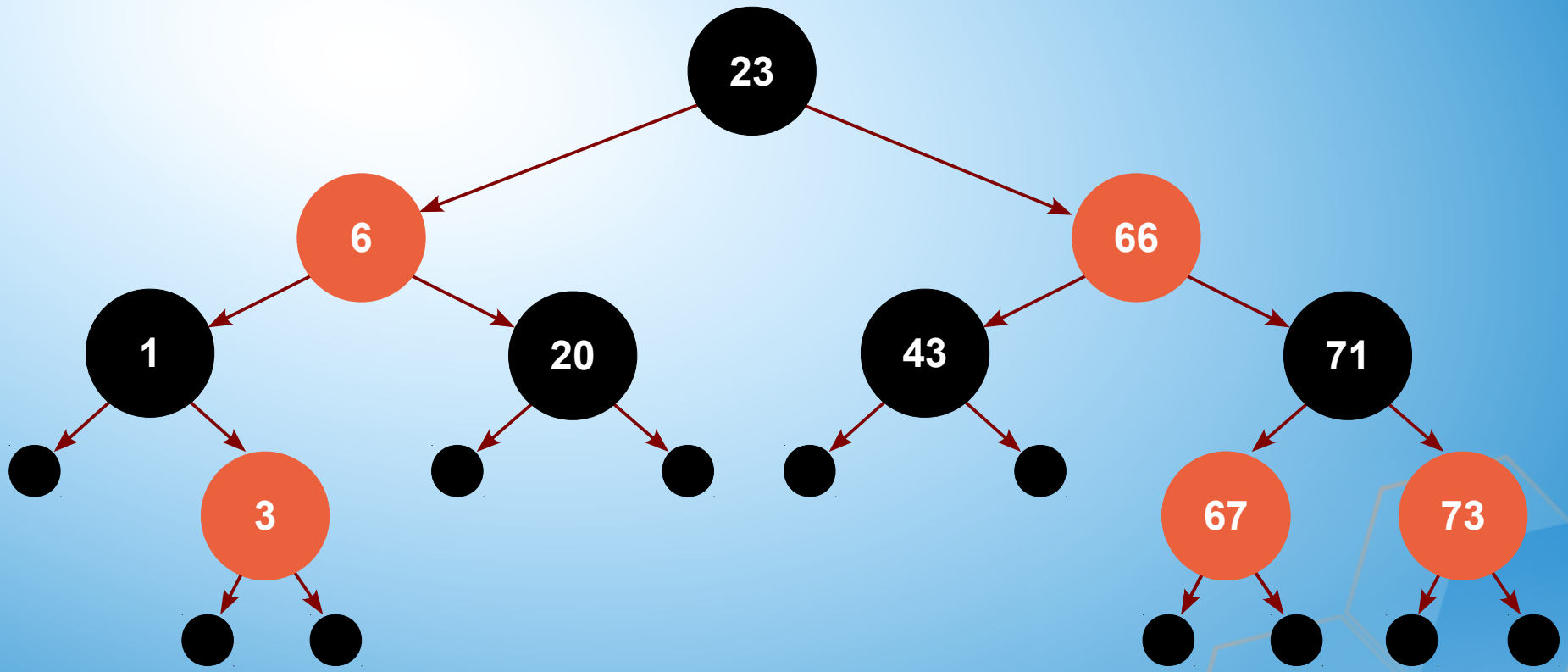
Właściwości:

- drzewo BR jest strukturą danych powiązaną z odpowiednimi algorytmami,
- operacje elementarne wymagają złożoności równej O(log *n*),
- dla *n* węzłów głębokość drzewa *h* wyniesie maksymalnie 2 log(*n*+1),
- **przywrócenie właściwości drzewa wymaga co najwyżej dwóch operacji rotacji**.

Operacja rotacji:



Drzewo czerwono-czarne – jest to samoorganizujące się drzewo binarne poszukiwań.



Drzewo BSP – ang. Binary Space Partition – jest to rodzaj drzewa binarnego stosowanego do przechowywania informacji o kształcie wielokąta.

Zastosowanie:

- przechowywanie informacji o wielokącie,
- upraszczanie detekcji przynależności punktu do wielokąta,
- upraszczanie wykonywanie operacji logicznych na siatkach brył,
- przechowywanie informacji o scenie trójwymiarowej, co pozwala na uproszczenie:
  - detekcji kolizji,
  - śledzenia promienia (ang. ray tracing),
  - usuwania niewidocznych powierzchni.

Drzewo BSP – ang. Binary Space Partition – jest to rodzaj drzewa binarnego stosowanego do przechowywania informacji o kształcie wielokąta.

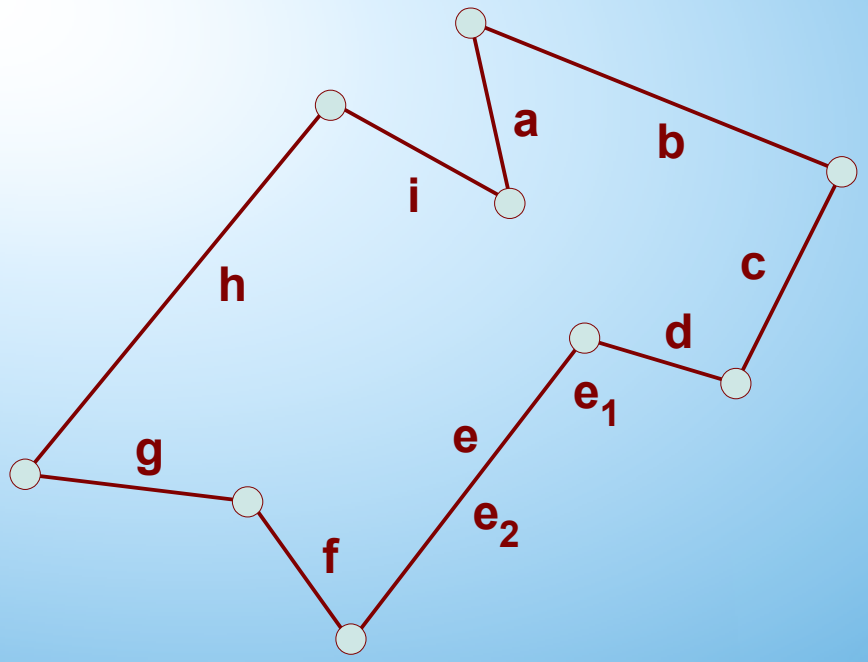
Zastosowanie:

- przechowywanie informacji o wielokącie,
- upraszczanie detekcji przynależności punktu do wielokąta,
- upraszczanie wykonywanie operacji logicznych na siatkach brył,
- przechowywanie informacji o scenie trójwymiarowej, co pozwala na uproszczenie:
  - detekcji kolizji,
  - śledzenia promienia (ang. ray tracing),
  - usuwania niewidocznych powierzchni.

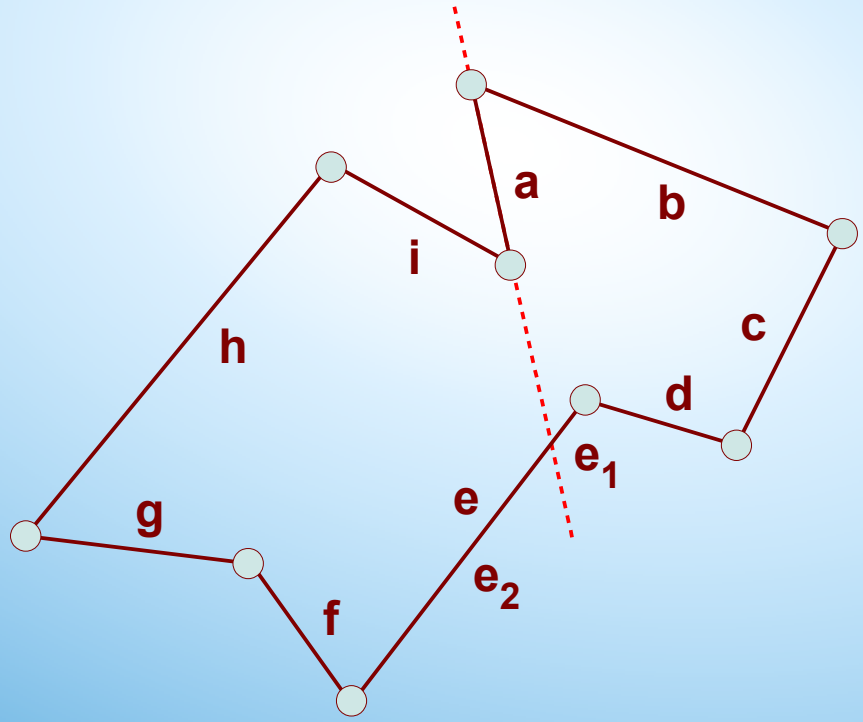
Tworzenie drzewa:

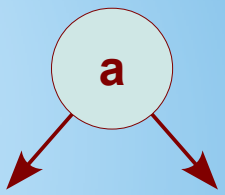
- wielokąt (bryłę) należy podzielić na rozłączne części przy pomocy hiperpłaszczyzn – prosta w 3D, płaszczyzna w 3D,
- do danego węzła dodaje się obiekt leżący na hiperpłaszczyźnie, następnie:
  - w lewym poddrzewie znajdują się obiekty w całości znajdujące się po odpowiedniej stronie hiperpłaszczyzny,
  - w prawym poddrzewie znajdują się obiekty w całości znajdujące się po drugiej stronie hiperpłaszczyzny niż w lewym poddrzewie,
  - jeśli obiektu nie można zakwalifikować do żadnego z poddrzew to należy go podzielić – najczęściej z wielokątów wydzielane są trójkąty.
- proces dodawania obiektów do węzłów wykonuje się rekurencyjnie.

Drzewo BSP – ang. Binary Space Partition – jest to rodzaj drzewa binarnego stosowanego do przechowywania informacji o kształcie wielokąta.

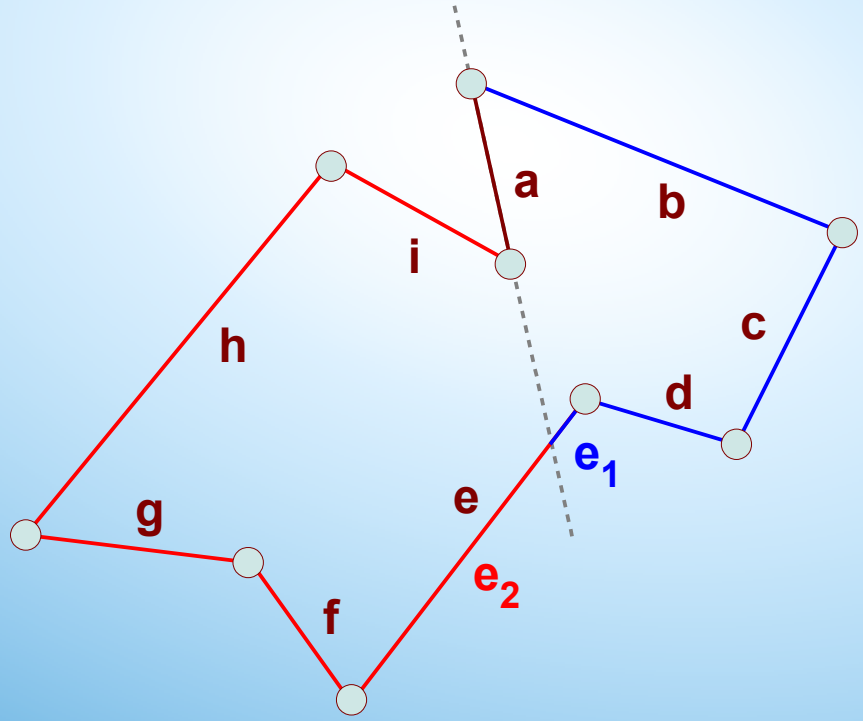


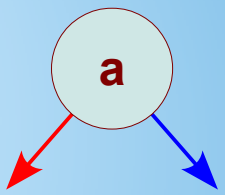
Drzewo BSP – ang. Binary Space Partition – jest to rodzaj drzewa binarnego stosowanego do przechowywania informacji o kształcie wielokąta.



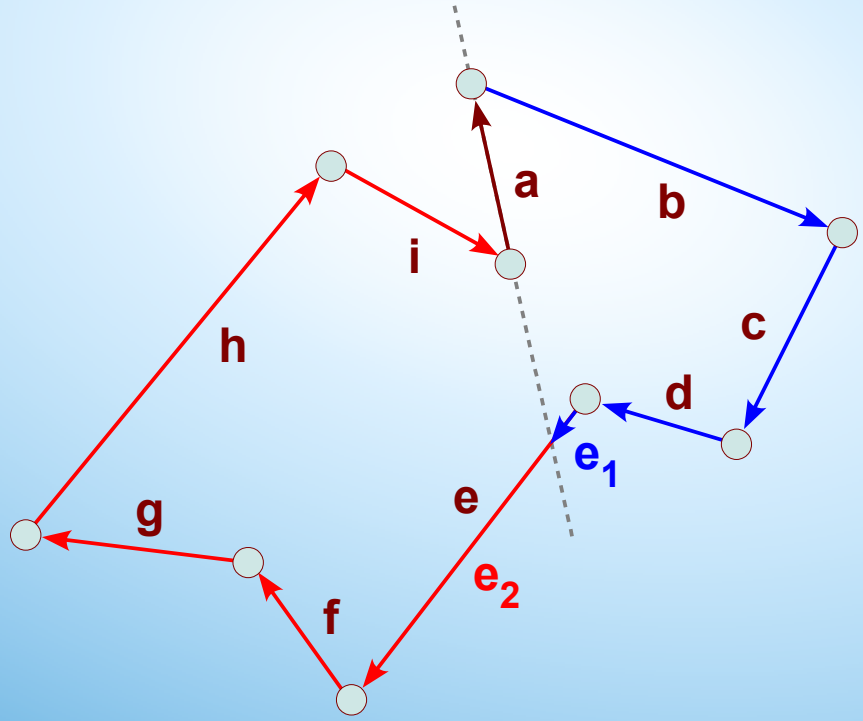


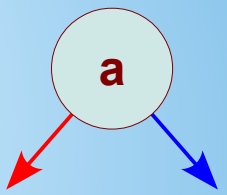
Drzewo BSP – ang. Binary Space Partition – jest to rodzaj drzewa binarnego stosowanego do przechowywania informacji o kształcie wielokąta.



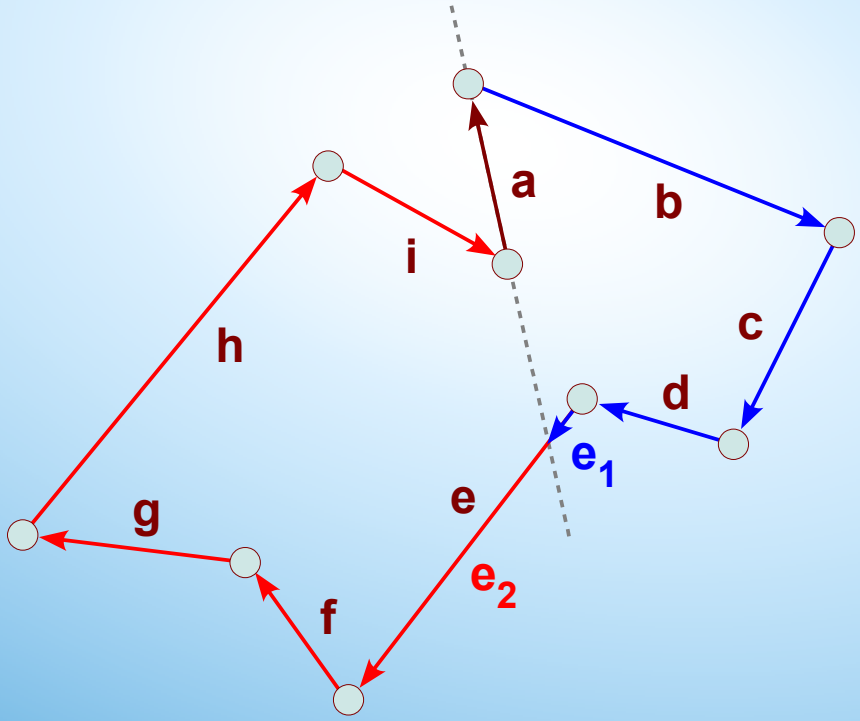


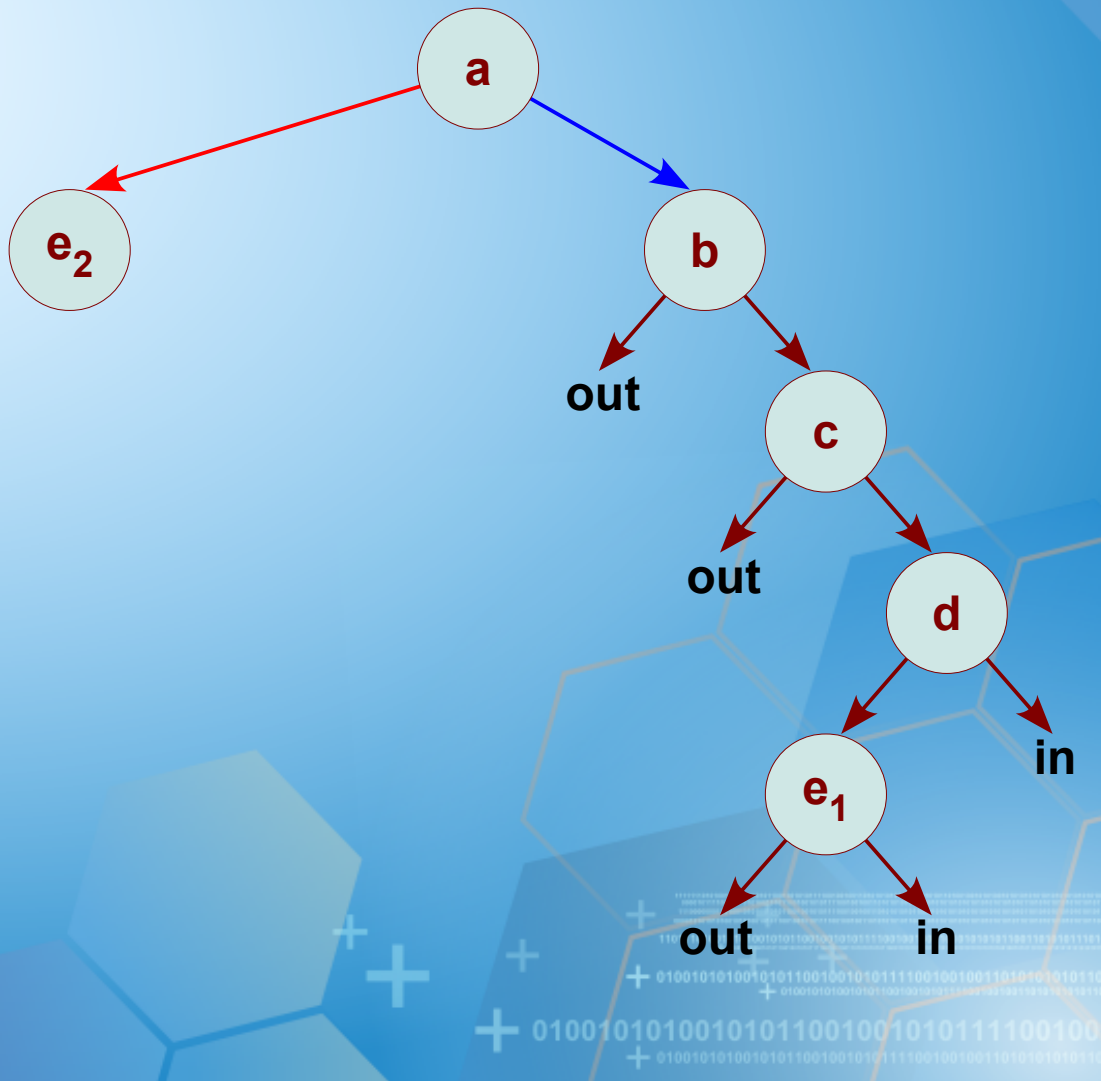
Drzewo BSP – ang. Binary Space Partition – jest to rodzaj drzewa binarnego stosowanego do przechowywania informacji o kształcie wielokąta.



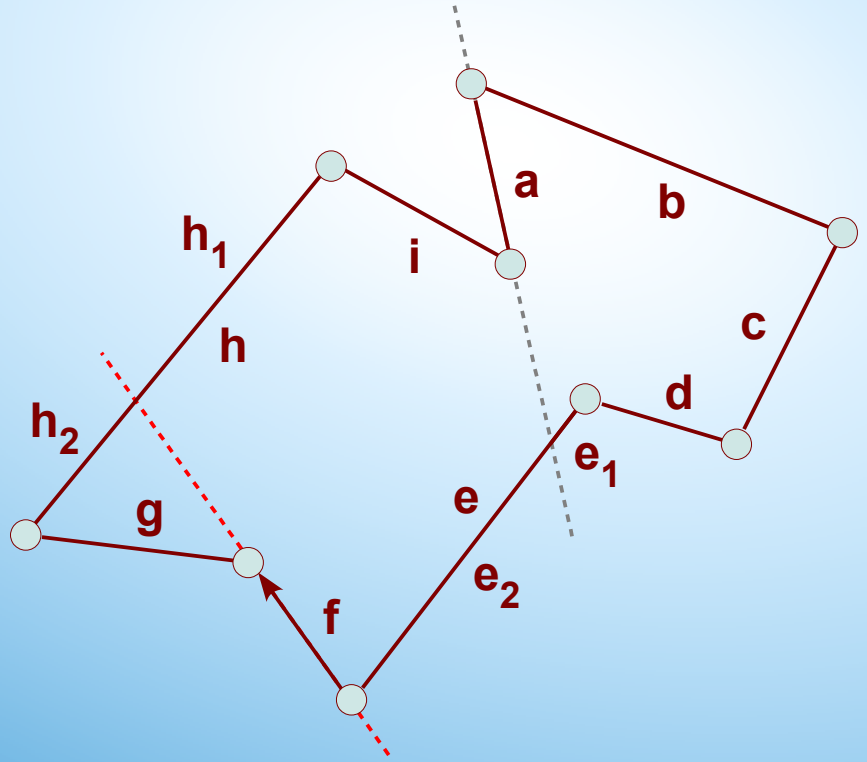


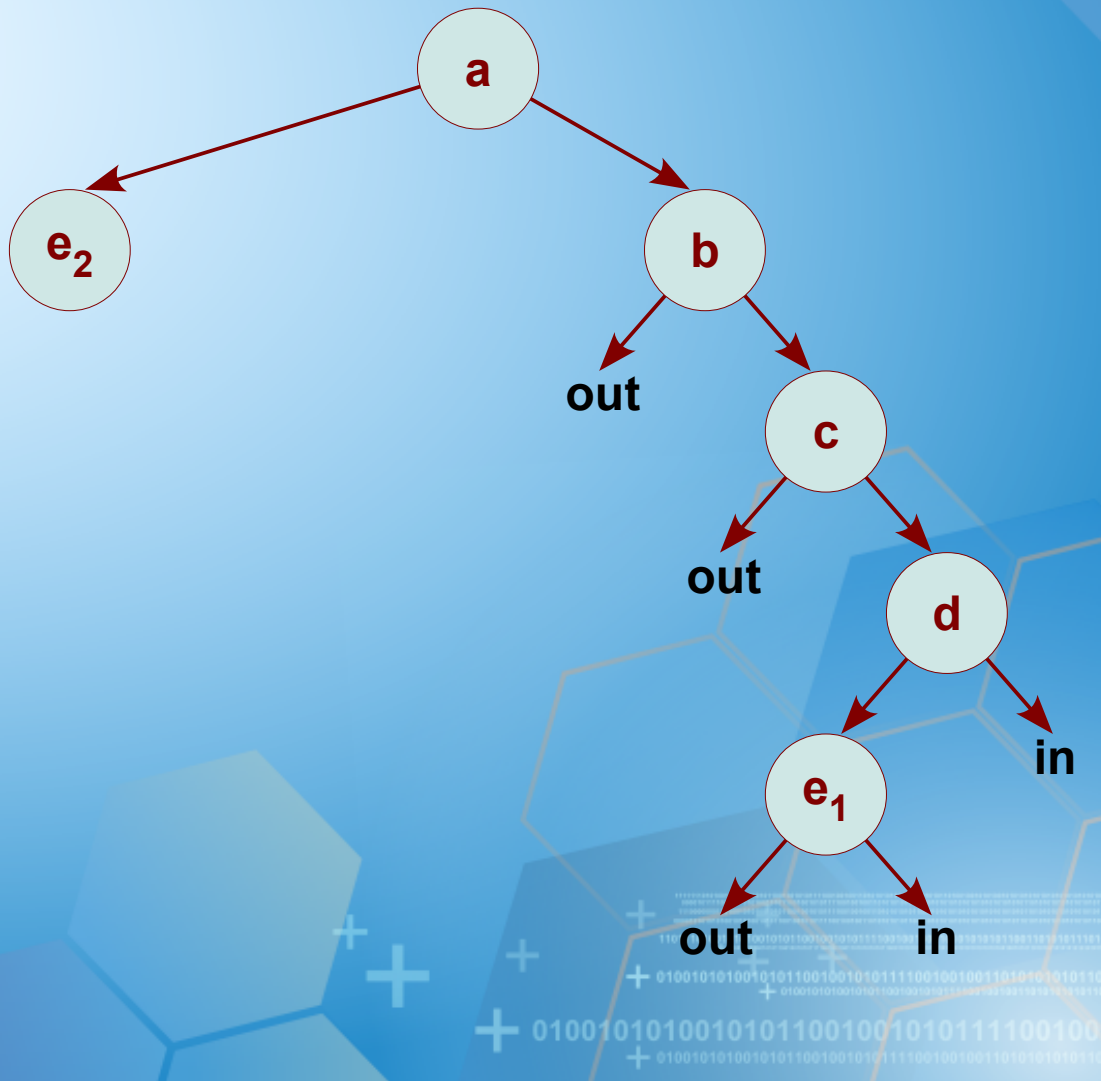
Drzewo BSP – ang. Binary Space Partition – jest to rodzaj drzewa binarnego stosowanego do przechowywania informacji o kształcie wielokąta.



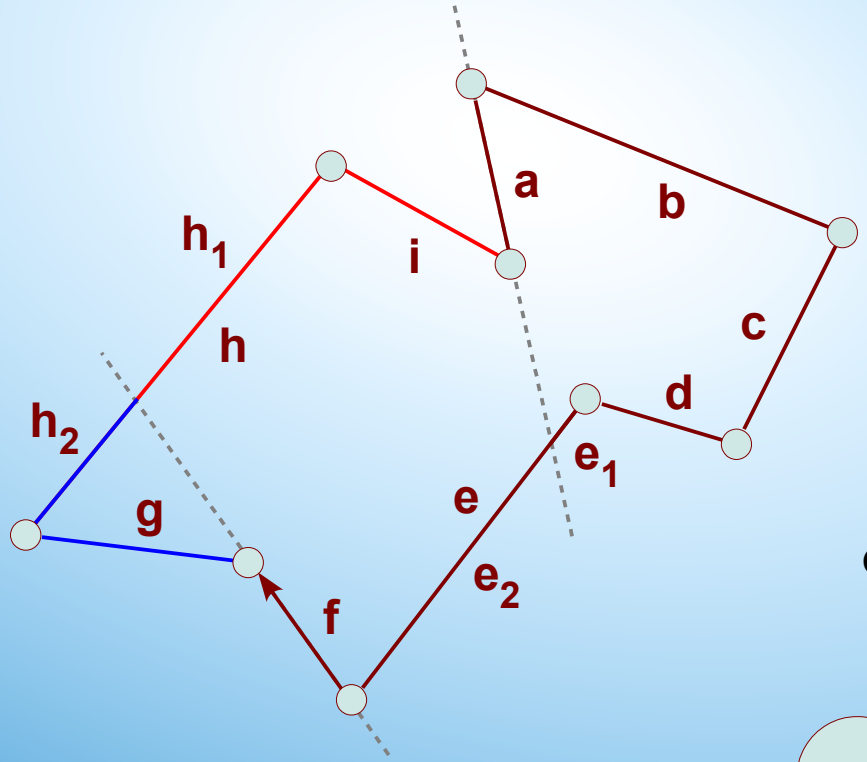


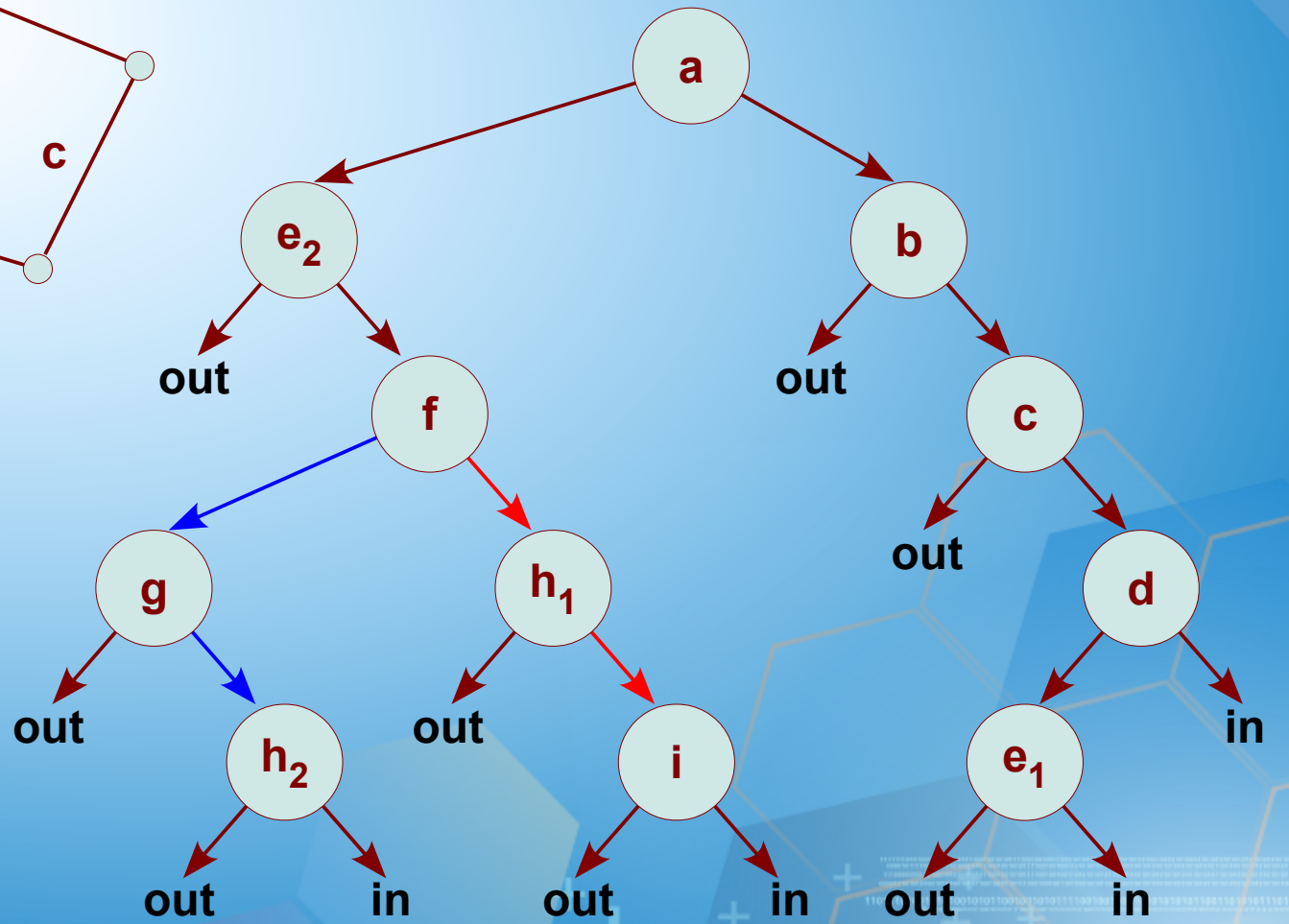
Drzewo BSP – ang. Binary Space Partition – jest to rodzaj drzewa binarnego stosowanego do przechowywania informacji o kształcie wielokąta.





Drzewo BSP – ang. Binary Space Partition – jest to rodzaj drzewa binarnego stosowanego do przechowywania informacji o kształcie wielokąta.





Drzewo ósemkowe – ang. octree, jest to rodzaj drzewa, w którym każdy węzeł ma osiem następników.

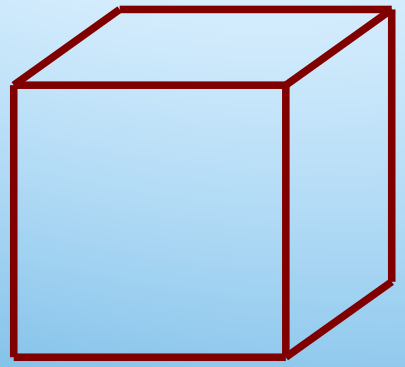
Zastosowanie:

- stosowane do podziału przestrzeni na regularne części,
- jako efektywna metoda przechowywania voxeli,
- do wykrywania kolizji obiektów,
- w kwantyzacji kolorów,
- w renderingu do usuwania obiektów niewidocznych.

Drzewo ósemkowe – ang. octree, jest to rodzaj drzewa, w którym każdy węzeł ma osiem następników.

Zastosowanie:

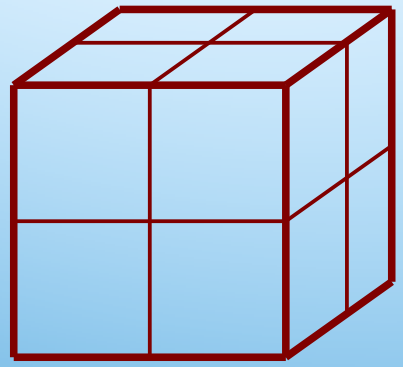
- stosowane do podziału przestrzeni na regularne części,
- jako efektywna metoda przechowywania voxeli,
- do wykrywania kolizji obiektów,
- w kwantyzacji kolorów,
- w renderingu do usuwania obiektów niewidocznych.

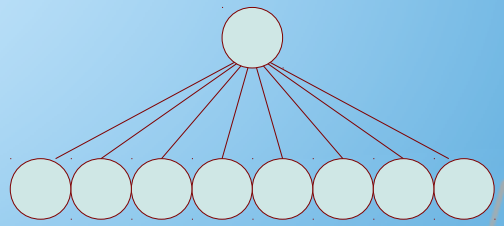


Drzewo ósemkowe – ang. octree, jest to rodzaj drzewa, w którym każdy węzeł ma osiem następników.

Zastosowanie:

- stosowane do podziału przestrzeni na regularne części,
- jako efektywna metoda przechowywania voxeli,
- do wykrywania kolizji obiektów,
- w kwantyzacji kolorów,
- w renderingu do usuwania obiektów niewidocznych.

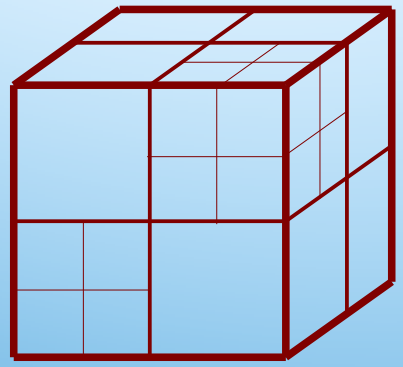


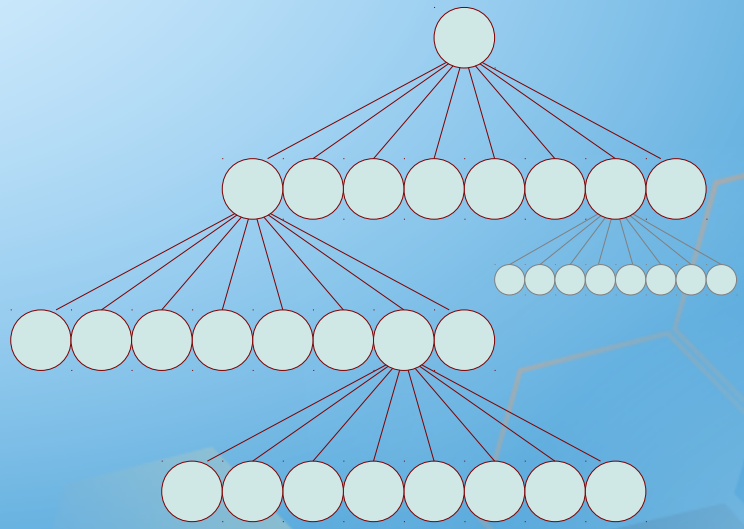


Drzewo ósemkowe – ang. octree, jest to rodzaj drzewa, w którym każdy węzeł ma osiem następników.

Zastosowanie:

- stosowane do podziału przestrzeni na regularne części,
- jako efektywna metoda przechowywania voxeli,
- do wykrywania kolizji obiektów,
- w kwantyzacji kolorów,
- w renderingu do usuwania obiektów niewidocznych.

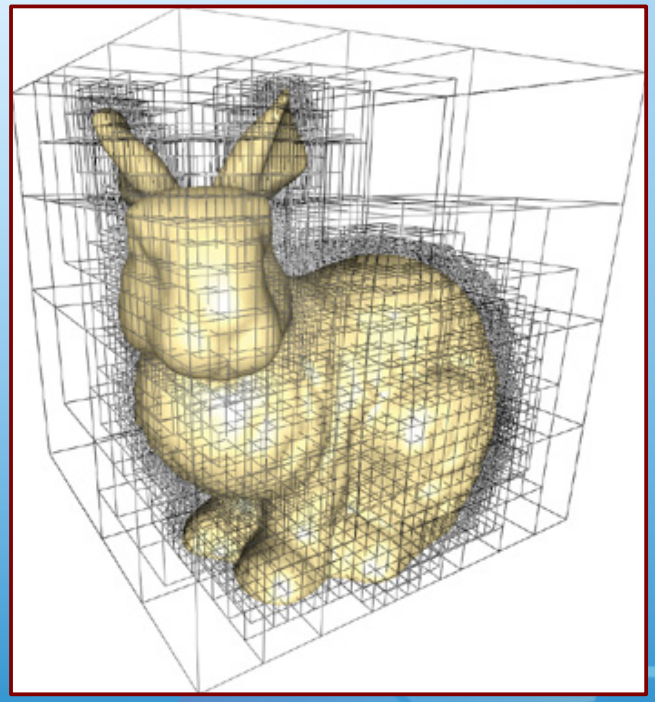




Drzewo ósemkowe – ang. octree, jest to rodzaj drzewa, w którym każdy węzeł ma osiem następników.

Zastosowanie:

- stosowane do podziału przestrzeni na regularne części,
- jako efektywna metoda przechowywania voxeli,
- do wykrywania kolizji obiektów,
- w kwantyzacji kolorów,
- w renderingu do usuwania obiektów niewidocznych.



Kopiec – ang. heap, jest to rodzaj drzewa spełniającego tzw. warunek kopca – każdy potomek jest w stałej relacji ze swoim rodzicem.

Kopiec – ang. heap, jest to rodzaj drzewa spełniającego tzw. warunek kopca – każdy potomek jest w stałej relacji ze swoim rodzicem.

Kopiec binarny zupełny:

- zbudowany jest w oparciu o drzewo zupełne,
- na poziomie *n* jest jest 2*n* węzłów (poza ostatnim poziomem) – łatwa implementacja w postaci tablicy,
- wysokość kopca jest logarytmiczna względem jego wielkości.

Kopiec – ang. heap, jest to rodzaj drzewa spełniającego tzw. warunek kopca – każdy potomek jest w stałej relacji ze swoim rodzicem.

Kopiec binarny zupełny:

- zbudowany jest w oparciu o drzewo zupełne,
- na poziomie *n* jest jest 2*n* węzłów (poza ostatnim poziomem) – łatwa implementacja w postaci tablicy,
- wysokość kopca jest logarytmiczna względem jego wielkości.

Drzewo jest zupełne jeśli wszystkie jego poziomy z wyjątkiem ostatniego są zapełnione, a na ostatnim poziomie liści są spójnie ułożone wg relacji kopca.

Kopiec – ang. heap, jest to rodzaj drzewa spełniającego tzw. warunek kopca – każdy potomek jest w stałej relacji ze swoim rodzicem.

Kopiec binarny zupełny:

- zbudowany jest w oparciu o drzewo zupełne,
- na poziomie *n* jest jest 2*n* węzłów (poza ostatnim poziomem) – łatwa implementacja w postaci tablicy,
- wysokość kopca jest logarytmiczna względem jego wielkości.

Drzewo jest zupełne jeśli wszystkie jego poziomy z wyjątkiem ostatniego są zapełnione, a na ostatnim poziomie liści są spójnie ułożone wg relacji kopca.

Kolejnym elementom drzewa nadaje się numery (od 1), zaczynając od szczytu, a następnie od lewej do prawej w każdym z poziomów, wtedy:
- jeśli potomek na numer *n*, to jego rodzic *n*/2,
- jeśli rodzic ma numer *n*, to jego lewy potomek ma numer 2*n*, a prawy 2*n*+1.

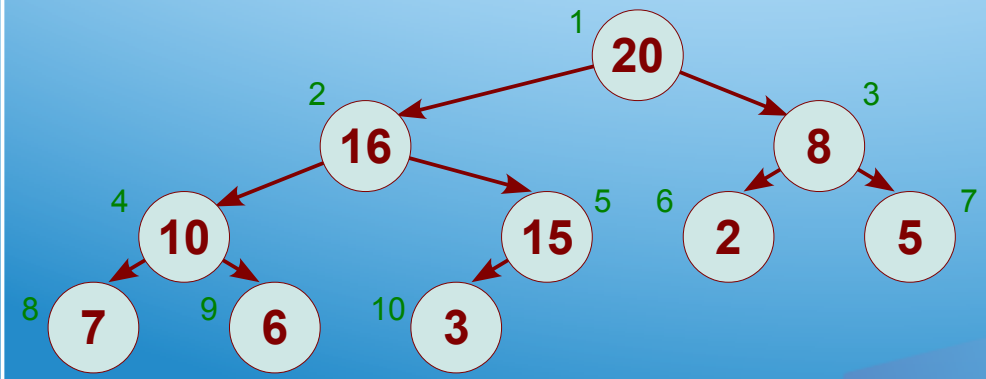
Kopiec – ang. heap, jest to rodzaj drzewa spełniającego tzw. warunek kopca – każdy potomek jest w stałej relacji ze swoim rodzicem.

Kopiec binarny zupełny:

- zbudowany jest w oparciu o drzewo zupełne,
- na poziomie *n* jest jest 2*n* węzłów (poza ostatnim poziomem) – łatwa implementacja w postaci tablicy,
- wysokość kopca jest logarytmiczna względem jego wielkości.

Drzewo jest zupełne jeśli wszystkie jego poziomy z wyjątkiem ostatniego są zapełnione, a na ostatnim poziomie liści są spójnie ułożone wg relacji kopca.

Kolejnym elementom drzewa nadaje się numery (od 1), zaczynając od szczytu, a następnie od lewej do prawej w każdym z poziomów, wtedy:
- jeśli potomek na numer *n*, to jego rodzic *n*/2,
- jeśli rodzic ma numer *n*, to jego lewy potomek ma numer 2*n*, a prawy 2*n*+1.



nr	1	2	3	4	5	6	7	8	9	10
W	20	16	8	10	15	2	5	7	6	3

Kopiec – ang. heap, jest to rodzaj drzewa spełniającego tzw. warunek kopca – każdy potomek jest w stałej relacji ze swoim rodzicem.

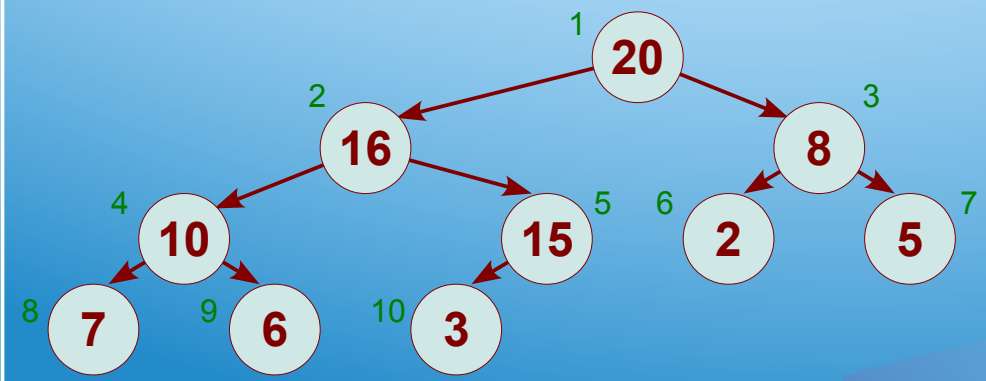
Kopiec binarny zupełny:

- zbudowany jest w oparciu o drzewo zupełne,
- na poziomie *n* jest jest 2*n* węzłów (poza ostatnim poziomem) – łatwa implementacja w postaci tablicy,
- wysokość kopca jest logarytmiczna względem jego wielkości.

Drzewo jest zupełne jeśli wszystkie jego poziomy z wyjątkiem ostatniego są zapełnione, a na ostatnim poziomie liści są spójnie ułożone wg relacji kopca.

Dla kopca binarnego operacje wstawiania i usuwania mają złożoność czasową równą O(log *n*).

Kolejnym elementom drzewa nadaje się numery (od 1), zaczynając od szczytu, a następnie od lewej do prawej w każdym z poziomów, wtedy:
- jeśli potomek na numer *n*, to jego rodzic *n*/2,
- jeśli rodzic ma numer *n*, to jego lewy potomek ma numer 2*n*, a prawy 2*n*+1.



nr	1	2	3	4	5	6	7	8	9	10
W	20	16	8	10	15	2	5	7	6	3

Koniec wykładu