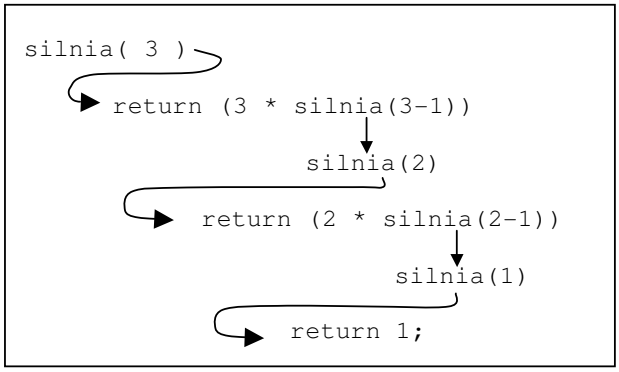
Wybrane algorytmy i struktury danych - implementacja w języku C

1. Rekurencyjne wywoływanie funkcji

Rekurencyjne wywołanie funkcji – funkcja wywołuje samą siebie, by rozwiązać część problemu (podproblem). Metoda taka obciąża stos programu – należy uważać, by nie wywoływać rekurencji ponad możliwości stosu.

Przykład rekurencyjnej funkcji do liczenia silni:

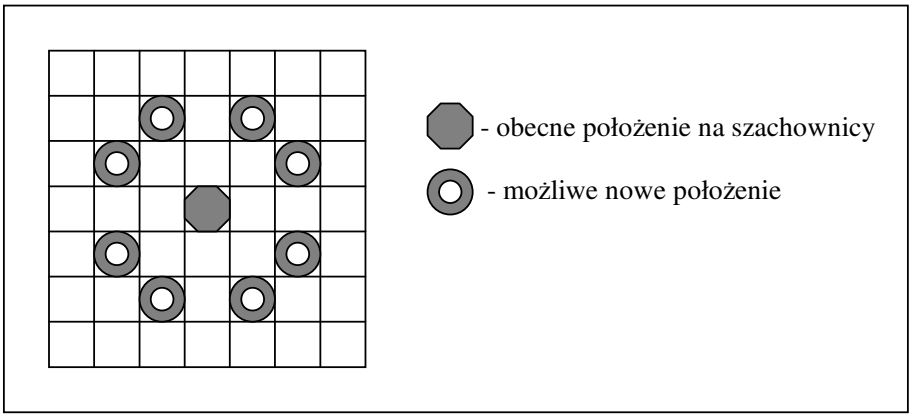
```
long silnia(long n)
{
    if( n == 1 )
        return 1;

    return ( n * silnia(n-1));
}


x = silnia( 3 );
```

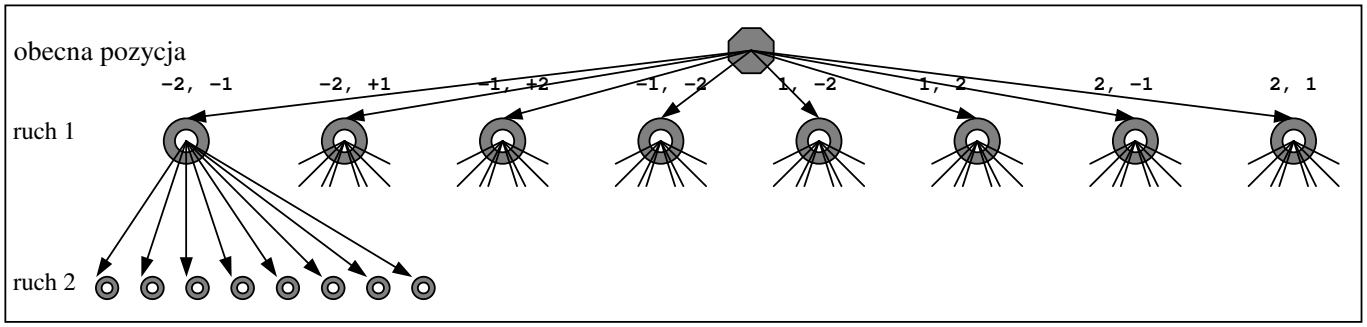



```
printf("\n%ld %ld %ld %ld %ld %ld\n", silnia(2), silnia(3), silnia(4), silnia(5), silnia(7));
```

Przykład rekurencyjnej funkcji do sprawdzania możliwych pozycji konika szachowego po *n* ruchach:

Konik szachowy:





```
int szachownica[22][22];  // 0 – pole wolne, 1 – pole zajęte

void czy_wolne_pole(int x, int y, int ruch)
{

  // jeżeli pole poza szachownica – powroc!

  if( x < 0 || x > 21 || y < 0 || y > 21)
        return;

  // jeżeli pole zajete – nie da się dalej wykonywac ruchu – powroc!

  if(szachownica[x][y] == 1)
        return;

  // jezeli osiagnięty zostal oczekiwany ruch – wyswietl mozliwosci i zakoncz galaz odwolan

  if( ruch == 0 )
  {
        printf("\n\t osiagnieto pole (%d, %d)", x, y);
        return;     // koniec odwolan!
  }

  printf("\n skok z pola (%d, %d), ruch %d", x, y, ruch);

  //  sprawdz mozliwosci skoku z biezacego miejsca

  czy_wolne_pole( x-2, y-1, ruch – 1);
  czy_wolne_pole( x-1, y-2, ruch – 1);
  czy_wolne_pole( x+2, y-1, ruch – 1);
  czy_wolne_pole( x+1, y-2, ruch – 1);
  czy_wolne_pole( x-2, y+1, ruch – 1);
  czy_wolne_pole( x-1, y+2, ruch – 1);
  czy_wolne_pole( x+2, y+1, ruch – 1);
  czy_wolne_pole( x+1, y+2, ruch – 1);

}
```

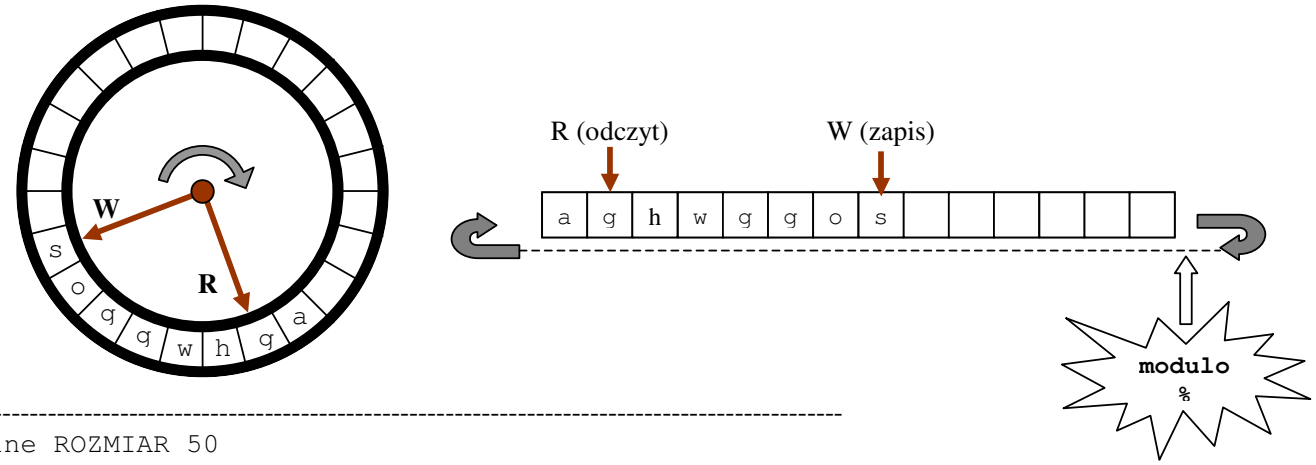
Przykład wywołania funkcji:

```
czy_wolne_pole(10,10,2);
```

```
skok z pola (10, 10), ruch 2
skok z pola (8, 9), ruch 1
        osiagnieto pole (6, 8)
        osiagnieto pole (7, 7)
        osiagnieto pole (10, 8)
        osiagnieto pole (9, 7)
        osiagnieto pole (6, 10)
        osiagnieto pole (7, 11)
        osiagnieto pole (10, 10)
        osiagnieto pole (9, 11)
skok z pola (9, 8), ruch 1
        osiagnieto pole (7, 7)
        osiagnieto pole (8, 6)
```

2. Bufor okrężny

Postać logiczna **Postać rzeczywista**



```
#define ROZMIAR 50

char bufor[ROZMIAR];
int poz_r=0, poz_w=0;

void wpisz_do_bufora(int w)
{
  poz_w++;
  poz_w = poz_w % ROZMIAR;
  bufor[poz_w] = w;
}

int odczytaj_z_bufora(void)
{
  if( poz_w == poz_r )   // nie ma nowych danych
        return -256;      // -256 – umowiony kod: nie ma nowych danych (poza zakresem char)

  poz_r++;
  poz_r = poz_r % ROZMIAR;

  return bufor[poz_r];
}
```

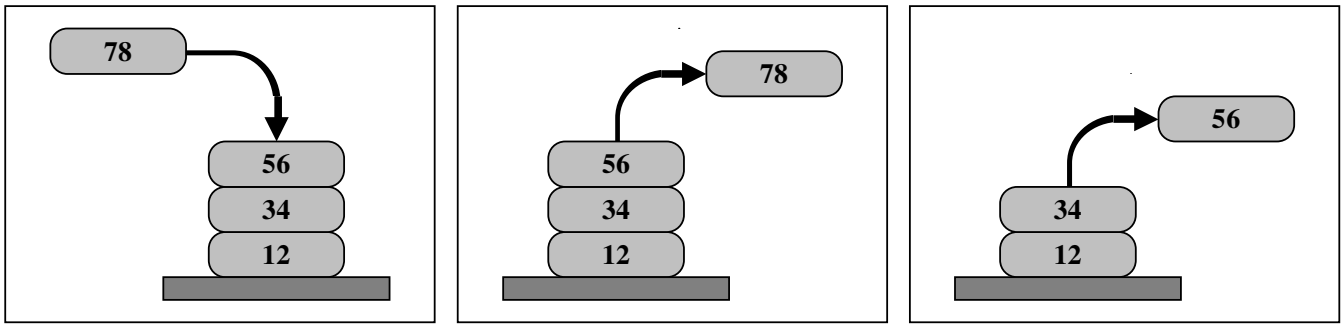
Przykład programu, który odczytuje wciśnięte klawisze i zapisuje je do bufora; przy wciśnięciu 'Enter' opróżnia bufor (na ekran). Wyjście – klawisz 'Esc'.

```
int kl=0, od;

while( kl != 27 )              // 27 - kod ASCII klawisza Esc
{
        kl = getch();

        if( kl != 13 )         // 13 - kod ASCII klawisza Enter
              wpisz_do_bufora(kl);
        else
              while( ( od = odczytaj_z_bufora() ) != -256 )
                 printf("%c", od);
}
```

3. Stos

Porządek: ostatni przyszedł – pierwszy obsłużony (ang. LIFO - Last In, First Out)

Zdefiniowane są dwie podstawowe operacje:
- odłożenie elementu na stos,
- ściągnięcie elementu ze stosu.



odłożenie na stos **ściągnięcie ze stosu** **ściągnięcie ze stosu**

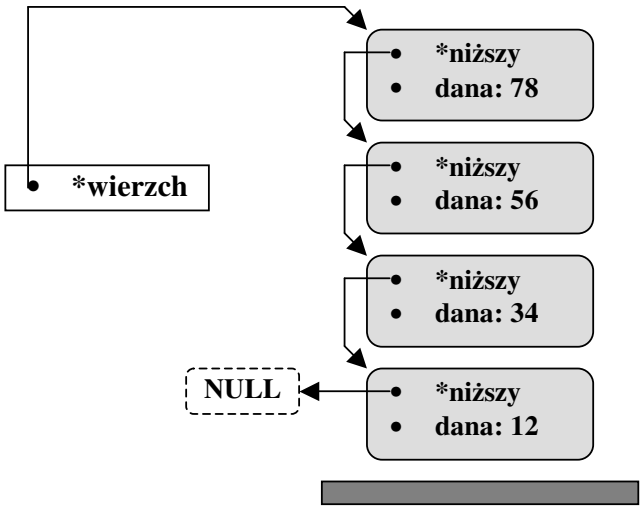
Implementacja stosu znaków char przy użyciu tablicy

```
#define ROZMIAR_STOSU 50

char stos[ROZMIAR_STOSU];
int stan_stosu = 0;

void odloz_na_stos(char element)
{
    stos[stan_stosu] = element;         // tablice numeruje sie od zera !
    stan_stosu ++;
}

int sciagnij_ze_stosu(void)
{
    int element;

    if(stan_stosu == 0)
        return -256;    //  umowiony kod oznaczajacy pusty stos (-256 poza zakresem char)

    element = stos[stan_stosu - 1];     // tablice numeruje sie od zera !
    stan_stosu --;

    return element;
}
```

Przykład programu, który odczytuje wciśnięte klawisze i odkłada ich znaki na stos; przy wciśnięciu 'Enter' opróżnia stos

```
int  kl = 0, od;
while(kl!= 27)          //  27 - kod ASCII klawisza Esc
{
    kl=getch();
    if(kl != 13)        //  13 - kod ASCII klawisza Enter
        odloz_na_stos(kl);
    else
        while( (od=sciagnij_ze_stosu()) != -256)
            printf("%c", od);
}
```

Implementacja stosu obiektów struktur – metoda z dynamicznym przydziałem pamięci




```
// definicja elementu stosu
struct element{
    char dana;

    struct element *nizszy;
};

// wskaznik wierzcholka stosu
struct element *wierzch_stosu = NULL;
```

Wyznaczanie liczby elementów stosu

```
int liczba_elementow_stos_dyn(void)
{
    int li=0;
    struct element * wskazywany = wierzch_stosu;

    while(wskazywany != NULL)
    {
        wskazywany = wskazywany->nizszy;
        li++;
    }

    return li;
}
```

Funkcje obsługi stosu, działające na wskaźnikach do obiektów struktur

```
void odloz_na_stos_dyn_strukture(struct element *na_stos)
{
    na_stos->nizszy = wierzch_stosu;
    wierzch_stosu = na_stos;
}


struct element* sciagnij_ze_stosu_dyn_strukture(void)
{
    struct element * sciagany;

    if( wierzch_stosu == NULL )
        return NULL;        // zwracany NULL, jesli nie bylo elementow na stosie

    sciagany = wierzch_stosu;    // zapamietanie wskaznika elementu z wierzchu stosu
    wierzch_stosu = wierzch_stosu->nizszy;  // przesuniecie wskaznika wierzchu stosu

    return sciagany;
}
```

Funkcje obsługi stosu, automatycznie przydzielające i zwalniające pamięć dla elementów stosu

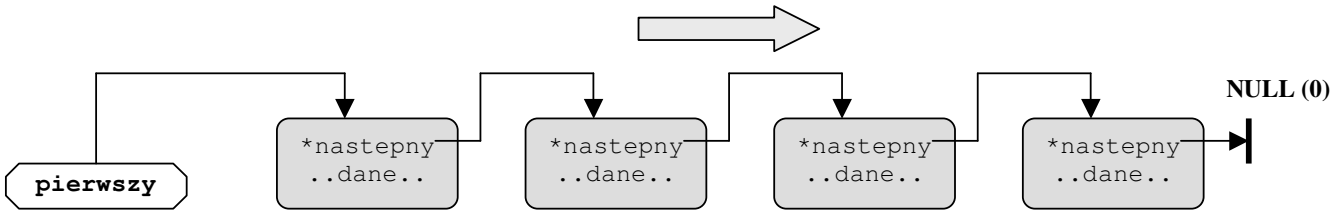
```
void odloz_na_stos_dyn(char wstawiana_dana)
{
    struct element *nowy;

    // dynamiczne tworzenie obiektu elementu
    nowy = (struct element *)malloc( sizeof(struct element) );
    nowy->dana =  wstawiana_dana;

    // odkladanie nowego obiektu na stos
    nowy->nizszy = wierzch_stosu;
    wierzch_stosu = nowy;
}


int sciagnij_ze_stosu_dyn(void)
{
    int sciagana_dana;
    struct element * sciagany;

    if(wierzch_stosu == NULL)
        return -256;     // umowiony kod oznaczajacy pusty stos (-256 poza zakresem char)

    // przepisanie wartosci wierzchniego elementu
    sciagana_dana = wierzch_stosu->dana;

    // usuwanie ze stosu wierzchniego elementu
    sciagany = wierzch_stosu;    // zapamietanie wskaznika elementu z wierzchu stosu
    wierzch_stosu = wierzch_stosu->nizszy;   // przesuniecie wskaznika wierzchu stosu

    // zwolnienie pamieci po bylym wierzchnim elemencie
    free(sciagany);

    return sciagana_dana;
}
```

Przykład programu, który odczytuje wciśnięte klawisze i odkłada ich znaki na stos; przy wciśnięciu 'Enter' opróżnia stos

```
int  kl = 0, od;

while(kl!= 27)           // 27 - kod ASCII klawisza Esc
{
        kl=getch();
        if(kl != 13)     // 13 - kod ASCII klawisza Enter
            odloz_na_stos_dyn(kl);
        else
            while( (od=sciagnij_ze_stosu_dyn()) != -256)
                printf("%c", od);
}
```

4. Lista jednokierunkowa

Porządek: pierwszy przyszedł – pierwszy obsłużony (ang. **FIFO** - First In, First Out)



Struktura elementu listy

```
struct element{

        struct element *nastepny;

        // dane przechowywane w elemencie listy

        char dane[100];
};

struct element *pierwszy = NULL;
```

Operacje dokonywane na liście

Wyznaczanie liczby elementów

```
int liczba_elementow()
{
    int n=0;
    struct element *wskazujacy;
    wskazujacy = pierwszy;

    while(wskazujacy != NULL)
    {
        n++;
        wskazujacy = wskazujacy->nastepny;
    }

    return n;
}
```

Wyświetlanie zawartości listy

```
void wyswietl_liste()
{
  struct element *wskazujacy;

  wskazujacy = pierwszy;

  while(wskazujacy != NULL)
  {
        printf("\nDane: %s", wskazujacy.dane );
        wskazujacy = wskazujacy->nastepny;
  }
}
```

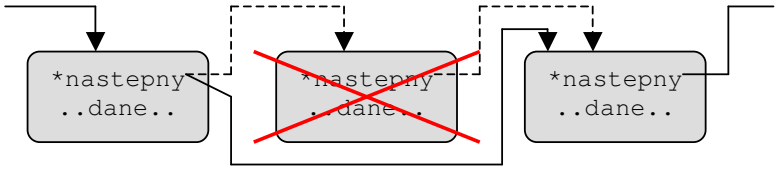
Pobieranie wskaźnika do n-tego elementu w liście

```
struct element* podaj_n_element(int n)   // numeracja jak w tablicy - od 0 do N-1
{
    int i=0;
    struct element *wskazujacy;

    wskazujacy = pierwszy;

    while(wskazujacy != NULL)
    {
        if( n == i )
            return wskazujacy;

        wskazujacy = wskazujacy->nastepny;
        i++;
    }

    return NULL;      // nie bylo n-tego elementu (albo zadnego)
}
```

Wstawianie nowego elementu na koniec listy (z alokacją pamięci)

```
void wstaw_dane_na_koniec_listy(char *nowedane)
{
    struct element *nowy;
    struct element *wskazujacy;
    wskazujacy = pierwszy;

    // stworzenie nowego obiektu
    nowy = (struct element *)malloc(sizeof(struct element));

    nowy->nastepny = NULL;           // ma byc ostatni – czyli wskazywac nastepnego = NULL
    strcpy(nowy->dane, nowedane);    // kopiowanie danych – można uzyc strncpy – bezpieczniej!

    // jeśli lista pusta – wpisanie pierwszego elementu jest trywialne
    if( pierwszy == NULL )
    {
        pierwszy = nowy;
        return;
    }

    // szukamy ostatniego elementu w liscie
    while(wskazujacy != NULL)
    {
        if( wskazujacy->nastepny == NULL )  // ostatni element wskazuje nastepny jako NULL
        {
            wskazujacy->nastepny = nowy;    // teraz wskazuje na nowy element, a nowy wskazuje na NULL
            break;
        }
        wskazujacy = wskazujacy->nastepny;
    }
}
```

Usuwanie n-tego elementu z listy

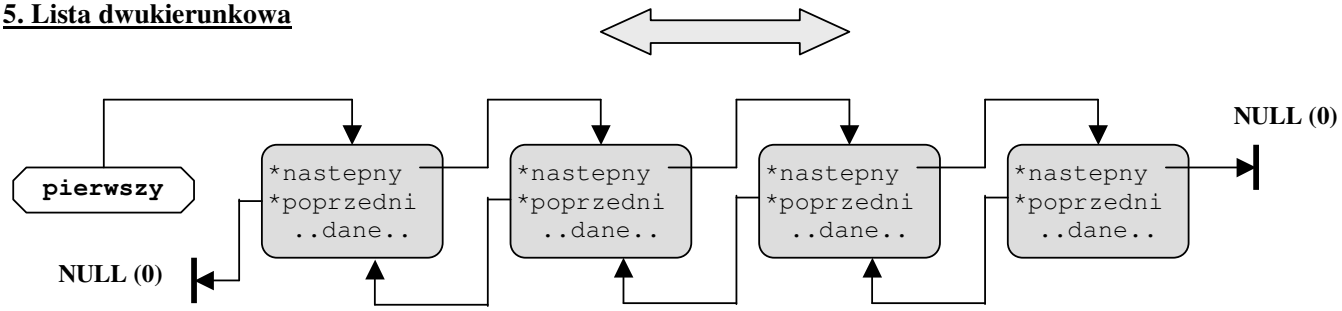



```
void usun_n_element(int n)   // numeracja jak w tablicy - od 0 do N-1
{
    int i=0;
    struct element *wskazujacy, *kasowany, *poprzedni;

    if( n == 0 )   // przypadek, gdy usuwany element nie ma poprzednika
    {
       kasowany = pierwszy;
       pierwszy = pierwszy->nastepny;
       free( kasowany );
       return;
    }

    poprzedni = pierwszy;
    wskazujacy = pierwszy;

    while(wskazujacy != NULL)
    {
        if( n == i )
        {
            // „latanie dziury" – poprzednik ma wskazywac na nastepnika biezacego elementu
            poprzedni->nastepny = wskazujacy->nastepny;

            free( wskazujacy );      // zwolnienie pamieci po usuwanym elemencie
            return;
        }

        // zapamietanie obecnego elementu, ktory w nastepnej petli bedzie poprzednim
        poprzedni = wskazujacy;

        wskazujacy = wskazujacy->nastepny;
        i++;
    }
}
```

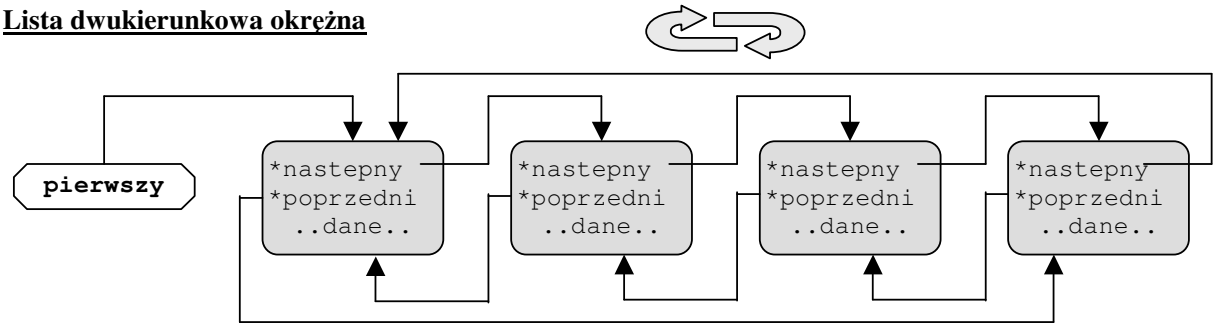
Usunięcie wszystkich elementów z listy

```
void usun_liste()
{
    struct element *wskazujacy, *kasowany;

    wskazujacy = pierwszy;
    pierwszy = NULL;

    while(wskazujacy != NULL)
    {
        kasowany = wskazujacy;
        wskazujacy = wskazujacy->nastepny;
        free(kasowany);        // pamiec nalezy zwolnic!!!
    }
}
```

5. Lista dwukierunkowa



Lista dwukierunkowa okrężna



Struktura elementu listy dwukierunkowej

```
struct element{

        struct element *nastepny;
        struct element *poprzedni;

        // dane przechowywane w elemencie listy

        char dane[100];
};

struct element *pierwszy = NULL;
```

Listy dwukierunkowe:

+ eleganckie implementacje np.:
  - usuwania elementów
  - zamiany kolejności elementów

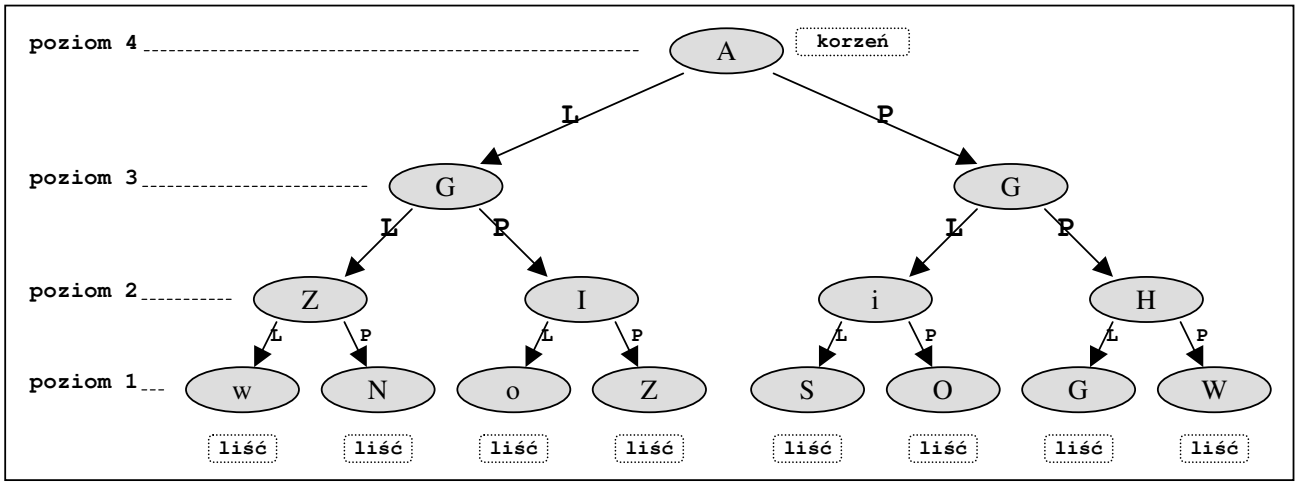
+ szybsze operacje

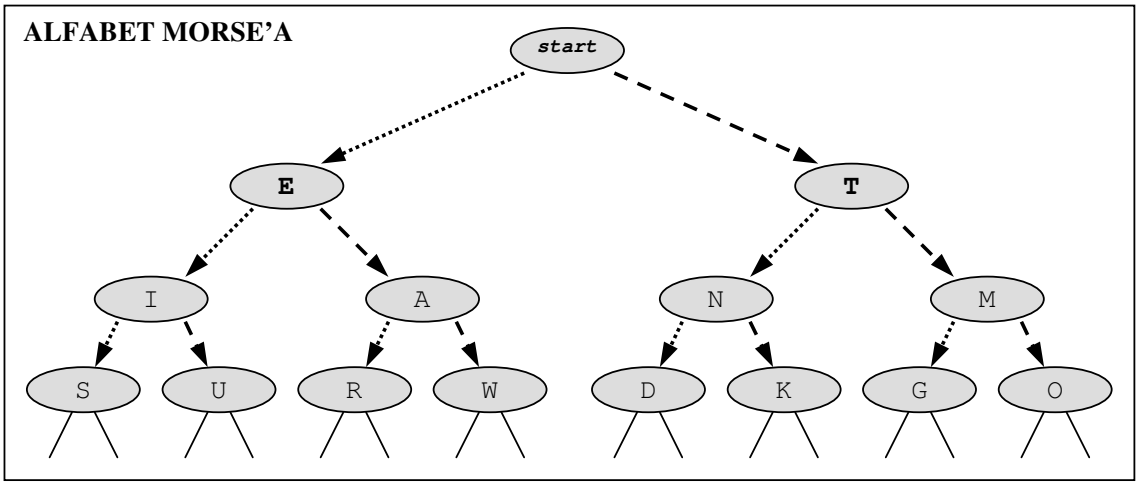
- utrudniona implementacja
  (więcej wskaźników do sprawdzenia)

Usunięcie elementu z listy

```
void usun_n_element(int n)
{
    // ....
    while(wskazujacy != NULL)
    {
        if( n == i )
        {
             // „latanie dziury" – poprzednik ma wskazywac na nastepnika biezacego elementu
            wskazujacy->poprzedni->nastepny = wskazujacy->nastepny;
            wskazujacy->nastepny->poprzedni = wskazujacy->poprzedni;

            free( wskazujacy );      // zwolnienie pamieci po usuwanym elemencie
            return;
        }

        wskazujacy = wskazujacy->nastepny;
        i++;
    }
}
```

6. Drzewa binarne





Litery – max 4 znaki
Wyszukanie litery w liście symboli – max 24 porównania
Wyszukanie litery w drzewie symboli – max 4 przejścia między węzłami (+przejście szybsze niż porównanie ciągu znaków)
Ogólnie: permutacja znaków {0,1}

liczba znaków w ciągu	max liczba porównań w liście*	liczba wyszukań w drzewie
2	4	2
3	8	3
4	16	4
5	32	5
n	***2^n***	***n***

* lista zawiera tylko ciągi o danej długości

Struktura węzła drzewa

```
struct wezel
{
   struct wezel *nadrzedny;

   struct wezel *prawy;
   struct wezel *lewy;

   int dana;
};
```

```
struct wezel
{
   struct wezel *nadrzedny;

   struct wezel *w_0;
   struct wezel *w_1;

   int dana;
};
```

```
struct wezel
{
   struct wezel *nadrzedny;

   struct wezel *w_kropka;
   struct wezel *w_kreska;

   int dana;
};
```

Implementacja wybranych funkcji drzewa

Tworzenie drzewa o *n* poziomach

```
struct wezel* tworz_drzewo(struct wezel *nadrz, int poziom)
{
       struct wezel *ten;
       ten = (struct wezel*) malloc(sizeof(struct wezel));

       ten->nadrzedny = nadrz;
```

```
int i;
printf("\n");
for(i=0; i < 10-poziom; i++)
     printf(" ");
printf("tworze wezel, poziom %d", poziom);
```

```
       if(poziom>1)
       {
            ten->prawy = tworz_drzewo(ten, poziom -1);
            ten->lewy  = tworz_drzewo(ten, poziom -1);
       }
       else
          ten->prawy = ten->lewy = NULL;

       return ten;
}
```

Przy pomocy funkcji `tworz_drzewo` można dołączać nowe poddrzewa do istniejących już liści.

Usuwanie drzewa z pamięci

```
void zniszcz_drzewo(struct wezel *gorny)
{
    if(gorny == NULL)
        return;

    if(gorny->prawy != NULL)
        zniszcz_drzewo(gorny->prawy);

    if(gorny->lewy != NULL)
        zniszcz_drzewo(gorny->lewy);

    free(gorny);
}
```

Wyszukiwanie węzła na podstawie ciągu danych

```
struct wezel* znajdz_wezel(char *ciag_sterujacy, struct wezel *poddrzewo)
{
     struct wezel *wez = poddrzewo;
     int i;
     for(i=0; i<strlen(ciag_sterujacy); i++)
        if( wez != NULL )
            if(ciag_sterujacy[i] == 'L' || ciag_sterujacy[i] == '.')
                wez = wez->lewy;
            else if(ciag_sterujacy[i] == 'P' || ciag_sterujacy[i] == '-')
                wez = wez->prawy;
    return wez;
}
```

Podawanie danych węzła wskazanego ciągiem sterującym

```
int podaj_dana_wezel(char *ciag_sterujacy, struct wezel *poddrzewo)
{
     struct wezel *wez = poddrzewo;
     int i;
     for(i=0; i<strlen(ciag_sterujacy); i++)
        if( wez != NULL )
            if(ciag_sterujacy[i] == 'L' || ciag_sterujacy[i] == '.')
                wez = wez->lewy;
            else if(ciag_sterujacy[i] == 'P' || ciag_sterujacy[i] == '-')
                wez = wez->prawy;

    if(wez == NULL)
        return 0;

    return wez->dana;
}
```

Wstawianie danych do węzła wskazanego ciągiem sterującym

```
int wstaw_dana_wezel(char *ciag_sterujacy, struct wezel *poddrzewo, int d)
{
     struct wezel *wez = poddrzewo;
     int i;
     for(i=0; i<strlen(ciag_sterujacy); i++)
        if( wez != NULL )
            if(ciag_sterujacy[i] == 'L' || ciag_sterujacy[i] == '.')
                wez = wez->lewy;
            else if(ciag_sterujacy[i] == 'P' || ciag_sterujacy[i] == '-')
                wez = wez->prawy;

    if(wez == NULL)
        return 0;    // nie udalo sie wstawic danej (nie znaleziono wezla o zadanym ciagu)

    wez->dana = d;

    return 1;        // znaleziono szukany wezel i wstawiono dana
}
```

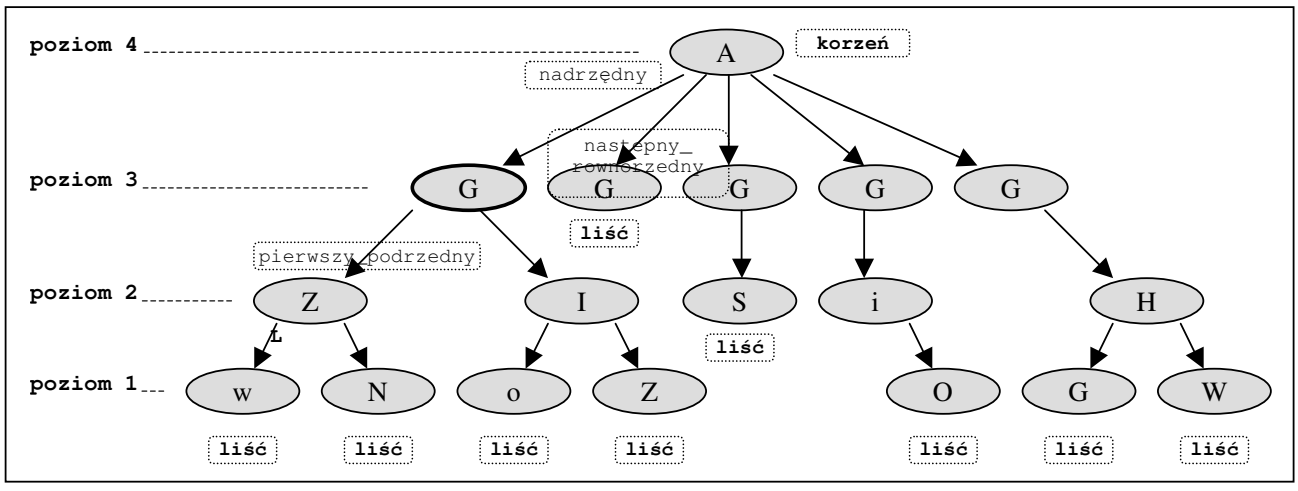
Przykładowy program do dekodowania liter alfabetu Morse'a

```
int main(int argc, char *argv[])
{
    struct wezel *korzen = NULL;

    korzen = tworz_drzewo(NULL, 3);

    wstaw_dana_wezel("",   korzen,'\0');
    wstaw_dana_wezel(".",  korzen, 'E');
    wstaw_dana_wezel("-",  korzen, 'T');
    wstaw_dana_wezel("..", korzen, 'I');
    wstaw_dana_wezel(".-", korzen, 'A');
    wstaw_dana_wezel("-.", korzen, 'N');
    wstaw_dana_wezel("--", korzen, 'M');

    // mozna uzywac zamiiennie {.,-} i {L,P}
    // wstaw_dana_wezel("LP", korzen, 'A');
```

```
    int i;
    char txt[100];

    printf("\npodaj ciag znakow:\n");
    for(i=0;i<5;i++)
    {
        scanf("%s", txt);
        printf("%s = %c\n", txt, podaj_dana_wezel(txt, korzen));
    }

    zniszcz_drzewo(korzen);
    return 0;
}
```

```
tworze wezel, poziom 4
  tworze wezel, poziom 3
    tworze wezel, poziom 2
      tworze wezel, poziom 1
      tworze wezel, poziom 1
    tworze wezel, poziom 2
      tworze wezel, poziom 1
      tworze wezel, poziom 1
  tworze wezel, poziom 3
    tworze wezel, poziom 2
      tworze wezel, poziom 1
      tworze wezel, poziom 1
    tworze wezel, poziom 2
      tworze wezel, poziom 1
      tworze wezel, poziom 1
```

7. Drzewa o dowolnej liczbie podwęzłów




```
struct wezel
{
   struct wezel *nadrzedny;

   struct wezel *nastepny_rownorzedny; //~LISTA!
   struct wezel *pierwszy_podrzedny;   //~LISTA!

   int dana;
};
```