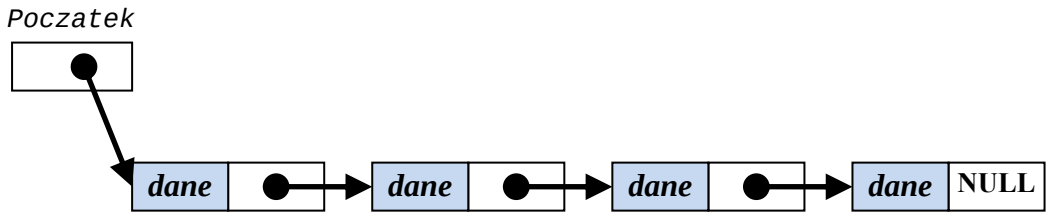
Algorytmy i złożoności

Wykład 3. Listy jednokierunkowe

Wstęp. Lista jednokierunkowa jest strukturą pozwalającą na pamiętanie danych w postaci uporządkowanej, a także na bardzo szybkie wstawianie i usuwanie elementów do i z listy. Pamiętana jest w postaci „kontenerków" zawierających porcję danych oraz wskaźnik (adres) następnego „kontenerka". W ten sposób wystarczy pamięta wskaźnik do pierwszego elementu listy, by pamiętać całą listę.

Aby zaimplementować listę jednokierunkową w języku C/C++ trzeba zdefiniować strukturę pełniącą rolę węzłów stanowiących kolejne elementy listy. Taka struktura składa się z dwóch części: pola (lub kilku pól), w którym pamiętane są przechowywane elementy (dane użytkowe) oraz pola wskaźnikowego, w którym będzie pamiętany wskazanie do następnego węzła. Dostęp do całej listy wymaga utworzenia zmiennej wskaźnikowej, zawierającej wskazanie na pierwszy węzeł listy lub wartość NULL jeśli lista list pusta (tzn. nie zawiera żadnych węzłów).



Rys. 1 Lista jednokierunkowa zawierająca 4 elementy.

Poniżej przedstawiono przykładową definicję typów danych umożliwiających tworzenie listy jednokierunkowej, w której będą pamiętane dane typu `Towar`. Kolejne węzły listy będą zmiennymi dynamicznymi typu `Wezel`, Zmienna wskaźnikowa `Poczatek` może być wykorzystywana do pamiętania wskazania na pierwszy element listy.

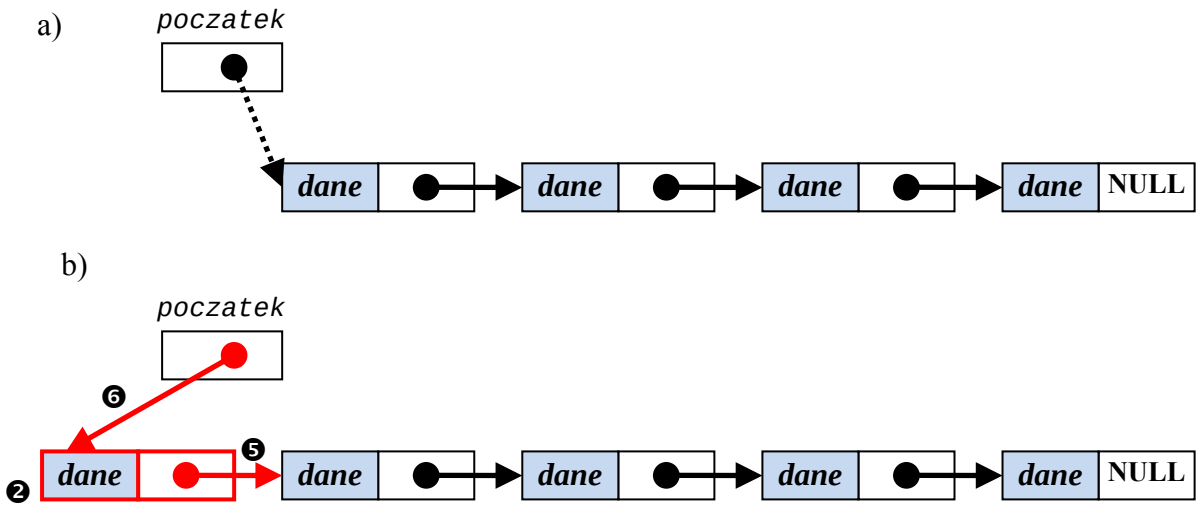
```
typedef struct Towar
          { char Nazwa[20];
            float Cena;
            int Ilosc;
          } Element;  // definicja typu danych
                      // przechowywanych w liście


typedef struct Wezel
          { Element dane;
            Wezel *nastepny;
          } Wezel;


Wezel *Poczatek = NULL:
```

Poniżej opisano typowe operacje wykonywane na elementach listy jednokierunkowej.

Dodawanie nowego elementu na początek listy. Podstawową operacją wykonywaną na liście jest dodawanie nowych elementów do listy. Trzeba wyróżnić dwa szczególne przypadki: dodawanie nowego elementu na początek listy oraz wstawianie nowego elementu za wskazanym elementem. Rysunek 2 ilustruje przypadek pierwszy:



Rys. 2 Ilustracja dodawania nowego elementu na początek listy a) stan listy przed rozpoczęciem operacji, b) stan listy po zakończeniu operacji.

Kolorem czerwonym zaznaczono nowy węzeł listy oraz nowe dowiązania, a numery na rysunku odpowiadają instrukcjom programu przedstawionego poniżej.

```
int DodajNaPoczatek(Wezel *&poczatek, Element dane)      ❶
{
  Wezel *tmp;                                            ❶
  tmp = new Wezel;    //(Wezel*)malloc(sizeof(Wezel));   ❷
  if (tmp==NULL)                                         ❸
      { printf("BRAK PAMIECI !");
        return 1;
      }
  tmp->dane = dane;                                      ❹
  tmp->nastepny = poczatek;                              ❺
  poczatek = tmp;                                        ❻
  return 0;
}
```

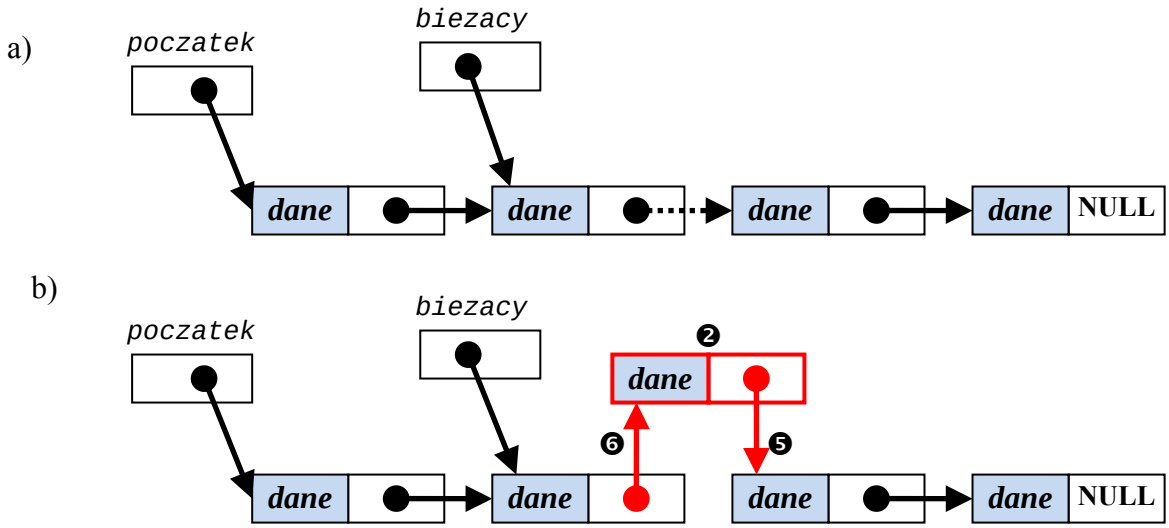
Nagłówek funkcji (linia ❶) zawiera dwa parametry: `poczatek` oraz `dane`. Pierwszy parametr jest **przekazanym przez referencję** wskaźnikiem na początek listy. Drugi parametr zawiera dane, które mają być zapamiętane w nowym elemencie listy. W linii ❶ zadeklarowano pomocniczą zmienną wskaźnikową `tmp`, w której będzie pamiętany adres utworzonej zmiennej dynamicznej (instrukcja ❷) będącej nowym węzłem listy. Zmienną dynamiczną można utworzyć za pomocą operatora `new` (tylko w języku C++) lub za pomocą funkcji `malloc` (zob. komentarz). W linii ❸ sprawdzana jest poprawność alokacji pamięci. W instrukcji ❹ do pola `dane` w nowym węźle są kopiowane dane, które mają być przechowywane w liście. Instrukcje ❺ i ❻ dokonują modyfikacji dowiązań. Powyższa funkcja zwraca wartość 0 gdy operacja dodania elementu przebiega poprawnie oraz wartość 1

gdy element nie zostanie dodany z powodu braku pamięci na utworzenie zmiennej dynamicznej będącej nowym węzłem listy.

Parametr `poczatek` musi być przekazany przez referencję, gdyż w wyniku poprawnego wykonania tej funkcji adres początkowego elementu listy ulegnie zmianie.

Wstawianie nowego elementu za wskazanym elementem

Czasami zachodzi konieczność wstawienia nowego elementu w środku lub na końcu listy. Wówczas miejsce wstawienia określa adres węzła, za którym ma być wstawiony nowy węzeł, który musi być pamiętany w jakiejś zmiennej wskaźnikowej. W poniższym przykładzie adres jest pamiętany w zmiennej `biezacy`. Ilustruje to rysunek 3.



Rys. 3 Ilustracja wstawiania nowego elementu za węzłem wskazywanym przez zmienną `biezacy` a) stan listy przed rozpoczęciem operacji, b) stan listy po zakończeniu operacji.

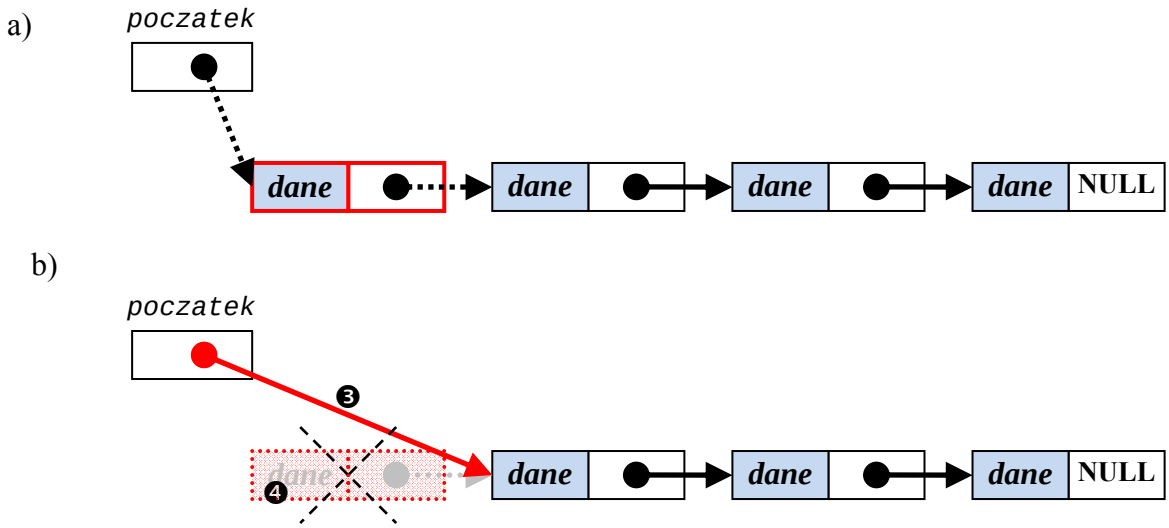
Kolorem czerwonym zaznaczono nowy węzeł listy oraz nowe dowiązania, a numery na rysunku odpowiadają instrukcjom programu przedstawionego poniżej.

```
int DodajZaBiazacy(Wezel *biezacy, Element dane)          ⓿
{ Wezel *tmp;                                             ❶
  tmp = new Wezel;  // (Wezel*)malloc(sizeof(Wezel));     ❷
  if (tmp==NULL)                                          ❸
      { printf("BRAK PAMIECI !");
        return 1;
      }
  tmp->dane = dane;    // tu powinno być wpisanie danych  ❹
  tmp->nastepny = biezacy->nastepny;                      ❺
  biezacy->nastepny = tmp;                                ❻
  return 0;
}
```

Nagłówek funkcji (linia ⓿) zawiera dwa parametry: `biezacy` oraz `dane`. Pierwszy parametr jest wskaźnikiem na element listy, za którym ma być dodany nowy element. Drugi parametr zawiera dane, które mają być zapamiętane w nowym elemencie listy. Powyższa funkcja zwraca wartość 0 gdy operacja dodania elementu przebiega poprawnie oraz wartość 1

gdy element nie zostanie dodany z powodu braku pamięci na utworzenie zmiennej dynamicznej.

Usuwanie początkowego elementu listy. Drugą ważną operacją wykonywaną na liście jest usuwanie elementu. Podobnie jak przy dodawaniu, trzeba wyróżnić dwa przypadki: usuwanie pierwszego elementu listy oraz usuwanie środkowego (ostatniego) elementu. Przypadek pierwszy ilustruje rysunek 4.



Rys. 4 Ilustracja usuwania pierwszego elementu listy a) stan listy przed rozpoczęciem operacji, b) stan listy po zakończeniu operacji.

Kolorem czerwonym zaznaczono usuwany węzeł oraz nowe dowiązanie, a numery na rysunku odpowiadają instrukcjom programu przedstawionego poniżej.

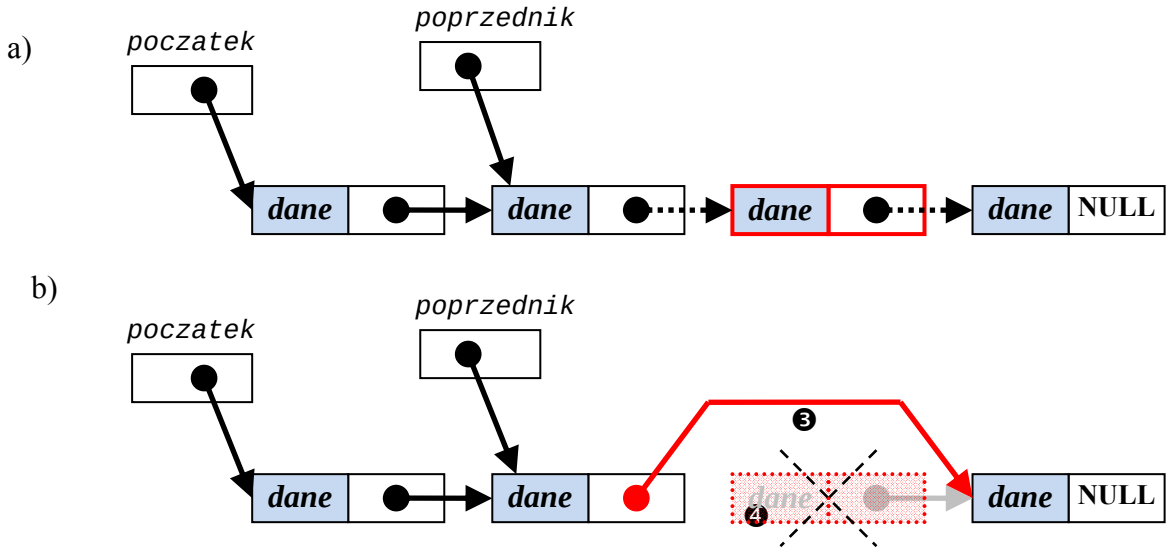
```
void UsunPoczatek(Wezel *&poczatek)                    ❶
{ Wezel *tmp;                                          ❶
  tmp = poczatek                                       ❷
  poczatek = tmp->nastepny;                            ❸
  delete tmp; // free(tmp);                            ❹
}
```

Nagłówek funkcji (linia ❶) zawiera jeden parametr, który jest **przekazanym przez referencję** wskaźnikiem na początek listy. Do wykonania tej operacji konieczna jest pomocnicza zmienna wskaźnikowa `tmp` (deklaracja ❶), w której zostanie chwilowo zapamiętany adres usuwanego węzła (instrukcja ❷). W instrukcji ❸ modyfikowane jest dowiązanie pamiętane w zmiennej `poczatek`. Instrukcja ❹ usuwa zmienną dynamiczną, w której był pamiętany usuwany węzeł. Można to zrobić za pomocą operatora `delete` (tylko w języku C++) lub za pomocą funkcji `free` (zob. komentarz).

Częstym błędem, przy usuwaniu elementu listy jest pomijanie instrukcji usuwania zmiennej dynamicznej. Jest to trudny do wykrycia błąd, który powoduje tzw. wycieki pamięci.

Parametr `poczatek` musi być przekazany przez referencję, gdyż w wyniku wykonania tej funkcji adres początkowego elementu listy ulegnie zmianie.

Usuwanie elementu listy znajdującego się za wskazanym elementem. Czasami zachodzi konieczność usuwania elementu w środku lub na końcu listy. Wówczas należy wskazać węzeł bezpośrednio poprzedzający usuwany węzeł. W poniższym przykładzie adres ten jest pamiętany w zmiennej `poprzednik`. Ilustruje to rysunek 5.



Rys. 5 Ilustracja usuwania środkowego elementu listy a) stan listy przed rozpoczęciem operacji, b) stan listy po zakończeniu operacji.

Kolorem czerwonym zaznaczono usuwany węzeł oraz nowe dowiązanie, a numery na rysunku odpowiadają instrukcjom programu przedstawionego poniżej.

```
void UsunNastępny(Wezel *poprzednik)                    ⓿
{ Wezel *tmp;                                           ❶
  tmp = poprzednik->nastepny;                           ❷
  poprzednik->nastepny = tmp->nastepny;                 ❸
  delete tmp; // free(tmp);                             ❹
}
```

Usuwanie wszystkich elementów listy. W niektórych sytuacjach zachodzi konieczność usunięcia wszystkich elementów listy. Wówczas można tego dokonać za pomocą prostej pętli, tak jak przedstawiono to w poniższym przykładzie:

```
void UsunWszystkie(Wezel *&poczatek)                    ⓿
{ Wezel *tmp;                                           ❶
  while (poczatek != NULL)                              ❷
     { tmp = poczatek;                                  ❸
       poczatek = tmp->nastepny;                        ❹
       delete tmp;  // free(tmp);                       ❺
     }
}
```

Przy usuwaniu wszystkich elementów nie wolno zapomnieć o usuwaniu zmiennych dynamicznych (instrukcja ❺), w których pamiętane są węzły listy. Pominięcie tej instrukcji jest trudnym do wykrycia błędem powodującym wycieki pamięci

Parametr `poczatek` musi być przekazany przez referencję, gdyż w wyniku wykonania tej funkcji adres początkowego elementu listy ulegnie zmianie.

Przechodzenie przez wszystkie elementy listy. Jedną z najczęściej wykonywanych operacji na listach jednokierunkowych jest przechodzenie przez wszystkie elementy listy od pierwszego do ostatniego. W trakcie takiego przechodzenia można wykonać na każdym elemencie jakąś operację. Typowym przykładem jest drukowanie na ekranie wszystkich danych zapamiętanych w liście. Ilustruje to poniższy przykład:

```
void DrukujListe(Wezel *poczatek)
{ Wezel *tmp;
  for(tmp=poczatek; tmp!=NULL; tmp=tmp->nastepny)
     {
       // tu nalezy wykonac operacje na elemencie
       // wskazywanym przez zmienną  tmp. Np...
       printf("   Adres elementu: %p\n", tmp)
       printf(" Wartosc elementu: %20s %5.2f %d\n",
                tmp->dane.Nazwa, tmp->dane.Cena,
                tmp->dane.Ilosc);
     }
}
```

Wyszukiwanie elementu spełniającego jakiś warunek. Często zachodzi konieczność wyszukania elementu listy, który spełnia określony warunek. Taki element może być później modyfikowany lub nawet usuwany. Wiele operacji, które mają być wykonane na wskazanym elemencie listy, mogą być wykonane tylko wówczas gdy znany jest element poprzedni. Przykładem takiej operacji jest usuwanie elementu, przedstawione w poprzednich punktach. Aby wykonanie takich operacji było możliwe należy pamiętać nie tylko adres bieżącego elementu, ale również adres poprzednika. Taki sposób wyszukiwania zastosowano w poniższej funkcji, która wyszukuje element listy zawierający towar o podanej nazwie.

```
int WyszukajTowar(Wezel *poczatek, char *nazwa,
                  Wezel *&biezacy, *&poprzednik)
{ poprzednik = NULL;
  biezacy = poczatek;
  while(biezacy != NULL)
     { if ( strcmp(biezacy->dane.nazwa, nazwa)==0 )      ➲
          { return 0;
          }
       poprzednik = biezacy;
       biezacy = biezacy->nastepny;
     }
  return 1;
}
```

W linii oznaczonej symbolem ➲ należy podać warunek który ma spełniać poszukiwany element listy. W tej funkcji jest porównywana nazwa towaru pamiętanego w bieżącym węźle listy z nazwą podaną w parametrze funkcji.

Po wykonaniu powyższej funkcji w zmiennej `biezacy` będzie pamiętany adres elementu listy, który spełnia postawiony warunek. Jeśli zmienna `biezacy` ma wartość `NULL` to w liście nie ma poszukiwanego elementu. Jeśli znaleziony element jest pierwszym elementem listy, to zmienna `poprzednik` ma wartość `NULL`. W przeciwnym wypadku, zmienna ta zawiera adres elementu bezpośrednio poprzedzającego element `biezacy`.

Funkcja zwraca wartość 0 jeśli element zostanie znaleziony lub wartość 1, gdy żaden z Węzłów listy nie zawiera towaru o podanej nazwie.

Parametry `biezacy` i `poprzednik` muszą być przekazane do funkcji przez referencję.

Odwracanie kolejności elementów listy. Inną operacją wykonywaną na elementach listy może być odwracanie kolejności elementów listy. Taką operację wykonuje poniższe funkcja:

```
void OdwracanieListy(Wezel *&poczatek)
{ Wezel * nowyPoczatek = NULL;
  Wezel *tmp;
  while(poczatek != NULL)
     { tmp = poczatek->nastepny;
       poczatek->nastepny = nowyPoczatek;
       nowyPoczatek = poczatek;
       poczatek = tmp;
     }
  poczatek = nowyPoczatek;
}
```

Odwracanie porządku polega na przepisaniu wszystkich elementów pierwotnej listy do pomocniczej listy, której początek jest pamiętany w zmiennej wskaźnikowej `nowyPoczatek`. Po przepisaniu wszystkich elementów listy, parametrowi `poczatek` przypisywana jest wartość nowy`Poczatek`.

Parametr `poczatek` musi być przekazany do funkcji przez referencję.

Sortowanie elementów listy. Sortowanie elementów listy jest jedną z bardziej złożonych operacji. Poważnym utrudnieniem jest brak możliwości bezpośredniego odwoływania się do elementów listy. Dlatego proces sortowania polega na budowaniu nowej listy z elementów listy pierwotnej (nieposortowanej). Sortowanie można realizować metodą selekcji lub wstawiania. W pierwszym przypadku z listy pierwotnej jest pobierany element „największy” i jest on wstawiany na początek listy sortowanej. Taka procedura jest powtarzana do momentu wyczerpania wszystkich elementów listy pierwotnej. Na zakończenie zmiennej wskazującej początek listy pierwotnej jest przypisywana wartość zmiennej wskazującej na początek listy posortowanej. Taki proces sortowania realizuje poniższa funkcja, która porządkuje elementy listy według ceny towarów pamiętanych w węzłach listy.

```
void SortujSelekcja(Wezel *&poczatek)
{ Wezel *nowyPoczatek = NULL;
  Wezel *max, *popmax, *tmp;
  // max – wskaźnik na element o maksymalnej cenie towaru
  // popmax – wskaźnik na element poprzedzający max
  //         lub NULL, gdy max jest początkowym elementem

  while( poczatek )
     { max = poczatek;
       popmax = NULL;
       tmp = poczatek;
       while(tmp->nastepny)
           { if (tmp->nastepny->dane.Cena>max->dane.Cena)
                { popmax = tmp;
                  max = tmp->nastepny;
                }
             tmp = tmp->nastepny;
           }
       if (popmax) popmax->nastepny = max->nastepny;
              else poczatek = max->nastepny;
       max->nastepny = nowyPoczatek;
       nowyPoczatek = max;
     }
  poczatek = nowyPoczatek;
}
```