1. Listy powiązane (linked lists; *lista jednostronnie wiązana*)
2. Abstrakcyjne typy danych (Abstract Data Types (ADTs); *stos w oparciu o listę jednostronnie wiązaną, kolejki w oparciu o listę dwustronną*)
3. Listy wyspecjalizowane (specialized lists; *listy dwukierunkowe*)

1. Lista jednostronnie wiązana

Lista powiązana (jednostronnie wiązana) jest strukturą cechującą szybkie wstawianie i kasowanie danych. Na ćwiczeniach nr 3 poznaliśmy struktury danych typu tablica oraz tablica posortowana. Obie struktury mają pewne wady. W przypadku tablic nieposortowanych, wyszukiwanie jest wolne, podczas gdy w tablicach posortowanych wolno odbywa się wstawianie nowych elementów. Czasochłonnym jest też usuwanie wybranego elementu tablicy w obu typach tablic.

W liście powiązanej, każdy element danych znajduje się wewnątrz obiektu typu **Link** (ogniwo). Ponieważ istnieje zwykle wiele podobnych obiektów (**link**s) w liście (list), jest rzeczą sensowną aby stworzyć dla nich oddzielną klasę w odróżnieniu od samej listy powiązanej. Każdy obiekt typu Link zawiera (jako pole klasy) wskaźnik (**pNext**) do następnego obiektu typu Link w liście.

Przykład definicji klasy Link (zawiera dane oraz wskaźnik na następny obiekt klasy Link):

```
class Link
{
    public:
        int iData;      //data
        double dData;   //data
        Link* pNext;    //pointer to next link
};
```

Tego typu definicja klasy nazywana jest samoreferencją (self-referential) ponieważ zawiera pole klasy – pNext - będącym wskaźnikiem do obiektu takiego samego typu co zdefiniowana klasa. Klasa Link zawiera tylko dwa pola typu int i double. Zwykle w aplikacjach tego typu klasa będzie zawierać więcej niż dwa pola, np. w aplikacji przechowującej dane personelu takimi polami mogą być: imię, adres, numer ubezpieczenia, tytuł, pensja. Często zamiast pól z danymi znajduje się wskaźnik do obiektu przechowującego tego typu dane:

```
class Link
{
    public:
        inventoryItem* pItem;  //pointer to object holding data
        Link* pNext;           //pointer to next link
};
```

Struktura definiowana przez relację zamiast pozycji
W tablicach każdy element zajmuje konkretną pozycję. Do elementu na danej pozycji można uzyskać dostęp poprzez numer indeksu. Jest to podobne do numeracji domów stojących w rzędzie: możesz odnaleźć konkretny dom korzystając z adresu.

W przypadku listy jedynym sposobem aby odnaleźć konkretny element jest podążanie wzdłuż łańcucha ogniw (links). Podobnie jest z relacjami między ludźmi: możesz zapytać Harry'ego gdzie jest Bob. Harry nie wie lecz myśli, że Jane może wiedzieć, więc możesz zapytać Jane. Jane widziała jak Bob opuścił biuro z Sally, więc możesz zadzwonić na komórkę Sally. Sally zostawiła Boba w biurze Petera, więc … . W ten sposób aby uzyskać dostęp do danych musisz użyć powiązania pomiędzy elementami (zaczynasz od pierwszego elementu, przechodzisz do drugiego itd. Postępujesz tak długo aż odnajdziesz szukany element).

Zanim otrzymamy kompletny program linkList.cpp, przyjrzymy się pewnym ważnym składowym klasy Link i klasy LinkList.

Klasa **Link**

```
class Link
{
    public:
        int iData;                          //data item
        double dData;                       //data item
        Link* pNext;                        //ptr to next link in list

        Link(int id, double dd) :           //constructor
        iData(id), dData(dd), pNext(NULL)
        {  }

        void displayLink()                  //display ourself {22, 2.99}
        {
            cout << "{" << iData << ", " << dData << "} ";
        }
};  //end class Link
```

displayLink() wyświetla dane obiektu typu Link w formacie {22, 2.99}.
Konstruktor inicjalizuje dane. Nie ma potrzeby inicjalizowania pola pNext gdyż jest automatycznie ustawiany na **NULL** w momencie tworzenia (pomimo że może być ustawiony jawnie na NULL dla większej przejrzystości kodu). Wartość NULL oznacza referencję do żadnego konkretnego obiektu (tak jest w sytuacji zanim link (obiekt typu Link) zostanie powiązany z innymi links)).

Klasa **LinkList**

Klasa LinkList zawiera tylko jeden element danych: wskaźnik na pierwszy element listy (będący obiektem typu Link). Wskaźnikiem tym jest **pFirst**. Jest to jedyna informacja jaką lista przechowuje (a tym samym o lokacji innych obiektach typu Link). Znalezienie innych elementów listy jest możliwe dzięki łańcuchowi wskaźników (pola pNext), który ma początek w pFirst.

```
class LinkList
{
    private:
        Link* pFirst;                   //ptr to first link on list
        public:

        LinkList()   : pFirst(NULL)     //constructor
        {  }                            //(no links on list yet)
```

```
        bool isEmpty()                  //true if list is empty
        { return pFirst==NULL; }

        ...                             //other methods go here
};  //end class LinkList
```

Konstruktor klasy LinkList ustawia wskaźnik pFirst na NULL. Kiedy pFirst ma wartość NULL, wiemy, że lista nie zawiera elementów. Jeśli lista zawiera jakąkolwiek liczbę elementów, wówczas pFirst zawierać będzie wskaźnik na pierwszy element. Fakt ten zostanie wykorzystany przy tworzeniu funkcji składowej **isEmpty()**.

Funkcja składowa insertFirst()

Funkcja składowa **insertFirst()** klasy LinkList wstawia nowy obiekt link na początek listy. Wstawienie obiektu link na początku listy jest najłatwiejszym rozwiązaniem ponieważ pFirst już wskazuje na pierwszy obiekt link. Aby wstawić nowy obiekt (link), potrzebujemy:

1) ustawić zmienną pola pNext w nowo utworzonym obiekcie link tak aby wskazywała na obiekt link będący uprzednio jako pierwszy,
2) Zmienić pFirst tak aby wskazywała na nowo utworzony link.

Metodę insertFirst() napiszemy w taki sposób aby umożliwić przesłanie do niej argumentów (przez wartość) typu int i double. Dane te użyte zostaną to utworzenia nowego obiektu link (typu Link). Następnie zmienimy wskaźniki obiektu link w powyżej opisany sposób.

```
//insert at start of list
void insertFirst(int id, double dd)
{                                   //make new link
    Link* pNewLink = new Link(id, dd);
    pNewLink->pNext = pFirst;       //newLink-->old first
    pFirst = pNewLink;              //first-->newLink
}
```

Funkcja składowa removeFirst()

Funkcja składowa **removeFirst()** usuwa link z listy (odłącza pierwszy obiekt link przestawiając pFirst na następny link - operacja ta nie usuwa z pamięci komputera obiektu będącego pierwotnie jako pierwszy (`pFirst = pFirst->pNext //(2);` stąd nazwa remove zamiast delete)).

```
void removeFirst()                  //delete first link
{                                   //(assumes list not empty)
    Link* pTemp = pFirst;           //(1) save first
    pFirst = pFirst->pNext;         //(2) unlink it: first-->old next
    delete pTemp;                   //(3) delete old first
}
```

Aby zapobiec „wycieku pamięci” (memory leak) potrzebujemy jeszcze usunąć obiekt link z pamięci. W tym celu wcześniej tworzymy tymczasową zmienną wskaźnikową do obiektu link (1) a następnie po reorganizacji listy (2) usuwamy obiekt z pamięci (3)).

Funkcja składowa displayList()

```
void displayList()
{
    cout << "List (first-->last): ";
    Link* pCurrent = pFirst;    //start at beginning of list
    while(pCurrent != NULL)     //until end of list,
    {
        pCurrent->displayLink();    //print data
        pCurrent = pCurrent->pNext; //move to next link
    }
    cout << endl;
}
```

Przykład 1 (C++, lista jednostronnie wiązana) Kompletny program linkList.cpp.

[linkList.cpp]

```
//linkList.cpp
//demonstrates linked list
#include <iostream>
using namespace std;

class Link
{
    public:
        int iData;                  //data item
        double dData;               //data item
        Link* pNext;                //ptr to next link in list

        Link(int id, double dd) :   //constructor
        iData(id), dData(dd), pNext(NULL)
        {  }

        void displayLink()          //display ourself {22, 2.99}
        {
        cout << "{" << iData << ", " << dData << "} ";
        }

};  //end class Link
//--------------------------------------------------------------
class LinkList
{
    private:
        Link* pFirst;                       //ptr to first link on list

    public:
        LinkList()  : pFirst(NULL)          //constructor
        {  }                                //(no links on list yet)

        bool isEmpty()                      //true if list is empty
        { return pFirst==NULL; }

        //insert at start of list
        void insertFirst(int id, double dd)
        {                                   //make new link
            Link* pNewLink = new Link(id, dd);
            pNewLink->pNext = pFirst;       //newLink-->old first
            pFirst = pNewLink;              //first-->newLink
        }

        Link* getFirst()                    //return first link
        { return pFirst; }
```

```
        void removeFirst()              //delete first link
        {                               //(assumes list not empty)
            Link* pTemp = pFirst;       //save first
            pFirst = pFirst->pNext;     //unlink it: first-->old next
            delete pTemp;               //delete old first
        }

        void displayList()
        {
            cout << "List (first-->last): ";
            Link* pCurrent = pFirst;    //start at beginning of list
            while(pCurrent != NULL)     //until end of list,
            {
                pCurrent->displayLink();//print data
                pCurrent = pCurrent->pNext;  //move to next link
            }
            cout << endl;
        }

};  //end class LinkList
//-----------------------------------------------------------
int main()
{
    LinkList theList;                       //make new list

    theList.insertFirst(22, 2.99);          //insert four items
    theList.insertFirst(44, 4.99);
    theList.insertFirst(66, 6.99);
    theList.insertFirst(88, 8.99);

    theList.displayList();                  //display list

    while( !theList.isEmpty() )             //until it's empty,
    {
        Link* pTemp = theList.getFirst();   //get first link
                                            //display its key
        cout << "Removing link with key " << pTemp->iData << endl;
        theList.removeFirst();              //remove it
    }
    theList.displayList();                  //display empty list
    return 0;
}  //end main()
```

Znajdowanie i usuwanie określonych obiektów (Links)

Kolejnym przykładowy program będzie zawierać dwie dodatkowe metody. Metody te będzie można zastosować do:

- wyszukania elementu listy zawierającego zmienną o określonej wartości,
- usuwania elementu listy zawierającego zmienną o określonej wartości.

Przykład 2 (C++, lista jednostronnie wiązana) Kompletny program linkList2.cpp.

[linkList2.cpp]

```
//linkList2.cpp
//demonstrates linked list
#include <iostream>
using namespace std;
```

```
class Link
{
    public:
        int iData;                  //data item (key)
        double dData;               //data item
        Link* pNext;                //next link in list

        Link(int id, double dd) :   //constructor
        iData(id), dData(dd), pNext(NULL)
        {  }

        void displayLink()          //display ourself: {22, 2.99}
        {
            cout << "{" << iData << ", " << dData << "} ";
        }
};  //end class Link
//--------------------------------------------------------------
class LinkList
{
    private:
        Link* pFirst;               //ptr to first link on list

    public:
        LinkList()  : pFirst(NULL)  //constructor
        {  }                        //(no links on list yet)

        ~LinkList()                 //destructor (deletes links)
        {
            Link* pCurrent = pFirst;    //start at beginning of list
            while(pCurrent != NULL)     //until end of list,
            {
                Link* pOldCur = pCurrent;    //save current link
                pCurrent = pCurrent->pNext;  //move to next link
                delete pOldCur;              //delete old current
            }
        }

        void insertFirst(int id, double dd)
        {                               //make new link
            Link* pNewLink = new Link(id, dd);
            pNewLink->pNext = pFirst;   //it points to old first link
            pFirst = pNewLink;          //now first points to this
        }

        Link* find(int key)             //find link with given key
        {                               //(assumes non-empty list)
            Link* pCurrent = pFirst;            //start at 'first'
            while(pCurrent->iData != key)       //while no match,
            {
                if(pCurrent->pNext == NULL)      //if end of list,
                    return NULL;                 //didn't find it
                else                             //not end of list,
                    pCurrent = pCurrent->pNext; //go to next link
            }
            return pCurrent;                     //found it
        }

        bool remove(int key)                //remove link with given key
        {                                   //(assumes non-empty list)
            Link* pCurrent = pFirst;        //search for link
```

```
            Link* pPrevious = pFirst;
            while(pCurrent->iData != key)
            {
                if(pCurrent->pNext == NULL)
                    return false;           //didn't find it
                else
                {
                    pPrevious = pCurrent;  //go to next link
                    pCurrent = pCurrent->pNext;
                }
            }                               //found it
            if(pCurrent == pFirst)          //if first link,
                pFirst = pFirst->pNext;     //change first
            else                                    //otherwise,
                pPrevious->pNext = pCurrent->pNext; //bypass it
            delete pCurrent;                //delete link
            return true;                    //successful removal
        }

        void displayList()                  //display the list
        {
            cout << "List (first-->last): ";
            Link* pCurrent = pFirst;        //start at beginning of list
            while(pCurrent != NULL)         //until end of list,
            {
                pCurrent->displayLink();     //print data
                pCurrent = pCurrent->pNext;  //move to next link
            }
            cout << endl;
        }
};  //end class LinkList
//-----------------------------------------------------------------
int main()
{
    LinkList theList;                       //make list

    theList.insertFirst(22, 2.99);          //insert 4 items
    theList.insertFirst(44, 4.99);
    theList.insertFirst(66, 6.99);
    theList.insertFirst(88, 8.99);

    theList.displayList();                  //display list

    int findKey = 44;                       //find item
    Link* pFind = theList.find(findKey);
    if( pFind != NULL)
        cout << "Found link with key " << pFind->iData << endl;
    else
        cout << "Can't find link" << endl;

    int remKey = 66;                        //remove item
    bool remOK = theList.remove(remKey);
    if( remOK )
        cout << "Removed link with key " << remKey << endl;
    else
        cout << "Can't remove link" << endl;

    theList.displayList();                  //display list
    return 0;
}  //end main()
```

Funkcja składowa find()

Funkcja **find()** pracuje podobnie jak displayList() z programu linkList.cpp. Początkowo wskaźnik pCurrent jest ustawiony tak aby wskazywał na ten sam obiekt co pFirst. Podążając za kolejnymi elementami listy ustawiamy za każdym razem pCurrent na pCurrent->pNext. Każdy element listy sprawdzany jest na zwartość szukanej wartości. Jeśli szukana wartość zostanie odnaleziona zwracany jest wskaźnik do elementu listy. Jeśli osiągnięty zostanie koniec listy bez odnalezienia elementu przechowującego szukaną wartość to zwracany jest NULL.

Unikanie wycieku pamięci

Aby utworzyć listę jednostronnie wiązaną program tworzy obiekt klasy LinkList. Obiekt klasy LinkList stanowi listę złożoną z obiektów links dodawanych za pomocą insertFirst(). Każdy obiekt listy link jest tworzony za pomocą operatora **new**. Oznacza to, że w momencie wyjścia z programu (z funkcji main(){}) zostanie skasowany obiekt typu LinkList a obiekty typu Link już nie. Tym samym spowodujemy wyciek pamięci (pamięć na elementy typu Link została niezwolniona). Aby przeciwdziałać takim sytuacjom tworzymy destruktor. Destruktor ma za zadanie przy niszczeniu obiektu usunięcie tych elementów obiektu na które rezerwacja pamięci odbyła się za pomocą operatora new. W rozważanym przypadku konstrukcja destruktora polega na przejściu przez wszystkie elementy listy i usuwaniu ich krok po kroku.

2. Abstrakcyjne Typy Danych

Czym są Abstrakcyjne Typy Danych (ADTs)? Z grubsza rzecz biorąc jest to sposób patrzenia na struktury danych: skupiamy się na tym co należy zrobić (ze strukturą danych) ignorując sposób działania struktury danych.

ADT jest to koncepcja struktur danych odseparowanych od implementacji. Za przykłady abstrakcyjnych typów danych można uznać stosy i kolejki. Oddzielenie struktur danych od ich realizacji pokażemy na przykładzie stosów i kolejek, które to zrealizujemy za pomocą listy powiązanej a nie jak wcześniej tablic.

Implementacja stosu w oparciu o listę jednostronnie wiązaną

Implementacja push() i pop()

Na zajęciach 04-05 do utworzenia stosu użyliśmy klasy vector (w celu przechowywania danych). Operacje na stosie za pomocą funkcji push() i pop() zostały zrealizowane jako operacje na wektorach tj. operacji dokładania do stosu

```
stackVect[++top] = data;
```

i operacji zdejmowania ze stosu

```
data = stackVect[top--];
```

Przechowywanie danych stosu może być też zrealizowane przy użyciu listy jednostronnie wiązanej. W tym wypadku operacje pop() i push() będą zrealizowane przez operacje takie jak

theList.insertFirst(data)

oraz

data = theList.deleteFirst()

Użytkownik klasy stos wywołuje push() i pop() do dokładania i kasowania elementu, bez wiedzy (lub potrzeby wiedzy) o tym czy stos jest zaimplementowany jako tablica czy też jako lista jednostronnie wiązana.

Przykład 3 (C++, lista jednostronnie wiązana) Przykład implementacji stosu (klasa LinkStack) przy użyciu klasy LinkList zamiast tablicy.

[linkStack.cpp]

```
//linkStack.cpp
//demonstrates a stack implemented as a list
#include <iostream>
using namespace std;

class Link
{
    public:
        double dData;               //data item
        Link* pNext;                //next link in list

        Link(double dd) : dData(dd), pNext(NULL)
        {  }                        //constructor

        void displayLink()          //display ourself
        { cout << dData << " "; }
};  //end class Link
//-----------------------------------------------------------
class LinkList
{
    private:
        Link* pFirst;                   //ptr to first item on list

    public:
        LinkList() : pFirst(NULL)       //constructor
        {  }

        ~LinkList()                     //destructor (deletes links)
        {
            Link* pCurrent = pFirst;    //start at beginning of list
            while(pCurrent != NULL)     //until end of list,
            {
                Link* pOldCur = pCurrent;    //save current link
                pCurrent = pCurrent->pNext;  //move to next link
                delete pOldCur;              //delete old current
            }
        }

        bool isEmpty()                  //true if list is empty
        { return (pFirst==NULL); }

        void insertFirst(double dd)     //insert at start of list
        {                               //make new link
            Link* pNewLink = new Link(dd);
            pNewLink->pNext = pFirst;   //newLink --> old first
            pFirst = pNewLink;          //first --> newLink
        }
```

```
        double deleteFirst()            //delete first item
        {                               //(assumes list not empty)
            Link* pTemp = pFirst;       //save old first link
            pFirst = pFirst->pNext;     //remove it: first-->old next
            double key = pTemp->dData;  //remember data
            delete pTemp;               //delete old first link
            return key;                 //return deleted link's data
        }

        void displayList()              //display all links
        {
            Link* pCurrent = pFirst;    //start at beginning of list
            while(pCurrent != NULL)     //until end of list,
            {
                pCurrent->displayLink(); //print data
                pCurrent = pCurrent->pNext;  //move to next link
            }
            cout << endl;
        }
};  //end class LinkList
//---------------------------------------------------------------
class LinkStack
{
    private:
        LinkList* pList;                //pointer to linked list
    public:
        LinkStack()                     //constructor
        { pList = new LinkList; }       //make a linked list

        ~LinkStack()                    //destructor
        { delete pList; }               //delete the linked list

        void push(double j)             //put item on top of stack
        { pList->insertFirst(j); }

        double pop()                    //take item from top of stack
        { return pList->deleteFirst(); }

        bool isEmpty()                  //true if stack is empty
        { return ( pList->isEmpty() ); }

        void displayStack()
        {
            cout << "Stack (top-->bottom): ";
            pList->displayList();
        }
};  //end class LinkStack
//---------------------------------------------------------------
int main()
{

    LinkStack theStack;             //make stack

    theStack.push(20);              //push items
    theStack.push(40);

    theStack.displayStack();        //display stack (40, 20)

    theStack.push(60);              //push items
    theStack.push(80);
```

```
    theStack.displayStack();       //display (80, 60, 40, 20,)

    theStack.pop();                //pop items (80, 60)
    theStack.pop();

    theStack.displayStack();       //display stack (40, 20)
    return 0;
}  //end main()
```

Powiązania między klasami

Funkcja main() jest powiązana tylko z klasą LinkStack. Klasa LinkStack z kolei jest tylko powiązana z klasą LinkList. Między main() a klasą LinkList nie ma bezpośredniej komunikacji.

Implementacja stosu za pomocą listy jednostronnie wiązanej zamiast tablic pokazuje jak możemy odłączyć interfejs (widocznej/dostępnej dla użytkownika klasy) od (ukrytej) implementacji stosu.

Listy dwustronne (double-ended lists)

Implementację kolejki przeprowadzimy na liście dwustronnej. Lista dwustronna jest podobna do zwykłej listy jednostronnie wiązanej z tym, że obiekt list ma referencję zarówno na pierwszy pFirst jak i ostatni pLast element listy. Referencja na ostatni element listy pozwala na równie łatwe dołączanie nowego elementu listy na jej koniec jak w przypadku dołączania elementu na początek listy (oczywiście możliwe jest dołączanie elementu w przypadku listy jednostronnie wiązanej (single-ended list), przechodząc przez wszystkie elementy listy lecz jest to trudniejsze i mniej efektywne). Łatwy dostęp zarówno do elementu początkowego jak i końcowego listy jest odpowiedni w sytuacjach w których zwykła single-ended lista jest nie efektywna. Przykładem jest tutaj implementacja kolejki.

Przykład 4 (C++) Implementacja listy dwustronnej.

[firstLastList.cpp]

```
//firstLastList.cpp
//demonstrates list with first and last references
#include <iostream>
using namespace std;

class Link
{
    public:
        double dData;                  //data item
        Link* pNext;                   //ptr to next link in list

        Link(double d) : dData(d), pNext(NULL)  //constructor
        {  }

        void displayLink()             //display this link
        { cout << dData << " "; }
};  //end class Link
//-----------------------------------------------------------
class FirstLastList
```

```
{
    private:
        Link* pFirst;                   //ptr to first link
        Link* pLast;                    //ptr to last link

    public:
        FirstLastList() : pFirst(NULL), pLast(NULL)   //constructor
        {  }

        ~FirstLastList()                        //destructor
        {                                       //  (deletes all links)
            Link* pCurrent = pFirst;            //start at beginning
            while(pCurrent != NULL)             //until end of list,
            {
                Link* pTemp = pCurrent;         //remember current
                pCurrent = pCurrent->pNext;     //move to next link
                delete pTemp;                   //delete old current
            }
        }

        bool isEmpty()                          //true if no links
        { return pFirst==NULL; }

        void insertFirst(double dd)             //insert at front of list
        {
            Link* pNewLink = new Link(dd);      //make new link
            if( isEmpty() )                     //if empty list,
                pLast = pNewLink;               //newLink <-- last
            pNewLink->pNext = pFirst;           //newLink --> old first
            pFirst = pNewLink;                  //first --> newLink
        }

        void insertLast(double dd)              //insert at end of list
        {
            Link* pNewLink = new Link(dd);      //make new link
            if( isEmpty() )                     //if empty list,
                pFirst = pNewLink;              //first --> newLink
            else
                pLast->pNext = pNewLink;        //old last --> newLink
            pLast = pNewLink;                   //newLink <-- last
        }

        void removeFirst()                      //remove first link
        {                                       //(assumes non-empty list)
            Link* pTemp = pFirst;               //remember first link
            if(pFirst->pNext == NULL)           //if only one item
                pLast = NULL;                   //NULL <-- last
            pFirst = pFirst->pNext;             //first --> old next
            delete pTemp;                       //delete the link
        }

        void displayList()
        {
            cout << "List (first-->last): ";
            Link* pCurrent = pFirst;            //start at beginning
            while(pCurrent != NULL)             //until end of list,
            {
                pCurrent->displayLink();        //print data
                pCurrent = pCurrent->pNext;     //move to next link
            }
            cout << endl;
```

```
        }
};  //end class FirstLastList
//-----------------------------------------------------------
int main()
{
    FirstLastList theList;            //make a new list

    theList.insertFirst(22);          //insert at front
    theList.insertFirst(44);
    theList.insertFirst(66);

    theList.insertLast(11);           //insert at rear
    theList.insertLast(33);
    theList.insertLast(55);

    theList.displayList();            //display the list

    cout << "Deleting first two items" << endl;
    theList.removeFirst();            //remove first two items
    theList.removeFirst();

    theList.displayList();            //display again
    return 0;
}  //end main()
```

Implementacja kolejki przy pomocy list dwustronnej

Przykład 5 (C++, lista dwustronna) Program realizujący strukturę danych typu kolejka (możesz porównać z programem Queue.cpp z zajęć nr 5, przykład 2).

[linkQueue.cpp]

```
//linkQueue.cpp
//demonstrates queue implemented as double-ended list
#include <iostream>
using namespace std;

class Link
{
    public:
        double dData;                 //data item
        Link* pNext;                  //ptr to next link in list

        Link(double d) :dData(d), pNext(NULL)  //constructor
        {  }

        void displayLink()            //display this link
        { cout << dData << " "; }
};  //end class Link
//-----------------------------------------------------------
class FirstLastList
{
    private:
        Link* pFirst;                 //ptr to first link
        Link* pLast;                  //ptr to last link

    public:
        FirstLastList() : pFirst(NULL), pLast(NULL)  //constructor
        {  }
```

```
        ~FirstLastList()                        //destructor
        {                                       //  (deletes all links)
            Link* pCurrent = pFirst;            //start at beginning
            while(pCurrent != NULL)             //until end of list,
            {
                Link* pTemp = pCurrent;         //remember current
                pCurrent = pCurrent->pNext;     //move to next link
                delete pTemp;                   //delete old current
            }
        }

        bool isEmpty()                          //true if no links
        { return pFirst==NULL; }

        void insertLast(double dd)              //insert at end of list
        {
            Link* pNewLink = new Link(dd);      //make new link
            if( isEmpty() )                     //if empty list,
                pFirst = pNewLink;              //first --> newLink
            else
                pLast->pNext = pNewLink;        //old last --> newLink
            pLast = pNewLink;                   //newLink <-- last
        }

        double removeFirst()                    //delete first link
        {                                       //(assumes non-empty list)
            Link* pTemp = pFirst;               //remember first link
            double temp = pFirst->dData;
            if(pFirst->pNext == NULL)           //if only one item
                pLast = NULL;                   //null <-- last
            pFirst = pFirst->pNext;             //first --> old next
            delete pTemp;                       //delete the link
            return temp;
        }

        void displayList()
        {
            Link* pCurrent = pFirst;            //start at beginning
            while(pCurrent != NULL)             //until end of list,
            {
                pCurrent->displayLink();        //print data
                pCurrent = pCurrent->pNext;     //move to next link
            }
            cout << endl;
        }
};  //end class FirstLastList
//----------------------------------------------------------------
class LinkQueue
{
    private:
        FirstLastList theList;

    public:
        bool isEmpty()              //true if queue is empty
        { return theList.isEmpty(); }

        void insert(double j)       //insert, rear of queue
        { theList.insertLast(j); }

        double remove()             //remove, front of queue
        {  return theList.removeFirst();  }
```

```
        void displayQueue()
        {
            cout << "Queue (front-->rear): ";
            theList.displayList();
        }
};  //end class LinkQueue
//-----------------------------------------------------------------
int main()
{
    LinkQueue theQueue;        //make a queue

    theQueue.insert(20);       //insert items
    theQueue.insert(40);

    theQueue.displayQueue();   //display queue (20, 40)

    theQueue.insert(60);       //insert items
    theQueue.insert(80);

    theQueue.displayQueue();   //display queue (20, 40, 60, 80)

    cout << "Removing two items" << endl;
    theQueue.remove();         //remove items (20, 40)
    theQueue.remove();

    theQueue.displayQueue();   //display queue (60, 80)
    return 0;
}  //end main()
```

3. Listy wyspecjalizowane

Implementacja listy dwukierunkowej (doubly linked list)

Lista dwukierunkowa dostarcza możliwość przechodzenia listy w obie strony. W liście dwukierunkowej każdy element listy ma dwa wskaźniki (zamiast jednego jak przy liście dwustronnej). Pierwszy wskaźnik wskazuje na następny element (jak w zwykłej liście). Drugi wskazuje na element poprzedzający.

Początek deklaracji listy dwukierunkowej:

```
class Link
{
    public:
        double dData;       //data item
        Link* pNext;        //next link in list
        Link* pPrevious;    //previous link in list
};
```

Dokładanie elementu na początek listy

```
void insertFirst(double dd)          //insert at front of list
{
    Link* pNewLink = new Link(dd);   //make new link
    if( isEmpty() )                  //if empty list,
```

```
        pLast = pNewLink;           //newLink <-- last
    else
        pFirst->pPrevious = pNewLink; //newLink <-- old first
    pNewLink->pNext = pFirst;       //newLink --> old first
    pFirst = pNewLink;              //first --> newLink
}
```

Wstawianie elementu do listy po elemencie z szukaną zmienną

```
if(pCurrent==pLast)             //if last link,
{
    pNewLink->pNext = NULL;     //newLink --> null
    pLast = pNewLink;           //newLink <-- last
}
else                            //not last link,
{                               //newLink --> old next
    pNewLink->pNext = pCurrent->pNext;
                               //newLink <-- old next
    pCurrent->pNext->pPrevious = pNewLink;
}
pNewLink->pPrevious = pCurrent; //old current <-- newLink
pCurrent->pNext = pNewLink;     //old current --> newLink
```

Przykład 6 (C++) Program realizujący strukturę danych typu lista dwukierunkowa.

[doublyLinked.cpp]

```
//doublyLinked.cpp
//demonstrates doubly-linked list
#include <iostream>
using namespace std;

class Link
{
    public:
        double dData;                   //data item
        Link* pNext;                    //next link in list
        Link* pPrevious;                //previous link in list

    public:
        Link(double dd) :               //constructor
            dData(dd), pNext(NULL), pPrevious(NULL)
        {  }

        void displayLink()              //display this link
        { cout << dData << " "; }
};  //end class Link
//--------------------------------------------------------------
class DoublyLinkedList
{
    private:
        Link* pFirst;                   //pointer to first item
        Link* pLast;                    //pointer to last item

    public:
        DoublyLinkedList() :            //constructor
            pFirst(NULL), pLast(NULL)
        {  }
```

```
~DoublyLinkedList()              //destructor (deletes links)
{
    Link* pCurrent = pFirst;     //start at beginning of list
    while(pCurrent != NULL)      //until end of list,
    {
        Link* pOldCur = pCurrent;   //save current link
        pCurrent = pCurrent->pNext; //move to next link
        delete pOldCur;             //delete old current
    }
}

bool isEmpty()                       //true if no links
{ return pFirst==NULL; }

void insertFirst(double dd)          //insert at front of list
{
    Link* pNewLink = new Link(dd);  //make new link
    if( isEmpty() )                 //if empty list,
        pLast = pNewLink;           //newLink <-- last
    else
        pFirst->pPrevious = pNewLink; //newLink <-- old first
    pNewLink->pNext = pFirst;       //newLink --> old first
    pFirst = pNewLink;              //first --> newLink
}

void insertLast(double dd)           //insert at end of list
{
    Link* pNewLink = new Link(dd);  //make new link
    if( isEmpty() )                 //if empty list,
        pFirst = pNewLink;          //first --> newLink
    else
    {
        pLast->pNext = pNewLink;    //old last --> newLink
        pNewLink->pPrevious = pLast;//old last <-- newLink
    }
    pLast = pNewLink;               //newLink <-- last
}

void removeFirst()                   //remove first link
{                                    //(assumes non-empty list)
    Link* pTemp = pFirst;
    if(pFirst->pNext == NULL)        //if only one item
        pLast = NULL;                //null <-- last
    else
        pFirst->pNext->pPrevious = NULL;  //null <-- old next
    pFirst = pFirst->pNext;          //first --> old next
    delete pTemp;                    //delete old first
}

void removeLast()                    //remove last link
{                                    //(assumes non-empty list)
    Link* pTemp = pLast;
    if(pFirst->pNext == NULL)        //if only one item
        pFirst = NULL;                //first --> null
    else
        pLast->pPrevious->pNext = NULL;  //old previous --> null
    pLast = pLast->pPrevious;        //old previous <-- last
    delete pTemp;                    //delete old last
}

                                     //insert dd just after key
```

```
bool insertAfter(double key, double dd)
{                                       //(assumes non-empty list)
    Link* pCurrent = pFirst;            //start at beginning
    while(pCurrent->dData != key)       //until match is found,
    {
        pCurrent = pCurrent->pNext; //move to next link
        if(pCurrent == NULL)
            return false;               //didn't find it
    }
    Link* pNewLink = new Link(dd);  //make new link
    if(pCurrent==pLast)                 //if last link,
    {
        pNewLink->pNext = NULL;         //newLink --> null
        pLast = pNewLink;               //newLink <-- last
    }
    else                                //not last link,
    {                                   //newLink --> old next
        pNewLink->pNext = pCurrent->pNext;
                                        //newLink <-- old next
        pCurrent->pNext->pPrevious = pNewLink;
    }
    pNewLink->pPrevious = pCurrent; //old current <-- newLink
    pCurrent->pNext = pNewLink;     //old current --> newLink
    return true;                        //found it, did insertion
}

bool removeKey(double key)              //remove item w/ given key
{                                       //(assumes non-empty list)
    Link* pCurrent = pFirst;            //start at beginning
    while(pCurrent->dData != key)       //until match is found,
    {
        pCurrent = pCurrent->pNext; //move to next link
        if(pCurrent == NULL)
            return false;               //didn't find it
    }
    if(pCurrent==pFirst)                //found it; first item?
        pFirst = pCurrent->pNext;       //first --> old next
    else                                //not first
                                        //old previous --> old next
        pCurrent->pPrevious->pNext = pCurrent->pNext;

    if(pCurrent==pLast)                 //last item?
        pLast = pCurrent->pPrevious;  //old previous <-- last
    else                                //not last
                                        //old previous <-- old next
        pCurrent->pNext->pPrevious = pCurrent->pPrevious;
    delete pCurrent;                    //delete item
    return true;                        //successful deletion
}

void displayForward()
{
    cout << "List (first-->last): ";
    Link* pCurrent = pFirst;            //start at beginning
    while(pCurrent != NULL)             //until end of list,
    {
        pCurrent->displayLink();    //display data
        pCurrent = pCurrent->pNext; //move to next link
    }
    cout << endl;
}
```

```
        void displayBackward()
        {
            cout << "List (last-->first): ";
            Link* pCurrent = pLast;          //start at end
            while(pCurrent != NULL)          //until start of list,
            {
                pCurrent->displayLink();     //display data
                pCurrent = pCurrent->pPrevious; //go to previous link
            }
            cout << endl;
        }
};  //end class DoublyLinkedList
//--------------------------------------------------------------
int main()
{
    DoublyLinkedList theList;      //make a new list

    theList.insertFirst(22);       //insert at front
    theList.insertFirst(44);
    theList.insertFirst(66);

    theList.insertLast(11);        //insert at rear
    theList.insertLast(33);
    theList.insertLast(55);

    theList.displayForward();      //display list forward
    theList.displayBackward();     //display list backward

    cout << "Deleting first, last, and 11" << endl;
    theList.removeFirst();         //remove first item
    theList.removeLast();          //remove last item
    theList.removeKey(11);         //remove item with key 11

    theList.displayForward();      //display list forward

    cout << "Inserting 77 after 22, and 88 after 33" << endl;
    theList.insertAfter(22, 77);   //insert 77 after 22
    theList.insertAfter(33, 88);   //insert 88 after 33

    theList.displayForward();      //display list forward
    return 0;
}  //end main()
```

Opracowano na podstawie:
Robert Lafore – "Teach Yourself Data Structures And Algorithms In 24 Hours"