1. Struktury danych
   - stos (stack),
   - kolejki (queue).

Stos
Do tej pory poznaliśmy strukturę przechowującą dane w postaci tablic. Tablice są właściwe dla pewnego typu danych które można odnaleźć w aplikacjach bazodanowych.

Stosy i kolejki jako narzędzia programisty
Stos jest to typ struktury często używany jako narzędzie programisty. Są one głównie koncepcyjną pomocą niż strukturą przechowującą dane. Ich czas życia jest typowo krótszy niż struktur bazodanowych. Są one tworzone i używane do wykonywania określonego zadania podczas działania funkcji w ramach programu (gdy zadanie jest ukończone są one usuwane).

Stosy i kolejki: ograniczony dostęp do danych
W przeciwieństwie do tablic, w stosach i kolejkach dostęp jest ograniczony tylko do jednego elementu (tylko jeden element może być przeczytany lub usunięty). Interfejs jest tak zaprojektowany aby egzekwować tego typu ograniczony dostęp.

Stosy i kolejki: bardziej abstrakcyjne byty
Tego typu struktury są głównie zdefiniowane przez ich interfejs. Interfejs określa jakie operacje mogą być na tych strukturach przeprowadzane.

Stos pozwala na dostęp do tylko jednego elementu danych: do ostatniego wstawionego. Usunięcie elementu pozwala uzyskać dostęp do elementu go poprzedzającego.

Przykład 1 (C++) Implementacja stosu w postaci klasy StackX.

[Stack.cpp]

```
//Stack.cpp
//demonstrates stacks
#include <iostream>
#include <vector>
using namespace std;

class StackX{
    private:
        int maxSize;                        //size of stack vector
        vector<double> stackVect;           //stack vector
        int top;                            //top of stack

    public:
        StackX(int s) : maxSize(s), top(-1) //constructor
        {
            stackVect.reserve(maxSize);     //size the vector
        }

        void push(double j)                 //put item on top
        {
            stackVect[++top] = j;           //increment top,
        }                                   //insert item

        double pop()                        //take item from top
        {
```

```
            return stackVect[top--];          //access item,
        }                                     //decrement top

        double peek()                         //peek at top of stack
        {
            return stackVect[top];
        }

        bool isEmpty()                        //true if stack is empty
        {
            return (top == -1);
        }

        bool isFull()                         //true if stack is full
        {
            return (top == maxSize-1);
        }
};  //end class StackX
//--------------------------------------------------------------
int main(){
    StackX theStack(10);              //make new stack, size 10
    theStack.push(20);                //push items onto stack
    theStack.push(40);
    theStack.push(60);
    theStack.push(80);

    while( !theStack.isEmpty() )      //until it's empty,
    {                                 //delete item from stack
        double value = theStack.pop();
        cout << value << " ";         //display it
    }  //end while
    cout << endl;

    return 0;
}  //end main()
```

Kolejki (Queues)

Kolejka jest strukturą danych podobną do stosu. W kolejce pierwszy element jest też pierwszym do usunięcia (FIFO – First In, First Out). W stosie ostatni element jest pierwszym do usunięcia (LIFO – Last In, First Out).

Przykład 2 (C++) Program realizujący strukturę danych typu kolejka. Program składa się z klasy Queue z metodami: insert(), remove(), peek(), isFull(), isEmpty() oraz size().

[Queue.cpp]

```
//Queue.cpp
//demonstrates queue
#include <iostream>
#include <vector>
using namespace std;

class Queue{
    private:
        int maxSize;
        vector<int> queVect;
        int front;
        int rear;
```

```
        int nItems;

    public:
        //constructor
        Queue(int s) : maxSize(s), front(0), rear(-1), nItems(0)
        { queVect.resize(maxSize); }

        void insert(int j)                  //put item at rear of queue
        {
            if(rear == maxSize-1)           //deal with wraparound
                rear = -1;
            queVect[++rear] = j;            //increment rear and insert
            nItems++;                       //one more item
        }

        int remove(){                       //take item from front of queue
            int temp = queVect[front++];    //get value and incr front
            if(front == maxSize)            //deal with wraparound
                front = 0;
            nItems--;                       //one less item
            return temp;
        }

        int peekFront()                     //peek at front of queue
        { return queVect[front]; }

        bool isEmpty()                      //true if queue is empty
        { return (nItems==0); }

        bool isFull()                       //true if queue is full
        { return (nItems==maxSize); }

        int size()                          //number of items in queue
        { return nItems; }
};  //end class Queue
//-----------------------------------------------------------------
int main(){
    Queue theQueue(5);              //queue holds 5 items

    theQueue.insert(10);            //insert 4 items
    theQueue.insert(20);
    theQueue.insert(30);
    theQueue.insert(40);

    theQueue.remove();              //remove 3 items
    theQueue.remove();              //    (10, 20, 30)
    theQueue.remove();

    theQueue.insert(50);            //insert 4 more items
    theQueue.insert(60);            //    (wraps around)
    theQueue.insert(70);
    theQueue.insert(80);

    while( !theQueue.isEmpty() )    //remove and display
    {                               //    all items
        int n = theQueue.remove();  //(40, 50, 60, 70, 80)
        cout << n << " ";
    }
    cout << endl;
    return 0;
}  //end main()
```

W klasie Queue znajdują się oprócz front (przód) i rear (koniec) składowa nItems numerująca obecną liczbę danych. Inne implementacje kolejki nie używają tej składowej.

Funkcja składowa: insert()
Funkcja składowa insert() zakłada, że kolejka nie jest pełna. Normalnie powinniśmy wywołać funkcje insert() tylko po uprzednim wywołaniu funkcji isFull() i sprawdzeniu, że zwróci wartość false (zazwyczaj korzystnie jest umieścić sprawdzenie czy kolejka jest pełna wewnątrz programu insert() i wyrzucić wyjątek w przypadku gdy ma miejsce próba wstawienia elementu do pełnej kolejki).

Normalnie wstawianie odbywa się przez inkrementacje rear a następnie wstawienie do komórki wskazywanej przez rear. Jeśli rear osiągnie koniec tablicy (dokładnie koniec wektora) tzn. jeśli `if(rear == maxSize-1)` wówczas musi nastąpić zawinięcie wokół dołu tablicy. Zrealizowane jest to przez ustawienie rear na -1, więc gdy nastąpi inkrementacja (patrz wnętrze insert()) rear przyjmie wartość 0, a składowa nItems zostanie zwiększona o jeden.

Funkcja składowa: remove()
Funkcja składowa remove() zakłada, że kolejka nie jest pusta. Sprawdzając przed wywołaniem remove(), że isEmpty() zwraca false możemy upewnić się, że kolejka nie jest pusta.

Usuwanie zawsze zaczyna się od obliczenia wartości w zmiennej front (`int temp = queVect[front++]`) i dopiero w dalszej kolejności zwiększenia front o jeden. Jednakże gdy wartość zmiennej front ma wskazywać poza koniec tablicy, musi nastąpić zawinięcie wokół 0. Zwracana wartość jest przechowywana tymczasowo do momentu gdy ta możliwość zostanie sprawdzona, po czym zmniejszana jest wartość zmiennej nItems o jeden.

Funkcja składowa: peekFront()
Funkcja ta zwraca wartość przechowywaną w wektorze o indeksie front.

Kolejki priorytetowe (Priority Queues)

Przykład 3 (C++) Program realizujący strukturę danych typu kolejka priorytetowa.

[priorityQ.cpp]

```
//priorityQ.cpp
//demonstrates priority queue
#include <iostream>
#include <vector>
using namespace std;

class PriorityQ{
    //vector in sorted order, from max at 0 to min at size-1
    private:
        int maxSize;
        vector<double> queVect;
        int nItems;

    public:
        PriorityQ(int s) : maxSize(s), nItems(0)     //constructor
```

```
        { queVect.resize(maxSize); }

        void insert(double item)                    //insert item
        {
            int j;
            if(nItems==0)                           //if no items,
                queVect[nItems++] = item;           //insert at 0
            else                                    //if items,
            {
                for(j=nItems-1; j>=0; j--)          //start at end,
                {
                    if( item > queVect[j] )         //if new item larger,
                        queVect[j+1] = queVect[j];  //shift upward
                    else                            //if smaller,
                        break;                      //done shifting
                }  //end for
                queVect[j+1] = item;                //insert it
                nItems++;
            }  //end else (nItems > 0)
        }  //end insert()

        double remove()                             //remove minimum item
        { return queVect[--nItems]; }

        double peekMin()                            //peek at minimum item
        { return queVect[nItems-1]; }

        bool isEmpty()                             //true if queue is empty
        { return (nItems==0); }

        bool isFull()                               //true if queue is full
        { return (nItems == maxSize); }

};  //end class PriorityQ
//-----------------------------------------------------------------
int main()
{
    PriorityQ thePQ(5);                 //priority queue, size 5

    thePQ.insert(30);                   //unsorted insertions
    thePQ.insert(50);
    thePQ.insert(10);
    thePQ.insert(40);
    thePQ.insert(20);

    while( !thePQ.isEmpty() )
    {                                   //sorted removals
        double item = thePQ.remove();
        cout << item << " ";            //10, 20, 30, 40, 50
    }  //end while
    cout << endl;
    return 0;
}  //end main()
```

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