Implementarea listă, stivă şi coadă în varianta standard:

```
================= List.h ============
#ifndef _LIST_
#define _LIST_

#include <iostream.h>

#define UNDEFINED -1

enum Where {CAP, COADA, MIJLOC};
enum boolean {FALSE, TRUE};

class Node {
            int inf;
            Node* next;

            Node(int _inf, Node* p) {
              inf = _inf; next = p;
            }

      public:
            int getInfo() {
              return inf;
            }

            friend class List;
};

class List {
      protected:
            Node* head;
            Node* tail;
            boolean sorted;

      public:
            List(boolean type = FALSE) {
              head = tail = NULL;
              sorted = type;
            }

            ~List();
            List& operator= (List&);

            void addElement(int, Where);
            void insertElement(int);
            boolean deleteElement(int);
            boolean  searchElement(int);
            void clear();
            void sort();

            void listare();
};

#endif

=============== List.cpp ==========
#include "list.h"

List::~List() {
  clear();
}
```

```
List& List::operator= (List& l) {
  if (this == &l)
    return *this;

  clear();
  Node *p = l.head;
  while (p) {
    addElement(p->inf, COADA);
    p = p->next;
  }
  sorted = l.sorted;
  return *this;
}

void List::addElement(int value, Where pos) {
  if (head == NULL || pos == CAP) {
    head = new Node(value, head);
    if (tail == NULL)
      tail = head;
  }
  else
    if (pos == COADA) {
      tail->next = new Node(value, NULL);
      tail = tail->next;
    }
  sorted = FALSE;
}

void List::insertElement(int value) {
  Node *curent;
  Node *previous;

  sort();
  previous = NULL; curent = head;
  while (curent &&  curent->inf < value) {
    previous = curent;
    curent = curent->next;
  }

  if (!previous) {
    addElement(value, CAP);
  }
  else {
    curent = new Node(value, curent);
    previous->next = curent;
    if (!curent->next)
      tail = curent;
  }
  sorted = TRUE;
}

boolean List::deleteElement(int value) {
  Node *curent, *previous;

  previous = NULL; curent = head;
  while (curent && curent->inf != value) {
    previous = curent;
    curent = curent->next;
  }
```

```
  if (curent) {
    if (!previous) {
      curent = head;
      if (tail == head)
        tail = NULL;
      head = head->next;
      delete curent;
    }
    else {
      previous->next = curent->next;
      if (tail == curent)
        tail = previous;
      delete curent;
    }
    return TRUE;
  }
  return FALSE;
}

boolean List::searchElement(int value) {
  Node *curent;

  while (curent && curent->inf != value)
    curent = curent->next;
  return (curent != NULL);
}

void List::clear() {
  Node *curent;

  while (head) {
    curent = head;
    head = head->next;
    delete curent;
  }
  tail = NULL;
  sorted = TRUE;
}

void List::sort() {
  if (sorted)
    return;

  sorted = TRUE;
  if (tail == NULL)
    return;

  Node *p, *q;
  int tmp;

  p = head;
  while (p) {
    q = p->next;
    while (q) {
      if (p->inf > q->inf) {
        tmp = p->inf; p->inf = q->inf; q->inf = tmp;
      }
      q = q->next;
    }
    p = p->next;
  }
```

```
}

void List::listare() {
  Node *curent = head;

  while (curent) {
    cout << curent -> inf << " ";
    curent = curent->next;
  }
  cout << endl;
}
```

========== Coada.h ==========

```
#ifndef _COADA_
#define _COADA_

#include "list.h"

class Coada: public List {
      public:
            Coada() : List() {}

            boolean isEmpty(void);
            boolean isFull(void);
            void enQueue(int);
            int deQueue(void);
};

#endif
```

========== Coada.cpp ==========

```
#include "coada.h"

boolean Coada::isEmpty(void) {
  return (tail == NULL);
}

boolean Coada::isFull(void) {
  return FALSE;
}

void Coada::enQueue(int value) {
  addElement(value, COADA);
}

int Coada::deQueue(void) {
  if (head == NULL)
    return UNDEFINED;

  int value = head->getInfo();
  deleteElement(value);
  return value;
}
```

========== Stiva.h ==========

```
#ifndef _STACK_
#define _STACK_

#include "list.h"
```

```
class Stiva: public List {
      public:
            Stiva() : List(FALSE) {};

            boolean isEmpty(void);
            boolean isFull(void);
            void push(int);
            int pop(void);
            int peek(void);
};
#endif
```

Implementarea listă, stivă şi coadă folosind template-uri:

========== List.h ==========

```
#ifndef _LIST_
#define _LIST_

#include <iostream.h>

#define UNDEFINED -1

enum Where {CAP, COADA, MIJLOC};
enum boolean {FALSE, TRUE};

template <class T>
class Node {
            T inf;
            Node<T>* next;

            Node(T _inf, Node* p) : inf(_inf) {
              next = p;
            }

      public:
            T getInfo() {
              return inf;
            }

            friend class List<T>;
};

template <class T>
class List {
      protected:
            Node<T>* head;
            Node<T>* tail;
            boolean sorted;

      public:
            List(boolean type = FALSE) {
              head = tail = NULL;
              sorted = type;
            }

            ~List() {
              clear();
            }
```

```
            List& operator= (List&);

            void addElement(T, Where);
            void insertElement(T);
            boolean deleteElement(T);
            boolean  searchElement(T);
            void clear();
            void sort();

            void listare();
};

template <class T>
List<T>& List<T>::operator= (List<T>& l) {
  if (this == &l)
    return *this;

  clear();
  Node<T> *p = l.head;
  while (p) {
    addElement(p->inf, COADA);
    p = p->next;
  }
  sorted = l.sorted;
  return *this;
}

template <class T>
void List<T>::addElement(T value, Where pos) {
  if (head == NULL || pos == CAP) {
    head = new Node<T>(value, head);
    if (tail == NULL)
      tail = head;
  }
  else
    if (pos == COADA) {
      tail->next = new Node<T>(value, NULL);
      tail = tail->next;
    }
  sorted = FALSE;
}

template <class T>
void List<T>::insertElement(T value) {
  Node<T> *curent;
  Node<T> *previous;

  sort();
  previous = NULL; curent = head;
  while (curent &&  curent->inf < value) {
    previous = curent;
    curent = curent->next;
  }
  if (!previous) {
    addElement(value, CAP);
  }
  else{
    curent = new Node<T>(value, curent);
    previous->next = curent;
    if (!curent->next)
      tail = curent;
```

```
  }
  sorted = TRUE;
}

template <class T>
boolean List<T>::deleteElement(T value) {
  Node<T> *curent, *previous;

  previous = NULL; curent = head;
  while (curent && curent->inf != value) {
    previous = curent;
    curent = curent->next;
  }

  if (curent) {
    if (!previous) {
      curent = head;
      if (tail == head)
        tail = NULL;
      head = head->next;
      delete curent;
    }
    else {
      previous->next = curent->next;
      if (tail == curent)
      tail = previous;
      delete curent;
    }
    return TRUE;
  }
  return FALSE;
}

template <class T>
boolean List<T>::searchElement(T value) {
  Node<T> *curent;

  while (curent && curent->inf != value)
    curent = curent->next;
  return (curent != NULL);
}

template <class T>
void List<T>::clear() {
  Node<T> *curent;

  while (head) {
    curent = head;
    head = head->next;
    delete curent;
  }
  tail = NULL;
  sorted = TRUE;
}

template <class T>
void List<T>::sort() {
  if (sorted)
    return;

  sorted = TRUE;
```

```
  if (tail == NULL)
    return;

  Node<T> *p, *q;
  T tmp;

  p = head;
  while (p) {
    q = p->next;
    while (q) {
      if (p->inf > q->inf) {
        tmp = p->inf; p->inf = q->inf; q->inf = tmp;
      }
      q = q->next;
    }
    p = p->next;
  }
}

template <class T>
void List<T>::listare() {
  Node<T> *curent = head;

  while (curent) {
    cout << curent -> inf << " ";
    current = curent->next;
  }
  cout << endl;
}
#endif
```

========== Stiva.h ==========

```
#ifndef _STACK_
#define _STACK_

#include "list.h"

template <class T>
class Stiva: public List<T> {
      public:
            Stiva() : List<T>(FALSE) {};

            boolean isEmpty(void);
            boolean isFull(void);
            void push(T);
            T pop(void);
            T peek(void);
};

...

#endif
```

========== Coada.h ==========

```
#ifndef _COADA_
#define _COADA_

#include "list.h"
```

```
template <class T>
class Coada: public List<T> {
      public:
            Coada() : List<T>() {}

            boolean isEmpty(void);
            boolean isFull(void);
            void enQueue(T);
            T deQueue(void);
};

template <class T>
boolean Coada<T>::isEmpty(void) {
  return (tail == NULL);
}

template <class T>
boolean Coada<T>::isFull(void) {
  return FALSE;
}

template <class T>
void Coada<T>::enQueue(int value) {
  addElement(value, COADA);
}

template <class T>
T Coada<T>::deQueue(void) {
  if (head == NULL)
    return UNDEFINED;

  T value = head->getInfo();
  deleteElement(value);
  return value;
}

#endif
```

========== Strlist.cpp ==========

```
#include <assert.h>
#include <string.h>
#include "list.h"

#define SLEN 30

class String{
            char name[SLEN];
      public:
            String(char* str = "test") {
              assert(strlen(str) <= SLEN);
              strcpy(name, str);
            }

            String(String& string) {
              strcpy(name, string.name);
            }

            String& operator = (String& string) {
              strcpy(name, string.name);
              return *this;
            }
```

```
            int operator < (String& string) {
              return (strcmp(name, string.name) < 0) ? 1 : 0;
            }

            int operator > (String& string) {
              return (strcmp(name, string.name) > 0) ? 1 : 0;
            }

            int operator == (String& string) {
              return (strcmp(name, string.name) == 0) ? 1 : 0;
            }

            int operator != (String& string) {
              return (strcmp(name, string.name) != 0) ? 1 : 0;
            }

            friend ostream& operator<< (ostream&, String&);
};

ostream& operator<< (ostream& os, String& string) {
  os << string.name << endl;
  return os;
}

void main(void) {
  List<String> ss;

  ss.addElement(*new String("Aconcagua"), COADA);
  ss.addElement(*new String("FujiYama"), CAP);
  ss.insertElement(*new String("Popocatepetl"));
  ss.insertElement(*new String("Vezuviu"));
  ss.addElement(*new String("Etna"), CAP);
  ss.listare();

  ss.deleteElement(*new String("FujiYama"));
  ss.listare();
}
```

========== Testcoad.cpp ==========

```
#include "coada.h"

void main(void) {
  Coada<int> c;

  c.enQueue(4);
  c.enQueue(7);
  c.enQueue(3);
  c.listare();

  c.deQueue();
  c.deQueue();
  c.listare();
}
```

Temă

1. Să se implementeze definiția clasei *Stiva* al cărei antet se află în **Stiva.h**.

Să se utilizeze această clasă pentru a calcula valorile funcției *Manna-Pnuelli* (variantă nerecursivă):

```
#include "stiva.h"

#define F 1

int m;

int mannaRec(int m) {
  if (m >= 12)
    return m - 1;
  else
    return mannaRec(mannaRec(m + 2));
}

int manna(int m) {
  Stiva s;
  int x;

  s.push(F);
  x = m;
  while (!s.isEmpty()) {
    s.pop();
    if (x >= 12)
      x = x - 1;
    else {
      s.push(F);
      s.push(F);
      x = x + 2;
    }
  }
  return x;
}

void main(void) {
  cout << "m = "; cin >> m;
  cout << "f(" << m << ")=" << manna(m) << endl;
  cout << "f(" << m << ")=" << mannaRec(m) << endl;
}
```

2. Să se implementeze definiția clasei *Stiva* folosind template-uri după modelul prezentat la clasele *List* şi *Coada*.
3. a) Să se modifice programul de la problema 1 pentru a putea utiliza stiva parametrizată pentru calculul valorilor funcției *Manna-Pnuelli*.
   b) Folosindu-se o stivă să se verifice dacă un şir de paranteze, ce formează o expresie, se închid corect. Şirul este un tablou de caractere în care se presupune că există numai paranteze deschise şi închise.

Bibliografie

http://inf.ucv.ro/~mirel/courses/poo/poo.html

1. *The C++ Programming Language*, B. Stroustrup, Addison-Wesley.
(Cap. 13 Templates)