A TYPICAL CLASS – ARRAY BASED STACK

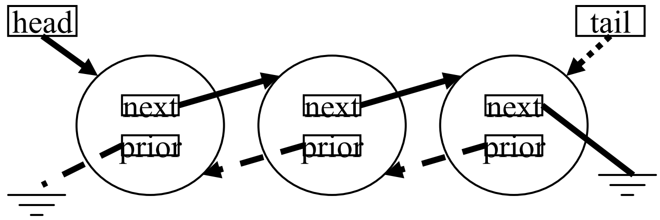
HEADER

```
class IntArrayStack {
private:

  int *data, // pointer to dynamically allocated array
      capacity; // capacity of stack, size of array
  bool elastic; // true if capacity of stack (i.e. the size of
                //  the array) can be changed as required.
  int count; // number of values on stack

public:

  // creates an "elastic" stack
  IntArrayStack ();
  // creates a stack with a fixed capacity
  IntArrayStack (int capacity);
  IntArrayStack (const IntArrayStack &otherStack);
  ~IntArrayStack ();
  // throws an "overflow_error" exception if the stack
  //  has a fixed capacity and is full.
  void push (int value);
  // throws an "overflow_error" exception if stack empty
  int pop ();
  bool isEmpty() const { return count == 0; }
  int getcount() const { return count; }
  // copies contents only (capacity attributes are not
  //  copied).  throws an "overflow_error" exception if the
  // destination stack has inadequate capacity
  IntArrayStack& operator=
                        (const IntArrayStack &otherStack);
};
```

IMPLEMENTATION

```
static const int initialCapacity = 10,
           capacityIncrement = 5;

IntArrayStack::IntArrayStack () {
   data = new int [initialCapacity];
  capacity = initialCapacity; elastic = true;
  count = 0;
}

IntArrayStack::IntArrayStack (int capacity) {
  if (capacity <= 0) {
    throw invalid_argument
      ("IntArrayStack::IntArrayStack - invalid capacity");
  }
  data = new int [capacity];
  this -> capacity = capacity; elastic = false;
  count = 0;
}

IntArrayStack::IntArrayStack
        (const IntArrayStack &otherStack) {
  data = new int [otherStack.capacity];
  for (int i = 0; i < count; i++) {
    data[i] = otherStack.data[i];
  }
  capacity = otherStack.capacity;
  elastic = otherStack.elastic;
  count = otherStack.count;
}

IntArrayStack::~IntArrayStack () {
  delete [] data;
}

void IntArrayStack::push (int value) {
  int *temp, i;
  if (elastic && (count == capacity)) {
    // time to increase the capacity
    capacity += capacityIncrement;
    temp = new int [capacity];
    for (i = 0; i < count; i++) temp[i] = data[i];
    delete [] data;
    data = temp;
  } else if (count == capacity) {
    throw overflow_error
        ("IntArrayStack::push - stack overflow");
  }
  data[count++] = value;
}

int IntArrayStack::pop () {
  if (count == 0) {
    throw overflow_error
        ("IntArrayStack::pop - stack underflow");
  }
  return data[--count];
}

IntArrayStack& IntArrayStack::operator=
        (const IntArrayStack &otherStack) {
  if (capacity < otherStack.count) {
    if (!elastic) {
      throw overflow_error
          ("IntArrayStack - copy overflow");
    }
    delete [] data;
    data = new int[otherStack.count];
    capacity = otherStack.count;
  }
  for (int i = 0; i < otherStack.count; i++) {
    data[i] = otherStack.data[i];
  }
  count = otherStack.count;
  return *this;
}
```

LINKED LISTS:



Doubly linked lists only   Only if tail pointer used

```
// outline of typical list-based class
class Typical {
public:

  // public methods go here

private:

  class Node {
    int data; // could be anything
    // prior pointers are optional
    Node *next, *prior;
  }

  // tail pointer is optional
  Node *head, *tail;
  int size;

}
```

```
// typical list traversal
Node *p, *c;
p = null; c = head;

while ( (c != NULL) &&
        (c -> data != data) ) {
  p = c; c = c-> next;
}

if (c == NULL) {
  // not found
}
```

String2002 CLASS:

```
String2002 a, b;
cout << "Type two words separated by a space: "
cin >> a >> b;
// String Comparisons:
// case sensitive
if (a < b ) {…}
elseif ( a==b ) {…}
elseif (a > b ) {…}
// case insensitive
if (a.isEqualCaseInsensitive("STOP")) == 0) {
   cout << "First word is STOP " <<
       "(ignoring case).\n";
} else {
   cout << "First word is not STOP.\n";
}
```

BINARY TREES:

```
// outline of typical tree-based class
class Typical {
public:

  // public methods go here

private:

  class TNode {
    int value // could be anything
    TNode *left, *right;
  }

  TNode *root;
  int size;

}

// typical recursive traversal

void Typical::subPrint (TNode *subRoot) {
  if (subRoot == NULL) return;
  subPrint (subRoot -> left);
  cout << subRoot -> value;
  subPrint (subRoot -> right);
}

void Typical::print() {
  subPrint (root);
}
```

HASHING:

Linear rehashing: if table[key] is in use, try table[key+1], and so on.

Hashing with buckets: table[key] is the head pointer for a linked list.

PARSING:

```
// a simple grammar
<operator>  ::=  + | - | * | /
<digit>  ::= 0|1|2|3|4|5|6|7|8|9
<number> ::= <digit> | <digit><number>
 <expression> ::= <number> |
                ( <expression><operator><expression> )
```

{…}* means that … is repeated zero or more times

```
// Pseudo code for recognizeExpression (for this grammar):
bool recognizeExpression () {

 if the the next input character is '('  {
   cross out  the '('
   if (!recognizeExpression()) return false;
   if (!recognizeOperator()) return false;
   if (!recognizeExpression())) return false;
   if the current input character isn't ')'  return false;
   cross out  the ')'
   return true;
 } else {
   return recognizeNumber ();
 }
```