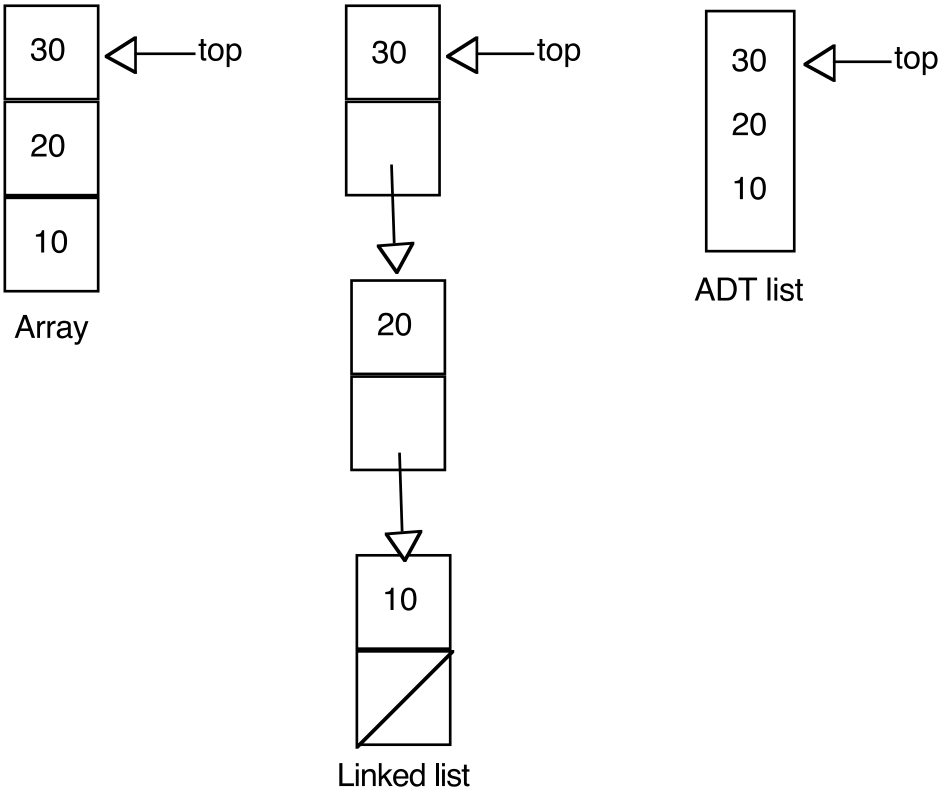
Chapter 6-3, Lecture notes

Implementations of the ADT Stack

1. We'll try three different implementations of the ADT stack.
   1.1 The first uses an array, second uses a linked list and the third uses the ADT list.
   1.2 Like in Figure 6-4 on page 296:



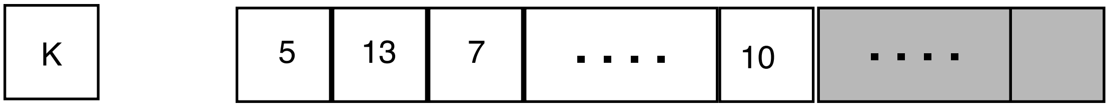
   1.3 And here is the StackException used for the stack implementations:

```
#include <stdexcept>
#include <string>

using namespace std;

class StackException : public logic_error
{
public:
        StackException(const string & message = “”)
                : logic_error(message.c_str())
        {}  // end constructor
};  // end StackException
```

1.4 Put the above in the StackException.h header file.

1.5 Using an array to implement a stack means that you have a variable, like “top”, point to the top item on the stack so that “items[top]” would give you the top of the stack.

1.5.1 Like so (figure 6-5 page 297):



1.5.2 The following definition would use the constructor as the ADT “createStack” and the destructor for the “destroyStack”.

1.5.3 The header file is:

```
/** @file StackA.h */

#include “StackException.h”
const int MAX_STACK = maximum-size-of-stack; // you decide the size
typedef desired-type-of-stack-item StackItemType; // you decide the “type-of-stack-item”

/** ADT stack – Array-based implementation */
class Stack
{
public:
// constructors and destructors:
        /** Default constructor */
        Stack();
        // copy constructor and destructor are supplied by the compiler

// stack operations:
        /** Determines whether this stack is empty
         *  @pre None
         *  @post None
         *  @return  True if this stack is empty, otherwise returns false */
        bool isEmtpy() const;

        /** Adds an item to the top of this stack
         *  @pre  newItem is the item to be added
         *  @post  If the insertion is successful, newItem is on the top of this stack
         *  @parm newItem   The given StackItemType
         *  @throw StackException   If the item cannot be placed on this stack */
        void push(const StackItemType& newItem) throw(stackException);
        /** Removes the top of this stack
         *  @pre None
         *  @post If this stack is not empty, the item that was added most recently
         *           is removed. However, if this stack is empty, deletion is impossible
         *  @throw  StackException  If this stack is empty */
        void pop() throw(StackException)

        /** Retrieves and removes the top of this stack
         *  @pre  None
         *  @post  If this stack is not empty, stackTop contains the item that was added
         *            most recently and the item is remove. However, if this stack is empty,
         *            deletion is impossible and stackTop is unchanged
         *  @throw  StackException   If this stack is empty */
        void pop(StackItemType& stackTop) throw(StackException);

        /** Retrieves the top of this stack
         *  @pre  None
         *  @post  If this stack is not empty, stackTop contains the item that was added
         *            most recently. However, if this stack is empty, the operation fails and
         *            stackTop is unchanged. This stack is unchanged
         *  @throw  StackException    If this stack is empty */
        void getTop(StackItemType& stackTop) const throw(StackException);

private:
        StackItemType items[MAX_STACK];  // Array of stack items
        int       top;     // Index to top of stack
}; // end Stack
// End header file
```

1.5.4 The implementation for the above follows below:

```
/** @file StackA.cpp */

#include “StackA.cpp */

Stack::Stack() : top(-1)
{
}

bool Stack::isEmpty() const
{
        return top < 0;
} // end isEmpty
```

```
void Stack::push(const StackItemType& newItem) throw(StackException)
{
// if stack has no more room for another item
        if (top > MAX_STACK-1)
                throw StackException(“StackException: stack full on push”);
        else
        {       ++top;
                item[top] = newItem;
        } // end if
} // push

void Stack::pop() throw(StackException)
{
        if (isEmpty())
                throw StackException(“StackException: stack empty on pop”);
        else
                --top; // stack is not empty; pop top
} // end pop

void Stack::pop(StackItemType& stackTop) throw(StackException)
{
        if (isEmpty())
                throw StackException(“StackException: stack empty on pop”);
        else
        {       // stack is not empty; retrieve top
                stackTop = items[top];
                --top; // pop top
        } // end if
} // end pop

void Stack::getTop(StackItemType& stackTop) const throw(StackException)
{
        if (isEmpty())
                throw StackException(“StackException: stack empty on getTop”);
        else  // stack is not empty; retrieve top
                stackTop = items[top];
} // end getTop
// End of implementation file
```

1.5.5 So the program part (or main) will look like:

```
#include <iostream>
#include “StackA.h”
using namespace std;
```

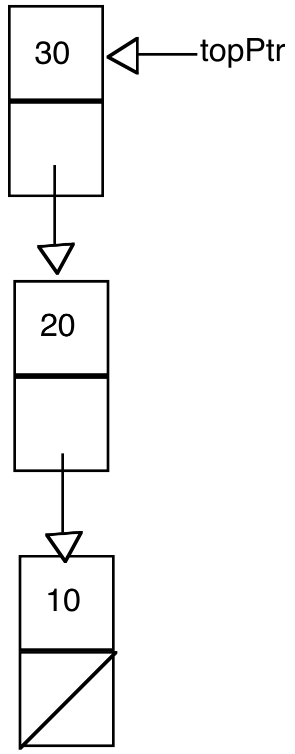
```
int main()
{
        StackItemType anITem;
        Stack aStack;

        cin >> anItem;           // read an item
        aStack.push(anItem);  // push it onto stack
        ….
}
```

1.5.6 We put up the ADT wall when we made the array and top private.

2. Now with implementing the stack as a linked list, we are not limited with the size of a static array.

   2.1 We can have topPtr point to the first of the linked list. Adding to the stack (push) and removing from the stack (pop) would involve the top of the linked list only, which makes handling the linked list easier.

   2.2 Like so (see figure 6-6 on page 301):



Linked list

   2.3 The header file is so (minus comments):

```
/** @file StackP.h */

#include “StackException.h”
typedef desired-type-of-stack-item StackItemType;

/**  ADT stack – Pointer-based implementation */
class Stack
{
public:
// Constructors and destructor:

        /** Default constructor
         *  @param aStack  The stack to copy */
        Stack(const Stack& aStack)

        /** Destructor */
        ~Stack();

// Stack operations:
        bool isEmpty() const;
        void push(const StackItemType& newItem) throw(StackException);
        void pop() throw(StackException);
        void pop(StackItemType& stackTop) throw(StackException);
        void getTop(StackItemType& stackTop) const throw(StackException);

private:
        struct StackNode        // A node on the stack
        {
                StackItemType item;  // A data item on the stack
                StackNode      *next;  // Pointer to next node
        }; // end StackNode

        StackNode *topPtr;    // Pointer to first node in the stack

};  // end Stack
// End of header file
```

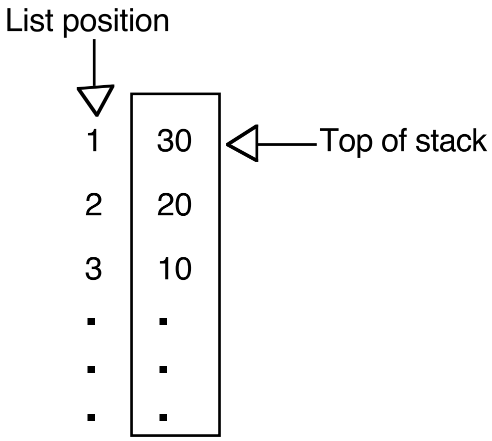
2.4 The implementation file is:

```
/** @file StackP.cpp  */

#include <cstddef>     // for NULL
#include <new>         // for bad_alloc
#include “StackP.h”   // header file
```

```
using namespace std;

Stack::Stack() : topPtr(NULL)
{
} // end default constructor

Stack::Stack(const Stack& aStack) throw(StackException)
{
    if (aStack.topPtr == NULL)
        topPtr = NULL;   // original list is empty
    else
    {   // copy first node
        topPtr = new StackNode;
        topPtr->item = aStack.topPtr->item;

        // copy rest of list
        StackNode *newPtr = topPtr;         // new list pointer
        for (StackNode *origPtr = aStack.topPtr->next;
            origPtr != NULL;  origPtr = origPtr->next)
        {   newPtr->next = new StackNode;
            newPtr = newPtr->next;
            newPtr->item = origPtr->item;
        } // end for

        newPtr->next = NULL;
    } // end if
} // end copy constructor

Stack::~Stack()
{
    // pop until stack is empty
    while (!isEmpty())
        pop();
    // Assertion: topPtr == NULL
} // end destructor

bool Stack::isEmpty() const
{   return topPtr == NULL;
} // end isEmpty

void Stack::push(const StackItemType& newItem) throw(StackException)
{   // create a new node
    try
    {   StackNode *newPtr = new StackNode;
        // set data portion of new node
        newPtr->item = newItem;
```

```
                // insert the new node
                newPtr->next = topPtr;
                topPtr = newPtr;
        }
        catch (bad_alloc e)
        {       throw StackException(“StackException: push cannot allocates memory.”);
        } // try
} // end push

void Stack::pop() throw(StackException)
{       if (isEmtpy())
                throw StackException(“StackException: stack empty on pop”);
        else
        {       // stack is not empty; delete top
                StackNode *temp = topPtr;
                topPtr = topPtr->next;
                // return deleted node to system
                temp->next = NULL; // safeguard
                delete temp;
        } // end if
} // end pop

void Stack::pop(StackItemType& stackTop) throw(StackException)
{       if (isEmpty())
                throw StackException(“StackExcepton: stack empty on pop”);
        else
        {       // stack is not empty; retrieve and delete top
                stackTop = topPtr->item;
                StackNode *temp = topPTr;
                topPtr = topPtr->next;

                // return deleted node to system
                temp->next = NULL; // safeguard
                delete temp;
        } // end if
} // end pop

void Stack::getTop(StackItemType& stackTop) const throw(StackException)
{       if (isEmpty())
                throw StackExcepton(“StackException: stack empty on getTop”);
        else    // stack is not empty; retrieve top
                stackTop = topPtr->item;
} // end getTop
// end of implementation file
```

2.5 The last implementation would use the ADT list, like so (see figure 6-7, page 305):



The head file is:

```
/** @file StackL.h */

#include "StackExcepton.h"
#include "ListP.h"                  // List operations

typedef ListItemType StackItemType;

/**  ADT stack – ADT list implementation */
class Stack
{
public:
// constructors and destructor:
        Stack();                        // default constructor
        Stack(const Stack& aStack);  // copy constructor
        ~Stack();                       // destructor

// Stack operations:
        bool isEmpty() const;
        void push(const StackItemType& newItem) throw(StackException);
        void pop() throw(StackException);
        void pop(StackItemType& stackTop) throw(StackException);
        void getTop(StackItemType& stackTop) const throw(StackException);

private:
        List aList;       // list of stack items
}; // end stack
// end of header file
```

2.6 The implementation file is:

```
/** @file StackL.cpp */

#include “StackL.h”   // header file

Stack::Stack()
{
} // end default constructor

Stack::Stack(const Stack& aStack) : aList(aStack.aList)
{
} // end copy constructor

Stack::~Stack()
{
} // end destructor

bool Stack::isEmpty() const
{       return aList.isEmpty();
} // end isEmpty

void Stack::push(const StackItemType& newItem) throw(StackException)
{       try
        {       aList.insert(1, newItem);
        }
        catch (ListException e)
        {       throw StackException(“StackException: cannot push item.”);
        }
        catch (ListIndexOutOf RangeException e)
        {       throw StackException(“StackException: cannot push item”);
        } // end try/catch
} // end push

void Stack::pop() throw(StackException)
{       try
        {       aList.remove(1);
        } // end try
        catch (ListIndexOutOfRangeException e)
        {       throw StackException(“StackException: stack empty on pop:”);
        } // end catch
} // end pop

void Stack::pop(StackItemType& stackTop) throw(StackException)
{       try
        {       aList.retrieve(1, stackTop);
                aList.remove(1);
        } // end try
        catch (ListIndexOutOfRangeException e)
        {       throw StackException(“StackException: stack empty on pop”);
        } // end catch
} // end pop

void Stack::getTop(StackItemType& stackTop) const throw(StackException)
{       try
        {       aList.retrieve(1, stackTop);
        } // end try
        catch (LIstIndexOutOfRangeException e)
        {       throw StackException(“StackException: stack empty on getTop”);
        } // end catch
} // end getTop
// end of implementation file
```

3 The above implementations can be either array based or pointer based.

3.1 The main reason to use one over the other is the issue of fixed verses dynamic.

3.2 You can also use the Standard Template Library class of stack.

3.3 If you include the <stack> header file, you can use the STL stack.