

Datastrukturen

2009-10

André Deutz

ADT Stack:

- **createStack()** // creates an empty stack
- **destroyStack()** // destroys a stack
- **stackIsEmpty()** // determines whether the stack is empty
- **push(newItem)** // Adds newItem to a stack.
- **pop()** // Removes from a stack the item that was added most recently.
- **getStackTop(stackTop)** // Retrieves into var *stackTop* the item that was added most recently to a stack, leaving the stack unchanged.

ADT Stack spec in C++

```
class StackClass {  
public:  
    StackClass();  
    StackClass(const StackClass & S); //copy constructor  
    ~StackClass();
```

```
// stack operations
```

```
    bool stackIsEmpty();
```

```
    // determines whether the stack is empty.
```

```
    // precondition: the constructor has been called
```

```
    // postcondition: Returns TRUE if the stack was empty, otherwise returns FALSE
```

```
    void push(stackItemType newItem);
```

```
    // adds an item to the top of the stack
```

```
    // precondition: the constructor has been called. newItem is the item to be
```

```
    // added.
```

```
    // postcondition: if insertion was successful, newItem is on top of the
```

```
    // stack.
```

```
    void pop();
```

```
    // Removes the top of stack.
```

```
    // precondition: the constructor has been called.
```

```
    // postcondition: if the stack was not empty, the item that was added
```

```
    // MOST RECENTLY is removed.
```

```
    void getStackTop(stackItemType & topItem);
```

```
    // Retrieves the top of the stack.
```

```
    // If the stack was not empty, topItem contains the item
```

```
    // that was added MOST RECENTLY.
```

```
private:
```

```
// belongs to implementation!
```

```
};
```

Resembles too much C programming, since *typedef* is used or needs to be used.

Use templates (generic programming) instead

Can use the ADT stack without knowing the implementation of the ops

S.createStack() ; //in C++ declare S as instance of the stack class, since //createStack() is implemented as the class' constructor

Read *newChar*;

While (*newChar* not eoln){

if (*newChar* is not '*'){

S.push(*newChar*);

} else {

if (! S.stackIsEmpty){

S.pop();

}

}

read *newChar*

} //end while

Program Read input line and
correct along
The way

Can use the ADT stack without knowing the implementation of the operations:

```
tempS.createStack();  
While (!S.isEmptyStack()) {  
    S.getTopStack(ch);  
    S.pop();  
    tempS.push(ch);  
}
```

```
// top of the stack  tempS  holds the first char of the line  
// entered !!
```

Program Read input line and
correct along
The way continued

```

class StackClass {
public:
    StackClass();
    StackClass(const StackClass & S); //copy constructor
    ~StackClass();
// stack operations
    bool stackIsEmpty();
    // determines whether the stack is empty.
    // precondition: the constructor has been called
    // postcondition: Returns TRUE if the stack was empty, otherwise returns FALSE
    void push(stackItemType newItem);
    // adds an item to the top of the stack
    // precondition: the constructor has been called. newItem is the item to be
    // added.
    // postcondition: if insertion was successful, newItem is on top of the
    // stack.
    void pop();
    // Removes the top of stack.
    // precondition: the constructor has been called.
    // postcondition: if the stack was not empty, the item that was added
    // MOST RECENTLY is removed.
    void getStackTop(stackItemType & toplItem);
    // Retrieves the top of the stack.
    // If the stack was not empty, toplItem contains the item
    // that was added MOST RECENTLY.
private:
    // belongs to implementation!
};

```

ADT Stack spec in C++

```
// header file; array based implementation
```

```
#ifndef _ARRAYIMPLSTACK_ // or use the directive #pragma once
```

```
#define _ARRAYIMPLSTACK_
```

```
#include <iostream>
```

```
using namespace std;
```

```
const int MAX_STACK = 100;
```

```
typedef char stackItemType;
```

```
class StackClass {
```

```
public:
```

```
    StackClass();
```

```
    StackClass(const StackClass & S); //copy constructor
```

```
    ~StackClass();
```

```
// stack operations
```

```
    bool stackIsEmpty();
```

```
    void push(stackItemType newItem);
```

```
    void pop();
```

```
    void getStackTop(stackItemType & topItem);
```

```
private:
```

```
    stackItemType items[MAX_STACK]; // information hiding, only accessible through public interface/contract
```

```
    int top; // information hiding
```

```
};
```

```
#endif
```

Resembles too much C programming, since *typedef* is used.
Use templates (generic programming) instead

```

// implementation file arrayImplStack.cpp for the ADT Stack; array-based
//implementation
#include "arrayImplStack.h"
StackClass::StackClass(): top(-1) {}
StackClass::StackClass(const StackClass & S): top(S.top) {
    for(int i=0; i<=S.top; i++){
        items[i] = S.items[i];
    }
}
StackClass::~StackClass() {}
bool StackClass::stackIsEmpty() {return bool (top<0);}
void StackClass::push(stackItemType newItem) {
    if (top<(MAX_STACK-1)){
        ++top;
        items[top]=newItem;
    }
}
void StackClass::pop() {
    if (!stackIsEmpty()) {
        --top;
    }
}
void StackClass::getStackTop(stackItemType& topItem){
    if (!stackIsEmpty()){
        topItem=items[top];
    }
}

```



```

// a client program that uses stack(s)
// (i.e., an algorithm
// that uses a stack, actually two
//stacks)
#include "arrayImplStack.h"
#include <iostream>
using namespace std;
int main (){
    stackItemType anItem;
    StackClass S;
    cin.get(anItem);
    while (anItem!='\n'){
        if (anItem!='*'){
            S.push(anItem);
        }else{
            if (!S.stackIsEmpty()){
                S.pop();
            }
            cin.get(anItem);
            cout << "\n"<< "\n";
            StackClass tempS;
            while(!S.stackIsEmpty()){
                S.getStackTop(anItem);
                S.pop();
                tempS.push(anItem);
            }
            // top of the stack tempS now contains
            //the first char of the entered line
            // you can print, for instance to the
            //screen
            while (!tempS.stackIsEmpty()){
                tempS.getStackTop(anItem);
                cout << anItem;
                tempS.pop();
            }
            cout << "\n";
            cout << "\n";
            return 1;
        }
    }
}

```

Client program of Stack,
i.e. Algorithm which uses Stack

ADT Stack Spec in C++, using templates

spec

```
#ifndef _ARRAYIMPLSTACK2 // or you can use: #pragma once
#define _ARRAYIMPLSTACK2
#include <iostream>
using namespace std;

const int MAX_STACK = 100;
//typedef char stackItemType; // C way of doing things: type checking is not enabled
```

Does not belong to spec

```
template<class T> //now type checking is enabled
class StackClass {
public:
    StackClass();
    StackClass(const StackClass<T> & S); //copy constructor
    ~StackClass();
    // stack operations
    bool stackIsEmpty();
    // determines whether the stack is empty.
    // precondition: the constructor has been called
    // postcondition: Returns TRUE if the stack was empty, otherwise returns FALSE

    void push(T newItem);
    // adds an item to the top of the stack
    // precondition: the constructor has been called, newItem is the item to be
    // added.
    // postcondition: if insertion was successful, newItem is on top of the
    // stack.

    void pop();
    // Removes the top of stack.
    // precondition: the constructor has been called.
    // postcondition: if the stack was not empty, the item that was added
    // MOST RECENTLY is removed.

    void getStackTop(T & topItem);
    // Retrieves the top of the stack.
    // If the stack was not empty, topItem contains the item
    // that was added MOST RECENTLY.
```

```
private:
    T items[MAX_STACK]; //implementation part; solely accessible through public interface
    // information hiding
    int top; //implementation
};

#endif
```

Better yet: T & getStackTop(); // see Chapter 4 in
Drozdek

Implementing ADT Stack in C++, template version

```
//Implementation file arrayImplStack.cpp: Array-based implementation
#include "arrayImplStack.h
#include<iostream>
using namespace std;
// implementation
template<class T>
StackClass<T>::StackClass(): top(-1) {}

template<class T>
StackClass<T>::StackClass(const StackClass<T> & S):top(S.top){
    top = S.top;
    for (int i=0; i<= S.top; ++i){
        items[i] = S.items[i];
    }
}

template<class T>
StackClass<T>::~~StackClass() {}

template<class T>
bool StackClass<T>::stackIsEmpty() {return bool (top<0);}

template<class T>
void StackClass<T>::push(T newItem) {
    if (top<(MAX_STACK-1)){
        ++top;
        items[top]=newItem;
    }
}

template<class T>
void StackClass<T>::pop() {
    if (!stackIsEmpty()) {
        --top;
    }
}

template<class T>
void StackClass<T>::getStackTop(T& topItem){
    if (!stackIsEmpty()){
        topItem=items[top];
    }
}
```

Client program of Stack, i.e. Algorithm which uses Stack

```
#include "arrayImplStack.h"
#include <iostream>
using namespace std;

int main (){

    char anItem;
    StackClass<char> S;

    cin.get(anItem);

    while (anItem!='\n'){
        if (anItem!='*'){
            S.push(anItem);
        }else{
            if (!S.stackIsEmpty()){
                S.pop();
            }
        }
        cin.get(anItem);
    }

    cout << "\n";
    StackClass<char> tempS;

    while(!S.stackIsEmpty()){
        S.getStackTop(anItem);
        S.pop();
        tempS.push(anItem);
    }

    // top of the stack tempS now contains the first char of the entered line
    // you can print, for instance to the screen
    while (!tempS.stackIsEmpty()){
        tempS.getStackTop(anItem);
        cout << anItem;
        tempS.pop();
    }
    cout << "\n";
    cout << "\n";
    return 1;
}
```

The spots to watch out for when changing implementations. Normally speaking the client program should not have to be changed. In this case the ADT designer changed the contract slightly in the typing by going from C-style to C++ style (templates). This required a change in the client code.

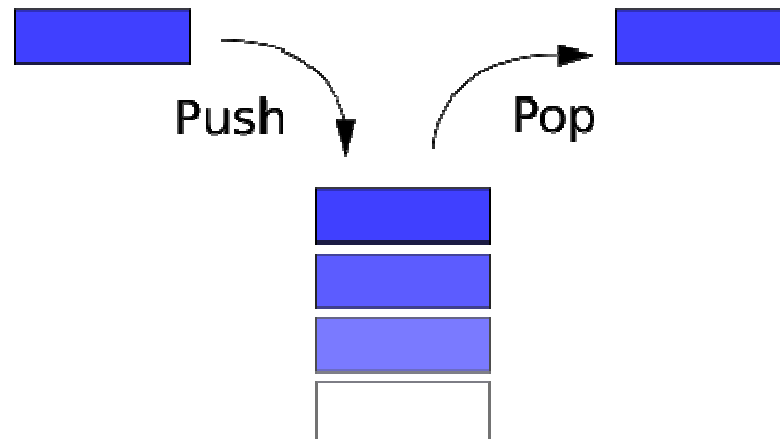
Discussion of Uses of Stacks

edgser dijkstra: invented stacks to implement recursively defined functions/programs

CA: stack frames used in function calls

visiting nodes in a tree

Ubiquitous! (see also Chapter 4 for more apps)



The end ☺. This means read and study Chapter 4 in Drozdek.

Start of Lecture of Sep 7

Data Structures

- Recap
 - Course Goals
 - ADTs in solution development (= from problem to ADT).
 - In our example a ADT Stack emerged and an ADT which is almost equal to the standard ADT Deque (double ended queue).
- In class : the ideas, in the werkgroep: high level practice; in the book most of the time you can find the details, sometimes gory.

Data Structures

- What are we trying to learn, explained on a very high level?
Answer: managing complexity. Slightly less high level:
producing reliable, correct, and efficient programs;

ADTs revisited

- More precise *informal* specification: pre- and postconditions (see example of previous lecture, written out in C++)
- Definition: programmed as an interface or can be mathematical
- Math spec \rightarrow axiomatic semantics

ADTs revisited

type T

uses types Boolean, Int

operators

Create $\rightarrow T$

P1 : $T * \text{Int} \rightarrow T$

P2 : $T \rightarrow T$

B : $T \rightarrow \text{Int}$

E : $T \rightarrow \text{Boolean}$

axioms

E(Create) = true

E(P1(t,i)) = false

P2(Create) = Create *

P2(P1(t,i)) = t

B(Create) = 0

B(P1(t,i)) = i *

end T

ADTs revisited

type Intstack

uses types Boolean, Int

operators

Create \rightarrow Intstack

Push : Intstack * Int \rightarrow Intstack

Pop : Intstack \rightarrow Intstack

Top : Intstack \rightarrow Int

Isempty : Intstack \rightarrow Boolean

axioms

Isempty(Create) = true

Isempty(Push(s,i)) = false

Pop(Create) = Create *

Pop(Push(s,i)) = s

Top(Create) = 0

Top(Push(s,i)) = i *

end Intstack

ADTs Revisited

- ADT is not a fancy name for a data structure: ultimately you implement the ADT using a data structure (= a structured data type) and data types
- Data type defines a set of values and allowable operations on those values.
- In most programming languages additional data types can be defined, usually by combining multiple elements of other types and defining valid operations on them .
- For example, a programmer might create a new data type named "Person" that specifies that data interpreted as Person would include a name and a date of birth.
- Client programs of an ADT are concerned with the interface, not the implementation as implementation can change in the future (Supports information hiding (or protecting client programs from design decision that are subject to change)
- Strength of ADT is that implementation is hidden from the user. Only interface is published – thus can implement ADT in various ways as long as you adhere to the interface/contract, client programs are unaffected.

ADTs Revisited

- There is distinction, sometimes subtle, between the abstract data type and the data structure used in its implementation.
- Example: ADT List can be represented as array-based implementation or linked-list implementation.
- A List is an ADT with well defined operations (add element, remove element , etc) while a linked-list is a pointer based data structure that can be used to create a representation of a List.
- Linked-list implementation is commonly used to represent ADT List and by abuse of language the terms are commonly interchanged.

ADT through the Glasses of a Modern Language (C++)

- Client uses class as abstraction
 - Invokes public operations only
 - Internal implementation not relevant!
- Client can't and shouldn't muck with internals
 - Class data should be private (supported by modern languages)
- Imagine a "wall" between client and implementer
 - Wall prevents either from getting involved in other's business
 - Interface is the "chink" in the wall
 - Conduit allows controlled access between the two
- Consider Lexicon
 - Abstraction is a word list, operations to verify word/prefix
 - How does it store list? using array? vector? set? does it matter to client?

Why ADTs?

- Abstraction
 - Client insulated from details, works at higher-level
- Encapsulation
 - (In modern language can make) internals private to ADT, not accessible by client (accidentally or on purpose)
- Independence
 - Separate tasks for each side (once agreed on interface)
- Flexibility
 - ADT implementation can be changed without affecting client (provided implementation adheres to the contract: that is, syntax and semantics of functions does not change)
- Once and Only Once
- Strongly supports Modularity

Interplay Algo -- ADT

- Example of interplay between (main) algorithm and ADT: how much intelligence do you put into the ADT and how much of it into the main Algorithm?
- Possibly without knowing it, last time you came up with a solution where more of the intelligence was put into the ADT

Recall problem of last time: How can you implement the “undo” for the following situation.

When you type a line of text at a keyboard, you are likely to make mistakes. We assume that you can use the usual ascii characters to enter lines of text except the asterisk ‘*’. With this character you can announce the wish for the undo in case you made a typo. The understanding is that you use the asterisk key to correct these mistakes, each asterisk erases the previous character entered. Consecutive asterisks are applied in sequence and so erase several characters.

For instance, if you type the line

abcc*ddde***ef*fg

the corrected input would be

abcdefg

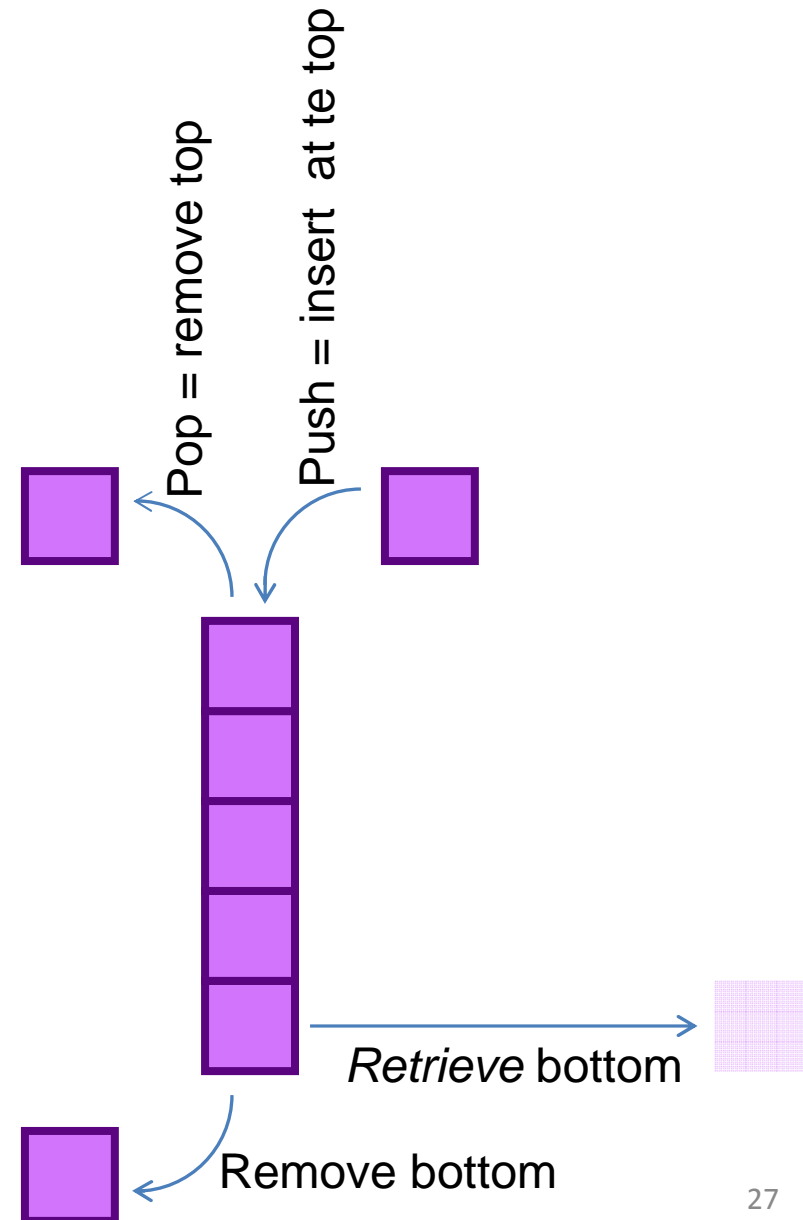
How can a program read the original line and get the corrected input AND DISPLAY (or PROCESS) IT IN THE ENTERED ORDER? *The algorithm you came up with uses the following ADT (see next slide) – the ADT was also your proposal*

Hybridization/Interlacing of Stack and Queue:

ADT HybridStackQueue (HSQ)
(yours ☺; it is almost
the ADT **Deque**)

NB pop, remove: loses
info forever;
Each of the ops
pop, push, and remove
changes the state of the data
structure

NB retrieve does not
change the state of
the data structure
= get the info



Spec of ADT HSQ

Create() // Creates an empty hsq
Destroy() // Destroys the hsq
isEmpty() //Determines whether the hsq is empty
insertAtTheTop(newItem) // Add a newItem
removeTop() // Removes from the hsq the MOST RECENTLY added item to the hsq
retrieveBottom(item) // Retrieves into var *item* the item that was added LEAST RECENTLY to a hsq, leaving the hsq unchanged.
removeBottom() // Removes the item that was added LEAST RECENTLY to a hsq.

Not needed: **retrieveTop(topItem)** //Retrieves into var *topItem* the item that was added MOST RECENTLY to a hsq, leaving the hsq unchanged

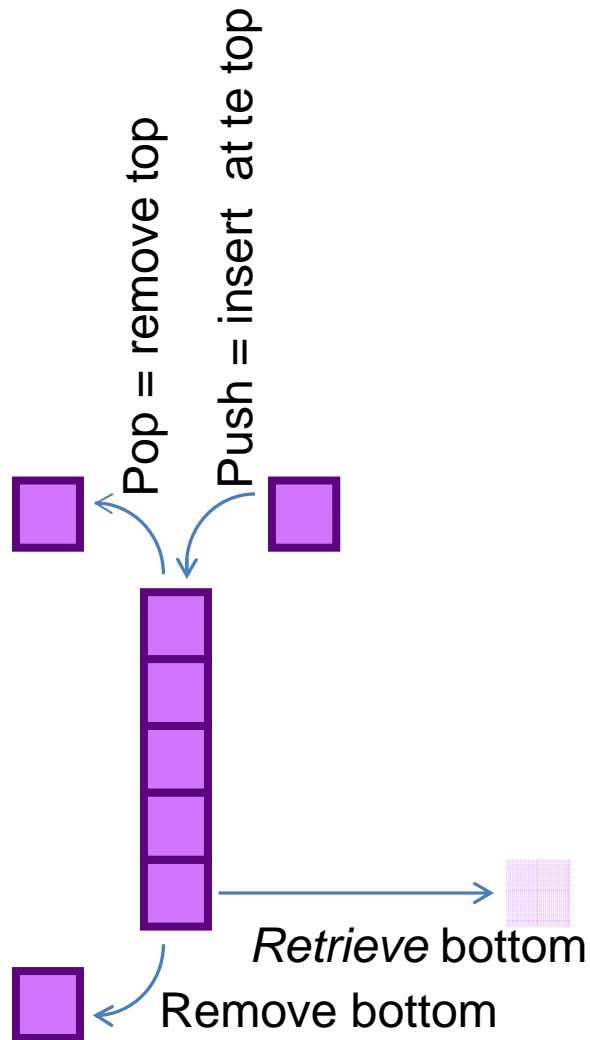
Given the ADT HSQ, how can we specify the “undo” algorithm/program? Again we don’t need to know the implementation of the ADT HSQ.

```
H.create()
Read newChar;
While (newChar not eoln) {
    if (newChar is not '*') {
        H.insertAtTheTop (newChar);
    } else {
        if (! H.isEmpty() ) {
            H.removeTop();
        }
    }
}
While (! H.isEmpty()){
    H.retrieveBottom(newChar);
    H.removeBottom();
    // process newChar e.g., display newChar
    // items will be process according to entry-
order
}
```

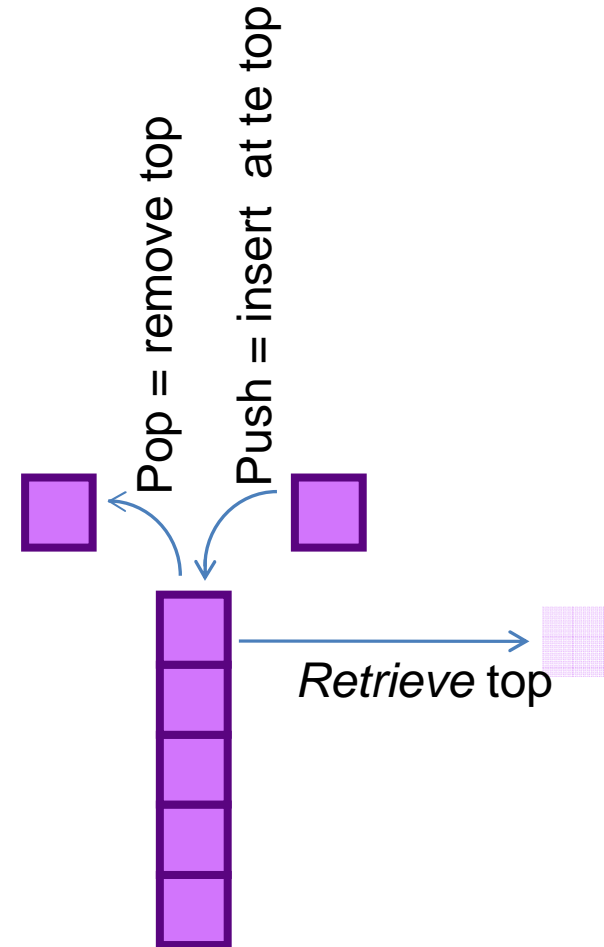
Recall Ops of HSQ:

```
Create()
Destroy()
isEmpty()
insertAtTheTop(newItem)
removeTop()
retrieveBottom(item)
removeBottom()
```

Which of the two ADTs will we choose for the “undo” algorithm?



ADT HSQ



ADT Stack

The interplay between ADT and the (main) algorithm using it

ADT HSQ

- The design of the “undo and entered-order processing” algorithm becomes easier (we put more *smartness* in the ADT): it is much easier to design the algorithm
- More work to come up with an implementation
- Array implementation either very inefficient (time) or subtle implementation (such as circular arrays – error prone)
- What about the efficiency of the various implementations
- Can already start processing once the first item is in!

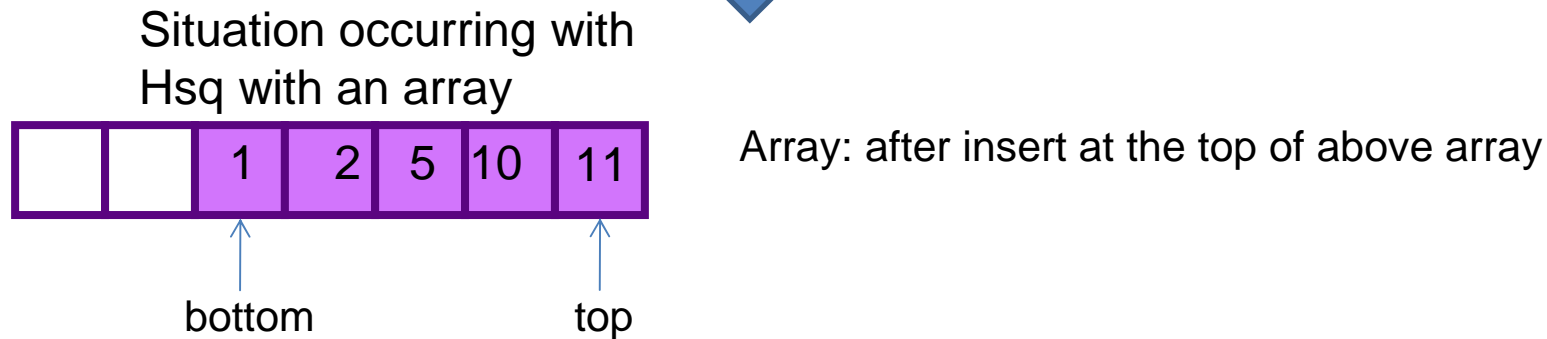
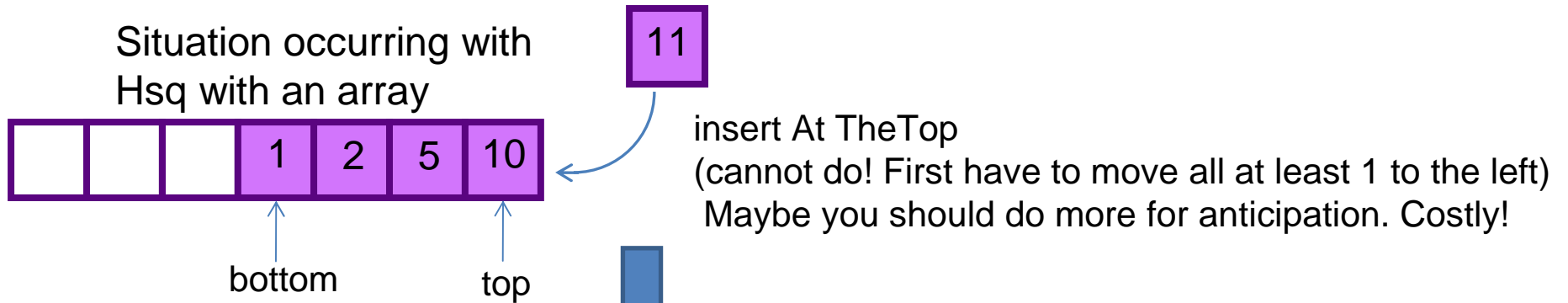
ADT Stack

- By the same token: Need to put more *smartness* into the design of the “undo and entered-order processing” algorithm.
- Less work to design the implementation
- Array implementation is done in a jiffy – important in the chicken and the egg problem (testing one module against another)
- What about the efficiency of the various implementations?
 - We could look at the efficiency of the ADT operations
 - Could also look at the total efficiency of the Algorithm

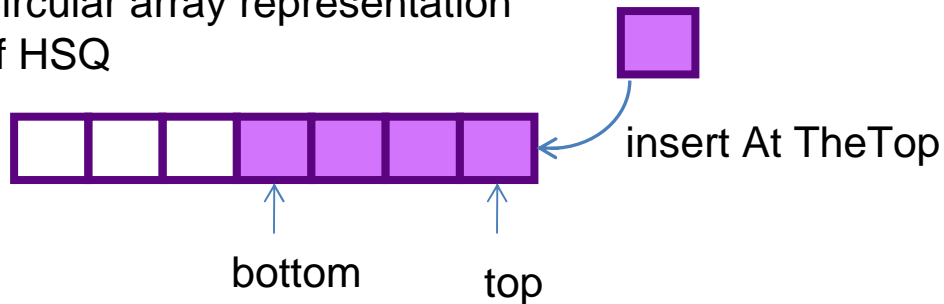
HSQ vs Stack

- Suppose we decide to implement HSQ with the array data structure. A straightforward implementation will invariably be not efficient. While implementing a Stack with an array is efficient. Both have the same problem can grow beyond the array size. What about using the STL vector?
- Circular array. Will be efficient for HSQ – more difficult to implement. Each operation is $O(1)$. Not needed though for Stacks (or stronger: don't use it for Stacks). See next slide.

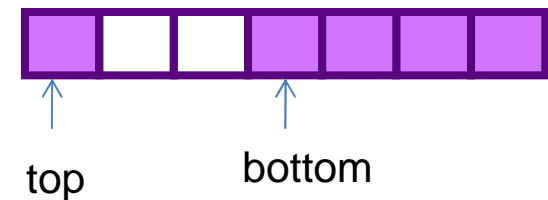
HSQ implementations



Circular array representation of HSQ

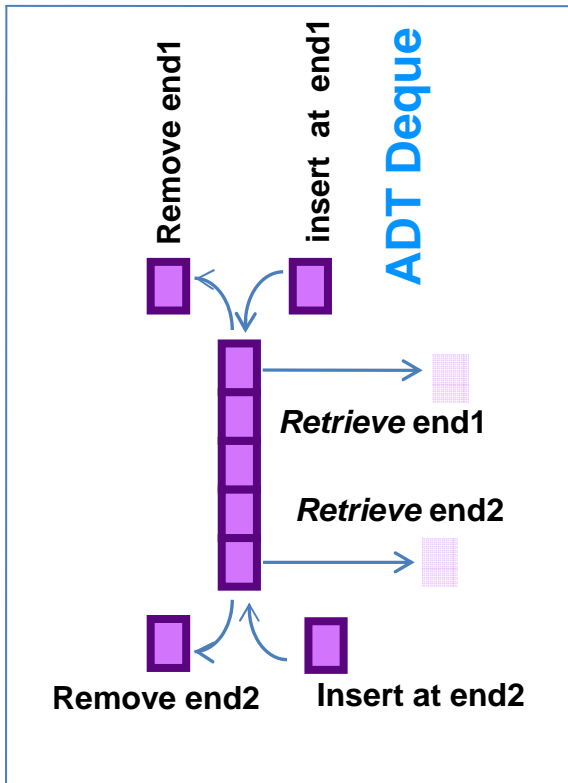


Circular array representation of HSQ: after insertAtTheTop

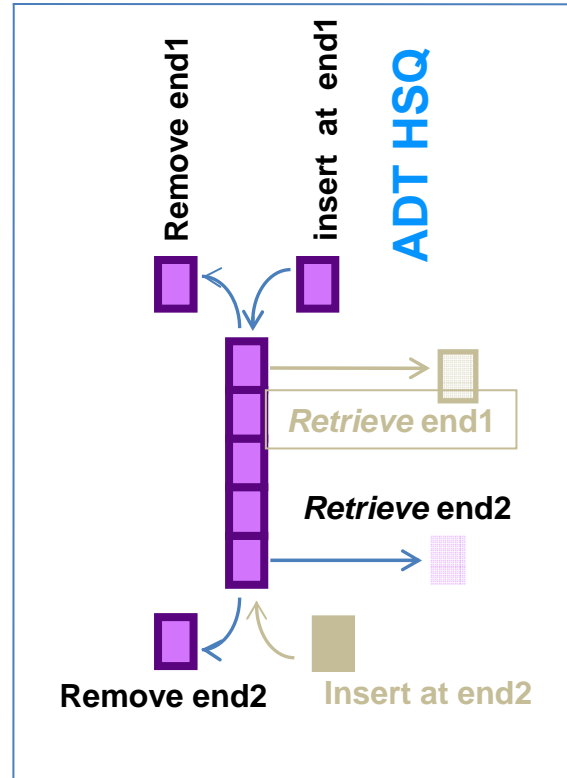


“Wrap around” by mod

HSQ implementations



Given an implementation of the ADT **Deque**, it is a freebie to provide an implementation for the ADT HSQ (just forget the operations Retrieve end1 and Insert at end2).



HSQ implementations

- In the next slides we define ADT List
- Given ADT List we can implement HSQ by using an implementation of ADT List

Lists

- Lists contain items of the same type
 - List of grocery items
 - List of phone numbers
- What can you do to the items of a list?
 - Determine length of the list
 - Add an item to the list
 - Remove an item from the list
 - Retrieve an item from the list
- Where do you want to add a new item and which item do you want to look at?
 - Various answers → various ADTs List

ADT List

createList()

// Creates an empty list.

destroyList()

// Deatroys a list.

listIsEmpty()

// Determines whether the list is empty.

listLength()

// Returns the number of items in list.

listInsert(newPosition, newItem, success)

// Inserts newItem at position newPosition of a list, if $1 \leq \text{newPosition} \leq \text{listLength()} + 1$

// if $\text{newPosition} \leq \text{listLength}()$, the items are shifted as follows

// the item at newPosition becomes the item at newPosition + 1, the item

// at newPosition + 2, and so on. Success indicates whether the insertion was successful.

listDelete(pos, success)

// Deletes the item at position pos of a list, if $1 \leq \text{pos} \leq \text{listLength}()$. If $\text{pos} < \text{lengthList}()$, the items are


// shifted as follows: the item at pos+1 becomes the at pos, the item at pos+1 becomes the item at pos+1, and so // on. Success indicates whether the deletion was successful.

listRetrieve(pos, dataItem, success)

// sets dataItem to the item at position pos of a list, if $1 \leq \text{pos} \leq \text{listLength}()$. The list is left unchanged by

// this operation. Success indicates whether the retrieval was successful.

This variant is *not*
Discussed in Drozdek



NB there is also ADT Ordered list

**Given the ADT List, how do you
implement the ADT HSQ?**

ADT List-Drozdek

- (We are going to study Chapter 3 of Drozdek via the notion of ADTs. NB this is not the way Drozdek has structured Chapter 3 of his book. More about this later on.)

The operations of this ADT are as follows:

createList()

// Creates an empty list. In C++ this is implemented as a constructor (provided we implement the ADT as a class)

destroyList()

// Destroys a list. In C++ this is implemented as the destructor (provided we implement the ADT as a class).

isEmpty()

// Determines whether the list is empty. The type of this operation is: Lists \rightarrow Bool

addToHead(int)

// The type of this operation is: Lists \times int \rightarrow Lists

addToTail(int)

// The type of this operation is: Lists \times int \rightarrow Lists

deleteFromHead()

// delete the head and return its info; thus the type of the operation is: Lists \rightarrow int

deleteFromTail()

// delete the tail and return its info; thus the type of the operation is: Lists \rightarrow int

isInList(int)

// checks whether the integer occurs in the list; the type of the operation is: Lists \rightarrow bool

**Given the ADT List-Drozdek, how do
implement the ADT HSQ?**

Chapter 3 of Drozdek a la ADT

- ADT List, ADT ListDrozdek, ADT ListDrozdek+wholsNext, ADT Ordered List, and ADT List Insertion Influenced Implicitly by Search
- Compare these ADTs
- **We are going to study Chapter 3 through the glasses of ADTs:**
- The different implementations ADT List Drozdek
 - Array Implementation
 - Singly Linked List Implementation
 - Doubly Linked List Implementation
 - Circular Singly Linked List Implementation
 - Circular Doubly Linked List Implementation
- The different Implementations of ADT List Drozdek +wholsNext
 - Circular Singly Linked List Implementation
 - Circular Doubly Linked List Implementation
- Implementation of ADT Ordered (Sorted) List
 - Skip List implementation
- “ADT List With Insertion Influenced Implicitly by Search”
 - Implementation^s with Self-Organization: move-to-front, transpose, count-method (excluding ordered implementation)

Chapter 3 of Drozdek a la ADT

- See previous slides for the specification of ADT List and ADT List Drozdek
- Here is how the spec for ADT List Drozdek appears in book on page 79 written in C++:

FIGURE 3.2 singly-linked list class to store integers

```

//*****  intSLLst.h  *****
//          singly-linked list class to store integers

#ifndef INT_LINKED_LIST
#define INT_LINKED_LIST

class IntSLLNode {
public:
    int info;
    IntSLLNode *next;
    IntSLLNode(int el, IntSLLNode *ptr = 0) {
        info = el; next = ptr;
    }
};

class IntSLLList {
public:
    IntSLLList() {
        head = tail = 0;
    }
    ~IntSLLList();
    int isEmpty() {
        return head == 0;
    }
    void addToHead(int);
    void addToTail(int);
    int deleteFromHead(); // delete the head and return its info;
    int deleteFromTail(); // delete the tail and return its info;
    void deleteNode(int);
    bool isInList(int) const;
};

```

Continues

(An Aside)

Recall: Self-referential Classes

```
#include <iostream>
using namespace std;

struct node {
    int height;
    node * next; // used before completely defined! the only place in C++ where this is allowed.
}; // so called self-referential structs (or classes)

/***** alternatively with classes *****/
class node {
public:
    int height;
    node * next;
}

int main () {
    node * head = new node();
    // froebeling with pointers:
    head -> height = 10;
    head -> next = new node();
    head -> next -> height = 11;
    head -> next -> next = NULL;
    return 1;
}
```

Self-referential class objects can be linked together to form useful Data structures (such as lists, queues, stacks or trees.

Compare Two ADTs

ADT ListDrozdek

ADT List; is not discussed
in the book

```
#pragma once

Class List {
public:
    List();
    ~List();
    int listLength() const;
    bool listIsEmpty() const;
    void listInsert(int, int, bool);
    void listDelete(int, bool);
    void listRetrieve(int, int &, bool) const;
//private:
    /*****
createList()
    Creates an empty list; taken care off by constructor
destroyList()
    Destroys a list; taken care of by destructor
listIsEmpty()
    Determines whether the list is empty.
listLength()
    Returns the number of items in list.
listInsert(newPosition, newItem, success)
    Inserts newItem at position newPosition of a list, if 1 = newPosition = listLength() + 1
    if newPosition = listLength(), the items are shifted as follows
    the item at newPosition becomes the item at newPosition + 1, the item
    at newPosition + 2, and so on. Success indicates whether the insertion was successful.
listDelete(pos, success)
    Deletes the item at position pos of a list, if 1 = pos = listLength(). If pos < lengthList(), the it
    shifted as follows: the item at pos+1 becomes the at pos, the item at pos+1 becomes the item at pos+1, and
    so on. Success indicates whether the deletion was successful.
listRetrieve(pos, dataItem, success)
    sets dataItem to the item at position pos of a list, if 1 = position = listLength(). The list is left unchange
    this operation. Success indicates whether the retrieval was successful.
*****/
};
```

```
//page 79 in Drozdek
//
#pragma once

class IntSLLNode {
public:
    int info;
    IntSLLNode * next;
    IntSLLNode (int el, IntSLLNode * ptr = 0) {
        info = el; next=ptr;
    }
};
```

// the following we can view as the programmed ADT ListDrozdek

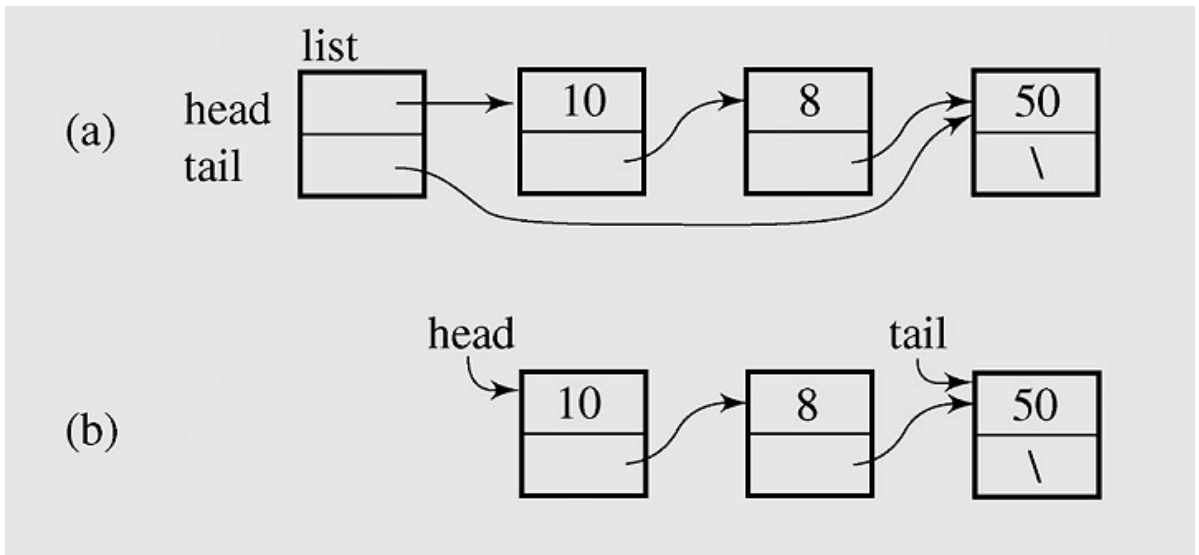
```
class IntSLLList {
public:
    IntSLLList(); //{head =0; tail=0}
    ~IntSLLList();
    bool isEmpty();
    void addToHead(int);
    void addToTail(int);
    int deleteFromHead(); // delete the head and return its info
    int deleteFromTail(); // delete the tail and return its info
    void deleteNode(int);
    bool isInList(int) const;
private:
    IntSLLNode * head, * tail;
};
```

ADT List Drozdek: array implementation

- Analyze each of the operations and determine the cost of each
 - Insert at the head
 - Delete at the head
 - Insert at the tail
 - Delete at the tail
 - Delete int
 - Determine whether int is in the list.

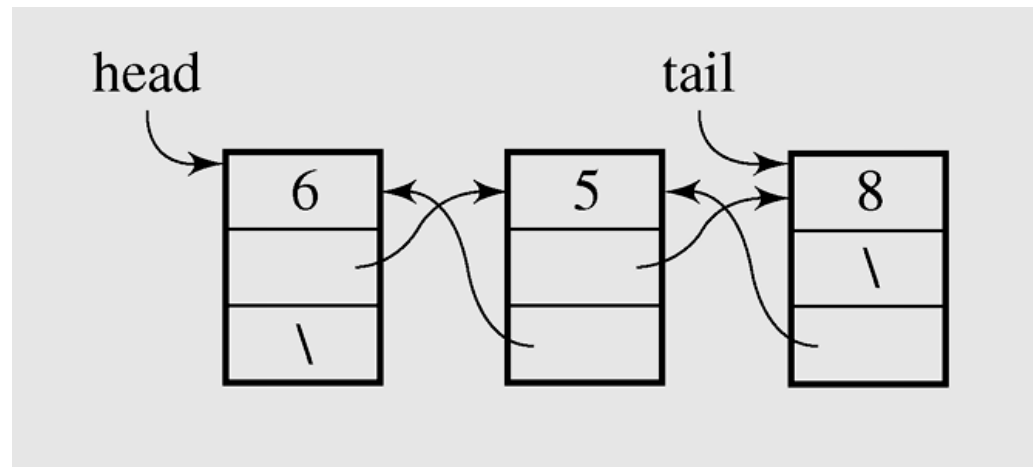
ADT List Drozdek: singly linked list implementation

- Analyze each of the operations and determine the cost of each
 - Insert at the head
 - Delete at the head
 - Insert at the tail
 - Delete at the tail
 - Delete int
 - Determine whether int is in the list.



ADT List Drozdek: doubly linked list implementation

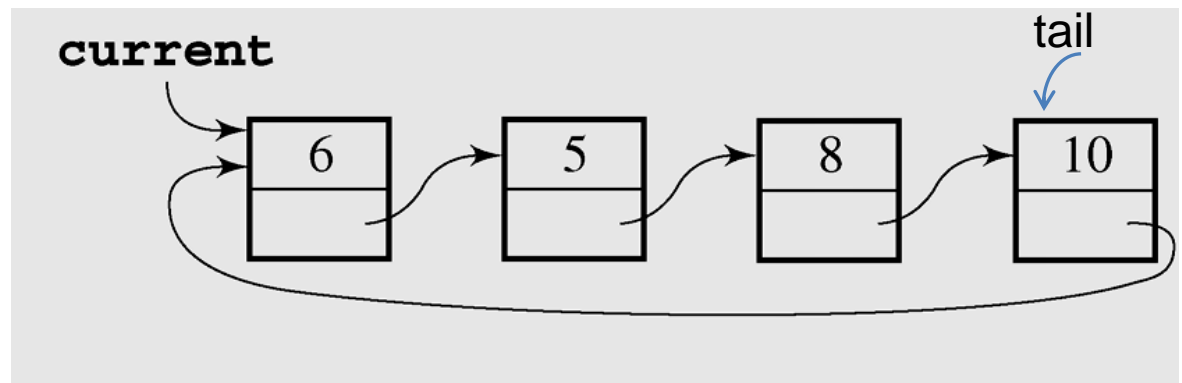
- Analyze each of the operations and determine the cost of each
 - Insert at the head
 - Delete at the head
 - Insert at the tail
 - Delete at the tail
 - Delete int
 - Determine whether int is in the list.



ADT List Drozdek: circular singly linked list implementation

- Analyze each of the operations and determine the cost of each

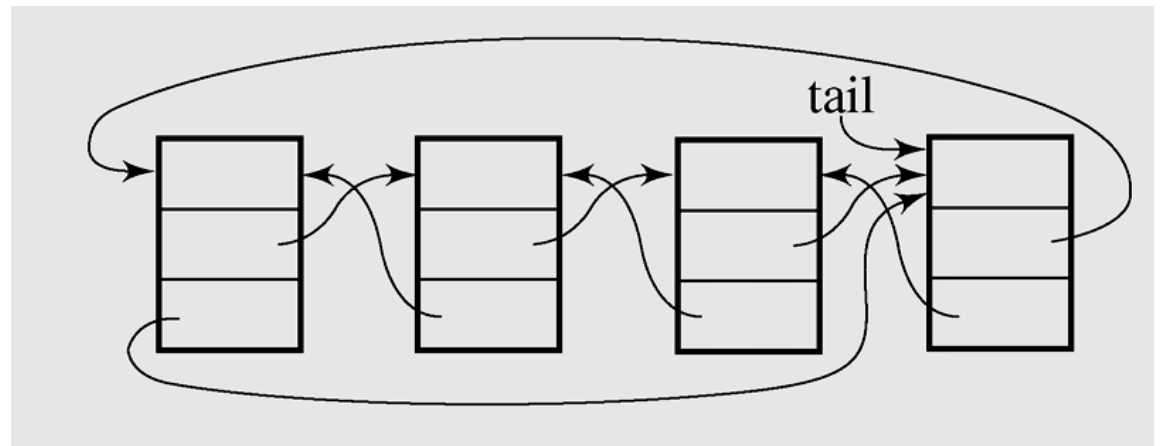
- Insert at the head
- Delete at the head
- Insert at the tail
- Delete at the tail
- Delete int
- Determine whether int is in the list.



- **NB In fact this implementation does more than is needed by the ADT List Drozdek: it implements an extra operation `wholsNext (=current)`. Hence, it satisfies another ADT, namely, ADT List Drozdek + the op `wholsNext`**

ADT List Drozdek: circular doubly linked list implementation

- Analyze each of the operations and determine the cost of each
 - Insert at the head
 - Delete at the head
 - Insert at the tail
 - Delete at the tail
 - Delete int
 - Determine whether int is in the list
- **NB In fact this implementation does more than is needed by the ADT List Drozdek: it easy to implement an extra operation wholsNext (by the intro of an extra pointer **current**). Hence, it satisfies another ADT, namely, ADT List Drozdek + the op wholsNext**



ADT List Drozdek + wholsNext operation

This ADT has all the operations of ADT List Drozdek plus an extra operation **wholsNext**.

As already said on the previous slides: Chapter 3 of Drozdek describes two implementations of this ADT:

- 1) A circular singly linked list
- 2) A circular doubly linked list

ADT Ordered (Sorted) List

Operations:

createSortedList // creates empty sorted list

destroySortedList // destroys sorted list

sortedListIsEmpty // determines whether sorted list is empty

sortedListLength //returns the number of items in the a sorted list

sortedListInsert(newItem, success) // inserts newItem into its
//proper sorted position in a sorted list.

sortedListDelete(item, success) // deletes item from a sorted list

sortedListRetrieve(*position*, dataItem, success) // sets dataItem
to the item position *position* of a sorted list, if $1 \leq \textit{position} \leq$
sortedListLength. The list is left unchanged by this operation

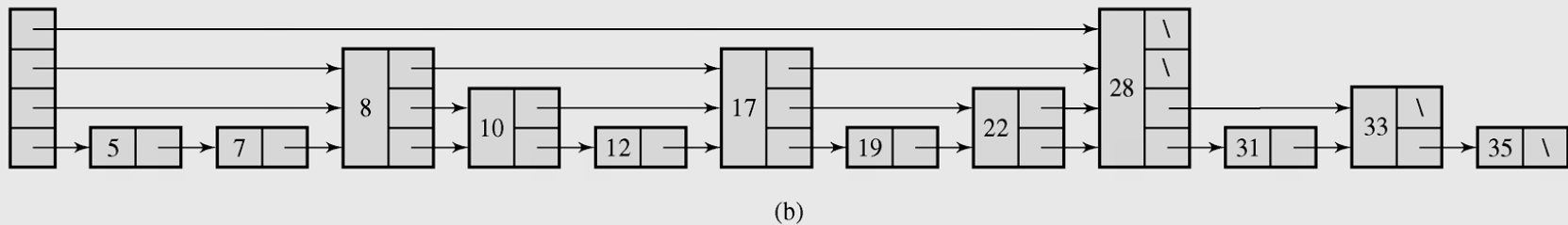
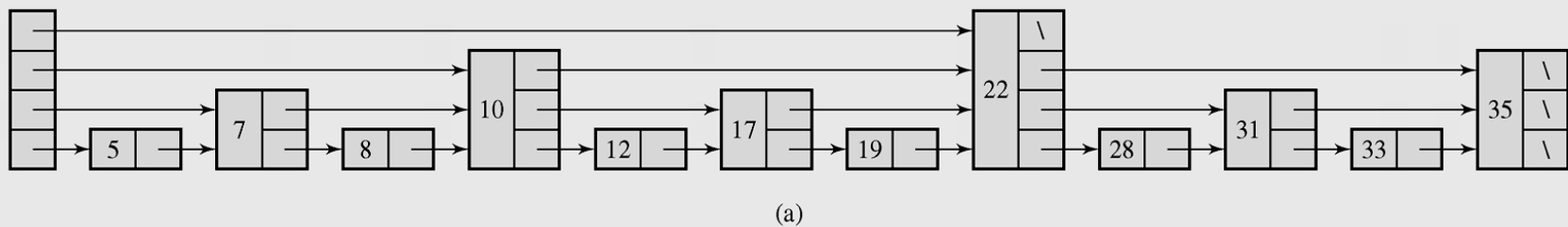
locatePosition (anItem, *position*, success) //sets position to the
position where anItem belongs or exists in a sorted list. Success indicates
whether anItem is currently in the list. AnItem and the list are unchanged.

(A variation on this ADT is to discard the last two operations and introduce an
operation `isInTheList(item)`; this variation is used by Drozdek implicitly in
the discussion on skip list and ordered self-organization.)

ADT Ordered (Sorted) List: skip list implementation

```
1  
1----->4----->6  
1----->3->4----->6----->9  
1->2->3->4->5->6->7->8->9->10
```

ADT Ordered List: skip list implementation



ADT Ordered List: skip list implementation

- Of course: there are other implementations of the ADT Ordered List.
- Remark: in the discussion of self-organization (see next slides) Drozdek uses one of the implementations of the ADT Ordered List; which implementation is used is not made explicit.

ADT List with Insertion Influenced by Search: self-organizing list implementation

There are four methods for organizing lists:

- **Move-to-front method** – after the desired element is located, put it at the beginning of the list (Figure 3.18a)
- **Transpose method** – after the desired element is located, swap it with its predecessor unless it is at the head of the list (Figure 3.18b)
- **Count method** – order the list by the number of times elements are being accessed (Figure 3.18c)

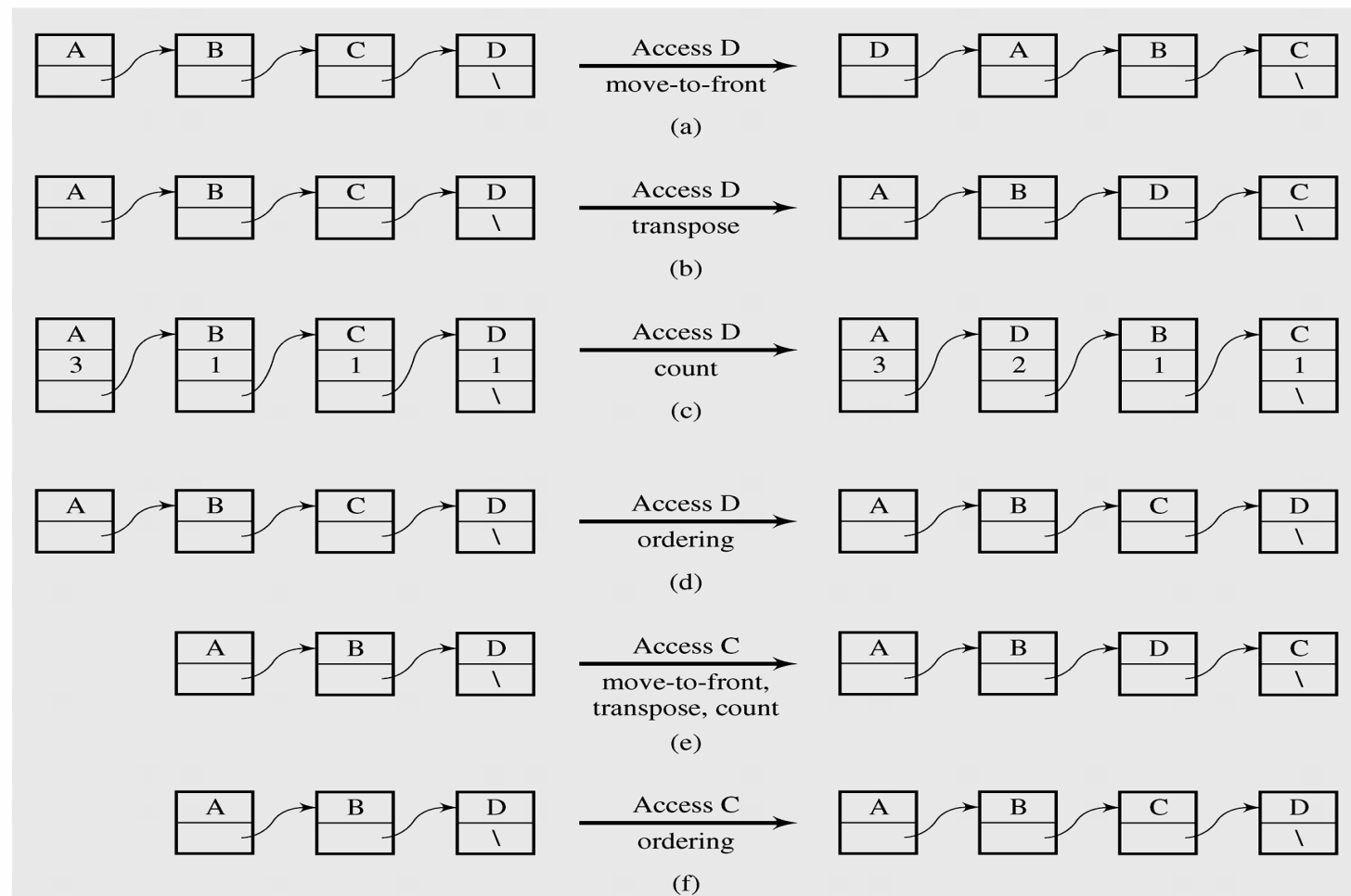
ADT List with Insertion Influenced by Search: self-organizing list implementation

There are four methods for organizing lists:

- **Ordering method** – order the list using certain criteria natural for the information under scrutiny (Figure 3.18d) – **is in fact an implementation of ADT Ordered List!**
- **Optimal static ordering** all the data are already ordered by the frequency of their occurrence in the body of data so that the list is used only for searching, not for inserting new items

ADT List with Insertion Influenced by Search: self-organizing list implementation

FIGURE 3.18 Accessing an element on a linked list and changes on the list depending on the self-organization technique applied: (a) move-to-front method, (b) transpose method, (c) count method, and (d) ordering method, in particular, alphabetical ordering, which leads to no change. In the case when the desired element is not in the list, (e) the first three methods add a new node with this element at the end of the list and (f) the ordering method maintains an order on the list.



ADT List with Insertion Influenced by Search: self-organizing list implementation

Element Searched For	Plain	Move-to- Front	Transpose	Count	Ordering
A:	A	A	A	A	A
C:	A C	A C	A C	A C	A C
B:	A C B	A C B	A C B	A C B	A B C
C:	A C B	C A B	C A B	C A B	A B C
D:	A C B D	C A B D	C A B D	C A B D	A B C D
A:	A C B D	A C B D	A C B D	C A B D	A B C D
D:	A C B D	D A C B	A C D B	D C A B	A B C D
A:	A C B D	A D C B	A C D B	A D C B	A B C D
C:	A C B D	C A D B	C A D B	C A D B	A B C D
A:	A C B D	A C D B	A C D B	A C D B	A B C D
C:	A C B D	C A D B	C A D B	A C D B	A B C D
C:	A C B D	C A D B	C A D B	C A D B	A B C D
E:	A C B D E	C A D B E	C A D B E	C A D B E	A B C D E
E:	A C B D E	E C A D B	C A D E B	C A E D B	A B C D E

Implementation of ADT List?

createList()

// Creates an empty list.

destroyList()

// Destroys a list.

listIsEmpty()

// Determines whether the list is empty.

listLength()

// Returns the number of items in list.

listInsert(newPosition, newItem, success)

// Inserts newItem at position newPosition of a list, if $1 \leq \text{newPosition} \leq \text{listLength()} + 1$

// if $\text{newPosition} \leq \text{listLength()}$, the items are shifted as follows

// the item at newPosition becomes the item at newPosition + 1, the item

// at newPosition + 2, and so on. Success indicates whether the insertion was successful.

listDelete(pos, success)

// Deletes the item at position pos of a list, if $1 \leq \text{pos} \leq \text{listLength()}$. If $\text{pos} < \text{lengthList()}$, the items are

// shifted as follows: the item at pos+1 becomes the at pos, the item at pos+1 becomes the item at pos+1, and


so // on. Success indicates whether the deletion was successful.

listRetrieve(pos, dataItem, success)

// sets dataItem to the item at position pos of a list, if $1 \leq \text{pos} \leq \text{listLength()}$. The list is left unchanged by

// this operation. Success indicates whether the retrieval was successful.

This variant is *not*
Discussed in Drozdek



Implementation of ADT List?

- Will be assignment:
 - Efficient array implementation
 - And another implementation with pointers.

Study Chapter 3 pages 76-106